4diac Bibliotheken: FB- & Adapter-Referenz (4.1)

Franz Höpfinger

Inhaltsverzeichnis

1 4diac Bibliotheken: FB- & Adapter-Referenz . . . 5

2 📚 Bibliotheken . . . 7

2.1 External Libraries . . . 7

2.1.1 BlinkMarine . . . 7
2.1.2 DataPanel . . . 13
2.1.3 Funk . . . 67
2.1.4 MyLib (test_AX) . . . 73
2.1.5 MyLib_AX_Option . . . 571
2.1.6 MyLib (test_B) . . . 579
2.1.7 MyLib_OPC . . . 740
2.1.8 OSCAT . . . 744
2.1.9 Option . . . 811
2.1.10 SafeArithmetic . . . 840
2.1.11 adapter . . . 852
2.1.12 iec61131-3-bool . . . 4569
2.1.13 isobus . . . 4674
2.1.14 logiBUS . . . 5119

2.2 Standard Libraries . . . 5825

2.2.1 convert . . . 5825
2.2.2 core . . . 5884
2.2.3 events . . . 5885

2.3 . . . 5997

2.4 . . . 6003

2.4.1 iec61131-3 . . . 6011
2.4.2 🔌 io . . . 6526
2.4.3 net . . . 6594
2.4.4 powerlink . . . 6728
2.4.5 reconfiguration . . . 6743
2.4.6 rtevents . . . 6757
2.4.7 signalprocessing . . . 6781
2.4.8 system . . . 6792
2.4.9 utils . . . 6795

2.5 ⚖️ Vergleich mit anderen Normen . . . 6878

2.5.1 IEC 61131-3 (Alte Norm) . . . 6878

3 static . . . 6883

3.1 Fonts . . . 6883

3.1.1 TGL Normschrift – _std Font Variants . . . 6883

1. 4diac Bibliotheken: FB- & Adapter-Referenz

Dieses Wiki (**4.1**) enthält die automatisch generierte Function-Block- und Adapter-Referenz für Eclipse 4diac (Standard- und External-Libraries), ausgelagert aus dem Haupt-Wiki Visuelle Programmiersprachen: Low-Code (4).

👉 Zur Bibliotheksübersicht

- 📄 PDF-Handbuch herunterladen

2. 📚 Bibliotheken

2.1 External Libraries

2.1.1 BlinkMarine

2.1.1.1 🔌 io

📺 Video

- Automation der Zukunft
- Die große Migration
- Digital decisions
- Ideale Dioden
- The Dual Wi Fi Solution

🎧 Podcast

- 4diac IDE: Wie der IEC 61499 Standard die Industrieautomatisierung revolutioniert
- 4diac-Präsentation: Zielgruppen, Struktur und Alleinstellungsmerkmal Schärfen
- Den Software-Drachen zähmen: Industrielle Automatisierung und die Zukunft der Produktion
- Eclipse 4diac: Innovationsmotor Forschung vs. Nutzerbedürfnisse – Was treibt die Entwicklung wirklich voran?
- Software-Drachen zähmen: Warum IEC 61499 die Industrie revolutionieren könnte

---

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 Diode & Halbleitergrundlagen auf ms-muc-docs.de
- 🌐 Smart Switch vs. Relais auf ms-muc-docs.de

DI

📺 Video

- Decoding the datasheet
- Die große Migration
- Die Kunst des Lötens
- Die Welt der Normung
- Digital decisions

🎧 Podcast

- 4diac IDE: Dein „Hello World“ der Automatisierung – Das Blinking Tutorial Lokal
- 4diac IDE: Dein Open-Source-Werkzeugkasten für verteilte Industrieautomatisierung nach IEC 61499
- 4diac IDE: Wie der IEC 61499 Standard die Industrieautomatisierung revolutioniert
- 4diac-Präsentation: Zielgruppen, Struktur und Alleinstellungsmerkmal Schärfen
- Den Software-Drachen zähmen: Industrielle Automatisierung und die Zukunft der Produktion

---

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

PKP-2200-LI

PKP_2200

• – ◦ • – ◦ • – ◦ •

Einleitung

PKP_2200 ist eine globale Konstante für die Blink Marine PowerKey-Schnittstelle. Diese Konstante dient als Standardwert für die Konfiguration von Digital-Input-Funktionalitäten im BlinkMarine-System.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

Keine Adapter vorhanden

Funktionsweise

PKP_2200 definiert eine globale Konstante vom Typ USINT (Unsigned Short Integer) mit dem hexadezimalen Wert 16#21 (dezimal 33). Diese Konstante wird als Standardwert für die PowerKey-Konfiguration verwendet.

Technische Besonderheiten

- **Datentyp**: USINT (8-Bit vorzeichenloser Integer)
- **Initialwert**: USINT#16#21 (hexadezimal 21, dezimal 33)
- **Paketzuordnung**: BlinkMarine::io::DI

Zustandsübersicht

Als globale Konstante besitzt PKP_2200 keine Zustandsübergänge und behält während der gesamten Laufzeit den fest definierten Wert.

Anwendungsszenarien

- Standardkonfiguration für Blink Marine PowerKey-Interfaces
- Verwendung als Referenzwert in Digital-Input-Funktionsblöcken
- Konfiguration von I/O-Parametern in maritimen Steuerungssystemen

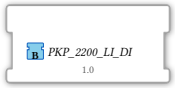
⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu Funktionsblöcken mit dynamischem Verhalten handelt es sich bei PKP_2200 um eine statische Konstante ohne Verhaltenslogik. Sie dient ausschließlich der Bereitstellung eines festen Konfigurationswerts.

Fazit

PKP_2200 stellt einen essentiellen Grundbaustein für die Konfiguration von Digital-Input-Schnittstellen im BlinkMarine-System dar. Durch die Verwendung als globale Konstante ermöglicht sie eine zentrale und konsistente Konfiguration über verschiedene Funktionsblöcke hinweg.

PKP_2200_LI_DI



• – ◦ • – ◦ • – ◦ •

Einleitung

Die PKP_2200_LI_DI ist eine globale Konstantendefinition für Blink Marine PowerKey Digital Inputs. Diese Konstante definiert die Pin-Zuordnungen für digitale Eingänge eines PowerKey-Systems und stellt vordefinierte Konfigurationen für verschiedene Tasten sowie einen ungültigen Zustand bereit.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden - handelt sich um eine globale Konstantendefinition

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden - handelt sich um eine globale Konstantendefinition

Daten-Eingänge

Keine Daten-Eingänge vorhanden - handelt sich um eine globale Konstantendefinition

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden - handelt sich um eine globale Konstantendefinition

Adapter

Keine Adapter vorhanden - handelt sich um eine globale Konstantendefinition

Funktionsweise

Die PKP_2200_LI_DI definiert globale Konstanten vom Typ `PKP_2200_LI_DI_S`, die die Pin-Nummern für digitale Eingänge eines Blink Marine PowerKey Systems festlegen. Jede Konstante repräsentiert eine spezifische Tastenkonfiguration mit zugeordneten Hardware-Pins.

Technische Besonderheiten

- **Datentyp**: Alle Konstanten verwenden den Strukturtyp `PKP_2200_LI_DI_S`
- **Pin-Zuordnungen**:
  - DigitalInput_Key_1: Pin 1
  - DigitalInput_Key_2: Pin 2
  - DigitalInput_Key_3: Pin 3
  - DigitalInput_Key_4: Pin 4
  - Invalid: Pin 255 (ungültiger Zustand)
- **Package-Zugehörigkeit**: BlinkMarine::io::DI

Zustandsübersicht

Keine Zustandsmaschine vorhanden - handelt sich um eine statische Konstantendefinition

Anwendungsszenarien

- Konfiguration von digitalen Eingängen in Marine-Steuerungssystemen
- Hardware-Abstraktion für PowerKey-Tastenbelegungen
- Vereinfachung der Pin-Zuordnung in IEC 61499-Applikationen
- Fehlerbehandlung durch definierten ungültigen Zustand

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu Funktionsblöcken mit aktiver Logik handelt es sich hier um eine reine Konstantendefinition, die keine aktive Verarbeitung durchführt. Sie dient ausschließlich der zentralen Verwaltung von Hardware-Konfigurationen.

Fazit

Die PKP_2200_LI_DI bietet eine standardisierte und wartbare Lösung für die Pin-Konfiguration digitaler Eingänge in Blink Marine PowerKey Systemen. Durch die zentrale Definition der Hardware-Zuordnungen wird die Konsistenz über verschiedene Applikationen hinweg gewährleistet und die Wiederverwendbarkeit des Codes verbessert.

PKP_2200_LI_DI_S

Blink Marine PowerKey DI Struct

Structure

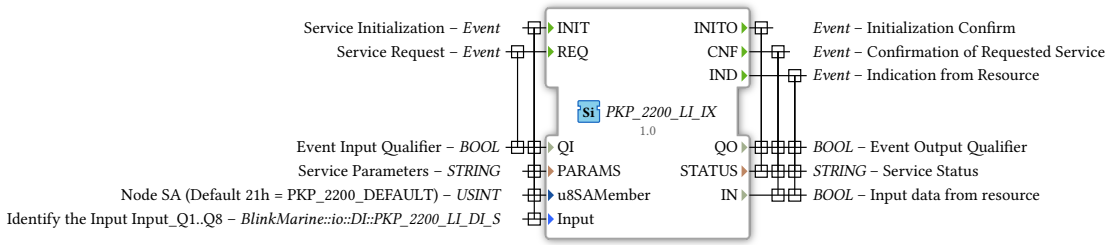
Name	Type	Initial Value
Pin	USINT	255

PKP_2200_LI_IX

• – ○ • – ○ • – ○ •

Einleitung

Der PKP_2200_LI_IX ist ein Service-Interface-Funktionsblock für boolesche Eingangsdaten. Dieser Baustein dient als Schnittstelle für digitale Eingänge und ermöglicht die Kommunikation mit entsprechenden Hardware-Ressourcen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung mit Parametern QI, PARAMS, u8SAMember und Input
- **REQ**: Service-Anfrage mit Parameter QI

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung mit Parametern QO und STATUS
- **CNF**: Bestätigung der angeforderten Service-Operation mit Parametern QO, STATUS und IN
- **IND**: Indikation von der Ressource mit Parametern QO, STATUS und IN

Daten-Eingänge

- **QI**: Event-Input-Qualifier (BOOL)
- **PARAMS**: Service-Parameter (STRING)
- **u8SAMember**: Node SA (USINT) - Standardwert: PKP_2200::PKP_2200_DEFAULT (21h)
- **Input**: Identifiziert den Eingang Input_Q1..Q8 (Typ: BlinkMarine::io::DI::PKP_2200_LI_DI_S) - Initialwert: Invalid

Daten-Ausgänge

- **QO**: Event-Output-Qualifier (BOOL)
- **STATUS**: Service-Status (STRING)
- **IN**: Eingangsdaten von der Ressource (BOOL)

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Funktionsblock initialisiert und verwaltet die Kommunikation mit digitalen Eingangsressourcen. Über den INIT-Event wird der Service mit den entsprechenden Parametern initialisiert. Der REQ-Event ermöglicht das Anfordern von Service-Operationen. Die Ausgangs-Events INITO, CNF und IND liefern Bestätigungen und Statusinformationen zurück.

Technische Besonderheiten

- Verwendet spezifische PKP_2200-Kommunikationsparameter
- Unterstützt bis zu 8 digitale Eingänge (Q1-Q8)
- Enthält Standardwerte für SA-Member (21h = PKP_2200_DEFAULT)
- Implementiert eine Invalid-Initialisierung für Eingänge

Zustandsübergänge

Der Baustein durchläuft folgende Zustände:

1. **Initialisierungsphase**: INIT-Event verarbeitet, INITO-Antwort generiert
2. **Service-Betrieb**: REQ-Event verarbeitet, CNF-Antwort generiert
3. **Ressourcen-Indikation**: Automatische IND-Ereignisse bei Statusänderungen

Anwendungsszenarien

- Anbindung von digitalen Eingangssensoren in Automatisierungssystemen
- Integration in Marine-Anwendungen (BlinkMarine-Umgebung)
- Kommunikation mit PKP_2200-kompatibler Hardware
- Überwachung von Schaltzuständen und Signalpegeln

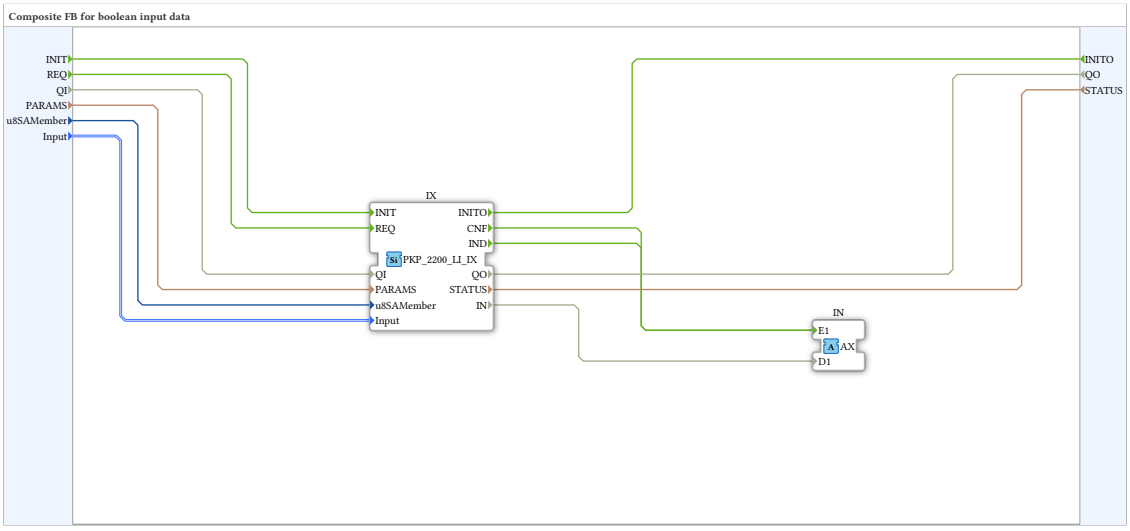
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Eingangsbausteinen bietet PKP_2200_LI_IX erweiterte Service-Funktionalitäten mit Statusrückmeldungen und Parametrisierungsmöglichkeiten. Die spezifische PKP_2200-Integration unterscheidet ihn von generischen IO-Bausteinen.

Fazit

Der PKP_2200_LI_IX ist ein spezialisierter Service-Interface-Baustein für die zuverlässige Anbindung digitaler Eingänge in industriellen Automatisierungssystemen, insbesondere in Marine-Umgebungen. Seine erweiterte Funktionalität mit Statusrückmeldungen und Parametrisierungsoptionen macht ihn zu einer robusten Lösung für anspruchsvolle IO-Anwendungen.

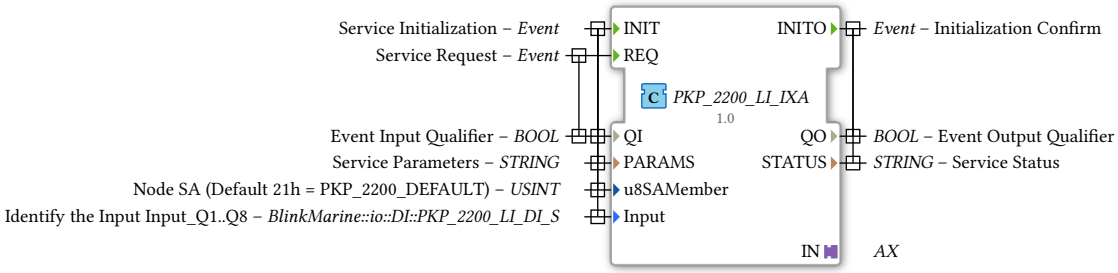
PKP_2200_LI_IXA



• – ○ • – ○ • – ○ •

Einleitung

Der PKP_2200_LI_IXA ist ein zusammengesetzter Funktionsblock zur Verarbeitung von booleschen Eingangsdaten. Er dient als Schnittstelle für digitale Eingänge und bietet standardisierte Initialisierungs- und Abfragefunktionen für Eingabesignale.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung (mit QI, PARAMS, u8SAMember, Input)
- **REQ**: Service-Anfrage (mit QI)

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung (mit QO, STATUS)

Daten-Eingänge

- **QI**: Ereignis-Eingangsqualifizierer (BOOL)
- **PARAMS**: Service-Parameter (STRING)
- **u8SAMember**: Knoten-SA (USINT) - Standardwert: PKP_2200::PKP_2200_DEFAULT
- **Input**: Eingangsidentifikation (BlinkMarine::io::DI::PKP_2200_LI_DI_S) - Standardwert: Invalid

Daten-Ausgänge

- **QO**: Ereignis-Ausgangsqualifizierer (BOOL)
- **STATUS**: Service-Status (STRING)

Adapter

- **IN**: Unidirektionaler Adapter vom Typ AX

Funktionsweise

Der Funktionsblock initialisiert sich über das INIT-Ereignis und bestätigt die erfolgreiche Initialisierung mit INITO. Über das REQ-Ereignis können Service-Anfragen gestellt werden. Die Eingabedaten werden über den Adapter IN verarbeitet und die Ergebnisse über die entsprechenden Ausgänge bereitgestellt.

Technische Besonderheiten

- Verwendet standardisierte Initialisierungsprozeduren nach EInit-Typ
- Unterstützt Konfiguration über SA-Member (Standard: 21h)
- Bietet Fehlerbehandlung über STATUS-Ausgabe
- Implementiert qualifizierte Ereignisverarbeitung

Zustandsübersicht

Der Funktionsblock durchläuft folgende Zustände:

1. **Nicht initialisiert**: Vor INIT-Ereignis
2. **Initialisierung**: Während INIT-Verarbeitung
3. **Bereit**: Nach erfolgreicher Initialisierung (INITO)
4. **Service-Anfrage**: Während REQ-Verarbeitung

Anwendungsszenarien

- Digitale Eingangssignalverarbeitung in Automatisierungssystemen
- IO-Kommunikation mit PKP-2200-kompatiblen Geräten
- Integration in verteilte Steuerungssysteme

- Boolesche Signalverarbeitung mit Qualitätsüberwachung

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Eingangsblöcken bietet PKP_2200_LI_IXA erweiterte Funktionalität durch:

- Parametrierbare SA-Member-Konfiguration
- Detaillierte Statusrückmeldungen
- Standardisierte Initialisierungsprozeduren
- Adapter-basierte Schnittstelle

Fazit

Der PKP_2200_LI_IXA ist ein robuster und flexibler Funktionsblock für die Verarbeitung digitaler Eingangssignale mit erweiterten Konfigurations- und Diagnosemöglichkeiten. Seine standardisierte Schnittstelle ermöglicht eine einfache Integration in komplexe Automatisierungssysteme.

2.1.2 DataPanel

2.1.2.1 Status

DataPanel_AMP_FEEDBACK_SAE_J1939

Amperage Feedback of 1 DataPanel Node

Interface

Event Inputs

Name	Comment	With
INIT	Service Initialization	QI, u8SAMember

Event Outputs

Name	Comment	With
INITO	Initialization Confirm	QO, STATUS
IND	Indication from Resource	QO, STATUS, Current_1A, Current_1B, Current_2A, Current_2B, Current_3A, Current_3B, Current_4A, Current_4B, Current_5A, Current_5B, Current_6A, Current_6B, Current_7A, Current_7B, Current_8A, Current_8B

Input Vars

Name	Type	Initial Value	Comment
QI	BOOL		Event Input Qualifier
u8SAMember	USINT	MI::MI_00	Node SA 224..239

Output Vars

Name	Type	Comment
QO	BOOL	Event Output Qualifier
STATUS	STRING	Service Status
Current_1A	UINT	Current in mA or 0.1A depending on module
Current_1B	UINT	
Current_2A	UINT	

Current_2B	UINT	
Current_3A	UINT	
Current_3B	UINT	
Current_4A	UINT	
Current_4B	UINT	
Current_5A	UINT	
Current_5B	UINT	
Current_6A	UINT	
Current_6B	UINT	
Current_7A	UINT	
Current_7B	UINT	
Current_8A	UINT	
Current_8B	UINT	

DataPanel_FREQ_SAE_J1939
Frequency/Counter Inputs 7A-8A Status

Interface

Event Inputs

Name	Comment	With
INIT	Service Initialization	QI, u8SAMember

Event Outputs

Name	Comment	With
INITO	Initialization Confirm	QO, STATUS
IND	Indication from Resource	QO, STATUS, Hertz_Count_7A, DutyCycle_SP_7A, Hertz_Count_8A, DutyCycle_SP_8A

Input Vars

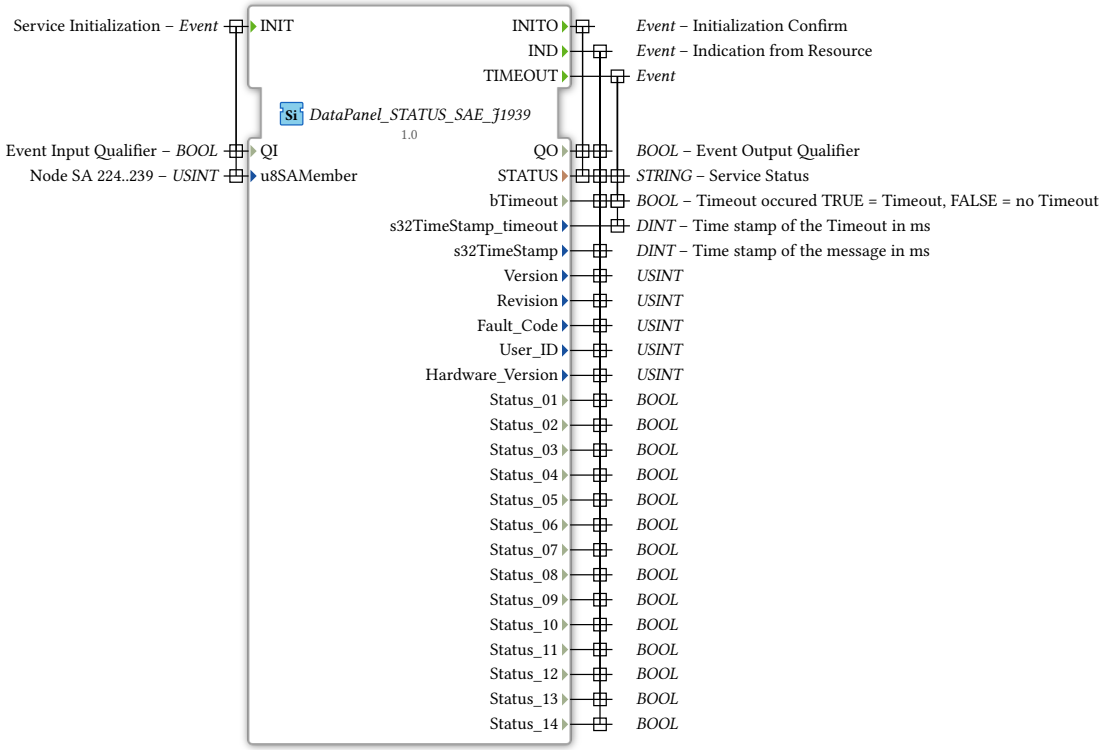
Name	Type	Initial Value	Comment
QI	BOOL		Event Input Qualifier
u8SAMember	USINT	MI::MI_00	Node SA 224..239

Output Vars

Name	Type	Comment
QO	BOOL	Event Output Qualifier
STATUS	STRING	Service Status
Hertz_Count_7A	UINT	Frequency (Hz) or Count for Input 7A
DutyCycle_SP_7A	UINT	Duty Cycle or Setpoint for Input 7A
Hertz_Count_8A	UINT	Frequency (Hz) or Count for Input 8A

DutyCycle_SP_8A	UINT	Duty Cycle or Setpoint for Input 8A

DataPanel_STATUS_SAE_J1939



Status Output of 1 DataPanel Node

Interface

Event Inputs

Name	Comment	With
INIT	Service Initialization	QI, u8SAMember

Event Outputs

Name	Comment	With
INITO	Initialization Confirm	QO, STATUS
IND	Indication from Resource	QO, s32TimeStamp, STATUS, bTimeout, Version, Revision, Fault_Code, User_ID, Hardware_Version, Status_01, Status_02, Status_03, Status_04, Status_05, Status_06, Status_07, Status_08, Status_09, Status_10, Status_11, Status_12, Status_13, Status_14
TIMEOUT		bTimeout, s32TimeStamp_timeout, STATUS, bTimeout

Input Vars

Name	Type	Initial Value	Comment
QI	BOOL		Event Input Qualifier
u8SAMember	USINT	MI::MI_00	Node SA 224..239

Output Vars

Name	Type	Initial Value	Comment
QO	BOOL		Event Output Qualifier
STATUS	STRING		Service Status
bTimeout	BOOL	FALSE	Timeout occurred TRUE = Timeout, FALSE = no Timeout
s32TimeStamp_timeout	DINT		Time stamp of the Timeout in ms
s32TimeStamp	DINT		Time stamp of the message in ms
Version	USINT		
Revision	USINT		
Fault_Code	USINT		
User_ID	USINT		
Hardware_Version	USINT		
Status_01	BOOL		
Status_02	BOOL		
Status_03	BOOL		
Status_04	BOOL		
Status_05	BOOL		
Status_06	BOOL		
Status_07	BOOL		
Status_08	BOOL		
Status_09	BOOL		
Status_10	BOOL		
Status_11	BOOL		
Status_12	BOOL		
Status_13	BOOL		
Status_14	BOOL		

2.1.2.2 🔌 io

📺 Video

- Automation der Zukunft
- Die große Migration
- Digital decisions
- Ideale Dioden
- The Dual Wi Fi Solution

🎧 Podcast

- 4diac IDE: Wie der IEC 61499 Standard die Industrieautomatisierung revolutioniert
- 4diac-Präsentation: Zielgruppen, Struktur und Alleinstellungsmerkmal Schärfen
- Den Software-Drachen zähmen: Industrielle Automatisierung und die Zukunft der Produktion
- Eclipse 4diac: Innovationsmotor Forschung vs. Nutzerbedürfnisse – Was treibt die Entwicklung wirklich voran?

- Software-Drachen zähmen: Warum IEC 61499 die Industrie revolutionieren könnte

---

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 Diode & Halbleitergrundlagen auf ms-muc-docs.de
- 🌐 Smart Switch vs. Relais auf ms-muc-docs.de

Modules with low IO density

DQ

DataPanel_LO_DO

• – ○ • – ○ • – ○ •

Einleitung
DataPanel_LO_DO ist eine globale Konstantendefinition für DataPanel-Module mit niedriger I/O-Dichte für digitale Ausgänge (DO). Die Konstante definiert eine strukturierte Zuordnung von Pin-Nummern zu spezifischen digitalen Ausgangskanälen und stellt einen ungültigen Zustand für Fehlerbehandlung bereit.

Schnittstellenstruktur

Ereignis-Eingänge
Nicht vorhanden - Es handelt sich um eine globale Konstantendefinition

Ereignis-Ausgänge
Nicht vorhanden - Es handelt sich um eine globale Konstantendefinition

Daten-Eingänge
Nicht vorhanden - Es handelt sich um eine globale Konstantendefinition

Daten-Ausgänge
Nicht vorhanden - Es handelt sich um eine globale Konstantendefinition

Adapter
Nicht vorhanden - Es handelt sich um eine globale Konstantendefinition

Funktionsweise
Die globale Konstante DataPanel_LO_DO definiert eine Sammlung von strukturierten Konstanten vom Typ `DataPanel_LO_DO_S`, die digitale Ausgangskanäle eines DataPanel-Moduls mit niedriger I/O-Dichte repräsentieren. Jede Konstante enthält eine Pin-Zuordnung für spezifische Ausgangskanäle.

Technische Besonderheiten
- **Datentyp**: Verwendet den strukturierten Datentyp `DataPanel_LO_DO_S`
- **Pin-Zuordnung**: Definiert 8 digitale Ausgangskanäle (1A-4B) mit fortlaufenden Pin-Nummern 1-8
- **Fehlerbehandlung**: Enthält eine spezielle `Invalid`-Konstante mit Pin-Wert 255 für ungültige Zustände
- **Modulare Struktur**: Kanalbenennung folgt dem Muster „DigitalOutput_XY“ für klare Identifikation

Zustandsübersicht
Nicht zutreffend - Es handelt sich um eine statische Konstantendefinition ohne Zustandsautomaten

Anwendungsszenarien
- Konfiguration von DataPanel-Modulen mit niedriger I/O-Dichte
- Vereinfachte Pin-Zuordnung in Steuerungsprogrammen

- Typsichere Verwendung von digitalen Ausgangskanälen
- Fehlerbehandlung durch definierte ungültige Zustände

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen I/O-Konstantendefinitionen bietet DataPanel_LO_DO:

- Spezifische Anpassung für Module mit niedriger I/O-Dichte
- Strukturierte Datentypen für bessere Typsicherheit
- Klare Benennungskonvention für bessere Lesbarkeit
- Integrierte Fehlerkonstante für robustere Programmierung

Fazit

DataPanel_LO_DO stellt eine effiziente und typsichere Lösung für die Konfiguration digitaler Ausgangskanäle in DataPanel-Modulen mit niedriger I/O-Dichte dar. Durch die strukturierte Definition und klare Pin-Zuordnung ermöglicht sie eine wartungsfreundliche und fehlerresistente Programmierung von Steuerungsanwendungen.

DataPanel_LO_DO_S

DataPanel Modules with low IO density DO Struct

Structure

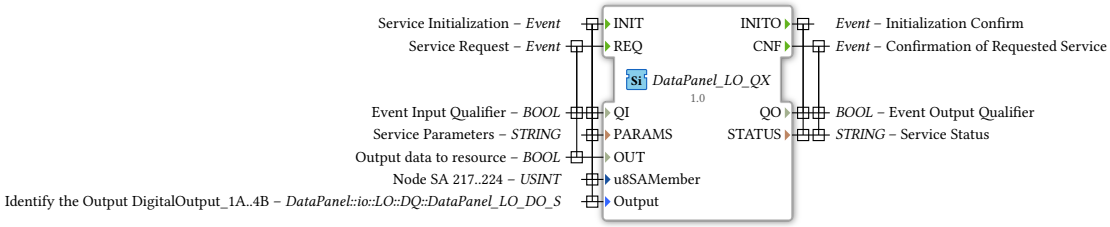
Name	Type	Initial Value
Pin	USINT	255

DataPanel_LO_QX

• – ◦ • – ◦ • – ◦ •

Einleitung

Der DataPanel_LO_QX ist ein Ausgabeservice-Schnittstellen-Funktionsblock für boolesche Ausgabedaten, der speziell für die Kommunikation mit Digitalausgängen in Automatisierungssystemen entwickelt wurde. Der Baustein ermöglicht die sichere und kontrollierte Ausgabe von binären Signalen an angeschlossene Hardware-Ressourcen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierungsereignis
- **REQ**: Service-Anfrageereignis

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
- **CNF**: Bestätigung der angeforderten Service-Operation

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangsqualifizierer
- **PARAMS** (STRING): Service-Parameter
- **OUT** (BOOL): Ausgabedaten zur Ressource
- **u8SAMember** (USINT): Knoten SA 217..224 (Initialwert: LO::LO_00)
- **Output** (DataPanel_LO_DO_S): Identifiziert den Digitalausgang 1A..4B (Initialwert: Invalid)

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifizierer
- **STATUS** (STRING): Service-Statusinformation

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Funktionsblock arbeitet nach einem typischen Service-Interface-Muster. Bei der Initialisierung (INIT-Event) werden die Service-Parameter und Hardware-Konfiguration übergeben. Die eigentliche Datenausgabe erfolgt über das REQ-Event, wobei der boolesche Wert am OUT-Eingang an die konfigurierte Hardware-Ressource ausgegeben wird. Jede Operation wird durch entsprechende Bestätigungsereignisse (INITO, CNF) quittiert.

Technische Besonderheiten

- Unterstützt spezifische Hardware-Adressierung über u8SAMember (SA-Knoten 217-224)
- Flexible Ausgangskonfiguration über Output-Parameter (DigitalOutput_1A bis 4B)
- Initialisierung mit Standardwerten für schnellen Einsatz
- String-basierte Statusrückmeldungen für detaillierte Fehlerdiagnose

Zustandsübersicht

Der Baustein verfügt über zwei Hauptzustände:

1. **Nicht initialisiert**: Wartet auf INIT-Event
2. **Initialisiert**: Bereit für REQ-Operationen

Nach erfolgreicher Initialisierung können kontinuierlich Ausgabeanforderungen verarbeitet werden.

Anwendungsszenarien

- Steuerung von Digitalausgängen in Prozessautomation
- Anbindung von Binärsignalen an Feldgeräte
- Integration in verteilte Steuerungssysteme
- Sicherheitskritische Schaltvorgänge

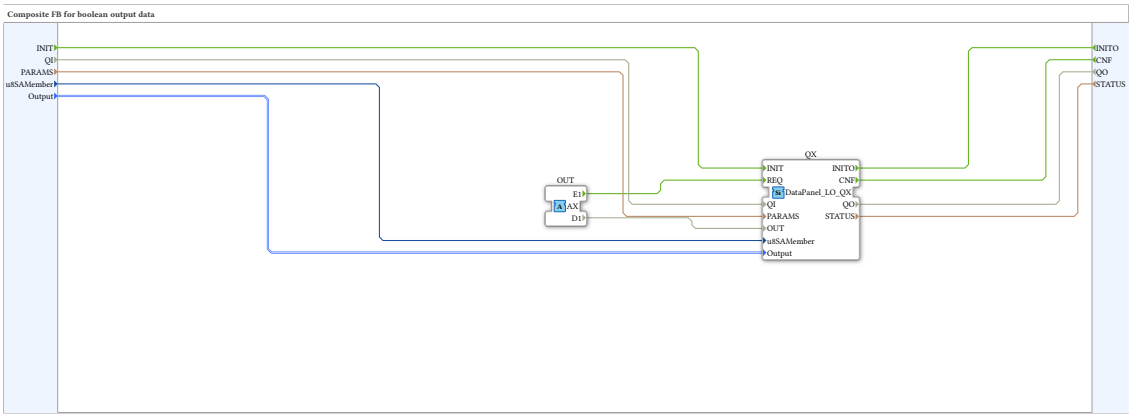
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen Digitalausgangsblöcken bietet DataPanel_LO_QX erweiterte Funktionalitäten wie parametrierbare Hardware-Adressierung, detaillierte Statusrückmeldungen und eine strukturierte Initialisierungsprozedur. Die spezifische SA-Member-Adressierung macht ihn besonders für komplexe verteilte Systeme geeignet.

Fazit

Der DataPanel_LO_QX ist ein robuster und flexibler Funktionsblock für die Ansteuerung von Digitalausgängen in industriellen Automatisierungssystemen. Seine strukturierte Schnittstelle und umfangreiche Konfigurationsmöglichkeiten machen ihn besonders für anspruchsvolle Anwendungen mit spezifischen Hardware-Anforderungen geeignet.

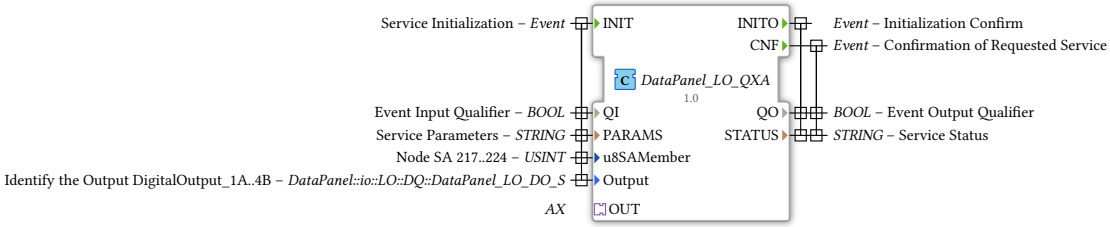
DataPanel_LO_QXA



• — ○ • — ○ • — ○ •

Einleitung

Der DataPanel_LO_QXA ist ein zusammengesetzter Funktionsblock für die Ausgabe von booleschen Daten in Automatisierungssystemen. Er dient als Schnittstelle für digitale Ausgabefunktionen und ermöglicht die Steuerung von Ausgängen über definierte Service-Parameter.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierungsereignis mit den zugehörigen Daten QI, PARAMS, u8SAMember und Output

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung mit QO und STATUS
- **CNF**: Bestätigung des angeforderten Services mit QO und STATUS

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangsqualifizierer
- **PARAMS** (STRING): Service-Parameter
- **u8SAMember** (USINT): Knoten-SA 217..224 mit Initialwert LO::LO_00
- **Output** (DataPanel_LO_DO_S): Identifiziert den Ausgang DigitalOutput_1A..4B mit Initialwert Invalid

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifizierer
- **STATUS** (STRING): Service-Status

Adapter

- **OUT**: Unidirektionaler Adapter vom Typ AX für die Ausgabesteuerung

Funktionsweise

Der Funktionsblock initialisiert sich über das INIT-Ereignis und gibt über INITO den Initialisierungsstatus zurück. Die eigentliche Ausgabesteuerung erfolgt über den Adapter OUT, der mit dem internen QX-Block verbunden ist. Der QX-Block verarbeitet die Eingabedaten und gibt über CNF den Status der Service-Anfrage zurück.

Technische Besonderheiten

- Verwendet spezifische SA-Member-Adressierung (217-224)
- Unterstützt verschiedene digitale Ausgänge (1A-4B)
- Implementiert Service-Parameter-Übergabe via STRING
- Enthält integrierte Fehlerbehandlung mit Invalid-Initialwert

Zustandsübersicht

1. **Nicht initialisiert**: Vor INIT-Ereignis
2. **Initialisierung läuft**: Während INIT-Verarbeitung
3. **Bereit**: Nach erfolgreicher INITO-Bestätigung
4. **Aktiv**: Bei Service-Anfragen über Adapter OUT

Anwendungsszenarien

- Steuerung digitaler Ausgänge in Automatisierungssystemen
- Integration in Prozessleitsysteme mit SA-Adressierung
- Anbindung an übergeordnete Steuerungssysteme via Service-Parameter
- Boolesche Signalausgabe in industriellen Anwendungen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Ausgabeblöcken bietet DataPanel_LO_QXA erweiterte Funktionalität durch:

- Parametrierbare Service-Schnittstelle
- SA-Member-basierte Adressierung
- Integrierte Statusrückmeldung
- Adapter-basierte Erweiterbarkeit

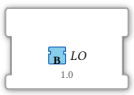
Fazit

Der DataPanel_LO_QXA ist ein leistungsfähiger zusammengesetzter Funktionsblock für die professionelle Steuerung digitaler Ausgänge in industriellen Automatisierungssystemen. Seine modulare Struktur und parametrierbare Schnittstelle machen ihn besonders geeignet für komplexe Anwendungen mit flexiblen Konfigurationsanforderungen.

const

LO

🎧 Podcast



- 4diac IDE: Dein „Hello World“ der Automatisierung – Das Blinking Tutorial Lokal
- Eclipse 4diac and FORTE: Unlocking Next-Gen Industrial Automation with Open-Source Power
- Feldgeräte: Augen, Ohren und Hände der industriellen Automatisierung – Warum diese unsichtbare Technologie so entscheidend ist
- Future Automation Unlocked: Powering „Plug & Produce“ and Protecting Your Privacy
- Industrial Revolution Reloaded: Unpacking Plug & Produce, Data Privacy, and ETFA 2025

📺 Video

- Der Katalog von 1863
- LogiBUS Projekt Update

Einleitung

LO ist eine globale Konstantendefinition für Module mit geringer I/O-Dichte. Diese Konstanten definieren spezifische Adressbereiche für I/O-Module und werden in der 4diac-IDE zur Konfiguration von Steuerungssystemen verwendet.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine Daten-Eingänge vorhanden

Daten-Ausgänge
Keine Daten-Ausgänge vorhanden

Adapter
Keine Adapter vorhanden

Funktionsweise
Bei LO handelt es sich um globale Konstanten, die als statische Werte definiert sind. Die Konstanten repräsentieren Adressbereiche für I/O-Module mit geringer Dichte und werden zur Kompilierzeit in die Anwendung integriert.

Technische Besonderheiten
- Alle Konstanten sind vom Typ USINT (Unsigned Short Integer)
- Wertebereich: 217 bis 224
- Statische Initialisierung zur Kompilierzeit
- Konstanten sind unveränderlich während der Laufzeit

Zustandsübersicht
Keine Zustandsübergänge vorhanden, da es sich um statische Konstanten handelt

Anwendungsszenarien
- Adressierung von I/O-Modulen mit geringer Dichte
- Konfiguration von Hardware-Schnittstellen
- Verwendung in Verbindung mit spezifischen I/O-Treibern
- Vereinfachung der Hardware-Konfiguration durch zentrale Konstantendefinition

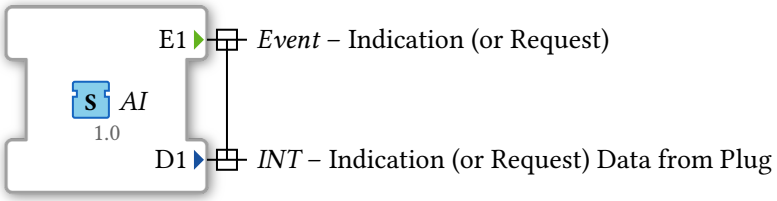
⚖️ Vergleich mit ähnlichen Bausteinen
Im Vergleich zu anderen globalen Konstanten-Bausteinen bietet LO spezifische Adressbereiche für Module mit geringer I/O-Dichte, während andere Bausteine möglicherweise andere Adressbereiche oder Konstantentypen definieren.

Fazit
Die LO-Konstanten bieten eine strukturierte und zentrale Möglichkeit zur Definition von Hardware-Adressen für I/O-Module mit geringer Dichte. Durch die Verwendung dieser globalen Konstanten wird die Wartbarkeit und Lesbarkeit von Steuerungsprogrammen verbessert.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Modules with medium IO density

AI



📺 Video

- AI: The Unknown Story
- The silent failure

🎧 Podcast

- Von klickenden Relais zum intelligenten Code: Wie Software die Industriesteuerung revolutionierte
- ISOBUS-Container: Dynamische Bedienfelder für klare Sicht und mehr Effizienz
- Analyse der Novellierung der Meisterprüfungsverordnung im Land- und Baumaschinenmechatroniker-Handwerk: Ein Detaillierter Vergleich der Verordnungen von 2024 und 2001
- Das Relais im Detail: Schaltverstärker, Schutz und die Geheimnisse von A1/A2, 85/86 und der Hysterese
- Das tödliche Dilemma des Relais: Warum Silberkontakte bei Kleinsignalen versagen und Gold bei Last schmilzt – Der Freibrenn-Effekt erklärt

---

DataPanel_MI_AI

DataPanel Modules with medium IO density AI constants

Global Constants

Name	Type	Initial Value
AnalogInput_1A	DataPanel::io::MI::AI::DataPanel_MI_AI_S	(Pin := 1)
AnalogInput_1B	DataPanel::io::MI::AI::DataPanel_MI_AI_S	(Pin := 2)
AnalogInput_2A	DataPanel::io::MI::AI::DataPanel_MI_AI_S	(Pin := 3)
AnalogInput_2B	DataPanel::io::MI::AI::DataPanel_MI_AI_S	(Pin := 4)
AnalogInput_3A	DataPanel::io::MI::AI::DataPanel_MI_AI_S	(Pin := 5)
AnalogInput_3B	DataPanel::io::MI::AI::DataPanel_MI_AI_S	(Pin := 6)
AnalogInput_4A	DataPanel::io::MI::AI::DataPanel_MI_AI_S	(Pin := 7)
AnalogInput_4B	DataPanel::io::MI::AI::DataPanel_MI_AI_S	(Pin := 8)
AnalogInput_5A	DataPanel::io::MI::AI::DataPanel_MI_AI_S	(Pin := 9)
AnalogInput_5B	DataPanel::io::MI::AI::DataPanel_MI_AI_S	(Pin := 10)
AnalogInput_6A	DataPanel::io::MI::AI::DataPanel_MI_AI_S	(Pin := 11)
AnalogInput_6B	DataPanel::io::MI::AI::DataPanel_MI_AI_S	(Pin := 12)
AnalogInput_7A	DataPanel::io::MI::AI::DataPanel_MI_AI_S	(Pin := 13)
AnalogInput_7B	DataPanel::io::MI::AI::DataPanel_MI_AI_S	(Pin := 14)
AnalogInput_8A	DataPanel::io::MI::AI::DataPanel_MI_AI_S	(Pin := 15)
AnalogInput_8B	DataPanel::io::MI::AI::DataPanel_MI_AI_S	(Pin := 16)
Invalid	DataPanel::io::MI::AI::DataPanel_MI_AI_S	(Pin := 255)

DataPanel_MI_AI_S
DataPanel Modules with medium IO density AI Struct

Structure

Name	Type	Initial Value
Pin	USINT	255

DataPanel_MI_IW
Input service interface function block for analog input data

Interface

Event Inputs

Name	Comment	With
INIT	Service Initialization	QI, PARAMS, u8SAMember, Input
REQ	Service Request	QI

Event Outputs

Name	Comment	With
INITO	Initialization Confirm	QO, STATUS
CNF	Confirmation of Requested Service	QO, STATUS, IN
IND	Indication from Resource	QO, STATUS, IN

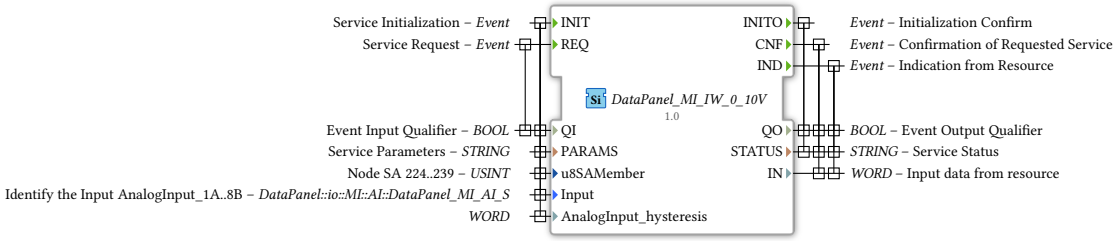
Input Vars

Name	Type	Initial Value	Comment
QI	BOOL		Event Input Qualifier
PARAMS	STRING		Service Parameters
u8SAMember	USINT	MI::MI_00	Node SA 224..239
Input	DataPanel::io::MI::AI::DataPanel_MI_AI_S	InvalId	Identify the Input AnalogInput_1A..8B

Output Vars

Name	Type	Comment
QO	BOOL	Event Output Qualifier
STATUS	STRING	Service Status
IN	WORD	Input data from resource

DataPanel_MI_IW_0_10V



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock **DataPanel_MI_IW_0_10V** ist ein service-orientierter Schnittstellenbaustein (SIFB) zur Erfassung eines analogen 0-10 V-Eingangssignals. Er ist als Bestandteil des **HR Agrartechnik DataPanel MI**-Systems konzipiert und ermöglicht die parametrierbare Initialisierung sowie den zyklischen oder ereignisgesteuerten Abruf von Messwerten.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Mitgeführte Daten	Beschreibung
`INIT`	`EInit`	`QI`, `PARAMS`, `u8SAMember`, `Input`, `AnalogInput_hysteresis`	Dienstinitialisierung: Konfiguration der Hardware-Anbindung
`REQ`	`Event`	`QI`	Dienstanforderung: Auslösen einer Messwertabfrage

Ereignis-Ausgänge

Ereignis	Typ	Mitgeführte Daten	Beschreibung
`INITO`	`EInit`	`QO`, `STATUS`	Bestätigung der Initialisierung
`CNF`	`Event`	`QO`, `STATUS`, `IN`	Bestätigung der Messwertabfrage
`IND`	`Event`	`QO`, `STATUS`, `IN`	Asynchrone Anzeige (z. B. spontane Messwertaktualisierung aus der Ressource)

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung
`QI`	`BOOL`	–	Eingangsqualifizierer (steuert die Ausführung)
`PARAMS`	`STRING`	–	Dienstparameter (z. B. Kommunikationskonfiguration)
`u8SAMember`	`USINT`	`MI::MI_00`	Knoten-SA-Adresse (gültiger Bereich 224…239)
`Input`	`DataPanel::io::MI::AI::DataPanel_MI_AI_S`	`AI_Invalid`	Auswahl des analogen Eingangs (z. B. `AnalogInput_1A` … `AnalogInput_8B`)
`AnalogInput_hysteresis`	`WORD`	–	Hysteresewert für die Signalglättung

Daten-Ausgänge

Name	Typ	Beschreibung
QO	BOOL	Ausgangsqualifizierer (zeigt gültige Verarbeitung an)
STATUS	STRING	Dienststatus (Fehler-/Erfolgsmeldung)
IN	WORD	Digitalisierter Analogwert (0…10 V, roher WORD-Wert)

Adapter

Keine Adapter vorhanden.

Funktionsweise

1. **Initialisierung (INIT)** Der Baustein wird mit den Parametern PARAMS, der Knotenadresse u8SAMember, dem ausgewählten Eingang Input und der Hysterese AnalogInput_hysteresis konfiguriert. Bei erfolgreicher Initialisierung wird der Ausgang INITO mit QO = TRUE und einem positiven STATUS gesendet.
2. **Messwertabfrage (REQ)** Durch Anlegen eines Ereignisses an REQ wird eine neue Messung angefordert. Das Ergebnis steht am Ausgang IN zur Verfügung, sobald das Ereignis CNF ausgelöst wird. Der Qualifizierer QO und der STATUS geben die Gültigkeit des Werts an.
3. **Asynchrone Anzeige (IND)** Der Baustein kann auch ohne explizite Anforderung von der Hardware ein Ereignis IND empfangen (z. B. bei einer spontanen Wertänderung oder einem periodischen Update). Auch hier werden IN, QO und STATUS aktualisiert.

Der digitalisierte Analogwert wird als 16-Bit-Wort (WORD) auf dem Ausgang IN bereitgestellt. Die Skalierung (z. B. 0 V ⟶ 0, 10 V ⟶ 65535) ist abhängig von der angeschlossenen Hardware und muss applikativ interpretiert werden.

Technische Besonderheiten

- **HR Agrartechnik-spezifisch**: Der FB ist für die DataPanel-MI-Familie vorgesehen und verwendet vordefinierte Konstanten aus den Paketen DataPanel::io::MI::const::MI und DataPanel::io::MI::AI::DataPanel_MI_AI.
- **Hysterese**: Mit dem Parameter AnalogInput_hysteresis kann eine Rauschunterdrückung oder Entprellung des Signals eingestellt werden.
- **Adressierung**: Über u8SAMember (USINT, Bereich 224–239) wird der Busknoten ausgewählt, Standard ist MI::MI_00.
- **Compiler-Integration**: Der Baustein ist als SIFB in das 4diac-IDE-Ökosystem eingebunden und besitzt einen eindeutigen Typ-Hash (eclipse4diac::core::TypeHash).

Zustandsübersicht

Da es sich um einen service-orientierten Baustein handelt, wird die interne Zustandsmaschine durch die Ereignisse gesteuert:

- **IDLE** – Nach erfolgreicher Initialisierung (oder nach Reset) wartet der Baustein auf REQ oder ein externes IND.
- **BUSY** – Nach einem REQ wird die Hardwareabfrage durchgeführt; während dieser Zeit ist kein weiterer REQ möglich. Nach Abschluss wird CNF gesendet und der Zustand geht zurück nach IDLE.
- **ERROR** – Tritt ein Fehler während der Initialisierung oder Messung auf, wird STATUS entsprechend gesetzt und QO = FALSE signalisiert.

Eine detaillierte Zustandsmaschine (ECC) liegt im XML-Modell nicht vor, das beschriebene Verhalten ist jedoch typisch für service-orientierte SIFBs.

Anwendungsszenarien

- **Agrartechnik**: Erfassung von Sensorsignalen (z. B. Druck, Temperatur, Füllstand) über 0-10 V-Schnittstellen an HR-Datenpanels.
- **Industrielle Steuerung**: Integration analoger Messwerte in eine SPS-Umgebung über 4diac-Anwendungen.
- **Parametrierbare Messstellen**: Mehrere Kanäle eines MI-Moduls können durch unterschiedliche Input-Werte abgefragt werden.

Vergleich mit ähnlichen Bausteinen

Baustein	Spannungsbereich	Plattform	Besonderheiten
`DataPanel_MI_IW_0_10V`	0-10 V	HR DataPanel MI	Hysterese, SA-Adressierung
`DataPanel_MI_IW_4_20mA`	4-20 mA	HR DataPanel MI	Analoge Stromeingänge
`GenericAnalogInput`	variabel	Standard IEC 61131	Allgemeiner Eingang, keine Hysterese

Der vorliegende Baustein ist speziell für die DataPanel-MI-Hardware optimiert und bietet eine enge Kopplung an die gerätespezifischen Parameter.

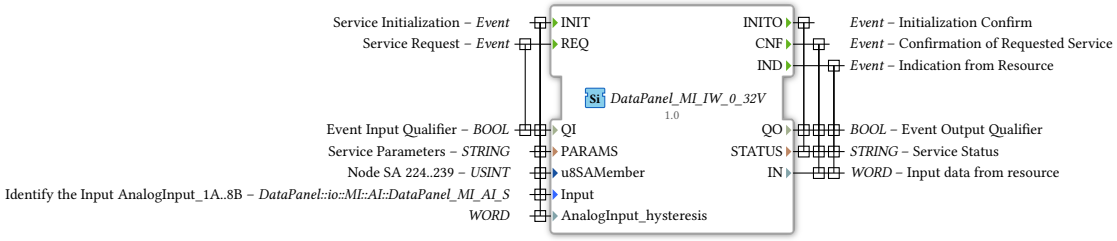
Fazit

Der **DataPanel_MI_IW_0_10V**-Funktionsblock stellt eine robuste und flexibel konfigurierbare Schnittstelle zur Erfassung analoger 0-10 V-Signale im HR Agrartechnik DataPanel-System dar. Durch die Möglichkeit der Kanalkonfiguration, Adressierung und Hysterese eignet er sich für präzise Messaufgaben in der landwirtschaftlichen Automatisierungstechnik.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

DataPanel_MI_IW_0_32V



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **DataPanel_MI_IW_0_32V** ist ein Service-Interface-Funktionsblock (SIFB) gemäß IEC 61499. Er dient dem Einlesen eines analogen Spannungssignals im Bereich 0–32 V über ein DataPanel-MI-Modul. Der Baustein übernimmt die Initialisierung des Kommunikationskanals, die Konfiguration eines Analog-Eingangs (z. B. AnalogInput_1A bis 8B) sowie die zyklische Abfrage oder den Empfang von Messwerten. Die Ausgabe erfolgt als **WORD**-Wert; der Status wird über **STATUS** und das Event **CNF** bzw. **IND** signalisiert.

Schnittstellenstruktur

Ereignis-Eingänge

Event	Kommentar	Mit (With)
INIT	Service-Initialisierung	QI, PARAMS, u8SAMember, Input, AnalogInput_hysteresis
REQ	Service-Anforderung (Messwertabruf)	QI

Ereignis-Ausgänge

Event	Kommentar	Mit (With)
INITO	Bestätigung der Initialisierung	QO, STATUS
CNF	Bestätigung der angeforderten Serviceleistung	QO, STATUS, IN

IND	Indikation eines neuen Messwerts von der Ressource	QO, STATUS, IN

Daten-Eingänge

Name	Typ	Initialwert	Kommentar
QI	BOOL	–	Ereignis-Eingangsqualifizierer
PARAMS	STRING	–	Service-Parameter (z. B. Adressen, Baudrate)
u8SAMember	USINT	`MI::MI_00`	Knoten-SA-Adresse (224..239)
Input	DataPanel::io::MI::AI::DataPanel_MI_AI_8	`Invalid`	Auswahl des analogen Eingangs (AnalogInput_1A … 8B)
AnalogInput_hysteresis	WORD	–	Hysterese für den Analogwert (Konfiguration)

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Ereignis-Ausgangsqualifizierer
STATUS	STRING	Dienststatus (z. B. Fehler-/Erfolgsmeldungen)
IN	WORD	Gemessener Analogwert (0..32 V skaliert als WORD)

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Baustein wird durch das **INIT**-Ereignis gestartet. Dabei werden die übergebenen Parameter (Knotenadresse, Auswahl des analogen Eingangs und Hysterese) an die Hardware weitergegeben. Nach erfolgreicher Initialisierung wird **INITO** mit `QO = TRUE` ausgelöst.

Ein **REQ**-Ereignis löst die Abfrage des aktuellen Analogwerts aus. Der Messwert wird über den Ausgang **IN** als **WORD** bereitgestellt und über das **CNF**-Ereignis quittiert. Die Ressource kann auch spontan einen neuen Wert senden, der dann über das **IND**-Ereignis signalisiert wird – ebenfalls mit aktuellem **IN** und **STATUS**.

STATUS liefert bei `"OK"` einen gültigen Wert; bei Fehlern (z. B. Kommunikationsfehler, ungültiger Kanal) wird eine Fehlermeldung ausgegeben und `QO` auf `FALSE` gesetzt.

Technische Besonderheiten

- **Spannungsbereich:** 0 V bis 32 V, dargestellt als WORD – die Skalierung erfolgt hardwareabhängig.
- **Hysterese-Konfiguration:** Der Parameter `AnalogInput_hysteresis` erlaubt das Einstellen einer Schwelle, um Rauschen oder kleine Spannungsschwankungen zu unterdrücken.
- **Initialisierung:** Der Eingang `u8SAMember` muss eine gültige Knotenadresse (224–239) enthalten; `Input` wählt den konkreten analogen Kanal aus. Voreingestellt ist `MI::MI_00` (Adresse des ersten MI-Moduls) und `Invalid`, d. h. vor dem ersten INIT muss ein gültiger Kanal gesetzt werden.
- **Service-Interface-Baustein:** Der FB kommuniziert direkt mit der Hardware und ist nicht durch eine ECC-Zustandsmaschine modelliert – die Ereignissteuerung erfolgt ereignisgesteuert nach dem Pattern von SIFBs.

Zustandsübersicht

Da es sich um einen Service-Interface-FB handelt, wird die interne Logik der Hardware überlassen. Aus Sicht des IEC-61499-Modells ergeben sich folgende typischen Phasen:

- **IDLE:** Nach dem Systemstart, bevor ein INIT ausgelöst wurde.

- **INITIATE:** Während der INIT-Bearbeitung (Hardware-Parametrierung).
- **OPERATIONAL:** Nach erfolgreichem INIT – REQ- und IND-Ereignisse werden verarbeitet.
- **ERROR:** Bei fehlerhafter Initialisierung oder Kommunikationsverlust – STATUS enthält Fehlertext.

Anwendungsszenarien

- **Spannungsüberwachung:** Erfassung von Sensorsignalen (z. B. Drucksensoren, Füllstandssensoren) mit einem Ausgangssignal von 0…10 V oder 0…32 V.
- **Analogwerterfassung in der Landtechnik:** Integration in ein DataPanel-Steuersystem zur Aufnahme von Analogwerten an verschiedenen Maschinenkomponenten.
- **Hysterese-gesteuerte Schwellwertauswertung:** Direkte Nutzung des IN-Wertes in einer nachgeschalteten Logik, die auf Über-/Unterschreitung von Schwellen reagiert.

Vergleich mit ähnlichen Bausteinen

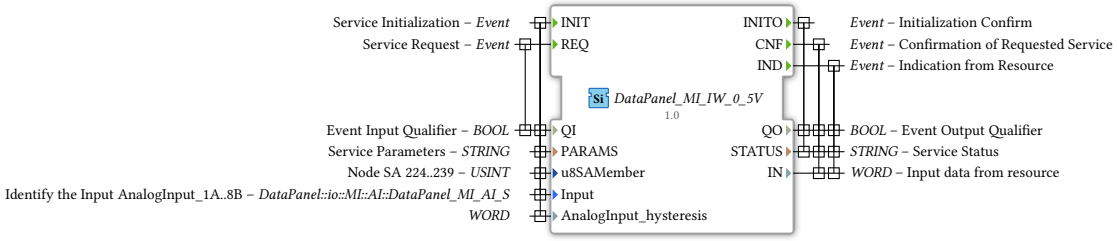
Im Gegensatz zu generischen analogen Eingangsbausteinen (z. B. `AI_BASIC`) ist der **DataPanel_MI_IW_0_32V** speziell für die DataPanel-MI-Hardware optimiert. Typische Unterschiede:

- **Parametrierung:** Spezifische Knotenadresse (`u8SAMember`) und Kanalauswahl (`Input`) statt generischer Konfigurationsstrings.
- **Hysterese:** Expliziter Eingang für die Hysterese – bei Standard-Bausteinen wird dies oft über Parametrierung des Kommunikationstreibers geregelt.
- **Spannungsbereich:** Fest auf 0–32 V ausgelegt; andere Bausteine bieten möglicherweise eine Konfiguration des Messbereichs.

Fazit

Der **DataPanel_MI_IW_0_32V** ist ein praktischer, für die DataPanel-Plattform maßgeschneiderter Funktionsblock zur Erfassung von Analogsignalen im Bereich 0–32 V. Durch die klare Schnittstelle (Initialisierung, Abruf, spontane Indikation) und die integrierte Hysterese eignet er sich hervorragend für den Einsatz in landtechnischen Steuerungen und anderen industriellen Applikationen mit DataPanel-Komponenten. Die einfache Bedienung über die Events INIT und REQ sowie die standardisierte Status-Rückmeldung ermöglichen eine schnelle Integration in IEC-61499-Projekte.

DataPanel_MI_IW_0_5V



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **DataPanel_MI_IW_0_5V** ist ein Service-Interface-Funktionsblock (SIFB) zur Erfassung analoger Eingangssignale im Spannungsbereich **0 … 5 V**. Er stellt die Schnittstelle zu einem analogen Eingangskanal der DataPanel-MI-IW-Reihe dar und ermöglicht die Initialisierung, die zyklische Abfrage sowie den asynchronen Empfang von Messwerten über ein Bussystem. Der Baustein ist als IEC 61499-konformer FB realisiert und verwendet systemspezifische Datentypen aus dem Package `DataPanel::io::MI::AI`.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung	Mit Variablen
`INIT`	EInit	Initialisiert die Service-Verbindung	`QI`, `PARAMS`, `u8SAMember`, `Input`, `AnalogInput_hysteresis`
`REQ`	Event	Fordert einen aktuellen Messwert an	`QI`

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung	Mit Variablen
INITO	EInit	Bestätigung der Initialisierung	QO, STATUS
CNF	Event	Bestätigung einer abgeschlossenen Anforderung	QO, STATUS, IN
IND	Event	Asynchrone Indikation eines Messwerts von der Ressource	QO, STATUS, IN

Daten-Eingänge

Variable	Typ	Beschreibung	Initialwert
QI	BOOL	Qualifizierer für Ereignis-Eingänge	–
PARAMS	STRING	Service-Parameter (z. B. Kommunikationsadresse)	–
u8SAMember	USINT	Knoten-SA-Adresse (224 … 239)	MI::MI_00
Input	DataPanel::io::MI::AI::DataPanel_MI_AI	Auswahl des analogen Eingangskanals (z. B. AnalogInput_1A … 8B)	Invalid
AnalogInput_hysteresis	WORD	Hysteresewert für die Signalauswertung	–

Daten-Ausgänge

Variable	Typ	Beschreibung
QO	BOOL	Ausgangsqualifizierer (zeigt gültigen Status an)
STATUS	STRING	Statusmeldung des Service (z. B. Fehlertext)
IN	WORD	Gemessener Analogwert (Rohwert) aus der Ressource

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Funktionsblock arbeitet nach dem **Service-Interface-Prinzip**: Er kommuniziert über Ereignisse mit einer darunterliegenden Hardware-Ressource (z. B. einem Bussystem oder I/O-Modul).

1. **Initialisierung (INIT)**
   Ein INIT-Ereignis (mit gesetztem QI) startet die Verbindung. Die Parameter PARAMS, u8SAMember, Input und AnalogInput_hysteresis legen die Zieladresse und die Kanalkonfiguration fest. Nach erfolgreicher Initialisierung wird INITO ausgelöst und QO auf TRUE sowie STATUS auf einen entsprechenden Text gesetzt.
2. **Messwertanforderung (REQ)**
   Sobald der FB initialisiert ist, kann über REQ ein aktueller Messwert angefordert werden. Der FB sendet die Anfrage an die Ressource und signalisiert nach Erhalt des Ergebnisses mit CNF. Der Rohwert wird in der Ausgangsvariable IN (Typ WORD) bereitgestellt.
3. **Asynchrone Indikation (IND)**
   Die Ressource kann eigenständig Messwertänderungen oder Alarme melden. Diese werden über das IND-Ereignis mit dem zugehörigen Wert in IN gemeldet.

Bei negativem QI oder Fehlersituationen werden QO auf FALSE und STATUS auf eine Fehlerbeschreibung gesetzt.

Technische Besonderheiten

- **Systemspezifische Typen**
  Der FB verwendet die benutzerdefinierten Strukturen `DataPanel_MI_AI_S` (Kanalauswahl) und die Konstanten `MI::MI_00` (Adressvorgabe). Diese sind im Package `DataPanel::io::MI::AI` definiert.
- **Hysterese**
  Über den Eingang `AnalogInput_hysteresis` kann eine Hysterese als `WORD`-Wert vorgegeben werden, um das Rauschen des analogen Signals zu unterdrücken.
- **Kanaladressierung**
  Die Auswahl des analogen Eingangskanals erfolgt über den `Input`-Parameter. Gültige Werte sind z. B. `AnalogInput_1A` bis `AnalogInput_8B`; der Initialwert `Invalid` muss vor der ersten Nutzung durch einen gültigen Kanal ersetzt werden.

Zustandsübersicht

Da die XML-Definition keine ECC (Execution Control Chart) enthält, ergibt sich die Zustandslogik aus dem typischen Verhalten eines SIFB. Eine abstrakte Zustandsmaschine lässt sich wie folgt beschreiben:

Zustand	Beschreibung	Ereignis	Aktion
IDLE	Warten auf Initialisierung	`INIT` (QI=TRUE)	Starte Verbindungsaufbau
INIT	Initialisierung läuft	–	Warte auf Bestätigung der Ressource
READY	Bereit für Anforderungen	`INITO`	Setze QO=TRUE
BUSY	Messwertanforderung läuft	`REQ`	Sende Anfrage an Ressource
DONE	Antwort empfangen	`CNF`	Lade `IN` und setze QO=TRUE
ERROR	Fehlerzustand	–	Setze QO=FALSE, STATUS=Fehlertext

Asynchrone `IND`-Ereignisse können in den Zuständen **READY** oder **BUSY** eintreten und führen zur sofortigen Bereitstellung des Werts.

Anwendungsszenarien

- **Landwirtschaftliche Sensorik** (z. B. Füllstandsensoren, Drucksensoren, Temperatursensoren mit 0-5 V Ausgang)
- **Datenerfassung in Stationärmotoren** oder **Fahrzeugsteuerungen** der DataPanel-Familie
- **Mehrkanal-Analogwerterfassung** durch parallele Instanzen des FB mit unterschiedlichen `Input`- und `u8SAMember`-Werten

Vergleich mit ähnlichen Bausteinen

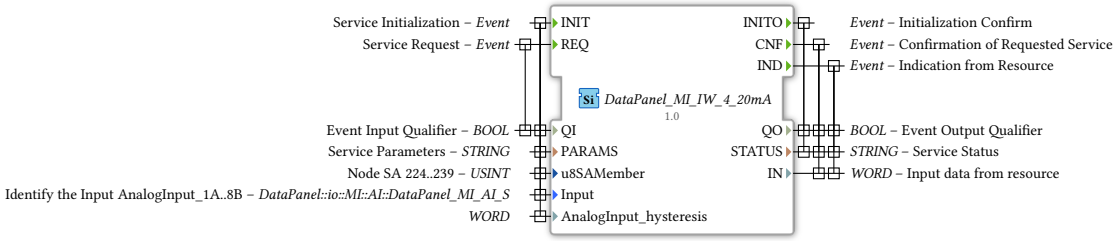
Merkmal	DataPanel_MI_IW_0_5V	Generischer Analogeingang (z. B. IEC 61499-Standard)
Spannungsbereich	0 – 5 V	Meist konfigurierbar (0-10 V, 4-20 mA u. a.)
Kanalauswahl	Spezifischer Typ `DataPanel_MI_AI_S`	Meist `INT`- oder `STRING`-Parameter
Hysterese	Separate Variable (`WORD`)	Oft nicht enthalten
Busanbindung	Proprietär (DataPanel-MI-IW)	Abstrakte Kommunikationsschnittstelle

Der FB ist stark auf die DataPanel-Hardware zugeschnitten und bietet daher weniger Flexibilität als ein generischer Analogeingang, dafür aber eine direkte, optimierte Anbindung.

Fazit

Der **DataPanel_MI_IW_0_5V** ist ein spezialisierter Service-Interface-Funktionsblock für die Erfassung von 0–5 V-Analogsignalen im DataPanel-Umfeld. Er kapselt die komplexe Buskommunikation und bietet eine einfache, ereignisgesteuerte Schnittstelle zur Anwendungslogik. Dank der integrierten Hysterese und der klaren Kanaladressierung eignet er sich besonders für robuste Messaufgaben in der landtechnischen Automatisierung.

DataPanel_MI_IW_4_20mA



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **DataPanel_MI_IW_4_20mA** ist ein Service-Interface-Funktionsblock (SIFB) zur Erfassung analoger Eingangsdaten im 4-20 mA Bereich. Er dient als Schnittstelle zwischen dem Automatisierungssystem und einem hardwarenahen analogen Eingabemodul. Der Baustein übernimmt die Initialisierung der Kommunikation (z. B. Bus-Anbindung), die zyklische Messwertanforderung sowie die Bereitstellung der gemessenen Rohdaten über Ausgangsereignisse.

• – ◦ • – ◦ • – ◦ •

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung	Mitgeführte Variablen
`INIT`	EInit	Initialisierung des Servicebausteins	QI, PARAMS, u8SAMember, Input, AnalogInput_hysteresis
`REQ`	Event	Anforderung einer Messwertaktualisierung	QI

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung	Mitgeführte Variablen
`INITO`	EInit	Bestätigung der erfolgreichen Initialisierung	QO, STATUS
`CNF`	Event	Bestätigung einer angeforderten Messung	QO, STATUS, IN
`IND`	Event	Asynchrone Anzeige eines eingehenden Messwerts	QO, STATUS, IN

Daten-Eingänge

Variable	Typ	Beschreibung	Initialwert
`QI`	BOOL	Qualifikator für den Ereigniseingang	–
`PARAMS`	STRING	Parameter für die Serviceinitialisierung	–
`u8SAMember`	USINT	Node-Adresse (224…239) des Slave-Geräts	`MI::MI_00` (224)
`Input`	*DataPanel::io::MI::AI::DataPanel_MI_AI_S*	Identifikation des analogen Eingangskanals (z. B. AnalogInput_1A..8B)	`Invalid`
`AnalogInput_hysteresis`	WORD	Hysteresewert für die Signalglättung	–

Daten-Ausgänge

Variable	Typ	Beschreibung
QO	BOOL	Qualifikator für den Ereignisausgang
STATUS	STRING	Statusmeldung (z. B. Fehler, Initialisierung)
IN	WORD	Gelesener Rohwert des analogen Eingangs

Adapter

Keine Adapter definiert.

• – ◦ • – ◦ • – ◦ •

Funktionsweise

Der Funktionsblock realisiert eine asynchrone Kommunikation mit einem Slave-Gerät (z. B. einem Analog-Eingangsmodul mit 4-20 mA Schnittstelle) über einen proprietären Bus.

1. **Initialisierung** (INIT): Die Parameter PARAMS (z. B. Baudrate, Protokoll-Einstellungen), die Knotenadresse (u8SAMember) und der konkrete analoge Eingangskanal (Input) werden gesetzt. Eine Hysterese (AnalogInput_hysteresis) kann zur Stabilisierung des Rohwerts angegeben werden. Nach erfolgreicher Verbindung wird INITO mit QO = TRUE gesendet.
2. **Messwertanforderung** (REQ): Der Baustein fordert den aktuellen Messwert des konfigurierten Kanals an. Die Antwort wird asynchron über den Ausgang CNF (bei erfolgreicher Anforderung) oder ggf. über IND (bei spontanen Wertänderungen oder zyklischen Meldungen des Slaves) geliefert. Der gelesene Wert erscheint in der Ausgangsvariablen IN als 16-Bit-Rohwert.
3. **Asynchrone Indikation** (IND): Falls das Slave-Gerät selbstständig (z. B. bei Überschreitung einer Schwelle) Daten sendet, wird IND ausgelöst. Dadurch können auch nicht angeforderte Messwerte erfasst werden.

Die Ausgänge QO und STATUS geben Aufschluss über den Erfolg der Operationen (z. B. Initialisierungsfehler, Kommunikationsfehler).

• – ◦ • – ◦ • – ◦ •

Technische Besonderheiten

- **Zielplattform**: Der Baustein ist für das „DataPanel"-System der HR Agrartechnik GmbH (Version 1.0, Jahr 2026) ausgelegt.
- **Knotenadressbereich**: Die Slave-Adressen u8SAMember sind auf den Bereich 224–239 beschränkt; der Initialwert MI::MI_00 entspricht der kleinsten Adresse (224).
- **Eingangskanal-Identifikation**: Der Datentyp DataPanel_MI_AI_S definiert mögliche Kanäle (AnalogInput_1A … 8B). Der Initialwert Invalid zeigt an, dass vor der ersten Initialisierung kein Kanal ausgewählt ist.
- **Hysterese**: Der AnalogInput_hysteresis-Wert wird als 16-Bit-Wort übergeben und wirkt als digitaler Filter zur Vermeidung von Rauschen oder Pendeln.
- **Typ-Hash**: Ein Attribut eclipse4diac::core::TypeHash ist enthalten, wird aber mit leerem String initialisiert – kann später zur Laufzeit gesetzt werden.

• – ◦ • – ◦ • – ◦ •

Zustandsübersicht

Da es sich um einen Service-Interface-FB handelt, ist das Verhalten durch die Ereignislogik und das zugrundeliegende Kommunikations-Protokoll bestimmt. Ein interner Zustandsautomat wird nicht in der XML-Definition abgebildet, typischerweise existieren jedoch folgende Phasen:

- **OFF / UNINITIALIZED**: Vor dem ersten INIT-Ereignis. Keine Kommunikation aktiv.
- **INIT**-Phase: Nach Empfang von INIT, bis INITO gesendet wird. Konfiguration des Slaves.
- **IDLE** (bereit): Nach erfolgreicher Initialisierung. Der FB kann REQ-Ereignisse annehmen.
- **BUSY** (Anforderung läuft): Nach REQ bis zum Eintreffen der Antwort (gefolgt von CNF).

- **INDICATION**-Zustand: Bei asynchron eintreffenden Daten wird `IND` ausgelöst, danach kehrt der FB in den IDLE-Zustand zurück.

Fehlerzustände (z. B. Kommunikationsabbruch) werden über `STATUS` gemeldet.

• – ◦ • – ◦ • – ◦ •

Anwendungsszenarien

- **4-20 mA Sensoranbindung**: Ein Drucksensor, Füllstandssensor oder Temperaturfühler mit 4-20 mA Ausgang wird an das DataPanel-System angeschlossen. Der FB liest zyklisch oder auf Anforderung den Messwert.
- **Mehrkanal-Erfassung**: Über verschiedene Instanzen des Bausteins (mit unterschiedlichen `Input`-Parametern) können mehrere analoge Kanäle parallel bedient werden.
- **Datenlogger-System**: Kombiniert mit einem FB zur Datenaufzeichnung kann der `IND`-Ausgang genutzt werden, um Statusänderungen oder Alarme zu protokollieren.
- **Landtechnische Steuerung**: In der Agrartechnik (HR Agrartechnik GmbH) werden so z. B. Flüssigkeitsstände, Durchflüsse oder Achslasten erfasst.

• – ◦ • – ◦ • – ◦ •

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem allgemeinen analogen Eingangs-FB (z. B. einem FB für standardisierte Feldbusse wie PROFIBUS oder IO-Link) zeichnet sich dieser Baustein durch folgende Punkte aus:

- **Spezifische Hardware-Zuordnung**: Er ist direkt auf das „DataPanel::io::MI::AI“-Protokoll zugeschnitten.
- **Knotenadressierung**: Die Begrenzung auf 224–239 und die vordefinierten Konstanten aus `MI::MI_xx` erleichtern die Konfiguration in festen Netzwerken.
- **Ereignis-Modell**: `IND` neben `CNF` erlaubt sowohl synchrone als auch asynchrone Messwerterfassung – flexibler als ein reines Polling-Modell.
- **Hysterese-Parameter**: Bietet eine einfache Entprellung auf FB-Ebene, die in vielen generischen Bausteinen nicht vorhanden ist.

• – ◦ • – ◦ • – ◦ •

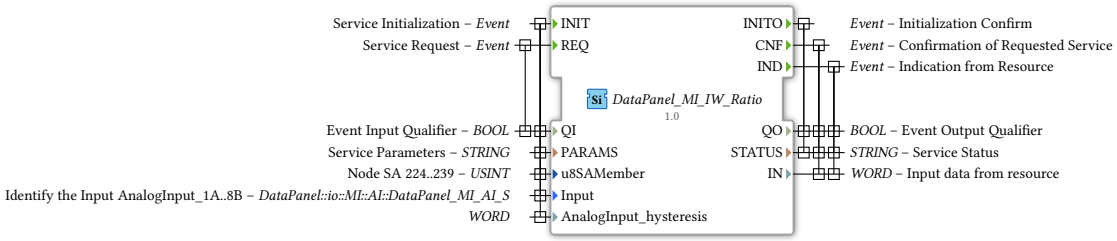
Fazit

Der Funktionsblock **DataPanel_MI_IW_4_20mA** stellt eine robuste und kompakte Lösung für die Erfassung von 4-20 mA-Signalen in einem proprietären DataPanel-System dar. Die klare Trennung von Initialisierung, Anforderung und spontaner Indikation sowie die integrierte Hysterese machen ihn für den praktischen Einsatz in der Agrartechnik geeignet. Durch die Parametrierbarkeit von Knotenadresse und Kanal ist er flexibel einsetzbar und in bestehende Automatisierungsnetzwerke integrierbar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

DataPanel_MI_IW_Ratio



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **DataPanel_MI_IW_Ratio** ist ein Service-Interface-Funktionsblock (SIFB) zur Erfassung analoger Eingangsdaten mit ratiometrischer Wandlung. Er stellt die Schnittstelle zu einem Sensor dar, der an einem Knoten (SA 224..239) angeschlossen ist und dessen Messwert als 16-Bit-Wort (WORD) ausgegeben wird. Der Baustein ist Teil der Bibliothek `DataPanel::io::MI::AI` und wird typischerweise in der Automatisierungstechnik, insbesondere im Bereich der Agrartechnik, eingesetzt.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Mit-Variablen	Kommentar
INIT	EInit	QI, PARAMS, u8SAMember, Input, AnalogInput_hysteresis	Service Initialization
REQ	Event	QI	Service Request

Ereignis-Ausgänge

Ereignis	Typ	Mit-Variablen	Kommentar
INITO	EInit	QO, STATUS	Initialization Confirm
CNF	Event	QO, STATUS, IN	Confirmation of Requested Service
IND	Event	QO, STATUS, IN	Indication from Resource

Daten-Eingänge

Variable	Typ	Initialwert	Kommentar
QI	BOOL	–	Event Input Qualifier
PARAMS	STRING	–	Service Parameters
u8SAMember	USINT	MI::MI_00	Node SA 224..239
Input	DataPanel::io::MI::AI::DataPanel_MI_AI_IS	Invalid	Identify the Input AnalogInput_1A..8B
AnalogInput_hysteresis	WORD	–	(keine Angabe)

Daten-Ausgänge

Variable	Typ	Kommentar
QO	BOOL	Event Output Qualifier
STATUS	STRING	Service Status
IN	WORD	Input data from resource

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Baustein arbeitet als zustandsgesteuerter Service-Interface-Block:

1. **Initialisierung (INIT)**: Über das INIT-Ereignis wird der Funktionsblock konfiguriert. Die Eingänge `PARAMS`, `u8SAMember`, `Input` und `AnalogInput_hysteresis` legen die Bus-/Knotenadresse, den analogen Kanal und die Hysterese fest. Der `QI`-Eingang muss auf TRUE gesetzt sein, um die Initialisierung zu starten. Nach erfolgreicher Konfiguration wird das INITO-Ereignis mit `QO = TRUE` und `STATUS = "OK"` (o. ä.) ausgegeben.

2. **Messwertanforderung (REQ)**: Mit dem REQ-Ereignis wird eine neue Messung des Sensors angefordert. Der Baustein wertet die Hardware aus und liefert das Ergebnis über das CNF-Ereignis am Ausgang `IN` zurück. Auch hier zeigt `QO` die Gültigkeit des Wertes an.
3. **Indikation (IND)**: Falls die Hardware asynchrone Ereignisse (z. B. zyklische Aktualisierungen) unterstützt, wird das IND-Ereignis verwendet, um den aktuellen Messwert unaufgefordert zu melden.

Der Messwert wird als 16-Bit-Wort (WORD) im ratiometrischen Format ausgegeben. Dies bedeutet, dass der digitale Wert direkt proportional zum Verhältnis der gemessenen Spannung zur Referenzspannung ist.

Technische Besonderheiten

- **Ratiometrische Messung**: Der Funktionsblock ist speziell für ratiometrische Sensoren ausgelegt, bei denen der Ausgang proportional zur Versorgungsspannung ist. Dadurch werden Messfehler durch Spannungsschwankungen minimiert.
- **Konfiguration über Konstanten**: Der initiale Wert von `u8SAMember` (`MI::MI_00`) und der Datentyp `DataPanel_MI_AI_S` stammen aus importierten Bibliotheken (`DataPanel::io::MI::const::MI` und `DataPanel::io::MI::AI::DataPanel_MI_AI`). Der gültige Adressbereich für den Knoten liegt zwischen 224 und 239 (Node SA 224..239).
- **Hysterese**: Mit `AnalogInput_hysteresis` kann ein Schwellwert-Hystereseverhalten implementiert werden; der genaue Effekt ist vom zugrundeliegenden Treiber abhängig.
- **TypeHash-Attribut**: Das Attribut `eclipse4diac::core::TypeHash` dient der eindeutigen Typidentifikation im Laufzeitsystem.

Zustandsübersicht

Da der Baustein als Service-Interface-Funktionsblock realisiert ist, besitzt er eine interne Zustandsmaschine. Die typischen Zustände sind:

- **IDLE**: Warten auf INIT-Ereignis.
- **INIT**: Ausführung der Initialisierung (Parametrierung, Hardwarezugriff).
- **READY**: Initialisierung abgeschlossen, auf REQ oder IND wartend.
- **REQ**: Verarbeitung einer Messanforderung.
- **IND**: Asynchrone Indikation wird bearbeitet.

Im Fehlerfall wird `QO = FALSE` gesetzt und `STATUS` enthält einen entsprechenden Fehlertext.

Anwendungsszenarien

- **Landmaschinen**: Erfassung von Sensorwerten (z. B. Füllstände, Druck, Position) über den ratiometrischen Eingang eines Datenpanels.
- **Industrielle Automatisierung**: Anbindung von analogen Sensoren mit Spannungsausgang (0…5 V, 0…10 V), die ratiometrisch arbeiten.
- **Frühe Prototypen**: Der Baustein kann direkt aus der 4diac-IDE in eine Steuerung eingebunden und mit beliebigen Parametern getestet werden.

Vergleich mit ähnlichen Bausteinen

Baustein	Merkmal
`DataPanel_MI_AI`	Standardanaloger Eingang ohne explizite ratiometrische Auslegung.
`DataPanel_MI_IW_Voltage`	Spannungsmessung mit absoluten Werten (z. B. mV).
DataPanel_MI_IW_Ratio	Speziell für ratiometrische Sensoren optimiert.

Der hier beschriebene Baustein unterscheidet sich vor allem durch die Verwendung des ratiometrischen Messprinzips, das bei vielen modernen Sensoren (z. B. Hallgebern, Potentiometern) zum Einsatz kommt.

Fazit

Der `DataPanel_MI_IW_Ratio` ist ein spezialisierter Service-Interface-Funktionsblock für die zuverlässige Erfassung ratiometrischer Analogsignale. Mit seiner klar definierten Schnittstelle (INIT/REQ/IND) und den konfigurierbaren Parametern eignet er sich ideal für den Einsatz in agrar- und automatisierungstechnischen Anwendungen. Die Einbindung in die 4diac-IDE erfolgt über die Bibliothek `DataPanel::io::MI::AI`.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

DI

📺 Video

- Decoding the datasheet
- Die große Migration
- Die Kunst des Lötens
- Die Welt der Normung
- Digital decisions

🎧 Podcast

- 4diac IDE: Dein „Hello World“ der Automatisierung – Das Blinking Tutorial Lokal
- 4diac IDE: Dein Open-Source-Werkzeugkasten für verteilte Industrieautomatisierung nach IEC 61499
- 4diac IDE: Wie der IEC 61499 Standard die Industrieautomatisierung revolutioniert
- 4diac-Präsentation: Zielgruppen, Struktur und Alleinstellungsmerkmal Schärfen
- Den Software-Drachen zähmen: Industrielle Automatisierung und die Zukunft der Produktion

---

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

DataPanel_MI_DI

• – ◦ • – ◦ • – ◦ •

Einleitung

DataPanel_MI_DI ist eine globale Konstantendefinition für DataPanel-Module mit mittlerer I/O-Dichte für digitale Eingänge (DI). Diese Konstante definiert die Pin-Zuordnungen für verschiedene digitale Eingangskanäle eines DataPanel-Systems.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereigniseingänge vorhanden

Ereignis-Ausgänge

Keine Ereignisausgänge vorhanden

Daten-Eingänge

Keine Dateneingänge vorhanden

Daten-Ausgänge

Keine Datenausgänge vorhanden

Adapter

Keine Adapter vorhanden

Funktionsweise

Bei DataPanel_MI_DI handelt es sich um eine globale Konstantendefinition, die feste Werte für die Pin-Zuordnungen der digitalen Eingänge bereitstellt. Die Konstante definiert strukturierte Daten vom Typ `DataPanel_MI_DI_S` mit Pin-Nummern für verschiedene digitale Eingangskanäle.

Technische Besonderheiten

- Definiert Pin-Zuordnungen für 8 digitale Eingangskanäle (5A, 5B, 6A, 6B, 7A, 7B, 8A, 8B)
- Pin-Nummern reichen von 9 bis 16
- Enthält eine spezielle „Invalid"-Konstante mit Pin-Wert 255 für Fehlerbehandlung
- Verwendet den strukturierten Datentyp `DataPanel_MI_DI_S`
- Organisiert im Package `DataPanel::io::MI::DI`

Zustandsübersicht

Keine Zustandsübergänge vorhanden - handelt es sich um eine statische Konstantendefinition

Anwendungsszenarien

- Konfiguration von DataPanel-Systemen mit mittlerer I/O-Dichte
- Hardware-Abstraktion für digitale Eingänge in Steuerungssystemen
- Vereinfachung der Pin-Zuordnung in IEC 61499-basierten Anwendungen
- Wiederverwendbare Konstantendefinition für mehrere Projekte

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Konstantendefinitionen bietet DataPanel_MI_DI:

- Spezifische Pin-Zuordnungen für DataPanel-Hardware
- Strukturierte Datentypen für bessere Typsicherheit
- Vordefinierte Invalid-Konstante für Robustheit

Fazit

DataPanel_MI_DI stellt eine standardisierte und typsichere Möglichkeit dar, digitale Eingangspins in DataPanel-Systemen mit mittlerer I/O-Dichte zu konfigurieren. Die Verwendung strukturierter Konstanten erhöht die Wartbarkeit und Reduziert Konfigurationsfehler in Automatisierungsprojekten.

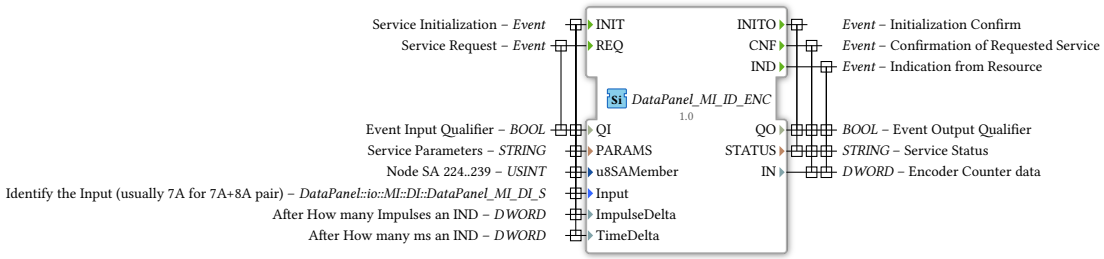
DataPanel_MI_DI_S

DataPanel Modules with medium IO density DI Struct

Structure

Name	Type	Initial Value
Pin	USINT	255

DataPanel_MI_ID_ENC



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **DataPanel_MI_ID_ENC** ist ein Eingangs-Service-Interface-Funktionsblock zur Erfassung von Encoder-Impulsdaten. Er ist für die Verarbeitung eines 7A+8A-Encoder-Paares ausgelegt und gibt den aktuellen Zählerstand sowie Statusinformationen aus. Die Initialisierung erfolgt über Parameter wie die Knotenadresse (SA-Member), die Eingangskonfiguration sowie Schwellwerte für impuls- und zeitbasierte Ereignisauslösung.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung	Mitgeführte Daten
INIT	Service-Initialisierung	QI, PARAMS, u8SAMember, Input, ImpulseDelta, TimeDelta
REQ	Service-Anforderung	QI

Ereignis-Ausgänge

Ereignis	Beschreibung	Mitgeführte Daten
INITO	Bestätigung der Initialisierung	QO, STATUS
CNF	Bestätigung der angeforderten Aktion	QO, STATUS, IN
IND	Asynchrone Anzeige eines Ereignisses (Impuls- oder Zeitüberschreitung)	QO, STATUS, IN

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung
QI	BOOL	–	Ereignis-Eingangsqualifizierer
PARAMS	STRING	–	Service-Parameter
u8SAMember	USINT	MI::MI_00	Knotenadresse (Bereich 224..239)
Input	DataPanel::io::MI::DI::DataPanel_MI_DI_S	Invalid	Identifikation des Eingangs (üblicherweise 7A für 7A+8A-Paar)
ImpulseDelta	DWORD	–	Anzahl der Impulse, nach denen ein IND ausgelöst wird
TimeDelta	DWORD	–	Zeit in Millisekunden, nach der ein IND ausgelöst wird

Daten-Ausgänge

Name	Typ	Beschreibung
QO	BOOL	Ereignis-Ausgangsqualifizierer
STATUS	STRING	Servicestatus
IN	DWORD	Aktueller Encoder-Zählerwert

Adapter

Keine.

Funktionsweise

1. **Initialisierung (INIT → INITO)**
   Der Baustein wird mit den Parametern `PARAMS`, der Knotenadresse `u8SAMember`, der Eingangsspezifikation `Input` sowie den Schwellwerten `ImpulseDelta` und `TimeDelta` konfiguriert. Nach erfolgreicher Initialisierung wird das Ereignis `INITO` mit `QO` und `STATUS` quittiert.
2. **Anforderung (REQ → CNF)**
   Mit `REQ` wird eine gezielte Abfrage des aktuellen Zählerstands ausgelöst. Der Baustein antwortet mit `CNF` und liefert den aktuellen Encoderwert über `IN` sowie Statusinformationen.
3. **Asynchrone Ereignisse (IND)**
   Unabhängig von einer expliziten Anforderung wird `IND` ausgelöst, sobald entweder die in `ImpulseDelta` festgelegte Anzahl von Encoder-Impulsen überschritten wurde oder die in `TimeDelta`

definierte Zeitspanne abgelaufen ist. Dies ermöglicht eine ereignisgesteuerte Verarbeitung ohne ständiges Pollen.

Der Ausgang `IN` enthält zu jedem Ereignis (CNF und IND) den aktuellen 32-Bit-Zählerwert des Encoders.

Technische Besonderheiten

- **Benutzerdefinierte Typen**: Der Eingang `Input` basiert auf dem Datentyp `DataPanel_MI_DI_S`, der eine spezifische Eingangskonfiguration (z. B. 7A) erwartet. Der Konstantwert `Invalid` dient als Platzhalter für nicht definierte Eingänge.
- **Konfigurierbare Ereignisauslösung**: Durch kombinierte Nutzung von `ImpulseDelta` und `TimeDelta` kann das System entweder nach einer bestimmten Impulsanzahl, nach einer Zeitspanne oder nach dem zuerst eintretenden Ereignis (UND-Verknüpfung) einen IND auslösen.
- **Knotenadressierung**: Über `u8SAMember` (Wertebereich 224–239) wird der physikalische Knoten im Bus-System ausgewählt. Der Initialwert `MI::MI_00` verweist auf eine Konstante des Moduls `MI`.

Zustandsübersicht

Zustand	Beschreibung
IDLE	Warten auf INIT oder REQ
INIT	Initialisierung läuft, Parametrierung wird übernommen
ACTIVE	Initialisierung abgeschlossen, bereit für REQ und IND
ERROR	Fehlerzustand (z. B. fehlerhafte Initialisierung)

Die tatsächliche Zustandsmaschine ist im vorliegenden Code nicht explizit abgebildet; die dargestellten Zustände leiten sich aus dem typischen Verhalten von Service-Interface-FBs ab.

Anwendungsszenarien

- **Landmaschinensteuerung**: Erfassung von Drehzahlen an Antriebswellen über Inkrementalgeber (7A+8A-Paar) zur Überwachung und Regelung von Arbeitsprozessen.
- **Positionserfassung**: Verwendung als Impulszähler zur Wegmessung, z. B. für Fahrantriebe oder Stellglieder.
- **Ereignisgesteuerte Datenerfassung**: Durch die konfigurierbaren IND-Schwellen können Lastspitzen im Datenverkehr vermieden und Logging-Aufgaben optimiert werden.

Vergleich mit ähnlichen Bausteinen

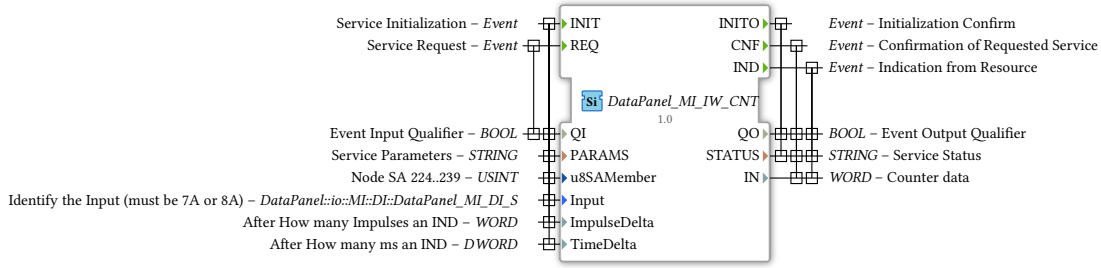
Gegenüber einfachen Encoder-Zählern bietet der DataPanel_MI_ID_ENC:

- **Flexible Auslösekriterien**: Statt nur auf Polling angewiesen zu sein, können Impuls- oder Zeit-Schwellwerte gesetzt werden.
- **Strukturierte Initialisierung**: Die Verwendung eines benutzerdefinierten Typs (`DataPanel_MI_DI_S`) ermöglicht eine klare Zuordnung zu spezifischen Hardware-Eingängen.
- **Ereignisbasierte Ausgabe**: Die Trennung von `CNF` (synchron zur Anfrage) und `IND` (asynchron) erlaubt eine entkoppelte Verarbeitung in höheren Steuerungsebenen.

Fazit

Der Funktionsblock **DataPanel_MI_ID_ENC** ist ein leistungsfähiger Service-Interface-Baustein für die Encoder-Datenerfassung in industriellen Steuerungssystemen. Durch seine parametrierbaren Schwellwerte und die Unterstützung sowohl synchroner als auch asynchroner Ereignisse eignet er sich besonders für Echtzeitanwendungen in der Landtechnik. Die klare Schnittstellendefinition und die Verwendung spezifischer Datentypen ermöglichen eine einfache Integration in bestehende Automatisierungsumgebungen.

DataPanel_MI_IW_CNT



• – ○ • – ○ • – ○ •

Einleitung

Der **DataPanel_MI_IW_CNT** ist ein Service-Interface-Funktionsblock aus der DataPanel-Familie der HR Agrartechnik GmbH. Er dient der Erfassung von **Impulszählerdaten** über die speziellen Hardware-Eingänge 7A und 8A des zugrunde liegenden Bussystems. Der Baustein kapselt die Initialisierung, die zyklische Abfrage und die ereignisgesteuerte Ausgabe der Zählerwerte. Typische Einsatzbereiche sind landtechnische Anwendungen, in denen Impulsgeber (z. B. Drehzahl-, Durchfluss- oder Positionssensoren) ausgewertet werden müssen. Der FB ist für die IEC 61499-konforme 4diac IDE ausgelegt und nutzt die dort definierten Service-Interface-Muster.

Schnittstellenstruktur

Ereignis-Eingänge

Event	Typ	Beschreibung
INIT	EInit	Initialisiert den Kanal (setzt Hardware-Parameter)
REQ	Event	Fordert einen aktuellen Zählerwert an

Ereignis-Ausgänge

Event	Typ	Beschreibung
INITO	EInit	Quittierung der erfolgreichen Initialisierung
CNF	Event	Bestätigung einer angeforderten REQ-Operation
IND	Event	Asynchrone Indikation (ausgelöst durch Impuls- oder Zeitintervall)

Daten-Eingänge

Name	Typ	Beschreibung
QI	BOOL	Qualifiziert das INIT- / REQ-Ereignis
PARAMS	STRING	Service-Parameter (herstellerabhängige Konfiguration)
u8SAMember	USINT	Knoten-SA (224..239); Default = `MI::MI_00`
Input	DataPanel_MI_DI_S	Identifikation des Eingangs (muss „7A“ oder „8A“ sein); Default = `Invalid`
ImpulseDelta	WORD	Impulsschwelle für asynchrone IND-Auslösung (Anzahl Impulse)
TimeDelta	DWORD	Zeitschwelle für asynchrone IND-Auslösung (in ms)

Daten-Ausgänge

Name	Typ	Beschreibung

QO	BOOL	Qualifiziert die Ereignisausgänge
STATUS	STRING	Statusmeldung (z. B. „OK“ oder Fehlercode)
IN	WORD	Aktueller 16-Bit-Zählerwert (Impulszähler)

Adapter

Keine Adapter-Schnittstellen definiert.

Funktionsweise

Der **DataPanel_MI_IW_CNT** arbeitet als Service-Interface-FB und kommuniziert direkt mit der Hardware-Schnittstelle des DataPanel-Systems. Nach dem Start muss über das **INIT**-Ereignis der Kanal initialisiert werden. Dabei werden die Parameter (SA-Adresse, Eingangsnummer, Impuls- und Zeitkonfiguration) übergeben. Der FB versucht, den entsprechenden Hardware-Kanal zu belegen und bereitzustellen.

Nach erfolgreicher Initialisierung (signalisiert durch **INITO**) kann der Zählerstand über das **REQ**-Ereignis abgefragt werden. Die Antwort wird über **CNF** geliefert: `IN` enthält den aktuellen Zählerwert, `STATUS` den Betriebszustand.

Zusätzlich kann der Baustein asynchron eine **IND**-Indikation ausgeben, wenn entweder die eingestellte Anzahl an Impulsen (`ImpulseDelta`) erreicht wurde oder die Zeitspanne (`TimeDelta`) verstrichen ist. Dies erlaubt eine ereignisgesteuerte Verarbeitung ohne zyklische Abfragen.

Die Fehlerbehandlung erfolgt über den `STATUS`-Ausgang und das `QO`-Flag. Tritt ein Fehler auf (z. B. ungültige Eingangskonfiguration, Hardware nicht erreichbar), wird `QO` auf `FALSE` gesetzt und ein entsprechender Fehlertext in `STATUS` ausgegeben.

Technische Besonderheiten

- **Eingangsidentifikation:** Der Parameter `Input` muss auf einen gültigen DataPanel-Eingangstyp gesetzt werden (7A oder 8A). Der Defaultwert `Invalid` verhindert eine fehlerhafte Initialisierung.
- **Knoten-SA (`u8SAMember`):** Definiert die Slave-Adresse des DataPanel-Teilnehmers. Gültig sind Werte von 224 bis 239. Der vordefinierte Wert `MI::MI_00` steht für den ersten Slave.
- **Asynchrone Indikation:** Durch `ImpulseDelta` und `TimeDelta` kann der FB selbstständig **IND**-Ereignisse erzeugen. Beide Schwellen können unabhängig voneinander aktiv sein. Überschreitet einer der Werte den konfigurierten Schwellwert, wird ein IND ausgelöst. Dies reduziert die Buslast im Vergleich zu zyklischen Polling.
- **Datenformat:** Der Zählerwert `IN` ist ein 16-Bit-Word und ermöglicht Werte von 0 bis 65535. Bei Überlauf wird der Zähler auf 0 zurückgesetzt.
- **Copyright & Version:** Der Baustein ist unter der Version 1.0 für das Jahr 2026 von HR Agrartechnik GmbH veröffentlicht.

Zustandsübersicht

Der FB durchläuft klassische Service-Interface-Zustände:

Zustand	Beschreibung
IDLE	Warten auf INIT oder REQ. Hardware ist noch nicht belegt.
INIT	INIT empfangen – Parametrierung und Hardware-Reservierung laufen.
ACTIVE	Initialisierung erfolgreich – Kanal betriebsbereit.
REQUEST	REQ empfangen – Abfrage des aktuellen Zählerstands.
INDICATE	Asynchrone Bedingung erfüllt – Senden eines IND-Ereignisses.
ERROR	Fehler aufgetreten (z. B. falscher Parameter, Hardware-Fehler).

Nach einem Fehler kann nur ein erneutes INIT den FB zurück in den IDLE-Zustand versetzen.

Anwendungsszenarien

1. **Drehzahlmessung:** Ein Radarsensor oder magnetischer Impulsgeber liefert Rechteckimpulse. Der Baustein zählt die Impulse und gibt den Wert auf CNF oder IND aus. `TimeDelta` kann für eine zeitbasierte Drehzahlberechnung genutzt werden.

2. **Durchflusserfassung:** In der landtechnischen Beregnung werden Impulsdurchflussmesser (z. B. Hall-Effekt) eingesetzt. `ImpulseDelta` erzeugt ein Ereignis nach jeder definierten Flüssigkeitsmenge.
3. **Positionserfassung:** Mit einem Inkrementalgeber werden Wegstrecken gemessen. Durch die Kombination von Impuls- und Zeitüberwachung kann sowohl die Position als auch die Geschwindigkeit ermittelt werden.

Vergleich mit ähnlichen Bausteinen

- **DataPanel_MI_DI:** Ein reiner Digital-Eingangs-FB (Status 0/1) ohne Zählfunktion. Der IW_CNT erweitert ihn um die Impulszählung und asynchrone Ereignisauslösung.
- **DataPanel_MI_AI:** Ein Analog-Eingangs-FB für Spannungs- oder Stromsignale. Im Gegensatz dazu verarbeitet der IW_CNT ausschließlich Impulse (diskret) und keinen kontinuierlichen Wert.
- **CTUD** (IEC 61499-Standard-Zähler): Dieser Standardbaustein kann vorwärts/rückwärts zählen, benötigt aber keine hardwarenahe Initialisierung. Der DataPanel_MI_IW_CNT ist speziell an die DataPanel-Hardware angebunden und bietet die IND-Funktion, die in Standardbausteinen oft fehlt.

Fazit

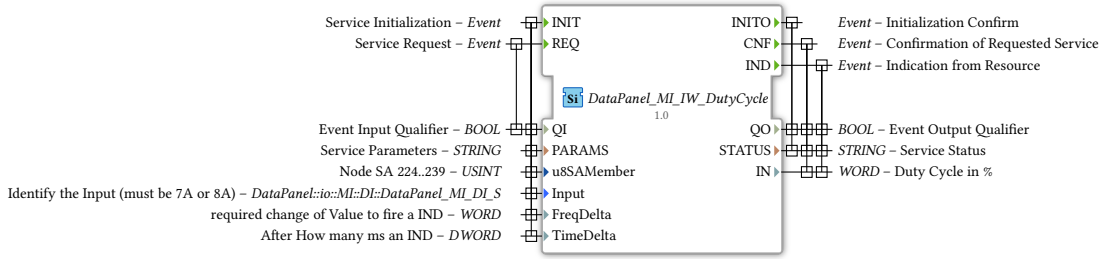
Der **DataPanel_MI_IW_CNT** ist ein leistungsfähiger Service-Interface-Funktionsblock für die flexible Erfassung von Impulszählerdaten in landtechnischen Steuerungen. Seine integrierte asynchrone Indikation auf Basis von Impuls- oder Zeitschwellen reduziert die Reaktionszeit und entlastet das Steuerungssystem. Dank der klar strukturierten Schnittstelle und der spezifischen Konfigurationsmöglichkeiten (SA-Adresse, Eingangswahl) lässt er sich nahtlos in das DataPanel-Ökosystem einbinden. Er stellt damit eine zuverlässige Komponente für Anwendungen dar, die eine präzise Impulsauswertung auf Systemebene erfordern.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

DataPanel_MI_IW_DutyCycle

Kein Bild verfügbar.



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `DataPanel_MI_IW_DutyCycle` ist ein Service-Interface-FB zur Erfassung und Verarbeitung von Frequenzeingangssignalen (Typ 7A/8A) im Hardwaresystem. Er berechnet aus dem eingehenden Frequenzsignal das Tastverhältnis (Duty Cycle) und gibt dieses als Prozentwert aus. Der Baustein unterstützt eine initiale Parametrierung sowie ereignisgesteuerte Ausgaben bei signifikanten Wertänderungen oder zeitgesteuerten Abfragen.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Mit Variablen	Kommentar
`INIT`	EInit	QI, PARAMS, u8SAMember, Input, FreqDelta, TimeDelta	Initialisierung des Dienstes
`REQ`	Event	QI	Dienstanforderung

Ereignis-Ausgänge

Ereignis	Typ	Mit Variablen	Kommentar
INITO	EInit	QO, STATUS	Bestätigung der Initialisierung
CNF	Event	QO, STATUS, IN	Bestätigung der angeforderten Aktion
IND	Event	QO, STATUS, IN	Indikation vom Ressourcen (bei Wertänderung/Timer)

Daten-Eingänge

Variable	Typ	Kommentar
QI	BOOL	Ereignis-Eingangsqualifizierer
PARAMS	STRING	Dienstparameter (z. B. Konfigurationsstring)
u8SAMember	USINT	Knotenadresse (SA 224..239, Standard: MI::MI_00)
Input	DataPanel_MI_DI_S	Identifikation des Eingangs (muss 7A oder 8A sein, Initialwert: Invalid)
FreqDelta	WORD	Erforderliche Wertänderung in %, um eine IND auszulösen
TimeDelta	DWORD	Zeitintervall in ms, nach dem eine IND ausgelöst wird

Daten-Ausgänge

Variable	Typ	Kommentar
QO	BOOL	Ereignis-Ausgangsqualifizierer
STATUS	STRING	Dienststatus (Fehler-/Erfolgsmeldung)
IN	WORD	Gemessenes Tastverhältnis (Duty Cycle) in Prozent (0..100)

Adapter

Keine Adapter definiert.

Funktionsweise

Der Baustein initialisiert sich über das Ereignis INIT. Dabei werden die Parameter QI (Aktivierung), PARAMS (allgemeine Konfiguration), die Knotenadresse u8SAMember, der physikalische Eingang (Input), das Änderungsdelta FreqDelta und das Zeitintervall TimeDelta übergeben. Nach erfolgreicher Initialisierung wird INITO mit gültigem QO und STATUS ausgegeben.

Ein REQ-Ereignis löst eine sofortige Abfrage des aktuellen Duty-Cycle aus. Das Ergebnis wird über CNF mit dem aktuellen Wert in IN ausgegeben.

Der Baustein überwacht den Eingang kontinuierlich. Ändert sich der gemessene Duty-Cycle um mehr als den in FreqDelta angegebenen Betrag, so wird ein IND-Ereignis mit dem neuen Wert gesendet. Zusätzlich wird nach Ablauf von TimeDelta Millisekunden (sofern keine zwischenzeitliche Änderung erfolgte) ebenfalls ein IND ausgelöst, um eine regelmäßige Aktualisierung zu gewährleisten.

Technische Besonderheiten

- Der Baustein ist für den Anschluss an Frequenzeingänge vom Typ 7A oder 8A ausgelegt. Der Eingang wird über die Struktur DataPanel_MI_DI_S identifiziert – standardmäßig auf Invalid gesetzt, bis eine gültige Initialisierung erfolgt.
- Die Knotenadresse (SA) ist auf den Bereich 224–239 beschränkt. Voreingestellt ist MI::MI_00.
- Die Parameter FreqDelta und TimeDelta ermöglichen eine flexible Konfiguration des Meldeverhaltens: Entweder reine Änderungsauslösung, reine Zeitauslösung oder eine Kombination beider.
- Der Status QO zeigt die Gültigkeit der Ausgangsdaten an; STATUS enthält detaillierte Fehler- oder Erfolgsmeldungen.
- Compiler-Importe verweisen auf die Pakete DataPanel::io::MI::const::MI und DataPanel::io::MI::DI::DataPanel_MI_DI::Invalid sowie auf eclipse4diac::core::TypeHash.

Zustandsübersicht

Eine explizite Zustandsmaschine ist im XML nicht definiert. Aus dem Verhalten der Ereignisse lässt sich jedoch folgender Ablauf ableiten:

1. **Initialisierungszustand** – Nach `INIT` wird der Baustein konfiguriert und geht in den Betriebszustand über. Fehler führen zu `INITO` mit negativem `QO`.
2. **Betriebszustand** – Auf `REQ` folgt `CNF` mit dem aktuellen Duty-Cycle. Bei Änderungen oder Timerablauf wird `IND` gesendet.
3. **Fehlerzustand** – Bei ungültigen Parametern oder Hardwarefehlern wird `STATUS` entsprechend gesetzt und `QO` auf `FALSE`.

Anwendungsszenarien

- **Drehzahlerfassung in Landmaschinen** – Überwachung von Frequenzsignalen (z. B. von Drehzahlsensoren) und Ausgabe des Tastverhältnisses als Indikator für Motordrehzahl oder Fördergeschwindigkeit.
- **Pulsweitenmodulation (PWM)-Analyse** – Messung des Duty-Cycle eines PWM-Signals zur Steuerung von Aktoren oder zur Rückmeldung über den Füllstand.
- **Schwingungs-/Frequenzüberwachung** – Erfassung von periodischen Signalen mit konfigurierbarer Empfindlichkeit (FreqDelta) und Aktualisierungsrate (TimeDelta).

Vergleich mit ähnlichen Bausteinen

- **`DataPanel_MI_DI`**: Ein digitaler Eingangsbaustein ohne Frequenz- oder Duty-Cycle-Berechnung. Liefert nur binäre Zustände.
- **`DataPanel_MI_AI`**: Analoger Eingangsbaustein für Spannungs- oder Stromsignale; nicht für Frequenzsignale optimiert.
- **`DataPanel_MI_IW_Frequency`**: Misst die absolute Frequenz, nicht das Tastverhältnis. `DataPanel_MI_IW_DutyCycle` ergänzt diesen um die prozentuale Ein-/Aus-Zeit.

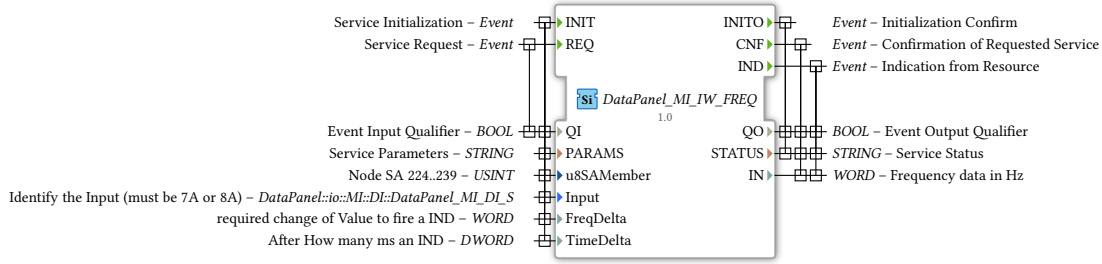
Fazit

Der Funktionsblock `DataPanel_MI_IW_DutyCycle` bietet eine robuste und konfigurierbare Schnittstelle zur Erfassung von Tastverhältnissen aus Frequenzeingängen im Bereich 7A/8A. Durch die Parameter `FreqDelta` und `TimeDelta` kann das Meldeverhalten flexibel an die Anwendung angepasst werden. Er eignet sich besonders für den Einsatz in der Agrartechnik und industriellen Steuerungen, bei denen eine zuverlässige Duty-Cycle-Überwachung erforderlich ist.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 Das PWM-Signal & Infografik auf ms-muc-docs.de

DataPanel_MI_IW_FREQ



Kein Bild verfügbar.

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **DataPanel_MI_IW_FREQ** ist ein Service-Interface-Funktionsblock (SIFB), der den Zugriff auf einen Frequenzeingang eines Datenpanels (Typ 7A/8A) kapselt. Er dient der Initialisierung, der zyklischen oder ereignisgesteuerten Abfrage sowie der asynchronen Benachrichtigung bei Frequenzänderungen. Der FB ist Teil einer modularen Steuerungsumgebung für landwirtschaftliche Maschinen (MI – Maschineninterface).

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung	Mit
INIT	Service-Initialisierung	`QI`, `PARAMS`, `u8SAMember`, `Input`, `FreqDelta`, `TimeDelta`
REQ	Service-Anforderung (aktuelle Frequenz auslesen)	`QI`

Ereignis-Ausgänge

Ereignis	Beschreibung	Mit
INITO	Bestätigung der Initialisierung	`QO`, `STATUS`
CNF	Bestätigung einer angeforderten Abfrage	`QO`, `STATUS`, `IN`
IND	Asynchrone Indikation (Frequenzänderung oder Zeitablauf)	`QO`, `STATUS`, `IN`

Daten-Eingänge

Name	Typ	Anfangswert	Beschreibung
`QI`	BOOL	–	Ereignis-Eingangsqualifizierer (Steuert die Verarbeitung)
`PARAMS`	STRING	–	Service-Parameter (gerätespezifische Konfiguration)
`u8SAMember`	USINT	`MI::MI_00`	Knotenadresse (SA) des Datensammelmoduls (Wertebereich 224…239)
`Input`	`DataPanel::io::MI::DI::DataPanel_MI_DI_Invalid`	–	Identifiziert den physikalischen Eingang (muss `7A` oder `8A` sein)
`FreqDelta`	WORD	–	Schwellwert der Frequenzänderung [Hz], der eine `IND` auslöst
`TimeDelta`	DWORD	–	Zeitintervall [ms], nach dem eine `IND` gesendet wird (auch ohne Änderung)

Daten-Ausgänge

Name	Typ	Beschreibung
`QO`	BOOL	Ereignis-Ausgangsqualifizierer (zeigt erfolgreiche Ausführung an)
`STATUS`	STRING	Servicestatus (z. B. Fehlermeldungen oder Bestätigungstexte)
`IN`	WORD	Aktuelle Frequenz in Hz

Adapter

Keine vorhanden.

Funktionsweise

1. **Initialisierung** – Durch das Ereignis `INIT` wird die Verbindung zum konfigurierten Frequenzeingang hergestellt. Die Parameter `PARAMS`, `u8SAMember`, `Input`, `FreqDelta` und `TimeDelta` werden übernommen. Nach erfolgreicher Initialisierung wird das Ausgangsereignis `INITO` mit `QO = TRUE` und einem positiven `STATUS` ausgelöst.

2. **Abfrage (REQ/CNF)** – Ein `REQ`-Ereignis fordert den aktuellen Frequenzwert an. Der FB sendet die Anfrage an das Datenpanel und liefert bei Antwort das Ereignis `CNF` mit dem aktuellen Wert in `IN`. `QO` zeigt an, ob die Abfrage erfolgreich war.
3. **Asynchrone Indikation (IND)** – Der FB überwacht den Frequenzeingang kontinuierlich. Eine `IND` wird ausgelöst, wenn:
   - sich der Frequenzwert um mindestens `FreqDelta` [Hz] ändert, oder
   - seit der letzten `IND` die in `TimeDelta` [ms] festgelegte Zeitspanne verstrichen ist.

   Dies ermöglicht sowohl schwellwertbasierte als auch zeitgesteuerte Aktualisierungen.

Die Ereignisse `INIT` und `REQ` werden nur ausgeführt, wenn der zugehörige Qualifizierer `QI` den Wert `TRUE` hat. Die Ausgangsqualifizierer `QO` signalisieren die erfolgreiche Durchführung.

Technische Besonderheiten

- **Typabhängigkeit**: Der FB verwendet spezielle Typen aus dem Paket `DataPanel::io::MI::DI`, die eine typsichere Konfiguration des Eingangs (z. B. `DataPanel_MI_DI_S`) sowie Konstanten (`MI::MI_00`) ermöglichen.
- **Initialisierung des Eingangs**: Der Daten-Eingang `Input` ist standardmäßig auf `Invalid` gesetzt. Vor dem ersten `INIT` muss dieser auf einen gültigen Wert (`7A` oder `8A`) gesetzt werden, sonst schlägt die Initialisierung fehl.
- **Frequenzdarstellung**: Die gemessene Frequenz wird als 16-Bit-Wert (`WORD`) in Hz ausgegeben.
- **Ereignisgesteuerter Abruf**: Der `INIT`-Dienst muss vor der Nutzung von `REQ` oder dem Empfang von `IND` einmal erfolgreich ausgeführt werden.
- **Attribut-Hash**: Ein `eclipse4diac::core::TypeHash`-Attribut ist vorhanden, jedoch ohne Wert gesetzt – dient der Laufzeitoptimierung in 4diac.

Zustandsübersicht

Der FB durchläuft intern folgende logische Zustände:

Zustand	Beschreibung
IDLE	Warten auf `INIT` – keine Verbindung
INIT	Verbindung zum Eingang aufbauen und Parameter anwenden
RUN	Betriebsbereit – wartet auf `REQ` oder sendet `IND` bei Änderungen/Zeitablauf
ERROR	Fehlerzustand (z. B. falscher `Input`, Kommunikationsfehler) – `STATUS` enthält Fehlermeldung

Ein Wechsel in den Fehlerzustand erfolgt, wenn die Initialisierung fehlschlägt. Aus dem Fehlerzustand kann nur durch erneutes `INIT` zurückgekehrt werden.

Anwendungsszenarien

- **Drehzahlmessung** an einer landwirtschaftlichen Maschine: Ein Frequenzsensor (z. B. magnetischer Geber) liefert Impulse, die über das Datenpanel erfasst werden. Mit `FreqDelta = 5 Hz` und `TimeDelta = 100 ms` erhält die Steuerung sowohl bei wesentlichen Drehzahländerungen als auch regelmäßig aktuelle Werte.
- **Überwachung einer konstanten Frequenz**: Bei niedriger Änderungsschwelle (`FreqDelta = 1 Hz`) und kurzem Zeitintervall (`TimeDelta = 50 ms`) eignet sich der FB zur Echtzeitüberwachung kritischer Prozesse.
- **Redundante Frequenzerfassung**: Zwei parallele Instanzen des FBs auf unterschiedlichen Eingängen (7A und 8A) ermöglichen eine Plausibilisierung der Messwerte.

Vergleich mit ähnlichen Bausteinen

Funktionsblock	Typ	Besonderheit
`DataPanel_MI_IW_FREQ`	Frequenzeingang (SIFB)	Ereignisgesteuert, asynchrone IND, parametrierbare Schwell- und Zeitwerte

`DataPanel_MI_DI`	Digitaleingang (SIFB)	Nur binäre Zustände, keine frequenzabhängigen Auslöser
Generischer `SIFB` mit INIT/REQ/IND	Allgemein	Keine eingebauten Frequenzfunktionen, muss selbst entwickelt werden

Der **DataPanel_MI_IW_FREQ** ist speziell für die Verarbeitung von Frequenzsignalen optimiert, während andere Bausteine entweder nur diskrete Zustände oder generische Schnittstellen bereitstellen.

Fazit

Der **DataPanel_MI_IW_FREQ** ist ein leistungsfähiger und flexibler Service-Interface-Funktionsblock für die Erfassung von Frequenzdaten über ein Datenpanel (Typ 7A/8A). Durch die Kombination aus schwellwertbasierter und zeitgesteuerter Indikation eignet er sich sowohl für einfache Messaufgaben als auch für sicherheitskritische Überwachungen. Die typsichere Konfiguration und die klare Ereignisschnittstelle erleichtern die Integration in Automatisierungslösungen der Landtechnik.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

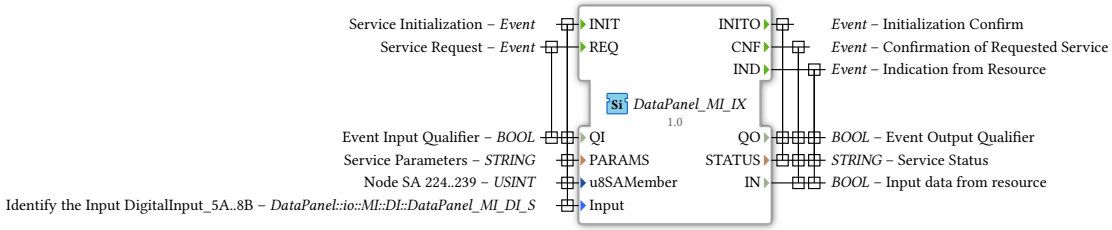
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

DataPanel_MI_IX

• – ◦ • – ◦ • – ◦ •

Einleitung

Der DataPanel_MI_IX ist ein Dienstschnittstellen-Funktionsblock für boolesche Eingabedaten, der speziell für die Kommunikation mit Digital-Eingängen in Automatisierungssystemen entwickelt wurde. Dieser Baustein ermöglicht die Initialisierung und Abfrage von digitalen Eingangssignalen über standardisierte Service-Schnittstellen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung mit Parametern QI, PARAMS, u8SAMember und Input
- **REQ**: Service-Anfrage mit Parameter QI

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung mit Parametern QO und STATUS
- **CNF**: Bestätigung der angeforderten Service-Operation mit Parametern QO, STATUS und IN
- **IND**: Indikation von der Ressource mit Parametern QO, STATUS und IN

Daten-Eingänge

- **QI**: BOOL - Ereignis-Eingangsqualifizierer
- **PARAMS**: STRING - Service-Parameter
- **u8SAMember**: USINT - Knoten SA 224..239 (Initialwert: MI::MI_00)
- **Input**: DataPanel_MI_DI_S - Identifiziert den Digital-Eingang 5A..8B (Initialwert: DataPanel_MI_DI::Invalid)

Daten-Ausgänge

- **QO**: BOOL - Ereignis-Ausgangsqualifizierer
- **STATUS**: STRING - Service-Status

- **IN**: BOOL - Eingabedaten von der Ressource

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Funktionsblock initialisiert über das INIT-Ereignis die Kommunikation mit einem spezifischen Digital-Eingang. Nach erfolgreicher Initialisierung können über das REQ-Ereignis Abfragen des Eingangszustands durchgeführt werden. Der Baustein liefert über die Ausgangsereignisse CNF und IND den aktuellen Zustand des Digital-Eingangs sowie Statusinformationen zurück.

Technische Besonderheiten

- Unterstützt spezifische SA-Member-Adressierung im Bereich 224-239
- Verwendet strukturierte Eingangsidentifikation über den DataPanel_MI_DI_S-Datentyp
- Bietet sowohl anforderungsbasierte (REQ/CNF) als auch indikationsbasierte (IND) Kommunikation
- Initialisierung mit konfigurierbaren Service-Parametern

Zustandsübersicht

Der Funktionsblock durchläuft folgende Zustände:

1. **Nicht initialisiert**: Vor INIT-Aufruf
2. **Initialisiert**: Nach erfolgreicher INIT/INITO-Sequenz
3. **Bereit**: Kann REQ/CNF-Zyklen verarbeiten
4. **Aktiv**: Verarbeitet eingehende IND-Ereignisse von der Ressource

Anwendungsszenarien

- Überwachung von Digital-Eingängen in Schaltschränken
- Integration in SPS-Steuerungen für Binärsignal-Erfassung
- Industrieautomation mit spezifischen SA-Adressierungsanforderungen
- Anlagen mit verteilten E/A-Systemen

⚖️ Vergleich mit ähnlichen Bausteinen

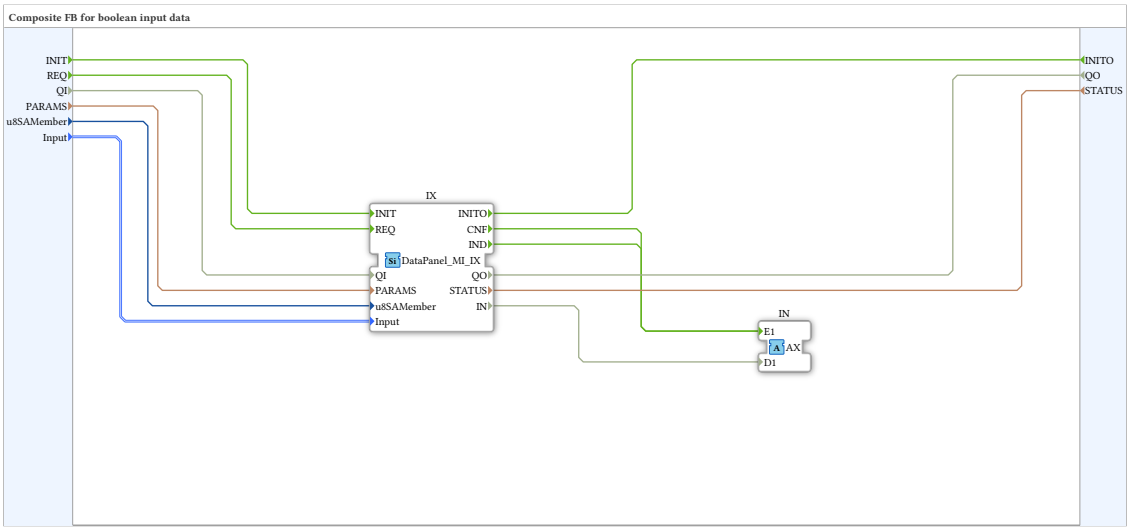
Im Vergleich zu generischen Digital-Input-Blöcken bietet DataPanel_MI_IX:

- Spezifische Adressierung für DataPanel-Systeme
- Erweiterte Initialisierungsmöglichkeiten mit Parameter-Strings
- Zusätzliche IND-Funktionalität für ereignisbasierte Kommunikation
- Strukturierte Eingangsidentifikation

Fazit

Der DataPanel_MI_IX ist ein spezialisierter Funktionsblock für die zuverlässige Erfassung digitaler Eingangssignale in DataPanel-Umgebungen. Durch seine flexible Parametrierung und beide Betriebsmodi (anforderungs- und ereignisbasiert) eignet er sich ideal für anspruchsvolle Automatisierungsanwendungen mit spezifischen Adressierungsanforderungen.

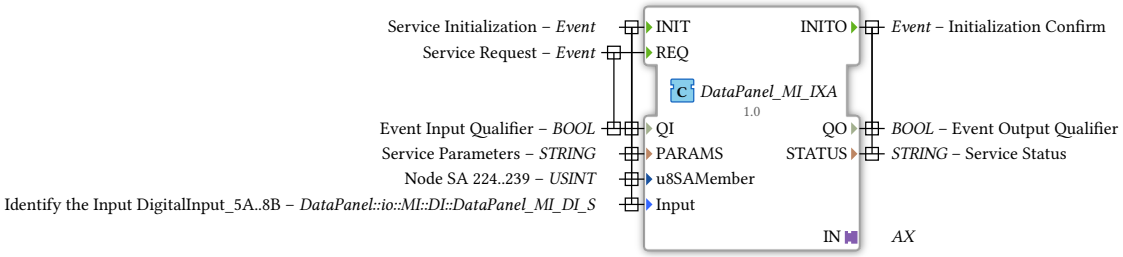
DataPanel_MI_IXA



• – ○ • – ○ • – ○ •

Einleitung

Der DataPanel_MI_IXA ist ein zusammengesetzter Funktionsblock für die Verarbeitung von booleschen Eingabedaten. Er dient als Schnittstelle für digitale Eingänge und ermöglicht die Initialisierung und Abfrage von Eingangsdaten über standardisierte Service-Schnittstellen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung mit den zugehörigen Daten QI, PARAMS, u8SAMember und Input
- **REQ**: Service-Anfrage mit dem zugehörigen Daten QI

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung mit den zugehörigen Daten QO und STATUS

Daten-Eingänge

- **QI**: Ereignis-Eingangsqualifizierer (BOOL)
- **PARAMS**: Service-Parameter (STRING)
- **u8SAMember**: Knoten SA 224..239 (USINT) mit Initialwert MI::MI_00
- **Input**: Identifizierung des digitalen Eingangs DigitalInput_5A..8B mit Initialwert Invalid

Daten-Ausgänge

- **QO**: Ereignis-Ausgangsqualifizierer (BOOL)
- **STATUS**: Service-Status (STRING)

Adapter

- **IN**: Unidirektionaler Adapter vom Typ AX für die Datenkommunikation

Funktionsweise

Der Funktionsblock fungiert als Wrapper für den internen DataPanel_MI_IX-Baustein und leitet alle Ereignisse und Daten entsprechend weiter. Bei der INIT-Initialisierung werden alle Parameter an den internen Baustein übergeben, und die Bestätigung erfolgt über INITO. REQ-Anfragen werden direkt an den internen Baustein weitergeleitet.

Technische Besonderheiten

- Verwendet spezielle Typen aus dem DataPanel::io::MI::DI-Namensraum
- Unterstützt SA-Mitglieder im Bereich 224-239
- Bietet Initialisierung mit Standardwerten für u8SAMember und Input
- Implementiert Fehlerbehandlung über den Invalid-Initialwert

Zustandsübersicht

Der Baustein verfügt über einen initialisierten und einen Betriebszustand. Nach erfolgreicher INIT-Initialisierung ist der Baustein betriebsbereit und kann REQ-Anfragen verarbeiten.

Anwendungsszenarien

- Anbindung digitaler Eingänge in Automatisierungssystemen
- Integration in DataPanel-IO-Strukturen
- Verwendung in Steuerungssystemen mit SA-Bus-Kommunikation
- Abstraktion von Hardware-Eingängen in standardisierten Schnittstellen

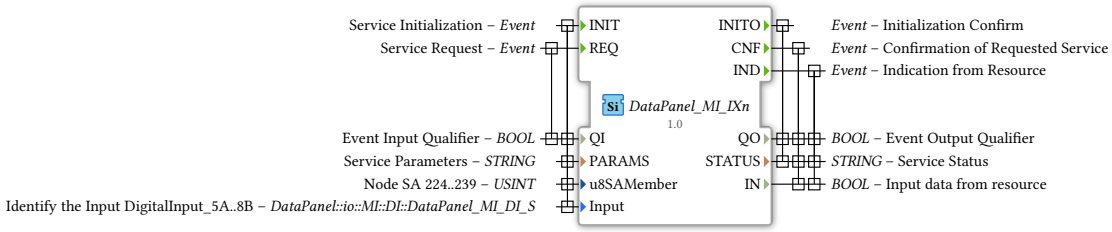
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Eingangsbausteinen bietet DataPanel_MI_IXA erweiterte Initialisierungsmöglichkeiten und Parameterisierung über den PARAMS-Eingang. Die SA-Mitgliedsnummerierung ermöglicht eine spezifische Adressierung in größeren Systemen.

Fazit

Der DataPanel_MI_IXA ist ein robuster und flexibler Funktionsblock für die Integration digitaler Eingänge in komplexe Automatisierungssysteme. Seine strukturierte Schnittstelle und erweiterte Parametrierungsmöglichkeiten machen ihn besonders für anspruchsvolle Anwendungen in industriellen Umgebungen geeignet.

DataPanel_MI_IXn



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock DataPanel_MI_IXn ist ein serviceorientierter Eingangsbaustein für digitale Binäreingänge (Ground Switching / npn). Er dient zum Auslesen eines einzelnen booleschen Eingangssignals über das MI-Protokoll (Module Interface) aus einem DataPanel-System. Der Baustein wird typischerweise in der 4diac-IDE verwendet, um Feldgeräte wie Schalter oder Sensoren anzubinden.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisiert den Baustein. Die Parameter QI, PARAMS, u8SAMember und Input werden übergeben.
- **REQ** (Event): Fordert das Lesen des aktuellen Eingangswerts an. Qualifiziert durch QI.

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigt die erfolgreiche Initialisierung. Liefert QO und STATUS.
- **CNF** (Event): Bestätigung einer ausgeführten Anforderung (REQ). Enthält QO, STATUS und den gelesenen Wert IN.
- **IND** (Event): Zeigt eine asynchrone Änderung des Eingangssignals vom Ressourcentreiber an. Enthält QO, STATUS und IN.

Daten-Eingänge

- **QI** (BOOL): Qualifizierer für die Ereigniseingänge; typischerweise auf TRUE gesetzt, um die Verarbeitung zu aktivieren.
- **PARAMS** (STRING): Dienstspezifische Parameter zur Konfiguration (z. B. Adressierung oder Protokolleinstellungen).
- **u8SAMember** (USINT): Node-SA-Adresse (Subscriber Address) des Geräts im Bereich 224…239. Initialwert: MI::MI_00.
- **Input** (DataPanel::io::MI::DI::DataPanel_MI_DI_S): Identifikation des digitalen Eingangs (z. B. DigitalInput_5A…8B). Initialwert: Invalid.

Daten-Ausgänge

- **QO** (BOOL): Ausgangsqualifizierer; zeigt an, ob die Ausgangsdaten gültig sind.
- **STATUS** (STRING): Statusmeldung des Dienstes (z. B. Fehler oder Erfolgsmeldung).
- **IN** (BOOL): Der gelesene Binäreingangswert vom angeschlossenen Sensor oder Schalter.

Adapter

Keine Adapter definiert.

Funktionsweise

Der Baustein kommuniziert über das MI-Protokoll mit einem DataPanel-Modul. Nach dem Start wird der Baustein mit INIT initialisiert, wobei die Kommunikationsparameter (PARAMS, u8SAMember, Input) konfiguriert werden. Bei erfolgreicher Initialisierung wird INITO ausgelöst. Anschließend kann über REQ der aktuelle Zustand des konfigurierten digitalen Eingangs abgefragt werden; die Antwort wird über CNF mit dem Wert IN geliefert. Zusätzlich kann der Baustein asynchrone Ereignisse (IND) empfangen, wenn sich der Eingangszustand ohne explizite Anforderung ändert (z. B. durch Interrupts der Hardware). Der Ausgang QO zeigt die Gültigkeit der Daten an, STATUS liefert diagnostische Informationen.

Technische Besonderheiten

- Der Baustein ist als Service-Interface-FB (SIFB) realisiert und erwartet eine hardwarenahe Implementierung in der 4diac-Runtime.
- Die Eingangskonfiguration (Input) verwendet einen benutzerdefinierten Strukturtyp `DataPanel_MI_DI_S`, der die genaue Kanalzuordnung ermöglicht.
- Der Parameter `u8SAMember` ist auf Werte 224..239 beschränkt; dies entspricht typischen SA-Adressen für MI-basierte Subsysteme.
- Die Initialisierung kann nur mit gültigen Parametern erfolgen; bei ungültigem Input (z. B. Invalid) schlägt die Initialisierung fehl.

Zustandsübersicht

Eine explizite Zustandsmaschine ist im XML nicht hinterlegt. Typischerweise durchläuft der Baustein folgende Zustände:

1. **IDLE** – nach dem Start, wartet auf INIT.
2. **INIT** – Initialisierung wird durchgeführt, Konfiguration des Hardwarekanals.
3. **OPERATE** – Bereit für REQ und Empfang von IND-Ereignissen.
4. **ERROR** – bei fehlgeschlagener Initialisierung oder Kommunikationsfehlern. Alternative Zustände können je nach Implementierung des Kommunikationstreibers abweichen.

Anwendungsszenarien

- **Feldgeräteanbindung**: Ein binärer Näherungssensor (npn) wird an einen DataPanel-Eingang angeschlossen. Der Baustein liest den Schaltzustand aus und stellt ihn in der Steuerungsapplikation zur Verfügung.

- **Schalterabfrage**: Ein handbetätigter Schalter (Ground Switching) wird über den FB überwacht; Änderungen werden asynchron über IND gemeldet.
- **Diagnose**: Über STATUS können Fehler wie Drahtbruch oder Adresskonflikte erkannt werden.

Vergleich mit ähnlichen Bausteinen

Ähnliche Bausteine wie `DataPanel_MI_DI_Xn` (für andere Eingangsgruppen) oder `DataPanel_MI_DO_Xn` (Ausgänge) teilen das gleiche Interface, unterscheiden sich jedoch in der Anzahl der Kanäle oder der Datenrichtung. Der vorliegende Baustein fokussiert auf einen einzelnen digitalen Eingang und eignet sich für punktgenaue Überwachung.

Fazit

Der Funktionsblock `DataPanel_MI_IXn` bietet eine standardisierte Schnittstelle zum Auslesen eines Binäreingangs in DataPanel-Systemen über das MI-Protokoll. Durch die serviceorientierte Struktur (INIT/REQ/IND) lässt er sich flexibel in ereignisgesteuerte Automatisierungslösungen integrieren. Die Konfiguration über dedizierte Parameter sorgt für eine einfache und robuste Anbindung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

DQ

DataPanel_MI_DO

• – ◦ • – ◦ • – ◦ •

Einleitung

DataPanel_MI_DO ist eine globale Konstantendefinition für DataPanel-Module mit mittlerer I/O-Dichte für digitale Ausgänge (Digital Output). Die Konstante definiert eine strukturierte Zuordnung von Pins für verschiedene digitale Ausgangskanäle und Stromversorgungsports.

Schnittstellenstruktur

Ereignis-Eingänge

Nicht vorhanden - Es handelt sich um eine globale Konstantendefinition

Ereignis-Ausgänge

Nicht vorhanden - Es handelt sich um eine globale Konstantendefinition

Daten-Eingänge

Nicht vorhanden - Es handelt sich um eine globale Konstantendefinition

Daten-Ausgänge

Nicht vorhanden - Es handelt sich um eine globale Konstantendefinition

Adapter

Nicht vorhanden - Es handelt sich um eine globale Konstantendefinition

Funktionsweise

Die globale Konstante DataPanel_MI_DO definiert feste Pin-Zuordnungen für ein DataPanel-Modul mit mittlerer I/O-Dichte. Jede Konstante repräsentiert einen spezifischen Ausgangskanal oder Stromversorgungsport und weist diesem einen festen Pin-Wert zu. Die Struktur basiert auf dem Datentyp `DataPanel_MI_DO_S` mit einem Pin-Feld.

Technische Besonderheiten

- Definiert 16 digitale Ausgangskanäle (1A-8B) mit Pin-Zuordnungen 1-16

- Enthält 4 Stromversorgungsports (Input_Power_Port_5-8) mit Pin-Zuordnungen 17-20
- Bereitstellung einer Invalid-Konstante mit Pin-Wert 255 für Fehlerbehandlung
- Strukturierte Organisation in Paaren (A/B) für bessere Übersichtlichkeit
- Feste Pin-Zuordnungen garantieren konsistente Hardware-Ansteuerung

Zustandsübersicht

Nicht zutreffend - Es handelt sich um statische Konstantendefinitionen

Anwendungsszenarien

- Hardware-Konfiguration von DataPanel-Modulen mit mittlerer I/O-Dichte
- Vereinfachung der Pin-Zuordnung in Steuerungsprogrammen
- Wiederverwendbare Konstantendefinitionen in größeren Automatisierungsprojekten
- Hardware-abstrakte Programmierung durch symbolische Pin-Namen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einzelnen Pin-Definitionen bietet diese globale Konstante:

- Zentrale Verwaltung aller Pin-Zuordnungen
- Typisierte Zugriffsmöglichkeiten durch strukturierte Datentypen
- Bessere Wartbarkeit und Lesbarkeit des Codes
- Konsistente Namenskonventionen über das gesamte Projekt

Fazit

DataPanel_MI_DO bietet eine effiziente und strukturierte Möglichkeit, die Pin-Zuordnungen für Data-Panel-Module mit mittlerer digitaler Ausgangsdichte zu verwalten. Durch die Verwendung globaler Konstanten wird die Code-Wartbarkeit verbessert und Hardware-Änderungen können zentral verwaltet werden.

DataPanel_MI_DO_S

DataPanel Modules with medium IO density DO Struct

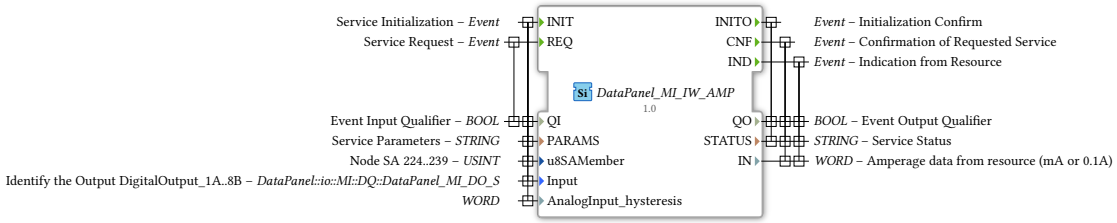
Structure

Name	Type	Initial Value
Pin	USINT	255

🛠️ Zugehörige Übungen

- Funk_IX_TO_DataPanel_MI_QX
- Funk_IXA_TO_DataPanel_MI_QXA

DataPanel_MI_IW_AMP



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock DataPanel_MI_IW_AMP ist ein Eingangs-Schnittstellenbaustein (Service Interface Function Block) für die Erfassung der Stromrückmeldung (Amperage) eines einzelnen Kanals. Er dient zur Integration von Strommesswerten in die 4diac-IDE und ermöglicht die Kommunikation mit einer übergeordneten Steuerung oder einem Datenpanel. Der Baustein initialisiert die hardwarenahe Kommunikation und liefert bei jeder Anforderung den aktuellen Messwert als WORD aus.

Schnittstellenstruktur

Ereignis-Eingänge

- `INIT (EInit)` – Initialisierung des Dienstes. Wird ausgelöst, um den FB zu konfigurieren und die Verbindung zur Hardware herzustellen. Mit diesem Ereignis werden die Parameter QI, PARAMS, u8SAMember, Input und AnalogInput_hysteresis übernommen.
- `REQ (Event)` – Dienstanforderung. Löst eine neue Messwertabfrage aus. Der FB antwortet mit dem Ereignis CNF und liefert den aktuellen Amperage-Wert auf dem Ausgang IN.

Ereignis-Ausgänge

- `INITO (EInit)` – Bestätigung der Initialisierung. Wird nach erfolgreicher Initialisierung gesendet. Gibt den Status QO und STATUS aus.
- `CNF (Event)` – Bestätigung der Dienstanforderung. Wird als Antwort auf REQ gesendet und enthält den aktuellen Messwert IN sowie die Statusinformationen QO und STATUS.
- `IND (Event)` – Indikation von der Ressource. Wird bei asynchronen Ereignissen von der Hardware (z. B. spontane Wertänderung) ausgelöst. Liefert ebenfalls den aktuellen Messwert IN.

Daten-Eingänge

- `QI (BOOL)` – Eingangsqualifizierer. Steuert, ob der FB aktiv ist (TRUE) oder nicht (FALSE).
- `PARAMS (STRING)` – Dienstparameter. Enthält Konfigurationsdaten für die Initialisierung, z. B. Busschnittstelle oder Adressierung.
- `u8SAMember (USINT)` – Knoten-SA (Source Address) im Bereich 224..239. Definiert die eindeutige Adresse des angeschlossenen Geräts. Initialwert: `MI::MI_00`.
- `Input (DataPanel_MI_DO_S)` – Identifiziert den digitalen Ausgang, dessen Strom gemessen werden soll (z. B. DigitalOutput_1A..8B). Initialwert: `Invalid`.
- `AnalogInput_hysteresis (WORD)` – Hysterese für den analogen Eingang. Kann zur Vermeidung von Flanken bei geringen Schwankungen genutzt werden.

Daten-Ausgänge

- `QO (BOOL)` – Ausgangsqualifizierer. Zeigt an, ob der FB korrekt arbeitet (TRUE) oder ein Fehler vorliegt (FALSE).
- `STATUS (STRING)` – Dienststatus. Enthält Fehlermeldungen oder Hinweise zum Betriebszustand.
- `IN (WORD)` – Gemessener Stromwert in Einheiten von mA oder 0,1 A (abhängig von der Konfiguration). Liefert den aktuellen Amperage-Wert des ausgewählten Kanals.

Adapter

Keine Adapter definiert.

Funktionsweise

Der FB agiert als Dienst-Schnittstelle zwischen der Applikation und der Hardwareebene. Bei Auslösen von `INIT` werden die Konfigurationsparameter (Adresse, Kanalzuordnung, Hysterese) an die Hardware übergeben. Nach erfolgreicher Initialisierung wird `INITO` gesendet. Ein `REQ`-Ereignis löst eine sofortige Abfrage des Strommesswerts aus, der auf dem Ausgang `IN` bereitgestellt und mit `CNF` bestätigt wird. Zusätzlich kann die Hardware bei Änderungen ein `IND`-Ereignis generieren, um die Applikation asynchron zu informieren.

Die Werte von `QI` und `QO` steuern den Aktivitätszustand: Nur wenn `QI`=TRUE und `QO`=TRUE kann der FB Daten liefern. Bei einem Fehler wird `QO` auf FALSE gesetzt und eine Fehlermeldung in `STATUS` hinterlegt.

Technische Besonderheiten

- Der FB ist als generischer Dienst-Schnittstellenbaustein konzipiert und erfordert eine hardwareabhängige Implementierung der Servicefunktionen.
- Die Adressierung über `u8SAMember` folgt einem vordefinierten Konstanten-Set (z. B. `MI::MI_00` … `MI::MI_15`), was eine einfache Konfiguration mehrerer Kanäle ermöglicht.
- Der Ausgangstyp `DataPanel_MI_DO_S` dient zur Auswahl eines spezifischen digitalen Ausgangs; der FB wird nur dann korrekt arbeiten, wenn dieser Wert gültig ist (nicht `Invalid`).
- Die Hysterese (`AnalogInput_hysteresis`) kann verwendet werden, um Messrauschen zu unterdrücken und unnötige Ereignisse zu vermeiden.
- Die Amperage-Ausgabe erfolgt als WORD, je nach Konfiguration in mA (z. B. 0–65535 mA) oder in 0,1 A-Schritten.

Zustandsübersicht

Der FB durchläuft typischerweise folgende Zustände:

1. **IDLE** – Ruhezustand nach dem Start, wartet auf INIT.
2. **INIT** – Initialisierung wird durchgeführt; nach Erfolg Übergang zu OPERATIONAL, dann INITO.
3. **OPERATIONAL** – Betriebsbereit; REQ-Ereignisse werden bearbeitet, Messwerte via CNF geliefert; asynchrone IND möglich.
4. **ERROR** – Fehlerzustand (z. B. Kommunikationsfehler, ungültige Parameter); QO = FALSE, STATUS enthält Fehlertext; Rücksetzung nur durch erneutes INIT.

Eine detaillierte Zustandsmaschine ist in der Implementierung des Dienstes hinterlegt.

Anwendungsszenarien

- **Stromüberwachung einzelner Ausgänge** in einer Maschinensteuerung, z. B. zur Erkennung von Laständerungen oder Defekten.
- **Einbindung von Analog-Eingangsmodulen** eines Datenpanels, um pro Kanal die Stromaufnahme zu messen.
- **Parametrierung über INIT** ermöglicht den flexiblen Einsatz in verschiedenen Hardware-Konfigurationen.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem reinen Digital-Eingangs-FB bietet dieser Baustein eine analoge Messwertrückmeldung (Amperage), die mehr Informationen über den Lastzustand liefert. Ähnliche Bausteine für Spannungs- oder Temperaturmessung nutzen ein vergleichbares Interface-Schema (INIT/REQ/IND) mit anderen physikalischen Einheiten. Der Vorteil von `DataPanel_MI_IW_AMP` liegt in der spezifischen Optimierung für Strommessung und der Integration in die DataPanel-Familie.

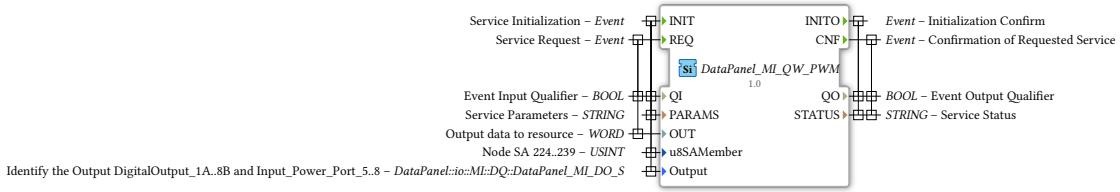
Fazit

Der Funktionsblock `DataPanel_MI_IW_AMP` stellt eine standardisierte und flexible Schnittstelle zur Erfassung von Stromwerten in Automatisierungssystemen dar. Durch die klar definierten Ereignisse und Parameter lässt er sich einfach in 4diac-Projekte einbinden und bietet sowohl synchrone als auch asynchrone Benachrichtigungen über Messwertänderungen. Die detaillierte Konfiguration mittels Adresse, Kanal und Hysterese ermöglicht eine anpassungsfähige Nutzung in unterschiedlichen Hardware-Umgebungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

DataPanel_MI_QW_PWM



Output service interface function block for PWM output data (Data Mode 2)

Interface

Event Inputs

Name	Comment	With
INIT	Service Initialization	QI, PARAMS, u8SAMember, Output
REQ	Service Request	QI, OUT

Event Outputs

Name	Comment	With
INITO	Initialization Confirm	QO, STATUS
CNF	Confirmation of Requested Service	QO, STATUS

Input Vars

Name	Type	Initial Value	Comment
QI	BOOL		Event Input Qualifier
PARAMS	STRING		Service Parameters
OUT	WORD		Output data to resource
u8SAMember	USINT	MI::MI_00	Node SA 224..239
Output	DataPanel::io::MI::DQ::DataPanel_MI_DO_S	Invalid	Identify the Output DigitalOutput_1A..8B and Input_Power_Port_5..8

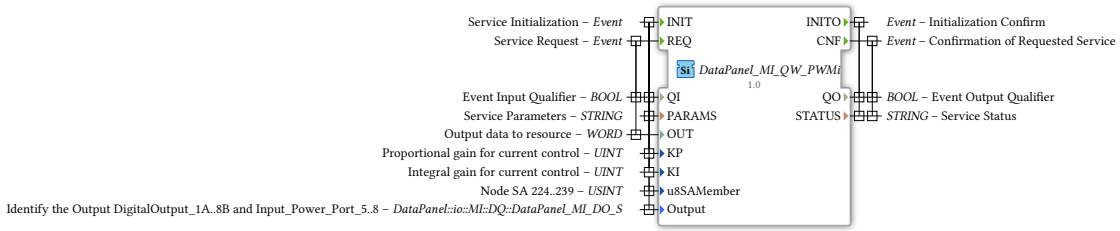
Output Vars

Name	Type	Comment
QO	BOOL	Event Output Qualifier
STATUS	STRING	Service Status

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Das PWM-Signal & Infografik auf ms-muc-docs.de

DataPanel_MI_QW_PWMi



Output service interface function block for Current Controlled PWM output data (Current Mode 4)

Interface

Event Inputs

Name	Comment	With
INIT	Service Initialization	QI, PARAMS, u8SAMember, Output, KP, KI
REQ	Service Request	QI, OUT, KP, KI

Event Outputs

Name	Comment	With
INITO	Initialization Confirm	QO, STATUS
CNF	Confirmation of Requested Service	QO, STATUS

Input Vars

Name	Type	Initial Value	Comment
QI	BOOL		Event Input Qualifier
PARAMS	STRING		Service Parameters
OUT	WORD		Output data to resource
KP	UINT	70	Proportional gain for current control
KI	UINT	30	Integral gain for current control
u8SAMember	USINT	MI::MI_00	Node SA 224..239
Output	DataPanel::io::MI::DQ::DataPanel_MI_DO_S	InvalidS	Identify the Output DigitalOutput_1A..8B and Input_Power_Port_5..8

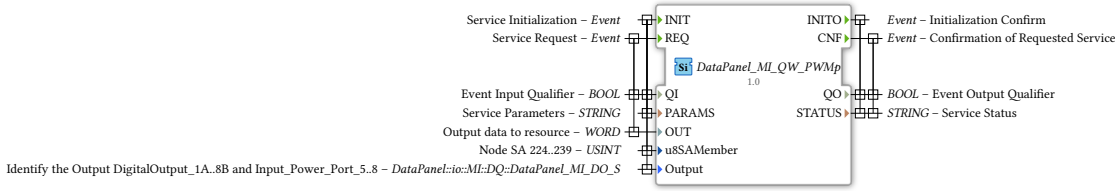
Output Vars

Name	Type	Comment
QO	BOOL	Event Output Qualifier
STATUS	STRING	Service Status

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Das PWM-Signal & Infografik auf ms-muc-docs.de

DataPanel_MI_QW_PWMp



Output service interface function block for PWM output data (Percent Mode 3)

Interface

Event Inputs

Name	Comment	With
INIT	Service Initialization	QI, PARAMS, u8SAMember, Output
REQ	Service Request	QI, OUT

Event Outputs

Name	Comment	With
INITO	Initialization Confirm	QO, STATUS
CNF	Confirmation of Requested Service	QO, STATUS

Input Vars

Name	Type	Initial Value	Comment

QI	BOOL		Event Input Qualifier
PARAMS	STRING		Service Parameters
OUT	WORD		Output data to resource
u8SAMember	USINT	MI::MI_00	Node SA 224..239
Output	DataPanel::io::MI::DQ::DataPanel_MI_DO_S	Invalid	Identify the Output DigitalOutput_1A..8B and Input_Power_Port_5..8

Output Vars

Name	Type	Comment
QO	BOOL	Event Output Qualifier
STATUS	STRING	Service Status

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

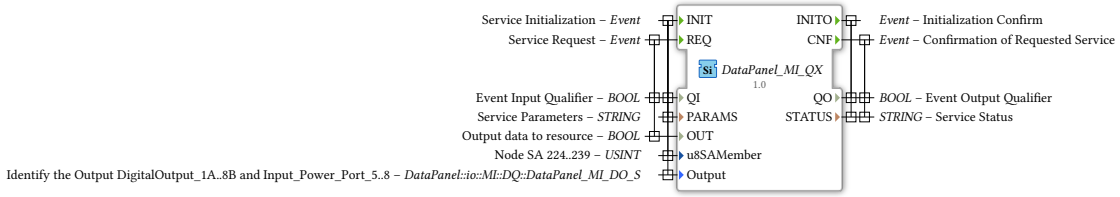
- 🌐 Das PWM-Signal & Infografik auf ms-muc-docs.de

DataPanel_MI_QX

• – ○ • – ○ • – ○ •

Einleitung

Der DataPanel_MI_QX ist ein Service Interface Function Block für die Ausgabe von booleschen Daten an eine Ressource. Dieser Baustein dient als Schnittstelle zwischen der Steuerungslogik und den physischen Ausgängen eines DataPanel MI-Systems und ermöglicht die Konfiguration und Steuerung von digitalen Ausgängen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierungsereignis
- **REQ**: Service-Anfrageereignis

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
- **CNF**: Bestätigung der angeforderten Service-Operation

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangsqualifikator
- **PARAMS** (STRING): Service-Parameter
- **OUT** (BOOL): Ausgabedaten zur Ressource
- **u8SAMember** (USINT): Node SA 224..239, Initialwert: MI::MI_00
- **Output** (DataPanel_MI_DO_S): Identifiziert den Ausgang (DigitalOutput_1A..8B und Input_Power_Port_5..8), Initialwert: Invalid

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifikator
- **STATUS** (STRING): Service-Status

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der DataPanel_MI_QX Function Block verwaltet die Kommunikation mit den digitalen Ausgängen eines DataPanel MI-Systems. Bei der Initialisierung (INIT-Ereignis) werden die Service-Parameter konfiguriert und die Hardware-Schnittstelle eingerichtet. Das REQ-Ereignis triggert die Ausgabe der booleschen Daten an die spezifizierte Ressource. Der Baustein bestätigt sowohl Initialisierungs- als auch Service-Operationen über die entsprechenden Ausgangsereignisse.

Technische Besonderheiten

- Unterstützt die Konfiguration von SA-Nodes im Bereich 224-239
- Ermöglicht die Identifikation spezifischer digitaler Ausgänge (1A bis 8B) und Power-Ports (5-8)
- Verwendet einen speziellen Datentyp (DataPanel_MI_DO_S) zur Ausgangsidentifikation
- Implementiert eine robuste Fehlerbehandlung durch STATUS-Ausgabe

Zustandsübersicht

Der Function Block durchläuft folgende Zustände:

1. **Nicht initialisiert**: Wartet auf INIT-Ereignis
2. **Initialisiert**: Bereit für Service-Anfragen
3. **Service-Ausführung**: Verarbeitet REQ-Ereignisse
4. **Bestätigung**: Sendet CNF/INITO-Ereignisse mit Statusinformationen

Anwendungsszenarien

- Steuerung von digitalen Ausgängen in Industrieautomationssystemen
- Anbindung von DataPanel MI-Hardwarekomponenten
- Implementierung von sicheren Ausgabeprotokollen
- Integration in größere Steuerungssysteme mit mehreren Ausgabekanälen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Ausgabeblöcken bietet DataPanel_MI_QX erweiterte Funktionalitäten wie:

- Parametrierbare SA-Node-Konfiguration
- Spezifische Hardware-Identifikation für DataPanel MI-Systeme
- Detaillierte Statusrückmeldungen
- Flexible Ausgangskonfiguration über den Output-Parameter

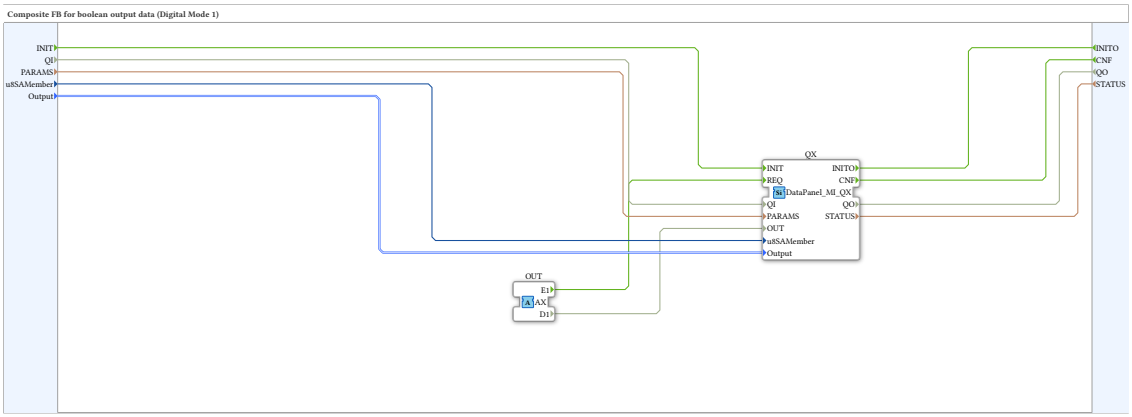
🛠️ Zugehörige Übungen

- Funk_IX_TO_DataPanel_MI_QX
- Uebung_010a4

Fazit

Der DataPanel_MI_QX Function Block stellt eine leistungsstarke und flexible Lösung für die Ansteuerung digitaler Ausgänge in DataPanel MI-Systemen dar. Durch seine umfangreiche Parametrierbarkeit und detaillierte Statusrückmeldungen eignet er sich besonders für komplexe Automatisierungsanwendungen, die eine zuverlässige und konfigurierbare Ausgabeschnittstelle erfordern.

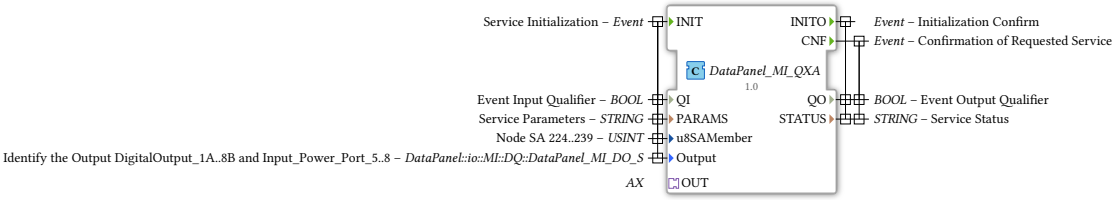
DataPanel_MI_QXA



• – ○ • – ○ • – ○ •

Einleitung

Der DataPanel_MI_QXA ist ein Composite-Funktionsblock zur Steuerung von booleschen Ausgabedaten in Automatisierungssystemen. Dieser Baustein dient als Schnittstelle für digitale Ausgänge und ermöglicht die Konfiguration und Ansteuerung von Ausgangskanälen über standardisierte Service-Parameter.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierungsereignis mit den zugehörigen Daten QI, PARAMS, u8SAMember und Output

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung mit QO und STATUS
- **CNF**: Bestätigung des angeforderten Services mit QO und STATUS

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangsqualifizierer
- **PARAMS** (STRING): Service-Parameter
- **u8SAMember** (USINT): Knoten-SA-Adresse im Bereich 224-239, initialisiert mit MI::MI_00
- **Output** (DataPanel_MI_DO_S): Identifizierung des Ausgangs (DigitalOutput_1A..8B und Input_Power_Port_5..8), initialisiert mit Invalid

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifizierer
- **STATUS** (STRING): Service-Status

Adapter

- **OUT**: Unidirektionaler Adapter vom Typ AX für die Ausgabesteuerung

Funktionsweise

Der Composite-Funktionsblock DataPanel_MI_QXA kapselt die Funktionalität des DataPanel_MI_QX-Bausteins und erweitert diesen um eine Adapter-Schnittstelle. Bei Initialisierung werden die Konfigurationsparameter an den internen QX-Baustein weitergeleitet. Der Adapter OUT ermöglicht die externe Steuerung der Ausgabefunktionen über standardisierte Schnittstellen.

Technische Besonderheiten

- Unterstützt SA-Adressen im Bereich 224-239
- Verwendet spezifische Ausgangsidentifikation für digitale Ausgänge und Power-Ports
- Integrierte Fehlerbehandlung durch Invalid-Initialisierung
- Unidirektionale Adapter-Architektur für saubere Schnittstellentrennung

Zustandsübersicht

Der Baustein durchläuft folgende Zustände:

1. **Initialisierungsphase**: INIT-Ereignis verarbeitet Konfigurationsparameter
2. **Bereitschaftszustand**: Wartet auf Steuerungsanforderungen über Adapter OUT
3. **Aktiver Betrieb**: Verarbeitet Ausgabebefehle und generiert CNF-Bestätigungen

Anwendungsszenarien

- Steuerung digitaler Ausgänge in Industrieautomation
- Anbindung von Power-Port-Steuerungen
- Integration in modulare Steuerungssysteme mit SA-Adressierung
- Einsatz in verteilten Automatisierungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Ausgabebausteinen bietet DataPanel_MI_QXA:

- Erweiterte Parametrierungsmöglichkeiten über STRING-Parameter
- SA-Adressierungsunterstützung für Netzwerkintegration
- Composite-Architektur mit Adapter-Schnittstelle
- Umfassende Statusrückmeldungen

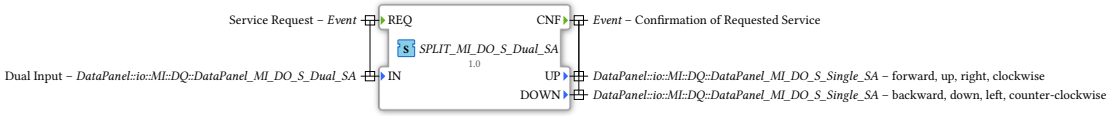
🛠️ Zugehörige Übungen

- Funk_IXA_TO_DataPanel_MI_QXA
- Uebung_010a4_AX

Fazit

Der DataPanel_MI_QXA ist ein leistungsstarker Composite-Funktionsblock für die Steuerung boolescher Ausgabedaten in komplexen Automatisierungsumgebungen. Durch seine modulare Architektur und standardisierten Schnittstellen eignet er sich ideal für den Einsatz in verteilten Steuerungssystemen mit hohen Anforderungen an Flexibilität und Wartbarkeit.

SPLIT_MI_DO_S_Dual_SA



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **SPLIT_MI_DO_S_Dual_SA** dient dazu, einen einzelnen Datenpanel-Eingang vom Typ `DataPanel_MI_DO_S_Dual_SA` in zwei separate Ausgänge vom Typ `DataPanel_MI_DO_S_Single_SA` aufzuteilen. Dabei werden die in der dualen Struktur enthaltenen Antriebsinformationen für eine Vorwärts-/Rückwärtsbewegung (bzw. oben/unten, rechts/links, im/gegen Uhrzeigersinn) auf zwei unabhängige Ausgänge verteilt.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung
REQ	Dienstanforderung: Der Baustein verarbeitet den aktuellen Wert des Dateneingangs IN.

Ereignis-Ausgänge

Ereignis	Beschreibung
CNF	Bestätigung der durchgeführten Verarbeitung. Wird nach der Ausführung des Algorithmus gesendet.

Daten-Eingänge

Name	Typ	Beschreibung
IN	`DataPanel::io::MI::DQ::DataPanel_MI_DO_S_Dual_SA`	Dualer Eingang, der sowohl Vorwärts- (UP) als auch Rückwärts- (DOWN) Informationen trägt.

Daten-Ausgänge

Name	Typ	Beschreibung
UP	`DataPanel::io::MI::DQ::DataPanel_MI_DO_S_Single_SA`	Ausgang für die Vorwärts-, Aufwärts-, Rechts- oder Rechtsdrehrichtung.
DOWN	`DataPanel::io::MI::DQ::DataPanel_MI_DO_S_Single_SA`	Ausgang für die Rückwärts-, Abwärts-, Links- oder Linksdrehrichtung.

Adapter

Keine.

Funktionsweise

Der Baustein wird durch das Ereignis **REQ** aktiviert. Im zugehörigen Algorithmus werden folgende Zuweisungen durchgeführt:

- Der gemeinsame Steuerwert `u8SAMember` der dualen Struktur wird unverändert an beide Ausgänge (`UP.u8SAMember` und `DOWN.u8SAMember`) weitergegeben.
- Die Richtungsinformationen werden getrennt: Der Ausgang `UP.Port` erhält den Wert des Feldes `IN.Up`, und der Ausgang `DOWN.Port` den Wert des Feldes `IN.Down`.
- Nach Ausführung des Algorithmus wird das Bestätigungsereignis **CNF** gesendet.

Technische Besonderheiten

- Der Baustein ist als einfacher Funktionsbaustein (SimpleFB) realisiert, d. h. die Logik wird innerhalb eines einzigen Algorithmus ohne Zustandsmaschine ausgeführt.
- Die verwendeten Datentypen stammen aus dem Paket `DataPanel::io::MI::DQ`. Sie sind als implizite Importe im Baustein deklariert.
- Die Ausgangsdaten sind zum Zeitpunkt des **CNF**-Ereignisses gültig.

Zustandsübersicht

Der Baustein besitzt nur einen einzigen EC-Zustand **REQ**, in dem der Algorithmus ausgeführt und anschließend das Ausgangsereignis **CNF** gesetzt wird. Es gibt keine weiteren Zustände oder Verzweigungen.

Anwendungsszenarien

- Aufteilung eines kombinierten Antriebssignals für einen bidirektionalen Motor in zwei separate Steuerleitungen (z. B. für Drehrichtungssteuerung).
- Trennung von Vorwärts- und Rückwärtssignalen in einer Maschinensteuerung, wenn nachgelagerte Bausteine getrennte Eingänge für jede Bewegungsrichtung erwarten.
- Vorbereitung von Daten für zwei unabhängige Aktoren, die mit demselben Betriebsmodus (z. B. Leistungsstufe) arbeiten, aber unterschiedliche Richtungen ansteuern.

Vergleich mit ähnlichen Bausteinen

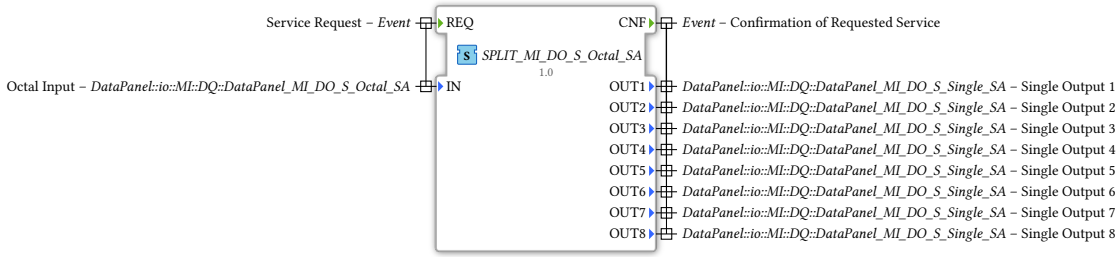
Ein möglicher alternativer Ansatz wäre die Verwendung eines allgemeinen Splitter-Bausteins, der beliebige Verbunddatentypen auftrennt. **SPLIT_MI_DO_S_Dual_SA** ist jedoch speziell auf die genannten

Datentypen zugeschnitten und übernimmt direkt die korrekte Zuordnung der Richtungsfelder, was die Fehleranfälligkeit reduziert und die Lesbarkeit des Steuerungscodes erhöht.

Fazit

SPLIT_MI_DO_S_Dual_SA bietet eine einfache und zuverlässige Möglichkeit, einen dualen Datenpanel-Steuerbefehl in zwei separate Ausgänge für die entgegengesetzten Bewegungsrichtungen aufzuteilen. Durch die klar definierte Schnittstelle und die direkte Verarbeitung aus einem Ereignisaufruf eignet er sich besonders für den Einsatz in modularen Steuerungssystemen, bei denen eine saubere Trennung der Richtungssignale erforderlich ist.

SPLIT_MI_DO_S_Octal_SA



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **SPLIT_MI_DO_S_Octal_SA** dient dazu, einen einzelnen Datenwert vom Typ `DataPanel_MI_DO_S_Octal_SA` in acht separate Ausgänge vom Typ `DataPanel_MI_DO_S_Single_SA` aufzuteilen. Dabei wird der gemeinsame Anteil (`u8SAMember`) unverändert übernommen und die jeweiligen Port-Informationen aus dem oktalen Eingang auf die einzelnen Ausgänge verteilt.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Kommentar
`REQ`	Service-Anforderung; die Verarbeitung wird gestartet, sobald das Ereignis eintritt. Der Benutzer muss den Daten-Eingang `IN` vorher gültig setzen.

Ereignis-Ausgänge

Ereignis	Kommentar
`CNF`	Bestätigung der abgeschlossenen Verarbeitung. Signalisiert, dass alle acht Ausgänge aktualisiert wurden.

Daten-Eingänge

Name	Typ	Kommentar
`IN`	`DataPanel_MI_DO_S_Octal_SA`	Oktaler Eingabewert (Port 1…8 und gemeinsamer `u8SAMember`)

Daten-Ausgänge

Name	Typ	Kommentar
`OUT1`	`DataPanel_MI_DO_S_Single_SA`	Einzelausgang 1
`OUT2`	`DataPanel_MI_DO_S_Single_SA`	Einzelausgang 2
`OUT3`	`DataPanel_MI_DO_S_Single_SA`	Einzelausgang 3

OUT4	DataPanel_MI_DO_S_Single_SA	Einzelausgang 4
OUT5	DataPanel_MI_DO_S_Single_SA	Einzelausgang 5
OUT6	DataPanel_MI_DO_S_Single_SA	Einzelausgang 6
OUT7	DataPanel_MI_DO_S_Single_SA	Einzelausgang 7
OUT8	DataPanel_MI_DO_S_Single_SA	Einzelausgang 8

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Baustein implementiert einen einfachen, ereignisgesteuerten Ablauf:

1. Ein eingehendes REQ-Ereignis löst die Ausführung des Algorithmus REQ aus.
2. Der Algorithmus kopiert das Feld u8SAMember aus dem Eingang IN in jeden der acht Ausgänge (OUT1 … OUT8).
3. Jeder Ausgang erhält zusätzlich den zugehörigen Port-Wert aus dem oktalen Eingang:
   - OUT1.Port := IN.Port1
   - OUT2.Port := IN.Port2
   - …
   - OUT8.Port := IN.Port8
4. Nach Abschluss wird das Ereignis CNF gesendet, das alle acht Ausgänge als gültig kennzeichnet.

Technische Besonderheiten

- Der Baustein ist als **SimpleFB** (einfacher Funktionsblock) realisiert und arbeitet ohne Zustandsmaschine mit mehreren Zuständen – die Verarbeitung erfolgt in einem Schritt.
- Es gibt keine Nebenläufigkeit oder zeitkritische Abläufe; die Aufteilung erfolgt synchron zum REQ-Ereignis.
- Die Typen DataPanel_MI_DO_S_Octal_SA und DataPanel_MI_DO_S_Single_SA müssen im gleichen Daten-Namespace (hier DataPanel::io::MI::DQ) definiert sein.

Zustandsübersicht

Der Baustein besitzt genau einen Zustand REQ:

- **Eintritt** durch REQ-Ereignis.
- **Aktion**: Ausführen des Algorithmus REQ.
- **Ausgang**: Senden des CNF-Ereignisses (alle Ausgänge werden gleichzeitig mitgeführt).

Es gibt keine weiteren Zustände, die auf Verzögerung oder Fehlerbehandlung hinweisen.

Anwendungsszenarien

- Aufteilung eines oktalen Daten-Busses (z.B. eines Moduls mit acht gleichartigen digitalen Ausgängen) in einzelne logische Kanäle.
- Schnittstelle zwischen einem oktalen Sensor-/Aktor-Panel und einer Steuerung, die einzelne Ausgänge separat verarbeitet.
- Vorverarbeitung in der Automatisierungstechnik, wenn ein Datenpaket mehrere Ports enthält, die getrennt weitergereicht werden sollen.

Vergleich mit ähnlichen Bausteinen

Vergleichbare Bausteine existieren für andere Datenstrukturen (z.B. SPLIT_MI_DO_S_Octal_SA für spezifische Panel-Typen). Der grundlegende Mechanismus – Aufteilen eines Verbunddatentyps in seine Bestandteile – ist universell, unterscheidet sich jedoch je nach Datenfeld-Namen und Typ.

Fazit

Der SPLIT_MI_DO_S_Octal_SA ist ein kompakter und klar strukturierter Funktionsblock zur Zerlegung eines oktalen Daten-Pakets in acht Einzelsignale. Er erleichtert die Modularisierung von Automatisierungssoftware, indem er komplexe Datentypen in handhabbare Einheiten aufteilt und die Port-Zuweisung direkt aus der Quellstruktur übernimmt.

const

MI

🎧 Podcast

- Eclipse 4diac 3.0: ST-Interpreter, FBE und 7200 Commits – Der Turbo für verteilte Automatisierung
- Industrie 4.0: Die Magie selbstkonfigurierender Systeme mit fortiss, IEC 61499 und Eclipse 4diac
- From Pyramid to Plug-and-Play: The Rise of Self-Configurable Industrial Automation
- Der E_CTU in der IEC 61499: Ereignisgesteuertes Zählen und warum der Minimalist im Maschinenbau überzeugt
- Der E_PERMIT-Baustein: Der „Türsteher“ für Ereignisse in IEC 61499-Systemen entschlüsselt

📺 Video

- Die große Migration

Einleitung

Die MI Global Constants stellen eine Sammlung von Konstanten für Module mit mittlerer I/O-Dichte dar. Diese Konstantendefinitionen werden in der 4diac-IDE verwendet, um einheitliche und wiederverwendbare Adressierungsinformationen für I/O-Module bereitzustellen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

Keine Adapter vorhanden

Funktionsweise

Bei den MI Global Constants handelt es sich um eine reine Konstantendeklaration ohne aktive Funktionalität. Die Konstanten definieren Adressbereiche für 16 verschiedene I/O-Module (MI_00 bis MI_15) mit jeweils einem eindeutigen numerischen Wert im Bereich von 224 bis 239.

Technische Besonderheiten

- Alle Konstanten sind vom Typ USINT (Unsigned Short Integer)
- Die Werte bilden einen fortlaufenden Bereich von 224 bis 239
- Konstanten sind als global und konstant deklariert
- Package-Zuordnung: DataPanel::io::MI::const

Zustandsübersicht

Keine Zustandsübergänge vorhanden, da es sich um statische Konstantendefinitionen handelt

Anwendungsszenarien

- Adressierung von I/O-Modulen in Steuerungsanwendungen
- Wiederverwendung in verschiedenen Funktionsblöcken

- Zentrale Verwaltung von Moduladressen
- Vereinfachung der Projektkonfiguration

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Konstantendefinitionen bietet MI spezifische Adressen für Module mit mittlerer I/O-Dichte. Andere Konstantensammlungen könnten unterschiedliche Adressbereiche oder Modultypen abdecken.

Fazit

Die MI Global Constants bieten eine praktische und zentrale Möglichkeit, Adressinformationen für I/O-Module zu verwalten. Durch die Verwendung dieser Konstanten wird die Wartbarkeit und Lesbarkeit von Steuerungsprogrammen verbessert, da Änderungen an Moduladressen zentral vorgenommen werden können.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.1.3 Funk

2.1.3.1 🎧 Podcast

- DIN EN 61499-1 Entmystifiziert: Funktionsbausteine, Objektorientierung und verteilte Systeme
- DIN EN 61499-1: Bauklötze für die Industrie 4.0 – So revolutionieren Funktionsbausteine die Automatisierung
- DIN EN 61499-1: Funktionsbausteine und der Paradigmenwechsel in der industriellen Softwareentwicklung
- DIN EN 61499: Industrielle Steuerungen modular und ereignisbasiert mit Funktionsbausteinen meistern – Der ESR-Schalter im Fokus
- EventFBs nach IEC 61499: Legosteine der Automatisierung – So funktionieren Ereignis-Funktionsbausteine

2.1.3.2 🔌 io

📺 Video

- Automation der Zukunft
- Die große Migration
- Digital decisions
- Ideale Dioden
- The Dual Wi Fi Solution

🎧 Podcast

- 4diac IDE: Wie der IEC 61499 Standard die Industrieautomatisierung revolutioniert
- 4diac-Präsentation: Zielgruppen, Struktur und Alleinstellungsmerkmal Schärfen
- Den Software-Drachen zähmen: Industrielle Automatisierung und die Zukunft der Produktion
- Eclipse 4diac: Innovationsmotor Forschung vs. Nutzerbedürfnisse – Was treibt die Entwicklung wirklich voran?
- Software-Drachen zähmen: Warum IEC 61499 die Industrie revolutionieren könnte

---

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 Diode & Halbleitergrundlagen auf ms-muc-docs.de
- 🌐 Smart Switch vs. Relais auf ms-muc-docs.de

DI

📺 Video

- Decoding the datasheet
- Die große Migration
- Die Kunst des Lötens
- Die Welt der Normung
- Digital decisions

🎧 Podcast

- 4diac IDE: Dein „Hello World“ der Automatisierung – Das Blinking Tutorial Lokal
- 4diac IDE: Dein Open-Source-Werkzeugkasten für verteilte Industrieautomatisierung nach IEC 61499
- 4diac IDE: Wie der IEC 61499 Standard die Industrieautomatisierung revolutioniert
- 4diac-Präsentation: Zielgruppen, Struktur und Alleinstellungsmerkmal Schärfen
- Den Software-Drachen zähmen: Industrielle Automatisierung und die Zukunft der Produktion

---

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Funk_DI

• – ◦ • – ◦ • – ◦ •

Einleitung

Funk_DI ist ein GlobalConstants-Paket für die 4diac-IDE, das Konstanten für digitale Eingänge in Funksystemen definiert. Das Paket stellt vordefinierte Pin-Konfigurationen für verschiedene digitale Eingangssignale zur Verfügung, die in Steuerungsanwendungen verwendet werden können.

Schnittstellenstruktur

Da es sich bei Funk_DI um ein GlobalConstants-Paket handelt, verfügt es über keine klassischen Funktionsblock-Schnittstellen. Stattdessen werden globale Konstanten bereitgestellt.

Ereignis-Eingänge

Nicht verfügbar - GlobalConstants-Paket

Ereignis-Ausgänge

Nicht verfügbar - GlobalConstants-Paket

Daten-Eingänge

Nicht verfügbar - GlobalConstants-Paket

Daten-Ausgänge

Nicht verfügbar - GlobalConstants-Paket

Adapter

Nicht verfügbar - GlobalConstants-Paket

Funktionsweise

Das Funk_DI-Paket definiert eine Sammlung von globalen Konstanten vom Typ `Funk_DI_S`, die digitale Eingangspins repräsentieren. Jede Konstante enthält eine Pin-Nummer, die den physischen oder logischen Eingang identifiziert. Die Konstanten können in anderen Funktionsblöcken verwendet werden, um auf spezifische digitale Eingänge zu verweisen.

Technische Besonderheiten

- **Pin-Zuordnungen**: Die Pin-Nummern folgen einem spezifischen Schema:
  - Standard-Eingänge: Pin 1-15
  - Spezielle Funktionen: Pin 0 (STOP), Pin 25 (START)
  - Systemfunktionen: Pin −1 (STATUS), Pin −6 (COUNTER)
  - Ungültiger Wert: Pin 127 (Invalid)
- **Datentyp**: Alle Konstanten verwenden den strukturierten Datentyp `Funk_DI_S`
- **Paketstruktur**: Organisiert im Namespace `Funk::io::DI`

Zustandsübersicht

Da es sich um ein Konstanten-Paket handelt, gibt es keine Zustandsübergänge. Alle Werte sind zur Kompilierzeit fest definiert und unveränderlich.

Anwendungsszenarien

- **Konfiguration digitaler Eingänge**: Vereinfachte Konfiguration von digitalen Eingängen in Steuerungsanwendungen
- **Standardisierung**: Sicherstellung konsistenter Pin-Zuordnungen über verschiedene Projekte hinweg
- **Wartbarkeit**: Zentrale Verwaltung aller digitaler Eingangskonfigurationen
- **Funksysteme**: Speziell für Anwendungen in drahtlosen Steuerungssystemen entwickelt

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen digitalen Eingangs-Konstantenpaketen bietet Funk_DI:

- Spezifische Pin-Zuordnungen für Funksysteme
- Vordefinierte Konstanten für spezielle Funktionen (START, STOP, STATUS, COUNTER)
- Strukturierte Organisation im IO-Bereich des Funk-Namespace

Fazit

Das Funk_DI GlobalConstants-Paket bietet eine praktische Sammlung vordefinierter Konstanten für digitale Eingänge in Funksystemen. Durch die standardisierten Pin-Zuordnungen und die klare Namensgebung ermöglicht es eine effiziente und wartungsfreundliche Konfiguration von Steuerungsanwendungen. Die speziellen Funktionen wie START, STOP und STATUS machen es besonders geeignet für industrielle Steuerungsanwendungen mit drahtlosen Komponenten.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Funk_DI_S

Funk_DI Struct

Structure

Name	Type	Initial Value
Pin	SINT	127

🛠️ Zugehörige Übungen

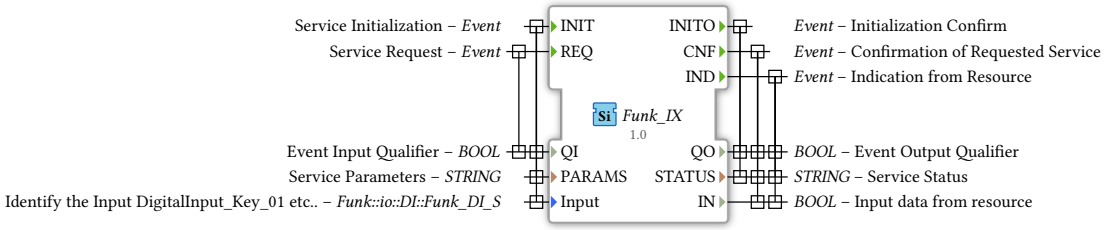
- Funk_IX_TO_DataPanel_MI_QX
- Funk_IXA_TO_DataPanel_MI_QXA

Funk_IX

• − ◦ • − ◦ • − ◦ •

Einleitung

Der Funk_IX ist ein Service Interface Function Block für boolesche Eingabedaten. Er dient als Schnittstelle zur Kommunikation mit digitalen Eingängen in verteilten Automatisierungssystemen und ermöglicht die Initialisierung, Abfrage und asynchrone Benachrichtigung von Eingangssignalen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service Initialisierung
  - Verknüpft mit: QI, PARAMS, Input
- **REQ**: Service Request
  - Verknüpft mit: QI

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
  - Verknüpft mit: QO, STATUS
- **CNF**: Bestätigung des angeforderten Services
  - Verknüpft mit: QO, STATUS, IN
- **IND**: Indikation von der Ressource
  - Verknüpft mit: QO, STATUS, IN

Daten-Eingänge

- **QI**: Event Input Qualifier (BOOL) - Qualifizierer für Ereigniseingänge
- **PARAMS**: Service Parameters (STRING) - Serviceparameter
- **Input**: Identifiziert den Eingang (Funk::io::DI::Funk_DI_S) - Identifiziert den digitalen Eingang (z.B. DigitalInput_Key_01)

Daten-Ausgänge

- **QO**: Event Output Qualifier (BOOL) - Qualifizierer für Ereignisausgänge
- **STATUS**: Service Status (STRING) - Servicestatus
- **IN**: Input data from resource (BOOL) - Eingangsdaten von der Ressource

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Funk_IX Function Block implementiert ein Service Interface Pattern für digitale Eingänge. Bei der Initialisierung (INIT-Event) werden die Service-Parameter konfiguriert und der spezifische digitale Eingang identifiziert. Über das REQ-Event können Eingangswerte abgefragt werden, was zu einer CNF-Antwort mit dem aktuellen Eingangswert führt. Zusätzlich unterstützt der Baustein asynchrone IND-Ereignisse, die bei Zustandsänderungen des Eingangssignals ausgelöst werden.

Technische Besonderheiten

- Verwendet den speziellen Datentyp `Funk::io::DI::Funk_DI_S` zur Identifikation digitaler Eingänge
- Initialwert für Input ist auf „Invalid“ gesetzt
- Unterstützt sowohl synchrone Abfragen (REQ/CNF) als auch asynchrone Benachrichtigungen (IND)
- Enthält umfangreiche Statusrückmeldungen über den STATUS-Ausgang

Zustandsübersicht

Der Function Block durchläuft folgende Zustände:

1. **Nicht initialisiert**: Vor INIT-Event
2. **Initialisiert**: Nach erfolgreicher INIT/INITO-Sequenz
3. **Abfragebereit**: Bereit für REQ/CNF-Zyklen
4. **Benachrichtigungsbereit**: Bereit für asynchrone IND-Ereignisse

Anwendungsszenarien

- Überwachung von digitalen Sensoren und Schaltern
- Integration in verteilte Steuerungssysteme
- Anbindung von Feldgeräten in Industrieautomation
- Zustandsüberwachung von Maschinen und Anlagen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Eingangsbausteinen bietet Funk_IX erweiterte Service-Funktionalität mit Parametrierbarkeit, Statusrückmeldungen und asynchronen Benachrichtigungen. Die spezielle Eingangsidentifikation ermöglicht eine präzise Adressierung in komplexen Systemen.

🛠️ Zugehörige Übungen

- Funk_IX_TO_DataPanel_MI_QX

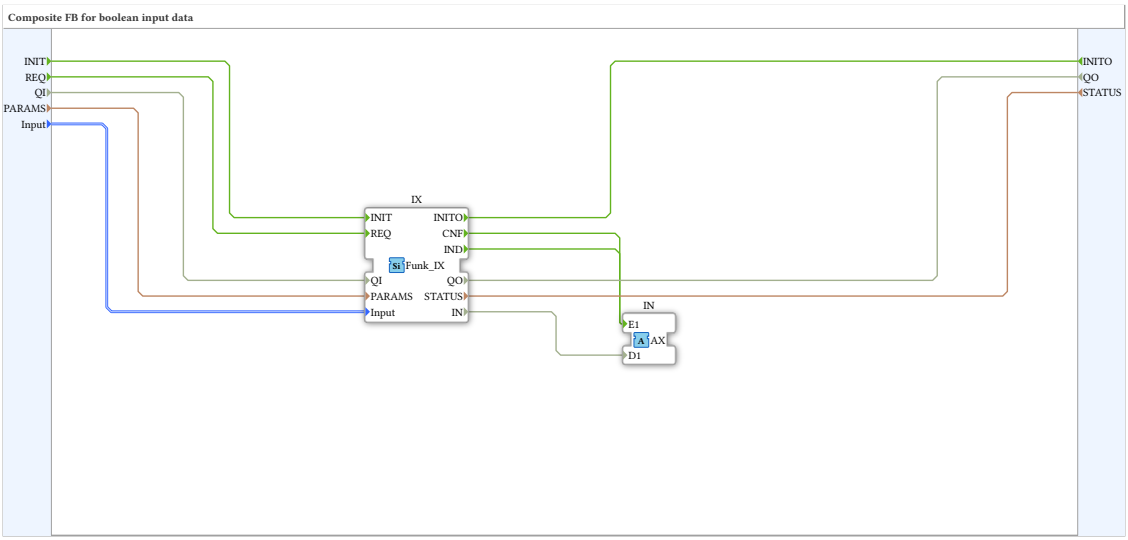
Fazit

Der Funk_IX Function Block stellt eine leistungsfähige und flexible Lösung für die Integration digitaler Eingänge in 4diac-basierte Automatisierungssysteme dar. Seine Service-orientierte Architektur mit Initialisierungs-, Abfrage- und Benachrichtigungsfunktionen macht ihn besonders geeignet für anspruchsvolle industrielle Anwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

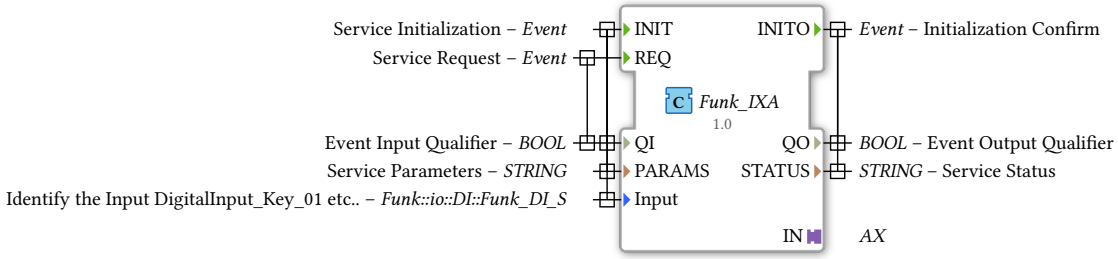
Funk_IXA



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **Funk_IXA** ist ein Composite-Funktionsblock zur Verarbeitung von booleschen Eingangsdaten. Er dient als Schnittstelle für digitale Eingänge und ermöglicht die Initialisierung und Abfrage von Eingangssignalen über eine standardisierte Schnittstelle.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung (mit QI, PARAMS, Input)
- **REQ**: Service-Anfrage (mit QI)

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung (mit QO, STATUS)

Daten-Eingänge

- **QI**: Ereignis-Eingangsqualifizierer (BOOL)
- **PARAMS**: Service-Parameter (STRING)
- **Input**: Identifizierung des digitalen Eingangs (Funk_DI_S, Initialwert: Invalid)

Daten-Ausgänge

- **QO**: Ereignis-Ausgangsqualifizierer (BOOL)
- **STATUS**: Service-Status (STRING)

Adapter

- **IN**: Unidirektionaler Adapter vom Typ AX

Funktionsweise

Der Funk_IXA-Block fungiert als Wrapper für den zugrundeliegenden Funk_IX-Block. Bei der INIT-Initialisierung werden die Parameter QI, PARAMS und Input an den internen IX-Block weitergeleitet. Die REQ-Anfrage triggert die Abfrage des aktuellen Eingangszustands. Der Adapter IN ermöglicht die Kommunikation mit externen Systemen.

Technische Besonderheiten

- Verwendet den spezifischen Datentyp Funk_DI_S zur Identifikation digitaler Eingänge
- Implementiert eine Invalid-Initialisierung für den Input-Parameter
- Unterstützt String-basierte Parameterübergabe für flexible Konfiguration
- Bietet Statusrückmeldungen über den STATUS-Ausgang

Zustandsübersicht

Der Block durchläuft folgende Zustände:

1. **Nicht initialisiert**: Vor INIT-Aufruf
2. **Initialisiert**: Nach erfolgreicher INIT-Verarbeitung
3. **Bereit**: Kann REQ-Anfragen verarbeiten
4. **Aktiv**: Während der REQ-Verarbeitung

Anwendungsszenarien

- Anbindung digitaler Eingänge in Automatisierungssystemen
- Integration in verteilte Steuerungssysteme
- Verwendung in landwirtschaftlichen Maschinensteuerungen
- Schnittstelle für Sensor-Datenerfassung

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Eingangsblöcken bietet Funk_IXA:

- Erweiterte Parameterisierungsmöglichkeiten
- Statusrückmeldungen
- Adapter-Schnittstelle für erweiterte Kommunikation
- Typisierte Eingangsidentifikation

🛠️ Zugehörige Übungen

- Funk_IXA_TO_DataPanel_MI_QXA

Fazit

Der Funk_IXA-Funktionsblock stellt eine robuste und flexible Lösung für die Verarbeitung digitaler Eingangssignale dar. Durch seine Composite-Struktur und die Verwendung standardisierter Schnittstellen eignet er sich besonders für komplexe Automatisierungsanwendungen, die zuverlässige Datenerfassung und Statusüberwachung erfordern.

2.1.4 MyLib (test_AX)

2.1.4.1 sys

🎧 Podcast

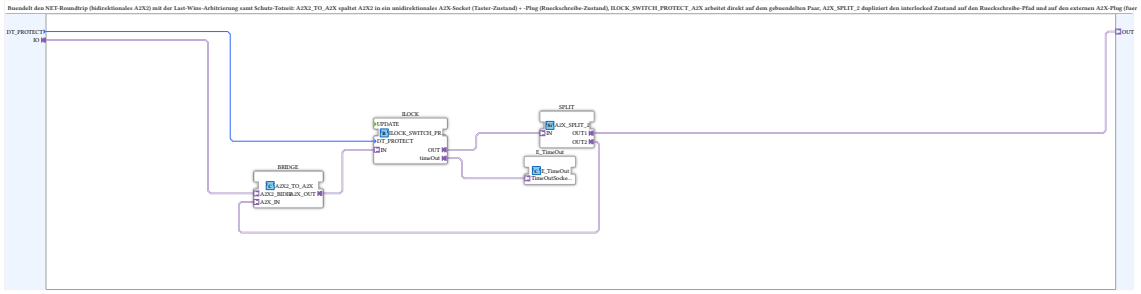
- Eclipse 4diac: Ihr Fahrrad für die Industrie 4.0 – Software-Aufwand meistern und verteilte Systeme steuern
- IEC 61499: Die Zukunft der industriellen Automatisierung und verteilten Systeme
- Industrie 4.0: Die Magie selbstkonfigurierender Systeme mit fortiss, IEC 61499 und Eclipse 4diac
- Eclipse 4diac: Revolutionizing Industrial Control with Open-Source Cyber-Physical Systems
- Der E_PERMIT-Baustein: Der „Türsteher“ für Ereignisse in IEC 61499-Systemen entschlüsselt

---

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

A2X2_ILOCK_SWITCH_PROTECT



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock **A2X2_ILOCK_SWITCH_PROTECT** ist eine Subapp, die einen bidirektionalen A2X2-Kommunikationskanal (z. B. für Tasterzustände und Rückschreibewerte) mit einer Last-Wins-Arbitrierung und einer optionalen Schutz-Totzeit kombiniert. Er kapselt die notwendigen Grundbausteine (Bridge, Interlock, Split) in einer wiederverwendbaren Composite-Struktur, sodass auf der Ressourcen-Ebene nur der gebündelte A2X2-Anschluss und der ausgehende A2X-Plug für physische Ausgänge sichtbar sind. Das ermöglicht eine saubere Trennung von Netzwerkkommunikation und lokaler Signalverarbeitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine (Ereignissteuerung erfolgt über die internen Adapterverbindungen)

Ereignis-Ausgänge

- Keine (das Zeitverhalten wird über den Adapter `timeOut` des internen ILOCK-Bausteins abgewickelt)

Daten-Eingänge

Name	Typ	Initialwert	Kommentar
DT_PROTECT	TIME	T#50ms	Schutz-Totzeit vor einem Richtungswechsel

Daten-Ausgänge

- Keine

Adapter

Richtung	Name	Typ	Kommentar
Socket	IO	adapter::types::bidirectional::A2X2	Gebündelter NET-Roundtrip: Taster-Zustand rein, Rückschreibe-Zustand raus
Plug	OUT	adapter::types::unidirectional::A2X	Interlocked UP/DOWN-Zustand, für physische Ausgänge

Funktionsweise

Die Subapp verarbeitet den gesamten Datenfluss von einem bidirektionalen A2X2-Kanal zur lokalen Interlock-Logik und zurück. Die interne Topologie besteht aus vier Funktionsbausteinen:

1. **BRIDGE** (A2X2_TO_A2X): Zerlegt das bidirektionale A2X2-Signal in zwei unidirektionale Pfade – den hereinkommenden Tasterzustand (A2X_OUT) und den zurückschreibenden Zustand (A2X_IN).
2. **ILOCK** (ILOCK_SWITCH_PROTECT_A2X): Implementiert die Last-Wins-Arbitrierung. Es wertet den ankommenden Tasterzustand aus und erzeugt einen stabilen Ausgangszustand (UP/DOWN), wobei ein Richtungswechsel erst nach Ablauf der durch DT_PROTECT definierten Totzeit erlaubt wird. Ein interner Timeout-Adapter signalisiert das Ende der Totzeit.
3. **E_TimeOut** (Ereignis-FB E_TimeOut): Steuert das Zeitverhalten des ILOCK-Bausteins, indem es nach Empfang des timeOut-Signals eine vordefinierte Verzögerung (die Totzeit) abwartet und dann das entsprechende Ereignis auslöst.
4. **SPLIT** (A2X_SPLIT_2): Dupliziert den interlocked Zustand von ILOCK.OUT auf zwei Ausgänge:
   - OUT1 ⟶ externe Schnittstelle OUT (für physische Aktoren)
   - OUT2 ⟶ zurück zum BRIDGE.A2X_IN, um den Zustand über den A2X2-Roundtrip an das übergeordnete System zu senden.

Die Verbindung ILOCK.timeOut mit E_TimeOut.TimeOutSocket sorgt dafür, dass der Interlock nach Ablauf der Schutzzeit wieder für neue Richtungswechsel bereit ist.

Technische Besonderheiten

- **Composite-Design**: Alle Bausteine sind innerhalb der Subapp gekapselt; die Ressource sieht nur die Adapter IO und OUT. Dadurch wird die Wiederverwendbarkeit erhöht und die Netzwerkkonfiguration vereinfacht.
- **Last-Wins-Arbitrierung**: Der zuletzt aktivierte Taster gewinnt. Eine schnelle Umkehr (z. B. versehentliches Entprellen) wird durch die Schutz-Totzeit verhindert.
- **Bidirektionale Kommunikation**: Über den A2X2-Socket werden nicht nur Tasterzustände empfangen, sondern auch Rückschreibewerte (z. B. für Statusanzeigen) gesendet. Der Rückschreibepfad wird vom SPLIT-Baustein direkt aus dem Interlock-Zustand abgeleitet.
- **Parametrierbare Totzeit**: DT_PROTECT ist als Eingang konfigurierbar und kann zur Laufzeit angepasst werden (Standard: 50 ms).
- **Ereignisgesteuerte Zeitsteuerung**: Die Totzeit wird über den Ereignisbaustein E_TimeOut realisiert, was eine präzise und deterministische Zeitbasis ermöglicht.

Zustandsübersicht

Die Subapp selbst besitzt keine eigenen Zustände, da sie lediglich eine Komposition interner FB ist. Der interne ILOCK-Baustein hat typischerweise folgende Zustände:

- **IDLE**: Kein Taster aktiv, Ausgang neutral.
- **UP**: Der Aufwärts-Taster wurde zuletzt gedrückt (Ausgang high).
- **DOWN**: Der Abwärts-Taster wurde zuletzt gedrückt (Ausgang low).

- **PROTECT**: Ein Richtungswechsel wurde angefordert, aber die Totzeit läuft noch. Nach Ablauf von `DT_PROTECT` wird in den entsprechenden Zielzustand gewechselt.

Diese Zustände werden über den `OUT`-Adapter nach außen als A2X-Signal sichtbar.

Anwendungsszenarien

- **Steuerung von Antrieben** (z. B. Hub-/Senkmechanismen) mit zwei Tastern (Auf/Ab) über ein Netzwerkprotokoll, wobei der letzte Tastendruck Priorität hat.
- **Sicherheitskritische Schaltungen**, bei denen ein Richtungswechsel eine kurze Verzögerung benötigt, um mechanische Schäden zu vermeiden (z. B. Pumpen, Ventile).
- **Dezentrale Automatisierungssysteme**, bei denen Taster an einer entfernten Station eingelesen und der endgültige Schaltzustand über den Rückschreibepfad an die Zentrale gemeldet wird.
- **Anlagen mit mehreren Bedienpunkten**, die denselben A2X2-Kanal nutzen und per Last-Wins-Logik koordiniert werden.

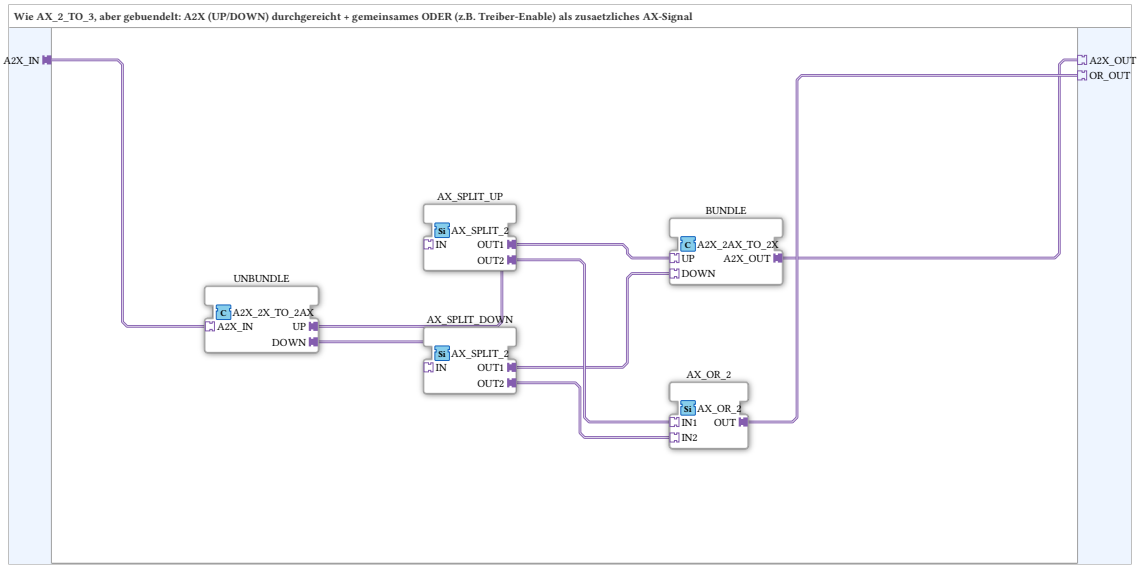
Vergleich mit ähnlichen Bausteinen

- **Ohne Totzeit** (`ILOCK_SWITCH` o. ä.): Reagieren sofort auf Richtungswechsel, können aber bei Prellen oder Rauschen zu unerwünschten Schaltvorgängen führen. `A2X2_ILOCK_SWITCH_PROTECT` fügt eine Schutzzeit ein.
- **Nur unidirektional** (`A2X_ILOCK...`): Verwenden einen separaten Rückkanal für den Rückschreibewert, während dieser FB den bidirektionalen A2X2-Kanal integriert.
- **Getrennte Bausteine**: Man könnte Bridge, Interlock und Split separat verwenden, müsste jedoch die Verbindungen selbst aufbauen und hätte eine komplexere Ressourcen-Konfiguration. Diese Subapp bietet eine vorgefertigte, kompakte Lösung.

Fazit

`A2X2_ILOCK_SWITCH_PROTECT` ist ein vielseitiger, in sich geschlossener Funktionsbaustein für Anwendungen, die eine robuste Taster-Interlock mit Totzeit übeir ein bidirektionales Netzwerkprotokoll benötigen. Die Kombination aus Bridge, Interlock und Split ermöglicht eine einfache Einbindung in bestehende Systeme, während die letztliche Arbitrierung zuverlässig und sicher arbeitet. Durch die Parametrierbarkeit der Schutzzeit lässt sich der Baustein flexibel an unterschiedliche Anforderungen anpassen.

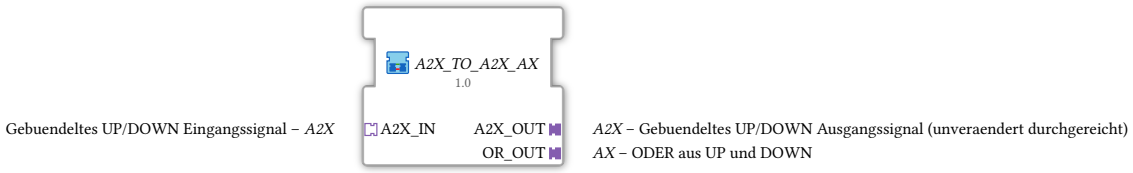
A2X_TO_A2X_AX



• – ○ • – ○ • – ○ •

Einleitung

`A2X_TO_A2X_AX` ist das gebündelte Pendant zu `AX_2_TO_3`: Statt zwei getrennter `AX`-Sockets/Plugs (`UP_IN`/`DOWN_IN`, `UP_OUT`/`DOWN_OUT`) nimmt dieser Baustein UP/DOWN gebündelt als ein einziges `A2X`-Signal entgegen, reicht es unverändert durch und liefert zusätzlich ein einzelnes `AX`-Signal mit dem ODER aus beiden Richtungen (z. B. für ein gemeinsames Treiber-Enable).



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: A2X_TO_A2X_AX

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **UNBUNDLE**: `adapter::conversion::unidirectional::A2X_2X_TO_2AX` — trennt das eingehende `A2X_IN` in zwei einzelne `AX`-Signale (UP, DOWN).
  - **AX_SPLIT_UP** / **AX_SPLIT_DOWN**: je `adapter::events::unidirectional::AX_SPLIT_2` — verzweigen je eine Richtung in einen Rückweg (zurück ins gebündelte Ausgangssignal) und einen Zweig zur ODER-Verknüpfung.
  - **BUNDLE**: `adapter::conversion::unidirectional::A2X_2AX_TO_2X` — fügt UP/DOWN wieder zu einem einzigen `A2X_OUT` zusammen.
  - **AX_OR_2**: `adapter::booleanOperators::AX_OR_2` — liefert auf `OR_OUT`, ob der Aktor gerade in irgendeine Richtung läuft.
- **Funktionsweise**: `A2X_IN` wird entbündelt, jede Richtung wird gesplittet (einmal zurück in die Bündelung, einmal in die ODER-Verknüpfung), UP/DOWN werden wieder gebündelt (`A2X_OUT`), und das ODER beider Richtungen liegt zusätzlich auf `OR_OUT`.

Programmablauf und Verbindungen

1. `A2X_IN` ⟶ `UNBUNDLE.A2X_IN` ⟶ `UNBUNDLE.UP` ⟶ `AX_SPLIT_UP.IN`; `UNBUNDLE.DOWN` ⟶ `AX_SPLIT_DOWN.IN`.
2. `AX_SPLIT_UP.OUT1` ⟶ `BUNDLE.UP`; `AX_SPLIT_DOWN.OUT1` ⟶ `BUNDLE.DOWN`; `BUNDLE.A2X_OUT` ⟶ `A2X_OUT`.
3. `AX_SPLIT_UP.OUT2` ⟶ `AX_OR_2.IN1`; `AX_SPLIT_DOWN.OUT2` ⟶ `AX_OR_2.IN2`; `AX_OR_2.OUT` ⟶ `OR_OUT`.

Technische Besonderheiten

- **Unveränderte Durchreichung trotz Bündelung**: Nach außen bleibt `A2X` gebündelt, intern wird UP/DOWN wie bei `AX_2_TO_3` unverändert durchgereicht.
- **Je ein Split pro Richtung**: Notwendig, weil jedes der beiden entbündelten Signale zu zwei Zielen muss — zurück in die Bündelung und in die ODER-Verknüpfung.

Anwendungsszenarien

- Schaltungen, in denen UP/DOWN ohnehin als gebündeltes `A2X`-Signal geführt werden (z. B. Richtung an einem Doppelwirkungsventil) und nicht extra in zwei einzelne `AX`-Signale aufgesplittet werden sollen.

Vergleich mit ähnlichen Bausteinen

Werden UP/DOWN ohnehin einzeln als `AX` geführt, ist stattdessen `AX_2_TO_3` zu verwenden — funktional identisch (Durchreichen plus ODER als drittes Signal), aber mit ungebündelter Schnittstelle.

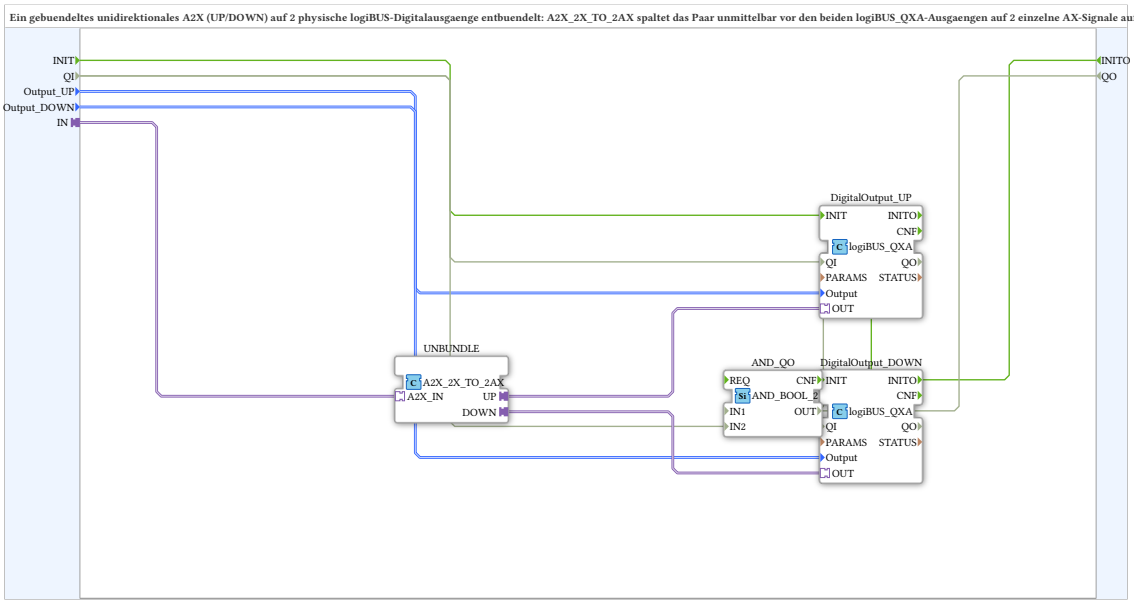
Zusammenfassung

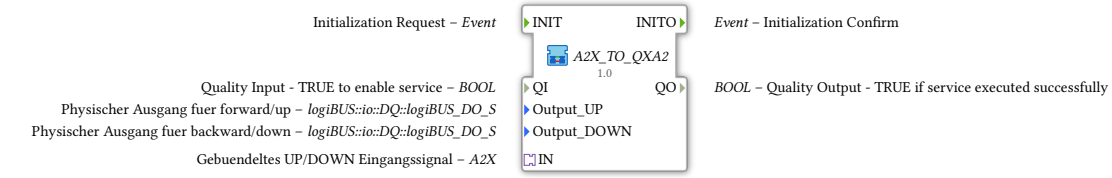
`A2X_TO_A2X_AX` reicht ein gebündeltes `A2X`-Signal unverändert durch und ergänzt es um ein zusätzliches `AX`-ODER-Signal — das gebündelte Gegenstück zu `AX_2_TO_3`.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

A2X_TO_QXA2





• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsbaustein **A2X_TO_QXA2** ist eine Subapplikation (Composite) zur Entbündelung eines unidirektionalen A2X-Signals (UP/DOWN) und zur direkten Ansteuerung zweier physischer logiBUS-Digitalausgänge. Die Aufteilung des gebündelten Signals in zwei einzelne AX-Signale erfolgt unmittelbar vor den beiden logiBUS_QXA-Ausgangsbausteinen. Dadurch wird die Entbündelung in die Composite-Struktur verlagert, nicht in die Resource des Geräts.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
`INIT`	`EInit`	Initialization Request

Ereignis-Ausgänge

Name	Typ	Kommentar
`INITO`	`EInit`	Initialization Confirm

Daten-Eingänge

Name	Typ	Initialwert	Kommentar
`QI`	`BOOL`	`TRUE`	Quality Input - TRUE, um den Dienst freizugeben
`Output_UP`	`logiBUS::io::DQ::logiBUS_DO_S`	`Invalid`	Physischer Ausgang für forward/up

Output_DOWN	logiBUS::io::DQ::logiBUS_DO_S	Invalid	Physischer Ausgang für backward/ down

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde

Adapter

Name	Typ	Richtung	Kommentar
IN	adapter::types::unidirectional::A2X	Socket	Gebündeltes UP/DOWN Eingangssignal

Funktionsweise

Die Subapplikation nimmt über den Socket IN ein gebündeltes unidirektionales A2X-Signal entgegen. Dieses Signal enthält zwei logisch getrennte Informationen: UP (Vorwärts) und DOWN (Rückwärts).

Im Inneren wird das A2X-Signal zunächst dem Funktionsbaustein UNBUNDLE (Typ adapter::conversion::unidirectional::A2X_2X_TO_2AX) zugeführt. Dieser Baustein splittet das gebündelte Signal in zwei einzelne AX-Signale, die an den Ausgängen UP und DOWN bereitstehen.

Anschließend werden diese beiden Signale über Adapterverbindungen an die beiden logiBUS_QXA-Ausgangsbausteine DigitalOutput_UP und DigitalOutput_DOWN übergeben. Die QXA-Bausteine wandeln die digitalen Signale in physische Ausgangswerte um. Die Sollwerte für diese Ausgänge werden über die Daten-Eingänge Output_UP und Output_DOWN von außen vorgegeben und mit den jeweiligen Output-Eingängen der QXA-Bausteine verbunden. Damit steuern die externen Vorgaben die tatsächlichen Ausgangszustände der logiBUS-Digitalausgänge.

Die Initialisierung wird durch die Ereigniskette INIT → DigitalOutput_UP.INIT, DigitalOutput_UP.INITO → DigitalOutput_DOWN.INIT, DigitalOutput_DOWN.INITO → INITO durchgereicht. QI wird an die QI-Eingänge beider QXA-Bausteine weitergegeben. Die QO-Ausgänge beider QXA-Bausteine werden im Baustein AND_QO (Typ iec61131::booleanOperators::AND_BOOL_2) per UND verknüpft und als QO ausgegeben; der Baustein AND_QO wird durch DigitalOutput_DOWN.INITO über REQ ausgelöst. Die Signalverarbeitung selbst (A2X → Ausgänge) erfolgt weiterhin rein datenflussorientiert über Adapterverbindungen. Die gesamte Logik ist in der Composite-Ebene gekapselt und benötigt keine zusätzlichen Ressourcen-Ressourcen.

Technische Besonderheiten

- **Composite-Stil:** Die Entbündelung und die physische Ausgangsansteuerung sind als Subapplikation realisiert und können somit in verschiedenen Ressourcen oder Geräten wiederverwendet werden.
- **Verwendung von logiBUS_QXA:** Die Ausgangsbausteine sind vom Typ logiBUS::io::DQ::logiBUS_QXA und erhalten ihren QI-Eingang vom QI-Eingang der Subapplikation (Initialwert TRUE).
- **Nur Initialisierungsereignisse:** Außer INIT/INITO gibt es keine Ereignisse; die Signalverarbeitung arbeitet rein signal- bzw. datengetrieben, was die Einbindung in zyklische oder ereignisgesteuerte Umgebungen vereinfacht.
- **Durchgereichter Status:** QO ist die UND-Verknüpfung (AND_QO) der QO-Ausgänge von DigitalOutput_UP und DigitalOutput_DOWN.
- **Daten-Eingänge als Sollwerte:** Output_UP und Output_DOWN sind als Strukturvariablen vom Typ logiBUS_DO_S definiert und standardmäßig auf Invalid gesetzt, was eine explizite Initialisierung durch die Anwendung erfordert.

Zustandsübersicht

Da es sich um eine rein strukturelle Subapplikation ohne eigene Zustandslogik handelt, existieren keine internen Zustände oder Zustandsübergänge. Die Funktion ergibt sich ausschließlich aus der Verdrahtung der enthaltenen Bausteine.

Anwendungsszenarien

- **Maschinensteuerung:** Ansteuerung zweier digitaler Ausgänge für Vorwärts-/Rückwärtsbewegungen (z.B. Förderbänder, Antriebe) über eine einzige A2X-Datenverbindung.
- **Logistik- und Materialfluss:** Integration in logiBUS-basierte Steuerungssysteme, bei denen gebündelte Signale auf physische Ausgänge verteilt werden müssen.
- **Modulare Composite-Verwendung:** Wiederverwendbare Einheit für Geräte, die eine direkte Aufteilung von A2X-Signalen auf zwei diskrete Ausgänge benötigen, ohne die Resource selbst mit Entbündelungslogik belasten zu müssen.

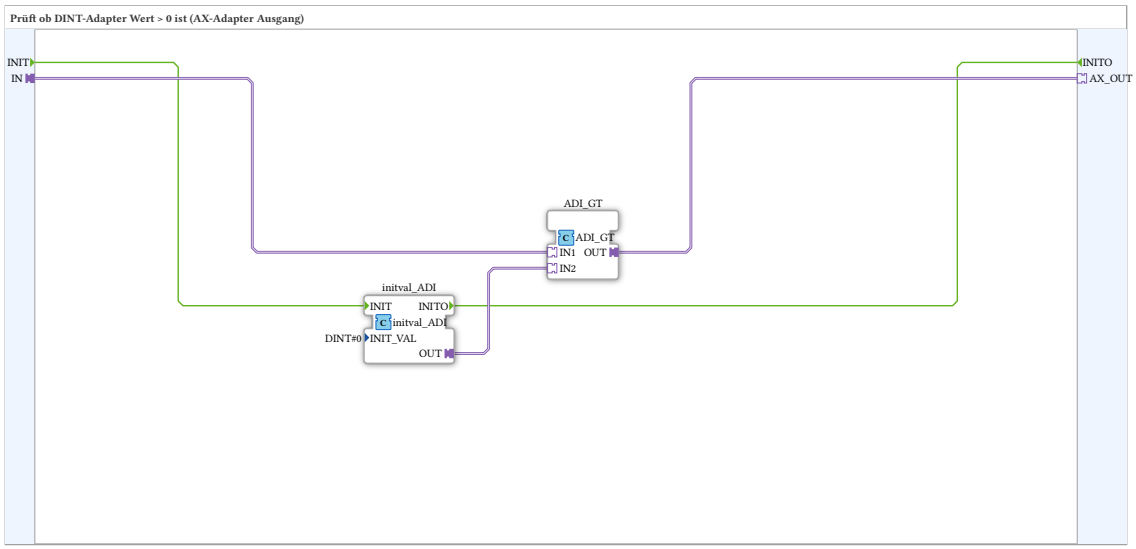
Vergleich mit ähnlichen Bausteinen

- **A2X_2X_TO_2AX (direkt):** Dieser Baustein liefert nur die entbündelten Signale, ohne physische Ansteuerung. `A2X_TO_QXA2` erweitert diese Funktionalität um die Integration der logiBUS-QXA-Ausgangsbausteine.
- **Manuelle Entbündelung auf Resource-Ebene:** Alternativ könnte die Entbündelung direkt in der Resource erfolgen; dieser Composite-Baustein kapselt diesen Schritt und reduziert so den Verbindungsaufwand im übergeordneten System.
- **A2X_TO_2X (mit diskreten Ausgängen):** Falls ausschließlich diskrete Signale ohne physische Ausgangstreiber benötigt werden, wäre dieser Baustein eine schlankere Variante – `A2X_TO_QXA2` ist speziell für die direkte logiBUS-Anbindung optimiert.

Fazit

Der Subapplikationsbaustein **A2X_TO_QXA2** bietet eine kompakte und wiederverwendbare Lösung, um ein gebündeltes unidirektionales A2X-Signal in zwei physische logiBUS-Digitalausgänge zu überführen. Durch die Integration der Entbündelungs- und Ausgangsbausteine in einer Composite-Struktur wird die Komplexität vom Anwender ferngehalten und die Wartung sowie Wiederverwendung vereinfacht. Die reine Datenflussarchitektur ermöglicht eine unkomplizierte Integration in verschiedene Steuerungsumgebungen.

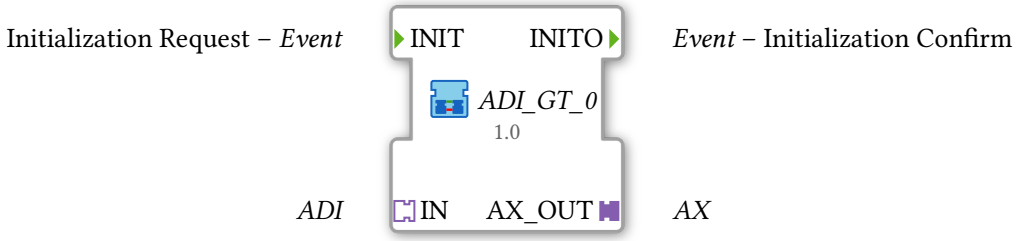
ADI_GT_0



• – ○ • – ○ • – ○ •

Einleitung

`ADI_GT_0` prüft, ob ein DINT-Adapter-Wert größer als 0 ist, und gibt das Ergebnis als boolesches AX-Adapter-Signal aus. Typischer Eingangswert ist ein vorzeichenbehafteter 32-Bit-Wert. Der Baustein ist für jeden Zahlentyp gleich aufgebaut (`AS`, `AI`, `ADI`, `ALI`, `AUS`, `AUI`, `AUDI`, `AULI`, `AR`, `ALR`).



Schnittstelle

- **Ereignis-Eingang**: `INIT` (Initialisierung)
- **Ereignis-Ausgang**: `INITO` (Initialisierung abgeschlossen, kommt vom internen `initval_ADI`)
- **Sockel**: `IN` (`ADI`) — der Wert, der gegen 0 verglichen wird
- **Stecker**: `AX_OUT` (`AX`) — TRUE, wenn `IN > 0`

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: ADI_GT_0

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **initval_ADI**: `adapter::types::unidirectional::ADI::initval::initval_ADI` — liefert die feste Vergleichskonstante `DINT#0` als Adapter-Wert (`INIT_VAL=DINT#0`). Ein Adapter-Sockel kann die Konstante nicht als Parameter bekommen, deshalb wird sie über diesen Baustein eingespeist.
  - **ADI_GT**: `adapter::iec61131::comparison::ADI_GT` — adapter-nativer Vergleich `IN1 > IN2`.
- **Funktionsweise**: `IN` ⟶ `ADI_GT.IN1`; Konstante 0 ⟶ `ADI_GT.IN2`; Ergebnis ⟶ `AX_OUT`.

Programmablauf und Verbindungen

1. `INIT` ⟶ `initval_ADI.INIT`; `initval_ADI.INITO` ⟶ `INITO`.
2. `IN` (Adapter) ⟶ `ADI_GT.IN1`.
3. `initval_ADI.OUT` (Konstante `DINT#0`) ⟶ `ADI_GT.IN2`.
4. `ADI_GT.OUT` ⟶ `AX_OUT` (Adapter).

Anwendungsszenarien

- Umwandlung von ein vorzeichenbehafteter 32-Bit-Wert in ein einfaches boolesches Adapter-Signal für VT-Statusanzeigen oder Freigabelogik.

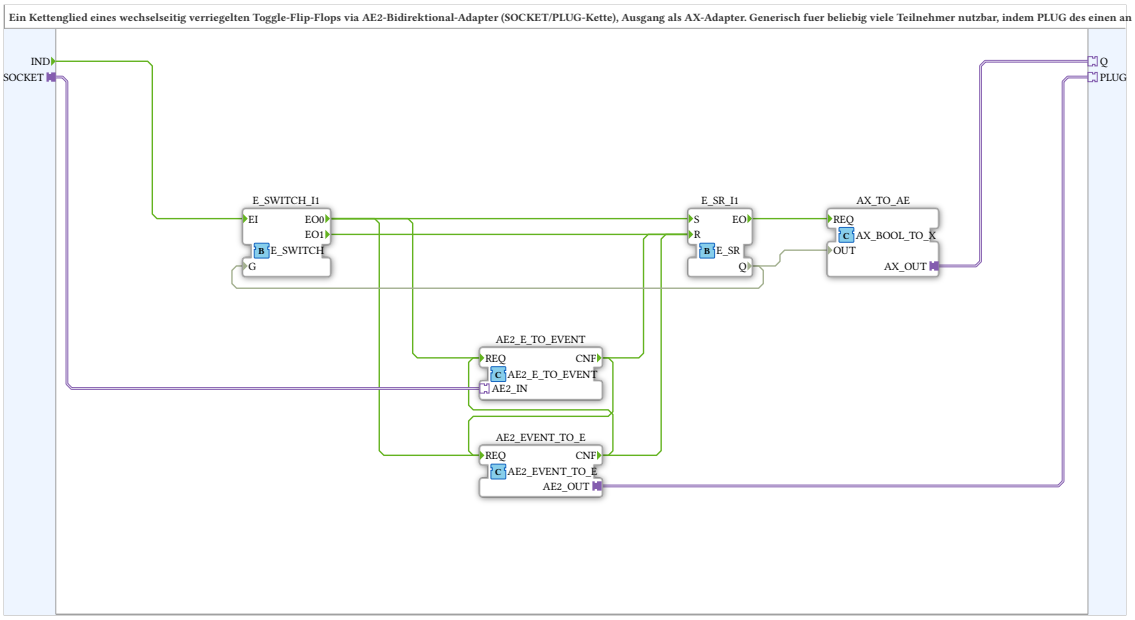
Zusammenfassung

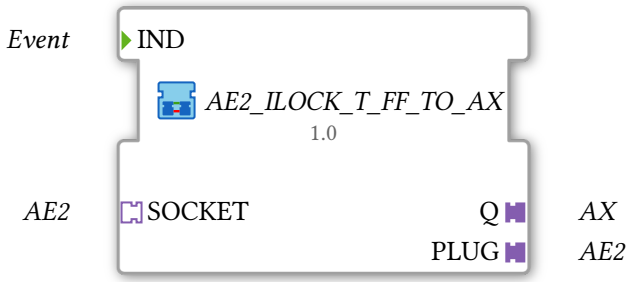
`ADI_GT_0` ist ein kompakter Wrapper um den adapter-nativen Vergleich `ADI_GT` — wandelt „DINT > 0“ direkt in ein AX-Adapter-Signal um, ohne dass der Aufrufer die Null-Konstante selbst anlegen und verdrahten muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AE2_ILOCK_T_FF_TO_AX





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AE2_ILOCK_T_FF_TO_AX` realisiert ein wechselseitig verriegeltes Toggle-Flip-Flop, das über bidirektionale AE2-Adapter (SOCKET und PLUG) mit weiteren gleichartigen Bausteinen zu einer Kette verbunden werden kann. Nur ein Glied der Kette kann gleichzeitig aktiv sein, d.h. nur ein Ausgang `Q` liefert einen logischen `TRUE`-Zustand. Der aktive Zustand wird über einen unidirektionalen AX-Adapter nach außen geführt.

Der Baustein ist generisch für beliebig viele Teilnehmer einsetzbar: Durch Verbinden des PLUG-Ausgangs eines Gliedes mit dem SOCKET-Eingang des nächsten entsteht eine verriegelte Kette, bei der ein Ereignisimpuls am Eingang `IND` eines beliebigen Gliedes dessen Zustand toggelt und dabei alle anderen Glieder in den inaktiven Zustand versetzt.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung
`IND`	Event	Externer Trigger, der das Toggle-Verhalten des Bausteins anstößt.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Name	Richtung	Typ	Beschreibung
SOCKET	Eingang (bidirektional)	adapter::types::bidirectional::AE2	Bidirektionale Schnittstelle zum Vorgängerglied der Kette. Empfängt Verriegelungssignale und gibt das aktive Zustandssignal weiter.
PLUG	Ausgang (bidirektional)	adapter::types::bidirectional::AE2	Bidirektionale Schnittstelle zum Nachfolgeglied. Sendet Verriegelungssignale und empfängt Rückmeldungen.
Q	Ausgang (unidirektional)	adapter::types::unidirectional::AX	Ausgabe des aktuellen Toggle-Zustands (TRUE = aktiv, FALSE = inaktiv) für die externe Anwendung.

Funktionsweise

Intern verwendet der Baustein folgende Funktionsblöcke:

- E_SR_I1 (Typ iec61499::events::E_SR): Ein SR-Flip-Flop, das den Toggle-Zustand speichert (Q-Ausgang).
- E_SWITCH_I1 (Typ iec61499::events::E_SWITCH): Verteilt ein eingehendes Ereignis abhängig vom Wert des Daten-Eingangs G auf zwei Ausgänge (EO0, EO1).
- AE2_EVENT_TO_E und AE2_E_TO_EVENT (adapter::conversion::bidirectional): Konvertieren Ereignisse in Signale über den AE2-Adapter und umgekehrt.
- AX_TO_AE (adapter::conversion::unidirectional::AX_BOOL_TO_X): Wandelt den Q-Zustand in ein AX-Signal für den Ausgangsadapter.

Signalfluss bei einem Ereignis am Eingang IND:

1. Das Ereignis IND erreicht den Eingang EI von E_SWITCH_I1.
2. E_SWITCH_I1 prüft den Wert seines Daten-Eingangs G, der mit E_SR_I1.Q verbunden ist.
   - Ist Q = FALSE, wird das Ereignis an EO0 ausgegeben.
   - Ist Q = TRUE, wird das Ereignis an EO1 ausgegeben.
3. Die Ausgänge EO0 und EO1 steuern das SR-Flip-Flop:
   - EO0 ⟶ S (Set) setzt Q auf TRUE.
   - EO1 ⟶ R (Reset) setzt Q auf FALSE.
4. Parallel dazu werden die Ereignisse von EO0 auch an die Konvertierungsblöcke AE2_EVENT_TO_E und AE2_E_TO_EVENT gesendet. Diese erzeugen über den AE2-Adapter Verriegelungssignale an die benachbarten Glieder der Kette.
5. Die von den Nachbarn empfangenen Ereignisse (über SOCKET oder PLUG) werden durch AE2_EVENT_TO_E bzw. AE2_E_TO_EVENT in Ereignisse zurückgewandelt und führen über CNF-Rückmeldungen zu einem Reset des eigenen SR-Flip-Flops, falls das eigene Glied nicht das aktive sein soll.
6. Nach dem Toggle wird E_SR_I1.EO (das Ausgangsereignis des SR-Flip-Flops) verwendet, um über AX_TO_AE den Ausgangsadapter Q zu aktualisieren.

Die gegenseitige Verriegelung wird durch die bidirektionale Kommunikation über den AE2-Adapter erreicht: Jedes Glied sendet seinen Zustand (aktiv/inaktiv) an seine Nachbarn. Ein aktives Glied verriegelt alle anderen, sodass diese bei einem internen Toggle-Versuch in den inaktiven Zustand zurückgesetzt werden.

Technische Besonderheiten

- **Bidirektionale Kette:** Der Baustein verwendet zwei bidirektionale AE2-Schnittstellen (SOCKET und PLUG), die eine Verkettung in einer Linie ermöglichen. Jedes Glied kann sowohl mit seinem Vorgänger als auch mit seinem Nachfolger kommunizieren.

- **Verriegelung durch Ereignisrückkopplung:** Die Konvertierungsblöcke `AE2_EVENT_TO_E` und `AE2_E_TO_EVENT` sind in einer Schleife verschaltet, sodass eingehende Ereignisse von Nachbarn ein Reset des eigenen Flip-Flops auslösen. Dadurch wird sichergestellt, dass immer nur ein Glied aktiv ist.
- **Ausgang als AX-Adapter:** Der Zustand `Q` wird über einen unidirektionalen AX-Adapter bereitgestellt, sodass er ohne weitere Konvertierung direkt von Applikationsbausteinen gelesen werden kann.
- **Generische Erweiterbarkeit:** Durch Aneinanderreihung mehrerer dieser Bausteine lässt sich eine Verriegelung mit beliebig vielen Teilnehmern aufbauen, ohne zusätzliche Logik außerhalb der Bausteine.

Zustandsübersicht

Der interne Zustand wird durch das SR-Flip-Flop `E_SR_I1` repräsentiert:

Zustand `Q`	Bedeutung	Verhalten bei `IND`
`FALSE` (inaktiv)	Das Glied ist nicht aktiv.	Ein Ereignis setzt `Q` auf `TRUE`; das Glied wird aktiv.
`TRUE` (aktiv)	Das Glied ist aktiv, alle anderen sind verriegelt.	Ein Ereignis setzt `Q` zurück auf `FALSE`; das Glied wird inaktiv.

Nach außen wird `Q` über den `AX_TO_AE`-Block an den Ausgangsadapter `Q` weitergegeben.

Anwendungsszenarien

- **Mehrere Verbraucher mit exklusiver Freigabe:** Z.B. mehrere Motoren oder Lichter, bei denen immer nur einer eingeschaltet sein darf. Jedes Gerät erhält ein eigenes Kettenglied; ein Taster an einem Glied schaltet es ein und alle anderen aus.
- **Wechselseitige Verriegelung in Steuerungssystemen:** Einsatz in sicherheitsrelevanten Anlagen, bei denen zwei Zustände sich gegenseitig ausschließen müssen.
- **Daisy-Chain-Konfiguration:** Einfache Verdrahtung über die AE2-Adapter, ohne zusätzliche Bus- oder Matrix-Logik.

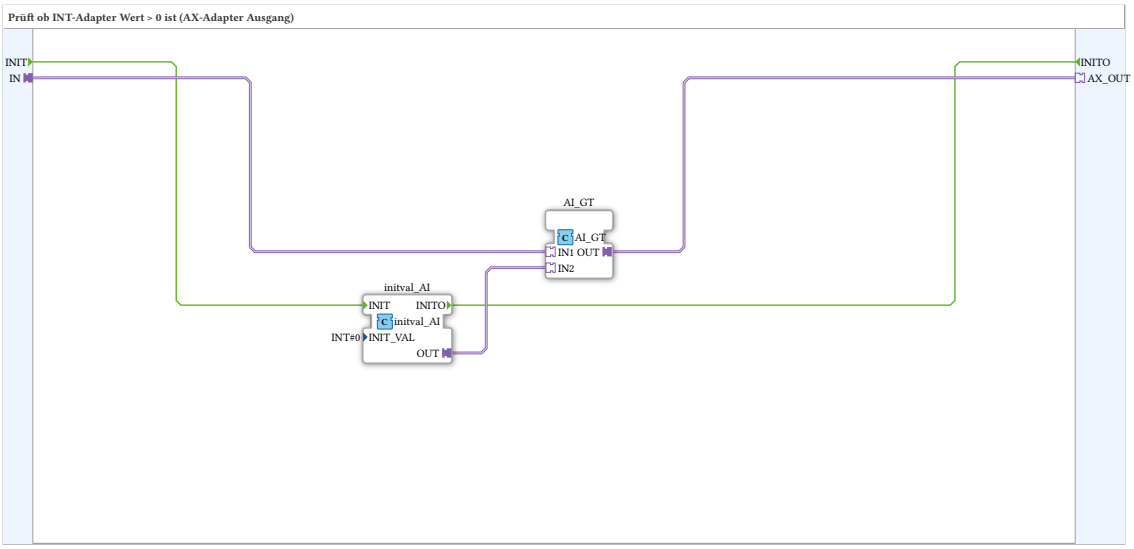
Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem einfachen Toggle-Flip-Flop (z.B. `E_Toggle`) bietet dieser Baustein zusätzlich die Verriegelung über eine bidirektionale Kommunikation. Ein einfacher Toggle hat keine Kenntnis über andere Bausteine und kann nicht gewährleisten, dass nur ein Ausgang aktiv ist. Der hier beschriebene Baustein integriert die Verriegelung direkt in die Kette, sodass eine exklusive Zustandsverteilung ohne übergeordnete Steuerung möglich ist.

Fazit

`AE2_ILOCK_T_FF_TO_AX` ist ein flexibler, generisch einsetzbarer Baustein für die Realisierung exklusiver Zustände in einer vernetzten Steuerung. Durch die Kombination eines Toggle-Flip-Flops mit bidirektionalen AE2-Adaptern entsteht eine robuste Lösung für Kettenverriegelungen, die sich mit minimalem Verdrahtungsaufwand zu beliebig großen Systemen erweitern lässt.

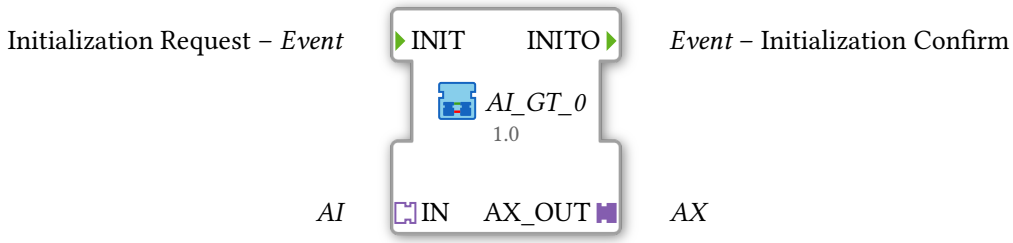
AI_GT_0



• – ◦ • – ◦ • – ◦ •

Einleitung

`AI_GT_0` prüft, ob ein INT-Adapter-Wert größer als 0 ist, und gibt das Ergebnis als boolesches AX-Adapter-Signal aus. Typischer Eingangswert ist ein vorzeichenbehafteter 16-Bit-Wert. Der Baustein ist für jeden Zahlentyp gleich aufgebaut (`AS`, `AI`, `ADI`, `ALI`, `AUS`, `AUI`, `AUDI`, `AULI`, `AR`, `ALR`).



Schnittstelle

- **Ereignis-Eingang**: `INIT` (Initialisierung)
- **Ereignis-Ausgang**: `INITO` (Initialisierung abgeschlossen, kommt vom internen `initval_AI`)
- **Sockel**: `IN` (`AI`) — der Wert, der gegen 0 verglichen wird
- **Stecker**: `AX_OUT` (`AX`) — TRUE, wenn `IN > 0`

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: AI_GT_0

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **initval_AI**: `adapter::types::unidirectional::AI::initval::initval_AI` — liefert die feste Vergleichskonstante `INT#0` als Adapter-Wert (`INIT_VAL=INT#0`). Ein Adapter-Sockel kann die Konstante nicht als Parameter bekommen, deshalb wird sie über diesen Baustein eingespeist.
  - **AI_GT**: `adapter::iec61131::comparison::AI_GT` — adapter-nativer Vergleich `IN1 > IN2`.
- **Funktionsweise**: `IN` ⟶ `AI_GT.IN1`; Konstante 0 ⟶ `AI_GT.IN2`; Ergebnis ⟶ `AX_OUT`.

Programmablauf und Verbindungen

1. `INIT` ⟶ `initval_AI.INIT`; `initval_AI.INITO` ⟶ `INITO`.
2. `IN` (Adapter) ⟶ `AI_GT.IN1`.
3. `initval_AI.OUT` (Konstante `INT#0`) ⟶ `AI_GT.IN2`.
4. `AI_GT.OUT` ⟶ `AX_OUT` (Adapter).

Anwendungsszenarien

- Umwandlung von ein vorzeichenbehafteter 16-Bit-Wert in ein einfaches boolesches Adapter-Signal für VT-Statusanzeigen oder Freigabelogik.

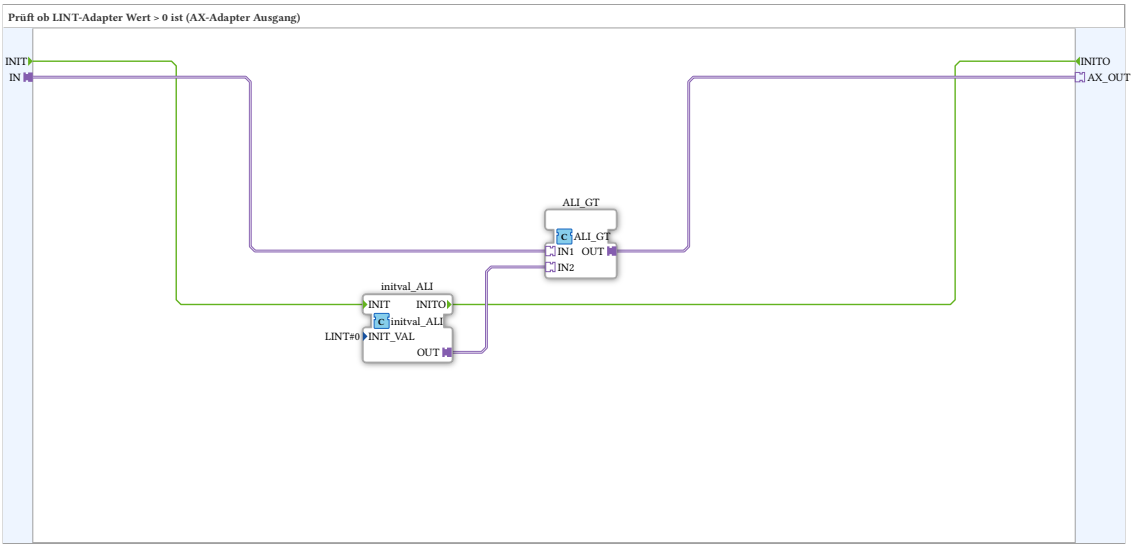
Zusammenfassung

`AI_GT_0` ist ein kompakter Wrapper um den adapter-nativen Vergleich `AI_GT` — wandelt „INT > 0“ direkt in ein AX-Adapter-Signal um, ohne dass der Aufrufer die Null-Konstante selbst anlegen und verdrahten muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

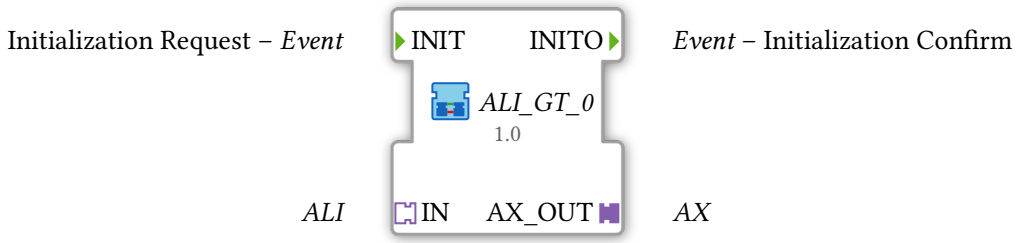
ALI_GT_0



• – ○ • – ○ • – ○ •

Einleitung

`ALI_GT_0` prüft, ob ein LINT-Adapter-Wert größer als 0 ist, und gibt das Ergebnis als boolesches AX-Adapter-Signal aus. Typischer Eingangswert ist ein vorzeichenbehafteter 64-Bit-Wert. Der Baustein ist für jeden Zahlentyp gleich aufgebaut (`AS`, `AI`, `ADI`, `ALI`, `AUS`, `AUI`, `AUDI`, `AULI`, `AR`, `ALR`).



Schnittstelle

- **Ereignis-Eingang**: `INIT` (Initialisierung)
- **Ereignis-Ausgang**: `INITO` (Initialisierung abgeschlossen, kommt vom internen `initval_ALI`)
- **Sockel**: `IN` (`ALI`) — der Wert, der gegen 0 verglichen wird
- **Stecker**: `AX_OUT` (`AX`) — TRUE, wenn `IN > 0`

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: ALI_GT_0

- **Typ**: SubAppType

- **Verwendete interne FBs**:
  - **initval_ALI**: `adapter::types::unidirectional::ALI::initval::initval_ALI` — liefert die feste Vergleichskonstante `LINT#0` als Adapter-Wert (`INIT_VAL=LINT#0`). Ein Adapter-Sockel kann die Konstante nicht als Parameter bekommen, deshalb wird sie über diesen Baustein eingespeist.
  - **ALI_GT**: `adapter::iec61131::comparison::ALI_GT` — adapter-nativer Vergleich `IN1 > IN2`.
- **Funktionsweise**: `IN` ⟶ `ALI_GT.IN1`; Konstante 0 ⟶ `ALI_GT.IN2`; Ergebnis ⟶ `AX_OUT`.

Programmablauf und Verbindungen

1. `INIT` ⟶ `initval_ALI.INIT`; `initval_ALI.INITO` ⟶ `INITO`.
2. `IN` (Adapter) ⟶ `ALI_GT.IN1`.
3. `initval_ALI.OUT` (Konstante `LINT#0`) ⟶ `ALI_GT.IN2`.
4. `ALI_GT.OUT` ⟶ `AX_OUT` (Adapter).

Anwendungsszenarien

- Umwandlung von ein vorzeichenbehafteter 64-Bit-Wert in ein einfaches boolesches Adapter-Signal für VT-Statusanzeigen oder Freigabelogik.

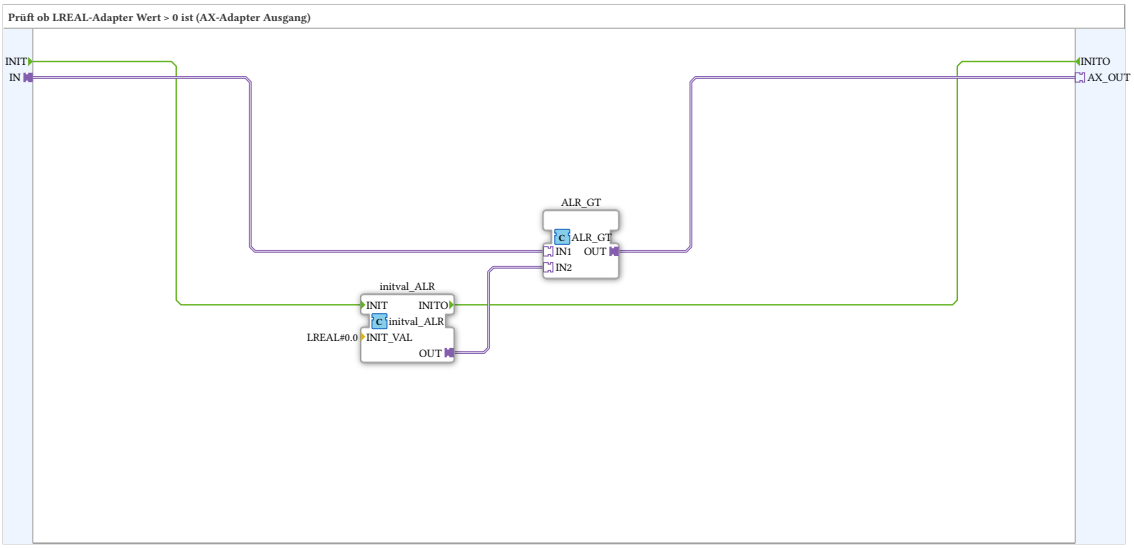
Zusammenfassung

`ALI_GT_0` ist ein kompakter Wrapper um den adapter-nativen Vergleich `ALI_GT` — wandelt „LINT > 0“ direkt in ein AX-Adapter-Signal um, ohne dass der Aufrufer die Null-Konstante selbst anlegen und verdrahten muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

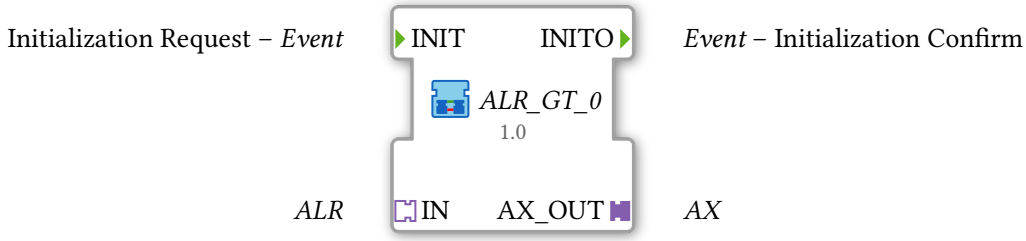
ALR_GT_0



• – ○ • – ○ • – ○ •

Einleitung

`ALR_GT_0` prüft, ob ein LREAL-Adapter-Wert größer als 0 ist, und gibt das Ergebnis als boolesches AX-Adapter-Signal aus. Typischer Eingangswert ist ein Gleitkommawert (LREAL). Der Baustein ist für jeden Zahlentyp gleich aufgebaut (`AS`, `AI`, `ADI`, `ALI`, `AUS`, `AUI`, `AUDI`, `AULI`, `AR`, `ALR`).



Schnittstelle

- **Ereignis-Eingang**: `INIT` (Initialisierung)
- **Ereignis-Ausgang**: `INITO` (Initialisierung abgeschlossen, kommt vom internen `initval_ALR`)
- **Sockel**: `IN` (`ALR`) — der Wert, der gegen 0 verglichen wird
- **Stecker**: `AX_OUT` (`AX`) — TRUE, wenn `IN > 0`

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: ALR_GT_0

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **initval_ALR**: `adapter::types::unidirectional::ALR::initval::initval_ALR` — liefert die feste Vergleichskonstante `LREAL#0.0` als Adapter-Wert (`INIT_VAL=LREAL#0.0`). Ein Adapter-Sockel kann die Konstante nicht als Parameter bekommen, deshalb wird sie über diesen Baustein eingespeist.
  - **ALR_GT**: `adapter::iec61131::comparison::ALR_GT` — adapter-nativer Vergleich `IN1 > IN2`.
- **Funktionsweise**: `IN` ⟶ `ALR_GT.IN1`; Konstante 0 ⟶ `ALR_GT.IN2`; Ergebnis ⟶ `AX_OUT`.

Programmablauf und Verbindungen

1. `INIT` ⟶ `initval_ALR.INIT`; `initval_ALR.INITO` ⟶ `INITO`.
2. `IN` (Adapter) ⟶ `ALR_GT.IN1`.
3. `initval_ALR.OUT` (Konstante `LREAL#0.0`) ⟶ `ALR_GT.IN2`.
4. `ALR_GT.OUT` ⟶ `AX_OUT` (Adapter).

Anwendungsszenarien

- Umwandlung von ein Gleitkommawert (LREAL) in ein einfaches boolesches Adapter-Signal für VT-Statusanzeigen oder Freigabelogik.

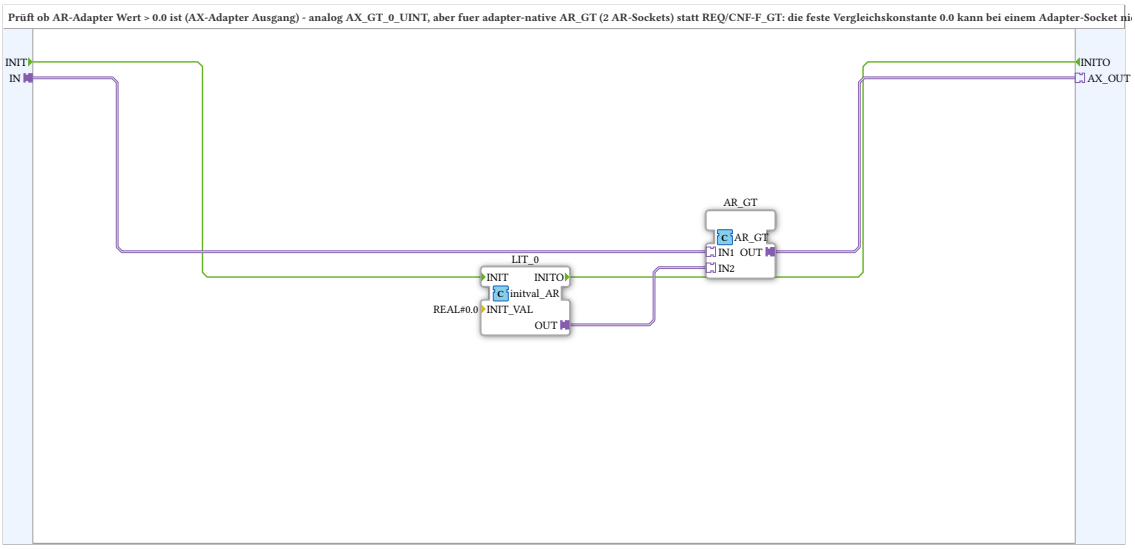
Zusammenfassung

`ALR_GT_0` ist ein kompakter Wrapper um den adapter-nativen Vergleich `ALR_GT` — wandelt „LREAL > 0“ direkt in ein AX-Adapter-Signal um, ohne dass der Aufrufer die Null-Konstante selbst anlegen und verdrahten muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

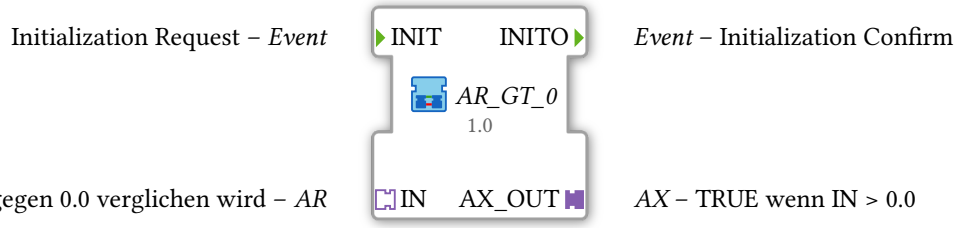
AR_GT_0



• – ◦ • – ◦ • – ◦ •

Einleitung

`AR_GT_0` prüft, ob ein REAL-Adapter-Wert größer als 0 ist, und gibt das Ergebnis als boolesches AX-Adapter-Signal aus. Typischer Eingangswert ist ein Gleitkommawert (REAL), z. B. eine Geschwindigkeit. Der Baustein ist für jeden Zahlentyp gleich aufgebaut (`AS`, `AI`, `ADI`, `ALI`, `AUS`, `AUI`, `AUDI`, `AULI`, `AR`, `ALR`).



Schnittstelle

- **Ereignis-Eingang**: `INIT` (Initialisierung)
- **Ereignis-Ausgang**: `INITO` (Initialisierung abgeschlossen, kommt vom internen `LIT_0`)
- **Sockel**: `IN` (`AR`) — der Wert, der gegen 0 verglichen wird
- **Stecker**: `AX_OUT` (`AX`) — TRUE, wenn `IN > 0`

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: AR_GT_0

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **LIT_0**: `adapter::types::unidirectional::AR::initval::initval_AR` — liefert die feste Vergleichskonstante `0.0` als Adapter-Wert (`INIT_VAL=0.0`). Ein Adapter-Sockel kann die Konstante nicht als Parameter bekommen, deshalb wird sie über diesen Baustein eingespeist.
  - **AR_GT**: `adapter::iec61131::comparison::AR_GT` — adapter-nativer Vergleich `IN1 > IN2`.
- **Funktionsweise**: `IN` ⟶ `AR_GT.IN1`; Konstante 0 ⟶ `AR_GT.IN2`; Ergebnis ⟶ `AX_OUT`.

Programmablauf und Verbindungen

1. `INIT` ⟶ `LIT_0.INIT`; `LIT_0.INITO` ⟶ `INITO`.
2. `IN` (Adapter) ⟶ `AR_GT.IN1`.
3. `LIT_0.OUT` (Konstante `0.0`) ⟶ `AR_GT.IN2`.
4. `AR_GT.OUT` ⟶ `AX_OUT` (Adapter).

Anwendungsszenarien

- Umwandlung von ein Gleitkommawert (REAL), z. B. eine Geschwindigkeit in ein einfaches boolesches Adapter-Signal für VT-Statusanzeigen oder Freigabelogik.

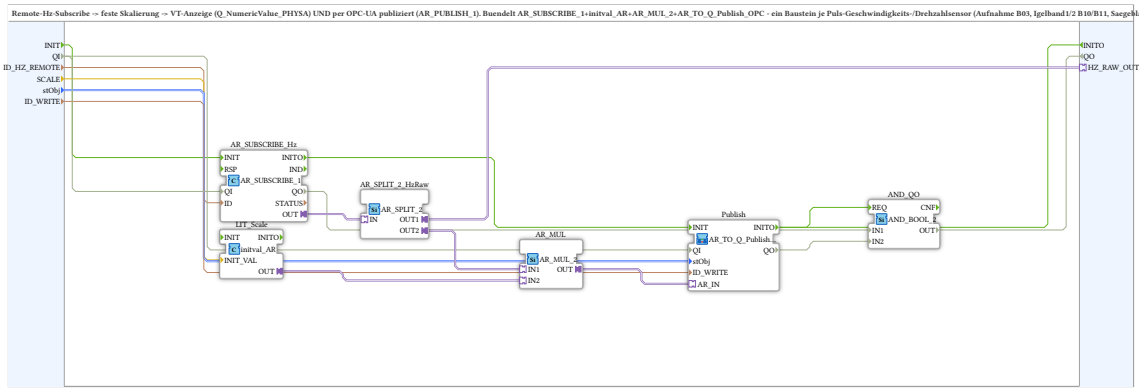
Zusammenfassung

`AR_GT_0` ist ein kompakter Wrapper um den adapter-nativen Vergleich `AR_GT` — wandelt „REAL > 0“ direkt in ein AX-Adapter-Signal um, ohne dass der Aufrufer die Null-Konstante selbst anlegen und verdrahten muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AR_HZ_TO_Q_Publish_OPC



• — ○ • — ○ • — ○ •

Einleitung

`AR_HZ_TO_Q_Publish_OPC` bündelt die Kette **Remote-Subscribe einer Frequenz (Hz) → feste Skalierung → VT-Anzeige (`Q_NumericValue_PHYSA`) und OPC-UA-Publish** zu einer wiederverwendbaren SubApp. Sie fasst `AR_SUBSCRIBE_1`, `initval_AR`, `AR_MUL_2` und `AR_TO_Q_Publish_OPC` zusammen – ein Baustein je Puls-Geschwindigkeits-/Drehzahlsensor (Aufnahme B03, Igelband1/2 B10/B11, Sägeblatt B08), nur die Parameter unterscheiden sich. Sie wurde aus `Inputs_STG1.SUB` herausgelöst, wo vier nahezu identische Ketten existierten.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; startet die Initialisierungskette (`AR_SUBSCRIBE_Hz` → `Publish`)

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung; kommt von `Publish.INITO`, nachdem die Kette durchlaufen ist

Daten-Eingänge

- **QI** (`BOOL`): Quality Input, Freigabe des Dienstes (Initialwert `TRUE`), wird an `AR_SUBSCRIBE_Hz.QI` und `Publish.QI` weitergereicht
- **ID_HZ_REMOTE** (`WSTRING`): Remote-Subscribe-Adresse der publizierten Frequenz (Hz), z. B. `FBs::Ventilsteuerung::Vent`
- **SCALE** (`REAL`): Skalierungsfaktor Hz → physikalische Einheit (z. B. `3.6/53` für km/h, `60.0` für U/min bei 1 Impuls/Umdrehung)
- **stObj** (`logiBUS::utils::conversion::phys::NumericObjectPool_S`): Object-Pool-Eigenschaften der VT-Zahlenvariable (`u16ObjId`, `scale`, `offset`, `decimals`), z. B. `FBs::const::DefaultPool_Numeric::NumberVariable_X_N`

- **ID_WRITE** (WSTRING): Lokale Publish-Adresse (REAL, physikalischer Wert) für Web-Mirror/Dashboard/Self-Subscribe

Daten-Ausgänge

- **QO** (BOOL): Quality Output, AND der QO-Ausgänge von AR_SUBSCRIBE_Hz und Publish (über AND_QO)

Adapter

- **HZ_RAW_OUT** (Plug): Unidirektionaler Adapter vom Typ adapter::types::unidirectional::AR – optionaler Nur-Lese-Spiegel des noch unskalierten Rohwerts (Hz) vor der SCALE-Multiplikation. Unverdrahtet lassen, wenn nicht gebraucht.

Verwendete Funktionsbausteine (FBs)

- **AR_SUBSCRIBE_Hz**: adapter::net::AR_SUBSCRIBE_1 (QI kommt von der SubApp-Schnittstelle), abonniert die Remote-Adresse ID_HZ_REMOTE
- **LIT_Scale**: adapter::types::unidirectional::AR::initval::initval_AR, liefert SCALE als AR-Wert
- **AR_SPLIT_2_HzRaw**: adapter::events::unidirectional::AR_SPLIT_2, verzweigt den Rohwert
- **AR_MUL**: adapter::iec61131::arithmetic::AR_MUL_2, multipliziert Rohwert mit SCALE
- **Publish**: SubApp MyLib::sys::AR_TO_Q_Publish_OPC, zeigt den Wert im VT an (stObj) und publiziert ihn per OPC-UA (ID_WRITE)
- **AND_QO**: iec61131::booleanOperators::AND_BOOL_2, verknüpft die QO-Ausgänge von AR_SUBSCRIBE_Hz und Publish zum SubApp-Ausgang QO

Programmablauf und Verbindungen

1. ID_HZ_REMOTE → AR_SUBSCRIBE_Hz.ID; der Ausgang AR_SUBSCRIBE_Hz.OUT führt auf AR_SPLIT_2_HzRaw.IN.
2. AR_SPLIT_2_HzRaw.OUT1 → Plug HZ_RAW_OUT (unskalierter Rohwert, optional).
3. AR_SPLIT_2_HzRaw.OUT2 → AR_MUL.IN1; SCALE → LIT_Scale.INIT_VAL, LIT_Scale.OUT → AR_MUL.IN2.
4. AR_MUL.OUT → Publish.AR_IN.
5. stObj → Publish.stObj, ID_WRITE → Publish.ID_WRITE.
6. Initialisierung: INIT → AR_SUBSCRIBE_Hz.INIT; AR_SUBSCRIBE_Hz.INITO → Publish.INIT; Publish.INITO → INITO und → AND_QO.REQ.
7. Qualität: QI → AR_SUBSCRIBE_Hz.QI und Publish.QI; AR_SUBSCRIBE_Hz.QO → AND_QO.IN1, Publish.QO → AND_QO.IN2; AND_QO.OUT → QO.

Technische Besonderheiten

- **HZ_RAW_OUT (Version 1.1)**: Der optionale Plug spiegelt den Rohwert (Hz) für eine Diagnose-Anzeige, ohne einen zweiten eigenständigen Remote-Subscribe desselben Knotens zu benötigen (FORTE meldet bei doppeltem Self-Subscribe Cannot add another one).
- **INIT/INITO und QI/QO**: Die Initialisierung wird durch die Kette INIT → AR_SUBSCRIBE_Hz → Publish → INITO gereicht; QO ist das AND der QO-Ausgänge beider Bausteine.
- Reine Verdrahtung vorhandener Bausteine, abgesehen von der AND_QO-Verknüpfung keine eigene Logik.

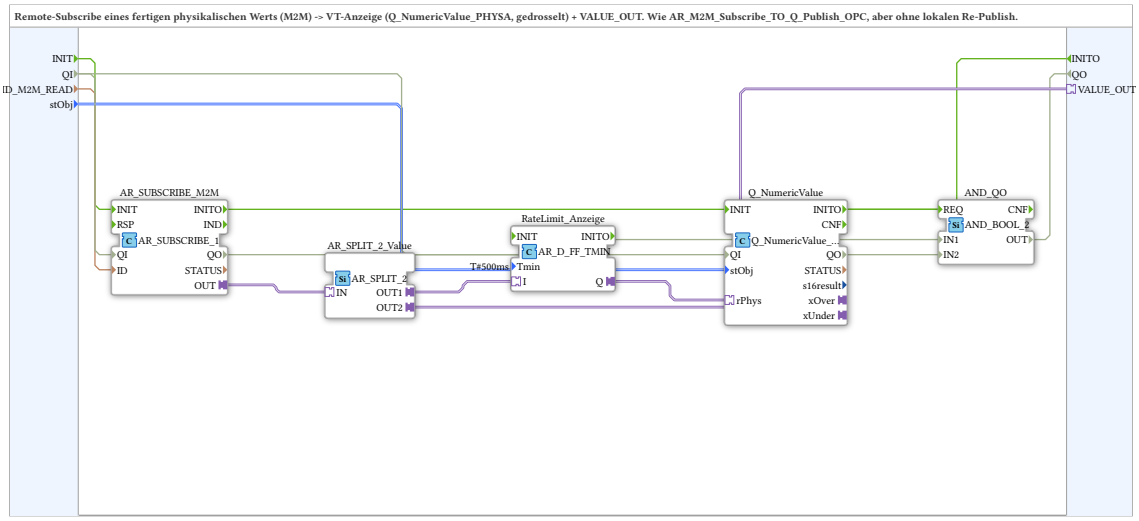
Anwendungsszenarien

- Anzeige und Publizieren von Geschwindigkeit bzw. Drehzahl aus einem Pulssensor-Frequenzwert (z. B. Aufnahme, Igelbänder, Sägeblatt)

Fazit

AR_HZ_TO_Q_Publish_OPC ersetzt die vierfach duplizierte Kette aus Subscribe, Skalierung und Publish durch einen einzigen parametrierbaren Baustein.

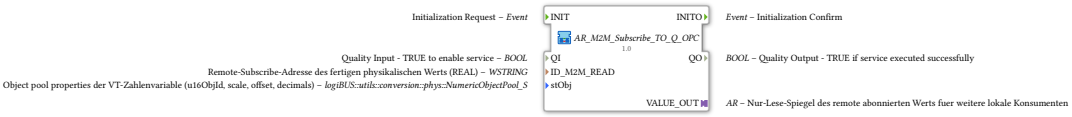
AR_M2M_Subscribe_TO_Q_OPC



• – ○ • – ○ • – ○ •

Einleitung

`AR_M2M_Subscribe_TO_Q_OPC` abonniert per Remote-Subscribe (M2M) einen fertigen physikalischen Wert (`REAL`) und zeigt ihn auf der VT an (`Q_NumericValue_PHYSA`, auf höchstens ein Update pro 500 ms gedrosselt). Zusätzlich steht der Wert als Nur-Lese-Spiegel am Plug `VALUE_OUT` für weitere lokale Konsumenten bereit. Der Baustein entspricht `AR_M2M_Subscribe_TO_Q_Publish_OPC`, jedoch ohne lokalen Re-Publish (kein `ID_WRITE`) – die schlanke Lib-Variante.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; startet die Kette `AR_SUBSCRIBE_M2M` ⟶ `Q_NumericValue`

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung; kommt von `Q_NumericValue.INITO`

Daten-Eingänge

Variable	Typ	Vorgabe	Kommentar
`QI`	BOOL	TRUE	Quality Input: Freigabe des Dienstes; wird an `AR_SUBSCRIBE_M2M` und `Q_NumericValue` weitergereicht.
`ID_M2M_READ`	WSTRING		Remote-Subscribe-Adresse des fertigen physikalischen Werts (`REAL`).
`stObj`	`logiBUS::utils::conversion::phys::NumericObjectPool_S`	`(u16ObjId := ID_NULL, r32Scale := 1.0, i32Offset := 0, u8Decimals := 0)`	Object-Pool-Eigenschaften der VT-Zahlenvariable (`u16ObjId`, Scale, Offset, Decimals).

Daten-Ausgänge

Variable	Typ	Vorgabe	Kommentar
QO	BOOL		Quality Output: UND aus AR_SUBSCRIBE_M2M.QO und Q_NumericValue.QO.

Adapter

Adapter	Typ	Richtung	Kommentar
VALUE_OUT	adapter::types::unidirectional::AR	Plug	Nur-Lese-Spiegel des remote abonnierten Werts für weitere lokale Konsumenten (ungedrosselt).

Verwendete Funktionsbausteine (FBs)

- **AR_SUBSCRIBE_M2M** (adapter::net::AR_SUBSCRIBE_1): Remote-Subscribe des Werts
- **AR_SPLIT_2_Value** (adapter::events::unidirectional::AR_SPLIT_2): teilt den Wert auf Anzeige-Zweig und VALUE_OUT
- **RateLimit_Anzeige** (adapter::events::unidirectional::AR_D_FF_TMIN, Tmin = T#500ms): drosselt nur die VT-Anzeige auf höchstens ein Update pro Tmin
- **Q_NumericValue** (isobus::UT::Q::Q_NumericValue_PHYSA): schreibt den Wert in die VT-Zahlenvariable
- **AND_QO** (iec61131::booleanOperators::AND_BOOL_2): verknüpft die beiden QO-Ausgänge

Programmablauf und Verbindungen

1. Initialisierung: INIT -> AR_SUBSCRIBE_M2M.INIT; AR_SUBSCRIBE_M2M.INITO -> Q_NumericValue.INIT; Q_NumericValue.INITO -> INITO und -> AND_QO.REQ.
2. Daten: ID_M2M_READ -> AR_SUBSCRIBE_M2M.ID; stObj -> Q_NumericValue.stObj.
3. Qualität: QI -> AR_SUBSCRIBE_M2M.QI und Q_NumericValue.QI; AR_SUBSCRIBE_M2M.QO -> AND_QO.IN1; Q_NumericValue.QO -> AND_QO.IN2; AND_QO.OUT -> QO.
4. Wertpfad: AR_SUBSCRIBE_M2M.OUT -> AR_SPLIT_2_Value.IN.
5. Anzeige-Zweig: AR_SPLIT_2_Value.OUT1 -> RateLimit_Anzeige.I; RateLimit_Anzeige.Q -> Q_NumericValue.rPhys.
6. Spiegel-Zweig: AR_SPLIT_2_Value.OUT2 -> VALUE_OUT.

Technische Besonderheiten

- Die Drosselung (T#500ms) wirkt nur auf dem Anzeige-Zweig; VALUE_OUT bleibt ungedrosselt wie im Original.
- Gegenüber AR_M2M_Subscribe_TO_Q_Publish_OPC entfallen der lokale Publish und ID_WRITE.

Anwendungsszenarien

- Anzeige eines von einer anderen Station gelieferten, bereits physikalisch skalierten Werts auf der VT, wenn kein lokaler Re-Publish benötigt wird.
- Weitergabe des abonnierten Werts über VALUE_OUT an weitere lokale Konsumenten.

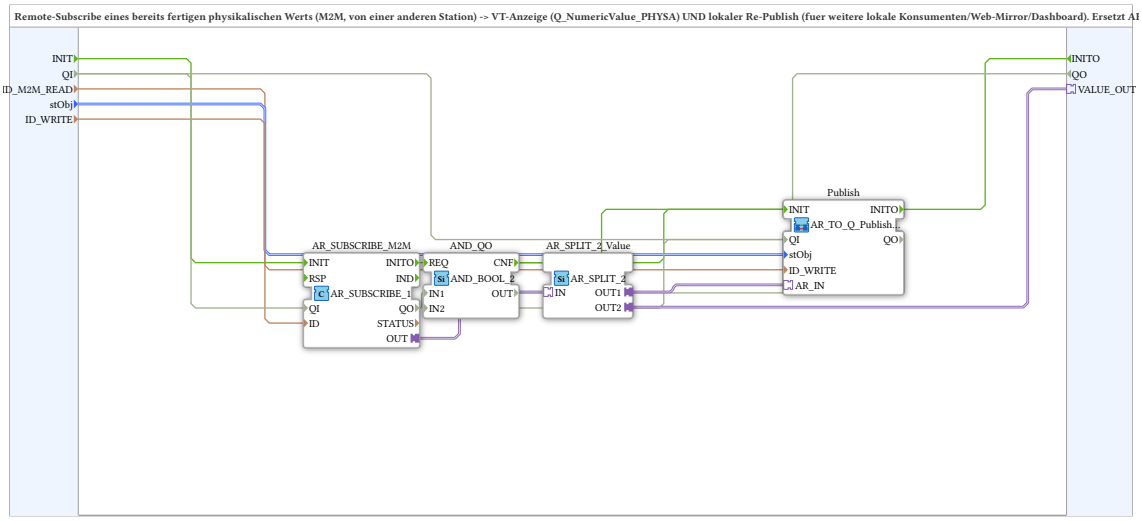
Zusammenfassung

AR_M2M_Subscribe_TO_Q_OPC abonniert einen fertigen physikalischen Wert remote, zeigt ihn gedrosselt auf der VT an und stellt ihn zusätzlich über VALUE_OUT bereit – ohne lokalen Re-Publish.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AR_M2M_Subscribe_TO_Q_Publish_OPC



• – ○ • – ○ • – ○ •

Einleitung

`AR_M2M_Subscribe_TO_Q_Publish_OPC` abonniert einen bereits fertig umgerechneten physikalischen Wert (REAL) per OPC-UA von einer anderen Station (M2M), zeigt ihn auf dem VT an (`Q_NumericValue_PHYSA`) und publiziert ihn lokal erneut, damit weitere lokale Konsumenten, Web-Mirror oder Dashboard ihn lesen können. Zusätzlich steht der Wert am Plug `VALUE_OUT` für lokale Adapter-Konsumenten bereit. Der Baustein ist die schlanke Nachfolgevariante von `AR_HZ_TO_Q_Publish_OPC` ohne eigene Skalierung, weil die Hz-zu-Einheit-Umrechnung inzwischen auf der Sensor-Station läuft (siehe `logiBUS_PI_Speed_Publish_OPC`).

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; wird an `AR_SUBSCRIBE_M2M.INIT` weitergereicht

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung; kommt von `Publish.INITO`

Daten-Eingänge

- **QI** (BOOL, Initialwert `TRUE`): Quality Input, schaltet den Dienst frei; wird an `AR_SUBSCRIBE_M2M.QI` und `Publish.QI` weitergereicht
- **ID_M2M_READ** (WSTRING): Remote-Subscribe-Adresse des fertigen physikalischen Werts (REAL), z. B. `FBs::Ventilsteuerung::Ventilsteuerung_STG1::RemoteSubStrings::AUFNAHME_GESCHWINDIGKEIT_KMH_M2M`
- **stObj** (`logiBUS::utils::conversion::phys::NumericObjectPool_S`): Object-Pool-Eigenschaften der VT-Zahlenvariable (`u16ObjId`, Skalierung, Offset, Nachkommastellen), z. B. `FBs::const::DefaultPool_Numeric::NumberVariable_X_N`
- **ID_WRITE** (WSTRING): lokale Publish-Adresse (REAL, physikalischer Wert) für Web-Mirror, Dashboard und Self-Subscribe weiterer lokaler Konsumenten

Daten-Ausgänge

- **QO** (BOOL): Quality Output, UND-Verknüpfung (`AND_QO`) der `QO`-Ausgänge von `AR_SUBSCRIBE_M2M` und `Publish`

Adapter

- **VALUE_OUT** (Plug, `adapter::types::unidirectional::AR`): Nur-Lese-Spiegel des remote abonnierten Werts für weitere lokale Konsumenten; vermeidet einen doppelten OPC-UA-Self-Subscribe desselben Remote-Knotens auf dieser Station

Funktionsweise

Das Netzwerk besteht aus vier Teilen:

- `AR_SUBSCRIBE_M2M` (`adapter::net::AR_SUBSCRIBE_1`, `QI` kommt vom Eingang `QI`): abonniert die Adresse `ID_M2M_READ` (`ID_M2M_READ` ⟶ `AR_SUBSCRIBE_M2M.ID`) und gibt den Wert als AR-Adapter aus.
- `AR_SPLIT_2_Value` (`adapter::events::unidirectional::AR_SPLIT_2`): verteilt den abonnierten Wert auf zwei Ziele (`AR_SUBSCRIBE_M2M.OUT` ⟶ `AR_SPLIT_2_Value.IN`).
- `Publish` (`MyLib::sys::AR_TO_Q_Publish_OPC`): zeigt den Wert am VT an und publiziert ihn lokal (`AR_SPLIT_2_Value.OUT1` ⟶ `Publish.AR_IN`, `stObj` ⟶ `Publish.stObj`, `ID_WRITE` ⟶ `Publish.ID_WRITE`).
- `AND_QO` (`iec61131::booleanOperators::AND_BOOL_2`): verknüpft `AR_SUBSCRIBE_M2M.QO` und `Publish.QO` zum Ausgang `QO`.

`AR_SPLIT_2_Value.OUT2` wird auf den Plug `VALUE_OUT` geführt.

Initialisierung und Qualität: `INIT` ⟶ `AR_SUBSCRIBE_M2M.INIT`, `AR_SUBSCRIBE_M2M.INITO` ⟶ `Publish.INIT`, `Publish.INITO` ⟶ `INITO` und `AND_QO.REQ`. `QI` wird an `AR_SUBSCRIBE_M2M.QI` und `Publish.QI` weitergereicht; `AR_SUBSCRIBE_M2M.QO` ⟶ `AND_QO.IN1`, `Publish.QO` ⟶ `AND_QO.IN2`, `AND_QO.OUT` ⟶ `QO`.

Technische Besonderheiten

- Kein `SCALE`/`AR_MUL_2` mehr: Der abonnierte Wert liegt bereits in der physikalischen Einheit vor
- Der `AR_SPLIT_2` vor `Publish.AR_IN` wurde eingefügt, statt `AR_SUBSCRIBE_M2M.OUT` direkt zu verdrahten, damit ein Adapter nie ohne Split verzweigt wird
- `VALUE_OUT` liefert den abonnierten Wert für Verbraucher wie Befüllautomatik oder Füllband-Erkennung, ohne einen zweiten Subscribe auf denselben Remote-Knoten anzulegen

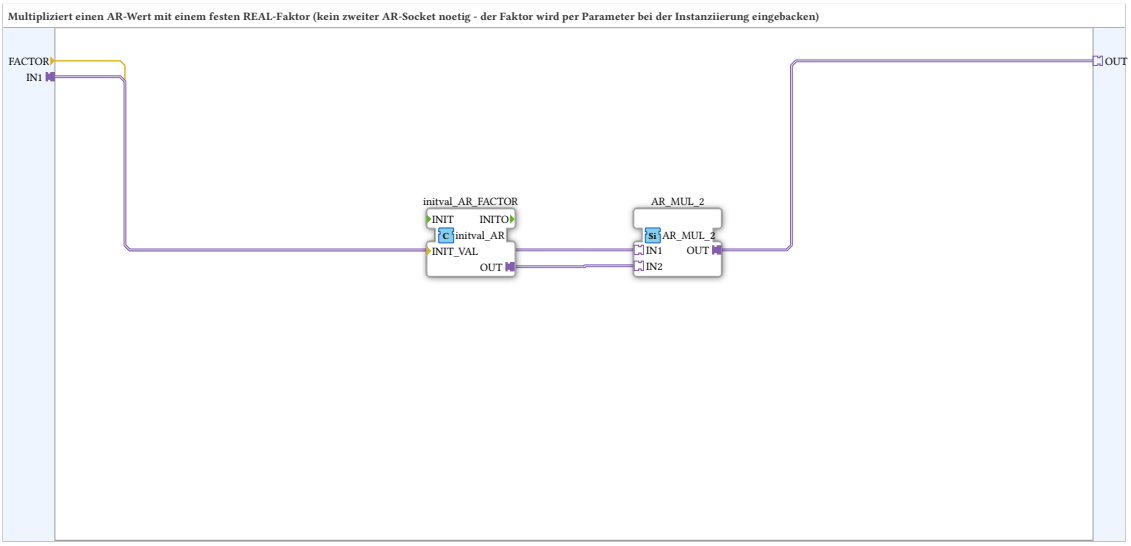
Anwendungsszenarien

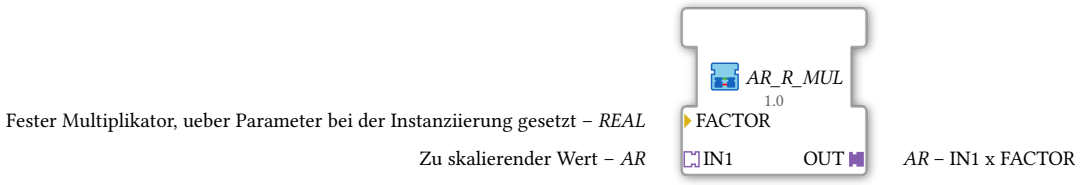
- Anzeige eines von einer anderen Station gemessenen Werts (z. B. Geschwindigkeit) am VT und gleichzeitige lokale Bereitstellung für Dashboard und Web-Client
- Weitergabe des Remote-Werts an lokale Regel- oder Automatik-Bausteine über `VALUE_OUT`

Fazit

Der Baustein bündelt Remote-Subscribe, VT-Anzeige und lokalen Re-Publish eines fertigen physikalischen Werts in einer SubApp und stellt den Wert zusätzlich als AR-Plug bereit.

AR_R_MUL





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AR_R_MUL** ist eine Subapplikation zur Multiplikation eines analogen Rohwertes (AR-Adapter) mit einem festen, bei der Instanziierung vorgegebenen REAL-Faktor. Er wurde entwickelt, um eine Skalierung von Analogwerten (z. B. Umrechnung von Rohwerten in physikalische Einheiten) ohne zusätzliche Verdrahtung eines zweiten variablen AR-Sockets zu ermöglichen. Der Faktor wird als Konfigurationsparameter in die Subapp eingebettet und bleibt während der Laufzeit konstant.

Die Subapp verwendet intern zwei Standard-Funktionsblöcke: einen `initval_AR`-Baustein, der den Faktor als konstanten AR-Wert erzeugt, und einen `AR_MUL_2`-Baustein, der die Multiplikation zweier AR-Werte ausführt. Somit wird eine einfache und effiziente Lösung für fixe Skalierungen bereitgestellt.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

- **FACTOR** (`REAL`): Der feste Multiplikationsfaktor. Er wird als Parameter bei der Instanziierung gesetzt und intern als Initialisierungswert für den `initval_AR`-Baustein verwendet. Standardwert: `REAL#1.0`.

Daten-Ausgänge

Keine.

Adapter

- **IN1** (`adapter::types::unidirectional::AR`, Socket): Der zu skalierende analoge Rohwert.
- **OUT** (`adapter::types::unidirectional::AR`, Plug): Das Ergebnis der Multiplikation (`IN1 * FACTOR`).

Funktionsweise

Die Subapp besitzt keinen internen Ereignis- oder Zustandsablauf; sie arbeitet rein datenflussorientiert. Beim Instanziieren wird der Wert des Daten-Eingangs `FACTOR` an den Initialisierungs-Eingang `INIT_VAL` des internen `initval_AR`-Bausteins übergeben. Dieser erzeugt einen konstanten AR-Wert mit dem übergebenen REAL-Faktor. Der Adapter-Eingang `IN1` (der Rohwert) wird direkt an den ersten Eingang `IN1` des Multiplizierers `AR_MUL_2` geführt. Der zweite Eingang `IN2` des Multiplizierers wird mit dem konstanten AR-Wert vom `initval_AR` gespeist. Das Ergebnis der Multiplikation wird am Ausgang `AR_MUL_2.OUT` abgegriffen und über den Adapter `OUT` nach außen geführt.

Die Verschaltung gewährleistet, dass immer `OUT = IN1 * FACTOR` gilt, wobei `FACTOR` fest in der Subapp-Konfiguration verankert ist.

Technische Besonderheiten

- **Feste Skalierung ohne zweites AR-Socket:** Im Gegensatz zu einem allgemeinen `AR_MUL` mit zwei variablen AR-Eingängen benötigt dieser Baustein nur einen AR-Eingang. Der zweite Faktor wird über einen REAL-Parameter gesetzt, wodurch die Verdrahtung im Gesamtsystem vereinfacht wird.
- **Verwendung von `initval_AR`:** Der interne `initval_AR`-Baustein wandelt den REAL-Wert in einen permanenten AR-Wert um, der als Multiplikator für den Multiplizierer dient.
- **Keine Laufzeitänderung des Faktors:** Der Multiplikationsfaktor kann während des Betriebs nicht dynamisch verändert werden; er ist bereits bei der Konfiguration der Instanz festgelegt.
- **Kompatibel mit IEC 61499-2:** Die Subapp folgt dem Standard für verteilte Automatisierungssysteme und ist in der 4diac-IDE einbindbar.

Zustandsübersicht

Da die Subapp keine ereignisgesteuerten Transitionen oder internen Zustände besitzt, existiert keine explizite Zustandsmaschine. Die Ausgangswerte hängen ausschließlich von den aktuellen Eingangswerten ab (kombinatorische Logik). Somit kann der Funktionsblock als **zustandslos** betrachtet werden.

Anwendungsszenarien

- **Skalierung von Analogsignalen:** Ein 4–20 mA-Signal wird in einen physikalischen Wert (z. B. Temperatur in °C) umgerechnet, indem ein konstanter Umrechnungsfaktor angewendet wird.
- **Einheitenumrechnung:** Umwandlung von Rohwerten in normierte Größen (z. B. von mV in V) mit festem Faktor.
- **Kalibrierung:** Kompensation einer bekannten, konstanten Verstärkungsabweichung eines Sensors.
- **Parameterierung in Fertigungsanlagen:** Festlegung einer produktspezifischen Skalierung, die bei der Projektierung einmalig konfiguriert wird.

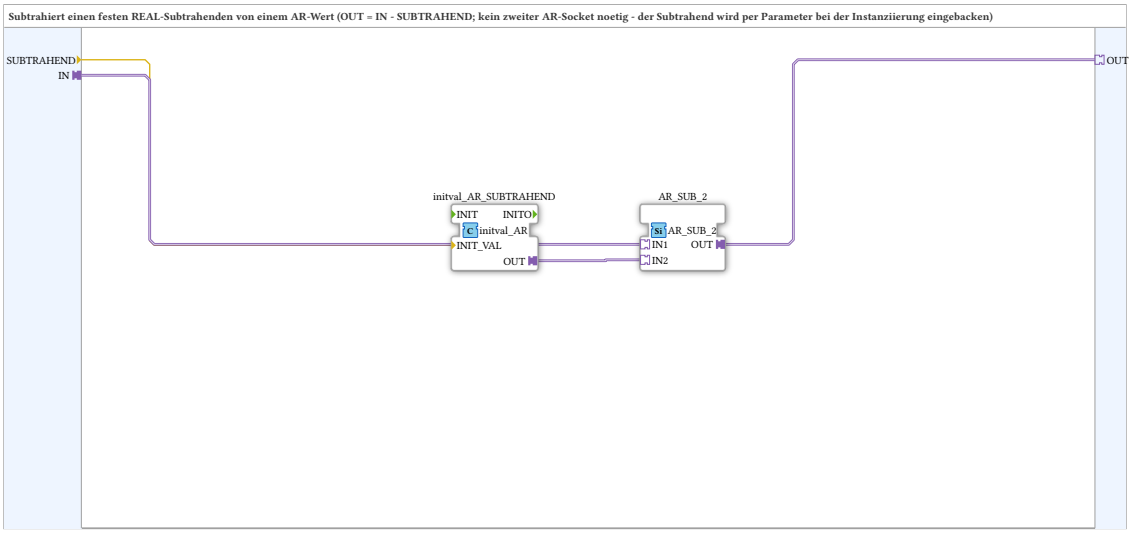
Vergleich mit ähnlichen Bausteinen

- **`AR_MUL` (Standard-Baustein):** Dieser besitzt zwei variable AR-Eingänge und erlaubt die Multiplikation zweier dynamischer analoger Werte. `AR_R_MUL` reduziert dies auf einen AR-Eingang und einen festen REAL-Faktor, was die Verdrahtung vereinfacht, aber die Flexibilität einschränkt.
- **`AR_SCALE` (falls vorhanden):** Ein Baustein, der oft eine Skalierung mit Offset und Bereichsumrechnung vornimmt; `AR_R_MUL` fokussiert sich auf eine reine Multiplikation ohne Offset.
- **Direkte Verwendung von `AR_MUL_2` mit konstantem Wert:** Dies erfordert eine zusätzliche Verdrahtung eines konstanten AR-Werts (z. B. über einen `initval_AR`), was der Benutzer manuell vornehmen müsste. Die Subapp kapselt diese Funktionalität und macht sie als eigenständigen Baustein verfügbar.

Fazit

Der Funktionsblock **AR_R_MUL** stellt eine kompakte und wiederverwendbare Lösung für feste Multiplikationen von analogen Rohwerten dar. Durch die Integration des konstanten Faktors in die Instanzparameter entfällt die Notwendigkeit eines zweiten adaptiven AR-Sockets, was die Projektierung vereinfacht und die Übersichtlichkeit erhöht. Die Verwendung standardisierter Bausteine im Inneren gewährleistet eine robuste und nach IEC 61499 konforme Funktionsweise. Damit eignet sich der Baustein hervorragend für alle Anwendungen, bei denen eine einmalige, unveränderliche Skalierung von Analogwerten benötigt wird.

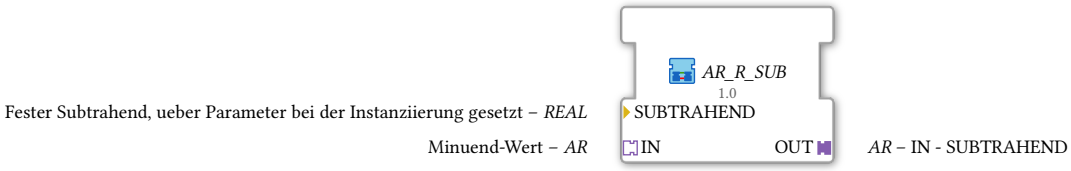
AR_R_SUB



• – ○ • – ○ • – ○ •

Einleitung

`AR_R_SUB` ist eine SubApp, die von einem `AR`-Wert einen festen `REAL`-Subtrahenden abzieht (`OUT = IN - SUBTRAHEND`). Der Subtrahend wird als Parameter bei der Instanziierung eingebacken, sodass kein zweiter `AR`-Socket verdrahtet werden muss. Sie ist das Subtraktions-Gegenstück zu `AR_R_MUL`.



Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

- **SUBTRAHEND** (`REAL`): Fester Subtrahend, über Parameter bei der Instanziierung gesetzt. Initialwert `REAL#0.0`.

Daten-Ausgänge

Keine.

Adapter

- **IN** (Socket, `adapter::types::unidirectional::AR`): Minuend-Wert
- **OUT** (Plug, `adapter::types::unidirectional::AR`): `IN - SUBTRAHEND`

Verwendete Funktionsbausteine (FBs)

- **`initval_AR_SUBTRAHEND`** (`adapter::types::unidirectional::AR::initval::initval_AR`): erzeugt aus `SUBTRAHEND` einen konstanten `AR`-Wert
- **`AR_SUB_2`** (`adapter::iec61131::arithmetic::AR_SUB_2`): subtrahiert zwei `AR`-Werte

Funktionsweise

- `SUBTRAHEND` ⟶ `initval_AR_SUBTRAHEND.INIT_VAL`
- `IN` ⟶ `AR_SUB_2.IN1` (Minuend)
- `initval_AR_SUBTRAHEND.OUT` ⟶ `AR_SUB_2.IN2` (Subtrahend)
- `AR_SUB_2.OUT` ⟶ `OUT`

Der Baustein besitzt keine eigenen Ereignisse; die Berechnung folgt dem Datenfluss des `AR`-Adapters.

Technische Besonderheiten

- Kein zweiter `AR`-Socket nötig, der Subtrahend ist fest in die Instanz eingebacken
- Keine Laufzeitänderung des Subtrahenden vorgesehen
- Reine Verdrahtung vorhandener Bausteine, keine eigene Logik

Anwendungsszenarien

- Konstanter Offset-Abzug von einem skalierten oder physikalischen `AR`-Wert

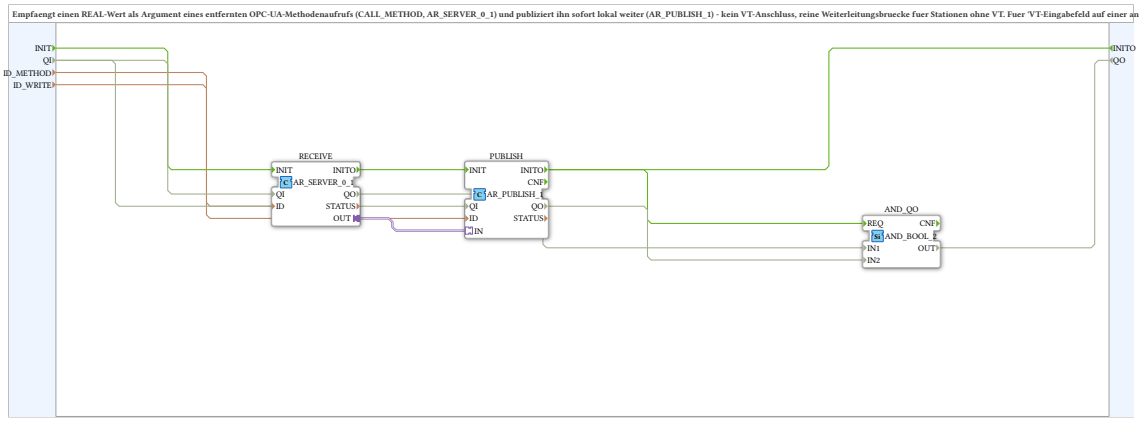
Zusammenfassung

`AR_R_SUB` subtrahiert einen fest parametrierten `REAL`-Wert von einem `AR`-Signal.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

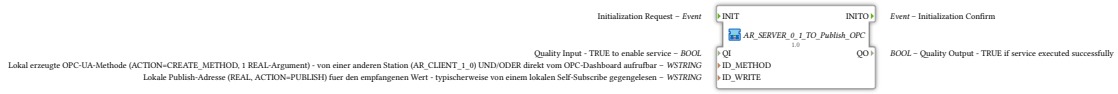
AR_SERVER_0_1_TO_Publish_OPC



• – ○ • – ○ • – ○ •

Einleitung

`AR_SERVER_0_1_TO_Publish_OPC` empfängt einen `REAL`-Wert als Argument eines entfernten OPC-UA-Methodenaufrufs (`AR_SERVER_0_1`) und publiziert ihn sofort lokal weiter (`AR_PUBLISH_1`). Der Baustein hat keinen VT-Anschluss und ist eine reine Weiterleitungsbrücke für Stationen ohne VT: Ein VT-Eingabefeld auf einer anderen Station oder ein Web-Dashboard rufen dieselbe Methode auf – und zwar nur bei echter Eingabe. Typisches Beispiel sind Druck-Referenzwerte (B20).



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: AR_SERVER_0_1_TO_Publish_OPC

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **RECEIVE**: `adapter::net::AR_SERVER_0_1` (`QI` von außen) – erzeugt die OPC-UA-Methode und liefert das empfangene `REAL`-Argument am Adapter `OUT`
  - **PUBLISH**: `adapter::net::AR_PUBLISH_1` (`QI` von außen) – publiziert den Wert unter der lokalen Adresse
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_2` – verknüpft die `QO` von `RECEIVE` und `PUBLISH`
- **Funktionsweise**: Der Wert aus dem Methodenaufruf wird ohne Zwischenlogik direkt an den Publisher weitergegeben.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; geht an `RECEIVE.INIT`

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierung abgeschlossen; kommt von `PUBLISH.INITO`

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input - TRUE aktiviert den Dienst; geht an `RECEIVE.QI` und `PUBLISH.QI`
- **ID_METHOD** (`WSTRING`): Lokal erzeugte OPC-UA-Methode (`ACTION=CREATE_METHOD`, 1 `REAL`-Argument) – von einer anderen Station (`AR_CLIENT_1_0`) und/oder direkt vom OPC-Dashboard aufrufbar
- **ID_WRITE** (`WSTRING`): Lokale Publish-Adresse (`REAL`, `ACTION=PUBLISH`) für den empfangenen Wert – typischerweise von einem lokalen Self-Subscribe gegengelesen

Daten-Ausgänge

- **QO** (BOOL): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde (AND aus RECEIVE.QO und PUBLISH.QO)

Programmablauf und Verbindungen

- ID_METHOD ⟶ RECEIVE.ID
- ID_WRITE ⟶ PUBLISH.ID
- INIT ⟶ RECEIVE.INIT
- RECEIVE.INITO ⟶ PUBLISH.INIT (Der Publisher wird initialisiert, sobald die Methode angelegt ist)
- PUBLISH.INITO ⟶ INITO und ⟶ AND_QO.REQ
- QI ⟶ RECEIVE.QI und PUBLISH.QI
- RECEIVE.QO ⟶ AND_QO.IN1, PUBLISH.QO ⟶ AND_QO.IN2, AND_QO.OUT ⟶ QO
- RECEIVE.OUT ⟶ PUBLISH.IN (Adapterverbindung des REAL-Wertes)

Technische Besonderheiten

- Kein VT-Anschluss: gedacht für Sensor-Stationen ohne eigene VT (z. B. STG4), die dennoch editierbare Referenzwerte von einer anderen Station (VT) oder vom Dashboard entgegennehmen sollen
- Die Methode wird nur bei echter Eingabe aufgerufen, es gibt keinen zyklischen Abgleich
- Version 1.0 (2026-09-23): bündelt AR_SERVER_0_1 (Empfang) und AR_PUBLISH_1 (lokale Weiterleitung)

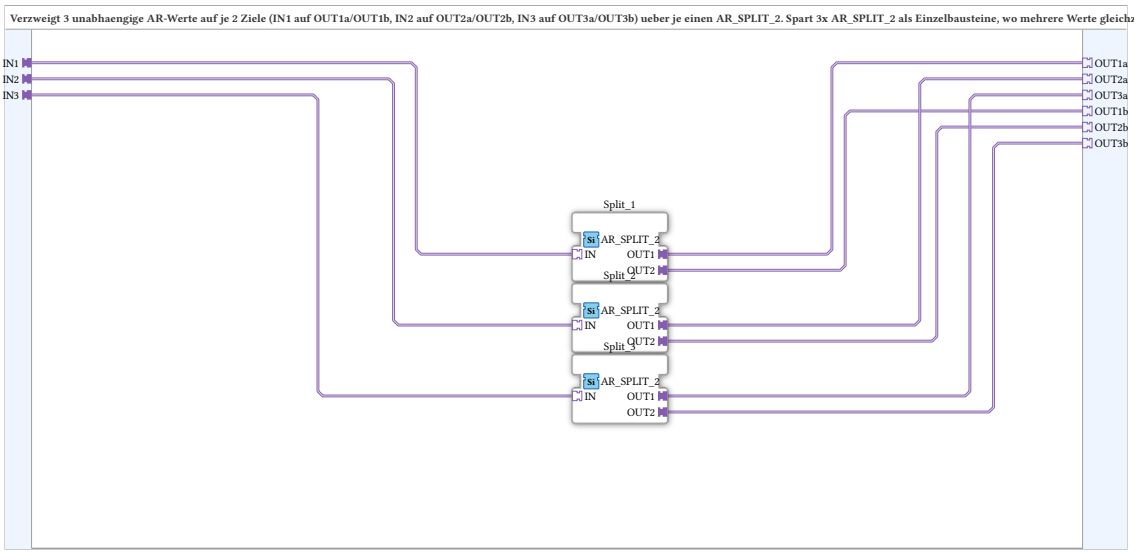
Anwendungsszenarien

- Druck-Referenzwerte (B20) einer Station ohne VT, die von einer anderen Station oder vom Web-Dashboard gesetzt werden
- Allgemein: Entgegennahme eines REAL-Sollwerts per OPC-UA-Methodenaufruf und lokale Bereitstellung als Publish-Wert

Zusammenfassung

AR_SERVER_0_1_TO_Publish_OPC kapselt AR_SERVER_0_1 und AR_PUBLISH_1 zu einer Weiterleitungsbrücke, die einen per OPC-UA-Methode empfangenen REAL-Wert lokal weiter publiziert.

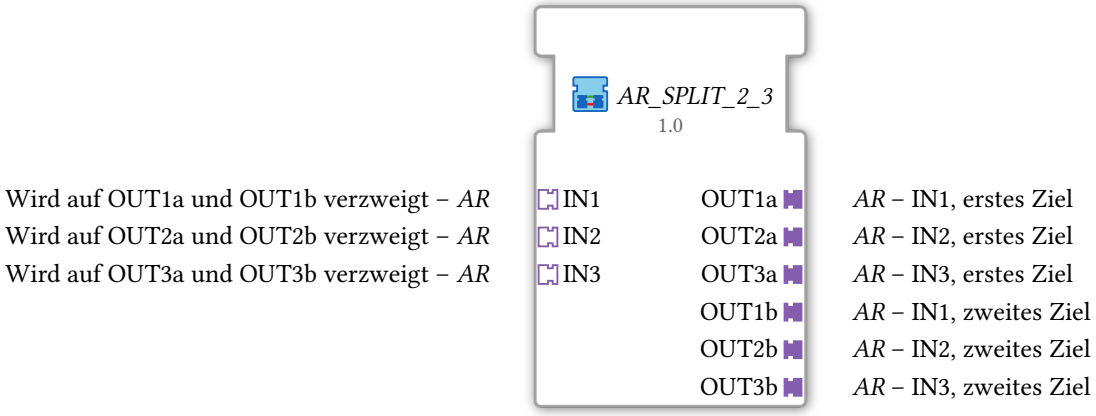
AR_SPLIT_2_3



• – ○ • – ○ • – ○ •

Einleitung

AR_SPLIT_2_3 verzweigt **drei unabhängige AR-Werte** auf je zwei Ziele (IN1 ⟶ OUT1a/OUT1b, IN2 ⟶ OUT2a/OUT2b, IN3 ⟶ OUT3a/OUT3b) über je einen AR_SPLIT_2. Sie spart drei einzelne AR_SPLIT_2-Bausteine, wenn mehrere Werte gleichzeitig an zwei unabhängige Verbraucher verteilt werden müssen (z. B. Schwelle/Totzone/Hysterese an Regel-Logik und an Positions-/Anzeige-Logik, wie in der Befüllautomatik).



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden – Ereignisse kommen über die Adapter-Sockets/-Plugs

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

Alle Adapter sind unidirektional vom Typ `adapter::types::unidirectional::AR`.

- **IN1** (Socket): wird auf `OUT1a` und `OUT1b` verzweigt
- **IN2** (Socket): wird auf `OUT2a` und `OUT2b` verzweigt
- **IN3** (Socket): wird auf `OUT3a` und `OUT3b` verzweigt
- **OUT1a**, **OUT2a**, **OUT3a** (Plugs): jeweils erstes Ziel von `IN1`/`IN2`/`IN3`
- **OUT1b**, **OUT2b**, **OUT3b** (Plugs): jeweils zweites Ziel von `IN1`/`IN2`/`IN3`

Verwendete Funktionsbausteine (FBs)

- **Split_1**, **Split_2**, **Split_3**: je `adapter::events::unidirectional::AR_SPLIT_2`

Programmablauf und Verbindungen

- `IN1` ⟶ `Split_1.IN`; `Split_1.OUT1` ⟶ `OUT1a`; `Split_1.OUT2` ⟶ `OUT1b`
- `IN2` ⟶ `Split_2.IN`; `Split_2.OUT1` ⟶ `OUT2a`; `Split_2.OUT2` ⟶ `OUT2b`
- `IN3` ⟶ `Split_3.IN`; `Split_3.OUT1` ⟶ `OUT3a`; `Split_3.OUT2` ⟶ `OUT3b`

Die drei Zweige sind vollständig voneinander unabhängig.

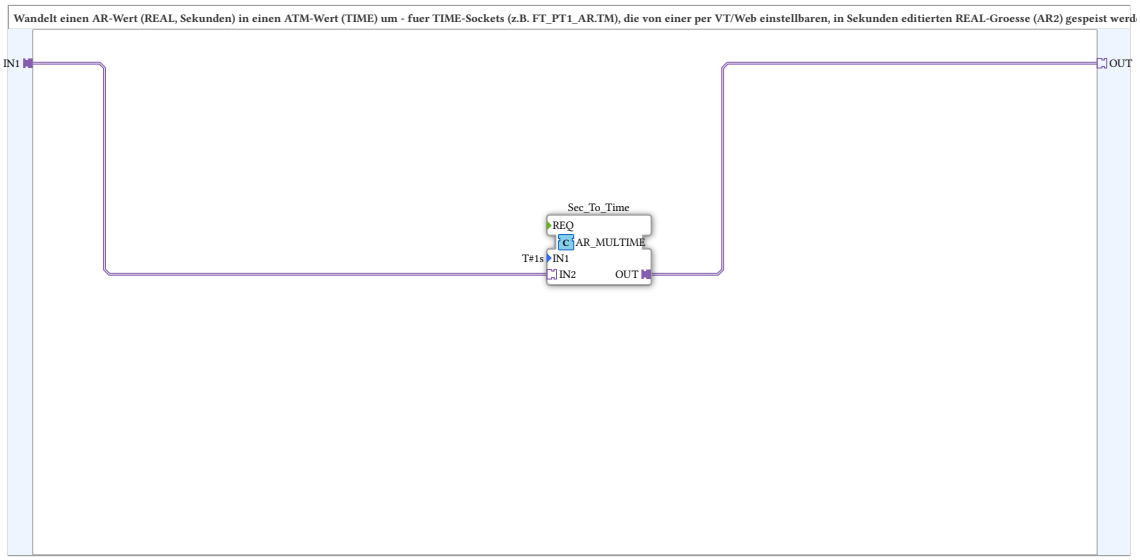
Anwendungsszenarien

- Verteilung mehrerer Parameter (Schwelle, Totzone, Hysterese) an zwei unabhängige Verbraucher gleichzeitig

Fazit

`AR_SPLIT_2_3` bündelt drei `AR_SPLIT_2` in einer SubApp und hält so das Netzwerk des aufrufenden Bausteins übersichtlich.

AR_S_TO_ATM

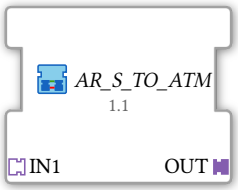


• – ○ • – ○ • – ○ •

Einleitung

`AR_S_TO_ATM` wandelt einen `AR`-Wert (REAL, Sekunden) in einen `ATM`-Wert (TIME) um. Er wird für TIME-Sockets (z. B. `FT_PT1_AR.TM`) benötigt, die von einer per VT oder Web einstellbaren, in Sekunden editierten REAL-Größe (AR2) gespeist werden sollen.

Zeit in Sekunden (REAL) – *AR*



ATM – IN1, umgerechnet von Sekunden (REAL) auf TIME

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **IN1** (Socket, `adapter::types::unidirectional::AR`): Zeit in Sekunden (REAL)
- **OUT** (Plug, `adapter::types::unidirectional::ATM`): `IN1`, umgerechnet von Sekunden (REAL) auf TIME

Funktionsweise

Intern arbeitet eine einzige Instanz `Sec_To_Time` des Typs `adapter::iec61131::arithmetic::AR_MULTIME` mit dem Parameter `IN1 = T#1s`. Der Socket `IN1` wird auf `Sec_To_Time.IN2` geführt, das Ergebnis `Sec_To_Time.OUT` auf den Plug `OUT`. Damit gilt `OUT = T#1s * IN1`, also echte IEC-61131-3-TIME-Arithmetik: Ein Eingang von `2.5` ergibt `T#2s500ms`.

Technische Besonderheiten

- Frühere Variante (`AR_R_MUL` × 1000 ⟶ `AR_TO_ADI` ⟶ `ADI_TO_ATM`) wurde ersetzt, weil sie auf einer nicht existierenden Funktion `iec61131::conversion::F_DINT_TO_TIME` beruhte
- Zustandslos; Ereignisse laufen über die Adapter (`E1` von `IN1` löst die Berechnung aus)

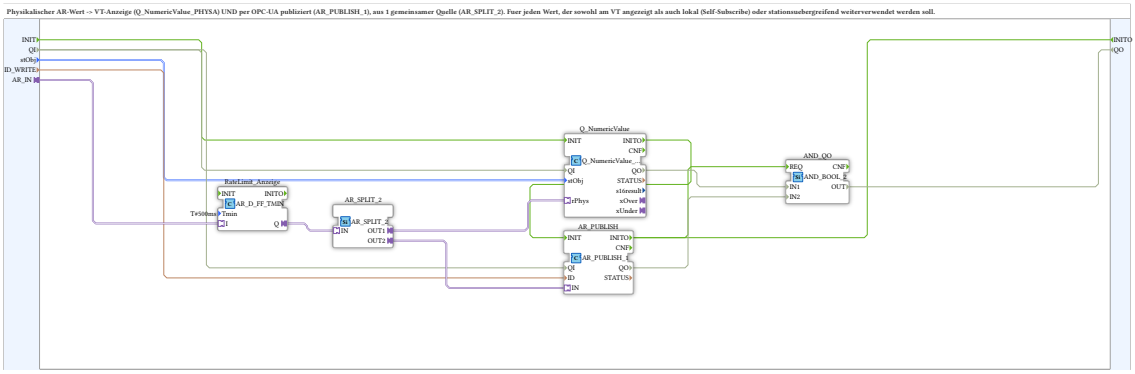
Anwendungsszenarien

- Einstellbare PT1-Filterzeit der Puls-Frequenz-Sensoren (siehe `logiBUS_PI_Speed_Publish_OPC`): in Sekunden editiert, als TIME an `FT_PT1_AR.TM` geführt
- Jede Stelle, an der eine in Sekunden eingegebene Größe an einen TIME-Socket übergeben werden muss

Fazit

`AR_S_TO_ATM` ist eine kleine Adapter-SubApp, die Sekunden-Werte per `T#1s * Sekunden` in TIME umrechnet.

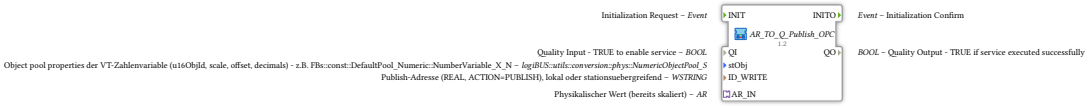
AR_TO_Q_Publish_OPC



• – ○ • – ○ • – ○ •

Einleitung

`AR_TO_Q_Publish_OPC` führt einen physikalischen AR-Wert aus **einer gemeinsamen Quelle** sowohl auf eine VT-Zahlenanzeige (`Q_NumericValue_PHYSA`) als auch auf einen OPC-UA-Publish (`AR_PUBLISH_1`). Sie ist für Werte gedacht, die am VT angezeigt und zusätzlich lokal (Self-Subscribe) oder stationsübergreifend weiterverwendet werden sollen. Herausgelöst aus `TECU_Speeds/TECU_RPTO`, wo die Kombination `AR_SPLIT_2` + `Q_NumericValue_PHYSA` + `AR_PUBLISH_1` häufig vorkam.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung

Daten-Eingänge

- **stObj** (`logiBUS::utils::conversion::phys::NumericObjectPool_S`): Objekt-Pool-Eigenschaften der VT-Zahlenvariable (`u16ObjId`, Skalierung, Offset, Nachkommastellen), z. B. `FBs::const::DefaultPool_Numeric::NumberVariable_X_N`
- **ID_WRITE** (`WSTRING`): Publish-Adresse (REAL, `ACTION=PUBLISH`), lokal oder stationsübergreifend
- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input – TRUE, um den Dienst (VT-Anzeige und Publish) freizugeben

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde (UND aus `Q_NumericValue.QO` und `AR_PUBLISH.QO`)

Adapter

- **AR_IN** (Socket, `adapter::types::unidirectional::AR`): physikalischer Wert (bereits skaliert)

Verwendete Funktionsbausteine (FBs)

- **`RateLimit_Anzeige`** (`adapter::events::unidirectional::AR_D_FF_TMIN`, `Tmin = T#500ms`): begrenzt die Update-Rate
- **`AR_SPLIT_2`** (`adapter::events::unidirectional::AR_SPLIT_2`): verteilt den Wert auf zwei Ziele
- **`Q_NumericValue`** (`isobus::UT::Q::Q_NumericValue_PHYSA`): schreibt den physikalischen Wert in die VT-Zahlenvariable
- **`AR_PUBLISH`** (`adapter::net::AR_PUBLISH_1`): publiziert den Wert per OPC-UA
- **`AND_QO`** (`iec61131::booleanOperators::AND_BOOL_2`): verknüpft die `QO`-Ausgänge von `Q_NumericValue` und `AR_PUBLISH` zum `QO` der SubApp

Funktionsweise

- `AR_IN` ⟶ `RateLimit_Anzeige.I`; `RateLimit_Anzeige.Q` ⟶ `AR_SPLIT_2.IN`
- `AR_SPLIT_2.OUT1` ⟶ `Q_NumericValue.rPhys` (VT-Anzeige); `stObj` ⟶ `Q_NumericValue.stObj`
- `AR_SPLIT_2.OUT2` ⟶ `AR_PUBLISH.IN`; `ID_WRITE` ⟶ `AR_PUBLISH.ID`
- `INIT` ⟶ `Q_NumericValue.INIT`; `Q_NumericValue.INITO` ⟶ `AR_PUBLISH.INIT`; `AR_PUBLISH.INITO` ⟶ `INITO` (und ⟶ `AND_QO.REQ`)
- `QI` ⟶ `Q_NumericValue.QI` und `QI` ⟶ `AR_PUBLISH.QI`
- `Q_NumericValue.QO` ⟶ `AND_QO.IN1`; `AR_PUBLISH.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`

Technische Besonderheiten

- **Ratenbegrenzung vor dem Split (Absicht):** Seit v1.2 sitzt `RateLimit_Anzeige` vor `AR_SPLIT_2` und drosselt damit **beide** Zweige (VT-Anzeige und Publish/Self-Subscribe-Konsumenten wie `CTRL_PID_AR.AR_ACT`) auf höchstens ein Update je 500 ms. In v1.1 war die Drossel nur im VT-Anzeige-Zweig. Anlass war ein träges Verhalten bei vielen gleichzeitig schnell wechselnden Anzeigen (vermutete VT-Bus-Last).
- Ein Adapter darf nicht auf zwei Ziele fan-outen, daher der explizite `AR_SPLIT_2`.

Anwendungsszenarien

- Drehzahl-, Geschwindigkeits- oder andere Messwerte, die am VT angezeigt und zugleich lokal oder an andere Stationen veröffentlicht werden

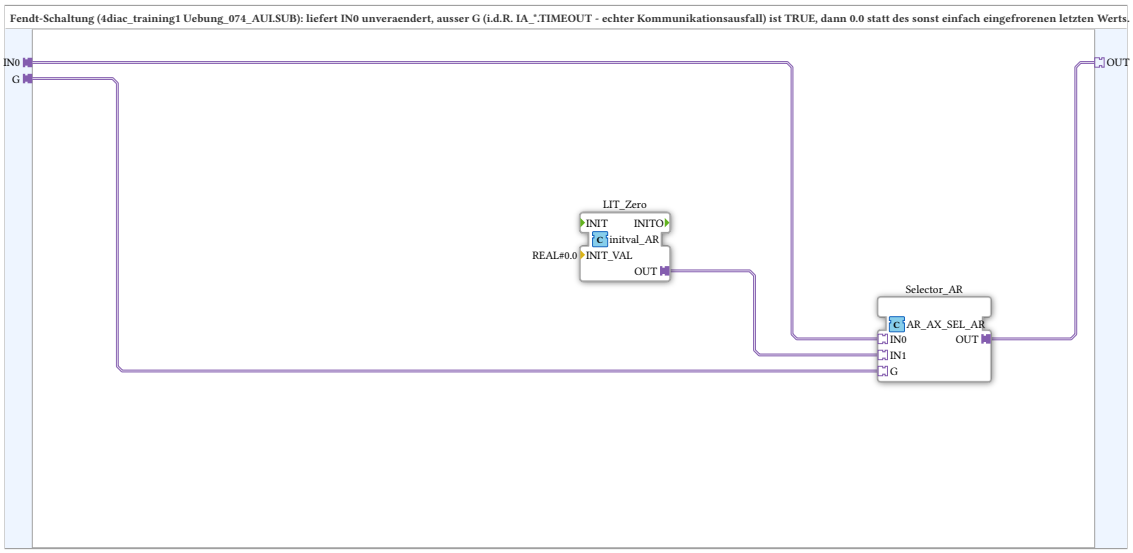
Zusammenfassung

`AR_TO_Q_Publish_OPC` bündelt Ratenbegrenzung, Aufteilung, VT-Anzeige und OPC-UA-Publish eines `AR`-Wertes in einer SubApp; `INIT`/`INITO` werden durch beide Dienste gekettet, `QO` ist das UND beider Dienste.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AR_Timeout_Zero

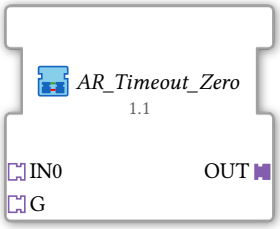


• – ○ • – ○ • – ○ •

Einleitung

`AR_Timeout_Zero` liefert den `REAL`-Wert `IN0` unverändert durch, außer das `AX`-Signal `G` (in der Regel `IA_*.TIMEOUT`, ein echter Kommunikationsausfall) ist `TRUE`: Dann wird `0.0` ausgegeben statt des sonst einfach eingefrorenen letzten Werts. Die Schaltung stammt aus der Fendt-Übung `Uebung_074_AUI`.

Live-Wert, z.B. FIELDBUS_*_SCALED.OUT – *AR*
TRUE = Timeout, liefert 0.0 statt IN0 – *AX*



AR – IN0, oder 0.0 bei G=TRUE

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: AR_Timeout_Zero

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **LIT_Zero**: `adapter::types::unidirectional::AR::initval::initval_AR` mit `INIT_VAL=0.0` – liefert die Konstante `0.0`
  - **Selector_AR**: `adapter::iec61131::selection::AR_AX_SEL_AR` – wählt anhand von `G` zwischen `IN0` und `IN1`
- **Funktionsweise**: Der Selektor gibt bei `G=FALSE` den Live-Wert `IN0`, bei `G=TRUE` die Konstante `0.0` aus.

Schnittstellenstruktur

Ereignis-Eingänge / Ereignis-Ausgänge

Keine – die SubApp hat keine eigenen Ereignis-Ports; Ereignisse laufen über die Adapter.

Daten-Eingänge / Daten-Ausgänge

Keine direkten Daten-Ports.

Adapter

- **IN0** (Socket): `adapter::types::unidirectional::AR` – Live-Wert, z. B. `FIELDBUS_*_SCALED.OUT`
- **G** (Socket): `adapter::types::unidirectional::AX` – `TRUE` = Timeout, liefert `0.0` statt `IN0`
- **OUT** (Plug): `adapter::types::unidirectional::AR` – `IN0`, oder `0.0` bei `G=TRUE`

Programmablauf und Verbindungen

- `IN0` ⟶ `Selector_AR.IN0`
- `LIT_Zero.OUT` ⟶ `Selector_AR.IN1`
- `G` ⟶ `Selector_AR.G`
- `Selector_AR.OUT` ⟶ `OUT`

Technische Besonderheiten

- Der Baustein ersetzt eine oft wiederkehrende Verdrahtung (`AR_AX_SEL_AR` mit `initval_AR=0`), die zuvor in den TECU-Bausteinen (`TECU_RPTO`, `TECU_FPTO`, `TECU_COGSOG`, `TECU_GBSD`, `TECU_WBSD`, `TECU_VDS`, `TECU_FHS`, `TECU_RHS`, `TECU_MSS`) jeweils einzeln aufgebaut war
- Version 1.1 (2026-09-16): nach `MyLib_AX-1.0.0::sys` verschoben, da generisch und nicht TECU-spezifisch

Anwendungsszenarien

- Nullsetzen eines Messwerts bei Kommunikationsausfall (Timeout), statt den letzten Wert einzufrieren
- Allgemein: Ersatzwert `0.0` für einen `AR`-Wert, gesteuert durch ein `AX`-Signal

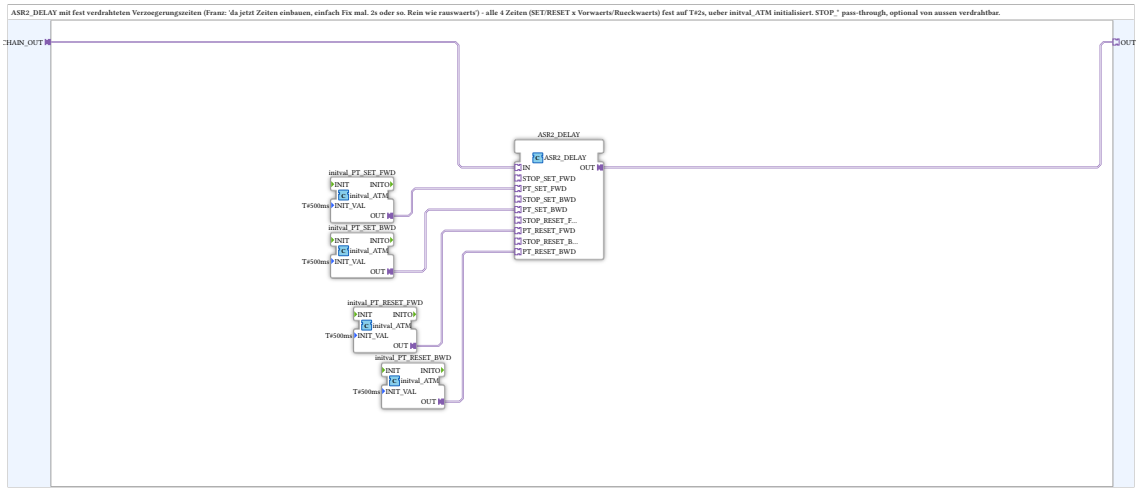
Zusammenfassung

`AR_Timeout_Zero` kapselt einen `AR`-Selektor mit Konstante `0.0` zu einem wiederverwendbaren Baustein, der bei Timeout den Wert auf Null setzt.

🛠️ Zugehörige Übungen

- Uebung_070d_AUI
- Uebung_079a_AUI

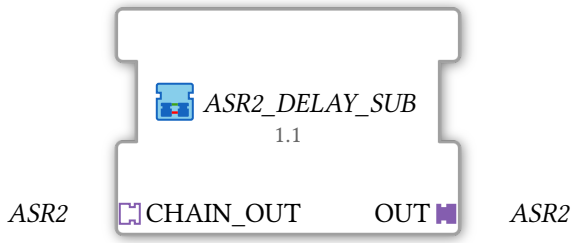
ASR2_DELAY_SUB



• – ○ • – ○ • – ○ •

Einleitung

`ASR2_DELAY_SUB` ist `ASR2_DELAY` mit fest verdrahteten Verzögerungszeiten: Die vier Zeiten (Set und Reset, jeweils vorwärts und rückwärts) werden über `initval_ATM`-Instanzen vorbelegt, sodass der Baustein ohne eigene Zeitverdrahtung als Verzögerungsglied in eine ASR2-Kette eingehängt werden kann.



Schnittstellenstruktur

Ereignis-Eingänge
Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge.

Ereignis-Ausgänge
Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge
Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge
Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter
- **CHAIN_OUT** (Socket, `adapter::types::bidirectional::ASR2`): Eingang der Kette; auf `ASR2_DELAY.IN` geführt
- **OUT** (Plug, `adapter::types::bidirectional::ASR2`): verzögerte Kette; kommt von `ASR2_DELAY.OUT`

Funktionsweise
Das Netzwerk besteht aus einer `ASR2_DELAY`-Instanz (`adapter::events::bidirectional::ASR2_DELAY`) und vier `initval_ATM`-Instanzen (`adapter::types::unidirectional::ATM::initval::initval_ATM`):

- `CHAIN_OUT` ⟶ `ASR2_DELAY.IN` und `ASR2_DELAY.OUT` ⟶ `OUT`
- `initval_PT_SET_FWD.OUT` ⟶ `ASR2_DELAY.PT_SET_FWD`
- `initval_PT_SET_BWD.OUT` ⟶ `ASR2_DELAY.PT_SET_BWD`
- `initval_PT_RESET_FWD.OUT` ⟶ `ASR2_DELAY.PT_RESET_FWD`
- `initval_PT_RESET_BWD.OUT` ⟶ `ASR2_DELAY.PT_RESET_BWD`

Alle vier `initval_ATM`-Instanzen sind mit `INIT_VAL = T#500ms` parametriert.

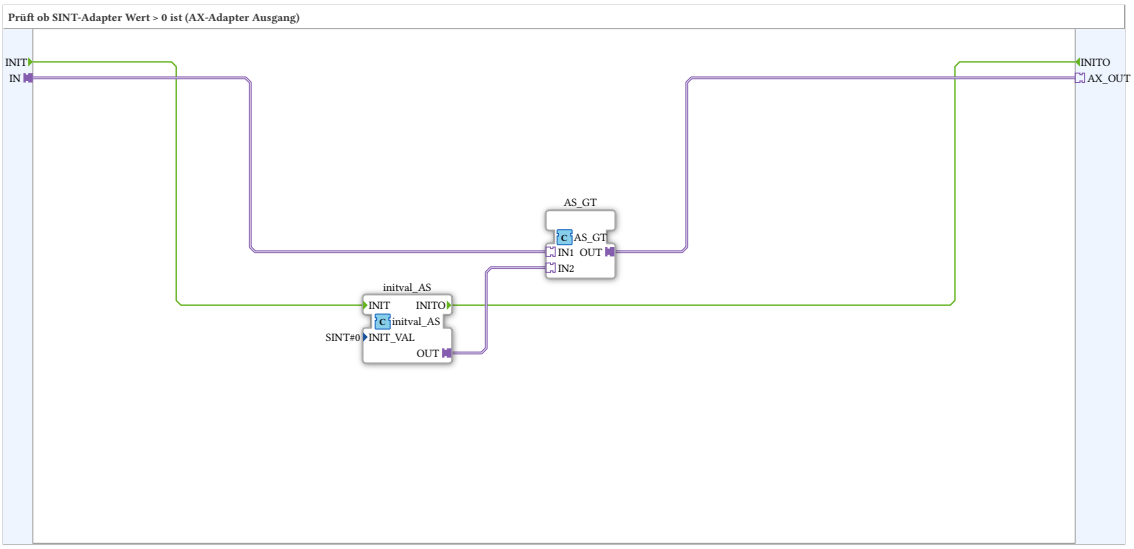
Technische Besonderheiten
- Die Zeiten sind Platzhalter-Festwerte ohne funktionale Vorgabe und können bei Bedarf später parametrierbar gemacht werden; der Typkommentar nennt als Beispiel `T#2s`, die Instanzen sind aktuell auf `T#500ms` gesetzt
- Die Schnittstelle besteht nur aus dem Eingang `CHAIN_OUT` und dem Ausgang `OUT`; die `STOP_*`-Sockets von `ASR2_DELAY` bleiben in dieser SubApp unverdrahtet
- Zustandslos auf SubApp-Ebene; der Zustand (laufende Timer) liegt in `ASR2_DELAY`

Anwendungsszenarien
- Zeitversetzte Ansteuerung von Ausgängen in einer Set/Reset-Kette (z. B. Bänder-Sequenzierung), wenn feste Verzögerungen genügen

⚖️ Vergleich mit ähnlichen Bausteinen
Für frei einstellbare Zeiten wird `ASR2_DELAY` direkt mit eigenen `ATM`-Quellen beschaltet; `ASR2_DELAY_SUB` kapselt dagegen die Festwerte. Für ein einzelnes Ereignis je Richtung siehe `AE2_DELAY`.

Fazit
`ASR2_DELAY_SUB` ist ein Einhänge-Verzögerungsglied für ASR2-Ketten mit fest vorbelegten Zeiten.

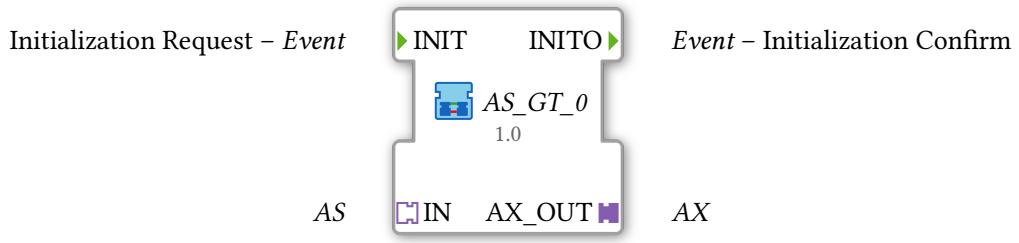
AS_GT_0



• – ◦ • – ◦ • – ◦ •

Einleitung

`AS_GT_0` prüft, ob ein SINT-Adapter-Wert größer als 0 ist, und gibt das Ergebnis als boolesches AX-Adapter-Signal aus. Typischer Eingangswert ist ein vorzeichenbehafteter 8-Bit-Wert. Der Baustein ist für jeden Zahlentyp gleich aufgebaut (`AS`, `AI`, `ADI`, `ALI`, `AUS`, `AUI`, `AUDI`, `AULI`, `AR`, `ALR`).



Schnittstelle

- **Ereignis-Eingang**: `INIT` (Initialisierung)
- **Ereignis-Ausgang**: `INITO` (Initialisierung abgeschlossen, kommt vom internen `initval_AS`)
- **Sockel**: `IN` (`AS`) — der Wert, der gegen 0 verglichen wird
- **Stecker**: `AX_OUT` (`AX`) — TRUE, wenn `IN > 0`

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: AS_GT_0

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **initval_AS**: `adapter::types::unidirectional::AS::initval::initval_AS` — liefert die feste Vergleichskonstante `SINT#0` als Adapter-Wert (`INIT_VAL=SINT#0`). Ein Adapter-Sockel kann die Konstante nicht als Parameter bekommen, deshalb wird sie über diesen Baustein eingespeist.
  - **AS_GT**: `adapter::iec61131::comparison::AS_GT` — adapter-nativer Vergleich `IN1 > IN2`.
- **Funktionsweise**: `IN` ⟶ `AS_GT.IN1`; Konstante 0 ⟶ `AS_GT.IN2`; Ergebnis ⟶ `AX_OUT`.

Programmablauf und Verbindungen

1. `INIT` ⟶ `initval_AS.INIT`; `initval_AS.INITO` ⟶ `INITO`.
2. `IN` (Adapter) ⟶ `AS_GT.IN1`.
3. `initval_AS.OUT` (Konstante `SINT#0`) ⟶ `AS_GT.IN2`.
4. `AS_GT.OUT` ⟶ `AX_OUT` (Adapter).

Anwendungsszenarien

- Umwandlung von ein vorzeichenbehafteter 8-Bit-Wert in ein einfaches boolesches Adapter-Signal für VT-Statusanzeigen oder Freigabelogik.

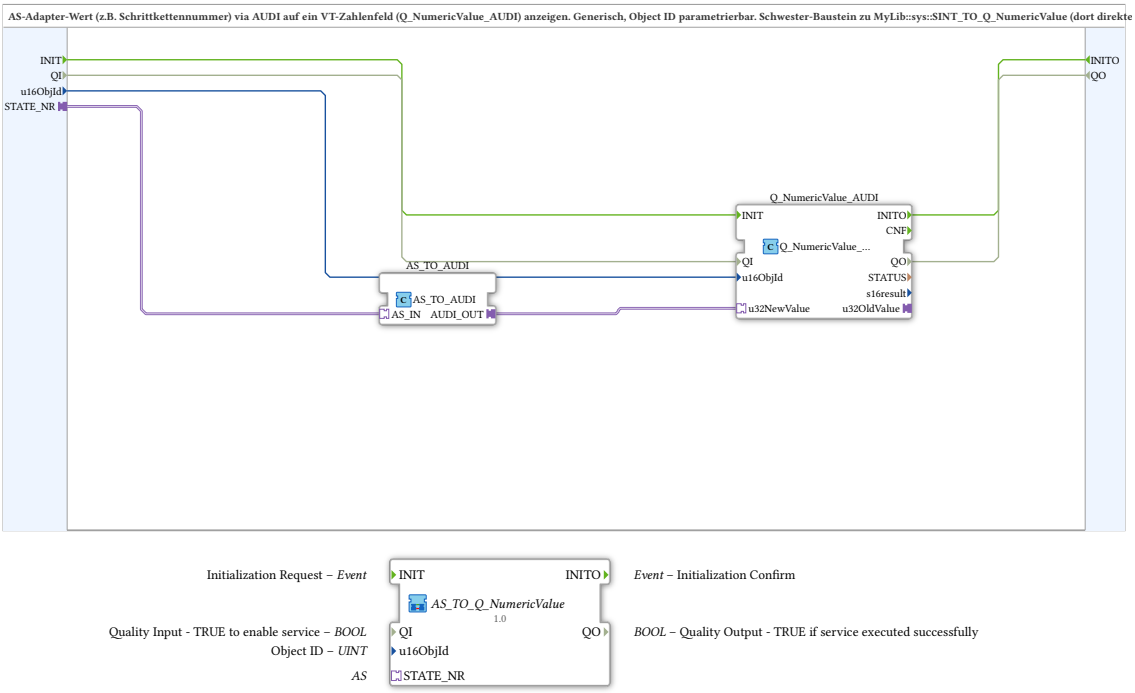
Zusammenfassung

`AS_GT_0` ist ein kompakter Wrapper um den adapter-nativen Vergleich `AS_GT` — wandelt „SINT > 0“ direkt in ein AX-Adapter-Signal um, ohne dass der Aufrufer die Null-Konstante selbst anlegen und verdrahten muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AS_TO_Q_NumericValue



• – ○ • – ○ • – ○ •

Einleitung

Der Subapp-Baustein **AS_TO_Q_NumericValue** dient zur Anzeige eines AS-Adapter-Werts (beispielsweise einer Schrittkettennummer) über einen AUDI-Konverter auf einem Visualisierungs-Zahlenfeld (VT-Zahlenfeld) vom Typ `Q_NumericValue_AUDI`. Er ist generisch aufgebaut und die Ziel-Object-ID ist über den Eingang `u16ObjId` parametrierbar. Dieser Baustein ist die Schwester-Variante zu `MyLib::sys::SINT_TO_Q_NumericValue`, bei dem anstelle des AS-Adapters eine direkte SINT-Verdrahtung verwendet wird.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung – wird an `Q_NumericValue_AUDI.INIT` durchgereicht.

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung – kommt von `Q_NumericValue_AUDI.INITO`.

Daten-Eingänge

Name	Datentyp	Initialwert	Beschreibung

QI	BOOL	TRUE	Quality Input – TRUE, um den Dienst freizugeben.
u16ObjId	UINT	ID_NULL	Object-ID für das anzuzeigende Zahlenfeld. Über diese ID wird das Zielgerät bzw. die Zielvariable in der Visualisierung adressiert.

Daten-Ausgänge

Name	Datentyp	Beschreibung
QO	BOOL	Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde (von Q_NumericValue_AUDI.QO).

Adapter

Name	Typ	Richtung	Beschreibung
STATE_NR	adapter::types::unidirectional::AS	Eingang (Socket)	Adapter für den AS-Wert (z.B. aktuelle Schrittkettennummer).

Funktionsweise

Der Subapp-Baustein nimmt über den Adapter STATE_NR einen AS-Wert entgegen. Dieser wird durch den internen FB AS_TO_AUDI (Typ adapter::conversion::unidirectional::AS_TO_AUDI) in ein AUDI-Signal umgewandelt. Das konvertierte Signal (AUDI_OUT) wird an den FB Q_NumericValue_AUDI (Typ isobus::UT::Q::Q_NumericValue_AUDI) übergeben, welcher den Wert in ein VT-Zahlenfeld schreibt. Die Ziel-Object-ID wird dabei über den Eingang u16ObjId gesetzt und direkt mit dem FB Q_NumericValue_AUDI verbunden.

Zusätzlich sind die Initialisierung und die Quality-Signale durchgereicht: INIT → Q_NumericValue_AUDI.INIT, Q_NumericValue_AUDI.INITO → INITO, QI → Q_NumericValue_AUDI.QI und Q_NumericValue_AUDI.QO → QO. Die eigentliche Wertübertragung bleibt datengetrieben über Daten- und Adapterverbindungen.

Technische Besonderheiten

- **Generische Parametrierung**: Durch den Eingang u16ObjId kann der Baustein für beliebige VT-Zahlenfelder verwendet werden, ohne dass eine feste Object-ID hartkodiert ist.
- **Adapter-Konvertierung**: Die Kombination aus AS_TO_AUDI und Q_NumericValue_AUDI ermöglicht die Anbindung von AS-Adapter-Daten an das AUDI-Protokoll der Visualisierung.
- **Wiederverwendbarkeit**: Der Baustein wurde aus der Übung Uebung_039_sub_NumbAnzeig_AS ausgelagert, um in mehreren Projekten unabhängig einsetzbar zu sein.
- **Datengetriebene Übertragung**: Außer dem Initialisierungspfad (INIT/INITO) gibt es keine Ereignisverdrahtung; die Werte werden über Daten- und Adapterverbindungen übertragen, was eine einfache und robuste Integration erlaubt.
- **Quality-Muster**: QI (Vorgabe TRUE) gibt den Dienst frei, QO meldet den Erfolg des inneren Q_NumericValue_AUDI.
- **Import**: Enthält einen Import für isobus::UT::Q::const::IDs::ID_NULL zur Initialisierung der Object-ID.

Zustandsübersicht

Da der Baustein keine Zustandsautomaten besitzt (INIT/INITO werden nur durchgereicht), gibt es keine expliziten Zustände. Die Funktionalität ist rein kombinatorisch und abhängig von den aktuellen Eingangswerten. Änderungen am Adapterwert oder an u16ObjId wirken sich unmittelbar auf die Ausgabe im Zahlenfeld aus.

Anwendungsszenarien

- **Visualisierung von Schrittkettennummern**: Anzeige der aktuellen Schrittnummer einer SPS-Steuerung über einen AS-Adapter auf einem HMI-Bildschirm.
- **Generische Werteanzeige**: Der Baustein kann verwendet werden, um beliebige numerische Werte (z.B. Messwerte, Zählerstände) über AS-Schnittstellen auf VT-Zahlenfelder zu projizieren, sofern eine AUDI-Konvertierung verfügbar ist.

- **Modulare HMI-Anbindung**: Durch die Parametrierbarkeit der Object-ID eignet sich der Baustein für den Einsatz in wiederverwendbaren Funktionsbaustein-Bibliotheken für Maschinen- oder Anlagenvisualisierungen.

Vergleich mit ähnlichen Bausteinen

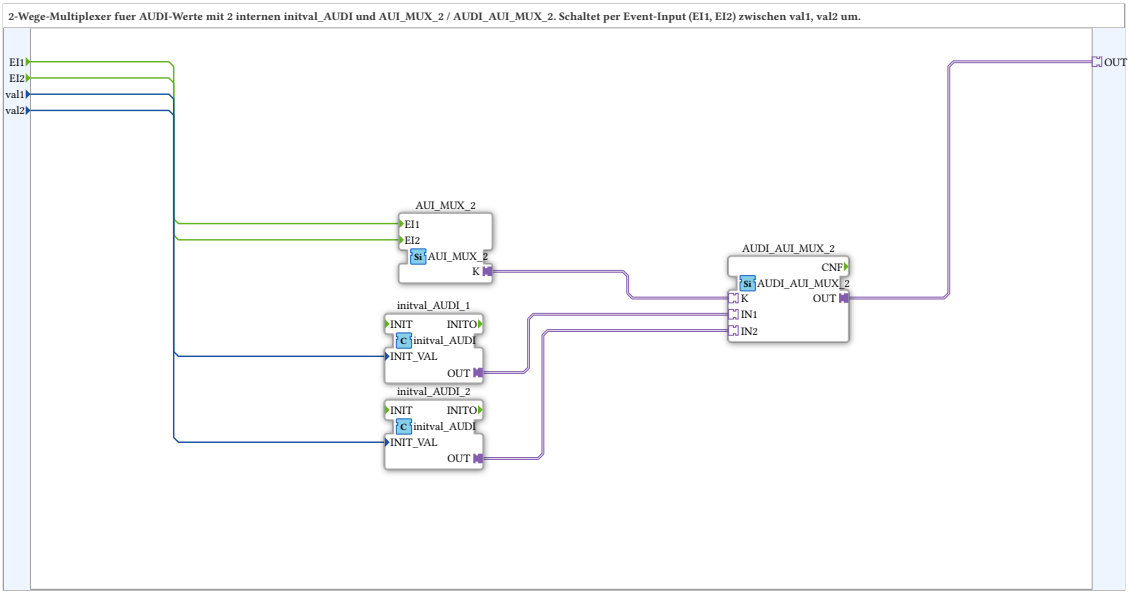
Merkmal	`AS_TO_Q_NumericValue`	`SINT_TO_Q_NumericValue` (Schwester)
Eingangsquelle	AS-Adapter	Direkter SINT-Wert
Konvertierung	`AS_TO_AUDI` → `Q_NumericValue_AUDI`	Keine (SINT wird direkt an `Q_NumericValue_AUDI` übergeben)
Parametrierung `u16ObjId`	Ja	Ja
Verwendung	AS-basierte Systeme	Direkte SINT-Verdrahtung
Komplexität	Höher (Adapterlogik)	Geringer

Beide Bausteine verfolgen das gleiche Ziel – die Anzeige eines Werts auf einem VT-Zahlenfeld – unterscheiden sich aber in der Art der Wertquelle. `AS_TO_Q_NumericValue` ist für Systeme gedacht, die über AS-Adapter kommunizieren, während die Schwester-Variante für einfache SINT-Signale optimiert ist.

Fazit

Der Subapp-Baustein `AS_TO_Q_NumericValue` bietet eine flexible und wiederverwendbare Lösung zur Anzeige von AS-Adapter-Werten auf VT-Zahlenfeldern. Durch die generische Object-ID-Ansteuerung und die klare Trennung von Adapterkonvertierung und Visualisierung eignet er sich hervorragend für modulare Automatisierungsprojekte. Die Einbettung in die Bibliothek `MyLib::sys` gewährleistet eine konsistente Integration und Reduzierung von Redundanz in zukünftigen Anwendungen.

AUDI_AUI_MUX_2_VAL



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUDI_AUI_MUX_2_VAL** ist ein 2-Wege-Multiplexer für AUDI-Werte. Er wählt in Abhängigkeit von den beiden Ereigniseingängen `EI1` und `EI2` zwischen zwei unterschiedlichen Werten (`val1`, `val2`) aus und stellt den ausgewählten Wert über einen AUDI-Adapter (`OUT`) bereit. Die Umschaltung erfolgt ereignisgesteuert, wobei die beiden passenden Eingangswerte zuvor intern über spezielle Initialisierungsbausteine in AUDI-Adapter umgewandelt werden.

Schnittstellenstruktur

Ereignis-Eingänge

Bezeichnung	Datentyp	Beschreibung
`EI1`	`Event`	Event zur Auswahl von `val1`
`EI2`	`Event`	Event zur Auswahl von `val2`

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Bezeichnung	Datentyp	Beschreibung
`val1`	`UDINT`	Initialer Ausgabewert bei `EI1`
`val2`	`UDINT`	Initialer Ausgabewert bei `EI2`

Daten-Ausgänge

Keine vorhanden.

Adapter

Bezeichnung	Typ	Beschreibung
`OUT`	`adapter::types::unidirectional::AUDI`	Ausgewählter AUDI-Adapter als Ausgang

Funktionsweise

Der Baustein besitzt ein internes Netzwerk, das aus mehreren Funktionsbausteinen besteht:

- **`AUI_MUX_2`** – Ein Ereignis-Multiplexer, der die eingehenden Events `EI1` und `EI2` verarbeitet und über einen Steuerkanal `K` den ausgewählten Kanal signalisiert.
- **`AUDI_AUI_MUX_2`** – Ein Selektionsbaustein, der anhand des Steuerkanals `K` einen der beiden Eingänge `IN1` oder `IN2` auswählt und diesen auf den Ausgang `OUT` durchschaltet.
- **`initval_AUDI_1`** und **`initval_AUDI_2`** – Zwei Initialisierungsbausteine, die die Werte `val1` bzw. `val2` als `INIT_VAL` übernehmen und daraus jeweils einen AUDI-Adapter mit diesem Initialwert erzeugen.

Die Ereignisse `EI1` und `EI2` werden direkt an den internen Baustein `AUI_MUX_2` weitergeleitet. Dieser generiert ein Steuersignal auf seinem Ausgang `K`, welches den Selektionsbaustein `AUDI_AUI_MUX_2` veranlasst, entweder den von `initval_AUDI_1` erzeugten AUDI-Wert (Kanal 1) oder den von `initval_AUDI_2` erzeugten Wert (Kanal 2) auf den externen Ausgangs-Adapter `OUT` zu legen.

Durch diese interne Verschaltung ist eine zuverlässige und ereignisgesteuerte Umschaltung zwischen zwei AUDI-Werten gewährleistet, ohne dass auf externe Multiplexer-Bausteine zurückgegriffen werden muss.

Technische Besonderheiten

- **SubApp-basiert:** Der Baustein ist als Subapplikation (SubApp) modelliert und stellt seine Funktionalität über ein internes Netzwerk bereit.
- **Verwendung spezieller Bausteintypen:**
  - `adapter::events::unidirectional::AUI_MUX_2`
  - `adapter::selection::unidirectional::AUDI_AUI_MUX_2`
  - `adapter::types::unidirectional::AUDI::initval::initval_AUDI`

- **Keine eigene Algorithmuslogik:** Die Funktion ist vollständig durch die Verbindung der internen Bausteine realisiert.
- **Initialwertaufbereitung:** Die Eingänge `val1` und `val2` werden intern zu AUDI-Adapter-Objekten umgewandelt, sodass der Ausgang stets einen vollwertigen AUDI-Adapter liefert.
- **Skalierbarkeit:** Durch die modulare Netzwerkstruktur kann der Baustein leicht auf weitere Kanäle erweitert werden (beispielsweise zur 3-fach Variante).

Zustandsübersicht

Der Baustein selbst besitzt keine expliziten Zustände, da die Umschaltung rein ereignisgesteuert erfolgt. Es lassen sich jedoch zwei logische Zustände unterscheiden:

1. **Kanal 1 aktiv:** Nach Eintreffen von `EI1` wird der Wert `val1` über den AUDI-Adapter ausgegeben.
2. **Kanal 2 aktiv:** Nach Eintreffen von `EI2` wird der Wert `val2` über den AUDI-Adapter ausgegeben.

Beide Zustände sind stabil, bis das jeweils andere Ereignis eintrifft. Es gibt keine internen Speicherzustände oder zyklische Verarbeitung; die Umschaltung erfolgt unmittelbar bei Ereigniseintritt.

Anwendungsszenarien

- **Umschaltung zwischen zwei festen Konfigurationswerten** in einer Automatisierungsumgebung.
- **Auswahl eines von zwei Initialwerten** für einen nachgeschalteten AUDI-Kommunikationskanal.
- **Ereignisgesteuertes Multiplexing** von AUDI-Adaptern in modularen Steuerungssystemen.
- **Einfache Redundanzlösung:** Wenn ein Wert fehlerhaft ist, kann auf einen zweiten Wert umgeschaltet werden.

Vergleich mit ähnlichen Bausteinen

Der Baustein ist eine **2-fach-Variante** des `AUDI_AUI_MUX_3_VAL`, der drei Eingänge unterstützt. Im Gegensatz zu einem einfachen `AUI_MUX_2` (rein ereignisbasierte Umschaltung) kombiniert `AUDI_AUI_MUX_2_VAL` die Ereignisumschaltung mit der automatischen Bereitstellung von AUDI-Initialwerten über interne `initval_AUDI`-Bausteine. Dadurch ist er besonders für Anwendungen geeignet, bei denen die AUDI-Werte direkt aus einfachen Ganzzahlen (UDINT) erzeugt werden sollen, ohne dass externe Initialisierungsbausteine benötigt werden.

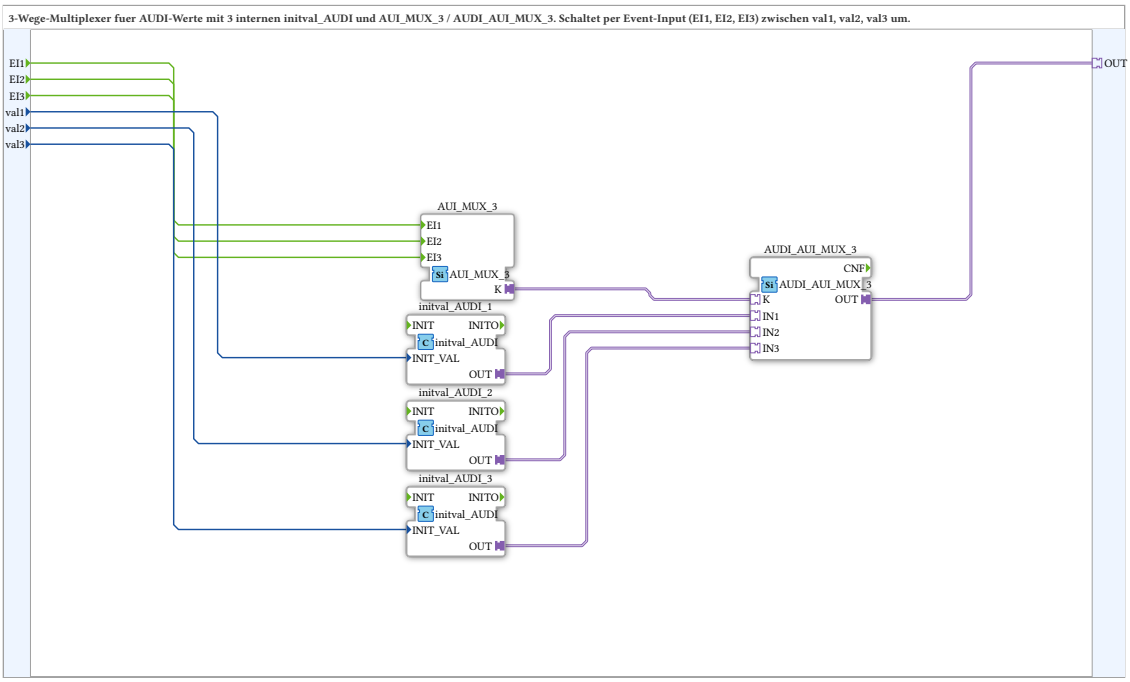
Weitere Unterschiede zu einem generischen Multiplexer:

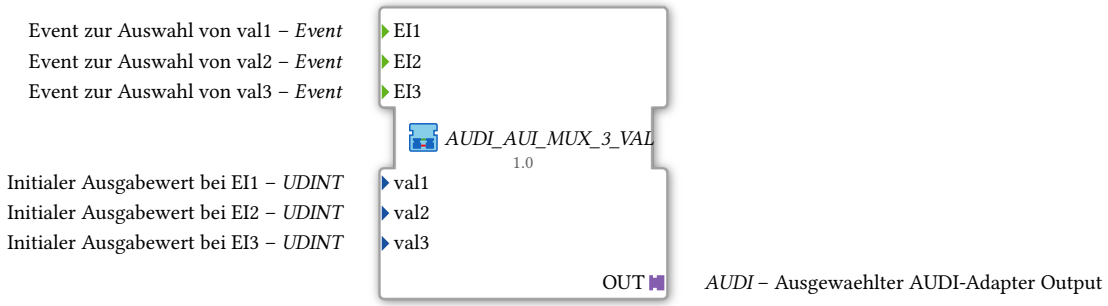
- **Keine Datenausgänge:** Die Auswahl wird ausschließlich über den AUDI-Adapter ausgegeben, nicht als physischer Datentyp.
- **Eingangsdaten sind UDINT:** Die Werte werden erst intern in AUDI-Adapter umgewandelt, was eine einfache Konfiguration mit Ganzzahlen ermöglicht.

Fazit

Der **AUDI_AUI_MUX_2_VAL** ist ein kompakter und flexibel einsetzbarer 2-Wege-Multiplexer für AUDI-Adapter. Durch die interne Aufbereitung der Eingangswerte und die ereignisgesteuerte Umschaltung eignet er sich ideal für Anwendungen, in denen zwei verschiedene AUDI-Werte ohne großen Aufwand ausgewählt werden müssen. Die modulare SubApp-Struktur erlaubt zudem eine einfache Wartung und Erweiterung, ohne die äußere Schnittstelle zu verändern.

AUDI_AUI_MUX_3_VAL





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUDI_AUI_MUX_3_VAL** ist eine Subapplikation (SubApp), die als 3-Wege-Multiplexer für AUDI-Werte dient. Über drei Ereignis-Eingänge (EI1, EI2, EI3) wird jeweils einer von drei Eingangswerten (val1, val2, val3) ausgewählt und über den Adapter-Ausgang **OUT** bereitgestellt. Die SubApp kapselt dabei die benötigte Logik aus Ereignis-Multiplexing, Wertinitialisierung und Adapter-Umschaltung in einer wiederverwendbaren Einheit.

Schnittstellenstruktur

Die SubApp besitzt drei Ereignis-Eingänge, drei Daten-Eingänge und einen Adapter-Ausgang. Ereignis-Ausgänge und Daten-Ausgänge sind nicht vorhanden.

Ereignis-Eingänge

Name	Typ	Kommentar
EI1	Event	Event zur Auswahl von val1
EI2	Event	Event zur Auswahl von val2
EI3	Event	Event zur Auswahl von val3

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Name	Typ	Kommentar
val1	UDINT	Initialer Ausgabewert bei EI1
val2	UDINT	Initialer Ausgabewert bei EI2
val3	UDINT	Initialer Ausgabewert bei EI3

Daten-Ausgänge

Keine.

Adapter

Name	Typ	Kommentar
OUT	adapter::types::unidirectional::AUDI	Ausgewählter AUDI-Adapter-Output

Funktionsweise

Die SubApp realisiert einen Multiplexer für AUDI-Adapterwerte. Intern werden die drei Eingangswerte (val1...val3) zunächst in jeweils einen **initval_AUDI**-Baustein geladen, der diese Werte als Initialwerte für einen AUDI-Adapter bereitstellt. Über den Baustein **AUI_MUX_3** (Ereignis-Multiplexer) wird das an EI1, EI2 oder EI3 anliegende Ereignis an den entsprechenden Kanal des Bausteins **AUDI_AUI_MUX_3** weitergeleitet. Dieser wählt abhängig vom empfangenen Ereignis den passenden Adapter-Eingang (IN1, IN2, IN3) aus und verbindet ihn mit dem Ausgang OUT. Dadurch erscheint am Ausgang der Wert, der mit dem auslösenden Ereignis assoziiert ist.

Technische Besonderheiten

- **Typisierung:** Die Eingangswerte sind als UDINT (32-Bit-Unsigned Integer) definiert, was eine klare Datenbreite und Bereichsdefinition ermöglicht.
- **Adapter-Architektur:** Die Verwendung unidirektionaler Adapter (`adapter::types::unidirectional::AUDI`) sorgt für eine saubere Schnittstellen-Kapselung und Wiederverwendbarkeit.
- **Drei unabhängige Initialwerte:** Durch die drei getrennten `initval_AUDI`-Instanzen können die Werte unabhängig voneinander initialisiert und später über die Ereignisse geschaltet werden.
- **Ereignis-gesteuert:** Kein zyklisches Polling, sondern ereignisbasierte Auswahl – geeignet für zeitkritische Anwendungen.

Zustandsübersicht

Die SubApp selbst besitzt keinen expliziten Zustandsautomaten. Ihr Verhalten wird durch die intern verschalteten Funktionsblöcke bestimmt. Der aktive Ausgang wird ausschließlich durch das zuletzt eingetroffene Ereignis (EI1, EI2 oder EI3) festgelegt. Es gibt keine internen Zustandsregister, die über die Auswahl hinausgehen – die Logik ist rein ereignis- und datengetrieben.

Anwendungsszenarien

- **Messwertumschaltung:** Auswahl zwischen mehreren Sensoren oder Messkanälen über Steuersignale.
- **Parameterauswahl:** Dynamisches Umschalten zwischen verschiedenen Konfigurationswerten (z. B. Sollwerten oder Grenzwerten) in einer Steuerung.
- **Redundante Systeme:** Umschalten zwischen redundanten Signalquellen, wobei die Initialwerte im Fehlerfall auf definierte Alternativen gesetzt werden.
- **Testumgebungen:** Simulierte Wertebereitstellung mit Ereignissteuerung in Automatisierungstests.

Vergleich mit ähnlichen Bausteinen

Ein einfacher Multiplexer ohne Adapter-Schnittstelle würde lediglich die Datenwerte direkt durchschalten und hätte keine Kapselung in einem AUDI-Adapter. Der **AUDI_AUI_MUX_3_VAL** bietet gegenüber einem solchen Baustein:

- Die Möglichkeit, vordefinierte Initialwerte über Adapter zu nutzen, was die Integration in bestehende Adapter-Netzwerke erleichtert.
- Eine klare Trennung zwischen Wertquelle (Eingänge) und Wertsenke (Adapter-Ausgang).

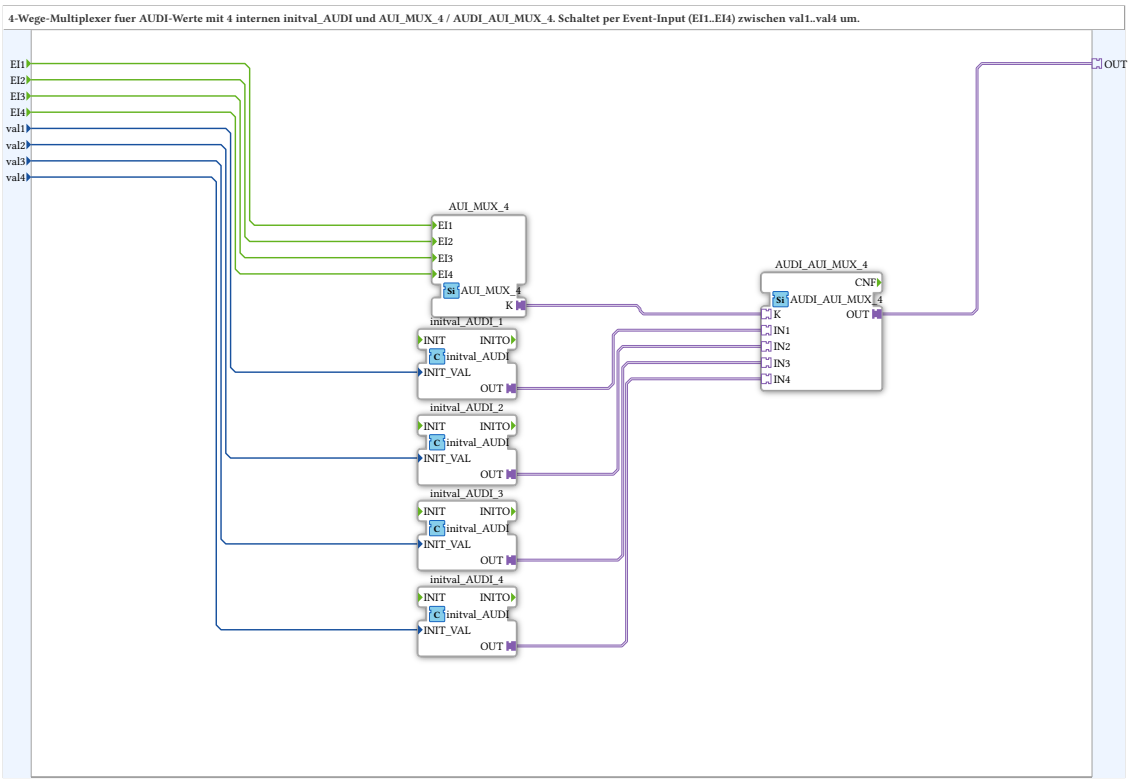
- Eine ereignisbasierte Steuerung, die parallele Datenleitungen ersetzt und so die Verbindungsanzahl reduziert.

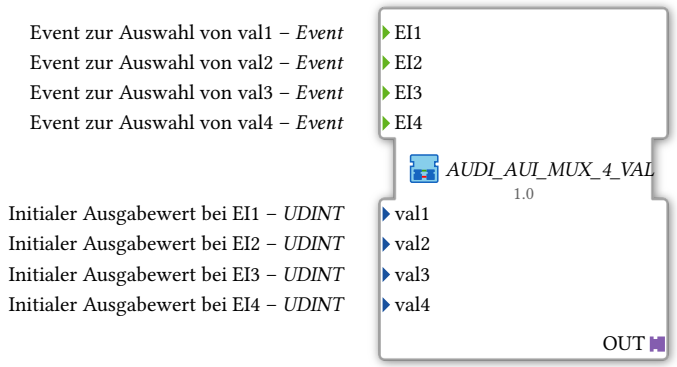
Im Vergleich zu einer Variante ohne integrierte Initialwert-Bausteine müsste der Anwender die Wertzuweisung selbst übernehmen; hier wird die Initialisierung durch `initval_AUDI` explizit vorgenommen und vereinfacht die Verwendung.

Fazit

Der **AUDI_AUI_MUX_3_VAL** ist ein kompakter, ereignisgesteuerter 3-Kanal-Multiplexer für AUDI-Adapterwerte. Durch die Kombination aus Ereignis-Multiplexing, Initialwert-Bausteinen und Adapter-Ausgang bietet er eine flexible und klar strukturierte Lösung für die Auswahl unterschiedlicher Werte in Automatisierungsanwendungen. Die saubere Kapselung macht ihn wiederverwendbar und erleichtert die Einbindung in komplexe Adapter-Netzwerke. Dank der reinen Ereignissteuerung eignet er sich besonders für Echtzeitsysteme, in denen deterministische Reaktionen erforderlich sind.

AUDI_AUI_MUX_4_VAL





AUDI – Ausgewaehlter AUDI-Adapter Output

• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AUDI_AUI_MUX_4_VAL** ist eine Subapp und realisiert einen 4-Wege-Multiplexer für AUDI-Werte. Über die Ereignis-Eingänge `EI1` bis `EI4` wird gesteuert, welcher der vier Werte `val1` bis `val4`

an den AUDI-Adapter-Ausgang `OUT` durchgeschaltet wird. Die eigentliche Auswahl übernehmen interne Funktionsbausteine vom Typ `AUI_MUX_4`, `AUDI_AUI_MUX_4` und vier `initval_AUDI`-Instanzen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
`EI1`	Event	Event zur Auswahl von `val1`
`EI2`	Event	Event zur Auswahl von `val2`
`EI3`	Event	Event zur Auswahl von `val3`
`EI4`	Event	Event zur Auswahl von `val4`

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Name	Typ	Kommentar
`val1`	UDINT	Initialer Ausgabewert bei `EI1`
`val2`	UDINT	Initialer Ausgabewert bei `EI2`
`val3`	UDINT	Initialer Ausgabewert bei `EI3`
`val4`	UDINT	Initialer Ausgabewert bei `EI4`

Daten-Ausgänge

Keine.

Adapter

Name	Typ	Richtung	Kommentar
`OUT`	`adapter::types::unidirectional::AUDI`	Plug / Ausgang	Ausgewählter AUDI-Adapter-Output

Funktionsweise

Die Subapp arbeitet als ereignisgesteuerter Multiplexer:

1. Die Daten-Eingänge `val1` bis `val4` werden jeweils an eine interne `initval_AUDI`-Instanz übergeben.
2. Diese vier Instanzen wandeln die UDINT-Werte in unidirektionale AUDI-Adapterwerte um.
3. Ein Ereignis an `EI1`, `EI2`, `EI3` oder `EI4` wird vom internen Baustein `AUI_MUX_4` verarbeitet.
4. `AUI_MUX_4` erzeugt daraus ein internes Selektionssignal `K`.
5. Der Baustein `AUDI_AUI_MUX_4` wählt anhand von `K` einen der vier AUDI-Eingänge `IN1` bis `IN4` aus.
6. Der ausgewählte AUDI-Wert wird über den Adapter-Ausgang `OUT` bereitgestellt.

Die Auswahl bleibt nach dem Auslösen eines Ereignisses aktiv, bis ein anderes Ereignis einen anderen Kanal anwählt.

Technische Besonderheiten

- Es handelt sich um eine reine Subapp-Verdrahtung ohne eigenen ECC; das Verhalten ergibt sich aus den intern verbauten Funktionsbausteinen.
- Die vier Eingangswerte sind als `UDINT` deklariert, der Ausgang transportiert jedoch einen AUDI-Adapterwert.
- Die Umwandlung von UDINT nach AUDI übernimmt jeweils ein eigener `initval_AUDI`-Baustein.
- Die Auswahl wird über eine Kombination aus Ereignis-Multiplexer (`AUI_MUX_4`) und Adapter-Selektor (`AUDI_AUI_MUX_4`) realisiert.

- Die Subapp besitzt weder Ereignis-Ausgänge noch Daten-Ausgänge; die Ergebnisübergabe erfolgt ausschließlich über den Adapter `OUT`.

Zustandsübersicht

Zustand	Auslöser	Wirkung
Kanal 1 aktiv	`EI1`	Der über `val1` initialisierte AUDI-Wert wird an `OUT` ausgegeben.
Kanal 2 aktiv	`EI2`	Der über `val2` initialisierte AUDI-Wert wird an `OUT` ausgegeben.
Kanal 3 aktiv	`EI3`	Der über `val3` initialisierte AUDI-Wert wird an `OUT` ausgegeben.
Kanal 4 aktiv	`EI4`	Der über `val4` initialisierte AUDI-Wert wird an `OUT` ausgegeben.

Nach einem Ereignis bleibt der zuletzt gewählte Kanal aktiv, bis ein anderes Ereignis eintrifft.

Anwendungsszenarien

- Umschalten zwischen vier konfigurierbaren AUDI-Werten durch einzelne Ereignisse.
- Auswahl unterschiedlicher Sollwerte oder Parameterwerte in einer Steuerungsanwendung.
- Einsatz in Test- und Simulationsumgebungen, in denen zwischen vier AUDI-basierten Datenquellen umgeschaltet werden muss.
- Wiederverwendbare Subapp in IEC-61499-Systemen, die eine strukturierte Adapter-Kommunikation verwenden.

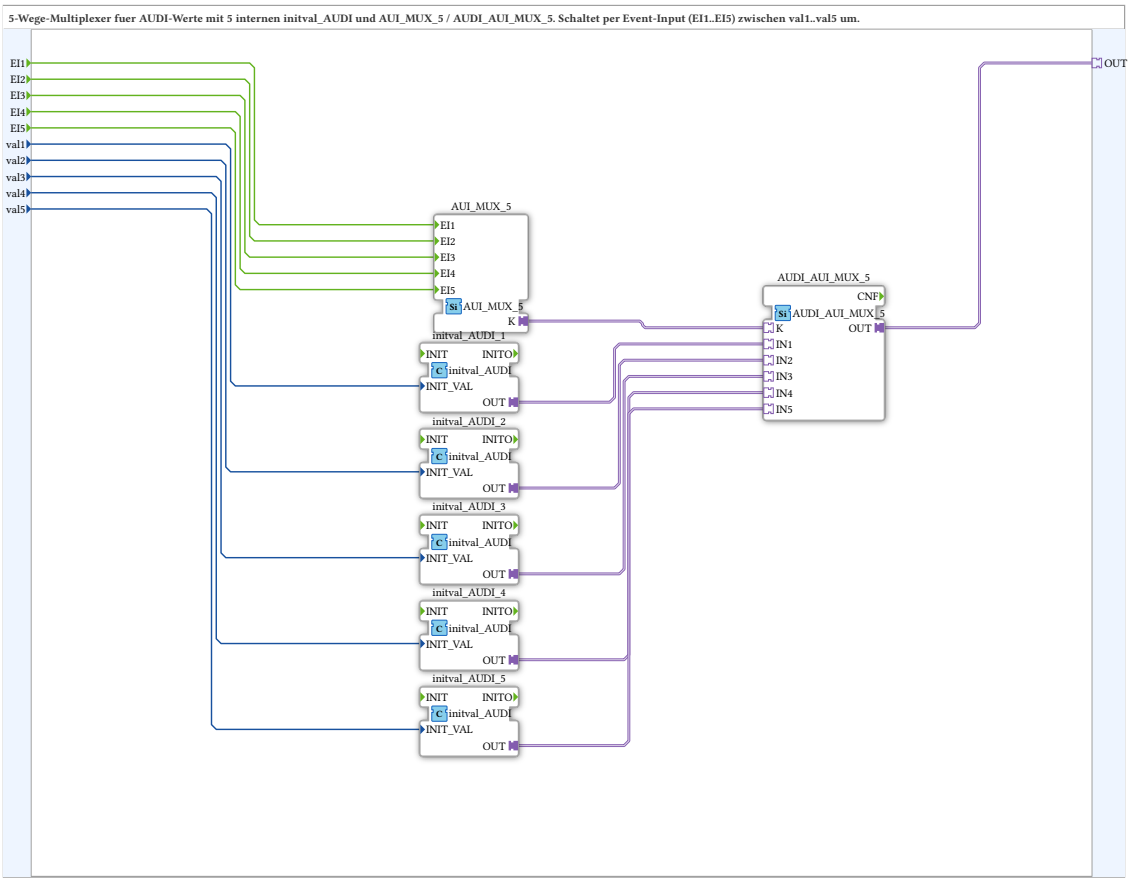
Vergleich mit ähnlichen Bausteinen

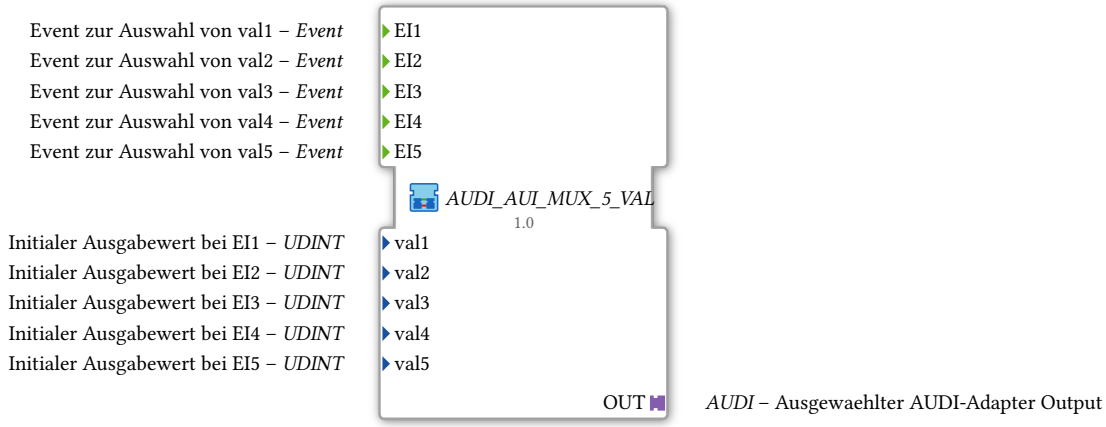
Der Baustein ist eine 4-fach-Variante des Prinzips eines AUDI-AUI-Multiplexers. Während eine 3-fach-Variante nur drei Eingänge und Ereignisse besitzt, erweitert `AUDI_AUI_MUX_4_VAL` dieses Konzept auf vier Kanäle. Im Gegensatz zu einem reinen Daten-Multiplexer werden die Werte nicht über einzelne Datenausgänge, sondern über einen unidirektionalen AUDI-Adapter ausgegeben. Dadurch ist der Baustein besonders für adapterbasierte Kommunikationsstrukturen geeignet.

Fazit

`AUDI_AUI_MUX_4_VAL` ist ein kompakter und klar strukturierter 4-Wege-Multiplexer für AUDI-Werte. Er kombiniert UDINT-Eingänge, `initval_AUDI`-Konvertierung und die interne `AUDI_AUI_MUX_4`-Selektion zu einer einfachen, ereignisgesteuerten Schnittstelle. Damit eignet er sich gut für Anwendungen, die zwischen mehreren AUDI-Werten umschalten müssen.

AUDI_AUI_MUX_5_VAL





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUDI_AUI_MUX_5_VAL** ist ein 5-Wege-Multiplexer für AUDI-Werte. Er wählt über fünf Ereignis-Eingänge (EI1...EI5) einen von fünf Datenwerten (val1...val5) aus und stellt diesen am Adapter-Ausgang OUT bereit. Die SubApp kapselt die notwendige Konvertierung der numerischen Eingänge in den AUDI-Datentyp sowie die eigentliche Multiplex-Logik.

Schnittstellenstruktur

Ereignis-Eingänge

Bezeichnung	Typ	Kommentar

EI1	Event	Event zur Auswahl von val1
EI2	Event	Event zur Auswahl von val2
EI3	Event	Event zur Auswahl von val3
EI4	Event	Event zur Auswahl von val4
EI5	Event	Event zur Auswahl von val5

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Bezeichnung	Typ	Kommentar
val1	UDINT	Initialer Ausgabewert bei EI1
val2	UDINT	Initialer Ausgabewert bei EI2
val3	UDINT	Initialer Ausgabewert bei EI3
val4	UDINT	Initialer Ausgabewert bei EI4
val5	UDINT	Initialer Ausgabewert bei EI5

Daten-Ausgänge

Keine.

Adapter

Bezeichnung	Typ	Kommentar
OUT	adapter::types::unidirectional::AUDI	Ausgewählter AUDI-Adapter-Output

Funktionsweise

Die SubApp verbindet die Eingangsereignisse direkt mit dem internen Baustein **AUI_MUX_5**, der die Ereignisse über seinen Ausgang **K** dem Auswahlbaustein **AUDI_AUI_MUX_5** mitteilt. Die numerischen Datenwerte val1…val5 werden jeweils über einen eigenen **initval_AUDI**-Baustein in den AUDI-Datentyp umgewandelt. Diese fünf konvertierten Werte werden als IN1…IN5 an **AUDI_AUI_MUX_5** geführt. Der Auswahlbaustein wählt anhand des über **K** empfangenen Ereignisses den passenden Eingang aus und stellt das Ergebnis über seinen Ausgang **OUT** dem SubApp-Adapter zur Verfügung. Dadurch wird bei einem Ereignis auf EI1 der Wert val1 ausgegeben, bei EI2 der Wert val2 usw.

Technische Besonderheiten

- Die SubApp verwendet fünf Instanzen des FB **initval_AUDI**, um die UDINT-Eingänge in den komplexen AUDI-Typ zu überführen.
- Die eigentliche Multiplex-Logik ist in zwei getrennten Bausteinen realisiert: **AUI_MUX_5** für die Ereignisverarbeitung und **AUDI_AUI_MUX_5** für die Datenauswahl.
- Durch die Kapselung wird die Verwendung von vier verschiedenen FB-Typen im Anwenderprogramm auf eine einzige SubApp reduziert.
- Die SubApp ist als 5-fach Variante des vorhandenen **AUDI_AUI_MUX_3_VAL** ausgelegt und erweitert dessen Funktionsumfang entsprechend.

Zustandsübersicht

Die SubApp besitzt keine eigenen Zustände. Sie besteht ausschließlich aus einer kaskadierten Verdrahtung von Funktionsbausteinen, die ereignisgesteuert arbeiten – eine explizite Zustandsverwaltung ist daher nicht erforderlich.

Anwendungsszenarien

- Auswahl eines von fünf Sollwerten oder Konfigurationswerten in Automatisierungssystemen, die auf dem AUDI-Datentyp basieren.
- Umschaltung zwischen verschiedenen Betriebsmodi oder Profilen zur Laufzeit.
- Einsatz in Anlagen, in denen mehrere alternative Parameter eines Prozesses über ein einheitliches Protokoll bereitgestellt werden.

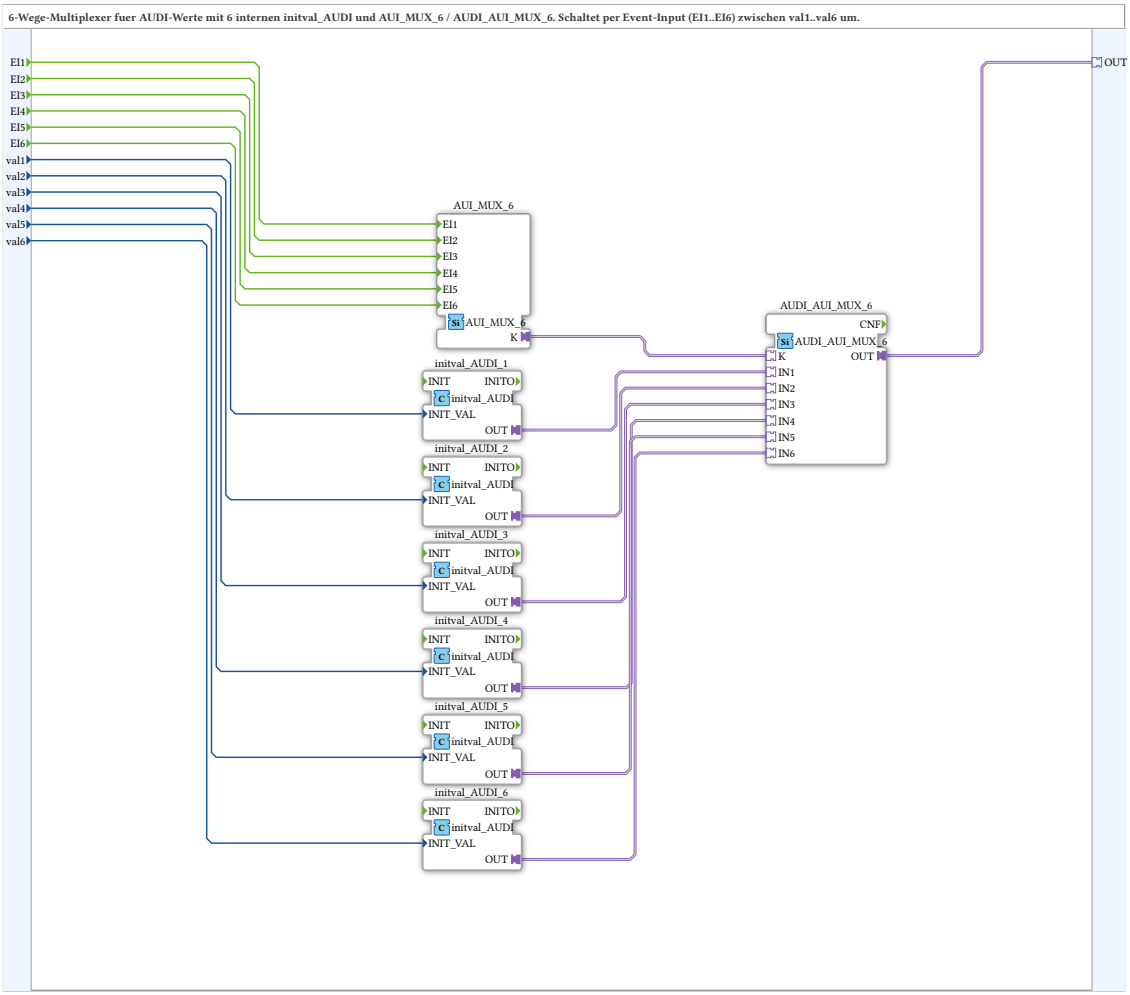
Vergleich mit ähnlichen Bausteinen

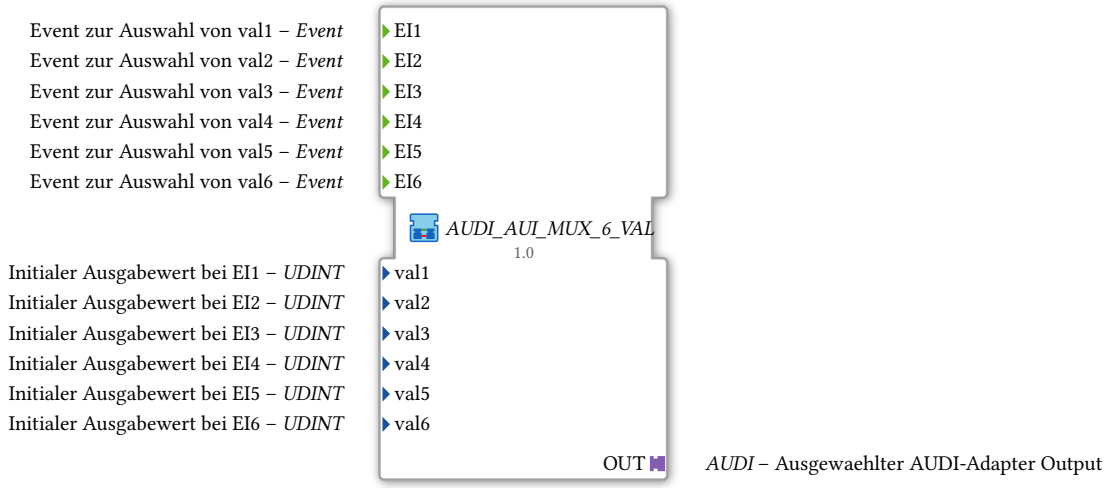
Die SubApp **AUDI_AUI_MUX_5_VAL** ist die erweiterte Version des Bausteins **AUDI_AUI_MUX_3_VAL** (3-Wege-Multiplexer). Sie bietet fünf statt drei Eingangskanäle und dieselbe interne Struktur, bestehend aus initval_AUDI, AUI_MUX und AUDI_AUI_MUX. Gegenüber einem direkten Einsatz der Einzelbausteine vereinfacht sie die Einbindung in Entwicklungsumgebungen, da nur noch die SubApp mit ihren Ereignissen und Datenwerten konfiguriert werden muss.

Fazit

AUDI_AUI_MUX_5_VAL ist ein komfortabler und kompakter Multiplexer für bis zu fünf AUDI-Werte. Er kombiniert Ereignissteuerung, Datenkonvertierung und Auswahllogik in einer wiederverwendbaren SubApp und ermöglicht eine saubere, wenig fehleranfällige Programmierung von Auswahlfunktionen. Die klare Schnittstelle und die einfache Integration machen den Baustein zu einem nützlichen Werkzeug in IEC 61499-basierten Steuerungsanwendungen.

AUDI_AUI_MUX_6_VAL





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUDI_AUI_MUX_6_VAL** ist ein 6-Wege-Multiplexer für AUDI-Adapterwerte. Er wählt über Ereignis-Eingänge einen von sechs UDINT-Datenwerten aus und stellt diesen als AUDI-Adapter am Ausgang bereit. Intern kombiniert er Ereignis- und Adapter-Multiplexer sowie Initialisierungseinheiten, um die eingehenden Werte zunächst in AUDI-Adapter zu konvertieren und anschließend den aktiven Kanal durchzuschalten. Die Auswahl erfolgt ausschließlich über die Ereignis-Eingänge **EI1** bis **EI6**, wodurch eine zeitlich präzise Umschaltung ermöglicht wird.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
EI1	Event	Event zur Auswahl von **val1**
EI2	Event	Event zur Auswahl von **val2**
EI3	Event	Event zur Auswahl von **val3**
EI4	Event	Event zur Auswahl von **val4**
EI5	Event	Event zur Auswahl von **val5**
EI6	Event	Event zur Auswahl von **val6**

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Name	Typ	Kommentar
val1	UDINT	Initialer Ausgabewert bei EI1
val2	UDINT	Initialer Ausgabewert bei EI2
val3	UDINT	Initialer Ausgabewert bei EI3
val4	UDINT	Initialer Ausgabewert bei EI4
val5	UDINT	Initialer Ausgabewert bei EI5
val6	UDINT	Initialer Ausgabewert bei EI6

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden; die Ausgabe erfolgt ausschließlich über den Adapter.

Adapter

Name	Typ	Kommentar
OUT	`adapter::types::unidirectional::AUDI`	Ausgewählter AUDI-Adapter-Output

Funktionsweise

Der Funktionsblock ist intern als Subapplikation realisiert und besteht aus mehreren Komponenten:

1. **Werteinitialisierung**: Die sechs externen UDINT-Werte **val1** ... **val6** werden jeweils einer Instanz des Bausteins `initval_AUDI` zugeführt. Diese Instanzen wandeln den übergebenen Zahlenwert in einen AUDI-Adapter um und stellen ihn an ihrem `OUT`-Port bereit.
2. **Ereignis-Multiplexer**: Der Baustein `AUI_MUX_6` empfängt die externen Ereignisse **EI1** ... **EI6**. Anhand dieser Ereignisse erzeugt er ein Auswahlsignal (am Port `K`), das den gewünschten Kanal für den nachgeschalteten Adapter-Multiplexer festlegt.
3. **Adapter-Multiplexer**: Der Baustein `AUDI_AUI_MUX_6` verfügt über sechs Eingänge (`IN1` ... `IN6`), an denen die initialisierten AUDI-Adapter anliegen, sowie über einen Steuereingang `K`. Der Steuerwert bestimmt, welcher der sechs Eingänge auf den Ausgang `OUT` durchgeschaltet wird.
4. **Ausgabe**: Der ausgewählte AUDI-Adapter wird über den externen Adapter-Port `OUT` ausgegeben.

Durch die Kombination dieser Bausteine realisiert `AUDI_AUI_MUX_6_VAL` eine zuverlässige und ereignisgesteuerte Umschaltung zwischen sechs vordefinierten Werten, ohne dass zusätzliche Logik von außen implementiert werden muss.

Technische Besonderheiten

- **Struktur als Subapplikation**: Der FB ist als SubApp definiert und nutzt dabei bereits vorhandene Typen (`AUI_MUX_6`, `AUDI_AUI_MUX_6` und `initval_AUDI`). Dies erlaubt eine kompakte Wiederverwendung erprobter Bausteine und eine klare Modularisierung.
- **Datenkonvertierung**: Eingangsdaten sind vom Typ `UDINT`. Durch die `initval_AUDI`-Instanzen werden diese Werte in den spezifischen Adaptertyp `AUDI` transformiert, der für die Ausgabe benötigt wird.
- **Exklusive Ereignissteuerung**: Die Auswahl des aktiven Kanals erfolgt ausschließlich über Ereignis-Eingänge. Es gibt keinen Daten-Eingang für die Auswahl, was die Kopplung zwischen Ereignis und Wertwechsel sehr direkt gestaltet.
- **Keine Rückmeldung über Ereignis-Ausgänge**: Nach erfolgreicher Umschaltung wird kein spezielles Ereignis erzeugt; der Zustand ist implizit über das zuletzt anstehende Ereignis definiert.

Zustandsübersicht

Da der Baustein ereignisgesteuert arbeitet, gibt es keine persistenten internen Zustände im eigentlichen Sinne. Nach jedem Ereignis wird der entsprechende Kanal gesetzt und bleibt aktiv, bis ein weiteres Ereignis eintrifft:

Zustand	Auslöser	Ergebnis (Ausgang `OUT`)
Kanal 1	EI1	Adapter aus `val1`
Kanal 2	EI2	Adapter aus `val2`
Kanal 3	EI3	Adapter aus `val3`
Kanal 4	EI4	Adapter aus `val4`
Kanal 5	EI5	Adapter aus `val5`
Kanal 6	EI6	Adapter aus `val6`

Überlappende Ereignisse sind nicht definiert; es wird allgemein erwartet, dass jeweils nur ein Ereignis aktiv ist.

Anwendungsszenarien

- **Konfigurationsumschaltung** in Automatisierungssystemen, bei denen je nach Betriebsmodus unterschiedliche Parameterwerte (z. B. Geschwindigkeitsgrenzen, Temperatursollwerte) als AUDI-Daten an eine übergeordnete Steuerung übergeben werden.
- **Quellenselektion** in Multimedia- oder Datenübertragungssystemen, bei denen zwischen mehreren Signalquellen umgeschaltet werden muss, wobei die Werte als UDINT-Konstanten vorliegen.
- **Test- und Simulationsumgebungen**: Auswahl unterschiedlicher Testmuster oder Simulationsdaten über Ereignisse, ohne die Datenwerte zur Laufzeit ändern zu müssen.
- **Redundante Wertbereitstellung**: Bereitstellung von sechs verschiedenen vordefinierten Werten, die je nach Prozessschritt über Events angefordert werden.

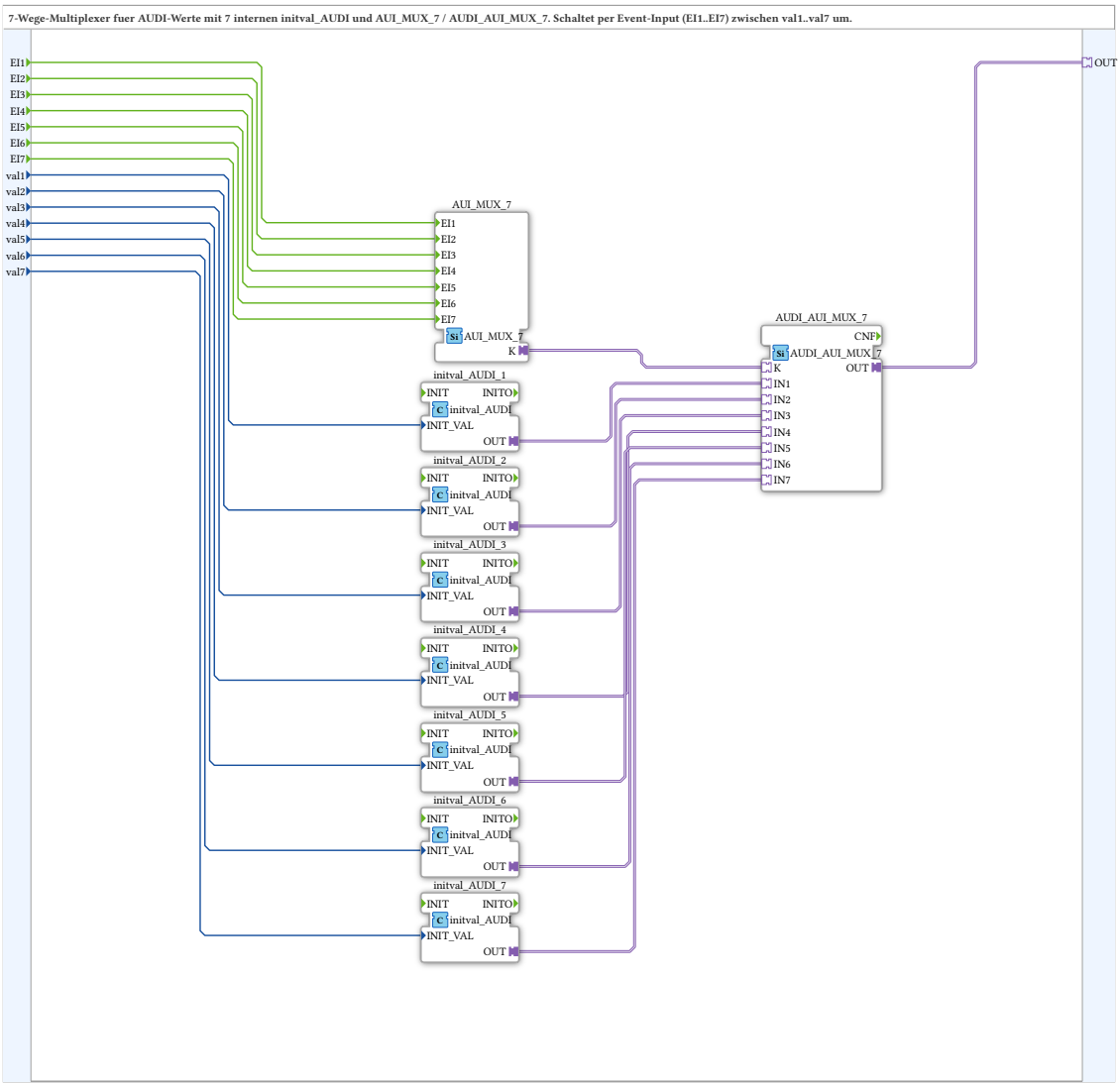
Vergleich mit ähnlichen Bausteinen

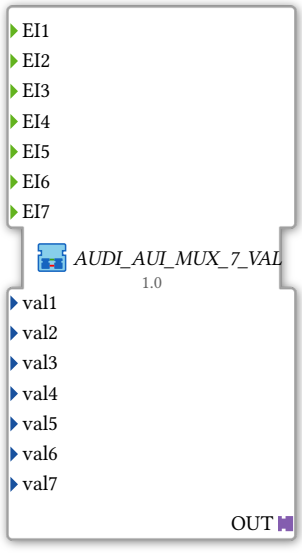
- **AUDI_AUI_MUX_3_VAL**: Eine kleinere Variante mit nur drei Eingängen, die analog funktioniert. `AUDI_AUI_MUX_6_VAL` erweitert diese um weitere drei Kanäle und ist daher für Anwendungen mit mehr Auswahlmöglichkeiten geeignet.
- **Generische Multiplexer-Bausteine**: Ohne spezifische Datenumwandlung, meist mit numerischem Auswahl-Eingang (z. B. INT), während dieser Baustein die Auswahl ausschließlich über Ereignisse vornimmt und die Werte in einen speziellen Adaptertyp umwandelt.
- **Bausteine mit Daten-Ausgängen**: Im Gegensatz zu Bausteinen, die einen skalaren Datenwert ausgeben, liefert dieser FB einen Adapter, was ihn für die Kommunikation mit anderen Adapter-basierten Komponenten prädestiniert.

Fazit

Der Funktionsblock `AUDI_AUI_MUX_6_VAL` bietet eine flexible und kompakte Lösung zur ereignisgesteuerten Auswahl aus sechs UDINT-Werten, die als AUDI-Adapter nach außen geführt werden. Durch den modularen Aufbau als Subapplikation und die Wiederverwendung erprobter Komponenten ist er zuverlässig und gut in bestehende 4diac-Projekte integrierbar. Die Kombination aus Ereignis-Multiplexer, Datenkonvertierung und Adapter-Multiplexer erfüllt die Anforderungen vieler Automatisierungs- und Steuerungsszenarien, ohne dass zusätzliche Logik erforderlich ist.

AUDI_AUI_MUX_7_VAL





Event zur Auswahl von val1 – *Event*

Event zur Auswahl von val2 – *Event*

Event zur Auswahl von val3 – *Event*

Event zur Auswahl von val4 – *Event*

Event zur Auswahl von val5 – *Event*

Event zur Auswahl von val6 – *Event*

Event zur Auswahl von val7 – *Event*

Initialer Ausgabewert bei EI1 – *UDINT*

Initialer Ausgabewert bei EI2 – *UDINT*

Initialer Ausgabewert bei EI3 – *UDINT*

Initialer Ausgabewert bei EI4 – *UDINT*

Initialer Ausgabewert bei EI5 – *UDINT*

Initialer Ausgabewert bei EI6 – *UDINT*

Initialer Ausgabewert bei EI7 – *UDINT*

AUDI – Ausgewaehlter AUDI-Adapter Output

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **AUDI_AUI_MUX_7_VAL** ist ein 7-wege-Multiplexer für AUDI-Werte. Er wählt anhand eines Ereigniseingangs (EI1..EI7) einen von sieben Eingangswerten (val1..val7) aus und stellt diesen über den AUDI-Adapterausgang (OUT) bereit. Intern kombiniert der Baustein einen AUI_MUX_7-Index-Selektor, einen AUDI_AUI_MUX_7-Auswahlbaustein und sieben initval_AUDI-Initialisierungsbausteine, um die Rohwerte in den AUDI-Datentyp zu überführen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
EI1	Event	Event zur Auswahl von val1
EI2	Event	Event zur Auswahl von val2
EI3	Event	Event zur Auswahl von val3
EI4	Event	Event zur Auswahl von val4
EI5	Event	Event zur Auswahl von val5
EI6	Event	Event zur Auswahl von val6
EI7	Event	Event zur Auswahl von val7

Ereignis-Ausgänge

Keine definiert.

Daten-Eingänge

Name	Typ	Kommentar
val1	UDINT	Initialer Ausgabewert bei EI1
val2	UDINT	Initialer Ausgabewert bei EI2
val3	UDINT	Initialer Ausgabewert bei EI3
val4	UDINT	Initialer Ausgabewert bei EI4
val5	UDINT	Initialer Ausgabewert bei EI5
val6	UDINT	Initialer Ausgabewert bei EI6
val7	UDINT	Initialer Ausgabewert bei EI7

Daten-Ausgänge

Keine definiert.

Adapter

Name	Typ	Kommentar
OUT	adapter::types::unidirectional::AUDI	Ausgewählter AUDI-Adapter Output

Funktionsweise

Der Baustein arbeitet ereignisgesteuert. Wird ein Ereignis an einem der Eingänge EI1 bis EI7 empfangen, so wird der zugehörige Datenwert (val1..val7) als Initialisierungswert an den entsprechenden internen initval_AUDI-Baustein übergeben. Diese Bausteine konvertieren den rohen `UDINT`-Wert in einen AUDI-Adapterdatensatz.

Parallel dazu erzeugt der Ereigniseingang über den internen AUI_MUX_7-Baustein ein Indexsignal (K), das den gewünschten Kanal (1..7) selektiert. Der anschließende AUDI_AUI_MUX_7-Baustein greift

auf die sieben von den initval_AUDI-Bausteinen erzeugten AUDI-Werte zu und gibt den über den Index ausgewählten Wert über den Adapterausgang OUT zurück.

Die Umschaltung ist synchron: Ein einzelnes Ereignis reicht aus, um den Datenwert am Ausgang zu ändern, wobei der gewählte Wert unmittelbar nach Verarbeitung des Ereignisses zur Verfügung steht.

Technische Besonderheiten

- **SubApp-Struktur**: Der Baustein ist als SubApp (SubAppType) realisiert und besteht aus mehreren internen Bausteinen (`AUI_MUX_7`, `AUDI_AUI_MUX_7`, `initval_AUDI_1`..`initval_AUDI_7`).
- **Adapterbasierte Kommunikation**: Die Verbindung zum Ausgang erfolgt über einen unidirektionalen AUDI-Adapter, der eine standardisierte Schnittstelle für AUDI-Daten bietet.
- **Typumwandlung**: Die rohen UDINT-Werte werden über die `initval_AUDI`-Bausteine in den AUDI-Datentyp konvertiert, sodass der Baustein mit unterschiedlichen Datenformaten umgehen kann.
- **Keine Ereignisausgänge**: Die SubApp erzeugt selbst keine Ereignisse; die gesamte Steuerung läuft über die Ereigniseingänge und den Datenfluss.

Zustandsübersicht

Der Baustein besitzt keinen internen Zustandsspeicher. Es handelt sich um eine rein kombinatorische Auswahlschaltung, deren Ergebnis ausschließlich von den aktuellen Eingangswerten und dem zuletzt empfangenen Ereignis abhängt. Eine separate Zustandsmaschine ist nicht erforderlich.

Anwendungsszenarien

- **Konfigurationsumschaltung**: Auswahl eines von sieben vorgegebenen AUDI-Werten (z.B. Parameter, Profile, Betriebsmodi) durch gezielte Ereignissignale.
- **Redundante Wertbereitstellung**: Bereitstellung mehrerer möglicher Ausgabewerte, die je nach Ereignis aktiviert werden.
- **Test- und Simulationsumgebungen**: Einsatz bei Teststänaden, um unterschiedliche AUDI-Datensätze schnell durchschalten zu können.
- **Multiplexing in Automatisierungssystemen**: Wenn mehrere Datenquellen vorliegen und nur eine davon gleichzeitig ausgegeben werden soll.

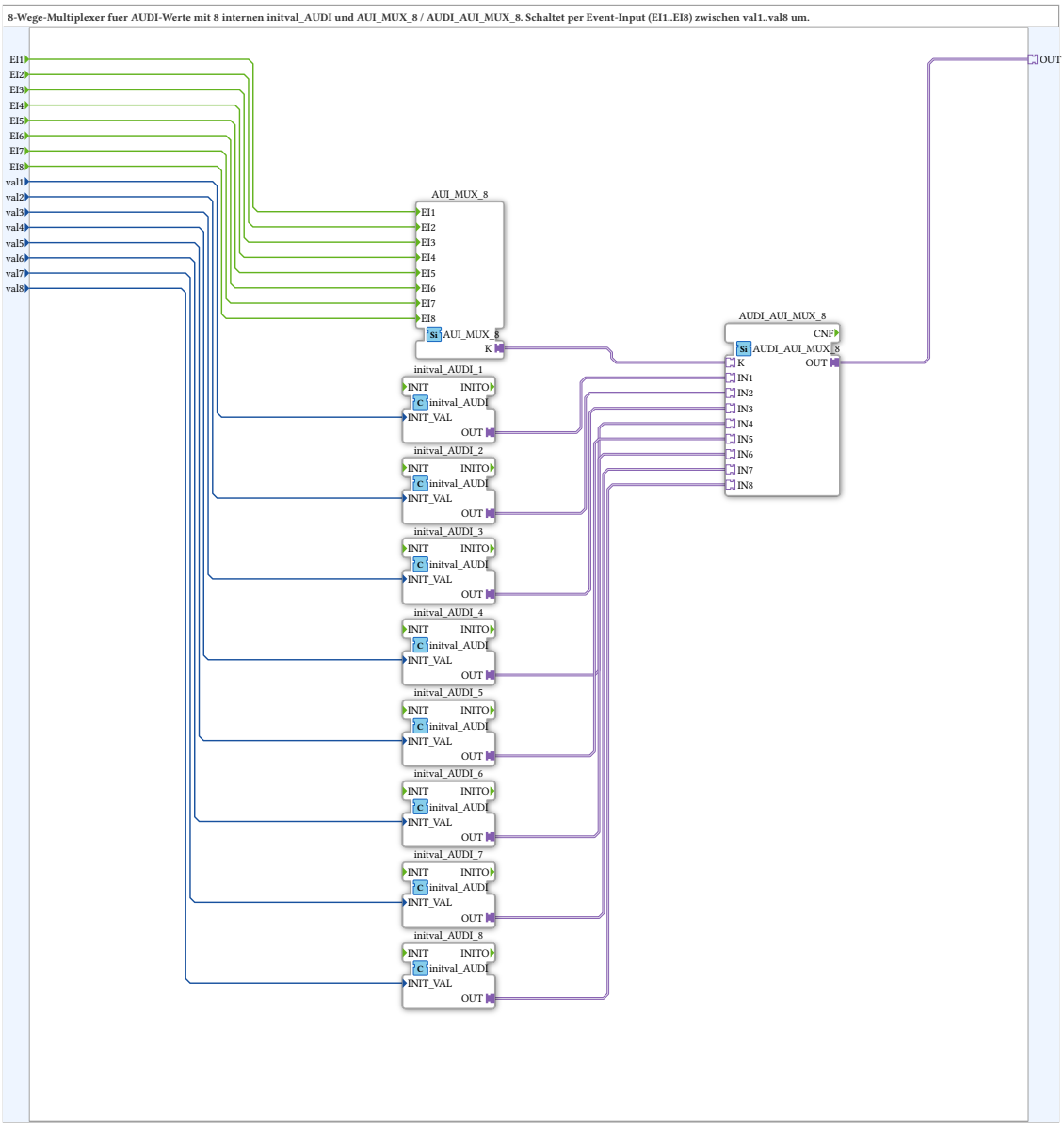
Vergleich mit ähnlichen Bausteinen

Es existieren bereits Varianten wie `AUDI_AUI_MUX_3_VAL` (3-Wege-Multiplexer). Der vorliegende Baustein ist die 7-fach Variante und erweitert die Auswahlmöglichkeiten entsprechend. Gemeinsam ist ihnen die prinzipielle Arbeitsweise: Ereignisgesteuerte Auswahl über AUI_MUX-Komponenten und initval_AUDI-Initialisierung. Der 7-Wege-Multiplexer bietet eine höhere Anzahl an Eingängen und ist für umfangreichere Auswahlszenarien geeignet, benötigt jedoch auch mehr interne Ressourcen.

Fazit

Der Funktionsbaustein **AUDI_AUI_MUX_7_VAL** stellt eine robuste und flexible Lösung zur Auswahl eines von sieben AUDI-Werten dar. Durch die Einbettung als SubApp mit standardisierten Adaptern ist er leicht in übergeordnete Systeme integrierbar. Die klare ereignisbasierte Schnittstelle ermöglicht eine unkomplizierte Ansteuerung und erfüllt die Anforderungen an einen vielseitigen Multiplexer in Automatisierungs- und Steuerungsumgebungen.

AUDI_AUI_MUX_8_VAL



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AUDI_AUI_MUX_8_VAL** ist ein 8-Wege-Multiplexer für Werte vom Typ **AUDI** (Adapter). Über acht separate Ereignis-Eingänge kann jeweils einer von acht Datenwerten (val1 … val8) ausgewählt und als einziger Adapter-Ausgang **OUT** bereitgestellt werden. Der Baustein kombiniert einen Ereignis-Multiplexer mit einer Adapter-Selektionslogik und eignet sich besonders für die dynamische Umschaltung zwischen unterschiedlichen Konfigurations- oder Steuerwerten in IEC 61499-Applikationen.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Datentyp	Kommentar
EI1	Event	Event zur Auswahl von val1
EI2	Event	Event zur Auswahl von val2
EI3	Event	Event zur Auswahl von val3
EI4	Event	Event zur Auswahl von val4
EI5	Event	Event zur Auswahl von val5
EI6	Event	Event zur Auswahl von val6
EI7	Event	Event zur Auswahl von val7
EI8	Event	Event zur Auswahl von val8

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Name	Datentyp	Kommentar
val1	UDINT	Initialer Ausgabewert bei EI1
val2	UDINT	Initialer Ausgabewert bei EI2

val3	UDINT	Initialer Ausgabewert bei EI3
val4	UDINT	Initialer Ausgabewert bei EI4
val5	UDINT	Initialer Ausgabewert bei EI5
val6	UDINT	Initialer Ausgabewert bei EI6
val7	UDINT	Initialer Ausgabewert bei EI7
val8	UDINT	Initialer Ausgabewert bei EI8

Daten-Ausgänge

Keine direkten Datenausgänge vorhanden.

Adapter

Name	Typ	Kommentar
OUT	adapter::types::unidirectional::AUDI	Ausgewählter AUDI-Adapter-Ausgang

Funktionsweise

Der Baustein ist intern in drei Funktionsblöcke gegliedert:

- **AUI_MUX_8** (Ereignis-Multiplexer): nimmt die Ereignisse EI1…EI8 entgegen und erzeugt ein entsprechendes Ereignis an seinem Ausgang.
- **AUDI_AUI_MUX_8** (Adapter-Multiplexer): wählt anhand des zugehörigen Ereignisses einen der acht internen AUDI-Adapter-Eingänge aus und verbindet ihn mit dem Ausgangs-Adapter.
- **8× initval_AUDI** (Initialisierungswerte): wandelt den jeweiligen UDINT-Eingangswert in einen AUDI-Adapter um. Jede Instanz ist fest mit einem der Daten-Eingänge (val1…val8) verbunden.

Der Ablauf: Wenn ein Ereignis an einem der **EI**-Eingänge auftritt, wird dieses über den Ereignis-Multiplexer an den Adapter-Multiplexer weitergeleitet. Dieser wählt den entsprechenden vorbereiteten **initval_AUDI**-Ausgang aus und stellt ihn am **OUT**-Adapter bereit. Somit liegt am Ausgang unmittelbar der zuvor über das Dateneingangsfeld konfigurierte AUDI-Wert an.

Technische Besonderheiten

- **Zweistufige Multiplexlogik**: Ereignis- und Adapterpfad sind getrennt, wodurch eine klare Trennung von Steuerung und Datenfluss erreicht wird.
- **8 parallele initval_AUDI-Instanzen**: Jeder Datenwert wird bereits beim Systemstart in einen AUDI-Adapter konvertiert und steht damit ohne Latenz zur Verfügung.
- **Keine Ereignisausgänge**: Die SubApp gibt kein Quittungs- oder Folgeereignis aus; die Auswahl ist rein ereignisgesteuert.
- **Adapterbasierte Ausgabe**: Der Ausgang ist ein unidirektionaler AUDI-Adapter, der direkt mit anderen Bausteinen desselben Typs verbunden werden kann.

Zustandsübersicht

Der Baustein besitzt keinen expliziten internen Zustandsautomaten. Er arbeitet ereignisgesteuert; es gibt keine dauerhaft gespeicherten Zustände. Nach einem Ereignis wird der entsprechende Datenwert unmittelbar am Ausgang ausgegeben. Eine parallele Verarbeitung mehrerer Ereignisse ist nicht vorgesehen – das zuletzt eingetroffene Ereignis bestimmt den Ausgang.

Anwendungsszenarien

- **Konfigurationsumschaltung**: Auswahl verschiedener Betriebsmodi oder Parameter zur Laufzeit (z. B. Geschwindigkeitsprofile, PIDs).
- **Test- und Simulationssysteme**: Umschalten zwischen mehreren simulierten Sensorwerten.
- **Redundante Datenquellen**: Auswahl einer aktiven Datenquelle aus mehreren parallelen Eingängen.
- **Audiotechnik**: Steuerung von Audioparametern (z. B. Lautstärke, Klangregelung) über ereignisgesteuerte Umschaltung.

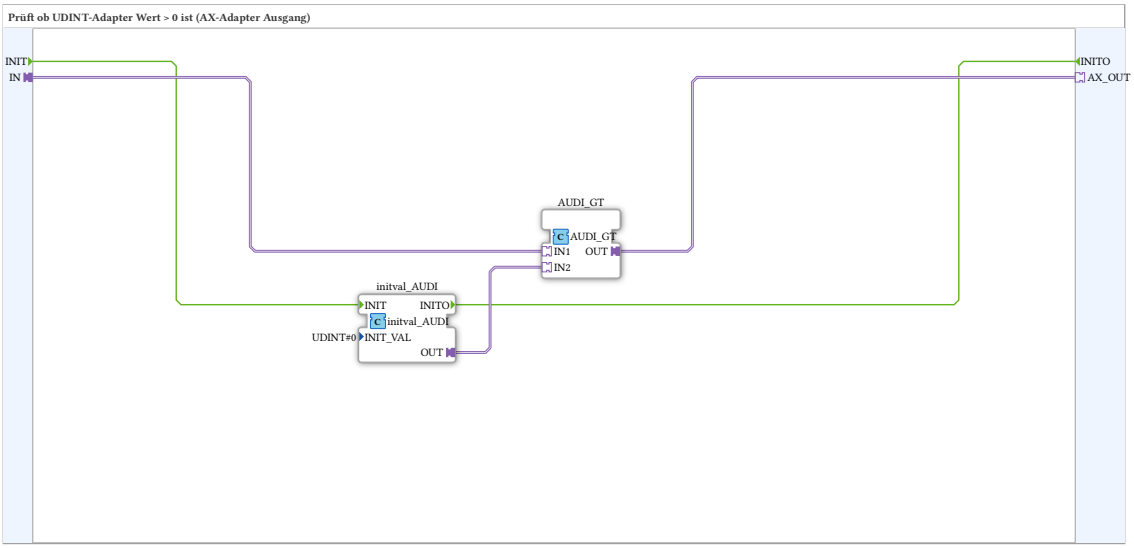
Vergleich mit ähnlichen Bausteinen

Gegenüber einem einfachen Daten-Multiplexer (z. B. mit direkten UDINT-Ausgaben) bietet dieser Baustein den Vorteil der **Adapter-Kompatibilität**: Die Ausgabe kann direkt an andere AUDI-fähige Bausteine angeschlossen werden, ohne zusätzliche Konvertierungslogik. Im Vergleich zu fest verdrahteten Auswahllogiken ermöglicht er eine flexible, ereignisgetriebene Steuerung. Nachteilig ist, dass keine Ereignis-Rückmeldung erfolgt und der Baustein ausschließlich für den AUDI-Typ ausgelegt ist.

Fazit

AUDI_AUI_MUX_8_VAL stellt eine robuste und einfach einsetzbare Lösung zur ereignisgesteuerten Auswahl eines von acht AUDI-Werten dar. Die Kombination aus Ereignis-Multiplexer und Adapter-Selektion reduziert den Verdrahtungsaufwand und erhöht die Wiederverwendbarkeit in komplexen IEC 61499-Systemen. Durch die Initialisierung der Eingangswerte beim Start ist die Ausgabe sofort verfügbar, ohne auf Datenkonvertierung warten zu müssen.

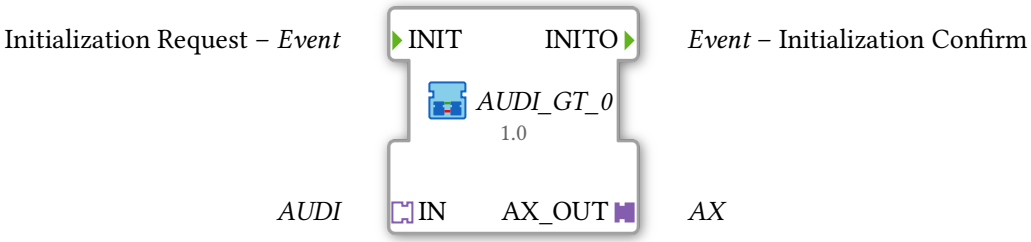
AUDI_GT_0



• – ○ • – ○ • – ○ •

Einleitung

`AUDI_GT_0` prüft, ob ein UDINT-Adapter-Wert größer als 0 ist, und gibt das Ergebnis als boolesches AX-Adapter-Signal aus. Typischer Eingangswert ist ein vorzeichenloser 32-Bit-Zählerwert. Der Baustein ist für jeden Zahlentyp gleich aufgebaut (`AS`, `AI`, `ADI`, `ALI`, `AUS`, `AUI`, `AUDI`, `AULI`, `AR`, `ALR`).



Schnittstelle

- **Ereignis-Eingang**: `INIT` (Initialisierung)
- **Ereignis-Ausgang**: `INITO` (Initialisierung abgeschlossen, kommt vom internen `initval_AUDI`)
- **Sockel**: `IN` (`AUDI`) — der Wert, der gegen 0 verglichen wird
- **Stecker**: `AX_OUT` (`AX`) — TRUE, wenn `IN > 0`

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: AUDI_GT_0

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **initval_AUDI**: `adapter::types::unidirectional::AUDI::initval::initval_AUDI` — liefert die feste Vergleichskonstante `UDINT#0` als Adapter-Wert (`INIT_VAL=UDINT#0`). Ein Adapter-Sockel kann die Konstante nicht als Parameter bekommen, deshalb wird sie über diesen Baustein eingespeist.
  - **AUDI_GT**: `adapter::iec61131::comparison::AUDI_GT` — adapter-nativer Vergleich `IN1 > IN2`.
- **Funktionsweise**: `IN` ⟶ `AUDI_GT.IN1`; Konstante 0 ⟶ `AUDI_GT.IN2`; Ergebnis ⟶ `AX_OUT`.

Programmablauf und Verbindungen

1. `INIT` ⟶ `initval_AUDI.INIT`; `initval_AUDI.INITO` ⟶ `INITO`.
2. `IN` (Adapter) ⟶ `AUDI_GT.IN1`.
3. `initval_AUDI.OUT` (Konstante `UDINT#0`) ⟶ `AUDI_GT.IN2`.
4. `AUDI_GT.OUT` ⟶ `AX_OUT` (Adapter).

Anwendungsszenarien

- Umwandlung von ein vorzeichenloser 32-Bit-Zählerwert in ein einfaches boolesches Adapter-Signal für VT-Statusanzeigen oder Freigabelogik.

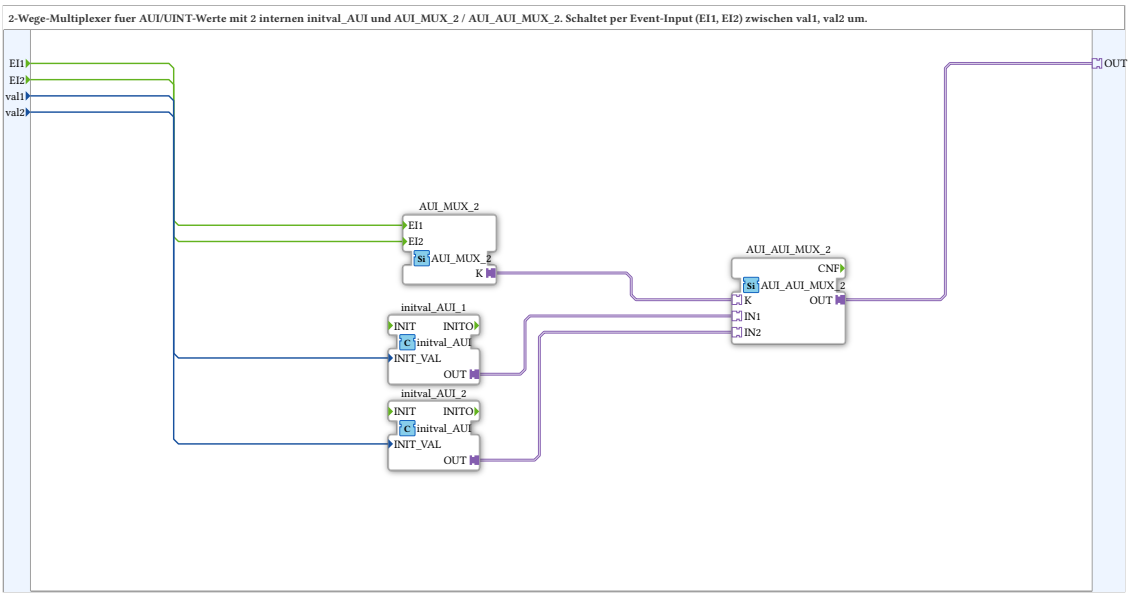
Zusammenfassung

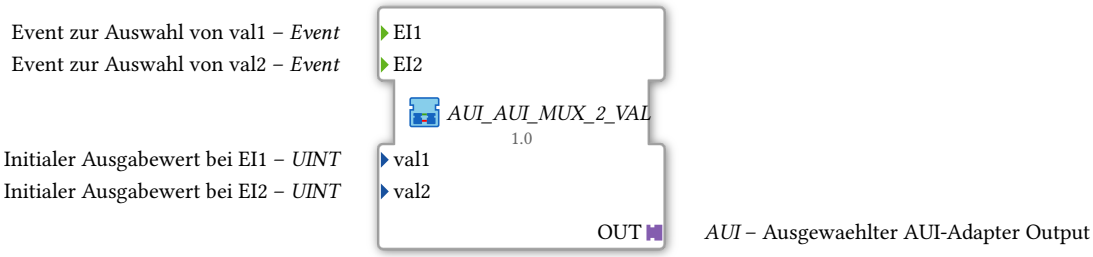
`AUDI_GT_0` ist ein kompakter Wrapper um den adapter-nativen Vergleich `AUDI_GT` — wandelt „UDINT > 0“ direkt in ein AX-Adapter-Signal um, ohne dass der Aufrufer die Null-Konstante selbst anlegen und verdrahten muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUI_AUI_MUX_2_VAL





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AUI_AUI_MUX_2_VAL` ist eine Subapplikation (SubApp), die einen 2-Wege-Multiplexer für AUI/UINT-Werte realisiert. Er nimmt zwei UINT-Eingangswerte entgegen und stellt je nach ausgelöstem Event den entsprechenden Wert über einen AUI-Adapterausgang bereit. Die Umschaltung erfolgt über die Ereigniseingänge `EI1` und `EI2`.

Schnittstellenstruktur

Ereignis-Eingänge

- **EI1** (Event): Wählt den Wert von `val1` als aktiven Ausgangswert aus.
- **EI2** (Event): Wählt den Wert von `val2` als aktiven Ausgangswert aus.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

- **val1** (UINT): Erster Auswahlwert. Er wird beim Eintreffen von `EI1` auf den Ausgang gelegt.
- **val2** (UINT): Zweiter Auswahlwert. Er wird beim Eintreffen von `EI2` auf den Ausgang gelegt.

Daten-Ausgänge

Keine direkten Datenausgänge. Der ausgewählte Wert wird ausschließlich über den Adapterausgang `OUT` bereitgestellt.

Adapter

- **OUT** (AUI-Adapter, unidirektional): Ausgewählter AUI-Adapterausgang. Liefert den über `EI1` bzw. `EI2` ausgewählten Wert als AUI-Datenstrom.

Funktionsweise

Die SubApp enthält intern die Funktionsbausteine `initval_AUI_1`, `initval_AUI_2`, `AUI_MUX_2` und `AUI_AUI_MUX_2`. Die an `val1` und `val2` anliegenden UINT-Werte werden über die beiden `initval_AUI`-Bausteine in AUI-Adapterwerte umgewandelt und den Eingängen `IN1` bzw. `IN2` des Bausteins `AUI_AUI_MUX_2` zugeführt.

Wird das Ereignis `EI1` ausgelöst, erzeugt der Baustein `AUI_MUX_2` über den Ausgang `K` ein Auswahlsignal für den Multiplexer `AUI_AUI_MUX_2`. Dadurch wird der am Eingang `IN1` anliegende AUI-Wert ausgewählt und zum Ausgang `OUT` durchgeschaltet. Analog dazu wählt `EI2` den Wert am Eingang `IN2` aus.

Technische Besonderheiten

- Die SubApp kombiniert einen eventgesteuerten Auswahlbaustein (`AUI_MUX_2`) mit einem adapterbasierten Multiplexer (`AUI_AUI_MUX_2`).
- Die Umwandlung von UINT nach AUI erfolgt über zwei interne Instanzen des Bausteins `initval_AUI`.
- Die SubApp besitzt keinen direkten Datenausgang; der Ergebniswert wird ausschließlich über den Adapter `OUT` übertragen.
- Die Auswahl-Events beeinflussen nicht den Datenfluss, sondern steuern nur die Umschaltung des Multiplexers.

Zustandsübersicht

Da es sich um eine SubApp handelt, gibt es keinen expliziten Zustandsautomaten auf oberster Ebene. Logisch lassen sich folgende Zustände unterscheiden:

- **Bereit**: Die SubApp wartet auf ein Ereignis an `EI1` oder `EI2`.
- **Auswahl val1**: Nach `EI1` wird der Wert von `IN1` (aus `val1`) über `OUT` ausgegeben.
- **Auswahl val2**: Nach `EI2` wird der Wert von `IN2` (aus `val2`) über `OUT` ausgegeben.

Anwendungsszenarien

- Umschaltung zwischen zwei Sollwerten oder Parametern in einer Steuerung.
- Auswahl unterschiedlicher Konfigurationswerte zur Laufzeit.
- Wechsel zwischen zwei AUI-basierten Signalquellen in einer Automatisierungsanwendung.
- Bereitstellung von Initialwerten für nachgeschaltete AUI-basierte Funktionsbausteine.

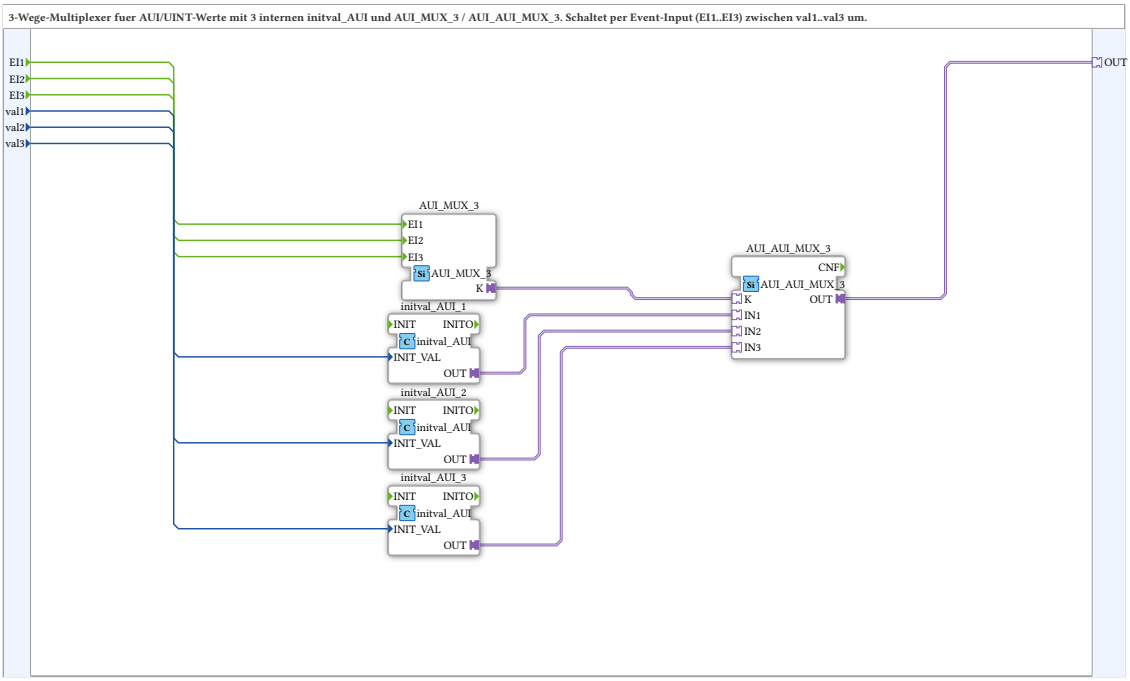
Vergleich mit ähnlichen Bausteinen

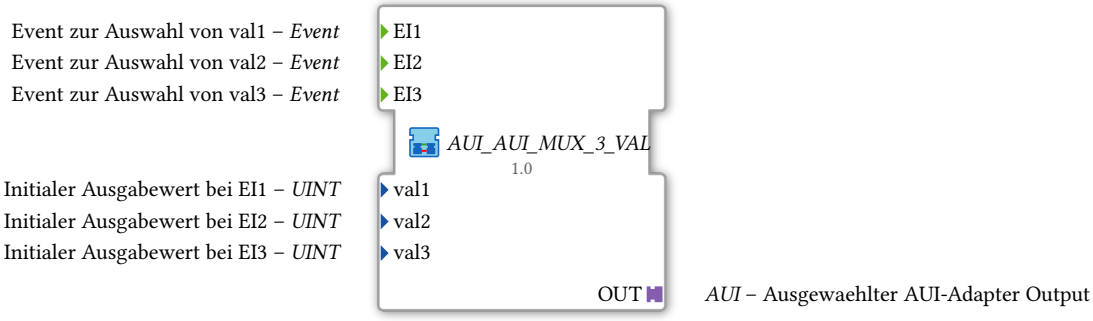
Im Gegensatz zu einem direkten `AUI_AUI_MUX_2`-Baustein, der bereits fertige AUI-Werte an seinen Eingängen erwartet, besitzt `AUI_AUI_MUX_2_VAL` zusätzlich die beiden UINT-Eingänge `val1` und `val2`. Dadurch kann der Anwender die Auswahlwerte direkt als einfache Ganzzahlen vorgeben, ohne vorher eine separate Umwandlung in AUI-Adapter vornehmen zu müssen. Der Baustein `AUI_MUX_2` übernimmt hierbei die Ereignisverarbeitung und erzeugt das Auswahlsignal für den Adapter-Multiplexer.

Fazit

Der Funktionsblock `AUI_AUI_MUX_2_VAL` bietet eine kompakte und einfach zu verwendende Lösung zur Auswahl zwischen zwei UINT-Werten über AUI-Adapter. Durch die interne Kombination von UINT-zu-AUI-Wandlung und adapterbasiertem Multiplexing wird die Anwendung in Steuerungslogiken deutlich vereinfacht.

AUI_AUI_MUX_3_VAL





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUI_AUI_MUX_3_VAL** ist ein 3-Wege-Multiplexer für AUI-Adapterwerte (unidirektionaler Typ) mit drei vorgegebenen UINT-Initialwerten. Er ermöglicht es, über drei Ereignis-Eingänge (EI1-EI3) zwischen drei verschiedenen Eingangswerten (val1-val3) umzuschalten. Der ausgewählte Wert wird über einen AUI-Adapterausgang (OUT) bereitgestellt. Der Baustein integriert dabei einen Ereignis-Multiplexer (AUI_MUX_3), einen Adapter-Multiplexer (AUI_AUI_MUX_3) sowie drei Initialisierungsbausteine (initval_AUI), die die Anfangswerte setzen und intern in Adapterwerte umwandeln.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Datentyp	Kommentar
EI1	Event	Event zur Auswahl von val1
EI2	Event	Event zur Auswahl von val2
EI3	Event	Event zur Auswahl von val3

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Name	Datentyp	Kommentar
val1	UINT	Initialer Ausgabewert bei EI1
val2	UINT	Initialer Ausgabewert bei EI2
val3	UINT	Initialer Ausgabewert bei EI3

Daten-Ausgänge

Keine vorhanden.

Adapter

Name	Richtung	Typ	Kommentar
OUT	Ausgang	adapter::types::unidirectional::AUI	Ausgewählter AUI-Adapter Output

Funktionsweise

Die Eingangsereignisse EI1, EI2 und EI3 werden direkt an den internen Ereignis-Multiplexer `AUI_MUX_3` weitergeleitet. Abhängig davon, welches Ereignis eintritt, setzt dieser das Steuersignal für den Adapter-Multiplexer `AUI_AUI_MUX_3` entsprechend. Gleichzeitig werden die drei UINT-Eingangswerte val1, val2 und val3 über je einen `initval_AUI`-Baustein in AUI-Adapterwerte umgewandelt und als IN1, IN2 und IN3 an den Adapter-Multiplexer übergeben. Der Adapter-Multiplexer wählt anhand des Steuersignals (K) genau einen dieser drei Werte aus und stellt ihn am Ausgang OUT bereit.

Die Verbindung zwischen `AUI_MUX_3.K` und `AUI_AUI_MUX_3.K` überträgt die vom Ereignis-Multiplexer erzeugte Auswahlinformation. Somit entspricht jeder Ereignis-Eingang einem festen Daten-Eingang: EI1 → val1, EI2 → val2, EI3 → val3.

Technische Besonderheiten

- **Initialisierung über initval_AUI:** Die drei UINT-Werte werden durch die internen FB `initval_AUI_1`, `initval_AUI_2` und `initval_AUI_3` in AUI-Adapterwerte konvertiert. Dies ermöglicht eine saubere Schnittstellenanpassung zwischen Standard-Daten (UINT) und dem AUI-Adapterprotokoll.
- **Zwei-Stufen-Implementierung:** Die Trennung von Ereignis- und Adapter-Multiplexer erlaubt eine flexible Erweiterung oder Wiederverwendung einzelner Komponenten.
- **Unidirektionaler Adapter:** Der Ausgang ist als unidirektionaler AUI-Adapter definiert, d.h. er sendet nur Daten in eine Richtung.

Zustandsübersicht

Der Baustein besitzt keine expliziten Zustände oder Zustandsautomaten. Die Funktionsweise ist rein ereignisgesteuert: Jedes Ereignis selektiert sofort den zugehörigen Eingangswert. Nach der Verarbeitung bleibt der Ausgangswert bis zum nächsten Ereignis stabil.

Anwendungsszenarien

- Umschaltung zwischen verschiedenen Messwerten (als UINT) in einer Automatisierungsumgebung, die über AUI-Adapter kommuniziert.
- Auswahl unterschiedlicher Sollwerte für einen Prozessregler basierend auf Betriebsmodi oder Sensoren.
- Konfiguration von Parametern zur Laufzeit, indem verschiedene vordefinierte Werte über Ereignisse ausgewählt werden.

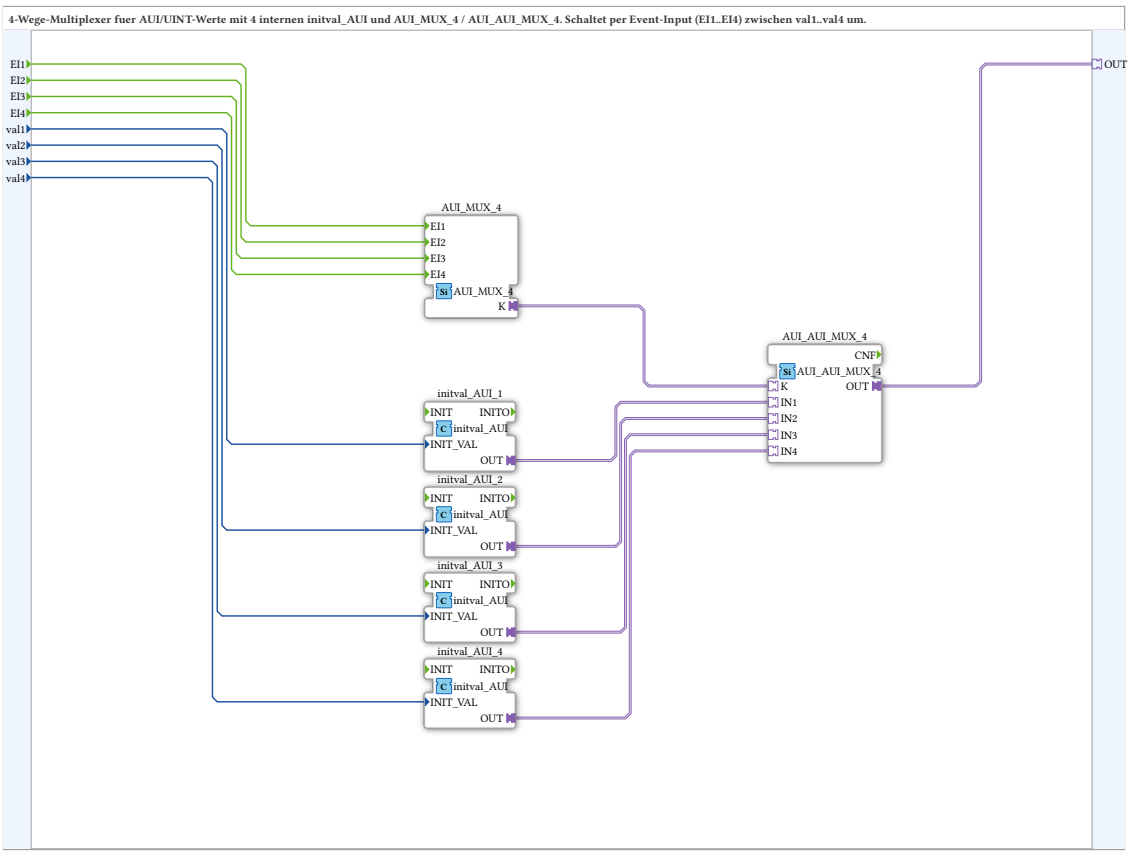
Vergleich mit ähnlichen Bausteinen

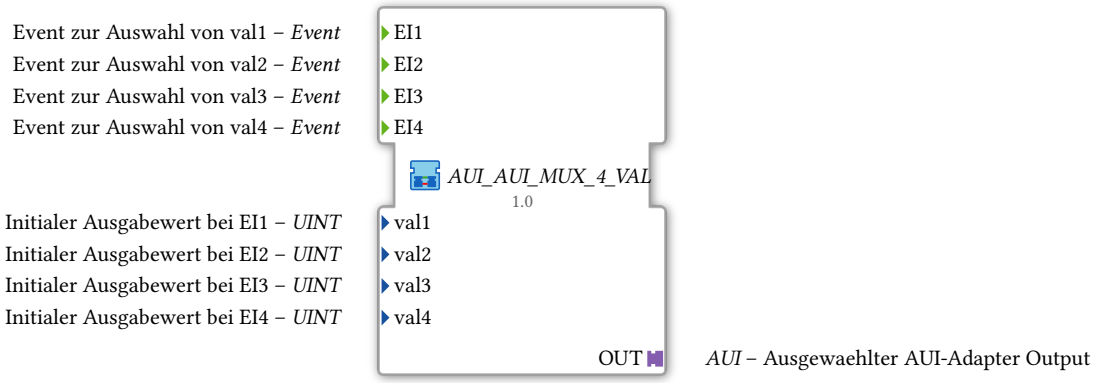
Im Vergleich zu einem einfachen Daten-Multiplexer (z.B. MUX) bietet dieser Baustein eine direkte Integration mit AUI-Adaptertypen und eine ereignisbasierte Auswahl, wodurch er sich für IEC 61499-Anwendungen eignet, die adapterbasierte Kommunikation verwenden. Gegenüber einem zusätzlichen manuellen Setzen des Auswahlindexes werden hier die Ereignisse direkt verwendet, was die logische Verknüpfung vereinfacht. Ähnliche Bausteine wie `AUI_MUX_2_VAL` oder `AUI_AUI_MUX_4_VAL` könnten für geringere oder höhere Kanalzahlen existieren, bieten jedoch dieselbe prinzipielle Funktionsweise.

Fazit

Der FB `AUI_AUI_MUX_3_VAL` stellt eine kompakte und robuste Lösung zur Auswahl eines von drei AUI-Werten dar, wobei die Werte über UINT-Eingänge vorgegeben und initialisiert werden. Die Kombination aus Ereignis- und Adapter-Multiplexer sowie die eingebauten Initialisierungsbausteine machen ihn zu einem nützlichen Werkzeug in industriellen Automatisierungssystemen, die auf IEC 61499 und AUI-Adapter basieren. Durch seine klare Schnittstellenstruktur und ereignisgesteuerte Arbeitsweise lässt er sich einfach in bestehende Steuerungsarchitekturen integrieren.

AUI_AUI_MUX_4_VAL





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUI_AUI_MUX_4_VAL** ist ein 4-Wege-Multiplexer für AUI/UINT-Werte. Er wählt über ein Ereignis an einem der vier Eingänge einen der vier Datenwerte (val1...val4) aus und stellt diesen als AUI-Adapter-Ausgang bereit. Die Auswahl erfolgt ereignisgesteuert und wird durch die Kombination eines Ereignis-Multiplexers und eines Adapter-Multiplexers realisiert. Zusätzlich werden die Eingangswerte über interne `initval_AUI`-Bausteine initialisiert, um ein definiertes Startverhalten zu gewährleisten.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Kommentar
EI1	Event zur Auswahl von val1
EI2	Event zur Auswahl von val2
EI3	Event zur Auswahl von val3
EI4	Event zur Auswahl von val4

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Daten	Typ	Kommentar
val1	UINT	Initialer Ausgabewert bei EI1
val2	UINT	Initialer Ausgabewert bei EI2
val3	UINT	Initialer Ausgabewert bei EI3
val4	UINT	Initialer Ausgabewert bei EI4

Daten-Ausgänge

Keine (Ausgabe erfolgt über den Adapter).

Adapter

Adapter	Typ	Richtung	Kommentar
OUT	`adapter::types::unidirectional::AUI`	Ausgang	Ausgewählter AUI-Adapter Output

Funktionsweise

Der Baustein verwendet intern drei Arten von Bausteinen:

- **`AUI_MUX_4`**: Ein Ereignis-Multiplexer, der die ankommenden Events (EI1…EI4) entgegennimmt und ein entsprechendes Auswahlsignal an den Adapter-Multiplexer weiterleitet.
- **`AUI_AUI_MUX_4`**: Ein Adapter-Multiplexer, der anhand des erhaltenen Auswahlsignals einen der vier Eingänge (IN1…IN4) auf den Ausgang (OUT) schaltet.
- **Vier `initval_AUI`-Instanzen**: Diese setzen den jeweils übergebenen UINT-Wert (val1…val4) in einen AUI-Adapter-Wert um und stellen diesen am jeweiligen Eingang des Adapter-Multiplexers bereit.

Wird ein Ereignis an einem der Eingänge ausgelöst (z. B. EI1), so aktiviert der `AUI_MUX_4` die entsprechende Auswahlleitung. Gleichzeitig werden die aktuellen Werte der Daten-Eingänge über die `initval_AUI`-Bausteine an die Eingänge des `AUI_AUI_MUX_4` geführt. Der Adapter-Multiplexer übernimmt dann den passenden Wert an seinem Ausgang `OUT`. Somit wird der Wert, der zum Zeitpunkt des Ereignisses an `val1` anliegt, als AUI-Wert ausgegeben.

Technische Besonderheiten

- **Verwendung von AUI-Adaptern**: Die Ausgabe ist als unidirektionaler AUI-Adapter ausgeführt, was eine einfache Kopplung an weitere Bausteine mit AUI-Schnittstelle ermöglicht.
- **Zweistufige Multiplex-Logik**: Die Trennung von Ereignis-Auswahl (durch `AUI_MUX_4`) und Daten-Auswahl (durch `AUI_AUI_MUX_4`) erlaubt eine klare Struktur und erleichtert die Erweiterung auf mehr Kanäle.
- **Initialisierung**: Jeder Eingangswert wird über eine `initval_AUI`-Instanz initialisiert. Dadurch ist gewährleistet, dass bereits vor dem ersten Ereignis definierte Werte an den Multiplexer-Eingängen anliegen.
- **Reine Ereignissteuerung**: Es gibt keinen internen Zustandsspeicher; die aktive Auswahl wird eindeutig durch das zuletzt empfangene Ereignis bestimmt.

Zustandsübersicht

Der Baustein besitzt keinen expliziten internen Zustand im Sinne einer Zustandsmaschine. Die aktive Auswahl wird ausschließlich durch das zuletzt empfangene Ereignis (EI1…EI4) bestimmt. Nach einem Reset oder einer Initialisierung ist kein Wert aktiv – erst ein ankommendes Ereignis legt den auszugebenden Wert fest. Daher kann man von einem ereignisgesteuerten Auswahlverhalten ohne persistenten Zustand sprechen.

Anwendungsszenarien

- **Parametrisierung**: Auswahl unterschiedlicher Konfigurationswerte (z. B. Geschwindigkeit, Druck, Temperatur) in einer Steuerung über einzelne Triggersignale.
- **Datenumschaltung**: Umschalten zwischen verschiedenen Datenquellen (z. B. Sensoren) bei Bedarf.
- **Test- und Simulationsumgebungen**: Gezieltes Einspielen von Testwerten über die Events.
- **Adapter-basierte Architekturen**: Einsatz in Systemen, die auf AUI-Adaptern für die Kommunikation zwischen Funktionsblöcken basieren.

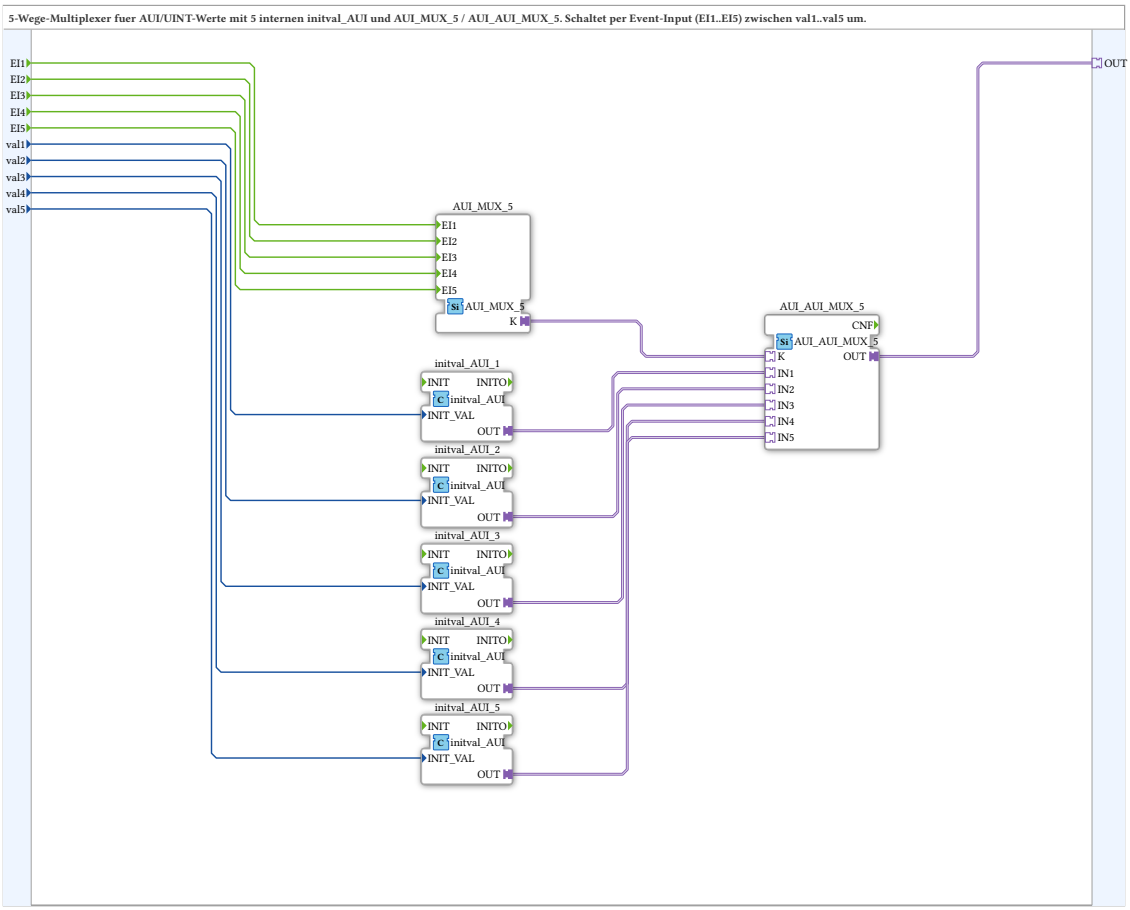
Vergleich mit ähnlichen Bausteinen

Gegenüber einem einfachen Daten-Multiplexer (z. B. mit `SELECT`-Eingang) bietet dieser Baustein eine ereignisgesteuerte Auswahl, was in Echtzeit-Systemen oft vorteilhaft ist, da keine aktive Polling-Logik benötigt wird. Im Vergleich zu einem Multiplexer ohne `initval`-Unterstützung wird hier eine definierte Initialisierung der Eingänge gewährleistet. Die Verwendung von AUI-Adaptern auf der Ausgabeseite macht den Baustein besonders für Systeme geeignet, die bereits auf AUI-Kommunikation setzen.

Fazit

Der **AUI_AUI_MUX_4_VAL** ist ein flexibler und klar strukturierter 4-Kanal-Multiplexer für AUI-Werte. Durch die Kombination eines Ereignis-Multiplexers mit einem Adapter-Multiplexer und der Integration von Initialisierungs-Bausteinen bietet er eine robuste und erweiterbare Lösung zur Auswahl von Datenwerten. Die ereignisgesteuerte Arbeitsweise macht ihn ideal für Automatisierungsanwendungen, bei denen schnelle und deterministische Umschaltungen erforderlich sind. Die Dokumentation und die klare Schnittstellenstruktur ermöglichen eine einfache Integration in bestehende Projekte.

AUI_AUI_MUX_5_VAL



Event zur Auswahl von val1 – *Event* — EI1
Event zur Auswahl von val2 – *Event* — EI2
Event zur Auswahl von val3 – *Event* — EI3
Event zur Auswahl von val4 – *Event* — EI4
Event zur Auswahl von val5 – *Event* — EI5

AUI_AUI_MUX_5_VAL
1.0

Initialer Ausgabewert bei EI1 – *UINT* — val1
Initialer Ausgabewert bei EI2 – *UINT* — val2
Initialer Ausgabewert bei EI3 – *UINT* — val3
Initialer Ausgabewert bei EI4 – *UINT* — val4
Initialer Ausgabewert bei EI5 – *UINT* — val5

OUT — *AUI* – Ausgewaehlter AUI-Adapter Output

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUI_AUI_MUX_5_VAL** ist ein 5-Wege-Multiplexer für AUI/UINT-Werte. Er wählt abhängig von einem eingehenden Ereignis (EI1 bis EI5) einen von fünf konfigurierbaren Ausgabewerten (val1 bis val5) aus und stellt diesen über einen AUI-Adapter (OUT) bereit. Intern werden die Eingangswerte mithilfe von `initval_AUI`-Bausteinen in AUI-Adapter konvertiert und über einen Adapter-Multiplexer an den Ausgang geschaltet.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
EI1	Event	Event zur Auswahl von val1
EI2	Event	Event zur Auswahl von val2
EI3	Event	Event zur Auswahl von val3
EI4	Event	Event zur Auswahl von val4
EI5	Event	Event zur Auswahl von val5

Ereignis-Ausgänge
Keine.

Daten-Eingänge

Name	Typ	Kommentar
val1	UINT	Initialer Ausgabewert bei EI1
val2	UINT	Initialer Ausgabewert bei EI2
val3	UINT	Initialer Ausgabewert bei EI3
val4	UINT	Initialer Ausgabewert bei EI4
val5	UINT	Initialer Ausgabewert bei EI5

Daten-Ausgänge
Keine – die Ausgabe erfolgt ausschließlich über den Adapterausgang.

Adapter

Name	Richtung	Typ	Kommentar
OUT	Ausgang	adapter::types::unidirectional::AUI	Ausgewählter AUI-Adapter-Output

Funktionsweise
Die Subapp besteht aus mehreren internen Funktionsblöcken, die wie folgt verschaltet sind:

1. **Ereignisweiterleitung** – Die fünf Ereignisse EI1 bis EI5 werden direkt an den Funktionsblock `AUI_MUX_5` (vom Typ `adapter::events::unidirectional::AUI_MUX_5`) geleitet. Dieser dient als Ereignis-Multiplexer und gibt das empfangene Ereignis an den Adapter-Multiplexer weiter.
2. **Wertkonvertierung** – Die Dateneingänge val1 bis val5 werden jeweils einem eigenen `initval_AUI`-Baustein (Typ `adapter::types::unidirectional::AUI::initval::initval_AUI`) zugeführt. Diese Bausteine erzeugen aus einem UINT-Initialwert einen AUI-Adapter mit dem entsprechenden Wert.
3. **Adapterauswahl** – Der Funktionsblock `AUI_AUI_MUX_5` (Typ `adapter::selection::unidirectional::AUI_AUI_MUX_5`) empfängt die von den `initval_AUI`-Bausteinen bereitgestellten Adapter (IN1 bis IN5) sowie das vom Ereignis-Multiplexer übergebene Steuersignal (K). Anhand dieses Signals wird der passende Eingang auf den Ausgang OUT geschaltet.
4. **Ausgabe** – Der ausgewählte AUI-Adapter wird über den externen Adapterausgang OUT der Subapp bereitgestellt.

Durch das Auslösen eines Ereignisses EIx wird also der zugehörige Wert valx (als AUI-Adapter) weitergegeben.

Technische Besonderheiten

- **Verwendung von AUI-Adaptern**: Die Datenübertragung erfolgt nicht über herkömmliche Datenausgänge, sondern über unidirektionale Adapter, was eine flexible und wiederverwendbare Schnittstellengestaltung ermöglicht.
- **Interne Wertinitialisierung**: Die `initval_AUI`-Bausteine dienen dazu, aus einem einfachen UINT-Wert einen AUI-Adapter zu erzeugen. Dies erlaubt es, die Ausgabewerte direkt an der Subapp zu konfigurieren.
- **Paketzuordnung**: Die Subapp ist im Paket `MyLib::sys` eingeordnet und folgt dem Standard IEC 61499-2.
- **Lizenz**: Der Baustein ist unter der Eclipse Public License 2.0 (EPL-2.0) verfügbar (siehe Copyright-Hinweis in der XML-Beschreibung).

Zustandsübersicht

Da es sich um einen Multiplexer handelt, existiert kein komplexer Zustandsautomat. Das Verhalten ist rein ereignisgesteuert: Zu jedem Zeitpunkt ist genau einer der fünf Eingänge aktiviert, und der Ausgang liefert den entsprechenden Wert. Ein interner Zustand wird lediglich durch das zuletzt ausgelöste Ereignis bestimmt, solange kein neues Ereignis eintrifft. Eine explizite Initialisierung ist nicht erforderlich; sobald das erste Ereignis eintritt, wird der zugehörige Wert ausgegeben.

Anwendungsszenarien

- **Konfigurationsauswahl in Automatisierungssystemen**: Auswahl unterschiedlicher Parameter oder Profile (z. B. Geschwindigkeiten, Zeitwerte) über Ereignisse.
- **Umschaltung zwischen mehreren Sensorwerten**: Wenn mehrere Sensoren Daten über AUI-Schnittstellen bereitstellen, kann je nach Ereignis ein bestimmter Sensorwert zur weiteren Verarbeitung ausgewählt werden.
- **Test- und Simulationsumgebungen**: Schnelles Umschalten zwischen verschiedenen Testwerten während der Entwicklung.

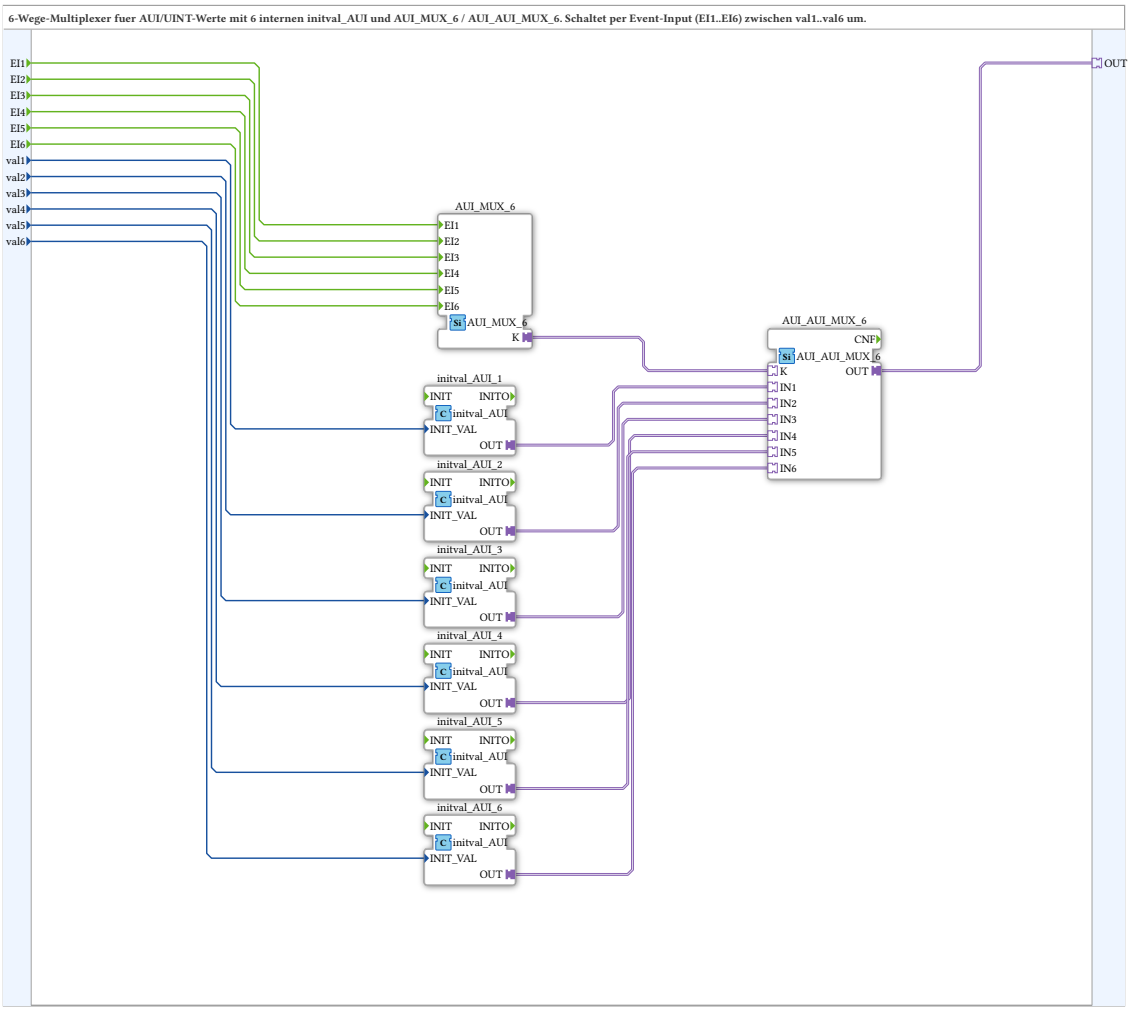
Vergleich mit ähnlichen Bausteinen

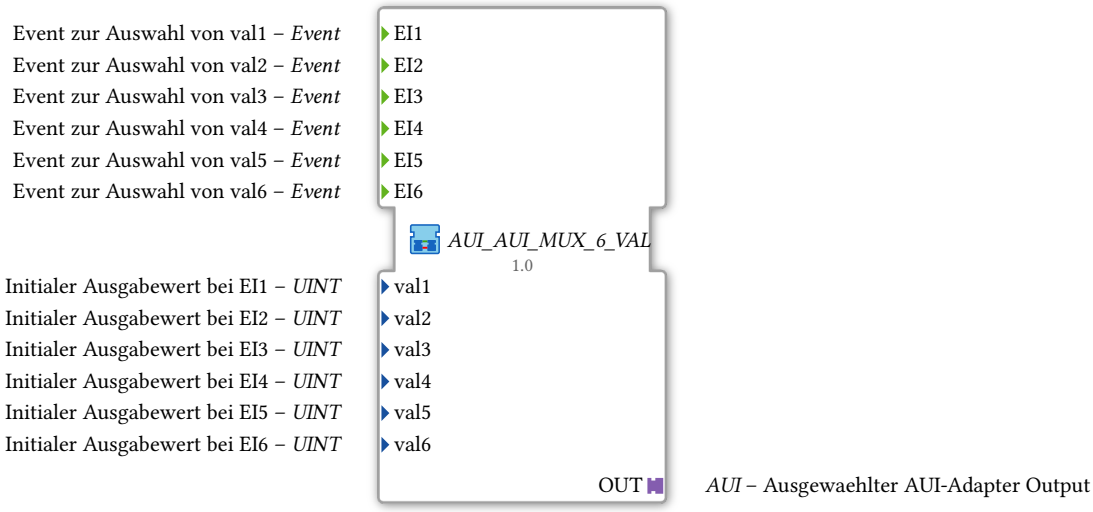
Ein einfacher Multiplexer (z. B. `MUX`) arbeitet direkt mit elementaren Datentypen wie UINT und besitzt separate Datenausgänge. Im Gegensatz dazu nutzt `AUI_AUI_MUX_5_VAL` ausschließlich AUI-Adapter für die Ausgabe und integriert eine Initialisierung der Werte über `initval_AUI`. Dadurch ist eine direkte Anbindung an andere AUI-basierte Komponenten möglich, ohne zusätzliche Konvertierungslogik. Der Baustein bietet eine kompakte und wiederverwendbare Lösung für AUI-basierte Multiplexaufgaben.

Fazit

Der Funktionsblock **AUI_AUI_MUX_5_VAL** stellt eine flexible und klar strukturierte Lösung zur Auswahl eines von fünf AUI-Werten dar. Durch die Kombination von Ereignissteuerung, UINT-Initialisierung und Adapterausgabe eignet er sich ideal für den Einsatz in modularen Automatisierungs- und Steuerungssystemen, die auf AUI-Schnittstellen basieren. Die interne Verschaltung ist nachvollziehbar und erleichtert Wartung und Erweiterung.

AUI_AUI_MUX_6_VAL





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUI_AUI_MUX_6_VAL** ist ein 6-Wege-Multiplexer für AUI/UINT-Werte. Er enthält sechs interne `initval_AUI`-Bausteine sowie die beiden Adapter-Bausteine `AUI_MUX_6` und `AUI_AUI_MUX_6`. Über die Ereignis-Eingänge `EI1` bis `EI6` kann jeweils einer der sechs Eingangswerte `val1` bis `val6` ausgewählt und als AUI-Adapter-Signal am Ausgang `OUT` bereitgestellt werden.

Schnittstellenstruktur

Ereignis-Eingänge

- `EI1` : Event – Event zur Auswahl von `val1`
- `EI2` : Event – Event zur Auswahl von `val2`
- `EI3` : Event – Event zur Auswahl von `val3`
- `EI4` : Event – Event zur Auswahl von `val4`
- `EI5` : Event – Event zur Auswahl von `val5`
- `EI6` : Event – Event zur Auswahl von `val6`

Ereignis-Ausgänge

Keine

Daten-Eingänge

- `val1` : UINT – Initialer Ausgabewert bei `EI1`
- `val2` : UINT – Initialer Ausgabewert bei `EI2`
- `val3` : UINT – Initialer Ausgabewert bei `EI3`
- `val4` : UINT – Initialer Ausgabewert bei `EI4`
- `val5` : UINT – Initialer Ausgabewert bei `EI5`
- `val6` : UINT – Initialer Ausgabewert bei `EI6`

Daten-Ausgänge

Keine

Adapter

- `OUT` : Adapter vom Typ `adapter::types::unidirectional::AUI` – Ausgewählter AUI-Adapter Output

Funktionsweise

Die externen Ereignisse `EI1` bis `EI6` werden direkt an den Adapter-Baustein `AUI_MUX_6` weitergeleitet. Parallel dazu werden die UINT-Eingangswerte `val1` bis `val6` über die internen `initval_AUI`-Bausteine in AUI-Adapter-Signale umgewandelt. Diese sechs AUI-Signale werden dem Auswahl-Baustein `AUI_AUI_MUX_6` über die Eingänge `IN1` bis `IN6` zugeführt. Das vom `AUI_MUX_6` erzeugte Selektionssignal (K) bestimmt, welcher der sechs Eingänge auf den Ausgang `OUT` durchgeschaltet wird. Somit wird durch das Auftreten eines Ereignisses `EIi` der zugehörige Wert `vali` als AUI-Signal am Ausgang aktiviert.

Technische Besonderheiten

- Verwendung unidirektionaler AUI-Adapter für den Datentransport.
- Interne Umwandlung von UINT-Werten in AUI-Adapter über `initval_AUI`.
- Ereignisgesteuerte Auswahl ohne Datenausgänge – das Ergebnis wird ausschließlich über den Adapter `OUT` bereitgestellt.
- Keine internen Zustände; die Funktion ist rein kombinatorisch und ereignisgetrieben.

Zustandsübersicht

Der Baustein besitzt keinen expliziten internen Zustand. Die Auswahl erfolgt direkt durch die eintreffenden Ereignisse. Nach einem Ereignis `EIi` bleibt der am Ausgang anliegende Wert `vali` solange aktiv, bis ein anderes Ereignis ausgelöst wird.

Anwendungsszenarien

- **Umschaltung von Sollwerten oder Parametern**: In Automatisierungssystemen können über Events verschiedene Konfigurationswerte (z. B. Geschwindigkeiten, Temperaturen) ausgewählt und als AUI-Signale an weitere Funktionen weitergereicht werden.
- **Flexible Datenquellen**: Wenn mehrere AUI-Quellen zur Verfügung stehen und je nach Betriebsmodus eine davon aktiv sein soll, kann dieser Multiplexer die Umschaltung übernehmen.
- **Test- und Simulationsumgebungen**: Zur gezielten Aktivierung unterschiedlicher Testwerte über Ereignissteuerung.

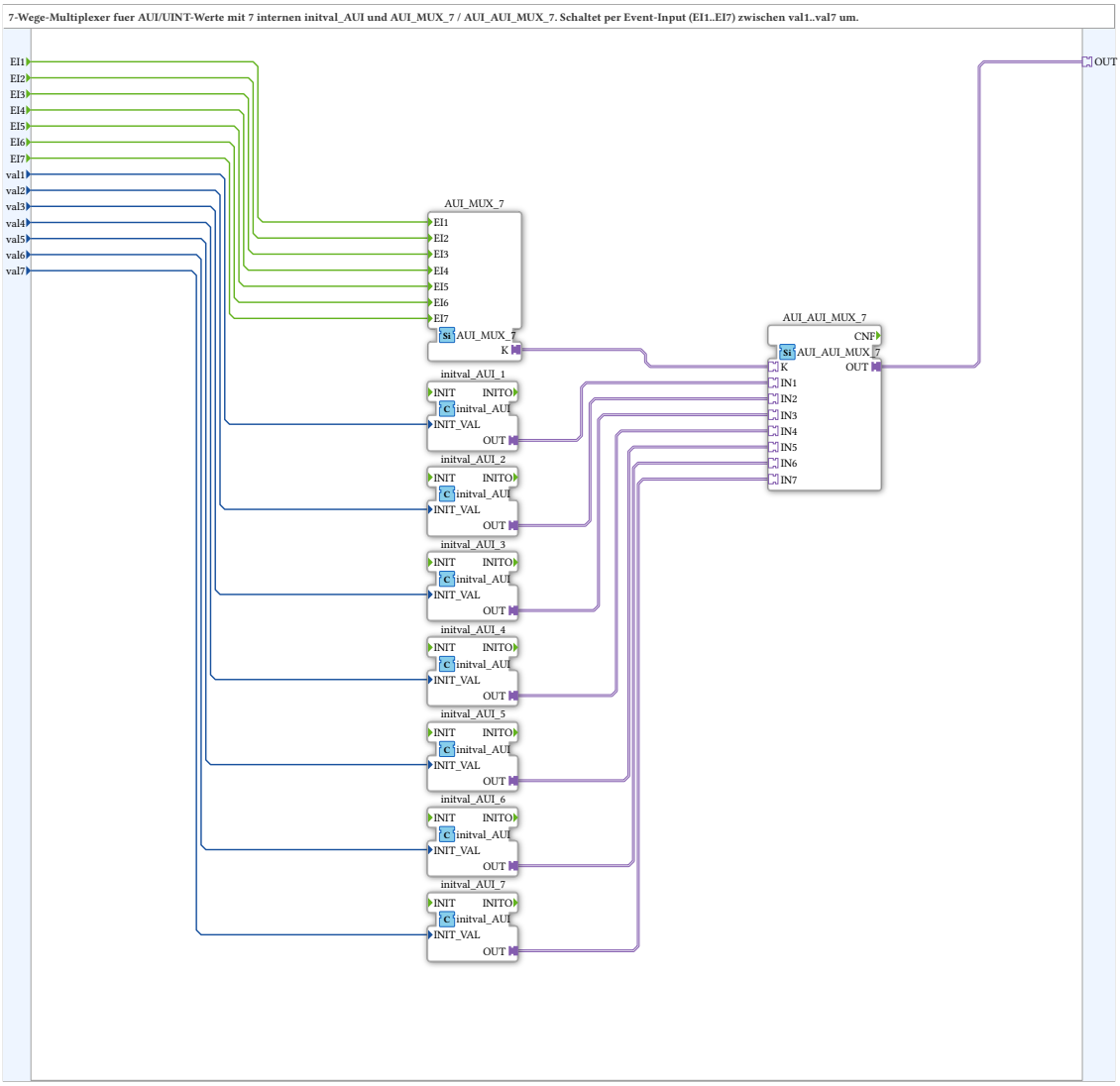
Vergleich mit ähnlichen Bausteinen

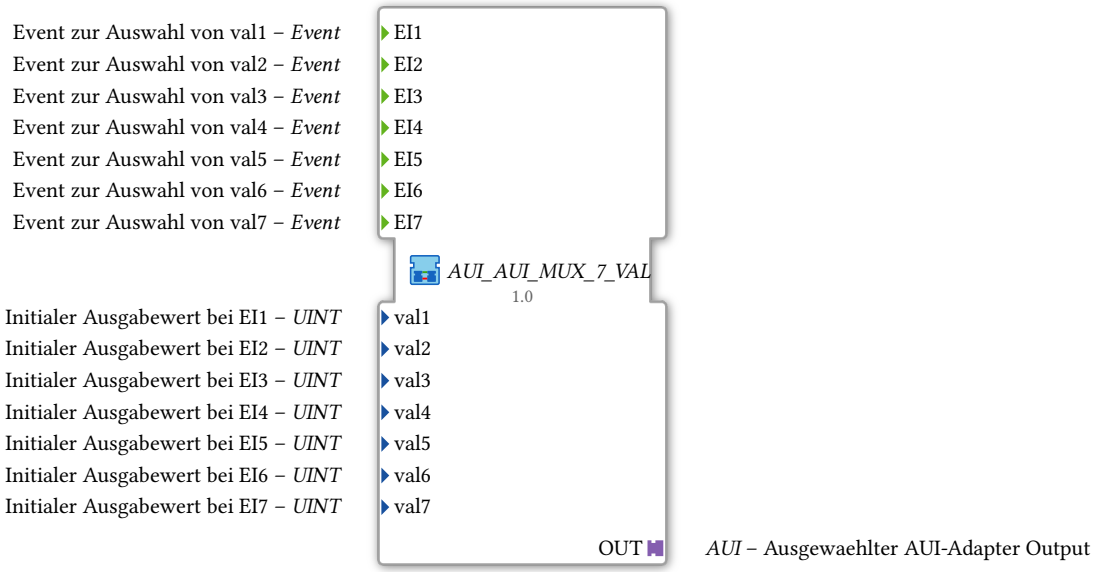
- Im Gegensatz zu einem klassischen Multiplexer mit mehreren Daten-Eingängen und einem Datenausgang arbeitet dieser Baustein mit Adapter-Schnittstellen (AUI). Die ausgewählten Werte werden nicht als direkte Variable, sondern als Adapter-Signal weitergereicht.
- Innerhalb der 4diac-Adapternetzwerke bieten solche Bausteine eine typsichere und ereignisorientierte Auswahl, während einfache MUX-Bausteine meist über einen selektierenden Datenwert gesteuert werden.

Fazit

Der **AUI_AUI_MUX_6_VAL** stellt eine effiziente und flexible Lösung dar, um aus sechs UINT-Werten über Ereignisse einen auszuwählen und als AUI-Adapter-Signal auszugeben. Durch die Verwendung von Adaptern wird eine saubere und typisierte Anbindung an weitere Komponenten ermöglicht. Die integrierte Umwandlung von UINT zu AUI macht den Baustein direkt einsetzbar, ohne zusätzliche Konvertierungslogik. Dadurch eignet er sich hervorragend für Anwendungen, in denen mehrere konfigurierbare Parameter je nach Steuerungszustand aktiviert werden müssen.

AUI_AUI_MUX_7_VAL





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AUI_AUI_MUX_7_VAL** ist ein 7-Wege-Multiplexer, der AUI/UINT-Werte verarbeitet. Über sieben Ereignis-Eingänge (EI1..EI7) kann jeweils einer von sieben UINT-Werten (val1..val7) ausgewählt und als AUI-Adapterausgang bereitgestellt werden. Intern werden die eingehenden UINT-Werte in AUI-Adapter konvertiert und über eine Kombination aus Multiplexer-Bausteinen auf einen einzigen Ausgangs-Adapter geschaltet.

Schnittstellenstruktur

Ereignis-Eingänge

- **EI1**: Event zur Auswahl von val1
- **EI2**: Event zur Auswahl von val2
- **EI3**: Event zur Auswahl von val3
- **EI4**: Event zur Auswahl von val4
- **EI5**: Event zur Auswahl von val5
- **EI6**: Event zur Auswahl von val6
- **EI7**: Event zur Auswahl von val7

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

- **val1** (UINT): Initialer Ausgabewert bei EI1
- **val2** (UINT): Initialer Ausgabewert bei EI2
- **val3** (UINT): Initialer Ausgabewert bei EI3
- **val4** (UINT): Initialer Ausgabewert bei EI4
- **val5** (UINT): Initialer Ausgabewert bei EI5
- **val6** (UINT): Initialer Ausgabewert bei EI6
- **val7** (UINT): Initialer Ausgabewert bei EI7

Daten-Ausgänge

- Keine direkten Daten-Ausgänge. Die Ausgabe erfolgt über den Adapter `OUT`.

Adapter

- **OUT** (Plug): Typ `adapter::types::unidirectional::AUI`
  Ausgewählter AUI-Adapter-Output – stellt den aktuell über die Ereignisse selektierten Wert bereit.

Funktionsweise

1. Die sieben UINT-Eingangswerte `val1`..`val7` werden jeweils an einen internen Baustein `initval_AUI_i` übergeben. Diese Bausteine konvertieren die UINT-Werte in AUI-Adapter und geben sie an ihrem Ausgang `OUT` weiter.
2. Die Ereignisse `EI1`..`EI7` werden parallel an den Baustein `AUI_MUX_7` geleitet, der basierend auf dem eingehenden Ereignis ein internes Selektionssignal (Adapter `K`) erzeugt.
3. Der Baustein `AUI_AUI_MUX_7` empfängt alle sieben vorbereiteten AUI-Adapter (`IN1`..`IN7`) sowie das Selektionssignal `K`. Er wählt denjenigen Eingang aus, dessen Index dem aktiven Ereignis entspricht, und leitet ihn an seinen Ausgang `OUT` weiter.
4. Der Ausgang `OUT` von `AUI_AUI_MUX_7` ist mit dem Adapter-Port `OUT` der Subapp verbunden und stellt so den gewünschten AUI-Wert nach außen bereit.

Technische Besonderheiten

- **Interne Struktur**: Die Subapp besteht aus den Bausteinen `AUI_MUX_7`, `AUI_AUI_MUX_7` und sieben `initval_AUI`-Instanzen. Diese modulare Bauweise ermöglicht eine klare Trennung von Ereignissteuerung, Selektion und Wertkonvertierung.
- **Adapterbasierte Ausgabe**: Die Ausgabe erfolgt ausschließlich über den AUI-Adapter `OUT`, wodurch der Baustein nahtlos in AUI-basierte Kommunikationsketten integriert werden kann.
- **Initialwerte**: Die UINT-Werte werden als Initialwerte für die internen AUI-Adapter verwendet. Ein Ereignis setzt den Ausgang unmittelbar auf den zugehörigen Wert.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Endzustand im Sinne einer Zustandsmaschine. Er verhält sich wie ein asynchroner Multiplexer: Der aktuelle Ausgangszustand wird ausschließlich durch das zuletzt empfangene Ereignis bestimmt. Nach Aktivierung von `EIi` bleibt der Ausgang so lange auf dem Wert `vali`, bis ein anderes Ereignis ein Umschalten auslöst.

Anwendungsszenarien

- **Parameterumschaltung**: Auswahl unterschiedlicher Konfigurationswerte (z. B. Sollwerte, Grenzwerte) durch entsprechende Ereignisse.
- **Signalrouting**: Weiterleitung von bis zu sieben AUI-Werten an eine gemeinsame Senke, wobei die Auswahl über externe Events erfolgt.
- **Test- und Diagnoseumgebungen**: Umschaltung zwischen verschiedenen Testsignalen oder simulierten Messwerten.

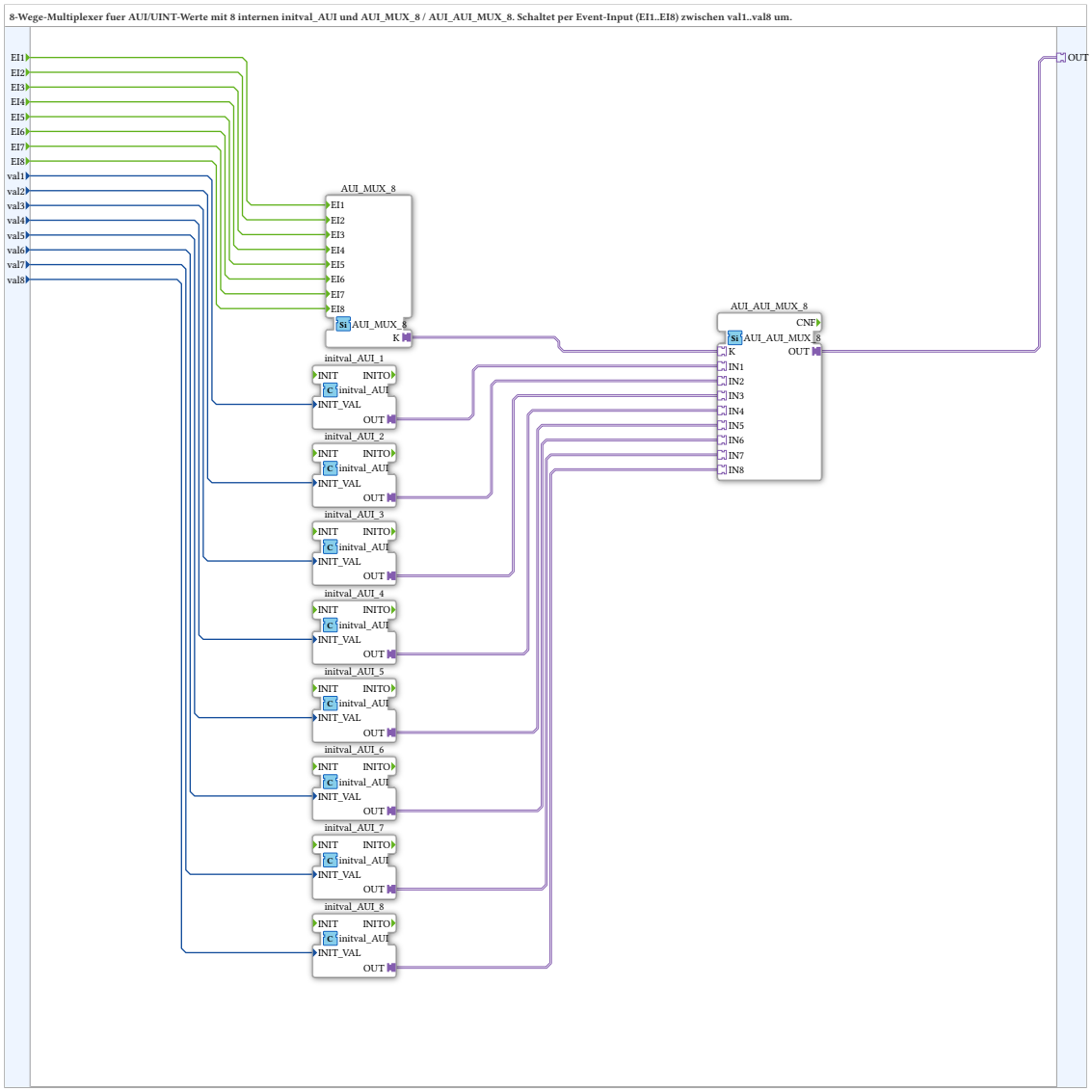
Vergleich mit ähnlichen Bausteinen

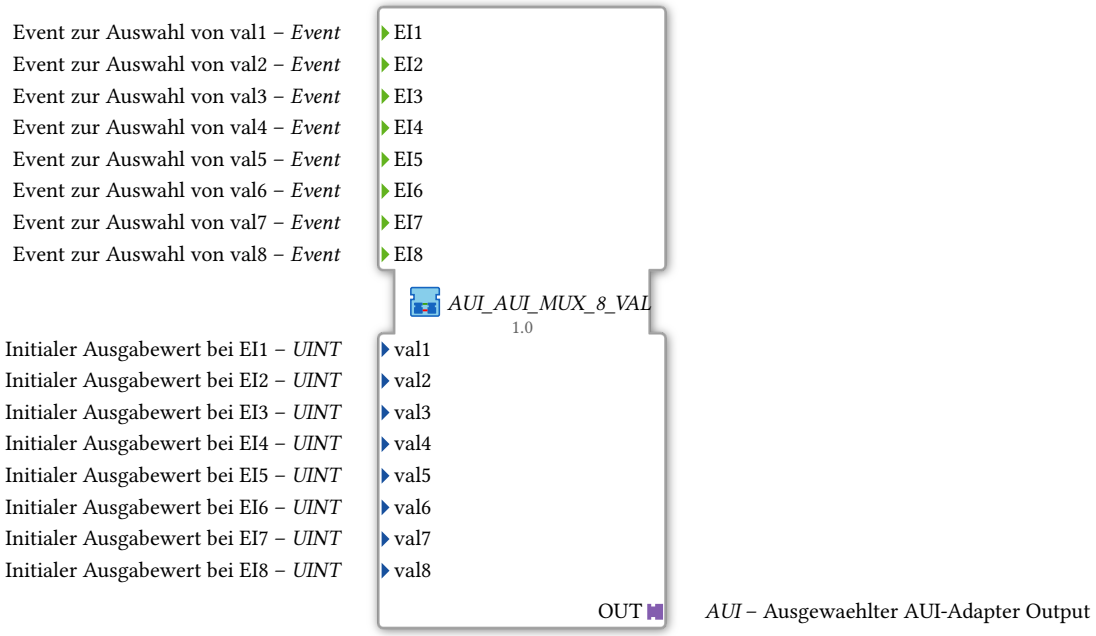
Im Vergleich zu einem einfachen UINT-Multiplexer (z. B. `MUX` mit `UINT`-Datentyp) bietet dieser Baustein den Vorteil, dass die Ausgabe bereits als AUI-Adapter erfolgt. Dadurch entfällt eine nachgelagerte Konvertierung. Zudem sind die Eingangswerte als UINT definiert, was eine einfache Parametrierung aus SPS-Programmen ermöglicht. Ähnliche Bausteine ohne INIT-VAL-Unterstützung erfordern externe Initialisierung oder besitzen einen zusätzlichen Init-Eingang.

Fazit

Der Baustein **AUI_AUI_MUX_7_VAL** ist eine kompakte und flexible Lösung zur Auswahl eines von sieben UINT-Werten mit AUI-Adapterausgang. Durch die Trennung von Ereignislogik und Adapterselektion bleibt die Funktionsweise klar und ist gut in industrielle Automatisierungskonzepte integrierbar. Die Bereitstellung von Initialwerten erleichtert die Inbetriebnahme und reduziert den externen Konfigurationsaufwand.

AUI_AUI_MUX_8_VAL





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUI_AUI_MUX_8_VAL** ist ein als Subapplikation realisierter 8-Wege-Multiplexer für AUI-Adapter-Werte. Er wählt über acht Ereignis-Eingänge (EI1…EI8) einen von acht UINT-Eingangswerten (val1…val8) aus und stellt diesen Wert über einen unidirektionalen AUI-Adapter an seinem Ausgang (OUT) bereit. Die Auswahl erfolgt rein ereignisgesteuert; die aktuellen Werte werden über interne Init-Bausteine in AUI-Signale umgewandelt.

Schnittstellenstruktur

Ereignis-Eingänge

- **EI1** (Event): Wählt den Wert von **val1** aus.
- **EI2** (Event): Wählt den Wert von **val2** aus.
- **EI3** (Event): Wählt den Wert von **val3** aus.
- **EI4** (Event): Wählt den Wert von **val4** aus.
- **EI5** (Event): Wählt den Wert von **val5** aus.
- **EI6** (Event): Wählt den Wert von **val6** aus.
- **EI7** (Event): Wählt den Wert von **val7** aus.
- **EI8** (Event): Wählt den Wert von **val8** aus.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

- **val1** (UINT): Initialer Ausgabewert für Aktivierung über EI1.
- **val2** (UINT): Initialer Ausgabewert für Aktivierung über EI2.
- **val3** (UINT): Initialer Ausgabewert für Aktivierung über EI3.
- **val4** (UINT): Initialer Ausgabewert für Aktivierung über EI4.
- **val5** (UINT): Initialer Ausgabewert für Aktivierung über EI5.
- **val6** (UINT): Initialer Ausgabewert für Aktivierung über EI6.
- **val7** (UINT): Initialer Ausgabewert für Aktivierung über EI7.
- **val8** (UINT): Initialer Ausgabewert für Aktivierung über EI8.

Daten-Ausgänge

Keine vorhanden; der Ausgang erfolgt ausschließlich über den Adapter **OUT**.

Adapter

- **OUT** (adapter::types::unidirectional::AUI): Liefert den aktuell ausgewählten Wert als AUI-Signal.

Funktionsweise

Die Subapplikation verknüpft drei interne Funktionsbaustein-Typen:

1. **AUI_MUX_8** – Ein Ereignis-Multiplexer, der anhand der ankommenden Events (EI1 bis EI8) ein Steuersignal (K) für den Adapter-Multiplexer erzeugt.
2. **AUI_AUI_MUX_8** – Ein Adapter-Multiplexer, der den entsprechenden Eingang (IN1…IN8) auf den Ausgang (OUT) durchschaltet.
3. **initval_AUI** – Acht Instanzen dieses Bausteins (initval_AUI_1 … initval_AUI_8) wandeln die übergebenen UINT-Werte (val1…val8) in AUI-Adapter-Signale um und stellen sie als initiale Werte an die Eingänge des Adapter-Multiplexers bereit.

Werden ein oder mehrere Ereignisse an den Eingängen EI1…EI8 ausgelöst, so aktiviert der Ereignis-Multiplexer den Adapter-Multiplexer, der dann den passenden der acht vorbereiteten AUI-Werte an den Ausgang OUT legt. Da nur ein Auswahl-Event gleichzeitig wirksam sein kann (typische Multiplexer-Logik), wird immer nur ein Eingang aktiv geschaltet.

Technische Besonderheiten

- **Unidirektionale AUI-Kommunikation**: Der Ausgang ist ein unidirektionaler AUI-Adapter; es sind keine Rückkanäle vorgesehen.
- **Initialwert-Aufbereitung**: Die UINT-Eingangswerte werden über die internen `initval_AUI`-Bausteine in AUI-kompatible Signale umgewandelt, sodass sie direkt an den Adapter-Multiplexer übergeben werden können.
- **Event-gesteuerte Auswahl**: Die Umschaltung erfolgt rein über Ereignisse – es gibt keinen „Select“-Dateneingang. Jedes Event steht für einen festen Kanal.
- **Keine Ereignis-Ausgänge**: Nach der Auswahl wird kein Bestätigungs-Event ausgegeben; die Auswahl wird ausschließlich über den Adapter-Ausgang sichtbar.

Zustandsübersicht

Der Funktionsblock besitzt keine expliziten endlichen Zustände im klassischen Sinne. Es lassen sich acht stabile Auswahlzustände definieren, die durch das jeweils zuletzt ausgelöste Ereignis bestimmt werden:

- **Zustand 1** (EI1 aktiv): OUT liefert den Wert von val1.
- **Zustand 2** (EI2 aktiv): OUT liefert den Wert von val2.
- …
- **Zustand 8** (EI8 aktiv): OUT liefert den Wert von val8.

Überschneidende Events (z. B. gleichzeitige Aktivierung mehrerer Eingänge) werden vom internen Multiplexer auf eine eindeutige, feste Priorität aufgelöst (üblicherweise der niedrigste Index gewinnt), ohne dass ein expliziter Zustand dafür definiert ist.

Anwendungsszenarien

- **Parameterumschaltung**: Auswahl eines von acht vordefinierten Betriebsparametern zur Laufzeit, z. B. Geschwindigkeitsprofile oder Regelungskonstanten.
- **Profilwahl**: Umschalten zwischen verschiedenen Konfigurationsprofilen in einer Maschinensteuerung.
- **Test- und Simulationsumgebungen**: Bereitstellung unterschiedlicher Messwerte oder Sollwerte über einen AUI-Datenstrom.
- **Redundanzlösungen**: Auswahl eines aktiven Signalpfads aus mehreren redundanten Quellen.

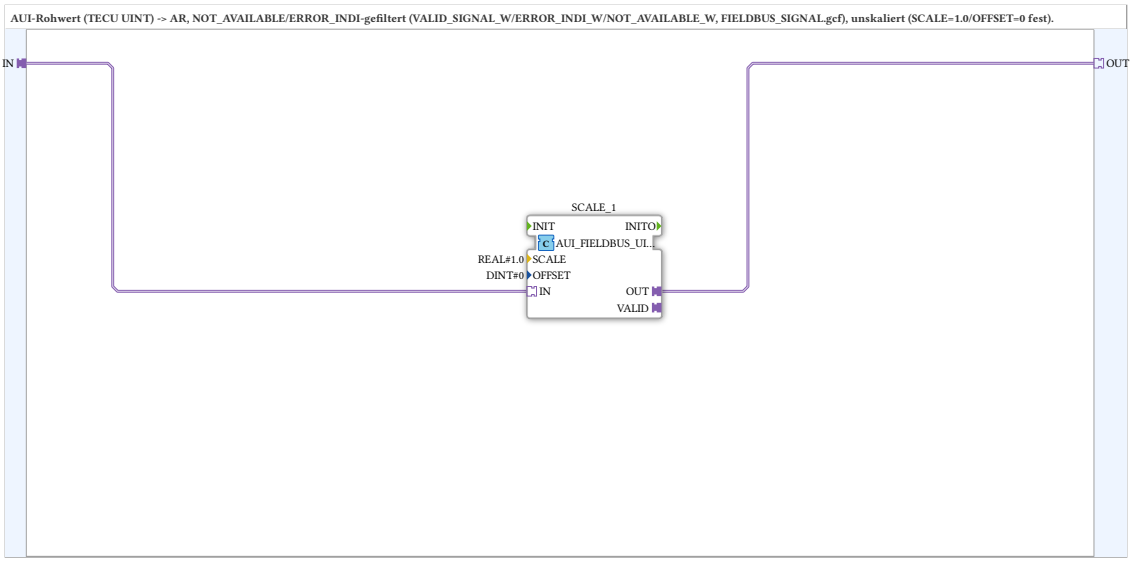
Vergleich mit ähnlichen Bausteinen

Gegenüber einfachen Multiplexern (z. B. ohne `initval_AUI`) bietet dieser Baustein den Vorteil, dass die UINT-Eingangswerte bereits intern in AUI-Adapter-Signale konvertiert werden und direkt mit aui-fähigen Komponenten verbunden werden können. Zudem sind die Auswahl-Events klar einem bestimmten Kanal zugeordnet, was die Verdrahtung übersichtlicher macht. Im Vergleich zu einem generischen Adapter-Multiplexer ohne Ereignis-Steuerung ist dieser Baustein speziell für ereignisbasierte Umschaltungen optimiert.

Fazit

Der **AUI_AUI_MUX_8_VAL** ist ein kompakter, ereignisgesteuerter 8-Kanal-Multiplexer für AUI-Werte. Er kombiniert die Vorteile einer einfachen Datenauswahl mit der direkten Anbindung an unidirektionale AUI-Schnittstellen. Durch die internen Initial-Bausteine entfällt eine separate Konvertierung und die Logik bleibt klar strukturiert. Er eignet sich ideal für Anwendungen, in denen mehrere vordefinierte Werte über Ereignisse schnell und zuverlässig ausgewählt werden müssen.

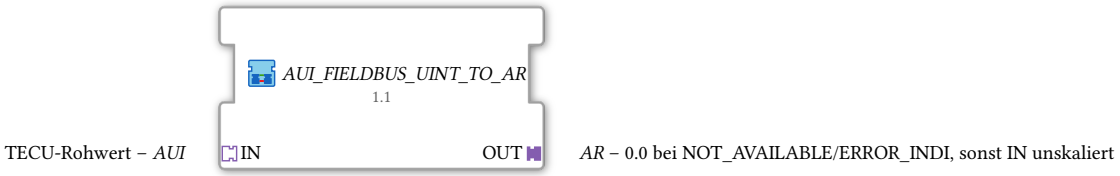
AUI_FIELDBUS_UINT_TO_AR



• – ○ • – ○ • – ○ •

Einleitung

`AUI_FIELDBUS_UINT_TO_AR` wandelt einen `AUI`-Rohwert (TECU, `UINT`) in einen `AR`-Wert um und filtert dabei die Feldbus-Sonderwerte `NOT_AVAILABLE` und `ERROR_INDI` heraus. Die Umrechnung erfolgt unskaliert (`SCALE = 1.0`, `OFFSET = 0` fest). Der Baustein wurde herausgelöst, weil dieselbe Parametrierung zuvor in `TECU_RPTO`, `TECU_FPTO`, `TECU_COGSOG`, `TECU_GBSD`, `TECU_WBSD`, `TECU_VDS`, `TECU_FHS` und `TECU_RHS` jedes Mal einzeln gesetzt wurde. Er liegt in `MyLib_AX::sys`, da er generisch und nicht TECU-spezifisch ist.



Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

- **IN** (Socket, `adapter::types::unidirectional::AUI`): TECU-Rohwert

- **OUT** (Plug, `adapter::types::unidirectional::AR`): `0.0` bei `NOT_AVAILABLE/ERROR_INDI`, sonst `IN` unskaliert

Verwendete Funktionsbausteine (FBs)

- **`SCALE_1`** (`logiBUS::signalprocessing::fieldbus::AUI_FIELDBUS_UINT_TO_SIGNAL_SCALED`): Parameter `SCALE = 1.0`, `OFFSET = 0`

Funktionsweise

`IN` ⟶ `SCALE_1.IN` und `SCALE_1.OUT` ⟶ `OUT`. Die Filterung der Feldbus-Sonderwerte (`VALID_SIGNAL_W`/`ERROR_INDI_W`/`NOT_AVAILABLE_W`, definiert in `FIELDBUS_SIGNAL.gcf`) übernimmt der interne Baustein; die SubApp legt nur Skalierung und Offset fest.

Technische Besonderheiten

- Keine eigenen Ereignisse oder Daten-Ports, nur Adapter
- Fester Skalierungsfaktor 1.0 und Offset 0 (generische, unskalierte Variante)

Anwendungsszenarien

- Übernahme von TECU-Rohwerten (z. B. Drehzahlen) aus dem Feldbus als `AR`-Signal ohne Skalierung

Zusammenfassung

`AUI_FIELDBUS_UINT_TO_AR` kapselt `AUI_FIELDBUS_UINT_TO_SIGNAL_SCALED` mit `SCALE = 1.0/OFFSET = 0` als wiederverwendbare SubApp.

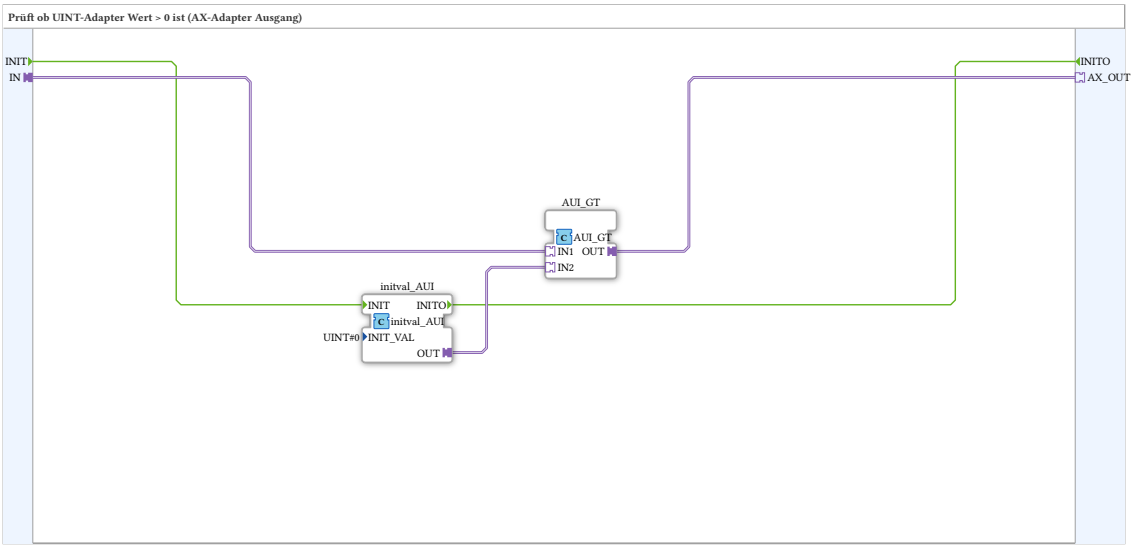
🛠️ Zugehörige Übungen

- Uebung_078_AUI

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

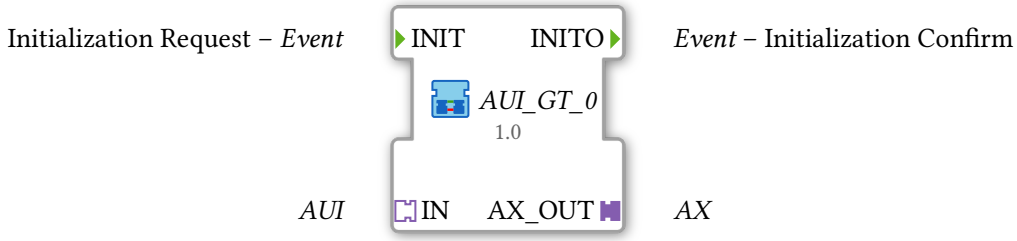
AUI_GT_0



• – ○ • – ○ • – ○ •

Einleitung

`AUI_GT_0` prüft, ob ein UINT-Adapter-Wert größer als 0 ist, und gibt das Ergebnis als boolesches AX-Adapter-Signal aus. Typischer Eingangswert ist ein Objekt-ID- oder Zählerwert (z. B. „wie viele Kanäle sind aktiv“). Der Baustein ist für jeden Zahlentyp gleich aufgebaut (`AS`, `AI`, `ADI`, `ALI`, `AUS`, `AUI`, `AUDI`, `AULI`, `AR`, `ALR`).



Schnittstelle

- **Ereignis-Eingang**: `INIT` (Initialisierung)
- **Ereignis-Ausgang**: `INITO` (Initialisierung abgeschlossen, kommt vom internen `initval_AUI`)
- **Sockel**: `IN` (`AUI`) — der Wert, der gegen 0 verglichen wird
- **Stecker**: `AX_OUT` (`AX`) — TRUE, wenn `IN > 0`

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: AUI_GT_0

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **initval_AUI**: `adapter::types::unidirectional::AUI::initval::initval_AUI` — liefert die feste Vergleichskonstante `UINT#0` als Adapter-Wert (`INIT_VAL=UINT#0`). Ein Adapter-Sockel kann die Konstante nicht als Parameter bekommen, deshalb wird sie über diesen Baustein eingespeist.
  - **AUI_GT**: `adapter::iec61131::comparison::AUI_GT` — adapter-nativer Vergleich `IN1 > IN2`.
- **Funktionsweise**: `IN` ⟶ `AUI_GT.IN1`; Konstante 0 ⟶ `AUI_GT.IN2`; Ergebnis ⟶ `AX_OUT`.

Programmablauf und Verbindungen

1. `INIT` ⟶ `initval_AUI.INIT`; `initval_AUI.INITO` ⟶ `INITO`.
2. `IN` (Adapter) ⟶ `AUI_GT.IN1`.
3. `initval_AUI.OUT` (Konstante `UINT#0`) ⟶ `AUI_GT.IN2`.
4. `AUI_GT.OUT` ⟶ `AX_OUT` (Adapter).

Anwendungsszenarien

- Umwandlung von ein Objekt-ID- oder Zählerwert (z. B. „wie viele Kanäle sind aktiv“) in ein einfaches boolesches Adapter-Signal für VT-Statusanzeigen oder Freigabelogik.

Zusammenfassung

`AUI_GT_0` ist ein kompakter Wrapper um den adapter-nativen Vergleich `AUI_GT` — wandelt „UINT > 0“ direkt in ein AX-Adapter-Signal um, ohne dass der Aufrufer die Null-Konstante selbst anlegen und verdrahten muss.

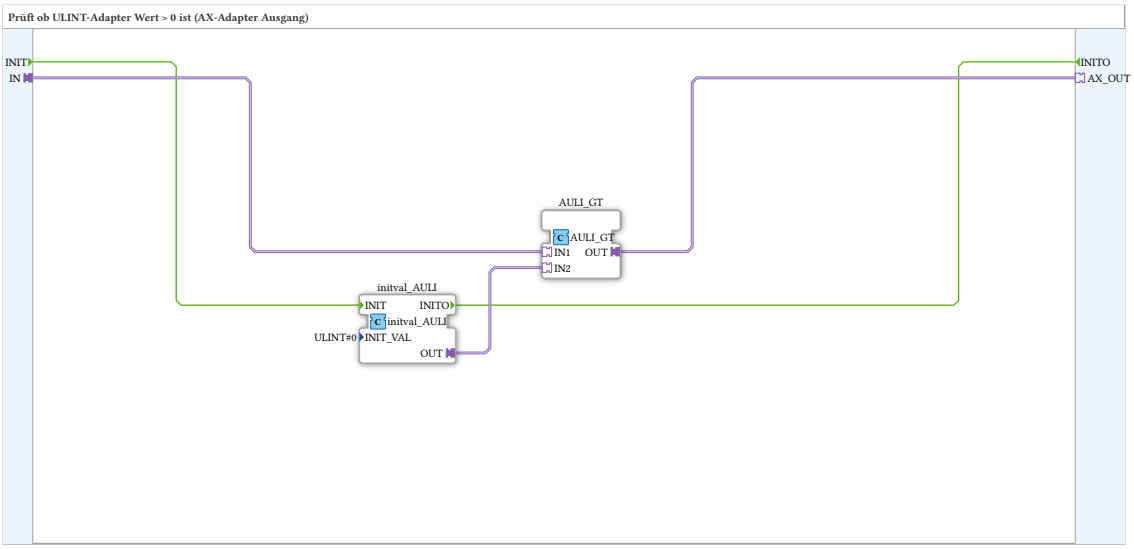
🛠️ Zugehörige Übungen

- Uebung_071_AUI
- Uebung_071a_AUI

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

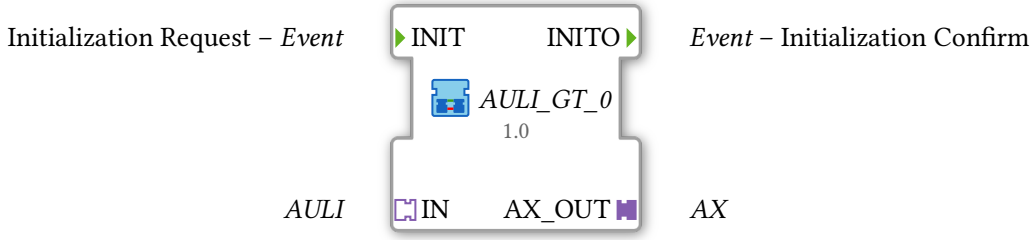
AULI_GT_0



• – ○ • – ○ • – ○ •

Einleitung

`AULI_GT_0` prüft, ob ein ULINT-Adapter-Wert größer als 0 ist, und gibt das Ergebnis als boolesches AX-Adapter-Signal aus. Typischer Eingangswert ist ein vorzeichenloser 64-Bit-Zählerwert. Der Baustein ist für jeden Zahlentyp gleich aufgebaut (`AS`, `AI`, `ADI`, `ALI`, `AUS`, `AUI`, `AUDI`, `AULI`, `AR`, `ALR`).



Schnittstelle

- **Ereignis-Eingang**: `INIT` (Initialisierung)
- **Ereignis-Ausgang**: `INITO` (Initialisierung abgeschlossen, kommt vom internen `initval_AULI`)
- **Sockel**: `IN` (`AULI`) — der Wert, der gegen 0 verglichen wird
- **Stecker**: `AX_OUT` (`AX`) — TRUE, wenn `IN > 0`

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: AULI_GT_0

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **initval_AULI**: `adapter::types::unidirectional::AULI::initval::initval_AULI` — liefert die feste Vergleichskonstante `ULINT#0` als Adapter-Wert (`INIT_VAL=ULINT#0`). Ein Adapter-Sockel kann die Konstante nicht als Parameter bekommen, deshalb wird sie über diesen Baustein eingespeist.
  - **AULI_GT**: `adapter::iec61131::comparison::AULI_GT` — adapter-nativer Vergleich `IN1 > IN2`.
- **Funktionsweise**: `IN` ⟶ `AULI_GT.IN1`; Konstante 0 ⟶ `AULI_GT.IN2`; Ergebnis ⟶ `AX_OUT`.

Programmablauf und Verbindungen

1. `INIT` ⟶ `initval_AULI.INIT`; `initval_AULI.INITO` ⟶ `INITO`.
2. `IN` (Adapter) ⟶ `AULI_GT.IN1`.
3. `initval_AULI.OUT` (Konstante `ULINT#0`) ⟶ `AULI_GT.IN2`.
4. `AULI_GT.OUT` ⟶ `AX_OUT` (Adapter).

Anwendungsszenarien

- Umwandlung von ein vorzeichenloser 64-Bit-Zählerwert in ein einfaches boolesches Adapter-Signal für VT-Statusanzeigen oder Freigabelogik.

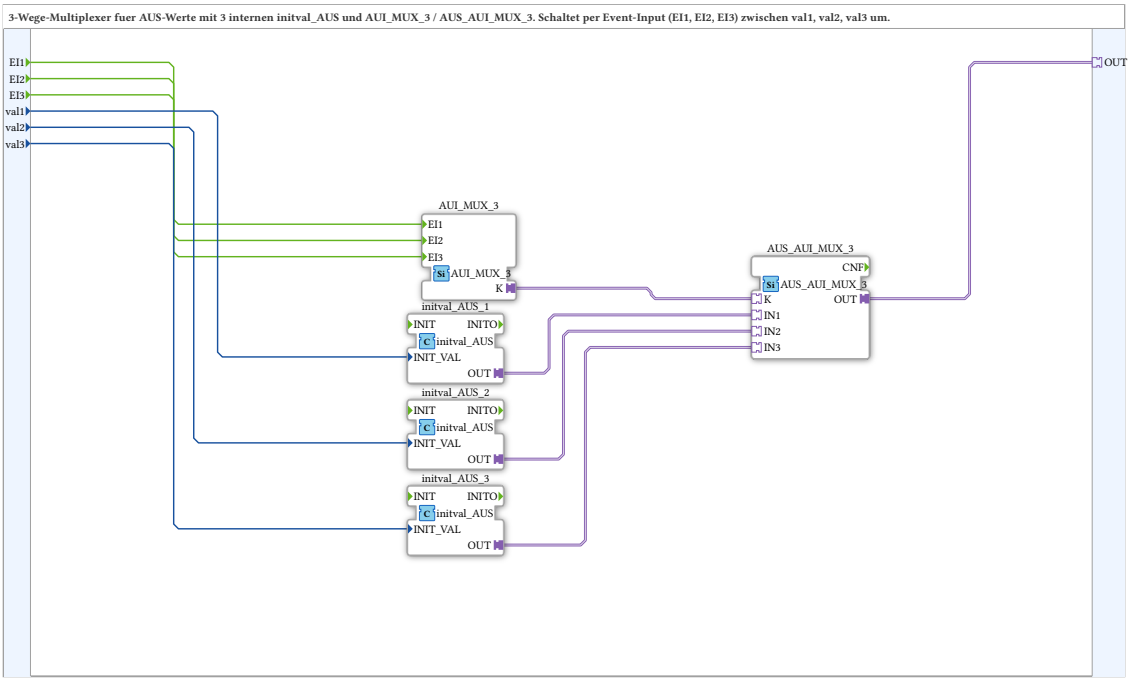
Zusammenfassung

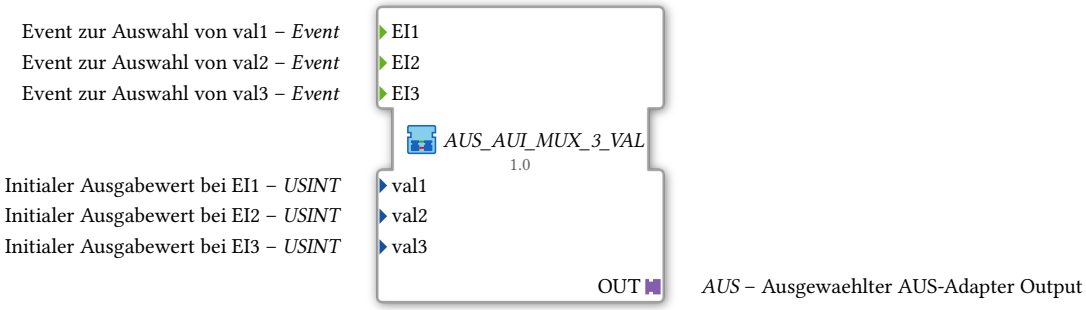
`AULI_GT_0` ist ein kompakter Wrapper um den adapter-nativen Vergleich `AULI_GT` — wandelt „ULINT > 0“ direkt in ein AX-Adapter-Signal um, ohne dass der Aufrufer die Null-Konstante selbst anlegen und verdrahten muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUS_AUI_MUX_3_VAL





• – ○ • – ○ • – ○ •

Einleitung

Der **AUS_AUI_MUX_3_VAL** ist ein Funktionsblock (SubApp), der als 3-Wege-Multiplexer für AUS-Werte dient. Er ermöglicht es, über drei verschiedene Ereignis-Eingänge einen von drei konfigurierbaren Ausgabewerten auszuwählen. Die Auswahl wird über einen AUS-Adapter am Ausgang `OUT` zur Verfügung gestellt. Intern werden die Werte über eigene `initval_AUS`-Bausteine initialisiert und über einen Adapter-Multiplexer (`AUS_AUI_MUX_3`) weitergeleitet. Die Ereignissteuerung erfolgt über den Baustein `AUI_MUX_3`.

Schnittstellenstruktur

Ereignis-Eingänge

- **EI1** (Event): Event zur Auswahl von `val1`
- **EI2** (Event): Event zur Auswahl von `val2`
- **EI3** (Event): Event zur Auswahl von `val3`

Ereignis-Ausgänge

Keine.

Daten-Eingänge

- **val1** (USINT): Initialer Ausgabewert bei EI1
- **val2** (USINT): Initialer Ausgabewert bei EI2
- **val3** (USINT): Initialer Ausgabewert bei EI3

Daten-Ausgänge

Keine direkten Datenausgänge. Der ausgewählte Wert wird über den Adapter `OUT` bereitgestellt.

Adapter

- **OUT** (AUS, unidirektional): Ausgewählter AUS-Adapter-Output.

Funktionsweise

Die SubApp besitzt drei Daten-Eingänge (`val1`, `val2`, `val3`) und drei Ereignis-Eingänge (`EI1`, `EI2`, `EI3`). Bei einem Ereignis an einem der drei Eingänge wird der entsprechende Wert über einen internen `initval_AUS`-Baustein auf einen AUS-Adapter gesetzt. Der Baustein `AUI_MUX_3` leitet das Ereignis an den Adapter-Multiplexer `AUS_AUI_MUX_3` weiter, der dann den zugehörigen Eingang (`IN1`, `IN2` oder `IN3`) auswählt und dessen Wert auf den Ausgang `OUT` schaltet. Somit wird der an dem jeweiligen Ereignis-Eingang eingestellte Datenwert als AUS-Wert am Ausgang ausgegeben.

Technische Besonderheiten

- Der Baustein ist als SubApp in der 4diac-IDE realisiert und nutzt interne Bausteine (`initval_AUS`, `AUI_MUX_3`, `AUS_AUI_MUX_3`).
- Die Datenwerte sind als `USINT` definiert, werden aber über den AUS-Adapter als strukturierte Werte übertragen.
- Die Lizenzierung erfolgt unter Eclipse Public License 2.0 (EPL-2.0).

Zustandsübersicht

Der Baustein weist keine expliziten Zustände auf. Er reagiert ereignisgesteuert: Jedes Ereignis wählt direkt den entsprechenden Wert aus. Die Auswahl wird sofort am Ausgang `OUT` aktiv.

Anwendungsszenarien

- Umschaltung zwischen verschiedenen Betriebsmodi, die als AUS-Werte abgebildet sind.
- Auswahl von unterschiedlichen Konfigurationswerten in Automatisierungssystemen über diskrete Ereignisse.
- Einfache Ereignis-gesteuerte Wertauswahl in verteilten Steuerungen.

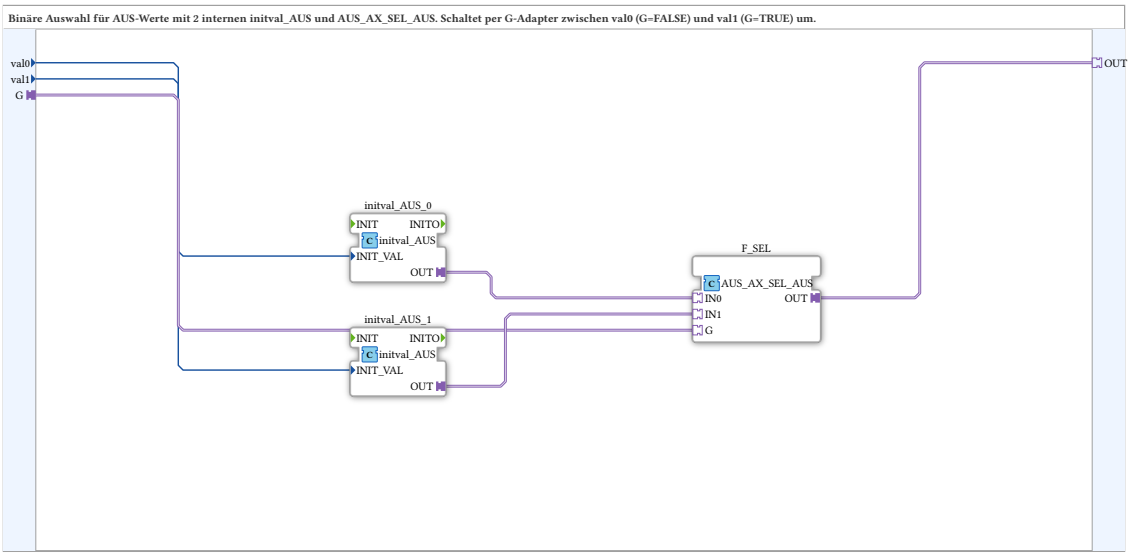
Vergleich mit ähnlichen Bausteinen

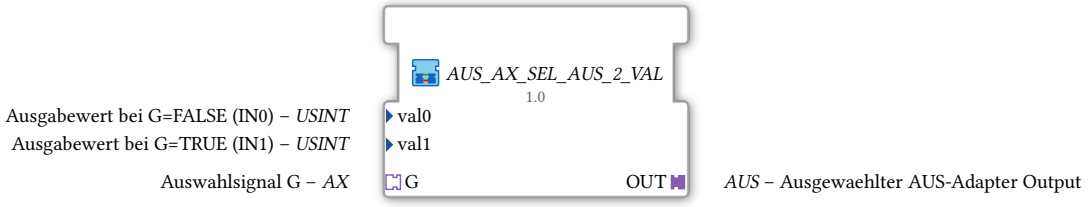
Im Gegensatz zu einem klassischen Multiplexer (z. B. `MUX`), der über ein Daten-Signal auswählt, erfolgt hier die Auswahl ausschließlich über Ereignis-Eingänge. Dadurch ist die Umschaltung ereignisbasiert und direkt an bestimmte Ereignisse gekoppelt, was für zeitkritische Steuerungen vorteilhaft sein kann. Zudem arbeitet der Baustein mit AUS-Adaptern, die in der 4diac-Umgebung standardisiert sind.

Fazit

Der **AUS_AUI_MUX_3_VAL** bietet eine kompakte und klare Lösung, um zwischen drei AUS-Werten umzuschalten. Durch die Kombination aus Ereigniseingängen und konfigurierbaren Datenwerten eignet er sich hervorragend für modulare und ereignisbasierte Automatisierungslösungen. Der Aufbau als SubApp erleichtert die Wiederverwendung und Integration in größere Systeme.

AUS_AX_SEL_AUS_2_VAL





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUS_AX_SEL_AUS_2_VAL** ist eine Subapplikation (SubAppType) innerhalb der Bibliothek `MyLib::sys`. Er realisiert eine binäre Auswahl zwischen zwei USINT-Werten (`val0` und `val1`) und stellt das Ergebnis über einen AUS-Adapter (`OUT`) bereit. Die Auswahl wird über einen AX-Adapter (`G`) getriggert: Ist G FALSE, wird `val0` übergeben; ist G TRUE, wird `val1` übergeben. Die Subapp dient als wiederverwendbare Komponente für Umschaltlogiken in IEC 61499-basierten Systemen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine – die Subapp besitzt keine ereignisgesteuerten Eingänge.

Ereignis-Ausgänge

Keine – die Subapp besitzt keine ereignisgesteuerten Ausgänge.

Daten-Eingänge

- **val0** (USINT): Ausgabewert, der bei G=FALSE (IN0) verwendet wird.
- **val1** (USINT): Ausgabewert, der bei G=TRUE (IN1) verwendet wird.

Daten-Ausgänge

Keine direkten Datenausgänge; das Ergebnis wird ausschließlich über den Adapter-Ausgang `OUT` bereitgestellt.

Adapter

- **G** (Socket, Typ `adapter::types::unidirectional::AX`): Binäres Auswahlsignal. G=FALSE wählt `val0`, G=TRUE wählt `val1`.
- **OUT** (Plug, Typ `adapter::types::unidirectional::AUS`): Liefert den ausgewählten Wert als AUS-Adapterausgang.

Funktionsweise

Die Subapp nutzt zwei Instanzen des Adapters `initval_AUS` (ein Adapter vom Typ `unidirectional::AUS` mit Initialwert-Eingang), um die eingehenden USINT-Werte in AUS-Adapterwerte umzuwandeln. Diese Werte werden an einen internen Funktionsblock `F_SEL` vom Typ `adapter::iec61131::selection::AUS_AX_SEL_AUS` übergeben. Der FB `F_SEL` besitzt zwei AUS-Eingänge (`IN0`, `IN1`) und einen AX-Eingang (`G`). Gemäß dem Signal an `G` wählt er den entsprechenden Eingang aus und gibt das Ergebnis über seinen AUS-Ausgang (`OUT`) an das `OUT`-Plug der übergeordneten Subapp weiter.

Im Einzelnen:

1. `val0` und `val1` werden über die Datenverbindungen den INIT_VAL-Eingängen der jeweiligen `initval_AUS`-Adapter zugeführt.
2. Diese Adapter erzeugen stabile AUS-Werte an ihren Ausgängen.
3. Die Ausgänge werden auf `F_SEL.IN0` bzw. `F_SEL.IN1` geschaltet.
4. Das Signal `G` wird direkt an `F_SEL.G` angeschlossen.
5. Der interne FB wählt anhand von `G` den passenden Eingang aus und leitet ihn an `F_SEL.OUT` weiter.
6. Dieser Wert wird schließlich auf den Ausgangs-Adapter `OUT` der Subapp gespiegelt.

Technische Besonderheiten

- **Adapterbasierte Schnittstellen**: Anstelle von einfachen Datentypen werden spezialisierte Adapter (`AX`, `AUS`) verwendet, die eine saubere Trennung von Steuer- und Datenfluss ermöglichen und an der Systemgrenze konfigurierbare Initialwerte bieten.
- **Keine Ereignisverarbeitung**: Die Subapp ist rein daten- und adaptergetrieben – sie benötigt keine Ereignisse, was die Einbindung in datenflussorientierte Netzwerke erleichtert.
- **Interne Konvertierung**: Die Umwandlung von `USINT` zu `AUS` erfolgt durch `initval_AUS`-Adapter, wodurch der eigentliche Auswahlbaustein `AUS_AX_SEL_AUS` unverändert wiederverwendet werden kann.
- **Lizenz** : Die Implementation ist unter der Eclipse Public License 2.0 veröffentlicht (Copyright 2026 Meisterschulen am Ostbahnhof).

Zustandsübersicht

Die Subapp besitzt keinen internen Zustand. Sie arbeitet kombinatorisch: Die Ausgabe erfolgt unmittelbar und direkt in Abhängigkeit von den aktuellen Eingangswerten (`G`, `val0`, `val1`). Es gibt keine sequenziellen Abläufe oder Zustandsübergänge.

Anwendungsszenarien

- **Parametervorwahl**: Umschaltung zwischen zwei Konfigurationswerten (z.B. Betriebsmodus, Sollwerte) über ein Bool-Signal.
- **Redundanzumschaltung**: Auswahl zwischen zwei alternativen Quellen (z.B. Sensoren) mit einem binären Steuersignal.
- **Test- und Diagnosefunktionen**: Umschalten zwischen Normalwert und Testwert in industriellen Steuerungen.

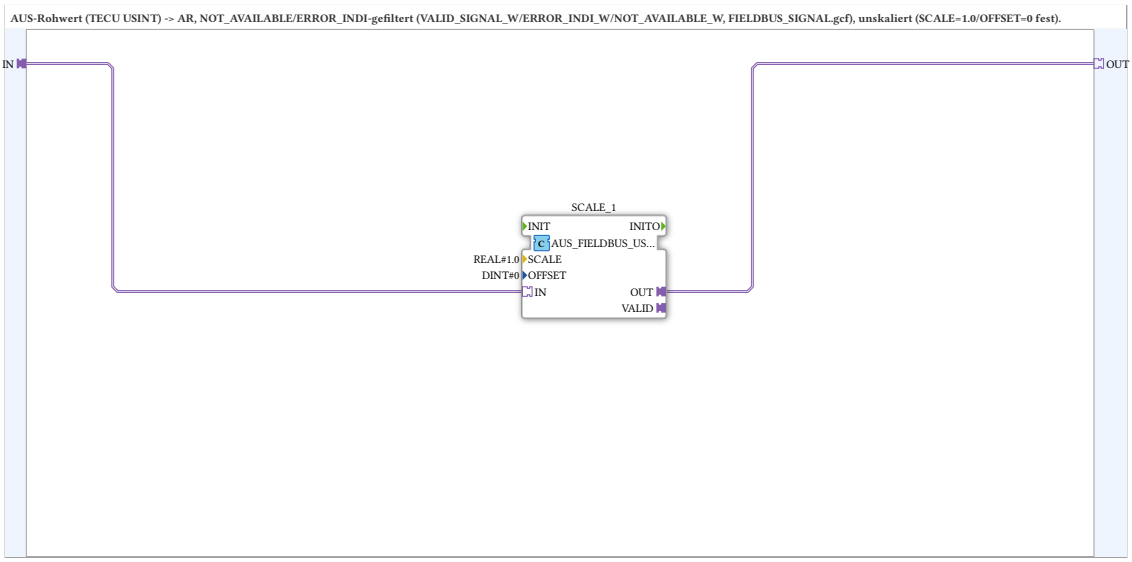
Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem klassischen IEC 61131-3 `SEL`-Funktionsblock bietet diese Subapp eine adapterbasierte Schnittstelle (`AUS`/`AX`), die für verteilte IEC 61499-Systeme optimiert ist. Zudem ist die eigentliche Auswahlkomponente (`AUS_AX_SEL_AUS`) unabhängig von den verwendeten Datentypen und kann durch andere Adaptertypen erweitert werden. Gegenüber einer Verwendung von einfachen `USINT`-Eingängen und -Ausgängen wird hier eine höhere Abstraktionsebene erreicht, die die Kompatibilität mit komplexen Adapterbibliotheken sicherstellt.

Fazit

Der FB **AUS_AX_SEL_AUS_2_VAL** ist eine kompakte, ereignislose Subapplikation zur binären Auswahl von Werten, die sich nahtlos in adapterbasierte IEC 61499-Netzwerke integrieren lässt. Durch die Verwendung von `initval_AUS` und `AUS_AX_SEL_AUS` wird eine robuste und flexible Lösung geschaffen, die für vielfältige Steuerungsaufgaben eingesetzt werden kann. Die einfache Konfiguration über nur zwei Daten-Eingänge und ein Binärsignal macht sie besonders gut für parametrierbare Systeme geeignet.

AUS_FIELDBUS_USINT_TO_AR

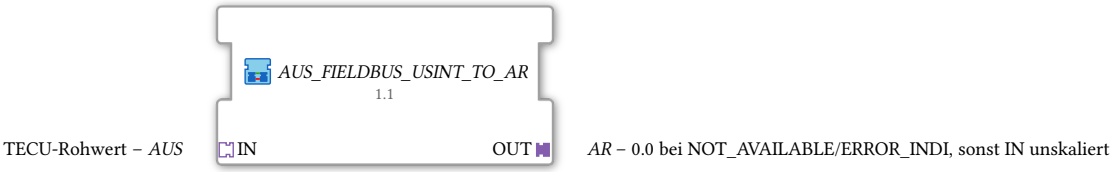


• – ○ • – ○ • – ○ •

Einleitung

`AUS_FIELDBUS_USINT_TO_AR` wandelt einen **AUS-Rohwert** (TECU-`USINT`) in einen `AR`-Wert um. Die Werte `NOT_AVAILABLE/ERROR_INDI` werden dabei gefiltert (`VALID_SIGNAL_W/ERROR_INDI_W/NOT_AVAILABLE_W`, `FIELDBUS_SIGNAL.gcf`); die Skalierung ist fest auf `SCALE=1.0` und `OFFSET=0` eingestellt, der Wert bleibt also unskaliert.

Die SubApp wurde aus `AUS_FIELDBUS_USINT_TO_SIGNAL_SCALED` herausgelöst, da in `TECU_FHS`/`TECU_RHS` `SCALE=1.0`/`OFFSET=0` bisher jedes Mal einzeln parametriert wurde; sie liegt in `MyLib_AX::sys`, weil sie generisch und nicht TECU-spezifisch ist.



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden – Ereignisse kommen über die Adapter-Sockets/-Plugs

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

- **IN** (Socket): Unidirektionaler Adapter vom Typ `adapter::types::unidirectional::AUS` – TECU-Rohwert
- **OUT** (Plug): Unidirektionaler Adapter vom Typ `adapter::types::unidirectional::AR` – `0.0` bei `NOT_AVAILABLE/ERROR_INDI`, sonst `IN` unskaliert

Verwendete Funktionsbausteine (FBs)

- **SCALE_1**: `logiBUS::signalprocessing::fieldbus::AUS_FIELDBUS_USINT_TO_SIGNAL_SCALED`

– Parameter: `SCALE=1.0`, `OFFSET=0`

Programmablauf und Verbindungen

- `IN` ⟶ `SCALE_1.IN`
- `SCALE_1.OUT` ⟶ `OUT`

Der `VALID`-Plug des inneren Bausteins wird nicht nach außen geführt.

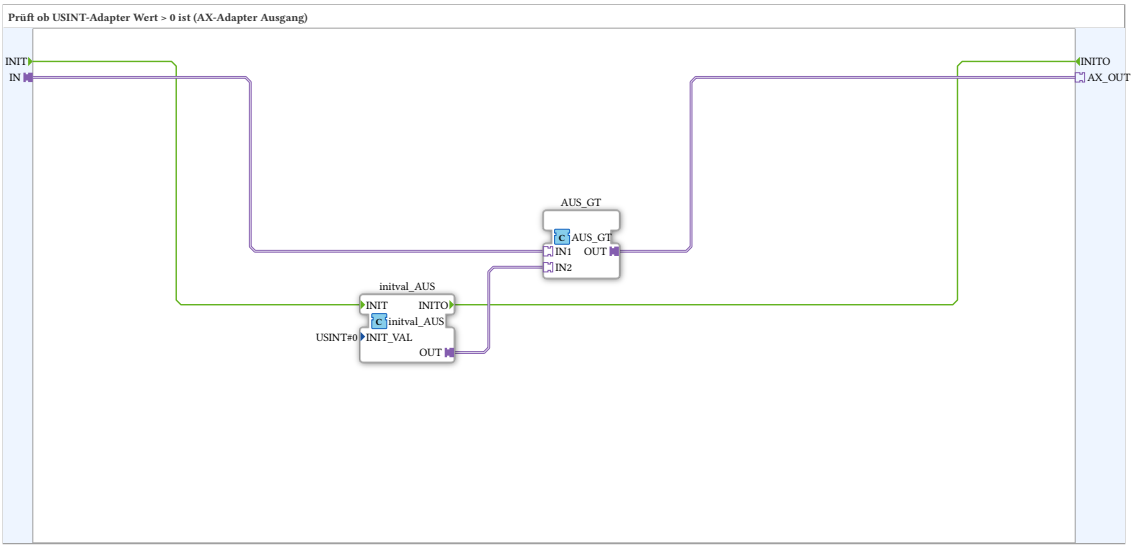
Anwendungsszenarien

- Übernahme von TECU-Werten (`USINT`) als gefilterter, unskalierter `AR`-Wert

Fazit

`AUS_FIELDBUS_USINT_TO_AR` ist eine kleine Wrapper-SubApp, die `AUS_FIELDBUS_USINT_TO_SIGNAL_SCALED` mit festen Parametern `SCALE=1.0`/`OFFSET=0` bereitstellt.

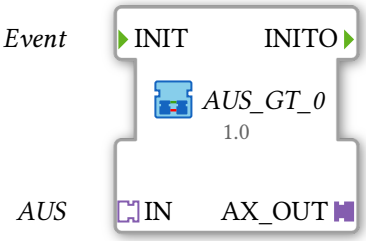
AUS_GT_0



• – ◦ • – ◦ • – ◦ •

Einleitung

`AUS_GT_0` prüft, ob ein USINT-Adapter-Wert größer als 0 ist, und gibt das Ergebnis als boolesches AX-Adapter-Signal aus. Typischer Eingangswert ist ein vorzeichenloser 8-Bit-Wert. Der Baustein ist für jeden Zahlentyp gleich aufgebaut (`AS`, `AI`, `ADI`, `ALI`, `AUS`, `AUI`, `AUDI`, `AULI`, `AR`, `ALR`).



Schnittstelle

- **Ereignis-Eingang**: `INIT` (Initialisierung)
- **Ereignis-Ausgang**: `INITO` (Initialisierung abgeschlossen, kommt vom internen `initval_AUS`)
- **Sockel**: `IN` (`AUS`) — der Wert, der gegen 0 verglichen wird
- **Stecker**: `AX_OUT` (`AX`) — TRUE, wenn `IN > 0`

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: AUS_GT_0

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **initval_AUS**: `adapter::types::unidirectional::AUS::initval::initval_AUS` — liefert die feste Vergleichskonstante `USINT#0` als Adapter-Wert (`INIT_VAL=USINT#0`). Ein Adapter-Sockel kann die Konstante nicht als Parameter bekommen, deshalb wird sie über diesen Baustein eingespeist.
  - **AUS_GT**: `adapter::iec61131::comparison::AUS_GT` — adapter-nativer Vergleich `IN1 > IN2`.
- **Funktionsweise**: `IN` ⟶ `AUS_GT.IN1`; Konstante 0 ⟶ `AUS_GT.IN2`; Ergebnis ⟶ `AX_OUT`.

Programmablauf und Verbindungen

1. `INIT` ⟶ `initval_AUS.INIT`; `initval_AUS.INITO` ⟶ `INITO`.
2. `IN` (Adapter) ⟶ `AUS_GT.IN1`.
3. `initval_AUS.OUT` (Konstante `USINT#0`) ⟶ `AUS_GT.IN2`.
4. `AUS_GT.OUT` ⟶ `AX_OUT` (Adapter).

Anwendungsszenarien

- Umwandlung von ein vorzeichenloser 8-Bit-Wert in ein einfaches boolesches Adapter-Signal für VT-Statusanzeigen oder Freigabelogik.

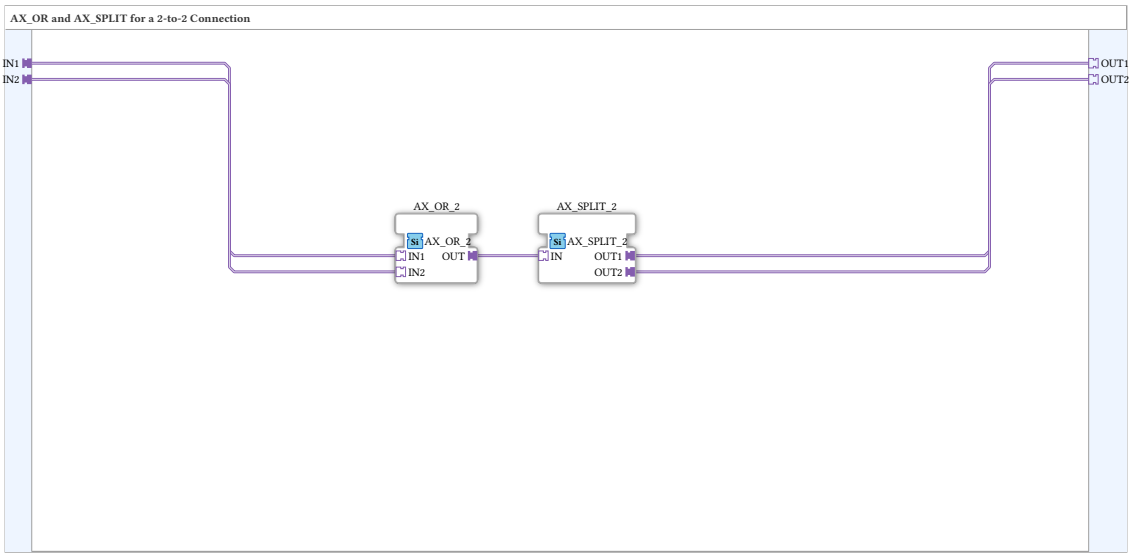
Zusammenfassung

`AUS_GT_0` ist ein kompakter Wrapper um den adapter-nativen Vergleich `AUS_GT` — wandelt „USINT > 0“ direkt in ein AX-Adapter-Signal um, ohne dass der Aufrufer die Null-Konstante selbst anlegen und verdrahten muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

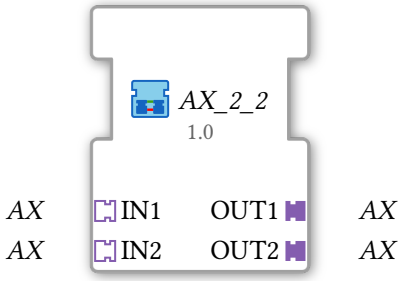
AX_2_2



• – ○ • – ○ • – ○ •

Einleitung

`AX_2_2` verodert zwei AX-Adapter-Eingänge (`IN1`, `IN2`) und verteilt das Ergebnis auf zwei AX-Adapter-Ausgänge (`OUT1`, `OUT2`) — eine „2-zu-2“-Verschaltung für Fälle, in denen zwei unabhängige boolesche Quellen kombiniert und das Ergebnis an zwei unabhängige Ziele weitergegeben werden muss.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: AX_2_2

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **AX_OR_2**: `adapter::booleanOperators::AX_OR_2` — ODER-Verknüpfung zweier Adapter-Eingänge.
  - **AX_SPLIT_2**: `adapter::events::unidirectional::AX_SPLIT_2` — verzweigt das ODER-Ergebnis auf zwei Ausgänge.
- **Funktionsweise**: `IN1 OR IN2` ⟶ `OUT1` und `OUT2` gleichzeitig.

Programmablauf und Verbindungen

1. `IN1` ⟶ `AX_OR_2.IN1`, `IN2` ⟶ `AX_OR_2.IN2`.
2. `AX_OR_2.OUT` ⟶ `AX_SPLIT_2.IN`.
3. `AX_SPLIT_2.OUT1` ⟶ `OUT1`, `AX_SPLIT_2.OUT2` ⟶ `OUT2`.

Anwendungsszenarien

- Jede Stelle, an der zwei unabhängige Bool-Adapter-Quellen ODER-verknüpft und das gemeinsame Ergebnis an zwei unabhängige Verbraucher (z. B. zwei VT-Statusanzeigen) weitergereicht werden muss, ohne eine doppelte Adapterverbindung anzulegen.

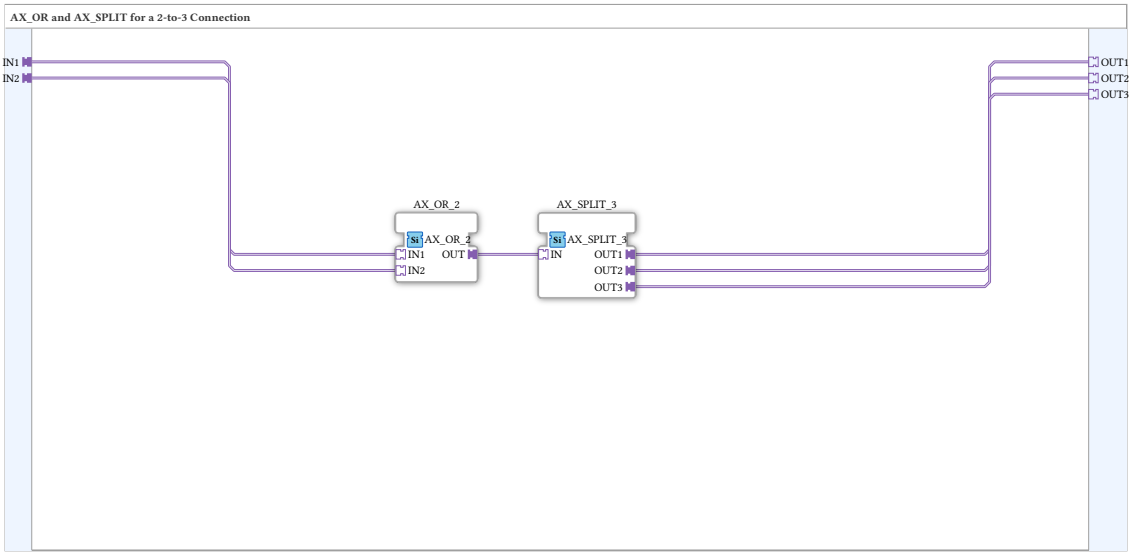
Zusammenfassung

`AX_2_2` ist ein minimaler Baustein für „zwei Quellen ODER-verknüpfen, an zwei Ziele verteilen“ — reine Adapter-Verdrahtung ohne eigene Zustandslogik.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

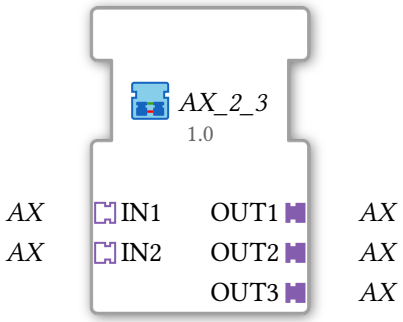
AX_2_3



• – ◦ • – ◦ • – ◦ •

Einleitung

AX_2_3 kombiniert AX_OR_2 und AX_SPLIT_3 zu einem fertigen Baustein: Zwei AX-Quellen (IN1, IN2) werden per ODER zu einem gemeinsamen Signal verknüpft und anschließend auf drei Ziele (OUT1, OUT2, OUT3) verteilt.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: AX_2_3

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **AX_OR_2**: adapter::booleanOperators::AX_OR_2 — verknüpft IN1/IN2 per ODER.
  - **AX_SPLIT_3**: adapter::events::unidirectional::AX_SPLIT_3 — verteilt das ODER-Ergebnis auf drei Ausgänge.
- **Funktionsweise**: IN1/IN2 ⟶ AX_OR_2.OUT ⟶ AX_SPLIT_3.IN ⟶ OUT1/OUT2/OUT3.

Programmablauf und Verbindungen

1. IN1 ⟶ AX_OR_2.IN1; IN2 ⟶ AX_OR_2.IN2.
2. AX_OR_2.OUT ⟶ AX_SPLIT_3.IN.
3. AX_SPLIT_3.OUT1 ⟶ OUT1; AX_SPLIT_3.OUT2 ⟶ OUT2; AX_SPLIT_3.OUT3 ⟶ OUT3.

Anwendungsszenarien

- Zusammenführen zweier AX-Quellen (z. B. zwei Kommandowege), deren gemeinsames Ergebnis an drei unabhängige Ziele (z. B. Hardware, VT-Anzeige, OPC-UA-Publish) verteilt werden muss.

Vergleich mit ähnlichen Bausteinen

Für andere Quellen-/Ziel-Anzahlen: AX_3_2 (3⟶2), AX_3_3 (3⟶3). Nicht zu verwechseln mit AX_2_T0_3: dieser reicht UP/DOWN einzeln durch und liefert das ODER nur als drittes Zusatzsignal, statt generisch ODER+SPLIT zu kombinieren.

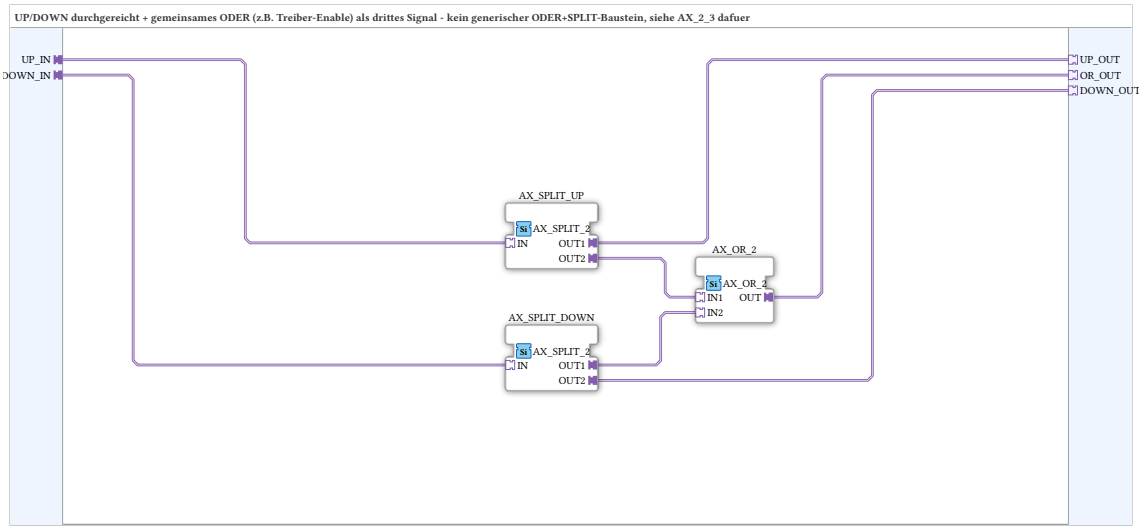
Zusammenfassung

AX_2_3 ist eine reine Verdrahtungshilfe, die zwei AX-Quellen per ODER zusammenführt und auf drei Ziele verteilt — ohne eigene Zustandslogik.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

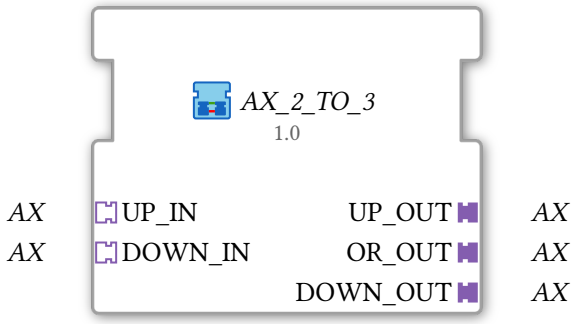
AX_2_TO_3



• – ◦ • – ◦ • – ◦ •

Einleitung

`AX_2_TO_3` nimmt zwei unabhängige AX-Adapter-Signale (`UP_IN`, `DOWN_IN`) entgegen und stellt drei Ausgänge bereit: die beiden Eingänge unverändert durchgereicht (`UP_OUT`, `DOWN_OUT`) sowie zusätzlich deren ODER-Verknüpfung (`OR_OUT`) — typischerweise für „Hoch“/„Runter“-Taster, bei denen zusätzlich ein gemeinsames „irgendeine Richtung aktiv“-Signal gebraucht wird.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: AX_2_TO_3

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **AX_SPLIT_UP** / **AX_SPLIT_DOWN**: je `adapter::events::unidirectional::AX_SPLIT_2` — verzweigen `UP_IN` bzw. `DOWN_IN` je in einen direkten Durchgang und einen Zweig zur ODER-Verknüpfung.
  - **AX_OR_2**: `adapter::booleanOperators::AX_OR_2` — verknüpft die beiden Zweige zu `OR_OUT`.
- **Funktionsweise**: `UP_IN` ⟶ `UP_OUT` (direkt) und in `AX_OR_2`; `DOWN_IN` ⟶ `DOWN_OUT` (direkt) und in `AX_OR_2`; `AX_OR_2.OUT` ⟶ `OR_OUT`.

Programmablauf und Verbindungen

1. `UP_IN` ⟶ `AX_SPLIT_UP.IN` ⟶ `AX_SPLIT_UP.OUT1` ⟶ `UP_OUT`, `AX_SPLIT_UP.OUT2` ⟶ `AX_OR_2.IN1`.
2. `DOWN_IN` ⟶ `AX_SPLIT_DOWN.IN` ⟶ `AX_SPLIT_DOWN.OUT2` ⟶ `DOWN_OUT`, `AX_SPLIT_DOWN.OUT1` ⟶ `AX_OR_2.IN2`.
3. `AX_OR_2.OUT` ⟶ `OR_OUT`.

Anwendungsszenarien

- Rampen-/Ramp-Tasten-Verschaltungen (z. B. „Hoch“/„Runter“), bei denen neben den einzelnen Richtungssignalen auch ein gemeinsames „eine der beiden Tasten ist aktiv“-Signal benötigt wird (z. B. um eine Statusanzeige nur während aktiver Bedienung aufleuchten zu lassen).

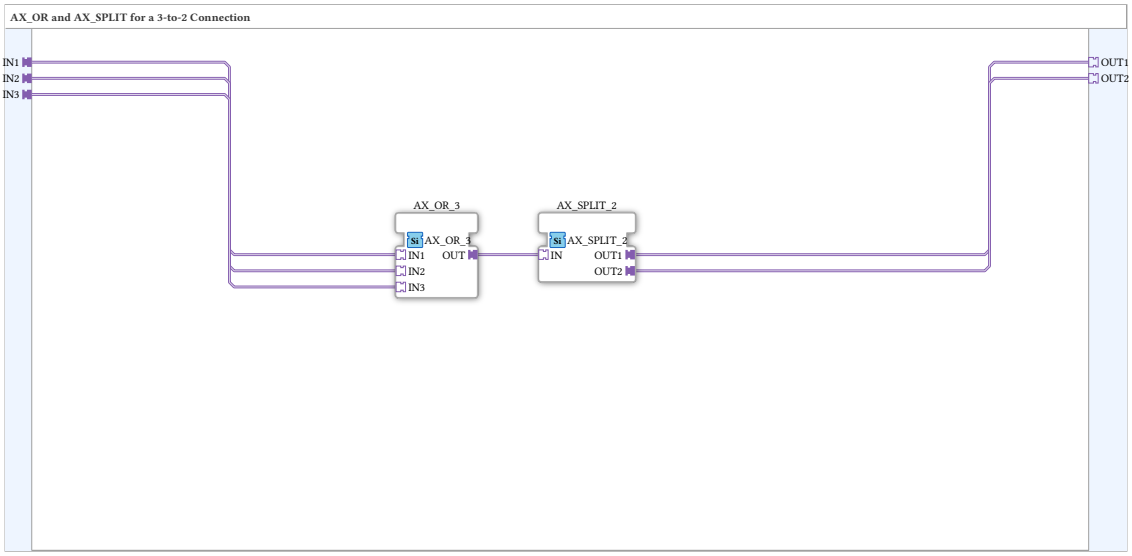
Zusammenfassung

`AX_2_TO_3` reicht zwei Adapter-Signale unverändert durch und ergänzt sie um ihre ODER-Verknüpfung als drittes Signal — reine Verdrahtungshilfe ohne eigene Zustandslogik.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

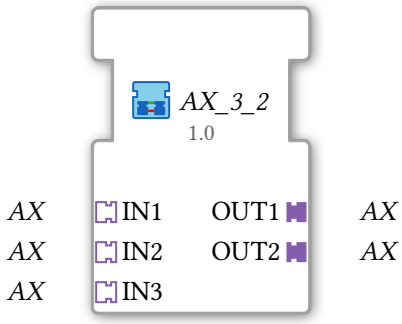
AX_3_2



• – ○ • – ○ • – ○ •

Einleitung

`AX_3_2` kombiniert `AX_OR_3` und `AX_SPLIT_2` zu einem fertigen Baustein: Drei `AX`-Quellen (`IN1`, `IN2`, `IN3`) werden per ODER zu einem gemeinsamen Signal verknüpft und anschließend auf zwei Ziele (`OUT1`, `OUT2`) verteilt.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: AX_3_2

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **AX_OR_3**: `adapter::booleanOperators::AX_OR_3` — verknüpft `IN1`/`IN2`/`IN3` per ODER.
  - **AX_SPLIT_2**: `adapter::events::unidirectional::AX_SPLIT_2` — verteilt das ODER-Ergebnis auf zwei Ausgänge.

- **Funktionsweise**: IN1/IN2/IN3 ⟶ AX_OR_3.OUT ⟶ AX_SPLIT_2.IN ⟶ OUT1/OUT2.

Programmablauf und Verbindungen

1. IN1 ⟶ AX_OR_3.IN1; IN2 ⟶ AX_OR_3.IN2; IN3 ⟶ AX_OR_3.IN3.
2. AX_OR_3.OUT ⟶ AX_SPLIT_2.IN.
3. AX_SPLIT_2.OUT1 ⟶ OUT1; AX_SPLIT_2.OUT2 ⟶ OUT2.

Anwendungsszenarien

- Zusammenführen dreier AX-Quellen (z. B. drei Kommandowege), deren gemeinsames Ergebnis an zwei unabhängige Ziele verteilt werden muss.

Vergleich mit ähnlichen Bausteinen

Für andere Quellen-/Ziel-Anzahlen: AX_2_3 (2⟶3), AX_3_3 (3⟶3).

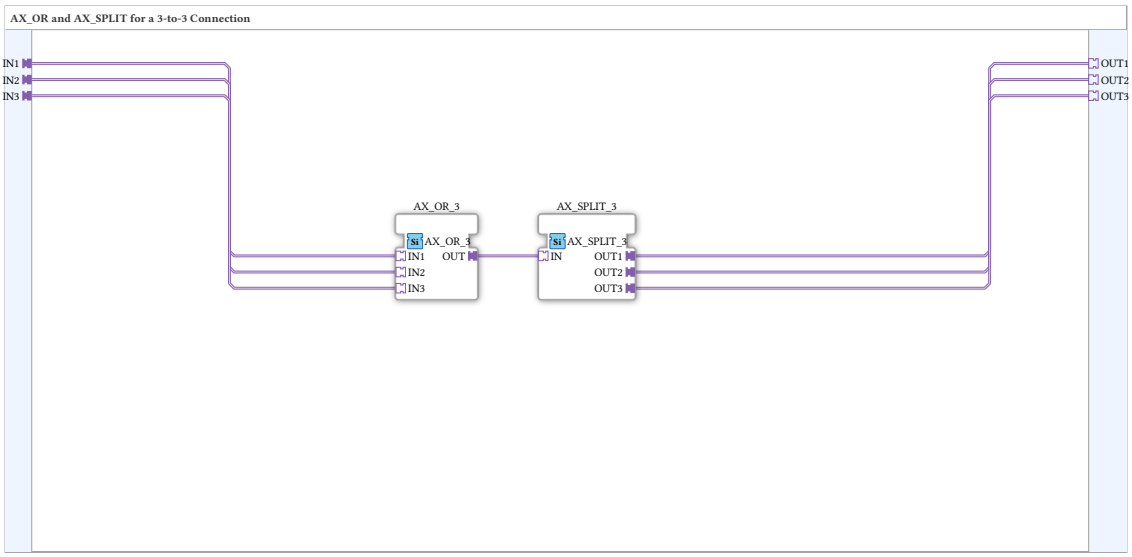
Zusammenfassung

AX_3_2 ist eine reine Verdrahtungshilfe, die drei AX-Quellen per ODER zusammenführt und auf zwei Ziele verteilt — ohne eigene Zustandslogik.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

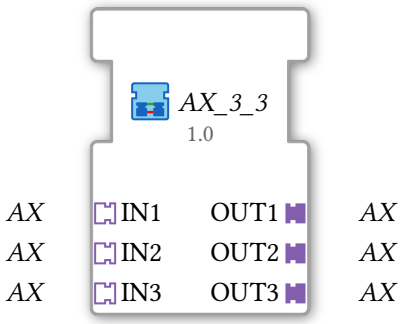
AX_3_3



• – ◦ • – ◦ • – ◦ •

Einleitung

AX_3_3 kombiniert AX_OR_3 und AX_SPLIT_3 zu einem fertigen Baustein: Drei AX-Quellen (IN1, IN2, IN3) werden per ODER zu einem gemeinsamen Signal verknüpft und anschließend auf drei Ziele (OUT1, OUT2, OUT3) verteilt.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: AX_3_3

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **AX_OR_3**: `adapter::booleanOperators::AX_OR_3` — verknüpft `IN1`/`IN2`/`IN3` per ODER.
  - **AX_SPLIT_3**: `adapter::events::unidirectional::AX_SPLIT_3` — verteilt das ODER-Ergebnis auf drei Ausgänge.
- **Funktionsweise**: `IN1`/`IN2`/`IN3` ⟶ `AX_OR_3.OUT` ⟶ `AX_SPLIT_3.IN` ⟶ `OUT1`/`OUT2`/`OUT3`.

Programmablauf und Verbindungen

1. `IN1` ⟶ `AX_OR_3.IN1`; `IN2` ⟶ `AX_OR_3.IN2`; `IN3` ⟶ `AX_OR_3.IN3`.
2. `AX_OR_3.OUT` ⟶ `AX_SPLIT_3.IN`.
3. `AX_SPLIT_3.OUT1` ⟶ `OUT1`; `AX_SPLIT_3.OUT2` ⟶ `OUT2`; `AX_SPLIT_3.OUT3` ⟶ `OUT3`.

Anwendungsszenarien

- Zusammenführen dreier `AX`-Quellen (z. B. drei Kommandowege), deren gemeinsames Ergebnis an drei unabhängige Ziele verteilt werden muss.

Vergleich mit ähnlichen Bausteinen

Für andere Quellen-/Ziel-Anzahlen: `AX_2_3` (2⟶3), `AX_3_2` (3⟶2).

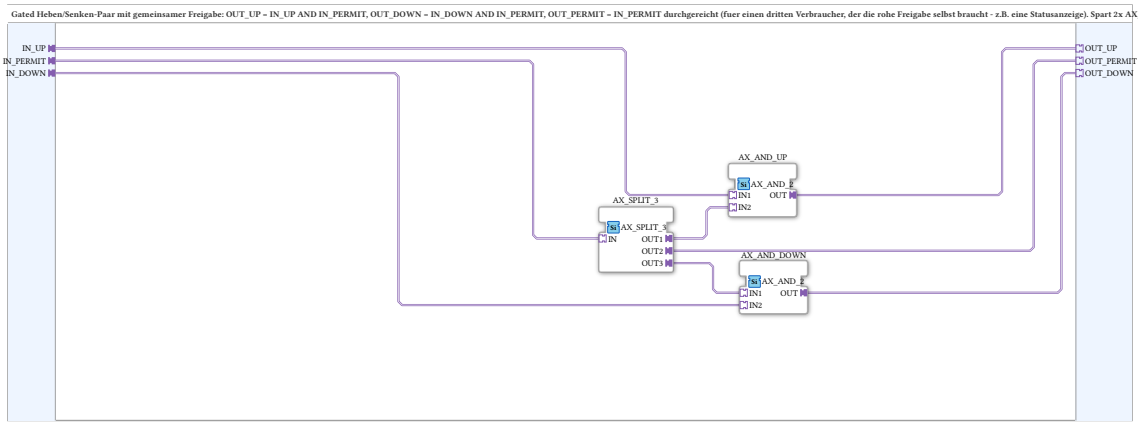
Zusammenfassung

`AX_3_3` ist eine reine Verdrahtungshilfe, die drei `AX`-Quellen per ODER zusammenführt und auf drei Ziele verteilt — ohne eigene Zustandslogik.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

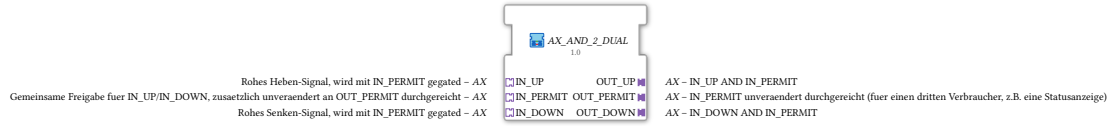
AX_AND_2_DUAL



• — ○ • — ○ • — ○ •

Einleitung

`AX_AND_2_DUAL` ist ein gegatetes Heben/Senken-Paar mit gemeinsamer Freigabe: `OUT_UP = IN_UP AND IN_PERMIT` und `OUT_DOWN = IN_DOWN AND IN_PERMIT`. Zusätzlich wird `IN_PERMIT` unverändert als `OUT_PERMIT` durchgereicht, z. B. für einen dritten Verbraucher wie eine Statusanzeige. Die SubApp ersetzt zweimal `AX_AND_2` plus einmal `AX_SPLIT_3` als Einzelbausteine.



Schnittstellenstruktur

Ereignis-Eingänge
Keine.

Ereignis-Ausgänge
Keine.

Daten-Eingänge
Keine.

Daten-Ausgänge
Keine.

Adapter
Alle Adapter sind vom Typ `adapter::types::unidirectional::AX`.

- **IN_UP** (Socket): rohes Heben-Signal, wird mit `IN_PERMIT` gegated
- **IN_PERMIT** (Socket): gemeinsame Freigabe für `IN_UP`/`IN_DOWN`, zusätzlich unverändert an `OUT_PERMIT` durchgereicht
- **IN_DOWN** (Socket): rohes Senken-Signal, wird mit `IN_PERMIT` gegated
- **OUT_UP** (Plug): `IN_UP AND IN_PERMIT`
- **OUT_PERMIT** (Plug): `IN_PERMIT` unverändert (für einen dritten Verbraucher, z. B. Statusanzeige)
- **OUT_DOWN** (Plug): `IN_DOWN AND IN_PERMIT`

Verwendete Funktionsbausteine (FBs)
- **`AX_SPLIT_3`** (`adapter::events::unidirectional::AX_SPLIT_3`): verteilt `IN_PERMIT` auf drei Ziele
- **`AX_AND_UP`**, **`AX_AND_DOWN`** (je `adapter::booleanOperators::AX_AND_2`): UND-Verknüpfung

Funktionsweise
- `IN_PERMIT` ⟶ `AX_SPLIT_3.IN`
- `AX_SPLIT_3.OUT1` ⟶ `AX_AND_UP.IN2`; `IN_UP` ⟶ `AX_AND_UP.IN1`; `AX_AND_UP.OUT` ⟶ `OUT_UP`
- `AX_SPLIT_3.OUT3` ⟶ `AX_AND_DOWN.IN1`; `IN_DOWN` ⟶ `AX_AND_DOWN.IN2`; `AX_AND_DOWN.OUT` ⟶ `OUT_DOWN`
- `AX_SPLIT_3.OUT2` ⟶ `OUT_PERMIT`

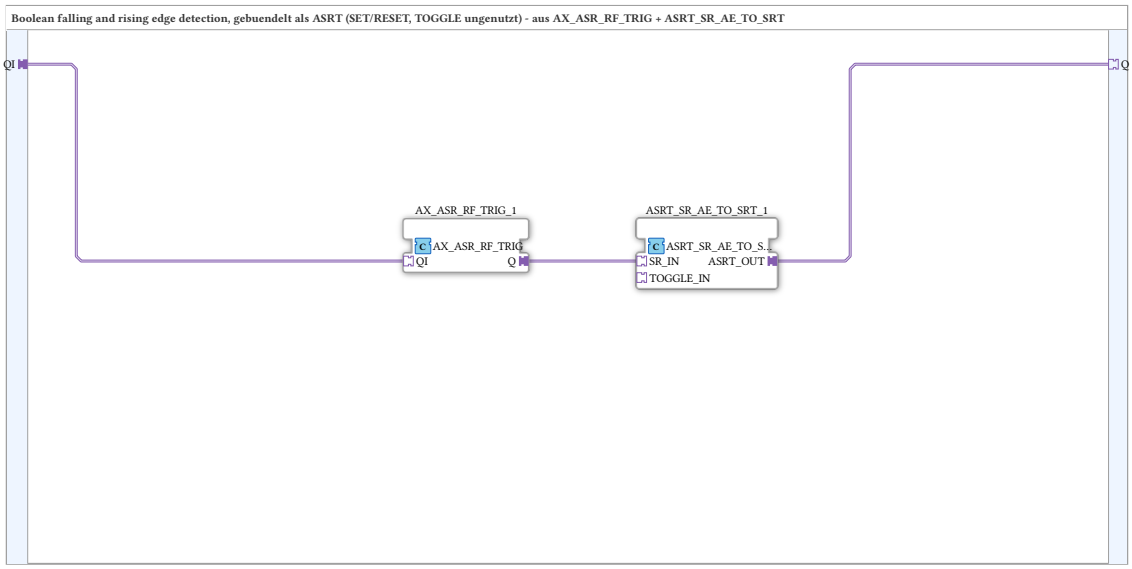
Technische Besonderheiten
- Da ein Adapter nicht auf mehrere Ziele verzweigen darf, verteilt `AX_SPLIT_3` die Freigabe auf beide UND-Glieder und den Durchreich-Ausgang
- Keine eigene Logik außerhalb der verwendeten Standardbausteine

Anwendungsszenarien
- Jedes Heben/Senken-Signalpaar, das dieselbe Freigabe benötigt (z. B. Verriegelung), wobei die Freigabe zusätzlich für eine Anzeige zur Verfügung stehen soll

Zusammenfassung
`AX_AND_2_DUAL` gatet ein Heben/Senken-Paar mit einer gemeinsamen Freigabe und gibt die Freigabe zusätzlich durch.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AX_ASRT_RF_TRIG



• – ○ • – ○ • – ○ •

Einleitung

`AX_ASRT_RF_TRIG` erkennt steigende und fallende Flanken eines `AX`-Signals und bündelt sie als `ASRT`-Adapter (Set/Reset, Toggle bleibt ungenutzt). Statt eines neuen Low-Level-`FBType` ist der Baustein als Composite aus dem bestehenden `AX_ASR_RF_TRIG` (liefert ein `ASR` aus einer Flanke) und `ASRT_SR_AE_TO_SRT` (kombiniert ein `ASR` mit einem optionalen AE-Toggle-Event zu einem `ASRT`) aufgebaut.

value to check for a rising or falling edge – *AX*

ASRT – Set/Reset Output (Toggle wird nie gefeuert)

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: AX_ASRT_RF_TRIG

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **AX_ASR_RF_TRIG_1**: `adapter::events::unidirectional::AX_ASR_RF_TRIG` — steigende Flanke an `QI` ⟶ SET, fallende Flanke ⟶ RESET, gebündelt als `ASR`.
  - **ASRT_SR_AE_TO_SRT_1**: `adapter::conversion::unidirectional::ASRT_SR_AE_TO_SRT` — übernimmt das `ASR` als `SR_IN` und bündelt es zu `ASRT_OUT`; der zweite Socket (`TOGGLE_IN`, ein einzelnes AE-Event) bleibt unverdrahtet und feuert nie, da sich aus einem reinen Flankensignal kein unabhängiges drittes Ereignis ableiten lässt.
- **Funktionsweise**: `AX_ASR_RF_TRIG_1` erzeugt aus der Flanke von `QI` ein `ASR`, das direkt als `SR_IN` in `ASRT_SR_AE_TO_SRT_1` gespeist wird, dessen `ASRT_OUT` den Plug `Q` bedient.

Programmablauf und Verbindungen

1. `QI` (Socket) ⟶ `AX_ASR_RF_TRIG_1.QI`.
2. `AX_ASR_RF_TRIG_1.Q` ⟶ `ASRT_SR_AE_TO_SRT_1.SR_IN`.
3. `ASRT_SR_AE_TO_SRT_1.ASRT_OUT` ⟶ `Q` (Plug).

Technische Besonderheiten

- **Composite statt neuem Low-Level-Baustein**: Wiederverwendet zwei bestehende Bausteine, statt einen neuen `FBType` zu implementieren.
- **TOGGLE-Socket bleibt ungenutzt**: `ASRT_SR_AE_TO_SRT_1.TOGGLE_IN` wird nie verdrahtet und feuert daher nie — der Baustein liefert also nur SET/RESET, nie ein echtes Toggle-Ereignis.

Anwendungsszenarien

- Als eine von mehreren ASRT-Quellen für einen ASRT_MERGE_2, z. B. in Uebung_232_AX.

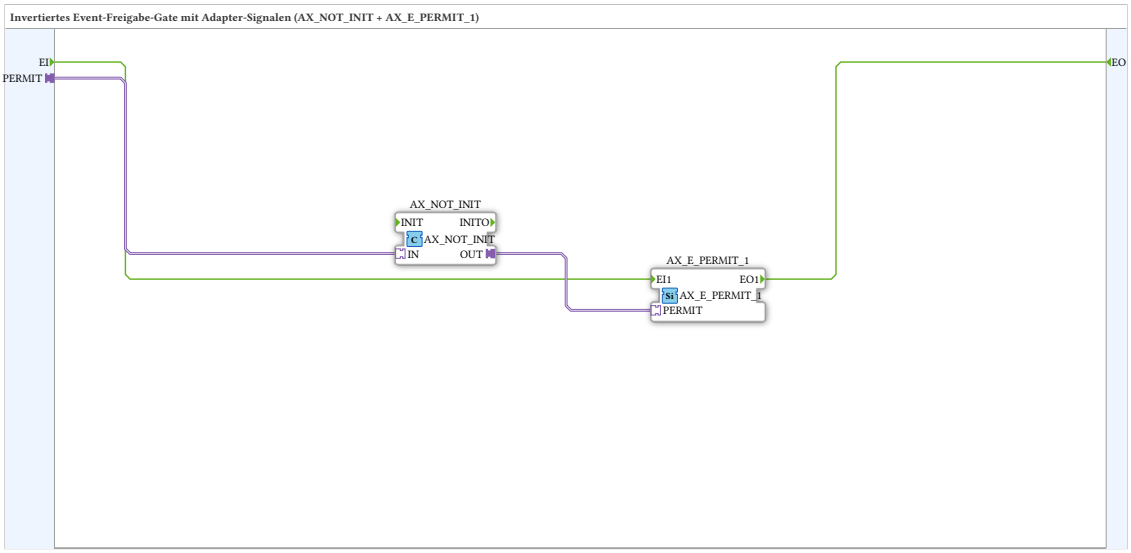
Zusammenfassung

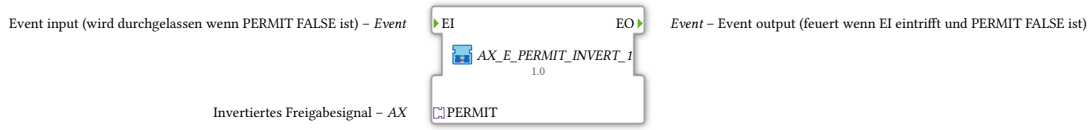
AX_ASRT_RF_TRIG liefert aus einer einzigen Flankenerkennung ein ASRT-Adapter-Signal (nur Set/Reset, Toggle ungenutzt) und ist damit direkt an Bausteine anschließbar, die einen ASRT-Socket erwarten.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AX_E_PERMIT_INVERT_1





• — ○ • — ○ • — ○ •

Einleitung

Der Baustein AX_E_PERMIT_INVERT_1 realisiert eine invertierte Event-Freigabe auf Basis von Adapter-Signalen. Er kombiniert die Funktionalität eines invertierenden Adapter-Operators (AX_NOT_INIT) mit einem ereignisbasierten Freigabe-Gate (AX_E_PERMIT_1). Dadurch wird ein Ereignis nur dann vom Eingang EI zum Ausgang EO durchgereicht, wenn das über den Adapter anliegende Freigabesignal PERMIT den logischen Zustand FALSE besitzt. Der Baustein ist als Subapplikation (SubApp) in der Bibliothek MyLib::sys verfügbar.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
EI	Event	Ereigniseingang – wird durchgelassen, wenn das Freigabesignal FALSE ist.

Ereignis-Ausgänge

Name	Typ	Kommentar
EO	Event	Ereignisausgang – feuert, wenn EI eintrifft und das Freigabesignal FALSE ist.

Daten-Eingänge

Es existieren keine Daten-Eingänge.

Daten-Ausgänge

Es existieren keine Daten-Ausgänge.

Adapter

Name	Richtung	Typ	Kommentar
PERMIT	Socket	adapter::types::unidirectional::AX	Invertiertes Freigabesignal; der eingehende Adapter-Wert wird intern negiert.

Funktionsweise

Der Baustein verarbeitet ein Ereignis, das am Eingang EI anliegt, abhängig vom aktuellen Wert des Adapter-Signals PERMIT. Die interne Verschaltung besteht aus zwei Funktionsblöcken:

1. **AX_NOT_INIT** – Dieser Block erhält das Adapter-Signal PERMIT und erzeugt an seinem Ausgang die logische Negation des empfangenen Wertes.
2. **AX_E_PERMIT_1** – Dieser Block stellt ein ereignisbasiertes Freigabe-Gate dar. Er besitzt einen Ereigniseingang EI1, einen Ereignisausgang EO1 und einen Adapter-Eingang PERMIT. Ein Ereignis wird nur dann vom Eingang zum Ausgang durchgereicht, wenn das am Adapter-Eingang anliegende Signal den Zustand TRUE besitzt.

Die Verbindungen innerhalb der Subapp sind:

- Ereignis EI ⟶ AX_E_PERMIT_1.EI1
- Ereignis AX_E_PERMIT_1.EO1 ⟶ EO
- Adapter PERMIT ⟶ AX_NOT_INIT.IN
- Adapter AX_NOT_INIT.OUT ⟶ AX_E_PERMIT_1.PERMIT

Damit ergibt sich folgende logische Kette:

- **Wenn PERMIT = TRUE:**
  AX_NOT_INIT invertiert zu FALSE. Das Freigabe-Gate AX_E_PERMIT_1 ist gesperrt, und das Ereignis EI wird nicht nach EO durchgelassen.
- **Wenn PERMIT = FALSE:**
  AX_NOT_INIT invertiert zu TRUE. Das Freigabe-Gate ist geöffnet, und ein am Eingang EI eintreffendes Ereignis wird unverzögert an den Ausgang EO weitergegeben.

Technische Besonderheiten

- Der Baustein nutzt Adapter-Signale (Typ AX) zur Steuerung, nicht klassische Daten-Ein-/Ausgänge. Dadurch ist eine enge Kopplung an das Adapter-Protokoll AX gegeben.
- Die Invertierung wird durch den separaten Funktionsblock AX_NOT_INIT realisiert, der als eigenständige wiederverwendbare Komponente in der Bibliothek vorhanden ist.
- Es werden keine Zustände gespeichert; die Verarbeitung erfolgt rein ereignisgesteuert und kombinatorisch.
- Die Subapp ist als einfache Kombination zweier Standard-FBs aufgebaut und ermöglicht eine hohe Wartbarkeit.

Zustandsübersicht

Da der Baustein keinen internen Speicher besitzt, lassen sich lediglich zwei externe logische Zustände unterscheiden:

Zustand	Bedingung (PERMIT)	Verhalten am Ausgang EO
Freigegeben	FALSE	Eingang EI wird zum Ausgang EO durchgereicht.
Gesperrt	TRUE	Eingang EI wird ignoriert; EO bleibt inaktiv.

Anwendungsszenarien

- **Sicherheitslogik:** Freigabe von Ereignissen nur bei inaktivem/permit-negativem Signal, z. B. in Schutzeinrichtungen oder Not-Aus-Ketten.
- **Signalsteuerung:** Deaktivierung von Ereignisströmen in Automatisierungssystemen, wenn ein bestimmter Adapter-Zustand aktiv ist.
- **Protokollanpassung:** Einfache Umkehrung einer Freigabe-Logik in industriellen Steuerungen, ohne zusätzliche Programmlogik in der aufrufenden Applikation.

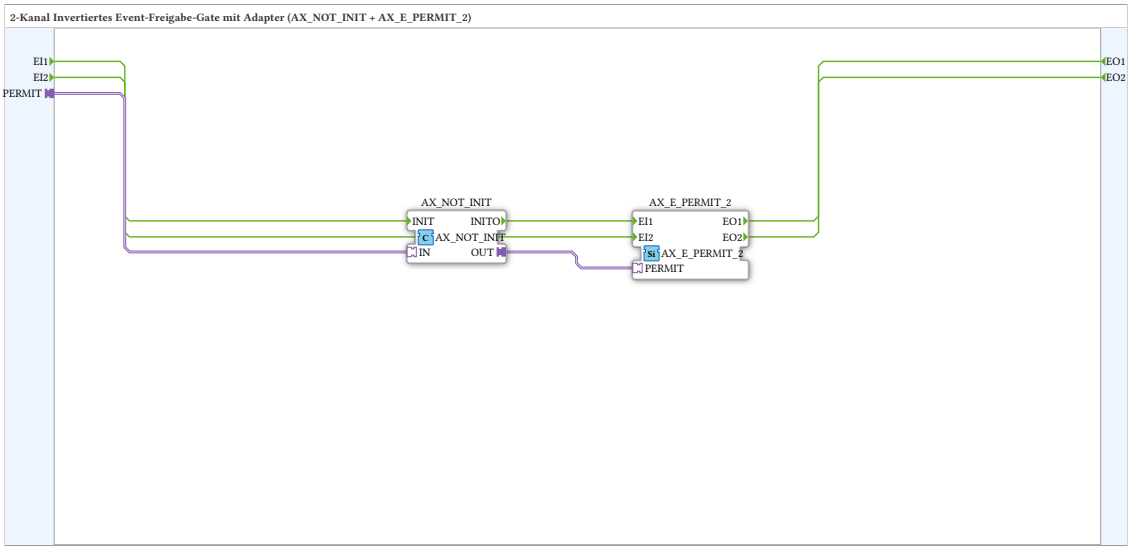
Vergleich mit ähnlichen Bausteinen

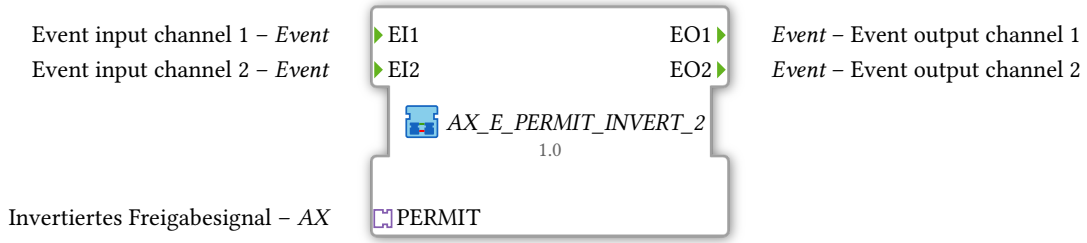
Im Gegensatz zu einem normalen Event-Freigabe-Gate (`AX_E_PERMIT_1`), das Ereignisse bei `TRUE` durchlässt, ist `AX_E_PERMIT_INVERT_1` genau invers: Es lässt Ereignisse bei `FALSE` durch. Vorteile gegenüber einer externen Invertierung des Signals sind die kompakte Kapselung und die klare Lesbarkeit. Ähnliche Bausteine könnten mit einem negierten Adapter-Signal oder einer anderen Grundlogik aufgebaut sein, aber dieser Baustein bietet eine vordefinierte, wiederverwendbare Lösung.

Fazit

`AX_E_PERMIT_INVERT_1` ist ein nützlicher Baustein, um ereignisbasierte Freigaben in invertierter Form zu realisieren. Er kombiniert die bewährten Funktionen `AX_NOT_INIT` und `AX_E_PERMIT_1` zu einer kompakten Einheit und erleichtert damit die Implementierung von „Freigabe bei inaktivem Signal“-Logiken. Die klare Schnittstelle und die rein kombinatorische Funktionsweise machen ihn zu einer zuverlässigen Komponente in STEP- und IEC-61499-Anwendungen.

AX_E_PERMIT_INVERT_2





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AX_E_PERMIT_INVERT_2** ist eine Subapplikation, die zwei Ereignis-Eingänge über ein invertiertes Freigabesignal auf zwei Ereignis-Ausgänge durchschaltet. Er kombiniert dabei einen Inverter (AX_NOT_INIT) mit einem 2-kanaligen Ereignis-Freigabe-Gate (AX_E_PERMIT_2). Die Subapplikation erlaubt es, Ereignisse nur dann weiterzuleiten, wenn das über den Adapter `PERMIT` anliegende

Freigabesignal **inaktiv** ist. Dadurch eignet sie sich für Logiken, bei denen eine fehlende Freigabe (z. B. bei Sicherheitsfunktionen) die Ereignisweiterleitung blockiert.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
EI1	Event	Event input channel 1
EI2	Event	Event input channel 2

Ereignis-Ausgänge

Name	Typ	Kommentar
EO1	Event	Event output channel 1
EO2	Event	Event output channel 2

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Name	Typ	Kommentar
PERMIT	adapter::types::unidirectional::AX	Invertiertes Freigabesignal

Der Adapter `PERMIT` ist ein unidirektionaler Eingangsadapter, der das Freigabesignal als booleschen Wert liefert.

Funktionsweise

Die Subapplikation verbindet intern die beiden Ereignis-Eingänge `EI1` und `EI2` direkt mit den entsprechenden Eingängen des Bausteins `AX_E_PERMIT_2`. Dieser Baustein leitet ein Ereignis genau dann von seinem Eingang `EIx` zu seinem Ausgang `EOx` weiter, wenn sein Freigabesignal `PERMIT` aktiv (logisch 1) ist.

Das Freigabesignal der Subapplikation wird jedoch **vor** der Verwendung durch den Inverter `AX_NOT_INIT` logisch negiert. Dadurch dreht sich die Freigabelogik um:

- Ist das äußere `PERMIT`-Signal **aktiv** (Wert = TRUE), dann wird es nach der Invertierung zu FALSE. Das Gate `AX_E_PERMIT_2` sperrt die Ereignisweiterleitung.
- Ist das äußere `PERMIT`-Signal **inaktiv** (Wert = FALSE), dann wird es nach der Invertierung zu TRUE. Das Gate lässt die Ereignisse von `EI1` zu `EO1` und von `EI2` zu `EO2` durch.

Somit fungiert die Subapplikation als **invertiertes Freigabegate**: Ereignisse werden nur dann weitergegeben, wenn das Freigabesignal nicht ansteht.

Technische Besonderheiten

- **Kapselung**: Die interne Verdrahtung ist in der Subapplikation verborgen; nach außen erscheint nur die Schnittstelle mit den Ereignissen und dem Adapter.
- **Keine Datenwege**: Es existieren keine Daten-Ein-/Ausgänge – die Subapplikation arbeitet ausschließlich mit Ereignissen und booleschen Werten über den Adapter.
- **Adaptertyp**: Der Adapter `PERMIT` ist vom Typ `adapter::types::unidirectional::AX`, ein unidirektionaler Eingangsadapter, der für reine Signalübertragung ohne Rückkanal ausgelegt ist.
- **Zwei Kanäle**: Die Ereignispaare (EI1/EO1, EI2/EO2) sind unabhängig, werden aber gemeinsam durch das Freigabesignal gesteuert.

Zustandsübersicht

Die Subapplikation selbst besitzt keinen expliziten Zustand, da sie ausschließlich aus kombinatorischen und ereignisgesteuerten Funktionen besteht. Die interne Funktionalität lässt sich wie folgt beschreiben:

Zustand / Bedingung	Verhalten
`PERMIT` = FALSE	Ereignisse an EI1/EI2 werden an EO1/EO2 durchgereicht
`PERMIT` = TRUE	Ereignisse an EI1/EI2 werden blockiert

Ein definierter Initialzustand ist nicht vorhanden; das Verhalten hängt ausschließlich vom aktuellen Wert des Adapters ab.

Anwendungsszenarien

- **Sicherheitslogik**: Durchschalten von Ereignissen nur dann, wenn ein Freigabesignal **nicht** aktiv ist (z. B. bei Not-Aus-Pfaden, bei denen Ereignisse bei aktiver Störung unterdrückt werden).
- **Invertierte Sperre**: In Automatisierungsanlagen, wo Ereignisse (z. B. Taktimpulse) nur bei abgeschalteter Freigabe verarbeitet werden sollen.
- **Test-Szenarien**: Bei der Simulation von Fehlerzuständen, bei denen das Fehlen eines Signals die normale Verarbeitung erlaubt.

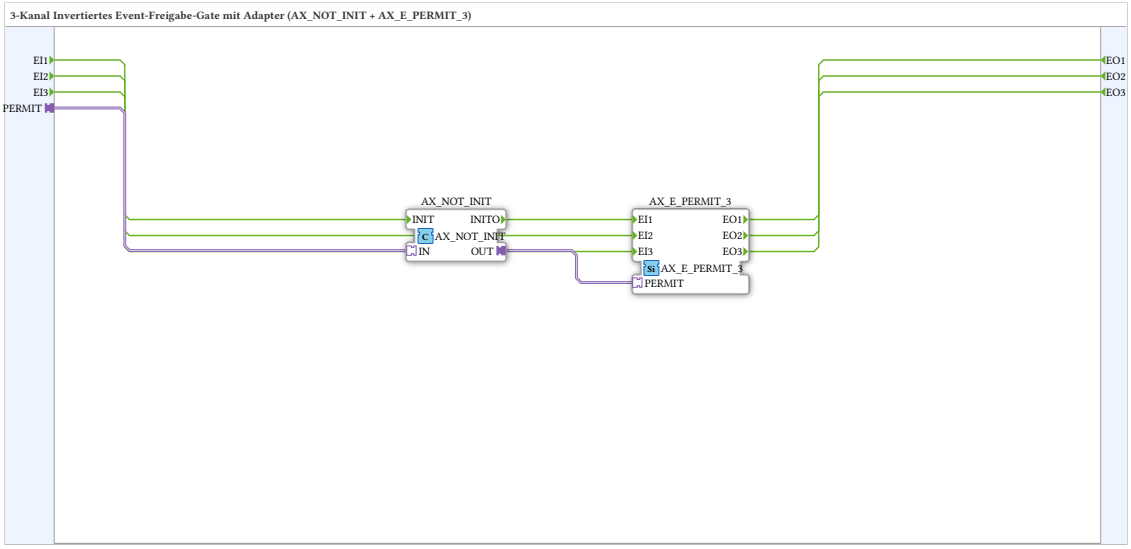
Vergleich mit ähnlichen Bausteinen

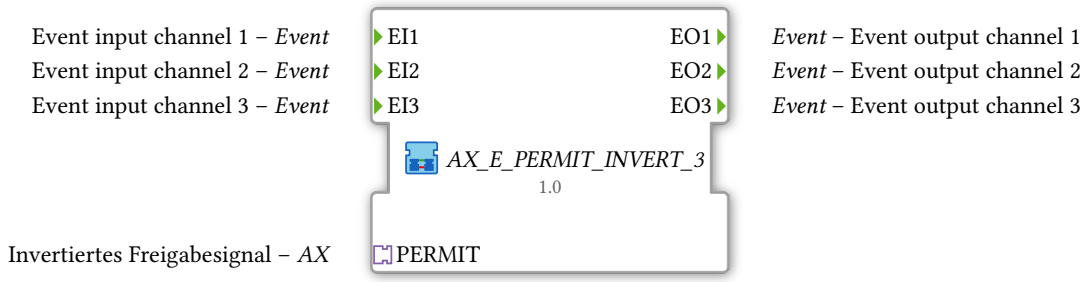
Baustein	Beschreibung	Unterschied
`AX_E_PERMIT_2`	2-Kanal Freigabegate: Ereignisse werden bei **aktivem** Freigabesignal durchgelassen.	`AX_E_PERMIT_INVERT_2` invertiert das Freigabesignal, sodass bei inaktivem Signal durchgeschaltet wird.
`AX_E_PERMIT_1`	1-Kanal Variante ohne Invertierung	Nur ein Ereigniskanal, keine Invertierung.
`AX_NOT_INIT`	Wandler für boolesche Signale mit Invertierung	Nur als Einzelbaustein für logische Negation; keine Ereignissteuerung.

Fazit

Der Funktionsblock **AX_E_PERMIT_INVERT_2** stellt eine kompakte, wiederverwendbare Lösung für eine invertierte, 2-kanalige Ereignisfreigabe dar. Durch die Kombination eines Inverters mit einem standardisierten Freigabegate wird eine klare und robuste Steuerungslogik erreicht. Die weglassung von Daten-Ein-/Ausgängen und die reine Ereignisorientierung machen ihn ideal für ereignisbasierte Automatisierungssysteme, die eine präzise Freigabesteuerung benötigen.

AX_E_PERMIT_INVERT_3





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AX_E_PERMIT_INVERT_3** ist eine Subapplikation (SubApp) zur Realisierung eines **3-Kanal invertierten Event-Freigabe-Gates**. Er kombiniert die Bausteine `AX_NOT_INIT` (logische Negation eines Adaptersignals) und `AX_E_PERMIT_3` (3-Kanal Event-Freigabe) zu einer gekapselten, wiederverwendbaren Einheit.

Im Gegensatz zu einer direkten Freigabe werden die eingehenden Events **nur dann durchgeschaltet**, wenn das über den Adapter anliegende Freigabesignal **FALSE** ist. Ein aktives Freigabesignal (TRUE) sperrt dagegen alle drei Eventkanäle.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
EI1	Event	Ereignis-Eingangskanal 1
EI2	Event	Ereignis-Eingangskanal 2
EI3	Event	Ereignis-Eingangskanal 3

Ereignis-Ausgänge

Name	Typ	Kommentar
EO1	Event	Ereignis-Ausgangskanal 1
EO2	Event	Ereignis-Ausgangskanal 2
EO3	Event	Ereignis-Ausgangskanal 3

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

Name	Richtung	Typ	Kommentar
PERMIT	Socket	`adapter::types::unidirectional::AX`	Invertiertes Freigabesignal

Funktionsweise

Die SubApp besteht aus zwei intern verbundenen Funktionsbausteinen:

1. **AX_NOT_INIT** – Ein unidirektionaler Adapterbaustein, der das am Eingang `IN` anliegende Signal logisch negiert und am Ausgang `OUT` bereitstellt.

2. **AX_E_PERMIT_3** – Ein 3-Kanal-Event-Freigabebaustein, der die Ereignisse von EI1/EI2/EI3 an die entsprechenden Ausgänge EO1/EO2/EO3 nur dann weiterleitet, wenn das Signal an seinem Freigabeeingang PERMIT **TRUE** ist.

Intern wird das äußere Freigabesignal PERMIT der SubApp an AX_NOT_INIT.IN geführt. Das negierte Ausgangssignal AX_NOT_INIT.OUT wird mit dem Freigabeeingang PERMIT von AX_E_PERMIT_3 verbunden. Somit gilt für jeden der drei Kanäle:

- Ist das äußere Freigabesignal PERMIT **FALSE**, ist das invertierte Signal **TRUE** und die Events werden durchgeschaltet.
- Ist das äußere Freigabesignal PERMIT **TRUE**, ist das invertierte Signal **FALSE** und die Events werden blockiert.

Die Verzögerung ist rein ereignisgesteuert und rein kombinatorisch; es werden keine Datenwerte verarbeitet oder gespeichert.

Technische Besonderheiten

- **SubApp-Kapselung**: Die Kombination aus Negation und Freigabe wird als eigener Baustein bereitgestellt und kann in verschiedenen Anwendungen wiederverwendet werden.
- **Unidirektionaler Adapter**: Der Adapter PERMIT vom Typ AX ist unidirektional und überträgt ausschließlich ein boolesches Freigabesignal.
- **Keine Datenpfade**: Die SubApp besitzt weder Daten-Ein- noch Daten-Ausgänge; die Funktionalität ist rein ereignisbasiert.
- **Keine interne Zustandshaltung**: Das Verhalten ist deterministisch und hängt nur vom aktuellen Wert des Freigabesignals ab.
- **Drei unabhängige Kanäle**: Die Kanäle 1–3 werden parallel und unabhängig voneinander gesteuert.

Zustandsübersicht

Da die SubApp keine eigenen internen Zustände besitzt, ergibt sich die Zustandsübersicht aus der Kombination von Freigabesignal und Ereignisweiterleitung:

PERMIT (extern)	Invertiertes Signal	Ereignis EIx ⟶ EOx
FALSE	TRUE	durchgeschaltet
TRUE	FALSE	blockiert

Diese Tabelle gilt für jeden einzelnen Kanal (x = 1, 2, 3) gleichermaßen.

Anwendungsszenarien

- **Sperrlogik mit invertiertem Freigabesignal**: Wenn ein Steuerungssystem ein Sicherheitssignal als „Sperre“ interpretiert (TRUE = gesperrt) und Events nur im ungesperrten Zustand durchlaufen sollen.
- **Freigabe bei inaktivem Signal**: Anwendungen, bei denen eine Maschine oder ein Prozess nur dann Ereignisse verarbeiten darf, wenn ein bestimmtes Signal **nicht** aktiv ist – z. B. bei Ruhezustand oder manuellem Modus.
- **Mehrkanalige Ereignisverarbeitung**: Drei parallele Ereignispfade, die gemeinsam über ein einziges Freigabesignal gesteuert werden sollen, ohne dass die Negation separat implementiert werden muss.

Vergleich mit ähnlichen Bausteinen

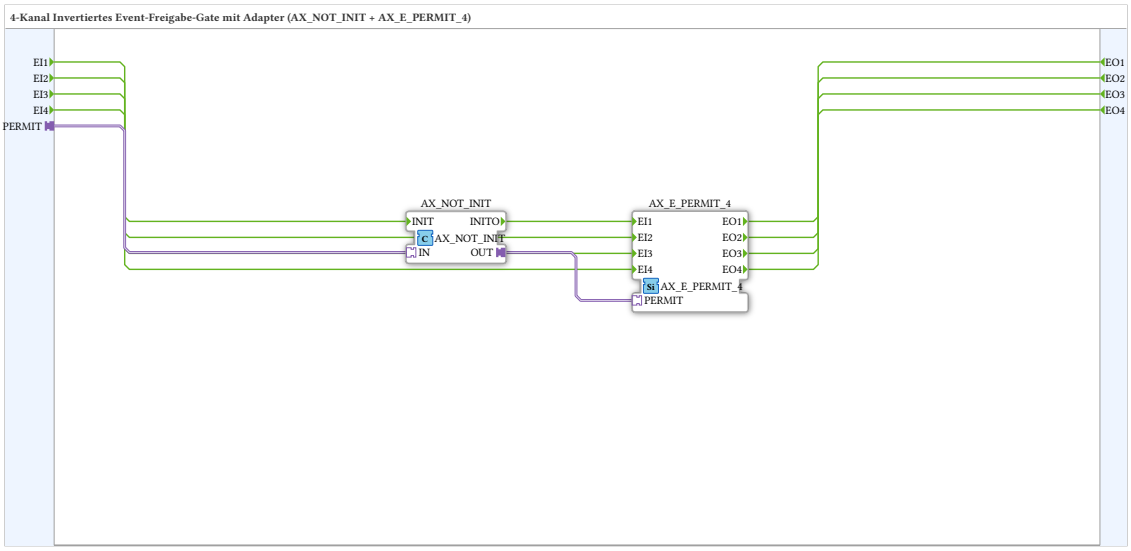
- **AX_E_PERMIT_3**: Der direkte Freigabebaustein leitet Events nur bei **TRUE** am Freigabeeingang weiter. AX_E_PERMIT_INVERT_3 ist die invertierte Variante davon.
- **AX_NOT_INIT + AX_E_PERMIT_3** (einzeln verwendet): Bietet dieselbe Funktionalität, erfordert aber eine manuelle Verdrahtung. Die SubApp kapselt diese Verbindung und reduziert so die Fehleranfälligkeit und den Engineering-Aufwand.
- **FBs mit Daten-Eingängen**: Alternative Bausteine könnten ein Freigabesignal über einen Daten-Eingang statt über einen Adapter erhalten – AX_E_PERMIT_INVERT_3 nutzt dagegen eine standardisierte Adapterschnittstelle.

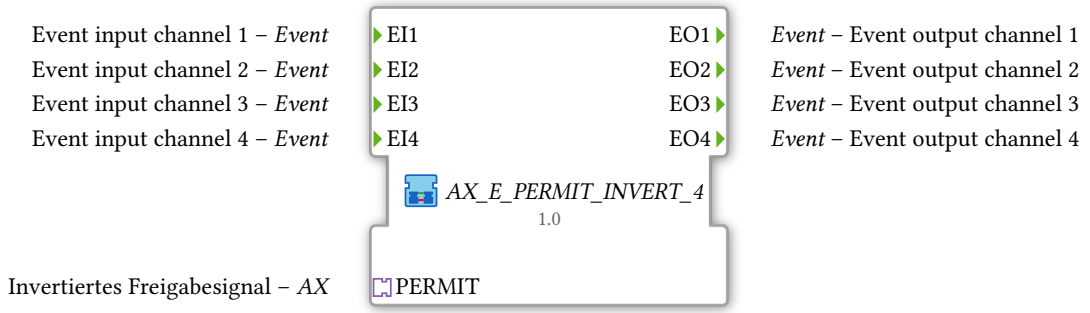
Fazit

Die SubApp **AX_E_PERMIT_INVERT_3** stellt eine kompakte und klar strukturierte Lösung für eine dreikanalige, invertierte Event-Freigabe dar. Durch die Kombination von Negation und Freigabe in einem Baustein vereinfacht sie das Engineering, erhöht die Wiederverwendbarkeit und macht die Applikati-

onslogik besser lesbar. Sie eignet sich besonders für Szenarien, in denen Ereignisse nur bei inaktivem Freigabesignal verarbeitet werden dürfen.

AX_E_PERMIT_INVERT_4





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock (SubApp) **AX_E_PERMIT_INVERT_4** realisiert ein 4-Kanal-Event-Freigabe-Gate mit invertierter Freigabelogik. Er kombiniert die Bausteine `AX_NOT_INIT` und `AX_E_PERMIT_4` zu einer kompakten Einheit, die über einen Adapter gesteuert wird. Die vier Event-Eingänge werden nur dann auf die jeweiligen Ausgänge durchgeschaltet, wenn das Freigabesignal **inaktiv** (FALSE) ist – eine logische Invertierung des üblichen Freigabe-Verhaltens.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
EI1	Event	Event-Eingang Kanal 1
EI2	Event	Event-Eingang Kanal 2
EI3	Event	Event-Eingang Kanal 3
EI4	Event	Event-Eingang Kanal 4

Ereignis-Ausgänge

Name	Typ	Kommentar

EO1	Event	Event-Ausgang Kanal 1
EO2	Event	Event-Ausgang Kanal 2
EO3	Event	Event-Ausgang Kanal 3
EO4	Event	Event-Ausgang Kanal 4

Daten-Eingänge
Keine vorhanden.

Daten-Ausgänge
Keine vorhanden.

Adapter

Name	Typ	Kommentar
PERMIT	adapter::types::unidirectional::AX	Invertiertes Freigabesignal (Socket)

Funktionsweise
Die SubApp `AX_E_PERMIT_INVERT_4` besteht intern aus zwei miteinander verbundenen Funktionsbausteinen:

1. **AX_NOT_INIT** – invertiert das über den Adapter `PERMIT` eingehende Freigabesignal.
2. **AX_E_PERMIT_4** – nutzt das invertierte Signal als Freigabe für vier parallele Event-Kanäle.

Die Adapterverbindung stellt sicher, dass das am `PERMIT`-Socket anliegende Signal zuerst invertiert wird, bevor es auf den Freigabeeingang des Event-Tores gelangt. Dadurch ergibt sich folgende Logik:

- Wenn `PERMIT = TRUE` ⟶ Invertiertes Signal = `FALSE` ⟶ Alle vier Event-Kanäle sind **gesperrt**.
- Wenn `PERMIT = FALSE` ⟶ Invertiertes Signal = `TRUE` ⟶ Alle vier Event-Kanäle sind **durchgeschaltet**.

Ereignisse an `EI1` bis `EI4` werden also nur dann zu den entsprechenden Ausgängen `EO1` bis `EO4` weitergeleitet, wenn das Freigabesignal am Adapter NICHT aktiv ist.

Technische Besonderheiten
- Die SubApp verwendet Adapter-Typen (`unidirectional::AX`), die eine lose Kopplung zwischen den Bausteinen ermöglichen.
- Die Invertierung und die Freigabelogik sind in einer einzigen SubApp gekapselt, was die Wiederverwendung und Wartung erleichtert.
- Es sind keine zeitlichen Verzögerungen oder internen Zustandsmaschinen vorhanden – die Durchschaltung erfolgt rein ereignis- und signalabhängig.
- Die Anordnung der internen Bausteine (x, y-Koordinaten) ist für die Netzwerk-Ausführung irrelevant, lediglich die Verdrahtung ist entscheidend.

Zustandsübersicht
Die SubApp selbst besitzt keinen internen Zustand. Der Durchlasszustand der vier Kanäle hängt ausschließlich vom aktuellen Wert des Adaptersignals `PERMIT` ab. Der effektive Zustand lässt sich wie folgt beschreiben:

Adapter PERMIT	Invertiertes Signal	Kanalverhalten
TRUE	FALSE	Events werden blockiert
FALSE	TRUE	Events werden durchgelassen

Anwendungsszenarien
- **Sicherheitslogik**: Freigabe für den Normalbetrieb, Sperre bei aktivem Alarmsignal.
- **NOT-Aus-Logik**: Maschinen- oder Prozesssteuerung, bei der Ereignisse nur bei fehlendem Freigabesignal verarbeitet werden dürfen.
- **Datenfluss-Steuerung**: Selektives Durchschalten von Event-Strömen in Abhängigkeit von einer invertierten Bedingung.

- **Prototypen-Entwicklung**: Einfache Erweiterung vorhandener Freigabe-Logiken durch Hinzufügen eines `NOT`-Gliedes.

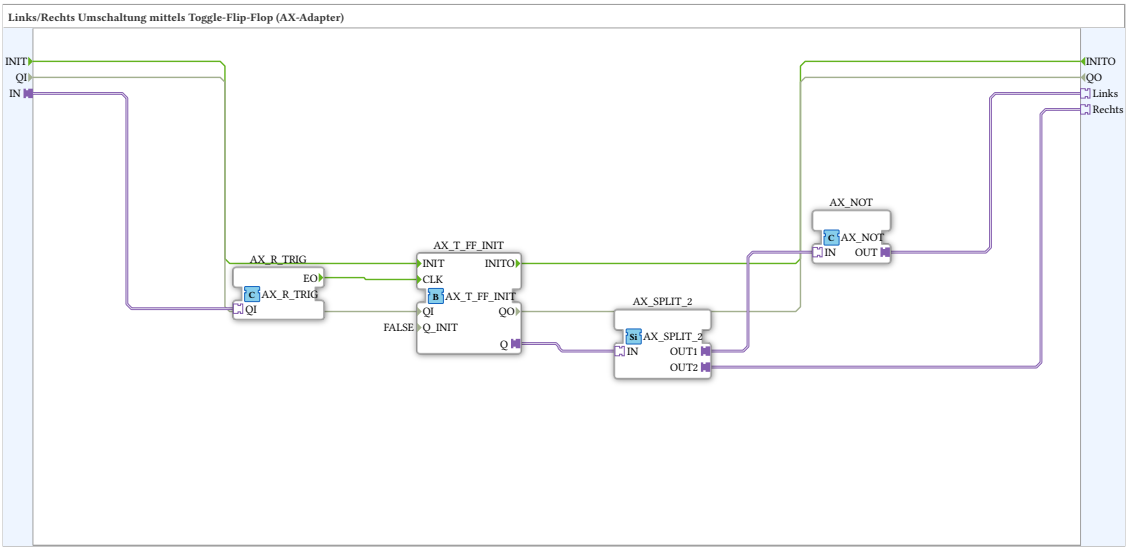
Vergleich mit ähnlichen Bausteinen

- **AX_E_PERMIT_4** (ohne Invertierung): Erlaubt die Durchschaltung bei `PERMIT = TRUE`. `AX_E_PERMIT_INVERT_4` kehrt diese Logik um.
- **AX_NOT_INIT + AX_E_PERMIT_4** als separate Bausteine: Bietet dieselbe Funktionalität, erfordert aber eine manuelle Verkabelung. Die SubApp kapselt diese Kombination und vereinfacht die Einbindung in bestehende Systeme.
- **NICHT-Gatter** (rein boolesch): Ohne Event-Verarbeitung möglich, aber diese SubApp kombiniert Invertierung mit Event-Gating in einem Baustein.

Fazit

`AX_E_PERMIT_INVERT_4` ist eine kompakte und wiederverwendbare Lösung für alle Anwendungen, die eine invertierte Freigabesteuerung über mehrere Kanäle benötigen. Durch die Kombination von Adapter, Inverter und Event-Gate wird eine klare, leicht verständliche Schnittstelle geschaffen, die sich nahtlos in industrielle Steuerungssysteme integrieren lässt. Die fehlenden Zustandsübergänge und die rein kombinatorische Logik machen den Baustein besonders robust und deterministisch.

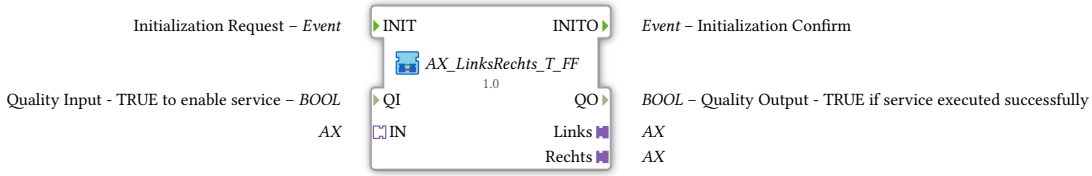
AX_LinksRechts_T_FF



• – ○ • – ○ • – ○ •

Einleitung

`AX_LinksRechts_T_FF` wandelt einen einzelnen Taster (`IN`) über ein Toggle-Flipflop in zwei komplementäre Adapter-Ausgänge `Links`/`Rechts` um — jeder Tastendruck schaltet zwischen „Links aktiv“ und „Rechts aktiv“ um.



Schnittstellenstruktur

- **Ereignis-Eingänge**: `INIT` (Initialization Request)
- **Ereignis-Ausgänge**: `INITO` (Initialization Confirm)
- **Daten-Eingänge**: `QI` (`BOOL`, Default `TRUE`) — Quality Input
- **Daten-Ausgänge**: `QO` (`BOOL`) — Quality Output
- **Adapter**: Socket `IN`, Plugs `Links` und `Rechts` (jeweils `adapter::types::unidirectional::AX`)

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: AX_LinksRechts_T_FF

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **AX_R_TRIG**: `adapter::events::unidirectional::AX_R_TRIG` — erkennt die steigende Flanke des Tasters.
  - **AX_T_FF_INIT**: `adapter::events::unidirectional::AX_T_FF_INIT` — Toggle-Flipflop mit definiertem Startzustand (`Q_INIT=FALSE`; `QI` kommt vom Wrapper-Eingang `QI`).
  - **AX_SPLIT_2**: `adapter::events::unidirectional::AX_SPLIT_2` — verzweigt den Flipflop-Zustand.
  - **AX_NOT**: `adapter::booleanOperators::AX_NOT` — negiert einen der beiden Zweige für das komplementäre Signal.
- **Funktionsweise**: Jede steigende Flanke an `IN` togglet `AX_T_FF_INIT.Q`. Der direkte Zustand geht an `Rechts`, der negierte Zustand an `Links` — dadurch sind `Links` und `Rechts` immer exakt komplementär (siehe Kommentar im Quellcode: „der AX_T_FF_INIT schaltet den Ausgang Aktiv auf FALSE, somit ist LINKS dann TRUE“).

Programmablauf und Verbindungen

1. `IN` ⟶ `AX_R_TRIG.QI`; `AX_R_TRIG.EO` ⟶ `AX_T_FF_INIT.CLK` (togglet bei jeder steigenden Flanke).
2. `AX_T_FF_INIT.Q` ⟶ `AX_SPLIT_2.IN`.
3. `AX_SPLIT_2.OUT1` ⟶ `AX_NOT.IN` ⟶ `AX_NOT.OUT` ⟶ `Links`.
4. `AX_SPLIT_2.OUT2` ⟶ `Rechts`.
5. `INIT` ⟶ `AX_T_FF_INIT.INIT`; `AX_T_FF_INIT.INITO` ⟶ `INITO`.
6. `QI` ⟶ `AX_T_FF_INIT.QI`; `AX_T_FF_INIT.QO` ⟶ `QO`.

Anwendungsszenarien

- Ein-Taster-Umschaltung zwischen zwei sich gegenseitig ausschließenden Zuständen (z. B. Fahrtrichtung, Anzeige-Umschaltung), ohne zwei separate Taster zu benötigen.

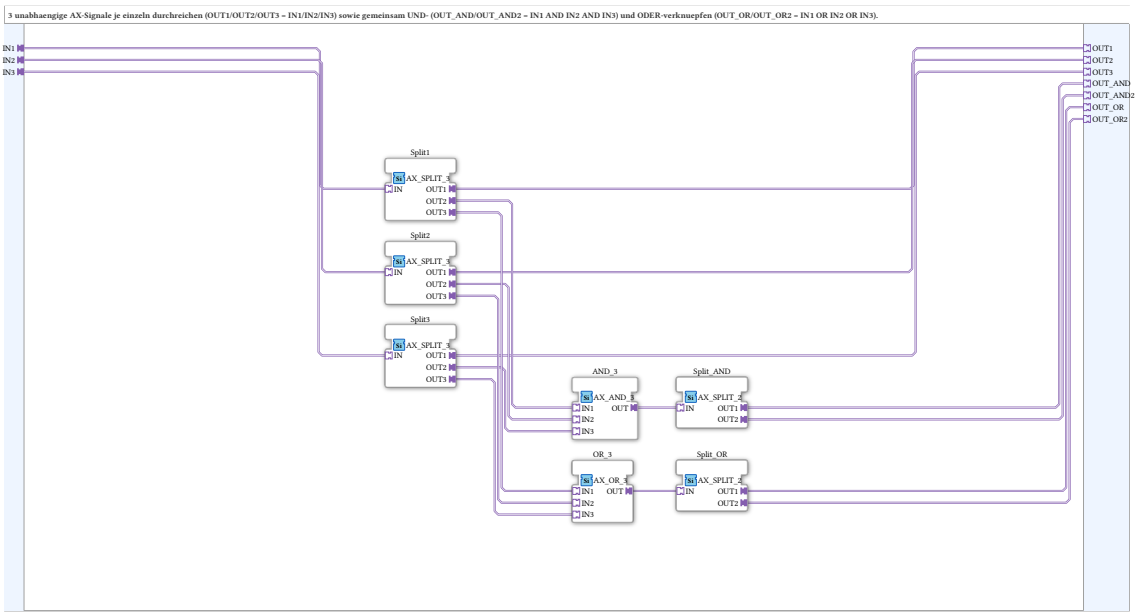
Zusammenfassung

`AX_LinksRechts_T_FF` realisiert eine klassische Ein-Taster-Umschaltung zwischen zwei komplementären Zuständen über ein Toggle-Flipflop mit definiertem Startzustand und eine Negation für das Gegensignal.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

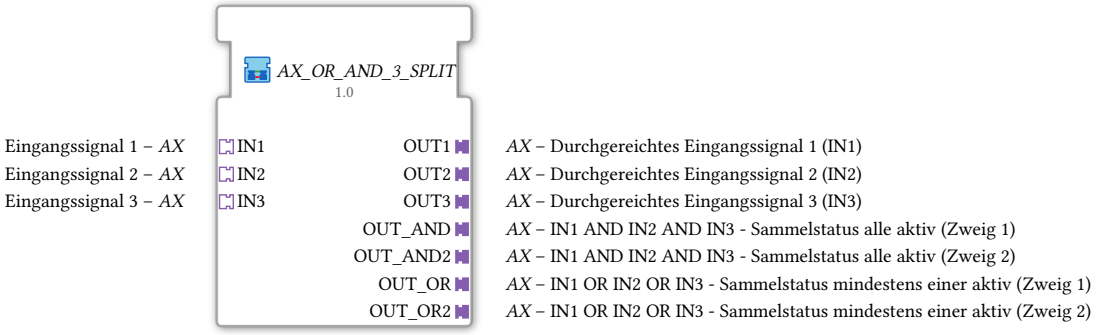
AX_OR_AND_3_SPLIT



• – ◦ • – ◦ • – ◦ •

Einleitung

`AX_OR_AND_3_SPLIT` reicht **drei unabhängige AX-Signale** einzeln durch (`OUT1`/`OUT2`/`OUT3` = `IN1`/`IN2`/`IN3`) und verknüpft sie zusätzlich gemeinsam **UND** (`OUT_AND`/`OUT_AND2` = `IN1 AND IN2 AND IN3`) sowie **ODER** (`OUT_OR`/`OUT_OR2` = `IN1 OR IN2 OR IN3`). Die Sammelausgänge sind jeweils doppelt (Zweig 1/Zweig 2) ausgeführt.



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden – Ereignisse kommen über die Adapter-Sockets/-Plugs

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

Alle Adapter sind unidirektional vom Typ `adapter::types::unidirectional::AX`.

- **IN1**, **IN2**, **IN3** (Sockets): Eingangssignale 1 bis 3
- **OUT1**, **OUT2**, **OUT3** (Plugs): Durchgereichte Eingangssignale 1 bis 3
- **OUT_AND**, **OUT_AND2** (Plugs): `IN1 AND IN2 AND IN3` – Sammelstatus „alle aktiv“ (Zweig 1/Zweig 2)
- **OUT_OR**, **OUT_OR2** (Plugs): `IN1 OR IN2 OR IN3` – Sammelstatus „mindestens einer aktiv“ (Zweig 1/Zweig 2)

Verwendete Funktionsbausteine (FBs)

- **Split1**, **Split2**, **Split3**: je `adapter::events::unidirectional::AX_SPLIT_3` (verzweigt jedes Eingangssignal auf drei Ziele)
- **AND_3**: `adapter::booleanOperators::AX_AND_3`
- **OR_3**: `adapter::booleanOperators::AX_OR_3`
- **Split_AND**, **Split_OR**: je `adapter::events::unidirectional::AX_SPLIT_2` (verdoppeln den jeweiligen Sammelausgang)

Programmablauf und Verbindungen

- `INx` ⟶ `Splitx.IN`
- `Splitx.OUT1` ⟶ `OUTx` (Durchreichen)
- `Splitx.OUT2` ⟶ `AND_3.INx`
- `Splitx.OUT3` ⟶ `OR_3.INx`
- `AND_3.OUT` ⟶ `Split_AND.IN`; `Split_AND.OUT1` ⟶ `OUT_AND`, `Split_AND.OUT2` ⟶ `OUT_AND2`
- `OR_3.OUT` ⟶ `Split_OR.IN`; `Split_OR.OUT1` ⟶ `OUT_OR`, `Split_OR.OUT2` ⟶ `OUT_OR2`

(mit x = 1, 2, 3)

Technische Besonderheiten

- Die UND- und ODER-Sammelausgänge sind aufgedoppelt (`AX_SPLIT_2`), damit je zwei Verbraucher versorgt werden können.

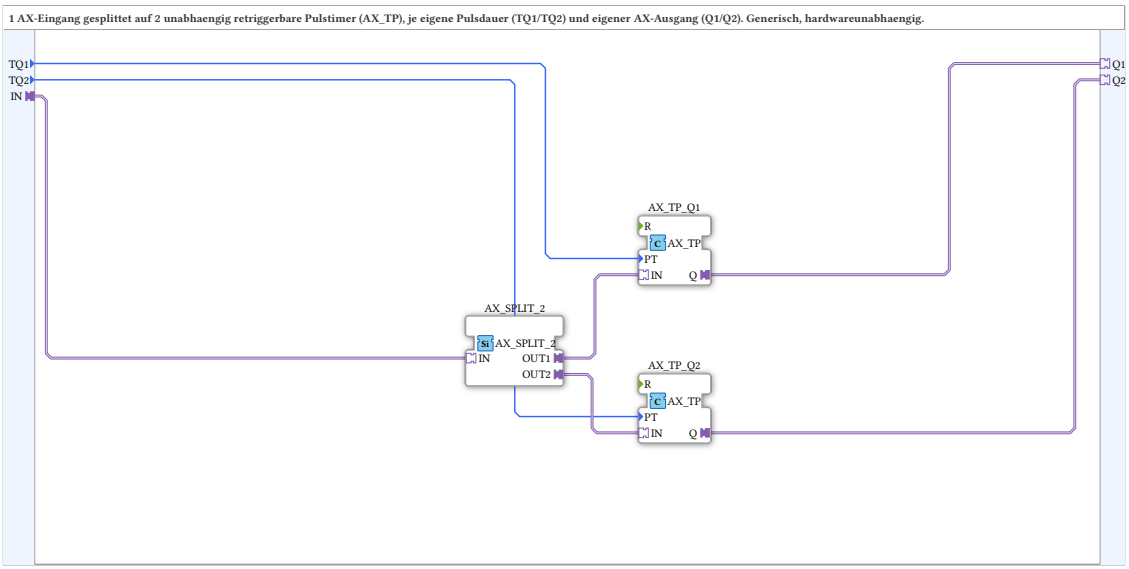
Anwendungsszenarien

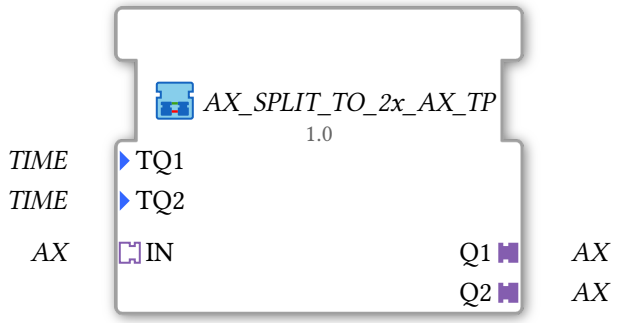
- Mehrere Statussignale (z. B. Bänder) einzeln weiterverwenden und zusätzlich einen gemeinsamen Status „alle aktiv“ bzw. „mindestens einer aktiv“ bilden

Fazit

`AX_OR_AND_3_SPLIT` bündelt Durchreichen, UND- und ODER-Verknüpfung dreier AX-Signale in einer SubApp.

AX_SPLIT_TO_2x_AX_TP





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `AX_SPLIT_TO_2x_AX_TP` ist eine wiederverwendbare Subapplikation (SubApp), die einen einzelnen AX-Eingang auf zwei unabhängig voneinander arbeitende, retriggerbare Pulstimer (Typ `AX_TP`) aufteilt. Jeder Ausgang besitzt eine eigene, einstellbare Pulsdauer (`TQ1` bzw. `TQ2`) und stellt einen separaten AX-Ausgang (`Q1` bzw. `Q2`) bereit. Die SubApp ist generisch aufgebaut und hardwareunabhängig einsetzbar.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine separaten Ereignis-Eingänge vorhanden. Ereignissignale werden über die Adapter-Schnittstelle `IN` übertragen.

Ereignis-Ausgänge

Es sind keine separaten Ereignis-Ausgänge definiert. Die ereignisbasierten Signale werden über die Adapter-Ausgänge `Q1` und `Q2` ausgegeben.

Daten-Eingänge

Name	Typ	Beschreibung
TQ1	TIME	Pulsdauer für den Timer `AX_TP_Q1`
TQ2	TIME	Pulsdauer für den Timer `AX_TP_Q2`

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden. Die Ergebnisse werden über die Adapter-Ausgänge `Q1` und `Q2` zur Verfügung gestellt.

Adapter

Richtung	Name	Typ	Beschreibung
Eingang	IN	`adapter::types::unidirectional::AX`	AX-Eingangssignal, das aufgeteilt wird
Ausgang	Q1	`adapter::types::unidirectional::AX`	AX-Ausgang des ersten Timers
Ausgang	Q2	`adapter::types::unidirectional::AX`	AX-Ausgang des zweiten Timers

Funktionsweise

Die SubApp arbeitet in drei Stufen:

1. **Split:** Das an `IN` anliegende AX-Signal wird durch den internen Baustein `AX_SPLIT_2` auf zwei identische Ausgänge (`OUT1` und `OUT2`) aufgeteilt.
2. **Timer:** Jeder der beiden geteilten Signale wird einem eigenen `AX_TP`-Timer (`AX_TP_Q1` bzw. `AX_TP_Q2`) zugeführt. Diese Timer sind retriggerbar, d.h. bei jedem neuen Ereignis am Eingang startet die voreingestellte Pulsdauer erneut.
3. **Ausgabe:** Die Ausgangssignale der Timer (`Q`) werden direkt an die Adapter-Plugs `Q1` bzw. `Q2` weitergeleitet.

Die Pulsdauern werden über die Dateneingänge `TQ1` (für `AX_TP_Q1`) und `TQ2` (für `AX_TP_Q2`) festgelegt. Da beide Timer vollständig unabhängig arbeiten, können unterschiedliche Pulszeiten gleichzeitig verwendet werden.

Technische Besonderheiten

- **Vollständig entkoppelte Ausgänge:** Beide Ausgangskanäle besitzen eigene Timer und Pulsdauern – eine Beeinflussung ist nicht möglich.
- **Retriggerbarkeit:** Die eingesetzten Timer vom Typ `AX_TP` unterstützen das erneute Triggern während laufender Pulsdauer, was eine präzise Steuerung bei häufigen Ereignissen ermöglicht.
- **Generischer Aufbau:** Die SubApp verwendet keine hardwareabhängigen Elemente und kann in verschiedenen Projektumgebungen eingesetzt werden.
- **Wiederverwendbarkeit:** Der Baustein ist durch die klare Schnittstellentrennung leicht in größere Systeme integrierbar.

Zustandsübersicht

Die SubApp selbst besitzt keinen eigenen Zustandsautomaten. Die internen Bausteine `AX_SPLIT_2` und `AX_TP` verfügen über typische Zustände eines AX-Puls-Timers:
„Idle“, „Timer läuft“ und „Puls aktiv“. Diese sind jedoch nicht Bestandteil dieser Dokumentation, da die SubApp nur deren Verschaltung kapselt.

Anwendungsszenarien

- **Zwei unabhängige Ventile:** Ein gemeinsames Startsignal muss zwei Ventile mit unterschiedlicher Öffnungszeit ansteuern.
- **Signalkaskadierung:** Aufteilung eines Ereignisses auf zwei parallele Signalpfade mit getrennten Zeitvorgaben.

- **Testumgebungen:** Erzeugung unterschiedlicher Pulsbreiten aus einem Quellsignal für Prüfzwecke.

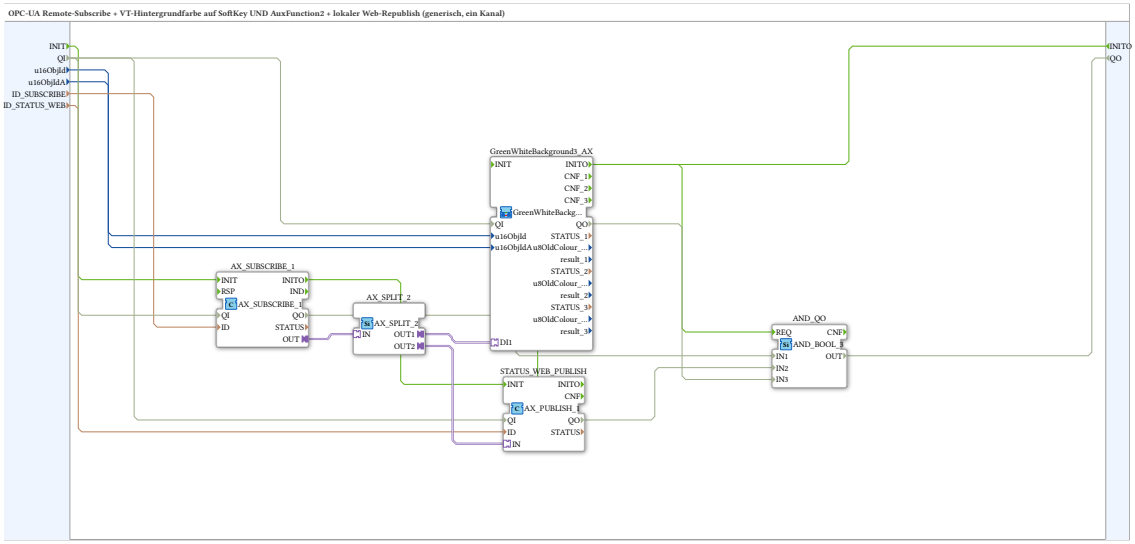
Vergleich mit ähnlichen Bausteinen

Baustein / Ansatz	Vorteile	Nachteile
`AX_SPLIT_TO_2x_AX_TP` (vorliegend)	Zwei getrennt einstellbare Timer, unabhängig	Etwas mehr Overhead durch interne FBs
Einfacher Split (ohne Timer)	Minimaler Ressourcenbedarf	Keine Zeitsteuerung, nur Verteilung
Split mit einem gemeinsamen Timer	Geringe Komplexität	Keine getrennten Pulszeiten möglich

Fazit

`AX_SPLIT_TO_2x_AX_TP` ist eine flexible und robuste Lösung, um ein einzelnes AX-Ereignis in zwei zeitlich unabhängig steuerbare Ausgangssignale aufzuteilen. Durch die klare Trennung der Pulsdauern und die Retriggerbarkeit eignet sich der Baustein besonders für Anwendungen, bei denen unterschiedliche zeitliche Bedingungen pro Ausgang erforderlich sind. Die generische Bauweise ermöglicht eine unkomplizierte Integration in unterschiedliche Steuerungsarchitekturen.

AX_SUBSCRIBE_BG3_WEB_OPC



• — ○ • — ○ • — ○ •

Einleitung

`AX_SUBSCRIBE_BG3_WEB_OPC` ist die Status-Hälfte von `Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC`, herausgelöst als eigenständiger Baustein: ein Remote-Subscribe-Wert wird sowohl als Hintergrundfarbe auf einem SoftKey UND einer AuxFunction2 dargestellt (`GreenWhiteBackground3_AX`) als auch lokal per OPC-UA erneut veröffentlicht, damit ein Web-Client (z. B. vt-ui-mirror) auf demselben Modul denselben Status ohne eigene Verbindung zum Zielmodul lesen kann.

Schnittstelle

- **Ereignis-Eingang**: `INIT` (`EInit`, Initialization Request)
- **Ereignis-Ausgang**: `INITO` (`EInit`, Initialization Confirm)
- **Daten-Eingänge**: `QI` (`BOOL`, Initialwert `TRUE`, Quality Input), `u16ObjId`, `u16ObjIdA`, `ID_SUBSCRIBE`, `ID_STATUS_WEB`
- **Daten-Ausgang**: `QO` (`BOOL`, Quality Output)

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: AX_SUBSCRIBE_BG3_WEB_OPC

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **AX_SUBSCRIBE_1**: `adapter::net::AX_SUBSCRIBE_1` (`QI` vom Baustein-Eingang `QI`) — Remote-Subscribe der Status/Farbe (`ID_SUBSCRIBE`).
  - **AX_SPLIT_2**: `adapter::events::unidirectional::AX_SPLIT_2` — verzweigt das empfangene Signal auf die VT-Anzeige und den lokalen Web-Republish.
  - **GreenWhiteBackground3_AX** (SubApp, `MyLib::sys`): färbt sowohl einen SoftKey (`u16ObjId`) als auch eine AuxFunction2 (`u16ObjIdA`) synchron; besitzt selbst `INIT`/`INITO` und `QI`/`QO`.
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_3` — verknüpft die `QO`-Ausgänge von `AX_SUBSCRIBE_1`, `STATUS_WEB_PUBLISH` und `GreenWhiteBackground3_AX` per UND zum Baustein-Ausgang `QO`.
  - **STATUS_WEB_PUBLISH**: `adapter::net::AX_PUBLISH_1` (`QI` vom Baustein-Eingang `QI`) — veröffentlicht den empfangenen Zustand erneut auf dem lokalen OPC-UA-Server dieses Moduls (`ID_STATUS_WEB`).
- **Funktionsweise**: Der per Remote-Subscribe empfangene Zustand wird über `AX_SPLIT_2` gleichzeitig an die VT-Hintergrundfarbe und an den lokalen Web-Republish weitergereicht.

Programmablauf und Verbindungen

1. `u16ObjId` → `GreenWhiteBackground3_AX.u16ObjId`; `u16ObjIdA` → `GreenWhiteBackground3_AX.u16ObjIdA`; `ID_SUBSCRIBE` → `AX_SUBSCRIBE_1.ID`; `ID_STATUS_WEB` → `STATUS_WEB_PUBLISH.ID` (Datenverbindungen, ausgeblendet).
2. `AX_SUBSCRIBE_1.OUT` → `AX_SPLIT_2.IN` → `AX_SPLIT_2.OUT1` → `GreenWhiteBackground3_AX.DI1` (VT-Anzeige), `AX_SPLIT_2.OUT2` → `STATUS_WEB_PUBLISH.IN` (Web-Republish).
3. Initialisierungskette: `INIT` → `AX_SUBSCRIBE_1.INIT`, `AX_SUBSCRIBE_1.INITO` → `STATUS_WEB_PUBLISH.INIT`, `STATUS_WEB_PUBLISH.INITO` → `GreenWhiteBackground3_AX.INIT`, `GreenWhiteBackground3_AX.INITO` → `INITO`; zusätzlich `GreenWhiteBackground3_AX.INITO` → `AND_QO.REQ`.
4. `QI` → `AX_SUBSCRIBE_1.QI`, `STATUS_WEB_PUBLISH.QI` und `GreenWhiteBackground3_AX.QI`.
5. `AX_SUBSCRIBE_1.QO` → `AND_QO.IN1`, `STATUS_WEB_PUBLISH.QO` → `AND_QO.IN2`, `GreenWhiteBackground3_AX.QO` → `AND_QO.IN3`; `AND_QO.OUT` → `QO`.

Technische Besonderheiten

- **Erweiterung von `AX_SUBSCRIBE_BG_OPC`**: Gegenüber der einfacheren Variante (nur Subscribe + Hintergrundfarbe) kommen zwei Dinge hinzu: `GreenWhiteBackground3_AX` statt `GreenWhiteBackground1_AX` (färbt SoftKey und Aux gemeinsam) sowie der lokale Web-Republish über `AX_SPLIT_2`/`STATUS_WEB_PUBLISH`.
- **Entkopplung vom Zielmodul**: Ein Web-Client, der nur mit diesem Modul verbunden ist, muss keine eigene Verbindung zum Zielmodul aufbauen, um denselben Status wie das echte VT zu sehen.

Anwendungsszenarien

- Statusanzeige für Funktionen, die sowohl auf einem SoftKey als auch auf einer AuxFunction2 (Joystick-Zuweisung) synchron dargestellt werden müssen, plus Web-Spiegelung.
- Für nur einen SoftKey ohne Aux ist stattdessen `AX_SUBSCRIBE_BG_OPC` zu verwenden (kein Web-Republish).

Vergleich mit ähnlichen Bausteinen

Für die komplette, vorverdrahtete Kombination mit der Kommando-Seite (SoftKey/Aux-Auslesen + Remote-Write) ist weiterhin `Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC` zu verwenden, das diesen Baustein als Status-Teilbaustein enthält.

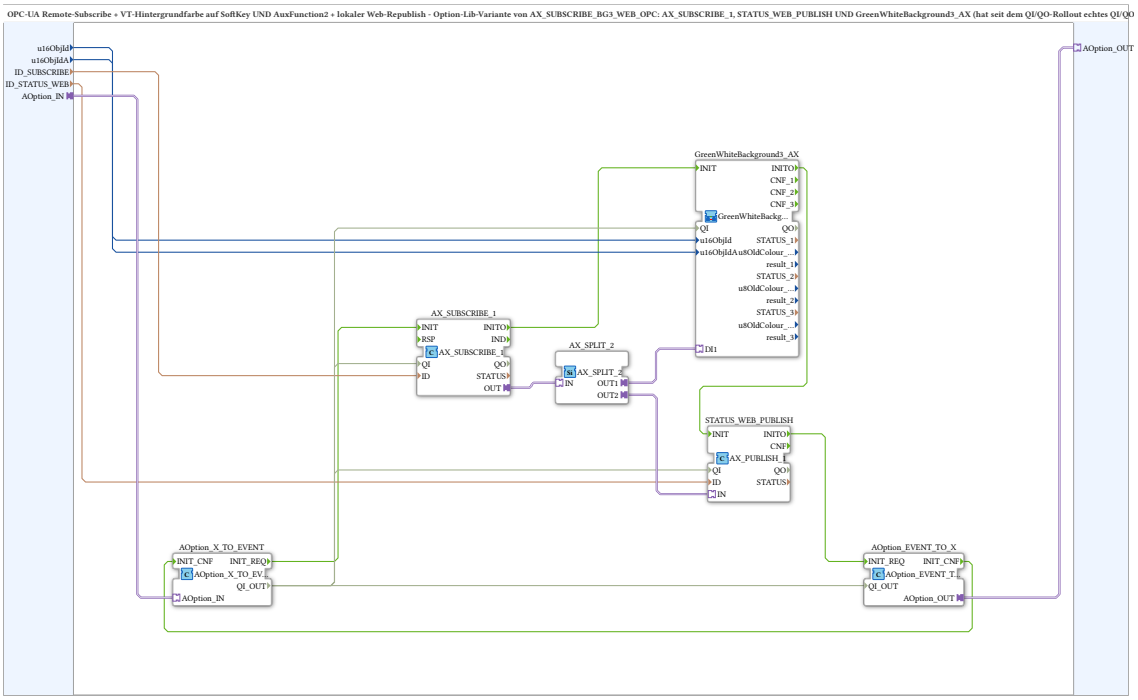
Zusammenfassung

`AX_SUBSCRIBE_BG3_WEB_OPC` kapselt die Status-Seite eines Remote-Kanals: Empfang, synchrone SoftKey/Aux-Hintergrundfarbe und lokaler Web-Republish in einem eigenständig wiederverwendbaren Baustein.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

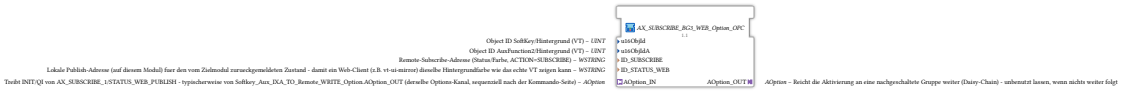
AX_SUBSCRIBE_BG3_WEB_Option_OPC



• — ○ • — ○ • — ○ •

Einleitung

`AX_SUBSCRIBE_BG3_WEB_Option_OPC` ist die Option-Lib-Variante von `AX_SUBSCRIBE_BG3_WEB_OPC`: OPC-UA-Remote-Subscribe mit VT-Hintergrundfarbe auf SoftKey UND AuxFunction2 sowie lokalem Web-Republish. `AX_SUBSCRIBE_1`, `STATUS_WEB_PUBLISH` und `GreenWhiteBackground3_AX` (hat seit dem QI/QO-Rollout echtes `QI`/`QO`) werden hier vom Adapter-Socket `Option::AOption` initialisiert und gated: alle drei sequenziell in der Reihenfolge `AX_SUBSCRIBE_1` -> `GreenWhiteBackground3_AX` -> `STATUS_WEB_PUBLISH`. Der Plug `AOption_OUT` reicht die Aktivierung an eine nachgeschaltete Gruppe weiter (Daisy-Chain).



Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden (die Initialisierung läuft über den Adapter `AOption_IN`).

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Variable	Typ	Vorgabe	Kommentar
`u16ObjId`	UINT	`ID_NULL`	Objekt-ID SoftKey/Hintergrund (VT).
`u16ObjIdA`	UINT	`ID_NULL`	Objekt-ID AuxFunction2/Hintergrund (VT).
`ID_SUBSCRIBE`	WSTRING		Remote-Subscribe-Adresse (Status/Farbe, `ACTION=SUBSCRIBE`).

ID_STATUS_WEB	WSTRING		Lokale Publish-Adresse für den vom Zielmodul zurückgemeldeten Zustand, damit ein Web-Client (z. B. vt-ui-mirror) dieselbe Hintergrundfarbe wie das echte VT zeigen kann.

Daten-Ausgänge
Keine vorhanden.

Adapter

Adapter	Typ	Richtung	Kommentar
AOption_IN	Option::AOption	Socket	Treibt INIT/QI von AX_SUBSCRIBE_1, STATUS_WEB_PUBLISH und GreenWhiteBackground3_AX; typischerweise vom AOption_OUT von Softkey_Aux_IXA_TO_Remote_WRITE_Option gespeist.
AOption_OUT	Option::AOption	Plug	Reicht die Aktivierung an eine nachgeschaltete Gruppe weiter (Daisy-Chain); unbenutzt lassen, wenn nichts folgt.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: AX_SUBSCRIBE_BG3_WEB_Option_OPC

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **AOption_X_TO_EVENT** (Option::conversion::AOption_X_TO_EVENT): setzt den Socket AOption_IN in INIT_REQ/QI_OUT um, siehe AOption_X_TO_EVENT.
  - **AX_SUBSCRIBE_1** (adapter::net::AX_SUBSCRIBE_1): OPC-UA-Subscribe auf ID_SUBSCRIBE.
  - **AX_SPLIT_2** (adapter::events::unidirectional::AX_SPLIT_2): verteilt den Subscribe-Wert auf zwei Zweige.
  - **GreenWhiteBackground3_AX** (SubApp MyLib::sys::GreenWhiteBackground3_AX): setzt die Hintergrundfarbe auf SoftKey und AuxFunction2; wird vom Option-Socket nach AX_SUBSCRIBE_1 und vor STATUS_WEB_PUBLISH sequenziell initialisiert und gated.
  - **STATUS_WEB_PUBLISH** (adapter::net::AX_PUBLISH_1): lokaler Publish des Status auf ID_STATUS_WEB.
  - **AOption_EVENT_TO_X** (Option::conversion::AOption_EVENT_TO_X): gibt die Aktivierung an AOption_OUT weiter.

Programmablauf und Verbindungen

1. Option-Kette: AOption_X_TO_EVENT.INIT_REQ -> AX_SUBSCRIBE_1.INIT; AX_SUBSCRIBE_1.INITO -> GreenWhiteBackground3_AX.INIT; GreenWhiteBackground3_AX.INITO -> STATUS_WEB_PUBLISH.INIT; STATUS_WEB_PUBLISH.INITO -> AOption_EVENT_TO_X.INIT_REQ; AOption_EVENT_TO_X.INIT_CNF -> AOption_X_TO_EVENT.INIT_CNF.
2. AOption_X_TO_EVENT.QI_OUT -> AX_SUBSCRIBE_1.QI, STATUS_WEB_PUBLISH.QI, GreenWhiteBackground3_AX.QI und AOption_EVENT_TO_X.QI_OUT.
3. Adapter: AOption_IN -> AOption_X_TO_EVENT.AOption_IN; AOption_EVENT_TO_X.AOption_OUT -> AOption_OUT.
4. Daten: AX_SUBSCRIBE_1.OUT -> AX_SPLIT_2.IN; AX_SPLIT_2.OUT1 -> GreenWhiteBackground3_AX.DI1; AX_SPLIT_2.OUT2 -> STATUS_WEB_PUBLISH.IN.
5. Parameter: ID_SUBSCRIBE -> AX_SUBSCRIBE_1.ID; ID_STATUS_WEB -> STATUS_WEB_PUBLISH.ID; u16ObjId/u16ObjIdA -> GreenWhiteBackground3_AX.u16ObjId/u16ObjIdA.
6. Der Baustein hat keine Ereignis- und keine QI/QO-Schnittstelle; Aktivierung und Bestätigung laufen ausschließlich über die Adapter AOption_IN/AOption_OUT.

Technische Besonderheiten

- **Alle drei Bausteine gated:** `AX_SUBSCRIBE_1`, `STATUS_WEB_PUBLISH` und `GreenWhiteBackground3_AX` hängen am Option-Socket (Version 1.1; `GreenWhiteBackground3_AX` hat seit dem QI/QO-Rollout echtes `QI`/`QO`, die frühere Annahme ohne Gating war veraltet).
- **Sequenzielle Initialisierung:** Die Option-Kette initialisiert nacheinander `AX_SUBSCRIBE_1`, `GreenWhiteBackground3_AX` und `STATUS_WEB_PUBLISH`; erst nach `STATUS_WEB_PUBLISH.INITO` feuert `AOption_EVENT_TO_X.INIT_REQ`.
- **Option-Initialisierung statt `INIT`/`QI`:** Der Baustein hat keine eigenen `INIT`/`INITO`/`QI`/`QO`-Anschlüsse; die Initialisierung läuft über den `INIT_REQ`/`INIT_CNF`-Handshake der Option-Kette, die Bestätigung geht über `AOption_EVENT_TO_X.INIT_CNF` zurück an `AOption_X_TO_EVENT`.
- **Daisy-Chain:** `AOption_OUT` erlaubt, weitere Instanzen oder Gruppen an dieselbe Option zu hängen.
- Das Gegenstück ohne Option ist `AX_SUBSCRIBE_BG3_WEB_OPC`.

Anwendungsszenarien

- Status-Hälfte von `Softkey_Aux_IXA_TO_Remote_WRITE_BG_Option_OPC`: Hintergrundfarbe auf SoftKey und Aux aus einem Remote-Subscribe, nur aktiv, wenn die Option ausgewählt ist.

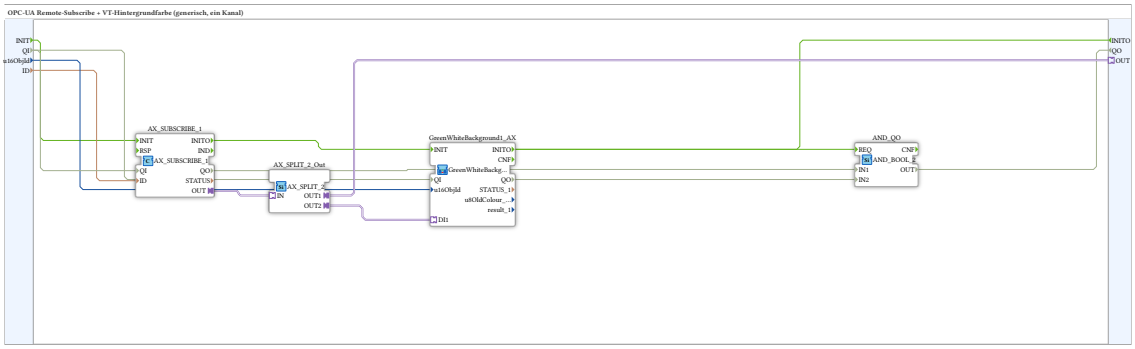
Zusammenfassung

`AX_SUBSCRIBE_BG3_WEB_Option_OPC` kombiniert Remote-Subscribe, Hintergrundfarbe auf SoftKey und Aux sowie lokalen Web-Republish und gated Subscribe, Publish und Hintergrundfarbe über einen `AOption`-Socket mit Weiterreichung per `AOption_OUT`.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AX_SUBSCRIBE_BG_OPC



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock **AX_SUBSCRIBE_BG_OPC** ist eine Subapplikation, die eine OPC-UA-Remote-Subscription mit einer Hintergrundfarbsteuerung für Visualisierungselemente (VT) verbindet. Er dient dazu, über eine OPC-UA-Verbindung einen Datenwert zu empfangen und daraus die Hintergrundfarbe eines benannten Rechtecks in einer Visualisierung dynamisch anzupassen. Die Subapp ist generisch aufgebaut und unterstützt einen Kanal.

Schnittstellenstruktur

Die Subapp besitzt den Ereigniseingang `INIT`, den Ereignisausgang `INITO`, die Dateneingänge `QI`, `u16ObjId` und `ID` sowie den Datenausgang `QO`; externe Adapter gibt es nicht. Die Verbindungen zu internen Bausteinen erfolgen über die eingebettete Netzwerkstruktur.

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsanforderung; startet die Initialisierungskette AX_SUBSCRIBE_1 → GreenWhiteBackground1_AX.

Ereignis-Ausgänge

- **INITO** (EInit): Initialisierungsbestätigung; kommt von GreenWhiteBackground1_AX.INITO.

Daten-Eingänge

Name	Typ	Kommentar
QI	BOOL	Quality Input, Freigabe des Dienstes (Initialwert TRUE)
u16ObjId	UINT	Objekt-ID des Hintergrund-Rechtecks (VT)
ID	WSTRING	OPC-UA-Remote-Subscribe-Adresse

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Quality Output, AND der QO-Ausgänge von AX_SUBSCRIBE_1 und GreenWhiteBackground1_AX

Adapter

Extern sind keine Adapter angelegt. Innerhalb der Subapp wird ein Adapter vom Typ adapter::net::AX_SUBSCRIBE_1 verwendet, dessen Ausgang OUT mit dem Eingang DI1 der Subapp GreenWhiteBackground1_AX verbunden ist.

Funktionsweise

Die Subapp kombiniert drei interne Komponenten:

1. **AX_SUBSCRIBE_1** – ein Adapter, der eine OPC-UA-Remote-Subscription abwickelt. Über den Dateneingang ID wird die OPC-UA-Adresse (z.B. eine Node-ID) bereitgestellt. Der Adapter empfängt fortlaufend Wertänderungen von dieser Adresse und stellt sie an seinem Ausgangs-Adapter OUT bereit.
2. **GreenWhiteBackground1_AX** – eine weitere Subapp, die die Hintergrundfarbe eines Visualisierungsobjekts (VT) steuert. Sie erhält über u16ObjId die Objekt-ID des Rechtecks und über den Adaptereingang DI1 die empfangenen Datenwerte. Abhängig vom Wert wechselt sie zwischen einer grünen und einer weißen Hintergrundfarbe.
3. **AND_QO** – ein iec61131::booleanOperators::AND_BOOL_2, der die QO-Ausgänge von AX_SUBSCRIBE_1 und GreenWhiteBackground1_AX zum SubApp-Ausgang QO verknüpft.

Durch die interne Verdrahtung wird der vom OPC-UA-Adapter gelieferte Wert direkt an die Hintergrundsteuerung weitergeleitet. Somit reagiert das Visualisierungselement unmittelbar auf Änderungen des OPC-UA-Datenpunkts.

Initialisierung und Qualität: INIT → AX_SUBSCRIBE_1.INIT, AX_SUBSCRIBE_1.INITO → GreenWhiteBackground1_AX.INIT, GreenWhiteBackground1_AX.INITO → INITO und → AND_QO.REQ. QI wird an AX_SUBSCRIBE_1.QI und GreenWhiteBackground1_AX.QI weitergereicht (kein fester QI=TRUE mehr im Subscribe-Baustein); AX_SUBSCRIBE_1.QO → AND_QO.IN1, GreenWhiteBackground1_AX.QO → AND_QO.IN2, AND_QO.OUT → QO.

Technische Besonderheiten

- **OPC-UA-Integration**: Verwendet den Adapter AX_SUBSCRIBE_1, der für das Abonnement von OPC-UA-Variablen optimiert ist.
- **Wiederverwendbarkeit**: Die Subapp ist vollständig parametrierbar über die Eingänge QI, u16ObjId und ID und kann ohne Änderung der internen Logik in unterschiedlichen Visualisierungsprojekten eingesetzt werden.
- **Standardkonformität**: Basierend auf IEC 61499-2, gekennzeichnet durch das <Identification>-Tag.
- **Kompatibilität**: Einbettung in eine Bibliothek MyLib::sys; verweist auf das Const isobus::UT::Q::const::IDs::ID_NULL als Initialwert für u16ObjId.

- **INIT/INITO**: Die Initialisierung wird durch die Subapp gereicht (`INIT` ⟶ `AX_SUBSCRIBE_1` ⟶ `GreenWhiteBackground1_AX` ⟶ `INITO`); im laufenden Betrieb erfolgt die Verarbeitung rein daten- und adapterbasiert.

Zustandsübersicht

Die Subapp selbst besitzt keinen eigenen Zustandsautomaten; ihr Verhalten wird vollständig durch die internen Bausteine bestimmt. Der Adapter `AX_SUBSCRIBE_1` verwaltet den Zustand der OPC-UA-Subscription (z.B. verbunden, verbindungslos, im Fehlerfall). Die Subapp `GreenWhiteBackground1_AX` realisiert einen einfachen Zwei-Zustands-Automat für die Hintergrundfarbe (grün/weiß), abhängig vom aktuellen Wert am Eingang.

Anwendungsszenarien

- **Visualisierung von OPC-UA-Daten**: Anzeige von Maschinenzuständen (z.B. „Läuft“ = grün, „Stopp“ = weiß) in einem SCADA-System.
- **Echtzeit-Farbwechsel**: Dynamische Kennzeichnung von Prozesswerten basierend auf Schwellwerten oder booleschen Signalen.
- **Plug-and-Play-Komponente**: Einsatz in größeren Subapplikationen, bei denen eine einfache, generische OPC-UA-gesteuerte Hintergrundfärbung benötigt wird.

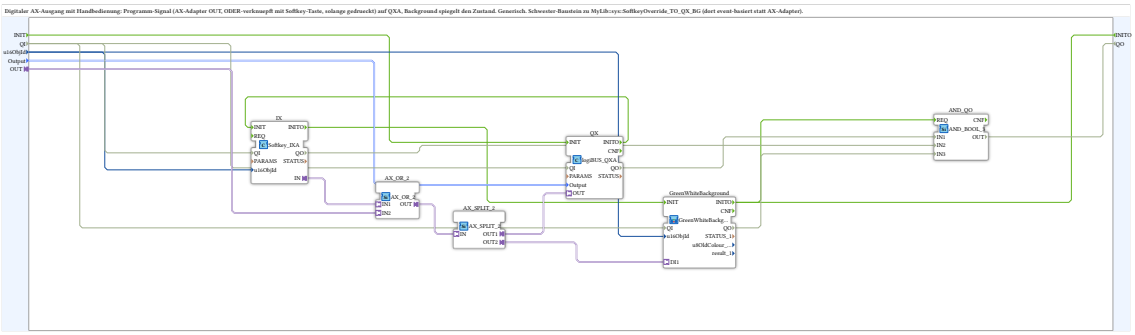
Vergleich mit ähnlichen Bausteinen

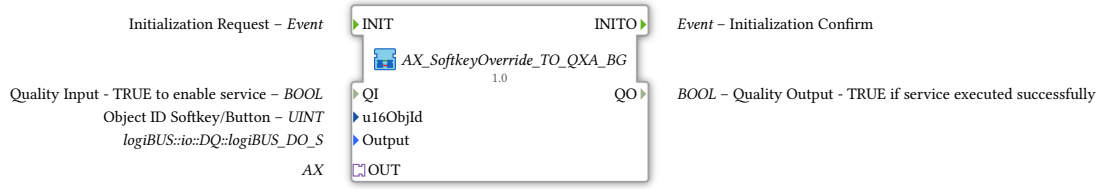
Im Vergleich zu direkt verdrahteten Subscribe-Bausteinen (z.B. einem einzelnen `AX_SUBSCRIBE`-Funktionsblock) bietet diese Subapp eine höhere Abstraktion: Sie bündelt Subscription und Farbsteuerung in einer Einheit, sodass der Anwender nur noch Adresse und Objekt-ID angeben muss. Gegenüber alternativen Lösungen, die separate Blöcke für OPC-UA-Lesen und Farbwechsel vorsehen, reduziert sie den Verdrahtungsaufwand und erhöht die Wiederverwendbarkeit. Einschränkung: Sie ist auf einen einzigen Kanal ausgelegt, während andere Bausteine evtl. mehrere Kanäle unterstützen.

Fazit

AX_SUBSCRIBE_BG_OPC ist eine kompakte Subapplikation, die OPC-UA-Datenabonnement und Visualisierungs-Hintergrundsteuerung nahtlos verbindet. Sie eignet sich hervorragend für Projekte, die eine schnelle und zuverlässige Farbkennzeichnung auf Basis von OPC-UA-Daten erfordern, ohne dass der Entwickler sich um die zugrunde liegende Kommunikation oder Zustandslogik kümmern muss. Durch die einfache Parametrierbarkeit und die klare Schnittstellenstruktur lässt sie sich unkompliziert in bestehende Anlagen integrieren.

AX_SoftkeyOverride_TO_QXA_BG





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein `AX_SoftkeyOverride_TO_QXA_BG` ist eine wiederverwendbare Subapp zur Steuerung eines digitalen Ausgangs (QXA) mit manueller Übersteuerung über eine Softkey-Taste. Das

Programm-Signal (AX-Adapter) wird mit dem Zustand der Taste ODER-verknüpft und auf den Ausgang gegeben. Ein integrierter Hintergrundbaustein visualisiert den aktuellen Zustand des Ausgangs. Der Baustein ist generisch aufgebaut und ermöglicht eine flexible Zuordnung von Objekt-IDs und Ausgangswerten.

Schnittstellenstruktur

Ereignis-Eingänge
- `INIT` (EInit): Initialisierungsanforderung; startet die Init-Kette bei `QX.INIT`.

Ereignis-Ausgänge
- `INITO` (EInit): Initialisierungsbestätigung; kommt von `GreenWhiteBackground.INITO`, dem Ende der Init-Kette.

Daten-Eingänge
- `QI` (BOOL): Quality Input, schaltet den Dienst frei (Initialwert: `TRUE`); wird an `QX.QI`, `IX.QI` und `GreenWhiteBackground.QI` weitergereicht.
- `u16ObjId` (UINT): Objekt-ID der Softkey-Taste bzw. des Buttons (Initialwert: `ID_NULL`).
- `Output` (logiBUS::io::DQ::logiBUS_DO_S): Ausgangswert, der an den digitalen Ausgang übergeben wird (Initialwert: `logiBUS_DO::Invalid`).

Daten-Ausgänge
- `QO` (BOOL): Quality Output, UND-Verknüpfung (`AND_QO`, `AND_BOOL_3`) der `QO`-Ausgänge von `QX`, `IX` und `GreenWhiteBackground`.

Adapter
- `OUT` (Socket, Typ `adapter::types::unidirectional::AX`): Eingangs-Adapter für das Programm-Signal (AX-Datenstrom).

Funktionsweise

Das über den Adapter `OUT` ankommende AX-Programmsignal wird zusammen mit dem Softkey-Zustand logisch ODER-verknüpft. Der Softkey-Zustand wird über den Baustein `Softkey_IXA` mit der übergebenen Objekt-ID `u16ObjId` ermittelt. Die ODER-Verknüpfung erfolgt über den Baustein `AX_OR_2`, der die beiden AX-Signale (Programm und Taste) zusammenführt.

Das Ergebnis der Verknüpfung wird anschließend über `AX_SPLIT_2` in zwei Pfade aufgeteilt:

- **Pfad 1** (OUT1) steuert den Ausgangsbaustein `logiBUS_QXA` an, der das digitale Signal auf dem QXA-Port setzt. Der Datenwert `Output` wird dabei als Ausgangswert übergeben.
- **Pfad 2** (OUT2) speist den Hintergrundbaustein `GreenWhiteBackground1_AX`, der den Ausgangszustand visuell darstellt (z. B. Mit Farbe oder Text) und ebenfalls die Objekt-ID `u16ObjId` erhält.

Initialisierung und Qualität: `INIT` → `QX.INIT`, `QX.INITO` → `IX.INIT`, `IX.INITO` → `GreenWhiteBackground.INIT`, `GreenWhiteBackground.INITO` → `INITO` und `AND_QO.REQ`. `QI` wird an `QX`, `IX` und `GreenWhiteBackground` weitergereicht; `QX.QO`, `IX.QO` und `GreenWhiteBackground.QO` werden im Baustein `AND_QO` (`iec61131::booleanOperators::AND_BOOL_3`) zum Ausgang `QO` verknüpft.

Durch diese Struktur wird eine einfache Handbedienung ermöglicht: Solange die Taste gedrückt ist oder das Programm-Signal aktiv ist, ist der Ausgang aktiv.

Technische Besonderheiten
- Verwendung von AX-Adapter-basierten Bausteinen (`AX_OR_2`, `AX_SPLIT_2`) zur Verarbeitung unidirektionaler AX-Signale – typisch für zyklische Signalverarbeitung ohne Event-Synchronisation.
- Der Baustein ist generisch und kann durch die Parameter `u16ObjId` und `Output` an verschiedene Anwendungen angepasst werden.
- Die Aufteilung des AX-Signals über `AX_SPLIT_2` ermöglicht die parallele Ausgabe an den eigentlichen Ausgang und an die Visualisierung (Hintergrund).
- Die Init-Kette (`INIT` → `QX` → `IX` → `GreenWhiteBackground` → `INITO`) und die UND-Verknüpfung der Qualitätsausgänge (`QI`/`QO`) folgen dem Standard-Schnittstellenmuster der Wrapper-Subapps.
- Die Subapp ist modular aufgebaut und kann in übergeordnete Projekte als wiederverwendbare Komponente eingebunden werden.

Zustandsübersicht

Der Baustein besitzt keinen eigenen Zustandsautomaten. Das Verhalten lässt sich jedoch logisch wie folgt beschreiben:

Zustand des AX-Eingangs	Softkey gedrückt	Ausgang (QXA)
inaktiv	nein	inaktiv
aktiv	nein	aktiv
inaktiv	ja	aktiv
aktiv	ja	aktiv

Der Hintergrund spiegelt den Zustand des Ausgangs entsprechend wider (z. B. Grün bei aktiv, Weiß bei inaktiv).

Anwendungsszenarien

- **Manuelles Übersteuern** eines digitalen Ausgangs in Maschinensteuerungen oder Prozessanlagen, z. B. für Testzwecke oder im Wartungsbetrieb.
- **Not- oder Freigabe-Taster**: Ein Softkey kann verwendet werden, um eine sicherheitsrelevante Ausgabe kurzfristig zu aktivieren.
- **Visualisierung und Fernbedienung**: Der Hintergrundbaustein dient zur Zustandsanzeige auf Bedienpanels oder in SCADA-Systemen.

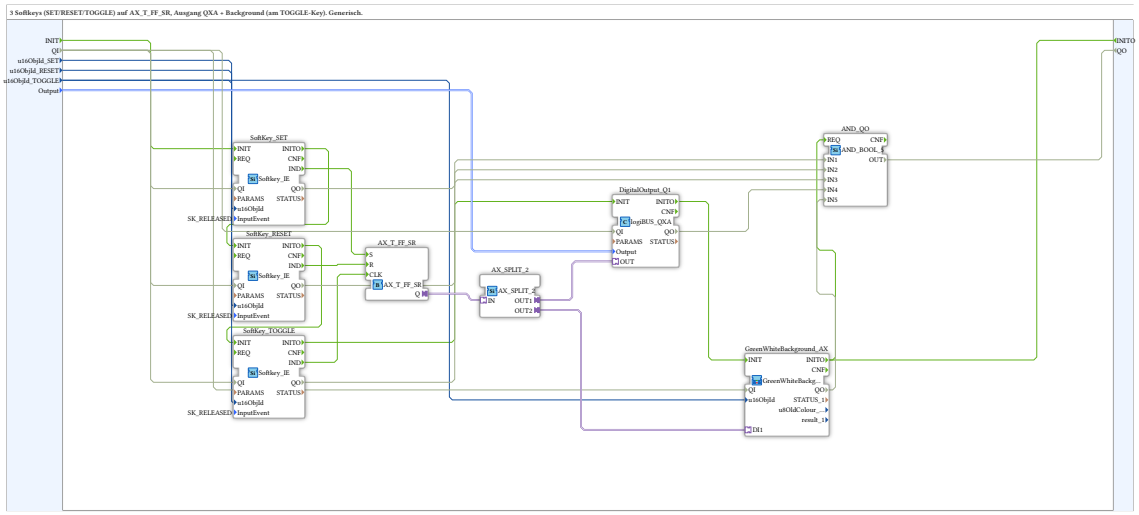
Vergleich mit ähnlichen Bausteinen

Der Baustein `AX_SoftkeyOverride_TO_QXA_BG` ist ein Schwester-Baustein zu `MyLib::sys::SoftkeyOverride_TO_QX_BG`. Letzterer arbeitet event-basiert (über Event-Eingänge) statt mit AX-Adaptern. Während der event-basierte Ansatz eher für ereignisgesteuerte Abläufe geeignet ist, eignet sich der AX-basierte Baustein besser für zyklische Verarbeitung oder wenn bereits AX-Signale im System verwendet werden. Beide erfüllen die gleiche grundlegende Funktion: ODER-Verknüpfung von Programm- und Tastersignal auf einen digitalen Ausgang. Die Wahl hängt von der vorhandenen Signalarchitektur ab.

Fazit

`AX_SoftkeyOverride_TO_QXA_BG` bietet eine robuste und flexible Lösung, um digitale Ausgänge manuell zu übersteuern und gleichzeitig den Zustand visuell darzustellen. Durch die Verwendung von AX-Adaptern lässt sich der Baustein nahtlos in moderne, zyklische Steuerungsarchitekturen integrieren. Seine generische Parametrierung und die modulare Bauweise machen ihn zu einem wertvollen Bestandteil in der Automatisierungstechnik, insbesondere wenn eine Mischung aus automatischem und manuellem Betrieb gewünscht ist.

AX_SoftkeySR_TO_QXA_BG



• – ◦ • – ◦ • – ◦ •

Einleitung

Die Subapplikation **AX_SoftkeySR_TO_QXA_BG** ermöglicht die Steuerung eines digitalen Ausgangs (QXA) über drei Softkeys. Sie kombiniert die Funktionen *SET*, *RESET* und *TOGGLE* unter Verwendung eines SR-Flip-Flops und eines Signal-Splitters. Zusätzlich wird ein Hintergrund-Submodul angesteuert, das bei jedem Toggle-Schritt den Zustand des Hintergrunds wechselt. Die Bausteine sind generisch konfigurierbar und für eine Wiederverwendung in verschiedenen Applikationen ausgelegt.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; startet die Initialisierungskette der inneren Bausteine.

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierung abgeschlossen; kommt von `GreenWhiteBackground_AX.INITO`, dem letzten Glied der Kette.

Daten-Eingänge

Name	Typ	Initialwert	Kommentar
`QI`	`BOOL`	`TRUE`	Quality Input - TRUE aktiviert den Dienst; wird an alle inneren Bausteine verteilt.
`u16ObjId_SET`	`UINT`	`ID_NULL`	Objekt-ID des Softkeys für die SET-Funktion.
`u16ObjId_RESET`	`UINT`	`ID_NULL`	Objekt-ID des Softkeys für die RESET-Funktion.
`u16ObjId_TOGGLE`	`UINT`	`ID_NULL`	Objekt-ID des Softkeys für die TOGGLE-Funktion (auch für Hintergrund).
`Output`	`logiBUS::io::DQ::logiBUS_DO_S`	`Invalid`	Ausgangskanal (z. B. `Output_Q1` … `Output_Q8`) für die digitale Ausgabe.

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde (AND der `QO` aller fünf inneren Instanzen).

Adapter

Keine (interne Adapter werden verwendet, aber nicht nach außen geführt).

Funktionsweise

Die Subapplikation besitzt drei interne Softkey-Eingangsbausteine (`SoftKey_SET`, `SoftKey_RESET`, `SoftKey_TOGGLE`), die jeweils einen `Softkey_IE`-Funktionsbaustein repräsentieren. Diese Bausteine werden über die externen Objekt-IDs konfiguriert. Die Ereignisse `IND` der Softkeys werden auf ein SR-Flip-Flop (`AX_T_FF_SR`) geleitet:

- **SET** ⟶ setzt den Ausgang `Q` auf TRUE,

- **RESET** ⟶ setzt den Ausgang `Q` auf FALSE,
- **TOGGLE** ⟶ invertiert den aktuellen Zustand von `Q`.

Der Ausgang `Q` wird über den Adapter `AX_SPLIT_2` auf zwei Pfade aufgeteilt:

1. **Digitaler Ausgang** (`logiBUS_QXA`): Der Wert wird an den über `Output` spezifizierten Kanal weitergegeben.
2. **Hintergrund-Submodul** (`GreenWhiteBackground1_AX`): Dieses erhält zusätzlich die Objekt-ID des TOGGLE-Softkeys, um bei jedem Toggle den Hintergrundzustand (z. B. grün/weiß) zu wechseln.

Initialisierung und Qualität: `INIT` ⟶ `SoftKey_SET.INIT`, `SoftKey_SET.INITO` ⟶ `SoftKey_RESET.INIT`, `SoftKey_RESET.INITO` ⟶ `SoftKey_TOGGLE.INIT`, `SoftKey_TOGGLE.INITO` ⟶ `DigitalOutput_Q1.INIT`, `DigitalOutput_Q1.INITO` ⟶ `GreenWhiteBackground_AX.INIT` und `GreenWhiteBackground_AX.INITO` ⟶ `INITO`. `QI` geht an `SoftKey_SET`, `SoftKey_RESET`, `SoftKey_TOGGLE`, `DigitalOutput_Q1` und `GreenWhiteBackground_AX`. `QO` ist der Ausgang der Instanz `AND_QO` (`iec61131::booleanOperators::AND_BOOL_5`), die die `QO` dieser fünf Instanzen verknüpft; ihr `REQ` wird von `GreenWhiteBackground_AX.INITO` ausgelöst.

Die Verbindungen zwischen den Bausteinen sind in der SubApp fest verdrahtet, sodass die gesamte Logik durch die externen Parameter konfiguriert werden kann.

Technische Besonderheiten

- **QI/INIT/INITO/QO:** `QI` ist nicht mehr intern fest auf `TRUE` gesetzt, sondern wird von außen an alle inneren Bausteine durchgereicht; die Initialisierung läuft nacheinander durch die Kette von `INIT` bis `INITO`.
- **Generische Konfiguration:** Die Objekt-IDs und der Ausgangskanal sind als generische Eingänge definiert. Standardmäßig sind sie auf `ID_NULL` bzw. `Invalid` gesetzt, wodurch die SubApp ohne Anpassung keine aktive Funktion besitzt. Erst durch Zuweisung gültiger Werte wird die Steuerung aktiv.
- **Wiederverwendbarkeit:** Die SubApp ist aus einem Übungsprojekt ausgelagert und kann in verschiedenen Anwendungen eingesetzt werden, ohne Änderungen an der internen Logik vorzunehmen.
- **Verwendete Typen:** Die Eingänge nutzen `UINT` und den spezifischen Typ `logiBUS_DO_S` aus der logiBUS-Bibliothek. Intern werden Adapter und Funktionsbausteine aus den Bibliotheken `isobus` und `adapter` verwendet.
- **Signalaufteilung:** Der Splitter `AX_SPLIT_2` ermöglicht die parallele Nutzung eines einzigen Ausgangssignals für zwei Ziele.

Zustandsübersicht

Die SubApp besitzt einen internen binären Zustand, der durch das SR-Flip-Flop `AX_T_FF_SR` gehalten wird. Die möglichen Zustände sind:

- **Zustand 0 (Reset):** Ausgang `Q` = FALSE.
  Übergänge:
  - SET-Event ⟶ Zustand 1
  - TOGGLE-Event ⟶ Zustand 1
  - RESET-Event bleibt in Zustand 0
- **Zustand 1 (Set):** Ausgang `Q` = TRUE.
  Übergänge:
  - RESET-Event ⟶ Zustand 0
  - TOGGLE-Event ⟶ Zustand 0
  - SET-Event bleibt in Zustand 1

Der Ausgang `Q` wird in beide Richtungen über den Splitter an die nachgelagerten Bausteine (Ausgang und Hintergrund) verteilt.

Anwendungsszenarien

- **Maschinensteuerung:** Ein einzelner Ausgang (z. B. Licht, Ventil) kann über drei verschiedene Softkeys ein- (SET), aus- (RESET) und umgeschaltet (TOGGLE) werden.
- **Visualisierung:** Die TOGGLE-Funktion kann gleichzeitig einen Hintergrundfarbwechsel (z. B. grün/weiß) an einem Bedienpanel auslösen, um den aktuellen Zustand visuell darzustellen.
- **Flexible Zuordnung:** Durch die parametrierbaren Objekt-IDs können die Softkeys je nach Station oder Bedienkontext dynamisch zugewiesen werden.

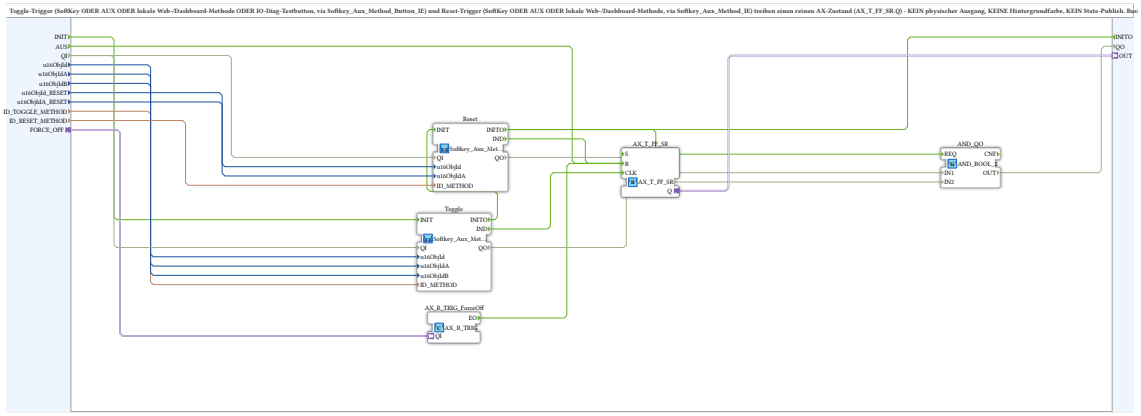
Vergleich mit ähnlichen Bausteinen

- **Einfacher SR-Flip-Flop:** Im Gegensatz zu einem nackten SR-Flip-Flop integriert diese SubApp bereits die Softkey-Anbindung und die Ausgangsansteuerung, wodurch weniger externe Verdrahtung nötig ist.
- **Einzel-Softkey-Schalter:** Bausteine, die nur einen einzelnen Softkey verwenden, besitzen keine SET/RESET/Toggle-Kombination. Diese SubApp bietet eine vollständige Drei-Tasten-Bedienung.
- **Mit Hintergrundsteuerung:** Viele ähnliche Bausteine steuern nur einen Ausgang, ohne die zusätzliche Hintergrundlogik. Die Einbindung von `GreenWhiteBackground1_AX` erweitert den Funktionsumfang.

Fazit

Die SubApplikation **AX_SoftkeySR_TO_QXA_BG** stellt eine kompakte und wiederverwendbare Lösung zur Steuerung eines digitalen Ausgangs über drei Softkeys dar. Sie vereint SR-Flip-Flop-Logik, Ausgangstreiber und eine optionale Hintergrundumschaltung in einem einzigen Modul. Durch die generischen Parameter ist sie flexibel an verschiedene Objekt-IDs und Ausgangskanäle anpassbar und eignet sich daher für vielfältige Anwendungen in der Automatisierungstechnik und Visualisierung.

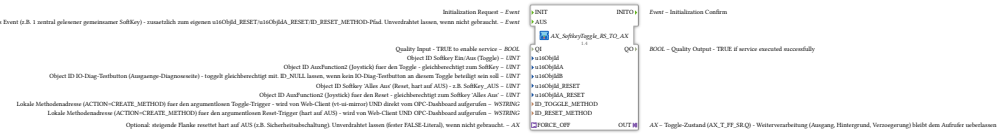
AX_SoftkeyToggle_RS_TO_AX



• – ○ • – ○ • – ○ •

Einleitung

`AX_SoftkeyToggle_RS_TO_AX` erzeugt einen reinen AX-Zustand aus einem Toggle-Trigger und einem Reset-Trigger. Der Toggle-Trigger kommt von SoftKey, AUX, einer lokalen Web-/Dashboard-Methode oder einem IO-Diag-Testbutton (über `Softkey_Aux_Method_Button_IE`), der Reset-Trigger von SoftKey, AUX oder einer lokalen Methode (über `Softkey_Aux_Method_IE`). Beide treiben ein `AX_T_FF_SR`, dessen Zustand am Plug `OUT` anliegt. Es gibt **keinen** physischen Ausgang, **keine** Hintergrundfarbe und **keinen** State-Publish; das bleibt dem Aufrufer überlassen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung
- **AUS**: setzt den Zustand hart auf AUS zurück, von außen als Ereignis (z. B. ein zentral gelesener gemeinsamer SoftKey ‚Alles Aus‘); zusätzlich zum eigenen Reset-Pfad über `u16ObjId_RESET`/`u16ObjIdA_RESET`/`ID_RESET_METHOD`. Unverdrahtet lassen, wenn nicht gebraucht.

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung

Daten-Eingänge

- **QI** (BOOL): Quality Input, TRUE gibt den Dienst frei (Default `TRUE`)

- **u16ObjId** (UINT): Object ID des SoftKeys Ein/Aus (Toggle), Default `ID_NULL`
- **u16ObjIdA** (UINT): Object ID der AuxFunction2 (Joystick) für den Toggle, gleichberechtigt zum SoftKey
- **u16ObjIdB** (UINT): Object ID des IO-Diag-Testbuttons (Ausgänge-Diagnoseseite), toggelt gleichberechtigt mit; `ID_NULL` lassen, wenn kein solcher Button beteiligt ist
- **u16ObjId_RESET** (UINT): Object ID des SoftKeys ‚Alles Aus' (Reset, hart auf AUS)
- **u16ObjIdA_RESET** (UINT): Object ID der AuxFunction2 (Joystick) für den Reset, gleichberechtigt zum SoftKey ‚Alles Aus'
- **ID_TOGGLE_METHOD** (WSTRING): lokale Methodenadresse (`ACTION=CREATE_METHOD`) für den argumentlosen Toggle-Trigger; wird vom Web-Client (vt-ui-mirror) und direkt vom OPC-Dashboard aufgerufen
- **ID_RESET_METHOD** (WSTRING): lokale Methodenadresse (`ACTION=CREATE_METHOD`) für den argumentlosen Reset-Trigger (hart auf AUS); wird vom Web-Client und vom OPC-Dashboard aufgerufen

Daten-Ausgänge

- **QO** (BOOL): Quality Output, UND aus `Toggle.QO` und `Reset.QO`

Adapter

- **OUT** (Plug, `adapter::types::unidirectional::AX`): Toggle-Zustand (`AX_T_FF_SR.Q`); Weiterverarbeitung (Ausgang, Hintergrund, Verzögerung) bleibt dem Aufrufer überlassen
- **FORCE_OFF** (Socket, `adapter::types::unidirectional::AX`): optional; eine steigende Flanke setzt hart auf AUS zurück (z. B. Sicherheitsabschaltung). Unverdrahtet bzw. auf festes FALSE lassen, wenn nicht gebraucht

Funktionsweise

Das Netzwerk enthält fünf Teile:

- `Toggle` (`MyLib::sys::Softkey_Aux_Method_Button_IE`): liest `u16ObjId`, `u16ObjIdA`, `u16ObjIdB` und `ID_TOGGLE_METHOD` (an `Toggle.ID_METHOD`); sein `IND` führt auf `AX_T_FF_SR.CLK` und schaltet den Zustand um.
- `Reset` (`MyLib::sys::Softkey_Aux_Method_IE`): liest `u16ObjId_RESET` (an `Reset.u16ObjId`), `u16ObjIdA_RESET` (an `Reset.u16ObjIdA`) und `ID_RESET_METHOD`; sein `IND` führt auf `AX_T_FF_SR.R`.
- `AX_T_FF_SR` (`adapter::events::unidirectional::AX_T_FF_SR`): Toggle-Flipflop mit Set/Reset; `AX_T_FF_SR.Q` ⟶ `OUT`.
- `AX_R_TRIG_ForceOff` (`adapter::events::unidirectional::AX_R_TRIG`): erkennt die steigende Flanke an `FORCE_OFF` (`FORCE_OFF` ⟶ `AX_R_TRIG_ForceOff.QI`); sein `EO` führt ebenfalls auf `AX_T_FF_SR.R`.
- `AND_QO` (`iec61131::booleanOperators::AND_BOOL_2`): verknüpft `Toggle.QO` (`IN1`) und `Reset.QO` (`IN2`) zum `QO` der SubApp.

Das Ereignis `AUS` ist ein dritter, additiver Weg auf `AX_T_FF_SR.R`.

Initialisierung und Qualität: `INIT` ⟶ `Toggle.INIT`; `Toggle.INITO` ⟶ `Reset.INIT`; `Reset.INITO` ⟶ `INITO` (und ⟶ `AND_QO.REQ`). `QI` wird an `Toggle.QI` und `Reset.QI` verteilt.

Technische Besonderheiten

- Basis-Baustein: Für **einen** Ausgang gibt es `MyLib::sys::AX_SoftkeyToggle_RS_TO_QXA_BG`, das diesen Baustein kapselt. Für mehrere zeitversetzte Ausgänge (z. B. Bänder-Sequenzierung) wird dieser Baustein direkt verwendet und der `OUT`-Zustand pro Kanal mit `AX_TON`/`AX_TOF` weiterverarbeitet
- Drei gleichberechtigte Reset-Wege (VT/AUX/Methode, Ereignis `AUS`, Flanke an `FORCE_OFF`) laufen alle auf `AX_T_FF_SR.R`
- Der Reset-Eingang heißt `AUS`, passend zum zentralen ‚Alles Aus'-Softkey, den mehrere Toggle-Instanzen gemeinsam verteilen

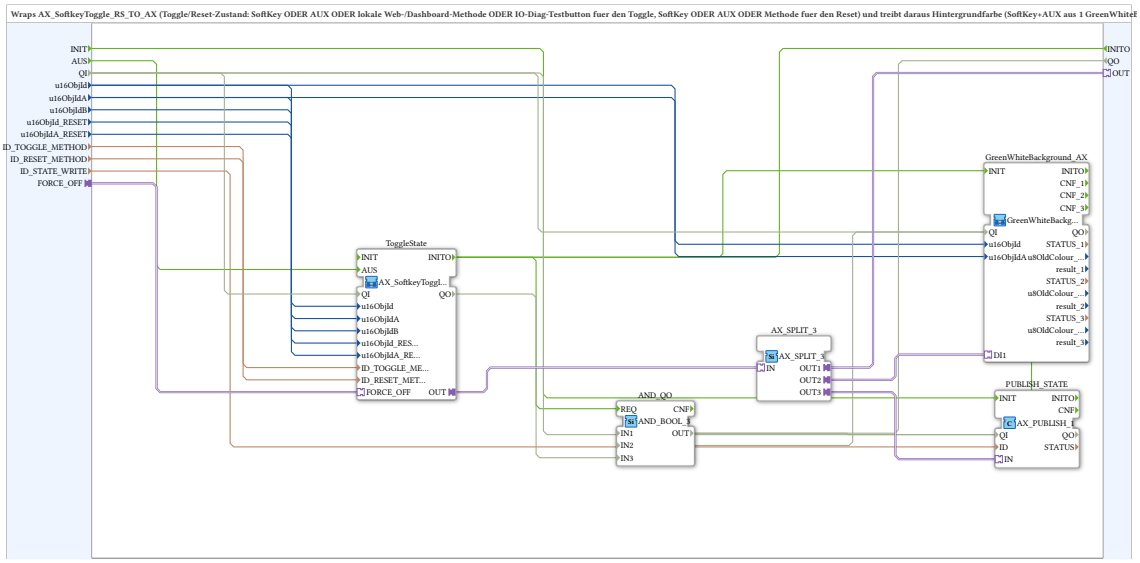
Anwendungsszenarien

- Ein-/Ausschalter für Antriebe, der gleichermaßen per SoftKey, Joystick-Zuweisung, Web/Dashboard und Diagnose-Button bedient wird
- Zentrales ‚Alles Aus' für mehrere Toggle-Instanzen über das Ereignis `AUS`
- Sicherheitsabschaltung über `FORCE_OFF` (z. B. Zapfwellen-Überwachung)

Fazit

`AX_SoftkeyToggle_RS_TO_AX` liefert den reinen Toggle-Zustand mit mehreren gleichberechtigten Bedien- und Reset-Wegen und überlässt Ausgang, Hintergrundfarbe und Publish dem Aufrufer.

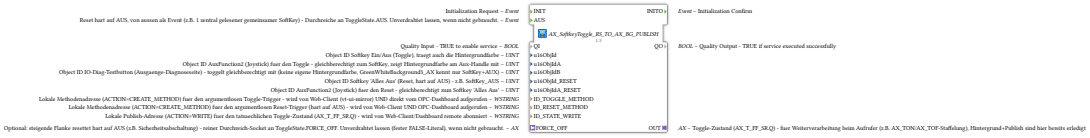
AX_SoftkeyToggle_RS_TO_AX_BG_PUBLISH



• – ○ • – ○ • – ○ •

Einleitung

`AX_SoftkeyToggle_RS_TO_AX_BG_PUBLISH` kapselt `AX_SoftkeyToggle_RS_TO_AX` (Toggle-/Reset-Zustand) und treibt daraus die Hintergrundfarbe sowie einen lokalen State-Publish, **ohne** physischen Ausgang. Der Toggle wird ausgelöst durch SoftKey, AUX, lokale Web-/Dashboard-Methode oder IO-Diag-Testbutton; der Reset durch SoftKey, AUX oder Methode. Die Hintergrundfarbe (SoftKey und AUX) kommt aus einer gemeinsamen `GreenWhiteBackground3_AX`. Der Baustein ist das Gegenstück zu `AX_SoftkeyToggle_RS_TO_QXA_BG`, das zusätzlich noch einen `logiBUS_QXA` treibt. Er ist für Fälle gedacht, in denen der Aufrufer den Toggle-Zustand weiterverarbeitet, bevor er auf einen oder mehrere physische Ausgänge geht (z. B. Bänder-Sequenzierung: Taste 5 toggelt, danach erst `AX_TON`/`AX_TOF` je Kanal). Der Toggle-Zustand bleibt zusätzlich als Plug `OUT` verfügbar.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialization Request (Kette: `PUBLISH_STATE` → `GreenWhiteBackground_AX` → `ToggleState`)
- **AUS**: Reset hart auf AUS, von außen als Event (z. B. ein zentral gelesener gemeinsamer SoftKey). Durchreiche an `ToggleState.AUS`. Unverdrahtet lassen, wenn nicht gebraucht.

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialization Confirm (von `ToggleState.INITO`)

Daten-Eingänge

- **QI** (`BOOL`): Quality Input, TRUE aktiviert den Dienst (Default `TRUE`)
- **u16ObjId** (`UINT`): Object ID Softkey Ein/Aus (Toggle), trägt auch die Hintergrundfarbe (Default `ID_NULL`)
- **u16ObjIdA** (`UINT`): Object ID AuxFunction2 (Joystick) für den Toggle, gleichberechtigt zum SoftKey, zeigt die Hintergrundfarbe am Aux-Handle mit (Default `ID_NULL`)

- **u16ObjIdB** (UINT): Object ID IO-Diag-Testbutton (Ausgänge-Diagnoseseite), toggelt gleichberechtigt mit; keine eigene Hintergrundfarbe, da GreenWhiteBackground3_AX nur SoftKey und AUX kennt (Default ID_NULL)
- **u16ObjId_RESET** (UINT): Object ID Softkey „Alles Aus“ (Reset, hart auf AUS), z. B. SoftKey_AUS (Default ID_NULL)
- **u16ObjIdA_RESET** (UINT): Object ID AuxFunction2 (Joystick) für den Reset, gleichberechtigt zum Softkey „Alles Aus“ (Default ID_NULL)
- **ID_TOGGLE_METHOD** (WSTRING): lokale Methodenadresse (ACTION=CREATE_METHOD) für den argumentlosen Toggle-Trigger, wird vom Web-Client (vt-ui-mirror) und direkt vom OPC-Dashboard aufgerufen
- **ID_RESET_METHOD** (WSTRING): lokale Methodenadresse (ACTION=CREATE_METHOD) für den argumentlosen Reset-Trigger (hart auf AUS), wird von Web-Client und OPC-Dashboard aufgerufen
- **ID_STATE_WRITE** (WSTRING): lokale Publish-Adresse (ACTION=WRITE) für den tatsächlichen Toggle-Zustand (AX_T_FF_SR.Q), wird von Web-Client/Dashboard remote abonniert

Daten-Ausgänge

- **QO** (BOOL): Quality Output, UND-Verknüpfung (AND_QO) der QO-Ausgänge von PUBLISH_STATE, GreenWhiteBackground_AX und ToggleState

Adapter

- **FORCE_OFF** (Socket, adapter::types::unidirectional::AX): optional; eine steigende Flanke resettet hart auf AUS (z. B. Sicherheitsabschaltung). Reiner Durchreich-Socket an ToggleState.FORCE_OFF. Unverdrahtet lassen (fester FALSE-Literal), wenn nicht gebraucht.
- **OUT** (Plug, adapter::types::unidirectional::AX): Toggle-Zustand (AX_T_FF_SR.Q) zur Weiterverarbeitung beim Aufrufer; Hintergrund und Publish sind hier bereits erledigt

Verwendete Funktionsbausteine (FBs)

- **ToggleState** (SubApp MyLib::sys::AX_SoftkeyToggle_RS_TO_AX): erzeugt den Toggle-/Reset-Zustand
- **AX_SPLIT_3** (adapter::events::unidirectional::AX_SPLIT_3): verteilt den Zustand auf drei Ziele
- **GreenWhiteBackground_AX** (SubApp MyLib::sys::GreenWhiteBackground3_AX): Hintergrundfarbe grün/weiß
- **PUBLISH_STATE** (adapter::net::AX_PUBLISH_1): publiziert den Zustand (QI kommt vom Wrapper-Eingang QI)
- **AND_QO** (iec61131::booleanOperators::AND_BOOL_3): verknüpft die QO-Ausgänge der drei inneren Bausteine zu QO

Funktionsweise

- AUS ⟶ ToggleState.AUS; FORCE_OFF ⟶ ToggleState.FORCE_OFF
- u16ObjId, u16ObjIdA, u16ObjIdB, u16ObjId_RESET, u16ObjIdA_RESET, ID_TOGGLE_METHOD, ID_RESET_METHOD ⟶ gleichnamige Eingänge von ToggleState
- ToggleState.OUT ⟶ AX_SPLIT_3.IN
- AX_SPLIT_3.OUT1 ⟶ OUT
- AX_SPLIT_3.OUT2 ⟶ GreenWhiteBackground_AX.DI1; u16ObjId, u16ObjIdA ⟶ GreenWhiteBackground_AX
- AX_SPLIT_3.OUT3 ⟶ PUBLISH_STATE.IN; ID_STATE_WRITE ⟶ PUBLISH_STATE.ID
- Initialisierung: INIT ⟶ PUBLISH_STATE.INIT; PUBLISH_STATE.INITO ⟶ GreenWhiteBackground_AX.INIT; GreenWhiteBackground_AX.INITO ⟶ ToggleState.INIT; ToggleState.INITO ⟶ INITO und ⟶ AND_QO.REQ
- Qualität: QI ⟶ PUBLISH_STATE.QI, GreenWhiteBackground_AX.QI, ToggleState.QI; PUBLISH_STATE.QO ⟶ AND_QO.IN1, GreenWhiteBackground_AX.QO ⟶ AND_QO.IN2, ToggleState.QO ⟶ AND_QO.IN3; AND_QO.OUT ⟶ QO

Technische Besonderheiten

- Herausgelöst aus Baender_EinAus (ToggleState/AX_SPLIT_3-Anteil für Hintergrund und Publish, GreenWhiteBackground_AX, PUBLISH_STATE). Die zeitversetzten Ausgänge (TOF_IgelElevator/TON_Querband-Staffelung) bleiben beim Aufrufer.
- Versionsverlauf: FORCE_OFF (v1.1, für übergeordnete Sicherheitsabschaltung), Durchreiche RESET (v1.2), Umbenennung des Eingangs RESET in AUS (v1.3).

- Importiert `isobus::UT::Q::const::IDs::ID_NULL` als Default der Objekt-IDs.

Anwendungsszenarien

- Ein-/Aus-Toggle mit Farbanzeige und Web-Publish, dessen Zustand vor dem physischen Ausgang weiterverarbeitet wird (z. B. gestaffelte Bänder)

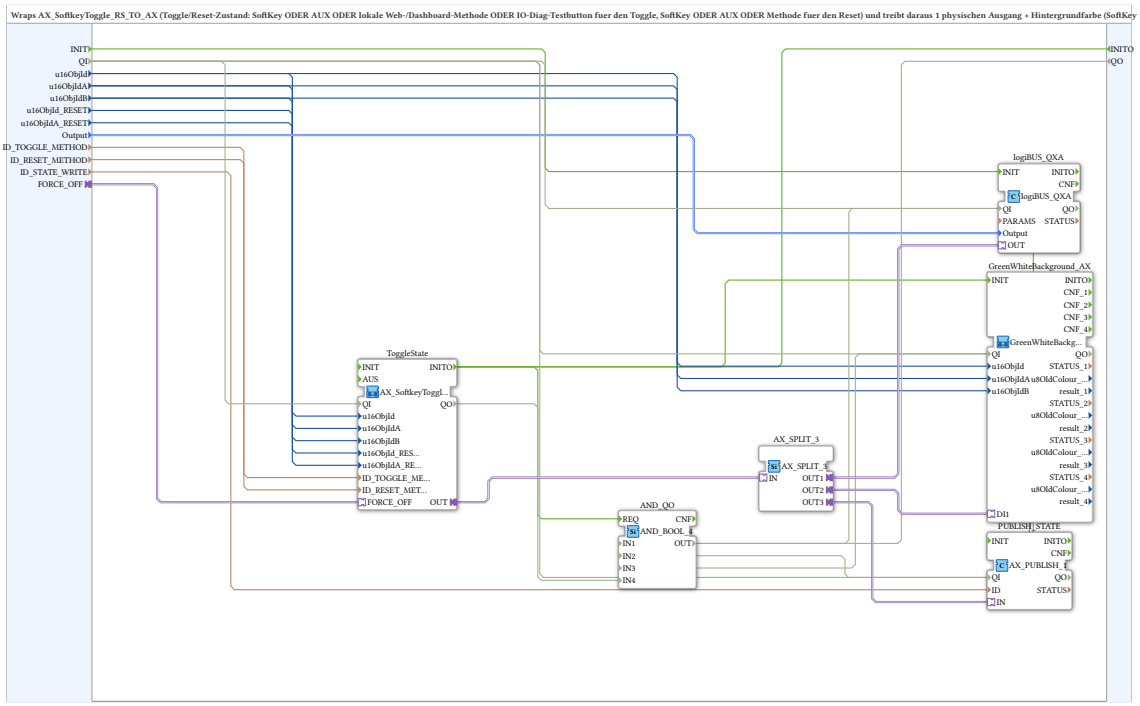
Zusammenfassung

`AX_SoftkeyToggle_RS_TO_AX_BG_PUBLISH` liefert einen Toggle-/Reset-Zustand mit Hintergrundfarbe und State-Publish und gibt den Zustand als `AX`-Plug zur Weiterverarbeitung aus.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AX_SoftkeyToggle_RS_TO_QXA_BG



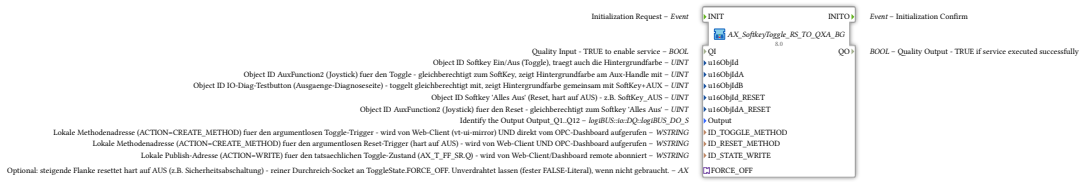
• — ○ • — ○ • — ○ •

Einleitung

`AX_SoftkeyToggle_RS_TO_QXA_BG` kapselt `AX_SoftkeyToggle_RS_TO_AX` (Toggle-/Reset-Zustand) und treibt daraus **einen physischen Ausgang**, die **Hintergrundfarbe** und einen **lokalen State-Publish**:

- Toggle: SoftKey **oder** AUX **oder** lokale Web-/Dashboard-Methode **oder** IO-Diag-Testbutton
- Reset: SoftKey **oder** AUX **oder** Methode
- Hintergrundfarbe: SoftKey, AUX-Handle und IO-Diag-Testbutton gemeinsam aus einer `GreenWhiteBackground4_AX`
- Publish: Zustand für Web-Client/OPC-Dashboard-Hintergrundfarbe

Generisch für jeden Ein/Aus-Ausgang, der von einer globalen Sammel-AUS-Taste mit zurückgesetzt werden muss (z. B. Sägeblattantrieb).



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: AX_SoftkeyToggle_RS_TO_QXA_BG

- **Typ**: SubAppType
- **Verwendete interne Bausteine**:
  - **ToggleState**: SubApp `MyLib::sys::AX_SoftkeyToggle_RS_TO_AX` – Toggle-/Reset-Zustand (inkl. `FORCE_OFF`)
  - **AX_SPLIT_3**: `adapter::events::unidirectional::AX_SPLIT_3` – verteilt den Zustand auf drei Verbraucher
  - **logiBUS_QXA**: `logiBUS::io::DQ::logiBUS_QXA` (`QI` vom Baustein-Eingang `QI`) – physischer Ausgang
  - **GreenWhiteBackground_AX**: SubApp `MyLib::sys::GreenWhiteBackground4_AX` – Hintergrundfarbe für SoftKey, AUX und Testbutton
  - **PUBLISH_STATE**: `adapter::net::AX_PUBLISH_1` (`QI` vom Baustein-Eingang `QI`) – lokaler State-Publish
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_4` – verknüpft die `QO`-Ausgänge von `logiBUS_QXA`, `PUBLISH_STATE`, `GreenWhiteBackground_AX` und `ToggleState` per UND zum Baustein-Ausgang `QO`
- **Importe**: `ID_NULL`, `logiBUS_DO`, `logiBUS_DO_S`

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialization Request

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialization Confirm

Die Signalverarbeitung (Toggle/Reset) läuft weiterhin über die Adapter.

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input – TRUE, um den Dienst freizugeben
- **u16ObjId** (`UINT`, Vorgabe `ID_NULL`): Object-ID Softkey Ein/Aus (Toggle), trägt auch die Hintergrundfarbe
- **u16ObjIdA** (`UINT`, Vorgabe `ID_NULL`): Object-ID `AuxFunction2` (Joystick) für den Toggle – gleichberechtigt zum SoftKey, zeigt die Hintergrundfarbe am Aux-Handle mit
- **u16ObjIdB** (`UINT`, Vorgabe `ID_NULL`): Object-ID IO-Diag-Testbutton (Ausgänge-Diagnoseseite) – toggelt gleichberechtigt mit, zeigt die Hintergrundfarbe gemeinsam mit SoftKey und AUX
- **u16ObjId_RESET** (`UINT`, Vorgabe `ID_NULL`): Object-ID Softkey „Alles Aus“ (Reset, hart auf AUS), z. B. `SoftKey_AUS`
- **u16ObjIdA_RESET** (`UINT`, Vorgabe `ID_NULL`): Object-ID `AuxFunction2` (Joystick) für den Reset – gleichberechtigt zum Softkey „Alles Aus“
- **Output** (`logiBUS::io::DQ::logiBUS_DO_S`, Vorgabe `logiBUS_DO::Invalid`): Ausgang `Output_Q1..Q12`
- **ID_TOGGLE_METHOD** (`WSTRING`): Lokale Methodenadresse (`ACTION=CREATE_METHOD`) für den argumentlosen Toggle-Trigger – wird vom Web-Client (`vt-ui-mirror`) und direkt vom OPC-Dashboard aufgerufen
- **ID_RESET_METHOD** (`WSTRING`): Lokale Methodenadresse (`ACTION=CREATE_METHOD`) für den argumentlosen Reset-Trigger (hart auf AUS) – vom Web-Client und OPC-Dashboard aufgerufen
- **ID_STATE_WRITE** (`WSTRING`): Lokale Publish-Adresse (`ACTION=WRITE`) für den tatsächlichen Toggle-Zustand (`AX_T_FF_SR.Q`) – vom Web-Client/Dashboard remote abonniert

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde (UND der `QO` der inneren Bausteine)

Adapter

- **FORCE_OFF** (Socket, `adapter::types::unidirectional::AX`): Optional: steigende Flanke resettet hart auf AUS (z. B. Sicherheitsabschaltung) – reiner Durchreich-Socket an `ToggleState.FORCE_OFF`. Unverdrahtet lassen (fester FALSE-Literal), wenn nicht gebraucht.

Programmablauf und Verbindungen

- `u16ObjId`, `u16ObjIdA`, `u16ObjIdB`, `u16ObjId_RESET`, `u16ObjIdA_RESET`, `ID_TOGGLE_METHOD`, `ID_RESET_METHOD` ⟶ gleichnamige Eingänge von `ToggleState`
- `FORCE_OFF` ⟶ `ToggleState.FORCE_OFF`
- `ToggleState.OUT` ⟶ `AX_SPLIT_3.IN`
- `AX_SPLIT_3.OUT1` ⟶ `logiBUS_QXA.OUT` (`Output` ⟶ `logiBUS_QXA.Output`)
- `AX_SPLIT_3.OUT2` ⟶ `GreenWhiteBackground_AX.DI1` (`u16ObjId`, `u16ObjIdA`, `u16ObjIdB` ⟶ gleichnamige Eingänge)
- `AX_SPLIT_3.OUT3` ⟶ `PUBLISH_STATE.IN` (`ID_STATE_WRITE` ⟶ `PUBLISH_STATE.ID`)
- Initialisierungskette: `INIT` ⟶ `logiBUS_QXA.INIT`, `logiBUS_QXA.INITO` ⟶ `PUBLISH_STATE.INIT`, `PUBLISH_STATE.INITO` ⟶ `GreenWhiteBackground_AX.INIT`, `GreenWhiteBackground_AX.INITO` ⟶ `ToggleState.INIT`, `ToggleState.INITO` ⟶ `INITO`; zusätzlich `ToggleState.INITO` ⟶ `AND_QO.REQ`
- `QI` ⟶ `logiBUS_QXA.QI`, `PUBLISH_STATE.QI`, `GreenWhiteBackground_AX.QI` und `ToggleState.QI`
- `logiBUS_QXA.QO` ⟶ `AND_QO.IN1`, `PUBLISH_STATE.QO` ⟶ `AND_QO.IN2`, `GreenWhiteBackground_AX.QO` ⟶ `AND_QO.IN3`, `ToggleState.QO` ⟶ `AND_QO.IN4`; `AND_QO.OUT` ⟶ `QO`

Technische Besonderheiten

- Seit Version 7.0 nur noch Wrapper: Toggle, Reset und `AX_T_FF_SR` stecken in `AX_SoftkeyToggle_RS_TO_AX`, das auch für Mehrkanal-Sequenzierung (mehrere zeitversetzte Ausgänge) verwendet wird
- Seit Version 6.0: ein `GreenWhiteBackground4_AX` statt zweier `GreenWhiteBackground1_AX`, damit auch das Joystick-Handle eine Rückmeldung bekommt
- Version 8.0 (2026-09-15): `FORCE_OFF`-Socket ergänzt (rein additiv; bestehende Instanzen müssen ihn verdrahten, z. B. auf einen festen FALSE-Literal)
- Historie: 1.0 Reset-Set-Toggle-Variante von `AX_SoftkeyToggle_TO_QXA_BG`; 2.0 IO-Diag-Testbutton, zwei lokale RPC-Methoden und State-Publish; 3.0 AUX ergänzt; 4.0/5.0 Auslagerung in `Softkey_Aux_Method_IE` bzw. `Softkey_Aux_Method_Button_IE`

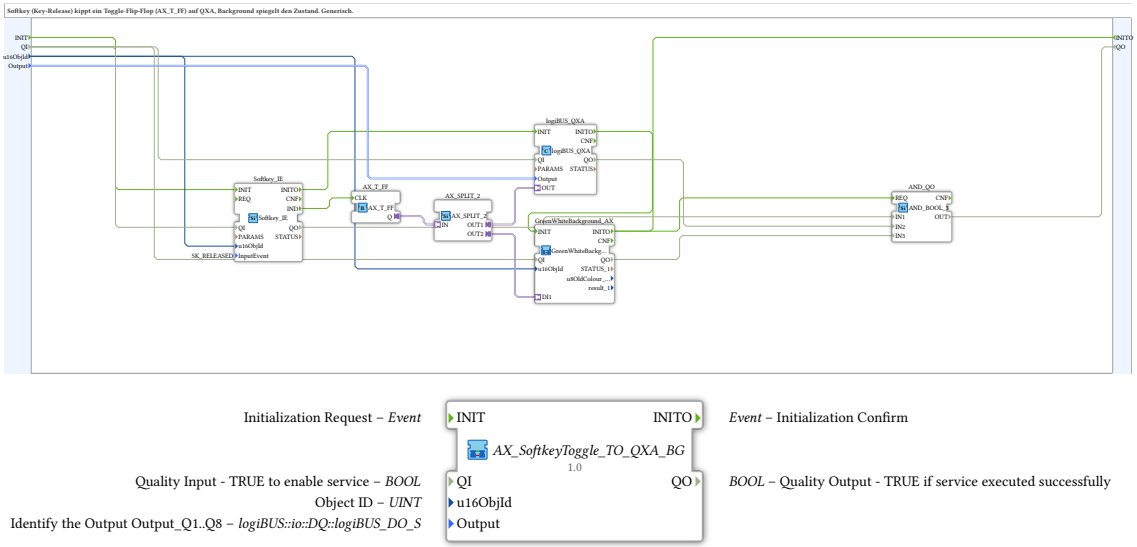
Anwendungsszenarien

- Ein/Aus-Ausgang mit SoftKey, Joystick, IO-Diag-Testbutton und Web-/Dashboard-Bedienung sowie einer globalen „Alles Aus“-Taste (z. B. Sägeblattantrieb)

Zusammenfassung

`AX_SoftkeyToggle_RS_TO_QXA_BG` verbindet den Toggle-/Reset-Zustand aus `AX_SoftkeyToggle_RS_TO_AX` mit physischem Ausgang, Hintergrundfarbe und lokalem State-Publish.

AX_SoftkeyToggle_TO_QXA_BG



• – ○ • – ○ • – ○ •

Einleitung

Die SubApp **AX_SoftkeyToggle_TO_QXA_BG** realisiert eine Softkey-basierte Toggle-Funktion mit visueller Zustandsrückmeldung. Ein Tastendruck beziehungsweise ein definiertes Softkey-Release-Ereignis kippt ein internes Toggle-Flip-Flop und schaltet darüber einen logiBUS-Ausgang (QXA). Parallel wird der Hintergrund eines verbundenen Anzeige-Bausteins an den aktuellen Schaltzustand angepasst. Die SubApp ist generisch aufgebaut und kann über zwei Eingangsparameter flexibel an unterschiedliche Softkeys und Ausgänge angepasst werden.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; startet die Kette `Softkey_IE` ⟶ `logiBUS_QXA` ⟶ `GreenWhiteBackground_AX`.

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung; kommt von `GreenWhiteBackground_AX.INITO`.

Daten-Eingänge

Name	Datentyp	Initialwert	Beschreibung
`QI`	`BOOL`	`TRUE`	Quality Input, Freigabe des Dienstes; wird an `Softkey_IE`, `logiBUS_QXA` und `GreenWhiteBackground_AX` weitergereicht.
`u16ObjId`	`UINT`	`ID_NULL`	Objekt-ID des zu überwachenden Softkeys.
`Output`	`logiBUS::io::DQ::logiBUS_DO_S`	`logiBUS_DO::Invalid`	Identifiziert den physischen Ausgang (`Output_Q1` bis `Output_Q8`).

Daten-Ausgänge

Name	Datentyp	Beschreibung
`QO`	`BOOL`	Quality Output, `AND` der `QO`-Ausgänge von `Softkey_IE`, `logiBUS_QXA` und `GreenWhiteBackground_AX` (über `AND_QO`).

Adapter

Keine.

Funktionsweise

Die SubApp verarbeitet intern eine Softkey-Eingabe und setzt daraus einen stabilen binären Zustand:

1. Der Funktionsbaustein **Softkey_IE** überwacht den über `u16ObjId` spezifizierten Softkey. Er ist so konfiguriert, dass er auf das Ereignis **SK_RELEASED** (Loslassen der Taste) reagiert.
2. Bei einem erkannten Release erzeugt **Softkey_IE** ein Ereignis an seinem Ausgang `IND`.
3. Dieses Ereignis wird an den Takteingang `CLK` des Toggle-Flip-Flops **AX_T_FF** weitergeleitet.
4. Das Flip-Flop wechselt bei jedem Takt seinen Zustand `Q` zwischen `FALSE` und `TRUE`.
5. Der Adapter **AX_SPLIT_2** verteilt den Zustand auf zwei parallele Pfade:
   - `OUT1` führt zum Funktionbaustein **logiBUS_QXA**, der den physischen Ausgang entsprechend setzt.
   - `OUT2` führt zum SubApp-Baustein **GreenWhiteBackground_AX**, der die Hintergrundfarbe des Bedienelements an den Zustand anpasst.

Somit bewirkt jeder Softkey-Release einen Wechsel des Ausgangszustands und der dargestellten Hintergrundfarbe.

Initialisierung und Qualität: `INIT` ⟶ `Softkey_IE.INIT`, `Softkey_IE.INITO` ⟶ `logiBUS_QXA.INIT`, `logiBUS_QXA.INITO` ⟶ `GreenWhiteBackground_AX.INIT`, `GreenWhiteBackground_AX.INITO` ⟶ `INITO`

und → `AND_QO.REQ`. `QI` geht an `Softkey_IE.QI`, `logiBUS_QXA.QI` und `GreenWhiteBackground_AX.QI`; deren `QO`-Ausgänge werden über `AND_QO` (`iec61131::booleanOperators::AND_BOOL_3`) zu `QO` verknüpft.

Technische Besonderheiten

- Nach außen besitzt die SubApp nur `INIT`/`INITO` als Ereignisse sowie die Daten-Eingänge `QI`, `u16ObjId`, `Output` und den Daten-Ausgang `QO`. Die Taktereignisse (`IND`, `CLK`) und Adapter werden nur intern verwendet und sind nach außen nicht sichtbar.
- `QI` ist nicht mehr fest auf `TRUE` verdrahtet, sondern wird von außen durchgereicht (Initialwert `TRUE`).
- Die Parameter `u16ObjId` und `Output` sind als generische Eingänge ausgeführt. Dadurch kann der Baustein für beliebige Softkeys und Ausgänge verwendet werden, ohne die interne Struktur zu verändern.
- Die Initialwerte `ID_NULL` beziehungsweise `logiBUS_DO::Invalid` stellen sicher, dass der Baustein ohne explizite Konfiguration zunächst keine Aktion ausführt.
- Die Verbindungen der Eingänge zu den internen Funktionbausteinen sind im Editor als unsichtbar markiert (`Visible = false`), wodurch die Netzübersicht kompakt bleibt.
- Die Kopplung von Ausgangstreiber und Hintergrund-Feedback über einen gemeinsamen Adapter-Splitter reduziert den Verdrahtungsaufwand und gewährleistet einen synchronen Zustandswechsel.

Zustandsübersicht

Das interne Toggle-Flip-Flop besitzt zwei stabile Zustände:

Zustand des Flip-Flops	Bedeutung für den Ausgang	Hintergrundanzeige
`Q = FALSE`	Ausgang inaktiv	Weiß
`Q = TRUE`	Ausgang aktiv	Grün

Jedes Softkey-Release-Ereignis erzeugt einen Wechsel zwischen diesen beiden Zuständen. Der aktuelle Zustand bleibt solange erhalten, bis das nächste Ereignis eintrifft.

Anwendungsszenarien

- **Bedienfelder mit Softkeys**: Umschaltung zwischen zwei Betriebsarten oder Funktionen über einen einzelnen Softkey.
- **Visuelle Statusanzeige**: Der Hintergrund des Softkeys oder eines zugeordneten Elements zeigt den aktuellen Schaltzustand an.
- **EIN/AUS-Steuerung**: Ein Softkey dient als Taster für einen dauerhaft aktiven oder inaktiven Ausgang.
- **Generische Wiederverwendung**: Durch die konfigurierbaren Eingänge kann derselbe Baustein an verschiedenen Softkey-Positionen und Ausgängen eingesetzt werden.

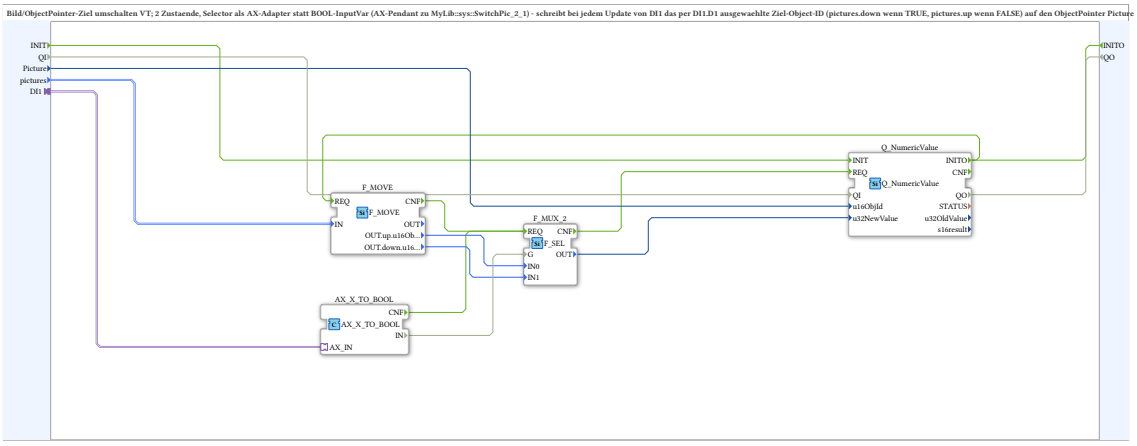
Vergleich mit ähnlichen Bausteinen

Gegenüber einem einzelnen Toggle-Flip-Flop wie **AX_T_FF** bietet dieser Baustein eine höhere Integration: Die Softkey-Erkennung, die Toggle-Logik, die Ausgangsansteuerung und die Hintergrundrückmeldung sind in einer wiederverwendbaren SubApp zusammengefasst. Im Unterschied zu einem einfachen Taster-Baustein, der nur während des Drückens aktiv ist, bleibt der Zustand hier über das Release-Ereignis hinweg erhalten. Die Kapselung erleichtert die Wartung und reduziert die Komplexität im übergeordneten Anwendungsnetz.

Fazit

Die SubApp **AX_SoftkeyToggle_TO_QXA_BG** ist eine kompakte und flexibel einsetzbare Lösung für Softkey-basierte Umschaltaufgaben. Sie kombiniert Ereignisverarbeitung, Zustandsspeicherung, Ausgangsansteuerung und visuelles Feedback in einem einzigen, generisch konfigurierbaren Baustein. Durch die klare Trennung der internen Funktionen und die über Daten-Eingänge erfolgende Parametrierung eignet sie sich hervorragend für den Einsatz in unterschiedlichen Bedien- und Anzeigesystemen.

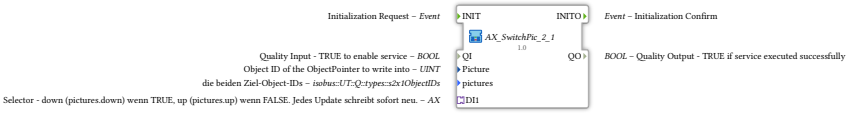
AX_SwitchPic_2_1



• — ○ • — ○ • — ○ •

Einleitung

`AX_SwitchPic_2_1` schaltet das Ziel eines VT-**ObjectPointers** (Bild) zwischen zwei Zuständen um. Der Selektor kommt als **AX-Adapter** statt als `BOOL`-Eingangsvariable (AX-Pendant zu `MyLib::sys::SwitchPic_2_1`): Bei jedem Update von `DI1` wird die per `DI1.D1` gewählte Ziel-Object-ID (`pictures.down` bei `TRUE`, `pictures.up` bei `FALSE`) auf den ObjectPointer `Picture` geschrieben. Ein externer Trigger ist nicht nötig, da das eigene `E1/D1` von `DI1` direkt speist. Die Initialisierung läuft über `INIT/INITO` des inneren `Q_NumericValue` (Quality-Muster mit `QI/QO`).



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung – an `Q_NumericValue.INIT`; weitere Ereignisse kommen über den Adapter-Socket

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung – von `Q_NumericValue.INITO`

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input – TRUE, um den Dienst freizugeben (an `Q_NumericValue.QI`)
- **Picture** (`UINT`): Object-ID des ObjectPointers, in den geschrieben wird
- **pictures** (`isobus::UT::Q::types::s2x1ObjectIDs`): Die beiden Ziel-Object-IDs (`up`, `down`); Vorbelegung: beide `ID_NULL`

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde (von `Q_NumericValue.QO`)

Adapter

- **DI1** (Socket): Unidirektionaler Adapter vom Typ `adapter::types::unidirectional::AX` – Selektor: `down` (`pictures.down`) bei `TRUE`, `up` (`pictures.up`) bei `FALSE`. Jedes Update schreibt sofort neu.

Verwendete Funktionsbausteine (FBs)

- **F_MOVE**: `iec61131::selection::F_MOVE` (`DataType=isobus::UT::Q::types::s2x1ObjectIDs`) – kopiert `pictures` auf `OUT.up.u16ObjId/OUT.down.u16ObjId`
- **AX_X_TO_BOOL**: `adapter::conversion::unidirectional::AX_X_TO_BOOL` – wandelt den AX-Selektor in Ereignis `CNF` und Wert `IN` (`BOOL`)
- **F_MUX_2**: `iec61131::selection::F_SEL` – wählt `IN0` (`up`) bei `G=FALSE`, `IN1` (`down`) bei `G=TRUE`

- **Q_NumericValue**: `isobus::UT::Q::Q_NumericValue` – schreibt den gewählten Wert (`u32NewValue`) in das VT-Objekt `u16ObjId`

Programmablauf und Verbindungen

1. `INIT` ⟶ `Q_NumericValue.INIT`; `Q_NumericValue.INITO` ⟶ `INITO` und ⟶ `F_MOVE.REQ`; `F_MOVE.CNF` ⟶ `F_MUX_2.REQ` (Initialisierung der Ziel-Object-IDs).
2. `DI1` ⟶ `AX_X_TO_BOOL.AX_IN`; `AX_X_TO_BOOL.CNF` ⟶ `F_MUX_2.REQ`, `AX_X_TO_BOOL.IN` ⟶ `F_MUX_2.G`.
3. `F_MOVE.OUT.up.u16ObjId` ⟶ `F_MUX_2.IN0`; `F_MOVE.OUT.down.u16ObjId` ⟶ `F_MUX_2.IN1`.
4. `F_MUX_2.CNF` ⟶ `Q_NumericValue.REQ`; `F_MUX_2.OUT` ⟶ `Q_NumericValue.u32NewValue`; `Picture` ⟶ `Q_NumericValue.u16ObjId`.
5. `QI` ⟶ `Q_NumericValue.QI`; `Q_NumericValue.QO` ⟶ `QO`.

Technische Besonderheiten

- Im Gegensatz zu `SwitchPic_2_1` (dort `DI1: BOOL` plus externes `EI`-Ereignis) benötigt die AX-Variante keinen externen Trigger.
- Das Ziel `ID_NULL` kann z. B. genutzt werden, um einen ObjectPointer „leer“ zu schalten.

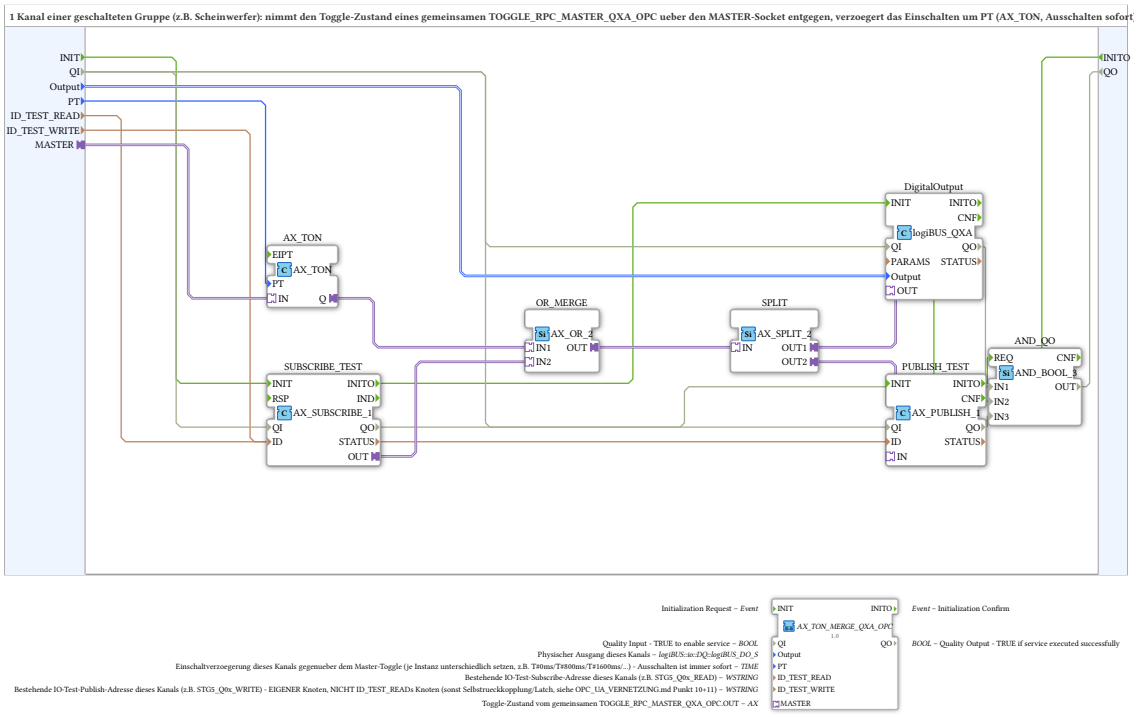
Anwendungsszenarien

- Umschalten eines ObjectPointers zwischen z. B. `Rectangle_Aktiv` und `ID_NULL` je nach Betriebszustand (z. B. Anzeige für Elevatorband, Igelband, Befüllband)

Fazit

`AX_SwitchPic_2_1` schreibt bei jedem Update eines AX-Selektors eine von zwei konfigurierbaren Object-IDs auf einen VT-ObjectPointer.

AX_TON_MERGE_QXA_OPC



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein `AX_TON_MERGE_QXA_OPC` realisiert einen Kanal einer geschalteten Gruppe (z. B. eine Scheinwerfer-Gruppe). Er nimmt den Toggle-Zustand eines gemeinsamen Masters über den `MASTER`-Socket entgegen, verzögert das Einschalten um eine einstellbare Zeit (`PT`) und schaltet den physischen Ausgang sofort aus. Das bestehende IO-Test-Kommando des Kanals wird über eine ODER-Verknüpfung mit eingebunden. Dadurch lassen sich mehrere Kanäle mit unterschiedlichen Einschaltverzögerungen zu einer Gruppe zusammenfassen, um kapazitive Einschaltstrom-Spitzen zu vermeiden.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsanforderung; startet die Init-Kette bei SUBSCRIBE_TEST.INIT.

Die eigentliche Signalverarbeitung läuft weiterhin rein daten- und adaptergesteuert.

Ereignis-Ausgänge

- **INITO** (EInit): Initialisierungsbestätigung; kommt von PUBLISH_TEST.INITO, dem Ende der Init-Kette.

Daten-Eingänge

Name	Typ	Beschreibung
QI	BOOL	Quality Input, schaltet den Dienst frei; Initialwert TRUE; wird an SUBSCRIBE_TEST, DigitalOutput und PUBLISH_TEST weitergereicht
Output	logiBUS::io::DQ::logiBUS_DO_S	Physischer Ausgang dieses Kanals; Initialwert logiBUS_DO::Invalid
PT	TIME	Einschaltverzögerung gegenüber dem Master-Toggle (z. B. T#0ms, T#800ms, T#1600ms); Ausschalten erfolgt immer sofort
ID_TEST_READ	WSTRING	Adresse für die bestehende IO-Test-Subscribe (z. B. STG5_Q0x_READ)
ID_TEST_WRITE	WSTRING	Adresse für die bestehende IO-Test-Publish (z. B. STG5_Q0x_WRITE) – **eigener Knoten**, nicht derselbe wie ID_TEST_READ, um Selbstrückkopplung/Latch zu vermeiden

Daten-Ausgänge

Name	Typ	Beschreibung
QO	BOOL	Quality Output, UND-Verknüpfung (AND_QO, AND_BOOL_3) der QO-Ausgänge von SUBSCRIBE_TEST, DigitalOutput und PUBLISH_TEST

Adapter

Name	Typ	Richtung	Beschreibung
MASTER	adapter::types::unidirectional::AX	Socket (Eingang)	Toggle-Zustand vom gemeinsamen TOGGLE_RPC_MASTER_QXA_OPC.OUT

Funktionsweise

Der MASTER-Socket überträgt den Toggle-Zustand eines zentralen Masters. Dieser Zustand wird an den Timer-Baustein AX_TON weitergeleitet. AX_TON verzögert nur das **Einschalten** um die konfigurierte Zeit PT; das Ausschalten geschieht unmittelbar. Das verzögerte Signal wird zusammen mit dem bestehenden IO-Test-Befehl des Kanals (über AX_SUBSCRIBE_1) in einer ODER-Verknüpfung (AX_OR_2) kombiniert. Das Ergebnis wird über einen Split (AX_SPLIT_2) sowohl an den physischen Ausgang (logiBUS_QXA) als auch an die Publish-Schnittstelle (AX_PUBLISH_1) gesendet. Dadurch bleibt der IO-Test-Befehlsweg erhalten und das verzögerte Toggle-Signal wirkt parallel dazu.

Initialisierung und Qualität: INIT → SUBSCRIBE_TEST.INIT, SUBSCRIBE_TEST.INITO → DigitalOutput.INIT, DigitalOutput.INITO → PUBLISH_TEST.INIT, PUBLISH_TEST.INITO → INITO und AND_QO.REQ. QI wird an SUBSCRIBE_TEST.QI, DigitalOutput.QI und PUBLISH_TEST.QI weitergereicht; deren QO-Ausgänge werden in AND_QO (iec61131::booleanOperators::AND_BOOL_3, IN1 bis IN3) zum Ausgang QO verknüpft.

Technische Besonderheiten

- **Verzögerung nur beim Einschalten**: AX_TON sorgt dafür, dass das Einschalten um PT verzögert wird, das Ausschalten jedoch sofort erfolgt. Dies vermeidet unerwünschte Latenzen beim Abschalten.
- **Mehrere Kanäle mit einem Master**: Mehrere Instanzen dieses Bausteins können denselben Master-Socket verwenden. Die Verdrahtung erfolgt lokal im SubApp-Netzwerk, eine OPC-UA-Anbindung ist nicht erforderlich.
- **Integration des IO-Tests**: Das bestehende Test-Kommando (Subscribe) wird über eine ODER-Verknüpfung in den Ausgangssignalpfad integriert. Dadurch bleiben Diagnose- und Testfunktionen erhalten.
- **Vermeidung von Selbstrückkopplung**: Für ID_TEST_READ und ID_TEST_WRITE müssen unterschiedliche OPC-UA-Knoten verwendet werden, um ungewolltes Latch-Verhalten zu verhindern.
- **Flexible Verzögerungszeiten**: PT ist pro Instanz einstellbar, sodass eine gestaffelte Aktivierung mehrerer Kanäle einfach konfiguriert werden kann.

Zustandsübersicht

Der Baustein selbst besitzt keinen expliziten Zustandsautomaten. Die Zustandslogik wird über die internen Funktionsbausteine bestimmt:

- **AX_TON**: Interner Timer mit den Zuständen IN=TRUE (Timer läuft) und Q (Ausgang nach Ablauf der Verzögerung). Bei IN=FALSE wird Q sofort FALSE.
- **AX_OR_2**: Logische ODER-Verknüpfung der beiden Eingänge.
- **AX_SPLIT_2**: Verteilt das Eingangssignal auf zwei Ausgänge.

Daraus ergibt sich folgendes Verhalten des Gesamtbausteins:

Master-Toggle	Test-Kommando	Verzögerte Ausgangslogik	Phys. Ausgang
TRUE (mit PT)	FALSE	nach Ablauf PT → TRUE	TRUE
TRUE (sofort)	FALSE	sofort TRUE (falls PT=0)	TRUE
FALSE (sofort)	FALSE	sofort FALSE	FALSE
egal	TRUE	TRUE (Test aktiv)	TRUE

Anwendungsszenarien

- **Beleuchtungsgruppen mit mehreren Scheinwerfer-Kanälen**: Eine Gruppe von 6 LED-Kanal-Instanzen wird über einen gemeinsamen Master geschaltet. Durch gestaffelte PT-Werte (z. B. 0 ms, 800 ms, 1600 ms, …) wird der Einschaltstrom der LED-Treiber zeitlich verteilt, sodass kapazitive Lasten nicht gleichzeitig zugeschaltet werden.
- **Steuerung von Verbrauchern mit hohem Anlaufstrom**: Überall dort, wo mehrere Verbraucher nacheinander hochfahren sollen, um Spannungseinbrüche zu vermeiden.
- **Integration in bestehende IO-Test-Infrastruktur**: Der Baustein lässt sich problemlos in Anlagen mit vorhandenen OPC-UA-Testadressen einbinden, ohne die Testlogik zu verändern.

Vergleich mit ähnlichen Bausteinen

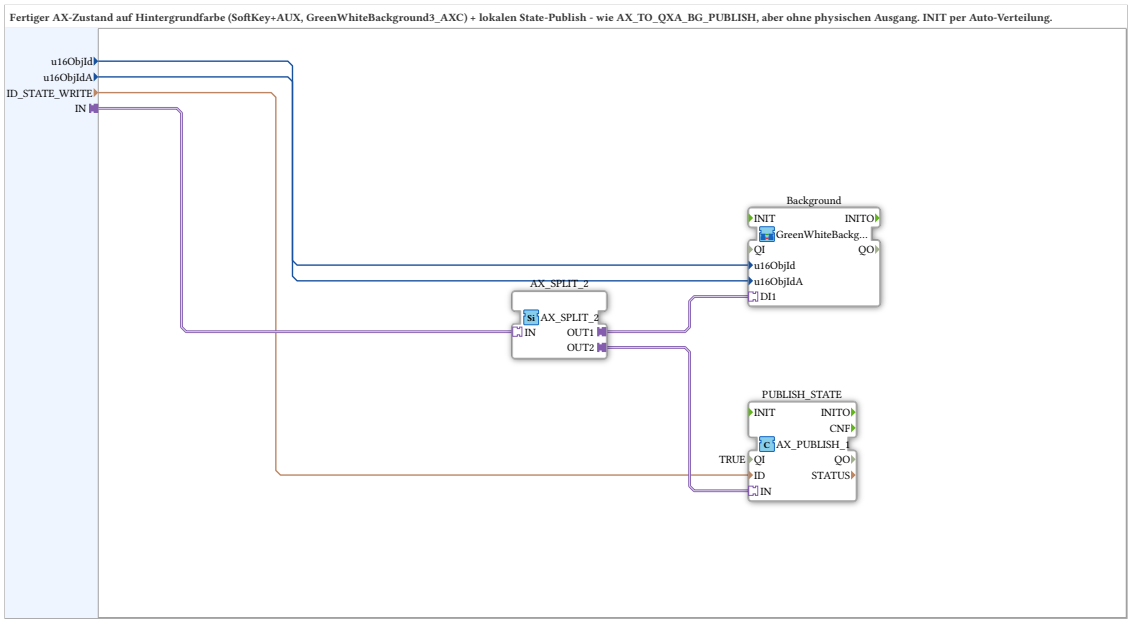
- **TOGGLE_RPC_MERGE_QXA_OPC**: Dieser Baustein bietet eine **vollständig eigenständige** Kanalsteuerung mit eigenem SoftKey, ohne Abhängigkeit von einem externen Master. Er ist geeignet, wenn jeder Kanal unabhängig geschaltet werden soll.
- **MERGE_SWITCH_1_QXA_OPC**: Vereint ähnliche Funktionen, jedoch ohne die Option einer gemeinsamen Master-Ansteuerung. Er eignet sich für einfache Ein-/Aus-Schaltungen.

Im Gegensatz zu diesen Bausteinen ermöglicht AX_TON_MERGE_QXA_OPC eine **zentrale** Toggle-Steuerung durch einen Master und eine **zeitlich versetzte** Aktivierung der einzelnen Kanäle, was bei großen Lastgruppen von Vorteil ist.

Fazit

AX_TON_MERGE_QXA_OPC ist ein flexibler Baustein für die verteilte Ansteuerung mehrerer Ausgangskanäle mit einem gemeinsamen Toggle-Master. Die integrierte Einschaltverzögerung reduziert kritische Stromspitzen, ohne die Abschaltzeiten zu beeinflussen. Durch die ODER-Verknüpfung mit dem bestehenden IO-Test-Befehl bleibt die Diagnosefähigkeit der Anlage vollständig erhalten. Die Parameter PT, ID_TEST_READ und ID_TEST_WRITE erlauben eine einfache Anpassung an die jeweilige Umgebung.

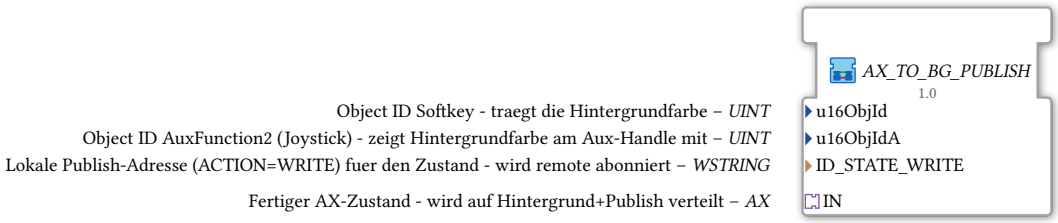
AX_TO_BG_PUBLISH



• – ○ • – ○ • – ○ •

Einleitung

`AX_TO_BG_PUBLISH` bringt einen fertigen AX-Zustand auf die Hintergrundfarbe (SoftKey und AUX, `GreenWhiteBackground3_AXC`) und auf einen lokalen State-Publish. Er entspricht `AX_TO_QXA_BG_PUBLISH`, jedoch ohne physischen Ausgang. Die Initialisierung erfolgt per Auto-Verteilung (keine eigenen Ereignisanschlüsse). Der Baustein wurde aus `Baender_EinAus.SUB` (`Background_Alle/AX_PUBLISH_Alle`) als wiederverwendbarer Baustein herausgelöst.



Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Variable	Typ	Vorgabe	Kommentar
`u16ObjId`	UINT	`ID_NULL`	Object ID SoftKey; trägt die Hintergrundfarbe.
`u16ObjIdA`	UINT	`ID_NULL`	Object ID AuxFunction2 (Joystick); zeigt die Hintergrundfarbe am Aux-Handle mit.
`ID_STATE_WRITE`	WSTRING		Lokale Publish-Adresse (`ACTION=WRITE`) für den Zustand; wird remote abonniert.

Daten-Ausgänge

Keine vorhanden.

Adapter

Adapter	Typ	Richtung	Kommentar
IN	adapter::types::unidirectional::AX	Socket	Fertiger AX-Zustand; wird auf Hintergrund und Publish verteilt.

Verwendete Funktionsbausteine (FBs)

- **AX_SPLIT_2** (adapter::events::unidirectional::AX_SPLIT_2): Adapter-Fanout (strikt 1:1)
- **Background** (SubApp MyLib::sys::GreenWhiteBackground3_AXC): setzt die Hintergrundfarbe von SoftKey und Aux; siehe GreenWhiteBackground3_AXC
- **PUBLISH_STATE** (adapter::net::AX_PUBLISH_1, QI = TRUE): lokaler State-Publish

Programmablauf und Verbindungen

1. IN -> AX_SPLIT_2.IN.
2. AX_SPLIT_2.OUT1 -> Background.DI1; AX_SPLIT_2.OUT2 -> PUBLISH_STATE.IN.
3. u16ObjId -> Background.u16ObjId; u16ObjIdA -> Background.u16ObjIdA; ID_STATE_WRITE -> PUBLISH_STATE.ID.
4. Der Baustein hat keine Ereignis- und QI/QO-Anschlüsse; PUBLISH_STATE.QI ist fest auf TRUE parametriert.

Technische Besonderheiten

- Der AX_SPLIT_2 ist nötig, weil der Adapter-Fanout strikt 1:1 ist.
- Die Initialisierung läuft per Auto-Verteilung statt über einen eigenen INIT-Eingang.
- Gegenüber AX_TO_QXA_BG_PUBLISH fehlen der physische Ausgang (Output) sowie QI/QO und INIT/INITO.

Anwendungsszenarien

- Reine Anzeige- und Publish-Kanäle ohne eigenen Hardware-Ausgang, z. B. gleich aufgebaute Kanäle mit Hintergrundfarbe am SoftKey und Aux-Handle sowie Zustandsveröffentlichung.

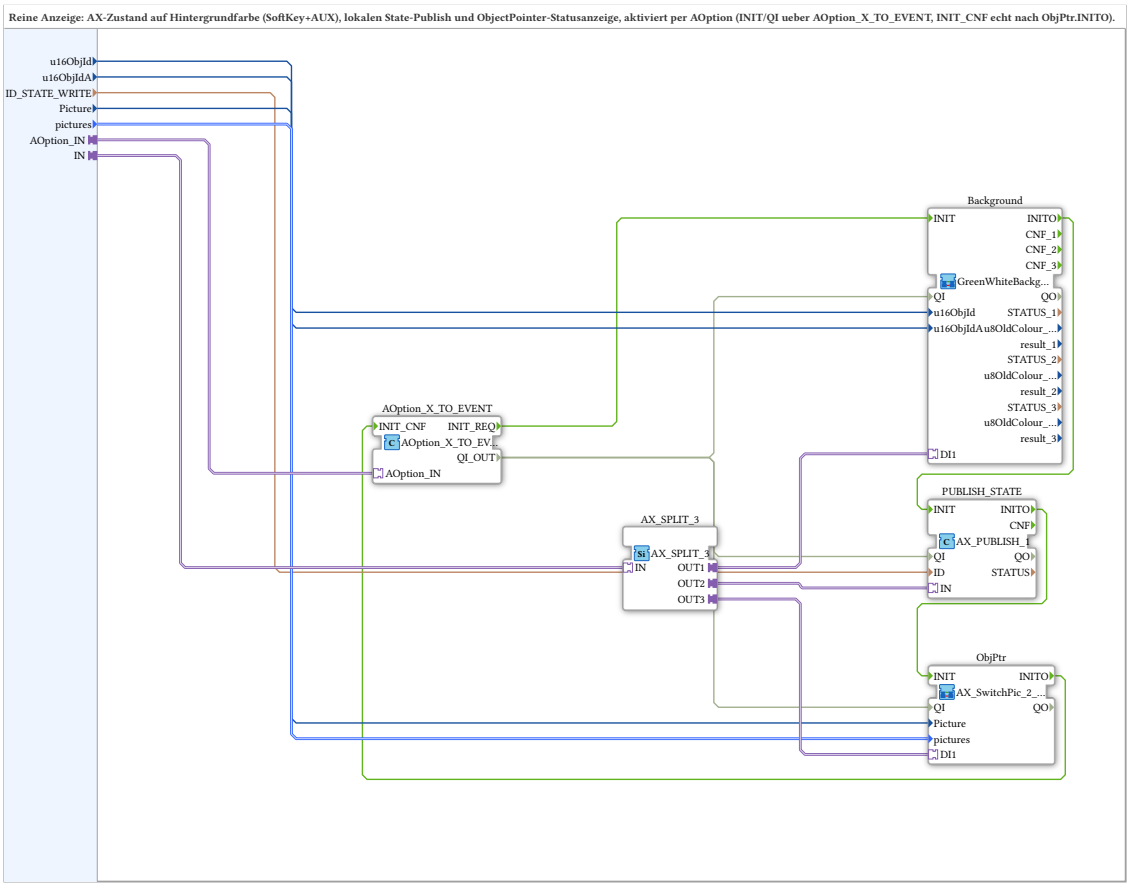
Zusammenfassung

AX_TO_BG_PUBLISH verteilt einen fertigen AX-Zustand auf die Hintergrundfarbe von SoftKey und Aux und einen lokalen State-Publish, ohne physischen Ausgang.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

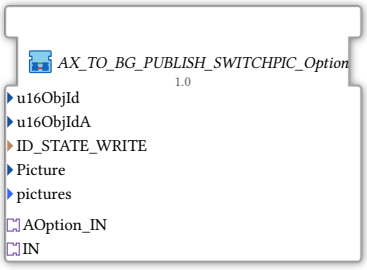
AX_TO_BG_PUBLISH_SWITCHPIC_Option



• — ○ • — ○ • — ○ •

Einleitung

`AX_TO_BG_PUBLISH_SWITCHPIC_Option` ist eine reine Anzeige: Er bringt einen fertigen AX-Zustand auf die Hintergrundfarbe (SoftKey und AUX), einen lokalen State-Publish und die ObjectPointer-Statusanzeige (`AX_SwitchPic_2_1`). Aktiviert wird er per `AOption`: `INIT` und `QI` kommen über `AOption_X_TO_EVENT`, das `INIT_CNF` kommt echt nach `ObjPtr.INITO` zurück. Er ist der Anzeige-Teil von `AX_TO_QXA_BG_PUBLISH_SWITCHPIC_OPT` als eigener Baustein und entspricht `AX_TO_QXA_BG_PUBLISH_SWITCHPIC` ohne physischen Ausgang.



Object ID Softkey - traegt die Hintergrundfarbe – *UINT*
Object ID AuxFunction2 (Joystick) - zeigt Hintergrundfarbe am Aux-Handle mit – *UINT*
Lokale Publish-Adresse (ACTION=WRITE) fuer den Zustand - wird remote abonniert – *WSTRING*
Object ID des ObjectPointer, der den Status-Kaestchen-Inhalt traegt – *UINT*
die beiden Ziel-Object-IDs (up=aus, down=an) – *isobus::UT::Q::types::s2x1ObjectIDs*
Aktiviert/deaktiviert die Anzeige – *AOption*
Fertiger AX-Zustand fuer Hintergrund, Publish und ObjectPointer – *AX*

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Variable	Typ	Vorgabe	Kommentar
u16ObjId	UINT	ID_NULL	Object ID SoftKey; trägt die Hintergrundfarbe.
u16ObjIdA	UINT	ID_NULL	Object ID AuxFunction2 (Joystick); zeigt die Hintergrundfarbe am Aux-Handle mit.
ID_STATE_WRITE	WSTRING		Lokale Publish-Adresse (ACTION=WRITE) für den Zustand; wird remote abonniert.
Picture	UINT		Object ID des Object-Pointers, der den Status-Kästchen-Inhalt trägt.
pictures	isobus::UT::Q::types::s2x1ObjectIDs	(up := (u16ObjId := ID_NULL), down := (u16ObjId := ID_NULL))	Die beiden Ziel-Object-IDs (up = aus, down = an).

Daten-Ausgänge

Keine vorhanden.

Adapter

Adapter	Typ	Richtung	Kommentar
AOption_IN	Option::AOption	Socket	Aktiviert/deaktiviert die Anzeige.
IN	adapter::types::unidirectional::AX	Socket	Fertiger AX-Zustand für Hintergrund, Publish und ObjectPointer.

Verwendete Funktionsbausteine (FBs)

- **AOption_X_TO_EVENT** (Option::conversion::AOption_X_TO_EVENT): wandelt die Option in INIT_REQ/QI_OUT und nimmt INIT_CNF entgegen
- **AX_SPLIT_3** (adapter::events::unidirectional::AX_SPLIT_3): Adapter-Fanout (strikt 1:1)
- **Background** (SubApp MyLib::sys::GreenWhiteBackground3_AX): Hintergrundfarbe von SoftKey und Aux; siehe GreenWhiteBackground3_AX
- **PUBLISH_STATE** (adapter::net::AX_PUBLISH_1): lokaler State-Publish
- **ObjPtr** (SubApp MyLib::sys::AX_SwitchPic_2_1): ObjectPointer-Statusanzeige; siehe AX_SwitchPic_2_1

Programmablauf und Verbindungen

1. Initialisierungskette: AOption_X_TO_EVENT.INIT_REQ -> Background.INIT; Background.INITO -> PUBLISH_STATE.INIT; PUBLISH_STATE.INITO -> ObjPtr.INIT; ObjPtr.INITO -> AOption_X_TO_EVENT.INIT_CNF.
2. Qualifier: AOption_X_TO_EVENT.QI_OUT -> Background.QI, PUBLISH_STATE.QI und ObjPtr.QI.
3. Option: AOption_IN -> AOption_X_TO_EVENT.AOption_IN.
4. AX-Zustand: IN -> AX_SPLIT_3.IN; OUT1 -> Background.DI1; OUT2 -> PUBLISH_STATE.IN; OUT3 -> ObjPtr.DI1.
5. Daten: u16ObjId -> Background.u16ObjId; u16ObjIdA -> Background.u16ObjIdA; ID_STATE_WRITE -> PUBLISH_STATE.ID; Picture -> ObjPtr.Picture; pictures -> ObjPtr.pictures.
6. Der Baustein hat keine eigenen Ereignis- und QI/QO-Anschlüsse und keinen Plug; Aktivierung und Bestätigung laufen über AOption_IN.

Technische Besonderheiten

- Der AX_SPLIT_3 ist nötig, weil der Adapter-Fanout strikt 1:1 ist.

- Das `INIT_CNF` kommt echt vom Ende der Kette (`ObjPtr.INITO`), nicht aus einer Kurzschlussverbindung.
- Es gibt keinen `AOption_OUT`-Plug; der Baustein kann daher nicht daisy-chained werden.

Anwendungsszenarien

- Reine Anzeigeseite eines Kanals (Hintergrundfarbe, State-Publish, Status-Bild), die nur bei ausgewählter Option aktiv ist, z. B. an einer `AOption_OR_2`-Verknüpfung mehrerer Varianten.

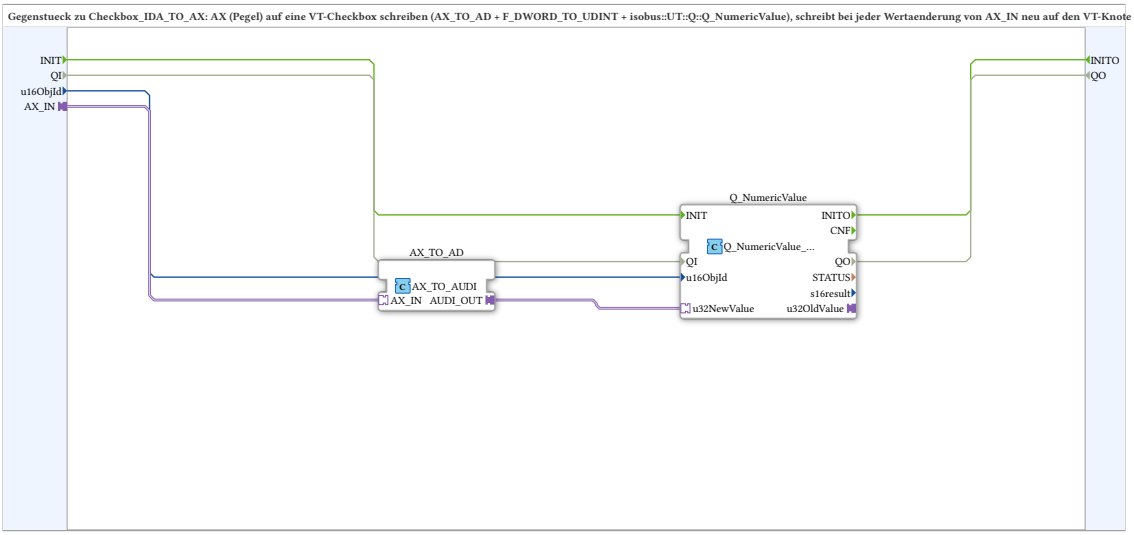
Zusammenfassung

`AX_TO_BG_PUBLISH_SWITCHPIC_Option` zeigt einen fertigen AX-Zustand als Hintergrundfarbe, State-Publish und Status-Bild an und wird per `AOption` aktiviert.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

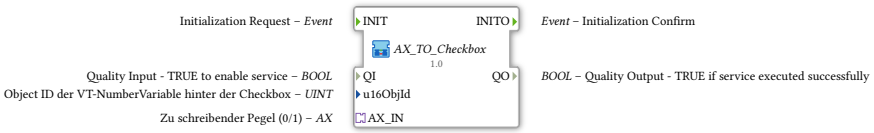
AX_TO_Checkbox



• – ○ • – ○ • – ○ •

Einleitung

`AX_TO_Checkbox` ist das Gegenstück zu `Checkbox_IDA_TO_AX`: Es schreibt einen Pegel (`AX_IN`, 0/1) auf eine VT-Checkbox (ECU → VT statt VT → ECU). Intern wandelt `AX_TO_AD` (`adapter::conversion::unidirectional::AX_TO_AUDI`) den Pegel in einen UDINT-Wert, den `Q_NumericValue` (`isobus::UT::Q::Q_NumericValue_AUDI`) über `u32NewValue` zum VT sendet.



Schnittstelle

Ereignisanschlüsse

- **Eingang `INIT`** (`EInit`): Initialisierungsanforderung – an `Q_NumericValue.INIT`.
- **Ausgang `INITO`** (`EInit`): Initialisierungsbestätigung – von `Q_NumericValue.INITO`.

Eingangsvariablen

- `QI` (BOOL, Default `TRUE`): Quality Input – TRUE, um den Dienst freizugeben (an `Q_NumericValue.QI`).
- `u16ObjId` (UINT, Default `ID_NULL`): ObjectID der VT-NumberVariable hinter der Checkbox.

Ausgangsvariablen

- `QO` (BOOL): Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde (von `Q_NumericValue.QO`).

Adapter

- **Socket `AX_IN`** (`adapter::types::unidirectional::AX`): zu schreibender Pegel (0/1).

Außer `INIT`/`INITO` besitzt der Baustein keine eigenen Ereignisanschlüsse; das Schreiben wird durch den Adapter `AX_IN` ausgelöst.

Funktionsweise

1. `AX_IN` wird über `AX_TO_AD` in einen UDINT-Wert gewandelt.
2. `Q_NumericValue` schreibt diesen Wert unter `u16ObjId` auf den VT-Knoten. Laut Quellcode-Kommentar wird bei jeder Wertänderung von `AX_IN` neu geschrieben.
3. `INIT` ⟶ `Q_NumericValue.INIT`, `Q_NumericValue.INITO` ⟶ `INITO`, `QI` ⟶ `Q_NumericValue.QI`, `Q_NumericValue.QO` ⟶ `QO`.

Technische Besonderheiten

- `u16ObjId` ist die ObjectID der **CNumberVariable**, die die Checkbox als Wertspeicher referenziert - **nicht** die ObjectID der Checkbox (`CInputBoolean`) selbst. Das VT erwartet Wertänderungen unter der NumberVariable-ID (analog `Checkbox_IDA_TO_AX`).
- Instanz-Eindeutigkeit: `Q_NumericValue` erlaubt nur eine Instanz je Ziel-ObjectID im gesamten Projekt.

Zusammenfassung

Schreibt einen AX-Pegel auf eine VT-Checkbox über die zugehörige NumberVariable.

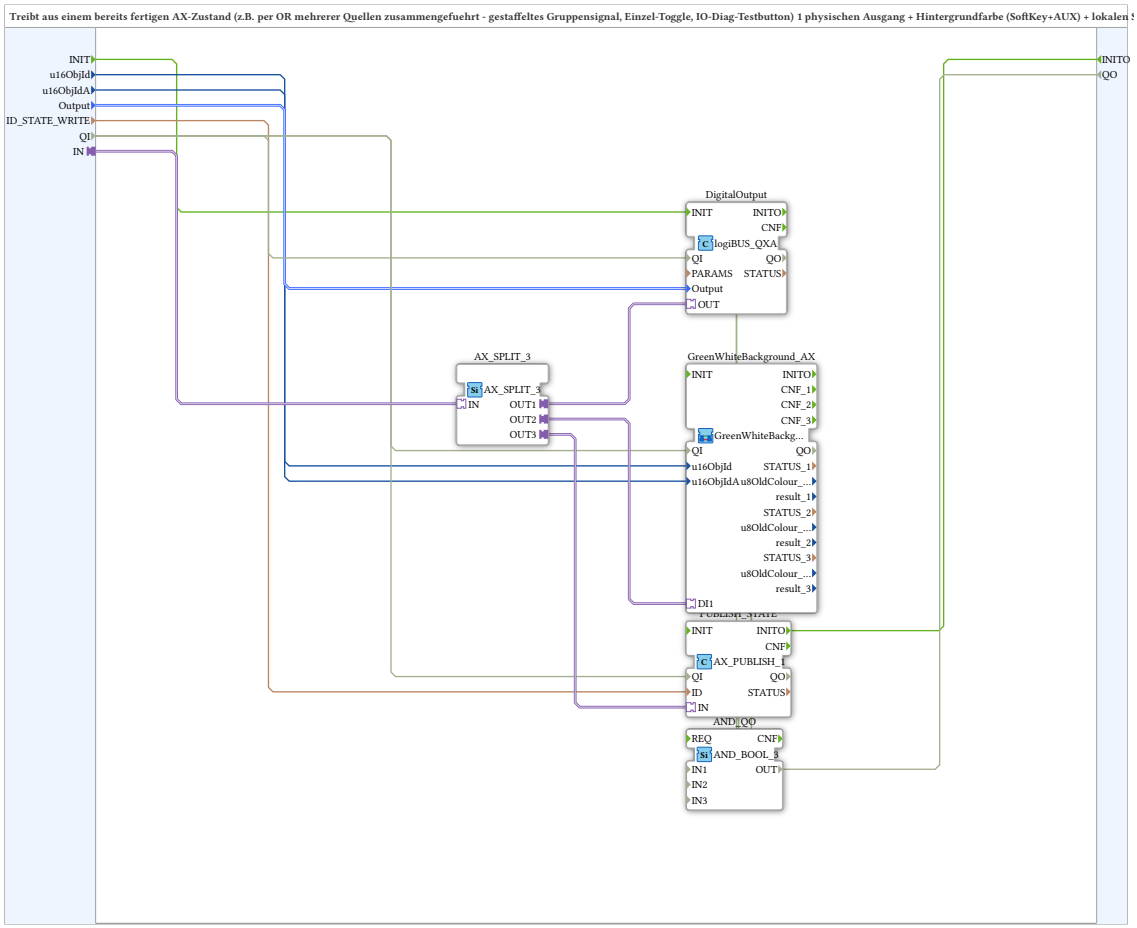
🛠️ Zugehörige Übungen

- Uebung_094b5_AX

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

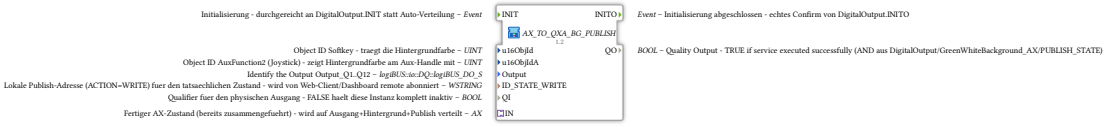
AX_TO_QXA_BG_PUBLISH



• — ○ • — ○ • — ○ •

Einleitung

`AX_TO_QXA_BG_PUBLISH` treibt aus einem bereits fertigen AX-Zustand (z. B. per ODER mehrerer Quellen zusammengeführt, etwa gestaffeltes Gruppensignal, Einzel-Toggle oder IO-Diag-Testbutton) **einen** physischen Ausgang, die Hintergrundfarbe (SoftKey und AUX) und einen lokalen State-Publish für Web-Client und OPC-Dashboard. Er ist das Gegenstück zu `MyLib::sys::AX_SoftkeyToggle_RS_TO_QXA_BG`, das den Toggle/Reset noch selbst erzeugt; hier liegt der AX-Zustand bereits fertig vor.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierung; wird an `DigitalOutput.INIT` durchgereicht statt per Auto-Verteilung

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierung abgeschlossen; echtes Confirm von `PUBLISH_STATE.INITO` am Ende der Kette `DigitalOutput` ⟶ `GreenWhiteBackground_AX` ⟶ `PUBLISH_STATE`

Daten-Eingänge

- **QI** (BOOL): Qualifier; geht an `DigitalOutput`, `GreenWhiteBackground_AX` und `PUBLISH_STATE`, FALSE hält diese Instanz komplett inaktiv (Default `TRUE`)
- **u16ObjId** (UINT): Object ID des SoftKeys, trägt die Hintergrundfarbe (Default `ID_NULL`)

- **u16ObjIdA** (UINT): Object ID der AuxFunction2 (Joystick), zeigt die Hintergrundfarbe am Aux-Handle mit (Default `ID_NULL`)
- **Output** (`logiBUS::io::DQ::logiBUS_DO_S`): Auswahl des Ausgangs `Output_Q1`..`Output_Q12` (Default `logiBUS_DO::Invalid`)
- **ID_STATE_WRITE** (WSTRING): lokale Publish-Adresse (`ACTION=WRITE`) für den tatsächlichen Zustand; wird vom Web-Client/Dashboard remote abonniert

Daten-Ausgänge

- **QO** (BOOL): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde (AND aus `DigitalOutput`, `GreenWhiteBackground_AX` und `PUBLISH_STATE`)

Adapter

- **IN** (Socket, `adapter::types::unidirectional::AX`): fertiger, bereits zusammengeführter AX-Zustand; wird auf Ausgang, Hintergrund und Publish verteilt

Funktionsweise

`IN` wird auf einen `AX_SPLIT_3` (`adapter::events::unidirectional::AX_SPLIT_3`) geführt, der den Zustand auf drei Ziele verteilt:

- `AX_SPLIT_3.OUT1` → `DigitalOutput.OUT`: `DigitalOutput` (`logiBUS::io::DQ::logiBUS_QXA`) schaltet den physischen Ausgang; `Output` → `DigitalOutput.Output`, `QI` → `DigitalOutput.QI` und das Ereignis `INIT` → `DigitalOutput.INIT`.
- `AX_SPLIT_3.OUT2` → `GreenWhiteBackground_AX.DI1`: `GreenWhiteBackground_AX` (`MyLib::sys::GreenWhiteBackground3_AX`) färbt den Hintergrund grün (TRUE) bzw. weiß (FALSE); `u16ObjId` und `u16ObjIdA` gehen an dessen gleichnamige Eingänge, `QI` an `GreenWhiteBackground_AX.QI`.
- `AX_SPLIT_3.OUT3` → `PUBLISH_STATE.IN`: `PUBLISH_STATE` (`adapter::net::AX_PUBLISH_1`, `QI` von außen) publiziert den Zustand an die Adresse `ID_STATE_WRITE` (`ID_STATE_WRITE` → `PUBLISH_STATE.ID`).

Initialisierung und Qualität: `INIT` → `DigitalOutput.INIT`, `DigitalOutput.INITO` → `GreenWhiteBackground_AX.INIT`, `GreenWhiteBackground_AX.INITO` → `PUBLISH_STATE.INIT`, `PUBLISH_STATE.INITO` → `INITO`. `QO` ist der Ausgang der Instanz `AND_QO` (`iec61131::booleanOperators::AND_BOOL_3`) mit `DigitalOutput.QO`, `GreenWhiteBackground_AX.QO` und `PUBLISH_STATE.QO` an `IN1`..`IN3`; ihr `REQ` wird von `PUBLISH_STATE.INITO` ausgelöst.

Technische Besonderheiten

- Herausgelöst aus `Baender_EinAus` (dort dreimal identisch für Igel, Elevator und Querband verdrahtet)
- `ID_STATE_WRITE` ist neu hinzugekommen, weil der Web-Mirror für die Einzel-Bänder-Softkeys einen eigenen State-Publish benötigt
- Version 1.1: `QI` und `INIT` werden an `DigitalOutput` durchgereicht (statt hart `TRUE` und Auto-Verteilung), damit z. B. Klassisch-vs-PVEA-Varianten mit gemeinsamen Pins eine Instanz inaktiv halten und die Initialisierung vom Aufrufer sequenziert werden kann
- Version 1.2: neuer Ereignisausgang `INITO` (Initialisierung abgeschlossen) und Datenausgang `QO` (AND der inneren `QO`); die Initialisierung läuft als Kette `DigitalOutput` → `GreenWhiteBackground_AX` → `PUBLISH_STATE`, `QI` wird an alle drei Instanzen verteilt
- Es gibt keine eigenen Ereignis- oder Datenports für den Zustand; alles läuft über den AX-Adapter `IN`

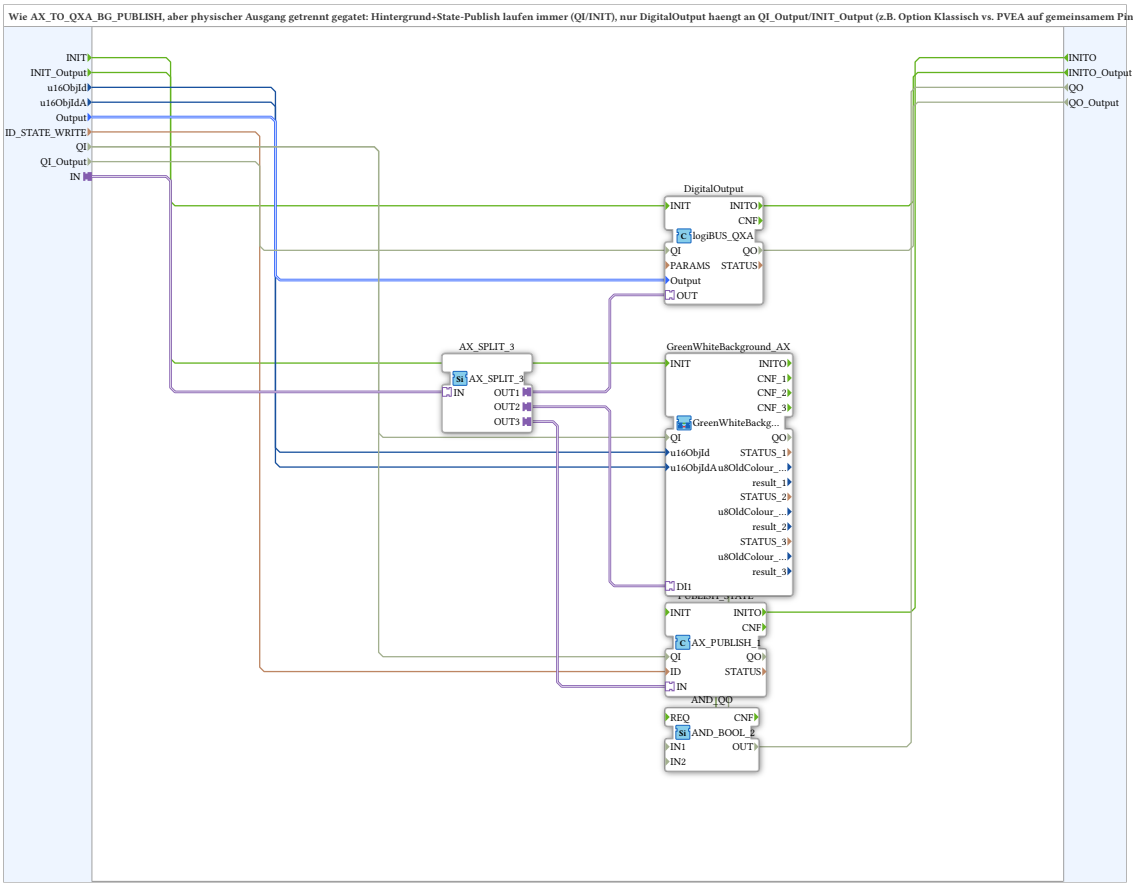
Anwendungsszenarien

- Gruppensignal oder ODER-verknüpfter Zustand, der einen Ausgang schalten, die Softkey-Farbe anzeigen und für das Web veröffentlicht werden soll
- Mehrere zeitversetzte Kanäle, die jeweils einen eigenen Zustand liefern

Fazit

`AX_TO_QXA_BG_PUBLISH` verteilt einen fertigen AX-Zustand auf physischen Ausgang, Hintergrundfarbe und State-Publish.

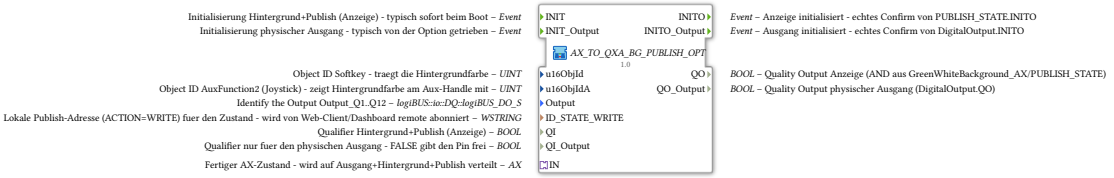
AX_TO_QXA_BG_PUBLISH_OPT



• — ○ • — ○ • — ○ •

Einleitung

`AX_TO_QXA_BG_PUBLISH_OPT` entspricht `AX_TO_QXA_BG_PUBLISH`, schaltet den physischen Ausgang aber getrennt: Hintergrund und State-Publish laufen immer über `QI/INIT`, nur der `DigitalOutput` hängt an `QI_Output/INIT_Output` (z. B. Option „Klassisch“ gegenüber „PVEA“ auf einem gemeinsamen Pin). Der Baustein ist eine Kopie von `AX_TO_QXA_BG_PUBLISH` v1.2; Grund war, dass bei der Variante PVEA mit dem Ausgang auch SoftKey-Hintergrund und State-Publish inaktiv blieben.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierung Hintergrund und Publish (Anzeige); typisch sofort beim Boot
- **INIT_Output** (`EInit`): Initialisierung des physischen Ausgangs; typisch von der Option getrieben

Ereignis-Ausgänge

- **INITO** (`EInit`): Anzeige initialisiert; echtes Confirm von `PUBLISH_STATE.INITO`
- **INITO_Output** (`EInit`): Ausgang initialisiert; echtes Confirm von `DigitalOutput.INITO`

Daten-Eingänge

Variable	Typ	Vorgabe	Kommentar

u16ObjId	UINT	ID_NULL	Object ID SoftKey; trägt die Hintergrundfarbe.
u16ObjIdA	UINT	ID_NULL	Object ID AuxFunction2 (Joystick); zeigt die Hintergrundfarbe am Aux-Handle mit.
Output	logiBUS::io::DQ::logiBUS_DO_1	logiBUS_DO::Invalid	Auswahl des Ausgangs Output_Q1..Output_Q12.
ID_STATE_WRITE	WSTRING		Lokale Publish-Adresse (ACTION=WRITE) für den Zustand; wird von Web-Client/Dashboard remote abonniert.
QI	BOOL	TRUE	Qualifier für Hintergrund und Publish (Anzeige).
QI_Output	BOOL	TRUE	Qualifier nur für den physischen Ausgang; FALSE gibt den Pin frei.

Daten-Ausgänge

Variable	Typ	Vorgabe	Kommentar
QO	BOOL		Quality Output Anzeige (UND aus GreenWhiteBackground_AX.QO und PUBLISH_STATE.QO).
QO_Output	BOOL		Quality Output physischer Ausgang (DigitalOutput.QO).

Adapter

Adapter	Typ	Richtung	Kommentar
IN	adapter::types::unidirectional::AX	Socket	Fertiger AX-Zustand; wird auf Ausgang, Hintergrund und Publish verteilt.

Verwendete Funktionsbausteine (FBs)

- **AX_SPLIT_3** (adapter::events::unidirectional::AX_SPLIT_3): Adapter-Fanout (strikt 1:1)
- **DigitalOutput** (logiBUS::io::DQ::logiBUS_QXA): physischer Ausgang
- **GreenWhiteBackground_AX** (SubApp MyLib::sys::GreenWhiteBackground3_AX): Hintergrundfarbe von SoftKey und Aux; siehe GreenWhiteBackground3_AX
- **PUBLISH_STATE** (adapter::net::AX_PUBLISH_1): lokaler State-Publish
- **AND_QO** (iec61131::booleanOperators::AND_BOOL_2): verknüpft die QO der Anzeige

Programmablauf und Verbindungen

1. AX-Zustand: IN -> AX_SPLIT_3.IN; OUT1 -> DigitalOutput.OUT; OUT2 -> GreenWhiteBackground_AX.DI1; OUT3 -> PUBLISH_STATE.IN.
2. Initialisierung Ausgang: INIT_Output -> DigitalOutput.INIT; DigitalOutput.INITO -> INITO_Output.
3. Initialisierung Anzeige: INIT -> GreenWhiteBackground_AX.INIT; GreenWhiteBackground_AX.INITO -> PUBLISH_STATE.INIT; PUBLISH_STATE.INITO -> INITO und -> AND_QO.REQ.
4. Qualifier: QI -> GreenWhiteBackground_AX.QI und PUBLISH_STATE.QI; QI_Output -> DigitalOutput.QI.
5. Qualität: GreenWhiteBackground_AX.QO -> AND_QO.IN1; PUBLISH_STATE.QO -> AND_QO.IN2; AND_QO.OUT -> QO; DigitalOutput.QO -> QO_Output.

6. Daten: `u16ObjId`, `u16ObjIdA -> GreenWhiteBackground_AX`; `Output -> DigitalOutput.Output`; `ID_STATE_WRITE -> PUBLISH_STATE.ID`.

Technische Besonderheiten

- `QI/INIT` gelten nur noch für Hintergrund und Publish; `QI_Output/INIT_Output/INITO_Output/QO_Output` gehören ausschließlich zum `DigitalOutput`.
- Der `AX_SPLIT_3` ist nötig, weil der Adapter-Fanout strikt 1:1 ist.

Anwendungsszenarien

- Kanäle, bei denen sich der physische Ausgang per Option (z. B. Klassisch gegenüber PVEA auf gemeinsamem Pin) abschalten lässt, SoftKey-Hintergrund und State-Publish aber immer laufen sollen.

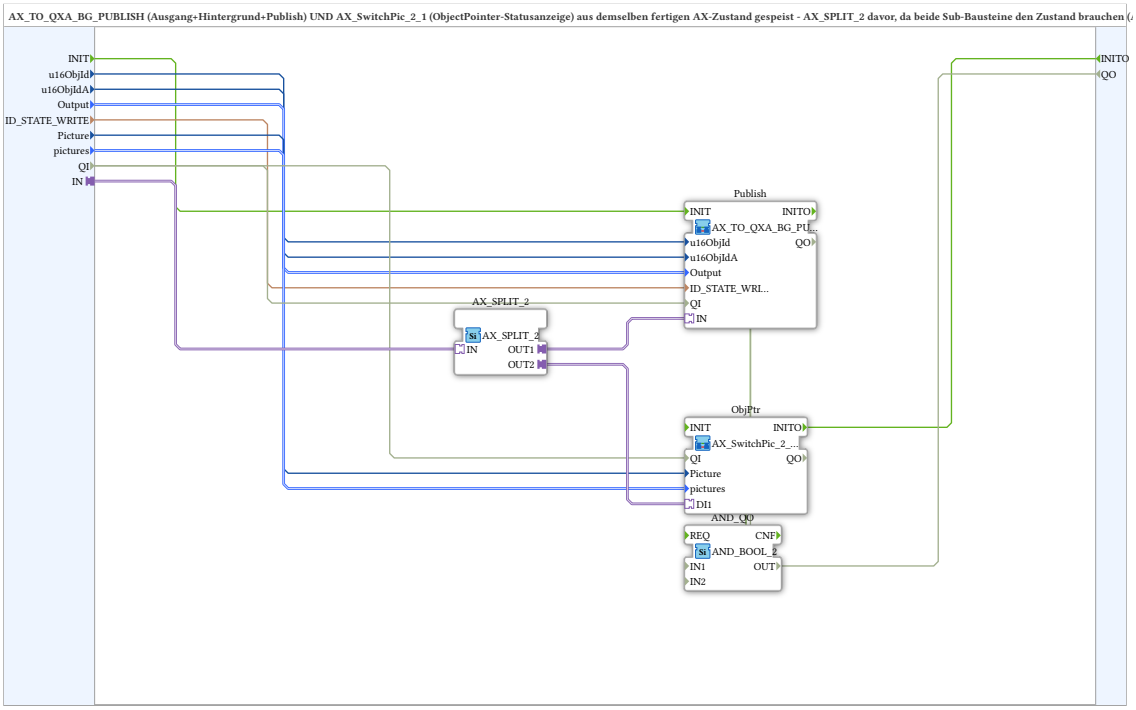
Zusammenfassung

`AX_TO_QXA_BG_PUBLISH_OPT` verteilt einen fertigen AX-Zustand auf Ausgang, Hintergrund und Publish, wobei der physische Ausgang getrennt freigegeben und initialisiert wird.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

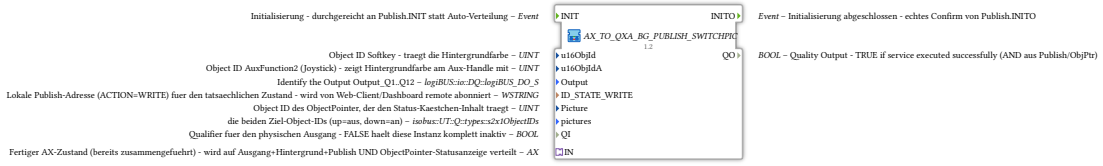
AX_TO_QXA_BG_PUBLISH_SWITCHPIC



• — ○ • — ○ • — ○ •

Einleitung

`AX_TO_QXA_BG_PUBLISH_SWITCHPIC` speist `AX_TO_QXA_BG_PUBLISH` (Ausgang + Hintergrund + Publish) **und** `AX_SwitchPic_2_1` (ObjectPointer-Statusanzeige) aus demselben fertigen AX-Zustand. Ein `AX_SPLIT_2` davor ist nötig, da beide Sub-Bausteine den Zustand brauchen (Adapter-Fanout-Regel). Herausgelöst aus `Baender_EinAus`, wo diese Kombination dreimal identisch (Igel, Elevator, Querband) vorhanden war.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierung; wird an `Publish.INIT` durchgereicht statt per Auto-Verteilung

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierung abgeschlossen; echtes Confirm von `ObjPtr.INITO` (Ende der Kette `Publish` ⟶ `ObjPtr`)

Daten-Eingänge

- **QI** (`BOOL`): Qualifier für den physischen Ausgang; FALSE hält diese Instanz komplett inaktiv (Default `TRUE`)
- **u16ObjId** (`UINT`): Object ID des SoftKeys, trägt die Hintergrundfarbe (Default `ID_NULL`)
- **u16ObjIdA** (`UINT`): Object ID der AuxFunction2 (Joystick), zeigt die Hintergrundfarbe am Aux-Handle mit (Default `ID_NULL`)
- **Output** (`logiBUS::io::DQ::logiBUS_DO_S`): Auswahl des Ausgangs `Output_Q1`..`Output_Q12` (Default `logiBUS_DO::Invalid`)
- **ID_STATE_WRITE** (`WSTRING`): lokale Publish-Adresse (`ACTION=WRITE`) für den tatsächlichen Zustand, wird von Web-Client/Dashboard remote abonniert
- **Picture** (`UINT`): Object ID des ObjectPointers, der den Status-Kästchen-Inhalt trägt
- **pictures** (`isobus::UT::Q::types::s2x1ObjectIDs`): die beiden Ziel-Object-IDs (`up` = aus, `down` = an); Default beide `ID_NULL`

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output, UND aus `Publish.QO` und `ObjPtr.QO`

Adapter

- **IN** (Socket, `adapter::types::unidirectional::AX`): fertiger, bereits zusammengeführter AX-Zustand, wird auf Ausgang + Hintergrund + Publish und die ObjectPointer-Statusanzeige verteilt

Verwendete Funktionsbausteine (FBs)

- **`AX_SPLIT_2`** (`adapter::events::unidirectional::AX_SPLIT_2`)
- **`Publish`** (SubApp `MyLib::sys::AX_TO_QXA_BG_PUBLISH`)
- **`ObjPtr`** (SubApp `MyLib::sys::AX_SwitchPic_2_1`)
- **`AND_QO`** (`iec61131::booleanOperators::AND_BOOL_2`)

Funktionsweise

- `IN` ⟶ `AX_SPLIT_2.IN`
- `AX_SPLIT_2.OUT1` ⟶ `Publish.IN`; `u16ObjId`, `u16ObjIdA`, `Output`, `ID_STATE_WRITE`, `QI` ⟶ gleichnamige Eingänge von `Publish`
- `INIT` ⟶ `Publish.INIT`; `Publish.INITO` ⟶ `ObjPtr.INIT`; `ObjPtr.INITO` ⟶ `INITO` (und ⟶ `AND_QO.REQ`)
- `QI` ⟶ `ObjPtr.QI` (zusätzlich zu `Publish.QI`)
- `Publish.QO` ⟶ `AND_QO.IN1`; `ObjPtr.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`
- `AX_SPLIT_2.OUT2` ⟶ `ObjPtr.DI1`; `Picture` ⟶ `ObjPtr.Picture`, `pictures` ⟶ `ObjPtr.pictures`

Technische Besonderheiten

- Der `AX_SPLIT_2` ist notwendig, weil ein Adapter nicht auf zwei Sub-Bausteine verzweigen darf
- Keine eigene Logik, nur Zusammenfassung zweier vorhandener SubApps
- Version 1.1: `QI` und `INIT` werden an `Publish` (`AX_TO_QXA_BG_PUBLISH` v1.1) durchgereicht (QI-Gate Klassisch-vs-PVEA, Initialisierung vom Aufrufer sequenziert)
- Version 1.2: neuer Ereignis-Ausgang `INITO`, durchgereicht von `Publish.INITO` über `ObjPtr`; zusätzlich `QO` als UND aus `Publish.QO` und `ObjPtr.QO`

Anwendungsszenarien

- Mehrere gleich aufgebaute Kanäle (z. B. Förderbänder), die je einen Ausgang schalten, Softkey-Farbe zeigen, publizieren und zusätzlich eine Status-Bildanzeige umschalten

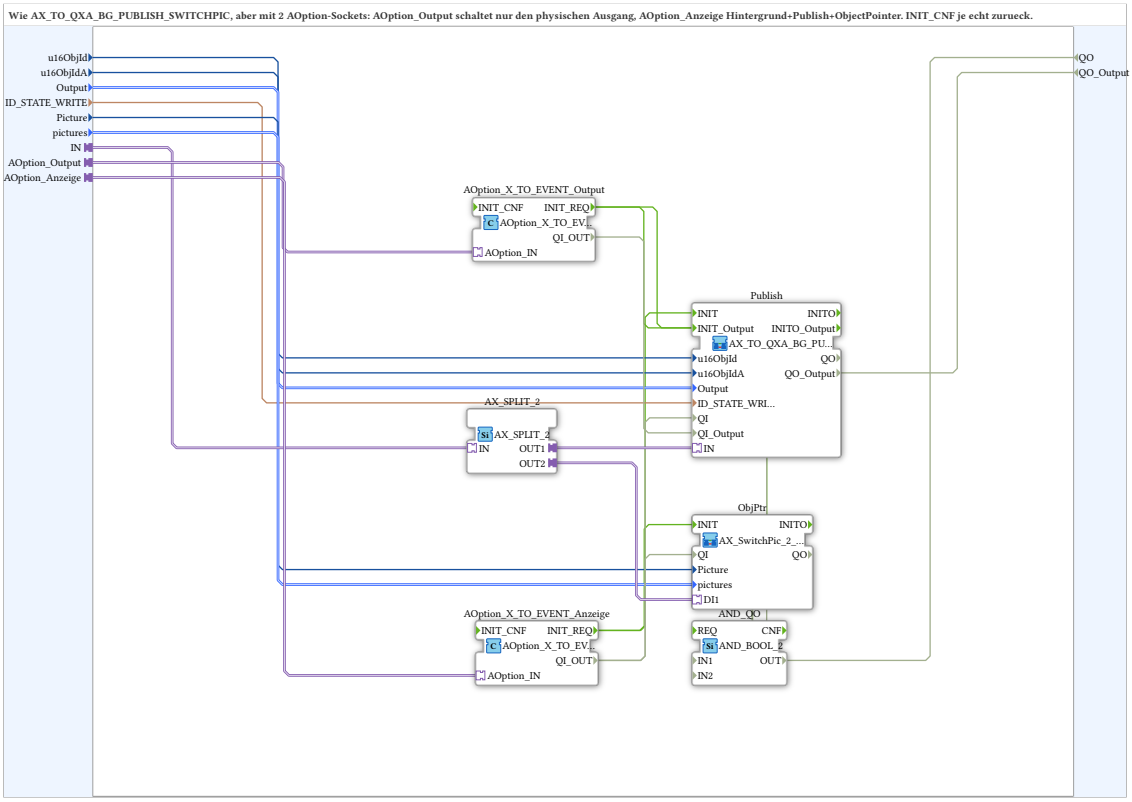
Zusammenfassung

AX_TO_QXA_BG_PUBLISH_SWITCHPIC verteilt einen fertigen AX-Zustand auf Ausgang/Hintergrund/Publish und eine ObjectPointer-Statusanzeige.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

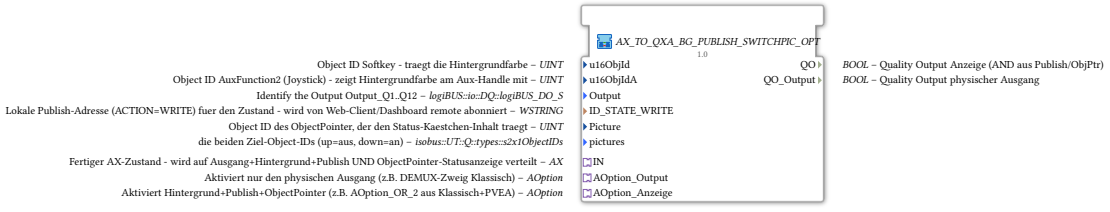
AX_TO_QXA_BG_PUBLISH_SWITCHPIC_OPT



• — ○ • — ○ • — ○ •

Einleitung

AX_TO_QXA_BG_PUBLISH_SWITCHPIC_OPT entspricht AX_TO_QXA_BG_PUBLISH_SWITCHPIC, hat aber statt QI/INIT zwei AOption-Sockets: AOption_Output schaltet nur den physischen Ausgang, AOption_Anzeige schaltet Hintergrund, Publish und ObjectPointer. Das INIT_CNF kommt jeweils echt zurück. Der Baustein ist eine Kopie von AX_TO_QXA_BG_PUBLISH_SWITCHPIC v1.2; Grund war, dass bei der Variante PVEA SoftKey-Hintergrund, ObjectPointer und State-Publish inaktiv blieben. Der Ausgangs-Teil gibt es auch einzeln als AX_TO_QXA_Option, der Anzeige-Teil als AX_TO_BG_PUBLISH_SWITCHPIC_Option.



Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge
Keine vorhanden.

Daten-Eingänge

Variable	Typ	Vorgabe	Kommentar
u16ObjId	UINT	ID_NULL	Object ID SoftKey; trägt die Hintergrundfarbe.
u16ObjIdA	UINT	ID_NULL	Object ID AuxFunction2 (Joystick); zeigt die Hintergrundfarbe am Aux-Handle mit.
Output	logiBUS::io::DQ::logiBUS_DO_S	logiBUS_DO::Invalid	Auswahl des Ausgangs Output_Q1..Output_Q12.
ID_STATE_WRITE	WSTRING		Lokale Publish-Adresse (ACTION=WRITE) für den Zustand; wird von Web-Client/Dashboard remote abonniert.
Picture	UINT		Object ID des Object-Pointers, der den Status-Kästchen-Inhalt trägt.
pictures	isobus::UT::Q::types::s2x1Object	(up := (u16ObjId := ID_NULL), down := (u16ObjId := ID_NULL))	Die beiden Ziel-Object-IDs (up = aus, down = an).

Daten-Ausgänge

Variable	Typ	Vorgabe	Kommentar
Q0	BOOL		Quality Output Anzeige (UND aus Publish.Q0 und ObjPtr.Q0).
Q0_Output	BOOL		Quality Output physischer Ausgang (Publish.Q0_Output).

Adapter

Adapter	Typ	Richtung	Kommentar
IN	adapter::types::unidirectional::AX	Socket	Fertiger AX-Zustand; wird auf Ausgang, Hintergrund, Publish und ObjectPointer-Statusanzeige verteilt.
AOption_Output	Option::AOption	Socket	Aktiviert nur den physischen Ausgang (z. B. DEMUX-Zweig Klassisch).
AOption_Anzeige	Option::AOption	Socket	Aktiviert Hintergrund, Publish und ObjectPointer (z. B. AOption_OR_2 aus Klassisch und PVEA).

Verwendete Funktionsbausteine (FBs)

- **AX_SPLIT_2** (adapter::events::unidirectional::AX_SPLIT_2): Adapter-Fanout (strikt 1:1)
- **AOption_X_TO_EVENT_Output** (Option::conversion::AOption_X_TO_EVENT): Option für den physischen Ausgang

- **AOption_X_TO_EVENT_Anzeige** (`Option::conversion::AOption_X_TO_EVENT`): Option für die Anzeige
- **Publish** (SubApp `MyLib::sys::AX_TO_QXA_BG_PUBLISH_OPT`): Ausgang, Hintergrund und State-Publish; siehe `AX_TO_QXA_BG_PUBLISH_OPT`
- **ObjPtr** (SubApp `MyLib::sys::AX_SwitchPic_2_1`): ObjectPointer-Statusanzeige; siehe `AX_SwitchPic_2_1`
- **AND_QO** (`iec61131::booleanOperators::AND_BOOL_2`): verknüpft die `QO` der Anzeige

Programmablauf und Verbindungen

1. AX-Zustand: `IN` -> `AX_SPLIT_2.IN`; `OUT1` -> `Publish.IN`; `OUT2` -> `ObjPtr.DI1`.
2. Optionen: `AOption_Output` -> `AOption_X_TO_EVENT_Output.AOption_IN`; `AOption_Anzeige` -> `AOption_X_TO_EVENT_Anzeige.AOption_IN`.
3. Initialisierung Anzeige: `AOption_X_TO_EVENT_Anzeige.INIT_REQ` -> `Publish.INIT`; `Publish.INITO` -> `ObjPtr.INIT`; `ObjPtr.INITO` -> `AOption_X_TO_EVENT_Anzeige.INIT_CNF` und -> `AND_QO.REQ`.
4. Initialisierung Ausgang: `AOption_X_TO_EVENT_Output.INIT_REQ` -> `Publish.INIT_Output`; `Publish.INITO_Output` -> `AOption_X_TO_EVENT_Output.INIT_CNF`.
5. Qualifier: `AOption_X_TO_EVENT_Anzeige.QI_OUT` -> `Publish.QI` und `ObjPtr.QI`; `AOption_X_TO_EVENT_Output.QI_OUT` -> `Publish.QI_Output`.
6. Qualität: `Publish.QO` -> `AND_QO.IN1`; `ObjPtr.QO` -> `AND_QO.IN2`; `AND_QO.OUT` -> `QO`; `Publish.QO_Output` -> `QO_Output`.
7. Daten: `u16ObjId`, `u16ObjIdA`, `Output`, `ID_STATE_WRITE` -> gleichnamige Eingänge von `Publish` (`ID_STATE_WRITE` -> `Publish.ID_STATE_WRITE`); `Picture` -> `ObjPtr.Picture`; `pictures` -> `ObjPtr.pictures`.

Technische Besonderheiten

- Der Baustein hat keine eigenen Ereigniseingänge/-ausgänge; die Initialisierung läuft über die beiden `AOption`-Sockets.
- Ausgang und Anzeige lassen sich unabhängig voneinander aktivieren.
- Der `AX_SPLIT_2` ist nötig, weil der Adapter-Fanout strikt 1:1 ist.

Anwendungsszenarien

- Mehrere gleich aufgebaute Kanäle (z. B. Förderbänder) mit Ausgang, SoftKey-Farbe, Publish und Status-Bild, bei denen der Ausgang nur für eine Option (z. B. Klassisch) aktiv sein soll, die Anzeige aber für mehrere Optionen (z. B. Klassisch und PVEA).

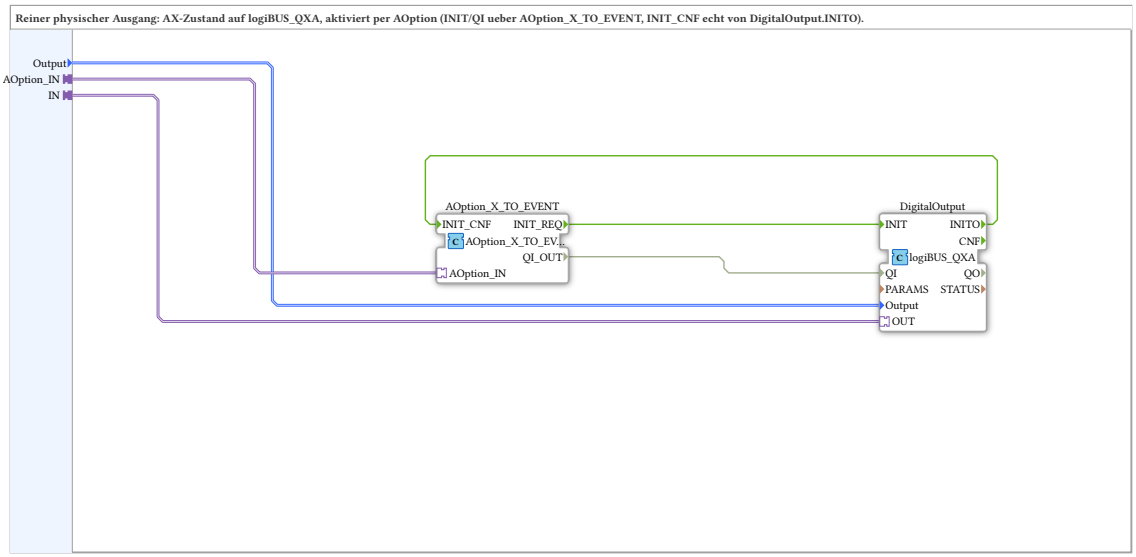
Zusammenfassung

`AX_TO_QXA_BG_PUBLISH_SWITCHPIC_OPT` verteilt einen fertigen AX-Zustand auf Ausgang, Hintergrund, Publish und ObjectPointer-Statusanzeige und aktiviert Ausgang und Anzeige über zwei getrennte `AOption`-Sockets.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AX_TO_QXA_Option



• – ○ • – ○ • – ○ •

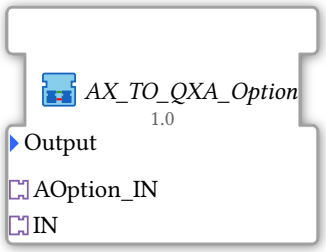
Einleitung

`AX_TO_QXA_Option` ist ein reiner physischer Ausgang: Er bringt einen fertigen AX-Zustand auf `logiBUS_QXA`. Aktiviert wird er per `AOption` (`INIT`/`QI` über `AOption_X_TO_EVENT`, `INIT_CNF` echt von `DigitalOutput.INITO`). Er ist der Ausgangs-Teil von `AX_TO_QXA_BG_PUBLISH_SWITCHPIC_OPT` als eigener Baustein, nach dem Muster von `OptionalGroup_001`.

Identify the Output Output_Q1..Q12 – *logiBUS::io::DQ::logiBUS_DO_S*

Aktiviert/deaktiviert den Ausgang – *AOption*

Fertiger AX-Zustand fuer den Ausgang – *AX*



Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Variable	Typ	Vorgabe	Kommentar
`Output`	`logiBUS::io::DQ::logiBUS_DO_S`	`logiBUS_DO::Invalid`	Auswahl des Ausgangs `Output_Q1..Output_Q12`.

Daten-Ausgänge

Keine vorhanden.

Adapter

Adapter	Typ	Richtung	Kommentar
`AOption_IN`	`Option::AOption`	Socket	Aktiviert/deaktiviert den Ausgang.

IN	adapter::types::unidirectional::AX	Socket	Fertiger AX-Zustand für den Ausgang.

Verwendete Funktionsbausteine (FBs)

- **AOption_X_TO_EVENT** (`Option::conversion::AOption_X_TO_EVENT`): wandelt die Option in `INIT_REQ`/`QI_OUT` und nimmt `INIT_CNF` entgegen
- **DigitalOutput** (`logiBUS::io::DQ::logiBUS_QXA`): physischer Ausgang

Programmablauf und Verbindungen

1. Option: `AOption_IN` -> `AOption_X_TO_EVENT.AOption_IN`.
2. Initialisierung: `AOption_X_TO_EVENT.INIT_REQ -> DigitalOutput.INIT; DigitalOutput.INITO -> AOption_X_TO_EVENT.INIT_CNF`.
3. Qualifier: `AOption_X_TO_EVENT.QI_OUT` -> `DigitalOutput.QI`.
4. AX-Zustand: `IN` -> `DigitalOutput.OUT`.
5. Daten: `Output` -> `DigitalOutput.Output`.
6. Der Baustein hat keine eigenen Ereignis- und `QI`/`QO`-Anschlüsse und keinen Plug.

Technische Besonderheiten

- Keine eigene Logik: nur Option-Anbindung und `logiBUS_QXA`.
- Das `INIT_CNF` kommt echt von `DigitalOutput.INITO`.
- Es gibt keinen `AOption_OUT`-Plug; der Baustein kann daher nicht daisy-chained werden.

Anwendungsszenarien

- Physischer Ausgang, der nur bei ausgewählter Option (z. B. Klassisch) geschaltet werden soll, getrennt von Anzeige und Publish.

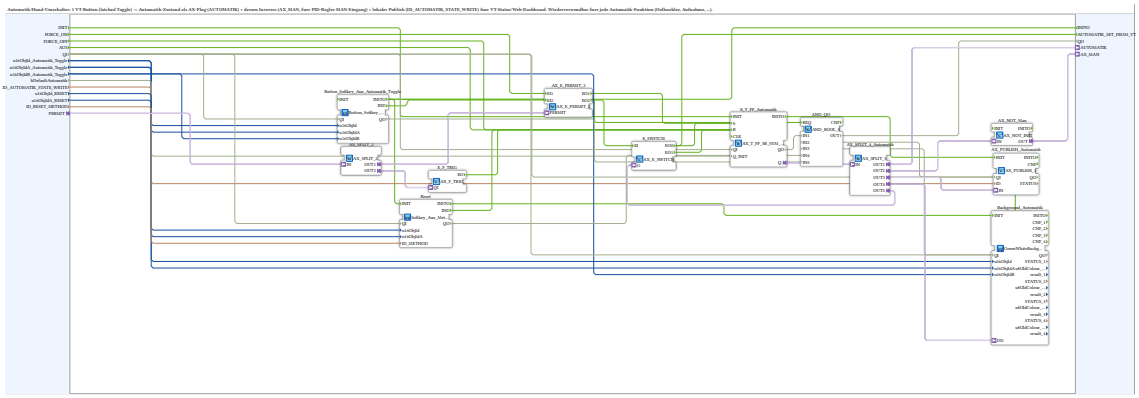
Zusammenfassung

`AX_TO_QXA_Option` schaltet einen physischen Ausgang aus einem fertigen AX-Zustand und wird per `AOption` aktiviert.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Automatik_Toggle_Button



• – ○ • – ○ • – ○ •

Einleitung

`Automatik_Toggle_Button` ist ein **Automatik/Hand-Umschalter**: Ein VT-Button (bzw. Softkey oder AUX-Funktion) schaltet den Automatik-Zustand um. Dieser wird als AX-Plug (`AUTOMATIK`), als Inverses (`AX_MAN`, z. B. für den `MAN`-Eingang eines PID-Reglers) und als lokaler Publish (`ID_AUTOMATIK_STATE_WRITE`) für VT-Status und Web-Dashboard ausgegeben. Die SubApp ist für jede Automatik-Funktion (Ölkühler, Aufnahme, …) wiederverwendbar und wurde aus `Aufnahme_Automatik.SUB` herausgelöst.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialization Request
- **FORCE_ON**: Schaltet die Automatik hart EIN (z. B. von einer übergeordneten Automatik oder einem Methodenaufruf). Läuft über `AX_E_PERMIT_2` und ist damit `PERMIT`-gefiltert. Unverdrahtet lassen, wenn nicht gebraucht.
- **FORCE_OFF**: Schaltet die Automatik hart AUS (z. B. durch einen Kollisionsschutz). Unverdrahtet lassen, wenn nicht gebraucht.
- **AUS**: Reset hart auf Hand/AUS von außen als Event (z. B. ein zentral gelesener gemeinsamer Softkey), zusätzlich zum eigenen `u16ObjId_RESET`/`u16ObjIdA_RESET`/`ID_RESET_METHOD`-Pfad. Unverdrahtet lassen, wenn nicht gebraucht.

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialization Confirm
- **AUTOMATIK_SET_FROM_VT**: Feuert nur bei einem Klick (Button/Softkey/Aux, `PERMIT`-gefiltert), der die Automatik von AUS auf EIN schaltet. Ein Klick, der die Automatik ausschaltet, feuert dieses Ereignis nicht. Es ist ein echtes Display-Klick-Ereignis, kein Zustandspegel oder Republish.

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input, TRUE aktiviert den Dienst
- **u16ObjId_Automatik_Toggle** (`UINT`, Vorgabe `ID_NULL`): Object-ID des VT-Softkeys (Automatik/Hand-Umschalter)
- **u16ObjIdA_Automatik_Toggle** (`UINT`, Vorgabe `ID_NULL`): Object-ID der AuxFunction2 (Joystick), gleichberechtigt zum VT-Button; optional
- **u16ObjIdB_Automatik_Toggle** (`UINT`): Object-ID des VT-Buttons (Automatik/Hand-Umschalter)
- **bDefaultAutomatik** (`BOOL`, Vorgabe `FALSE`): Automatik-Zustand beim Start
- **ID_AUTOMATIK_STATE_WRITE** (`WSTRING`): Lokale Publish-Adresse (`BOOL`, `ACTION=WRITE`) für den Automatik-Ein/Aus-Zustand (Status-Text auf der VT, Web-Client/Dashboard)
- **u16ObjId_RESET** (`UINT`, Vorgabe `ID_NULL`): Object-ID des Softkeys „Alles Aus“ (Reset, hart auf Hand/AUS), z. B. `SoftKey_AUS`
- **u16ObjIdA_RESET** (`UINT`, Vorgabe `ID_NULL`): Object-ID der AuxFunction2 (Joystick) für den Reset
- **ID_RESET_METHOD** (`WSTRING`): Lokale Methodenadresse (`ACTION=CREATE_METHOD`) für den argumentlosen Reset-Trigger; wird von Web-Client und OPC-Dashboard aufgerufen

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output, UND-Verknüpfung (`AND_QO`) der `QO`-Ausgänge von `E_T_FF_Automatik`, `AX_PUBLISH_Automatik`, `Background_Automatik`, `Reset` und `Button_Softkey_Aux_Automatik_Toggle`

Adapter

- **PERMIT** (Socket, `adapter::types::unidirectional::AX`): Freigabe. Gatet das gesamte Ein/Aus; beim Übergang 1⟶0 wird einmalig `RESET` ausgelöst.
- **AUTOMATIK** (Plug, `AX`): Automatik aktiv (roh, z. B. für eine Freigabe-UND-Verknüpfung)
- **AX_MAN** (Plug, `AX`): NICHT Automatik aktiv (für den `MAN`-Eingang eines PID-Reglers)

Verwendete Funktionsbausteine (FBs)

- **Button_Softkey_Aux_Automatik_Toggle**: SubApp `MyLib::sys::Button_Softkey_Aux_IE` – führt Button, Softkey und Aux zu einem `IND` zusammen
- **Reset**: SubApp `MyLib::sys::Softkey_Aux_Method_IE` – Reset per Softkey, Aux oder Methodenaufruf
- **E_T_FF_Automatik**: `AX_T_FF_SR_SYM_INIT` (`QI` vom Wrapper-Eingang) – speichert den Automatik-Zustand, Startwert `bDefaultAutomatik`
- **AX_E_PERMIT_2**: `AX_E_PERMIT_2` – lässt `FORCE_ON` und den Klick nur bei `PERMIT=TRUE` durch
- **AX_SPLIT_2**: verteilt `PERMIT` auf `AX_E_PERMIT_2` und `E_F_TRIG`
- **E_F_TRIG**: `AX_F_TRIG` – fallende Flanke auf `PERMIT`

- **E_SWITCH**: AX_E_SWITCH – filtert den Klick nach dem Zustand vor dem Klick
- **AX_SPLIT_4_Automatik**: AX_SPLIT_5 – verteilt den Zustand auf fünf Ziele
- **AX_NOT_Man**: AX_NOT_INIT – Inverses für AX_MAN
- **AX_PUBLISH_Automatik**: AX_PUBLISH_1 (QI vom Wrapper-Eingang) – publiziert den Zustand unter ID_AUTOMATIK_STATE_WRITE
- **Background_Automatik**: GreenWhiteBackground4_AX – färbt Button und AUX grün/weiß
- **AND_QO**: iec61131::booleanOperators::AND_BOOL_5 – verknüpft die QO-Ausgänge der fünf Bausteine E_T_FF_Automatik, AX_PUBLISH_Automatik, Background_Automatik, Reset und Button_Softkey_Aux_Automatik_Toggle zu QO

Programmablauf und Verbindungen

1. Button_Softkey_Aux_Automatik_Toggle.IND → AX_E_PERMIT_2.EI2; FORCE_ON → AX_E_PERMIT_2.EI1.
2. AX_E_PERMIT_2.EO1 → E_T_FF_Automatik.S (Einschalten über FORCE_ON).
3. AX_E_PERMIT_2.EO2 → E_SWITCH.EI; E_SWITCH.G erhält den Zustand Q vor dem Klick (AX_SPLIT_4_Automatik.OUT5).
   - E_SWITCH.EO0 (AUS → EIN) → AUTOMATIK_SET_FROM_VT und E_T_FF_Automatik.S
   - E_SWITCH.EO1 (EIN → AUS) → E_T_FF_Automatik.R
4. Reset.IND, AUS, FORCE_OFF und E_F_TRIG.EO → E_T_FF_Automatik.R.
5. PERMIT → AX_SPLIT_2 → AX_E_PERMIT_2.PERMIT und E_F_TRIG.QI.
6. E_T_FF_Automatik.Q → AX_SPLIT_4_Automatik: OUT1 → AUTOMATIK, OUT2 → AX_NOT_Man → AX_MAN, OUT3 → AX_PUBLISH_Automatik, OUT4 → Background_Automatik.DI1, OUT5 → E_SWITCH.G.
7. Initialisierungskette: INIT → E_T_FF_Automatik.INIT → AX_PUBLISH_Automatik.INIT → Background_Automatik.INIT → Reset.INIT → Button_Softkey_Aux_Automatik_Toggle.INIT; Button_Softkey_Aux_Automatik_Toggle.INITO → INITO und → AND_QO.REQ.
8. Qualität: QI → .QI von E_T_FF_Automatik, AX_PUBLISH_Automatik, Background_Automatik, Reset und Button_Softkey_Aux_Automatik_Toggle; deren QO → AND_QO.IN1 bis AND_QO.IN5 (in dieser Reihenfolge); AND_QO.OUT → QO.

Technische Besonderheiten

- **Button statt Softkey**: Der VT-Button muss non-latching (Latchable=0) sein; das Feedback der Automatik-Anzeige übernimmt Background_Automatik.
- **PERMIT** gatet das gesamte Ein/Aus und setzt beim Übergang 1→0 einmalig zurück – es wirkt dabei wie ein AX_PERMIT.
- FORCE_ON/FORCE_OFF/AUS sind echte Ereignisse; Aufrufer mit pegelbasierten Quellen müssen die Flankenerkennung selbst vorschalten.
- AUTOMATIK_SET_FROM_VT reagiert nur auf den Display-Klick AUS → EIN, nicht auf FORCE_ON-Echos.

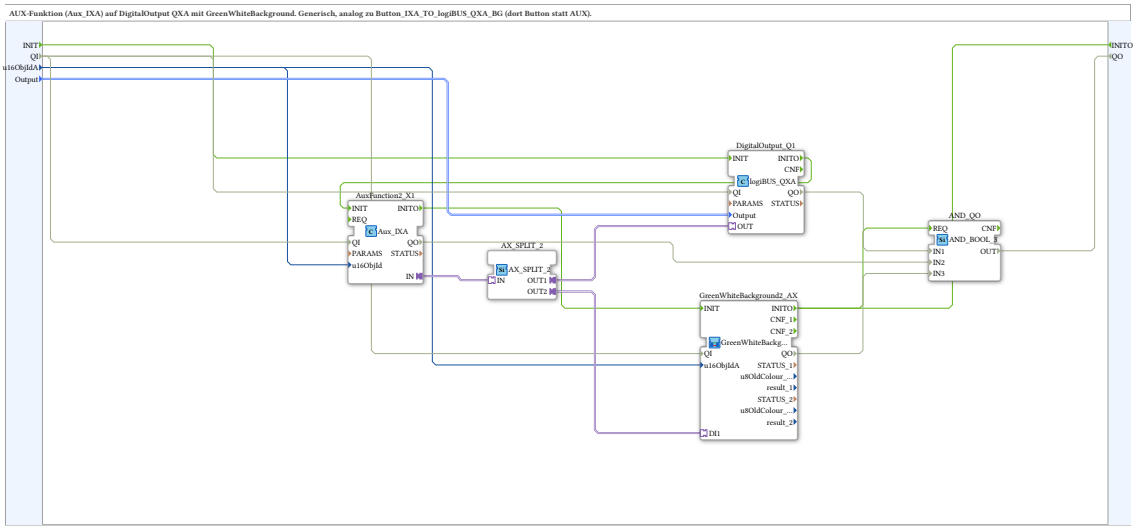
Anwendungsszenarien

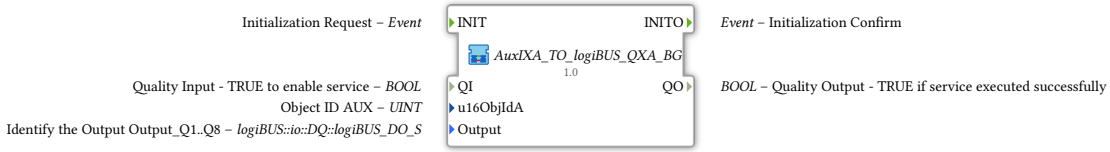
- Automatik/Hand-Umschaltung mit Statusanzeige für Automatik-Funktionen (Ölkühler, Aufnahme, Bunker-Ernte, …)
- Handbetrieb-Signal für PID-Regler (AX_MAN)

Fazit

Automatik_Toggle_Button kapselt Bedienung, Zustandsspeicher, Freigabe, Reset und Statusausgabe einer Automatik-Funktion in einer wiederverwendbaren SubApp.

AuxIXA_TO_logiBUS_QXA_BG





• – ○ • – ○ • – ○ •

Einleitung

Der Subapplikationsbaustein **AuxIXA_TO_logiBUS_QXA_BG** realisiert eine AUX-Funktion (Aux_IXA) auf einem digitalen Ausgang (QXA) mit Green-White-Hintergrunddarstellung. Er ist generisch aufgebaut und analog zu dem Baustein `Button_IXA_TO_logiBUS_QXA_BG` implementiert, wobei hier statt eines Buttons eine AUX-Eingangsfunktion verwendet wird. Der Baustein wurde aus einer Übungseinheit ausgelagert, um eine Wiederverwendung in verschiedenen Projekten zu ermöglichen. Er kombiniert eine ISO-bus-konforme AUX-Schnittstelle mit einem logiBUS-Digitalausgang und einer visuellen Statusanzeige.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialization Request – startet die Initialisierungskette der internen Bausteine.

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialization Confirm – wird nach Abschluss der Initialisierungskette ausgegeben.

Die Signalverarbeitung selbst erfolgt weiterhin daten- und adaptergesteuert.

Daten-Eingänge

Name	Datentyp	Initialwert	Kommentar
`QI`	`BOOL`	`TRUE`	Quality Input - TRUE, um den Dienst freizugeben
`u16ObjIdA`	`UINT`	`ID_NULL`	Object ID der AUX-Funktion (z. B. ISOBUS-Objekt-ID)
`Output`	`logiBUS::io::DQ::logiBUS_DO_S`	`logiBUS_DO::Invalid`	Identifiziert den Ausgang (Output_Q1...Q8)

Daten-Ausgänge

Name	Datentyp	Kommentar
QO	BOOL	Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde

QO ist die UND-Verknüpfung (AND_QO, Typ iec61131::booleanOperators::AND_BOOL_3) der QO-Ausgänge von DigitalOutput_Q1, AuxFunction2_X1 und GreenWhiteBackground2_AX. Die Ausgabe an den Ausgang erfolgt über den internen Funktionsblock DigitalOutput_Q1 und die Adapterverbindungen.

Adapter

Der Baustein besitzt über die interne Verdrahtung Adapterverbindungen, die jedoch nicht als öffentliche Schnittstellen nach außen geführt werden. Die internen Adapterverbindungen dienen der Kommunikation zwischen der AUX-Eingangsfunktion, dem Digitalausgang und der Statusanzeige.

Funktionsweise

Der Baustein nimmt eine AUX-Objekt-ID (u16ObjIdA) und eine Ausgangsauswahl (Output) entgegen. Intern wird die AUX-Eingangsfunktion AuxFunction2_X1 (Typ isobus::UT::io::Auxiliary::IN::Aux_IXA) mit der übergebenen Objekt-ID versorgt. Deren Ereignis-/Adapterausgang (IN) wird über einen Ereignis-Splitter (AX_SPLIT_2) auf zwei Pfade aufgeteilt:

1. **Pfad 1** – führt zum Funktionsblock DigitalOutput_Q1 (Typ logiBUS::io::DQ::logiBUS_QXA), der den digitalen Ausgang gemäß der Auswahl Output ansteuert.
2. **Pfad 2** – führt zum Subapplikationsbaustein GreenWhiteBackground2_AX (Typ MyLib::sys::GreenWhiteBackground2_AX), der die grafische Darstellung (grün/weißer Hintergrund) für die Statusanzeige übernimmt.

Die übergebene Objekt-ID u16ObjIdA wird ebenfalls an GreenWhiteBackground2_AX weitergeleitet, um eine konsistente Darstellung sicherzustellen. Der Ausgang Output wird direkt an den Digitalausgangsbaustein übergeben, welcher den tatsächlichen physischen Ausgang (Q1…Q8) des logiBUS-Systems aktiviert.

Initialisierung und Status: INIT → DigitalOutput_Q1.INIT, DigitalOutput_Q1.INITO → AuxFunction2_X1.INIT, AuxFunction2_X1.INITO → GreenWhiteBackground2_AX.INIT, GreenWhiteBackground2_AX.INITO → INITO (und zusätzlich → AND_QO.REQ). QI wird an DigitalOutput_Q1, AuxFunction2_X1 und GreenWhiteBackground2_AX weitergegeben; deren QO laufen auf AND_QO.IN1..IN3, AND_QO.OUT → QO.

Durch die Aufteilung des Ereignisses in zwei Zweige kann sowohl die digitale Ausgabe als auch die visuelle Rückmeldung parallel erfolgen, ohne dass eine separate Logik erforderlich ist.

Technische Besonderheiten

- **Generische Wiederverwendung**: Der Baustein ist modular aufgebaut und kann in verschiedenen Steuerungsanwendungen eingesetzt werden, bei denen eine AUX-Funktion auf einen logiBUS-Digitalausgang abgebildet werden soll.
- **Kombination von ISOBUS und logiBUS**: Er verbindet die ISOBUS-AUX-Schnittstelle (Standard 61499-2) mit der logiBUS-Ausgangssteuerung.
- **Visuelle Statusanzeige**: Durch die Einbindung des Bausteins GreenWhiteBackground2_AX wird eine farbliche Rückmeldung (grün/weiß) auf einem Bedienpanel oder einer Visualisierung ermöglicht.
- **Parametrierung**: Der Funktionsblock DigitalOutput_Q1 erhält QI vom QI-Eingang des Bausteins (Vorgabe TRUE) und akzeptiert über eine unsichtbare PARAMS-Konfiguration zusätzliche Parameter.
- **Interner Ereignis-Splitter**: Der AX_SPLIT_2 (adapter::events::unidirectional) verteilt das eingehende Ereignis ohne Verzögerung an zwei unabhängige Zielbausteine.

Zustandsübersicht

Da der Baustein keine eigenen Zustandsautomaten besitzt, hängt sein Verhalten von den internen Bausteinen ab. Im Wesentlichen gibt es zwei Betriebszustände:

- **Inaktiv**: Keine gültige AUX-Objekt-ID oder kein aktivierter Ausgang – der Digitalausgang ist deaktiviert, die Statusanzeige zeigt den Neutralzustand.
- **Aktiv**: Eine gültige AUX-Objekt-ID liegt an und das Ereignis aktiviert den Ausgang – der Digitalausgang schaltet den entsprechenden Kanal, die Statusanzeige wechselt auf die aktive Farbe (grün).

Der Übergang zwischen den Zuständen erfolgt ereignisgesteuert über die Adapterschnittstelle des Aux_IXA-Bausteins.

Anwendungsszenarien

- **Landmaschinensteuerung**: Abbildung einer AUX-Funktion (z. B. „Hebearm anheben“) auf einen digitalen Ausgang eines logiBUS-Moduls zur Ansteuerung eines Ventils oder Relais.
- **Visualisierung**: Anzeige des Schaltzustands auf einem Terminal mit grün/weißer Hintergrundbeleuchtung für Klarheit und Benutzerfreundlichkeit.
- **Modulare Automatisierung**: Wiederverwendbarkeit in verschiedenen Maschinen, ohne jedes Mal die Logik neu zu programmieren.
- **Test- und Simulationsumgebungen**: Einsatz in Laboraufbauten zur Verifikation von ISOBUS-zu-logiBUS-Verbindungen.

Vergleich mit ähnlichen Bausteinen

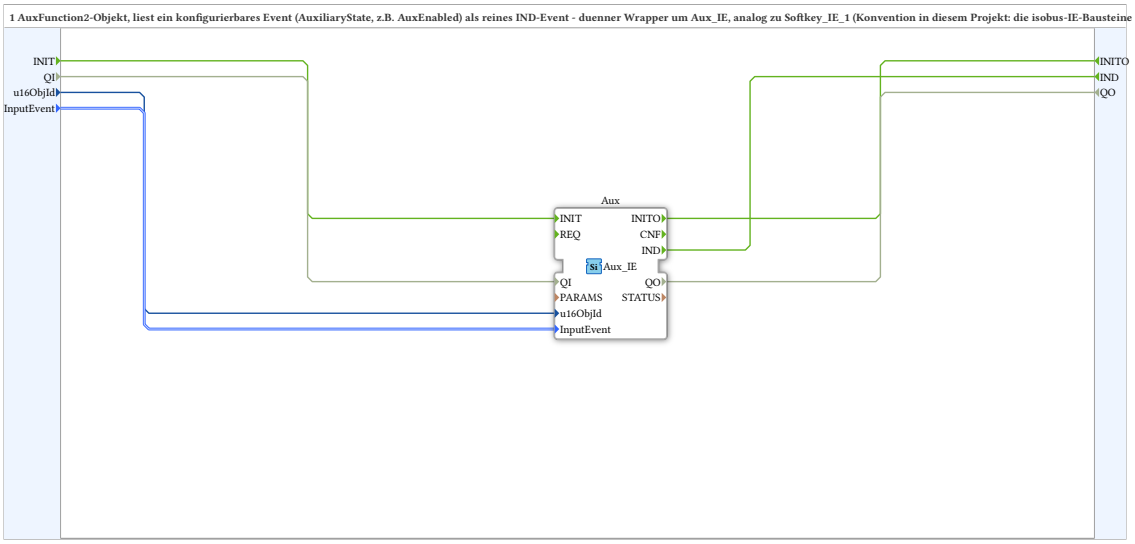
Der Baustein ist konzeptionell identisch mit `Button_IXA_TO_logiBUS_QXA_BG`, unterscheidet sich jedoch in der Quelle des Eingangsereignisses: Hier wird ein AUX-Eingang (`Aux_IXA`) statt eines Button-Eingangs verwendet. Der AUX-Eingang basiert auf ISOBUS-Standard, während ein Button typischerweise ein binäres Signal liefert. Dadurch unterstützt dieser Baustein komplexere, objektorientierte Steuerbefehle (über Objekt-IDs), während der Button-Baustein einfacher aufgebaut ist. Beide teilen sich die Ausgangslogik und die Statusanzeige, was die Wartbarkeit erhöht.

Andere mögliche Bausteine, z. B. direkte Treiber ohne Visualisierung, verzichten auf die Green-WhiteBackground-Subapplikation und sind dementsprechend kompakter, bieten aber keine optische Rückmeldung.

Fazit

Der Baustein `AuxIXA_TO_logiBUS_QXA_BG` ist ein gut durchdachter, modularer Baustein, der eine ISOBUS-AUX-Funktion nahtlos in ein logiBUS-Ausgangssystem integriert. Durch die Kombination mit einer visuellen Statusanzeige erhöht er die Bedienbarkeit und Transparenz in Steuerungssystemen. Seine generische Struktur ermöglicht eine einfache Wiederverwendung und Anpassung an unterschiedliche Anforderungen. Die saubere Trennung von Eingangslogik, Ausgangstreiber und Anzeige macht ihn wartungsfreundlich und erweiterbar.

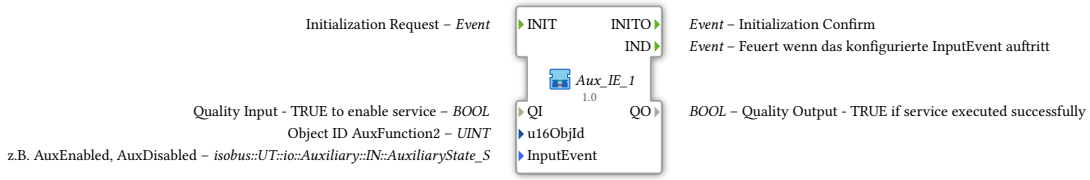
Aux_IE_1



• – ◦ • – ◦ • – ◦ •

Einleitung

`Aux_IE_1` liest ein AuxFunction2-Objekt und gibt ein konfigurierbares Ereignis (`AuxiliaryState`, z. B. `AuxEnabled`) als reines `IND`-Ereignis aus. Der Baustein ist ein dünner Wrapper um `isobus::UT::io::Auxiliary::IN::Aux_IE`, analog zu `Softkey_IE_1` und `Button_IE_1`. Konvention in diesem Projekt: Die isobus-IE-Bausteine werden nie roh inline verwendet, sondern immer gewrappt.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; wird an `Aux.INIT` durchgereicht

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung; kommt von `Aux.INITO`
- **IND**: feuert, wenn das konfigurierte `InputEvent` auftritt

Daten-Eingänge

- **QI** (BOOL): Quality Input, Freigabe des Dienstes (Initialwert `TRUE`), wird an `Aux.QI` durchgereicht
- **u16ObjId** (UINT): Object ID der AuxFunction2 (Default `ID_NULL`)
- **InputEvent** (`isobus::UT::io::Auxiliary::IN::AuxiliaryState_S`): z. B. `AuxEnabled`, `AuxDisabled` (Default `Invalid`)

Daten-Ausgänge

- **QO** (BOOL): Quality Output, kommt von `Aux.QO`

Adapter

Es werden keine Adapter verwendet.

Funktionsweise

Das Netzwerk enthält genau eine Instanz `Aux` des Typs `isobus::UT::io::Auxiliary::IN::Aux_IE`. `u16ObjId` und `InputEvent` werden unverändert auf die gleichnamigen Eingänge der Instanz geführt, und deren Ereignis `IND` wird direkt als `IND` der SubApp ausgegeben. Zusätzlich werden `INIT` ⟶ `Aux.INIT`, `Aux.INITO` ⟶ `INITO`, `QI` ⟶ `Aux.QI` und `Aux.QO` ⟶ `QO` durchgereicht.

Technische Besonderheiten

- Reine Durchreichung ohne eigene Logik oder Zustand
- `QI` ist nicht mehr fest auf TRUE verdrahtet, sondern wird zusammen mit `INIT`/`INITO` und `QO` durchgereicht (Initialwert von `QI` ist `TRUE`)
- Weil `InputEvent` ein Parameter ist, kann dieselbe SubApp für unterschiedliche Ereignisarten (z. B. Drücken oder Loslassen) verwendet werden

Anwendungsszenarien

- Einlesen einer einzelnen VT-Bedienung als Ereignis, ohne den rohen IE-Baustein direkt im Netzwerk zu verwenden

⚖️ Vergleich mit ähnlichen Bausteinen

Gleichartige Wrapper sind `Button_IE_1` und `Softkey_IE_1` sie unterscheiden sich nur im VT-Objekttyp und im Aufzählungstyp des `InputEvent`.

Fazit

`Aux_IE_1` ist ein minimaler Wrapper, der ein konfigurierbares AuxFunction2-Ereignis als `IND` bereitstellt.

Background-Farbbausteine: Gemeinsames Muster

• – ◦ • – ◦ • – ◦ •

Einleitung

`MyLib::sys` enthält in beiden Trainingssystemen (`test_AX` und `test_B`) eine große Familie fast identischer Bausteine, die **die VT-Hintergrundfarbe eines oder mehrerer Objekte anhand eines einzigen**

booleschen Signals umschalten — z. B. `GreenWhiteBackground1_AX`, `GreenRedBackground4_AXS`, `RedWhiteBackground2_AXC`. Diese Seite erklärt das gemeinsame Muster einmal vollständig; die einzelnen Bausteinseiten verweisen hierher zurück und nennen nur noch die konkreten Abweichungen (Farbpaar, Objektanzahl, Variante).

Namensschema

`<Farbe1><Farbe2>Background<N>[_aux][_AX][S][C]`

Bestandteil	Bedeutung
Farbpaar (`GreenWhite`, `GreenRed`, `GreenBlue`, `RedGreen`, `RedWhite`)	`TRUE` → erste Farbe, `FALSE` → zweite Farbe (z. B. `GreenWhiteBackground`: TRUE→Grün, FALSE→Weiß)
N (1–4)	Anzahl der VT-Objekte, die **gleichzeitig aus demselben Selector-Bit** eingefärbt werden (nicht: Anzahl unabhängiger Kanäle)
`_aux` (nur bei N=1)	Nutzt `Q_BackgroundColourAux` statt `Q_BackgroundColour` — zielt auf ein ISOBUS-**Auxiliary-Function**-Objekt statt ein normales Softkey-/Button-/DataMask-Objekt. Bei N≥2-Varianten ist diese Rolle ggf. bereits fest in eine der Positionen eingebaut (siehe Einzelseite).
`_AX` (nur test_AX)	Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`), nicht als reine Dateneingang. In test_B fehlt dieses Suffix — dort ist `DI1` ein einfacher `BOOL`-Dateneingang.
`S`	Objekt-ID(s) werden über den strukturierten Typ `isobus::UT::Q::types::s1ObjectID` (`u16ObjIds`, intern per `F_MOVE` entpackt) übergeben statt über einzelne `UINT u16ObjId`-Eingänge — eine spätere Vereinheitlichung der Objekt-ID-Übergabe in der Bibliothek.
`C` („Kompakt“)	Dünner Wrapper, der nur die Basisvariante (`_AX` bzw. `_AXS`) instanziiert und die `CNF`-Ereignisausgänge/Zwischenwerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nicht nach außen führt — für den Regelfall, wo diese Diagnosewerte nicht gebraucht werden.

Beispiele: `GreenWhiteBackground1_AX` (Basis, 1 Objekt, Adapter), `GreenWhiteBackground1_AXC` (Kompakt-Wrapper davon), `GreenWhiteBackground1_AXS` (mit Struct-Objekt-ID), `GreenWhiteBackground1_AXSC` (Struct-ID + Kompakt), `GreenWhiteBackground1_aux_AX` (wie Basis, aber AUX-Funktion-Objekt).

Funktionsweise (Basisvariante, N=1)

1. Das Selector-Signal (`DI1`-Adapter bei `_AX`, sonst `DI1`-Datenwert) geht an `AX_SEL`/`F_SEL` (Binärauswahl), parametriert mit den beiden Farbkonstanten (`IN0`=zweite Farbe, `IN1`=erste Farbe, z. B. `IN0=COLOR_WHITE`, `IN1=COLOR_GREEN`).
2. `AX_SEL.CNF`/`F_SEL.CNF` löst `Q_BackgroundColour.REQ` (bzw. `Q_BackgroundColourAux.REQ` bei `_aux`) aus.
3. `Q_BackgroundColour` setzt die Hintergrundfarbe des per `u16ObjId` identifizierten VT-Objekts und meldet `CNF` mit `STATUS`, `u8OldColour` (vorherige Farbe) und `s16result` (Fehlercode) zurück.

Funktionsweise (N≥2)

Bei mehreren Objekten (`GreenRedBackground4_AX` u. ä.) gibt es weiterhin **nur einen** Selector (`DI1`/`AX_SEL`/`F_SEL`), dessen Ausgang parallel an mehrere `Q_BackgroundColour_n`-Instanzen (nummeriert `_1`..`_N`) verteilt wird — jede mit eigenem u16ObjId/u16ObjIdA/u16ObjIdB(unterschiedliche Objekt-IDs, teils für unterschiedliche Objektrollen wie

`Softkey/AUX/Button) und eigenen`STATUS_n/u8OldColour_n/result_n/CNF_n`-Ausgängen. **Welche Position` Q_BackgroundColourAux`statt`Q_BackgroundColour` nutzt, ist pro Baustein individuell** (nicht immer dieselbe Position) — die Einzelseite jedes Bausteins nennt die konkrete Zuordnung.

Technische Besonderheiten

- **Ein Selector, mehrere Ziele**: Der Sinn von N>1 ist, dieselbe boolesche Bedingung (z. B. „Kanal aktiv“) gleichzeitig auf mehrere VT-Repräsentationen desselben logischen Zustands anzuwenden (z. B. Softkey-Hintergrund UND Auxiliary-Function-Hintergrund UND Button-Hintergrund derselben Funktion).
- **`_aux/Q_BackgroundColourAux`**: ISOBUS-Auxiliary-Function-Objekte (Bedienelemente, die dem Fahrer frei zuweisbar sind) benötigen einen eigenen Q-Baustein, da sie einen anderen ObjectID-Bereich/eine andere VT-Nachricht verwenden als normale Softkeys/Buttons.
- **`S`-Variante als spätere Vereinheitlichung**: Erkennbar an den späteren `VersionInfo`-Daten (2026 vs. 2022) — die `S`-Varianten sind ein Nachzügler-Refactoring, das die Objekt-ID-Übergabe an einen später etablierten Standard (`sIObjectID`) angleicht, ohne die älteren Basisvarianten zu ersetzen.

Übersicht der Familie

Farbpaar	test_AX (Basis/_aux, jeweils auch S/SC/C)	test_B (Basis/_aux, jeweils auch S/SC/C)
Green/White	1, 2, 3, 4	1, 2, 3, 4
Green/Red	1, 2, 3, 4	1, 2, 3, 4
Green/Blue	1	1
Red/Green	1, 4	1, 4
Red/White	1, 2, 3, 4	1, 2, 3, 4

In test_AX trägt jede Kombination zusätzlich das Suffix `_AX`; in test_B entfällt es (Selector als `BOOL`-Dateneingang statt Adapter-Socket) — siehe MyLib (test_B) → sys.

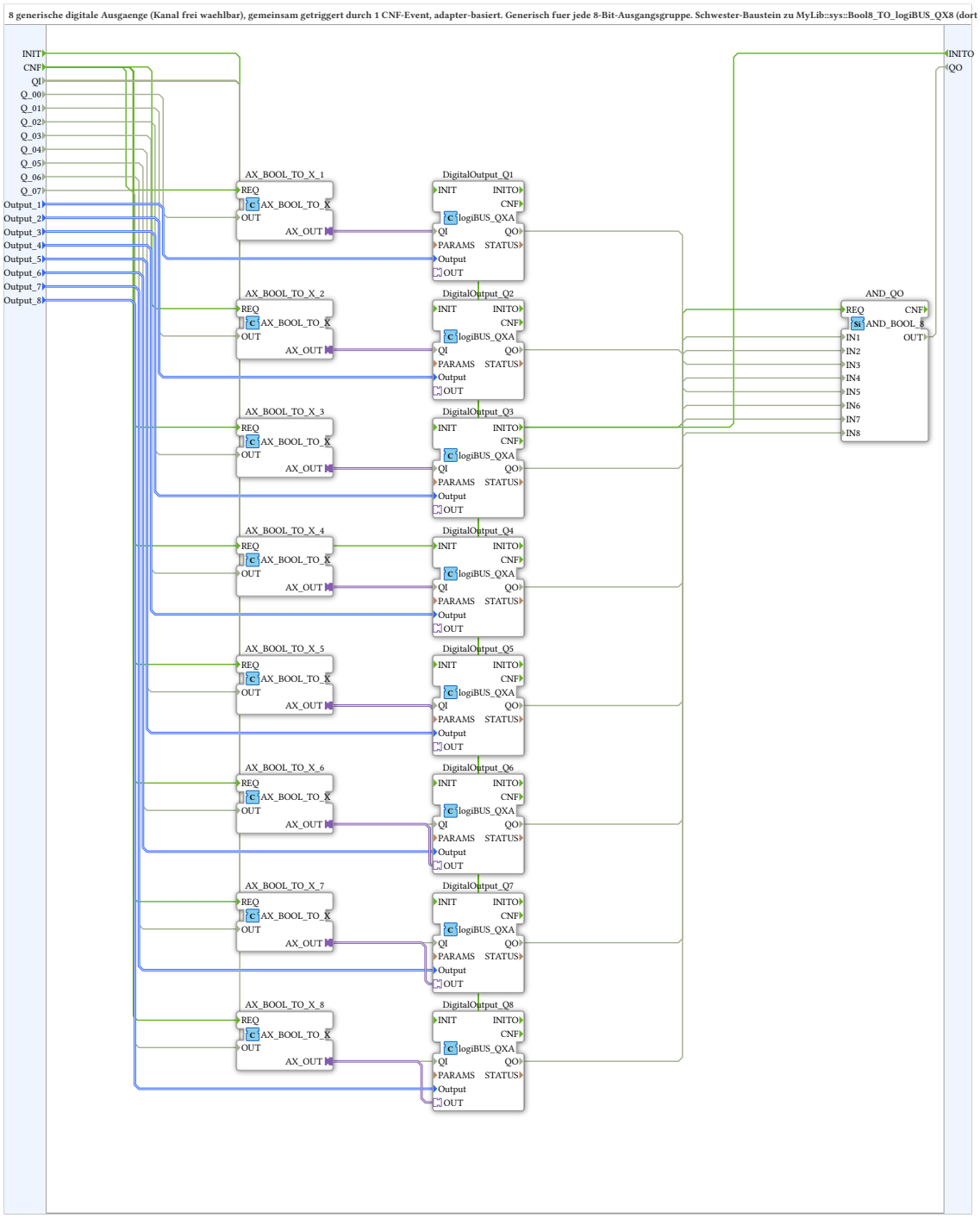
Zusammenfassung

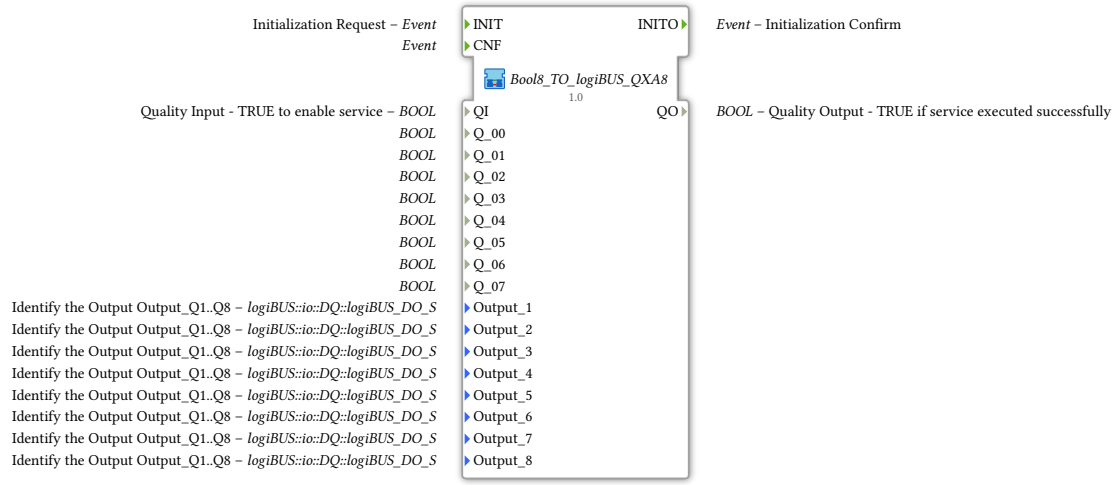
Die Background-Farbbausteine sind eine einzige wiederkehrende Idee (boolesches Signal → Farbauswahl → `Q_BackgroundColour`) in vielen kleinen Varianten für unterschiedliche Farbpaare, Objektanzahlen und Objekt-ID-Konventionen. Wer eine Variante versteht, versteht alle — die Einzelseiten nennen nur noch die konkreten Parameter.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Bool8_TO_logiBUS_QXA8





• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **Bool8_TO_logiBUS_QXA8** ist eine Subapplikation zur Ansteuerung von acht digitalen Ausgängen über einen logiBUS-Bus. Er stellt eine generische, adapterbasierte Lösung dar, bei der die Zielkanäle pro Ausgang frei konfiguriert werden können. Der Baustein wird über ein gemeinsames Ereignis (CNF) getriggert und überträgt die anliegenden Bool-Werte auf die spezifizierten Ausgänge. Er ist als Schwester-Baustein zu `MyLib::sys::Bool8_TO_logiBUS_QX8` konzipiert und unterscheidet sich durch die Verwendung des QXA-Adaptertyps.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; startet die Init-Kette bei `DigitalOutput_Q4.INIT`.
- **CNF** (Event): Dieses Ereignis löst die Übernahme aller acht Eingangswerte und deren Übertragung auf die konfigurierten Ausgänge aus.

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung; kommt von `DigitalOutput_Q3.INITO`, dem Ende der Init-Kette.

Daten-Eingänge

- **QI** (BOOL, Initialwert `TRUE`): Quality Input, schaltet den Dienst frei; wird an alle acht `DigitalOutput_Qx.QI` weitergereicht.
- **Q_00** (BOOL): Binärwert für Ausgang 1.
- **Q_01** (BOOL): Binärwert für Ausgang 2.
- **Q_02** (BOOL): Binärwert für Ausgang 3.
- **Q_03** (BOOL): Binärwert für Ausgang 4.
- **Q_04** (BOOL): Binärwert für Ausgang 5.
- **Q_05** (BOOL): Binärwert für Ausgang 6.
- **Q_06** (BOOL): Binärwert für Ausgang 7.
- **Q_07** (BOOL): Binärwert für Ausgang 8.
- **Output_1** (logiBUS::io::DQ::logiBUS_DO_S): Kanalkonfiguration für Ausgang 1 (Initialwert: `logiBUS_DO::Invalid`).
- **Output_2** (logiBUS::io::DQ::logiBUS_DO_S): Kanalkonfiguration für Ausgang 2 (Initialwert: `logiBUS_DO::Invalid`).
- **Output_3** (logiBUS::io::DQ::logiBUS_DO_S): Kanalkonfiguration für Ausgang 3 (Initialwert: `logiBUS_DO::Invalid`).
- **Output_4** (logiBUS::io::DQ::logiBUS_DO_S): Kanalkonfiguration für Ausgang 4 (Initialwert: `logiBUS_DO::Invalid`).
- **Output_5** (logiBUS::io::DQ::logiBUS_DO_S): Kanalkonfiguration für Ausgang 5 (Initialwert: `logiBUS_DO::Invalid`).

- **Output_6** (logiBUS::io::DQ::logiBUS_DO_S): Kanalkonfiguration für Ausgang 6 (Initialwert: `logiBUS_DO::Invalid`).
- **Output_7** (logiBUS::io::DQ::logiBUS_DO_S): Kanalkonfiguration für Ausgang 7 (Initialwert: `logiBUS_DO::Invalid`).
- **Output_8** (logiBUS::io::DQ::logiBUS_DO_S): Kanalkonfiguration für Ausgang 8 (Initialwert: `logiBUS_DO::Invalid`).

Daten-Ausgänge

- **QO** (BOOL): Quality Output, UND-Verknüpfung (`AND_QO`, `AND_BOOL_8`) der `QO`-Ausgänge aller acht `DigitalOutput_Qx`.

Adapter

An der Subapp-Schnittstelle sind keine Adapterports definiert. Intern werden jedoch Adapterverbindungen zwischen den Konvertern und den logiBUS-Ausgangsbausteinen verwendet.

Funktionsweise

Die Subapplikation enthält acht Instanzen des Funktionsblocks `logiBUS::io::DQ::logiBUS_QXA` (jeweils für einen Ausgang) sowie acht Konverter `adapter::conversion::unidirectional::AX_BOOL_TO_X`. Beim Eintreten des Ereignisses `CNF` werden alle acht Bool-Werte parallel über die Konverter in Adapterdaten umgewandelt und an die zugehörigen `logiBUS_QXA`-Bausteine weitergeleitet. Diese setzen die digitalen Ausgänge entsprechend dem konfigurierten Kanal (über die `Output_x`-Variablen) auf den jeweiligen Binärzustand. Die Auswahl der Zielkanäle erfolgt über die Dateneingänge `Output_1` bis `Output_8`, die als logiBUS-Kanal-Identifikatoren dienen.

Initialisierung und Qualität: `INIT` wird durch alle acht `logiBUS_QXA`-Instanzen gekettet (`INIT` → `DigitalOutput_Q4` → `DigitalOutput_Q7` → `DigitalOutput_Q1` → `DigitalOutput_Q5` → `DigitalOutput_Q2` → `DigitalOutput_Q6` → `DigitalOutput_Q8` → `DigitalOutput_Q3`, jeweils `INITO` → nächstes `INIT`); `DigitalOutput_Q3.INITO` führt zu `INITO` und `AND_QO.REQ`. `QI` wird an alle acht `DigitalOutput_Qx.QI` verteilt; deren `QO`-Ausgänge gehen auf `AND_QO.IN1` bis `AND_QO.IN8` (`iec61131::booleanOperators::AND_BOOL_8`), dessen `OUT` den Ausgang `QO` bildet.

Technische Besonderheiten

- **Adapterbasierte Konvertierung:** Die Umwandlung von BOOL in adapter-kompatible Daten erfolgt über die Bausteine `AX_BOOL_TO_X`, die eine unidirektionale Konvertierung ermöglichen.
- **Parallele Verarbeitung:** Alle acht Ausgänge werden gleichzeitig (parallel) bedient, daher ist die Reaktionszeit unabhängig von der Anzahl der Ausgänge.
- **Flexible Kanalzuordnung:** Durch die externen Variablen `Output_1` bis `Output_8` kann jeder Ausgang separat einem beliebigen logiBUS-Kanal zugeordnet werden.
- **Wiederverwendbarkeit:** Als Subapplikation lässt sich der Baustein in verschiedene Projekte einbinden und durch die Parametrierung der Ausgangskanäle universell einsetzen.

Zustandsübersicht

Da es sich um eine Subapplikation ohne eigene Zustandslogik handelt, existieren keine internen Zustände. Die Funktionalität ist rein ereignisgesteuert: Jedes Auftreten von `CNF` führt zu einer einmaligen Ausgabeaktualisierung. Die internen `logiBUS_QXA`-Bausteine können jedoch eigene Zustände besitzen, die hier nicht näher betrachtet werden.

Anwendungsszenarien

- **Automatisierungstechnik:** Steuerung von acht digitalen Ausgängen (z. B. Ventile, Lampen, Schütze) über einen logiBUS-Feldbus, wobei die Kanäle je nach Maschinenkonfiguration frei gewählt werden können.
- **Testumgebungen:** In Prüfständen oder Simulationsumgebungen, in denen mehrere Ausgänge schnell und gemeinsam geschaltet werden müssen.
- **Modulare Steuerungen:** Als wiederverwendbarer Baustein in übergeordneten Steuerungslogiken, wenn viele ähnliche Ausgangsgruppen benötigt werden.

Vergleich mit ähnlichen Bausteinen

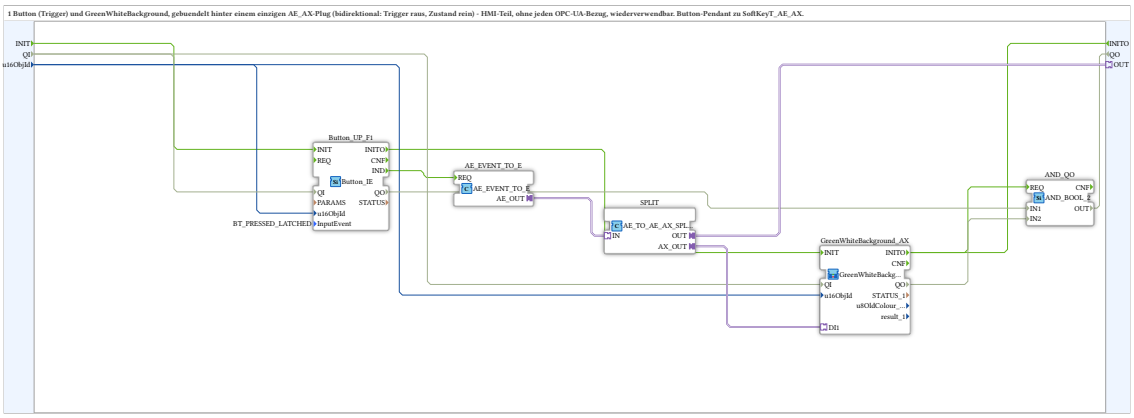
Der Schwester-Baustein `Bool8_TO_logiBUS_QX8` verwendet den Adaptertyp `logiBUS_QX` anstelle von `logiBUS_QXA`. Dies führt zu unterschiedlichen Adapterschnittstellen und kann bei Verwendung verschiedener logiBUS-Hardware oder Protokollvarianten relevant sein. `Bool8_TO_logiBUS_QXA8` ist somit für

Systeme ausgelegt, die den QXA-Adapter unterstützen, während der Schwester-Baustein für QX-kompatible Umgebungen gedacht ist. Beide Bausteine bieten identische Funktionalität hinsichtlich der Anzahl und Ansteuerung der Ausgänge.

Fazit

`Bool8_TO_logiBUS_QXA8` ist ein flexibel einsetzbarer Baustein zur Ansteuerung von acht digitalen Ausgängen über logiBUS. Durch die Kombination aus Bool-Eingängen, frei wählbaren Kanal-Identifikatoren und einem gemeinsamen Trigger bietet er eine einfache und effiziente Lösung für vielfältige Steuerungsaufgaben. Die adapterbasierte Architektur macht ihn kompatibel mit verschiedenen logiBUS-Implementierungen und ermöglicht eine klare Trennung zwischen logischer Steuerung und physischer Ausgabe.

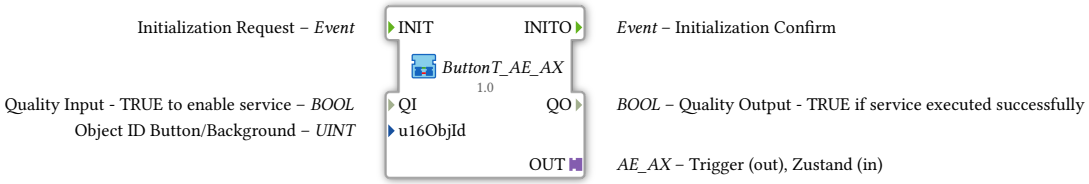
ButtonT_AE_AX



• – ○ • – ○ • – ○ •

Einleitung

`ButtonT_AE_AX` bündelt **einen Button (Trigger)** und eine `GreenWhiteBackground` hinter einem einzigen bidirektionalen `AE_AX`-Plug (Trigger raus, Zustand rein). Es ist der reine HMI-Teil ohne jeden OPC-UA-Bezug und wiederverwendbar. Der Baustein ist das Button-Pendant zu `SoftKeyT_AE_AX`: `Softkey_IE` wurde durch `Button_IE` ersetzt, `SK_RELEASED` durch `BT_PRESSED_LATCHED`.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; geht an `Button_UP_F1.INIT`

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierung abgeschlossen; kommt von `GreenWhiteBackground_AX.INITO`

Daten-Eingänge

- **QI** (`BOOL`): Quality Input - TRUE aktiviert den Dienst (Default `TRUE`); geht an `Button_UP_F1.QI` und `GreenWhiteBackground_AX.QI`
- **u16ObjId** (`UINT`): Object ID Button/Background (Default `ID_NULL`)

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde (AND aus `Button_UP_F1.QO` und `GreenWhiteBackground_AX.QO`)

Adapter

- **OUT** (Plug, `adapter::types::bidirectional::AE_AX`): Trigger (raus), Zustand (rein)

Verwendete Funktionsbausteine (FBs)

- **`Button_UP_F1`** (`isobus::UT::io::Button::Button_IE`): Parameter `InputEvent` = `BT_PRESSED_LATCHED` (`QI` kommt von außen)
- **`AE_EVENT_TO_E`** (`adapter::conversion::unidirectional::AE_EVENT_TO_E`): wandelt das Button-Ereignis in einen AE-Adapter
- **`SPLIT`** (`adapter::events::bidirectional::AE_TO_AE_AX_SPLIT`): trennt den bidirektionalen `AE_AX`-Plug in Trigger-Richtung und Zustands-Richtung
- **`GreenWhiteBackground_AX`** (SubApp `MyLib::sys::GreenWhiteBackground1_AX`): Hintergrundfarbe grün/weiß
- **`AND_QO`** (`iec61131::booleanOperators::AND_BOOL_2`): verknüpft die `QO` von `Button_UP_F1` und `GreenWhiteBackground_AX`

Funktionsweise

- `Button_UP_F1.IND` ⟶ `AE_EVENT_TO_E.REQ`
- `AE_EVENT_TO_E.AE_OUT` ⟶ `SPLIT.IN`
- `SPLIT.OUT` ⟶ `OUT` (Trigger nach außen, Zustand von außen)
- `SPLIT.AX_OUT` ⟶ `GreenWhiteBackground_AX.DI1` (Zustand färbt den Hintergrund)
- `u16ObjId` ⟶ `Button_UP_F1.u16ObjId` und `GreenWhiteBackground_AX.u16ObjId`
- Initialisierung: `INIT` ⟶ `Button_UP_F1.INIT`, `Button_UP_F1.INITO` ⟶ `GreenWhiteBackground_AX.INIT`, `GreenWhiteBackground_AX.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`
- Qualität: `QI` ⟶ `Button_UP_F1.QI` und `GreenWhiteBackground_AX.QI`; `Button_UP_F1.QO` ⟶ `AND_QO.IN1`, `GreenWhiteBackground_AX.QO` ⟶ `AND_QO.IN2`, `AND_QO.OUT` ⟶ `QO`

Technische Besonderheiten

- Dieselbe Objekt-ID dient dem Button und der Hintergrundfarbe
- `BT_PRESSED_LATCHED` statt `SK_RELEASED` des Softkey-Pendants
- Kein OPC-UA-Bezug, damit in beliebigen Szenarien wiederverwendbar

Anwendungsszenarien

- Button im VT als Trigger, dessen Zustand (z. B. von einem Toggle) als Hintergrundfarbe zurückgemeldet wird, über einen einzigen `AE_AX`-Anschluss

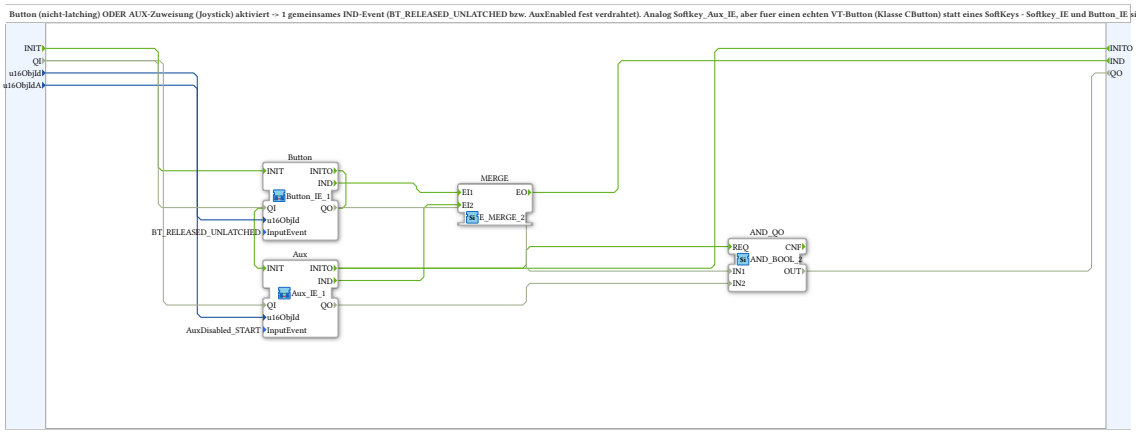
Zusammenfassung

`ButtonT_AE_AX` bündelt Button-Trigger und Zustandsfarbe hinter einem `AE_AX`-Plug.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Button_Aux_IE

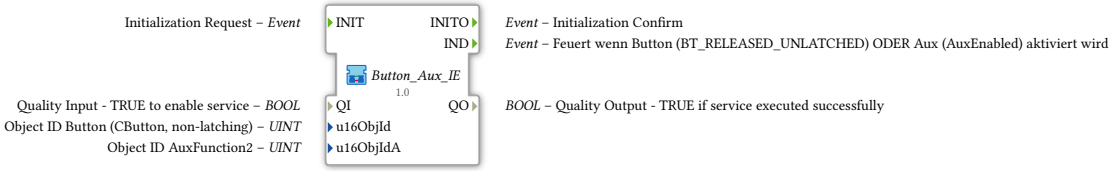


• – ○ • – ○ • – ○ •

Einleitung

`Button_Aux_IE` fasst einen nicht-latchenden VT-Button **oder** eine AUX-Zuweisung (Joystick) zu einem gemeinsamen `IND`-Ereignis zusammen. Die Aktivierungsart ist fest verdrahtet (`BT_RELEASED_UNLATCHED` bzw. `AuxEnabled`). Der Baustein entspricht `Softkey_Aux_IE`, jedoch für einen echten VT-Button (Klasse `CButton`) statt eines SoftKeys: `Softkey_IE` und `Button_IE` sind zwei getrennte FORTE-Service-Interface-Bausteine für unterschiedliche VT-Activation-Messages, ein Button liefert nie ein `SK_PRESSED`.

Der Button **muss** non-latching sein (`Latchable=0` im Pool). Ein latchender Button kann sinnvoll nur über `Button_IXA` (Dauerzustand/Pegel) gelesen werden, nicht über dieses Event-/`InputEvent`-Muster.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Button_Aux_IE

- **Typ**: SubAppType
- **Verwendete interne Bausteine**:
  - **Button**: SubApp `Button_IE_1` mit `InputEvent=BT_RELEASED_UNLATCHED`
  - **Aux**: SubApp `Aux_IE_1` mit `InputEvent=AuxDisabled_START`
  - **MERGE**: `iec61499::events::E_MERGE_2` – führt beide `IND`-Ereignisse zusammen
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_2` – verknüpft `Button.QO` und `Aux.QO` zum `QO`
- **Importe**: `BT_RELEASED_UNLATCHED` (`ButtonActivationCode`), `AuxDisabled_START` (`AuxiliaryState`)

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung
- **IND**: Feuert, wenn der Button (`BT_RELEASED_UNLATCHED`) oder die Aux-Zuweisung (`AuxEnabled`) aktiviert wird

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input – TRUE gibt den Dienst frei
- **u16ObjId** (`UINT`): Objekt-ID des Buttons (`CButton`, non-latching)
- **u16ObjIdA** (`UINT`): Objekt-ID der `AuxFunction2`

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output – UND aus `Button.QO` und `Aux.QO`

Programmablauf und Verbindungen

- `u16ObjId` ⟶ `Button.u16ObjId`
- `u16ObjIdA` ⟶ `Aux.u16ObjId`
- `Button.IND` ⟶ `MERGE.EI1`
- `Aux.IND` ⟶ `MERGE.EI2`
- `MERGE.EO` ⟶ `IND`
- `INIT` ⟶ `Button.INIT`; `Button.INITO` ⟶ `Aux.INIT`; `Aux.INITO` ⟶ `INITO` (und ⟶ `AND_QO.REQ`)
- `QI` ⟶ `Button.QI` und `QI` ⟶ `Aux.QI`
- `Button.QO` ⟶ `AND_QO.IN1`; `Aux.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`

Technische Besonderheiten

- Bugfix für `Automatik_Toggle_Button`: dort wurde zuvor `Softkey_Aux_IE` mit einer Button-Objekt-ID verwendet, `Softkey_IE` empfängt aber keine Button-Activation-Message (bestätigt vom Autor der FORTE-Implementierung)
- Der `E_MERGE_2` leitet das Ereignis weiter, welche der beiden Quellen auch feuert

- Version 1.0 (2026-09-20)

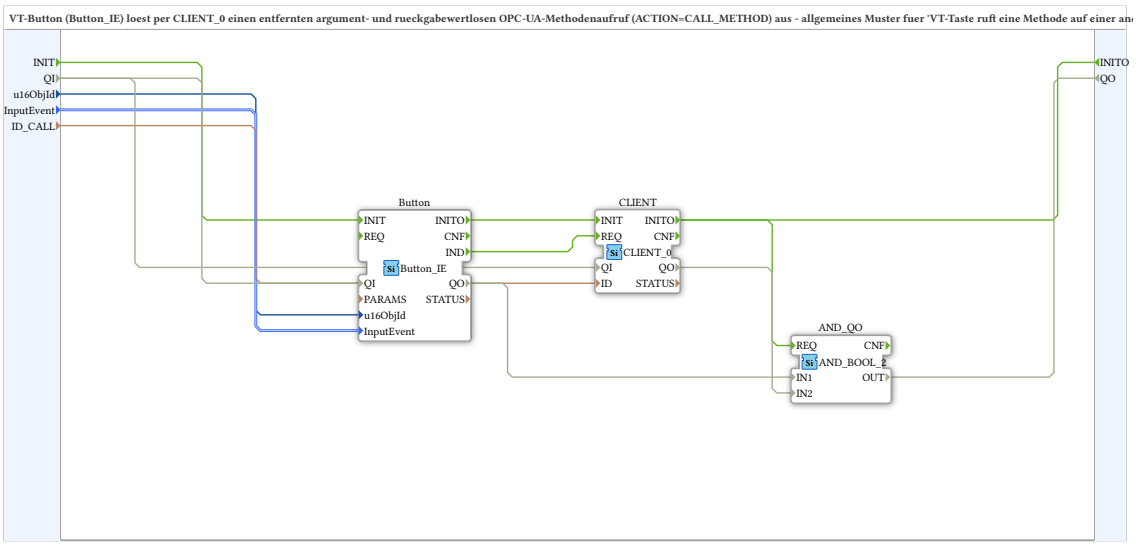
Anwendungsszenarien

- Ein Bedienelement, das wahlweise über einen VT-Button oder einen AUX-Joystick-Eingang ausgelöst werden soll
- Toggle-Logik (z. B. `Automatik_Toggle_Button`) mit zwei gleichwertigen Auslösern

Zusammenfassung

`Button_Aux_IE` kombiniert `Button_IE_1` und `Aux_IE_1` über einen `E_MERGE_2` zu einem einzigen `IND`-Ereignis für Button und AUX.

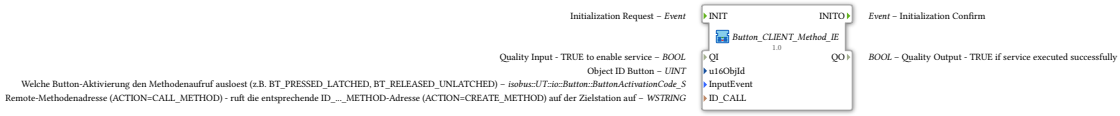
Button_CLIENT_Method_IE



• – ○ • – ○ • – ○ •

Einleitung

`Button_CLIENT_Method_IE` löst per VT-Button (`Button_IE`) über `CLIENT_0` einen entfernten, argument- und rückgabewertlosen **OPC-UA-Methodenaufruf** (`ACTION=CALL_METHOD`) aus. Es ist das allgemeine Muster „VT-Taste ruft eine Methode auf einer anderen Station auf“ und wurde aus `logiBUS_AI_Calibrated_Subscribe_VT_OPC` herausgelöst, wo es zweimal identisch vorkam. Im Gegensatz zu `Button_Method_IE` (Button ODER lokale Methode ⟶ gemeinsames `IND`) ist dies der umgekehrte „Caller“-Fall: Button ⟶ entfernte Methode.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialization Request

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialization Confirm

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input, TRUE aktiviert den Dienst
- **u16ObjId** (`UINT`): Object-ID des Buttons
- **InputEvent** (`isobus::UT::io::Button::ButtonActivationCode_S`): Welche Button-Aktivierung den Methodenaufruf auslöst (z. B. `BT_PRESSED_LATCHED`, `BT_RELEASED_UNLATCHED`)

- **ID_CALL** (`WSTRING`): Remote-Methodenadresse (`ACTION=CALL_METHOD`); ruft die entsprechende `ID_..._METHOD`-Adresse (`ACTION=CREATE_METHOD`) auf der Zielstation auf

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output, UND-Verknüpfung (`AND_QO`) der `QO`-Ausgänge von `Button` und `CLIENT`

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Verwendete Funktionsbausteine (FBs)

- **Button**: `isobus::UT::io::Button::Button_IE` (`QI` vom Wrapper-Eingang)
- **CLIENT**: `iec61499::net::CLIENT_0` (`QI` vom Wrapper-Eingang)
- **AND_QO**: `iec61131::booleanOperators::AND_BOOL_2` – verknüpft `Button.QO` und `CLIENT.QO` zu `QO`

Programmablauf und Verbindungen

- `u16ObjId` ⟶ `Button.u16ObjId`; `InputEvent` ⟶ `Button.InputEvent`
- `Button.IND` ⟶ `CLIENT.REQ`
- `ID_CALL` ⟶ `CLIENT.ID`
- Initialisierung: `INIT` ⟶ `Button.INIT`; `Button.INITO` ⟶ `CLIENT.INIT`; `CLIENT.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`
- Qualität: `QI` ⟶ `Button.QI` und `CLIENT.QI`; `Button.QO` ⟶ `AND_QO.IN1`; `CLIENT.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`

Bei jeder Button-Aktivierung gemäß `InputEvent` wird somit der Methodenaufruf `CLIENT_0` angestoßen.

Technische Besonderheiten

- `InputEvent` ist ein Parameter und nicht auf `BT_PRESSED_LATCHED` festgelegt, damit auch andere Aktivierungsarten (z. B. Loslassen) wiederverwendbar sind.
- Die Rückmeldung (`CNF`) von `CLIENT_0` wird nicht nach außen geführt.

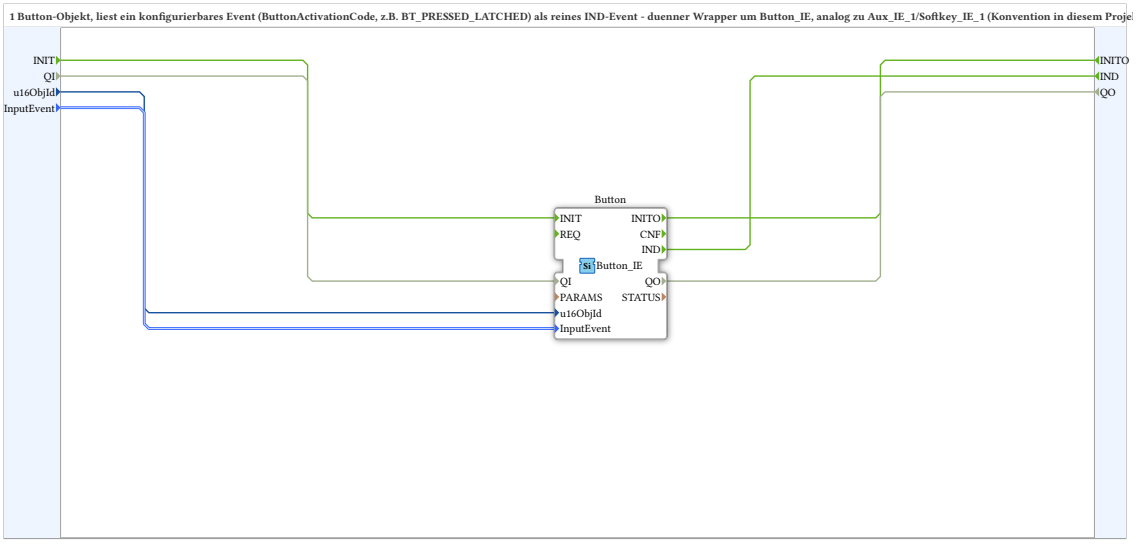
Anwendungsszenarien

- VT-Taste, die eine Methode auf einer anderen Station aufruft (z. B. Kalibrierung per OPC-UA)

Fazit

`Button_CLIENT_Method_IE` ist ein kleines, generisches Muster für den Aufruf einer entfernten OPC-UA-Methode per VT-Taste.

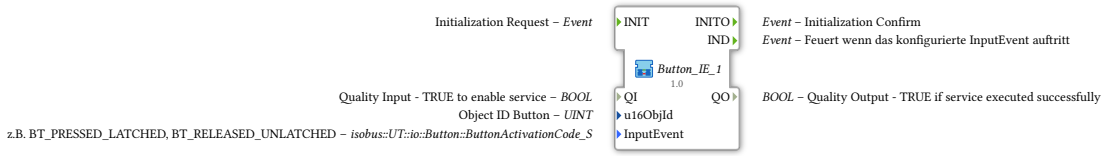
Button_IE_1



• – ◦ • – ◦ • – ◦ •

Einleitung

`Button_IE_1` liest ein Button-Objekt und gibt ein konfigurierbares Ereignis (`ButtonActivationCode`, z. B. `BT_PRESSED_LATCHED`) als reines `IND`-Ereignis aus. Der Baustein ist ein dünner Wrapper um `isobus::UT::io::Button::Button_IE`, analog zu `Aux_IE_1` und `Softkey_IE_1`. Konvention in diesem Projekt: Die isobus-IE-Bausteine werden nie roh inline verwendet, sondern immer gewrappt.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialization Request

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialization Confirm
- **IND**: feuert, wenn das konfigurierte `InputEvent` auftritt

Daten-Eingänge

- **QI** (`BOOL`, Default `TRUE`): Quality Input - TRUE, um den Dienst freizugeben
- **u16ObjId** (UINT): Object ID des Buttons (Default `ID_NULL`)
- **InputEvent** (`isobus::UT::io::Button::ButtonActivationCode_S`): z. B. `BT_PRESSED_LATCHED`, `BT_RELEASED_UNLATCHED` (Default `Invalid`)

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde (durchgereicht von `Button.QO`)

Adapter

Es werden keine Adapter verwendet.

Funktionsweise

Das Netzwerk enthält genau eine Instanz `Button` des Typs `isobus::UT::io::Button::Button_IE`. `u16ObjId`, `InputEvent` und `QI` werden unverändert auf die gleichnamigen Eingänge der Instanz geführt, `INIT` → `Button.INIT`; deren Ereignisse `IND` und `INITO` sowie der Ausgang `QO` werden direkt als `IND`, `INITO` und `QO` der SubApp ausgegeben.

Technische Besonderheiten

- Reine Durchreichung ohne eigene Logik oder Zustand
- `QI` wird von außen an den inneren Baustein durchgereicht (Vorgabe `TRUE`)
- Weil `InputEvent` ein Parameter ist, kann dieselbe SubApp für unterschiedliche Ereignisarten (z. B. Drücken oder Loslassen) verwendet werden

Anwendungsszenarien

- Einlesen einer einzelnen VT-Bedienung als Ereignis, ohne den rohen IE-Baustein direkt im Netzwerk zu verwenden

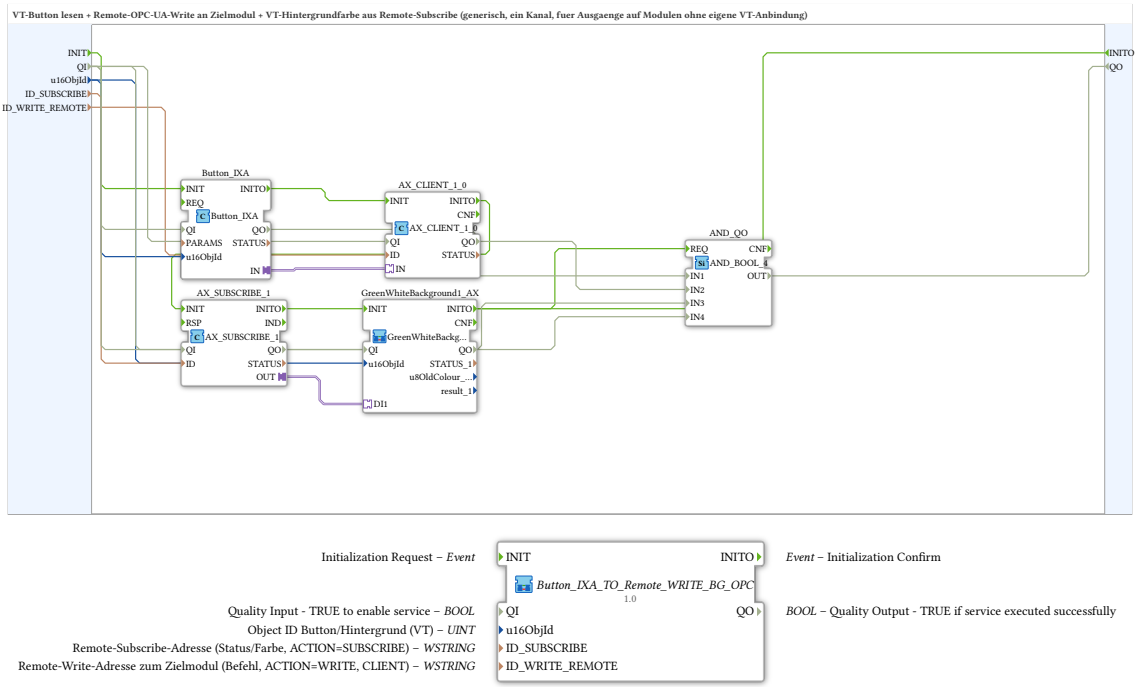
⚖️ Vergleich mit ähnlichen Bausteinen

Gleichartige Wrapper sind `Aux_IE_1` und `Softkey_IE_1` sie unterscheiden sich nur im VT-Objekttyp und im Aufzählungstyp des `InputEvent`.

Fazit

`Button_IE_1` ist ein minimaler Wrapper, der ein konfigurierbares Button-Ereignis als `IND` bereitstellt.

Button_IXA_TO_Remote_WRITE_BG_OPC



• – ○ • – ○ • – ○ •

Einleitung

Die Subapp `Button_IXA_TO_Remote_WRITE_BG_OPC` dient zum Lesen eines VT-Buttons und zum Senden eines Remote-OPC-UA-Write-Befehls an ein Zielmodul. Zusätzlich wird die VT-Hintergrundfarbe des Buttons über einen Remote-Subscribe (OPC-UA) aktualisiert. Die Subapp ist generisch ausgelegt und unterstützt einen Kanal; sie eignet sich für Ausgänge auf Modulen, die keine eigene VT-Anbindung besitzen.

Schnittstellenstruktur

Die Subapp besitzt nach außen den Ereigniseingang `INIT`, den Ereignisausgang `INITO`, die Dateneingänge `QI`, `u16ObjId`, `ID_SUBSCRIBE` und `ID_WRITE_REMOTE` sowie den Datenausgang `QO`. Es sind keine Adapter nach außen geführt.

Ereignis-Eingänge

- `INIT` (EInit): Initialisierungsanforderung; startet die Kette `Button_IXA` ⟶ `AX_CLIENT_1_0` ⟶ `AX_SUBSCRIBE_1` ⟶ `GreenWhiteBackground1_AX`

Ereignis-Ausgänge

- `INITO` (EInit): Initialisierungsbestätigung; kommt von `GreenWhiteBackground1_AX.INITO`

Daten-Eingänge

- `QI` (BOOL): Quality Input, Freigabe des Dienstes. Initialwert: `TRUE`
- `u16ObjId` (UINT): Objekt-ID des Buttons/Hintergrunds (VT). Initialwert: `ID_NULL`
- `ID_SUBSCRIBE` (WSTRING): Remote-Subscribe-Adresse für Status/Farbe (ACTION=SUBSCRIBE)
- `ID_WRITE_REMOTE` (WSTRING): Remote-Write-Adresse zum Zielmodul für den Befehl (ACTION=WRITE, CLIENT)

Daten-Ausgänge

- `QO` (BOOL): Quality Output, `AND` der `QO`-Ausgänge von `Button_IXA`, `AX_CLIENT_1_0`, `AX_SUBSCRIBE_1` und `GreenWhiteBackground1_AX` (über `AND_QO`)

Adapter

Nach außen werden keine Adapter bereitgestellt. Intern werden die Adapter `AX_CLIENT_1_0` und `AX_SUBSCRIBE_1` verwendet.

Funktionsweise

Der Baustein stellt eine Verbindung zwischen einem VT-Bedienelement und einem entfernten Modul über OPC-UA her. Der `Button_IXA`-FB liest den Zustand des über `u16ObjId` referenzierten Buttons. Wird der Button betätigt, sendet der `AX_CLIENT_1_0`-Adapter den Schaltbefehl über die in `ID_WRITE_REMOTE` konfigurierte Remote-Write-Adresse an das Zielmodul.

Parallel empfängt der `AX_SUBSCRIBE_1`-Adapter über die in `ID_SUBSCRIBE` angegebene Adresse den aktuellen Status bzw. die Farbe des Zielmoduls. Diese Information wird über die Adapterverbindung an die Subapp `GreenWhiteBackground1_AX` übergeben, die die Hintergrundfarbe des VT-Buttons entsprechend aktualisiert. So wird eine Rückmeldung des Schaltzustands visualisiert.

Die `u16ObjId` wird sowohl an den `Button_IXA` als auch an die Subapp `GreenWhiteBackground1_AX` geleitet, um sicherzustellen, dass sich beide auf dasselbe Objekt beziehen.

Initialisierung und Qualität: `INIT` → `Button_IXA.INIT`, `Button_IXA.INITO` → `AX_CLIENT_1_0.INIT`, `AX_CLIENT_1_0.INITO` → `AX_SUBSCRIBE_1.INIT`, `AX_SUBSCRIBE_1.INITO` → `GreenWhiteBackground1_AX.INIT`, `GreenWhiteBackground1_AX.INITO` → `INITO` und → `AND_QO.REQ`. `QI` geht an `Button_IXA.QI`, `AX_CLIENT_1_0.QI`, `AX_SUBSCRIBE_1.QI` und `GreenWhiteBackground1_AX.QI`; deren vier `QO`-Ausgänge laufen auf `AND_QO.IN1` bis `IN4` (`iec61131::booleanOperators::AND_BOOL_4`), `AND_QO.OUT` → `QO`.

Technische Besonderheiten

- Verwendung von OPC-UA-Adaptern (`adapter::net::AX_CLIENT_1_0` und `adapter::net::AX_SUBSCRIBE_1`)
- Kombination von Remote-Write (Befehl) und Remote-Subscribe (Status/Farbe)
- Generische Konfiguration über Objekt-ID und Adressen als Eingänge
- Nach außen nur `INIT`/`INITO` als Ereignisse; im Betrieb arbeitet die Subapp rein daten- und adaptergesteuert
- `QI` der internen FBs ist nicht mehr fest auf `TRUE` gesetzt, sondern wird vom SubApp-Eingang `QI` (Initialwert `TRUE`) durchgereicht
- `QO` ist das `AND` der `QO`-Ausgänge aller vier internen Bausteine

Zustandsübersicht

Die Subapp besitzt keine eigene Zustandsmaschine. Das Verhalten ergibt sich aus den internen Funktionsbausteinen. Der FB `Button_IXA` kann je nach Implementierung interne Zustände (z.B. gedrückt/nicht gedrückt) besitzen, die jedoch nicht nach außen sichtbar sind.

Anwendungsszenarien

- Fernbedienung von Ausgängen an dezentralen Modulen über OPC-UA
- Visualisierung von Schaltzuständen in einer VT durch Hintergrundfarben (grün/weiß)
- Anbindung von Bedienelementen an Steuerungen, die keine direkte VT-Anbindung bieten
- Erweiterung bestehender HMI-Projekte um Remote-I/O-Funktionalität

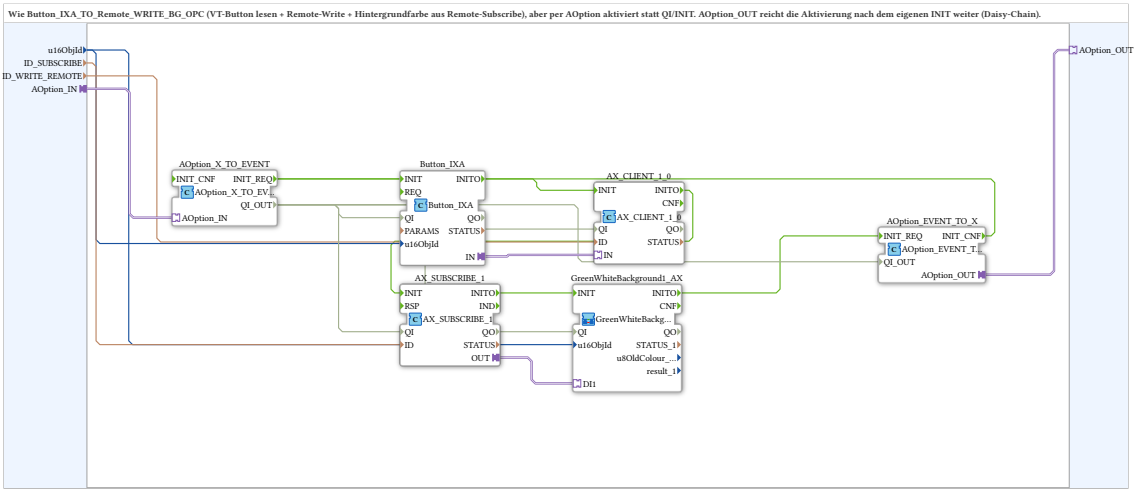
Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu direkt gekoppelten VT-Buttons, die lokal über den Bus angebunden sind, realisiert dieser Baustein die Kommunikation vollständig über OPC-UA. Dadurch ist er flexibler einsetzbar, benötigt jedoch eine konfigurierte OPC-UA-Verbindung. Gegenüber einem generischen OPC-UA-Client bietet die Subapp eine höhere Abstraktion, da sie Button-Logik und Hintergrundfarbsteuerung integriert.

Fazit

Die Subapp `Button_IXA_TO_Remote_WRITE_BG_OPC` ist eine nützliche Komponente für verteilte Automatisierungssysteme, bei denen Bedienelemente und Ausgänge über OPC-UA gekoppelt werden. Sie vereinfacht die Projektierung durch ihre generische Konfiguration und die integrierte Rückmelde-Visualisierung.

Button_IXA_TO_Remote_WRITE_BG_Option_OPC



• – ○ • – ○ • – ○ •

Einleitung

`Button_IXA_TO_Remote_WRITE_BG_Option_OPC` liest einen VT-Button, schreibt per Remote-OPC-UA-Write an ein Zielmodul und setzt die Hintergrundfarbe aus einem Remote-Subscribe – wie `Button_IXA_TO_Remote_WRITE_BG_OPC`, aber per `AOption` aktiviert statt über `QI`/`INIT`. `AOption_OUT` reicht die Aktivierung nach dem eigenen `INIT` weiter (Daisy-Chain wie `OptionalGroup_001a`). Der Baustein ist eine Kopie von `Button_IXA_TO_Remote_WRITE_BG_OPC`, bei der `INIT`/`INITO`/`QI`/`QO` durch `AOption_IN` und `AOption_OUT` ersetzt wurden.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Variable	Typ	Vorgabe	Kommentar
`u16ObjId`	UINT	`ID_NULL`	Object ID Button/Hintergrund (VT).
`ID_SUBSCRIBE`	WSTRING		Remote-Subscribe-Adresse (Status/Farbe, `ACTION=SUBSCRIBE`).
`ID_WRITE_REMOTE`	WSTRING		Remote-Write-Adresse zum Zielmodul (Befehl, `ACTION=WRITE`, CLIENT).

Daten-Ausgänge

Keine vorhanden.

Adapter

Adapter	Typ	Richtung	Kommentar

AOption_IN	Option::AOption	Socket	Treibt INIT/QI; anschließen an AOption_AUI_DEMUX_N.OUTx oder AOption_OUT einer vorgeschalteten Instanz.
AOption_OUT	Option::AOption	Plug	Reicht die Aktivierung nach dem eigenen INIT an das nächste Glied weiter (Daisy-Chain); muss verbunden sein, Kettenende z. B. DummyGroup.

Verwendete Funktionsbausteine (FBs)

- **AOption_X_TO_EVENT** (Option::conversion::AOption_X_TO_EVENT): wandelt die Option in INIT_REQ/QI_OUT und nimmt INIT_CNF entgegen
- **Button_IXA** (isobus::UT::io::Button::Button_IXA): liest den VT-Button
- **AX_CLIENT_1_0** (adapter::net::AX_CLIENT_1_0): Remote-Write zum Zielmodul
- **AX_SUBSCRIBE_1** (adapter::net::AX_SUBSCRIBE_1): Remote-Subscribe von Status/Farbe
- **GreenWhiteBackground1_AX** (SubApp MyLib::sys::GreenWhiteBackground1_AX): Hintergrundfarbe; siehe GreenWhiteBackground1_AX
- **AOption_EVENT_TO_X** (Option::conversion::AOption_EVENT_TO_X): gibt die Aktivierung an AOption_OUT weiter

Programmablauf und Verbindungen

1. Initialisierungskette: AOption_X_TO_EVENT.INIT_REQ -> Button_IXA.INIT; Button_IXA.INITO -> AX_CLIENT_1_0.INIT; AX_CLIENT_1_0.INITO -> AX_SUBSCRIBE_1.INIT; AX_SUBSCRIBE_1.INITO -> GreenWhiteBackground1_AX.INIT; GreenWhiteBackground1_AX.INITO -> AOption_EVENT_TO_X.INIT_REQ; AOption_EVENT_TO_X.INIT_CNF -> AOption_X_TO_EVENT.INIT_CNF.
2. Qualifier: AOption_X_TO_EVENT.QI_OUT -> Button_IXA.QI, AX_CLIENT_1_0.QI, AX_SUBSCRIBE_1.QI, GreenWhiteBackground1_AX.QI und AOption_EVENT_TO_X.QI_OUT.
3. Option: AOption_IN -> AOption_X_TO_EVENT.AOption_IN; AOption_EVENT_TO_X.AOption_OUT -> AOption_OUT.
4. Befehl: Button_IXA.IN -> AX_CLIENT_1_0.IN.
5. Status: AX_SUBSCRIBE_1.OUT -> GreenWhiteBackground1_AX.DI1.
6. Daten: u16ObjId -> Button_IXA.u16ObjId und GreenWhiteBackground1_AX.u16ObjId; ID_WRITE_REMOTE -> AX_CLIENT_1_0.ID; ID_SUBSCRIBE -> AX_SUBSCRIBE_1.ID.

Technische Besonderheiten

- Der Baustein hat keine eigenen Ereignis- und QI/QO-Anschlüsse; Aktivierung und Bestätigung laufen ausschließlich über AOption_IN/AOption_OUT.
- AOption_OUT muss weiterverbunden sein (z. B. DummyGroup als Kettenende), sonst kommt kein INIT_CNF zurück.

Anwendungsszenarien

- Optionale Bedienstelle mit VT-Button für einen entfernten Aktor samt farbiger Statusanzeige, aktiv nur bei ausgewählter Option.
- Mehrere Instanzen, die dieselbe Option teilen und per AOption_OUT verkettet werden.

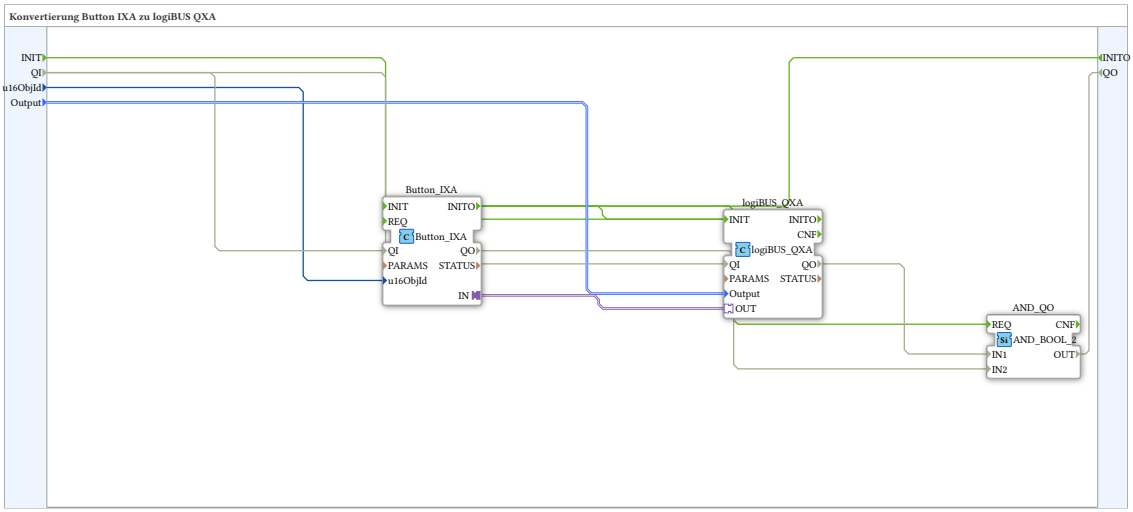
Zusammenfassung

Button_IXA_TO_Remote_WRITE_BG_Option_OPC bündelt VT-Button, Remote-Write und Hintergrundfarbe aus Remote-Subscribe hinter einem AOption-Socket und reicht die Aktivierung per AOption_OUT weiter.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

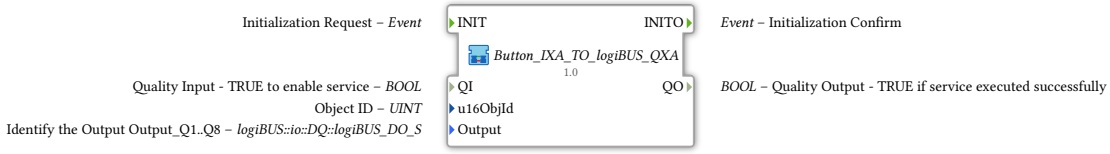
Button_IXA_TO_logiBUS_QXA



• – ○ • – ○ • – ○ •

Einleitung

`Button_IXA_TO_logiBUS_QXA` verbindet einen VT-Taster (`Button_IXA`) direkt mit einem physischen digitalen Ausgang (`logiBUS_QXA`) — die einfachste Form eines VT-schaltbaren Ausgangs, ohne Statusanzeige und ohne OPC-UA. Für die Variante mit VT-Statusfarbe siehe `Button_IXA_TO_logiBUS_QXA_BG`.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Button_IXA_TO_logiBUS_QXA

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **Button_IXA**: `isobus::UT::io::Button::Button_IXA` — VT-Taster-Adapter, `QI` kommt vom Eingang `QI`, `u16ObjId` identifiziert die VT-Taste.
  - **logiBUS_QXA**: `logiBUS::io::DQ::logiBUS_QXA` — physischer digitaler Ausgang, `QI` kommt vom Eingang `QI`.
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_2` — verknüpft `logiBUS_QXA.QO` und `Button_IXA.QO` zum Ausgang `QO`.
- **Funktionsweise**: Der Adapter-Ausgang des Tasters wird direkt auf den Adapter-Eingang des physischen Ausgangs verdrahtet — keine Zwischenlogik. Zusätzlich hat die SubApp die Schnittstelle `INIT`/`INITO` (Ereignisse) und `QI`/`QO` (BOOL): `INIT` startet die Init-Kette, `QI` (Initialwert `TRUE`) wird an beide FBs durchgereicht, `QO` ist die UND-Verknüpfung ihrer `QO`-Ausgänge.

Programmablauf und Verbindungen

1. `u16ObjId` ⟶ `Button_IXA.u16ObjId`; `Output` ⟶ `logiBUS_QXA.Output`.
2. `Button_IXA.IN` (Adapter) ⟶ `logiBUS_QXA.OUT` (Adapter) — direkte Durchschaltung.
3. `INIT` ⟶ `logiBUS_QXA.INIT`; `logiBUS_QXA.INITO` ⟶ `Button_IXA.INIT`; `Button_IXA.INITO` ⟶ `INITO` und `AND_QO.REQ`.
4. `QI` ⟶ `logiBUS_QXA.QI` und `Button_IXA.QI`.
5. `logiBUS_QXA.QO` ⟶ `AND_QO.IN1`; `Button_IXA.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`.

Anwendungsszenarien

- Minimaler VT-Taster-zu-Ausgang-Baustein für Übungen, die noch keine Statusanzeige oder Fernbedienung benötigen.

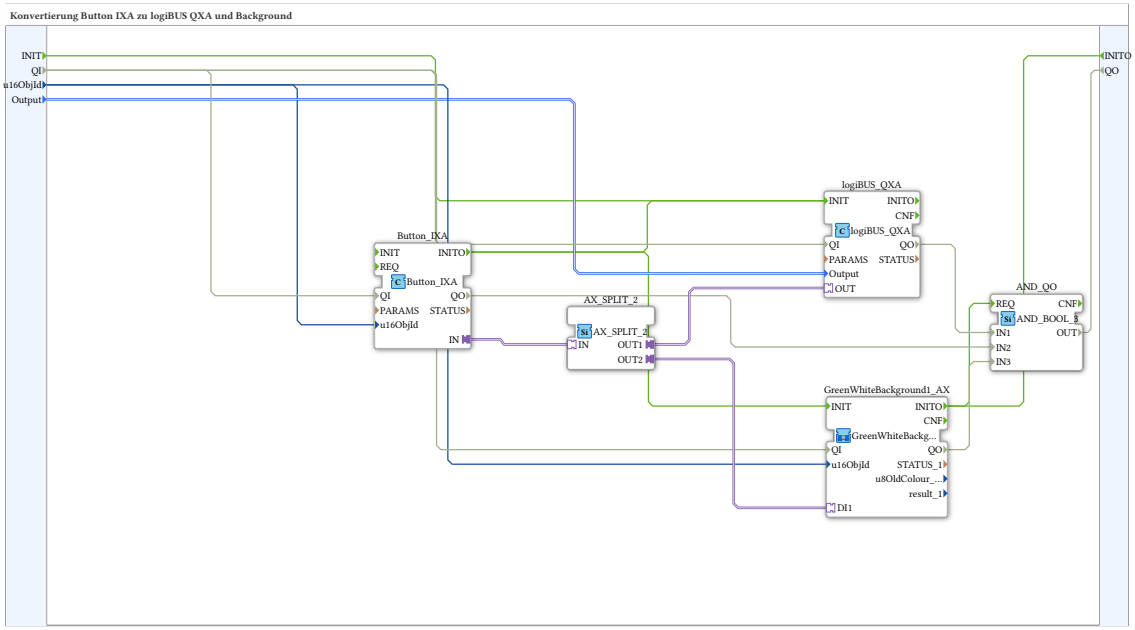
Zusammenfassung

Die einfachste Bausteinvariante dieser Familie: ein VT-Taster direkt auf einen physischen Ausgang, ohne Zusatzfunktion.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

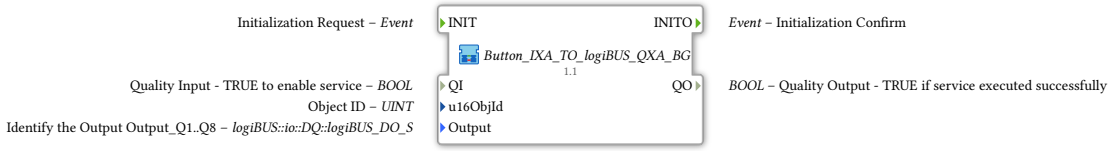
Button_IXA_TO_logiBUS_QXA_BG



• – ○ • – ○ • – ○ •

Einleitung

`Button_IXA_TO_logiBUS_QXA_BG` erweitert `Button_IXA_TO_logiBUS_QXA` um eine VT-Statusfarbe: Der Taster-Zustand schaltet nicht nur den physischen Ausgang, sondern zusätzlich die Hintergrundfarbe des Tasters selbst (Grün/Weiß). Die OPC-UA-fähige Weiterentwicklung ist `Button_IXA_TO_logiBUS_QXA_BG_OPC`.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Button_IXA_TO_logiBUS_QXA_BG

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **Button_IXA**: `isobus::UT::io::Button::Button_IXA` — VT-Taster-Adapter.
  - **logiBUS_QXA**: `logiBUS::io::DQ::logiBUS_QXA` — physischer digitaler Ausgang.
  - **AX_SPLIT_2**: `adapter::events::unidirectional::AX_SPLIT_2` — verzweigt das Taster-Signal.
  - **GreenWhiteBackground1_AX** (SubApp, `MyLib::sys`): setzt die VT-Hintergrundfarbe passend zum Zustand (siehe Background-Farbbausteine).
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_3` — verknüpft die `QO` von `logiBUS_QXA`, `Button_IXA` und `GreenWhiteBackground1_AX`.
- **Funktionsweise**: `Button_IXA.IN` wird über `AX_SPLIT_2` sowohl an `logiBUS_QXA.OUT` (physischer Ausgang) als auch an `GreenWhiteBackground1_AX.DI1` (Statusfarbe) verteilt.

Programmablauf und Verbindungen

1. `u16ObjId` ⟶ `Button_IXA.u16ObjId` und `GreenWhiteBackground1_AX.u16ObjId`; `Output` ⟶ `logiBUS_QXA.Output`.
2. `Button_IXA.IN` ⟶ `AX_SPLIT_2.IN` ⟶ `AX_SPLIT_2.OUT1` ⟶ `logiBUS_QXA.OUT`, `AX_SPLIT_2.OUT2` ⟶ `GreenWhiteBackground1_AX.DI1`.
3. Initialisierung: Ereigniseingang `INIT` ⟶ `logiBUS_QXA.INIT`, `logiBUS_QXA.INITO` ⟶ `Button_IXA.INIT`, `Button_IXA.INITO` ⟶ `GreenWhiteBackground1_AX.INIT`, `GreenWhiteBackground1_AX.INITO` ⟶ Ereignisausgang `INITO` und ⟶ `AND_QO.REQ`.
4. Qualität: Dateneingang `QI` (`BOOL`, Default `TRUE`) ⟶ `logiBUS_QXA.QI`, `Button_IXA.QI` und `GreenWhiteBackground1_AX.QI`; `logiBUS_QXA.QO` ⟶ `AND_QO.IN1`, `Button_IXA.QO` ⟶ `AND_QO.IN2`, `GreenWhiteBackground1_AX.QO` ⟶ `AND_QO.IN3`, `AND_QO.OUT` ⟶ Datenausgang `QO`.

Anwendungsszenarien

- VT-Taster mit direkter optischer Rückmeldung (Statusfarbe), aber ohne Fernbedienung über OPC-UA.

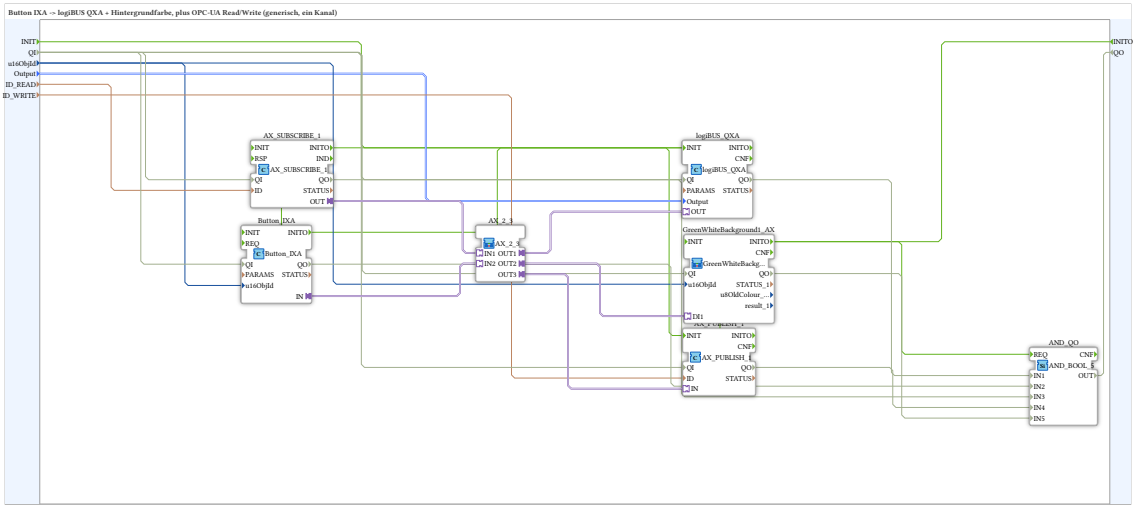
Zusammenfassung

Fügt der einfachen Taster-zu-Ausgang-Verdrahtung eine VT-Statusfarbe hinzu — die Vorstufe zur OPC-UA-fähigen Variante.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

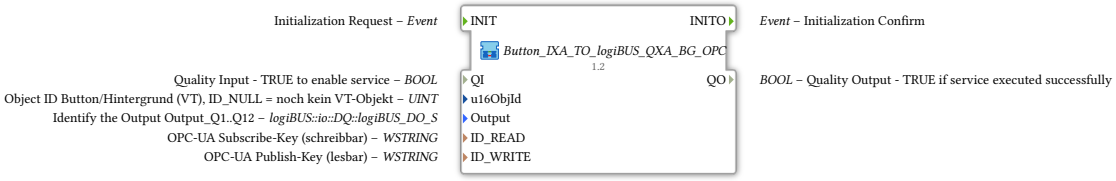
Button_IXA_TO_logiBUS_QXA_BG_OPC



• – ○ • – ○ • – ○ •

Einleitung

`Button_IXA_TO_logiBUS_QXA_BG_OPC` ist der wiederverwendbare Baustein für **einen einzelnen digitalen Ausgang**, der sowohl über einen VT-Taster als auch über OPC-UA (Web-Client) geschaltet werden kann — inklusive VT-Statusanzeige. Er wird 12-fach parametrisiert in `InputOutputTesterButton_DIDO_OPC_UA` instanziiert.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Button_IXA_TO_logiBUS_QXA_BG_OPC

- **Typ**: SubAppType
- **Schnittstelle (zusätzlich)**: Ereignis-Eingang `INIT`, Ereignis-Ausgang `INITO` (je `EInit`), Daten-Eingang `QI` (`BOOL`, Vorgabe `TRUE`), Daten-Ausgang `QO` (`BOOL`).
- **Verwendete interne FBs**:
  - **Button_IXA**: `isobus::UT::io::Button::Button_IXA` — VT-Taster, `u16ObjId` identifiziert die VT-Taste. Seit Version 1.2 vorübergehend **non-latching** (tastender Pegel, kein Klick-Toggle mehr).
  - **AX_2_3**: SubApp `AX_2_3` — verknüpft `IN1` (OPC-UA-Subscribe) und `IN2` (VT-Taster) per `AX_OR_2` zu einem gemeinsamen Pegel und verteilt ihn per `AX_SPLIT_3` auf drei Ziele (`OUT1`..`OUT3`).
  - **logiBUS_QXA**: `logiBUS::io::DQ::logiBUS_QXA` — physischer digitaler Ausgang.
  - **GreenWhiteBackground1_AX** (SubApp, `MyLib::sys`): VT-Hintergrundfarbe passend zum Ausgangszustand.
  - **AX_SUBSCRIBE_1** / **AX_PUBLISH_1**: `adapter::net::AX_SUBSCRIBE_1`/`AX_PUBLISH_1` — OPC-UA-Lese-/Schreibzugriff.
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_5` — verknüpft die `QO`-Ausgänge von `logiBUS_QXA`, `Button_IXA`, `AX_SUBSCRIBE_1`, `AX_PUBLISH_1` und `GreenWhiteBackground1_AX` zum `QO` der SubApp.
- **Funktionsweise**: Zwei unabhängige Schaltquellen (VT-Taster, OPC-UA-Schreibzugriff) werden in `AX_2_3` ODER-verknüpft; der resultierende Pegel speist gleichzeitig den physischen Ausgang, die VT-Statusfarbe und das OPC-UA-Echo.

Programmablauf und Verbindungen

1. **VT-Taster-Pfad**: `Button_IXA.IN` (Adapter) ⟶ `AX_2_3.IN2`.
2. **OPC-UA-Pfad**: `AX_SUBSCRIBE_1.OUT` (Adapter, Web-Schreibzugriff) ⟶ `AX_2_3.IN1`.
3. **Gemeinsamer Zustand**: `AX_2_3.OUT1` ⟶ `logiBUS_QXA.OUT` (physischer Ausgang), `AX_2_3.OUT2` ⟶ `GreenWhiteBackground1_AX.DI1` (VT-Statusfarbe), `AX_2_3.OUT3` ⟶ `AX_PUBLISH_1.IN` (OPC-UA-Echo an den Web-Client).
4. **Initialisierungskette**: `INIT` ⟶ `logiBUS_QXA.INIT` ⟶ `Button_IXA.INIT` ⟶ `AX_SUBSCRIBE_1.INIT`; `AX_SUBSCRIBE_1.INITO` ⟶ `AX_PUBLISH_1.INIT` (ausgeblendete Verbindung) — stellt sicher, dass der Publish-Adapter erst initialisiert wird, nachdem der Subscribe-Adapter bereit ist; `AX_PUBLISH_1.INITO` ⟶ `GreenWhiteBackground1_AX.INIT`; `GreenWhiteBackground1_AX.INITO` ⟶ `INITO` (und ⟶ `AND_QO.REQ`).
5. **Parameter**: `u16ObjId` ⟶ `Button_IXA.u16ObjId` und `GreenWhiteBackground1_AX.u16ObjId`; `Output` ⟶ `logiBUS_QXA.Output`; `ID_READ` ⟶ `AX_SUBSCRIBE_1.ID`; `ID_WRITE` ⟶ `AX_PUBLISH_1.ID`.
6. **Qualität**: `QI` ⟶ `.QI` von `logiBUS_QXA`, `Button_IXA`, `AX_SUBSCRIBE_1`, `AX_PUBLISH_1` und `GreenWhiteBackground1_AX`; deren `QO`-Ausgänge ⟶ `AND_QO.IN1`..`IN5`; `AND_QO.OUT` ⟶ `QO`. Die Instanzen sind nicht mehr fest auf `QI=TRUE` parametriert, sondern werden über `QI` der SubApp freigegeben.

Technische Besonderheiten

- **Tastender Pegel statt Klick-Toggle**: Seit Version 1.2 ersetzt ein einfaches `AX_OR_2` (in `AX_2_3`) den früheren SR-Latch (`AX_ASR_RF_TRIG` × 2 + `ASR_MERGE_2` + `ASR_AX_SR`). VT-Taster **und** Remote-Kommando sind dadurch tastend (Pegel); der Ausgang folgt der ODER-Verknüpfung beider Quellen. Die Variante mit Flankenerkennung und Latch ist `Button_IXA_TO_logiBUS_QXA_BG_OPC_LATCHING`.
- **Getrennte OPC-UA-Schlüssel**: `ID_READ` (Subscribe-Key, schreibbar) und `ID_WRITE` (Publish-Key, lesbar) sind getrennte Parameter; der Echo-Pfad (`AX_PUBLISH_1`) schreibt den ODER-Pegel, `AX_SUBSCRIBE_1` liest nur den Schreibzugriff des Web-Clients.
- **`RSP`-Event von `AX_SUBSCRIBE_1` ist wirkungslos**: Quellcode-geprüft in FORTE (`CCommFB::sendData()`, `commfb.cpp`) — für den Service-Typ *Subscriber* ist der `RSP`-Zweig ein kompletter No-Op, das Event-Paar `INIT`/`RSP` existiert nur, weil `CCommFB` die gemeinsame Basisklasse aller „Responder"-artigen Dienst-FBs ist (z. B. `SERVER`, wo `RSP` real etwas auslöst).
- **Boot-Reihenfolge**: `AX_SUBSCRIBE_1.INITO` löst `AX_PUBLISH_1.INIT` aus, um Race Conditions beim Deployment zu vermeiden (vgl. dasselbe Muster im PWM-Beispiel `RampLimitFS_TO_logiBUS_QDA_PWM_OPC`).

Anwendungsszenarien

- Jeder digitale Ausgang, der sowohl vor Ort (VT-Taster) als auch fernbedient (Web/OPC-UA) im Tastbetrieb schaltbar sein soll, mit VT-Statusfarbe und OPC-UA-Echo.

Vergleich mit ähnlichen Bausteinen

Gegenüber dem Eingangs-Pendant `logiBUS_IXA_BG_OPC` kommt hier die zweite Schreibquelle (OPC-UA-Subscribe) hinzu, die per ODER (`AX_2_3`) mit dem VT-Taster zusammengeführt wird. Die rastende Variante mit Flankenerkennung und Latch ist `Button_IXA_TO_logiBUS_QXA_BG_OPC_LATCHING`.

Zusammenfassung

`Button_IXA_TO_logiBUS_QXA_BG_OPC` verbindet einen digitalen Ausgang bidirektional mit OPC-UA: VT-Taster und Web-Kommando werden in `AX_2_3` ODER-verknüpft und der Pegel an Hardware, VT-Anzeige und OPC-UA-Echo verteilt; `INIT`/`INITO` sind durch die Dienste gekettet, `QO` ist das UND der Dienste.

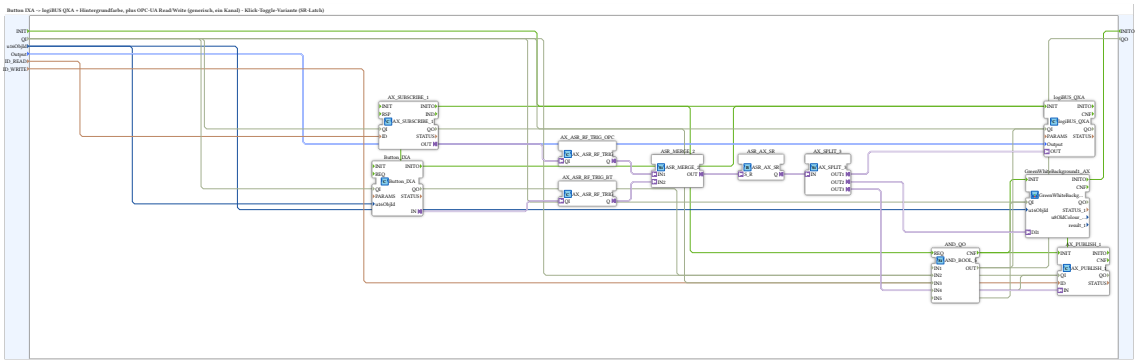
🛠️ Zugehörige Übungen

- InputOutputTesterButton_DIDO_OPC_UA

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

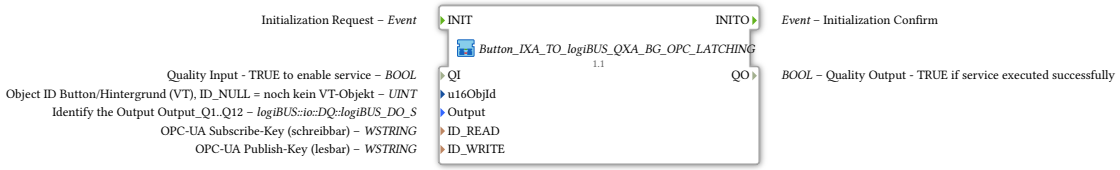
Button_IXA_TO_logiBUS_QXA_BG_OPC_LATCHING



• — ○ • — ○ • — ○ •

Einleitung

`Button_IXA_TO_logiBUS_QXA_BG_OPC_LATCHING` ersetzt bei `Button_IXA_TO_logiBUS_QXA_BG_OPC` die einfache ODER-Verknüpfung (`AX_OR_2`) durch eine Flankenerkennungs-/Merge-/Latch-Kette (`AX_ASR_RF_TRIG` × 2, `ASR_MERGE_2`, `ASR_AX_SR`). Das ist **kein** echtes Klick-Toggle: `AX_ASR_RF_TRIG` bildet die steigende Flanke (Drücken) auf `SET` und die fallende Flanke (Loslassen) auf `RESET` ab, `ASR_AX_SR` setzt/rücksetzt entsprechend — bei nur einer aktiven Quelle verhält sich der Baustein also tastend, genau wie die `AX_OR_2`-Variante (EIN nur solange gedrückt). Der Unterschied zeigt sich erst, wenn sich VT-Taste und OPC-UA-Kommando zeitlich überlappen: Anders als bei einer echten ODER-Verknüpfung (die EIN bleibt, solange irgendeine Quelle aktiv ist) gewinnt hier immer die zuletzt eingetroffene Flanke, unabhängig davon, von welcher Quelle sie stammt — löst z. B. die OPC-UA-Seite ein Loslassen aus, während die VT-Taste noch gehalten wird, schaltet der Ausgang trotzdem sofort AUS. Der Baustein ist aufbewahrt für Anwendungsfälle, in denen genau dieses Last-Wins-Verhalten zwischen zwei Quellen gebraucht wird. Beide Bausteine teilen dasselbe Interface (`u16ObjId`/`Output`/`ID_READ`/`ID_WRITE`) und sind schnittstellenkompatibel — ihr Schaltverhalten unterscheidet sich jedoch in genau diesem Überlappungsfall.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Button_IXA_TO_logiBUS_QXA_BG_OPC_LATCHING

- **Typ**: SubAppType

- **Schnittstelle (zusätzlich)**: Ereignis-Eingang `INIT`, Ereignis-Ausgang `INITO` (je `EInit`), Daten-Eingang `QI` (`BOOL`, Vorgabe `TRUE`), Daten-Ausgang `QO` (`BOOL`).
- **Verwendete interne FBs**:
  - **Button_IXA**: `isobus::UT::io::Button::Button_IXA` (`QI` vom Wrapper-Eingang) — VT-Taster, `u16ObjId` identifiziert die VT-Taste.
  - **AX_ASR_RF_TRIG_BT** / **AX_ASR_RF_TRIG_OPC**: je `adapter::events::unidirectional::AX_ASR_RF_TRIG` — wandeln steigende/fallende Flanke der VT-Taste bzw. des OPC-UA-Subscribe-Werts in ein Set/Reset-Ereignispaar (ASR) um.
  - **ASR_MERGE_2**: `adapter::events::unidirectional::ASR_MERGE_2` — führt die beiden ASR-Quellen (Button, OPC-UA) zusammen.
  - **ASR_AX_SR**: `adapter::events::unidirectional::ASR_AX_SR` — Set/Reset-Latch, das aus dem zusammengeführten ASR-Signal den gelatchten Ausgangszustand bildet (Last-Wins).
  - **AX_SPLIT_3**: `adapter::events::unidirectional::AX_SPLIT_3` — verteilt den Latch-Ausgang auf drei Ziele.
  - **logiBUS_QXA**: `logiBUS::io::DQ::logiBUS_QXA` (`QI` vom Wrapper-Eingang) — physischer digitaler Ausgang.
  - **GreenWhiteBackground1_AX** (SubApp, `MyLib::sys`): VT-Hintergrundfarbe passend zum Ausgangszustand.
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_5` — verknüpft die `QO`-Ausgänge von `logiBUS_QXA`, `Button_IXA`, `AX_SUBSCRIBE_1`, `AX_PUBLISH_1` und `GreenWhiteBackground1_AX` zu `QO`.
  - **AX_SUBSCRIBE_1** / **AX_PUBLISH_1**: je `adapter::net::AX_SUBSCRIBE_1`/`AX_PUBLISH_1` (`QI` vom Wrapper-Eingang) — OPC-UA-Lese-/Schreibzugriff.
- **Funktionsweise**: Beide Schaltquellen (VT-Taster, OPC-UA-Kommando) werden je über `AX_ASR_RF_TRIG` in Set/Reset-Ereignisse gewandelt, per `ASR_MERGE_2` zusammengeführt und in `ASR_AX_SR` gelatcht — der Latch-Zustand speist gleichzeitig Hardware, VT-Statusfarbe und OPC-UA-Echo.

Programmablauf und Verbindungen

1. `Button_IXA.IN` (Adapter) → `AX_ASR_RF_TRIG_BT.QI` → `AX_ASR_RF_TRIG_BT.Q` → `ASR_MERGE_2.IN2`.
2. `AX_SUBSCRIBE_1.OUT` → `AX_ASR_RF_TRIG_OPC.QI` → `AX_ASR_RF_TRIG_OPC.Q` → `ASR_MERGE_2.IN1`.
3. `ASR_MERGE_2.OUT` → `ASR_AX_SR.S_R` (Set/Reset-Eingang des Latch).
4. `ASR_AX_SR.Q` → `AX_SPLIT_3.IN` → `AX_SPLIT_3.OUT1` → `logiBUS_QXA.OUT`, `AX_SPLIT_3.OUT2` → `GreenWhiteBackground1_AX.DI1`, `AX_SPLIT_3.OUT3` → `AX_PUBLISH_1.IN`.
5. Initialisierung: `AX_SUBSCRIBE_1.INITO` → `AX_PUBLISH_1.INIT` (ausgeblendete Verbindung). Die Kette läuft von außen: `INIT` → `logiBUS_QXA.INIT`; `logiBUS_QXA.INITO` → `Button_IXA.INIT`; `Button_IXA.INITO` → `AX_SUBSCRIBE_1.INIT`; `AX_PUBLISH_1.INITO` → `GreenWhiteBackground1_AX.INIT`; `GreenWhiteBackground1_AX.INITO` → `INITO` und → `AND_QO.REQ`.
6. Parameter: `u16ObjId` → `Button_IXA.u16ObjId` und `GreenWhiteBackground1_AX.u16ObjId`; `Output` → `logiBUS_QXA.Output`; `ID_READ` → `AX_SUBSCRIBE_1.ID`; `ID_WRITE` → `AX_PUBLISH_1.ID`.
7. Qualität: `QI` → `.QI` von `logiBUS_QXA`, `Button_IXA`, `AX_SUBSCRIBE_1`, `AX_PUBLISH_1` und `GreenWhiteBackground1_AX`; deren `QO` → `AND_QO.IN1` bis `AND_QO.IN5` (in dieser Reihenfolge); `AND_QO.OUT` → `QO`.

Technische Besonderheiten

- **Last-Wins über ASR-Latch, kein echtes Toggle**: Jede Betätigung erzeugt je Quelle ein Set- (Drücken) bzw. Reset-Ereignis (Loslassen); `ASR_AX_SR` latcht daraus den gemeinsamen Zustand. Bei nur einer aktiven Quelle ist das Verhalten rein tastend (EIN nur solange gedrückt) — identisch zur `AX_OR_2`-Variante. Erst wenn sich beide Quellen überlappen, zeigt sich der Unterschied: die zuletzt eingetroffene Flanke (Set oder Reset, unabhängig von welcher Quelle) bestimmt den Zustand, nicht eine kontinuierliche ODER-Verknüpfung.
- **Zwei unabhängige Last-Wins-Quellen**: VT-Button und OPC-UA-Kommando liefern je eigenständig Set-/Reset-Ereignisse; `ASR_MERGE_2` führt beide zu einem gemeinsamen Latch-Eingang zusammen, ohne eine Quelle zu priorisieren.
- **Schnittstellenkompatibles Außeninterface**: Die Schnittstelle ist identisch zu `Button_IXA_TO_logiBUS_QXA_BG_OPC`, sodass beide Varianten austauschbar instanziiert werden können. Das Schaltverhalten ist aber nur bei einer einzelnen aktiven Quelle gleich — bei überlappender Nutzung beider Quellen unterscheidet es sich (Last-Wins vs. echte ODER-Verknüpfung).

Anwendungsszenarien

- Ausgänge mit zwei unabhängigen Schaltquellen (z. B. lokale VT-Taste UND Remote-OPC-UA-Kommando), bei denen die jeweils zuletzt eingetroffene Aktion (Drücken oder Loslassen, von welcher Quelle auch immer) gewinnen soll, statt dass eine dauerhaft aktive Quelle die andere blockiert — genau das würde eine echte ODER-Verknüpfung tun.

Vergleich mit ähnlichen Bausteinen

Gegenüber `Button_IXA_TO_logiBUS_QXA_BG_OPC` (aktuell: tastendes `AX_OR_2`) ersetzt dieser Baustein die einfache ODER-Verknüpfung durch eine Flankenerkennungs-/Merge-/Latch-Kette (`AX_ASR_RF_TRIG` × 2, `ASR_MERGE_2`, `ASR_AX_SR`). Bei nur einer aktiven Quelle ist das Verhalten identisch (tastend); der Unterschied zeigt sich nur, wenn sich beide Quellen überlappen (Last-Wins statt ODER).

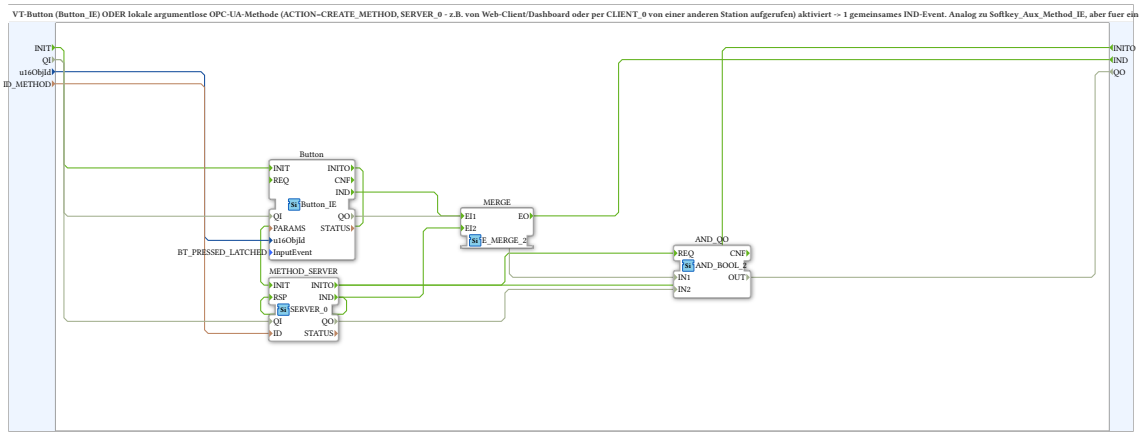
Zusammenfassung

`Button_IXA_TO_logiBUS_QXA_BG_OPC_LATCHING` bietet bei nur einer aktiven Quelle dieselbe tastende Grundfunktion wie `Button_IXA_TO_logiBUS_QXA_BG_OPC` — der Unterschied liegt im Last-Wins-Verhalten bei zeitlich überlappenden VT-/OPC-UA-Kommandos, nicht in einem Klick-Toggle. Beide Bausteine sind schnittstellenkompatibel, aber nicht in jedem Szenario verhaltensgleich.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

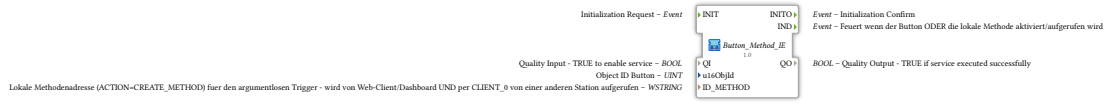
Button_Method_IE



• — ○ • — ○ • — ○ •

Einleitung

`Button_Method_IE` löst ein gemeinsames `IND`-Ereignis aus, wenn entweder ein **VT-Button** (`Button_IE`) aktiviert oder eine **lokale argumentlose OPC-UA-Methode** (`ACTION=CREATE_METHOD`, `SERVER_0`) aufgerufen wird, z. B. vom Web-Client/Dashboard oder per `CLIENT_0` von einer anderen Station. Der Baustein ist analog zu `Softkey_Aux_Method_IE`, aber für einen regulären VT-Button statt SoftKey/AUX. Er wurde aus `logiBUS_AI_Calibrate_IDA_OPC` herausgelöst, wo die Kombination (CO/CS) zweimal identisch vorkam.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialization Request

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialization Confirm

- **IND**: feuert, wenn der Button **oder** die lokale Methode aktiviert bzw. aufgerufen wird

Daten-Eingänge

- **QI** (BOOL, Vorgabe TRUE): Quality Input - TRUE, um den Dienst freizugeben
- **u16ObjId** (UINT): Object ID des Buttons
- **ID_METHOD** (WSTRING): lokale Methodenadresse (ACTION=CREATE_METHOD) für den argumentlosen Trigger; wird vom Web-Client/Dashboard und per CLIENT_0 von einer anderen Station aufgerufen

Daten-Ausgänge

- **QO** (BOOL): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde (UND von Button.QO und METHOD_SERVER.QO)

Adapter

Keine.

Verwendete Funktionsbausteine (FBs)

- **Button** (isobus::UT::io::Button::Button_IE): Parameter InputEvent = BT_PRESSED_LATCHED, QI vom Baustein-Eingang
- **METHOD_SERVER** (iec61499::net::SERVER_0): QI vom Baustein-Eingang, stellt die lokale Methode bereit
- **MERGE** (iec61499::events::E_MERGE_2): führt beide Auslöser zusammen
- **AND_QO** (iec61131::booleanOperators::AND_BOOL_2): verknüpft Button.QO und METHOD_SERVER.QO per UND zu QO

Funktionsweise

- u16ObjId ⟶ Button.u16ObjId; ID_METHOD ⟶ METHOD_SERVER.ID
- Button.IND ⟶ MERGE.EI1
- METHOD_SERVER.IND ⟶ MERGE.EI2
- METHOD_SERVER.IND ⟶ METHOD_SERVER.RSP (sofortige Antwort)
- MERGE.EO ⟶ IND
- Initialisierungskette: INIT ⟶ Button.INIT, Button.INITO ⟶ METHOD_SERVER.INIT, METHOD_SERVER.INITO ⟶ INITO und ⟶ AND_QO.REQ
- QI ⟶ Button.QI und METHOD_SERVER.QI
- Button.QO ⟶ AND_QO.IN1, METHOD_SERVER.QO ⟶ AND_QO.IN2, AND_QO.OUT ⟶ QO

Technische Besonderheiten

- **RSP muss auf IND zurückverdrahtet werden.** Der Trigger ist argumentlos und braucht keine Verarbeitung. Ohne Rückverdrahtung blockiert onServerMethodCall in FORTE den OPC-UA-Server-Thread bis zu 4 s und hält dabei den globalen serviceMutex von open62541, den jeder AX_PUBLISH_1 auf einem anderen FORTE-Thread ebenfalls benötigt (siehe Ventilsteuerung/OPC_UA_VERNETZUNG.md, Punkt 12).
- BT_PRESSED_LATCHED als Button-Ereignis (rastendes Drücken/Loslassen)

Anwendungsszenarien

- Button, der zusätzlich per Web-Client, OPC-Dashboard oder von einer anderen Station ausgelöst werden soll (z. B. Kalibrier-Trigger)

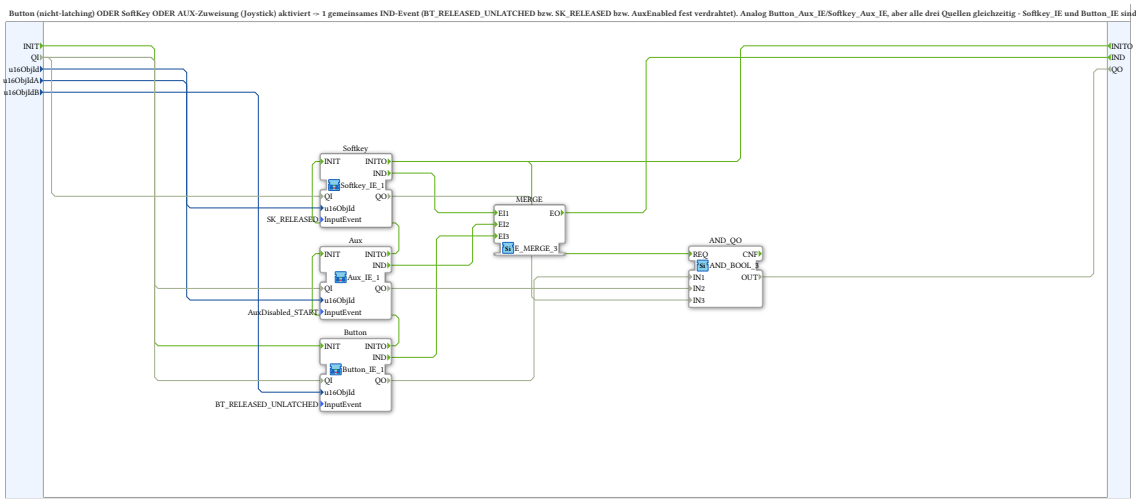
Zusammenfassung

Button_Method_IE vereint VT-Button und lokale OPC-UA-Methode zu einem gemeinsamen IND-Ereignis.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Button_Softkey_Aux_IE

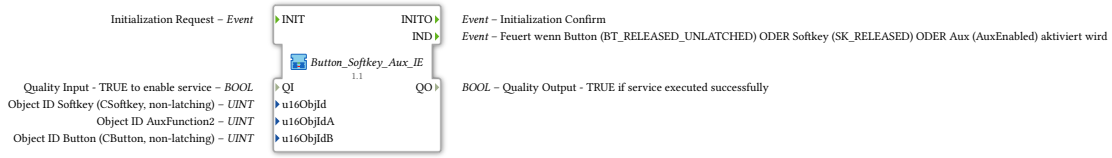


• — ○ • — ○ • — ○ •

Einleitung

`Button_Softkey_Aux_IE` fasst einen nicht-latchenden VT-Button **oder** einen SoftKey **oder** eine AUX-Zuweisung (Joystick) zu einem gemeinsamen `IND`-Ereignis zusammen. Die Aktivierungsarten sind fest verdrahtet (`BT_RELEASED_UNLATCHED`, `SK_RELEASED` bzw. `AuxEnabled`). Der Baustein entspricht `Button_Aux_IE` und `Softkey_Aux_IE`, bedient aber alle drei Quellen gleichzeitig. `Softkey_IE` und `Button_IE` sind getrennte FORTE-Service-Interface-Bausteine für unterschiedliche VT-Activation-Messages; ein Button liefert nie ein `SK_PRESSED`.

Der Button **muss** non-latching sein (`Latchable=0` im Pool); ein latchender Button ist sinnvoll nur über `Button_IXA` (Pegel) lesbar.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Button_Softkey_Aux_IE

- **Typ**: SubAppType
- **Verwendete interne Bausteine**:
  - **Button**: SubApp `Button_IE_1` mit `InputEvent=BT_RELEASED_UNLATCHED`
  - **Softkey**: SubApp `Softkey_IE_1` mit `InputEvent=SK_RELEASED`
  - **Aux**: SubApp `Aux_IE_1` mit `InputEvent=AuxDisabled_START`
  - **MERGE**: `iec61499::events::E_MERGE_3` – führt die drei `IND`-Ereignisse zusammen
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_3` – verknüpft die `QO`-Ausgänge von `Button`, `Aux` und `Softkey` zum SubApp-Ausgang `QO`
- **Importe**: `BT_RELEASED_UNLATCHED`, `AuxDisabled_START`, `SK_RELEASED`

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; startet die Kette `Button` ⟶ `Aux` ⟶ `Softkey`

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung; kommt von `Softkey.INITO`
- **IND**: Feuert, wenn Button (`BT_RELEASED_UNLATCHED`), Softkey (`SK_RELEASED`) oder Aux (`AuxEnabled`) aktiviert wird

Daten-Eingänge

- **QI** (`BOOL`): Quality Input, Freigabe des Dienstes (Initialwert `TRUE`); wird an `Button.QI`, `Aux.QI` und `Softkey.QI` weitergereicht
- **u16ObjId** (`UINT`): Objekt-ID des Softkeys (`CSoftkey`, non-latching)
- **u16ObjIdA** (`UINT`): Objekt-ID der `AuxFunction2`
- **u16ObjIdB** (`UINT`): Objekt-ID des Buttons (`CButton`, non-latching)

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output, `AND` der `QO`-Ausgänge von `Button`, `Aux` und `Softkey` (über `AND_QO`)

Programmablauf und Verbindungen

- `u16ObjId` ⟶ `Softkey.u16ObjId`; `Softkey.IND` ⟶ `MERGE.EI1`
- `u16ObjIdA` ⟶ `Aux.u16ObjId`; `Aux.IND` ⟶ `MERGE.EI2`
- `u16ObjIdB` ⟶ `Button.u16ObjId`; `Button.IND` ⟶ `MERGE.EI3`
- `MERGE.EO` ⟶ `IND`
- Initialisierung: `INIT` ⟶ `Button.INIT`; `Button.INITO` ⟶ `Aux.INIT`; `Aux.INITO` ⟶ `Softkey.INIT`; `Softkey.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`
- Qualität: `QI` ⟶ `Button.QI`, `Aux.QI` und `Softkey.QI`; `Button.QO` ⟶ `AND_QO.IN1`, `Aux.QO` ⟶ `AND_QO.IN2`, `Softkey.QO` ⟶ `AND_QO.IN3`; `AND_QO.OUT` ⟶ `QO`

Technische Besonderheiten

- Version 1.1 (2026-09-29): `packageName`-Tippfehler behoben (`MyLib_AX::sys` ⟶ `MyLib::sys`), Instanz `Softkey_IE_1` in `Softkey` umbenannt, Kommentar korrigiert (beschrieb nur 2 von 3 Quellen)
- Version 1.0 (2026-09-20): Bugfix für `Automatik_Toggle_Button`, der bisher `Softkey_Aux_IE` mit einer Button-Objekt-ID nutzte

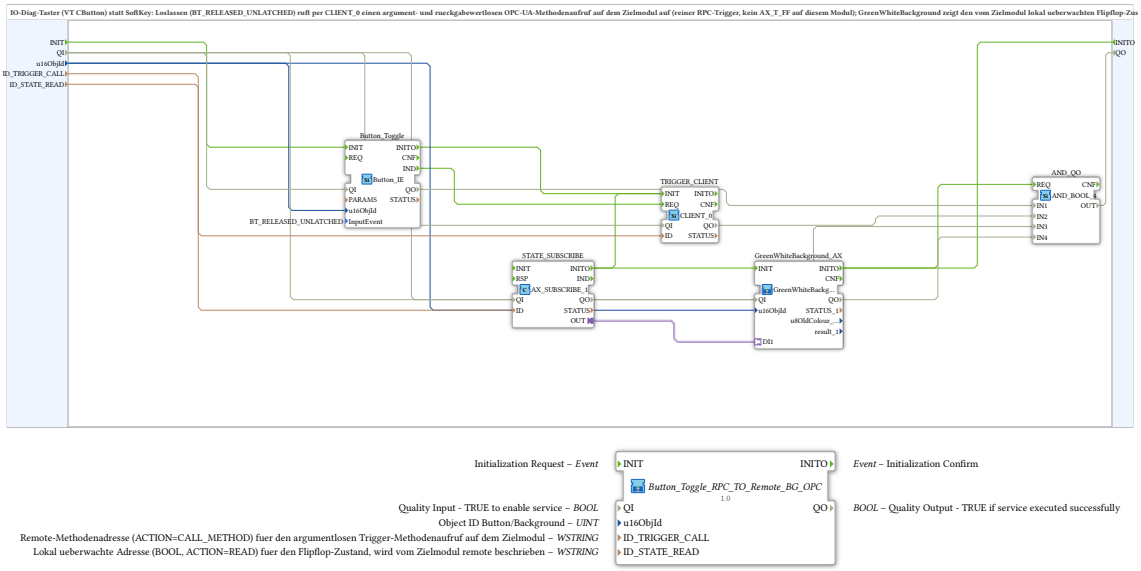
Anwendungsszenarien

- Ein Bedienelement, das wahlweise über Button, SoftKey oder AUX-Eingang ausgelöst werden kann

Zusammenfassung

`Button_Softkey_Aux_IE` bündelt `Button_IE_1`, `Softkey_IE_1` und `Aux_IE_1` über einen `E_MERGE_3` zu einem einzigen `IND`-Ereignis.

Button_Toggle_RPC_TO_Remote_BG_OPC



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein `Button_Toggle_RPC_TO_Remote_BG_OPC` ist eine Subapplikation (SubApp) zur Realisierung eines Taster-basierten Remote-Prozeduraufrufs (RPC) über OPC-UA. Er dient als Analogie zum vorhandenen Softkey-Baustein `Softkey_Toggle_RPC_TO_Remote_BG_OPC`, ist jedoch für den Einsatz mit

IO-Diagnose-Tastern (z. B. CButton) konzipiert. Der Baustein reagiert auf das Ereignis „Loslassen“ (BT_RELEASED_UNLATCHED) eines Tasters und löst daraufhin einen argument- und rückgabewertlosen OPC-UA-Methodenaufruf auf einem entfernten Zielmodul aus. Zusätzlich überwacht er den Zustand eines Flipflops auf dem Zielmodul und visualisiert diesen über eine Hintergrundfarb-Darstellung (GreenWhiteBackground). Das Protokoll ist identisch zum Softkey-Pendant, verwendet jedoch eine Taster-Eingangsschnittstelle (Button_IE) statt einer Softkey-Schnittstelle.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; startet die Init-Kette bei `Button_Toggle.INIT`.

Die eigentliche Ereignisauslösung erfolgt weiterhin intern über den verwendeten `Button_IE`-Funktionsbaustein, der auf das Tasterereignis `BT_RELEASED_UNLATCHED` reagiert.

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung; kommt von `GreenWhiteBackground_AX.INITO`, dem Ende der Init-Kette.

Sonst gibt es keine Ereignis-Ausgänge. Der Baustein ist als reiner Trigger konzipiert; die Ergebnisrückmeldung erfolgt über den überwachten Zustand.

Daten-Eingänge

Name	Typ	Initialwert	Kommentar
`QI`	BOOL	`TRUE`	Quality Input, schaltet den Dienst frei; wird an `Button_Toggle`, `TRIGGER_CLIENT`, `STATE_SUBSCRIBE` und `GreenWhiteBackground_AX` weitergereicht.
`u16ObjId`	UINT	`ID_NULL`	Objekt-ID des Tasters bzw. der Hintergrundanzeige. Wird sowohl an den Taster-Baustein als auch an die Anzeige-SubApp weitergegeben.
`ID_TRIGGER_CALL`	WSTRING	–	Remote-Methodenadresse (ACTION=CALL_METHOD) für den argumentlosen Trigger-Methodenaufruf auf dem Zielmodul.
`ID_STATE_READ`	WSTRING	–	Lokal überwachte Adresse (BOOL, ACTION=READ) für den Flipflop-Zustand, der vom Zielmodul remote beschrieben wird.

Daten-Ausgänge

- **QO** (BOOL): Quality Output, UND-Verknüpfung (`AND_QO`, `AND_BOOL_4`) der `QO`-Ausgänge von `Button_Toggle`, `TRIGGER_CLIENT`, `STATE_SUBSCRIBE` und `GreenWhiteBackground_AX`.

Adapter

Der Baustein verwendet intern einen Adapter `AX_SUBSCRIBE_1` für die Abonnement-Übertragung des Zustandswerts. Von außen ist kein Adapter sichtbar, da die Verbindung intern verdrahtet ist.

Funktionsweise

Der Baustein kombiniert mehrere Funktionsbausteine und SubApplikationen:

1. **Taster-Erkennung**: Der FB `Button_IE` (vom Typ `isobus::UT::io::Button::Button_IE`) wird mit dem Aktivierungscode `BT_RELEASED_UNLATCHED` konfiguriert. Dieser reagiert auf das Loslassen eines Tasters (unverriegelt) und erzeugt ein Ereignis am Ausgang `IND`.
2. **Remote-Call-Triggerung**: Das Ereignis `IND` wird mit dem Eingang `REQ` des FB `TRIGGER_CLIENT` (vom Typ `iec61499::net::CLIENT_0`) verbunden. Der Client führt einen OPC-UA-Methodenaufruf gemäß der übergebenen Adresse `ID_TRIGGER_CALL` aus. Da keine Parameter und Rückgabewerte vorhanden sind, dient dies als reiner RPC-Trigger.

3. **Zustandsüberwachung**: Der FB `STATE_SUBSCRIBE` (Typ `adapter::net::AX_SUBSCRIBE_1`) abonniert die über `ID_STATE_READ` spezifizierte Bool-Adresse auf dem Zielmodul. Der empfangene Zustand wird über den Adapterausgang `OUT` an die SubApp `GreenWhiteBackground_AX` weitergegeben.
4. **Hintergrundvisualisierung**: Die SubApp `GreenWhiteBackground_AX` (Typ `MyLib::sys::GreenWhiteBackground1_AX`) empfängt den Zustand über den Adapter `DI1` und stellt den Flipflop-Zustand farblich dar (z. B. grün für aktiv, weiß für inaktiv).
5. **Initialisierung und Qualität**: `INIT` → `Button_Toggle.INIT`, `Button_Toggle.INITO` → `TRIGGER_CLIENT.INIT`, `TRIGGER_CLIENT.INITO` → `STATE_SUBSCRIBE.INIT`, `STATE_SUBSCRIBE.INITO` → `GreenWhiteBackground_AX.INIT`, `GreenWhiteBackground_AX.INITO` → `INITO` und `AND_QO.REQ`. `QI` wird an alle vier Bausteine weitergereicht; deren `QO`-Ausgänge (`Button_Toggle`, `TRIGGER_CLIENT`, `STATE_SUBSCRIBE`, `GreenWhiteBackground_AX`) gehen auf `AND_QO.IN1` bis `AND_QO.IN4` (`iec61131::booleanOperators::AND_BOOL_4`), dessen `OUT` den Ausgang `QO` bildet.

Die Verbindung zwischen `u16ObjId` und den internen Bausteinen stellt sicher, dass die korrekte Objekt-ID sowohl für den Taster als auch für die Anzeige verwendet wird.

Technische Besonderheiten

- **Identisches Protokoll**: Der Baustein verwendet exakt dasselbe Kommunikationsprotokoll wie der Softkey-Baustein `Softkey_Toggle_RPC_TO_Remote_BG_OPC`, nur die Eingangsschnittstelle unterscheidet sich (Button statt Softkey).
- **Keine Örtliche Logik**: Im Gegensatz zu einem lokalen Toggle-Flipflop (z. B. `AX_T_FF`) wird der Flipflop-Zustand ausschließlich auf dem Zielmodul gehalten und überwacht.
- **RPC ohne Argmente**: Der Methodenaufruf erfolgt ohne Argumente und Rückgabewerte – reiner Trigger.
- **SubApp-Kapselung**: Die gesamte Funktionalität ist in einer SubApp gekapselt, was Wiederverwendung und Übersichtlichkeit erhöht.
- **Attribut „Visible“**: Die internen Datenverbindungen sind mit `Visible=false` markiert, um die Netzansicht in der IDE zu verschlanken.

Zustandsübersicht

Der Baustein selbst besitzt keinen expliziten Zustandsautomaten. Der überwachte Flipflop-Zustand (Bool) kann zwei Werte annehmen:

- **TRUE**: Taster gedrückt/aktiviert → Hintergrund grün
- **FALSE**: Taster losgelassen/deaktiviert → Hintergrund weiß

Die Zustandsänderung wird durch den OPC-UA-Aufruf am Zielmodul ausgelöst und über das Abonnement zurückgelesen.

Anwendungsszenarien

- **Remote-Toggle-Schalter**: Ein Taster an einer IO-Diagnose-Maske soll einen Flipflop auf einem entfernten Steuerungsmodul umschalten und den aktuellen Zustand vor Ort anzeigen.
- **Freigabeschalter**: In einer Fertigungsanlage können Taster zum Ein-/Ausschalten von Funktionen (z. B. Aufnahme links/rechts) verwendet werden, wobei der Zustand auf dem Zielmodul gehalten wird.
- **Bedienpanel**: Einsatz in Bedienoberflächen, bei denen physische Taster statt Softkeys verwendet werden, aber das gleiche Kommunikationsprotokoll wie bei Softkeys genutzt werden soll.

Vergleich mit ähnlichen Bausteinen

Baustein	Eingangsschnittstelle	Verwendung
`Button_Toggle_RPC_TO_Remote_BG_OPC`	Button_IE	Taster an IO-Diag-DataMask
`Softkey_Toggle_RPC_TO_Remote_BG_OPC`	Softkey_IE	Softkey an SoftkeyMask

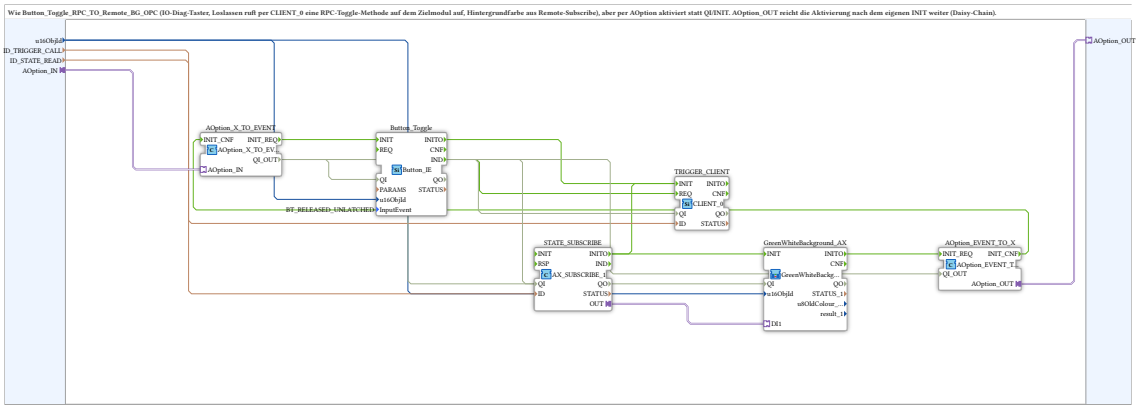
Beide Bausteine nutzen das gleiche RPC-Trigger- und Zustandsüberwachungsprinzip, unterscheiden sich jedoch in der Art der Bedienung (physischer Taster vs. Softkey).

Ein alternativer Ansatz wäre ein lokales Flipflop (z. B. `AX_T_FF`), bei dem der Zustand nicht remote sondern lokal gehalten wird. Dies würde jedoch zusätzliche Synchronisationslogik erfordern.

Fazit

Der Baustein `Button_Toggle_RPC_TO_Remote_BG_OPC` bietet eine kompakte und wiederverwendbare Lösung für die Anbindung physischer Taster an eine OPC-UA-basierte Remote-Steuerung. Durch die Kapselung in eine SubApp und die Verwendung standardisierter Schnittstellen (`CLIENT_0`, `AX_SUBSCRIBE_1`) ist er flexibel und in verschiedene Automatisierungssysteme integrierbar. Die klare Trennung zwischen Trigger und Zustandsvisualisierung erleichtert Wartung und Anpassung an unterschiedliche Zielmodule.

Button_Toggle_RPC_TO_Remote_BG_Option_OPC



• — ○ • — ○ • — ○ •

Einleitung

`Button_Toggle_RPC_TO_Remote_BG_Option_OPC` ist ein IO-Diag-Taster: Beim Loslassen ruft er per `CLIENT_0` eine RPC-Toggle-Methode auf dem Zielmodul auf, die Hintergrundfarbe kommt aus einem Remote-Subscribe. Der Baustein entspricht `Button_Toggle_RPC_TO_Remote_BG_OPC`, wird aber per `AOption` aktiviert statt über `QI`/`INIT`. `AOption_OUT` reicht die Aktivierung nach dem eigenen `INIT` weiter (Daisy-Chain wie `OptionalGroup_001a`). Er ist eine Kopie von `Button_Toggle_RPC_TO_Remote_BG_OPC`, bei der `INIT`/`INITO`/`QI`/`QO` durch `AOption_IN` und `AOption_OUT` ersetzt wurden.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Variable	Typ	Vorgabe	Kommentar
`u16ObjId`	UINT	`ID_NULL`	Object ID Button/Hintergrund.
`ID_TRIGGER_CALL`	WSTRING		Remote-Methodenadresse (`ACTION=CALL_METHOD`) für den argumentlosen Trigger-Methodenaufruf auf dem Zielmodul.
`ID_STATE_READ`	WSTRING		Lokal überwachte Adresse (`BOOL`, `ACTION=READ`) für den Flipflop-Zustand; wird vom Zielmodul remote beschrieben.

Daten-Ausgänge

Keine vorhanden.

Adapter

Adapter	Typ	Richtung	Kommentar
AOption_IN	Option::AOption	Socket	Treibt INIT/QI; anschließen an AOption_AUI_DEMUX_N.OUTx oder AOption_OUT einer vorgeschalteten Instanz.
AOption_OUT	Option::AOption	Plug	Reicht die Aktivierung nach dem eigenen INIT an das nächste Glied weiter (Daisy-Chain); muss verbunden sein, Kettenende z. B. DummyGroup.

Verwendete Funktionsbausteine (FBs)

- **AOption_X_TO_EVENT** (Option::conversion::AOption_X_TO_EVENT): wandelt die Option in INIT_REQ/QI_OUT und nimmt INIT_CNF entgegen
- **Button_Toggle** (isobus::UT::io::Button::Button_IE, InputEvent = BT_RELEASED_UNLATCHED): VT-Button, Ereignis beim Loslassen
- **TRIGGER_CLIENT** (iec61499::net::CLIENT_0): ruft die Methode auf dem Zielmodul auf
- **STATE_SUBSCRIBE** (adapter::net::AX_SUBSCRIBE_1): Remote-Subscribe des Zustands
- **GreenWhiteBackground_AX** (SubApp MyLib::sys::GreenWhiteBackground1_AX): Hintergrundfarbe; siehe GreenWhiteBackground1_AX
- **AOption_EVENT_TO_X** (Option::conversion::AOption_EVENT_TO_X): gibt die Aktivierung an AOption_OUT weiter

Programmablauf und Verbindungen

1. Initialisierungskette: AOption_X_TO_EVENT.INIT_REQ -> Button_Toggle.INIT; Button_Toggle.INITO -> TRIGGER_CLIENT.INIT; TRIGGER_CLIENT.INITO -> STATE_SUBSCRIBE.INIT; STATE_SUBSCRIBE.INITO -> GreenWhiteBackground_AX.INIT; GreenWhiteBackground_AX.INITO -> AOption_EVENT_TO_X.INIT_REQ; AOption_EVENT_TO_X.INIT_CNF -> AOption_X_TO_EVENT.INIT_CNF.
2. Methodenaufruf: Button_Toggle.IND -> TRIGGER_CLIENT.REQ.
3. Qualifier: AOption_X_TO_EVENT.QI_OUT -> Button_Toggle.QI, TRIGGER_CLIENT.QI, STATE_SUBSCRIBE.QI, GreenWhiteBackground_AX.QI und AOption_EVENT_TO_X.QI_OUT.
4. Option: AOption_IN -> AOption_X_TO_EVENT.AOption_IN; AOption_EVENT_TO_X.AOption_OUT -> AOption_OUT.
5. Status: STATE_SUBSCRIBE.OUT -> GreenWhiteBackground_AX.DI1.
6. Daten: u16ObjId -> Button_Toggle.u16ObjId und GreenWhiteBackground_AX.u16ObjId; ID_TRIGGER_CALL -> TRIGGER_CLIENT.ID; ID_STATE_READ -> STATE_SUBSCRIBE.ID.

Technische Besonderheiten

- Der Baustein hat keine eigenen Ereignis- und QI/QO-Anschlüsse; Aktivierung und Bestätigung laufen ausschließlich über AOption_IN/AOption_OUT.
- AOption_OUT muss weiterverbunden sein (z. B. DummyGroup als Kettenende), sonst kommt kein INIT_CNF zurück.
- Der Zustand wird nicht hier berechnet, sondern vom Zielmodul remote geliefert.

Anwendungsszenarien

- Optionaler Diagnose-Taster, der ein Ausgangs-Toggle auf einem entfernten Modul auslöst und dessen Zustand farbig anzeigt, aktiv nur bei ausgewählter Option.
- Mehrere Instanzen, die dieselbe Option teilen und per AOption_OUT verkettet werden.

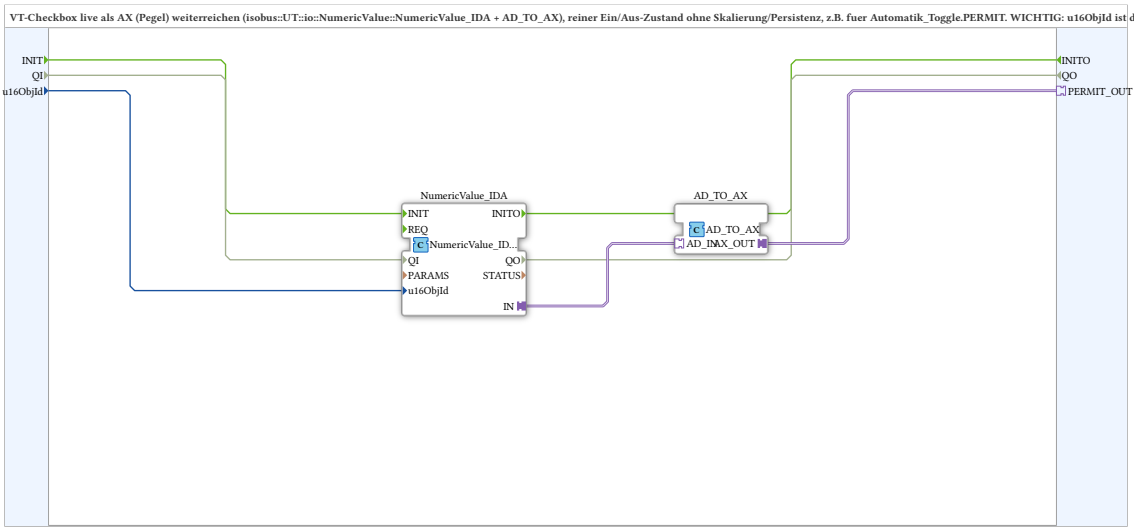
Zusammenfassung

Button_Toggle_RPC_TO_Remote_BG_Option_OPC löst beim Loslassen einen Remote-Methodenaufruf aus, zeigt den remote gelieferten Zustand als Hintergrundfarbe und wird per AOption aktiviert und verkettet.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Checkbox_IDA_TO_AX

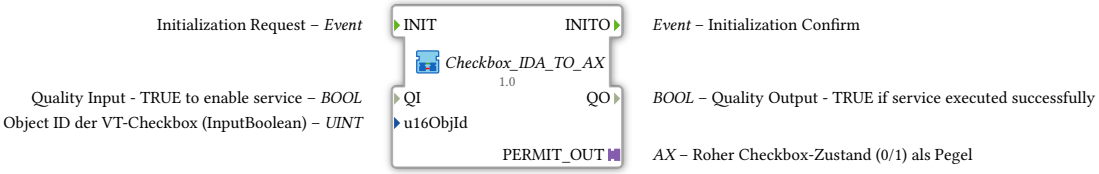


• – ○ • – ○ • – ○ •

Einleitung

`Checkbox_IDA_TO_AX` reicht den Zustand einer **VT-Checkbox live als AX-Pegel** weiter (`isobus::UT::io::NumericValue::NumericValue_IDA + AD_TO_AX`). Es ist ein reiner Ein/Aus-Zustand ohne Skalierung und Persistenz, z. B. für `Automatik_Toggle.PERMIT`.

Wichtig: `u16ObjId` ist die Object ID der **`CNumberVariable`**, die die Checkbox als Wertspeicher referenziert (z. B. `NumberVariable_Bunker_Fuellstand_Hoehe_Automatik`) – **nicht** die Object ID der Checkbox (`CInputBoolean`) selbst. Die VT meldet Wertänderungen unter der NumberVariable-ID (live verifiziert).



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; geht an `NumericValue_IDA.INIT`

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierung abgeschlossen; kommt von `NumericValue_IDA.INITO`

Daten-Eingänge

- **QI** (`BOOL`): Quality Input - TRUE aktiviert den Dienst (Default `TRUE`); geht an `NumericValue_IDA.QI`
- **u16ObjId** (`UINT`): Object ID der Zahlenvariable, die die VT-Checkbox (InputBoolean) als Wertspeicher nutzt (Default `ID_NULL`)

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde (kommt von `NumericValue_IDA.QO`)

Adapter

- **PERMIT_OUT** (Plug, `adapter::types::unidirectional::AX`): roher Checkbox-Zustand (0/1) als Pegel

Verwendete Funktionsbausteine (FBs)

- **`NumericValue_IDA`** (`isobus::UT::io::NumericValue::NumericValue_IDA`): liefert den Wert der Zahlenvariable als `AD`-Adapter
- **`AD_TO_AX`** (`adapter::conversion::unidirectional::AD_TO_AX`): wandelt den Wert in einen `AX`-Pegel

Funktionsweise

- `u16ObjId` ⟶ `NumericValue_IDA.u16ObjId`
- `NumericValue_IDA.IN` ⟶ `AD_TO_AX.AD_IN`
- `INIT` ⟶ `NumericValue_IDA.INIT`, `NumericValue_IDA.INITO` ⟶ `INITO`
- `QI` ⟶ `NumericValue_IDA.QI`, `NumericValue_IDA.QO` ⟶ `QO`
- `AD_TO_AX.AX_OUT` ⟶ `PERMIT_OUT`

Technische Besonderheiten

- Rein als Pegel; keine Skalierung, keine Persistenz
- Variante mit Initialisierung: Checkbox_IDA_TO_AX_INIT_Force1

Anwendungsszenarien

- Freigabe (`PERMIT`) einer Automatik per Checkbox im VT, z. B. für `Automatik_Toggle`

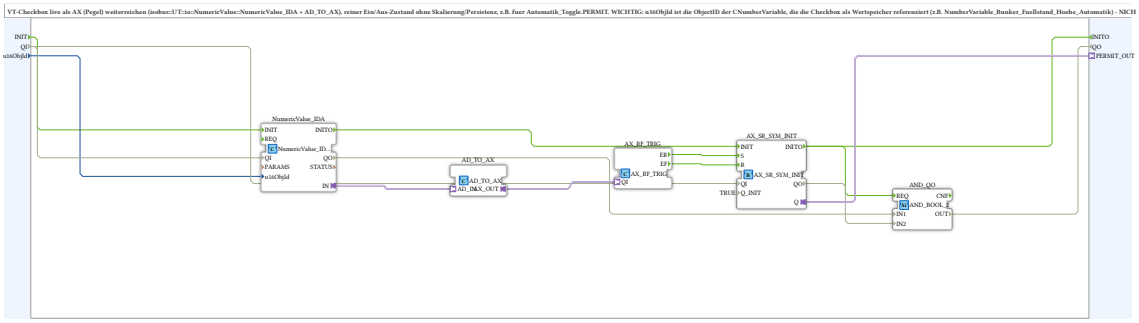
Zusammenfassung

`Checkbox_IDA_TO_AX` macht aus einer VT-Checkbox einen `AX`-Pegel für die weitere Logik.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Checkbox_IDA_TO_AX_INIT

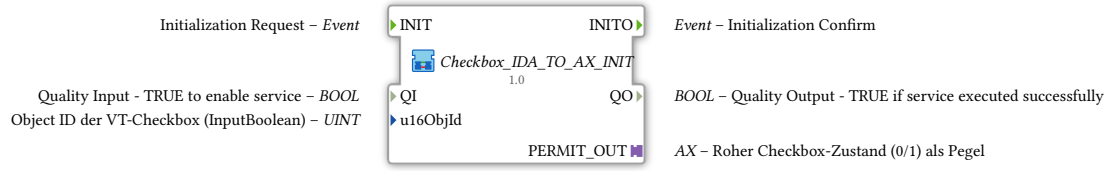


• – ○ • – ○ • – ○ •

Einleitung

`Checkbox_IDA_TO_AX_INIT` reicht den Zustand einer VT-Checkbox live als `AX`-Pegel weiter (`isobus::UT::io::NumericValue::NumericValue_IDA` + `AD_TO_AX`). Es ist ein reiner Ein/Aus-Zustand ohne Skalierung oder Persistenz, z. B. für `Automatik_Toggle.PERMIT`. Der Zustand wird über einen Flankenwächter und ein SR-Flipflop mit definiertem Initialwert gehalten (`_INIT`).

Wichtig: `u16ObjId` ist die ObjectID der `CNumberVariable`, die die Checkbox als Wertspeicher referenziert (z. B. `NumberVariable_Bunker_Fuellstand_Hoehe_Automatik`) – **nicht** die ObjectID der Checkbox (`CInputBoolean`) selbst. Die VT meldet Wertänderungen unter der NumberVariable-ID (live verifiziert).



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Checkbox_IDA_TO_AX_INIT

- **Typ**: SubAppType

- **Verwendete interne FBs**:
  - **NumericValue_IDA**: `isobus::UT::io::NumericValue::NumericValue_IDA` – liest den Rohwert per ObjectID
  - **AD_TO_AX**: `adapter::conversion::unidirectional::AD_TO_AX` – wandelt den `DWORD`-Wert in ein `AX`-Signal (Wert `<> 0`)
  - **AX_RF_TRIG**: `adapter::events::unidirectional::AX_RF_TRIG` – erkennt steigende (`ER`) und fallende (`EF`) Flanke
  - **AX_SR_SYM_INIT**: `adapter::events::unidirectional::AX_SR_SYM_INIT` (`Q_INIT=TRUE`) – Flip-flop mit definiertem Initialwert
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_2` – verknüpft `NumericValue_IDA.QO` und `AX_SR_SYM_INIT.QO` zum `QO`
- **Import**: `isobus::UT::Q::const::IDs::ID_NULL` (Initialwert von `u16ObjId`)

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung

Die Nutzereignisse laufen weiterhin über die Adapter.

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input – TRUE gibt den Dienst frei
- **u16ObjId** (`UINT`, Initialwert `ID_NULL`): Objekt-ID der VT-Checkbox bzw. der von ihr referenzierten NumberVariable

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output – UND aus `NumericValue_IDA.QO` und `AX_SR_SYM_INIT.QO`

Adapter

- **PERMIT_OUT** (Plug): `adapter::types::unidirectional::AX` – roher Checkbox-Zustand (0/1) als Pegel

Programmablauf und Verbindungen

- `u16ObjId` ⟶ `NumericValue_IDA.u16ObjId`
- `NumericValue_IDA.IN` ⟶ `AD_TO_AX.AD_IN`
- `AD_TO_AX.AX_OUT` ⟶ `AX_RF_TRIG.QI`
- `AX_RF_TRIG.ER` ⟶ `AX_SR_SYM_INIT.S` (Checkbox gesetzt)
- `AX_RF_TRIG.EF` ⟶ `AX_SR_SYM_INIT.R` (Checkbox gelöscht)
- `AX_SR_SYM_INIT.Q` ⟶ `PERMIT_OUT`
- `INIT` ⟶ `NumericValue_IDA.INIT`; `NumericValue_IDA.INITO` ⟶ `AX_SR_SYM_INIT.INIT`; `AX_SR_SYM_INIT.INITO` ⟶ `INITO` (und ⟶ `AND_QO.REQ`)
- `QI` ⟶ `NumericValue_IDA.QI` und `QI` ⟶ `AX_SR_SYM_INIT.QI`
- `NumericValue_IDA.QO` ⟶ `AND_QO.IN1`; `AX_SR_SYM_INIT.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`

Technische Besonderheiten

- Initialwert: `AX_SR_SYM_INIT` ist mit `Q_INIT=TRUE` parametriert
- Die Checkbox wird roh per ObjectID gelesen; es gibt keine Skalierung und keine Persistenz
- Version 1.0 (2026-09-27): wirkt wie ein `AX_PERMIT`, um das gesamte An/Aus der Befüllautomatik zu gaten

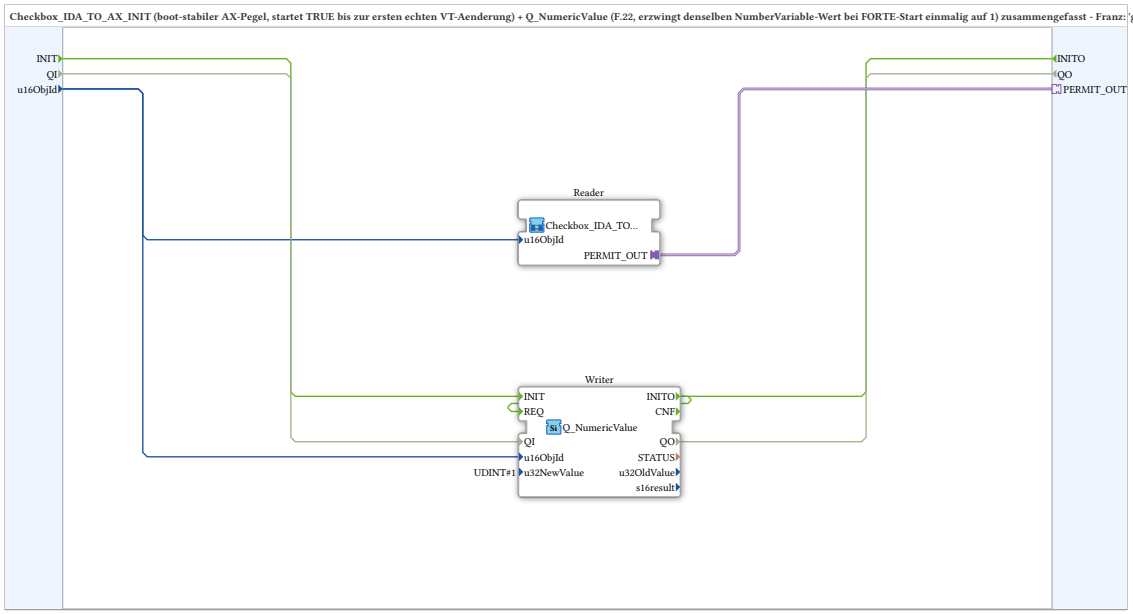
Anwendungsszenarien

- Freigabe (`PERMIT`) einer Automatik über eine VT-Checkbox, z. B. `Automatik_Toggle.PERMIT`

Zusammenfassung

`Checkbox_IDA_TO_AX_INIT` liest eine VT-Checkbox über ihre NumberVariable und stellt den Zustand als `AX`-Pegel mit definiertem Initialwert bereit.

Checkbox_IDA_TO_AX_INIT_Force1



• – ○ • – ○ • – ○ •

Einleitung

`Checkbox_IDA_TO_AX_INIT_Force1` fasst `Checkbox_IDA_TO_AX_INIT` (boot-stabiler AX-Pegel, startet `TRUE` bis zur ersten echten VT-Änderung) und `Q_NumericValue` (erzwingt beim FORTE-Start einmalig den Wert 1 der zugehörigen NumberVariable) zusammen. Beide Bausteine lesen bzw. schreiben dieselbe `u16ObjId`, daher gehören sie zusammen; die `u16ObjId` ist als Parameter wiederverwendbar. Die SubApp wurde aus `Aufnahme_Automatik.SUB` herausgelöst (`Permit_Automatik` plus `Q_NumericValue_Force1_Geschwindigkeitsregelung`).

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung – an `Writer.INIT`

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung – von `Writer.INITO`

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input – TRUE, um den Dienst freizugeben (an `Writer.QI`)
- **u16ObjId** (`UINT`, Vorgabe `ID_NULL`): Object-ID der VT-Checkbox-NumberVariable (dasselbe Objekt für Lesen und Force1-Schreiben)

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde (von `Writer.QO`)

Adapter

- **PERMIT_OUT** (Plug, `adapter::types::unidirectional::AX`): Boot-stabiler Checkbox-Zustand (startet `TRUE`, folgt danach der echten VT-Checkbox)

Verwendete Funktionsbausteine (FBs)

- **Reader**: SubApp `MyLib_AX::sys::Checkbox_IDA_TO_AX_INIT` – liefert den boot-stabilen Checkbox-Zustand als AX

- **Writer**: `isobus::UT::Q::Q_NumericValue` mit `u32NewValue=1` – fester FB-seitiger Einmalschreiber (nicht die `InitialValue`-Property des VT-Pools)

Programmablauf und Verbindungen

- `u16ObjId` ⟶ `Reader.u16ObjId` und `Writer.u16ObjId`
- `Writer.INITO` ⟶ `Writer.REQ`: Beim Start wird einmalig der Wert 1 geschrieben.
- `INIT` ⟶ `Writer.INIT`; `Writer.INITO` ⟶ `INITO`
- `QI` ⟶ `Writer.QI`; `Writer.QO` ⟶ `QO`
- `Reader.PERMIT_OUT` ⟶ `PERMIT_OUT`

Technische Besonderheiten

- Der Writer löst durch die Rückführung `INITO` ⟶ `REQ` genau einmal beim Initialisieren aus; danach bleibt die Checkbox unter der Kontrolle des Anwenders.

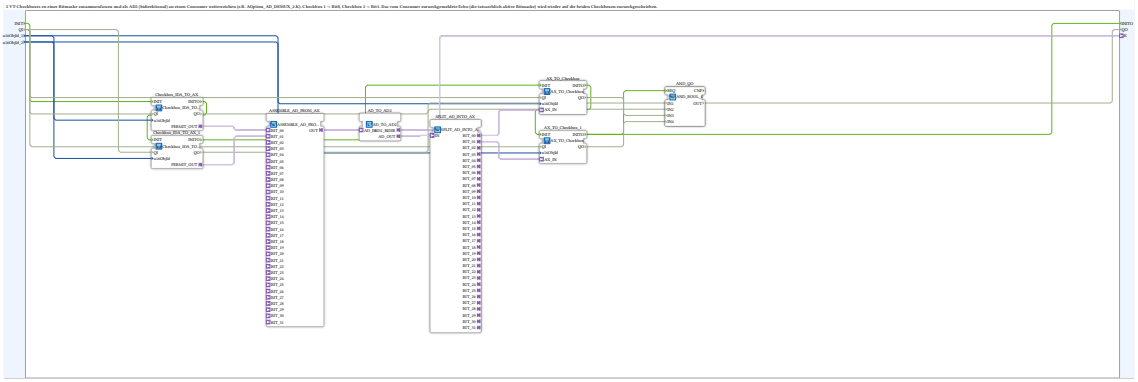
Anwendungsszenarien

- Freigabe-Checkbox (z. B. für eine Automatik), die beim Start aktiv sein soll und deren NumberVariable beim FORTE-Start konsistent auf 1 gesetzt wird

Fazit

`Checkbox_IDA_TO_AX_INIT_Force1` stellt eine beim Start aktive, boot-stabile Checkbox-Freigabe als AX bereit und setzt das zugehörige VT-Objekt konsistent auf 1.

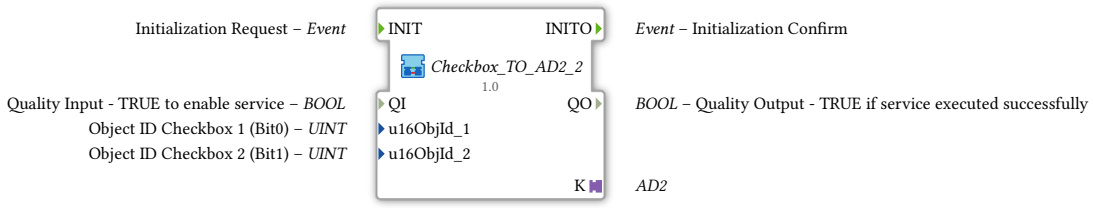
Checkbox_TO_AD2_2



• – ○ • – ○ • – ○ •

Einleitung

`Checkbox_TO_AD2_2` fasst zwei VT-Checkboxen zu einer Bitmaske zusammen und reicht sie bidirektional als AD2-Adapter (`K`) an einen Consumer weiter (z. B. `AOption_AD_DEMUX_2.K`). Checkbox 1 entspricht Bit 0, Checkbox 2 Bit 1. Die vom Consumer zurückgemeldete (tatsächlich aktive) Bitmaske wird wieder auf die beiden Checkboxen zurückgeschrieben.



Schnittstelle

Ereignis-Eingänge

- `INIT` (`EInit`): Initialization Request.

Ereignis-Ausgänge

- `INITO` (`EInit`): Initialization Confirm.

Eingangsvariablen

- `QI` (BOOL, Default `TRUE`): Quality Input, TRUE aktiviert den Dienst.
- `u16ObjId_1` (UINT, Default `ID_NULL`): ObjectID Checkbox 1 (Bit 0).
- `u16ObjId_2` (UINT, Default `ID_NULL`): ObjectID Checkbox 2 (Bit 1).

Ausgangsvariablen

- `QO` (BOOL): Quality Output, UND-Verknüpfung (`AND_QO`, `AND_BOOL_4`) der `QO`-Ausgänge der beiden `Checkbox_IDA_TO_AX`- und der beiden `AX_TO_Checkbox`-Instanzen.

Adapter

- **Plug `K`** (`adapter::types::bidirectional::AD2`): Bitmaske zum Consumer und Echo zurück.

Die Initialisierung läuft als Kette durch die vier Unterbausteine (siehe unten).

Funktionsweise

Hinrichtung (VT → Consumer):

1. Zwei Instanzen von `Checkbox_IDA_TO_AX` lesen je eine Checkbox und liefern einen AX-Pegel (`PERMIT_OUT`).
2. `ASSEMBLE_AD_FROM_AX` setzt die Pegel zu einem DWORD zusammen (`BIT_00` und `BIT_01`).
3. `AD_TO_AD2` wandelt in den bidirektionalen AD2-Adapter, der an `K` ausgegeben wird.

Rückrichtung (Consumer → VT):

1. Das über `K` zurückgemeldete Echo erscheint an `AD_TO_AD2.AD_OUT`.
2. `SPLIT_AD_INTO_AX` zerlegt das DWORD in `BIT_00` und `BIT_01`.
3. Zwei Instanzen von `AX_TO_Checkbox` schreiben die Bits auf die jeweilige Checkbox.

Initialisierung und Qualität:

1. `INIT` → `Checkbox_IDA_TO_AX.INIT`; `Checkbox_IDA_TO_AX.INITO` → `Checkbox_IDA_TO_AX_1.INIT`; `Checkbox_IDA_TO_AX_1.INITO` → `AX_TO_Checkbox.INIT`; `AX_TO_Checkbox.INITO` → `AX_TO_Checkbox_1.INIT`; `AX_TO_Checkbox_1.INITO` → `INITO` und → `AND_QO.REQ`.
2. `QI` → `.QI` aller vier Unterbausteine; deren `QO` → `AND_QO.IN1` bis `AND_QO.IN4` (`Checkbox_IDA_TO_AX`, `Checkbox_IDA_TO_AX_1`, `AX_TO_Checkbox`, `AX_TO_Checkbox_1`); `AND_QO.OUT` → `QO`.

`u16ObjId_1` und `u16ObjId_2` gehen jeweils sowohl an die `Checkbox_IDA_TO_AX`- als auch an die `AX_TO_Checkbox`-Instanz der betreffenden Checkbox.

Technische Besonderheiten

- Ausgelagert aus `Uebung_094b5_AX` zur Wiederverwendung; ausgelegt für N=2 Bits/Checkboxen, analog zu `AOption_AD_DEMUX_2`.
- Speichert nicht selbst; die Persistierung obliegt dem Consumer.

Zusammenfassung

Bündelt zwei VT-Checkboxen als Bitmaske auf einen bidirektionalen AD2-Plug und spiegelt das Consumer-Echo auf die Checkboxen zurück.

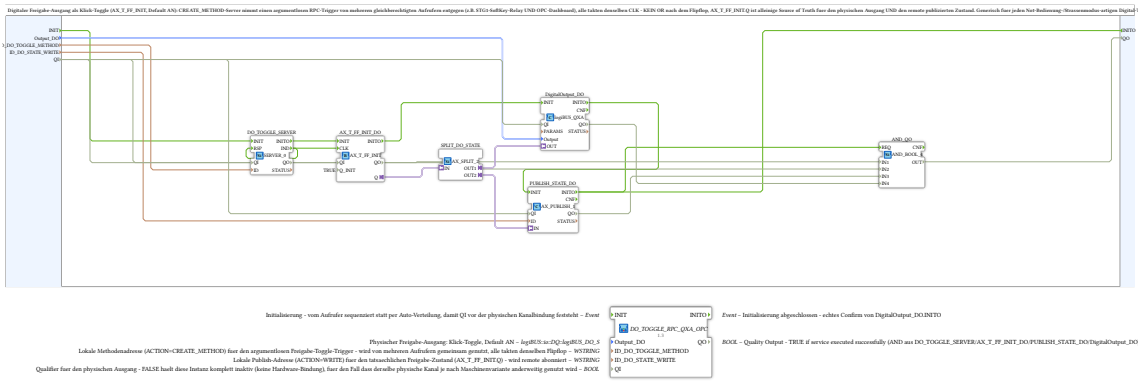
🛠️ Zugehörige Übungen

- Uebung_094b5b_AX

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

DO_TOGGLE_RPC_QXA_OPC



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **DO_TOGGLE_RPC_QXA_OPC** realisiert einen digitalen Freigabe-Ausgang als Klick-Toggle. Er wird über einen RPC-Mechanismus (Open-S62541/CREATE_METHOD) von mehreren gleichberechtigten Aufrufern (z. B. SoftKey-Relais oder OPC-Dashboard) gesteuert. Der Ausgangszustand wird sowohl als physisches Signal (logiBUS-QXA) als auch als remote publizierter Wert (OPC-UA-Write) bereitgestellt. Die Besonderheit liegt darin, dass alle Aufrufer denselben Takt (CLK) auf ein Flipflop geben – ohne nachgelagerte ODER-Verknüpfung. Der Flipflop-Ausgang ist die alleinige „Source of Truth“ für den physischen Ausgang und den remote publizierten Zustand.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierung; vom Aufrufer sequenziert statt per Auto-Verteilung, damit `QI` vor der physischen Kanalbindung feststeht; startet die Initialisierungskette `DO_TOGGLE_SERVER` ⟶ `AX_T_FF_INIT_DO` ⟶ `DigitalOutput_DO` ⟶ `PUBLISH_STATE_DO`

Die auslösenden Toggle-Ereignisse werden weiterhin intern über den RPC-Server generiert.

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierung abgeschlossen; echtes Confirm von `PUBLISH_STATE_DO.INITO` (Ende der Initialisierungskette)

Daten-Eingänge

Name	Typ	Beschreibung
`QI`	`BOOL`	Qualifier für den physischen Ausgang; FALSE hält die Instanz komplett inaktiv (keine Hardware-Bindung), Default `TRUE`; wird an `DO_TOGGLE_SERVER.QI`, `AX_T_FF_INIT_DO.QI`, `PUBLISH_STATE_DO.QI` und `DigitalOutput_DO.QI` durchgereicht.
`Output_DO`	`logiBUS::io::DQ::logiBUS_DO_S`	Physischer Freigabe-Ausgang; steuert den digitalen Output (Klick-Toggle, Default AN).
`ID_DO_TOGGLE_METHOD`	`WSTRING`	Lokale Methodenadresse (ACTION=CREATE_METHOD) für den argumentlosen RPC-Trigger; wird von mehreren Aufrufern gemeinsam genutzt.

ID_DO_STATE_WRITE	WSTRING	Lokale Publish-Adresse (ACTION=WRITE) für den aktuellen Freigabe-Zustand (Flipflop-Q); wird remote abonniert.

Daten-Ausgänge

Name	Typ	Beschreibung
QO	BOOL	Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde; UND (AND_QO, AND_BOOL_4) der QO von DO_TOGGLE_SERVER, AX_T_FF_INIT_DO, PUBLISH_STATE_DO und DigitalOutput_DO.

Adapter

Keine externen Adapter. Intern werden Adapter (AX_T_FF_INIT, AX_SPLIT_2, AX_PUBLISH_1) verwendet, um den Zustand zu verarbeiten und zu verteilen.

Funktionsweise

Der Baustein besteht intern aus folgenden Funktionsblöcken:

- **DO_TOGGLE_SERVER** (iec61499::net::SERVER_0): Ein OPC-UA-Server, der eingehende RPC-Aufrufe auf der konfigurierten Methodenadresse (ID_DO_TOGGLE_METHOD) empfängt. Bei jedem Trigger wird das Ereignis IND ausgelöst.
- **AX_T_FF_INIT_DO** (adapter::events::unidirectional::AX_T_FF_INIT): Ein Flipflop mit initialem Ausgang TRUE (AN). Jedes CLK-Ereignis toggelt den Zustand.
- **SPLIT_DO_STATE** (adapter::events::unidirectional::AX_SPLIT_2): Verteilt den Zustand (Q) auf zwei parallele Ausgänge (OUT1 und OUT2).
- **DigitalOutput_DO** (logiBUS::io::DQ::logiBUS_QXA): Schaltet den physischen Ausgang entsprechend dem Flipflop-Zustand; QI kommt von der Schnittstelle der SubApp; INIT kommt von AX_T_FF_INIT_DO.INITO, sein INITO startet PUBLISH_STATE_DO.INIT.
- **PUBLISH_STATE_DO** (adapter::net::AX_PUBLISH_1): Veröffentlicht den Zustand als OPC-UA-Write auf der Adresse ID_DO_STATE_WRITE; INIT kommt von DigitalOutput_DO.INITO, sein INITO wird als INITO ausgegeben (letztes Glied der Initialisierungskette).
- **AND_QO** (iec61131::booleanOperators::AND_BOOL_4): Verknüpft die QO-Ausgänge von DO_TOGGLE_SERVER, AX_T_FF_INIT_DO, PUBLISH_STATE_DO und DigitalOutput_DO per UND zum Baustein-Ausgang QO; ausgelöst durch PUBLISH_STATE_DO.INITO (REQ).

Initialisierung: INIT → DO_TOGGLE_SERVER.INIT, DO_TOGGLE_SERVER.INITO → AX_T_FF_INIT_DO.INIT, AX_T_FF_INIT_DO.INITO → DigitalOutput_DO.INIT, DigitalOutput_DO.INITO → PUBLISH_STATE_DO.INIT, PUBLISH_STATE_DO.INITO → INITO und → AND_QO.REQ.

Ablauf: Ein RPC-Trigger (z. B. vom OPC-Dashboard oder SoftKey) ruft die Methode auf. Der Server erzeugt ein IND-Ereignis, das direkt den Flipflop (CLK) taktet. Gleichzeitig wird RSP sofort zurückgesendet, um den Server-Thread nicht zu blockieren (siehe Technische Besonderheiten). Der Flipflop-Ausgang Q wechselt seinen Zustand (toggle). Über den Splitter wird dieser Zustand gleichzeitig an den physischen Ausgang und an den Publisher übergeben. Der Publisher aktualisiert den remote überwachten Wert.

Technische Besonderheiten

- **Rückverdrahtung RSP → IND:** Der Server muss RSP unmittelbar auf IND zurückführen. Ohne diese direkte Verbindung blockiert FORTE den OPC-UA-Server-Thread bis zu 4 Sekunden (Methode-Timeout) und hält dabei den globalen open62541-Mutex, was auch andere OPC-UA-Kommunikation (z. B. Publish) behindern kann. Die direkte RSP-Verdrahtung ist zwingend erforderlich.
- **Keine ODER-Verknüpfung nach dem Flipflop:** Anders als ähnliche Bausteine (z. B. TOGGLE_RPC_MERGE_QXA_OPC) wird hier kein OR nach dem Flipflop verwendet. Da alle Aufrufer denselben CLK-Eingang des Flipflops speisen, ist keine Zusammenführung auf Datenebene notwendig. Dadurch bleibt die Zustandslogik eindeutig und einfach.
- **Eine einzige „Source of Truth“:** Der Ausgang von AX_T_FF_INIT_DO.Q bestimmt sowohl den physischen als auch den remote publizierten Zustand. Es gibt keine Abweichungen oder Doppelquellen.

Zustandsübersicht

Zustand	Beschreibung
`TRUE` (AN)	Freigabe aktiv (Default). Physischer Ausgang und OPC-Wert sind `TRUE`.
`FALSE` (AUS)	Freigabe inaktiv. Physischer Ausgang und OPC-Wert sind `FALSE`.

Jeder RPC-Trigger wechselt zwischen diesen Zuständen. Der Initialzustand ist `TRUE` (AN).

Anwendungsszenarien

- **Not-Bedienung:** Ein manueller Schalter (SoftKey) und ein OPC-Dashboard können abwechselnd oder gleichzeitig denselben digitalen Ausgang (z. B. Freigabe für eine Maschine) per Toggle aktivieren/deaktivieren.
- **Straßenmodus / Verkehrssteuerung:** Umschalten zwischen „Freigabe“ und „Sperrung“ über mehrere Bedienstellen.
- **Generische digitale Toggle-Ausgänge** mit Mehrfach-Bedienung, bei denen eine konsistente Zustandsverwaltung ohne zusätzliche Logik gefordert ist.

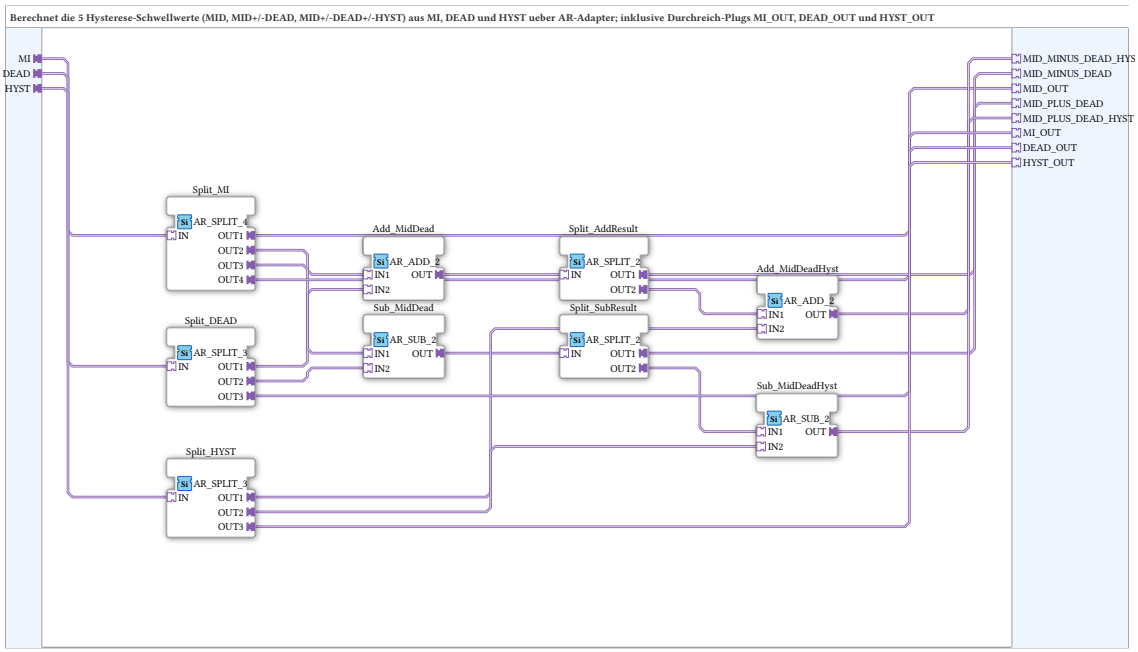
Vergleich mit ähnlichen Bausteinen

Der Baustein `DO_TOGGLE_RPC_QXA_OPC` unterscheidet sich von Bausteinen wie `TOGGLE_RPC_MERGE_QXA_OPC` (z. B. für Blitzersteuerung) dadurch, dass er **kein ODER** nach dem Flipflop verwendet. In einem Szenario, in dem mehrere Aufrufer potenziell unterschiedliche Zustände anlegen könnten, wäre ein OR notwendig. Hier jedoch wird nichts gemerged – alle Aufrufer takten denselben Flipflop. Dies vereinfacht die Logik, reduziert Latenz und vermeidet Konsistenzprobleme, da es keine parallelen Pfade gibt.

Fazit

Der Baustein **DO_TOGGLE_RPC_QXA_OPC** ist eine saubere, effiziente Lösung für digitale Freigabe-Ausgänge, die über RPC von mehreren Stellen getoggelt werden. Durch die Verwendung eines einzigen Flipflops als Quelle wird sichergestellt, dass der physische und der kommunizierte Zustand immer übereinstimmen. Die direkte RSP-Rückkopplung verhindert Blockaden im OPC-UA-Server und gewährleistet eine schnelle Reaktionszeit. Die einfache Struktur erleichtert Wartung und Wiederverwendung in verschiedenen Anwendungen.

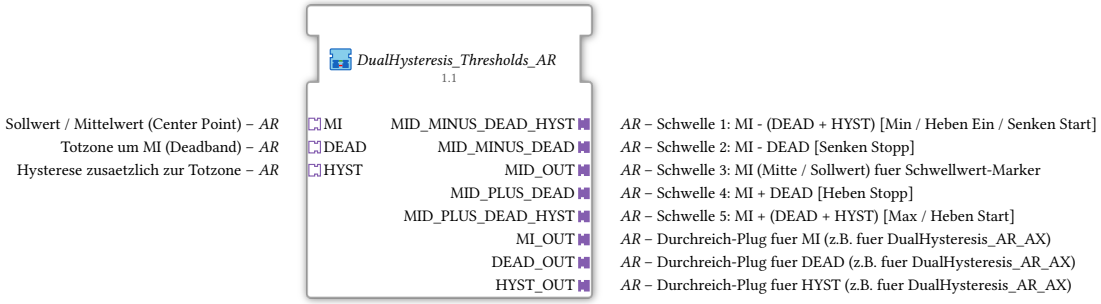
DualHysteresis_Thresholds_AR



• – ○ • – ○ • – ○ •

Einleitung

`DualHysteresis_Thresholds_AR` berechnet aus dem Sollwert `MI` (Mittelwert), der Totzone `DEAD` und der Hysterese `HYST` die fünf Schwellwerte einer Zwei-Wege-Hysterese (z. B. Heben/Senken) ausschließlich über AR-Adapter. Zusätzlich werden `MI`, `DEAD` und `HYST` über die Durchreich-Plugs `MI_OUT`, `DEAD_OUT` und `HYST_OUT` weitergegeben.



Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **MI** (Socket, `AR`): Sollwert / Mittelwert (Center Point)
- **DEAD** (Socket, `AR`): Totzone um `MI` (Deadband)
- **HYST** (Socket, `AR`): Hysterese zusätzlich zur Totzone
- **MID_MINUS_DEAD_HYST** (Plug, `AR`): Schwelle 1 = `MI - (DEAD + HYST)` [Min / Heben Ein / Senken Start]
- **MID_MINUS_DEAD** (Plug, `AR`): Schwelle 2 = `MI - DEAD` [Senken Stopp]
- **MID_OUT** (Plug, `AR`): Schwelle 3 = `MI` (Mitte / Sollwert), z. B. für einen Schwellwert-Marker
- **MID_PLUS_DEAD** (Plug, `AR`): Schwelle 4 = `MI + DEAD` [Heben Stopp]
- **MID_PLUS_DEAD_HYST** (Plug, `AR`): Schwelle 5 = `MI + (DEAD + HYST)` [Max / Heben Start]
- **MI_OUT**, **DEAD_OUT**, **HYST_OUT** (Plugs, `AR`): Durchreich-Plugs für `MI`, `DEAD` und `HYST` (z. B. für `DualHysteresis_AR_AX`)

Funktionsweise

Jeder Eingang wird zuerst durch einen Split-Baustein auf mehrere Verbraucher verteilt, weil ein Adapter nie ohne Split verzweigt werden soll:

- `MI` ⟶ `Split_MI` (`AR_SPLIT_4`): `OUT1` ⟶ `MID_OUT`, `OUT2` ⟶ `Add_MidDead.IN1`, `OUT3` ⟶ `Sub_MidDead.IN1`, `OUT4` ⟶ `MI_OUT`.
- `DEAD` ⟶ `Split_DEAD` (`AR_SPLIT_3`): `OUT1` ⟶ `Add_MidDead.IN2`, `OUT2` ⟶ `Sub_MidDead.IN2`, `OUT3` ⟶ `DEAD_OUT`.
- `HYST` ⟶ `Split_HYST` (`AR_SPLIT_3`): `OUT1` ⟶ `Add_MidDeadHyst.IN2`, `OUT2` ⟶ `Sub_MidDeadHyst.IN2`, `OUT3` ⟶ `HYST_OUT`.

Die Rechenkette besteht aus `AR_ADD_2` und `AR_SUB_2` (`adapter::iec61131::arithmetic`):

- `Add_MidDead = MI + DEAD` ⟶ `Split_AddResult` (`AR_SPLIT_2`): `OUT1` ⟶ `MID_PLUS_DEAD`, `OUT2` ⟶ `Add_MidDeadHyst.IN1`.
- `Sub_MidDead = MI - DEAD` ⟶ `Split_SubResult` (`AR_SPLIT_2`): `OUT1` ⟶ `MID_MINUS_DEAD`, `OUT2` ⟶ `Sub_MidDeadHyst.IN1`.

- `Add_MidDeadHyst = (MI + DEAD) + HYST` ⟶ `MID_PLUS_DEAD_HYST`.
- `Sub_MidDeadHyst = (MI - DEAD) - HYST` ⟶ `MID_MINUS_DEAD_HYST`.

Technische Besonderheiten

- Reiner Rechen-/Verdrahtungsbaustein ohne eigenen Zustand; die Ausgänge folgen den Eingängen
- `MI_OUT` ist vom Schwellwert `MID_OUT` getrennt (Version 1.1: `Split_MI` von `AR_SPLIT_3` auf `AR_SPLIT_4` erweitert), damit jeder Plug eine eigene Quelle hat
- Die Beschriftungen in eckigen Klammern (Heben/Senken) sind die vorgesehene Verwendung der Schwellen in der Aufrufer-Logik

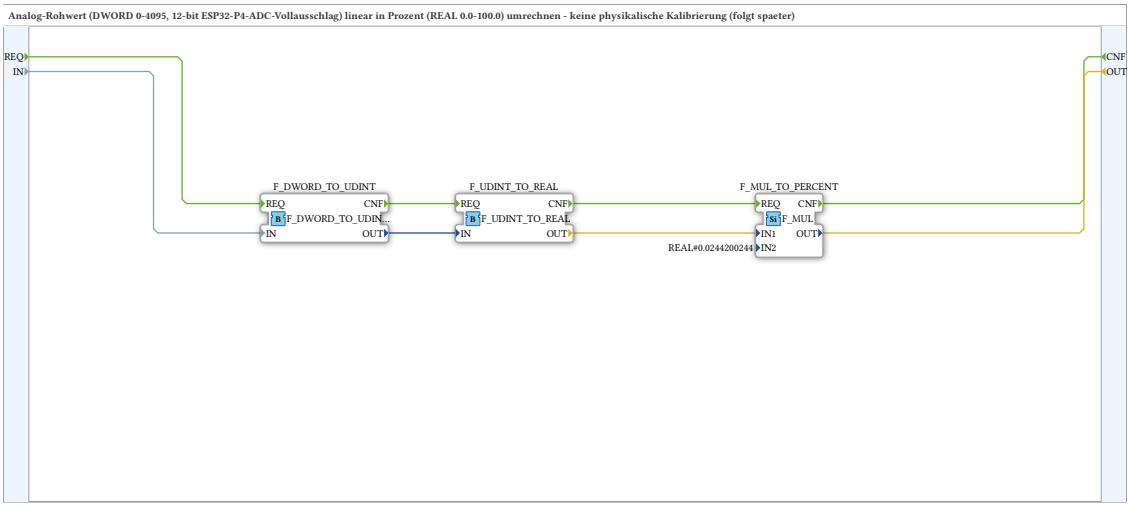
Anwendungsszenarien

- Zwei-Wege-Regelung (Heben/Senken) mit Totzone und Hysterese, z. B. Füllband- oder Hangausgleich-Höhenregelung, einschließlich Schwellwert-Markern am VT

Fazit

`DualHysteresis_Thresholds_AR` liefert aus `MI`, `DEAD` und `HYST` per AR-Adapter-Arithmetik die fünf Schwellwerte `MI ± DEAD` und `MI ± (DEAD + HYST)` sowie `MI` selbst und reicht die drei Eingangsgrößen unverändert weiter.

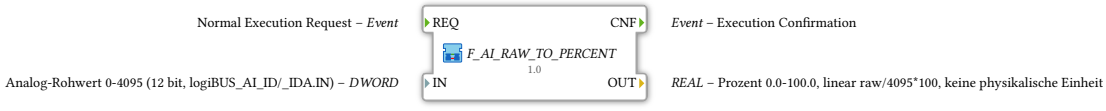
F_AI_RAW_TO_PERCENT



• — ○ • — ○ • — ○ •

Einleitung

`F_AI_RAW_TO_PERCENT` rechnet den Analog-Rohwert von `logiBUS_AI_ID/logiBUS_AI_IDA` (DWORD, 0-4095, 12-bit ESP32-P4-ADC-Vollausschlag) **linear** in Prozent (REAL 0.0-100.0) um — ohne physikalische Kalibrierung (die folgt als späterer Ausbauschritt). Datenbasierte Variante; die volladapterbasierte Alternative ist `F_AI_RAW_TO_PERCENT_AD`.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: F_AI_RAW_TO_PERCENT

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **F_DWORD_TO_UDINT**: `iec61131::conversion::F_DWORD_TO_UDINT` — Bit-Reinterpretation DWORD⟶UDINT, hier gültig (gleiche 32-Bit-Darstellung eines vorzeichenlosen Integers).
  - **F_UDINT_TO_REAL**: `iec61131::conversion::F_UDINT_TO_REAL` — echter numerischer Cast UDINT⟶REAL.

  - **F_MUL_TO_PERCENT**: `iec61131::arithmetic::F_MUL` — Multiplikation mit `REAL#0.0244200244` (= 100/4095).
- **Funktionsweise**: `IN` (Rohwert 0-4095) ⟶ UDINT ⟶ REAL ⟶ ×(100/4095) = Prozent.

Programmablauf und Verbindungen

1. `IN` ⟶ `F_DWORD_TO_UDINT.IN` ⟶ `F_UDINT_TO_REAL.IN` ⟶ `F_MUL_TO_PERCENT.IN1`.
2. `F_MUL_TO_PERCENT.IN2 = REAL#0.0244200244` (Parameter, = 100/4095).
3. `F_MUL_TO_PERCENT.OUT` ⟶ `OUT` (Prozent 0.0-100.0).

Technische Besonderheiten

- **Numerisch korrekte Kette**: DWORD⟶UDINT ist eine gültige Bit-Reinterpretation (siehe Numerisch vs. bitweise), UDINT⟶REAL ist ein echter numerischer Cast — zusammen ergibt das eine korrekte DWORD⟶REAL-Umwandlung, ohne die Bit-Reinterpretations-Falle von `F_DWORD_TO_REAL`.
- **Keine physikalische Kalibrierung**: Der Faktor 100/4095 liefert nur den linearen Prozentanteil des ADC-Vollausschlags, keine physikalische Einheit (Volt, Bar, etc.) — das ist laut Kommentar ein späterer Ausbauschritt.

Anwendungsszenarien

- Anzeige/Weiterverarbeitung eines rohen Analogeingangswerts als Prozentwert, z. B. für ein AI-Trainingsbeispiel.

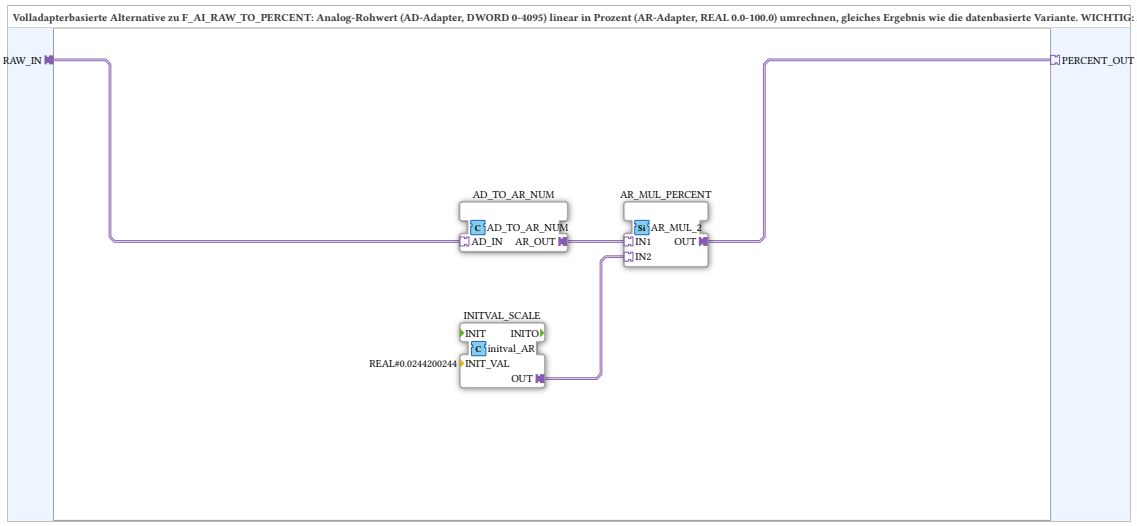
Zusammenfassung

`F_AI_RAW_TO_PERCENT` demonstriert die numerisch korrekte DWORD⟶REAL-Kette (über UDINT) am konkreten Beispiel eines Analog-Rohwerts — datenbasierte Variante, siehe `F_AI_RAW_TO_PERCENT_AD` für die adapterbasierte Alternative.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

F_AI_RAW_TO_PERCENT_AD



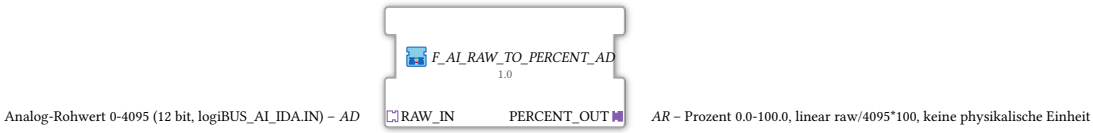
• – ○ • – ○ • – ○ •

Einleitung

`F_AI_RAW_TO_PERCENT_AD` ist die volladapterbasierte Alternative zu `F_AI_RAW_TO_PERCENT`: derselbe rohe Analog-Rohwert (0-4095) wird über AD-/AR-Adapter statt über normale Datenverbindungen linear in Prozent umgerechnet — identisches Ergebnis, andere Verdrahtungsart.

> ⚠️ **Wichtiger Hinweis aus dem Baustein-Kommentar:** `AD_TO_AR`/`F_DWORD_TO_REAL` wäre hier **falsch** — das ist eine Bit-Reinterpretation (IEEE754-Cast), keine numerische Umwandlung (siehe

`forte_real.cpp`, `CIEC_REAL::setValue`, `case e_DWORD: setValueSimple`). Der korrekte Weg ist die numerische Konvertierung `AD_TO_AR_NUM` (DWORD→UDINT per Bit-Reinterpretation, hier gültig, gleiche Bitbreite/Darstellung; danach `F_UDINT_TO_REAL`, echte numerische Umwandlung). Details siehe Numerisch vs. bitweise: Die Konvertierungs-Falle in FORTE.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: F_AI_RAW_TO_PERCENT_AD

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **AD_TO_AR_NUM**: `adapter::conversion::unidirectional::AD_TO_AR_NUM` — numerische Konvertierung DWORD-Adapter→REAL-Adapter über UDINT, siehe `AD_TO_AR_NUM`.
  - **INITVAL_SCALE**: `adapter::types::unidirectional::AR::initval::initval_AR` — liefert die Konstante `REAL#0.0244200244` (= 100/4095) als Adapter.
  - **AR_MUL_PERCENT**: `adapter::iec61131::arithmetic::AR_MUL_2` — Multiplikation zweier REAL-Adapter.
- **Funktionsweise**: `RAW_IN` (AD-Adapter, 0-4095) ⟶ `AD_TO_AR_NUM` ⟶ multipliziert mit der Konstante ⟶ `PERCENT_OUT` (AR-Adapter, 0.0-100.0).

Programmablauf und Verbindungen

1. `RAW_IN` ⟶ `AD_TO_AR_NUM.AD_IN`; `AD_TO_AR_NUM.AR_OUT` ⟶ `AR_MUL_PERCENT.IN1`.
2. `INITVAL_SCALE.OUT` (Konstante 0.0244200244) ⟶ `AR_MUL_PERCENT.IN2`.
3. `AR_MUL_PERCENT.OUT` ⟶ `PERCENT_OUT`.

Technische Besonderheiten

- **Bewusstes Negativ-Beispiel im Kommentar**: Der Baustein-Kommentar dokumentiert explizit, warum `AD_TO_AR` hier falsch wäre — eine seltene, aber wertvolle „Anti-Pattern“-Dokumentation direkt im Quellcode.
- **Adapterkonstante statt Parameter**: Die Skalierungskonstante wird über `initval_AR` als Adapter bereitgestellt statt als klassischer FB-Parameter — konsistent mit der volladapterbasierten Philosophie des Bausteins.

Anwendungsszenarien

- Wie `F_AI_RAW_TO_PERCENT`, aber für SubApp-Netzwerke, die durchgängig auf Adapterverbindungen statt klassischen Datenverbindungen aufgebaut sind.

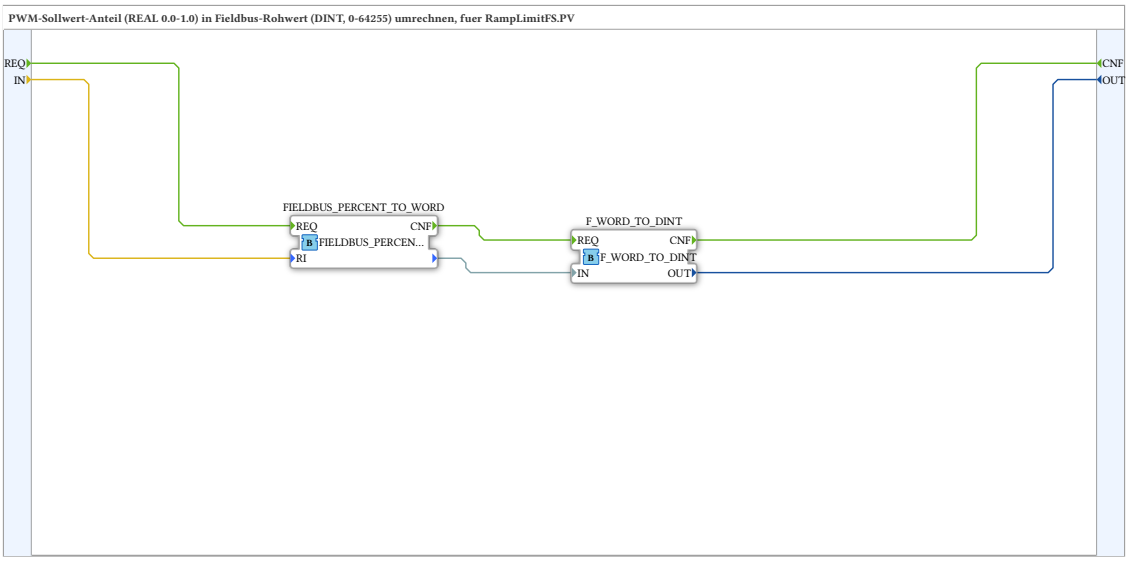
Zusammenfassung

`F_AI_RAW_TO_PERCENT_AD` liefert dieselbe numerisch korrekte DWORD⟶REAL-Prozentumrechnung wie `F_AI_RAW_TO_PERCENT`, vollständig adapterbasiert — und dokumentiert dabei explizit die Bit-Reinterpretations-Falle, die bei einer naiven `AD_TO_AR`-Verdrahtung entstünde.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

F_PWM_PERCENT_TO_RAW



• – ○ • – ○ • – ○ •

Einleitung

`F_PWM_PERCENT_TO_RAW` rechnet einen **PWM-Sollwert-Anteil (REAL 0.0–1.0)** in den **Fieldbus-Rohwert (DINT, 0–64255)** um, den `RampLimitFS.PV` erwartet. Trotz des Namens erwartet der Baustein keinen Prozentwert (0–100), sondern einen Anteil (0.0–1.0) — die Umrechnung von Prozent auf Anteil übernimmt vorgeschaltet `logiBUS::signalprocessing::fieldbus::F_PERCENT_TO_FRACTION`.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: F_PWM_PERCENT_TO_RAW

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **FIELDBUS_PERCENT_TO_WORD**: `eclipse4diac::signalprocessing::FIELDBUS_PERCENT_TO_WORD`
    - Dateneingang: `RI` (Anteil 0.0-1.0)
    - Ereignisausgang: `CNF`
  - **F_WORD_TO_DINT**: `iec61131::conversion::F_WORD_TO_DINT`
    - Dateneingang: `IN`, Datenausgang: `OUT`
- **Funktionsweise**: Der Standard-4diac-Baustein `FIELDBUS_PERCENT_TO_WORD` liefert den Fieldbus-Rohwert bereits als `WORD` (16-Bit); `F_WORD_TO_DINT` konvertiert anschließend auf `DINT`, da `RampLimitFS.PV` diesen Datentyp erwartet.

Programmablauf und Verbindungen

1. `REQ` (SubApp-Ereigniseingang) ⟶ `FIELDBUS_PERCENT_TO_WORD.REQ`
2. `IN` (Anteil 0.0-1.0) ⟶ `FIELDBUS_PERCENT_TO_WORD.RI`
3. `FIELDBUS_PERCENT_TO_WORD.CNF` ⟶ `F_WORD_TO_DINT.REQ`, Datenausgang ⟶ `F_WORD_TO_DINT.IN`
4. `F_WORD_TO_DINT.OUT` ⟶ `OUT` (Fieldbus-Rohwert 0-64255)
5. `F_WORD_TO_DINT.CNF` ⟶ `CNF` (SubApp-Ereignisausgang)

Technische Besonderheiten

- Verwendet den Standard-4diac-Baustein `FIELDBUS_PERCENT_TO_WORD` (`eclipse4diac::signalprocessing`) statt einer eigenen Umrechnungsformel — konsistent mit der SAE-J1939/ISO-11783-Konvention `VALID_SIGNAL_W`.
- Erwartet **Anteil 0.0-1.0**, nicht Prozent 0-100 — leicht zu verwechseln mit dem baugleich benannten `F_PERCENT_TO_FRACTION` (Prozent ⟶ Anteil), das vorgeschaltet werden muss.

Anwendungsszenarien

- Jede Übung, die einen analogen 0-100%-Sollwert auf einen `RampLimitFS`- oder anderen Fieldbus-Rohwert-Eingang (0-64255) abbilden muss.

Zusammenfassung

`F_PWM_PERCENT_TO_RAW` ist ein dünner Adapter um den Standard-Baustein `FIELDBUS_PERCENT_TO_WORD`, der den Anteil-zu-Fieldbus-Rohwert-Übergang auf den von `RampLimitFS.PV` erwarteten `DINT`-Typ bringt.

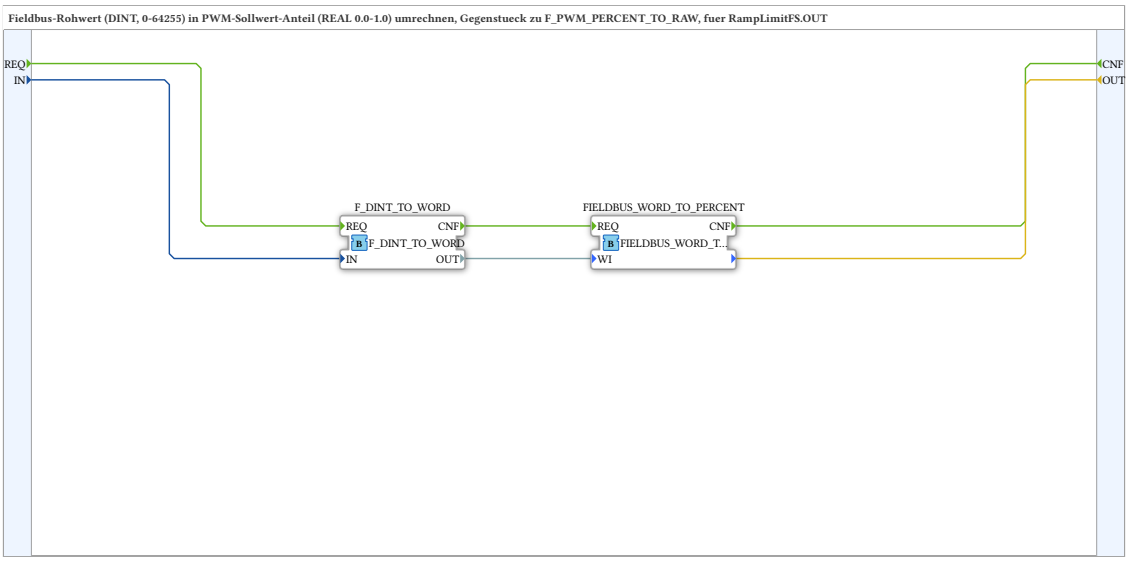
🛠️ Zugehörige Übungen

- RampLimitFS_TO_logiBUS_QDA_PWM_OPC
- F_PWM_RAW_TO_PERCENT (Gegenstück)
- InputOutputTesterButton_PWM_OPC_UA

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

F_PWM_RAW_TO_PERCENT



• – ◦ • – ◦ • – ◦ •

Einleitung

`F_PWM_RAW_TO_PERCENT` ist das Gegenstück zu `F_PWM_PERCENT_TO_RAW`: Es rechnet den **Fieldbus-Rohwert (DINT, 0–64255)** von `RampLimitFS.OUT` zurück in einen **Anteil (REAL 0.0–1.0)** um. Trotz des Namens liefert der Baustein keinen Prozentwert — die Umrechnung von Anteil auf Prozent übernimmt nachgeschaltet `logiBUS::signalprocessing::fieldbus::F_FRACTION_TO_PERCENT`.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: F_PWM_RAW_TO_PERCENT

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **F_DINT_TO_WORD**: `iec61131::conversion::F_DINT_TO_WORD`
    - Dateneingang: `IN`, Datenausgang: `OUT`
  - **FIELDBUS_WORD_TO_PERCENT**: `eclipse4diac::signalprocessing::FIELDBUS_WORD_TO_PERCENT`
    - Dateneingang: `WI` (Fieldbus-Rohwert als `WORD`)

  - Ereignisausgang: `CNF`
- **Funktionsweise**: `RampLimitFS.OUT` liefert den Rohwert als `DINT`; `F_DINT_TO_WORD` konvertiert zurück auf `WORD`, da der Standard-4diac-Baustein `FIELDBUS_WORD_TO_PERCENT` diesen Typ erwartet und daraus den Anteil (0.0-1.0) berechnet.

Programmablauf und Verbindungen

1. `REQ` (SubApp-Ereigniseingang) ⟶ `F_DINT_TO_WORD.REQ`
2. `IN` (Fieldbus-Rohwert 0-64255, DINT) ⟶ `F_DINT_TO_WORD.IN`
3. `F_DINT_TO_WORD.CNF` ⟶ `FIELDBUS_WORD_TO_PERCENT.REQ`, Datenausgang ⟶ `FIELDBUS_WORD_TO_PERCENT.WI`
4. `FIELDBUS_WORD_TO_PERCENT` (Datenausgang) ⟶ `OUT` (Anteil 0.0-1.0)
5. `FIELDBUS_WORD_TO_PERCENT.CNF` ⟶ `CNF` (SubApp-Ereignisausgang)

Technische Besonderheiten

- Verwendet den Standard-4diac-Baustein `FIELDBUS_WORD_TO_PERCENT` (`eclipse4diac::signalprocessing`) — konsistent mit der SAE-J1939/ISO-11783-Konvention `VALID_SIGNAL_W`.
- Liefert **Anteil 0.0-1.0**, nicht Prozent — für eine Prozent-Anzeige muss der Aufrufer noch mit 100.0 multiplizieren bzw. `F_FRACTION_TO_PERCENT` nachschalten.

Anwendungsszenarien

- Jede Übung, die einen `RampLimitFS`- oder anderen Fieldbus-Rohwert (0-64255) auf einen analogen 0-100%-Sollwert zur Anzeige oder Weiterleitung (z. B. OPC-UA-Publish) zurückrechnen muss.

Zusammenfassung

`F_PWM_RAW_TO_PERCENT` ist ein dünner Adapter um den Standard-Baustein `FIELDBUS_WORD_TO_PERCENT`, der den Fieldbus-Rohwert-zu-Anteil-Übergang auf den von `RampLimitFS.OUT` gelieferten `DINT`-Typ bringt.

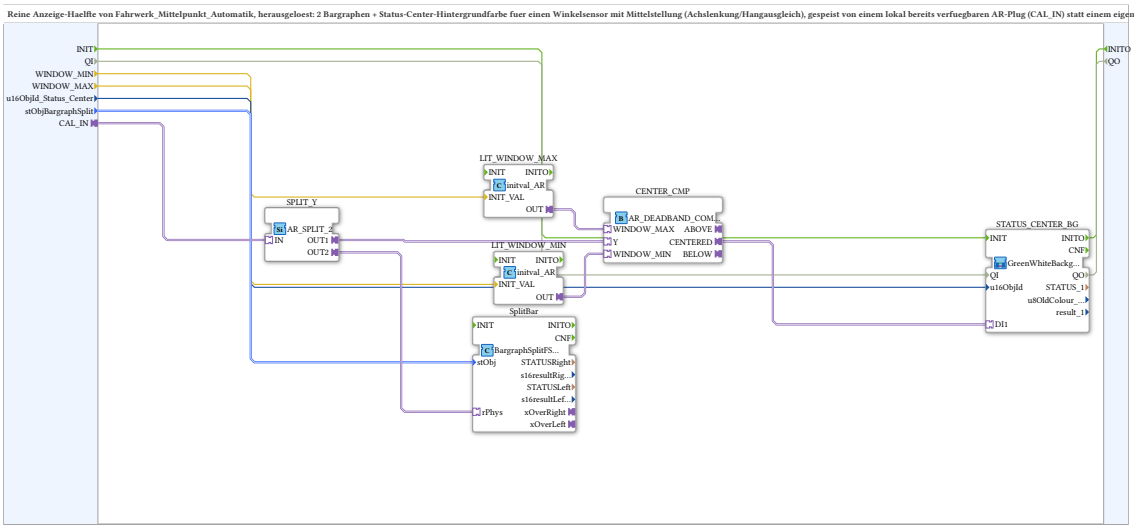
🛠️ Zugehörige Übungen

- RampLimitFS_TO_logiBUS_QDA_PWM_OPC
- F_PWM_PERCENT_TO_RAW (Gegenstück)
- InputOutputTesterButton_PWM_OPC_UA

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Fahrwerk_Mittelpunkt_Anzeige

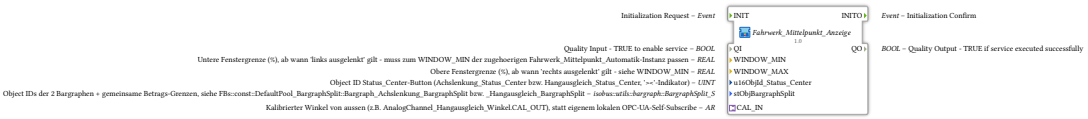


• – ○ • – ○ • – ○ •

Einleitung

`Fahrwerk_Mittelpunkt_Anzeige` ist die **reine Anzeige-Hälfte** von `Fahrwerk_Mittelpunkt_Automatik`: zwei Bargraphen plus Status-Center-Hintergrundfarbe für einen Winkelsensor mit Mittelstellung (Achslenkung/Hangausgleich). Gespeist wird sie von einem lokal bereits verfügbaren `AR`-Plug (`CAL_IN`) statt von einem eigenen OPC-UA-Self-Subscribe.

Sie ist für Fälle gedacht, in denen die eigentliche Automatik (`AUTO_LINKS`/`AUTO_RECHTS`-Regelung) auf einer anderen Station läuft als die VT (z. B. Hangausgleich seit dem Umzug nach STG4), dort aber trotzdem eine Anzeige gebraucht wird. Ein zusätzliches Forwarding ist nicht nötig: Der Kalibrierkanal-Konsument (`logiBUS_AI_Calibrated_Subscribe_VT_(3P_)OPC`) hat den Wert bereits lokal als `CAL_OUT`-Plug, der hier direkt weiterverdrahtet wird.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Fahrwerk_Mittelpunkt_Anzeige

- **Typ**: SubAppType
- **Verwendete interne Bausteine**:
  - **SPLIT_Y**: `adapter::events::unidirectional::AR_SPLIT_2` – verteilt den Winkel auf zwei Verbraucher
  - **CENTER_CMP**: `adapter::Engineering::measurements::AR_DEADBAND_COMPARE` – Fenstervergleich
  - **LIT_WINDOW_MIN**, **LIT_WINDOW_MAX**: `adapter::types::unidirectional::AR::initval::initval_AR` – wandeln die Parameter `WINDOW_MIN`/`WINDOW_MAX` in `AR`-Signale
  - **SplitBar**: `isobus::UT::Q::BargraphSplitFS_AR` – zwei Bargraphen (links/rechts der Mitte)
  - **STATUS_CENTER_BG**: SubApp `MyLib::sys::GreenWhiteBackground1_AX` – Hintergrundfarbe des `Status_Center`-Buttons

Schnittstellenstruktur

Ereignis-Eingänge / Ereignis-Ausgänge

- **INIT** (`EInit`, Eingang): Initialisierungsanforderung; wird an `STATUS_CENTER_BG.INIT` durchgereicht
- **INITO** (`EInit`, Ausgang): Initialisierungsbestätigung; kommt von `STATUS_CENTER_BG.INITO`

Die Laufzeit-Ereignisse laufen weiterhin über den Adapter.

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input, Freigabe des Dienstes; wird an `STATUS_CENTER_BG.QI` durchgereicht
- **WINDOW_MIN** (`REAL`, Vorgabe `-2.0`): Untere Fenstergrenze (%), ab wann „links ausgelenkt“ gilt – muss zum `WINDOW_MIN` der zugehörigen `Fahrwerk_Mittelpunkt_Automatik`-Instanz passen
- **WINDOW_MAX** (`REAL`, Vorgabe `2.0`): Obere Fenstergrenze (%), ab wann „rechts ausgelenkt“ gilt – siehe `WINDOW_MIN`
- **u16ObjId_Status_Center** (`UINT`): Object-ID des `Status_Center`-Buttons (`Achslenkung_Status_Center` bzw. `Hangausgleich_Status_Center`, „><“-Indikator)
- **stObjBargraphSplit** (`isobus::utils::bargraph::BargraphSplit_S`): Object-IDs der zwei Bargraphen plus gemeinsame Betrags-Grenzen (siehe `FBs::const::DefaultPool_BargraphSplit::Bargraph_Achslenkung_BargraphSplit` bzw. `_Hangausgleich_BargraphSplit`)

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output, kommt von `STATUS_CENTER_BG.QO`

Adapter

- **CAL_IN** (Socket, `adapter::types::unidirectional::AR`): Kalibrierter Winkel von außen (z. B. `AnalogChannel_Hangausgleich_Winkel.CAL_OUT`)

Programmablauf und Verbindungen

- `CAL_IN` ⟶ `SPLIT_Y.IN`
- `SPLIT_Y.OUT1` ⟶ `CENTER_CMP.Y`; `SPLIT_Y.OUT2` ⟶ `SplitBar.rPhys`
- `WINDOW_MIN` ⟶ `LIT_WINDOW_MIN.INIT_VAL`; `LIT_WINDOW_MIN.OUT` ⟶ `CENTER_CMP.WINDOW_MIN`
- `WINDOW_MAX` ⟶ `LIT_WINDOW_MAX.INIT_VAL`; `LIT_WINDOW_MAX.OUT` ⟶ `CENTER_CMP.WINDOW_MAX`
- `CENTER_CMP.CENTERED` ⟶ `STATUS_CENTER_BG.DI1`
- `u16ObjId_Status_Center` ⟶ `STATUS_CENTER_BG.u16ObjId`
- `stObjBargraphSplit` ⟶ `SplitBar.stObj`
- Initialisierung: `INIT` ⟶ `STATUS_CENTER_BG.INIT`; `STATUS_CENTER_BG.INITO` ⟶ `INITO`
- Qualität: `QI` ⟶ `STATUS_CENTER_BG.QI`; `STATUS_CENTER_BG.QO` ⟶ `QO`

Technische Besonderheiten

- Herausgelöst aus `Fahrwerk_Mittelpunkt_Automatik` (dort kamen `SUBSCRIBE_Y`, `SPLIT_Y`, `CENTER_CMP`, `SplitBar` und `STATUS_CENTER_BG` bereits genauso vor)
- Die Automatik selbst läuft seit v3.0 des Hangausgleichs auf STG4 (ohne VT; dort sind `u16ObjId_Status_Center` und `stObjBargraphSplit` bewusst `ID_NULL`); diese SubApp läuft stattdessen auf STG1, gespeist vom ohnehin lokal vorhandenen `CAL_OUT`-Plug – kein zusätzlicher Remote-Round-Trip
- Version 1.0 (2026-09-26)

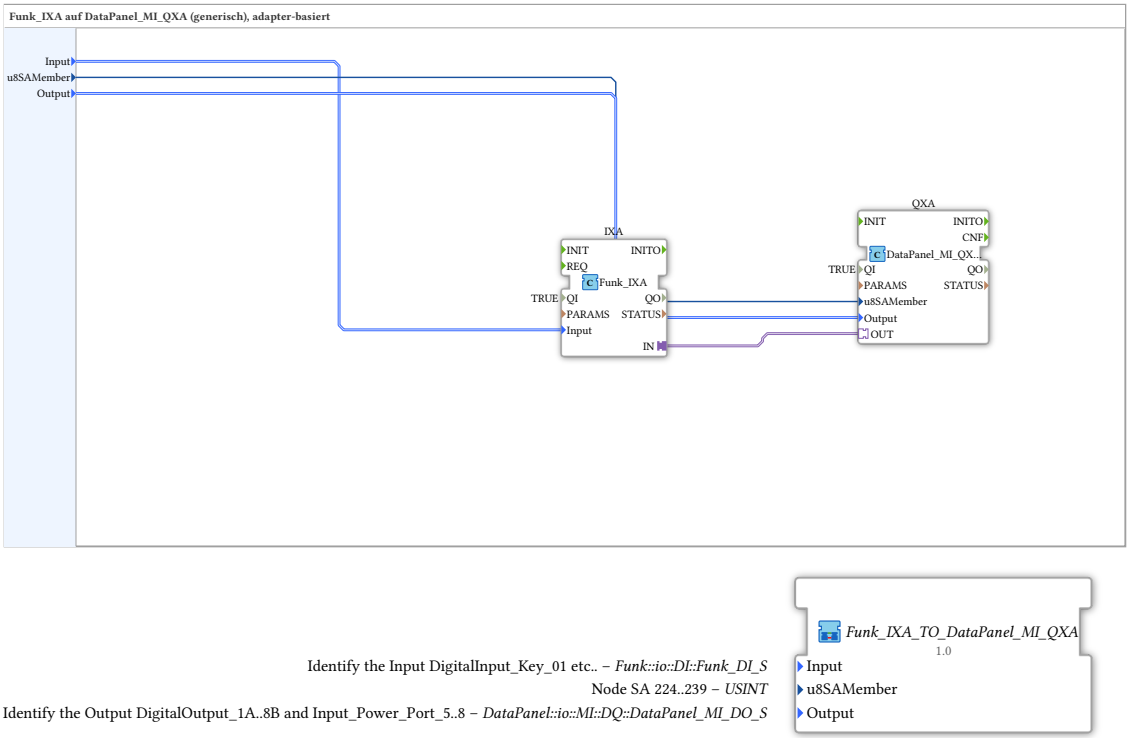
Anwendungsszenarien

- Anzeige von Achslenkung oder Hangausgleich (zwei Bargraphen plus Mitte-Indikator) auf einer Station, deren Regelung auf einer anderen Station läuft

Zusammenfassung

`Fahrwerk_Mittelpunkt_Anzeige` zeigt einen kalibrierten Winkel als zwei Bargraphen und als Status-Center-Hintergrundfarbe an, ohne eigene Regelung und ohne OPC-UA-Subscribe.

Funk_IXA_TO_DataPanel_MI_QXA



Identify the Input DigitalInput_Key_01 etc.. – *Funk::io::DI::Funk_DI_S*
Node SA 224..239 – *USINT*
Identify the Output DigitalOutput_1A..8B and Input_Power_Port_5..8 – *DataPanel::io::MI::DQ::DataPanel_MI_DO_S*

• – ○ • – ○ • – ○ •

Einleitung

Die SubApp `Funk_IXA_TO_DataPanel_MI_QXA` kapselt eine adapter-basierte Verbindung zwischen einem Funk-Eingangsmodul und einem DataPanel-Ausgangsmodul. Sie ermöglicht die nahtlose Übertragung

von empfangenen Funkdaten an ein Anzeige- oder Steuergerät, wobei die spezifischen Parameter für die Subnetzadresse und den Ausgangskanal extern konfigurierbar sind. Die SubApp ist generisch aufgebaut und kann in verschiedenen Kontexten wiederverwendet werden.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

- **Input** (Typ: `Funk::io::DI::Funk_DI_S`)
  Identifiziert den digitalen Eingang, z. B. `DigitalInput_Key_01` etc. Initialwert: `Funk_DI::Invalid`.
- **u8SAMember** (Typ: `USINT`)
  Node-SA (Subnetzadresse) im Bereich 224..239. Initialwert: `MI::MI_00`.
- **Output** (Typ: `DataPanel::io::MI::DQ::DataPanel_MI_DO_S`)
  Identifiziert den digitalen Ausgang, z. B. `DigitalOutput_1A..8B` und `Input_Power_Port_5..8`. Initialwert: `Invalid`.

Daten-Ausgänge

Keine.

Adapter

Keine externen Adapter vorhanden. Die Adapterverbindung wird intern zwischen den enthaltenen Funktionsblöcken genutzt.

Funktionsweise

Die SubApp enthält zwei Funktionsbausteine:

- **IXA** (Typ: `Funk::io::DI::Funk_IXA`) – verarbeitet den eingehenden Funkdatenstrom.
- **QXA** (Typ: `DataPanel::io::MI::DQ::DataPanel_MI_QXA`) – steuert die Ausgabe auf einem DataPanel.

Der Datenfluss erfolgt wie folgt:

1. Der externe Daten-Eingang `Input` wird an den Daten-Eingang `IXA.Input` weitergeleitet.
2. Der Funktionsblock `IXA` verarbeitet die Funkdaten und stellt das Ergebnis über seinen Adapter-Ausgang `IN` bereit.
3. Über eine interne Adapterverbindung werden diese Daten an den Adapter-Eingang `OUT` des Funktionsblocks `QXA` übertragen.
4. Die externen Eingänge `u8SAMember` und `Output` werden direkt an die gleichnamigen Daten-Eingänge des Blocks `QXA` angeschlossen, um die Zieladresse und den Ausgangskanal festzulegen.

Beide Funktionsblöcke sind mit `QI = TRUE` parametrisiert, sodass sie im aktivierten Zustand betrieben werden. Der Parameter `PARAMS` des Blocks `IXA` ist leer und nur intern sichtbar.

Technische Besonderheiten

- **Adapter-basierte Kopplung**: Die Kommunikation zwischen den beiden Funktionsblöcken erfolgt über eine standardisierte Adapterschnittstelle, was eine flexible und erweiterbare Verbindung ermöglicht.
- **Generische Parametrierung**: Die SubApp kann durch die externen Eingänge `u8SAMember` und `Output` für verschiedene Zieladressen und Ausgangskanäle konfiguriert werden, ohne Änderungen an der internen Struktur.
- **Wiederverwendbarkeit**: Die Kapselung der Verbindung in einer SubApp erlaubt eine einfache Integration in übergeordnete Systeme und fördert die Modularität.
- **Keine eigenen Zustände**: Als reine Datenfluss- und Adapterverbindung besitzt die SubApp keinen internen Zustandsautomaten.

Zustandsübersicht

Die SubApp selbst besitzt keine internen Zustände, da sie ausschließlich eine statische Verbindung zwischen den enthaltenen Funktionsblöcken herstellt. Das Zustandsverhalten wird vollständig durch die

zugrunde liegenden Funktionsblöcke (z. B. `Funk_IXA` und `DataPanel_MI_QXA`) definiert. Auf Ebene der SubApp besteht keine aktive Logik oder Zustandshaltung.

Anwendungsszenarien

- **Funkempfang und Anzeige**: Anbindung eines Funkempfängers an ein DataPanel, um empfangene Werte oder Statusmeldungen anzuzeigen.
- **Drahtlose Steuerung**: Verwendung in Systemen, in denen über Funk empfangene Signale direkt eine Ausgabe auf einem Panel steuern sollen.
- **Adressierbare Ausgänge**: Einsetzbar in Mehrpunkt-Konfigurationen, bei denen über die `u8SAMember`-Adresse unterschiedliche Panels oder Ausgänge selektiert werden.
- **Prototypen und Schulungen**: Als Vorlage für ähnliche Verbindungen in Lern- und Übungsumgebungen (z. B. im Rahmen von 4diac-Projekten).

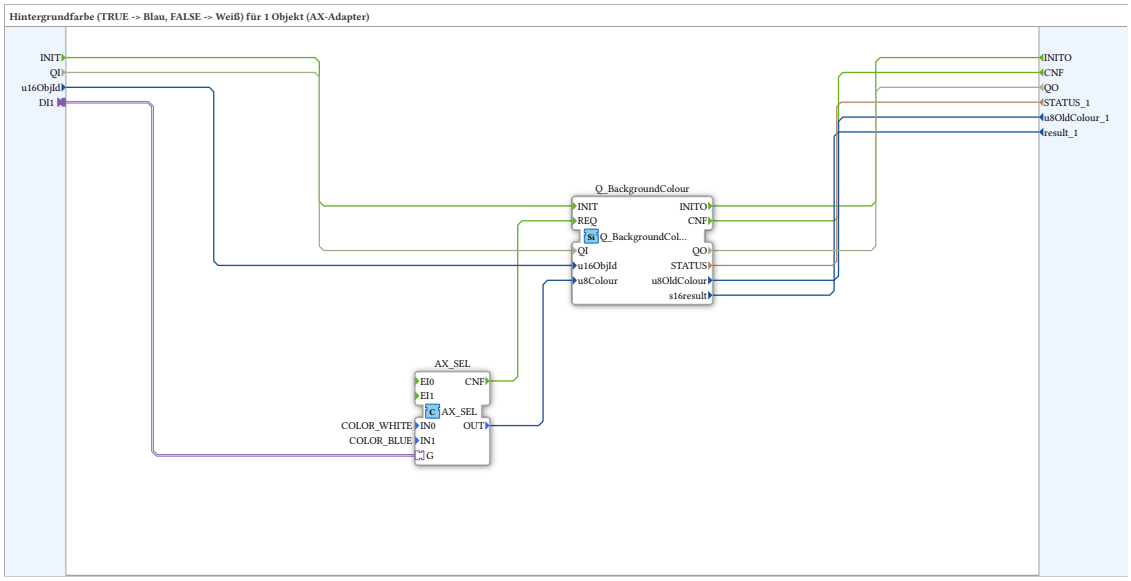
Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einer direkten Verbindung zwischen `Funk_IXA` und `DataPanel_MI_QXA` bietet diese SubApp eine **gekapselte, wiederverwendbare Einheit**. Sie reduziert die Komplexität im übergeordneten Netzwerk, da die Adapter- und Datenverbindungen bereits fest verdrahtet sind. Ein ähnlicher Baustein könnte eine separate Lösung sein, die nicht adapterbasiert arbeitet, sondern direkte Datenverbindungen nutzt – hier wird jedoch die standardisierte Adapter-Kommunikation bevorzugt, die eine spätere Erweiterung (z. B. Austausch der Funk- oder Ausgangsmodule) erleichtert.

Fazit

Die SubApp `Funk_IXA_TO_DataPanel_MI_QXA` stellt eine praktische, generische Lösung zur Kopplung von Funkempfang und DataPanel-Ausgabe dar. Durch die Kombination von Daten- und Adapterverbindungen wird eine klare Schnittstelle bereitgestellt, die einfach in größere Systeme integriert werden kann. Ihre modulare Struktur und Parametrierbarkeit machen sie für vielfältige Einsatzszenarien geeignet.

GreenBlueBackground1_AX



• – ○ • – ○ • – ○ •

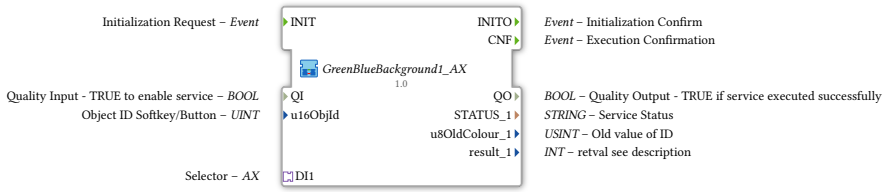
Einleitung

`GreenBlueBackground1_AX` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` → **Blau**, `FALSE` → **Weiß**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den Eingang `u16ObjId` übergeben.

Allgemeines Muster (Selector → `AX_SEL`/`F_SEL` → `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Quality-Muster: Die SubApp besitzt den Ereigniseingang `INIT`, den Ereignisausgang `INITO`, den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`, „Quality Input – TRUE to enable service“) und den Datenausgang `QO` (`BOOL`, „Quality Output – TRUE if service executed successfully“). Sie werden direkt an

`Q_BackgroundColour` durchgereicht: `INIT` ⟶ `Q_BackgroundColour.INIT`, `Q_BackgroundColour.INITO` ⟶ `INITO`, `QI` ⟶ `Q_BackgroundColour.QI`, `Q_BackgroundColour.QO` ⟶ `QO`.



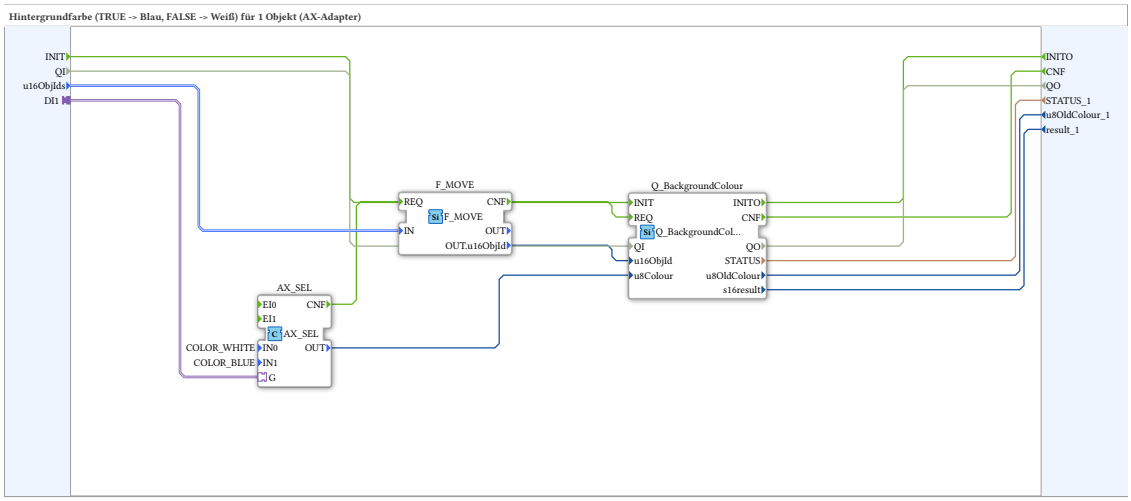
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Blau/Weiß, 1 Objekt, Adapter-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenBlueBackground1_AXS



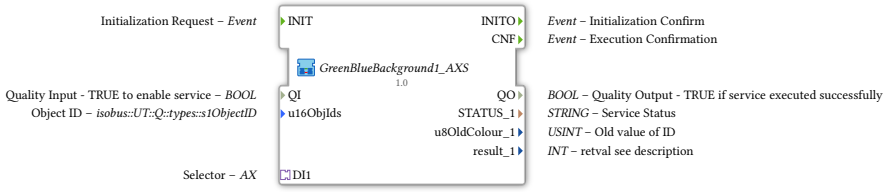
• – ○ • – ○ • – ○ •

Einleitung

`GreenBlueBackground1_AXS` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Blau**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Quality-Muster: Die SubApp besitzt den Ereigniseingang `INIT`, den Ereignisausgang `INITO`, den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`, „Quality Input – TRUE to enable service“) und den Datenausgang `QO` (`BOOL`, „Quality Output – TRUE if service executed successfully“). Sie werden direkt an `Q_BackgroundColour` durchgereicht: `INIT` ⟶ `Q_BackgroundColour.INIT`, `Q_BackgroundColour.INITO` ⟶ `INITO`, `QI` ⟶ `Q_BackgroundColour.QI`, `Q_BackgroundColour.QO` ⟶ `QO`.



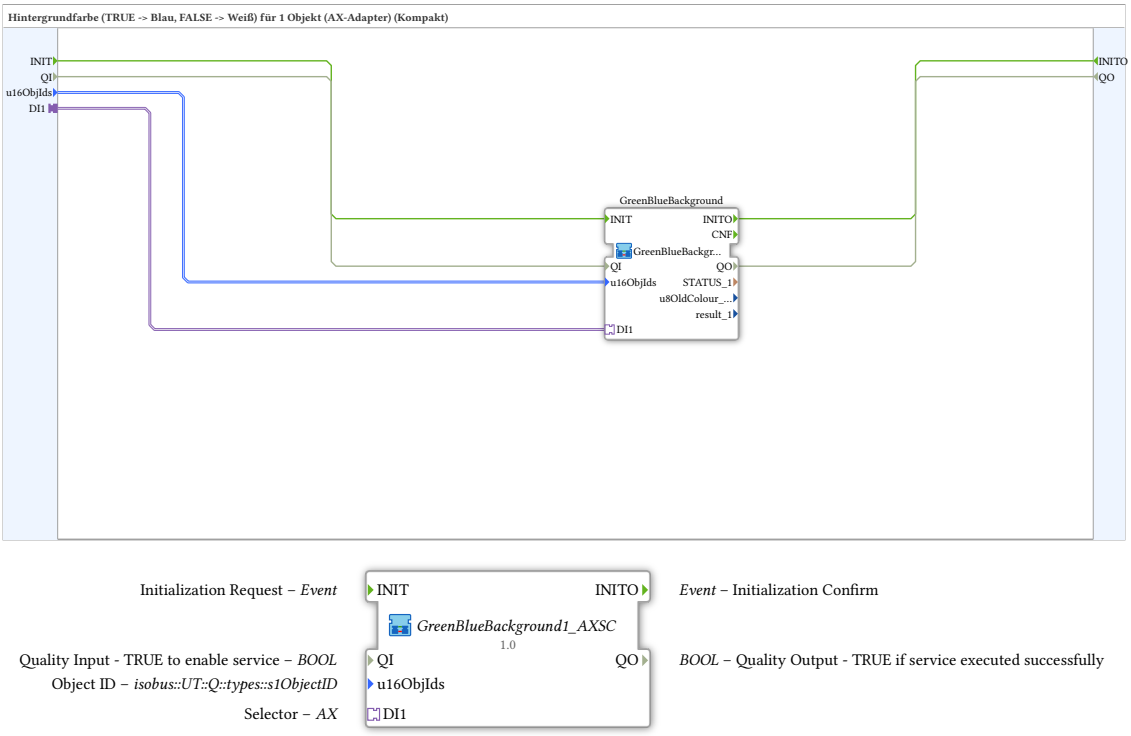
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Blau/Weiß, 1 Objekt, Adapter-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenBlueBackground1_AXSC



• – ○ • – ○ • – ○ •

Einleitung

`GreenBlueBackground1_AXSC` ist ein **Kompakt-Wrapper** um `GreenBlueBackground1_AXS` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Zusätzlich werden Initialisierung und Qualität durchgereicht: `INIT` ⟶ `GreenBlueBackground.INIT`, `GreenBlueBackground.INITO` ⟶ `INITO`, `QI` (Initialwert `TRUE`) ⟶ `GreenBlueBackground.QI` und `GreenBlueBackground.QO` ⟶ `QO`.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

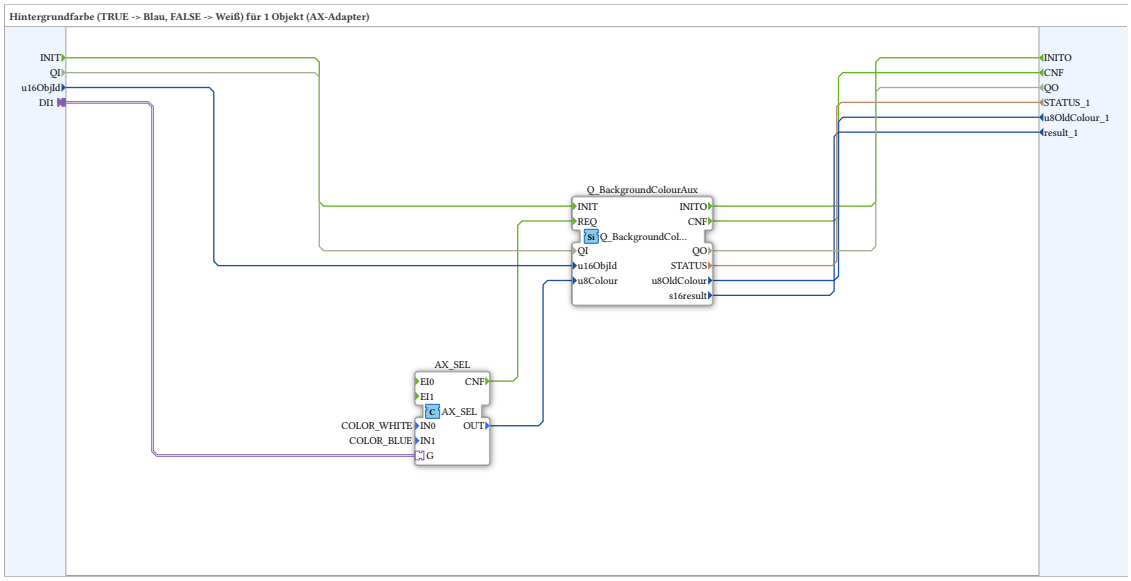
Zusammenfassung

Reine Weiterleitung an `GreenBlueBackground1_AXS` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenBlueBackground1_aux_AX



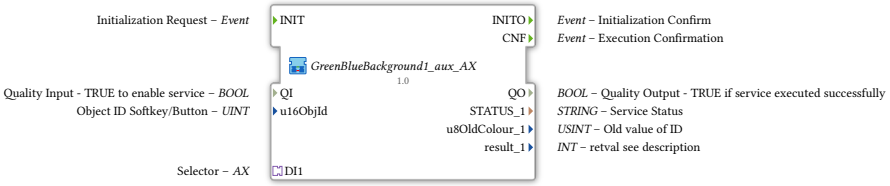
• — ○ • — ○ • — ○ •

Einleitung

`GreenBlueBackground1_aux_AX` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Blau**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den Eingang `u16ObjId` übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Quality-Muster: Die SubApp besitzt den Ereigniseingang `INIT`, den Ereignisausgang `INITO`, den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`, „Quality Input – TRUE to enable service“) und den Datenausgang `QO` (`BOOL`, „Quality Output – TRUE if service executed successfully“). Sie werden direkt an `Q_BackgroundColourAux` durchgereicht: `INIT` ⟶ `Q_BackgroundColourAux.INIT`, `Q_BackgroundColourAux.INITO` ⟶ `INITO`, `QI` ⟶ `Q_BackgroundColourAux.QI`, `Q_BackgroundColourAux.QO` ⟶ `QO`.



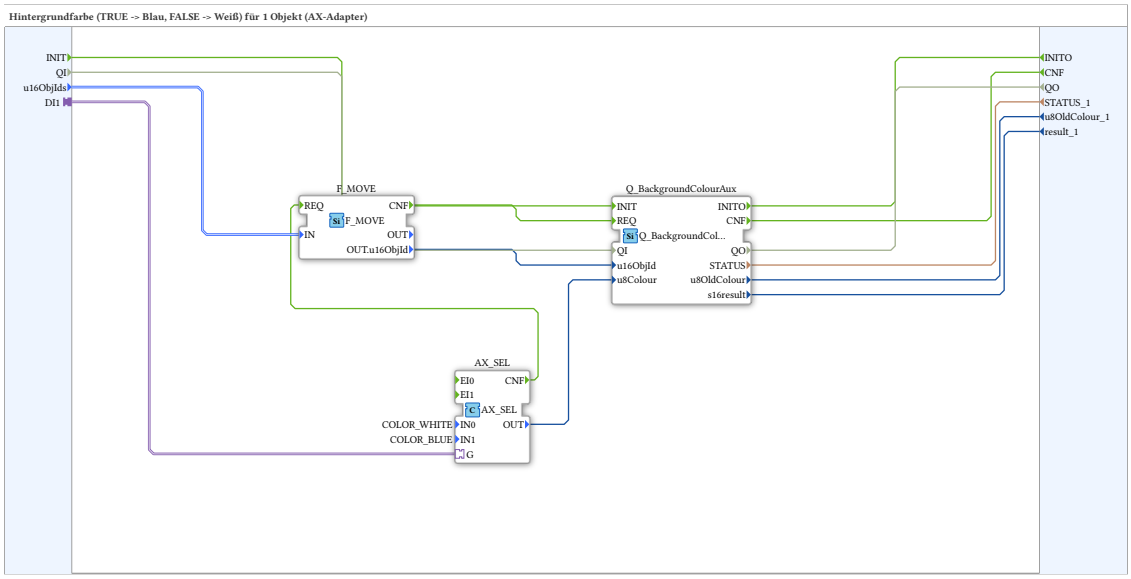
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Blau/Weiß, 1 Objekt, Adapter-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenBlueBackground1_aux_AXS



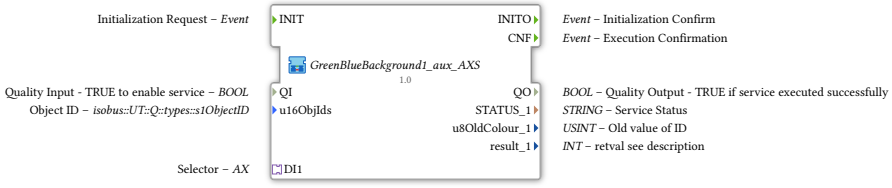
• — ○ • — ○ • — ○ •

Einleitung

`GreenBlueBackground1_aux_AXS` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Blau**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Quality-Muster: Die SubApp besitzt den Ereigniseingang `INIT`, den Ereignisausgang `INITO`, den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`, „Quality Input – TRUE to enable service“) und den Datenausgang `QO` (`BOOL`, „Quality Output – TRUE if service executed successfully“). Sie werden direkt an `Q_BackgroundColourAux` durchgereicht: `INIT` ⟶ `Q_BackgroundColourAux.INIT`, `Q_BackgroundColourAux.INITO` ⟶ `INITO`, `QI` ⟶ `Q_BackgroundColourAux.QI`, `Q_BackgroundColourAux.QO` ⟶ `QO`.



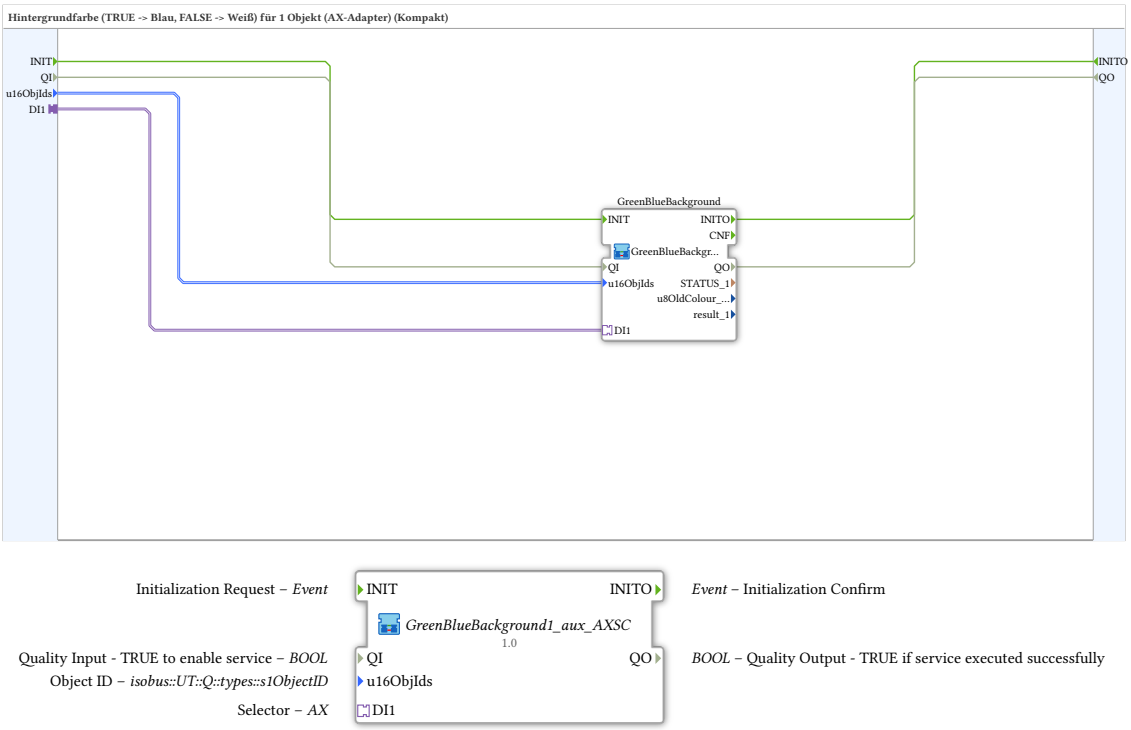
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Blau/Weiß, 1 Objekt, Adapter-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenBlueBackground1_aux_AXSC



• — ○ • — ○ • — ○ •

Einleitung

`GreenBlueBackground1_aux_AXSC` ist ein **Kompakt-Wrapper** um `GreenBlueBackground1_aux_AXS` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

Zusätzlich führt der Wrapper die Schnittstelle für Initialisierung und Qualität nach außen: Ereigniseingang `INIT`, Ereignisausgang `INITO`, Dateneingang `QI` (`BOOL`, Initialwert `TRUE`) und Datenausgang `QO` (`BOOL`). Sie sind 1:1 an die innere Instanz `GreenBlueBackground` durchgeschleift: `INIT` ⟶ `GreenBlueBackground.INIT`, `GreenBlueBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenBlueBackground.QI`, `GreenBlueBackground.QO` ⟶ `QO`.

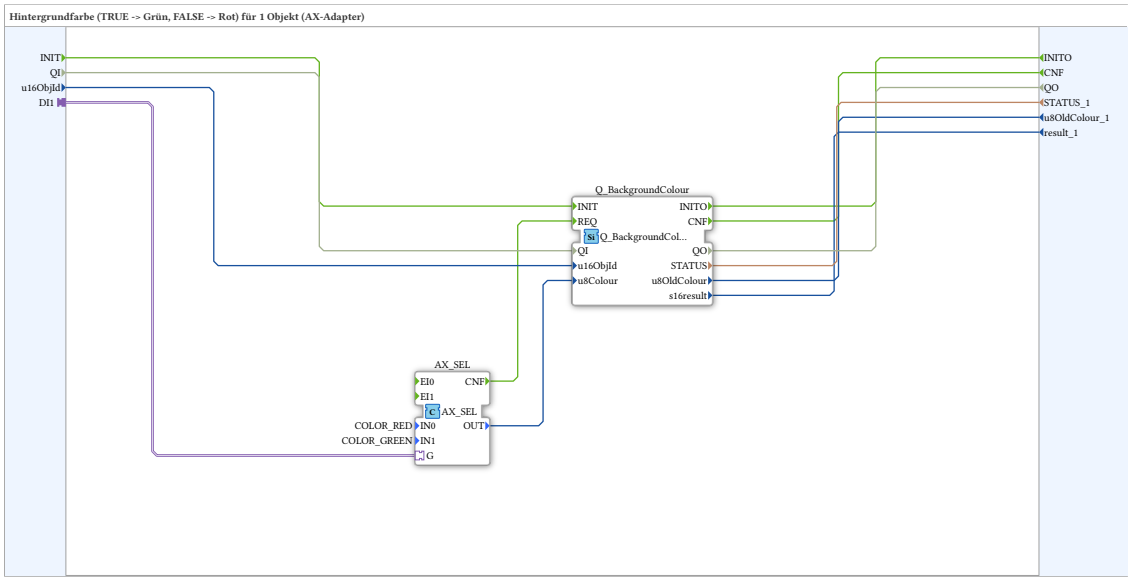
Zusammenfassung

Reine Weiterleitung an `GreenBlueBackground1_aux_AXS` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenRedBackground1_AX



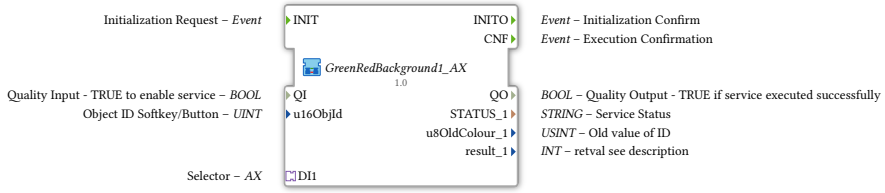
• — ○ • — ○ • — ○ •

Einleitung

`GreenRedBackground1_AX` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den Eingang `u16ObjId` übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Quality-Muster: Die SubApp besitzt den Ereigniseingang `INIT`, den Ereignisausgang `INITO`, den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`, „Quality Input – TRUE to enable service“) und den Datenausgang `QO` (`BOOL`, „Quality Output – TRUE if service executed successfully“). Sie werden direkt an `Q_BackgroundColour` durchgereicht: `INIT` ⟶ `Q_BackgroundColour.INIT`, `Q_BackgroundColour.INITO` ⟶ `INITO`, `QI` ⟶ `Q_BackgroundColour.QI`, `Q_BackgroundColour.QO` ⟶ `QO`.



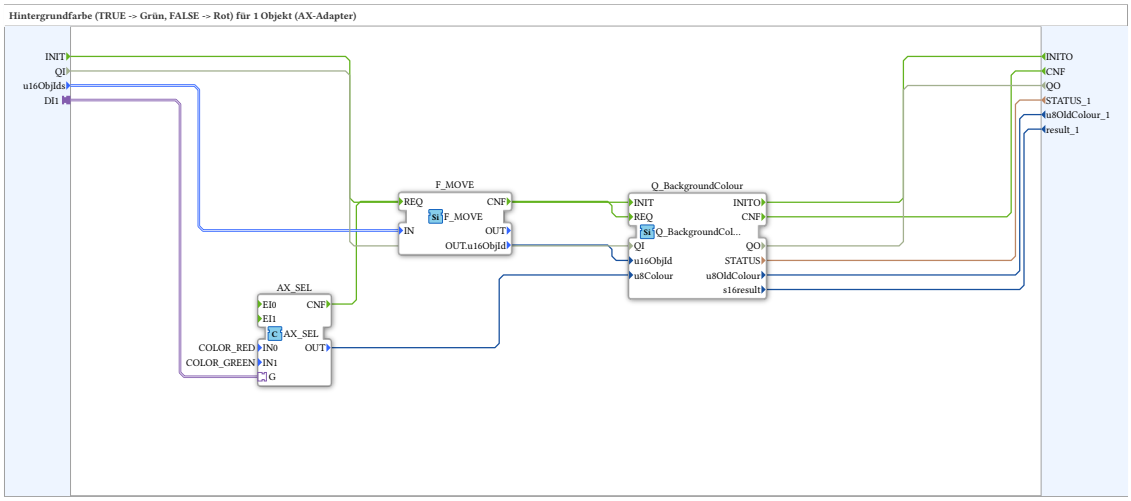
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Rot, 1 Objekt, Adapter-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenRedBackground1_AXS



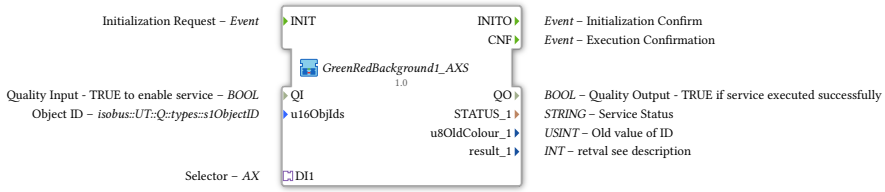
• – ○ • – ○ • – ○ •

Einleitung

`GreenRedBackground1_AXS` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Quality-Muster: Die SubApp besitzt den Ereigniseingang `INIT`, den Ereignisausgang `INITO`, den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`, „Quality Input – TRUE to enable service“) und den Datenausgang `QO` (`BOOL`, „Quality Output – TRUE if service executed successfully“). Sie werden direkt an `Q_BackgroundColour` durchgereicht: `INIT` ⟶ `Q_BackgroundColour.INIT`, `Q_BackgroundColour.INITO` ⟶ `INITO`, `QI` ⟶ `Q_BackgroundColour.QI`, `Q_BackgroundColour.QO` ⟶ `QO`.



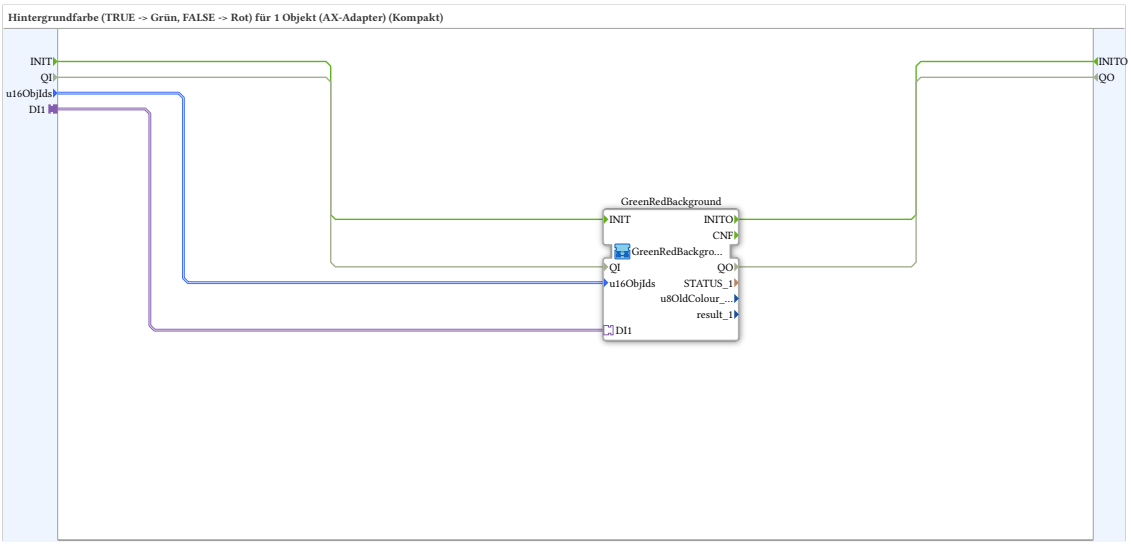
Zusammenfassung

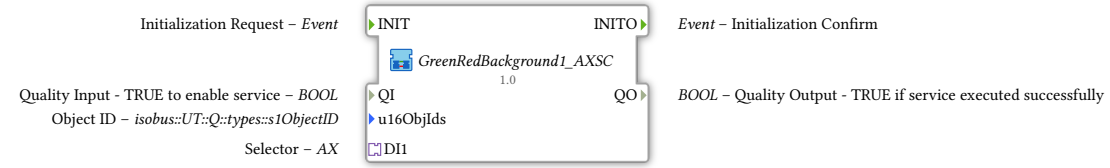
Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Rot, 1 Objekt, Adapter-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenRedBackground1_AXSC





• – ○ • – ○ • – ○ •

Einleitung

`GreenRedBackground1_AXSC` ist ein **Kompakt-Wrapper** um `GreenRedBackground1_AXS` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

Initialisierung und Qualität

- **Ereignis-Eingang `INIT`** (`EInit`) und **Ereignis-Ausgang `INITO`** (`EInit`) werden durchgereicht: `INIT` ⟶ `GreenRedBackground.INIT`; `GreenRedBackground.INITO` ⟶ `INITO`.
- **Dateneingang `QI`** (`BOOL`, Vorgabe `TRUE`) ⟶ `GreenRedBackground.QI`; **Datenausgang `QO`** (`BOOL`) ⟵ `GreenRedBackground.QO`.

Die innere SubApp-Instanz `GreenRedBackground` ist vom Typ `GreenRedBackground1_AXS`.

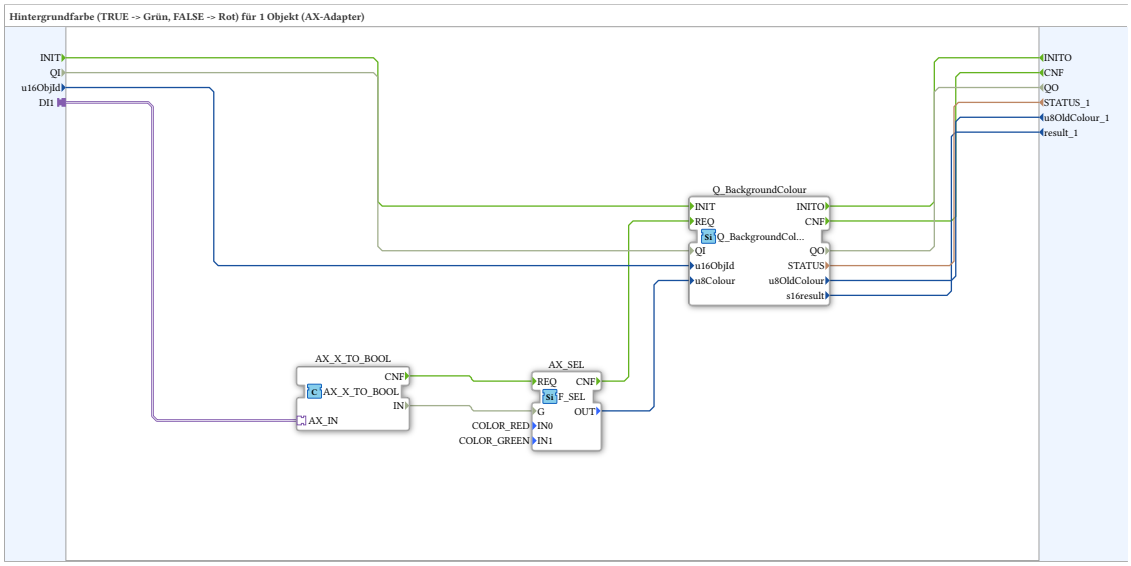
Zusammenfassung

Reine Weiterleitung an `GreenRedBackground1_AXS` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

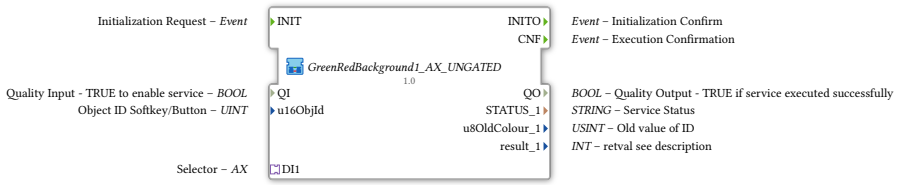
GreenRedBackground1_AX_UNGATED



• — ○ • — ○ • — ○ •

Einleitung

`GreenRedBackground1_AX_UNGATED` setzt die **Hintergrundfarbe** eines VT-Objekts über einen AX-Adapter: `TRUE` ⟶ Grün, `FALSE` ⟶ Rot. „UNGATED“ bedeutet, dass der Selektor nicht durch ein zusätzliches Freigabesignal gegatet wird – jedes Update von `DI1` schreibt die Farbe sofort.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung – an `Q_BackgroundColour.INIT`; weitere Ereignisse kommen über den Adapter-Socket

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung (von `Q_BackgroundColour.INITO`)
- **CNF**: Ausführungsbestätigung (von `Q_BackgroundColour.CNF`)

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input – TRUE, um den Dienst freizugeben
- **u16ObjId** (`UINT`, Vorgabe `ID_NULL`): Object-ID des Softkeys/Buttons

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde (von `Q_BackgroundColour.QO`)
- **STATUS_1** (`STRING`): Dienststatus
- **u8OldColour_1** (`USINT`): Alter Farbwert des Objekts
- **result_1** (`INT`): Rückgabewert (siehe `Q_BackgroundColour`)

Adapter

- **DI1** (Socket, `adapter::types::unidirectional::AX`): Selektor – `TRUE` für Grün, `FALSE` für Rot

Verwendete Funktionsbausteine (FBs)

- **AX_X_TO_BOOL**: `adapter::conversion::unidirectional::AX_X_TO_BOOL` – wandelt den AX-Selektor in Ereignis `CNF` und Wert `IN` (`BOOL`)
- **AX_SEL**: `iec61131::selection::F_SEL` mit `IN0 = COLOR_RED`, `IN1 = COLOR_GREEN` – wählt die Farbe
- **Q_BackgroundColour**: `isobus::UT::Q::Q_BackgroundColour` – schreibt die Hintergrundfarbe auf das VT-Objekt

Programmablauf und Verbindungen

1. `DI1` ⟶ `AX_X_TO_BOOL.AX_IN`; `AX_X_TO_BOOL.IN` ⟶ `AX_SEL.G`; `AX_X_TO_BOOL.CNF` ⟶ `AX_SEL.REQ`.
2. `AX_SEL.OUT` ⟶ `Q_BackgroundColour.u8Colour`; `AX_SEL.CNF` ⟶ `Q_BackgroundColour.REQ`.
3. `u16ObjId` ⟶ `Q_BackgroundColour.u16ObjId`.
4. `Q_BackgroundColour.CNF` ⟶ `CNF`; `STATUS`, `u8OldColour`, `s16result` ⟶ `STATUS_1`, `u8OldColour_1`, `result_1`.
5. `INIT` ⟶ `Q_BackgroundColour.INIT`; `Q_BackgroundColour.INITO` ⟶ `INITO`; `QI` ⟶ `Q_BackgroundColour.QI`; `Q_BackgroundColour.QO` ⟶ `QO`.

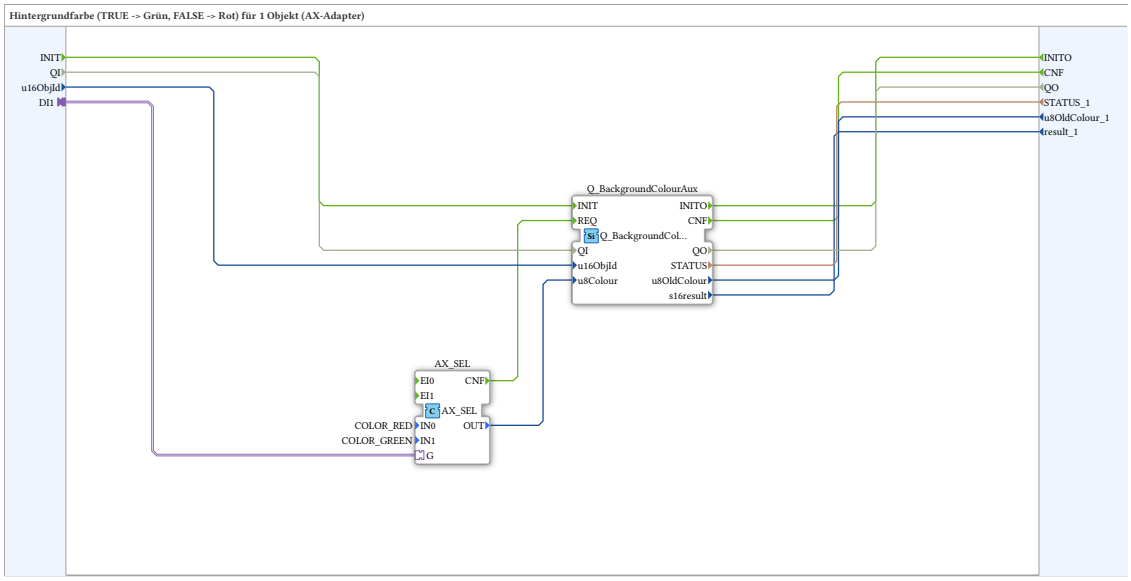
Anwendungsszenarien

- Grün/Rot-Statusanzeige für Buttons oder Flächen (z. B. Betrieb/Störung) ohne zusätzliche Freigabelogik

Fazit

`GreenRedBackground1_AX_UNGATED` färbt ein VT-Objekt abhängig von einem AX-Signal sofort grün oder rot.

GreenRedBackground1_aux_AX



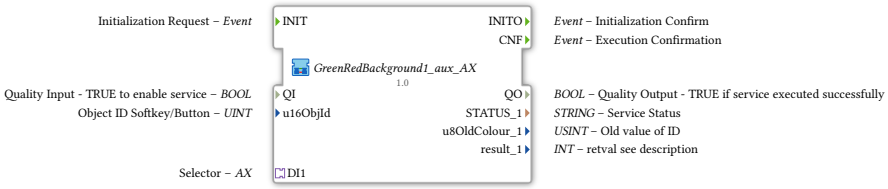
• – ○ • – ○ • – ○ •

Einleitung

`GreenRedBackground1_aux_AX` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den Eingang `u16ObjId` übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Quality-Muster: Die SubApp besitzt den Ereigniseingang `INIT`, den Ereignisausgang `INITO`, den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`, „Quality Input – TRUE to enable service“) und den Datenausgang `QO` (`BOOL`, „Quality Output – TRUE if service executed successfully“). Sie werden direkt an `Q_BackgroundColourAux` durchgereicht: `INIT` ⟶ `Q_BackgroundColourAux.INIT`, `Q_BackgroundColourAux.INITO` ⟶ `INITO`, `QI` ⟶ `Q_BackgroundColourAux.QI`, `Q_BackgroundColourAux.QO` ⟶ `QO`.



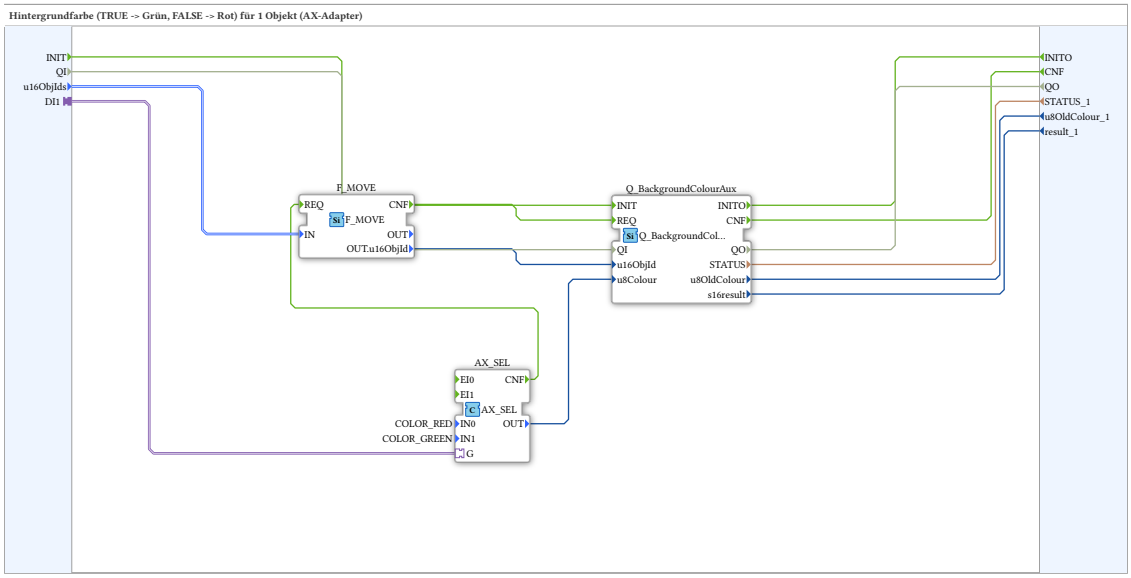
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Rot, 1 Objekt, Adapter-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenRedBackground1_aux_AXS



• — ○ • — ○ • — ○ •

Einleitung

`GreenRedBackground1_aux_AXS` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Quality-Muster: Die SubApp besitzt den Ereigniseingang `INIT`, den Ereignisausgang `INITO`, den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`, „Quality Input – TRUE to enable service“) und den Datenausgang `QO` (`BOOL`, „Quality Output – TRUE if service executed successfully“). Sie werden direkt an `Q_BackgroundColourAux` durchgereicht: `INIT` ⟶ `Q_BackgroundColourAux.INIT`, `Q_BackgroundColourAux.INITO` ⟶ `INITO`, `QI` ⟶ `Q_BackgroundColourAux.QI`, `Q_BackgroundColourAux.QO` ⟶ `QO`.

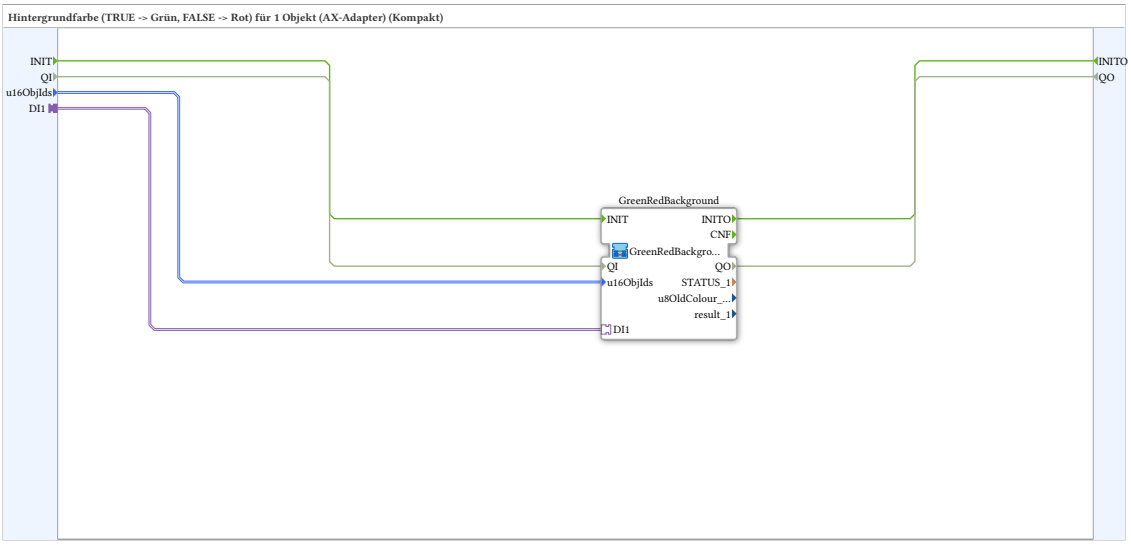
Zusammenfassung

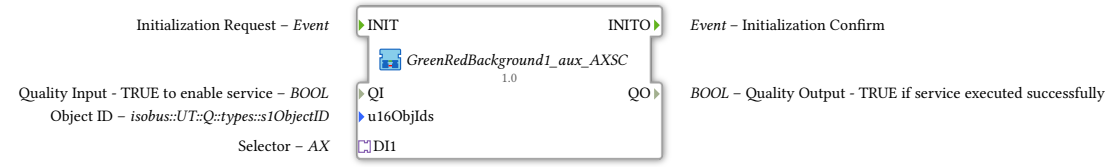
Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Rot, 1 Objekt, Adapter-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenRedBackground1_aux_AXSC





• – ○ • – ○ • – ○ •

Einleitung

`GreenRedBackground1_aux_AXSC` ist ein **Kompakt-Wrapper** um `GreenRedBackground1_aux_AXS` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Schnittstelle (Quality-Muster): Der Wrapper besitzt zusätzlich den Ereignis-Eingang `INIT` und den Ereignis-Ausgang `INITO` (je `EInit`) sowie den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`) und den Datenausgang `QO` (`BOOL`). Sie sind direkt an die innere Instanz `GreenRedBackground` durchgeschleift: `INIT` ⟶ `GreenRedBackground.INIT`, `GreenRedBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenRedBackground.QI`, `GreenRedBackground.QO` ⟶ `QO`.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

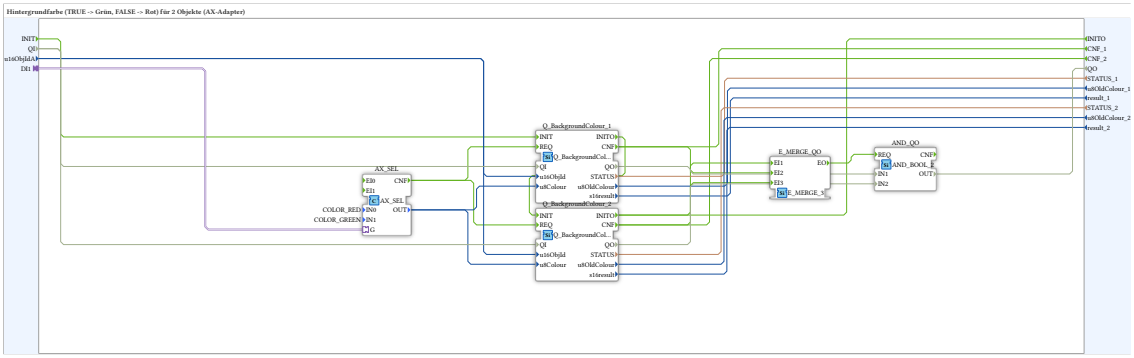
Zusammenfassung

Reine Weiterleitung an `GreenRedBackground1_aux_AXS` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenRedBackground2_AX



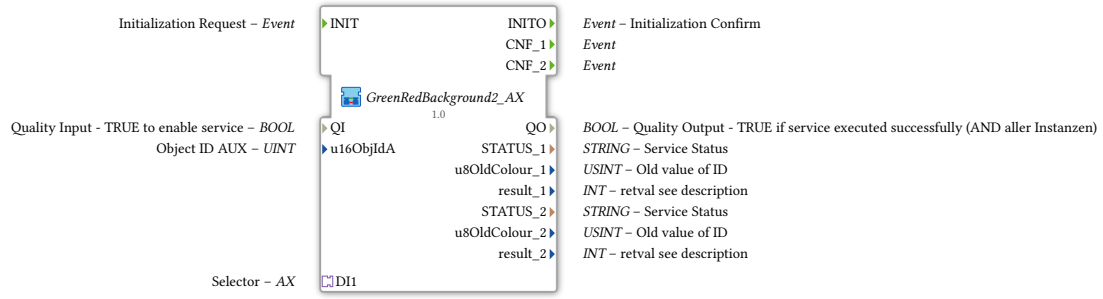
• — ○ • — ○ • — ○ •

Einleitung

`GreenRedBackground2_AX` schaltet die VT-Hintergrundfarbe von 2 Objekten anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über die Eingänge `u16ObjIdA` übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
2	`u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Initialisierung und Status (INIT/INITO, QI/QO)

- **Ereignis-Eingang** `INIT` (`EInit`, Initialization Request) und **Ereignis-Ausgang** `INITO` (`EInit`, Initialization Confirm)
- **Daten-Eingang** `QI` (`BOOL`, Vorgabe `TRUE`, Quality Input) wird an `Q_BackgroundColour_1.QI` und `Q_BackgroundColour_2.QI` weitergegeben
- **Daten-Ausgang** `QO` (`BOOL`, Quality Output): UND (`AND_QO`, Typ `iec61131::booleanOperators::AND_BOOL_2`) der `QO`-Ausgänge von `Q_BackgroundColour_1.QO` und `Q_BackgroundColour_2.QO`
- **Initialisierungskette**: `INIT` ⟶ `Q_BackgroundColour_1.INIT`, `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`, `Q_BackgroundColour_2.INITO` ⟶ `INITO`
- `E_MERGE_QO` (Typ `iec61499::events::E_MERGE_3`) führt `Q_BackgroundColour_2.INITO`, `Q_BackgroundColour_1.CNF` und `Q_BackgroundColour_2.CNF` zusammen und löst `AND_QO.REQ` aus

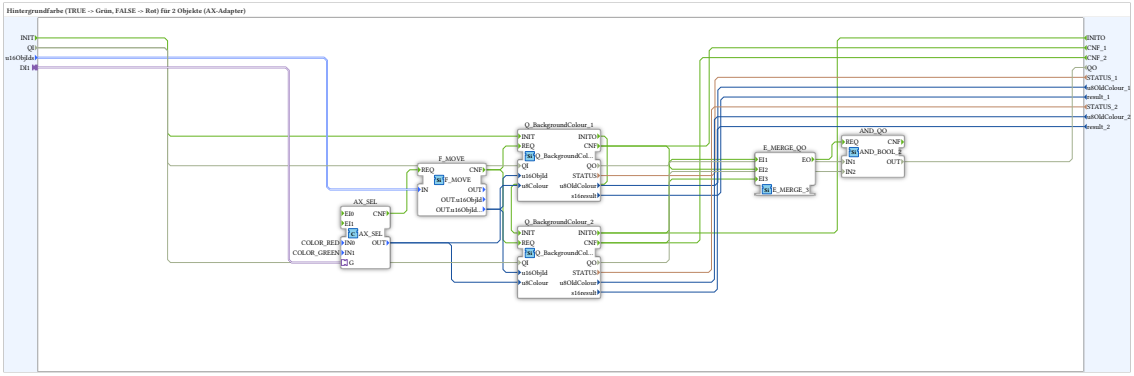
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Rot, 2 Objekte, Adapter-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenRedBackground2_AXS



• – ○ • – ○ • – ○ •

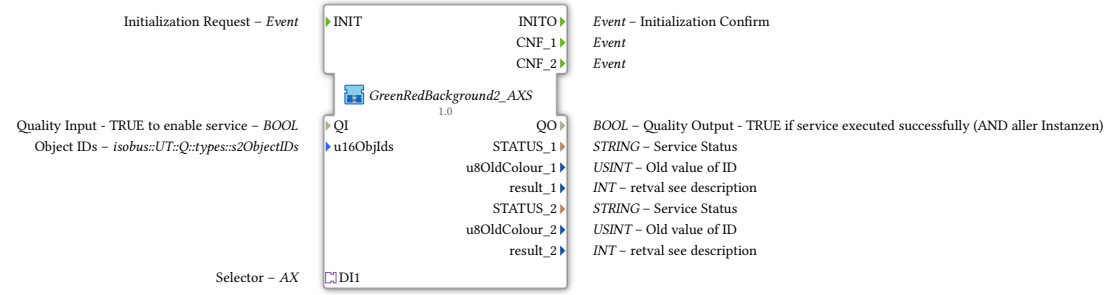
Einleitung

`GreenRedBackground2_AXS` schaltet die VT-Hintergrundfarbe von 2 Objekten anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s2ObjectIDs` (`u16ObjIds`) übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Initialisierung und Qualität: Die SubApp besitzt `INIT`/`INITO` sowie `QI`/`QO`. `INIT` ⟶ `Q_BackgroundColour_1.INIT`, `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`, `Q_BackgroundColour_2.INITO` ⟶ `INITO`. `QI` geht an beide `Q_BackgroundColour`-Instanzen. `QO` ist das `AND` (`AND_QO`, `AND_BOOL_2`) der beiden `QO`-Ausgänge; `AND_QO.REQ` wird über `E_MERGE_QO` (`E_MERGE_3`: `Q_BackgroundColour_2.INITO`, `Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF`) ausgelöst.



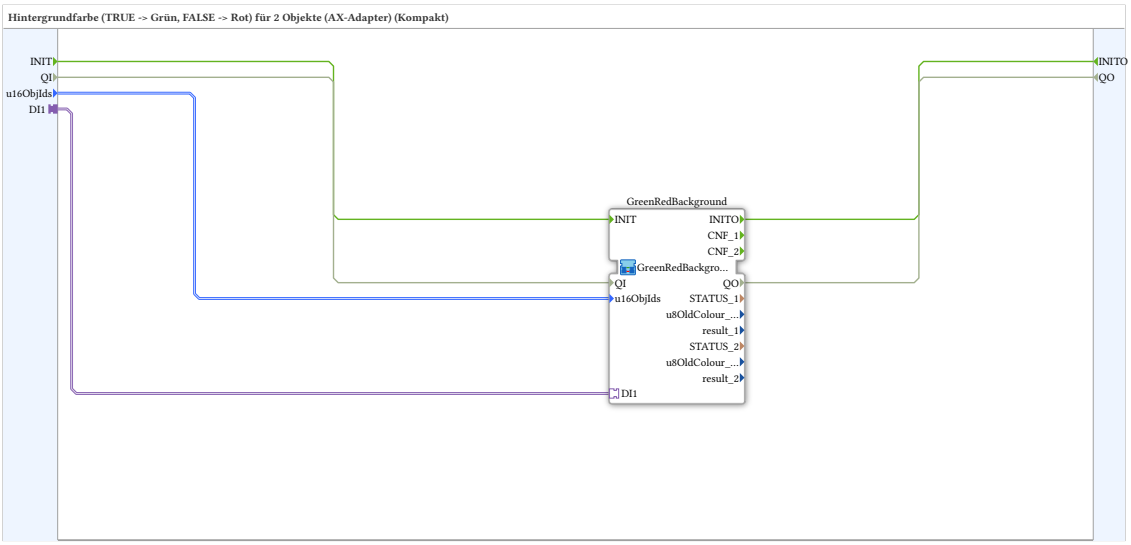
Zusammenfassung

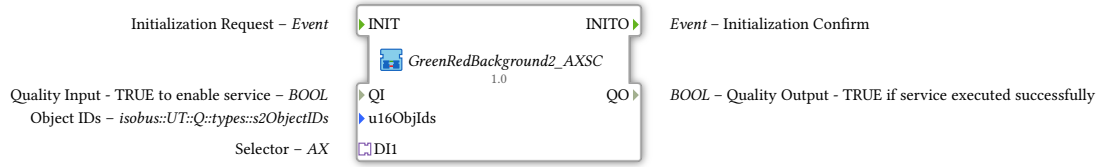
Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Rot, 2 Objekte, Adapter-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenRedBackground2_AXSC





• – ○ • – ○ • – ○ •

Einleitung

`GreenRedBackground2_AXSC` ist ein **Kompakt-Wrapper** um `GreenRedBackground2_AXS` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Zusätzlich werden Initialisierung und Qualität durchgereicht: `INIT` ⟶ `GreenRedBackground.INIT`, `GreenRedBackground.INITO` ⟶ `INITO`, `QI` (Initialwert `TRUE`) ⟶ `GreenRedBackground.QI` und `GreenRedBackground.QO` ⟶ `QO`.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

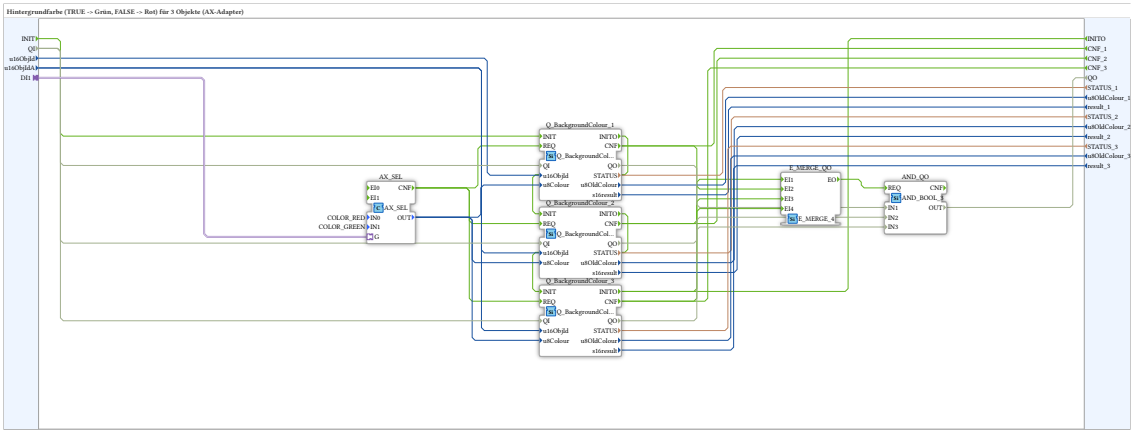
Zusammenfassung

Reine Weiterleitung an `GreenRedBackground2_AXS` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenRedBackground3_AX



• – ○ • – ○ • – ○ •

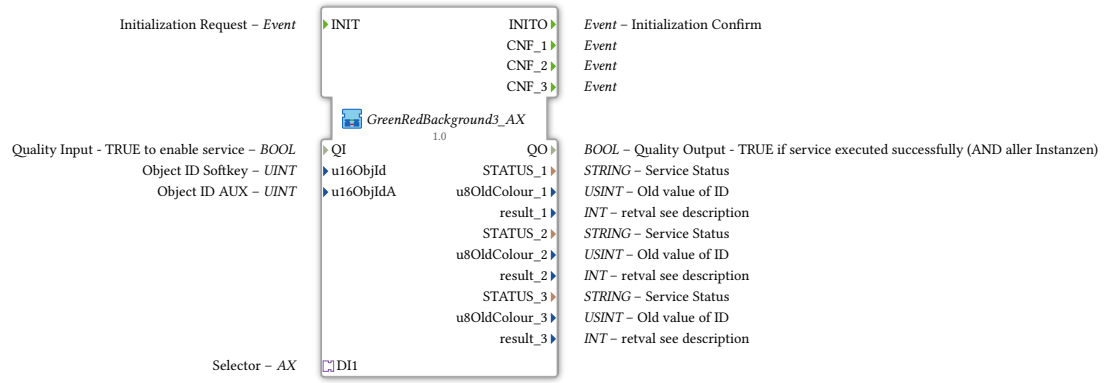
Einleitung

`GreenRedBackground3_AX` schaltet die VT-Hintergrundfarbe von 3 Objekten anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über die Eingänge `u16ObjId, u16ObjIdA` übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`u16ObjId`	`Q_BackgroundColour` (normales Objekt)
2	`u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
3	`u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Zusätzlich besitzt der Baustein die Schnittstelle für Initialisierung und Qualität: Ereigniseingang `INIT`, Ereignisausgang `INITO`, Dateneingang `QI` (`BOOL`, Initialwert `TRUE`) und Datenausgang `QO` (`BOOL`, AND aller Instanzen). `INIT` ⟶ `Q_BackgroundColour_1.INIT`, `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`, `Q_BackgroundColour_2.INITO` ⟶ `Q_BackgroundColour_3.INIT`, `Q_BackgroundColour_3.INITO` ⟶ `INITO` (die drei Instanzen werden also nacheinander initialisiert). `QI` geht an `QI` aller drei Instanzen. `QO` ist der Ausgang der Instanz `AND_QO` (`iec61131::booleanOperators::AND_BOOL_3`), die `Q_BackgroundColour_1.QO`, `Q_BackgroundColour_2.QO` und `Q_BackgroundColour_3.QO` verknüpft; ihr `REQ` wird von `E_MERGE_QO` (`iec61499::events::E_MERGE_4`) ausgelöst, das `Q_BackgroundColour_3.INITO` und die `CNF`-Ereignisse der drei Instanzen zusammenführt.



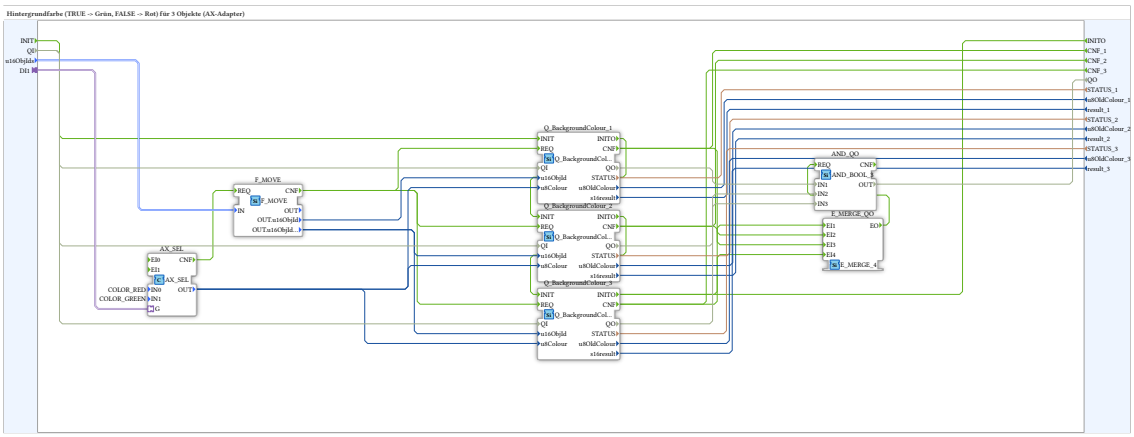
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Rot, 3 Objekte, Adapter-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenRedBackground3_AXS



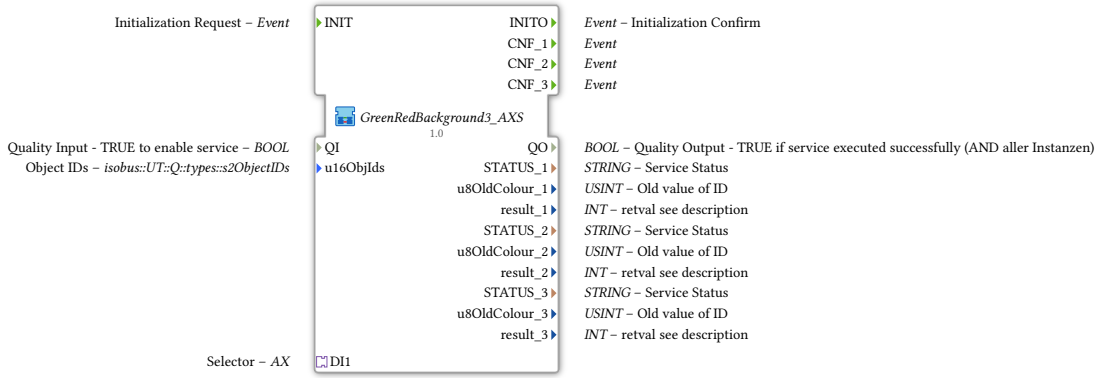
• — ○ • — ○ • — ○ •

Einleitung

`GreenRedBackground3_AXS` schaltet die VT-Hintergrundfarbe von 3 Objekten anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s2ObjectIDs` (`u16ObjIds`) übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`F_MOVE.OUT.u16ObjId`	`Q_BackgroundColour` (normales Objekt)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
3	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Initialisierung und Qualität

- **Ereignis-Eingang `INIT`** (`EInit`) und **Ereignis-Ausgang `INITO`** (`EInit`): die Initialisierung wird durch die `Q_BackgroundColour`-Instanzen gekettet: `INIT` ⟶ `Q_BackgroundColour_1.INIT`; `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` ⟶ `Q_BackgroundColour_3.INIT`; `Q_BackgroundColour_3.INITO` ⟶ `INITO`.
- **Dateneingang `QI`** (`BOOL`, Vorgabe `TRUE`) wird an `.QI` aller Instanzen verteilt (`Q_BackgroundColour_1`, `Q_BackgroundColour_2`, `Q_BackgroundColour_3`).
- **Datenausgang `QO`** (`BOOL`): UND aller Instanzen über `AND_QO` (`iec61131::booleanOperators::AND_BOOL_3`): `Q_BackgroundColour_1.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN2`, `Q_BackgroundColour_3.QO` ⟶ `AND_QO.IN3`; `AND_QO.OUT` ⟶ `QO`.

- `E_MERGE_QO` (`iec61499::events::E_MERGE_4`) vereint `Q_BackgroundColour_3.INITO` und die `CNF`-Ereignisse aller drei Instanzen (`Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF`, `Q_BackgroundColour_3.CNF`) und löst damit `AND_QO.REQ` aus.

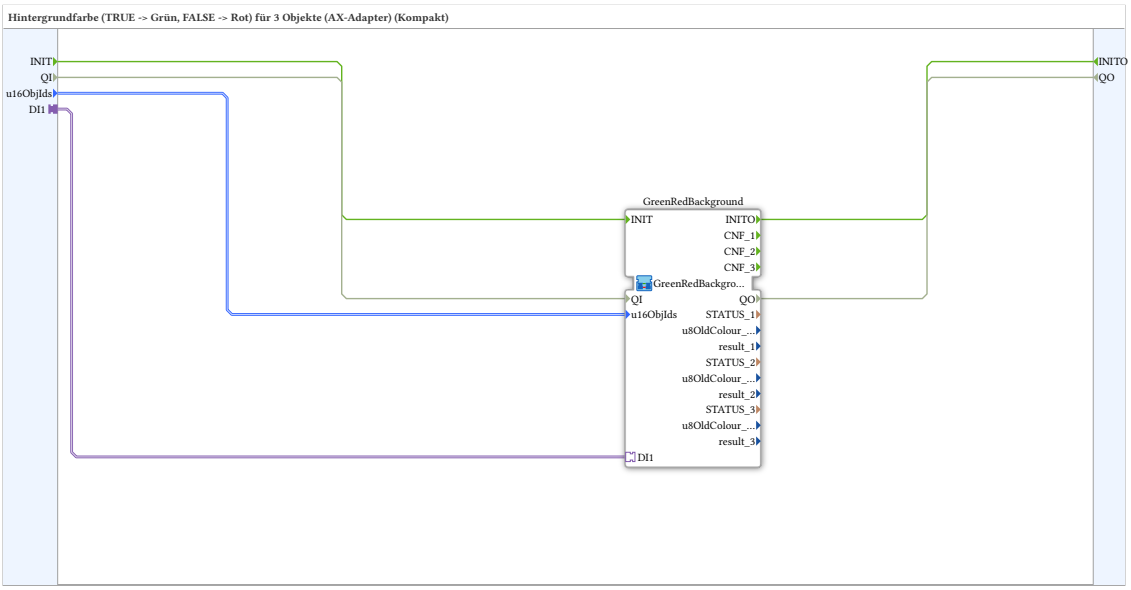
Zusammenfassung

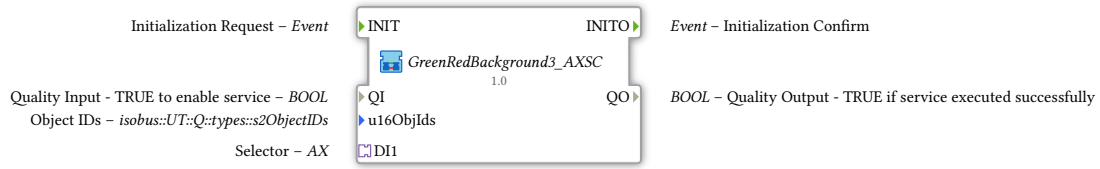
Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Rot, 3 Objekte, Adapter-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenRedBackground3_AXSC





• — ○ • — ○ • — ○ •

Einleitung

`GreenRedBackground3_AXSC` ist ein **Kompakt-Wrapper** um `GreenRedBackground3_AXS` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Schnittstelle (Quality-Muster): Der Wrapper besitzt zusätzlich den Ereignis-Eingang `INIT` und den Ereignis-Ausgang `INITO` (je `EInit`) sowie den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`) und den Datenausgang `QO` (`BOOL`). Sie sind direkt an die innere Instanz `GreenRedBackground` durchgeschleift: `INIT` ⟶ `GreenRedBackground.INIT`, `GreenRedBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenRedBackground.QI`, `GreenRedBackground.QO` ⟶ `QO`.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

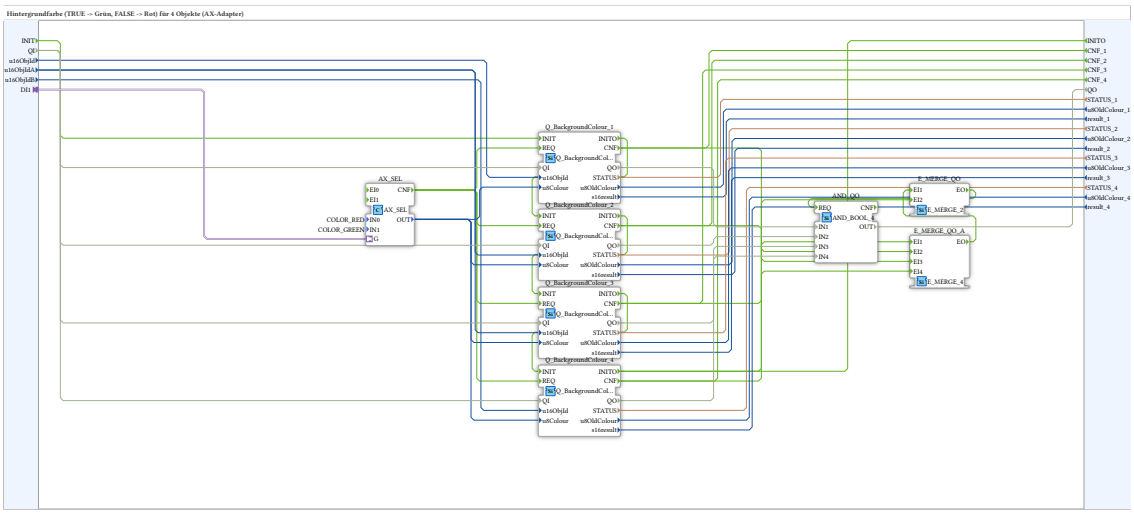
Zusammenfassung

Reine Weiterleitung an `GreenRedBackground3_AXS` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenRedBackground4_AX



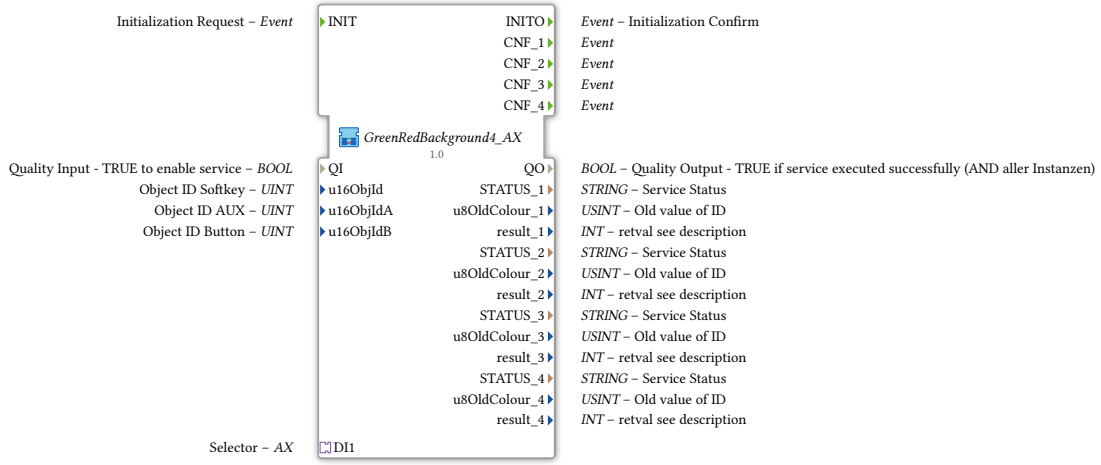
• — ○ • — ○ • — ○ •

Einleitung

`GreenRedBackground4_AX` schaltet die VT-Hintergrundfarbe von 4 Objekten anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über die Eingänge `u16ObjId, u16ObjIdA, u16ObjIdB` übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`u16ObjId`	`Q_BackgroundColour` (normales Objekt)
2	`u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
3	`u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)
4	`u16ObjIdB`	`Q_BackgroundColour` (normales Objekt)

Allgemeines Muster (Selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Initialisierung und Status (INIT/INITO, QI/QO)

- **Ereignis-Eingang** `INIT` (`EInit`, Initialization Request) und **Ereignis-Ausgang** `INITO` (`EInit`, Initialization Confirm)
- **Daten-Eingang** `QI` (`BOOL`, Vorgabe `TRUE`, Quality Input) wird an `Q_BackgroundColour_1.QI`, `Q_BackgroundColour_2.QI`, `Q_BackgroundColour_3.QI` und `Q_BackgroundColour_4.QI` weitergegeben

- **Daten-Ausgang** `QO` (`BOOL`, Quality Output): UND (`AND_QO`, Typ `iec61131::booleanOperators::AND_BOOL_4`) der `QO`-Ausgänge von `Q_BackgroundColour_1.QO`, `Q_BackgroundColour_2.QO`, `Q_BackgroundColour_3.QO` und `Q_BackgroundColour_4.QO`
- **Initialisierungskette**: `INIT` → `Q_BackgroundColour_1.INIT`, `Q_BackgroundColour_1.INITO` → `Q_BackgroundColour_2.INIT`, `Q_BackgroundColour_2.INITO` → `Q_BackgroundColour_3.INIT`, `Q_BackgroundColour_3.INITO` → `Q_BackgroundColour_4.INIT`, `Q_BackgroundColour_4.INITO` → `INITO`
- `E_MERGE_QO_A` (Typ `iec61499::events::E_MERGE_4`) führt `Q_BackgroundColour_4.INITO`, `Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF` und `Q_BackgroundColour_3.CNF` zusammen und gibt das Ergebnis an `E_MERGE_QO.EI1`
- `E_MERGE_QO` (Typ `iec61499::events::E_MERGE_2`) führt `E_MERGE_QO_A.EO` und `Q_BackgroundColour_4.CNF` zusammen und löst `AND_QO.REQ` aus

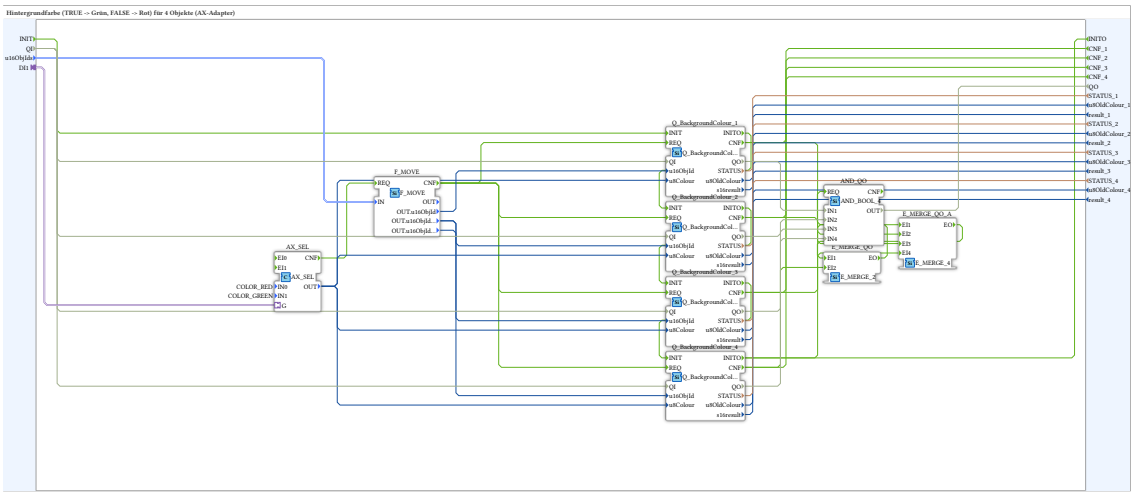
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Rot, 4 Objekte, Adapter-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenRedBackground4_AXS



• – ○ • – ○ • – ○ •

Einleitung

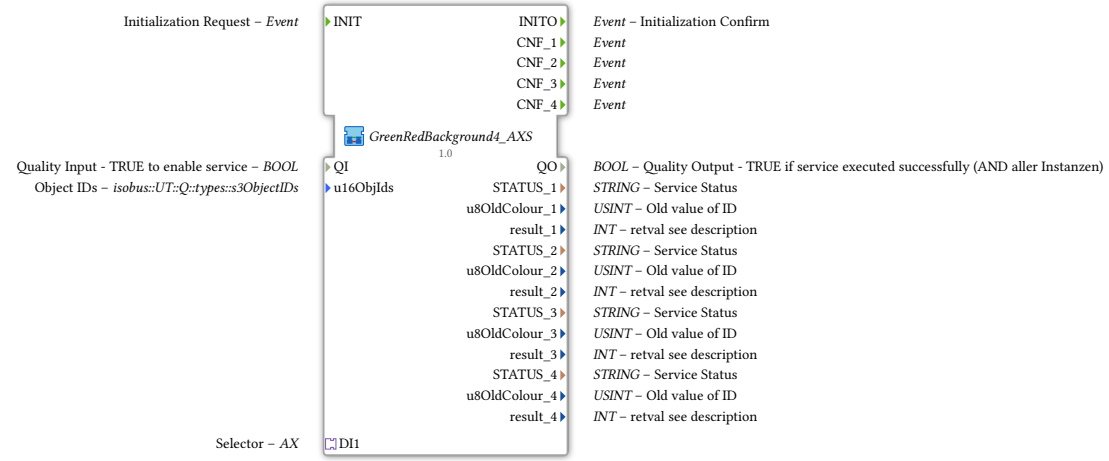
`GreenRedBackground4_AXS` schaltet die VT-Hintergrundfarbe von 4 Objekten anhand eines booleschen Selector-Signals: `TRUE` → **Grün**, `FALSE` → **Rot**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s3ObjectIDs` (`u16ObjIds`) übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`F_MOVE.OUT.u16ObjId`	`Q_BackgroundColour` (normales Objekt)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
3	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)
4	`F_MOVE.OUT.u16ObjIdB`	`Q_BackgroundColour` (normales Objekt)

Allgemeines Muster (Selector → `AX_SEL`/`F_SEL` → `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Initialisierung und Qualität: Die SubApp besitzt `INIT`/`INITO` sowie `QI`/`QO`. `INIT` → `Q_BackgroundColour_1.INIT` → `Q_BackgroundColour_2.INIT` → `Q_BackgroundColour_3.INIT` → `Q_BackgroundColour_4.INIT` (jeweils über `INITO`), `Q_BackgroundColour_4.INITO` → `INITO`. `QI` geht an alle vier Instanzen. `QO` ist das `AND` (`AND_QO`, `AND_BOOL_4`) der vier `QO`-Ausgänge; `AND_QO.REQ` wird über die

Kaskade `E_MERGE_QO_A` (`E_MERGE_4`: `Q_BackgroundColour_4.INITO`, `Q_BackgroundColour_1.CNF`, `_2.CNF`, `_3.CNF`) und `E_MERGE_QO` (`E_MERGE_2`: Ausgang von `E_MERGE_QO_A` und `Q_BackgroundColour_4.CNF`) ausgelöst.



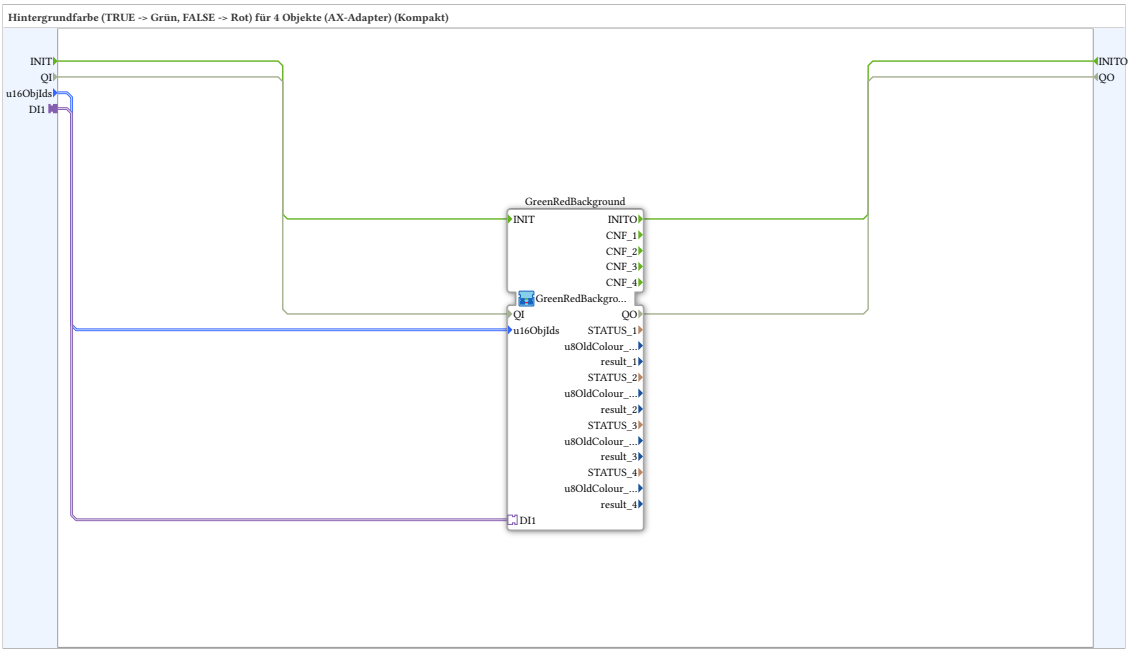
Zusammenfassung

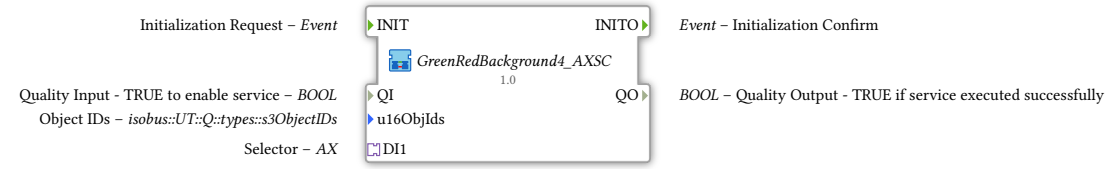
Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Rot, 4 Objekte, Adapter-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenRedBackground4_AXSC





• – ○ • – ○ • – ○ •

Einleitung

`GreenRedBackground4_AXSC` ist ein **Kompakt-Wrapper** um `GreenRedBackground4_AXS` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Zusätzlich werden Initialisierung und Qualität durchgereicht: `INIT` ⟶ `GreenRedBackground.INIT`, `GreenRedBackground.INITO` ⟶ `INITO`, `QI` (Initialwert `TRUE`) ⟶ `GreenRedBackground.QI` und `GreenRedBackground.QO` ⟶ `QO`.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

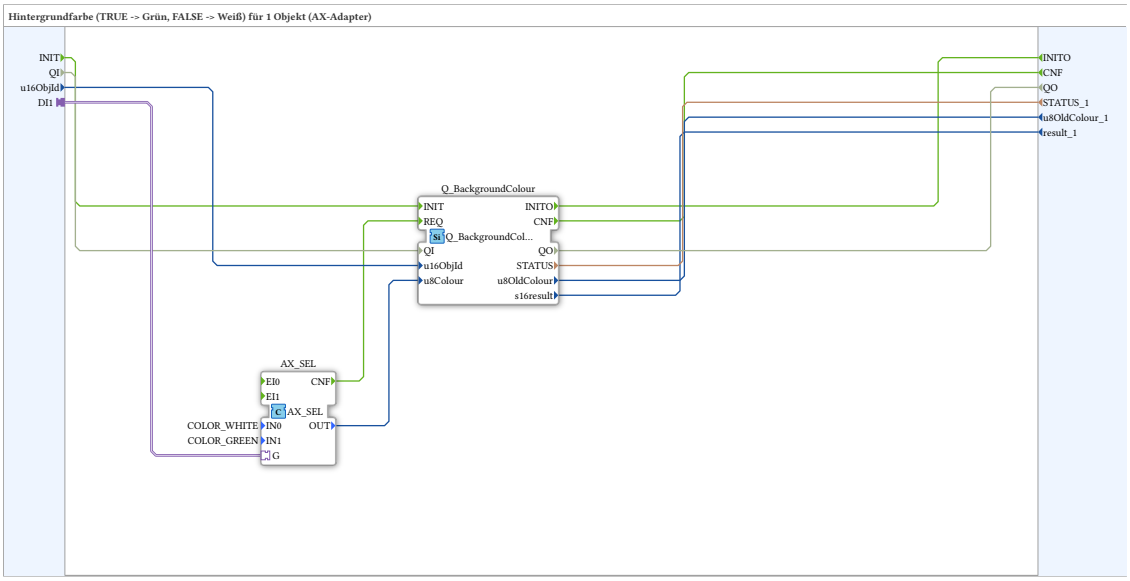
Zusammenfassung

Reine Weiterleitung an `GreenRedBackground4_AXS` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground1_AX



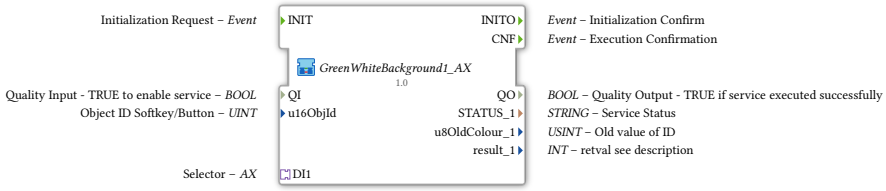
• – ○ • – ○ • – ○ •

Einleitung

`GreenWhiteBackground1_AX` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den Eingang `u16ObjId` übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Quality-Muster: Die SubApp besitzt den Ereigniseingang `INIT`, den Ereignisausgang `INITO`, den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`, „Quality Input – TRUE to enable service“) und den Datenausgang `QO` (`BOOL`, „Quality Output – TRUE if service executed successfully“). Sie werden direkt an `Q_BackgroundColour` durchgereicht: `INIT` ⟶ `Q_BackgroundColour.INIT`, `Q_BackgroundColour.INITO` ⟶ `INITO`, `QI` ⟶ `Q_BackgroundColour.QI`, `Q_BackgroundColour.QO` ⟶ `QO`.



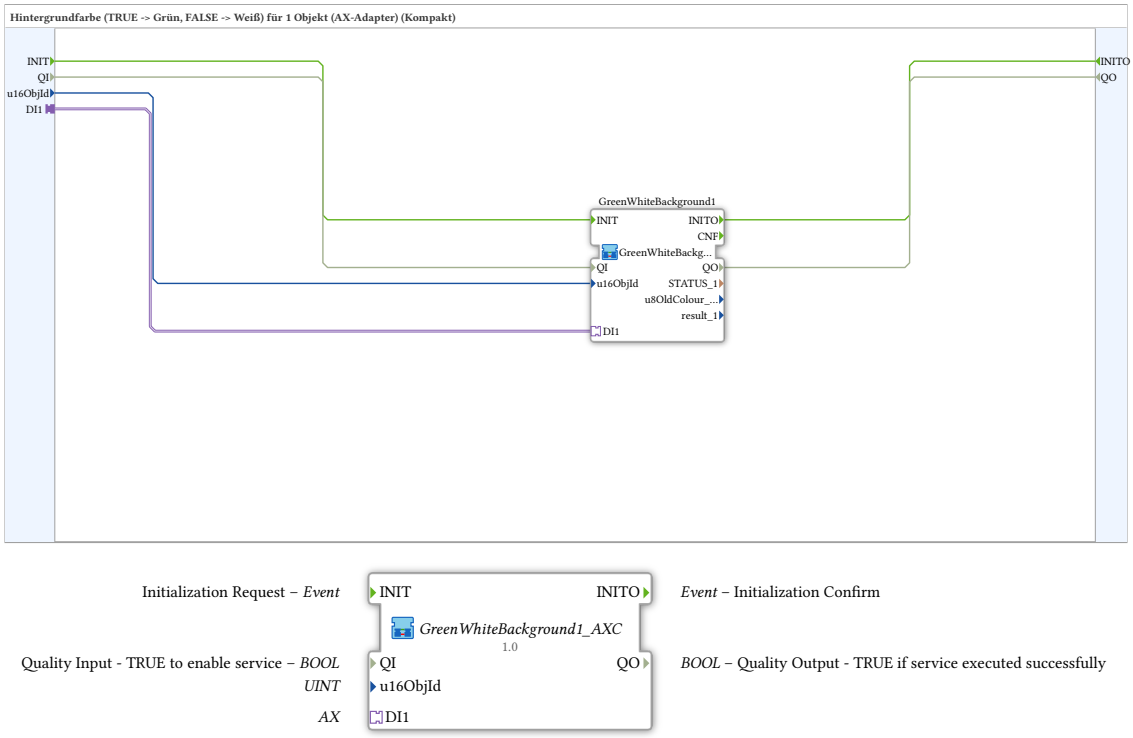
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Weiß, 1 Objekt, Adapter-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground1_AXC



• — ○ • — ○ • — ○ •

Einleitung

`GreenWhiteBackground1_AXC` ist ein **Kompakt-Wrapper** um `GreenWhiteBackground1_AX` — reicht `u16ObjId` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

Zusätzlich führt der Wrapper die Schnittstelle für Initialisierung und Qualität nach außen: Ereigniseingang `INIT`, Ereignisausgang `INITO`, Dateneingang `QI` (`BOOL`, Initialwert `TRUE`) und Datenausgang `QO` (`BOOL`). Sie sind 1:1 an die innere Instanz `GreenWhiteBackground1` durchgeschleift: `INIT` ⟶ `GreenWhiteBackground1.INIT`, `GreenWhiteBackground1.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground1.QI`, `GreenWhiteBackground1.QO` ⟶ `QO`.

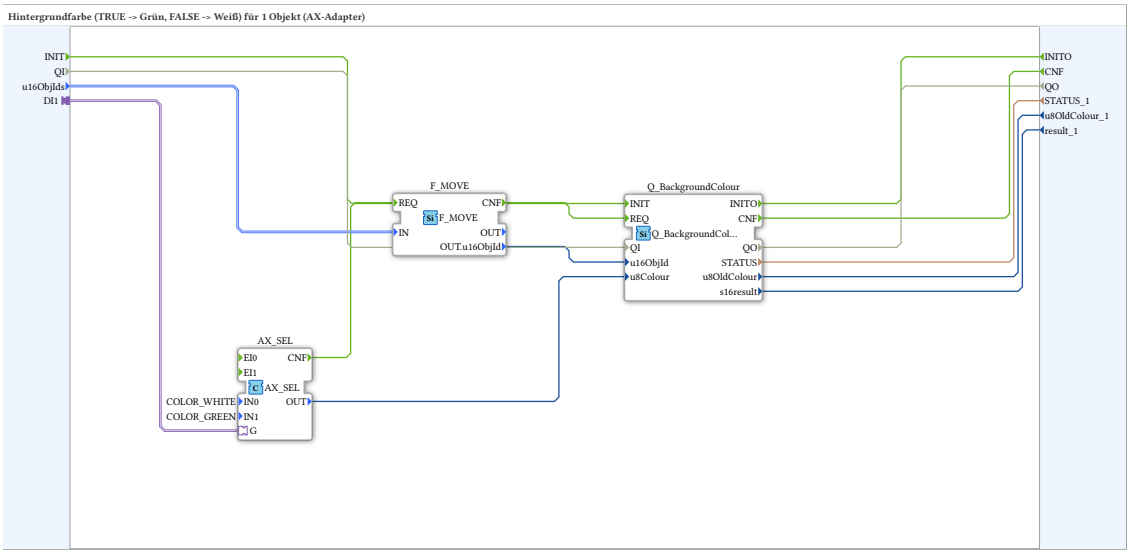
Zusammenfassung

Reine Weiterleitung an `GreenWhiteBackground1_AX` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground1_AXS



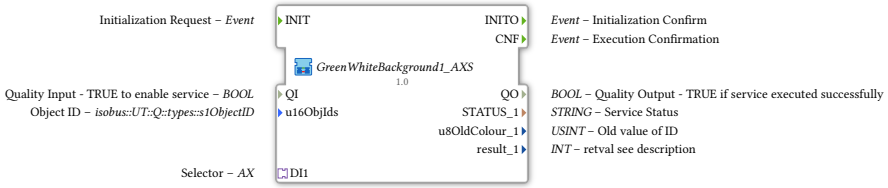
• – ○ • – ○ • – ○ •

Einleitung

`GreenWhiteBackground1_AXS` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Quality-Muster: Die SubApp besitzt den Ereigniseingang `INIT`, den Ereignisausgang `INITO`, den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`, „Quality Input – TRUE to enable service“) und den Datenausgang `QO` (`BOOL`, „Quality Output – TRUE if service executed successfully“). Sie werden direkt an `Q_BackgroundColour` durchgereicht: `INIT` ⟶ `Q_BackgroundColour.INIT`, `Q_BackgroundColour.INITO` ⟶ `INITO`, `QI` ⟶ `Q_BackgroundColour.QI`, `Q_BackgroundColour.QO` ⟶ `QO`.



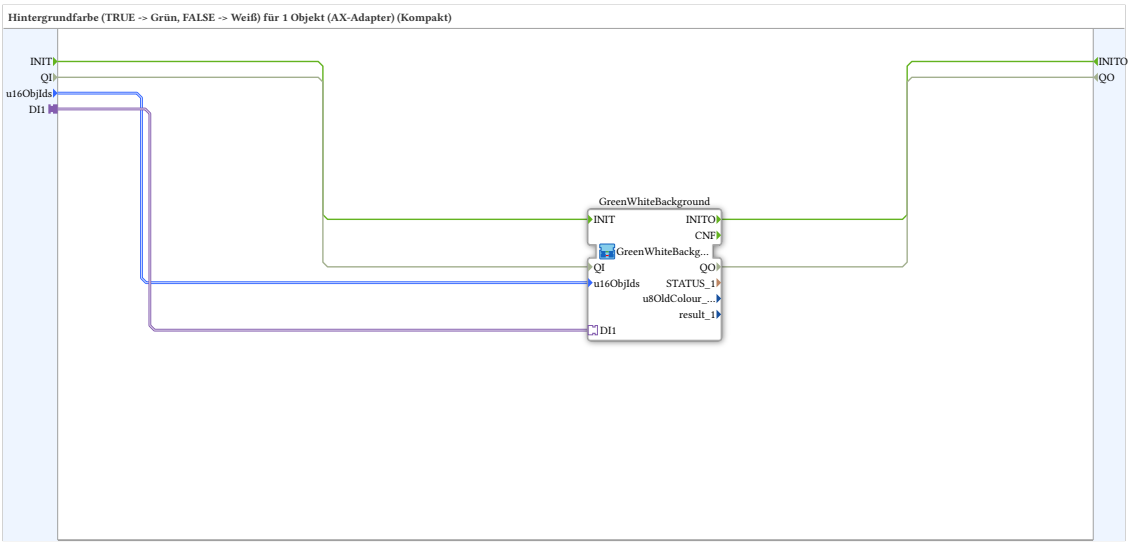
Zusammenfassung

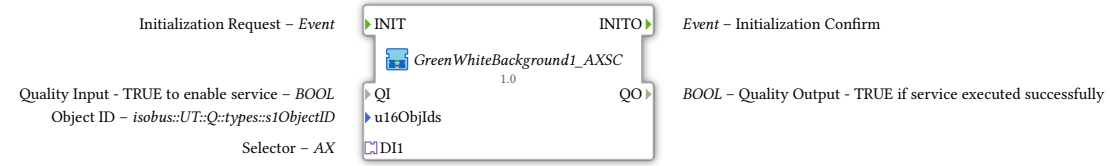
Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Weiß, 1 Objekt, Adapter-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground1_AXSC





• – ○ • – ○ • – ○ •

Einleitung

`GreenWhiteBackground1_AXSC` ist ein **Kompakt-Wrapper** um `GreenWhiteBackground1_AXS` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

Initialisierung und Qualität

- **Ereignis-Eingang `INIT`** (`EInit`) und **Ereignis-Ausgang `INITO`** (`EInit`) werden durchgereicht: `INIT` ⟶ `GreenWhiteBackground.INIT`; `GreenWhiteBackground.INITO` ⟶ `INITO`.
- **Dateneingang `QI`** (`BOOL`, Vorgabe `TRUE`) ⟶ `GreenWhiteBackground.QI`; **Datenausgang `QO`** (`BOOL`) ⟵ `GreenWhiteBackground.QO`.

Die innere SubApp-Instanz `GreenWhiteBackground` ist vom Typ `GreenWhiteBackground1_AXS`.

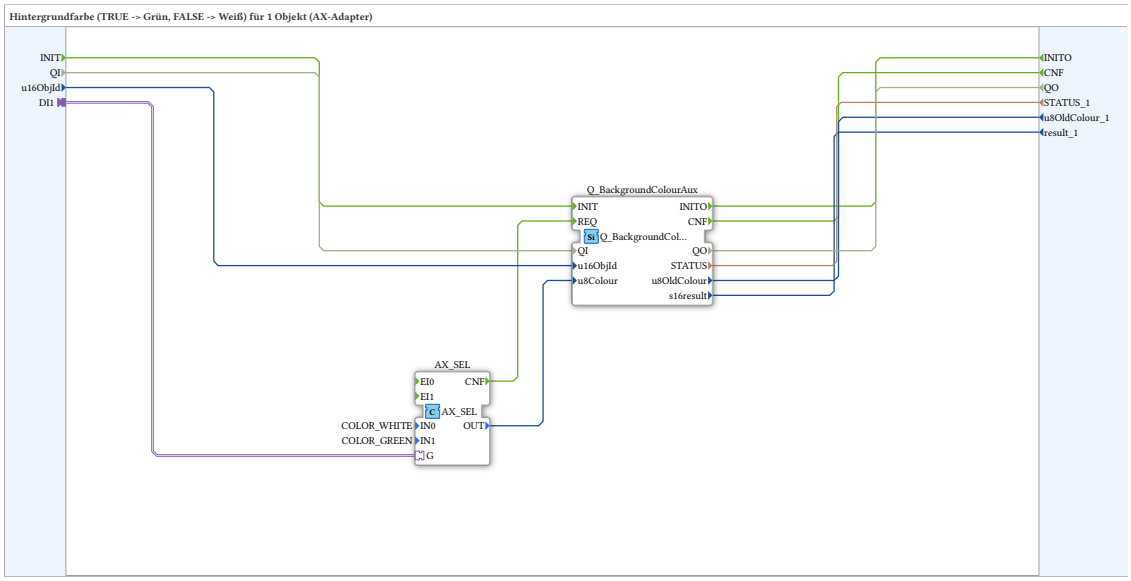
Zusammenfassung

Reine Weiterleitung an `GreenWhiteBackground1_AXS` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground1_aux_AX



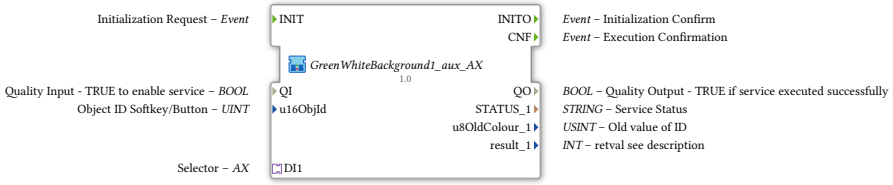
• — ○ • — ○ • — ○ •

Einleitung

`GreenWhiteBackground1_aux_AX` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den Eingang `u16ObjId` übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Quality-Muster: Die SubApp besitzt den Ereigniseingang `INIT`, den Ereignisausgang `INITO`, den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`, „Quality Input – TRUE to enable service“) und den Datenausgang `QO` (`BOOL`, „Quality Output – TRUE if service executed successfully“). Sie werden direkt an `Q_BackgroundColourAux` durchgereicht: `INIT` ⟶ `Q_BackgroundColourAux.INIT`, `Q_BackgroundColourAux.INITO` ⟶ `INITO`, `QI` ⟶ `Q_BackgroundColourAux.QI`, `Q_BackgroundColourAux.QO` ⟶ `QO`.



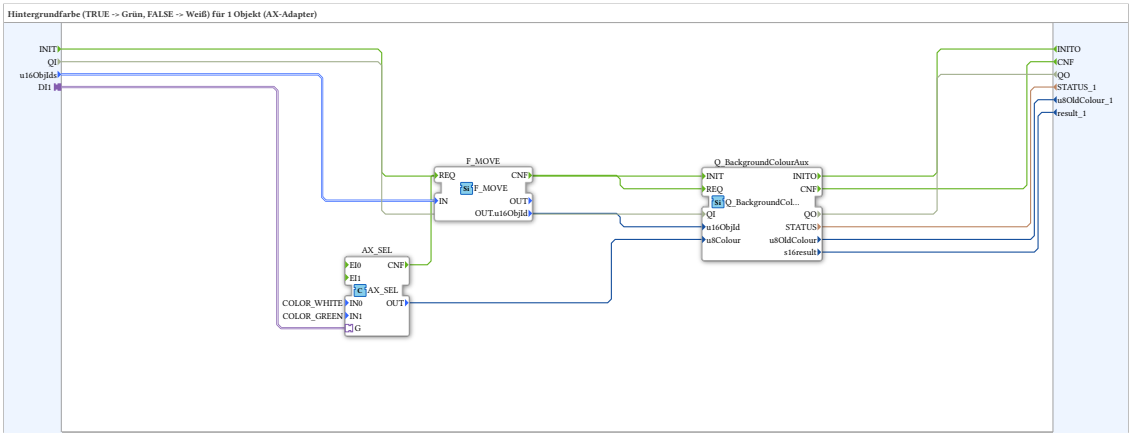
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Weiß, 1 Objekt, Adapter-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground1_aux_AXS



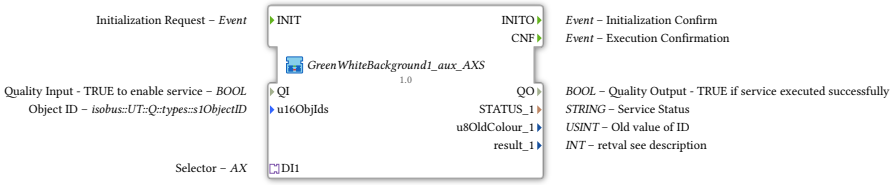
• – ○ • – ○ • – ○ •

Einleitung

`GreenWhiteBackground1_aux_AXS` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Quality-Muster: Die SubApp besitzt den Ereigniseingang `INIT`, den Ereignisausgang `INITO`, den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`, „Quality Input – TRUE to enable service“) und den Datenausgang `QO` (`BOOL`, „Quality Output – TRUE if service executed successfully“). Sie werden direkt an `Q_BackgroundColourAux` durchgereicht: `INIT` ⟶ `Q_BackgroundColourAux.INIT`, `Q_BackgroundColourAux.INITO` ⟶ `INITO`, `QI` ⟶ `Q_BackgroundColourAux.QI`, `Q_BackgroundColourAux.QO` ⟶ `QO`.



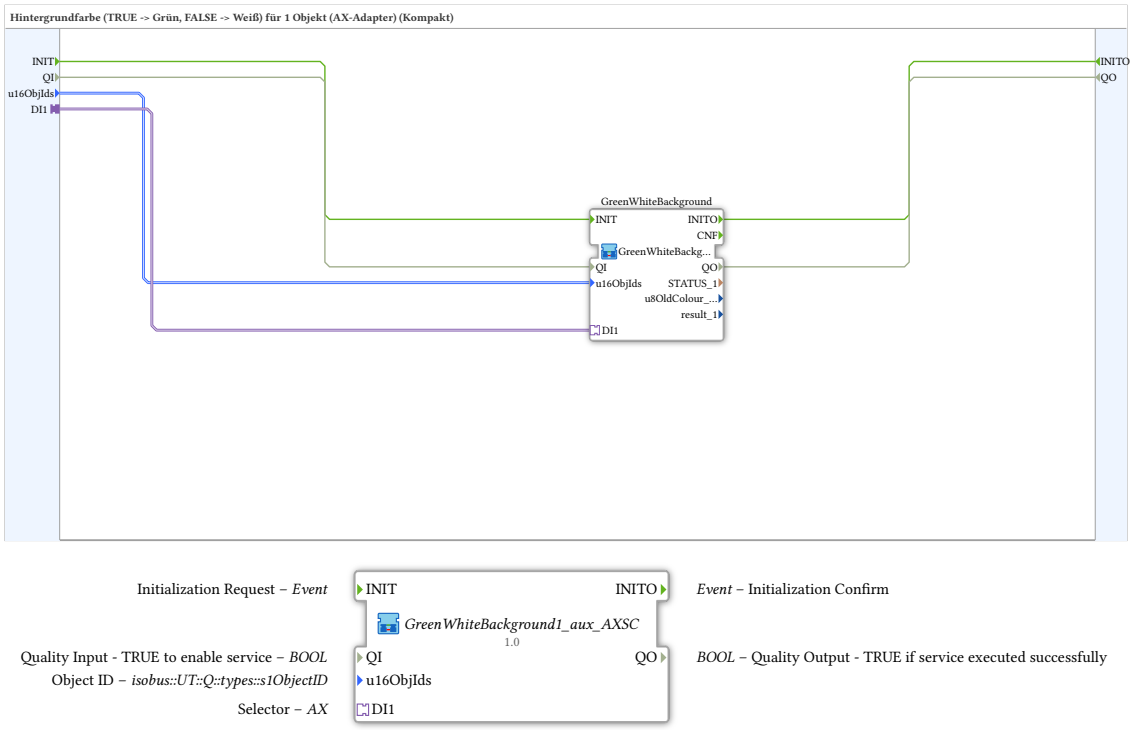
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Weiß, 1 Objekt, Adapter-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground1_aux_AXSC



• — ○ • — ○ • — ○ •

Einleitung

`GreenWhiteBackground1_aux_AXSC` ist ein **Kompakt-Wrapper** um `GreenWhiteBackground1_aux_AXS` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Schnittstelle (Quality-Muster): Der Wrapper besitzt zusätzlich den Ereignis-Eingang `INIT` und den Ereignis-Ausgang `INITO` (je `EInit`) sowie den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`) und den Datenausgang `QO` (`BOOL`). Sie sind direkt an die innere Instanz `GreenWhiteBackground` durchgeschleift: `INIT` ⟶ `GreenWhiteBackground.INIT`, `GreenWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground.QI`, `GreenWhiteBackground.QO` ⟶ `QO`.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

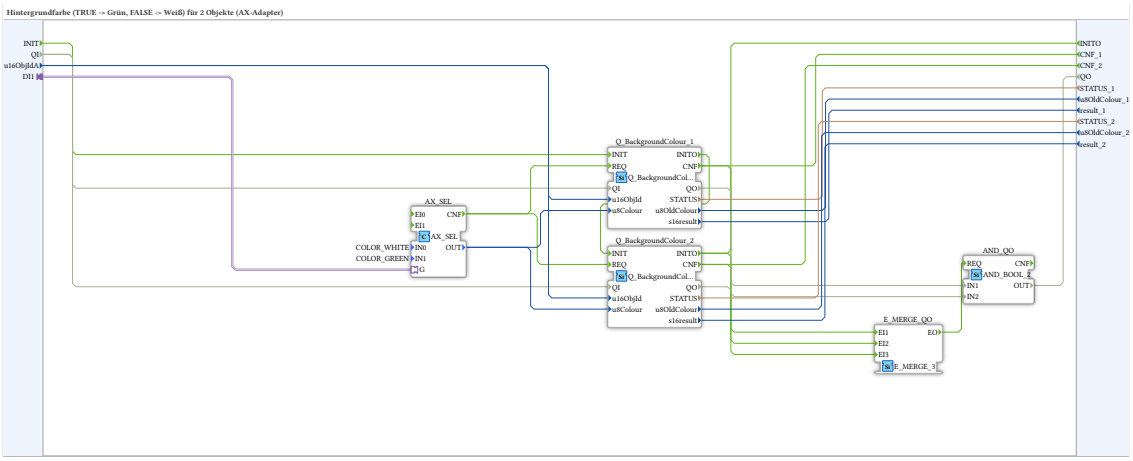
Zusammenfassung

Reine Weiterleitung an `GreenWhiteBackground1_aux_AXS` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground2_AX



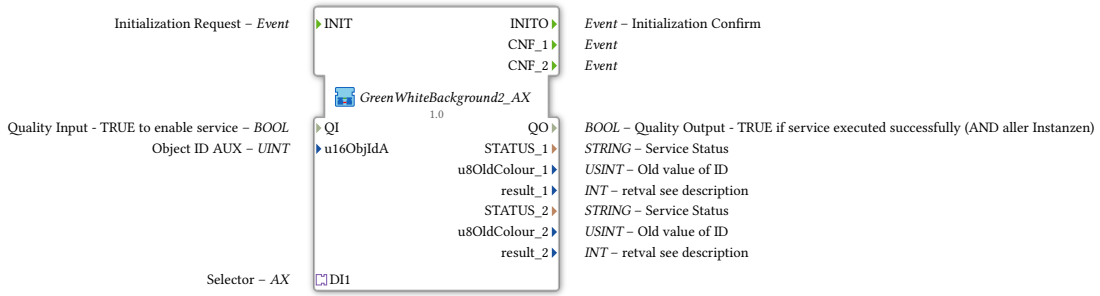
• – ○ • – ○ • – ○ •

Einleitung

`GreenWhiteBackground2_AX` schaltet die VT-Hintergrundfarbe von 2 Objekten anhand eines booleschen Selector-Signals: `TRUE` → **Grün**, `FALSE` → **Weiß**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über die Eingänge `u16ObjIdA` übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
2	`u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)

Allgemeines Muster (Selector → `AX_SEL/F_SEL` → `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Initialisierung und Status (INIT/INITO, QI/QO)

- **Ereignis-Eingang** `INIT` (`EInit`, Initialization Request) und **Ereignis-Ausgang** `INITO` (`EInit`, Initialization Confirm)
- **Daten-Eingang** `QI` (`BOOL`, Vorgabe `TRUE`, Quality Input) wird an `Q_BackgroundColour_1.QI` und `Q_BackgroundColour_2.QI` weitergegeben
- **Daten-Ausgang** `QO` (`BOOL`, Quality Output): UND (`AND_QO`, Typ `iec61131::booleanOperators::AND_BOOL_2`) der `QO`-Ausgänge von `Q_BackgroundColour_1.QO` und `Q_BackgroundColour_2.QO`
- **Initialisierungskette**: `INIT` → `Q_BackgroundColour_1.INIT`, `Q_BackgroundColour_1.INITO` → `Q_BackgroundColour_2.INIT`, `Q_BackgroundColour_2.INITO` → `INITO`
- `E_MERGE_QO` (Typ `iec61499::events::E_MERGE_3`) führt `Q_BackgroundColour_2.INITO`, `Q_BackgroundColour_1.CNF` und `Q_BackgroundColour_2.CNF` zusammen und löst `AND_QO.REQ` aus

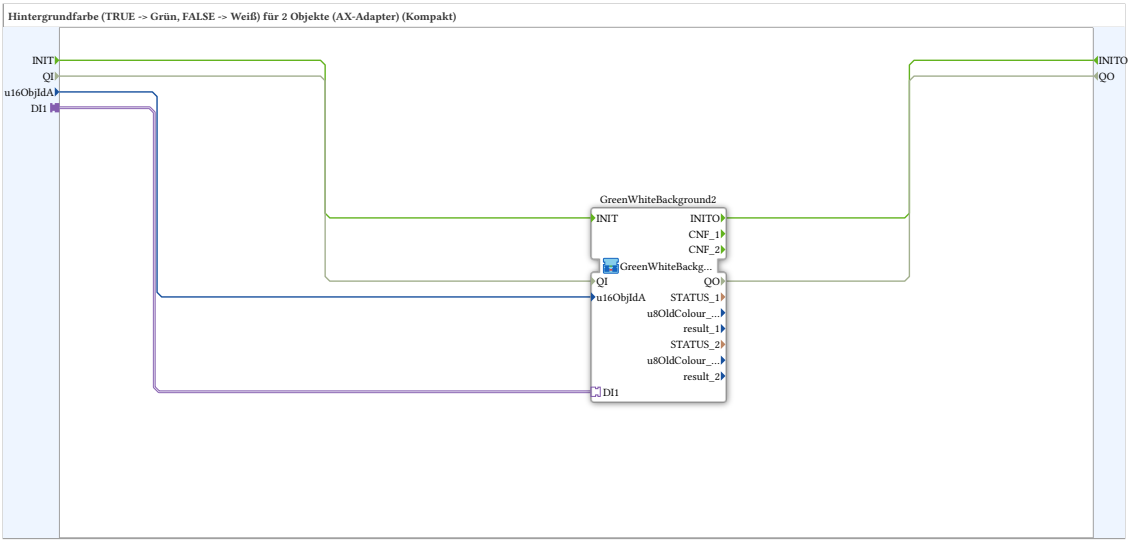
Zusammenfassung

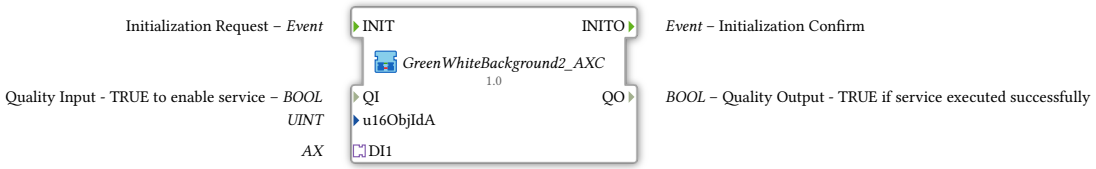
Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Weiß, 2 Objekte, Adapter-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground2_AXC





• – ○ • – ○ • – ○ •

Einleitung

`GreenWhiteBackground2_AXC` ist ein **Kompakt-Wrapper** um `GreenWhiteBackground2_AX` — reicht `u16ObjId` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen. Zusätzlich werden `INIT` ⟶ `GreenWhiteBackground2.INIT`, `GreenWhiteBackground2.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground2.QI` (Initialwert `TRUE`) und `GreenWhiteBackground2.QO` ⟶ `QO` durchgereicht.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

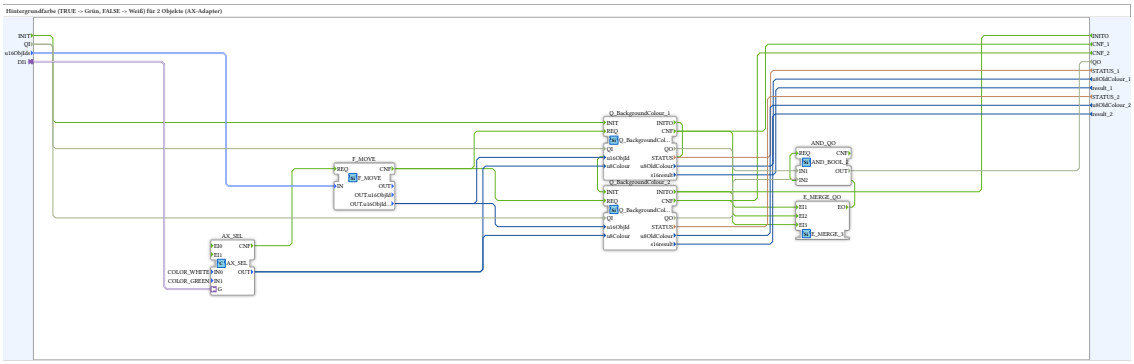
Zusammenfassung

Reine Weiterleitung an `GreenWhiteBackground2_AX` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground2_AXS



• – ○ • – ○ • – ○ •

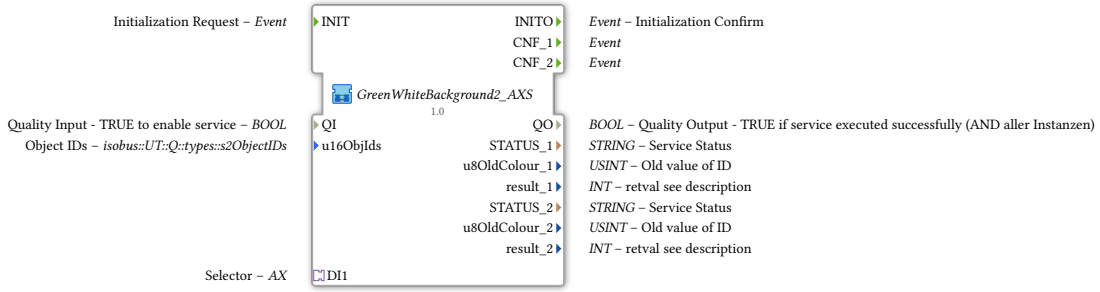
Einleitung

`GreenWhiteBackground2_AXS` schaltet die VT-Hintergrundfarbe von 2 Objekten anhand eines booleschen Selector-Signals: `TRUE` → **Grün**, `FALSE` → **Weiß**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)

Initialisierung und Qualität: Zusätzlich zu den bisherigen Ereignissen besitzt der Baustein den Ereigniseingang `INIT`, den Ereignisausgang `INITO` sowie `QI` (BOOL, Eingang, Initialwert `TRUE`) und `QO` (BOOL, Ausgang, UND aller Instanzen). Init-Kette: `INIT` → `Q_BackgroundColour_1` → `Q_BackgroundColour_2` → `INITO` (jeweils `INITO` → nächstes `INIT`). `QI` wird an alle `Q_BackgroundColour_n.QI` weitergereicht. `QO` bildet `AND_QO` (`iec61131::booleanOperators::AND_BOOL_2`): `Q_BackgroundColour_1.QO` → `AND_QO.IN1`, `Q_BackgroundColour_2.QO` → `AND_QO.IN2`, `AND_QO.OUT` → `QO`. `E_MERGE_QO` (`E_MERGE_3`) führt `Q_BackgroundColour_2.INITO` und die `CNF`-Ausgänge `Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF` zusammen; `E_MERGE_QO.EO` triggert `AND_QO.REQ`.

Allgemeines Muster (Selector → `AX_SEL`/`F_SEL` → `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



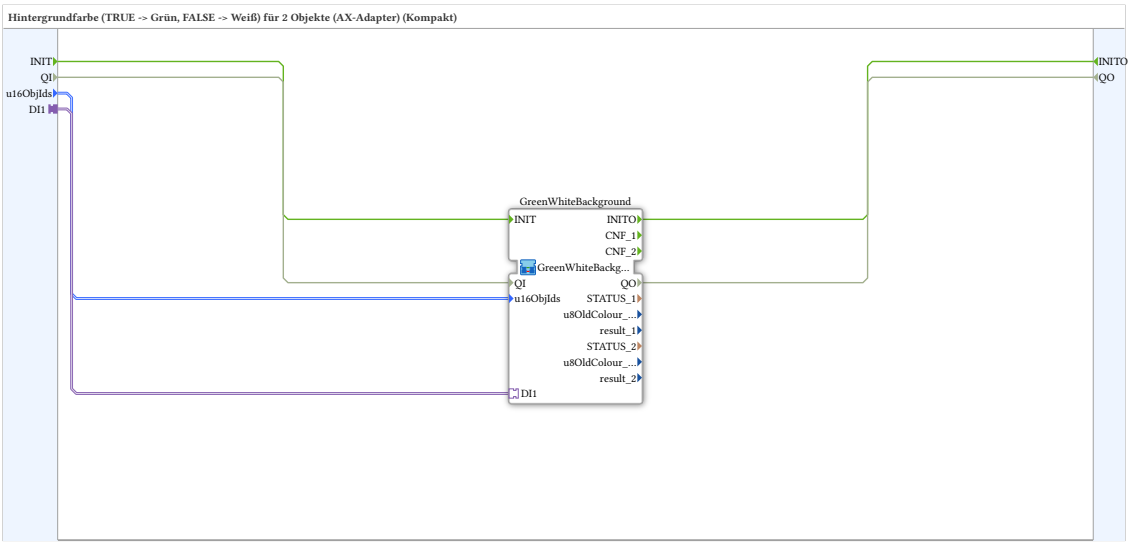
Zusammenfassung

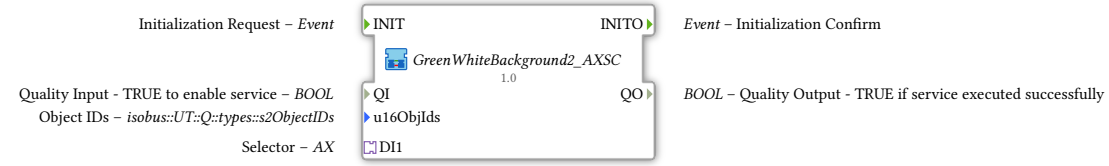
Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Weiß, 2 Objekte, Adapter-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground2_AXSC





• – ○ • – ○ • – ○ •

Einleitung

`GreenWhiteBackground2_AXSC` ist ein **Kompakt-Wrapper** um `GreenWhiteBackground2_AXS` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

Zusätzlich führt der Wrapper die Schnittstelle für Initialisierung und Qualität nach außen: Ereigniseingang `INIT`, Ereignisausgang `INITO`, Dateneingang `QI` (`BOOL`, Initialwert `TRUE`) und Datenausgang `QO` (`BOOL`). Sie sind 1:1 an die innere Instanz `GreenWhiteBackground` durchgeschleift: `INIT` ⟶ `GreenWhiteBackground.INIT`, `GreenWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground.QI`, `GreenWhiteBackground.QO` ⟶ `QO`.

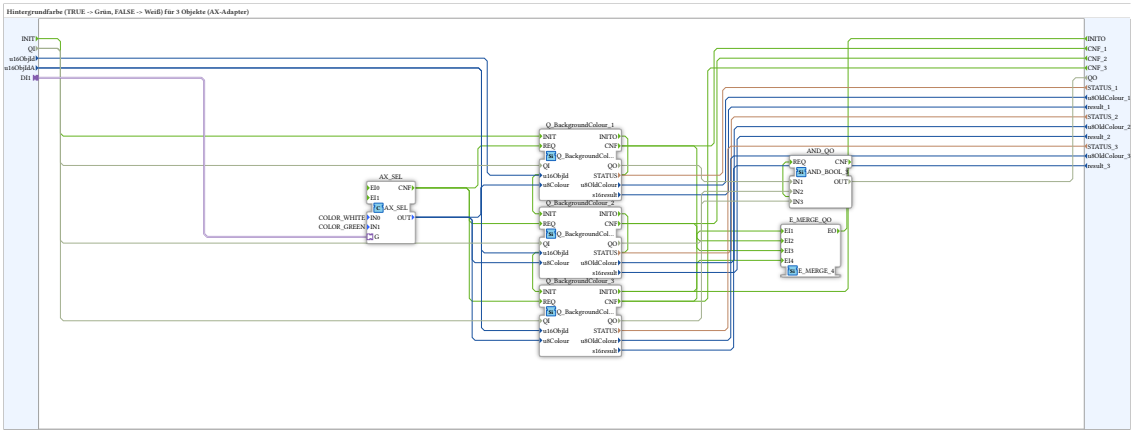
Zusammenfassung

Reine Weiterleitung an `GreenWhiteBackground2_AXS` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground3_AX



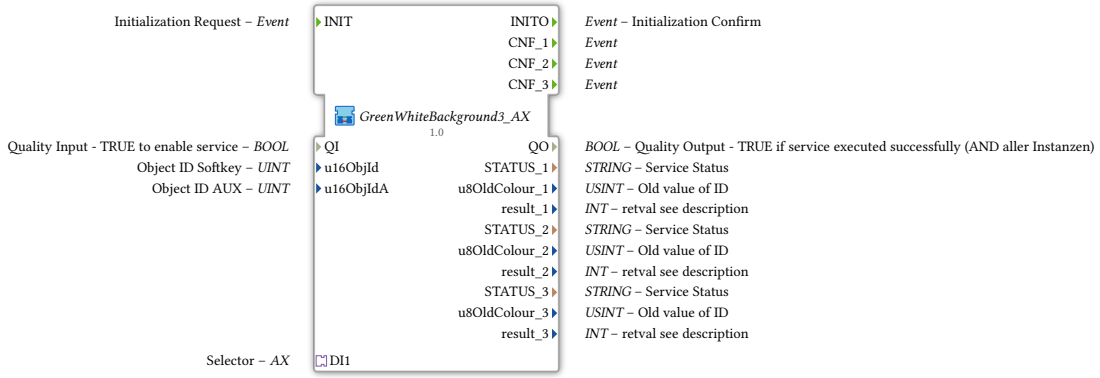
• – ○ • – ○ • – ○ •

Einleitung

`GreenWhiteBackground3_AX` schaltet die VT-Hintergrundfarbe von 3 Objekten anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über die Eingänge `u16ObjId, u16ObjIdA` übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`u16ObjId`	`Q_BackgroundColour` (normales Objekt)
2	`u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
3	`u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)

Allgemeines Muster (Selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Initialisierung und Qualität

- **Ereignis-Eingang `INIT`** (`EInit`) und **Ereignis-Ausgang `INITO`** (`EInit`): die Initialisierung wird durch die `Q_BackgroundColour`-Instanzen gekettet: `INIT` ⟶ `Q_BackgroundColour_1.INIT`; `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` ⟶ `Q_BackgroundColour_3.INIT`; `Q_BackgroundColour_3.INITO` ⟶ `INITO`.
- **Dateneingang `QI`** (`BOOL`, Vorgabe `TRUE`) wird an `.QI` aller Instanzen verteilt (`Q_BackgroundColour_1`, `Q_BackgroundColour_2`, `Q_BackgroundColour_3`).
- **Datenausgang `QO`** (`BOOL`): UND aller Instanzen über `AND_QO` (`iec61131::booleanOperators::AND_BOOL_3`): `Q_BackgroundColour_1.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN2`, `Q_BackgroundColour_3.QO` ⟶ `AND_QO.IN3`; `AND_QO.OUT` ⟶ `QO`.
- `E_MERGE_QO` (`iec61499::events::E_MERGE_4`) vereint `Q_BackgroundColour_3.INITO` und die `CNF`-Ereignisse aller drei Instanzen (`Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF`, `Q_BackgroundColour_3.CNF`) und löst damit `AND_QO.REQ` aus.

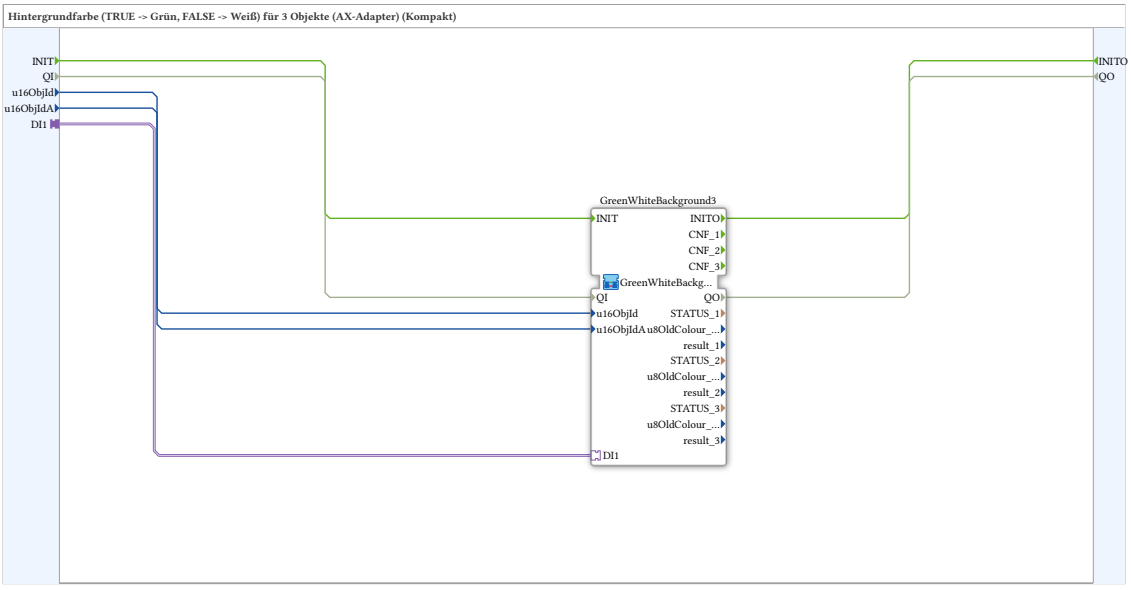
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Weiß, 3 Objekte, Adapter-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground3_AXC



• – ○ • – ○ • – ○ •

Einleitung

`GreenWhiteBackground3_AXC` ist ein **Kompakt-Wrapper** um `GreenWhiteBackground3_AX` — reicht `u16ObjId` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Schnittstelle (Quality-Muster): Der Wrapper besitzt zusätzlich den Ereignis-Eingang `INIT` und den Ereignis-Ausgang `INITO` (je `EInit`) sowie den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`) und den Datenausgang `QO` (`BOOL`). Sie sind direkt an die innere Instanz `GreenWhiteBackground3` durchgeschleift: `INIT` ⟶ `GreenWhiteBackground3.INIT`, `GreenWhiteBackground3.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground3.QI`, `GreenWhiteBackground3.QO` ⟶ `QO`.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

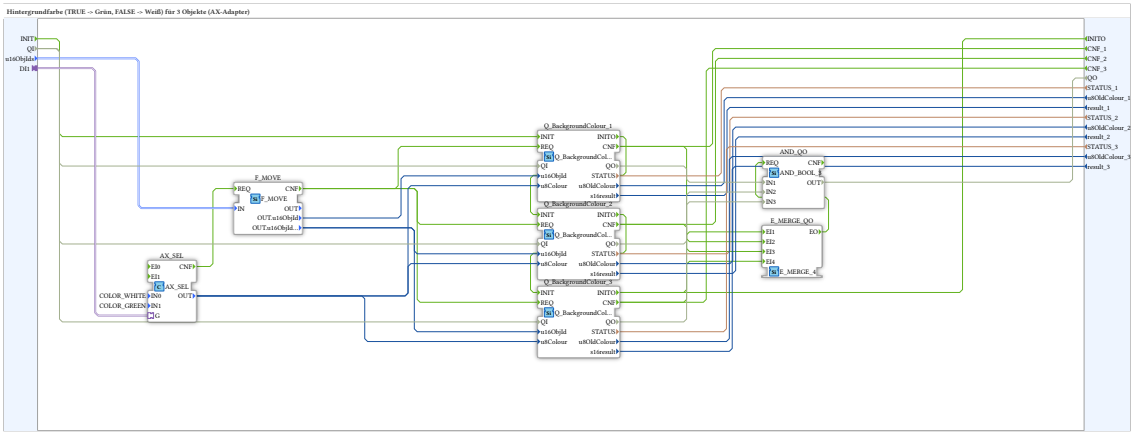
Zusammenfassung

Reine Weiterleitung an `GreenWhiteBackground3_AX` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground3_AXS



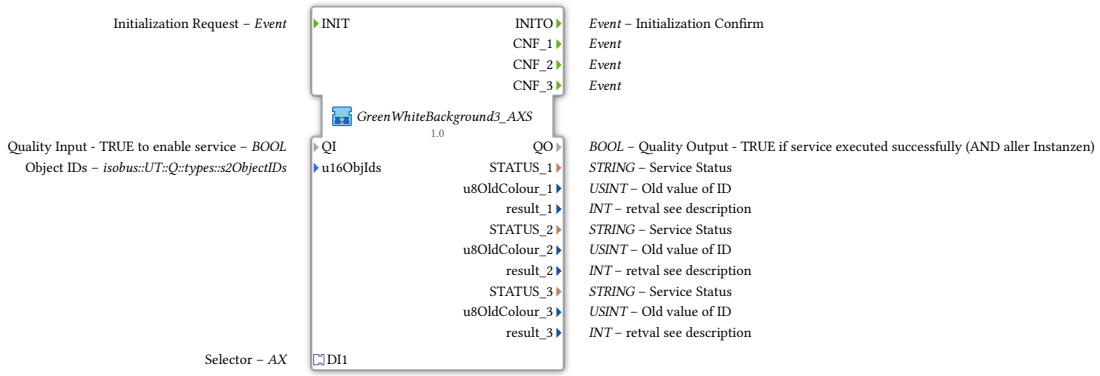
• – ○ • – ○ • – ○ •

Einleitung

`GreenWhiteBackground3_AXS` schaltet die VT-Hintergrundfarbe von 3 Objekten anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`F_MOVE.OUT.u16ObjId`	`Q_BackgroundColour` (normales Objekt)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
3	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Initialisierung und Status (INIT/INITO, QI/QO)

- **Ereignis-Eingang** `INIT` (`EInit`, Initialization Request) und **Ereignis-Ausgang** `INITO` (`EInit`, Initialization Confirm)
- **Daten-Eingang** `QI` (`BOOL`, Vorgabe `TRUE`, Quality Input) wird an `Q_BackgroundColour_1.QI`, `Q_BackgroundColour_2.QI` und `Q_BackgroundColour_3.QI` weitergegeben
- **Daten-Ausgang** `QO` (`BOOL`, Quality Output): UND (`AND_QO`, Typ `iec61131::booleanOperators::AND_BOOL_3`) der `QO`-Ausgänge von `Q_BackgroundColour_1.QO`, `Q_BackgroundColour_2.QO` und `Q_BackgroundColour_3.QO`
- **Initialisierungskette**: `INIT` ⟶ `Q_BackgroundColour_1.INIT`, `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`, `Q_BackgroundColour_2.INITO` ⟶ `Q_BackgroundColour_3.INIT`, `Q_BackgroundColour_3.INITO` ⟶ `INITO`
- `E_MERGE_QO` (Typ `iec61499::events::E_MERGE_4`) führt `Q_BackgroundColour_3.INITO`, `Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF` und `Q_BackgroundColour_3.CNF` zusammen und löst `AND_QO.REQ` aus

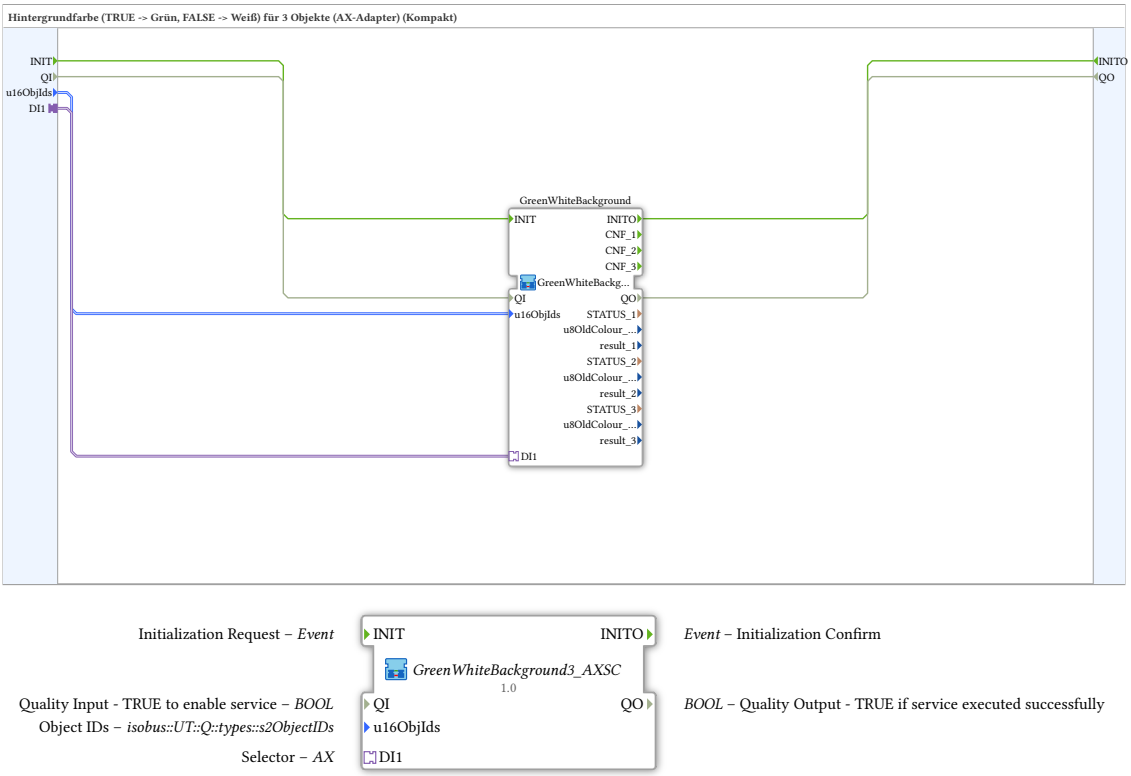
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Weiß, 3 Objekte, Adapter-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground3_AXSC



• – ○ • – ○ • – ○ •

Einleitung

`GreenWhiteBackground3_AXSC` ist ein **Kompakt-Wrapper** um `GreenWhiteBackground3_AXS` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen. Zusätzlich werden `INIT` ⟶ `GreenWhiteBackground.INIT`, `GreenWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground.QI` (Initialwert `TRUE`) und `GreenWhiteBackground.QO` ⟶ `QO` durchgereicht.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

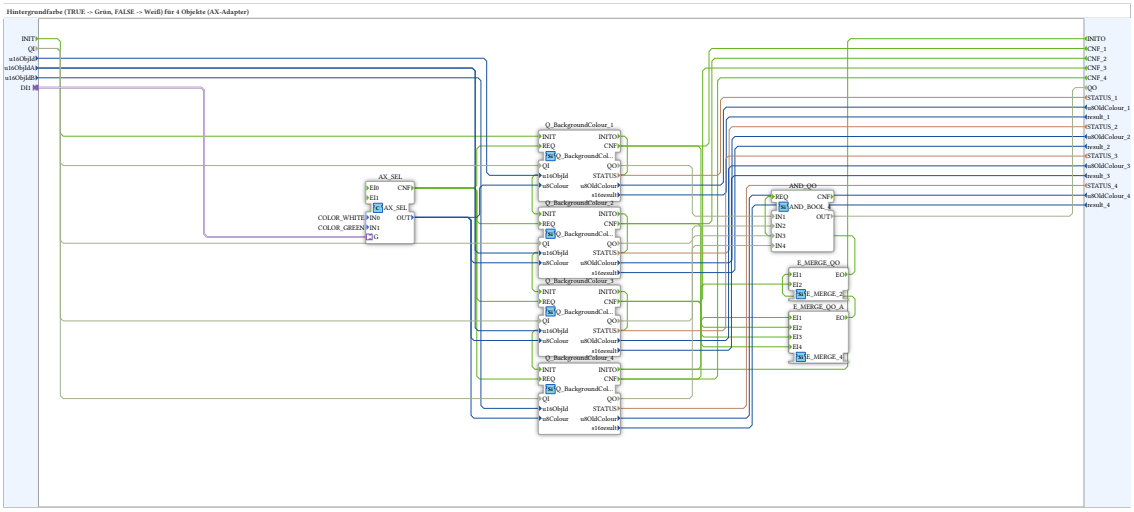
Zusammenfassung

Reine Weiterleitung an `GreenWhiteBackground3_AXS` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground4_AX



• – ○ • – ○ • – ○ •

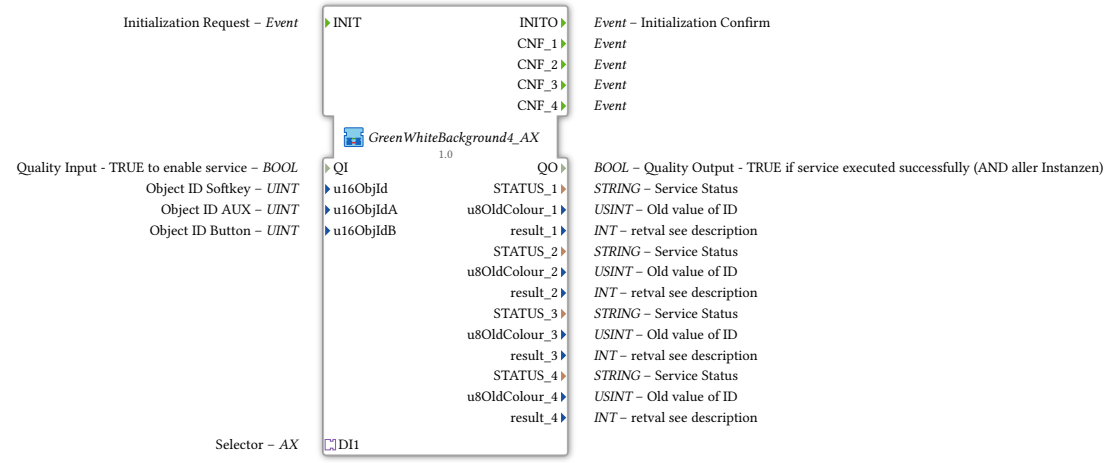
Einleitung

`GreenWhiteBackground4_AX` schaltet die VT-Hintergrundfarbe von 4 Objekten anhand eines booleschen Selector-Signals: `TRUE` → **Grün**, `FALSE` → **Weiß**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über die Eingänge `u16ObjId, u16ObjIdA, u16ObjIdB` übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`u16ObjId`	`Q_BackgroundColour` (normales Objekt)
2	`u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
3	`u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)
4	`u16ObjIdB`	`Q_BackgroundColour` (normales Objekt)

Initialisierung und Qualität: Zusätzlich zu den bisherigen Ereignissen besitzt der Baustein den Ereigniseingang `INIT`, den Ereignisausgang `INITO` sowie `QI` (BOOL, Eingang, Initialwert `TRUE`) und `QO` (BOOL, Ausgang, UND aller Instanzen). Init-Kette: `INIT` → `Q_BackgroundColour_1` → `Q_BackgroundColour_2` → `Q_BackgroundColour_3` → `Q_BackgroundColour_4` → `INITO` (jeweils `INITO` → nächstes `INIT`). `QI` wird an alle `Q_BackgroundColour_n.QI` weitergereicht. `QO` bildet `AND_QO` (`iec61131::booleanOperators::AND_BOOL_4`): `Q_BackgroundColour_1.QO` → `AND_QO.IN1`, `Q_BackgroundColour_2.QO` → `AND_QO.IN2`, `Q_BackgroundColour_3.QO` → `AND_QO.IN3`, `Q_BackgroundColour_4.QO` → `AND_QO.IN4`, `AND_QO.OUT` → `QO`. `E_MERGE_QO_A` (`E_MERGE_4`) führt `Q_BackgroundColour_4.INITO` und die `CNF`-Ausgänge von `Q_BackgroundColour_1`, `Q_BackgroundColour_2` und `Q_BackgroundColour_3` zusammen, `E_MERGE_QO` (`E_MERGE_2`) fügt `Q_BackgroundColour_4.CNF` hinzu; `E_MERGE_QO.EO` triggert `AND_QO.REQ`.

Allgemeines Muster (Selector → `AX_SEL`/`F_SEL` → `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



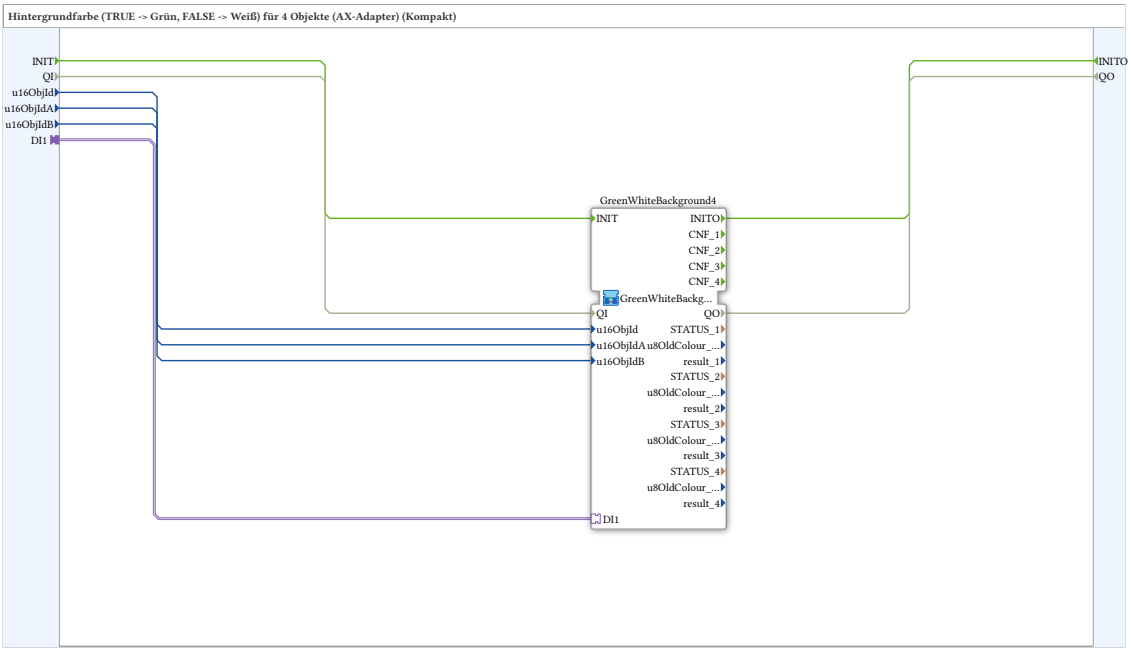
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Weiß, 4 Objekte, Adapter-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground4_AXC



• – ○ • – ○ • – ○ •

Einleitung

`GreenWhiteBackground4_AXC` ist ein **Kompakt-Wrapper** um `GreenWhiteBackground4_AX` — reicht `u16ObjId` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

Zusätzlich führt der Wrapper die Schnittstelle für Initialisierung und Qualität nach außen: Ereigniseingang `INIT`, Ereignisausgang `INITO`, Dateneingang `QI` (`BOOL`, Initialwert `TRUE`) und Datenausgang `QO` (`BOOL`). Sie sind 1:1 an die innere Instanz `GreenWhiteBackground4` durchgeschleift: `INIT` ⟶ `GreenWhiteBackground4.INIT`, `GreenWhiteBackground4.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground4.QI`, `GreenWhiteBackground4.QO` ⟶ `QO`.

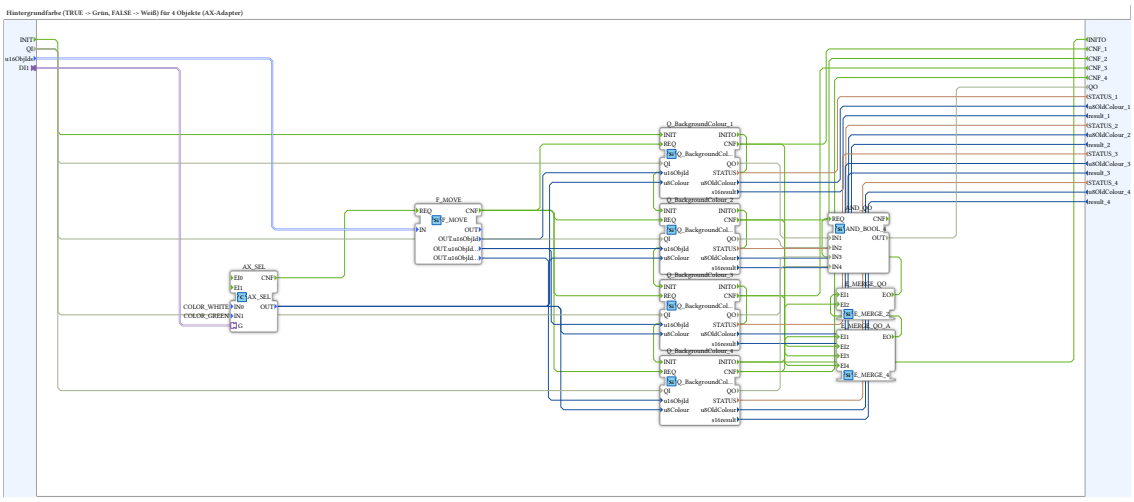
Zusammenfassung

Reine Weiterleitung an `GreenWhiteBackground4_AX` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground4_AXS



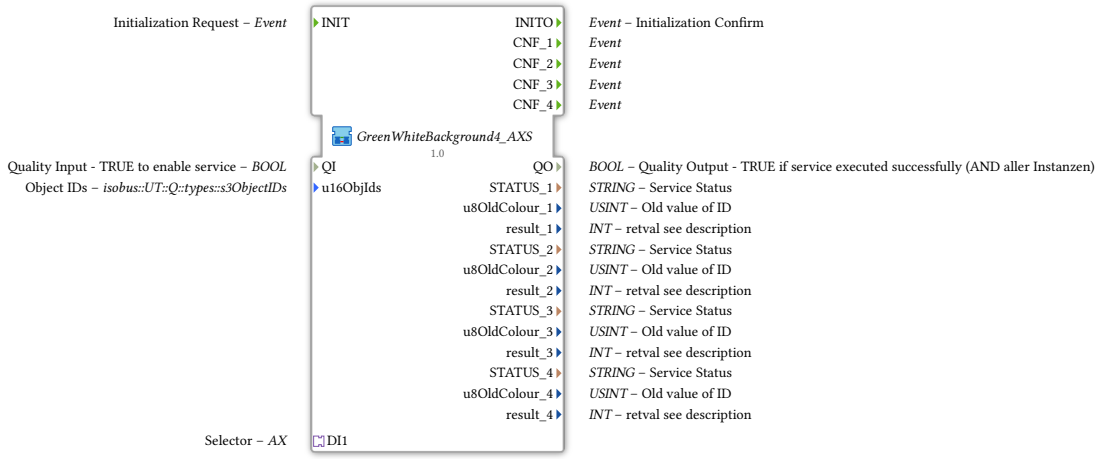
• – ○ • – ○ • – ○ •

Einleitung

`GreenWhiteBackground4_AXS` schaltet die VT-Hintergrundfarbe von 4 Objekten anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s3ObjectIDs` (`u16ObjIds`) übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`F_MOVE.OUT.u16ObjId`	`Q_BackgroundColour` (normales Objekt)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
3	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)
4	`F_MOVE.OUT.u16ObjIdB`	`Q_BackgroundColour` (normales Objekt)

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Initialisierung und Qualität

- **Ereignis-Eingang `INIT`** (`EInit`) und **Ereignis-Ausgang `INITO`** (`EInit`): die Initialisierung wird durch die `Q_BackgroundColour`-Instanzen gekettet: `INIT` ⟶ `Q_BackgroundColour_1.INIT`; `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` ⟶ `Q_BackgroundColour_3.INIT`; `Q_BackgroundColour_3.INITO` ⟶ `Q_BackgroundColour_4.INIT`; `Q_BackgroundColour_4.INITO` ⟶ `INITO`.
- **Dateneingang `QI`** (`BOOL`, Vorgabe `TRUE`) wird an `.QI` aller Instanzen verteilt (`Q_BackgroundColour_1`, `Q_BackgroundColour_2`, `Q_BackgroundColour_3`, `Q_BackgroundColour_4`).
- **Datenausgang `QO`** (`BOOL`): UND aller Instanzen über `AND_QO` (`iec61131::booleanOperators::AND_BOOL_4`): `Q_BackgroundColour_1.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN2`, `Q_BackgroundColour_3.QO` ⟶ `AND_QO.IN3`, `Q_BackgroundColour_4.QO` ⟶ `AND_QO.IN4`; `AND_QO.OUT` ⟶ `QO`.
- `E_MERGE_QO_A` (`iec61499::events::E_MERGE_4`) vereint `Q_BackgroundColour_4.INITO` und die `CNF`-Ereignisse der Instanzen 1 bis 3; `E_MERGE_QO` (`iec61499::events::E_MERGE_2`) vereint dessen Ausgang `E_MERGE_QO_A.EO` mit `Q_BackgroundColour_4.CNF` und löst damit `AND_QO.REQ` aus.

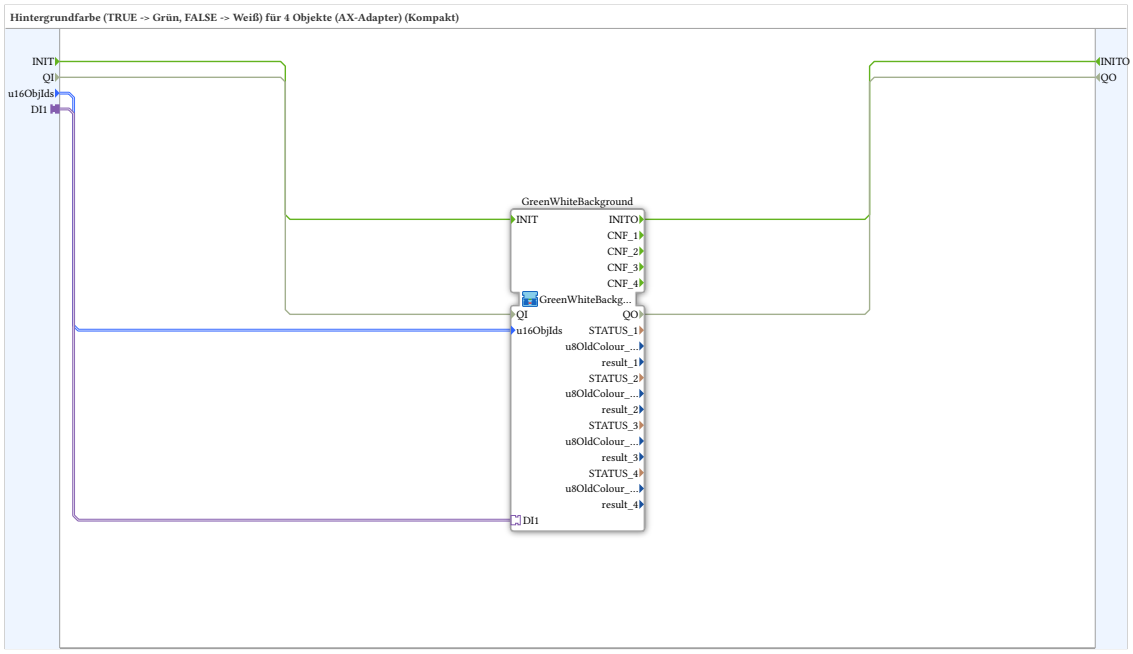
Zusammenfassung

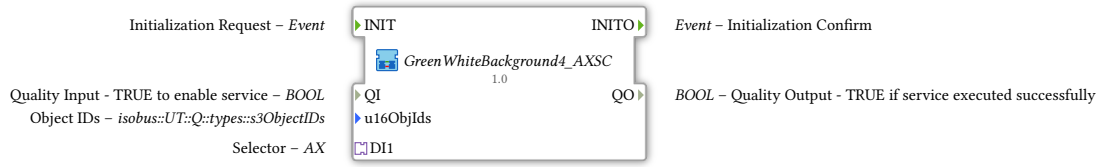
Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Weiß, 4 Objekte, Adapter-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground4_AXSC





• – ○ • – ○ • – ○ •

Einleitung

`GreenWhiteBackground4_AXSC` ist ein **Kompakt-Wrapper** um `GreenWhiteBackground4_AXS` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Schnittstelle (Quality-Muster): Der Wrapper besitzt zusätzlich den Ereignis-Eingang `INIT` und den Ereignis-Ausgang `INITO` (je `EInit`) sowie den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`) und den Datenausgang `QO` (`BOOL`). Sie sind direkt an die innere Instanz `GreenWhiteBackground` durchgeschleift: `INIT` ⟶ `GreenWhiteBackground.INIT`, `GreenWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground.QI`, `GreenWhiteBackground.QO` ⟶ `QO`.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

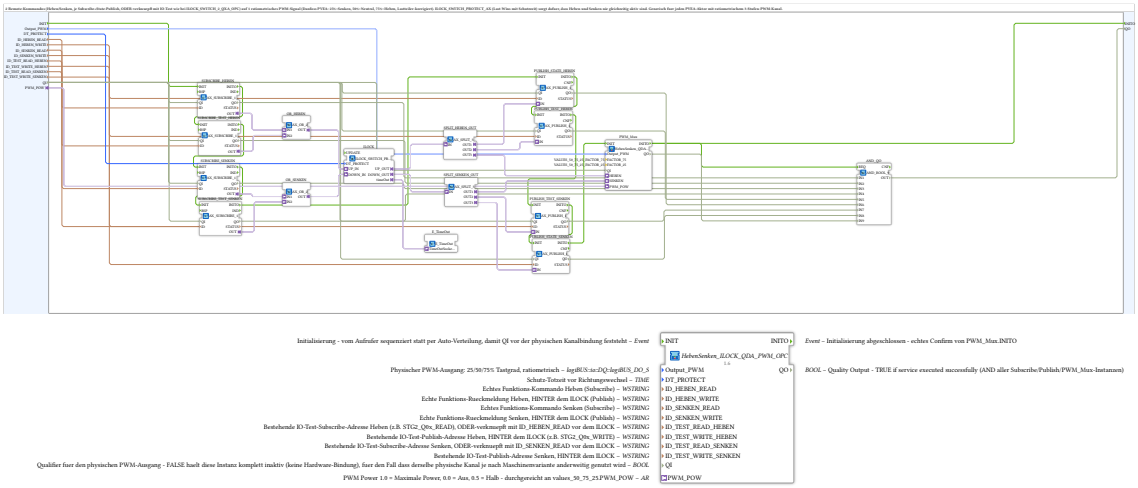
Zusammenfassung

Reine Weiterleitung an `GreenWhiteBackground4_AXS` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

HebenSenken_ILOCK_QDA_PWM_OPC



• – ○ • – ○ • – ○ •

Einleitung

Die SubApp `HebenSenken_ILOCK_QDA_PWM_OPC` realisiert die Ansteuerung eines ratiometrischen PWM-Ausgangs (z.B. für ein proportionales Wegeventil wie Danfoss PVEA) für zwei fernbedienbare Kommandos *Heben* und *Senken*. Sie integriert zwei OPC-UA-fähige Subscribe-/Publish-Paare für die Fernkommandos sowie zwei bestehende IO-Test-Kanäle und verknüpft diese logisch per ODER. Eine Interlock-Schaltung mit Schutzzeit (Last-Wins-Prinzip) verhindert, dass beide Richtungen gleichzeitig aktiv sind. Das resultierende 2-Bit-Signal wird über einen 3-Wege-Multiplexer in einen analogen Wert umgesetzt, der über einen digitalen PWM-Ausgang (QDA) ein proportionales Signal mit 25 % (Senken), 50 % (Neutral) oder 75 % (Heben) Tastgrad erzeugt. Eine Lastteiler-Korrektur und Skalierung sorgen für einen linearen Zusammenhang zwischen Kommando und physischem Ausgang.

Die SubApp ist generisch für jeden Aktor mit ratiometrischem 3-Stufen-PWM-Kanal einsetzbar.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierung; vom Aufrufer sequenziert statt per Auto-Verteilung, damit `QI` vor der physischen Kanalbindung feststeht; startet die Initialisierungskette `SUBSCRIBE_HEBEN` ⟶ `SUBSCRIBE_TEST_HEBEN` ⟶ `SUBSCRIBE_SENKEN` ⟶ `SUBSCRIBE_TEST_SENKEN` ⟶ `PUBLISH_STATE_HEBEN` ⟶ `PUBLISH_TEST_HEBEN` ⟶ `PUBLISH_TEST_SENKEN` ⟶ `PUBLISH_STATE_SENKEN` ⟶ `PWM_Mux` (`HebenSenken_QDA_PWM`)

Alle übrigen Ereignisse werden ausschließlich über die internen Adapter (Subscribe/Publish) mit den angeschlossenen Kommunikationskanälen ausgetauscht.

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierung abgeschlossen; echtes Confirm von `PWM_Mux.INITO` (Ende der Initialisierungskette)

Weitere Ereignis-Ausgänge sind an der SubApp-Schnittstelle nicht definiert. Die Rückmeldungen (State-Publish) werden über die ADAPTER-Schnittstellen der internen Publikations-FBs gesendet.

Daten-Eingänge

Name	Typ	Beschreibung
`QI`	`BOOL`	Qualifier für den physischen PWM-Ausgang (Default `TRUE`); FALSE hält die Instanz komplett inaktiv (keine Hardware-Bindung), falls derselbe physische Kanal je nach Maschinenvariante anderweitig genutzt wird;

		wird an alle Subscribe-/Publish-Bausteine (`SUBSCRIBE_*`, `PUBLISH_*`) und an `PWM_Mux.QI` durchgereicht.
`Output_PWM`	`logiBUS::io::DQ::logiBUS_DO_S`	Physischer PWM-Ausgang (logiBUS Digitalausgang mit PWM-Fähigkeit), wird auf 25/50/75 % Tastgrad gesetzt.
`DT_PROTECT`	`TIME`	Schutz-Totzeit vor Richtungswechsel (default: `T#300ms`). Verhindert schnelles Umschalten zwischen Heben und Senken.
`ID_HEBEN_READ`	`WSTRING`	Adresse für das echte Funktions-Kommando „Heben“ (Subscribe).
`ID_HEBEN_WRITE`	`WSTRING`	Adresse für die echte Funktions-Rückmeldung „Heben“ (Publish), gesendet hinter dem ILOCK.
`ID_SENKEN_READ`	`WSTRING`	Adresse für das echte Funktions-Kommando „Senken“ (Subscribe).
`ID_SENKEN_WRITE`	`WSTRING`	Adresse für die echte Funktions-Rückmeldung „Senken“ (Publish), gesendet hinter dem ILOCK.
`ID_TEST_READ_HEBEN`	`WSTRING`	Bestehende IO-Test-Subscribe-Adresse für Heben (ODER-verknüpft mit `ID_HEBEN_READ`).
`ID_TEST_WRITE_HEBEN`	`WSTRING`	Bestehende IO-Test-Publish-Adresse für Heben (gesendet hinter dem ILOCK).
`ID_TEST_READ_SENKEN`	`WSTRING`	Bestehende IO-Test-Subscribe-Adresse für Senken (ODER-verknüpft mit `ID_SENKEN_READ`).
`ID_TEST_WRITE_SENKEN`	`WSTRING`	Bestehende IO-Test-Publish-Adresse für Senken (gesendet hinter dem ILOCK).

Daten-Ausgänge

Name	Typ	Beschreibung
`QO`	`BOOL`	Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde; UND (`AND_QO`, Typ `iec61131::booleanOperators::AND_BOOL_9`) der `QO` aller vier `SUBSCRIBE_*`, aller vier `PUBLISH_*` und von `PWM_Mux`; ausgelöst durch `PWM_Mux.INITO` (`REQ`).

Die Ausgangswerte werden über die internen Publisher und den physischen PWM-Ausgang `Output_PWM` bereitgestellt.

Adapter

Die SubApp besitzt keine öffentlichen Adapter-Schnittstellen. Alle Kommunikation erfolgt über die konfigurierbaren OPC-UA-Adressen (`WSTRING`-Parameter) und den logiBUS-Ausgang.

Funktionsweise

1. **Kommando-Empfang:** Zwei unabhängige Subscribe-Dienste (`SUBSCRIBE_HEBEN`, `SUBSCRIBE_SENKEN`) empfangen die echten Funktionskommandos von der OPC-UA-Schnittstelle. Parallel dazu empfangen zwei weitere Subscribe-Dienste (`SUBSCRIBE_TEST_HEBEN`, `SUBSCRIBE_TEST_SENKEN`) die IO-Test-Signale.

2. **ODER-Verknüpfung:** Die empfangenen BOOL-Werte werden je Richtung über ein ODER-Gatter (`OR_HEBEN`, `OR_SENKEN`) zusammengeführt. Somit kann ein Befehl entweder von der Fernsteuerung oder vom IO-Test kommen.
3. **Interlock:** Der ILOCK-Baustein (`ILOCK_SWITCH_PROTECT_AX`) nimmt die beiden ODER-Ausgänge als `UP_IN` und `DOWN_IN`. Er stellt sicher, dass nie beide Signale gleichzeitig `TRUE` sind (Last-Wins-Prinzip). Zusätzlich schützt eine Totzeit (`DT_PROTECT`) vor schnellen Richtungswechseln.
4. **Signalverteilung:** Die Ausgänge des ILOCK (`UP_OUT`, `DOWN_OUT`) werden über Split-Bausteine (`SPLIT_HEBEN_OUT`, `SPLIT_SENKEN_OUT`) auf drei Pfade verteilt:
   - **Pfad 1:** Rückmeldung an den jeweiligen Publish-Dienst für den echten Funktions-State.
   - **Pfad 2:** Rückmeldung an den Publish-Dienst für den IO-Test-State.
   - **Pfad 3:** Zusammenführung der beiden Bits (`ASSEMBLE_AB_FROM_AX`) zu einem 2-Bit-Wort (Bit0 = Heben, Bit1 = Senken).
5. **Adress-Umsetzung:** Das 2-Bit-Wort wird über den Konverter `AB_TO_AUI` in einen analogen Integer-Wert (0, 1, 2, 3) umgewandelt und dient als Auswahlwert für den 3-Wege-Multiplexer `AR_AUI_MUX_3`.
6. **Wertauswahl:** Die SubApp `values_50_75_25` liefert drei Konstanten: 50 % (Neutral), 75 % (Heben) und 25 % (Senken). Diese werden entsprechend dem Multiplexer-Ausgang gewählt.
7. **Korrektur und Skalierung:** Der ausgewählte Wert wird mit zwei Multiplikationsstufen verarbeitet:
   - **Korrekturfaktor:** `FACTOR = 1.17619` (entspricht 1/0.8502) – kompensiert die Nichtlinearität der PVEA-Ansteuerung.
   - **Skalierung auf 13-Bit-PWM:** `FACTOR = 81.91` (8191/100) – skaliert den Prozentwert auf den vollen Wertebereich des QDA-Ausgangs.
8. **Digitaler PWM-Ausgang:** Der resultierende Analogwert wird über `AR_TO_AD_NUM` in einen numerischen Wert für den `logiBUS_QDA_PWM`-Baustein gewandelt, der den physischen Ausgang `Output_PWM` entsprechend setzt. `QI` der SubApp wird an alle Subscribe-/Publish-Bausteine und an `PWM_Mux` durchgereicht. `INIT` durchläuft die Initialisierungskette `SUBSCRIBE_HEBEN` → `SUBSCRIBE_TEST_HEBEN` → `SUBSCRIBE_SENKEN` → `SUBSCRIBE_TEST_SENKEN` → `PUBLISH_STATE_HEBEN` → `PUBLISH_TEST_HEBEN` → `PUBLISH_TEST_SENKEN` → `PUBLISH_STATE_SENKEN` → `PWM_Mux` (`HebenSenken_QDA_PWM`); `PWM_Mux.INITO` wird als `INITO` ausgegeben und löst zugleich `AND_QO.REQ` aus. `QO` ist die UND-Verknüpfung (`AND_QO`) der neun `QO`-Ausgänge.

Technische Besonderheiten

- **Interlock mit Schutzzeit:** Der Baustein `ILOCK_SWITCH_PROTECT_AX` implementiert ein Last-Wins-Verhalten. Bei simultanen Kommandos gewinnt das zuletzt ankommende Signal, und nach einer Änderung wird eine einstellbare Totzeit `DT_PROTECT` eingehalten, um mechanische Belastungen zu reduzieren.
- **ODER-Verknüpfung von Fern- und Testbefehlen:** Die Trennung der echten Kommandos von den IO-Test-Pfaden ermöglicht eine sichere Inbetriebnahme und Wartung, ohne die Fernsteuerung zu beeinträchtigen.
- **Ratiometrische PWM-Steuerung:** Die Ausgangswerte (25 %, 50 %, 75 %) entsprechen den drei Zuständen eines proportionalen Wegeventils (z.B. Danfoss PVEA). Durch die Korrektur wird ein linearer Zusammenhang zwischen Kommando und hydraulischer Bewegung hergestellt.
- **Verwendung eines 3-Wege-Multiplexers:** Da der ILOCK den Zustand „beide aktiv“ (K=3) verhindert, genügt ein 3-Wege-Mux; der vierte Fall (beide Bits gesetzt) kann nicht auftreten.
- **Wiederverwendbarkeit:** Die SubApp ist aus einem größeren System ausgelagert und damit unabhängig einsetzbar; die Freigabe-Logik (DO) wurde entfernt, um die PWM-Ansteuerung separat verwenden zu können.

Zustandsübersicht

Die SubApp besitzt durch den ILOCK definierte Zustände für die beiden Richtungen:

Zustand	Heben (`UP`)	Senken (`DOWN`)	PWM-Tastgrad	Bedeutung
Neutral	0	0	50 %	Kein Kommando, Aktor in Mittelstellung
Heben aktiv	1	0	75 %	Aktor fährt aus (Heben)
Senken aktiv	0	1	25 %	Aktor fährt ein (Senken)

Unzulässig	1	1	–	Durch ILOCK verhindert

Während der Schutzzeit (`DT_PROTECT`) nach einem Richtungswechsel wird der alte Zustand gehalten, bis die Totzeit abgelaufen ist.

Anwendungsszenarien

- **Hydraulische Steuerung mit proportionalem Wegeventil:** Ansteuerung eines PVEA-Ventils für Hub-/Senkbewegungen (z.B. an Kränen, Bühnen, Landmaschinen).
- **Fernbedienung über OPC-UA:** Integration in Leitsysteme oder SCADA, bei denen Heben/Senken über dedizierte OPC-UA-Nachrichten erfolgt.
- **Inbetriebnahme und Test:** Nutzung der IO-Test-Kanäle, um ohne Fernsteuerungssignal den Aktor direkt anzusteuern und die Verdrahtung zu prüfen.
- **Wiederverwendbare Bausteinbibliothek:** Als Teil einer Bibliothek für logiBUS-basierte Steuerungen, die mehrere Aktoren mit gleicher PWM-Charakteristik ansteuern.

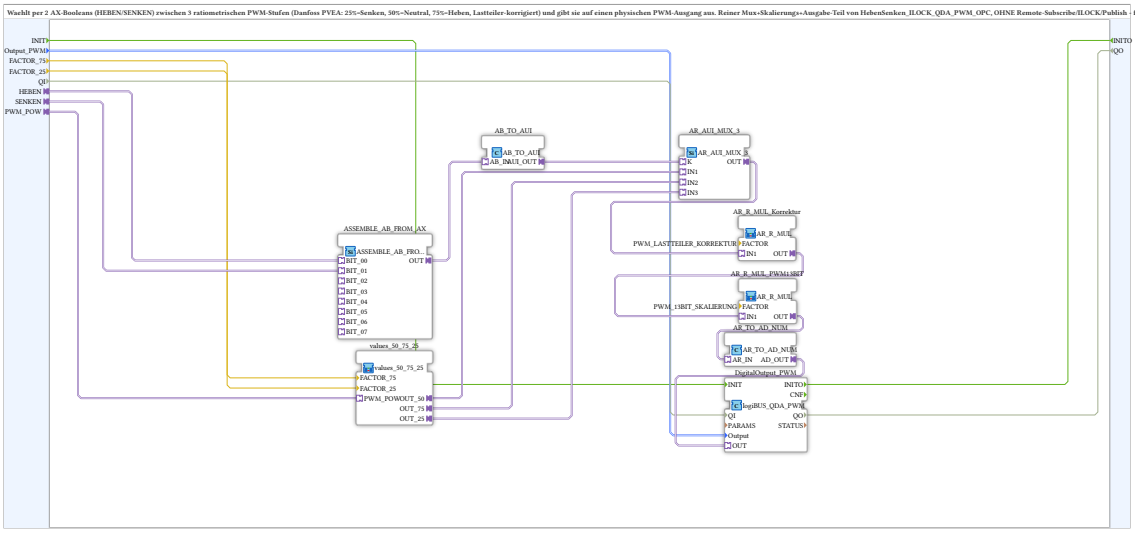
Vergleich mit ähnlichen Bausteinen

- **`ILOCK_SWITCH_2_QXA_OPC`:** Ein Vorgänger, der ohne PWM-Ausgang nur zwei digitale Ausgänge (QXA) ansteuert. Die vorliegende SubApp erweitert dies um eine ratiometrische PWM-Ausgabe.
- **Direkte PWM-Ansteuerung ohne Interlock:** Einfache Bausteine, die Heben/Senken direkt als analoge Werte setzen, riskieren bei gleichzeitigen Kommandos einen Kurzschluss oder mechanische Schäden. Durch den ILOCK wird dies sicher verhindert.
- **Bausteine mit 4-Wege-Multiplexer:** Manche Implementierungen benötigen einen 4-Wege-Mux, da sie den Fall „beide aktiv“ zulassen; hier wird durch die Interlock-Logik ein 3-Wege-Mux verwendet, was Ressourcen spart.

Fazit

Die SubApp `HebenSenken_ILOCK_QDA_PWM_OPC` bietet eine robuste und flexible Lösung zur Ansteuerung eines proportionalen PWM-Ausgangs für Heben/Senken. Durch die Kombination von OPC-UA-Kommunikation, Interlock-Schutz, ODER-Verknüpfung mit Testsignalen und sorgfältiger Skalierung wird eine zuverlässige und wartungsfreundliche Funktionalität erreicht. Sie ist ideal für den Einsatz in industriellen Steuerungen, bei denen Sicherheit und Präzision erforderlich sind. Die klare Trennung von Fern- und Testkommandos erleichtert Inbetriebnahme und Diagnose erheblich und macht den Baustein zu einem wertvollen Bestandteil einer Automatisierungsbibliothek.

HebenSenken_QDA_PWM

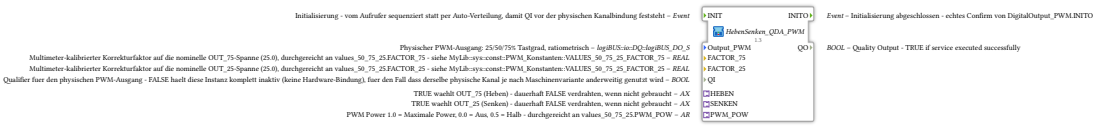


• – ○ • – ○ • – ○ •

Einleitung

`HebenSenken_QDA_PWM` wählt über zwei AX-Booleans (`HEBEN`/`SENKEN`) zwischen drei ratiometrischen PWM-Stufen (Danfoss PVEA: 25 % = Senken, 50 % = Neutral, 75 % = Heben, Lastteiler-korrigiert) und gibt die gewählte Stufe auf einen physischen PWM-Ausgang aus. Er ist der reine Mux-, Skalierungs-

und Ausgabe-Teil von `HebenSenken_ILOCK_QDA_PWM_OPC`, ohne Remote-Subscribe, ILOCK und Publish. Er eignet sich für Aufrufer, die `HEBEN`/`SENKEN` bereits anderweitig liefern (z. B. schon gegatet/exklusiv oder mit nur einer genutzten Richtung).



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierung; vom Aufrufer sequenziert statt per Auto-Verteilung, damit `QI` vor der physischen Kanalbindung feststeht; wird an `DigitalOutput_PWM.INIT` durchgereicht

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierung abgeschlossen; echtes Confirm von `DigitalOutput_PWM.INITO`

Daten-Eingänge

- **Output_PWM** (`logiBUS::io::DQ::logiBUS_DO_S`): physischer PWM-Ausgang (25/50/75 % Tastgrad, ratiometrisch), Default `logiBUS_DO::Invalid`
- **FACTOR_75** (REAL): Multimeter-kalibrierter Korrekturfaktor auf die nominelle `OUT_75`-Spanne (25.0), durchgereicht an `values_50_75_25.FACTOR_75` (Default `REAL#1.0`)
- **FACTOR_25** (REAL): Multimeter-kalibrierter Korrekturfaktor auf die nominelle `OUT_25`-Spanne (25.0), durchgereicht an `values_50_75_25.FACTOR_25` (Default `REAL#1.0`)
- **QI** (BOOL): Qualifier für den physischen PWM-Ausgang (Default TRUE); FALSE hält die Instanz komplett inaktiv (keine Hardware-Bindung), falls derselbe physische Kanal je nach Maschinenvariante anderweitig genutzt wird

Daten-Ausgänge

- **QO** (BOOL): Quality Output, kommt von `DigitalOutput_PWM.QO`

Adapter

- **HEBEN** (Socket, `AX`): TRUE wählt `OUT_75` (Heben); dauerhaft FALSE verdrahten, wenn nicht gebraucht
- **SENKEN** (Socket, `AX`): TRUE wählt `OUT_25` (Senken); dauerhaft FALSE verdrahten, wenn nicht gebraucht
- **PWM_POW** (Socket, `AR`): PWM-Power, 1.0 = maximale Power, 0.0 = Aus, 0.5 = halb; durchgereicht an `values_50_75_25.PWM_POW`

Funktionsweise

Die Kette im Netzwerk:

1. `HEBEN` ⟶ `ASSEMBLE_AB_FROM_AX.BIT_00` und `SENKEN` ⟶ `ASSEMBLE_AB_FROM_AX.BIT_01`: beide Boole-ans werden zu einem Byte zusammengesetzt (`HEBEN` ist das niederwertigste Bit).
2. `AB_TO_AUI` wandelt das Byte in einen `AUI`-Wert, der als Auswahl `K` an `AR_AUI_MUX_3` geht: `K = 0` (beide FALSE) wählt `IN1` = `OUT_50` (Neutral), `K = 1` (`HEBEN`) wählt `IN2` = `OUT_75`, `K = 2` (`SENKEN`) wählt `IN3` = `OUT_25`.
3. `values_50_75_25` liefert aus `PWM_POW` sowie `FACTOR_75`/`FACTOR_25` die drei Sollwerte `OUT_50`, `OUT_75`, `OUT_25`.
4. `AR_AUI_MUX_3.OUT` ⟶ `AR_R_MUL_Korrektur` (Faktor `PWM_LASTTEILER_KORREKTUR`) ⟶ `AR_R_MUL_PWM13BIT` (Faktor `PWM_13BIT_SKALIERUNG`) ⟶ `AR_TO_AD_NUM` ⟶ `DigitalOutput_PWM.OUT`.
5. `DigitalOutput_PWM` (`logiBUS::io::DQ::logiBUS_QDA_PWM`) gibt den Wert am Ausgang `Output_PWM` aus; `QI` wird direkt an `DigitalOutput_PWM.QI` und das Ereignis `INIT` an `DigitalOutput_PWM.INIT` geführt; `DigitalOutput_PWM.INITO` wird als `INITO` und `DigitalOutput_PWM.QO` als `QO` ausgegeben.

Technische Besonderheiten

- Es gibt **keine** Verriegelung: Der Aufrufer muss sicherstellen, dass `HEBEN` und `SENKEN` nie gleichzeitig TRUE sind, sonst liefert `AR_AUI_MUX_3` für `K = 3` ein undefiniertes Verhalten
- Die Plug-/Variablennamen `_75`/`_25` sind historisch; die tatsächlich kalibrierten Werte liegen bei `PWM_POW = 1.0` bei 90.0 bzw. 15.0 (siehe `values_50_75_25`)

- Der Eingang `QI` (Version 1.1) erlaubt, die Instanz komplett inaktiv zu halten, wenn der Kanal anderweitig genutzt wird
- Der Ereigniseingang `INIT` (Version 1.2) ersetzt die Auto-Verteilung der Initialisierung: der Aufrufer löst sie selbst aus („Init wie bei Tecu“), sodass `QI` vor der physischen Kanalbindung feststeht
- Der Ereignisausgang `INITO` und der Datenausgang `QO` (Version 1.3) reichen das echte Confirm bzw. den Quality-Status von `DigitalOutput_PWM` durch; so kann der Aufrufer (z. B. `Aufnahme_Option.SUB` über den `Option::AOption`-Socket) auf ein echtes `INIT_CNF` statt auf ein Timing-Delay warten
- Erster Aufrufer: `Bunkerboden_PWM.SUB`, die nur `SENKEN` (Vollgas) nutzt und `HEBEN` dauerhaft auf FALSE legt

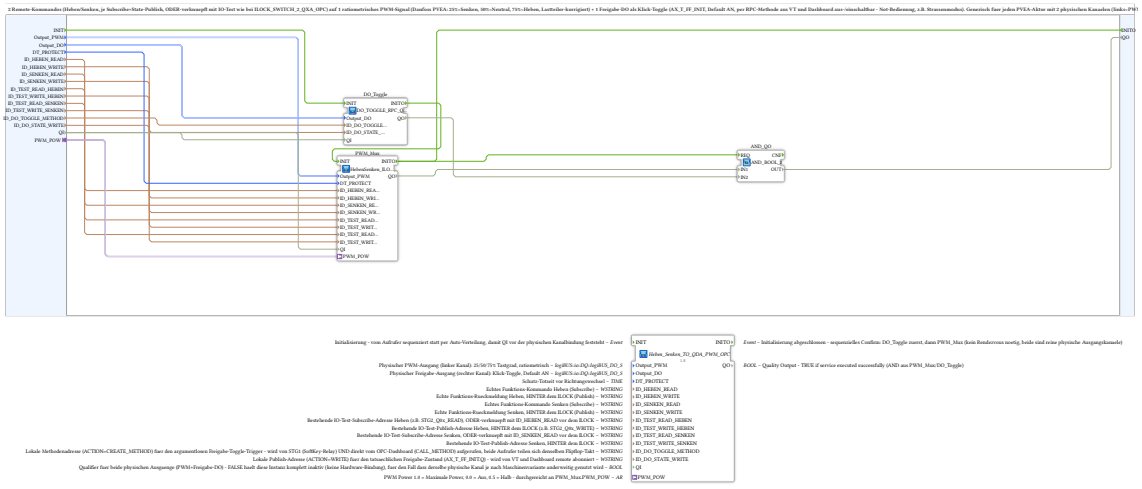
Anwendungsszenarien

- Ansteuerung eines Proportionalventils (Danfoss PVEA) mit 25/50/75 % PWM für Senken/Neutral/Heben
- Anwendungen mit bereits gegateten oder nur einseitig genutzten Richtungssignalen

Fazit

`HebenSenken_QDA_PWM` setzt zwei Richtungs-Booleans ohne Verriegelung in eine der drei ratiometrischen PWM-Stufen um und gibt sie skaliert am physischen PWM-Ausgang aus.

Heben_Senken_TO_QDA_PWM_OPC



• — ○ • — ○ • — ○ •

Einleitung

Die SubApp `Heben_Senken_TO_QDA_PWM_OPC` ist ein generischer Baustein zur Steuerung eines Danfoss PVEA-Aktors mit zwei physischen Kanälen: linkem PWM-Kanal (25% Senken, 50% Neutral, 75% Heben, lastteilerkorrigiert) und rechtem Freigabe-Kanal (Klick-Toggle, Default AN). Sie verarbeitet zwei Remote-Kommandos (Heben/Senken) über Subscribe/Publish, verknüpft diese ODER mit bestehenden IO-Test-Signalen und erzeugt daraus ein ratiometrisches PWM-Signal sowie einen umschaltbaren Freigabe-DO. Der Baustein ist für die Bedienung von einem Modul ohne eigenes VT ausgelegt und unterstützt Not-Bedienung (z. B. Straßenmodus) über eine RPC-Methode.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierung; vom Aufrufer sequenziert statt per Auto-Verteilung, damit `QI` vor der physischen Kanalbindung feststeht; wird an `DO_Toggle.INIT` durchgereicht; `DO_Toggle.INITO` startet `PWM_Mux.INIT` (sequenzielle Kette)

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierung abgeschlossen; sequenzielles Confirm: `DO_Toggle` zuerst, dann `PWM_Mux`; `INITO` kommt von `PWM_Mux.INITO` (kein Rendezvous nötig, beide sind reine physische Ausgangskanäle)

Daten-Eingänge

Name	Typ	Initialwert	Kommentar
QI	BOOL	TRUE	Qualifier für beide physischen Ausgänge (PWM + Freigabe-DO); FALSE hält die Instanz komplett inaktiv (keine Hardware-Bindung), falls derselbe Kanal je nach Maschinenvariante anderweitig genutzt wird
Output_PWM	logiBUS::io::DQ::logiBUS_DO_6	logiBUS_DO::Invalid	Physischer PWM-Ausgang (linker Kanal): 25/50/75% Tastgrad, ratiometrisch
Output_DO	logiBUS::io::DQ::logiBUS_DO_5	logiBUS_DO::Invalid	Physischer Freigabe-Ausgang (rechter Kanal): Klick-Toggle, Default AN
DT_PROTECT	TIME	T#300ms	Schutz-Totzeit vor Richtungswechsel
ID_HEBEN_READ	WSTRING	–	Echtes Funktions-Kommando Heben (Subscribe)
ID_HEBEN_WRITE	WSTRING	–	Echte Funktions-Rückmeldung Heben, HINTER dem ILOCK (Publish)
ID_SENKEN_READ	WSTRING	–	Echtes Funktions-Kommando Senken (Subscribe)
ID_SENKEN_WRITE	WSTRING	–	Echte Funktions-Rückmeldung Senken, HINTER dem ILOCK (Publish)
ID_TEST_READ_HEBEN	WSTRING	–	Bestehende IO-Test-Subscribe-Adresse Heben (z.B. STG2_Q0x_READ), ODER-verknüpft mit ID_HEBEN_READ vor dem ILOCK
ID_TEST_WRITE_HEBEN	WSTRING	–	Bestehende IO-Test-Publish-Adresse Heben, HINTER dem ILOCK (z.B. STG2_Q0x_WRITE)
ID_TEST_READ_SENKEN	WSTRING	–	Bestehende IO-Test-Subscribe-Adresse Senken, ODER-verknüpft

			mit ID_SENKEN_READ vor dem ILOCK
ID_TEST_WRITE_SENKEN	WSTRING	–	Bestehende IO-Test-Publish-Adresse Senken, HINTER dem ILOCK
ID_DO_TOGGLE_METHOD	WSTRING	–	Lokale Methodenadresse (ACTION=CREATE_METHOD) für den argumentlosen Freigabe-Toggle-Trigger – wird von STG1 (SoftKey-Relay) UND direkt vom OPC-Dashboard (CALL_METHOD) aufgerufen
ID_DO_STATE_WRITE	WSTRING	–	Lokale Publish-Adresse (ACTION=WRITE) für den tatsächlichen Freigabe-Zustand (AX_T_FF_INIT.Q) – wird von VT und Dashboard remote abonniert

Daten-Ausgänge

- **QO** (BOOL): Quality Output, UND-Verknüpfung (AND_QO, AND_BOOL_2) aus PWM_Mux.QO und DO_Toggle.QO

Adapter

Keine Adapter vorhanden.

Funktionsweise

Die SubApp ist eine Zusammensetzung aus zwei internen SubApps:

1. **PWM_Mux** (MyLib::sys::HebenSenken_ILOCK_QDA_PWM_OPC):
   - Empfängt die Funktions- und IO-Test-Kommandos (Subscribe).
   - Verknüpft jeweils ID_HEBEN_READ mit ID_TEST_READ_HEBEN (ODER) und ID_SENKEN_READ mit ID_TEST_READ_SENKEN (ODER).
   - Leitet die kombinierten Befehle durch eine ILOCK-Logik, die Richtungswechsel nur nach Ablauf der Schutz-Totzeit DT_PROTECT zulässt.
   - Erzeugt das ratiometrische PWM-Signal auf Output_PWM (25% Senken, 50% Neutral, 75% Heben) mit Lastteilerkorrektur.
   - Publiziert die tatsächlichen Zustände über ID_HEBEN_WRITE, ID_SENKEN_WRITE sowie die entsprechenden Test-Publish-Adressen nach dem ILOCK.
2. **DO_Toggle** (MyLib::sys::DO_TOGGLE_RPC_QXA_OPC):
   - Implementiert einen Klick-Toggle-Flipflop (AX_T_FF_INIT) für den Freigabe-Ausgang Output_DO.
   - Der Ausgang ist initial AN und kann über die RPC-Methode ID_DO_TOGGLE_METHOD umgeschaltet werden.
   - Der aktuelle Zustand wird über ID_DO_STATE_WRITE publiziert.

QI wird gemeinsam an PWM_Mux.QI und DO_Toggle.QI geführt, das Ereignis INIT an DO_Toggle.INIT; DO_Toggle.INITO löst PWM_Mux.INIT aus. Beide physischen Kanäle werden also gemeinsam inaktiv gehalten bzw. nacheinander vom Aufrufer initialisiert.

Die Bestätigung ist sequenziell (kein E_REND_2-Baustein): PWM_Mux.INITO ⟶ INITO und AND_QO.REQ. AND_QO (iec61131::booleanOperators::AND_BOOL_2) verknüpft PWM_Mux.QO (IN1) und DO_Toggle.QO (IN2) zum Ausgang QO.

Die beiden Teile sind in der Schnittstelle vollständig gekapselt; extern ist nur die Parameterisierung und die Kommunikationsadressen sichtbar.

Technische Besonderheiten

- **ODER-Verknüpfung von Fernbedienung und IO-Test** vor der Schutzlogik (ILOCK), dadurch können sowohl echte Funktionskommandos als auch Testsignale den gleichen Wirkpfad nutzen.
- **Zwei getrennte Bibliotheksbausteine** für PWM-/ILOCK-Logik und DO-Toggle, was die Wiederverwendbarkeit erhöht.
- **Freigabe-DO als Klick-Toggle** mit RPC-Methode: Erlaubt Not-Bedienung aus Dashboard oder Softkey, ohne eigene VT.
- **PWM-Ausgang mit lastteilerkorrigiertem Tastgrad** für den PVEA-Aktor.
- **Schutz-Totzeit** (`DT_PROTECT`) verhindert schädliche Richtungswechsel während des Betriebs.
- **Publish-Adressen für Status** liegen hinter dem ILOCK, sodass nur tatsächlich ausgeführte Befehle zurückgemeldet werden.

Zustandsübersicht

Die SubApp selbst besitzt keinen eigenen Zustandsautomaten, sie delegiert die Zustandsverwaltung an die internen Bausteine:

- **PWM_Mux**: Enthält einen ILOCK (Interlock) mit Zuständen wie „Neutral“, „Heben aktiv“, „Senken aktiv“ und „Totzeit“. Die Umschaltung zwischen den Zuständen unterliegt der Totzeitüberwachung.
- **DO_Toggle**: Ein einfaches Flipflop (AX_T_FF_INIT) mit Zuständen AN/AUS. Der Zustand wird durch ein RPC-Triggerereignis getoggelt und bleibt auch nach einem Neustart des Bausteins erhalten (sofern der Initialwert definiert ist).

Anwendungsszenarien

- Steuerung eines Danfoss PVEA-Aktors (z. B. für Anbaugeräte an Traktoren oder Kommunalfahrzeugen) über Remote-Kommandos.
- Einsatz in Systemen, in denen ein Modul ohne eigenes Visualisierungsterminal (VT) die Bedienung von Heben/Senken und Freigabe übernehmen muss.
- Not-Bedienung über ein OPC-Dashboard oder Softkey-Relay (z. B. Straßenmodus), wenn die normale Bedienoberfläche nicht verfügbar ist.
- Integration in bereits bestehende IO-Test-Strukturen, ohne die bisherigen Testadressen zu verlieren.

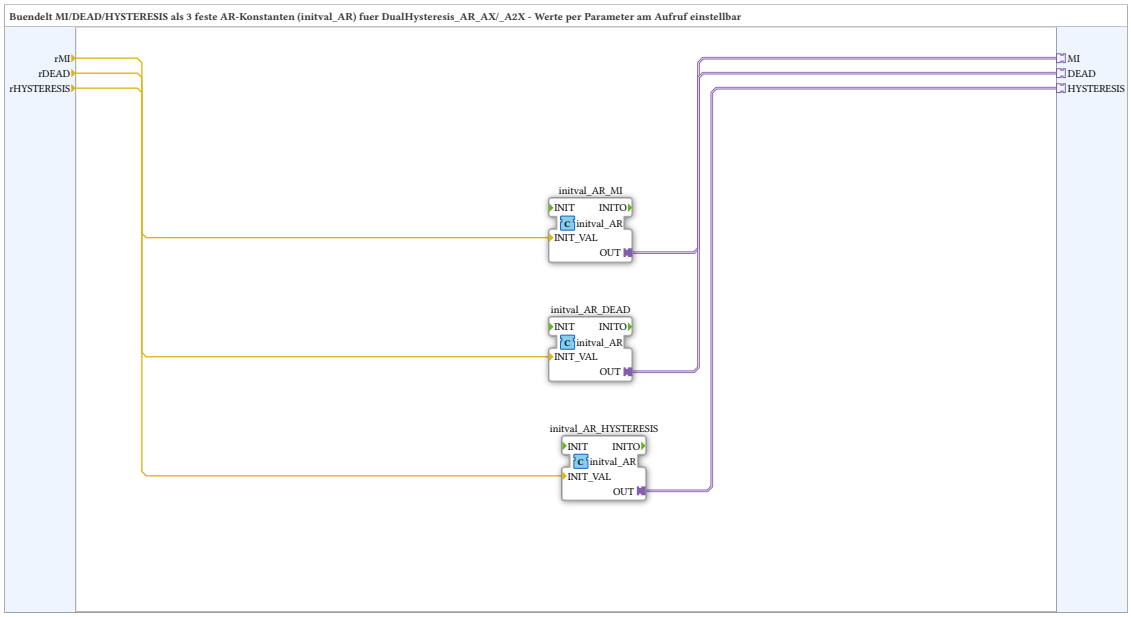
Vergleich mit ähnlichen Bausteinen

- **ILOCK_SWITCH_2_QXA_OPC** (im Kommentar erwähnt): Dieser Baustein steuert vermutlich zwei digitale Ausgänge mit ILOCK, hat aber keine PWM-Erzeugung und keinen separaten Freigabe-Toggle. `Heben_Senken_TO_QDA_PWM_OPC` erweitert dieses Konzept um den PWM-Mux und die RPC-gesteuerte Freigabe.
- **Direkte PWM-Steuerung ohne ILOCK**: Bau steine ohne Interlock sind schneller, bergen aber das Risiko von mechanischen Schäden bei sofortigem Richtungswechsel. Dieser Baustein bietet durch `DT_PROTECT` eine sichere Alternative.
- **Separate Bausteine für PWM und DO**: Die Kombination in einer SubApp reduziert den Verdrahtungsaufwand und kapselt die Logik in einer wiederverwendbaren Einheit.

Fazit

`Heben_Senken_TO_QDA_PWM_OPC` vereint PWM-Erzeugung, ILOCK-Schutz, IO-Test-Option und einen RPC-umschaltbaren Freigabeausgang in einem kompakten Baustein. Durch die modulare Aufteilung in bestehende Bibliotheksbausteine bleibt die Wartung einfach und die Schnittstelle klar. Die Berücksichtigung von Schutz-Totzeit und Status-Publishing nach dem ILOCK macht den Baustein robust für reale Einsatzfälle, insbesondere in der mobilen Automation.

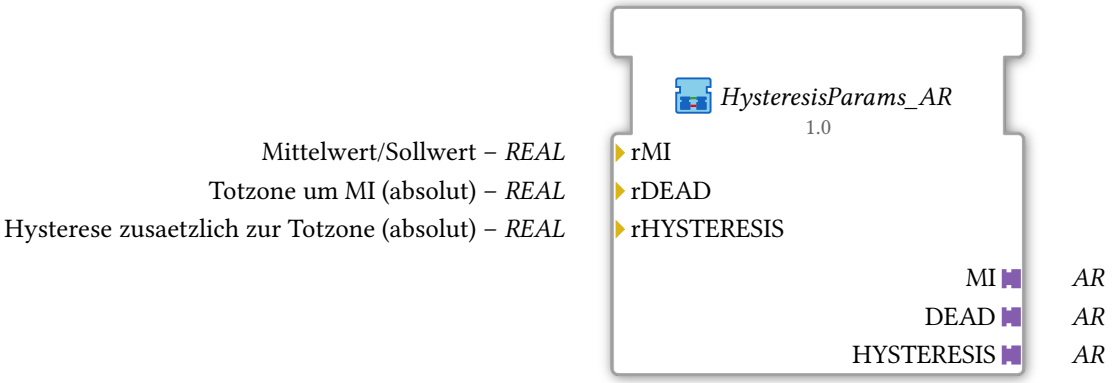
HysteresisParams_AR



• – ○ • – ○ • – ○ •

Einleitung

`HysteresisParams_AR` bündelt die drei Parameter `MI` (Mittelwert/Sollwert), `DEAD` (Totzone) und `HYSTERESIS` (zusätzliche Hysterese) als feste `AR`-Adapter-Konstanten für `DualHysteresis_AR_AX`/`DualHysteresis_AR_A2X`. Statt drei `initval_AR`-Instanzen in jeder Übung/SubApp einzeln zu wiederholen, fasst dieser Baustein sie zusammen.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: HysteresisParams_AR

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **initval_AR_MI** / **initval_AR_DEAD** / **initval_AR_HYSTERESIS**: je `adapter::types::unidirectional::AR::initval::initval_AR` — wandeln einen `REAL`-Eingangswert in einen festen `AR`-Adapter-Plug um.
- **Funktionsweise**: Die drei `InputVars` `rMI`/`rDEAD`/`rHYSTERESIS` werden je über eine `initval_AR`-Instanz in die entsprechenden `AR`-Plugs (`MI`/`DEAD`/`HYSTERESIS`) umgewandelt.

Programmablauf und Verbindungen

1. `rMI` ⟶ `initval_AR_MI.INIT_VAL`; `rDEAD` ⟶ `initval_AR_DEAD.INIT_VAL`; `rHYSTERESIS` ⟶ `initval_AR_HYSTERESIS.INIT_VAL` (Datenverbindungen).
2. `initval_AR_MI.OUT` ⟶ `MI`; `initval_AR_DEAD.OUT` ⟶ `DEAD`; `initval_AR_HYSTERESIS.OUT` ⟶ `HYSTERESIS` (Adapterverbindungen, direkt an die SubApp-Schnittstelle).

Technische Besonderheiten

- **Parametrierbare Standardwerte**: `rMI/rDEAD/rHYSTERESIS` besitzen sinnvolle Standardwerte (`500.0/20.0/30.0`), die am Aufruf per Parameter überschreibbar sind — analog zu `u16ObjId` bei den GreenWhiteBackground-Bausteinen.
- **Wiederverwendung statt Wiederholung**: Ersetzt das manuelle Duplizieren dreier `initval_AR`-Instanzen, wie es zuerst in `Uebung_234_AX`/`Uebung_235_AX` gemacht wurde.

Anwendungsszenarien

- Parametrierung von `DualHysteresis_AR_AX`/`DualHysteresis_AR_A2X`, wenn Sollwert, Totzone und Hysterese als feste, am Aufruf überschreibbare Konstanten benötigt werden.
- Konsistente Parametrierung mehrerer vergleichbarer Übungen (z. B. `Uebung_234_AX` für Hardware und `Uebung_235_AX` für VT), die mit identischen Werten direkt vergleichbar bleiben sollen.

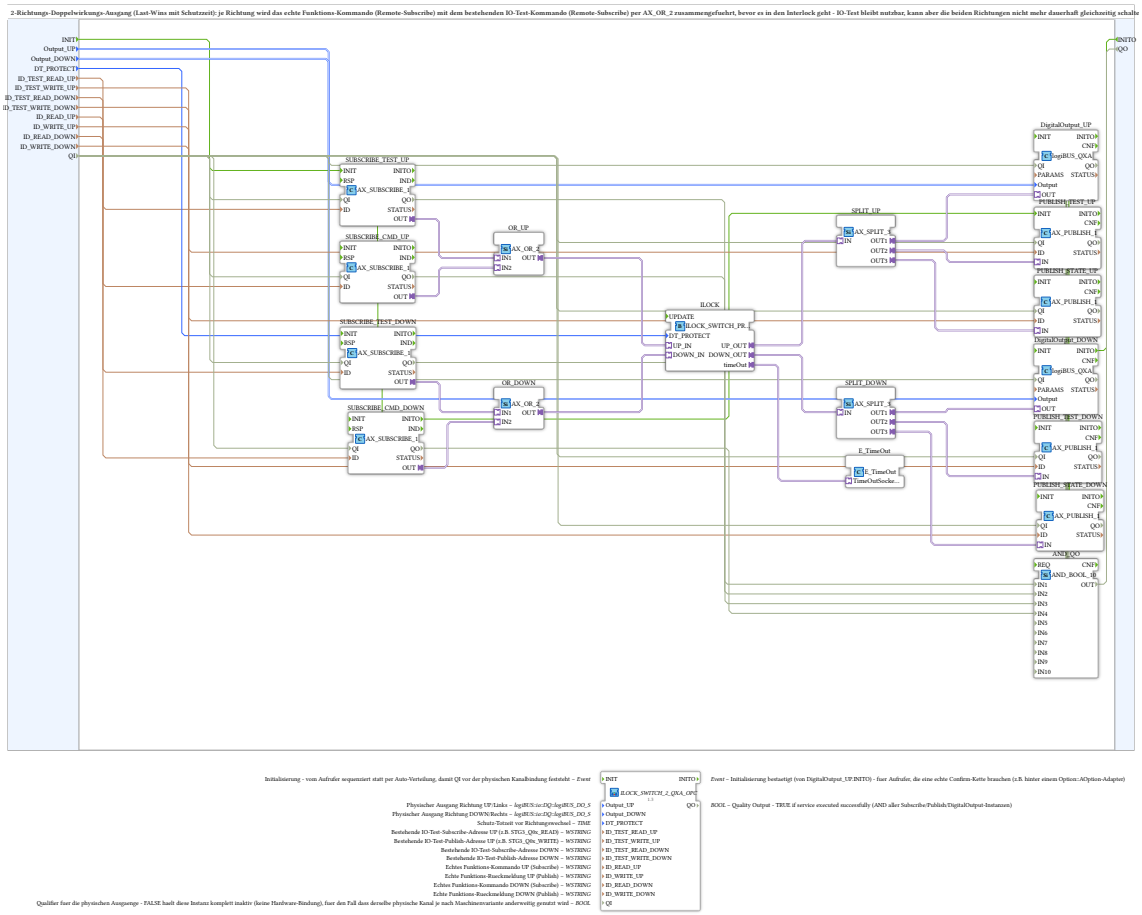
Zusammenfassung

`HysteresisParams_AR` bündelt drei feste `AR`-Parameter-Konstanten für die Hysterese-Bausteine hinter einem einzigen, parametrierbaren Baustein.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ILOCK_SWITCH_2_QXA_OPC



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsbaustein **ILOCK_SWITCH_2_QXA_OPC** ist eine generische Subapplikation zur Ansteuerung eines digitalen Doppelwirkungs-Ausgangs (z. B. für Ventile oder Schaltanlagen) mit integriertem Schutz-Interlock und Rückmeldung über OPC. Er kombiniert zwei unabhängige Steuerkanäle (Richtung „UP“ und „DOWN“), die über logiBUS-Module physisch geschaltet werden.

Der Baustein vereint drei Hauptaufgaben:

- **Zusammenführung von Funktions- und Testkommandos** für jede Richtung,
- **Verriegelung gegen gleichzeitiges Schalten** beider Richtungen mit einstellbarer Schutzzeit,
- **Rückmeldung** des tatsächlichen Ausgangszustands über OPC-Publizierung, sowohl für die Funktionssteuerung als auch für das IO-Test-System.

Er ist speziell für Anlagen konzipiert, bei denen die Bedienung über ein zentrales Modul ohne eigenes Vor-Ort-Terminal erfolgt und eine zuverlässige Absicherung gegen Fehlschaltungen erforderlich ist.

Schnittstellenstruktur

Die Subapplikation besitzt je einen Ereignis-Eingang `INIT` und einen Ereignis-Ausgang `INITO`, keine Adapter an der Subapplikationsgrenze, **Daten-Eingänge** (InputVars) und den Daten-Ausgang `QO`. Die interne Steuerung erfolgt über diese Datenwerte und asynchrone Kommunikation (über die intern eingebetteten Subscribe/Publish-Bausteine).

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierung; vom Aufrufer sequenziert statt per Auto-Verteilung, damit `QI` vor der physischen Kanalbindung feststeht; startet die Initialisierungskette durch alle inneren Instanzen (`SUBSCRIBE_TEST_UP` → `SUBSCRIBE_CMD_UP` → `SUBSCRIBE_TEST_DOWN` → `SUBSCRIBE_CMD_DOWN` → `PUBLISH_TEST_UP` → `PUBLISH_STATE_UP` → `PUBLISH_TEST_DOWN` → `PUBLISH_STATE_DOWN` → `DigitalOutput_UP` → `DigitalOutput_DOWN`); die physischen Ausgänge werden also zuletzt initialisiert. Der Aufrufer muss `INIT` selbst auslösen (z. B. ohne Verzögerung für unverändertes Sofort-Verhalten oder verzögert, wenn auf einen INI-Wert gewartet werden muss).

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierung bestätigt (von `DigitalOutput_DOWN.INITO`, dem Ende der Kette); für Aufrufer, die eine echte Confirm-Kette brauchen (z. B. hinter einem `Option::AOption`-Adapter). Rein additiv, unverdrahtet harmlos.

Daten-Eingänge

Name	Datentyp	Kommentar
`QI`	`BOOL`	Qualifier für die physischen Ausgänge; FALSE hält die Instanz komplett inaktiv (keine Hardware-Bindung), Default `TRUE`; wird an alle Subscribe-, Publish- und DigitalOutput-Instanzen verteilt
`Output_UP`	`logiBUS::io::DQ::logiBUS_DO_S`	Physischer Ausgang Richtung UP/Links. Initialwert: `logiBUS_DO::Invalid`
`Output_DOWN`	`logiBUS::io::DQ::logiBUS_DO_S`	Physischer Ausgang Richtung DOWN/Rechts. Initialwert: `logiBUS_DO::Invalid`
`DT_PROTECT`	`TIME`	Schutz-Totzeit vor Richtungswechsel. Initialwert: `T#300ms`
`ID_TEST_READ_UP`	`WSTRING`	Bestehende IO-Test-Subscribe-Adresse UP (z. B. `STG3_Q0x_READ`)
`ID_TEST_WRITE_UP`	`WSTRING`	Bestehende IO-Test-Publish-Adresse UP (z. B. `STG3_Q0x_WRITE`)
`ID_TEST_READ_DOWN`	`WSTRING`	Bestehende IO-Test-Subscribe-Adresse DOWN
`ID_TEST_WRITE_DOWN`	`WSTRING`	Bestehende IO-Test-Publish-Adresse DOWN
`ID_READ_UP`	`WSTRING`	Echtes Funktions-Kommando UP (Subscribe)

ID_WRITE_UP	WSTRING	Echte Funktions-Rückmeldung UP (Publish)
ID_READ_DOWN	WSTRING	Echtes Funktions-Kommando DOWN (Subscribe)
ID_WRITE_DOWN	WSTRING	Echte Funktions-Rückmeldung DOWN (Publish)

Daten-Ausgänge

- **QO** (BOOL): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde (AND aller Subscribe-/Publish-/DigitalOutput-Instanzen).

Adapter

Keine (nicht vorhanden).

Funktionsweise

Die Subapplikation verarbeitet pro Richtung (UP/DOWN) zwei Signale: ein **Funktions-Kommando** und ein **IO-Test-Kommando**. Diese werden über die jeweiligen Subscribe-Bausteine (SUBSCRIBE_CMD_UP, SUBSCRIBE_TEST_UP, etc.) empfangen und anschließend über eine ODER-Verknüpfung (AX_OR_2) zusammengeführt. Das Ergebnis ist das effektive Schaltkommando für diese Richtung.

Das effektive Kommando wird an den zentralen Interlock-Baustein (ILOCK_SWITCH_PROTECT_AX) übergeben. Dieser Baustein verhindert, dass beide Richtungen gleichzeitig aktiv werden. Zusätzlich implementiert er eine Schutzzeit (DT_PROTECT), die vor einem Richtungswechsel eingehalten werden muss. Nach Ablauf der Schutzzeit wird das Kommando freigegeben.

Der Ausgang des Interlocks wird über einen Splitter (AX_SPLIT_3) an drei Stellen verteilt:

1. An den physischen logiBUS-Kanal (logiBUS_QXA), der den tatsächlichen Ausgang steuert.
2. An einen Publish-Baustein für die **IO-Test-Rückmeldung** (PUBLISH_TEST_UP bzw. PUBLISH_TEST_DOWN), damit das IO-Test-System den realen Zustand erhält.
3. An einen Publish-Baustein für die **Funktions-Rückmeldung** (PUBLISH_STATE_UP bzw. PUBLISH_STATE_DOWN), z. B. für die Anzeige „GreenWhiteBackground“ am SoftKey.

Initialisierung und Qualität: INIT läuft als Kette SUBSCRIBE_TEST_UP → SUBSCRIBE_CMD_UP → SUBSCRIBE_TEST_DOWN → SUBSCRIBE_CMD_DOWN → PUBLISH_TEST_UP → PUBLISH_STATE_UP → PUBLISH_TEST_DOWN → PUBLISH_STATE_DOWN → DigitalOutput_UP → DigitalOutput_DOWN (jeweils INITO → nächstes INIT); DigitalOutput_DOWN.INITO geht an INITO und an AND_QO.REQ. QI wird an alle acht Subscribe-/Publish-Instanzen und an beide DigitalOutput-Instanzen verteilt. QO ist der Ausgang der Instanz AND_QO (iec61131::booleanOperators::AND_BOOL_10), die die QO dieser zehn Instanzen verknüpft.

Die Rückmeldungen werden also immer von dem tatsächlichen, verriegelten Zustand abgeleitet. Dadurch bleiben IO-Test und Funktionssteuerung konsistent und es kann nicht zu Fehlinterpretationen kommen.

Technische Besonderheiten

- **Last-Wins-Logik:** Die ODER-Verknüpfung zwischen Funktions- und Testkommando bedeutet, dass das zuletzt eintreffende Signal (egal ob Test oder Funktion) den Zustand bestimmt – es gilt also „Last-Wins“. Der Interlock sorgt aber dafür, dass nie beide Richtungen gleichzeitig geschaltet werden können.
- **Schutzzeit:** Der Parameter DT_PROTECT definiert eine Totzeit, die ein Richtungswechsel dauern darf. Diese Zeit verhindert ein schnelles Umschalten, das mechanische oder elektrische Schäden verursachen könnte. Der Interlock-Baustein signalisiert über sein timeOut-Ereignis das Ende dieser Zeit.
- **IO-Test bleibt nutzbar:** Durch die Zusammenführung vor dem Interlock kann der IO-Test weiterhin jeden Kanal einzeln ansteuern – jedoch nicht dauerhaft beide gleichzeitig, was durch den Interlock unterbunden wird.
- **OPC-Anbindung:** Die Kommunikation erfolgt über OPC UA (implizit durch die Subscribe/Publish-Bausteine). Die Adressen werden über die String-Parameter konfiguriert.

Zustandsübersicht

Der Interlock-Baustein (ILOCK_SWITCH_PROTECT_AX) verwaltet im Wesentlichen drei Zustände:

- **Neutral:** Keiner der beiden Eingänge (UP_IN, DOWN_IN) ist aktiv; der Ausgang ist inaktiv.

- **UP aktiv:** Der Eingang `UP_IN` ist aktiv und die Schutzzeit für einen Wechsel von DOWN auf UP ist abgelaufen; der Ausgang `UP_OUT` ist gesetzt.
- **DOWN aktiv:** Analog für die Richtung DOWN.

Während der Schutzzeit nach einem aktiven Zustand wechselt der Interlock in einen Blockierzustand, in dem das Schalten der anderen Richtung verzögert wird. Die Dauer wird durch `DT_PROTECT` bestimmt. Das Ereignis `timeOut` wird ausgelöst, wenn die Schutzzeit abgelaufen ist und ein Richtungswechsel erlaubt wird.

Der Ausgang des Interlocks ist also **entweder UP oder DOWN oder keiner**, nie beide gleichzeitig. Diese Zustandslogik wird direkt auf den physischen Kanal übertragen.

Anwendungsszenarien

- **Steuerung von Doppelwirkungs-Ventilen** oder Klappen, die mit zwei getrennten Ausgängen (Auf/Zu) angesteuert werden.
- **Anlagen mit separater IO-Test-Funktion**, bei denen der IO-Test parallel zur normalen Funktionssteuerung laufen soll, jedoch ohne die Gefahr von Kurzschluss-Kommandos.
- **Systeme mit OPC-UA-Anbindung**, bei denen die Rückmeldungen des tatsächlichen Schaltzustands für Visualisierung oder Leittechnik benötigt werden.
- **Module ohne eigenes Bedien-Terminal**, bei denen die Steuerung zentral über eine übergeordnete Steuerung erfolgt.

Vergleich mit ähnlichen Bausteinen

Gegenüber einfachen Ausgangsbausteinen (z. B. ohne Interlock) bietet dieser FB erweiterte Sicherheit: Er verhindert unzulässige Doppelansteuerung und führt eine definierte Totzeit ein. Im Gegensatz zu einem Baustein, der nur Funktionskommandos verarbeitet, integriert er zusätzlich das IO-Test-Signal, wodurch zwei getrennte Kommunikationspfade vereinheitlicht werden. Vergleicht man ihn mit einem reinen IO-Test-Baustein, so ergänzt er die Rückmeldung für die Funktionssteuerung und sorgt für eine konsistente Zustandsanzeige.

Einfachere Alternativen bieten möglicherweise keine Schutzzeit oder keine separate Test-/Funktions-Rückmeldung; dieser Baustein bündelt alle Anforderungen in einer kompakten Subapplikation.

Fazit

`ILOCK_SWITCH_2_QXA_OPC` ist eine vielseitige und sichere Lösung für die Ansteuerung digitaler Doppelwirkungs-Aktoren. Durch die Kombination von Interlock, Schutzzeit und OPC-Anbindung wird sowohl die physische Sicherheit als auch die Informationskonsistenz gewährleistet. Die klare Trennung von Funktions- und Testbefehlen ermöglicht einen flexiblen Einsatz in industriellen Umgebungen, in denen mehrere Steuerungswege existieren. Die Parametrierung der Kommunikationsadressen über die Daten-Eingänge macht den Baustein leicht an verschiedene Systemarchitekturen anpassbar.

INI_IN_AND_STORE / NVS_IN_AND_STORE: Gemeinsames Muster

• – ◦ • – ◦ • – ◦ •

Einleitung

`MyLib::sys` enthält eine Familie von Bausteinen, die einen über VT eingegebenen Wert **persistent speichern** und beim Deployment automatisch wieder laden — entweder in einer INI-Datei (`INI_IN_AND_STORE_*`) oder im ESP32-Flash (NVS, `NVS_IN_AND_STORE_*`). Diese Seite erklärt das gemeinsame Muster.

Namensschema

`{INI|NVS}_IN_AND_STORE_<Typ>`, wobei `<Typ>` den Datentyp bestimmt: `AIS` (String-Adapter), `AR` (REAL-Adapter, mit physikalischer Skalierung `NumericObjectPool_S`), `AUDI` (UDINT-Adapter) in test_AX; `UDINT` (klassisch, ohne Adapter) in test_B.

Funktionsweise (test_AX, adapterbasiert)

1. Ein VT-Eingabefeld (`StringValue_AIS`/`NumericValue_PHYSA`/…) liefert den vom Nutzer eingegebenen Wert als Adapter.
2. `INI_<Typ>`/ `NVS_<Typ>` (`eclipse4diac::storage::INI_*` bzw. `logiBUS::storage::esp32_nvs::NVS_*`) speichert diesen Wert persistent unter `KEY`/`SECTION` (nur INI) und gibt beim `INIT` den zuletzt gespeicherten Wert zurück (`SETM=TRUE`: Speichern aktiv, `DEFAULT_VALUE`: Anfangswert, falls noch nichts gespeichert).

3. Der (neu geladene oder neu eingegebene) Wert wird über `<Typ>_SPLIT_2` verzweigt: einmal als Plug `VALUEO` nach außen (Weiterverwendung im aufrufenden Netzwerk) und einmal an `Q_StringValue_AIS`/`Q_NumericValue_PHYSA` (Rückschreiben auf das VT-Anzeigefeld, damit Eingabefeld und Anzeige synchron bleiben).

Funktionsweise (test_B, klassisch)

Die `test_B`-Variante (`INI_IN_AND_STORE_UDINT`/`NVS_IN_AND_STORE_UDINT`) ist älter und nutzt klassische Ereignis-/Datenverbindungen statt Adapter: Ein `NumericValue_ID`-Eingabefeld liefert einen DWORD-Wert, der über `F_DWORD_TO_UDINT` in UDINT gewandelt und an `INI`/`NVS` (generische, nicht typspezifische Bausteine) übergeben wird. Der gespeicherte Wert geht sowohl als `VALUEO` nach außen als auch zurück an `Q_NumericValue` für die Anzeige.

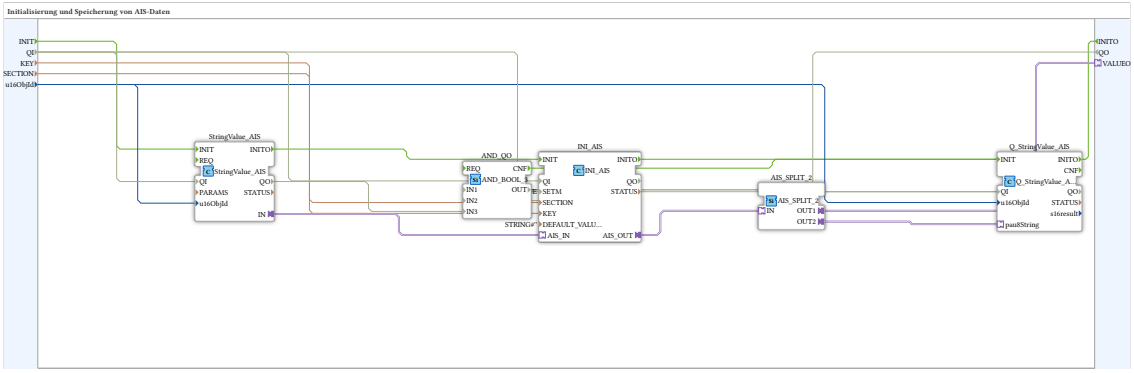
Zusammenfassung

Beide Varianten lösen dasselbe Problem — „vom Nutzer eingegebenen Wert über Neustarts hinweg merken“ — mit demselben Grundmuster (Eingabefeld ⟶ Speicher-FB mit KEY/SECTION ⟶ Verzweigung auf Ausgang + Anzeige-Rückschreibung), nur mit unterschiedlichem Speicherort (INI-Datei vs. NVS-Flash) und unterschiedlichem Verdrahtungsstil (Adapter in test_AX, klassisch in test_B).

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

INI_IN_AND_STORE_AIS



• – ○ • – ○ • – ○ •

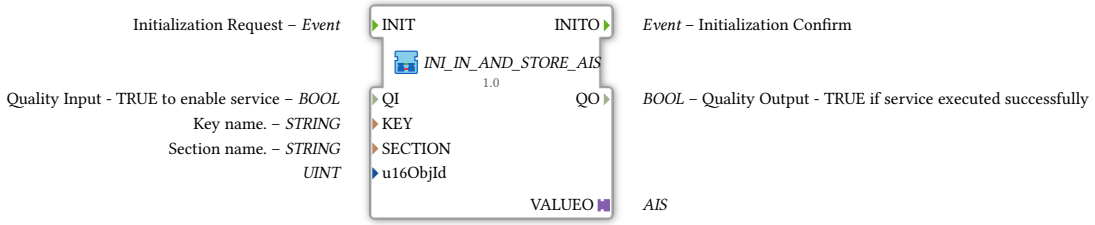
Einleitung

`INI_IN_AND_STORE_AIS` speichert einen über VT eingegebenen String-Wert (AIS-Adapter) persistent in einer INI-Datei und lädt ihn beim Deployment automatisch wieder.

Allgemeines Muster siehe INI_IN_AND_STORE / NVS_IN_AND_STORE (gemeinsames Muster).

Schnittstelle (Quality-Muster): Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`, „Quality Input – TRUE to enable service“) und Datenausgang `QO` (`BOOL`, „Quality Output – TRUE if service executed successfully“) sind mit allen drei internen Dienst-Bausteinen (`StringValue_AIS`, `INI_AIS`, `Q_StringValue_AIS`) verbunden: `QI` ⟶ `Q_StringValue_AIS.QI`, `INI_AIS.QI` und `StringValue_AIS.QI`; `QO` ist das UND (`AND_QO`, `iec61131::booleanOperators::AND_BOOL_3`) aus `Q_StringValue_AIS.QO`, `INI_AIS.QO` und `StringValue_AIS.QO`.

Zusätzlich besitzt der Baustein die Ereignisse `INIT` (Eingang, `EInit`) und `INITO` (Ausgang, `EInit`); die Initialisierung wird gekettet: `INIT` ⟶ `StringValue_AIS.INIT`; `StringValue_AIS.INITO` ⟶ `INI_AIS.INIT`; `INI_AIS.INITO` ⟶ `Q_StringValue_AIS.INIT`; `Q_StringValue_AIS.INITO` ⟶ `INITO` (und ⟶ `AND_QO.REQ`).



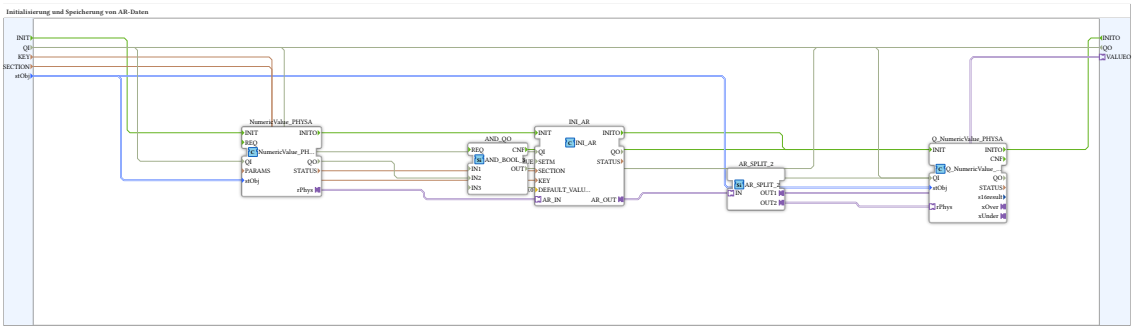
Zusammenfassung

AIS-Variante (String) der INI-Speicherfamilie.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

INI_IN_AND_STORE_AR



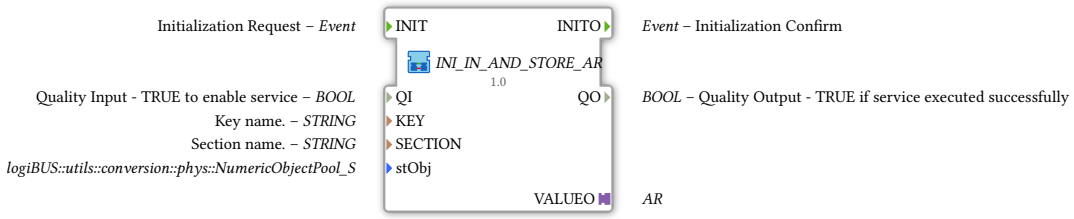
• – ○ • – ○ • – ○ •

Einleitung

`INI_IN_AND_STORE_AR` speichert einen über VT eingegebenen physikalisch skalierten REAL-Wert (AR-Adapter, `NumericValue_PHYSA`/`Q_NumericValue_PHYSA` mit `NumericObjectPool_S`-Skalierung: `r32Scale`, `i32Offset`, `u8Decimals`) persistent in einer INI-Datei.

Allgemeines Muster siehe INI_IN_AND_STORE / NVS_IN_AND_STORE (gemeinsames Muster).

Schnittstelle (Quality-Muster): Ereignis-Eingang `INIT` und Ereignis-Ausgang `INITO` (je `EInit`), Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`, „Quality Input – TRUE to enable service“) und Datenausgang `QO` (`BOOL`, „Quality Output – TRUE if service executed successfully“). Initialisierung: `INIT` ⟶ `NumericValue_PHYSA.INIT`; `NumericValue_PHYSA.INITO` ⟶ `INI_AR.INIT`; `INI_AR.INITO` ⟶ `Q_NumericValue_PHYSA.INIT`; `Q_NumericValue_PHYSA.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`. Qualität: `QI` ⟶ `Q_NumericValue_PHYSA.QI`, `NumericValue_PHYSA.QI`, `INI_AR.QI`; `Q_NumericValue_PHYSA.QO`, `NumericValue_PHYSA.QO`, `INI_AR.QO` ⟶ `AND_QO.IN1` bis `AND_QO.IN3` (in dieser Reihenfolge); `AND_QO.OUT` ⟶ `QO` (`AND_QO`: `iec61131::booleanOperators::AND_BOOL_3`).



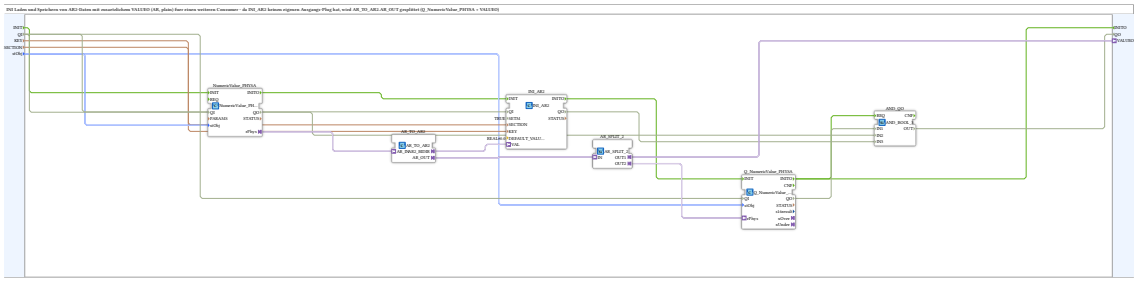
Zusammenfassung

AR-Variante (physikalisch skaliertes REAL) der INI-Speicherfamilie.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

INI_IN_AND_STORE_AR2

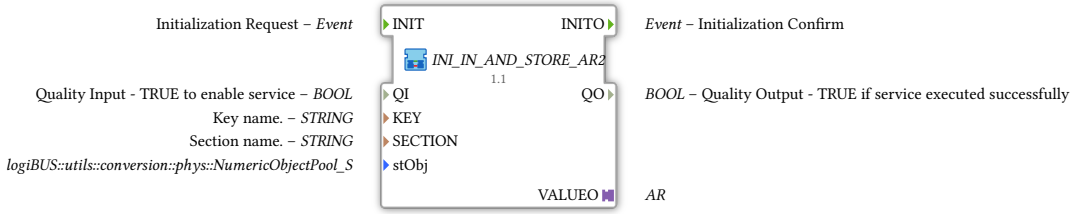


• – ○ • – ○ • – ○ •

Einleitung

`INI_IN_AND_STORE_AR2` ist die AR2-Adapter-Variante von `INI_IN_AND_STORE_AR`: ein per VT eingegebener physikalisch skalierter REAL-Wert (`NumericValue_PHYSA`) wird persistent in einer INI-Datei gespeichert (`eclipse4diac::storage::INI_AR2`, Speicherung ueber den AR2-Adaptersocket statt Klartextwert). Da `INI_AR2` keinen eigenen Ausgangs-Plug besitzt, wird der `AR_TO_AR2`-Ausgang per `AR_SPLIT_2` verdoppelt: einmal zurueck auf die VT-Anzeige (`Q_NumericValue_PHYSA`), einmal nach aussen auf `VALUEO` fuer einen weiteren Consumer.

Allgemeines Muster siehe INI_IN_AND_STORE / NVS_IN_AND_STORE (gemeinsames Muster).



Technische Besonderheiten

- `SETM=TRUE`: eine Live-Aenderung waehrend des laufenden Betriebs wird sofort bestaetigt/zurueckgemeldet, nicht nur beim Boot - wichtig, da der frische Wert im selben Lauf ueber `VALUEO` weiterverwendet werden kann.
- `AR_TO_AR2` wandelt den plain-AR-Wert des VT-Eingabefelds in den bidirektionalen AR2-Adapter, den `INI_AR2` fuer die Speicherung erwartet.

Qualitätsanschlüsse (QI/QO)

Der Baustein besitzt den Ereigniseingang `INIT` (`EInit`, Initialization Request), den Ereignisausgang `INITO` (`EInit`, Initialization Confirm), den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`, Quality Input - TRUE aktiviert den Dienst) und den Datenausgang `QO` (`BOOL`, Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde).

- Initialisierungskette: `INIT` ⟶ `NumericValue_PHYSA.INIT`, `NumericValue_PHYSA.INITO` ⟶ `INI_AR2.INIT`, `INI_AR2.INITO` ⟶ `Q_NumericValue_PHYSA.INIT`, `Q_NumericValue_PHYSA.INITO` ⟶ `INITO` (und zusätzlich ⟶ `AND_QO.REQ`)
- `QI` ist mit `Q_NumericValue_PHYSA.QI`, `NumericValue_PHYSA.QI` und `INI_AR2.QI` verbunden
- `QO` ist die UND-Verknüpfung (`AND_QO`, Typ `iec61131::booleanOperators::AND_BOOL_3`) von `Q_NumericValue_PHYSA.QO`, `NumericValue_PHYSA.QO` und `INI_AR2.QO`

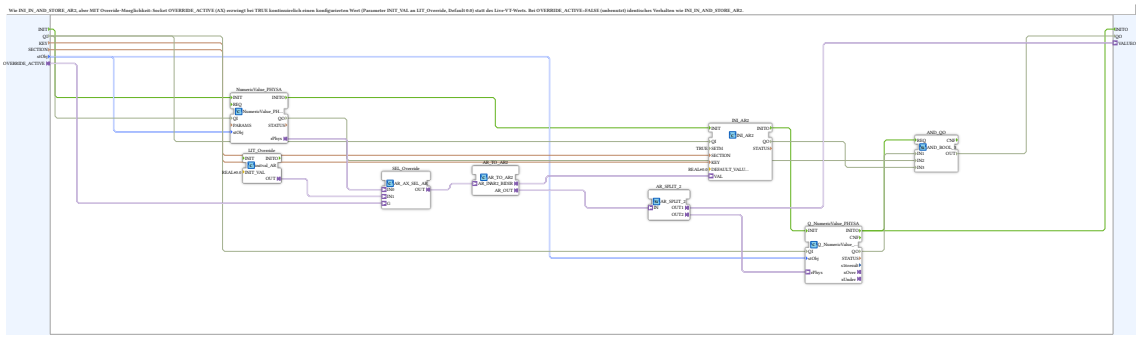
Zusammenfassung

AR2-Adapter-Variante der INI-Speicherfamilie, mit zusaetzlichem `VALUEO`-Ausgang fuer einen weiteren Consumer.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

INI_IN_AND_STORE_AR2_OVERRIDE



• – ○ • – ○ • – ○ •

Einleitung

`INI_IN_AND_STORE_AR2_OVERRIDE` ist wie `INI_IN_AND_STORE_AR2`, jedoch mit Override-Möglichkeit: Ist der Socket `OVERRIDE_ACTIVE` (AX) dauerhaft TRUE, wird statt des Live-VT-Werts ein konfigurierter Wert erzwungen (Parameter `INIT_VAL` an `LIT_Override`, Default `0.0`). Bei `OVERRIDE_ACTIVE=FALSE` (unbenutzt) ist das Verhalten identisch zu `INI_IN_AND_STORE_AR2`.

Allgemeines Muster siehe INI_IN_AND_STORE / NVS_IN_AND_STORE (gemeinsames Muster).

Schnittstelle

Eingangsvariablen

- `QI` (BOOL, Vorgabe `TRUE`): Quality Input - TRUE aktiviert den Dienst; wird an `NumericValue_PHYSA`, `INI_AR2` und `Q_NumericValue_PHYSA` durchgereicht.
- `KEY` (STRING): Schlüsselname in der INI-Datei.
- `SECTION` (STRING): Abschnittsname in der INI-Datei.
- `stObj` (`logiBUS::utils::conversion::phys::NumericObjectPool_S`): VT-Objekt mit `u16ObjId`, `r32Scale`, `i32Offset`, `u8Decimals` (Default `u16ObjId := ID_NULL`, `r32Scale := 1.0`, `i32Offset := 0`, `u8Decimals := 0`).

Ausgangsvariablen

- `QO` (BOOL): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde; `AND` der `QO`-Ausgänge von `Q_NumericValue_PHYSA`, `NumericValue_PHYSA` und `INI_AR2` (über `AND_QO`).

Adapter

- **Plug `VALUEO`** (`adapter::types::unidirectional::AR`): Wert für einen weiteren Consumer.
- **Socket `OVERRIDE_ACTIVE`** (`adapter::types::unidirectional::AX`): bei dauerhaft TRUE wird der Override-Wert erzwungen.

Ereignisse

- `INIT` (EInit, Eingang): Initialisierungsanforderung; startet die Kette `NumericValue_PHYSA` ⟶ `INI_AR2` ⟶ `Q_NumericValue_PHYSA`.
- `INITO` (EInit, Ausgang): Initialisierungsbestätigung; kommt von `Q_NumericValue_PHYSA.INITO`.

Funktionsweise

1. `NumericValue_PHYSA` liest das VT-Eingabefeld als physikalischen Wert (`rPhys`).
2. `SEL_Override` (`AR_AX_SEL_AR`) wählt: `IN0` = Live-VT-Wert, `IN1` = Override-Wert aus `LIT_Override` (`initval_AR`, `INIT_VAL=REAL#0.0`), Auswahl über `G` = `OVERRIDE_ACTIVE`.
3. `AR_TO_AR2` wandelt in den bidirektionalen AR2-Adapter für `INI_AR2` (`eclipse4diac::storage::INI_AR2`; `SETM=TRUE`, `DEFAULT_VALUE=REAL#0.0`; `KEY`/`SECTION` von außen).

4. `AR_TO_AR2.AR_OUT` wird per `AR_SPLIT_2` verdoppelt: `OUT1` an `VALUEO`, `OUT2` an `Q_NumericValue_PHYSA` (VT-Anzeige, `stObj`).

`QI` wird an `NumericValue_PHYSA.QI`, `INI_AR2.QI` und `Q_NumericValue_PHYSA.QI` durchgereicht (kein fester `QI`=TRUE mehr). Die Initialisierung läuft als Kette `INIT` ⟶ `NumericValue_PHYSA.INIT`, `NumericValue_PHYSA.INITO` ⟶ `INI_AR2.INIT`, `INI_AR2.INITO` ⟶ `Q_NumericValue_PHYSA.INIT`, `Q_NumericValue_PHYSA.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`. `QO` ist das `AND` (`AND_QO`, `AND_BOOL_3`) der `QO`-Ausgänge von `Q_NumericValue_PHYSA`, `NumericValue_PHYSA` und `INI_AR2`.

Technische Besonderheiten

- Im Unterschied zur AUDI2-Variante (`INI_IN_AND_STORE_AUDI2_OVERRIDE`, Enum/Dropdown, 1=Deaktiviert) ist AR ein kontinuierlicher physikalischer Wert: Der Override-Wert hat keine feste Bedeutung, sondern ist über `INIT_VAL` an `LIT_Override` frei konfigurierbar.
- `SETM`=TRUE: Der Boot-Wert lädt automatisch (INIT ⟶ INITO ⟶ GET ⟶ GETO, unabhängig von `SETM`); `SETM`=TRUE sorgt zusätzlich dafür, dass auch eine Live-Änderung im laufenden Betrieb sofort bestätigt/zurückgemeldet wird (z. B. für die Weiterverwendung über `VALUEO`).

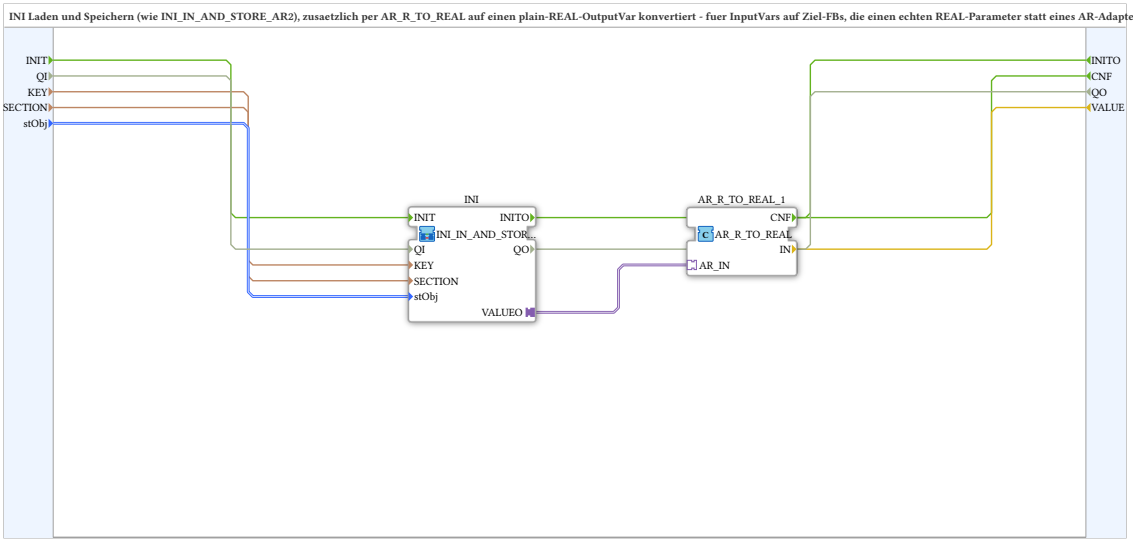
Zusammenfassung

AR2-Speicherbaustein mit zuschaltbarem, konfigurierbarem Override-Wert.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

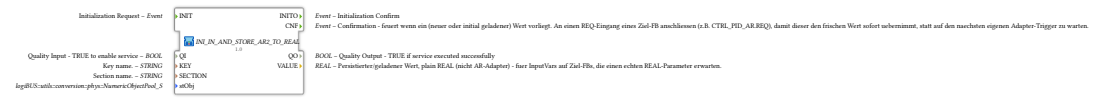
INI_IN_AND_STORE_AR2_TO_REAL



• – ○ • – ○ • – ○ •

Einleitung

`INI_IN_AND_STORE_AR2_TO_REAL` lädt und speichert einen Wert in der INI-Datei (wie `INI_IN_AND_STORE_AR2`) und konvertiert ihn zusätzlich per `AR_R_TO_REAL` auf eine einfache REAL-Ausgangsvariable. Das wird für Eingangsvariablen von Ziel-Bausteinen benötigt, die einen echten REAL-Parameter statt eines AR-Adapter-Sockets erwarten (z. B. `CTRL_PID_AR.KP/TN/TV`).



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; wird an `INI.INIT` weitergereicht

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung; kommt von `INI.INITO`
- **CNF** (Confirmation): feuert, wenn ein (neuer oder initial geladener) Wert vorliegt. An einen `REQ`-Eingang eines Ziel-FB anschließen (z. B. `CTRL_PID_AR.REQ`), damit dieser den frischen Wert sofort übernimmt, statt auf den nächsten eigenen Adapter-Trigger zu warten

Daten-Eingänge

- **QI** (BOOL, Initialwert `TRUE`): Quality Input, schaltet den Dienst frei; wird an `INI.QI` weitergereicht
- **KEY** (STRING): Schlüsselname in der INI-Datei
- **SECTION** (STRING): Sektionsname in der INI-Datei
- **stObj** (`logiBUS::utils::conversion::phys::NumericObjectPool_S`): Object-Pool-Eigenschaften der VT-Zahlenvariable (Default: `u16ObjId := ID_NULL, r32Scale := 1.0, i32Offset := 0, u8Decimals := 0`)

Daten-Ausgänge

- **QO** (BOOL): Quality Output; kommt von `INI.QO`
- **VALUE** (REAL): persistierter/geladener Wert als einfaches REAL (kein AR-Adapter)

Adapter

Es werden keine Adapter verwendet.

Funktionsweise

Das Netzwerk enthält zwei Instanzen:

- `INI` (`MyLib::sys::INI_IN_AND_STORE_AR2`): `KEY` ⟶ `INI.KEY`, `SECTION` ⟶ `INI.SECTION`, `stObj` ⟶ `INI.stObj`. Der Wert steht an `INI.VALUEO` (AR) bereit.
- `AR_R_TO_REAL_1` (`adapter::conversion::unidirectional::AR_R_TO_REAL`): `INI.VALUEO` ⟶ `AR_R_TO_REAL_1.AR_IN`; `AR_R_TO_REAL_1.IN` ⟶ `VALUE` und `AR_R_TO_REAL_1.CNF` ⟶ `CNF`.

Initialisierung und Qualität werden direkt durchgereicht: `INIT` ⟶ `INI.INIT`, `INI.INITO` ⟶ `INITO`, `QI` ⟶ `INI.QI` und `INI.QO` ⟶ `QO`.

Technische Besonderheiten

- Herausgelöst aus `Aufnahme_Automatik.SUB`, wo diese Kombination dreimal identisch (KP/TN/TV) verdrahtet war
- `CNF` entsteht erst nach der AR-zu-REAL-Konvertierung und signalisiert damit, dass `VALUE` aktuell ist

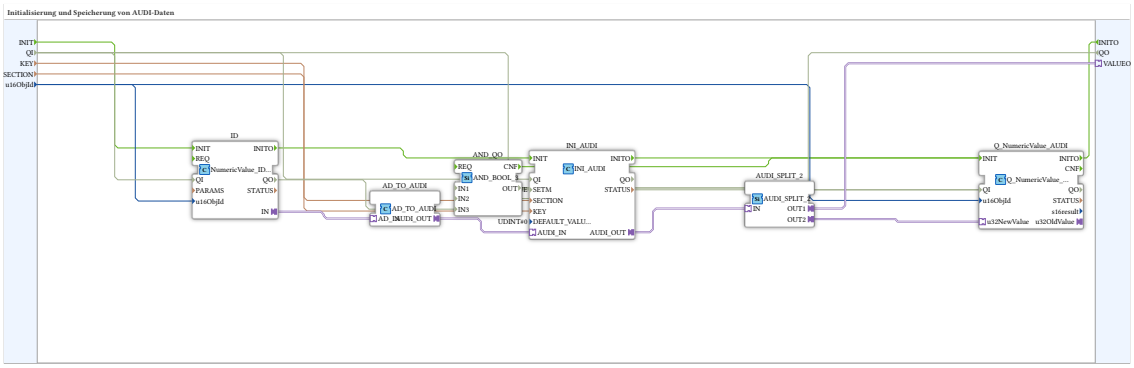
Anwendungsszenarien

- Persistierte Reglerparameter (z. B. KP/TN/TV eines PID-Reglers), die an REAL-Eingänge von Ziel-FBs angeschlossen werden

Fazit

`INI_IN_AND_STORE_AR2_TO_REAL` kombiniert INI-Persistenz mit einer AR-zu-REAL-Wandlung und liefert den Wert samt `CNF`-Ereignis an einer einfachen REAL-Ausgangsvariable.

INI_IN_AND_STORE_AUDI

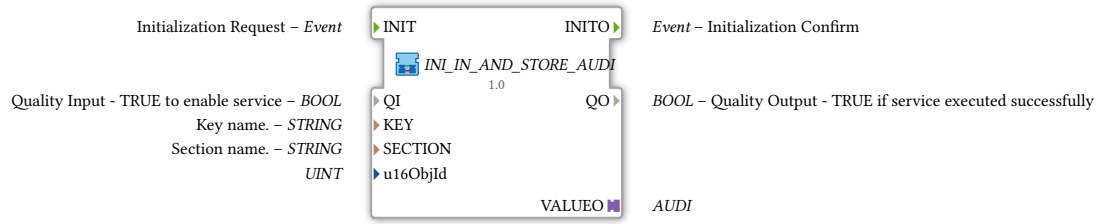


• – ○ • – ○ • – ○ •

Einleitung

`INI_IN_AND_STORE_AUDI` speichert einen über VT eingegebenen UDINT-Wert (AUDI-Adapter) persistent in einer INI-Datei.

Allgemeines Muster siehe INI_IN_AND_STORE / NVS_IN_AND_STORE (gemeinsames Muster).



Ereignis- und Qualitätsanschlüsse (INIT/INITO, QI/QO)

Der Baustein besitzt den Ereigniseingang `INIT` und den Ereignisausgang `INITO` (je `EInit`), den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`, Quality Input - TRUE aktiviert den Dienst) und den Datenausgang `QO` (`BOOL`, Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde).

Die Initialisierung läuft als Kette durch die inneren Bausteine: `INIT` ⟶ `ID.INIT`, `ID.INITO` ⟶ `INI_AUDI.INIT`, `INI_AUDI.INITO` ⟶ `Q_NumericValue_AUDI.INIT`, `Q_NumericValue_AUDI.INITO` ⟶ `INITO` (und ⟶ `AND_QO.REQ`). `QI` geht an `ID`, `INI_AUDI` und `Q_NumericValue_AUDI`. `QO` ist der Ausgang der Instanz `AND_QO` (`iec61131::booleanOperators::AND_BOOL_3`), die `Q_NumericValue_AUDI.QO`, `INI_AUDI.QO` und `ID.QO` verknüpft.

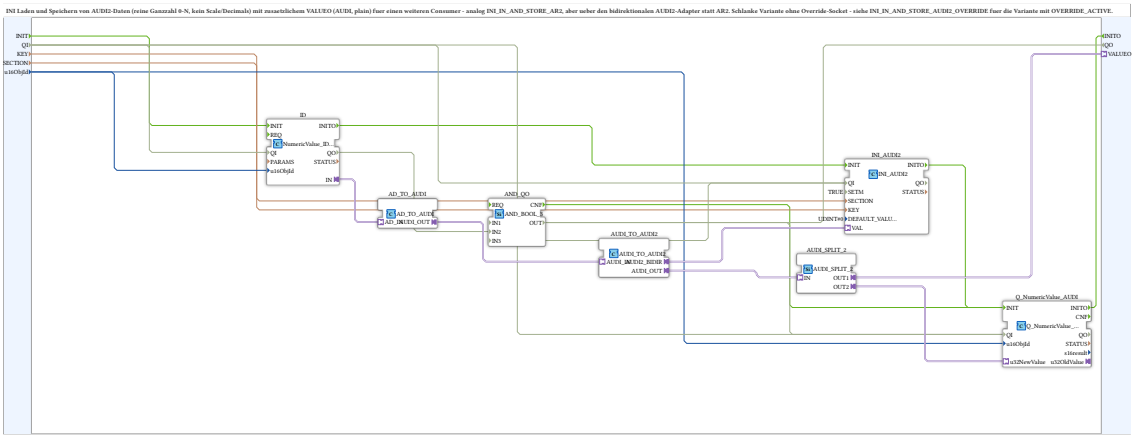
Zusammenfassung

AUDI-Variante (UDINT) der INI-Speicherfamilie.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

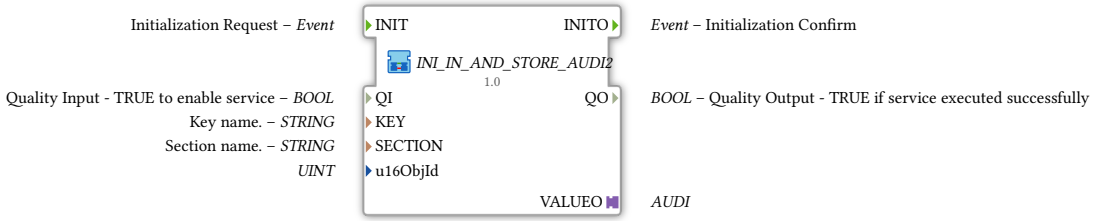
INI_IN_AND_STORE_AUDI2



• – ○ • – ○ • – ○ •

Einleitung

`INI_IN_AND_STORE_AUDI2` lädt und speichert `AUDI2`-Daten (reine Ganzzahl 0..N, ohne Skalierung/Nachkommastellen) persistent in einer INI-Datei und stellt zusätzlich einen Ausgang `VALUEO` (`AUDI`, plain) für einen weiteren Consumer bereit. Der Baustein ist analog zu `INI_IN_AND_STORE_AR2`, arbeitet aber über den bidirektionalen `AUDI2`-Adapter statt `AR2`. Das allgemeine Muster beschreibt INI_IN_AND_STORE / NVS_IN_AND_STORE eine Variante mit `AUDI` ist `INI_IN_AND_STORE_AUDI`.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input - TRUE aktiviert den Dienst
- **KEY** (`STRING`): Schlüsselname in der INI-Datei
- **SECTION** (`STRING`): Abschnittsname in der INI-Datei
- **u16ObjId** (`UINT`): Object ID der VT-Zahlenvariable (Default `ID_NULL`)

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde (UND aus `Q_NumericValue_AUDI.QO`, `ID.QO` und `INI_AUDI2.QO`)

Adapter

- **VALUEO** (Plug, `adapter::types::unidirectional::AUDI`): Wert für einen weiteren Consumer

Verwendete Funktionsbausteine (FBs)

- **`ID`** (`isobus::UT::io::NumericValue::NumericValue_IDA`): liefert den vom Bediener eingegebenen VT-Wert
- **`AD_TO_AUDI`** (`adapter::conversion::unidirectional::AD_TO_AUDI`): wandelt den `AD`-Wert in `AUDI`
- **`AUDI_TO_AUDI2`** (`adapter::conversion::bidirectional::AUDI_TO_AUDI2`): wandelt `AUDI` in den bidirektionalen `AUDI2`-Adapter
- **`INI_AUDI2`** (`eclipse4diac::storage::INI_AUDI2`): `SETM = TRUE`, `DEFAULT_VALUE = UDINT#0`; lädt und speichert den Wert
- **`AUDI_SPLIT_2`** (`adapter::events::unidirectional::AUDI_SPLIT_2`): verdoppelt den Wert
- **`Q_NumericValue_AUDI`** (`isobus::UT::Q::Q_NumericValue_AUDI`): schreibt den Wert in die VT-Zahlenvariable
- **`AND_QO`** (`iec61131::booleanOperators::AND_BOOL_3`): verknüpft die `QO`-Ausgänge von `Q_NumericValue_AUDI`, `ID` und `INI_AUDI2`

Funktionsweise

- `ID.IN` ⟶ `AD_TO_AUDI.AD_IN` ⟶ `AUDI_TO_AUDI2.AUDI_IN`
- `AUDI_TO_AUDI2.AUDI2_BIDIR` ⟶ `INI_AUDI2.VAL` (Laden/Speichern)
- `AUDI_TO_AUDI2.AUDI_OUT` ⟶ `AUDI_SPLIT_2.IN`
- `AUDI_SPLIT_2.OUT1` ⟶ `VALUEO`; `AUDI_SPLIT_2.OUT2` ⟶ `Q_NumericValue_AUDI.u32NewValue` (VT-Anzeige)
- `KEY`, `SECTION` ⟶ `INI_AUDI2`; `u16ObjId` ⟶ `ID` und `Q_NumericValue_AUDI`
- `INIT` ⟶ `ID.INIT`; `ID.INITO` ⟶ `INI_AUDI2.INIT`; `INI_AUDI2.INITO` ⟶ `Q_NumericValue_AUDI.INIT`; `Q_NumericValue_AUDI.INITO` ⟶ `INITO` (und ⟶ `AND_QO.REQ`)
- `QI` ⟶ `Q_NumericValue_AUDI.QI`, `ID.QI` und `INI_AUDI2.QI`
- `Q_NumericValue_AUDI.QO` ⟶ `AND_QO.IN1`; `ID.QO` ⟶ `AND_QO.IN2`; `INI_AUDI2.QO` ⟶ `AND_QO.IN3`; `AND_QO.OUT` ⟶ `QO`

Technische Besonderheiten

- **`SETM = TRUE`:** Der Boot-Wert wird ohnehin automatisch geladen (`INIT` ⟶ `INITO` ⟶ `GET` ⟶ `GETO`, unabhängig von `SETM`). `SETM = TRUE` sorgt dafür, dass auch eine Live-Änderung im laufenden Betrieb sofort bestätigt bzw. zurückgemeldet wird; wichtig, wenn der frische Wert im selben Lauf weiterverwendet werden soll (hier über `VALUEO`).
- `INI_AUDI2` besitzt keinen eigenen Ausgangs-Plug, daher wird der Ausgang per `AUDI_SPLIT_2` verdoppelt
- Reine Ganzzahl (`UDINT`) ohne Skalierung

Anwendungsszenarien

- Persistent gespeicherte Ganzzahl-Einstellung (z. B. Zähler, Auswahlindex), die am VT eingegeben und weiterverwendet wird

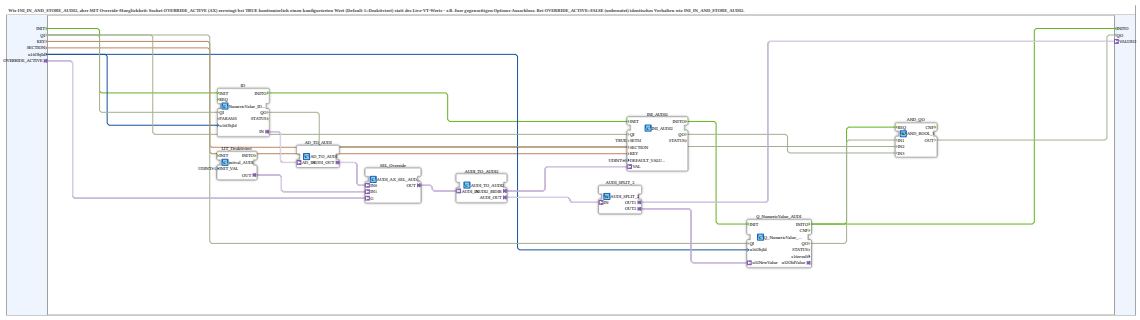
Zusammenfassung

`INI_IN_AND_STORE_AUDI2` speichert einen VT-Ganzzahlwert in einer INI-Datei (über `AUDI2`) und gibt ihn zusätzlich als `AUDI` aus.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

INI_IN_AND_STORE_AUDI2_OVERRIDE



• – ○ • – ○ • – ○ •

Einleitung

`INI_IN_AND_STORE_AUDI2_OVERRIDE` ist wie `INI_IN_AND_STORE_AUDI2`, jedoch mit Override-Möglichkeit: Ist der Socket `OVERRIDE_ACTIVE` (AX) dauerhaft TRUE, wird statt des Live-VT-Werts der Wert 1 (Deaktiviert) erzwungen - z. B. für gegenseitigen Optionsausschluss. Bei `OVERRIDE_ACTIVE`=FALSE (unbenutzt) ist das Verhalten identisch zu `INI_IN_AND_STORE_AUDI2`. Die Variante ohne `OVERRIDE_ACTIVE` ist als `INI_IN_AND_STORE_AUDI2` separat verfügbar (Umbenennung in Version 1.2).

Allgemeines Muster siehe INI_IN_AND_STORE / NVS_IN_AND_STORE (gemeinsames Muster).

Schnittstelle

Ereignis-Eingänge

- `INIT` (`EInit`): Initialization Request.

Ereignis-Ausgänge

- `INITO` (`EInit`): Initialization Confirm.

Eingangsvariablen

- `QI` (BOOL, Vorgabe `TRUE`): Quality Input - TRUE aktiviert den Dienst.
- `KEY` (STRING): Schlüsselname in der INI-Datei.
- `SECTION` (STRING): Abschnittsname in der INI-Datei.

- `u16ObjId` (UINT, Default `ID_NULL`): ObjectID des VT-Eingabefelds.

Ausgangsvariablen

- `QO` (BOOL): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde (UND-Verknüpfung `AND_QO` der `QO`-Ausgänge der inneren Bausteine).

Adapter

- **Plug `VALUEO`** (`adapter::types::unidirectional::AUDI`): Wert für einen weiteren Consumer.
- **Socket `OVERRIDE_ACTIVE`** (`adapter::types::unidirectional::AX`): bei dauerhaft TRUE wird VAL=1 (Deaktiviert) erzwungen.

Die Ereignisse `INIT`/`INITO` werden als Kette durch die inneren Bausteine geführt (siehe Funktionsweise).

Funktionsweise

1. `ID` (`NumericValue_IDA`) liest das VT-Eingabefeld; `AD_TO_AUDI` wandelt in AUDI.
2. `SEL_Override` (`AUDI_AX_SEL_AUDI`) wählt: `IN0` = Live-VT-Wert, `IN1` = Override-Wert aus `LIT_Deaktiviert` (`initval_AUDI`, `INIT_VAL=UDINT#1`), Auswahl über `G` = `OVERRIDE_ACTIVE`.
3. `AUDI_TO_AUDI2` wandelt in den bidirektionalen AUDI2-Adapter für `INI_AUDI2` (`eclipse4diac::storage::INI_AUDI2`; `SETM`=TRUE, `DEFAULT_VALUE=UDINT#0`).
4. `AUDI_TO_AUDI2.AUDI_OUT` wird per `AUDI_SPLIT_2` verdoppelt: `OUT1` an `VALUEO`, `OUT2` an `Q_NumericValue_AUDI` (`u32NewValue`, VT-Anzeige).

Initialisierung: `INIT` ⟶ `ID.INIT`; `ID.INITO` ⟶ `INI_AUDI2.INIT`; `INI_AUDI2.INITO` ⟶ `Q_NumericValue_AUDI.INIT`; `Q_NumericValue_AUDI.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`. Qualität: `QI` ⟶ `Q_NumericValue_AUDI.QI`, `ID.QI`, `INI_AUDI2.QI`; `Q_NumericValue_AUDI.QO`, `ID.QO`, `INI_AUDI2.QO` ⟶ `AND_QO.IN1` bis `AND_QO.IN3` (in dieser Reihenfolge); `AND_QO.OUT` ⟶ `QO` (`AND_QO`: `iec61131::booleanOperators::AND_BOOL_3`).

Technische Besonderheiten

- Der Override läuft über denselben `INI_AUDI2`/`Q_NumericValue_AUDI`-Pfad: `Q_NumericValue` erlaubt laut eigener Doku nur eine Instanz je ObjectID, eine zweite Schreibinstanz ist daher nicht möglich.
- Für reine Enum-/Ganzzahlwerte (z. B. Dropdowns 0-3), keine Dezimalstellen.
- `SETM`=TRUE: Der Boot-Wert lädt automatisch (INIT ⟶ INITO ⟶ GET ⟶ GETO); `SETM`=TRUE sorgt zusätzlich dafür, dass auch eine Live-Änderung sofort bestätigt/zurückgemeldet wird.

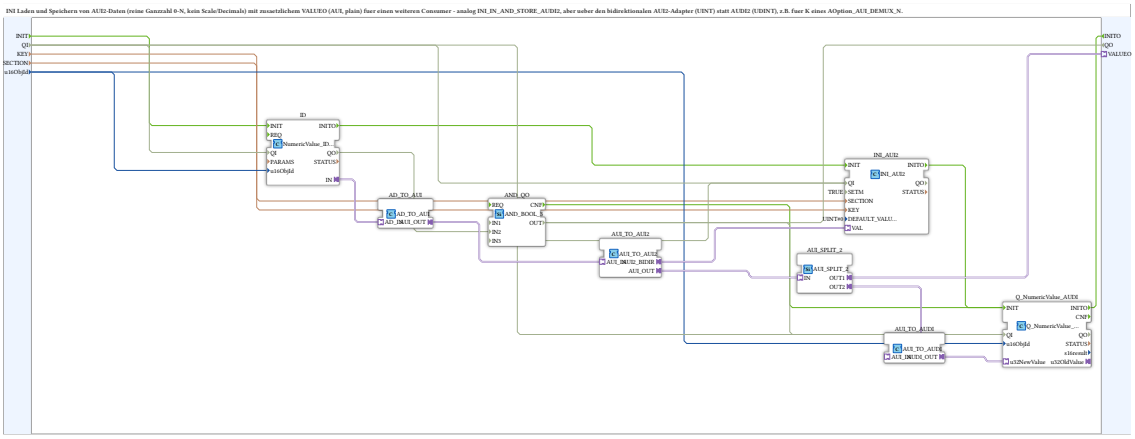
Zusammenfassung

AUDI2-Speicherbaustein mit zuschaltbarem Override auf 1 (Deaktiviert), z. B. für gegenseitigen Optionsausschluss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

INI_IN_AND_STORE_AUI2

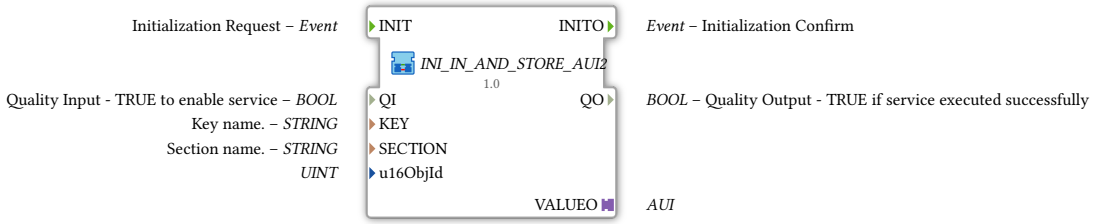


• – ○ • – ○ • – ○ •

Einleitung

`INI_IN_AND_STORE_AUI2` lädt und speichert AUI2-Daten (reine Ganzzahl 0-N, kein Scale/Decimals) in einer INI-Datei und stellt zusätzlich `VALUEO` (AUI, plain) für einen weiteren Consumer bereit. Er entspricht `INI_IN_AND_STORE_AUDI2`, nutzt aber den bidirektionalen AUI2-Adapter (UINT) statt AUDI2 (UDINT) - z. B. für `K` eines `AOption_AUI_DEMUX_N`.

Allgemeines Muster siehe INI_IN_AND_STORE / NVS_IN_AND_STORE (gemeinsames Muster).



Schnittstelle

Eingangsvariablen

- `QI` (BOOL, Vorgabe `TRUE`): Quality Input - TRUE aktiviert den Dienst.
- `KEY` (STRING): Schlüsselname in der INI-Datei.
- `SECTION` (STRING): Abschnittsname in der INI-Datei.
- `u16ObjId` (UINT, Default `ID_NULL`): ObjectID des VT-Eingabefelds.

Ausgangsvariablen

- `QO` (BOOL): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde (UND von `Q_NumericValue_AUDI.QO`, `ID.QO` und `INI_AUI2.QO`).

Adapter

- **Plug `VALUEO`** (`adapter::types::unidirectional::AUI`): Wert für einen weiteren Consumer.

Ereignisse

- Eingang `INIT` (`EInit`): Initialization Request
- Ausgang `INITO` (`EInit`): Initialization Confirm

Funktionsweise

1. `ID` (`NumericValue_IDA`) liest das VT-Eingabefeld; `AD_TO_AUI` wandelt in AUI.
2. `AUI_TO_AUI2` wandelt in den bidirektionalen AUI2-Adapter für `INI_AUI2` (`eclipse4diac::storage::INI_AUI2`; `SETM`=TRUE, `DEFAULT_VALUE=UINT#0`).
3. `AUI_TO_AUI2.AUI_OUT` wird per `AUI_SPLIT_2` verdoppelt: `OUT1` an `VALUEO`, `OUT2` über `AUI_TO_AUDI` an `Q_NumericValue_AUDI` (`u32NewValue`, VT-Anzeige).

`QI` wird an `ID.QI`, `INI_AUI2.QI` und `Q_NumericValue_AUDI.QI` durchgereicht. Die Initialisierungskette lautet `INIT` → `ID.INIT`, `ID.INITO` → `INI_AUI2.INIT`, `INI_AUI2.INITO` → `Q_NumericValue_AUDI.INIT`, `Q_NumericValue_AUDI.INITO` → `INITO` (und zusätzlich → `AND_QO.REQ`). `QO` ist die UND-Verknüpfung (`AND_QO`, Typ `iec61131::booleanOperators::AND_BOOL_3`) von `Q_NumericValue_AUDI.QO`, `ID.QO` und `INI_AUI2.QO`.

Technische Besonderheiten

- Strukturell identisch zu `INI_IN_AND_STORE_AUDI2`, nur mit AUI2 statt AUDI2.
- `SETM`=TRUE: Der Boot-Wert lädt automatisch (INIT → INITO → GET → GETO); `SETM`=TRUE sorgt zusätzlich dafür, dass auch eine Live-Änderung sofort bestätigt/zurückgemeldet wird (z. B. für die Weiterverwendung über `VALUEO`).

Zusammenfassung

AUI2-Variante der INI-Speicherfamilie für reine Ganzzahlen mit zusätzlichem `VALUEO`-Ausgang.

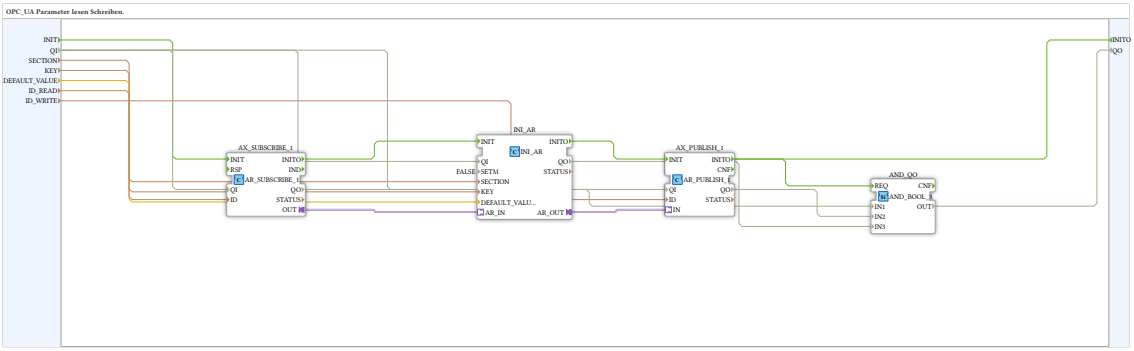
🛠️ Zugehörige Übungen

- Uebung_094b4_AX

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

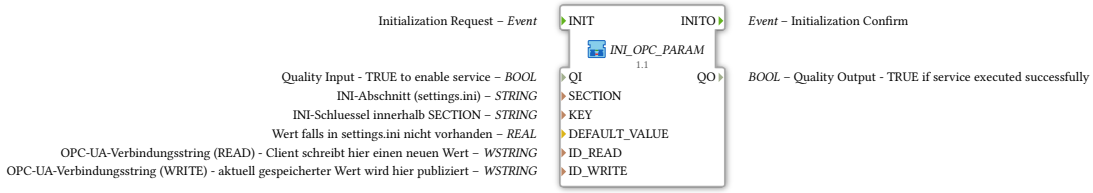
INI_OPC_PARAM



• – ○ • – ○ • – ○ •

Einleitung

`INI_OPC_PARAM` liest und schreibt einen REAL-Parameter per OPC-UA und speichert ihn remanent in der INI-Datei (`settings.ini`). Ein Client kann über `ID_READ` einen neuen Wert setzen; der aktuell gespeicherte Wert wird über `ID_WRITE` veröffentlicht. Nach einem Neustart wird der zuletzt gespeicherte Wert (oder `DEFAULT_VALUE`, falls noch keiner vorhanden ist) automatisch geladen und wieder publiziert.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: INI_OPC_PARAM

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **AX_SUBSCRIBE_1**: `adapter::net::AR_SUBSCRIBE_1` — abonniert neue Werte vom OPC-UA-Client (`ID_READ`).
  - **INI_AR**: `eclipse4diac::storage::INI_AR` (`SETM=FALSE`) — speichert den Wert remanent in der INI-Datei unter `SECTION`/`KEY`, mit `DEFAULT_VALUE` als Startwert.
  - **AX_PUBLISH_1**: `adapter::net::AR_PUBLISH_1` — veröffentlicht den aktuellen Wert per OPC-UA (`ID_WRITE`).
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_3` — verknüpft die `QO`-Ausgänge von `AX_SUBSCRIBE_1`, `AX_PUBLISH_1` und `INI_AR` zum SubApp-Ausgang `QO`.
- **Schnittstelle**: Ereigniseingang `INIT`, Ereignisausgang `INITO`, Dateneingang `QI` (Initialwert `TRUE`, an alle drei FBs durchgereicht) und Datenausgang `QO` (`AND` der drei `QO`-Ausgänge).
- **Funktionsweise**: Ein per OPC-UA empfangener Wert wird an `INI_AR` weitergereicht, dort gespeichert, und der (neu empfangene oder beim Start geladene) Wert wird per `AX_PUBLISH_1` erneut veröffentlicht.

Programmablauf und Verbindungen

1. **Initialisierungskette**: `INIT` ⟶ `AX_SUBSCRIBE_1.INIT`; `AX_SUBSCRIBE_1.INITO` ⟶ `INI_AR.INIT`; `INI_AR.INITO` ⟶ `AX_PUBLISH_1.INIT`; `AX_PUBLISH_1.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`.
2. **Qualität**: `QI` ⟶ `AX_SUBSCRIBE_1.QI`, `AX_PUBLISH_1.QI` und `INI_AR.QI`; `AX_SUBSCRIBE_1.QO` ⟶ `AND_QO.IN1`, `AX_PUBLISH_1.QO` ⟶ `AND_QO.IN2`, `INI_AR.QO` ⟶ `AND_QO.IN3`; `AND_QO.OUT` ⟶ `QO`.
3. **Parameter**: `SECTION` ⟶ `INI_AR.SECTION`; `KEY` ⟶ `INI_AR.KEY`; `DEFAULT_VALUE` ⟶ `INI_AR.DEFAULT_VALUE`; `ID_READ` ⟶ `AX_SUBSCRIBE_1.ID`; `ID_WRITE` ⟶ `AX_PUBLISH_1.ID`.
4. **Adapterkette**: `AX_SUBSCRIBE_1.OUT` ⟶ `INI_AR.AR_IN` (neuer Wert vom Client) ⟶ `INI_AR.AR_OUT` ⟶ `AX_PUBLISH_1.IN` (gespeicherter/aktueller Wert zurück an den Client).

Technische Besonderheiten

- **`SETM=FALSE`**: `INI_AR` speichert nur bei explizitem Schreibereignis über `AR_IN`, nicht bei jedem Zyklus.
- **Sequentielle Initialisierung**: `AX_SUBSCRIBE_1.INITO` ⟶ `INI_AR.INIT` ⟶ `INI_AR.INITO` ⟶ `AX_PUBLISH_1.INIT` stellt sicher, dass der gespeicherte Wert erst geladen wird, bevor er veröffentlicht wird.
- **Kein eigener Adapter-Ausgang**: Im Gegensatz zu `INI_OPC_PARAM_AR` stellt dieser Baustein den gespeicherten Wert nicht als Plug für andere Bausteine im aufrufenden Netzwerk bereit — er dient ausschließlich der OPC-UA-Anbindung.

Anwendungsszenarien

- Ein per OPC-UA/Web-Client editierbarer Zahlenparameter (z. B. Sollwert), der über Neustarts hinweg erhalten bleiben muss, aber im lokalen Netzwerk nicht weiterverwendet wird.

Vergleich mit ähnlichen Bausteinen

`INI_OPC_PARAM_AR` erweitert diesen Baustein um einen zusätzlichen `AR`-Plug (`OUT`) für die lokale Weiterverwendung des gespeicherten Werts. `INI_OPC_PARAM_ATM` liefert zusätzlich eine `TIME`-Umrechnung, `INI_OPC_PARAM_AX` ist die BOOL-Variante mit `AX`-Ausgang.

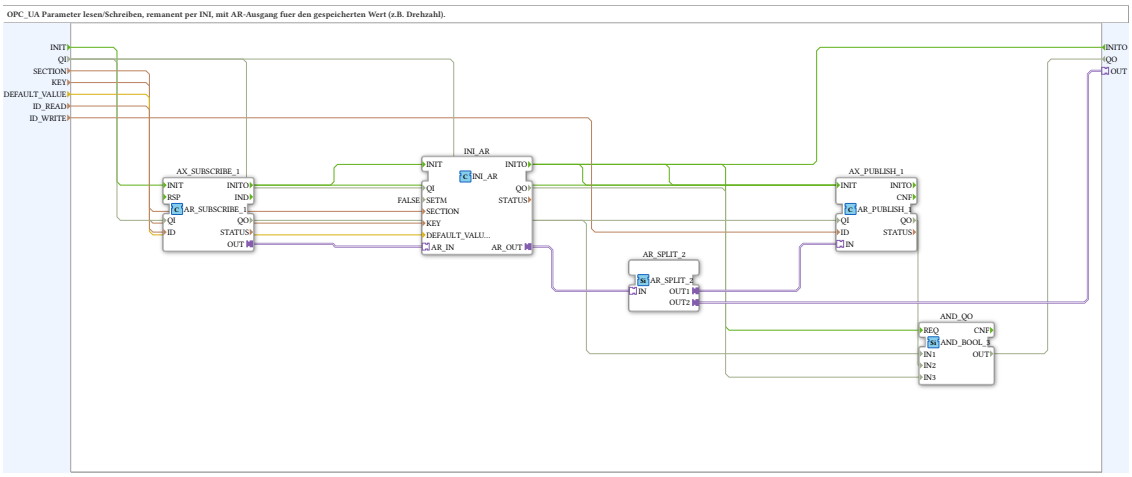
Zusammenfassung

`INI_OPC_PARAM` verbindet OPC-UA-Lese-/Schreibzugriff mit remanenter INI-Speicherung eines REAL-Parameters, ohne den Wert lokal als Adapter bereitzustellen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

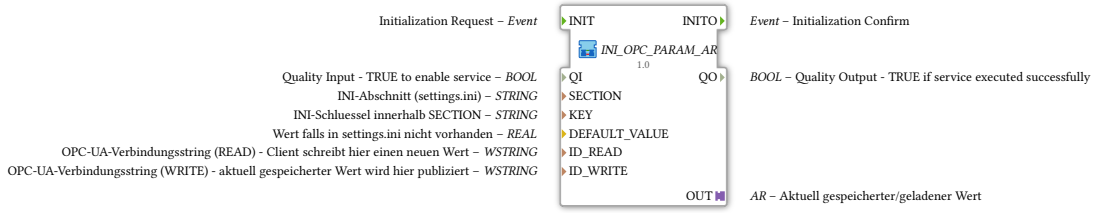
INI_OPC_PARAM_AR



• – ○ • – ○ • – ○ •

Einleitung

`INI_OPC_PARAM_AR` erweitert `INI_OPC_PARAM` um einen zusätzlichen `AR`-Adapter-Ausgang (`OUT`), der den aktuell gespeicherten/geladenen Wert direkt für die Weiterverwendung im aufrufenden Netzwerk bereitstellt — z. B. für eine lokal einzustellende Drehzahl, die sowohl per OPC-UA editierbar sein als auch direkt in derselben Applikation verwendet werden soll.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: INI_OPC_PARAM_AR

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **AX_SUBSCRIBE_1**: `adapter::net::AR_SUBSCRIBE_1` (`QI` kommt vom Eingang `QI`) — abonniert neue Werte vom OPC-UA-Client (`ID_READ`).
  - **INI_AR**: `eclipse4diac::storage::INI_AR` (`QI` kommt vom Eingang `QI`, `SETM=FALSE`) — speichert den Wert remanent in der INI-Datei unter `SECTION`/`KEY`, mit `DEFAULT_VALUE` als Startwert.
  - **AX_PUBLISH_1**: `adapter::net::AR_PUBLISH_1` (`QI` kommt vom Eingang `QI`) — veröffentlicht den aktuellen Wert per OPC-UA (`ID_WRITE`).
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_3` — verknüpft die `QO`-Ausgänge der drei FBs zum Ausgang `QO`.
- **Funktionsweise**: Identisch zu `INI_OPC_PARAM`, zusätzlich wird `INI_AR.AR_OUT` direkt auf den Plug `OUT` geführt. Die SubApp-Schnittstelle hat außerdem die Ereignisse `INIT`/`INITO` und die Daten `QI` (BOOL, Initialwert `TRUE`) und `QO` (BOOL).

Programmablauf und Verbindungen

1. **Initialisierungskette**: `INIT` ⟶ `AX_SUBSCRIBE_1.INIT`; `AX_SUBSCRIBE_1.INITO` ⟶ `INI_AR.INIT` und ⟶ `AX_PUBLISH_1.INIT`; `AX_PUBLISH_1.INITO` ⟶ `INI_AR.INIT`; `INI_AR.INITO` ⟶ `AX_PUBLISH_1.INIT`, ⟶ `INITO` und ⟶ `AND_QO.REQ`.
2. **Parameter**: `SECTION` ⟶ `INI_AR.SECTION`; `KEY` ⟶ `INI_AR.KEY`; `DEFAULT_VALUE` ⟶ `INI_AR.DEFAULT_VALUE`; `ID_READ` ⟶ `AX_SUBSCRIBE_1.ID`; `ID_WRITE` ⟶ `AX_PUBLISH_1.ID`.
3. **Adapterkette**: `AX_SUBSCRIBE_1.OUT` ⟶ `INI_AR.AR_IN` ⟶ `INI_AR.AR_OUT` ⟶ `AX_PUBLISH_1.IN` **und** ⟶ `OUT` (Plug an die SubApp-Schnittstelle).
4. **Qualität**: `QI` ⟶ `AX_SUBSCRIBE_1.QI`, `AX_PUBLISH_1.QI` und `INI_AR.QI`; `AX_SUBSCRIBE_1.QO` ⟶ `AND_QO.IN1`, `AX_PUBLISH_1.QO` ⟶ `AND_QO.IN2`, `INI_AR.QO` ⟶ `AND_QO.IN3`; `AND_QO.OUT` ⟶ `QO`.

Technische Besonderheiten

- **Ein Ausgang, zwei Ziele**: `INI_AR.AR_OUT` speist gleichzeitig den OPC-UA-Publish-Adapter und den externen `OUT`-Plug — beide erhalten denselben Wert ohne zusätzlichen Split, da es sich um zwei getrennte Adapterverbindungen von derselben Quelle handelt.
- Ansonsten identisch zu `INI_OPC_PARAM` (siehe dort für Details zu `SETM`/Initialisierungsreihenfolge).

Anwendungsszenarien

- Ein per OPC-UA/Web-Client editierbarer und remanent gespeicherter REAL-Parameter (z. B. Drehzahl-Sollwert), der zusätzlich direkt an andere Bausteine im selben Netzwerk weitergegeben werden muss.

Vergleich mit ähnlichen Bausteinen

Gegenüber `INI_OPC_PARAM` kommt nur der zusätzliche `AR`-Plug `OUT` hinzu. `INI_OPC_PARAM_ATM` rechnet den Wert zusätzlich in `TIME` um, `INI_OPC_PARAM_AX` ist die BOOL-Variante.

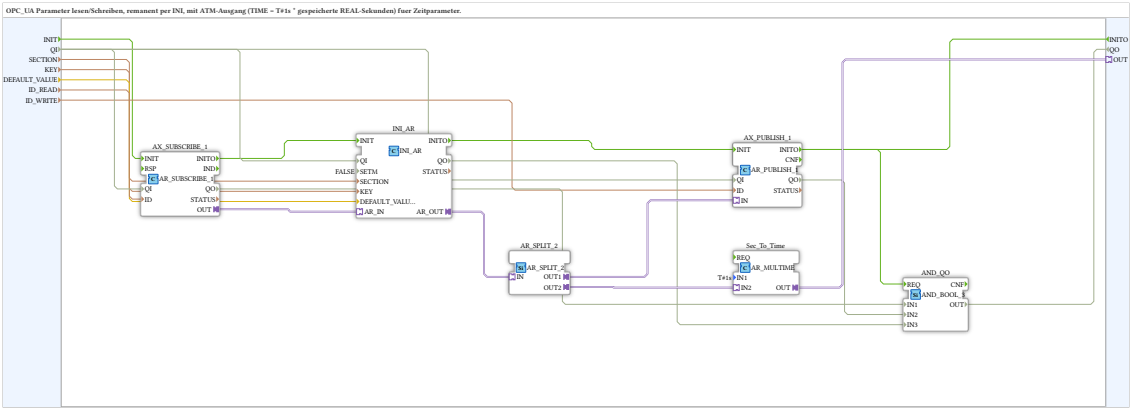
Zusammenfassung

`INI_OPC_PARAM_AR` ist `INI_OPC_PARAM` mit zusätzlichem `AR`-Ausgang für die lokale Weiterverwendung des remanent gespeicherten Parameters.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

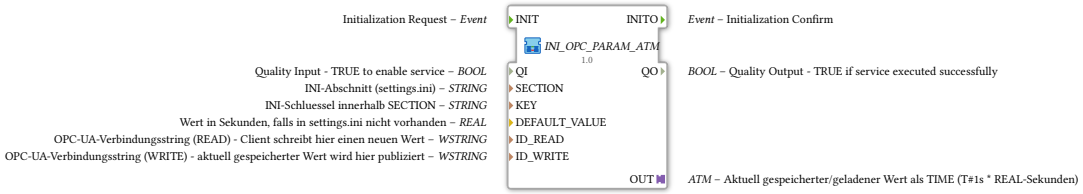
INI_OPC_PARAM_ATM



• – ○ • – ○ • – ○ •

Einleitung

`INI_OPC_PARAM_ATM` ist die Zeitparameter-Variante von `INI_OPC_PARAM_AR`: Der remanent gespeicherte REAL-Wert wird zusätzlich in einen `ATM`-Adapter (`TIME`) umgerechnet — `DEFAULT_VALUE` und der gespeicherte Wert werden dabei als Sekunden interpretiert (`T#1s * Sekunden`).



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: INI_OPC_PARAM_ATM

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **AX_SUBSCRIBE_1**: `adapter::net::AR_SUBSCRIBE_1` (`QI` von außen) — abonniert neue Werte vom OPC-UA-Client (`ID_READ`).
  - **INI_AR**: `eclipse4diac::storage::INI_AR` (`SETM=FALSE`, `QI` von außen) — speichert den Wert remanent in der INI-Datei unter `SECTION`/`KEY`, mit `DEFAULT_VALUE` (in Sekunden) als Startwert.
  - **AX_PUBLISH_1**: `adapter::net::AR_PUBLISH_1` (`QI` von außen) — veröffentlicht den aktuellen REAL-Wert per OPC-UA (`ID_WRITE`).
  - **Sec_To_Time**: `adapter::iec61131::arithmetic::AR_MULTIME` (`IN1=T#1s`) — multipliziert den gespeicherten REAL-Wert (Sekunden) mit `T#1s`, um ihn als `TIME`-Wert bereitzustellen.
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_3` — verknüpft die `QO` von `AX_SUBSCRIBE_1`, `AX_PUBLISH_1` und `INI_AR` zum Ausgang `QO`.
- **Funktionsweise**: Wie `INI_OPC_PARAM_AR`, zusätzlich wird `INI_AR.AR_OUT` über `Sec_To_Time` (`AR_MULTIME`) in `TIME` umgerechnet und als `ATM`-Plug `OUT` bereitgestellt.

Programmablauf und Verbindungen

1. **Initialisierungskette**: Ereigniseingang `INIT` ⟶ `AX_SUBSCRIBE_1.INIT`; `AX_SUBSCRIBE_1.INITO` ⟶ `INI_AR.INIT`; `INI_AR.INITO` ⟶ `AX_PUBLISH_1.INIT`; `AX_PUBLISH_1.INITO` ⟶ Ereignisausgang `INITO` und ⟶ `AND_QO.REQ`.
2. **Parameter**: `SECTION` ⟶ `INI_AR.SECTION`; `KEY` ⟶ `INI_AR.KEY`; `DEFAULT_VALUE` (Sekunden) ⟶ `INI_AR.DEFAULT_VALUE`; `ID_READ` ⟶ `AX_SUBSCRIBE_1.ID`; `ID_WRITE` ⟶ `AX_PUBLISH_1.ID`.
3. **Adapterkette**: `AX_SUBSCRIBE_1.OUT` ⟶ `INI_AR.AR_IN` ⟶ `INI_AR.AR_OUT` ⟶ `AX_PUBLISH_1.IN` (REAL, unverändert per OPC-UA veröffentlicht) **und** ⟶ `Sec_To_Time.IN2` ⟶ `Sec_To_Time.OUT` ⟶ `OUT` (als `TIME`).
4. **Qualität**: Dateneingang `QI` (`BOOL`, Default `TRUE`) ⟶ `AX_SUBSCRIBE_1.QI`, `AX_PUBLISH_1.QI` und `INI_AR.QI`; deren `QO` ⟶ `AND_QO.IN1`, `AND_QO.IN2`, `AND_QO.IN3`; `AND_QO.OUT` ⟶ Datenausgang `QO`.

Technische Besonderheiten

- **`AR_MULTIME` mit festem `IN1=T#1s`**: Die Multiplikation `T#1s * REAL-Sekunden` wandelt den gespeicherten Zahlenwert direkt in einen `TIME`-Wert um, ohne dass der Aufrufer selbst umrechnen muss.
- **OPC-UA-Publish bleibt REAL**: `ID_WRITE` veröffentlicht weiterhin den ursprünglichen REAL-Wert (Sekunden) — nur der zusätzliche `OUT`-Plug liefert `TIME`.
- Ansonsten identisch zu `INI_OPC_PARAM_AR`.

Anwendungsszenarien

- Remanent gespeicherte Zeitparameter (z. B. Verzögerungs- oder Timeout-Zeiten in Sekunden), die per OPC-UA als einfache Zahl editierbar sein sollen, aber intern als `TIME`-Wert (z. B. für einen `E_DELAY` oder `timeOut`-Adapter) gebraucht werden.

Vergleich mit ähnlichen Bausteinen

Gegenüber `INI_OPC_PARAM_AR` kommt nur die zusätzliche `Sec_To_Time`-Umrechnung (`AR_MULTIME`) hinzu; der `OUT`-Plug ist hier vom Typ `ATM` statt `AR`. `INI_OPC_PARAM_AX` ist die BOOL-Variante.

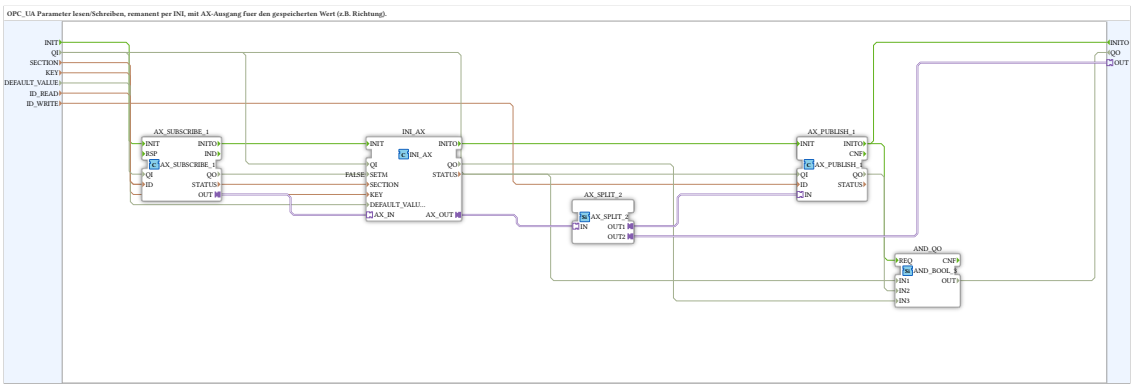
Zusammenfassung

`INI_OPC_PARAM_ATM` liefert einen remanent gespeicherten, per OPC-UA editierbaren Zeitparameter als fertigen `TIME`-Adapter, berechnet aus einem in Sekunden gespeicherten REAL-Wert.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

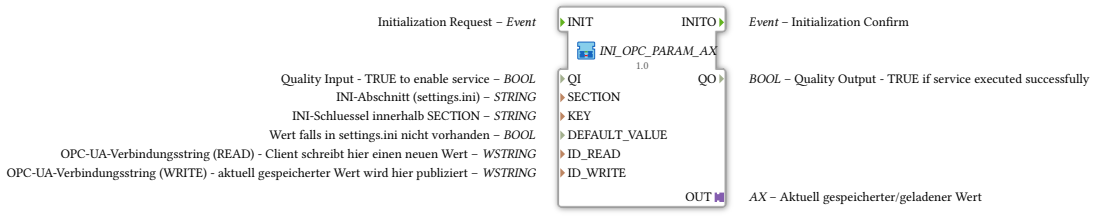
INI_OPC_PARAM_AX



• — ○ • — ○ • — ○ •

Einleitung

`INI_OPC_PARAM_AX` ist die BOOL-Variante von `INI_OPC_PARAM_AR`: Statt eines REAL-Werts wird ein BOOL-Parameter (z. B. eine Richtung oder ein Betriebsmodus-Schalter) per OPC-UA gelesen/geschrieben, remanent in der INI-Datei gespeichert und als `AX`-Adapter für die lokale Weiterverwendung bereitgestellt.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: INI_OPC_PARAM_AX

- **Typ**: SubAppType
- **Verwendete interne FBs**:

  - **AX_SUBSCRIBE_1**: `adapter::net::AX_SUBSCRIBE_1` — abonniert neue BOOL-Werte vom OPC-UA-Client (`ID_READ`).
  - **INI_AX**: `eclipse4diac::storage::INI_AX` (`SETM=FALSE`) — speichert den BOOL-Wert remanent in der INI-Datei unter `SECTION`/`KEY`, mit `DEFAULT_VALUE` als Startwert.
  - **AX_PUBLISH_1**: `adapter::net::AX_PUBLISH_1` — veröffentlicht den aktuellen Wert per OPC-UA (`ID_WRITE`).
  - **AX_SPLIT_2**: `adapter::events::unidirectional::AX_SPLIT_2` — verzweigt den Ausgang von `INI_AX` auf `AX_PUBLISH_1` und `OUT`.
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_3` — verknüpft die `QO`-Ausgänge von `AX_SUBSCRIBE_1`, `AX_PUBLISH_1` und `INI_AX` zum `QO` der SubApp.
- **Funktionsweise**: Strukturell identisch zu `INI_OPC_PARAM_AR`, jedoch mit `AX`-Adaptern (`AX_SUBSCRIBE_1`/`INI_AX`/`AX_PUBLISH_1`) statt `AR`-Adaptern, passend zum BOOL-Datentyp von `DEFAULT_VALUE`.

Programmablauf und Verbindungen

1. **Initialisierungskette**: `INIT` ⟶ `AX_SUBSCRIBE_1.INIT`; `AX_SUBSCRIBE_1.INITO` ⟶ `INI_AX.INIT`; `INI_AX.INITO` ⟶ `AX_PUBLISH_1.INIT`; `AX_PUBLISH_1.INITO` ⟶ `INITO` (und ⟶ `AND_QO.REQ`).
2. **Parameter**: `SECTION` ⟶ `INI_AX.SECTION`; `KEY` ⟶ `INI_AX.KEY`; `DEFAULT_VALUE` (BOOL) ⟶ `INI_AX.DEFAULT_VALUE`; `ID_READ` ⟶ `AX_SUBSCRIBE_1.ID`; `ID_WRITE` ⟶ `AX_PUBLISH_1.ID`.
3. **Qualität**: `QI` ⟶ `.QI` von `AX_SUBSCRIBE_1`, `AX_PUBLISH_1` und `INI_AX` (nicht mehr fest `QI=TRUE`); `AX_SUBSCRIBE_1.QO` ⟶ `AND_QO.IN1`, `AX_PUBLISH_1.QO` ⟶ `AND_QO.IN2`, `INI_AX.QO` ⟶ `AND_QO.IN3`; `AND_QO.OUT` ⟶ `QO`.
4. **Adapterkette**: `AX_SUBSCRIBE_1.OUT` ⟶ `INI_AX.AX_IN` ⟶ `INI_AX.AX_OUT` ⟶ `AX_SPLIT_2.IN`; `AX_SPLIT_2.OUT1` ⟶ `AX_PUBLISH_1.IN` **und** `AX_SPLIT_2.OUT2` ⟶ `OUT` (Plug an die SubApp-Schnittstelle).

Technische Besonderheiten

- **BOOL statt REAL**: Einziger fachlicher Unterschied zu `INI_OPC_PARAM_AR` ist der Datentyp (`DEFAULT_VALUE` ist hier `BOOL`) und die entsprechenden `AX`-Adapter-Typen anstelle von `AR`.
- Ansonsten identisch zu `INI_OPC_PARAM_AR` (Initialisierungsreihenfolge, `SETM=FALSE`, ein Ausgang mit zwei Zielen).

Anwendungsszenarien

- Remanent gespeicherte BOOL-Parameter (z. B. Drehrichtung, Betriebsmodus-Schalter), die per OPC-UA/Web-Client editierbar sein und zusätzlich lokal als `AX`-Adapter weiterverwendet werden sollen.

Vergleich mit ähnlichen Bausteinen

`INI_OPC_PARAM_AX` verhält sich zu `INI_OPC_PARAM_AR` wie `AX` zu `AR`: gleiche Struktur, anderer Datentyp. `INI_OPC_PARAM` und `INI_OPC_PARAM_ATM` sind die REAL-basierten Varianten (ohne bzw. mit zusätzlicher `TIME`-Umrechnung).

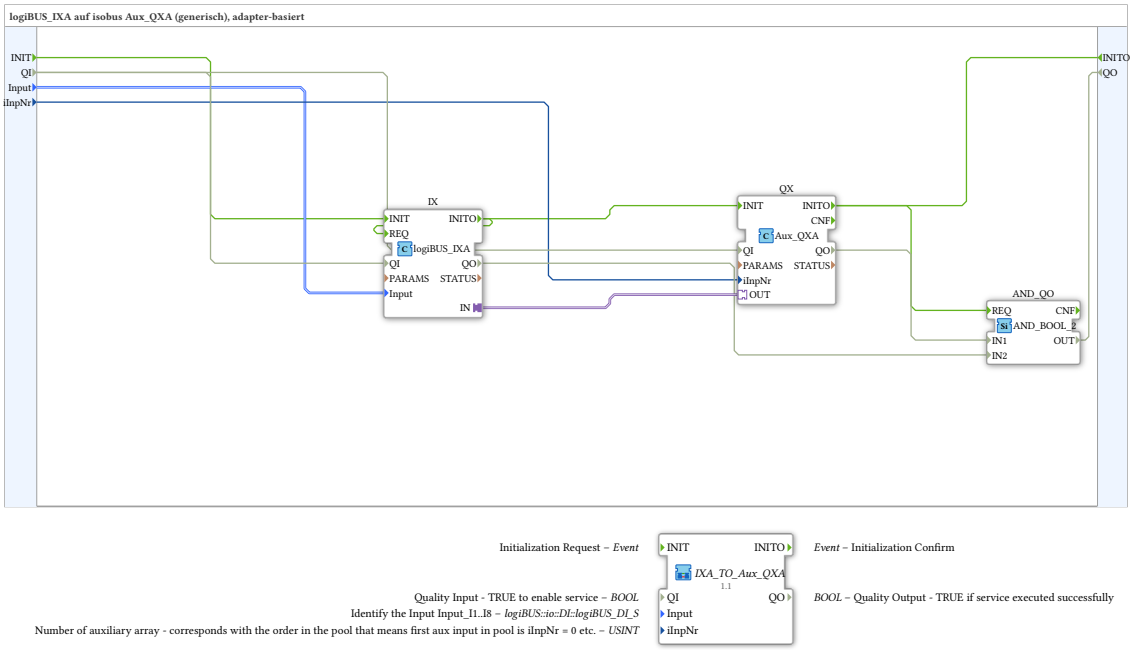
Zusammenfassung

`INI_OPC_PARAM_AX` ist die BOOL-Variante der `INI_OPC_PARAM`-Familie: OPC-UA-Lese-/Schreibzugriff, remanente INI-Speicherung und ein lokaler `AX`-Ausgang für einen booleschen Parameter.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

IXA_TO_Aux_QXA



• — ○ • — ○ • — ○ •

Einleitung

Die SubApp **IXA_TO_Aux_QXA** dient als generischer, adapterbasierter Konverter zwischen einem logiBUS-Eingangsbaustein (`logiBUS_IXA`) und einem isobus-Ausgangsbaustein (`isobus::UT::io::Auxiliary::OUT::Aux_QXA`). Sie ermöglicht die Übertragung digitaler Eingangssignale des logiBUS-Systems auf einen Auxiliary-Kanal des isobus-Netzwerks. Die SubApp wurde aus einer bestehenden Übung ausgelagert, um sie als wiederverwendbaren Baustein in verschiedenen Projekten einzusetzen. Alle Verbindungen erfolgen über standardisierte Adapter, wodurch eine flexible und schnelle Integration in 4diac-Applikationen möglich ist.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialization Request; wird an `IX.INIT` weitergegeben.

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialization Confirm; kommt von `QX.INITO`.

Daten-Eingänge

Name	Typ	Beschreibung
`Input`	`logiBUS::io::DI::logiBUS_DI_S`	Identifiziert den logiBUS-Eingang (z. B. Input_I1..I8). Initialwert: `Invalid`
`QI`	`BOOL`	Quality Input – TRUE aktiviert den Dienst. Initialwert: `TRUE`
`iInpNr`	`USINT`	Nummer des Auxiliary-Arrays, entspricht der Reihenfolge im Pool (d. h. erster Aux-Eingang im Pool = 0). Initialwert: `0`

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output – UND-Verknüpfung (`AND_QO`, `iec61131::booleanOperators::AND_BOOL_2`) von `QX.QO` und `IX.QO`.

Adapter

Keine an der SubApp selbst. Intern sind die Adapter der enthaltenen Funktionsbausteine `IX` und `QX` miteinander verbunden (Adapter-Ausgang von `IX` mit Adapter-Eingang von `QX`).

Funktionsweise

Die SubApp enthält zwei Funktionsbausteine:

- **`IX`** – ein `logiBUS_IXA`-Baustein, der die über den Daten-Eingang `Input` zugeführten logiBUS-Daten (vom Typ `logiBUS_DI_S`) in ein adapterspezifisches Format wandelt und über seinen Adapter-Ausgang `IN` bereitstellt.
- **`QX`** – ein `Aux_QXA`-Baustein, der die über seinen Adapter-Eingang `OUT` empfangenen Daten auf einen isobus-Auxiliary-Kanal ausgibt. Die Auswahl des Kanals erfolgt über den Eingang `iInpNr`.

Die beiden Bausteine sind über eine Adapterverbindung (`IX.IN` ⟶ `QX.OUT`) direkt gekoppelt, sodass die logiBUS-Daten ohne zusätzliche Konfiguration auf den isobus-Kanal übertragen werden. Die Schnittstelle nach außen besteht aus den Daten-Eingängen `Input` und `iInpNr`, dem Quality-Paar `QI`/`QO` sowie den Ereignissen `INIT`/`INITO`. Initialisierung: `INIT` ⟶ `IX.INIT`; `IX.INITO` ⟶ `QX.INIT`; `QX.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`. Qualität: `QI` ⟶ `IX.QI` und `QX.QI`; `QX.QO` ⟶ `AND_QO.IN1`, `IX.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`. Durch die Verwendung von Adaptern wird die SubApp unabhängig von spezifischen Implementierungsdetails und leicht in verschiedene Systemumgebungen integrierbar.

Technische Besonderheiten

- **Adapterbasierte Kopplung**: Die Kommunikation zwischen `logiBUS_IXA` und `Aux_QXA` erfolgt ausschließlich über standardisierte Adapter, wodurch die SubApp protokollunabhängig einsetzbar ist.
- **Generischer Ansatz**: Die SubApp ist generisch ausgelegt und kann durch unterschiedliche Konfigurationen der internen Bausteine (z. B. Änderung der `PARAMS` oder Zuweisung von Eingängen) an verschiedene Anforderungen angepasst werden.
- **Kein Event-Handling im Datenpfad**: Die Datenübertragung erfolgt rein datengetrieben über die Adapterverbindung; Ereignisse (`INIT`/`INITO`) werden nur für die Initialisierung der Bausteinkette und die Qualitätsverknüpfung benötigt.
- **Wiederverwendbarkeit**: Die SubApp wurde aus einer bestehenden Übung (Uebung_003c_sub_AX) ausgelagert, um eine klare, modulare Wiederverwendung zu ermöglichen.
- **Initialisierung und Qualität**: Beide Funktionsbausteine erhalten `QI` vom SubApp-Eingang `QI` (Vorgabe `TRUE`, also standardmäßig aktiv); es gibt keinen fest verdrahteten `QI`-Parameter mehr. Der `PARAMS`-Parameter des `IX`-Bausteins ist leer gesetzt, was auf Standardeinstellungen hinweist.

Zustandsübersicht

Da die SubApp keine eigenen Zustände besitzt, wird das Zustandsverhalten durch die internen Funktionsbausteine `logiBUS_IXA` und `Aux_QXA` bestimmt. Diese Bausteine verfügen über eine interne Zustandslogik, die von der jeweiligen Implementierung abhängt (z. B. Initialisierung, Fehlerbehandlung, Datenaustausch). Die SubApp delegiert alle Operationen an diese Bausteine und bietet eine zusammengefasste funktionale Schnittstelle.

Anwendungsszenarien

- **Integration von logiBUS-Eingängen in isobus-Systeme**: Die SubApp eignet sich, um digitale Eingangssignale (z. B. Sensoren oder Schalter) eines logiBUS-Netzwerks als Auxiliary-Ausgänge im isobus-Netzwerk verfügbar zu machen.
- **Wiederverwendbare Konverter-Komponente**: In Projekten, in denen wiederholt logiBUS-Signale auf isobus-Auxiliary-Kanäle abgebildet werden müssen, bietet die SubApp eine einheitliche und getestete Lösung.
- **Modulare Systemarchitektur**: Durch die Kapselung der Konvertierung in einer SubApp wird die Gesamtapplikation übersichtlicher und leichter wartbar. Änderungen an der Konvertierungslogik können zentral vorgenommen werden.

Vergleich mit ähnlichen Bausteinen

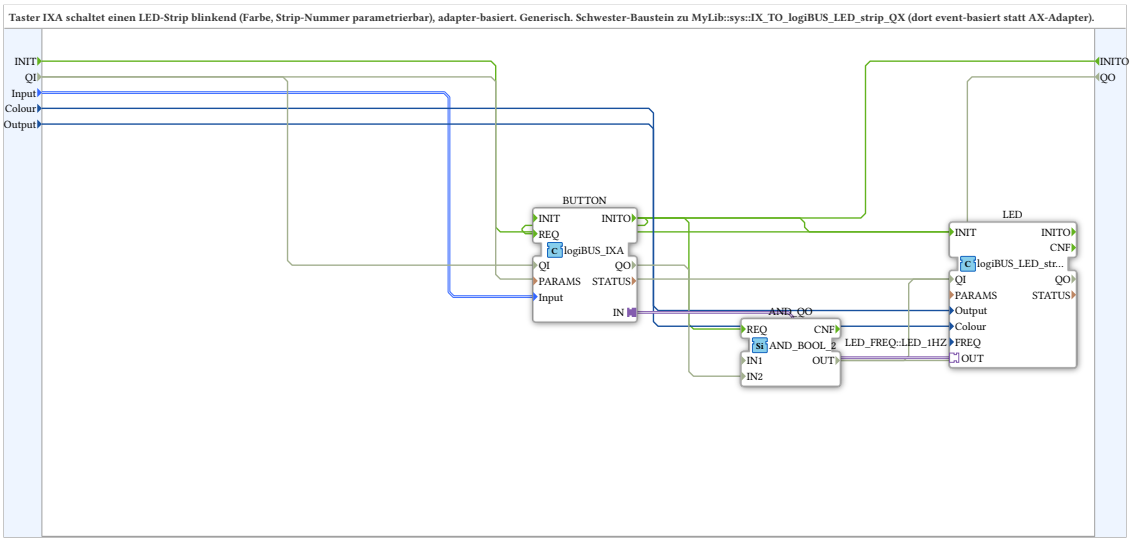
- **Direkte Verbindung ohne SubApp**: Statt die beiden Bausteine `logiBUS_IXA` und `Aux_QXA` direkt in einer Applikation zu verdrahten, kapselt die SubApp die Verbindung logisch und ermöglicht eine einfachere Wiederverwendung.

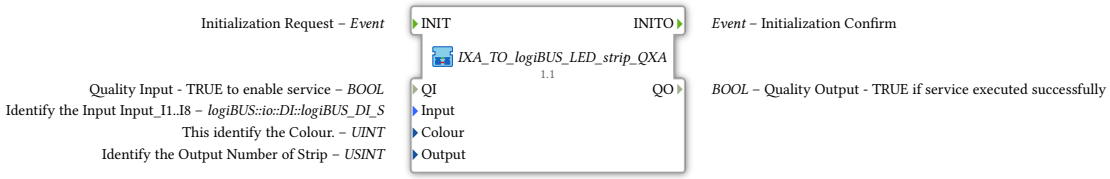
- **Andere Konverter-SubApps**: Es existieren möglicherweise SubApps, die mehrere Eingänge bündeln oder zusätzliche Logik (z. B. Filter, Skalierung) integrieren. `IXA_TO_Aux_QXA` ist bewusst einfach gehalten und fokussiert ausschließlich auf die Adapterkopplung.
- **Generische Protokoll-Adapter**: Im Vergleich zu universellen Protokoll-Konvertern ist dieser Baustein auf die spezifische Kombination aus logiBUS und isobus-Auxiliary optimiert und dadurch weniger konfigurabel, aber auch einfacher in der Anwendung.

Fazit

Die SubApp **IXA_TO_Aux_QXA** stellt eine kompakte und effiziente Lösung zur Kopplung von logiBUS-Eingängen mit isobus-Auxiliary-Ausgängen dar. Durch den adapterbasierten Aufbau ist sie flexibel, standardkonform und leicht in bestehende 4diac-Anwendungen integrierbar. Mit den beiden Daten-Eingängen `Input` und `iInpNr` sowie dem üblichen Quality-/Init-Anschlusspaar (`QI`/`QO`, `INIT`/`INITO`) ist die Schnittstelle minimal und klar. Die SubApp eignet sich hervorragend für Projekte, die eine saubere Trennung und Wiederverwendbarkeit der Konvertierungslogik verlangen.

IXA_TO_logiBUS_LED_strip_QXA





• — ○ • — ○ • — ○ •

Einleitung

Der Baustein `IXA_TO_logiBUS_LED_strip_QXA` ist eine generische, adapterbasierte Subapp aus `MyLib::sys`. Sie verbindet einen Taster-Baustein `logiBUS_IXA` mit einem LED-Strip-Baustein `logiBUS_LED_strip_QXA`. Über die Daten-Eingänge lassen sich der Tastereingang, die Farbe und die Strip-Nummer parametrieren. Der Baustein ist die adapterbasierte Schwester-Variante des eventbasierten Bausteins `MyLib::sys::IX_TO_logiBUS_LED_strip_QX`.

Schnittstellenstruktur

Die Subapp besitzt die Initialisierungs-Ereignisse `INIT`/`INITO`, die Status-Anschlüsse `QI`/`QO` und Daten-Eingänge. Es gibt keine externen Adapter.

Ereignis-Eingänge

Name	Typ	Beschreibung
INIT	`EInit`	Initialization Request

Ereignis-Ausgänge

Name	Typ	Beschreibung
INITO	`EInit`	Initialization Confirm

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung
QI	`BOOL`	`TRUE`	Quality Input - TRUE aktiviert den Dienst.
Input	`logiBUS::io::DI::logiBUS_DI_S`	`Invalid`	Identifiziert den Tastereingang `Input_I1..I8`.
Colour	`UINT`	`LED_COLOURS::LED_GREEN`	Identifiziert die Farbe des LED-Strips.
Output	`USINT`	–	Identifiziert die Ausgangsnummer des Strips.

Daten-Ausgänge

Name	Typ	Beschreibung
QO	`BOOL`	Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde (UND von `LED.QO` und `BUTTON.QO`).

Adapter

Keine externen Adapter-Schnittstellen. Intern ist der Adapter `BUTTON.IN` mit dem Adapter `LED.OUT` verbunden. Dadurch wird die Tasterinformation ohne Ereignisse direkt an den LED-Strip weitergegeben.

Funktionsweise

Der Eingang `Input` wählt den gewünschten Tastereingang aus und wird an den internen Funktionsbaustein `BUTTON` durchgereicht. Die Eingänge `Colour` und `Output` werden direkt an den internen LED-Strip-Baustein `LED` übergeben.

Die eigentliche Kopplung zwischen Taster und LED-Strip erfolgt über die interne Adapterverbindung `BUTTON.IN` → `LED.OUT`. Der LED-Baustein ist dabei mit `FREQ=LED_FREQ::LED_1HZ` parametriert; `QI` kommt vom `QI`-Eingang der Subapp. Das bedeutet, der angeschlossene LED-Strip blinkt bei Betätigung des Tasters mit 1 Hz in der eingestellten Farbe.

Da die Kopplung über Adapter und nicht über Ereignisse realisiert ist, benötigt die Subapp außer `INIT`/`INITO` keine Ereignisse. Die Initialisierung läuft als Kette `INIT` → `LED.INIT`, `LED.INITO` → `BUTTON.INIT`, `BUTTON.INITO` → `INITO` (und zusätzlich → `AND_QO.REQ`). `QI` wird an `LED.QI` und `BUTTON.QI` weitergegeben; `LED.QO` und `BUTTON.QO` werden im Baustein `AND_QO` (Typ `iec61131::booleanOperators::AND_BOOL_2`) per UND zu `QO` verknüpft.

Technische Besonderheiten

- Adapterbasierte Kopplung statt ereignisbasierter Kopplung.
- Die Subapp besitzt außer `INIT`/`INITO` keine Ereignisse und keine externen Adapter-Schnittstellen.
- Alle wesentlichen Parameter sind über die Daten-Eingänge konfigurierbar:
  - Tastereingang (`Input`)
  - Farbe (`Colour`)
  - Strip-Nummer (`Output`)
- Der interne LED-Baustein wird fest mit `FREQ=LED_FREQ::LED_1HZ` betrieben; `QI` stammt vom Subapp-Eingang `QI` (Vorgabe `TRUE`).
- Der Eingang `Input` ist initial mit `Invalid` belegt und muss vor der Inbetriebnahme auf einen gültigen Wert gesetzt werden.
- Der Baustein verwendet Typen und Konstanten aus der Bibliothek `logiBUS`, z. B. `logiBUS_DI_S`, `LED_COLOURS` und `LED_FREQ`.

Zustandsübersicht

Die Subapp selbst besitzt keine eigene Zustandsmaschine. Das Verhalten ergibt sich aus den intern verdrahteten Funktionsbausteinen:

- **Taster nicht betätigt:** Keine Tasterinformation wird über den Adapter gesendet; der LED-Strip leuchtet nicht.
- **Taster betätigt:** Der Adapter überträgt die Information an den LED-Strip; der Strip blinkt mit 1 Hz in der parametrierten Farbe.

Je nach Implementierung von `logiBUS_IXA` kann zusätzlich zwischen Flanke, Zustand oder Impuls unterschieden werden. Die Subapp selbst bildet diese Logik jedoch nicht ab, sondern reicht die Tasterinformation über den Adapter weiter.

Anwendungsszenarien

- Generischer Taster-Schalter für einen LED-Strip: Der Anwender wählt den Tastereingang, die Farbe und die Strip-Nummer.
- Wiederverwendbare Kapselung einer Taster-LED-Strip-Kombination in einer Anlage.
- Einsatz als adapterbasierte Alternative zum eventbasierten Schwester-Baustein `IX_TO_logiBUS_LED_strip_QX`.
- Mehrere Instanzen der Subapp können mit unterschiedlichen Eingängen, Farben und Strip-Nummern parametriert werden, ohne die innere Verdrahtung anzupassen.

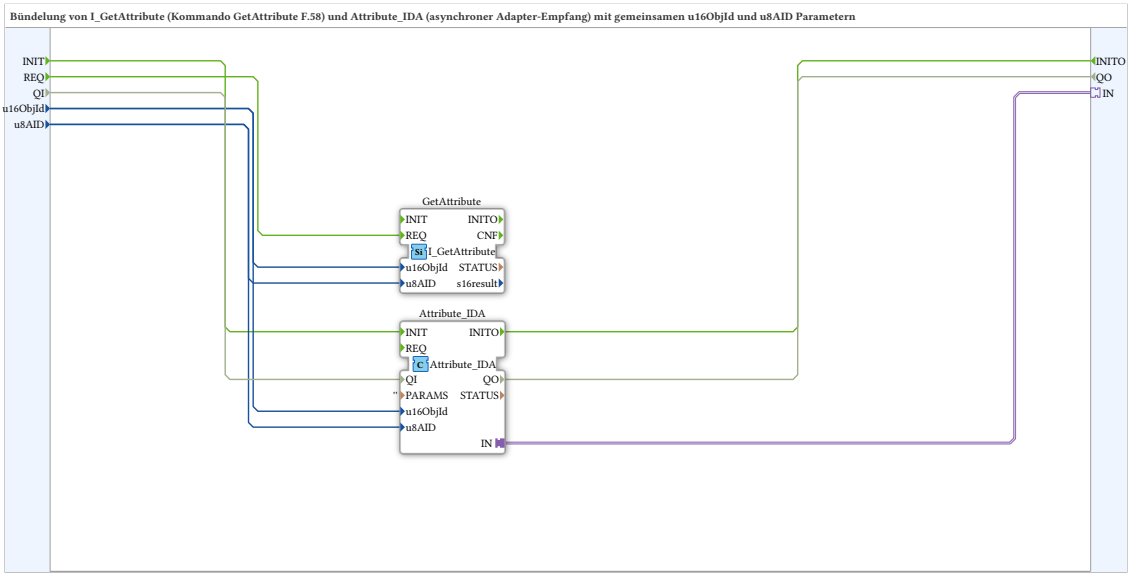
Vergleich mit ähnlichen Bausteinen

- `MyLib::sys::IX_TO_logiBUS_LED_strip_QX`: Eventbasierte Variante. Die Tasterinformation wird über Ereignisse an den LED-Strip übergeben.
- `IXA_TO_logiBUS_LED_strip_QXA`: Adapterbasierte Variante. Die Tasterinformation wird über eine interne Adapterverbindung direkt übertragen.
- Gegenüber einer direkten Verwendung von `logiBUS_IXA` und `logiBUS_LED_strip_QXA` bietet die Subapp eine fertig verdrahtete, parametrierbare Einheit.

Fazit

Der Baustein `IXA_TO_logiBUS_LED_strip_QXA` bietet eine kompakte und flexible Möglichkeit, einen Taster mit einem blinkenden LED-Strip zu verbinden. Durch die adapterbasierte interne Verknüpfung entfallen externe Ereignisse, und die Parametrierung über die Daten-Eingänge macht den Baustein vielseitig einsetzbar. Er eignet sich besonders für wiederverwendbare Taster-LED-Kombinationen in logiBUS-Umgebungen.

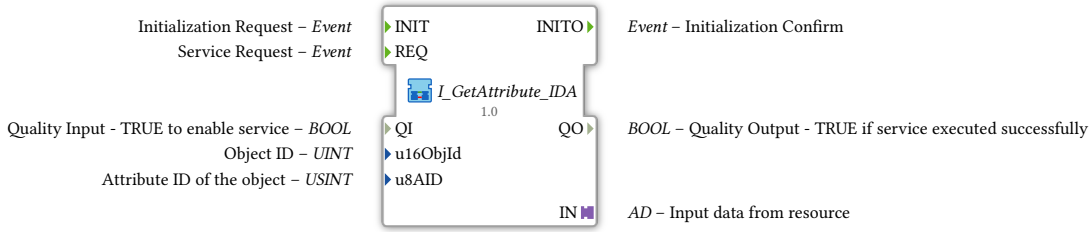
I_GetAttribute_IDA



• – ○ • – ○ • – ○ •

Einleitung

`I_GetAttribute_IDA` bündelt `I_GetAttribute` (Kommando *GetAttribute*, F.58) und `Attribute_IDA` (asynchroner Adapter-Empfang) mit gemeinsamen Parametern `u16ObjId` und `u8AID`. Mit `REQ` wird das Attribut eines VT-Objekts angefordert; die Antwort kommt asynchron über den Adapter `IN`.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung (an `Attribute_IDA.INIT`)
- **REQ**: Serviceanforderung (an `GetAttribute.REQ`)

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung (von `Attribute_IDA.INITO`)

Die Antwort selbst kommt über den Adapter-Plug.

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input, Freigabe des Dienstes (an `Attribute_IDA.QI`)
- **u16ObjId** (`UINT`, Vorgabe `ID_NULL`): Object-ID
- **u8AID** (`USINT`): Attribut-ID des Objekts

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output (von `Attribute_IDA.QO`)

Adapter

- **IN** (Plug, `adapter::types::unidirectional::AD`): Eingangsdaten (Antwort) aus der Ressource

Verwendete Funktionsbausteine (FBs)

- **GetAttribute**: `isobus::UT::I::I_GetAttribute` – sendet das Kommando *GetAttribute*
- **Attribute_IDA**: `isobus::UT::io::Attribute::Attribute_IDA` (`PARAMS=''`, `QI` kommt von der SubApp-Schnittstelle) – empfängt die Antwort asynchron als Adapter

Programmablauf und Verbindungen

- `REQ` ⟶ `GetAttribute.REQ`
- `u16ObjId` ⟶ `GetAttribute.u16ObjId` und `Attribute_IDA.u16ObjId`
- `u8AID` ⟶ `GetAttribute.u8AID` und `Attribute_IDA.u8AID`
- `Attribute_IDA.IN` ⟶ `IN`
- `INIT` ⟶ `Attribute_IDA.INIT`; `Attribute_IDA.INITO` ⟶ `INITO`
- `QI` ⟶ `Attribute_IDA.QI`; `Attribute_IDA.QO` ⟶ `QO`

Technische Besonderheiten

- Durch die gemeinsamen Parameter wird sichergestellt, dass Anforderung und Empfang auf dasselbe Objekt und Attribut zeigen.

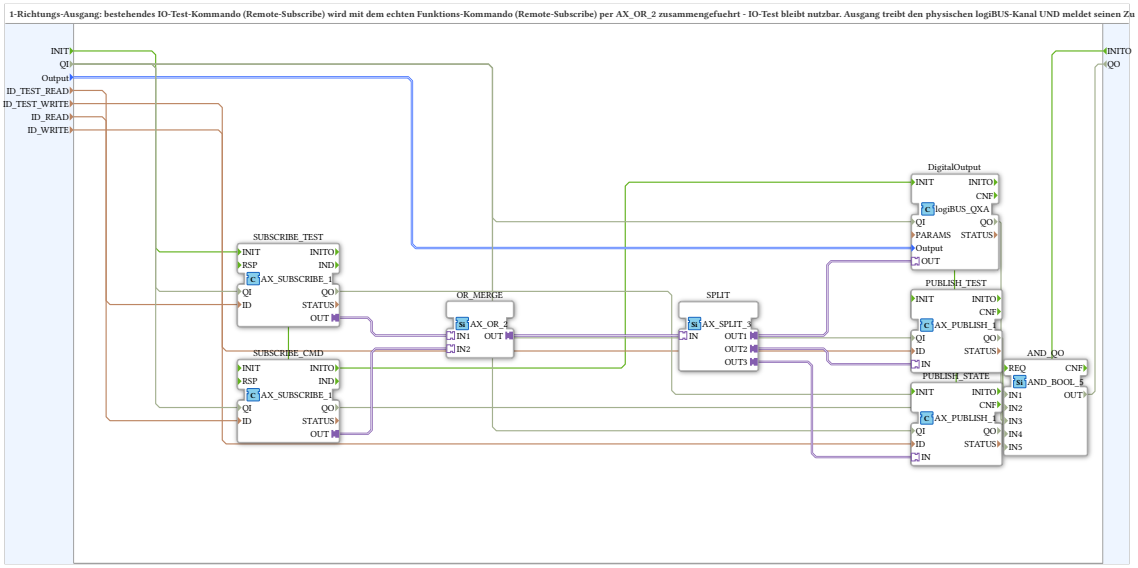
Anwendungsszenarien

- Auslesen eines Attributs eines VT-Objekts (z. B. aktueller Wert, Farbe) und Weiterverarbeiten der Antwort als `AD`-Adapterwert

Fazit

`I_GetAttribute_IDA` kombiniert Anforderung und Empfang eines VT-Objektattributs in einer SubApp.

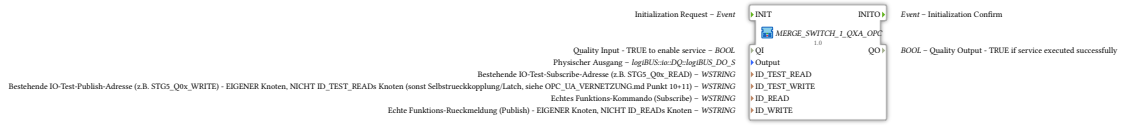
MERGE_SWITCH_1_QXA_OPC



• — ○ • — ○ • — ○ •

Einleitung

`MERGE_SWITCH_1_QXA_OPC` treibt einen einzelnen (nicht doppelwirkenden) logiBUS-Ausgang aus zwei OPC-UA-Quellen: dem bestehenden IO-Test-Kommando (Remote-Subscribe) und dem echten Funktions-Kommando (ebenfalls Remote-Subscribe, z. B. von einem SoftKey/AUX über `Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC`). Beide werden per Pegel-ODER zusammengeführt, sodass der bestehende IO-Test weiterhin nutzbar bleibt, während gleichzeitig die echte Funktion denselben Ausgang ansteuern kann. Für doppelwirkende Aktoren (Links/Rechts, Auf/Ab) ist stattdessen `ILOCK_SWITCH_2_QXA_OPC` zu verwenden.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: MERGE_SWITCH_1_QXA_OPC

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **SUBSCRIBE_TEST** / **SUBSCRIBE_CMD**: je `adapter::net::AX_SUBSCRIBE_1` (`QI` kommt vom Eingang `QI`) — abonnieren das bestehende IO-Test-Kommando (`ID_TEST_READ`) bzw. das echte Funktions-Kommando (`ID_READ`).
  - **OR_MERGE**: `adapter::booleanOperators::AX_OR_2` — führt beide Subscribe-Quellen per Pegel-ODER zusammen.
  - **SPLIT**: `adapter::events::unidirectional::AX_SPLIT_3` — verteilt das zusammengeführte Signal auf den physischen Ausgang und zwei Rückmelde-Kanäle.
  - **DigitalOutput**: `logiBUS::io::DQ::logiBUS_QXA` (`QI` kommt vom Eingang `QI`) — physischer digitaler Ausgang.
  - **PUBLISH_TEST** / **PUBLISH_STATE**: je `adapter::net::AX_PUBLISH_1` (`QI` kommt vom Eingang `QI`) — melden den resultierenden Zustand an die bestehende IO-Test-Überwachung (`ID_TEST_WRITE`) bzw. an die echte Funktions-Rückmeldung (`ID_WRITE`, z. B. `GreenWhiteBackground` am SoftKey).
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_5` — verknüpft die `QO`-Ausgänge der fünf Dienst-FBs zum Ausgang `QO`.
- **Funktionsweise**: Zwei unabhängige Remote-Subscribe-Quellen werden per `AX_OR_2` zusammengeführt und treiben gemeinsam den physischen Ausgang sowie zwei getrennte Rückmeldekanäle. Die SubApp-Schnittstelle besitzt außerdem die Ereignisse `INIT`/`INITO` und die Daten `QI` (BOOL, Initialwert `TRUE`) und `QO` (BOOL).

Programmablauf und Verbindungen

1. `ID_TEST_READ` ⟶ `SUBSCRIBE_TEST.ID`; `ID_READ` ⟶ `SUBSCRIBE_CMD.ID` (Datenverbindungen, ausgeblendet).
2. `SUBSCRIBE_TEST.OUT` ⟶ `OR_MERGE.IN1`; `SUBSCRIBE_CMD.OUT` ⟶ `OR_MERGE.IN2`.
3. `OR_MERGE.OUT` ⟶ `SPLIT.IN` ⟶ `SPLIT.OUT1` ⟶ `DigitalOutput.OUT` (physischer Ausgang), `SPLIT.OUT2` ⟶ `PUBLISH_TEST.IN`, `SPLIT.OUT3` ⟶ `PUBLISH_STATE.IN`.
4. Parameter: `Output` ⟶ `DigitalOutput.Output`; `ID_TEST_WRITE` ⟶ `PUBLISH_TEST.ID`; `ID_WRITE` ⟶ `PUBLISH_STATE.ID`.
5. Initialisierungskette: `INIT` ⟶ `SUBSCRIBE_TEST.INIT` ⟶ `SUBSCRIBE_CMD.INIT` ⟶ `DigitalOutput.INIT` ⟶ `PUBLISH_TEST.INIT` ⟶ `PUBLISH_STATE.INIT` (jeweils `INITO` ⟶ nächstes `INIT`); `PUBLISH_STATE.INITO` ⟶ `INITO` und `AND_QO.REQ`.
6. Qualität: `QI` ⟶ `SUBSCRIBE_TEST.QI`, `SUBSCRIBE_CMD.QI`, `DigitalOutput.QI`, `PUBLISH_TEST.QI` und `PUBLISH_STATE.QI`; deren `QO`-Ausgänge ⟶ `AND_QO.IN1` bis `AND_QO.IN5` (in dieser Reihenfolge); `AND_QO.OUT` ⟶ `QO`.

Technische Besonderheiten

- **Zwei getrennte Rückmelde-Knoten**: Der resultierende Zustand wird auf zwei eigene, vom jeweiligen Kommando-Knoten getrennte OPC-UA-Knoten zurückgemeldet (`ID_TEST_WRITE` für die bestehende IO-Diagnoseanzeige, `ID_WRITE` für die echte Funktionsrückmeldung). Kommando- und Status-Knoten dürfen sich nie denselben OPC-UA-Knoten teilen — sonst liest das eigene Subscribe das eigene Publish-Echo als neuen Befehl und der Ausgang hängt fest.
- **Pegel-ODER statt Flankenlogik**: Im Gegensatz zu den latchenden Button-Bausteinen (z. B. `Button_IXA_TO_logiBUS_QXA_BG_OPC_LATCHING`) verwendet dieser Baustein ein einfaches, dauerhaftes ODER (`AX_OR_2`) — passend für ein Kommando, das durchgehalten (nicht getoggelt) wird.
- **Generisch für einfache Ausgänge**: Anwendbar auf jeden einfachen, nicht doppelwirkenden Ausgang mit Bedienung von einem Modul ohne eigenes VT.

Anwendungsszenarien

- Physische Ausgänge, deren bestehender IO-Test-Kanal auch nach Einführung der echten Funktion weiter nutzbar bleiben soll, ohne dass beide Kommandoquellen sich gegenseitig blockieren.
- Einfache, nicht doppelwirkende Aktoren (z. B. Blitzlicht, Beleuchtung), die sowohl vom IO-Test als auch von einer echten Bedienfunktion (SoftKey/AUX) angesteuert werden können.

Vergleich mit ähnlichen Bausteinen

Für doppelwirkende Aktoren (Links/Rechts, Auf/Ab), bei denen sich die beiden Richtungen gegenseitig ausschließen müssen, ist `ILOCK_SWITCH_2_QXA_OPC` zu verwenden — `MERGE_SWITCH_1_QXA_OPC` besitzt keinen Interlock, da es nur eine einzige Ausgangsrichtung ansteuert.

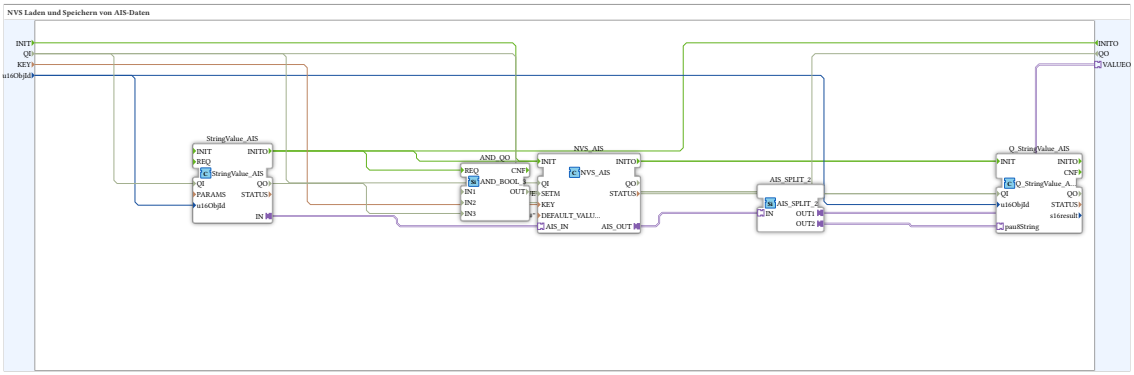
Zusammenfassung

`MERGE_SWITCH_1_QXA_OPC` führt IO-Test-Kommando und echtes Funktions-Kommando per Pegel-ODER zu einem gemeinsamen, einfachen Ausgang zusammen und meldet den Zustand getrennt an beide Überwachungswege zurück.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

NVS_IN_AND_STORE_AIS

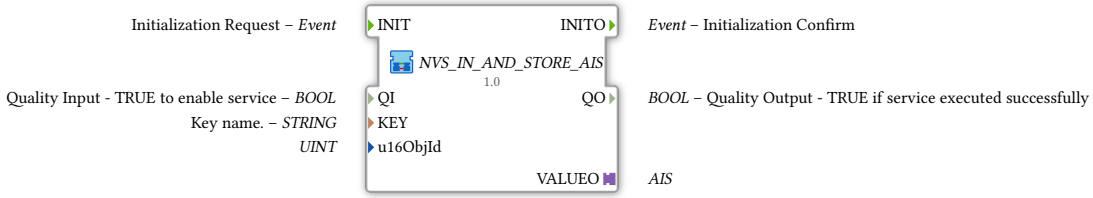


• – ○ • – ○ • – ○ •

Einleitung

`NVS_IN_AND_STORE_AIS` speichert einen über VT eingegebenen String-Wert (AIS-Adapter) persistent im ESP32-Flash (NVS) statt in einer INI-Datei — sonst identisch zu `INI_IN_AND_STORE_AIS` (kein `SECTION`-Parameter, da NVS anders organisiert ist als INI-Dateien).

Allgemeines Muster siehe INI_IN_AND_STORE / NVS_IN_AND_STORE (gemeinsames Muster).



Qualitätsanschlüsse (QI/QO)

Der Baustein besitzt den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`, Quality Input - TRUE aktiviert den Dienst) und den Datenausgang `QO` (`BOOL`, Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde). `QI` geht an `Q_StringValue_AIS`, `NVS_AIS` und `StringValue_AIS`. `QO` ist der Ausgang der Instanz `AND_QO` (`iec61131::booleanOperators::AND_BOOL_3`), die `Q_StringValue_AIS.QO`, `NVS_AIS.QO` und `StringValue_AIS.QO` verknüpft.

Zusätzlich besitzt der Baustein die Ereignisse `INIT` (Eingang, `EInit`, Initialisierungsanforderung) und `INITO` (Ausgang, `EInit`, Initialisierungsbestätigung): `INIT` ⟶ `Q_StringValue_AIS.INIT`, `Q_StringValue_AIS.INITO` ⟶ `NVS_AIS.INIT`, `NVS_AIS.INITO` ⟶ `StringValue_AIS.INIT`, `StringValue_AIS.INITO` ⟶ `INITO` (und ⟶ `AND_QO.REQ`).

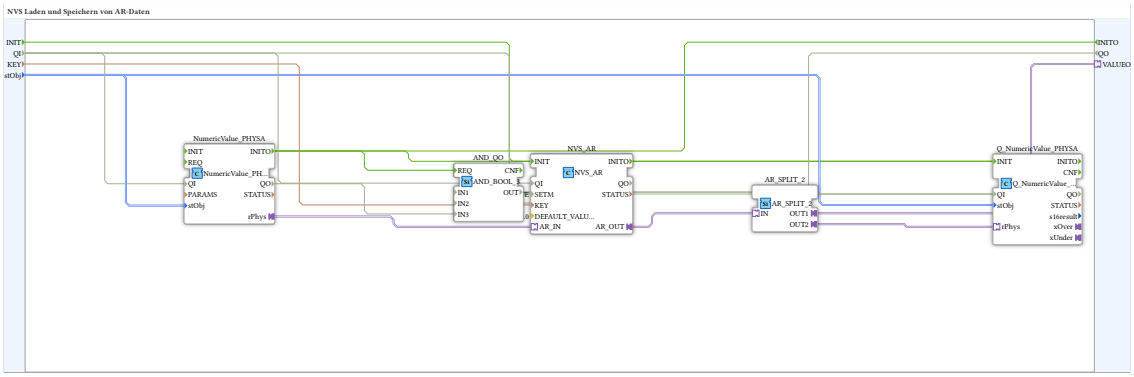
Zusammenfassung

NVS-Pendant zu `INI_IN_AND_STORE_AIS`.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

NVS_IN_AND_STORE_AR

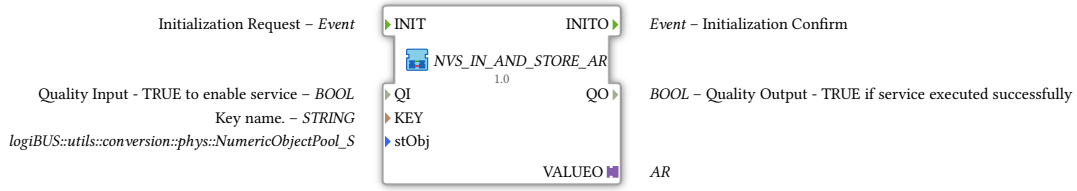


• – ○ • – ○ • – ○ •

Einleitung

`NVS_IN_AND_STORE_AR` speichert einen über VT eingegebenen physikalisch skalierten REAL-Wert (AR-Adapter) persistent im ESP32-Flash (NVS) — sonst identisch zu `INI_IN_AND_STORE_AR`.

Allgemeines Muster siehe INI_IN_AND_STORE / NVS_IN_AND_STORE (gemeinsames Muster).



Qualitätsanschlüsse (QI/QO)

Der Baustein besitzt den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`, Quality Input - TRUE aktiviert den Dienst) und den Datenausgang `QO` (`BOOL`, Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde). `QI` ist mit `Q_NumericValue_PHYSA.QI`, `NVS_AR.QI` und `NumericValue_PHYSA.QI` verbunden; `QO` ist das UND (`AND_QO`, `iec61131::booleanOperators::AND_BOOL_3`) aus `Q_NumericValue_PHYSA.QO`, `NVS_AR.QO` und `NumericValue_PHYSA.QO`.

Zusätzlich besitzt der Baustein die Ereignisse `INIT` (Eingang, `EInit`, Initialisierungsanforderung) und `INITO` (Ausgang, `EInit`, Initialisierungsbestätigung): die Initialisierung wird gekettet: `INIT` ⟶ `Q_NumericValue_PHYSA.INIT`; `Q_NumericValue_PHYSA.INITO` ⟶ `NVS_AR.INIT`; `NVS_AR.INITO` ⟶ `NumericValue_PHYSA.INIT`; `NumericValue_PHYSA.INITO` ⟶ `INITO` (und ⟶ `AND_QO.REQ`).

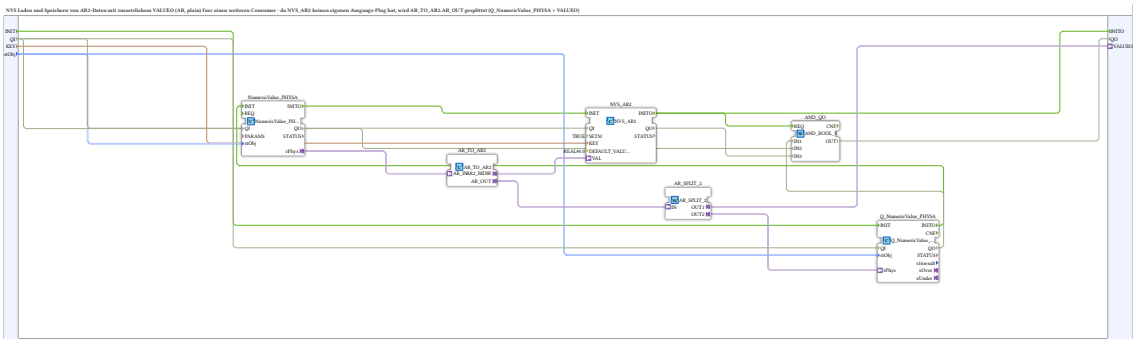
Zusammenfassung

NVS-Pendant zu `INI_IN_AND_STORE_AR`.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

NVS_IN_AND_STORE_AR2

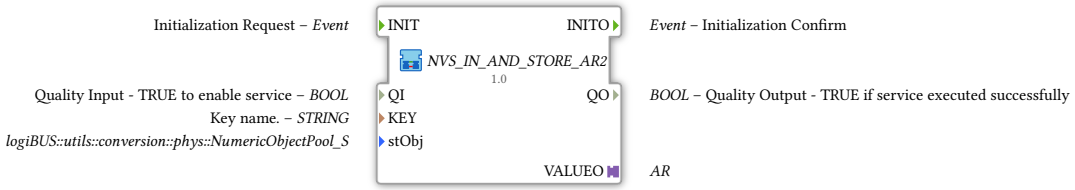


• – ○ • – ○ • – ○ •

Einleitung

`NVS_IN_AND_STORE_AR2` ist die AR2-Adapter-Variante von `NVS_IN_AND_STORE_AR` (ESP32-Flash-Speicherung statt INI-Datei): ein per VT eingegebener physikalisch skalierter REAL-Wert wird ueber `logiBUS::storage::esp32_nvs::NVS_AR2` persistent im NVS-Flash gespeichert. Da `NVS_AR2` (anders als die INI-Variante) auch kein eigenes `SECTION` kennt, entfaellt dieser Parameter gegenueber `INI_IN_AND_STORE_AR2`. Wie dort wird `AR_TO_AR2.AR_OUT` per `AR_SPLIT_2` verdoppelt: VT-Anzeige (`Q_NumericValue_PHYSA`) + `VALUEO` fuer einen weiteren Consumer.

Allgemeines Muster siehe INI_IN_AND_STORE / NVS_IN_AND_STORE (gemeinsames Muster).



Technische Besonderheiten

- `SETM=TRUE`, gleiche Begruendung wie bei `INI_IN_AND_STORE_AR2`: Live-Aenderungen werden sofort bestaetigt, nicht erst beim naechsten Boot.
- Kein `SECTION`-Parameter, da NVS-Keys flach organisiert sind (kein INI-Abschnittskonzept).

Qualitätsanschlüsse (QI/QO)

Der Baustein besitzt den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`, Quality Input - TRUE aktiviert den Dienst) und den Datenausgang `QO` (`BOOL`, Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde). `QI` wird an `Q_NumericValue_PHYSA`, `NumericValue_PHYSA` und `NVS_AR2` verteilt; deren `QO`-Ausgänge werden von `AND_QO` (`iec61131::booleanOperators::AND_BOOL_3`) zu `QO` verknüpft.

Zusätzlich besitzt der Baustein die Ereignisse `INIT` (Eingang, `EInit`, Initialisierungsanforderung) und `INITO` (Ausgang, `EInit`, Initialisierungsbestätigung). Initialisierung: `INIT` → `Q_NumericValue_PHYSA.INIT`; `Q_NumericValue_PHYSA.INITO` → `NumericValue_PHYSA.INIT`; `NumericValue_PHYSA.INITO` → `NVS_AR2.INIT`; `NVS_AR2.INITO` → `INITO` und → `AND_QO.REQ`. Qualität: `QI` → `Q_NumericValue_PHYSA.QI`, `NumericValue_PHYSA.QI`, `NVS_AR2.QI`; `Q_NumericValue_PHYSA.QO`, `NumericValue_PHYSA.QO`, `NVS_AR2.QO` → `AND_QO.IN1` bis `AND_QO.IN3` (in dieser Reihenfolge); `AND_QO.OUT` → `QO` (`AND_QO`: `iec61131::booleanOperators::AND_BOOL_3`).

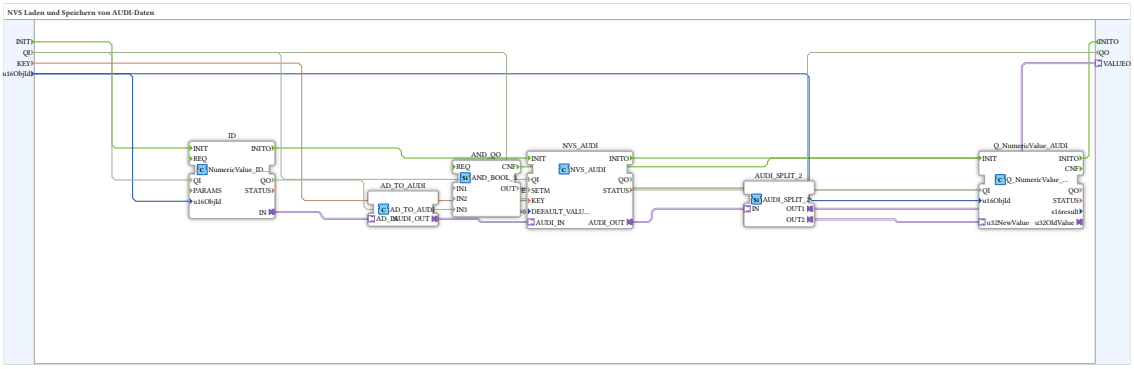
Zusammenfassung

AR2-Adapter-Variante der NVS-Speicherfamilie (ESP32-Flash statt INI-Datei), mit zusaetzlichem `VALUEO`-Ausgang.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

NVS_IN_AND_STORE_AUDI

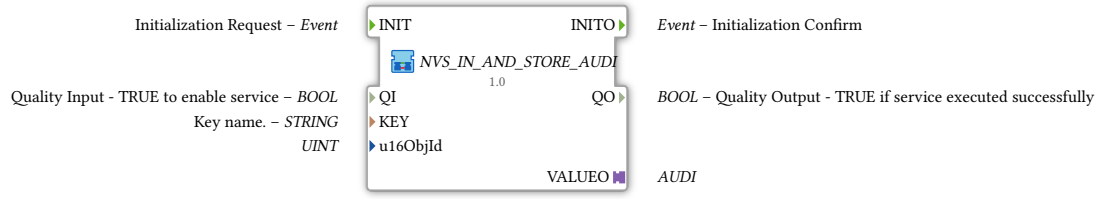


• – ○ • – ○ • – ○ •

Einleitung

`NVS_IN_AND_STORE_AUDI` speichert einen über VT eingegebenen UDINT-Wert (AUDI-Adapter) persistent im ESP32-Flash (NVS) — sonst identisch zu `INI_IN_AND_STORE_AUDI`.

Allgemeines Muster siehe INI_IN_AND_STORE / NVS_IN_AND_STORE (gemeinsames Muster).



Qualitätsanschlüsse (QI/QO)

Der Baustein besitzt den Ereigniseingang `INIT` (`EInit`, Initialization Request), den Ereignisausgang `INITO` (`EInit`, Initialization Confirm), den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`, Quality Input - TRUE aktiviert den Dienst) und den Datenausgang `QO` (`BOOL`, Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde).

- Initialisierungskette: `INIT` ⟶ `ID.INIT`, `ID.INITO` ⟶ `NVS_AUDI.INIT`, `NVS_AUDI.INITO` ⟶ `Q_NumericValue_AUDI.INIT`, `Q_NumericValue_AUDI.INITO` ⟶ `INITO` (und zusätzlich ⟶ `AND_QO.REQ`)
- `QI` ist mit `Q_NumericValue_AUDI.QI`, `NVS_AUDI.QI` und `ID.QI` verbunden
- `QO` ist die UND-Verknüpfung (`AND_QO`, Typ `iec61131::booleanOperators::AND_BOOL_3`) von `Q_NumericValue_AUDI.QO`, `NVS_AUDI.QO` und `ID.QO`

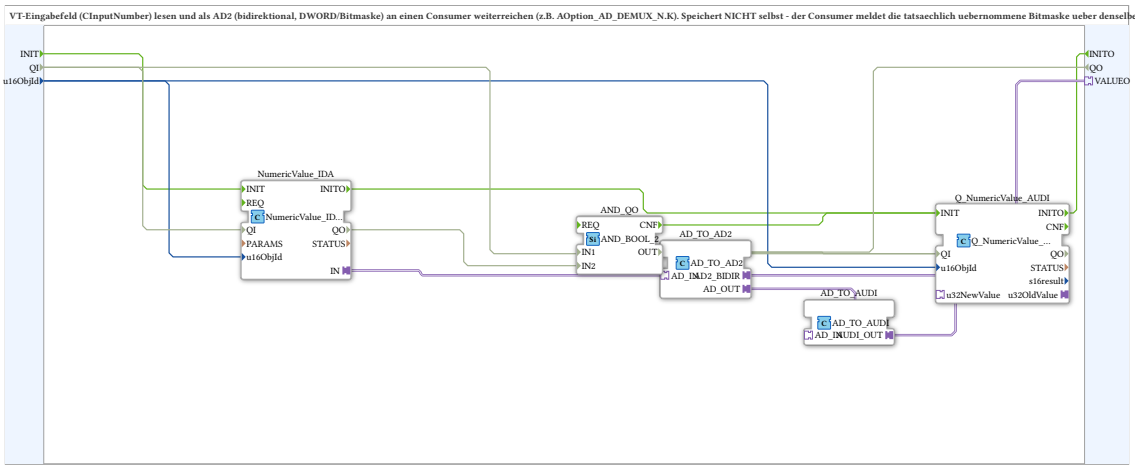
Zusammenfassung

NVS-Pendant zu `INI_IN_AND_STORE_AUDI`.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

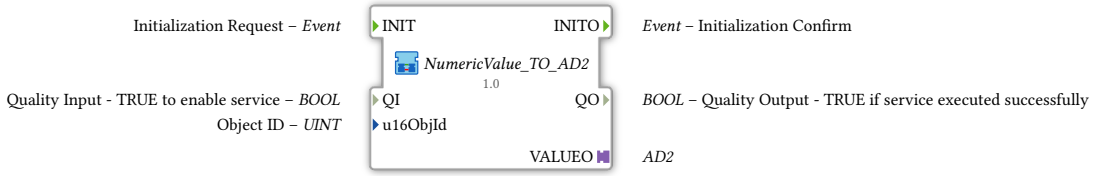
NumericValue_TO_AD2



• – ○ • – ○ • – ○ •

Einleitung

`NumericValue_TO_AD2` liest ein VT-Eingabefeld (CInputNumber) und reicht den Wert bidirektional als AD2-Adapter (DWORD/Bitmaske, `VALUEO`) an einen Consumer weiter (z. B. `AOption_AD_DEMUX_N.K`). Der Baustein speichert **nicht** selbst: Der Consumer meldet die tatsächlich übernommene Bitmaske über denselben AD2-Plug zurück, das Echo wird auf das VT-Feld zurückgeschrieben.



Schnittstelle

Eingangsvariablen

- `QI` (BOOL, Vorgabe `TRUE`): Quality Input - TRUE aktiviert den Dienst; wird an `NumericValue_IDA` und `Q_NumericValue_AUDI` durchgereicht.
- `u16ObjId` (UINT, Default `ID_NULL`): ObjectID des VT-Eingabefelds.

Ausgangsvariablen

- `QO` (BOOL): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde; `AND` der `QO`-Ausgänge von `Q_NumericValue_AUDI` und `NumericValue_IDA` (über `AND_QO`).

Adapter

- **Plug `VALUEO`** (`adapter::types::bidirectional::AD2`): Bitmaske zum Consumer und Echo zurück.

Ereignisse

- `INIT` (EInit, Eingang): Initialisierungsanforderung; startet die Kette `NumericValue_IDA` → `Q_NumericValue_AUDI`.
- `INITO` (EInit, Ausgang): Initialisierungsbestätigung; kommt von `Q_NumericValue_AUDI.INITO`.

Funktionsweise

1. `NumericValue_IDA` liest das VT-Eingabefeld.
2. `AD_TO_AD2` wandelt den Wert in den bidirektionalen AD2-Adapter, der an `VALUEO` ausgegeben wird.
3. Das vom Consumer zurückgemeldete Echo erscheint an `AD_TO_AD2.AD_OUT`, wird per `AD_TO_AUDI` in UDINT gewandelt und von `Q_NumericValue_AUDI` (`u32NewValue`) unter `u16ObjId` auf das VT-Feld geschrieben.

`QI` wird an `NumericValue_IDA.QI` und `Q_NumericValue_AUDI.QI` durchgereicht (kein fester `QI`=TRUE mehr). Die Initialisierung läuft als Kette `INIT` ⟶ `NumericValue_IDA.INIT`, `NumericValue_IDA.INITO` ⟶ `Q_NumericValue_AUDI.INIT`, `Q_NumericValue_AUDI.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`. `QO` ist das `AND` (`AND_QO`, `AND_BOOL_2`) der `QO`-Ausgänge von `Q_NumericValue_AUDI` und `NumericValue_IDA`.

Technische Besonderheiten

- DWORD-/Bitmasken-Pendant zu `NumericValue_TO_AUI2` - für `AOption_AD_DEMUX_N` (Multi-Select) statt `AOption_AUI_DEMUX_N` (Single-Select).
- Kein `AD_TO_AUI`-Zwischenschritt nötig, da AD2 bereits direkt DWORD ist.
- Reine VT-zu-AD2-Brücke ohne Speicher-FB; vgl. `NumericValue_TO_AR2`.

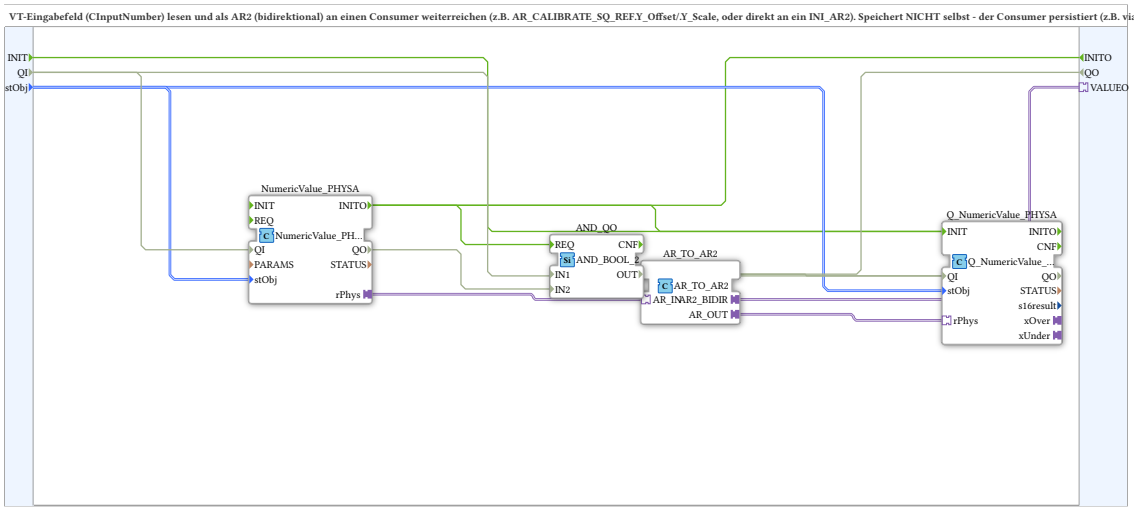
Zusammenfassung

Bidirektionale Brücke vom VT-Eingabefeld zu einem AD2-Plug (Bitmaske) ohne eigene Persistierung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

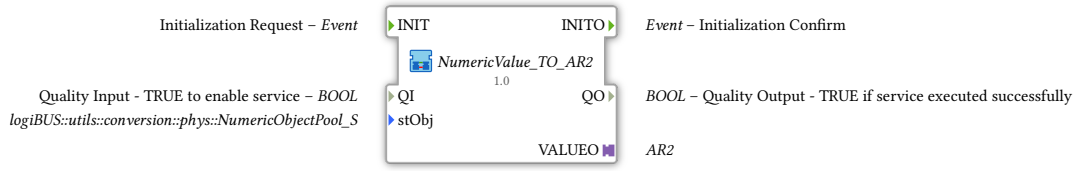
NumericValue_TO_AR2



• – ○ • – ○ • – ○ •

Einleitung

`NumericValue_TO_AR2` liest ein VT-Eingabefeld (`NumericValue_PHYSA`) und reicht den Wert bidirektional als AR2-Adapter (`VALUEO`) an einen Consumer weiter - im Gegensatz zu `INI_IN_AND_STORE_AR2` speichert dieser Baustein selbst **nichts**: die Persistierung obliegt dem Consumer (z. B. `AR_CALIBRATE.Y_Offset/.Y_Scale`, oder direkt ein `INI_AR2`). Was der Consumer ueber denselben AR2-Plug zurueckechot, wird auf das VT-Feld zurueckgeschrieben - auch beim Boot, sodass Eingabefeld und tatsaechlich aktiver Wert synchron bleiben.



Technische Besonderheiten

- Reine VT-zu-AR2-Bruecke: kein Speicher-FB im Netzwerk, im Gegensatz zur INI/NVS-Familie.
- Historischer Hinweis: laut Quellcode-Kommentar aus `INI_IN_AND_STORE_AR2` umbenannt, als der Baustein von einer eigenstaendigen Speicherfunktion zu einer reinen Bruecke wurde.

Qualitätsanschlüsse (QI/QO)

Der Baustein besitzt den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`, Quality Input - TRUE aktiviert den Dienst) und den Datenausgang `QO` (`BOOL`, Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde). `QI` wird an `Q_NumericValue_PHYSA.QI` und `NumericValue_PHYSA.QI` weitergereicht; deren `QO`-Ausgänge werden im Baustein `AND_QO` (`iec61131::booleanOperators::AND_BOOL_2`, `Q_NumericValue_PHYSA.QO` ⟶ `IN1`, `NumericValue_PHYSA.QO` ⟶ `IN2`) zum Ausgang `QO` verknüpft.

Zusätzlich besitzt der Baustein die Ereignisse `INIT` (Eingang, `EInit`, Initialisierungsanforderung) und `INITO` (Ausgang, `EInit`, Initialisierungsbestätigung): Init-Kette: `INIT` ⟶ `Q_NumericValue_PHYSA.INIT`, `Q_NumericValue_PHYSA.INITO` ⟶ `NumericValue_PHYSA.INIT`, `NumericValue_PHYSA.INITO` ⟶ `INITO` und `AND_QO.REQ`.

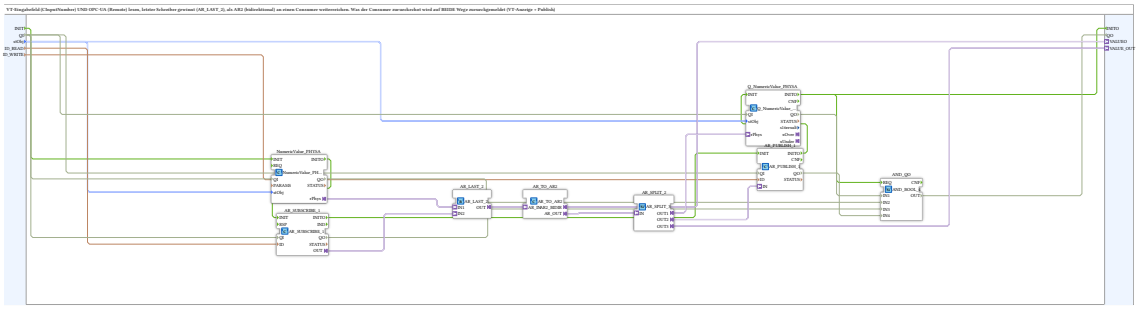
Zusammenfassung

Bidirektionale VT-Eingabefeld-zu-AR2-Bruecke ohne eigene Persistierung - der Consumer speichert, dieser Baustein zeigt nur an und leitet weiter.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

NumericValue_TO_AR2_OPC



• — ○ • — ○ • — ○ •

Einleitung

`NumericValue_TO_AR2_OPC` ist der VT+OPC-UA-Zwilling von `NumericValue_TO_AR2` (VT-only) und `OPC_TO_AR2` (OPC-only): sowohl das VT-Eingabefeld als auch ein per OPC-UA remote geschriebener Wert (`AR_SUBSCRIBE_1`) speisen `AR_LAST_2` - „letzter Schreiber gewinnt“. Das Ergebnis geht als AR2-Adapter (`VALUE0`) an den Consumer; was dieser zurueckechot, wird auf BEIDE Wege zurueckgemeldet: VT-Anzeige (`Q_NumericValue_PHYSA`) und OPC-UA-Publish (`AR_PUBLISH_1`).

Verwendete Funktionsbausteine (FBs)

- **NumericValue_PHYSA**: VT-Eingabefeld, lokaler Wert.
- **AR_SUBSCRIBE_1**: abonniert einen remote geschriebenen REAL-Wert unter `ID_READ`.
- **AR_LAST_2**: verschmilzt beide Quellen nach „letzter Schreiber gewinnt“.
- **AR_TO_AR2**: wandelt den gemergten plain-AR-Wert in den bidirektionalen AR2-Adapter fuer den Consumer.
- **AR_SPLIT_2**: verzweigt das Echo des Consumers auf VT-Anzeige und OPC-Publish.
- **AR_PUBLISH_1**: publiziert den aktuellen/persistierten Wert unter `ID_WRITE` (getriggert von `AR_SUBSCRIBE_1.INITO`, damit auch der initiale Wert sofort publiziert wird).

Technische Besonderheiten

- Merge-Strategie identisch zum bereits bestehenden VT/Web-Merge in `RampLimitFS_TO_logiBUS_QDA_PWM_OPC.SUB` (laut Quellcode-Kommentar).

Ereignis- und Qualitätsanschlüsse (INIT/INITO, QI/QO)

Der Baustein besitzt den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`, Quality Input - TRUE aktiviert den Dienst) und den Datenausgang `QO` (`BOOL`, Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde). `QI` geht an `NumericValue_PHYSA`, `AR_SUBSCRIBE_1`, `AR_PUBLISH_1` und `Q_NumericValue_PHYSA`. `QO` ist der Ausgang der Instanz `AND_QO` (`iec61131::booleanOperators::AND_BOOL_4`), die `Q_NumericValue_PHYSA.QO`, `NumericValue_PHYSA.QO`, `AR_SUBSCRIBE_1.QO` und `AR_PUBLISH_1.QO` verknüpft.

Zusätzlich besitzt der Baustein den Ereigniseingang `INIT` und den Ereignisausgang `INITO` (je `EInit`). Die Initialisierung läuft als Kette: `INIT` ⟶ `NumericValue_PHYSA.INIT`, `NumericValue_PHYSA.INITO` ⟶ `AR_SUBSCRIBE_1.INIT`, `AR_SUBSCRIBE_1.INITO` ⟶ `AR_PUBLISH_1.INIT`, `AR_PUBLISH_1.INITO` ⟶ `Q_NumericValue_PHYSA.INIT`, `Q_NumericValue_PHYSA.INITO` ⟶ `INITO` (und ⟶ `AND_QO.REQ`).

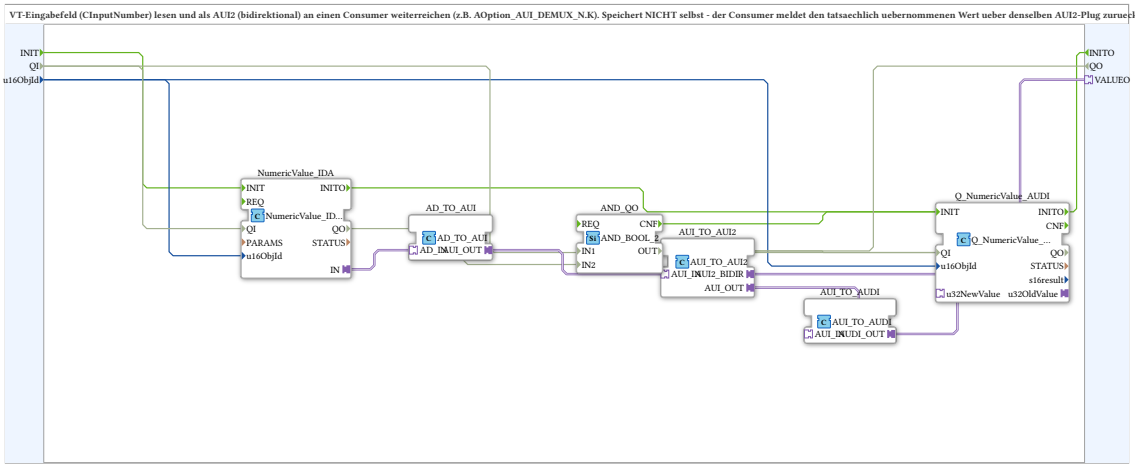
Zusammenfassung

VT- und OPC-UA-Eingabe fuer denselben AR2-Consumer, mit Last-Writer-Wins-Merge und beidseitiger Echo-Rueckmeldung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

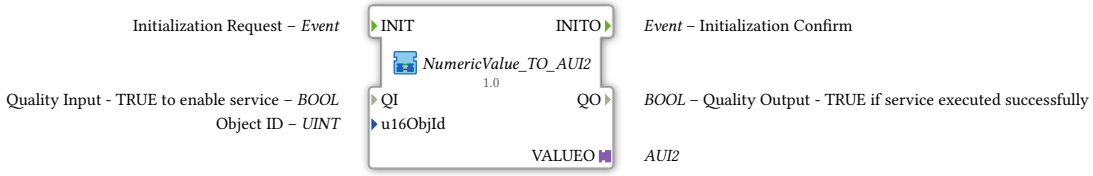
NumericValue_TO_AUI2



• – ○ • – ○ • – ○ •

Einleitung

`NumericValue_TO_AUI2` liest ein VT-Eingabefeld (CInputNumber) und reicht den Wert bidirektional als AUI2-Adapter (`VALUEO`) an einen Consumer weiter (z. B. `AOption_AUI_DEMUX_N.K`). Der Baustein speichert **nicht** selbst: Der Consumer meldet den tatsächlich übernommenen Wert über denselben AUI2-Plug zurück, das Echo wird auf das VT-Feld zurückgeschrieben.



Schnittstelle

Eingangsvariablen

- `QI` (BOOL, Vorgabe `TRUE`): Quality Input - TRUE aktiviert den Dienst.
- `u16ObjId` (UINT, Default `ID_NULL`): ObjectID des VT-Eingabefelds.

Ausgangsvariablen

- `QO` (BOOL): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde (UND aus `Q_NumericValue_AUDI.QO` und `NumericValue_IDA.QO`).

Adapter

- **Plug `VALUEO`** (`adapter::types::bidirectional::AUI2`): Wert zum Consumer und Echo zurück.

Ereignisse

- **INIT** (Ereignis-Eingang, `EInit`): Initialisierungsanforderung
- **INITO** (Ereignis-Ausgang, `EInit`): Initialisierungsbestätigung

Funktionsweise

1. `NumericValue_IDA` liest das VT-Eingabefeld; `AD_TO_AUI` wandelt in AUI.
2. `AUI_TO_AUI2` wandelt in den bidirektionalen AUI2-Adapter, der an `VALUEO` ausgegeben wird.
3. Das vom Consumer zurückgemeldete Echo erscheint an `AUI_TO_AUI2.AUI_OUT`, wird per `AUI_TO_AUDI` in UDINT gewandelt und von `Q_NumericValue_AUDI` (`u32NewValue`) unter `u16ObjId` auf das VT-Feld geschrieben.

`QI` wird an `Q_NumericValue_AUDI.QI` und `NumericValue_IDA.QI` verteilt; `Q_NumericValue_AUDI.QO` und `NumericValue_IDA.QO` werden über `AND_QO` (`iec61131::booleanOperators::AND_BOOL_2`) zu `QO` verknüpft.

Initialisierung: `INIT` ⟶ `NumericValue_IDA.INIT`; `NumericValue_IDA.INITO` ⟶ `Q_NumericValue_AUDI.INIT`; `Q_NumericValue_AUDI.INITO` ⟶ `INITO` (und ⟶ `AND_QO.REQ`).

Technische Besonderheiten

- Ausgelagert aus `Uebung_094c_AX`, wo dies je Dropdown inline dupliziert war.
- Analog zu `NumericValue_TO_AR2`, aber UINT/AUI2 statt REAL/AR2 und einfaches `u16ObjId` statt `stObj` (kein Scale/Offset/Decimals für einen reinen Indexwert nötig).
- Mit INI-Persistenz siehe `NumericValue_TO_AUI2_AND_STORE`.

Zusammenfassung

Bidirektionale Brücke vom VT-Eingabefeld zu einem AUI2-Plug ohne eigene Persistierung.

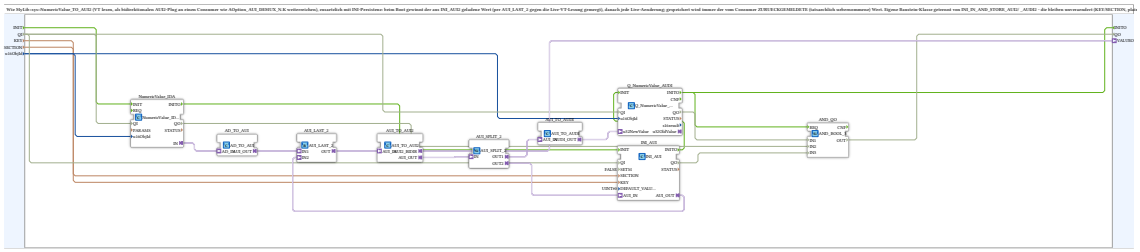
🛠️ Zugehörige Übungen

- Uebung_094b3_AX
- Uebung_094b4_AX
- Uebung_094c2_AX

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

NumericValue_TO_AUI2_AND_STORE



• – ○ • – ○ • – ○ •

Einleitung

`NumericValue_TO_AUI2_AND_STORE` entspricht `NumericValue_TO_AUI2` (VT lesen, als bidirektionalen AUI2-Plug `VALUEO` an einen Consumer wie `AOption_AUI_DEMUX_N.K` weiterreichen), ergänzt um INI-Persistenz: Beim Boot gewinnt der aus der INI-Datei geladene Wert (per `AUI_LAST_2` gegen die Live-VT-Lesung gemergt), danach jede Live-Änderung. Gespeichert wird immer der vom Consumer **zurückgemeldete** (tatsächlich übernommene) Wert. Der Baustein ist eine eigene Klasse getrennt von `INI_IN_AND_STORE_AUI2`/`_AUDI2`, die unverändert bleiben (KEY/SECTION, plain `VALUEO`), da sie in zahlreichen anderen Projekten verwendet werden.

Allgemeines Muster siehe INI_IN_AND_STORE / NVS_IN_AND_STORE (gemeinsames Muster).

Schnittstelle

Ereignis-Eingänge

- `INIT` (`EInit`): Initialization Request.

Ereignis-Ausgänge

- `INITO` (`EInit`): Initialization Confirm.

Eingangsvariablen

- `QI` (BOOL, Vorgabe `TRUE`): Quality Input - TRUE aktiviert den Dienst.
- `KEY` (STRING): Schlüsselname in der INI-Datei.
- `SECTION` (STRING): Abschnittsname in der INI-Datei.
- `u16ObjId` (UINT, Default `ID_NULL`): ObjectID des VT-Eingabefelds.

Ausgangsvariablen

- `QO` (BOOL): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde (UND-Verknüpfung `AND_QO` der `QO`-Ausgänge der inneren Bausteine).

Adapter

- **Plug `VALUEO`** (`adapter::types::bidirectional::AUI2`): Wert zum Consumer und Echo zurück.

Die Ereignisse `INIT`/`INITO` werden als Kette durch die inneren Bausteine geführt (siehe Funktionsweise).

Funktionsweise

1. `NumericValue_IDA` liest das VT-Eingabefeld; `AD_TO_AUI` wandelt in AUI und liefert es an `AUI_LAST_2.IN1`.
2. `AUI_LAST_2` (Last-Writer-Wins) führt zusammen: `IN1` = Live-VT-Änderung, `IN2` = INI-Wert beim Boot. Der Ausgang geht an `AUI_TO_AUI2` (Brücke zum externen Consumer), dessen bidirektionaler Ausgang an `VALUEO`.
3. Das vom Consumer zurückgemeldete Ergebnis erscheint an `AUI_TO_AUI2.AUI_OUT` und wird per `AUI_SPLIT_2` verdoppelt: `OUT1` über `AUI_TO_AUDI` an `Q_NumericValue_AUDI` (`u32NewValue`, VT-Anzeige unter `u16ObjId`), `OUT2` an `INI_AUI` (`eclipse4diac::storage::INI_AUI`, `AUI_IN`; `SETM`=FALSE, `DEFAULT_VALUE=UINT#0`).
4. `INI_AUI.AUI_OUT` führt den geladenen Wert zu `AUI_LAST_2.IN2` zurück.

Initialisierung: `INIT` → `NumericValue_IDA.INIT`; `NumericValue_IDA.INITO` → `INI_AUI.INIT`; `INI_AUI.INITO` → `Q_NumericValue_AUDI.INIT`; `Q_NumericValue_AUDI.INITO` → `INITO` und → `AND_QO.REQ`. Qualität: `QI` → `Q_NumericValue_AUDI.QI`, `NumericValue_IDA.QI`, `INI_AUI.QI`; `Q_NumericValue_AUDI.QO`, `NumericValue_IDA.QO`, `INI_AUI.QO` → `AND_QO.IN1` bis `AND_QO.IN3` (in dieser Reihenfolge); `AND_QO.OUT` → `QO` (`AND_QO`: `iec61131::booleanOperators::AND_BOOL_3`).

Technische Besonderheiten

- `SETM`=FALSE: `INI_AUI` quittiert nur beim Boot automatisch (INIT → INITO → GET → GETO), nicht bei jedem späteren Schreibvorgang. Bei `SETM`=TRUE würde jeder Schreibvorgang ein neues Echo auslösen, das erneut bei `AUI_LAST_2` ankäme und Richtung `VALUEO` geschickt würde - ein Dauerkreislauf.
- Die Live-VT-Lesung geht direkt an `VALUEO` (ein einziges `AUI_TO_AUI2`); gespeichert wird erst das vom Consumer zurückgemeldete Ergebnis über den Split.
- Der Boot-Wert gewinnt in `AUI_LAST_2` einmalig gegen die Live-VT-Lesung.

Zusammenfassung

Wie `NumericValue_TO_AUI2`, mit INI-Persistenz des vom Consumer bestätigten Werts und Boot-Wert-Merge.

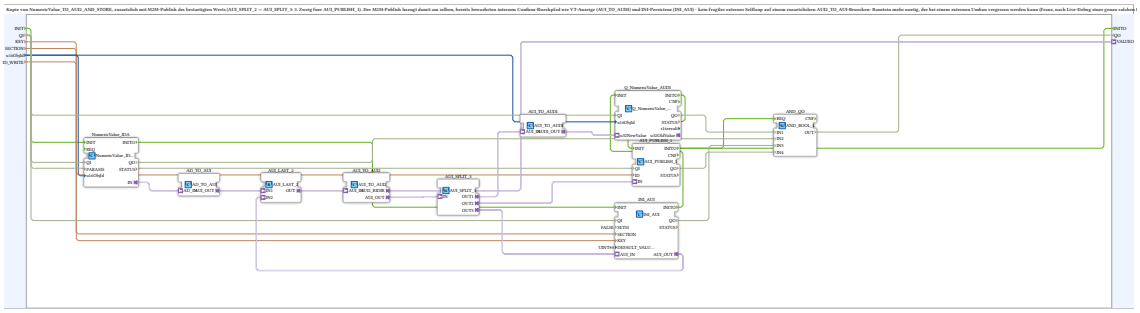
🛠️ Zugehörige Übungen

- Uebung_094b3b_AX
- Uebung_094b4b_AX
- Uebung_094c2b_AX

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

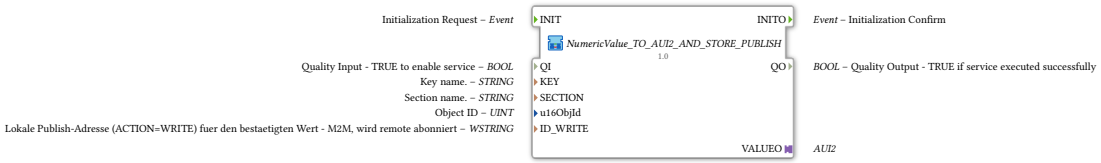
NumericValue_TO_AUI2_AND_STORE_PUBLISH



• – ○ • – ○ • – ○ •

Einleitung

`NumericValue_TO_AUI2_AND_STORE_PUBLISH` ist eine Kopie von `NumericValue_TO_AUI2_AND_STORE`, die zusätzlich den bestätigten Wert per M2M-Publish veröffentlicht: `AUI_SPLIT_2` wurde zu `AUI_SPLIT_3` erweitert, der dritte Zweig speist `AUI_PUBLISH_1`. Der M2M-Publish hängt damit am selben internen Confirm-Rückpfad wie die VT-Anzeige (`AUI_TO_AUDI`) und die INI-Persistenz (`INI_AUI`). Ein zusätzlicher externer Selfloop über einen `AUI2_TO_AUI`-Brückenbaustein ist nicht mehr nötig.



Schnittstelle

Eingangsvariablen

- `QI` (BOOL, Vorgabe `TRUE`): Quality Input - TRUE aktiviert den Dienst.
- `KEY` (STRING): Schlüsselname in der INI-Datei.
- `SECTION` (STRING): Abschnittsname in der INI-Datei.
- `u16ObjId` (UINT, Default `ID_NULL`): ObjectID des VT-Eingabefelds.
- `ID_WRITE` (WSTRING): Lokale Publish-Adresse (`ACTION=WRITE`) für den bestätigten Wert; M2M, wird remote abonniert.

Ausgangsvariablen

- `QO` (BOOL): Quality Output - TRUE, wenn alle beteiligten Bausteine erfolgreich waren (UND-Verknüpfung über `Q_NumericValue_AUDI`, `NumericValue_IDA`, `INI_AUI`, `AUI_PUBLISH_1`).

Adapter

- **Plug `VALUEO`** (`adapter::types::bidirectional::AUI2`): Wert zum Consumer und Echo zurück.

Der Baustein besitzt die Ereignisse `INIT` (Eingang) und `INITO` (Ausgang).

Verwendete Funktionsbausteine (FBs)

- **NumericValue_IDA** (`isobus::UT::io::NumericValue::NumericValue_IDA`): liest das VT-Eingabefeld.
- **AD_TO_AUI** (`adapter::conversion::unidirectional::AD_TO_AUI`): wandelt den Wert in AUI.
- **AUI_LAST_2** (`adapter::events::unidirectional::AUI_LAST_2`): Last-Writer-Wins; beim Boot gewinnt der INI-Wert (`IN2`), danach jede Live-VT-Änderung (`IN1`).
- **AUI_TO_AUI2** (`adapter::conversion::bidirectional::AUI_TO_AUI2`): Brücke zum externen Consumer (z. B. `AOption_AUI_DEMUX_N.K`).
- **AUI_SPLIT_3** (`adapter::events::unidirectional::AUI_SPLIT_3`): dritter Zweig `OUT3` für `INI_AUI`, `OUT2` für `AUI_PUBLISH_1`, `OUT1` für die VT-Anzeige.
- **AUI_TO_AUDI** (`adapter::conversion::unidirectional::AUI_TO_AUDI`) und **Q_NumericValue_AUDI** (`isobus::UT::Q::Q_NumericValue_AUDI`): VT-Anzeige unter `u16ObjId`.
- **INI_AUI** (`eclipse4diac::storage::INI_AUI`): Parameter `SETM=FALSE`, `DEFAULT_VALUE=UINT#0`.

- **AUI_PUBLISH_1** (`adapter::net::AUI_PUBLISH_1`): M2M-Publish des bestätigten Werts auf `ID_WRITE`.
- **AND_QO** (`iec61131::booleanOperators::AND_BOOL_4`): verknüpft die vier `QO`.

Funktionsweise

1. `NumericValue_IDA` liest das VT-Eingabefeld; `AD_TO_AUI` wandelt in AUI und liefert es an `AUI_LAST_2.IN1`.
2. `AUI_LAST_2` geht an `AUI_TO_AUI2`, dessen bidirektionaler Ausgang an `VALUE0`.
3. Das vom Consumer zurückgemeldete Ergebnis erscheint an `AUI_TO_AUI2.AUI_OUT` und wird von `AUI_SPLIT_3` verteilt: `OUT1` -> `AUI_TO_AUDI` -> `Q_NumericValue_AUDI.u32NewValue` (VT-Anzeige), `OUT2` -> `AUI_PUBLISH_1.IN` (M2M-Publish), `OUT3` -> `INI_AUI.AUI_IN` (Persistenz).
4. `INI_AUI.AUI_OUT` führt den geladenen Wert zu `AUI_LAST_2.IN2` zurück.
5. Initialisierungskette: `INIT` -> `NumericValue_IDA` -> `INI_AUI` -> `Q_NumericValue_AUDI` -> `AUI_PUBLISH_1` -> `INITO`. Mit `AUI_PUBLISH_1.INITO` wird zusätzlich `AND_QO.REQ` ausgelöst, dessen Ergebnis an `QO` geht.
6. `QI` wird an `Q_NumericValue_AUDI`, `NumericValue_IDA`, `INI_AUI` und `AUI_PUBLISH_1` verteilt.

Technische Besonderheiten

- `SETM`=FALSE: `INI_AUI` quittiert nur beim Boot automatisch, nicht bei jedem späteren Schreibvorgang. Bei `SETM`=TRUE würde jeder Schreibvorgang ein neues Echo auslösen und einen Dauerkreislauf erzeugen (siehe auch `NumericValue_TO_AUI2_AND_STORE`).
- Der M2M-Publish liegt bewusst im bereits bewährten internen Rückpfad, damit er bei einem externen Umbau nicht verloren gehen kann.
- Der Baustein ist eine eigene Klasse; `NumericValue_TO_AUI2_AND_STORE` bleibt unverändert.

Anwendungsszenarien

- Numerisches VT-Eingabefeld mit INI-Persistenz, bei dem der vom Consumer bestätigte Wert zusätzlich an andere Module veröffentlicht werden soll.

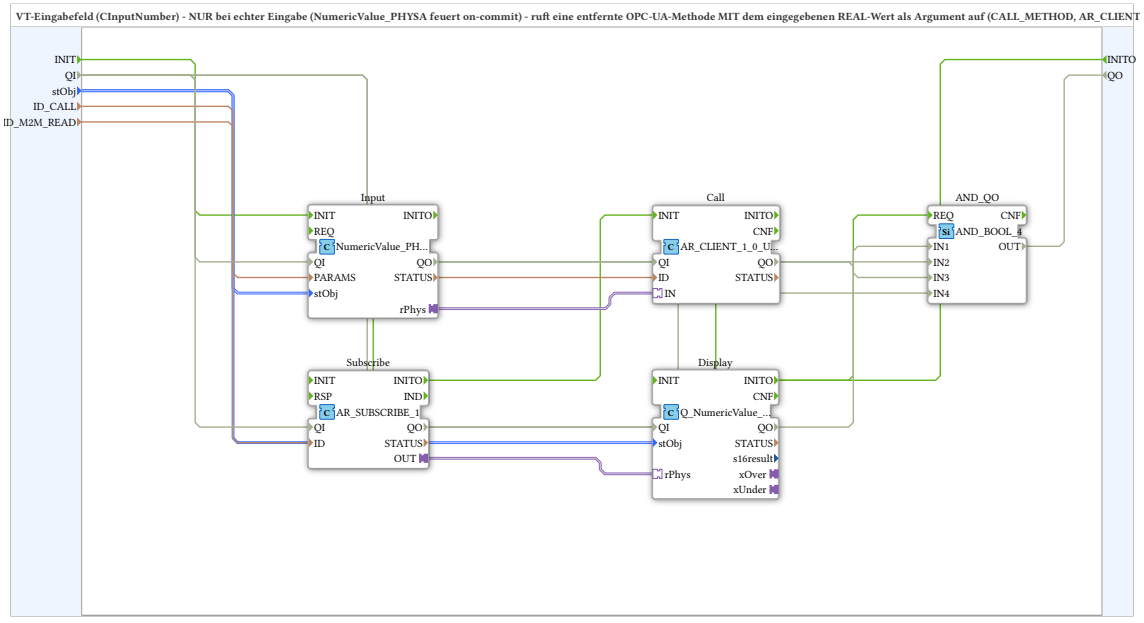
Zusammenfassung

Wie `NumericValue_TO_AUI2_AND_STORE`, mit zusätzlichem M2M-Publish des bestätigten Werts am dritten Zweig von `AUI_SPLIT_3`.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

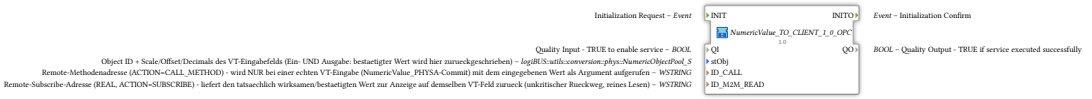
NumericValue_TO_CLIENT_1_0_OPC



• — ○ • — ○ • — ○ •

Einleitung

`NumericValue_TO_CLIENT_1_0_OPC` verbindet ein VT-Eingabefeld (`CInputNumber`) mit einer entfernten **OPC-UA-Methode**: **Nur bei einer echten Eingabe** (`NumericValue_PHYSA` feuert beim Commit) wird die Methode (`CALL_METHOD`, `AR_CLIENT_1_0`) mit dem eingegebenen `REAL`-Wert als Argument aufgerufen. Der Rückweg – Anzeige des tatsächlich wirksamen, bestätigten Werts auf demselben VT-Feld – läuft separat und unkritisch über einen normalen Subscribe (`Q_NumericValue_PHYSA`). Es gibt dadurch keinen Zirkelbezug und kein Merge/Last-Writer-Wins: Auf der Empfängerseite gibt es nur einen Aufrufer pro Methode, serialisiert durch den Methodenaufruf selbst.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialization Request

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialization Confirm

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input - TRUE aktiviert den Dienst
- **stObj** (`logiBUS::utils::conversion::phys::NumericObjectPool_S`): Object-ID und Scale/Offset/Decimals des VT-Eingabefelds; wird für Ein- und Ausgabe verwendet (der bestätigte Wert wird hier zurückgeschrieben). Vorgabe: `(u16ObjId := ID_NULL, r32Scale := 1.0, i32Offset := 0, u8Decimals := 1)`
- **ID_CALL** (`WSTRING`): Remote-Methodenadresse (`ACTION=CALL_METHOD`); wird nur bei einer echten VT-Eingabe mit dem eingegebenen Wert als Argument aufgerufen
- **ID_M2M_READ** (`WSTRING`): Remote-Subscribe-Adresse (`REAL`, `ACTION=SUBSCRIBE`); liefert den tatsächlich wirksamen Wert zur Anzeige auf demselben VT-Feld zurück (reines Lesen)

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde (UND von `Display.QO`, `Input.QO`, `Call.QO` und `Subscribe.QO`)

Adapter

Dieser Funktionsblock verwendet keine Adapter-Sockets/-Plugs an der Schnittstelle; intern werden `AR`-Adapter verwendet.

Verwendete Funktionsbausteine (FBs)

- **Input**: `isobus::UT::io::NumericValue::NumericValue_PHYSA` (`QI` vom Baustein-Eingang) – liefert den eingegebenen physikalischen Wert (`rPhys`) beim Commit
- **Call**: `adapter::net::AR_CLIENT_1_0_UNGATED` (`QI` vom Baustein-Eingang) – ruft die Remote-Methode mit dem Wert als Argument auf
- **Subscribe**: `adapter::net::AR_SUBSCRIBE_1` (`QI` vom Baustein-Eingang) – abonniert den bestätigten Wert
- **Display**: `isobus::UT::Q::Q_NumericValue_PHYSA` – schreibt den Wert auf das VT-Feld
- **AND_QO**: `iec61131::booleanOperators::AND_BOOL_4` – verknüpft die `QO`-Ausgänge von `Display`, `Input`, `Call` und `Subscribe` per UND

Programmablauf und Verbindungen

1. **Hinweg**: `Input.rPhys` ⟶ `Call.IN` (Eingabe ⟶ Methodenaufruf).
2. **Rückweg**: `Subscribe.OUT` ⟶ `Display.rPhys` (bestätigter Wert ⟶ VT-Anzeige).
3. Initialisierungskette: `INIT` ⟶ `Input.INIT`, `Input.INITO` ⟶ `Subscribe.INIT`, `Subscribe.INITO` ⟶ `Call.INIT` (der Client wird erst nach der Initialisierung des Subscribe initialisiert), `Call.INITO` ⟶ `Display.INIT`, `Display.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`.
4. `stObj` ⟶ `Input.stObj` und `Display.stObj`; `ID_CALL` ⟶ `Call.ID`; `ID_M2M_READ` ⟶ `Subscribe.ID`.

5. QI ⟶ Display.QI, Input.QI, Call.QI und Subscribe.QI; Display.QO ⟶ AND_QO.IN1, Input.QO ⟶ AND_QO.IN2, Call.QO ⟶ AND_QO.IN3, Subscribe.QO ⟶ AND_QO.IN4; AND_QO.OUT ⟶ QO

Technische Besonderheiten

- Hin- und Rückweg sind getrennt, deshalb ist kein Merge nötig.
- AR_CLIENT_1_0_UNGATED ist die ungegatete Variante des Clients.

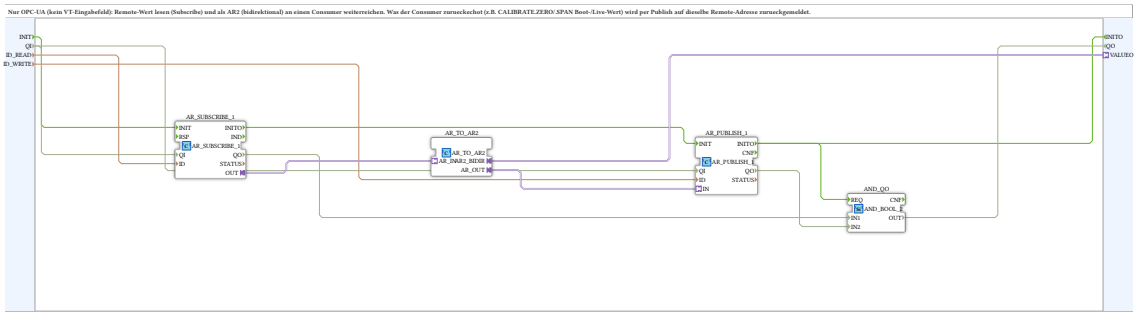
Anwendungsszenarien

- Sollwert-Eingabe am VT, die auf einer anderen Station per OPC-UA-Methode wirksam wird, mit Rückmeldung des tatsächlich gültigen Werts

Fazit

NumericValue_TO_CLIENT_1_0_OPC schreibt VT-Eingaben per OPC-UA-Methodenaufruf zur Zielstation und zeigt den bestätigten Wert unabhängig davon über einen Subscribe an.

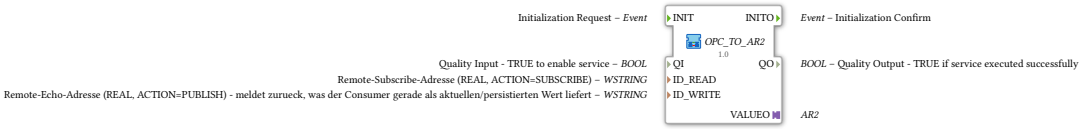
OPC_TO_AR2



• – ○ • – ○ • – ○ •

Einleitung

OPC_TO_AR2 ist der reine OPC-UA-Zwilling von NumericValue_TO_AR2 - kein VT-Eingabefeld beteiligt. Ein remote geschriebener Wert wird per AR_SUBSCRIBE_1 abonniert und als AR2-Adapter (VALUEO) an einen Consumer weitergereicht (z. B. CALIBRATE.ZERO/.SPAN). Was der Consumer zurueckechot (aktueller/persistierter Wert) wird per AR_PUBLISH_1 auf dieselbe Remote-Adresse zurueckgemeldet.



Schnittstelle und Initialisierung

- Ereignisse: INIT (EInit, Eingang) und INITO (EInit, Ausgang, kommt von AR_PUBLISH_1.INITO).
- Daten: QI (BOOL, Vorgabe TRUE) sowie ID_READ und ID_WRITE (WSTRING) als Eingänge, QO (BOOL) als Ausgang; Adapter-Plug VALUEO (AR2).
- Initialisierung: INIT ⟶ AR_SUBSCRIBE_1.INIT, AR_SUBSCRIBE_1.INITO ⟶ AR_PUBLISH_1.INIT, AR_PUBLISH_1.INITO ⟶ INITO und ⟶ AND_QO.REQ.
- Qualität: QI ⟶ AR_SUBSCRIBE_1.QI und AR_PUBLISH_1.QI (kein fester QI=TRUE mehr); AR_SUBSCRIBE_1.QO ⟶ AND_QO.IN1, AR_PUBLISH_1.QO ⟶ AND_QO.IN2 (AND_BOOL_2), AND_QO.OUT ⟶ QO.

Technische Besonderheiten

- AR_PUBLISH_1 wird von AR_SUBSCRIBE_1.INITO getriggert, damit der Boot-Wert sofort publiziert wird, nicht erst bei der ersten Aenderung.

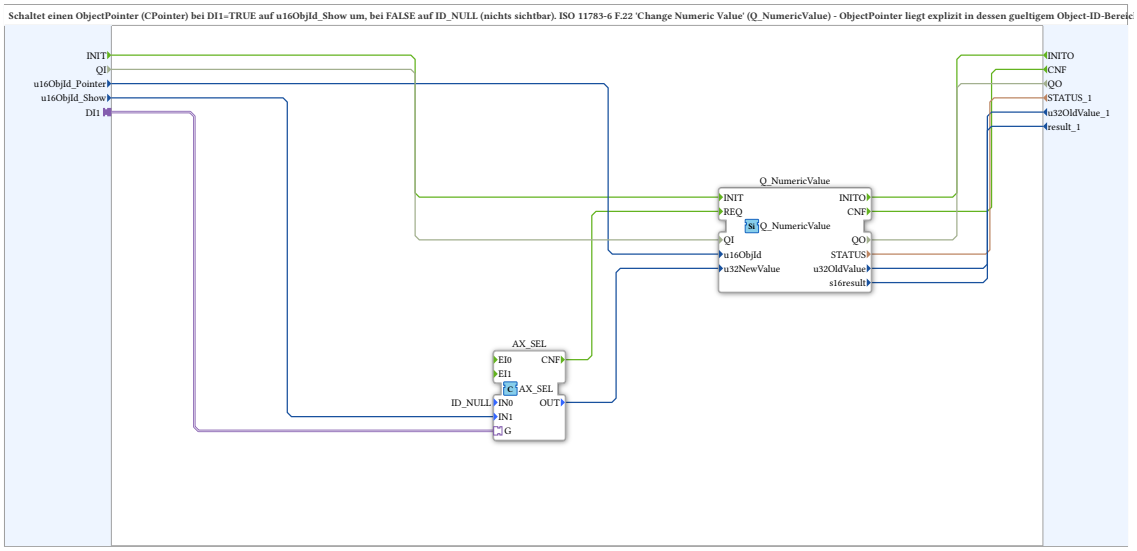
Zusammenfassung

Rein remote (OPC-UA) betriebene AR2-Bruecke ohne jeden VT-Bezug - fuer Consumer, die ausschliesslich per Leitwarte parametriert werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

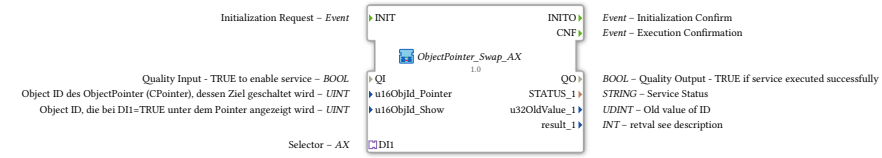
ObjectPointer_Swap_AX



• – ○ • – ○ • – ○ •

Einleitung

`ObjectPointer_Swap_AX` schaltet einen ObjectPointer (CPointer) im VT um: Bei `DI1 = TRUE` zeigt er auf `u16ObjId_Show`, bei FALSE auf `ID_NULL` (nichts sichtbar). Die Umschaltung nutzt ISO 11783-6 F.22 „Change Numeric Value“ (`Q_NumericValue`); der ObjectPointer liegt dabei explizit in dessen gültigem Object-ID-Bereich (27000-27999). F.38 „Change Attribute“ (`Q_Attribute`) darf nicht verwendet werden, weil `AID_OP_PTR.VALUE` dort schreibgeschützt ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung (an `Q_NumericValue.INIT`)

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung (von `Q_NumericValue.INITO`)
- **CNF**: Execution Confirmation, durchgereicht von `Q_NumericValue.CNF`

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input - TRUE aktiviert den Dienst
- **u16ObjId_Pointer** (UINT): Object ID des ObjectPointers (CPointer), dessen Ziel geschaltet wird (Default `ID_NULL`)
- **u16ObjId_Show** (UINT): Object ID, die bei `DI1 = TRUE` unter dem Pointer angezeigt wird (Default `ID_NULL`)

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde
- **STATUS_1** (STRING): Service-Status von `Q_NumericValue`
- **u32OldValue_1** (UDINT): alter Wert der ID
- **result_1** (INT): Rückgabewert (siehe `Q_NumericValue`)

Adapter

- **DI1** (Socket, `adapter::types::unidirectional::AX`): Selektor

Funktionsweise

Das Netzwerk besteht aus zwei Bausteinen:

- `AX_SEL` (`adapter::iec61131::selection::AX_SEL`): Binäre Auswahl mit `IN0 = ID_NULL` (Parameter), `IN1` = `u16ObjId_Show` und Selektor `G` = `DI1`. Ein Ausgangsereignis entsteht nur, wenn sich `G` ändert.
- `Q_NumericValue` (`isobus::UT::Q::Q_NumericValue`): `AX_SEL.OUT` ⟶ `u32NewValue`, `u16ObjId_Pointer` ⟶ `u16ObjId`, ausgelöst durch `AX_SEL.CNF` ⟶ `Q_NumericValue.REQ`. `Q_NumericValue.CNF` ⟶ `CNF`; `STATUS`, `u32OldValue` und `s16result` werden als `STATUS_1`, `u32OldValue_1` und `result_1` ausgegeben.

Ändert sich `DI1`, wird so der neue Zielwert (`u16ObjId_Show` oder `ID_NULL`) als numerischer Wert an den ObjectPointer geschrieben.

Zusätzlich gilt: `INIT` ⟶ `Q_NumericValue.INIT`; `Q_NumericValue.INITO` ⟶ `INITO`; `QI` ⟶ `Q_NumericValue.QI`; `Q_NumericValue.QO` ⟶ `QO`.

Technische Besonderheiten

- Aufbau analog zu `GreenWhiteBackground1_AX` (AX_SEL-Wrapper um einen `Q_`-FB)
- `Q_NumericValue` (F.22) ist der richtige Weg für ObjectPointer; `Q_Attribute` (F.38) wurde verworfen
- Muster siehe Übung `Uebung_015` im Workspace `test_B`

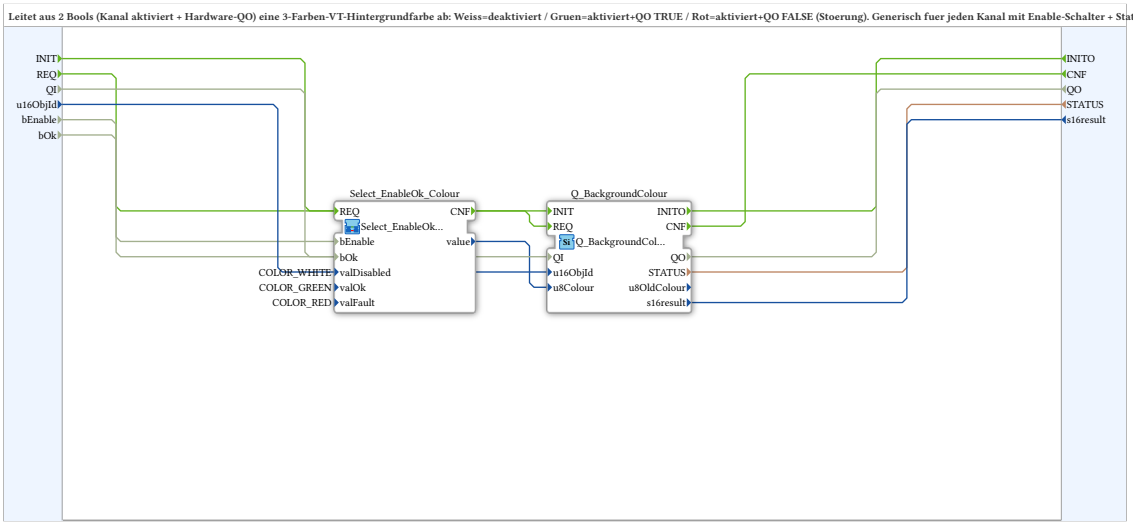
Anwendungsszenarien

- Ein- und Ausblenden eines VT-Objekts (z. B. Warnsymbol) an einem festen Platz über einen AX-Zustand
- Umschalten zwischen einem leeren Zustand und einem Inhalt per ObjectPointer

Fazit

`ObjectPointer_Swap_AX` übersetzt einen AX-Zustand in eine ObjectPointer-Umschaltung zwischen `u16ObjId_Show` und `ID_NULL` über `Q_NumericValue`.

Q_BackgroundColour_EnableOk_A



• – ○ • – ○ • – ○ •

Einleitung

`Q_BackgroundColour_EnableOk_A` leitet aus zwei Booleschen Werten (Kanal aktiviert und Hardware-`QO`) eine 3-Farben-VT-Hintergrundfarbe ab: **Weiß** = deaktiviert, **Grün** = aktiviert und `QO` TRUE, **Rot** = aktiviert und `QO` FALSE (Störung). Generisch für jeden Kanal mit Enable-Schalter und Status-LED (PWM, PI, ...).

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Q_BackgroundColour_EnableOk_A

- **Typ**: SubAppType
- **Verwendete interne Bausteine**:
  - **Select_EnableOk_Colour**: SubApp `MyLib::sys::Select_EnableOk_a` mit `valDisabled=COLOR_WHITE`, `valOk=COLOR_GREEN`, `valFault=COLOR_RED` – wählt anhand von `bEnable` und `bOk` eine der drei Farben
  - **Q_BackgroundColour**: `isobus::UT::Q::Q_BackgroundColour` – schreibt die gewählte Farbe auf das VT-Objekt `u16ObjId`
- **Importe**: `COLOR_GREEN`, `COLOR_RED`, `COLOR_WHITE`

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung (an `Q_BackgroundColour.INIT`)
- **REQ**: Neu auswerten – vom Aufrufer einmal pro möglicher Änderung von `bEnable` oder `bOk` gefeuert (Schalter getoggelt oder frisches `QO` eingetroffen); beide Werte werden zusammen mitgegeben

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung (von `Q_BackgroundColour.INITO`)
- **CNF**: Farbe gesendet (Durchreichung von `Q_BackgroundColour.CNF`)

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input - TRUE aktiviert den Dienst
- **u16ObjId** (`UINT`): Object-ID der Status-Anzeige (`CButton`, nur Hintergrundfarbe, kein Text) – einmalig bei der Instanziierung per Parameter gesetzt
- **bEnable** (`BOOL`): Kanal aktiviert (z. B. Enable-Flipflop `Q`)
- **bOk** (`BOOL`): Hardware bestätigt aktiv (z. B. `logiBUS_..._IDA.QO`) – nur relevant, wenn `bEnable=TRUE`

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde
- **STATUS** (`STRING`): Dienststatus (Durchreichung von `Q_BackgroundColour.STATUS`)
- **s16result** (`INT`): Rückgabewert (Durchreichung von `Q_BackgroundColour.s16result`)

Programmablauf und Verbindungen

- `REQ` ⟶ `Select_EnableOk_Colour.REQ`; `Select_EnableOk_Colour.CNF` ⟶ `Q_BackgroundColour.REQ`; `Q_BackgroundColour.CNF` ⟶ `CNF`
- `bEnable` ⟶ `Select_EnableOk_Colour.bEnable`; `bOk` ⟶ `Select_EnableOk_Colour.bOk`
- `Select_EnableOk_Colour.value` ⟶ `Q_BackgroundColour.u8Colour`
- `u16ObjId` ⟶ `Q_BackgroundColour.u16ObjId`
- `Q_BackgroundColour.STATUS` ⟶ `STATUS`; `Q_BackgroundColour.s16result` ⟶ `s16result`
- `INIT` ⟶ `Q_BackgroundColour.INIT`; `Q_BackgroundColour.INITO` ⟶ `INITO`
- `QI` ⟶ `Q_BackgroundColour.QI`; `Q_BackgroundColour.QO` ⟶ `QO`

Technische Besonderheiten

- Die reine Logik „2 Bool ⟶ 1 von 3 Werten“ ist in `MyLib::sys::Select_EnableOk` ausgelagert (generisch, nicht auf Farben beschränkt); dieser Baustein instanziiert sie nur mit den drei Farben als Parameter
- Variante mit AX-Adaptern: `Q_BackgroundColour_EnableOk_AX`
- Version 1.1 (2026-08-31)

Zustandsübersicht

`bEnable`	`bOk`	Farbe
FALSE	beliebig	Weiß (`COLOR_WHITE`)
TRUE	TRUE	Grün (`COLOR_GREEN`)

TRUE	FALSE	Rot (`COLOR_RED`)

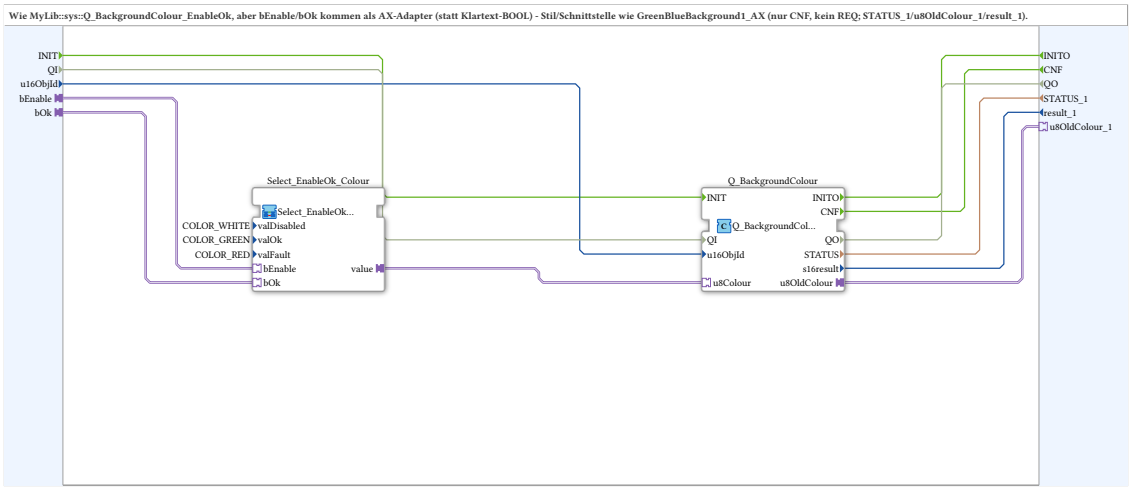
Anwendungsszenarien

- Status-Anzeige eines Kanals mit Enable-Schalter und Hardware-Rückmeldung (PWM, PI, ...)

Fazit

`Q_BackgroundColour_EnableOk_A` setzt die Hintergrundfarbe eines VT-Objekts nach dem Schema Weiß/Grün/Rot aus `bEnable` und `bOk`.

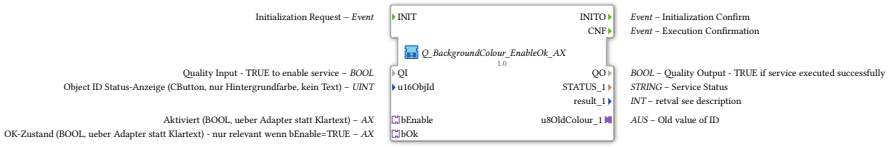
Q_BackgroundColour_EnableOk_AX



• – ○ • – ○ • – ○ •

Einleitung

`Q_BackgroundColour_EnableOk_AX` entspricht `Q_BackgroundColour_EnableOk`, aber `bEnable`/`bOk` kommen als AX-Adapter (statt Klartext-BOOL) - Stil/Schnittstelle wie die bestehende GreenBlueBackground1_AX/GreenRedBackground1_AX-Familie: nur `CNF` (kein `REQ`), zusaetzlich `INIT`/`INITO` sowie `QI`/`QO`, ausserdem `STATUS_1`/`u8OldColour_1`/`result_1` als benannte Ausgaenge.



Verwendete Funktionsbausteine (FBs)

- **Select_EnableOk_Colour** (SubApp, Typ `MyLib::sys::Select_EnableOk_AX`): `valDisabled=COLOR_WHITE`, `valOk=COLOR_GREEN`, `valFault=COLOR_RED`.
- **Q_BackgroundColour** (`isobus::UT::Q::Q_BackgroundColour_AUS`): AUS-Adapter-Variante, schreibt die Farbe auf `u16ObjId` und liefert `u8OldColour_1` (alte Farbe) als AUS-Adapter zurueck.

Qualitätsanschlüsse (QI/QO)

Der Baustein besitzt den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`, Quality Input - TRUE aktiviert den Dienst) und den Datenausgang `QO` (`BOOL`, Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde). `QI` ist mit `Q_BackgroundColour.QI` verbunden, `Q_BackgroundColour.QO` mit `QO`.

Zusätzlich besitzt der Baustein die Ereignisse `INIT` (Eingang, `EInit`, Initialisierungsanforderung) und `INITO` (Ausgang, `EInit`, Initialisierungsbestätigung): `INIT` ⟶ `Q_BackgroundColour.INIT`, `Q_BackgroundColour.INITO` ⟶ `INITO`.

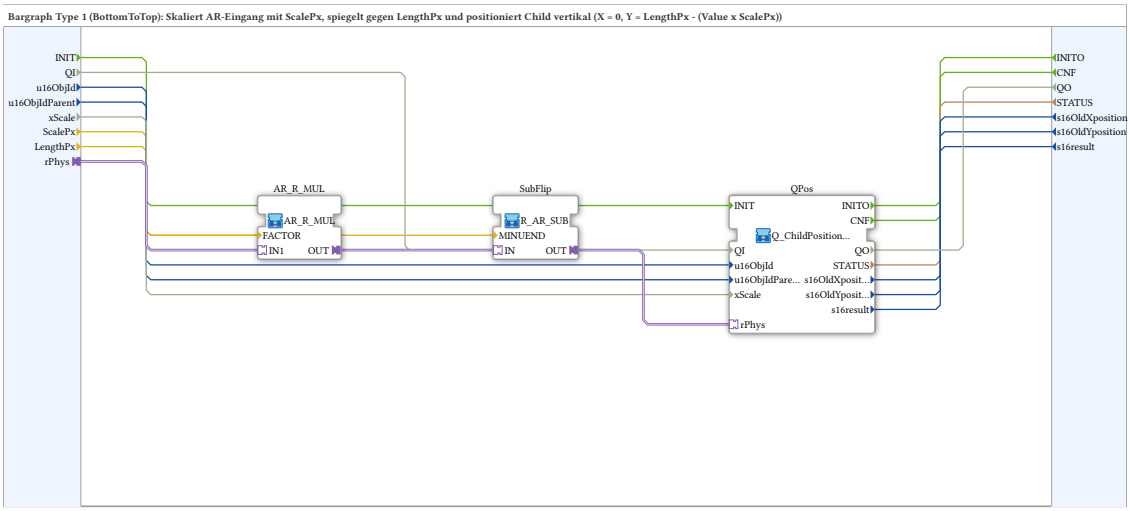
Zusammenfassung

AX-Adapter-Variante von `Q_BackgroundColour_EnableOk`, passend zur bestehenden GreenBlueBackground1_AX/GreenRedBackground1_AX-Familie.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

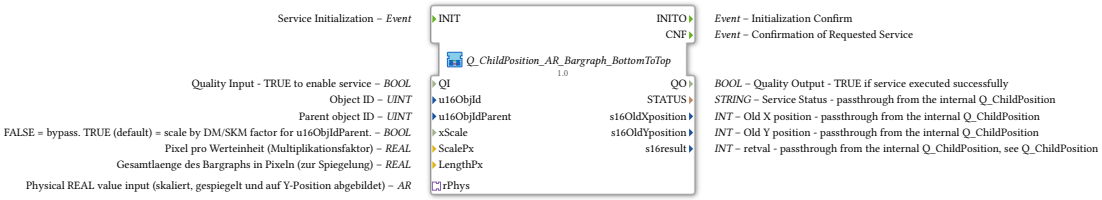
Q_ChildPosition_AR_Bargraph_BottomToTop



• — ○ • — ○ • — ○ •

Einleitung

`Q_ChildPosition_AR_Bargraph_BottomToTop` ist ein **Bargraph Typ 1 (von unten nach oben)**: Der AR-Eingang wird mit `ScalePx` skaliert, an `LengthPx` gespiegelt und als vertikale Position des Child-Objekts im VT ausgegeben (`X = 0`, `Y = LengthPx - (Wert × ScalePx)`). Ein größerer Wert schiebt das Child damit nach oben.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierung (an `QPos.INIT`)

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung (von `QPos.INITO`)
- **CNF**: Bestätigung des angeforderten Dienstes (von `QPos.CNF`)

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input – TRUE aktiviert den Dienst (an `QPos.QI`)
- **u16ObjId** (`UINT`, Vorgabe `ID_NULL`): Object-ID des Child-Objekts
- **u16ObjIdParent** (`UINT`, Vorgabe `ID_NULL`): Object-ID des Parent-Objekts
- **xScale** (`BOOL`, Vorgabe `TRUE`): `FALSE` = Bypass; `TRUE` = Skalierung mit dem DM/SKM-Faktor für `u16ObjIdParent`
- **ScalePx** (`REAL`, Vorgabe `1.0`): Pixel pro Werteinheit (Multiplikationsfaktor)
- **LengthPx** (`REAL`, Vorgabe `100.0`): Gesamtlänge des Bargraphs in Pixeln (zur Spiegelung)

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde (von `QPos.QO`)
- **STATUS** (`STRING`): Dienststatus (Durchreichung vom inneren `Q_ChildPosition`)
- **s16OldXposition** (`INT`): Alte X-Position (Durchreichung)

- **s16OldYposition** (`INT`): Alte Y-Position (Durchreichung)
- **s16result** (`INT`): Rückgabewert (Durchreichung, siehe `Q_ChildPosition`)

Adapter

- **rPhys** (Socket, `adapter::types::unidirectional::AR`): Physikalischer `REAL`-Eingangswert (wird skaliert, gespiegelt und auf die Y-Position abgebildet)

Verwendete Funktionsbausteine (FBs)

- **AR_R_MUL**: SubApp `MyLib::sys::AR_R_MUL` – multipliziert den Wert mit `ScalePx` (`FACTOR`)
- **SubFlip**: SubApp `MyLib::sys::R_AR_SUB` – subtrahiert vom `MINUEND` (`LengthPx`) und spiegelt damit
- **QPos**: SubApp `MyLib::sys::Q_ChildPosition_AR_Vertikal` – setzt die vertikale Child-Position

Programmablauf und Verbindungen

1. `rPhys` ⟶ `AR_R_MUL.IN1`; `ScalePx` ⟶ `AR_R_MUL.FACTOR`.
2. `AR_R_MUL.OUT` ⟶ `SubFlip.IN`; `LengthPx` ⟶ `SubFlip.MINUEND`.
3. `SubFlip.OUT` ⟶ `QPos.rPhys`.
4. `INIT` ⟶ `QPos.INIT`; `QPos.INITO` ⟶ `INITO`; `QPos.CNF` ⟶ `CNF`; `QI` ⟶ `QPos.QI`; `QPos.QO` ⟶ `QO`.
5. `u16ObjId`, `u16ObjIdParent`, `xScale` ⟶ `QPos`; `STATUS`, `s16OldXposition`, `s16OldYposition`, `s16result` von `QPos` nach außen.

Technische Besonderheiten

- Die Spiegelung ist nötig, weil die VT-Y-Koordinate von oben nach unten wächst.
- `xScale` hat hier die Vorgabe `TRUE` (bei `Q_ChildPosition_AR_Horizontal` dagegen `FALSE`).

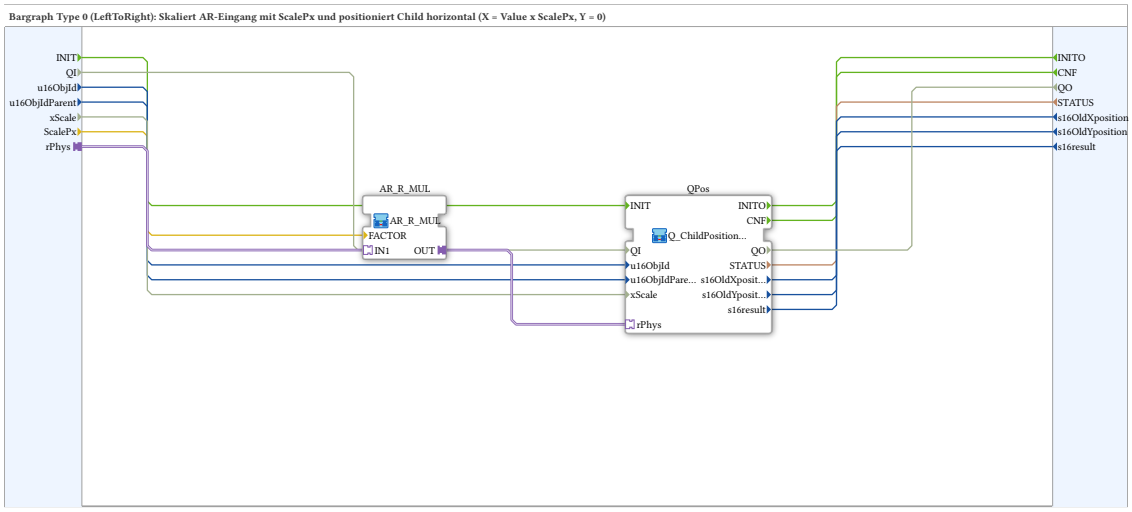
Anwendungsszenarien

- Senkrechter Balken/Füllstandsanzeige, der von unten nach oben wächst

Fazit

`Q_ChildPosition_AR_Bargraph_BottomToTop` bildet einen AR-Wert über Skalierung und Spiegelung auf die vertikale Position eines VT-Child-Objekts ab.

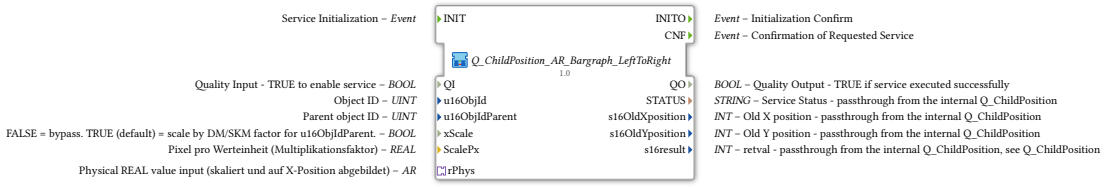
Q_ChildPosition_AR_Bargraph_LeftToRight



• – ○ • – ○ • – ○ •

Einleitung

`Q_ChildPosition_AR_Bargraph_LeftToRight` realisiert einen Bargraph vom Typ 0 (von links nach rechts): Der AR-Eingang `rPhys` wird mit `ScalePx` (Pixel pro Werteinheit) skaliert und positioniert das Kind-Objekt horizontal (X = Wert × `ScalePx`, Y = 0).



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung (`EInit`)

Ereignis-Ausgänge

- **INITO**: Initialisierungs-Bestätigung (`EInit`)
- **CNF**: Bestätigung des angeforderten Service

Daten-Eingänge

- **QI** (BOOL): Quality Input - TRUE aktiviert den Dienst (Default `TRUE`), geht an `QPos.QI`
- **u16ObjId** (UINT): Object ID des Kind-Objekts (Default `ID_NULL`)
- **u16ObjIdParent** (UINT): Object ID des Eltern-Objekts (Default `ID_NULL`)
- **xScale** (BOOL): FALSE = Bypass; TRUE (Default) = mit dem DM/SKM-Faktor für `u16ObjIdParent` skalieren
- **ScalePx** (REAL): Pixel pro Werteinheit (Multiplikationsfaktor), Default `REAL#1.0`

Daten-Ausgänge

- **QO** (BOOL): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde, kommt von `QPos.QO`
- **STATUS** (STRING): Service-Status, durchgereicht vom internen `Q_ChildPosition`
- **s16OldXposition** (INT): alte X-Position, durchgereicht
- **s16OldYposition** (INT): alte Y-Position, durchgereicht
- **s16result** (INT): Rückgabewert, durchgereicht (siehe `Q_ChildPosition`)

Adapter

- **rPhys** (Socket, `adapter::types::unidirectional::AR`): physikalischer REAL-Wert (wird skaliert und auf die X-Position abgebildet)

Funktionsweise

Das Netzwerk besteht aus zwei SubApps:

- `AR_R_MUL` (`MyLib::sys::AR_R_MUL`): `rPhys` ⟶ `AR_R_MUL.IN1`, `ScalePx` ⟶ `AR_R_MUL.FACTOR`; multipliziert den AR-Wert mit dem Faktor.
- `QPos` (`MyLib::sys::Q_ChildPosition_AR_Horizontal`): `AR_R_MUL.OUT` ⟶ `QPos.rPhys`; setzt die X-Position des Kind-Objekts über den AR-Adapter (Y fest 0).

Die Ereignisse `INIT`, `INITO` und `CNF` sowie `QI`, `u16ObjId`, `u16ObjIdParent`, `xScale` und die Ausgänge `QO` und die vier Statusausgänge werden direkt von bzw. nach `QPos` durchgereicht.

Technische Besonderheiten

- Typ 0 (LeftToRight) der Bargraph-Varianten: Die Position wächst mit dem Wert in X-Richtung, Y bleibt 0
- `xScale` hat hier den Default TRUE (Skalierung mit DM/SKM-Faktor), im Gegensatz zu `Q_ChildPosition_AR_Vertikal` mit Default FALSE
- Zustandslos auf SubApp-Ebene; alle Statusausgänge stammen vom inneren `Q_ChildPosition`

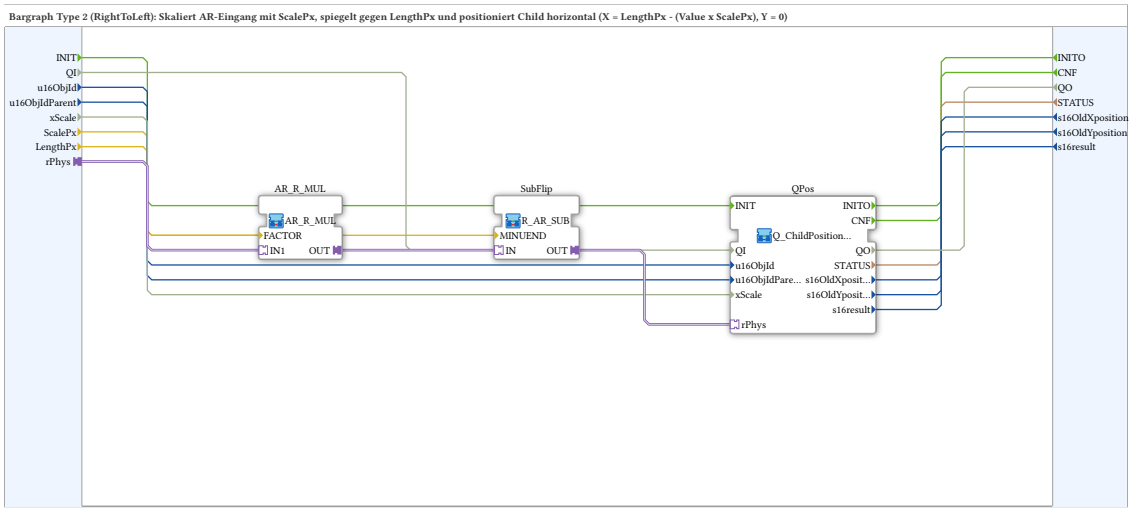
Anwendungsszenarien

- Balkenanzeige, bei der ein Kind-Objekt proportional zu einem Messwert horizontal verschoben wird

Fazit

`Q_ChildPosition_AR_Bargraph_LeftToRight` skaliert einen AR-Wert mit `ScalePx` und setzt damit die X-Position eines Kind-Objekts für einen Links-nach-rechts-Bargraph.

Q_ChildPosition_AR_Bargraph_RightToLeft



• – ○ • – ○ • – ○ •

Einleitung

`Q_ChildPosition_AR_Bargraph_RightToLeft` ist ein Bargraph vom Typ 2 (von rechts nach links): Der `AR`-Eingang wird mit `ScalePx` skaliert, gegen `LengthPx` gespiegelt und als horizontale Position eines Kindobjekts im VT gesetzt (`X = LengthPx - (Wert × ScalePx)`, `Y = 0`). Gegenstück zu `Q_ChildPosition_AR_Bargraph_LeftToRight`.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Service-Initialisierung (durchgereicht an `QPos.INIT`)

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung (von `QPos.INITO`)
- **CNF**: Bestätigung der angeforderten Positionierung (von `QPos.CNF`)

Daten-Eingänge

- **QI** (`BOOL`, Default `TRUE`): Quality Input – TRUE gibt den Dienst frei (durchgereicht an `QPos.QI`)
- **u16ObjId** (`UINT`): Object ID des Kindobjekts (Default `ID_NULL`)
- **u16ObjIdParent** (`UINT`): Object ID des Elternobjekts (Default `ID_NULL`)
- **xScale** (`BOOL`): `FALSE` = Bypass; `TRUE` (Default) = Skalierung mit dem DM/SKM-Faktor für `u16ObjIdParent`
- **ScalePx** (`REAL`): Pixel pro Werteinheit (Multiplikationsfaktor), Default `REAL#1.0`
- **LengthPx** (`REAL`): Gesamtlänge des Bargraphs in Pixeln (zur Spiegelung), Default `REAL#100.0`

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde (von `QPos.QO`)
- **STATUS** (`STRING`): Service-Status, durchgereicht vom internen `Q_ChildPosition`
- **s16OldXposition** (`INT`): alte X-Position, durchgereicht
- **s16OldYposition** (`INT`): alte Y-Position, durchgereicht
- **s16result** (`INT`): Rückgabewert, durchgereicht vom internen `Q_ChildPosition`

Adapter

- **rPhys** (Socket, `adapter::types::unidirectional::AR`): physikalischer `REAL`-Wert; wird skaliert, gespiegelt und auf die X-Position abgebildet

Verwendete Funktionsbausteine (FBs)

- **`AR_R_MUL`** (SubApp `MyLib::sys::AR_R_MUL`, AR_R_MUL): multipliziert den Wert mit `ScalePx`
- **`SubFlip`** (SubApp `MyLib::sys::R_AR_SUB`): zieht den skalierten Wert von `LengthPx` ab (`MINUEND = LengthPx`)
- **`QPos`** (SubApp `MyLib::sys::Q_ChildPosition_AR_Horizontal`, Q_ChildPosition_AR_Horizontal): setzt die horizontale Position des Kindobjekts

Funktionsweise

- `rPhys` ⟶ `AR_R_MUL.IN1`; `ScalePx` ⟶ `AR_R_MUL.FACTOR`
- `AR_R_MUL.OUT` ⟶ `SubFlip.IN`; `LengthPx` ⟶ `SubFlip.MINUEND`
- `SubFlip.OUT` ⟶ `QPos.rPhys`
- `INIT` ⟶ `QPos.INIT`; `QPos.INITO` ⟶ `INITO`; `QPos.CNF` ⟶ `CNF`
- `QI` ⟶ `QPos.QI`; `QPos.QO` ⟶ `QO`
- `u16ObjId`, `u16ObjIdParent`, `xScale` ⟶ gleichnamige Eingänge von `QPos`
- `QPos.STATUS`, `QPos.s16OldXposition`, `QPos.s16OldYposition`, `QPos.s16result` ⟶ gleichnamige Ausgänge

Technische Besonderheiten

- Die Spiegelung (`LengthPx - Wert × ScalePx`) lässt den Balken von rechts nach links wachsen
- Alle Status- und Positionsausgänge sind reine Durchreichungen des internen `Q_ChildPosition`

Anwendungsszenarien

- Horizontale Balkenanzeige im VT, die von rechts nach links wächst (z. B. Füllstand, Auslastung)

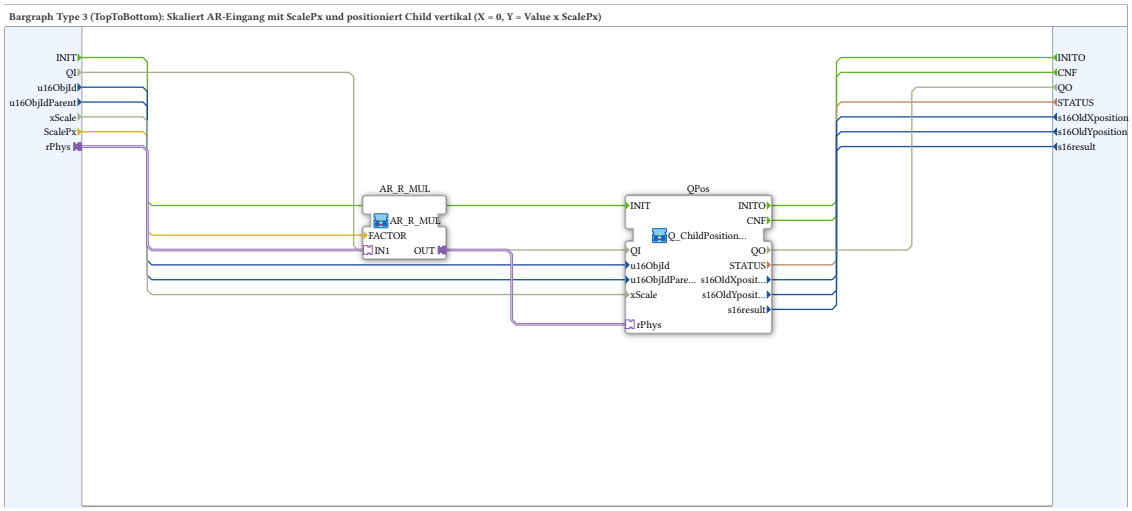
Zusammenfassung

`Q_ChildPosition_AR_Bargraph_RightToLeft` bildet einen `AR`-Wert gespiegelt auf die X-Position eines VT-Kindobjekts ab.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

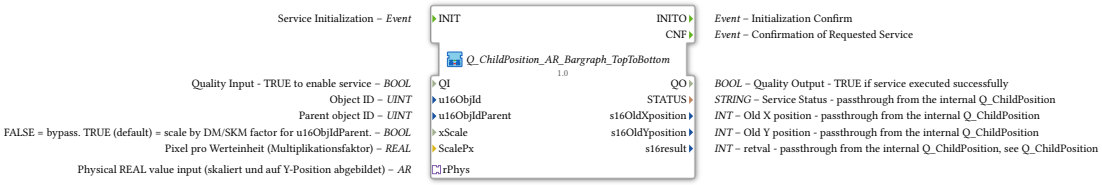
Q_ChildPosition_AR_Bargraph_TopToBottom



• – ○ • – ○ • – ○ •

Einleitung

`Q_ChildPosition_AR_Bargraph_TopToBottom` ist ein **Bargraph Typ 3 (von oben nach unten)**: Der AR-Eingang wird mit `ScalePx` skaliert und als vertikale Position des Child-Objekts im VT ausgegeben (`X = 0`, `Y = Wert × ScalePx`). Ein größerer Wert schiebt das Child damit nach unten. Im Gegensatz zu `Q_ChildPosition_AR_Bargraph_BottomToTop` erfolgt keine Spiegelung.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierung (an `QPos.INIT`)

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung (von `QPos.INITO`)
- **CNF**: Bestätigung des angeforderten Dienstes (von `QPos.CNF`)

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input, TRUE aktiviert den Dienst
- **u16ObjId** (`UINT`, Vorgabe `ID_NULL`): Object-ID des Child-Objekts
- **u16ObjIdParent** (`UINT`, Vorgabe `ID_NULL`): Object-ID des Parent-Objekts
- **xScale** (`BOOL`, Vorgabe `TRUE`): `FALSE` = Bypass; `TRUE` = Skalierung mit dem DM/SKM-Faktor für `u16ObjIdParent`
- **ScalePx** (`REAL`, Vorgabe `1.0`): Pixel pro Werteinheit (Multiplikationsfaktor)

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output (von `QPos.QO`)
- **STATUS** (`STRING`): Dienststatus (Durchreichung vom inneren `Q_ChildPosition`)
- **s16OldXposition** (`INT`): Alte X-Position (Durchreichung)
- **s16OldYposition** (`INT`): Alte Y-Position (Durchreichung)
- **s16result** (`INT`): Rückgabewert (Durchreichung, siehe `Q_ChildPosition`)

Adapter

- **rPhys** (Socket, `adapter::types::unidirectional::AR`): Physikalischer `REAL`-Eingangswert (wird skaliert und auf die Y-Position abgebildet)

Verwendete Funktionsbausteine (FBs)

- **AR_R_MUL**: SubApp `MyLib::sys::AR_R_MUL` – multipliziert den Wert mit `ScalePx` (`FACTOR`)
- **QPos**: SubApp `MyLib::sys::Q_ChildPosition_AR_Vertikal` – setzt die vertikale Child-Position

Programmablauf und Verbindungen

1. `rPhys` ⟶ `AR_R_MUL.IN1`; `ScalePx` ⟶ `AR_R_MUL.FACTOR`.
2. `AR_R_MUL.OUT` ⟶ `QPos.rPhys`.
3. `INIT` ⟶ `QPos.INIT`; `QPos.INITO` ⟶ `INITO`; `QPos.CNF` ⟶ `CNF`; `QI` ⟶ `QPos.QI`; `QPos.QO` ⟶ `QO`.
4. `u16ObjId`, `u16ObjIdParent`, `xScale` ⟶ `QPos`; `STATUS`, `s16OldXposition`, `s16OldYposition`, `s16result` von `QPos` nach außen.

Technische Besonderheiten

- Keine Spiegelung: Da die VT-Y-Koordinate von oben nach unten wächst, entspricht ein größerer Wert direkt einer weiter unten liegenden Position.
- `xScale` hat hier die Vorgabe `TRUE`.
- Version 1.0 (2026-09-27): `AR_R_MUL` + `Q_ChildPosition_AR_Vertikal`

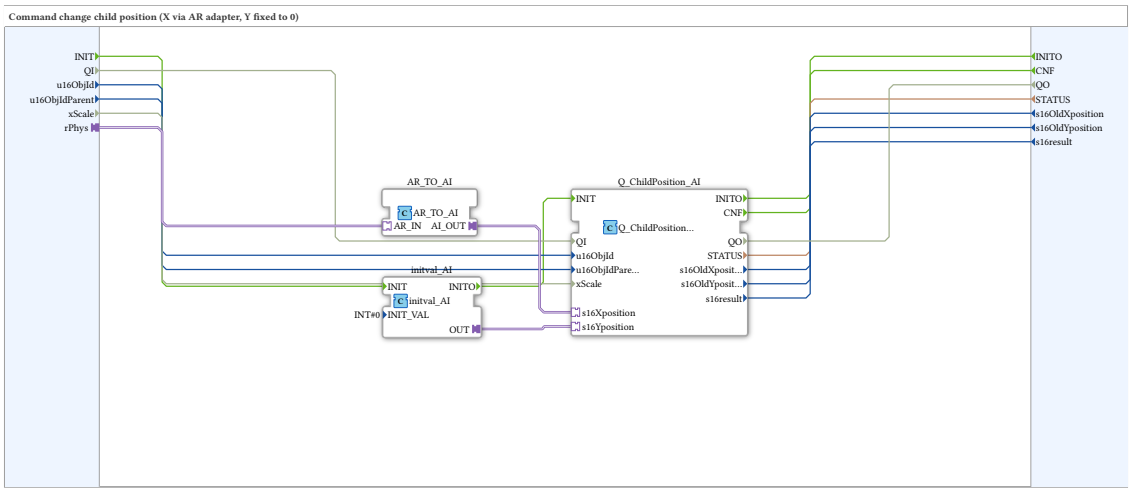
Anwendungsszenarien

- Senkrechter Balken/Anzeige, der von oben nach unten wächst

Fazit

`Q_ChildPosition_AR_Bargraph_TopToBottom` bildet einen AR-Wert über Skalierung direkt auf die vertikale Position eines VT-Child-Objekts ab.

Q_ChildPosition_AR_Horizontal



• – ○ • – ○ • – ○ •

Einleitung

`Q_ChildPosition_AR_Horizontal` ändert die Position eines VT-Child-Objekts (Kommando *Change Child Position*): Die **X-Position** kommt über einen AR-Adapter, die **Y-Position ist fest 0**. Der `REAL`-Wert wird dazu in `INT` gewandelt und auf die X-Position abgebildet.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierung (an `initval_AI.INIT`)

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung (von `Q_ChildPosition_AI.INITO`)
- **CNF**: Bestätigung des angeforderten Dienstes (von `Q_ChildPosition_AI.CNF`)

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input - TRUE aktiviert den Dienst
- **u16ObjId** (`UINT`, Vorgabe `ID_NULL`): Object-ID des Child-Objekts
- **u16ObjIdParent** (`UINT`, Vorgabe `ID_NULL`): Object-ID des Parent-Objekts
- **xScale** (`BOOL`, Vorgabe `FALSE`): `FALSE` = Bypass, X-/Y-Position werden unverändert gesendet; `TRUE` = Skalierung mit dem DM/SKM-Faktor für `u16ObjIdParent`

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde
- **STATUS** (`STRING`): Dienststatus (Durchreichung vom inneren `Q_ChildPosition`)
- **s16OldXposition** (`INT`): Alte X-Position (Durchreichung)
- **s16OldYposition** (`INT`): Alte Y-Position (Durchreichung)
- **s16result** (`INT`): Rückgabewert (Durchreichung, siehe `Q_ChildPosition`)

Adapter

- **rPhys** (Socket, `adapter::types::unidirectional::AR`): Physikalischer `REAL`-Eingangswert (wird in `INT` gewandelt und auf die X-Position abgebildet)

Verwendete Funktionsbausteine (FBs)

- **AR_TO_AI**: `adapter::conversion::unidirectional::AR_TO_AI` – wandelt `REAL` in `INT`
- **initval_AI**: `adapter::types::unidirectional::AI::initval::initval_AI (INIT_VAL=0)` – liefert die konstante Y-Position 0
- **Q_ChildPosition_AI**: `isobus::UT::Q::Q_ChildPosition_AI` – sendet die Positionsänderung

Programmablauf und Verbindungen

1. `rPhys` ⟶ `AR_TO_AI.AR_IN`; `AR_TO_AI.AI_OUT` ⟶ `Q_ChildPosition_AI.s16Xposition`.
2. `initval_AI.OUT` ⟶ `Q_ChildPosition_AI.s16Yposition` (konstant 0).
3. `INIT` ⟶ `initval_AI.INIT`; `initval_AI.INITO` ⟶ `Q_ChildPosition_AI.INIT`; `Q_ChildPosition_AI.INITO` ⟶ `INITO`; `Q_ChildPosition_AI.CNF` ⟶ `CNF`.
4. `u16ObjId`, `u16ObjIdParent`, `xScale` ⟶ `Q_ChildPosition_AI`; Statusausgänge von dort nach außen.
5. `QI` ⟶ `Q_ChildPosition_AI.QI`; `Q_ChildPosition_AI.QO` ⟶ `QO`

Technische Besonderheiten

- Die Initialisierung läuft in Reihe: erst `initval_AI`, dann `Q_ChildPosition_AI`.
- Im Gegensatz zu `Q_ChildPosition_AR_Bargraph_BottomToTop` erfolgt keine Skalierung/Spiegelung; `xScale` hat die Vorgabe `FALSE`.

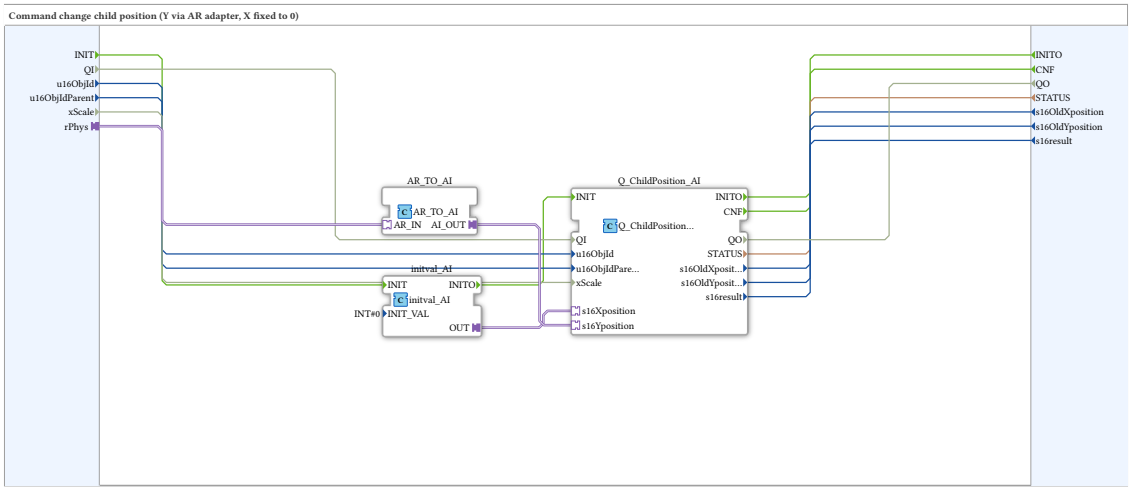
Anwendungsszenarien

- Horizontales Verschieben eines VT-Objekts (z. B. Zeiger oder Balken) abhängig von einem `REAL`-Wert

Fazit

`Q_ChildPosition_AR_Horizontal` verschiebt ein VT-Child-Objekt horizontal anhand eines AR-Werts bei fester Y-Position 0.

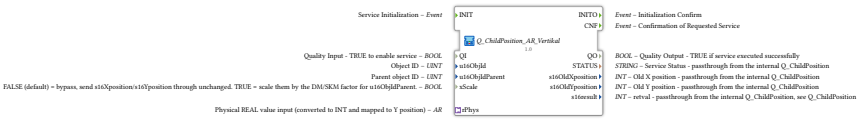
Q_ChildPosition_AR_Vertikal



• – ○ • – ○ • – ○ •

Einleitung

`Q_ChildPosition_AR_Vertikal` setzt die Position eines Kind-Objekts im VT: Die Y-Position kommt über einen AR-Adapter (`rPhys`, in INT umgewandelt), die X-Position ist fest 0.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung (`EInit`)

Ereignis-Ausgänge

- **INITO**: Initialisierungs-Bestätigung (`EInit`)
- **CNF**: Bestätigung des angeforderten Service

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input - TRUE aktiviert den Dienst
- **u16ObjId** (UINT): Object ID des Kind-Objekts (Default `ID_NULL`)
- **u16ObjIdParent** (UINT): Object ID des Eltern-Objekts (Default `ID_NULL`)
- **xScale** (BOOL): FALSE (Default) = Bypass, X/Y-Position unverändert senden; TRUE = mit dem DM/SKM-Faktor für `u16ObjIdParent` skalieren

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde
- **STATUS** (STRING): Service-Status, durchgereicht vom internen `Q_ChildPosition`
- **s16OldXposition** (INT): alte X-Position, durchgereicht
- **s16OldYposition** (INT): alte Y-Position, durchgereicht
- **s16result** (INT): Rückgabewert, durchgereicht (siehe `Q_ChildPosition`)

Adapter

- **rPhys** (Socket, `adapter::types::unidirectional::AR`): physikalischer REAL-Wert (wird in INT umgewandelt und auf die Y-Position abgebildet)

Funktionsweise

Das Netzwerk besteht aus drei Bausteinen:

- `AR_TO_AI` (`adapter::conversion::unidirectional::AR_TO_AI`): `rPhys` ⟶ `AR_TO_AI.AR_IN`; wandelt den REAL-Wert in einen INT-Adapterwert.
- `initval_AI` (`adapter::types::unidirectional::AI::initval::initval_AI`, `INIT_VAL = 0`): liefert den festen Wert 0 für die X-Position.
- `Q_ChildPosition_AI` (`isobus::UT::Q::Q_ChildPosition_AI`): `initval_AI.OUT` ⟶ `s16Xposition`, `AR_TO_AI.AI_OUT` ⟶ `s16Yposition`.

Initialisierung: `INIT` ⟶ `initval_AI.INIT`, `initval_AI.INITO` ⟶ `Q_ChildPosition_AI.INIT`, `Q_ChildPosition_AI.INITO` ⟶ `INITO`. `Q_ChildPosition_AI.CNF` ⟶ `CNF`. `u16ObjId`, `u16ObjIdParent`, `xScale` und die vier Ausgänge gehen direkt von bzw. nach `Q_ChildPosition_AI`.

`QI` wird an `Q_ChildPosition_AI.QI` durchgereicht, `Q_ChildPosition_AI.QO` an `QO`.

Technische Besonderheiten

- Die Initialisierung läuft in Reihe: zuerst `initval_AI`, danach `Q_ChildPosition_AI`
- `xScale` hat hier den Default FALSE (Bypass); im Gegensatz dazu hat `Q_ChildPosition_AR_Bargraph_LeftToRight` den Default TRUE
- Es erfolgt keine Skalierung des AR-Werts: `rPhys` wird nur in INT umgewandelt und direkt als Y-Position verwendet

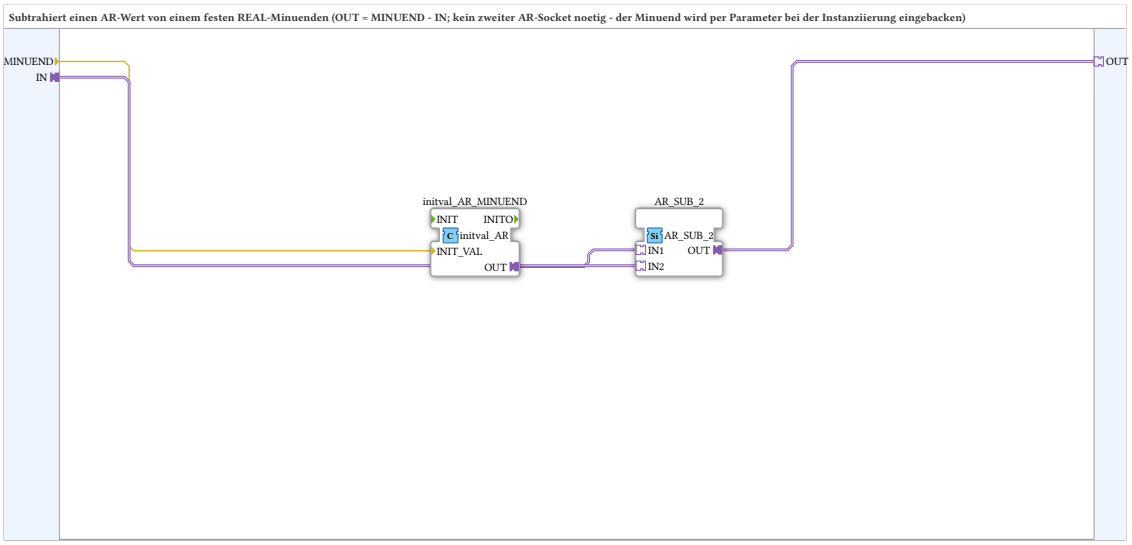
Anwendungsszenarien

- Vertikales Verschieben eines VT-Objekts (z. B. Marker) proportional zu einem Messwert, X-Position fest links

Fazit

`Q_ChildPosition_AR_Vertikal` bildet einen AR-Wert über `AR_TO_AI` auf die Y-Position eines Kind-Objekts ab und hält die X-Position fest bei 0.

R_AR_SUB



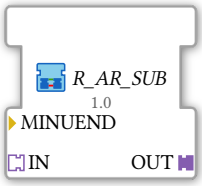
• – ◦ • – ◦ • – ◦ •

Einleitung

`R_AR_SUB` subtrahiert einen `AR`-Wert von einem festen `REAL`-Minuenden: `OUT = MINUEND - IN`. Ein zweiter `AR`-Socket ist nicht nötig, da der Minuend per Parameter bei der Instanziierung eingebacken wird.

Fester Minuend, ueber Parameter bei der Instanziierung gesetzt – *REAL*

Zu subtrahierender Subtrahend – *AR*



AR – MINUEND - IN

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: R_AR_SUB

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **initval_AR_MINUEND**: `adapter::types::unidirectional::AR::initval::initval_AR` – wandelt den Parameter `MINUEND` in ein `AR`-Signal
  - **AR_SUB_2**: `adapter::iec61131::arithmetic::AR_SUB_2` – bildet die Differenz `IN1 - IN2`

Schnittstellenstruktur

Ereignis-Eingänge / Ereignis-Ausgänge

Keine – Ereignisse laufen über die Adapter.

Daten-Eingänge

- **MINUEND** (`REAL`, Initialwert `0.0`): Fester Minuend, über Parameter bei der Instanziierung gesetzt

Daten-Ausgänge

Keine.

Adapter

- **IN** (Socket): `adapter::types::unidirectional::AR` – zu subtrahierender Subtrahend
- **OUT** (Plug): `adapter::types::unidirectional::AR` – `MINUEND - IN`

Programmablauf und Verbindungen

- `MINUEND` ⟶ `initval_AR_MINUEND.INIT_VAL`
- `initval_AR_MINUEND.OUT` ⟶ `AR_SUB_2.IN1`

- `IN` ⟶ `AR_SUB_2.IN2`
- `AR_SUB_2.OUT` ⟶ `OUT`

Technische Besonderheiten

- Der Minuend steht fest als Parameter; es wird kein zweiter `AR`-Eingang benötigt
- Version 1.0 (2026-09-27)

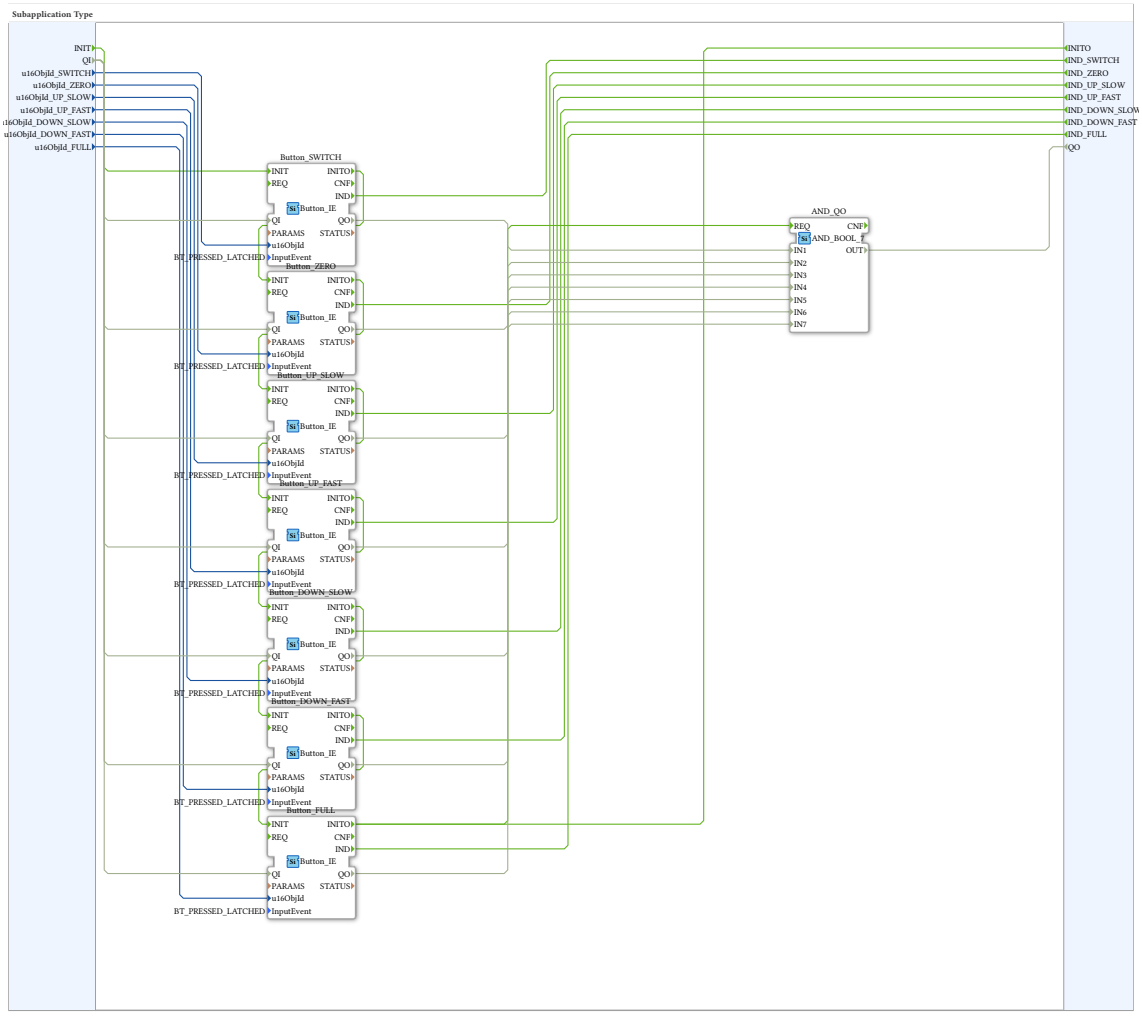
Anwendungsszenarien

- Invertierung relativ zu einer festen Größe, z. B. `100 - Wert`

Zusammenfassung

`R_AR_SUB` liefert `MINUEND - IN` für einen `AR`-Wert mit einem per Parameter festgelegten Minuenden.

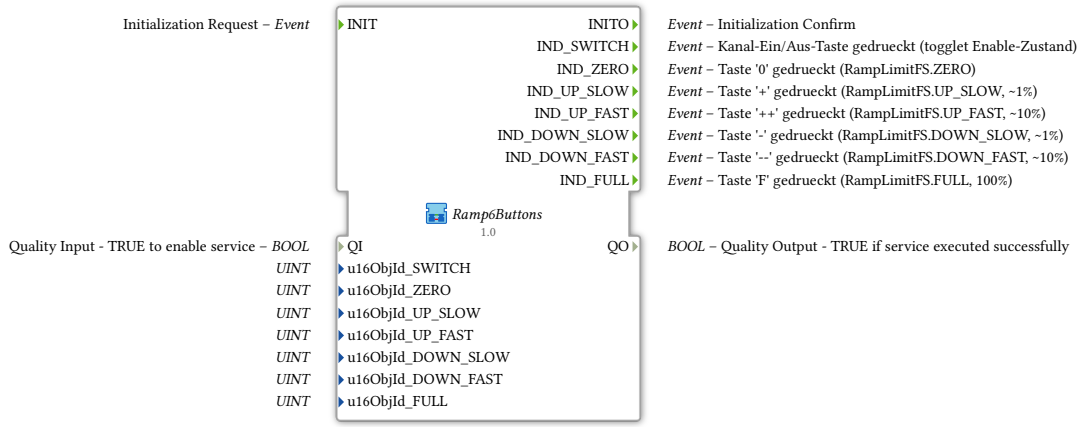
Ramp6Buttons



• – ○ • – ○ • – ○ •

Einleitung

`Ramp6Buttons` kapselt die **7 VT-Taster eines PWM-Kanals** — die 6 Ramp-Tasten (`0 -- - + ++ F`) plus den Kanal-Ein/Aus-Schalter — als eigene, wiederverwendbare SubApp. Sie wurde aus `RampLimitFS_TO_logiBUS_QDA_PWM_OPC` extrahiert, um dessen Netzwerk zu entlasten und die Tasten-Logik unabhängig wiederverwendbar zu machen.



Schnittstelle

- **Ereignis-Eingang**: `INIT` (`EInit`, Initialization Request)
- **Ereignis-Ausgänge**: `INITO` (`EInit`, Initialization Confirm), `IND_SWITCH`, `IND_ZERO`, `IND_UP_SLOW`, `IND_UP_FAST`, `IND_DOWN_SLOW`, `IND_DOWN_FAST`, `IND_FULL`
- **Daten-Eingänge**: `QI` (`BOOL`, Vorgabe `TRUE`, Quality Input) sowie `u16ObjId_SWITCH`, `u16ObjId_ZERO`, `u16ObjId_UP_SLOW`, `u16ObjId_UP_FAST`, `u16ObjId_DOWN_SLOW`, `u16ObjId_DOWN_FAST`, `u16ObjId_FULL`
- **Daten-Ausgang**: `QO` (`BOOL`, Quality Output)

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Ramp6Buttons

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **Button_SWITCH**, **Button_ZERO**, **Button_UP_SLOW**, **Button_UP_FAST**, **Button_DOWN_SLOW**, **Button_DOWN_FAST**, **Button_FULL**: je `isobus::UT::io::Button::Button_IE`
    - Parameter: `InputEvent=BT_PRESSED_LATCHED` (rastendes Drücken/Loslassen-Ereignis, kein Adapter-Bridging über `AX_RF_TRIG` nötig)
    - Dateneingänge: `u16ObjId` (Objekt-ID der jeweiligen VT-Taste), `QI` (vom SubApp-Eingang `QI`)
    - Ereignisausgang: `IND` (Taste gedrückt)
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_7` – verknüpft die `QO`-Ausgänge aller sieben `Button_*` per UND zum SubApp-Ausgang `QO`
- **Funktionsweise**: Sieben gleichartige `Button_IE`-Instanzen, je an eine eigene VT-Objekt-ID gebunden, reichen ihr `IND`-Ereignis unverändert als eigenes SubApp-Ausgangsereignis nach außen durch — reine 1:1-Durchreichung ohne zusätzliche Logik. Zusätzlich wird die Initialisierung durch alle sieben Instanzen gekettet und der Status `QO` aggregiert.

Programmablauf und Verbindungen

Für jeden der 7 Taster gilt dasselbe Muster (`u16ObjId_<NAME>` ⟶ `Button_<NAME>.u16ObjId`, `Button_<NAME>.IND` ⟶ `IND_<NAME>`):

- `u16ObjId_SWITCH` ⟶ `Button_SWITCH.u16ObjId`; `Button_SWITCH.IND` ⟶ `IND_SWITCH` (Kanal-Enable togglen)
- `u16ObjId_ZERO` ⟶ `Button_ZERO.u16ObjId`; `Button_ZERO.IND` ⟶ `IND_ZERO` (`RampLimitFS.ZERO`)
- `u16ObjId_UP_SLOW` ⟶ `Button_UP_SLOW.u16ObjId`; `Button_UP_SLOW.IND` ⟶ `IND_UP_SLOW` (`RampLimitFS.UP_SLOW`, 1 %)
- `u16ObjId_UP_FAST` ⟶ `Button_UP_FAST.u16ObjId`; `Button_UP_FAST.IND` ⟶ `IND_UP_FAST` (`RampLimitFS.UP_FAST`, 10 %)
- `u16ObjId_DOWN_SLOW` ⟶ `Button_DOWN_SLOW.u16ObjId`; `Button_DOWN_SLOW.IND` ⟶ `IND_DOWN_SLOW` (`RampLimitFS.DOWN_SLOW`, 1 %)
- `u16ObjId_DOWN_FAST` ⟶ `Button_DOWN_FAST.u16ObjId`; `Button_DOWN_FAST.IND` ⟶ `IND_DOWN_FAST` (`RampLimitFS.DOWN_FAST`, 10 %)
- `u16ObjId_FULL` ⟶ `Button_FULL.u16ObjId`; `Button_FULL.IND` ⟶ `IND_FULL` (`RampLimitFS.FULL`, 100 %)

Zusätzlich gilt:

- Initialisierungskette: `INIT` ⟶ `Button_SWITCH.INIT`, `Button_SWITCH.INITO` ⟶ `Button_ZERO.INIT`, `Button_ZERO.INITO` ⟶ `Button_UP_SLOW.INIT`, `Button_UP_SLOW.INITO` ⟶ `Button_UP_FAST.INIT`, `Button_UP_FAST.INITO` ⟶ `Button_DOWN_SLOW.INIT`, `Button_DOWN_SLOW.INITO` ⟶ `Button_DOWN_FAST.INIT`, `Button_DOWN_FAST.INITO` ⟶ `Button_FULL.INIT`, `Button_FULL.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`
- `QI` ⟶ `QI` aller sieben `Button_*`-Instanzen
- `Button_SWITCH.QO`, `Button_ZERO.QO`, `Button_UP_SLOW.QO`, `Button_UP_FAST.QO`, `Button_DOWN_SLOW.QO`, `Button_DOWN_FAST.QO`, `Button_FULL.QO` ⟶ `AND_QO.IN1..IN7`; `AND_QO.OUT` ⟶ `QO`

Der aufrufende Baustein verdrahtet `IND_ZERO`/`IND_UP_SLOW`/`IND_UP_FAST`/`IND_DOWN_SLOW`/`IND_DOWN_FAST`/`IND_FULL` direkt auf die gleichnamigen Ereigniseingänge von `RampLimitFS` und `IND_SWITCH` auf den Kanal-Enable-Flipflop.

Technische Besonderheiten

- **`BT_PRESSED_LATCHED` statt `Button_IXA+AX_RF_TRIG`**: einfacheres Muster ohne Adapter-Bridging, da `Button_IE` das Drücken-Ereignis direkt als `IND` ausgibt.
- **Event-Reihenfolge an `RampLimitFS` angeglichen**: Die Ausgangsevents (`SWITCH, ZERO, UP_SLOW, UP_FAST, DOWN_SLOW, DOWN_FAST, FULL`) folgen bewusst der Reihenfolge, in der `RampLimitFS` seine eigenen Ereigniseingänge deklariert, um die Zuordnung beim Verdrahten zu erleichtern.

Anwendungsszenarien

- Jede Übung mit einem Rampen-Sollwert (`RampLimitFS` oder vergleichbar), der über 6 Tasten plus einen Freigabe-Schalter bedient werden soll.

Zusammenfassung

`Ramp6Buttons` bündelt sieben gleichartige `Button_IE`-Instanzen zu einer einzigen wiederverwendbaren SubApp und hält damit die Tasten-Verdrahtung aus dem eigentlichen PWM-Kanal-Baustein heraus.

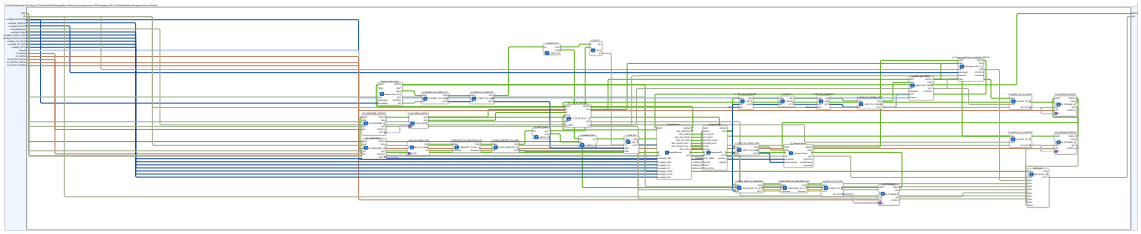
🛠️ Zugehörige Übungen

- RampLimitFS_TO_logiBUS_QDA_PWM_OPC
- InputOutputTesterButton_PWM_OPC_UA

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

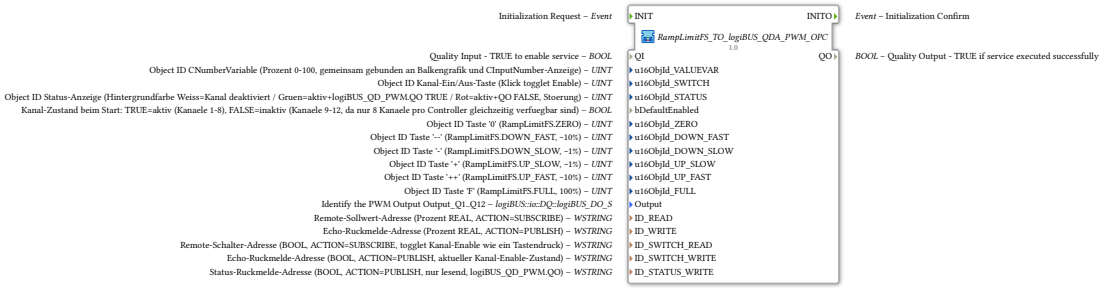
RampLimitFS_TO_logiBUS_QDA_PWM_OPC



• – ○ • – ○ • – ○ •

Einleitung

`RampLimitFS_TO_logiBUS_QDA_PWM_OPC` ist der wiederverwendbare Baustein für **einen einzelnen PWM-Ausgangskanal (0–100 % Duty)** mit VT-Zahlenfeld, Balkengrafik, 6 Ramp-Tasten, Kanal-Ein/Aus-Schalter, 3-Farben-Statusanzeige und bidirektionaler OPC-UA-Anbindung. Er wird 12× parametrisiert in `InputOutputTesterButton_PWM_OPC_UA` instanziiert und ist das PWM-Pendant zum einfacheren, rein digitalen `RampLimitFS_TO_logiBUS_QDA_OPC`.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: RampLimitFS_TO_logiBUS_QDA_PWM_OPC

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **RampLimitFS**: `eclipse4diac::signalprocessing::RampLimitFS`
    - Parameter: `VAL_ZERO=DINT#0`, `SLOW=DINT#643` (1 %), `FAST=DINT#6426` (10 %), `VAL_FULL=DINT#64255`
    - Dateneingang: `PV` (Sollwert-Mux-Ausgang), Ereigniseingänge: `ZERO`/`UP_SLOW`/`UP_FAST`/`DOWN_SLOW`/`DOWN_FAST`/`FULL`/`LOAD`
    - Datenausgang: `OUT` (0–64255)
  - **Ramp6Buttons** (SubApp, `MyLib::sys`): kapselt die 7 VT-Taster (6 Ramp-Tasten + Kanal-Schalter), siehe eigene Dokumentation
  - **F_PWM_PERCENT_TO_RAW** / **F_PWM_RAW_TO_PERCENT** (SubApp, `MyLib::sys`): Umrechnung Anteil 0.0-1.0 ↔ Fieldbus-Rohwert, siehe F_PWM_PERCENT_TO_RAW / F_PWM_RAW_TO_PERCENT
  - **F_PERCENT_TO_FRACTION_SUB** / **F_FRACTION_TO_PERCENT_PUB**: `logiBUS::signalprocessing::fieldbus::F_PERCENT_TO_FRACTION/F_FRACTION_TO_PERCENT` — Prozent 0-100 ↔ Anteil 0.0-1.0
  - **E_RS_PV** (`iec61499::events::E_RS`) + **F_SEL_PV** (`iec61131::selection::F_SEL`): Zwei-Quellen-Merge (VT-Zahlenfeld und Web-Sollwert) auf den einen Dateneingang `RampLimitFS.PV`, da 4diac IDE keine Mehrfachverbindung auf einen Dateneingang zulässt
  - **E_T_FF_SR_SYM_INIT** (`E_T_FF_SR_SWITCH`): Kanal-Enable-Zustand, gesetzt/rückgesetzt durch VT-Taster (`CLK`, echtes Toggle) und Web-Schreibzugriff (`S`/`R`, über `AX_RF_TRIG` erkannter Flankenwechsel, kein Toggle)
  - **F_MUL_TO_PWM13BIT/F_DIV_TO_PWM13BIT** (`iec61131::arithmetic::F_MUL/F_DIV`): rechnen `RampLimitFS.OUT` (0-64255) über `×8191 ÷ 64255` auf den 13-Bit-Rohwert (0-8191) für `logiBUS_QD_PWM.OUT` um
  - **logiBUS_QD_PWM**: physischer PWM-Ausgang (`Output`-Parameter identifiziert `Output_Q1..Q12`)
  - **F_SEL_OK_FAULT/F_SEL_STATUS** (`iec61131::selection::F_SEL`) + **Q_BackgroundColour**: 3-Farben-Statuslogik (Weiß=deaktiviert, Grün=aktiv+`QO`=TRUE, Rot=aktiv+`QO`=FALSE)
  - **AR_SUBSCRIBE_1/AR_PUBLISH_1**, **AX_SUBSCRIBE_1/AX_PUBLISH_1** (×2): OPC-UA-Adapter für Sollwert (REAL), Schalter (BOOL) und Status (BOOL)
  - **AND_QO** (`iec61131::booleanOperators::AND_BOOL_10`): verknüpft zehn `QO`-Quellen zum SubApp-Ausgang `QO`
- **Funktionsweise**: Zwei unabhängige Sollwertquellen — das VT-Zahlenfeld/die Ramp-Tasten und ein per OPC-UA-Subscribe empfangener Web-Prozentwert — werden über `E_RS+F_SEL` auf denselben `RampLimitFS`-Rampenbaustein gemuxt (die zuletzt aktive Quelle gewinnt). Der Rampenausgang (0-64255) geht dreifach weiter: als 13-Bit-Rohwert an den physischen `logiBUS_QD_PWM`-Ausgang, als Prozentwert zurück an das VT-Zahlenfeld/die Balkengrafik und als Prozentwert per OPC-UA-Publish an den Web-Client. Parallel togglet ein VT-Taster oder ein Web-Schreibzugriff den Kanal-Enable-Zustand (`E_T_FF_SR_SWITCH`), der über `logiBUS_QD_PWM.QI` den physischen Ausgang scharf-/unscharf schaltet und in die 3-Farben-Statusanzeige einfließt.

Programmablauf und Verbindungen

1. **VT-Sollwertpfad**: `NumericValue_Duty.IND` (Zahlenfeld geändert) ⟶ `F_DWORD_TO_UDINT_VT` ⟶ `F_UDINT_TO_DINT_VT` ⟶ `E_SPLIT_VT` ⟶ einerseits `E_RS_PV.R` (Reset: „Web hat Vorrang, bis VT wieder aktiv wird“ — siehe unten), andererseits `E_MERGE_SEL` ⟶ `F_SEL_PV.REQ`.

2. **Web-Sollwertpfad**: AR_SUBSCRIBE_1 (OPC-UA-Subscribe, Prozent REAL) ⟶ AR_R_TO_REAL_SUB ⟶ F_PERCENT_TO_FRACTION_SUB (Prozent ⟶ Anteil) ⟶ F_PWM_PERCENT_TO_RAW (Anteil ⟶ Fieldbus-Rohwert) ⟶ E_SPLIT_WEB ⟶ einerseits E_RS_PV.S (Set: Web-Quelle aktiv), andererseits E_MERGE_SEL ⟶ F_SEL_PV.REQ.
3. **Mux und Rampe**: E_RS_PV.Q steuert F_SEL_PV.G (0=VT-Wert IN0, 1=Web-Wert IN1); F_SEL_PV.OUT lädt RampLimitFS.PV per LOAD-Event. Ramp6Buttons liefert zusätzlich die 6 Ramp-Ereignisse direkt an RampLimitFS (ZERO/UP_SLOW/UP_FAST/DOWN_SLOW/DOWN_FAST/FULL).
4. **Ausgabe (physisch)**: RampLimitFS.OUT (0-64255) ⟶ F_MUL_TO_PWM13BIT (×8191) ⟶ F_DIV_TO_PWM13BIT (÷64255) ⟶ F_DINT_TO_DWORD_OUT ⟶ logiBUS_QD_PWM.OUT.
5. **Ausgabe (VT-Anzeige)**: RampLimitFS.OUT ⟶ F_DINT_TO_UDINT_DISP ⟶ Q_NumericValue.REQ (aktualisiert Zahlenfeld + gemeinsam gebundene Balkengrafik).
6. **Ausgabe (OPC-UA-Publish)**: RampLimitFS.OUT ⟶ F_PWM_RAW_TO_PERCENT (Rohwert ⟶ Anteil) ⟶ F_FRACTION_TO_PERCENT_PUB (Anteil ⟶ Prozent) ⟶ AR_REAL_TO_R_PUB ⟶ AR_PUBLISH_1.
7. **Kanal-Schalter**: Ramp6Buttons.IND_SWITCH (VT-Taster) ⟶ E_T_FF_SR_SWITCH.CLK (togglet). AX_SUBSCRIBE_SWITCH (Web-Schreibzugriff, BOOL) ⟶ AX_RF_TRIG_SWITCH erkennt echten Flankenwechsel ⟶ ER⟶S / EF⟶R (setzt statt togglet, damit zwei gleiche Web-Schreibzugriffe nicht versehentlich invertieren). bDefaultEnabled speist E_T_FF_SR_SWITCH.Q_INIT für den Startzustand.
8. **Statuskette**: E_T_FF_SR_SWITCH.Q ⟶ logiBUS_QD_PWM.QI (scharf/unscharf) und ⟶ F_SEL_STATUS.G. logiBUS_QD_PWM.INITO/.QO speisen F_SEL_OK_FAULT (Rot/Grün nach QO), dessen Ergebnis über F_SEL_STATUS (Weiß, falls deaktiviert) an Q_BackgroundColour_STATUS geht. Zusätzlich spiegeln AX_BOOL_TO_X_SWITCH/AX_PUBLISH_SWITCH und AX_BOOL_TO_X_STATUS/AX_PUBLISH_STATUS Enable-Zustand und QO per OPC-UA-Publish an den Web-Client.
9. **Boot-Reihenfolge**: Die SubApp besitzt den Ereigniseingang INIT und den Ereignisausgang INITO (beide neu). INIT ⟶ AR_SUBSCRIBE_1.INIT; die fünf OPC-UA-Adapter werden strikt verkettet initialisiert (AR_SUBSCRIBE_1.INITO → AR_PUBLISH_1.INIT → AX_SUBSCRIBE_SWITCH.INIT → AX_PUBLISH_SWITCH.INIT → AX_PUBLISH_STATUS.INIT); erst danach feuert AX_PUBLISH_STATUS.INITO das E_T_FF_SR_SWITCH.INIT, damit der erste publizierte Enable-Zustand nicht von einem noch nicht bereiten Adapter verworfen wird. Ebenfalls von AX_PUBLISH_STATUS.INITO ausgehend läuft die VT-Initialisierungskette Q_NumericValue.INIT → Q_BackgroundColour_EnableOk_STATUS.INIT → Ramp6Buttons.INIT → NumericValue_Duty.INIT; NumericValue_Duty.INITO ⟶ INITO und ⟶ AND_QO.REQ.
10. **QI/QO-Durchreichung**: QI (BOOL, Vorgabe TRUE) wird an zehn Instanzen verteilt: Q_NumericValue, NumericValue_Duty, AR_SUBSCRIBE_1, AR_PUBLISH_1, E_T_FF_SR_SWITCH, AX_SUBSCRIBE_SWITCH, AX_PUBLISH_SWITCH, AX_PUBLISH_STATUS, Q_BackgroundColour_EnableOk_STATUS und Ramp6Buttons (die internen FBs tragen kein festes QI=TRUE mehr). Deren QO-Ausgänge — Q_NumericValue (IN1), Q_BackgroundColour_EnableOk_STATUS (IN2), Ramp6Buttons (IN3), NumericValue_Duty (IN4), AR_SUBSCRIBE_1 (IN5), AR_PUBLISH_1 (IN6), E_T_FF_SR_SWITCH (IN7), AX_SUBSCRIBE_SWITCH (IN8), AX_PUBLISH_SWITCH (IN9), AX_PUBLISH_STATUS (IN10) — laufen auf AND_QO (AND_BOOL_10), AND_QO.OUT ⟶ QO (BOOL, Quality Output). Dieses QO ist unabhängig vom logiBUS_QD_PWM.QO der Statuskette.

Technische Besonderheiten

- **Wertebereich**: Intern durchgängig SAE-J1939/ISO-11783-Fieldbus-Konvention VALID_SIGNAL_W (0-64255), nicht Prozent — vermeidet Rundungsfehler und ist konsistent mit RampLimitFS.
- **13-Bit-PWM-Rohwert**: logiBUS_QD_PWM.OUT erwartet empirisch bestätigt 0-8191 (nicht 0-64255) — daher die zusätzliche ×8191÷64255-Umrechnungskette.
- **Mehrfachverbindungs-Workaround**: E_RS+F_SEL statt einer (in 4diac IDE nicht erlaubten) doppelten Datenverbindung auf RampLimitFS.PV.
- **Kein PERMIT-Gating**: Der Kanal-Schalter verdrahtet QI direkt als Datenverbindung (Muster aus Uebung_094a), keine E_PERMIT-Bausteine.
- **Set/Reset statt Toggle bei Web-Schreibzugriff**: AX_RF_TRIG + E_T_FF_SR_SYM_INIT statt eines reinen Toggle-Flipflops, damit wiederholte Schreibzugriffe mit demselben Wert nicht zufällig invertieren.

Anwendungsszenarien

- Trainingsbeispiel für analoge (PWM-)Ausgänge mit Fernbedienung über OPC-UA/Web, parallel zur klassischen VT-Bedienung.

- Vorlage für beliebige weitere Mehrkanal-Analogausgang-Übungen mit Sollwert-Ramping und Kanal-Freigabe.

Vergleich mit ähnlichen Bausteinen

Gegenüber dem einfacheren, rein digitalen Pendant `RampLimitFS_TO_logiBUS_QDA_OPC` unterscheidet sich `RampLimitFS_TO_logiBUS_QDA_PWM_OPC` durch den analogen Sollwert (Rampe statt Bit), die zusätzliche Skalierungskette (Prozent ↔ Anteil ↔ Fieldbus-Rohwert ↔ 13-Bit-PWM) und die 3-Farben- statt 2-Farben-Statuslogik.

Zusammenfassung

`RampLimitFS_TO_logiBUS_QDA_PWM_OPC` kapselt einen vollständigen PWM-Kanal — Sollwert-Ramping, Zwei-Quellen-Mux zwischen VT und Web, Skalierung auf den physischen 13-Bit-PWM-Ausgang, Kanal-Enable-Schalter mit robustem Set/Reset-Verhalten und 3-Farben-Statusrückmeldung — in einem einzigen, 12-fach wiederverwendbaren Baustein.

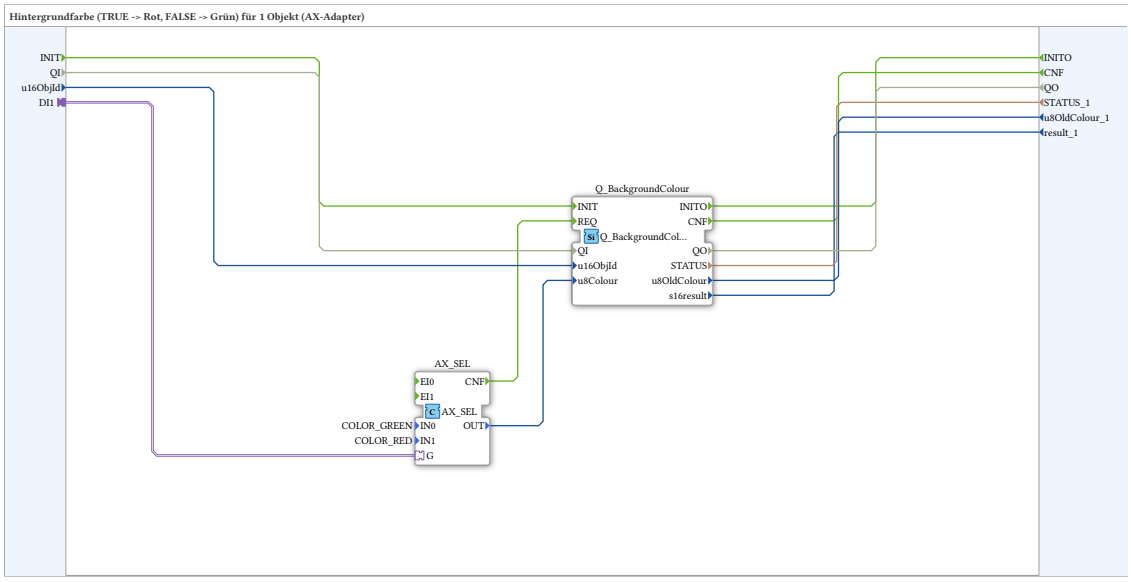
🛠️ Zugehörige Übungen

- InputOutputTesterButton_PWM_OPC_UA

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedGreenBackground1_AX

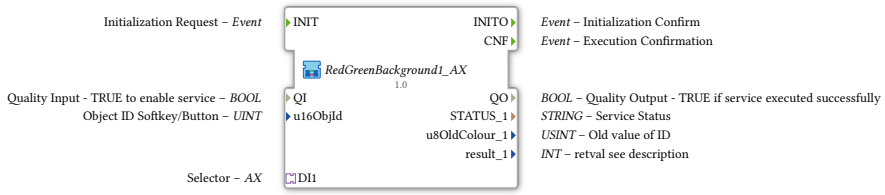


• – ○ • – ○ • – ○ •

Einleitung

`RedGreenBackground1_AX` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Grün**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den Eingang `u16ObjId` übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Schnittstelle: Initialisierung und Qualität

- **INIT** (Ereignis-Eingang, `EInit`) ⟶ `Q_BackgroundColour.INIT`; `Q_BackgroundColour.INITO` ⟶ **INITO** (Ereignis-Ausgang, `EInit`).
- **QI** (Daten-Eingang, `BOOL`, Vorgabe `TRUE`): Quality Input – `TRUE` aktiviert den Dienst; durchgereicht an `Q_BackgroundColour.QI`.
- **QO** (Daten-Ausgang, `BOOL`): Quality Output – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde; kommt von `Q_BackgroundColour.QO`.

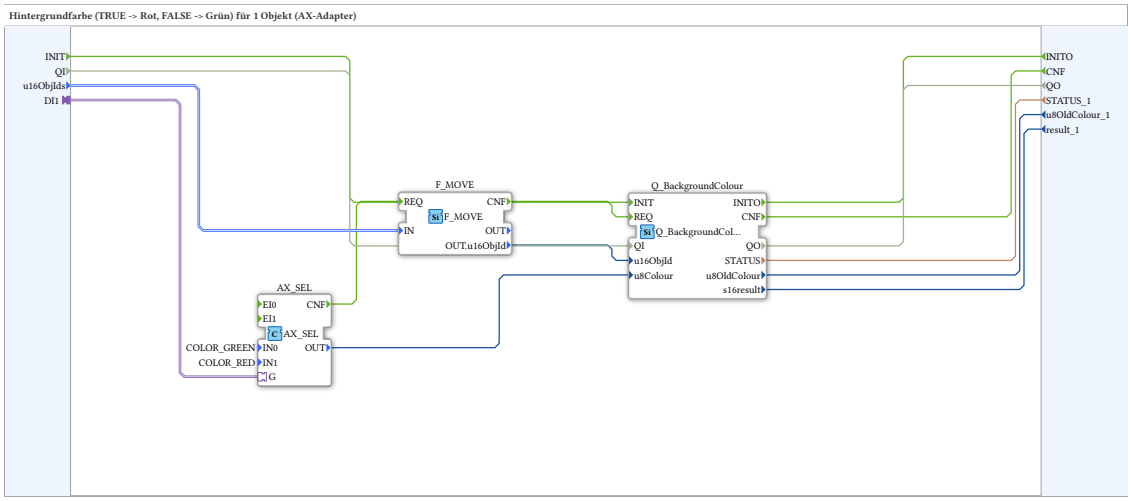
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Grün, 1 Objekt, Adapter-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedGreenBackground1_AXS

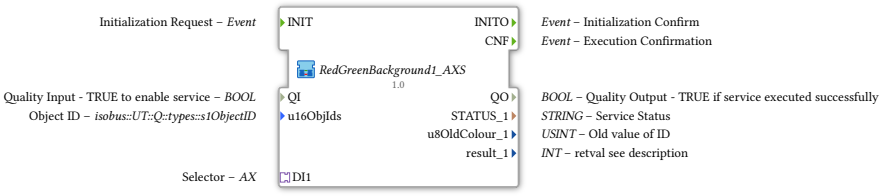


• – ○ • – ○ • – ○ •

Einleitung

`RedGreenBackground1_AXS` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Grün**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Schnittstelle: Initialisierung und Qualität

- **INIT** (Ereignis-Eingang, `EInit`) ⟶ `Q_BackgroundColour.INIT`; `Q_BackgroundColour.INITO` ⟶ **INITO** (Ereignis-Ausgang, `EInit`).
- **QI** (Daten-Eingang, `BOOL`, Vorgabe `TRUE`): Quality Input – `TRUE` aktiviert den Dienst; durchgereicht an `Q_BackgroundColour.QI`.
- **QO** (Daten-Ausgang, `BOOL`): Quality Output – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde; kommt von `Q_BackgroundColour.QO`.

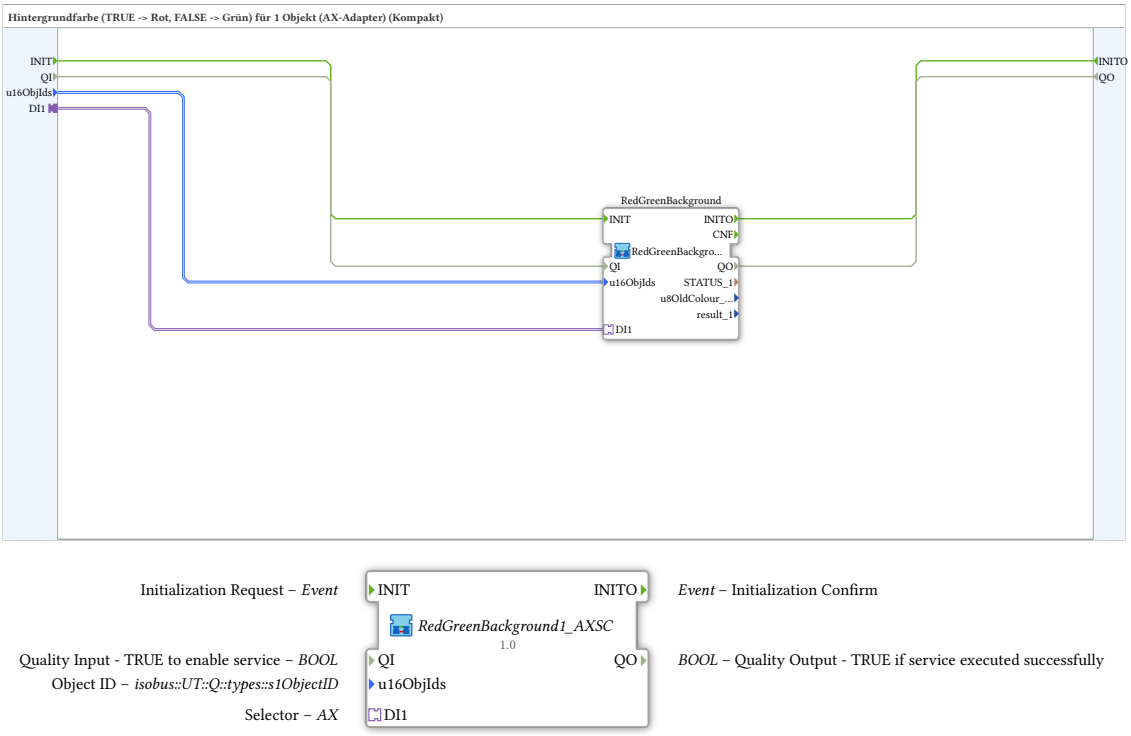
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Grün, 1 Objekt, Adapter-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedGreenBackground1_AXSC



• – ○ • – ○ • – ○ •

Einleitung

`RedGreenBackground1_AXSC` ist ein **Kompakt-Wrapper** um `RedGreenBackground1_AXS` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Zusätzlich werden Initialisierung und Qualität durchgereicht: `INIT` ⟶ `RedGreenBackground.INIT`, `RedGreenBackground.INITO` ⟶ `INITO`, `QI` (Initialwert `TRUE`) ⟶ `RedGreenBackground.QI` und `RedGreenBackground.QO` ⟶ `QO`.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

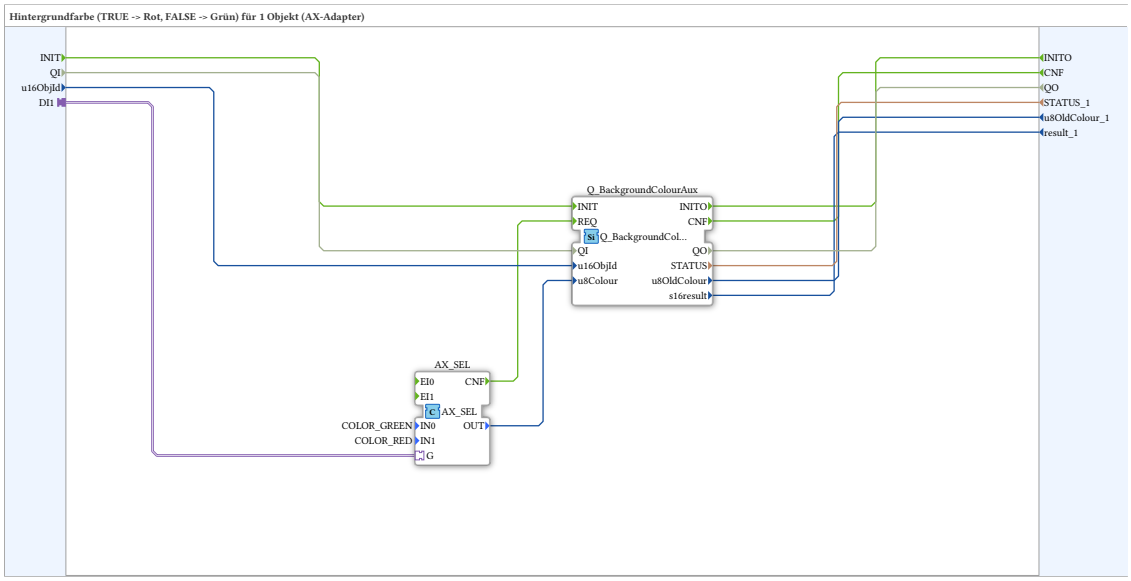
Zusammenfassung

Reine Weiterleitung an `RedGreenBackground1_AXS` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedGreenBackground1_aux_AX

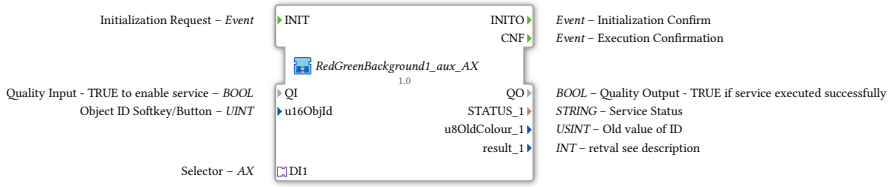


• — ○ • — ○ • — ○ •

Einleitung

`RedGreenBackground1_aux_AX` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Grün**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den Eingang `u16ObjId` übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Schnittstelle: Initialisierung und Qualität

- **INIT** (Ereignis-Eingang, `EInit`) ⟶ `Q_BackgroundColourAux.INIT`; `Q_BackgroundColourAux.INITO` ⟶ **INITO** (Ereignis-Ausgang, `EInit`).
- **QI** (Daten-Eingang, `BOOL`, Vorgabe `TRUE`): Quality Input – `TRUE` aktiviert den Dienst; durchgereicht an `Q_BackgroundColourAux.QI`.
- **QO** (Daten-Ausgang, `BOOL`): Quality Output – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde; kommt von `Q_BackgroundColourAux.QO`.

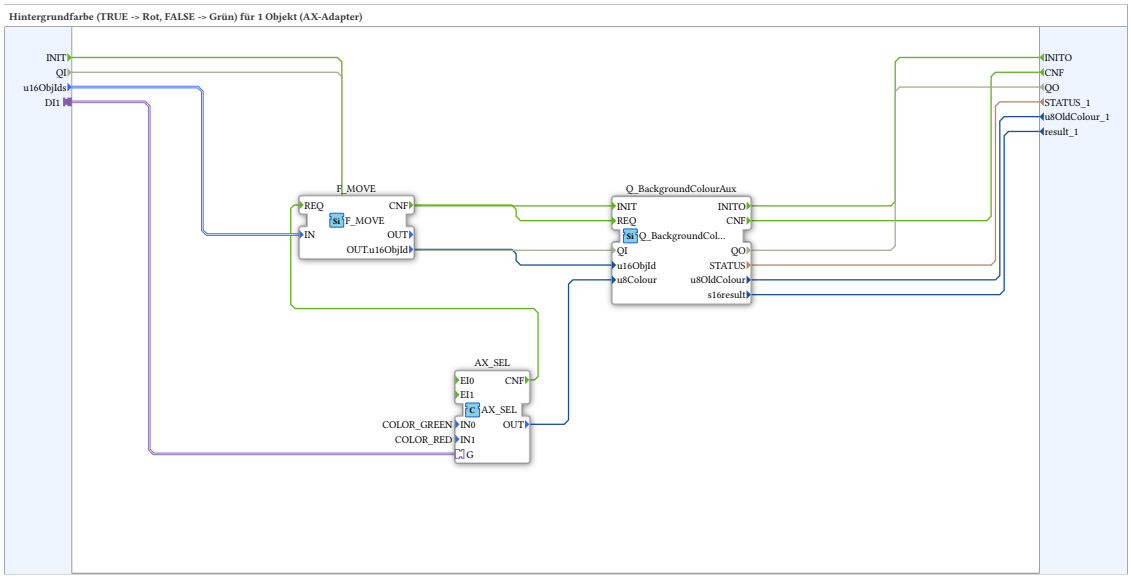
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Grün, 1 Objekt, Adapter-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedGreenBackground1_aux_AXS

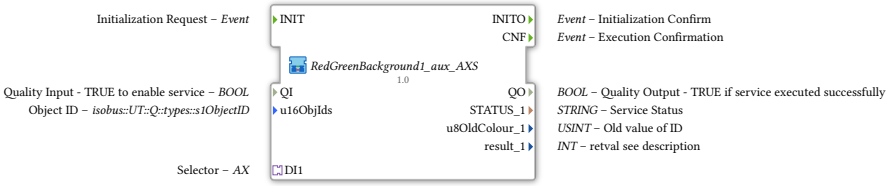


• — ○ • — ○ • — ○ •

Einleitung

`RedGreenBackground1_aux_AXS` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Grün**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Schnittstelle: Initialisierung und Qualität

- **INIT** (Ereignis-Eingang, `EInit`) ⟶ `Q_BackgroundColourAux.INIT`; `Q_BackgroundColourAux.INITO` ⟶ **INITO** (Ereignis-Ausgang, `EInit`).
- **QI** (Daten-Eingang, `BOOL`, Vorgabe `TRUE`): Quality Input – `TRUE` aktiviert den Dienst; durchgereicht an `Q_BackgroundColourAux.QI`.
- **QO** (Daten-Ausgang, `BOOL`): Quality Output – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde; kommt von `Q_BackgroundColourAux.QO`.

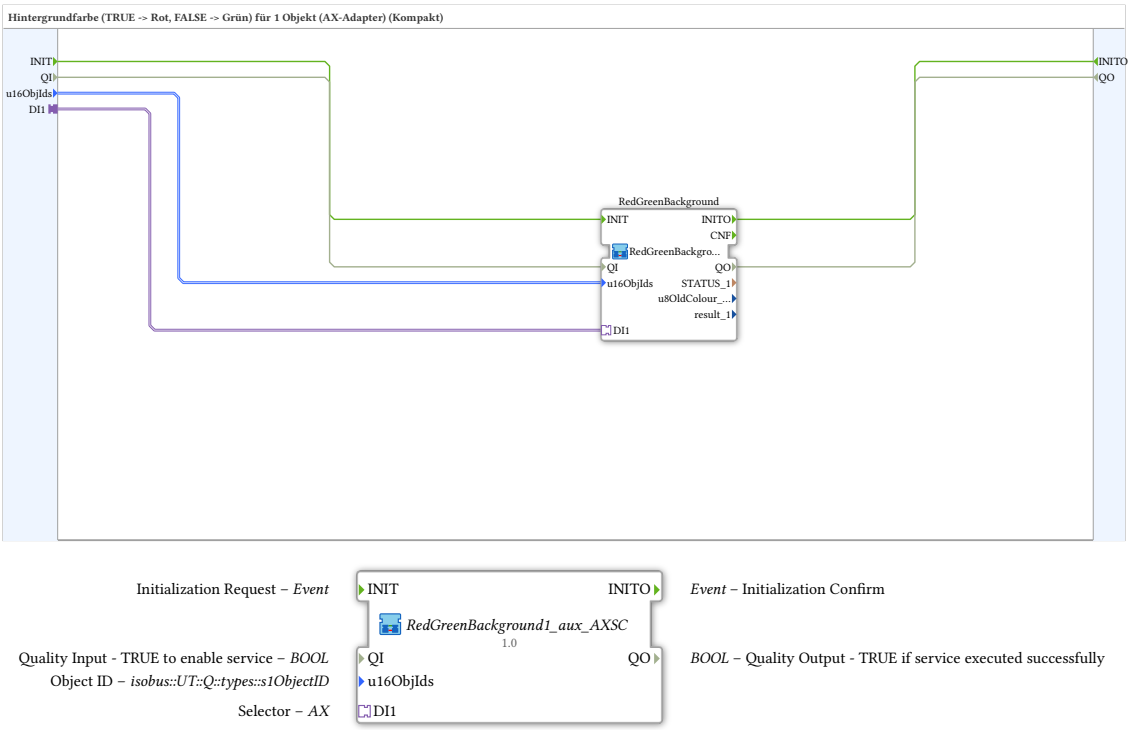
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Grün, 1 Objekt, Adapter-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedGreenBackground1_aux_AXSC



• — ○ • — ○ • — ○ •

Einleitung

`RedGreenBackground1_aux_AXSC` ist ein **Kompakt-Wrapper** um `RedGreenBackground1_aux_AXS` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

Zusätzlich führt der Wrapper die Schnittstelle für Initialisierung und Qualität nach außen: Ereigniseingang `INIT`, Ereignisausgang `INITO`, Dateneingang `QI` (`BOOL`, Initialwert `TRUE`) und Datenausgang `QO` (`BOOL`). Sie sind 1:1 an die innere Instanz `RedGreenBackground` durchgeschleift: `INIT` ⟶ `RedGreenBackground.INIT`, `RedGreenBackground.INITO` ⟶ `INITO`, `QI` ⟶ `RedGreenBackground.QI`, `RedGreenBackground.QO` ⟶ `QO`.

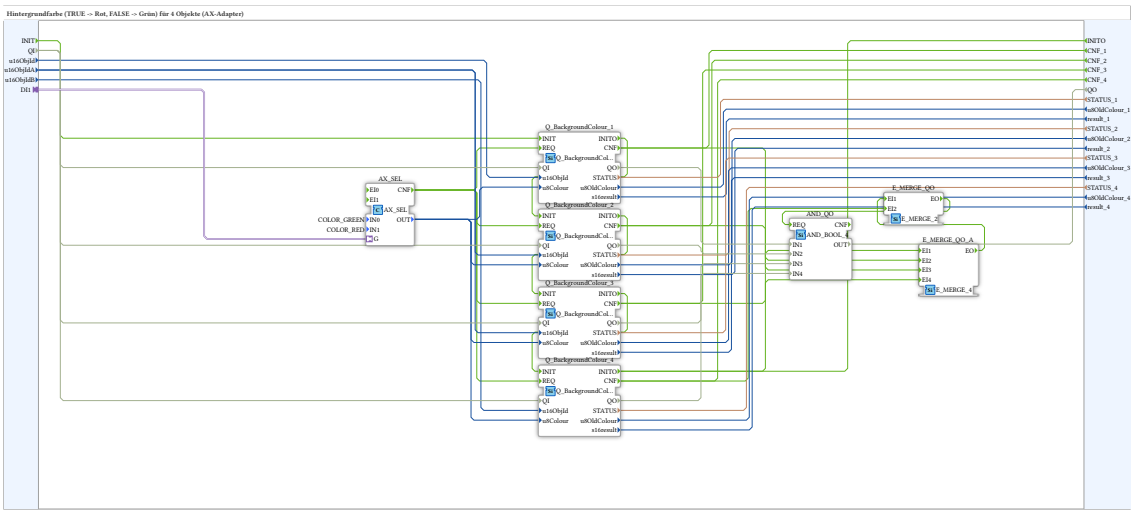
Zusammenfassung

Reine Weiterleitung an `RedGreenBackground1_aux_AXS` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedGreenBackground4_AX



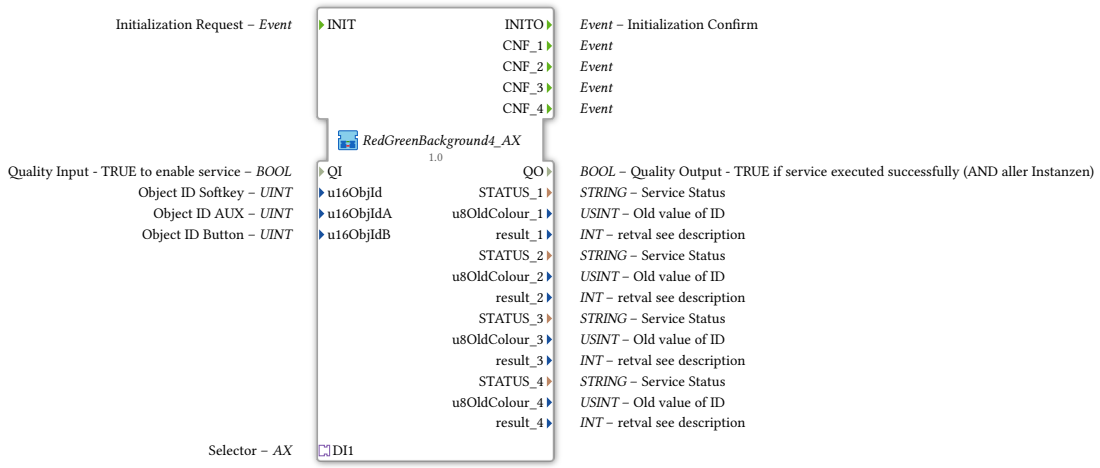
• — ○ • — ○ • — ○ •

Einleitung

`RedGreenBackground4_AX` schaltet die VT-Hintergrundfarbe von 4 Objekten anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Grün**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über die Eingänge `u16ObjId, u16ObjIdA, u16ObjIdB` übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`u16ObjId`	`Q_BackgroundColour` (normales Objekt)
2	`u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
3	`u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)
4	`u16ObjIdB`	`Q_BackgroundColour` (normales Objekt)

Allgemeines Muster (Selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Initialisierung und Qualität

- **Ereignis-Eingang `INIT`** (`EInit`) und **Ereignis-Ausgang `INITO`** (`EInit`): die Initialisierung wird durch die `Q_BackgroundColour`-Instanzen gekettet: `INIT` ⟶ `Q_BackgroundColour_1.INIT`; `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` ⟶ `Q_BackgroundColour_3.INIT`; `Q_BackgroundColour_3.INITO` ⟶ `Q_BackgroundColour_4.INIT`; `Q_BackgroundColour_4.INITO` ⟶ `INITO`.

- **Dateneingang `QI`** (`BOOL`, Vorgabe `TRUE`) wird an `.QI` aller Instanzen verteilt (`Q_BackgroundColour_1`, `Q_BackgroundColour_2`, `Q_BackgroundColour_3`, `Q_BackgroundColour_4`).
- **Datenausgang `QO`** (`BOOL`): UND aller Instanzen über `AND_QO` (`iec61131::booleanOperators::AND_BOOL_4`): `Q_BackgroundColour_1.QO` → `AND_QO.IN1`, `Q_BackgroundColour_2.QO` → `AND_QO.IN2`, `Q_BackgroundColour_3.QO` → `AND_QO.IN3`, `Q_BackgroundColour_4.QO` → `AND_QO.IN4`; `AND_QO.OUT` → `QO`.
- `E_MERGE_QO_A` (`iec61499::events::E_MERGE_4`) vereint `Q_BackgroundColour_4.INITO` und die `CNF`-Ereignisse der Instanzen 1 bis 3; `E_MERGE_QO` (`iec61499::events::E_MERGE_2`) vereint dessen Ausgang `E_MERGE_QO_A.EO` mit `Q_BackgroundColour_4.CNF` und löst damit `AND_QO.REQ` aus.

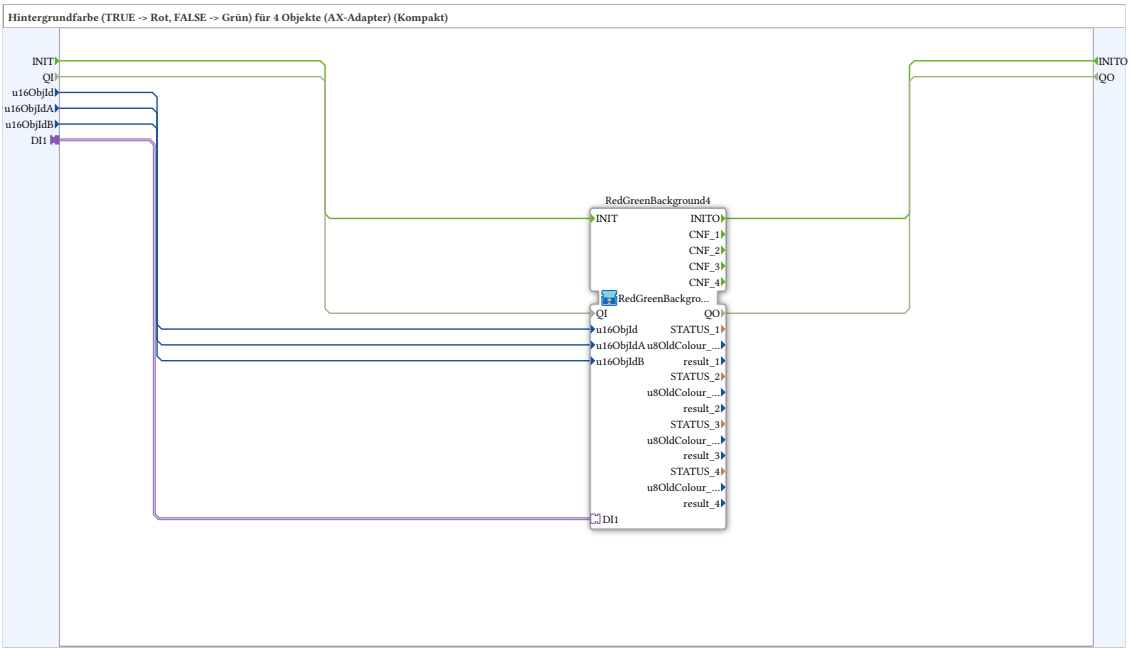
Zusammenfassung

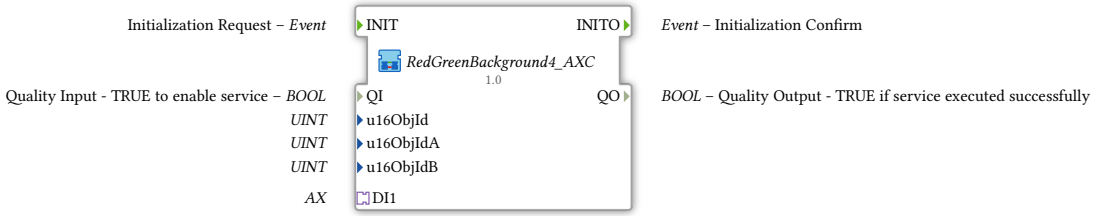
Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Grün, 4 Objekte, Adapter-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedGreenBackground4_AXC





• – ◦ • – ◦ • – ◦ •

Einleitung

`RedGreenBackground4_AXC` ist ein **Kompakt-Wrapper** um `RedGreenBackground4_AX` — reicht `u16ObjId` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Schnittstelle (Quality-Muster): Der Wrapper besitzt zusätzlich den Ereignis-Eingang `INIT` und den Ereignis-Ausgang `INITO` (je `EInit`) sowie den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`) und den Datenausgang `QO` (`BOOL`). Sie sind direkt an die innere Instanz `RedGreenBackground4` durchgeschleift: `INIT` →

`RedGreenBackground4.INIT`, `RedGreenBackground4.INITO` ⟶ `INITO`, `QI` ⟶ `RedGreenBackground4.QI`, `RedGreenBackground4.QO` ⟶ `QO`.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

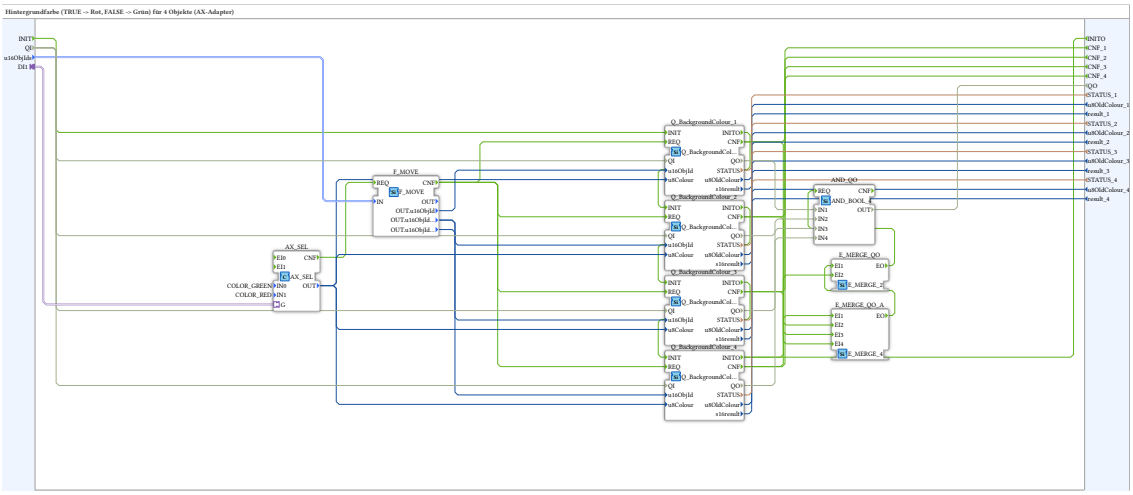
Zusammenfassung

Reine Weiterleitung an `RedGreenBackground4_AX` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedGreenBackground4_AXS



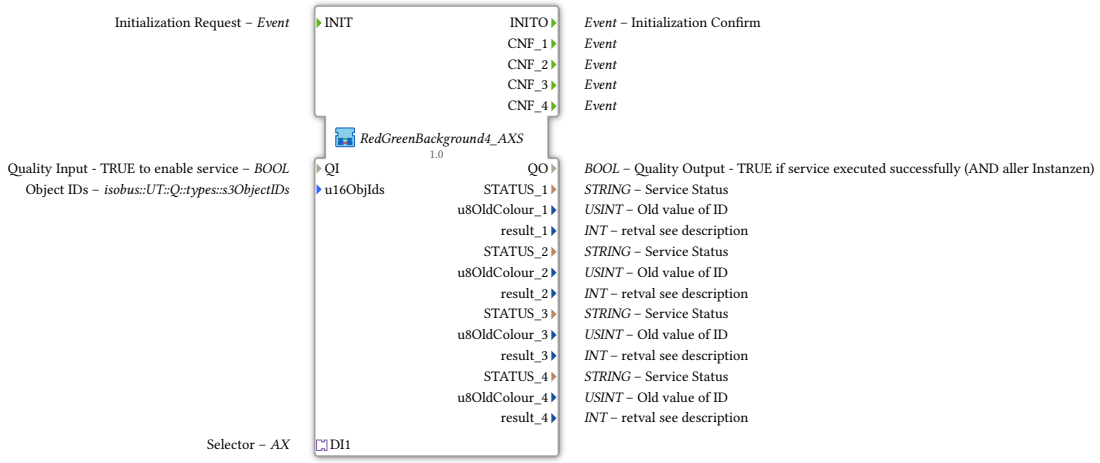
• – ○ • – ○ • – ○ •

Einleitung

`RedGreenBackground4_AXS` schaltet die VT-Hintergrundfarbe von 4 Objekten anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Grün**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`F_MOVE.OUT.u16ObjId`	`Q_BackgroundColour` (normales Objekt)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
3	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)
4	`F_MOVE.OUT.u16ObjIdB`	`Q_BackgroundColour` (normales Objekt)

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Initialisierung und Status (INIT/INITO, QI/QO)

- **Ereignis-Eingang** `INIT` (`EInit`, Initialization Request) und **Ereignis-Ausgang** `INITO` (`EInit`, Initialization Confirm)
- **Daten-Eingang** `QI` (`BOOL`, Vorgabe `TRUE`, Quality Input) wird an `Q_BackgroundColour_1.QI`, `Q_BackgroundColour_2.QI`, `Q_BackgroundColour_3.QI` und `Q_BackgroundColour_4.QI` weitergegeben
- **Daten-Ausgang** `QO` (`BOOL`, Quality Output): UND (`AND_QO`, Typ `iec61131::booleanOperators::AND_BOOL_4`) der `QO`-Ausgänge von `Q_BackgroundColour_1.QO`, `Q_BackgroundColour_2.QO`, `Q_BackgroundColour_3.QO` und `Q_BackgroundColour_4.QO`
- **Initialisierungskette**: `INIT` → `Q_BackgroundColour_1.INIT`, `Q_BackgroundColour_1.INITO` → `Q_BackgroundColour_2.INIT`, `Q_BackgroundColour_2.INITO` → `Q_BackgroundColour_3.INIT`, `Q_BackgroundColour_3.INITO` → `Q_BackgroundColour_4.INIT`, `Q_BackgroundColour_4.INITO` → `INITO`
- `E_MERGE_QO_A` (Typ `iec61499::events::E_MERGE_4`) führt `Q_BackgroundColour_4.INITO`, `Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF` und `Q_BackgroundColour_3.CNF` zusammen und gibt das Ergebnis an `E_MERGE_QO.EI1`
- `E_MERGE_QO` (Typ `iec61499::events::E_MERGE_2`) führt `E_MERGE_QO_A.EO` und `Q_BackgroundColour_4.CNF` zusammen und löst `AND_QO.REQ` aus

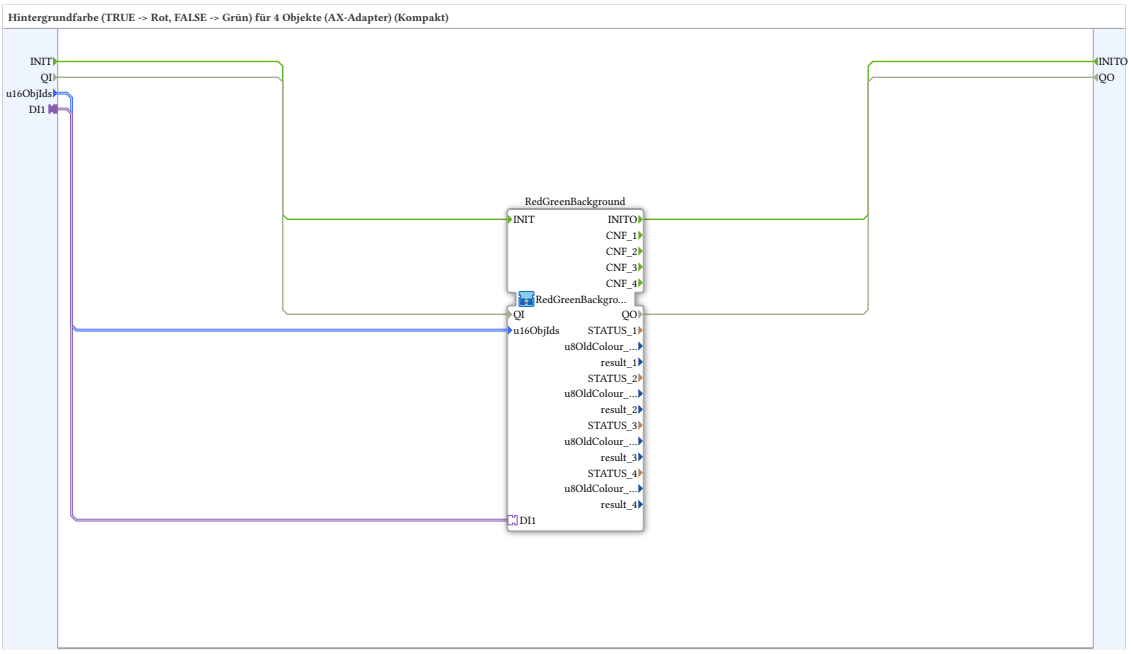
Zusammenfassung

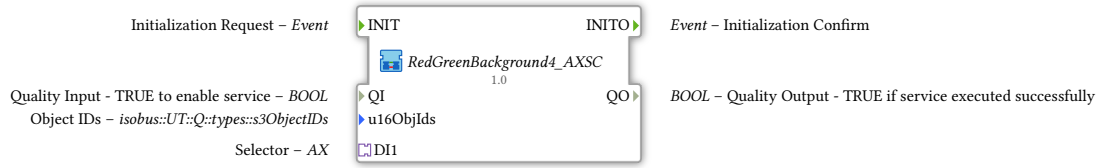
Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Grün, 4 Objekte, Adapter-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedGreenBackground4_AXSC





• — ○ • — ○ • — ○ •

Einleitung

`RedGreenBackground4_AXSC` ist ein **Kompakt-Wrapper** um `RedGreenBackground4_AXS` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen. Zusätzlich werden `INIT` ⟶ `RedGreenBackground.INIT`, `RedGreenBackground.INITO` ⟶ `INITO`, `QI` ⟶ `RedGreenBackground.QI` (Initialwert `TRUE`) und `RedGreenBackground.QO` ⟶ `QO` durchgereicht.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

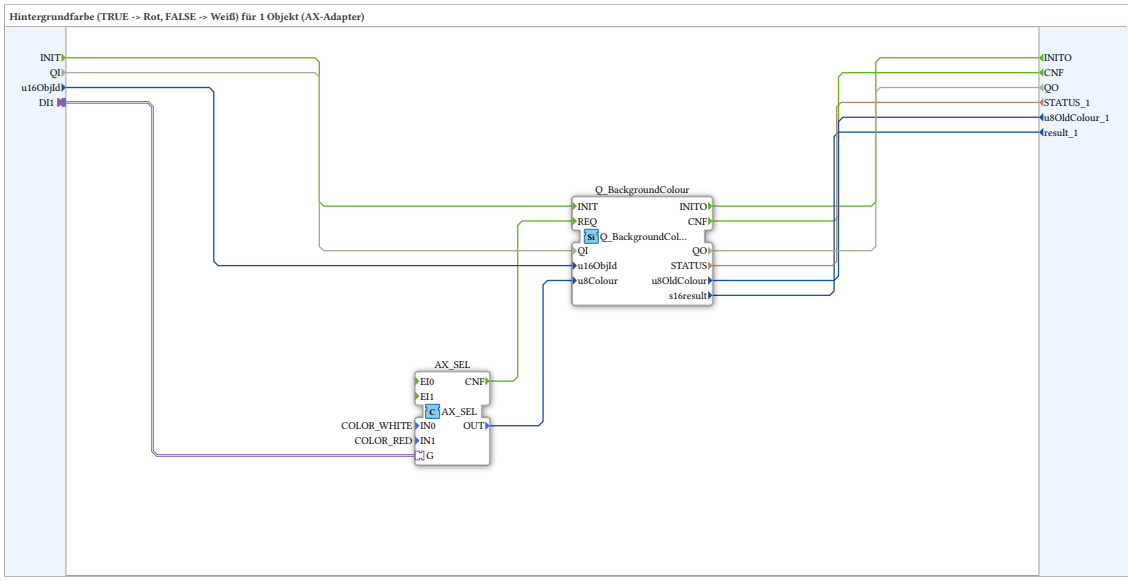
Zusammenfassung

Reine Weiterleitung an `RedGreenBackground4_AXS` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground1_AX

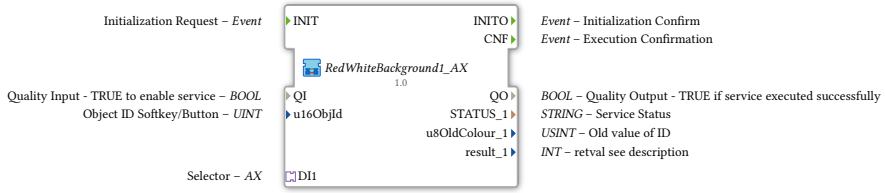


• — ○ • — ○ • — ○ •

Einleitung

`RedWhiteBackground1_AX` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den Eingang `u16ObjId` übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Schnittstelle: Initialisierung und Qualität

- **INIT** (Ereignis-Eingang, `EInit`) ⟶ `Q_BackgroundColour.INIT`; `Q_BackgroundColour.INITO` ⟶ **INITO** (Ereignis-Ausgang, `EInit`).
- **QI** (Daten-Eingang, `BOOL`, Vorgabe `TRUE`): Quality Input – `TRUE` aktiviert den Dienst; durchgereicht an `Q_BackgroundColour.QI`.
- **QO** (Daten-Ausgang, `BOOL`): Quality Output – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde; kommt von `Q_BackgroundColour.QO`.

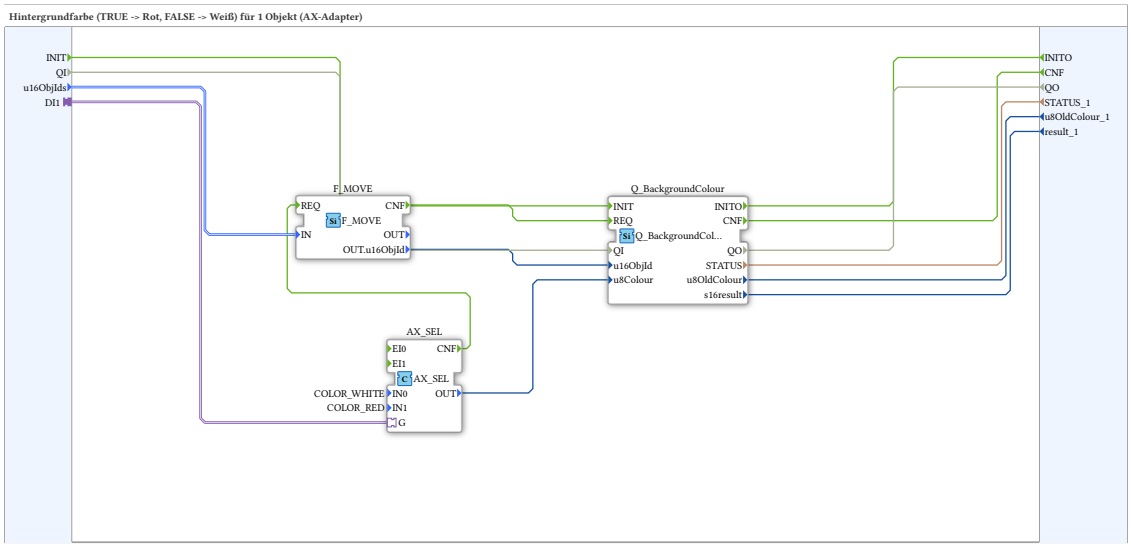
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Weiß, 1 Objekt, Adapter-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground1_AXS

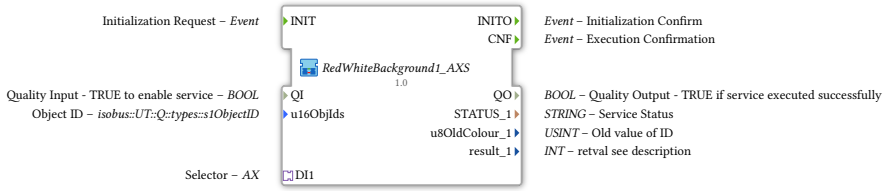


• – ○ • – ○ • – ○ •

Einleitung

`RedWhiteBackground1_AXS` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Schnittstelle: Initialisierung und Qualität

- **INIT** (Ereignis-Eingang, `EInit`) ⟶ `Q_BackgroundColour.INIT`; `Q_BackgroundColour.INITO` ⟶ **INITO** (Ereignis-Ausgang, `EInit`).
- **QI** (Daten-Eingang, `BOOL`, Vorgabe `TRUE`): Quality Input – `TRUE` aktiviert den Dienst; durchgereicht an `Q_BackgroundColour.QI`.
- **QO** (Daten-Ausgang, `BOOL`): Quality Output – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde; kommt von `Q_BackgroundColour.QO`.

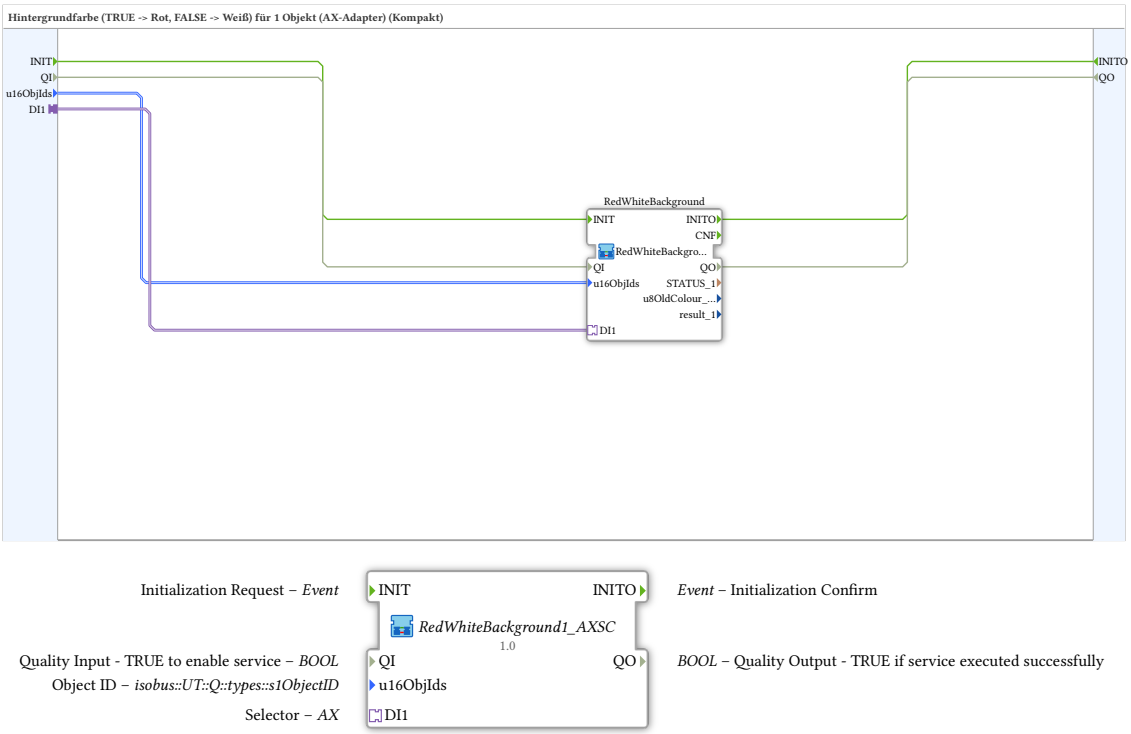
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Weiß, 1 Objekt, Adapter-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground1_AXSC



• – ○ • – ○ • – ○ •

Einleitung

`RedWhiteBackground1_AXSC` ist ein **Kompakt-Wrapper** um `RedWhiteBackground1_AXS` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Zusätzlich werden Initialisierung und Qualität durchgereicht: `INIT` ⟶ `RedWhiteBackground.INIT`, `RedWhiteBackground.INITO` ⟶ `INITO`, `QI` (Initialwert `TRUE`) ⟶ `RedWhiteBackground.QI` und `RedWhiteBackground.QO` ⟶ `QO`.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

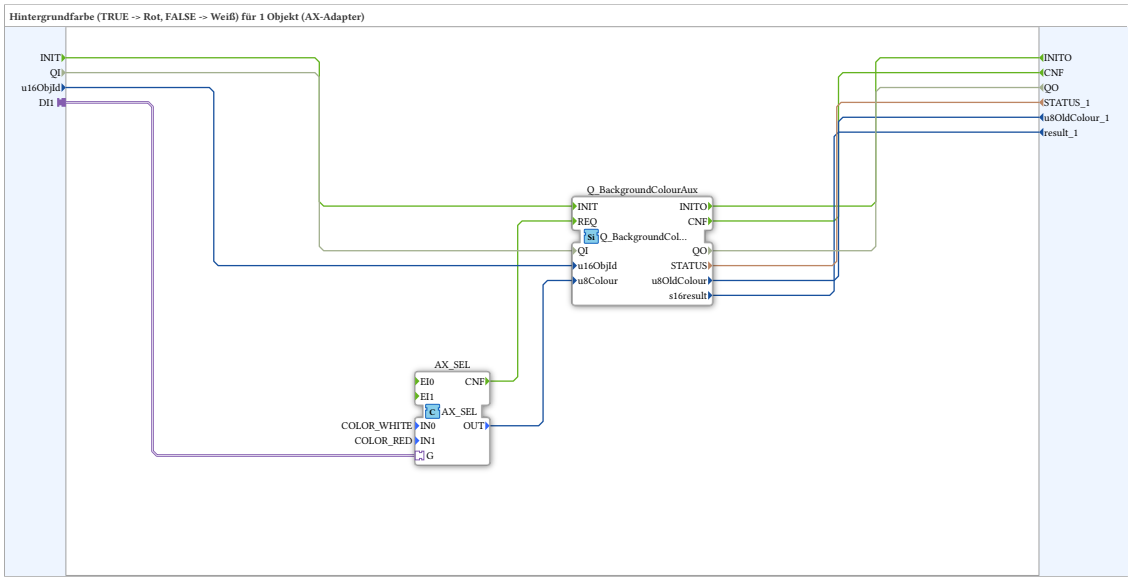
Zusammenfassung

Reine Weiterleitung an `RedWhiteBackground1_AXS` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground1_aux_AX



• — ○ • — ○ • — ○ •

Einleitung

`RedWhiteBackground1_aux_AX` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` → **Rot**, `FALSE` → **Weiß**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den Eingang `u16ObjId` übergeben.

Allgemeines Muster (Selector → `AX_SEL`/`F_SEL` → `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Schnittstelle: Initialisierung und Qualität

- **INIT** (Ereignis-Eingang, `EInit`) → `Q_BackgroundColourAux.INIT`; `Q_BackgroundColourAux.INITO` → **INITO** (Ereignis-Ausgang, `EInit`).
- **QI** (Daten-Eingang, `BOOL`, Vorgabe `TRUE`): Quality Input – `TRUE` aktiviert den Dienst; durchgereicht an `Q_BackgroundColourAux.QI`.
- **QO** (Daten-Ausgang, `BOOL`): Quality Output – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde; kommt von `Q_BackgroundColourAux.QO`.

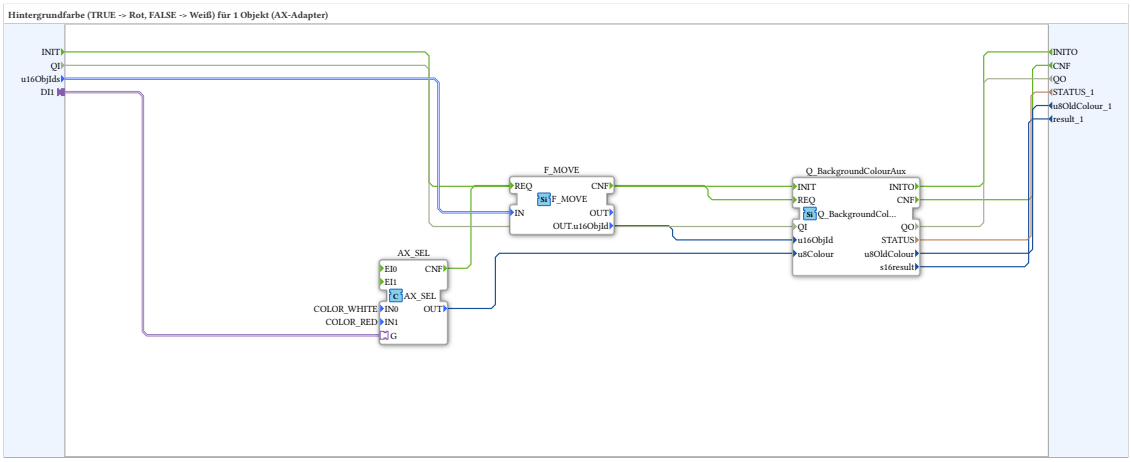
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Weiß, 1 Objekt, Adapter-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground1_aux_AXS

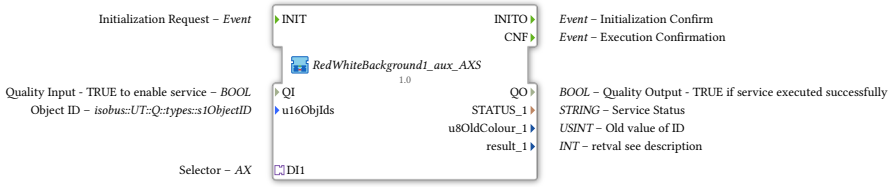


• – ○ • – ○ • – ○ •

Einleitung

`RedWhiteBackground1_aux_AXS` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Schnittstelle: Initialisierung und Qualität

- **INIT** (Ereignis-Eingang, `EInit`) ⟶ `Q_BackgroundColourAux.INIT`; `Q_BackgroundColourAux.INITO` ⟶ **INITO** (Ereignis-Ausgang, `EInit`).
- **QI** (Daten-Eingang, `BOOL`, Vorgabe `TRUE`): Quality Input – `TRUE` aktiviert den Dienst; durchgereicht an `Q_BackgroundColourAux.QI`.
- **QO** (Daten-Ausgang, `BOOL`): Quality Output – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde; kommt von `Q_BackgroundColourAux.QO`.

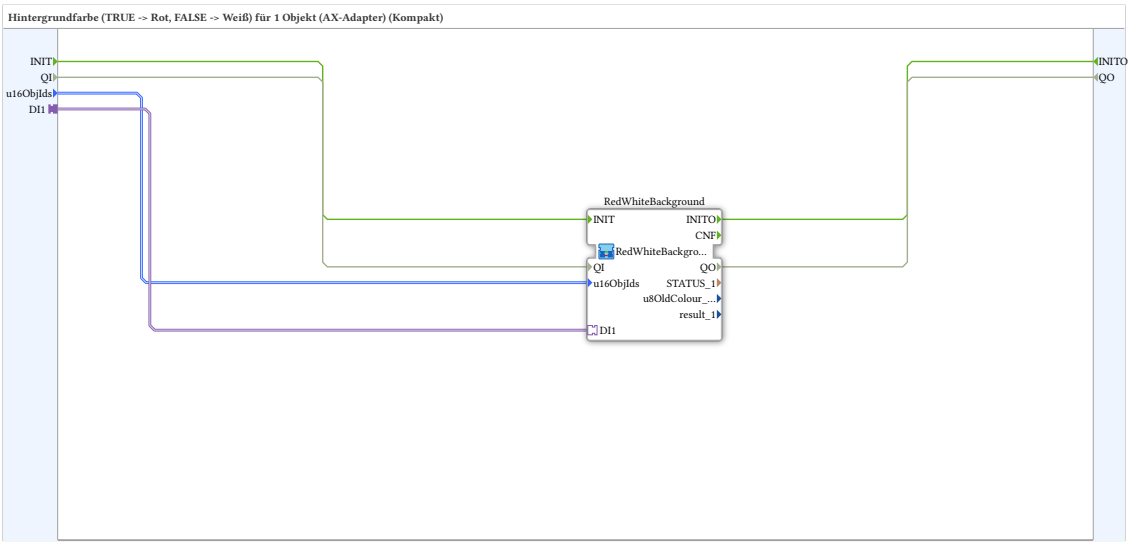
Zusammenfassung

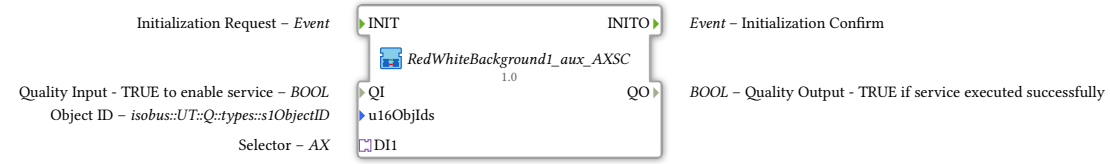
Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Weiß, 1 Objekt, Adapter-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground1_aux_AXSC





• — ○ • — ○ • — ○ •

Einleitung

`RedWhiteBackground1_aux_AXSC` ist ein **Kompakt-Wrapper** um `RedWhiteBackground1_aux_AXS` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

Zusätzlich führt der Wrapper die Schnittstelle für Initialisierung und Qualität nach außen: Ereigniseingang `INIT`, Ereignisausgang `INITO`, Dateneingang `QI` (`BOOL`, Initialwert `TRUE`) und Datenausgang `QO` (`BOOL`). Sie sind 1:1 an die innere Instanz `RedWhiteBackground` durchgeschleift: `INIT` ⟶ `RedWhiteBackground.INIT`, `RedWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `RedWhiteBackground.QI`, `RedWhiteBackground.QO` ⟶ `QO`.

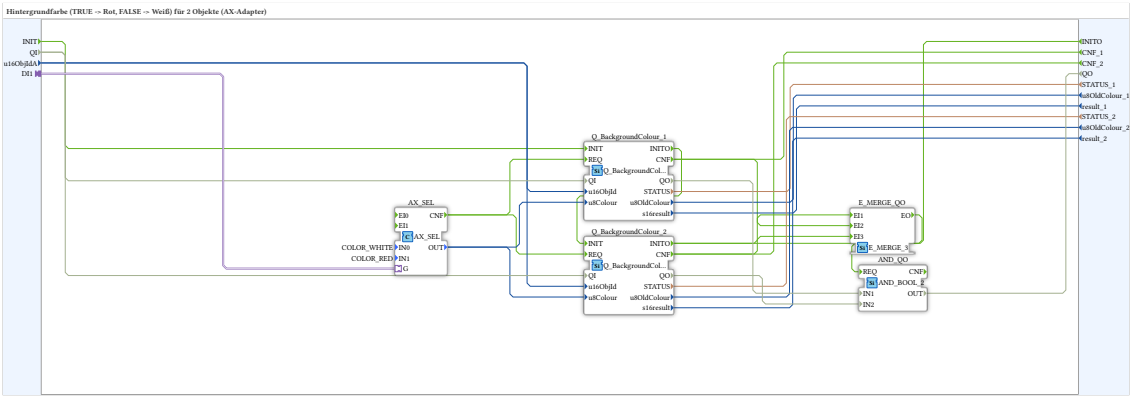
Zusammenfassung

Reine Weiterleitung an `RedWhiteBackground1_aux_AXS` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground2_AX



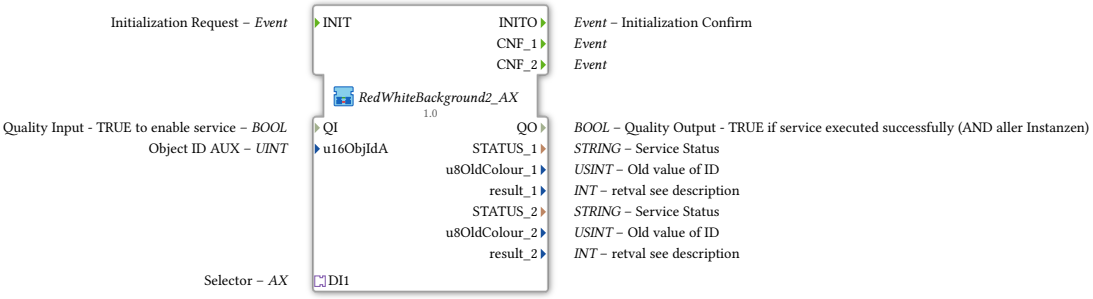
• — ○ • — ○ • — ○ •

Einleitung

`RedWhiteBackground2_AX` schaltet die VT-Hintergrundfarbe von 2 Objekten anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über die Eingänge `u16ObjIdA` übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
2	`u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)

Allgemeines Muster (Selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Initialisierung und Qualität

- **Ereignis-Eingang `INIT`** (`EInit`) und **Ereignis-Ausgang `INITO`** (`EInit`): die Initialisierung wird durch die `Q_BackgroundColour`-Instanzen gekettet: `INIT` ⟶ `Q_BackgroundColour_1.INIT`; `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` ⟶ `INITO`.
- **Dateneingang `QI`** (`BOOL`, Vorgabe `TRUE`) wird an `.QI` aller Instanzen verteilt (`Q_BackgroundColour_1`, `Q_BackgroundColour_2`).
- **Datenausgang `QO`** (`BOOL`): UND aller Instanzen über `AND_QO` (`iec61131::booleanOperators::AND_BOOL_2`): `Q_BackgroundColour_1.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`.
- `E_MERGE_QO` (`iec61499::events::E_MERGE_3`) vereint `Q_BackgroundColour_2.INITO` und die `CNF`-Ereignisse beider Instanzen (`Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF`) und löst damit `AND_QO.REQ` aus.

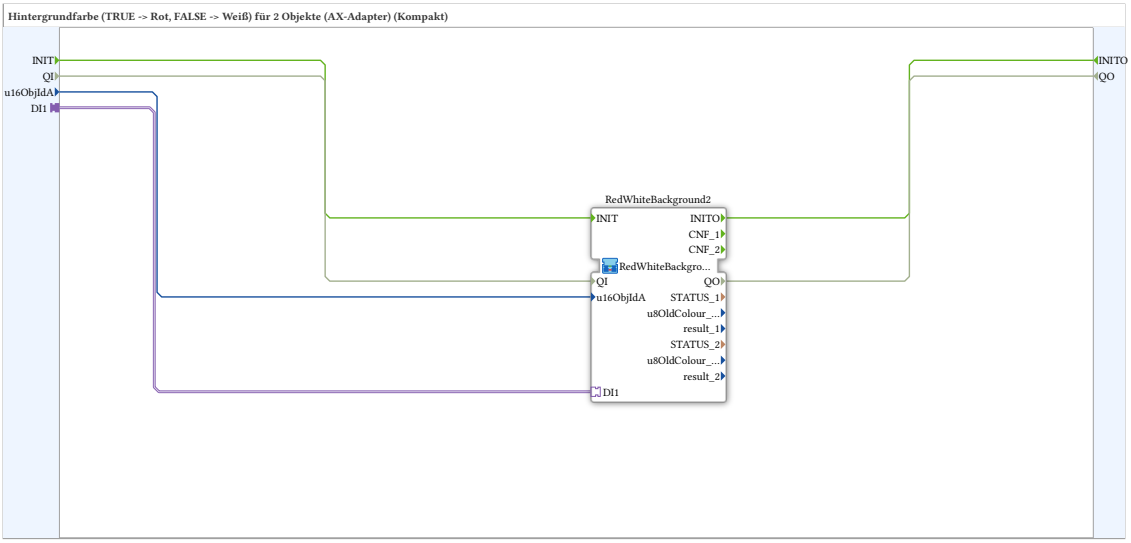
Zusammenfassung

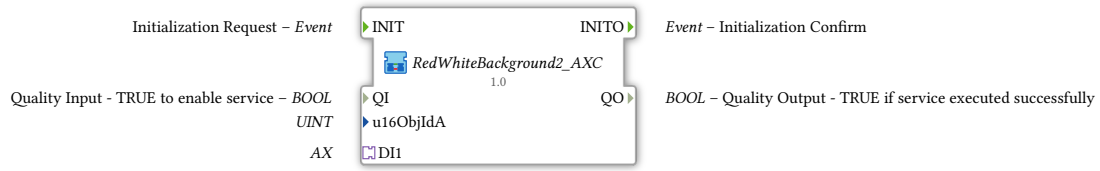
Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Weiß, 2 Objekte, Adapter-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground2_AXC





• – ○ • – ○ • – ○ •

Einleitung

`RedWhiteBackground2_AXC` ist ein **Kompakt-Wrapper** um `RedWhiteBackground2_AX` — reicht `u16ObjId` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Schnittstelle (Quality-Muster): Der Wrapper besitzt zusätzlich den Ereignis-Eingang `INIT` und den Ereignis-Ausgang `INITO` (je `EInit`) sowie den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`) und den Datenausgang `QO` (`BOOL`). Sie sind direkt an die innere Instanz `RedWhiteBackground2` durchgeschleift: `INIT` ⟶ `RedWhiteBackground2.INIT`, `RedWhiteBackground2.INITO` ⟶ `INITO`, `QI` ⟶ `RedWhiteBackground2.QI`, `RedWhiteBackground2.QO` ⟶ `QO`.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

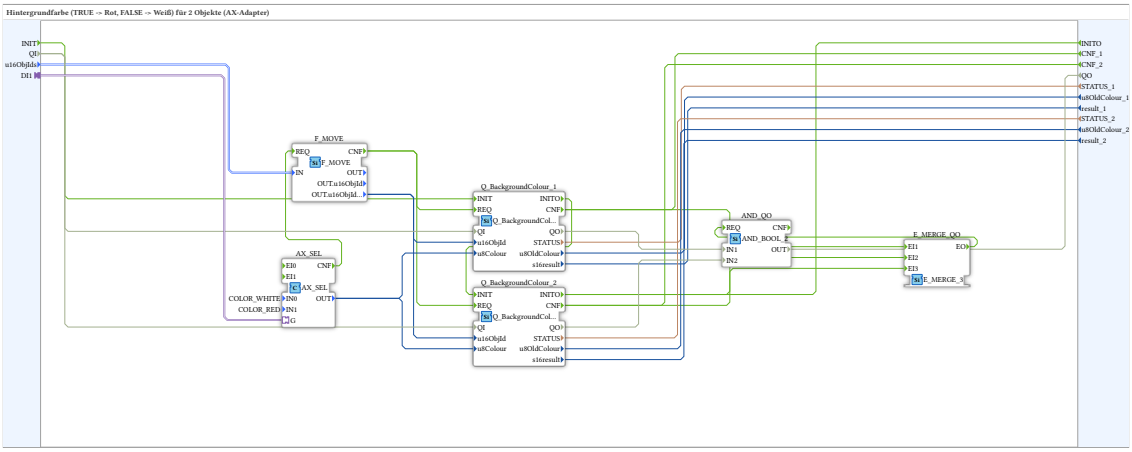
Zusammenfassung

Reine Weiterleitung an `RedWhiteBackground2_AX` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground2_AXS



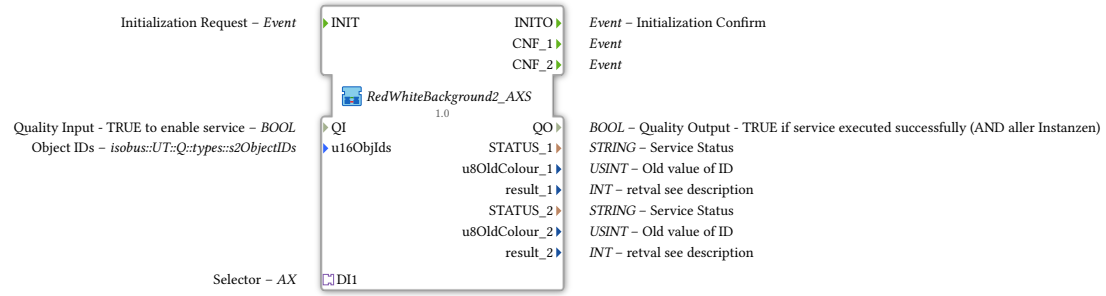
• – ○ • – ○ • – ○ •

Einleitung

`RedWhiteBackground2_AXS` schaltet die VT-Hintergrundfarbe von 2 Objekten anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Initialisierung und Status (INIT/INITO, QI/QO)

- **Ereignis-Eingang** `INIT` (`EInit`, Initialization Request) und **Ereignis-Ausgang** `INITO` (`EInit`, Initialization Confirm)
- **Daten-Eingang** `QI` (`BOOL`, Vorgabe `TRUE`, Quality Input) wird an `Q_BackgroundColour_1.QI` und `Q_BackgroundColour_2.QI` weitergegeben
- **Daten-Ausgang** `QO` (`BOOL`, Quality Output): UND (`AND_QO`, Typ `iec61131::booleanOperators::AND_BOOL_2`) der `QO`-Ausgänge von `Q_BackgroundColour_1.QO` und `Q_BackgroundColour_2.QO`
- **Initialisierungskette**: `INIT` ⟶ `Q_BackgroundColour_1.INIT`, `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`, `Q_BackgroundColour_2.INITO` ⟶ `INITO`
- `E_MERGE_QO` (Typ `iec61499::events::E_MERGE_3`) führt `Q_BackgroundColour_2.INITO`, `Q_BackgroundColour_1.CNF` und `Q_BackgroundColour_2.CNF` zusammen und löst `AND_QO.REQ` aus

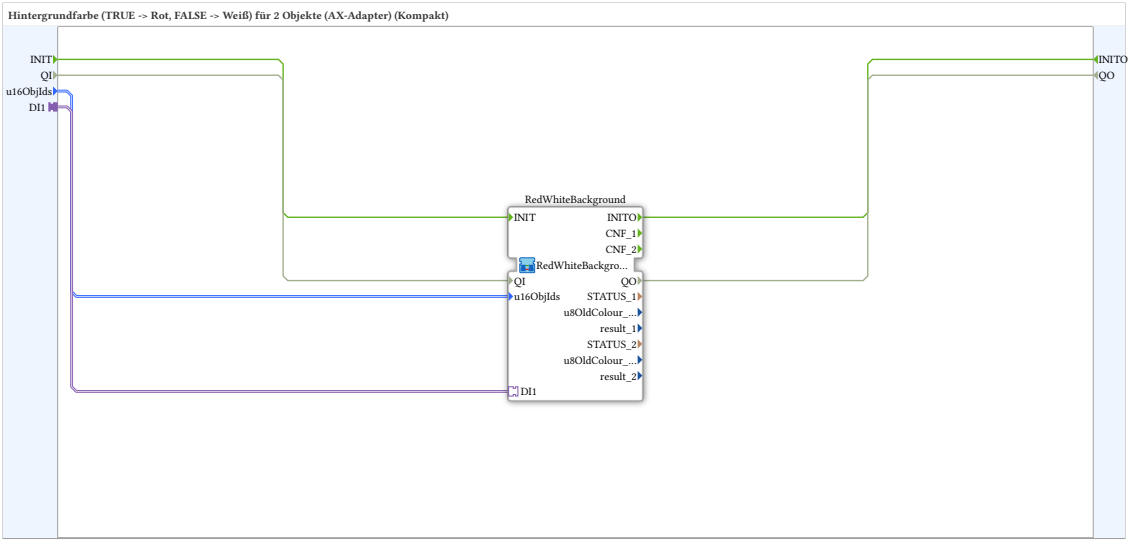
Zusammenfassung

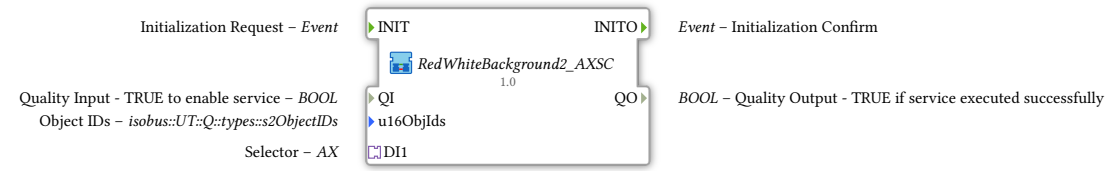
Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Weiß, 2 Objekte, Adapter-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground2_AXSC





• — ○ • — ○ • — ○ •

Einleitung

`RedWhiteBackground2_AXSC` ist ein **Kompakt-Wrapper** um `RedWhiteBackground2_AXS` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen. Zusätzlich werden `INIT` ⟶ `RedWhiteBackground.INIT`, `RedWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `RedWhiteBackground.QI` (Initialwert `TRUE`) und `RedWhiteBackground.QO` ⟶ `QO` durchgereicht.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

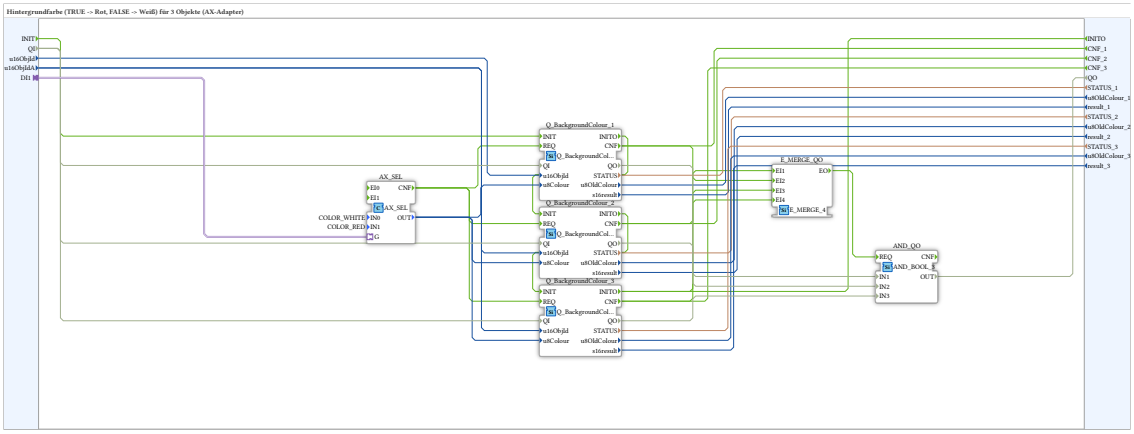
Zusammenfassung

Reine Weiterleitung an `RedWhiteBackground2_AXS` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground3_AX



• – ○ • – ○ • – ○ •

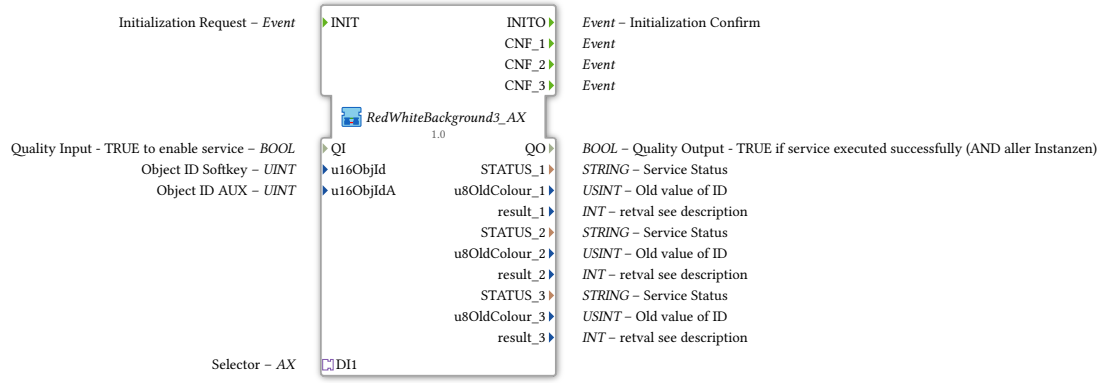
Einleitung

`RedWhiteBackground3_AX` schaltet die VT-Hintergrundfarbe von 3 Objekten anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über die Eingänge `u16ObjId, u16ObjIdA` übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`u16ObjId`	`Q_BackgroundColour` (normales Objekt)
2	`u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
3	`u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)

Initialisierung und Qualität: Zusätzlich zu den bisherigen Ereignissen besitzt der Baustein den Ereigniseingang `INIT`, den Ereignisausgang `INITO` sowie `QI` (BOOL, Eingang, Initialwert `TRUE`) und `QO` (BOOL, Ausgang, UND aller Instanzen). Init-Kette: `INIT` ⟶ `Q_BackgroundColour_1` ⟶ `Q_BackgroundColour_2` ⟶ `Q_BackgroundColour_3` ⟶ `INITO` (jeweils `INITO` ⟶ nächstes `INIT`). `QI` wird an alle `Q_BackgroundColour_n.QI` weitergereicht. `QO` bildet `AND_QO` (`iec61131::booleanOperators::AND_BOOL_3`): `Q_BackgroundColour_1.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN2`, `Q_BackgroundColour_3.QO` ⟶ `AND_QO.IN3`, `AND_QO.OUT` ⟶ `QO`. `E_MERGE_QO` (`E_MERGE_4`) führt `Q_BackgroundColour_3.INITO` und die `CNF`-Ausgänge `Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF`, `Q_BackgroundColour_3.CNF` zusammen; `E_MERGE_QO.EO` triggert `AND_QO.REQ`.

Allgemeines Muster (Selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



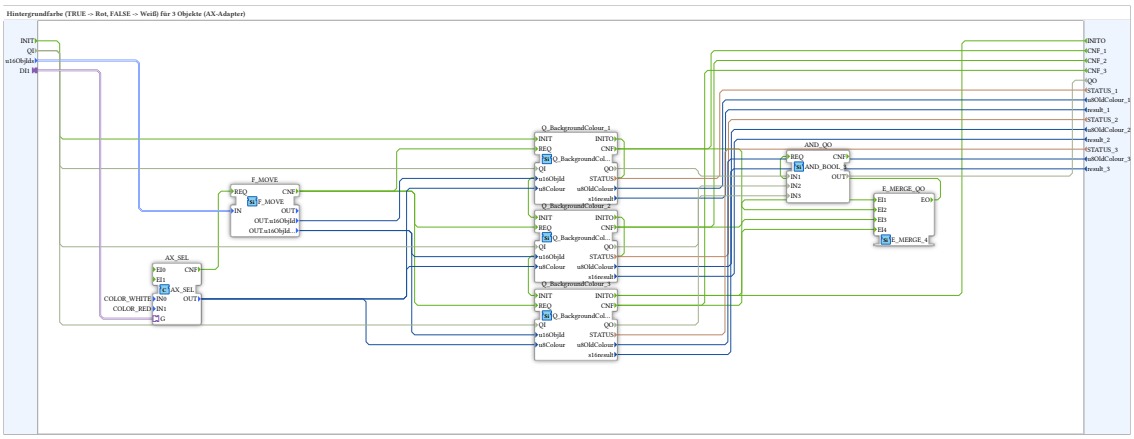
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Weiß, 3 Objekte, Adapter-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground3_AXS



• — ○ • — ○ • — ○ •

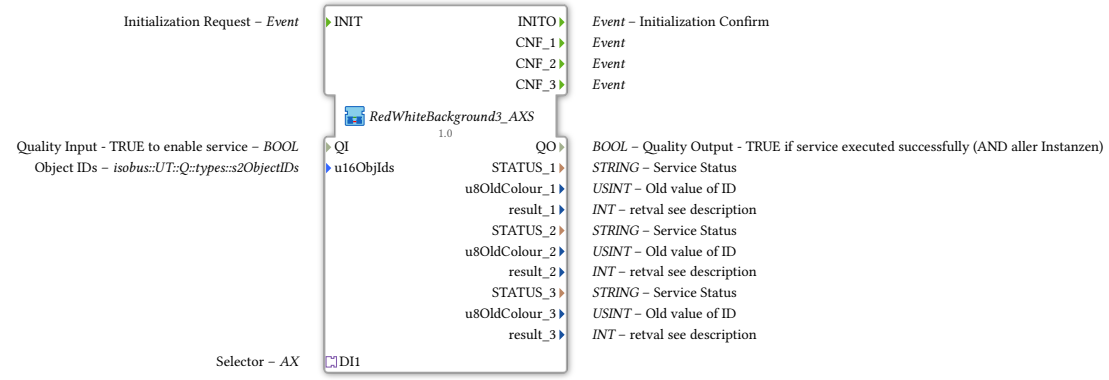
Einleitung

`RedWhiteBackground3_AXS` schaltet die VT-Hintergrundfarbe von 3 Objekten anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`F_MOVE.OUT.u16ObjId`	`Q_BackgroundColour` (normales Objekt)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
3	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Zusätzlich besitzt der Baustein die Schnittstelle für Initialisierung und Qualität: Ereigniseingang `INIT`, Ereignisausgang `INITO`, Dateneingang `QI` (`BOOL`, Initialwert `TRUE`) und Datenausgang `QO` (`BOOL`, AND aller Instanzen). `INIT` ⟶ `Q_BackgroundColour_1.INIT`, `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`, `Q_BackgroundColour_2.INITO` ⟶ `Q_BackgroundColour_3.INIT`, `Q_BackgroundColour_3.INITO` ⟶ `INITO` (die drei Instanzen werden also nacheinander initialisiert). `QI` geht an `QI` aller drei Instanzen. `QO` ist der Ausgang der Instanz `AND_QO` (`iec61131::booleanOperators::AND_BOOL_3`), die `Q_BackgroundColour_1.QO`, `Q_BackgroundColour_2.QO` und `Q_BackgroundColour_3.QO` verknüpft; ihr `REQ` wird von `E_MERGE_QO` (`iec61499::events::E_MERGE_4`) ausgelöst, das `Q_BackgroundColour_3.INITO` und die `CNF`-Ereignisse der drei Instanzen zusammenführt.



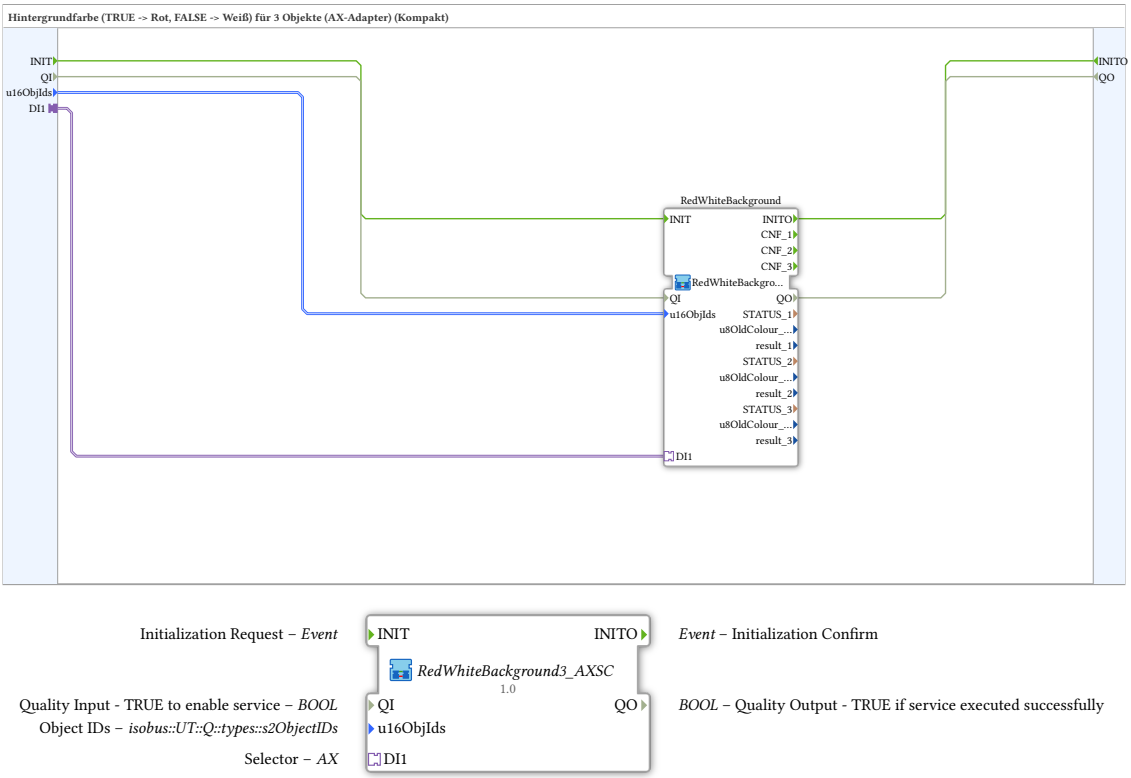
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Weiß, 3 Objekte, Adapter-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground3_AXSC



• — ○ • — ○ • — ○ •

Einleitung

`RedWhiteBackground3_AXSC` ist ein **Kompakt-Wrapper** um `RedWhiteBackground3_AXS` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

Initialisierung und Qualität

- **Ereignis-Eingang `INIT`** (`EInit`) und **Ereignis-Ausgang `INITO`** (`EInit`) werden durchgereicht: `INIT` ⟶ `RedWhiteBackground.INIT`; `RedWhiteBackground.INITO` ⟶ `INITO`.
- **Dateneingang `QI`** (`BOOL`, Vorgabe `TRUE`) ⟶ `RedWhiteBackground.QI`; **Datenausgang `QO`** (`BOOL`) ⟵ `RedWhiteBackground.QO`.

Die innere SubApp-Instanz `RedWhiteBackground` ist vom Typ `RedWhiteBackground3_AXS`.

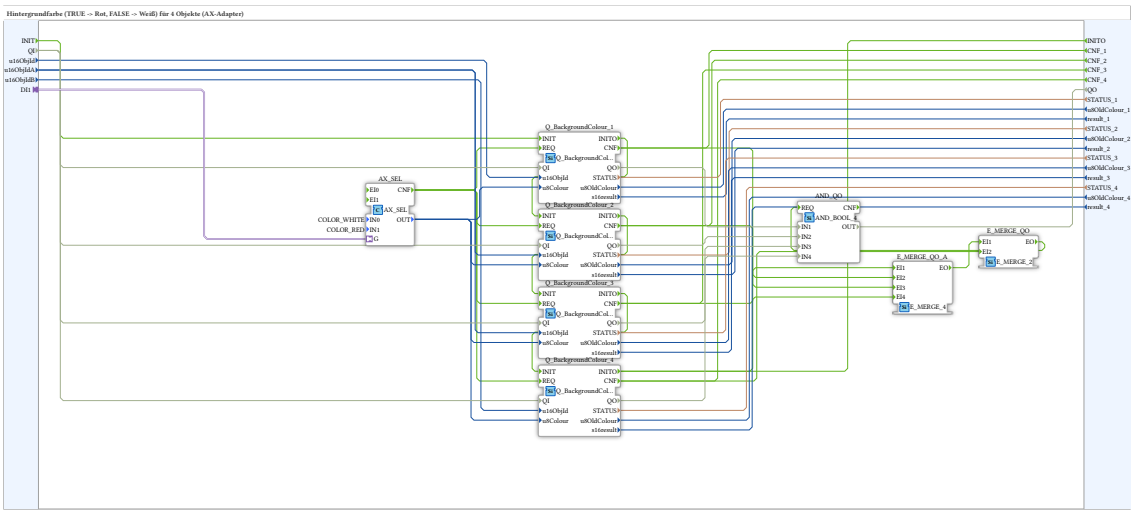
Zusammenfassung

Reine Weiterleitung an `RedWhiteBackground3_AXS` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground4_AX



• – ○ • – ○ • – ○ •

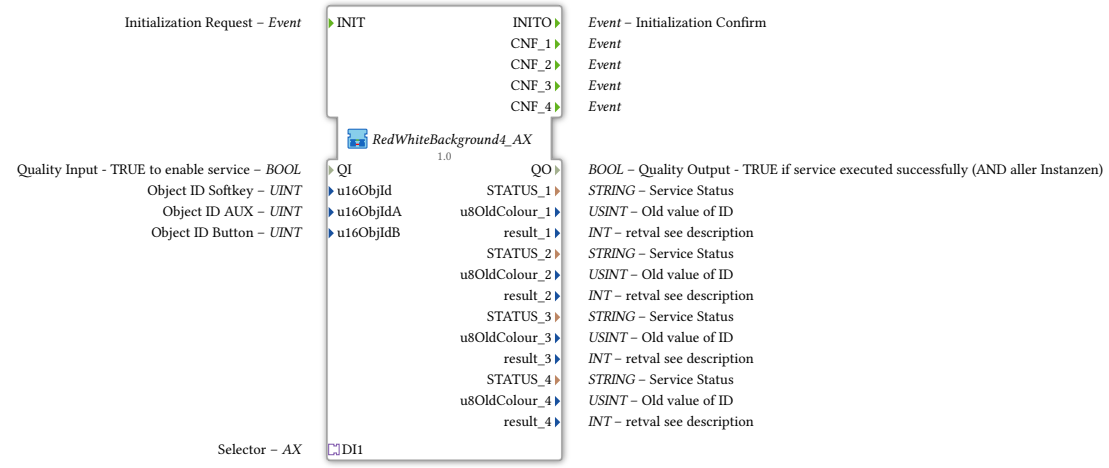
Einleitung

`RedWhiteBackground4_AX` schaltet die VT-Hintergrundfarbe von 4 Objekten anhand eines booleschen Selector-Signals: `TRUE` → **Rot**, `FALSE` → **Weiß**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über die Eingänge `u16ObjId, u16ObjIdA, u16ObjIdB` übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`u16ObjId`	`Q_BackgroundColour` (normales Objekt)
2	`u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
3	`u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)
4	`u16ObjIdB`	`Q_BackgroundColour` (normales Objekt)

Schnittstelle (Quality-Muster): Ereignis-Eingang `INIT` und Ereignis-Ausgang `INITO` (je `EInit`), Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`) und Datenausgang `QO` (`BOOL`, UND aller Instanzen). Initialisierung: `INIT` → `Q_BackgroundColour_1.INIT`; `Q_BackgroundColour_1.INITO` → `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` → `Q_BackgroundColour_3.INIT`; `Q_BackgroundColour_3.INITO` → `Q_BackgroundColour_4.INIT`; `Q_BackgroundColour_4.INITO` → `INITO`. Qualität: `QI` → `Q_BackgroundColour_1.QI`, `Q_BackgroundColour_2.QI`, `Q_BackgroundColour_3.QI`, `Q_BackgroundColour_4.QI`; `Q_BackgroundColour_1.QO`, `Q_BackgroundColour_2.QO`, `Q_BackgroundColour_3.QO`, `Q_BackgroundColour_4.QO` → `AND_QO.IN1` bis `AND_QO.IN4` (`AND_QO`: `iec61131::booleanOperators::AND_BOOL_4`); `AND_QO.OUT` → `QO`. Der Trigger `AND_QO.REQ` kommt über zwei Merge-Bausteine: `E_MERGE_QO_A` (`E_MERGE_4`) führt `Q_BackgroundColour_4.INITO` und die `CNF` von `Q_BackgroundColour_1` bis `_3` zusammen, `E_MERGE_QO` (`E_MERGE_2`) dessen Ausgang und `Q_BackgroundColour_4.CNF`; `E_MERGE_QO.EO` → `AND_QO.REQ`.

Allgemeines Muster (Selector → `AX_SEL`/`F_SEL` → `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



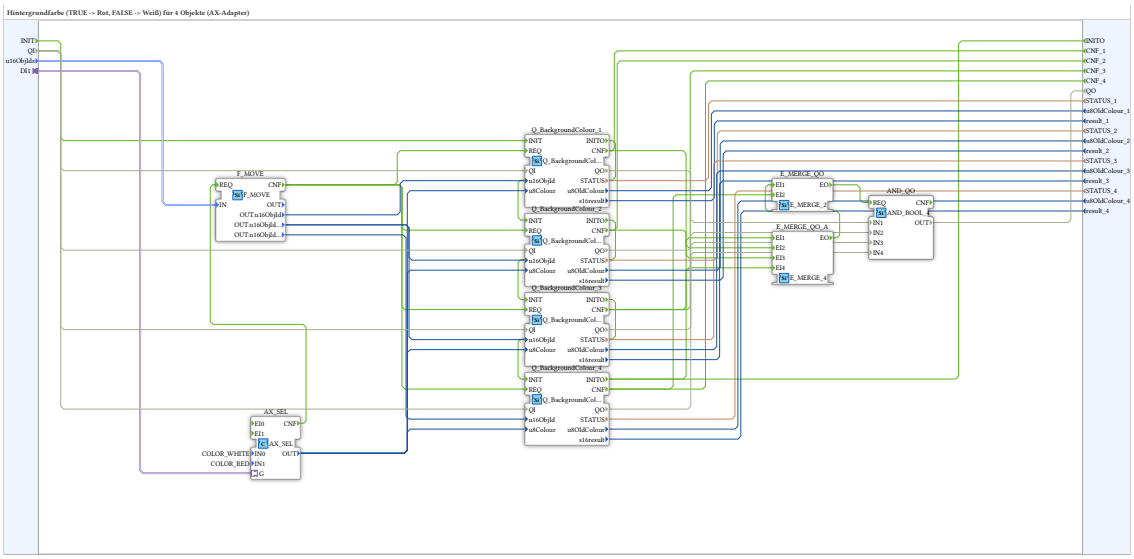
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Weiß, 4 Objekte, Adapter-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground4_AXS



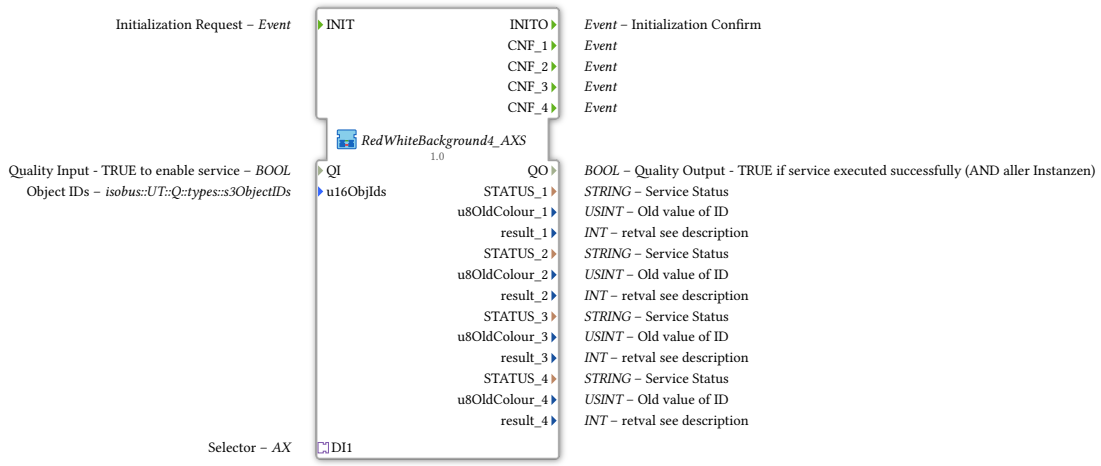
• — ○ • — ○ • — ○ •

Einleitung

`RedWhiteBackground4_AXS` schaltet die VT-Hintergrundfarbe von 4 Objekten anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt über einen `AX`-Adapter-Socket (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s10bjectID` (`u16ObjIds`) übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`F_MOVE.OUT.u16ObjId`	`Q_BackgroundColour` (normales Objekt)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
3	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)
4	`F_MOVE.OUT.u16ObjIdB`	`Q_BackgroundColour` (normales Objekt)

Allgemeines Muster (Selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Initialisierung und Status (INIT/INITO, QI/QO)

- **Ereignis-Eingang** `INIT` (`EInit`, Initialization Request) und **Ereignis-Ausgang** `INITO` (`EInit`, Initialization Confirm)
- **Daten-Eingang** `QI` (`BOOL`, Vorgabe `TRUE`, Quality Input) wird an `Q_BackgroundColour_1.QI`, `Q_BackgroundColour_2.QI`, `Q_BackgroundColour_3.QI` und `Q_BackgroundColour_4.QI` weitergegeben
- **Daten-Ausgang** `QO` (`BOOL`, Quality Output): UND (`AND_QO`, Typ `iec61131::booleanOperators::AND_BOOL_4`) der `QO`-Ausgänge von `Q_BackgroundColour_1.QO`, `Q_BackgroundColour_2.QO`, `Q_BackgroundColour_3.QO` und `Q_BackgroundColour_4.QO`
- **Initialisierungskette**: `INIT` ⟶ `Q_BackgroundColour_1.INIT`, `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`, `Q_BackgroundColour_2.INITO` ⟶ `Q_BackgroundColour_3.INIT`, `Q_BackgroundColour_3.INITO` ⟶ `Q_BackgroundColour_4.INIT`, `Q_BackgroundColour_4.INITO` ⟶ `INITO`
- `E_MERGE_QO_A` (Typ `iec61499::events::E_MERGE_4`) führt `Q_BackgroundColour_4.INITO`, `Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF` und `Q_BackgroundColour_3.CNF` zusammen und gibt das Ergebnis an `E_MERGE_QO.EI1`
- `E_MERGE_QO` (Typ `iec61499::events::E_MERGE_2`) führt `E_MERGE_QO_A.EO` und `Q_BackgroundColour_4.CNF` zusammen und löst `AND_QO.REQ` aus

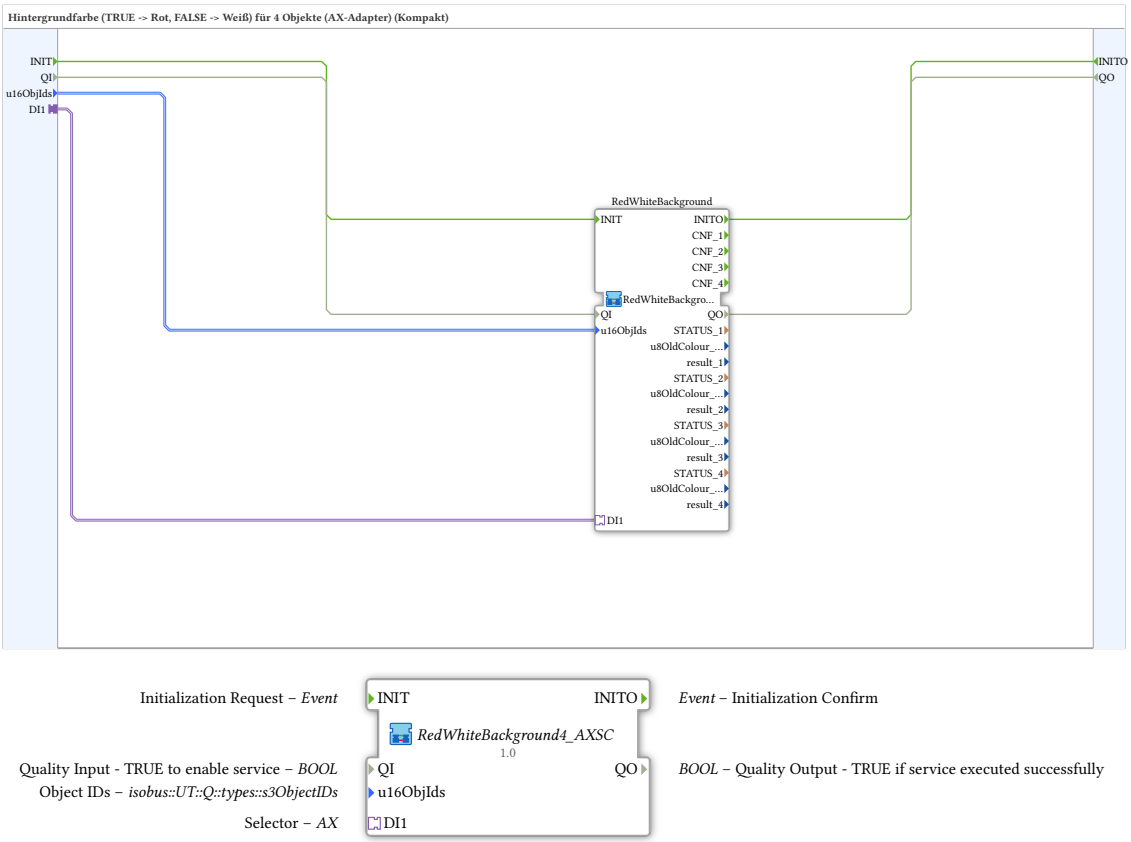
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Weiß, 4 Objekte, Adapter-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground4_AXSC



• — ○ • — ○ • — ○ •

Einleitung

`RedWhiteBackground4_AXSC` ist ein **Kompakt-Wrapper** um `RedWhiteBackground4_AXS` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen. Zusätzlich werden `INIT` ⟶ `RedWhiteBackground.INIT`, `RedWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `RedWhiteBackground.QI` (Initialwert `TRUE`) und `RedWhiteBackground.QO` ⟶ `QO` durchgereicht.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

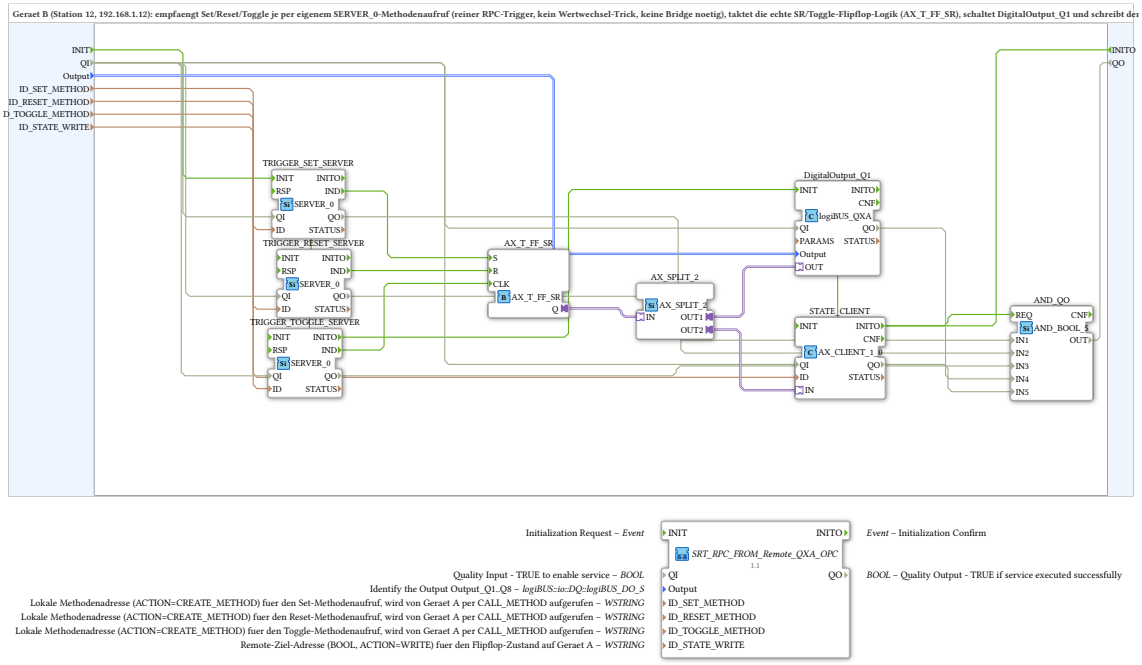
Zusammenfassung

Reine Weiterleitung an `RedWhiteBackground4_AXS` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SRT_RPC_FROM_Remote_QXA_OPC



• — ○ • — ○ • — ○ •

Einleitung

Die SubApp `SRT_RPC_FROM_Remote_QXA_OPC` dient als RPC-basierte Fernsteuerung für ein Set/Reset/Toggle-Flipflop auf einem entfernten Gerät (Gerät B). Sie empfängt über drei separate OPC-UA-Methoden (Set, Reset, Toggle) Trigger-Aufrufe von einem übergeordneten System (Gerät A), steuert die interne Flipflop-Logik, schaltet einen Digitalausgang und schreibt den neuen Zustand aktiv über einen OPC-UA-Client zurück an Gerät A. Die komplette Protokolllogik ist in der SubApp gekapselt, sodass keine zusätzliche Konfiguration in der Geräteressource erforderlich ist.

Schnittstellenstruktur

Die SubApp besitzt die Ereignisse `INIT`/`INITO`, die Daten `QI`/`QO` sowie die Daten-Eingänge für Ausgang und Adressen, aber keine externen Adapter-Schnittstellen. Die Kommunikation erfolgt intern über OPC-UA-RPC-Trigger, die in den SERVER_0-Blöcken erzeugt werden.

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; startet die Init-Kette bei `TRIGGER_SET_SERVER.INIT`

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung; kommt von `STATE_CLIENT.INITO`, dem Ende der Init-Kette

Daten-Eingänge

Name	Typ	Beschreibung
`QI`	`BOOL`	Quality Input, Initialwert `TRUE`; wird an alle fünf Dienst-FBs (drei `SERVER_0`, `DigitalOutput_Q1`, `STATE_CLIENT`) weitergereicht
`Output`	`logiBUS::io::DQ::logiBUS_DO_S`	Identifiziert den Digitalausgang (z.B. Q1..Q8). Initialwert: `logiBUS_DO::Invalid`
`ID_SET_METHOD`	`WSTRING`	Lokale Methodenadresse (ACTION=CREATE_METHOD) für den Set-Methodenaufruf

ID_RESET_METHOD	WSTRING	Lokale Methodenadresse für den Reset-Methodenaufruf
ID_TOGGLE_METHOD	WSTRING	Lokale Methodenadresse für den Toggle-Methodenaufruf
ID_STATE_WRITE	WSTRING	Remote-Zieladresse (BOOL, ACTION=WRITE) für den Flipflop-Zustand auf Gerät A

Daten-Ausgänge

- **QO** (BOOL): Quality Output, UND-Verknüpfung (AND_QO, AND_BOOL_5) der QO-Ausgänge der drei SERVER_0-Instanzen, von DigitalOutput_Q1 und von STATE_CLIENT

Adapter-Anschlüsse

Keine externen Adapter – die SubApp verwendet intern Adapterverbindungen für den Datenaustausch zwischen den Funktionsblöcken.

Funktionsweise

Die SubApp realisiert eine vollständige Fernsteuerkette:

1. **Empfang**: Drei verschiedene SERVER_0-Instanzen (TRIGGER_SET_SERVER, TRIGGER_RESET_SERVER, TRIGGER_TOGGLE_SERVER) warten auf eingehende OPC-UA-Methodenaufrufe. Jeder Aufruf erzeugt am jeweiligen IND-Ereignis einen Impuls.
2. **Flipflop-Logik**: Diese Impulse werden an die Eingänge des Funktionsblocks AX_T_FF_SR weitergeleitet: Set → S, Reset → R, Toggle → CLK. Dadurch entsteht ein SR-Flipflop mit zusätzlicher Toggle-Funktion.
3. **Ausgangsverteilung**: Der Ausgang Q des Flipflops wird über den Adapter AX_SPLIT_2 auf zwei Pfade aufgeteilt:
   - **Pfad 1** → DigitalOutput_Q1 (logiBUS_QXA) steuert den physischen Digitalausgang.
   - **Pfad 2** → STATE_CLIENT (AX_CLIENT_1_0) übernimmt den Zustand und schreibt ihn aktiv an die in ID_STATE_WRITE hinterlegte Remote-Adresse (Gerät A).
4. **Rückmeldung**: Die RSP-Ereignisse der drei SERVER_0-Blöcke sind direkt mit den jeweiligen IND-Ereignissen verbunden, sodass die Methodenbestätigung sofort erfolgt und keine Wartezeit entsteht.
5. **Initialisierung und Qualität**: INIT wird gekettet: INIT → TRIGGER_SET_SERVER → TRIGGER_RESET_SERVER → TRIGGER_TOGGLE_SERVER → DigitalOutput_Q1 → STATE_CLIENT (jeweils INITO → nächstes INIT); STATE_CLIENT.INITO → INITO und AND_QO.REQ. QI geht an alle fünf FBs; deren QO-Ausgänge gehen in dieser Reihenfolge auf AND_QO.IN1 bis AND_QO.IN5 (iec61131::booleanOperators::AND_BOOL_5), AND_QO.OUT → QO.

Technische Besonderheiten

- **Drei getrennte Methodenblöcke**: Set, Reset und Toggle werden als eigenständige OPC-UA-Methoden realisiert, was eine klare und semantisch eindeutige Kommunikation ermöglicht – kein Wertwechsel-Trick oder zusätzliche Bridge nötig.
- **Optimierte RSP-Verdrahtung**: Die direkte Verdrahtung von IND zu RSP verhindert Blockaden des OPC-UA-Server-Threads und beschleunigt die Antwortzeit erheblich.
- **Gekapselte Logik**: Die gesamte Protokoll- und Steuerlogik liegt im MyLib::sys-Composite, nicht in der Geräteressource. Dadurch ist die SubApp portabel und einfach in andere Projekte integrierbar.
- **Verwendung von logiBUS**-Typen und Adaptern für den Datenaustausch mit den Hardware-Ausgängen.

Zustandsübersicht

Die interne Flipflop-Logik (AX_T_FF_SR) besitzt genau einen Zustand (Q). Die Übergänge sind:

- **Set** (S=TRUE): Q wird TRUE.
- **Reset** (R=TRUE): Q wird FALSE.
- **Toggle** (CLK): Q wird invertiert. Der Digitalausgang DigitalOutput_Q1 spiegelt den Zustand von Q wider. Der per STATE_CLIENT zurückgeschriebene Wert entspricht ebenfalls Q.

Anwendungsszenarien

- **Fernsteuerung digitaler Ausgänge** über OPC-UA-Methoden, z.B. in verteilten Automatisierungssystemen, wenn ein zentrales Steuerungssystem (Gerät A) Aktionen auf einem entfernten I/O-Gerät (Gerät B) auslösen muss.
- **Zustandsrückmeldung**: Der aktuelle Flipflop-Zustand wird aktiv zurückgeschrieben, wodurch die Synchronisation zwischen beiden Geräten gewährleistet ist.
- **Modularer Einsatz**: Die SubApp kann in unterschiedlichen Umgebungen wiederverwendet werden, ohne die Ressourcenkonfiguration des Zielgeräts zu ändern.

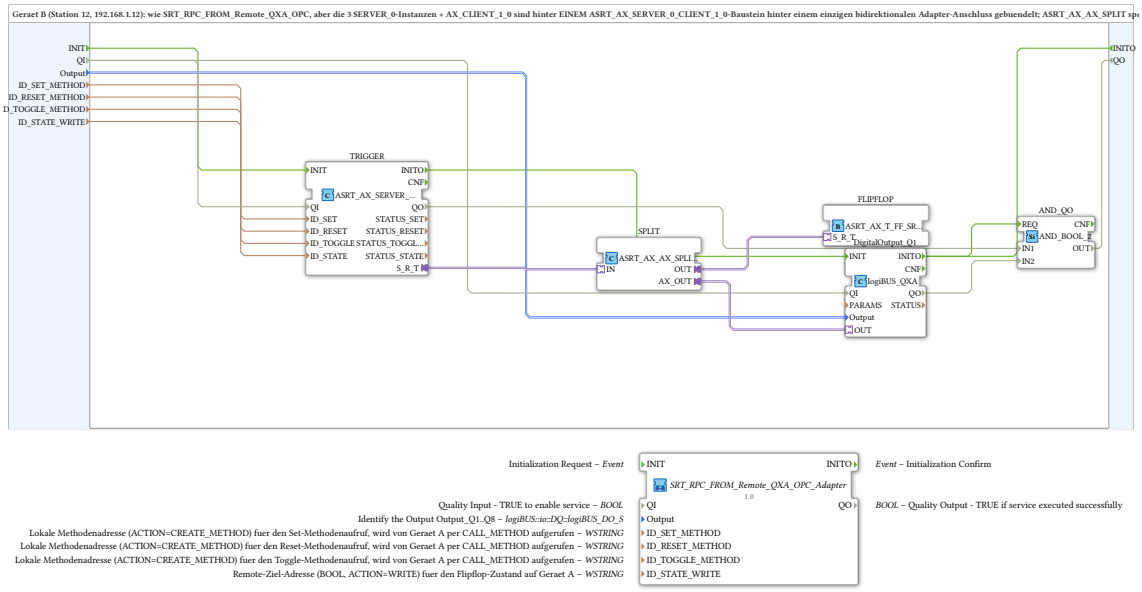
Vergleich mit ähnlichen Bausteinen

- Gegenüber einem einfachen OPC-UA-Server, der nur Werte akzeptiert, bietet dieser Baustein klar definierte Methoden, die spezifische Aktionen auslösen.
- Die aktive Rückschreibung über einen Client ist robuster als eine reine Wertüberwachung und reduziert Latenzen bei der Zustandsverteilung.
- Durch die Integration des Flipflops können komplexe Logikoperationen direkt in der SubApp abgebildet werden, ohne zusätzliche externe Logikblöcke.

Fazit

Die SubApp `SRT_RPC_FROM_Remote_QXA_OPC` stellt eine effiziente und zuverlässige Lösung zur Fernsteuerung digitaler Ausgänge über OPC-UA dar. Die getrennten Methoden, die optimierte RSP-Verdrahtung und die aktive Zustandsrückmeldung machen sie besonders geeignet für industrielle Automatisierungsumgebungen, in denen klare Kommunikation und kurze Reaktionszeiten gefordert sind. Die gekapselte Logik erhöht zudem die Wiederverwendbarkeit und vereinfacht die Projektintegration.

SRT_RPC_FROM_Remote_QXA_OPC_Adapter



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock `SRT_RPC_FROM_Remote_QXA_OPC_Adapter` ist eine SubApp (SubApplication) vom Typ `MyLib::sys` und dient als gekapselte Lösung zur Fernsteuerung eines digitalen Ausgangs (QXA) über ein Netzwerkprotokoll. Er bündelt die Funktionen eines Server-/Client-Adapters, eines Signalverteilers und einer SR/Toggle-Flipflop-Logik in einem einzigen bidirektionalen Adapter-Zugang. Das Protokoll wird dabei vollständig in der SubApp selbst implementiert und nicht in der Ressource des Geräts – ein Ansatz, der die Wiederverwendung und Modularität erhöht.

Schnittstellenstruktur

Die SubApp besitzt die Initialisierungs-Ereignisse `INIT`/`INITO`, die Qualitäts-Ports `QI`/`QO` und die übrigen **Daten-Eingänge** Adapter-Schnittstellen nach außen sind nicht vorhanden. Die Eingänge werden innerhalb der SubApp an die internen Funktionsblöcke weitergeleitet.

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; geht an `TRIGGER.INIT`

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierung abgeschlossen; kommt von `DigitalOutput_Q1.INITO`

Daten-Eingänge

Name	Typ	Beschreibung
`QI`	`BOOL`	Quality Input - TRUE aktiviert den Dienst; geht an `TRIGGER.QI` und `DigitalOutput_Q1.QI`. Initialwert: `TRUE`.
`Output`	`logiBUS::io::DQ::logiBUS_DO_S`	Identifiziert den gewünschten digitalen Ausgang (Q1..Q8). Initialwert: `logiBUS_DO::Invalid`.
`ID_SET_METHOD`	`WSTRING`	Lokale Methodenadresse (ACTION = `CREATE_METHOD`) für den Set-Methodenaufruf. Wird von einem entfernten Gerät per `CALL_METHOD` angesprochen.
`ID_RESET_METHOD`	`WSTRING`	Lokale Methodenadresse (ACTION = `CREATE_METHOD`) für den Reset-Methodenaufruf.
`ID_TOGGLE_METHOD`	`WSTRING`	Lokale Methodenadresse (ACTION = `CREATE_METHOD`) für den Toggle-Methodenaufruf.
`ID_STATE_WRITE`	`WSTRING`	Remote-Zieladresse (BOOL, ACTION = `WRITE`) für den Flipflop-Zustand, der auf dem entfernten Gerät geschrieben wird.

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde (AND aus `TRIGGER.QO` und `DigitalOutput_Q1.QO`)

Adapter

- Keine direkten Adapter-Schnittstellen nach außen. Der zugrunde liegende Adapter `ASRT_AX_SERVER_0_CLIENT_1_0` wird intern instanziiert.

Funktionsweise

Die SubApp realisiert einen Fernzugriff auf einen digitalen Ausgang eines anderen Geräts. Die Logik ist in vier Funktionsblöcke unterteilt:

1. **`TRIGGER`** – Baustein vom Typ `adapter::net::ASRT_AX_SERVER_0_CLIENT_1_0`.
   Dies ist der zentrale Kommunikations-Adapter, der einen Server- und einen Client-Anschluss bündelt. Über die Eingangsvariablen `ID_SET_METHOD`, `ID_RESET_METHOD`, `ID_TOGGLE_METHOD` und `ID_STATE_WRITE` werden die Methodenadressen bzw. Zieladressen bereitgestellt, die der Adapter nutzt, um entfernte Aufrufe zu empfangen bzw. zu senden.
2. **`SPLIT`** – Baustein vom Typ `adapter::events::bidirectional::ASRT_AX_AX_SPLIT`.
   Er teilt das vom Adapter kommende Signal `S_R_T` (Set/Reset/Toggle) in zwei Pfade:
   - `OUT` ⟶ an die Flipflop-Logik
   - `AX_OUT` ⟶ an den digitalen Ausgangsbaustein
3. **`FLIPFLOP`** – Baustein vom Typ `adapter::events::bidirectional::ASRT_AX_T_FF_SR_2`.
   Eine SR/Toggle-Flipflop-Logik, die den Zustand eines Bits verwaltet. Dieser Zustand kann über die `ID_STATE_WRITE`-Adresse an ein entferntes Gerät zurückgeschrieben werden.

4. **DigitalOutput_Q1** – Baustein vom Typ logiBUS::io::DQ::logiBUS_QXA.
   Steuert den realen digitalen Ausgang auf Grundlage des empfangenen Signals. Der Eingang Output (aus den SubApp-Eingängen) bestimmt dabei, welcher Ausgang (Q1..Q8) aktiviert wird.

Die Datenverbindungen verdrahten die externen Eingänge mit dem TRIGGER-Baustein, während die Adapterverbindungen die Signalweiterleitung zwischen den internen Blöcken sicherstellen. QI wird an TRIGGER und DigitalOutput_Q1 durchgereicht (statt einer festen Konstante TRUE).

Initialisierung und Qualität: INIT → TRIGGER.INIT, TRIGGER.INITO → DigitalOutput_Q1.INIT, DigitalOutput_Q1.INITO → INITO und → AND_QO.REQ. QO ist der Ausgang der Instanz AND_QO (iec61131::booleanOperators::AND_BOOL_2), die TRIGGER.QO und DigitalOutput_Q1.QO verknüpft.

Technische Besonderheiten

- **Gekapseltes Protokoll** – Die gesamte Kommunikationslogik liegt in der SubApp, nicht in der Geräteressource. Das erleichtert die Portierung und Wiederverwendung.
- **Bidirektionaler Adapter** – Der Bündelungsadapter ASRT_AX_SERVER_0_CLIENT_1_0 kombiniert Server- und Client-Funktionen in einem Baustein und reduziert so die Anzahl externer Schnittstellen.
- **Signalsplit** – Der ASRT_AX_AX_SPLIT trennt die Steuerbefehle für den Ausgang von denen für die Flipflop-Logik, wodurch beide unabhängig voneinander aktualisiert werden können.
- **Flexible Auswahl des Ausgangs** – Über den Parameter Output wird der zu steuernde logiBUS-Ausgang (Q1..Q8) dynamisch festgelegt.
- **Verwendung von WSTRING-Adressen** – Die Methoden- und Zieldressen sind als WSTRING definiert, was eine einfache Anpassung an verschiedene Netzwerkadressen erlaubt.

Zustandsübersicht

Die SubApp selbst hat keinen expliziten Zustandsautomaten, da sie über Funktionsblöcke realisiert wird. Die internen Zustände ergeben sich aus:

- **Flipflop-Zustand** (FLIPFLOP): Die Bausteine ASRT_AX_T_FF_SR_2 unterstützen die Zustände *Set*, *Reset* und *Toggle*. Die Ausgabe steuert den logischen Pegel des digitalen Ausgangs.
- **Digitalausgang** (DigitalOutput_Q1): Der Ausgang kann aktiv (TRUE) oder inaktiv (FALSE) sein, abhängig vom empfangenen Signal.
- **Kommunikationszustand** (TRIGGER): Der Adapter wechselt zwischen *Empfangen* und *Senden* von Nachrichten, abhängig von den eingehenden Methodenaufrufen und den Zieladressen.

Eine detaillierte Zustandsbeschreibung müsste sich auf die einzelnen Funktionsblock-Dokumentationen stützen.

Anwendungsszenarien

- **Fernsteuerung von Industrieausgängen** – Ein zentrales Steuerungssystem (Gerät A) sendet Set/Reset/Toggle-Befehle über das Netzwerk an einen dezentralen Ausgangsbaustein (Gerät B, hier Station 12).
- **Redundante Steuerung per RPC** – Über die bereitgestellten Methodenadressen können entfernte Systeme den Ausgangszustand lesen oder schreiben, ohne dass eine direkte Verdrahtung erforderlich ist.
- **Modulare Wiederverwendung** – Da das Protokoll in der SubApp gekapselt ist, kann derselbe Baustein auf verschiedenen Geräten ohne Anpassung der Ressourcen-Logik eingesetzt werden.

Vergleich mit ähnlichen Bausteinen

Gegenüber einer direkten Implementierung in der Ressource (z. B. in einem SRT_RPC_FROM_Remote_QXA_OPC-Baustein) bietet die SubApp-Variante folgende Vorteile:

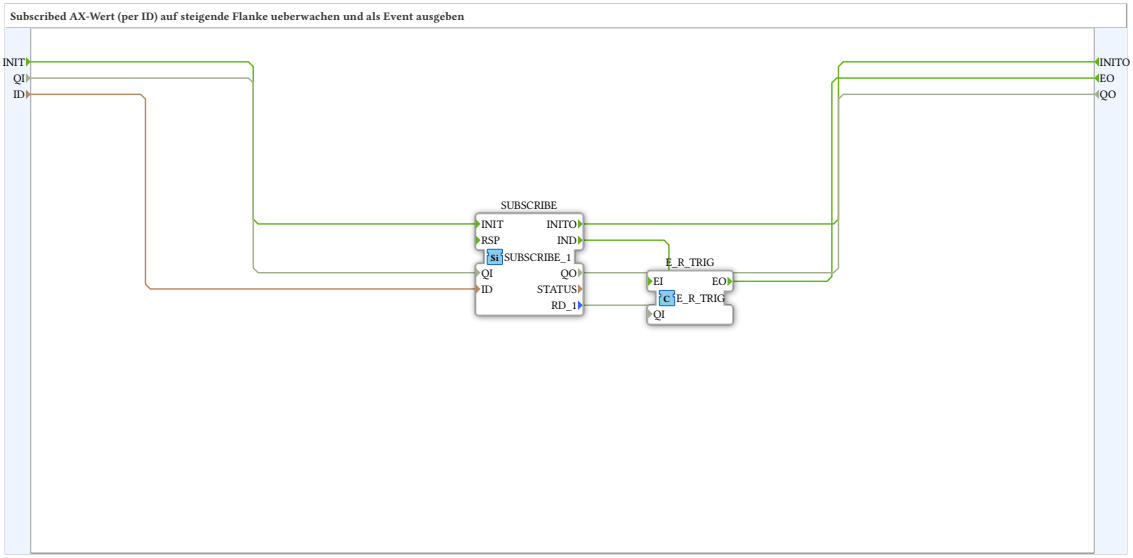
- **Erhöhte Wartbarkeit** – Änderungen am Protokoll betreffen nur die SubApp, nicht die gesamte Ressource.
- **Geringere externe Vernetzung** – Alle internen Verbindungen sind in der SubApp verborgen; die Schnittstelle nach außen ist auf die Eingangsvariablen reduziert.
- **Austauschbarkeit** – Die SubApp kann durch eine andere Implementierung mit gleicher Schnittstelle ersetzt werden, ohne die umgebende Applikation zu verändern.

Anders als direkt verdrahtete Lösungen benötigt dieser Baustein einen Adapter zur Kommunikation, was eine zusätzliche Abstraktionsebene darstellt.

Fazit

Der `SRT_RPC_FROM_Remote_QXA_OPC_Adapter` ist eine leistungsfähige SubApp zur fernsteuerbaren Ansteuerung eines logiBUS-Ausgangs. Durch die Kombination von Adapter, Splitter und Flipflop-Logik in einem gekapselten Modul wird eine hohe Flexibilität und Wiederverwendbarkeit erreicht. Die mehrschichtige Struktur ermöglicht eine klare Trennung von Kommunikations- und Steuerungslogik und eignet sich ideal für verteilte Automatisierungssysteme, bei denen ein einheitliches Schnittstellenverhalten erforderlich ist.

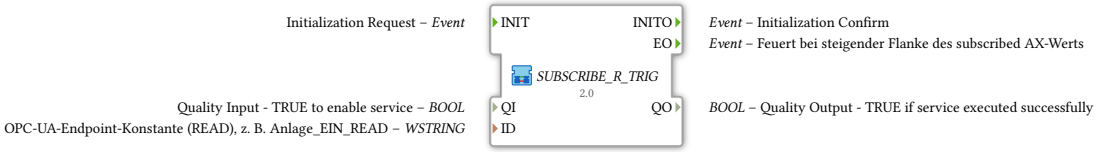
SUBSCRIBE_R_TRIG



• – ○ • – ○ • – ○ •

Einleitung

`SUBSCRIBE_R_TRIG` überwacht einen per OPC-UA abonnierten `AX`-Wert (BOOL) auf eine steigende Flanke und gibt diese als einzelnes Ereignis (`EO`) aus. Damit lässt sich ein entfernter, remote gesetzter BOOL-Wert direkt als auslösendes Ereignis verwenden, ohne den Pegel selbst weiterverarbeiten zu müssen.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: SUBSCRIBE_R_TRIG

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **SUBSCRIBE**: `iec61499::net::SUBSCRIBE_1` — abonniert den entfernten BOOL-Wert (`ID`, z. B. `Anlage_EIN_READ`).
  - **E_R_TRIG**: `iec61499::events::E_R_TRIG` — erkennt die steigende Flanke des abonnierten Werts und feuert `EO`.
- **Funktionsweise**: Seit Version 2.0 sind das rohe `SUBSCRIBE_1` und `E_R_TRIG` direkt verdrahtet (kein doppelter Flankenspeicher mehr): `SUBSCRIBE.IND` und `SUBSCRIBE.RD_1` führen direkt in `E_R_TRIG`, dessen Flankenerkennung das SubApp-Ereignis `EO` auslöst.
- **Schnittstelle**: Ereignis-Eingang `INIT` (`EInit`), Ereignis-Ausgänge `INITO` (`EInit`) und `EO`; Dateneingänge `QI` (`BOOL`, Vorgabe `TRUE`) und `ID` (`WSTRING`); Datenausgang `QO` (`BOOL`).

Programmablauf und Verbindungen

1. `ID` ⟶ `SUBSCRIBE.ID` (Datenverbindung, ausgeblendet).
2. `SUBSCRIBE.IND` ⟶ `E_R_TRIG.EI`; `SUBSCRIBE.RD_1` ⟶ `E_R_TRIG.QI`.

3. `E_R_TRIG.EO` ⟶ `EO` (SubApp-Ereignisausgang).
4. `INIT` ⟶ `SUBSCRIBE.INIT`; `SUBSCRIBE.INITO` ⟶ `INITO`.
5. `QI` ⟶ `SUBSCRIBE.QI`; `SUBSCRIBE.QO` ⟶ `QO`.

Anwendungsszenarien

- Ein entferntes OPC-UA-Signal (z. B. „Anlage EIN“) soll ein einmaliges Ereignis auslösen (z. B. Start einer Initialisierung), statt als dauerhafter Pegel weiterverarbeitet zu werden.

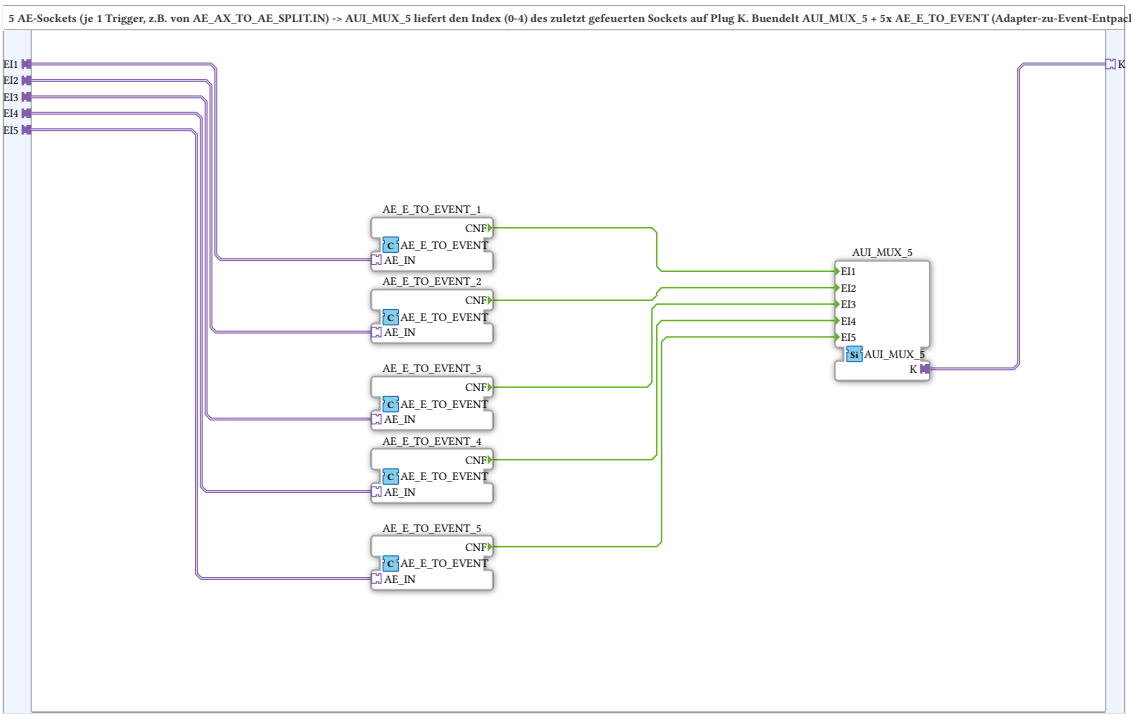
Zusammenfassung

`SUBSCRIBE_R_TRIG` verbindet OPC-UA-Subscribe direkt mit einer Flankenerkennung und liefert ein einzelnes Ereignis bei steigender Flanke des entfernten Werts.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

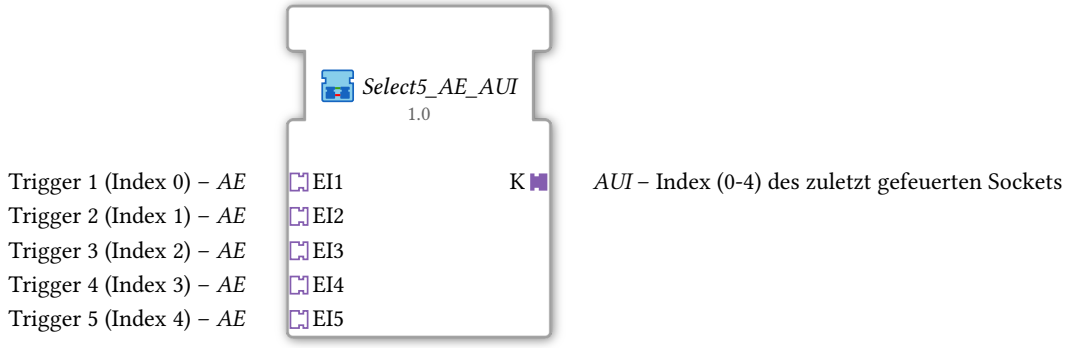
Select5_AE_AUI



• – ○ • – ○ • – ○ •

Einleitung

`Select5_AE_AUI` nimmt fünf `AE`-Sockets (je ein Trigger, z. B. von `AE_AX_TO_AE_SPLIT.IN`) entgegen und liefert über den Plug `K` den Index (0-4) des zuletzt gefeuerten Sockets. Der Baustein bündelt `AUI_MUX_5` und fünf `AE_E_TO_EVENT` (Adapter-zu-Event-Entpacker) zu einem wiederverwendbaren Baustein.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Select5_AE_AUI

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **AE_E_TO_EVENT_1** bis **AE_E_TO_EVENT_5**: je `adapter::conversion::unidirectional::AE_E_TO_EVENT` – wandelt ein `AE`-Adapterereignis in ein normales Ereignis (`CNF`)
  - **AUI_MUX_5**: `adapter::events::unidirectional::AUI_MUX_5` – liefert den Index des zuletzt aktiven Eingangs auf `K`

Schnittstellenstruktur

Ereignis-Eingänge / Ereignis-Ausgänge

Keine – Ereignisse laufen über die Adapter.

Daten-Eingänge / Daten-Ausgänge

Keine direkten Daten-Ports.

Adapter

- **EI1** (Socket, `adapter::types::unidirectional::AE`): Trigger 1 (Index 0)
- **EI2** (Socket, `AE`): Trigger 2 (Index 1)
- **EI3** (Socket, `AE`): Trigger 3 (Index 2)
- **EI4** (Socket, `AE`): Trigger 4 (Index 3)
- **EI5** (Socket, `AE`): Trigger 5 (Index 4)
- **K** (Plug, `adapter::types::unidirectional::AUI`): Index (0-4) des zuletzt gefeuerten Sockets

Programmablauf und Verbindungen

- `EI1`…`EI5` ⟶ `AE_E_TO_EVENT_1`…`AE_E_TO_EVENT_5` (`AE_IN`)
- `AE_E_TO_EVENT_n.CNF` ⟶ `AUI_MUX_5.EIn` (n = 1…5)
- `AUI_MUX_5.K` ⟶ `K`

Technische Besonderheiten

- Index-Zuordnung: `EI1` ⟶ 0, `EI2` ⟶ 1, `EI3` ⟶ 2, `EI4` ⟶ 3, `EI5` ⟶ 4
- Version 1.0 (2026-09-12): gekapselt für die exklusive TECU-Geschwindigkeitsquellen-Auswahl (5x `AE_AX_TO_AE_SPLIT.IN` als Quelle)

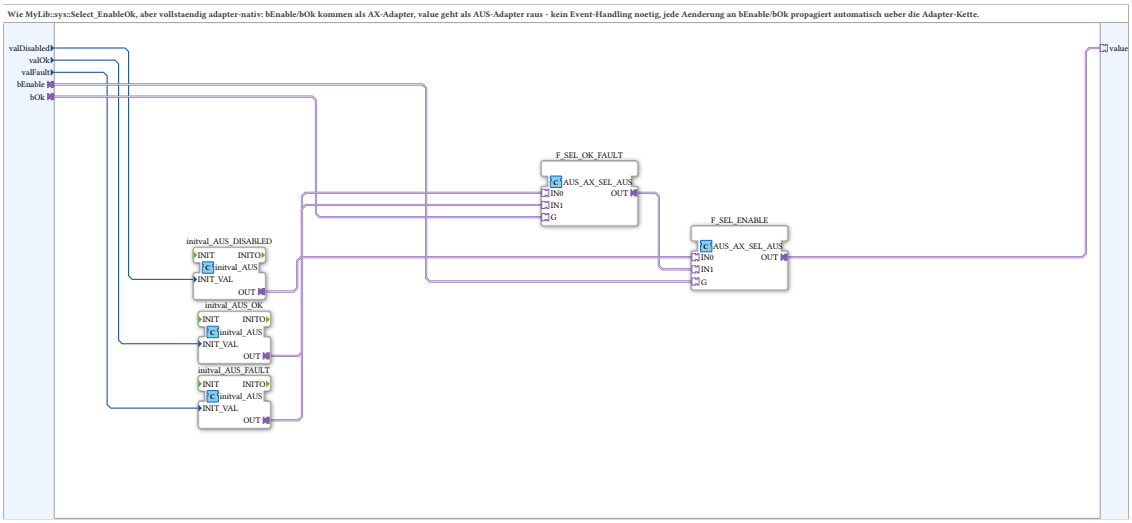
Anwendungsszenarien

- Exklusive Quellenauswahl: der zuletzt aktivierte von fünf Triggern bestimmt den gewählten Index (z. B. TECU-Geschwindigkeitsquelle)

Zusammenfassung

`Select5_AE_AUI` wandelt fünf `AE`-Trigger in den Index des zuletzt gefeuerten Triggers als `AUI`-Signal um.

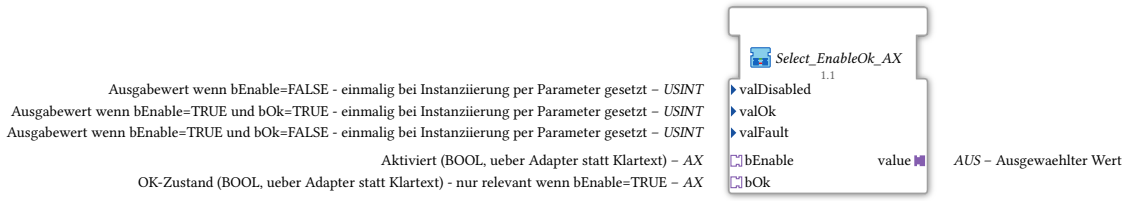
Select_EnableOk_AX



• – ○ • – ○ • – ○ •

Einleitung

`Select_EnableOk_AX` ist die adapter-native Variante von `Select_EnableOk`: `bEnable`/`bOk` kommen als AX-Adapter (statt Klartext-BOOL), `value` geht als AUS-Adapter raus. Kein Event-Handling noetig - jede Aenderung an `bEnable`/`bOk` propagiert automatisch ueber die Adapter-Kette.



Verwendete Funktionsbausteine (FBs)

- **initval_AUS_DISABLED** / **initval_AUS_OK** / **initval_AUS_FAULT** (`adapter::types::unidirectional::AUS::initval::initval_AUS`): speisen die 3 Parameter-Werte (`valDisabled`/`valOk`/`valFault`) als AUS-Adapter ein, einmalig bei Instanziierung.
- **F_SEL_OK_FAULT** (`adapter::iec61131::selection::AUS_AX_SEL_AUS`): waehlt zwischen Fault/Ok-Adapter nach `bOk` (Adapter-Gate statt Event-Selektor).
- **F_SEL_ENABLE** (`adapter::iec61131::selection::AUS_AX_SEL_AUS`): waehlt zwischen Disabled-Adapter/Zwischenergebnis nach `bEnable`.

Technische Besonderheiten

- 2-stufig wie die Event-Variante `Select_EnableOk`, aber ohne `REQ`/`CNF` auf dieser Ebene - die Adapter-Verbindungen selbst propagieren jede Aenderung.

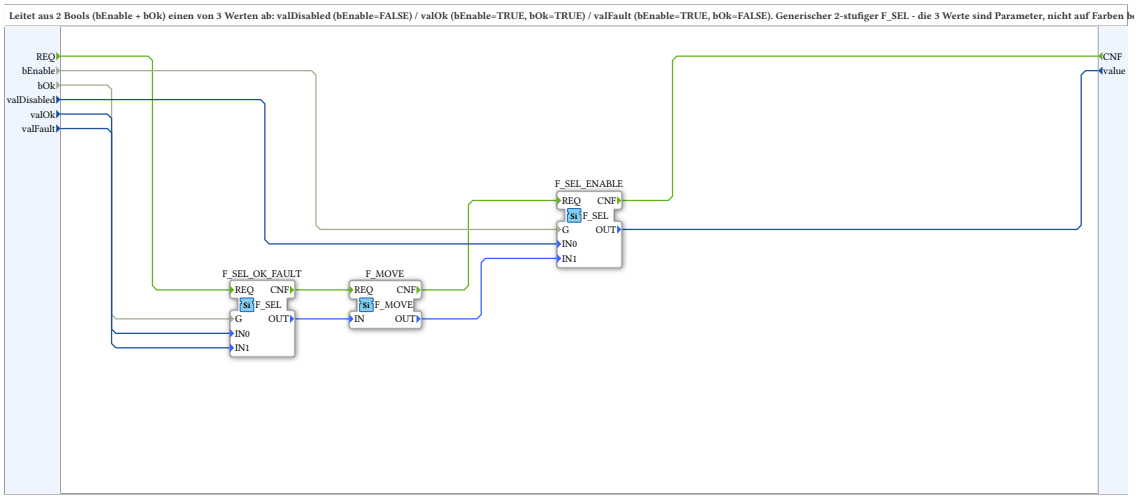
Zusammenfassung

Vollstaendig adapter-native Variante von `Select_EnableOk`, passend zur GreenBlueBackground1_AX/GreenRedBackground1_AX-Familie.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

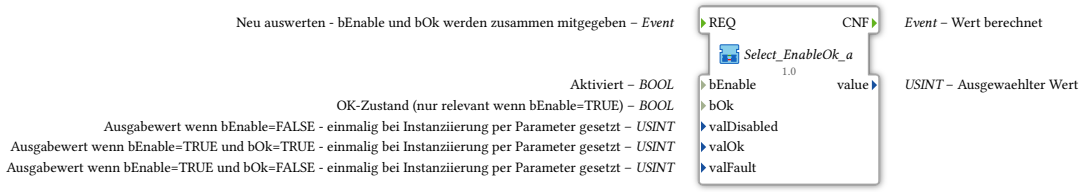
Select_EnableOk_a



• — ○ • — ○ • — ○ •

Einleitung

`Select_EnableOk_a` leitet aus zwei `BOOL`-Werten (`bEnable` und `bOk`) einen von drei Werten ab: `valDisabled` (`bEnable = FALSE`), `valOk` (`bEnable = TRUE`, `bOk = TRUE`) oder `valFault` (`bEnable = TRUE`, `bOk = FALSE`). Es ist ein generischer, zweistufiger `F_SEL`; die drei Werte sind Parameter und nicht auf Farben beschränkt.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Neu auswerten – `bEnable` und `bOk` werden zusammen mitgegeben

Ereignis-Ausgänge

- **CNF**: Wert berechnet

Daten-Eingänge

- **bEnable** (`BOOL`): Aktiviert
- **bOk** (`BOOL`): OK-Zustand (nur relevant, wenn `bEnable = TRUE`)
- **valDisabled** (`USINT`): Ausgabewert bei `bEnable = FALSE`; einmalig bei Instanziierung per Parameter gesetzt
- **valOk** (`USINT`): Ausgabewert bei `bEnable = TRUE` und `bOk = TRUE`; einmalig bei Instanziierung per Parameter gesetzt
- **valFault** (`USINT`): Ausgabewert bei `bEnable = TRUE` und `bOk = FALSE`; einmalig bei Instanziierung per Parameter gesetzt

Daten-Ausgänge

- **value** (`USINT`): Ausgewählter Wert

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Verwendete Funktionsbausteine (FBs)

- **F_SEL_OK_FAULT**: `iec61131::selection::F_SEL` – wählt mit `G = bOk` zwischen `IN0 = valFault` und `IN1 = valOk`
- **F_MOVE**: `iec61131::selection::F_MOVE` (`DataType=USINT`) – reicht das Ergebnis weiter
- **F_SEL_ENABLE**: `iec61131::selection::F_SEL` – wählt mit `G = bEnable` zwischen `IN0 = valDisabled` und `IN1` = Ergebnis von `F_MOVE`

Programmablauf und Verbindungen

1. `REQ` ⟶ `F_SEL_OK_FAULT.REQ` ⟶ `F_MOVE.REQ` ⟶ `F_SEL_ENABLE.REQ` ⟶ `CNF` (Ereigniskette).
2. `bOk` ⟶ `F_SEL_OK_FAULT.G`; `valFault` ⟶ `IN0`; `valOk` ⟶ `IN1`.
3. `F_SEL_OK_FAULT.OUT` ⟶ `F_MOVE.IN` ⟶ `F_SEL_ENABLE.IN1`.
4. `bEnable` ⟶ `F_SEL_ENABLE.G`; `valDisabled` ⟶ `F_SEL_ENABLE.IN0`.
5. `F_SEL_ENABLE.OUT` ⟶ `value`.

Technische Besonderheiten

- Die Wertetabelle:

bEnable	bOk	value
FALSE	beliebig	`valDisabled`
TRUE	TRUE	`valOk`
TRUE	FALSE	`valFault`

- Die drei Werte sind `USINT` und können z. B. Farbcodes, aber auch beliebige andere Zustandswerte sein.
- Die AX-Variante des Bausteins ist `Select_EnableOk_AX`.

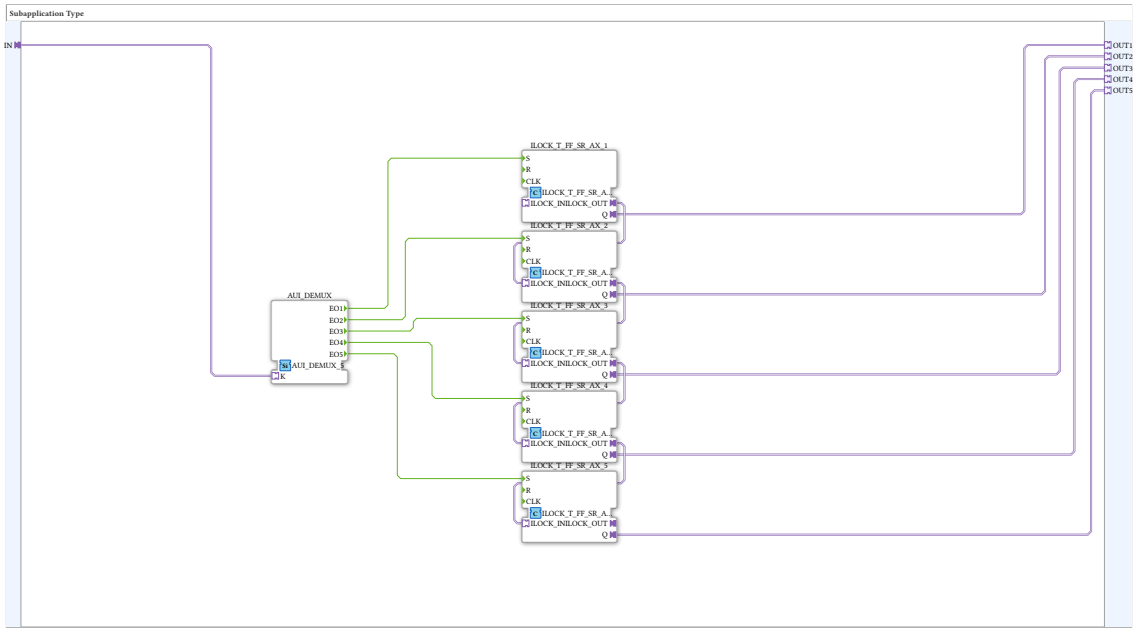
Anwendungsszenarien

- Dreistufige Statusanzeige (aus/ok/Fehler), z. B. Farbwahl für ein VT-Objekt

Fazit

`Select_EnableOk_a` wählt aus Freigabe und OK-Zustand einen von drei parametrierbaren `USINT`-Werten.

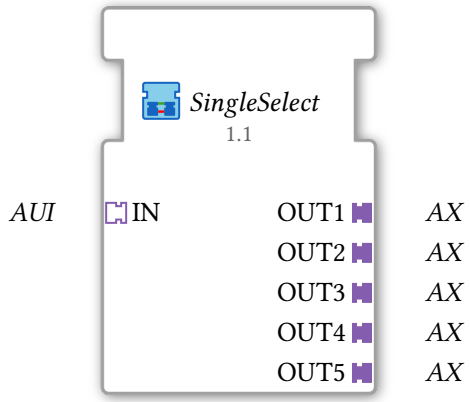
SingleSelect



• – ○ • – ○ • – ○ •

Einleitung

`SingleSelect` ist eine Einfachauswahl (Radio-Button-Verhalten) mit fünf Ausgängen: Ein Auswahlindex am AUI-Socket `IN` aktiviert genau einen der fünf AX-Ausgänge `OUT1`..`OUT5`; die übrigen werden dabei zurückgesetzt. Die Verriegelung übernimmt eine Kette aus fünf `ILOCK_T_FF_SR_AX`.



Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **IN** (Socket, `adapter::types::unidirectional::AUI`): Auswahlindex (UINT)
- **OUT1**..**OUT5** (Plugs, `adapter::types::unidirectional::AX`): Zustand der fünf Auswahlpositionen

Funktionsweise

- `IN` ⟶ `AUI_DEMUX.K`: `AUI_DEMUX` (`adapter::events::unidirectional::AUI_DEMUX_5`) leitet das Ereignis abhängig vom Index weiter: `K = 0` ⟶ `EO1`, `K = 1` ⟶ `EO2`, `K = 2` ⟶ `EO3`, `K = 3` ⟶ `EO4`, `K = 4` ⟶ `EO5`.
- `AUI_DEMUX.EOn` ⟶ `ILOCK_T_FF_SR_AX_n.S` (n = 1..5): Das gewählte Flipflop wird gesetzt.
- `ILOCK_T_FF_SR_AX_n.Q` ⟶ `OUTn`: Der Zustand jedes Flipflops erscheint am zugehörigen Ausgang.
- `ILOCK_T_FF_SR_AX_n.ILOCK_OUT` ⟶ `ILOCK_T_FF_SR_AX_(n+1).ILOCK_IN` (n = 1..4): Die fünf Flipflops sind über ihre bidirektionalen `AE2`-Verriegelungs-Adapter zu einer Kette verbunden. Setzt ein Glied seinen Ausgang, meldet es dies über die Kette, und die anderen Glieder werden zurückgesetzt (siehe `ILOCK_T_FF_SR_AX`).

Technische Besonderheiten

- Der Baustein hat keine Ereignis- oder Datenports; alles läuft über Adapter
- Version 1.1 räumte das Layout auf: `AUI_DEMUX_4` wurde in `AUI_DEMUX` umbenannt (das `_4` hatte keine Bedeutung)
- Gültige Indizes sind 0 bis 4; die Zuordnung zwischen Index und Ausgang ist `K = n - 1` ↔ `OUTn`

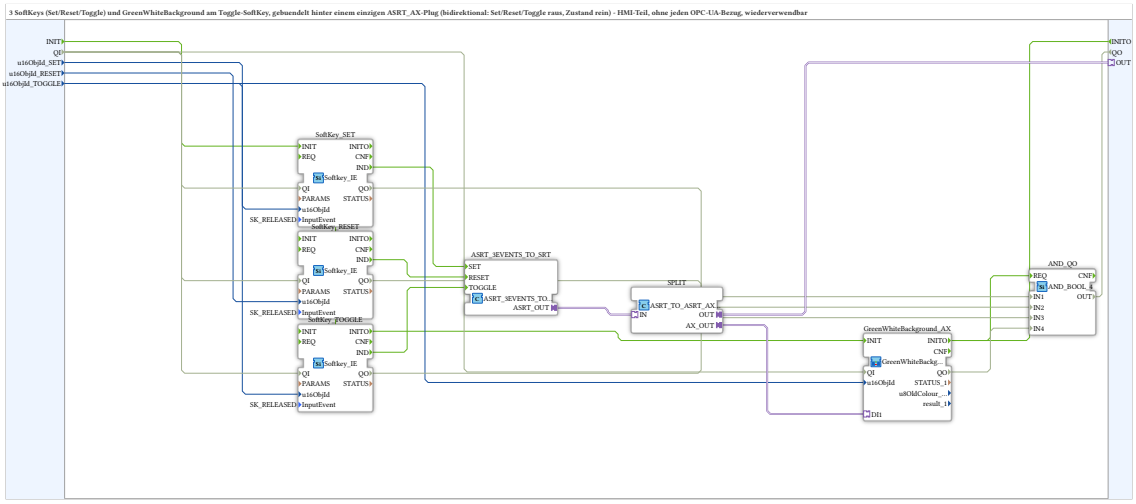
Anwendungsszenarien

- Auswahlmenüs oder Betriebsartenwahl, bei denen immer nur eine von fünf Optionen aktiv sein darf (z. B. per VT-Auswahlliste)

Fazit

`SingleSelect` setzt einen Auswahlindex mit Hilfe eines Demultiplexers und einer Interlock-Kette aus fünf `ILOCK_T_FF_SR_AX` in eine Einfachauswahl mit fünf AX-Ausgängen um.

SoftKeySRT_ASRT_AX



• – ○ • – ○ • – ○ •

Einleitung

`SoftKeySRT_ASRT_AX` buendelt 3 SoftKeys (Set/Reset/Toggle) und ein `GreenWhiteBackground1_AX` (am Toggle-SoftKey) hinter einem einzigen `ASRT_AX`-Plug (bidirektional: Set/Reset/Toggle gehen raus, Zustand kommt rein). Reiner HMI-Baustein ohne jeden OPC-UA-Bezug - das SR+Toggle-Gegenstueck zu `SoftKeySR_ASR_AX`.

Verwendete Funktionsbausteine (FBs)

- **SoftKey_SET / SoftKey_RESET / SoftKey_TOGGLE** (`isobus::UT::io::Softkey::Softkey_IE`, `InputEvent=SK_RELEASED`): die 3 physischen SoftKeys.
- **ASRT_3EVENTS_TO_SRT** (`adapter::conversion::unidirectional::ASRT_3EVENTS_TO_SRT`): wandelt die 3 SoftKey-Events in ein ASRT-Adapter-Ereignistripel.
- **SPLIT** (`adapter::events::bidirectional::ASRT_TO_ASRT_AX_SPLIT`): verzweigt den bidirektionalen `ASRT_AX`-Verkehr auf `OUT` und auf das Background-Element.
- **GreenWhiteBackground_AX** (SubApp, Typ `MyLib::sys::GreenWhiteBackground1_AX`): Statusfarbe am Toggle-SoftKey.
- **AND_QO** (`iec61131::booleanOperators::AND_BOOL_4`): verknüpft die `QO`-Ausgänge der inneren Bausteine zu `QO`.

Initialisierung und Qualität

Schnittstelle (Quality-Muster): Ereignis-Eingang `INIT` und Ereignis-Ausgang `INITO` (je `EInit`), Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`) und Datenausgang `QO` (`BOOL`). Initialisierung: `INIT` ⟶ `SoftKey_SET.INIT`; `SoftKey_SET.INITO` ⟶ `SoftKey_RESET.INIT`; `SoftKey_RESET.INITO` ⟶ `SoftKey_TOGGLE.INIT`; `SoftKey_TOGGLE.INITO` ⟶ `GreenWhiteBackground_AX.INIT`; `GreenWhiteBackground_AX.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`. Qualität: `QI` ⟶ `SoftKey_SET.QI`, `SoftKey_RESET.QI`, `SoftKey_TOGGLE.QI`, `GreenWhiteBackground_AX.QI`; `SoftKey_SET.QO`, `SoftKey_RESET.QO`, `SoftKey_TOGGLE.QO`, `GreenWhiteBackground_AX.QO` ⟶ `AND_QO.IN1` bis `AND_QO.IN4` (in dieser Reihenfolge); `AND_QO.OUT` ⟶ `QO` (`AND_QO`: `iec61131::booleanOperators::AND_BOOL_4`).

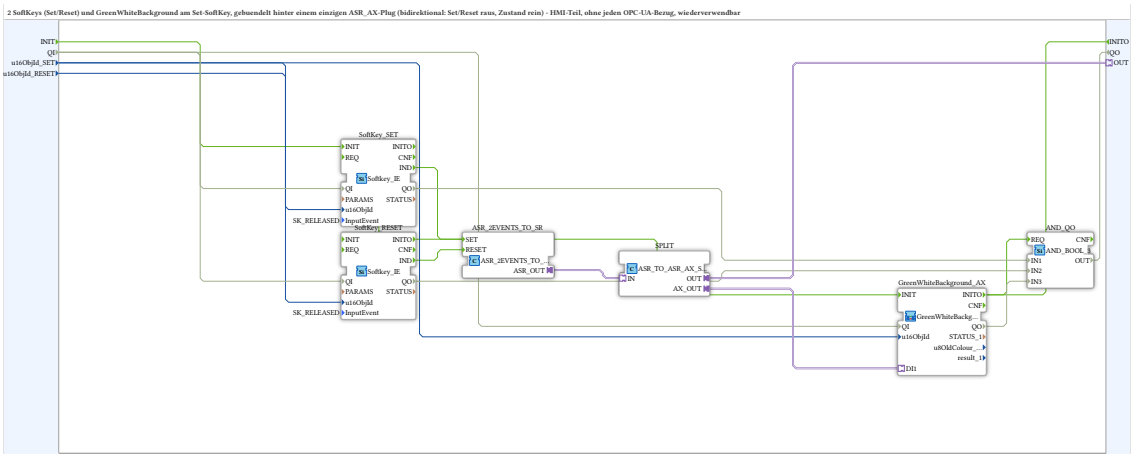
Zusammenfassung

Wiederverwendbarer Set/Reset/Toggle-mit-Statusanzeige-Baustein, gebuendelt hinter einem `ASRT_AX`-Adapter - das HMI-Gegenstueck zu `Uebung_010e_PC_A_OPC_Adapter`/`Uebung_010e_PC_B_OPC_Adapter`.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

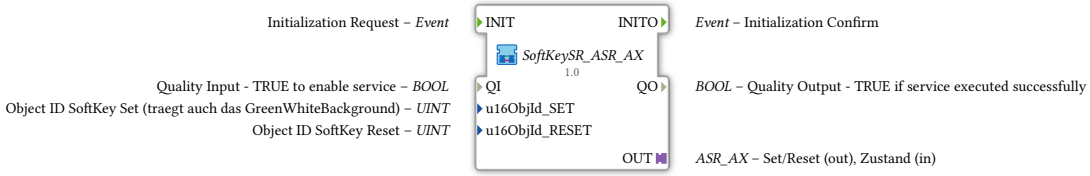
SoftKeySR_ASR_AX



• – ○ • – ○ • – ○ •

Einleitung

`SoftKeySR_ASR_AX` buendelt 2 SoftKeys (Set/Reset) und ein `GreenWhiteBackground1_AX` (am Set-SoftKey) hinter einem einzigen `ASR_AX`-Plug (bidirektional: Set/Reset gehen raus, Zustand kommt rein). Reiner HMI-Baustein ohne jeden OPC-UA-Bezug.



Schnittstelle

- **Ereignis-Eingang**: `INIT` (`EInit`, Initialization Request)
- **Ereignis-Ausgang**: `INITO` (`EInit`, Initialization Confirm)
- **Daten-Eingänge**: `QI` (`BOOL`, Vorgabe `TRUE`, Quality Input), `u16ObjId_SET`, `u16ObjId_RESET` (`UINT`, Vorgabe `ID_NULL`)
- **Daten-Ausgang**: `QO` (`BOOL`, Quality Output)
- **Plug**: `OUT` (`adapter::types::bidirectional::ASR_AX`)

Verwendete Funktionsbausteine (FBs)

- **SoftKey_SET / SoftKey_RESET** (`isobus::UT::io::Softkey::Softkey_IE`, `InputEvent=SK_RELEASED`): die beiden physischen SoftKeys.
- **ASR_2EVENTS_TO_SR** (`adapter::conversion::unidirectional::ASR_2EVENTS_TO_SR`): wandelt die 2 SoftKey-Events in ein ASR-Adapter-Ereignispaar.
- **SPLIT** (`adapter::events::bidirectional::ASR_TO_ASR_AX_SPLIT`): verzweigt den bidirektionalen `ASR_AX`-Verkehr auf `OUT` und auf das Background-Element.
- **GreenWhiteBackground_AX** (SubApp, Typ `MyLib::sys::GreenWhiteBackground1_AX`): Statusfarbe am Set-SoftKey.
- **AND_QO** (`iec61131::booleanOperators::AND_BOOL_3`): verknüpft die `QO`-Ausgänge von `SoftKey_SET`, `SoftKey_RESET` und `GreenWhiteBackground_AX` per UND zum Baustein-Ausgang `QO`.

Initialisierung und Status

- Initialisierungskette: `INIT` ⟶ `SoftKey_SET.INIT`, `SoftKey_SET.INITO` ⟶ `SoftKey_RESET.INIT`, `SoftKey_RESET.INITO` ⟶ `GreenWhiteBackground_AX.INIT`, `GreenWhiteBackground_AX.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`
- `QI` ⟶ `SoftKey_SET.QI`, `SoftKey_RESET.QI` und `GreenWhiteBackground_AX.QI`
- `SoftKey_SET.QO` ⟶ `AND_QO.IN1`, `SoftKey_RESET.QO` ⟶ `AND_QO.IN2`, `GreenWhiteBackground_AX.QO` ⟶ `AND_QO.IN3`; `AND_QO.OUT` ⟶ `QO`

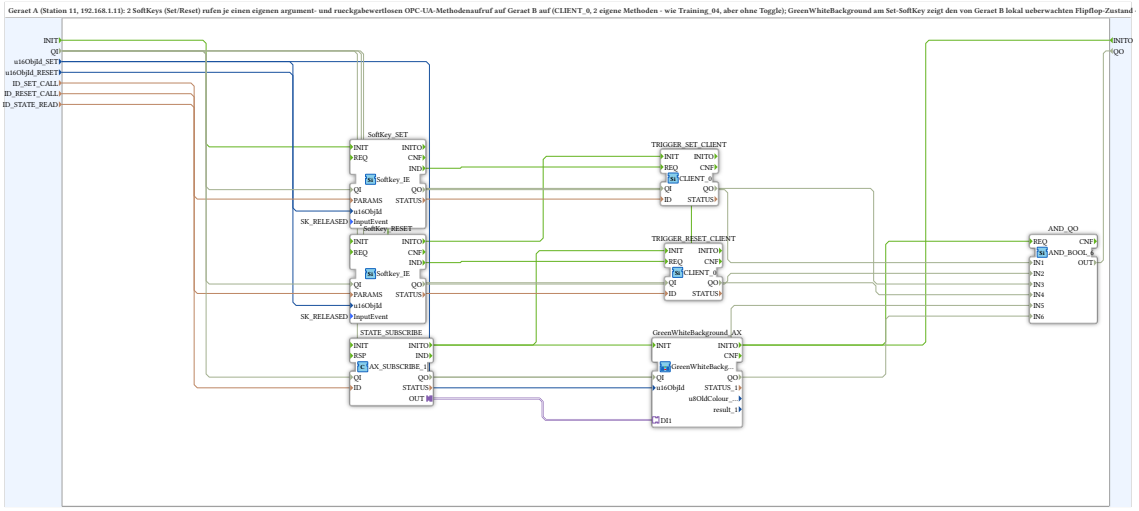
Zusammenfassung

Wiederverwendbarer Set/Reset-mit-Statusanzeige-Baustein, gebuendelt hinter einem `ASR_AX`-Adapter - das HMI-Gegenstueck zu `SoftKeySR_PC_A_OPC_Adapter`/`SoftKeySR_PC_B_OPC_Adapter`.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

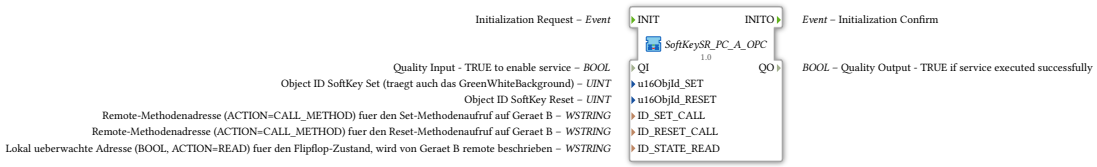
SoftKeySR_PC_A_OPC



• – ○ • – ○ • – ○ •

Einleitung

`SoftKeySR_PC_A_OPC` ist die Geraet-A-Seite (Station 11, 192.168.1.11) eines PC-zu-PC-OPC-UA-Set/Reset-Musters: 2 SoftKeys (Set/Reset) rufen je einen eigenen argument- und rueckgabewertlosen OPC-UA-Methodenaufruf auf Geraet B auf (`CLIENT_0`, 2 eigene Methoden - wie Training_04, aber ohne Toggle). `GreenWhiteBackground1_AX` am Set-SoftKey zeigt den von Geraet B lokal ueberwachten Flipflop-Zustand. „SUB style“: das Protokoll steckt im `MyLib::sys`-Composite, nicht in der Resource des Geraets - Gegenstueck: `SoftKeySR_PC_B_OPC`.



Verwendete Funktionsbausteine (FBs)

- **SoftKey_SET / SoftKey_RESET** (`isobus::UT::io::Softkey::Softkey_IE`): die beiden physischen SoftKeys.
- **TRIGGER_SET_CLIENT / TRIGGER_RESET_CLIENT** (`iec61499::net::CLIENT_0`): rufen je eine eigene argumentlose Remote-Methode auf Geraet B auf (`ID_SET_CALL`/`ID_RESET_CALL`).
- **STATE_SUBSCRIBE** (`adapter::net::AX_SUBSCRIBE_1`): abonniert den Flipflop-Zustand unter `ID_STATE_READ`.
- **GreenWhiteBackground_AX** (SubApp, Typ `MyLib::sys::GreenWhiteBackground1_AX`): zeigt den Zustand am Set-SoftKey an.

- **AND_QO** (`iec61131::booleanOperators::AND_BOOL_6`): verknüpft die `QO`-Ausgänge der sechs Bausteine oben zum SubApp-Ausgang `QO`.

Initialisierung und Qualität

- Ereignisse: `INIT` (EInit, Eingang) und `INITO` (EInit, Ausgang, kommt von `GreenWhiteBackground_AX.INITO`).
- Dateneingang `QI` (BOOL, Vorgabe `TRUE`) und Datenausgang `QO` (BOOL); die internen FBs tragen kein festes `QI=TRUE` mehr.
- Initialisierungskette: `INIT` ⟶ `SoftKey_SET.INIT` ⟶ `SoftKey_RESET.INIT` ⟶ `TRIGGER_SET_CLIENT.INIT` ⟶ `TRIGGER_RESET_CLIENT.INIT` ⟶ `STATE_SUBSCRIBE.INIT` ⟶ `GreenWhiteBackground_AX.INIT` (jeweils über `INITO`); `GreenWhiteBackground_AX.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`.
- Qualität: `QI` ⟶ `.QI` aller sechs Bausteine; deren `QO`-Ausgänge ⟶ `AND_QO.IN1` bis `IN6` (`SoftKey_SET`, `SoftKey_RESET`, `TRIGGER_SET_CLIENT`, `TRIGGER_RESET_CLIENT`, `STATE_SUBSCRIBE`, `GreenWhiteBackground_AX`); `AND_QO.OUT` ⟶ `QO`.

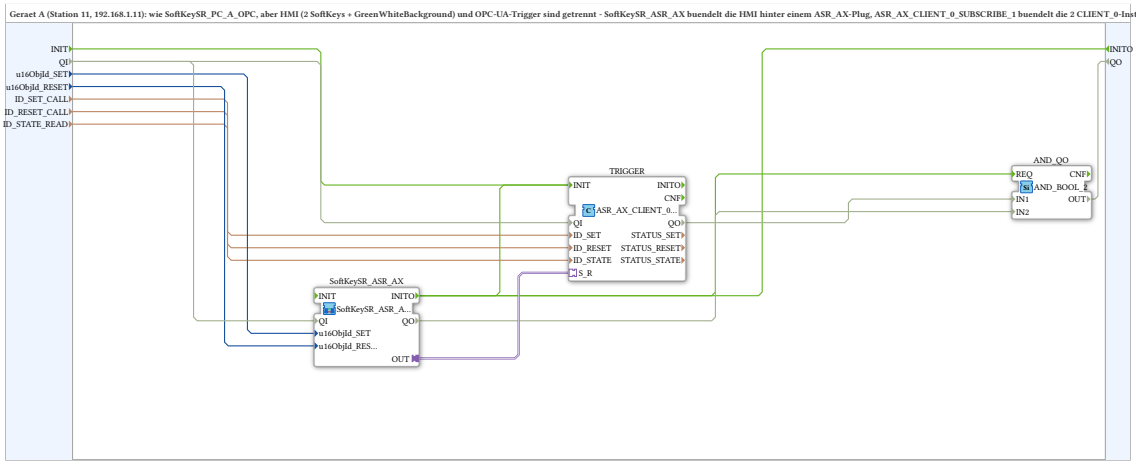
Zusammenfassung

Geraet-A-Seite eines Set/Reset-RPC-Musters mit 2 eigenen Methodenaufrufen (statt einem String-Parameter) und Zustandsrueckmeldung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

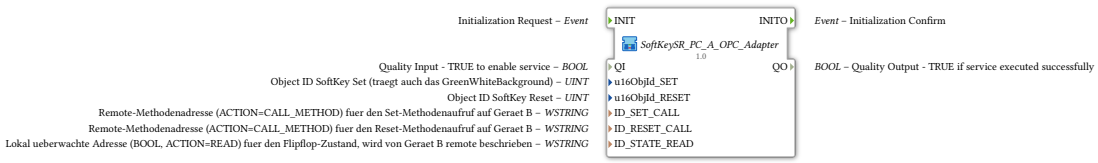
SoftKeySR_PC_A_OPC_Adapter



• — ○ • — ○ • — ○ •

Einleitung

`SoftKeySR_PC_A_OPC_Adapter` ist die adapter-gebuendelte Variante von `SoftKeySR_PC_A_OPC` (Geraet A, Station 11): HMI (2 SoftKeys + `GreenWhiteBackground`) und OPC-UA-Trigger sind getrennt. `SoftKeySR_ASR_AX` buendelt die HMI hinter einem `ASR_AX`-Plug; `ASR_AX_CLIENT_0_SUBSCRIBE_1` buendelt die 2 `CLIENT_0`-Instanzen + `AX_SUBSCRIBE_1` hinter einem `ASR_AX`-Socket. Gegenstueck: `SoftKeySR_PC_B_OPC_Adapter`.



Verwendete Funktionsbausteine (FBs)

- **SoftKeySR_ASR_AX** (SubApp, Typ `MyLib::sys::SoftKeySR_ASR_AX`): 2 SoftKeys + Statusanzeige, gebuendelt hinter einem `ASR_AX`-Plug.

- **TRIGGER** (`adapter::net::ASR_AX_CLIENT_0_SUBSCRIBE_1`): buendelt 2 Methodenaufrufe (`ID_SET_CALL`/`ID_RESET_CALL`) und Zustands-Abo (`ID_STATE_READ`) hinter einem einzigen `ASR_AX`-Socket.
- **AND_QO** (`iec61131::booleanOperators::AND_BOOL_2`): verknüpft `TRIGGER.QO` und `SoftKeySR_ASR_AX.QO` zum Ausgang `QO`.

Initialisierung und Qualität: Die SubApp besitzt die Ereignisse `INIT`/`INITO` sowie `QI` (BOOL, Initialwert `TRUE`) und `QO` (BOOL). Init-Kette: `INIT` ⟶ `TRIGGER.INIT`, `TRIGGER.INITO` ⟶ `SoftKeySR_ASR_AX.INIT`, `SoftKeySR_ASR_AX.INITO` ⟶ `INITO` und `AND_QO.REQ`. `QI` wird an `TRIGGER.QI` und `SoftKeySR_ASR_AX.QI` weitergereicht; `TRIGGER.QO` ⟶ `AND_QO.IN1`, `SoftKeySR_ASR_AX.QO` ⟶ `AND_QO.IN2`, `AND_QO.OUT` ⟶ `QO`.

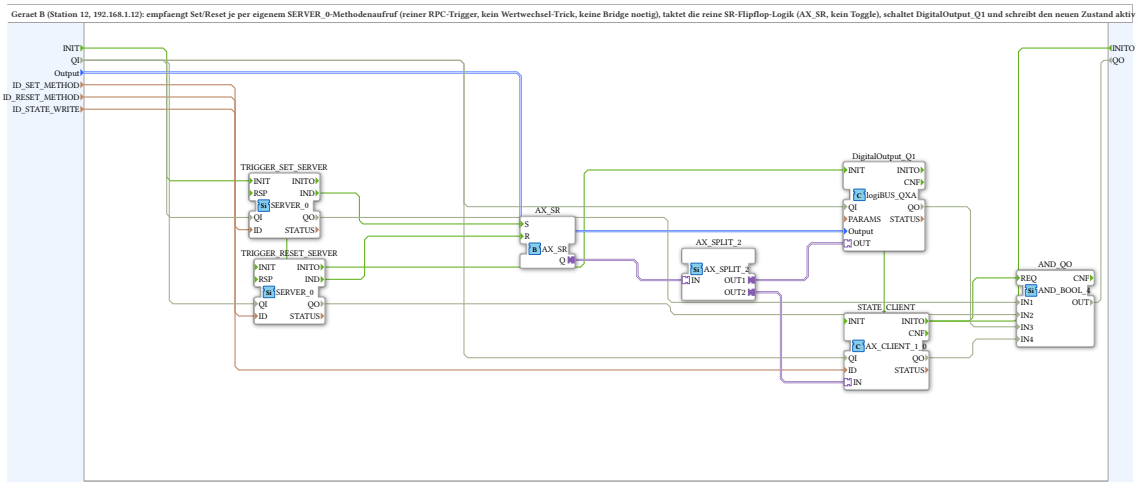
Zusammenfassung

Adapter-gebuendelte Variante von `SoftKeySR_PC_A_OPC`: HMI und Netzwerkprotokoll sauber getrennt, wiederverwendbar ueber `SoftKeySR_ASR_AX`.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

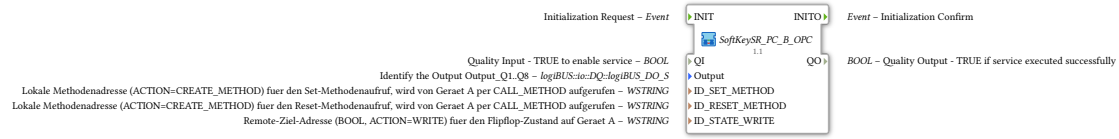
SoftKeySR_PC_B_OPC



• – ○ • – ○ • – ○ •

Einleitung

`SoftKeySR_PC_B_OPC` ist die Geraet-B-Seite (Station 12, 192.168.1.12): empfaengt Set/Reset je per eigenem `SERVER_0`-Methodenaufruf (reiner RPC-Trigger, kein Wertwechsel-Trick, keine Bridge noetig), taktet die reine SR-Flipflop-Logik (`AX_SR`, kein Toggle), schaltet `DigitalOutput_Q1` und schreibt den neuen Zustand aktiv per `AX_CLIENT_1_0` zurueck auf Geraet A. Gegenstueck: `SoftKeySR_PC_A_OPC`.



Verwendete Funktionsbausteine (FBs)

- **TRIGGER_SET_SERVER / TRIGGER_RESET_SERVER** (`iec61499::net::SERVER_0`): empfangen je einen eigenen Methodenaufruf (`ID_SET_METHOD`/`ID_RESET_METHOD`).
- **AX_SR** (`adapter::events::unidirectional::AX_SR`): reine Set/Reset-Flipflop-Logik, kein Toggle.
- **AX_SPLIT_2** (`adapter::events::unidirectional::AX_SPLIT_2`): verzweigt den Zustand auf physischen Ausgang + Rueckmeldung.
- **DigitalOutput_Q1** (`logiBUS::io::DQ::logiBUS_QXA`): physischer Ausgang.
- **STATE_CLIENT** (`adapter::net::AX_CLIENT_1_0`): schreibt den Zustand aktiv auf Geraet A unter `ID_STATE_WRITE`.

Initialisierung und Qualität

Die SubApp besitzt den Ereigniseingang `INIT`, den Ereignisausgang `INITO` (je `EInit`), den Dateneingang `QI` (`BOOL`, Default `TRUE`) und den Datenausgang `QO` (`BOOL`). Die Initialisierung läuft als Kette: `INIT` → `TRIGGER_SET_SERVER.INIT`, `TRIGGER_SET_SERVER.INITO` → `TRIGGER_RESET_SERVER.INIT`, `TRIGGER_RESET_SERVER.INITO` → `DigitalOutput_Q1.INIT`, `DigitalOutput_Q1.INITO` → `STATE_CLIENT.INIT`, `STATE_CLIENT.INITO` → `INITO` (und → `AND_QO.REQ`). `QI` geht an `TRIGGER_SET_SERVER`, `TRIGGER_RESET_SERVER`, `DigitalOutput_Q1` und `STATE_CLIENT`. `QO` ist der Ausgang der Instanz `AND_QO` (`iec61131::booleanOperators::AND_BOOL_4`), die die `QO` dieser vier Instanzen verknüpft.

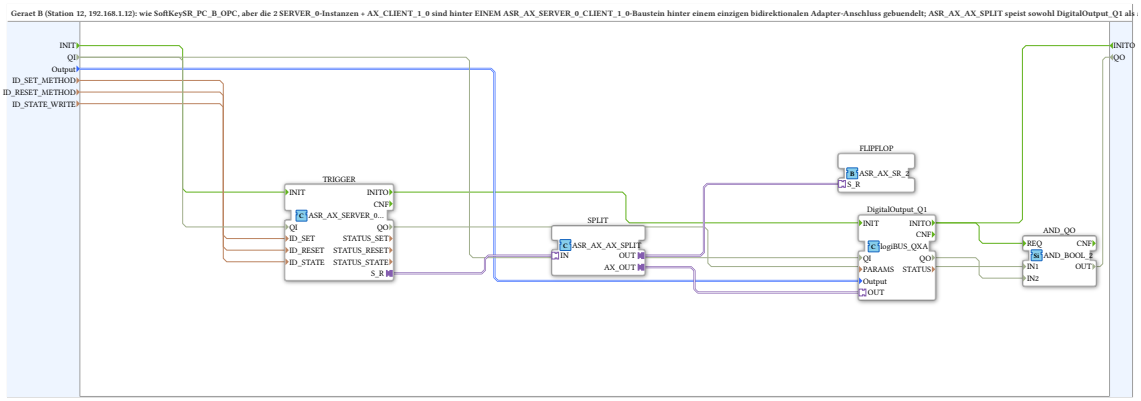
Zusammenfassung

Geraet-B-Seite: empfaengt Set/Reset per RPC, haelt die reine SR-Flipflop-Logik und meldet den Zustand aktiv zurueck.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

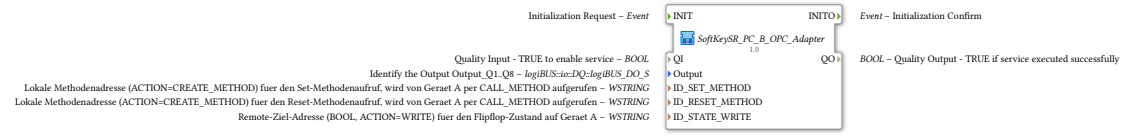
SoftKeySR_PC_B_OPC_Adapter



• – ○ • – ○ • – ○ •

Einleitung

`SoftKeySR_PC_B_OPC_Adapter` ist die adapter-gebuendelte Variante von `SoftKeySR_PC_B_OPC` (Geraet B, Station 12): die 2 `SERVER_0`-Instanzen + `AX_CLIENT_1_0` sind hinter EINEM `ASR_AX_SERVER_0_CLIENT_1_0`-Baustein hinter einem einzigen bidirektionalen Adapter-Anschluss gebuendelt; `ASR_AX_AX_SPLIT` speist sowohl `DigitalOutput_Q1` als auch die reine SR-Flipflop-Logik (`ASR_AX_SR_2`, kein Toggle). Gegenstueck: `SoftKeySR_PC_A_OPC_Adapter`.



Verwendete Funktionsbausteine (FBs)

- **TRIGGER** (`adapter::net::ASR_AX_SERVER_0_CLIENT_1_0`): buendelt Server-Empfang (`ID_SET_METHOD`/`ID_RESET_METHOD`) und Zustands-Rueckmeldung (`ID_STATE_WRITE`).
- **SPLIT** (`adapter::events::bidirectional::ASR_AX_AX_SPLIT`): verzweigt auf physischen Ausgang + Flipflop-Logik.
- **FLIPFLOP** (`adapter::events::bidirectional::ASR_AX_SR_2`): reine SR-Flipflop-Logik als Adapter-Baustein, kein Toggle.
- **DigitalOutput_Q1** (`logiBUS::io::DQ::logiBUS_QXA`): physischer Ausgang.
- **AND_QO** (`iec61131::booleanOperators::AND_BOOL_2`): verknüpft `TRIGGER.QO` und `DigitalOutput_Q1.QO` zum `QO` der SubApp.

Initialisierung und Qualität

- **Ereignis-Eingang `INIT`** (`EInit`) und **Ereignis-Ausgang `INITO`** (`EInit`): `INIT` ⟶ `TRIGGER.INIT`; `TRIGGER.INITO` ⟶ `DigitalOutput_Q1.INIT`; `DigitalOutput_Q1.INITO` ⟶ `INITO` (und ⟶ `AND_QO.REQ`).
- **Dateneingang `QI`** (`BOOL`, Vorgabe `TRUE`) ⟶ `TRIGGER.QI` und `DigitalOutput_Q1.QI`.
- **Datenausgang `QO`** (`BOOL`): `TRIGGER.QO` ⟶ `AND_QO.IN1`; `DigitalOutput_Q1.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`.

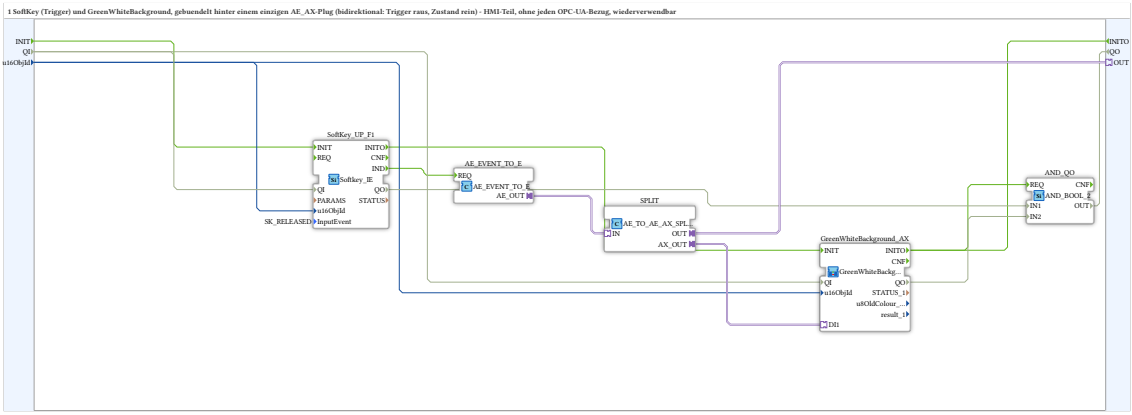
Zusammenfassung

Adapter-gebuendelte Variante von `SoftKeySR_PC_B_OPC`: Netzwerkprotokoll und SR-Flipflop-Logik als wiederverwendbare Adapter-Bausteine.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

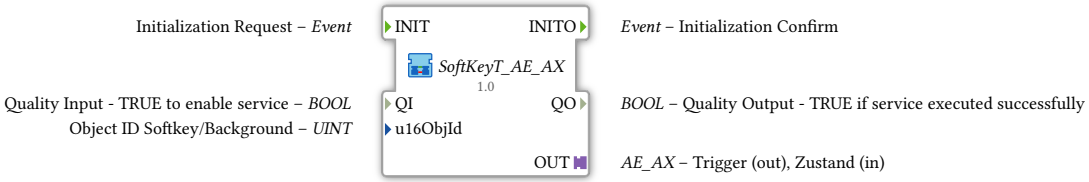
SoftKeyT_AE_AX



• – ○ • – ○ • – ○ •

Einleitung

`SoftKeyT_AE_AX` buendelt 1 SoftKey (Trigger) und ein `GreenWhiteBackground1_AX` hinter einem einzigen `AE_AX`-Plug (bidirektional: Trigger geht raus, Zustand kommt rein). Reiner HMI-Baustein ohne jeden OPC-UA-Bezug - wiederverwendbar in jedem Netzwerk, das einen einzelnen Trigger-SoftKey mit Zustandsanzeige braucht.



Verwendete Funktionsbausteine (FBs)

- **SoftKey_UP_F1** (`isobus::UT::io::Softkey::Softkey_IE`, `InputEvent=SK_RELEASED`): physischer SoftKey.
- **AE_EVENT_TO_E** (`adapter::conversion::unidirectional::AE_EVENT_TO_E`): wandelt den SoftKey-Event in ein AE-Adapter-Ereignis.
- **GreenWhiteBackground_AX** (SubApp, Typ `MyLib::sys::GreenWhiteBackground1_AX`): Statusfarbe direkt auf demselben Objekt.
- **SPLIT** (`adapter::events::bidirectional::AE_TO_AE_AX_SPLIT`): verzweigt den bidirektionalen `AE_AX`-Verkehr auf den externen Plug `OUT` und intern auf das Background-Element.
- **AND_QO** (`iec61131::booleanOperators::AND_BOOL_2`): verknüpft die `QO`-Ausgänge der inneren Bausteine zu `QO`.

Programmablauf und Verbindungen

`SoftKey_UP_F1.IND -> AE_EVENT_TO_E.REQ -> SPLIT.IN -> OUT` (nach aussen) und `SPLIT.AX_OUT -> GreenWhiteBackground_AX.DI1` (Zustandsanzeige).

Initialisierung und Qualität

Schnittstelle (Quality-Muster): Ereignis-Eingang `INIT` und Ereignis-Ausgang `INITO` (je `EInit`), Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`) und Datenausgang `QO` (`BOOL`). Initialisierung: `INIT` ⟶ `SoftKey_UP_F1.INIT`; `SoftKey_UP_F1.INITO` ⟶ `GreenWhiteBackground_AX.INIT`; `GreenWhiteBackground_AX.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`. Qualität: `QI` ⟶ `SoftKey_UP_F1.QI`, `GreenWhiteBackground_AX.QI`; `SoftKey_UP_F1.QO`, `GreenWhiteBackground_AX.QO` ⟶ `AND_QO.IN1` bis `AND_QO.IN2` (in dieser Reihenfolge); `AND_QO.OUT` ⟶ `QO` (`AND_QO`: `iec61131::booleanOperators::AND_BOOL_2`).

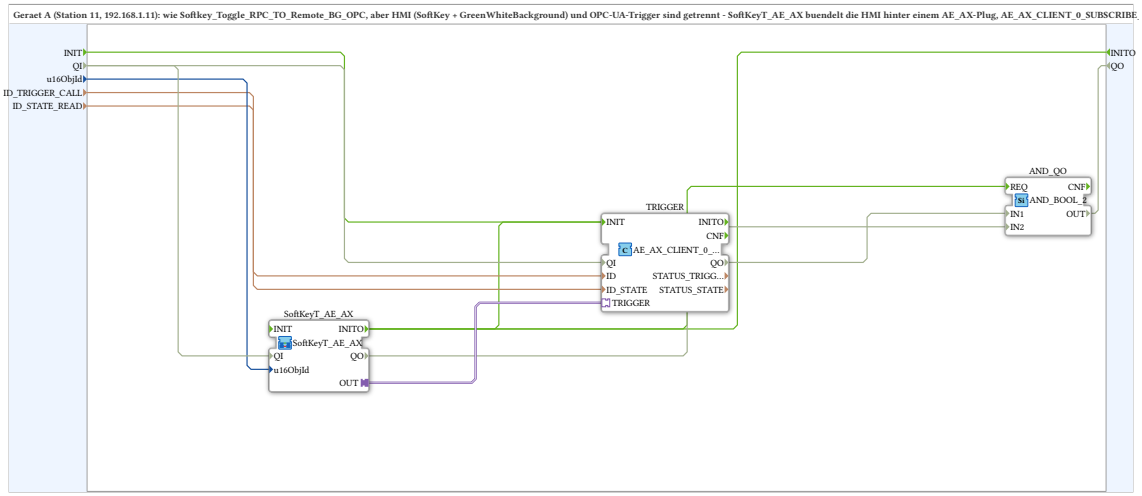
Zusammenfassung

Wiederverwendbarer Ein-Tasten-Trigger-mit-Statusanzeige-Baustein, gebuendelt hinter einem `AE_AX`-Adapter - das HMI-Gegenstueck zu den PC_A/PC_B-OPC-Netzwerkbausteinen wie `SoftKeyT_PC_A_OPC_Adapter`.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

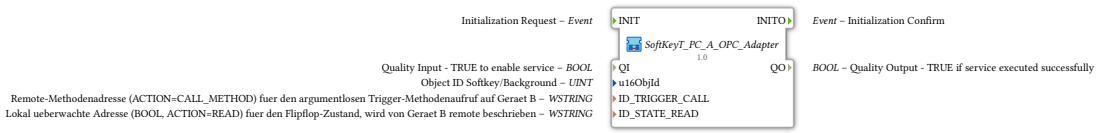
SoftKeyT_PC_A_OPC_Adapter



• – ○ • – ○ • – ○ •

Einleitung

`SoftKeyT_PC_A_OPC_Adapter` ist die adapter-gebuendelte Variante von `Uebung_010d_PC_A_OPC` (Geraet A, Station 11): HMI (SoftKey + `GreenWhiteBackground`) und OPC-UA-Trigger sind getrennt. `SoftKeyT_AE_AX` buendelt die HMI hinter einem `AE_AX`-Plug; `AE_AX_CLIENT_0_SUBSCRIBE_1` buendelt `CLIENT_0` + `AX_SUBSCRIBE_1` hinter einem `AE_AX`-Socket. „SUB style“: das Protokoll steckt weiterhin im `MyLib::sys`-Composite. Gegenstueck: `SoftKeyT_PC_B_OPC_Adapter`.



Schnittstelle

- **Ereignis-Eingang**: `INIT` (`EInit`, Initialization Request)
- **Ereignis-Ausgang**: `INITO` (`EInit`, Initialization Confirm)
- **Daten-Eingänge**: `QI` (`BOOL`, Vorgabe `TRUE`, Quality Input), `u16ObjId`, `ID_TRIGGER_CALL`, `ID_STATE_READ`
- **Daten-Ausgang**: `QO` (`BOOL`, Quality Output)

Verwendete Funktionsbausteine (FBs)

- **SoftKeyT_AE_AX** (SubApp, Typ `MyLib::sys::SoftKeyT_AE_AX`): SoftKey + Statusanzeige, gebuendelt hinter einem `AE_AX`-Plug.
- **TRIGGER** (`adapter::net::AE_AX_CLIENT_0_SUBSCRIBE_1`): buendelt Methodenaufruf (`ID_TRIGGER_CALL`) und Zustands-Abo (`ID_STATE_READ`) hinter einem einzigen `AE_AX`-Socket.
- **AND_QO** (`iec61131::booleanOperators::AND_BOOL_2`): verknüpft `TRIGGER.QO` und `SoftKeyT_AE_AX.QO` per UND zum Baustein-Ausgang `QO`.

Programmablauf und Verbindungen

`SoftKeyT_AE_AX.OUT -> TRIGGER.TRIGGER` (bidirektional: Trigger raus, Zustand zurueck an die HMI-SubApp).

Initialisierungskette: `INIT -> TRIGGER.INIT`, `TRIGGER.INITO -> SoftKeyT_AE_AX.INIT`, `SoftKeyT_AE_AX.INITO -> INITO` und `-> AND_QO.REQ`. `QI -> TRIGGER.QI` und `SoftKeyT_AE_AX.QI`; `TRIGGER.QO -> AND_QO.IN1`, `SoftKeyT_AE_AX.QO -> AND_QO.IN2`, `AND_QO.OUT -> QO`.

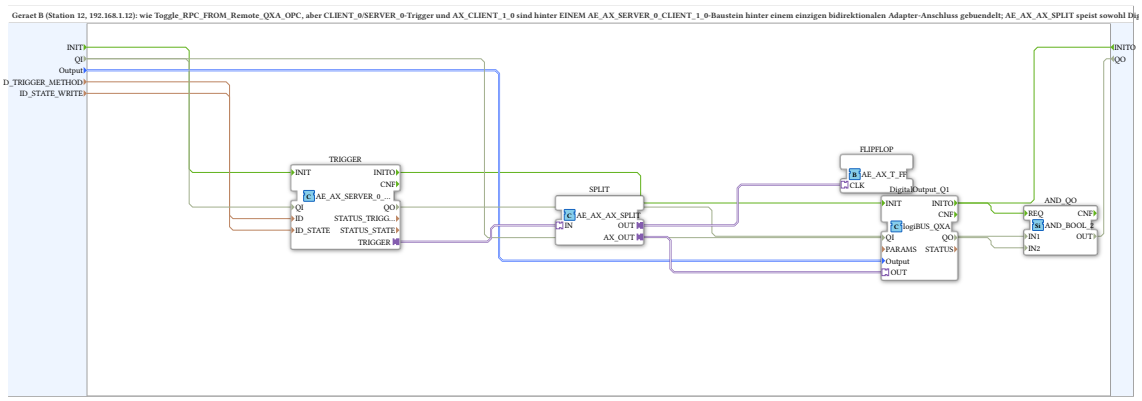
Zusammenfassung

Adapter-gebuendelte Variante von `Uebung_010d_PC_A_OPC`: dieselbe Funktion, aber HMI und Netzwerkprotokoll sind sauber in 2 wiederverwendbare Bausteine getrennt statt in einem einzigen Composite verdrahtet.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

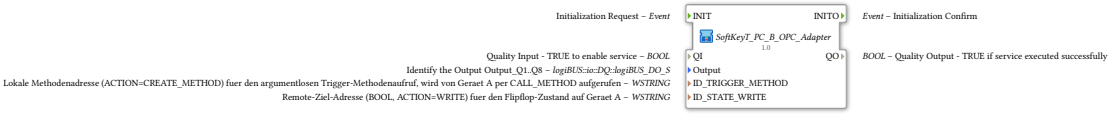
SoftKeyT_PC_B_OPC_Adapter



• – ○ • – ○ • – ○ •

Einleitung

`SoftKeyT_PC_B_OPC_Adapter` ist die adapter-gebuendelte Variante von `Uebung_010d_PC_B_OPC` (Geraet B, Station 12): `CLIENT_0`/`SERVER_0`-Trigger und `AX_CLIENT_1_0` sind hinter EINEM `AE_AX_SERVER_0_CLIENT_1_0`-Baustein hinter einem einzigen bidirektionalen Adapter-Anschluss gebuendelt; `AE_AX_AX_SPLIT` speist sowohl `DigitalOutput_Q1` als auch die Toggle-Flipflop-Logik (`AE_AX_T_FF`). Gegenstueck: `SoftKeyT_PC_A_OPC_Adapter`.



Verwendete Funktionsbausteine (FBs)

- **TRIGGER** (`adapter::net::AE_AX_SERVER_0_CLIENT_1_0`): buendelt Server-Empfang (`ID_TRIGGER_METHOD`) und Zustands-Rueckmeldung (`ID_STATE_WRITE`).
- **SPLIT** (`adapter::events::bidirectional::AE_AX_AX_SPLIT`): verzweigt den Trigger auf physischen Ausgang + Flipflop-Logik.
- **FLIPFLOP** (`adapter::events::bidirectional::AE_AX_T_FF`): Toggle-Flipflop-Logik als Adapter-Baustein.

- **DigitalOutput_Q1** (`logiBUS::io::DQ::logiBUS_QXA`): physischer Ausgang.
- **AND_QO** (`iec61131::booleanOperators::AND_BOOL_2`): verknüpft `TRIGGER.QO` und `DigitalOutput_Q1.QO` zum SubApp-Ausgang `QO`.

Initialisierung und Qualität

- Ereignisse: `INIT` (EInit, Eingang) und `INITO` (EInit, Ausgang, kommt von `DigitalOutput_Q1.INITO`).
- Dateneingang `QI` (BOOL, Vorgabe `TRUE`) und Datenausgang `QO` (BOOL); `TRIGGER` und `DigitalOutput_Q1` tragen kein festes `QI=TRUE` mehr.
- Initialisierung: `INIT` ⟶ `TRIGGER.INIT`, `TRIGGER.INITO` ⟶ `DigitalOutput_Q1.INIT`, `DigitalOutput_Q1.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`.
- Qualität: `QI` ⟶ `TRIGGER.QI` und `DigitalOutput_Q1.QI`; `TRIGGER.QO` ⟶ `AND_QO.IN1`, `DigitalOutput_Q1.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`.

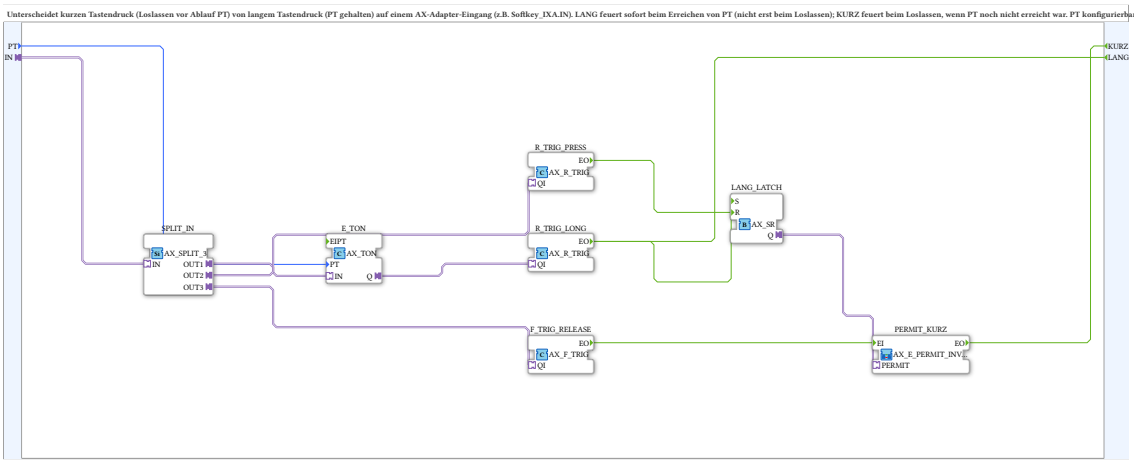
Zusammenfassung

Adapter-gebuendelte Variante von `Uebung_010d_PC_B_OPC`: Netzwerkprotokoll und Flipflop-Logik als wiederverwendbare Adapter-Bausteine statt einzeln verdrahteter FBs.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SoftKey_Kurz_Lang



• — ○ • — ○ • — ○ •

Einleitung

`SoftKey_Kurz_Lang` unterscheidet einen **kurzen** Tastendruck (Loslassen vor Ablauf von `PT`) von einem **langen** Tastendruck (`PT` gehalten) auf einem `AX`-Adapter-Eingang (z. B. `Softkey_IXA.IN`). `LANG` feuert sofort beim Erreichen von `PT` (nicht erst beim Loslassen); `KURZ` feuert beim Loslassen, wenn `PT` noch nicht erreicht war. `PT` ist konfigurierbar (Default 5 s).

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

- **KURZ**: kurzer Druck bestätigt (Loslassen vor Ablauf `PT`)
- **LANG**: langer Druck bestätigt (`PT` erreicht, feuert sofort, nicht erst beim Loslassen)

Daten-Eingänge

- **PT** (TIME): Haltezeit für „lang“, Default T#5s

Daten-Ausgänge

Keine.

Adapter

- **IN** (Socket, adapter::types::unidirectional::AX): Rohzustand der Taste (z. B. Softkey_IXA.IN)

Verwendete Funktionsbausteine (FBs)

- **SPLIT_IN** (adapter::events::unidirectional::AX_SPLIT_3): verteilt IN auf drei Ziele
- **E_TON** (adapter::events::unidirectional::timers::AX_TON): Einschaltverzögerung mit PT
- **R_TRIG_LONG** (adapter::events::unidirectional::AX_R_TRIG): steigende Flanke von E_TON.Q
- **R_TRIG_PRESS** (adapter::events::unidirectional::AX_R_TRIG): steigende Flanke von IN (Tastendruck-Beginn)
- **LANG_LATCH** (adapter::events::unidirectional::AX_SR): merkt, dass LANG in diesem Zyklus schon gefeuert hat
- **F_TRIG_RELEASE** (adapter::events::unidirectional::AX_F_TRIG): fallende Flanke von IN (Loslassen)
- **PERMIT_KURZ** (SubApp MyLib::sys::AX_E_PERMIT_INVERT_1, AX_E_PERMIT_INVERT_1): lässt das Loslass-Ereignis nur durch, wenn die Freigabe FALSE ist

Funktionsweise

- IN ⟶ SPLIT_IN.IN; SPLIT_IN.OUT1 ⟶ E_TON.IN; PT ⟶ E_TON.PT
- SPLIT_IN.OUT2 ⟶ R_TRIG_PRESS.QI; SPLIT_IN.OUT3 ⟶ F_TRIG_RELEASE.QI
- E_TON.Q ⟶ R_TRIG_LONG.QI
- R_TRIG_LONG.EO ⟶ LANG und LANG_LATCH.S
- R_TRIG_PRESS.EO ⟶ LANG_LATCH.R
- F_TRIG_RELEASE.EO ⟶ PERMIT_KURZ.EI; LANG_LATCH.Q ⟶ PERMIT_KURZ.PERMIT; PERMIT_KURZ.EO ⟶ KURZ

E_TON läuft, solange IN gehalten wird. Die steigende Flanke von E_TON.Q (Zeitpunkt, an dem PT erreicht ist, während die Taste noch gehalten wird) löst LANG sofort aus und setzt LANG_LATCH. Beim Loslassen erkennt F_TRIG_RELEASE die fallende Flanke; PERMIT_KURZ lässt sie nur durch, wenn LANG_LATCH.Q FALSE ist, LANG also in diesem Druckzyklus noch nicht gefeuert wurde. Beim nächsten Tastendruck-Beginn (R_TRIG_PRESS) wird LANG_LATCH zurückgesetzt.

Technische Besonderheiten

- **Bugfix gegenüber Version 1.0:** Die Prüfung darf nicht direkt auf E_TON.Q erfolgen. E_TON.Q wird durch denselben IN-Flankenwechsel (Loslassen), der auch F_TRIG_RELEASE auslöst, auf FALSE zurückgesetzt (AX_TON: kein IN ⟶ kein Q). Da beide Ziele vom selben IN-Ereignis gespeist werden, ist die Ausführungsreihenfolge zwischen ihnen nicht garantiert; E_TON.Q wäre bei der Prüfung praktisch immer schon FALSE, und KURZ hätte auch nach bereits gefeuertem LANG zusätzlich gefeuert.
- LANG_LATCH ist vom Reset-Timing komplett entkoppelt (gesetzt während des Haltens, zurückgesetzt erst beim nächsten Tastendruck) und macht die Prüfung robust.
- PT ist als Parameter konfigurierbar.

Anwendungsszenarien

- Taste mit Doppelbelegung: kurzer Druck für eine Funktion, langer Druck (z. B. 5 s) für eine andere, zusammen mit Softkey_Web_IXA für Tasten, die auch per Web-Client bedient werden sollen

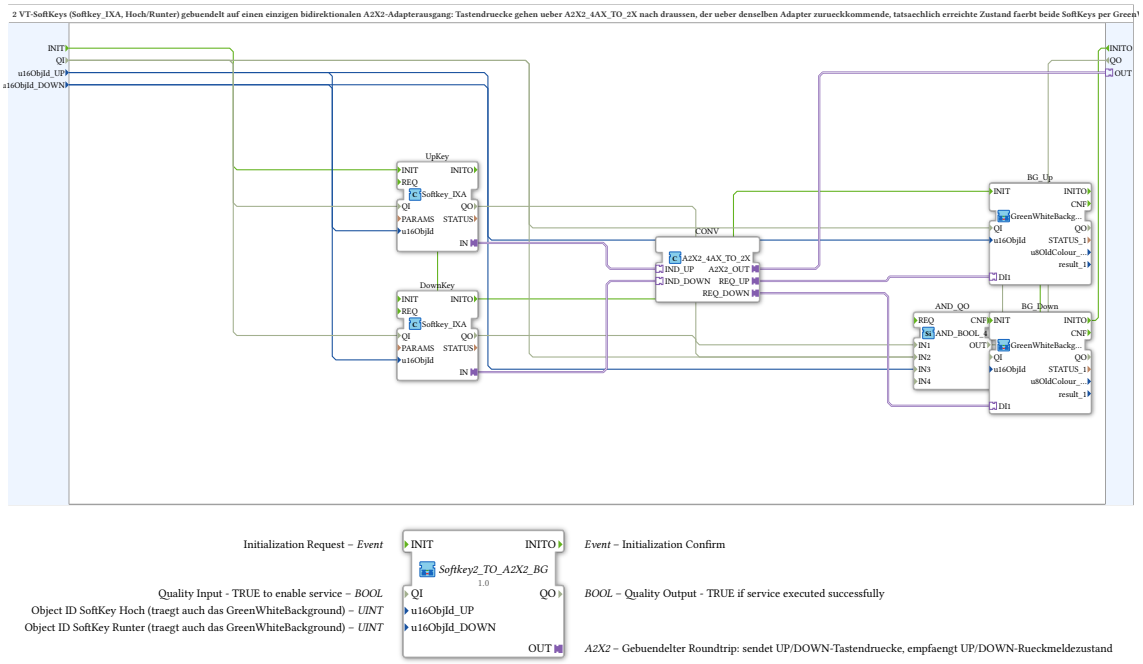
Zusammenfassung

SoftKey_Kurz_Lang liefert aus einem AX-Tastenzustand die getrennten Ereignisse KURZ und LANG mit konfigurierbarer Haltezeit.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Softkey2_TO_A2X2_BG



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock **Softkey2_TO_A2X2_BG** ist ein Composite-SubApp, der zwei virtuelle Softkeys (Hoch/Runter) eines ISOBUS-Terminals über einen einzigen bidirektionalen A2X2-Adapter nach außen führt. Er bündelt die Tastendrücke beider Softkeys und sendet sie über den Adapter. Gleichzeitig wertet er den über denselben Adapter zurückkommenden Ist-Zustand aus und färbt die Softkeys entsprechend grün oder weiß (GreenWhiteBackground). Die Bündelung erfolgt im Composite selbst, nicht in der Geräteressource.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; startet die Init-Kette bei `UpKey.INIT`

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung; kommt von `BG_Down.INITO`, dem Ende der Init-Kette

Daten-Eingänge

Name	Datentyp	Initialwert	Kommentar
`QI`	`BOOL`	`TRUE`	Quality Input, schaltet den Dienst frei; wird an `UpKey`, `DownKey`, `BG_Up` und `BG_Down` weitergereicht.
`u16ObjId_UP`	`UINT`	`ID_NULL`	Objekt-ID des SoftKeys „Hoch“ (trägt auch das Green-White-Background).
`u16ObjId_DOWN`	`UINT`	`ID_NULL`	Objekt-ID des SoftKeys „Runter“ (trägt auch das Green-White-Background).

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output, UND-Verknüpfung (`AND_QO`, `AND_BOOL_4`) der `QO`-Ausgänge von `UpKey`, `DownKey`, `BG_Up` und `BG_Down`

Adapter

Name	Typ	Richtung	Kommentar
OUT	adapter::types::bidirectional::A2X2	bidirektional	Gebündelter Roundtrip: sendet UP/DOWN-Tastendrücke, empfängt UP/DOWN-Rückmeldezustand.

Funktionsweise

Der SubApp enthält zwei Instanzen des Softkey-Bausteins `Softkey_IXA` (`UpKey` und `DownKey`), die jeweils einen eigenen Tastendruck liefern. Diese beiden Einzelkanäle werden über den Adapterkonverter `A2X2_4AX_TO_2X` zusammengeführt und als ein gemeinsamer A2X2-Datenstrom über den Adapter `OUT` ausgegeben.

Parallel dazu wird der vom Adapter zurückkommende Zustand (der tatsächlich erreichte Softkey-Zustand) durch denselben Konverter in zwei separate Rückmeldesignale `REQ_UP` und `REQ_DOWN` aufgeteilt. Diese werden an die beiden SubApps `GreenWhiteBackground1_AX` übergeben, die die Objekt-IDs der Softkeys mit der entsprechenden Hintergrundfarbe (grün = aktiv, weiß = inaktiv) belegen.

Initialisierung und Qualität: `INIT` → `UpKey.INIT`, `UpKey.INITO` → `DownKey.INIT`, `DownKey.INITO` → `BG_Up.INIT`, `BG_Up.INITO` → `BG_Down.INIT`, `BG_Down.INITO` → `INITO` und `AND_QO.REQ`. `QI` wird an `UpKey`, `DownKey`, `BG_Up` und `BG_Down` weitergereicht; deren `QO`-Ausgänge gehen in dieser Reihenfolge auf `AND_QO.IN1` bis `AND_QO.IN4` (`iec61131::booleanOperators::AND_BOOL_4`), `AND_QO.OUT` → `QO`.

Die Verbindung der Objekt-IDs von den Eingängen zu den beiden Softkeys sowie zu den Hintergrund-SubApps erfolgt intern über Datenverbindungen.

Technische Besonderheiten

- **Nur Init-Ereignisse:** Die Signalverarbeitung arbeitet daten- und adapterbasiert; als Ereignisse besitzt der Baustein nur `INIT` und `INITO` für die Init-Kette.
- **Bidirektionale Kommunikation:** Der A2X2-Adapter ermöglicht gleichzeitiges Senden (Tastendrücke) und Empfangen (Rückmeldung) über dieselbe Verbindung.
- **Interner Konverter `A2X2_4AX_TO_2X`:** Bündelt zwei Einzelkanäle zu einem A2X2-Datenstrom und teilt den eingehenden A2X2-Strom wieder in zwei getrennte Signale auf.
- **Wiederverwendbare SubApps:** Für die Hintergrundfärbung wird die SubApp `GreenWhiteBackground1_AX` aus der Bibliothek `MyLib::sys` eingesetzt.
- **Freigabe über `QI`:** Die Softkey-Instanzen und die Hintergrund-SubApps sind nicht mehr fest mit `QI = TRUE` parametriert, sondern erhalten `QI` vom Baustein-Eingang `QI` (Initialwert `TRUE`).

Zustandsübersicht

Der Baustein selbst besitzt keine eigenen Zustände. Die internen Softkey-Bausteine `Softkey_IXA` verfügen jedoch über eine Zustandslogik, die über die Rückmeldung den tatsächlichen Softkey-Zustand (z. B. gedrückt/aktiv) abbildet. Diese Zustände werden über die Hintergrundfärbung sichtbar gemacht.

Anwendungsszenarien

- **ISOBUS-Terminals mit Softkeys:** Zwei Softkeys (z. B. „Hoch“ und „Runter“) sollen über einen einzigen A2X2-Adapter mit einer externen Steuerung kommunizieren.
- **Visualisierung von Rückmeldungen:** Der tatsächlich erreichte Zustand der Softkeys (z. B. aktiv/inaktiv) wird durch Grün/Weiß-Hintergrund direkt auf dem Terminal angezeigt.
- **Platzsparende Verdrahtung:** Durch die Bündelung im Composite kann die Anzahl der Adapter-Verbindungen reduziert werden, ohne die Logik in die Ressource zu verlagern.

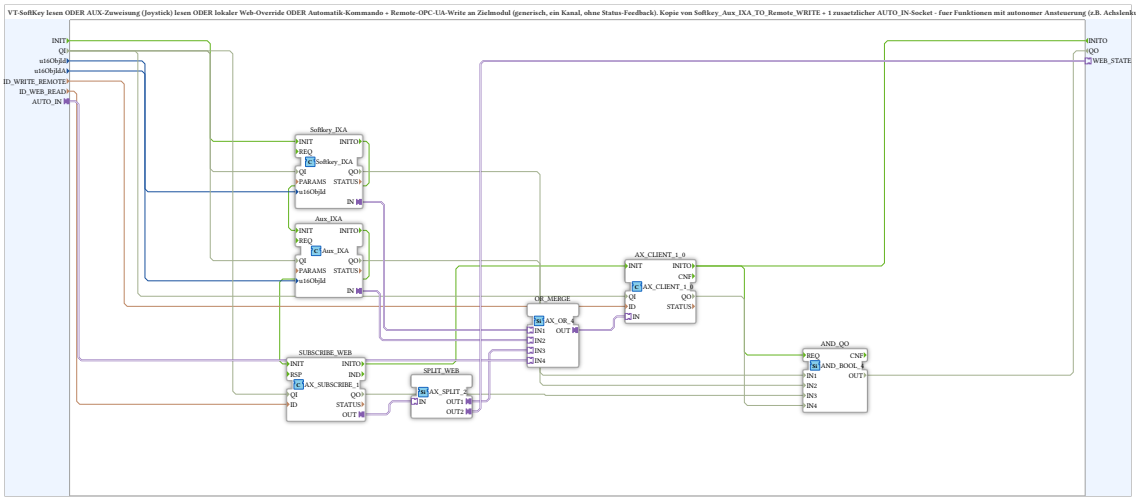
Vergleich mit ähnlichen Bausteinen

- **Ohne Bündelung:** Ein direkter Anschluss jedes Softkeys an einen eigenen A2X2-Adapter würde zwei Adapter erfordern. Der vorliegende Baustein spart eine Adapterverbindung.
- **Mit mehr als zwei Softkeys:** Für mehr Softkeys könnte das Prinzip erweitert werden, indem der Konverter entsprechend mehr Kanäle zusammenfasst (z. B. `A2X2_6AX_TO_3X`). Der Baustein ist speziell für zwei Softkeys ausgelegt.
- **SubApp statt Ressource:** Das Bündeln erfolgt im Composite, nicht in der Geräteressource – dies erleichtert die Wiederverwendung und Portierung.

Fazit

`Softkey2_TO_A2X2_BG` ist ein kompaktes, wiederverwendbares Composite, das zwei Softkeys über einen einzigen bidirektionalen A2X2-Adapter anbindet und gleichzeitig die Rückmeldung visualisiert. Durch die klare Trennung von Tastendruck und Zustandsanzeige sowie die Verwendung standardisierter Bausteine eignet er sich gut für ISOBUS-Bedienoberflächen.

Softkey_Aux_Auto_IXA_TO_Remote_WRITE



• – ○ • – ○ • – ○ •

Einleitung

`Softkey_Aux_Auto_IXA_TO_Remote_WRITE` liest einen **VT-SoftKey**, eine **AUX-Zuweisung (Joystick)**, einen **lokalen Web-Override** oder ein **Automatik-Kommando** und schreibt das Ergebnis per Remote-OPC-UA-Write an ein Zielmodul (generisch, ein Kanal, ohne Status-Feedback). Der Baustein ist eine Kopie von `Softkey_Aux_IXA_TO_Remote_WRITE` mit einem zusätzlichen Socket `AUTO_IN` für Funktionen mit autonomer Ansteuerung (z. B. Achslenkung-Mittelpunkt-Automatik).

Es gibt weiterhin genau **einen** Publisher pro Kanal auf `ID_WRITE_REMOTE`, auch wenn vier statt drei Quellen zusammengeführt werden – kein Self-Latch-Risiko. Für Status-Feedback/Hintergrundfarbe siehe die vorverdrahtete Kombination `Softkey_Aux_Auto_IXA_TO_Remote_WRITE_BG_OPC`.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Softkey_Aux_Auto_IXA_TO_Remote_WRITE

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **Softkey_IXA**: `isobus::UT::io::Softkey::Softkey_IXA` – SoftKey-Pegel
  - **Aux_IXA**: `isobus::UT::io::Auxiliary::IN::Aux_IXA` – AUX-Pegel
  - **SUBSCRIBE_WEB**: `adapter::net::AX_SUBSCRIBE_1` – lokaler Subscribe für einen Web-Client
  - **SPLIT_WEB**: `adapter::events::unidirectional::AX_SPLIT_2` – verteilt den Web-Zustand auf `OR_MERGE` und `WEB_STATE`
  - **OR_MERGE**: `adapter::booleanOperators::AX_OR_4` – ODER-Verknüpfung der vier Quellen
  - **AX_CLIENT_1_0**: `adapter::net::AX_CLIENT_1_0` – Remote-Write an das Zielmodul
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_4` – verknüpft die `QO` von `Softkey_IXA`, `Aux_IXA`, `SUBSCRIBE_WEB` und `AX_CLIENT_1_0`
- **Import**: `ID_NULL`

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; geht an `Softkey_IXA.INIT`

Ereignis-Ausgänge

- **INITO** (EInit): Initialisierung abgeschlossen; kommt von AX_CLIENT_1_0.INITO, dem Ende der Kette

Die Schaltereignisse selbst laufen weiterhin über die Adapter.

Daten-Eingänge

- **QI** (BOOL, Vorgabe TRUE): Quality Input - TRUE aktiviert den Dienst; geht an Softkey_IXA, Aux_IXA, SUBSCRIBE_WEB und AX_CLIENT_1_0
- **u16ObjId** (UINT, Vorgabe ID_NULL): Object-ID SoftKey (VT)
- **u16ObjIdA** (UINT, Vorgabe ID_NULL): Object-ID AuxFunction2 (VT)
- **ID_WRITE_REMOTE** (WSTRING): Remote-Write-Adresse zum Zielmodul (Befehl, ACTION=WRITE, CLIENT)
- **ID_WEB_READ** (WSTRING): Lokale Subscribe-Adresse für einen Web-Client (z. B. vt-ui-mirror) auf demselben Modul, ODER-verknüpft mit SoftKey und AUX

Daten-Ausgänge

- **QO** (BOOL): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde (AND der QO von Softkey_IXA, Aux_IXA, SUBSCRIBE_WEB und AX_CLIENT_1_0)

Adapter

- **AUTO_IN** (Socket, adapter::types::unidirectional::AX): Autonomes Kommando (z. B. Achslenkung-Mittelpunkt-Automatik), ODER-verknüpft mit SoftKey/Aux/Web
- **WEB_STATE** (Plug, adapter::types::unidirectional::AX): Web-Zustand (= SUBSCRIBE_WEB.OUT) nach außen durchgereicht, damit andere Composite-Instanzen denselben Wert per Datenverbindung nutzen können, statt selbst nochmals lokal zu abonnieren (nur ein lokaler OPC-UA-Subscriber pro Knoten zulässig)

Programmablauf und Verbindungen

- u16ObjId ⟶ Softkey_IXA.u16ObjId; Softkey_IXA.IN ⟶ OR_MERGE.IN1
- u16ObjIdA ⟶ Aux_IXA.u16ObjId; Aux_IXA.IN ⟶ OR_MERGE.IN2
- ID_WEB_READ ⟶ SUBSCRIBE_WEB.ID; SUBSCRIBE_WEB.OUT ⟶ SPLIT_WEB.IN
- SPLIT_WEB.OUT1 ⟶ OR_MERGE.IN3; SPLIT_WEB.OUT2 ⟶ WEB_STATE
- AUTO_IN ⟶ OR_MERGE.IN4
- OR_MERGE.OUT ⟶ AX_CLIENT_1_0.IN; ID_WRITE_REMOTE ⟶ AX_CLIENT_1_0.ID
- Initialisierung: INIT ⟶ Softkey_IXA.INIT, Softkey_IXA.INITO ⟶ Aux_IXA.INIT, Aux_IXA.INITO ⟶ SUBSCRIBE_WEB.INIT, SUBSCRIBE_WEB.INITO ⟶ AX_CLIENT_1_0.INIT, AX_CLIENT_1_0.INITO ⟶ INITO und ⟶ AND_QO.REQ
- Qualität: QI ⟶ Softkey_IXA.QI, Aux_IXA.QI, SUBSCRIBE_WEB.QI und AX_CLIENT_1_0.QI; deren QO ⟶ AND_QO.IN1..IN4; AND_QO.OUT ⟶ QO

Technische Besonderheiten

- Version 1.2 (2026-09-16): SPLIT_WEB (AX_SPLIT_2) zwischengesetzt, da Adapter-Plugs strikt 1:1 sind (kein Fan-out wie bei Daten-/Ereignisverbindungen); zwei Ziele am selben Plug führten zu einem FORTE-Bootfehler (INVALID_STATE beim zweiten CREATE Connection)
- Version 1.1 (2026-09-16): Plug WEB_STATE ergänzt; behebt den Doppel-Self-Subscribe auf ID_WEB_READ (FORTE: „Cannot add another one“)
- Version 1.0 (2026-09-11): Kopie von Softkey_Aux_IXA_TO_Remote_WRITE mit AX_OR_4 statt AX_OR_3; nur für Kanäle mit Automatik-Bedarf verwenden, alle anderen Verwender bleiben auf dem Originaltyp
- Kein Status-Feedback und keine Hintergrundfarbe in diesem Baustein

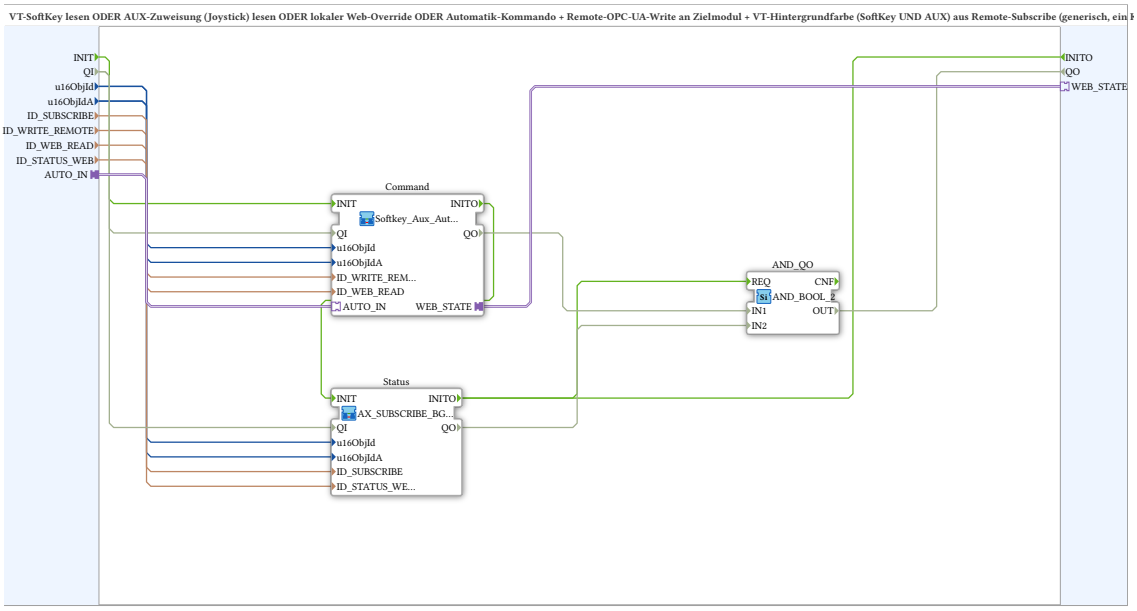
Anwendungsszenarien

- Remote-Kommando, das gleichzeitig per SoftKey, Joystick, Web-Client und autonomer Automatik ausgelöst werden kann (z. B. Achslenkung-Mittelpunkt-Automatik)
- Quelle des WEB_STATE für Softkey_Aux_Web_IE (Override-Erkennung)

Zusammenfassung

Softkey_Aux_Auto_IXA_TO_Remote_WRITE verknüpft SoftKey, AUX, Web-Override und Automatik per ODER und schreibt das Ergebnis über genau einen Publisher an das Zielmodul.

Softkey_Aux_Auto_IXA_TO_Remote_WRITE_BG_OPC



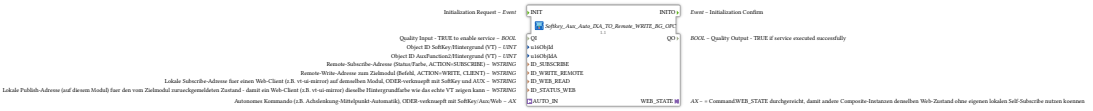
• – ○ • – ○ • – ○ •

Einleitung

`Softkey_Aux_Auto_IXA_TO_Remote_WRITE_BG_OPC` ist ein dünnwandiger Wrapper (ein Kanal), der zwei SubApps nebeneinanderstellt:

- **Command** (`Softkey_Aux_Auto_IXA_TO_Remote_WRITE`): VT-SoftKey lesen ODER AUX-Zuweisung (Joystick) lesen ODER lokaler Web-Override ODER Automatik-Kommando ⟶ Remote-OPC-UA-Write an das Zielmodul.
- **Status** (`AX_SUBSCRIBE_BG3_WEB_OPC`): VT-Hintergrundfarbe (SoftKey UND AUX) aus einem Remote-Subscribe sowie lokaler Web-Republish (unverändert).

Der Baustein ist eine Kopie von `Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC`, die als Kommandohälfte `Softkey_Aux_Auto_IXA_TO_Remote_WRITE` statt `Softkey_Aux_IXA_TO_Remote_WRITE` verwendet. Er ist nur für Kanäle mit Automatik-Bedarf gedacht (aktuell: Achslenkung links/rechts, Hangausgleich links/rechts); für alle anderen bleibt `Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC` zuständig.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung

Die Nutzereignisse kommen weiterhin über den Adapter-Socket.

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung (von `Status.INITO`)

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input – TRUE gibt den Dienst frei
- **u16ObjId** (`UINT`): Object-ID SoftKey/Hintergrund (VT)
- **u16ObjIdA** (`UINT`): Object-ID AuxFunction2/Hintergrund (VT)
- **ID_SUBSCRIBE** (`WSTRING`): Remote-Subscribe-Adresse (Status/Farbe, `ACTION=SUBSCRIBE`)
- **ID_WRITE_REMOTE** (`WSTRING`): Remote-Write-Adresse zum Zielmodul (Befehl, `ACTION=WRITE`, CLIENT)

- **ID_WEB_READ** (WSTRING): Lokale Subscribe-Adresse für einen Web-Client (z. B. vt-ui-mirror) auf demselben Modul, ODER-verknüpft mit SoftKey und AUX
- **ID_STATUS_WEB** (WSTRING): Lokale Publish-Adresse (auf diesem Modul) für den vom Zielmodul zurückgemeldeten Zustand, damit ein Web-Client dieselbe Hintergrundfarbe wie das echte VT zeigen kann

Daten-Ausgänge

- **QO** (BOOL): Quality Output – UND aus Command.QO und Status.QO

Adapter

- **AUTO_IN** (Socket, adapter::types::unidirectional::AX): Autonomes Kommando (z. B. Achslenkung-Mittelpunkt-Automatik), ODER-verknüpft mit SoftKey/Aux/Web
- **WEB_STATE** (Plug, adapter::types::unidirectional::AX): Command.WEB_STATE durchgereicht, damit andere Composite-Instanzen denselben Web-Zustand ohne eigenen lokalen Self-Subscribe nutzen können

Verwendete Funktionsbausteine (FBs)

- **Command**: SubApp MyLib::sys::Softkey_Aux_Auto_IXA_TO_Remote_WRITE
- **Status**: SubApp MyLib::sys::AX_SUBSCRIBE_BG3_WEB_OPC
- **AND_QO** (iec61131::booleanOperators::AND_BOOL_2): verknüpft Command.QO und Status.QO zum QO

Programmablauf und Verbindungen

- u16ObjId, u16ObjIdA ⟶ Command und Status
- ID_WRITE_REMOTE, ID_WEB_READ ⟶ Command
- ID_SUBSCRIBE, ID_STATUS_WEB ⟶ Status
- AUTO_IN ⟶ Command.AUTO_IN
- Command.WEB_STATE ⟶ WEB_STATE
- INIT ⟶ Command.INIT; Command.INITO ⟶ Status.INIT; Status.INITO ⟶ INITO (und ⟶ AND_QO.REQ)
- QI ⟶ Command.QI und Status.QI
- Command.QO ⟶ AND_QO.IN1; Status.QO ⟶ AND_QO.IN2; AND_QO.OUT ⟶ QO

Command und Status sind nicht direkt miteinander verbunden; die Rückmeldung läuft über den Remote-Subscribe.

Technische Besonderheiten

- WEB_STATE (Version 1.1 von Softkey_Aux_Auto_IXA_TO_Remote_WRITE) ist ein durchgereichter Fan-out-Plug.
- Nur für Kanäle mit Automatik-Bedarf verwenden.

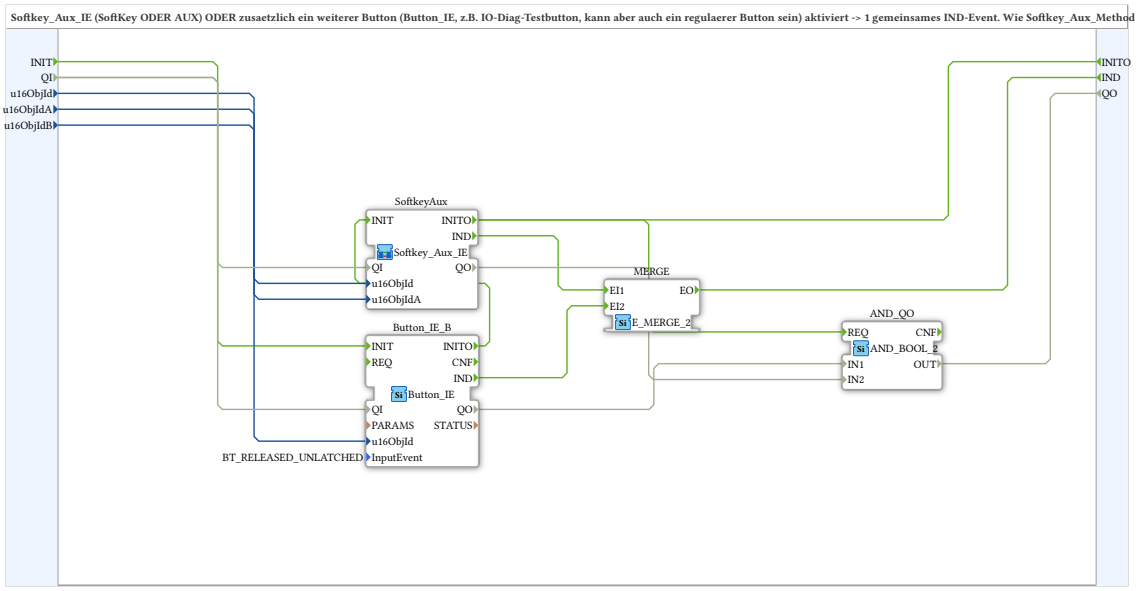
Anwendungsszenarien

- VT-Taste/AUX/Web/Automatik, die einen Befehl per OPC-UA an ein Zielmodul schreibt, mit Hintergrundfarbe aus dem zurückgemeldeten Zustand (z. B. Achslenkung, Hangausgleich)

Fazit

Softkey_Aux_Auto_IXA_TO_Remote_WRITE_BG_OPC kombiniert Kommando (inkl. Automatik) und Statusrückmeldung eines Kanals in einer SubApp.

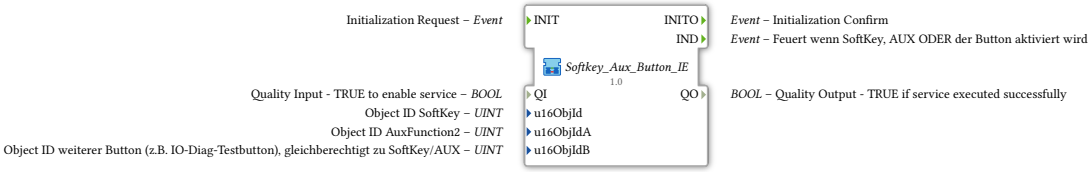
Softkey_Aux_Button_IE



• – ○ • – ○ • – ○ •

Einleitung

`Softkey_Aux_Button_IE` fasst `Softkey_Aux_IE` (SoftKey ODER AUX) und zusätzlich einen weiteren Button (`Button_IE`, z. B. den IO-Diag-Testbutton, es kann aber auch ein regulärer Button sein) zu **einem** gemeinsamen `IND`-Ereignis zusammen. Er entspricht `Softkey_Aux_Method_Button_IE`, aber ohne die lokale OPC-UA-Methode, und ist für Fälle gedacht, die nur SoftKey, AUX und Button gleichberechtigt brauchen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialization Request

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialization Confirm
- **IND**: feuert, wenn SoftKey, AUX **oder** der Button aktiviert wird

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input, TRUE aktiviert den Dienst
- **u16ObjId** (UINT): Object ID des SoftKeys
- **u16ObjIdA** (UINT): Object ID der AuxFunction2
- **u16ObjIdB** (UINT): Object ID eines weiteren Buttons (z. B. IO-Diag-Testbutton), gleichberechtigt zu SoftKey/AUX

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output, UND-Verknüpfung (`AND_QO`) der `QO`-Ausgänge von `Button_IE_B` und `SoftkeyAux`

Adapter

Es werden keine Adapter verwendet.

Funktionsweise

Das Netzwerk enthält vier Bausteine:

- `SoftkeyAux` (`MyLib::sys::Softkey_Aux_IE`): `u16ObjId` ⟶ `SoftkeyAux.u16ObjId`, `u16ObjIdA` ⟶ `SoftkeyAux.u16ObjIdA`; sein `IND` geht an `MERGE.EI1`.
- `Button_IE_B` (`isobus::UT::io::Button::Button_IE`, `InputEvent = BT_RELEASED_UNLATCHED`): `u16ObjIdB` ⟶ `Button_IE_B.u16ObjId`; sein `IND` geht an `MERGE.EI2`.
- `MERGE` (`iec61499::events::E_MERGE_2`): `MERGE.EO` ⟶ `IND`.
- `AND_QO` (`iec61131::booleanOperators::AND_BOOL_2`): verknüpft `Button_IE_B.QO` (`IN1`) und `SoftkeyAux.QO` (`IN2`) zu `QO`.

Initialisierung: `INIT` ⟶ `Button_IE_B.INIT`; `Button_IE_B.INITO` ⟶ `SoftkeyAux.INIT`; `SoftkeyAux.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`. Qualität: `QI` ⟶ `Button_IE_B.QI` und `SoftkeyAux.QI`; `AND_QO.OUT` ⟶ `QO`.

Technische Besonderheiten

- Der zusätzliche Button löst beim Loslassen (`BT_RELEASED_UNLATCHED`) aus
- Herausgelöst aus `Baender_Individual_Toggle.SUB`, wo dieselbe Kombination (`Softkey_Aux_IE` + `Button_IE` + `E_MERGE_2`) dreimal für Igel, Elevator und Querband verdrahtet war
- Zustandslos

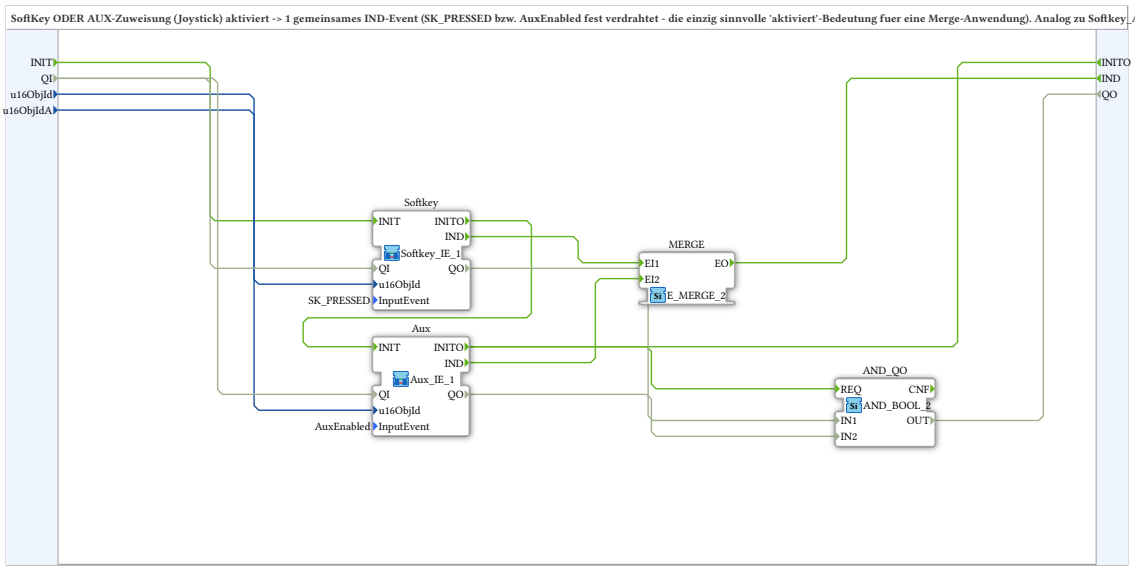
Anwendungsszenarien

- Ein Ausgang, der gleichberechtigt per SoftKey, Joystick-Zuweisung und zusätzlichem Button (z. B. IO-Diag-Testbutton) ausgelöst werden soll

Fazit

`Softkey_Aux_Button_IE` vereinigt SoftKey, AUX und einen weiteren Button zu einem einzigen `IND`-Ereignis.

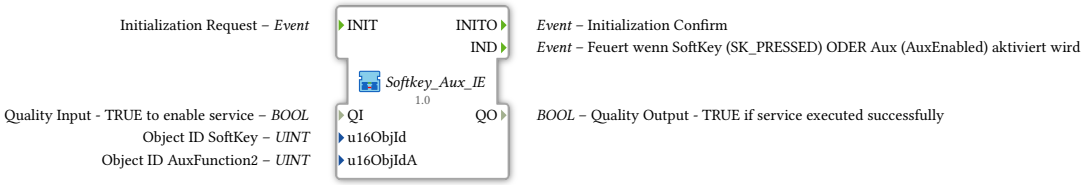
Softkey_Aux_IE



• – ○ • – ○ • – ○ •

Einleitung

`Softkey_Aux_IE` löst ein gemeinsames `IND`-Ereignis aus, wenn ein **SoftKey** ODER eine **AUX-Zuweisung (Joystick)** aktiviert wird. `SK_PRESSED` bzw. `AuxEnabled` sind fest verdrahtet, da dies die einzig sinnvolle „aktiviert"-Bedeutung für eine Merge-Anwendung ist. Der Baustein ist analog zu `Softkey_Aux_IXA_TO_Remote_WRITE` (Dauerzustand-ODER), aber für den diskreten Event-Kanal. Er kollidiert daher nicht mit einer bereits bestehenden `Softkey_IXA`/`Aux_IXA`-Dauerzustand-Abonnierung auf demselben Objekt (zwei verschiedene Subscription-Typen).



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialization Request

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialization Confirm
- **IND**: feuert, wenn der SoftKey (`SK_PRESSED`) ODER Aux (`AuxEnabled`) aktiviert wird

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input - TRUE, um den Dienst freizugeben
- **u16ObjId** (`UINT`): Object ID SoftKey
- **u16ObjIdA** (`UINT`): Object ID AuxFunction2

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde (UND von `Softkey.QO` und `Aux.QO`)

Adapter

Keine.

Verwendete Funktionsbausteine (FBs)

- **`Softkey`** (SubApp `MyLib::sys::Softkey_IE_1`, Softkey_IE_1): Parameter `InputEvent = SK_PRESSED`
- **`Aux`** (SubApp `MyLib::sys::Aux_IE_1`, Aux_IE_1): Parameter `InputEvent = AuxEnabled`
- **`MERGE`** (`iec61499::events::E_MERGE_2`): führt beide Ereignisse zusammen
- **`AND_QO`** (`iec61131::booleanOperators::AND_BOOL_2`): verknüpft `Softkey.QO` und `Aux.QO` per UND zu `QO`

Funktionsweise

- `u16ObjId` ⟶ `Softkey.u16ObjId`; `u16ObjIdA` ⟶ `Aux.u16ObjId`
- `Softkey.IND` ⟶ `MERGE.EI1`; `Aux.IND` ⟶ `MERGE.EI2`
- `MERGE.EO` ⟶ `IND`
- Initialisierungskette: `INIT` ⟶ `Softkey.INIT`, `Softkey.INITO` ⟶ `Aux.INIT`, `Aux.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`
- `QI` ⟶ `Softkey.QI` und `Aux.QI`
- `Softkey.QO` ⟶ `AND_QO.IN1`, `Aux.QO` ⟶ `AND_QO.IN2`, `AND_QO.OUT` ⟶ `QO`

Technische Besonderheiten

- Importiert `isobus::UT::io::Softkey::SoftKeyActivationCode::SK_PRESSED` und `isobus::UT::io::Auxiliary::IN::AuxiliaryState::AuxEnabled`
- Arbeitet über den diskreten Event-Kanal, nicht über den Dauerzustand

Anwendungsszenarien

- Ein Ereignis aus SoftKey oder Joystick-Zuweisung, z. B. für Toggle- oder Trigger-Funktionen

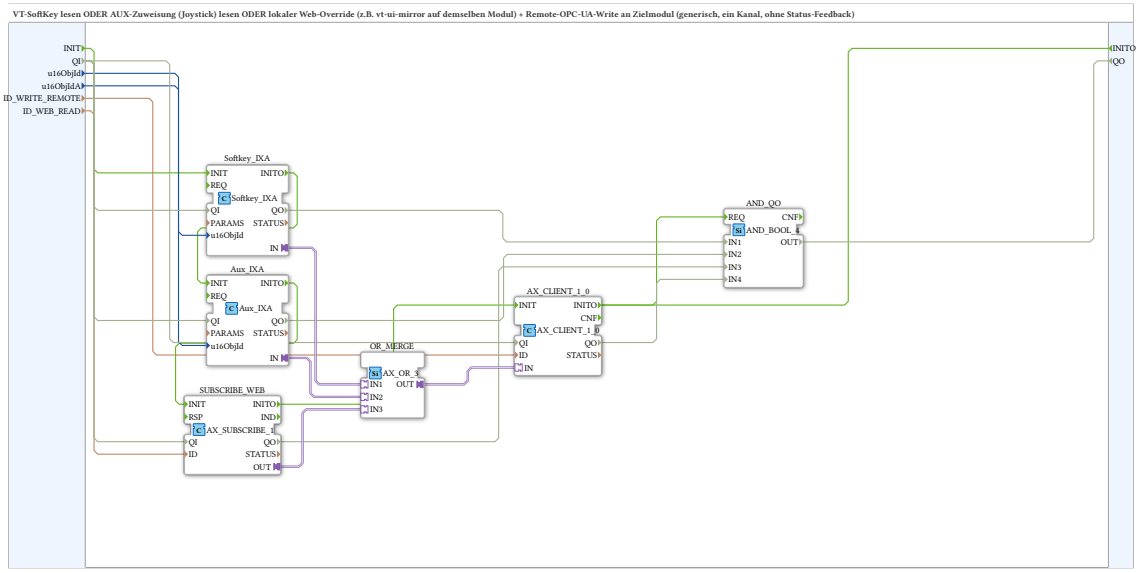
Zusammenfassung

`Softkey_Aux_IE` vereint SoftKey- und AUX-Aktivierung zu einem gemeinsamen `IND`-Ereignis.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

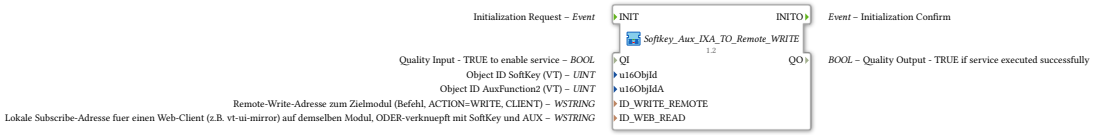
Softkey_Aux_IXA_TO_Remote_WRITE



• – ○ • – ○ • – ○ •

Einleitung

`Softkey_Aux_IXA_TO_Remote_WRITE` ist die Kommando-Hälfte von `Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC`, herausgelöst als eigenständiger Baustein: VT-SoftKey, AUX-Zuweisung (Joystick) und ein lokaler Web-Override (z. B. vt-ui-mirror auf demselben Modul) werden per ODER zusammengeführt und per Remote-Write an ein Zielmodul geschrieben — ohne Status-Feedback.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Softkey_Aux_IXA_TO_Remote_WRITE

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **Softkey_IXA**: `isobus::UT::io::Softkey::Softkey_IXA` — VT-SoftKey-Adapter (`u16ObjId`).
  - **Aux_IXA**: `isobus::UT::io::Auxiliary::IN::Aux_IXA` — AUX-Zuweisung/Joystick-Adapter (`u16ObjIdA`).
  - **SUBSCRIBE_WEB**: `adapter::net::AX_SUBSCRIBE_1` — lokaler Web-Subscribe (`ID_WEB_READ`), z. B. von einem vt-ui-mirror auf demselben Modul.
  - **OR_MERGE**: `adapter::booleanOperators::AX_OR_3` — führt SoftKey, Aux und Web-Override per ODER zusammen.
  - **AX_CLIENT_1_0**: `adapter::net::AX_CLIENT_1_0` — schreibt das zusammengeführte Signal per Remote-Write (`ID_WRITE_REMOTE`) an das Zielmodul.
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_4` — verknüpft die `QO`-Ausgänge von `Softkey_IXA`, `Aux_IXA`, `SUBSCRIBE_WEB` und `AX_CLIENT_1_0` zum SubApp-Ausgang `QO`.
- **Schnittstelle**: Ereigniseingang `INIT`, Ereignisausgang `INITO`, Dateneingang `QI` (Vorgabe `TRUE`, an die vier FBs durchgereicht), Datenausgang `QO`.
- **Funktionsweise**: Drei unabhängige Kommandoquellen (SoftKey, Aux, lokaler Web-Override) werden per `AX_OR_3` zusammengeführt und als einzelnes Remote-Write-Kommando an das Zielmodul übertragen.

Programmablauf und Verbindungen

1. `u16ObjId` ⟶ `Softkey_IXA.u16ObjId`; `u16ObjIdA` ⟶ `Aux_IXA.u16ObjId`; `ID_WEB_READ` ⟶ `SUBSCRIBE_WEB.ID`; `ID_WRITE_REMOTE` ⟶ `AX_CLIENT_1_0.ID` (Datenverbindungen, ausgeblendet).

2. `Softkey_IXA.IN` ⟶ `OR_MERGE.IN1`; `Aux_IXA.IN` ⟶ `OR_MERGE.IN2`; `SUBSCRIBE_WEB.OUT` ⟶ `OR_MERGE.IN3`.
3. `OR_MERGE.OUT` ⟶ `AX_CLIENT_1_0.IN`.
4. Initialisierung: `INIT` ⟶ `Softkey_IXA.INIT`; `Softkey_IXA.INITO` ⟶ `Aux_IXA.INIT`; `Aux_IXA.INITO` ⟶ `SUBSCRIBE_WEB.INIT`; `SUBSCRIBE_WEB.INITO` ⟶ `AX_CLIENT_1_0.INIT`; `AX_CLIENT_1_0.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`.
5. Qualität: `QI` ⟶ `Softkey_IXA.QI`, `Aux_IXA.QI`, `SUBSCRIBE_WEB.QI` und `AX_CLIENT_1_0.QI`; deren `QO`-Ausgänge ⟶ `AND_QO.IN1` bis `IN4`; `AND_QO.OUT` ⟶ `QO`.

Technische Besonderheiten

- **Drei gleichberechtigte Kommandoquellen**: SoftKey, AUX-Zuweisung und lokaler Web-Override wirken über das ODER gleichwertig — jede der drei Quellen kann das Kommando auslösen.
- **Kein Status-Feedback**: Im Gegensatz zu `Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC` enthält dieser Baustein keine Rückmeldung/Hintergrundfarbe; dafür ist `AX_SUBSCRIBE_BG3_WEB_OPC` mit denselben `u16ObjId`/`u16ObjIdA`-Werten danebenzustellen. Wichtig bei dieser manuellen Kombination: `ID_WEB_READ` (hier) und `ID_STATUS_WEB` (in `AX_SUBSCRIBE_BG3_WEB_OPC`) müssen unterschiedliche lokale OPC-UA-Knoten sein, sonst entsteht dieselbe Feedback-Loop-Gefahr wie in `Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC` beschrieben. `ID_WRITE_REMOTE` (hier) und `ID_SUBSCRIBE` (dort) zeigen dagegen beide auf das entfernte Zielmodul und dürfen denselben Remote-Knoten referenzieren.

Anwendungsszenarien

- Fern-Bedienung eines Zielmoduls über SoftKey, AUX-Joystick-Zuweisung oder einen lokalen Web-Client, ohne dass am selben Modul auch eine Statusanzeige benötigt wird.

Vergleich mit ähnlichen Bausteinen

Für die komplette, vorverdrahtete Kombination beider Seiten (Kommando + Status) ist weiterhin `Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC` zu verwenden, das diesen Baustein als Kommando-Teilbaustein enthält.

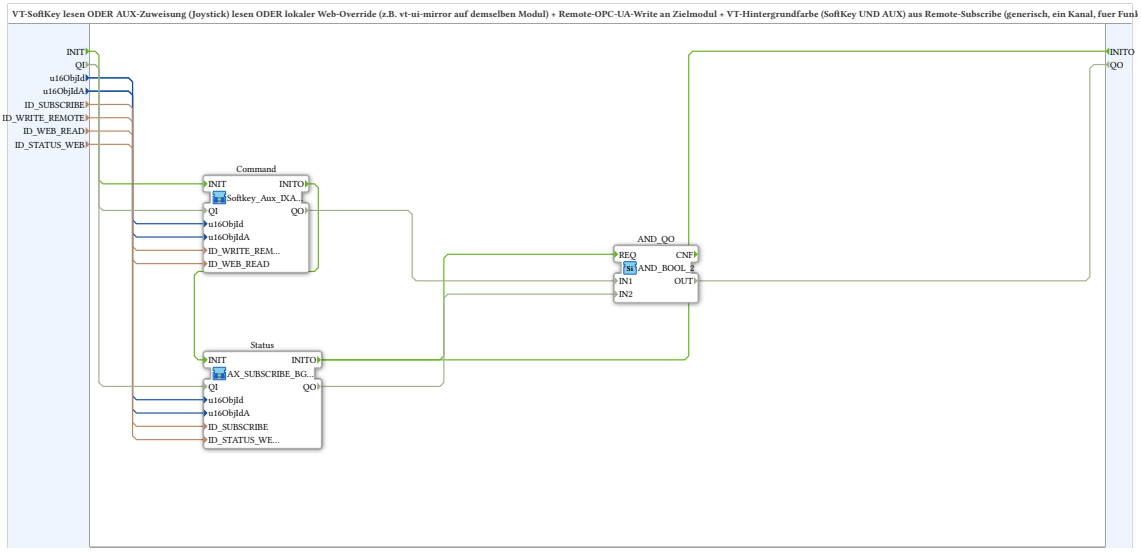
Zusammenfassung

`Softkey_Aux_IXA_TO_Remote_WRITE` kapselt die reine Kommando-Seite eines Remote-Kanals: drei ODER-verknüpfte Quellen (SoftKey, Aux, lokaler Web-Override), geschrieben per Remote-Write an ein Zielmodul.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC

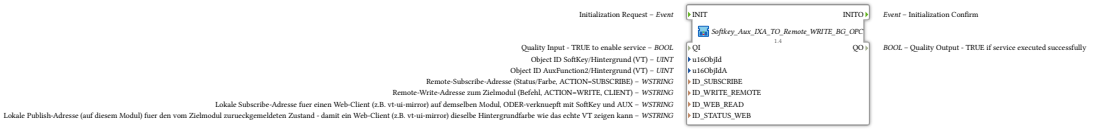


• – ○ • – ○ • – ○ •

Einleitung

`Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC` liest einen VT-SoftKey ODER eine AUX-Zuweisung (Joystick) ODER einen lokalen Web-Override und schreibt das Ergebnis per Remote-OPC-UA-Write an ein Zielmodul — zusätzlich wird die VT-Hintergrundfarbe (SoftKey UND AUX) aus einem Remote-Subscribe des Zielmoduls gespeist. Gedacht für Funktionen ohne lokalen physischen Ausgang, bei denen die Bedienung auf einem Modul (z. B. STG1) sitzt, der Aktor aber auf einem anderen Modul ohne eigene VT-Anbindung.

Seit Version 1.2 ist der Baustein ein dünnwandiger Wrapper, der zwei eigenständig wiederverwendbare Teilbausteine nebeneinanderstellt: `Softkey_Aux_IXA_TO_Remote_WRITE` (Kommando) und `AX_SUBSCRIBE_BG3_WEB_OPC` (Status). Die externe Schnittstelle blieb dabei zunächst unverändert; inzwischen besitzt der Wrapper zusätzlich die Ereignisse `INIT`/`INITO` sowie `QI` (BOOL, Initialwert `TRUE`) und `QO` (BOOL).



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **Command** (SubApp, `MyLib::sys::Softkey_Aux_IXA_TO_Remote_WRITE`): SoftKey ODER Aux ODER lokaler Web-Override ⟶ Remote-Write an das Zielmodul.
  - **Status** (SubApp, `MyLib::sys::AX_SUBSCRIBE_BG3_WEB_OPC`): Remote-Subscribe ⟶ Hintergrundfarbe auf SoftKey UND Aux + lokaler Web-Republish.
  - **AND_QO** (`iec61131::booleanOperators::AND_BOOL_2`): verknüpft `Command.QO` und `Status.QO` zum Ausgang `QO`.
- **Funktionsweise**: Beide Teilbausteine laufen datenseitig unabhängig nebeneinander (nur die Init-Kette und die Qualitätsverknüpfung verbinden sie); sie teilen sich lediglich dieselben `u16ObjId`/`u16ObjIdA`-Werte, damit Kommando und Statusanzeige auf demselben VT-Objekt wirken.

Programmablauf und Verbindungen

1. `u16ObjId` ⟶ `Command.u16ObjId` und `Status.u16ObjId`; `u16ObjIdA` ⟶ `Command.u16ObjIdA` und `Status.u16ObjIdA` (Datenverbindungen, ausgeblendet).
2. `ID_WRITE_REMOTE` ⟶ `Command.ID_WRITE_REMOTE`; `ID_WEB_READ` ⟶ `Command.ID_WEB_READ`.
3. `ID_SUBSCRIBE` ⟶ `Status.ID_SUBSCRIBE`; `ID_STATUS_WEB` ⟶ `Status.ID_STATUS_WEB`.
4. Abgesehen von der Init-Kette (Punkt 5) gibt es keine direkte SubApp-Verbindung zwischen `Command` und `Status` — beide SubApps sind vollständig eigenständig und kommunizieren nur indirekt über das Zielmodul (Command schreibt dorthin, Status abonniert von dort). Lokal können ihre Pfade dennoch verknüpft sein, wenn `ID_WEB_READ` und `ID_STATUS_WEB` versehentlich auf denselben OPC-UA-Knoten konfiguriert werden (siehe Feedback-Loop-Hinweis unten).
5. Initialisierung: `INIT` ⟶ `Command.INIT`; `Command.INITO` ⟶ `Status.INIT`; `Status.INITO` ⟶ `INITO` und `AND_QO.REQ`.
6. Qualität: `QI` ⟶ `Command.QI` und `Status.QI`; `Command.QO` ⟶ `AND_QO.IN1`; `Status.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`.

Technische Besonderheiten

- **Reiner Wrapper seit Version 1.2**: Die Zerlegung in `Command`/`Status` ändert nichts am Verhalten, erlaubt aber die unabhängige Wiederverwendung beider Hälften (z. B. `AX_SUBSCRIBE_BG3_WEB_OPC` allein für reine Statusanzeigen).
- **Kein lokaler physischer Ausgang**: Der Aktor sitzt auf einem anderen Modul; dieser Baustein bildet nur Bedienung und Statusrückmeldung auf dem Bedienmodul ab.
- **Getrennte lokale Knoten für Web-Override und Web-Status**: `ID_WRITE_REMOTE` (Schreiben) und `ID_SUBSCRIBE` (Lesen) zeigen beide auf das entfernte Zielmodul und dürfen dort durchaus denselben Remote-Knoten referenzieren — Write und Subscribe sind unterschiedliche OPC-UA-Operationen, kein Rückkopplungsrisiko auf diesem Modul. Die tatsächliche Feedback-Loop-Gefahr liegt bei den beiden *lokalen* Knoten dieses Moduls: `ID_WEB_READ` (von `Command`, per `AX_SUBSCRIBE_1` gelesen und ins Kom-

mando-ODER gemischt) und `ID_STATUS_WEB` (von `Status`, per `AX_PUBLISH_1` mit dem vom Zielmodul zurückgemeldeten Zustand beschrieben) müssen unterschiedliche lokale OPC-UA-Knoten sein — sonst liest `Command` den von `Status` republizierten Echo-Zustand als vermeintlichen Web-Override und schreibt ihn als neues Kommando zurück ans Zielmodul.

Anwendungsszenarien

- Bedienstellen ohne eigenen physischen Ausgang, bei denen sowohl SoftKey als auch AUX-Joystick-Zuweisung einen entfernten Aktor schalten sollen und der Bedienbildschirm zusätzlich dessen Status farblich anzeigen muss.

Vergleich mit ähnlichen Bausteinen

Wird nur ein SoftKey ohne Aux benötigt, ist `Softkey_IXA_TO_Remote_WRITE_BG_OPC` zu verwenden. Für Kommando bzw. Status allein (ohne die jeweils andere Hälfte) sind die Teilbausteine `Softkey_Aux_IXA_TO_Remote_WRITE` bzw. `AX_SUBSCRIBE_BG3_WEB_OPC` direkt einsetzbar.

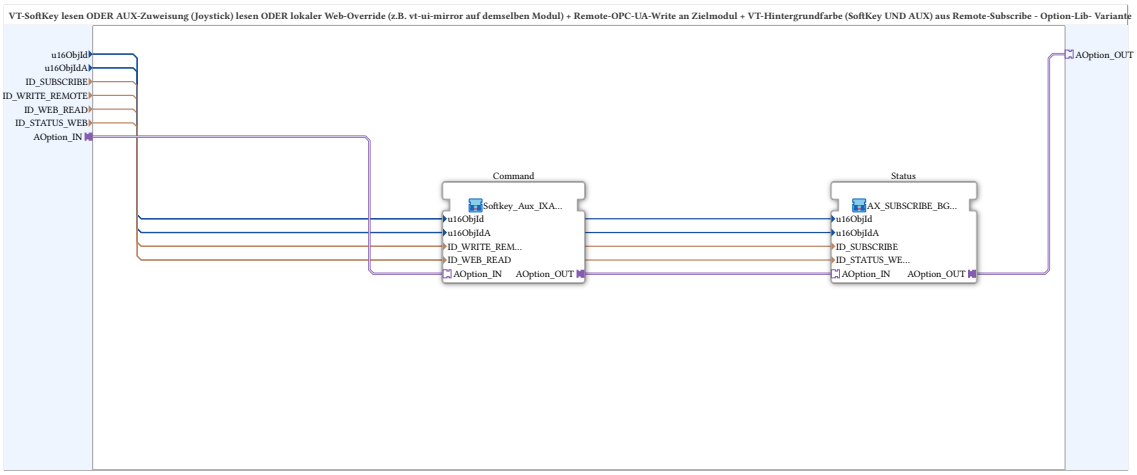
Zusammenfassung

`Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC` bündelt Kommando- und Status-Seite eines Remote-Kanals mit SoftKey- und Aux-Bedienung hinter einer stabilen, bekannten Schnittstelle — seit Version 1.2 als reiner Wrapper um zwei eigenständige Teilbausteine.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Softkey_Aux_IXA_TO_Remote_WRITE_BG_Option_OPC



• — ○ • — ○ • — ○ •

Einleitung

`Softkey_Aux_IXA_TO_Remote_WRITE_BG_Option_OPC` ist die Option-Lib-Variante von `Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC`: VT-SoftKey lesen ODER AUX-Zuweisung (Joystick) lesen ODER lokaler Web-Override, Remote-OPC-UA-Write an ein Zielmodul und VT-Hintergrundfarbe (SoftKey und AUX) aus einem Remote-Subscribe. Ist die Option nicht ausgewählt, sind sowohl die Kommando- Seite als auch die Status-Seite inaktiv (echtes `INIT_REQ`/`INIT_CNF`, kein VT-Object-Binding). Der Baustein ist ein dünnwandiger Wrapper aus zwei Teilbausteinen, die über den `AOption`-Adapter sequenziell verkettet werden.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden (die Initialisierung läuft über den Adapter `AOption_IN`).

Ereignis-Ausgänge
Keine vorhanden.

Daten-Eingänge

Variable	Typ	Vorgabe	Kommentar
u16ObjId	UINT		Object ID SoftKey/Hintergrund (VT).
u16ObjIdA	UINT		Object ID AuxFunction2/Hintergrund (VT).
ID_SUBSCRIBE	WSTRING		Remote-Subscribe-Adresse (Status/Farbe, ACTION=SUBSCRIBE).
ID_WRITE_REMOTE	WSTRING		Remote-Write-Adresse zum Zielmodul (Befehl, ACTION=WRITE, CLIENT).
ID_WEB_READ	WSTRING		Lokale Subscribe-Adresse für einen Web-Client auf demselben Modul, ODER-verknüpft mit SoftKey und AUX.
ID_STATUS_WEB	WSTRING		Lokale Publish-Adresse für den vom Zielmodul zurückgemeldeten Zustand.

Daten-Ausgänge
Keine vorhanden.

Adapter

Adapter	Typ	Richtung	Kommentar
AOption_IN	Option::AOption	Socket	Treibt INIT/QI von Command UND Status sequenziell; anschließen z. B. an AOption_AUI_DEMUX_N.OUTx oder das AOption_OUT einer vorgeschalteten Instanz.
AOption_OUT	Option::AOption	Plug	Reicht die Aktivierung an eine nachgeschaltete Instanz weiter (Daisy-Chain, z. B. eine zweite Instanz mit derselben Option); unbenutzt lassen, wenn nichts folgt.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Softkey_Aux_IXA_TO_Remote_WRITE_BG_Option_OPC

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **Command** (SubApp, MyLib::sys::Softkey_Aux_IXA_TO_Remote_WRITE_Option): SoftKey ODER Aux ODER Web-Override -> Remote-Write; siehe Softkey_Aux_IXA_TO_Remote_WRITE_Option.
  - **Status** (SubApp, MyLib::sys::AX_SUBSCRIBE_BG3_WEB_Option_OPC): Remote-Subscribe -> Hintergrundfarbe auf SoftKey und Aux plus lokaler Web-Republish; siehe AX_SUBSCRIBE_BG3_WEB_Option_OPC.

Programmablauf und Verbindungen

1. Adapterkette: AOption_IN -> Command.AOption_IN; Command.AOption_OUT -> Status.AOption_IN; Status.AOption_OUT -> AOption_OUT. Die Option aktiviert damit zuerst die Kommando-Seite, dann die Status-Seite, anschließend eine nachgeschaltete Instanz.
2. u16ObjId -> Command.u16ObjId und Status.u16ObjId; u16ObjIdA -> Command.u16ObjIdA und Status.u16ObjIdA.
3. ID_WRITE_REMOTE -> Command.ID_WRITE_REMOTE; ID_WEB_READ -> Command.ID_WEB_READ; ID_SUBSCRIBE -> Status.ID_SUBSCRIBE; ID_STATUS_WEB -> Status.ID_STATUS_WEB.
4. Der Baustein hat keine eigenen Ereignis- und QI/QO-Anschlüsse; Aktivierung und Bestätigung laufen ausschließlich über die Adapter AOption_IN/AOption_OUT.

Technische Besonderheiten

- **Command und Status sind beide Option-gated:** Alle Bausteine mit echtem `QI` sind über den `AOption`-Socket abschaltbar.
- **Sequenzielle Kette statt Socket-Split:** `Command.AOption_OUT -> Status.AOption_IN` (strikt 1:1), analog zum Daisy-Chain-Stil von `OptionalGroup_001a`. Der eigene `AOption_OUT` gibt die Kette an weitere Instanzen weiter, die dieselbe Option teilen.
- Wie im Original gilt: `ID_WEB_READ` (von `Command`) und `ID_STATUS_WEB` (von `Status`) müssen auf unterschiedliche lokale OPC-UA-Knoten zeigen, sonst entsteht eine Rückkopplung (siehe `Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC`).

Anwendungsszenarien

- Optionale Bedienstelle mit SoftKey und Aux-Zuweisung für einen entfernten Aktor samt farbiger Statusanzeige, die nur bei ausgewählter Option aktiv ist.
- Mehrere Instanzen, die dieselbe Option teilen und per `AOption_OUT` verkettet werden.

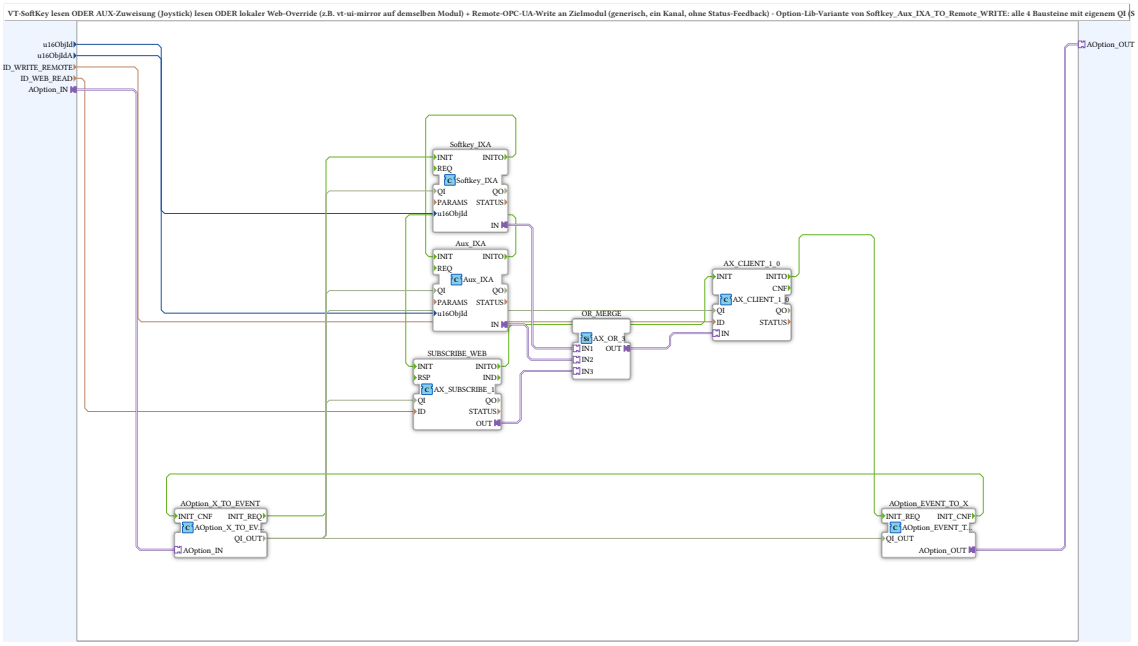
Zusammenfassung

`Softkey_Aux_IXA_TO_Remote_WRITE_BG_Option_OPC` bündelt Kommando- und Status-Seite eines Remote-Kanals hinter einem gemeinsamen `AOption`-Socket und reicht die Aktivierung per `AOption_OUT` weiter.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Softkey_Aux_IXA_TO_Remote_WRITE_Option



• – ○ • – ○ • – ○ •

Einleitung

`Softkey_Aux_IXA_TO_Remote_WRITE_Option` liest einen VT-SoftKey ODER eine AUX-Zuweisung (Joystick) ODER einen lokalen Web-Override (z. B. `vt-ui-mirror` auf demselben Modul) und schreibt das Ergebnis per Remote-OPC-UA-Write an ein Zielmodul (generisch, ein Kanal, ohne Status-Feedback). Es ist die Option-Lib-Variante von `Softkey_Aux_IXA_TO_Remote_WRITE`: Alle vier Bausteine mit eigenem `QI` (`Softkey_IXA`, `Aux_IXA`, `SUBSCRIBE_WEB`, `AX_CLIENT_1_0`) werden von einem einzigen `Option::AOption`-Socket sequenziell initialisiert und gated. Der Plug `AOption_OUT` reicht die Aktivierung an eine nachgeschaltete Gruppe weiter (Daisy-Chain).

Schnittstellenstruktur

Ereignis-Eingänge / -Ausgänge

Keine vorhanden (die Initialisierung läuft über den Adapter `AOption_IN`).

Daten-Eingänge

Variable	Typ	Vorgabe	Kommentar
`u16ObjId`	UINT	`ID_NULL`	Object ID SoftKey (VT).
`u16ObjIdA`	UINT	`ID_NULL`	Object ID AuxFunction2 (VT).
`ID_WRITE_REMOTE`	WSTRING		Remote-Write-Adresse zum Zielmodul (Befehl, `ACTION=WRITE`, CLIENT).
`ID_WEB_READ`	WSTRING		Lokale Subscribe-Adresse für einen Web-Client (z. B. `vt-ui-mirror`) auf demselben Modul, ODER-verknüpft mit SoftKey und AUX.

Daten-Ausgänge

Keine vorhanden.

Adapter

Adapter	Typ	Richtung	Kommentar
`AOption_IN`	`Option::AOption`	Socket	Treibt `INIT/QI` von `Softkey_IXA`, `Aux_IXA`, `SUBSCRIBE_WEB` und `AX_CLIENT_1_0`; anschließen z. B. an `AOption_AUI_DEMUX_N.OUTx`.
`AOption_OUT`	`Option::AOption`	Plug	Reicht die Aktivierung an eine nachgeschaltete Gruppe weiter (Daisy-Chain); unbenutzt lassen, wenn nichts folgt.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Softkey_Aux_IXA_TO_Remote_WRITE_Option

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **AOption_X_TO_EVENT** (`Option::conversion::AOption_X_TO_EVENT`): setzt den Socket `AOption_IN` in `INIT_REQ/QI_OUT` um, siehe `AOption_X_TO_EVENT`.
  - **Softkey_IXA** (`isobus::UT::io::Softkey::Softkey_IXA`): liest den SoftKey (`u16ObjId`).
  - **Aux_IXA** (`isobus::UT::io::Auxiliary::IN::Aux_IXA`): liest die AUX-Zuweisung (`u16ObjIdA`).
  - **SUBSCRIBE_WEB** (`adapter::net::AX_SUBSCRIBE_1`): liest den lokalen Web-Override (`ID_WEB_READ`).
  - **OR_MERGE** (`adapter::booleanOperators::AX_OR_3`): ODER-Verknüpfung der drei Quellen.
  - **AX_CLIENT_1_0** (`adapter::net::AX_CLIENT_1_0`): Remote-Write auf `ID_WRITE_REMOTE`.
  - **AOption_EVENT_TO_X** (`Option::conversion::AOption_EVENT_TO_X`): gibt die Aktivierung an `AOption_OUT` weiter.

Programmablauf und Verbindungen

1. Sequenzielle Initialisierung: `AOption_X_TO_EVENT.INIT_REQ -> Softkey_IXA.INIT; Softkey_IXA.INITO -> Aux_IXA.INIT; Aux_IXA.INITO -> SUBSCRIBE_WEB.INIT; SUBSCRIBE_WEB.INITO -> AX_CLIENT_1_0.INIT; AX_CLIENT_1_0.INITO -> AOption_EVENT_TO_X.INIT_REQ; AOption_EVENT_TO_X.INIT_CNF -> AOption_X_TO_EVENT.INIT_CNF`.

2. `AOption_X_TO_EVENT.QI_OUT` -> `QI` von `Softkey_IXA`, `Aux_IXA`, `SUBSCRIBE_WEB`, `AX_CLIENT_1_0` und `AOption_EVENT_TO_X.QI_OUT`.
3. Daten: `Softkey_IXA.IN` -> `OR_MERGE.IN1`; `Aux_IXA.IN` -> `OR_MERGE.IN2`; `SUBSCRIBE_WEB.OUT` -> `OR_MERGE.IN3`; `OR_MERGE.OUT` -> `AX_CLIENT_1_0.IN`.
4. Parameter: `u16ObjId` -> `Softkey_IXA.u16ObjId`; `u16ObjIdA` -> `Aux_IXA.u16ObjId`; `ID_WEB_READ` -> `SUBSCRIBE_WEB.ID`; `ID_WRITE_REMOTE` -> `AX_CLIENT_1_0.ID`.
5. Adapter: `AOption_IN` -> `AOption_X_TO_EVENT.AOption_IN`; `AOption_EVENT_TO_X.AOption_OUT` -> `AOption_OUT`.

Technische Besonderheiten

- **Echtes `INIT_REQ`/`INIT_CNF` statt nur eines `QI`-Bools:** Ist die Option nicht ausgewählt, bleiben alle vier Bausteine inaktiv.
- **Sequenziell statt parallel:** Die Initialisierung läuft als Daisy-Chain (`Softkey_IXA` -> `Aux_IXA` -> `SUBSCRIBE_WEB` -> `AX_CLIENT_1_0`); `AOption_OUT` erlaubt, weitere Instanzen mit derselben Option anzuhängen (z. B. eine zweite SoftKey-Instanz) oder, wie in `Softkey_Aux_IXA_TO_Remote_WRITE_BG_Option_OPC`, die Status-Hälfte.
- Das Gegenstück ohne Option ist `Softkey_Aux_IXA_TO_Remote_WRITE`.

Anwendungsszenarien

- Optionale Bedienfunktion, die nur bei ausgewählter Option SoftKey, Aux und Web-Override liest und per Remote-Write an ein Zielmodul schreibt.
- Mehrere SoftKeys, die dieselbe Option teilen, per `AOption_OUT` verkettet.

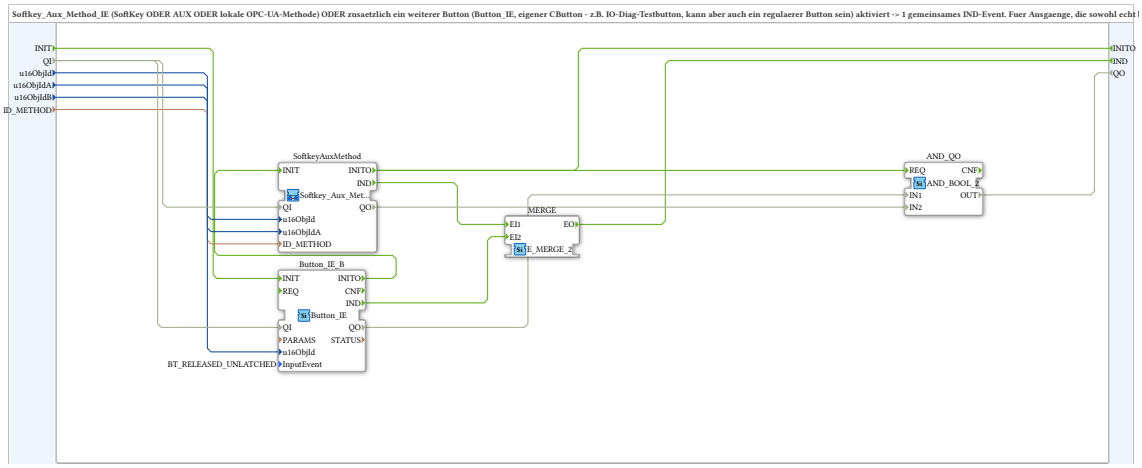
Zusammenfassung

`Softkey_Aux_IXA_TO_Remote_WRITE_Option` ist die Option-gated Variante des Remote-Write-Kommandos mit SoftKey, Aux und Web-Override, deren vier `QI`-Bausteine über einen `AOption`-Socket aktiviert werden und die die Aktivierung per `AOption_OUT` weiterreicht.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Softkey_Aux_Method_Button_IE



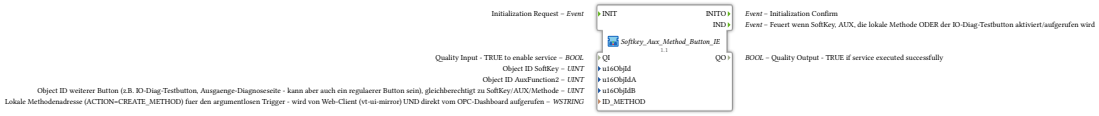
• — ○ • — ○ • — ○ •

Einleitung

`Softkey_Aux_Method_Button_IE` erweitert `Softkey_Aux_Method_IE` (SoftKey ODER AUX ODER lokale OPC-UA-Methode) um einen **weiteren Button** (`Button_IE`, eigener `CButton` – z. B. ein IO-Diag-Testbutton, kann aber auch ein regulärer Button sein). Aktiviert eine dieser Quellen, entsteht ein gemeinsames `IND`-Ereignis.

Der Baustein dient für Ausgänge, die sowohl echt bedient (SoftKey/AUX/Web/Dashboard) als auch per zusätzlichem Button gleichberechtigt getriggert werden sollen. Der Hintergrund des Buttons wird

vom Aufrufer separat aus dem resultierenden Zustand gespeist; dafür bleibt `u16ObjIdB` beim Aufrufer verfügbar.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Softkey_Aux_Method_Button_IE

- **Typ**: SubAppType
- **Verwendete interne Bausteine**:
  - **SoftkeyAuxMethod**: SubApp `MyLib::sys::Softkey_Aux_Method_IE` – SoftKey, AUX oder lokale Methode
  - **Button_IE_B**: `isobus::UT::io::Button::Button_IE (InputEvent=BT_RELEASED_UNLATCHED)` – weiterer Button
  - **MERGE**: `iec61499::events::E_MERGE_2` – führt beide `IND`-Ereignisse zusammen
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_2` – verknüpft die `QO` von `Button_IE_B` und `SoftkeyAuxMethod`
- **Importe**: `ID_NULL`, `BT_RELEASED_UNLATCHED`

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; geht an `Button_IE_B.INIT`

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierung abgeschlossen; kommt von `SoftkeyAuxMethod.INITO`
- **IND**: Feuert, wenn SoftKey, AUX, die lokale Methode oder der zusätzliche Button aktiviert bzw. aufgerufen wird

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input - TRUE aktiviert den Dienst; geht an `Button_IE_B` und `SoftkeyAuxMethod`
- **u16ObjId** (`UINT`): Object-ID des SoftKeys
- **u16ObjIdA** (`UINT`): Object-ID der `AuxFunction2`
- **u16ObjIdB** (`UINT`, Vorgabe `ID_NULL`): Object-ID eines weiteren Buttons (z. B. IO-Diag-Testbutton auf der Ausgänge-Diagnoseseite; kann auch ein regulärer Button sein), gleichberechtigt zu SoftKey/AUX/Methode
- **ID_METHOD** (`WSTRING`): Lokale Methodenadresse (`ACTION=CREATE_METHOD`) für den argumentlosen Trigger – wird vom Web-Client (`vt-ui-mirror`) und direkt vom OPC-Dashboard aufgerufen

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde (AND der `QO` von `Button_IE_B` und `SoftkeyAuxMethod`)

Programmablauf und Verbindungen

- `u16ObjId`, `u16ObjIdA`, `ID_METHOD` ⟶ `SoftkeyAuxMethod`
- `u16ObjIdB` ⟶ `Button_IE_B.u16ObjId`
- `SoftkeyAuxMethod.IND` ⟶ `MERGE.EI1`; `Button_IE_B.IND` ⟶ `MERGE.EI2`
- `MERGE.EO` ⟶ `IND`
- Initialisierung: `INIT` ⟶ `Button_IE_B.INIT`, `Button_IE_B.INITO` ⟶ `SoftkeyAuxMethod.INIT`, `SoftkeyAuxMethod.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`
- Qualität: `QI` ⟶ `Button_IE_B.QI` und `SoftkeyAuxMethod.QI`; `Button_IE_B.QO` ⟶ `AND_QO.IN1`, `SoftkeyAuxMethod.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`

Technische Besonderheiten

- Version 1.1 (2026-09-15): `u16ObjId_TEST` in `u16ObjIdB` umbenannt (`Button_IE_TEST` ⟶ `Button_IE_B`), da der zusätzliche Button nicht immer ein IO-Diag-Testbutton ist

- Wird u. a. von `AX_SoftkeyToggle_RS_TO_AX` für den Toggle-Zweig genutzt

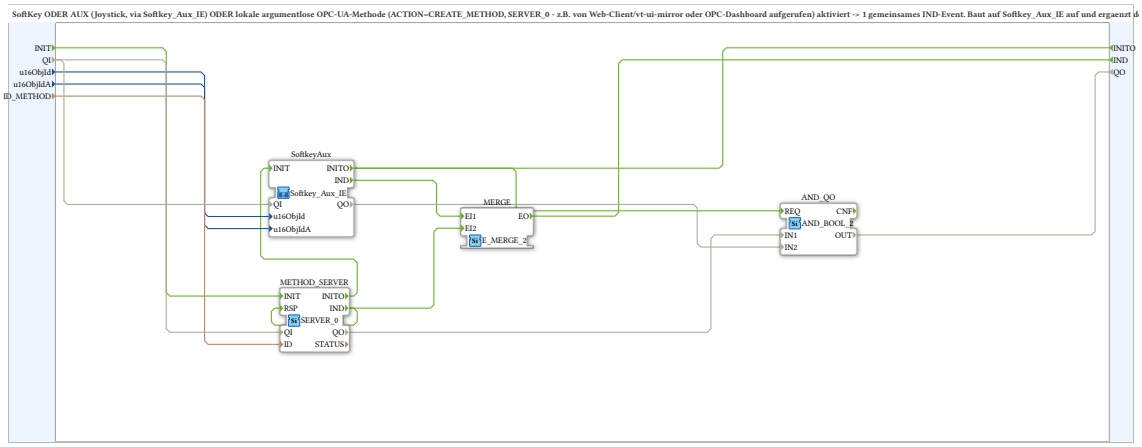
Anwendungsszenarien

- Toggle-Ausgang, der per SoftKey, Joystick, Web/Dashboard und zusätzlichem Diagnose-Button gleichberechtigt bedient werden soll

Zusammenfassung

`Softkey_Aux_Method_Button_IE` ergänzt `Softkey_Aux_Method_IE` um einen weiteren Button und liefert ein gemeinsames `IND`-Ereignis.

Softkey_Aux_Method_IE



• — ○ • — ○ • — ○ •

Einleitung

`Softkey_Aux_Method_IE` führt drei Auslöser zu **einem gemeinsamen `IND`-Ereignis** zusammen: einen VT-SoftKey, eine AUX-Funktion (Joystick, über `Softkey_Aux_IE`) oder eine **lokale argumentlose OPC-UA-Methode** (`ACTION=CREATE_METHOD`, `SERVER_0`), die z. B. vom Web-Client (`vt-ui-mirror`) oder vom OPC-Dashboard aufgerufen wird. Der Baustein baut auf `Softkey_Aux_IE` auf und ergänzt den Methodenkanal. Im Unterschied zu `Softkey_Aux_Web_IE` (Web per `SUBSCRIBE_R_TRIG` auf eine geschriebene Variable) erfolgt der Web-/Dashboard-Auslöser hier per echtem RPC-Methodenaufruf, ohne gemeinsame Variable.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung
- **IND**: Feuert, wenn SoftKey, AUX oder die lokale Methode aktiviert bzw. aufgerufen wird

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input – TRUE gibt den Dienst frei
- **u16ObjId** (`UINT`): Object-ID des SoftKeys
- **u16ObjIdA** (`UINT`): Object-ID der AuxFunction2
- **ID_METHOD** (`WSTRING`): Lokale Methodenadresse (`ACTION=CREATE_METHOD`) für den argumentlosen Trigger; wird vom Web-Client (`vt-ui-mirror`) und direkt vom OPC-Dashboard aufgerufen

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output – UND aus `METHOD_SERVER.QO` und `SoftkeyAux.QO`

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Verwendete Funktionsbausteine (FBs)

- **SoftkeyAux**: SubApp `MyLib::sys::Softkey_Aux_IE` – SoftKey oder AUX
- **METHOD_SERVER**: `iec61499::net::SERVER_0` – lokale Methode
- **MERGE**: `iec61499::events::E_MERGE_2` – führt die beiden Auslöser zusammen
- **AND_QO**: `iec61131::booleanOperators::AND_BOOL_2` – verknüpft `METHOD_SERVER.QO` und `SoftkeyAux.QO` zum `QO`

Programmablauf und Verbindungen

- `SoftkeyAux.IND` ⟶ `MERGE.EI1`
- `METHOD_SERVER.IND` ⟶ `MERGE.EI2`
- `METHOD_SERVER.IND` ⟶ `METHOD_SERVER.RSP` (Rückverdrahtung)
- `MERGE.EO` ⟶ `IND`
- `u16ObjId`, `u16ObjIdA` ⟶ `SoftkeyAux`; `ID_METHOD` ⟶ `METHOD_SERVER.ID`
- `INIT` ⟶ `METHOD_SERVER.INIT`; `METHOD_SERVER.INITO` ⟶ `SoftkeyAux.INIT`; `SoftkeyAux.INITO` ⟶ `INITO` (und ⟶ `AND_QO.REQ`)
- `QI` ⟶ `METHOD_SERVER.QI` und `SoftkeyAux.QI`
- `METHOD_SERVER.QO` ⟶ `AND_QO.IN1`; `SoftkeyAux.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`

Technische Besonderheiten

- **`RSP` muss auf `IND` zurückverdrahtet werden.** Der Trigger ist argumentlos und braucht keine Verarbeitung; ohne diese Verbindung blockiert FORTEs `onServerMethodCall` den OPC-UA-Server-Thread bis zu 4 s und hält dabei den globalen `serviceMutex` von open62541, den jeder `AX_PUBLISH_1` auf einem anderen FORTE-Thread ebenfalls braucht (siehe `Ventilsteuerung/OPC_UA_VERNETZUNG.md`, Punkt 12).

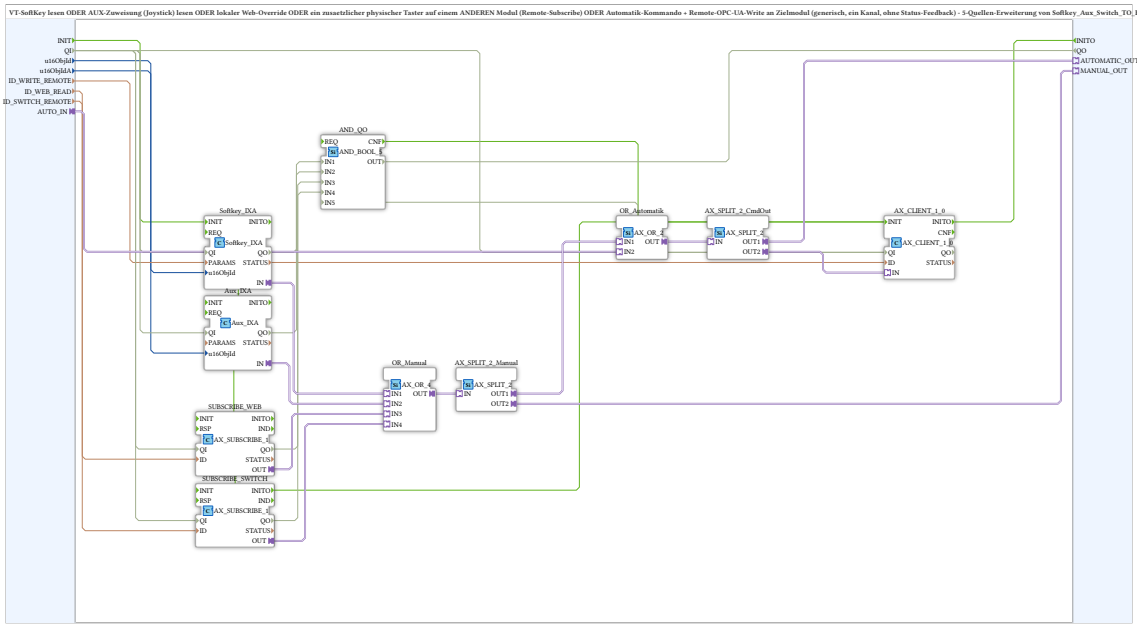
Anwendungsszenarien

- Taste, Joystick-Zuweisung und Web-/Dashboard-Aufruf, die dieselbe Aktion auslösen (z. B. „Alles Aus“)

Fazit

`Softkey_Aux_Method_IE` vereint SoftKey, AUX und einen OPC-UA-Methodenaufruf zu einem gemeinsamen `IND`-Ereignis.

Softkey_Aux_Switch_Auto_TO_Remote_WRITE



• – ○ • – ○ • – ○ •

Einleitung

`Softkey_Aux_Switch_Auto_TO_Remote_WRITE` verknüpft per ODER fünf Quellen eines Befehls: den VT-SoftKey, die AUX-Zuweisung (Joystick), einen lokalen Web-Override, einen zusätzlichen physischen Taster auf einem anderen Modul (Remote-Subscribe) und ein autonomes Automatik-Kommando (`AUTO_IN`). Das Ergebnis wird per Remote-OPC-UA-Write an ein Zielmodul geschrieben (generisch, ein Kanal, ohne Status-Feedback). Der Baustein ist die 5-Quellen-Erweiterung von `Softkey_Aux_Switch_TO_Remote_WRITE` um den zusätzlichen Socket `AUTO_IN` (z. B. Befüllautomatik-Hysterese, die dieselbe Füllband-Heben-Leitung ansteuert wie SoftKey/AUX/Web/physischer Taster).

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialization Request

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialization Confirm

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input, TRUE aktiviert den Dienst
- **u16ObjId** (UINT): Object ID des SoftKeys (VT), Default `ID_NULL`
- **u16ObjIdA** (UINT): Object ID der AuxFunction2 (VT), Default `ID_NULL`
- **ID_WRITE_REMOTE** (WSTRING): Remote-Write-Adresse zum Zielmodul (Befehl, `ACTION=WRITE`, CLIENT)
- **ID_WEB_READ** (WSTRING): lokale Subscribe-Adresse für einen Web-Client (z. B. vt-ui-mirror) auf demselben Modul, ODER-verknüpft mit SoftKey und AUX
- **ID_SWITCH_REMOTE** (WSTRING): Remote-Subscribe-Adresse eines zusätzlichen physischen Tasters auf einem anderen Modul, ODER-verknüpft mit SoftKey/AUX/Web/Automatik

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output, UND-Verknüpfung (`AND_QO`, `AND_BOOL_5`) der `QO`-Ausgänge von `Softkey_IXA`, `Aux_IXA`, `SUBSCRIBE_WEB`, `SUBSCRIBE_SWITCH` und `AX_CLIENT_1_0`

Adapter

- **AUTO_IN** (Socket, `AX`): autonomes Kommando (z. B. Befüllautomatik-Hysterese), ODER-verknüpft mit SoftKey/AUX/Web/physischem Taster
- **AUTOMATIC_OUT** (Plug, `AX`): optional; lokal gemergter Rohbefehl (SoftKey ODER AUX ODER Web ODER Switch ODER Automatik) vor dem Remote-Write. Unverdrahtet lassen, wenn nicht gebraucht
- **MANUAL_OUT** (Plug, `AX`): optional; nur die manuellen Quellen (SoftKey ODER AUX ODER Web ODER Switch), **ohne** `AUTO_IN`. Damit kann z. B. eine Automatik abgeschaltet werden, wenn der Bediener manuell eingreift, ohne dass sich die Automatik durch ihr eigenes Kommando selbst abschaltet. Unverdrahtet lassen, wenn nicht gebraucht

Funktionsweise

Das Netzwerk arbeitet in zwei ODER-Stufen:

1. **Manuelle Quellen**: `Softkey_IXA` (`isobus::UT::io::Softkey::Softkey_IXA`, `u16ObjId`) → `OR_Manual.IN1`, `Aux_IXA` (`isobus::UT::io::Auxiliary::IN::Aux_IXA`, `u16ObjIdA`) → `OR_Manual.IN2`, `SUBSCRIBE_WEB` (`AX_SUBSCRIBE_1`, Adresse `ID_WEB_READ`) → `OR_Manual.IN3`, `SUBSCRIBE_SWITCH` (`AX_SUBSCRIBE_1`, Adresse `ID_SWITCH_REMOTE`) → `OR_Manual.IN4`. `OR_Manual` ist ein `AX_OR_4`.
2. `OR_Manual.OUT` → `AX_SPLIT_2_Manual`: `OUT1` → `OR_Automatik.IN1`, `OUT2` → `MANUAL_OUT`.
3. **Automatik**: `AUTO_IN` → `OR_Automatik.IN2`. `OR_Automatik` ist ein `AX_OR_2`.
4. `OR_Automatik.OUT` → `AX_SPLIT_2_CmdOut`: `OUT1` → `AUTOMATIC_OUT`, `OUT2` → `AX_CLIENT_1_0.IN`.
5. `AX_CLIENT_1_0` (`adapter::net::AX_CLIENT_1_0`) schreibt den Befehl per OPC-UA an die Adresse `ID_WRITE_REMOTE`.

6. `AND_QO` (`iec61131::booleanOperators::AND_BOOL_5`) verknüpft die `QO`-Ausgänge von `Softkey_IXA` (`IN1`), `Aux_IXA` (`IN2`), `SUBSCRIBE_WEB` (`IN3`), `SUBSCRIBE_SWITCH` (`IN4`) und `AX_CLIENT_1_0` (`IN5`) zu `QO`.
7. Initialisierung: `INIT` ⟶ `Softkey_IXA.INIT` ⟶ `Aux_IXA.INIT` ⟶ `SUBSCRIBE_WEB.INIT` ⟶ `SUBSCRIBE_SWITCH.INIT` ⟶ `AX_CLIENT_1_0.INIT`; `AX_CLIENT_1_0.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`. `QI` wird an alle fünf Bausteine verteilt.

Technische Besonderheiten

- Der frühere `AX_OR_5` wurde in `OR_Manual` (`AX_OR_4`) und `OR_Automatik` (`AX_OR_2`) aufgeteilt, damit `MANUAL_OUT` ohne Automatik-Anteil zur Verfügung steht; das Gesamtverhalten für `AUTOMATIC_OUT` und den Remote-Write blieb unverändert
- `MANUAL_OUT` verhindert Selbstabschaltung: Würde die Automatik ihr eigenes Kommando zur Abschaltung nutzen, schaltete sie sich durch dieses selbst ab
- Jeder Adapter wird nur über `AX_SPLIT_2` verzweigt

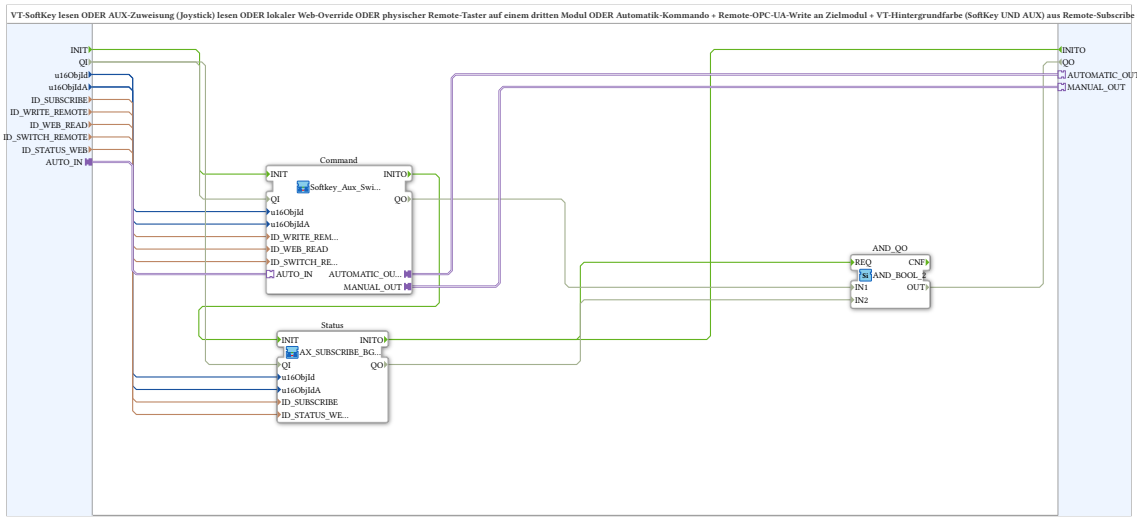
Anwendungsszenarien

- Füllband-Heben: Der Befehl kann vom Bediener (SoftKey/AUX/Web/physischer Taster S08) oder von der Befüllautomatik-Hysterese kommen und wird an das Zielmodul geschrieben
- Automatik, die sich abschaltet, sobald der Bediener manuell eingreift (über `MANUAL_OUT`)

Fazit

`Softkey_Aux_Switch_Auto_TO_Remote_WRITE` führt fünf Befehlsquellen per ODER zusammen, schreibt das Ergebnis per OPC-UA an ein Zielmodul und stellt über `AUTOMATIC_OUT` und `MANUAL_OUT` optional zwei Zwischenstände bereit.

Softkey_Aux_Switch_Auto_TO_Remote_WRITE_BG_OPC



• — ○ • — ○ • — ○ •

Einleitung

`Softkey_Aux_Switch_Auto_TO_Remote_WRITE_BG_OPC` kombiniert für einen Befehl an ein Zielmodul: VT-**SoftKey** lesen ODER **AUX-Zuweisung** (Joystick) lesen ODER lokaler **Web-Override** ODER physischer **Remote-Taster** auf einem dritten Modul ODER **Automatik-Kommando**, dazu ein **Remote-OPC-UA-Write** an das Zielmodul und die **VT-Hintergrundfarbe** (SoftKey und AUX) aus einem Remote-Subscribe. Es ist die 5-Quellen-Variante von `Softkey_Aux_Switch_TO_Remote_WRITE_BG_OPC` mit einem zusätzlichen `AUTO_IN`-Socket, z. B. Füllband Heben: SoftKey/AUX/Web/physischer Taster S08 UND Befüllautomatik-Hysterese.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialization Request

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialization Confirm

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input - TRUE, um den Dienst freizugeben
- **u16ObjId** (`UINT`): Object ID SoftKey/Hintergrund (VT)
- **u16ObjIdA** (`UINT`): Object ID AuxFunction2/Hintergrund (VT)
- **ID_SUBSCRIBE** (`WSTRING`): Remote-Subscribe-Adresse (Status/Farbe, `ACTION=SUBSCRIBE`)
- **ID_WRITE_REMOTE** (`WSTRING`): Remote-Write-Adresse zum Zielmodul (Befehl, `ACTION=WRITE`, CLIENT)
- **ID_WEB_READ** (`WSTRING`): lokale Subscribe-Adresse für einen Web-Client (z. B. vt-ui-mirror) auf demselben Modul, ODER-verknüpft mit SoftKey und AUX
- **ID_SWITCH_REMOTE** (`WSTRING`): Remote-Subscribe-Adresse eines zusätzlichen physischen Tasters auf einem anderen Modul, ODER-verknüpft mit SoftKey/AUX/Web/Automatik
- **ID_STATUS_WEB** (`WSTRING`): lokale Publish-Adresse (auf diesem Modul) für den vom Zielmodul zurückgemeldeten Zustand, damit ein Web-Client dieselbe Hintergrundfarbe wie das echte VT zeigen kann

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde (UND von `Command.QO` und `Status.QO`)

Adapter

Alle Adapter sind vom Typ `adapter::types::unidirectional::AX`.

- **AUTO_IN** (Socket): autonomes Kommando (z. B. Befüllautomatik-Hysterese), ODER-verknüpft mit SoftKey/Aux/Web/physischem Taster
- **AUTOMATIC_OUT** (Plug): optional, lokal gemergter Rohbefehl, durchgereicht von `Command.AUTOMATIC_OUT`; unverdrahtet lassen, wenn nicht gebraucht
- **MANUAL_OUT** (Plug): optional, nur die manuellen Quellen (SoftKey/AUX/Web/Switch) ohne Automatik-Anteil, durchgereicht von `Command.MANUAL_OUT`; unverdrahtet lassen, wenn nicht gebraucht

Verwendete Funktionsbausteine (FBs)

- **`Command`** (SubApp `MyLib::sys::Softkey_Aux_Switch_Auto_TO_Remote_WRITE`, Softkey_Aux_Switch_Auto_TO_Remote_WRITE): bildet den Befehl aus allen Quellen und schreibt ihn remote
- **`Status`** (SubApp `MyLib::sys::AX_SUBSCRIBE_BG3_WEB_OPC`, AX_SUBSCRIBE_BG3_WEB_OPC): abonniert den Zustand, färbt den Hintergrund und publiziert ihn für den Web-Client
- **`AND_QO`** (`iec61131::booleanOperators::AND_BOOL_2`): verknüpft `Command.QO` und `Status.QO` per UND zu `QO`

Funktionsweise

- `u16ObjId`, `u16ObjIdA`, `ID_WRITE_REMOTE`, `ID_WEB_READ`, `ID_SWITCH_REMOTE` ⟶ gleichnamige Eingänge von `Command`
- `AUTO_IN` ⟶ `Command.AUTO_IN`
- `Command.AUTOMATIC_OUT` ⟶ `AUTOMATIC_OUT`; `Command.MANUAL_OUT` ⟶ `MANUAL_OUT`
- `u16ObjId`, `u16ObjIdA`, `ID_SUBSCRIBE`, `ID_STATUS_WEB` ⟶ gleichnamige Eingänge von `Status`
- Initialisierungskette: `INIT` ⟶ `Command.INIT`, `Command.INITO` ⟶ `Status.INIT`, `Status.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`
- `QI` ⟶ `Command.QI` und `Status.QI`
- `Command.QO` ⟶ `AND_QO.IN1`, `Status.QO` ⟶ `AND_QO.IN2`, `AND_QO.OUT` ⟶ `QO`

Technische Besonderheiten

- Befehlspfad (`Command`) und Statuspfad (`Status`) sind getrennt: der Befehl geht per Remote-Write zum Zielmodul, die Hintergrundfarbe kommt aus dem vom Zielmodul zurückgemeldeten Zustand

- Optionale Ausgänge `AUTOMATIC_OUT`/`MANUAL_OUT` erlauben die lokale Weiterverwendung des gemergten Befehls bzw. nur des manuellen Anteils

Anwendungsszenarien

- Gemischter manueller/automatischer Bedienpunkt, z. B. Füllband Heben (SoftKey, AUX, Web, physischer Taster S08 und Befüllautomatik)

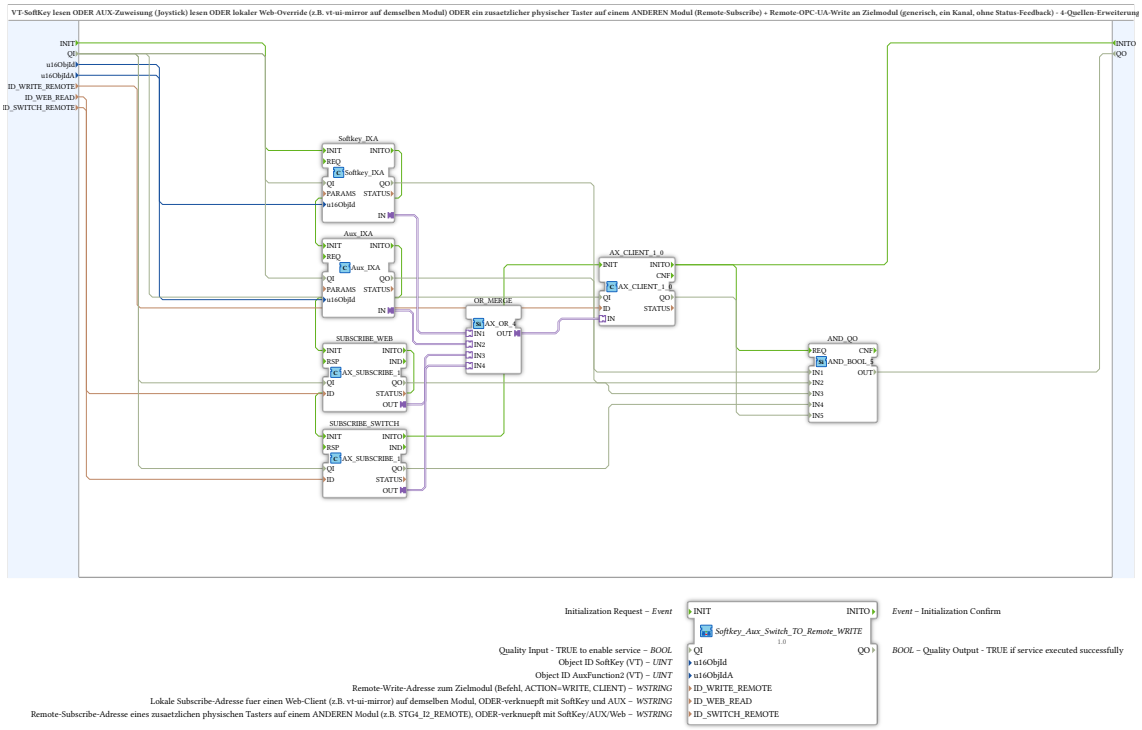
Zusammenfassung

Die SubApp bündelt fünf Befehlsquellen, einen Remote-Write an das Zielmodul und die rückgemeldete VT-Hintergrundfarbe.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Softkey_Aux_Switch_TO_Remote_WRITE



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsbaustein `Softkey_Aux_Switch_TO_Remote_WRITE` ist eine Subapp, die vier unabhängige Quellen – einen VT-Softkey, eine Auxiliary-Funktion (z. B. Joystick), einen lokalen Web-Override (z. B. vt-ui-mirror) und einen zusätzlichen physischen Taster auf einem anderen Modul (Remote-Subscribe) – zu einer einzigen ODER-Verknüpfung zusammenführt. Das resultierende Signal wird anschließend per OPC-UA-Client-Write als Kommando an ein entferntes Zielmodul gesendet. Die Subapp erweitert das Konzept von `Softkey_Aux_IXA_TO_Remote_WRITE` um eine vierte Quelle und ist speziell für Anwendungen gedacht, bei denen neben SoftKey/AUX/Web noch ein fest verbauter Taster auf einem dritten Modul vorhanden ist, z. B. bei einer Schnittverstellung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; startet die Kette `Softkey_IXA` ⟶ `Aux_IXA` ⟶ `SUBSCRIBE_WEB` ⟶ `SUBSCRIBE_SWITCH` ⟶ `AX_CLIENT_1_0`.

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung; kommt von `AX_CLIENT_1_0.INITO`.

Daten-Eingänge

Name	Typ	Kommentar
`QI`	`BOOL`	Quality Input, Freigabe des Dienstes (Vorgabe `TRUE`); wird an alle fünf internen FBs weitergereicht
`u16ObjId`	`UINT`	Object ID SoftKey (VT)
`u16ObjIdA`	`UINT`	Object ID AuxFunction2 (VT)
`ID_WRITE_REMOTE`	`WSTRING`	Remote-Write-Adresse zum Zielmodul (Befehl, ACTION=WRITE, CLIENT)
`ID_WEB_READ`	`WSTRING`	Lokale Subscribe-Adresse für einen Web-Client (z. B. vt-ui-mirror) auf demselben Modul, ODER-verknüpft mit SoftKey und AUX
`ID_SWITCH_REMOTE`	`WSTRING`	Remote-Subscribe-Adresse eines zusätzlichen physischen Tasters auf einem **anderen** Modul (z. B. STG4_I2_REMOTE), ODER-verknüpft mit SoftKey/AUX/Web

Alle Eingänge sind als `InputVars` definiert; der einzige Daten-Ausgang ist `QO`.

Daten-Ausgänge

Name	Typ	Kommentar
`QO`	`BOOL`	Quality Output, `AND` der `QO`-Ausgänge von `Softkey_IXA`, `Aux_IXA`, `SUBSCRIBE_WEB`, `SUBSCRIBE_SWITCH` und `AX_CLIENT_1_0` (über `AND_QO`, `AND_BOOL_5`)

Adapter

Die Subapp besitzt **keine externen Adapter-Schnittstellen**. Die internen Adapterverbindungen dienen ausschließlich zur Verdrahtung der enthaltenen Funktionsbausteine.

Funktionsweise

Die Subapp kombiniert vier boolesche Zustände logisch ODER:

1. **SoftKey** – über `u16ObjId` wird der aktive Zustand eines VT-Softkeys gelesen (intern über `Softkey_IXA`).
2. **Auxiliary-Funktion** – über `u16ObjIdA` wird der Zustand einer Auxiliary-Zuweisung (z. B. Joystick-Taste) erfasst (intern über `Aux_IXA`).
3. **Web-Override** – über `ID_WEB_READ` wird ein lokal per OPC-UA-Subscription empfangenes Signal eines Web-Clients (z. B. vt-ui-mirror) integriert (intern über `SUBSCRIBE_WEB`).
4. **Remote-Taster** – über `ID_SWITCH_REMOTE` wird ein zusätzlicher physischer Taster auf einem anderen Modul per Remote-Subscription gelesen (intern über `SUBSCRIBE_SWITCH`).

Alle vier Signale werden im Baustein `AX_OR_4` zusammengeführt. Das Ergebnis dient als Eingang für den OPC-UA-Client `AX_CLIENT_1_0`, der das Kommando (als Write-Request) an die in `ID_WRITE_REMOTE` spezifizierte Zieladresse sendet. Die gesamte Verarbeitung erfolgt asynchron; es gibt keine Rückmeldung über den Erfolg des Schreibvorgangs.

Initialisierung und Qualität: `INIT` → `Softkey_IXA.INIT`, `Softkey_IXA.INITO` → `Aux_IXA.INIT`, `Aux_IXA.INITO` → `SUBSCRIBE_WEB.INIT`, `SUBSCRIBE_WEB.INITO` → `SUBSCRIBE_SWITCH.INIT`, `SUBSCRIBE_SWITCH.INITO` → `AX_CLIENT_1_0.INIT`, `AX_CLIENT_1_0.INITO` → `INITO` und → `AND_QO.REQ`. `QI` geht an alle fünf FBs; deren `QO`-Ausgänge laufen auf `AND_QO.IN1` bis `IN5`, `AND_QO.OUT` → `QO`. Die internen FBs tragen kein festes `QI=TRUE` mehr.

Technische Besonderheiten

- **Lokale ODER-Verknüpfung**: Alle vier Quellen werden im Bedienmodul (z. B. STG1) zusammengeführt und **erst dann** als ein einziges Kommando an das Zielmodul gesendet. Dies vermeidet Race-Conditions oder „Last-Write-Wins"-Probleme, die bei mehreren unabhängigen Writern auf denselben Zielknoten auftreten könnten.

- **Kein Status-Feedback**: Der Baustein sendet ausschließlich Kommandos. Für eine Rückmeldung (z. B. Hintergrundfarbe) muss ein separater Baustein wie `AX_SUBSCRIBE_BG3_WEB_OPC` verwendet werden, dem dieselben Objekt-IDs (`u16ObjId`, `u16ObjIdA`) durchgereicht werden.
- **Generisch**: Die Adressen (`ID_WRITE_REMOTE`, `ID_WEB_READ`, `ID_SWITCH_REMOTE`) sind frei konfigurierbar, sodass die Subapp für verschiedene Zielmodule und Netztopologien einsetzbar ist.
- **Vier Quellen**: Im Vergleich zu `Softkey_Aux_IXA_TO_Remote_WRITE` wird eine vierte Quelle (Remote-Taster) unterstützt, was für Anwendungen mit mehreren Bedienorten wichtig ist.

Zustandsübersicht

Die Subapp implementiert eine **kombinatorische Logik** sie besitzt keinen internen Zustand. Der Ausgang (das Write-Kommando) ist eine reine Funktion der aktuellen Eingangswerte. Alle Eingänge werden kontinuierlich überwacht und jede Änderung kann sofort ein Kommando auslösen.

Anwendungsszenarien

- **Schnittverstellung**:
  - SoftKey „Aufnahme Heben/Senken“ auf Bedienmodul STG1
  - Auxiliary-Funktion (Joystick) auf STG1
  - Web-Override über vt-ui-mirror auf STG1
  - Physischer Taster S06/S07 auf STG4, per Remote-Subscribe (STG4_I2/I3_REMOTE) eingelesen
  - Aktor (z. B. Hydraulikventil) auf STG2, angesteuert über `ID_WRITE_REMOTE`
- **Alternative Bedienkonzepte** in Maschinen mit mehreren Bedienständen oder Fernbedienungen.

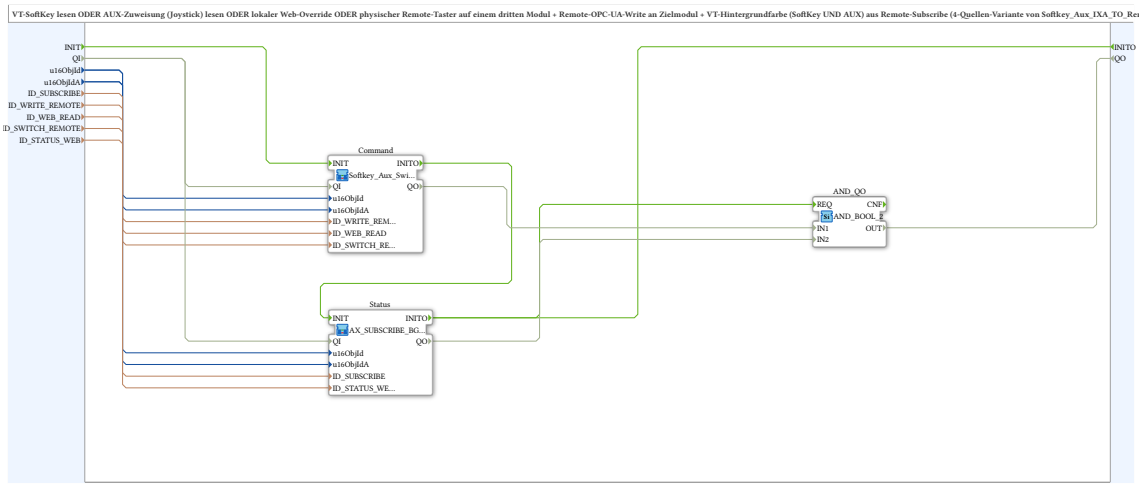
Vergleich mit ähnlichen Bausteinen

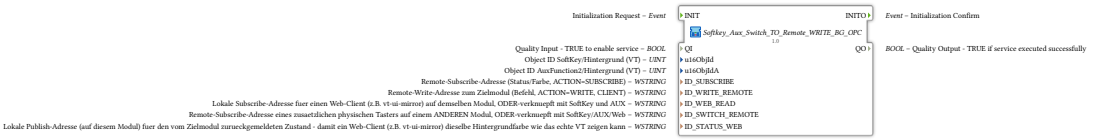
- **`Softkey_Aux_IXA_TO_Remote_WRITE`**: Kombiniert nur drei Quellen (SoftKey, AUX, Web) und verfügt über keinen zusätzlichen Remote-Taster-Eingang. Das hier dokumentierte SubApp-Element erweitert diese Logik um `ID_SWITCH_REMOTE` und verwendet einen vierfachen ODER-Baustein (`AX_OR_4`) anstelle eines dreifachen.
- **Direkte Mehrfach-Writer-Lösungen**: Wenn mehrere Clients direkt auf denselben Zielknoten schreiben, treten häufig unerwünschte Überlagerungen auf. Dieser Baustein konsolidiert alle Quellen an einer zentralen Stelle.

Fazit

`Softkey_Aux_Switch_TO_Remote_WRITE` ist ein leistungsfähiger und flexibler Baustein, der eine zuverlässige ODER-Verknüpfung von bis zu vier Bedienquellen ermöglicht und das Ergebnis als einzelnes OPC-UA-Write-Kommando an ein entferntes Zielmodul überträgt. Durch die zentrale Auswertung wird die Gefahr von Race-Conditions minimiert. Die fehlende Statusrückmeldung wird durch die Existenz separater Statusbausteine kompensiert, sodass sich der Baustein ideal für reine Steuerungsaufgaben eignet. Seine Einfachheit und Erweiterbarkeit machen ihn zu einer guten Wahl für verteilte Automatisierungssysteme.

Softkey_Aux_Switch_TO_Remote_WRITE_BG_OPC





• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsbaustein (SubApp) **Softkey_Aux_Switch_TO_Remote_WRITE_BG_OPC** ist eine 4-Quellen-Variante des Bausteins `Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC`. Er kombiniert die Zustände von vier verschiedenen Quellen – SoftKey, AUX-Joystick, lokalem Web-Override und einem zusätzlichen physischen Taster auf einem dritten Modul – und überträgt diese als OPC-UA-Write-Befehl an ein Zielmodul. Gleichzeitig wird der vom Zielmodul rückgemeldete Status über einen Remote-Subscribe-Kanal empfangen und als Hintergrundfarbe für die Visualisierung (SoftKey und AUX) verwendet sowie lokal für Web-Clients veröffentlicht.

Die SubApp besteht aus zwei internen Bausteinen:

- **Command** (`Softkey_Aux_Switch_TO_Remote_WRITE`): verarbeitet die vier Quellen und erzeugt den Remote-Write-Befehl.
- **Status** (`AX_SUBSCRIBE_BG3_WEB_OPC`): abonniert den Remote-Status und setzt die Hintergrundfarben sowie den lokalen Web-Republish um.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; startet die Init-Kette bei `Command.INIT`

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung; kommt von `Status.INITO`, dem Ende der Init-Kette

Daten-Eingänge

Name	Typ	Kommentar
`QI`	BOOL	Quality Input, Initialwert `TRUE`; wird an `Command.QI` und `Status.QI` weitergereicht
`u16ObjId`	UINT	Object ID SoftKey/Hintergrund (VT)
`u16ObjIdA`	UINT	Object ID AuxFunction2/Hintergrund (VT)
`ID_SUBSCRIBE`	WSTRING	Remote-Subscribe-Adresse (Status/Farbe, ACTION=SUBSCRIBE)
`ID_WRITE_REMOTE`	WSTRING	Remote-Write-Adresse zum Zielmodul (Befehl, ACTION=WRITE, CLIENT)
`ID_WEB_READ`	WSTRING	Lokale Subscribe-Adresse für einen Web-Client (z. B. vt-ui-mirror) auf demselben Modul, ODER-verknüpft mit SoftKey und AUX
`ID_SWITCH_REMOTE`	WSTRING	Remote-Subscribe-Adresse eines zusätzlichen physischen Tasters auf einem ANDEREN Modul, ODER-verknüpft mit SoftKey/AUX/Web
`ID_STATUS_WEB`	WSTRING	Lokale Publish-Adresse (auf diesem Modul) für den vom Zielmodul zurückgemeldeten Zustand – damit ein Web-Client (z. B. vt-ui-mirror) dieselbe Hintergrundfarbe wie das echte VT zeigen kann

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output, UND-Verknüpfung (`AND_QO`, `AND_BOOL_2`) aus `Command.QO` und `Status.QO`

Adapter

Keine.

Funktionsweise

Die SubApp leitet alle Eingabedaten an die beiden internen Bausteine weiter. Der **Command-Block** erhält die Objekt-IDs (`u16ObjId`, `u16ObjIdA`), die Remote-Write-Adresse (`ID_WRITE_REMOTE`), die lokale Web-Subscribe-Adresse (`ID_WEB_READ`) und die Remote-Switch-Adresse (`ID_SWITCH_REMOTE`). Er verknüpft die vier Quellen (SoftKey, AUX, Web, Remote-Taster) per ODER-Logik und sendet bei einer Zustandsänderung einen OPC-UA-Write-Befehl an das Zielmodul.

Der **Status-Block** erhält die Objekt-IDs (`u16ObjId`, `u16ObjIdA`), die Remote-Subscribe-Adresse (`ID_SUBSCRIBE`) und die lokale Publish-Adresse (`ID_STATUS_WEB`). Er abonniert den Status des Zielmoduls (z. B. aktive Hintergrundfarbe) und setzt entsprechend die Hintergrundfarben für SoftKey und AUX in der Visualisierung. Zusätzlich veröffentlicht er den empfangenen Status auf `ID_STATUS_WEB`, sodass Web-Clients die gleiche Hintergrundfarbe anzeigen können.

Initialisierung und Qualität: `INIT` → `Command.INIT`, `Command.INITO` → `Status.INIT`, `Status.INITO` → `INITO` und `AND_QO.REQ`. `QI` wird an `Command.QI` und `Status.QI` weitergereicht; `Command.QO` → `AND_QO.IN1`, `Status.QO` → `AND_QO.IN2`, `AND_QO.OUT` → `QO`.

Die beiden internen Bausteine arbeiten datenseitig parallel und unabhängig; abgesehen von der Init-Kette und der UND-Verknüpfung der Qualitätsausgänge besteht keine direkte Kommunikation zwischen ihnen.

Technische Besonderheiten

- **4-Quellen-Variante**: Im Unterschied zur 3-Quellen-Variante `Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC` wird hier ein zusätzlicher physischer Taster auf einem dritten Modul über `ID_SWITCH_REMOTE` mit einbezogen.
- **Nur Init-Ereignisse**: Der Baustein arbeitet datengetrieben; als Ereignisse besitzt er nur `INIT`/`INITO` für die Init-Kette, die internen Bausteine besitzen eigene Ereignisbehandlungen.
- **Wrapper-Struktur**: Als dünner Wrapper werden zwei vorhandene Bausteine kombiniert, um die gewünschte Funktionalität zu bündeln.
- **OPC-UA-Kommunikation**: Remote-Write und Remote-Subscribe werden über OPC-UA realisiert; die Adressen sind als WSTRING-Parameter konfigurierbar.

Zustandsübersicht

Da es sich um eine SubApp ohne eigenen Zustandsautomaten handelt, gibt es auf dieser Ebene keine definierten Zustände. Die Funktionalität wird durch die internen Bausteine erbracht. Der Command-Block besitzt vermutlich Zustände für die Quellenverarbeitung und das Senden des Remote-Writes, während der Status-Block Zustände für das Abonnement und die Aktualisierung der Hintergrundfarben hat. Diese internen Zustände sind von außen nicht sichtbar.

Anwendungsszenarien

- **Schnittverstellung**: Laut Kommentar ist die 4-Quellen-Variante für Funktionen gedacht, die einen zusätzlichen fest verbauten Taster auf einem dritten Modul besitzen, z. B. bei einer Schnittverstellung.
- **Maschinensteuerung**: Wenn eine Maschine über SoftKey, Joystick (AUX), Web-Override und einen entfernten physischen Taster bedient wird und der aktuelle Zustand (z. B. aktive Funktion, Hintergrundfarbe) auf einem Zielmodul angezeigt und überwacht werden muss.
- **Visualisierungs-Synchronisation**: Der Baustein stellt sicher, dass die Hintergrundfarben von SoftKey und AUX auf dem lokalen Bedienpanel und auf Web-Clients konsistent sind, basierend auf dem vom Zielmodul rückgemeldeten Status.

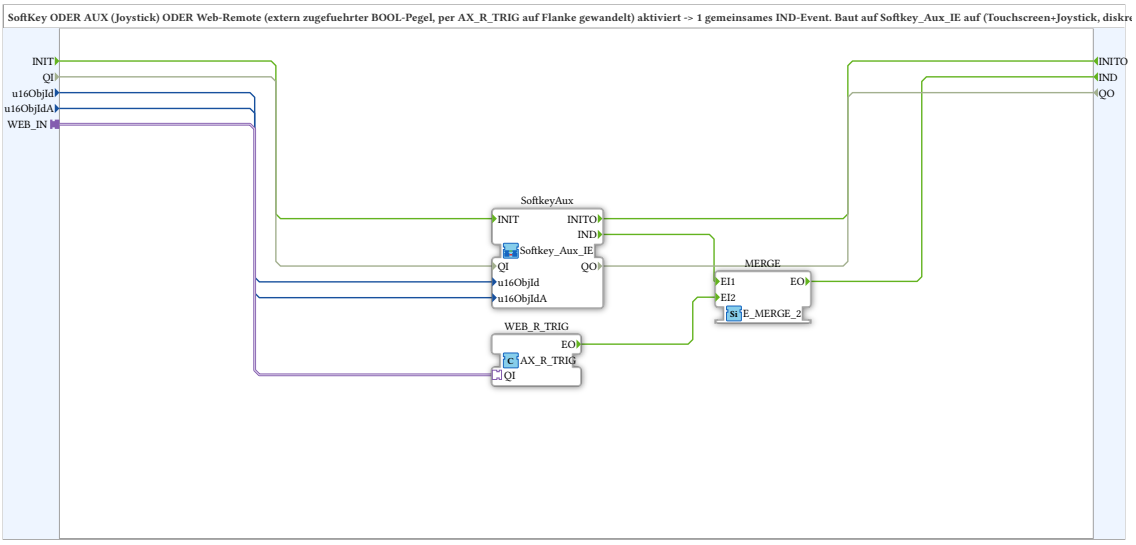
Vergleich mit ähnlichen Bausteinen

Baustein	Unterschied
`Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC`	3-Quellen-Variante: SoftKey, AUX, Web – ohne zusätzlichen externen Taster.
`Softkey_Aux_Switch_TO_Remote_WRITE_BG_OPC`	4-Quellen-Variante: zusätzlich ein physischer Taster auf einem dritten Modul (über `ID_SWITCH_REMOTE`).
`Softkey_Aux_Switch_TO_Remote_WRITE` (ohne BG)	Nur Befehlsteil, ohne Status- und Hintergrundfarben-Verarbeitung.

Fazit

Der Funktionsbaustein `Softkey_Aux_Switch_TO_Remote_WRITE_BG_OPC` erweitert die 3-Quellen-Logik um eine vierte Quelle (externer Taster) und kombiniert die Befehlsübertragung per Remote-Write mit der Status-Rückmeldung per Remote-Subscribe. Als modularer Wrapper nutzt er bewährte Bausteine und ermöglicht eine flexible Integration in Steuerungssysteme. Durch die klare Trennung von Befehl und Status eignet er sich besonders für Anwendungen, bei denen eine konsistente Visualisierung über mehrere Bedienorte hinweg erforderlich ist.

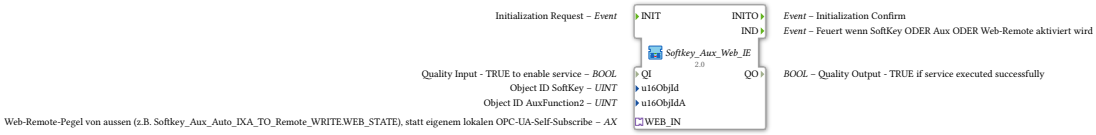
Softkey_Aux_Web_IE



• – ○ • – ○ • – ○ •

Einleitung

`Softkey_Aux_Web_IE` löst ein gemeinsames `IND`-Ereignis aus, wenn **SoftKey ODER AUX (Joystick) ODER Web-Remote** aktiviert wird. Der Web-Kanal ist ein extern zugeführter `BOOL`-Pegel (`AX`), der per `AX_R_TRIG` in eine Flanke gewandelt wird. Der Baustein baut auf `Softkey_Aux_IE` auf (Touchscreen + Joystick, diskreter Event-Kanal) und merged zusätzlich den Web-Kanal – ein Muster für „egal von welcher Quelle aus aktiviert, ein Ergebnis-Ereignis“, z. B. für die Override-Erkennung.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Softkey_Aux_Web_IE

- **Typ**: SubAppType
- **Verwendete interne Bausteine**:
  - **SoftkeyAux**: SubApp `MyLib::sys::Softkey_Aux_IE` – SoftKey oder AUX
  - **WEB_R_TRIG**: `adapter::events::unidirectional::AX_R_TRIG` – wandelt den Web-Pegel in eine steigende Flanke (`EO`)
  - **MERGE**: `iec61499::events::E_MERGE_2` – führt beide Ereignisse zusammen

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; geht an `SoftkeyAux.INIT`

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierung abgeschlossen; kommt von `SoftkeyAux.INITO`
- **IND**: Feuert, wenn SoftKey, Aux oder Web-Remote aktiviert wird

Daten-Eingänge

- **QI** (BOOL, Vorgabe TRUE): Quality Input - TRUE aktiviert den Dienst; geht an SoftkeyAux.QI
- **u16ObjId** (UINT): Object-ID des SoftKeys
- **u16ObjIdA** (UINT): Object-ID der AuxFunction2

Daten-Ausgänge

- **QO** (BOOL): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde; kommt von SoftkeyAux.QO

Adapter

- **WEB_IN** (Socket, adapter::types::unidirectional::AX): Web-Remote-Pegel von außen (z. B. Softkey_Aux_Auto_IXA_TO_Remote_WRITE.WEB_STATE) statt eines eigenen lokalen OPC-UA-Self-Subscribe

Programmablauf und Verbindungen

- u16ObjId ⟶ SoftkeyAux.u16ObjId; u16ObjIdA ⟶ SoftkeyAux.u16ObjIdA
- SoftkeyAux.IND ⟶ MERGE.EI1
- WEB_IN ⟶ WEB_R_TRIG.QI; WEB_R_TRIG.EO ⟶ MERGE.EI2
- MERGE.EO ⟶ IND
- Initialisierung: INIT ⟶ SoftkeyAux.INIT; SoftkeyAux.INITO ⟶ INITO
- Qualität: QI ⟶ SoftkeyAux.QI; SoftkeyAux.QO ⟶ QO

Technische Besonderheiten

- Version 2.0 (2026-09-16): ID-Parameter und interner Self-Subscribe (SUBSCRIBE_R_TRIG) entfernt, ersetzt durch den Socket WEB_IN plus AX_R_TRIG. Das behebt den Doppel-Self-Subscribe (FORTE: „Cannot add another one“), da der Web-Knoten bereits von der Kommando-Seite (Softkey_Aux_Auto_IXA_TO_Remote_WRITE.WEB_STATE) abonniert wird
- Es wird nur die steigende Flanke des Web-Pegels ausgewertet
- Version 1.0 (2026-09-11): initiale Implementierung

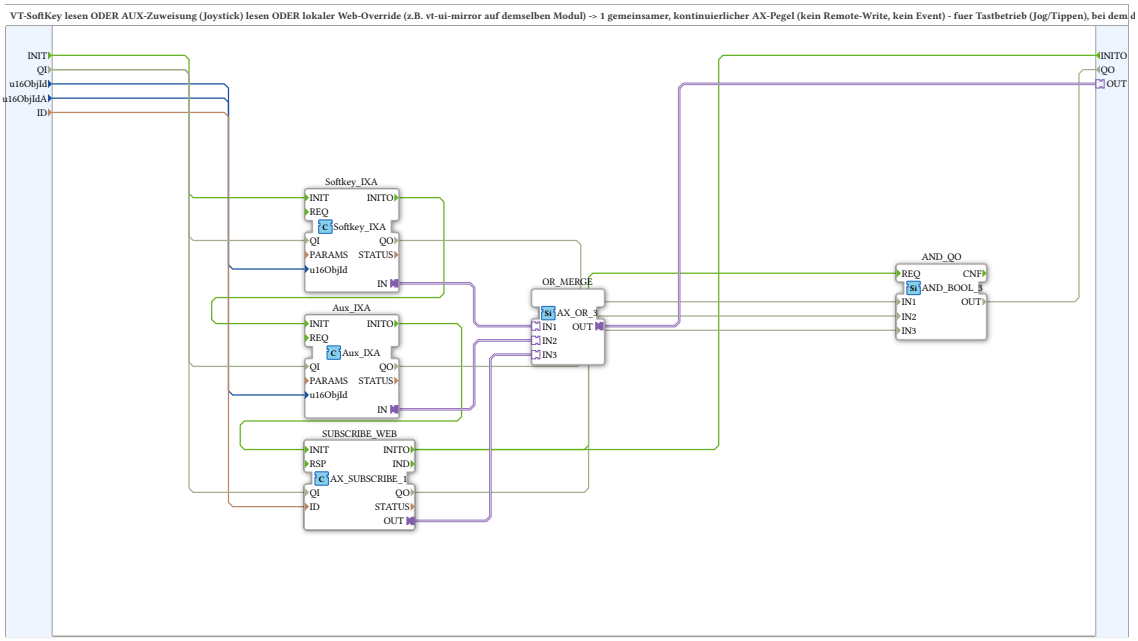
Anwendungsszenarien

- Override-Erkennung: Ereignis, sobald der Bediener über SoftKey, Joystick oder Web eingreift
- Zusammenarbeit mit Softkey_Aux_Auto_IXA_TO_Remote_WRITE (WEB_STATE)

Zusammenfassung

Softkey_Aux_Web_IE bündelt SoftKey, AUX und einen extern zugeführten Web-Pegel zu einem einzigen IND-Ereignis.

Softkey_Aux_Web_IXA



• — ○ • — ○ • — ○ •

Einleitung

`Softkey_Aux_Web_IXA` führt einen VT-SoftKey, eine AUX-Zuweisung (Joystick) und einen lokalen Web-Override (z. B. `vt-ui-mirror` auf demselben Modul) zu **einem gemeinsamen, kontinuierlichen AX-Pegel** zusammen – ohne Remote-Write und ohne Ereignis. Er ist für den Tastbetrieb (Jog/Tippen) gedacht, bei dem der Pegel selbst (nicht eine Flanke) weiterverarbeitet werden soll, z. B. als `G`-Eingang eines `AX_AX_SEL_AR` oder als `DI1` einer GreenWhiteBackground-Rückmeldung. Er entspricht der Kommandohälfte von `Softkey_Aux_IXA_TO_Remote_WRITE`, liefert aber `OUT` statt eines Remote-Writes.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung

Außer `INITO` gibt es keine Ereignis-Ausgänge – Nutzereignisse laufen über den Adapter-Plug.

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input – TRUE gibt den Dienst frei
- **u16ObjId** (`UINT`, Vorgabe `ID_NULL`): Object-ID des SoftKeys (VT)
- **u16ObjIdA** (`UINT`, Vorgabe `ID_NULL`): Object-ID der AuxFunction2 (VT)
- **ID** (`WSTRING`): Lokale Subscribe-Adresse für einen Web-Client (z. B. `vt-ui-mirror`) auf demselben Modul, ODER-verknüpft mit SoftKey und AUX

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output – UND aus `Softkey_IXA.QO`, `Aux_IXA.QO` und `SUBSCRIBE_WEB.QO`

Adapter

- **OUT** (Plug, `adapter::types::unidirectional::AX`): SoftKey ODER Aux ODER Web-Override, kontinuierlicher Pegel (`TRUE`, solange gehalten/aktiv)

Verwendete Funktionsbausteine (FBs)

- **Softkey_IXA**: `isobus::UT::io::Softkey::Softkey_IXA`
- **Aux_IXA**: `isobus::UT::io::Auxiliary::IN::Aux_IXA`
- **SUBSCRIBE_WEB**: `adapter::net::AX_SUBSCRIBE_1`
- **OR_MERGE**: `adapter::booleanOperators::AX_OR_3`
- **AND_QO**: `iec61131::booleanOperators::AND_BOOL_3`

Programmablauf und Verbindungen

- `u16ObjId` ⟶ `Softkey_IXA.u16ObjId`; `u16ObjIdA` ⟶ `Aux_IXA.u16ObjId`; `ID` ⟶ `SUBSCRIBE_WEB.ID`
- `Softkey_IXA.IN` ⟶ `OR_MERGE.IN1`
- `Aux_IXA.IN` ⟶ `OR_MERGE.IN2`
- `SUBSCRIBE_WEB.OUT` ⟶ `OR_MERGE.IN3`
- `OR_MERGE.OUT` ⟶ `OUT`
- `INIT` ⟶ `Softkey_IXA.INIT`; `Softkey_IXA.INITO` ⟶ `Aux_IXA.INIT`; `Aux_IXA.INITO` ⟶ `SUBSCRIBE_WEB.INIT`; `SUBSCRIBE_WEB.INITO` ⟶ `INITO` (und ⟶ `AND_QO.REQ`)
- `QI` ⟶ `Softkey_IXA.QI`, `Aux_IXA.QI` und `SUBSCRIBE_WEB.QI`
- `Softkey_IXA.QO` ⟶ `AND_QO.IN1`; `Aux_IXA.QO` ⟶ `AND_QO.IN2`; `SUBSCRIBE_WEB.QO` ⟶ `AND_QO.IN3`; `AND_QO.OUT` ⟶ `QO`

Technische Besonderheiten

- Kontinuierlicher Pegel statt Flanke: `OUT` bleibt `TRUE`, solange mindestens eine Quelle aktiv ist.
- Kein Remote-Write und kein Nutzereignis-Ausgang (nur `INITO`).

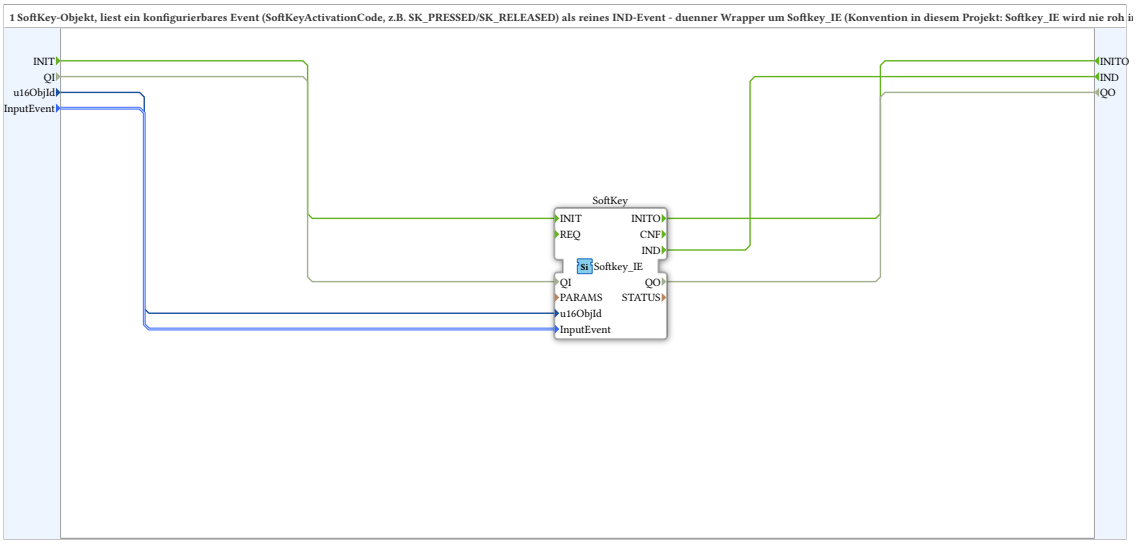
Anwendungsszenarien

- Tastbetrieb (Jog/Tippen), bei dem der Pegel an einen Selektor oder eine Hintergrundfarben-Rückmeldung gehen soll

Fazit

`Softkey_Aux_Web_IXA` verodert SoftKey, Aux und Web-Override zu einem AX-Pegel.

Softkey_IE_1

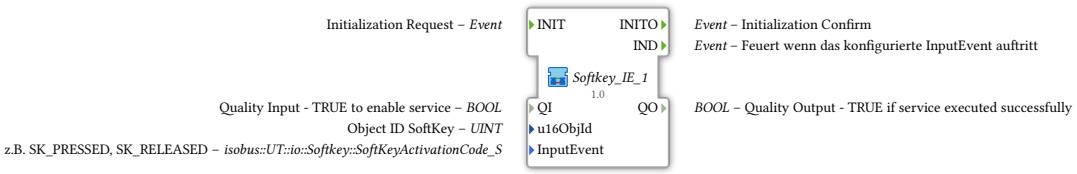


• – ◦ • – ◦ • – ◦ •

Einleitung

`Softkey_IE_1` liest ein SoftKey-Objekt und gibt ein konfigurierbares Ereignis (`SoftKeyActivationCode`, z. B. `SK_PRESSED`/`SK_RELEASED`) als reines `IND`-Ereignis aus. Der Baustein ist ein dünner Wrapper um

`isobus::UT::io::Softkey::Softkey_IE`, analog zu `Aux_IE_1` und `Button_IE_1`. Konvention in diesem Projekt: `Softkey_IE` wird nie roh inline verwendet, sondern immer gewrappt.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialization Request

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialization Confirm
- **IND**: feuert, wenn das konfigurierte `InputEvent` auftritt

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input, TRUE aktiviert den Dienst
- **u16ObjId** (UINT): Object ID des SoftKeys (Default `ID_NULL`)
- **InputEvent** (`isobus::UT::io::Softkey::SoftKeyActivationCode_S`): z. B. `SK_PRESSED`, `SK_RELEASED` (Default `Invalid`)

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output, direkt von `SoftKey.QO`

Adapter

Es werden keine Adapter verwendet.

Funktionsweise

Das Netzwerk enthält genau eine Instanz `SoftKey` des Typs `isobus::UT::io::Softkey::Softkey_IE` (`QI` kommt vom SubApp-Eingang `QI`). `u16ObjId` und `InputEvent` werden unverändert auf die gleichnamigen Eingänge der Instanz geführt, und deren Ereignis `IND` wird direkt als `IND` der SubApp ausgegeben. Zusätzlich gilt: `INIT` ⟶ `SoftKey.INIT`, `SoftKey.INITO` ⟶ `INITO`, `QI` ⟶ `SoftKey.QI`, `SoftKey.QO` ⟶ `QO`.

Technische Besonderheiten

- Reine Durchreichung ohne eigene Logik oder Zustand
- `QI` ist als Eingang herausgeführt (Vorgabe `TRUE`) und `QO` kommt direkt von `SoftKey.QO`; unbeschaltet ist der Baustein also aktiv
- Weil `InputEvent` ein Parameter ist, kann dieselbe SubApp für unterschiedliche Ereignisarten (z. B. Drücken oder Loslassen) verwendet werden

Anwendungsszenarien

- Einlesen einer einzelnen VT-Bedienung als Ereignis, ohne den rohen IE-Baustein direkt im Netzwerk zu verwenden

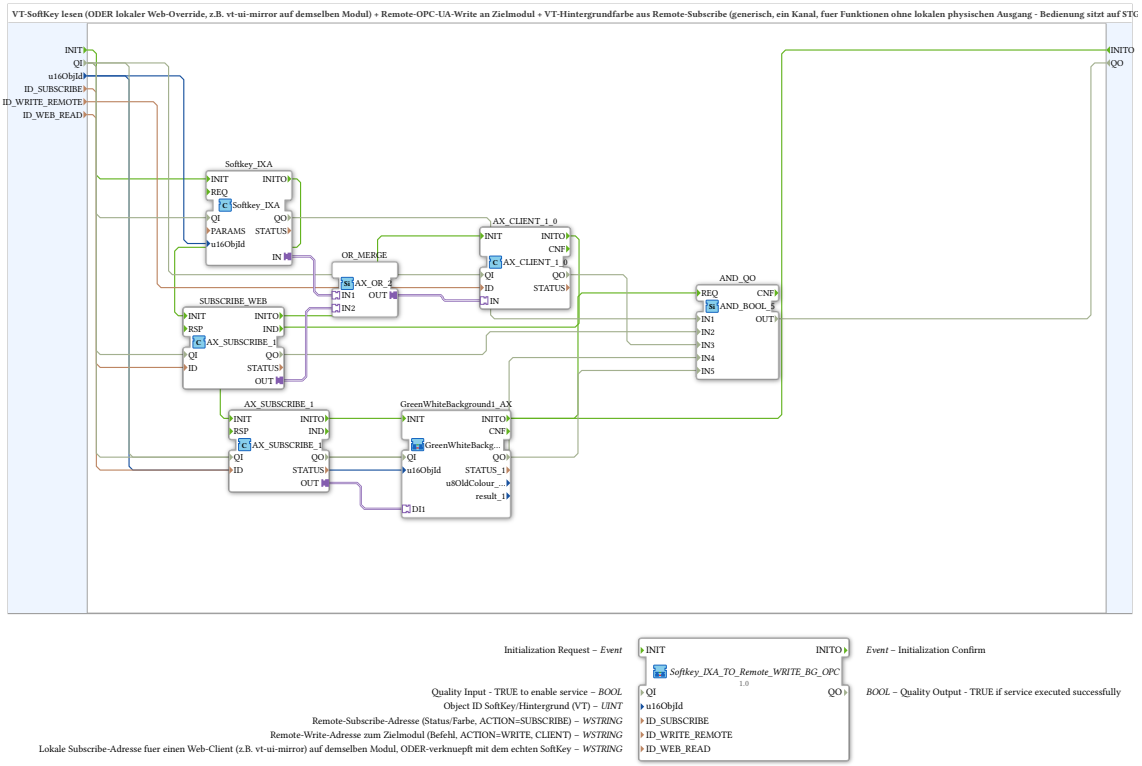
⚖️ Vergleich mit ähnlichen Bausteinen

Gleichartige Wrapper sind `Aux_IE_1` und `Button_IE_1` sie unterscheiden sich nur im VT-Objekttyp und im Aufzählungstyp des `InputEvent`.

Fazit

`Softkey_IE_1` ist ein minimaler Wrapper, der ein konfigurierbares SoftKey-Ereignis als `IND` bereitstellt.

Softkey_IXA_TO_Remote_WRITE_BG_OPC



• — ○ • — ○ • — ○ •

Einleitung

Dieser Baustein realisiert eine Fernbedienungs-Funktion für einen SoftKey, der auf einem VT-Steuergerät (STG1) sitzt, während der eigentliche Aktor auf einem anderen Modul ohne eigene VT-Anbindung angeordnet ist. Der lokale SoftKey wird gelesen und per OPC-UA als Remote-Write an das Zielmodul gesendet. Der tatsächliche Zustand des Aktors wird per Remote-Subscribe zurückgeholt und als Hintergrundfarbe direkt am SoftKey visualisiert. Zusätzlich kann ein lokaler Web-Client (z. B. ein vt-ui-mirror auf demselben Modul) denselben Befehl auslösen; beide Quellen werden per ODER-Verknüpfung kombiniert.

Schnittstellenstruktur

Die SubApp besitzt die Initialisierungs-Ereignisse `INIT`/`INITO`, die Status-Anschlüsse `QI`/`QO` und Dateneingänge. Es gibt keine Adapter als externe Schnittstellen.

Ereignis-Eingänge

- **INIT** (`EInit`): Initialization Request

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialization Confirm

Daten-Eingänge

Name	Datentyp	Beschreibung
`QI`	BOOL	Quality Input - TRUE, um den Dienst freizugeben. Initialwert: `TRUE`
`u16ObjId`	UINT	Object-ID des SoftKeys bzw. des Hintergrund-Objekts (VT). Initialwert: `ID_NULL`
`ID_SUBSCRIBE`	WSTRING	Remote-Subscribe-Adresse (Status/Farbe, ACTION=SUBSCRIBE)
`ID_WRITE_REMOTE`	WSTRING	Remote-Write-Adresse zum Zielmodul (Befehl, ACTION=WRITE, CLIENT)

ID_WEB_READ	WSTRING	Lokale Subscribe-Adresse für einen Web-Client (z. B. vt-ui-mirror) auf demselben Modul, ODER-verknüpft mit dem echten SoftKey

Daten-Ausgänge

Name	Datentyp	Beschreibung
QO	BOOL	Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde (UND von Softkey_IXA, SUBSCRIBE_WEB, AX_CLIENT_1_0, AX_SUBSCRIBE_1 und GreenWhiteBackground1_AX)

Adapter

Keine.

Funktionsweise

Die SubApp besteht aus mehreren Funktionsblöcken, die über Adapterverbindungen und Datenverbindungen zusammenwirken:

1. **Softkey_IXA** (Typ isobus::UT::io::Softkey::Softkey_IXA): Liest den lokalen SoftKey-Zustand anhand der übergebenen u16ObjId. Der Zustand wird über seinen Adapter-Ausgang IN bereitgestellt.
2. **SUBSCRIBE_WEB** (Typ adapter::net::AX_SUBSCRIBE_1): Abonniert die Adresse aus ID_WEB_READ. Der empfangene Wert (z. B. vom Web-Client) wird über den Adapter-Ausgang OUT geliefert.
3. **OR_MERGE** (Typ adapter::booleanOperators::AX_OR_2): Verknüpft die Signale von Softkey_IXA.IN (Eingang IN1) und SUBSCRIBE_WEB.OUT (Eingang IN2) logisch per ODER. Das Ergebnis erscheint am Ausgang OUT.
4. **AX_CLIENT_1_0** (Typ adapter::net::AX_CLIENT_1_0): Nimmt das ODER-Signal entgegen und führt über die Adresse ID_WRITE_REMOTE einen OPC-UA-Write-Befehl auf dem Zielmodul aus.
5. **AX_SUBSCRIBE_1** (Typ adapter::net::AX_SUBSCRIBE_1): Abonniert die Adresse ID_SUBSCRIBE, über die der tatsächliche Aktorzustand vom Zielmodul zurückgemeldet wird. Der Ausgang OUT liefert den binären Status.
6. **GreenWhiteBackground1_AX** (SubApp vom Typ MyLib::sys::GreenWhiteBackground1_AX): Erhält den Status vom Subscribe-Ausgang an ihrem Eingang DI1 und setzt die VT-Hintergrundfarbe des SoftKeys entsprechend: grün für aktiv, weiß für inaktiv.
7. **AND_QO** (Typ iec61131::booleanOperators::AND_BOOL_5): Verknüpft die QO-Ausgänge von Softkey_IXA, SUBSCRIBE_WEB, AX_CLIENT_1_0, AX_SUBSCRIBE_1 und GreenWhiteBackground1_AX per UND zum SubApp-Ausgang QO.

Die Adapterverbindungen gewährleisten den Datenfluss:
Softkey_IXA.IN ⟶ OR_MERGE.IN1,
SUBSCRIBE_WEB.OUT ⟶ OR_MERGE.IN2,
OR_MERGE.OUT ⟶ AX_CLIENT_1_0.IN,
AX_SUBSCRIBE_1.OUT ⟶ GreenWhiteBackground1_AX.DI1.

Initialisierung und Status:
INIT ⟶ Softkey_IXA.INIT, Softkey_IXA.INITO ⟶ SUBSCRIBE_WEB.INIT, SUBSCRIBE_WEB.INITO ⟶ AX_CLIENT_1_0.INIT, AX_CLIENT_1_0.INITO ⟶ AX_SUBSCRIBE_1.INIT, AX_SUBSCRIBE_1.INITO ⟶ GreenWhiteBackground1_AX.INIT, GreenWhiteBackground1_AX.INITO ⟶ INITO und ⟶ AND_QO.REQ.
QI ⟶ Softkey_IXA.QI, SUBSCRIBE_WEB.QI, AX_CLIENT_1_0.QI, AX_SUBSCRIBE_1.QI und GreenWhiteBackground1_AX.QI; deren QO ⟶ AND_QO.IN1..IN5; AND_QO.OUT ⟶ QO.

Technische Besonderheiten

- Der Baustein verwendet isolierte Funktionsblöcke aus den Bibliotheken isobus, adapter und MyLib::sys.
- Der Initialwert von u16ObjId ist ID_NULL, eine im Compiler-Import definierte Konstante.
- Die OPC-UA-Kommunikation ist zweigeteilt: Ein Write-Kanal für den Befehl und ein Subscribe-Kanal für das Rückfeedback. Dadurch wird eine unabhängige Status-Aktualisierung ermöglicht.
- Die SubApp ist generisch für **genau einen** Kanal ausgelegt; durch Konfiguration verschiedener Adressen kann sie für unterschiedliche Objekte verwendet werden.

- Es existieren Varianten für Auxiliary-Funktionen (Joystick) sowie für VT-Buttons, die in der Baustein-Dokumentation erwähnt werden.

Zustandsübersicht

Die SubApp besitzt keinen expliziten Zustandsautomaten. Aus Sicht des Zielsystems gibt es jedoch zwei stabile Zustände, die über die Hintergrundfarbe visualisiert werden:

- **Aktiv** (Grün): Der Aktor ist eingeschaltet – der über `ID_SUBSCRIBE` empfangene Status ist `TRUE`.
- **Inaktiv** (Weiß): Der Aktor ist ausgeschaltet – der Status ist `FALSE`.

Der Übergang zwischen diesen Zuständen wird durch den ODER-verknüpften Befehl (SoftKey oder Web-Client) ausgelöst. Der Rückkanal über das Subscribe-Abonnement aktualisiert die Anzeige unmittelbar nach dem Write-Befehl.

Anwendungsszenarien

- **Fernbedienung eines Aktors**: Der Bediener drückt auf einem VT-Steuergerät (STG1) einen SoftKey und erhält über die Hintergrundfarbe eine visuelle Rückmeldung des Aktorzustands, ohne dass ein weiteres Display am Zielmodul erforderlich ist.
- **Einbindung eines Web-Clients**: Ein auf demselben Steuergerät laufender Web-Client (z. B. Spiegel der Bedienoberfläche) kann dieselbe Funktion über `ID_WEB_READ` auslösen; die ODER-Verknüpfung stellt sicher, dass beide Bedienwege funktionieren.
- **Modul ohne eigene VT-Anbindung**: Wenn der Aktor auf einem Modul sitzt, das weder Display noch Tastatur hat, wird die Bedienung über OPC-UA realisiert – die Kommunikation läuft vollständig über die konfigurierten Adressen.

Vergleich mit ähnlichen Bausteinen

Die SubApp `Softkey_IXA_TO_Remote_WRITE_BG_OPC` ist eine spezialisierte Variante in einer Produktfamilie. In der internen Dokumentation werden zwei verwandte Typen erwähnt:

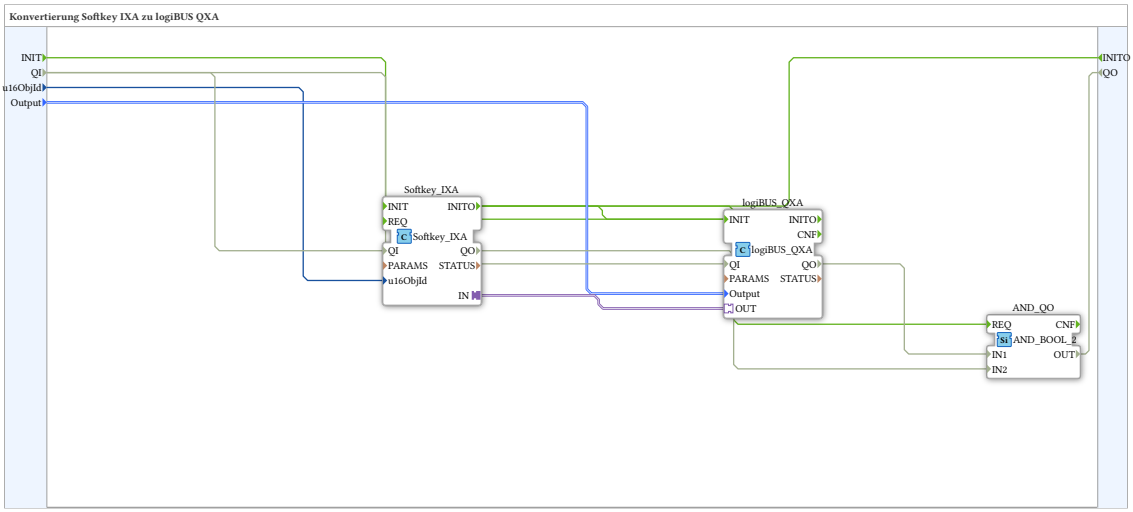
- **Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC**: Unterstützt zusätzlich eine dritte Quelle (z. B. eine Auxiliary-Funktion/Joystick). Der logische Aufbau ist ähnlich, jedoch wird `GreenWhiteBackground3_AX` anstelle von `GreenWhiteBackground1_AX` verwendet.
- **Button_IXA_TO_Remote_WRITE_BG_OPC**: Für einen regulären VT-Button anstelle eines Soft-Keys; die Grundstruktur bleibt gleich.

Diese Varianten unterscheiden sich hauptsächlich in der Anzahl der möglichen Eingabequellen und in der Art des Bedienelements.

Fazit

Die SubApp `Softkey_IXA_TO_Remote_WRITE_BG_OPC` bietet eine kompakte und generische Lösung, um einen SoftKey-basierten Befehl über OPC-UA an ein entferntes Modul zu senden und den Zustand des Aktors direkt am SoftKey sichtbar zu machen. Durch die ODER-Verknüpfung mit einem Web-Client wird die Bedienung flexibel, und die Trennung von Befehls- und Statuskanal erhöht die Zuverlässigkeit. Sie ist besonders für verteilte Systeme geeignet, bei denen die Bedienoberfläche und der Aktor getrennt sind.

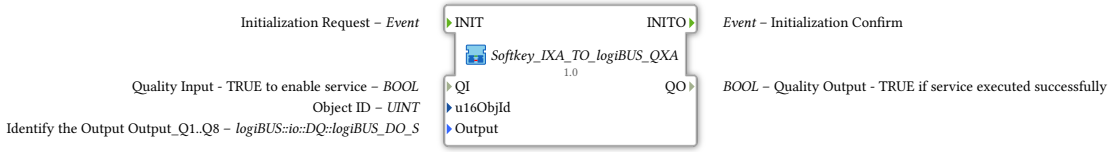
Softkey_IXA_TO_logiBUS_QXA



• – ○ • – ○ • – ○ •

Einleitung

`Softkey_IXA_TO_logiBUS_QXA` verbindet einen VT-Softkey (`Softkey_IXA`) direkt mit einem physischen digitalen Ausgang (`logiBUS_QXA`) — funktional identisch zu `Button_IXA_TO_logiBUS_QXA`, aber für Softkeys statt Buttons.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Softkey_IXA_TO_logiBUS_QXA

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **Softkey_IXA**: `isobus::UT::io::Softkey::Softkey_IXA` — VT-Softkey-Adapter.
  - **logiBUS_QXA**: `logiBUS::io::DQ::logiBUS_QXA` — physischer digitaler Ausgang.
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_2` — verknüpft `logiBUS_QXA.QO` und `Softkey_IXA.QO` zum SubApp-Ausgang `QO`.
- **Schnittstelle**: Ereigniseingang `INIT`, Ereignisausgang `INITO`, Dateneingang `QI` (Vorgabe `TRUE`, an beide FBs durchgereicht), Datenausgang `QO`.
- **Funktionsweise**: Direkte Adapterverbindung `Softkey_IXA.IN` ⟶ `logiBUS_QXA.OUT`, keine Zwischenlogik (außer der `AND_QO`-Verknüpfung der Qualitätssignale).

Programmablauf und Verbindungen

1. `u16ObjId` ⟶ `Softkey_IXA.u16ObjId`; `Output` ⟶ `logiBUS_QXA.Output`.
2. `Softkey_IXA.IN` ⟶ `logiBUS_QXA.OUT`.
3. Initialisierung: `INIT` ⟶ `logiBUS_QXA.INIT`; `logiBUS_QXA.INITO` ⟶ `Softkey_IXA.INIT`; `Softkey_IXA.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`.
4. Qualität: `QI` ⟶ `logiBUS_QXA.QI` und `Softkey_IXA.QI`; `logiBUS_QXA.QO` ⟶ `AND_QO.IN1`, `Softkey_IXA.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`.

Anwendungsszenarien

- Minimaler Softkey-zu-Ausgang-Baustein ohne Statusanzeige oder Fernbedienung.

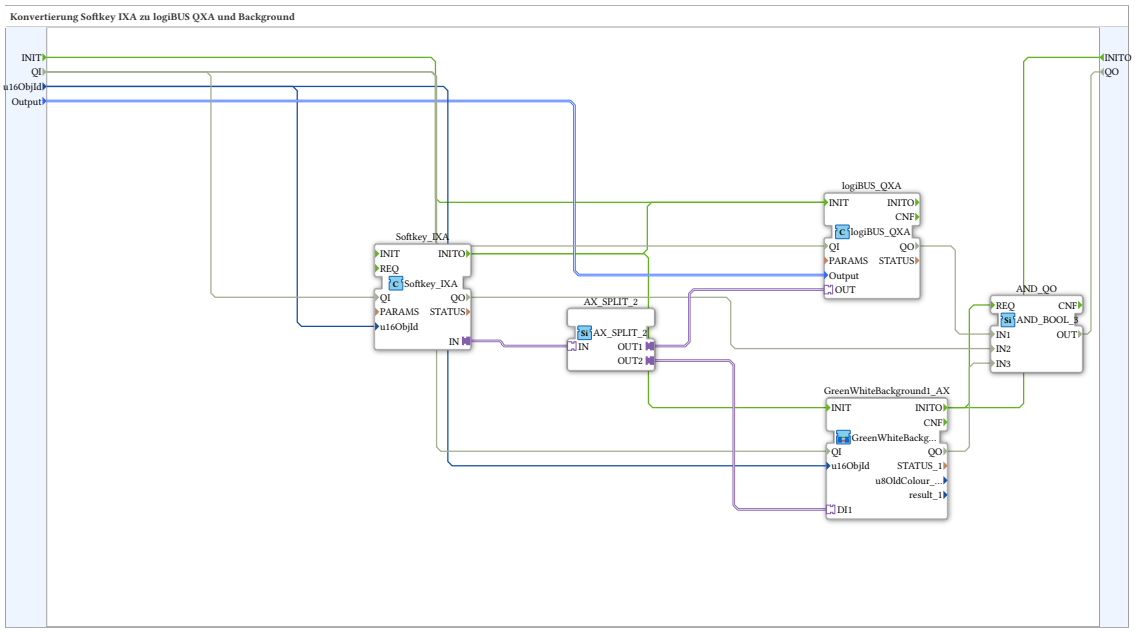
Zusammenfassung

Softkey-Pendant zu `Button_IXA_TO_logiBUS_QXA` — identisches Muster, anderer Tastertyp.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

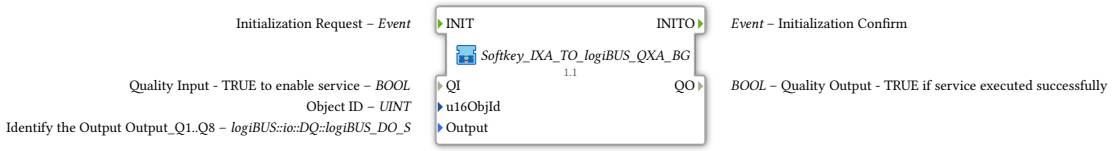
Softkey_IXA_TO_logiBUS_QXA_BG



• – ○ • – ○ • – ○ •

Einleitung

`Softkey_IXA_TO_logiBUS_QXA_BG` erweitert `Softkey_IXA_TO_logiBUS_QXA` um eine VT-Statusfarbe — das Softkey-Pendant zu `Button_IXA_TO_logiBUS_QXA_BG`.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Softkey_IXA_TO_logiBUS_QXA_BG

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **Softkey_IXA**: `isobus::UT::io::Softkey::Softkey_IXA` — VT-Softkey-Adapter.
  - **logiBUS_QXA**: `logiBUS::io::DQ::logiBUS_QXA` — physischer digitaler Ausgang.
  - **AX_SPLIT_2**: `adapter::events::unidirectional::AX_SPLIT_2` — verzweigt das Softkey-Signal.
  - **GreenWhiteBackground1_AX** (SubApp, `MyLib::sys`): setzt die VT-Statusfarbe (siehe Background-Farbbausteine).
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_3` — verknüpft `logiBUS_QXA.QO`, `Softkey_IXA.QO` und `GreenWhiteBackground1_AX.QO` zum Ausgang `QO`.
- **Funktionsweise**: `Softkey_IXA.IN` wird über `AX_SPLIT_2` sowohl an `logiBUS_QXA.OUT` als auch an `GreenWhiteBackground1_AX.DI1` verteilt. Die SubApp-Schnittstelle besitzt außerdem die Ereignisse `INIT`/`INITO` sowie `QI` (BOOL, Initialwert `TRUE`) und `QO` (BOOL).

Programmablauf und Verbindungen

1. `u16ObjId` ⟶ `Softkey_IXA.u16ObjId` und `GreenWhiteBackground1_AX.u16ObjId`; `Output` ⟶ `logiBUS_QXA.Output`.
2. `Softkey_IXA.IN` ⟶ `AX_SPLIT_2.IN` ⟶ `AX_SPLIT_2.OUT1` ⟶ `logiBUS_QXA.OUT`, `AX_SPLIT_2.OUT2` ⟶ `GreenWhiteBackground1_AX.DI1`.

3. Initialisierungskette: `INIT` ⟶ `logiBUS_QXA.INIT`, `logiBUS_QXA.INITO` ⟶ `Softkey_IXA.INIT`, `Softkey_IXA.INITO` ⟶ `GreenWhiteBackground1_AX.INIT`, `GreenWhiteBackground1_AX.INITO` ⟶ `INITO` und `AND_QO.REQ`.
4. Qualität: `QI` ⟶ `logiBUS_QXA.QI`, `Softkey_IXA.QI` und `GreenWhiteBackground1_AX.QI`; deren `QO`-Ausgänge ⟶ `AND_QO.IN1` bis `AND_QO.IN3` (in dieser Reihenfolge); `AND_QO.OUT` ⟶ `QO`.

Anwendungsszenarien

- VT-Softkey mit direkter optischer Rückmeldung, ohne OPC-UA.

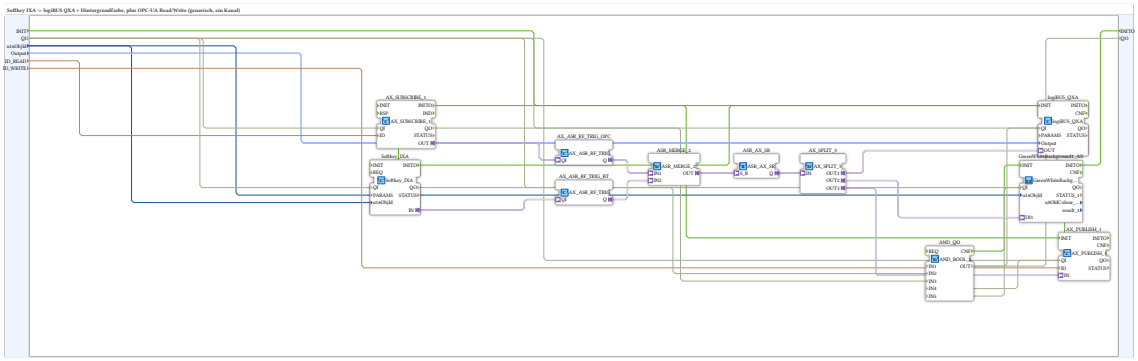
Zusammenfassung

Softkey-Pendant zu `Button_IXA_TO_logiBUS_QXA_BG` — identisches Muster, anderer Tastertyp.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

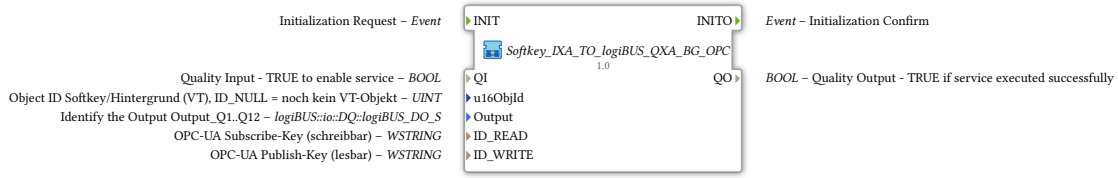
Softkey_IXA_TO_logiBUS_QXA_BG_OPC



• – ○ • – ○ • – ○ •

Einleitung

`Softkey_IXA_TO_logiBUS_QXA_BG_OPC` ist das SoftKey-Pendant zu `Button_IXA_TO_logiBUS_QXA_BG_OPC_LATCHING`: Ein VT-SoftKey und ein OPC-UA-Remote-Kommando werden je über eine Flankenerkennung in Set/Reset-Ereignisse gewandelt und in einem gemeinsamen Latch zusammengeführt. Das ist **kein** echtes Klick-Toggle: Drücken löst `SET` aus, Loslassen `RESET` — bei nur einer aktiven Quelle also rein tastendes Verhalten. Der Unterschied zu einer einfachen ODER-Verknüpfung zeigt sich erst, wenn sich SoftKey und OPC-UA-Kommando zeitlich überlappen: Es gewinnt immer die zuletzt eingetroffene Flanke, unabhängig von der Quelle (Last-Wins), statt dass der Ausgang so lange EIN bleibt, wie irgendeine der beiden Quellen aktiv ist.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Softkey_IXA_TO_logiBUS_QXA_BG_OPC

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **Softkey_IXA**: `isobus::UT::io::Softkey::Softkey_IXA` — VT-SoftKey, `u16ObjId` identifiziert den SoftKey.
  - **AX_ASR_RF_TRIG_BT** / **AX_ASR_RF_TRIG_OPC**: je `adapter::events::unidirectional::AX_ASR_RF_TRIG` — wandeln steigende/fallende Flanke des SoftKeys bzw. des OPC-UA-Subscribe-Werts in ein Set/Reset-Ereignispaar (ASR).

  - **ASR_MERGE_2**: `adapter::events::unidirectional::ASR_MERGE_2` — führt die beiden ASR-Quellen (SoftKey, OPC-UA) zusammen.
  - **ASR_AX_SR**: `adapter::events::unidirectional::ASR_AX_SR` — Set/Reset-Latch, bildet aus dem zusammengeführten ASR-Signal den gelatchten Ausgangszustand.
  - **AX_SPLIT_3**: `adapter::events::unidirectional::AX_SPLIT_3` — verteilt den Latch-Ausgang auf drei Ziele.
  - **logiBUS_QXA**: `logiBUS::io::DQ::logiBUS_QXA` — physischer digitaler Ausgang.
  - **GreenWhiteBackground1_AX** (SubApp, `MyLib::sys`): VT-Hintergrundfarbe passend zum Ausgangszustand.
  - **AX_SUBSCRIBE_1** / **AX_PUBLISH_1**: je `adapter::net::AX_SUBSCRIBE_1/AX_PUBLISH_1` — OPC-UA-Lese-/Schreibzugriff.
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_5` — verknüpft die `QO` von `logiBUS_QXA`, `Softkey_IXA`, `AX_SUBSCRIBE_1`, `AX_PUBLISH_1` und `GreenWhiteBackground1_AX`.
- **Funktionsweise**: Identisch zur Latching-Struktur von `Button_IXA_TO_logiBUS_QXA_BG_OPC_LATCHING`, jedoch mit `Softkey_IXA` statt `Button_IXA` als lokale Bedienquelle.

Programmablauf und Verbindungen

1. `Softkey_IXA.IN` (Adapter) → `AX_ASR_RF_TRIG_BT.QI` → `AX_ASR_RF_TRIG_BT.Q` → `ASR_MERGE_2.IN2`.
2. `AX_SUBSCRIBE_1.OUT` → `AX_ASR_RF_TRIG_OPC.QI` → `AX_ASR_RF_TRIG_OPC.Q` → `ASR_MERGE_2.IN1`.
3. `ASR_MERGE_2.OUT` → `ASR_AX_SR.S_R`.
4. `ASR_AX_SR.Q` → `AX_SPLIT_3.IN` → `AX_SPLIT_3.OUT1` → `logiBUS_QXA.OUT`, `AX_SPLIT_3.OUT2` → `GreenWhiteBackground1_AX.DI1`, `AX_SPLIT_3.OUT3` → `AX_PUBLISH_1.IN`.
5. Initialisierung: Ereigniseingang `INIT` → `logiBUS_QXA.INIT`, `logiBUS_QXA.INITO` → `Softkey_IXA.INIT`, `Softkey_IXA.INITO` → `AX_SUBSCRIBE_1.INIT`, `AX_SUBSCRIBE_1.INITO` → `AX_PUBLISH_1.INIT` (ausgeblendete Verbindung), `AX_PUBLISH_1.INITO` → `GreenWhiteBackground1_AX.INIT`, `GreenWhiteBackground1_AX.INITO` → Ereignisausgang `INITO` und → `AND_QO.REQ`.
6. Parameter: `u16ObjId` → `Softkey_IXA.u16ObjId` und `GreenWhiteBackground1_AX.u16ObjId`; `Output` → `logiBUS_QXA.Output`; `ID_READ` → `AX_SUBSCRIBE_1.ID`; `ID_WRITE` → `AX_PUBLISH_1.ID`.
7. Qualität: Dateneingang `QI` (`BOOL`, Default `TRUE`) → `logiBUS_QXA.QI`, `Softkey_IXA.QI`, `AX_SUBSCRIBE_1.QI`, `AX_PUBLISH_1.QI` und `GreenWhiteBackground1_AX.QI`; deren `QO` → `AND_QO.IN1`..`IN5`; `AND_QO.OUT` → Datenausgang `QO`.

Technische Besonderheiten

- **Last-Wins über ASR-Latch, kein echtes Toggle**: Wie bei `Button_IXA_TO_logiBUS_QXA_BG_OPC_LATCHING` ist das Verhalten bei nur einer aktiven Quelle rein tastend (EIN nur solange gedrückt). Der Unterschied zu einer tastenden ODER-Verschaltung zeigt sich erst, wenn sich beide Quellen überlappen: die zuletzt eingetroffene Flanke (Set oder Reset, unabhängig von der Quelle) bestimmt den Zustand.
- **Zwei unabhängige Last-Wins-Quellen**: SoftKey und OPC-UA-Kommando liefern je eigenständig Set-/Reset-Ereignisse; `ASR_MERGE_2` führt beide zu einem gemeinsamen Latch-Eingang zusammen, ohne eine Quelle zu priorisieren.

Anwendungsszenarien

- Digitale Ausgänge mit zwei unabhängigen Schaltquellen (SoftKey UND Remote-OPC-UA-Kommando), bei denen die zuletzt eingetroffene Aktion gewinnen soll, statt dass eine dauerhaft aktive Quelle die andere blockiert — mit VT-Statusanzeige und OPC-UA-Rückmeldung.

Vergleich mit ähnlichen Bausteinen

Strukturell identisch zu `Button_IXA_TO_logiBUS_QXA_BG_OPC_LATCHING`, nur mit `Softkey_IXA` statt `Button_IXA` als lokale Bedienquelle — inklusive derselben Last-Wins-Charakteristik (nicht Klick-Toggle). Gegenüber dem einfacheren `Softkey_IXA_TO_logiBUS_QXA_BG` (nur lokaler SoftKey, keine OPC-UA-Anbindung, keine Latch-Logik) kommen hier die komplette Flankenerkennungs-/Merge-/Latch-Kette sowie OPC-UA-Subscribe/Publish hinzu.

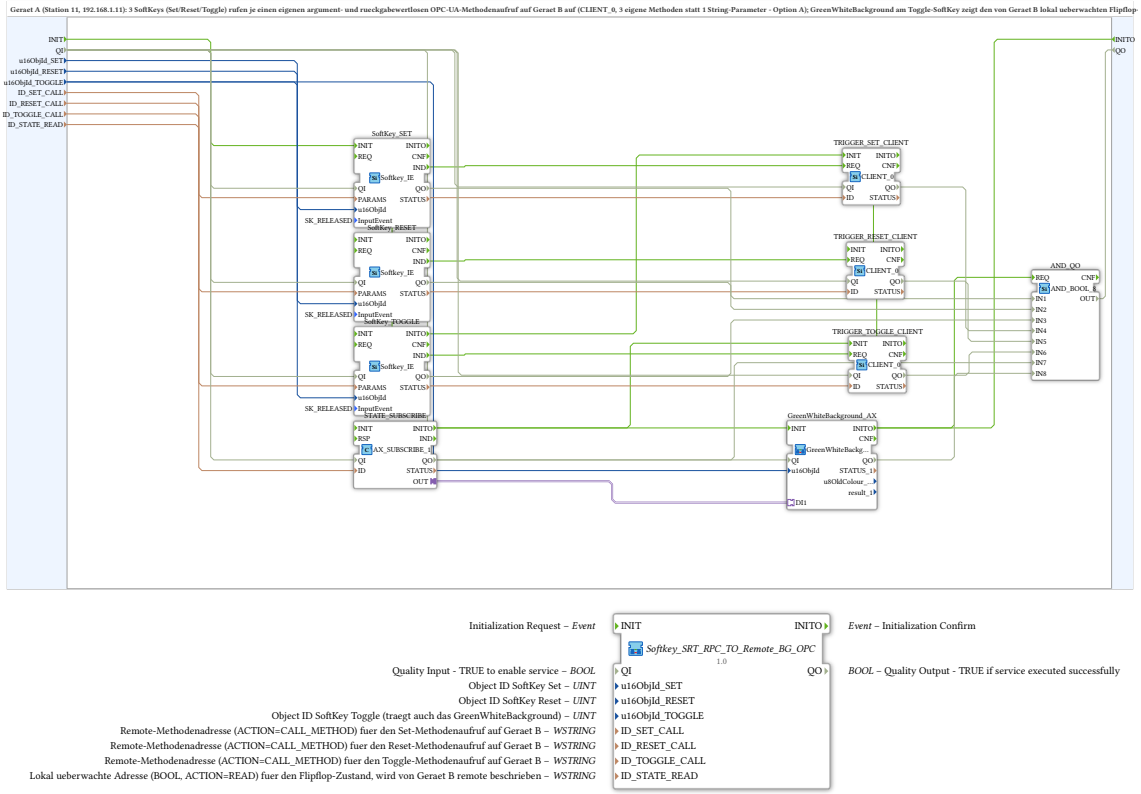
Zusammenfassung

`Softkey_IXA_TO_logiBUS_QXA_BG_OPC` schaltet einen digitalen logiBUS-Ausgang per SoftKey und/oder Remote-OPC-UA-Kommando, inklusive VT-Statusfarbe und OPC-UA-Echo. Bei nur einer aktiven Quelle ist das Verhalten rein tastend; überlappen sich beide Quellen, gewinnt Last-Wins statt einer echten ODER-Verknüpfung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Softkey_SRT_RPC_TO_Remote_BG_OPC



• — ○ • — ○ • — ○ •

Einleitung

Die SubApp **Softkey_SRT_RPC_TO_Remote_BG_OPC** verbindet physische Softkeys einer Bedieneinheit mit einem entfernten OPC-UA-Server (Gerät B). Über drei separate OPC-UA-Clients (`CLIENT_0`) werden Methodenaufrufe zum Setzen, Rücksetzen und Toggeln eines Binärausgangs auf Gerät B ausgelöst. Ein zusätzlicher Abonnement-Adapter überwacht den zugehörigen Flipflop-Zustand und übergibt ihn an eine Visualisierungs-SubApp (GreenWhiteBackground), die den Zustand farblich darstellt. Die gesamte Kommunikationslogik ist in dieser SubApp gekapselt und daher wiederverwendbar.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung. Ansonsten besitzt die SubApp keine externen Ereigniseingänge; die Softkey-Erkennung erfolgt über intern implementierte Funktionsblöcke.

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung (von `GreenWhiteBackground_AX.INITO`)

Daten-Eingänge

Name	Typ	Beschreibung
QI	BOOL	Quality Input, TRUE gibt den Dienst frei (Vorgabe TRUE)
u16ObjId_SET	UINT	Objekt-ID für den Softkey „Set“ (initial ID_NULL)
u16ObjId_RESET	UINT	Objekt-ID für den Softkey „Reset“
u16ObjId_TOGGLE	UINT	Objekt-ID für den Softkey „Toggle“ (trägt auch das GreenWhite-Background)
ID_SET_CALL	WSTRING	Remote-Methodenadresse (ACTION=CALL_METHOD) für den Set-Aufruf auf Gerät B
ID_RESET_CALL	WSTRING	Remote-Methodenadresse (ACTION=CALL_METHOD) für den Reset-Aufruf auf Gerät B
ID_TOGGLE_CALL	WSTRING	Remote-Methodenadresse (ACTION=CALL_METHOD) für den Toggle-Aufruf auf Gerät B
ID_STATE_READ	WSTRING	Adresse (BOOL, ACTION=READ) für den Flipflop-Zustand, der von Gerät B überwacht wird

Daten-Ausgänge

- **QO** (BOOL): Quality Output, UND (AND_QO, iec61131::booleanOperators::AND_BOOL_8) aus den QO-Ausgängen aller acht internen Bausteine (3 × Softkey_IE, 3 × CLIENT_0, STATE_SUBSCRIBE, GreenWhiteBackground_AX)

Adapter-Anschlüsse

Keine externen Adapter.

Funktionsweise

Die SubApp nutzt drei Instanzen des Funktionsblocks isobus::UT::io::Softkey::Softkey_IE, die auf das Loslassen eines Softkeys reagieren (konfiguriert über SK_RELEASED). Jedes Drücken eines Softkeys erzeugt ein internes Ereignis (IND), das einen zugeordneten CLIENT_0-FB (OPC-UA-Client) dazu veranlasst, die konfigurierte Methode (Set, Reset, Toggle) auf dem entfernten Gerät aufzurufen. Die dafür benötigten Methodenadressen werden über die Eingänge ID_SET_CALL, ID_RESET_CALL und ID_TOGGLE_CALL bereitgestellt.

Initialisierung und Qualität: INIT durchläuft eine Kette SoftKey_SET → SoftKey_RESET → SoftKey_TOGGLE → TRIGGER_SET_CLIENT → TRIGGER_RESET_CLIENT → TRIGGER_TOGGLE_CLIENT → STATE_SUBSCRIBE → GreenWhiteBackground_AX (jeweils INITO → nächstes INIT); GreenWhiteBackground_AX.INITO → INITO (und → AND_QO.REQ). QI wird an .QI aller acht Bausteine verteilt; deren QO-Ausgänge laufen auf AND_QO.IN1..IN8, AND_QO.OUT → QO. Die Instanzen sind nicht mehr fest auf QI=TRUE parametriert.

Parallel abonniert die SubApp über den Adapter STATE_SUBSCRIBE kontinuierlich den booleschen Zustand eines Flipflops auf Gerät B (Adresse aus ID_STATE_READ). Dieser Zustand wird an die SubApp GreenWhiteBackground_AX übergeben, die eine farbige Rückmeldung (grün/weiß) erzeugt – typischerweise für den Toggle-Softkey.

Die SubApp ist in zwei Ebenen organisiert: die Bedienlogik (Softkeys + OPC-UA-Clients) und die Anzeigelogik (Abonnement + GreenWhiteBackground). Dadurch wird eine klare Trennung von Aktorik und Sensorik erreicht.

Technische Besonderheiten

- **Drei separate OPC-UA-Clients** (CLIENT_0) pro Methode, was eine eindeutige Zuordnung zwischen Softkey und Fernaufruf ermöglicht und Fehlkonfigurationen vermeidet.
- **CALL_METHOD ohne Parameter und Rückgabewert** – die Methode wird als „fire-and-forget“-Aufruf behandelt.
- **Zustandsüberwachung per Abonnement** (adapter::net::AX_SUBSCRIBE_1) statt Polling – dies liefert ereignisgenaue Statusänderungen ohne zusätzliche Latenz.

- **Parametrierbarkeit** über Objekt-IDs (`UINT`) und Methodenadressen (`WSTRING`) ermöglicht die Wiederverwendung in verschiedenen Anlagen und Konfigurationen.
- **Import externer Konstanten** (`ID_NULL`, `SK_RELEASED`) zur sicheren Initialisierung und einheitlichen Ereignisdefinition.
- **SubApp-basierte Kapselung** – das Protokoll liegt in der SubApp, nicht in der Ressource eines Geräts, was die Portabilität erhöht.

Zustandsübersicht

Die SubApp selbst besitzt keinen expliziten endlichen Zustandsautomaten. Der interne Ablauf wird durch die externen Ereignisse der Softkeys gesteuert:

- **Softkey-Ereignis** (Loslassen) ⟶ löst einen OPC-UA-Methodenaufruf aus.
- **Abonnement** liefert asynchron Änderungen des Flipflop-Zustands (0 oder 1), die an die Anzeige weitergegeben werden. Der aktuelle Zustand des Flipflops (0 = weiß, 1 = grün) wird visuell dargestellt, ohne dass eine zusätzliche Zustandslogik in der SubApp erforderlich ist.

Anwendungsszenarien

- Fernsteuerung eines Binärausgangs (z.B. Ventil, Lampe, Motor) über Taster an einer Bedieneinheit.
- Visualisierung eines übergeordneten Steuerbits (Toggle-Funktion) mit Farbwechsel zur Statusanzeige.
- Integration in Maschinensteuerungen, bei denen OPC-UA als Kommunikationsprotokoll zwischen Bedienpanel und SPS dient.
- Einsatz als wiederverwendbare Bibliothekskomponente in verschiedenen Anlagen, da alle Adressen und IDs extern gesetzt werden können.

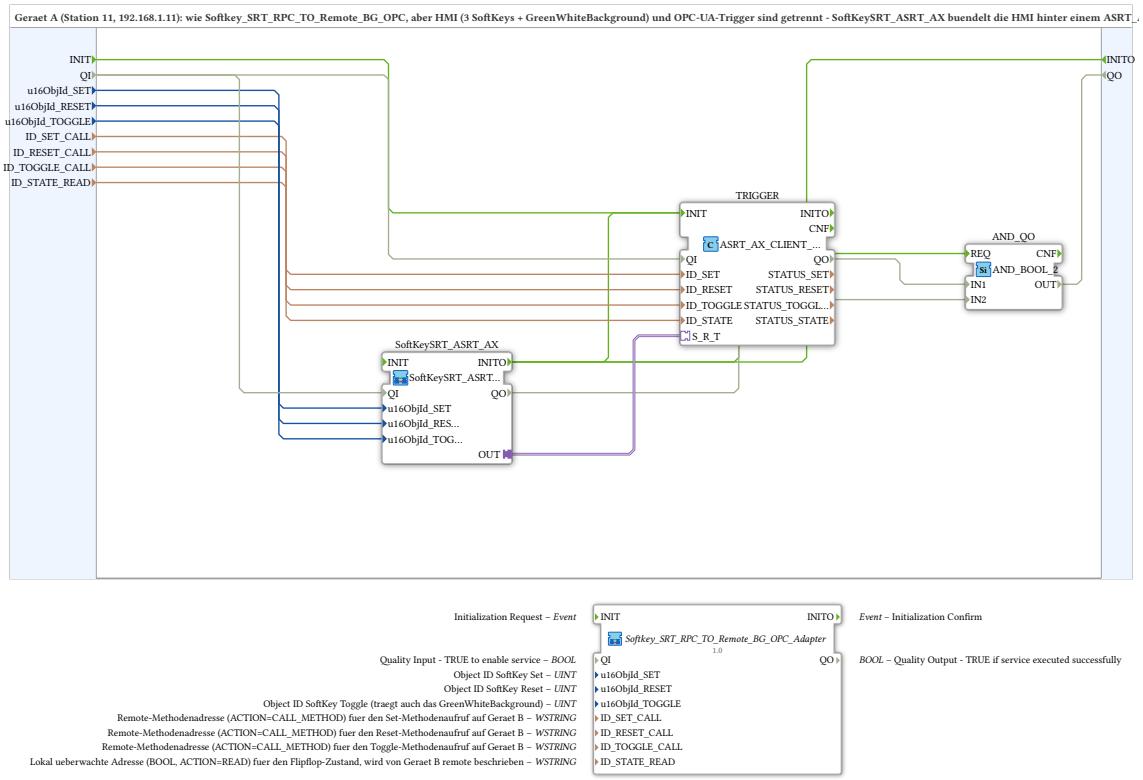
Vergleich mit ähnlichen Bausteinen

- **Einzelner Client mit String-Parametern:** Eine Alternative wäre ein CLIENT_0-FB, der Methoden mit einem String-Parameter aufruft, um die Methode zu unterscheiden. Hierfür wären jedoch zusätzliche Parsing-Logik und Fehlerbehandlung nötig, während die SubApp durch separate Clients pro Aktion eine klarere und sicherere Zuordnung bietet.
- **Polling-basierte Zustandsabfrage:** Statt eines Abonnements könnte der Zustand zyklisch gelesen werden. Dies würde jedoch Verzögerungen verursachen und unnötige Netzwerklast erzeugen. Das hier verwendete Abonnement ist effizienter und aktueller.
- **Monolithische Ressourcenlogik:** Ohne SubApp-Struktur wäre die Kommunikationslogik in der Ressource des Geräts integriert, was die Wiederverwendung und Wartung erschwert. Die SubApp kapselt die gesamte Logik und macht sie portabel.

Fazit

Die SubApp `Softkey_SRT_RPC_TO_Remote_BG_OPC` ist eine durchdachte, modulare Lösung zur Fernsteuerung und Zustandsanzeige über OPC-UA. Durch die Kombination von Softkey-Erkennung, dedizierten OPC-UA-Clients und einem ereignisbasierten Zustandsabonnement bietet sie eine hohe Zuverlässigkeit, Flexibilität und einfache Integration in industrielle Steuerungssysteme. Ihre parametrierbaren Eingänge machen sie universell einsetzbar, und die klare Trennung von Bedienung und Anzeige erleichtert die Wartung und Erweiterung.

Softkey_SRT_RPC_TO_Remote_BG_OPC_Adapter



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock `Softkey_SRT_RPC_TO_Remote_BG_OPC_Adapter` ist eine Subapplikation (SubApp) zur Anbindung einer lokalen HMI (Softkeys mit Green-White-Background) an ein entferntes Gerät (Gerät B) über OPC-UA. Die SubApp entkoppelt die HMI-Logik von der OPC-UA-Trigger-Funktionalität und ermöglicht so eine flexible Integration in Automatisierungsumgebungen. Sie wird auf Gerät A (z. B. Station 11, IP 192.168.1.11) eingesetzt und kommuniziert mit Gerät B über Remote-Methodenaufrufe.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialization Request

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialization Confirm

Daten-Eingänge

Name	Typ	Beschreibung
`QI`	BOOL	Quality Input – TRUE aktiviert den Dienst (Standard: `TRUE`).
`u16ObjId_SET`	UINT	Object ID für den SoftKey „Set“ (Standard: `ID_NULL`).
`u16ObjId_RESET`	UINT	Object ID für den SoftKey „Reset“ (Standard: `ID_NULL`).
`u16ObjId_TOGGLE`	UINT	Object ID für den SoftKey „Toggle“ (inkl. Green-White-Background) (Standard: `ID_NULL`).
`ID_SET_CALL`	WSTRING	Remote-Methodenadresse (ACTION=CALL_METHOD) für den Set-Aufruf auf Gerät B.

ID_RESET_CALL	WSTRING	Remote-Methodenadresse (ACTION=CALL_METHOD) für den Reset-Aufruf auf Gerät B.
ID_TOGGLE_CALL	WSTRING	Remote-Methodenadresse (ACTION=CALL_METHOD) für den Toggle-Aufruf auf Gerät B.
ID_STATE_READ	WSTRING	Lokal überwachte Adresse (BOOL, ACTION=READ) für den Flipflop-Zustand, der von Gerät B remote beschrieben wird.

Daten-Ausgänge

- **QO** (BOOL): Quality Output, UND-Verknüpfung (AND_QO, AND_BOOL_2) der QO-Ausgänge von TRIGGER und SoftKeySRT_ASRT_AX

Adapter

Keine externen Adapter an der SubApp-Schnittstelle. Die Verdrahtung zu den internen Bausteinen erfolgt ausschließlich über Datenverbindungen und einen internen Adapter-Socket.

Funktionsweise

Die SubApp integriert zwei interne Bausteine sowie AND_QO (AND_BOOL_2), das die QO-Ausgänge von TRIGGER und SoftKeySRT_ASRT_AX zu QO verknüpft. Initialisierung: INIT → TRIGGER.INIT; TRIGGER.INITO → SoftKeySRT_ASRT_AX.INIT; SoftKeySRT_ASRT_AX.INITO → INITO und → AND_QO.REQ. QI wird an TRIGGER.QI und SoftKeySRT_ASRT_AX.QI verteilt.

1. **SoftKeySRT_ASRT_AX** (SubApp vom Typ MyLib::sys::SoftKeySRT_ASRT_AX):
   Bündelt die HMI-Logik für die drei SoftKeys (Set, Reset, Toggle) und das Green-White-Background. Sie stellt über einen **ASRT_AX-Plug** eine standardisierte Schnittstelle nach außen bereit.
2. **TRIGGER** (FB vom Typ adapter::net::ASRT_AX_CLIENT_0_SUBSCRIBE_1):
   Implementiert den OPC-UA-Client-Teil. Er verbindet sich mit Gerät B, überwacht den Zustand des Flipflops (über ID_STATE_READ) und führt bei Bedarf die Remote-Methodenaufrufe für Set, Reset und Toggle aus (über ID_SET_CALL, ID_RESET_CALL, ID_TOGGLE_CALL).

Die Verbindung zwischen HMI und Trigger wird über den **ASRT_AX**-Adapter hergestellt: Der Plug von SoftKeySRT_ASRT_AX (Ausgang OUT) wird mit dem Socket S_R_T des TRIGGER-FBs verbunden. Dadurch werden die SoftKey-Ereignisse direkt an die OPC-UA-Trigger-Logik weitergeleitet.

Die Objekt-IDs (u16ObjId_SET, u16ObjId_RESET, u16ObjId_TOGGLE) werden über Datenverbindungen an SoftKeySRT_ASRT_AX übergeben und dort zur Identifikation der jeweiligen SoftKeys im HMI verwendet. Die Methodenadressen (ID_*_CALL) sowie die Zustandsadresse (ID_STATE_READ) werden an den TRIGGER-FB übergeben und dort zur Kommunikation mit Gerät B genutzt.

Technische Besonderheiten

- **Trennung von HMI und OPC-UA-Trigger**: Die SubApp ist eine Weiterentwicklung des Bausteins Softkey_SRT_RPC_TO_Remote_BG_OPC, bei dem diese Funktionalitäten nicht getrennt waren. Diese Entkopplung erleichtert Wartung und Test.
- **Verwendung des ASRT_AX-Adapterkonzepts**: Der interne Plug/Socket-Mechanismus gewährleistet eine klare Schnittstelle zwischen den Bausteinen und ermöglicht eine hohe Wiederverwendbarkeit.
- **Standardisierte OPC-UA-Kommunikation**: Der TRIGGER-FB basiert auf dem Typ ASRT_AX_CLIENT_0_SUBSCRIBE_1, der einen OPC-UA-Client mit Subscribe-Funktionalität darstellt. Die Adressen für Remote-Methoden und Variablen sind als WSTRING konfigurierbar.
- **Initialwert ID_NULL**: Die Objekt-IDs sind voreingestellt auf ID_NULL, was bedeutet, dass sie erst vor Inbetriebnahme gesetzt werden müssen, um einen gültigen SoftKey zu referenzieren.
- **Compiler-Informationen**: Die SubApp ist im Paket MyLib::sys eingebettet und importiert den globalen Konstanten-Wert ID_NULL.

Zustandsübersicht

Die SubApp selbst besitzt keinen eigenen Zustandsautomaten. Die internen Bausteine SoftKeySRT_ASRT_AX und TRIGGER können jedoch eigene Zustände aufweisen:

- SoftKeySRT_ASRT_AX realisiert die Zustände der SoftKeys (z. B. „gedrückt“/„nicht gedrückt“) und das Ein/Aus-Verhalten des Green-White-Backgrounds.

- Der `TRIGGER`-FB verwaltet den Verbindungsstatus zum OPC-UA-Server und den Ablauf der Remote-Methodenaufrufe.

Diese Zustände sind nach außen nicht direkt sichtbar, beeinflussen aber das Verhalten der SubApp, insbesondere die Ausführung der Remote-Aufrufe.

Anwendungsszenarien

Typische Einsatzfälle sind:

- **Bediengeräte in der Automatisierung**: Ein lokales HMI-Panel (z. B. an einer Station) steuert über OPC-UA Funktionen an einer entfernten Maschine oder Anlage.
- **Fernwartung und -diagnose**: SoftKeys werden verwendet, um z. B. Maschinenzustände (Set, Reset, Toggle) aus der Ferne zu ändern und den aktuellen Flipflop-Status zu überwachen.
- **Flexible Systemarchitekturen**: Durch die Trennung von HMI-Logik und Netzwerk-Trigger kann die HMI unabhängig vom Kommunikationsprotokoll entwickelt oder ausgetauscht werden.

Vergleich mit ähnlichen Bausteinen

Der Baustein `Softkey_SRT_RPC_TO_Remote_BG_OPC` stellt die ungetrennte Variante dar. Bei diesem sind HMI-Logik und OPC-UA-Trigger direkt gekoppelt. Der `Softkey_SRT_RPC_TO_Remote_BG_OPC_Adapter` trennt diese beiden Aspekte durch die Einführung einer separaten `SoftKeySRT_ASRT_AX`-SubApp und eines dedizierten `TRIGGER`-FBs mit Adapterschnittstelle. Vorteile dieser Trennung:

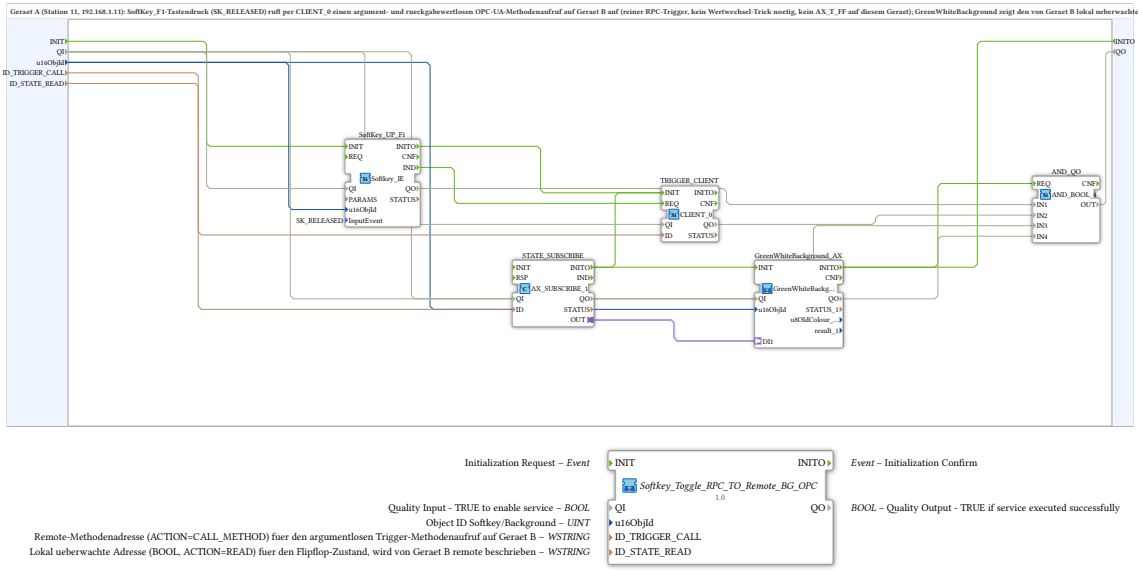
- Bessere Testbarkeit und Modularität.
- Austauschbarkeit der OPC-UA-Implementierung ohne Änderung der HMI-Logik.
- Wiederverwendung der HMI-SubApp in anderen Kontexten (z. B. mit anderen Kommunikationsprotokollen).

Nachteilig ist der etwas höhere Verdrahtungsaufwand durch die zusätzliche Adapter-Verbindung.

Fazit

Der `Softkey_SRT_RPC_TO_Remote_BG_OPC_Adapter` ist ein leistungsfähiger Baustein für die Anbindung einer HMI an ein entferntes OPC-UA-Gerät. Durch die klare Trennung der Verantwortlichkeiten (HMI-Logik vs. Kommunikation) bietet er eine hohe Flexibilität und erleichtert Integration sowie Wartung. Die Verwendung standardisierter Adapter (ASRT_AX) und konfigurierbarer Adressen macht ihn für vielfältige Automatisierungsszenarien geeignet.

Softkey_Toggle_RPC_TO_Remote_BG_OPC



• – ○ • – ○ • – ○ •

Einleitung

Die Subapp `Softkey_Toggle_RPC_TO_Remote_BG_OPC` kapselt ein schlankes RPC-Protokoll über OPC-UA. Sie ist für eine Zwei-Geräte-Anwendung ausgelegt: Ein Softkey-Ereignis auf Gerät A löst über einen

OPC-UA-Methodenaufruf einen Trigger auf Gerät B aus. Parallel wird ein von Gerät B remote gesetzter Bool-Zustand abonniert und über eine grafische SubApp als grüner oder weißer Hintergrund dargestellt. Das Protokoll ist vollständig in diesem Composite-Baustein gekapselt und nicht an eine bestimmte Geräteresource gebunden.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung
INIT	EInit	Initialization Request

Ereignis-Ausgänge

Name	Typ	Beschreibung
INITO	EInit	Initialization Confirm

Daten-Eingänge

Name	Typ	Beschreibung
QI	BOOL	Quality Input - TRUE, um den Dienst freizugeben; Initialwert TRUE.
u16ObjId	UINT	Objekt-ID für Softkey und Hintergrund; Initialwert ID_NULL.
ID_TRIGGER_CALL	WSTRING	Remote-Methodenadresse (ACTION=CALL_METHOD) für den argumentlosen Trigger-Methodenaufruf auf Gerät B.
ID_STATE_READ	WSTRING	Lokal überwachte Adresse (BOOL, ACTION=READ) für den Flipflop-Zustand, wird von Gerät B remote beschrieben.

Daten-Ausgänge

Name	Typ	Beschreibung
QO	BOOL	Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde (UND von SoftKey_UP_F1, TRIGGER_CLIENT, STATE_SUBSCRIBE und GreenWhiteBackground_AX).

Adapter

Keine an der Subapp-Schnittstelle. Intern wird die Adapterverbindung STATE_SUBSCRIBE.OUT an den Adaptereingang DI1 der SubApp GreenWhiteBackground_AX genutzt.

Funktionsweise

Der Baustein enthält drei funktionale Pfade sowie die Initialisierungs- und Status-Verdrahtung:

1. **Trigger-Pfad:** Der Eingangsbaustein SoftKey_UP_F1 (Typ Softkey_IE) überwacht einen Softkey auf Gerät A. Beim konfigurierten Ereignis SK_RELEASED (Softkey F1 losgelassen) erzeugt er am Ausgang IND ein Ereignis, das den Eingang REQ des TRIGGER_CLIENT (Typ CLIENT_0) ansteuert. Dadurch wird ein OPC-UA-Methodenaufruf an die Adresse ID_TRIGGER_CALL ausgelöst. Es handelt sich um einen reinen RPC-Trigger ohne Argumente und ohne Rückgabewert.
2. **Zustands-Pfad:** Der Baustein STATE_SUBSCRIBE (Typ AX_SUBSCRIBE_1) abonniert kontinuierlich eine Bool-Variable auf Gerät B unter der Adresse ID_STATE_READ. Dieser Wert wird über den Adapterausgang OUT an den Adaptereingang DI1 der SubApp GreenWhiteBackground_AX übergeben.
3. **Anzeige-Pfad:** Die SubApp GreenWhiteBackground_AX (Typ GreenWhiteBackground1_AX) wertet den abonnierten Bool-Zustand aus und steuert die Hintergrundfarbe (grün oder weiß) auf Gerät A.
4. **Initialisierung und Status:** INIT → SoftKey_UP_F1.INIT, SoftKey_UP_F1.INITO → TRIGGER_CLIENT.INIT, TRIGGER_CLIENT.INITO → STATE_SUBSCRIBE.INIT, STATE_SUBSCRIBE.INITO → GreenWhiteBackground_AX.INIT, GreenWhiteBackground_AX.INITO → INITO und → AND_QO.REQ. QI wird an SoftKey_UP_F1, TRIGGER_CLIENT, STATE_SUBSCRIBE und GreenWhiteBackground_AX wei-

tergegeben; deren QO werden im Baustein AND_QO (Typ iec61131::booleanOperators::AND_BOOL_4) per UND zu QO verknüpft.

Der ursprüngliche Kommentar beschreibt, dass auf Gerät A kein Wertwechsel-Trick und kein AX_T_FF (lokaler Toggle-Funktionsbaustein) erforderlich ist. Der Flipflop-Zustand wird bereits auf Gerät B lokal überwacht und muss von Gerät A nur noch angezeigt werden. Der Aufbau folgt dem „SUB style“: Das komplette Protokoll ist im MyLib::sys-Composite enthalten, nicht in der Ressource des Geräts.

Technische Besonderheiten

- Verwendet OPC-UA-Mechanismen: CALL_METHOD für den Remote-Trigger und READ für die Zustandsüberwachung.
- An der Subapp-Schnittstelle gibt es nur INIT0 und Q0 als Ausgänge; die Subapp ist ansonsten rein intern verdrahtet und wirkt direkt auf die grafische Anzeige.
- Alle Eingänge sind als Konfigurationsparameter ausgeführt: Objekt-ID, Trigger-Adresse und State-Adresse (plus QI).
- Konstante Initialwerte: QI=TRUE (Subapp-Eingang), InputEvent=SK_RELEASED, u16ObjId=ID_NULL.
- Importiert Definitionen aus isobus::UT::Q::const::IDs und isobus::UT::io::Softkey.
- Die verwendete SubApp GreenWhiteBackground_AX stammt aus der Bibliothek MyLib::sys und ist Teil des gekapselten Composite-Protokolls.

Zustandsübersicht

Die Subapp selbst besitzt keinen expliziten Zustandsautomaten. Der relevante Zustand ist der remote auf Gerät B gehaltene Flipflop-Zustand, der über AX_SUBSCRIBE_1 kontinuierlich beobachtet wird. Dieser wird in der Anzeige-SubApp GreenWhiteBackground_AX interpretiert und als eine von zwei Farben dargestellt:

- **Grün:** Bool-Zustand aktiv (Flipflop gesetzt)
- **Weiß:** Bool-Zustand inaktiv (Flipflop gelöscht)

Durch den Softkey-Trigger wird der Flipflop auf Gerät B umgeschaltet, wodurch sich der abonnommene Zustand und damit auch die Hintergrundfarbe ändert.

Anwendungsszenarien

- Bedienpanel auf Gerät A mit einem Softkey F1, der eine Methode auf einem entfernten Gerät B aufruft, um einen Flipflop zu toggeln.
- Anzeige des entfernten Flipflop-Zustands als grün/weißer Hintergrund auf dem Bedienpanel.
- Wiederverwendbarer Protokollbaustein in einer Composite-Bibliothek zur Entkopplung von Geräteressource und Anwendungslogik.
- Einsatz überall dort, wo ein einfacher RPC-Trigger mit optionaler Rückmeldung über OPC-UA benötigt wird.

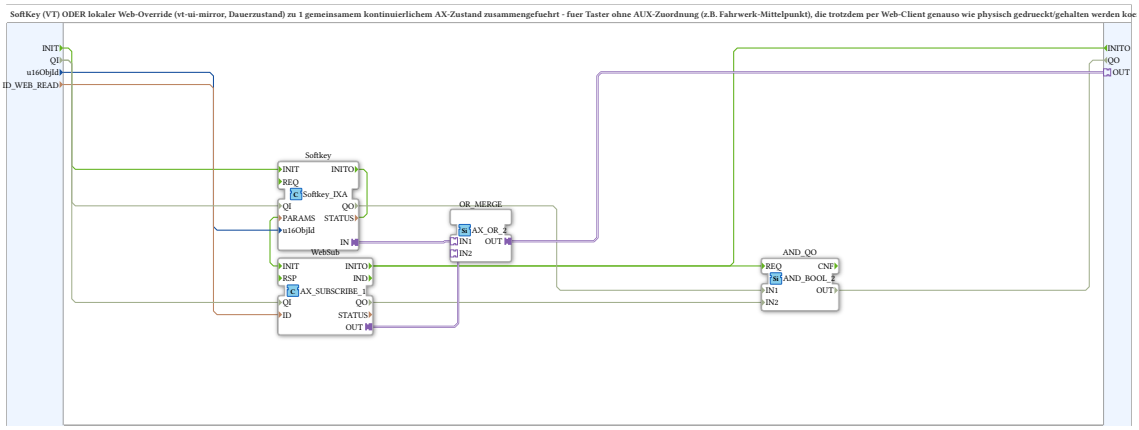
Vergleich mit ähnlichen Bausteinen

- Gegenüber einem einfachen CLIENT_0-Baustein bietet diese Subapp eine abgeschlossene, kombinierte Trigger- und Anzeigefunktionalität.
- Im Unterschied zu einer Variante mit lokalem AX_T_FF (Toggle-Funktionsbaustein) wird hier kein lokaler Flipflop benötigt; der Zustand wird remote erzeugt und nur gespiegelt.
- Während andere Bausteine den Zustand als Datenausgang liefern, wirkt diese Subapp direkt auf eine grafische SubApp (GreenWhiteBackground_AX) und benötigt keine externen Ausgänge.
- Die Kapselung des gesamten Protokolls in einem Composite-Baustein unterscheidet sie von einer direkten Ressourcen-Verdrahtung und erhöht die Wiederverwendbarkeit.

Fazit

Softkey_Toggle_RPC_TO_Remote_BG_OPC ist ein kompakter, wiederverwendbarer Subapp-Baustein zur Kopplung eines lokalen Bedienelements mit einem entfernten OPC-UA-Gerät. Er verbindet einen RPC-Trigger mit einer Zustandsanzeige und kapselt das gesamte Protokoll im MyLib::sys-Composite. Die klare Trennung von Trigger- und Anzeigepfad sowie die minimale Schnittstelle mit nur drei Konfigurationseingängen (plus QI) machen den Baustein einfach konfigurierbar und gut in verschiedene Bedienpanels integrierbar.

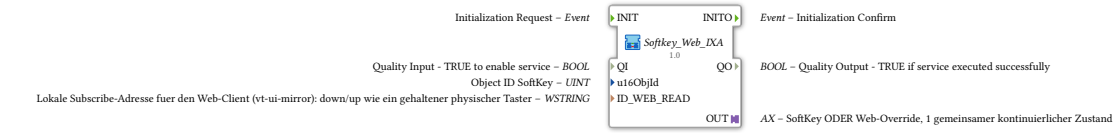
Softkey_Web_IXA



• – ○ • – ○ • – ○ •

Einleitung

`Softkey_Web_IXA` führt einen **SoftKey (VT)** ODER einen lokalen **Web-Override** (vt-ui-mirror, Dauerzustand) zu **einem** gemeinsamen kontinuierlichen `AX`-Zustand zusammen. Er ist für Taster ohne AUX-Zuordnung gedacht (z. B. Fahrwerk-Mittelpunkt), die trotzdem per Web-Client genauso wie physisch gedrückt oder gehalten werden können sollen. Analog zu `Softkey_Aux_IXA_TO_Remote_WRITE`, dort zusätzlich mit AUX und Remote-Write; hier nur der reine Merge ohne AUX und ohne Schreiben. Der Aufrufer verarbeitet den gemergten Zustand selbst weiter (z. B. mit `SoftKey_Kurz_Lang`).



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; startet die Kette `Softkey` ⟶ `WebSub`

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung; kommt von `WebSub.INITO`

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input, Freigabe des Dienstes; wird an `Softkey` und `WebSub` durchgereicht
- **u16ObjId** (`UINT`): Object ID SoftKey
- **ID_WEB_READ** (`WSTRING`): lokale Subscribe-Adresse für den Web-Client (vt-ui-mirror): down/up wie ein gehaltener physischer Taster

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output, `AND` der `QO`-Ausgänge von `Softkey` und `WebSub` (über `AND_QO`)

Adapter

- **OUT** (Plug, `adapter::types::unidirectional::AX`): SoftKey ODER Web-Override als ein gemeinsamer kontinuierlicher Zustand

Verwendete Funktionsbausteine (FBs)

- **`Softkey`** (`isobus::UT::io::Softkey::Softkey_IXA`): liest den SoftKey-Zustand
- **`WebSub`** (`adapter::net::AX_SUBSCRIBE_1`): abonniert den Web-Override
- **`OR_MERGE`** (`adapter::booleanOperators::AX_OR_2`): ODER-Verknüpfung beider Zustände
- **`AND_QO`** (`iec61131::booleanOperators::AND_BOOL_2`): verknüpft `Softkey.QO` und `WebSub.QO` zum SubApp-Ausgang `QO`

Funktionsweise

- `u16ObjId` ⟶ `Softkey.u16ObjId`; `ID_WEB_READ` ⟶ `WebSub.ID`
- `Softkey.IN` ⟶ `OR_MERGE.IN1`; `WebSub.OUT` ⟶ `OR_MERGE.IN2`
- `OR_MERGE.OUT` ⟶ `OUT`
- Initialisierung: `INIT` ⟶ `Softkey.INIT`; `Softkey.INITO` ⟶ `WebSub.INIT`; `WebSub.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`
- Qualität: `QI` ⟶ `Softkey.QI` und `WebSub.QI`; `Softkey.QO` ⟶ `AND_QO.IN1`, `WebSub.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`

Technische Besonderheiten

- Dauerzustand (nicht Ereignis): Der Web-Client kann die Taste halten wie eine physische
- Kein AUX, kein Remote-Write

Anwendungsszenarien

- Taster ohne AUX-Zuordnung, die auch per Web-Client bedienbar sein sollen und deren Zustand z. B. von `SoftKey_Kurz_Lang` weiterverarbeitet wird

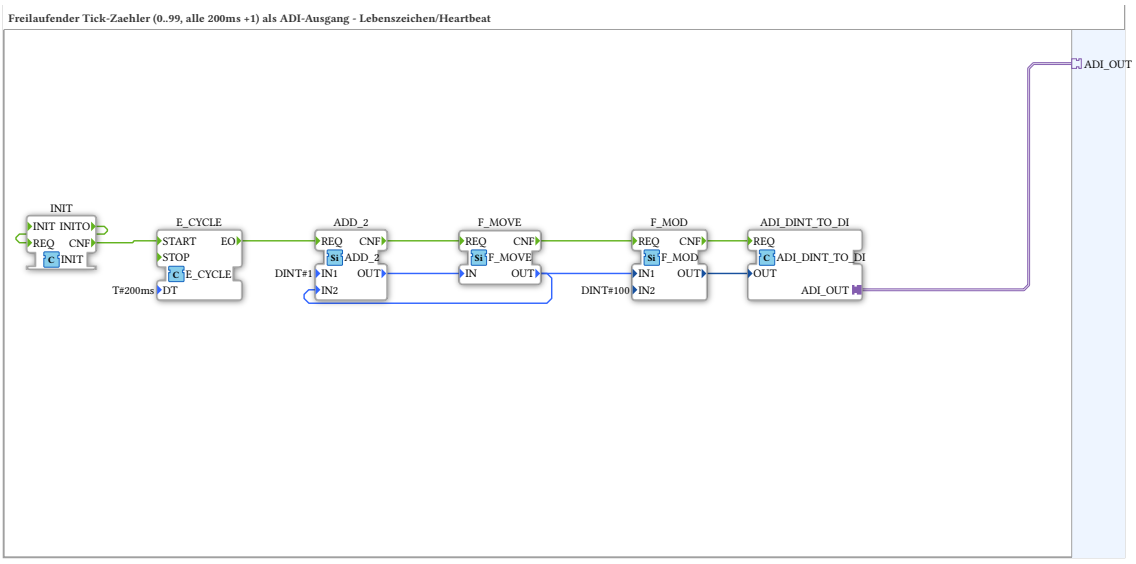
Zusammenfassung

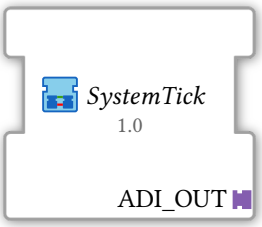
`Softkey_Web_IXA` verodert SoftKey und Web-Override zu einem kontinuierlichen `AX`-Zustand.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SystemTick





ADI – Aktueller Tick-Zaehlerstand (0..99), aendert sich alle 200ms

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `SystemTick` ist eine Subapplikation, die einen freilaufenden Tick-Zähler realisiert. Er zählt alle 200 ms um 1 hoch, beginnend bei 0, und wird bei Erreichen von 100 auf 0 zurückgesetzt. Der aktuelle Zählerstand (0…99) wird über einen unidirektionalen ADI-Adapter (`ADI_OUT`) bereitgestellt.

Der Baustein dient als Lebenszeichen (Heartbeat): Solange sich der Ausgabewert periodisch ändert, ist die erzeugende Steuerung aktiv.

Schnittstellenstruktur

Ereignis-Eingänge
- Keine

Ereignis-Ausgänge
- Keine

Daten-Eingänge
- Keine

Daten-Ausgänge
- Keine

Adapter
- **`ADI_OUT`** (Plug, Typ: `adapter::types::unidirectional::ADI`)
  Liefert den aktuellen Tick-Zählerstand als DINT-Wert (0…99). Der Adapter enthält typischerweise ein Ereignis und einen Datenwert.

Funktionsweise

Die Subapplikation wird durch das Initialisierungsereignis `INIT` gestartet. Dieses Ereignis löst den internen Baustein `E_CYCLE` aus, der mit einer Periodendauer von 200 ms konfiguriert ist. Nach Ablauf jeder Periode erzeugt `E_CYCLE` ein Ereignis, das den Additionsbaustein `ADD_2` anstößt. `ADD_2` addiert den konstanten Wert 1 (`IN1 = DINT#1`) auf den aktuellen Zählerstand, der im Baustein `F_MOVE` gespeichert ist. Das Ergebnis wird wieder in `F_MOVE` geschrieben und steht als interner Wert zur Verfügung.

Gleichzeitig wird der interne Wert dem Modulo-Baustein `F_MOD` zugeführt, der den Wert auf den Bereich 0…99 begrenzt (Modulo 100). Das Ergebnis wird über den Konvertierungsbaustein `ADI_DINT_TO_DI` in das `ADI`-Adapterformat umgewandelt und über den Plug `ADI_OUT` ausgegeben.

Die Verkettung der Ereignisse ist so aufgebaut, dass nach der Initialisierung zyklisch weitergezählt wird, ohne dass ein externes Ereignis erforderlich ist.

Technische Besonderheiten
- **Speicherung des Zählerstandes:** Der Baustein `F_MOVE` fungiert als Speicher für den aktuellen Zählerwert. Die Rückkopplung von `F_MOVE.OUT` zu `ADD_2.IN2` stellt die Aktualisierung im nächsten Zyklus sicher.
- **Modulare Grenze:** Der Modulo-Baustein (`F_MOD`) mit Parameter `IN2 = DINT#100` begrenzt den Wertebereich hart auf 100 – eine Änderung der Zykluslänge erfordert nur die Anpassung dieses Parameters.
- **Periodengesteuert:** Die Periodendauer wird über den Baustein `E_CYCLE` mit `DT = T#200ms` festgelegt. Eine Änderung der Taktrate erfolgt durch Anpassung dieses Zeitparameters.
- **Adapter-Konvertierung:** Der Baustein `ADI_DINT_TO_DI` kapselt die Umwandlung eines DINT-Werts in das ADI-Protokoll, sodass der Ausgang direkt an andere 4diac-Komponenten angeschlossen werden kann.

Zustandsübersicht

Die Subapplikation besitzt keinen expliziten Zustandsautomaten. Der einzige relevante Zustand ist der aktuelle Zählerstand, der sich nach jedem 200-ms-Intervall um 1 erhöht und bei 99 wieder auf 0 springt. Der Zyklus dauert 20 Sekunden (100 × 200 ms). Nach der Initialisierung läuft der Zähler ununterbrochen weiter, bis die Subapplikation deaktiviert wird.

Anwendungsszenarien
- **Heartbeat/ Lebenszeichen:** Durch Beobachten des `ADI_OUT`-Werts kann eine übergeordnete Steuerung erkennen, ob die Subapplikation (bzw. der sie ausführende Controller) noch aktiv ist.
- **Synchronisation:** Der Tick-Zähler kann als gemeinsames Zeitraster für mehrere Teilnehmer dienen, die über den gleichen Zählstand ihren Ablauf abstimmen.
- **Diagnose:** Der periodisch wechselnde Wert lässt sich in Visualisierungen oder OPC-UA-Bereichen nutzen, um die Kommunikation zu überprüfen.

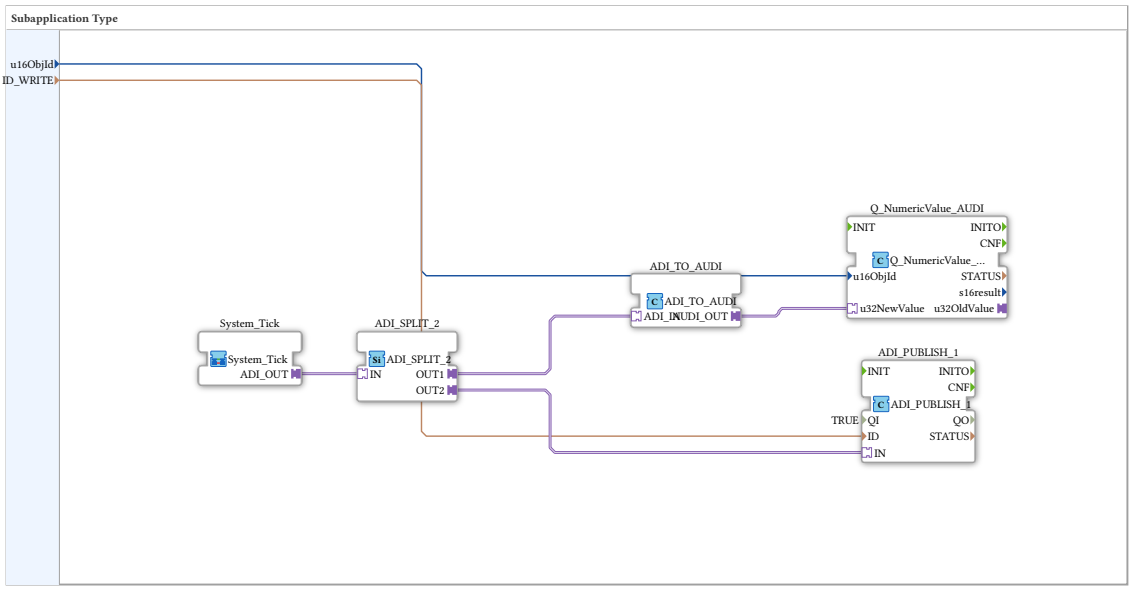
Vergleich mit ähnlichen Bausteinen

Ein einfacher Zähler mit fester Periode könnte alternativ mit einem `E_CYCLE` und einem IEC-61131-3-Zählerbaustein (z. B. `CTU`) aufgebaut werden. Der Unterschied liegt darin, dass `SystemTick` als Subapplikation einen standardisierten ADI-Ausgang bietet und die Wertbegrenzung über Modulo realisiert, während ein herkömmlicher Zähler einen Reset-Eingang benötigt und typischerweise keine Adapter-Schnittstelle besitzt. Durch die modulare Struktur lässt sich `SystemTick` leicht in bestehende 4diac-Systeme integrieren.

Fazit

`SystemTick` ist eine kompakte, periodisch arbeitende Subapplikation zur Erzeugung eines fortlaufenden Zählerwerts im Bereich 0…99. Sie eignet sich hervorragend als Lebenszeichen für verteilte Steuerungen und bietet durch den ADI-Ausgang eine saubere Schnittstelle für den Datenaustausch in 4diac-Umgebungen. Aufgrund der einfachen Parameter (Periodendauer, Modulowert) ist sie flexibel an unterschiedliche Anforderungen anpassbar.

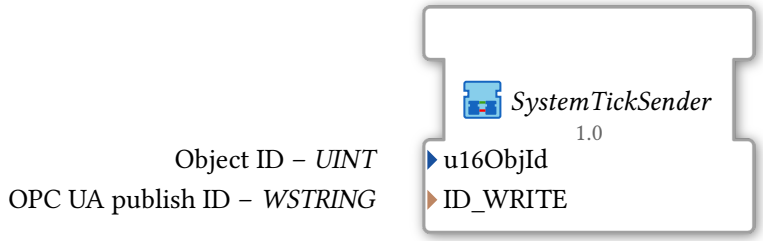
SystemTickSender



• – ◦ • – ◦ • – ◦ •

Einleitung

`SystemTickSender` liefert ein zyklisch hochzählendes „Lebenszeichen“ für VT und Web-Client — eine einfache, sofort sichtbare Bestätigung, dass die Steuerung läuft und Ereignisse verarbeitet. Der Baustein wird in mehreren Trainingsbeispielen dieses Systems eingesetzt, u. a. in `InputOutputTesterButton_DIDO_OPC_UA` und `InputOutputTesterButton_PWM_OPC_UA`.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: SystemTickSender

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **System_Tick** (SubApp, `MyLib::sys`): liefert einen zyklisch hochzählenden Wert als `ADI`-Adapter (DINT).

  - **ADI_SPLIT_2**: `adapter::events::unidirectional::ADI_SPLIT_2` — verzweigt den Zähler-Adapter in zwei Richtungen.
  - **ADI_TO_AUDI**: `adapter::conversion::unidirectional::ADI_TO_AUDI` — wandelt DINT-Adapter in UDINT-Adapter um (numerisch korrekt, siehe Numerisch vs. bitweise: `ANY_INT`⟶`ANY_INT` ist ein sicherer Cast).
  - **Q_NumericValue_AUDI**: `isobus::UT::Q::Q_NumericValue_AUDI` — schreibt den Zählerwert in ein VT-Zahlenfeld.
  - **ADI_PUBLISH_1**: `adapter::net::ADI_PUBLISH_1` — OPC-UA-Publish des rohen DINT-Zählerwerts an den Web-Client.
- **Funktionsweise**: Der intern erzeugte Zähler wird einmal für die VT-Anzeige (über UDINT-Zwischenschritt, da `Q_NumericValue_AUDI` diesen Typ erwartet) und einmal roh als DINT für OPC-UA-Publish verwendet.

Programmablauf und Verbindungen

1. `System_Tick.ADI_OUT` ⟶ `ADI_SPLIT_2.IN`.
2. `ADI_SPLIT_2.OUT1` ⟶ `ADI_TO_AUDI.ADI_IN` ⟶ `ADI_TO_AUDI.AUDI_OUT` ⟶ `Q_NumericValue_AUDI.u32NewValue` (VT-Anzeige).
3. `ADI_SPLIT_2.OUT2` ⟶ `ADI_PUBLISH_1.IN` (OPC-UA-Publish).
4. Parameter: `u16ObjId` ⟶ `Q_NumericValue_AUDI.u16ObjId`; `ID_WRITE` ⟶ `ADI_PUBLISH_1.ID`.

Technische Besonderheiten

- **Zwei parallele Verwendungen desselben Zählers**: VT-Anzeige (über `ADI_TO_AUDI`, da `Q_NumericValue_AUDI` einen UDINT-Adapter erwartet) und OPC-UA-Publish (direkt als DINT) laufen unabhängig über `ADI_SPLIT_2`.
- **Numerisch korrekte Konvertierung**: `ADI_TO_AUDI` (DINT⟶UDINT) ist ein `ANY_INT`⟶`ANY_INT`-Cast, kein Bit-Reinterpretations-Trap wie z. B. `AD_TO_AR`.

Anwendungsszenarien

- „Herzschlag“-Anzeige auf VT und Web-UI, um auf einen Blick zu erkennen, dass die Steuerung aktiv ist und nicht abgestürzt/eingefroren ist.

Zusammenfassung

`SystemTickSender` ist ein einfacher, aber in fast jedem Trainingsbeispiel dieses Systems wiederverwendeter Baustein, der einen zyklischen Zählerwert parallel an VT-Zahlenfeld und OPC-UA-Web-Client verteilt — ein minimales, aber wirksames Lebenszeichen der Steuerung.

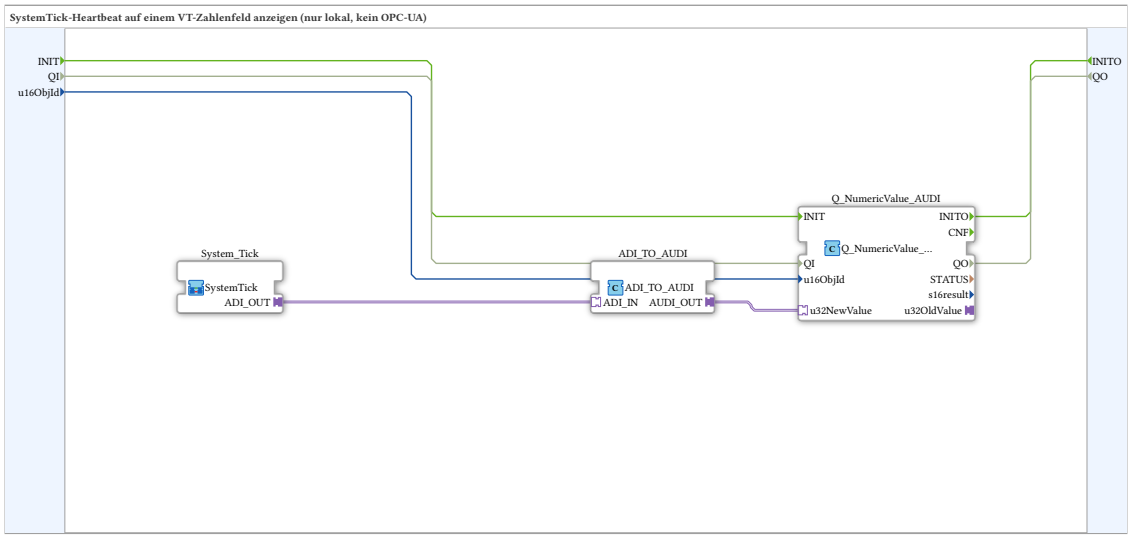
🛠️ Zugehörige Übungen

- InputOutputTesterButton_DIDO_OPC_UA
- InputOutputTesterButton_PWM_OPC_UA

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SystemTickSender_ISO



• — ○ • — ○ • — ○ •

Einleitung

`SystemTickSender_ISO` zeigt den `System_Tick`-Heartbeat rein lokal auf einem VT-Zahlenausgabefeld an, ohne OPC-UA. Der laufende Tick-Zähler wird direkt in ein VT-Zahlenfeld geschrieben, sodass am Bildschirm auf einen Blick erkennbar ist, ob die Steuerung noch lebt (der Wert muss sich alle 200 ms ändern).

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: SystemTickSender_ISO

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **System_Tick** (SubApp, `MyLib::sys`): liefert den Zählerstand als `ADI`-Adapter (DINT).
  - **ADI_TO_AUDI**: `adapter::conversion::unidirectional::ADI_TO_AUDI` — wandelt den DINT-Wert in `AUDI` (UDINT) um, weil `Q_NumericValue_AUDI` einen vorzeichenlosen Wert erwartet.
  - **Q_NumericValue_AUDI**: `isobus::UT::Q::Q_NumericValue_AUDI` — VT-Kommando „Change numeric value“ (Part 6 – F.22), schreibt den Wert in das Zahlenausgabefeld `u16ObjId`.
- **Funktionsweise**: `System_Tick` liefert den Zählerstand, `ADI_TO_AUDI` wandelt ihn in den von `Q_NumericValue_AUDI` erwarteten Datentyp um.

Programmablauf und Verbindungen

1. `u16ObjId` ⟶ `Q_NumericValue_AUDI.u16ObjId` (Datenverbindung, ausgeblendet).
2. `System_Tick.ADI_OUT` ⟶ `ADI_TO_AUDI.ADI_IN` ⟶ `ADI_TO_AUDI.AUDI_OUT` ⟶ `Q_NumericValue_AUDI.u32NewValue`.
3. `INIT` ⟶ `Q_NumericValue_AUDI.INIT`; `Q_NumericValue_AUDI.INITO` ⟶ `INITO` (neuer Ereignis-Eingang `INIT` / Ereignis-Ausgang `INITO`, jeweils `EInit`).
4. `QI` (`BOOL`, Vorgabe `TRUE`) ⟶ `Q_NumericValue_AUDI.QI`; `Q_NumericValue_AUDI.QO` ⟶ `QO` (`BOOL`, Quality Output – `TRUE`, wenn der VT-Schreibzugriff erfolgreich war).

Technische Besonderheiten

- **Rein lokal, kein OPC-UA**: Im Gegensatz zu `SystemTickSender_ISO_OPC` besitzt dieser Baustein keinen `ADI_PUBLISH_1`-Adapter.

Anwendungsszenarien

- Module mit eigener VT-Anbindung, die den Heartbeat nur lokal sichtbar machen müssen, ohne Remote-Überwachung.

Vergleich mit ähnlichen Bausteinen

Für zusätzliche Remote-Verteilung ist `SystemTickSender_ISO_OPC` zu verwenden; für ein Modul ohne eigene VT-Anbindung (nur OPC-UA) `SystemTickSender_OPC`. Gegenüber `SystemTickSender` (VT + OPC-UA, älteres Muster mit `ADI_SPLIT_2`) ist dieser Baustein die reine VT-only-Variante ohne Split/Publish.

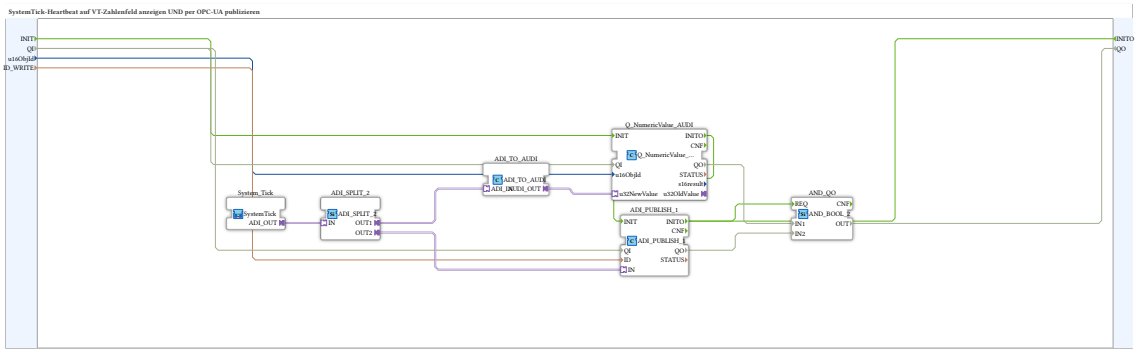
Zusammenfassung

`SystemTickSender_ISO` zeigt den SystemTick-Heartbeat ausschließlich auf einem lokalen VT-Zahlenfeld an — die schlankste Variante für Module mit eigener VT-Anbindung ohne Remote-Überwachung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

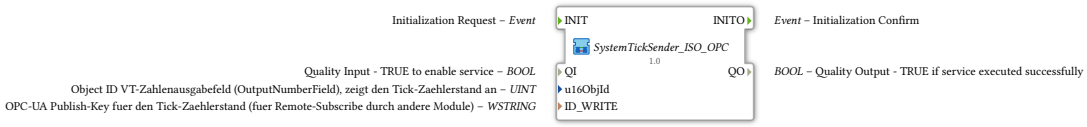
SystemTickSender_ISO_OPC



• — ○ • — ○ • — ○ •

Einleitung

`SystemTickSender_ISO_OPC` kombiniert `SystemTickSender_ISO` (Anzeige auf einem lokalen VT-Zahlenfeld) mit `SystemTickSender_OPC` (Remote-Publish per OPC-UA), damit dasselbe Lebenszeichen sowohl am eigenen Bildschirm sichtbar ist als auch von anderen Modulen per Remote-Subscribe überwacht werden kann.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: SystemTickSender_ISO_OPC

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **System_Tick** (SubApp, `MyLib::sys`): liefert den Zählerstand als `ADI`-Adapter (DINT).
  - **ADI_SPLIT_2**: `adapter::events::unidirectional::ADI_SPLIT_2` — verzweigt `System_Tick.ADI_OUT` auf zwei Ziele, da ein Adapter-Socket nur eine Quelle annehmen kann: ein Ausgang geht (über `ADI_TO_AUDI`) an die VT-Anzeige, der andere an `ADI_PUBLISH_1` für den OPC-UA-Publish.
  - **ADI_TO_AUDI**: `adapter::conversion::unidirectional::ADI_TO_AUDI` — wandelt DINT in UDINT um, für die VT-Anzeige.
  - **Q_NumericValue_AUDI**: `isobus::UT::Q::Q_NumericValue_AUDI` — zeigt den Wert lokal auf dem VT an (`u16ObjId`).
  - **ADI_PUBLISH_1**: `adapter::net::ADI_PUBLISH_1` (`QI` kommt vom Eingang `QI`) — veröffentlicht denselben Zählerwert per OPC-UA (`ID_WRITE`).

  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_2` — verknüpft `Q_NumericValue_AUDI.QO` und `ADI_PUBLISH_1.QO` zum Ausgang `QO`.
- **Funktionsweise**: Der Zählerstand von `System_Tick` wird per `ADI_SPLIT_2` verzweigt: ein Zweig geht wie bei `SystemTickSender_ISO` über `ADI_TO_AUDI` an die VT-Anzeige, der andere direkt an `ADI_PUBLISH_1` für den OPC-UA-Publish — wie bei `SystemTickSender_OPC`.

Programmablauf und Verbindungen

1. `System_Tick.ADI_OUT` ⟶ `ADI_SPLIT_2.IN`.
2. `ADI_SPLIT_2.OUT1` ⟶ `ADI_TO_AUDI.ADI_IN` ⟶ `ADI_TO_AUDI.AUDI_OUT` ⟶ `Q_NumericValue_AUDI.u32NewValue` (VT-Anzeige).
3. `ADI_SPLIT_2.OUT2` ⟶ `ADI_PUBLISH_1.IN` (OPC-UA-Publish).
4. Parameter: `u16ObjId` ⟶ `Q_NumericValue_AUDI.u16ObjId`; `ID_WRITE` ⟶ `ADI_PUBLISH_1.ID`.
5. Initialisierungskette (Ereignis-Eingang `INIT` / Ereignis-Ausgang `INITO`, jeweils `EInit`): `INIT` ⟶ `Q_NumericValue_AUDI.INIT`; `Q_NumericValue_AUDI.INITO` ⟶ `ADI_PUBLISH_1.INIT`; `ADI_PUBLISH_1.INITO` ⟶ `INITO` und `AND_QO.REQ`.
6. Qualität: `QI` (`BOOL`, Vorgabe `TRUE`) ⟶ `Q_NumericValue_AUDI.QI` und `ADI_PUBLISH_1.QI`; `Q_NumericValue_AUDI.QO` ⟶ `AND_QO.IN1`; `ADI_PUBLISH_1.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO` (`BOOL`, Quality Output – `TRUE`, wenn VT-Schreibzugriff und OPC-UA-Publish erfolgreich waren).

Technische Besonderheiten

- **Ein Zähler, zwei Verwendungen**: `ADI_SPLIT_2` erlaubt die parallele Nutzung desselben `System_Tick`-Zählers für VT-Anzeige und OPC-UA-Publish, ohne den Zähler zu duplizieren.

Anwendungsszenarien

- Module mit eigener VT-Anbindung, deren Heartbeat zusätzlich von anderen Modulen per Remote-Subscribe überwacht werden soll.

Vergleich mit ähnlichen Bausteinen

Wird nur eine der beiden Ausgaben gebraucht, sind die einfacheren `SystemTickSender_ISO` (nur VT) bzw. `SystemTickSender_OPC` (nur OPC-UA) zu verwenden. Dieses Muster (Split ⟶ VT-Anzeige + OPC-UA-Publish) entspricht strukturell dem älteren `SystemTickSender`.

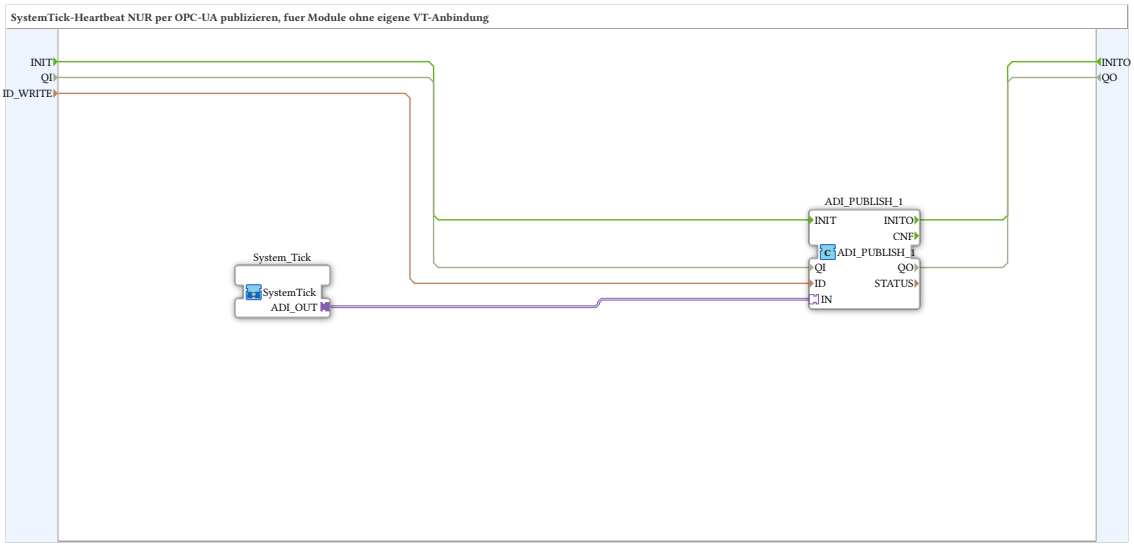
Zusammenfassung

`SystemTickSender_ISO_OPC` zeigt den SystemTick-Heartbeat gleichzeitig lokal auf dem VT und remote per OPC-UA an, indem es die beiden schlankeren Einzelvarianten zu einem Baustein zusammenführt.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SystemTickSender_OPC



• – ○ • – ○ • – ○ •

Einleitung

`SystemTickSender_OPC` ist die schlankste SystemTick-Variante für Module ohne eigene VT-Anbindung: Der laufende Tick-Zähler von `System_Tick` wird direkt per OPC-UA veröffentlicht, ohne lokale VT-Anzeige. Ein anderes Modul mit VT kann diesen Wert per Remote-Subscribe abonnieren und anzeigen, um die Lebendigkeit dieses Moduls zu überwachen.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: SystemTickSender_OPC

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **System_Tick** (SubApp, `MyLib::sys`): liefert den Zählerstand als `ADI`-Adapter (DINT).
  - **ADI_PUBLISH_1**: `adapter::net::ADI_PUBLISH_1` — veröffentlicht den Zählerstand direkt per OPC-UA (`ID_WRITE`).
- **Funktionsweise**: Der Zählerstand von `System_Tick` wird ohne Umwandlung direkt an `ADI_PUBLISH_1` weitergereicht.

Programmablauf und Verbindungen

1. `ID_WRITE` ⟶ `ADI_PUBLISH_1.ID` (Datenverbindung, ausgeblendet).
2. `System_Tick.ADI_OUT` ⟶ `ADI_PUBLISH_1.IN`.
3. Initialisierung: Ereigniseingang `INIT` ⟶ `ADI_PUBLISH_1.INIT`; `ADI_PUBLISH_1.INITO` ⟶ Ereignisausgang `INITO`.
4. Qualität: Dateneingang `QI` (`BOOL`, Default `TRUE`) ⟶ `ADI_PUBLISH_1.QI`; `ADI_PUBLISH_1.QO` ⟶ Datenausgang `QO`.

Technische Besonderheiten

- **Keine Typumwandlung nötig**: Da keine VT-Anzeige beteiligt ist, entfällt die `ADI_TO_AUDI`-Umwandlung — der rohe DINT-Wert wird direkt veröffentlicht.

Anwendungsszenarien

- Module ohne eigene VT-Anbindung (z. B. reine IO-/Feldmodule), deren Lebendigkeit von einem anderen Modul mit VT überwacht werden soll.

Vergleich mit ähnlichen Bausteinen

Wird zusätzlich eine lokale VT-Anzeige gebraucht, ist `SystemTickSender_ISO_OPC` (VT + OPC-UA) bzw. `SystemTickSender_ISO` (nur VT) zu verwenden.

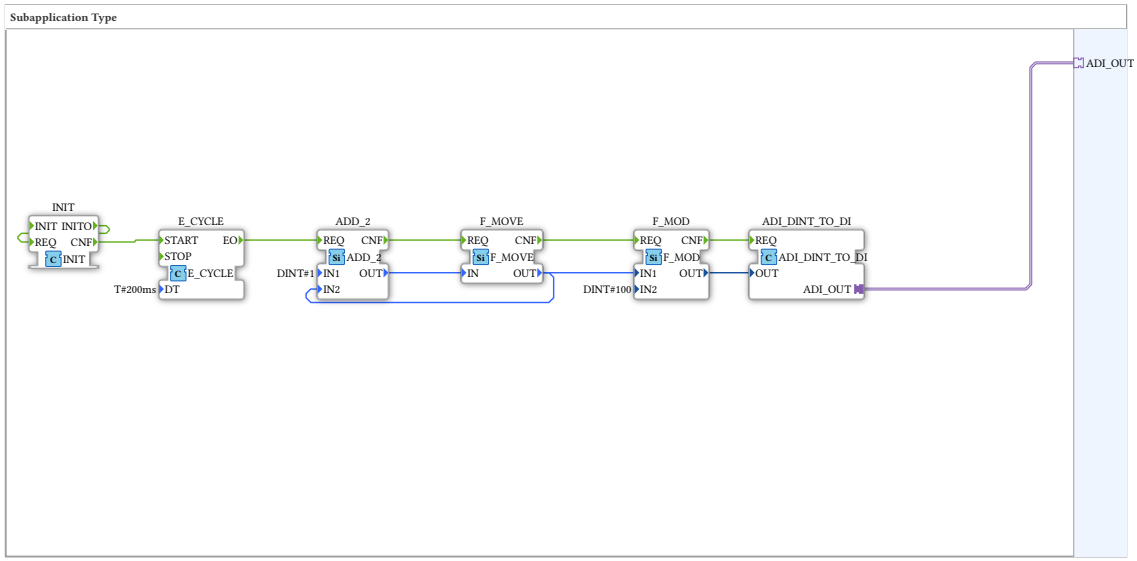
Zusammenfassung

`SystemTickSender_OPC` veröffentlicht den SystemTick-Heartbeat ausschließlich per OPC-UA — die minimale Variante für Module ohne eigene VT-Anbindung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

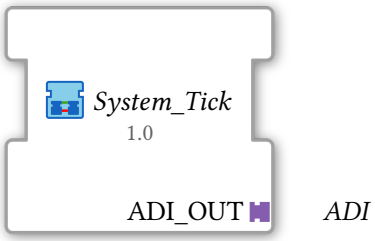
System_Tick



• – ○ • – ○ • – ○ •

Einleitung

`System_Tick` erzeugt einen fortlaufenden, zyklisch inkrementierten DINT-Zähler (200-ms-Takt, Wertebereich 1..100, danach Rücksprung auf 1 via Modulo) und stellt ihn über einen `ADI`-Adapter (DINT) für andere Bausteine bereit — ein einfacher „Lebenszeichen"- bzw. Heartbeat-Zähler.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: System_Tick

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **INIT**: `iec61131::booleanOperators::INIT` — feuert einmalig beim Start (`INITO` ⟶ `REQ` rückgekoppelt) und startet darüber den Zyklus-Timer.
  - **E_CYCLE**: `iec61499::events::E_CYCLE` — zyklisches Ereignis, `DT=T#200ms`, gestartet über `INIT.CNF`.
  - **ADD_2**: `iec61131::arithmetic::ADD_2` — addiert `IN1=DINT#1` auf den aktuellen Zählerwert (`IN2`, rückgekoppelt aus `F_MOVE.OUT`).

  - **F_MOVE**: `iec61131::selection::F_MOVE` (`DataType=DINT`) — hält/übergibt den aktuellen Zählerwert, dient als Zwischenspeicher der Rückkopplungsschleife.
  - **F_MOD**: `iec61131::arithmetic::F_MOD` — Modulo-Operation mit `IN2=DINT#100`, begrenzt den Wertebereich auf 0..99.
  - **ADI_DINT_TO_DI**: `adapter::conversion::unidirectional::ADI_DINT_TO_DI` — wandelt den DINT-Wert in den `ADI_OUT`-Adapter zur externen Nutzung.
- **Funktionsweise**: `INIT` startet den `E_CYCLE`-Timer; jeder Zyklus erhöht den Zählerwert über `ADD_2`/`F_MOVE` um 1, `F_MOD` hält ihn im Bereich 0..99, und `ADI_DINT_TO_DI` legt das Ergebnis auf den Ausgangsadapter.

Programmablauf und Verbindungen

1. `INIT.INITO` ⟶ `INIT.REQ` (Selbstauslösung beim Start); `INIT.CNF` ⟶ `E_CYCLE.START`.
2. `E_CYCLE.EO` ⟶ `ADD_2.REQ` (alle 200 ms).
3. `F_MOVE.OUT` ⟶ `ADD_2.IN2` (Rückkopplung des aktuellen Werts); `ADD_2.OUT` ⟶ `F_MOVE.IN`.
4. `ADD_2.CNF` ⟶ `F_MOVE.REQ` ⟶ `F_MOD.REQ` (Ereigniskette); `F_MOVE.OUT` ⟶ `F_MOD.IN1`.
5. `F_MOD.CNF` ⟶ `ADI_DINT_TO_DI.REQ`; `F_MOD.OUT` ⟶ `ADI_DINT_TO_DI.OUT`.
6. `ADI_DINT_TO_DI.ADI_OUT` (Adapter) ⟶ `ADI_OUT` (SubApp-Schnittstelle).

Technische Besonderheiten

- Der Zähler läuft rein intern im Baustein (keine externen Dateneingänge) — er ist ein autonomer Taktgeber, der sich selbst über `INIT` anstößt.
- Die Modulo-Grenze `DINT#100` ergibt einen Wertebereich von 0..99, nicht 1..100 wie man beim ersten Blick vermuten könnte — der Startwert nach `INIT` ist implizit 0, danach zählt `ADD_2` jeden Zyklus um 1 hoch.

Anwendungsszenarien

- Heartbeat- oder Lebenszeichen-Signal für Diagnosezwecke, z. B. um auf dem VT oder per OPC-UA sichtbar zu machen, dass die Steuerung aktiv zyklisch arbeitet.
- Einfacher Sekundentakt-Ersatz (5 Zyklen à 200 ms ≈ 1 s) für Testzwecke, ohne dass ein dedizierter Zeitbaustein konfiguriert werden muss.

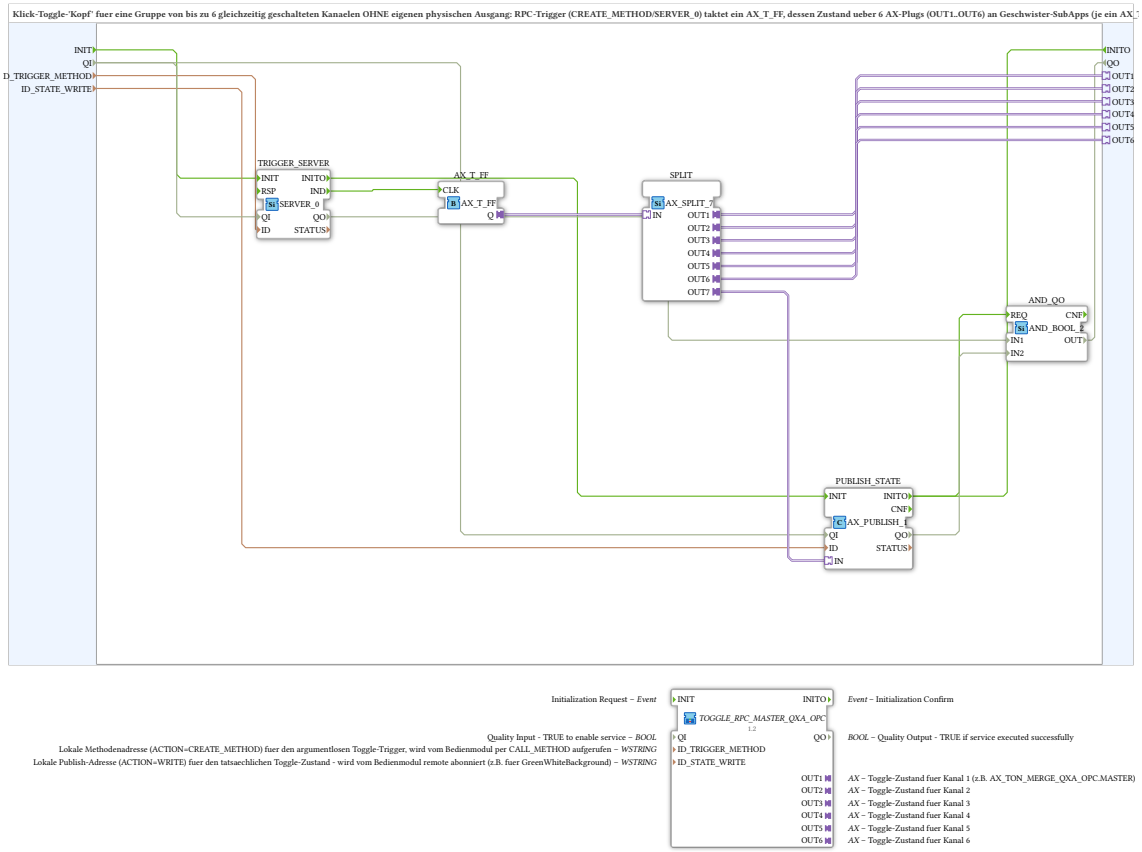
Zusammenfassung

Autonomer, selbstauslösender 200-ms-Zähler mit Modulo-Begrenzung, bereitgestellt über einen DINT-Adapter — ein einfacher Heartbeat-Baustein.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

TOGGLE_RPC_MASTER_QXA_OPC



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock `TOGGLE_RPC_MASTER_QXA_OPC` ist eine Subapplikation (SubApp) und dient als „Klick-Toggle-Kopf“ für eine Gruppe von bis zu sechs gleichzeitig geschalteten Kanälen. Er besitzt **keinen eigenen physischen Ausgang**, sondern gibt den Toggle-Zustand über sechs nach außen geführte Adapter-Plugs (`OUT1` bis `OUT6`) an sogenannte Geschwister-SubApps weiter – typischerweise eine `AX_TON_MERGE_QXA_OPC`-Instanz pro Kanal. Zusätzlich wird der Zustand über eine OPC-UA-Publish-Adresse lokal veröffentlicht, damit ein Bedienmodul den aktuellen Zustand (z.B. als Hintergrundfarbe) anzeigen kann.

Die SubApp ist speziell für Anwendungen konzipiert, bei denen ein einzelner Softkey mehrere Ausgänge (z.B. eine Scheinwerferbank) synchron toggeln soll, wobei jeder Kanal mit individueller, zeitversetzter Einschaltrampe (über `PT` der nachgeschalteten Bausteine) betrieben werden kann, um kapazitive Einschaltstrom-Spitzen zu vermeiden.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung (von `PUBLISH_STATE.INITO`)

Daten-Eingänge

Name	Typ	Kommentar
`QI`	`BOOL`	Quality Input, TRUE gibt den Dienst frei (Vorgabe `TRUE`); wird an `TRIGGER_SERVER.QI` und `PUBLISH_STATE.QI` verteilt.

ID_TRIGGER_METHOD	WSTRING	Lokale Methodenadresse (ACTION=CREATE_METHOD) für den argumentlosen Toggle-Trigger, wird vom Bedienmodul per CALL_METHOD aufgerufen.
ID_STATE_WRITE	WSTRING	Lokale Publish-Adresse (ACTION=WRITE) für den tatsächlichen Toggle-Zustand – wird vom Bedienmodul remote abonniert (z.B. für GreenWhiteBackground).

Daten-Ausgänge

- **QO** (BOOL): Quality Output, UND aus TRIGGER_SERVER.QO und PUBLISH_STATE.QO (AND_QO, iec61131::booleanOperators::AND_BOOL_2)

Adapter

Name	Typ	Kommentar
OUT1	adapter::types::unidirectional::AX	Toggle-Zustand für Kanal 1 (z.B. an AX_TON_MERGE_QXA_OPC.MASTER).
OUT2	adapter::types::unidirectional::AX	Toggle-Zustand für Kanal 2.
OUT3	adapter::types::unidirectional::AX	Toggle-Zustand für Kanal 3.
OUT4	adapter::types::unidirectional::AX	Toggle-Zustand für Kanal 4.
OUT5	adapter::types::unidirectional::AX	Toggle-Zustand für Kanal 5.
OUT6	adapter::types::unidirectional::AX	Toggle-Zustand für Kanal 6.

Funktionsweise

Die SubApp realisiert einen Remote-Toggle mit OPC-UA. Intern sind folgende Funktionsbausteine miteinander verdrahtet:

- **SERVER_0** (iec61499::net::SERVER_0): Ein OPC-UA-Server mit CREATE_METHOD-Support. Er bietet die unter ID_TRIGGER_METHOD definierte Methodenadresse an.
- **AX_T_FF** (adapter::events::unidirectional::AX_T_FF): Ein Toggle-Flip-Flop, das bei jedem ankommenden Ereignis seinen Ausgangszustand Q umschaltet.
- **AX_SPLIT_7** (adapter::events::unidirectional::AX_SPLIT_7): Verteilt den Zustand Q auf sieben unidirektionale Adapter-Ausgänge.
- **AX_PUBLISH_1** (adapter::net::AX_PUBLISH_1): Ein OPC-UA-Publisher, der den Zustand auf die unter ID_STATE_WRITE vorgegebene Adresse schreibt.
- **AND_QO** (iec61131::booleanOperators::AND_BOOL_2): Verknüpft TRIGGER_SERVER.QO und PUBLISH_STATE.QO zum QO der SubApp.

Ablauf:

Wird die Methode per CALL_METHOD aufgerufen, erzeugt der Server ein IND-Ereignis. Dieses Ereignis taktet das Toggle-Flip-Flop (CLK), wodurch sich dessen Ausgang Q (binär 0/1) ändert. Gleichzeitig wird das IND-Ereignis direkt zurück auf RSP gelegt, um die Anfrage sofort zu quittieren und den OPC-UA-Server nicht zu blockieren. Der aktuelle Zustand wird über den Split auf die sieben Ausgänge verteilt: Sechs davon werden auf die externen Plugs OUT1…OUT6 geführt, der siebte (OUT7) geht an den Publisher, der den Zustand als WRITE-Operation auf der OPC-UA-Adresse veröffentlicht. Dadurch kann ein Bedienmodul den Zustand abonnieren und z.B. die Hintergrundfarbe eines Softkeys ändern.

Außer INIT/INITO hat die SubApp **keine** Ereignis-Eingänge oder -Ausgänge; im Betrieb wird sie ausschließlich über den OPC-UA-Methodenaufruf gesteuert.

Initialisierung und Qualität: INIT → TRIGGER_SERVER.INIT; TRIGGER_SERVER.INITO → PUBLISH_STATE.INIT; PUBLISH_STATE.INITO → INITO (und → AND_QO.REQ). QI → TRIGGER_SERVER.QI und PUBLISH_STATE.QI; TRIGGER_SERVER.QO → AND_QO.IN1, PUBLISH_STATE.QO → AND_QO.IN2, AND_QO.OUT → QO.

Technische Besonderheiten

- **Direkte RSP-Verdrahtung:** Das Ereignis IND wird nicht nur an das Flip-Flop, sondern auch direkt an RSP gelegt. Dadurch wird die OPC-UA-Anfrage sofort beantwortet, ohne auf weitere Bausteine zu

warten. Frühere Versionen ließen dies aus, was den OPC-UA-Server bis zu 4 Sekunden blockieren konnte (globaler `serviceMutex` in open62541) und jede Statusmeldung im gesamten Projekt verzögerte.
- **1:1-Adapterverbindungen:** Da IEC 61499-Adapterverbindungen strikt 1:1 sind, kann ein einzelner Adapter-Plug nicht mehrere Sockets gleichzeitig versorgen. Deshalb besitzt die SubApp sechs separate `OUT`-Plugs (`AX_SPLIT_7`), sodass bis zu sechs unabhängige Kanäle angeschlossen werden können.
- **Kein eigener physischer Ausgang:** Der Toggle-Zustand wird nur als Software-Signal über die Adapter weitergegeben. Für einen einzelnen Kanal mit eigenem Ausgang ist der Baustein `TOGGLE_RPC_MERGE_QXA_OPC` vorgesehen.
- **QI-Verteilung:** `QI` der Server- und Publisher-Bausteine ist nicht mehr fest auf `TRUE` gesetzt, sondern wird vom `QI`-Eingang der SubApp (Vorgabe `TRUE`) gespeist.

Zustandsübersicht

Die SubApp selbst besitzt keine eigene Zustandsmaschine, der wesentliche Zustand ist das interne Toggle-Flip-Flop `AX_T_FF`. Dessen Ausgang `Q` kann zwei Zustände annehmen:

Zustand	Bedeutung
`0`	Kanalgruppe ausgeschaltet (z.B. Licht aus)
`1`	Kanalgruppe eingeschaltet (z.B. Licht an)

Jeder am `OUT`-Plug angeschlossene nachfolgende Baustein (z.B. `AX_TON_MERGE_QXA_OPC`) wertet diesen Wert aus und steuert den physischen Ausgang entsprechend. Der Zustand wird auch über den Publisher kontinuierlich verfügbar gemacht.

Anwendungsszenarien

- **Scheinwerferbank:** Ein einziger Softkey toggelt eine Gruppe von bis zu sechs Scheinwerfern. Jeder Kanal besitzt seine eigene `AX_TON_MERGE_QXA_OPC`-Instanz, die den Toggle-Zustand entgegennimmt und mit einer individuellen Einschaltverzögerung (`PT`) versieht, um Stromspitzen zu vermeiden.
- **Zentrale Lichtsteuerung:** Ein Bedienmodul sendet per OPC-UA einen Methodenaufruf an die SubApp; gleichzeitig abonniert es den Zustand und visualisiert ihn farblich (z.B. grün/weiß).
- **Redundante Steuerung:** Da mehrere Kanäle parallel geschaltet werden, können Ausfälle einzelner Ausgänge erkannt werden, ohne die gesamte Gruppe zu beeinträchtigen.

Vergleich mit ähnlichen Bausteinen

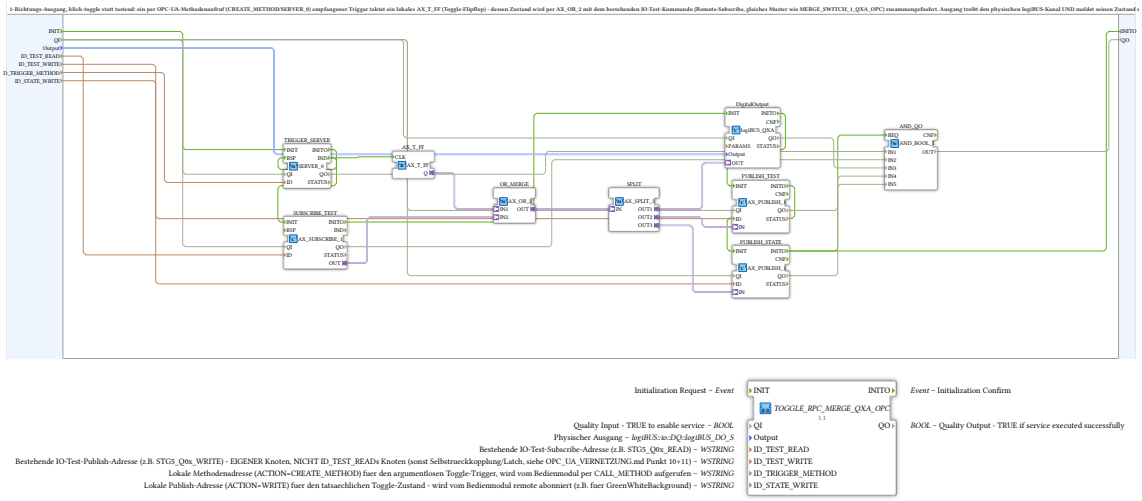
Baustein	Beschreibung	Unterschied
`TOGGLE_RPC_MASTER_QXA_OPC`	**Ohne** eigenen physischen Ausgang, verteilt Zustand an bis zu 6 externe Kanäle.	Besitzt keine eigene Ausgangslogik; erzeugt nur das Software-Toggle-Signal.
`TOGGLE_RPC_MERGE_QXA_OPC`	Wie oben, aber **mit** eigenem physischen Ausgang (für einen einzelnen Kanal).	Besitzt den physischen Ausgang integriert; nur für einen Kanal geeignet.

Der Master-Baustein eignet sich also für Mehrkanal-Szenarien, bei denen die Ausgangslogik auf separaten Bausteinen liegt – beispielsweise wenn eine Sortierung oder Verzögerung pro Kanal gewünscht ist.

Fazit

`TOGGLE_RPC_MASTER_QXA_OPC` ist ein kompakter und flexibler Baustein zur zentralen Steuerung mehrerer Ausgangskanäle über einen einzigen OPC-UA-Methodenaufruf. Durch die klare Trennung von Toggle-Logik und Ausgangsverarbeitung (per Adapter) lässt sich eine große Anzahl von Kanälen mit minimalem Ressourceneinsatz realisieren. Die saubere Interaktion mit OPC-UA (direkte RSP-Antwort, Publish) gewährleistet ein reaktionsschnelles Verhalten auch in verteilten Systemen. Für Anwendungen mit nur einem Kanal ist jedoch der spezifischere Baustein `TOGGLE_RPC_MERGE_QXA_OPC` empfehlenswert.

TOGGLE_RPC_MERGE_QXA_OPC



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock **TOGGLE_RPC_MERGE_QXA_OPC** realisiert einen „Klick-Toggle-Ausgang“ für den logiBUS. Ein einmaliger Tastendruck (z. B. auf einem SoftKey) schaltet einen physischen Ausgang dauerhaft ein oder aus – genau wie bei einem Blitzlicht. Der Trigger erfolgt über einen OPC-UA-Methodenaufruf (RPC), nicht über einen Wertwechsel. Der interne Zustand wird durch ein Toggle-Flipflop gehalten und mit einem bestehenden IO-Test-Kommando (Remote-Subscribe) logisch ODER-verknüpft, sodass der physische Ausgang sowohl durch die echte Funktion als auch durch den IO-Test angesteuert werden kann. Zusätzlich werden zwei getrennte Rückmeldekanäle über OPC-UA-Publish bereitgestellt: einer für die IO-Diagnose, einer für die eigentliche Funktionsrückmeldung (z. B. zur Hintergrundbeleuchtung eines SoftKeys).

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialization Request

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialization Confirm

Daten-Eingänge

Name	Typ	Beschreibung
`QI`	`BOOL`	Quality Input – TRUE aktiviert den Dienst (Initialwert: `TRUE`)
`Output`	`logiBUS::io::DQ::logiBUS_DO_S`	Physischer Ausgangswert (Initialwert: `logiBUS_DO::Invalid`)
`ID_TEST_READ`	`WSTRING`	Bestehende IO-Test-Subscribe-Adresse (z. B. `STG5_Q0x_READ`)
`ID_TEST_WRITE`	`WSTRING`	Bestehende IO-Test-Publish-Adresse (z. B. `STG5_Q0x_WRITE`) – eigener Knoten, nicht `ID_TEST_READ` (sonst Selbstkopplung/ Latch)
`ID_TRIGGER_METHOD`	`WSTRING`	Lokale Methodenadresse (ACTION=CREATE_METHOD) für den argumentlosen Toggle-Trigger

ID_STATE_WRITE	WSTRING	Lokale Publish-Adresse (ACTION=WRITE) für den tatsächlichen Toggle-Zustand (remote abonniert, z. B. für GreenWhite-Background)

Daten-Ausgänge

- **QO** (BOOL): Quality Output, UND-Verknüpfung (AND_QO, AND_BOOL_5) der QO-Ausgänge von TRIGGER_SERVER, SUBSCRIBE_TEST, DigitalOutput, PUBLISH_TEST und PUBLISH_STATE. Die eigentlichen Ergebnisse werden weiterhin über die internen OPC-UA-Publish-Bausteine geliefert.

Adapter

Keine.

Funktionsweise

1. **Trigger-Empfang**: Ein OPC-UA-Client ruft über die durch ID_TRIGGER_METHOD definierte Methode den SERVER_0-Baustein (TRIGGER_SERVER) auf. Das Ereignis IND wird ausgelöst.
2. **Toggle-Flipflop**: Das Ereignis IND taktet das Flipflop AX_T_FF. Bei jedem Aufruf wechselt dessen Ausgang Q zwischen TRUE und FALSE. Gleichzeitig wird IND direkt auf RSP des Servers verdrahtet, um eine sofortige Bestätigung zu geben (keine Blockade des OPC-UA-Threads).
3. **Zusammenführung mit IO-Test**: Der Subskriptionsbaustein SUBSCRIBE_TEST empfängt kontinuierlich den Zustand des bestehenden IO-Test-Kanals (ID_TEST_READ) und gibt ihn als Booleschen Wert an IN2 des ODER-Gatters (OR_MERGE) weiter. Der Flipflop-Zustand (Q) wird an IN1 gelegt.
4. **ODER-Verknüpfung**: Der Ausgang des ODER-Gatters ist TRUE, wenn entweder der Toggle-Zustand oder das IO-Test-Kommando aktiv ist. Dadurch bleibt der physische Ausgang während eines IO-Tests unabhängig von der Toggle-Logik steuerbar.
5. **Signalverteilung**: Das ODER-Signal wird über den Splitter SPLIT an drei Ziele geführt:
   - OUT1 ⟶ physischer Ausgang (DigitalOutput)
   - OUT2 ⟶ IO-Test-Rückmeldung über PUBLISH_TEST (Adresse ID_TEST_WRITE)
   - OUT3 ⟶ Funktions-Rückmeldung über PUBLISH_STATE (Adresse ID_STATE_WRITE)
6. **Publish**: Beide Publikationsbausteine senden den aktuellen Zustand als OPC-UA-Variable an die jeweiligen Adressen. Das Bedienmodul abonniert diese Werte für Anzeige oder Hintergrundbeleuchtung.
7. **Initialisierung und Qualität**: INIT ⟶ TRIGGER_SERVER.INIT ⟶ SUBSCRIBE_TEST.INIT ⟶ DigitalOutput.INIT ⟶ PUBLISH_TEST.INIT ⟶ PUBLISH_STATE.INIT; PUBLISH_STATE.INITO ⟶ INITO und ⟶ AND_QO.REQ. QI wird an alle fünf Bausteine verteilt (kein fest verdrahteter QI-Parameter mehr); deren QO laufen in dieser Reihenfolge auf AND_QO.IN1 bis AND_QO.IN5, AND_QO.OUT ⟶ QO.

Technische Besonderheiten

- **RPC-getriggert statt Pegel-getriggert**: Der Toggle-Impuls kommt über einen Methodenaufruf (CALL_METHOD), nicht über einen Wertwechsel. Dadurch ist die Funktion unabhängig von Signalflanken und benötigt kein Polling.
- **Kein aktives Client-Write für Rückmeldung**: Anders als das ursprüngliche Übungsmodell (Toggle_RPC_FROM_Remote_QXA_OPC) wird der Zustand nicht aktiv per CLIENT_1_0 ans Bedienmodul geschrieben, sondern lokal publiziert und remote abonniert – konsistent mit allen anderen Funktionen des Projekts (z. B. ILOCK_SWITCH_2_QXA_OPC).
- **Getrennte Rückmeldeknoten**: Für IO-Test und Funktionszustand werden unterschiedliche OPC-UA-Knoten verwendet, um Selbstkopplung und Latch-Effekte zu vermeiden (siehe OPC_UA_VERNETZUNG.md, Punkte 10 und 11).
- **Sofortige RSP-Antwort**: Der TRIGGER_SERVER beantwortet den Methodenaufruf sofort (direkte Verbindung IND → RSP). Dies verhindert Verzögerungen des OPC-UA-Servers und blockiert nicht den globalen Service-Mutex.

Zustandsübersicht

Die Subapp selbst besitzt keinen eigenen Zustandsautomaten. Der relevante Zustand wird im Toggle-Flipflop AX_T_FF gehalten:

- **Zustand Q = FALSE** (Aus): Der Ausgang ist inaktiv. Ein Trigger-Event setzt Q auf TRUE.
- **Zustand Q = TRUE** (Ein): Der Ausgang ist aktiv. Ein Trigger-Event setzt Q zurück auf FALSE.

Der physische Ausgang ist die ODER-Verknüpfung aus Q und dem IO-Test-Signal. Dadurch kann der Ausgang auch dann aktiv sein, wenn Q = FALSE, solange der IO-Test ein `TRUE` liefert.

Anwendungsszenarien

- **Blitzlichtsteuerung**: Ein SoftKey wird kurz gedrückt, um das Licht ein- oder auszuschalten (Toggle). Der Trigger kommt per RPC, der Zustand wird zurückgemeldet.
- **Beleuchtungsumschaltung**: Umschalten zwischen zwei Betriebsmodi (z. B. Automatik/Manuell) über eine Taste.
- **Erweiterung vorhandener IO-Test-Funktionen**: Integration in bestehende logiBUS-Kanäle, ohne den IO-Test zu beeinträchtigen.

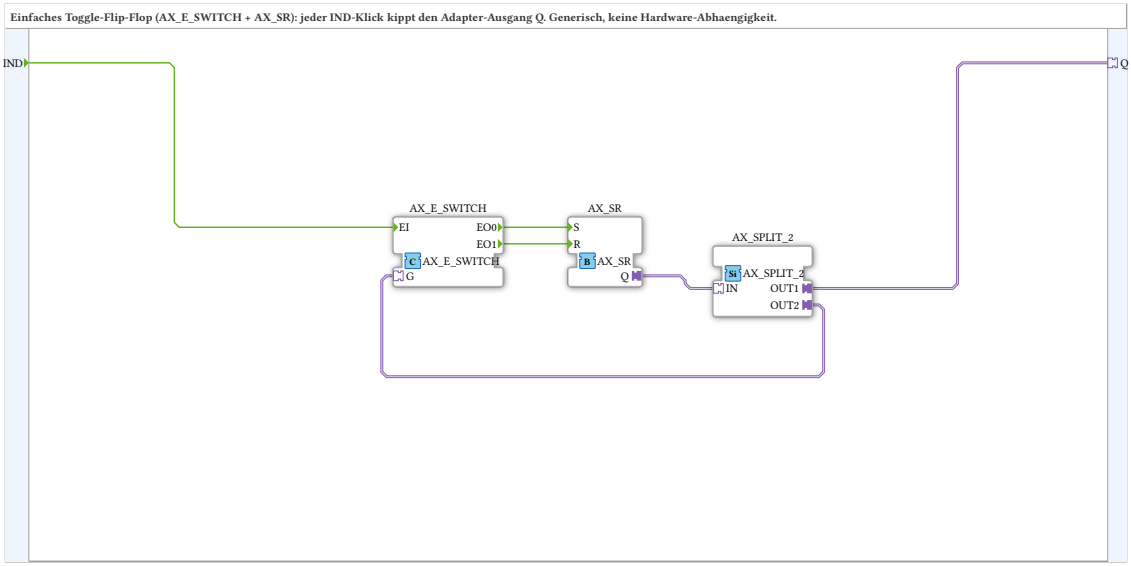
Vergleich mit ähnlichen Bausteinen

- **MERGE_SWITCH_1_QXA_OPC**: Dieser Baustein schaltet wechselweise (tastend) – der Ausgang ist nur während des gedrückten Zustands aktiv. `TOGGLE_RPC_MERGE_QXA_OPC` erweitert dies um ein Flipflop, sodass der Zustand bis zum nächsten Tastendruck erhalten bleibt.
- **ILOCK_SWITCH_2_QXA_OPC**: Ähnlich wie `MERGE_SWITCH_1_QXA_OPC`, aber mit einer Verriegelungslogik. Der hier vorgestellte Baustein ist speziell für Klick-Toggle-Aktionen konzipiert und verwendet eine andere Trigger-Quelle (RPC-Methode statt Pegel-Subscription).
- **Toggle_RPC_FROM_Remote_QXA_OPC** (Übungsmodell): Das ursprüngliche Vorbild nutzt noch ein aktives Client-Write für die Rückmeldung; dieser Baustein wurde aus Konsistenzgründen auf das Publish/Subscribe-Muster umgestellt.

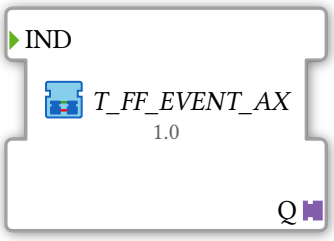
Fazit

`TOGGLE_RPC_MERGE_QXA_OPC` bietet eine robuste und konsistente Lösung für Klick-Toggle-Funktionen auf Basis von OPC-UA-Methodenaufrufen. Durch die Kombination von Toggle-Flipflop und IO-Test-Einspeisung sowie die getrennten Rückmeldekanäle ist er flexibel in bestehende logiBUS-Umgebungen integrierbar und vermeidet typische Fehler wie Selbstkopplung oder Threadblockaden.

T_FF_EVENT_AX



Event Input (Clock / Click) – *Event*



AX – Adapter Output Q

• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **T_FF_EVENT_AX** ist eine Subapp, die ein einfaches Toggle-Flip-Flop realisiert. Jeder Ereignisimpuls am Eingang `IND` kippt den Zustand des Adapter-Ausgangs `Q`. Die Implementierung verwendet die modularen 4diac-Bausteine `AX_E_SWITCH`, `AX_SR` und `AX_SPLIT_2`. Sie ist generisch aufgebaut und besitzt keine Hardware-Abhängigkeit, wodurch sie in verschiedenen Umgebungen eingesetzt werden kann.

Schnittstellenstruktur

Ereignis-Eingänge

- **IND** : Ereignis (Event) – Jeder Impuls an diesem Eingang löst einen Schaltvorgang aus.

Ereignis-Ausgänge

- Keine.

Daten-Eingänge

- Keine.

Daten-Ausgänge

- Keine.

Adapter

- **Q** : Ausgang (Typ `adapter::types::unidirectional::AX`) – Liefert den aktuellen Zustand des Flip-Flops (0 oder 1) als universelles Adaptersignal.

Funktionsweise

Der Baustein arbeitet als Toggle-Flip-Flop auf Ereignisbasis. Ein eingehendes Ereignis an `IND` wird an den internen Baustein `AX_E_SWITCH` weitergeleitet. Dieser vergleicht den aktuellen Zustand von `Q` (zurückgeführt über `AX_SPLIT_2.OUT2` an den Eingang `G`) und löst daraufhin entweder den Ausgang `EO0` oder `EO1` aus:

- Ist `Q` = 0 (Low), wird `EO0` aktiviert und an den Set-Eingang `S` des `AX_SR`-Flip-Flops gesendet.
- Ist `Q` = 1 (High), wird `EO1` aktiviert und an den Reset-Eingang `R` des `AX_SR`-Flip-Flops gesendet.

Der `AX_SR` (Set/Reset-Flip-Flop) reagiert auf die Ereignisse und setzt bzw. löscht seinen internen Zustand. Der Ausgang `Q` von `AX_SR` wird über den Splitter `AX_SPLIT_2` auf zwei Pfade verteilt: `OUT1` führt zum externen Ausgang `Q` der Subapp, `OUT2` wird als Rückkopplung an `AX_E_SWITCH` übergeben. Dadurch wird bei jedem neuen `IND`-Ereignis der Zustand deterministisch umgeschaltet.

Die Schaltung realisiert damit ein klassisches Toggle-Verhalten: Jeder Klick oder Impuls invertiert den logischen Pegel am Ausgang.

Technische Besonderheiten

- **Modularer Aufbau** : Die Funktionalität wird durch Kombination einfacher, wiederverwendbarer Bausteine erreicht.
- **Rückkopplung** : Über den Splitter wird der aktuelle Zustand intern zurückgeführt, um die Schaltrichtung korrekt zu bestimmen (Set vs. Reset).
- **Keine Hardwareabhängigkeit** : Die Subapp arbeitet rein ereignisbasiert und kann ohne Änderungen in verschiedenen Steuerungsumgebungen eingesetzt werden.
- **Einfache Schnittstelle** : Nur ein Ereigniseingang und ein Adapterausgang, daher sehr leicht integrierbar.

Zustandsübersicht

Zustand	Bedeutung	Reaktion auf `IND`
`Q = 0`	Ausgang Low	Set-Ereignis an `AX_SR` ⟶ Zustand wechselt zu `Q = 1`
`Q = 1`	Ausgang High	Reset-Ereignis an `AX_SR` ⟶ Zustand wechselt zu `Q = 0`

Das Flip-Flop besitzt zwei stabile Zustände, die sich bei jedem Ereignis am Eingang abwechseln.

Anwendungsszenarien

- **Taster-Umschaltung** : Ein einfacher Taster oder Schalter kann verwendet werden, um zwischen zwei Betriebszuständen (z. B. Ein/Aus, Hoch/Runter) umzuschalten.
- **Ereigniszähler** : Durch Zählen der Impulse kann der Ausgang als Halbzähler (Toggle) verwendet werden.
- **Zustandssteuerung** : Steuerung von binären Ausgängen in Automatisierungsprozessen, bei denen sich der Zustand bei jedem Impuls ändern soll.

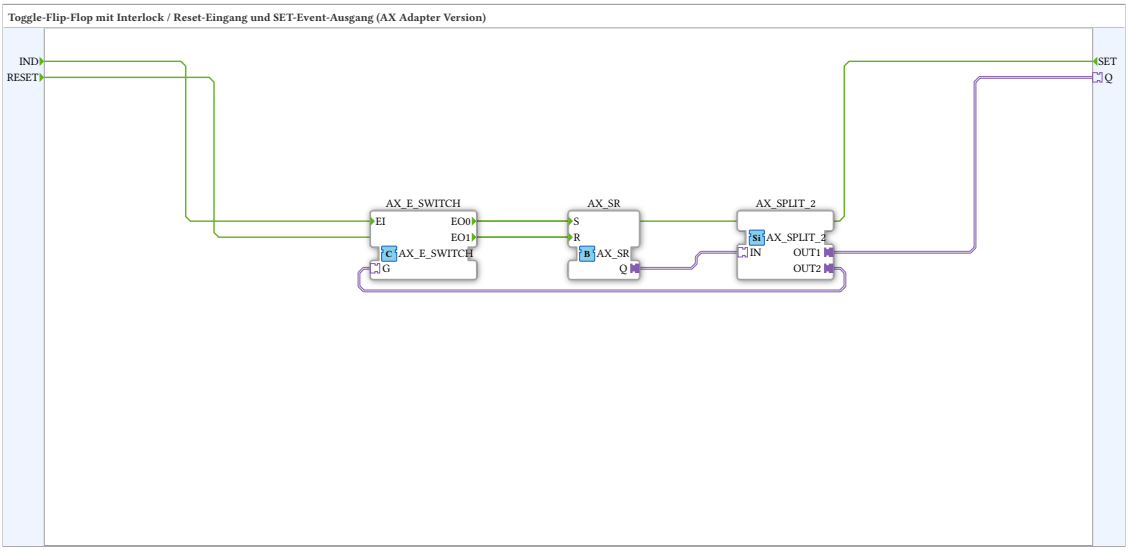
Vergleich mit ähnlichen Bausteinen

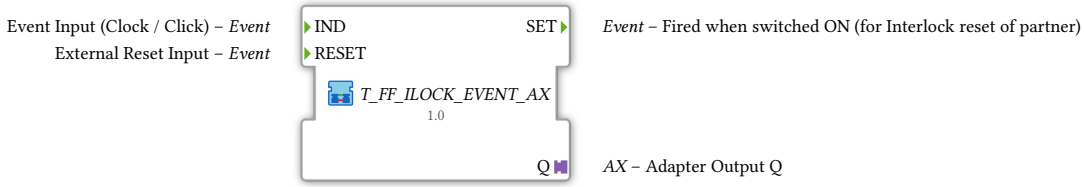
Es gibt in 4diac fertige Bausteine wie SR oder RS-Flip-Flops, die ebenfalls Set/Reset-Funktion bieten. Direkte Toggle-Bausteine sind jedoch oft nicht als Einzel-FB vorhanden. Der hier vorgestellte T_FF_EVENT_AX kombiniert vorhandene Elemente zu einem Toggle-Verhalten, ohne dass eine eigene zyklische Zustandslogik geschrieben werden muss. Gegenüber einem spezialisierten Toggle-FB bietet er den Vorteil, dass er vollständig aus standardisierten, wiederverwendbaren Komponenten aufgebaut ist und dadurch leichter angepasst oder erweitert werden kann.

Fazit

T_FF_EVENT_AX ist eine kompakte, modulare Lösung zur Realisierung eines Toggle-Flip-Flops in IEC 61499-Umgebungen. Der Aufbau aus bewährten Bausteinen gewährleistet Robustheit und Wiederverwendbarkeit. Durch die klare Schnittstelle (nur ein Ereigniseingang und ein Adapterausgang) ist die Integration in übergeordnete Steuerungssysteme unkompliziert. Besonders geeignet ist er für Anwendungen, in denen jeder Impuls den Zustand eines binären Ausgangs wechseln soll.

T_FF_ILOCK_EVENT_AX





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock T_FF_ILOCK_EVENT_AX ist eine Subapplikation gemäß IEC 61499 und realisiert ein Toggle-Flip-Flop (T-FF) mit zusätzlicher Interlock-Funktionalität und explizitem Reset-Eingang. Er besitzt einen ereignisgesteuerten Eingang IND (Takt), einen Reset-Eingang RESET und einen Ereignis-Ausgang SET. Der aktuelle Zustand wird über einen unidirektionalen Adapterausgang Q (Typ AX) ausgegeben. Der Baustein dient typischerweise zur Verriegelung zweier gegenläufiger Aktoren, z. B. Ventile oder Motoren, wobei ein Set-Ereignis verwendet wird, um den Gegenpart zu deaktivieren.

Schnittstellenstruktur

Ereignis-Eingänge

- **IND** (`Event`): Takt-/Click-Eingang. Bei jedem ansteigenden Ereignis wechselt der Ausgangszustand (Toggle).
- **RESET** (`Event`): Externer Reset-Eingang. Setzt den Ausgang zurück (Zustand „Aus“).

Ereignis-Ausgänge

- **SET** (`Event`): Wird ausgelöst, wenn der Ausgang auf „Ein“ geschaltet wird (Toggle von Aus auf Ein). Dieses Ereignis kann verwendet werden, um verriegelte Partnerfunktionsblöcke zurückzusetzen.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

- **Q** (Ausgang, Typ `adapter::types::unidirectional::AX`): Adapter zur Ausgabe des aktuellen Binärzustands („Ein/Aus“) als unidirektionales Signal.

Funktionsweise

Der Baustein verhält sich wie ein Toggle-Flip-Flop mit zusätzlichem Reset und Interlock-Funktion. Intern werden drei Adapter-basierte Bausteine verwendet: `AX_E_SWITCH`, `AX_SR` und `AX_SPLIT_2`.

- **AX_E_SWITCH** agiert als Umschalter (Toggle). Bei einem `IND`-Ereignis wechselt er seinen internen Schaltzustand und aktiviert abhängig vom vorherigen Zustand entweder den Ausgang `E00` oder `E01`.
- **AX_SR** ist ein Set/Reset-Flip-Flop. Wird `E00` aktiviert, erfolgt ein Set (Ausgang auf „Ein“). Wird `E01` aktiviert oder ein `RESET` empfangen, erfolgt ein Reset (Ausgang auf „Aus“). Das `SET`-Ereignis der Subapplikation wird parallel zum Set-Signal ausgegeben.
- **AX_SPLIT_2** teilt den Ausgang des SR-Flip-Flops in zwei identische Signale: eines wird über `Q` nach außen geführt, das andere als Rückkopplung `G` an den `AX_E_SWITCH` zurückgeführt. Diese Rückkopplung speichert den aktuellen Zustand, sodass bei einem nächsten `IND`-Ereignis der Toggle in die entgegengesetzte Richtung schalten kann.

Die Interlock-Logik besteht darin, dass bei einem Übergang von „Aus“ auf „Ein“ ein `SET`-Ereignis erzeugt wird, das typischerweise an einen verriegelten Partnerbaustein gesendet wird, um dessen Ausgang zurückzusetzen. Dadurch wird verhindert, dass zwei gegeneinander wirkende Aktoren gleichzeitig aktiv sind.

Technische Besonderheiten

- **Adapter-basierte Schnittstellen:** Der Zustand wird über einen `AX`-Adapter (unidirektional) ausgegeben, was eine einfache Verbindung mit anderen Adapter-kompatiblen Bausteinen ermöglicht.
- **Rückkopplungsschleife:** Die Rückführung des Ausgangs über `AX_SPLIT_2` an den `AX_E_SWITCH` stellt den Toggle-Mechanismus sicher, ohne dass zusätzliche Speicherbausteine in der Subapp benötigt werden.
- **Ereignisbasierter Toggle:** Der Toggle erfolgt ausschließlich über Ereignisse (`IND`), was eine schnelle, asynchrone Umschaltung in der verteilten Umgebung erlaubt.
- **Klare Interlock-Schnittstelle:** Der `SET`-Ausgang liefert ein dediziertes Ereignis, das zur Verriegelung externer Komponenten dient.

Zustandsübersicht

Der Baustein kennt zwei Zustände:

Zustand	Beschreibung
Aus (0)	Ausgang `Q` = false, keine aktive Ausgabe. Nach einem `RESET` oder nach einem Toggle aus dem Zustand „Ein“ hin zu „Aus“.
Ein (1)	Ausgang `Q` = true, aktive Ausgabe. Nach einem Toggle aus „Aus“ oder nach Set (implizit durch Toggle). Das `SET`-Ereignis wird beim Übergang von „Aus“ zu „Ein“ ausgelöst.

Der Zustandsübergang erfolgt bei jedem `IND`-Ereignis (toggle) oder bei `RESET` (setzt auf „Aus“). Während des Toggle wird der alte Zustand beibehalten und dann umgeschaltet.

Anwendungsszenarien

- **Verriegelung von Ventilen:** Zwei Ventile, die nicht gleichzeitig geöffnet sein dürfen. Ein T-FF steuert Ventil A, der `SET`-Ausgang aktiviert den Reset eines zweiten T-FF für Ventil B, sodass nur eines offen ist.
- **Handsteuerung mit Sicherheits-Reset:** Ein Bedienimpuls (Taster) schaltet eine Last ein/aus; ein übergeordnetes Sicherheitssystem kann durch `RESET` die Anlage immer in den sicheren Zustand „Aus“ bringen.
- **Motorsteuerung mit Richtungswechsel:** Wechselnde Richtungsanweisungen (vor/zurück) werden durch Toggle realisiert, wobei die Interlock verhindert, dass beide Richtungen gleichzeitig aktiv sind.

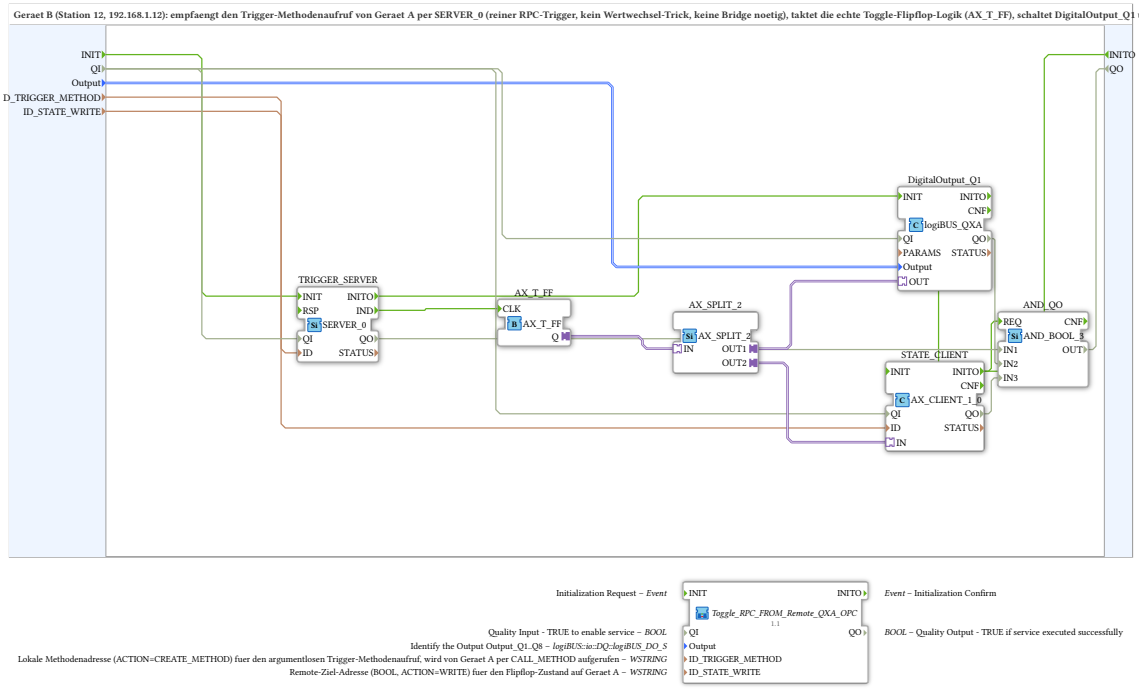
Vergleich mit ähnlichen Bausteinen

- **SR-Flipflop:** Ein simples SR-Flipflop benötigt separate Set- und Reset-Eingänge und besitzt kein Toggle-Verhalten. `T_FF_ILOCK_EVENT_AX` fügt den Toggle- und Interlock-Aspekt hinzu.
- **T-FF ohne Reset:** Ein einfaches Toggle-Flipflop hat keinen externen Reset und keine Interlock-Unterstützung. Dieser Baustein erweitert die Funktionalität um sichere Reset- und Verriegelungsmechanismen.
- **Standard-FB `SR`-Baustein aus IEC 61499:** Solche Bausteine sind meist nicht adapterbasiert und bieten keine automatische Rückkopplung für Toggle-Betrieb. Die Adapter-Schnittstelle von `T_FF_ILOCK_EVENT_AX` erleichtert die Einbindung in moderne Adapter-basierte Architekturen.

Fazit

`T_FF_ILOCK_EVENT_AX` ist ein vielseitiger Baustein für verriegelte Umschalt- und Toggle-Anwendungen. Er kombiniert ein Toggle-Flip-Flop mit einem Reset-Eingang und einem speziellen Interlock-Event-Ausgang, der in Automatisierungssystemen häufig benötigt wird. Die Verwendung von Adaptern und die interne Struktur aus spezialisierten AX-Bausteinen machen ihn robust und flexibel in verteilten Steuerungsumgebungen einsetzbar. Durch die klare Trennung von Ereignis- und Signalkommunikation eignet er sich besonders für sicherheitsrelevante Steuerungen.

Toggle_RPC_FROM_Remote_QXA_OPC



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein `Toggle_RPC_FROM_Remote_QXA_OPC` ist eine Subapplikation, die auf einem Gerät B (z.B. Station 12 mit IP-Adresse 192.168.1.12) eingesetzt wird. Sie empfängt über einen OPC-UA-Server

(SERVER_0) einen entfernten Trigger-Methodenaufruf von Gerät A. Der Trigger dient als Takt für eine echte Flip-Flop-Logik (AX_T_FF), die den Zustand eines Digitalausgangs (DigitalOutput_Q1) umschaltet. Der neue Zustand wird anschließend aktiv über einen OPC-UA-Client (STATE_CLIENT) zurück an Gerät A geschrieben. Die Kommunikation erfolgt im „SUB-style“: Das gesamte Protokoll ist in einem Composite (Bibliothek MyLib::sys) gekapselt und benötigt keine zusätzliche Logik in der Ressource des Geräts.

Schnittstellenstruktur

Ereignis-Eingänge

- INIT (EInit): Initialization Request

Ereignis-Ausgänge

- INITO (EInit): Initialization Confirm

Daten-Eingänge

- QI (Typ: BOOL): Quality Input - TRUE, um den Dienst freizugeben. Initialwert: TRUE.
- Output (Typ: logiBUS::io::DQ::logiBUS_DO_S): Identifiziert den digitalen Ausgang (Q1–Q8) des angeschlossenen logiBUS-Moduls. Initialwert: logiBUS_DO::Invalid.
- ID_TRIGGER_METHOD (Typ: WSTRING): Lokale Methodenadresse (ACTION=CREATE_METHOD) für den argumentlosen Trigger-Aufruf, der von Gerät A per CALL_METHOD ausgeführt wird.
- ID_STATE_WRITE (Typ: WSTRING): Remote-Zieladresse (BOOL, ACTION=WRITE) für das Zurückschreiben des Flip-Flop-Zustands an Gerät A.

Daten-Ausgänge

- QO (Typ: BOOL): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde (UND von TRIGGER_SERVER.QO, DigitalOutput_Q1.QO und STATE_CLIENT.QO).

Adapter

Keine (die Subapplikation besitzt keine externen Adapter; die interne Kommunikation erfolgt über die oben genannten Daten-Eingänge und die internen Funktionsblöcke).

Funktionsweise

Die Subapplikation implementiert eine gesteuerte Toggle-Funktion für einen Digitalausgang mit Rückmeldung. Der Ablauf ist wie folgt:

1. **Trigger-Empfang**: Der OPC-UA-Server (TRIGGER_SERVER, Typ iec61499::net::SERVER_0) wartet auf einen eingehenden Methodenaufruf. Die Methode ist durch den Parameter ID_TRIGGER_METHOD definiert.
2. **Taktung des Flip-Flops**: Das eingehende Ereignis (IND-Event) des Servers wird direkt auf den Takteingang (CLK) des Toggle-Flip-Flops (AX_T_FF) geschaltet. Gleichzeitig wird das IND-Event sofort als Bestätigung (RSP) an den Server zurückgegeben, um eine Blockierung des OPC-UA-Threads zu vermeiden.
3. **Zustandsänderung**: Das Flip-Flop wechselt bei jedem Takt seinen binären Zustand ($0 \leftrightarrow 1$). Der Ausgang Q des Flip-Flops wird an einen Splitter (AX_SPLIT_2) übergeben, der das Signal auf zwei parallele Pfade aufteilt.
4. **Ausgangssignal**: Über den ersten Ausgang (OUT1) wird das Signal an den Funktionsblock DigitalOutput_Q1 (Typ logiBUS::io::DQ::logiBUS_QXA) geleitet, der den entsprechenden physischen Ausgang (identifiziert durch den Eingang Output) steuert.
5. **Rückmeldung**: Über den zweiten Ausgang (OUT2) wird das Signal an den OPC-UA-Client (STATE_CLIENT, Typ adapter::net::AX_CLIENT_1_0) gesendet. Dieser Client schreibt den aktuellen Zustand (als BOOL) an die Adresse ID_STATE_WRITE auf Gerät A.

Die Daten-Eingänge der Subapplikation werden direkt an die entsprechenden Funktionsblöcke verdrahtet:

- Output ⟶ DigitalOutput_Q1.Output
- ID_TRIGGER_METHOD ⟶ TRIGGER_SERVER.ID
- ID_STATE_WRITE ⟶ STATE_CLIENT.ID

Initialisierung und Status:

- `INIT` ⟶ `TRIGGER_SERVER.INIT`, `TRIGGER_SERVER.INITO` ⟶ `DigitalOutput_Q1.INIT`, `DigitalOutput_Q1.INITO` ⟶ `STATE_CLIENT.INIT`, `STATE_CLIENT.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`
- `QI` ⟶ `TRIGGER_SERVER.QI`, `DigitalOutput_Q1.QI` und `STATE_CLIENT.QI`
- `TRIGGER_SERVER.QO` ⟶ `AND_QO.IN1`, `DigitalOutput_Q1.QO` ⟶ `AND_QO.IN2`, `STATE_CLIENT.QO` ⟶ `AND_QO.IN3` (Baustein `AND_QO`, Typ `iec61131::booleanOperators::AND_BOOL_3`); `AND_QO.OUT` ⟶ `QO`

Technische Besonderheiten

- **RPC-Trigger ohne Wertwechsel-Trick**: Im Gegensatz zu anderen Lösungen wird hier ein echtes Methodenaufruf-Trigger verwendet, was die Notwendigkeit einer Bridge oder eines Wertwechsel-Kunstgriffs eliminiert.
- **Direkte RSP-Verdrahtung**: Das `IND`-Ereignis des Servers wird unmittelbar an `RSP` zurückgeführt. Dadurch wird verhindert, dass der OPC-UA-Server-Thread (open62541) bis zu 4 s blockiert, was ansonsten zu Verzögerungen aller OPC-UA-Statusmeldungen im gesamten Projekt geführt hätte.
- **Kapselung im Composite**: Das Kommunikationsprotokoll liegt vollständig in der Subapplikation (`MyLib::sys`), sodass die Ressource des Geräts keine zusätzliche Protokolllogik enthalten muss.

Zustandsübersicht

Die Subapplikation selbst besitzt keinen expliziten Zustandsautomaten. Der interne Funktionsblock `AX_T_FF` realisiert ein Toggle-Flip-Flop mit zwei stabilen Zuständen:

- **Zustand 0**: Ausgang `Q` = FALSE ⟶ Digitalausgang aus, Rückmeldung FALSE.
- **Zustand 1**: Ausgang `Q` = TRUE ⟶ Digitalausgang ein, Rückmeldung TRUE.

Jedes eingehende Trigger-Ereignis bewirkt einen Zustandswechsel (Toggle). Der Zustand bleibt bis zum nächsten Trigger erhalten.

Anwendungsszenarien

- **Fernsteuerung eines Digitalausgangs**: Gerät A kann über eine OPC-UA-Methode den Ausgang Q1 eines entfernten logiBUS-Moduls (angeschlossen an Gerät B) umschalten. Der aktuelle Zustand wird nach jedem Umschalten an Gerät A zurückgemeldet.
- **Synchronisation von Betriebszuständen**: Die Subapplikation eignet sich für verteilte Systeme, in denen ein zentrales Steuergerät (Gerät A) den Status eines peripheren Geräts (Gerät B) überwacht und ändert.
- **Redundante Zustandsüberwachung**: Durch die aktive Rückmeldung des Flip-Flop-Zustands ist der Zustand des Ausgangs jederzeit im übergeordneten System bekannt, auch wenn die ursprünglichen Trigger verloren gehen.

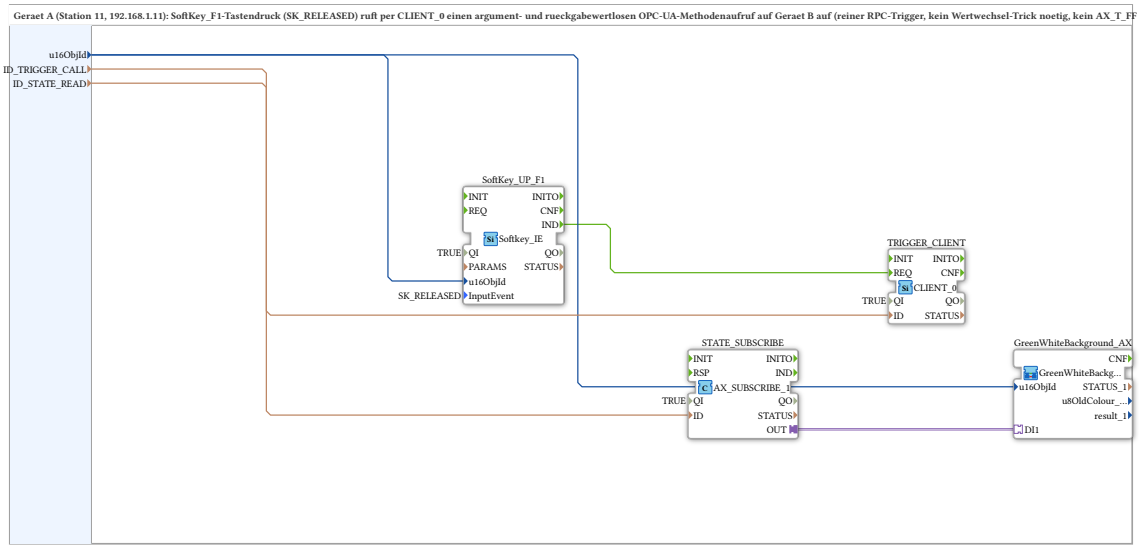
Vergleich mit ähnlichen Bausteinen

- **Ohne RPC, mit Wertwechsel-Trick**: Bei älteren Implementierungen wurde ein einfacher Dateneingang als Trigger verwendet, was zu unerwünschten Nebenwirkungen bei der Wertinterpretation führen konnte. Der vorliegende Baustein verwendet dagegen einen expliziten Methodenaufruf.
- **Mit Bridge**: Andere Lösungen benötigen eine Bridge-Komponente, um OPC-UA- mit anderen Protokollen zu verbinden. Hier entfällt dies, da das gesamte Handling innerhalb des Composites gekapselt ist.
- **Mit separatem Toggle-FB**: Der Einsatz des vorgefertigten `AX_T_FF` sorgt für eine klare und wartbare Flip-Flop-Logik im Vergleich zu frei verdrahteten SR-Latches.

Fazit

`Toggle_RPC_FROM_Remote_QXA_OPC` stellt eine robuste und effiziente Lösung für ferngesteuerte Toggle-Schaltungen mit Rückmeldung dar. Die Kombination aus RPC-Trigger, getakteten Flip-Flop und aktiver Zustandsrückmeldung macht den Baustein ideal für moderne industrielle Anwendungen, bei denen OPC-UA als Kommunikationsstandard verwendet wird. Die direkte RSP-Verdrahtung verbessert zudem die Reaktionszeit des Gesamtsystems erheblich. Durch die Kapselung in einem Composite bleibt die Ressource des Zielgeräts schlank und die Wiederverwendbarkeit hoch.

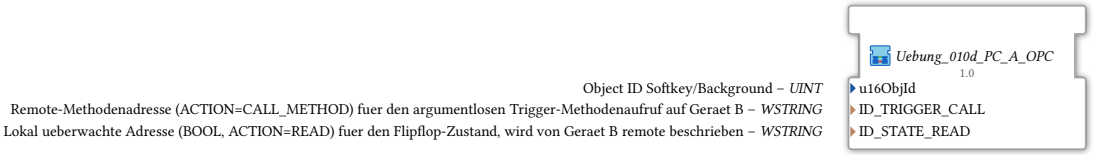
Uebung_010d_PC_A_OPC



• – ○ • – ○ • – ○ •

Einleitung

`Uebung_010d_PC_A_OPC` ist die Geraet-A-Seite (Station 11, 192.168.1.11) der PC-zu-PC-OPC-UA-Variante von Uebung 010d (Toggle-Flipflop via SoftKey): ein SoftKey-Tastendruck ruft per `CLIENT_0` einen argument- und rueckgabewertlosen OPC-UA-Methodenaufruf auf Geraet B auf (reiner RPC-Trigger, kein Wertwechsel-Trick noetig, keine Toggle-Logik auf diesem Geraet). `GreenWhiteBackground1_AX` zeigt den von Geraet B lokal ueberwachten Flipflop-Zustand. „SUB style“: das Protokoll steckt im `MyLib::sys`-Composite, nicht in der Resource des Geraets - Gegenstueck: `Uebung_010d_PC_B_OPC`.



Verwendete Funktionsbausteine (FBs)

- **SoftKey_UP_F1** (`isobus::UT::io::Softkey::Softkey_IE`): physischer SoftKey (F1), `InputEvent=SK_RELEASED`.
- **TRIGGER_CLIENT** (`iec61499::net::CLIENT_0`): ruft die argumentlose Remote-Methode unter `ID_TRIGGER_CALL` auf.
- **STATE_SUBSCRIBE** (`adapter::net::AX_SUBSCRIBE_1`): abonniert den von Geraet B geschriebenen Flipflop-Zustand unter `ID_STATE_READ`.
- **GreenWhiteBackground_AX** (SubApp, Typ `MyLib::sys::GreenWhiteBackground1_AX`): zeigt den abonnierten Zustand am SoftKey an.

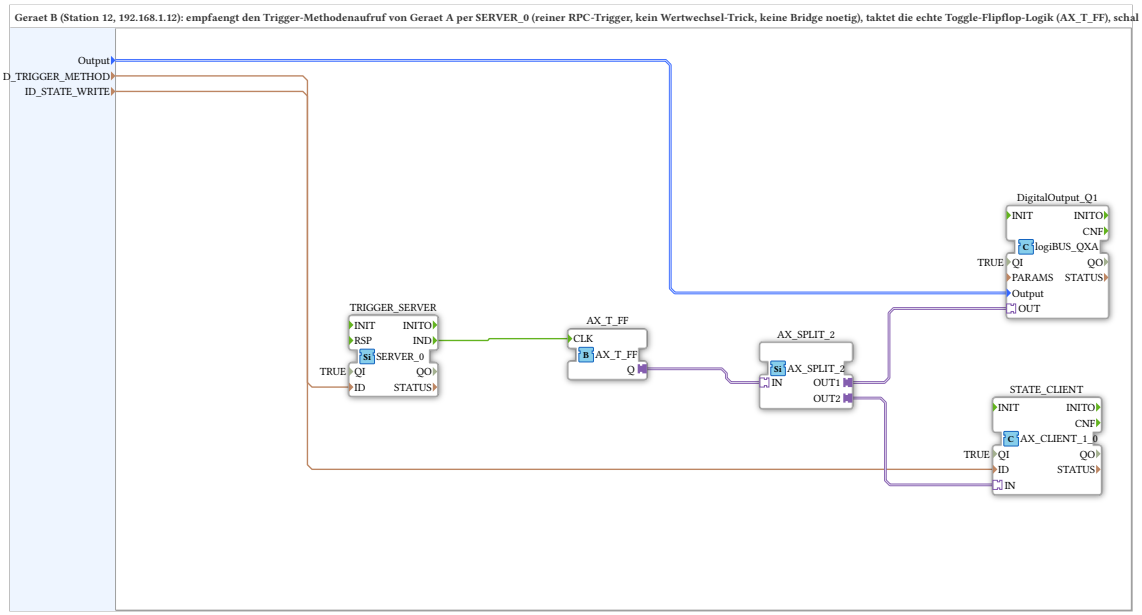
Zusammenfassung

Geraet-A-Seite eines PC-zu-PC-Toggle-Triggers: SoftKey ruft eine Remote-Methode auf, der tatsaechliche Flipflop-Zustand kommt zurueckgemeldet von Geraet B.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

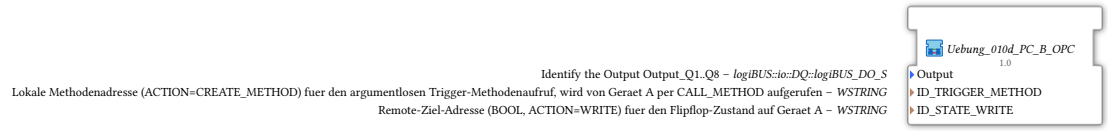
Uebung_010d_PC_B_OPC



• – ○ • – ○ • – ○ •

Einleitung

`Uebung_010d_PC_B_OPC` ist die Geraet-B-Seite (Station 12, 192.168.1.12) der PC-zu-PC-OPC-UA-Variante von Uebung 010d: empfaengt den Trigger-Methodenaufruf von Geraet A per `SERVER_0` (reiner RPC-Trigger, kein Wertwechsel-Trick, keine Bridge noetig), taktet die echte Toggle-Flipflop-Logik (`AX_T_FF`), schaltet `DigitalOutput_Q1` und schreibt den neuen Zustand aktiv per `AX_CLIENT_1_0` zurueck auf Geraet A. Gegenstueck: `Uebung_010d_PC_A_OPC`.



Verwendete Funktionsbausteine (FBs)

- **TRIGGER_SERVER** (`iec61499::net::SERVER_0`): empfaengt den Methodenaufruf unter `ID_TRIGGER_METHOD`.
- **AX_T_FF** (`adapter::events::unidirectional::AX_T_FF`): Toggle-Flipflop, getaktet von `TRIGGER_SERVER.IND`.
- **AX_SPLIT_2** (`adapter::events::unidirectional::AX_SPLIT_2`): verzweigt den Flipflop-Zustand auf physischen Ausgang + Rueckmeldung.
- **DigitalOutput_Q1** (`logiBUS::io::DQ::logiBUS_QXA`): physischer Ausgang (`Output` waehlt `Output_Q1..Q8`).
- **STATE_CLIENT** (`adapter::net::AX_CLIENT_1_0`): schreibt den neuen Zustand aktiv auf Geraet A unter `ID_STATE_WRITE`.

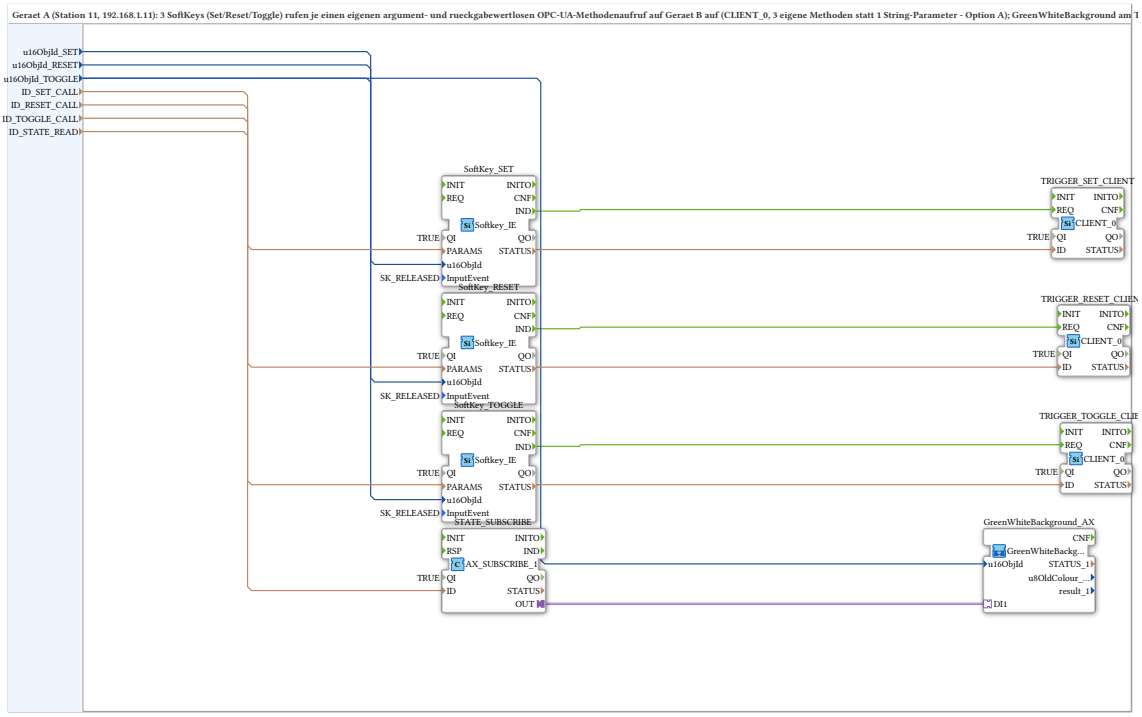
Zusammenfassung

Geraet-B-Seite: empfaengt den Trigger per RPC, haelt die tatsaechliche Toggle-Logik und meldet den Zustand aktiv an Geraet A zurueck.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

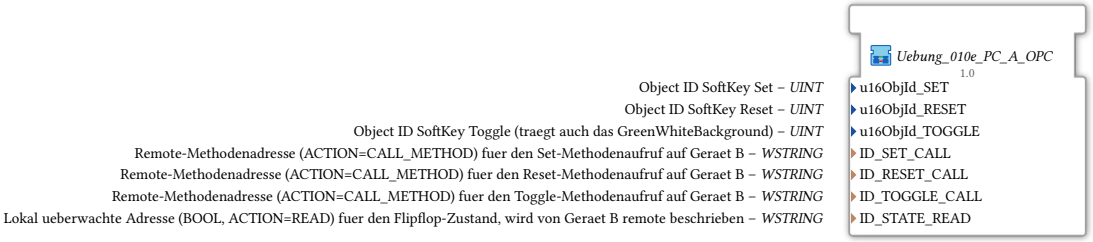
Uebung_010e_PC_A_OPC



• – ○ • – ○ • – ○ •

Einleitung

`Uebung_010e_PC_A_OPC` ist die Geraet-A-Seite (Station 11, 192.168.1.11) der PC-zu-PC-OPC-UA-Variante von Uebung 010e (SR+Toggle-Flipflop via 3 SoftKeys): 3 SoftKeys (Set/Reset/Toggle) rufen je einen eigenen argument- und rueckgabewertlosen OPC-UA-Methodenaufruf auf Geraet B auf (`CLIENT_0`, 3 eigene Methoden statt 1 String-Parameter - Option A). `GreenWhiteBackground1_AX` am Toggle-SoftKey zeigt den von Geraet B lokal ueberwachten Flipflop-Zustand. Gegenstueck: `Uebung_010e_PC_B_OPC`.



Verwendete Funktionsbausteine (FBs)

- **SoftKey_SET / SoftKey_RESET / SoftKey_TOGGLE** (`isobus::UT::io::Softkey::Softkey_IE`): die 3 physischen SoftKeys.
- **TRIGGER_SET_CLIENT / TRIGGER_RESET_CLIENT / TRIGGER_TOGGLE_CLIENT** (`iec61499::net::CLIENT_0`): je eine eigene Remote-Methode (`ID_SET_CALL`/`ID_RESET_CALL`/`ID_TOGGLE_CALL`).
- **STATE_SUBSCRIBE** (`adapter::net::AX_SUBSCRIBE_1`): abonniert den Flipflop-Zustand unter `ID_STATE_READ`.
- **GreenWhiteBackground_AX** (SubApp, Typ `MyLib::sys::GreenWhiteBackground1_AX`): zeigt den Zustand am Toggle-SoftKey an.

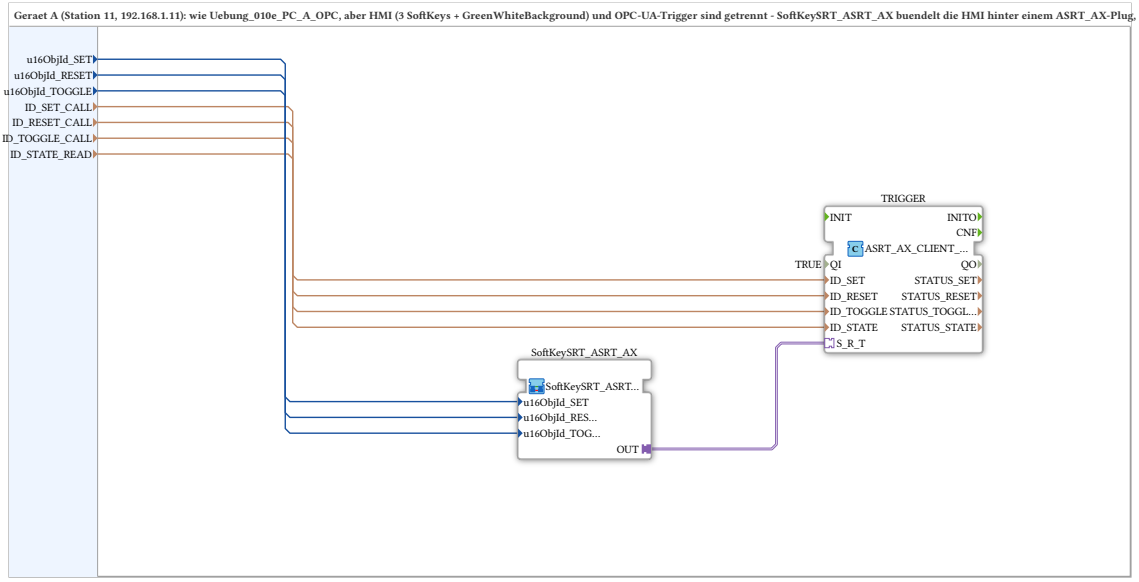
Zusammenfassung

Geraet-A-Seite eines Set/Reset/Toggle-RPC-Musters mit 3 eigenen Methodenaufrufen und Zustandsrueckmeldung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

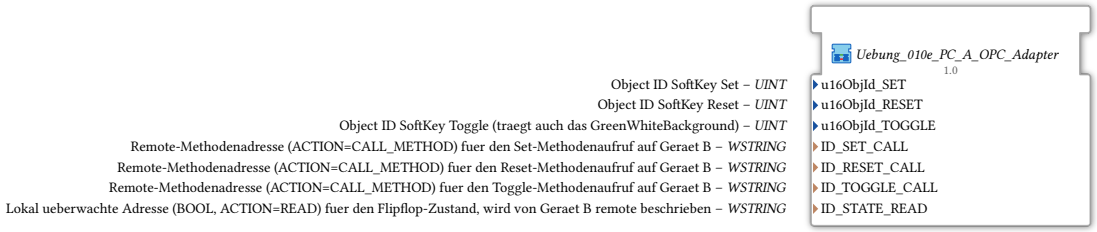
Uebung_010e_PC_A_OPC_Adapter



• — ○ • — ○ • — ○ •

Einleitung

`Uebung_010e_PC_A_OPC_Adapter` ist die adapter-gebuendelte Variante von `Uebung_010e_PC_A_OPC` (Geraet A, Station 11): HMI (3 SoftKeys + `GreenWhiteBackground`) und OPC-UA-Trigger sind getrennt. `SoftKeySRT_ASRT_AX` buendelt die HMI hinter einem `ASRT_AX`-Plug; `ASRT_AX_CLIENT_0_SUBSCRIBE_1` buendelt die 3 `CLIENT_0`-Instanzen + `AX_SUBSCRIBE_1` hinter einem `ASRT_AX`-Socket. Gegenstueck: `Uebung_010e_PC_B_OPC_Adapter`.



Verwendete Funktionsbausteine (FBs)

- **SoftKeySRT_ASRT_AX** (SubApp, Typ `MyLib::sys::SoftKeySRT_ASRT_AX`): 3 SoftKeys + Statusanzeige, gebuendelt hinter einem `ASRT_AX`-Plug.
- **TRIGGER** (`adapter::net::ASRT_AX_CLIENT_0_SUBSCRIBE_1`): buendelt 3 Methodenaufrufe (`ID_SET_CALL`/`ID_RESET_CALL`/`ID_TOGGLE_CALL`) und Zustands-Abo (`ID_STATE_READ`) hinter einem einzigen `ASRT_AX`-Socket.

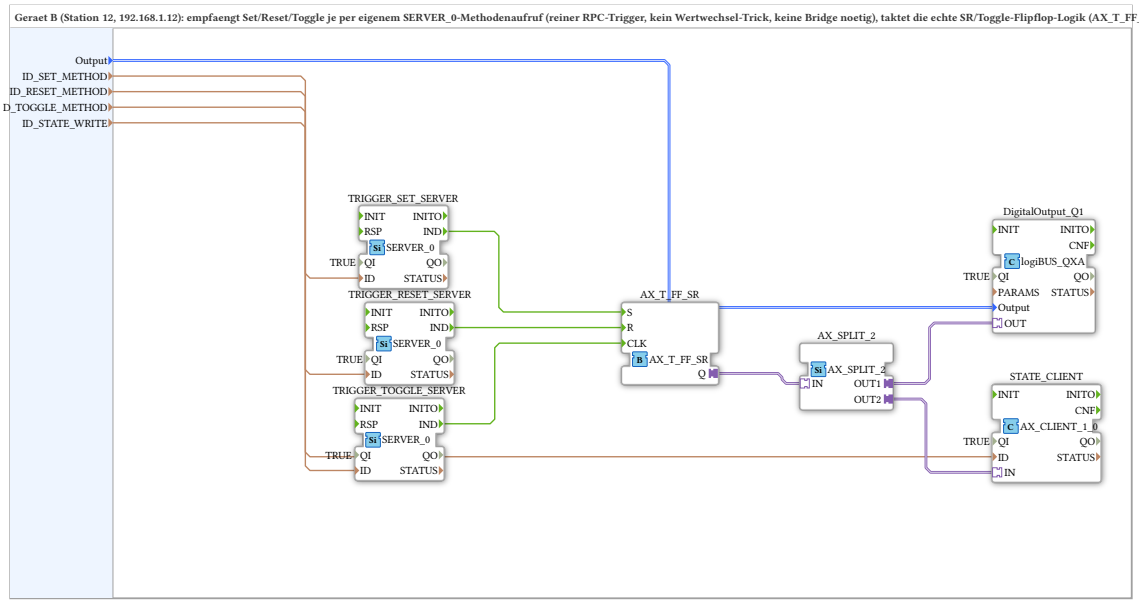
Zusammenfassung

Adapter-gebuendelte Variante von `Uebung_010e_PC_A_OPC`: HMI und Netzwerkprotokoll sauber getrennt, wiederverwendbar ueber `SoftKeySRT_ASRT_AX`.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

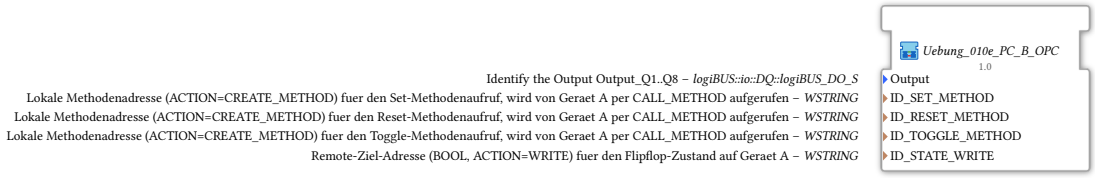
Uebung_010e_PC_B_OPC



• – ○ • – ○ • – ○ •

Einleitung

`Uebung_010e_PC_B_OPC` ist die Geraet-B-Seite (Station 12, 192.168.1.12): empfaengt Set/Reset/Toggle je per eigenem `SERVER_0`-Methodenaufruf (reiner RPC-Trigger, kein Wertwechsel-Trick, keine Bridge noetig), taktet die echte SR/Toggle-Flipflop-Logik (`AX_T_FF_SR`), schaltet `DigitalOutput_Q1` und schreibt den neuen Zustand aktiv per `AX_CLIENT_1_0` zurueck auf Geraet A. Gegenstueck: `Uebung_010e_PC_A_OPC`.



Verwendete Funktionsbausteine (FBs)

- **TRIGGER_SET_SERVER / TRIGGER_RESET_SERVER / TRIGGER_TOGGLE_SERVER** (`iec61499::net::SERVER_0`): empfangen je einen eigenen Methodenaufruf.
- **AX_T_FF_SR** (`adapter::events::unidirectional::AX_T_FF_SR`): kombinierte Set/Reset/Toggle-Flipflop-Logik.
- **AX_SPLIT_2** (`adapter::events::unidirectional::AX_SPLIT_2`): verzweigt den Zustand auf physischen Ausgang + Rueckmeldung.
- **DigitalOutput_Q1** (`logiBUS::io::DQ::logiBUS_QXA`): physischer Ausgang.
- **STATE_CLIENT** (`adapter::net::AX_CLIENT_1_0`): schreibt den Zustand aktiv auf Geraet A unter `ID_STATE_WRITE`.

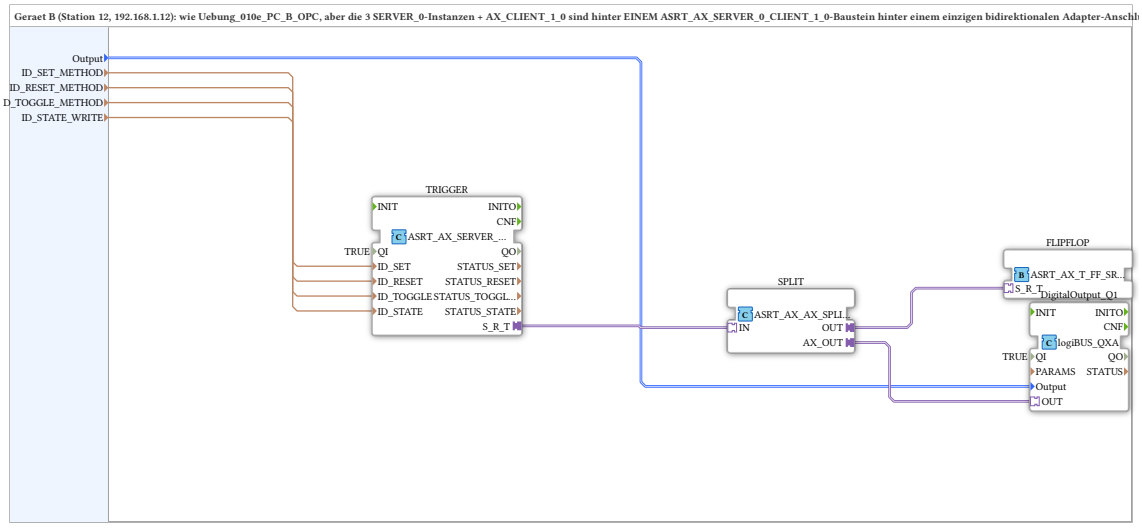
Zusammenfassung

Geraet-B-Seite: empfaengt Set/Reset/Toggle per RPC, haelt die kombinierte SR/Toggle-Flipflop-Logik und meldet den Zustand aktiv zurueck.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

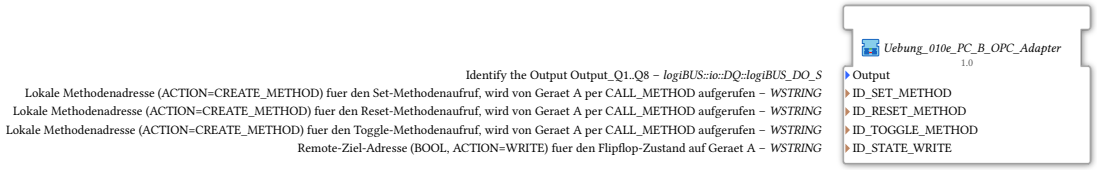
Uebung_010e_PC_B_OPC_Adapter



• – ○ • – ○ • – ○ •

Einleitung

`Uebung_010e_PC_B_OPC_Adapter` ist die adapter-gebuendelte Variante von `Uebung_010e_PC_B_OPC` (Geraet B, Station 12): die 3 `SERVER_0`-Instanzen + `AX_CLIENT_1_0` sind hinter EINEM `ASRT_AX_SERVER_0_CLIENT_1_0`-Baustein hinter einem einzigen bidirektionalen Adapter-Anschluss gebuendelt; `ASRT_AX_AX_SPLIT` speist sowohl `DigitalOutput_Q1` als auch die SR/Toggle-Flipflop-Logik (`ASRT_AX_T_FF_SR_2`). Gegenstueck: `Uebung_010e_PC_A_OPC_Adapter`.



Verwendete Funktionsbausteine (FBs)

- **TRIGGER** (`adapter::net::ASRT_AX_SERVER_0_CLIENT_1_0`): buendelt Server-Empfang (`ID_SET_METHOD`/`ID_RESET_METHOD`/`ID_TOGGLE_METHOD`) und Zustands-Rueckmeldung (`ID_STATE_WRITE`).
- **SPLIT** (`adapter::events::bidirectional::ASRT_AX_AX_SPLIT`): verzweigt auf physischen Ausgang + Flipflop-Logik.
- **FLIPFLOP** (`adapter::events::bidirectional::ASRT_AX_T_FF_SR_2`): kombinierte SR/Toggle-Flipflop-Logik als Adapter-Baustein.
- **DigitalOutput_Q1** (`logiBUS::io::DQ::logiBUS_QXA`): physischer Ausgang.

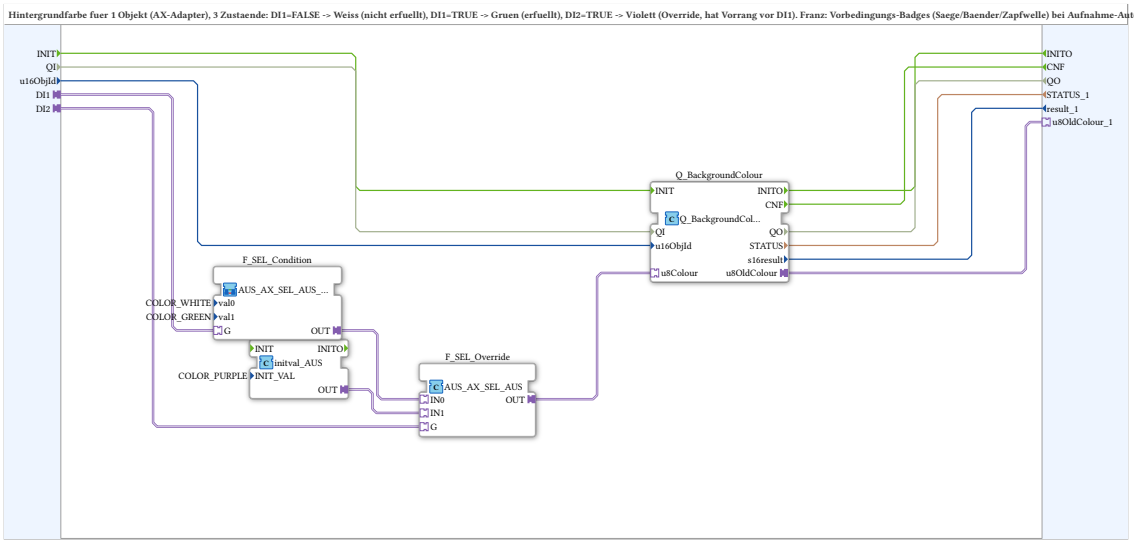
Zusammenfassung

Adapter-gebuendelte Variante von `Uebung_010e_PC_B_OPC`: Netzwerkprotokoll und SR/Toggle-Flipflop-Logik als wiederverwendbare Adapter-Bausteine.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

WhiteGreenPurpleBackground1_AX



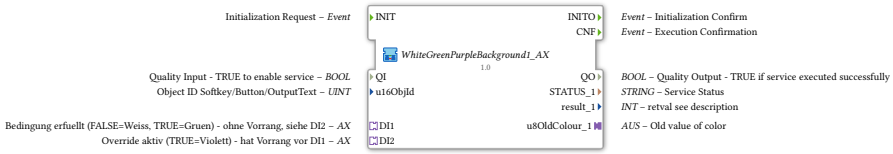
• – ○ • – ○ • – ○ •

Einleitung

`WhiteGreenPurpleBackground1_AX` setzt die Hintergrundfarbe für **ein Objekt** (AX-Adapter) mit drei Zuständen:

- `DI1=FALSE` ⟶ **Weiß** (nicht erfüllt)
- `DI1=TRUE` ⟶ **Grün** (erfüllt)
- `DI2=TRUE` ⟶ **Violett** (Override, hat Vorrang vor `DI1`)

Verwendet wird er z. B. für Vorbedingungs-Badges (Säge/Bänder/Zapfwelle) bei der Aufnahme-Automatik. Der Baustein ist analog zu `GreenWhiteBackground1_AX` aufgebaut.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: WhiteGreenPurpleBackground1_AX

- **Typ**: SubAppType
- **Verwendete interne Bausteine**:
  - **F_SEL_Condition**: SubApp `MyLib::sys::AUS_AX_SEL_AUS_2_VAL` mit `val0=COLOR_WHITE`, `val1=COLOR_GREEN` – wählt anhand von `DI1` Weiß oder Grün
  - **F_SEL_Override**: `adapter::iec61131::selection::AUS_AX_SEL_AUS` – wählt anhand von `DI2` zwischen dem Ergebnis der Bedingung (`IN0`) und Violett (`IN1`)
  - **initval_AUS**: `adapter::types::unidirectional::AUS::initval::initval_AUS` mit `INIT_VAL=COLOR_PURPLE`
  - **Q_BackgroundColour**: `isobus::UT::Q::Q_BackgroundColour_AUS` – schreibt die Farbe auf das VT-Objekt `u16ObjId`
- **Importe**: `ID_NULL`, `COLOR_GREEN`, `COLOR_WHITE`, `COLOR_PURPLE`

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung (⟶ `Q_BackgroundColour.INIT`)

Weitere Ereignisse kommen über die Adapter `DI1` und `DI2`.

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung (von `Q_BackgroundColour.INITO`)
- **CNF**: Ausführungsbestätigung (von `Q_BackgroundColour.CNF`)

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input – `TRUE` aktiviert den Dienst (⟶ `Q_BackgroundColour.QI`)
- **u16ObjId** (`UINT`, Vorgabe `ID_NULL`): Object-ID Softkey/Button/OutputText

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde (von `Q_BackgroundColour.QO`)
- **STATUS_1** (`STRING`): Dienststatus (von `Q_BackgroundColour.STATUS`)
- **result_1** (`INT`): Rückgabewert (von `Q_BackgroundColour.s16result`)

Adapter

- **DI1** (Socket, `adapter::types::unidirectional::AX`): Bedingung erfüllt (FALSE = Weiß, TRUE = Grün), ohne Vorrang, siehe `DI2`
- **DI2** (Socket, `adapter::types::unidirectional::AX`): Override aktiv (TRUE = Violett), hat Vorrang vor `DI1`
- **u8OldColour_1** (Plug, `adapter::types::unidirectional::AUS`): Alter Farbwert

Programmablauf und Verbindungen

- `DI1` ⟶ `F_SEL_Condition.G`
- `DI2` ⟶ `F_SEL_Override.G`
- `F_SEL_Condition.OUT` ⟶ `F_SEL_Override.IN0`; `initval_AUS.OUT` ⟶ `F_SEL_Override.IN1`
- `F_SEL_Override.OUT` ⟶ `Q_BackgroundColour.u8Colour`
- `u16ObjId` ⟶ `Q_BackgroundColour.u16ObjId`
- `INIT` ⟶ `Q_BackgroundColour.INIT`; `Q_BackgroundColour.INITO` ⟶ `INITO`
- `QI` ⟶ `Q_BackgroundColour.QI`; `Q_BackgroundColour.QO` ⟶ `QO`
- `Q_BackgroundColour.u8OldColour` ⟶ `u8OldColour_1`; `.STATUS` ⟶ `STATUS_1`; `.s16result` ⟶ `result_1`; `.CNF` ⟶ `CNF`

Technische Besonderheiten

- Zwei verschachtelte Selektoren (nicht ein einzelner `AX_SEL`), da ein einzelner `AX_SEL` nur bei Änderung seines **eigenen** `G` neu auswertet und die zweite Stufe bei einer Änderung von `DI1` sonst nicht nachziehen würde
- Beide Selektoren werden über einen gemeinsamen Trigger (`E_MERGE_2` aus `DI1.E1`/`DI2.E1`) in Reihe neu ausgewertet, damit eine Änderung von `DI1` **oder** `DI2` immer die komplette, korrekte Farbe liefert
- Version 1.0 (2026-10-01)

Zustandsübersicht

DI1	DI2	Farbe
FALSE	FALSE	Weiß (`COLOR_WHITE`)
TRUE	FALSE	Grün (`COLOR_GREEN`)
beliebig	TRUE	Violett (`COLOR_PURPLE`)

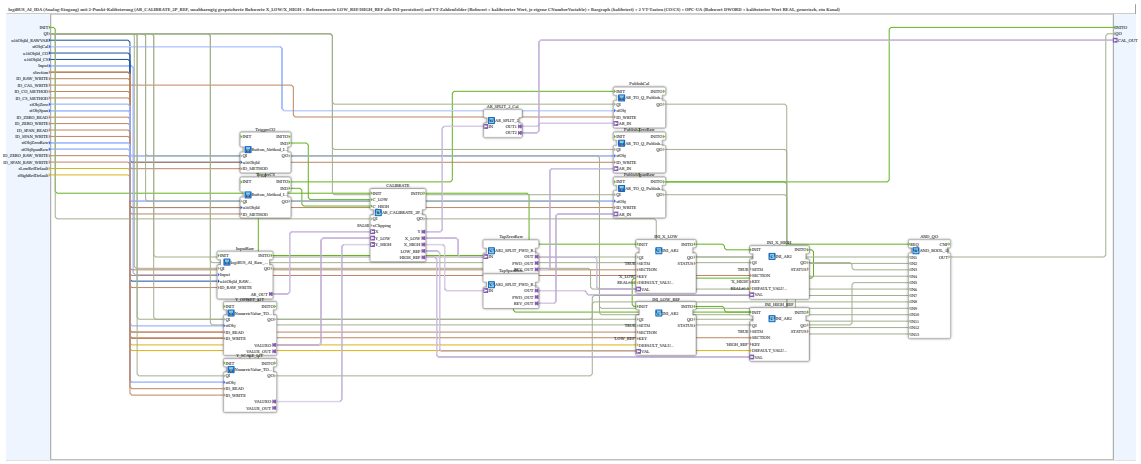
Anwendungsszenarien

- Vorbedingungs-Badges (Säge, Bänder, Zapfwelle) einer Aufnahme-Automatik mit Override-Anzeige

Zusammenfassung

`WhiteGreenPurpleBackground1_AX` färbt ein VT-Objekt weiß, grün oder violett, wobei der Override `DI2` Vorrang vor der Bedingung `DI1` hat.

logiBUS_AI_Calibrate_2P_IDA_OPC



• – ○ • – ○ • – ○ •

Einleitung

`logiBUS_AI_Calibrate_2P_IDA_OPC` bindet einen physischen Analogeingang (`logiBUS_AI_IDA`) an eine vollständige VT- und OPC-UA-gestützte 2-Punkt-Kalibrierung (`AR_CALIBRATE_2P_REF`) an — genutzt vom AI_Calibrate_2P-Trainingsbeispiel. Der Rohwert des Eingangs wird über die 2-Punkt-Kalibrierkette in einen physikalisch skalierten Wert gewandelt; Nullpunkt (`Zero`) und Spanne (`Span`) sowie deren gemessene Rohwerte (`ZeroRaw`/`SpanRaw`) sind sowohl per VT als auch per OPC-UA einstell- bzw. ablesbar und werden per INI-Datei persistiert. Bei Teilkalibrierung (nur ein Punkt kalibriert) greift sofort eine saubere Ausweichlösung auf den verfügbaren Referenzwert.

Verwendete Funktionsbausteine (FBs)

- **logiBUS_AI_IDA** (`logiBUS::io::AI::logiBUS_AI_IDA`): physischer Analogeingang, liefert den Rohwert als Adapter.
- **AR_CALIBRATE_2P_REF** (Adapter-Composite): 2-Punkt-Kalibrierung (Nullpunkt/Spanne) mit Rohwert-Diagnose (`ZeroRaw`/`SpanRaw`), Teilkalibrierungs-Ausweichlogik und konfigurierbarem Clipping (`xClipping`).
- **VT- und OPC-UA-Brücken** (analog zu `NumericValue_TO_AR2_OPC`/`OPC_TO_AR2` und `INI_IN_AND_STORE_AR2`): stellen `Zero`/`Span` und `ZeroRaw`/`SpanRaw` sowohl über VT-Eingabefelder/Anzeigen als auch über OPC-UA bereit und persistieren die Kalibrierwerte per INI-Datei.

Initialisierung und Qualität

- Ereignisse: `INIT` (EInit, Eingang) und `INITO` (EInit, Ausgang, kommt von `PublishSpanRaw.INITO`).
- Dateneingang `QI` (BOOL, Vorgabe `TRUE`) und Datenausgang `QO` (BOOL); die internen FBs (`CALIBRATE`, `INI_X_LOW`, `INI_X_HIGH`, `INI_LOW_REF`, `INI_HIGH_REF`) tragen kein festes `QI=TRUE` mehr.
- Initialisierungskette (jeweils `INITO` → `INIT` des nächsten): `INIT` → `CALIBRATE` → `INI_X_LOW` → `INI_X_HIGH` → `INI_LOW_REF` → `INI_HIGH_REF` → `InputRaw` → `Y_OFFSET_LIT` → `Y_SCALE_LIT` → `TriggerCO` → `TriggerCS` → `PublishCal` → `PublishZeroRaw` → `PublishSpanRaw`; `PublishSpanRaw.INITO` → `INITO` und → `AND_QO.REQ`.
- Qualität: `QI` wird an alle 13 Instanzen dieser Kette weitergereicht; deren `QO`-Ausgänge laufen in genau dieser Reihenfolge auf `AND_QO.IN1` bis `IN13` (ein einzelner `iec61131::booleanOperators::AND_BOOL_13`; die frühere Kaskade aus `AND_QO_A` (`AND_BOOL_5`) und `AND_QO` (`AND_BOOL_9`) ist ersetzt, Version 1.12); `AND_QO.OUT` → `QO`.

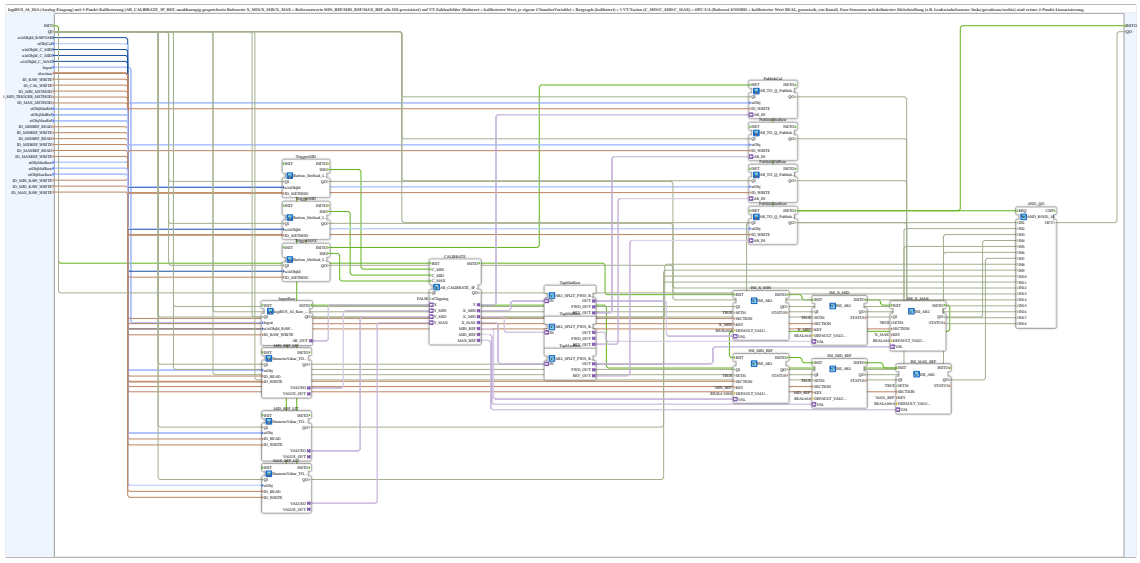
Zusammenfassung

Kompletter 2-Punkt-Kalibrier-Baustein für einen physischen Analogeingang: Rohwert-Erfassung, 2-Punkt-Kalibrierung mit Rohwert-Diagnose, sofort wirksamer Teilkalibrierung, VT-Anzeige/-Eingabe und OPC-UA-Anbindung in einem Composite.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

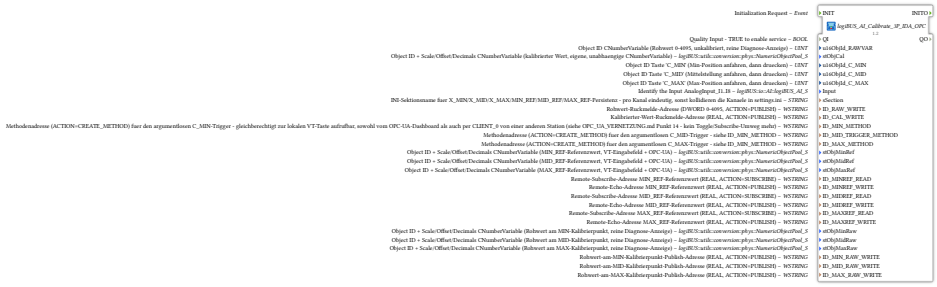
logiBUS_AI_Calibrate_3P_IDA_OPC



• — ○ • — ○ • — ○ •

Einleitung

`logiBUS_AI_Calibrate_3P_IDA_OPC` liest einen logiBUS-Analogeingang (`logiBUS_AI_IDA`) und kalibriert ihn mit einer **3-Punkt-Kalibrierung** (`AR_CALIBRATE_3P_REF`): Die drei Rohwerte `X_MIN`/`X_MID`/`X_MAX` und die drei Referenzwerte `MIN_REF`/`MID_REF`/`MAX_REF` werden unabhängig voneinander gespeichert und alle in einer INI-Datei persistiert. Der Roh- und der kalibrierte Wert werden auf VT-Zahlenfelder (je eigene `CNumberVariable`) ausgegeben, drei VT-Tasten (`C_MIN`/`C_MID`/`C_MAX`) lösen die Kalibrierung aus, und die Werte werden per OPC-UA veröffentlicht (Rohwert `DWORD`, kalibrierter Wert `REAL`; generisch, ein Kanal). Der Baustein ist für Sensoren mit definierter Mittelstellung gedacht (z. B. Lenkwinkelsensor: links/geradeaus/rechts) statt einer reinen 2-Punkt-Linearisierung.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; startet die Init-Kette bei `CALIBRATE.INIT`

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung; kommt von `PublishMaxRaw.INITO`, dem Ende der Init-Kette

Daten-Eingänge

Allgemein:

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input, schaltet den Dienst frei; wird an alle 18 Dienst-Instanzen (siehe `AND_QO`) weitergereicht

Eingang und Anzeige:

- **Input** (`logiBUS::io::AI::logiBUS_AI_S`, Vorgabe `logiBUS_AI::Invalid`): Identifiziert den Analogeingang `AnalogInput_I1..I8`
- **u16ObjId_RAWVAR** (`UINT`, Vorgabe `ID_NULL`): Object-ID der `CNumberVariable` für den unkalibrierten Rohwert (0-4095, reine Diagnose-Anzeige)
- **stObjCal** (`NumericObjectPool_S`): Object-ID + Scale/Offset/Decimals der `CNumberVariable` für den kalibrierten Wert (eigene, unabhängige Variable)
- **sSection** (`STRING`): INI-Sektionsname für die Persistenz von `X_MIN`/`X_MID`/`X_MAX`/`MIN_REF`/`MID_REF`/`MAX_REF`; pro Kanal eindeutig, sonst kollidieren die Kanäle in `settings.ini`

VT-Tasten und Methoden:

- **u16ObjId_C_MIN**, **u16ObjId_C_MID**, **u16ObjId_C_MAX** (`UINT`, Vorgabe `ID_NULL`): Object-IDs der Tasten `C_MIN` (Min-Position anfahren, dann drücken), `C_MID` (Mittelstellung) und `C_MAX` (Max-Position)
- **ID_MIN_METHOD**, **ID_MID_TRIGGER_METHOD**, **ID_MAX_METHOD** (`WSTRING`): Methodenadressen (`ACTION=CREATE_METHOD`) für die argumentlosen Trigger; gleichberechtigt zur lokalen VT-Taste, aufrufbar vom OPC-UA-Dashboard und per `CLIENT_0` von einer anderen Station

Referenzwerte (VT-Eingabefeld + OPC-UA):

- **stObjMinRef**, **stObjMidRef**, **stObjMaxRef** (`NumericObjectPool_S`): Eingabefelder für `MIN_REF`/`MID_REF`/`MAX_REF`
- **ID_MINREF_READ**, **ID_MIDREF_READ**, **ID_MAXREF_READ** (`WSTRING`): Remote-Subscribe-Adressen der Referenzwerte (`REAL`, `ACTION=SUBSCRIBE`)
- **ID_MINREF_WRITE**, **ID_MIDREF_WRITE**, **ID_MAXREF_WRITE** (`WSTRING`): Remote-Echo-Adressen der Referenzwerte (`REAL`, `ACTION=PUBLISH`)

Rohwerte an den Kalibrierpunkten (reine Diagnose-Anzeige):

- **stObjMinRaw**, **stObjMidRaw**, **stObjMaxRaw** (`NumericObjectPool_S`): Anzeigefelder für den Rohwert am MIN-/MID-/MAX-Kalibrierpunkt
- **ID_MIN_RAW_WRITE**, **ID_MID_RAW_WRITE**, **ID_MAX_RAW_WRITE** (`WSTRING`): Publish-Adressen (`REAL`, `ACTION=PUBLISH`)

OPC-UA-Ausgänge:

- **ID_RAW_WRITE** (`WSTRING`): Rohwert-Rückmelde-Adresse (`DWORD` 0-4095, `ACTION=PUBLISH`)
- **ID_CAL_WRITE** (`WSTRING`): Rückmelde-Adresse des kalibrierten Werts (`REAL`, `ACTION=PUBLISH`)

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output, UND-Verknüpfung aller 18 `QO`-Quellen in einem einzigen Baustein `AND_QO` (`iec61131::booleanOperators::AND_BOOL_18`)

Adapter

Dieser Funktionsblock hat keine Adapter an der Schnittstelle; intern werden `AR`-Adapter verwendet.

Verwendete Funktionsbausteine (FBs)

- **InputRaw**: SubApp `MyLib::sys::logiBUS_AI_Raw_VT_AR_OPC` – liest den Analogeingang, zeigt den Rohwert an und publiziert ihn
- **CALIBRATE**: `adapter::Engineering::measurements::AR_CALIBRATE_3P_REF` (`xClipping=FALSE`, `QI` kommt vom Eingang `QI`) – 3-Punkt-Kalibrierung
- **MIN_REF_LIT**, **MID_REF_LIT**, **MAX_REF_LIT**: SubApp `MyLib::sys::NumericValue_TO_AR2_OPC` – Referenzwerte aus VT-Eingabefeld bzw. OPC-UA
- **TriggerMIN**, **TriggerMID**, **TriggerMAX**: SubApp `MyLib::sys::Button_Method_IE` – Taste ODER lokale Methode → `IND`

- **INI_X_MIN**, **INI_X_MID**, **INI_X_MAX**: `eclipse4diac::storage::INI_AR2` (Schlüssel `X_MIN`/`X_MID`/`X_MAX`, Vorgabe `0.0`)
- **INI_MIN_REF**, **INI_MID_REF**, **INI_MAX_REF**: `INI_AR2` (Schlüssel `MIN_REF`/`MID_REF`/`MAX_REF`, Vorgaben `-100.0`/`0.0`/`100.0`)
- **TapMinRaw**, **TapMidRaw**, **TapMaxRaw**: `adapter::events::bidirectional::AR2_SPLIT_FWD_REV`
- **PublishCal**, **PublishMinRaw**, **PublishMidRaw**, **PublishMaxRaw**: SubApp `MyLib::sys::AR_TO_Q_Publish_OPC` – VT-Anzeige und OPC-UA-Publish
- **AND_QO**: `iec61131::booleanOperators::AND_BOOL_18` – verknüpft die `QO`-Ausgänge aller 18 Dienst-Instanzen (`CALIBRATE`, 6× `INI_AR2`, `InputRaw`, 3× `*_REF_LIT`, 3× `Trigger*`, `PublishCal`, 3× `Publish*Raw`) zum Ausgang `QO` (früher eine Kaskade aus mehreren kleineren UND-Bausteinen)

Alle `INI_AR2` verwenden `SETM=TRUE`; ihr `QI` kommt vom Eingang `QI`; `sSection` ist ihre gemeinsame `SECTION`.

Programmablauf und Verbindungen

1. `InputRaw.AR_OUT` ⟶ `CALIBRATE.X` (Rohwert).
2. `TriggerMIN.IND`/`TriggerMID.IND`/`TriggerMAX.IND` ⟶ `CALIBRATE.C_MIN`/`C_MID`/`C_MAX`.
3. Referenzwerte: `MIN_REF_LIT.VALUEO`/`MID_REF_LIT.VALUEO`/`MAX_REF_LIT.VALUEO` ⟶ `CALIBRATE.Y_MIN`/`Y_MID`/`Y_MAX`.
4. Gespeicherte Rohwerte: `CALIBRATE.X_MIN`/`X_MID`/`X_MAX` ⟶ `TapMinRaw.IN`/`TapMidRaw.IN`/`TapMaxRaw.IN`; deren `OUT` ⟶ `INI_X_MIN.VAL`/`INI_X_MID.VAL`/`INI_X_MAX.VAL`; `REV_OUT` ⟶ `PublishMinRaw`/`PublishMidRaw`/`PublishMaxRaw.AR_IN`.
5. Referenzwerte (Persistenz): `CALIBRATE.MIN_REF`/`MID_REF`/`MAX_REF` ⟶ `INI_MIN_REF.VAL`/`INI_MID_REF.VAL`/`INI_MAX_REF.VAL`.
6. Kalibrierter Wert: `CALIBRATE.Y` ⟶ `PublishCal.AR_IN`.
7. Initialisierungskette: `INIT` ⟶ `CALIBRATE` ⟶ `INI_X_MIN` ⟶ `INI_X_MID` ⟶ `INI_X_MAX` ⟶ `INI_MIN_REF` ⟶ `INI_MID_REF` ⟶ `INI_MAX_REF` ⟶ `InputRaw` ⟶ `MIN_REF_LIT` ⟶ `MID_REF_LIT` ⟶ `MAX_REF_LIT` ⟶ `TriggerMIN` ⟶ `TriggerMID` ⟶ `TriggerMAX` ⟶ `PublishCal` ⟶ `PublishMinRaw` ⟶ `PublishMidRaw` ⟶ `PublishMaxRaw` (jeweils `INITO` ⟶ nächstes `INIT`); `PublishMaxRaw.INITO` ⟶ `INITO` und `AND_QO.REQ`.
8. Qualität: `QI` ⟶ `.QI` aller 18 Instanzen; `CALIBRATE.QO` ⟶ `AND_QO.IN1`; `INI_X_MIN`/`INI_X_MID`/`INI_X_MAX.QO` ⟶ `IN2`/`IN3`/`IN4`; `INI_MIN_REF`/`INI_MID_REF`/`INI_MAX_REF.QO` ⟶ `IN5`/`IN6`/`IN7`; `InputRaw.QO` ⟶ `IN8`; `MIN_REF_LIT`/`MID_REF_LIT`/`MAX_REF_LIT.QO` ⟶ `IN9`/`IN10`/`IN11`; `TriggerMIN`/`TriggerMID`/`TriggerMAX.QO` ⟶ `IN12`/`IN13`/`IN14`; `PublishCal.QO` ⟶ `IN15`; `PublishMinRaw`/`PublishMidRaw`/`PublishMaxRaw.QO` ⟶ `IN16`/`IN17`/`IN18`; `AND_QO.OUT` ⟶ `QO`.

Technische Besonderheiten

- Alle sechs Kalibrier-Eckwerte (drei Rohwerte, drei Referenzwerte) sind unabhängig voneinander über je einen `INI_AR2` INI-persistiert.
- `xClipping=FALSE`: Der kalibrierte Wert wird nicht begrenzt.
- `sSection` muss pro Kanal eindeutig sein.

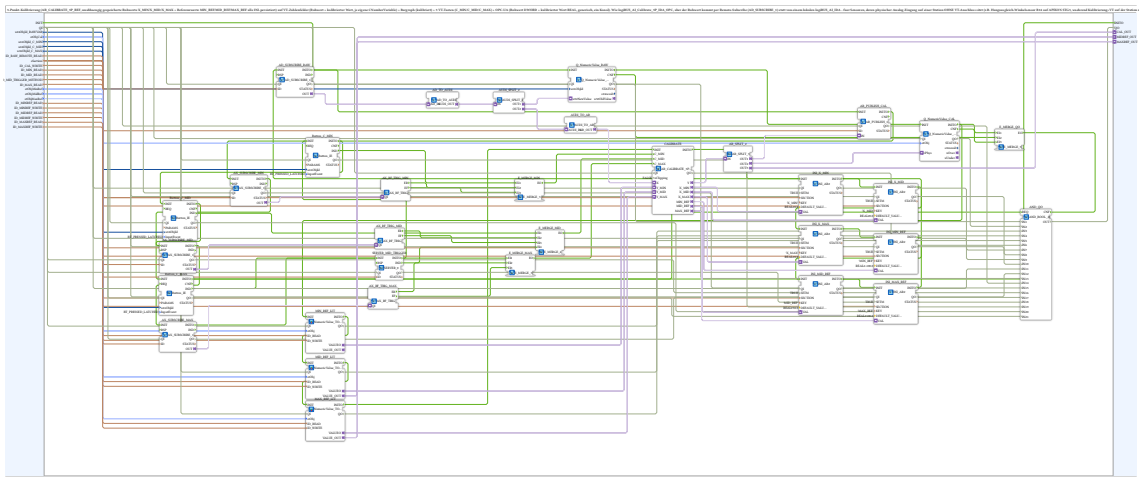
Anwendungsszenarien

- Sensoren mit definierter Mittelstellung, z. B. Lenkwinkelsensor (links/geradeaus/rechts)
- Kalibrierung per VT-Taste, OPC-UA-Dashboard oder Remote-Aufruf von einer anderen Station

Fazit

`logiBUS_AI_Calibrate_3P_IDA_OPC` bündelt Analogeingang, 3-Punkt-Kalibrierung, INI-Persistenz, VT-Anzeige und OPC-UA-Anbindung eines Kanals in einer SubApp.

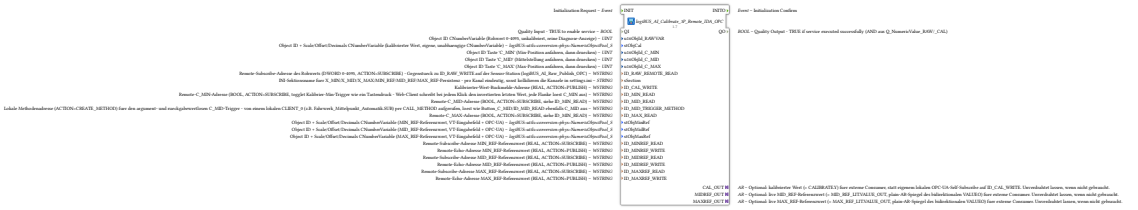
logiBUS_AI_Calibrate_3P_Remote_IDA_OPC



• – ○ • – ○ • – ○ •

Einleitung

`logiBUS_AI_Calibrate_3P_Remote_IDA_OPC` führt eine **3-Punkt-Kalibrierung** (`AR_CALIBRATE_3P_REF`) für einen Analog-Rohwert durch. Die Rohwerte `X_MIN`/`X_MID`/`X_MAX` und die Referenzwerte `MIN_REF`/`MID_REF`/`MAX_REF` werden unabhängig voneinander gespeichert und alle INI-persistiert. Die Ergebnisse erscheinen auf VT-Zahlenfeldern (Rohwert und kalibrierter Wert, je eigene CNumberVariable), drei VT-Tasten (`C_MIN`/`C_MID`/`C_MAX`) lösen die Kalibrierpunkte aus, und per OPC-UA werden der Rohwert (DWORD) und der kalibrierte Wert (REAL) generisch für einen Kanal bereitgestellt. Im Unterschied zu `logiBUS_AI_Calibrate_3P_IDA_OPC` kommt der Rohwert **per Remote-Subscribe** (`AD_SUBSCRIBE_1`) statt von einem lokalen `logiBUS_AI_IDA`; so lassen sich Sensoren auf einer Station ohne VT-Anschluss kalibrieren (z. B. Hangausgleich-Winkelsensor B14 auf STG3), während Kalibrierung und VT auf der Station mit VT-Anschluss laufen. Der Rohwert wird dort per `logiBUS_AI_Raw_Publish_OPC` publiziert.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; startet die Initialisierungskette (erstes Glied `Q_NumericValue_RAW.INIT`)

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierung abgeschlossen; kommt von `Button_C_MAX.INITO`, dem Ende der Kette

Daten-Eingänge

- **QI** (BOOL): Quality Input - TRUE aktiviert den Dienst (Default `TRUE`); wird an alle 21 inneren Instanzen verteilt
- **u16ObjId_RAWVAR** (UINT): Object ID der CNumberVariable für den Rohwert (0-4095, unkalibriert, reine Diagnose-Anzeige)
- **stObjCal** (`NumericObjectPool_S`): Object ID + Skalierung/Offset/Dezimalstellen der CNumberVariable für den kalibrierten Wert (eigene, unabhängige Variable)
- **u16ObjId_C_MIN**, **u16ObjId_C_MID**, **u16ObjId_C_MAX** (UINT): Object IDs der Tasten `C_MIN`, `C_MID`, `C_MAX` (Position anfahren, dann drücken)
- **ID_RAW_REMOTE_READ** (WSTRING): Remote-Subscribe-Adresse des Rohwerts (DWORD 0-4095, `ACTION=SUBSCRIBE`); Gegenstück zu `ID_RAW_WRITE` auf der Sensor-Station

- **sSection** (STRING): INI-Sektionsname für die Persistenz von X_MIN/X_MID/X_MAX/MIN_REF/MID_REF/MAX_REF; pro Kanal eindeutig, sonst kollidieren die Kanäle in settings.ini
- **ID_CAL_WRITE** (WSTRING): Rückmelde-Adresse des kalibrierten Werts (REAL, ACTION=PUBLISH)
- **ID_MIN_READ**, **ID_MID_READ**, **ID_MAX_READ** (WSTRING): Remote-Adressen (BOOL, ACTION=SUBSCRIBE) für C_MIN/C_MID/C_MAX; jede Flanke (der Web-Client schreibt bei jedem Klick den invertierten letzten Wert) löst den Kalibrierpunkt wie ein Tastendruck aus
- **ID_MID_TRIGGER_METHOD** (WSTRING): lokale Methodenadresse (ACTION=CREATE_METHOD) für den argument- und rückgabewertlosen C_MID-Trigger; wird von einem lokalen CLIENT_0 (z. B. Fahrwerk_Mittelpunkt_Automatik.SUB) per CALL_METHOD aufgerufen und löst C_MID wie Button_C_MID/ID_MID_READ aus
- **stObjMinRef**, **stObjMidRef**, **stObjMaxRef** (NumericObjectPool_S): Object ID + Skalierung/Offset/Dezimalstellen der Referenzwerte MIN_REF/MID_REF/MAX_REF (VT-Eingabefeld + OPC-UA)
- **ID_MINREF_READ**, **ID_MIDREF_READ**, **ID_MAXREF_READ** (WSTRING): Remote-Subscribe-Adressen der Referenzwerte (REAL, ACTION=SUBSCRIBE)
- **ID_MINREF_WRITE**, **ID_MIDREF_WRITE**, **ID_MAXREF_WRITE** (WSTRING): Remote-Echo-Adressen der Referenzwerte (REAL, ACTION=PUBLISH)

Daten-Ausgänge

- **QO** (BOOL): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde (AND der QO aller 21 inneren Instanzen, siehe Abschnitt Funktionsweise)

Adapter

- **CAL_OUT** (Plug, AR): optional; kalibrierter Wert (CALIBRATE.Y) für externe Konsumenten, statt eines eigenen lokalen OPC-UA-Self-Subscribe auf ID_CAL_WRITE
- **MIDREF_OUT** (Plug, AR): optional; live-MID_REF-Referenzwert (MID_REF_LIT.VALUE_OUT, einfacher AR-Spiegel des bidirektionalen VALUE0) für externe Konsumenten
- **MAXREF_OUT** (Plug, AR): optional; live-MAX_REF-Referenzwert (MAX_REF_LIT.VALUE_OUT) für externe Konsumenten

Funktionsweise

Rohwert: AD_SUBSCRIBE_RAW (adapter::net::AD_SUBSCRIBE_1, QI von außen, Adresse ID_RAW_REMOTE_READ) ⟶ AD_TO_AUDI ⟶ AUDI_SPLIT_2. OUT1 ⟶ Q_NumericValue_RAW.u32NewValue (Q_NumericValue_AUDI, Rohwert-Anzeige auf u16ObjId_RAWVAR); OUT2 ⟶ AUDI_TO_AR ⟶ CALIBRATE.X.

Kalibrierung: CALIBRATE (adapter::Engineering::measurements::AR_CALIBRATE_3P_REF, xClipping = FALSE, QI von außen) bildet den Rohwert X über die drei Punkte auf Y ab. Y_MIN/Y_MID/Y_MAX kommen von MIN_REF_LIT/MID_REF_LIT/MAX_REF_LIT (je MyLib::sys::NumericValue_TO_AR2_OPC VT-Eingabefeld und OPC-UA, letzter Schreiber gewinnt).

Ergebnis: CALIBRATE.Y ⟶ AR_SPLIT_2 (AR_SPLIT_3): OUT1 ⟶ AR_PUBLISH_CAL (AR_PUBLISH_1, Adresse ID_CAL_WRITE), OUT2 ⟶ Q_NumericValue_CAL.rPhys (Q_NumericValue_PHYSA, Anzeige mit stObjCal), OUT3 ⟶ CAL_OUT. MID_REF_LIT.VALUE_OUT ⟶ MIDREF_OUT und MAX_REF_LIT.VALUE_OUT ⟶ MAXREF_OUT.

Trigger: Jeder Kalibrierpunkt hat einen eigenen Event-Merge. C_MIN: Button_C_MIN.IND (Button_IE, BT_PRESSED_LATCHED), AX_RF_TRIG_MIN.ER und .EF (beide Flanken des Remote-Subscribe AX_SUBSCRIBE_MIN, Adresse ID_MIN_READ) ⟶ E_MERGE_MIN (E_MERGE_3) ⟶ CALIBRATE.C_MIN. C_MAX analog über Button_C_MAX, AX_SUBSCRIBE_MAX, AX_RF_TRIG_MAX, E_MERGE_MAX (E_MERGE_3). C_MID zusätzlich mit SERVER_MID_TRIGGER (iec61499::net::SERVER_0, Methode ID_MID_TRIGGER_METHOD) ⟶ E_MERGE_MID (E_MERGE_4) ⟶ CALIBRATE.C_MID.

Persistenz: CALIBRATE.X_MIN/X_MID/X_MAX und MIN_REF/MID_REF/MAX_REF gehen an je eine INI_AR2-Instanz (INI_X_MIN, INI_X_MID, INI_X_MAX, INI_MIN_REF, INI_MID_REF, INI_MAX_REF; SETM = TRUE, SECTION aus sSection, KEY = Name). Die Defaults sind X_MIN/X_MID/X_MAX = REAL#0.0 sowie MIN_REF = REAL#-100.0, MID_REF = REAL#50.0, MAX_REF = REAL#100.0.

Initialisierung: INIT läuft als eine einzige Kette (jeweils INITO ⟶ nächstes INIT) durch Q_NumericValue_RAW ⟶ Q_NumericValue_CAL ⟶ MIN_REF_LIT ⟶ MID_REF_LIT ⟶ MAX_REF_LIT ⟶ CALIBRATE ⟶ INI_X_MIN ⟶ INI_X_MID ⟶ INI_X_MAX ⟶ INI_MIN_REF ⟶ INI_MID_REF ⟶ INI_MAX_REF ⟶ AD_SUBSCRIBE_RAW ⟶ AR_PUBLISH_CAL ⟶ AX_SUBSCRIBE_MIN ⟶ AX_SUBSCRIBE_MID ⟶ AX_SUBSCRIBE_MAX ⟶ SERVER_MID_TRIGGER ⟶ Button_C_MIN ⟶ Button_C_MID ⟶ Button_C_MAX; Button_C_MAX.INITO geht an den Ereignisausgang INITO und an AND_QO.REQ.

Qualität: QI geht an alle 21 Instanzen der Kette. QO ist der Ausgang der Instanz AND_QO (iec61131::booleanOperators::AND_BOOL_21), die die QO dieser 21 Instanzen in einem einzigen Baustein verknüpft (IN1..IN21: Q_NumericValue_RAW, Q_NumericValue_CAL, MIN_REF_LIT, MID_REF_LIT,

MAX_REF_LIT, CALIBRATE, INI_X_MIN, INI_X_MID, INI_X_MAX, INI_MIN_REF, INI_MID_REF, INI_MAX_REF, AD_SUBSCRIBE_RAW, AR_PUBLISH_CAL, AX_SUBSCRIBE_MIN, AX_SUBSCRIBE_MID, AX_SUBSCRIBE_MAX, SERVER_MID_TRIGGER, Button_C_MIN, Button_C_MID, Button_C_MAX). Zusätzlich existiert E_MERGE_QO (E_MERGE_3), das Q_NumericValue_CAL.INITO sowie die CNF-Ereignisse von Q_NumericValue_RAW und Q_NumericValue_CAL zusammenführt; sein Ausgang EO triggert AND_QO.REQ, sodass QO bei jedem neuen Q_NumericValue_RAW-/Q_NumericValue_CAL-Wert (CNF) neu ausgewertet wird (bis v1.6 war EO nicht angeschlossen).

Technische Besonderheiten

- Die Defaults von X_MID/X_MAX stehen bewusst auf 0.0: Mit 0.5/1.0 latchten die Kalibriert-Flags xMidCal/xMaxCal von AR_CALIBRATE_3P_REF bei jedem Neustart sofort auf TRUE, obwohl MID/MAX nie kalibriert wurden (Fix in Version 1.6)
- CALIBRATE hat ein INIT/QI/QO-Enable-Muster; ohne QI = TRUE bleibt der Baustein im Zustand START hängen und REQ läuft nie (Version 1.5); QI kommt hier vom Eingang QI der SubApp (Default TRUE) statt aus einem festen Parameter
- MIDREF_OUT/MAXREF_OUT werden aus VALUE_OUT gespeist, nicht aus dem bidirektionalen VALUEO: ein AR_SPLIT_2 direkt auf einen AR2-Plug wäre ein Typfehler (Version 1.2)
- Die optionalen Plugs CAL_OUT/MIDREF_OUT/MAXREF_OUT erlauben externen Konsumenten, den bereits kalibrierten Wert per Adapter zu lesen, statt denselben OPC-UA-Knoten lokal nochmals zu abonnieren (FORTE: ‚Cannot add another one'); unverdrahtet lassen, wenn nicht gebraucht (z. B. Hangausgleich-Kanal)
- Die Merge-Ketten für C_MIN/C_MID/C_MAX sind je ein einziger E_MERGE_3/E_MERGE_4 statt mehrerer E_MERGE_2 (ODER ist assoziativ, funktional identisch); C_MIN/C_MID/C_MAX von CALIBRATE sind reine EventInputs
- Remote-Trigger lösen bei **jeder** Flanke aus (steigend und fallend), da der Web-Client bei jedem Klick den invertierten letzten Wert schreibt

Anwendungsszenarien

- Kalibrierung eines Winkel- oder Positionssensors (z. B. Hangausgleich), dessen Analog-Eingang auf einer anderen Station sitzt als das VT
- Dreipunktkalibrierung mit Min-, Mitte- und Max-Stellung samt einstellbaren Referenzwerten und persistierten Rohwerten

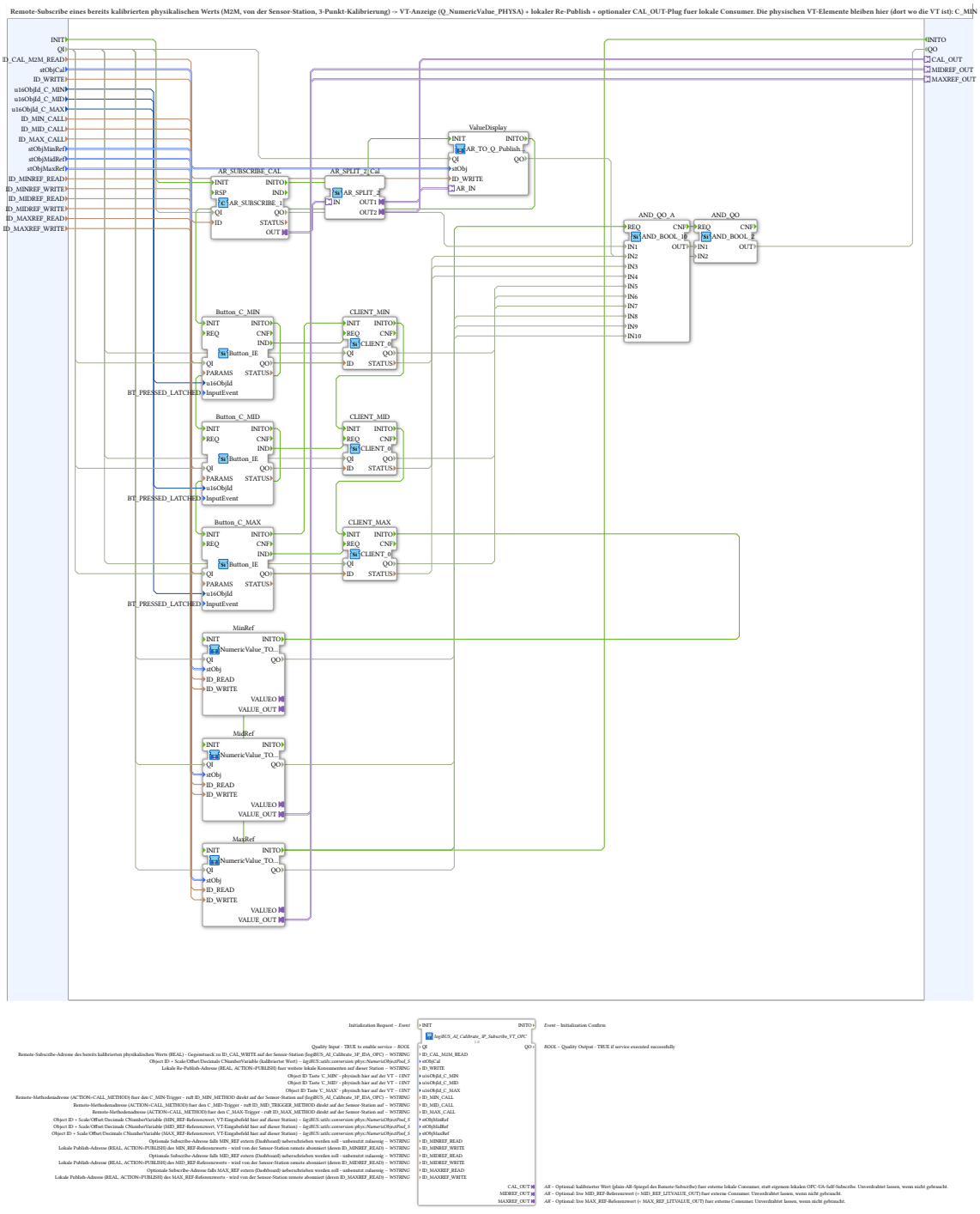
⚖️ Vergleich mit ähnlichen Bausteinen

logiBUS_AI_Calibrate_3P_IDA_OPC liest den Rohwert lokal von einem logiBUS_AI_IDA; diese Remote-Variante ersetzt dessen logiBUS_AI_IDA + AD_SPLIT_2 + AD_PUBLISH_RAW durch ein AD_SUBSCRIBE_1. Die Kette von Kalibrierung und VT-Anbindung ist unverändert. Die Sensor-Seite ist logiBUS_AI_Raw_Publish_OPC.

Fazit

logiBUS_AI_Calibrate_3P_Remote_IDA_OPC kalibriert einen remote abonnierten Analog-Rohwert über drei Punkte, persistiert alle Kalibrierdaten, zeigt Rohwert und kalibrierten Wert am VT und stellt sie per OPC-UA bereit.

logiBUS_AI_Calibrate_3P_Subscribe_VT_OPC



• — ○ • — ○ • — ○ •

Einleitung

`logiBUS_AI_Calibrate_3P_Subscribe_VT_OPC` ist die VT-seitige Gegenstelle zur Sensor-Station `logiBUS_AI_Calibrate_3P_IDA_OPC`: Die SubApp abonniert per OPC-UA (M2M) den **bereits kalibrierten** physikalischen Wert der 3-Punkt-Kalibrierung, zeigt ihn in der VT an, publiziert ihn lokal erneut und stellt ihn optional über den Plug `CAL_OUT` für lokale Consumer bereit. Die physischen VT-Elemente bleiben hier (dort, wo die VT ist): Die Tasten `C_MIN`/`C_MID`/`C_MAX` rufen per `CLIENT_0` direkt die entsprechenden Methoden auf der Sensor-Station auf (echter Ereignis-zu-Ereignis-Methodenaufruf, kein Toggle/Publish/Subscribe-Umweg). Die drei Referenzwert-Eingabefelder der `MIN_REF`/`MID_REF`/`MAX_REF` (`NumericValue_TO_AR2_OPC`) werden weiterhin **lokal** publiziert und von der Sensor-Station remote abonniert; `MID_REF` und `MAX_REF` werden zusätzlich als plain-AR-

Plugs (MIDREF_OUT/MAXREF_OUT) re-exponiert. Genutzt vom 3-Punkt-Kalibrierungs-Trainingsbeispiel InputOutputTesterButton_AI_Calibrate_3P_OPC_UA.

Schnittstellenstruktur

Ereignis-Eingänge
- **INIT** (EInit): Initialisierungsanforderung

Ereignis-Ausgänge
- **INITO** (EInit): Initialisierungsbestätigung (von MaxRef.INITO)

Daten-Eingänge
- **QI** (BOOL, Vorgabe TRUE): Quality Input – TRUE gibt die Dienste frei
- **ID_CAL_M2M_READ** (WSTRING): Remote-Subscribe-Adresse des bereits kalibrierten physikalischen Werts (REAL), Gegenstück zu ID_CAL_WRITE auf der Sensor-Station
- **stObjCal** (NumericObjectPool_S): Object ID + Scale/Offset/Decimals der CNumberVariable für den kalibrierten Wert
- **ID_WRITE** (WSTRING): lokale Re-Publish-Adresse (REAL, ACTION=PUBLISH) für weitere lokale Konsumenten
- **u16ObjId_C_MIN**, **u16ObjId_C_MID**, **u16ObjId_C_MAX** (UINT, Vorgabe ID_NULL): Object IDs der Tasten C_MIN/C_MID/C_MAX auf der VT
- **ID_MIN_CALL**, **ID_MID_CALL**, **ID_MAX_CALL** (WSTRING): Remote-Methodenadressen (ACTION=CALL_METHOD) für die drei Kalibrier-Trigger auf der Sensor-Station
- **stObjMinRef**, **stObjMidRef**, **stObjMaxRef** (NumericObjectPool_S): Object ID + Scale/Offset/Decimals der Referenzwert-Eingabefelder auf dieser Station
- **ID_MINREF_READ**, **ID_MIDREF_READ**, **ID_MAXREF_READ** (WSTRING): optionale Subscribe-Adressen, falls der Referenzwert extern (Dashboard) überschrieben werden soll; unbenutzt zulässig
- **ID_MINREF_WRITE**, **ID_MIDREF_WRITE**, **ID_MAXREF_WRITE** (WSTRING): lokale Publish-Adressen (REAL, ACTION=PUBLISH) der Referenzwerte, werden von der Sensor-Station remote abonniert

Daten-Ausgänge
- **QO** (BOOL): Quality Output – TRUE, wenn alle Dienste erfolgreich waren (UND aus AND_QO_A.OUT und ValueDisplay.QO)

Adapter
- **CAL_OUT** (Plug, adapter::types::unidirectional::AR): optional, kalibrierter Wert (plain-AR-Spiegel des Remote-Subscribe) für externe lokale Consumer; unverdrahtet lassen, wenn nicht gebraucht
- **MIDREF_OUT** (Plug, adapter::types::unidirectional::AR): optional, live MID_REF-Referenzwert
- **MAXREF_OUT** (Plug, adapter::types::unidirectional::AR): optional, live MAX_REF-Referenzwert

Verwendete Funktionsbausteine (FBs)
- **AR_SUBSCRIBE_CAL** (adapter::net::AR_SUBSCRIBE_1): abonniert den kalibrierten Wert von der Sensor-Station (ID_CAL_M2M_READ).
- **AR_SPLIT_2_Cal** (adapter::events::unidirectional::AR_SPLIT_2): verteilt den Wert auf ValueDisplay und CAL_OUT.
- **ValueDisplay** (SubApp AR_TO_Q_Publish_OPC): VT-Anzeige und lokaler Re-Publish (stObjCal, ID_WRITE).
- **Button_C_MIN**, **Button_C_MID**, **Button_C_MAX** (isobus::UT::io::Button::Button_IE, InputEvent = BT_PRESSED_LATCHED): lesen die drei VT-Tasten.
- **CLIENT_MIN**, **CLIENT_MID**, **CLIENT_MAX** (iec61499::net::CLIENT_0): rufen die Methoden auf der Sensor-Station auf (ID_MIN_CALL/ID_MID_CALL/ID_MAX_CALL).
- **MinRef**, **MidRef**, **MaxRef** (SubApp NumericValue_TO_AR2_OPC): VT-Eingabefelder der Referenzwerte, lokal publiziert.
- **AND_QO_A** (iec61131::booleanOperators::AND_BOOL_10) und **AND_QO** (iec61131::booleanOperators::AND_BOOL_2): verknüpfen die QO-Ausgänge aller Teilbausteine zum QO der SubApp.

Funktionsweise

- **Kalibrierter Wert:** `AR_SUBSCRIBE_CAL.OUT` ⟶ `AR_SPLIT_2_Cal.IN`; `AR_SPLIT_2_Cal.OUT1` ⟶ `ValueDisplay.AR_IN`; `AR_SPLIT_2_Cal.OUT2` ⟶ `CAL_OUT`.
- **Kalibriertasten:** `Button_C_MIN.IND` ⟶ `CLIENT_MIN.REQ`, `Button_C_MID.IND` ⟶ `CLIENT_MID.REQ`, `Button_C_MAX.IND` ⟶ `CLIENT_MAX.REQ`.
- **Referenzwerte:** `MinRef/MidRef/MaxRef` erhalten `stObjMinRef/stObjMidRef/stObjMaxRef`, `ID_*REF_READ` und `ID_*REF_WRITE`; `MidRef.VALUE_OUT` ⟶ `MIDREF_OUT`, `MaxRef.VALUE_OUT` ⟶ `MAXREF_OUT`.
- **Initialisierungskette:** `INIT` ⟶ `AR_SUBSCRIBE_CAL` ⟶ `ValueDisplay` ⟶ `Button_C_MIN` ⟶ `Button_C_MID` ⟶ `Button_C_MAX` ⟶ `CLIENT_MIN` ⟶ `CLIENT_MID` ⟶ `CLIENT_MAX` ⟶ `MinRef` ⟶ `MidRef` ⟶ `MaxRef` (jeweils `INITO` ⟶ nächstes `INIT`); `MaxRef.INITO` ⟶ `INITO` und ⟶ `AND_QO_A.REQ`; `AND_QO_A.CNF` ⟶ `AND_QO.REQ`.
- **Qualität:** `QI` ⟶ `.QI` von `AR_SUBSCRIBE_CAL`, `Button_C_MIN/_MID/_MAX`, `CLIENT_MIN/_MID/_MAX`, `MinRef`, `MidRef`, `MaxRef` und `ValueDisplay`. `AR_SUBSCRIBE_CAL.QO`, die drei `Button_C_*.QO`, die drei `CLIENT_*.QO` sowie `MinRef.QO/MidRef.QO/MaxRef.QO` ⟶ `AND_QO_A.IN1..IN10`; `AND_QO_A.OUT` ⟶ `AND_QO.IN1`, `ValueDisplay.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`.

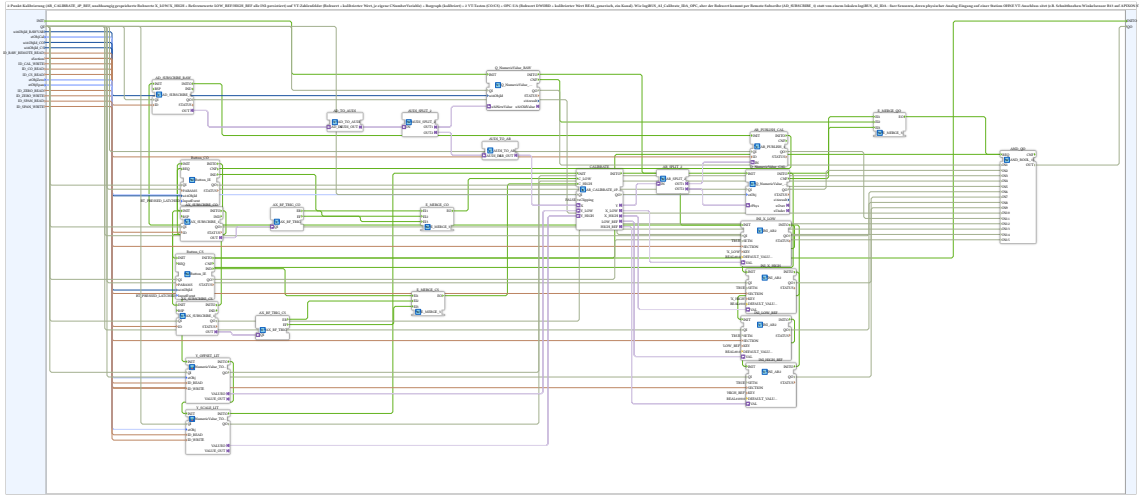
Zusammenfassung

Remote-Anzeige-Baustein für einen bereits kalibrierten 3-Punkt-Wert: Subscribe des kalibrierten Werts, VT-Anzeige mit lokalem Re-Publish, Kalibriertasten als Methodenaufrufe an die Sensor-Station und lokal publizierte Referenzwert-Eingabefelder in einem Composite.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

logiBUS_AI_Calibrate_Remote_IDA_OPC



• – ○ • – ○ • – ○ •

Einleitung

`logiBUS_AI_Calibrate_Remote_IDA_OPC` ist eine 2-Punkt-Kalibrierung (`AR_CALIBRATE_2P_REF`, unabhängig gespeicherte Rohwerte `X_LOW/X_HIGH` und Referenzwerte `LOW_REF/HIGH_REF`, alle INI-persistiert) mit VT-Zahlenfeldern (Rohwert und kalibrierter Wert, je eigene `CNumberVariable`), zwei VT-Tasten (CO/CS) und OPC-UA (Rohwert `DWORD`, kalibrierter Wert `REAL`, generisch, ein Kanal). Es entspricht `logiBUS_AI_Calibrate_2P_IDA_OPC`, aber der Rohwert kommt per **Remote-Subscribe** (`AD_SUBSCRIBE_1`) statt von einem lokalen `logiBUS_AI_IDA`. Das ist für Sensoren gedacht, deren physischer Analogeingang auf einer Station **ohne VT-Anschluss** sitzt (z. B. Schnitthöhen-Winkelsensor B13 auf APIXON STG3, während Kalibrierung und VT auf der Station mit VT-Anschluss laufen). Der Rohwert wird dort per `logiBUS_AI_Raw_Publish_OPC` publiziert.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialization Request

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialization Confirm

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input, TRUE aktiviert den Dienst
- **u16ObjId_RAWVAR** (`UINT`): Object ID `CNumberVariable` (Rohwert 0..4095, unkalibriert, reine Diagnose-Anzeige), Default `ID_NULL`
- **stObjCal** (`NumericObjectPool_S`): Object ID plus Scale/Offset/Decimals der `CNumberVariable` für den kalibrierten Wert (eigene, unabhängige Variable)
- **u16ObjId_CO** (`UINT`): Object ID Taste „CO“ (Kalibrier-Nullpunkt: Referenz 1 anlegen, dann drücken)
- **u16ObjId_CS** (`UINT`): Object ID Taste „CS“ (Kalibrier-Spanne: Referenz 2 anlegen, dann drücken; erst nach CO möglich)
- **ID_RAW_REMOTE_READ** (`WSTRING`): Remote-Subscribe-Adresse des Rohwerts (`DWORD` 0..4095, `ACTION=SUBSCRIBE`), Gegenstück zu `ID_RAW_WRITE` auf der Sensor-Station
- **sSection** (`STRING`): INI-Sektionsname für die Persistenz von `X_LOW/X_HIGH/LOW_REF/HIGH_REF`; pro Kanal eindeutig, sonst kollidieren die Kanäle in `settings.ini`
- **ID_CAL_WRITE** (`WSTRING`): Rückmelde-Adresse des kalibrierten Werts (`REAL`, `ACTION=PUBLISH`)
- **ID_CO_READ** (`WSTRING`): Remote-CO-Adresse (`BOOL`, `ACTION=SUBSCRIBE`): togglet den Kalibrier-Nullpunkt-Trigger wie ein Tastendruck; der Web-Client schreibt bei jedem Klick den invertierten letzten Wert, jede Flanke (steigend oder fallend) löst CO aus
- **ID_CS_READ** (`WSTRING`): Remote-CS-Adresse (`BOOL`, `ACTION=SUBSCRIBE`): togglet den Kalibrier-Spanne-Trigger, siehe `ID_CO_READ`
- **stObjZero** (`NumericObjectPool_S`): `CNumberVariable` des ZERO-Referenzwerts (VT-Eingabefeld + OPC-UA)
- **stObjSpan** (`NumericObjectPool_S`): `CNumberVariable` des SPAN-Referenzwerts (VT-Eingabefeld + OPC-UA)
- **ID_ZERO_READ** (`WSTRING`): Remote-Subscribe-Adresse ZERO-Referenzwert (`REAL`, `ACTION=SUBSCRIBE`)
- **ID_ZERO_WRITE** (`WSTRING`): Remote-Echo-Adresse ZERO-Referenzwert (`REAL`, `ACTION=PUBLISH`)
- **ID_SPAN_READ** (`WSTRING`): Remote-Subscribe-Adresse SPAN-Referenzwert (`REAL`, `ACTION=SUBSCRIBE`)
- **ID_SPAN_WRITE** (`WSTRING`): Remote-Echo-Adresse SPAN-Referenzwert (`REAL`, `ACTION=PUBLISH`)

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output, UND-Verknüpfung (`AND_QO`, ein einzelner `AND_BOOL_15`) der `QO`-Ausgänge aller 15 inneren Bausteine mit `QO`: `Q_NumericValue_RAW`, `Q_NumericValue_CAL`, `Y_OFFSET_LIT`, `Y_SCALE_LIT`, `CALIBRATE`, `INI_X_LOW`, `INI_X_HIGH`, `INI_LOW_REF`, `INI_HIGH_REF`, `AD_SUBSCRIBE_RAW`, `AR_PUBLISH_CAL`, `AX_SUBSCRIBE_CO`, `AX_SUBSCRIBE_CS`, `Button_CO`, `Button_CS`

Adapter

Keine.

Verwendete Funktionsbausteine (FBs)

- **`AD_SUBSCRIBE_RAW`** (`adapter::net::AD_SUBSCRIBE_1`): abonniert den Remote-Rohwert
- **`AD_TO_AUDI`**, **`AUDI_SPLIT_2`**, **`AUDI_TO_AR`**: Umwandlung und Aufteilung des Rohwerts
- **`Q_NumericValue_RAW`** (`isobus::UT::Q::Q_NumericValue_AUDI`): VT-Rohwertanzeige

- **CALIBRATE** (adapter::Engineering::measurements::AR_CALIBRATE_2P_REF, xClipping = FALSE): 2-Punkt-Kalibrierung
- **Y_OFFSET_LIT**, **Y_SCALE_LIT** (SubApp MyLib::sys::NumericValue_TO_AR2_OPC, NumericValue_TO_AR2_OPC): ZERO-/SPAN-Referenzwerte als AR2
- **Button_CO**, **Button_CS** (isobus::UT::io::Button::Button_IE, InputEvent = BT_PRESSED_LATCHED): VT-Tasten
- **AX_SUBSCRIBE_CO**, **AX_SUBSCRIBE_CS** (adapter::net::AX_SUBSCRIBE_1): Remote-Trigger
- **AX_RF_TRIG_CO**, **AX_RF_TRIG_CS** (adapter::events::unidirectional::AX_RF_TRIG): erkennen steigende und fallende Flanke
- **E_MERGE_CO**, **E_MERGE_CS** (iec61499::events::E_MERGE_3): führen Taste, steigende und fallende Flanke zusammen
- **INI_X_LOW**, **INI_X_HIGH**, **INI_LOW_REF**, **INI_HIGH_REF** (eclipse4diac::storage::INI_AR2, SETM = TRUE): Persistenz; Schlüssel X_LOW/X_HIGH/LOW_REF/HIGH_REF, Defaults 0.0/0.0/0.0/100.0
- **AND_QO** (iec61131::booleanOperators::AND_BOOL_15, seit v1.5): ersetzt die frühere Kaskade AND_QO_A (AND_BOOL_4) + AND_QO_B (AND_BOOL_10) + AND_QO (AND_BOOL_3) durch einen einzelnen Baustein für alle 15 QO-Quellen
- **AR_SPLIT_2** (adapter::events::unidirectional::AR_SPLIT_2), **AR_PUBLISH_CAL** (adapter::net::AR_PUBLISH_1), **Q_NumericValue_CAL** (isobus::UT::Q::Q_NumericValue_PHYSA): Ausgabe des kalibrierten Werts

Funktionsweise

- Rohwertpfad: AD_SUBSCRIBE_RAW.OUT → AD_TO_AUDI.AD_IN → AUDI_SPLIT_2.IN; AUDI_SPLIT_2.OUT1 → Q_NumericValue_RAW.u32NewValue (VT-Anzeige); AUDI_SPLIT_2.OUT2 → AUDI_TO_AR.AUDI_IN → CALIBRATE.X
- Persistenz: CALIBRATE.X_LOW/X_HIGH/LOW_REF/HIGH_REF ↔ INI_X_LOW/INI_X_HIGH/INI_LOW_REF/INI_HIGH_REF.VAL; sSection → SECTION aller vier
- Referenzen: Y_OFFSET_LIT.VALUEO → CALIBRATE.Y_LOW; Y_SCALE_LIT.VALUEO → CALIBRATE.Y_HIGH
- Kalibrierter Wert: CALIBRATE.Y → AR_SPLIT_2.IN; OUT1 → AR_PUBLISH_CAL.IN; OUT2 → Q_NumericValue_CAL.rPhys
- CO-Trigger: Button_CO.IND → E_MERGE_CO.EI1; AX_SUBSCRIBE_CO.OUT → AX_RF_TRIG_CO.QI; AX_RF_TRIG_CO.ER → E_MERGE_CO.EI2, AX_RF_TRIG_CO.EF → E_MERGE_CO.EI3; E_MERGE_CO.EO → CALIBRATE.C_LOW
- CS-Trigger: analog mit Button_CS, AX_SUBSCRIBE_CS, AX_RF_TRIG_CS, E_MERGE_CS → CALIBRATE.C_HIGH
- Initialisierungskette: INIT → Q_NumericValue_RAW.INIT → Q_NumericValue_CAL.INIT → Y_OFFSET_LIT.INIT → Y_SCALE_LIT.INIT → CALIBRATE.INIT → INI_X_LOW.INIT → INI_X_HIGH.INIT → INI_LOW_REF.INIT → INI_HIGH_REF.INIT → AD_SUBSCRIBE_RAW.INIT → AR_PUBLISH_CAL.INIT → AX_SUBSCRIBE_CO.INIT → AX_SUBSCRIBE_CS.INIT → Button_CO.INIT → Button_CS.INIT; Button_CS.INITO → INITO und → AND_QO.REQ. Zusätzlich führt E_MERGE_QO (E_MERGE_3) Q_NumericValue_CAL.INITO sowie die CNF-Ereignisse von Q_NumericValue_RAW und Q_NumericValue_CAL zusammen; sein Ausgang EO triggert AND_QO.REQ, sodass QO bei jedem neuen Wert neu ausgewertet wird (ab v1.6; vorher war EO nicht angeschlossen). QI wird an alle inneren Bausteine verteilt; ihre QO laufen in der Reihenfolge der Kette auf AND_QO.IN1 bis AND_QO.IN15, AND_QO.OUT → QO. (Die bisherige Teilkette AD_SUBSCRIBE_RAW.INITO → AR_PUBLISH_CAL.INIT → AX_SUBSCRIBE_CO.INIT → AX_SUBSCRIBE_CS.INIT ist Teil dieser Kette.)

Technische Besonderheiten

- **Rohwert per Remote-Subscribe:** logiBUS_AI_IDA, AD_SPLIT_2 und AD_PUBLISH_RAW der lokalen Variante sind durch AD_SUBSCRIBE_1 ersetzt, der Rest der Kalibrier- und VT-Kette ist unverändert.
- **CO/CS ohne AX-Sockets:** Seit v1.1 sind C_LOW/C_HIGH von AR_CALIBRATE_2P_REF reine Ereigniseingänge; die E_MERGE_3-Bausteine verbinden direkt per Ereignisverbindung (v1.2: ein E_MERGE_3 statt zweier E_MERGE_2 in Reihe, funktional identisch).
- **QI an CALIBRATE:** Seit v1.3 hat AR_CALIBRATE_2P_REF ein INIT/QI/QO-Enable-Muster; ohne QI = TRUE bliebe der Baustein im Zustand START und REQ liefe nie. Das QI aller inneren Bausteine kommt jetzt vom SubApp-Eingang QI (Vorgabe TRUE) statt aus festen Parametern.
- **Bugfix v1.4 (Code-Review PR #260):** INI_X_HIGH.DEFAULT_VALUE stand auf REAL#1.0 statt 0.0; dadurch latchte das Kalibriert-Flag xHighCal bei jedem Boot sofort auf TRUE, obwohl HIGH nie kalibriert wurde. Betrifft Schnitthöhe, Öltemperatur und Druck.

- **Web-Trigger:** CO/CS werden von jeder Flanke (steigend oder fallend) ausgelöst, da der Web-Client bei jedem Klick den invertierten letzten Wert schreibt.
- Dieser Typ wird durch `logiBUS_AI_Calibrated_Subscribe_VT_OPC` (lokale Analog-Verarbeitung) abgelöst; die Anpassungen halten die Datei kompilierbar.

Anwendungsszenarien

- Kalibrierung eines Sensors, dessen Analogeingang auf einer Station ohne VT-Anschluss sitzt, mit VT-Bedienung auf einer anderen Station

⚖️ Vergleich mit ähnlichen Bausteinen

- `logiBUS_AI_Calibrate_2P_IDA_OPC`: Rohwert von einem lokalen `logiBUS_AI_IDA`
- `logiBUS_AI_Calibrate_3P_Remote_IDA_OPC`: Drei-Punkt-Variante mit Remote-Rohwert

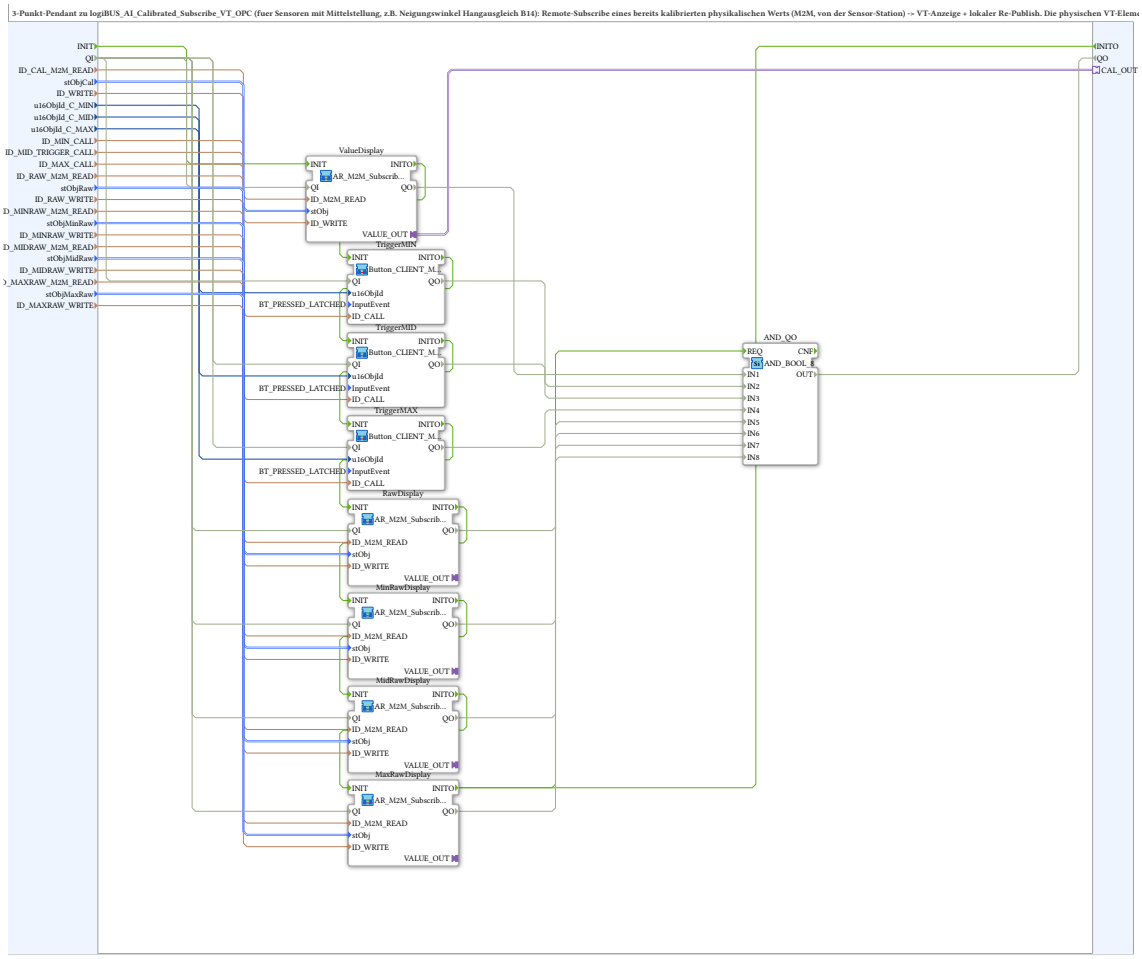
Zusammenfassung

`logiBUS_AI_Calibrate_Remote_IDA_OPC` kalibriert einen remote publizierten Analog-Rohwert in zwei Punkten, zeigt Roh- und kalibrierten Wert am VT an und veröffentlicht den kalibrierten Wert per OPC-UA.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

logiBUS_AI_Calibrated_Subscribe_VT_3P_OPC



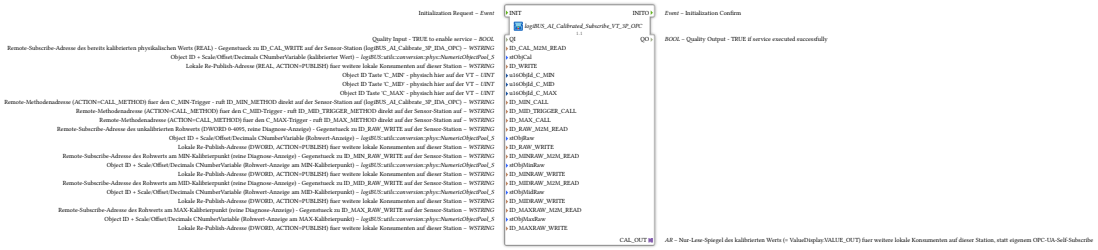
• – ○ • – ○ • – ○ •

Einleitung

`logiBUS_AI_Calibrated_Subscribe_VT_3P_OPC` ist das **3-Punkt-Pendant** zu `logiBUS_AI_Calibrated_Subscribe_VT_OPC` für Sensoren mit Mittelstellung (z. B. Neigungswinkel

Hangausgleich B14). Er abonniert per Remote-Subscribe (M2M, von der Sensor-Station) einen **bereits kalibrierten physikalischen Wert**, zeigt ihn auf der VT an und publiziert ihn lokal erneut.

Die physischen VT-Elemente bleiben dort, wo die VT ist: Die Tasten **C_MIN/C_MID/C_MAX** rufen per `CLIENT_0` direkt die Methoden `ID_MIN_METHOD/ID_MID_TRIGGER_METHOD/ID_MAX_METHOD` auf der Sensor-Station auf (`logiBUS_AI_Calibrate_3P_IDA_OPC`) – ein echter Event-zu-Event-Methodenaufruf, kein Toggle/Publish/Subscribe-Umweg (siehe `OPC_UA_VERNETZUNG.md`, Punkt 14). Die Referenzwert-Eingabefelder `MIN_REF/MID_REF/MAX_REF` bleiben bewusst außerhalb dieser SubApp (eigene `NumericValue_TO_CLIENT_1_0_OPC`-Instanzen je Aufrufer).



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialization Request

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialization Confirm

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input - TRUE, um den Dienst freizugeben

Kalibrierter Wert:

- **ID_CAL_M2M_READ** (`WSTRING`): Remote-Subscribe-Adresse des bereits kalibrierten physikalischen Werts (`REAL`); Gegenstück zu `ID_CAL_WRITE` auf der Sensor-Station (`logiBUS_AI_Calibrate_3P_IDA_OPC`)
- **stObjCal** (`NumericObjectPool_S`, Vorgabe `u16ObjId := ID_NULL, r32Scale := 1.0, i32Offset := 0, u8Decimals := 1`): Object-ID + Scale/Offset/Decimals der `CNumberVariable` (kalibrierter Wert)
- **ID_WRITE** (`WSTRING`): Lokale Re-Publish-Adresse (`REAL`, `ACTION=PUBLISH`) für weitere lokale Konsumenten auf dieser Station

Tasten C_MIN/C_MID/C_MAX:

- **u16ObjId_C_MIN**, **u16ObjId_C_MID**, **u16ObjId_C_MAX** (`UINT`, Vorgabe `ID_NULL`): Object-IDs der Tasten „C_MIN“, „C_MID“, „C_MAX“, physisch hier auf der VT
- **ID_MIN_CALL**, **ID_MID_TRIGGER_CALL**, **ID_MAX_CALL** (`WSTRING`): Remote-Methodenadressen (`ACTION=CALL_METHOD`); rufen `ID_MIN_METHOD`, `ID_MID_TRIGGER_METHOD` bzw. `ID_MAX_METHOD` direkt auf der Sensor-Station auf

Rohwert-Anzeigen (reine Diagnose):

- **ID_RAW_M2M_READ** (`WSTRING`): Remote-Subscribe-Adresse des unkalibrierten Rohwerts (`DWORD` 0-4095); Gegenstück zu `ID_RAW_WRITE` auf der Sensor-Station
- **stObjRaw** (`NumericObjectPool_S`, `u8Decimals := 0`): Anzeigefeld Rohwert
- **ID_RAW_WRITE** (`WSTRING`): Lokale Re-Publish-Adresse (`DWORD`, `ACTION=PUBLISH`)
- **ID_MINRAW_M2M_READ** (`WSTRING`): Remote-Subscribe-Adresse des Rohwerts am MIN-Kalibrierpunkt; Gegenstück zu `ID_MIN_RAW_WRITE` auf der Sensor-Station
- **stObjMinRaw** (`NumericObjectPool_S`, `u8Decimals := 0`): Anzeigefeld Rohwert am MIN-Kalibrierpunkt
- **ID_MINRAW_WRITE** (`WSTRING`): Lokale Re-Publish-Adresse (`DWORD`, `ACTION=PUBLISH`)
- **ID_MIDRAW_M2M_READ** (`WSTRING`): Remote-Subscribe-Adresse des Rohwerts am MID-Kalibrierpunkt; Gegenstück zu `ID_MID_RAW_WRITE` auf der Sensor-Station
- **stObjMidRaw** (`NumericObjectPool_S`, `u8Decimals := 0`): Anzeigefeld Rohwert am MID-Kalibrierpunkt
- **ID_MIDRAW_WRITE** (`WSTRING`): Lokale Re-Publish-Adresse (`DWORD`, `ACTION=PUBLISH`)

- **ID_MAXRAW_M2M_READ** (WSTRING): Remote-Subscribe-Adresse des Rohwerts am MAX-Kalibrierpunkt; Gegenstück zu ID_MAX_RAW_WRITE auf der Sensor-Station
- **stObjMaxRaw** (NumericObjectPool_S, u8Decimals := 0): Anzeigefeld Rohwert am MAX-Kalibrierpunkt
- **ID_MAXRAW_WRITE** (WSTRING): Lokale Re-Publish-Adresse (DWORD, ACTION=PUBLISH)

Daten-Ausgänge

- **QO** (BOOL): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde (UND der QO von ValueDisplay, TriggerMIN, TriggerMID, TriggerMAX, RawDisplay, MinRawDisplay, MidRawDisplay und MaxRawDisplay)

Adapter

- **CAL_OUT** (Plug, adapter::types::unidirectional::AR): Nur-Lese-Spiegel des kalibrierten Werts (ValueDisplay.VALUE_OUT) für weitere lokale Konsumenten auf dieser Station (z. B. Fahrwerk_Mittelpunkt_Anzeige für Bargraph/Status-Center), statt eines eigenen OPC-UA-Self-Subscribe

Verwendete Funktionsbausteine (FBs)

- **ValueDisplay**: SubApp MyLib::sys::AR_M2M_Subscribe_TO_Q_Publish_OPC – kalibrierter Wert
- **RawDisplay**, **MinRawDisplay**, **MidRawDisplay**, **MaxRawDisplay**: dieselbe SubApp für den Rohwert und die Rohwerte der drei Kalibrierpunkte
- **TriggerMIN**, **TriggerMID**, **TriggerMAX**: SubApp MyLib::sys::Button_CLIENT_Method_IE (InputEvent=BT_PRESSED_LATCHED) – Taste ⟶ Remote-Methodenaufruf (ID_CALL)
- **AND_QO**: iec61131::booleanOperators::AND_BOOL_8 – verknüpft die QO-Ausgänge der acht inneren SubApps per UND zum Baustein-Ausgang QO
- **Importe**: ID_NULL, BT_PRESSED_LATCHED, NumericObjectPool_S

Programmablauf und Verbindungen

- ID_CAL_M2M_READ, stObjCal, ID_WRITE ⟶ ValueDisplay (ID_M2M_READ, stObj, ID_WRITE); ValueDisplay.VALUE_OUT ⟶ CAL_OUT
- u16ObjId_C_MIN/ID_MIN_CALL ⟶ TriggerMIN (u16ObjId/ID_CALL); entsprechend C_MID ⟶ TriggerMID, C_MAX ⟶ TriggerMAX
- ID_RAW_M2M_READ, stObjRaw, ID_RAW_WRITE ⟶ RawDisplay
- ID_MINRAW_M2M_READ, stObjMinRaw, ID_MINRAW_WRITE ⟶ MinRawDisplay
- ID_MIDRAW_M2M_READ, stObjMidRaw, ID_MIDRAW_WRITE ⟶ MidRawDisplay
- ID_MAXRAW_M2M_READ, stObjMaxRaw, ID_MAXRAW_WRITE ⟶ MaxRawDisplay
- Initialisierungskette: INIT ⟶ ValueDisplay.INIT, ValueDisplay.INITO ⟶ TriggerMIN.INIT, TriggerMIN.INITO ⟶ TriggerMID.INIT, TriggerMID.INITO ⟶ TriggerMAX.INIT, TriggerMAX.INITO ⟶ RawDisplay.INIT, RawDisplay.INITO ⟶ MinRawDisplay.INIT, MinRawDisplay.INITO ⟶ MidRawDisplay.INIT, MidRawDisplay.INITO ⟶ MaxRawDisplay.INIT, MaxRawDisplay.INITO ⟶ INITO und ⟶ AND_QO.REQ
- QI ⟶ QI aller acht inneren SubApps
- ValueDisplay.QO, TriggerMIN.QO, TriggerMID.QO, TriggerMAX.QO, RawDisplay.QO, MinRawDisplay.QO, MidRawDisplay.QO, MaxRawDisplay.QO ⟶ AND_QO.IN1..IN8; AND_QO.OUT ⟶ QO

Technische Besonderheiten

- Die Kalibrierung selbst läuft auf der Sensor-Station; dieser Baustein enthält keine Kalibrierlogik
- Version 1.1 (2026-09-26): Plug CAL_OUT ergänzt – spiegelt den bereits remote abonnierten Wert für weitere Konsumenten auf STG1, kein zusätzliches Self-Subscribe nötig
- Version 1.0 (2026-09-26): 3-Punkt-Pendant zu logiBUS_AI_Calibrated_Subscribe_VT_OPC v1.8, für Hangausgleich-Winkel B14 (Kalibrierung zieht von STG1 nach STG3 um)

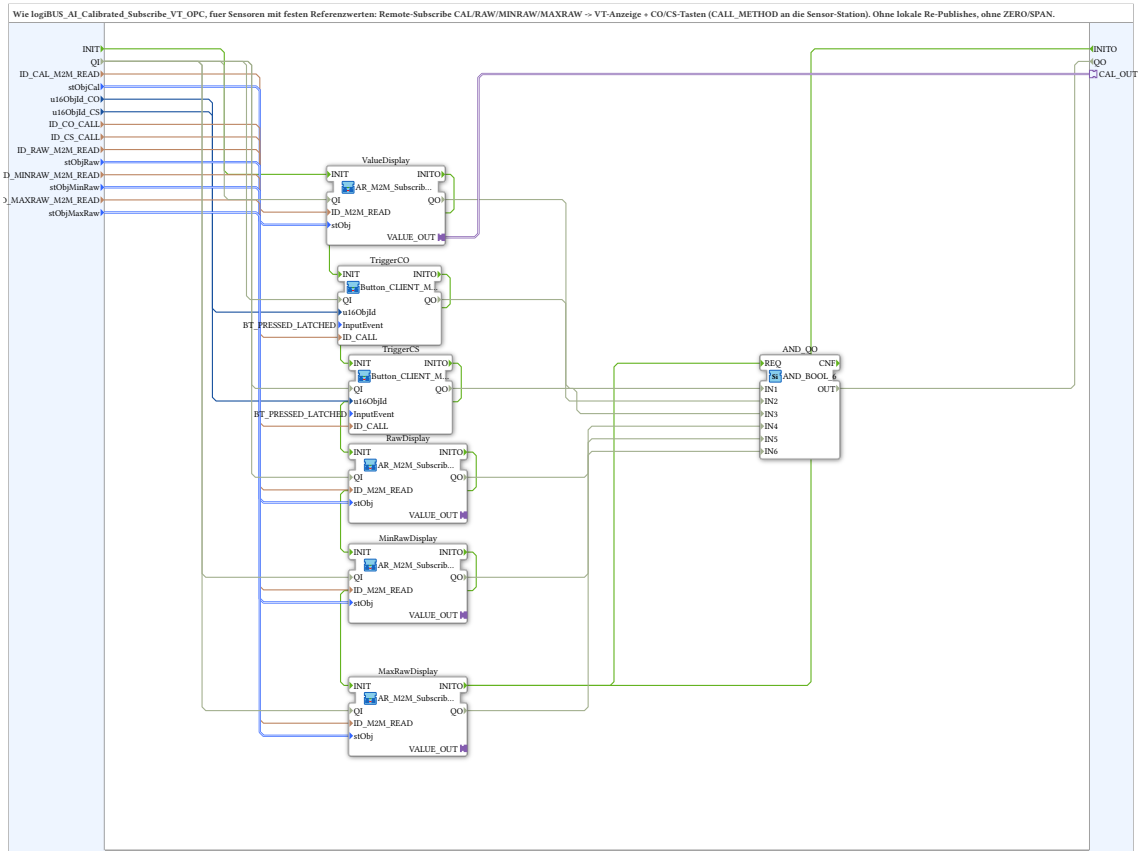
Anwendungsszenarien

- VT-Station, die den kalibrierten Neigungswinkel (B14) einer anderen Station anzeigt und die 3-Punkt-Kalibrierung (MIN/MID/MAX) per Taste auslöst

Fazit

`logiBUS_AI_Calibrated_Subscribe_VT_3P_OPC` zeigt remote kalibrierte Werte und Rohwerte der Kalibrierpunkte auf der VT an und leitet die Tasten C_MIN/C_MID/C_MAX per OPC-UA-Methodenaufruf an die Sensor-Station weiter.

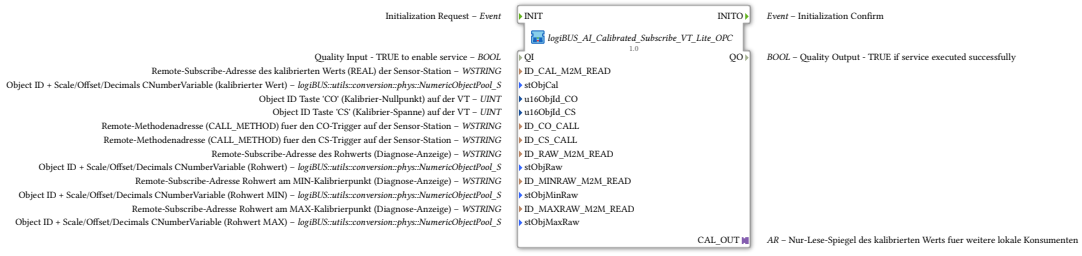
logiBUS_AI_Calibrated_Subscribe_VT_Lite_OPC



• — ○ • — ○ • — ○ •

Einleitung

`logiBUS_AI_Calibrated_Subscribe_VT_Lite_OPC` ist die schlanke Variante von `logiBUS_AI_Calibrated_Subscribe_VT_OPC` für Sensoren mit festen Referenzwerten. Per Remote-Subscribe werden kalibrierter Wert, Rohwert sowie die Rohwerte am MIN- und MAX-Kalibrierpunkt von der Sensor-Station abonniert und auf der VT angezeigt; die Tasten **CO**/**CS** rufen per `CALL_METHOD` die Kalibriermethoden auf der Sensor-Station auf. Gegenüber dem Original gibt es keine lokalen Re-Publishes (keine `ID_*_WRITE`-Spiegel) und keine `ZERO`/`SPAN`-Referenzwert-Eingabefelder (`ZeroRef`/`SpanRef` entfallen, die Referenzwerte sind fest auf der Sensor-Station). Statt `AR_M2M_Subscribe_TO_Q_Publish_OPC` kommt viermal `AR_M2M_Subscribe_TO_Q_OPC` zum Einsatz.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; startet die Kette `ValueDisplay` ⟶ `TriggerCO` ⟶ `TriggerCS` ⟶ `RawDisplay` ⟶ `MinRawDisplay` ⟶ `MaxRawDisplay`

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung; kommt von `MaxRawDisplay.INITO`

Daten-Eingänge

Variable	Typ	Vorgabe	Kommentar
`QI`	BOOL	TRUE	Quality Input: Freigabe des Dienstes; wird an alle sechs Instanzen weitergereicht.
`ID_CAL_M2M_READ`	WSTRING		Remote-Subscribe-Adresse des kalibrierten Werts (`REAL`) der Sensor-Station.
`stObjIdCal`	`tcutils::conversion::phys::NumericObjectID`	`(bQI := TRUE, sID := ID_NULL, r32Scale := 1.0, i32Offset := 0, u8Decimals := 1)`	Object ID plus Scale/Offset/Decimals der `CNumberVariable` (kalibrierter Wert).
`u16ObjId_CO`	UINT	`ID_NULL`	Object ID Taste „CO“ (Kalibrier-Nullpunkt) auf der VT.
`u16ObjId_CS`	UINT	`ID_NULL`	Object ID Taste „CS“ (Kalibrier-Spanne) auf der VT.
`ID_CO_CALL`	WSTRING		Remote-Methodenadresse (`CALL_METHOD`) für den CO-Trigger auf der Sensor-Station.
`ID_CS_CALL`	WSTRING		Remote-Methodenadresse (`CALL_METHOD`) für den CS-Trigger auf der Sensor-Station.
`ID_RAW_M2M_READ`	WSTRING		Remote-Subscribe-Adresse des Rohwerts (Diagnose-Anzeige).
`stObjIdRaw`	`tcutils::conversion::phys::NumericObjectID`	`(bQI := TRUE, sID := ID_NULL, r32Scale := 1.0, i32Offset := 0, u8Decimals := 0)`	Object ID plus Scale/Offset/Decimals der `CNumberVariable` (Rohwert).
`ID_MINRAW_M2M_READ`	WSTRING		Remote-Subscribe-Adresse Rohwert am MIN-Kalibrierpunkt (Diagnose-Anzeige).
`stObjIdMinRaw`	`tcutils::conversion::phys::NumericObjectID`	`(bQI := TRUE, sID := ID_NULL, r32Scale := 1.0, i32Offset := 0, u8Decimals := 0)`	Object ID plus Scale/Offset/Decimals der

			CNumberVariable (Rohwert MIN).
ID_MAXRAW_M2M_READ	WSTRING		Remote-Subscribe-Adresse Rohwert am MAX-Kalibrierpunkt (Diagnose-Anzeige).
stObjMaxRaw	mtUtils::conversion::phys::Numeric_u16ObjId_Bool_S	:= ID_NULL, r32Scale := 1.0, i32Offset := 0, u8Decimals := 0)	Object ID plus Scale/Offset/Decimals der CNumberVariable (Rohwert MAX).

Daten-Ausgänge

Variable	Typ	Vorgabe	Kommentar
QO	BOOL		Quality Output: AND der QO-Ausgänge aller sechs Instanzen (über AND_QO, AND_BOOL_6).

Adapter

Adapter	Typ	Richtung	Kommentar
CAL_OUT	adapter::types::unidirectional::AR	Plug	Nur-Lese-Spiegel des kalibrierten Werts (ValueDisplay.VALUE_OUT) für weitere lokale Konsumenten.

Verwendete Funktionsbausteine (FBs)

- **ValueDisplay** (SubApp MyLib::sys::AR_M2M_Subscribe_TO_Q_OPC): kalibrierter Wert
- **TriggerCO**, **TriggerCS** (SubApp MyLib::sys::Button_CLIENT_Method_IE, InputEvent = BT_PRESSED_LATCHED): Tasten CO und CS mit Methodenaufruf
- **RawDisplay**, **MinRawDisplay**, **MaxRawDisplay** (SubApp MyLib::sys::AR_M2M_Subscribe_TO_Q_OPC): Rohwert sowie Rohwerte der Kalibrierpunkte MIN und MAX
- **AND_QO** (iec61131::booleanOperators::AND_BOOL_6): verknüpft die sechs QO-Ausgänge

Programmablauf und Verbindungen

1. Wert: ID_CAL_M2M_READ -> ValueDisplay.ID_M2M_READ; stObjCal -> ValueDisplay.stObj; ValueDisplay.VALUE_OUT -> CAL_OUT.
2. Tasten: u16ObjId_CO -> TriggerCO.u16ObjId; ID_CO_CALL -> TriggerCO.ID_CALL; u16ObjId_CS -> TriggerCS.u16ObjId; ID_CS_CALL -> TriggerCS.ID_CALL.
3. Rohwert: ID_RAW_M2M_READ -> RawDisplay.ID_M2M_READ; stObjRaw -> RawDisplay.stObj.
4. MIN-Rohwert: ID_MINRAW_M2M_READ -> MinRawDisplay.ID_M2M_READ; stObjMinRaw -> MinRawDisplay.stObj.
5. MAX-Rohwert: ID_MAXRAW_M2M_READ -> MaxRawDisplay.ID_M2M_READ; stObjMaxRaw -> MaxRawDisplay.stObj.
6. Initialisierung: INIT -> ValueDisplay.INIT -> TriggerCO.INIT -> TriggerCS.INIT -> RawDisplay.INIT -> MinRawDisplay.INIT -> MaxRawDisplay.INIT (jeweils über INITO); MaxRawDisplay.INITO -> INITO und -> AND_QO.REQ.
7. Qualifier: QI -> .QI der sechs Instanzen.
8. Qualität: ValueDisplay.QO -> AND_QO.IN1; TriggerCO.QO -> IN2; TriggerCS.QO -> IN3; RawDisplay.QO -> IN4; MinRawDisplay.QO -> IN5; MaxRawDisplay.QO -> IN6; AND_QO.OUT -> QO.

Technische Besonderheiten

- Ohne lokale Re-Publishes: Es gibt keine ID_*_WRITE-Eingänge.
- Ohne ZERO/SPAN: Die Referenzwerte sind fest auf der Sensor-Station hinterlegt (z. B. Füllband-Winkel).
- Die Kalibrierung selbst läuft auf der Sensor-Station; dieser Baustein enthält keine Kalibrierlogik.

Anwendungsszenarien

- VT-Station, die kalibrierte Sensorwerte einer anderen Station anzeigt und deren Kalibrierung per Taste auslöst, wenn der Sensor feste Referenzwerte hat (z. B. Füllband-Winkel) und keine lokalen Re-Publishes benötigt werden.

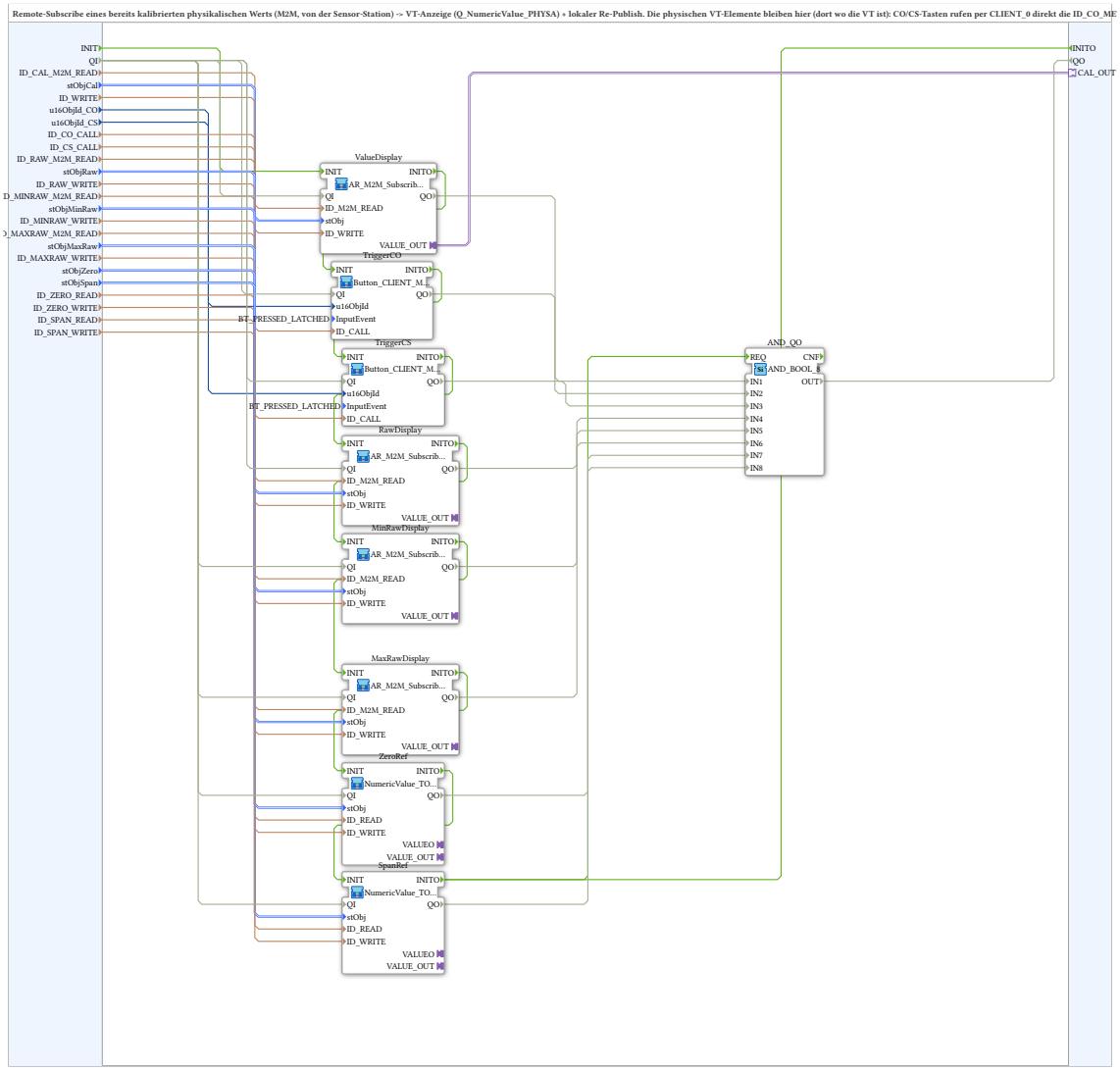
Zusammenfassung

`logiBUS_AI_Calibrated_Subscribe_VT_Lite_OPC` zeigt remote kalibrierte Werte und Rohwerte auf der VT an und leitet die Kalibriertasten per Methodenaufruf an die Sensor-Station weiter – ohne lokale Re-Publishes und ohne `ZERO`/`SPAN`.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

logiBUS_AI_Calibrated_Subscribe_VT_OPC



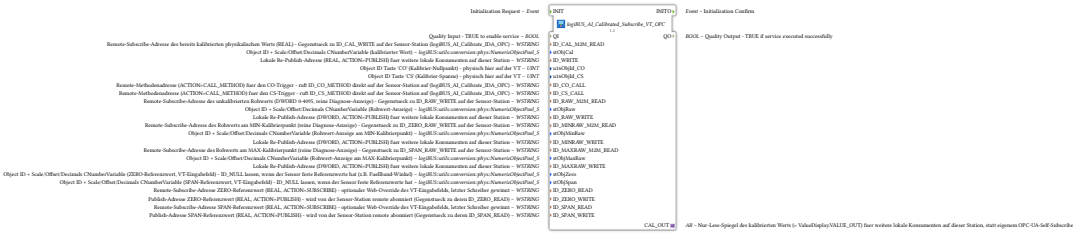
• – ○ • – ○ • – ○ •

Einleitung

`logiBUS_AI_Calibrated_Subscribe_VT_OPC` abonniert per Remote-Subscribe (M2M, von der Sensor-Station) einen **bereits kalibrierten physikalischen Wert** und zeigt ihn auf der VT an (`Q_NumericValue_PHYSA`) mit lokalem Re-Publish. Die physischen VT-Elemente bleiben dort, wo die VT ist: Die Tasten **CO**/**CS** rufen per `CLIENT_0` direkt die Methoden `ID_CO_METHOD`/`ID_CS_METHOD` auf der

Sensor-Station auf (`logiBUS_AI_Calibrate_IDA_OPC`) – ein echter Event-zu-Event-Methodenaufruf, kein Toggle/Publish/Subscribe-Umweg (siehe `OPC_UA_VERNETZUNG.md`, Punkt 14).

Der Baustein ersetzt `logiBUS_AI_Calibrate_Remote_IDA_OPC`: Die Kalibrierung selbst läuft jetzt auf der Sensor-Station, hier bleiben nur Anzeige und Weiterleitung der VT-Bedienelemente. Die `ZERO`/`SPAN`-Referenzwert-Eingabefelder (`stObjZero`/`stObjSpan`) sind optional: `ID_NULL` lassen, wenn der Sensor feste Referenzwerte hat (Füllband-Winkel: 0/100 als Literal auf der Sensor-Station), echte Werte durchreichen, wenn der Bediener sie selbst eingibt (Druck: 0/250 bar, kein Kalibrator immer griffbereit).



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; startet die Kette `ValueDisplay` ⟶ `TriggerCO` ⟶ `TriggerCS` ⟶ `RawDisplay` ⟶ `MinRawDisplay` ⟶ `MaxRawDisplay` ⟶ `ZeroRef` ⟶ `SpanRef`

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung; kommt von `SpanRef.INITO`

Daten-Eingänge

Freigabe:

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input, Freigabe des Dienstes; wird an alle acht Instanzen (`ValueDisplay`, `TriggerCO`, `TriggerCS`, `RawDisplay`, `MinRawDisplay`, `MaxRawDisplay`, `ZeroRef`, `SpanRef`) weitergereicht

Kalibrierter Wert:

- **ID_CAL_M2M_READ** (`WSTRING`): Remote-Subscribe-Adresse des bereits kalibrierten physikalischen Werts (`REAL`); Gegenstück zu `ID_CAL_WRITE` auf der Sensor-Station
- **stObjCal** (`NumericObjectPool_S`): Object-ID + Scale/Offset/Decimals der `CNumberVariable` (kalibrierter Wert)
- **ID_WRITE** (`WSTRING`): Lokale Re-Publish-Adresse (`REAL`, `ACTION=PUBLISH`) für weitere lokale Konsumenten

Tasten CO/CS:

- **u16ObjId_CO** (`UINT`, Vorgabe `ID_NULL`): Object-ID der Taste „CO“ (Kalibrier-Nullpunkt), physisch hier auf der VT
- **u16ObjId_CS** (`UINT`, Vorgabe `ID_NULL`): Object-ID der Taste „CS“ (Kalibrier-Spanne), physisch hier auf der VT
- **ID_CO_CALL**, **ID_CS_CALL** (`WSTRING`): Remote-Methodenadressen (`ACTION=CALL_METHOD`); rufen `ID_CO_METHOD`/`ID_CS_METHOD` direkt auf der Sensor-Station auf

Rohwert-Anzeigen (reine Diagnose):

- **ID_RAW_M2M_READ** (`WSTRING`): Remote-Subscribe-Adresse des unkalibrierten Rohwerts (`DWORD` 0-4095); Gegenstück zu `ID_RAW_WRITE` auf der Sensor-Station
- **stObjRaw** (`NumericObjectPool_S`, `u8Decimals := 0`): Anzeigefeld Rohwert
- **ID_RAW_WRITE** (`WSTRING`): Lokale Re-Publish-Adresse (`DWORD`, `ACTION=PUBLISH`)
- **ID_MINRAW_M2M_READ** (`WSTRING`): Remote-Subscribe-Adresse des Rohwerts am MIN-Kalibrierpunkt; Gegenstück zu `ID_ZERO_RAW_WRITE` auf der Sensor-Station
- **stObjMinRaw** (`NumericObjectPool_S`, `u8Decimals := 0`): Anzeigefeld Rohwert am MIN-Kalibrierpunkt
- **ID_MINRAW_WRITE** (`WSTRING`): Lokale Re-Publish-Adresse (`DWORD`, `ACTION=PUBLISH`)
- **ID_MAXRAW_M2M_READ** (`WSTRING`): Remote-Subscribe-Adresse des Rohwerts am MAX-Kalibrierpunkt; Gegenstück zu `ID_SPAN_RAW_WRITE` auf der Sensor-Station

- **stObjMaxRaw** (`NumericObjectPool_S`, `u8Decimals := 0`): Anzeigefeld Rohwert am MAX-Kalibrierpunkt
- **ID_MAXRAW_WRITE** (`WSTRING`): Lokale Re-Publish-Adresse (`DWORD`, `ACTION=PUBLISH`)

Referenzwerte (optional):

- **stObjZero**, **stObjSpan** (`NumericObjectPool_S`): VT-Eingabefelder für den ZERO-/SPAN-Referenzwert; `ID_NULL` lassen, wenn der Sensor feste Referenzwerte hat
- **ID_ZERO_READ**, **ID_SPAN_READ** (`WSTRING`): Remote-Subscribe-Adressen (`REAL`, `ACTION=SUBSCRIBE`) – optionaler Web-Override des VT-Eingabefelds, letzter Schreiber gewinnt
- **ID_ZERO_WRITE**, **ID_SPAN_WRITE** (`WSTRING`): Publish-Adressen (`REAL`, `ACTION=PUBLISH`), werden von der Sensor-Station remote abonniert (Gegenstück zu deren `ID_ZERO_READ`/`ID_SPAN_READ`)

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output, `AND` der `QO`-Ausgänge aller acht Instanzen (über `AND_QO`, `AND_BOOL_8`)

Adapter

- **CAL_OUT** (Plug, `adapter::types::unidirectional::AR`): Nur-Lese-Spiegel des kalibrierten Werts (`ValueDisplay.VALUE_OUT`) für weitere lokale Konsumenten auf dieser Station, statt eines eigenen OPC-UA-Self-Subscribe

Verwendete Funktionsbausteine (FBs)

- **ValueDisplay**: SubApp `MyLib::sys::AR_M2M_Subscribe_TO_Q_Publish_OPC` – kalibrierter Wert
- **RawDisplay**, **MinRawDisplay**, **MaxRawDisplay**: dieselbe SubApp für Rohwert und Rohwerte der Kalibrierpunkte
- **TriggerCO**, **TriggerCS**: SubApp `MyLib::sys::Button_CLIENT_Method_IE` (`InputEvent=BT_PRESSED_LATCHED`)
- **ZeroRef**, **SpanRef**: SubApp `MyLib::sys::NumericValue_TO_AR2_OPC` – Referenzwert-Eingabefelder
- **AND_QO**: `iec61131::booleanOperators::AND_BOOL_8` – verknüpft die acht `QO`-Ausgänge zum SubApp-Ausgang `QO`

Programmablauf und Verbindungen

- `ID_CAL_M2M_READ`, `stObjCal`, `ID_WRITE` ⟶ `ValueDisplay`; `ValueDisplay.VALUE_OUT` ⟶ `CAL_OUT`
- `u16ObjId_CO`/`ID_CO_CALL` ⟶ `TriggerCO`; `u16ObjId_CS`/`ID_CS_CALL` ⟶ `TriggerCS`
- `ID_RAW_M2M_READ`, `stObjRaw`, `ID_RAW_WRITE` ⟶ `RawDisplay`
- `ID_MINRAW_M2M_READ`, `stObjMinRaw`, `ID_MINRAW_WRITE` ⟶ `MinRawDisplay`
- `ID_MAXRAW_M2M_READ`, `stObjMaxRaw`, `ID_MAXRAW_WRITE` ⟶ `MaxRawDisplay`
- `stObjZero`, `ID_ZERO_READ`, `ID_ZERO_WRITE` ⟶ `ZeroRef`
- `stObjSpan`, `ID_SPAN_READ`, `ID_SPAN_WRITE` ⟶ `SpanRef`
- Initialisierung: `INIT` ⟶ `ValueDisplay.INIT` ⟶ `TriggerCO.INIT` ⟶ `TriggerCS.INIT` ⟶ `RawDisplay.INIT` ⟶ `MinRawDisplay.INIT` ⟶ `MaxRawDisplay.INIT` ⟶ `ZeroRef.INIT` ⟶ `SpanRef.INIT` (jeweils über `INITO`); `SpanRef.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`
- Qualität: `QI` ⟶ `.QI` der acht Instanzen; `ValueDisplay.QO` ⟶ `AND_QO.IN1`, `TriggerCO.QO` ⟶ `IN2`, `TriggerCS.QO` ⟶ `IN3`, `RawDisplay.QO` ⟶ `IN4`, `MinRawDisplay.QO` ⟶ `IN5`, `MaxRawDisplay.QO` ⟶ `IN6`, `ZeroRef.QO` ⟶ `IN7`, `SpanRef.QO` ⟶ `IN8`; `AND_QO.OUT` ⟶ `QO`

Technische Besonderheiten

- Der Button-+-`CLIENT_0`-Aufruf kam hier früher zweimal identisch (CO und CS) vor und wurde in `Button_CLIENT_Method_IE` ausgelagert.
- Die Kalibrierung selbst läuft auf der Sensor-Station; dieser Baustein enthält keine Kalibrierlogik.

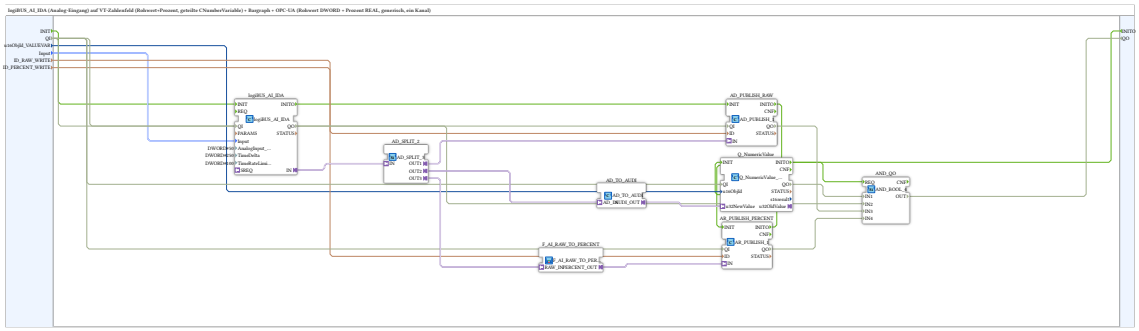
Anwendungsszenarien

- VT-Station, die kalibrierte Sensorwerte einer anderen Station anzeigt und deren Kalibrierung per Taste auslöst (z. B. Füllband-Winkel, Druck)

Fazit

`logiBUS_AI_Calibrated_Subscribe_VT_OPC` zeigt remote kalibrierte Werte und Rohwerte auf der VT an und leitet die Kalibriertasten per OPC-UA-Methodenaufruf an die Sensor-Station weiter.

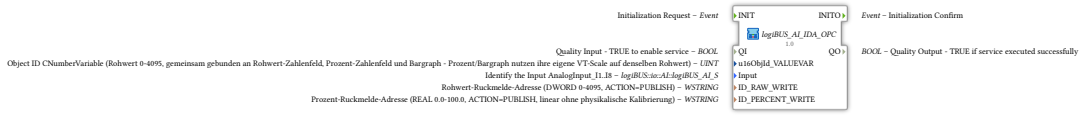
logiBUS_AI_IDA_OPC



• – ○ • – ○ • – ○ •

Einleitung

`logiBUS_AI_IDA_OPC` bindet einen physischen Analogeingang (`logiBUS_AI_IDA`, Rohwert 0-4095, siehe logiBUS_AI Hardware-Dokumentation) gleichzeitig an ein VT-Zahlenfeld (Rohwert), ein zweites VT-Zahlenfeld/Bargraph (Prozent) und an OPC-UA an: Rohwert (DWORD) und linear umgerechneter Prozentwert (REAL) werden beide separat per OPC-UA publiziert. Die Prozent-Umrechnung erfolgt über `F_AI_RAW_TO_PERCENT_AD`.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: logiBUS_AI_IDA_OPC

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **logiBUS_AI_IDA**: `logiBUS::io::AI::logiBUS_AI_IDA` — physischer Analogeingang, Rohwert 0-4095 (12 Bit), `AnalogInput_hysteresis=DWORD#0`, `TimeDelta=DWORD#250`, `TimeRateLimit=DWORD#100`, `QI` kommt vom Eingang `QI`.
  - **AD_SPLIT_2** (Typ `AD_SPLIT_3`): `adapter::events::unidirectional::AD_SPLIT_3` — verzweigt den DWORD-Adapterwert auf drei unabhängige Ausgänge (`OUT1..OUT3`).
  - **AD_PUBLISH_RAW** (Typ `AD_PUBLISH_1`): `adapter::net::AD_PUBLISH_1` — publiziert den Rohwert (DWORD) per OPC-UA, Zieladresse `ID_RAW_WRITE`, `QI` kommt vom Eingang `QI`.
  - **Q_NumericValue** (Typ `Q_NumericValue_AUDI`): `isobus::UT::Q::Q_NumericValue_AUDI` — schreibt den Rohwert in das VT-Zahlenfeld `u16ObjId_VALUEVAR`.
  - **F_AI_RAW_TO_PERCENT** (SubApp, Typ `F_AI_RAW_TO_PERCENT_AD`): `MyLib::sys::F_AI_RAW_TO_PERCENT_AD` — rechnet den Rohwert linear in Prozent um (siehe F_AI_RAW_TO_PERCENT / F_AI_RAW_TO_PERCENT_AD).
  - **AR_PUBLISH_PERCENT** (Typ `AR_PUBLISH_1`): `adapter::net::AR_PUBLISH_1` — publiziert den Prozentwert (REAL) per OPC-UA, Zieladresse `ID_PERCENT_WRITE`, `QI` kommt vom Eingang `QI`.
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_4` — verknüpft die `QO`-Ausgänge von `Q_NumericValue`, `logiBUS_AI_IDA`, `AD_PUBLISH_RAW` und `AR_PUBLISH_PERCENT` zum Ausgang `QO`.
  - **AD_TO_AUDI**: `adapter::conversion::unidirectional::AD_TO_AUDI` — wandelt den DWORD-Rohwert in UDINT für das VT-Zahlenfeld.
- **Funktionsweise**: Der Rohwert wird über `AD_SPLIT_2` verdreifacht: ein Zweig geht direkt an die OPC-UA-Publikation des Rohwerts, ein zweiter über `AD_TO_AUDI` an das VT-Rohwert-Zahlenfeld, ein dritter über `F_AI_RAW_TO_PERCENT_AD` zur Prozent-Umrechnung und anschließenden OPC-UA-Publikation.

Programmablauf und Verbindungen

1. `Input` ⟶ `logiBUS_AI_IDA.Input`; `u16ObjId_VALUEVAR` ⟶ `Q_NumericValue.u16ObjId`; `ID_RAW_WRITE` ⟶ `AD_PUBLISH_RAW.ID`; `ID_PERCENT_WRITE` ⟶ `AR_PUBLISH_PERCENT.ID`.
2. `logiBUS_AI_IDA.IN` (Adapter) ⟶ `AD_SPLIT_2.IN` (Adapter).
3. `AD_SPLIT_2.OUT1` ⟶ `AD_PUBLISH_RAW.IN` (Rohwert-OPC-UA-Zweig).

4. `AD_SPLIT_2.OUT2` ⟶ `AD_TO_AUDI.AD_IN`; `AD_TO_AUDI.AUDI_OUT` ⟶ `Q_NumericValue.u32NewValue` (VT-Rohwert-Zweig).
5. `AD_SPLIT_2.OUT3` ⟶ `F_AI_RAW_TO_PERCENT.RAW_IN`; `F_AI_RAW_TO_PERCENT.PERCENT_OUT` ⟶ `AR_PUBLISH_PERCENT.IN` (Prozent-OPC-UA-Zweig).
6. `QI` (`BOOL`, Vorgabe `TRUE`) ⟶ `Q_NumericValue.QI`, `logiBUS_AI_IDA.QI`, `AD_PUBLISH_RAW.QI` und `AR_PUBLISH_PERCENT.QI`; deren `QO`-Ausgänge ⟶ `AND_QO.IN1` (`Q_NumericValue`), `IN2` (`logiBUS_AI_IDA`), `IN3` (`AD_PUBLISH_RAW`), `IN4` (`AR_PUBLISH_PERCENT`); `AND_QO.OUT` ⟶ `QO` (`BOOL`, Quality Output – `TRUE`, wenn alle vier Dienste erfolgreich waren).
7. Initialisierungskette (Ereignis-Eingang `INIT` / Ereignis-Ausgang `INITO`, jeweils `EInit`): `INIT` ⟶ `logiBUS_AI_IDA.INIT` ⟶ `AD_PUBLISH_RAW.INIT` ⟶ `AR_PUBLISH_PERCENT.INIT` ⟶ `Q_NumericValue.INIT` (jeweils `INITO` ⟶ nächstes `INIT`); `Q_NumericValue.INITO` ⟶ `INITO` und `AND_QO.REQ`.

Technische Besonderheiten

- Rohwert-Bereich 0-4095 (12 Bit) — siehe hierzu die verifizierte AI-Hardwaredokumentation.
- Die Prozent-Umrechnung ist rein linear und unkalibriert (kein physikalisches Scaling), wie im Kommentar der SubApp-Definition ausdrücklich vermerkt.
- Drei unabhängige Konsumenten (zwei VT-Anzeigen + zwei OPC-UA-Publikationen) laufen parallel aus derselben `AD_SPLIT_3`-Verzweigung, wodurch alle Werte synchron denselben Erfassungszyklus abbilden.

Anwendungsszenarien

- Analoger Prozesswert (z. B. Füllstand, Druck), der sowohl lokal am VT (Rohwert und Prozent) angezeigt als auch von einer übergeordneten Leitwarte per OPC-UA in beiden Darstellungen (roh und Prozent) gelesen werden soll.

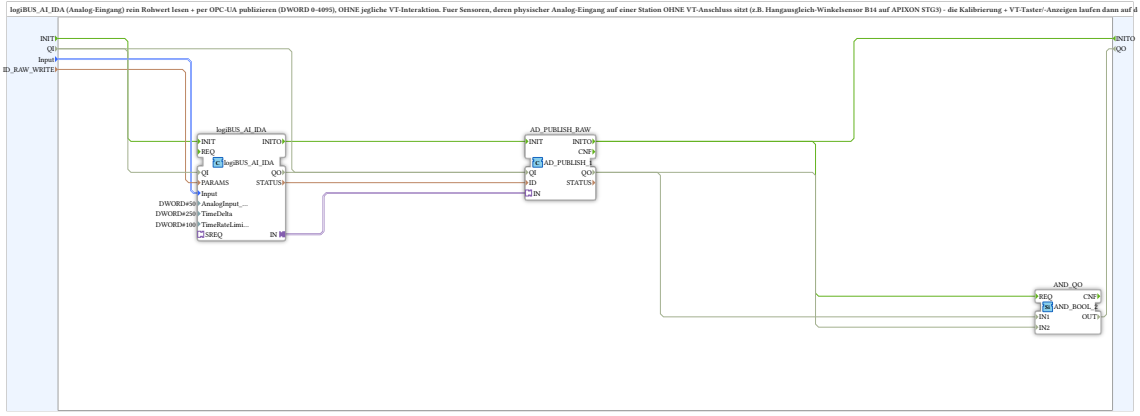
Zusammenfassung

Vollständige VT- und OPC-UA-Anbindung eines physischen Analogeingangs mit paralleler Rohwert- und Prozentdarstellung, aufgebaut aus Standard-Adapterbausteinen und der wiederverwendbaren Prozent-Umrechnung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

logiBUS_AI_Raw_Publish_OPC

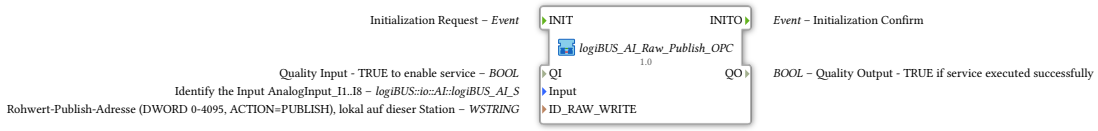


• – ○ • – ○ • – ○ •

Einleitung

`logiBUS_AI_Raw_Publish_OPC` liest einen Analog-Eingang (`logiBUS_AI_IDA`) als Rohwert (DWORD, 0-4095) und publiziert ihn per OPC-UA, **ohne jegliche VT-Interaktion**. Er ist für Sensoren gedacht, deren physischer Analog-Eingang auf einer Station ohne VT-Anschluss sitzt (z. B. Hangausgleich-Winkelsensor B14 auf APIXON STG3). Die Kalibrierung sowie VT-Taster und -Anzeigen laufen dann

auf der Station mit VT-Anschluss und werden per Remote-Subscribe dieses Rohwerts gespeist (siehe logiBUS_AI_Calibrate_3P_Remote_IDA_OPC).



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; geht an `logiBUS_AI_IDA.INIT`

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierung abgeschlossen; kommt von `AD_PUBLISH_RAW.INITO`

Daten-Eingänge

- **QI** (BOOL): Quality Input - TRUE aktiviert den Dienst (Default `TRUE`); geht an `logiBUS_AI_IDA.QI` und `AD_PUBLISH_RAW.QI`
- **Input** (`logiBUS::io::AI::logiBUS_AI_S`): Auswahl des Analog-Eingangs `AnalogInput_I1..I8` (Default `logiBUS_AI::Invalid`)
- **ID_RAW_WRITE** (WSTRING): Publish-Adresse des Rohwerts (DWORD 0-4095, `ACTION=PUBLISH`), lokal auf dieser Station

Daten-Ausgänge

- **QO** (BOOL): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde (AND aus `logiBUS_AI_IDA.QO` und `AD_PUBLISH_RAW.QO`)

Adapter

Es werden keine Adapter verwendet.

Funktionsweise

Das Netzwerk besteht aus drei Bausteinen (zwei Funktionsbausteine plus die Verknüpfung `AND_QO`):

- `logiBUS_AI_IDA` (`logiBUS::io::AI::logiBUS_AI_IDA`, `QI` von außen): liest den gewählten Analog-Eingang (`Input` ⟶ `logiBUS_AI_IDA.Input`). Parametriert sind `AnalogInput_hysteresis = DWORD#50`, `TimeDelta = DWORD#250` und `TimeRateLimit = DWORD#100`.
- `AD_PUBLISH_RAW` (`adapter::net::AD_PUBLISH_1`, `QI` von außen): `logiBUS_AI_IDA.IN` ⟶ `AD_PUBLISH_RAW.IN`; publiziert den Rohwert an die Adresse `ID_RAW_WRITE` (`ID_RAW_WRITE` ⟶ `AD_PUBLISH_RAW.ID`).
- `AND_QO` (`iec61131::booleanOperators::AND_BOOL_2`): verknüpft `logiBUS_AI_IDA.QO` (`IN1`) und `AD_PUBLISH_RAW.QO` (`IN2`) zum Ausgang `QO`; `REQ` kommt von `AD_PUBLISH_RAW.INITO`.

Initialisierung: `INIT` ⟶ `logiBUS_AI_IDA.INIT`, `logiBUS_AI_IDA.INITO` ⟶ `AD_PUBLISH_RAW.INIT`, `AD_PUBLISH_RAW.INITO` ⟶ `INITO` (und ⟶ `AND_QO.REQ`).

Technische Besonderheiten

- Rohwert-Hälfte von `logiBUS_AI_Calibrate_3P_IDA_OPC`: Die Kalibrierung findet nicht hier, sondern auf der Station mit VT-Anschluss statt
- Keine weiteren Ports außer `INIT`/`INITO`, `QI`/`QO` und den beiden Eingangsvariablen `Input` und `ID_RAW_WRITE`; keine Adapter
- Das Verhalten des Rohwert-Eingangs (Hysterese, `TimeDelta`, `TimeRateLimit`) wird über die Parameter von `logiBUS_AI_IDA` festgelegt

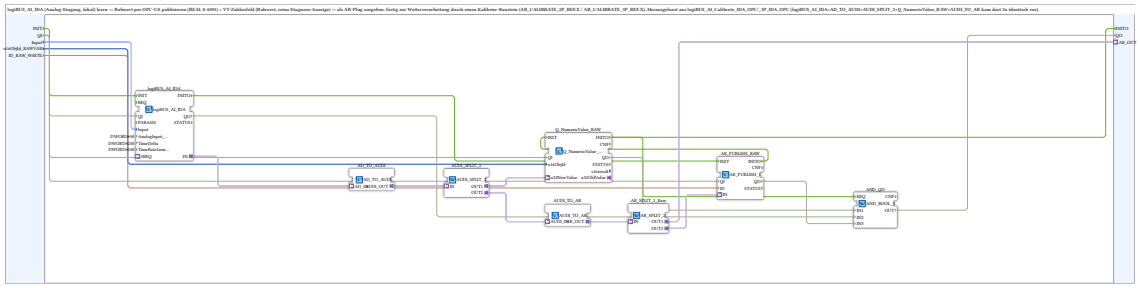
Anwendungsszenarien

- Sensoren an Stationen ohne VT-Anschluss, deren Kalibrierung und Anzeige auf einer anderen Station laufen

Fazit

`logiBUS_AI_Raw_Publish_OPC` stellt den Rohwert eines Analog-Eingangs per OPC-UA bereit und ist damit die Sensor-Seite einer stationsübergreifenden Kalibrierung.

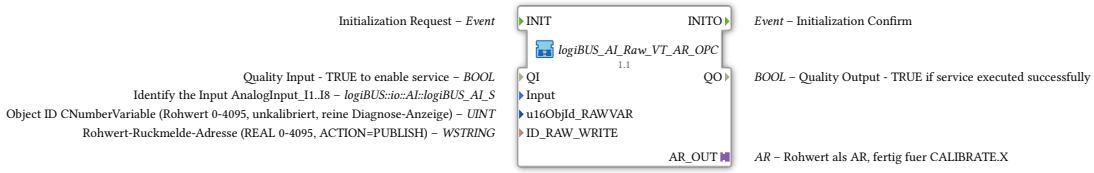
logiBUS_AI_Raw_VT_AR_OPC



• – ○ • – ○ • – ○ •

Einleitung

`logiBUS_AI_Raw_VT_AR_OPC` liest einen lokalen logiBUS-Analogeingang (`logiBUS_AI_IDA`), publiziert den Rohwert per OPC-UA (`REAL`, 0..4095) und zeigt ihn in einem VT-Zahlenfeld an (Rohwert, reine Diagnose-Anzeige). Zusätzlich wird er als `AR`-Plug ausgegeben, fertig zur Weiterverarbeitung durch einen Kalibrier-Baustein (`AR_CALIBRATE_2P_REF.X` bzw. `AR_CALIBRATE_3P_REF.X`). Der Baustein wurde aus `logiBUS_AI_Calibrate_IDA_OPC` und `logiBUS_AI_Calibrate_3P_IDA_OPC` herausgelöst, wo die Kombination `logiBUS_AI_IDA + AD_TO_AUDI + AUDI_SPLIT_2 + Q_NumericValue_RAW + AUDI_TO_AR` zweimal identisch vorkam. Verwandt: `logiBUS_AI_Raw_Publish_OPC`.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung (von `Q_NumericValue_RAW.INITO`)

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input – `TRUE` aktiviert die Dienste (→ `Q_NumericValue_RAW.QI`, `logiBUS_AI_IDA.QI` und `AR_PUBLISH_RAW.QI`)
- **Input** (`logiBUS::io::AI::logiBUS_AI_S`): Auswahl des Eingangs `AnalogInput_I1..I8` (Default `logiBUS_AI::Invalid`)
- **u16ObjId_RAWVAR** (`UINT`): Object ID der `CNumberVariable` (Rohwert 0..4095, unkalibriert, reine Diagnose-Anzeige)
- **ID_RAW_WRITE** (`WSTRING`): Rohwert-Rückmelde-Adresse (`REAL` 0..4095, `ACTION=PUBLISH`)

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output – `TRUE`, wenn alle Dienste erfolgreich waren (UND aus `Q_NumericValue_RAW.QO`, `logiBUS_AI_IDA.QO` und `AR_PUBLISH_RAW.QO`)

Adapter

- **AR_OUT** (Plug, `adapter::types::unidirectional::AR`): Rohwert als `AR`, fertig für `CALIBRATE.X`

Verwendete Funktionsbausteine (FBs)

- **`logiBUS_AI_IDA`** (`logiBUS::io::AI::logiBUS_AI_IDA`): Parameter `AnalogInput_hysteresis = DWORD#50`, `TimeDelta = DWORD#250`, `TimeRateLimit = DWORD#100`
- **`AD_TO_AUDI`** (`adapter::conversion::unidirectional::AD_TO_AUDI`)
- **`AUDI_SPLIT_2`** (`adapter::events::unidirectional::AUDI_SPLIT_2`)
- **`Q_NumericValue_RAW`** (`isobus::UT::Q::Q_NumericValue_AUDI`): VT-Rohwertanzeige
- **`AUDI_TO_AR`** (`adapter::conversion::unidirectional::AUDI_TO_AR`)

- **AR_SPLIT_2_Raw** (adapter::events::unidirectional::AR_SPLIT_2)
- **AR_PUBLISH_RAW** (adapter::net::AR_PUBLISH_1): OPC-UA-Publish
- **AND_QO** (iec61131::booleanOperators::AND_BOOL_3): verknüpft die QO-Ausgänge von Q_NumericValue_RAW, logiBUS_AI_IDA und AR_PUBLISH_RAW

Funktionsweise

- Input ⟶ logiBUS_AI_IDA.Input
- logiBUS_AI_IDA.IN ⟶ AD_TO_AUDI.AD_IN ⟶ AUDI_SPLIT_2.IN
- AUDI_SPLIT_2.OUT1 ⟶ Q_NumericValue_RAW.u32NewValue (VT-Anzeige); u16ObjId_RAWVAR ⟶ Q_NumericValue_RAW.u16ObjId
- INIT ⟶ logiBUS_AI_IDA.INIT; logiBUS_AI_IDA.INITO ⟶ AR_PUBLISH_RAW.INIT; AR_PUBLISH_RAW.INITO ⟶ Q_NumericValue_RAW.INIT; Q_NumericValue_RAW.INITO ⟶ INITO (und ⟶ AND_QO.REQ)
- QI ⟶ Q_NumericValue_RAW.QI, logiBUS_AI_IDA.QI und AR_PUBLISH_RAW.QI
- Q_NumericValue_RAW.QO ⟶ AND_QO.IN1; logiBUS_AI_IDA.QO ⟶ AND_QO.IN2; AR_PUBLISH_RAW.QO ⟶ AND_QO.IN3; AND_QO.OUT ⟶ QO
- AUDI_SPLIT_2.OUT2 ⟶ AUDI_TO_AR.AUDI_IN ⟶ AR_SPLIT_2_Raw.IN
- AR_SPLIT_2_Raw.OUT1 ⟶ AR_OUT
- AR_SPLIT_2_Raw.OUT2 ⟶ AR_PUBLISH_RAW.IN; ID_RAW_WRITE ⟶ AR_PUBLISH_RAW.ID

Technische Besonderheiten

- Der Rohwert ist unkalibriert (0..4095); die Kalibrierung erfolgt im nachgeschalteten Baustein
- Zwei Splitter (AUDI_SPLIT_2, AR_SPLIT_2_Raw), da ein Adapter nicht auf mehrere Ziele verzweigen darf
- Eingangsparameter von logiBUS_AI_IDA: Hysterese 50, TimeDelta 250, TimeRateLimit 100

Anwendungsszenarien

- Rohwert eines Analogeingangs anzeigen, per OPC-UA veröffentlichen und einem Kalibrier-Baustein zuführen

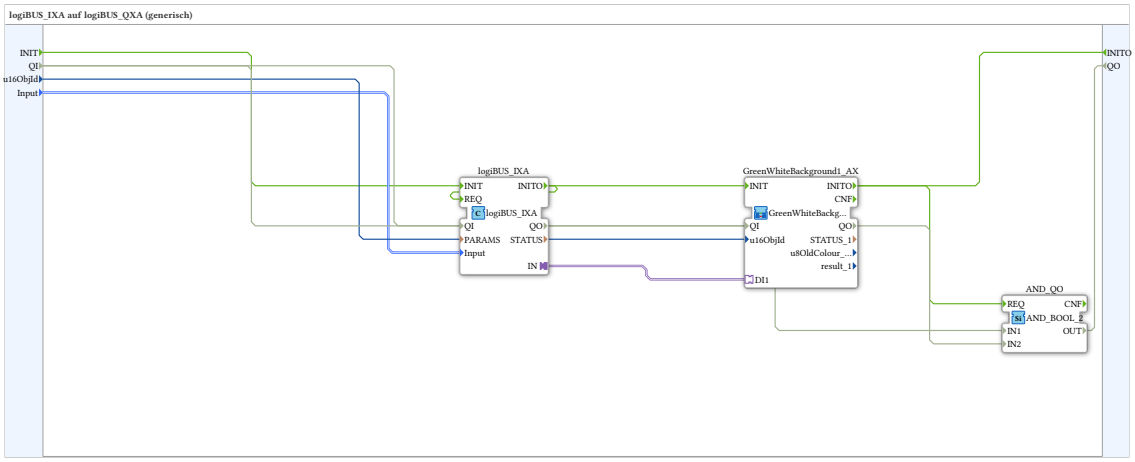
Zusammenfassung

logiBUS_AI_Raw_VT_AR_OPC liefert den Analog-Rohwert gleichzeitig an VT, OPC-UA und einen AR-Ausgang.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

logiBUS_IXA_BG

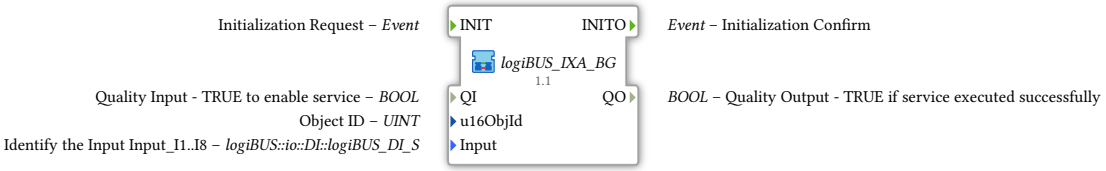


• – ○ • – ○ • – ○ •

Einleitung

logiBUS_IXA_BG verbindet einen physischen digitalen Eingang (logiBUS_IXA) mit einer VT-Statusanzeige über die Hintergrundfarbfamilie GreenWhiteBackground1_AX (siehe Background-Farbbausteine (gemein-

sames Muster)) — der aktuelle Zustand des Eingangs wird als grün/weißer Hintergrund auf dem VT sichtbar gemacht, ohne OPC-UA-Anbindung. Für die Variante mit zusätzlicher OPC-UA-Rückmeldung siehe `logiBUS_IXA_BG_OPC`.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: logiBUS_IXA_BG

- **Typ**: SubAppType
- **Schnittstelle (zusätzlich)**: Ereignis-Eingang `INIT`, Ereignis-Ausgang `INITO` (je `EInit`), Daten-Eingang `QI` (`BOOL`, Vorgabe `TRUE`), Daten-Ausgang `QO` (`BOOL`).
- **Verwendete interne FBs**:
  - **logiBUS_IXA**: `logiBUS::io::DI::logiBUS_IXA` — physischer digitaler Eingang, Adapter-Ausgang `IN`; `QI` kommt vom SubApp-Eingang `QI`.
  - **GreenWhiteBackground1_AX** (SubApp): `MyLib::sys::GreenWhiteBackground1_AX` — Ein-Objekt-Hintergrundfarbbaustein aus der Background-Farbbaustein-Familie, steuert die VT-Objekthintergrundfarbe anhand von `DI1`.
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_2` — verknüpft die `QO`-Ausgänge von `logiBUS_IXA` und `GreenWhiteBackground1_AX` zu `QO`.
- **Funktionsweise**: Der Adapter-Ausgang des physischen Eingangs wird direkt auf den `DI1`-Adaptereingang des Hintergrundfarbbausteins geführt, der die Farbumschaltung eines VT-Objekts übernimmt.

Programmablauf und Verbindungen

1. `Input` ⟶ `logiBUS_IXA.Input`; `u16ObjId` ⟶ `GreenWhiteBackground1_AX.u16ObjId`.
2. `logiBUS_IXA.IN` (Adapter) ⟶ `GreenWhiteBackground1_AX.DI1` (Adapter).
3. Initialisierung: `INIT` ⟶ `logiBUS_IXA.INIT`; `logiBUS_IXA.INITO` ⟶ `GreenWhiteBackground1_AX.INIT`; `GreenWhiteBackground1_AX.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`.
4. Qualität: `QI` ⟶ `logiBUS_IXA.QI` und `GreenWhiteBackground1_AX.QI`; `logiBUS_IXA.QO` ⟶ `AND_QO.IN1`, `GreenWhiteBackground1_AX.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`.

Anwendungsszenarien

- Rein visuelle Statusanzeige eines physischen digitalen Eingangs auf dem VT (z. B. Endschalter-Zustand), ohne dass der Wert per OPC-UA nach außen gemeldet werden muss.

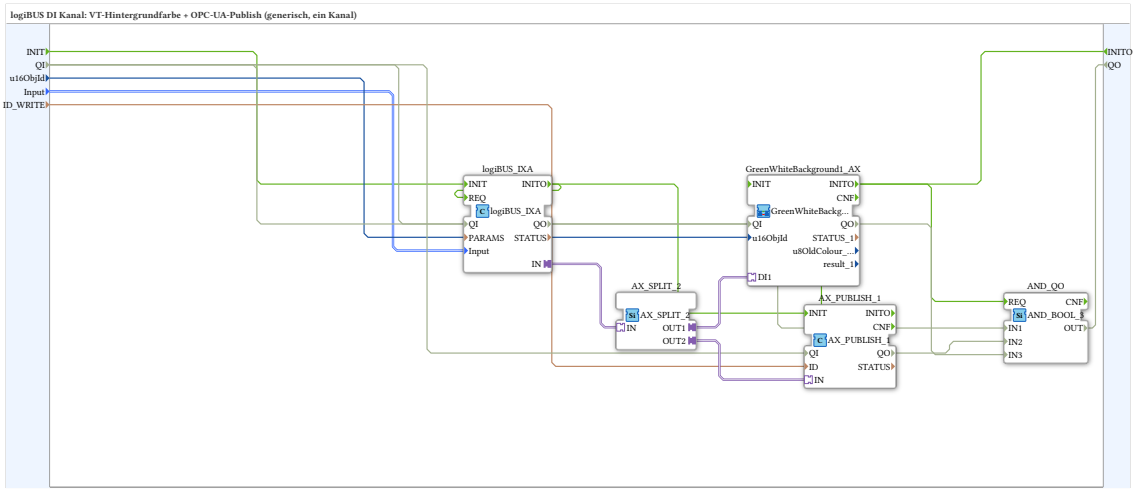
Zusammenfassung

Kombiniert einen physischen digitalen Eingang mit der Standard-Hintergrundfarbfamilie zu einer einfachen VT-Statusanzeige.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

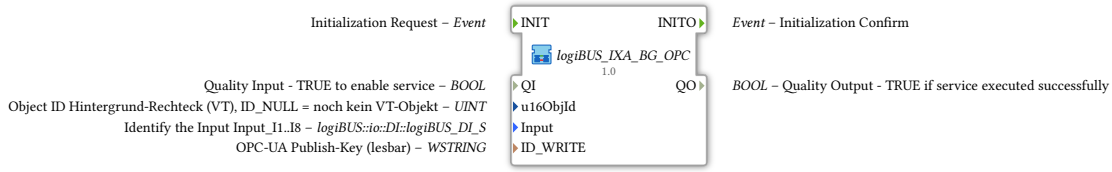
logiBUS_IXA_BG_OPC



• – ○ • – ○ • – ○ •

Einleitung

`logiBUS_IXA_BG_OPC` ist der wiederverwendbare Baustein für **einen einzelnen digitalen Eingang** mit VT-Statusanzeige (Hintergrundfarbe) und OPC-UA-Publish. Er wird 8-fach parametrisiert in `InputOutputTesterButton_DIDO_OPC_UA` instanziiert und unverändert auch im PWM-Beispiel `InputOutputTesterButton_PWM_OPC_UA` für dessen 8 Eingänge wiederverwendet.



Schnittstelle

- **Ereignis-Eingang**: `INIT` (`EInit`, Initialization Request)
- **Ereignis-Ausgang**: `INITO` (`EInit`, Initialization Confirm)
- **Daten-Eingänge**: `QI` (`BOOL`, Vorgabe `TRUE`, Quality Input), `u16ObjId`, `Input`, `ID_WRITE`
- **Daten-Ausgang**: `QO` (`BOOL`, Quality Output)

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: logiBUS_IXA_BG_OPC

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **logiBUS_IXA**: `logiBUS::io::DI::logiBUS_IXA` — Adapter-basierter digitaler Eingang; `QI` vom Baustein-Eingang, `Input` identifiziert `Input_I1..I8`.
  - **AX_SPLIT_2**: `adapter::events::unidirectional::AX_SPLIT_2` — verzweigt das eine Adapter-Signal von `logiBUS_IXA.IN` in zwei unabhängige Ziele.
  - **GreenWhiteBackground1_AX** (SubApp, `MyLib::sys`): setzt die VT-Hintergrundfarbe (Grün/Weiß) passend zum Eingangszustand.
  - **AX_PUBLISH_1**: `adapter::net::AX_PUBLISH_1` — OPC-UA-Publish-Adapter, `QI` vom Baustein-Eingang.
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_3` — verknüpft die `QO`-Ausgänge von `logiBUS_IXA`, `AX_PUBLISH_1` und `GreenWhiteBackground1_AX` per UND zum Baustein-Ausgang `QO`.
- **Funktionsweise**: Reiner Einweg-Datenfluss vom physischen Eingang zu zwei Anzeige-/Meldewegen — kein Rückschreiben möglich, da ein digitaler Eingang nicht extern gesetzt werden kann.

Programmablauf und Verbindungen

1. `Input` (Parameter, identifiziert `Input_I1..I8`) ⟶ `logiBUS_IXA.Input` (Datenverbindung, ausgeblendet).

2. `u16ObjId` (VT-Objekt-ID der Statusanzeige) ⟶ `GreenWhiteBackground1_AX.u16ObjId`.
3. `ID_WRITE` (OPC-UA-Adresse) ⟶ `AX_PUBLISH_1.ID`.
4. **Adapterkette**: `logiBUS_IXA.IN` ⟶ `AX_SPLIT_2.IN` ⟶ `AX_SPLIT_2.OUT1` ⟶ `GreenWhiteBackground1_AX.DI1` (VT-Anzeige) und `AX_SPLIT_2.OUT2` ⟶ `AX_PUBLISH_1.IN` (OPC-UA-Publish).
5. **Initialisierungskette**: `INIT` ⟶ `logiBUS_IXA.INIT`, `logiBUS_IXA.INITO` ⟶ `AX_PUBLISH_1.INIT`, `AX_PUBLISH_1.INITO` ⟶ `GreenWhiteBackground1_AX.INIT`, `GreenWhiteBackground1_AX.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`.
6. `QI` ⟶ `logiBUS_IXA.QI`, `AX_PUBLISH_1.QI` und `GreenWhiteBackground1_AX.QI`.
7. `logiBUS_IXA.QO` ⟶ `AND_QO.IN1`, `AX_PUBLISH_1.QO` ⟶ `AND_QO.IN2`, `GreenWhiteBackground1_AX.QO` ⟶ `AND_QO.IN3`; `AND_QO.OUT` ⟶ `QO`.

Technische Besonderheiten

- **AX_SPLIT_2 statt zweier Datenverbindungen**: Da ein Adapter-Ausgang nur an eine Stelle direkt verbunden werden kann, verzweigt `AX_SPLIT_2` das Signal für VT-Anzeige und OPC-UA-Publish.
- **Unidirektional**: Im Gegensatz zum Ausgangs-Pendant `Button_IXA_TO_logiBUS_QXA_BG_OPC` gibt es hier keinen `AX_SUBSCRIBE_1` — ein Eingang wird nicht vom Web aus geschrieben.

Anwendungsszenarien

- Jede Übung mit digitalen Eingängen, die sowohl auf dem VT (Statusfarbe) als auch per OPC-UA (Web-Client) sichtbar sein sollen.

Zusammenfassung

`logiBUS_IXA_BG_OPC` kapselt die Standard-Kombination „digitaler Eingang + VT-Statusfarbe + OPC-UA-Publish“ in einem einzigen, mehrfach wiederverwendbaren Baustein.

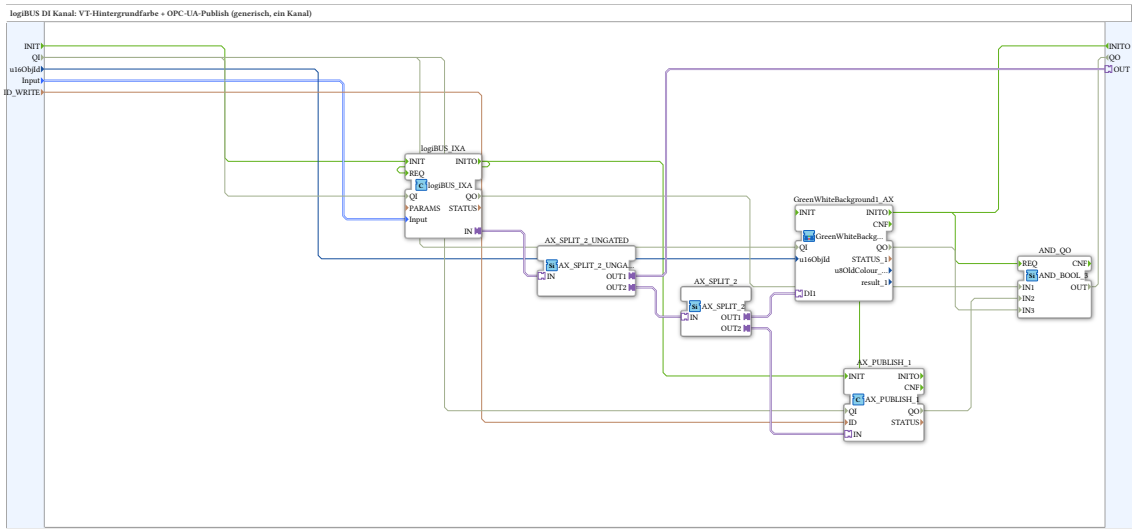
🛠️ Zugehörige Übungen

- InputOutputTesterButton_DIDO_OPC_UA
- InputOutputTesterButton_PWM_OPC_UA

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

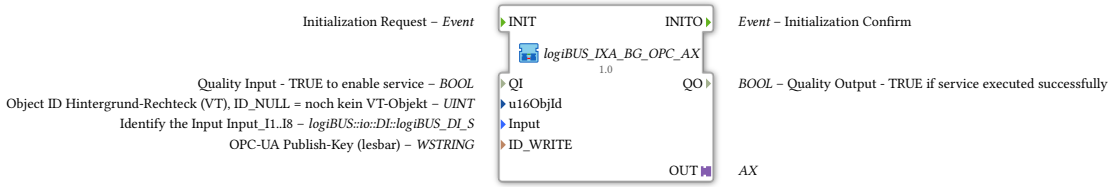
logiBUS_IXA_BG_OPC_AX



• – ○ • – ○ • – ○ •

Einleitung

`logiBUS_IXA_BG_OPC_AX` ist ein generischer Baustein für **einen logiBUS-DI-Kanal** mit VT-Hintergrundfarbe und OPC-UA-Publish. Gegenüber `logiBUS_IXA_BG_OPC` reicht er den Eingangszustand zusätzlich als AX-Plug `OUT` nach außen durch, sodass weitere Bausteine den Pegel verwenden können.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: logiBUS_IXA_BG_OPC_AX

- **Typ**: SubAppType
- **Verwendete interne Bausteine**:
  - **logiBUS_IXA**: `logiBUS::io::DI::logiBUS_IXA` – digitaler Eingang; `Input` identifiziert `Input_I1..I8`
  - **AX_SPLIT_2_UNGATED**: `adapter::events::unidirectional::AX_SPLIT_2_UNGATED` – verzweigt den Eingangszustand in den Plug `OUT` und in den zweiten Zweig
  - **AX_SPLIT_2**: `adapter::events::unidirectional::AX_SPLIT_2` – verzweigt den zweiten Zweig in VT-Anzeige und Publish
  - **GreenWhiteBackground1_AX**: SubApp `MyLib::sys::GreenWhiteBackground1_AX` – setzt die Hintergrundfarbe (Grün/Weiß)
  - **AX_PUBLISH_1**: `adapter::net::AX_PUBLISH_1` – OPC-UA-Publish
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_3` – verknüpft die `QO`-Ausgänge von `logiBUS_IXA`, `AX_PUBLISH_1` und `GreenWhiteBackground1_AX` zum SubApp-Ausgang `QO`
- **Importe**: `ID_NULL`, `logiBUS_DI`, `logiBUS_DI_S`

Schnittstellenstruktur

Ereignis-Eingänge / Ereignis-Ausgänge

- **INIT** (`EInit`, Eingang): Initialisierungsanforderung; wird an `logiBUS_IXA.INIT` durchgereicht
- **INITO** (`EInit`, Ausgang): Initialisierungsbestätigung; kommt von `GreenWhiteBackground1_AX.INITO`

Die Laufzeit-Ereignisse laufen weiterhin über den Adapter.

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input, Freigabe des Dienstes; wird an `logiBUS_IXA`, `AX_PUBLISH_1` und `GreenWhiteBackground1_AX` weitergereicht
- **u16ObjId** (`UINT`, Vorgabe `ID_NULL`): Object-ID des Hintergrund-Rechtecks (VT); `ID_NULL` = noch kein VT-Objekt
- **Input** (`logiBUS::io::DI::logiBUS_DI_S`, Vorgabe `logiBUS_DI::Invalid`): Identifiziert den Eingang `Input_I1..I8`
- **ID_WRITE** (`WSTRING`): OPC-UA-Publish-Schlüssel (lesbar)

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output, `AND` der `QO`-Ausgänge von `logiBUS_IXA`, `AX_PUBLISH_1` und `GreenWhiteBackground1_AX` (über `AND_QO`)

Adapter

- **OUT** (Plug): `adapter::types::unidirectional::AX` – Eingangszustand als Pegel

Programmablauf und Verbindungen

- `Input` ⟶ `logiBUS_IXA.Input`; `logiBUS_IXA.INITO` ⟶ `logiBUS_IXA.REQ` (erste Abfrage nach der Initialisierung)
- `logiBUS_IXA.IN` ⟶ `AX_SPLIT_2_UNGATED.IN`
- `AX_SPLIT_2_UNGATED.OUT1` ⟶ `OUT`
- `AX_SPLIT_2_UNGATED.OUT2` ⟶ `AX_SPLIT_2.IN`

- AX_SPLIT_2.OUT1 ⟶ GreenWhiteBackground1_AX.DI1; u16ObjId ⟶ GreenWhiteBackground1_AX.u16ObjId
- AX_SPLIT_2.OUT2 ⟶ AX_PUBLISH_1.IN; ID_WRITE ⟶ AX_PUBLISH_1.ID
- Initialisierung: INIT ⟶ logiBUS_IXA.INIT; logiBUS_IXA.INITO ⟶ AX_PUBLISH_1.INIT (zusätzlich zu logiBUS_IXA.INITO ⟶ logiBUS_IXA.REQ); AX_PUBLISH_1.INITO ⟶ GreenWhiteBackground1_AX.INIT; GreenWhiteBackground1_AX.INITO ⟶ INITO und ⟶ AND_QO.REQ
- Qualität: QI ⟶ logiBUS_IXA.QI, AX_PUBLISH_1.QI und GreenWhiteBackground1_AX.QI; deren QO-Ausgänge ⟶ AND_QO.IN1 (logiBUS_IXA), IN2 (AX_PUBLISH_1), IN3 (GreenWhiteBackground1_AX); AND_QO.OUT ⟶ QO

Technische Besonderheiten

- Zwei Splitter hintereinander, weil ein Adapter-Plug nur an ein Ziel angeschlossen werden kann und hier drei Ziele (OUT, VT-Anzeige, Publish) versorgt werden
- Einweg-Datenfluss: ein digitaler Eingang wird nicht von außen geschrieben
- Version 1.0 (2023-10-23)

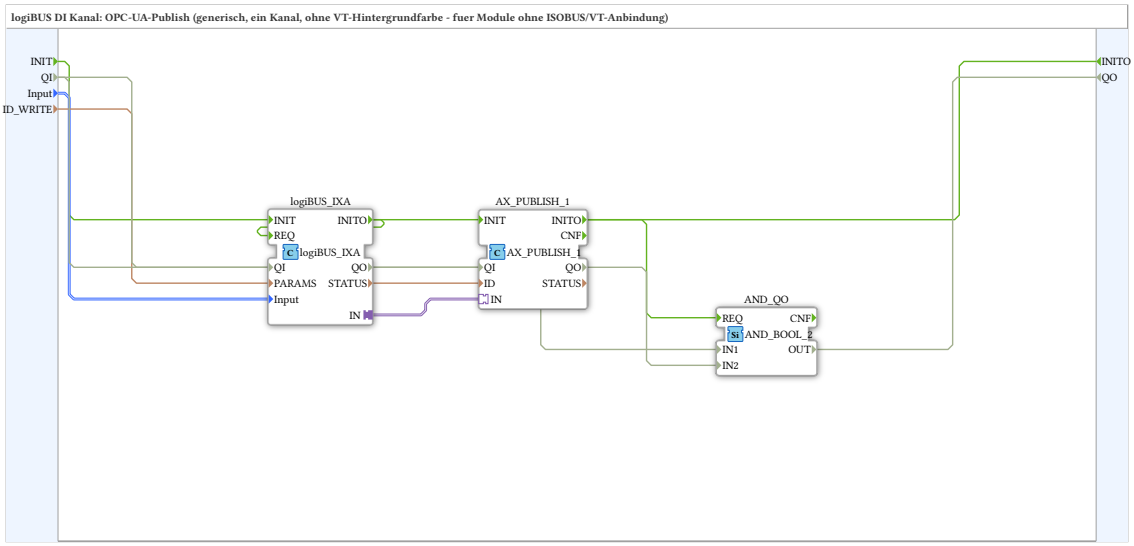
Anwendungsszenarien

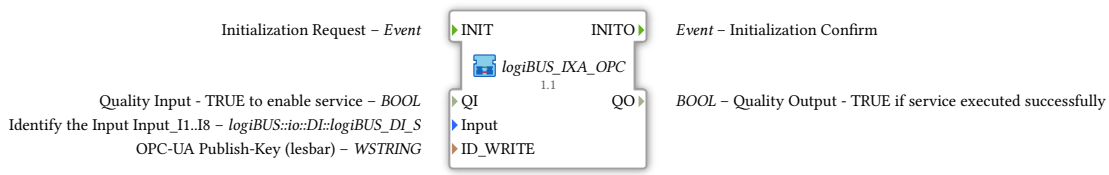
- Digitaler Eingang mit VT-Statusfarbe und OPC-UA-Publish, dessen Zustand zusätzlich von weiteren Bausteinen genutzt werden soll

Zusammenfassung

logiBUS_IXA_BG_OPC_AX erweitert logiBUS_IXA_BG_OPC um einen durchgereichten AX-Ausgang OUT für den Eingangszustand.

logiBUS_IXA_OPC





Einleitung

Die SubApp logiBUS_IXA_OPC realisiert die OPC-UA-Veröffentlichung eines einzelnen logiBUS-Digital-Eingangskanals (DI) ohne Visualisierungshintergrund (VT) und eignet sich für Module ohne ISOBUS/VT-Anbindung. Sie kombiniert den Baustein logiBUS_IXA (Kanalverarbeitung) mit dem Adapter AX_PUBLISH_1 (OPC-UA-Publishing) zu einer durchgängigen, generischen Ein-Kanal-Lösung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; startet die Init-Kette bei `logiBUS_IXA.INIT`

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung; kommt von `AX_PUBLISH_1.INITO`, dem Ende der Init-Kette

Daten-Eingänge

- **`QI`** – Typ: `BOOL` – Quality Input, Initialwert `TRUE`; wird an `logiBUS_IXA.QI` und `AX_PUBLISH_1.QI` weitergereicht.
- **`Input`** – Typ: `logiBUS::io::DI::logiBUS_DI_S` – Identifiziert den Eingangskanal (Input_I1..I8). Initialwert: `logiBUS_DI::Invalid`.
- **`ID_WRITE`** – Typ: `WSTRING` – OPC-UA-Publish-Key (lesbar).

Daten-Ausgänge

- **`QO`** – Typ: `BOOL` – Quality Output, UND-Verknüpfung (`AND_QO`, `AND_BOOL_2`) aus `logiBUS_IXA.QO` und `AX_PUBLISH_1.QO`.

Adapter

Keine an der Schnittstelle. Intern wird der Adapterausgang `IN` von `logiBUS_IXA` mit dem Adaptereingang `IN` von `AX_PUBLISH_1` verbunden.

Funktionsweise

Die SubApp nimmt über den Dateneingang `Input` die Daten eines logiBUS-DI-Kanals entgegen. Dieses Signal wird dem internen Baustein `logiBUS_IXA` zugeführt, der die Kanalverarbeitung übernimmt und die aufbereiteten Daten über seinen Adapterausgang `IN` bereitstellt. Dieser Ausgang ist mit dem Adaptereingang `IN` des Bausteins `AX_PUBLISH_1` verbunden. `AX_PUBLISH_1` veröffentlicht die empfangenen Daten über OPC-UA unter dem Schlüssel, der über den Eingang `ID_WRITE` gesetzt wird. Somit entsteht eine transparente Datenpipeline: logiBUS-DI → `logiBUS_IXA` → `AX_PUBLISH_1` → OPC-UA.

Initialisierung und Qualität: `INIT` → `logiBUS_IXA.INIT`, `logiBUS_IXA.INITO` → `AX_PUBLISH_1.INIT`, `AX_PUBLISH_1.INITO` → `INITO` und `AND_QO.REQ`. `QI` wird an `logiBUS_IXA.QI` und `AX_PUBLISH_1.QI` weitergereicht; `logiBUS_IXA.QO` → `AND_QO.IN1`, `AX_PUBLISH_1.QO` → `AND_QO.IN2` (`iec61131::booleanOperators::AND_BOOL_2`), `AND_QO.OUT` → `QO`.

Technische Besonderheiten

- **Generische Ein-Kanal-Lösung** ohne VT-Hintergrundfarbe.
- **OPC-UA-Publishing** über den Adapter `adapter::net::AX_PUBLISH_1`.
- Der Parameter `PARAMS` von `logiBUS_IXA` ist leer und im Editor unsichtbar (`Visible=false`).
- Die internen Bausteine `logiBUS_IXA` und `AX_PUBLISH_1` erhalten ihr `QI` vom Eingang `QI` (Initialwert `TRUE`).
- Die SubApp besitzt als Ereignisse nur `INIT`/`INITO` und als Ausgang nur `QO`; die eigentlichen Messdaten werden ausschließlich per OPC-UA publiziert, nicht über Ausgangsports.

Zustandsübersicht

Da die SubApp selbst keine expliziten Zustände definiert, sind keine Zustände über die Schnittstelle sichtbar. Die internen Zustände werden durch die Bausteine `logiBUS_IXA` (z. B. Initialisierung, Kanalverarbeitung) und `AX_PUBLISH_1` (z. B. Verbindungsstatus, Veröffentlichung aktiv) bestimmt. Sie sind jedoch nicht nach außen geführt und können nur über die jeweiligen Baustein-Dokumentationen eingesehen werden.

Anwendungsszenarien

- **Integration in OPC-UA-Steuerungen**: Einbindung eines logiBUS-DI-Kanals in ein übergeordnetes OPC-UA-System zur Überwachung oder Steuerung.
- **Module ohne ISOBUS/VT-Anbindung**: Einsatz in Umgebungen, die ausschließlich OPC-UA unterstützen und keine proprietären Visualisierungslösungen benötigen.
- **Komprimierte Ein-Kanal-Publikation**: Wenn nur ein einzelner digitaler Eingang publiziert werden muss und ein schlanker, ressourcenschonender Baustein gewünscht ist.

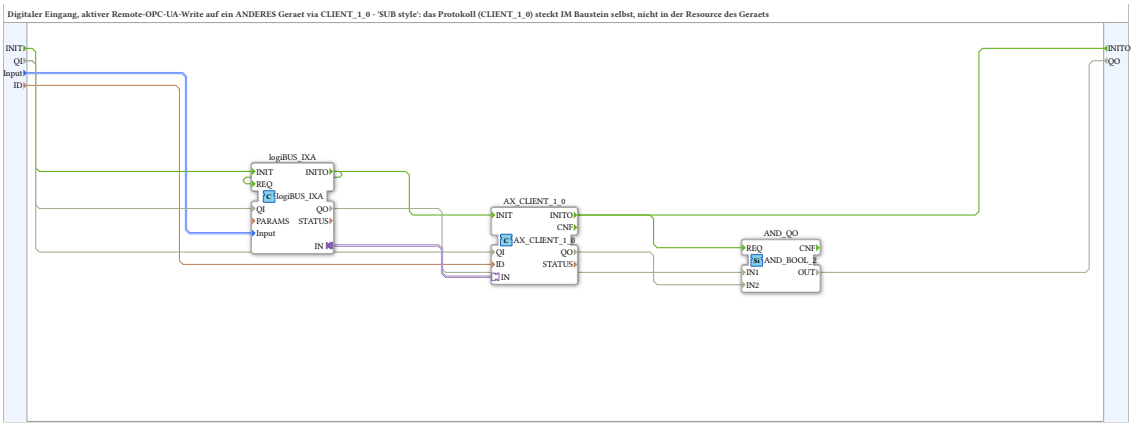
Vergleich mit ähnlichen Bausteinen

Gegenüber anderen logiBUS-Publish-Bausteinen, die mehrere Kanäle oder VT-Hintergrunddarstellung unterstützen, fokussiert `logiBUS_IXA_OPC` bewusst auf eine minimale, einzelne OPC-UA-Schnittstelle ohne Visualisierungsfunktionen. Dies reduziert Komplexität und Speicherbedarf und eignet sich besonders für einfache Anwendungen mit strengen Ressourcengrenzen.

Fazit

`logiBUS_IXA_OPC` ist eine kompakte SubApp, die einen logiBUS-DI-Kanal nahtlos über OPC-UA publiziert. Sie wurde speziell für Module ohne ISOBUS/VT-Anbindung konzipiert und bietet eine einfache Konfiguration über die beiden Eingänge. Durch die klare Trennung von Datenerfassung (`logiBUS_IXA`) und Veröffentlichung (`AX_PUBLISH_1`) ist eine flexible und zuverlässige Integration in industrielle Netzwerke gewährleistet.

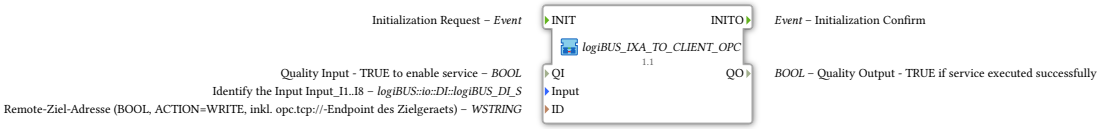
logiBUS_IXA_TO_CLIENT_OPC



• – ○ • – ○ • – ○ •

Einleitung

`logiBUS_IXA_TO_CLIENT_OPC` liest einen digitalen Eingang (`logiBUS_IXA`) und schreibt den Wert aktiv per `AX_CLIENT_1_0` auf ein ANDERES Geraet via OPC-UA - „Krauternter-Stil“ (laut Quellcode-Kommentar): das Protokoll (`AX_CLIENT_1_0`) steckt IM Baustein selbst, nicht in der Resource des Geraets, im Gegensatz zum „SUB style“ der PC_A/PC_B-Bausteine, wo Trigger/Server-Instanzen extra im Composite verkabelt sind.



Verwendete Funktionsbausteine (FBs)

- **logiBUS_IXA** (`logiBUS::io::DI::logiBUS_IXA`): physischer digitaler Eingang, identifiziert ueber `Input_I1..I8`.
- **AX_CLIENT_1_0** (`adapter::net::AX_CLIENT_1_0`): schreibt den Adapterwert aktiv auf die Remote-Zieladresse `ID` (inkl. `opc.tcp://`-Endpoint des Zielgeraets).

Initialisierung und Qualität

Der Baustein besitzt den Ereigniseingang `INIT`, den Ereignisausgang `INITO` (je `EInit`), den Dateneingang `QI` (`BOOL`, Default `TRUE`) und den Datenausgang `QO` (`BOOL`). `INIT` → `logiBUS_IXA.INIT`, `logiBUS_IXA.INITO` → `AX_CLIENT_1_0.INIT` (und → `logiBUS_IXA.REQ`), `AX_CLIENT_1_0.INITO` → `INITO` und → `AND_QO.REQ`. `QI` geht an `logiBUS_IXA` und `AX_CLIENT_1_0`. `QO` ist der Ausgang der Instanz `AND_QO` (`iec61131::booleanOperators::AND_BOOL_2`), die `logiBUS_IXA.QO` und `AX_CLIENT_1_0.QO` verknüpft.

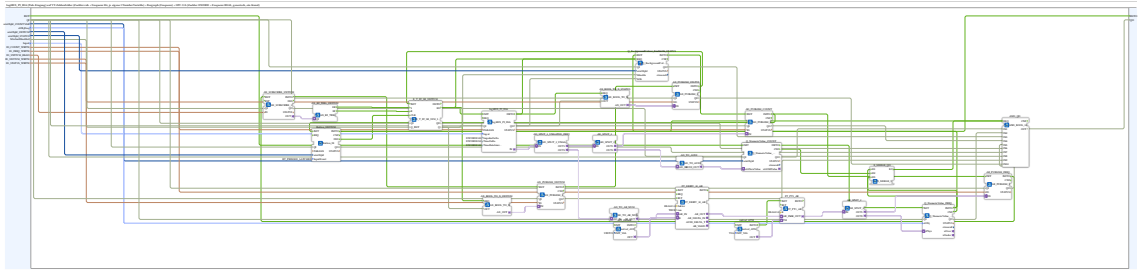
Zusammenfassung

Ein-Baustein-Bruecke von einem physischen digitalen Eingang zu einem aktiven Remote-Write auf ein anderes Geraet - Gegenstueck zu `logiBUS_QXA_FROM_SUBSCRIBE_OPC` auf der Empfaengerseite.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

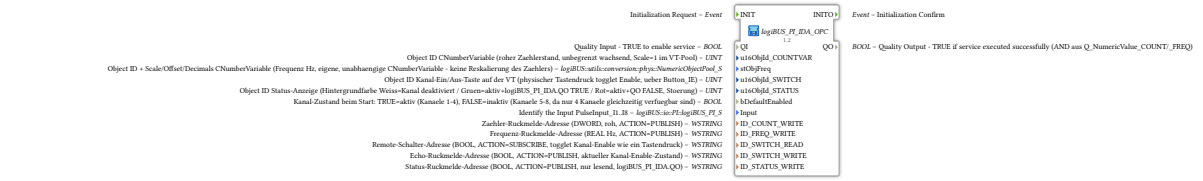
logiBUS_PI_IDA_OPC



• – ○ • – ○ • – ○ •

Einleitung

`logiBUS_PI_IDA_OPC` bindet einen physischen Puls-/Zähler-Eingang (`logiBUS_PI_IDA`, ein Kanal) an zwei unabhängige VT-Zahlenfelder (roher Zählerstand und daraus abgeleitete Frequenz in Hz) sowie an OPC-UA an: der rohe Zählerstand (DWORD) wird direkt publiziert, die Frequenz wird per zeitlicher Ableitung (`FT_DERIV_10_AR`) aus dem Zählerstand berechnet, mit einem PT1-Glied (`FT_PT1_AR`) geglättet und separat publiziert (REAL, physikalisch skaliert über `NumericObjectPool_S`). Zusätzlich lässt sich der Kanal per VT-Taste oder per OPC-UA-Schalter ein- und ausschalten (Enable); eine Status-Anzeige (Hintergrundfarbe) und ein OPC-UA-Status melden Deaktiviert/Aktiv/Störung.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung (von `Button_SWITCH.INITO`)

Daten-Eingänge

- **QI** (`BOOL`, Vorgabe `TRUE`): Quality Input – TRUE gibt die Dienste frei
- **u16ObjId_COUNTVAR** (`UINT`, Vorgabe `ID_NULL`): Object ID der `CNumberVariable` für den rohen Zählerstand (unbegrenzt wachsend, Scale=1 im VT-Pool)
- **stObjFreq** (`NumericObjectPool_S`): Object ID + Scale/Offset/Decimals der `CNumberVariable` für die Frequenz in Hz (eigene, unabhängige Variable, keine Reskalierung des Zählers)
- **u16ObjId_SWITCH** (`UINT`, Vorgabe `ID_NULL`): Object ID der Kanal-Ein/Aus-Taste auf der VT (ein Tastendruck togglet das Enable, über `Button_IE`)
- **u16ObjId_STATUS** (`UINT`, Vorgabe `ID_NULL`): Object ID der Status-Anzeige (Hintergrundfarbe Weiß = Kanal deaktiviert, Grün = aktiv und `logiBUS_PI_IDA.QO` TRUE, Rot = aktiv und `QO` FALSE, Störung)
- **bDefaultEnabled** (`BOOL`): Kanal-Zustand beim Start (TRUE = aktiv, z. B. Kanäle 1–4; FALSE = inaktiv, z. B. Kanäle 5–8, da nur 4 Kanäle gleichzeitig verfügbar sind)
- **Input** (`logiBUS::io::PI::logiBUS_PI_S`): Auswahl des Eingangs `PulseInput_I1..I8` (Vorgabe `logiBUS_PI::Invalid`)
- **ID_COUNT_WRITE** (`WSTRING`): Zähler-Rückmelde-Adresse (DWORD, roh, `ACTION=PUBLISH`)

- **ID_FREQ_WRITE** (WSTRING): Frequenz-Rückmelde-Adresse (REAL Hz, ACTION=PUBLISH)
- **ID_SWITCH_READ** (WSTRING): Remote-Schalter-Adresse (BOOL, ACTION=SUBSCRIBE, togglet das Kanal-Enable wie ein Tastendruck)
- **ID_SWITCH_WRITE** (WSTRING): Echo-Rückmelde-Adresse (BOOL, ACTION=PUBLISH, aktueller Kanal-Enable-Zustand)
- **ID_STATUS_WRITE** (WSTRING): Status-Rückmelde-Adresse (BOOL, ACTION=PUBLISH, nur lesend, logiBUS_PI_IDA.QO)

Daten-Ausgänge

- **QO** (BOOL): Quality Output – TRUE, wenn die Dienste erfolgreich ausgeführt wurden (AND_QO, AND_BOOL_10: UND aus Q_NumericValue_COUNT.QO, Q_NumericValue_FREQ.QO, Q_BackgroundColour_EnableOk_STATUS.QO, AD_PUBLISH_COUNT.QO, AR_PUBLISH_FREQ.QO, E_T_FF_SR_SWITCH.QO, Button_SWITCH.QO, AX_SUBSCRIBE_SWITCH.QO, AX_PUBLISH_SWITCH.QO und AX_PUBLISH_STATUS.QO)

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: logiBUS_PI_IDA_OPC

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **logiBUS_PI_IDA**: logiBUS::io::PI::logiBUS_PI_IDA — physischer Puls-Eingang, ImpulseDelta=DWORD#100, TimeDelta=DWORD#250, TimeRateLimit=DWORD#100; QI kommt vom Kanal-Enable (E_T_FF_SR_SWITCH.Q).
  - **AD_SPLIT_2_UNGATED_FREQ**: adapter::events::unidirectional::AD_SPLIT_2_UNGATED — verzweigt den DWORD-Zählerwert in den Zähler- und den Frequenzzweig.
  - **AD_SPLIT_2**: adapter::events::unidirectional::AD_SPLIT_2 — verzweigt den Zählerwert auf OPC-UA-Publish und VT-Anzeige.
  - **AD_PUBLISH_COUNT**: adapter::net::AD_PUBLISH_1 — publiziert den rohen Zählerstand per OPC-UA, Zieladresse ID_COUNT_WRITE.
  - **AD_TO_AUDI**: adapter::conversion::unidirectional::AD_TO_AUDI — Bit-Reinterpretation DWORD→UDINT für das VT-Zählerfeld.
  - **Q_NumericValue_COUNT**: isobus::UT::Q::Q_NumericValue_AUDI — schreibt den rohen Zählerstand in u16ObjId_COUNTVAR.
  - **AD_TO_AR_NUM**: adapter::conversion::unidirectional::AD_TO_AR_NUM — numerisch korrekte Umwandlung DWORD-Zählerstand ⟶ REAL für die Ableitung.
  - **FT_DERIV_10_AR**: OSCAT_adapter::Control::FT_DERIV_10_AR (Faktor=1.0, run=TRUE) — zeitliche Ableitung, liefert die Änderungsrate des Zählerstands = Frequenz in Hz; der Parameter K ist ein AUI-Adapter-Socket und wird von LIT_DerivK (adapter::types::unidirectional::AUI::initval::initval_AUI, INIT_VAL=9) gespeist.
  - **FT_PT1_AR**: OSCAT_adapter::Control::FT_PT1_AR (K=1.0) — PT1-Glättung der Frequenz; die Zeitkonstante TM kommt von initval_ATM (adapter::types::unidirectional::ATM::initval::initval_ATM, INIT_VAL=T#2s).
  - **AR_SPLIT_2**: adapter::events::unidirectional::AR_SPLIT_2 — verzweigt den geglätteten Frequenzwert auf zwei Ausgänge.
  - **AR_PUBLISH_FREQ**: adapter::net::AR_PUBLISH_1 — publiziert die Frequenz per OPC-UA, Zieladresse ID_FREQ_WRITE.
  - **Q_NumericValue_FREQ**: isobus::UT::Q::Q_NumericValue_PHYSA — schreibt die Frequenz physikalisch skaliert (stObjFreq: r32Scale, i32Offset, u8Decimals) in das VT-Zahlenfeld/Bargraph.
  - **Button_SWITCH**: isobus::UT::io::Button::Button_IE (InputEvent=BT_PRESSED_LATCHED) — liest die Kanal-Ein/Aus-Taste (u16ObjId_SWITCH).
  - **AX_SUBSCRIBE_SWITCH** / **AX_RF_TRIG_SWITCH**: adapter::net::AX_SUBSCRIBE_1 und adapter::events::unidirectional::AX_RF_TRIG — Remote-Schalter (ID_SWITCH_READ) mit Flankenerkennung (ER ⟶ Set, EF ⟶ Reset des Enable-Flipflops).
  - **E_T_FF_SR_SWITCH**: iec61499::events::E_T_FF_SR_SYM_INIT — Enable-Flipflop (Q_INIT = bDefaultEnabled); Button_SWITCH.IND taktet CLK.
  - **AX_BOOL_TO_X_SWITCH** / **AX_PUBLISH_SWITCH**: adapter::conversion::unidirectional::AX_BOOL_TO_X und adapter::net::AX_PUBLISH_1 — publizieren den Enable-Zustand (ID_SWITCH_WRITE).

  - **Q_BackgroundColour_EnableOk_STATUS**: SubApp `MyLib::sys::Q_BackgroundColour_EnableOk_A` — Status-Hintergrundfarbe (`u16ObjId_STATUS`, `bEnable` = Enable, `bOk` = `logiBUS_PI_IDA.QO`).
  - **AX_BOOL_TO_X_STATUS** / **AX_PUBLISH_STATUS**: `adapter::conversion::unidirectional::AX_BOOL_TO_X` und `adapter::net::AX_PUBLISH_1` — publizieren `logiBUS_PI_IDA.QO` (`ID_STATUS_WRITE`).
  - **E_MERGE_QO**: `iec61499::events::E_MERGE_3` — vereint `Q_NumericValue_FREQ.INITO`, `Q_NumericValue_COUNT.CNF` und `Q_NumericValue_FREQ.CNF`; sein Ausgang `EO` triggert `AND_QO.REQ`, sodass `QO` bei jedem neuen `Q_NumericValue_COUNT`-/`Q_NumericValue_FREQ`-Wert (`CNF`) neu ausgewertet wird (bis v1.1 war `EO` nicht angeschlossen, `QO` wurde nur beim Start berechnet).
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_10` — bildet `QO` (siehe Daten-Ausgänge).
- **Funktionsweise**: Der rohe Zählerstand wird verzweigt: ein Zweig geht direkt an die OPC-UA-Publikation, ein zweiter über `AD_TO_AUDI` an das VT-Zählerfeld, ein dritter über `AD_TO_AR_NUM`, `FT_DERIV_10_AR` und `FT_PT1_AR` zur Frequenzberechnung, deren Ergebnis wiederum verdoppelt wird (OPC-UA-Publikation und VT-Anzeige). Das Kanal-Enable (Taste, Remote-Schalter) gibt `logiBUS_PI_IDA` frei.

Programmablauf und Verbindungen

1. `Input` ⟶ `logiBUS_PI_IDA.Input`; `u16ObjId_COUNTVAR` ⟶ `Q_NumericValue_COUNT.u16ObjId`; `stObjFreq` ⟶ `Q_NumericValue_FREQ.stObj`; `ID_COUNT_WRITE` ⟶ `AD_PUBLISH_COUNT.ID`; `ID_FREQ_WRITE` ⟶ `AR_PUBLISH_FREQ.ID`.
2. `logiBUS_PI_IDA.IN` (Adapter) ⟶ `AD_SPLIT_2_UNGATED_FREQ.IN`; `AD_SPLIT_2_UNGATED_FREQ.OUT1` ⟶ `AD_SPLIT_2.IN`; `AD_SPLIT_2_UNGATED_FREQ.OUT2` ⟶ `AD_TO_AR_NUM.AD_IN`.
3. `AD_SPLIT_2.OUT1` ⟶ `AD_PUBLISH_COUNT.IN` (Zählerstand-OPC-UA-Zweig).
4. `AD_SPLIT_2.OUT2` ⟶ `AD_TO_AUDI.AD_IN`; `AD_TO_AUDI.AUDI_OUT` ⟶ `Q_NumericValue_COUNT.u32NewValue` (VT-Zählerstand-Zweig).
5. `AD_TO_AR_NUM.AR_OUT` ⟶ `FT_DERIV_10_AR.AR_IN`; `LIT_DerivK.OUT` ⟶ `FT_DERIV_10_AR.K`; `FT_DERIV_10_AR.AR_OUT` ⟶ `FT_PT1_AR.AR_IN` (mit `initval_ATM.OUT` ⟶ `FT_PT1_AR.TM`); `FT_PT1_AR.AR_OUT` ⟶ `AR_SPLIT_2.IN` (Frequenz-Berechnungszweig).
6. `AR_SPLIT_2.OUT1` ⟶ `AR_PUBLISH_FREQ.IN` (Frequenz-OPC-UA-Zweig).
7. `AR_SPLIT_2.OUT2` ⟶ `Q_NumericValue_FREQ.rPhys` (Frequenz-VT-Zweig).
8. **Kanal-Enable**: `Button_SWITCH.IND` ⟶ `E_T_FF_SR_SWITCH.CLK`; `AX_SUBSCRIBE_SWITCH.OUT` ⟶ `AX_RF_TRIG_SWITCH.QI`, `AX_RF_TRIG_SWITCH.ER` ⟶ `E_T_FF_SR_SWITCH.S`, `AX_RF_TRIG_SWITCH.EF` ⟶ `E_T_FF_SR_SWITCH.R`; `bDefaultEnabled` ⟶ `E_T_FF_SR_SWITCH.Q_INIT`; `E_T_FF_SR_SWITCH.Q` ⟶ `logiBUS_PI_IDA.QI`, `Q_BackgroundColour_EnableOk_STATUS.bEnable` und `AX_BOOL_TO_X_SWITCH.OUT` (⟶ `AX_PUBLISH_SWITCH.IN`); `E_T_FF_SR_SWITCH.EO` ⟶ `logiBUS_PI_IDA.INIT`, `Q_BackgroundColour_EnableOk_STATUS.REQ` und `AX_BOOL_TO_X_SWITCH.REQ`.
9. **Status**: `logiBUS_PI_IDA.QO` ⟶ `Q_BackgroundColour_EnableOk_STATUS.bOk` und `AX_BOOL_TO_X_STATUS.OUT` (⟶ `AX_PUBLISH_STATUS.IN`); `logiBUS_PI_IDA.INITO` ⟶ `Q_BackgroundColour_EnableOk_STATUS.REQ` und `AX_BOOL_TO_X_STATUS.REQ`; `u16ObjId_STATUS` ⟶ `Q_BackgroundColour_EnableOk_STATUS.u16ObjId`.
10. **Initialisierungskette**: `INIT` ⟶ `Q_NumericValue_COUNT.INIT` ⟶ `Q_NumericValue_FREQ.INIT` ⟶ `Q_BackgroundColour_EnableOk_STATUS.INIT` ⟶ `AD_PUBLISH_COUNT.INIT` ⟶ `Button_SWITCH.INIT` ⟶ `INITO` (und ⟶ `AND_QO.REQ`; zusätzlich triggert `E_MERGE_QO.EO` ⟶ `AND_QO.REQ`). Parallel läuft von `AD_PUBLISH_COUNT.INITO` die Kette `AR_PUBLISH_FREQ` ⟶ `AX_SUBSCRIBE_SWITCH` ⟶ `AX_PUBLISH_SWITCH` ⟶ `AX_PUBLISH_STATUS` ⟶ `E_T_FF_SR_SWITCH` (jeweils `INITO` ⟶ nächstes `INIT`); `initval_ATM.INITO` ⟶ `FT_PT1_AR.INIT`.
11. **Qualität**: `QI` ⟶ `.QI` von `Q_NumericValue_COUNT`, `Q_NumericValue_FREQ`, `AD_PUBLISH_COUNT`, `AR_PUBLISH_FREQ`, `E_T_FF_SR_SWITCH`, `Button_SWITCH`, `AX_SUBSCRIBE_SWITCH`, `AX_PUBLISH_SWITCH`, `AX_PUBLISH_STATUS` und `Q_BackgroundColour_EnableOk_STATUS`; deren `QO`-Ausgänge ⟶ `AND_QO.IN1`..`IN10`; `AND_QO.OUT` ⟶ `QO`.

Technische Besonderheiten

- Die Frequenz wird nicht vom Hardware-FB selbst geliefert, sondern rein aus der zeitlichen Ableitung (`FT_DERIV_10_AR`, `Faktor=1.0`, `K=9`) des rohen, monoton wachsenden Zählerstands berechnet und mit `FT_PT1_AR` (`TM=T#2s`) geglättet — der Zähler selbst wird nie zurückgesetzt.
- `K` von `FT_DERIV_10_AR` ist ein AUI-Adapter-Socket und kein Parameter mehr (Version 1.1); der Wert 9 kommt über `LIT_DerivK`.

- `AD_TO_AR_NUM` statt `AD_TO_AR` wird bewusst verwendet, um eine numerisch korrekte DWORD⟶REAL-Umwandlung zu erhalten statt einer Bit-Reinterpretation (dieselbe Falle wie bei `F_AI_RAW_TO_PERCENT_AD` beschrieben).
- Zählerstand und Frequenz nutzen unabhängige VT-Zahlenfelder (`u16ObjId_COUNTVAR` und `stObjFreq`) mit unterschiedlicher Skalierung — die Frequenz-Anzeige nutzt zusätzlich `NumericObjectPool_S` für physikalisches Scaling/Decimals, der Zählerstand nicht.
- Die `QI`-Eingänge der internen Bausteine sind nicht mehr fest auf `TRUE` parametriert, sondern werden über `QI` der SubApp freigegeben (Ausnahme: `logiBUS_PI_IDA.QI` hängt am Kanal-Enable).

Anwendungsszenarien

- Durchflusszähler oder Drehzahlmesser, bei denen sowohl der rohe (unbegrenzt wachsende) Zählerstand für Diagnosezwecke als auch eine daraus abgeleitete, sofort interpretierbare Frequenz (z. B. Hz, U/min) angezeigt und übertragen werden sollen; der Kanal ist per VT-Taste oder OPC-UA ein- und ausschaltbar.

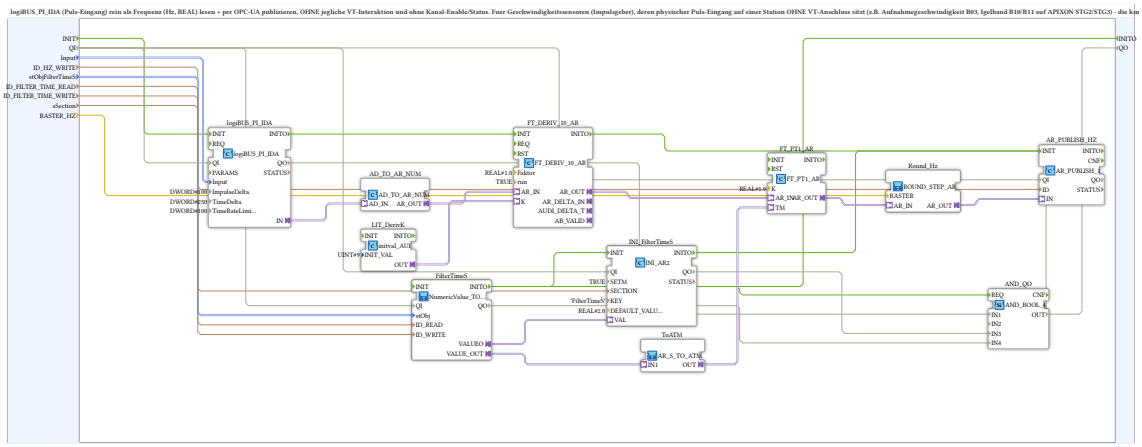
Zusammenfassung

Vollständige VT- und OPC-UA-Anbindung eines physischen Puls-Eingangs mit parallelem rohem Zählerstand, daraus per zeitlicher Ableitung und PT1-Glättung berechneter Frequenz sowie schaltbarem Kanal-Enable mit Status-Anzeige; `INIT`/`INITO` sind durch die Dienste gekettet, `QO` ist das UND der Dienste.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

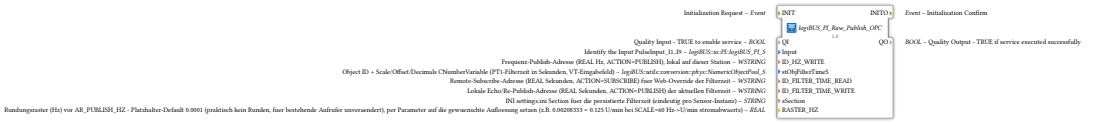
logiBUS_PI_Raw_Publish_OPC



• — ○ • — ○ • — ○ •

Einleitung

`logiBUS_PI_Raw_Publish_OPC` liest einen logiBUS-Pulseingang (`logiBUS_PI_IDA`) rein als **Frequenz (Hz, `REAL`)** und veröffentlicht diese per OPC-UA – **ohne jegliche VT-Interaktion** und ohne Kanal-Enable/Status. Er ist für Geschwindigkeitssensoren (Impulsgeber) gedacht, deren physischer Puls-Eingang auf einer Station **ohne VT-Anschluss** sitzt (z. B. Aufnahmegeschwindigkeit B03, Igelband B10/B11 auf APIXON STG2/STG3). Die km/h-Umrechnung und VT-Anzeige laufen dann auf der Station mit VT-Anschluss, gespeist per Remote-Subscribe dieser Frequenz (analog zu `logiBUS_AI_Raw_Publish_OPC`/`logiBUS_AI_Calibrate_Remote_IDA_OPC`).



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialization Request

Ereignis-Ausgänge

- **INITO** (EInit): Initialization Confirm

Daten-Eingänge

- **QI** (BOOL, Vorgabe TRUE): Quality Input, TRUE aktiviert den Dienst
- **Input** (logiBUS::io::PI::logiBUS_PI_S, Vorgabe logiBUS_PI::Invalid): Identifiziert den Pulseingang PulseInput_I1..I9
- **ID_HZ_WRITE** (WSTRING): Frequenz-Publish-Adresse (REAL Hz, ACTION=PUBLISH), lokal auf dieser Station
- **stObjFilterTimeS** (NumericObjectPool_S): Object-ID + Scale/Offset/Decimals der CNumberVariable für die PT1-Filterzeit in Sekunden (VT-Eingabefeld)
- **ID_FILTER_TIME_READ** (WSTRING): Remote-Subscribe-Adresse (REAL Sekunden, ACTION=SUBSCRIBE) für den Web-Override der Filterzeit
- **ID_FILTER_TIME_WRITE** (WSTRING): Lokale Echo-/Re-Publish-Adresse (REAL Sekunden, ACTION=PUBLISH) der aktuellen Filterzeit
- **sSection** (STRING): INI-Sektion in settings.ini für die persistierte Filterzeit (eindeutig pro Sensor-Instanz)
- **RASTER_HZ** (REAL, Vorgabe 0.0001): Rundungsraster (Hz) vor AR_PUBLISH_HZ. Der Vorgabewert entspricht praktisch keinem Runden (für bestehende Aufrufer unverändert); per Parameter auf die gewünschte Auflösung setzen (z. B. 0.00208333 = 0,125 U/min bei SCALE=60 Hz → U/min stromabwärts)

Daten-Ausgänge

- **QO** (BOOL): Quality Output, UND-Verknüpfung (AND_QO, AND_BOOL_4) der QO-Ausgänge von logiBUS_PI_IDA, AR_PUBLISH_HZ, INI_FilterTimeS und FilterTimeS

Adapter

Dieser Funktionsblock hat keine Adapter an der Schnittstelle; intern werden AR-/AD-Adapter verwendet.

Verwendete Funktionsbausteine (FBs)

- **logiBUS_PI_IDA**: logiBUS::io::PI::logiBUS_PI_IDA (ImpulseDelta=100, TimeDelta=250, TimeRateLimit=100) – Pulseingang als AD-Adapter
- **AD_TO_AR_NUM**: adapter::conversion::unidirectional::AD_TO_AR_NUM – wandelt den Zählwert in REAL
- **FT_DERIV_10_AR**: OSCAT_adapter::Control::FT_DERIV_10_AR (Faktor=1.0, run=TRUE) – Ableitung (Impulse pro Zeit = Frequenz)
- **LIT_DerivK**: initval_AUI (INIT_VAL=9) – Konstante K der Ableitung
- **FT_PT1_AR**: OSCAT_adapter::Control::FT_PT1_AR (K=1.0) – PT1-Filter
- **Round_Hz**: OSCAT_adapter::Mathematical::ROUND_STEP_AR – rundet auf das Raster RASTER_HZ
- **AR_PUBLISH_HZ**: adapter::net::AR_PUBLISH_1 – publiziert die Frequenz
- **FilterTimeS**: SubApp MyLib::sys::NumericValue_TO_AR2_OPC – Filterzeit aus VT-Eingabefeld bzw. OPC-UA
- **INI_FilterTimeS**: eclipse4diac::storage::INI_AR2 (SETM=TRUE, KEY='FilterTimeS', Vorgabe 2.0) – persistiert die Filterzeit
- **ToATM**: SubApp MyLib::sys::AR_S_TO_ATM – wandelt Sekunden in einen ATM-Zeitwert
- **AND_QO**: iec61131::booleanOperators::AND_BOOL_4 – verknüpft die QO-Ausgänge von logiBUS_PI_IDA, AR_PUBLISH_HZ, INI_FilterTimeS und FilterTimeS zu QO

Programmablauf und Verbindungen

1. Input → logiBUS_PI_IDA.Input; logiBUS_PI_IDA.IN → AD_TO_AR_NUM.AD_IN.
2. AD_TO_AR_NUM.AR_OUT → FT_DERIV_10_AR.AR_IN; LIT_DerivK.OUT → FT_DERIV_10_AR.K.
3. FT_DERIV_10_AR.AR_OUT → FT_PT1_AR.AR_IN.
4. Filterzeit: FilterTimeS.VALUEO → INI_FilterTimeS.VAL (Persistenz); FilterTimeS.VALUE_OUT → ToATM.IN1; ToATM.OUT → FT_PT1_AR.TM.
5. FT_PT1_AR.AR_OUT → Round_Hz.AR_IN (RASTER_HZ → Round_Hz.RASTER) → AR_PUBLISH_HZ.IN.
6. ID_HZ_WRITE → AR_PUBLISH_HZ.ID.
7. Initialisierung: INIT → logiBUS_PI_IDA.INIT; logiBUS_PI_IDA.INITO → AR_PUBLISH_HZ.INIT; AR_PUBLISH_HZ.INITO → INI_FilterTimeS.INIT; INI_FilterTimeS.INITO → FilterTimeS.INIT; FilterTimeS.INITO → INITO und → AND_QO.REQ.

8. Qualität: `QI` wird an `logiBUS_PI_IDA`, `AR_PUBLISH_HZ`, `INI_FilterTimeS` und `FilterTimeS` verteilt (kein fest verdrahteter `QI`-Parameter mehr); deren `QO` ⟶ `AND_QO.IN1` bis `AND_QO.IN4` (in dieser Reihenfolge); `AND_QO.OUT` ⟶ `QO`.

Technische Besonderheiten

- Keine VT-Interaktion für die Frequenz selbst, kein Kanal-Enable und kein Status.
- Die Filterzeit ist INI-persistiert und kann über das VT-Eingabefeld oder OPC-UA geändert werden.
- Mit `RASTER_HZ` lässt sich die Auflösung begrenzen, um unnötige Publish-Ereignisse zu vermeiden.

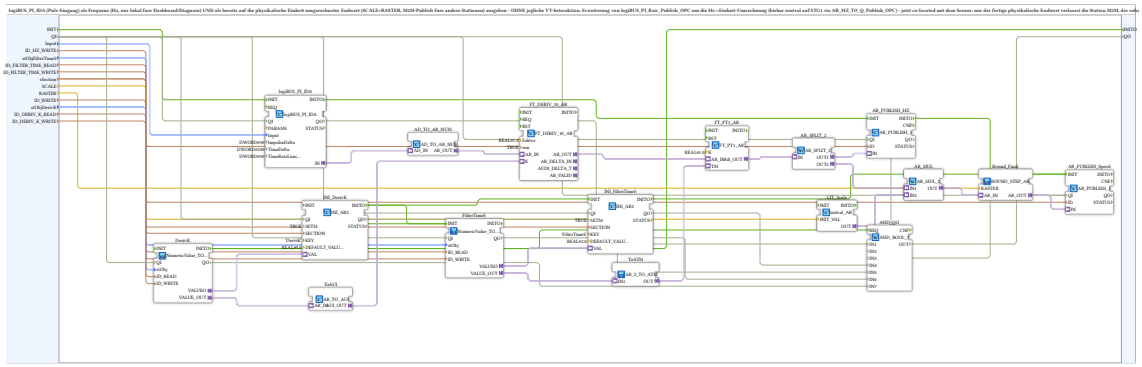
Anwendungsszenarien

- Geschwindigkeits-/Drehzahlsensoren auf Stationen ohne VT, deren Frequenz per OPC-UA zur VT-Station übertragen wird (z. B. ergänzt durch `AR_HZ_TO_Q_Publish_OPC` auf der VT-Station)

Fazit

`logiBUS_PI_Raw_Publish_OPC` erzeugt aus einem Pulseingang eine gefilterte, gerundete Frequenz in Hz und publiziert sie per OPC-UA.

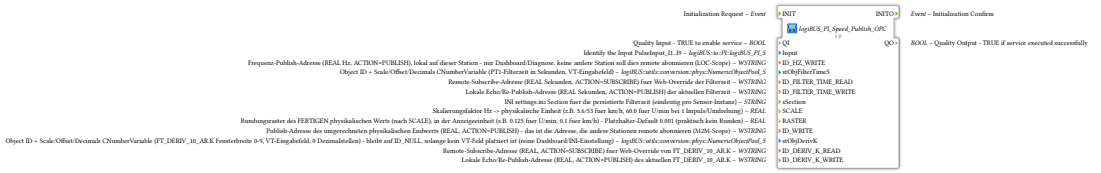
logiBUS_PI_Speed_Publish_OPC



• – ○ • – ○ • – ○ •

Einleitung

`logiBUS_PI_Speed_Publish_OPC` gibt einen Puls-Eingang (`logiBUS_PI_IDA`) auf zwei Wegen aus: als **Frequenz in Hz** (nur lokal für Dashboard/Diagnose) und als **bereits auf die physikalische Einheit umgerechneten Endwert** (SCALE + RASTER, M2M-Publish für andere Stationen). Es gibt **keine** VT-Interaktion. Der Baustein erweitert `logiBUS_PI_Raw_Publish_OPC` um die Hz-zu-Einheit-Umrechnung, die früher zentral auf STG1 lief (`AR_HZ_TO_Q_Publish_OPC`) und jetzt am Sensor liegt: Nur der fertige physikalische Endwert verlässt die Station per M2M, der rohe Hz-Wert bleibt lokal.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialization Request

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialization Confirm

Daten-Eingänge

- **QI** (BOOL, Vorgabe `TRUE`): Quality Input - TRUE, um den Dienst freizugeben
- **Input** (`logiBUS::io::PI::logiBUS_PI_S`): Auswahl des Puls-Eingangs `PulseInput_I1..I9` (Default `logiBUS_PI::Invalid`)

- **ID_HZ_WRITE** (WSTRING): Frequenz-Publish-Adresse (REAL Hz, `ACTION=PUBLISH`), lokal; nur Dashboard/Diagnose, keine andere Station soll sie remote abonnieren (Scope LOC)
- **stObjFilterTimeS** (`NumericObjectPool_S`): Object ID + Skalierung/Offset/Dezimalstellen der CNumberVariable für die PT1-Filterzeit in Sekunden (VT-Eingabefeld); Default `u8Decimals := 1`
- **ID_FILTER_TIME_READ** (WSTRING): Remote-Subscribe-Adresse (REAL Sekunden, `ACTION=SUBSCRIBE`) für einen Web-Override der Filterzeit
- **ID_FILTER_TIME_WRITE** (WSTRING): lokale Echo-/Re-Publish-Adresse (REAL Sekunden, `ACTION=PUBLISH`) der aktuellen Filterzeit
- **sSection** (STRING): INI-Sektion in `settings.ini` für die persistierten Werte (Filterzeit und `DerivK`), eindeutig pro Sensor-Instanz
- **SCALE** (REAL): Skalierungsfaktor Hz ⟶ physikalische Einheit (z. B. `3.6/53` für km/h, `60.0` für U/min bei 1 Impuls/Umdrehung)
- **RASTER** (REAL): Rundungsraster des **fertigen** physikalischen Werts (nach `SCALE`) in der Anzeigeeinheit (z. B. 0.125 für U/min, 0.1 für km/h); Platzhalter-Default `0.001` (praktisch kein Runden)
- **ID_WRITE** (WSTRING): Publish-Adresse des umgerechneten physikalischen Endwerts (REAL, `ACTION=PUBLISH`); diese Adresse abonnieren andere Stationen remote (Scope M2M)
- **stObjDerivK** (`NumericObjectPool_S`): Object ID + Skalierung/Offset/Dezimalstellen der CNumberVariable für `FT_DERIV_10_AR.K` (Fensterbreite 0-9, 0 Dezimalstellen); bleibt auf `ID_NULL`, solange kein VT-Feld platziert ist (reine Dashboard/INI-Einstellung)
- **ID_DERIV_K_READ** (WSTRING): Remote-Subscribe-Adresse (REAL, `ACTION=SUBSCRIBE`) für einen Web-Override von `FT_DERIV_10_AR.K`
- **ID_DERIV_K_WRITE** (WSTRING): lokale Echo-/Re-Publish-Adresse (REAL, `ACTION=PUBLISH`) des aktuellen `FT_DERIV_10_AR.K`

Daten-Ausgänge

- **QO** (BOOL): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde (UND der `QO` von `logiBUS_PI_IDA`, `AR_PUBLISH_HZ`, `AR_PUBLISH_Speed`, `INI_FilterTimeS`, `INI_DerivK`, `FilterTimeS` und `DerivK`; Baustein `AND_QO`, Typ `iec61131::booleanOperators::AND_BOOL_7`)

Adapter

Es werden keine Adapter verwendet.

Funktionsweise

Messkette (Frequenz): `logiBUS_PI_IDA` (`logiBUS::io::PI::logiBUS_PI_IDA`, `QI` vom Baustein-Eingang, `ImpulseDelta = DWORD#50`, `TimeDelta = DWORD#250`, `TimeRateLimit = DWORD#50`) ⟶ `AD_TO_AR_NUM` ⟶ `FT_DERIV_10_AR` (`OSCAT_adapter::Control`, `Faktor = 1.0`, `run = TRUE`) ⟶ `FT_PT1_AR` (`K = 1.0`) ⟶ `AR_SPLIT_2`.

Zwei Ausgabewege aus `AR_SPLIT_2`:

- `OUT1` ⟶ `AR_PUBLISH_HZ` (`AR_PUBLISH_1`, `QI` vom Baustein-Eingang): publiziert die Frequenz in Hz an `ID_HZ_WRITE` (lokal, ungerundet).
- `OUT2` ⟶ `AR_MUL.IN1` (`AR_MUL_2`), `LIT_Scale.OUT` ⟶ `AR_MUL.IN2` (`LIT_Scale` ist ein `initval_AR` mit `INIT_VAL` aus `SCALE`) ⟶ `Round_Final` (`OSCAT_adapter::Mathematical::ROUND_STEP_AR`, `RASTER`) ⟶ `AR_PUBLISH_Speed` (`AR_PUBLISH_1`, `QI` vom Baustein-Eingang): publiziert den umgerechneten, auf `RASTER` gerundeten Endwert an `ID_WRITE`.

Einstellbare PT1-Filterzeit: `FilterTimeS` (`MyLib::sys::NumericValue_TO_AR2_OPC` `stObjFilterTimeS`, `ID_FILTER_TIME_READ`, `ID_FILTER_TIME_WRITE`) liefert die Filterzeit in Sekunden. `FilterTimeS.VALUEO` ↔ `INI_FilterTimeS.VAL` (`eclipse4diac::storage::INI_AR2`, `QI` vom Baustein-Eingang, `SETM = TRUE`, `KEY = 'FilterTimeS'`, `DEFAULT_VALUE = REAL#2.0`, `SECTION` aus `sSection`) persistiert sie. `FilterTimeS.VALUE_OUT` ⟶ `ToATM.IN1` (`MyLib::sys::AR_S_TO_ATM`), `ToATM.OUT` ⟶ `FT_PT1_AR.TM`.

Einstellbare Ableitungs-Fensterbreite: `DerivK` (`NumericValue_TO_AR2_OPC`; `stObjDerivK`, `ID_DERIV_K_READ`, `ID_DERIV_K_WRITE`) liefert `K`. `DerivK.VALUEO` ↔ `INI_DerivK.VAL` (`INI_AR2`, `KEY = 'DerivK'`, `DEFAULT_VALUE = REAL#9.0`, `SECTION` aus `sSection`) persistiert den Wert. `DerivK.VALUE_OUT` ⟶ `ToAUI.AR_IN` (`AR_TO_AUI`), `ToAUI.AUI_OUT` ⟶ `FT_DERIV_10_AR.K`.

Initialisierung und Status: Die Initialisierungskette lautet `INIT` ⟶ `logiBUS_PI_IDA.INIT`, `logiBUS_PI_IDA.INITO` ⟶ `AR_PUBLISH_HZ.INIT`, `AR_PUBLISH_HZ.INITO` ⟶ `AR_PUBLISH_Speed.INIT`, `AR_PUBLISH_Speed.INITO` ⟶ `INI_FilterTimeS.INIT`, `INI_FilterTimeS.INITO` ⟶ `INI_DerivK.INIT`, `INI_DerivK.INITO` ⟶ `FilterTimeS.INIT`, `FilterTimeS.INITO` ⟶ `DerivK.INIT`, `DerivK.INITO` ⟶ `INITO`

und ⟶ `AND_QO.REQ`. `QI` wird an `logiBUS_PI_IDA`, `AR_PUBLISH_HZ`, `AR_PUBLISH_Speed`, `INI_FilterTimeS`, `INI_DerivK`, `FilterTimeS` und `DerivK` weitergegeben; deren `QO` ⟶ `AND_QO.IN1`..`IN7`, `AND_QO.OUT` ⟶ `QO`.

Technische Besonderheiten

- Der Hz-Wert (`ID_HZ_WRITE`) bleibt bewusst ungerundet und rein lokal (Scope LOC); nur `ID_WRITE` ist für M2M-Abonnenten gedacht
- `RASTER` rundet den **fertigen** physikalischen Wert nach der `SCALE`-Multiplikation (nicht mehr in Hz wie früher `RASTER_HZ` bei `logiBUS_PI_Raw_Publish_OPC`), sodass intuitive Werte direkt in der Anzeigeeinheit entstehen
- Filterzeit (`FilterTimeS`) und Fensterbreite (`DerivK`) sind per VT-Feld, Web-Override und INI-Persistenz einstellbar; `K` von `FT_DERIV_10_AR` ist selbst ein AUI-Adapter-Socket, sodass die frühere Entpack-Umgehung entfiel
- Die Konfiguration gilt automatisch für alle fünf Puls-Sensoren, die diesen gemeinsamen Baustein nutzen (Aufnahme B03, Igelband 1/2 B10/B11, Zapfwelle B21, Sägeblatt B08)
- Beide INI-Einträge teilen sich dieselbe `sSection`, besitzen aber unterschiedliche `KEY`s (`FilterTimeS`, `DerivK`)

Anwendungsszenarien

- Geschwindigkeits- oder Drehzahlmessung (km/h, U/min) an Stationen ohne VT-Anschluss mit stationsübergreifender Bereitstellung des fertigen Werts

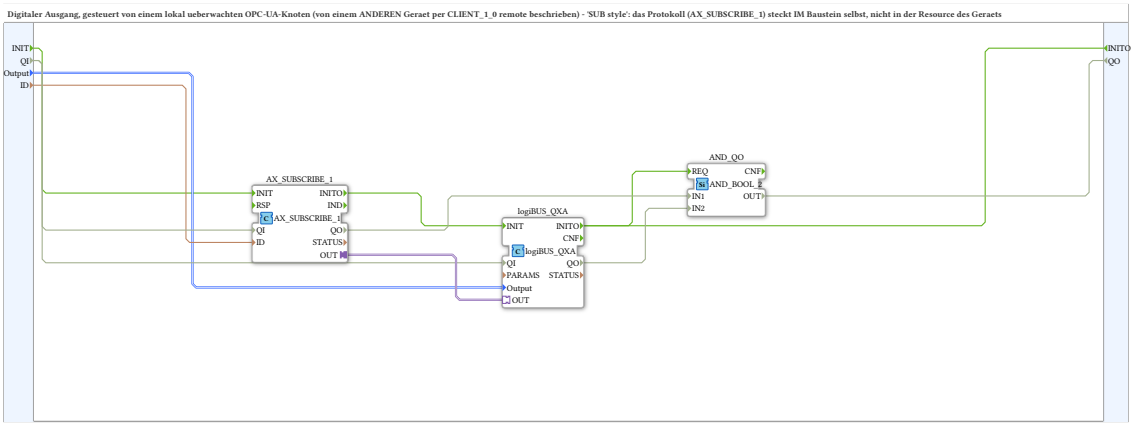
⚖️ Vergleich mit ähnlichen Bausteinen

Gegenüber `logiBUS_PI_Raw_Publish_OPC` rundet dieser Baustein im Anzeigeeinheit-Raster und liefert zusätzlich den umgerechneten Endwert. Der abonnierende Gegenpart ist `AR_M2M_Subscribe_TO_Q_Publish_OPC`.

Fazit

`logiBUS_PI_Speed_Publish_OPC` bildet die komplette Sensor-Seite einer Geschwindigkeits-/Drehzahlmessung ab: Frequenzermittlung mit einstellbarem Filter, Umrechnung und Rundung in die Anzeigeeinheit sowie lokale und M2M-Publikation.

logiBUS_QXA_FROM_SUBSCRIBE_OPC



• – ○ • – ○ • – ○ •

Einleitung

`logiBUS_QXA_FROM_SUBSCRIBE_OPC` steuert einen digitalen Ausgang (`logiBUS_QXA`) von einem lokal ueberwachten OPC-UA-Knoten, der von einem ANDEREN Geraet per `AX_CLIENT_1_0` remote beschrieben wird - „Krauternter-Stil“ (laut Quellcode-Kommentar): das Protokoll (`AX_SUBSCRIBE_1`) steckt IM Baustein selbst, nicht in der Resource des Geraets.

Verwendete Funktionsbausteine (FBs)

- **AX_SUBSCRIBE_1** (`adapter::net::AX_SUBSCRIBE_1`): abonniert die lokal ueberwachte Adresse `ID` (BOOL, remote beschrieben).
- **logiBUS_QXA** (`logiBUS::io::DQ::logiBUS_QXA`): physischer digitaler Ausgang, identifiziert ueber `Output_Q1..Q8`.
- **AND_QO** (`iec61131::booleanOperators::AND_BOOL_2`): verknuepft `AX_SUBSCRIBE_1.QO` und `logiBUS_QXA.QO` zum SubApp-Ausgang `QO`.

Initialisierung und Qualitaet

- Ereignisse: `INIT` (EInit, Eingang) und `INITO` (EInit, Ausgang, kommt von `logiBUS_QXA.INITO`).
- Dateneingang `QI` (BOOL, Vorgabe `TRUE`) und Datenausgang `QO` (BOOL); `AX_SUBSCRIBE_1` und `logiBUS_QXA` tragen kein festes `QI=TRUE` mehr.
- Initialisierung: `INIT` ⟶ `AX_SUBSCRIBE_1.INIT`, `AX_SUBSCRIBE_1.INITO` ⟶ `logiBUS_QXA.INIT`, `logiBUS_QXA.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`.
- Qualitaet: `QI` ⟶ `AX_SUBSCRIBE_1.QI` und `logiBUS_QXA.QI`; `AX_SUBSCRIBE_1.QO` ⟶ `AND_QO.IN1`, `logiBUS_QXA.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`.

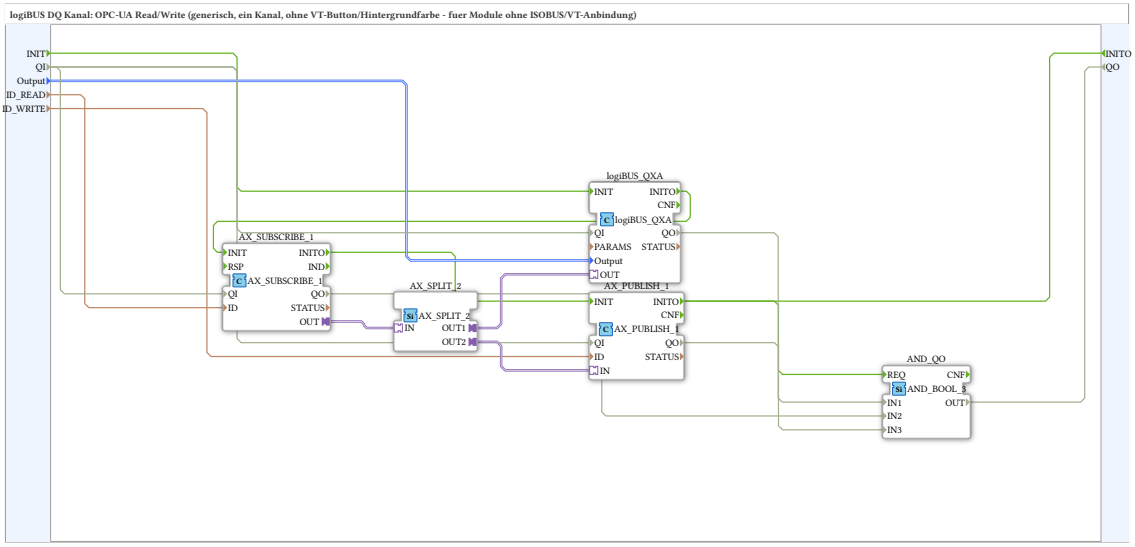
Zusammenfassung

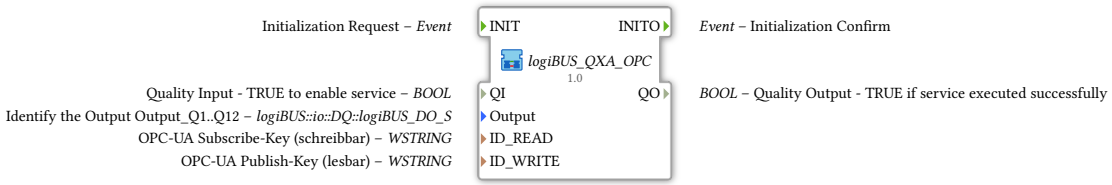
Ein-Baustein-Bruecke von einem lokal abonnierten OPC-UA-Wert zu einem physischen digitalen Ausgang - Gegenstueck zu `logiBUS_IXA_TO_CLIENT_OPC` auf der Sendeseite.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

logiBUS_QXA_OPC





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock (SubApp) `logiBUS_QXA_OPC` stellt einen generischen OPC-UA Read/Write-Kanal für logiBUS-Digitalausgänge (DQ) bereit. Er dient als Schnittstelle zwischen einer OPC-UA-basierten Steuerung und einem logiBUS-Modul, das keinen ISOBUS/VT-Anschluss unterstützt. Die SubApp kapselt die notwendigen Adapter-Logik für das Abonnieren (Subscribe) und Veröffentlichen (Publish) von OPC-UA-Daten und ermöglicht so das Lesen und Schreiben eines einzelnen DQ-Kanals über OPC-UA.

Schnittstellenstruktur

Die SubApp besitzt die Ereignisse `INIT`/`INITO`, die Daten `QI`/`QO` sowie drei weitere Dateneingänge. Es gibt keine Adapter-Anschlüsse an der Schnittstelle. Die eigentlichen Ereignis- und Datenflüsse werden intern über Adapter-Verbindungen realisiert.

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; startet die Init-Kette bei `logiBUS_QXA.INIT`

Sonst werden Ereignisse über die internen Adapter (AX_SUBSCRIBE_1) empfangen.

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung; kommt von `AX_PUBLISH_1.INITO`, dem Ende der Init-Kette

Sonst werden Ereignisse über die internen Adapter (AX_PUBLISH_1) ausgegeben.

Daten-Eingänge

Name	Typ	Initialwert	Kommentar
`QI`	`BOOL`	`TRUE`	Quality Input, schaltet den Dienst frei; wird an `logiBUS_QXA`, `AX_SUBSCRIBE_1` und `AX_PUBLISH_1` weitergereicht
`Output`	`logiBUS::io::DQ::logiBUS_DO_S`	`logiBUS_DO::Invalid`	Identifiziert den Ausgang Q1..Q12
`ID_READ`	`WSTRING`	–	OPC-UA Subscribe-Key (schreibbar) – legt die zu abonnierende OPC-UA-Variable fest
`ID_WRITE`	`WSTRING`	–	OPC-UA Publish-Key (lesbar) – legt die zu beschreibende OPC-UA-Variable fest

Daten-Ausgänge

- **`QO`** (`BOOL`): Quality Output, UND-Verknüpfung (`AND_QO`, `AND_BOOL_3`) der `QO`-Ausgänge von `logiBUS_QXA`, `AX_SUBSCRIBE_1` und `AX_PUBLISH_1`

Die verarbeiteten Daten werden über die internen Adapter und den integrierten FB `logiBUS_QXA` ausgetauscht.

Adapter

An der SubApp-Schnittstelle sind keine Adapter definiert. Intern werden die folgenden Adapter verwendet:

- `AX_SUBSCRIBE_1` (Typ: `adapter::net::AX_SUBSCRIBE_1`) – empfängt OPC-UA-Subscribe-Ereignisse.
- `AX_SPLIT_2` (Typ: `adapter::events::unidirectional::AX_SPLIT_2`) – verteilt eingehende Ereignisse auf zwei Ausgänge.
- `AX_PUBLISH_1` (Typ: `adapter::net::AX_PUBLISH_1`) – sendet OPC-UA-Publish-Ereignisse.

Funktionsweise

Die SubApp realisiert einen bidirektionalen OPC-UA-Zugriff auf einen einzelnen logiBUS-Digitalausgang:

1. **Dateneingänge**: Der gewünschte Ausgang wird über `Output` (Struktur mit Kanal-ID) ausgewählt. Über `ID_READ` wird der OPC-UA-Knoten (Subscribe-Key) spezifiziert, von dem Werte gelesen werden sollen, und über `ID_WRITE` der OPC-UA-Knoten (Publish-Key), an den Werte geschrieben werden sollen.
2. **Subscribe-Pfad**:
   - `AX_SUBSCRIBE_1` empfängt OPC-UA-Datenänderungen gemäß `ID_READ`.
   - Das Ereignis wird über `AX_SPLIT_2` in zwei parallele Ströme aufgeteilt:

   - **OUT1** ⟶ an den internen FB `logiBUS_QXA.OUT` (vermutlich zur Aktualisierung des Ausgangsstatus).
   - **OUT2** ⟶ an `AX_PUBLISH_1.IN` (optional zur Weiterleitung/Rückmeldung).

3. **Publish-Pfad**:
   - `AX_PUBLISH_1` veröffentlicht Daten gemäß `ID_WRITE` an den OPC-UA-Server. Diese Daten stammen aus dem internen FB `logiBUS_QXA`, der den aktuellen Zustand des logiBUS-Ausgangs bereitstellt.
4. **Interner FB `logiBUS_QXA`**:
   - Typ `logiBUS::io::DQ::logiBUS_QXA`
   - Erhält `QI` vom Eingang `QI` und eine leere `PARAMS`.
   - Verknüpft den ausgewählten Ausgang (`Output`) mit der internen Logik zur Steuerung des realen DQ-Kanals.

Initialisierung und Qualität: `INIT` ⟶ `logiBUS_QXA.INIT`, `logiBUS_QXA.INITO` ⟶ `AX_SUBSCRIBE_1.INIT`, `AX_SUBSCRIBE_1.INITO` ⟶ `AX_PUBLISH_1.INIT`, `AX_PUBLISH_1.INITO` ⟶ `INITO` und `AND_QO.REQ`. `QI` wird an `logiBUS_QXA.QI`, `AX_SUBSCRIBE_1.QI` und `AX_PUBLISH_1.QI` weitergereicht; deren `QO`-Ausgänge gehen in dieser Reihenfolge auf `AND_QO.IN1` bis `AND_QO.IN3` (`iec61131::booleanOperators::AND_BOOL_3`), `AND_QO.OUT` ⟶ `QO`.

Die Ereignis- und Datenflüsse sind so verschaltet, dass sowohl das Lesen (Subscribe) als auch das Schreiben (Publish) von OPC-UA-Werten auf den logiBUS-Kanal abgebildet werden. Die SubApp ist bewusst einfach gehalten und verzichtet auf Visualisierungs-Buttons oder Hintergrundfarben, da sie für Module ohne ISOBUS/VT-Anbindung gedacht ist.

Technische Besonderheiten

- **OPC-UA-Anbindung**: Die Kommunikation erfolgt vollständig über die Adapter `AX_SUBSCRIBE_1` und `AX_PUBLISH_1`, die für Netzwerk-Kommunikation ausgelegt sind. Dies ermöglicht eine entkoppelte Integration in OPC-UA-Systeme.
- **Generische Kanalauswahl**: Durch die Struktur `logiBUS_DO_S` kann einer von 12 Ausgängen (Q1..Q12) ausgewählt werden. Der Initialwert `Invalid` verhindert eine versehentliche Aktivierung ohne explizite Konfiguration.
- **Ereignis-Splitting**: Der Adapter `AX_SPLIT_2` teilt jedes eingehende Subscribe-Ereignis in zwei separate Ströme auf. Dies ermöglicht eine parallele Verarbeitung (z.B. Aktualisierung des Ausgangs und Weiterleitung an einen Publish-Kanal).
- **Eingebettete FB-Parameter**: Der interne FB `logiBUS_QXA` wird nicht mehr mit festem `QI=TRUE` betrieben, sondern erhält `QI` vom Eingang `QI`; die Dynamik wird über die Dateneingänge der SubApp gesteuert.
- **Keine grafische Anbindung**: Die SubApp ist bewusst ohne VT-Button und Hintergrundfarbe gestaltet, um Platz zu sparen und die Wiederverwendung in headless-Modulen zu erleichtern.

Zustandsübersicht

Die SubApp selbst besitzt keinen expliziten Zustandsautomaten. Der interne FB `logiBUS_QXA` könnte einen eigenen Zustand besitzen, der jedoch nicht im XML dokumentiert ist. Die SubApp agiert als transparente Verdrahtungsebene zwischen OPC-UA-Adaptern und dem logiBUS-Ausgangs-FB. Der Zustand des Ausgangs wird durch die OPC-UA-Werte und die interne Logik von `logiBUS_QXA` bestimmt.

Anwendungsszenarien

- **Fernsteuerung von logiBUS-Ausgängen über OPC-UA**: Steuerung eines Digitalausgangs (z.B. Schalten einer Pumpe) über einen OPC-UA-Server, ohne direkte SPS-Anbindung.
- **Integration in Gebäudeautomationssysteme**: Anbindung logiBUS-Komponenten an übergeordnete Leitsysteme, die OPC-UA als Kommunikationsprotokoll nutzen.
- **Einfache Datenbrücke**: Verwendung als Baustein zum Testen von OPC-UA-Verbindungen mit einem einzelnen logiBUS-Kanal.
- **Module ohne VT-Anzeige**: Einsatz in Systemen, bei denen keine Visualisierung am Modul selbst gewünscht ist.

Vergleich mit ähnlichen Bausteinen

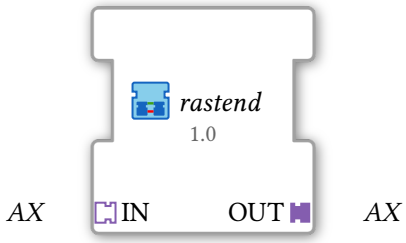
Im Vergleich zu einem direkt mit Ereignis- und Datenanschlüssen versehenen FB bietet `logiBUS_QXA_OPC` eine gekapselte OPC-UA-Anbindung, sodass der Anwender keine detaillierten Adapter-Kenntnisse benötigt. Andere Bausteine könnten zusätzliche Visualisierungsfunktionen oder mehrere Kanäle unterstützen.

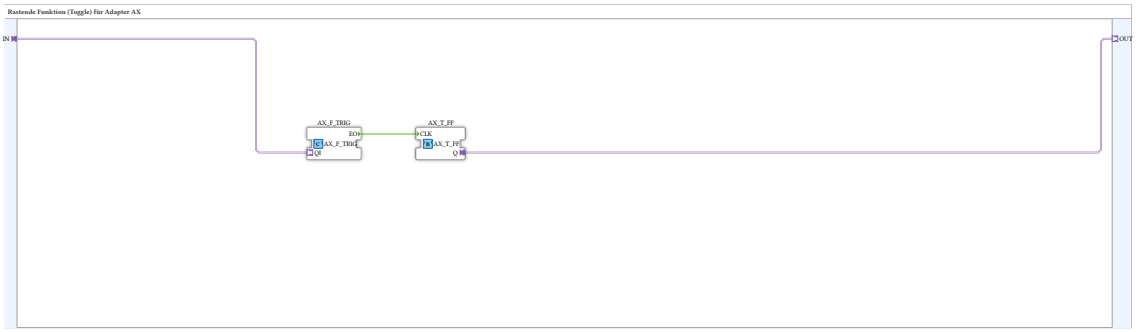
Dieser Baustein konzentriert sich auf einen einzelnen Kanal und ist bewusst schlank gehalten. Er ähnelt einem „Wrapper“ für den Basis-FB `logiBUS_QXA`, ergänzt um OPC-UA-Netzwerkadapter.

Fazit

`logiBUS_QXA_OPC` bietet eine kompakte und flexible Lösung zur Anbindung eines logiBUS-Digitalausgangs an ein OPC-UA-System. Durch die interne Verdrahtung von Subscribe- und Publish-Adaptern wird die bidirektionale Kommunikation vereinfacht. Die minimale Schnittstellenstruktur (Init-Ereignisse, `QI`/`QO` und drei Dateneingänge) erleichtert die Integration in bestehende Steuerungslogik. Der Baustein ist ideal für Anwendungen, bei denen eine direkte OPC-UA-Anbindung ohne zusätzliche Visualisierung benötigt wird.

Rastend





• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung implementiert eine rastende Schaltfunktion, die beim Loslassen eines Signals gestartet wird. Die Subapplikation verwendet Adapter für die Signalverarbeitung und Ereignissteuerung, um eine zuverlässige Rastfunktionalität zu realisieren.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: rastend

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **E_T_FF**: iec61499::events::E_T_FF
    - Ereigniseingang: CLK
    - Ereignisausgang: EO
    - Datenausgang: Q
  - **E_SWITCH**: iec61499::events::E_SWITCH
    - Ereigniseingang: EI
    - Ereignisausgang: EO0
    - Dateneingang: G
  - **AX_X_TO_BOOL**: adapter::conversion::unidirectional::AX_X_TO_BOOL
    - Adaptereingang: AX_IN
    - Ereignisausgang: CNF
    - Dateneingang: IN
  - **AX_BOOL_TO_X**: adapter::conversion::unidirectional::AX_BOOL_TO_X
    - Ereigniseingang: REQ

  - Adapterausgang: AX_OUT
  - Dateneingang: OUT

- **Funktionsweise**: Die Subapplikation wandelt eingehende AX-Signale in boolesche Werte um, verarbeitet diese über einen Toggle-Flipflop mit Schalterfunktion und gibt das Ergebnis wieder als AX-Signal aus.

Programmablauf und Verbindungen

Der Programmablauf folgt dieser Signalverarbeitungskette:

1. **Eingangssignalverarbeitung**: Das eingehende AX-Signal wird über den Adapter IN an AX_X_TO_BOOL weitergeleitet
2. **Signalumwandlung**: AX_X_TO_BOOL konvertiert das AX-Signal in einen booleschen Wert
3. **Schaltersteuerung**: Der boolesche Wert steuert den E_SWITCH, der Ereignisse an den Toggle-Flipflop weiterleitet
4. **Rastfunktion**: E_T_FF realisiert die eigentliche Rastfunktion und speichert den Zustand
5. **Ausgangssignalgenerierung**: AX_BOOL_TO_X wandelt den booleschen Ausgangswert zurück in ein AX-Signal

Verbindungen im Detail:

- Ereignisverbindungen: E_SWITCH.EO0 → E_T_FF.CLK, E_T_FF.EO → AX_BOOL_TO_X.REQ, AX_X_TO_BOOL.CNF → E_SWITCH.EI
- Datenverbindungen: E_T_FF.Q → AX_BOOL_TO_X.OUT, AX_X_TO_BOOL.IN → E_SWITCH.G
- Adapterverbindungen: IN → AX_X_TO_BOOL.AX_IN, AX_BOOL_TO_X.AX_OUT → OUT

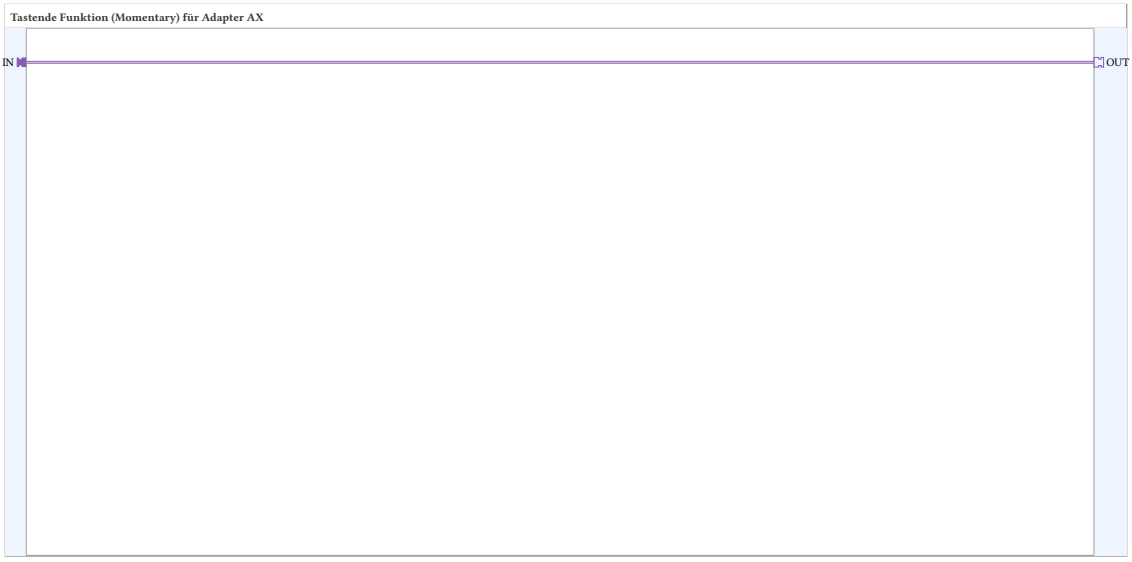
Zusammenfassung

Diese Übung demonstriert die Implementierung einer rastenden Schaltfunktion mit Adaptern für Signalumwandlung. Sie zeigt die Verwendung von Toggle-Flipflops für Zustandsspeicherung und die strukturierte Signalverarbeitung über verschiedene Funktionsbausteine. Die Lösung ist besonders geeignet für Anwendungen, bei denen ein Signal nach dem Loslassen aktiv bleiben soll.

🛠️ Zugehörige Übungen

- Uebung_103
- Uebung_103c
- Uebung_103c2

Tastend



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung beschäftigt sich mit der Erstellung und Verwendung eines Sub-Applikationstyps in 4diac-IDE. Der Funktionsbaustein „tastend“ dient als Sub-Applikation und implementiert eine unidirektionale Adapterverbindung zwischen Eingangs- und Ausgangsadaptern.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: tastend

- **Typ**: SubAppType
- **Verwendete interne FBs**: Keine internen FBs vorhanden
- **Funktionsweise**: Der Sub-Baustein „tastend“ stellt eine Adapterverbindung zwischen einem Eingangs-Adapter (IN) und einem Ausgangs-Adapter (OUT) her. Beide Adapter verwenden den Typ „adapter::types::unidirectional::AX“ und ermöglichen eine unidirektionale Datenübertragung.

Programmablauf und Verbindungen

Der Sub-Baustein „tastend“ verfügt über folgende Schnittstellen:

- **Eingangsadapter**: IN (Typ: adapter::types::unidirectional::AX)
- **Ausgangsadapter**: OUT (Typ: adapter::types::unidirectional::AX)

Die Verbindung erfolgt direkt vom Eingangsadapter IN zum Ausgangsadapter OUT. Diese Konfiguration ermöglicht eine einfache Weiterleitung von Signalen oder Daten durch den Sub-Baustein.

Schwierigkeitsgrad: Einfach **Benötigte Vorkenntnisse**: Grundkenntnisse in 4diac-IDE, Verständnis von Adapterverbindungen und Sub-Applikationen **Start der Übung**: Der Sub-Baustein kann in größere Applikationen integriert und über die definierten Adapterschnittstellen angesteuert werden.

Zusammenfassung

Die Übung „tastend“ demonstriert die Erstellung eines einfachen Sub-Applikationstyps mit Adapterverbindungen. Sie zeigt die grundlegende Struktur von Sub-Bausteinen in 4diac-IDE und deren Integration in größere Steuerungsapplikationen durch standardisierte Adapterschnittstellen.

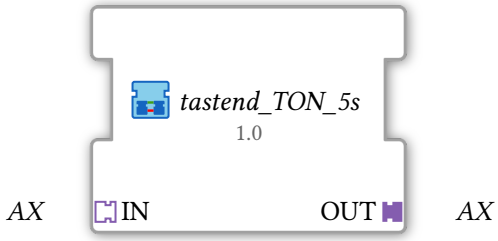
🛠️ Zugehörige Übungen

- Uebung_103
- Uebung_103c
- Uebung_103c2

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

tastend_TON_5s



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert die Verwendung eines Zeitrelais (Timer On-Delay) mit einer festen Zeitvorgabe von 5 Sekunden. Die Subapplikation dient als wiederverwendbare Komponente zur Signalverarbeitung mit Zeitverzögerung.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: tastend_TON_5s

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **AX_X_TO_BOOL**: adapter::conversion::unidirectional::AX_X_TO_BOOL
    - Parameter: Keine
    - Ereignisausgang/-eingang: CNF (Ausgang), REQ (Eingang)
    - Datenausgang/-eingang: IN (Eingang), OUT (Ausgang)
  - **E_TON**: iec61499::events::timers::E_TON
    - Parameter: PT = t#5s
    - Ereignisausgang/-eingang: CNF (Ausgang), REQ (Eingang)
    - Datenausgang/-eingang: IN (Eingang), Q (Ausgang)
  - **AX_BOOL_TO_X**: adapter::conversion::unidirectional::AX_BOOL_TO_X
    - Parameter: Keine
    - Ereignisausgang/-eingang: REQ (Eingang), CNF (Ausgang)
    - Datenausgang/-eingang: OUT (Eingang), AX_OUT (Ausgang)
- **Funktionsweise**: Die Subapplikation wandelt ein eingehendes Adaptersignal (IN) über AX_X_TO_BOOL in einen Booleschen Wert um, der dann an den E_TON Timer weitergeleitet wird. Der Timer wartet 5 Sekunden, bevor er das Ausgangssignal über Q aktiviert. Dieses Signal wird anschließend über AX_BOOL_TO_X zurück in ein Adaptersignal (OUT) konvertiert.

Programmablauf und Verbindungen

Adapterverbindungen:

- Eingang: IN ⟶ AX_X_TO_BOOL.AX_IN
- Ausgang: AX_BOOL_TO_X.AX_OUT ⟶ OUT

Ereignisverbindungen:

- AX_X_TO_BOOL.CNF ⟶ E_TON.REQ
- E_TON.CNF ⟶ AX_BOOL_TO_X.REQ

Datenverbindungen:

- AX_X_TO_BOOL.IN ⟶ E_TON.IN
- E_TON.Q ⟶ AX_BOOL_TO_X.OUT

Lernziele:

- Verständnis von Timer-Funktionsbausteinen (E_TON)
- Umgang mit Adapter-Konvertierungsbausteinen
- Aufbau von wiederverwendbaren Subapplikationen
- Zeitgesteuerte Signalverarbeitung

Schwierigkeitsgrad: Einsteiger

Benötigte Vorkenntnisse: Grundlagen der 4diac-IDE, Verständnis von Funktionsbausteinen und Adaptern

Start der Übung: Die Subapplikation kann in größere Applikationen integriert werden, wo eine 5-Sekunden-Verzögerung benötigt wird.

Zusammenfassung

Die Subapplikation tastend_TON_5s stellt eine kompakte Lösung für zeitverzögerte Signalverarbeitung dar. Durch die feste Zeitvorgabe von 5 Sekunden und die Verwendung standardisierter Adapter-Schnittstellen kann sie einfach in verschiedene Steuerungsanwendungen integriert werden. Die klare Struktur mit Signalwandlung, Timer-Funktionalität und Rückwandlung macht die Funktionsweise transparent und nachvollziehbar.

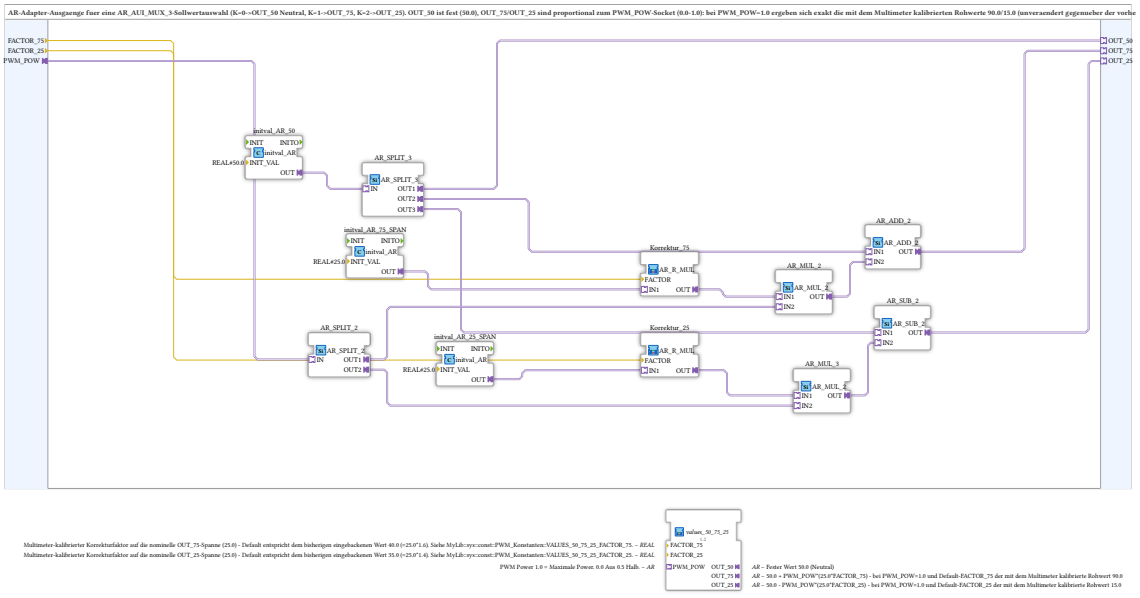
🛠️ Zugehörige Übungen

- Uebung_103
- Uebung_103c

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

values_50_75_25



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `values_50_75_25` ist eine Subapplikation (SubApp), die drei feste reelle Konstanten als Ausgangswerte über unidirektionale AR-Adapter bereitstellt. Sie wurde entwickelt, um einem übergeordneten Funktionsblock (z. B. einem `AR_AUI_MUX_3`) die Sollwerte 50, 75 und 25 zur Auswahl bereitzustellen – gesteuert über einen Kanalindex `K` (0 ⟶ 50, 1 ⟶ 75, 2 ⟶ 25).

Schnittstellenstruktur

Die SubApp besitzt keinerlei Ereignis- oder Dateneingänge. Ihre gesamte Funktionalität wird ausschließlich über **drei Adapter-Ausgänge** realisiert.

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

Name	Typ	Richtung	Kommentar
`OUT_50`	`adapter::types::unidirectional::AR`	Ausgang	Fester Wert 50.0
`OUT_75`	`adapter::types::unidirectional::AR`	Ausgang	Fester Wert 75.0
`OUT_25`	`adapter::types::unidirectional::AR`	Ausgang	Fester Wert 25.0

Funktionsweise

Die SubApp verwendet intern drei Instanzen des Bausteins `adapter::types::unidirectional::AR::initval::initval_AR`. Jede Instanz ist über den Parameter `INIT_VAL` mit einem festen `REAL`-Wert (50, 75 bzw. 25) parametrisiert. Die Ausgänge dieser Instanzen werden direkt auf die entsprechenden Adapter-Ausgänge `OUT_50`, `OUT_75` und `OUT_25` gelegt. Dadurch liefert die SubApp permanent drei konstante Real-Werte, ohne dass eine externe Ansteuerung oder Dateneingabe erforderlich ist.

Technische Besonderheiten

- Verwendung von **unidirektionalen AR-Adaptern** – diese dienen ausschließlich als Ausgangsschnittstellen.
- Die Werte sind als **REAL**-Konstanten fest im Baustein verdrahtet und können nur durch Änderung der `INIT_VAL`-Parameter in den internen FB-Instanzen angepasst werden.
- Es sind keinerlei Ereignisse oder Steuersignale vorhanden – die Ausgänge sind sofort gültig und statisch.
- Die SubApp ist für den Einsatz in einem Sollwert-Multiplexer (`AR_AUI_MUX_3`) ausgelegt, der diese Werte über einen Kanalindex auswählt.

Zustandsübersicht

Da die SubApp weder Ereignis- verarbeitende Zustände noch sonstige Logik besitzt, existiert kein impliziter Zustandsautomat. Sie arbeitet rein kombinatorisch und gibt permanent die festen Werte aus.

Anwendungsszenarien

Die SubApp eignet sich überall dort, wo drei feste Sollwerte – z. B. für eine Prozessregelung, einen Wahlschalter oder eine Umschaltung – benötigt werden. Häufig wird sie zusammen mit einem Multiplexer eingesetzt, um abhängig von einem Steuersignal `K` den geeigneten der drei Werte auszuwählen. Das konkrete Beispiel hier ist die Nutzung mit einem `AR_AUI_MUX_3`, wobei `K=0` den Wert 50, `K=1` den Wert 75 und `K=2` den Wert 25 liefert.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu generischen Konstante-Bausteinen (z. B. einzelnen `CONSTANT`-FBs) bündelt diese SubApp drei Konstanten in einer einzigen Komponente und stellt sie über standardisierte AR-Adapter bereit. Dadurch wird die Verdrahtung im übergeordneten Netz vereinfacht. Gegenüber einer wiederverwendbaren parametrisierten SubApp, die Werte per Eingabe setzt, ist `values_50_75_25` bewusst fix konfiguriert und erfordert keine externe Parametrisierung zur Laufzeit.

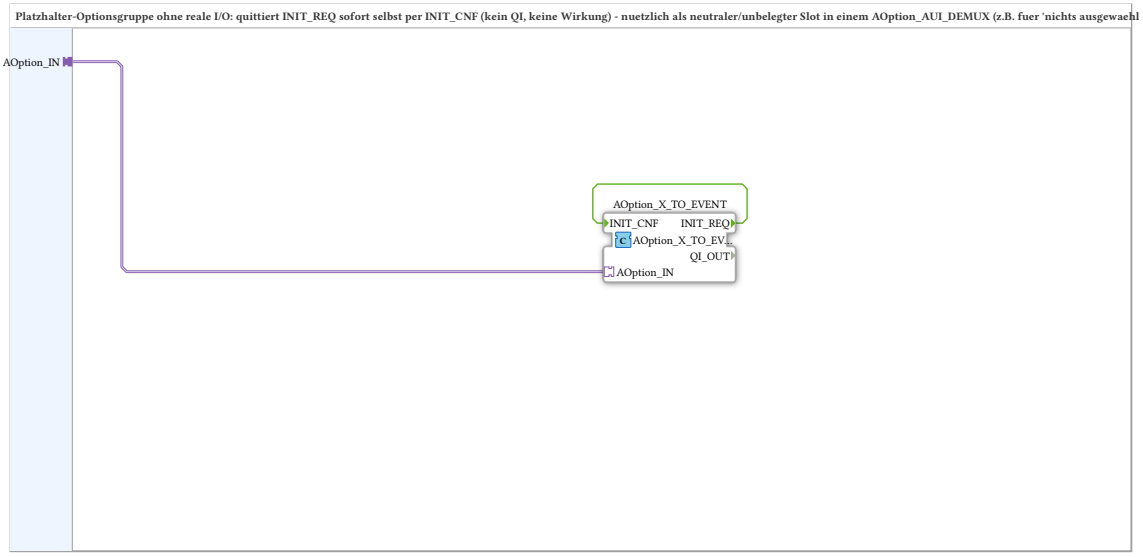
Fazit

Die SubApp `values_50_75_25` ist eine einfache und zweckmäßige Lösung, um drei feste Real-Sollwerte in einem modularen Funktionsblock-System bereitzustellen. Ihre klare, ereignislose Schnittstelle und die direkte Verwendung von AR-Adaptern machen sie zu einem robusten Baustein für Multiplexer-Anwendungen und andere Szenarien mit festen Konstanten.

2.1.5 MyLib_AX_Option

2.1.5.1 sys

DummyGroup



• – ◦ • – ◦ • – ◦ •

Einleitung

`DummyGroup` ist eine Platzhalter-Optionsgruppe ohne reale E/A. Sie quittiert `INIT_REQ` sofort selbst per `INIT_CNF` (kein `QI`-Einfluss, keine Wirkung). Sie dient als neutraler, unbelegter Slot in einem `AOption_AUI_DEMUX`, z. B. für „nichts ausgewählt“ bei `K=1`, wenn echte Optionen erst ab `K=2` beginnen. So entfällt ein Override-Literal für „welcher Zahlenwert bedeutet Aus“.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: DummyGroup

- **Typ**: SubAppType
- **Schnittstelle**: Socket `AOption_IN` vom Typ `Option::AOption`.
- **Verwendete interne FBs**:

– **AOption_X_TO_EVENT**: `Option::conversion::AOption_X_TO_EVENT`.

Programmablauf und Verbindungen

- `AOption_IN` -> `AOption_X_TO_EVENT.AOption_IN`
- `AOption_X_TO_EVENT.INIT_REQ` -> `AOption_X_TO_EVENT.INIT_CNF` (Selbstquittung)

`QI_OUT` bleibt unbenutzt.

Technische Besonderheiten

- Keine E/A-Bausteine, keine Wirkung auf die Anwendung.

Anwendungsszenarien

- Neutraler Slot in einem Option-DEMUX, z. B. `AOption_AUI_DEMUX_n`.

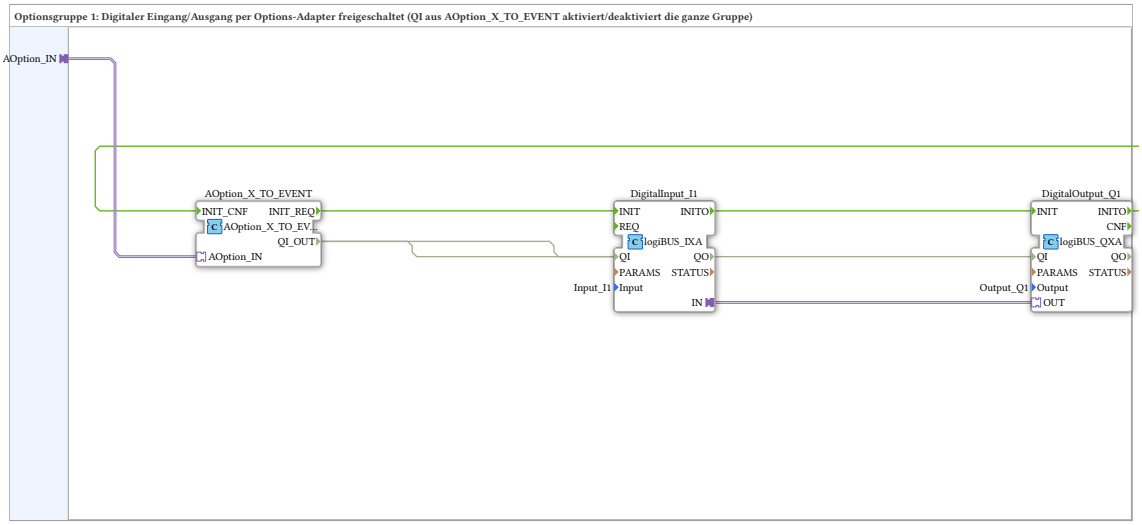
Zusammenfassung

`DummyGroup` ist eine wirkungslose Optionsgruppe, die Anforderungen sofort bestätigt.

🛠️ Zugehörige Übungen

- Uebung_094b2_AX
- Uebung_094b3_AX
- Uebung_094b3b_AX

OptionalGroup_001



• – ○ • – ○ • – ○ •

Einleitung

`OptionalGroup_001` ist eine SubApp (Bibliothek `MyLib_AX_Option`): Optionsgruppe 1, deren digitaler Eingang `Input_I1` und digitaler Ausgang `Output_Q1` (logiBUS) per Options-Adapter freigeschaltet werden. Das `QI` aus `AOption_X_TO_EVENT` aktiviert bzw. deaktiviert die ganze Gruppe.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: OptionalGroup_001

- **Typ**: SubAppType
- **Schnittstelle**: Socket `AOption_IN` vom Typ `Option::AOption`.
- **Verwendete interne FBs**:
  - **AOption_X_TO_EVENT**: `Option::conversion::AOption_X_TO_EVENT`, setzt den Adapter in `INIT_REQ/QI_OUT/INIT_CNF` um.
  - **DigitalInput_I1**: `logiBUS::io::DI::logiBUS_IXA`, Parameter `Input = Input_I1`.
  - **DigitalOutput_Q1**: `logiBUS::io::DQ::logiBUS_QXA`, Parameter `Output = Output_Q1`.

Programmablauf und Verbindungen

- `AOption_IN` -> `AOption_X_TO_EVENT.AOption_IN`
- Init-Kette: `AOption_X_TO_EVENT.INIT_REQ -> DigitalInput_I1.INIT; DigitalInput_I1.INITO -> DigitalOutput_Q1.INIT; DigitalOutput_Q1.INITO -> AOption_X_TO_EVENT.INIT_CNF`
- `AOption_X_TO_EVENT.QI_OUT` -> `DigitalInput_I1.QI` und `DigitalOutput_Q1.QI`
- Adapterverbindung `DigitalInput_I1.IN` -> `DigitalOutput_Q1.OUT`: der Eingang wird auf den Ausgang durchgeschaltet.

Technische Besonderheiten

- Die Initialisierung (Eingang, dann Ausgang) wird erst dann an den Plug quittiert, wenn beide Bausteine fertig sind.
- Das `QI` der Gruppe wird an `QI` von Eingang und Ausgang weitergegeben.

Anwendungsszenarien

- Optionaler digitaler Ein-/Ausgang, der über einen Auswahlbaustein wie `AOption_AD_DEMUX_2` zu- oder abgeschaltet wird.

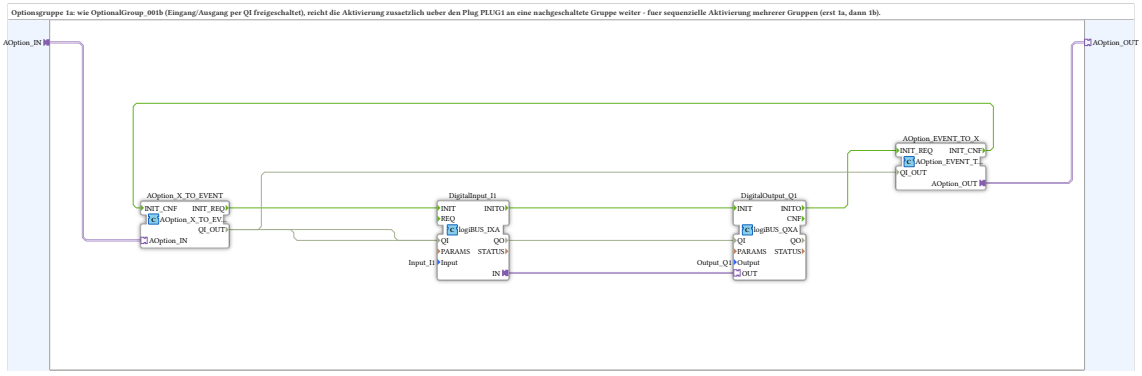
Zusammenfassung

`OptionalGroup_001` bündelt `Input_I1` und `Output_Q1` zu einer per `AOption` ein- und ausschaltbaren Optionsgruppe.

🛠️ Zugehörige Übungen

- Uebung_094b2_AX
- Uebung_094b3_AX
- Uebung_094b3b_AX
- Uebung_094b4_AX
- Uebung_094b4b_AX
- Uebung_094b5_AX
- Uebung_094b5b_AX
- Uebung_094b_AX
- Uebung_094c2_AX
- Uebung_094c2b_AX
- Uebung_094c_AX

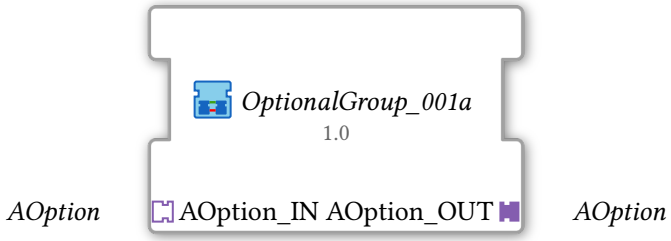
OptionalGroup_001a



• – ◦ • – ◦ • – ◦ •

Einleitung

`OptionalGroup_001a` ist eine SubApp (Bibliothek `MyLib_AX_Option`): Optionsgruppe 1a. Wie `OptionalGroup_001b` werden digitaler Eingang `Input_I1` und digitaler Ausgang `Output_Q1` (logiBUS) per Options-Adapter freigeschaltet. Zusätzlich reicht die Gruppe die Aktivierung über den Plug `AOption_OUT` an eine nachgeschaltete Gruppe weiter - für die sequenzielle Aktivierung mehrerer Gruppen (erst 1a, dann 1b).



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: OptionalGroup_001a

- **Typ**: SubAppType
- **Schnittstelle**: Socket `AOption_IN` vom Typ `Option::AOption` (Adapter-Eingang) und Plug `AOption_OUT` vom Typ `Option::AOption` (Adapter-Ausgang).
- **Verwendete interne FBs**:
  - **AOption_X_TO_EVENT**: `Option::conversion::AOption_X_TO_EVENT`, setzt den Adapter am Socket in `INIT_REQ/QI_OUT/INIT_CNF` um.
  - **DigitalInput_I1**: `logiBUS::io::DI::logiBUS_IXA`, Parameter `Input = Input_I1`.
  - **DigitalOutput_Q1**: `logiBUS::io::DQ::logiBUS_QXA`, Parameter `Output = Output_Q1`.
  - **AOption_EVENT_TO_X**: `Option::conversion::AOption_EVENT_TO_X`, setzt `INIT_REQ/QI_OUT` in den Adapter am Plug `AOption_OUT` um und meldet `INIT_CNF` zurück.

Programmablauf und Verbindungen

- `AOption_IN` -> `AOption_X_TO_EVENT.AOption_IN`
- Init-Kette: `AOption_X_TO_EVENT.INIT_REQ -> DigitalInput_I1.INIT; DigitalInput_I1.INITO -> DigitalOutput_Q1.INIT; DigitalOutput_Q1.INITO -> AOption_EVENT_TO_X.INIT_REQ; AOption_EVENT_TO_X.INIT_CNF -> AOption_X_TO_EVENT.INIT_CNF`
- `AOption_X_TO_EVENT.QI_OUT -> DigitalInput_I1.QI, DigitalOutput_Q1.QI` und `AOption_EVENT_TO_X.QI_OUT`
- `AOption_EVENT_TO_X.AOption_OUT` -> `AOption_OUT`
- Adapterverbindung `DigitalInput_I1.IN -> DigitalOutput_Q1.OUT`: der Eingang wird auf den Ausgang durchgeschaltet.

Technische Besonderheiten

- `AOption_OUT` wird erst ausgelöst, wenn `DigitalOutput_Q1` selbst initialisiert ist (`DigitalOutput_Q1.INITO` -> `AOption_EVENT_TO_X.INIT_REQ`).
- Dessen `INIT_CNF` wird an den eigenen Socket `AOption_IN` zurückgemeldet. Der aufrufende DEMUX sieht die Gruppe daher erst als fertig, wenn die gesamte Kette (inklusive nachgeschalteter Gruppen an `AOption_OUT`) bestätigt hat.
- Das `QI` der Gruppe wird an Eingang, Ausgang und den Plug `AOption_OUT` weitergegeben.

Anwendungsszenarien

- Erstes Glied einer Daisy-Chain: `AOption_OUT` wird mit dem Socket einer nachgeschalteten Gruppe wie `OptionalGroup_001b` verbunden, sodass beide Gruppen nacheinander (erst a, dann b) aktiviert werden.

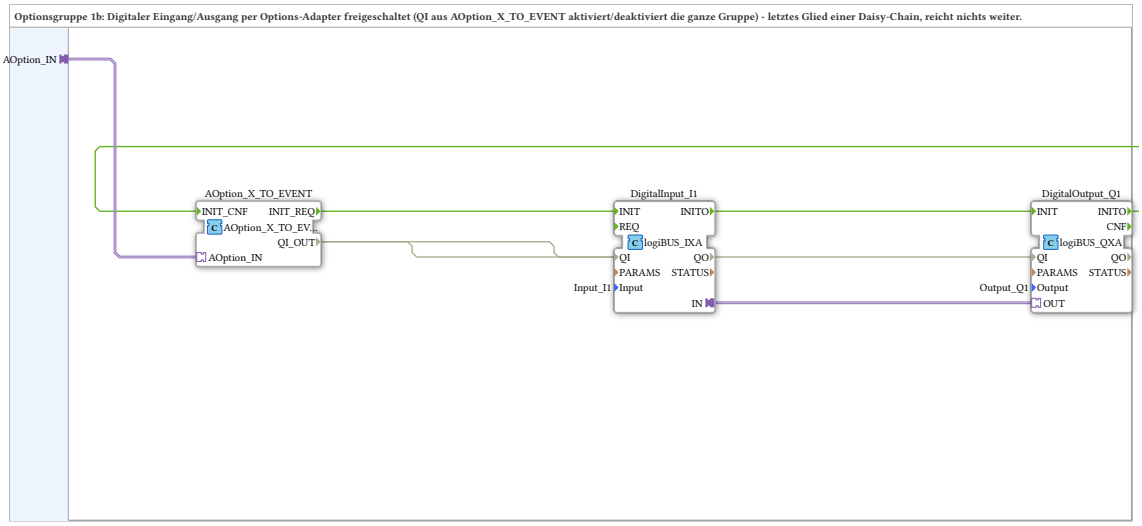
Zusammenfassung

`OptionalGroup_001a` bündelt `Input_I1` und `Output_Q1` zu einer per `AOption` schaltbaren Optionsgruppe und reicht die Aktivierung über `AOption_OUT` an die nächste Gruppe weiter.

🛠️ Zugehörige Übungen

- Uebung_094b3c_AX

OptionalGroup_001b



• – ◦ • – ◦ • – ◦ •

Einleitung

`OptionalGroup_001b` ist eine SubApp (Bibliothek `MyLib_AX_Option`): Optionsgruppe 1b, deren digitaler Eingang `Input_I1` und digitaler Ausgang `Output_Q1` (logiBUS) per Options-Adapter freigeschaltet werden. Das `QI` aus `AOption_X_TO_EVENT` aktiviert bzw. deaktiviert die ganze Gruppe. Sie ist das letzte Glied einer Daisy-Chain und reicht nichts weiter.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: OptionalGroup_001b

- **Typ**: SubAppType
- **Schnittstelle**: Socket `AOption_IN` vom Typ `Option::AOption`.
- **Verwendete interne FBs**:
  - **AOption_X_TO_EVENT**: `Option::conversion::AOption_X_TO_EVENT`, setzt den Adapter in `INIT_REQ`/`QI_OUT`/`INIT_CNF` um.
  - **DigitalInput_I1**: `logiBUS::io::DI::logiBUS_IXA`, Parameter `Input = Input_I1`.
  - **DigitalOutput_Q1**: `logiBUS::io::DQ::logiBUS_QXA`, Parameter `Output = Output_Q1`.

Programmablauf und Verbindungen

- `AOption_IN` -> `AOption_X_TO_EVENT.AOption_IN`
- Init-Kette: `AOption_X_TO_EVENT.INIT_REQ -> DigitalInput_I1.INIT; DigitalInput_I1.INITO -> DigitalOutput_Q1.INIT; DigitalOutput_Q1.INITO -> AOption_X_TO_EVENT.INIT_CNF`
- `AOption_X_TO_EVENT.QI_OUT` -> `DigitalInput_I1.QI` und `DigitalOutput_Q1.QI`
- Adapterverbindung `DigitalInput_I1.IN -> DigitalOutput_Q1.OUT`: der Eingang wird auf den Ausgang durchgeschaltet.

Technische Besonderheiten

- Baugleich zu `OptionalGroup_001a`, aber ohne Plug `AOption_OUT`.
- Die Initialisierung (Eingang, dann Ausgang) wird erst dann an den Socket quittiert, wenn beide Bausteine fertig sind.

Anwendungsszenarien

- Wird typischerweise an `AOption_OUT` einer vorgeschalteten `OptionalGroup_001a` angehängt, damit beide Gruppen nacheinander (erst a, dann b) aktiviert werden, bevor der ursprüngliche Aufrufer seine Bestätigung erhält.

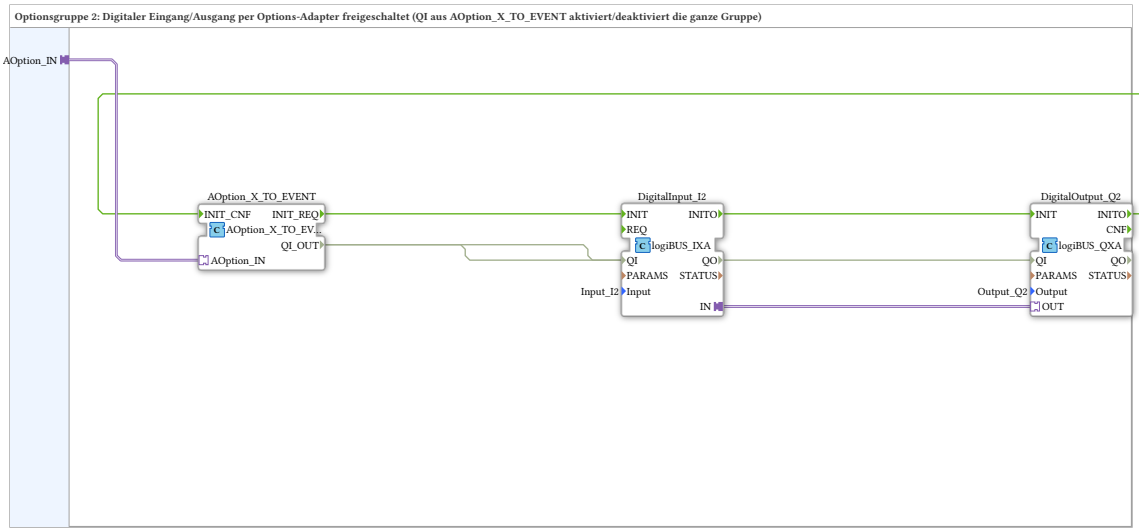
Zusammenfassung

`OptionalGroup_001b` bündelt `Input_I1` und `Output_Q1` zu einer per `AOption` schaltbaren Optionsgruppe und bildet das Ende einer Daisy-Chain.

🛠️ Zugehörige Übungen

- Uebung_094b3c_AX

OptionalGroup_002



• – ○ • – ○ • – ○ •

Einleitung

`OptionalGroup_002` ist eine SubApp (Bibliothek `MyLib_AX_Option`): Optionsgruppe 2, deren digitaler Eingang `Input_I2` und digitaler Ausgang `Output_Q2` (logiBUS) per Options-Adapter freigeschaltet werden. Das `QI` aus `AOption_X_TO_EVENT` aktiviert bzw. deaktiviert die ganze Gruppe.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: OptionalGroup_002

- **Typ**: SubAppType
- **Schnittstelle**: Socket `AOption_IN` vom Typ `Option::AOption`.
- **Verwendete interne FBs**:

- **AOption_X_TO_EVENT**: `Option::conversion::AOption_X_TO_EVENT`, setzt den Adapter in `INIT_REQ/QI_OUT/INIT_CNF` um.
- **DigitalInput_I2**: `logiBUS::io::DI::logiBUS_IXA`, Parameter `Input = Input_I2`.
- **DigitalOutput_Q2**: `logiBUS::io::DQ::logiBUS_QXA`, Parameter `Output = Output_Q2`.

Programmablauf und Verbindungen

- `AOption_IN` -> `AOption_X_TO_EVENT.AOption_IN`
- Init-Kette: `AOption_X_TO_EVENT.INIT_REQ -> DigitalInput_I2.INIT; DigitalInput_I2.INITO -> DigitalOutput_Q2.INIT; DigitalOutput_Q2.INITO -> AOption_X_TO_EVENT.INIT_CNF`
- `AOption_X_TO_EVENT.QI_OUT` -> `DigitalInput_I2.QI` und `DigitalOutput_Q2.QI`
- Adapterverbindung `DigitalInput_I2.IN -> DigitalOutput_Q2.OUT`: der Eingang wird auf den Ausgang durchgeschaltet.

Technische Besonderheiten

- Die Initialisierung (Eingang, dann Ausgang) wird erst dann an den Plug quittiert, wenn beide Bausteine fertig sind.
- Das `QI` der Gruppe wird an `QI` von Eingang und Ausgang weitergegeben.

Anwendungsszenarien

- Optionaler digitaler Ein-/Ausgang, der über einen Auswahlbaustein wie `AOption_AD_DEMUX_2` zu- oder abgeschaltet wird.

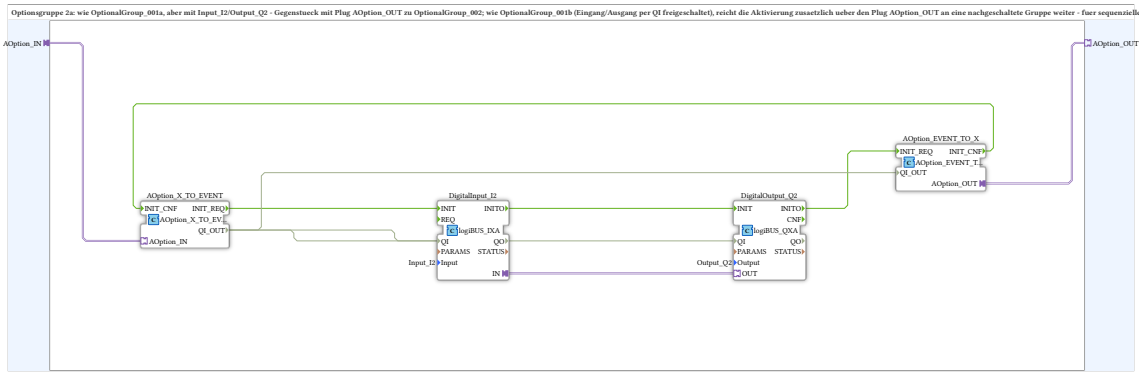
Zusammenfassung

`OptionalGroup_002` bündelt `Input_I2` und `Output_Q2` zu einer per `AOption` ein- und ausschaltbaren Optionsgruppe.

🛠️ Zugehörige Übungen

- Uebung_094b2_AX
- Uebung_094b3_AX
- Uebung_094b3b_AX
- Uebung_094b4_AX
- Uebung_094b4b_AX
- Uebung_094b5_AX
- Uebung_094b5b_AX
- Uebung_094b_AX
- Uebung_094c2_AX
- Uebung_094c2b_AX
- Uebung_094c_AX

OptionalGroup_002a

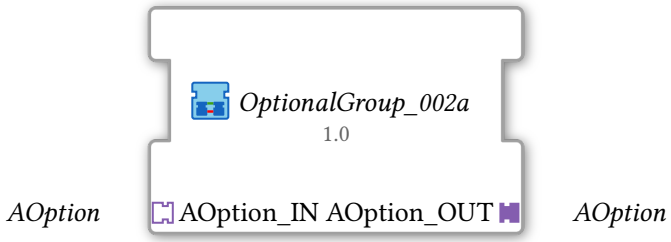


• – ○ • – ○ • – ○ •

Einleitung

`OptionalGroup_002a` ist eine SubApp (Bibliothek `MyLib_AX_Option`): Optionsgruppe 2a. Wie `OptionalGroup_001b` werden digitaler Eingang `Input_I2` und digitaler Ausgang `Output_Q2` (logiBUS) per Options-Adapter freigeschaltet. Zusätzlich reicht die Gruppe die Aktivierung über den Plug `AOption_OUT`

an eine nachgeschaltete Gruppe weiter - für die sequenzielle Aktivierung mehrerer Gruppen (erst 2a, dann die nachgeschaltete Gruppe).



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: OptionalGroup_002a

- **Typ**: SubAppType
- **Schnittstelle**: Socket `AOption_IN` vom Typ `Option::AOption` (Adapter-Eingang) und Plug `AOption_OUT` vom Typ `Option::AOption` (Adapter-Ausgang).
- **Verwendete interne FBs**:
  - **AOption_X_TO_EVENT**: `Option::conversion::AOption_X_TO_EVENT`, setzt den Adapter am Socket in `INIT_REQ/QI_OUT/INIT_CNF` um.
  - **DigitalInput_I2**: `logiBUS::io::DI::logiBUS_IXA`, Parameter `Input = Input_I2`.
  - **DigitalOutput_Q2**: `logiBUS::io::DQ::logiBUS_QXA`, Parameter `Output = Output_Q2`.
  - **AOption_EVENT_TO_X**: `Option::conversion::AOption_EVENT_TO_X`, setzt `INIT_REQ/QI_OUT` in den Adapter am Plug `AOption_OUT` um und meldet `INIT_CNF` zurück.

Programmablauf und Verbindungen

- `AOption_IN -> AOption_X_TO_EVENT.AOption_IN`
- Init-Kette: `AOption_X_TO_EVENT.INIT_REQ -> DigitalInput_I2.INIT; DigitalInput_I2.INITO -> DigitalOutput_Q2.INIT; DigitalOutput_Q2.INITO -> AOption_EVENT_TO_X.INIT_REQ; AOption_EVENT_TO_X.INIT_CNF -> AOption_X_TO_EVENT.INIT_CNF`
- `AOption_X_TO_EVENT.QI_OUT -> DigitalInput_I2.QI, DigitalOutput_Q2.QI` und `AOption_EVENT_TO_X.QI_OUT`
- `AOption_EVENT_TO_X.AOption_OUT -> AOption_OUT`
- Adapterverbindung `DigitalInput_I2.IN -> DigitalOutput_Q2.OUT`: der Eingang wird auf den Ausgang durchgeschaltet.

Technische Besonderheiten

- `AOption_OUT` wird erst ausgelöst, wenn `DigitalOutput_Q2` selbst initialisiert ist (`DigitalOutput_Q2.INITO -> AOption_EVENT_TO_X.INIT_REQ`).
- Dessen `INIT_CNF` wird an den eigenen Socket `AOption_IN` zurückgemeldet. Der aufrufende DEMUX sieht die Gruppe daher erst als fertig, wenn die gesamte Kette (inklusive nachgeschalteter Gruppen an `AOption_OUT`) bestätigt hat.
- Das `QI` der Gruppe wird an Eingang, Ausgang und den Plug `AOption_OUT` weitergegeben.

Anwendungsszenarien

- Erstes Glied einer Daisy-Chain: `AOption_OUT` wird mit dem Socket einer nachgeschalteten Gruppe wie `OptionalGroup_001b` verbunden, sodass beide Gruppen nacheinander (erst a, dann b) aktiviert werden.

Zusammenfassung

`OptionalGroup_002a` bündelt `Input_I2` und `Output_Q2` zu einer per `AOption` schaltbaren Optionsgruppe und reicht die Aktivierung über `AOption_OUT` an die nächste Gruppe weiter.

🛠️ Zugehörige Übungen

- Uebung_094b3f_AX

2.1.6 MyLib (test_B)

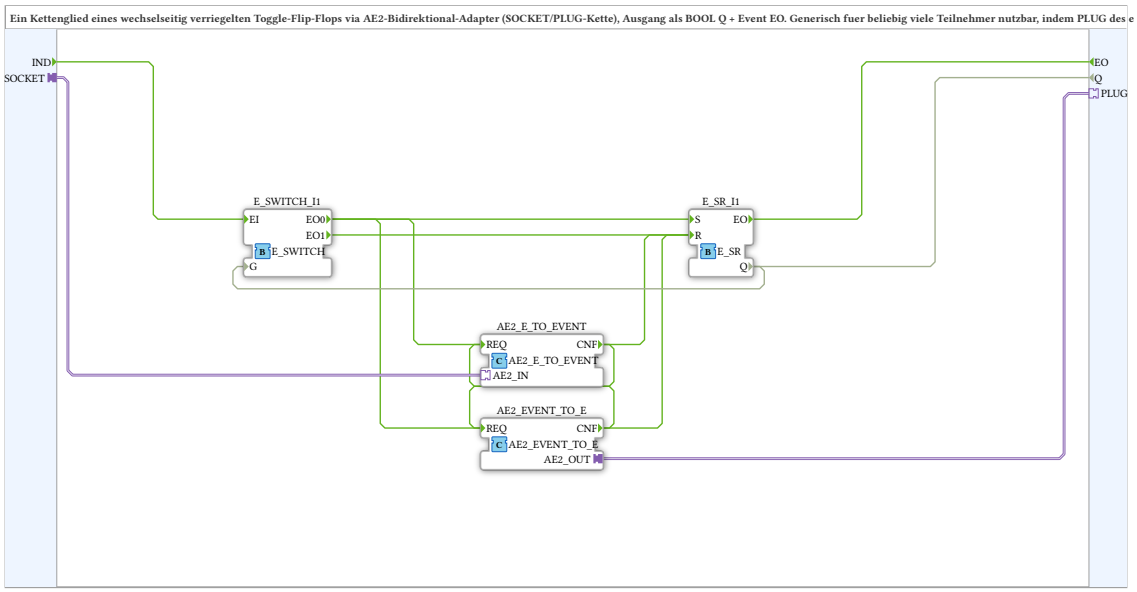
2.1.6.1 sys (test_B)

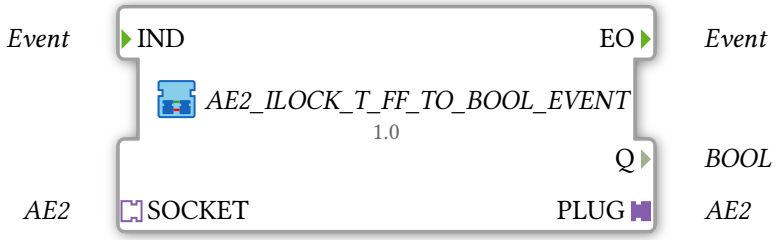
Eigenständige `MyLib::sys`-Bausteinbibliothek des Trainingssystems `test_B` — keine Namensüberschneidung mit MyLib (test_AX). Siehe insbesondere Background-Farbbausteine (gemeinsames Muster) für die große Familie der Hintergrundfarb-Bausteine, die es hier ohne `_AX`-Suffix gibt (Selector als `BOOL` statt Adapter).

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AE2_ILOCK_T_FF_TO_BOOL_EVENT





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **AE2_ILOCK_T_FF_TO_BOOL_EVENT** ist ein Kettenglied eines wechselseitig verriegelten Toggle-Flip-Flops. Er nutzt einen bidirektionalen **AE2-Adapter** als SOCKET/PLUG-Kette, wodurch beliebig viele Teilnehmer hintereinander geschaltet werden können. Der Ausgang wird als **BOOL Q** und als **Event EO** bereitgestellt.

Der Baustein ist generisch einsetzbar: Der PLUG eines Kettenglieds wird mit dem SOCKET des nächsten verbunden. Auf diese Weise entsteht eine wechselseitige Verriegelung, bei der genau ein Teilnehmer aktiv ist und den Zustand kontrolliert weitergibt.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung
IND	Event	Startet die Verriegelungs- und Weitergabelogik des Kettenglieds.

Ereignis-Ausgänge

Name	Typ	Beschreibung
EO	Event	Wird ausgelöst, wenn das interne SR-Flipflop seinen Zustand ändert.

Daten-Eingänge

Keine.

Daten-Ausgänge

Name	Typ	Beschreibung
Q	BOOL	Aktueller Zustand des internen Flipflops.

Adapter

Name	Richtung	Typ	Beschreibung
SOCKET	Eingang	`adapter::types::bidirectional::AE2`	Empfängt Ereignisse vom vorherigen Kettenglied.
PLUG	Ausgang	`adapter::types::bidirectional::AE2`	Sendet Ereignisse an das nächste Kettenglied.

Funktionsweise

Der Baustein besteht intern aus einem SR-Flipflop (`E_SR_I1`), einem Ereignis-Schalter (`E_SWITCH_I1`) sowie zwei AE2-Konvertern (`AE2_EVENT_TO_E` und `AE2_E_TO_EVENT`).

Ein eingehendes Ereignis an `IND` wird zunächst dem Ereignis-Schalter zugeführt. Dieser prüft den aktuellen Zustand von `Q`:

- Ist `Q = 1` (aktiv), wird der Pfad `EO0` aktiviert. Dieser setzt das SR-Flipflop über `S` und stößt gleichzeitig die Kommunikation über die AE2-Adapter an.
- Ist `Q = 0` (inaktiv), wird der Pfad `EO1` aktiviert. Dieser setzt das SR-Flipflop über `R` zurück, ohne die Adapterkette zu verwenden.

Die beiden AE2-Konverter sind so verschaltet, dass Ereignisse vom `SOCKET` zum `PLUG` und umgekehrt weitergereicht werden. Erst nach Abschluss dieser Übertragung wird das SR-Flipflop über die Rückkopplung (`CNF`) zurückgesetzt. Dadurch wird sichergestellt, dass die Weitergabe an das nächste Kettenglied vollständig abgeschlossen ist, bevor der eigene Zustand geändert wird.

Der Ausgang `Q` gibt den aktuellen Zustand des internen SR-Flipflops aus. Das Ereignis `EO` signalisiert eine Zustandsänderung und kann zum Beispiel zur Weiterverarbeitung in der Steuerung verwendet werden.

Technische Besonderheiten

- **AE2-Bidirektional-Kette:** Durch die Kombination von `SOCKET` und `PLUG` ist eine einfache Verkettung mehrerer Instanzen möglich.
- **Handshake über AE2-Konverter:** Die Konverter `AE2_EVENT_TO_E` und `AE2_E_TO_EVENT` sind kreuzgekoppelt, sodass eine gesicherte Weitergabe über die Kette erfolgt.
- **Kombinierter Ausgang:** Der Zustand wird sowohl als BOOL (`Q`) als auch als Event (`EO`) bereitgestellt.
- **Wiederverwendbar:** Der Baustein ist als SubApp gekapselt und kann in verschiedenen Steuerungsprojekten mehrfach instanziiert werden.

Zustandsübersicht

Q (vor IND)	Zustand	Reaktion auf IND
`0`	inaktiv	`EO1` wird aktiviert, das SR-Flipflop wird zurückgesetzt, `Q` bleibt `0`. Keine Adapteraktivität.
`1`	aktiv	`EO0` wird aktiviert, das SR-Flipflop wird gesetzt und die Adapterkette wird angestoßen. Nach erfolgreicher Weitergabe wird `Q` auf `0` zurückgesetzt.

Anwendungsszenarien

- **Rundlaufsteuerung:** Mehrere Teilnehmer werden zu einer Kette verbunden; genau ein Teilnehmer ist aktiv und gibt den Aktivitätszustand an den nächsten weiter.
- **Wechselseitige Verriegelung:** Maschinen oder Anlagen, bei denen nie zwei Teilnehmer gleichzeitig aktiv sein dürfen.
- **Erweiterte Ausgabe:** Wenn neben der Adapter-Kommunikation auch ein BOOL-Signal und ein Ereignis für die übergeordnete Steuerung benötigt werden.

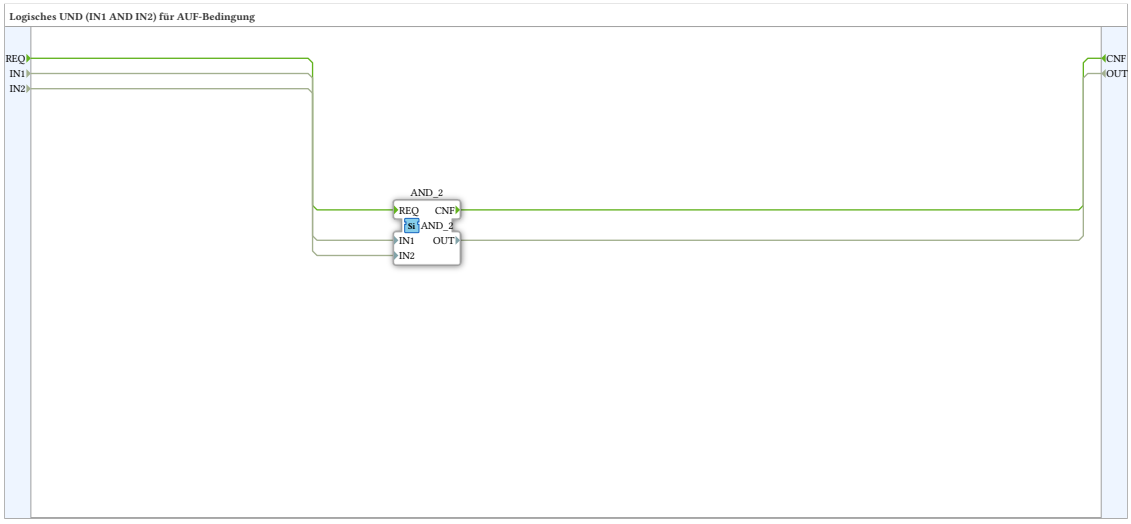
Vergleich mit ähnlichen Bausteinen

Baustein	Unterschied
`AE2_ILOCK_T_FF_TO_BOOL_EVENT`	Ausgang als `BOOL Q` und `Event EO`.
`MyLib::sys::AE2_ILOCK_T_FF_TO_AX`	Gleiche interne Logik, aber Ausgang als AX-Adapter anstelle von BOOL und Event.
Einfaches `E_SR` ohne Adapter	Keine Kopplung mehrerer Bausteine über AE2, keine wechselseitige Verriegelung.

Fazit

`AE2_ILOCK_T_FF_TO_BOOL_EVENT` ist eine vielseitig einsetzbare SubApp für verriegelte Toggle-Ketten. Durch die Kombination aus AE2-Adapterkette, internem SR-Flipflop und BOOL/Event-Ausgang eignet er sich besonders für Steuerungsaufgaben, bei denen mehrere Teilnehmer wechselseitig gesperrt werden müssen und gleichzeitig ein direkter digitaler Zustand sowie ein Ereignis benötigt werden.

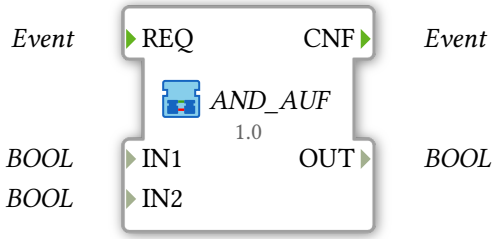
AND_AUF



• – ○ • – ○ • – ○ •

Einleitung

`AND_AUF` bildet die logische Verknüpfung `IN1 AND IN2` und liefert damit die Freigabebedingung für eine „AUF"-Bewegung (z. B. Ventil/Klappe öffnen): beide Bedingungen müssen gleichzeitig erfüllt sein. Für die komplementäre „ZU"-Bedingung siehe `AND_ZU`.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: AND_AUF

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **AND_2**: `iec61131::bitwiseOperators::AND_2` — bitweises UND, hier auf BOOL-Ebene als logisches UND verwendet.
- **Funktionsweise**: `REQ` löst `AND_2` aus; `IN1` und `IN2` werden verundet und als `OUT` sowie `CNF` zurückgegeben.

Programmablauf und Verbindungen

1. `REQ` ⟶ `AND_2.REQ`.
2. `IN1` ⟶ `AND_2.IN1`; `IN2` ⟶ `AND_2.IN2`.
3. `AND_2.OUT` ⟶ `OUT`; `AND_2.CNF` ⟶ `CNF`.

Anwendungsszenarien

- Freigabelogik für Öffnen-Bewegungen, bei denen zwei unabhängige Bedingungen (z. B. Endschalter ZU nicht aktiv UND Freigabe-Taste) gleichzeitig erfüllt sein müssen.

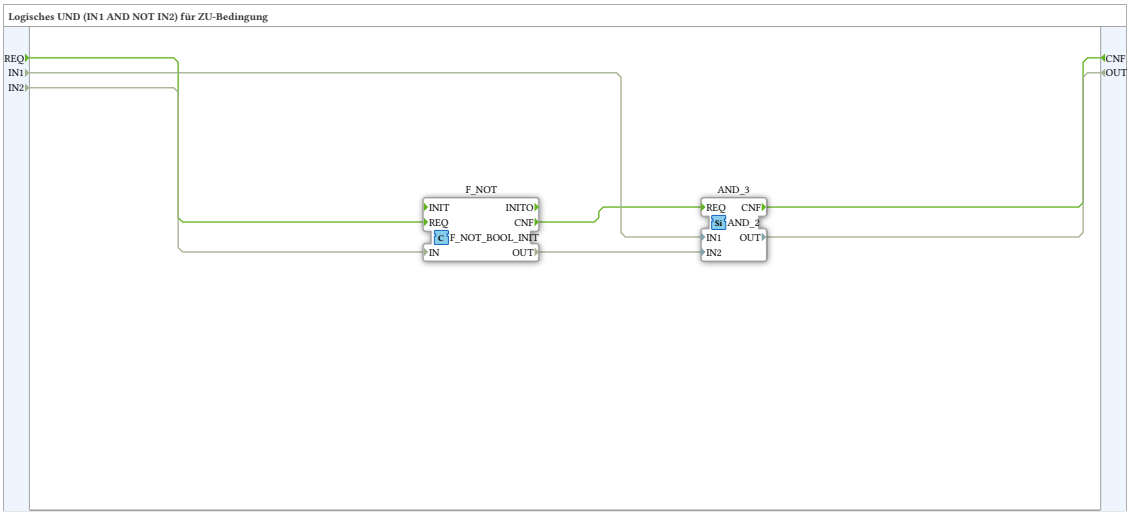
Zusammenfassung

Einfacher, wiederverwendbarer UND-Baustein als sprechend benannter Baustein für die AUF-Bedingung einer Ventil-/Klappensteuerung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

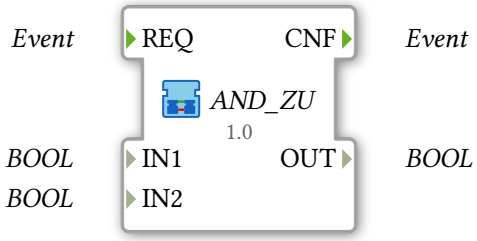
AND_ZU



• – ○ • – ○ • – ○ •

Einleitung

`AND_ZU` bildet die logische Verknüpfung `IN1 AND NOT IN2` und liefert damit die Freigabebedingung für eine „ZU“-Bewegung (z. B. Ventil/Klappe schließen): `IN1` muss erfüllt und `IN2` muss negiert erfüllt sein. Komplementär zu `AND_AUF`.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: AND_ZU

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **F_NOT** (`F_NOT`): `iec61131::booleanOperators::F_NOT_BOOL_INIT` — negiert `IN2`, bevor verundet wird.
  - **AND_3** (`AND_2`-Typ): `iec61131::bitwiseOperators::AND_2` — verundet `IN1` mit dem negierten `IN2`.
- **Funktionsweise**: `REQ` löst zuerst `F_NOT` aus, dessen `CNF` wiederum `AND_3` auslöst — die Negation erfolgt somit vor der Verundung, sowohl datentechnisch als auch ereignistechnisch sequenziell.

Programmablauf und Verbindungen

1. `REQ` ⟶ `F_NOT.REQ`; `IN2` ⟶ `F_NOT.IN`.
2. `F_NOT.CNF` ⟶ `AND_3.REQ` (Ereigniskette).
3. `IN1` ⟶ `AND_3.IN1`; `F_NOT.OUT` ⟶ `AND_3.IN2`.
4. `AND_3.OUT` ⟶ `OUT`; `AND_3.CNF` ⟶ `CNF`.

Anwendungsszenarien

- Freigabelogik für Schließen-Bewegungen, bei denen eine Bedingung erfüllt und eine zweite explizit *nicht* erfüllt sein darf (z. B. Freigabe-Taste UND Endschalter AUF nicht aktiv).

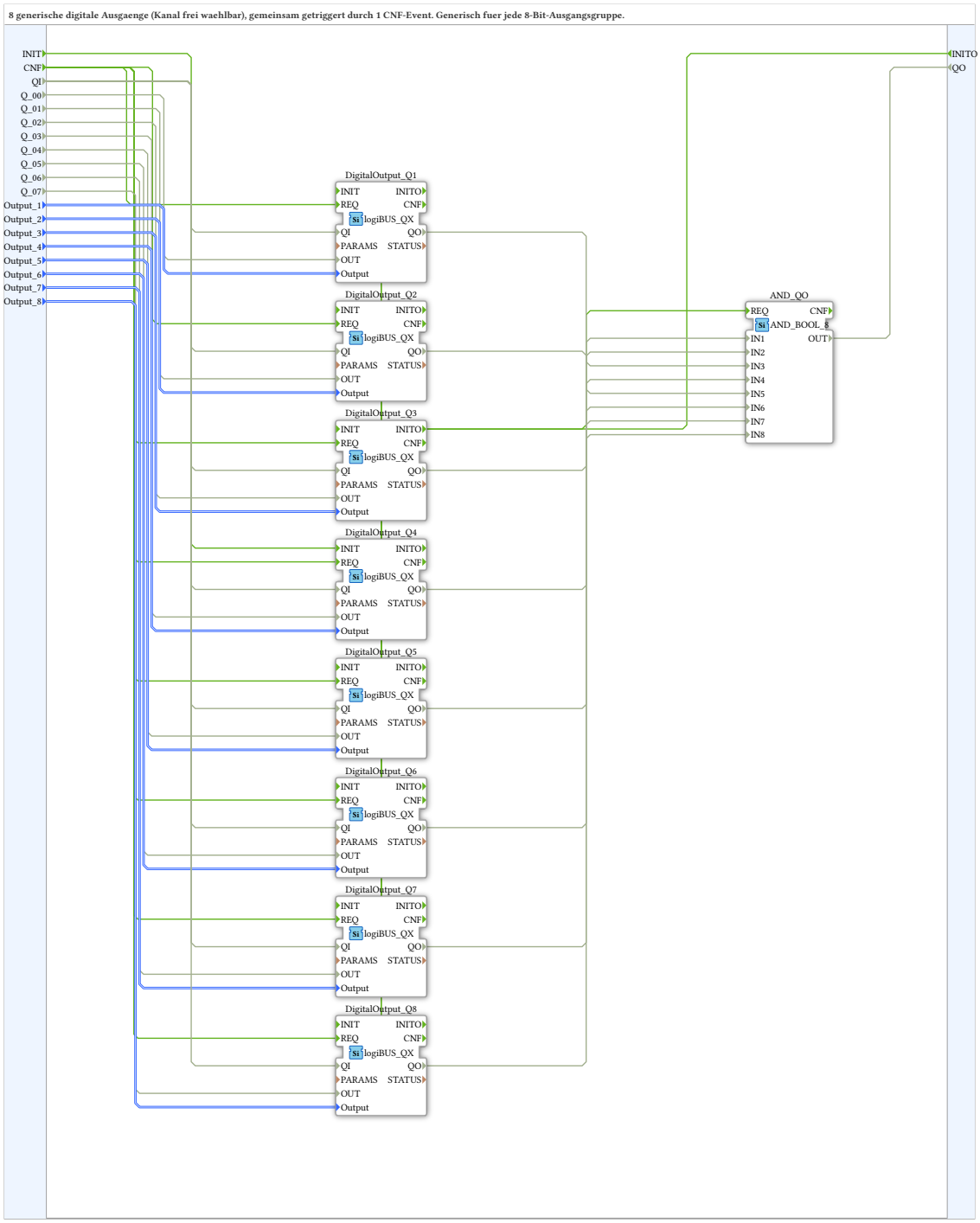
Zusammenfassung

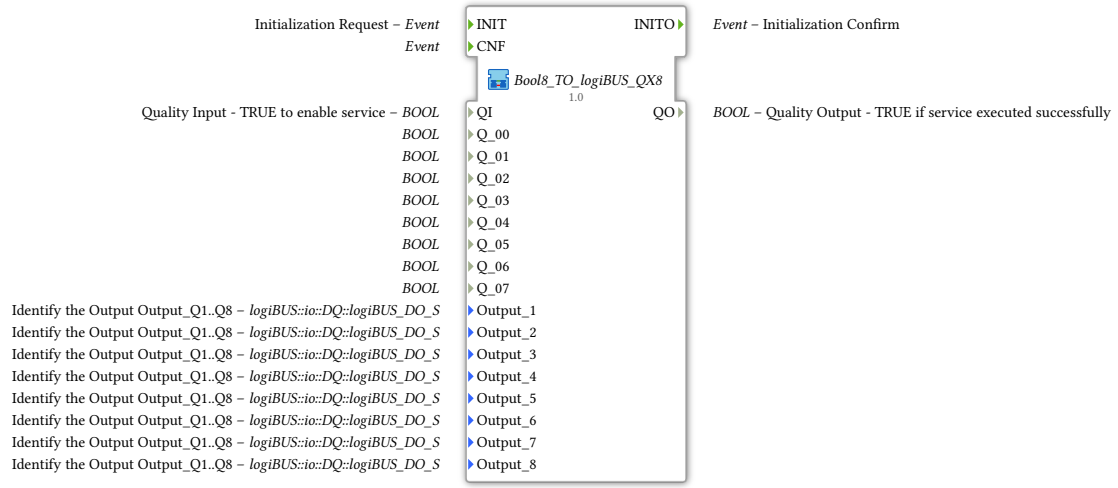
Erweiterung von `AND_AUF` um eine Negation von `IN2` — die passende Gegenstück-Logik für die ZU-Bedingung einer Ventil-/Klappensteuerung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Bool8_TO_logiBUS_QX8





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **Bool8_TO_logiBUS_QX8** ist eine wiederverwendbare Subapplikation, die acht digitale Ausgangssignale (BOOL) über den logiBUS in eine frei parametrierbare 8-Bit-Ausgangsgruppe überträgt. Die Ausgangskanäle werden nicht fest verdrahtet, sondern über separate Konfigurationseingänge (`Output_1` … `Output_8`) adressiert. Dadurch kann derselbe Baustein für unterschiedliche Hardware-Ausgänge eingesetzt werden, ohne die interne Logik zu verändern. Ein gemeinsames Ereignis (`CNF`) triggert alle acht Ausgaben gleichzeitig.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
`CNF`	Event	Gemeinsamer Trigger für alle Ausgänge
`INIT`	EInit	Initialisierungsanforderung; startet die Initialisierungskette der acht `logiBUS_QX`-Instanzen

Ereignis-Ausgänge

Name	Typ	Kommentar
`INITO`	EInit	Initialisierung abgeschlossen; kommt von `DigitalOutput_Q3.INITO`, dem Ende der Kette

Daten-Eingänge

Name	Typ	Kommentar
`QI`	BOOL	Quality Input - TRUE aktiviert den Dienst (Default `TRUE`); wird an alle acht `logiBUS_QX`-Instanzen verteilt
`Q_00`	BOOL	Binärwert für Ausgangskanal 1
`Q_01`	BOOL	Binärwert für Ausgangskanal 2
`Q_02`	BOOL	Binärwert für Ausgangskanal 3
`Q_03`	BOOL	Binärwert für Ausgangskanal 4
`Q_04`	BOOL	Binärwert für Ausgangskanal 5

Q_05	BOOL	Binärwert für Ausgangskanal 6
Q_06	BOOL	Binärwert für Ausgangskanal 7
Q_07	BOOL	Binärwert für Ausgangskanal 8
Output_1	logiBUS::io::DQ::logiBUS_DO_S	Adresse für Ausgangskanal 1 (initial Invalid)
Output_2	logiBUS::io::DQ::logiBUS_DO_S	Adresse für Ausgangskanal 2 (initial Invalid)
Output_3	logiBUS::io::DQ::logiBUS_DO_S	Adresse für Ausgangskanal 3 (initial Invalid)
Output_4	logiBUS::io::DQ::logiBUS_DO_S	Adresse für Ausgangskanal 4 (initial Invalid)
Output_5	logiBUS::io::DQ::logiBUS_DO_S	Adresse für Ausgangskanal 5 (initial Invalid)
Output_6	logiBUS::io::DQ::logiBUS_DO_S	Adresse für Ausgangskanal 6 (initial Invalid)
Output_7	logiBUS::io::DQ::logiBUS_DO_S	Adresse für Ausgangskanal 7 (initial Invalid)
Output_8	logiBUS::io::DQ::logiBUS_DO_S	Adresse für Ausgangskanal 8 (initial Invalid)

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde (AND der QO aller acht logiBUS_QX-Instanzen)

Adapter

Keine.

Funktionsweise

Die Subapplikation enthält acht interne Instanzen des Funktionsblocks logiBUS::io::DQ::logiBUS_QX. Jede Instanz ist fest mit einem der acht Eingangssignale (Q_00 … Q_07) verbunden und erhält über die zugehörige Adressvariable (Output_1 … Output_8) die Zieladresse des logiBUS-Ausgangs.

Der gemeinsame Ereigniseingang CNF ist mit dem REQ-Eingang aller acht QX-Bausteine verbunden. Solange QI (Dateneingang der SubApp, Default TRUE, an alle internen Bausteine verteilt) auf TRUE steht, wird bei jedem ankommenden CNF-Event der aktuelle Zustand der Bool-Eingänge parallel an die jeweiligen logiBUS-Ausgänge übertragen.

Die Konfiguration der Ausgangsadressen erfolgt über die Eingänge Output_1 … Output_8. Diese sind vom Typ logiBUS_DO_S und müssen vor der ersten Nutzung mit gültigen logiBUS-Adressen belegt werden (initial stehen sie auf Invalid).

Initialisierung und Qualität: INIT läuft als Kette DigitalOutput_Q4 → DigitalOutput_Q7 → DigitalOutput_Q1 → DigitalOutput_Q5 → DigitalOutput_Q2 → DigitalOutput_Q6 → DigitalOutput_Q8 → DigitalOutput_Q3 (jeweils INITO → nächstes INIT); DigitalOutput_Q3.INITO geht an INITO und an AND_QO.REQ. QO ist der Ausgang der Instanz AND_QO (iec61131::booleanOperators::AND_BOOL_8), die die QO der acht Instanzen verknüpft.

Technische Besonderheiten

- **Generische 8-Bit-Ausgangsgruppe:** Durch die Parametrierung der Ausgangsadressen ist der Baustein nicht an eine feste Hardware gebunden und kann für beliebige logiBUS-Ausgangskanäle wiederverwendet werden.
- **Gemeinsame Triggerung:** Ein einziges CNF-Event synchronisiert die Ausgabe aller acht Kanäle – keine zeitlichen Versätze.
- **Interne Verdrahtung:** QI ist nicht mehr intern fest auf TRUE gesetzt, sondern wird von außen an alle acht QX-Bausteine durchgereicht; die Steuerung der Ausgabe erfolgt über die Bool-Eingänge und das Event.
- **Rückmeldung nur über Qualität und Initialisierung:** Es gibt keine Daten-Ausgänge außer QO und keinen Ereignis-Ausgang außer INITO; die Subapplikation ist ansonsten rein „send-oriented“.
- **Wiederverwendung:** Durch die losgelöste Adressverwaltung eignet sich der Baustein für modulare Automatisierungskonzepte.

Zustandsübersicht

Da es sich um eine Subapplikation handelt, besitzt der Baustein keinen eigenen Zustandsautomaten. Er ist **zustandslos** in dem Sinne, dass er nur auf Ereignisse reagiert und sein Verhalten nicht von internen Zuständen abhängt. Der interne QX-Baustein kann selbst einen Zustandsautomaten haben (z. B. für die Kommunikation mit dem logiBUS), dieser ist jedoch in den QX-Komponenten gekapselt.

Anwendungsszenarien

- **Steuerung von 8 Aktoren** über einen logiBUS-Feldbus, z. B. Ventile, Lampen oder Relais.
- **Modulare I/O-Erweiterung** in Automatisierungsanlagen, bei denen die physikalischen Ausgänge nicht fest verdrahtet sind.
- **Synchronisierte Ausgabe** mehrerer Binärsignale, z. B. für Anzeige- oder Diagnosezwecke.
- **Einsatz in verteilten Systemen**, wo mehrere 8-Bit-Gruppen über denselben Baustein parametriert werden können.

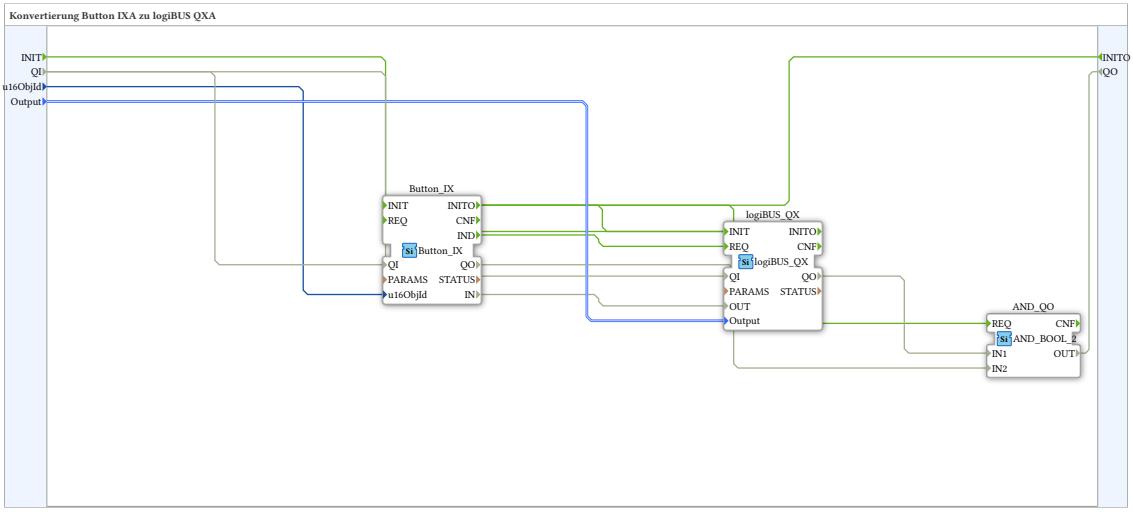
Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu einfachen Einzel-Ausgangs-Bausteinen (z. B. einem einzelnen `logiBUS_QX`) bündelt `Bool8_TO_logiBUS_QX8` acht Kanäle in einer logischen Einheit. Dadurch wird die Anzahl der Verbindungen im aufrufenden FB reduziert und die Synchronisation der Ausgaben vereinfacht. Gegenüber einer Lösung mit acht separaten QX-Bausteinen bietet die Subapplikation den Vorteil einer definierten Schnittstelle und einer einheitlichen Konfiguration.

Fazit

`Bool8_TO_logiBUS_QX8` stellt eine flexible und kompakte Lösung für die Ansteuerung von acht digitalen Ausgängen über logiBUS dar. Die Trennung von Logik und Adressierung ermöglicht eine breite Wiederverwendung und trägt zur Modularisierung und Wartbarkeit von Automatisierungsprojekten bei. Durch die gemeinsame Triggerung wird eine konsistente Ausgabe aller Kanäle sichergestellt.

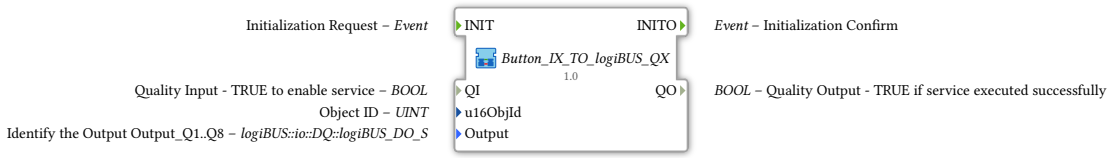
Button_IX_TO_logiBUS_QX



• — ○ • — ○ • — ○ •

Einleitung

`Button_IX_TO_logiBUS_QX` ist das test_B-Gegenstück zu `Button_IXA_TO_logiBUS_QXA`: Ein VT-Taster (`Button_IX`) schaltet direkt einen physischen digitalen Ausgang (`logiBUS_QX`) — hier ohne Adapter, mit klassischen Ereignis-/Datenverbindungen.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Button_IX_TO_logiBUS_QX

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **Button_IX**: `isobus::UT::io::Button::Button_IX` — VT-Taster (Nicht-Adapter-Variante).
  - **logiBUS_QX**: `logiBUS::io::DQ::logiBUS_QX` — physischer digitaler Ausgang (Nicht-Adapter-Variante).
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_2` — verknüpft `logiBUS_QX.QO` und `Button_IX.QO` zum `QO` der SubApp.
- **Funktionsweise**: `Button_IX.IND` löst `logiBUS_QX.REQ` aus; der Datenwert `Button_IX.IN` wird direkt auf `logiBUS_QX.OUT` verdrahtet.

Programmablauf und Verbindungen

1. `u16ObjId` ⟶ `Button_IX.u16ObjId`; `Output` ⟶ `logiBUS_QX.Output`.
2. `Button_IX.IND` ⟶ `logiBUS_QX.REQ`; `Button_IX.IN` ⟶ `logiBUS_QX.OUT`.
3. **Initialisierung**: Ereignis-Eingang `INIT` (`EInit`) ⟶ `logiBUS_QX.INIT`; `logiBUS_QX.INITO` ⟶ `Button_IX.INIT`; `Button_IX.INITO` ⟶ Ereignis-Ausgang `INITO` (`EInit`) und ⟶ `AND_QO.REQ`.
4. **Qualität**: Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`) ⟶ `logiBUS_QX.QI` und `Button_IX.QI`; `logiBUS_QX.QO` ⟶ `AND_QO.IN1`; `Button_IX.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ Datenausgang `QO` (`BOOL`).

Anwendungsszenarien

- Minimaler VT-Taster-zu-Ausgang-Baustein für test_B, ohne Statusanzeige.

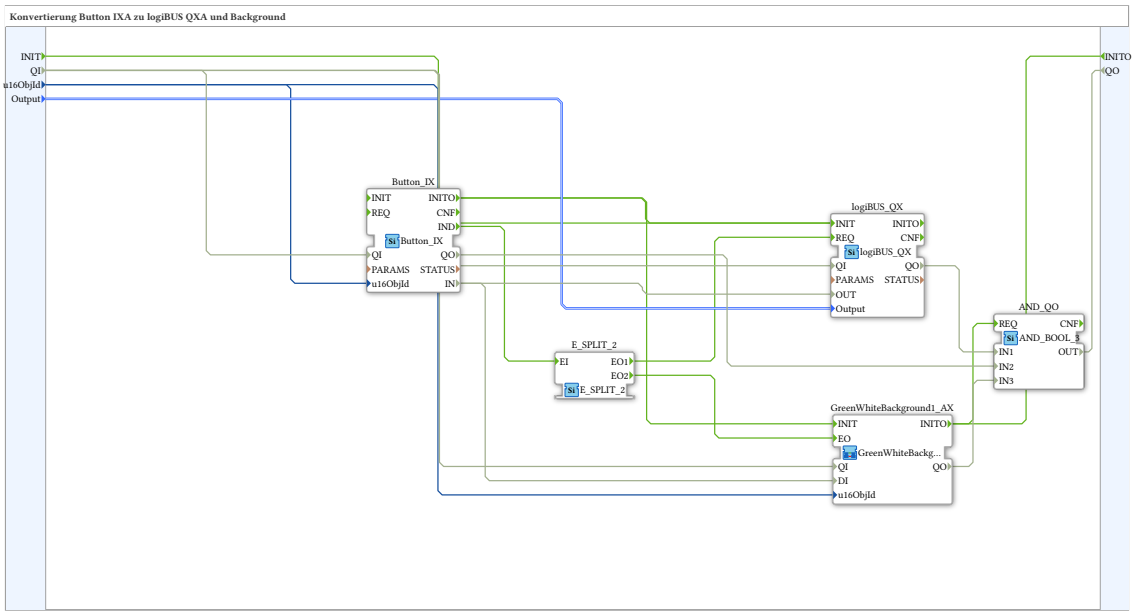
Zusammenfassung

test_B-Pendant zu `Button_IXA_TO_logiBUS_QXA`: dieselbe Funktion, klassische Ereignis-/Datenverbindungen statt Adapter.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

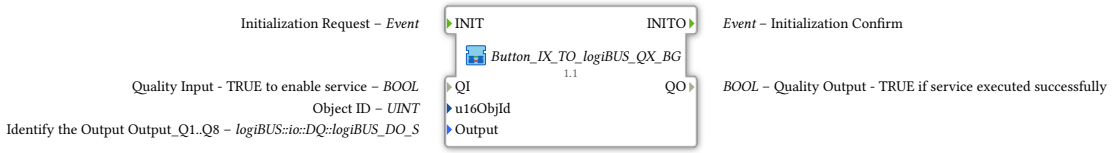
Button_IX_TO_logiBUS_QX_BG



• – ○ • – ○ • – ○ •

Einleitung

`Button_IX_TO_logiBUS_QX_BG` erweitert `Button_IX_TO_logiBUS_QX` um eine VT-Statusfarbe — das test_B-Gegenstück zu `Button_IXA_TO_logiBUS_QXA_BG`.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Button_IX_TO_logiBUS_QX_BG

- **Typ**: SubAppType
- **Schnittstelle (zusätzlich)**: Ereignis-Eingang `INIT`, Ereignis-Ausgang `INITO` (je `EInit`), Daten-Eingang `QI` (`BOOL`, Vorgabe `TRUE`), Daten-Ausgang `QO` (`BOOL`).
- **Verwendete interne FBs**:
  - **Button_IX**: `isobus::UT::io::Button::Button_IX` — VT-Taster.
  - **logiBUS_QX**: `logiBUS::io::DQ::logiBUS_QX` — physischer digitaler Ausgang.
  - **E_SPLIT_2**: `iec61499::events::E_SPLIT_2` — verzweigt das Tasterereignis.
  - **GreenWhiteBackground1_AX** (SubApp, tatsächlich vom Typ `MyLib::sys::GreenWhiteBackground1C` — kompakter Wrapper der test_B-Basisvariante, siehe Background-Farbbausteine): setzt die Statusfarbe.
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_3` — verknüpft die `QO`-Ausgänge von `logiBUS_QX`, `Button_IX` und `GreenWhiteBackground1_AX` zu `QO`.
- **Funktionsweise**: `Button_IX.IND` wird über `E_SPLIT_2` an `logiBUS_QX.REQ` und an den Farbwechsel-Baustein verteilt; der Datenwert `Button_IX.IN` geht sowohl an `logiBUS_QX.OUT` als auch an `GreenWhiteBackground1_AX.DI`.

Programmablauf und Verbindungen

1. `u16ObjId` ⟶ `Button_IX.u16ObjId` und (Farbbaustein-Instanz) `.u16ObjId`; `Output` ⟶ `logiBUS_QX.Output`.
2. `Button_IX.IND` ⟶ `E_SPLIT_2.EI` ⟶ `E_SPLIT_2.EO1` ⟶ `logiBUS_QX.REQ`, `E_SPLIT_2.EO2` ⟶ Farbbaustein `.EO`.
3. `Button_IX.IN` ⟶ `logiBUS_QX.OUT` und ⟶ Farbbaustein `.DI`.
4. Initialisierung: `INIT` ⟶ `logiBUS_QX.INIT`; `logiBUS_QX.INITO` ⟶ `Button_IX.INIT`; `Button_IX.INITO` ⟶ `GreenWhiteBackground1_AX.INIT`; `GreenWhiteBackground1_AX.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`.
5. Qualität: `QI` ⟶ `.QI` von `logiBUS_QX`, `Button_IX` und `GreenWhiteBackground1_AX`; deren `QO` ⟶ `AND_QO.IN1` bis `AND_QO.IN3` (in dieser Reihenfolge); `AND_QO.OUT` ⟶ `QO`.

Anwendungsszenarien

- VT-Taster mit optischer Statusrückmeldung für test_B, ohne OPC-UA.

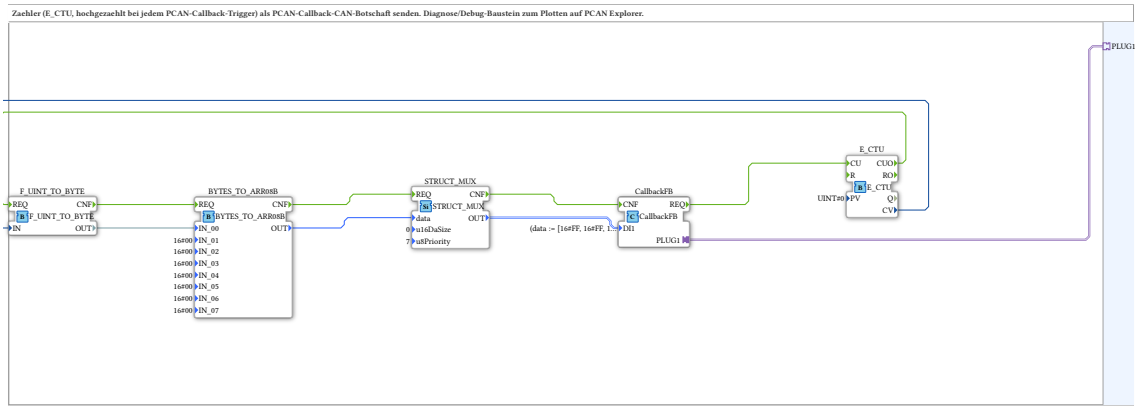
Zusammenfassung

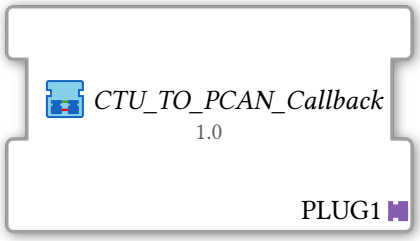
test_B-Pendant zu `Button_IXA_TO_logiBUS_QXA_BG`, mit `E_SPLIT_2` statt `AX_SPLIT_2` und der Kompakt-Wrapper-Variante des Background-Bausteins.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

CTU_TO_PCAN_Callback





Callback

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `CTU_TO_PCAN_Callback` ist eine Subapplikation (SubApp), die als Diagnose- und Debug-Baustein dient. Er zählt jedes Ereignis, das von einem PCAN-Callback ausgelöst wird, und sendet den aktuellen Zählerwert als CAN-Botschaft über den PCAN-Explorer aus. Dadurch können Ereignisse oder Triggerzustände auf dem CAN-Bus direkt visualisiert und analysiert werden.

Die SubApp kapselt die notwendige Verarbeitungskette: einen Aufwärtszähler (E_CTU), eine Konvertierung des Zählerstands in eine CAN-Nachricht und die Übertragung über den mitgelieferten Adapter. Sie wurde zur Wiederverwendung aus einer größeren Übung ausgelagert.

Schnittstellenstruktur

Ereignis-Eingänge

Keine externen Ereignis-Eingänge vorhanden. Die SubApp wird ausschließlich über den Adapter angesteuert, der Ereignisse vom PCAN-Callback empfängt.

Ereignis-Ausgänge

Keine externen Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine externen Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine externen Daten-Ausgänge vorhanden.

Adapter

- **PLUG1** (Typ: `isobus::pgn::tx::Callback`)
  Dieser Adapter verbindet die SubApp mit dem PCAN-Callback-Sender. Über ihn werden die Callback-Ereignisse empfangen (REQ) und die CAN-Botschaften gesendet (über den internen CallbackFB).

Funktionsweise

Die SubApp arbeitet ereignisgesteuert:

1. Über den Adapter `PLUG1` wird ein Callback-Ereignis empfangen (dies entspricht einem Trigger vom PCAN-System).

2. Das Ereignis wird an den internen Funktionsblock `CallbackFB` (vom Typ `isobus::pgn::tx::CallbackFB`) über die Verbindung `CallbackFB.REQ` weitergeleitet.
3. Gleichzeitig wird das Ereignis an den Zähler `E_CTU` (Ereignis-Eingang `CU`) gesendet, sodass dieser bei jedem Trigger um 1 erhöht wird.
4. Nach der Zählung gibt `E_CTU` den aktuellen Zählerstand (`CV`) als `UINT`-Wert aus. Dieser wird über `F_UINT_TO_BYTE` in ein einzelnes Byte (0-255) umgewandelt.
5. Das Byte wird in `BYTES_TO_ARR08B` in ein Byte-Array (8 Bytes) eingefügt (an Position 0, die restlichen Bytes sind mit 0 initialisiert).
6. Das Byte-Array wird über `STRUCT_MUX` in eine CAN-Nachrichtenstruktur (`isobus::pgn::CAN_MSG`) verpackt. Dabei werden zusätzlich die Parameter `u8Priority` (7) und `u16DaSize` (0) gesetzt.
7. Die fertige CAN-Nachricht wird über `STRUCT_MUX` an `CallbackFB.DI1` übergeben und durch das Ereignis `CallbackFB.CNF` (das nach erfolgreicher Verarbeitung ausgelöst wird) zurückbestätigt.
8. `CallbackFB` sendet die Nachricht über den Adapter `PLUG1` an das PCAN-System, wo sie im PCAN-Explorer dargestellt werden kann.

Durch diese Kette wird bei jedem PCAN-Callback der Zählerstand als CAN-Botschaft mit der CAN-ID und Priorität 7 gesendet.

Technische Besonderheiten

- **Zählerbegrenzung**: Der Zähler `E_CTU` ist auf 0 als Startwert (`PV = 0`) gesetzt und zählt bei jedem Ereignis hoch. Da der Ausgang `CV` über `F_UINT_TO_BYTE` in ein Byte umgewandelt wird, ist der effektive Wertebereich auf 0–255 begrenzt. Nach 255 läuft der Zähler über.
- **Konvertierungskette**: Mehrere Konvertierungsbausteine (`F_UINT_TO_BYTE`, `BYTES_TO_ARR08B`, `STRUCT_MUX`) werden verwendet, um den Zählerstand in das Format einer CAN-Nachricht zu bringen. Diese Kette ist erforderlich, weil die PCAN-Callback-Schnittstelle eine spezifische Datenstruktur (`CAN_MSG`) erwartet.
- **Adapter-Kapselung**: Die SubApp besitzt ausschließlich einen Adapter als Schnittstelle. Dadurch ist sie vollständig in bestehende PCAN-Callback-Netzwerke integrierbar, ohne dass zusätzliche Ein-/Ausgänge erforderlich sind.
- **Diagnosezweck**: Der Baustein ist für Debugging- und Plotting-Anwendungen konzipiert, z. B. um Ereignisse oder Zustandsänderungen auf dem CAN-Bus sichtbar zu machen.

Zustandsübersicht

Der Baustein besitzt keine eigenen expliziten Zustände. Er arbeitet rein ereignisgesteuert:

- Ruhezustand: Kein Callback-Ereignis, keine Aktivität.
- Aktiver Zustand: Ein Callback-Ereignis tritt ein, der Zähler wird inkrementiert, die Nachricht wird zusammengesetzt und gesendet.
- Nach erfolgreichem Senden kehrt der Baustein in den Ruhezustand zurück.

Die interne Zustandslogik wird vollständig von den verwendeten Funktionsblöcken (`E_CTU`, `CallbackFB`) übernommen.

Anwendungsszenarien

- **Debugging von CAN-Kommunikation**: Überwachung, wie oft bestimmte CAN-Botschaften oder Ereignisse auftreten (z. B. Zähler für Fehler- oder Statusmeldungen).
- **PCAN-Explorer-Plotting**: Darstellung des Zählerstands als kontinuierliche CAN-Botschaft, um zeitliche Verläufe zu visualisieren.
- **Ereignisüberwachung**: Zählen von Callbacks, die von anderen Systemkomponenten ausgelöst werden, und Übertragung dieser Zählerwerte auf den CAN-Bus.
- **Integration in Testumgebungen**: Als diagnostischer Zusatzbaustein in bestehenden PCAN-Callback-Anwendungen, ohne Änderungen an der Hauptlogik.

Vergleich mit ähnlichen Bausteinen

Es gibt ähnliche Diagnose-Bausteine, die Zählerstände senden, jedoch unterscheiden sich diese oft in der Art der Ereignisauslösung oder der Datenaufbereitung:

- **E_CTU direkt + manueller Sender**: Statt einer SubApp könnte man E_CTU und einen Sender getrennt verdrahten. Der Vorteil der SubApp ist die Wiederverwendbarkeit und klare Kapselung der Konvertierungskette.

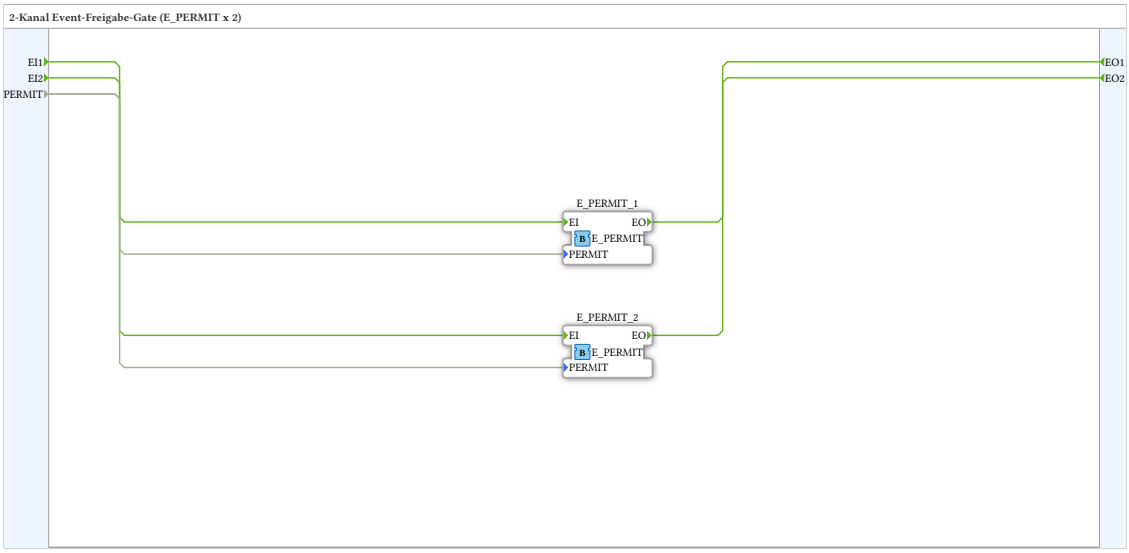
- **Andere Callback-Sender**: Bausteine, die nur CAN-Nachrichten senden, ohne Zähler. Dieser Baustein integriert Zählung und Senden in einem Modul.
- **Bausteine mit fester CAN-ID**: Im Gegensatz zu Bausteinen, bei denen die CAN-ID konfiguriert werden muss, verwendet dieser Baustein die im CallbackFB definierte Konfiguration (hier als Parameter `DI1` vorgegeben – im Standardfall alle Bytes 0xFF).

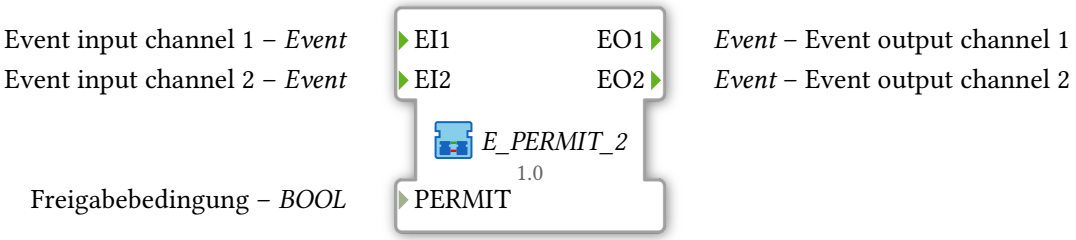
Die SubApp zeichnet sich durch ihre einfache Integration, die automatische Zählfunktion und die vollständige Datenkonvertierung aus.

Fazit

`CTU_TO_PCAN_Callback` ist ein kompakter und wiederverwendbarer Diagnosebaustein für PCAN-Callback-Anwendungen. Er kombiniert einen Zähler mit der Übertragung des Zählerstands als CAN-Botschaft auf elegante Weise. Die SubApp ist ideal für Entwickler, die Ereigniszähler auf dem CAN-Bus beobachten oder plotten möchten, ohne sich um die Details der Datenkonvertierung kümmern zu müssen. Dank der Adapter-Schnittstelle lässt sie sich nahtlos in bestehende PCAN-Callback-Netzwerke integrieren und trägt zur besseren Debugging- und Überwachungsfähigkeit bei.

E_PERMIT_2





• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **E_PERMIT_2** ist eine Subapplikation (SubApp), die zwei unabhängige Event-Freigabe-Kanäle realisiert. Er basiert auf zwei Instanzen des Standard-Funktionsblocks `E_PERMIT` aus der IEC 61499-1-Bibliothek. Die SubApp bietet eine kompakte und wiederverwendbare Lösung zur Steuerung von Ereignisdurchgriffen über ein gemeinsames Freigabesignal.

Die Bezeichnung „2-Kanal Event-Freigabe-Gate“ verdeutlicht die Funktion: Ereignisse, die an den Eingängen `EI1` und `EI2` anliegen, werden nur dann an die Ausgänge `EO1` bzw. `EO2` weitergegeben, wenn das Eingangssignal `PERMIT` den Wert `TRUE` besitzt. Andernfalls werden die Ereignisse blockiert.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Datentyp	Kommentar
EI1	EVENT	Ereigniseingang für Kanal 1
EI2	EVENT	Ereigniseingang für Kanal 2

Ereignis-Ausgänge

Name	Datentyp	Kommentar
EO1	EVENT	Ereignisausgang für Kanal 1
EO2	EVENT	Ereignisausgang für Kanal 2

Daten-Eingänge

Name	Datentyp	Kommentar
PERMIT	BOOL	Freigabebedingung (gemeinsam für beide Kanäle)

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Die SubApp enthält intern zwei `E_PERMIT`-Funktionsblöcke (benannt `E_PERMIT_1` und `E_PERMIT_2`). Das Eingangssignal `PERMIT` wird parallel an beide internen Bausteine weitergeleitet. Ereignisse, die an `EI1` bzw. `EI2` ankommen, werden an den jeweiligen `EI`-Eingang der internen Instanz geführt. Die `EO`-Ausgänge der internen Bausteine sind mit den SubApp-Ausgängen `EO1` und `EO2` verbunden.

Die Funktion eines einzelnen `E_PERMIT`-Bausteins ist wie folgt definiert:

- Ist `PERMIT = TRUE`, werden eingehende Ereignisse (an `EI`) unverändert an `EO` weitergegeben.
- Ist `PERMIT = FALSE`, werden eingehende Ereignisse ignoriert und es wird kein Ereignis am Ausgang erzeugt.

Da beide Kanäle dasselbe Freigabesignal verwenden, verhalten sie sich synchron: Entweder beide Kanäle sind durchlässig oder beide sind gesperrt. Dies ist besonders für Anwendungen sinnvoll, bei denen zwei parallele Ereignispfade gleichzeitig freigegeben oder blockiert werden müssen.

Technische Besonderheiten

- **SubApp-Implementierung:** Die Funktionalität ist als SubApp (verschachtelte Applikation) realisiert, wodurch sie in größere Systeme eingebettet werden kann und die interne Struktur gekapselt bleibt.
- **Wiederverwendbarkeit:** Der Baustein kann mehrfach in einem Projekt verwendet werden, ohne dass die interne Logik neu modelliert werden muss.
- **Einheitliches Freigabesignal:** Ein einziger boolescher Eingang steuert beide Kanäle, was die Verdrahtung vereinfacht und eine konsistente Steuerung gewährleistet.
- **Keine Datenausgänge:** Der Baustein gibt keine Daten zurück; er dient ausschließlich der Steuerung von Ereignisflüssen.
- **Erweiterbarkeit:** Durch die modulare Struktur lässt sich die SubApp leicht um weitere Kanäle erweitern, indem zusätzliche `E_PERMIT`-Instanzen und entsprechende Verbindungen ergänzt werden.

Zustandsübersicht

Da es sich um eine SubApp handelt, besitzt der Baustein selbst keine eigenen Zustände. Die Zustandslogik wird vollständig durch die internen `E_PERMIT`-Funktionsblöcke bestimmt. Diese besitzen typischerweise die folgenden Zustände:

- **IDLE:** Wartet auf ein Ereignis am Eingang `EI`.

- **PERMIT_ACTIVE:** Zustand, in dem das Eingangssignal `PERMIT` geprüft wird. (Die genaue Zustandsmaschinen-Definition des `E_PERMIT`-Bausteins liegt außerhalb dieser Dokumentation, wird aber gemäß IEC 61499-Standard umgesetzt.)

Für die SubApp bedeutet dies, dass sie sich in einem „durchlässigen“ oder „blockierenden“ Modus befindet, abhängig vom Wert von `PERMIT`. Der Zustand wechselt nur durch Änderungen des Freigabesignals; Ereignisse werden entweder sofort weitergeleitet oder verworfen.

Anwendungsszenarien

- **Sicherheitsfreigaben in der Automatisierungstechnik:** Ein übergeordnetes Steuerungssystem setzt `PERMIT` auf `TRUE`, um zwei Prozesspfade (z. B. zwei Ventile oder Antriebe) gleichzeitig zu aktivieren.
- **Test- und Diagnoseschnittstellen:** Freigabe von Ereignissen für zwei unabhängige Überwachungskanäle in einem Diagnosesystem.
- **Redundante Systeme:** Wenn zwei getrennte Kommunikationskanäle nur bei aktiver Freigabe Daten (Ereignisse) senden sollen, kann `E_PERMIT_2` als gemeinsames Gate eingesetzt werden.
- **Bedingte Ereignisweiterleitung in modularen Anlagen:** Ein zentrales Freigabesignal kontrolliert den Durchgriff mehrerer Ereignisströme.

Vergleich mit ähnlichen Bausteinen

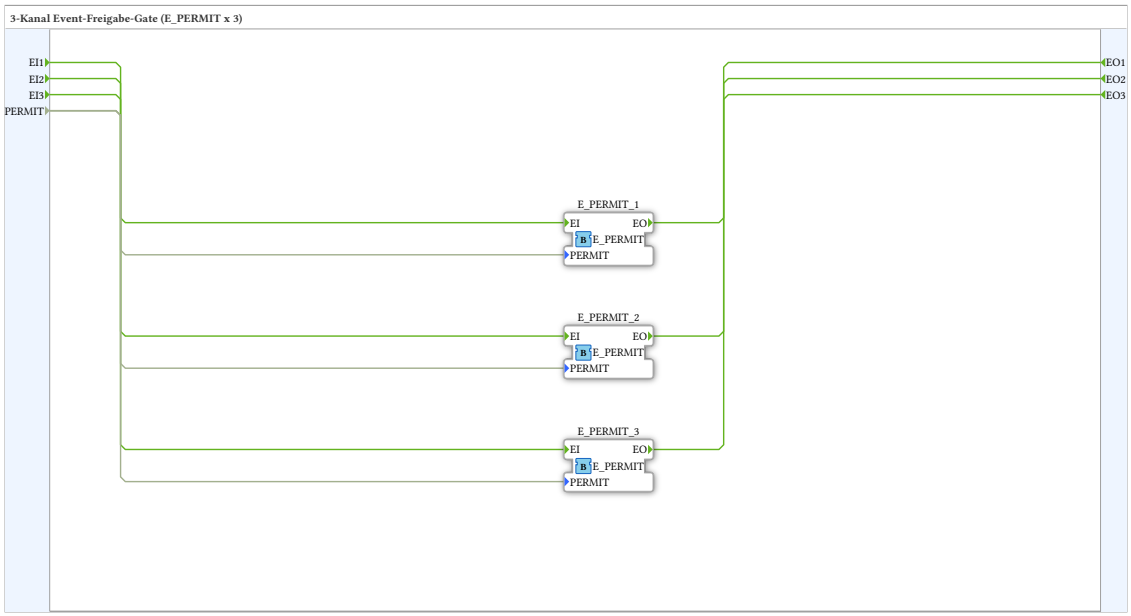
- **E_PERMIT (einfach):** Der einzelne `E_PERMIT`-Baustein steuert nur eine Ereignisverbindung. `E_PERMIT_2` bündelt zwei solche Pfade und reduziert dadurch den Verdrahtungsaufwand auf der übergeordneten Ebene.
- **E_SR (Set/Reset) oder andere Logikbausteine:** Diese arbeiten mit booleschen Signalen, nicht direkt mit Ereignissen. `E_PERMIT_2` ist speziell für die Durchschaltung von Ereignissen ausgelegt und verwendet `PERMIT` als Tor.
- **E_RS, E_CTU etc.:** Nicht vergleichbar, da sie unterschiedliche Funktionalitäten (Set, Reset, Zählen) implementieren.

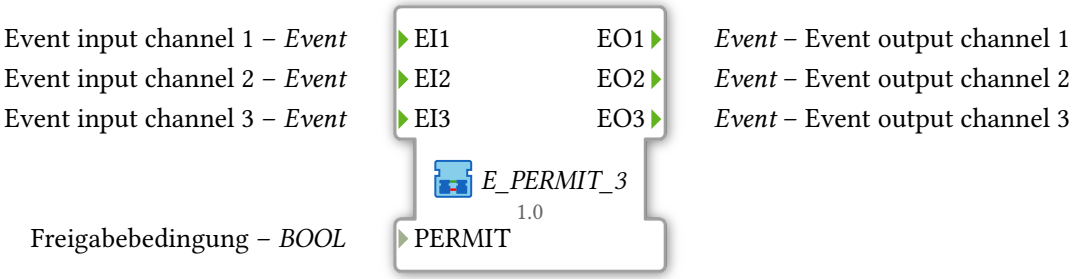
Gegenüber mehreren separaten `E_PERMIT`-Instanzen bietet `E_PERMIT_2` den Vorteil, dass das Freigabesignal nur einmal angeschlossen werden muss und eine gemeinsame Steuerung semantisch klar abgebildet wird.

Fazit

`E_PERMIT_2` ist ein nützlicher Baustein für Anwendungen, bei denen zwei Ereigniskanäle durch ein gemeinsames Freigabesignal kontrolliert werden müssen. Die SubApp-Implementierung kapselt die Logik und vereinfacht die Wiederverwendung. Mit nur einem booleschen Eingang und zwei Ereignis-Ein-/Ausgängen ist die Bedienung intuitiv und die Funktionsweise transparent. Dank der modularen Gestaltung lässt sich der Baustein in komplexe Automatisierungssysteme integrieren und bietet eine robuste Lösung für ereignisbasierte Steuerungen.

E_PERMIT_3





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **E_PERMIT_3** ist eine Subapplikation (SubApp), die drei unabhängige Ereignis-Freigabe-Gates (E_PERMIT) in einer kompakten Einheit bündelt. Sie dient dazu, drei parallele Ereigniskanäle wahlweise durchzuschalten oder zu blockieren – gesteuert durch ein gemeinsames Freigabesignal. Die SubApp ist geeignet für Anwendungen, in denen mehrere Ereignisse synchron freigegeben oder gesperrt werden müssen, ohne dass einzelne Instanzen separat konfiguriert werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Datentyp	Beschreibung
EI1	Event	Ereigniseingang für Kanal 1
EI2	Event	Ereigniseingang für Kanal 2
EI3	Event	Ereigniseingang für Kanal 3

Ereignis-Ausgänge

Name	Datentyp	Beschreibung
EO1	Event	Ereignisausgang für Kanal 1
EO2	Event	Ereignisausgang für Kanal 2

EO3	Event	Ereignisausgang für Kanal 3

Daten-Eingänge

Name	Datentyp	Beschreibung
PERMIT	BOOL	Zentrales Freigabesignal für alle drei Kanäle

Daten-Ausgänge

Keine vorhanden.

Adapter

Keine vorhanden.

Funktionsweise

Die SubApp **E_PERMIT_3** enthält intern drei Instanzen des standardisierten Funktionsblocks `iec61499::events::E_PERMIT`. Jeder dieser Bausteine fungiert als Ereignis-Gate: Er lässt ein ankommendes Ereignis (EI) nur dann zum Ausgang (EO) durch, wenn das zugehörige Freigabesignal (PERMIT) den Wert `TRUE` (bzw. `1`) besitzt. Ist das Freigabesignal `FALSE`, wird das Ereignis verworfen.

Die drei internen E_PERMIT-Bausteine sind parallel geschaltet und arbeiten völlig unabhängig voneinander – die Freigabebedingung wird jedoch über den externen Dateneingang `PERMIT` allen drei Bausteinen gleichzeitig zugeführt. Dadurch gilt die Freigabe global: Entweder werden alle drei Ereigniskanäle durchgeschaltet oder alle drei blockiert.

Die Ereignisverbindungen verbinden jeweils die externen Eingänge `EI1`, `EI2`, `EI3` mit den internen Eingängen der E_PERMIT-Instanzen, und die internen Ausgänge mit den externen Ausgängen `EO1`, `EO2`, `EO3`. Eine zusätzliche interne Logik oder zeitliche Verzögerung existiert nicht.

Technische Besonderheiten

- **Wiederverwendbarkeit:** Als SubApp ist E_PERMIT_3 als eigenständiger Baustein in Bibliotheken ablegbar und kann in verschiedenen Projekten mehrfach instanziiert werden.
- **Skalierbarkeit:** Durch die modulare Struktur lässt sich die Anzahl der Kanäle einfach erweitern, indem weitere E_PERMIT-Instanzen hinzugefügt werden.
- **Lizenz:** Der Baustein ist unter der Eclipse Public License 2.0 (EPL-2.0) veröffentlicht und kann frei verwendet und angepasst werden.
- **Kompatibilität:** Er folgt dem Standard IEC 61499-2 und ist damit in 4diac-IDE und anderen kompatiblen Systemen einsetzbar.

Zustandsübersicht

Da E_PERMIT_3 eine reine Zusammenstellung von E_PERMIT-Bausteinen ist, besitzt die SubApp selbst keinen internen Zustandsautomaten. Das Verhalten wird ausschließlich durch die enthaltenen E_PERMIT-Instanzen bestimmt.

Jede E_PERMIT-Instanz besitzt intern die folgenden Zustände:

Zustand	Beschreibung
IDLE	Warten auf ein Ereignis am Eingang EI.
PASS	Ereignis wird durchgereicht, sofern PERMIT = TRUE.
BLOCK	Ereignis wird verworfen, sofern PERMIT = FALSE.

Die Umschaltung zwischen PASS und BLOCK erfolgt unmittelbar, sobald sich der Wert von PERMIT ändert. Ereignisse, die während eines Zustandswechsels eintreffen, werden entsprechend dem aktuellen PERMIT-Wert behandelt.

Anwendungsszenarien

- **Mehrkanalige Sicherheitsfreigaben:** In Maschinensteuerungen müssen oft mehrere unabhängige Signale (z. B. Not-Halt, Schutztür, Startsignal) gleichzeitig freigegeben werden, bevor ein Prozess startet. E_PERMIT_3 kann als gemeinsames Gate dienen.
- **Synchronisierte Ereignisweiterleitung:** Wenn mehrere parallele Ereignisströme nur bei einer gemeinsamen Bedingung an nachgelagerte Logik weitergeleitet werden sollen.

- **Test- und Simulationsumgebungen:** Zur gezielten Unterdrückung oder Freischaltung mehrerer Ereignisquellen in einem System.

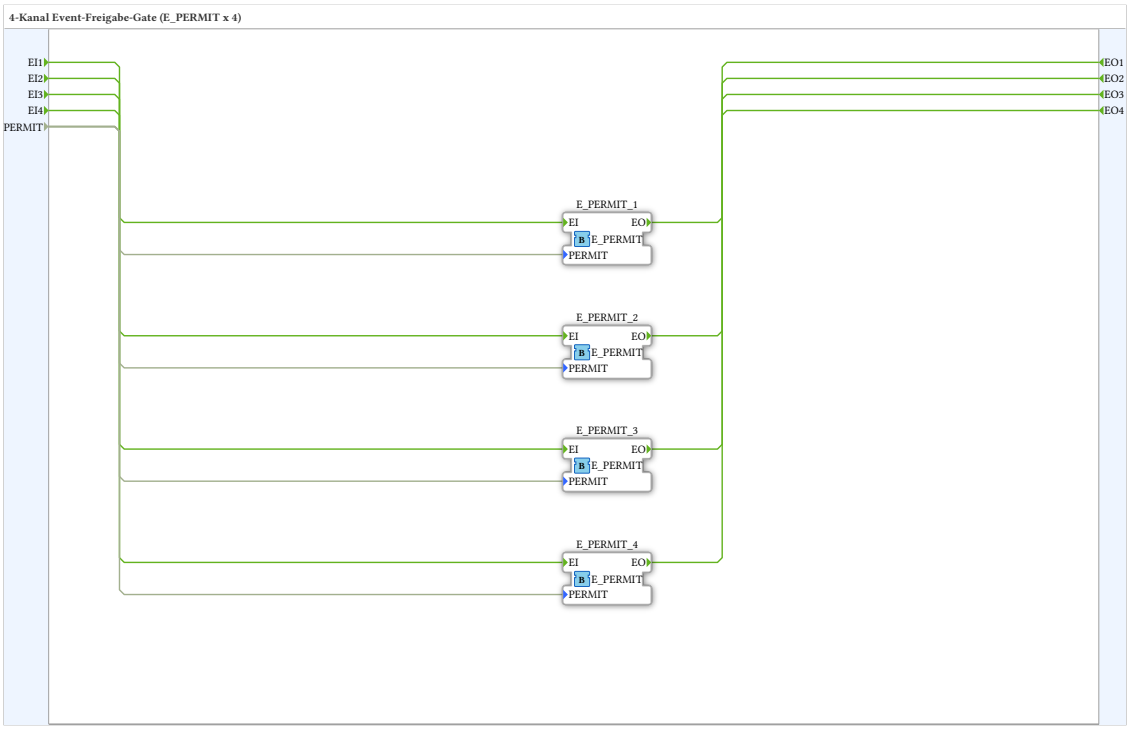
Vergleich mit ähnlichen Bausteinen

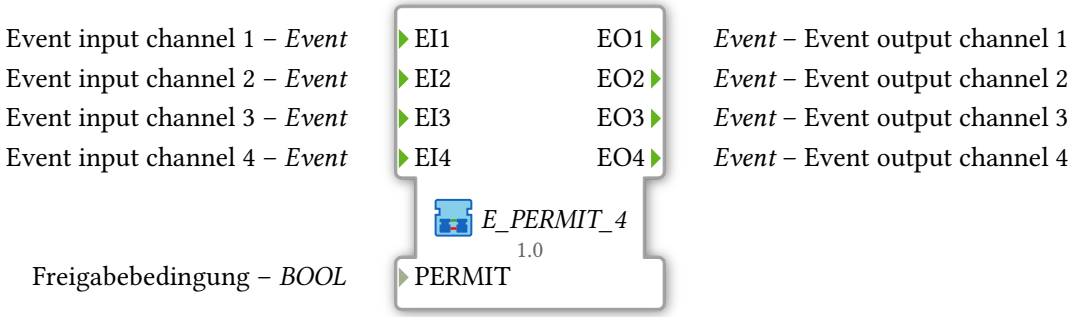
- **E_PERMIT (einzeln):** Bietet nur einen Kanal. Für mehrere Kanäle müssen mehrere Instanzen verdrahtet werden, wobei die Freigabesignale separat oder über gemeinsame Variablen verbunden werden müssen. E_PERMIT_3 vereinfacht dies durch eine vorgefertigte 3-Kanal-Lösung mit zentralem Freigabeeingang.
- **E_SWITCH:** Ein Ereignisverteiler, der Ereignisse an verschiedene Ausgänge weiterleitet, je nach Datenwert. E_PERMIT_3 hingegen blockiert oder lässt Ereignisse unverändert passieren, ohne eine Umleitung vorzunehmen.
- **E_REND:** Ein Ereignis-Gate, das nur bei steigender Flanke eines Bool-Signals durchlässig wird. E_PERMIT_3 verwendet stattdessen den aktuellen Pegel (Level) des Freigabesignals.

Fazit

Der Funktionsblock **E_PERMIT_3** ist eine einfach strukturierte, aber praxisorientierte SubApp zur effizienten Steuerung von drei Ereigniskanälen über eine gemeinsame Bool-Freigabe. Er reduziert den Verdrahtungsaufwand, erhöht die Übersichtlichkeit in Anlagen und ist dank seiner klaren Semantik vielseitig einsetzbar. Die Einhaltung des IEC-61499-Standards und die offene Lizenz machen ihn zu einem nützlichen Baustein für industrielle Automatisierungsprojekte und Lehrmaterialien gleichermaßen.

E_PERMIT_4





• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **E_PERMIT_4** ist eine Subapplikation (SubApp) und realisiert ein **4-Kanal-Event-Freigabe-Gate** auf Basis des IEC 61499-Standardbausteins `E_PERMIT`. Die SubApp bündelt vier voneinander unabhängige Ereignis-Freigabe-Kanäle, die über ein gemeinsames Freigabesignal gesteuert werden. Dadurch können vier separate Ereignispfade gleichzeitig mit einer einzigen Bedingung aktiviert oder gesperrt werden. Diese Struktur eignet sich besonders für Anwendungen, bei denen mehrere Event-Quellen synchron freigegeben oder blockiert werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
EI1	Event	Event input channel 1
EI2	Event	Event input channel 2
EI3	Event	Event input channel 3
EI4	Event	Event input channel 4

Ereignis-Ausgänge

Name	Typ	Kommentar
EO1	Event	Event output channel 1
EO2	Event	Event output channel 2
EO3	Event	Event output channel 3
EO4	Event	Event output channel 4

Daten-Eingänge

Name	Typ	Kommentar
PERMIT	BOOL	Freigabebedingung

Daten-Ausgänge

Keine vorhanden.

Adapter

Keine vorhanden.

Funktionsweise

Die SubApp enthält vier Instanzen des Standard-Funktionsbausteins `E_PERMIT` (`iec61499::events::E_PERMIT`), die parallel geschaltet sind. Der Daten-Eingang `PERMIT` wird auf alle vier Instanzen verteilt. Jeder der vier Ereignis-Eingänge (`EI1`–`EI4`) ist direkt mit dem Eingang `EI` der jeweiligen `E_PERMIT`-Instanz verbunden; die jeweiligen Ausgänge `EO` sind mit den entsprechenden Ereignis-Ausgängen (`EO1`–`EO4`) der SubApp verbunden.

Das Verhalten eines einzelnen `E_PERMIT`-Bausteins ist definiert als:

- Wenn `PERMIT = TRUE` und ein Ereignis am Eingang eintrifft, wird dieses Ereignis unmittelbar am Ausgang `EO` weitergegeben.
- Wenn `PERMIT = FALSE`, wird das eintreffende Ereignis nicht weitergeleitet und verworfen.

Da alle vier Kanäle dasselbe gemeinsame `PERMIT`-Signal verwenden, sind sie entweder alle gleichzeitig freigeschaltet oder alle gesperrt. Die Verarbeitung der vier Kanäle erfolgt unabhängig voneinander und ohne gegenseitige Beeinflussung.

Technische Besonderheiten

- **Gemeinsame Freigabe:** Ein einziges Bool-Signal `PERMIT` steuert alle vier Kanäle – keine Notwendigkeit für separate Freigaben pro Kanal.
- **Parallele Verarbeitung:** Jeder Kanal wird durch eine eigene `E_PERMIT`-Instanz realisiert, wodurch vollständige Unabhängigkeit der Ereignispfade gewährleistet ist.
- **Standardkonformität:** Es werden ausschließlich IEC 61499-Standardbausteine verwendet, was eine hohe Interoperabilität und Wiederverwendbarkeit ermöglicht.
- **Kein interner Zustand:** Die SubApp besitzt keine eigenen Zustandsvariablen oder Timer; sie ist rein ereignisgesteuert und reaktionsschnell.

Zustandsübersicht

Die SubApp selbst besitzt keinen expliziten internen Zustand. Da sie jedoch aus vier `E_PERMIT`-Bausteinen besteht, kann man pro Kanal die beiden logischen Zustände betrachten:

- **Freigegeben:** Solange `PERMIT = TRUE`, werden ankommende Ereignisse unverzögert an den Ausgang weitergegeben.
- **Gesperrt:** Bei `PERMIT = FALSE` werden eingehende Ereignisse blockiert und nicht weitergegeben.

Der Übergang zwischen diesen Zuständen erfolgt sofort bei einer Änderung des `PERMIT`-Signals und wirkt gleichzeitig auf alle vier Kanäle.

Anwendungsszenarien

- **Sicherheitsorientierte Steuerungen:** Freigabe von Ereignispfaden nur bei erfüllter Sicherheitsbedingung (z. B. Not-Halt nicht aktiv).
- **Parallele Aktorsteuerung:** Vier unabhängige Maschinen oder Prozesse werden über ein gemeinsames Freigabesignal gestartet oder gestoppt.
- **Event-Multiplexing:** Mehrere Ereignisquellen (z. B. Sensoren) sollen nur dann weiterverarbeitet werden, wenn eine übergeordnete Bedingung wahr ist.
- **Test- und Simulationsumgebungen:** Ein zentrales Enable-Signal kann genutzt werden, um komplette Ereignispfade für Testzwecke zu deaktivieren.

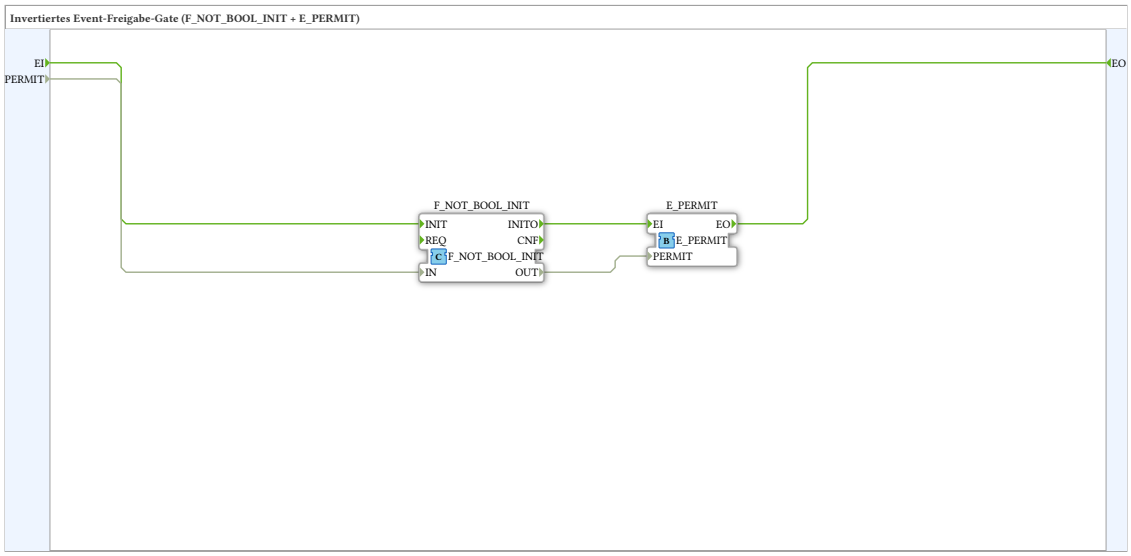
Vergleich mit ähnlichen Bausteinen

- **`E_PERMIT` (Standard):** Der einfache 1-Kanal-Baustein – diese SubApp erweitert ihn auf vier Kanäle mit gemeinsamer Freigabe, wodurch die Verdrahtung in Mehrkanal-Applikationen deutlich reduziert wird.
- **`E_SWITCH` (Event-Schalter):** Ermöglicht das Umschalten zwischen zwei Ereignispfaden (basiert auf einem Bool-Signal), während `E_PERMIT_4` nur zwischen Durchlassen und Blockieren unterscheidet.
- **`E_SPLIT` / `E_MERGE`:** Diese dienen der Verteilung bzw. Zusammenführung von Ereignissen unabhängig von einer Freigabebedingung – sie ergänzen `E_PERMIT_4` eher, als dass sie es ersetzen.
- **Eigene SubApp-Lösungen:** Viele Anwender bauen sich ähnliche Mehrfach-Freigaben selbst; `E_PERMIT_4` bietet eine vorgefertigte, klar dokumentierte und standardkonforme Alternative.

Fazit

Der Baustein **E_PERMIT_4** stellt eine kompakte und effiziente Lösung zur gleichzeitigen Freigabe von vier unabhängigen Ereignispfaden dar. Durch die Verwendung einer einzigen Freigabebedingung wird die Steuerlogik vereinfacht und die Übersichtlichkeit in komplexen Automatisierungsprojekten erhöht. Die SubApp ist ideal für Anwendungen, bei denen mehrere Ereignisströme synchron aktiviert oder deaktiviert werden müssen, und profitiert von der hohen Zuverlässigkeit der IEC 61499-Standardbausteine.

E_PERMIT_INVERT



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **E_PERMIT_INVERT** ist eine Subapplikation nach IEC 61499-2. Er kombiniert einen booleschen Negationsbaustein mit einem Ereignis-Freigabe-Baustein und realisiert so ein **invertiertes Event-Freigabe-Gate**. Ein ankommendes Ereignis an `EI` wird nur dann an `EO` weitergegeben, wenn der Eingang `PERMIT` den Wert `FALSE` besitzt.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
`EI`	Event	Event-Eingang, wird durchgelassen, wenn `PERMIT` `FALSE` ist.

Ereignis-Ausgänge

Name	Typ	Kommentar
`EO`	Event	Event-Ausgang, feuert, wenn `EI` eintrifft und `PERMIT` `FALSE` ist.

Daten-Eingänge

Name	Typ	Kommentar
`PERMIT`	BOOL	Invertierte Freigabebedingung. `FALSE` bedeutet Freigabe, `TRUE` bedeutet Sperrung.

Daten-Ausgänge

Keine.

Adapter

Keine.

Funktionsweise

Die Subapplikation besteht intern aus zwei Bausteinen:

- F_NOT_BOOL_INIT zur booleschen Negation
- E_PERMIT zur ereignisbasierten Freigabe

Der externe Eingang PERMIT wird an den Eingang IN des Negationsbausteins geführt. Dessen Ausgang OUT ist mit dem Freigabeeingang PERMIT des internen E_PERMIT verbunden. Ein Ereignis an EI wird direkt an den Ereigniseingang von E_PERMIT weitergeleitet.

Dadurch gilt:

- Externer PERMIT = FALSE ⟶ intern freigegeben ⟶ EI erzeugt EO
- Externer PERMIT = TRUE ⟶ intern gesperrt ⟶ EI erzeugt kein EO

Die Negation ist vollständig in der Subapplikation gekapselt. Die Umgebung sieht nur das gewünschte invertierte Freigabeverhalten.

Technische Besonderheiten

- E_PERMIT_INVERT ist als SubAppType mit einem internen Netzwerk modelliert.
- Die Negation wird durch den IEC-61131-Baustein F_NOT_BOOL_INIT ausgeführt.
- Das interne E_PERMIT bewertet die Freigabe erst beim Eintreffen eines Ereignisses an EI.
- Eine Änderung von PERMIT allein erzeugt kein Ausgangsereignis.
- Der Baustein besitzt keine Datenausgänge und keine Adapter.
- Die Freigabelogik ist in einem einzigen wiederverwendbaren Baustein gekapselt.

Zustandsübersicht

Da die Subapplikation keine eigene persistente Zustandsmaschine besitzt, ergibt sich die folgende äquivalente Zustandstabelle:

Externer PERMIT	Interner Freigabewert	Verhalten bei EI
FALSE	TRUE	EO wird ausgelöst
TRUE	FALSE	EO wird nicht ausgelöst

Anwendungsszenarien

- **Aktiv-Low-Freigabe:** Ein Ereignis soll nur dann durchlaufen, wenn ein Freigabesignal FALSE ist.
- **Sicherheitslogik:** Ereignisse werden blockiert, solange ein Sicherheitssignal TRUE ist.
- **Modusumschaltung:** Eine Handsteuerung wird nur bei deaktiviertem Automatikmodus zugelassen.
- **Wiederverwendbare Kapselung:** Die invertierte Freigabelogik kann an mehreren Stellen einer Anlage identisch verwendet werden.

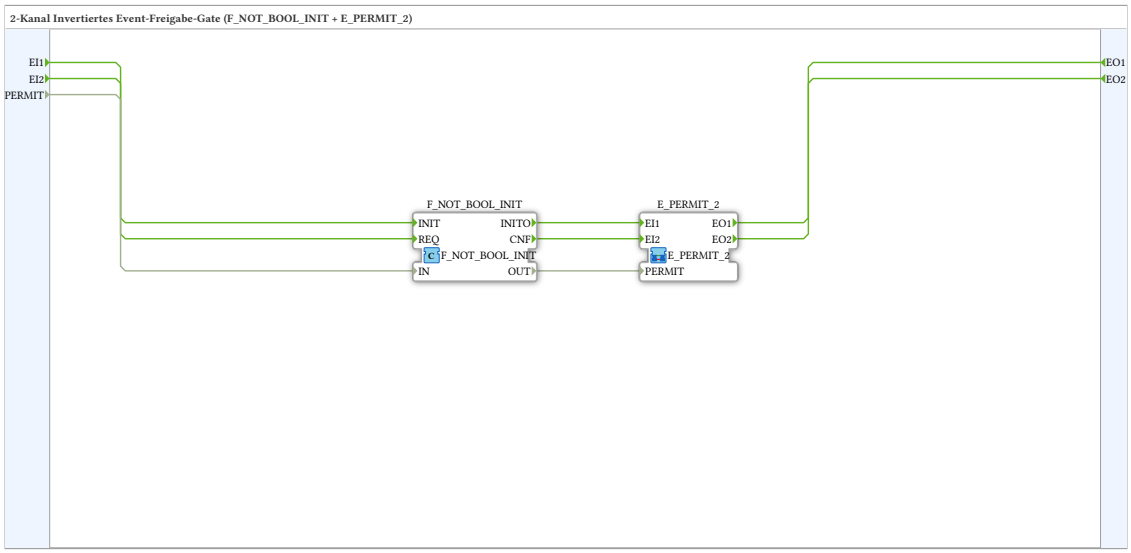
Vergleich mit ähnlichen Bausteinen

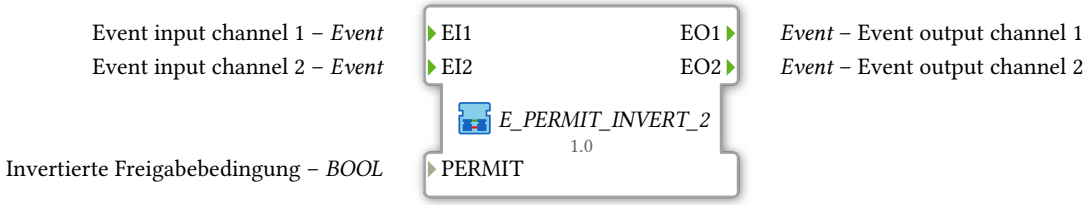
Baustein	Verhalten
E_PERMIT	Gibt EI an EO weiter, wenn PERMIT TRUE ist.
E_PERMIT_INVERT	Gibt EI an EO weiter, wenn PERMIT FALSE ist.
F_NOT_BOOL_INIT	Negiert einen BOOL-Wert, besitzt aber keine Event-Durchschaltung.
Externe Kombination aus F_NOT_BOOL_INIT und E_PERMIT	Gleiche Logik, benötigt jedoch zwei Bausteine und zusätzliche Verdrahtung.

Fazit

E_PERMIT_INVERT ist eine kompakte, klar abgegrenzte Subapplikation für invertierte Event-Freigaben. Sie kapselt die Negation und die Freigabe in einem einzigen Baustein, reduziert die externe Verdrahtung und macht die Steuerungslogik dadurch lesbarer und wartbarer. Besonders für aktiv-low-Freigaben ist dieser Baustein eine passende und wiederverwendbare Lösung.

E_PERMIT_INVERT_2





• – ◦ • – ◦ • – ◦ •

Einleitung

Bei `E_PERMIT_INVERT_2` handelt es sich um eine Subapplikation, die ein invertiertes Event-Freigabe-Gate mit zwei unabhängigen Ereigniskanälen realisiert. Der Baustein kombiniert einen IEC-61131-Not-Baustein `F_NOT_BOOL_INIT` mit einem IEC-61499-Event-Gate `E_PERMIT_2`.

Die äußere Freigabebedingung `PERMIT` wird vor der eigentlichen Ereignisweitergabe negiert. Dadurch ist der Baustein für Low-aktive Freigabesignale geeignet: Liegt `PERMIT` auf `FALSE`, werden Ereignisse durchgeschaltet; liegt `PERMIT` auf `TRUE`, werden Ereignisse blockiert.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
`EI1`	Event	Ereignis-Eingangskanal 1
`EI2`	Event	Ereignis-Eingangskanal 2

Ereignis-Ausgänge

Name	Typ	Kommentar
`EO1`	Event	Ereignis-Ausgangskanal 1
`EO2`	Event	Ereignis-Ausgangskanal 2

Daten-Eingänge

Name	Typ	Kommentar
`PERMIT`	BOOL	Invertierte Freigabebedingung

Daten-Ausgänge

Es sind keine Daten-Ausgänge vorhanden.

Adapter

Es sind keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Die Subapplikation besitzt zwei Ereignispfade:

- `EI1` ⟶ `EO1`
- `EI2` ⟶ `EO2`

Beide Pfade werden durch das interne Gate `E_PERMIT_2` gesteuert. Der externe Eingang `PERMIT` wird zunächst durch den Baustein `F_NOT_BOOL_INIT` invertiert. Das Ergebnis dieser Negation wird an den Freigabeeingang `PERMIT` von `E_PERMIT_2` übergeben.

Dadurch gilt:

- Externer `PERMIT` = `FALSE`
  Interner Freigabewert = `TRUE`
  Ereignisse werden von `EI1` nach `EO1` und von `EI2` nach `EO2` weitergeleitet.
- Externer `PERMIT` = `TRUE`
  Interner Freigabewert = `FALSE`
  Ereignisse werden blockiert und es wird kein Ereignisausgang aktiviert.

Technische Besonderheiten

- Es handelt sich um eine Subapplikation, nicht um einen direkt implementierten Funktionsblock.
- Der Baustein verwendet `F_NOT_BOOL_INIT` aus der IEC-61131-Bibliothek und `E_PERMIT_2` aus der IEC-61499-Ereignisbibliothek.
- Die beiden Ereigniskanäle sind getrennt geführt, werden aber gemeinsam über ein einziges Freigabesignal gesteuert.
- Der Baustein besitzt keine Datenausgänge und kein eigenes internes Zustandsgedächtnis.
- Durch die Integration der Negation in die Subapplikation wird eine extern sichtbare Freigabelogik mit invertierter Bedeutung erreicht.

Zustandsübersicht

Da die Subapplikation kein eigenes Speicherverhalten besitzt, ist die folgende Tabelle als logische Wirkungskette zu verstehen:

Externer `PERMIT`	Internes Gate `PERMIT`	Ereignisverhalten
`FALSE`	`TRUE`	`EI1` ⟶ `EO1`, `EI2` ⟶ `EO2`
`TRUE`	`FALSE`	Keine Ereignisweitergabe

Anwendungsszenarien

- Einsatz in Steuerungen, bei denen die Freigabe durch ein Low-aktives Signal erfolgt, zum Beispiel Sicherheitskontakte oder Schutztüren.
- Zweikanalige Ereignisweiterleitung mit einer gemeinsamen Sperrbedingung.
- Ersatz für eine externe Verdrahtung von `F_NOT_BOOL_INIT` und `E_PERMIT_2`, wodurch die Applikation übersichtlicher wird.

Vergleich mit ähnlichen Bausteinen

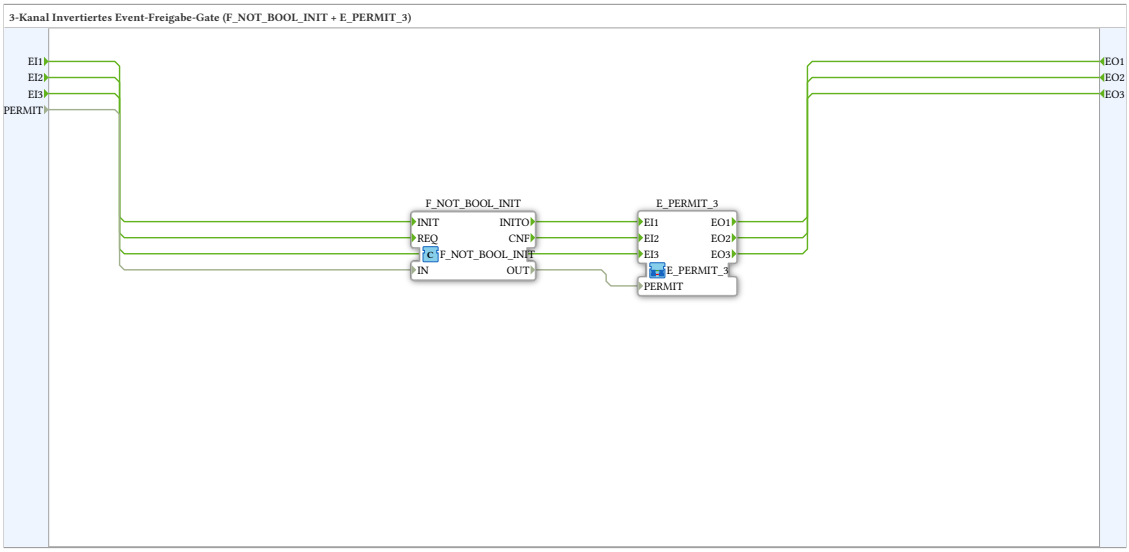
Im Gegensatz zum Standardbaustein `E_PERMIT_2`, bei dem ein Ereignis bei `PERMIT = TRUE` durchgeschaltet wird, arbeitet `E_PERMIT_INVERT_2` mit negierter Freigabe. Die Funktion entspricht einer fest verdrahteten Kombination aus `F_NOT_BOOL_INIT` und `E_PERMIT_2`.

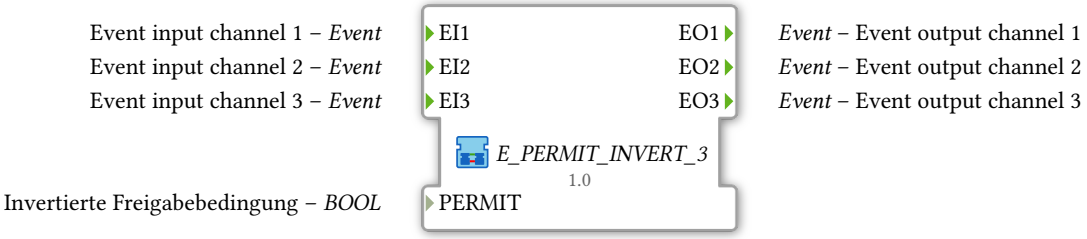
Vorteil der Subapplikation ist die Kapselung der invertierten Logik. Dadurch wird eine versehentliche falsche Verdrahtung vermieden und die Wiederverwendbarkeit erhöht.

Fazit

`E_PERMIT_INVERT_2` ist ein kompakter, wiederverwendbarer Baustein für zweikanalige Ereignis-Freigaben mit invertierter Logik. Er kombiniert Standardbausteine zu einer klaren Schnittstelle und ermöglicht eine einfache Integration Low-aktiver Freigabesignale in IEC-61499-Anwendungen.

E_PERMIT_INVERT_3





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **E_PERMIT_INVERT_3** ist eine Subapplikation (SubApp) für die 4diac-IDE. Er stellt ein 3-Kanal-Ereignis-Freigabe-Gate mit invertierter Freigabelogik dar. Die SubApp kombiniert die Funktionsbausteine `F_NOT_BOOL_INIT` (boolesche Negation) und `E_PERMIT_3` (3-Kanal-Event-Freigabe) zu einer kompakten, wiederverwendbaren Einheit.

Über den Eingang `PERMIT` wird gesteuert, ob die eingehenden Ereignisse an den jeweiligen Ausgängen durchgeschaltet oder blockiert werden. Die Besonderheit liegt in der Invertierung: Ist `PERMIT = FALSE`, sind die Ereignisse freigegeben; ist `PERMIT = TRUE`, werden sie gesperrt.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Kommentar
`EI1`	Event input channel 1
`EI2`	Event input channel 2
`EI3`	Event input channel 3

Ereignis-Ausgänge

Name	Kommentar

EO1	Event output channel 1
EO2	Event output channel 2
EO3	Event output channel 3

Daten-Eingänge

Name	Datentyp	Kommentar
PERMIT	BOOL	Invertierte Freigabebedingung

Daten-Ausgänge

Keine vorhanden.

Adapter

Keine vorhanden.

Funktionsweise

Die SubApp besteht intern aus zwei Funktionsbausteinen:

- F_NOT_BOOL_INIT – negiert den booleschen Eingang PERMIT.
- E_PERMIT_3 – ein 3-Kanal-Event-Freigabebaustein, der die eingehenden Ereignisse nur dann an die Ausgänge weiterleitet, wenn sein Freigabeeingang PERMIT den Wert TRUE hat.

Der Datenfluss ist wie folgt:

1. Der boolesche Wert von PERMIT wird an den Eingang IN des Bausteins F_NOT_BOOL_INIT übergeben.
2. F_NOT_BOOL_INIT invertiert diesen Wert und liefert das Ergebnis an seinem Ausgang OUT.
3. Das invertierte Signal wird an den Freigabeeingang PERMIT von E_PERMIT_3 angeschlossen.
4. Die Ereignisse an EI1, EI2 und EI3 werden intern mit den entsprechenden Eingängen von E_PERMIT_3 verbunden.
5. Abhängig vom negierten Freigabesignal werden die Ereignisse an EO1, EO2 und EO3 weitergeleitet oder blockiert.

Die folgende Wahrheitstabelle verdeutlicht das Verhalten für einen der drei Kanäle (beispielhaft für Kanal 1):

PERMIT (extern)	Negierter Wert (intern)	Ereignis EI1	Ausgang EO1
FALSE	TRUE	eintreffend	wird durchgeschaltet
TRUE	FALSE	eintreffend	wird blockiert

Technische Besonderheiten

- Die SubApp kapselt die Kombination aus Negation und Ereignisfreigabe in einem einzigen wiederverwendbaren Typ.
- Es werden keine eigenen Algorithmen oder internen Zustände definiert; die Logik wird vollständig durch die beiden Standardbausteine realisiert.
- Der Baustein F_NOT_BOOL_INIT ist eine initialisierbare Negation. In dieser Konfiguration wird kein Initialwert gesetzt; das Verhalten entspricht einer einfachen Negation des Eingangssignals.
- Die SubApp besitzt drei voneinander unabhängige Kanäle, sodass gleichzeitig mehrere Ereignispfade mit derselben Freigabebedingung gesteuert werden können.
- Da es sich um eine SubApplikation handelt, ist die innere Struktur in der 4diac-IDE sichtbar und kann bei Bedarf erweitert oder angepasst werden.

Zustandsübersicht

Die SubApp selbst besitzt keinen expliziten endlichen Zustandsautomaten. Das Verhalten wird durch die Freigabelogik bestimmt. Es lassen sich jedoch zwei logische Zustände für jeden Kanal unterscheiden:

Zustand	Bedingung (extern)	Verhalten

Freigegeben	`PERMIT = FALSE`	Eingehende Ereignisse werden an den entsprechenden Ausgang weitergeleitet.
Gesperrt	`PERMIT = TRUE`	Eingehende Ereignisse werden verworfen und nicht an den Ausgang durchgereicht.

Diese Zustände gelten für alle drei Kanäle gleichermaßen.

Anwendungsszenarien

- **Sicherheitslogik**: Ein Ereignis soll nur dann verarbeitet werden, wenn ein Sicherheitssignal *inaktiv* ist. Beispiel: `PERMIT` ist `TRUE`, wenn eine Schutzhaube geöffnet ist; die Ereignisse sollen dann gesperrt werden.
- **Betriebsartenumschaltung**: Wenn ein Signal wie „Handbetrieb“ aktiv ist (`TRUE`), sollen automatische Verarbeitungs-Events blockiert werden. Bei „Automatik“ (`FALSE`) werden die Events durchgelassen.
- **Mehrkanalige Freigabe**: Gleichzeitige Steuerung von drei unabhängigen Ereignispfaden mit einer einzigen, invertierten Freigabebedingung.
- **NOT-AUS-Verknüpfung**: Bei aktivem NOT-AUS (`PERMIT = TRUE`) werden alle prozessrelevanten Events unterdrückt; erst bei zurückgesetztem NOT-AUS (`PERMIT = FALSE`) werden die Events wieder zugelassen.

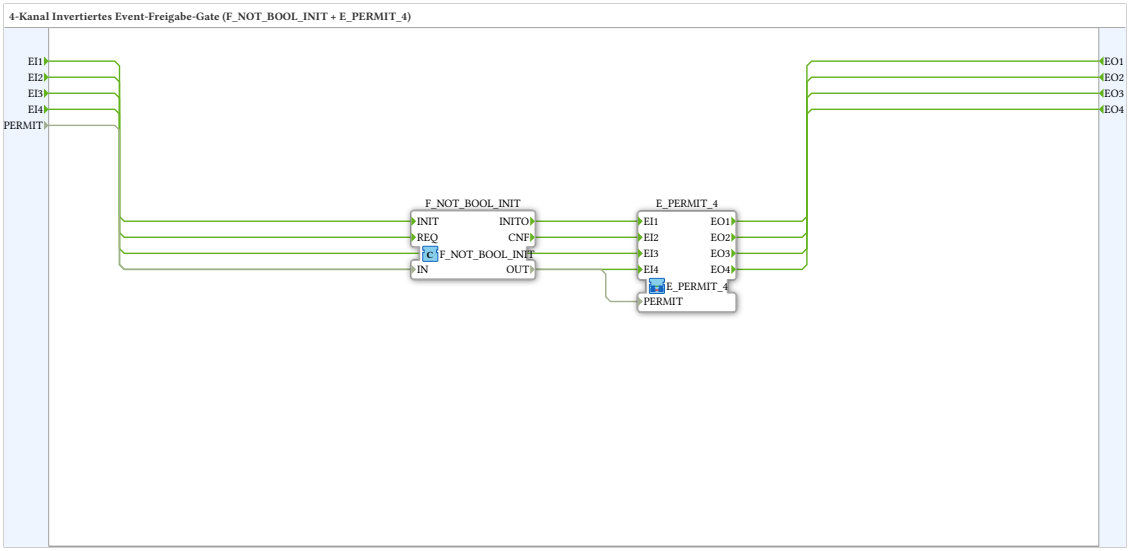
Vergleich mit ähnlichen Bausteinen

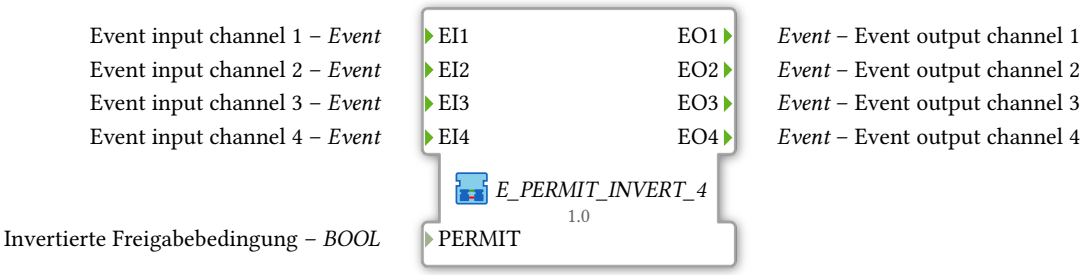
- **`E_PERMIT_3`**: Besitzt eine positive Freigabe. Ereignisse werden bei `PERMIT = TRUE` durchgelassen und bei `FALSE` blockiert. `E_PERMIT_INVERT_3` kehrt diese Logik um.
- **`E_PERMIT`**: Bietet die gleiche Funktion für nur einen Kanal. `E_PERMIT_INVERT_3` erweitert das Konzept auf drei Kanäle und integriert die Negation direkt.
- **Einzelne Negation + `E_PERMIT_3`**: Dieselbe Funktion könnte durch Vorschalten eines `F_NOT_BOOL_INIT` vor einem `E_PERMIT_3` erreicht werden. Die SubApp kapselt diese Verschaltung und erleichtert dadurch die Anwendung und Wartung.

Fazit

`E_PERMIT_INVERT_3` ist ein praktischer, wiederverwendbarer Baustein, der eine invertierte Ereignisfreigabe über drei Kanäle realisiert. Durch die Kombination von Negation und Multi-Permit-Logik in einer SubApp wird die Steuerungslogik übersichtlicher und der Verdrahtungsaufwand reduziert. Er eignet sich besonders für Sicherheits- und Betriebsartenkonzepte, bei denen Ereignisse nur im inaktiven Zustand eines Freigabesignals verarbeitet werden dürfen.

E_PERMIT_INVERT_4





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `E_PERMIT_INVERT_4` ist eine Subapplikation mit vier Ereigniskanälen. Sie realisiert ein invertiertes Event-Freigabe-Gate: Ein Ereignis wird nur dann vom Eingang zum Ausgang durchgeschaltet, wenn das Eingangssignal `PERMIT` den Wert `FALSE` besitzt. Dazu wird `PERMIT` intern durch den Funktionsbaustein `F_NOT_BOOL_INIT` invertiert und das invertierte Signal an einen `E_PERMIT_4`-Baustein übergeben.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
`EI1`	Event	Ereignis-Eingangskanal 1
`EI2`	Event	Ereignis-Eingangskanal 2
`EI3`	Event	Ereignis-Eingangskanal 3
`EI4`	Event	Ereignis-Eingangskanal 4

Ereignis-Ausgänge

Name	Typ	Kommentar
`EO1`	Event	Ereignis-Ausgangskanal 1
`EO2`	Event	Ereignis-Ausgangskanal 2
`EO3`	Event	Ereignis-Ausgangskanal 3
`EO4`	Event	Ereignis-Ausgangskanal 4

Daten-Eingänge

Name	Typ	Kommentar
`PERMIT`	BOOL	Invertierte Freigabebedingung; `TRUE` sperrt, `FALSE` gibt frei

Daten-Ausgänge

Keine.

Adapter

Keine.

Funktionsweise

Intern besteht die Subapplikation aus zwei verbundenen Bausteinen:

- `PERMIT` wird an den Eingang `IN` des Bausteins `F_NOT_BOOL_INIT` geführt.
- Der Ausgang `OUT` von `F_NOT_BOOL_INIT` ist mit dem Freigabeeingang `PERMIT` des Bausteins `E_PERMIT_4` verbunden.

- Die Ereignisse EI1 bis EI4 werden direkt auf die entsprechenden Ereigniseingänge von E_PERMIT_4 geführt.
- Die Ereignisausgänge von E_PERMIT_4 sind direkt mit EO1 bis EO4 verbunden.

E_PERMIT_4 lässt ein Ereignis nur dann passieren, wenn sein PERMIT-Eingang den Wert TRUE hat. Da F_NOT_BOOL_INIT das externe PERMIT-Signal invertiert, ist die interne Freigabe nur dann TRUE, wenn das externe PERMIT den Wert FALSE besitzt.

Die resultierende Logik:

Externer PERMIT	Interner PERMIT bei E_PERMIT_4	Ereignis EIx zu EOx
TRUE	FALSE	gesperrt / verworfen
FALSE	TRUE	durchgeschaltet

Eine Änderung des PERMIT-Signals allein erzeugt keine Ereignisse. Es wird ausschließlich entschieden, ob ein ankommendes Ereignis an den jeweiligen Ausgang weitergegeben wird oder nicht.

Technische Besonderheiten

- Die Subapplikation realisiert eine Low-aktive Freigabelogik: FALSE bedeutet „freigegeben“, TRUE bedeutet „gesperrt“.
- Sie kapselt die Invertierung des Freigabesignals und die vierkanalige Ereignisweitergabe in einem einzigen logischen Baustein.
- Es sind keine Adapter und keine Datenausgänge vorhanden.
- Die Subapplikation besitzt keine eigene Ereignisverarbeitung; sie ist rein durchschaltend und ereignisgetrieben.
- Die interne Logik reagiert nicht auf steigende oder fallende Flanken von PERMIT, sondern wertet den aktuellen Booleschen Wert nur dann aus, wenn ein Ereignis an EI1 bis EI4 eintrifft.

Zustandsübersicht

Die Subapplikation besitzt keine explizite Zustandsmaschine. Das Verhalten kann über die Freigabebedingung beschrieben werden:

- Solange PERMIT = TRUE ist, werden eintreffende Ereignisse an EI1 bis EI4 verworfen.
- Solange PERMIT = FALSE ist, werden eintreffende Ereignisse an EI1 bis EI4 zu den entsprechenden Ausgängen EO1 bis EO4 durchgeschaltet.

Damit ergibt sich ein einfaches, stateless Gate-Verhalten auf Basis des aktuellen Werts von PERMIT.

Anwendungsszenarien

- Einsatz als „Sperr-Gate“, wenn Ereignisse nur dann verarbeitet werden sollen, wenn ein bestimmtes Sperrsignal nicht aktiv ist.
- Verwendung in Steuerungslogiken, in denen ein Signal eine Aktion blockiert, aber nicht explizit freigibt.
- Einbindung in Sicherheits- oder Überwachungskonzepte, bei denen Ereignisse nur im nicht gesperrten Zustand weitergeleitet werden dürfen.
- Kapselung einer invertierten Freigabelogik für vier unabhängige Ereigniskanäle in einer wiederverwendbaren Subapplikation.

Vergleich mit ähnlichen Bausteinen

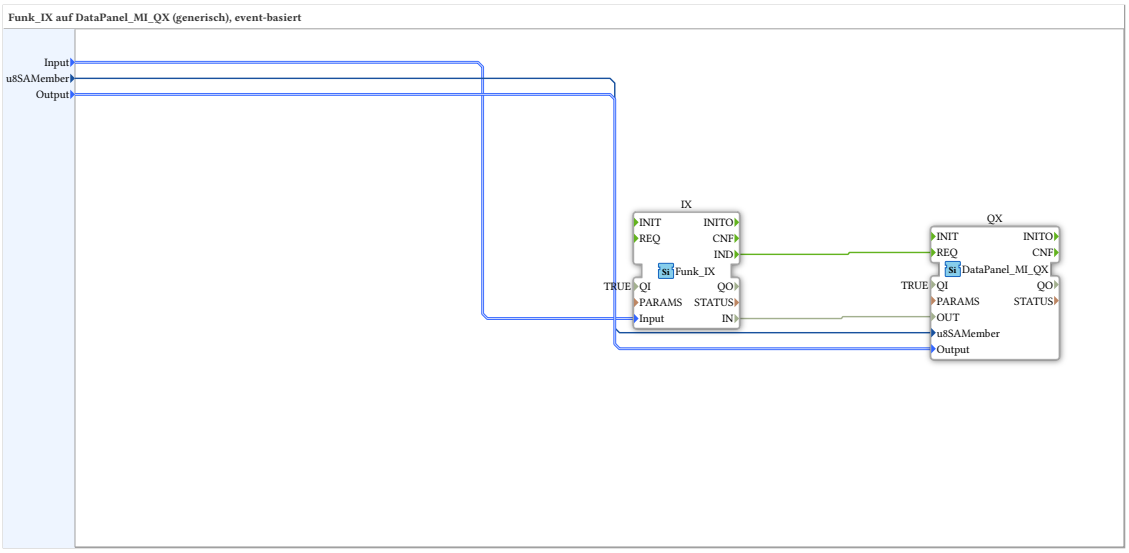
Im Vergleich zum Standardbaustein E_PERMIT_4 besitzt E_PERMIT_INVERT_4 eine invertierte Freigabelogik. Während E_PERMIT_4 bei PERMIT = TRUE durchschaltet, gibt E_PERMIT_INVERT_4 erst bei PERMIT = FALSE frei.

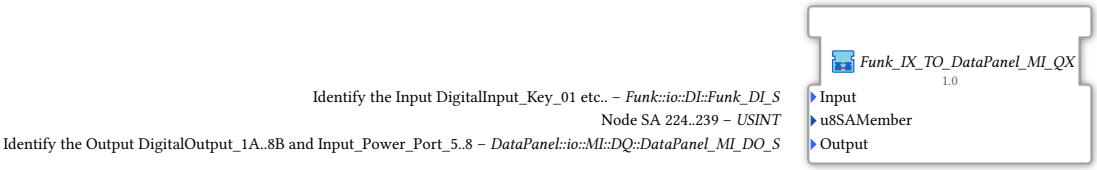
Gegenüber einer manuellen Kombination aus einem separaten NOT-Baustein und einem E_PERMIT_4 bietet die Subapplikation eine kompaktere und klar benannte Einheit. Dadurch wird die Freigabelogik an der Schnittstelle besser verständlich und die interne Verschaltung kann nicht versehentlich falsch aufgebaut werden.

Fazit

E_PERMIT_INVERT_4 ist ein nützlicher Baustein für Anwendungen, die eine invertierte Freigabebedingung über mehrere Ereigniskanäle benötigen. Durch die interne Kapselung von Invertierung und Freigabe wird die Applikationslogik übersichtlicher und die Wiederverwendung vereinfacht.

Funk_IX_TO_DataPanel_MI_QX





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **Funk_IX_TO_DataPanel_MI_QX** stellt eine wiederverwendbare Subapplikation dar, die einen drahtlosen Eingang (Funk_IX) mit einem Ausgabemodul des DataPanels (DataPanel_MI_QX) verbindet. Er dient dazu, ein über Funk empfangenes Signal (z. B. von einem Taster) in einen definierten Ausgangszustand am DataPanel umzusetzen. Die Subapplikation ist ereignisbasiert und ermöglicht eine generische Zuordnung von Eingangssignalen zu Ausgängen.

Schnittstellenstruktur

Die Subapplikation besitzt ausschließlich Dateneingänge und keine Ausgänge oder Ereignisschnittstellen nach außen.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Name	Typ	Kommentar	Initialwert
`Input`	`Funk::io::DI::Funk_DI_S`	Identifiziert den digitalen Eingang (z. B. `DigitalInput_Key_01`)	`Funk_DI::Invalid`
`u8SAMember`	`USINT`	Knoten-Subadresse (SA) im Bereich 224..239	`MI::MI_00`
`Output`	`DataPanel::io::MI::DQ::DataPanel_MI_DO_S`	Identifiziert den Ausgang (z. B. `DigitalOutput_1A..8B` oder `Input_Power_Port_5..8`)	`Invalid`

Daten-Ausgänge

Keine.

Adapter

Keine.

Funktionsweise

Die Subapplikation enthält zwei interne Funktionsblöcke:

- **Funk_IX**: Empfängt ein drahtloses Eingangssignal und liefert über seinen Ereignisausgang `IND` eine positive Flanke, wenn ein gültiges Signal eingetroffen ist.
- **DataPanel_MI_QX**: Verarbeitet den übergebenen Datenwert und steuert den entsprechenden Ausgang des DataPanels.

Der interne Ablauf:

1. Der eingehende Datenwert am externen Eingang `Input` wird an den Eingang `IX.Input` des Funk_IX-Blocks geleitet.
2. Sobald der Funk_IX ein Ereignis generiert (Ereignisausgang `IND`), wird dieses Ereignis an den Ereigniseingang `REQ` des DataPanel_MI_QX weitergegeben.
3. Gleichzeitig wird der Wert vom Ausgang `IX.IN` (der den empfangenen Zustand repräsentiert) an den Dateneingang `OUT` des DataPanel_MI_QX übergeben.
4. Der DataPanel_MI_QX verwendet die Werte `u8SAMember` und `Output`, um den Zielausgang zu identifizieren und den übermittelten Zustand zu setzen.

Die Parameter `QI` sind intern auf `TRUE` gesetzt, sodass die Blöcke permanent aktiviert sind. Die Zuordnung zwischen Eingang und Ausgang erfolgt durch die externen Parameter, sodass derselbe Baustein für verschiedene Funkkanäle und Ausgänge konfiguriert werden kann.

Technische Besonderheiten

- **Ereignisbasiert**: Die Übertragung des Signals erfolgt über ein Ereignis vom Funk_IX zum DataPanel-Modul, was eine effiziente Verarbeitung nur bei tatsächlichen Ereignissen ermöglicht.
- **Generische Konfiguration**: Durch die Parameter `Input`, `u8SAMember` und `Output` wird der Baustein universell einsetzbar, ohne die interne Logik ändern zu müssen.
- **Keine Ausgänge**: Die Subapplikation gibt keine Daten nach außen weiter; alle Ergebnisse werden direkt am DataPanel wirksam.
- **Interne Initialisierung**: Beide Funktionsblöcke sind mit `QI = TRUE` initialisiert, wodurch sie sofort betriebsbereit sind.
- Die Subapplikation ist für den Einsatz mit der Hardwarefamilie DataPanel und Funk-IO konzipiert.

Zustandsübersicht

Die Subapplikation besitzt keinen expliziten Zustandsautomaten. Ihr Verhalten ist rein ereignisgesteuert:

- Im Ruhezustand wartet der Funk_IX auf ein ankommendes Funksignal.
- Sobald ein Signal empfangen wird, erzeugt er ein Ereignis, das den DataPanel_MI_QX anstößt.
- Nach der Verarbeitung kehrt der Baustein in den Wartezustand zurück.

Es gibt also keine dauerhaften Zustände, sondern nur die aktive Verarbeitung eines Ereignisses pro Zyklus.

Anwendungsszenarien

- **Drahtlose Steuerung von Ausgängen**: Ein Funk-Taster sendet ein Signal, das über diesen Baustein einen bestimmten Ausgang am DataPanel (z. B. eine Leuchte oder einen Antrieb) aktiviert.
- **Einbindung in größere Automatisierungssysteme**: Die Subapplikation kann in übergeordnete Steuerungslogiken eingebettet werden, um Funkempfänger mit Ausgabemodulen zu verknüpfen.
- **Flexible Konfiguration**: Durch die Parametrierung kann derselbe Baustein für verschiedene Funkkanäle (unterschiedliche `Input`-Werte) und verschiedene Ausgänge (`Output`) eingesetzt werden, ohne Änderungen am internen Netzwerk vorzunehmen.

Vergleich mit ähnlichen Bausteinen

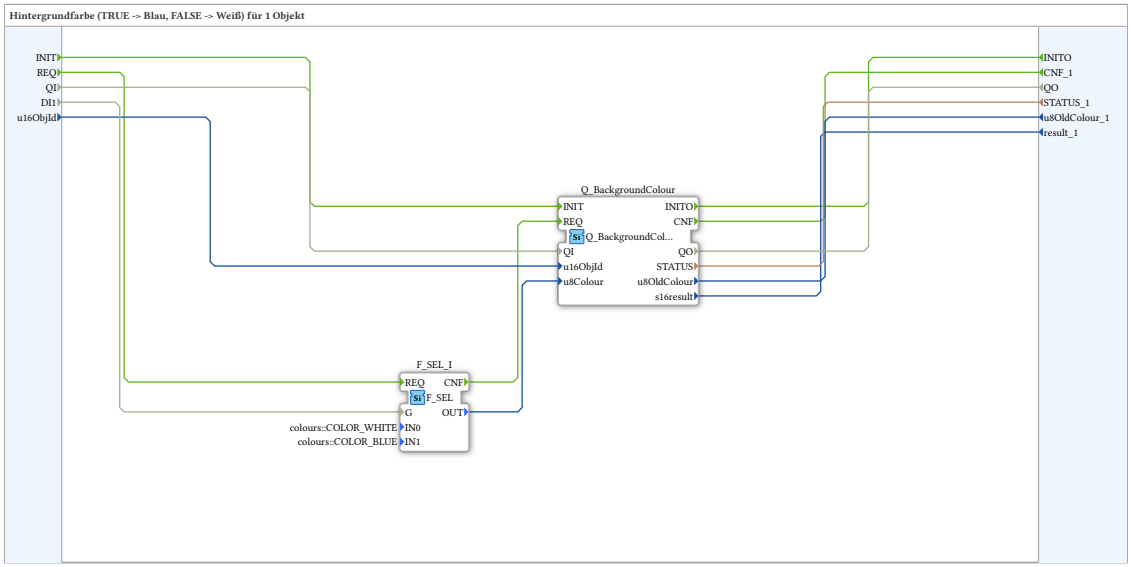
Im Gegensatz zu fest verdrahteten Koppelbausteinen bietet dieser Baustein eine höhere Flexibilität durch die generischen Parameter. Er ähnelt einem Adapter, der eine ereignisbasierte Verbindung zwischen einem Funk-Input-Modul und einem Output-Modul herstellt. Im Vergleich zu einem reinen Datenbaustein ohne

Ereignissteuerung ist hier die Reaktion unmittelbar an das Eintreffen eines Ereignisses gebunden, was unerwünschte Ausgaben ohne neues Funksignal vermeidet. Ein direkter Vergleich zu ähnlichen Subapplikationen ist nicht vorhanden, da diese spezifisch für die Kombination der IO-Module entwickelt wurde.

Fazit

Funk_IX_TO_DataPanel_MI_QX ist ein kompakter, ereignisbasierter Baustein, der die Integration von Funkempfängern und DataPanel-Modulen erheblich vereinfacht. Durch seine generische Parametrierung ist er wiederverwendbar und an verschiedene Anforderungen anpassbar. Die interne Architektur ist transparent und ermöglicht eine robuste Übertragung von Eingangssignalen zu definierten Ausgängen. Für Anwendungen, die eine schnelle und flexible drahtlose Steuerung erfordern, stellt dieser Baustein eine effektive Lösung dar.

GreenBlueBackground1

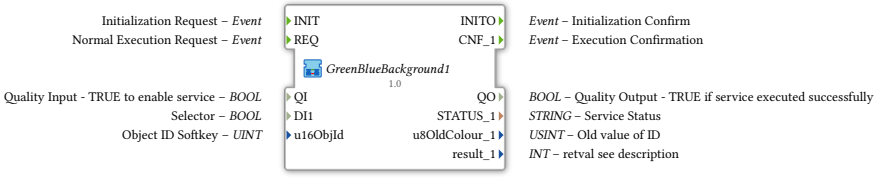


• – ○ • – ○ • – ○ •

Einleitung

`GreenBlueBackground1` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Blau**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den Eingang `u16ObjId` übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Schnittstelle: Initialisierung und Qualität

- **INIT** (Ereignis-Eingang, `EInit`) ⟶ `Q_BackgroundColour.INIT`; `Q_BackgroundColour.INITO` ⟶ **INITO** (Ereignis-Ausgang, `EInit`).
- **QI** (Daten-Eingang, `BOOL`, Vorgabe `TRUE`): Quality Input – `TRUE` aktiviert den Dienst; durchgereicht an `Q_BackgroundColour.QI`.
- **QO** (Daten-Ausgang, `BOOL`): Quality Output – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde; kommt von `Q_BackgroundColour.QO`.

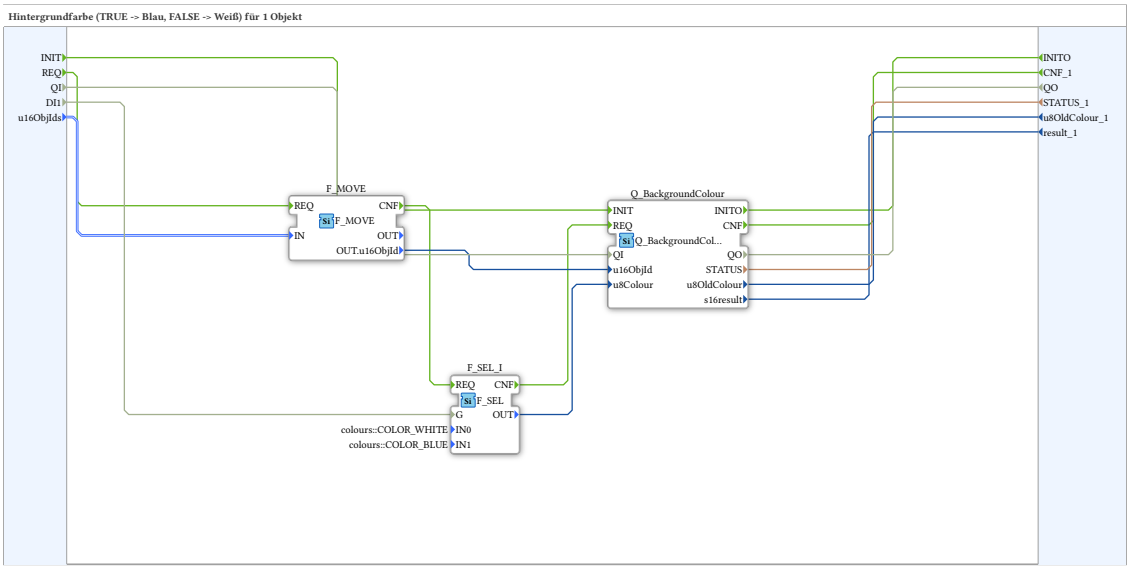
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Blau/Weiß, 1 Objekt, BOOL-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenBlueBackground1S

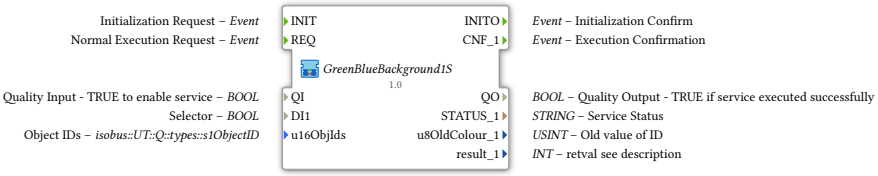


• — ○ • — ○ • — ○ •

Einleitung

`GreenBlueBackground1S` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Blau**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Schnittstelle: Initialisierung und Qualität

- **INIT** (Ereignis-Eingang, `EInit`) ⟶ `Q_BackgroundColour.INIT`; `Q_BackgroundColour.INITO` ⟶ **INITO** (Ereignis-Ausgang, `EInit`).
- **QI** (Daten-Eingang, `BOOL`, Vorgabe `TRUE`): Quality Input – `TRUE` aktiviert den Dienst; durchgereicht an `Q_BackgroundColour.QI`.
- **QO** (Daten-Ausgang, `BOOL`): Quality Output – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde; kommt von `Q_BackgroundColour.QO`.

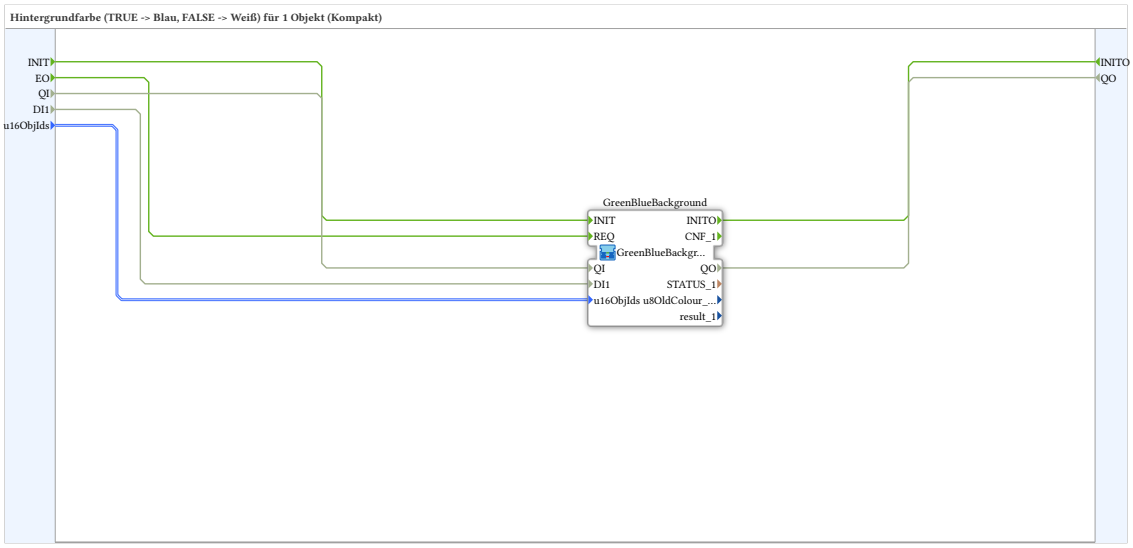
Zusammenfassung

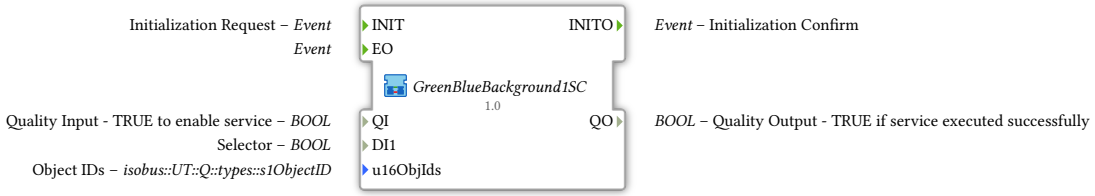
Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Blau/Weiß, 1 Objekt, BOOL-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenBlueBackground1SC





• – ○ • – ○ • – ○ •

Einleitung

`GreenBlueBackground1SC` ist ein **Kompakt-Wrapper** um `GreenBlueBackground1S` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

Initialisierung und Status (INIT/INITO, QI/QO)

- **Ereignis-Eingang** `INIT` (`EInit`, Initialization Request) und **Ereignis-Ausgang** `INITO` (`EInit`, Initialization Confirm)
- **Daten-Eingang** `QI` (`BOOL`, Vorgabe `TRUE`, Quality Input) wird an `GreenBlueBackground.QI` weitergegeben
- **Daten-Ausgang** `QO` (`BOOL`, Quality Output): durchgereicht von `GreenBlueBackground.QO`
- **Initialisierungskette**: `INIT` ⟶ `GreenBlueBackground.INIT`, `GreenBlueBackground.INITO` ⟶ `INITO`

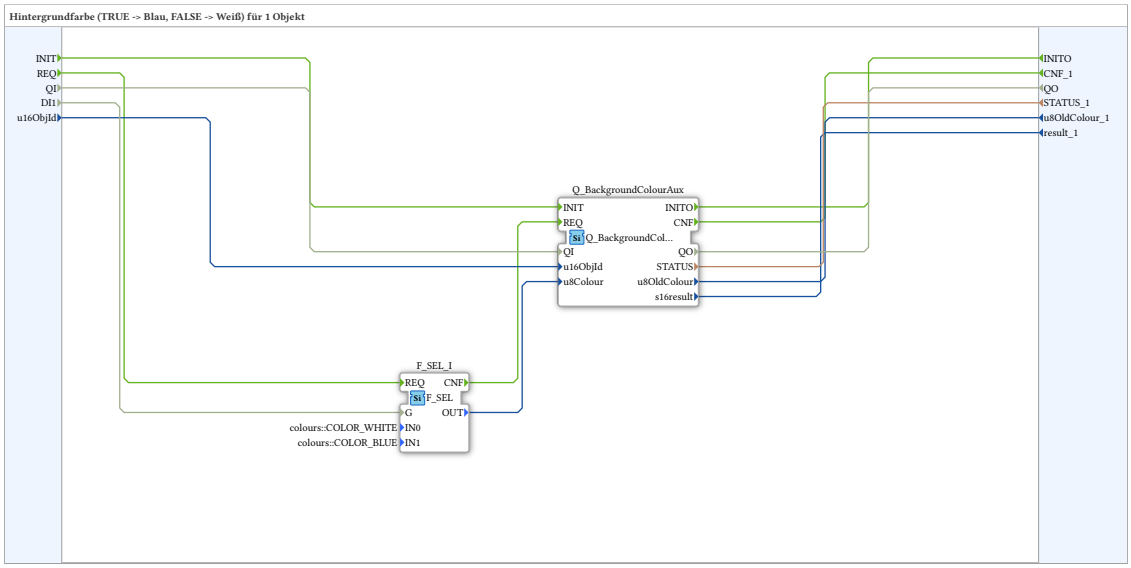
Zusammenfassung

Reine Weiterleitung an `GreenBlueBackground1S` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenBlueBackground1_aux

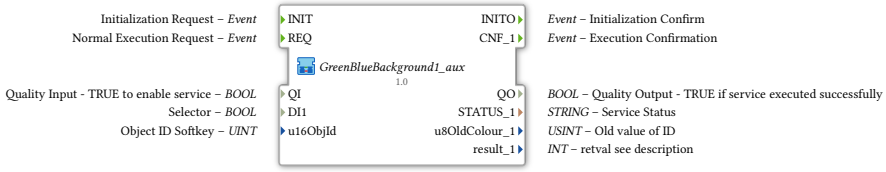


• – ○ • – ○ • – ○ •

Einleitung

`GreenBlueBackground1_aux` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Blau**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den Eingang `u16ObjId` übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Schnittstelle: Initialisierung und Qualität

- **INIT** (Ereignis-Eingang, `EInit`) ⟶ `Q_BackgroundColourAux.INIT`; `Q_BackgroundColourAux.INITO` ⟶ **INITO** (Ereignis-Ausgang, `EInit`).
- **QI** (Daten-Eingang, `BOOL`, Vorgabe `TRUE`): Quality Input – `TRUE` aktiviert den Dienst; durchgereicht an `Q_BackgroundColourAux.QI`.
- **QO** (Daten-Ausgang, `BOOL`): Quality Output – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde; kommt von `Q_BackgroundColourAux.QO`.

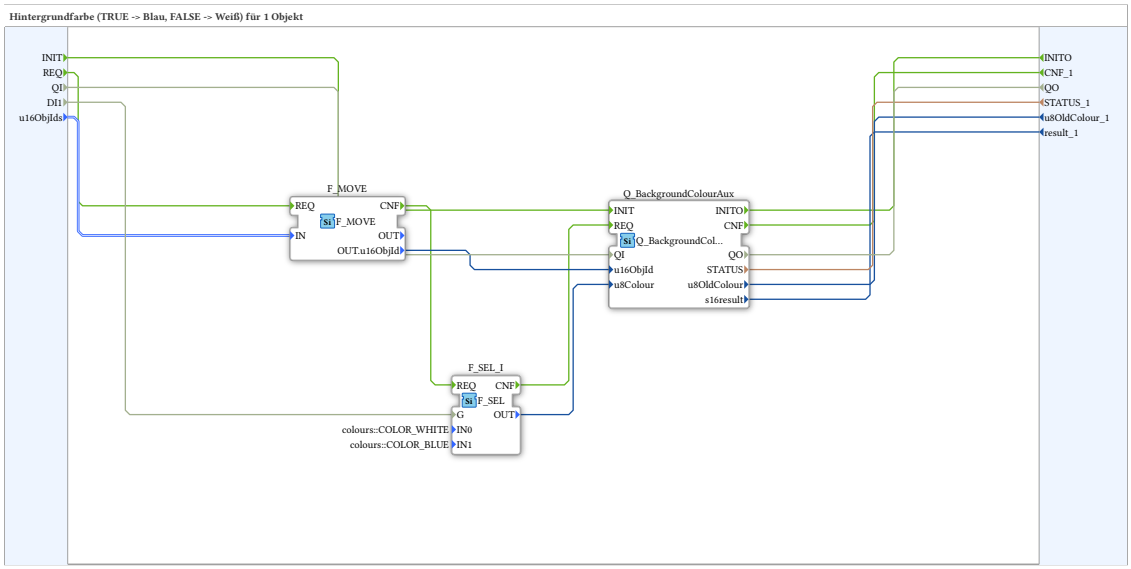
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Blau/Weiß, 1 Objekt, BOOL-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenBlueBackground1_auxS

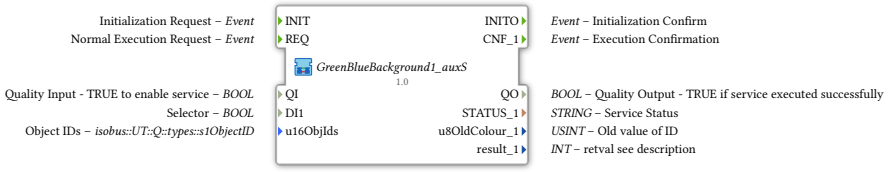


• – ○ • – ○ • – ○ •

Einleitung

`GreenBlueBackground1_auxS` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Blau**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Schnittstelle: Initialisierung und Qualität

- **INIT** (Ereignis-Eingang, `EInit`) ⟶ `Q_BackgroundColourAux.INIT`; `Q_BackgroundColourAux.INITO` ⟶ **INITO** (Ereignis-Ausgang, `EInit`).
- **QI** (Daten-Eingang, `BOOL`, Vorgabe `TRUE`): Quality Input – `TRUE` aktiviert den Dienst; durchgereicht an `Q_BackgroundColourAux.QI`.
- **QO** (Daten-Ausgang, `BOOL`): Quality Output – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde; kommt von `Q_BackgroundColourAux.QO`.

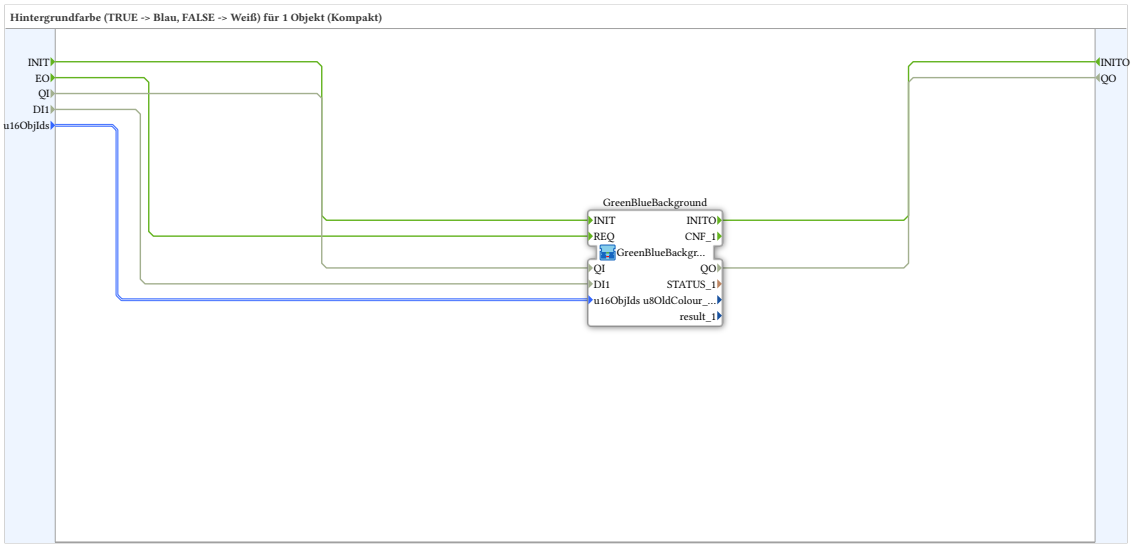
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Blau/Weiß, 1 Objekt, BOOL-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenBlueBackground1_auxSC



• — ○ • — ○ • — ○ •

Einleitung

`GreenBlueBackground1_auxSC` ist ein **Kompakt-Wrapper** um `GreenBlueBackground1_auxS` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen. Zusätzlich werden `INIT` ⟶ `GreenBlueBackground.INIT`, `GreenBlueBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenBlueBackground.QI` (Initialwert `TRUE`) und `GreenBlueBackground.QO` ⟶ `QO` durchgereicht.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

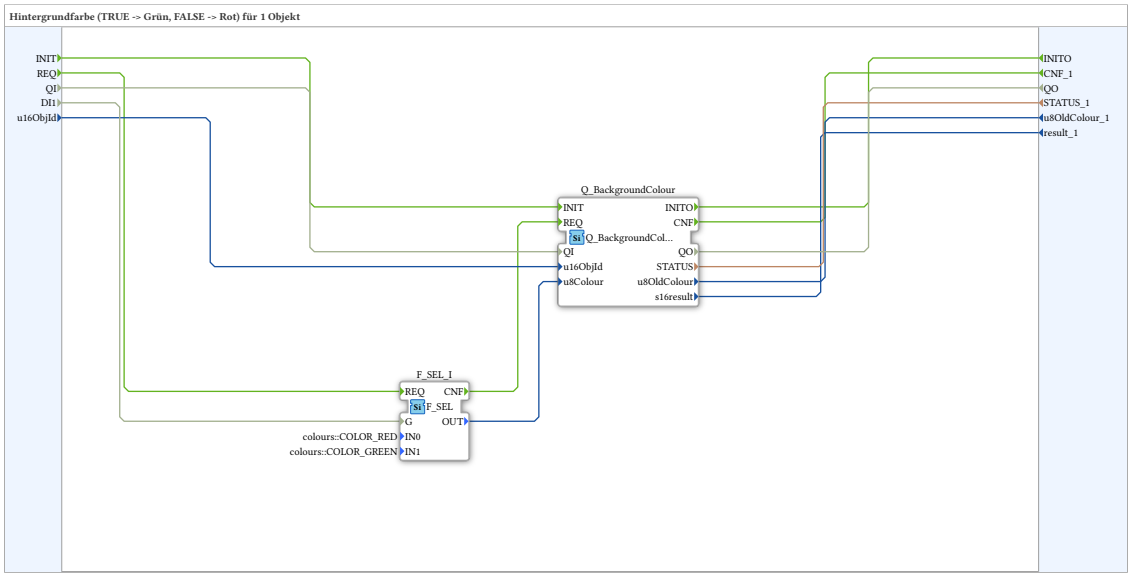
Zusammenfassung

Reine Weiterleitung an `GreenBlueBackground1_auxS` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenRedBackground1

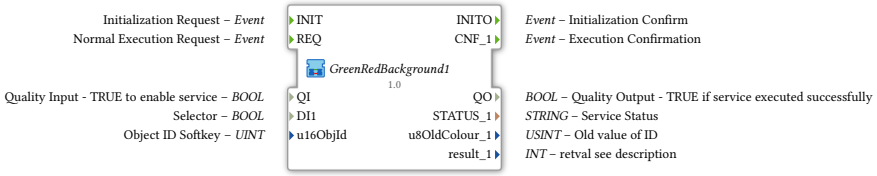


• – ○ • – ○ • – ○ •

Einleitung

`GreenRedBackground1` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den Eingang `u16ObjId` übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Schnittstelle: Initialisierung und Qualität

- **INIT** (Ereignis-Eingang, `EInit`) ⟶ `Q_BackgroundColour.INIT`; `Q_BackgroundColour.INITO` ⟶ **INITO** (Ereignis-Ausgang, `EInit`).
- **QI** (Daten-Eingang, `BOOL`, Vorgabe `TRUE`): Quality Input – `TRUE` aktiviert den Dienst; durchgereicht an `Q_BackgroundColour.QI`.
- **QO** (Daten-Ausgang, `BOOL`): Quality Output – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde; kommt von `Q_BackgroundColour.QO`.

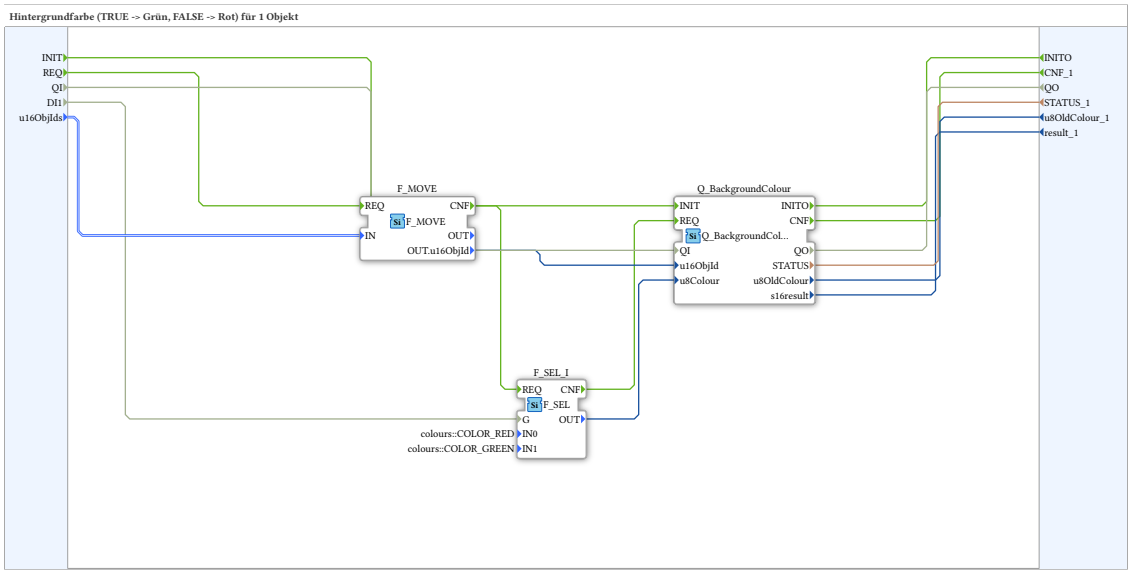
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Rot, 1 Objekt, BOOL-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenRedBackground1S

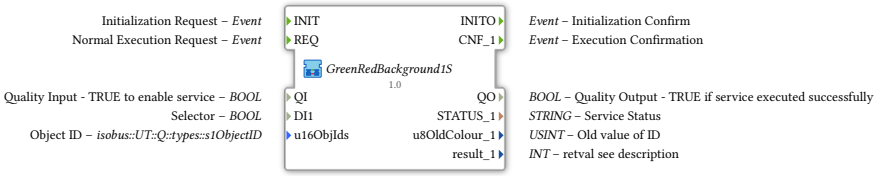


• – ○ • – ○ • – ○ •

Einleitung

`GreenRedBackground1S` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Schnittstelle: Initialisierung und Qualität

- **INIT** (Ereignis-Eingang, `EInit`) ⟶ `Q_BackgroundColour.INIT`; `Q_BackgroundColour.INITO` ⟶ **INITO** (Ereignis-Ausgang, `EInit`).
- **QI** (Daten-Eingang, `BOOL`, Vorgabe `TRUE`): Quality Input – `TRUE` aktiviert den Dienst; durchgereicht an `Q_BackgroundColour.QI`.
- **QO** (Daten-Ausgang, `BOOL`): Quality Output – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde; kommt von `Q_BackgroundColour.QO`.

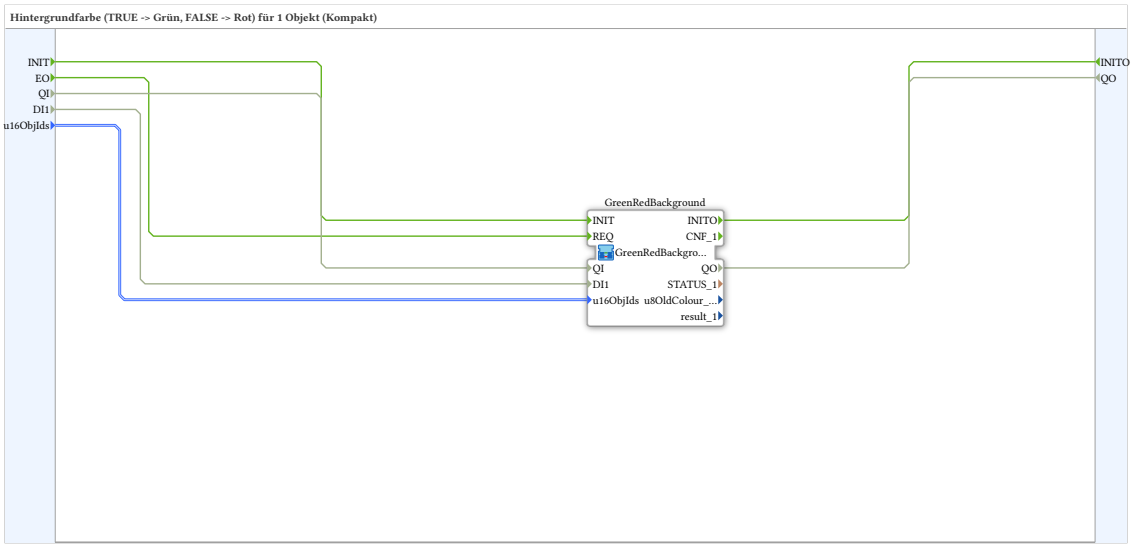
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Rot, 1 Objekt, BOOL-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenRedBackground1SC



• – ○ • – ○ • – ○ •

Einleitung

`GreenRedBackground1SC` ist ein **Kompakt-Wrapper** um `GreenRedBackground1S` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Zusätzlich werden Initialisierung und Qualität durchgereicht: `INIT` ⟶ `GreenRedBackground.INIT`, `GreenRedBackground.INITO` ⟶ `INITO`, `QI` (Initialwert `TRUE`) ⟶ `GreenRedBackground.QI` und `GreenRedBackground.QO` ⟶ `QO`.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

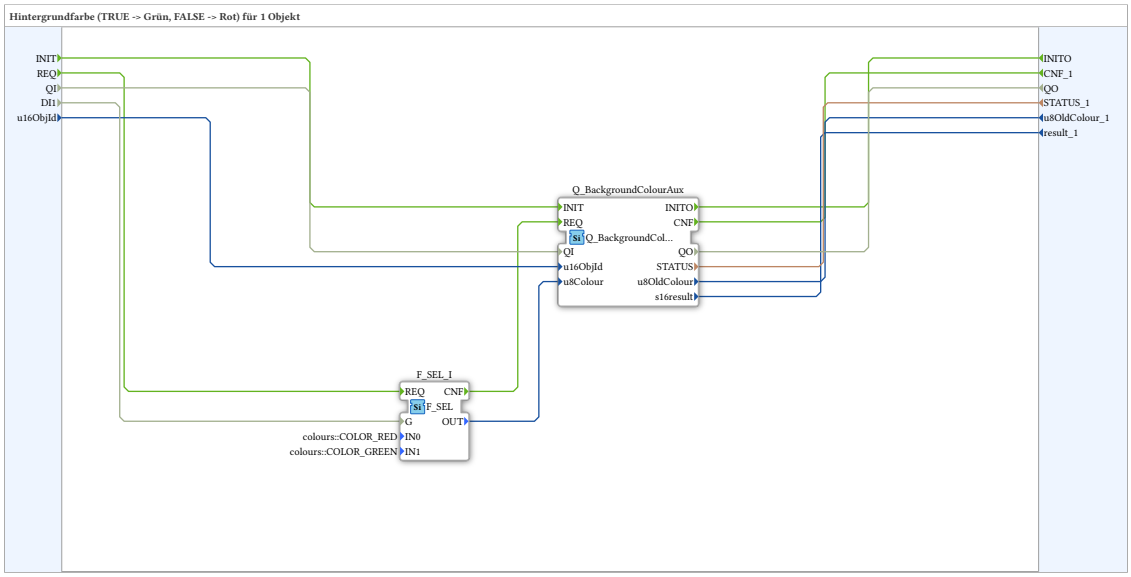
Zusammenfassung

Reine Weiterleitung an `GreenRedBackground1S` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenRedBackground1_aux



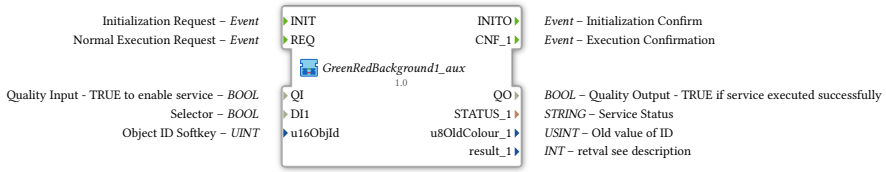
• – ○ • – ○ • – ○ •

Einleitung

`GreenRedBackground1_aux` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den Eingang `u16ObjId` übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Quality-Anschlüsse: Der Baustein besitzt zusätzlich das Ereignis `INIT` (Eingang) und `INITO` (Ausgang) sowie den Dateneingang `QI` (BOOL, Standard `TRUE`, „Quality Input – TRUE, um den Dienst freizugeben“) und den Datenausgang `QO` (BOOL, „Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde“). `INIT` wird an `Q_BackgroundColourAux.INIT` weitergereicht, `Q_BackgroundColourAux.INITO` an `INITO`; `QI` ist mit `Q_BackgroundColourAux.QI` und `Q_BackgroundColourAux.QO` mit `QO` verbunden.



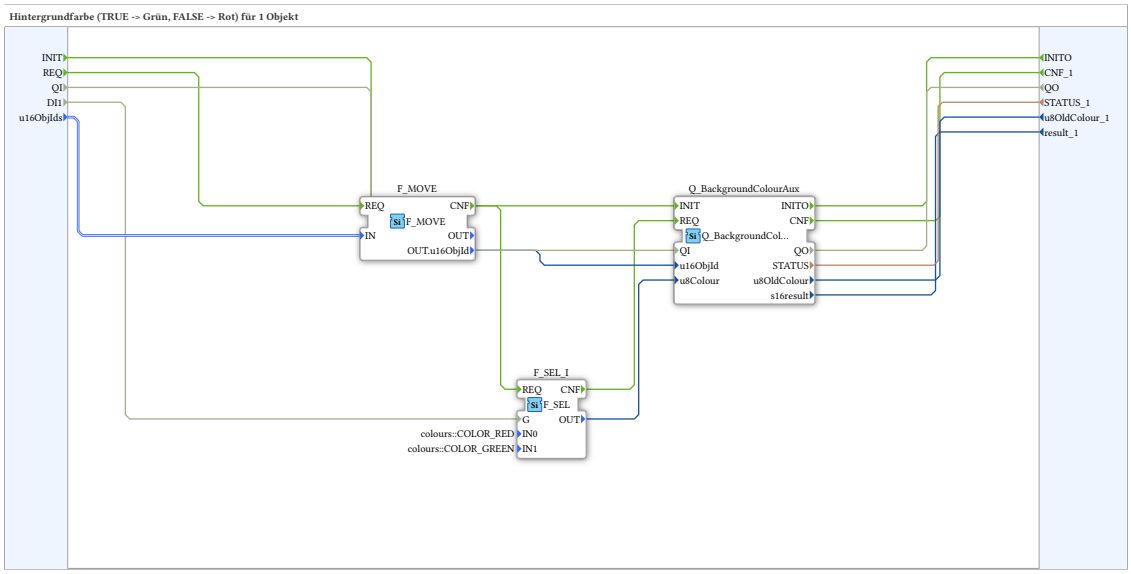
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Rot, 1 Objekt, BOOL-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenRedBackground1_auxS



• – ○ • – ○ • – ○ •

Einleitung

`GreenRedBackground1_auxS` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Quality-Anschlüsse: Der Baustein besitzt zusätzlich das Ereignis `INIT` (Eingang) und `INITO` (Ausgang) sowie den Dateneingang `QI` (BOOL, Standard `TRUE`, „Quality Input – TRUE, um den Dienst freizugeben“) und den Datenausgang `QO` (BOOL, „Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde“). `INIT` wird an `Q_BackgroundColourAux.INIT` weitergereicht, `Q_BackgroundColourAux.INITO` an `INITO`; `QI` ist mit `Q_BackgroundColourAux.QI` und `Q_BackgroundColourAux.QO` mit `QO` verbunden.

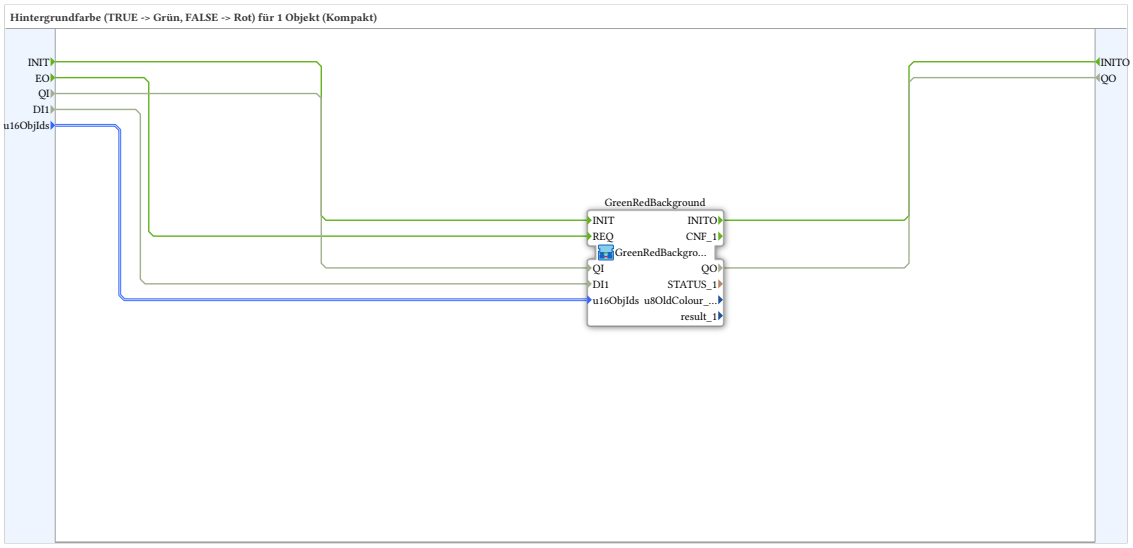
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Rot, 1 Objekt, BOOL-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenRedBackground1_auxSC



• – ○ • – ○ • – ○ •

Einleitung

`GreenRedBackground1_auxSC` ist ein **Kompakt-Wrapper** um `GreenRedBackground1_auxS` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

Zusätzlich führt der Wrapper die Schnittstelle für Initialisierung und Qualität nach außen: Ereigniseingang `INIT`, Ereignisausgang `INITO`, Dateneingang `QI` (`BOOL`, Initialwert `TRUE`) und Datenausgang `QO` (`BOOL`). Sie sind 1:1 an die innere Instanz `GreenRedBackground` durchgeschleift: `INIT` ⟶ `GreenRedBackground.INIT`, `GreenRedBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenRedBackground.QI`, `GreenRedBackground.QO` ⟶ `QO`.

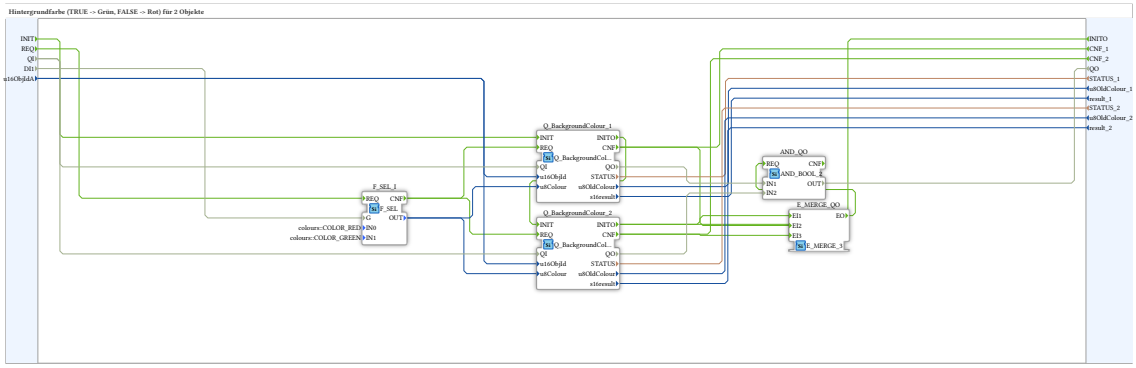
Zusammenfassung

Reine Weiterleitung an `GreenRedBackground1_auxS` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenRedBackground2



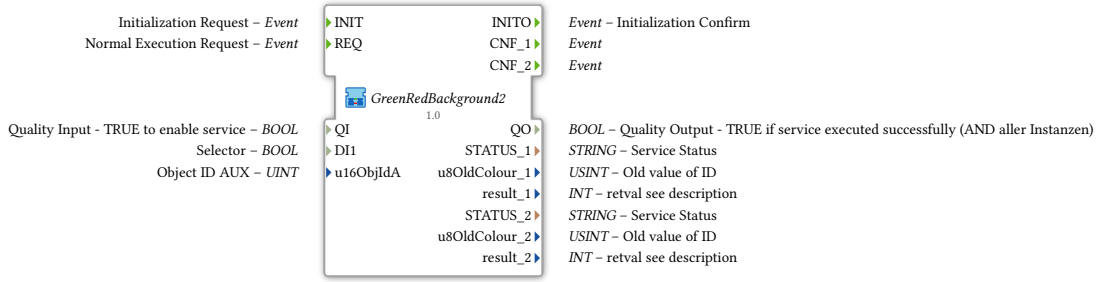
• – ○ • – ○ • – ○ •

Einleitung

`GreenRedBackground2` schaltet die VT-Hintergrundfarbe von 2 Objekten anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über die Eingänge `u16ObjIdA` übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
2	`u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)

Allgemeines Muster (Selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Initialisierung und Qualität

- **Ereignis-Eingang `INIT`** (`EInit`) und **Ereignis-Ausgang `INITO`** (`EInit`): die Initialisierung wird durch die `Q_BackgroundColour`-Instanzen gekettet: `INIT` ⟶ `Q_BackgroundColour_1.INIT`; `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` ⟶ `INITO`.
- **Dateneingang `QI`** (`BOOL`, Vorgabe `TRUE`) wird an `.QI` aller Instanzen verteilt (`Q_BackgroundColour_1`, `Q_BackgroundColour_2`).
- **Datenausgang `QO`** (`BOOL`): UND aller Instanzen über `AND_QO` (`iec61131::booleanOperators::AND_BOOL_2`): `Q_BackgroundColour_1.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`.
- `E_MERGE_QO` (`iec61499::events::E_MERGE_3`) vereint `Q_BackgroundColour_2.INITO` und die `CNF`-Ereignisse beider Instanzen (`Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF`) und löst damit `AND_QO.REQ` aus.

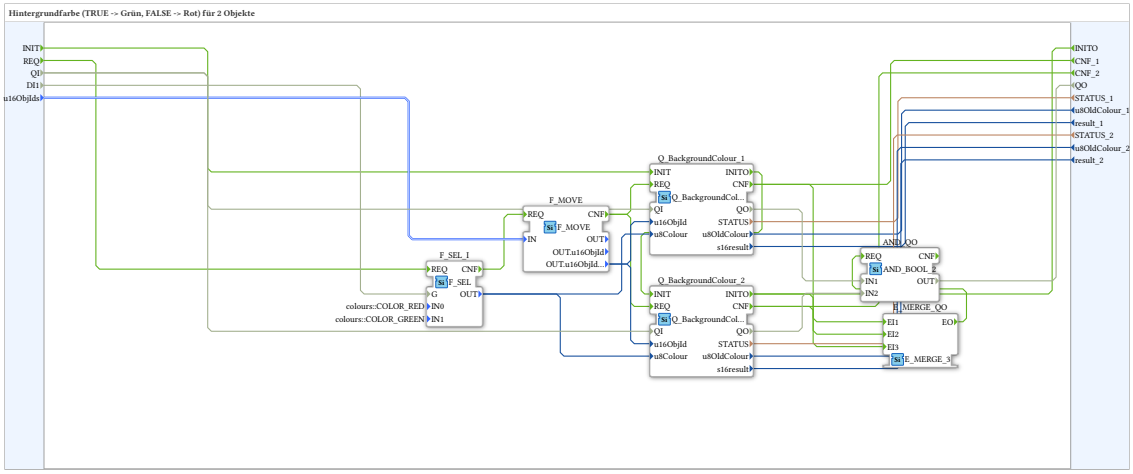
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Rot, 2 Objekte, BOOL-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenRedBackground2S



• – ○ • – ○ • – ○ •

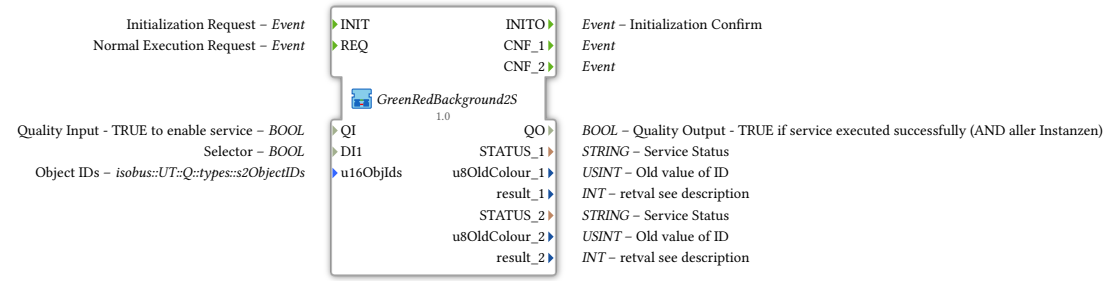
Einleitung

`GreenRedBackground2S` schaltet die VT-Hintergrundfarbe von 2 Objekten anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)

Schnittstelle (Quality-Muster): Ereignis-Eingang `INIT` und Ereignis-Ausgang `INITO` (je `EInit`), Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`) und Datenausgang `QO` (`BOOL`, UND aller Instanzen). Initialisierung: `INIT` ⟶ `Q_BackgroundColour_1.INIT`; `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` ⟶ `INITO`. Qualität: `QI` ⟶ `Q_BackgroundColour_1.QI`, `Q_BackgroundColour_2.QI`; `Q_BackgroundColour_1.QO`, `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN1` bis `AND_QO.IN2` (`AND_QO`: `iec61131::booleanOperators::AND_BOOL_2`); `AND_QO.OUT` ⟶ `QO`. Der Trigger `AND_QO.REQ` kommt über `E_MERGE_QO` (`E_MERGE_3`), der `Q_BackgroundColour_2.INITO` sowie die `CNF` von `Q_BackgroundColour_1` und `_2` zusammenführt; `E_MERGE_QO.EO` ⟶ `AND_QO.REQ`.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



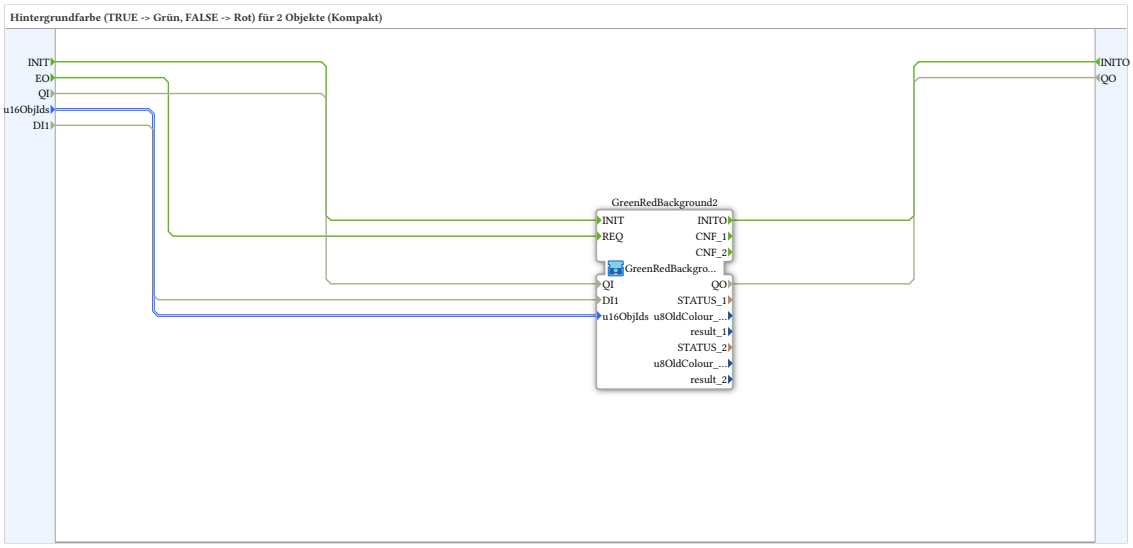
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Rot, 2 Objekte, BOOL-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenRedBackground2SC



• – ○ • – ○ • – ○ •

Einleitung

`GreenRedBackground2SC` ist ein **Kompakt-Wrapper** um `GreenRedBackground2S` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

Initialisierung und Status (INIT/INITO, QI/QO)

- **Ereignis-Eingang** `INIT` (`EInit`, Initialization Request) und **Ereignis-Ausgang** `INITO` (`EInit`, Initialization Confirm)
- **Daten-Eingang** `QI` (`BOOL`, Vorgabe `TRUE`, Quality Input) wird an `GreenRedBackground2.QI` weitergegeben
- **Daten-Ausgang** `QO` (`BOOL`, Quality Output): durchgereicht von `GreenRedBackground2.QO`
- **Initialisierungskette**: `INIT` ⟶ `GreenRedBackground2.INIT`, `GreenRedBackground2.INITO` ⟶ `INITO`

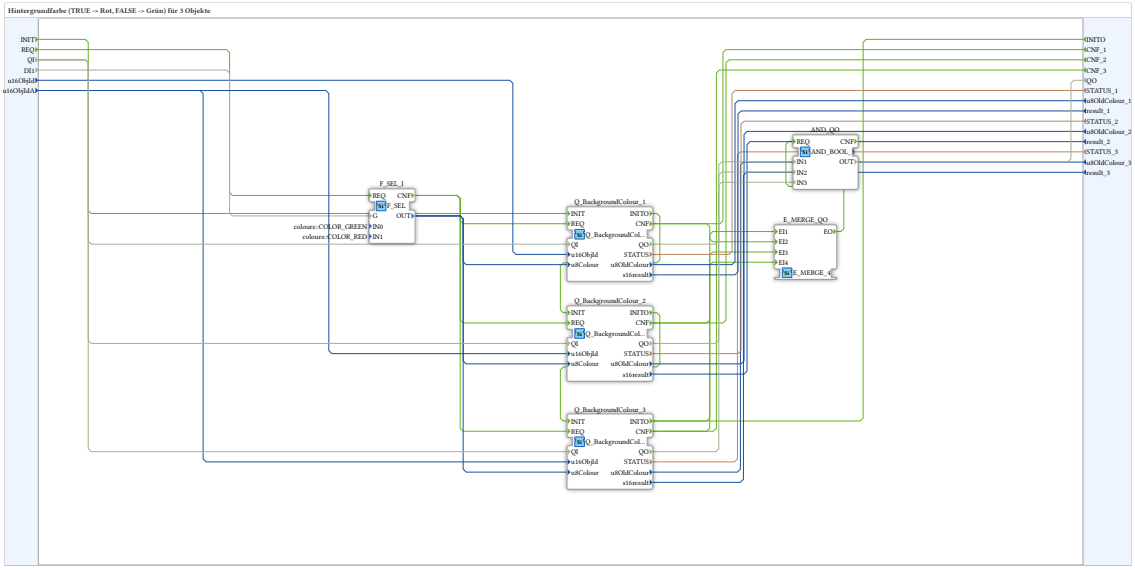
Zusammenfassung

Reine Weiterleitung an `GreenRedBackground2S` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenRedBackground3



• — ○ • — ○ • — ○ •

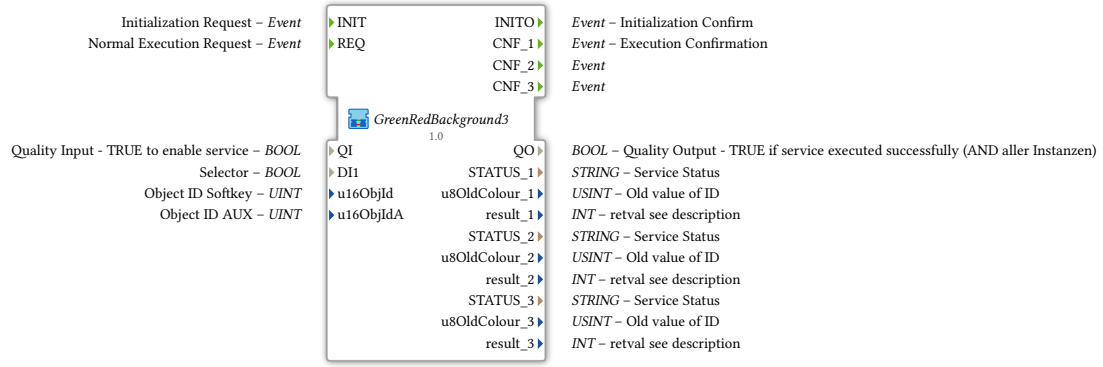
Einleitung

`GreenRedBackground3` schaltet die VT-Hintergrundfarbe von 3 Objekten anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Grün**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über die Eingänge `u16ObjId, u16ObjIdA` übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`u16ObjId`	`Q_BackgroundColour` (normales Objekt)
2	`u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
3	`u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Initialisierung und Qualität: Die SubApp besitzt `INIT`/`INITO` sowie `QI`/`QO`. `INIT` ⟶ `Q_BackgroundColour_1.INIT` ⟶ `Q_BackgroundColour_2.INIT` ⟶ `Q_BackgroundColour_3.INIT` (jeweils über `INITO`), `Q_BackgroundColour_3.INITO` ⟶ `INITO`. `QI` geht an alle drei Instanzen. `QO` ist das `AND` (`AND_QO`, `AND_BOOL_3`) der drei `QO`-Ausgänge; `AND_QO.REQ` wird über `E_MERGE_QO` (`E_MERGE_4`: `Q_BackgroundColour_3.INITO`, `Q_BackgroundColour_1.CNF`, `_2.CNF`, `_3.CNF`) ausgelöst.



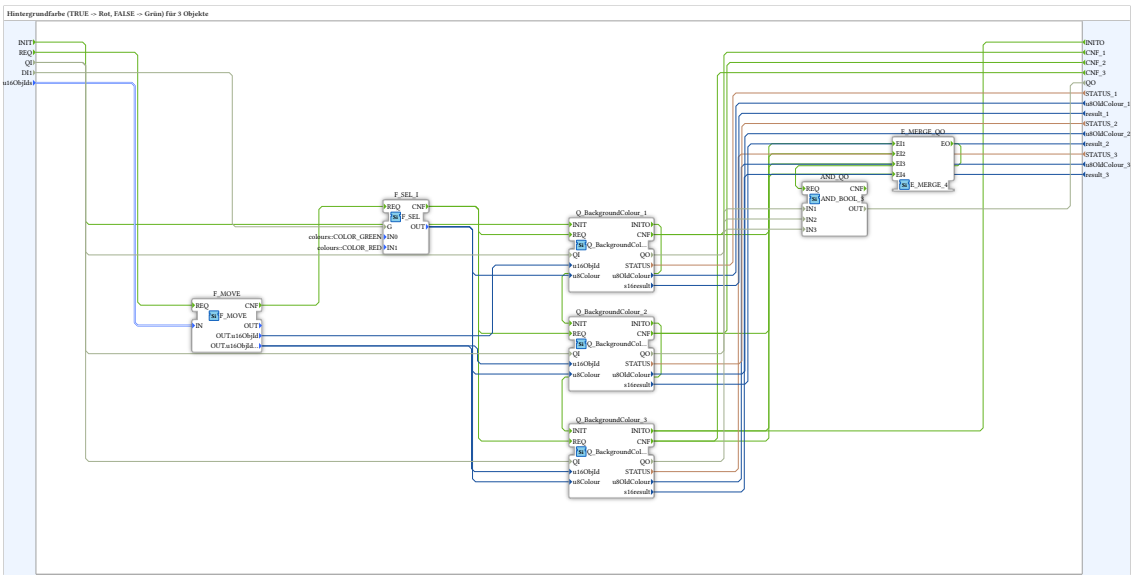
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Grün, 3 Objekte, BOOL-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenRedBackground3S



• – ○ • – ○ • – ○ •

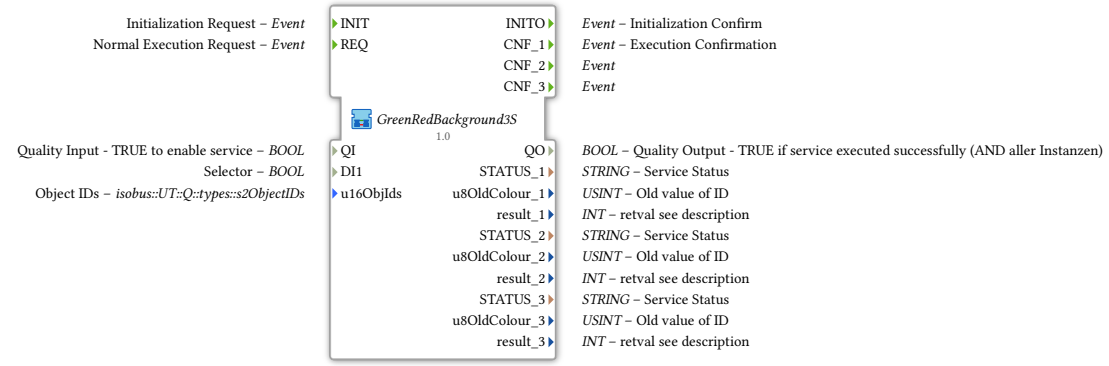
Einleitung

`GreenRedBackground3S` schaltet die VT-Hintergrundfarbe von 3 Objekten anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Grün**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`F_MOVE.OUT.u16ObjId`	`Q_BackgroundColour` (normales Objekt)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
3	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)

Initialisierung und Qualität: Zusätzlich zu den bisherigen Ereignissen besitzt der Baustein den Ereigniseingang `INIT`, den Ereignisausgang `INITO` sowie `QI` (BOOL, Eingang, Initialwert `TRUE`) und `QO` (BOOL, Ausgang, UND aller Instanzen). Init-Kette: `INIT` ⟶ `Q_BackgroundColour_1` ⟶ `Q_BackgroundColour_2` ⟶ `Q_BackgroundColour_3` ⟶ `INITO` (jeweils `INITO` ⟶ nächstes `INIT`). `QI` wird an alle `Q_BackgroundColour_n.QI` weitergereicht. `QO` bildet `AND_QO` (`iec61131::booleanOperators::AND_BOOL_3`): `Q_BackgroundColour_1.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN2`, `Q_BackgroundColour_3.QO` ⟶ `AND_QO.IN3`, `AND_QO.OUT` ⟶ `QO`. `E_MERGE_QO` (`E_MERGE_4`) führt `Q_BackgroundColour_3.INITO` und die `CNF`-Ausgänge `Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF`, `Q_BackgroundColour_3.CNF` zusammen; `E_MERGE_QO.EO` triggert `AND_QO.REQ`.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



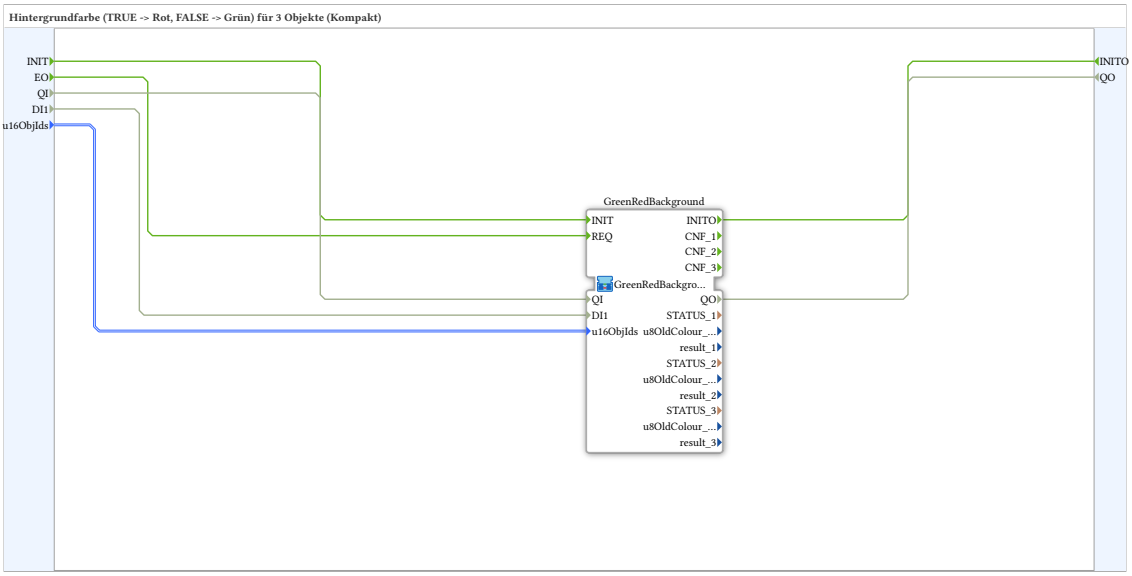
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Grün, 3 Objekte, BOOL-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenRedBackground3SC



• – ○ • – ○ • – ○ •

Einleitung

`GreenRedBackground3SC` ist ein **Kompakt-Wrapper** um `GreenRedBackground3S` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

Zusätzlich führt der Wrapper die Schnittstelle für Initialisierung und Qualität nach außen: Ereigniseingang `INIT`, Ereignisausgang `INITO`, Dateneingang `QI` (`BOOL`, Initialwert `TRUE`) und Datenausgang `QO` (`BOOL`). Sie sind 1:1 an die innere Instanz `GreenRedBackground` durchgeschleift: `INIT`

⟶ `GreenRedBackground.INIT`, `GreenRedBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenRedBackground.QI`, `GreenRedBackground.QO` ⟶ `QO`.

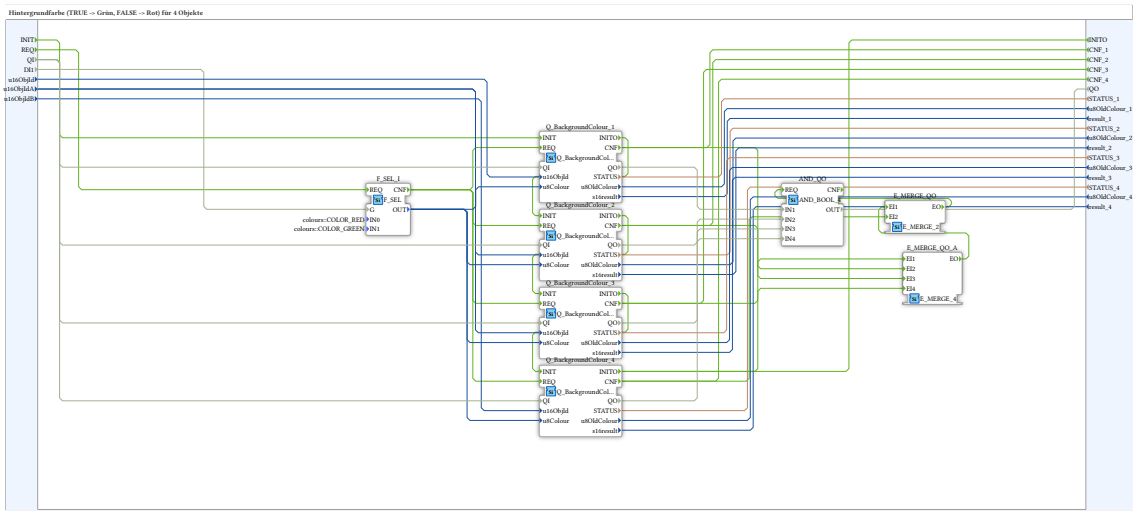
Zusammenfassung

Reine Weiterleitung an `GreenRedBackground3S` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenRedBackground4



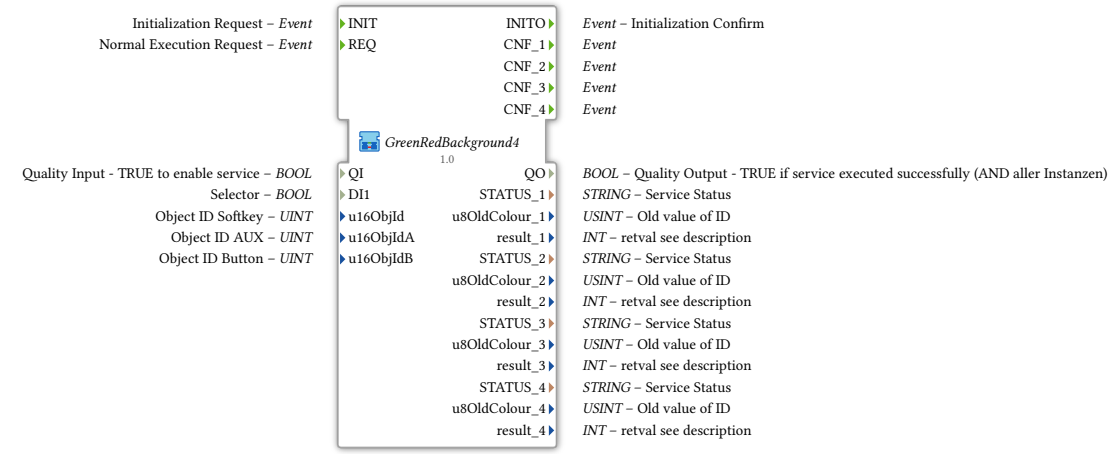
• – ○ • – ○ • – ○ •

Einleitung

`GreenRedBackground4` schaltet die VT-Hintergrundfarbe von 4 Objekten anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über die Eingänge `u16ObjId, u16ObjIdA, u16ObjIdB` übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`u16ObjId`	`Q_BackgroundColour` (normales Objekt)
2	`u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
3	`u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)
4	`u16ObjIdB`	`Q_BackgroundColour` (normales Objekt)

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Initialisierung und Qualität

- **Ereignis-Eingang `INIT`** (`EInit`) und **Ereignis-Ausgang `INITO`** (`EInit`): die Initialisierung wird durch die `Q_BackgroundColour`-Instanzen gekettet: `INIT` ⟶ `Q_BackgroundColour_1.INIT`; `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` ⟶ `Q_BackgroundColour_3.INIT`; `Q_BackgroundColour_3.INITO` ⟶ `Q_BackgroundColour_4.INIT`; `Q_BackgroundColour_4.INITO` ⟶ `INITO`.
- **Dateneingang `QI`** (`BOOL`, Vorgabe `TRUE`) wird an `.QI` aller Instanzen verteilt (`Q_BackgroundColour_1`, `Q_BackgroundColour_2`, `Q_BackgroundColour_3`, `Q_BackgroundColour_4`).
- **Datenausgang `QO`** (`BOOL`): UND aller Instanzen über `AND_QO` (`iec61131::booleanOperators::AND_BOOL_4`): `Q_BackgroundColour_1.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN2`, `Q_BackgroundColour_3.QO` ⟶ `AND_QO.IN3`, `Q_BackgroundColour_4.QO` ⟶ `AND_QO.IN4`; `AND_QO.OUT` ⟶ `QO`.
- `E_MERGE_QO_A` (`iec61499::events::E_MERGE_4`) vereint `Q_BackgroundColour_4.INITO` und die `CNF`-Ereignisse der Instanzen 1 bis 3; `E_MERGE_QO` (`iec61499::events::E_MERGE_2`) vereint dessen Ausgang `E_MERGE_QO_A.EO` mit `Q_BackgroundColour_4.CNF` und löst damit `AND_QO.REQ` aus.

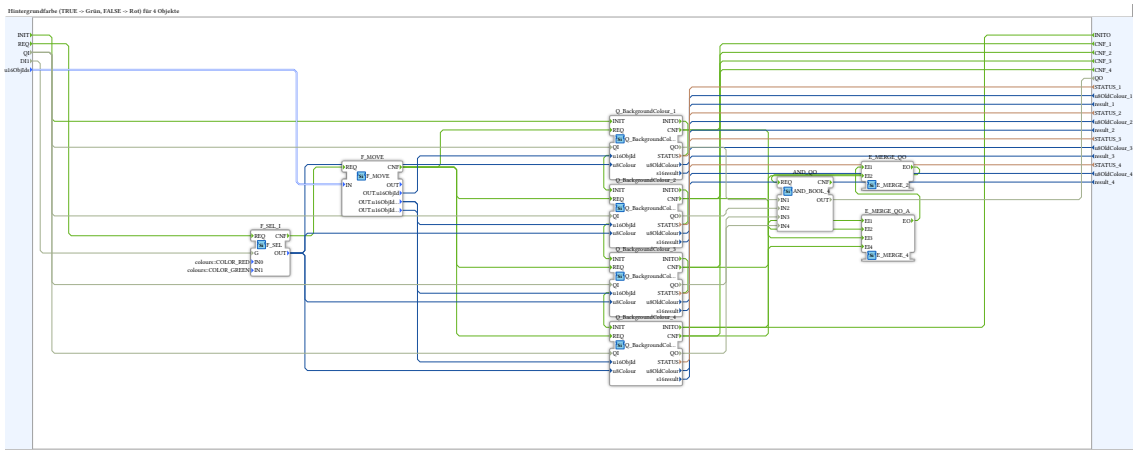
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Rot, 4 Objekte, BOOL-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenRedBackground4S



• – ○ • – ○ • – ○ •

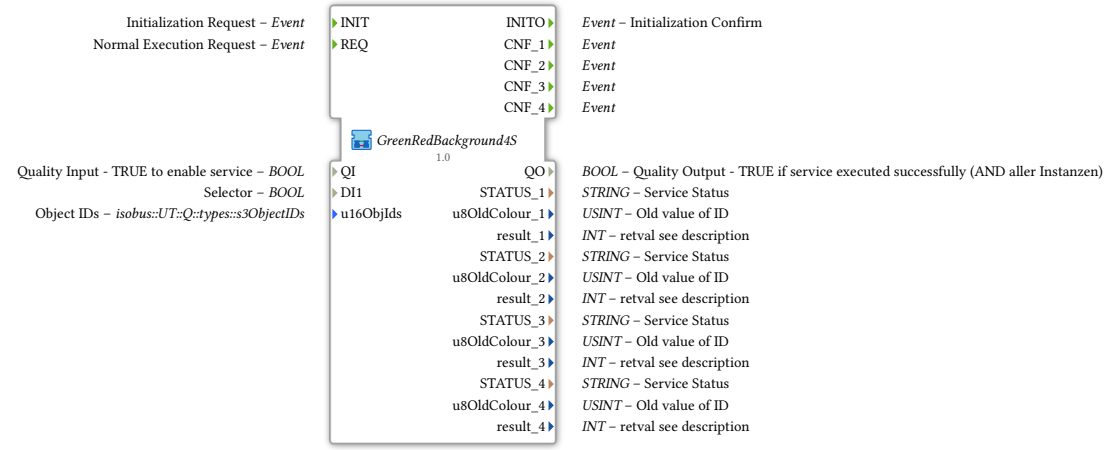
Einleitung

`GreenRedBackground4S` schaltet die VT-Hintergrundfarbe von 4 Objekten anhand eines booleschen Selector-Signals: `TRUE` → **Grün**, `FALSE` → **Rot**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`F_MOVE.OUT.u16ObjId`	`Q_BackgroundColour` (normales Objekt)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
3	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)
4	`F_MOVE.OUT.u16ObjIdB`	`Q_BackgroundColour` (normales Objekt)

Schnittstelle (Quality-Muster): Ereignis-Eingang `INIT` und Ereignis-Ausgang `INITO` (je `EInit`), Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`) und Datenausgang `QO` (`BOOL`, UND aller Instanzen). Initialisierung: `INIT` → `Q_BackgroundColour_1.INIT`; `Q_BackgroundColour_1.INITO` → `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` → `Q_BackgroundColour_3.INIT`; `Q_BackgroundColour_3.INITO` → `Q_BackgroundColour_4.INIT`; `Q_BackgroundColour_4.INITO` → `INITO`. Qualität: `QI` → `Q_BackgroundColour_1.QI`, `Q_BackgroundColour_2.QI`, `Q_BackgroundColour_3.QI`, `Q_BackgroundColour_4.QI`; `Q_BackgroundColour_1.QO`, `Q_BackgroundColour_2.QO`, `Q_BackgroundColour_3.QO`, `Q_BackgroundColour_4.QO` → `AND_QO.IN1` bis `AND_QO.IN4` (`AND_QO`: `iec61131::booleanOperators::AND_BOOL_4`); `AND_QO.OUT` → `QO`. Der Trigger `AND_QO.REQ` kommt über zwei Merge-Bausteine: `E_MERGE_QO_A` (`E_MERGE_4`) führt `Q_BackgroundColour_4.INITO` und die `CNF` von `Q_BackgroundColour_1` bis `_3` zusammen, `E_MERGE_QO` (`E_MERGE_2`) dessen Ausgang und `Q_BackgroundColour_4.CNF`; `E_MERGE_QO.EO` → `AND_QO.REQ`.

Allgemeines Muster (Selector → `AX_SEL`/`F_SEL` → `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



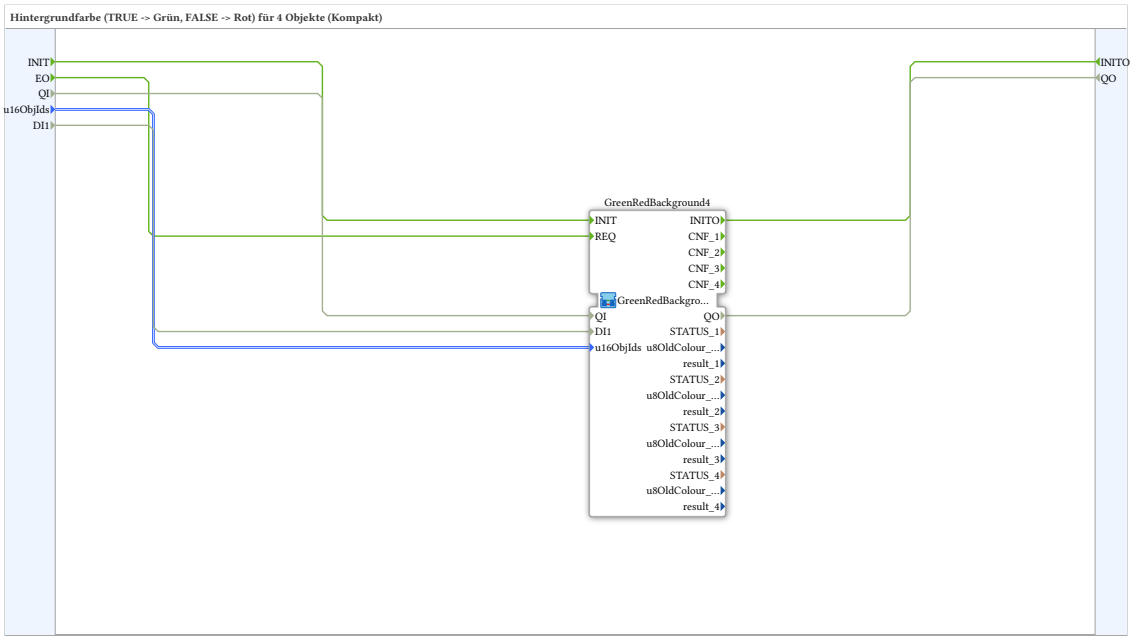
Zusammenfassung

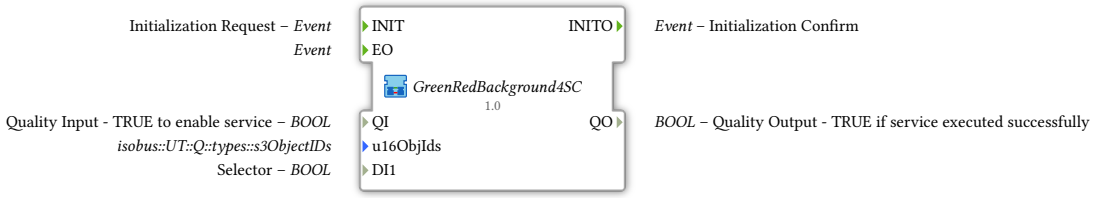
Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Rot, 4 Objekte, BOOL-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenRedBackground4SC





• – ○ • – ○ • – ○ •

Einleitung

`GreenRedBackground4SC` ist ein **Kompakt-Wrapper** um `GreenRedBackground4S` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

Initialisierung und Status (INIT/INITO, QI/QO)

- **Ereignis-Eingang** `INIT` (`EInit`, Initialization Request) und **Ereignis-Ausgang** `INITO` (`EInit`, Initialization Confirm)
- **Daten-Eingang** `QI` (`BOOL`, Vorgabe `TRUE`, Quality Input) wird an `GreenRedBackground4.QI` weitergegeben
- **Daten-Ausgang** `QO` (`BOOL`, Quality Output): durchgereicht von `GreenRedBackground4.QO`
- **Initialisierungskette**: `INIT` ⟶ `GreenRedBackground4.INIT`, `GreenRedBackground4.INITO` ⟶ `INITO`

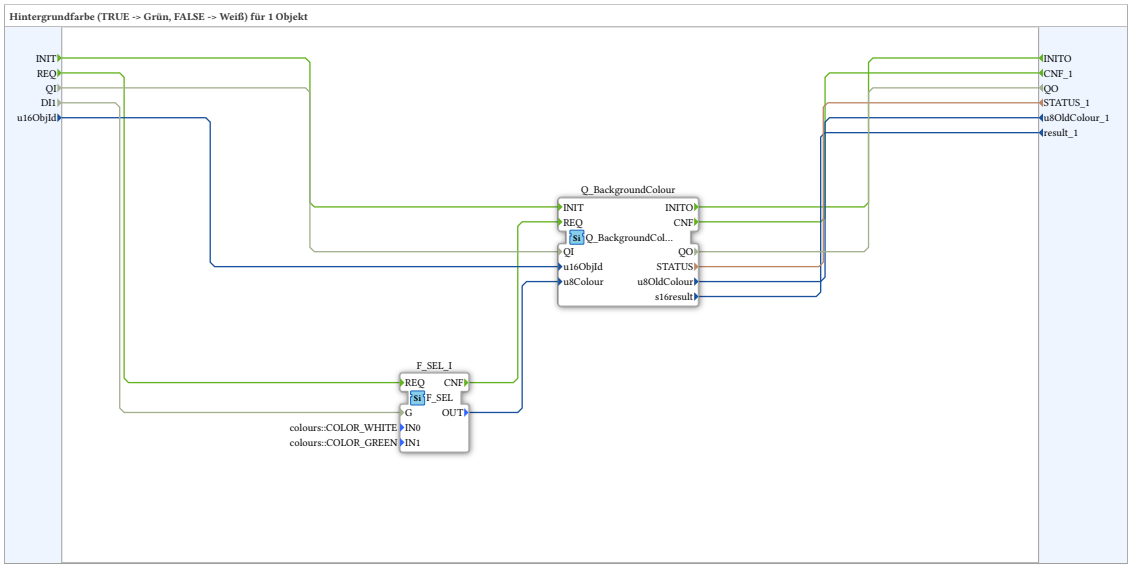
Zusammenfassung

Reine Weiterleitung an `GreenRedBackground4S` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground1



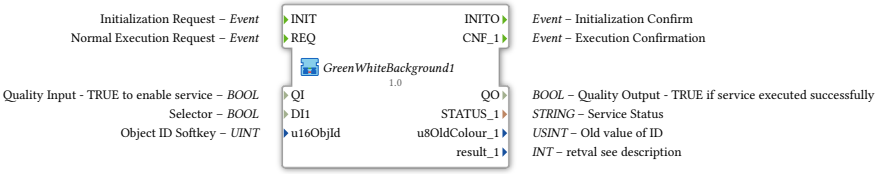
• — ○ • — ○ • — ○ •

Einleitung

`GreenWhiteBackground1` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den Eingang `u16ObjId` übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Quality-Anschlüsse: Der Baustein besitzt zusätzlich das Ereignis `INIT` (Eingang) und `INITO` (Ausgang) sowie den Dateneingang `QI` (BOOL, Standard `TRUE`, „Quality Input – TRUE, um den Dienst freizugeben“) und den Datenausgang `QO` (BOOL, „Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde“). `INIT` wird an `Q_BackgroundColour.INIT` weitergereicht, `Q_BackgroundColour.INITO` an `INITO`; `QI` ist mit `Q_BackgroundColour.QI` und `Q_BackgroundColour.QO` mit `QO` verbunden.



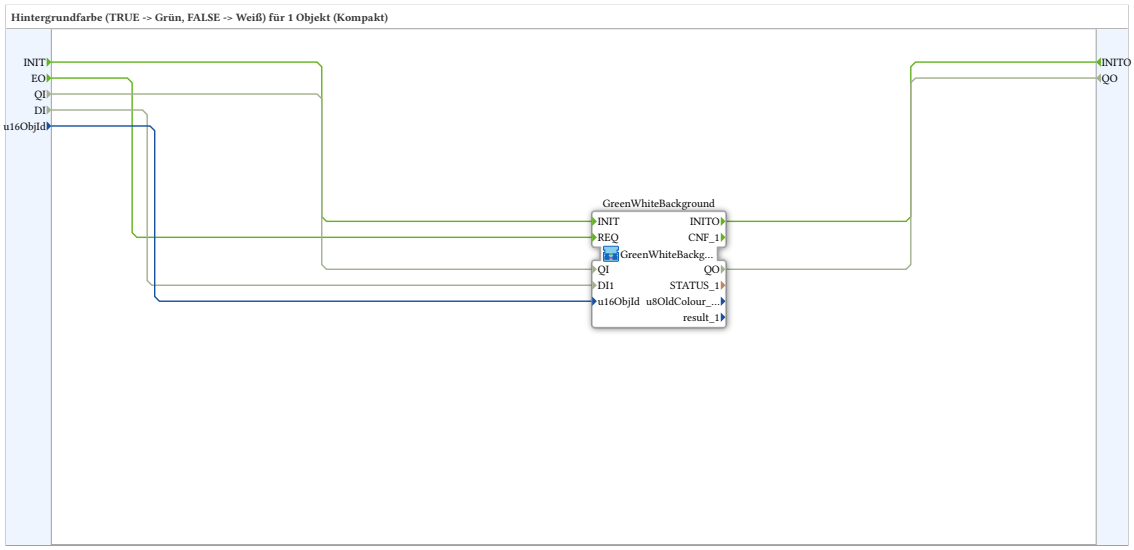
Zusammenfassung

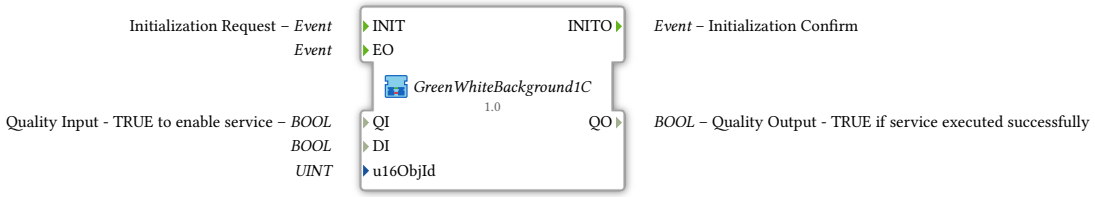
Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Weiß, 1 Objekt, BOOL-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground1C





• – ○ • – ○ • – ○ •

Einleitung

`GreenWhiteBackground1C` ist ein **Kompakt-Wrapper** um `GreenWhiteBackground1` — reicht `u16ObjId` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen. Zusätzlich werden `INIT` ⟶ `GreenWhiteBackground.INIT`, `GreenWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground.QI` (Initialwert `TRUE`) und `GreenWhiteBackground.QO` ⟶ `QO` durchgereicht.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

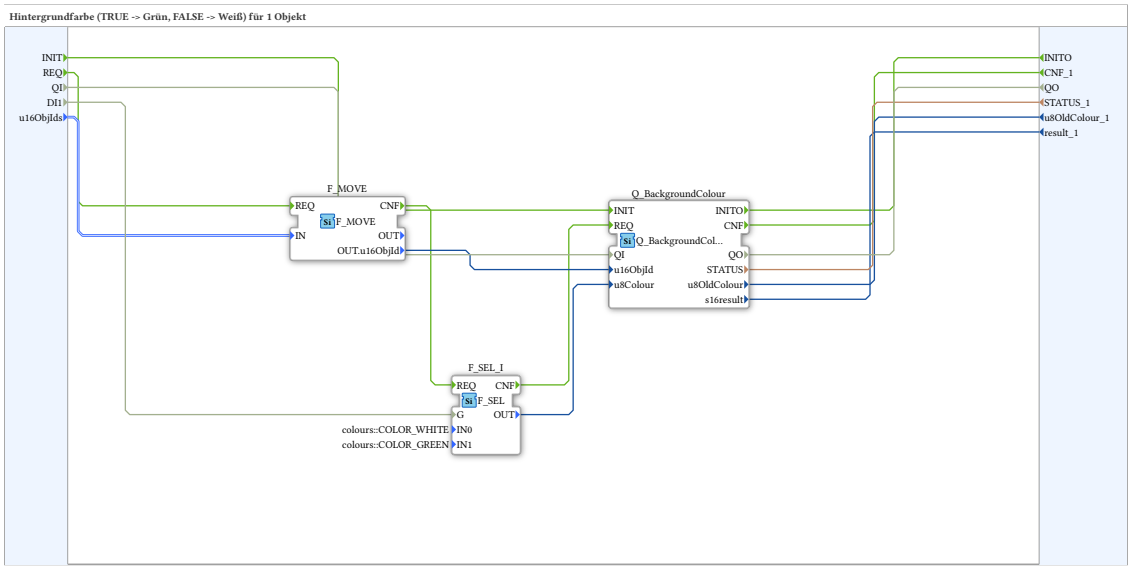
Zusammenfassung

Reine Weiterleitung an `GreenWhiteBackground1` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground1S



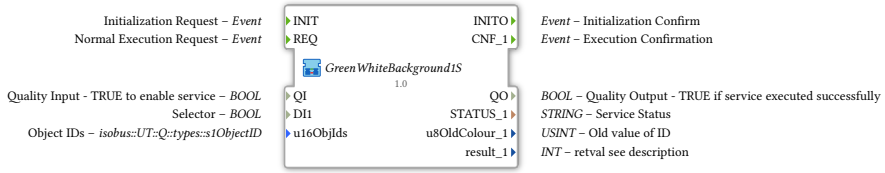
• – ○ • – ○ • – ○ •

Einleitung

`GreenWhiteBackground1S` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Quality-Anschlüsse: Der Baustein besitzt zusätzlich das Ereignis `INIT` (Eingang) und `INITO` (Ausgang) sowie den Dateneingang `QI` (BOOL, Standard `TRUE`, „Quality Input – TRUE, um den Dienst freizugeben“) und den Datenausgang `QO` (BOOL, „Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde“). `INIT` wird an `Q_BackgroundColour.INIT` weitergereicht, `Q_BackgroundColour.INITO` an `INITO`; `QI` ist mit `Q_BackgroundColour.QI` und `Q_BackgroundColour.QO` mit `QO` verbunden.



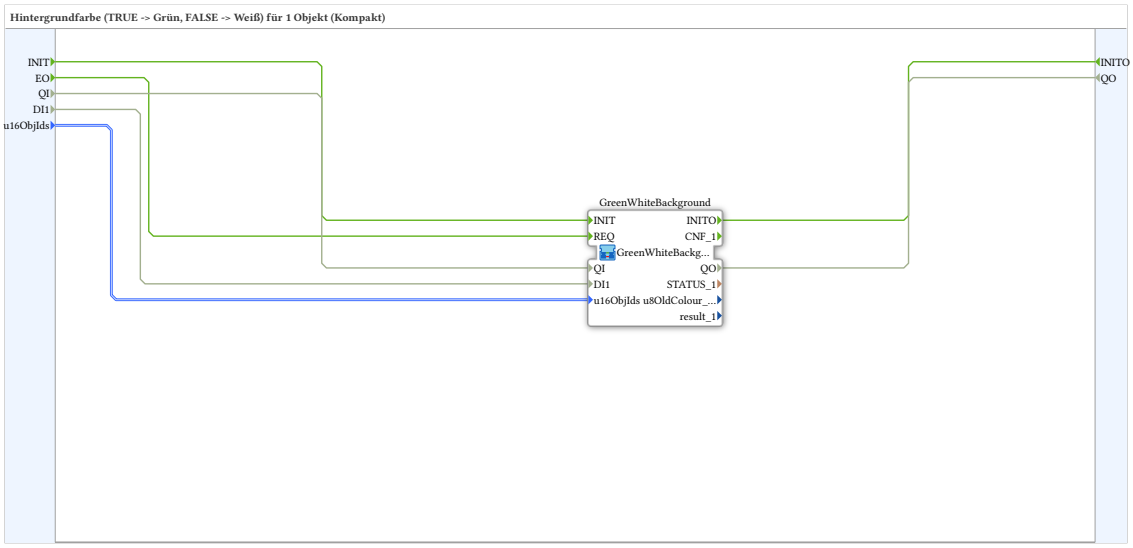
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Weiß, 1 Objekt, BOOL-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground1SC



• – ○ • – ○ • – ○ •

Einleitung

`GreenWhiteBackground1SC` ist ein **Kompakt-Wrapper** um `GreenWhiteBackground1S` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Zusätzlich werden Initialisierung und Qualität durchgereicht: `INIT` ⟶ `GreenWhiteBackground.INIT`, `GreenWhiteBackground.INITO` ⟶ `INITO`, `QI` (Initialwert `TRUE`) ⟶ `GreenWhiteBackground.QI` und `GreenWhiteBackground.QO` ⟶ `QO`.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

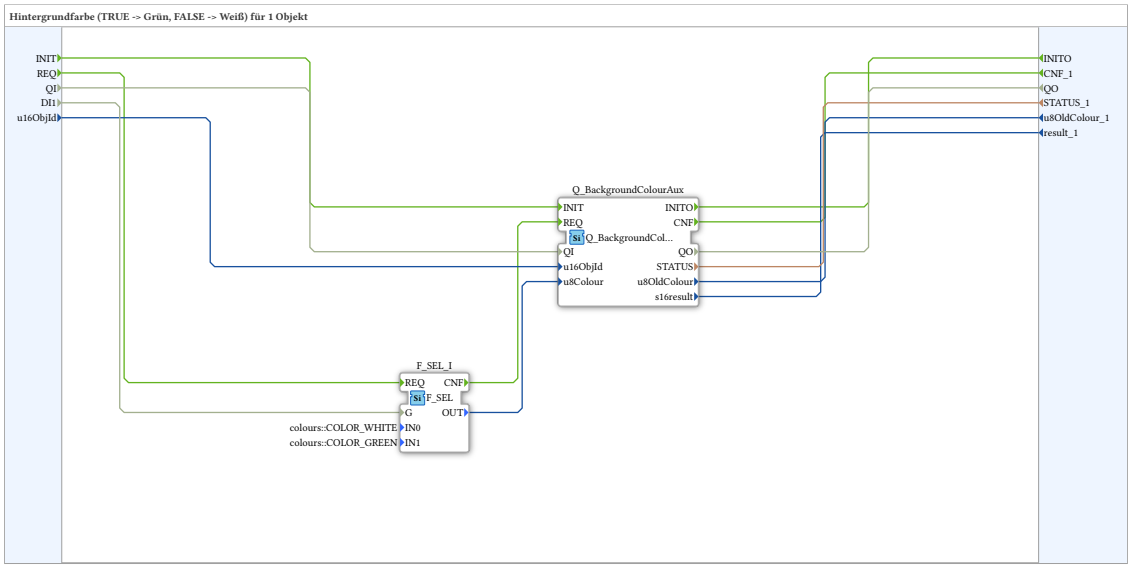
Zusammenfassung

Reine Weiterleitung an `GreenWhiteBackground1S` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground1_aux



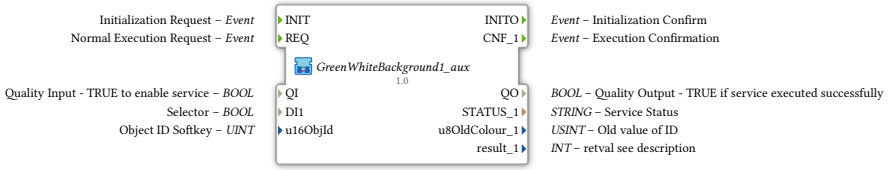
• — ○ • — ○ • — ○ •

Einleitung

`GreenWhiteBackground1_aux` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den Eingang `u16ObjId` übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Quality-Anschlüsse: Der Baustein besitzt zusätzlich das Ereignis `INIT` (Eingang) und `INITO` (Ausgang) sowie den Dateneingang `QI` (BOOL, Standard `TRUE`, „Quality Input – TRUE, um den Dienst freizugeben“) und den Datenausgang `QO` (BOOL, „Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde“). `INIT` wird an `Q_BackgroundColourAux.INIT` weitergereicht, `Q_BackgroundColourAux.INITO` an `INITO`; `QI` ist mit `Q_BackgroundColourAux.QI` und `Q_BackgroundColourAux.QO` mit `QO` verbunden.



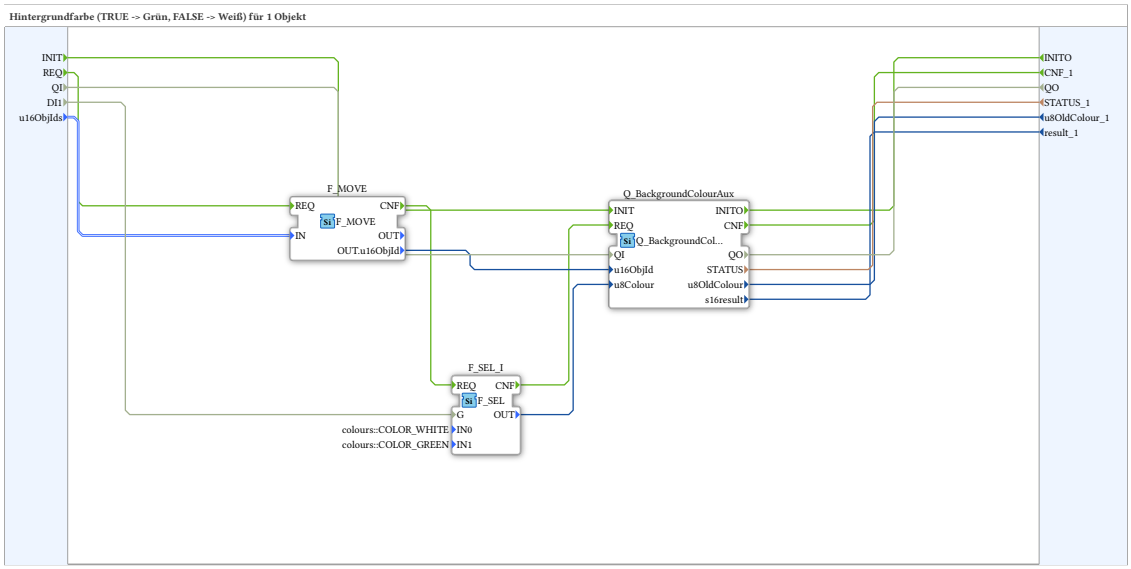
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Weiß, 1 Objekt, BOOL-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground1_auxS



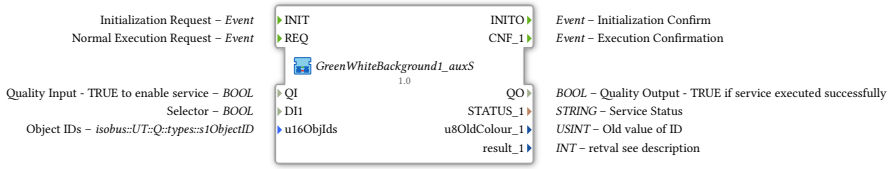
• – ○ • – ○ • – ○ •

Einleitung

`GreenWhiteBackground1_auxS` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Quality-Anschlüsse: Der Baustein besitzt zusätzlich das Ereignis `INIT` (Eingang) und `INITO` (Ausgang) sowie den Dateneingang `QI` (BOOL, Standard `TRUE`, „Quality Input – TRUE, um den Dienst freizugeben“) und den Datenausgang `QO` (BOOL, „Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde“). `INIT` wird an `Q_BackgroundColourAux.INIT` weitergereicht, `Q_BackgroundColourAux.INITO` an `INITO`; `QI` ist mit `Q_BackgroundColourAux.QI` und `Q_BackgroundColourAux.QO` mit `QO` verbunden.



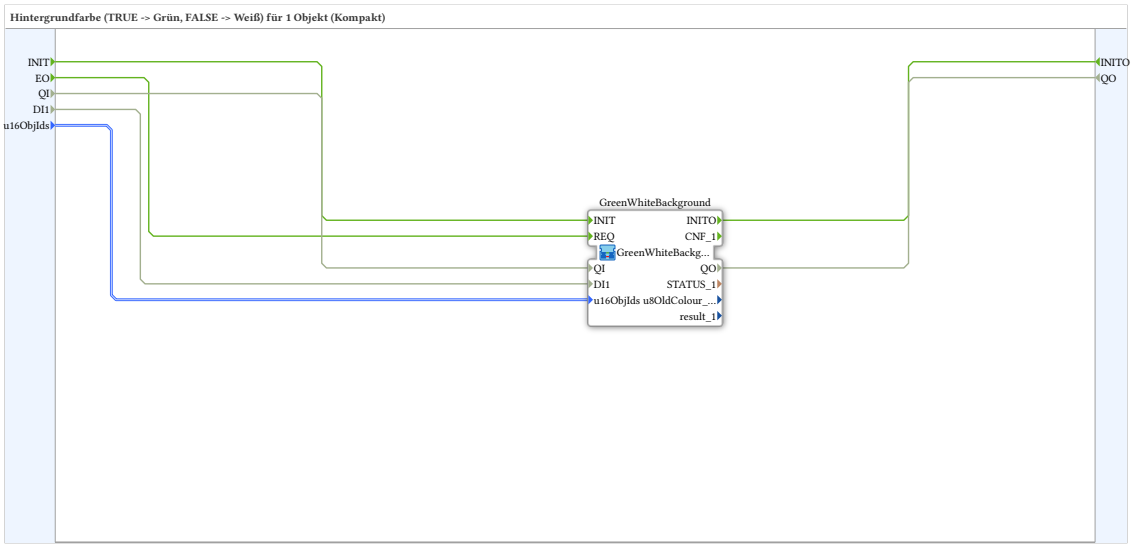
Zusammenfassung

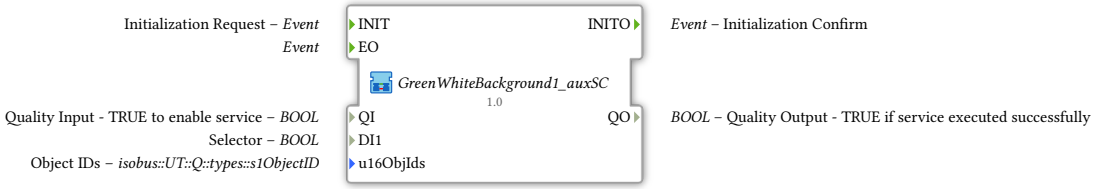
Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Weiß, 1 Objekt, BOOL-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground1_auxSC





• — ○ • — ○ • — ○ •

Einleitung

`GreenWhiteBackground1_auxSC` ist ein **Kompakt-Wrapper** um `GreenWhiteBackground1_auxS` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

Zusätzlich führt der Wrapper die Schnittstelle für Initialisierung und Qualität nach außen: Ereigniseingang `INIT`, Ereignisausgang `INITO`, Dateneingang `QI` (`BOOL`, Initialwert `TRUE`) und Datenausgang `QO` (`BOOL`). Sie sind 1:1 an die innere Instanz `GreenWhiteBackground` durchgeschleift: `INIT` ⟶ `GreenWhiteBackground.INIT`, `GreenWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground.QI`, `GreenWhiteBackground.QO` ⟶ `QO`.

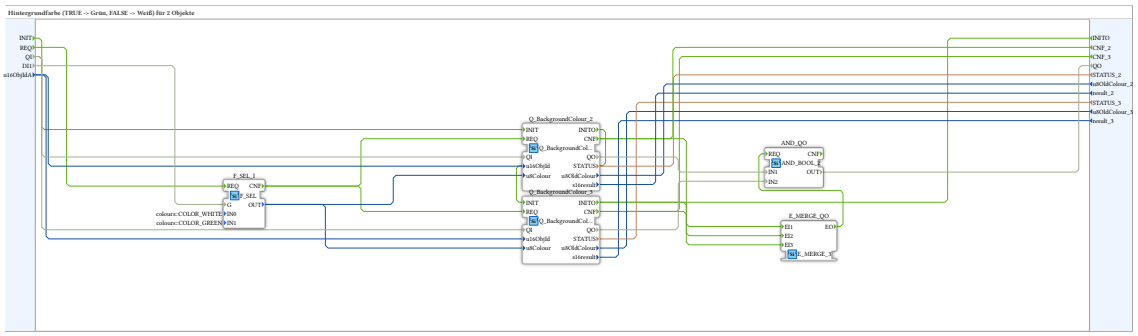
Zusammenfassung

Reine Weiterleitung an `GreenWhiteBackground1_auxS` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground2



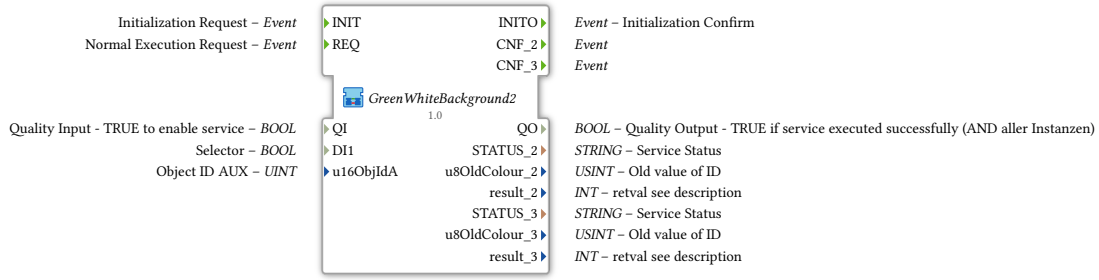
• — ○ • — ○ • — ○ •

Einleitung

`GreenWhiteBackground2` schaltet die VT-Hintergrundfarbe von 2 Objekten anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über die Eingänge `u16ObjIdA` übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
2	`u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Initialisierung und Qualität

- **Ereignis-Eingang `INIT`** (`EInit`) und **Ereignis-Ausgang `INITO`** (`EInit`): die Initialisierung wird durch die `Q_BackgroundColour`-Instanzen gekettet: `INIT` ⟶ `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` ⟶ `Q_BackgroundColour_3.INIT`; `Q_BackgroundColour_3.INITO` ⟶ `INITO`.
- **Dateneingang `QI`** (`BOOL`, Vorgabe `TRUE`) wird an `.QI` aller Instanzen verteilt (`Q_BackgroundColour_2`, `Q_BackgroundColour_3`).
- **Datenausgang `QO`** (`BOOL`): UND aller Instanzen über `AND_QO` (`iec61131::booleanOperators::AND_BOOL_2`): `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_3.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`.
- `E_MERGE_QO` (`iec61499::events::E_MERGE_3`) vereint `Q_BackgroundColour_3.INITO` und die `CNF`-Ereignisse beider Instanzen (`Q_BackgroundColour_2.CNF`, `Q_BackgroundColour_3.CNF`) und löst damit `AND_QO.REQ` aus.

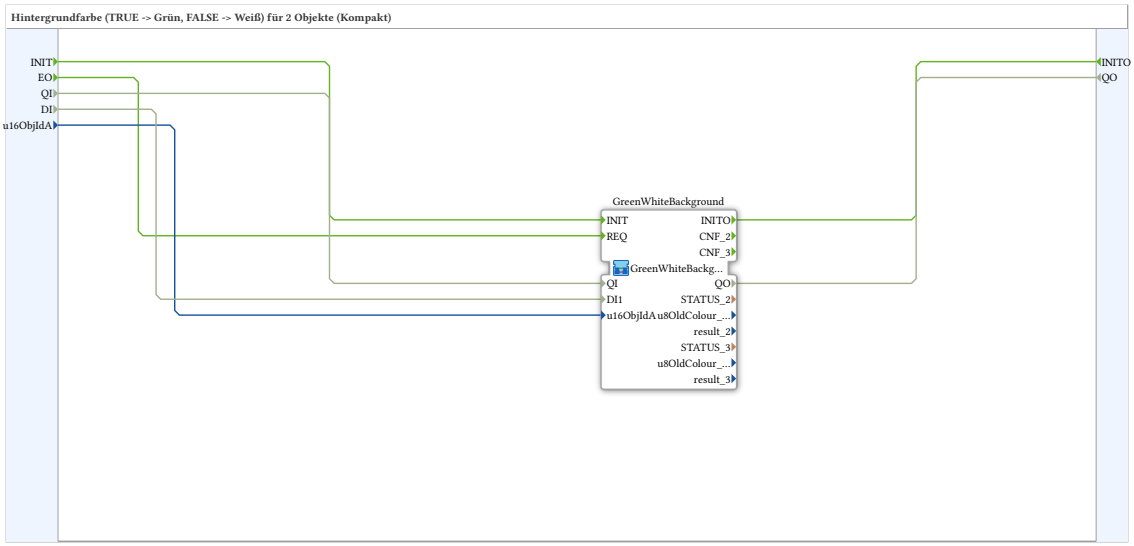
Zusammenfassung

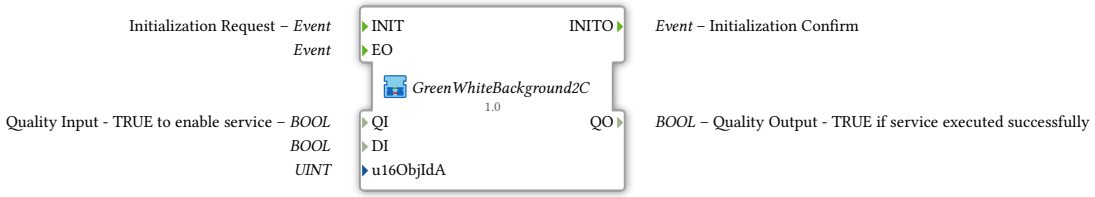
Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Weiß, 2 Objekte, BOOL-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground2C





• – ○ • – ○ • – ○ •

Einleitung

`GreenWhiteBackground2C` ist ein **Kompakt-Wrapper** um `GreenWhiteBackground2` — reicht `u16ObjId` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Schnittstelle (Quality-Muster): Der Wrapper besitzt zusätzlich den Ereignis-Eingang `INIT` und den Ereignis-Ausgang `INITO` (je `EInit`) sowie den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`) und den Datenausgang `QO` (`BOOL`). Sie sind direkt an die innere Instanz `GreenWhiteBackground` durchgeschleift: `INIT` ⟶ `GreenWhiteBackground.INIT`, `GreenWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground.QI`, `GreenWhiteBackground.QO` ⟶ `QO`.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

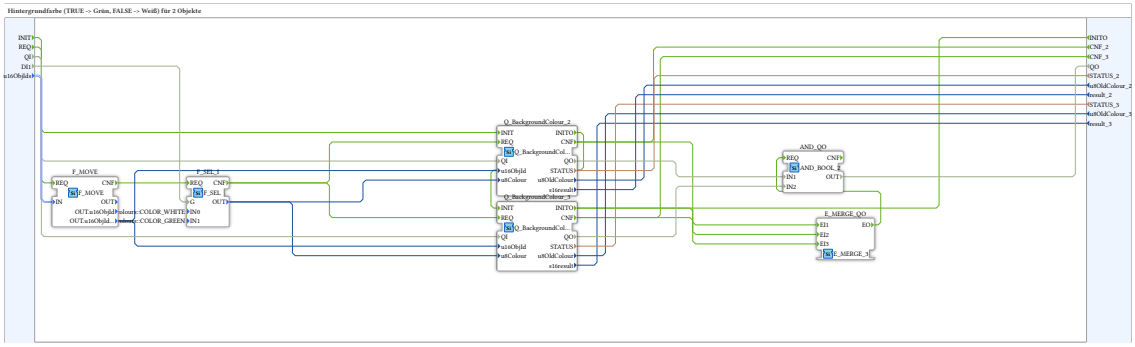
Zusammenfassung

Reine Weiterleitung an `GreenWhiteBackground2` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground2S



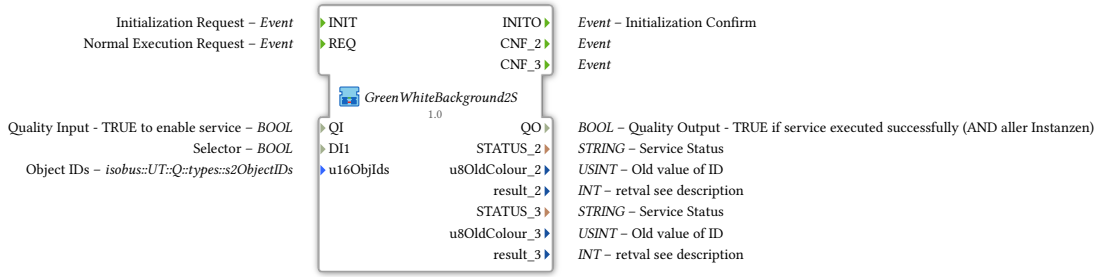
• – ○ • – ○ • – ○ •

Einleitung

`GreenWhiteBackground2S` schaltet die VT-Hintergrundfarbe von 2 Objekten anhand eines booleschen Selector-Signals: `TRUE` → **Grün**, `FALSE` → **Weiß**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)

Allgemeines Muster (Selector → `AX_SEL`/`F_SEL` → `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Initialisierung und Status (INIT/INITO, QI/QO)

- **Ereignis-Eingang** `INIT` (`EInit`, Initialization Request) und **Ereignis-Ausgang** `INITO` (`EInit`, Initialization Confirm)
- **Daten-Eingang** `QI` (`BOOL`, Vorgabe `TRUE`, Quality Input) wird an `Q_BackgroundColour_2.QI` und `Q_BackgroundColour_3.QI` weitergegeben
- **Daten-Ausgang** `QO` (`BOOL`, Quality Output): UND (`AND_QO`, Typ `iec61131::booleanOperators::AND_BOOL_2`) der `QO`-Ausgänge von `Q_BackgroundColour_2.QO` und `Q_BackgroundColour_3.QO`
- **Initialisierungskette**: `INIT` → `Q_BackgroundColour_2.INIT`, `Q_BackgroundColour_2.INITO` → `Q_BackgroundColour_3.INIT`, `Q_BackgroundColour_3.INITO` → `INITO`
- `E_MERGE_QO` (Typ `iec61499::events::E_MERGE_3`) führt `Q_BackgroundColour_3.INITO`, `Q_BackgroundColour_2.CNF` und `Q_BackgroundColour_3.CNF` zusammen und löst `AND_QO.REQ` aus

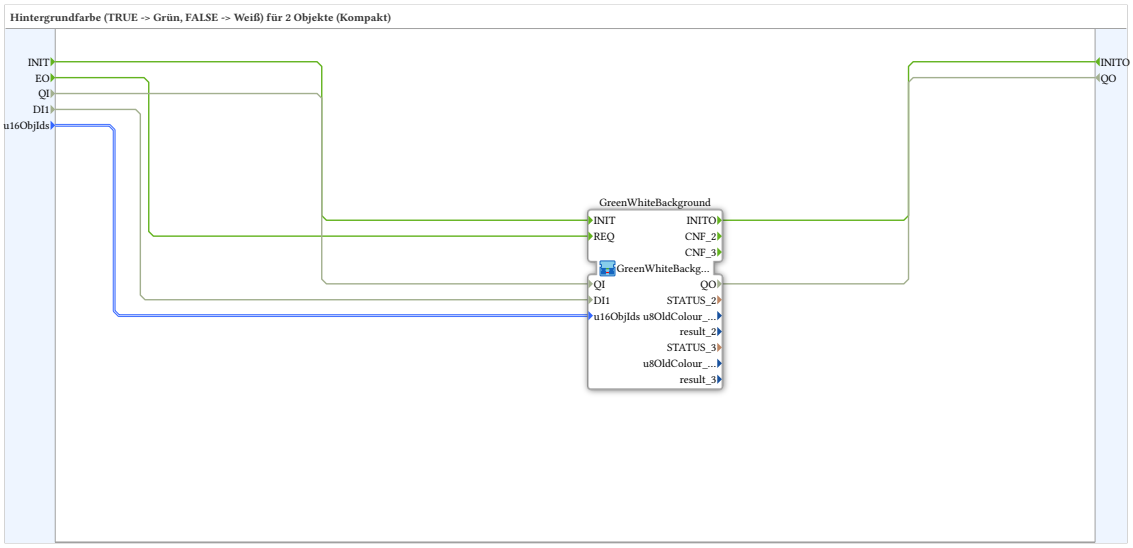
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Weiß, 2 Objekte, BOOL-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground2SC



• – ○ • – ○ • – ○ •

Einleitung

`GreenWhiteBackground2SC` ist ein **Kompakt-Wrapper** um `GreenWhiteBackground2S` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen. Zusätzlich werden `INIT` ⟶ `GreenWhiteBackground.INIT`, `GreenWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground.QI` (Initialwert `TRUE`) und `GreenWhiteBackground.QO` ⟶ `QO` durchgereicht.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

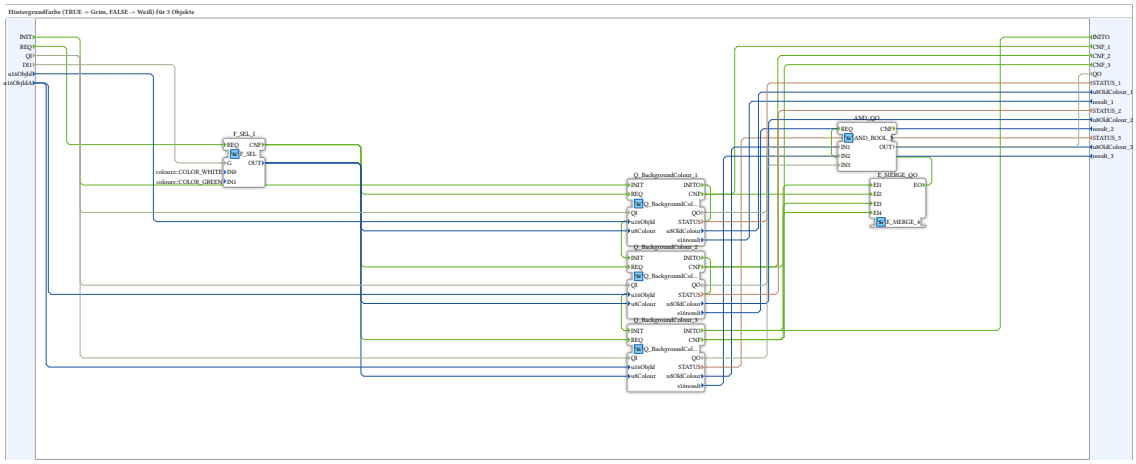
Zusammenfassung

Reine Weiterleitung an `GreenWhiteBackground2S` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground3



• – ○ • – ○ • – ○ •

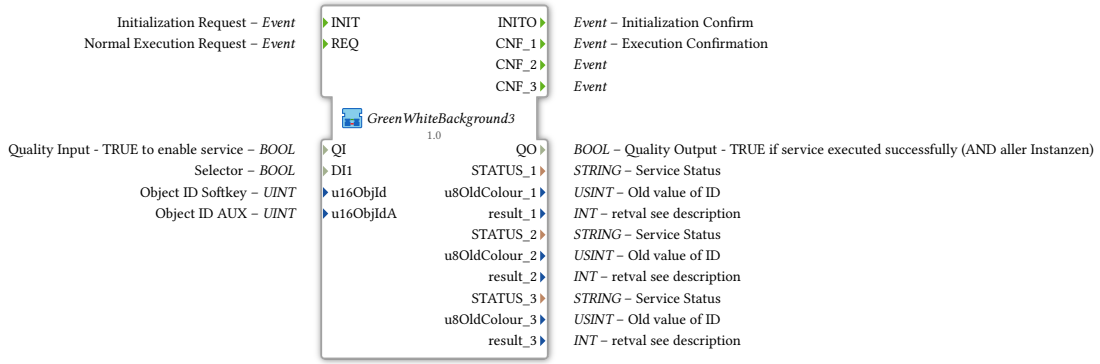
Einleitung

`GreenWhiteBackground3` schaltet die VT-Hintergrundfarbe von 3 Objekten anhand eines booleschen Selector-Signals: `TRUE` → **Grün**, `FALSE` → **Weiß**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über die Eingänge `u16ObjId, u16ObjIdA` übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`u16ObjId`	`Q_BackgroundColour` (normales Objekt)
2	`u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
3	`u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)

Initialisierung und Qualität: Zusätzlich zu den bisherigen Ereignissen besitzt der Baustein den Ereigniseingang `INIT`, den Ereignisausgang `INITO` sowie `QI` (BOOL, Eingang, Initialwert `TRUE`) und `QO` (BOOL, Ausgang, UND aller Instanzen). Init-Kette: `INIT` → `Q_BackgroundColour_1` → `Q_BackgroundColour_2` → `Q_BackgroundColour_3` → `INITO` (jeweils `INITO` → nächstes `INIT`). `QI` wird an alle `Q_BackgroundColour_n.QI` weitergereicht. `QO` bildet `AND_QO` (`iec61131::booleanOperators::AND_BOOL_3`): `Q_BackgroundColour_1.QO` → `AND_QO.IN1`, `Q_BackgroundColour_2.QO` → `AND_QO.IN2`, `Q_BackgroundColour_3.QO` → `AND_QO.IN3`, `AND_QO.OUT` → `QO`. `E_MERGE_QO` (`E_MERGE_4`) führt `Q_BackgroundColour_3.INITO` und die `CNF`-Ausgänge `Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF`, `Q_BackgroundColour_3.CNF` zusammen; `E_MERGE_QO.EO` triggert `AND_QO.REQ`.

Allgemeines Muster (Selector → `AX_SEL/F_SEL` → `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



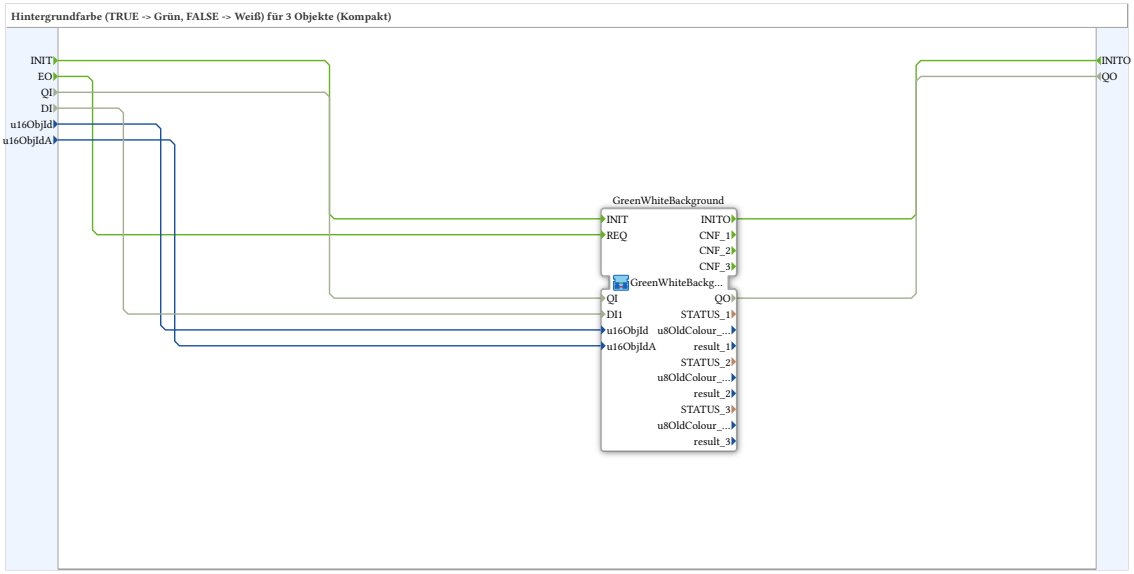
Zusammenfassung

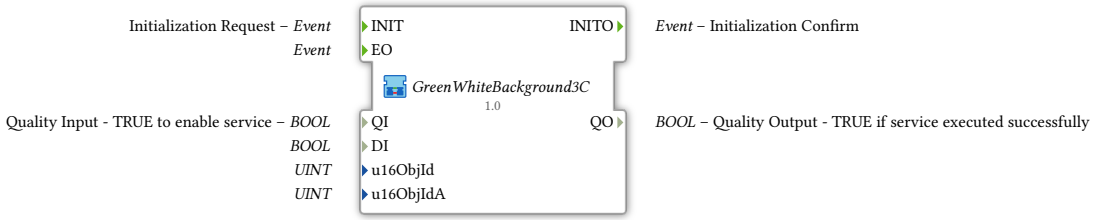
Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Weiß, 3 Objekte, BOOL-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground3C





• — ○ • — ○ • — ○ •

Einleitung

`GreenWhiteBackground3C` ist ein **Kompakt-Wrapper** um `GreenWhiteBackground3` — reicht `u16ObjId` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

Zusätzlich führt der Wrapper die Schnittstelle für Initialisierung und Qualität nach außen: Ereigniseingang `INIT`, Ereignisausgang `INITO`, Dateneingang `QI` (`BOOL`, Initialwert `TRUE`) und Datenausgang `QO` (`BOOL`). Sie sind 1:1 an die innere Instanz `GreenWhiteBackground` durchgeschleift: `INIT` ⟶ `GreenWhiteBackground.INIT`, `GreenWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground.QI`, `GreenWhiteBackground.QO` ⟶ `QO`.

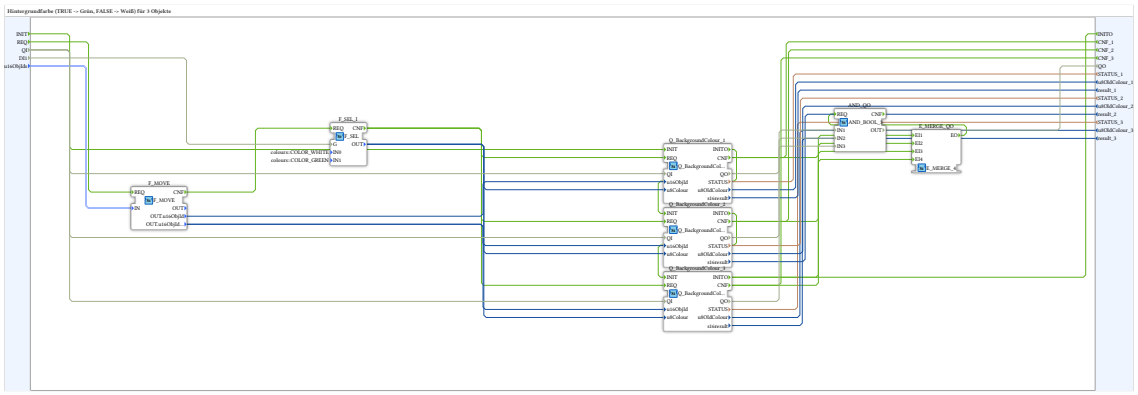
Zusammenfassung

Reine Weiterleitung an `GreenWhiteBackground3` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground3S



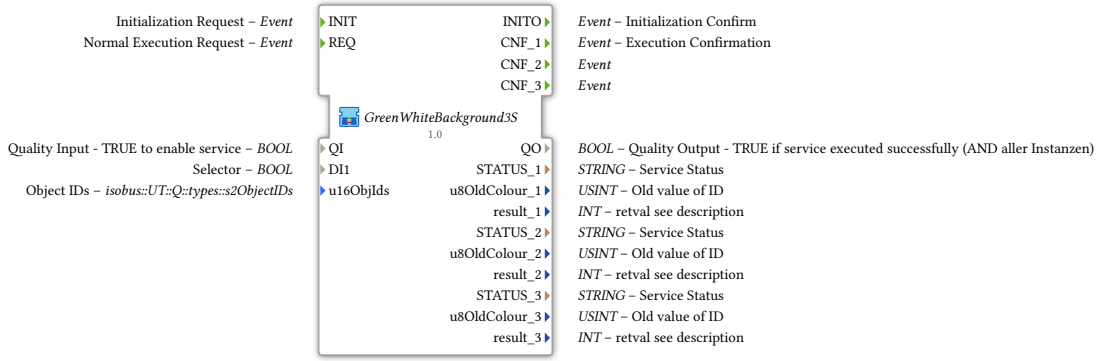
• – ○ • – ○ • – ○ •

Einleitung

`GreenWhiteBackground3S` schaltet die VT-Hintergrundfarbe von 3 Objekten anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s2ObjectIDs` (`u16ObjIds`) übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`F_MOVE.OUT.u16ObjId`	`Q_BackgroundColour` (normales Objekt)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
3	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Initialisierung und Qualität

- **Ereignis-Eingang `INIT`** (`EInit`) und **Ereignis-Ausgang `INITO`** (`EInit`): die Initialisierung wird durch die `Q_BackgroundColour`-Instanzen gekettet: `INIT` ⟶ `Q_BackgroundColour_1.INIT`; `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` ⟶ `Q_BackgroundColour_3.INIT`; `Q_BackgroundColour_3.INITO` ⟶ `INITO`.
- **Dateneingang `QI`** (`BOOL`, Vorgabe `TRUE`) wird an `.QI` aller Instanzen verteilt (`Q_BackgroundColour_1`, `Q_BackgroundColour_2`, `Q_BackgroundColour_3`).
- **Datenausgang `QO`** (`BOOL`): UND aller Instanzen über `AND_QO` (`iec61131::booleanOperators::AND_BOOL_3`): `Q_BackgroundColour_1.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN2`, `Q_BackgroundColour_3.QO` ⟶ `AND_QO.IN3`; `AND_QO.OUT` ⟶ `QO`.
- `E_MERGE_QO` (`iec61499::events::E_MERGE_4`) vereint `Q_BackgroundColour_3.INITO` und die `CNF`-Ereignisse aller drei Instanzen (`Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF`, `Q_BackgroundColour_3.CNF`) und löst damit `AND_QO.REQ` aus.

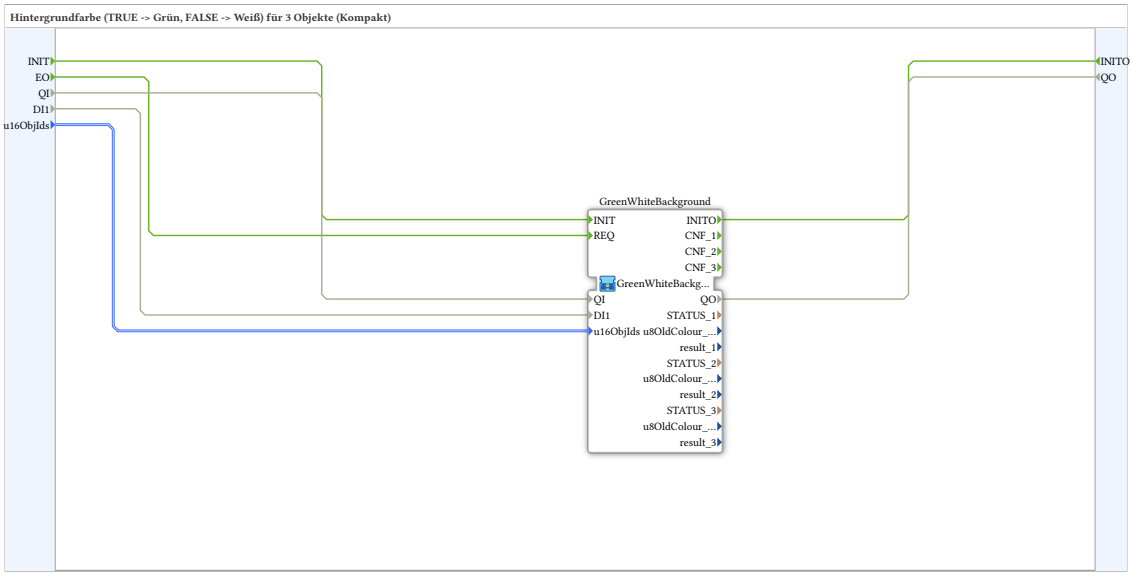
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Weiß, 3 Objekte, BOOL-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground3SC



• – ○ • – ○ • – ○ •

Einleitung

`GreenWhiteBackground3SC` ist ein **Kompakt-Wrapper** um `GreenWhiteBackground3S` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Schnittstelle (Quality-Muster): Der Wrapper besitzt zusätzlich den Ereignis-Eingang `INIT` und den Ereignis-Ausgang `INITO` (je `EInit`) sowie den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`) und den Datenausgang `QO` (`BOOL`). Sie sind direkt an die innere Instanz `GreenWhiteBackground` durchgeschleift: `INIT` ⟶ `GreenWhiteBackground.INIT`, `GreenWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground.QI`, `GreenWhiteBackground.QO` ⟶ `QO`.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

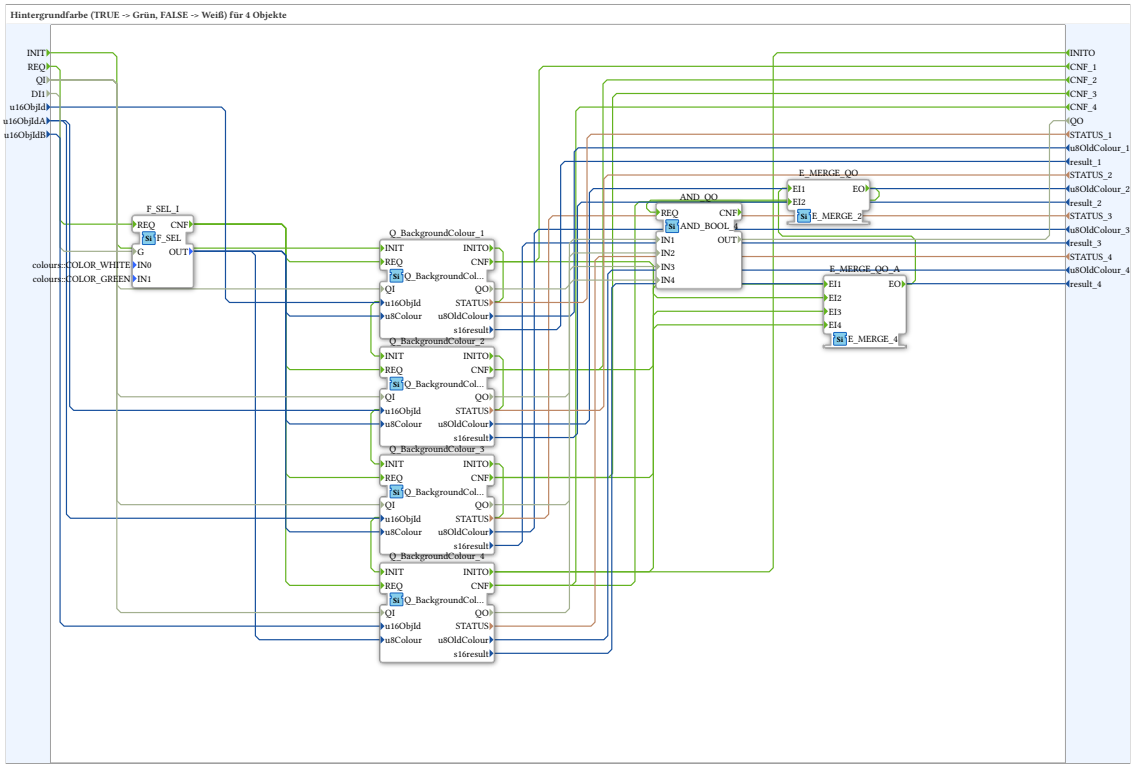
Zusammenfassung

Reine Weiterleitung an `GreenWhiteBackground3S` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground4



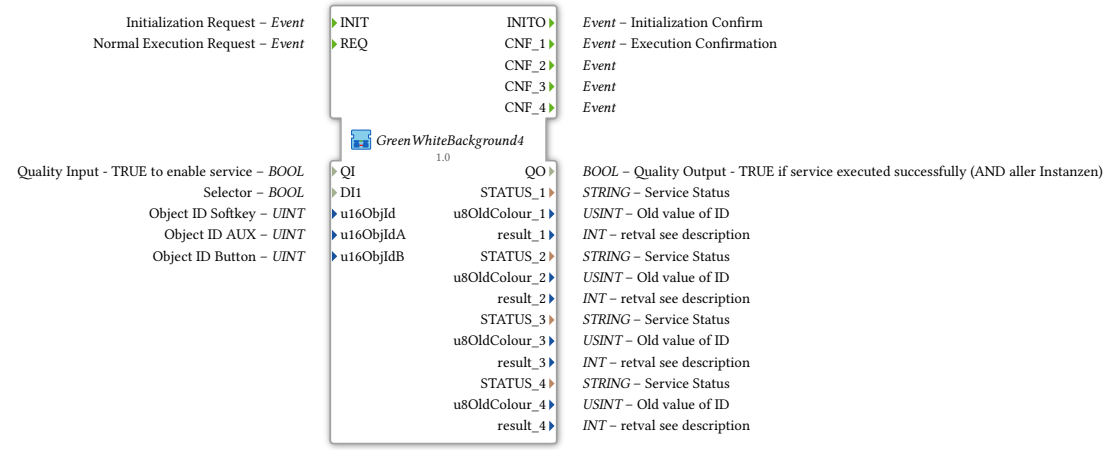
• – ○ • – ○ • – ○ •

Einleitung

`GreenWhiteBackground4` schaltet die VT-Hintergrundfarbe von 4 Objekten anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über die Eingänge `u16ObjId, u16ObjIdA, u16ObjIdB` übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`u16ObjId`	`Q_BackgroundColour` (normales Objekt)
2	`u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
3	`u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)
4	`u16ObjIdB`	`Q_BackgroundColour` (normales Objekt)

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Initialisierung und Status (INIT/INITO, QI/QO)

- **Ereignis-Eingang** `INIT` (`EInit`, Initialization Request) und **Ereignis-Ausgang** `INITO` (`EInit`, Initialization Confirm)
- **Daten-Eingang** `QI` (`BOOL`, Vorgabe `TRUE`, Quality Input) wird an `Q_BackgroundColour_1.QI`, `Q_BackgroundColour_2.QI`, `Q_BackgroundColour_3.QI` und `Q_BackgroundColour_4.QI` weitergegeben
- **Daten-Ausgang** `QO` (`BOOL`, Quality Output): UND (`AND_QO`, Typ `iec61131::booleanOperators::AND_BOOL_4`) der `QO`-Ausgänge von `Q_BackgroundColour_1.QO`, `Q_BackgroundColour_2.QO`, `Q_BackgroundColour_3.QO` und `Q_BackgroundColour_4.QO`
- **Initialisierungskette**: `INIT` → `Q_BackgroundColour_1.INIT`, `Q_BackgroundColour_1.INITO` → `Q_BackgroundColour_2.INIT`, `Q_BackgroundColour_2.INITO` → `Q_BackgroundColour_3.INIT`, `Q_BackgroundColour_3.INITO` → `Q_BackgroundColour_4.INIT`, `Q_BackgroundColour_4.INITO` → `INITO`
- `E_MERGE_QO_A` (Typ `iec61499::events::E_MERGE_4`) führt `Q_BackgroundColour_4.INITO`, `Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF` und `Q_BackgroundColour_3.CNF` zusammen und gibt das Ergebnis an `E_MERGE_QO.EI1`
- `E_MERGE_QO` (Typ `iec61499::events::E_MERGE_2`) führt `E_MERGE_QO_A.EO` und `Q_BackgroundColour_4.CNF` zusammen und löst `AND_QO.REQ` aus

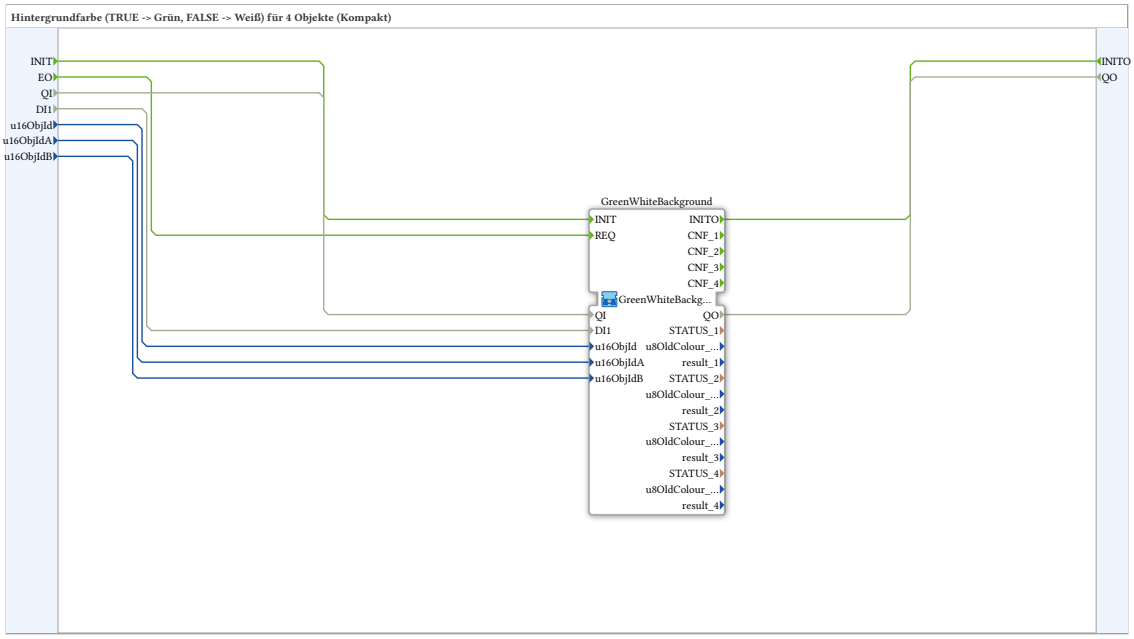
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Weiß, 4 Objekte, BOOL-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground4C



• – ○ • – ○ • – ○ •

Einleitung

`GreenWhiteBackground4C` ist ein **Kompakt-Wrapper** um `GreenWhiteBackground4` — reicht `u16ObjId` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen. Zusätzlich werden `INIT` ⟶ `GreenWhiteBackground.INIT`, `GreenWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground.QI` (Initialwert `TRUE`) und `GreenWhiteBackground.QO` ⟶ `QO` durchgereicht.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

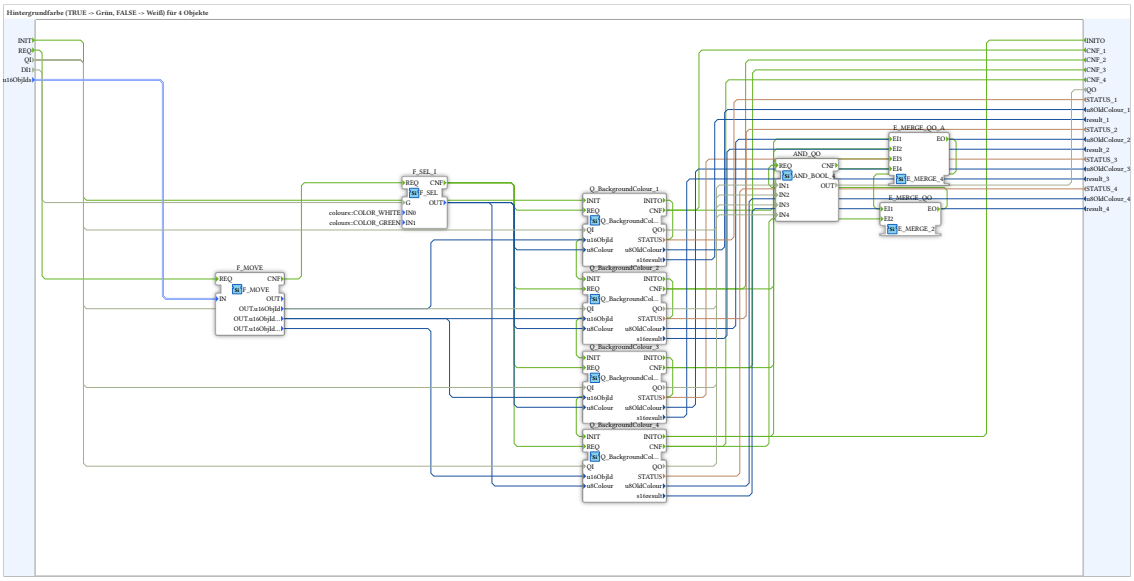
Zusammenfassung

Reine Weiterleitung an `GreenWhiteBackground4` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground4S



• – ○ • – ○ • – ○ •

Einleitung

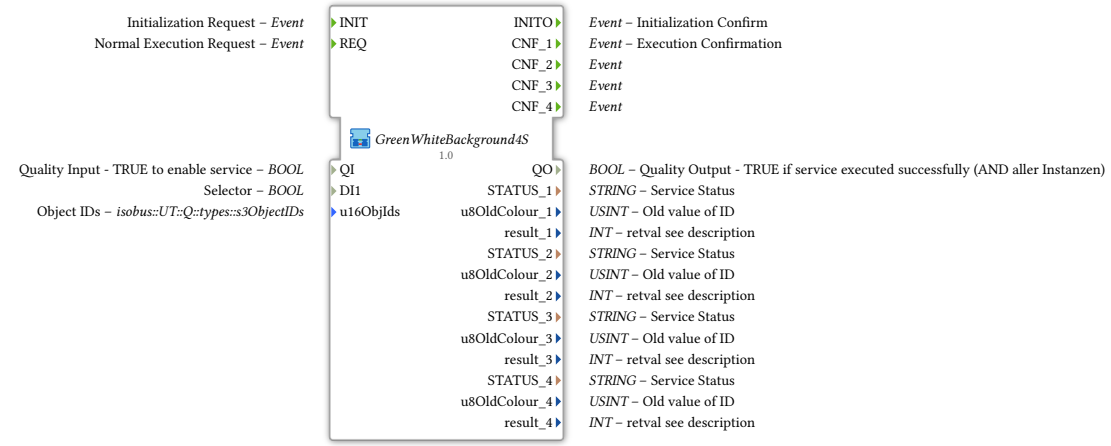
`GreenWhiteBackground4S` schaltet die VT-Hintergrundfarbe von 4 Objekten anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`F_MOVE.OUT.u16ObjId`	`Q_BackgroundColour` (normales Objekt)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
3	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)
4	`F_MOVE.OUT.u16ObjIdB`	`Q_BackgroundColour` (normales Objekt)

Initialisierung und Qualität: Zusätzlich zu den bisherigen Ereignissen besitzt der Baustein den Ereigniseingang `INIT`, den Ereignisausgang `INITO` sowie `QI` (BOOL, Eingang, Initialwert `TRUE`) und `QO` (BOOL, Ausgang, UND aller Instanzen). Init-Kette: `INIT` ⟶ `Q_BackgroundColour_1` ⟶ `Q_BackgroundColour_2` ⟶ `Q_BackgroundColour_3` ⟶ `Q_BackgroundColour_4` ⟶ `INITO` (jeweils `INITO` ⟶ nächstes `INIT`). `QI` wird an alle `Q_BackgroundColour_n.QI` weitergereicht. `QO` bildet `AND_QO` (`iec61131::booleanOperators::AND_BOOL_4`): `Q_BackgroundColour_1.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN2`, `Q_BackgroundColour_3.QO` ⟶ `AND_QO.IN3`, `Q_BackgroundColour_4.QO` ⟶ `AND_QO.IN4`, `AND_QO.OUT` ⟶ `QO`. `E_MERGE_QO_A`

(`E_MERGE_4`) führt `Q_BackgroundColour_4.INITO` und die `CNF`-Ausgänge von `Q_BackgroundColour_1`, `Q_BackgroundColour_2` und `Q_BackgroundColour_3` zusammen, `E_MERGE_QO` (`E_MERGE_2`) fügt `Q_BackgroundColour_4.CNF` hinzu; `E_MERGE_QO.EO` triggert `AND_QO.REQ`.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



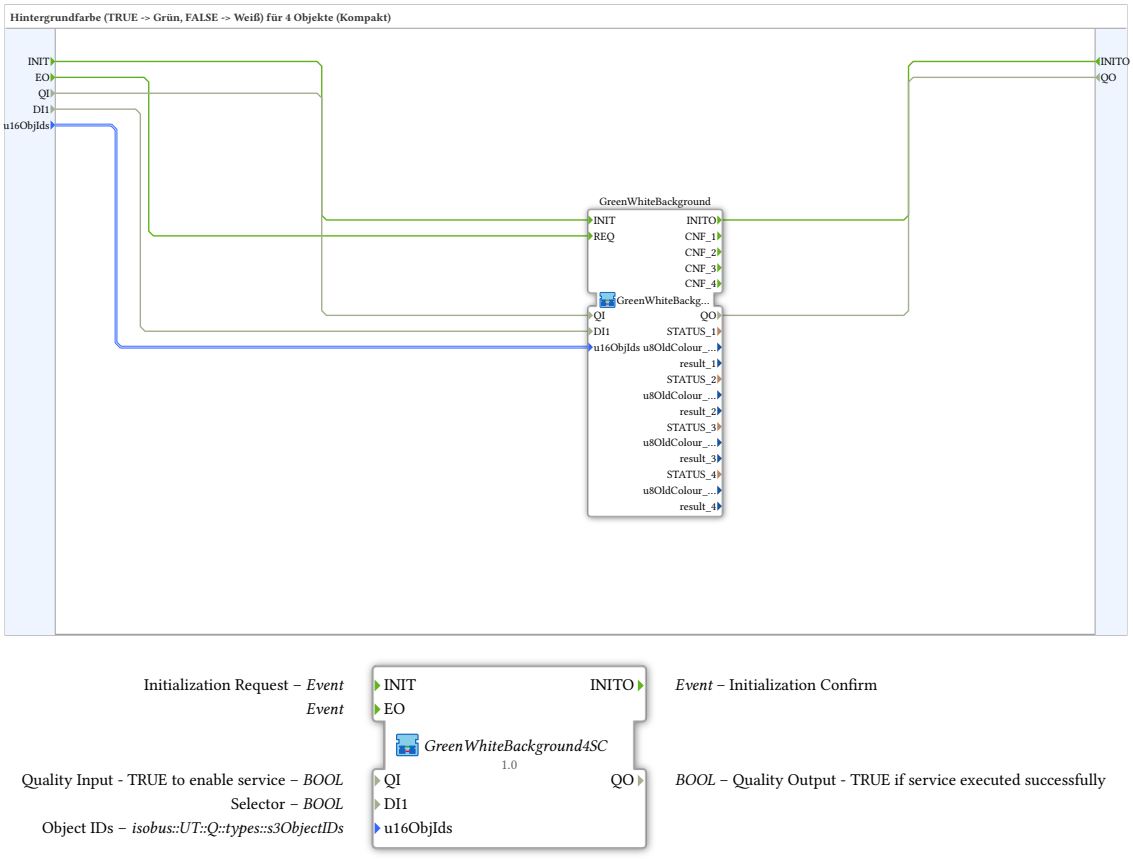
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Grün/Weiß, 4 Objekte, BOOL-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GreenWhiteBackground4SC



• – ○ • – ○ • – ○ •

Einleitung

`GreenWhiteBackground4SC` ist ein **Kompakt-Wrapper** um `GreenWhiteBackground4S` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

Zusätzlich führt der Wrapper die Schnittstelle für Initialisierung und Qualität nach außen: Ereigniseingang `INIT`, Ereignisausgang `INITO`, Dateneingang `QI` (`BOOL`, Initialwert `TRUE`) und Datenausgang `QO` (`BOOL`). Sie sind 1:1 an die innere Instanz `GreenWhiteBackground` durchgeschleift: `INIT` ⟶ `GreenWhiteBackground.INIT`, `GreenWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground.QI`, `GreenWhiteBackground.QO` ⟶ `QO`.

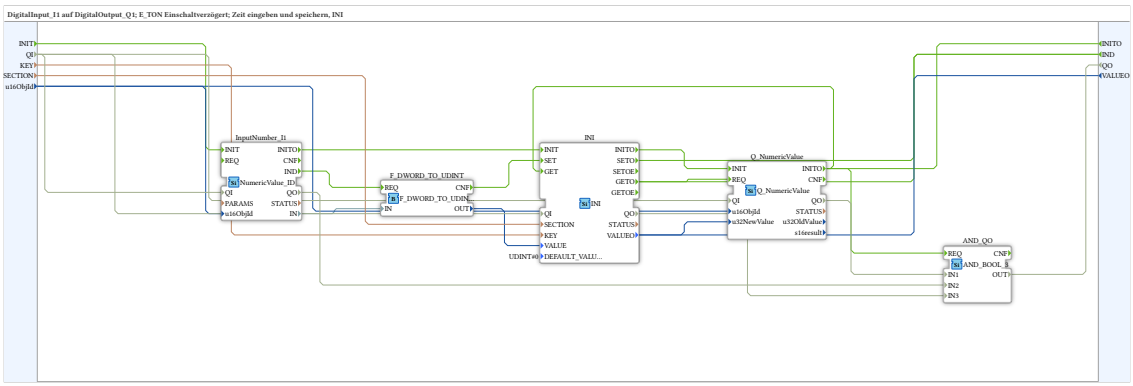
Zusammenfassung

Reine Weiterleitung an `GreenWhiteBackground4S` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

INI_IN_AND_STORE_UDINT



• – ○ • – ○ • – ○ •

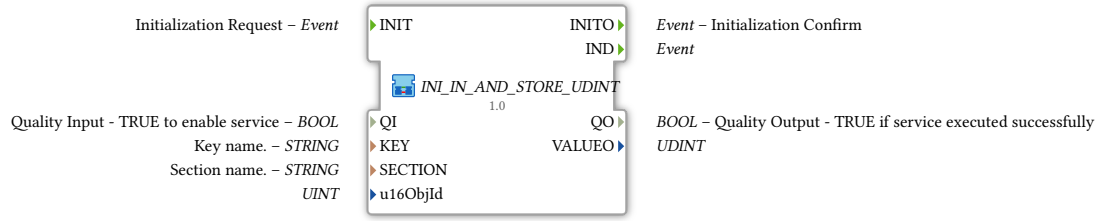
Einleitung

`INI_IN_AND_STORE_UDINT` speichert einen über VT eingegebenen UDINT-Wert persistent in einer INI-Datei und lädt ihn beim Deployment automatisch wieder. Dies ist die MyLib_B-Variante der aus MyLib_AX bekannten INI-Speicherfamilie.

Allgemeines Muster siehe INI_IN_AND_STORE / NVS_IN_AND_STORE (gemeinsames Muster).

Quality-Anschlüsse: Der Baustein besitzt zusätzlich den Dateneingang `QI` (BOOL, Standard `TRUE`, „Quality Input – TRUE, um den Dienst freizugeben“) und den Datenausgang `QO` (BOOL, „Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde“). `QI` ist mit `Q_NumericValue.QI`, `InputNumber_I1.QI` und `INI.QI` verbunden; `QO` ist das UND (`AND_QO`, `iec61131::booleanOperators::AND_BOOL_3`) aus `Q_NumericValue.QO`, `InputNumber_I1.QO` und `INI.QO`.

Zusätzlich besitzt der Baustein die Ereignisse `INIT` (Eingang, `EInit`) und `INITO` (Ausgang, `EInit`); die Initialisierung wird gekettet: `INIT` ⟶ `InputNumber_I1.INIT`; `InputNumber_I1.INITO` ⟶ `INI.INIT`; `INI.INITO` ⟶ `Q_NumericValue.INIT`; `Q_NumericValue.INITO` ⟶ `INITO` (und ⟶ `AND_QO.REQ`).



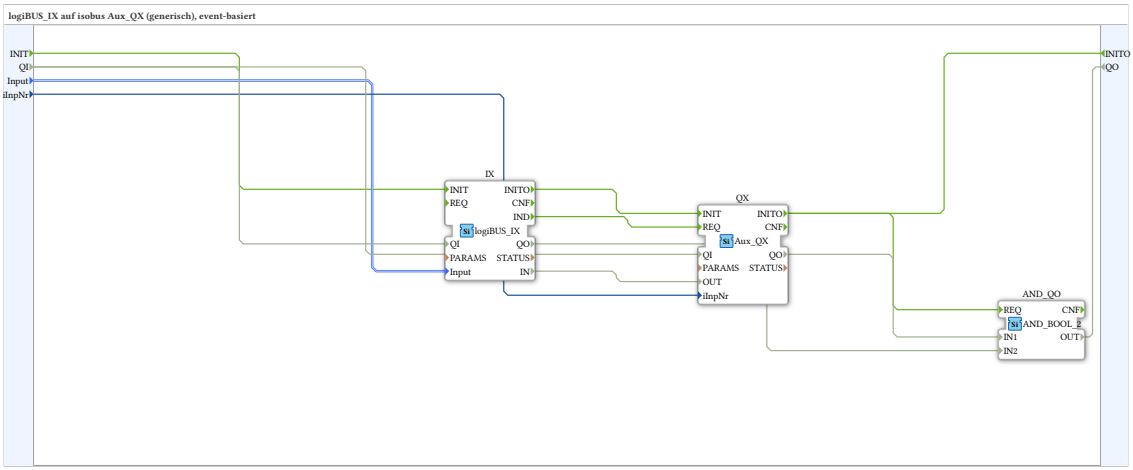
Zusammenfassung

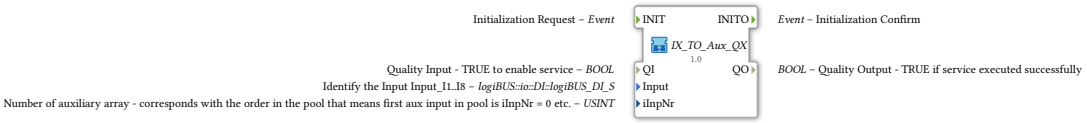
UDINT-Variante der INI-Speicherfamilie (MyLib_B).

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

IX_TO_Aux_QX





• — ○ • — ○ • — ○ •

Einleitung

Die SubApp `IX_TO_Aux_QX` dient als generische, ereignisbasierte Brücke zwischen einem logiBUS-Eingang (Typ `logiBUS_DI_S`) und einem ISOBUS-Auxiliary-Output (`Aux_QX`). Sie kapselt die notwendige Verbindungslogik, um einen einzelnen logiBUS-Digital-Eingang (I1..I8) in einen ISOBUS-Aux-Output umzuwandeln. Die übergebene Eingangsnummer `iInpNr` bestimmt dabei die Reihenfolge im Auxiliary-Output-Pool. Die SubApp wurde mit dem Standard IEC 61499-2 erstellt und ist als wiederverwendbare Komponente konzipiert.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialization Request; wird an `IX.INIT` weitergegeben.

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialization Confirm; kommt von `QX.INITO`.

Daten-Eingänge

Name	Typ	Kommentar
`Input`	`logiBUS::io::DI::logiBUS_DI_S`	Identifiziert den logiBUS-Eingang (I1..I8). Default: `Invalid`
`QI`	`BOOL`	Quality Input – TRUE aktiviert den Dienst. Default: `TRUE`
`iInpNr`	`USINT`	Nummer des Auxiliary-Arrays – entspricht der Reihenfolge im Pool (0 = erster Aux-Input)

Daten-Ausgänge

- **QO** (`BOOL`): Quality Output – UND-Verknüpfung (`AND_QO`, `AND_BOOL_2`) von `QX.QO` und `IX.QO`.

Adapter

Keine Adapter an der SubApp-Schnittstelle vorhanden.

Funktionsweise

Die SubApp instanziiert intern zwei Funktionsblöcke: `IX` (Typ `logiBUS::io::DI::logiBUS_IX`) und `QX` (Typ `isobus::UT::io::Auxiliary::OUT::Aux_QX`). Die Verbindung erfolgt auf Ereignis- und Datenebene:

- **Ereigniskette:** Das Ereignis `IND` des `IX`-Blocks wird direkt mit dem Eingang `REQ` von `QX` verbunden. Dadurch wird bei jedem neuen Eingangssignal des logiBUS-Moduls ein Ausgabeereignis im `QX` ausgelöst.
- **Datenpfad:** Der Datenwert `IN` von `IX` (der den aktuellen Zustand des logiBUS-Eingangs liefert) wird an den Daten-Eingang `OUT` von `QX` weitergegeben. Zusätzlich wird der SubApp-Parameter `iInpNr` direkt an `QX.iInpNr` angeschlossen, sodass die Zuordnung des Auxiliary-Outputs festgelegt wird.
- **Initialisierung und Qualität:** `INIT` ⟶ `IX.INIT`; `IX.INITO` ⟶ `QX.INIT`; `QX.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`. `QI` wird an `IX.QI` und `QX.QI` verteilt (kein fest verdrahteter `QI`-Parameter mehr); `QX.QO` ⟶ `AND_QO.IN1`, `IX.QO` ⟶ `AND_QO.IN2`, `AND_QO.OUT` ⟶ `QO`.

Somit verarbeitet die SubApp ein eingehendes logiBUS-Signal (z. B. von einem digitalen Eingang) und setzt es in einen ISOBUS-Auxiliary-Output um. Die Übertragung erfolgt streng ereignisgetrieben, was eine synchrone Reaktion ohne zusätzliche Polling-Logik ermöglicht.

Technische Besonderheiten

- **Ereignisbasiert:** Die gesamte Verarbeitung ist ereignisorientiert (durch `IND` bzw. `REQ`) und eignet sich für Echtzeitanwendungen.
- **Generische Konfiguration:** Über `iInpNr` wird der Ziel-Aux-Output dynamisch bestimmt, ohne dass die SubApp selbst geändert werden muss.
- **Typverwendung:** Die Schnittstelle nutzt den im Compiler importierten Typ `logiBUS::io::DI::logiBUS_DI_S`, der eine strukturierte Beschreibung eines logiBUS-Digital-Eingangs repräsentiert.
- **Initialwert:** Der Daten-Eingang `Input` ist mit `Invalid` initialisiert, was auf einen undefinierten Anfangszustand hinweist und eine bewusste Erstkonfiguration vor dem Start erzwingt.
- **Wiederverwendbarkeit:** Die SubApp ist aus einem größeren Projekt (Übung 003c) extrahiert und kann in verschiedenen Anwendungen eingesetzt werden, die eine logiBUS-zu-ISOBUS-Kopplung benötigen.

Zustandsübersicht

Die SubApp selbst besitzt keinen eigenen Zustandsautomaten. Der aktuelle Zustand wird vollständig durch die internen Funktionsblöcke bestimmt:

- `IX` (logiBUS-Input) nimmt je nach Eingangssignal die Zustände `IDLE`, `RUN`, oder Fehlerzustände an.
- `QX` (ISOBUS-Aux-Output) verwaltet den Zustand des Ausgangs (z. B. `INACTIVE`, `ACTIVE`).

Die SubApp verbindet lediglich die Ereignis- und Datenflüsse. Für eine detaillierte Zustandsbeschreibung wird auf die Dokumentation der jeweiligen Funktionsblöcke verwiesen.

Anwendungsszenarien

Typische Einsatzfälle liegen in der Landwirtschaft und mobilen Maschinen, wo logiBUS-Sensoren oder Schalter mit ISOBUS-Terminals oder Steuergeräten verbunden werden sollen:

- **Fernbedienung:** Ein logiBUS-Taster (z. B. für eine Hydraulikfunktion) steuert über den ISOBUS einen Auxiliary-Output (z. B. ein Magnetventil).
- **Ein-/Aus-Signale:** Digital-Eingänge von Maschinen werden als Steuerbefehle in eine ISOBUS-Steuerung eingespeist.
- **Test- und Simulationsumgebungen:** Die SubApp kann verwendet werden, um logiBUS-Signale in einer ISOBUS-Umgebung zu simulieren oder zu verifizieren.

Vergleich mit ähnlichen Bausteinen

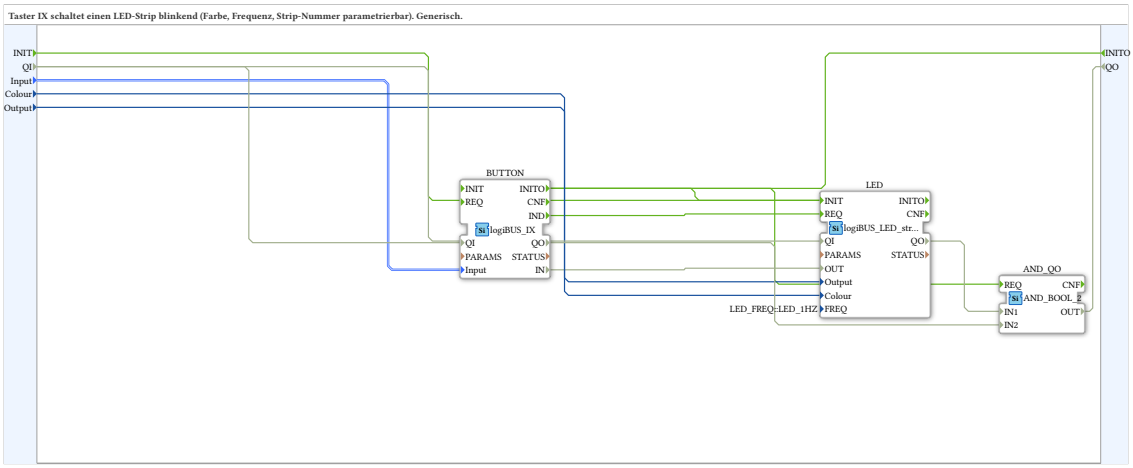
Es gibt möglicherweise spezialisiertere Bausteine, die mehrere logiBUS-Eingänge bündeln oder zusätzliche Logik (z. B. Filter, Entprellung) enthalten. `IX_TO_Aux_QX` zeichnet sich jedoch durch seine Einfachheit und Flexibilität aus: Es erfüllt genau eine Aufgabe – die direkte Durchschaltung eines einzelnen Eingangs – und lässt sich durch die `iInpNr`-Parametrierung an verschiedene Poolpositionen anpassen. Gegenüber

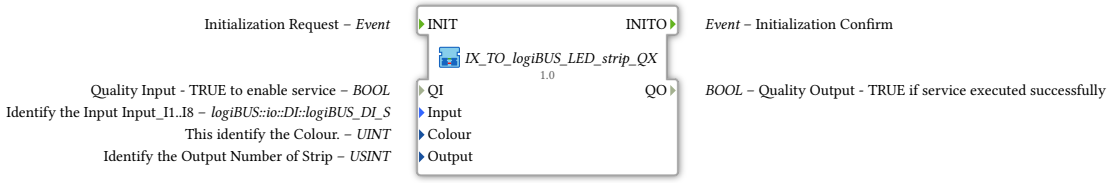
spezifisch programmierten Lösungen bietet es eine klare IEC 61499-konforme Struktur und ist dadurch leicht in größere Automatisierungssysteme integrierbar.

Fazit

Die SubApp `IX_TO_Aux_QX` stellt eine saubere, ereignisgesteuerte Kopplung zwischen einem logiBUS-Eingang und einem ISOBUS-Auxiliary-Output dar. Durch die geringe Schnittstelle (nur zwei Dateneingänge) ist sie leicht zu parametrieren und in verschiedensten Kontexten einsetzbar. Die interne Verwendung bewährter Funktionsblöcke gewährleistet eine robuste und standardkonforme Lösung. Sie eignet sich insbesondere für modulare Steuerungsarchitekturen, in denen logiBUS-Signale in ISOBUS-Systeme überführt werden müssen, ohne den Gesamtcode zu verkomplizieren.

IX_TO_logiBUS_LED_strip_QX





• – ○ • – ○ • – ○ •

Einleitung

Die Subapp **IX_TO_logiBUS_LED_strip_QX** verbindet einen Taster-Eingang vom Typ `logiBUS_IX` mit einem LED-Streifen-Baustein vom Typ `logiBUS_LED_strip_QX`. Sie wird eingesetzt, um einen LED-Streifen über einen Taster blinkend zu schalten. Dabei können die Farbe, der verwendete Taster-Eingang und die Nummer des LED-Streifens über die Dateneingänge der Subapp vorgegeben werden.

Die Subapp ist als generische, wiederverwendbare Einheit ausgelegt. Die eigentliche Steuerlogik ist intern verdrahtet; nach außen hin sind die Parametriereingänge, die Initialisierungs-Ereignisse `INIT`/`INITO` und die Status-Anschlüsse `QI`/`QO` sichtbar.

Schnittstellenstruktur

Die Subapp besitzt außer den Initialisierungs-Ereignissen `INIT`/`INITO` keine Ereignis-Schnittstellen. Sie arbeitet rein datengetrieben und kapselt die interne Ereignisverarbeitung vollständig.

Ereignis-Eingänge

- `INIT` (`EInit`): Initialization Request

Ereignis-Ausgänge

- `INITO` (`EInit`): Initialization Confirm

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung

QI	BOOL	TRUE	Quality Input - TRUE aktiviert den Dienst
Input	logiBUS::io::DI::logiBUS_DI_S	Invalid	Identifiziert den verwendeten Eingang Input_I1 bis Input_I8
Colour	UINT	LED_COLOURS::LED_GREEN	Bestimmt die Farbe des LED-Streifens
Output	USINT	–	Bestimmt die Nummer des anzusteuernden LED-Streifens

Daten-Ausgänge

- QO (BOOL): Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde (UND von LED.QO und BUTTON.QO, Baustein AND_QO vom Typ iec61131::booleanOperators::AND_BOOL_2)

Adapter

- Keine

Funktionsweise

Die Subapp setzt sich aus zwei internen Funktionsbausteinen zusammen:

- BUTTON vom Typ logiBUS_IX
- LED vom Typ logiBUS_LED_strip_QX

Der externe Eingang Input wird an den Dateneingang BUTTON.Input weitergeführt. Dadurch wird festgelegt, welcher physische Eingang des Taster-Moduls überwacht wird. Der interne Baustein BUTTON wird über QI (vom QI-Eingang der Subapp) aktiviert und wartet auf eine Änderung des Eingangssignals.

Wird der Taster betätigt oder losgelassen, erzeugt BUTTON das Ereignis IND. Dieses Ereignis ist direkt mit dem Ereigniseingang REQ des LED-Bausteins verbunden. Zeitgleich wird der aktuelle Eingangszustand von BUTTON.IN an den Dateneingang LED.OUT übergeben.

Der LED-Baustein wird durch das Ereignis REQ verarbeitet. Er verwendet:

- Output zur Auswahl des LED-Streifens
- Colour zur Bestimmung der Farbe
- den übergebenen Wert an OUT als Schaltsignal

Die Initialisierung läuft als Kette INIT → LED.INIT, LED.INITO → BUTTON.INIT, BUTTON.INITO → INITO (und zusätzlich → AND_QO.REQ). QI wird an LED.QI und BUTTON.QI weitergegeben; LED.QO und BUTTON.QO werden per UND zu QO verknüpft.

Der Parameter FREQ des LED-Bausteins ist intern auf LED_FREQ::LED_1HZ gesetzt. Dadurch blinkt der ausgewählte LED-Streifen mit 1 Hz, wenn der Tasterzustand aktiv ist. Ist der Tasterzustand inaktiv, wird der LED-Streifen ausgeschaltet.

Technische Besonderheiten

- Die Subapp besitzt außer INIT/INITO keine eigenen Ereignis-Ein- oder -Ausgänge.
- Sie besitzt keine Adapter.
- Alle Verbindungen zwischen BUTTON und LED sind intern verdrahtet.
- Die Eingänge Input, Colour und Output werden direkt auf die internen Funktionsbausteine durchgeschleift.
- Die Blinkfrequenz ist nicht als externer Eingang verfügbar, sondern als Parameter des internen LED-Bausteins hinterlegt. Aktuell ist LED_1HZ eingestellt.
- Die Farbe wird über die Konstante LED_COLOURS::LED_GREEN vorbelegt, kann aber zur Laufzeit über den Eingang Colour geändert werden.
- Die interne Verdrahtung verwendet die 4diac-typischen logiBUS-Datentypen und gewährleistet so eine typsichere Verbindung.

Zustandsübersicht

Die Subapp selbst besitzt keine eigene Zustandsmaschine. Die Zustände ergeben sich aus dem Verhalten der internen Funktionsbausteine:

Zustand	Beschreibung
Warten	`BUTTON` überwacht den konfigurierten Eingang. Es wurde kein Tasterereignis erkannt.
Ereignis	Bei einem Tasterwechsel erzeugt `BUTTON` das Ereignis `IND` und löst damit `LED.REQ` aus.
LED inaktiv	Der Tasterzustand ist `FALSE`. Der LED-Streifen erhält kein aktives Schaltsignal und bleibt aus.
LED blinkt	Der Tasterzustand ist `TRUE`. Der LED-Streifen wird mit der konfigurierten Farbe und Frequenz angesteuert.

Anwendungsszenarien

- Ein Taster an einem `logiBUS_IX`-Eingang schaltet einen Status-LED-Streifen an einer Maschine.
- Über den Eingang `Colour` kann die Signalfarbe der Anzeige geändert werden, etwa von Grün auf Rot.
- Über den Eingang `Output` kann zwischen verschiedenen LED-Streifen umgeschaltet werden.
- Die Subapp kann in übergeordneten Steuerungsprojekten mehrfach instanziiert werden, wenn mehrere Taster-LED-Streifen-Kombinationen benötigt werden.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einer direkten Verbindung eines `logiBUS_IX` mit einem `logiBUS_LED_strip_QX` bietet die Subapp eine gekapselte und wiederverwendbare Struktur. Die Verdrahtung muss nicht in jeder Anwendung neu erstellt werden.

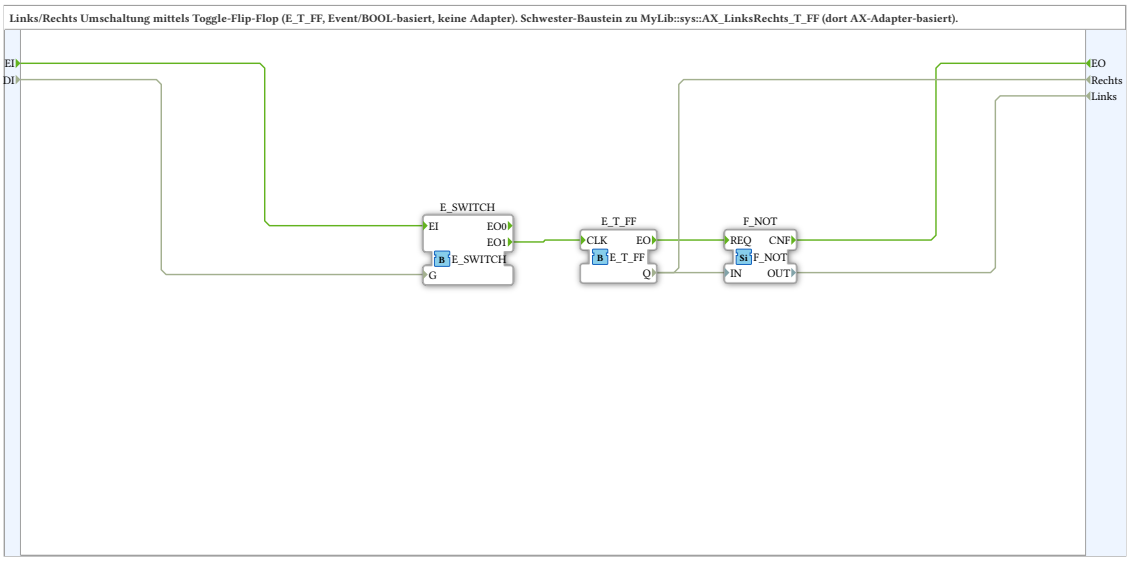
Gegenüber einem einfachen LED-Ausgangsbaustein ohne Blinkfunktion ermöglicht der interne LED-Streifen-Baustein eine fest eingestellte Blinkfrequenz. Zusätzlich können Farbe und Streifennummer flexibel parametriert werden.

Ein wesentlicher Unterschied zu vollständigen Funktionsbausteinen mit eigenen Ereignissen besteht darin, dass die Subapp außer der Initialisierungsbestätigung `INITO` und dem Status `QO` keine Rückmeldung nach außen gibt. Sie ist daher vor allem für einfache, autarke Taster-Leuchtanzeigen geeignet.

Fazit

Die Subapp `IX_TO_logiBUS_LED_strip_QX` ist eine kompakte und wiederverwendbare Lösung für die Ansteuerung eines blinkenden LED-Streifens über einen Taster. Sie kombiniert die logiBUS-Eingabe und den LED-Streifen-Baustein in einer übersichtlichen Einheit. Durch die Parametrierung von Eingang, Farbe und Streifennummer lässt sie sich flexibel in unterschiedlichen Steuerungsprojekten einsetzen.

LinksRechts_T_FF_Event



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **LinksRechts_T_FF_Event** ist eine Subapplikation (SubApp) zur einfachen Links/Rechts-Umschaltung auf Basis eines Toggle-Flip-Flops. Er verwendet ausschließlich Event- und BOOL-basierte Signale und benötigt keine Adapter. Die SubApp ist als Wiederverwendungsbaustein konzipiert und stellt eine ereignisgesteuerte Alternative zu adapterbasierten Schwesterbausteinen dar.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung
EI	Event	Ereigniseingang zur Auslösung der Umschaltlogik.

Ereignis-Ausgänge

Name	Typ	Beschreibung
EO	Event	Ereignisausgang, signalisiert die abgeschlossene Aktualisierung der Ausgangssignale.

Daten-Eingänge

Name	Typ	Beschreibung
DI	BOOL	Steuersignal für die Freigabe der Toggle-Funktion.

Daten-Ausgänge

Name	Typ	Beschreibung
Rechts	BOOL	Ausgangssignal für die Richtung „Rechts“, entspricht dem internen Flip-Flop-Zustand.
Links	BOOL	Ausgangssignal für die Richtung „Links“, ist die invertierte Version von `Rechts`.

Adapter

Keine. Die SubApp verwendet ausschließlich direkte Event- und Datenverbindungen.

Funktionsweise

Die SubApp realisiert eine Toggle-Funktion mit optionaler Freigabe:

1. Ein Ereignis an `EI` wird über den `E_SWITCH`-Baustein geführt.
2. Der Daten-Eingang `DI` steuert den Schalter `E_SWITCH.G`:
   - Ist `DI = TRUE`, wird das Ereignis an den Takteingang `CLK` des Toggle-Flip-Flops `E_T_FF` weitergeleitet.
   - Ist `DI = FALSE`, wird das Ereignis blockiert, und es findet keine Zustandsänderung statt.
3. Bei jedem wirksamen Takt wechselt der Ausgang `Q` des Flip-Flops seinen Zustand (Toggle):
   - `Q = FALSE` ⟶ `Rechts = FALSE` und `Links = TRUE`
   - `Q = TRUE` ⟶ `Rechts = TRUE` und `Links = FALSE`

4. Der Ausgang Q wird direkt an Rechts geführt und gleichzeitig über den Negator F_NOT invertiert, um Links zu erzeugen.
5. Nach der Negation wird das Bestätigungsereignis (CNF) von F_NOT an den Ausgang EO weitergegeben. Dieses Ereignis signalisiert, dass die Ausgänge Rechts und Links gültig sind.

Technische Besonderheiten

- Die SubApp besteht aus den Funktionsbausteinen E_T_FF, E_SWITCH und F_NOT.
- E_T_FF ist ein ereignisgesteuertes Toggle-Flip-Flop aus der IEC-61499-Standardbibliothek.
- E_SWITCH arbeitet als ereignisbasierter Schalter: Nur bei G = TRUE wird das Eingangsereignis an den Ausgang EO1 weitergeleitet.
- F_NOT ist ein bitweiser Negator aus der IEC-61131-Bibliothek und wird hier als logischer Inverter eingesetzt.
- Die Ausgänge Rechts und Links sind immer komplementär zueinander.
- Durch die SubApp-Kapselung entsteht ein wiederverwendbarer, adapterfreier Baustein, der sich leicht in größere Automatisierungsprojekte integrieren lässt.

Zustandsübersicht

Die SubApp besitzt keinen eigenen expliziten Zustandsautomaten, ihr Verhalten lässt sich jedoch über den internen Flip-Flop-Zustand beschreiben:

Zustand von Q	Rechts	Links	Bedeutung
FALSE	FALSE	TRUE	Links aktiv
TRUE	TRUE	FALSE	Rechts aktiv

Ein Wechsel zwischen den Zuständen erfolgt ausschließlich dann, wenn ein Ereignis an EI anliegt und DI = TRUE ist.

Anwendungsszenarien

- Richtungsumschaltung bei Förderbändern oder Transportanlagen.
- Alternierende Ansteuerung von zwei Signalleuchten (z. B. „links/rechts“-Anzeige).
- Toggle-Steuerung für Antriebe mit zwei Richtungsausgängen.
- Ereignisbasierte Umschaltung in IEC-61499-Applikationen, bei denen keine Adapter verwendet werden sollen.

Vergleich mit ähnlichen Bausteinen

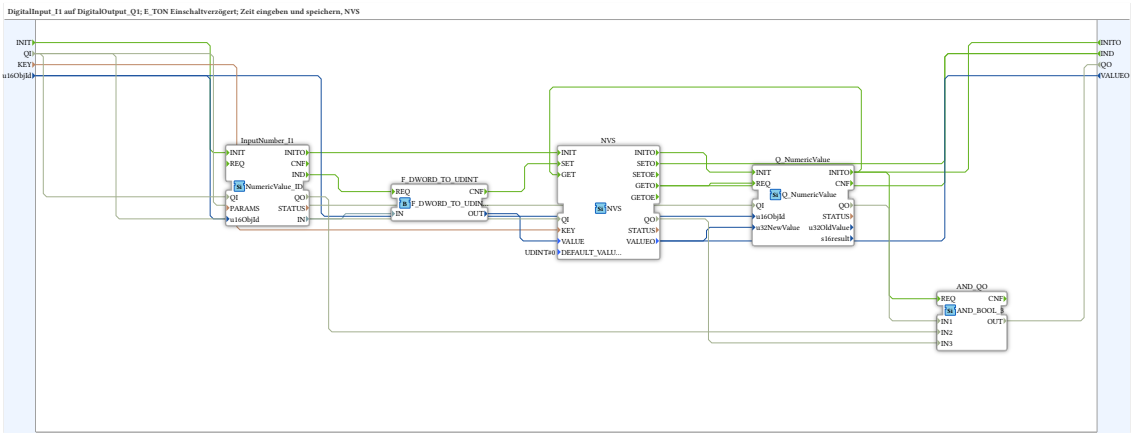
Baustein	Konzept	Unterschied
LinksRechts_T_FF_Event	Event-/BOOL-basiert, keine Adapter	Einfache Integration in reine Event-/Daten-Netzwerke.
MyLib::sys::AX_LinksRechts_T_FF	Adapter-basiert	Verwendet einen Adapter für strukturierte Verbindungen, ermöglicht kompaktere Applikationsnetze.
Klassisches RS-Flip-Flop	Set/Reset	Benötigt zwei getrennte Eingänge zum Setzen und Rücksetzen; hier genügt ein einzelnes Ereignis zum Umschalten.

Der ereignisbasierte Baustein ist besonders dann vorteilhaft, wenn eine einfache, adapterfreie Lösung mit minimalem Verdrahtungsaufwand gefordert ist.

Fazit

LinksRechts_T_FF_Event ist ein kompakter und klar strukturierter Funktionsbaustein für die Links/Rechts-Umschaltung mit Toggle-Verhalten. Durch die Kombination von E_T_FF, E_SWITCH und F_NOT entsteht eine robuste, ereignisgesteuerte Logik. Die adapterfreie Schnittstelle erleichtert die Wiederverwendung und Integration in unterschiedlichste IEC-61499-Applikationen.

NVS_IN_AND_STORE_UDINT



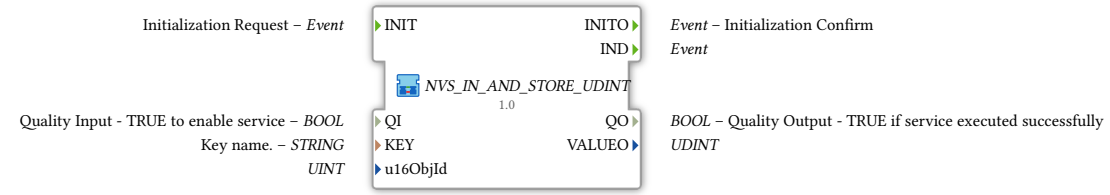
• – ○ • – ○ • – ○ •

Einleitung

`NVS_IN_AND_STORE_UDINT` speichert einen über VT eingegebenen UDINT-Wert persistent im ESP32-Flash (NVS) statt in einer INI-Datei — ansonsten identisch zu `INI_IN_AND_STORE_UDINT`.

Allgemeines Muster siehe INI_IN_AND_STORE / NVS_IN_AND_STORE (gemeinsames Muster).

Quality-Anschlüsse: Der Baustein besitzt zusätzlich den Dateneingang `QI` (BOOL, Standard `TRUE`, „Quality Input – TRUE, um den Dienst freizugeben“) und den Datenausgang `QO` (BOOL, „Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde“). `QI` wird an `Q_NumericValue.QI`, `InputNumber_I1.QI` und `NVS.QI` durchgereicht (kein fester `QI=TRUE` mehr an den internen FBs). `QO` ist das `AND` (`AND_QO`, `AND_BOOL_3`) der `QO`-Ausgänge von `Q_NumericValue` (`IN1`), `InputNumber_I1` (`IN2`) und `NVS` (`IN3`); `AND_QO.REQ` wird von `Q_NumericValue.INITO` ausgelöst (zusätzlich zu `Q_NumericValue.INITO` ⟶ `INITO`).



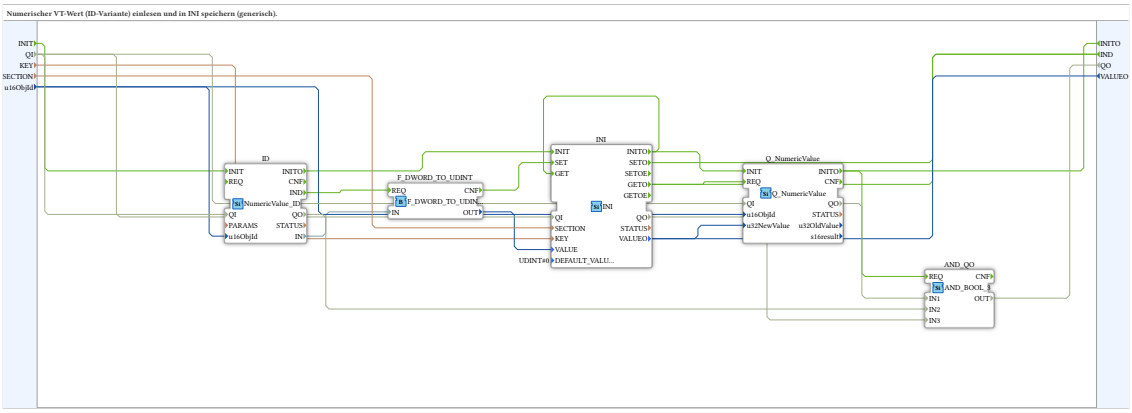
Zusammenfassung

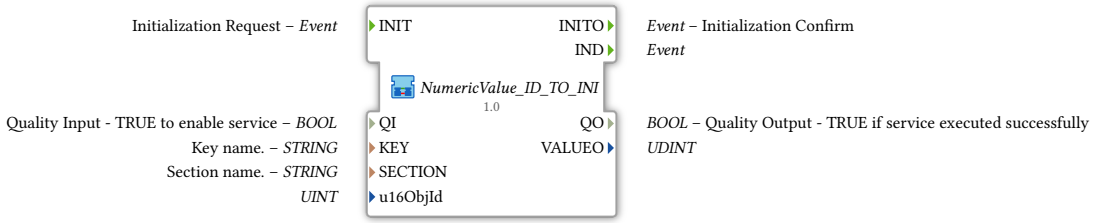
NVS-Pendant zu `INI_IN_AND_STORE_UDINT` (MyLib_B).

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

NumericValue_ID_TO_INI





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `NumericValue_ID_TO_INI` ist eine wiederverwendbare Subapplikation, die einen numerischen Wert von einem ISOBUS-Gerät (ID-Variante) einliest und persistent in einer INI-Datei speichert. Der gespeicherte Wert wird beim Initialisieren wieder geladen und an das Gerät zurückgegeben, sodass der Zustand über einen Neustart hinweg erhalten bleibt. Die Subapp ist generisch aufgebaut und kann für verschiedene Objekt-IDs und beliebige Speicherorte (Section/Key) verwendet werden.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`) – Initialisierungsanforderung; startet die Init-Kette bei `ID.INIT`

Die weitere Verarbeitung wird über interne Ereignisse der enthaltenen Funktionsbausteine angestoßen.

Ereignis-Ausgänge

- **INITO** (`EInit`) – Initialisierungsbestätigung; kommt von `Q_NumericValue.INITO`, dem Ende der Init-Kette
- **IND** – Wird ausgelöst, wenn ein neuer Wert erfolgreich in der INI-Datei gespeichert oder aus ihr geladen wurde. Dieses Ereignis signalisiert der Umgebung, dass der aktuelle Wert (VALUEO) gültig ist und verwendet werden kann.

Daten-Eingänge

- **QI** (BOOL) – Quality Input – TRUE, um den Dienst freizugeben; wird an `ID.QI`, `INI.QI` und `Q_NumericValue.QI` durchgereicht
- **KEY** (STRING) – Schlüsselname innerhalb der INI-Sektion, unter dem der Wert abgelegt wird.
- **SECTION** (STRING) – Name der INI-Sektion, in der der Wert gespeichert wird.
- **u16ObjId** (UINT) – Objekt-ID des numerischen Werts auf dem ISOBUS-Gerät. Standardmäßig auf `ID_NULL` gesetzt.

Daten-Ausgänge

- **QO** (BOOL) – Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde; UND-Verknüpfung (`AND_QO`, `AND_BOOL_3`) von `Q_NumericValue.QO`, `ID.QO` und `INI.QO`
- **VALUEO** (UDINT) – Der aktuell gespeicherte, konvertierte Wert als 32-Bit-Unsigned-Integer (UDINT). Dieser wird sowohl beim Speichern als auch beim Laden aktualisiert und nach außen bereitgestellt.

Adapter

Es sind keine Adapter definiert. Die Kommunikation erfolgt über die oben genannten Ein- und Ausgänge sowie über die interne Verbindung zum ISOBUS-Gerät (über die in der Subapp eingebetteten Funktionsbausteine).

Funktionsweise

Die Subapplikation besteht aus fünf Funktionsbausteinen:

- **NumericValue_ID** (`isobus::UT::io::NumericValue::NumericValue_ID`): Liest den aktuellen Wert von der angegebenen Objekt-ID ein und gibt ihn als `DWORD` aus (Ereignis `IND`).
- **F_DWORD_TO_UDINT** (`iec61131::conversion::F_DWORD_TO_UDINT`): Konvertiert den empfangenen `DWORD`-Wert in einen `UDINT`-Wert.
- **INI** (`eclipse4diac::storage::INI`): Verwaltet den Zugriff auf die INI-Datei. Er unterstützt das Speichern (`SET`) und Laden (`GET`) von Werten mit frei wählbarem Section/Key.

- **Q_NumericValue** (`isobus::UT::Q::Q_NumericValue`): Schreibt einen neuen Wert an das ISOBUS-Gerät zurück.
- **AND_QO** (`iec61131::booleanOperators::AND_BOOL_3`): Verknüpft die `QO`-Ausgänge von `Q_NumericValue`, `ID` und `INI` zum Ausgang `QO`.

Ablauf beim Speichern eines neu eingelesenen Wertes:

1. Der Baustein `NumericValue_ID` sendet bei einem neuen Wert das Ereignis `IND`.
2. Dieses löst die Konvertierung in `F_DWORD_TO_UDINT` aus (`REQ`). Der konvertierte `UDINT` wird am Ausgang `OUT` bereitgestellt.
3. Nach Abschluss der Konvertierung (`CNF`) wird der Wert über das Ereignis `SET` an den `INI`-Baustein übergeben. Der Wert wird zusammen mit den aktuellen `KEY` und `SECTION` gespeichert.
4. Nach erfolgreichem Speichern sendet `INI` das Ereignis `SETO`, das als `IND` an den Ausgang der Subapp weitergegeben wird.

Ablauf beim Laden bei Initialisierung:

1. Beim Start der Subapplikation initialisiert der `INI`-Baustein (`INITO`).
2. Dieses Ereignis löst intern ein `GET` aus, um den gespeicherten Wert aus der INI-Datei zu lesen.
3. Das `GETO`-Ereignis des `INI`-Bausteins wird an `Q_NumericValue.REQ` gesendet, wodurch der geladene Wert (`VALUEO`) an das ISOBUS-Gerät zurückgeschrieben wird.
4. Gleichzeitig wird das `GETO`-Ereignis auch als `IND` an den Ausgang der Subapp geleitet, sodass auch die Umgebung den geladenen Wert erhält.

Somit stellt die Subapp sicher, dass der Wert sowohl bei Änderungen als auch beim Systemstart konsistent ist.

Initialisierung und Quality-Anschlüsse: Die Init-Kette lautet `INIT` → `ID.INIT`, `ID.INITO` → `INI.INIT`, `INI.INITO` → `Q_NumericValue.INIT`, `Q_NumericValue.INITO` → `INITO` und `AND_QO.REQ` (zusätzlich bleibt `INI.INITO` → `INI.GET` bestehen). `QI` (Dateneingang, BOOL, Standard `TRUE`) wird an `Q_NumericValue.QI`, `ID.QI` und `INI.QI` weitergereicht; deren `QO`-Ausgänge gehen auf `AND_QO.IN1` (`Q_NumericValue`), `IN2` (`ID`) und `IN3` (`INI`), `AND_QO.OUT` → `QO` (BOOL). Damit lassen sich alle drei Dienste von außen freigeben und ihr gemeinsamer Erfolg abfragen.

Technische Besonderheiten

- **Persistente Speicherung:** Die Verwendung des `INI`-Bausteins ermöglicht das dauerhafte Ablegen des Werts, sodass dieser nach einem Neustart oder Spannungsausfall wieder verfügbar ist.
- **Generische Konfiguration:** Durch die Eingänge `KEY` und `SECTION` kann der Speicherort dynamisch festgelegt werden, ohne die Subapp intern zu verändern.
- **Konvertierung:** Die Konvertierung von `DWORD` zu `UDINT` stellt sicher, dass der Wert in ein für das ISOBUS-Protokoll übliches Format überführt wird.
- **Automatisches Rückschreiben:** Nach dem Laden wird der Wert sofort an das ISOBUS-Gerät gesendet, sodass die Hardware den zuletzt gespeicherten Zustand übernimmt.
- **Initialwert:** Für den Fall, dass noch kein Wert gespeichert ist, wird über den Parameter `DEFAULT_VALUE` des `INI`-Bausteins der Wert `0` als Fallback verwendet.

Zustandsübersicht

Die Subapplikation kann in folgende Hauptzustände unterteilt werden:

- **Initialisierungsphase:** Der `INI`-Baustein wird gestartet und versucht, den gespeicherten Wert zu laden (`INITO` → `GET`).
- **Warten auf neuen Wert:** Der Baustein `NumericValue_ID` wartet auf ein Ereignis von der Hardware.
- **Verarbeitung eines neuen Wertes:** Sobald ein Wert empfangen wird, erfolgt die Konvertierung und das Speichern in der INI-Datei.
- **Rückschreiben nach Laden:** Nach dem Laden wird der Wert an das Gerät gesendet (über `Q_NumericValue`).
- **Bereit:** Die Subapp ist inaktiv und wartet auf das nächste Ereignis, solange keine Änderung ansteht.

Der Wechsel zwischen den Zuständen erfolgt ereignisgesteuert über die internen Verbindungen der FB-Instanzen.

Anwendungsszenarien

- **ISOBUS-Konfiguration:** Speicherung von Konfigurationswerten (z. B. Einstellungen, Kalibrierdaten) eines landwirtschaftlichen Geräts, die über eine Objekt-ID gelesen und geschrieben werden.

- **Diagnose und Wartung:** Persistente Speicherung von Zählerständen oder Diagnosewerten, die nach einem Neustart wiederhergestellt werden müssen.
- **Maschinensteuerung:** Speichern von Sollwerten oder Parametern, die zur Laufzeit verändert werden und über einen Neustart hinweg erhalten bleiben sollen.
- **Generische Datenübernahme:** Einbinden in größere Applikationen, bei denen verschiedene numerische Parameter mit minimalem Aufwand persistent abgelegt werden sollen – durch einfaches Setzen von `KEY`, `SECTION` und `u16ObjId`.

Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu einfachen `NumericValue`-Bausteinen, die nur lesen oder schreiben, kombiniert diese Subapp die Funktionen *Einlesen*, *Persistieren* und *Rückschreiben*. Andere Implementierungen könnten nur einen Teil dieser Aufgaben abdecken:

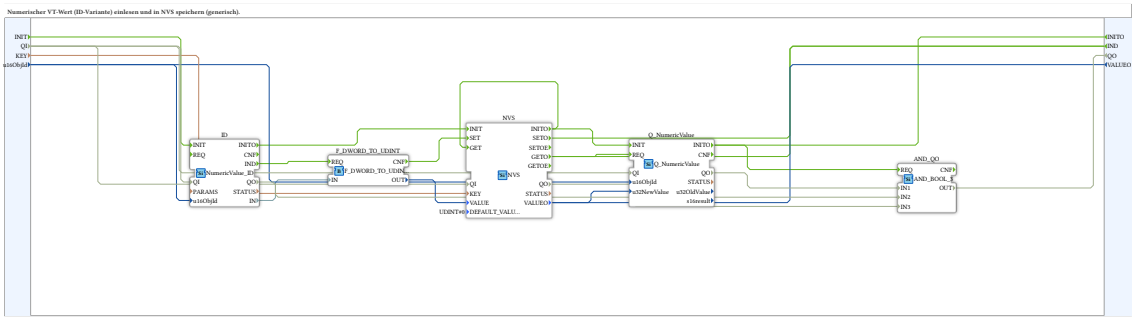
- **NumericValue_ID ohne Speicherung:** Liest nur den Wert und gibt ihn aus, verliert aber den Zustand bei Neustart.
- **INI-Baustein allein:** Erlaubt nur den Zugriff auf die INI-Datei, ohne direkte Verbindung zum ISOBUS-Gerät.
- **Manuelle Verkettung:** Die gleiche Funktionalität ließe sich auch durch separate FB-Instanzen realisieren, wäre aber weniger kompakt und schwerer wiederverwendbar.

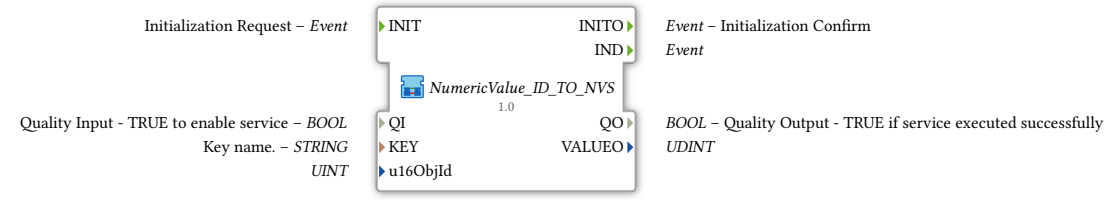
Diese Subapp bündelt die notwendigen Schritte in einer kompakten, wiederverwendbaren Einheit und reduziert so den Entwicklungsaufwand.

Fazit

Der Funktionsblock `NumericValue_ID_TO_INI` bietet eine robuste Lösung zur persistenten Speicherung von numerischen Werten aus ISOBUS-Geräten. Durch die Kombination aus Hardwarezugriff, Konvertierung, INI-Speicherung und automatischem Rückschreiben beim Start erfüllt er eine häufige Anforderung in industriellen und landwirtschaftlichen Anwendungen. Die generische Schnittstelle mit `KEY`, `SECTION` und `u16ObjId` erlaubt eine flexible Wiederverwendung ohne Änderungen an der internen Struktur. Die Dokumentation der Ereignis- und Datenflüsse zeigt eine klare und deterministische Verarbeitung, die in verteilten Steuerungssystemen einfach integriert werden kann. Daher ist dieser Baustein eine sinnvolle Ergänzung für jede Applikation, die eine dauerhafte Speicherung von Geräteparametern benötigt.

NumericValue_ID_TO_NVS





• – ○ • – ○ • – ○ •

Einleitung

Der Baustein `NumericValue_ID_TO_NVS` ist eine Subapplikation (SubApp) zur generischen Speicherung eines numerischen Werts, der über die ID-Variante des ISOBUS-Numeric-Value-Protokolls eingelesen wird, in den nichtflüchtigen Speicher (NVS) eines ESP32. Er kombiniert die Funktionen des Einlesens, Konvertierens, Speicherns und Wiederausgebens. Die Applikation ist modular aufgebaut und kann in

verschiedenen Kontexten wiederverwendet werden, in denen ein numerischer Wert über einen Schlüssel persistent gespeichert und bei Bedarf geladen werden soll.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung; geht an `ID.INIT` und startet die Initialisierungskette

Ereignis-Ausgänge

Ereignis	Typ	Kurzbeschreibung
`IND`	Event	Zeigt an, dass ein Wert aus dem NVS geladen oder gespeichert wurde.
`INITO`	EInit	Initialisierung abgeschlossen; kommt von `Q_NumericValue.INITO`, dem Ende der Kette `ID` ⟶ `NVS` ⟶ `Q_NumericValue`.

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung
`QI`	`BOOL`	`TRUE`	Quality Input – TRUE, um den Dienst freizugeben; wird an `ID.QI`, `NVS.QI` und `Q_NumericValue.QI` durchgereicht
`KEY`	STRING	–	Schlüsselname unter dem der Wert im NVS gespeichert wird.
`u16ObjId`	UINT	`ID_NULL`	Objekt-ID des numerischen Werts am ISOBUS-Terminal.

Daten-Ausgänge

Name	Typ	Beschreibung
`QO`	`BOOL`	Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde; AND aus `Q_NumericValue.QO`, `ID.QO` und `NVS.QO`
`VALUEO`	UDINT	Der aktuell gelesene bzw. gespeicherte numerische Wert.

Adapter

Keine.

Funktionsweise

Der Baustein arbeitet in zwei Hauptphasen:

1. **Initialisierung**
   Nach dem Start wird automatisch der im NVS gespeicherte Wert unter dem angegebenen `KEY` geladen. Dazu wird das intern verbundene `NVS`-FB mit dem Ereignis `GET` angestoßen. Das Ergebnis (`VALUEO`) wird über den Ausgang `VALUEO` der SubApp bereitgestellt und gleichzeitig an einen internen `Q_NumericValue`-FB übergeben, der den Wert an das ISOBUS-Terminal übermitteln kann.
2. **Wertaktualisierung über ISOBUS**
   Wenn ein neuer numerischer Wert über die ID-Variante am Terminal anliegt, löst das `NumericValue_ID`-FB das Ereignis `IND` aus. Dieses Ereignis wird über die interne Kette
   `ID.IND → F_DWORD_TO_UDINT.REQ → F_DWORD_TO_UDINT.CNF → NVS.SET`
   abgearbeitet. Der vom Terminal gelesene DWORD-Wert wird in einen `UDINT` konvertiert und als neuer Wert mit dem zugehörigen `KEY` im NVS gespeichert. Nach erfolgreichem Speichern wird das Ereignis `SETO` ausgelöst, das zum Ausgang `IND` der SubApp führt.

Parallel dazu wird bei jeder Speicher- oder Leseoperation auch der interne `Q_NumericValue`-FB mit dem aktuellen Wert und der Objekt-ID versorgt, sodass der Wert auch am Terminal angezeigt bzw. aktualisiert wird.

Die Datenflüsse sind in der SubApp über direkte Verbindungen realisiert, wobei die externen Anschlüsse der SubApp (`KEY`, `u16ObjId`, `VALUEO`) direkt mit den internen Bausteinen verdrahtet sind.

Initialisierung und Quality-Anschlüsse: `INIT` ⟶ `ID.INIT`, `ID.INITO` ⟶ `NVS.INIT`, `NVS.INITO` ⟶ `Q_NumericValue.INIT`, `Q_NumericValue.INITO` ⟶ `INITO` (und ⟶ `AND_QO.REQ`); zusätzlich löst

NVS.INIT0 weiterhin NVS.GET aus. QI (Dateneingang, BOOL, Standard TRUE) wird an ID.QI, NVS.QI und Q_NumericValue.QI weitergereicht. Der Ausgang QO (BOOL) ist der Ausgang der Instanz AND_QO (iec61131::booleanOperators::AND_BOOL_3), die Q_NumericValue.QO, ID.QO und NVS.QO verknüpft. Damit lässt sich der Dienst von außen freigeben und sein Erfolg abfragen.

Technische Besonderheiten

- **Persistenz:** Der Wert wird über den logiBUS::storage::esp32_nvs::NVS-FB dauerhaft im Flash-Speicher des ESP32 abgelegt.
- **Generische ID-Auswahl:** Die Objekt-ID (u16ObjId) ist konfigurierbar, sodass verschiedene numerische Werte des ISOBUS-Terminals verwendet werden können.
- **Typkonvertierung:** Der vom ISOBUS gelieferte DWORD-Wert wird durch F_DWORD_TO_UDINT in einen UDINT umgewandelt, passend zur internen Darstellung.
- **Automatisches Laden:** Beim Start wird der gespeicherte Wert ohne externe Anforderung geladen und über VALUE0 bereitgestellt.
- **Interne ISOBUS-Aktualisierung:** Der Q_NumericValue-FB dient dazu, den Wert am Terminal zu setzen bzw. zu aktualisieren, ohne dass ein separater Ausgang erforderlich ist.

Zustandsübersicht

Die SubApp kennt keine expliziten Zustände, da sie ereignisgesteuert arbeitet. Die relevanten Abläufe lassen sich wie folgt beschreiben:

- **Initialisierungszustand:** Nach dem Start wird der Wert aus dem NVS geladen (INIT0 → GET). Nach Abschluss werden GET0 sowie das externe Ereignis IND ausgelöst.
- **Speicherzustand:** Bei einem ankommenden Wert vom Terminal wird die Kette ID.IND → Konvertierung → NVS.SET durchlaufen. Nach erfolgreichem Speichern erscheint SET0 und wiederum IND.
- **Lesenzustand:** Ein explizites Laden (durch externe Ereignisse) ist nicht vorgesehen; das Lesen erfolgt nur bei Initialisierung. Allerdings kann der Wert über VALUE0 jederzeit abgelesen werden.

Die Ereignisse IND signalisieren also immer den Abschluss einer Speicher- oder Leseoperation.

Anwendungsszenarien

- **Konfigurationsspeicherung:** Geräteparameter, die über das ISOBUS-Terminal eingestellt und beim nächsten Start wiederhergestellt werden sollen.
- **Datenprotokollierung:** Speicherung von Messwerten oder Zählerständen, die dauerhaft erhalten bleiben müssen.
- **Firmware-Update-Einstellungen:** Persistente Speicherung von Benutzereinstellungen in landwirtschaftlichen Maschinen.
- **Wiederverwendbare SubApp:** Einbindung in größere Steuerungsanwendungen, die einen numerischen ISOBUS-Wert dauerhaft speichern müssen.

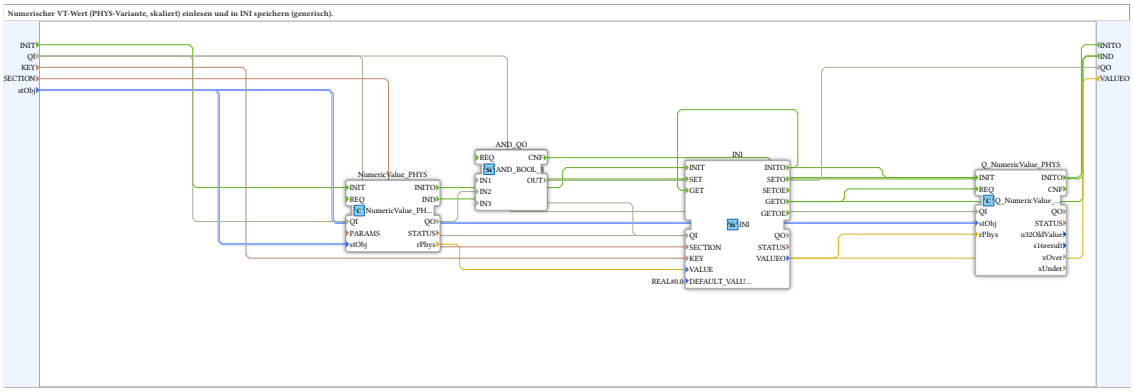
Vergleich mit ähnlichen Bausteinen

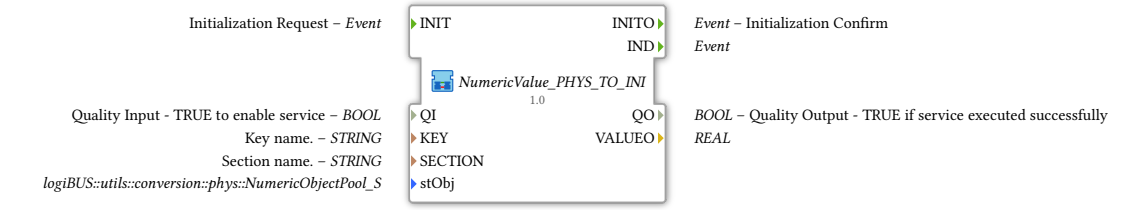
Im Gegensatz zu reinen Kommunkations-Bausteinen (z.B. NumericValue_ID allein) oder reinen Speicher-Bausteinen (z.B. NVS ohne ISOBUS-Anbindung) kombiniert NumericValue_ID_TO_NVS beide Funktionen in einer einzigen SubApp. Dadurch entfällt die manuelle Verdrahtung zwischen ISOBUS-Reader, Konverter und NVS. Ähnliche Bausteine könnten ohne ID-Variante (z.B. mit Index) oder ohne Konvertierung (direkter DWORD-Typ) arbeiten, bieten jedoch weniger Flexibilität. Diese SubApp ist durch die parametrierbare Objekt-ID und den konfigurierbaren Schlüssel besonders universell einsetzbar.

Fazit

NumericValue_ID_TO_NVS stellt eine robuste und wiederverwendbare Lösung dar, um numerische ISOBUS-Werte direkt und dauerhaft im NVS eines ESP32 zu speichern. Durch die klare Trennung von Einlesen, Konvertieren und Speichern sowie die automatische Initialisierung ist sie einfach in übergeordnete Steuerungen integrierbar. Die interne Aktualisierung des ISOBUS-Terminals über Q_NumericValue macht den Baustein darüber hinaus zu einem vollständigen Bindeglied zwischen Terminal und persistentem Speicher.

NumericValue_PHYS_TO_INI





• – ○ • – ○ • – ○ •

Einleitung

Die SubApp **NumericValue_PHYS_TO_INI** dient zum Einlesen eines numerischen Werts aus einer physikalischen Größe (z. B. einem Messwert mit Skalierung und Offset) und zum Speichern dieses Werts in eine INI-Datei. Sie kombiniert dabei die Funktionen eines physikalischen Wertaufnehmers (`NumericValue_PHYS`) mit einem INI-Speicher (`INI`) und einer Qualitätsüberwachung (`Q_NumericValue_PHYS`). Die SubApp ist generisch aufgebaut und ermöglicht die Wiederverwendung in verschiedenen Anwendungen, bei denen ein physikalischer Wert persistiert werden soll.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung. Die weitere interne Verarbeitung wird durch die eingebetteten Funktionsblöcke angestoßen.

Ereignis-Ausgänge

Name	Typ	Kommentar
`INITO`	EInit	Initialisierungsbestätigung (von `Q_NumericValue_PHYS.INITO`).
`IND`	Event	Wird ausgelöst, nachdem der Wert in die INI-Datei geschrieben oder aus dieser gelesen wurde.

Daten-Eingänge

Name	Datentyp	Kommentar
`QI`	`BOOL`	Quality Input – TRUE, um den Dienst freizugeben; wird an `Q_NumericValue_PHYS.QI`, `NumericValue_PHYS.QI` und `INI.QI` verteilt
`KEY`	`STRING`	Schlüsselname für den INI-Eintrag.
`SECTION`	`STRING`	Abschnittsname in der INI-Datei.
`stObj`	`logiBUS::utils::conversion::phys::NumericObjectPool_S`	Struktur mit physikalischen Parametern (z. B. Objekt-ID, Skalierung, Offset, De-

		zimalstellen). Initialwert: `(u16ObjId := ID_NULL, r32Scale := 1.0, i32Offset := 0, u8Decimals := 0)`.

Daten-Ausgänge

Name	Datentyp	Kommentar
QO	BOOL	Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde; UND aus `Q_NumericValue_PHYS.QO`, `NumericValue_PHYS.QO` und `INI.QO`
VALUEO	REAL	Ausgangswert; entspricht dem Wert, der aus der INI-Datei zurückgelesen wird.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Die SubApp verwendet vier interne Funktionsblöcke:

1. **`NumericValue_PHYS`** (Typ `isobus::UT::io::NumericValue::NumericValue_PHYS`):
   Liest den aktuellen physikalischen Wert (skaliert) und stellt ihn als REAL-Wert an seinem Ausgang `rPhys` bereit. Der Eingang `stObj` definiert die Parameter der physikalischen Umrechnung.
2. **`INI`** (Typ `eclipse4diac::storage::INI`):
   Dient zum Lesen und Schreiben von Werten in einer INI-Datei. Über die Eingänge `KEY` und `SECTION` wird der Zielort festgelegt. Der Eingang `VALUE` erhält den zu speichernden Wert, während der Ausgang `VALUEO` den aus der Datei gelesenen Wert liefert. Der Parameter `DEFAULT_VALUE` ist auf `REAL#0.0` gesetzt.
3. **`Q_NumericValue_PHYS`** (Typ `isobus::UT::Q::Q_NumericValue_PHYS`):
   Ein Qualitätsüberwachungsblock, der die Konsistenz des physikalischen Wertes prüft. Er erhält die gleichen physikalischen Parameter (`stObj`) sowie den aus der INI gelesenen Wert (`rPhys`) und wird über das Ereignis `REQ` aktiviert. Sein Qualitätseingang `QI` wird von außen über den SubApp-Eingang `QI` versorgt, sein Ausgang `QO` geht in `AND_QO` ein.
4. **`AND_QO`** (Typ `iec61131::booleanOperators::AND_BOOL_3`):
   Verknüpft `Q_NumericValue_PHYS.QO` (`IN1`), `NumericValue_PHYS.QO` (`IN2`) und `INI.QO` (`IN3`) zum SubApp-Ausgang `QO`.

Ablauf:

- Sobald `NumericValue_PHYS` einen neuen Wert erfasst, sendet es das Ereignis `IND` an den Eingang `SET` des INI-Blocks.
- Dadurch wird der aktuelle `rPhys`-Wert in die INI-Datei geschrieben. Nach erfolgreichem Schreiben gibt der INI-Block das Ereignis `SETO` aus.
- `SETO` triggert das Ereignis `IND` der SubApp, wodurch nach außen signalisiert wird, dass der Wert gespeichert wurde.
- Parallel dazu wird durch das Ereignis `INITO` (das bei der Initialisierung des INI-Blocks automatisch ausgelöst wird) der Eingang `GET` angestoßen. Dadurch wird der gespeicherte Wert aus der INI-Datei zurückgelesen.
- Nach dem Lesen erzeugt der INI-Block das Ereignis `GETO`. Dieses triggert sowohl den Qualitätsblock `Q_NumericValue_PHYS` (über `REQ`) als auch erneut das SubApp-Ereignis `IND`.
- Der aus der INI gelesene Wert steht am Ausgang `VALUEO` zur Verfügung und wird außerdem an den Qualitätsblock übergeben.

Die doppelte Ausgabe von `IND` (einmal nach `SETO` und einmal nach `GETO`) ermöglicht es dem Anwender, beide Schritte (Schreiben und Lesen) separat zu quittieren, sofern die Verbindungen entsprechend sichtbar geschaltet sind.

Quality-Anschlüsse: `QI` (Dateneingang, BOOL, Standard `TRUE`) wird an `Q_NumericValue_PHYS.QI`, `NumericValue_PHYS.QI` und `INI.QI` verteilt; der Ausgang `QO` (BOOL) ist das UND der `QO`-Ausgänge dieser drei Bausteine (`AND_QO`). Damit lässt sich der Dienst von außen freigeben und sein Erfolg abfragen.

Initialisierung: `INIT` ⟶ `NumericValue_PHYS.INIT`; `NumericValue_PHYS.INITO` ⟶ `INI.INIT`; `INI.INITO` ⟶ `Q_NumericValue_PHYS.INIT` (und intern ⟶ `INI.GET`); `Q_NumericValue_PHYS.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`.

Technische Besonderheiten

- **Generische Konfiguration:** Die SubApp ist vollständig über die Eingänge `KEY`, `SECTION` und `stObj` parametrierbar und kann dadurch für verschiedene physikalische Größen und INI-Abschnitte eingesetzt werden.
- **Integrierte Qualitätsprüfung:** Der Block `Q_NumericValue_PHYS` bietet eine zusätzliche Kontrollmöglichkeit, deren Qualitätsanschlüsse nach außen geführt sind: `QI` der SubApp wirkt auf `Q_NumericValue_PHYS.QI` (sowie `NumericValue_PHYS.QI` und `INI.QI`), dessen `QO` geht gemeinsam mit den `QO` der beiden anderen Bausteine über `AND_QO` in das `QO` der SubApp ein.
- **Initialwert des INI-Blocks:** Der `DEFAULT_VALUE` ist auf `REAL#0.0` gesetzt, was sicherstellt, dass beim Lesen eines nicht existierenden Eintrags ein definierter Wert zurückgegeben wird.
- **Verwendung von Strukturtypen:** Der Eingang `stObj` verwendet den komplexen Datentyp `NumericObjectPool_S`, der eine kompakte Bündelung aller physikalischen Parameter ermöglicht.
- **Interne Ereignisschleife:** Durch die Verbindung `INITO` ⟶ `GET` wird direkt nach der Initialisierung ein Lesevorgang angestoßen, sodass der INI-Wert sofort nach dem Start verfügbar ist.

Zustandsübersicht

Die SubApp besitzt keinen expliziten Zustandsautomaten auf oberster Ebene. Ihr Verhalten wird durch die internen Funktionsblöcke gesteuert. Die wesentlichen Zustände lassen sich anhand der Ereignisse ableiten:

- **Initialisierung:** Beim ersten Aktivieren des INI-Blocks wird `INITO` gesetzt, was den Lesevorgang auslöst.
- **Schreiben:** Das Ereignis `SET` wird durch `NumericValue_PHYS.IND` ausgelöst. Nach Abschluss des Schreibvorgangs wird `SETO` erzeugt.
- **Lesen:** Das Ereignis `GET` wird durch `INITO` (initial) oder durch externe Anforderungen ausgelöst. Nach Abschluss des Lesevorgangs wird `GETO` erzeugt.
- **Warten:** Nach Ausgabe von `IND` wartet die SubApp auf das nächste Ereignis von `NumericValue_PHYS`.

Eine feinere Unterteilung könnte durch die Qualitätsüberwachung des Blocks `Q_NumericValue_PHYS` vorgenommen werden, deren `QO` über das UND-Glied `AND_QO` in den SubApp-Ausgang `QO` eingeht.

Anwendungsszenarien

- **Persistente Speicherung von Messwerten:** Ein physikalischer Sensor (z. B. Temperatur, Druck) liefert über `NumericValue_PHYS` einen skalierten Wert. Dieser wird in eine INI-Datei geschrieben, sodass er nach einem Neustart des Systems wieder verfügbar ist.
- **Parameterverwaltung:** Die SubApp kann verwendet werden, um aktuelle Geräteparameter in einer Konfigurationsdatei abzulegen und bei Bedarf wieder einzulesen.
- **Abgleich zwischen Live-Wert und gespeichertem Wert:** Durch die gleichzeitige Ausgabe von `VALUEO` (gelesener Wert) und den internen Verbindungen kann ein Vergleich zwischen dem aktuellen Messwert und dem zuletzt gespeicherten Wert durchgeführt werden.
- **Wiederverwendbare Baustein-Bibliothek:** Durch die Auslagerung aus einer bestehenden Übung (`Uebung_012e_sub`) kann der Baustein in verschiedenen Projekten eingesetzt werden, ohne die interne Logik neu zu implementieren.

Vergleich mit ähnlichen Bausteinen

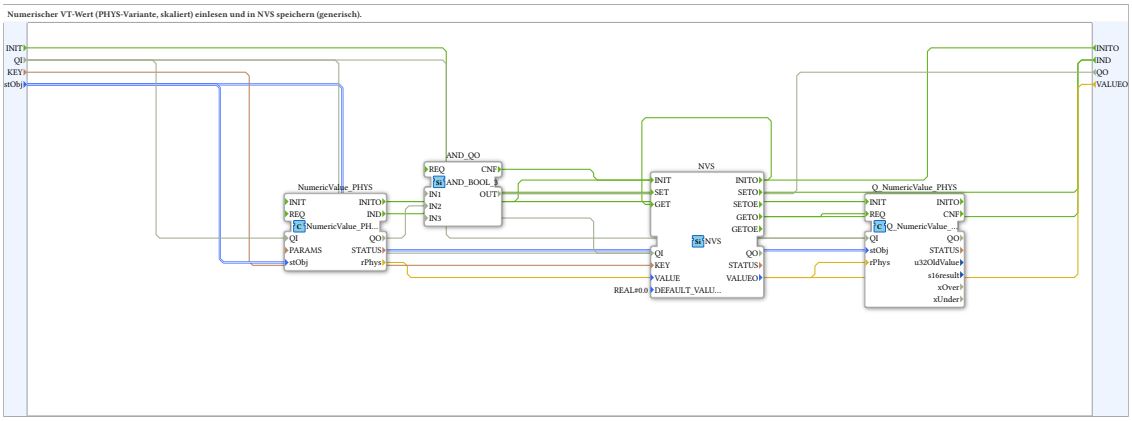
Im Vergleich zu einfachen INI-Schreib-/Lesebausteinen bietet `NumericValue_PHYS_TO_INI` eine integrierte physikalische Umrechnung und eine Qualitätsüberwachung. Während ein Standard-`INI`-Block nur die reine Speicherung übernimmt, übernimmt diese SubApp zusätzlich die Umrechnung eines rohen Messwerts in eine physikalische Größe (Skalierung, Offset) und die anschließende Persistierung. Der Qualitätsblock `Q_NumericValue_PHYS` hebt die SubApp von reinen Speicherlösungen ab und ermöglicht eine erweiterte Diagnose, über die Anschlüsse `QI` und `QO`, die an der SubApp zur Verfügung stehen.

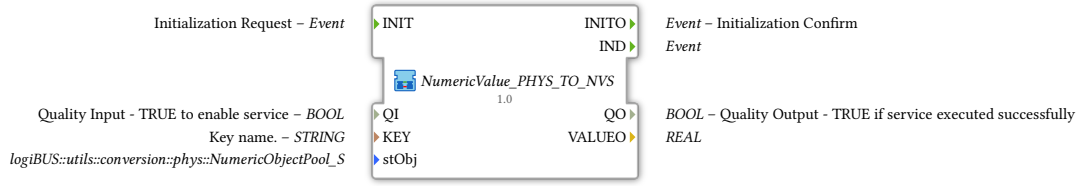
Fazit

Die SubApp **NumericValue_PHYS_TO_INI** stellt eine flexible und wiederverwendbare Lösung dar, um physikalische Messwerte dauerhaft in einer INI-Datei zu speichern und gleichzeitig den aktuellen sowie den gespeicherten Wert bereitzustellen. Durch die klare Trennung von Messwerterfassung, Speicherung

und optionaler Qualitätsprüfung eignet sie sich für verschiedenste Automatisierungs- und Steuerungsaufgaben. Die generische Parametrierung über `KEY`, `SECTION` und `stObj` macht sie zu einem vielseitigen Baustein für modulare Projekte.

NumericValue_PHYS_TO_NVS





• – ○ • – ○ • – ○ •

Einleitung

Die SubApp `NumericValue_PHYS_TO_NVS` bietet eine generische Lösung zum Einlesen eines numerischen Werts in der physikalischen (PHYS) Variante, der mithilfe einer Skalierung in einen technischen Wert umgerechnet wird. Dieser Wert wird anschließend persistent im NVS (Non-Volatile Storage) eines ESP32 gespeichert. Die SubApp kapselt dabei die vollständige Logik für Speichern, Laden und Umrechnen und stellt eine einfache Schnittstelle zur Verfügung, die sowohl das Schreiben als auch das Lesen des Wertes ermöglicht.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsanforderung („Initialization Request“). Wird an `Q_NumericValue_PHYS.INIT` durchgereicht und läuft dann über `NumericValue_PHYS` und `NVS`.

Ereignis-Ausgänge

- **INITO** (EInit): Initialisierungsbestätigung („Initialization Confirm“). Kommt von `NVS.INITO` (Ende der Initialisierungskette).
- **IND** (Event): Wird ausgelöst, wenn ein Speichervorgang oder ein Lesevorgang abgeschlossen ist. Konkret wird dieser Ausgang über `NVS.SETO` (nach erfolgreichem Speichern) und über `NVS.GETO` (nach erfolgreichem Laden) aktiviert.

Daten-Eingänge

- **QI** (BOOL): Quality Input – TRUE, um den Dienst freizugeben; wird an `Q_NumericValue_PHYS.QI`, `NumericValue_PHYS.QI` und `NVS.QI` verteilt
- **KEY** (STRING): Schlüsselname, unter dem der Wert im NVS abgelegt wird.
- **stObj** (Typ: `logiBUS::utils::conversion::phys::NumericObjectPool_S`): Struktur, die die Parameter für die physikalische Umrechnung enthält:
  - `u16ObjId` (Objektidentifikation, Standard: `ID_NULL`)
  - `r32Scale` (Skalierungsfaktor, Standard: `1.0`)
  - `i32Offset` (Offset, Standard: `0`)
  - `u8Decimals` (Anzahl der Nachkommastellen, Standard: `0`)

Daten-Ausgänge

- **QO** (BOOL): Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde; UND-Verknüpfung (`AND_QO`, `AND_BOOL_3`) von `Q_NumericValue_PHYS.QO`, `NumericValue_PHYS.QO` und `NVS.QO`
- **VALUEO** (REAL): Der aktuell geladene oder gespeicherte Wert als `REAL`. Dieser Wert entspricht der intern abgelegten, skalierten Größe.

Adapter

Keine vorhanden.

Funktionsweise

Die SubApp besteht aus drei internen Funktionsblöcken und einem UND-Baustein für die Qualität:

- **`NumericValue_PHYS`** – liest einen physikalischen Wert (z. B. von einem Sensor) und konvertiert ihn in einen skalierten Prozesswert.
- **`NVS`** – verwaltet den nichtflüchtigen Speicher (ESP32 NVS) und ermöglicht das Speichern (SET) und Laden (GET) von Werten.
- **`Q_NumericValue_PHYS`** – dient zur Rückkonvertierung des gespeicherten Wertes in die ursprüngliche physikalische Einheit (erhält `INIT` und `QI` von außen; sein `INITO` startet `NumericValue_PHYS`, sein `QO` geht an `AND_QO.IN1`).
- **`AND_QO`** (`iec61131::booleanOperators::AND_BOOL_3`) – verknüpft `Q_NumericValue_PHYS.QO` (`IN1`), `NumericValue_PHYS.QO` (`IN2`) und `NVS.QO` (`IN3`) zu `QO`.

Der Ablauf ist wie folgt:

1. **Initialisierung**: Das SubApp-Ereignis `INIT` wird an `Q_NumericValue_PHYS.INIT` weitergereicht; `Q_NumericValue_PHYS.INITO` ⟶ `NumericValue_PHYS.INIT`; `NumericValue_PHYS.INITO` ⟶ `NVS.INIT`; `NVS.INITO` erscheint als `INITO` der SubApp und stößt zusätzlich `AND_QO.REQ` an. Außerdem löst die Initialisierung des NVS-Bausteins (`NVS.INITO`) automatisch ein `GET` aus, um den unter `KEY` gespeicherten Wert zu laden.
2. **Laden**: Der geladene Wert (`NVS.VALUEO`) wird direkt an den Ausgang `VALUEO` gelegt und parallel an `Q_NumericValue_PHYS` übergeben, um ggf. eine Rückumrechnung vorzunehmen.
3. **Speichern**: Wird vom Baustein `NumericValue_PHYS` ein neuer Wert erzeugt (Ausgang `IND`), so wird dessen skaliertes Ergebnis (`rPhys`) an `NVS.VALUE` übergeben und mit `SET` im NVS gespeichert. Nach erfolgreichem Speichern signalisiert `SETO` den Abschluss.
4. **Ausgabe**: Sowohl nach dem Laden als auch nach dem Speichern wird der Ausgang `IND` der SubApp aktiviert. Dadurch kann eine übergeordnete Steuerung den Vorgang synchron verfolgen.

Die SubApp unterstützt damit sowohl das Schreiben als auch das Lesen eines persistenten, skalierten Werts.

Quality-Anschlüsse: `QI` (Dateneingang, BOOL, Standard `TRUE`) wird an `Q_NumericValue_PHYS.QI`, `NumericValue_PHYS.QI` und `NVS.QI` verteilt; der Ausgang `QO` (BOOL) ist die UND-Verknüpfung (`AND_QO`) der drei `QO`-Ausgänge. Damit lässt sich der Dienst von außen freigeben und sein Erfolg abfragen.

Technische Besonderheiten

- **Generische Konfiguration**: Durch das Objekt `stObj` kann die SubApp für verschiedene physikalische Messgrößen (z. B. Temperatur, Druck) konfiguriert werden, ohne die Logik zu ändern.
- **Eingebettete Systeme**: Der Einsatz des NVS-Bausteins ist speziell für ESP32-Mikrocontroller ausgelegt.
- **Skalierung**: Die Werte werden mit `r32Scale` und `i32Offset` linear skaliert; die Anzahl der Nachkommastellen (`u8Decimals`) ist für darstellende Zwecke vorgesehen.
- **Bibliotheksabhängigkeiten**: Die SubApp greift auf die Bibliotheken `isobus` und `logiBUS` zu, die in der CompilerInfo als Imports deklariert sind.

Zustandsübersicht

Die SubApp selbst besitzt keinen eigenen Zustandsautomaten, sondern delegiert die Zustandslogik an den internen `NVS`-Baustein. Typische Zustände des NVS sind:

- **INIT** – Initialisierung des Speichers
- **IDLE** – Warten auf Anweisungen
- **SET** – Speichern eines Wertes
- **GET** – Lesen eines Wertes

Die SubApp signalisiert den Abschluss dieser Operationen über das Ereignis `IND`.

Anwendungsszenarien

- **Persistente Sensorwerte**: Ein Temperatursensorwert wird periodisch gelesen und im NVS gespeichert, sodass nach einem Neustart der letzte gültige Wert verfügbar ist.
- **Kalibrierdaten**: Ein Kalibrierfaktor oder Offset wird in NVS abgelegt und bei jedem Start geladen.
- **Parametereinstellungen**: Benutzerdefinierte Einstellungen (z. B. Sollwerte) werden gespeichert und wieder ausgelesen.

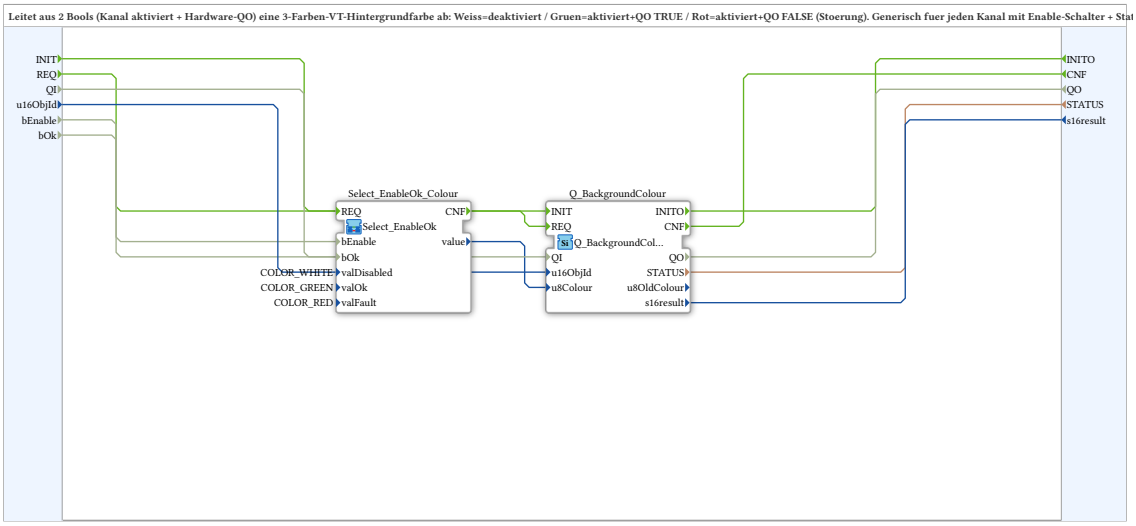
Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen NVS-Bausteinen, die nur Rohdaten speichern, bietet diese SubApp eine integrierte Skalierung und Rückumrechnung. Sie vereint mehrere Funktionen (Lesen, Speichern, Konvertieren) in einem Baustein und reduziert dadurch den Verdrahtungsaufwand. Gegenüber einem direkten Zugriff auf den NVS wird eine höhere Abstraktionsebene erreicht, die Fehlerquellen minimiert.

Fazit

Die SubApp `NumericValue_PHYS_TO_NVS` ist eine modular aufgebaute, wiederverwendbare Komponente für eingebettete Systeme, die einen skalierten Messwert dauerhaft speichert und bei Bedarf wieder bereitstellt. Sie kombiniert die Vorteile einer flexiblen Konfiguration über das `stObj` mit der einfachen Handhabung einer einheitlichen Schnittstelle und ist somit ideal für Anwendungen, die persistente Wertespeicherung erfordern.

Q_BackgroundColour_EnableOk

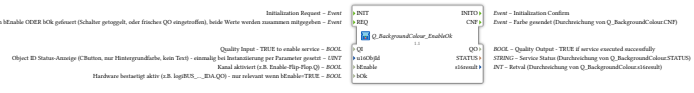


• – ○ • – ○ • – ○ •

Einleitung

`Q_BackgroundColour_EnableOk` leitet aus 2 Bools (Kanal aktiviert + Hardware-QO) eine 3-Farben-VT-Hintergrundfarbe ab: Weiss = deaktiviert, Gruen = aktiviert und QO TRUE, Rot = aktiviert und QO FALSE (Stoerung). Generisch fuer jeden Kanal mit Enable-Schalter + Status-LED (PWM, PI, ...). Die reine Auswahllogik ist mittlerweile in `Select_EnableOk` ausgelagert; dieser Baustein instanziiert sie nur noch mit den 3 Farben als Parameter und haengt `Q_BackgroundColour` dran.

Quality-Anschlüsse: Der Baustein besitzt zusätzlich das Ereignis `INIT` (Eingang) und `INITO` (Ausgang) sowie den Dateneingang `QI` (BOOL, Standard `TRUE`, „Quality Input – TRUE, um den Dienst freizugeben“) und den Datenausgang `QO` (BOOL, „Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde“). `INIT` wird an `Q_BackgroundColour.INIT` weitergereicht, `Q_BackgroundColour.INITO` an `INITO`; `QI` ist mit `Q_BackgroundColour.QI` und `Q_BackgroundColour.QO` mit `QO` verbunden.



Verwendete Funktionsbausteine (FBs)

- **Select_EnableOk_Colour** (SubApp, Typ `MyLib::sys::Select_EnableOk`): `valDisabled=COLOR_WHITE`, `valOk=COLOR_GREEN`, `valFault=COLOR_RED`.
- **Q_BackgroundColour** (`isobus::UT::Q::Q_BackgroundColour`): schreibt die berechnete Farbe auf `u16ObjId`.

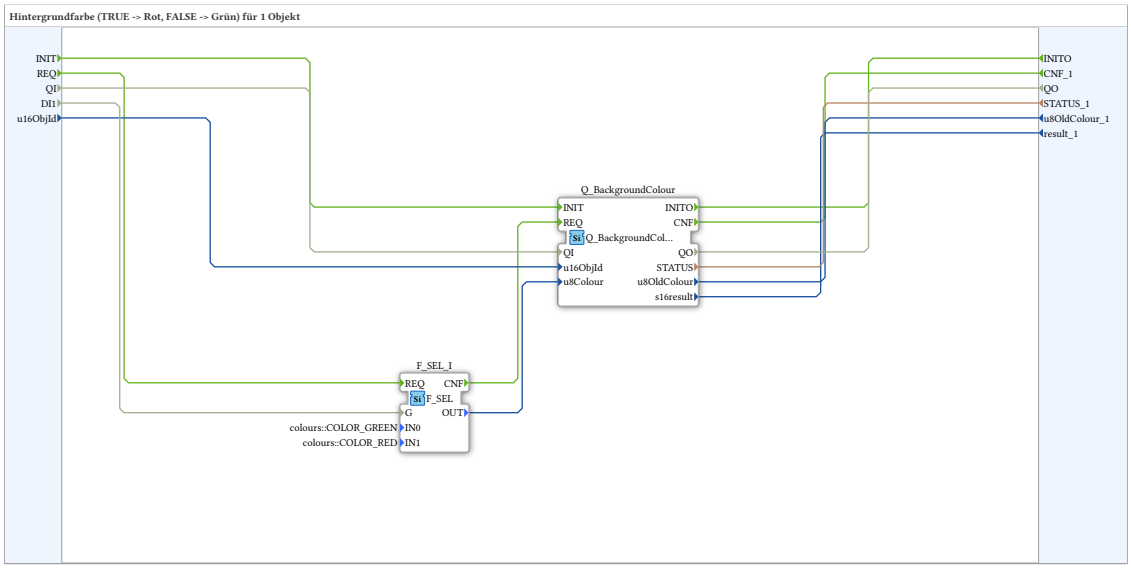
Zusammenfassung

Fertig parametrierte Farblogik (Weiss/Gruen/Rot) fuer Enable+Ok-Statusanzeigen, aufgebaut auf der generischen `Select_EnableOk`-Auswahl.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedGreenBackground1



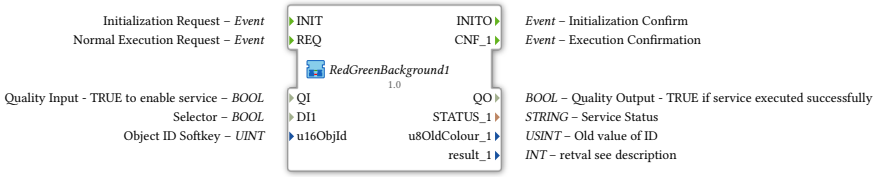
• — ○ • — ○ • — ○ •

Einleitung

`RedGreenBackground1` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Grün**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den Eingang `u16ObjId` übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Quality-Anschlüsse: Der Baustein besitzt zusätzlich das Ereignis `INIT` (Eingang) und `INITO` (Ausgang) sowie den Dateneingang `QI` (BOOL, Standard `TRUE`, „Quality Input – TRUE, um den Dienst freizugeben“) und den Datenausgang `QO` (BOOL, „Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde“). `INIT` wird an `Q_BackgroundColour.INIT` weitergereicht, `Q_BackgroundColour.INITO` an `INITO`; `QI` ist mit `Q_BackgroundColour.QI` und `Q_BackgroundColour.QO` mit `QO` verbunden.



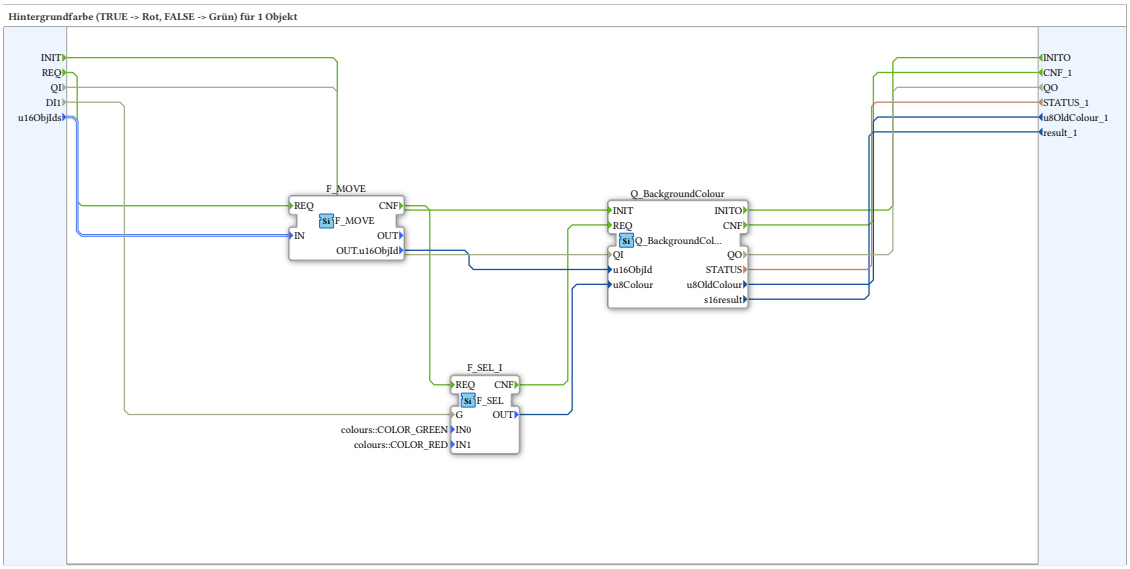
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Grün, 1 Objekt, BOOL-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedGreenBackground1S



• — ○ • — ○ • — ○ •

Einleitung

`RedGreenBackground1S` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Grün**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Quality-Anschlüsse: Der Baustein besitzt zusätzlich das Ereignis `INIT` (Eingang) und `INITO` (Ausgang) sowie den Dateneingang `QI` (BOOL, Standard `TRUE`, „Quality Input – TRUE, um den Dienst freizugeben“) und den Datenausgang `QO` (BOOL, „Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde“). `INIT` wird an `Q_BackgroundColour.INIT` weitergereicht, `Q_BackgroundColour.INITO` an `INITO`; `QI` ist mit `Q_BackgroundColour.QI` und `Q_BackgroundColour.QO` mit `QO` verbunden.

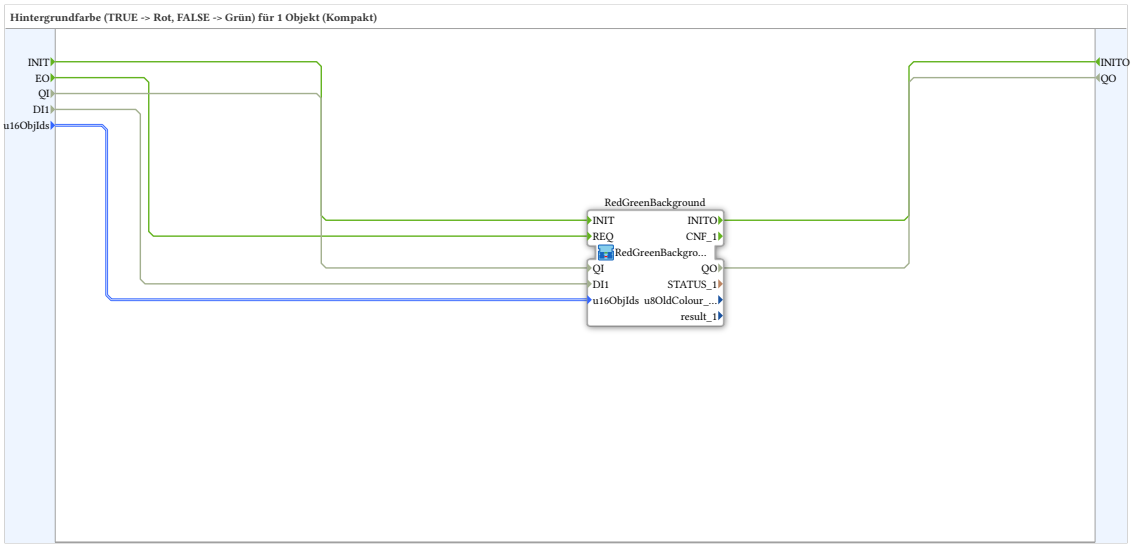
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Grün, 1 Objekt, BOOL-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedGreenBackground1SC



• – ◦ • – ◦ • – ◦ •

Einleitung

`RedGreenBackground1SC` ist ein **Kompakt-Wrapper** um `RedGreenBackground1S` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

Initialisierung und Status (INIT/INITO, QI/QO)

- **Ereignis-Eingang** `INIT` (`EInit`, Initialization Request) und **Ereignis-Ausgang** `INITO` (`EInit`, Initialization Confirm)
- **Daten-Eingang** `QI` (`BOOL`, Vorgabe `TRUE`, Quality Input) wird an `RedGreenBackground.QI` weitergegeben
- **Daten-Ausgang** `QO` (`BOOL`, Quality Output): durchgereicht von `RedGreenBackground.QO`
- **Initialisierungskette**: `INIT` ⟶ `RedGreenBackground.INIT`, `RedGreenBackground.INITO` ⟶ `INITO`

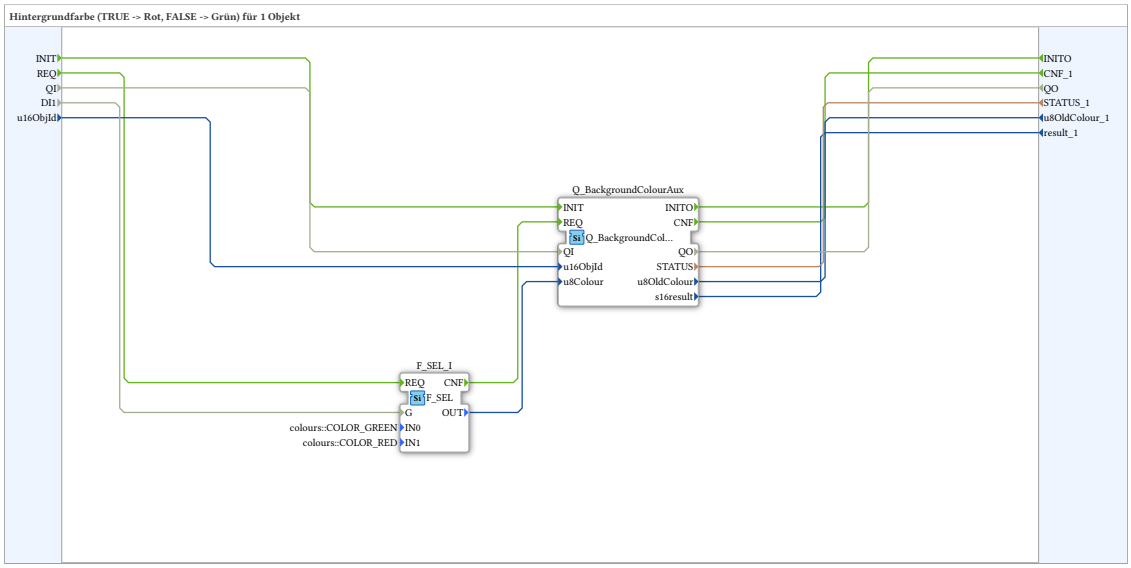
Zusammenfassung

Reine Weiterleitung an `RedGreenBackground1S` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedGreenBackground1_aux



• — ○ • — ○ • — ○ •

Einleitung

`RedGreenBackground1_aux` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Grün**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den Eingang `u16ObjId` übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Quality-Anschlüsse: Der Baustein besitzt zusätzlich das Ereignis `INIT` (Eingang) und `INITO` (Ausgang) sowie den Dateneingang `QI` (BOOL, Standard `TRUE`, „Quality Input – TRUE, um den Dienst freizugeben“) und den Datenausgang `QO` (BOOL, „Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde“). `INIT` wird an `Q_BackgroundColourAux.INIT` weitergereicht, `Q_BackgroundColourAux.INITO` an `INITO`; `QI` ist mit `Q_BackgroundColourAux.QI` und `Q_BackgroundColourAux.QO` mit `QO` verbunden.

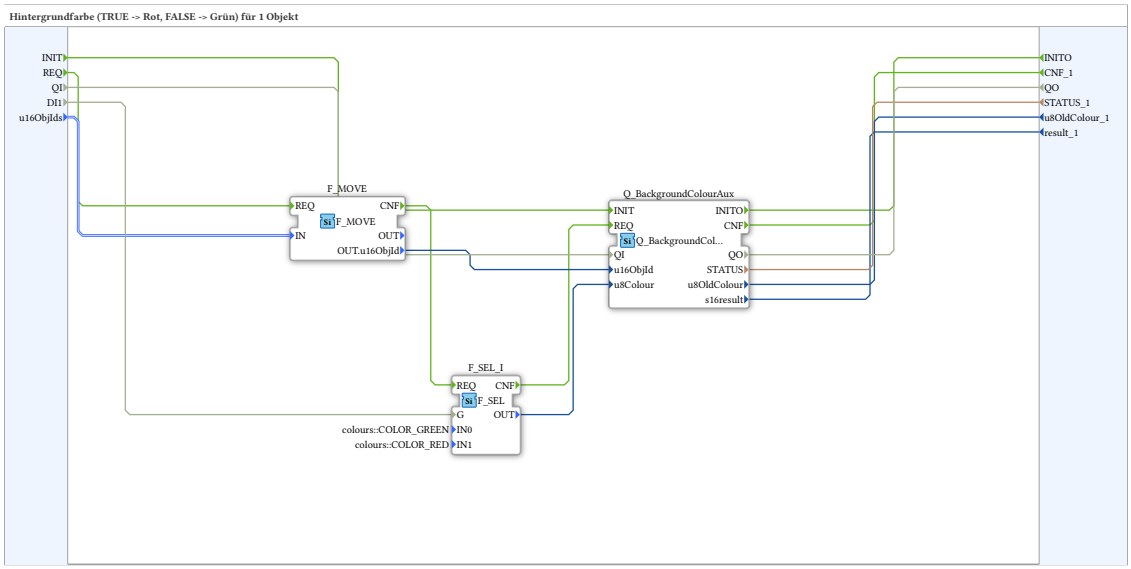
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Grün, 1 Objekt, BOOL-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedGreenBackground1_auxS



• – ○ • – ○ • – ○ •

Einleitung

`RedGreenBackground1_auxS` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Grün**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Quality-Anschlüsse: Der Baustein besitzt zusätzlich das Ereignis `INIT` (Eingang) und `INITO` (Ausgang) sowie den Dateneingang `QI` (BOOL, Standard `TRUE`, „Quality Input – TRUE, um den Dienst freizugeben“) und den Datenausgang `QO` (BOOL, „Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde“). `INIT` wird an `Q_BackgroundColourAux.INIT` weitergereicht, `Q_BackgroundColourAux.INITO` an `INITO`; `QI` ist mit `Q_BackgroundColourAux.QI` und `Q_BackgroundColourAux.QO` mit `QO` verbunden.

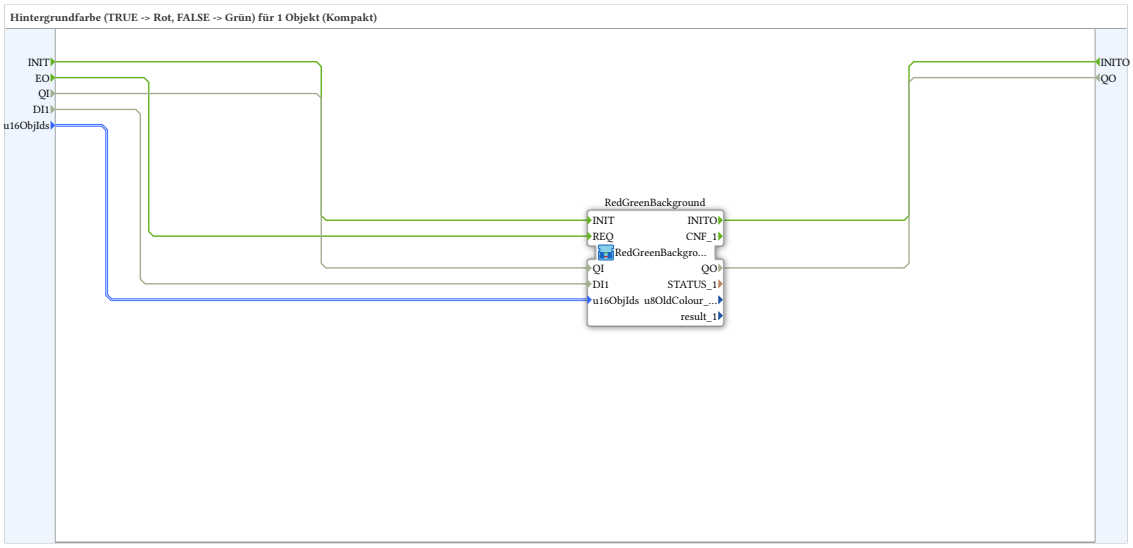
Zusammenfassung

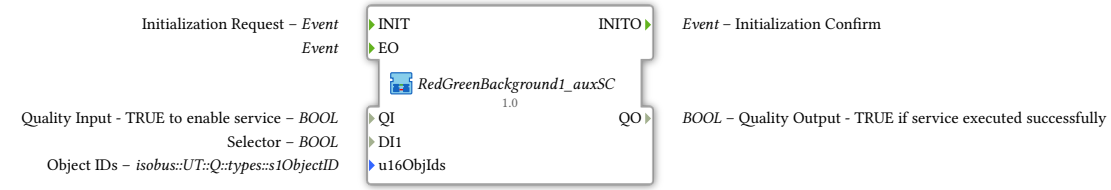
Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Grün, 1 Objekt, BOOL-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedGreenBackground1_auxSC





• — ○ • — ○ • — ○ •

Einleitung

`RedGreenBackground1_auxSC` ist ein **Kompakt-Wrapper** um `RedGreenBackground1_auxS` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen. Zusätzlich werden `INIT` ⟶ `RedGreenBackground.INIT`, `RedGreenBackground.INITO` ⟶ `INITO`, `QI` ⟶ `RedGreenBackground.QI` (Initialwert `TRUE`) und `RedGreenBackground.QO` ⟶ `QO` durchgereicht.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

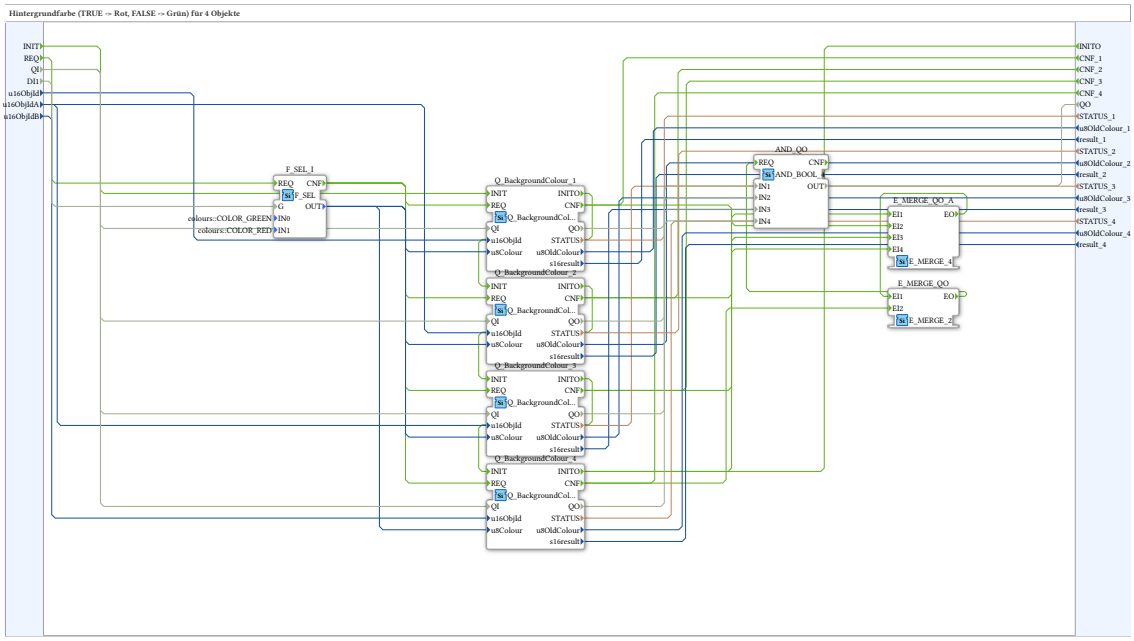
Zusammenfassung

Reine Weiterleitung an `RedGreenBackground1_auxS` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedGreenBackground4



• — ○ • — ○ • — ○ •

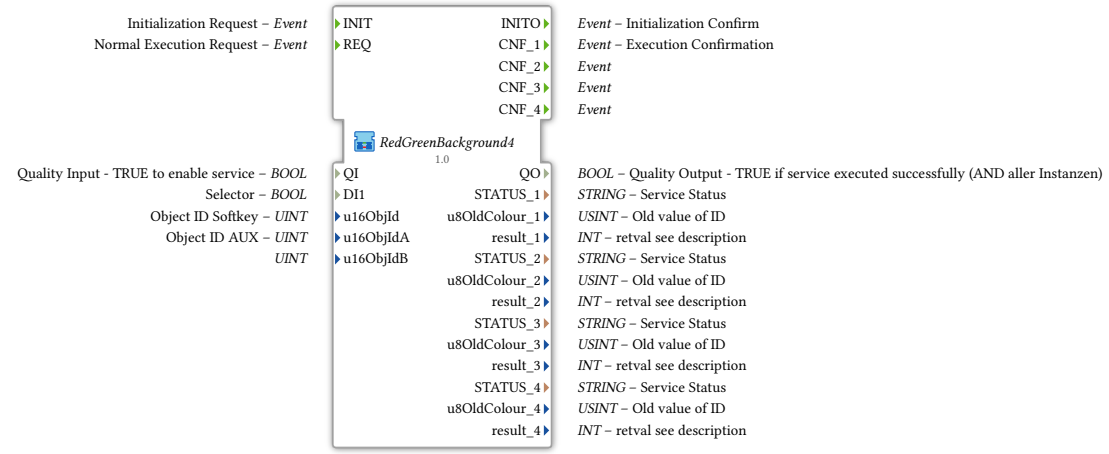
Einleitung

`RedGreenBackground4` schaltet die VT-Hintergrundfarbe von 4 Objekten anhand eines booleschen Selector-Signals: `TRUE` → **Rot**, `FALSE` → **Grün**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über die Eingänge `u16ObjId, u16ObjIdA, u16ObjIdB` übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`u16ObjId`	`Q_BackgroundColour` (normales Objekt)
2	`u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
3	`u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)
4	`u16ObjIdB`	`Q_BackgroundColour` (normales Objekt)

Initialisierung und Qualität: Zusätzlich zu den bisherigen Ereignissen besitzt der Baustein den Ereigniseingang `INIT`, den Ereignisausgang `INITO` sowie `QI` (BOOL, Eingang, Initialwert `TRUE`) und `QO` (BOOL, Ausgang, UND aller Instanzen). Init-Kette: `INIT` → `Q_BackgroundColour_1` → `Q_BackgroundColour_2` → `Q_BackgroundColour_3` → `Q_BackgroundColour_4` → `INITO` (jeweils `INITO` → nächstes `INIT`). `QI` wird an alle `Q_BackgroundColour_n.QI` weitergereicht. `QO` bildet `AND_QO` (`iec61131::booleanOperators::AND_BOOL_4`): `Q_BackgroundColour_1.QO` → `AND_QO.IN1`, `Q_BackgroundColour_2.QO` → `AND_QO.IN2`, `Q_BackgroundColour_3.QO` → `AND_QO.IN3`, `Q_BackgroundColour_4.QO` → `AND_QO.IN4`, `AND_QO.OUT` → `QO`. `E_MERGE_QO_A` (`E_MERGE_4`) führt `Q_BackgroundColour_4.INITO` und die `CNF`-Ausgänge von `Q_BackgroundColour_1`, `Q_BackgroundColour_2` und `Q_BackgroundColour_3` zusammen, `E_MERGE_QO` (`E_MERGE_2`) fügt `Q_BackgroundColour_4.CNF` hinzu; `E_MERGE_QO.EO` triggert `AND_QO.REQ`.

Allgemeines Muster (Selector → `AX_SEL`/`F_SEL` → `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



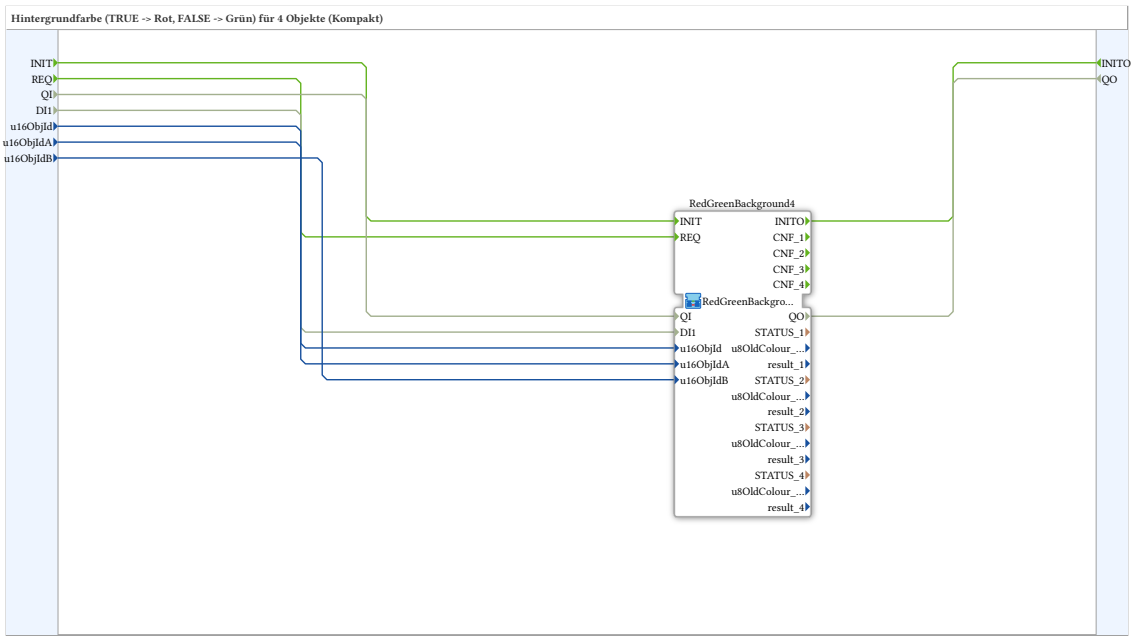
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Grün, 4 Objekte, BOOL-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedGreenBackground4C



• – ○ • – ○ • – ○ •

Einleitung

`RedGreenBackground4C` ist ein **Kompakt-Wrapper** um `RedGreenBackground4` — reicht `u16ObjId` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

Zusätzlich führt der Wrapper die Schnittstelle für Initialisierung und Qualität nach außen: Ereigniseingang `INIT`, Ereignisausgang `INITO`, Dateneingang `QI` (`BOOL`, Initialwert `TRUE`) und Datenausgang `QO` (`BOOL`). Sie sind 1:1 an die innere Instanz `RedGreenBackground4` durchgeschleift: `INIT` ⟶ `RedGreenBackground4.INIT`, `RedGreenBackground4.INITO` ⟶ `INITO`, `QI` ⟶ `RedGreenBackground4.QI`, `RedGreenBackground4.QO` ⟶ `QO`.

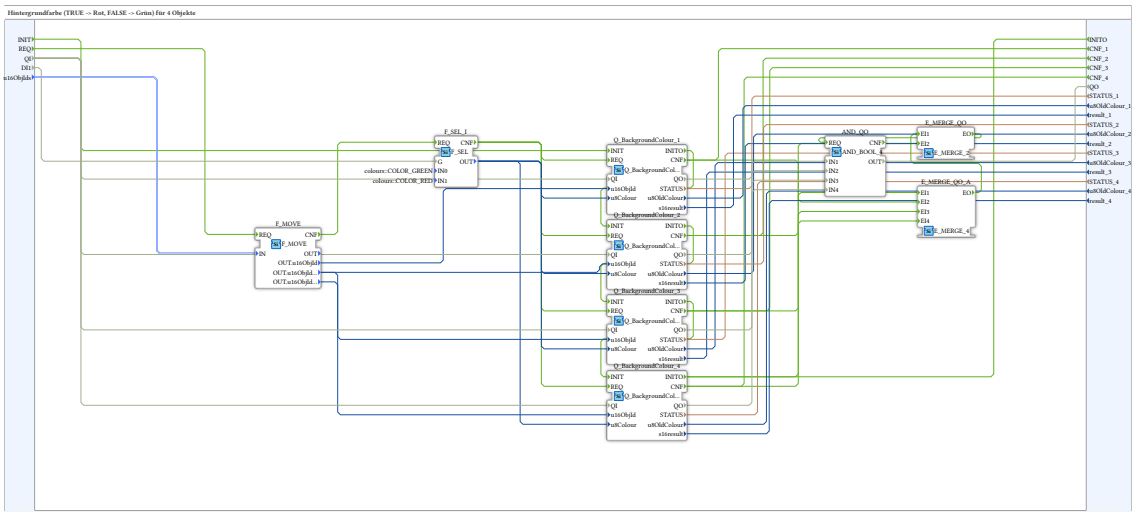
Zusammenfassung

Reine Weiterleitung an `RedGreenBackground4` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedGreenBackground4S



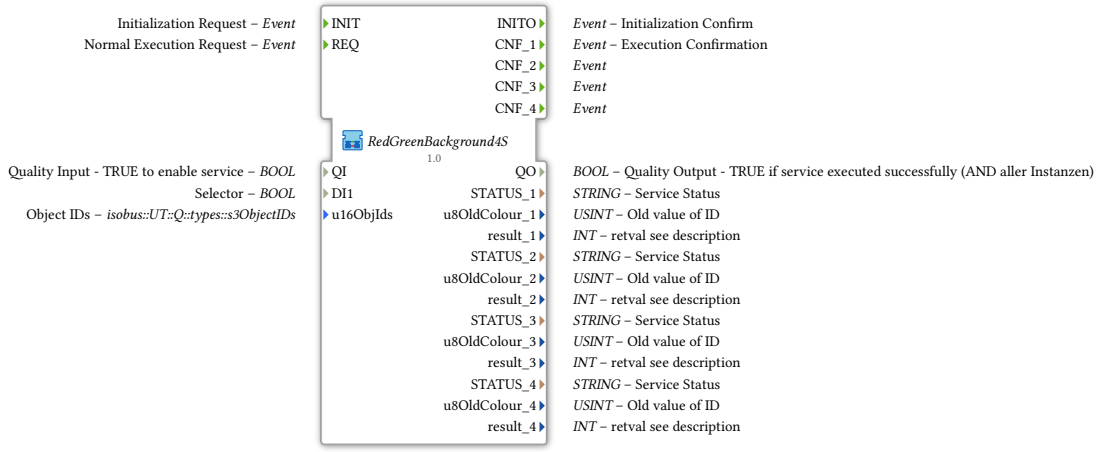
• – ○ • – ○ • – ○ •

Einleitung

`RedGreenBackground4S` schaltet die VT-Hintergrundfarbe von 4 Objekten anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Grün**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s3ObjectIDs` (`u16ObjIds`) übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`F_MOVE.OUT.u16ObjId`	`Q_BackgroundColour` (normales Objekt)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
3	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)
4	`F_MOVE.OUT.u16ObjIdB`	`Q_BackgroundColour` (normales Objekt)

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Initialisierung und Qualität

- **Ereignis-Eingang `INIT`** (`EInit`) und **Ereignis-Ausgang `INITO`** (`EInit`): die Initialisierung wird durch die `Q_BackgroundColour`-Instanzen gekettet: `INIT` ⟶ `Q_BackgroundColour_1.INIT`; `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` ⟶ `Q_BackgroundColour_3.INIT`; `Q_BackgroundColour_3.INITO` ⟶ `Q_BackgroundColour_4.INIT`; `Q_BackgroundColour_4.INITO` ⟶ `INITO`.
- **Dateneingang `QI`** (`BOOL`, Vorgabe `TRUE`) wird an `.QI` aller Instanzen verteilt (`Q_BackgroundColour_1`, `Q_BackgroundColour_2`, `Q_BackgroundColour_3`, `Q_BackgroundColour_4`).
- **Datenausgang `QO`** (`BOOL`): UND aller Instanzen über `AND_QO` (`iec61131::booleanOperators::AND_BOOL_4`): `Q_BackgroundColour_1.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN2`, `Q_BackgroundColour_3.QO` ⟶ `AND_QO.IN3`, `Q_BackgroundColour_4.QO` ⟶ `AND_QO.IN4`; `AND_QO.OUT` ⟶ `QO`.
- `E_MERGE_QO_A` (`iec61499::events::E_MERGE_4`) vereint `Q_BackgroundColour_4.INITO` und die `CNF`-Ereignisse der Instanzen 1 bis 3; `E_MERGE_QO` (`iec61499::events::E_MERGE_2`) vereint dessen Ausgang `E_MERGE_QO_A.EO` mit `Q_BackgroundColour_4.CNF` und löst damit `AND_QO.REQ` aus.

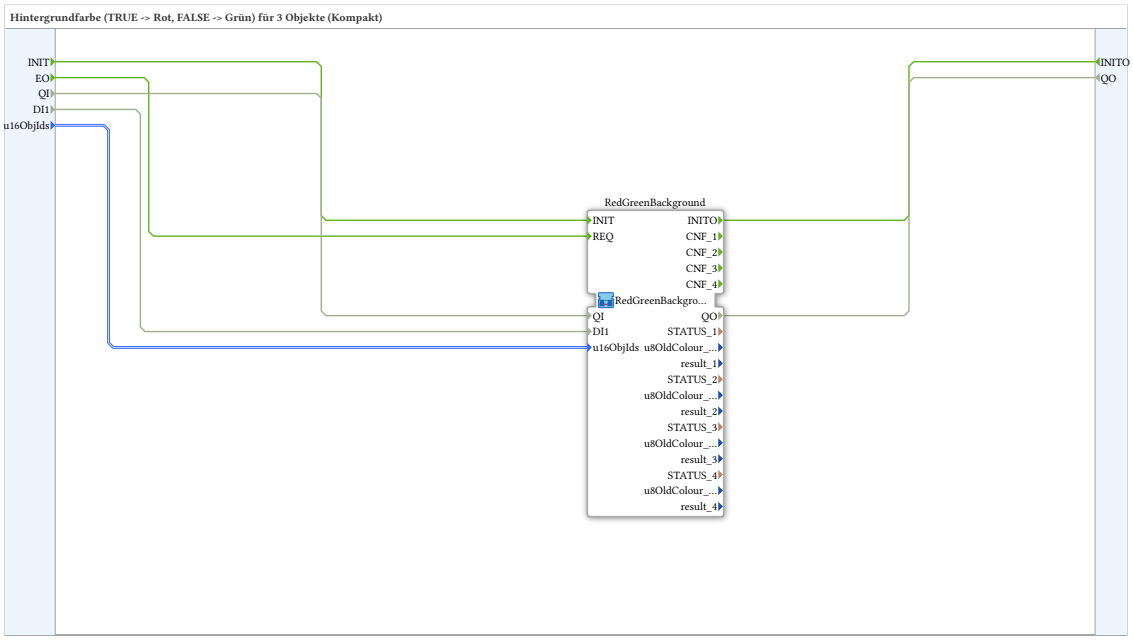
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Grün, 4 Objekte, BOOL-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedGreenBackground4SC



• – ○ • – ○ • – ○ •

Einleitung

`RedGreenBackground4SC` ist ein **Kompakt-Wrapper** um `RedGreenBackground4S` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Schnittstelle (Quality-Muster): Der Wrapper besitzt zusätzlich den Ereignis-Eingang `INIT` und den Ereignis-Ausgang `INITO` (je `EInit`) sowie den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`) und den Datenausgang `QO` (`BOOL`). Sie sind direkt an die innere Instanz `RedGreenBackground` durchgeschleift: `INIT` ⟶ `RedGreenBackground.INIT`, `RedGreenBackground.INITO` ⟶ `INITO`, `QI` ⟶ `RedGreenBackground.QI`, `RedGreenBackground.QO` ⟶ `QO`.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

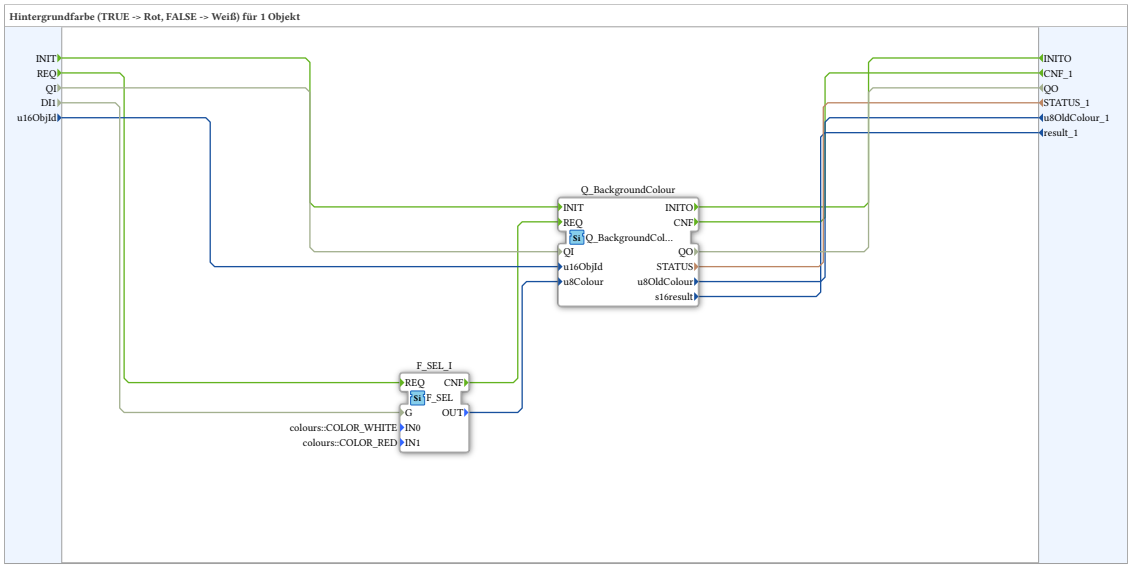
Zusammenfassung

Reine Weiterleitung an `RedGreenBackground4S` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground1



• — ○ • — ○ • — ○ •

Einleitung

`RedWhiteBackground1` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den Eingang `u16ObjId` übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Quality-Anschlüsse: Der Baustein besitzt zusätzlich das Ereignis `INIT` (Eingang) und `INITO` (Ausgang) sowie den Dateneingang `QI` (BOOL, Standard `TRUE`, „Quality Input – TRUE, um den Dienst freizugeben“) und den Datenausgang `QO` (BOOL, „Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde“). `INIT` wird an `Q_BackgroundColour.INIT` weitergereicht, `Q_BackgroundColour.INITO` an `INITO`; `QI` ist mit `Q_BackgroundColour.QI` und `Q_BackgroundColour.QO` mit `QO` verbunden.

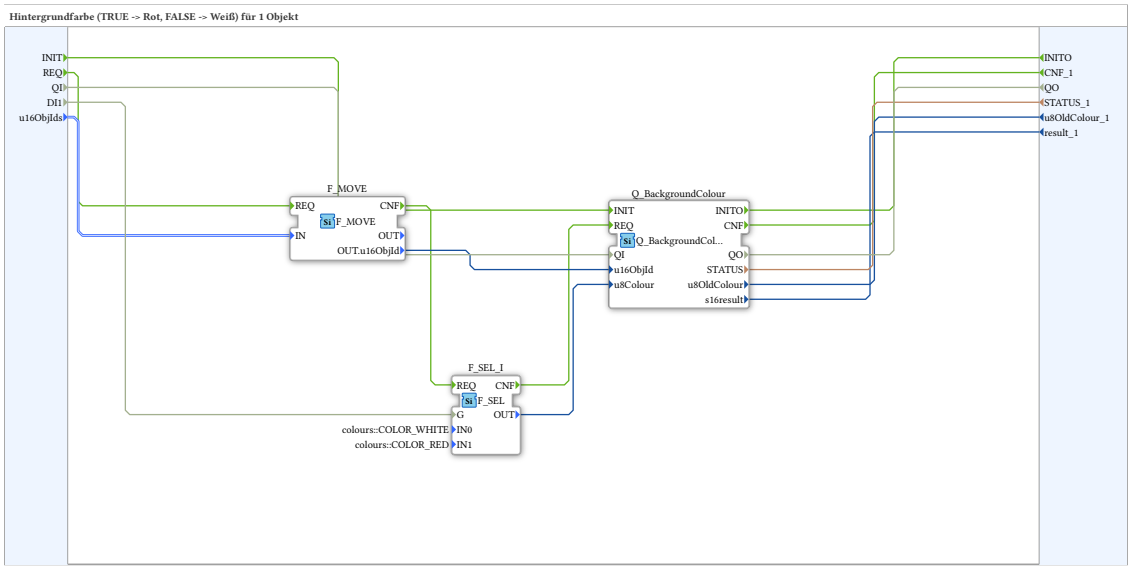
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Weiß, 1 Objekt, BOOL-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground1S



• – ○ • – ○ • – ○ •

Einleitung

`RedWhiteBackground1S` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Quality-Anschlüsse: Der Baustein besitzt zusätzlich das Ereignis `INIT` (Eingang) und `INITO` (Ausgang) sowie den Dateneingang `QI` (BOOL, Standard `TRUE`, „Quality Input – TRUE, um den Dienst freizugeben“) und den Datenausgang `QO` (BOOL, „Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde“). `INIT` wird an `Q_BackgroundColour.INIT` weitergereicht, `Q_BackgroundColour.INITO` an `INITO`; `QI` ist mit `Q_BackgroundColour.QI` und `Q_BackgroundColour.QO` mit `QO` verbunden.

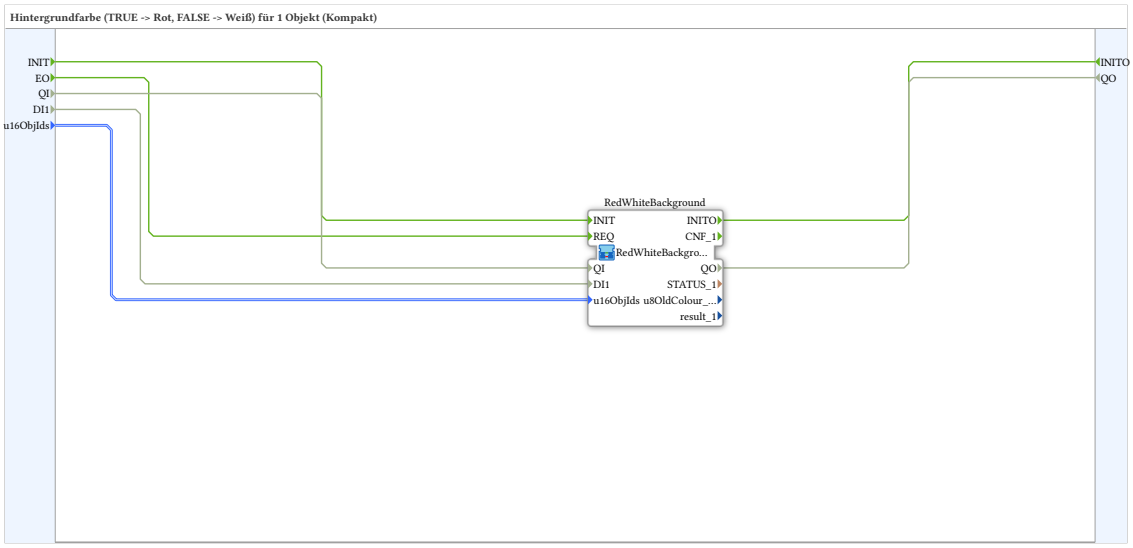
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Weiß, 1 Objekt, BOOL-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground1SC



• – ◦ • – ◦ • – ◦ •

Einleitung

`RedWhiteBackground1SC` ist ein **Kompakt-Wrapper** um `RedWhiteBackground1S` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

Initialisierung und Status (INIT/INITO, QI/QO)

- **Ereignis-Eingang** `INIT` (`EInit`, Initialization Request) und **Ereignis-Ausgang** `INITO` (`EInit`, Initialization Confirm)
- **Daten-Eingang** `QI` (`BOOL`, Vorgabe `TRUE`, Quality Input) wird an `RedWhiteBackground.QI` weitergegeben
- **Daten-Ausgang** `QO` (`BOOL`, Quality Output): durchgereicht von `RedWhiteBackground.QO`
- **Initialisierungskette**: `INIT` ⟶ `RedWhiteBackground.INIT`, `RedWhiteBackground.INITO` ⟶ `INITO`

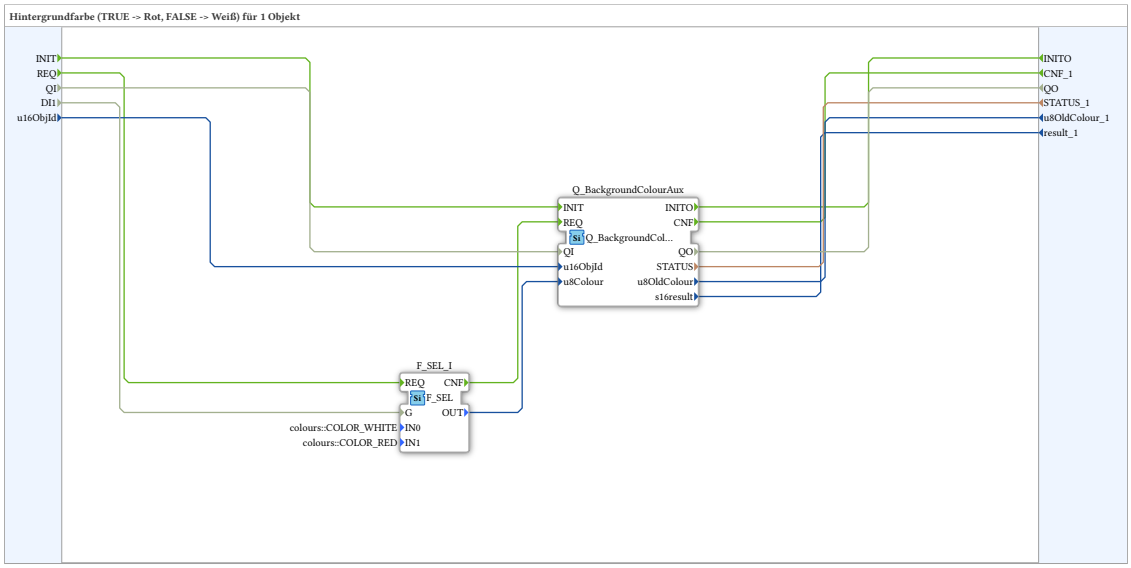
Zusammenfassung

Reine Weiterleitung an `RedWhiteBackground1S` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground1_aux



• — ○ • — ○ • — ○ •

Einleitung

`RedWhiteBackground1_aux` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den Eingang `u16ObjId` übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Quality-Anschlüsse: Der Baustein besitzt zusätzlich das Ereignis `INIT` (Eingang) und `INITO` (Ausgang) sowie den Dateneingang `QI` (BOOL, Standard `TRUE`, „Quality Input – TRUE, um den Dienst freizugeben“) und den Datenausgang `QO` (BOOL, „Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde“). `INIT` wird an `Q_BackgroundColourAux.INIT` weitergereicht, `Q_BackgroundColourAux.INITO` an `INITO`; `QI` ist mit `Q_BackgroundColourAux.QI` und `Q_BackgroundColourAux.QO` mit `QO` verbunden.

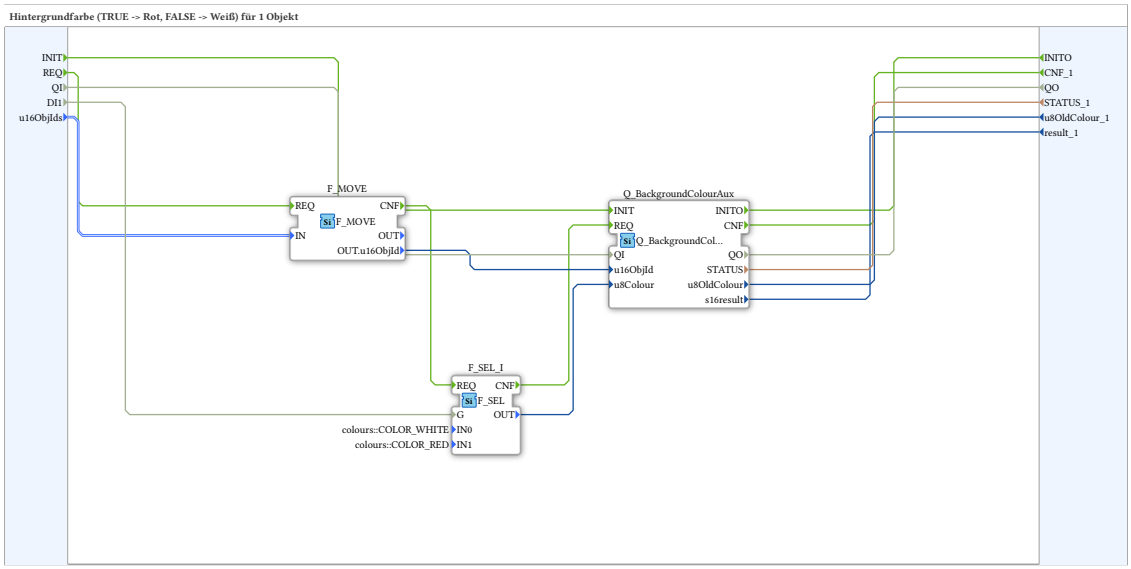
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Weiß, 1 Objekt, BOOL-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground1_auxS



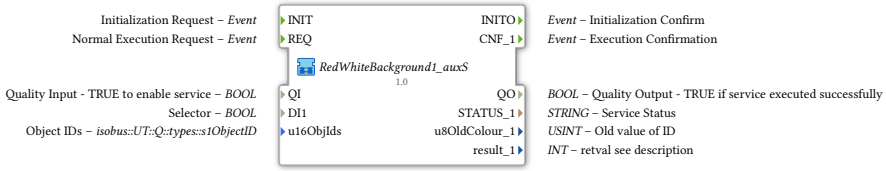
• – ○ • – ○ • – ○ •

Einleitung

`RedWhiteBackground1_auxS` schaltet die VT-Hintergrundfarbe eines Objekts anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s1ObjectID` (`u16ObjIds`) übergeben.

Allgemeines Muster (Selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Quality-Anschlüsse: Der Baustein besitzt zusätzlich das Ereignis `INIT` (Eingang) und `INITO` (Ausgang) sowie den Dateneingang `QI` (BOOL, Standard `TRUE`, „Quality Input – TRUE, um den Dienst freizugeben“) und den Datenausgang `QO` (BOOL, „Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde“). `INIT` wird an `Q_BackgroundColourAux.INIT` weitergereicht, `Q_BackgroundColourAux.INITO` an `INITO`; `QI` ist mit `Q_BackgroundColourAux.QI` und `Q_BackgroundColourAux.QO` mit `QO` verbunden.



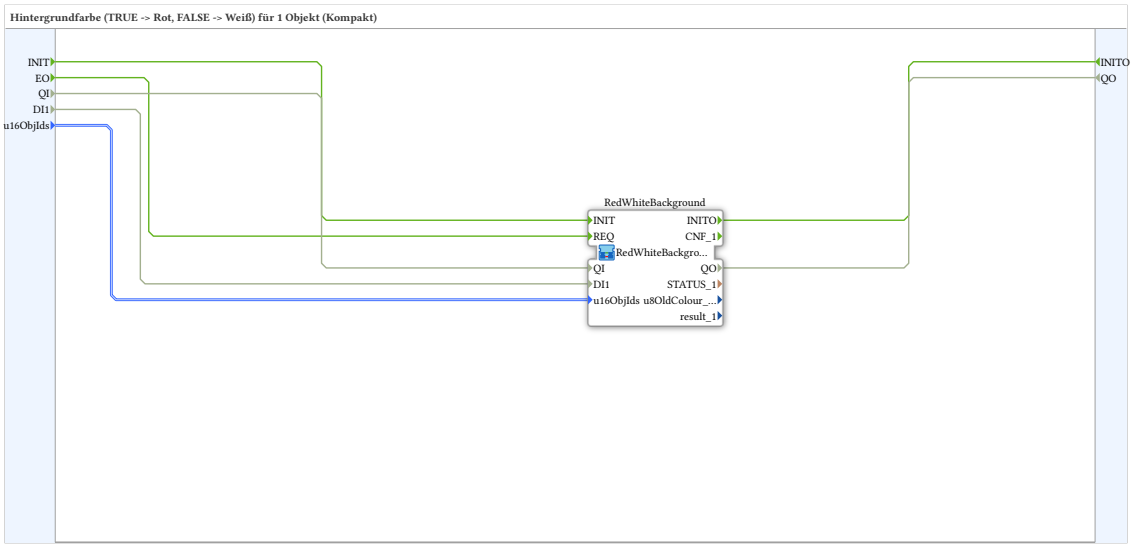
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Weiß, 1 Objekt, BOOL-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground1_auxSC



• – ○ • – ○ • – ○ •

Einleitung

`RedWhiteBackground1_auxSC` ist ein **Kompakt-Wrapper** um `RedWhiteBackground1_auxS` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen. Zusätzlich werden `INIT` ⟶ `RedWhiteBackground.INIT`, `RedWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `RedWhiteBackground.QI` (Initialwert `TRUE`) und `RedWhiteBackground.QO` ⟶ `QO` durchgereicht.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

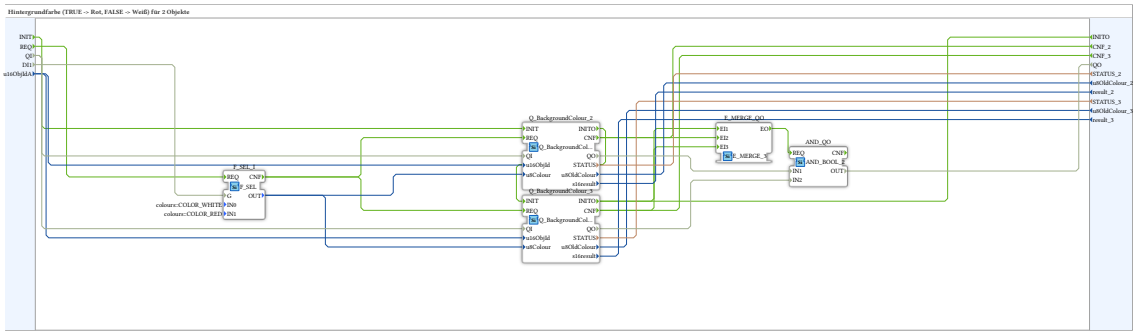
Zusammenfassung

Reine Weiterleitung an `RedWhiteBackground1_auxS` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground2



• – ○ • – ○ • – ○ •

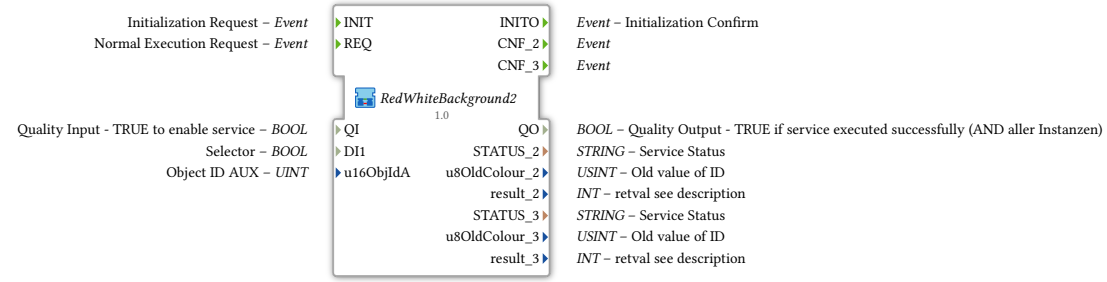
Einleitung

`RedWhiteBackground2` schaltet die VT-Hintergrundfarbe von 2 Objekten anhand eines booleschen Selector-Signals: `TRUE` → **Rot**, `FALSE` → **Weiß**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über die Eingänge `u16ObjIdA` übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
2	`u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)

Initialisierung und Qualität: Zusätzlich zu den bisherigen Ereignissen besitzt der Baustein den Ereigniseingang `INIT`, den Ereignisausgang `INITO` sowie `QI` (BOOL, Eingang, Initialwert `TRUE`) und `QO` (BOOL, Ausgang, UND aller Instanzen). Init-Kette: `INIT` → `Q_BackgroundColour_2` → `Q_BackgroundColour_3` → `INITO` (jeweils `INITO` → nächstes `INIT`). `QI` wird an alle `Q_BackgroundColour_n.QI` weitergereicht. `QO` bildet `AND_QO` (`iec61131::booleanOperators::AND_BOOL_2`): `Q_BackgroundColour_2.QO` → `AND_QO.IN1`, `Q_BackgroundColour_3.QO` → `AND_QO.IN2`, `AND_QO.OUT` → `QO`. `E_MERGE_QO` (`E_MERGE_3`) führt `Q_BackgroundColour_3.INITO` und die `CNF`-Ausgänge `Q_BackgroundColour_2.CNF`, `Q_BackgroundColour_3.CNF` zusammen; `E_MERGE_QO.EO` triggert `AND_QO.REQ`.

Allgemeines Muster (Selector → `AX_SEL`/`F_SEL` → `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



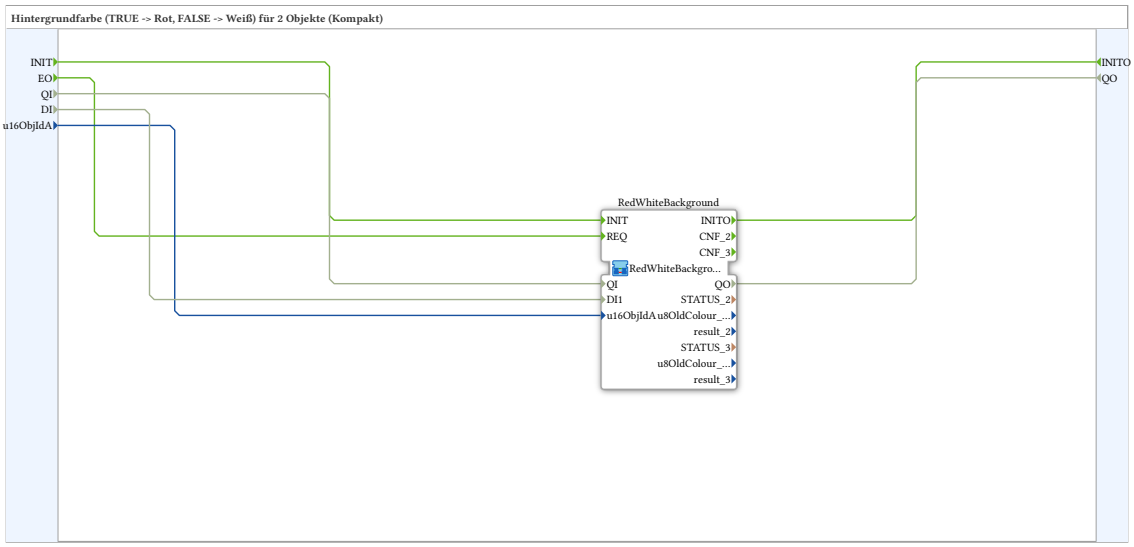
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Weiß, 2 Objekte, BOOL-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground2C



• – ○ • – ○ • – ○ •

Einleitung

`RedWhiteBackground2C` ist ein **Kompakt-Wrapper** um `RedWhiteBackground2` — reicht `u16ObjId` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

Zusätzlich führt der Wrapper die Schnittstelle für Initialisierung und Qualität nach außen: Ereigniseingang `INIT`, Ereignisausgang `INITO`, Dateneingang `QI` (`BOOL`, Initialwert `TRUE`) und Datenausgang `QO` (`BOOL`). Sie sind 1:1 an die innere Instanz `RedWhiteBackground` durchgeschleift: `INIT` ⟶ `RedWhiteBackground.INIT`, `RedWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `RedWhiteBackground.QI`, `RedWhiteBackground.QO` ⟶ `QO`.

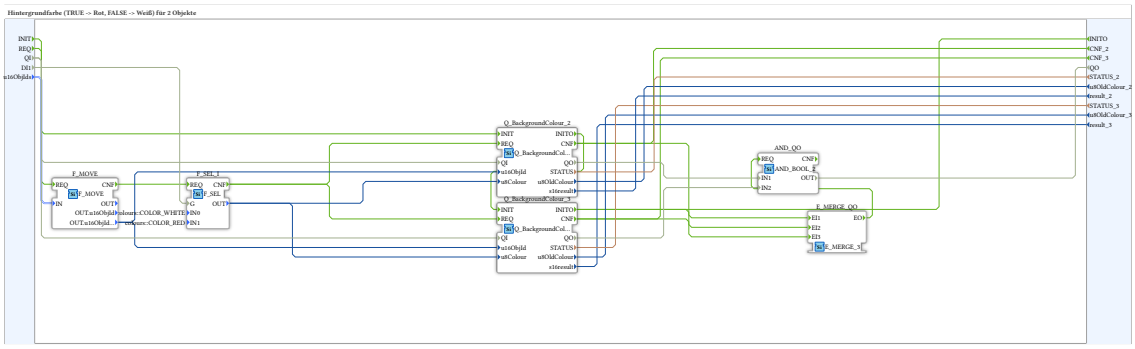
Zusammenfassung

Reine Weiterleitung an `RedWhiteBackground2` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground2S



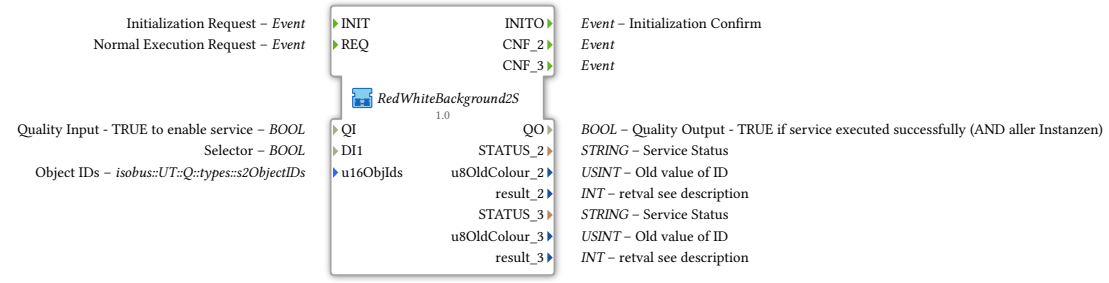
• – ○ • – ○ • – ○ •

Einleitung

`RedWhiteBackground2S` schaltet die VT-Hintergrundfarbe von 2 Objekten anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s2ObjectIDs` (`u16ObjIds`) übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Initialisierung und Qualität

- **Ereignis-Eingang `INIT`** (`EInit`) und **Ereignis-Ausgang `INITO`** (`EInit`): die Initialisierung wird durch die `Q_BackgroundColour`-Instanzen gekettet: `INIT` ⟶ `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` ⟶ `Q_BackgroundColour_3.INIT`; `Q_BackgroundColour_3.INITO` ⟶ `INITO`.
- **Dateneingang `QI`** (`BOOL`, Vorgabe `TRUE`) wird an `.QI` aller Instanzen verteilt (`Q_BackgroundColour_2`, `Q_BackgroundColour_3`).
- **Datenausgang `QO`** (`BOOL`): UND aller Instanzen über `AND_QO` (`iec61131::booleanOperators::AND_BOOL_2`): `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_3.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`.
- `E_MERGE_QO` (`iec61499::events::E_MERGE_3`) vereint `Q_BackgroundColour_3.INITO` und die `CNF`-Ereignisse beider Instanzen (`Q_BackgroundColour_2.CNF`, `Q_BackgroundColour_3.CNF`) und löst damit `AND_QO.REQ` aus.

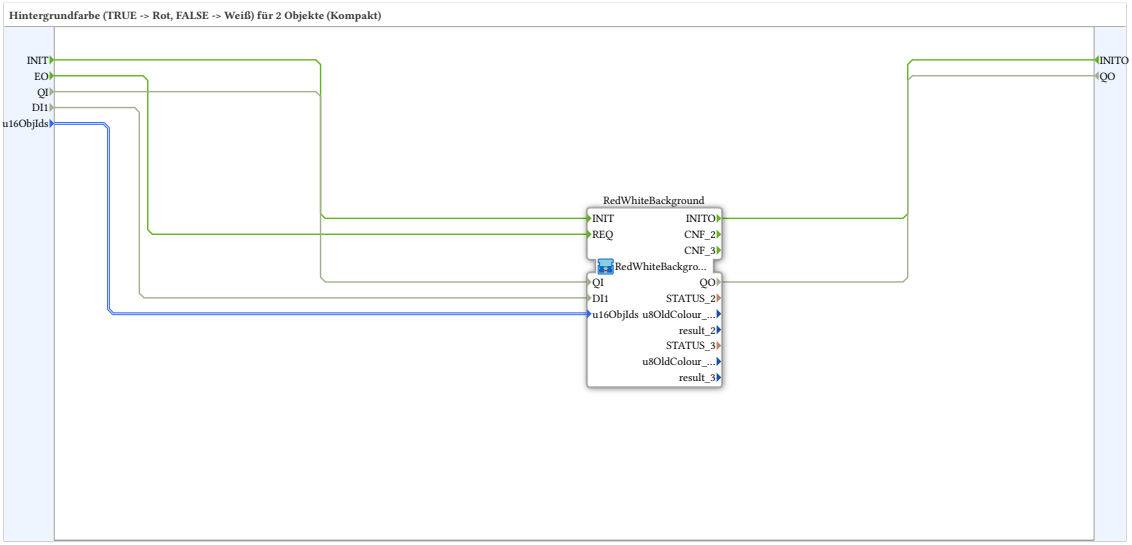
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Weiß, 2 Objekte, BOOL-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground2SC



• – ○ • – ○ • – ○ •

Einleitung

`RedWhiteBackground2SC` ist ein **Kompakt-Wrapper** um `RedWhiteBackground2S` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Schnittstelle (Quality-Muster): Der Wrapper besitzt zusätzlich den Ereignis-Eingang `INIT` und den Ereignis-Ausgang `INITO` (je `EInit`) sowie den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`) und den Datenausgang `QO` (`BOOL`). Sie sind direkt an die innere Instanz `RedWhiteBackground` durchgeschleift: `INIT` ⟶ `RedWhiteBackground.INIT`, `RedWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `RedWhiteBackground.QI`, `RedWhiteBackground.QO` ⟶ `QO`.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

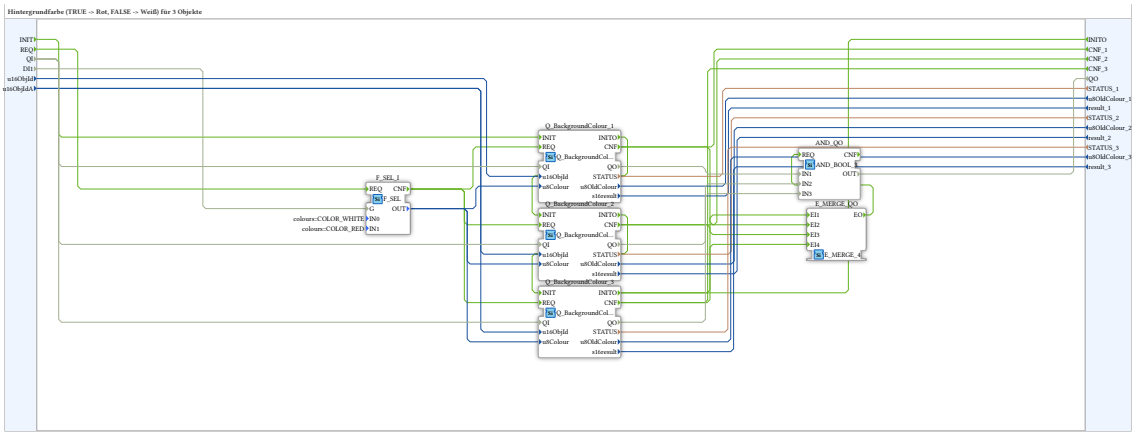
Zusammenfassung

Reine Weiterleitung an `RedWhiteBackground2S` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground3



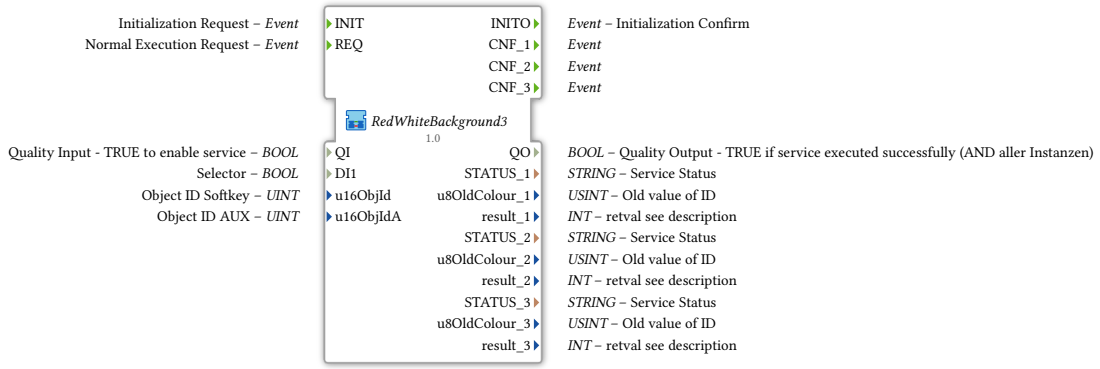
• – ○ • – ○ • – ○ •

Einleitung

`RedWhiteBackground3` schaltet die VT-Hintergrundfarbe von 3 Objekten anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über die Eingänge `u16ObjId, u16ObjIdA` übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`u16ObjId`	`Q_BackgroundColour` (normales Objekt)
2	`u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
3	`u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Initialisierung und Status (INIT/INITO, QI/QO)

- **Ereignis-Eingang** `INIT` (`EInit`, Initialization Request) und **Ereignis-Ausgang** `INITO` (`EInit`, Initialization Confirm)
- **Daten-Eingang** `QI` (`BOOL`, Vorgabe `TRUE`, Quality Input) wird an `Q_BackgroundColour_1.QI`, `Q_BackgroundColour_2.QI` und `Q_BackgroundColour_3.QI` weitergegeben
- **Daten-Ausgang** `QO` (`BOOL`, Quality Output): UND (`AND_QO`, Typ `iec61131::booleanOperators::AND_BOOL_3`) der `QO`-Ausgänge von `Q_BackgroundColour_1.QO`, `Q_BackgroundColour_2.QO` und `Q_BackgroundColour_3.QO`
- **Initialisierungskette**: `INIT` → `Q_BackgroundColour_1.INIT`, `Q_BackgroundColour_1.INITO` → `Q_BackgroundColour_2.INIT`, `Q_BackgroundColour_2.INITO` → `Q_BackgroundColour_3.INIT`, `Q_BackgroundColour_3.INITO` → `INITO`
- `E_MERGE_QO` (Typ `iec61499::events::E_MERGE_4`) führt `Q_BackgroundColour_3.INITO`, `Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF` und `Q_BackgroundColour_3.CNF` zusammen und löst `AND_QO.REQ` aus

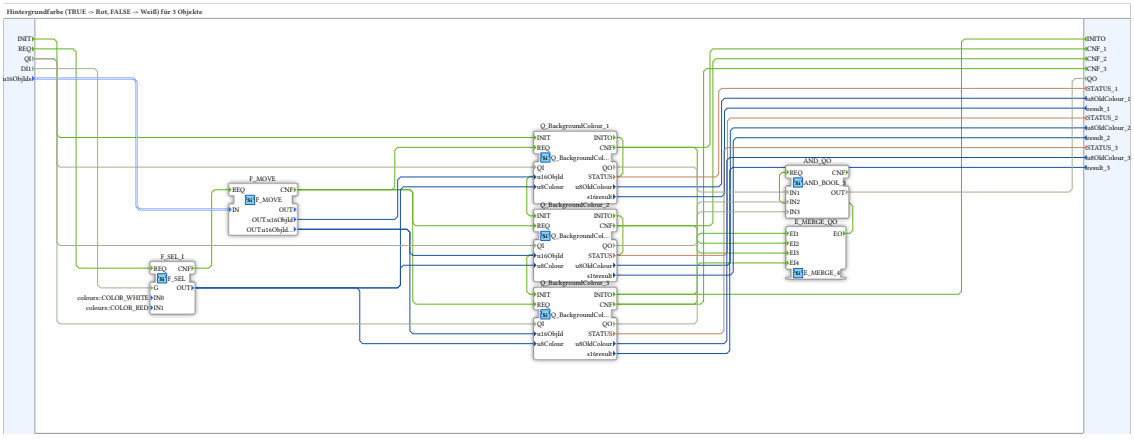
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Weiß, 3 Objekte, BOOL-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground3S



• – ○ • – ○ • – ○ •

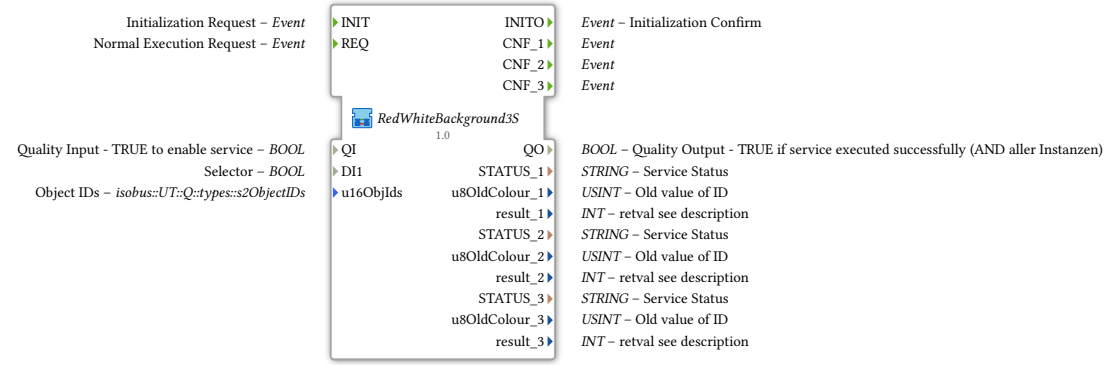
Einleitung

`RedWhiteBackground3S` schaltet die VT-Hintergrundfarbe von 3 Objekten anhand eines booleschen Selector-Signals: `TRUE` → **Rot**, `FALSE` → **Weiß**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s2ObjectIDs` (`u16ObjIds`) übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`F_MOVE.OUT.u16ObjId`	`Q_BackgroundColour` (normales Objekt)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
3	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Initialisierung und Qualität: Die SubApp besitzt `INIT`/`INITO` sowie `QI`/`QO`. `INIT` ⟶ `Q_BackgroundColour_1.INIT` ⟶ `Q_BackgroundColour_2.INIT` ⟶ `Q_BackgroundColour_3.INIT` (jeweils über `INITO`), `Q_BackgroundColour_3.INITO` ⟶ `INITO`. `QI` geht an alle drei Instanzen. `QO` ist das `AND` (`AND_QO`, `AND_BOOL_3`) der drei `QO`-Ausgänge; `AND_QO.REQ` wird über `E_MERGE_QO` (`E_MERGE_4`: `Q_BackgroundColour_3.INITO`, `Q_BackgroundColour_1.CNF`, `_2.CNF`, `_3.CNF`) ausgelöst.



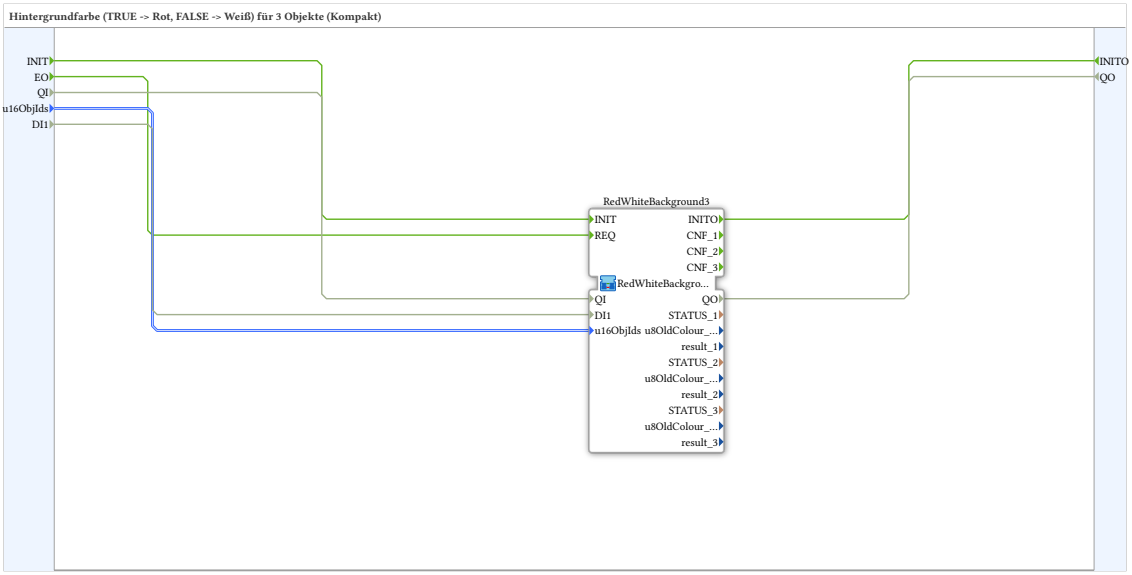
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Weiß, 3 Objekte, BOOL-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground3SC



• – ○ • – ○ • – ○ •

Einleitung

`RedWhiteBackground3SC` ist ein **Kompakt-Wrapper** um `RedWhiteBackground3S` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Zusätzlich werden Initialisierung und Qualität durchgereicht: `INIT` ⟶ `RedWhiteBackground3.INIT`, `RedWhiteBackground3.INITO` ⟶ `INITO`, `QI` (Initialwert `TRUE`) ⟶ `RedWhiteBackground3.QI` und `RedWhiteBackground3.QO` ⟶ `QO`.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

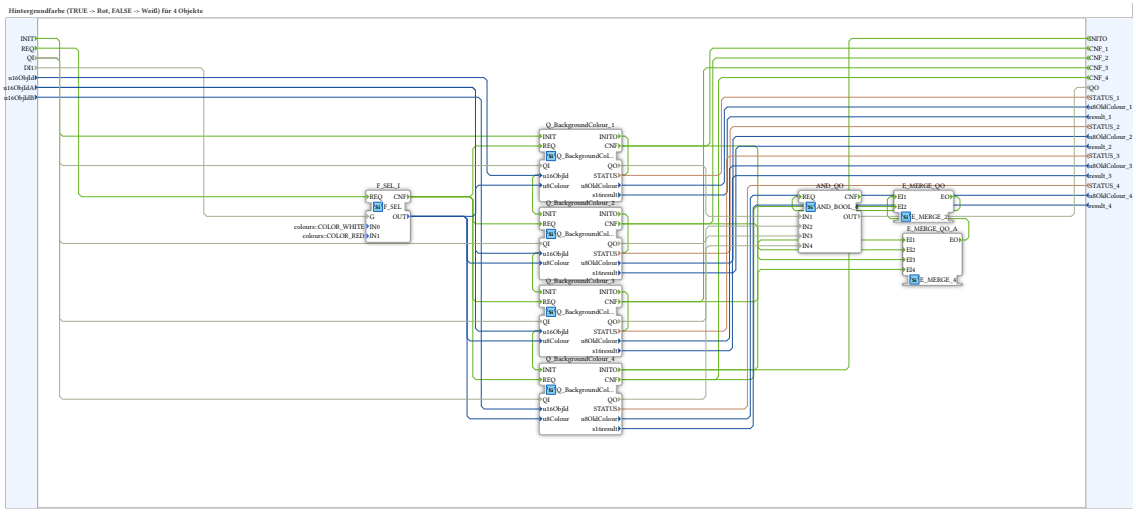
Zusammenfassung

Reine Weiterleitung an `RedWhiteBackground3S` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground4



• – ○ • – ○ • – ○ •

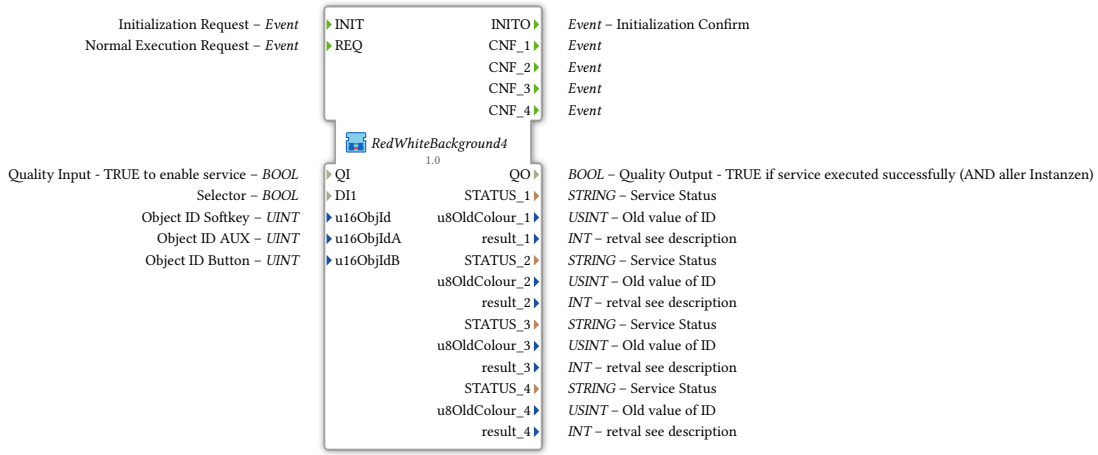
Einleitung

`RedWhiteBackground4` schaltet die VT-Hintergrundfarbe von 4 Objekten anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über die Eingänge `u16ObjId, u16ObjIdA, u16ObjIdB` übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`u16ObjId`	`Q_BackgroundColour` (normales Objekt)
2	`u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
3	`u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)
4	`u16ObjIdB`	`Q_BackgroundColour` (normales Objekt)

Allgemeines Muster (Selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).

Zusätzlich besitzt der Baustein die Schnittstelle für Initialisierung und Qualität: Ereigniseingang `INIT`, Ereignisausgang `INITO`, Dateneingang `QI` (`BOOL`, Initialwert `TRUE`) und Datenausgang `QO` (`BOOL`). `INIT` ⟶ `Q_BackgroundColour_1.INIT`, `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`, `Q_BackgroundColour_2.INITO` ⟶ `Q_BackgroundColour_3.INIT`, `Q_BackgroundColour_3.INITO` ⟶ `Q_BackgroundColour_4.INIT`, `Q_BackgroundColour_4.INITO` ⟶ `INITO` (die vier Instanzen werden also nacheinander initialisiert). `QI` geht an `QI` aller vier Instanzen. `QO` ist der Ausgang der Instanz `AND_QO` (`iec61131::booleanOperators::AND_BOOL_4`), die die `QO` der vier Instanzen verknüpft. Ihr `REQ` wird von `E_MERGE_QO` (`E_MERGE_2`) ausgelöst, das den Ausgang von `E_MERGE_QO_A` (`E_MERGE_4`: `Q_BackgroundColour_4.INITO` sowie `CNF` von `Q_BackgroundColour_1` bis `_3`) und `Q_BackgroundColour_4.CNF` zusammenführt.



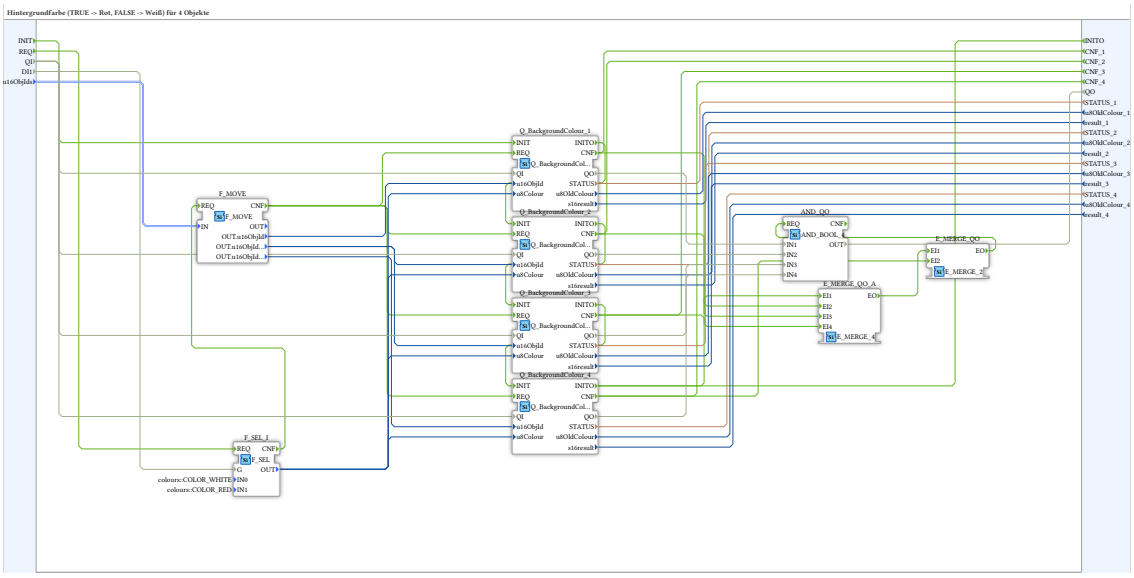
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Weiß, 4 Objekte, BOOL-Selector.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground4S



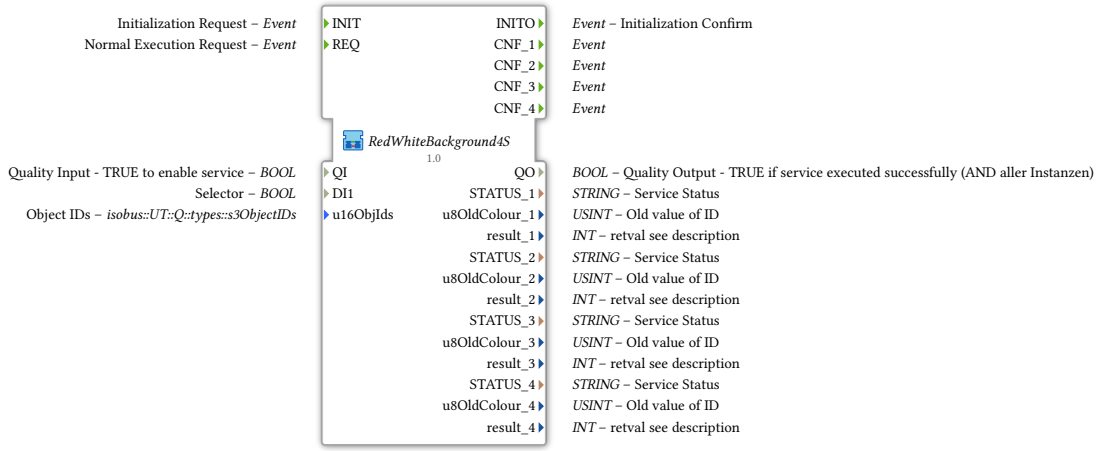
• – ○ • – ○ • – ○ •

Einleitung

`RedWhiteBackground4S` schaltet die VT-Hintergrundfarbe von 4 Objekten anhand eines booleschen Selector-Signals: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. Das Selector-Signal kommt als einfacher `BOOL`-Dateneingang (`DI1`). Die Objekt-ID wird über den strukturierten Typ `s3ObjectIDs` (`u16ObjIds`) übergeben.

Position	Objekt-ID-Quelle	Baustein
1	`F_MOVE.OUT.u16ObjId`	`Q_BackgroundColour` (normales Objekt)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (normales Objekt)
3	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (Auxiliary-Function-Objekt)
4	`F_MOVE.OUT.u16ObjIdB`	`Q_BackgroundColour` (normales Objekt)

Allgemeines Muster (Selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`) siehe Background-Farbbausteine (gemeinsames Muster).



Initialisierung und Qualität

- **Ereignis-Eingang `INIT`** (`EInit`) und **Ereignis-Ausgang `INITO`** (`EInit`): die Initialisierung wird durch die `Q_BackgroundColour`-Instanzen gekettet: `INIT` ⟶ `Q_BackgroundColour_1.INIT`; `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` ⟶ `Q_BackgroundColour_3.INIT`; `Q_BackgroundColour_3.INITO` ⟶ `Q_BackgroundColour_4.INIT`; `Q_BackgroundColour_4.INITO` ⟶ `INITO`.
- **Dateneingang `QI`** (`BOOL`, Vorgabe `TRUE`) wird an `.QI` aller Instanzen verteilt (`Q_BackgroundColour_1`, `Q_BackgroundColour_2`, `Q_BackgroundColour_3`, `Q_BackgroundColour_4`).
- **Datenausgang `QO`** (`BOOL`): UND aller Instanzen über `AND_QO` (`iec61131::booleanOperators::AND_BOOL_4`): `Q_BackgroundColour_1.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN2`, `Q_BackgroundColour_3.QO` ⟶ `AND_QO.IN3`, `Q_BackgroundColour_4.QO` ⟶ `AND_QO.IN4`; `AND_QO.OUT` ⟶ `QO`.
- `E_MERGE_QO_A` (`iec61499::events::E_MERGE_4`) vereint `Q_BackgroundColour_4.INITO` und die `CNF`-Ereignisse der Instanzen 1 bis 3; `E_MERGE_QO` (`iec61499::events::E_MERGE_2`) vereint dessen Ausgang `E_MERGE_QO_A.EO` mit `Q_BackgroundColour_4.CNF` und löst damit `AND_QO.REQ` aus.

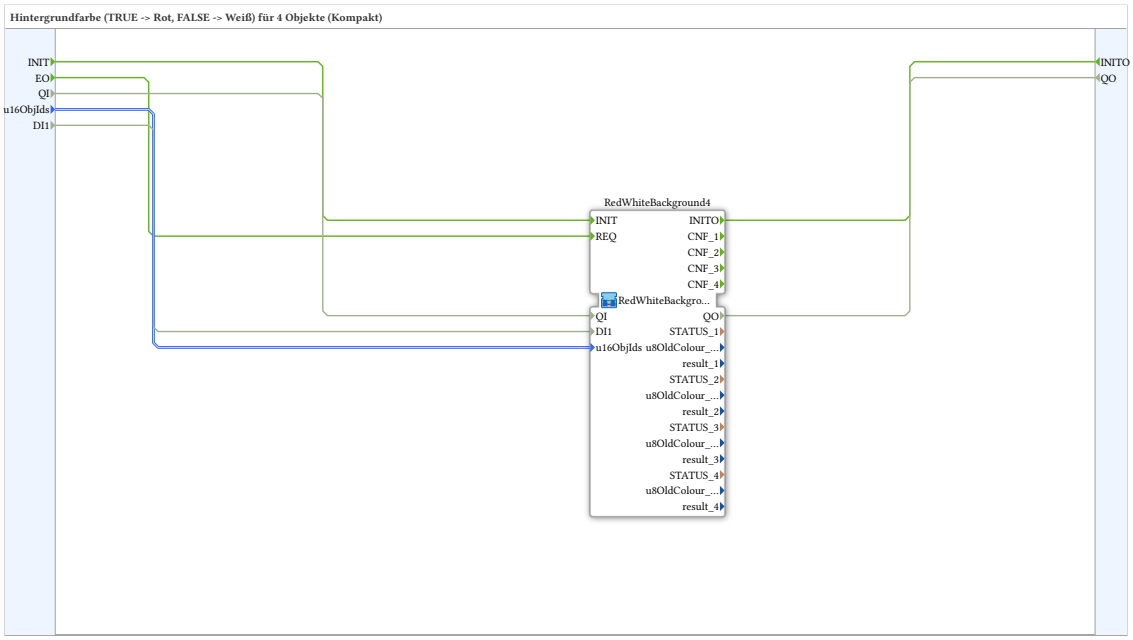
Zusammenfassung

Eine von vielen Varianten der Background-Farbbausteine-Familie: Farbpaar Rot/Weiß, 4 Objekte, BOOL-Selector, struct. Objekt-ID.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RedWhiteBackground4SC



• – ○ • – ○ • – ○ •

Einleitung

`RedWhiteBackground4SC` ist ein **Kompakt-Wrapper** um `RedWhiteBackground4S` — reicht `u16ObjIds` und den Selector durch, ohne die `CNF`-Ereignisausgänge und Diagnosewerte (`STATUS_n`, `u8OldColour_n`, `result_n`) nach außen zu führen.

Schnittstelle (Quality-Muster): Der Wrapper besitzt zusätzlich den Ereignis-Eingang `INIT` und den Ereignis-Ausgang `INITO` (je `EInit`) sowie den Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`) und den Datenausgang `QO` (`BOOL`). Sie sind direkt an die innere Instanz `RedWhiteBackground4` durchgeschleift: `INIT` ⟶ `RedWhiteBackground4.INIT`, `RedWhiteBackground4.INITO` ⟶ `INITO`, `QI` ⟶ `RedWhiteBackground4.QI`, `RedWhiteBackground4.QO` ⟶ `QO`.

Details zum gemeinsamen Muster siehe Background-Farbbausteine (gemeinsames Muster).

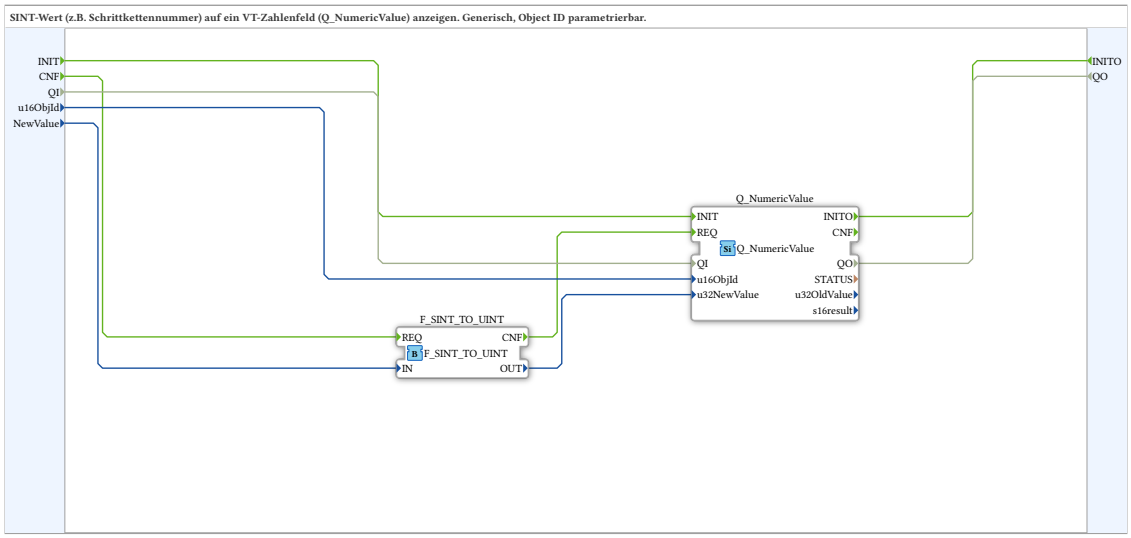
Zusammenfassung

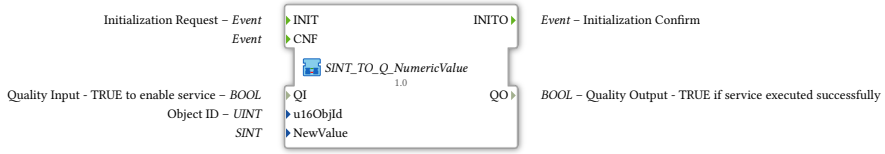
Reine Weiterleitung an `RedWhiteBackground4S` für den Regelfall, in dem die Diagnosewerte nicht gebraucht werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SINT_TO_Q_NumericValue





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **SINT_TO_Q_NumericValue** ist eine wiederverwendbare Subapplikation, die einen SINT-Wert (z. B. eine Schrittkettennummer) auf einem Bediengerät (VT) in einem Zahlenfeld anzeigt. Die Objekt-ID des Ziel-Zahlenfeldes ist parametrierbar und wird über den Eingang `u16ObjId` zugeführt. Intern erfolgt eine Typkonvertierung von SINT nach UINT, um die Anforderungen des verwendeten Anzeigebausteins `Q_NumericValue` zu erfüllen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Datentyp	Beschreibung
`INIT`	EInit	Initialisierungsanforderung; wird an `Q_NumericValue.INIT` weitergeleitet.
`CNF`	Event	Externer Trigger, der die Verarbeitung startet. Beim Eintreten wird der Wert `NewValue` konvertiert und an das Zahlenfeld übertragen.

Ereignis-Ausgänge

Name	Datentyp	Beschreibung
`INITO`	EInit	Initialisierungsbestätigung von `Q_NumericValue.INITO`.

Daten-Eingänge

Name	Datentyp	Beschreibung
`QI`	BOOL	Quality Input – TRUE aktiviert den Dienst (Standardwert `TRUE`); wird an `Q_NumericValue.QI` durchgereicht.
`u16ObjId`	UINT	Objekt-ID des Ziel-Zahlenfeldes. Standardwert ist `ID_NULL`, kann aber über diesen Eingang überschrieben werden.

NewValue	SINT	Der anzuzeigende Wert, der vor der Übertragung nach UINT konvertiert wird.

Daten-Ausgänge

Name	Datentyp	Beschreibung
QO	BOOL	Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde; kommt von Q_NumericValue.QO.

Adapter

Keine.

Funktionsweise

Die Subapplikation arbeitet ereignisgesteuert. Das Ereignis INIT wird an Q_NumericValue.INIT weitergereicht, dessen INITO erscheint am Ausgang INITO. QI wird an Q_NumericValue.QI und Q_NumericValue.QO an QO durchgereicht. Ein externes Ereignis am Eingang CNF löst die Verarbeitung aus:

1. Das Ereignis wird an den Funktionsblock F_SINT_TO_UINT (aus der Bibliothek iec61131::conversion) weitergeleitet.
2. Gleichzeitig wird der aktuelle Wert von NewValue als Eingang IN an den Konvertierungsbaustein übergeben.
3. Nach Abschluss der Konvertierung signalisiert F_SINT_TO_UINT über seinen Ausgang CNF fertig.
4. Dieses Ereignis triggert den Baustein Q_NumericValue (aus isobus::UT::Q) mit dessen Eingang REQ.
5. Der konvertierte Wert (UINT) wird über den Ausgang OUT von F_SINT_TO_UINT an den Eingang u32NewValue von Q_NumericValue gelegt.
6. Die Objekt-ID u16ObjId wird direkt (ohne Änderung) an den entsprechenden Eingang u16ObjId von Q_NumericValue durchgereicht.

Somit wird der SINT-Wert in ein geeignetes Format umgewandelt und auf dem virtuellen Terminal angezeigt.

Technische Besonderheiten

- Die Typkonvertierung nutzt den standardisierten FB F_SINT_TO_UINT aus der IEC-61131-Bibliothek, was eine robuste und normgerechte Umwandlung gewährleistet.
- Der Anzeigebaustein Q_NumericValue stammt aus der ISOBUS-UT-Bibliothek und erwartet einen 32-Bit-Wert; die Subapp übernimmt die notwendige Vorbereitung.
- Die Objekt-ID u16ObjId ist als UINT deklariert und mit dem Symbol ID_NULL vorbelegt, was eine einfache Parametrierung ermöglicht.
- Die Verbindung von u16ObjId zu Q_NumericValue ist als unsichtbar markiert, um die Übersichtlichkeit im Netzwerk zu erhöhen – sie wird aber dennoch vollständig ausgeführt.

Zustandsübersicht

Die Subapp besitzt keinen expliziten Zustandsautomaten, da sie rein ereignisgesteuert arbeitet. Der Ablauf ist sequenziell:

- **Idle:** Warten auf ein Ereignis an CNF (nach der Initialisierung über INIT/INITO).
- **Konvertierung:** NewValue wird von SINT nach UINT konvertiert.
- **Ausgabe:** Q_NumericValue wird mit dem konvertierten Wert und der Objekt-ID angesteuert.

Nach Abschluss der Ausgabe kehrt die Subapp in den idle-Zustand zurück und kann erneut getriggert werden.

Anwendungsszenarien

Typischer Einsatz ist die Visualisierung von numerischen Werten auf einem ISOBUS-Terminal (VT), z. B.:

- Anzeige der aktuellen Schrittkettennummer in einer Maschinensteuerung.
- Darstellung von SINT-basierten Prozesswerten (Geschwindigkeit, Temperatur, etc.) auf einem Bediendisplay.
- Wiederverwendbare Komponente in größeren Applikationen, bei denen mehrere solche Anzeigen mit unterschiedlichen Objekt-IDs benötigt werden.

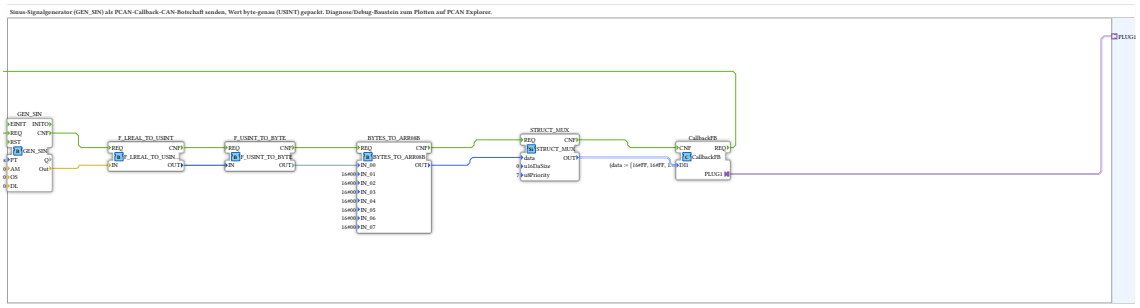
Vergleich mit ähnlichen Bausteinen

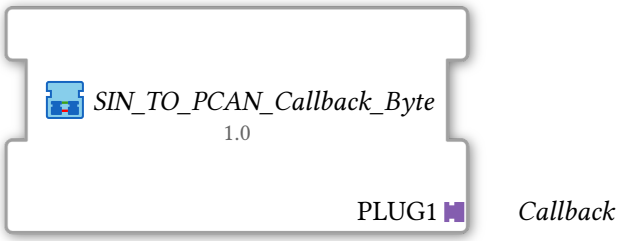
Andere Bausteine zur Zahlenanzeige existieren häufig in direkter Form (z.B. `Q_NumericValue` selbst), erfordern aber bereits einen UINT-/DINT-Wert. Die Subapp `SINT_TO_Q_NumericValue` ergänzt die fehlende Typanpassung und kapselt die Konvertierung sowie die Ansteuerung des Zielbausteins. Gegenüber einer direkten Verwendung von `F_SINT_TO_UINT` und `Q_NumericValue` in einer Applikation bietet sie eine höhere Wiederverwendbarkeit und reduziert die Verdrahtung.

Fazit

Die Subapp `SINT_TO_Q_NumericValue` ist eine kompakte, generische Lösung zur Anzeige von SINT-Werten auf einem ISOBUS-VT. Sie kombiniert eine standardisierte Typkonvertierung mit dem etablierten Anzeigebaustein `Q_NumericValue` und ist durch die parametrierbare Objekt-ID flexibel einsetzbar. Die klare Schnittstelle und die einfache Logik machen sie zu einer sinnvollen Komponente in Automatisierungsapplikationen.

SIN_TO_PCAN_Callback_Byte





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **SIN_TO_PCAN_Callback_Byte** erzeugt ein sinusförmiges Signal und sendet dessen aktuellen Wert als CAN-Botschaft über eine PCAN-Callback-Schnittstelle. Er ist als Diagnose- und Debug-Baustein konzipiert, um Messwerte direkt in einem PCAN-Explorer darzustellen. Der erzeugte Sinuswert wird byte-genau in ein USINT-Format umgewandelt und als erstes Byte einer 8-Byte-Nachricht übertragen.

Schnittstellenstruktur

Die Subapp besitzt ausschließlich eine **Adapter-Schnittstelle** als externe Anbindung. Es existieren keine separaten Ereignis- oder Datenein-/ausgänge – die gesamte Kommunikation erfolgt über den Adapter.

Ereignis-Eingänge

– Keine

Ereignis-Ausgänge

– Keine

Daten-Eingänge

– Keine

Daten-Ausgänge

– Keine

Adapter

Name	Typ	Richtung
PLUG1	isobus::pgn::tx::Callback	Ausgang (Plug)

Der Adapter stellt eine Callback-Schnittstelle für das Senden von CAN-Nachrichten zur Verfügung. Über ihn wird das Senden ausgelöst und die Nachricht an die externe Umgebung übergeben.

Funktionsweise

Der Baustein arbeitet ereignisgesteuert. Ein externes Triggerereignis (über den Adapter) startet den Ablauf:

1. **Ereignisauslösung** → `CallbackFB.REQ` aktiviert den Sinusgenerator `GEN_SIN`.
2. **Signalgenerierung** → `GEN_SIN` berechnet einen neuen Sinuswert basierend auf den Parametern (Amplitude, Offset, Periodendauer).
3. **Konvertierung** → Der LREAL-Wert wird zuerst in `USINT` umgewandelt (`F_LREAL_TO_USINT`) und danach in ein Byte (`F_USINT_TO_BYTE`).
4. **Byte-Array-Aufbau** → `BYTES_TO_ARR08B` setzt das erhaltene Byte an Position 0 eines 8-Byte-Arrays; die restlichen Bytes bleiben auf `16#00`.
5. **CAN-Nachricht erzeugen** → `STRUCT_MUX` packt das Byte-Array in eine CAN_MSG-Struktur (Priorität `7`, Datengröße `0`).
6. **Senden** → Der `CallbackFB` sendet die fertige Nachricht über den Adapter `PLUG1` nach außen.

Der Zyklus wiederholt sich bei jedem neuen Ereignis, sodass der Sinuswert kontinuierlich abgetastet und gesendet wird.

Technische Besonderheiten

- **Byte-genaue Übertragung:** Der Sinuswert wird über `F_LREAL_TO_USINT` und `F_USINT_TO_BYTE` exakt in ein einzelnes Byte umgewandelt und im ersten Byte der CAN-Nachricht platziert.
- **Einsatz eines Callback-Adapters:** Die Subapp nutzt ausschließlich eine Adapter-Schnittstelle, was eine flexible Einbindung in bestehende PCAN-Kommunikationssysteme ermöglicht.
- **Konfigurierbare Signalparameter:** Der Sinusgenerator besitzt Parameter für Amplitude (`AM=10.0`), Offset (`OS=5.0`), Periodendauer (`PT=10s`) und Verzögerung (`DL=0.0`). Dadurch kann das Signal an verschiedene Anforderungen angepasst werden.
- **Direkte Strukturerzeugung:** Über `STRUCT_MUX` wird die CAN-Nachricht als strukturierter Datentyp `CAN_MSG` mit vordefinierter Priorität und Datengröße erzeugt.
- **Debug-/Plot-Eignung:** Die Daten sind als rohe Bytes aufbereitet, sodass sie von Analysewerkzeugen wie PCAN-Explorer unmittelbar interpretiert werden können.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten. Er arbeitet rein ereignisgesteuert: Jedes eingehende Ereignis am `CallbackFB.REQ` (über den Adapter) führt zu einem vollständigen Durchlauf der Signalverarbeitungskette und anschließendem Senden der Nachricht. Es gibt keine internen Wartezustände oder Verzögerungen außer der vom Sinusgenerator vorgegebenen Periodendauer (der jedoch nur bei jedem Trigger einen neuen Wert liefert).

Anwendungsszenarien

- **Diagnose & Debug:** Darstellung eines kontinuierlich generierten Sinus-Signals in PCAN-Explorer zur Überwachung der CAN-Kommunikation.
- **Test von CAN-Knoten:** Erzeugen eines definierten Signalverlaufs zum Prüfen von Empfängern oder der Buslast.
- **Lehre und Ausbildung:** Demonstration der Verarbeitung eines Analogsignals zu einer CAN-Nachricht unter Verwendung standardisierter Umwandlungs- und Strukturierungsblöcke.

Vergleich mit ähnlichen Bausteinen

Es existieren andere Signalgenerator-Bausteine, die direkt CAN-Nachrichten senden (z.B. mit integrierter ID- oder DLC-Verwaltung). Der vorliegende Baustein zeichnet sich jedoch durch folgende Besonderheiten aus:

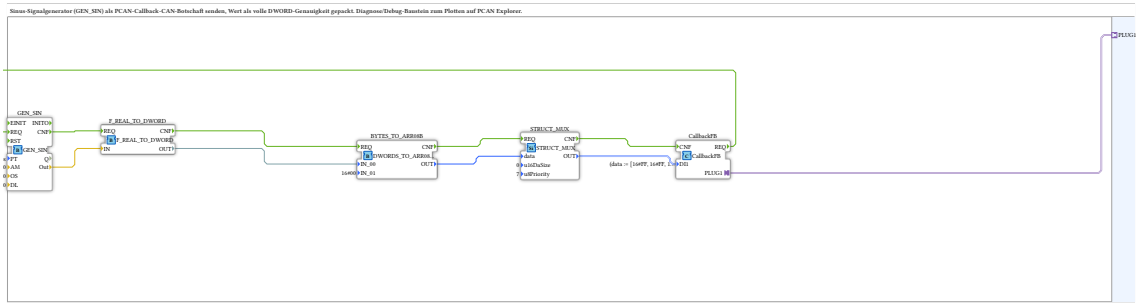
- **Fokussierung auf ein einzelnes Datenbyte** – reduziert die Komplexität und eignet sich für schmale Signalwerte.

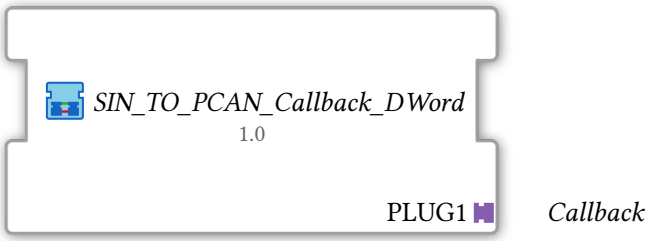
- **Adapterbasiertes Interface** – erlaubt eine einfache Integration in Systeme, die bereits einen Callback-Mechanismus verwenden.
- **Kombination aus Standard-Blöcken** (GEN_SIN, Konvertierung, Array-Bildung) ermöglicht eine transparente und erweiterbare Struktur, die leicht an andere Signaltypen angepasst werden kann.

Fazit

Der Funktionsbaustein **SIN_TO_PCAN_Callback_Byte** stellt eine kompakte und effiziente Lösung dar, um ein analoges Sinus-Signal als CAN-Botschaft mit minimalem Aufwand zu senden. Durch die klare Trennung von Signalgenerierung, Konvertierung und CAN-Aufbereitung ist er gut verständlich, erweiterbar und besonders für diagnostische Zwecke geeignet. Die ausschließliche Verwendung einer Adapter-Schnittstelle macht ihn flexibel einsetzbar und erleichtert die Integration in bestehende Systeme der Automatisierungstechnik.

SIN_TO_PCAN_Callback_DWord





• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **SIN_TO_PCAN_Callback_DWord** ist eine Subapplication, die ein intern erzeugtes Sinussignal als CAN-Botschaft über einen sogenannten **Callback-Adapter** an ein externes System (z. B. PCAN Explorer) sendet. Er dient als Diagnose- und Debug-Hilfe, um Signalverläufe direkt im CAN-Tool zu plotten. Der Sinuswert wird dabei mit voller **32-Bit-DWORD-Genauigkeit** in eine CAN-Nachricht verpackt und über standardisierte ISO-bus-PGN-Strukturen übertragen.

Die Subapplikation besitzt keine direkten Ein- oder Ausgänge; die gesamte Kommunikation erfolgt über einen einzigen Adapter (`PLUG1`) vom Typ `isobus::pgn::tx::Callback`. Dadurch ist sie als abgeschlossene, wiederverwendbare Funktionseinheit konzipiert, die sich einfach in bestehende CAN-Kommunikationsarchitekturen integrieren lässt.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Der Baustein wird ausschließlich über den angeschlossenen Adapter durch externe Ereignisse (z. B. Trigger des Callback-Dienstes) aktiviert.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge. Die interne Verarbeitung und das Senden der Botschaft erfolgt über den Adapter; ein explizites Quittierungssignal wird nicht nach außen geführt.

Daten-Eingänge

Keine direkten Daten-Eingänge. Alle Parameter (Amplitude, Offset, Periodendauer) sind fest im Netzwerk voreingestellt und nicht von außen veränderbar.

Daten-Ausgänge

Keine direkten Daten-Ausgänge. Die erzeugte CAN-Botschaft wird über den Adapter gesendet und nicht als separates Datenwort bereitgestellt.

Adapter

- **PLUG1** – Typ: `isobus::pgn::tx::Callback`
  Dieser Adapter (Stecker) stellt die Schnittstelle zum Senden von CAN-Botschaften bereit. Er wird mit einer externen Instanz verbunden, die den Callback-Mechanismus des ISO-bus-PGN-Protokolls implementiert (z. B. ein PCAN-Empfänger).

Funktionsweise

Die Subapplikation implementiert eine ereignisgesteuerte Datenkette:

1. **Auslösung:** Ein externes Ereignis (über den Adapter) erreicht den Baustein `CallbackFB`, der daraufhin das Signal `REQ` an den Sinusgenerator `GEN_SIN` sendet.
2. **Signalerzeugung:** `GEN_SIN` erzeugt mit den festen Parametern
   - Periodendauer `PT = 10 s`,
   - Amplitude `AM = 10.0`,
   - Offset `OS = 5.0`,
   - Verzögerung `DL = 0.0`
   einen kontinuierlichen Sinuswert und liefert diesen über das Ereignis `CNF` an die nächste Stufe.
3. **Konvertierung:** Der reelle Sinuswert wird vom Baustein `F_REAL_TO_DWORD` in eine 32-Bit-DWORD-Ganzzahl umgewandelt.
4. **Byte-Reihenfolge:** `DWORDS_TO_ARR08B` (vom Typ `logiBUS::utils::conversion::arr::reversing`) zerlegt das DWORD in ein Byte-Array und legt die Byte-Reihenfolge so fest, dass die Daten für den CAN-Bus korrekt interpretiert werden können (Little-Endian bzw. Big-Endian je nach Konfiguration; der Parameter `IN_01` zeigt die Startbelegung).
5. **Verpackung:** `STRUCT_MUX` (vom Typ `eclipse4diac::convert::STRUCT_MUX`) setzt die Bytes in die Struktur `isobus::pgn::CAN_MSG` ein. Dabei werden die Attribute `u8Priority` (=7) und `u16DaSize` (=0) gesetzt.
6. **Versand:** Die fertige CAN_Nachricht wird über `CallbackFB` an den Adapter `PLUG1` übergeben und von dort an das angeschlossene System (z. B. PCAN Explorer) gesendet.

Die Ereignis- und Datenverbindungen sind so verdrahtet, dass jeder Schritt nur nach erfolgreichem Abschluss des vorherigen ausgeführt wird (Kette: REQ → GEN_SIN → CNF → F_REAL_TO_DWORD → CNF → BYTES_TO_ARR08B → CNF → STRUCT_MUX → CNF → CallbackFB).

Technische Besonderheiten

- **Volle DWORD-Genauigkeit:** Der Sinuswert wird nicht in ein kleineres Format (z. B. Integer mit Skalierung) umgewandelt, sondern als 32-Bit-Rohtyp übertragen, wodurch Nachkommastellen und Dynamik ohne Informationsverlust erhalten bleiben.
- **Feste Parameter:** Die Parameter des Sinusgenerators sind hart im Netzwerk voreingestellt und können nur durch Änderung der Verbindungswerte angepasst werden. Dies macht den Baustein für schnelle Debug-Tests ohne Neukonfiguration geeignet.
- **ISO-bus-PGN-Standard:** Die CAN-Nachricht wird in der standardisierten Struktur `isobus::pgn::CAN_MSG` abgelegt, was die Kompatibilität mit ISO-bus-basierten Systemen gewährleistet.
- **Reversierende Byte-Anordnung:** Der Baustein `DWORDS_TO_ARR08B` kehrt die Reihenfolge der Bytes um, um den gebräuchlichen Big-Endian-Format auf dem CAN-Bus zu entsprechen (konfigurierbar).
- **Callback-Mechanismus:** Die Übertragung erfolgt über den Adaptertyp `isobus::pgn::tx::Callback`, der eine ereignisgesteuerte, asynchrone Datenübergabe unterstützt.

Zustandsübersicht

Die Subapplikation besitzt keinen expliziten Zustandsautomaten. Der Ablauf ist rein ereignisgesteuert und folgt der fest verdrahteten Kette der enthaltenen Funktionsblöcke. Es lassen sich vier logische Phasen unterscheiden:

1. **Bereit** – Warten auf ein Trigger-Ereignis vom Adapter.
2. **Signalerzeugung** – `GEN_SIN` berechnet den nächsten Sinuswert.
3. **Konvertierung & Verpackung** – Der Wert wird in DWORD, dann in Bytes und schließlich in die CAN_Msg-Struktur gewandelt.
4. **Sendung** – Die fertige Nachricht wird über den Adapter nach außen gegeben.

Diese Phasen wiederholen sich zyklisch, solange externe Trigger eintreffen.

Anwendungsszenarien

- **Debugging und Visualisierung:** Der Sinuswert wird auf einem PCAN-Explorer-Plot dargestellt, um die korrekte Übertragung von Gleitkommawerten über CAN zu verifizieren.
- **Test von CAN-Konfigurationen:** Die Subapplikation dient als einfacher Signalgenerator für die Überprüfung der CAN-Bus-Verschaltung und der Empfänger-Logik.
- **Referenz für Datenkonvertierung:** Sie demonstriert die Kette von REAL → DWORD → Byte-Array → CAN_Msg und kann als Vorlage für eigene Konvertierungen dienen.
- **Integration in ISO-bus-Systeme:** Dank der standardisierten PGN-Struktur kann sie direkt in bestehende ISOBUS-Kommunikationsframeworks eingebunden werden.

Vergleich mit ähnlichen Bausteinen

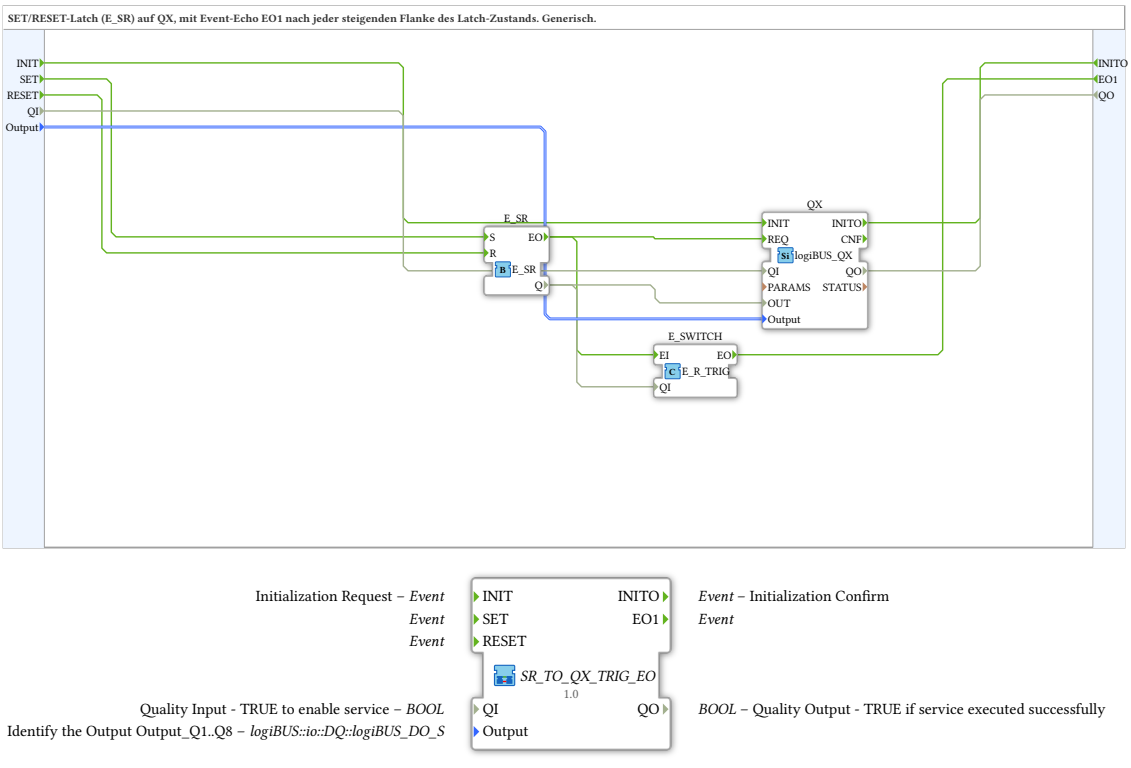
- **SIN_TO_PCAN_Callback_Real:** Eine mögliche Alternative, die den REAL-Wert direkt (ohne DWORD-Konvertierung) sendet – dadurch geringere Genauigkeit.
- **SIN_TO_PCAN_Callback_Word:** Würde nur ein 16-Bit-Wort übertragen, was den Wertebereich einschränkt.
- **Statische CAN-Sender:** Andere Bausteine, die feste Nachrichten senden, bieten keine dynamische Signalgenerierung.
- **Externe Signalgeneratoren:** Beispielsweise ein MATLAB-Skript, das CAN-Nachrichten erzeugt, erfordert zusätzliche Hardware und Software; dieser Baustein ist eine reine PLC-interne Lösung.

Der vorliegende Baustein kombiniert die Vorteile eines flexiblen Sinusgenerators mit einer verlustfreien 32-Bit-Übertragung und der Integration in den ISO-bus-Standard.

Fazit

SIN_TO_PCAN_Callback_DWord ist ein kompakter, wiederverwendbarer Baustein, der ohne externen Konfigurationsaufwand eine hochaufgelöste Sinusschwingung über CAN ausgibt. Durch die Verwendung eines Adapters vom Typ `isobus::pgn::tx::Callback` bleibt die Schnittstelle schlank und standardisiert. Die festen Parameter ermöglichen einen sofortigen Einsatz zu Test- und Debugzwecken, während die interne Struktur gut nachvollziehbar ist. Der Baustein ist ideal für Entwickler, die schnell einen Signalverlauf auf einem CAN-Analysewerkzeug darstellen möchten, ohne eine separate Signalquelle zu benötigen.

SR_TO_QX_TRIG_EO



Einleitung

Der Funktionsblock **SR_TO_QX_TRIG_EO** ist eine generische Subapplikation, die einen digitalen Ausgang über einen logiBUS-Baustein (QX) ansteuert und dabei eine steigende Flanke des internen Latch-Zustands als Ereignis am Ausgang `EO1` signalisiert. Er kombiniert einen SR-Latch (Set/Reset), einen Ausgangstreiber für die logiBUS-Plattform und eine Flankenerkennung. Dadurch eignet er sich für Anwendungen, bei denen ein Ausgang gesetzt oder zurückgesetzt wird und gleichzeitig eine Meldung über das aktive Setzen (steigende Flanke) benötigt wird.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Beschreibung
`INIT`	Initialization Request (`EInit`); wird an `QX.INIT` durchgereicht.
`SET`	Setzt den internen Latch-Zustand auf `TRUE` und löst damit die Ausgabe am QX aus.
`RESET`	Setzt den internen Latch-Zustand auf `FALSE` und deaktiviert den Ausgang.

Ereignis-Ausgänge

Name	Beschreibung
`INITO`	Initialization Confirm (`EInit`); durchgereicht von `QX.INITO`.
`EO1`	Wird ausgelöst, wenn der Latch-Zustand von `FALSE` auf `TRUE` wechselt (steigende Flanke).

Daten-Eingänge

Name	Datentyp	Beschreibung
`QI`	`BOOL`	Quality Input - TRUE, um den Dienst freizugeben. Initialwert: `TRUE`; wird an `QX.QI` durchgereicht.
`Output`	`logiBUS::io::DQ::logiBUS_DO_S`	Identifiziert den physischen Ausgang (z. B. `Output_Q1` bis `Output_Q8`). Initialwert: `logiBUS_DO::Invalid`.

Daten-Ausgänge

Name	Datentyp	Beschreibung
`QO`	`BOOL`	Quality Output - TRUE, wenn der Dienst erfolgreich ausgeführt wurde; durchgereicht von `QX.QO`.

Adapter

Keine.

Funktionsweise

Die Subapplikation besteht aus drei internen Bausteinen:

1. **E_SR** (SR-Latch): Speichert den Zustand, der durch die Events `SET` und `RESET` gesetzt bzw. zurückgesetzt wird.
2. **logiBUS_QX**: Schreibt den aktuellen Latch-Zustand (`Q`) auf den durch `Output` spezifizierten digitalen Ausgang.
3. **E_R_TRIG** (steigende Flankenerkennung): Überwacht das Signal `Q` und erzeugt bei einer Änderung von `FALSE` auf `TRUE` das Ausgangsereignis `EO1`.

Der Ablauf:

- Ein `SET`-Event setzt den Latch (`E_SR.S`), woraufhin `E_SR` das Ausgangsereignis `EO` auslöst. Gleichzeitig wird der Datenwert `Q` auf `TRUE` gesetzt.
- Das `EO`-Signal aktiviert sowohl den `QX` (um den Ausgang zu schreiben) als auch den `E_R_TRIG`. Der `E_R_TRIG` prüft die steigende Flanke von `Q` und generiert bei Bedarf `EO1`.

- Ein RESET-Event setzt den Latch zurück (E_SR.R), wodurch Q auf FALSE wechselt. Auch hier wird QX aktualisiert, aber keine steigende Flanke erkannt, sodass EO1 nicht ausgelöst wird.

Technische Besonderheiten

- Die Subapplikation nutzt die parametrisierbare Datenstruktur logiBUS_DO_S, um den Zielausgang flexibel zu wählen. Der Initialwert verhindert eine ungewollte Ausgabe.
- Die Verbindung zwischen E_SR.EO und E_SWITCH.EI (intern E_R_TRIG) ist so ausgelegt, dass die Flankenerkennung nur bei Zustandsänderungen des Latches ausgewertet wird – nicht bei jedem Clock.
- Der Ausgangsbaustein QX erhält QI vom QI-Eingang der Subapplikation (Initialwert TRUE); INIT wird an QX.INIT und QX.INITO an INITO durchgereicht, QX.QO an QO.
- Durch die Auslagerung als Subapplikation wird eine Wiederverwendung in verschiedenen Projekten ermöglicht, ohne den internen Aufbau erneut erstellen zu müssen.

Zustandsübersicht

Die Subapplikation besitzt implizit den Zustand des internen SR-Latches (Q). Dargestellt als Tabelle:

Zustand von Q	Bedingung	Aktivitäten
FALSE	Nach Initialisierung oder nach RESET	Ausgang ist inaktiv; bei SET wechselt Zustand auf TRUE und EO1 wird ausgelöst.
TRUE	Nach SET	Ausgang ist aktiv; bei RESET wechselt Zustand auf FALSE. Eine erneutes SET ohne zwischenzeitliches RESET bewirkt keine Flanke, also kein EO1 (da Q bereits TRUE ist).

Anwendungsszenarien

- **Ansteuerung eines logiBUS-Ausgangs mit Rückmeldung**: Wenn ein Ausgang gesetzt wird, soll eine Verarbeitung (z. B. in einer übergeordneten Steuerung) über das Setzen informiert werden.
- **Aufzug- oder Fördertechnik**: Signalisiert das Aktivieren eines Stellantriebs (Setzen) und meldet das erstmalige Aktivieren als Ereignis.
- **Prozesssteuerung**: Kombination von Ausgangsschalten und Flankenerkennung in einem einzigen Baustein reduziert die Komplexität des Netzwerks.

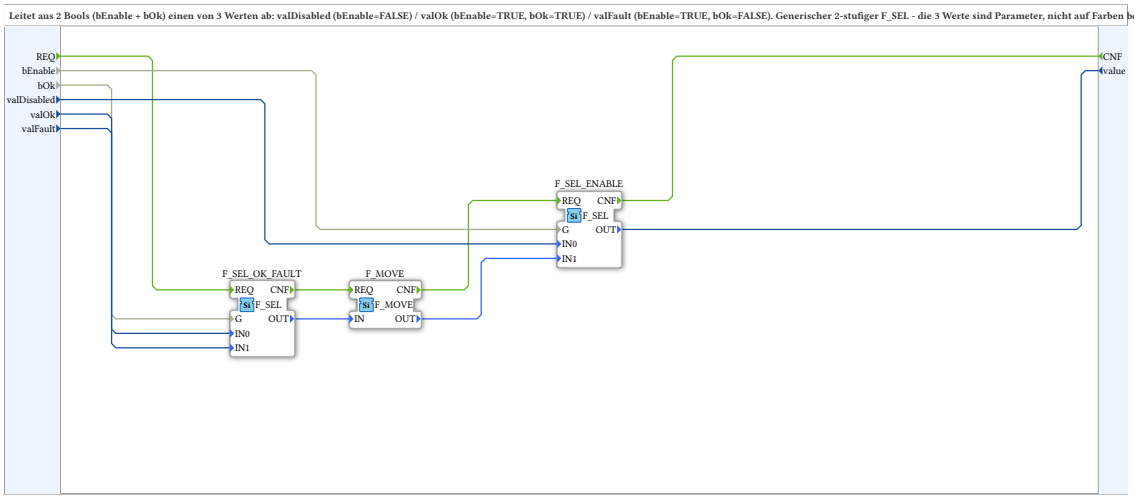
Vergleich mit ähnlichen Bausteinen

- **SR_TO_QX** (ohne Trigger): Dieser Baustein würde nur den Ausgang setzen/zurücksetzen, aber kein Ereignis bei steigender Flanke ausgeben.
- **E_SR mit EO** (Standard-Event-SR): Liefert das Ergebnis als Datenwert, muss aber extern mit einem Flankenerkenner verbunden werden – hier ist die Kombination bereits integriert.
- **SR_TO_QX_TRIG**: Ähnlich, aber ohne das Echo-Ereignis EO1; hier wird die steigende Flanke direkt als Event geliefert.

Fazit

Der Funktionsblock **SR_TO_QX_TRIG_EO** bietet eine kompakte und wiederverwendbare Lösung für die Ansteuerung digitaler Ausgänge mit integrierter Flankenerkennung. Er vereinfacht das Engineering, reduziert die Anzahl notwendiger Bausteine und erhöht die Übersichtlichkeit in Speicherprogrammierbaren Steuerungen (SPS). Dank der generischen Ausgangsauswahl ist er flexibel einsetzbar und eignet sich besonders für Anwendungen, die sowohl eine stabile Ausgangssteuerung als auch eine zuverlässige Ereignismeldung benötigen.

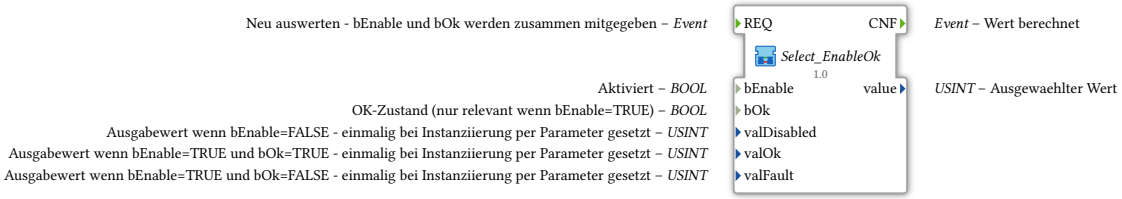
Select_EnableOk



• — ○ • — ○ • — ○ •

Einleitung

`Select_EnableOk` leitet aus 2 Bools (`bEnable`, `bOk`) einen von 3 parametrierten USINT-Werten ab: `valDisabled` (bEnable=FALSE), `valOk` (bEnable=TRUE und bOk=TRUE), `valFault` (bEnable=TRUE und bOk=FALSE). Generischer 2-stufiger `F_SEL`, nicht auf Farben beschraenkt - laut Quellcode-Kommentar aus `Q_BackgroundColour_EnableOk` extrahiert, um die reine Auswahllogik von der VT-Hintergrundfarben-Anwendung zu entkoppeln.



Verwendete Funktionsbausteine (FBs)

- **F_SEL_OK_FAULT** (`iec61131::selection::F_SEL`): waehlt zwischen `valFault`/`valOk` nach `bOk`.
- **F_MOVE** (`iec61131::selection::F_MOVE`, `DataType=USINT`): reicht den Zwischenwert unveraendert weiter (Ablauf-Kopplung der beiden Selektionsstufen).
- **F_SEL_ENABLE** (`iec61131::selection::F_SEL`): waehlt zwischen `valDisabled`/Zwischenergebnis nach `bEnable`.

Programmablauf und Verbindungen

`REQ -> F_SEL_OK_FAULT.REQ -> F_MOVE.REQ -> F_SEL_ENABLE.REQ -> CNF`. Datenseitig: `bOk`/`valFault`/`valOk -> F_SEL_OK_FAULT -> F_MOVE -> F_SEL_ENABLE.IN1`, parallel `bEnable`/`valDisabled -> F_SEL_ENABLE.IN0`/`.G -> value`.

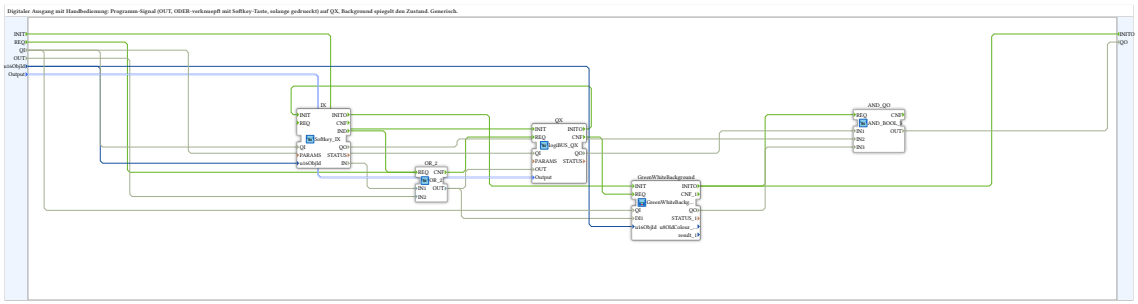
Zusammenfassung

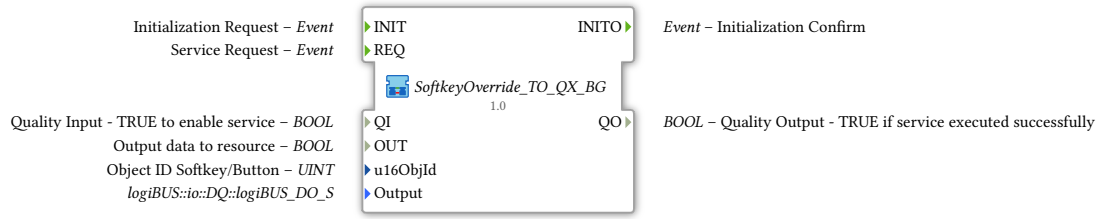
Generische 2-Bool-zu-1-von-3-Werten-Auswahl, wiederverwendbar ueber Hintergrundfarben hinaus (z. B. fuer beliebige Enable/Ok-Statusanzeigen).

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SoftkeyOverride_TO_QX_BG





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **SoftkeyOverride_TO_QX_BG** realisiert einen digitalen Ausgang mit manueller Bedienmöglichkeit über eine Softkey-Taste. Das Programm-Signal `OUT` wird dabei per ODER-Verknüpfung mit dem Zustand der Softkey-Taste kombiniert und auf den physikalischen Ausgang `QX` ausgegeben. Ein Hintergrundobjekt (GreenWhiteBackground) spiegelt den aktuellen Zustand des Ausgangs wider. Der Baustein ist generisch einsetzbar und wurde zur Wiederverwendung aus einer bestehenden Übungsstruktur ausgelagert.

Schnittstellenstruktur

Ereignis-Eingänge

- `INIT` (EInit) – Initialisierungsanforderung: startet die Kette `QX` ⟶ `IX` ⟶ `GreenWhiteBackground`.
- `REQ` (Event) – Service Request: Triggert die Verarbeitung der Eingangsdaten und die Aktualisierung des Ausgangs.

Ereignis-Ausgänge

- `INITO` (EInit) – Initialisierungsbestätigung: kommt von `GreenWhiteBackground.INITO`.
- Darüber hinaus gibt es keine Ereignis-Ausgänge; der Abschluss der Verarbeitung wird intern über `QX.CNF` an den Hintergrundbaustein weitergeleitet.

Daten-Eingänge

- `QI` (BOOL) – Quality Input, Freigabe des Dienstes (Initialwert `TRUE`); wird an `QX`, `IX` und `GreenWhiteBackground` weitergereicht.
- `OUT` (BOOL) – Programmsignal für den Ausgang.
- `u16ObjId` (UINT) – Objekt-ID der Softkey-Taste (Initialwert `ID_NULL`).
- `Output` (logiBUS::io::DQ::logiBUS_DO_S) – Konfigurationsparameter für den logiBUS-Ausgang (Initialwert `logiBUS_DO::Invalid`).

Daten-Ausgänge

- `QO` (BOOL) – Quality Output, `AND` der `QO`-Ausgänge von `QX`, `IX` und `GreenWhiteBackground` (über `AND_QO`, `AND_BOOL_3`). Der eigentliche Ausgang erfolgt über den eingebetteten `QX`-Funktionsblock auf den physischen Kanal.

Adapter

- Keine (Es sind keine Adapter-Schnittstellen vorhanden.)

Funktionsweise

Der Baustein kombiniert das Programm-Signal `OUT` mit dem Zustand einer Softkey-Taste (`IX.IN`). Die ODER-Verknüpfung (`OR_2`) erzeugt das effektive Ausgangssignal:

- `OR_2.OUT` wird `TRUE`, wenn entweder `OUT` oder die Softkey-Taste (`IX.IN`) aktiv ist.
- Dieses Signal wird an den Ausgangsbaustein `QX` übergeben, der den physischen Ausgang setzt.
- Parallel wird dasselbe Signal an den Hintergrundbaustein `GreenWhiteBackground` gesendet, um den optischen Zustand (z. B. Farbe) zu aktualisieren.

Die Ereignissteuerung:

- Das Eingangsereignis `REQ` und das Ereignis `IX.IND` (Softkey-Interrupt) triggern beide die `OR_2`-Verarbeitung.
- Nach Abschluss der QX-Aktualisierung signalisiert `QX.CNF` dem Hintergrundbaustein, dass eine Zustandsänderung vorliegt und der Hintergrund neu gezeichnet werden soll.

Initialisierung und Qualität:

- `INIT` ⟶ `QX.INIT`; `QX.INITO` ⟶ `IX.INIT`; `IX.INITO` ⟶ `GreenWhiteBackground.INIT`; `GreenWhiteBackground.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`.
- `QI` ⟶ `QX.QI`, `IX.QI` und `GreenWhiteBackground.QI` (kein fester `QI=TRUE` mehr an `QX` und `IX`); `QX.QO` ⟶ `AND_QO.IN1`, `IX.QO` ⟶ `AND_QO.IN2`, `GreenWhiteBackground.QO` ⟶ `AND_QO.IN3`; `AND_QO.OUT` ⟶ `QO`.

Technische Besonderheiten

- Verwendung spezifischer Bibliotheksbausteine: `logiBUS::io::DQ::logiBUS_QX` für den Ausgang und `isobus::UT::io::Softkey::Softkey_IX` für die Softkey-Eingabe.
- Die SubApp enthält einen Parametereingang `Output` vom Typ `logiBUS_DO_S`, der die Ausgangskonfiguration (z. B. Kanaladresse) übergeben bekommt.
- Die Objekt-ID (`u16ObjId`) wird sowohl an den Softkey- als auch an den Hintergrundbaustein weitergeleitet, um eine konsistente Zuordnung zu gewährleisten.
- Die Verbindungen sind teilweise als unsichtbar markiert (`Visible="false"`), was auf eine bewusste Kapselung interner Verdrahtung hinweist.
- Der Baustein ist generisch gestaltet und kann für verschiedene Softkey-/Ausgangskombinationen wiederverwendet werden.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten auf SubApp-Ebene. Der logische Zustand ergibt sich aus der ODER-Verknüpfung:

- **Ausgang inaktiv:** `OUT = FALSE` und Softkey nicht gedrückt (`IX.IN = FALSE`) ⟶ `QX.OUT = FALSE`, Hintergrund zeigt „Aus“.
- **Ausgang aktiv durch Programm:** `OUT = TRUE`, Softkey egal ⟶ `QX.OUT = TRUE`, Hintergrund zeigt „Ein“.
- **Ausgang aktiv durch Softkey:** `OUT = FALSE`, Softkey gedrückt (`IX.IN = TRUE`) ⟶ `QX.OUT = TRUE`, Hintergrund zeigt „Ein“.
- **Beide aktiv:** triviale ODER-Bedingung, Ausgang bleibt aktiv.

Der Hintergrund wird bei jeder Änderung des Ausgangssignals über `QX.CNF` aktualisiert.

Anwendungsszenarien

- **Handbetrieb an Maschinen:** Ein Bediener kann einen Ausgang (z. B. Ventil, Lampe) manuell einschalten, auch wenn das Programm das Signal nicht setzt – z. B. für Wartungs- oder Testzwecke.
- **Not-Aus-Override:** Temporäre Übersteuerung eines programmgesteuerten Ausgangs durch einen Softkey.
- **Visualisierung:** Der Hintergrundbaustein (GreenWhiteBackground) spiegelt den Zustand farblich wider (grün/weiß) und ermöglicht eine einfache Zustandsanzeige in HMI-Systemen.

Vergleich mit ähnlichen Bausteinen

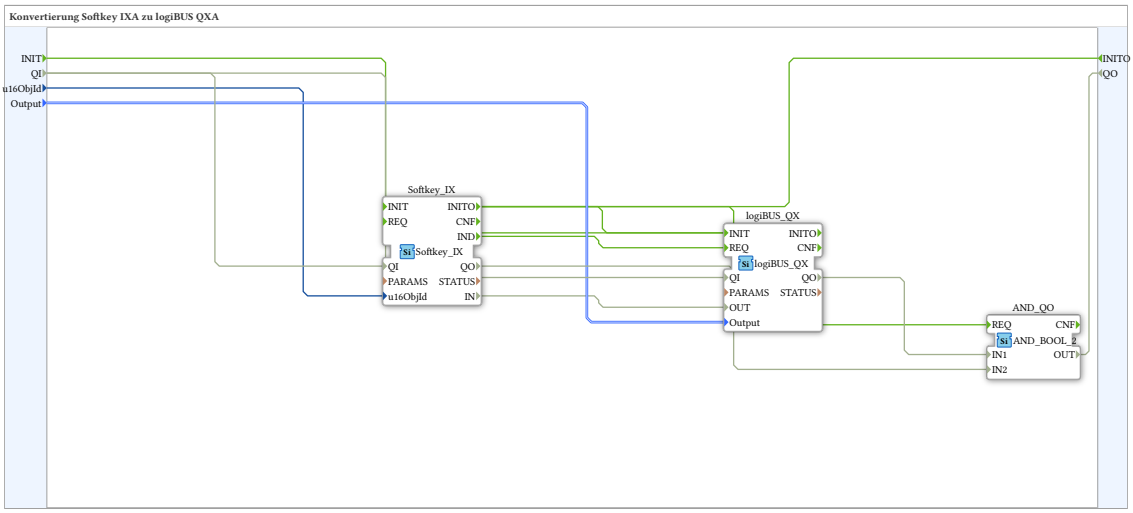
- **Einfacher QX-Baustein (ohne Override):** Nur das Programmsignal `OUT` steuert den Ausgang; keine manuelle Eingriffsmöglichkeit.
- **QX mit separatem Softkey-FB:** Man müsste die ODER-Verknüpfung und die Hintergrundaktualisierung extern verdrahten – hier ist dies gekapselt und wiederverwendbar.

- **Baustein mit Schaltfläche statt Softkey:** Die Funktionsweise wäre ähnlich, jedoch ist `Softkey_IX` speziell für physikalische Softkeys optimiert.

Fazit

Der Baustein **SoftkeyOverride_TO_QX_BG** bietet eine kompakte und wiederverwendbare Lösung für digitale Ausgänge mit manueller Übersteuerung. Durch die integrierte ODER-Logik und die Kopplung an ein Hintergrundobjekt werden sowohl die Funktionalität als auch die Anzeige zentral abgebildet. Die klare Schnittstelle mit zwei Ereignis- (`INIT`, `REQ`) und vier Dateneingängen (`QI`, `OUT`, `u16ObjId`, `Output`) erleichtert die Integration in übergeordnete Steuerungen. Der Baustein ist ideal für Anwendungen, bei denen ein Ausgang sowohl programmgesteuert als auch manuell bedienbar sein muss.

Softkey_IX_TO_logiBUS_QX



• – ○ • – ○ • – ○ •

Einleitung

`Softkey_IX_TO_logiBUS_QX` ist das test_B-Gegenstück zu `Softkey_IXA_TO_logiBUS_QXA` — ein VT-Softkey (`Softkey_IX`) schaltet direkt einen physischen Ausgang (`logiBUS_QX`), ohne Adapter.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Softkey_IX_TO_logiBUS_QX

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **Softkey_IX**: `isobus::UT::io::Softkey::Softkey_IX` — VT-Softkey (Nicht-Adapter-Variante).
  - **logiBUS_QX**: `logiBUS::io::DQ::logiBUS_QX` — physischer digitaler Ausgang.
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_2` — verknüpft `logiBUS_QX.QO` und `Softkey_IX.QO` zum Ausgang `QO`.
- **Funktionsweise**: `Softkey_IX.IND` löst `logiBUS_QX.REQ` aus; `Softkey_IX.IN` wird direkt auf `logiBUS_QX.OUT` verdrahtet. Die SubApp-Schnittstelle besitzt außerdem die Ereignisse `INIT`/`INITO` sowie `QI` (BOOL, Initialwert `TRUE`) und `QO` (BOOL).

Programmablauf und Verbindungen

1. `u16ObjId` ⟶ `Softkey_IX.u16ObjId`; `Output` ⟶ `logiBUS_QX.Output`.
2. `Softkey_IX.IND` ⟶ `logiBUS_QX.REQ`; `Softkey_IX.IN` ⟶ `logiBUS_QX.OUT`.
3. Initialisierungskette: `INIT` ⟶ `logiBUS_QX.INIT`, `logiBUS_QX.INITO` ⟶ `Softkey_IX.INIT`, `Softkey_IX.INITO` ⟶ `INITO` und `AND_QO.REQ`.

4. Qualität: `QI` ⟶ `logiBUS_QX.QI` und `Softkey_IX.QI`; `logiBUS_QX.QO` ⟶ `AND_QO.IN1`; `Softkey_IX.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`.

Anwendungsszenarien

- Minimaler Softkey-zu-Ausgang-Baustein für test_B.

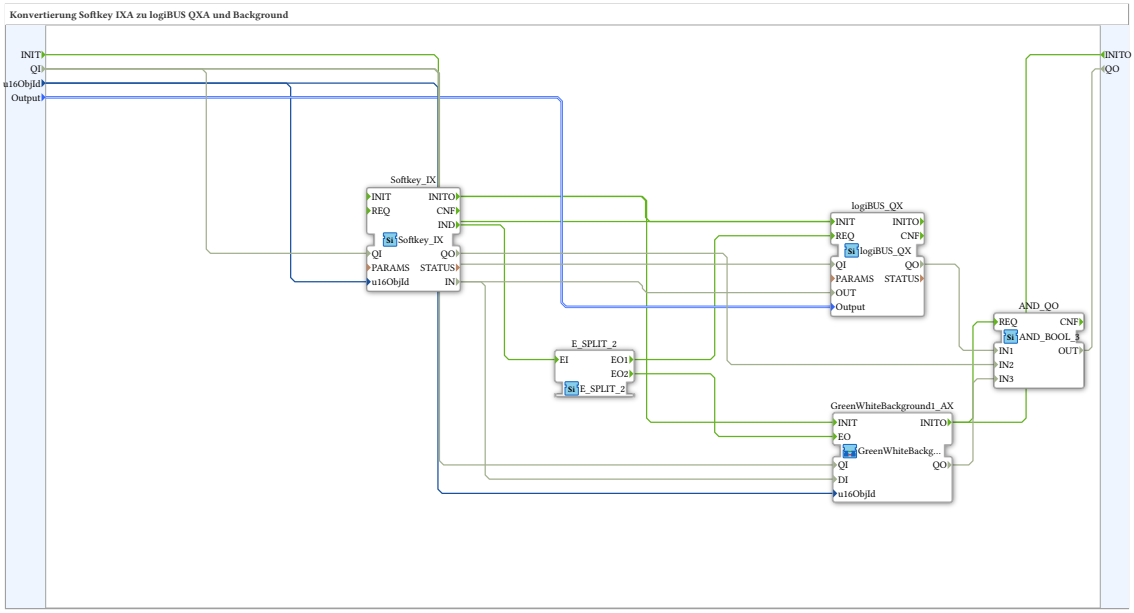
Zusammenfassung

test_B-Pendant zu `Softkey_IXA_TO_logiBUS_QXA`.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

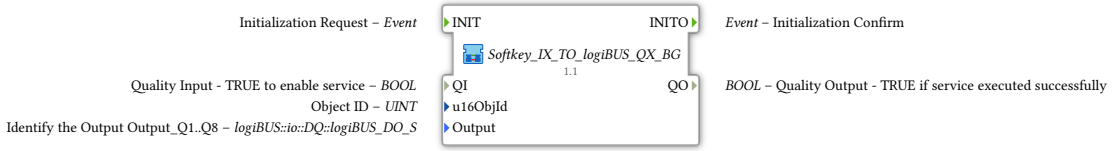
Softkey_IX_TO_logiBUS_QX_BG



• – ○ • – ○ • – ○ •

Einleitung

`Softkey_IX_TO_logiBUS_QX_BG` erweitert `Softkey_IX_TO_logiBUS_QX` um eine VT-Statusfarbe — das test_B-Gegenstück zu `Softkey_IXA_TO_logiBUS_QXA_BG`.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Softkey_IX_TO_logiBUS_QX_BG

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **Softkey_IX**: `isobus::UT::io::Softkey::Softkey_IX` — VT-Softkey.
  - **logiBUS_QX**: `logiBUS::io::DQ::logiBUS_QX` — physischer digitaler Ausgang.
  - **E_SPLIT_2**: `iec61499::events::E_SPLIT_2` — verzweigt das Softkey-Ereignis.
  - **Farbbaustein** (kompakte Background-Variante, siehe Background-Farbbausteine): setzt die Statusfarbe.
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_3` — verknüpft die `QO` von `logiBUS_QX`, `Softkey_IX` und dem Farbbaustein.

- **Funktionsweise**: `Softkey_IX.IND` wird über `E_SPLIT_2` an `logiBUS_QX.REQ` und den Farbwechsel-Baustein verteilt; `Softkey_IX.IN` geht an `logiBUS_QX.OUT` und an den Farbbaustein.

Programmablauf und Verbindungen

1. `u16ObjId` ⟶ `Softkey_IX.u16ObjId` und Farbbaustein `.u16ObjId`; `Output` ⟶ `logiBUS_QX.Output`.
2. `Softkey_IX.IND` ⟶ `E_SPLIT_2.EI` ⟶ `EO1` ⟶ `logiBUS_QX.REQ`, `EO2` ⟶ Farbbaustein `.EO`.
3. `Softkey_IX.IN` ⟶ `logiBUS_QX.OUT` und ⟶ Farbbaustein `.DI`.
4. Initialisierung: Ereigniseingang `INIT` ⟶ `logiBUS_QX.INIT`, `logiBUS_QX.INITO` ⟶ `Softkey_IX.INIT`, `Softkey_IX.INITO` ⟶ Farbbaustein `.INIT` (Instanz `GreenWhiteBackground1_AX`), dessen `INITO` ⟶ Ereignisausgang `INITO` und ⟶ `AND_QO.REQ`.
5. Qualität: Dateneingang `QI` (`BOOL`, Default `TRUE`) ⟶ `logiBUS_QX.QI`, `Softkey_IX.QI` und Farbbaustein `.QI`; `logiBUS_QX.QO` ⟶ `AND_QO.IN1`, `Softkey_IX.QO` ⟶ `AND_QO.IN2`, Farbbaustein `.QO` ⟶ `AND_QO.IN3`; `AND_QO.OUT` ⟶ Datenausgang `QO`.

Anwendungsszenarien

- VT-Softkey mit optischer Statusrückmeldung für test_B, ohne OPC-UA.

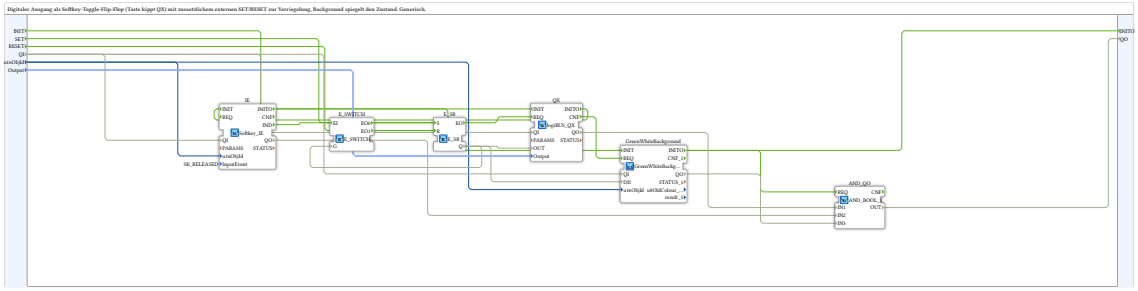
Zusammenfassung

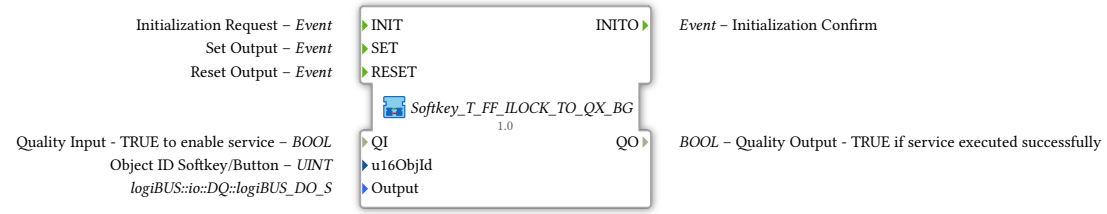
test_B-Pendant zu `Softkey_IXA_TO_logiBUS_QXA_BG`.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Softkey_T_FF_ILOCK_TO_QX_BG





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **Softkey_T_FF_ILOCK_TO_QX_BG** ist eine generische Subapplikation (SubApp) zur Realisierung eines digitalen Ausgangs, der durch einen Softkey (Taster) im Toggle-Betrieb geschaltet wird. Zusätzlich kann der Ausgang über externe Ereignisse `SET` und `RESET` verriegelt (gesetzt oder zurückgesetzt) werden. Ein Hintergrundobjekt (`GreenWhiteBackground`) visualisiert den aktuellen Schaltzustand.

Die SubApp kombiniert Funktionen einer Toggle-Flipflop-Schaltung mit einer Interlock-Logik und einer direkten Ankopplung an einen logiBUS-Digitalausgang (`QX`). Sie ist konfigurierbar über die Objekt-ID des Softkeys und die Ausgangskonfiguration.

Schnittstellenstruktur

Die SubApp besitzt die Initialisierungs-/Qualitäts-Schnittstelle `INIT`/`INITO` und `QI`/`QO`; Adapter sind nicht vorhanden.

Ereignis-Eingänge

- **INIT** (EInit) – Initialisierungsanforderung.
- **SET** – Setzt den Ausgang unabhängig vom Toggle-Verhalten auf TRUE.
- **RESET** – Setzt den Ausgang unabhängig vom Toggle-Verhalten auf FALSE.

Ereignis-Ausgänge

- **INITO** (EInit) – Initialisierungsbestätigung (von GreenWhiteBackground.INITO).

Daten-Eingänge

- **QI** (BOOL) – Quality Input, TRUE gibt den Dienst frei. Initialwert: TRUE.
- **u16ObjId** (UINT) – Objektidentifikation des Softkeys/Buttons. Initialwert: ID_NULL.
- **Output** (logiBUS::io::DQ::logiBUS_DO_S) – Ausgangskonfiguration für den logiBUS-Digitalausgang. Initialwert: logiBUS_DO::Invalid.

Daten-Ausgänge

- **QO** (BOOL) – Quality Output, UND aus QX.QO, IE.QO und GreenWhiteBackground.QO (AND_QO, iec61131::booleanOperators::AND_BOOL_3).

Adapter

Keine.

Funktionsweise

Die SubApp realisiert eine kombinierte Toggle- und Interlock-Logik für einen Digitalausgang. Der interne Ablauf ist wie folgt:

1. **Softkey-Ereignis**: Der Block Softkey_IE (intern IE) überwacht den zugeordneten Softkey und erzeugt bei Loslassen (Ereignis SK_RELEASED) ein IND-Ereignis.
2. **Toggle-Decision**: Das IND-Ereignis wird an den Block E_SWITCH weitergeleitet. Dessen Steuereingang G ist mit dem Ausgang Q des internen Set-Reset-Flipflops E_SR verbunden. Ist der aktuelle Zustand FALSE (Q=0), wird das Ereignis an den Ausgang EO0 durchgeschaltet, der das Setzen (S) von E_SR auslöst. Ist der Zustand TRUE (Q=1), geht das Ereignis über EO1 zum Rücksetzen (R) des Flipflops. Dadurch wird bei jeder Betätigung der Zustand invertiert (Toggle).
3. **Externes Set/Reset**: Über die Ereigniseingänge SET und RESET kann der Zustand des Flipflops unabhängig von der Toggle-Funktion direkt geändert werden. Ein SET setzt Q auf TRUE, ein RESET auf FALSE. Diese externen Ereignisse haben Vorrang und können als Verriegelung (Interlock) verwendet werden.
4. **Ausgangssignal**: Der Ausgang Q von E_SR speist den Daten-Eingang OUT des Digitalausgangsblocks QX (Typ logiBUS_QX). Dieser setzt den physischen Ausgang entsprechend dem Signalpegel.
5. **Hintergrundvisualisierung**: Parallel wird Q an den Eingang DI1 der SubApp GreenWhiteBackground übergeben. Diese steuert die Darstellung eines Hintergrundelements (z. B. Farbe) entsprechend dem Zustand. Die Objekt-ID aus u16ObjId wird sowohl an GreenWhiteBackground als auch an Softkey_IE weitergeleitet, um die richtige Zuordnung zu gewährleisten.
6. **Konfiguration**: Der externe Daten-Eingang Output (Typ logiBUS_DO_S) spezifiziert die Ausgangskonfiguration (z. B. logische Zuordnung) und wird direkt an den QX-Block übergeben. Der Qualifier QI des QX-Blocks ist nicht mehr fest auf TRUE gesetzt, sondern kommt vom SubApp-Eingang QI (Vorgabe TRUE); QI wird außerdem an IE.QI und GreenWhiteBackground.QI verteilt.
7. **Initialisierung und Qualität**: INIT → QX.INIT; QX.INITO → IE.INIT; IE.INITO → GreenWhiteBackground.INIT; GreenWhiteBackground.INITO → INITO (und → AND_QO.REQ). QX.QO → AND_QO.IN1; IE.QO → AND_QO.IN2; GreenWhiteBackground.QO → AND_QO.IN3; AND_QO.OUT → QO.

Technische Besonderheiten

- **Toggle mit Verriegelung**: Die Kombination aus E_SWITCH und E_SR erzeugt ein Toggle-Verhalten, während externe Set/Reset-Ereignisse den Zustand überschreiben können. Dies ermöglicht eine sichere Übersteuerung durch übergeordnete Steuerlogik.
- **Ereignisgesteuerte Verarbeitung**: Alle Zustandsänderungen werden über Ereignisse ausgelöst, was eine ressourcenschonende und reaktionsschnelle Implementierung in der 4diac-IDEs erlaubt.
- **Wiederverwendung**: Die SubApp ist aus einem größeren Projekt ausgelagert und als generischer Baustein konzipiert. Sie kann in verschiedenen Anwendungen mit unterschiedlichen Objekt-IDs und Ausgangskonfigurationen eingesetzt werden.

- **Interne Bausteine**: Verwendet werden Standard-Funktionsblöcke aus der IEC 61499-Bibliothek (`E_SR`, `E_SWITCH`) sowie logiBUS- und isobus-spezifische Erweiterungen.
- **Visualisierung**: Der Zustand wird über `GreenWhiteBackground` auf einem HMI (Human Machine Interface) dargestellt, was die Betriebsüberwachung erleichtert.

Zustandsübersicht

Die SubApp besitzt einen internen booleschen Zustand `Q`, der durch die Ereignisse verändert wird. Es ergeben sich zwei stabile Zustände:

- **Zustand `Q=0`** (Ausgang aus):
  - Bei Tastendruck (Release) wird das Ereignis über `E_SWITCH` auf `EO0` geleitet → Setzen → `Q` wird `1`.
  - Ein externes `RESET` hält den Zustand bei `0`, ein `SET` setzt auf `1`.
- **Zustand `Q=1`** (Ausgang ein):
  - Bei Tastendruck (Release) wird das Ereignis über `E_SWITCH` auf `EO1` geleitet → Rücksetzen → `Q` wird `0`.
  - Ein externes `SET` hält den Zustand bei `1`, ein `RESET` setzt auf `0`.

Die Übergänge sind deterministisch und werden durch die eingehenden Ereignisse gesteuert. Die Priorität der externen SET/RESET liegt über dem Toggle-Ereignis, da diese direkt auf `E_SR` wirken.

Anwendungsszenarien

Typische Einsatzszenarien sind:

- **Maschinenbedienung**: Ein Taster (Softkey) schaltet einen Ausgang (z. B. eine Leuchte oder ein Ventil) ein und aus. Übergeordnete Steuerungen können den Ausgang per `SET`/`RESET` verriegeln (z. B. Not-Aus oder Freigaben).
- **HMI-Integration**: In Verbindung mit dem `GreenWhiteBackground`-Baustein kann der Schaltzustand direkt auf einem Bedienpanel farblich dargestellt werden.
- **Steuerung von logiBUS-Ausgängen**: Der Baustein ist speziell für die Ansteuerung von logiBUS-Digitalausgängen (`logiBUS_QX`) ausgelegt und kann einfach in logiBUS-basierte Systeme eingebunden werden.
- **Wiederverwendbare Funktion**: Da die SubApp generisch ist, kann sie mit verschiedenen Softkey-Objekten und Ausgangskonfigurationen parametriert und in mehreren Projekten eingesetzt werden.

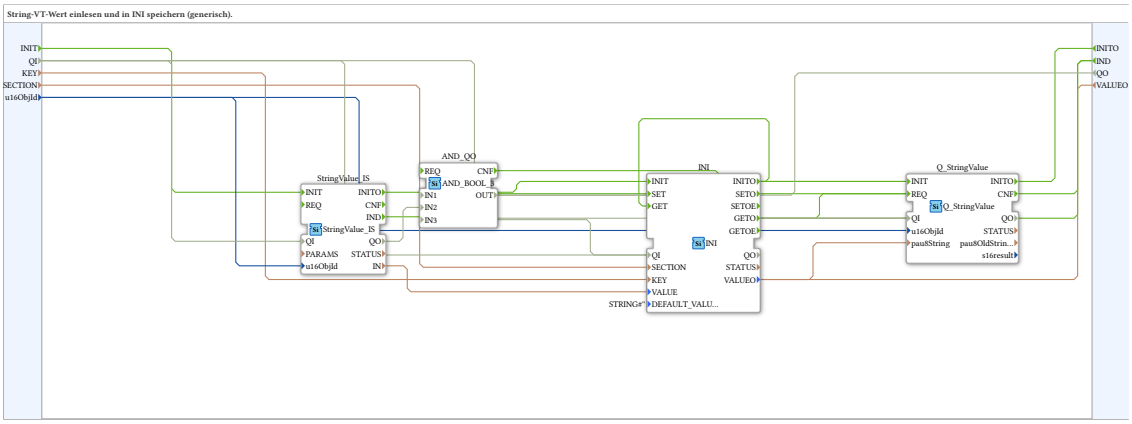
Vergleich mit ähnlichen Bausteinen

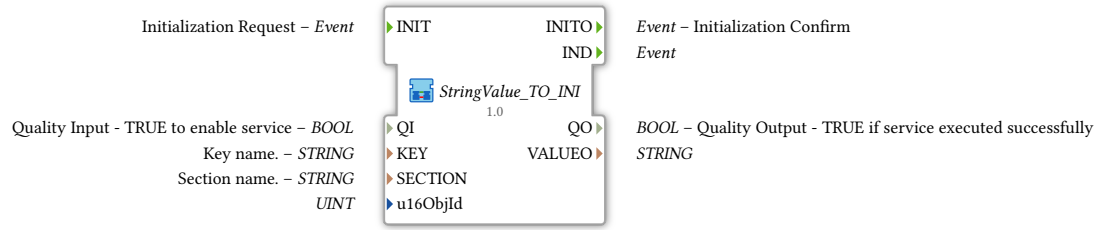
Im Vergleich zu einfachen Toggle-Flipflops (z. B. einem `E_SR` mit Rückkopplung) bietet dieser Baustein eine integrierte Visualisierung und eine klare Trennung zwischen Toggle- und Verriegelungsfunktion. Während ein einfaches Toggle-Flipflop nur den Zustand wechselt, erlaubt dieser Block eine externe Prioritätssteuerung, was die Sicherheit in Steuerungsanwendungen erhöht. Gegenüber Bausteinen ohne Hintergrundvisualisierung entfällt die Notwendigkeit, den Zustand separat an ein Anzeigeelement zu übertragen. Der Baustein ist außerdem an logiBUS-Ausgänge angepasst, was eine direkte, konfigurierbare Anbindung physischer Ausgänge erleichtert.

Fazit

Der Funktionsblock **Softkey_T_FF_ILOCK_TO_QX_BG** stellt eine robuste und flexible Lösung für die Ansteuerung digitaler Ausgänge über Softkeys dar. Durch die Kombination von Toggle-Logik, externer Verriegelung und integrierter Zustandsvisualisierung eignet er sich ideal für sicherheitsrelevante Bedienoberflächen in der Automatisierungstechnik. Die generische Ausführung ermöglicht eine breite Wiederverwendung in verschiedenen Projekten und vereinfacht die Projektierung erheblich.

StringValue_TO_INI





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **StringValue_TO_INI** ist eine generische Subapplikation, die einen String-Wert von einem ISOBUS-Gerät einliest und in einer INI-Datei persistent speichert. Die Konfiguration erfolgt über die Angabe von Sektion, Key und der ISOBUS-Objekt-ID. Zusätzlich wird beim Start der gespeicherte Wert aus der INI-Datei gelesen und über den Ausgang `VALUEO` bereitgestellt. Der Baustein ist als Wiederverwendungsmodul ausgelegt und kapselt die Kommunikation mit dem ISOBUS sowie den Zugriff auf das INI-Dateisystem.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`) – Initialization Request; startet die Initialisierungskette `StringValue_IS` ⟶ `INI` ⟶ `Q_StringValue`.

Ereignis-Ausgänge

- **INITO** (`EInit`) – Initialization Confirm; kommt von `Q_StringValue.INITO` (Ende der Kette).
- **IND** – Signalisiert, dass entweder ein Wert in die INI-Datei geschrieben oder der gespeicherte Wert gelesen und ausgegeben wurde.

Daten-Eingänge

- **QI** (`BOOL`, Initialwert `TRUE`) – Quality Input: TRUE aktiviert den Dienst; wird an `StringValue_IS.QI`, `INI.QI` und `Q_StringValue.QI` verteilt.
- **KEY** (`STRING`) – Bezeichner des Schlüssels innerhalb der Sektion, unter dem der Wert gespeichert wird.
- **SECTION** (`STRING`) – Name der Sektion in der INI-Datei.
- **u16ObjId** (`UINT`, Initialwert `ID_NULL`) – ISOBUS-Objekt-ID, unter der der String-Wert vom Bus gelesen wird.

Daten-Ausgänge

- **QO** (`BOOL`) – Quality Output: TRUE, wenn der Dienst erfolgreich ausgeführt wurde; UND-Verknüpfung (`AND_QO`, `AND_BOOL_3`) von `Q_StringValue.QO`, `StringValue_IS.QO` und `INI.QO`.
- **VALUEO** (`STRING`) – Ausgang, der den zuletzt gelesenen Wert aus der INI-Datei bzw. den gerade verarbeiteten Wert bereitstellt.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Die Subapplikation kombiniert drei Funktionsblöcke und einen UND-Baustein `AND_QO` (`iec61131::booleanOperators::AND_BOOL_3`) für das Quality-Ausgangssignal:

- **StringValue_IS** – Liest einen String-Wert von einem ISOBUS-Gerät basierend auf der angegebenen `u16ObjId`.
- **INI** – Speichert bzw. liest Werte aus einer INI-Datei unter der angegebenen `SECTION` und `KEY`.
- **Q_StringValue** – Ein Warteschlangen-Baustein, der den gelesenen Wert puffert (der konkret genutzte Zweck ist derzeit nicht vollständig ausgebaut, dient aber zur Vorbereitung auf spätere Erweiterungen).

Ablauf:

1. **Initialisierung:** Nach dem Start wird das Event `INITO` des INI-Bausteins ausgelöst, das intern einen `GET`-Vorgang startet. Dadurch wird der unter `KEY`/`SECTION` gespeicherte Wert aus der INI-Datei gelesen. Das `GETO`-Event wird anschließend an den `Q_StringValue` (REQ) und auch an den Ausgang `IND` weitergeleitet. Der Wert liegt gleichzeitig am Datenausgang `VALUEO` an.
2. **Schreiben eines neuen Werts:** Sobald `StringValue_IS` einen neuen String-Wert vom ISOBUS empfängt (Event `IND` von `StringValue_IS`), wird dieser über die Datenverbindung an den `INI`-Baustein übergeben und dort durch das Event `SET` gespeichert. Nach erfolgreichem Schreiben wird das Event `SETO` von `INI` an den Ausgang `IND` weitergegeben.
3. **Lesen nach einem Leseimpuls:** Der INI-Baustein kann auch durch ein externes `GET`-Event (das intern über die Init-Schleife erzeugt wird) angestoßen werden. Jedes `GETO`-Event triggert den `Q_StringValue` und gibt den gelesenen Wert über `VALUEO` aus sowie über `IND` als Ereignis.

Initialisierung: `INIT` ⟶ `StringValue_IS.INIT`; `StringValue_IS.INITO` ⟶ `INI.INIT`; `INI.INITO` ⟶ `Q_StringValue.INIT`; `Q_StringValue.INITO` ⟶ `INITO` und ⟶ `AND_QO.REQ`. Qualität: `QI` wird an `StringValue_IS.QI` und `INI.QI` (sowie `Q_StringValue.QI`) verteilt; `Q_StringValue.QO` ⟶ `AND_QO.IN1`, `StringValue_IS.QO` ⟶ `AND_QO.IN2`, `INI.QO` ⟶ `AND_QO.IN3`; `AND_QO.OUT` ⟶ `QO`. Die interne `GET`-Auslösung nach `INI.INITO` bleibt unverändert.

Die Parameter `KEY`, `SECTION` und `u16ObjId` werden während der Laufzeit über die Eingänge gesetzt und können so für verschiedene Anwendungen dynamisch konfiguriert werden.

Technische Besonderheiten

- **ISOBUS-Integration:** Über den Baustein `StringValue_IS` wird die Kommunikation mit dem ISOBUS abgewickelt. Die Objekt-ID (`u16ObjId`) bestimmt, welcher String-Wert vom Bus gelesen wird.
- **Generische INI-Speicherung:** Die Verwendung von `INI` ermöglicht das Speichern in beliebigen Sektionen und Schlüsseln. Der Baustein unterstützt die Angabe eines Default-Werts, der beim ersten Zugriff verwendet wird.
- **Warteschlangen-Baustein:** `Q_StringValue` dient als Puffer für den gelesenen String-Wert und kann in erweiterten Szenarien weitere Verarbeitungsschritte anstoßen.
- **Wiederverwendbarkeit:** Durch die ausgelagerte Implementierung kann der Baustein in verschiedenen Projekten ohne Anpassungen eingesetzt werden.

Zustandsübersicht

Eine explizite Zustandsmaschine ist in der Subapp nicht implementiert. Der Ablauf lässt sich jedoch in folgende Zustände gliedern:

- **Initialisierungszustand:** Beim Start wird einmalig ein `GET` ausgeführt und der gespeicherte Wert über `VALUEO` ausgegeben.
- **Wartezustand:** Nach der Initialisierung wartet der Baustein auf ein neues Event von `StringValue_IS` oder auf einen internen `GET`-Impuls (der gegenwärtig nur durch die Init-Schleife erzeugt wird).
- **Schreibzustand:** Bei Eintreffen eines neuen String-Werts wird dieser in die INI-Datei geschrieben und das `SETO`-Event ausgelöst.
- **Lesezustand:** Bei jedem `GET`-Event wird der aktuelle Wert aus der INI-Datei gelesen und über `VALUEO` sowie `IND` ausgegeben.

Anwendungsszenarien

- **Persistente Speicherung von ISOBUS-Einstellungen:** Geräteparameter wie z. B. Maschineneinstellungen, die über ISOBUS gelesen werden, können dauerhaft in einer INI-Datei abgelegt werden, damit sie nach einem Neustart wiederhergestellt werden.
- **Konfigurationsverwaltung:** Soll ein Wert aus einer zentralen Konfigurationsdatei in eine ISOBUS-Steuerung zurückgeschrieben werden, kann der Baustein um eine Rückrichtung erweitert werden.
- **Test- und Diagnosefunktionen:** Der gelesene Wert kann über `VALUEO` an übergeordnete Überwachungssysteme weitergegeben werden, während die Speicherung für spätere Auswertungen erfolgt.

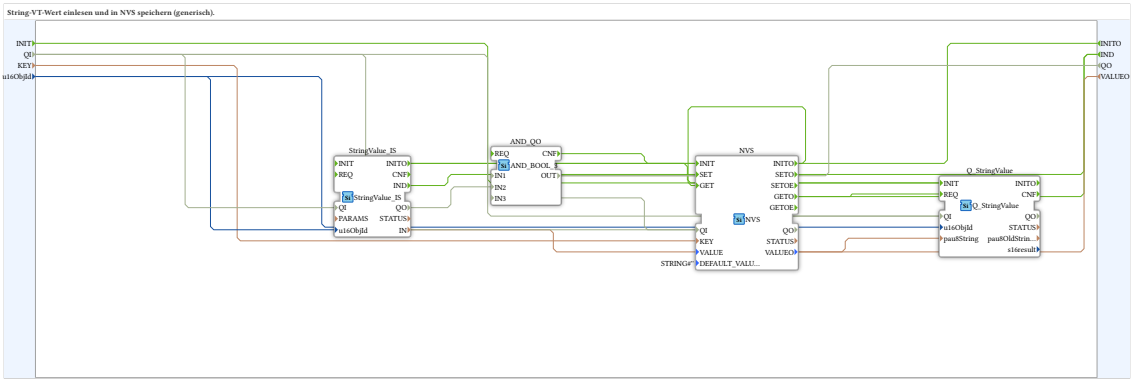
Vergleich mit ähnlichen Bausteinen

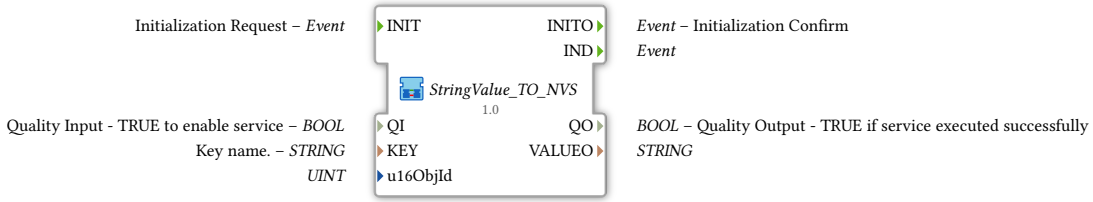
- **Direkte INI-Zugriffe:** Im Gegensatz zu Bausteinen, die nur direkt auf eine INI-Datei zugreifen, integriert `StringValue_TO_INI` den ISOBUS-Datenfluss. Das erspart dem Anwender die manuelle Verbindung von ISOBUS-Lese- und INI-Schreibfunktionen.
- **Generische String-Speicherbausteine:** Andere Bausteine erwarten möglicherweise feste Sektionen oder Schlüssel; dieser Baustein erlaubt die dynamische Konfiguration über die Eingänge.
- **ISOBUS-spezifische Speicherlösungen:** Es existieren spezielle ISOBUS-Speicherbausteine, die aber oft nur die Objekt-ID speichern, nicht den Wert selbst. Vorliegender Baustein speichert den Wert und lässt sich flexibel einsetzen.

Fazit

Der Funktionsblock **StringValue_TO_INI** stellt eine robuste und wiederverwendbare Lösung dar, um ISOBUS-Stringwerte dauerhaft in INI-Dateien zu speichern und bei Bedarf wieder auszulesen. Durch die klare Schnittstellenstruktur und die interne Verwendung etablierter Bibliotheksbausteine lässt er sich einfach in bestehende Projekte integrieren. Die Kombination aus ISOBUS-Anbindung und generischer INI-Konfiguration macht ihn zu einem nützlichen Werkzeug für Anwendungen, die eine dauerhafte Speicherung von Buskommunikationswerten erfordern.

StringValue_TO_NVS





• – ○ • – ○ • – ○ •

Einleitung

Die Subapp **StringValue_TO_NVS** dient dazu, einen Stringwert, der über eine Objekt-ID (z.B. aus einem ISO-bus-System) eingelesen wird, dauerhaft in einem nichtflüchtigen Speicher (NVS – Non-Volatile Storage) zu speichern. Sie ist generisch einsetzbar und trennt die Lese-, Speicher- und Ausgabefunktionalität, sodass sie leicht in verschiedene Anwendungen integriert werden kann.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Datentyp	Beschreibung
INIT	EInit	Initialisierungsanforderung; startet die Initialisierungskette (Q_StringValue → StringValue_IS → NVS).

Ereignis-Ausgänge

Name	Datentyp	Beschreibung
INITO	EInit	Initialisierungsbestätigung von NVS.INITO (Ende der Initialisierungskette).
IND	Event	Wird ausgelöst, nachdem ein Speicher- oder Lesevorgang abgeschlossen wurde (siehe Funktionsweise).

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung
QI	BOOL	TRUE	Quality Input – TRUE aktiviert den Dienst; wird an Q_StringValue.QI, StringValue_IS.QI und NVS.QI durchgereicht.
KEY	STRING	–	Name des Schlüssels, unter dem der Wert im NVS abgelegt wird.
u16ObjId	UINT	ID_NULL	Objekt-ID, die für den Zugriff auf den Stringwert verwendet wird (z.B. bei ISO-bus).

Daten-Ausgänge

Name	Typ	Beschreibung
QO	BOOL	Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde; UND (AND_QO, Typ iec61131::booleanOperators::AND_BOOL_3) der QO von Q_StringValue, StringValue_IS und NVS.
VALUEO	STRING	Liest den aktuell im NVS gespeicherten String zurück (nach einem Lesevorgang).

Adapter

Keine.

Funktionsweise

Die Subapp kombiniert drei Funktionsbausteine, um folgende Abläufe zu realisieren:

1. **Initialisierung**
   Nach dem Start (oder Einschalten) wird der Baustein NVS über dessen INITO-Ereignis initialisiert. Unmittelbar danach wird ein Lesevorgang gestartet (INITO → GET). Der im NVS gespeicherte Wert wird über VALUEO an den Ausgang VALUEO der Subapp und zusätzlich an den Baustein Q_StringValue weitergegeben. Das GETO-Ereignis löst gleichzeitig das Ausgangsereignis IND aus.
2. **Speichern eines neuen Werts**
   Wenn ein neuer Stringwert über die Objekt-ID verfügbar ist (Baustein StringValue_IS), wird dieser über dessen Ausgang IN an den Eingang VALUE des NVS-Bausteins übergeben. Das Ereignis IND von StringValue_IS triggert den Speichervorgang (SET). Nach erfolgreichem Speichern wird SETO ausgelöst, was wiederum das Ausgangsereignis IND der Subapp aktiviert.
3. **Rückgabe des gespeicherten Werts**
   Der Baustein Q_StringValue dient zur Weiterverarbeitung bzw. Quittierung des ausgelesenen Strings. Durch die Verbindung NVS.VALUEO → Q_StringValue.pau8String wird der zwischengespeicherte Wert verfügbar gemacht. Ein Zugriff über das GETO-Ereignis (das auch direkt als IND an der Subapp erscheint) löst die Verarbeitung in Q_StringValue aus.

Zusätzlich läuft die Initialisierungskette INIT → Q_StringValue.INIT, Q_StringValue.INITO → StringValue_IS.INIT, StringValue_IS.INITO → NVS.INIT, NVS.INITO → INITO und → AND_QO.REQ; QI geht an Q_StringValue.QI, StringValue_IS.QI und NVS.QI; deren QO werden im Baustein AND_QO (AND_BOOL_3) per UND zu QO verknüpft.

Die Ereignisverkabelung stellt sicher, dass sowohl nach einem Schreib- als auch nach einem Lesevorgang das Ausgangsereignis IND gesendet wird, sodass der aufrufende Baustein über den Abschluss informiert ist.

Technische Besonderheiten

- **Generische Verwendbarkeit**: Die Objekt-ID (u16ObjId) und der Schlüssel (KEY) sind als Eingänge parametrierbar, wodurch die Subapp für unterschiedliche Stringwerte und Speicherbereiche eingesetzt werden kann.
- **Verwendung von NVS**: Der Baustein NVS implementiert einen nichtflüchtigen Speicher auf ESP32-Basis, typischerweise über die integrierte Flash-Technologie. Dadurch bleiben Daten auch nach einem Reset erhalten.
- **Initialisierungslogik**: Durch die Verbindung von INITO → GET wird beim Start automatisch der aktuell gespeicherte Wert gelesen und über VALUEO verfügbar gemacht.
- **Quellcode-Kapselung**: Die interne Struktur (StringValue_IS, NVS, Q_StringValue) ist gekapselt, sodass die Subapp als einfacher, wiederverwendbarer Baustein in größere Systeme integriert werden kann.

Zustandsübersicht

Da die Subapp keine expliziten Zustandsautomaten besitzt, lässt sich der Ablauf am besten über die Ereignisse und die beteiligten Baustein-Zustände beschreiben:

- **Idle**: Es liegen keine anstehenden Ereignisse vor. Der NVS-Wert ist gespeichert (kein Zugriff aktiv).
- **Initialisieren**: Nach dem Einschalten wird der NVS initialisiert (INITO steht an); das Ereignis INIT der Subapp initialisiert nacheinander Q_StringValue, StringValue_IS und NVS und wird mit INITO (von NVS.INITO) bestätigt.
- **Lesen**: Nach der Initialisierung oder über einen internen Triggers wird ein Lesevorgang durchgeführt (GET). Nach Abschluss wird GETO erzeugt.
- **Schreiben**: Wenn ein neuer Stringwert über StringValue_IS empfangen wird, löst dessen IND-Ereignis den SET-Vorgang aus. Nach dem Schreiben wird SETO erzeugt.
- **Wert bereitstellen**: Nach einem Lesevorgang wird der Wert über VALUEO (und an Q_StringValue) bereitgestellt; nach einem Schreibvorgang wird der Wert ebenfalls gespeichert und bestätigt.

Die Ereignisausgänge IND signalisieren den Abschluss eines Zyklus, unabhängig davon, ob es sich um einen Lese- oder Schreibvorgang handelte.

Anwendungsszenarien

- **Konfigurationsspeicherung**: Ein Stringwert, der über eine Objekt-ID (z.B. aus einem CAN-basierten ISO-Bus-System) gelesen wird, soll dauerhaft abgelegt werden (z.B. Geräteeinstellungen, Kalibrierwerte).
- **Puffern von Werten**: Wenn ein Wert nur temporär verfügbar ist, kann er mit dieser Subapp in den NVS geschrieben und später wieder abgerufen werden.
- **Redundanz**: Durch die Speicherung in NVS kann der letzte bekannte Wert nach einem Neustart automatisch wiederhergestellt werden, ohne dass eine erneute Kommunikation mit dem Sensor erforderlich ist.

Vergleich mit ähnlichen Bausteinen

- **Direkter NVS-Zugriff**: Andere Bausteine (wie reine NVS-FBs) erfordern eine manuelle Anbindung an die Datenquelle und die Verwaltung von Schlüsseln. Diese Subapp kombiniert das Lesen (über eine Objekt-ID) und das Speichern in einem Schritt.
- **Ohne Q_StringValue**: Einige Implementierungen bieten nur die Speicherung ohne zusätzliche Quittierungslogik; hier übernimmt Q_StringValue eine Weiterverarbeitung, z.B. für ein Ereignis- oder Datenaustauschprotokoll.
- **SubApp vs. FB**: Als SubApp kapselt sie eine komplexere Logik und stellt nur eine einfache Schnittstelle bereit, was die Integration in übergeordnete Systeme vereinfacht.

Fazit

StringValue_TO_NVS ist eine praktische Subapp, die das Einlesen von Stringwerten über eine Objekt-ID und deren dauerhafte Speicherung im NVS vereinfacht. Durch die klar definierte Schnittstelle mit nur einem Ereignisausgang, zwei Dateneingängen und einem Datenausgang lässt sie sich schnell in existierende Projekte einbinden. Die automatische Initialisierung und das Ereignismodell gewährleisten eine zuverlässige Kommunikation zwischen den internen Bausteinen. Durch die Option, Schlüssel und Objekt-ID zu parametrieren, bietet sie eine hohe Wiederverwendbarkeit in unterschiedlichen Kontexten.

SwitchPic(Col)-Bausteine: Gemeinsames Muster

• – ◦ • – ◦ • – ◦ •

Einleitung

`MyLib::sys` (test_B) enthält eine Familie von Bausteinen, die **je nach Zustand ein anderes VT-Bild (Object-Pointer-Ziel) und/oder eine andere Hintergrundfarbe anzeigen** — z. B. eine Schieber-/Ventilanimation mit den Zuständen Unknown/Closed/Opening/Opened/Closing. Diese Seite erklärt das gemeinsame Muster; die Einzelseiten nennen nur die konkreten Abweichungen.

Namensschema

`SwitchPic[Col]_<Zustände>_<Variante>[_aux]`

Bestandteil	Bedeutung
`Col` (optional)	Schaltet zusätzlich zum Bild auch eine Hintergrundfarbe (`Q_BackgroundColour`) passend zum Zustand um, nicht nur ein Bild (`Q_NumericValue`).
Zustände (`2` oder `5`)	`2` = boolescher Selector `DI1` (up/down, Typ `s2x1ObjectIDs`/`s2x2ObjectIDs`); `5` = `iSTATE`-Selector (`USINT`, Schieber-Zustandsautomat Unknown/Closed/Opening/Opened/Closing, Typ `SchieberStruct`/`SchieberAuxInStruct`), ausgewertet über `F_MUX_5`.
Variante (`1`/`2`/`3`, nur bei `SwitchPic`)	Anzahl/Art der gleichzeitig aktualisierten VT-Objekte: `1`=nur normales VT-Objekt (Softkey/DataMask), `2`=zusätzlich AUX-Objekt, `3`=zusätzlich AUX-Objekt UND ein zweites normales Objekt („Button“).
`_aux`	Nur AUX-Objekt(e) werden geschaltet, kein normales VT-Objekt (Gegenstück zu Variante `1`, aber ausschließlich für Auxiliary-Function-Objekte).

Funktionsweise

1. Eine Struktur `pictures`/`Sets` (Typ abhängig von Zuständen/Variante) enthält für jeden möglichen Zustand die zugehörige(n) Object-ID(s) (Bild und ggf. Farbe).
2. `F_MOVE` entpackt diese Struktur in Einzelwerte.
3. Ein Multiplexer (`F_SEL` bei 2 Zuständen, `F_MUX_5` bei 5 Zuständen, gesteuert über `DI1` bzw. `iSTATE`) wählt die zum aktuellen Zustand passenden Werte aus.
4. `Q_NumericValue`/`Q_NumericValueAux` setzt den gewählten Object-Pointer-Wert auf das per `Picture`/`PictureA`/`PictureB` identifizierte VT-Objekt; bei `Col`-Varianten setzt zusätzlich `Q_BackgroundColour`/`Q_BackgroundColourAux` die Hintergrundfarbe auf das per `Color` identifizierte Objekt.
5. Bei mehreren Zielobjekten (Variante 2/3) werden mehrere `Q_NumericValue(Aux)`-Instanzen in einer festen INIT-Kette hintereinandergeschaltet (jede löst per `INITO` die nächste aus), bevor `INITO` nach außen gemeldet wird.

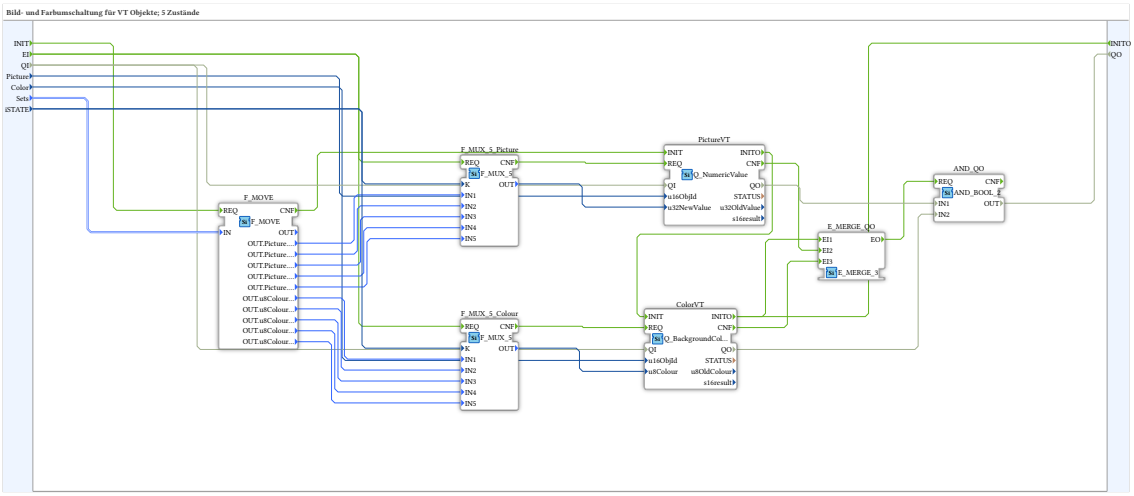
Zusammenfassung

Ein Multiplexer-Muster: Zustand → Struktur-Lookup (`F_MOVE`) → Auswahl (`F_SEL`/`F_MUX_5`) → ein oder mehrere `Q_NumericValue(Aux)`/`Q_BackgroundColour(Aux)`-Ziele. Die Variantenzahl (1/2/3) und `Col`/`_aux` bestimmen nur, wie viele Ziele mit welchem Typ (normal/AUX, Bild/Farbe) gleichzeitig aktualisiert werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SwitchPicCol_5_1



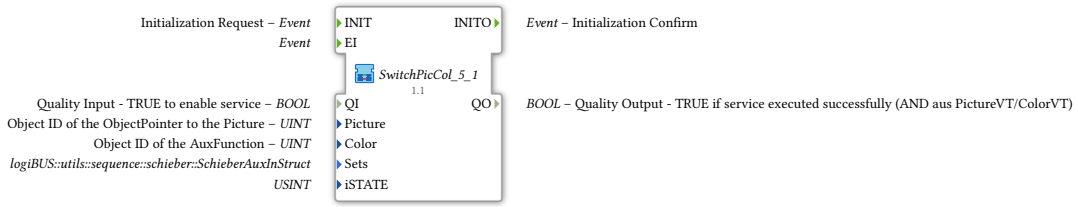
• – ○ • – ○ • – ○ •

Einleitung

`SwitchPicCol_5_1` schaltet zwischen 5 Zuständen (Schieber-/Ventilanimation) sowohl ein VT-Bild (`Picture`, via `Q_NumericValue`) **als auch** eine VT-Hintergrundfarbe (`Color`, via `Q_BackgroundColour`) um — zwei parallele `F_MUX_5`-Multiplexer (einer für Bild-IDs, einer für Farbwerte), beide vom selben `iSTATE`-Wert gesteuert, auf normale (nicht-AUX) Objekte.

Allgemeines Muster siehe SwitchPic(Col)-Bausteine (gemeinsames Muster).

Initialisierung und Qualität: `INIT` ⟶ `F_MOVE.REQ`; `F_MOVE.CNF` ⟶ `PictureVT.INIT`; `PictureVT.INITO` ⟶ `ColorVT.INIT`; `ColorVT.INITO` ⟶ `INITO`. Neu ist `QI` (`BOOL`, Vorgabe `TRUE`), das an `PictureVT.QI` und `ColorVT.QI` geht, sowie `QO` (`BOOL`): `AND` (`AND_QO`, `AND_BOOL_2`) aus `PictureVT.QO` und `ColorVT.QO`. `AND_QO.REQ` wird über `E_MERGE_QO` (`E_MERGE_3`: `ColorVT.INITO`, `PictureVT.CNF`, `ColorVT.CNF`) ausgelöst.



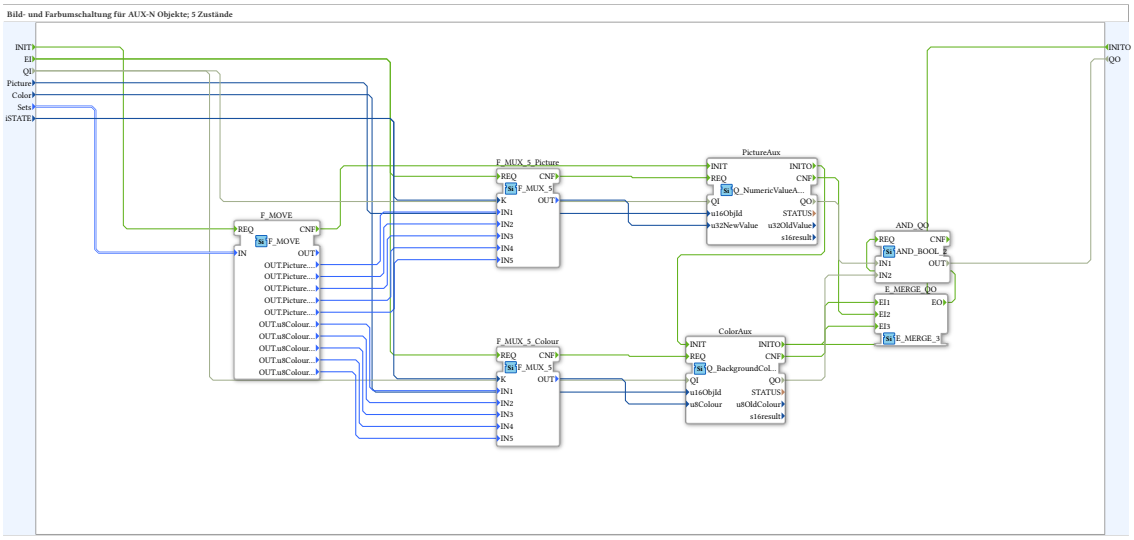
Zusammenfassung

Kombiniert Bild- und Farbumschaltung (`Col`) für 5 Zustände auf normalen VT-Objekten.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SwitchPicCol_5_1_aux



• — ○ • — ○ • — ○ •

Einleitung

`SwitchPicCol_5_1_aux` schaltet zwischen 5 Zuständen sowohl ein Bild als auch eine Hintergrundfarbe um — wie `SwitchPicCol_5_1`, aber ausschließlich für **Auxiliary-Function**-Objekte (`Q_NumericValueAux`/`Q_BackgroundColourAux`) statt normale VT-Objekte.

Initialisierung und Qualität: Die SubApp besitzt die Ereigniseingänge `INIT` und `EI`, den Ereignisausgang `INITO` sowie `QI` (BOOL, Initialwert `TRUE`) und `QO` (BOOL). Init-Kette: `INIT` ⟶ `F_MOVE.REQ`, `F_MOVE.CNF` ⟶ `PictureAux.INIT`, `PictureAux.INITO` ⟶ `ColorAux.INIT`, `ColorAux.INITO` ⟶ `INITO`. `QI` wird an `PictureAux.QI` und `ColorAux.QI` weitergereicht; deren `QO`-Ausgänge gehen auf `AND_QO.IN1` bzw. `AND_QO.IN2` (`iec61131::booleanOperators::AND_BOOL_2`), `AND_QO.OUT` ⟶ `QO`. `AND_QO.REQ` wird von `E_MERGE_QO` (`E_MERGE_3`) getriggert, das `ColorAux.INITO`, `PictureAux.CNF` und `ColorAux.CNF` zusammenführt.

Allgemeines Muster siehe SwitchPic(Col)-Bausteine (gemeinsames Muster).

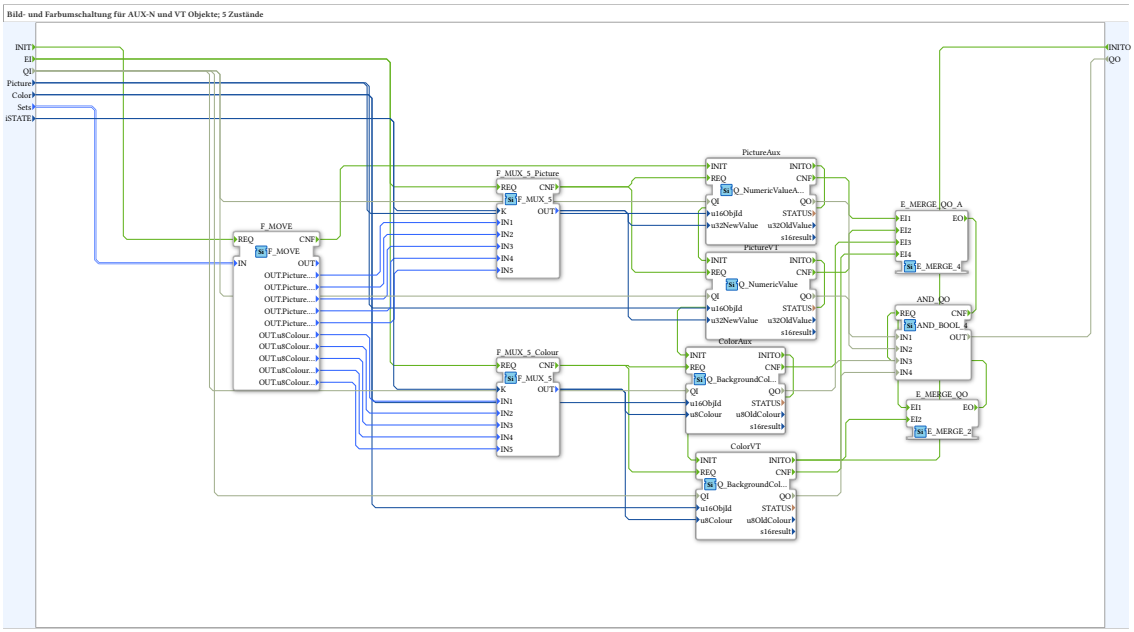
Zusammenfassung

AUX-Gegenstück zu `SwitchPicCol_5_1`.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SwitchPicCol_5_2

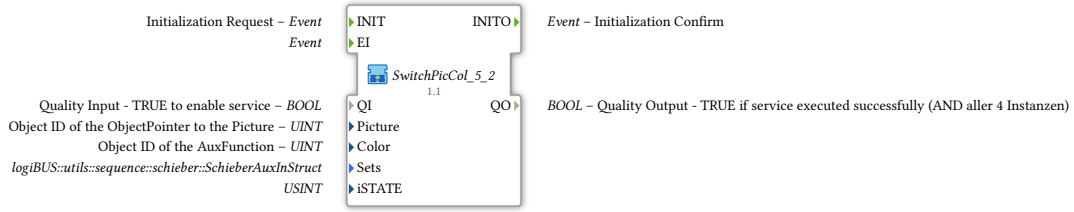


• – ○ • – ○ • – ○ •

Einleitung

`SwitchPicCol_5_2` schaltet zwischen 5 Zuständen sowohl Bild als auch Hintergrundfarbe um — auf **vier** Zielen gleichzeitig: normales Bild-Objekt (`PictureVT`), AUX-Bild-Objekt (`PictureAux`), normales Farb-Objekt (`ColorVT`) und AUX-Farb-Objekt (`ColorAux`), alle über dieselben zwei `F_MUX_5`-Multiplexer (Bild/Farbe) gesteuert.

Allgemeines Muster siehe SwitchPic(Col)-Bausteine (gemeinsames Muster).



Initialisierung und Qualität

Der Baustein besitzt den Ereigniseingang `INIT`, den Ereignisausgang `INITO` (je `EInit`), den Dateneingang `QI` (`BOOL`, Default `TRUE`) und den Datenausgang `QO` (`BOOL`). Die Initialisierung läuft als Kette: `INIT` ⟶ `F_MOVE.REQ`, `F_MOVE.CNF` ⟶ `PictureAux.INIT`, `PictureAux.INITO` ⟶ `PictureVT.INIT`, `PictureVT.INITO` ⟶ `ColorAux.INIT`, `ColorAux.INITO` ⟶ `ColorVT.INIT`, `ColorVT.INITO` ⟶ `INITO`. `QI` geht an `PictureAux`, `PictureVT`, `ColorAux` und `ColorVT`. `QO` ist der Ausgang der Instanz `AND_QO` (`iec61131::booleanOperators::AND_BOOL_4`), die deren vier `QO` verknüpft. Ihr `REQ` wird von `E_MERGE_QO` (`E_MERGE_2`) ausgelöst, das den Ausgang von `E_MERGE_QO_A` (`E_MERGE_4`, führt die `CNF` von `PictureAux`, `PictureVT`, `ColorAux` und `ColorVT` zusammen) und `ColorVT.INITO` zusammenführt.

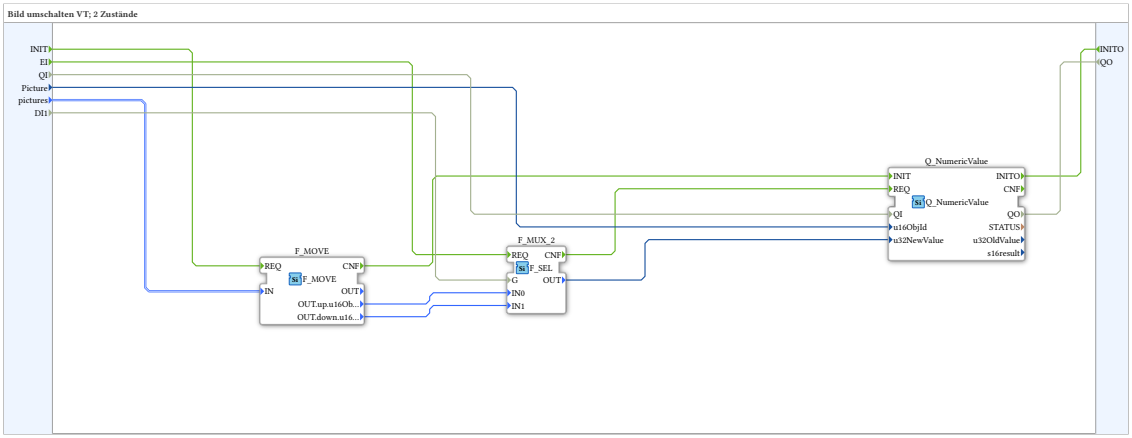
Zusammenfassung

Erweiterung von `SwitchPicCol_5_1` um die AUX-Objekte für Bild und Farbe gleichzeitig — die umfangreichste Variante der `SwitchPicCol`-Reihe.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SwitchPic_2_1

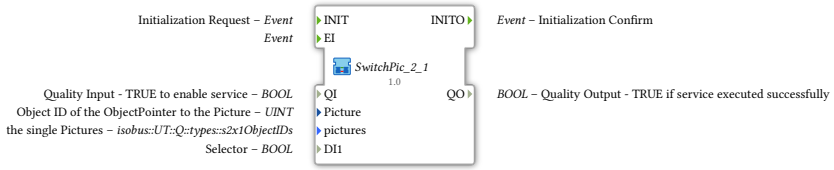


• – ○ • – ○ • – ○ •

Einleitung

`SwitchPic_2_1` schaltet ein VT-Bild (Object-Pointer) zwischen 2 Zuständen (`up`/`down`) anhand eines booleschen Selectors (`DI1`) um — nur das normale VT-Objekt (`Picture`, via `Q_NumericValue`), kein AUX-Objekt.

Allgemeines Muster siehe SwitchPic(Col)-Bausteine (gemeinsames Muster).



Schnittstelle (QI/QO)

Der Baustein besitzt den Dateneingang `QI` (`BOOL`, Standardwert `TRUE`, Quality Input) und den Datenausgang `QO` (`BOOL`, Quality Output). `QI` wird direkt an `Q_NumericValue.QI` und `Q_NumericValue.QO` direkt an `QO` durchgereicht.

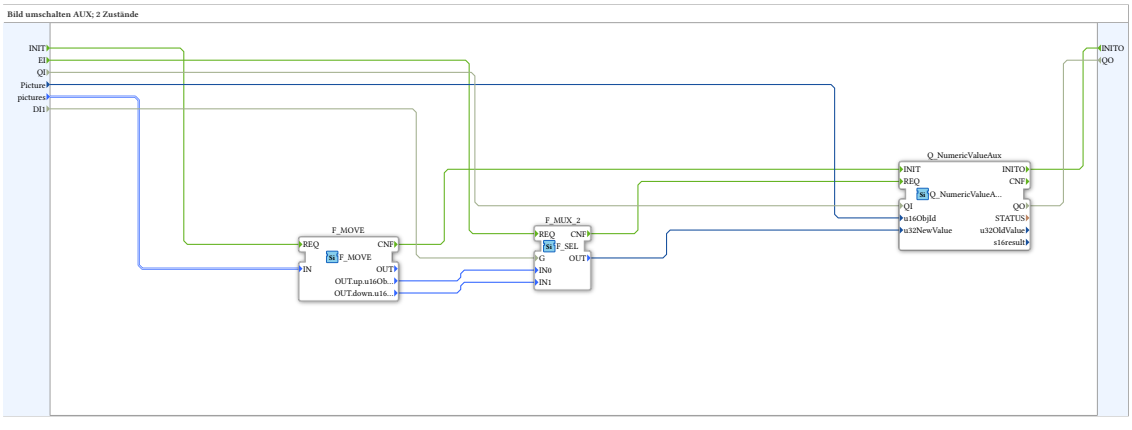
Zusammenfassung

Variante „1“ (nur normales Objekt) der 2-Zustände-Bildumschaltung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SwitchPic_2_1_aux

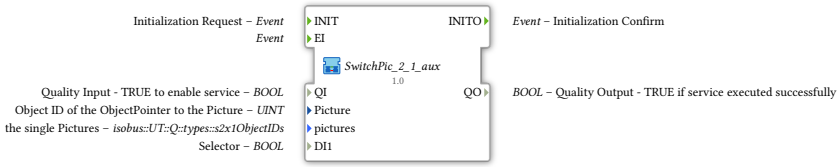


• – ○ • – ○ • – ○ •

Einleitung

`SwitchPic_2_1_aux` schaltet ein VT-Bild (Object-Pointer) zwischen 2 Zuständen (`up`/`down`) anhand eines booleschen Selectors (`DI1`) um — ausschließlich für ein **Auxiliary-Function**-Objekt (`Q_NumericValueAux`), kein normales VT-Objekt.

Allgemeines Muster siehe SwitchPic(Col)-Bausteine (gemeinsames Muster).



Schnittstelle (QI/QO)

Der Baustein besitzt den Dateneingang `QI` (`BOOL`, Standardwert `TRUE`, Quality Input) und den Datenausgang `QO` (`BOOL`, Quality Output). `QI` wird direkt an `Q_NumericValueAux.QI` und `Q_NumericValueAux.QO` direkt an `QO` durchgereicht.

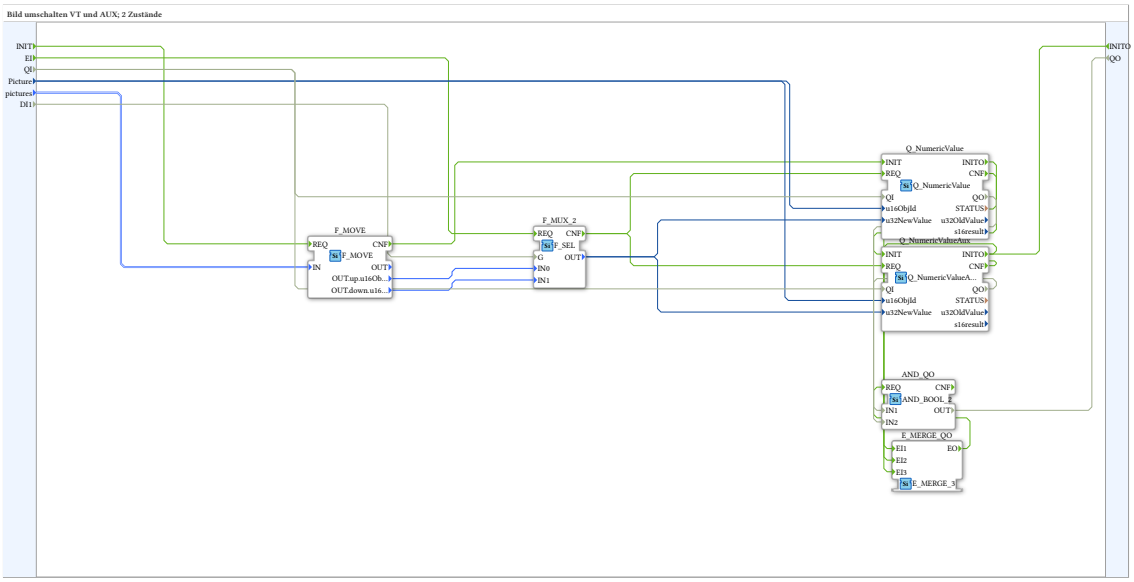
Zusammenfassung

AUX-Gegenstück zu `SwitchPic_2_1`: dieselbe 2-Zustände-Logik, aber nur für ein AUX-Objekt statt ein normales VT-Objekt.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SwitchPic_2_2

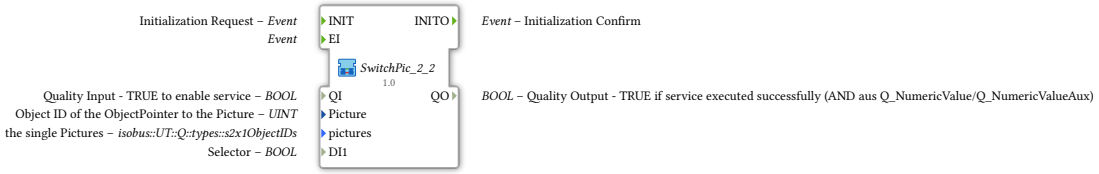


• — ○ • — ○ • — ○ •

Einleitung

`SwitchPic_2_2` schaltet ein VT-Bild zwischen 2 Zuständen (`up`/`down`) anhand eines booleschen Selectors (`DI1`) um — sowohl auf einem normalen VT-Objekt (`Picture`, `Q_NumericValue`) als auch gleichzeitig auf einem **Auxiliary-Function**-Objekt (`Q_NumericValueAux`), beide mit derselben Object-ID (`Picture`).

Allgemeines Muster siehe SwitchPic(Col)-Bausteine (gemeinsames Muster).



Initialisierung und Qualität

- **Ereignis-Eingänge** `INIT` (`EInit`) und `EI` (Umschaltanforderung); **Ereignis-Ausgang** `INITO` (`EInit`).
- **Dateneingang** **`QI`** (`BOOL`, Vorgabe `TRUE`) ⟶ `Q_NumericValue.QI` und `Q_NumericValueAux.QI`; **Datenausgang** **`QO`** (`BOOL`): UND aus `Q_NumericValue.QO` und `Q_NumericValueAux.QO` (`AND_QO`, `iec61131::booleanOperators::AND_BOOL_2`).
- Initialisierung: `INIT` ⟶ `F_MOVE.REQ`; `F_MOVE.CNF` ⟶ `Q_NumericValue.INIT`; `Q_NumericValue.INITO` ⟶ `Q_NumericValueAux.INIT`; `Q_NumericValueAux.INITO` ⟶ `INITO`.
- Der `AND_QO.REQ`-Trigger kommt von `E_MERGE_QO` (`iec61499::events::E_MERGE_3`), das `Q_NumericValueAux.INITO`, `Q_NumericValue.CNF` und `Q_NumericValueAux.CNF` vereint.

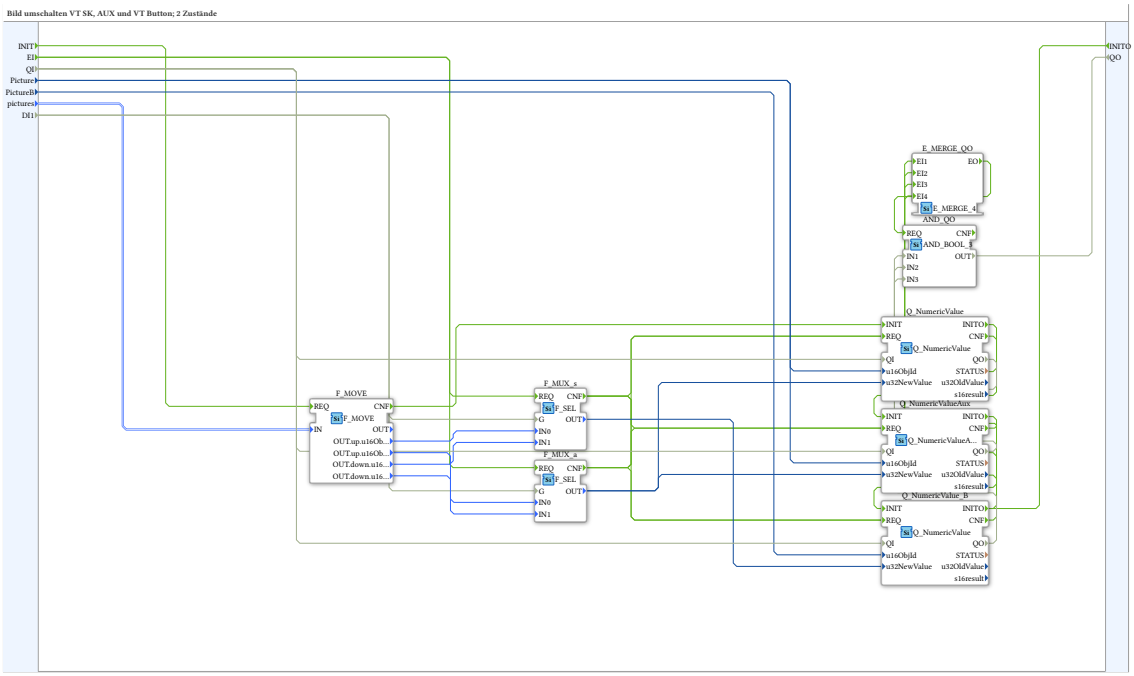
Zusammenfassung

Variante „2“ (normales Objekt + AUX-Objekt) der 2-Zustände-Bildumschaltung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SwitchPic_2_3



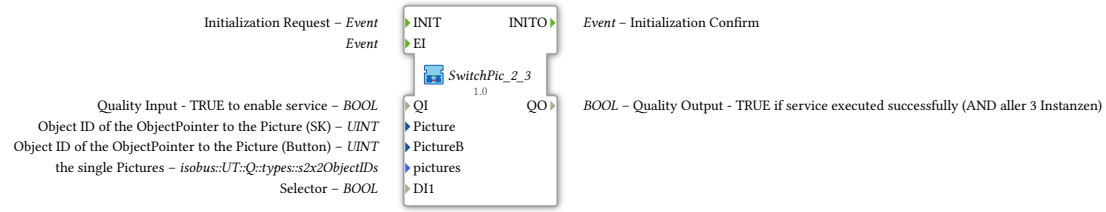
• – ○ • – ○ • – ○ •

Einleitung

`SwitchPic_2_3` schaltet ein VT-Bild zwischen 2 Zuständen (`up`/`down`) anhand eines booleschen Selectors (`DI1`) um — auf drei Zielen gleichzeitig: einem normalen Softkey-Objekt (`Picture`), einem AUX-Objekt (dieselbe ID `Picture`) und einem zweiten, unabhängigen normalen Objekt („Button“, `PictureB`, eigene Object-ID-Untermenge `u16ObjIdA` in der Struktur).

Schnittstelle (Quality-Muster): Dateneingang `QI` (`BOOL`, Vorgabe `TRUE`) wird an `Q_NumericValue`, `Q_NumericValueAux` und `Q_NumericValue_B` verteilt; der Datenausgang `QO` (`BOOL`) ist die UND-Verknüpfung (`AND_QO`: `iec61131::booleanOperators::AND_BOOL_3`) der `QO`-Ausgänge dieser drei Instanzen (`IN1` bis `IN3` in dieser Reihenfolge). Den Trigger `AND_QO.REQ` liefert `E_MERGE_QO` (`E_MERGE_4`), der `Q_NumericValue_B.INITO` sowie die `CNF` von `Q_NumericValue`, `Q_NumericValueAux` und `Q_NumericValue_B` zusammenführt.

Allgemeines Muster siehe SwitchPic(Col)-Bausteine (gemeinsames Muster).



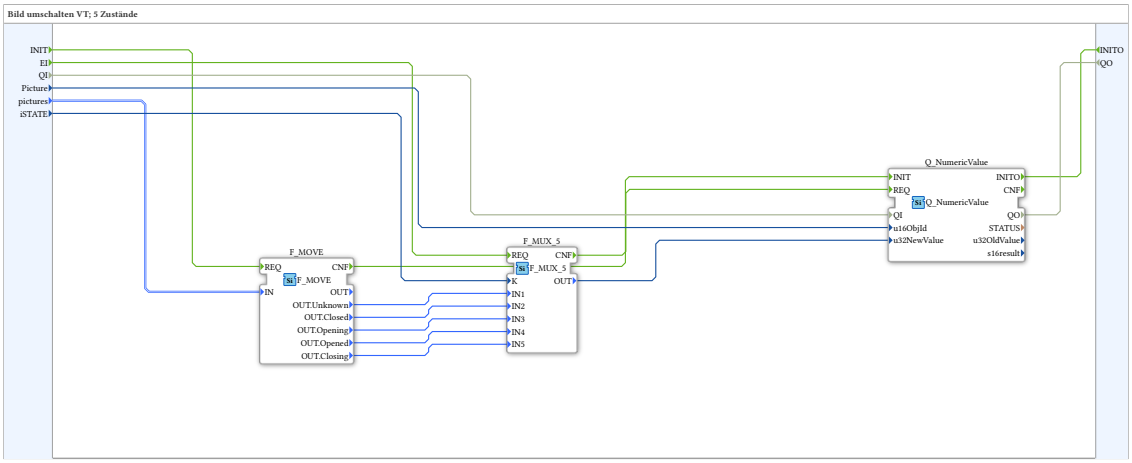
Zusammenfassung

Variante „3“ (normales Softkey-Objekt + AUX-Objekt + zweites normales Button-Objekt) der 2-Zustände-Bildumschaltung — die umfangreichste Variante der `SwitchPic_2_*`-Reihe.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SwitchPic_5_1

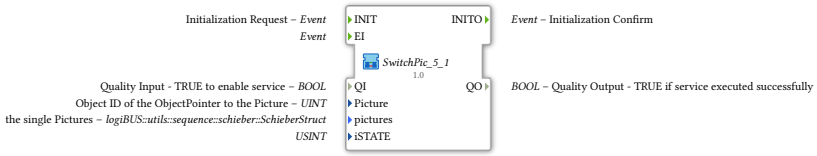


• – ○ • – ○ • – ○ •

Einleitung

`SwitchPic_5_1` schaltet ein VT-Bild zwischen 5 Zuständen (Schieber-/Ventilanimation: `Unknown`, `Closed`, `Opening`, `Opened`, `Closing`) anhand eines `iSTATE`-Werts (`USINT`) um — nur das normale VT-Objekt (`Picture`, via `Q_NumericValue`), kein AUX-Objekt.

Allgemeines Muster siehe SwitchPic(Col)-Bausteine (gemeinsames Muster).



Schnittstelle (QI/QO)

Der Baustein besitzt den Dateneingang `QI` (`BOOL`, Standardwert `TRUE`, Quality Input) und den Datenausgang `QO` (`BOOL`, Quality Output). `QI` wird direkt an `Q_NumericValue.QI` und `Q_NumericValue.QO` direkt an `QO` durchgereicht.

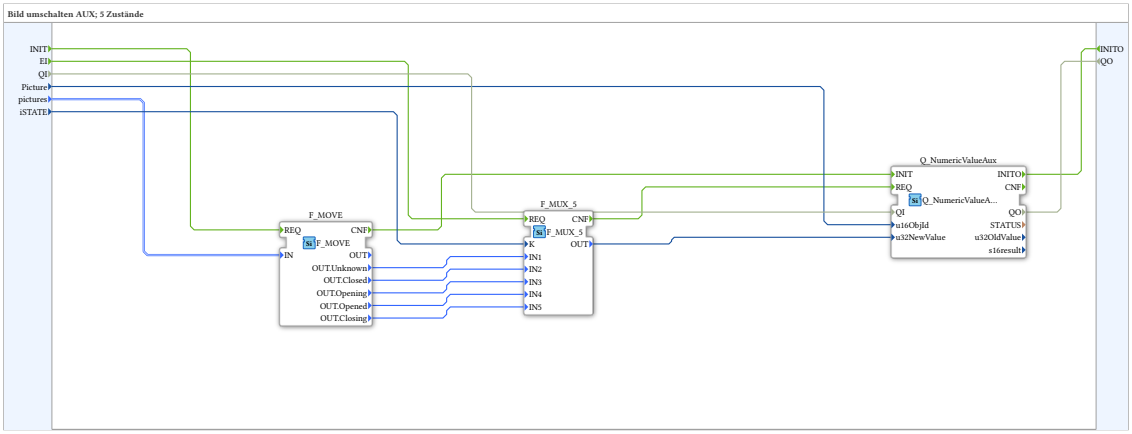
Zusammenfassung

Variante „1“ (nur normales Objekt) der 5-Zustände-Bildumschaltung — das 5-Zustände-Gegenstück zu `SwitchPic_2_1`.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SwitchPic_5_1_aux

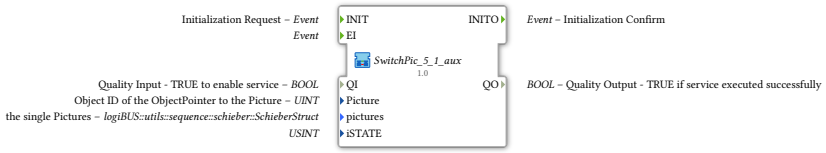


• – ○ • – ○ • – ○ •

Einleitung

`SwitchPic_5_1_aux` schaltet ein VT-Bild zwischen 5 Zuständen (Schieber-/Ventilanimation) anhand eines `iSTATE`-Werts um — ausschließlich für ein **Auxiliary-Function**-Objekt (`Q_NumericValueAux`), kein normales VT-Objekt.

Allgemeines Muster siehe SwitchPic(Col)-Bausteine (gemeinsames Muster).



Schnittstelle (QI/QO)

Der Baustein besitzt den Dateneingang `QI` (`BOOL`, Standardwert `TRUE`, Quality Input) und den Datenausgang `QO` (`BOOL`, Quality Output). `QI` wird direkt an `Q_NumericValueAux.QI` und `Q_NumericValueAux.QO` direkt an `QO` durchgereicht.

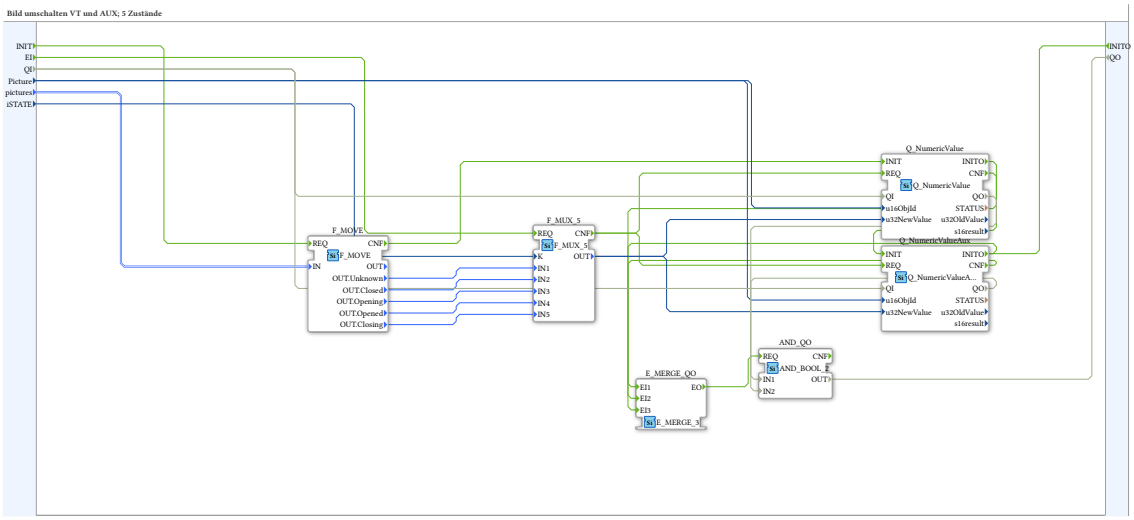
Zusammenfassung

AUX-Gegenstück zu `SwitchPic_5_1`.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SwitchPic_5_2

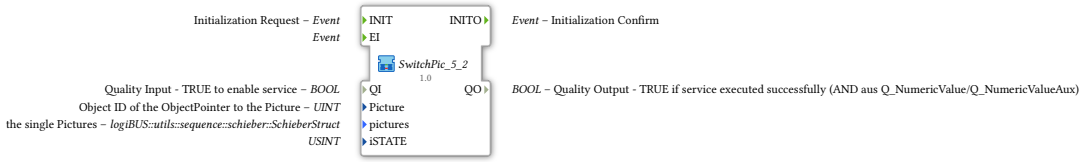


• – ○ • – ○ • – ○ •

Einleitung

`SwitchPic_5_2` schaltet ein VT-Bild zwischen 5 Zuständen (Schieber-/Ventilanimation) anhand eines `iSTATE`-Werts um — sowohl auf einem normalen VT-Objekt (`Picture`) als auch gleichzeitig auf einem AUX-Objekt (dieselbe ID `Picture`).

Allgemeines Muster siehe SwitchPic(Col)-Bausteine (gemeinsames Muster).



Initialisierung und Status (INIT/INITO, QI/QO)

- **Ereignis-Eingang** `INIT` (`EInit`) und **Ereignis-Ausgang** `INITO` (`EInit`)
- **Daten-Eingang** `QI` (`BOOL`, Vorgabe `TRUE`, Quality Input) wird an `Q_NumericValue.QI` und `Q_NumericValueAux.QI` weitergegeben
- **Daten-Ausgang** `QO` (`BOOL`, Quality Output): UND (`AND_QO`, Typ `iec61131::booleanOperators::AND_BOOL_2`) der `QO`-Ausgänge von `Q_NumericValue` und `Q_NumericValueAux`
- **Initialisierungskette**: `INIT` → `F_MOVE.REQ`, `F_MOVE.CNF` → `Q_NumericValue.INIT`, `Q_NumericValue.INITO` → `Q_NumericValueAux.INIT`, `Q_NumericValueAux.INITO` → `INITO`
- `E_MERGE_QO` (Typ `iec61499::events::E_MERGE_3`) führt `Q_NumericValueAux.INITO`, `Q_NumericValue.CNF` und `Q_NumericValueAux.CNF` zusammen und löst `AND_QO.REQ` aus

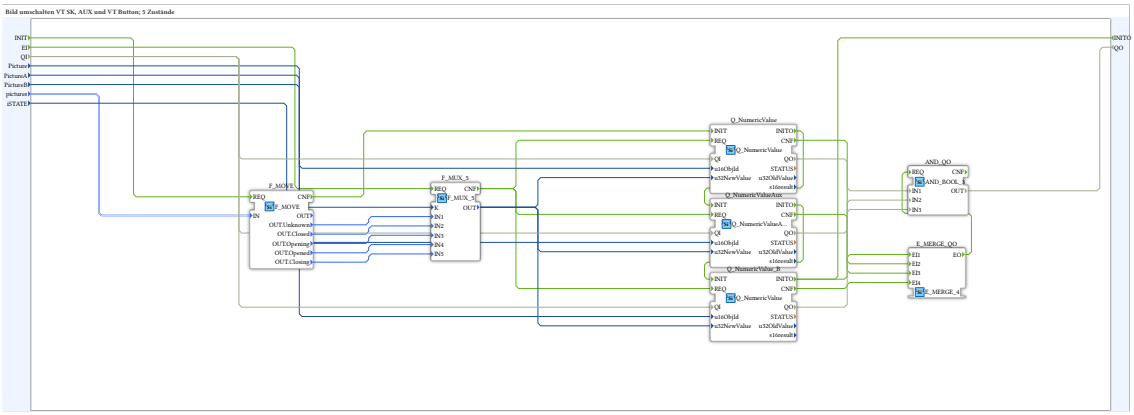
Zusammenfassung

5-Zustände-Gegenstück zu `SwitchPic_2_2`.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SwitchPic_5_3



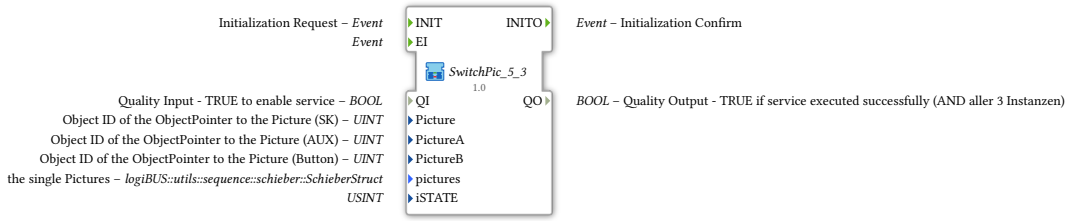
• — ○ • — ○ • — ○ •

Einleitung

`SwitchPic_5_3` schaltet ein VT-Bild zwischen 5 Zuständen (Schieber-/Ventilanimation) anhand eines `iSTATE`-Werts um — auf drei unabhängigen Zielen gleichzeitig: normales Softkey-Objekt (`Picture`), AUX-Objekt (`PictureA`) und zweites normales Button-Objekt (`PictureB`), jedes mit eigener Object-ID.

Allgemeines Muster siehe SwitchPic(Col)-Bausteine (gemeinsames Muster).

Initialisierung und Qualität: `INIT` ⟶ `F_MOVE.REQ`; `F_MOVE.CNF` ⟶ `Q_NumericValue.INIT` ⟶ `Q_NumericValueAux.INIT` ⟶ `Q_NumericValue_B.INIT` (jeweils über `INITO`); `Q_NumericValue_B.INITO` ⟶ `INITO`. Neu ist `QI` (`BOOL`, Vorgabe `TRUE`), das an alle drei `Q_NumericValue`-Instanzen geht, sowie `QO` (`BOOL`): `AND` (`AND_QO`, `AND_BOOL_3`) der drei `QO`-Ausgänge. `AND_QO.REQ` wird über `E_MERGE_QO` (`E_MERGE_4`: `Q_NumericValue_B.INITO`, `Q_NumericValue.CNF`, `Q_NumericValueAux.CNF`, `Q_NumericValue_B.CNF`) ausgelöst.



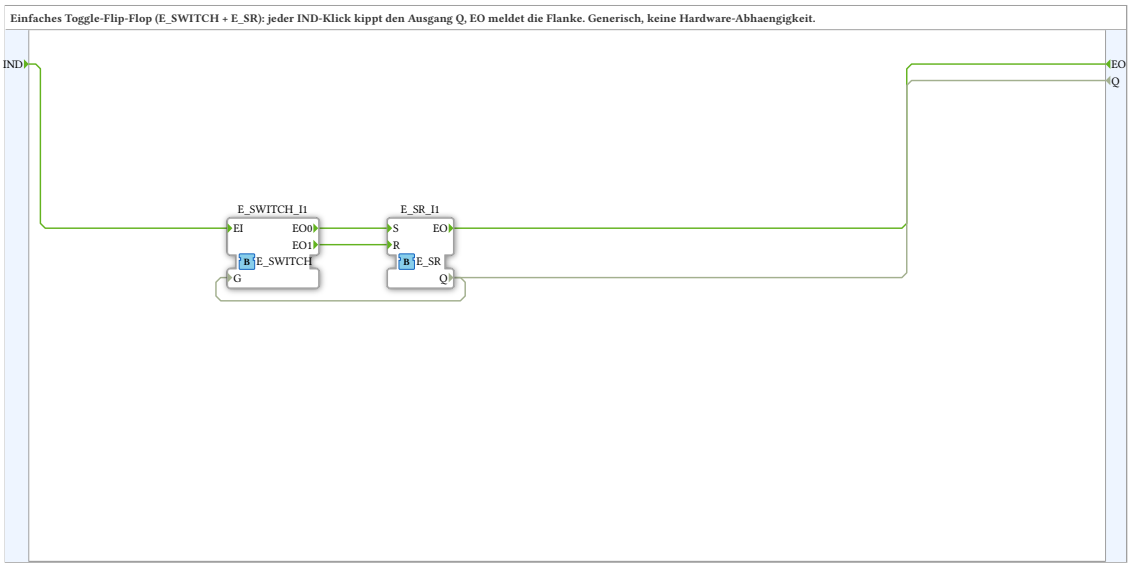
Zusammenfassung

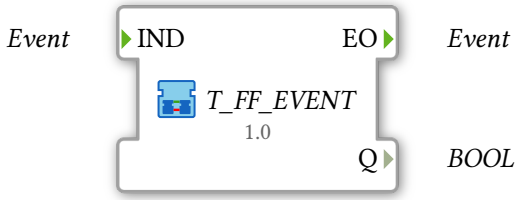
5-Zustände-Gegenstück zu `SwitchPic_2_3` — die umfangreichste Variante der `SwitchPic_5_*`-Reihe.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

T_FF_EVENT





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **T_FF_EVENT** ist ein Subapplikationstyp, der ein einfaches Toggle-Flip-Flop realisiert. Er verwendet die Grundbausteine `E_SWITCH` und `E_SR`, um bei jedem Ereignis am Eingang `IND` den Ausgangszustand `Q` zu kippen. Das Ereignis `EO` wird nach jedem Schaltvorgang ausgegeben. Der Baustein ist bewusst generisch aufgebaut – er besitzt keinerlei Hardware-Abhängigkeit und kann in unterschiedlichsten SPS- und Automatisierungsprojekten eingesetzt werden.

Schnittstellenstruktur

Ereignis-Eingänge

- **IND** : Eingangsereignis, das den Toggle-Vorgang auslöst. Jeder Impuls an `IND` verändert den Zustand von `Q`.

Ereignis-Ausgänge

- **EO** : Ausgangsereignis, das nach erfolgreicher Zustandsänderung (nach jeder Toggle-Aktion) gesendet wird. Es signalisiert die Verarbeitung des `IND`-Ereignisses.

Daten-Eingänge

- Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

- **Q** (BOOL) : Aktueller Zustand des Flip-Flops. Er wird bei jedem `IND`-Ereignis invertiert (von „false“ auf „true“ oder umgekehrt).

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Die interne Logik besteht aus den Bausteinen `E_SWITCH` und `E_SR`. Der Ereignis-Eingang `IND` ist mit dem Ereigniseingang `EI` des `E_SWITCH` verbunden. Der Datenausgang `Q` des `E_SR` wird auf den Eingang `G` (Gate) des `E_SWITCH` geführt. Je nach Zustand von `Q` leitet der `E_SWITCH` das ankommende Ereignis entweder an den Ausgang `EO0` (wenn `G = false`, also `Q = false`) oder an `EO1` (wenn `G = true`, also `Q = true`). Diese Ausgänge sind mit dem Setz- (`S`) bzw. Rücksetz-Eingang (`R`) des `E_SR` verbunden. Dadurch wird bei `Q = false` der `E_SR` gesetzt (Q wird `true`), bei `Q = true` wird er zurückgesetzt (Q wird `false`). Das Ausgangsereignis `EO` des `E_SR` wird als `EO` nach außen gegeben und der aktuelle Zustand `Q` wird auch als Datenausgang `Q` bereitgestellt. Damit entsteht ein klassisches Toggle-Verhalten: Jeder `IND`-Impuls invertiert den Zustand.

Technische Besonderheiten

- **Generische Bauweise**: Der Baustein ist vollständig in IEC 61499 implementiert und benötigt keine spezielle Hardware-Unterstützung.
- **Subapplikation**: Als SubAppType kann er in übergeordneten Anwendungen mehrfach instantiiert werden, wodurch eine Wiederverwendung in verschiedenen Projekten möglich wird.
- **Einfache Struktur**: Durch die Kombination von nur zwei Basisbausteinen bleibt die Logik transparent und leicht nachvollziehbar.
- **Kein Daten-Eingang**: Es sind keine externen Daten zum Konfigurieren erforderlich; der Baustein arbeitet ausschließlich ereignisgesteuert.

Zustandsübersicht

Der Baustein besitzt zwei stabile Zustände, die durch das Boolesche Ausgangssignal `Q` repräsentiert werden:

- **Zustand Q = false (0)**: Beim nächsten `IND`-Ereignis wird `Q` auf `true` gesetzt (Setzen über `S`).
- **Zustand Q = true (1)**: Beim nächsten `IND`-Ereignis wird `Q` auf `false` gesetzt (Rücksetzen über `R`).

Die Zustandsübergänge erfolgen ausschließlich bei einem Ereignis an `IND`. Wichtig: Es existiert kein weiterer Steuereingang – der Wechsel ist deterministisch und alternierend.

Anwendungsszenarien

- **Taster-Schaltung**: Ein impulsförmiger Taster (z. B. ein Lichttaster) kann mit `T_FF_EVENT` so realisiert werden, dass jeder Druck den Zustand einer Lampe oder eines Motors umschaltet.
- **Betriebsart-Umschaltung**: In Maschinensteuerungen kann zwischen zwei Betriebsmodi hin- und hergeschaltet werden.
- **Test- und Simulationsumgebungen**: Als einfacher Logik-Baustein dient er zur Erzeugung von Toggle-Verhalten in Test-Szenarien.
- **Blinkgenerator (mit zusätzlichem Takt)**: Kombiniert mit einem Timer kann ein periodisches Ein-/Ausschalten erzeugt werden.

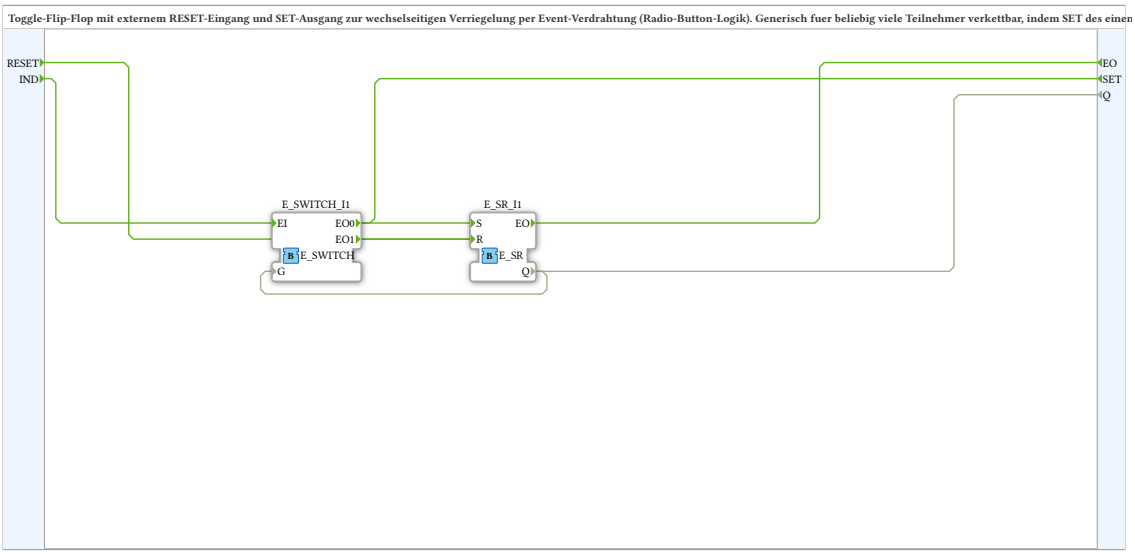
Vergleich mit ähnlichen Bausteinen

- **E_SR (Set/Reset-Flip-Flop)**: Das einfache `E_SR` wechselt den Zustand nur abhängig von separaten Set- und Reset-Ereignissen. Es besitzt keine interne Toggle-Funktion – **T_FF_EVENT** erweitert dies um die automatische Zustandsänderung bei jedem Ereignis.
- **E_RS (Reset/Set-Flip-Flop, priorisiert)**: Ähnlich wie `E_SR`, aber mit anderer Prioritätenlogik. Auch hier ist kein Toggle möglich.
- **T_FF mit Flankenauswertung**: Manche Bausteine benötigen einen zusätzlichen Eingang für die Flankenart (steigend/fallend). **T_FF_EVENT** ist dagegen ein Ereignis-getriebener Baustein, der direkt auf jedes Ereignis reagiert und dadurch einfacher in industriellen Steuerungen einsetzbar ist.

Fazit

Der Funktionsblock **T_FF_EVENT** stellt eine kompakte und zuverlässige Lösung zur Realisierung eines Toggle-Flip-Flops dar. Durch die Verwendung von Standard-Bausteinen und den Verzicht auf externe Abhängigkeiten eignet er sich hervorragend für modulare Wiederverwendung in unterschiedlichen Automatisierungsprojekten. Seine Ereignissteuerung passt nahtlos in das IEC-61499-Konzept und bietet eine klare, nachvollziehbare Funktionalität.

T_FF_ILOCK_EVENT



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **T_FF_ILOCK_EVENT** ist ein Toggle-Flip-Flop mit externem RESET-Eingang und einem SET-Ausgang zur wechselseitigen Verriegelung per Event-Verdrahtung. Er wurde konzipiert für die Umsetzung von Radio-Button-Logik, bei der aus einer Gruppe von Teilnehmern immer genau einer aktiv ist. Durch eine einfache Verkettung – SET des einen Bausteins mit RESET des nächsten verbunden – lässt sich eine beliebig große Kette verriegelter Toggles realisieren.

Schnittstellenstruktur

Ereignis-Eingänge

- **RESET** – Ereignis, das das Flip-Flop zurücksetzt (setzt Q auf FALSE).
- **IND** – Ereignis, das das Umschalten (Toggle) auslöst. Abhängig vom aktuellen Zustand Q wird entweder ein Setzen oder ein Rücksetzen ausgeführt.

Ereignis-Ausgänge

- **EO** – Ausgangs-Ereignis, das nach jeder Zustandsänderung (Setzen oder Rücksetzen) erzeugt wird.
- **SET** – Ereignis, das nur bei einem Umschalten von Q = FALSE auf Q = TRUE ausgegeben wird. Dieses Signal dient zur Verkettung mit nachfolgenden Teilnehmern.

Daten-Eingänge

Keine.

Daten-Ausgänge

- **Q** (BOOL) – Aktueller Zustand des Flip-Flops (FALSE oder TRUE).

Adapter

Keine.

Funktionsweise

Der Baustein nutzt intern zwei Standardbausteine der IEC 61499-Bibliothek: einen **E_SR** (Set/Reset-Flip-Flop) und einen **E_SWITCH** (Ereignis-Schalter). Der E_SWITCH erhält das eingehende Ereignis **IND** und leitet es abhängig vom Daten-Eingang G weiter, der mit dem aktuellen Ausgang Q verbunden ist:

- Ist Q = FALSE, wird das Ereignis an den **Set**-Eingang (S) des E_SR durchgeschaltet. Dadurch wird Q auf TRUE gesetzt und das Ereignis **SET** sowie das Echo **EO** ausgegeben.
- Ist Q = TRUE, wird das Ereignis an den **Reset**-Eingang (R) des E_SR weitergeleitet. Q wird auf FALSE gesetzt, und es wird nur das Echo **EO** erzeugt.

Das externe Ereignis **RESET** ist ebenfalls mit dem Reset-Eingang des E_SR verbunden und setzt das Flip-Flop unabhängig vom aktuellen Zustand auf FALSE zurück. In jedem Fall wird nach einer Zustandsänderung das Ereignis **EO** ausgelöst.

Die Kombination aus E_SWITCH und E_SR realisiert also ein Toggle-Verhalten, bei dem jedes **IND**-Ereignis den Zustand umschaltet, solange kein externer Reset aktiv ist.

Technische Besonderheiten

- Verwendung standardisierter Bausteine (**E_SR**, **E_SWITCH**) aus der IEC 61499-Bibliothek, was eine hohe Kompatibilität und einfache Wartung gewährleistet.
- Keine Daten-Eingänge – die Steuerung erfolgt ausschließlich über Ereignisse, was die Verkettung mehrerer Bausteine per Event-Logik erleichtert.
- Der dedizierte **SET**-Ausgang ermöglicht eine direkte Kaskadierung: Verbindet man SET eines Teilnehmers mit RESET des nächsten, so wird beim Aktivieren eines Teilnehmers automatisch der folgende zurückgesetzt – genau die gewünschte Sperrlogik.
- Die SubApp ist **generisch** einsetzbar, da sie keine feste Anzahl von Teilnehmern voraussetzt.

Zustandsübersicht

Q (vorher)	Ereignis	Q (nachher)	Ausgangs-Ereignisse
FALSE	IND	TRUE	EO, SET
TRUE	IND	FALSE	EO
FALSE	RESET	FALSE	EO
TRUE	RESET	FALSE	EO

Anwendungsszenarien

- **Radio-Button-Gruppen** – mehrere Schalter, bei denen stets nur einer aktiv sein darf (z. B. Betriebsartenauswahl).
- **Wechselseitige Sperre** – Verriegelung von Motoren oder Ventilen, sodass nicht mehrere gleichzeitig eingeschaltet werden können.
- **Zustandsketten** – in Automatisierungsabläufen, bei denen ein Teilnehmer die Freigabe für den nächsten gibt und ihn gleichzeitig sperrt.

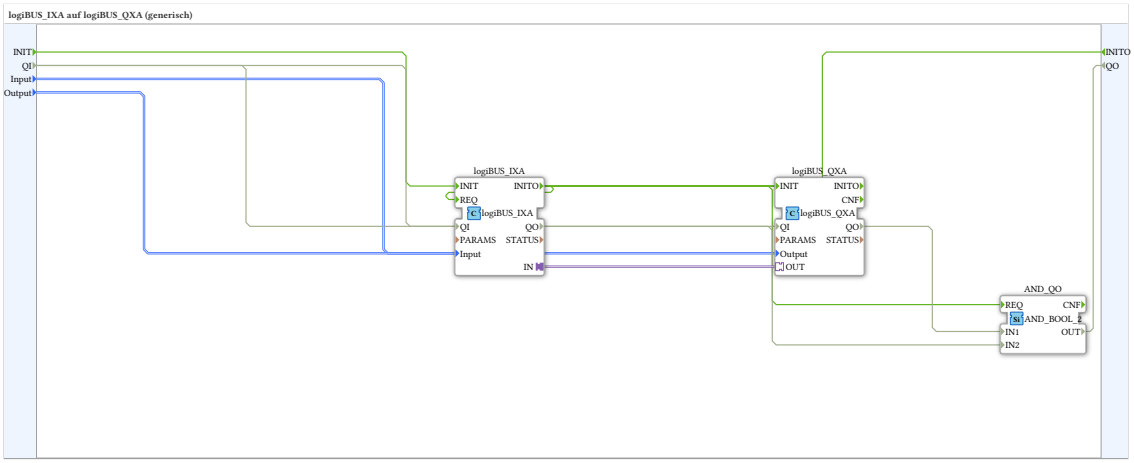
Vergleich mit ähnlichen Bausteinen

- **Einfaches Toggle-Flip-Flop (z. B. mit E_CTUD oder SR)**: Diese besitzen oft keinen separaten SET-Ausgang oder keinen externen Reset-Eingang, wodurch eine einfache Verkettung nicht möglich ist.
- **SR-Flip-Flop (E_SR)**: Erlaubt direktes Setzen und Rücksetzen über Ereignisse, bietet aber keine Toggle-Funktion und keinen Ereignisausgang für Sperrlogik.
- **E_SWITCH-basierte Konstruktionen**: Andere Realisierungen des Toggles sind meist weniger kompakt oder benötigen zusätzliche Logik zur Verarbeitung der Ausgangsereignisse.

Fazit

Der Baustein **T_FF_ILOCK_EVENT** stellt eine kompakte und flexible Lösung für eine verriegelte Toggle-Funktion dar. Durch die Kombination von Set/Reset-Verhalten und gezielter Ereignisausgabe lassen sich mit minimalem Aufwand komplexe Sperrsysteme aufbauen. Die saubere Trennung von Ereignis- und Datenpfaden sowie die standardisierten Bausteine machen ihn zu einer robusten und wiederverwendbaren Komponente in der IEC-61499-Programmierung.

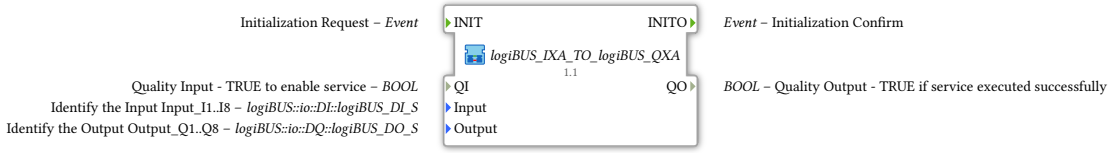
logiBUS_IXA_TO_logiBUS_QXA



• – ○ • – ○ • – ○ •

Einleitung

`logiBUS_IXA_TO_logiBUS_QXA` verdrahtet einen physischen digitalen Eingang (`logiBUS_IXA`) direkt auf einen physischen digitalen Ausgang (`logiBUS_QXA`) — eine reine Hardware-Durchschaltung ohne VT-Beteiligung, adapterbasiert (azyklisch mit Bestätigung, `QI` kommt vom Eingang `QI`). Für die ereignisgesteuerte Variante ohne Adapter siehe `logiBUS_IX_TO_logiBUS_QX`.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: logiBUS_IXA_TO_logiBUS_QXA

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **logiBUS_IXA**: `logiBUS::io::DI::logiBUS_IXA` — physischer digitaler Eingang, Adapter-Ausgang `IN`, `QI` kommt vom Eingang `QI`.
  - **logiBUS_QXA**: `logiBUS::io::DQ::logiBUS_QXA` — physischer digitaler Ausgang, Adapter-Eingang `OUT`, `QI` kommt vom Eingang `QI`.
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_2` — verknüpft `logiBUS_QXA.QO` und `logiBUS_IXA.QO` zum Ausgang `QO`.
- **Funktionsweise**: Der Adapter-Ausgang des Eingangs wird direkt auf den Adapter-Eingang des Ausgangs verdrahtet — keine Zwischenlogik, keine VT-Anbindung. Die SubApp-Schnittstelle besitzt außerdem die Ereignisse `INIT`/`INITO` sowie `QI` (BOOL, Initialwert `TRUE`) und `QO` (BOOL).

Programmablauf und Verbindungen

1. `Input` ⟶ `logiBUS_IXA.Input`; `Output` ⟶ `logiBUS_QXA.Output`.
2. `logiBUS_IXA.IN` (Adapter) ⟶ `logiBUS_QXA.OUT` (Adapter) — direkte Durchschaltung.
3. Initialisierungskette: `INIT` ⟶ `logiBUS_QXA.INIT`, `logiBUS_QXA.INITO` ⟶ `logiBUS_IXA.INIT`, `logiBUS_IXA.INITO` ⟶ `INITO` und `AND_QO.REQ` (zusätzlich `logiBUS_IXA.INITO` ⟶ `logiBUS_IXA.REQ`).
4. Qualität: `QI` ⟶ `logiBUS_QXA.QI` und `logiBUS_IXA.QI`; `logiBUS_QXA.QO` ⟶ `AND_QO.IN1`; `logiBUS_IXA.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`.

Anwendungsszenarien

- Reine Hardware-zu-Hardware-Verdrahtung, z. B. für einen physischen Notaus-Kontakt oder eine feste Verriegelung, die unabhängig vom VT funktionieren muss.

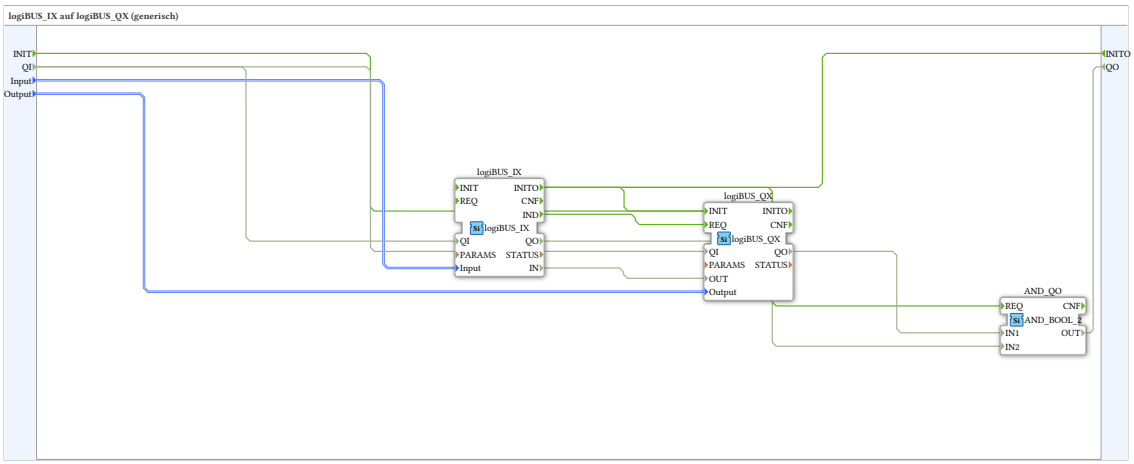
Zusammenfassung

Adapterbasierte Direktverdrahtung eines physischen Eingangs auf einen physischen Ausgang, ohne VT- oder Ereignislogik.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

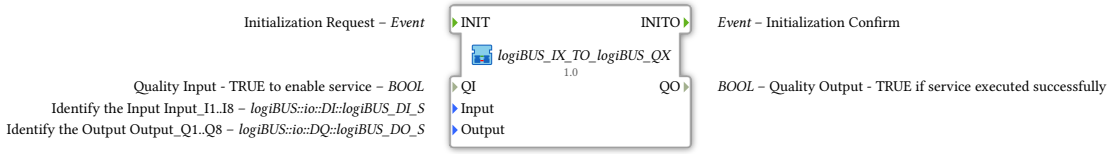
logiBUS_IX_TO_logiBUS_QX



• – ○ • – ○ • – ○ •

Einleitung

`logiBUS_IX_TO_logiBUS_QX` verdrahtet einen physischen digitalen Eingang (`logiBUS_IX`) direkt auf einen physischen digitalen Ausgang (`logiBUS_QX`) — funktional äquivalent zu `logiBUS_IXA_TO_logiBUS_QXA`, jedoch mit den nicht-adapterbasierten Bausteinvarianten `logiBUS_IX`/`logiBUS_QX`: die Verbindung erfolgt über eine explizite Ereignis- und eine Datenverbindung statt über einen Adapter.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: logiBUS_IX_TO_logiBUS_QX

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **logiBUS_IX**: `logiBUS::io::DI::logiBUS_IX` — physischer digitaler Eingang, feuert `IND` bei Zustandsänderung.
  - **logiBUS_QX**: `logiBUS::io::DQ::logiBUS_QX` — physischer digitaler Ausgang, wird über `REQ` angesteuert.
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_2` — verknüpft die `QO` von `logiBUS_QX` und `logiBUS_IX`.
- **Funktionsweise**: Das Eingangsereignis `IND` löst direkt `REQ` am Ausgang aus; der aktuelle Zustand wird parallel über eine Datenverbindung von `IN` nach `OUT` durchgereicht.

Programmablauf und Verbindungen

1. `Input` ⟶ `logiBUS_IX.Input`; `Output` ⟶ `logiBUS_QX.Output`.
2. `logiBUS_IX.IND` ⟶ `logiBUS_QX.REQ` (Ereignisverbindung).
3. `logiBUS_IX.IN` ⟶ `logiBUS_QX.OUT` (Datenverbindung).
4. Initialisierung: Ereigniseingang `INIT` ⟶ `logiBUS_QX.INIT`, `logiBUS_QX.INITO` ⟶ `logiBUS_IX.INIT`, `logiBUS_IX.INITO` ⟶ Ereignisausgang `INITO` und ⟶ `AND_QO.REQ`.

5. Qualität: Dateneingang `QI` (`BOOL`, Default `TRUE`) ⟶ `logiBUS_QX.QI` und `logiBUS_IX.QI`; `logiBUS_QX.QO` ⟶ `AND_QO.IN1`, `logiBUS_IX.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ Datenausgang `QO`.

Anwendungsszenarien

- Hardware-zu-Hardware-Durchschaltung in Trainingssystemen, die (im Gegensatz zu `test_AX`) auf die nicht-adapterbasierten I/O-Bausteine `logiBUS_IX`/`logiBUS_QX` setzen.

Zusammenfassung

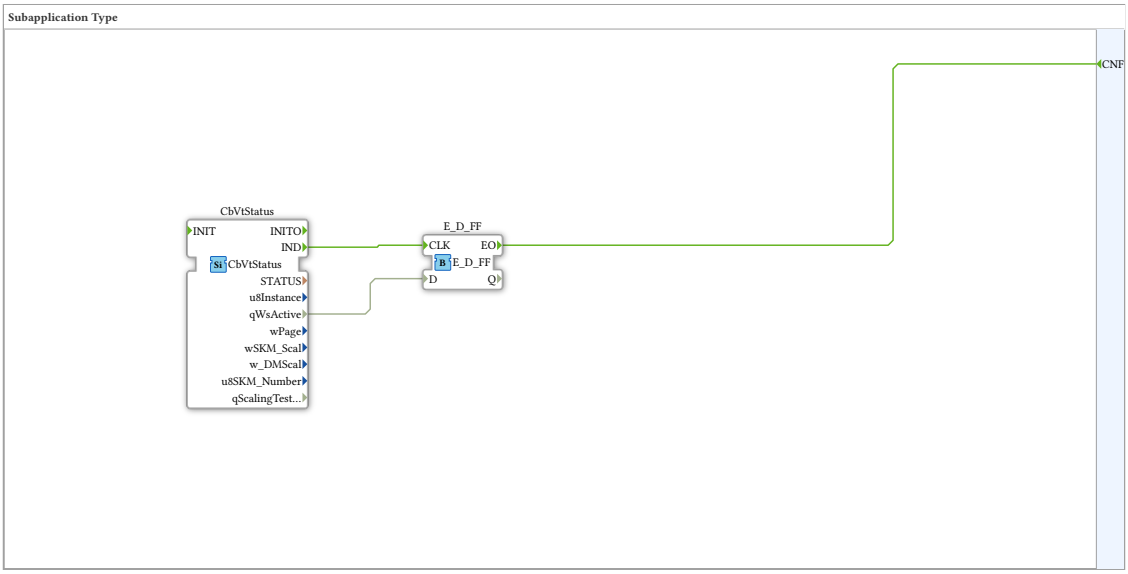
Nicht-adapterbasiertes Gegenstück zu `logiBUS_IXA_TO_logiBUS_QXA`: gleiche Funktion, explizite Ereignis-/Datenverbindung statt Adapterkopplung.

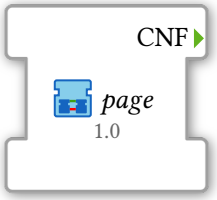
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Page

page





Event – Execution Confirmation

• – ○ • – ○ • – ○ •

Einleitung

Die Subapplication `page` ist ein gekapselter Baustein, der eine einfache Ereignis-Daten-Verarbeitung realisiert. Sie basiert auf zwei internen Funktionsblöcken: einem Status-überwachenden FB (CbVtStatus) und einem flankengetriggerten Flipflop (E_D_FF). Durch die Kombination dieser Bausteine wird ein spezifisches Verhalten erzeugt, das nach außen hin nur über einen Ereignis-Ausgang sichtbar ist.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Name	Typ	Kommentar
CNF	Event	Execution Confirmation

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Keine.

Funktionsweise

Die Subapplication besitzt keine externen Ein- oder Ausgänge außer dem Ereignis-Ausgang `CNF`. Intern sind zwei Funktionsblöcke verschaltet:

1. **CbVtStatus** (Typ: `isobus::UT::status::CbVtStatus`): Dieser Block überwacht einen bestimmten Status (vermutlich eines VT – Virtual Terminal) und gibt bei einer Änderung oder einem Ereignis das Signal `IND` aus. Zusätzlich liefert er das Datensignal `qWsActive` (Workstation aktiv?).
2. **E_D_FF** (Typ: `iec61499::events::E_D_FF`): Ein flankengetriggertes D-Flipflop, das auf die steigende Flanke des Ereignisses `CLK` reagiert. Das Datensignal `D` wird bei jeder Flanke übernommen und der Ausgang `EO` wird als Ereignis ausgelöst, sobald eine gültige Flanke erkannt wurde.

Die Verbindung ist wie folgt:

- Das Ereignis `IND` von `CbVtStatus` wird als Takt (`CLK`) an das Flipflop `E_D_FF` weitergeleitet.
- Das Datensignal `qWsActive` wird auf den Dateneingang `D` des Flipflops gelegt.
- Das Ereignis `EO` des Flipflops ist direkt mit dem externen Ausgang `CNF` verbunden.

Somit wird jedes Mal, wenn `CbVtStatus` ein `IND`-Ereignis erzeugt, das Flipflop getaktet. Der aktuelle Wert von `qWsActive` wird als Zustand übernommen und ein `CNF`-Ereignis wird ausgegeben. Die Bedingung für die Ausgabe ist also das Eintreten eines Ereignisses von `CbVtStatus` – unabhängig davon, ob der Wert von `qWsActive` sich geändert hat oder nicht. Dadurch wird eine Quittung (CNF) an den übergeordneten Baustein gesendet.

Technische Besonderheiten

- Es handelt sich um eine **Subapplication**, die die interne Logik vollständig kapselt. Nur der Ereignisausgang ist nach außen sichtbar; alle internen Verbindungen sind für den Aufrufer unsichtbar.
- Die Verwendung eines D-Flipflops (`E_D_FF`) bewirkt, dass der Zustand des Signals `qWsActive` bei jedem Ereignis `IND` erfasst und für die nächste Verarbeitung gespeichert wird. Dies kann zur Zustandsspeicherung oder zur Erzeugung eines einmaligen Quittungssignals verwendet werden.
- Die beiden Funktionsblöcke stammen aus unterschiedlichen Bibliotheken (`isobus` und `iec61499`), was auf eine modulare Integration hinweist.

Zustandsübersicht

Da intern ein D-Flipflop verwendet wird, können zwei stabile Zustände unterschieden werden:

- **Zustand Q=0** (Logisch 0): Der aktuelle Wert von `D` (also `qWsActive`) ist 0, und das Flipflop hat diesen Wert übernommen. Ein `CNF`-Ereignis wurde nach der letzten Taktflanke mit diesem Wert ausgegeben.
- **Zustand Q=1** (Logisch 1): Der aktuelle Wert von `D` ist 1, und das Flipflop hat diesen Wert übernommen. Ein `CNF`-Ereignis wurde mit diesem Wert ausgegeben.

Das Flipflop wechselt bei jedem `IND`-Ereignis (Taktflanke) seinen Zustand entsprechend dem aktuellen `D`-Signal. Es gibt keinen Reset-Eingang; der Anfangszustand hängt vom initialen Wert von `qWsActive` nach dem Start der Subapplication ab.

Anwendungsszenarien

- **Statusbenachrichtigung**: Überwachung eines Systems (z.B. eines ISO-Bus-Terminals) und Erzeugen eines Bestätigungsereignisses (`CNF`) bei jedem Statuswechsel, um übergeordnete Steuerungen zu informieren.
- **Ablaufsteuerung**: Einsatz als Trigger für nachfolgende Schritte, die nur bei einem bestimmten Ereignis ausgeführt werden sollen.
- **Test und Diagnose**: Einfache Möglichkeit, ein Ereignis von einer überwachten Komponente zu erhalten und gleichzeitig den Zustand eines Datenwerts intern zu speichern.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen Ereignis- oder Datenbausteinen bietet `page` die Besonderheit, dass es ein Ereignis mit einer Datenspeicherung kombiniert. Ein direkter Ereignis-Weiterleitungsbaustein (z.B. `E_SPLIT`) würde nur das Ereignis weiterleiten, ohne den Dateneingang zu berücksichtigen. Ein reines D-Flipflop ohne Ereignissteuerung würde nur bei explizitem Takt reagieren, hier ist der Takt intern mit dem Ereignis von `CbVtStatus` verbunden.

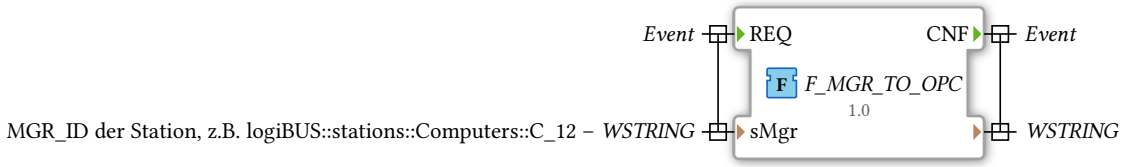
Fazit

Die Subapplication `page` realisiert eine kompakte Funktion: Sie verbindet die Statusüberwachung eines VT-Systems mit einem flankengetriggerten Flipflop und liefert bei jedem Ereignis eine Bestätigung (`CNF`). Durch die Kapselung in einer Subapplication bleibt die interne Logik übersichtlich und kann wiederverwendet werden, ohne dass externe Verbindungen geändert werden müssen. Dies macht sie zu einem nützlichen Baustein für Anwendungen, die eine ereignisbasierte Zustandsübernahme erfordern.

2.1.7 MyLib_OPC

- F_MGR_TO_OPC
- F_OPC_LOCAL
- F_OPC_REMOTE

2.1.7.1 F_MGR_TO_OPC: Funktion (IEC 61499 Function)



• – ◦ • – ◦ • – ◦ •

Einleitung

`F_MGR_TO_OPC` ist keine SubApp, sondern ein Funktionstyp (`.fct`, Standard 61499-1) im Paket `MyLib_OPC` mit dem Kommentar „OPC-UA-Endpunkt aus MGR_ID: 192.168.1.12:61499 -> opc.tcp:

Schnittstelle der Funktion

Instanz	Typ	Parameter	Erklärung
`REQ`	`Event` (Ereigniseingang)	`With = sMgr`	Anforderung zur Ausführung.
`CNF`	`Event` (Ereignisausgang)	`With` = (unbenannte Variable)	Bestätigung; liefert den Rückgabewert.
`sMgr`	`WSTRING` (Eingangsvariable)	-	MGR_ID der Station im Format `host:port`, z. B. `"192.168.1.12:61499"`. Das ist der Wert einer Stationskonstante (`logiBUS::stations::Computers::C_12` hat den Wert `"192.168.1.12:61499"`).

(unbenannt)	`WSTRING` (Ausgangsvariable)	-	Rückgabewert `F_MGR_TO_OPC` (leerer Ausgangsname).

Programmablauf und Verbindungen

Der Rumpf ist in Structured Text geschrieben:

```
PACKAGE MyLib_OPC;

(* OPC-UA-Endpunkt aus MGR_ID: 192.168.1.12:61499 -> opc.tcp://192.168.1.12:4840 *)
FUNCTION F_MGR_TO_OPC : WSTRING

VAR_INPUT
    sMgr : WSTRING; // MGR_ID der Station, z.B. logiBUS::stations::Computers::C_12
END_VAR

F_MGR_TO_OPC := CONCAT("opc.tcp://", LEFT(sMgr, FIND(sMgr, ":")), "4840");

END_FUNCTION
```

Hinweis: Der Quellkommentar `// MGR_ID der Station, z.B. logiBUS::stations::Computers::C_12` nennt die Stationskonstante. `sMgr` bekommt deren Wert im Format `host:port` (hier `"192.168.1.12:61499"`); ein hierarchischer Name ohne Doppelpunkt ergäbe keinen gültigen Endpunkt.

1. `FIND(sMgr, ":")` liefert die Position des ersten Doppelpunkts.
2. `LEFT(sMgr, ...)` schneidet den Text bis einschließlich dieses Doppelpunkts ab (`192.168.1.12:`); der MGR-Port `61499` entfällt.
3. `CONCAT` setzt `opc.tcp://`, diesen Teil und den festen OPC-UA-Port `4840` zusammen.

Beispiel

Aufruf und Ergebnis:

```
MyLib_OPC::F_MGR_TO_OPC("192.168.1.12:61499")  ->  opc.tcp://192.168.1.12:4840
```

Anwendung

Verwendet in `myOpcUaAddresses.gcf` (Projekt `test_VV`, Ordner `const/OPC_UA`, beschrieben in `OPC_UA_Adressen.md`) als `InitialValue` einer WSTRING-Konstante:

```
InitialValue="MyLib_OPC::F_MGR_TO_OPC(PC_A)"
```

`PC_A_OPC` und `PC_B_OPC` erhalten den OPC-UA-Endpunkt der jeweiligen Station über `MyLib_OPC::F_MGR_TO_OPC(PC_A)` bzw. `MyLib_OPC::F_MGR_TO_OPC(PC_B)`.

Der Ausdruck wird beim Export ausgewertet; im Bootfile stehen daher nur die fertigen Strings, keine Funktionsaufrufe.

Hinweis: Die Argumente müssen Konstanten sein (Literale oder andere Konstanten der `.gcf`), da die Auswertung zur Exportzeit erfolgt und nicht zur Laufzeit.

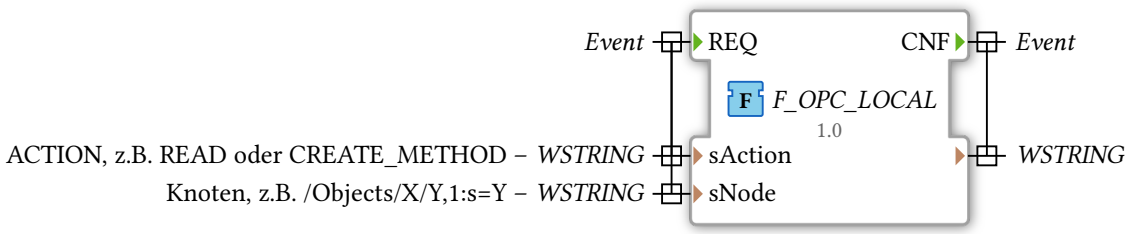
Zusammenfassung

`F_MGR_TO_OPC` ist eine kleine IEC-61499-Funktion der Bibliothek `MyLib_OPC`, die den OPC-UA-Endpunkt aus einer MGR_ID ableitet.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.1.7.2 F_OPC_LOCAL: Funktion (IEC 61499 Function)



• – ◦ • – ◦ • – ◦ •

Einleitung

F_OPC_LOCAL ist keine SubApp, sondern ein Funktionstyp (.fct, Standard 61499-1) im Paket MyLib_OPC mit dem Kommentar „OPC-UA-ID fuer einen lokalen Knoten: opc_ua[ACTION;NODE]“. Er hat einen Ereigniseingang REQ, einen Ereignisausgang CNF und einen Strukturierter-Text-Rumpf; es gibt keine Funktionsbaustein-Instanzen und keine Verbindungen.

Schnittstelle der Funktion

Instanz	Typ	Parameter	Erklärung
REQ	Event (Ereigniseingang)	With = sAction, sNode	Anforderung zur Ausführung.
CNF	Event (Ereignisausgang)	With = (unbenannte Variable)	Bestätigung; liefert den Rückgabewert.
sAction	WSTRING (Eingangsvariable)	-	ACTION, z.B. READ oder CREATE_METHOD
sNode	WSTRING (Eingangsvariable)	-	Knoten, z.B. /Objects/X/Y,1:s=Y
(unbenannt)	WSTRING (Ausgangsvariable)	-	Rückgabewert F_OPC_LOCAL (leerer Ausgangsname).

Programmablauf und Verbindungen

Der Rumpf ist in Structured Text geschrieben:

```
PACKAGE MyLib_OPC;

(* OPC-UA-ID fuer einen lokalen Knoten: opc_ua[ACTION;NODE] *)
FUNCTION F_OPC_LOCAL : WSTRING

VAR_INPUT
    sAction : WSTRING; // ACTION, z.B. READ oder CREATE_METHOD
    sNode : WSTRING; // Knoten, z.B. /Objects/X/Y,1:s=Y
END_VAR

F_OPC_LOCAL := CONCAT("opc_ua[", sAction, ";", sNode, "]");

END_FUNCTION
```

1. CONCAT setzt opc_ua[, sAction, ;, sNode und] zusammen.

Beispiel

Aufruf und Ergebnis:

```
MyLib_OPC::F_OPC_LOCAL("READ", "/Objects/X/Y,1:s=Y")  ->  opc_ua[READ;/Objects/X/
Y,1:s=Y]
```

Anwendung

Verwendet in myOpcUaAddresses.gcf (Projekt test_VV, Ordner const/OPC_UA, beschrieben in OPC_UA_Adressen.md) als InitialValue einer WSTRING-Konstante:

```
InitialValue="MyLib_OPC::F_OPC_LOCAL(&quot;READ&quot;, Q1_NODE)"
```

Q1_LOCAL_READ und FF1_TRIGGER_LOCAL_METHOD (lokal auf Station B) nutzen MyLib_OPC::F_OPC_LOCAL("READ", Q1_NODE) bzw. MyLib_OPC::F_OPC_LOCAL("CREATE_METHOD", FF1_TRIGGER_NODE).

Der Ausdruck wird beim Export ausgewertet; im Bootfile stehen daher nur die fertigen Strings, keine Funktionsaufrufe.

Hinweis: Die Argumente müssen Konstanten sein (Literale oder andere Konstanten der .gcf), da die Auswertung zur Exportzeit erfolgt und nicht zur Laufzeit.

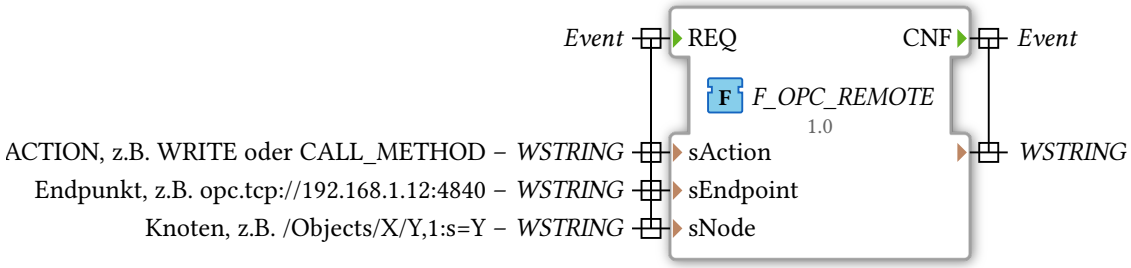
Zusammenfassung

`F_OPC_LOCAL` ist eine kleine IEC-61499-Funktion der Bibliothek `MyLib_OPC`, die die OPC-UA-ID für einen lokalen Knoten zusammensetzt.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.1.7.3 F_OPC_REMOTE: Funktion (IEC 61499 Function)



• – ◦ • – ◦ • – ◦ •

Einleitung

`F_OPC_REMOTE` ist keine SubApp, sondern ein Funktionstyp (`.fct`, Standard 61499-1) im Paket `MyLib_OPC` mit dem Kommentar „OPC-UA-ID fuer einen Knoten auf einer anderen Station: opc_ua[ACTION;ENDPOINT#;NODE]“. Er hat einen Ereigniseingang `REQ`, einen Ereignisausgang `CNF` und einen Strukturierter-Text-Rumpf; es gibt keine Funktionsbaustein-Instanzen und keine Verbindungen.

Schnittstelle der Funktion

Instanz	Typ	Parameter	Erklärung
`REQ`	`Event` (Ereigniseingang)	`With = sAction, sEndpoint, sNode`	Anforderung zur Ausführung.
`CNF`	`Event` (Ereignisausgang)	`With` = (unbenannte Variable)	Bestätigung; liefert den Rückgabewert.
`sAction`	`WSTRING` (Eingangsvariable)	-	ACTION, z.B. `WRITE` oder `CALL_METHOD`
`sEndpoint`	`WSTRING` (Eingangsvariable)	-	Endpunkt, z.B. `opc.tcp://192.168.1.12:4840`
`sNode`	`WSTRING` (Eingangsvariable)	-	Knoten, z.B. `/Objects/X/Y,1:s=Y`
(unbenannt)	`WSTRING` (Ausgangsvariable)	-	Rückgabewert `F_OPC_REMOTE` (leerer Ausgangsname).

Programmablauf und Verbindungen

Der Rumpf ist in Structured Text geschrieben:

```
PACKAGE MyLib_OPC;

(* OPC-UA-ID fuer einen Knoten auf einer anderen Station: opc_ua[ACTION;ENDPOINT#;NODE]
*)
FUNCTION F_OPC_REMOTE : WSTRING

VAR_INPUT
    sAction : WSTRING; // ACTION, z.B. WRITE oder CALL_METHOD
    sEndpoint : WSTRING; // Endpunkt, z.B. opc.tcp://192.168.1.12:4840
    sNode : WSTRING; // Knoten, z.B. /Objects/X/Y,1:s=Y
```

```
END_VAR

F_OPC_REMOTE := CONCAT("opc_ua[", sAction, ";", sEndpoint, "#;", sNode, "]");

END_FUNCTION
```

1. `CONCAT` setzt `opc_ua[`, `sAction`, `;`, `sEndpoint`, `#;`, `sNode` und `]` zusammen; das `#` schließt den Endpunkt ab.

Beispiel

Aufruf und Ergebnis:
`MyLib_OPC::F_OPC_REMOTE("WRITE", "opc.tcp://192.168.1.12:4840", "/Objects/X/Y,1:s=Y") -> opc_ua[WRITE;opc.tcp://192.168.1.12:4840#;/Objects/X/Y,1:s=Y]`

Anwendung

Verwendet in `myOpcUaAddresses.gcf` (Projekt `test_VV`, Ordner `const/OPC_UA`, beschrieben in `OPC_UA_Adressen.md`) als `InitialValue` einer WSTRING-Konstante:
`InitialValue="MyLib_OPC::F_OPC_REMOTE("WRITE", PC_B_OPC, Q1_NODE)"`
`Q1_REMOTE_WRITE` (Station A schreibt auf Station B) nutzt `MyLib_OPC::F_OPC_REMOTE("WRITE", PC_B_OPC, Q1_NODE)`; der Endpunkt `PC_B_OPC` stammt aus `F_MGR_TO_OPC`.

Der Ausdruck wird beim Export ausgewertet; im Bootfile stehen daher nur die fertigen Strings, keine Funktionsaufrufe.

Hinweis: Die Argumente müssen Konstanten sein (Literale oder andere Konstanten der `.gcf`), da die Auswertung zur Exportzeit erfolgt und nicht zur Laufzeit.

Zusammenfassung

`F_OPC_REMOTE` ist eine kleine IEC-61499-Funktion der Bibliothek `MyLib_OPC`, die die OPC-UA-ID für einen Knoten auf einer anderen Station zusammensetzt.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.1.8 OSCAT

2.1.8.1 Basic

Data_types

CONSTANTS_MATH

CONSTANTS_MATH

Structure

Name	Type	Initial Value	Comment
PI	REAL	3.14159265358979323846264338327950288	Kreiszahl PI
PI2	REAL	6.28318530717958647692528676655900576	PI * 2
PI4	REAL	12.56637061435917295385057353311801152	PI * 4
PI05	REAL	1.5707963267949	PI / 2
PI025	REAL	0.785398163397448	PI / 4
PI_INV	REAL	0.318309886183791	1 / PI
EUL	REAL	2.71828182845904523536028747135266249	Euler constant e
E_INV	REAL	0.367879441171442	1 / e
SQ2	REAL	1.4142135623731	Wurzel von 2
FACTS	DINT	[1,1,2,6,24,120,720,5040,40320,362880,3628800,39916800,479001600]	Factorials Array 0..12

Globals

Constants
Global constants

Global Constants

Name	Type	Initial Value
MATH	OSCAT::Basic::Data_types::CONSTANTS_MATH	

POUs

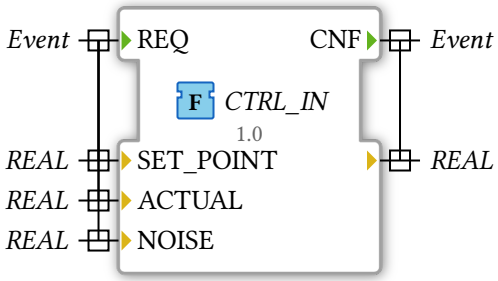
Engineering

Control

🎧 Podcast
- Eclipse 4diac: Revolutionizing Industrial Control with Open-Source Cyber-Physical Systems
- Simplifying Industrial Control: Your Deep Dive into Eclipse 4diac and IEC 61499
- The Future of Industrial Control: Decoding IEC 61499
- Decoding Industrial Control: Function Blocks, Object-Oriented Principles, and the Power of IEC 61499
- IEC 61499 vs. 61131: Unpacking the Architectural Revolution of Distributed Control and Industry 4.0

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

CTRL_IN



calculate the process error DIFF

Interface

Event Inputs

Name	Comment	With
REQ		SET_POINT, ACTUAL, NOISE

Event Outputs

Name	Comment	With
CNF		

Input Vars

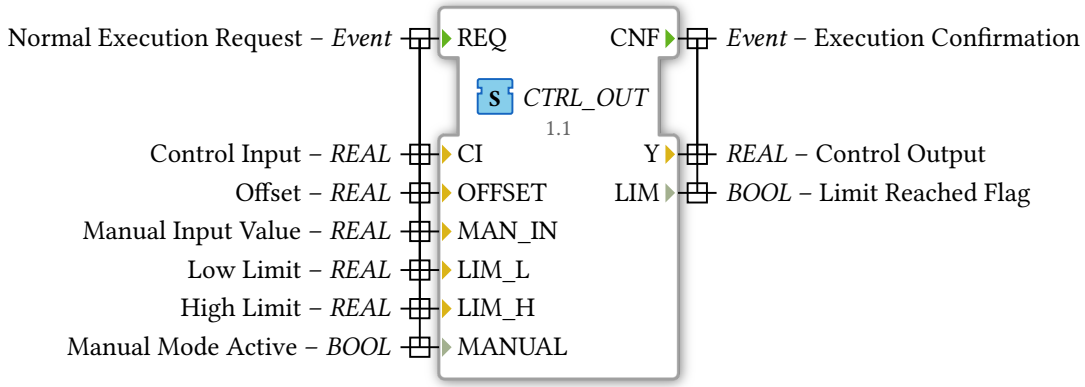
Name	Type	Comment
SET_POINT	REAL	

ACTUAL	REAL	
NOISE	REAL	

Output Vars

Name	Type	Comment
	REAL	

CTRL_OUT



Control Output with Limit and Manual Override

CTRL_OUT adds to the CI input the value of OFFSET and returns the result to Y if MANUAL = FALSE. If MANUAL is TRUE at output Y the input value of MAN_IN + OFFSET is issued. Y is always limited to the boundaries defined by LIM_L and LIM_H. If Y reaches one of the limits, then the output LIM is TRUE. CTRL_OUT can be used to build own rule modules.

Interface

Event Inputs

Name	Comment	With
REQ	Normal Execution Request	CI, OFFSET, MAN_IN, LIM_L, LIM_H, MANUAL

Event Outputs

Name	Comment	With
CNF	Execution Confirmation	Y, LIM

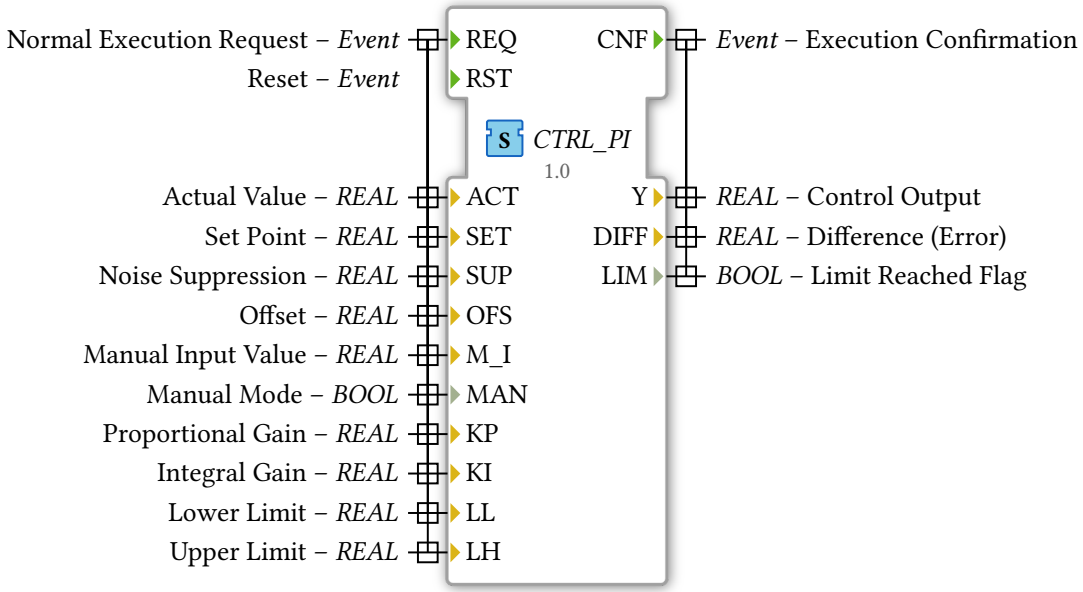
Input Vars

Name	Type	Comment
CI	REAL	Control Input
OFFSET	REAL	Offset
MAN_IN	REAL	Manual Input Value
LIM_L	REAL	Low Limit
LIM_H	REAL	High Limit
MANUAL	BOOL	Manual Mode Active

Output Vars

Name	Type	Comment
Y	REAL	Control Output
LIM	BOOL	Limit Reached Flag

CTRL_PI



FT_PI is a PI controller with manual functionality.

version 2.0 30. jun 2008 programmer hugo tested by oscat

FT_PI is a PI controller with manual functionality. The PID controller works according to the fomula Y = e *(KP+ KI * INTEG(e)) + offset, while e = set_point - actual. a rst will reset all internal data, while a switch to manual will cause the controller to follow the function Y = manual_in + offset. limit_h and Limit_l set the possible output range of Y. the output flags lim will signal that the output limits are active and overflow will signal that the integrator has reached its limits.

since rev 1.1 the “trapezregel is used for more accuracy. rev 1.2 added selective integratin which means the integrative component is only active within a small range of the target value this avoids the integrator to go to limits while an input setpoint change happened and is only causing overshoots. the int_band is by default 100 which means the int is active all the time and if set to for example to 0.1 the integrator is only active while the input is between 0.9 and 1.1 of the set_point value.

default values for KP = 1, TN = 1, TV = 1, LIMIT_L = −1000, LIMIT_H = +1000.

Interface

Event Inputs

Name	Comment	With
REQ	Normal Execution Request	ACT, SET, SUP, OFS, M_I, MAN, KP, KI, LL, LH
RST	Reset	

Event Outputs

Name	Comment	With
CNF	Execution Confirmation	Y, DIFF, LIM

Input Vars

Name	Type	Initial Value	Comment
ACT	REAL		Actual Value
SET	REAL		Set Point
SUP	REAL		Noise Suppression
OFS	REAL		Offset
M_I	REAL		Manual Input Value
MAN	BOOL		Manual Mode
KP	REAL	1.0	Proportional Gain
KI	REAL	1.0	Integral Gain
LL	REAL	−1000.0	Lower Limit
LH	REAL	1000.0	Upper Limit

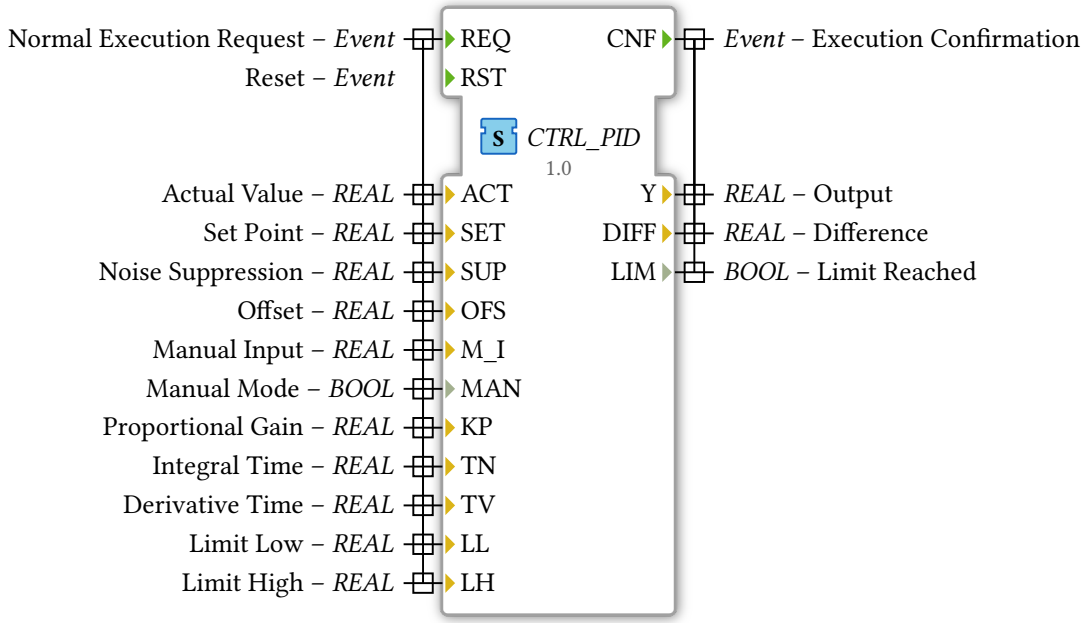
Output Vars

Name	Type	Comment
Y	REAL	Control Output
DIFF	REAL	Difference (Error)
LIM	BOOL	Limit Reached Flag

🛠️ Zugehörige Übungen

- Uebung_152

CTRL_PID



FT_PI is a PI controller with manual functionality.
version 2.0 30. jun. 2008 programmer hugo tested by oscat
FT_PI is a PI controller with manual functionality. The PID controller works according to the fomula Y = e *(KP+ KI * INTEG(e)) + offset, while e = set_point - actual. a rst will reset all internal data, while

a switch to manual will cause the controller to follow the function Y = manual_in + offset. limit_h and Limit_l set the possible output range of Y. the output flags lim will signal that the output limits are active.

Interface

Event Inputs

Name	Comment	With
REQ	Normal Execution Request	ACT, SET, SUP, OFS, M_I, MAN, KP, TN, TV, LL, LH
RST	Reset	

Event Outputs

Name	Comment	With
CNF	Execution Confirmation	Y, DIFF, LIM

Input Vars

Name	Type	Initial Value	Comment
ACT	REAL		Actual Value
SET	REAL		Set Point
SUP	REAL		Noise Suppression
OFS	REAL		Offset
M_I	REAL		Manual Input
MAN	BOOL		Manual Mode
KP	REAL	1.0	Proportional Gain
TN	REAL	1.0	Integral Time
TV	REAL	1.0	Derivative Time
LL	REAL	−1000.0	Limit Low
LH	REAL	1000.0	Limit High

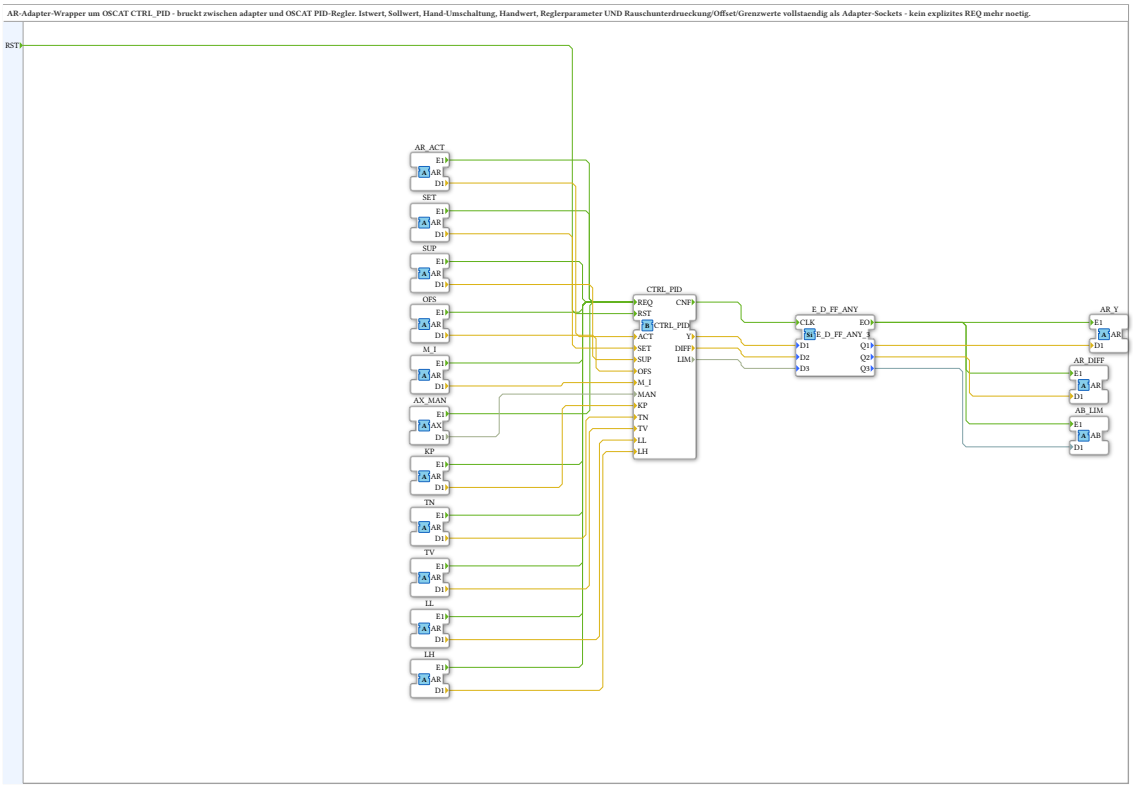
Output Vars

Name	Type	Comment
Y	REAL	Output
DIFF	REAL	Difference
LIM	BOOL	Limit Reached

🛠️ Zugehörige Übungen

- Uebung_153

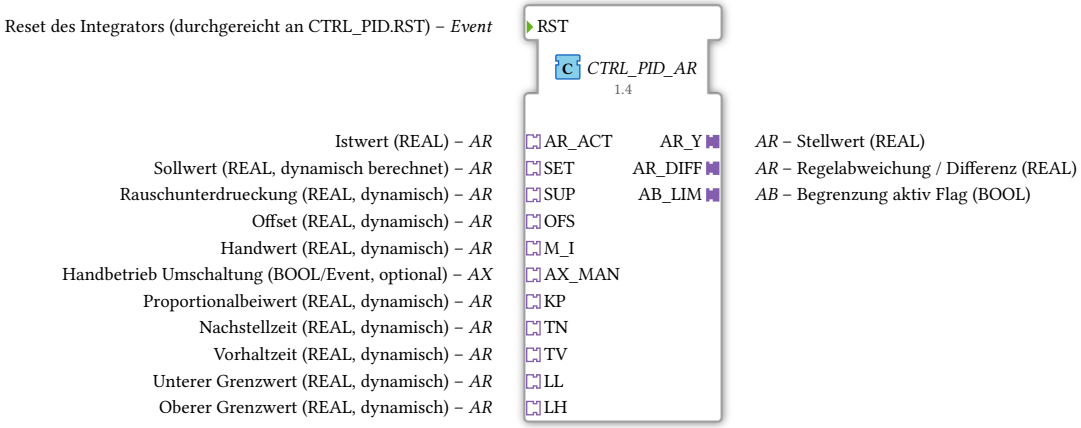
CTRL_PID_AR



• – ○ • – ○ • – ○ •

Einleitung

`CTRL_PID_AR` ist ein AR-Adapter-Wrapper um den OSCAT-PID-Regler `CTRL_PID` (`OSCAT::Basic::POUs::Engineering::Control::CTRL_PID`) und überbrückt zwischen Adaptern und dem OSCAT-Regler. Istwert, Sollwert, Handumschaltung, Handwert, Reglerparameter sowie Rauschunterdrückung, Offset und Grenzwerte sind vollständig als Adapter-Sockets ausgeführt – in derselben Reihenfolge wie die Eingangsvariablen des umschlossenen `CTRL_PID`. Ein explizites `REQ` ist nicht nötig, da jeder Socket bei Ankunft eines neuen Werts selbst `CTRL_PID.REQ` auslöst.



Schnittstellenstruktur

Ereignis-Eingänge

- **RST**: Reset des Integrators (durchgereicht an `CTRL_PID.RST`)

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden – Ereignisse laufen über die Adapter-Plugs

Daten-Eingänge

Keine Daten-Eingänge vorhanden – alle Eingänge sind Adapter-Sockets

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden – alle Ausgänge sind Adapter-Plugs

Adapter

Sockets (Eingänge), jeweils `E1` löst `CTRL_PID.REQ` aus, `D1` liefert den Wert:

- **AR_ACT** (`AR`): Istwert (REAL)
- **SET** (`AR`): Sollwert (REAL, dynamisch berechnet)
- **SUP** (`AR`): Rauschunterdrückung (REAL, dynamisch)
- **OFS** (`AR`): Offset (REAL, dynamisch)
- **M_I** (`AR`): Handwert (REAL, dynamisch)
- **AX_MAN** (`AX`): Handbetrieb-Umschaltung (BOOL/Event, optional)
- **KP** (`AR`): Proportionalbeiwert (REAL, dynamisch)
- **TN** (`AR`): Nachstellzeit (REAL, dynamisch)
- **TV** (`AR`): Vorhaltzeit (REAL, dynamisch)
- **LL** (`AR`): Unterer Grenzwert (REAL, dynamisch)
- **LH** (`AR`): Oberer Grenzwert (REAL, dynamisch)

Plugs (Ausgänge):

- **AR_Y** (`AR`): Stellwert (REAL)
- **AR_DIFF** (`AR`): Regelabweichung (`DIFF = SET - ACT`, REAL)
- **AB_LIM** (`AB`): Begrenzung-aktiv-Flag (`LIM`, BOOL)

Verwendete Funktionsbausteine (FBs)

- **CTRL_PID**: `OSCAT::Basic::POUs::Engineering::Control::CTRL_PID`
- **E_D_FF_ANY**: `iec61499::events::E_D_FF_ANY_3` – 3-fach-Datenlatch für `Y`, `DIFF`, `LIM`

Programmablauf und Verbindungen

1. Jedes `E1` der Sockets `AR_ACT`, `SET`, `SUP`, `OFS`, `M_I`, `AX_MAN`, `KP`, `TN`, `TV`, `LL`, `LH` ⟶ `CTRL_PID.REQ`.
2. `D1` der Sockets ⟶ gleichnamiger Eingang von `CTRL_PID` (`AR_ACT.D1` ⟶ `ACT`, `AX_MAN.D1` ⟶ `MAN`, sonst gleiche Namen).
3. `RST` ⟶ `CTRL_PID.RST`.
4. `CTRL_PID.CNF` ⟶ `E_D_FF_ANY.CLK`; `CTRL_PID.Y/DIFF/LIM` ⟶ `E_D_FF_ANY.D1/D2/D3`.
5. `E_D_FF_ANY.EO` ⟶ `AR_Y.E1`, `AR_DIFF.E1`, `AB_LIM.E1`; `Q1/Q2/Q3` ⟶ `AR_Y.D1`, `AR_DIFF.D1`, `AB_LIM.D1`.

Technische Besonderheiten

- Der Latch (`E_D_FF_ANY`) löst die Ausgänge erst nach dem `CNF` des Reglers aus; es gibt kein externes `CNF`-Ereignis mehr.
- Jeder Eingangs-Socket löst eine Berechnung aus; bei mehreren gleichzeitig geänderten Werten erfolgt daher mehrfaches `REQ`.
- Die Sockets sind in der Reihenfolge der `InputVars` von `CTRL_PID` sortiert (Version 1.4).

Anwendungsszenarien

- PID-Regelung in Adapter-Netzwerken, z. B. Geschwindigkeitsregelung der Aufnahme mit dynamisch berechnetem Sollwert und `AX_MAN` aus einem Automatik/Hand-Umschalter

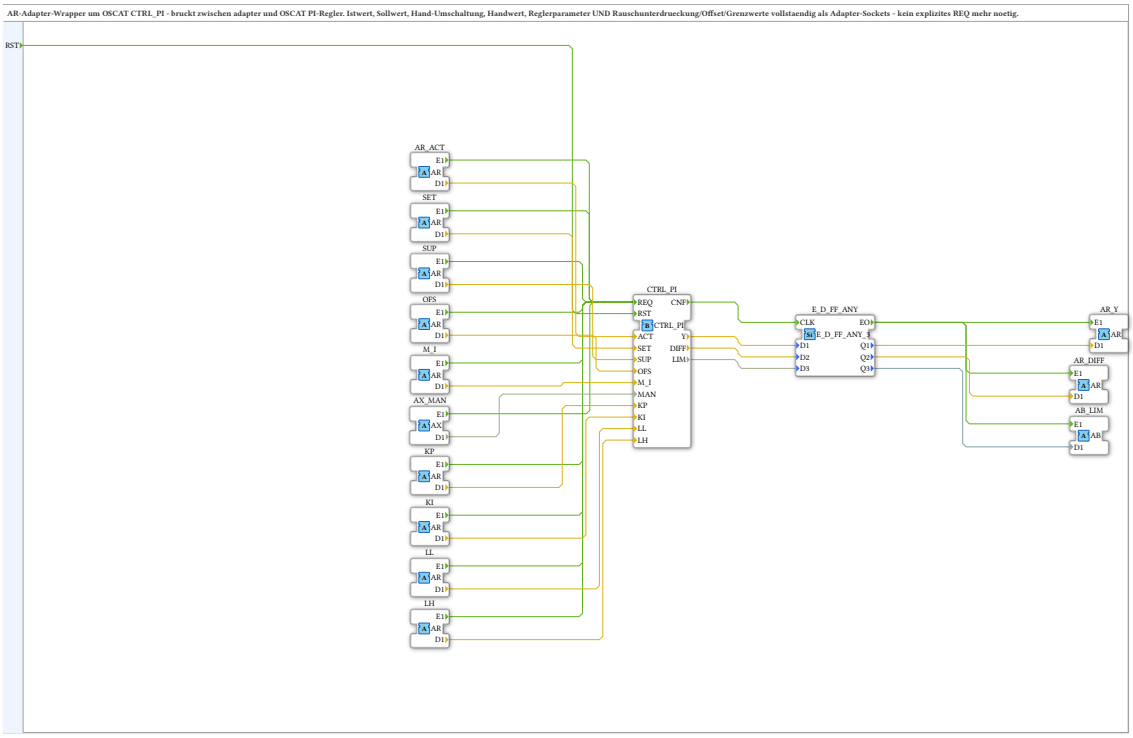
🛠️ Zugehörige Übungen

- Uebung_153_AX

Fazit

`CTRL_PID_AR` bindet den OSCAT-PID-Regler vollständig über AR-/AX-Adapter an, sodass Werte nur noch über Adapter-Verbindungen fließen und kein explizites `REQ` nötig ist.

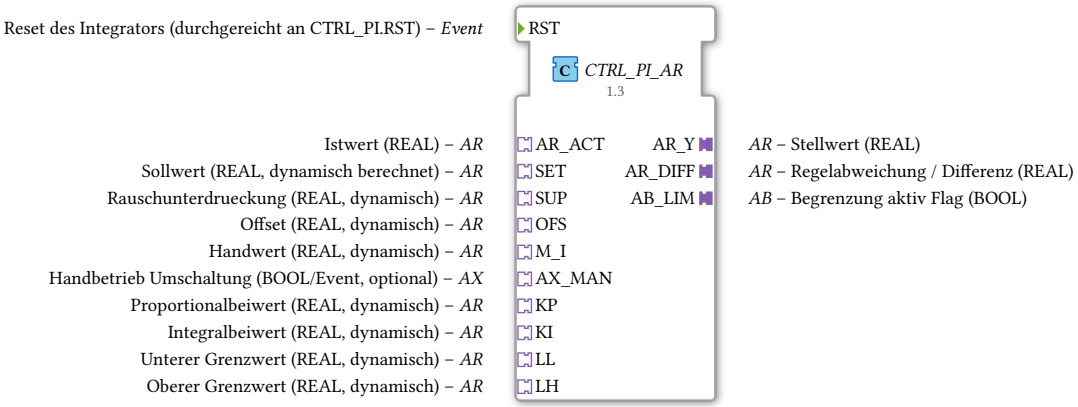
CTRL_PI_AR



• – ○ • – ○ • – ○ •

Einleitung

`CTRL_PI_AR` ist ein AR-Adapter-Wrapper um den OSCAT-PI-Regler `OSCAT::Basic::POUs::Engineering::Control::CTRL_PI`. Er überbrückt zwischen Adaptern und OSCAT, ohne dass eine der beiden Basisbibliotheken von der anderen abhängt. Istwert, Sollwert, Rauschunterdrückung, Offset, Handwert, Handbetrieb-Umschaltung, Reglerparameter und Grenzwerte sind vollständig als Adapter-Sockets ausgeführt; ein explizites `REQ` ist nicht mehr nötig, weil jeder Socket den Regler selbst auslöst.



Schnittstellenstruktur

Ereignis-Eingänge

- **RST**: Reset des Integrators (durchgereicht an `CTRL_PI.RST`)

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **AR_ACT** (Socket, AR): Istwert (REAL)
- **SET** (Socket, AR): Sollwert (REAL, dynamisch berechnet)
- **SUP** (Socket, AR): Rauschunterdrückung (REAL, dynamisch)
- **OFS** (Socket, AR): Offset (REAL, dynamisch)
- **M_I** (Socket, AR): Handwert (REAL, dynamisch)
- **AX_MAN** (Socket, AX): Umschaltung auf Handbetrieb (optional)
- **KP** (Socket, AR): Proportionalbeiwert (REAL, dynamisch)
- **KI** (Socket, AR): Integralbeiwert (REAL, dynamisch)
- **LL** (Socket, AR): unterer Grenzwert (REAL, dynamisch)
- **LH** (Socket, AR): oberer Grenzwert (REAL, dynamisch)
- **AR_Y** (Plug, AR): Stellwert (REAL)
- **AR_DIFF** (Plug, AR): Regelabweichung `DIFF = SET - ACT` (REAL)
- **AB_LIM** (Plug, AB): Flag ‚Begrenzung aktiv' (BOOL)

Funktionsweise

Das Netzwerk besteht aus dem Regler `CTRL_PI` (`OSCAT::Basic::POUs::Engineering::Control::CTRL_PI`) und einem Änderungsfilter `E_D_FF_ANY` (`iec61499::events::E_D_FF_ANY_3`):

- **Auslösen**: Das Ereignis `E1` jedes Sockets (`AR_ACT`, `SET`, `SUP`, `OFS`, `M_I`, `AX_MAN`, `KP`, `KI`, `LL`, `LH`) führt auf `CTRL_PI.REQ`. `RST` führt auf `CTRL_PI.RST`.
- **Daten**: `AR_ACT.D1` ⟶ `ACT`, `SET.D1` ⟶ `SET`, `SUP.D1` ⟶ `SUP`, `OFS.D1` ⟶ `OFS`, `M_I.D1` ⟶ `M_I`, `AX_MAN.D1` ⟶ `MAN`, `KP.D1` ⟶ `KP`, `KI.D1` ⟶ `KI`, `LL.D1` ⟶ `LL`, `LH.D1` ⟶ `LH` des Reglers.
- **Ausgabe**: `CTRL_PI.CNF` ⟶ `E_D_FF_ANY.CLK`; `CTRL_PI.Y`, `CTRL_PI.DIFF` und `CTRL_PI.LIM` gehen auf `E_D_FF_ANY.D1`..`D3`. Nur wenn sich mindestens einer der drei Werte ändert, gibt `E_D_FF_ANY.EO` das Ereignis auf `AR_Y.E1`, `AR_DIFF.E1` und `AB_LIM.E1`, und `Q1`..`Q3` gehen auf `AR_Y.D1`, `AR_DIFF.D1` und `AB_LIM.D1`.

Technische Besonderheiten

- Das Ereignis `CNF` des Reglers ist nur noch intern (für das Latch von `Y`/`DIFF`/`LIM`); ein externes `REQ`/`CNF` gibt es nicht mehr
- Jeder Socket löst den Regler aus, sodass bei mehreren gleichzeitig eintreffenden Werten mehrere Regler-Durchläufe möglich sind
- Die Sockets sind in der Reihenfolge der InputVars von `CTRL_PI` sortiert: `AR_ACT`, `SET`, `SUP`, `OFS`, `M_I`, `AX_MAN`, `KP`, `KI`, `LL`, `LH`
- Die Adapter-Brücke liegt in der Bibliothek `OSCAT_adapter`, die von `adapter` und `OSCAT` abhängt; so müssen diese beiden nicht voneinander abhängen

Anwendungsszenarien

- PI-Regelung in rein adapterbasierten Netzwerken, z. B. Geschwindigkeits- oder Drehzahlregelung mit dynamisch berechnetem Sollwert
- Regler, deren Parameter (KP, KI) und Grenzen zur Laufzeit per VT/OPC-UA veränderbar sind

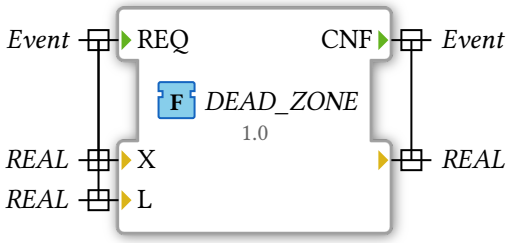
🛠️ Zugehörige Übungen

- Uebung_152_AX

Fazit

`CTRL_PI_AR` macht den OSCAT-PI-Regler adapterfähig: Alle Ein- und Ausgänge sind AR-/AX-/AB-Adapter, jeder Eingang löst die Berechnung aus, und Ergebnisse werden nur bei Änderung ausgegeben.

DEAD_ZONE



dead_zone2 is a linear transfer function which follows a linear function except for x is close to 0.

Interface

Event Inputs

Name	Comment	With
REQ		X, L

Event Outputs

Name	Comment	With
CNF		

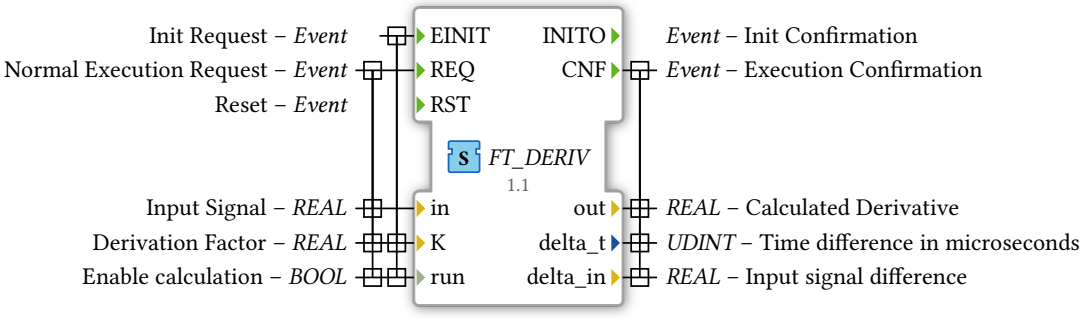
Input Vars

Name	Type	Comment
X	REAL	
L	REAL	

Output Vars

Name	Type	Comment
	REAL	

FT_DERIV



Calculate derivative over signal ‚in' with Factor ‚K'

Interface

Event Inputs

Name	Comment	With
EINIT	Init Request	K, run

REQ	Normal Execution Request	in, K, run
RST	Reset	

Event Outputs

Name	Comment	With
INITO	Init Confirmation	
CNF	Execution Confirmation	out, delta_t, delta_in

Input Vars

Name	Type	Initial Value	Comment
in	REAL		Input Signal
K	REAL	1.0	Derivation Factor
run	BOOL	1	Enable calculation

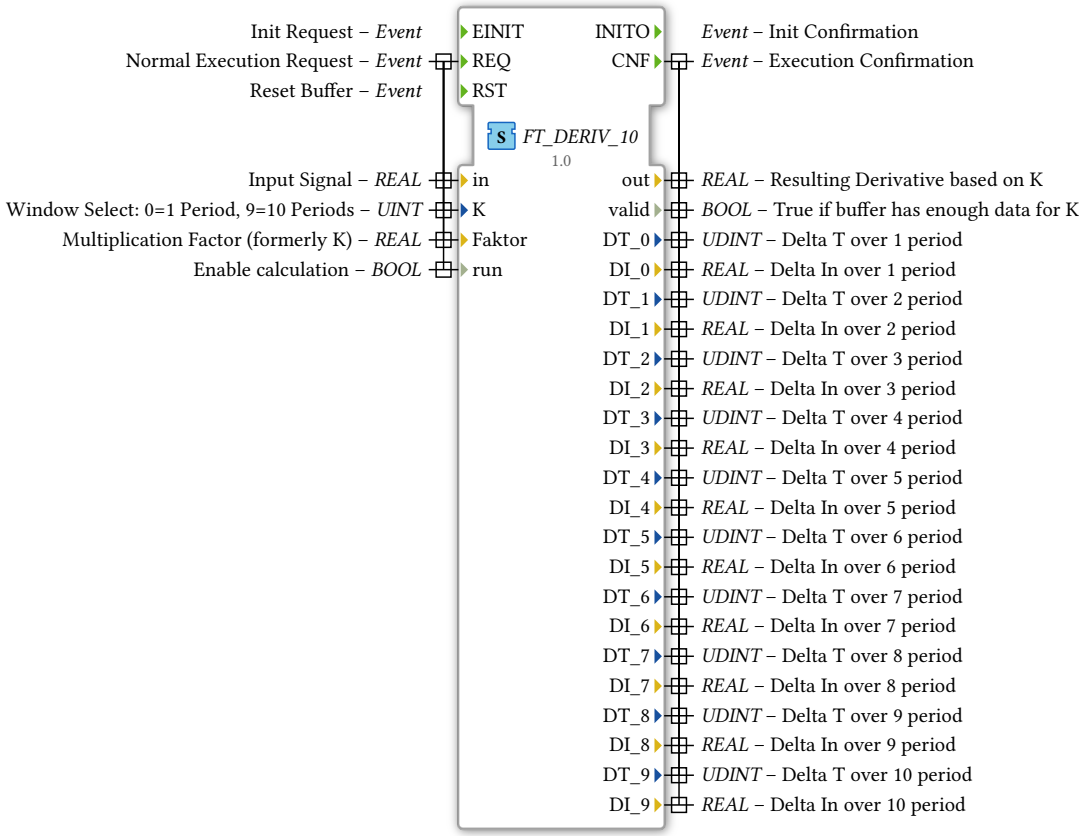
Output Vars

Name	Type	Comment
out	REAL	Calculated Derivative
delta_t	UDINT	Time difference in microseconds
delta_in	REAL	Input signal difference

🛠️ Zugehörige Übungen

- Uebung_151
- Uebung_151_AX

FT_DERIV_10



Derivative with variable window size (0..9 periods)

Interface

Event Inputs

Name	Comment	With
EINIT	Init Request	
REQ	Normal Execution Request	in, K, Faktor, run
RST	Reset Buffer	

Event Outputs

Name	Comment	With
INITO	Init Confirmation	
CNF	Execution Confirmation	out, valid, DT_0, DI_0, DT_1, DI_1, DT_2, DI_2, DT_3, DI_3, DT_4, DI_4, DT_5, DI_5, DT_6, DI_6, DT_7, DI_7, DT_8, DI_8, DT_9, DI_9

Input Vars

Name	Type	Initial Value	Comment
in	REAL		Input Signal
K	UINT	0	Window Select: 0=1 Period, 9=10 Periods
Faktor	REAL	1.0	Multiplication Factor (formerly K)

run	BOOL	1	Enable calculation

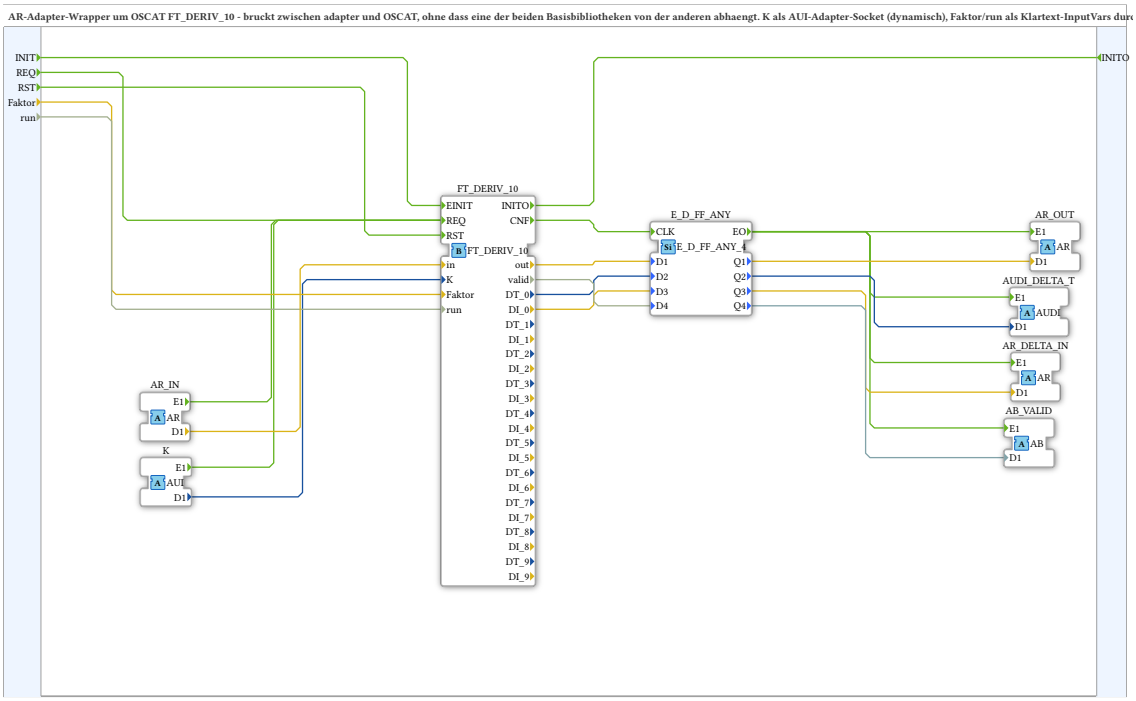
Output Vars

Name	Type	Comment
out	REAL	Resulting Derivative based on K
valid	BOOL	True if buffer has enough data for K
DT_0	UDINT	Delta T over 1 period
DI_0	REAL	Delta In over 1 period
DT_1	UDINT	Delta T over 2 period
DI_1	REAL	Delta In over 2 period
DT_2	UDINT	Delta T over 3 period
DI_2	REAL	Delta In over 3 period
DT_3	UDINT	Delta T over 4 period
DI_3	REAL	Delta In over 4 period
DT_4	UDINT	Delta T over 5 period
DI_4	REAL	Delta In over 5 period
DT_5	UDINT	Delta T over 6 period
DI_5	REAL	Delta In over 6 period
DT_6	UDINT	Delta T over 7 period
DI_6	REAL	Delta In over 7 period
DT_7	UDINT	Delta T over 8 period
DI_7	REAL	Delta In over 8 period
DT_8	UDINT	Delta T over 9 period
DI_8	REAL	Delta In over 9 period
DT_9	UDINT	Delta T over 10 period
DI_9	REAL	Delta In over 10 period

🛠️ Zugehörige Übungen

- Uebung_152
- Uebung_153

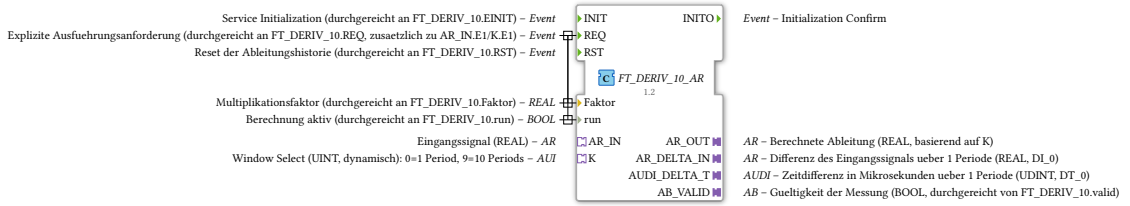
FT_DERIV_10_AR



• – ○ • – ○ • – ○ •

Einleitung

`FT_DERIV_10_AR` ist ein AR-Adapter-Wrapper um den OSCAT-Baustein `OSCAT::Basic::POUs::Engineering::Control::FT_DERIV_10` (Ableitung mit wählbarem Fenster von 1 bis 10 Perioden). Er überbrückt zwischen Adaptern und OSCAT, ohne dass eine der beiden Basisbibliotheken von der anderen abhängt. Gegenüber `FT_DERIV_AR` ist das Fenster `K` ein dynamischer `AUI`-Adapter-Socket. `Faktor` und `run` bleiben einfache Eingangsvariablen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Service-Initialisierung (durchgereicht an `FT_DERIV_10.EINIT`)
- **REQ**: explizite Ausführungsanforderung (durchgereicht an `FT_DERIV_10.REQ`, zusätzlich zu `AR_IN.E1`/`K.E1`)
  - **Verbundene Daten**: `Faktor`, `run`
- **RST**: Reset der Ableitungshistorie (durchgereicht an `FT_DERIV_10.RST`)

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung (von `FT_DERIV_10.INITO`)

Daten-Eingänge

- **Faktor** (`REAL`, Initialwert `1.0`): Multiplikationsfaktor (durchgereicht an `FT_DERIV_10.Faktor`)
- **run** (`BOOL`, Initialwert `TRUE`): Berechnung aktiv (durchgereicht an `FT_DERIV_10.run`)

Daten-Ausgänge

Keine. Alle Ergebniswerte werden über Adapter-Plugs ausgegeben.

Adapter

Schnittstelle	Richtung	Adaptertyp	Datentyp	Beschreibung
AR_IN	Socket	AR	REAL	Eingangssignal
K	Socket	AUI	UINT	Window Select (dynamisch): 0 = 1 Periode, 9 = 10 Perioden
AR_OUT	Plug	AR	REAL	Berechnete Ableitung (FT_DERIV_10.out, basierend auf K)
AR_DELTA_IN	Plug	AR	REAL	Differenz des Eingangssignals über 1 Periode (DI_0)
AUDI_DELTA_T	Plug	AUDI	UDINT	Zeitdifferenz in Mikrosekunden über 1 Periode (DT_0)
AB_VALID	Plug	AB	BOOL	Gültigkeit der Messung (FT_DERIV_10.valid)

Verwendete Funktionsbausteine (FBs)

- **FT_DERIV_10** (OSCAT::Basic::POUs::Engineering::Control::FT_DERIV_10): Ableitungsberechnung, siehe FT_DERIV_10
- **E_D_FF_ANY** (iec61499::events::E_D_FF_ANY_4): vierkanaliger Änderungsfilter/Synchronisation der Ergebnisse

Funktionsweise

- AR_IN.E1, K.E1 und REQ führen alle auf FT_DERIV_10.REQ
- AR_IN.D1 ⟶ FT_DERIV_10.in; K.D1 ⟶ FT_DERIV_10.K; Faktor ⟶ FT_DERIV_10.Faktor; run ⟶ FT_DERIV_10.run
- INIT ⟶ FT_DERIV_10.EINIT; FT_DERIV_10.INITO ⟶ INITO; RST ⟶ FT_DERIV_10.RST
- FT_DERIV_10.CNF ⟶ E_D_FF_ANY.CLK; die Ergebnisse out, DT_0, DI_0, valid gehen an E_D_FF_ANY.D1 bis D4
- E_D_FF_ANY.Q1…Q4 ⟶ AR_OUT.D1, AUDI_DELTA_T.D1, AR_DELTA_IN.D1, AB_VALID.D1; E_D_FF_ANY.EO löst E1 aller vier Plugs aus

Technische Besonderheiten

- **Reine Adapter-Grenze**: Alle Hauptausgänge von FT_DERIV_10 werden als eigene Adapter ausgegeben, kein Klartext-Ereignis/-Datum verlässt den Baustein (außer INITO)
- **K als Adapter** (Version 1.2, 2026-10-03): K.E1 löst wie AR_IN.E1 eine Berechnung aus, K.D1 liefert den Wert; K ist deshalb nicht mehr per With an REQ gebunden
- **Explizites REQ** (Version 1.1, 2026-09-20): Bugfix, damit Faktor und run per With-Verknüpfung zuverlässig gelesen werden (nach IEC 61499 werden ungebundene Variablen nicht zuverlässig gelesen); erlaubt außerdem manuellen oder periodischen Aufruf
- FT_DERIV_10 ist eine reine OSCAT-Funktion ohne Adapter-Schnittstelle; die Brücke liegt im dritten Bibliotheksprojekt OSCAT_adapter

Anwendungsszenarien

- Frequenz-/Drehzahlberechnung aus einem Impulszählerwert (AR_IN, z. B. von AD_TO_AR_NUM.AR_OUT) mit zur Laufzeit wählbarem Fenster
- Weitergabe der Ableitung z. B. an CTRL_PID_AR.AR_ACT

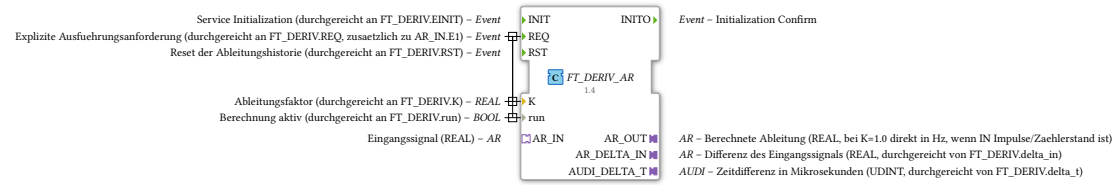
Zusammenfassung

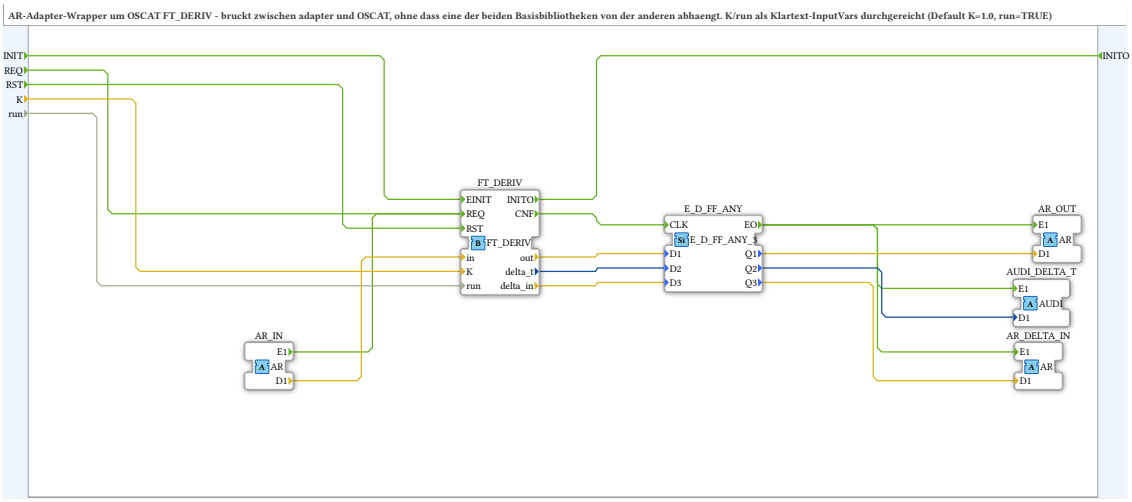
FT_DERIV_10_AR kapselt FT_DERIV_10 hinter einer reinen Adapter-Schnittstelle mit dynamisch wählbarem Fenster K und stellt Ableitung, Eingangsdifferenz, Zeitdifferenz und Gültigkeit als Adapter bereit.

🛠️ Zugehörige Übungen

- Uebung_152_AX
- Uebung_153_AX

FT_DERIV_AR





• – ○ • – ○ • – ○ •

Einleitung

`FT_DERIV_AR` ist ein Adapter-Wrapper um den OSCAT-Baustein `FT_DERIV`.
Er kapselt die Berechnung der zeitlichen Ableitung eines Eingangssignals hinter einer rein adapterbasierten Schnittstelle. Die Werte `K` (Ableitungsfaktor) und `run` (Berechnung aktiv) werden als gewöhnliche Input-Variablen durchgereicht und sind nicht Teil des Adapter-Datenflusses.

Der Baustein besitzt keine Klartext-Datenausgänge. Stattdessen werden alle von `FT_DERIV` erzeugten Werte über eigene Adapter-Plugs bereitgestellt:

- `AR_OUT` – die berechnete Ableitung
- `AR_DELTA_IN` – die Differenz des Eingangssignals
- `AUDI_DELTA_T` – die Zeitdifferenz in Mikrosekunden

Damit kann der Baustein in übergeordneten SubApp-Netzwerken über normale `AdapterConnections` mit anderen AR-/AUDI-Bausteinen verbunden werden, ohne dass auf interne `.E1`-/`.D1`-Signale fremdinstanziierter Bausteine zugegriffen werden muss.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung
`INIT`	`EInit`	Service-Initialisierung, wird an `FT_DERIV.EINIT` durchgereicht
`RST`	`Event`	Setzt die Ableitungshistorie von `FT_DERIV` zurück

Ereignis-Ausgänge

Name	Typ	Beschreibung
`INITO`	`EInit`	Initialisierungsbestätigung von `FT_DERIV`

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung

K	REAL	1.0	Ableitungsfaktor, wird an FT_DERIV.K durchgereicht
run	BOOL	TRUE	Schaltet die Berechnung aktiv, wird an FT_DERIV.run durchgereicht

Daten-Ausgänge

Keine. Alle Ergebniswerte werden über die Adapter-Plugs ausgegeben.

Adapter

Schnittstelle	Richtung	Adaptertyp	Datentyp	Beschreibung
AR_IN	Socket	AR	REAL	Eingangssignal, dessen Ableitung berechnet werden soll
AR_OUT	Plug	AR	REAL	Berechnete Ableitung (FT_DERIV.out)
AR_DELTA_IN	Plug	AR	REAL	Differenz des Eingangssignals (FT_DERIV.delta_in)
AUDI_DELTA_T	Plug	AUDI	UDINT	Zeitdifferenz des letzten Berechnungsschritts in Mikrosekunden (FT_DERIV.delta_t)

Funktionsweise

Der Baustein ist intern als FBNetzwerk aufgebaut und verwendet zwei zentrale Komponenten:

1. FT_DERIV aus der OSCAT-Bibliothek für die eigentliche Ableitungsberechnung.
2. E_D_FF_ANY_3, ein dreikanaliges D-Flipflop, das die berechneten Werte synchron übernimmt und an die Adapterausgänge weitergibt.

Ablauf:

- Ein Ereignis auf AR_IN.E1 löst FT_DERIV.REQ aus.
- Gleichzeitig wird AR_IN.D1 als Eingangssignal an FT_DERIV.in übergeben.
- K und run werden direkt als Parameter an FT_DERIV durchgereicht.
- Nach Abschluss der Berechnung signalisiert FT_DERIV.CNF das Ereignis CLK an das interne Flipflop.
- Das Flipflop übernimmt die Werte out, delta_t und delta_in und gibt sie über Q1, Q2 und Q3 aus.
- Mit dem Folgeereignis EO werden alle drei Adapter-Plugs AR_OUT, AR_DELTA_IN und AUDI_DELTA_T parallel bedient.

Der eigentliche Differenzierer arbeitet mit der Differenz des Eingangssignals und der verstrichenen Zeit. Der Faktor K skaliert das Ergebnis. Bei K = 1.0 liefert der Baustein die Ableitung direkt, beispielsweise in Hz, wenn das Eingangssignal ein Impuls- oder Zählerstand ist.

INIT und RST werden unverändert an die entsprechenden Eingänge von FT_DERIV durchgereicht:

- INIT initialisiert den Baustein.
- RST setzt die interne Ableitungshistorie zurück.

Technische Besonderheiten

- **Reine Adapter-Grenze:**
  Kein Klartext-Event und kein Klartext-Datum verlässt den Baustein. Alle Ergebnisse werden ausschließlich über Adapter-Plugs bereitgestellt.
- **Keine Abhängigkeit zwischen adapter und OSCAT:**
  Der Wrapper liegt in einem eigenen Brückenprojekt und kapselt die Verbindung zwischen der generischen Adapter-Bibliothek und der OSCAT-Bibliothek.
- **Durchreichung von K und run:**
  Beide Werte sind pro Berechnungsaufruf konstant. Deshalb werden sie als normale InputVars geführt und können bei der Instanziierung über Parameter gesetzt werden.
- **Synchronisation über E_D_FF_ANY_3:**
  Das interne Flipflop stellt sicher, dass die Datenausgänge erst nach Abschluss der Berechnung aktualisiert werden und ein gemeinsames Ausgangsereignis erzeugt wird.

- **Initialisierung:**
  Wenn `INIT` nicht verdrahtet ist, feuert das `EInit`-Ereignis beim Deployment automatisch einmalig. Damit wird `FT_DERIV` auch ohne explizite Initialisierung korrekt gestartet.

Zustandsübersicht

Da `FT_DERIV_AR` ein zusammengesetzter Funktionsbaustein ist, besitzt er keinen eigenen internen Zustandsautomaten. Die Zustandsübersicht ergibt sich aus den Ereignispfaden:

Zustand / Phase	Auslöser	Aktion
Initialisierung	`INIT`	`FT_DERIV.EINIT` wird ausgeführt, Bestätigung über `INITO`
Warten auf Messwert	`AR_IN.E1`	Eingangsdaten werden eingelesen, `FT_DERIV.REQ` wird ausgelöst
Berechnung	`FT_DERIV.CNF`	Ableitung, Zeitdifferenz und Eingangsdifferenz werden übernommen
Ausgabe	`EO` des Flipflops	`AR_OUT`, `AR_DELTA_IN` und `AUDI_DELTA_T` werden aktualisiert
Reset	`RST`	Ableitungshistorie von `FT_DERIV` wird zurückgesetzt

Anwendungsszenarien

- **Frequenz- und Drehzahlmessung:**
  Zählerstände oder Impulse werden über `AR_IN` zugeführt. Mit `K = 1.0` kann die Ableitung als Frequenz in Hertz interpretiert werden.
- **Signalglättung und Regelungstechnik:**
  Der Baustein kann als Differenzierer für Prozessgrößen eingesetzt werden. Über `run` lässt sich die Berechnung bei Bedarf deaktivieren.
- **Diagnose und Überwachung:**
  Über die zusätzlichen Adapterausgänge `AR_DELTA_IN` und `AUDI_DELTA_T` stehen die internen Differenzwerte zur weiteren Verarbeitung bereit, etwa für Plausibilitätsprüfungen.
- **Adapterbasierte SubApp-Integration:**
  `FT_DERIV_AR` kann in übergeordneten Netzwerken direkt mit anderen AR-/AUDI-Bausteinen über `AdapterConnections` verbunden werden. Eine Klartext-Konvertierung ist nur dann nötig, wenn ein nachgelagerter Baustein keinen Adapter unterstützt. Dafür kann beispielsweise `AR_R_TO_REAL` verwendet werden.

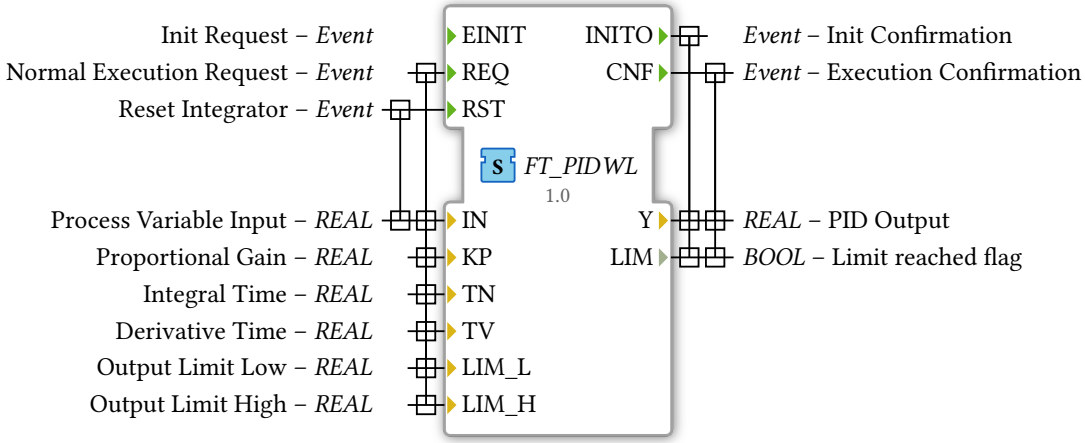
Vergleich mit ähnlichen Bausteinen

- **`FT_DERIV` (OSCAT):**
  Der ursprüngliche Baustein besitzt keine Adapter-Schnittstelle. Er arbeitet mit einfachen Events und Datenports. `FT_DERIV_AR` ergänzt diese fehlende Adapteranbindung.
- **`Q_NumericValue_AUDI` / `logiBUS_PI_IDA`:**
  Ähnliche Wrapper-Bausteine, die OSCAT-Funktionalität hinter einer Adaptergrenze kapseln. Sie folgen demselben Muster: Ein einfacher OSCAT-Baustein wird in ein eigenes FBType eingebettet, um die Anbindung an adapterbasierte Netzwerke zu ermöglichen.
- **`AR_R_TO_REAL`:**
  Kein Differenzierer, sondern ein Konvertierungsbaustein. Er wandelt einen `AR`-Adapterwert in einen Klartext-`REAL`-Wert um. Er kann in Kombination mit `FT_DERIV_AR` verwendet werden, wenn ein Klartextwert für nachgelagerte Bausteine benötigt wird.

Fazit

`FT_DERIV_AR` ist eine saubere und praktische Kapselung des OSCAT-Differenzierers `FT_DERIV` für adapterbasierte IEC-61499-Anwendungen. Durch die vollständig adapterbasierte Außengrenze bleibt der Baustein flexibel einsetzbar und vermeidet direkte Abhängigkeiten zwischen der `adapter`- und der `OSCAT`-Bibliothek. Die Durchreichung von `K` und `run` als einfache Parameter hält die Schnittstelle kompakt, während die zusätzlichen Ausgänge für `delta_in` und `delta_t` hilfreiche Diagnose- und Überwachungsmöglichkeiten bieten.

FT_PIDWL



PI Controller with Anti-Windup

version 1.3 13. nov. 2009 programmer hugo tested by tobias

FT_PIDWL is a PID controller with dynamic wind_up reset. The PID controller works according to the fomula Y = KP *(IN + KI * INTEG(e) + DERIV(e)). a rst will reset the integrator to 0 lim_h and Lim_l set the possible output range of the internal integrator. the output flags lim will signal that the output limits are active.

Interface

Event Inputs

Name	Comment	With
EINIT	Init Request	
REQ	Normal Execution Request	IN, KP, TN, TV, LIM_L, LIM_H
RST	Reset Integrator	IN

Event Outputs

Name	Comment	With
INITO	Init Confirmation	Y, LIM
CNF	Execution Confirmation	Y, LIM

Input Vars

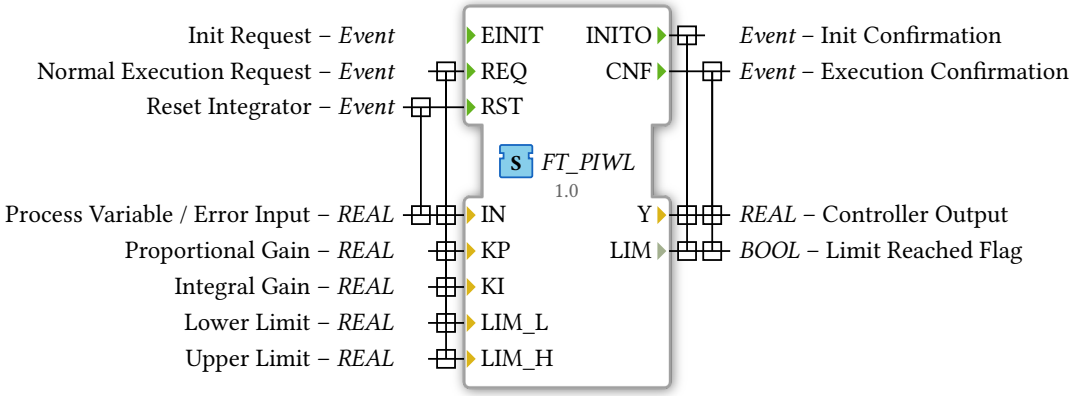
Name	Type	Initial Value	Comment
IN	REAL		Process Variable Input
KP	REAL	1.0	Proportional Gain
TN	REAL	1.0	Integral Time
TV	REAL	1.0	Derivative Time
LIM_L	REAL	−1.0E38	Output Limit Low
LIM_H	REAL	1.0E38	Output Limit High

Output Vars

Name	Type	Comment

Y	REAL	PID Output
LIM	BOOL	Limit reached flag

FT_PIWL



PI Controller with Anti-Windup

version 1.3 11. mar. 2009 programmer hugo tested by oscat

FT_PIWL is a PI controller. The PID controller works according to the fomula Y = IN *(KP+ KI * INTEG(e)). a rst will reset the integrator to 0 lim_h and Lim_l set the possible output range of the controller. the output flag lim will signal that the output limits are active. the integrator ist equipped with anti wind-up circuitry which limits trhe total output ranke to lim_l and lim_h

default values for KP = 1, KI = 1, ILIM_L = −1E37, iLIM_H = +1E38.

Interface

Event Inputs

Name	Comment	With
EINIT	Init Request	
REQ	Normal Execution Request	IN, KP, KI, LIM_L, LIM_H
RST	Reset Integrator	IN

Event Outputs

Name	Comment	With
INITO	Init Confirmation	Y, LIM
CNF	Execution Confirmation	Y, LIM

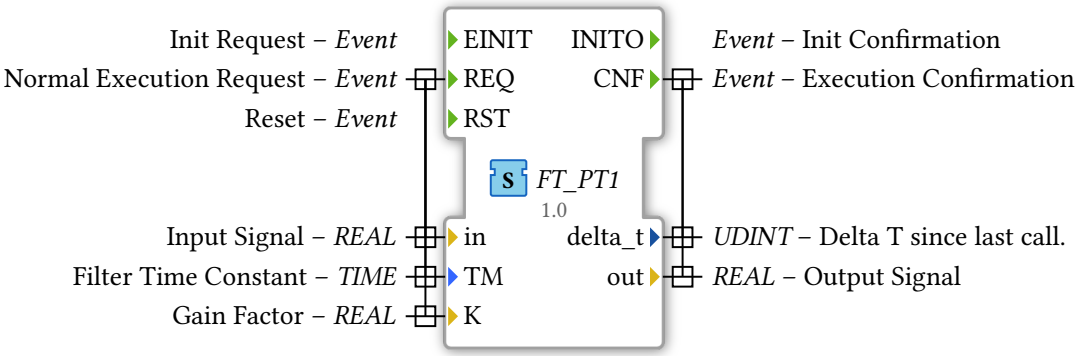
Input Vars

Name	Type	Initial Value	Comment
IN	REAL		Process Variable / Error Input
KP	REAL	1.0	Proportional Gain
KI	REAL	1.0	Integral Gain
LIM_L	REAL	−1.0E38	Lower Limit
LIM_H	REAL	1.0E38	Upper Limit

Output Vars

Name	Type	Comment
Y	REAL	Controller Output
LIM	BOOL	Limit Reached Flag

FT_PT1



• – ◦ • – ◦ • – ◦ •

Einleitung

`FT_PT1` ist ein Tiefpassfilter 1. Ordnung (PT1-Glied) aus der OSCAT-Bibliothek. Er dient zur Glättung von analogen Signalverläufen und Rauschunterdrückung mit einer programmierbaren Zeitkonstante `TM` und einem Verstärkungsfaktor `K`.

Der Baustein ist als `SimpleFB` (zustandsbehafteter Funktionsbaustein mit `EINIT`/`REQ`/`RST`) implementiert und führt eine zeitdiskrete Integration nach der PT1-Übertragungsfunktion durch:

$$T_M \cdot \frac{dy}{dt} + y(t) = K \cdot x(t) \tag{2.1}$$

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung	Mit Daten
`EINIT`	`Event`	Service-Initialisierung des Filters	
`REQ`	`Event`	Ausführungsanforderung für neuen Berechnungszyklus	`in`, `TM`, `K`
`RST`	`Event`	Invalidiert den Initialisierungszustand (`init = FALSE`) für Reinitialisierung beim nächsten `REQ`	

Ereignis-Ausgänge

Name	Typ	Beschreibung	Mit Daten
`INITO`	`Event`	Initialisierungsbestätigung	
`CNF`	`Event`	Ausführungsbestätigung nach Berechnung	`delta_t`, `out`

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung
`in`	`REAL`	`0.0`	Analoges Eingangssignal
`TM`	`TIME`	`T#0s`	Filter-Zeitkonstante. Bei `TM = T#0s` wird der Filter umgangen und `out = K * in` gesetzt.

K	REAL	1.0	Verstärkungsfaktor (Proportionalbeiwert)

Daten-Ausgänge

Name	Typ	Beschreibung
delta_t	UDINT	Vergangene Zeit seit dem letzten Aufruf in Mikrosekunden (μs)
out	REAL	Gefilterter Ausgangswert

Funktionsweise

1. **Initialisierung (EINIT)**:
   Setzt den Initialisierungszustand zurück, setzt `out = 0.0` und signalisiert Einsatzbereitschaft über INITO.
2. **Zyklische Berechnung (REQ)**:
   Bei jedem REQ-Ereignis ermittelt der Baustein die verstrichene Zeit seit dem letzten Aufruf (Δt) mikrosekundengenau über T_PLC_US().
   - Ist der Baustein noch nicht initialisiert (`init = FALSE`) oder ist `TM = T#0s`, wird der Filter (re-)initialisiert: `init := TRUE`, `out := K * in` wird direkt aus dem frisch abgetasteten Eingangssignal gesetzt, `delta_t := 0` und der Zeitstempel `last` aktualisiert.
   - Bei `TM > T#0s` wird der neue Ausgangswert nach der zeitdiskreten PT1-Formel berechnet:

$$\text{out}_{\text{neu}} = \text{out}_{\text{alt}} + (K \cdot \text{in} - \text{out}_{\text{alt}}) \cdot \frac{\Delta t}{T_{\text{eff}}} \tag{2.2}$$

     wobei $\Delta t = \text{delta_t} \cdot 10^{-6}\,\text{s}$ das gemessene Aufrufintervall in Sekunden (aus T_PLC_US()), $T_M = \text{TIME_TO_REAL}\ (TM)$ die Filterzeitkonstante in Sekunden und $T_{\text{eff}} = \max(T_M, \Delta t)$ die effektive Zeitkonstante ist. Wenn das Aufrufintervall Δt die eingestellte Filterzeit T_M überschreitet, wird T_{eff} auf Δt begrenzt, sodass der Gewichtungsfaktor $\frac{\Delta t}{T_{\text{eff}}}$ auf maximal 1.0 gedeckelt ist.
   - Um Unterläufe durch denormalisierte Fließkommazahlen zu vermeiden, werden Beträge $|out| < 1.0 \times 10^{-20}$ automatisch auf `0.0` gerundet.
3. **Filter-Reset (RST)**:
   Setzt den Initialisierungszustand zurück (setzt `init := FALSE`), ohne `out` oder den Zeitstempel unverzüglich zu verändern. Dadurch wird das Seeding bis zum nächsten REQ-Ereignis verzögert, wodurch das frische Eingangssignal (statt veralteter Werte oder `0.0`) abgetastet und die Zeitbasis aktualisiert wird.

Technische Besonderheiten

- **Zyklusunabhängige Zeitbasis**: Die Zeitdifferenz wird über T_PLC_US() in Mikrosekunden gemessen, wodurch Schwankungen der Aufrufzykluszeit kompensiert werden.
- **Begrenzung der effektiven Filterzeit ($T_{\text{eff}} = \max(T_M, \Delta t)$)**: Liegt das Aufrufintervall Δt über der Filterzeit T_M, deckelt der Baustein den Diskretisierungsfaktor $\frac{\Delta t}{T_{\text{eff}}}$ auf maximal 1.0. Dadurch wird verhindert, dass es bei langsamen Aufrufzyklen zu Oszillationen oder Euler-Überschwingen kommt.
- **Filter-Bypass bei `TM = 0`**: Ist `TM = T#0s`, schaltet der Baustein die Dämpfung ab und gibt das Eingangssignal direkt skaliert mit K aus.
- **Glatte Reset-Wiedereingliederung**: RST invalidiert den Initialisierungszustand (`init := FALSE`) ohne sofortiges Überschreiben des Ausgangs. Erst das nachfolgende REQ-Ereignis tastet das frische Eingangssignal ab, setzt `out := K * in` und aktualisiert den Zeitstempel `last`, wodurch verfälschte Sprünge durch veraltete Eingangswerte vermieden werden.
- **Projektspezifische Zeitkonvertierung**: Verwendet die projekteigene Hilfsfunktion TIME_TO_REAL.fct zur Umrechnung des TIME-Eingangs TM in Sekunden (REAL).

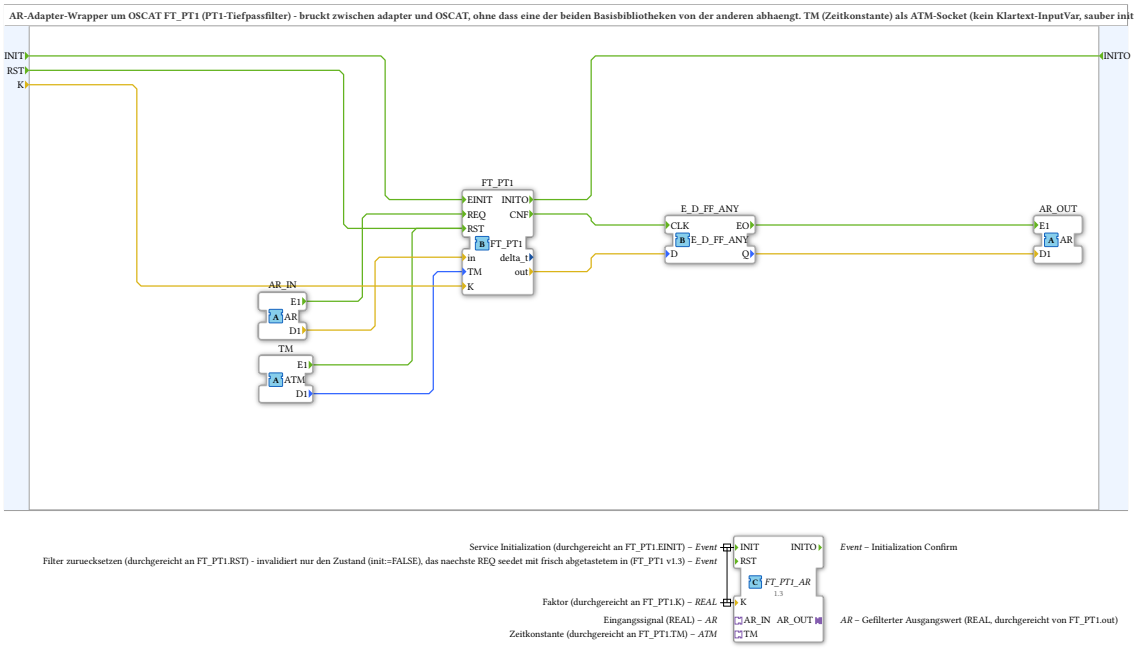
Anwendungsszenarien

- Dämpfung von stark schwankenden Sensormesswerten (z. B. Druck-, Temperatur- oder Spannungssignalen).
- Rauschfilterung für Steuerungseingänge.
- Sanftes Anfahren/Verzögern von Sollwerten.

Siehe auch

- `FT_PT1_AR` – AR-Adapter-Wrapper für `FT_PT1`.
- `FT_PT2` – Tiefpassfilter 2. Ordnung.
- `TIME_TO_REAL` – Hilfsfunktion zur Zeitkonvertierung.

FT_PT1_AR



Einleitung

`FT_PT1_AR` ist ein AR-Adapter-Wrapper um den OSCAT-Tiefpassfilterbaustein `FT_PT1`. Er kapselt die Tiefpassfilterung 1. Ordnung hinter einer rein adapterbasierten Schnittstelle für IEC 61499 Anwendungen.

Die Zeitkonstante `TM` ist als `ATM`-Socket ausgeführt (gemäß Section 13 des `iec61499-creator`-Skills), während der Verstärkungsfaktor `K` als gewöhnliche `InputVar` geführt wird. Der Baustein verwendet intern ein `E_D_FF_ANY` D-Flipflop, um Änderungen am gefilterten Wert zu erkennen und Adapter-Events nur bei tatsächlichen Wertänderungen (bzw. beim ersten Aufruf nach dem Start) zu senden.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung
`INIT`	`EInit`	Service-Initialisierung, wird an `FT_PT1.INIT` durchgereicht
`RST`	`Event`	Invalidiert den Filterzustand über `FT_PT1.RST` (`init := FALSE`)

Ereignis-Ausgänge

Name	Typ	Beschreibung
`INITO`	`EInit`	Initialisierungsbestätigung von `FT_PT1`

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung
`K`	`REAL`	`1.0`	Verstärkungsfaktor, wird an `FT_PT1.K` durchgereicht

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden. Die Ausgabe erfolgt ausschließlich über den Adapter-Plug `AR_OUT`.

Adapter

Schnittstelle	Richtung	Adaptertyp	Datentyp	Beschreibung
`AR_IN`	Socket	`AR`	`REAL`	Eingangssignal für die Filterung
`TM`	Socket	`ATM`	`TIME`	Zeitkonstante des Tiefpassfilters (`TM.D1` wird an `FT_PT1.TM` weitergeleitet)
`AR_OUT`	Plug	`AR`	`REAL`	Gefilterter Ausgangswert (`FT_PT1.out` via `E_D_FF_ANY`)

Funktionsweise

Der Baustein verbindet `FT_PT1` (OSCAT) und `E_D_FF_ANY` in einem internen Netzwerk:

1. **Messwertverarbeitung**:
   Ein Ereignis auf `AR_IN.E1` löst `FT_PT1.REQ` aus. `AR_IN.D1` liefert den aktuellen Messwert.
2. **Zeitkonstante & Reinitialisierung**:
   Die Filterzeit `TM.D1` kommt vom `ATM`-Socket und wird an `FT_PT1.TM` übergeben. Ein Ereignis auf `TM.E1` löst zusätzlich intern direkt `FT_PT1.RST` aus. Dadurch wird der Filterzustand invalidiert (`init := FALSE`), sodass beim nachfolgenden `AR_IN.E1`-Ereignis der Filter mit frischem Eingangssignal (`out := K * in`) und der neuen Zeitkonstante sauber reinitialisiert wird.
3. **Änderungserkennung & Entkopplung (`E_D_FF_ANY`)**:
   Nach der Berechnung feuert `FT_PT1.CNF` den `CLK`-Eingang des internen `E_D_FF_ANY`. Das Flipflop gibt beim ersten Aufruf nach dem Start den gefilterten Ausgangswert bedingungslos über `AR_OUT.E1` / `AR_OUT.D1` aus. Bei nachfolgenden Zyklen werden Ereignisse nur dann gefeuert, wenn sich der gefilterte Wert tatsächlich vom vorherigen Ausgangswert unterscheidet.
4. **Reset & Initialisierung**:
   - `INIT` steuert `FT_PT1.INIT` und meldet Vollzug über `INITO`. Unverdrahtete `INIT`-Events feuern beim Deployment automatisch einmalig.
   - `RST` sowie jedes eintreffende `TM.E1`-Ereignis werden an `FT_PT1.RST` weitergeleitet, um den Filterzustand zu invalidieren (`init := FALSE`). Die Reinitialisierung erfolgt beim nächsten `AR_IN.E1`-Aufruf.

Technische Besonderheiten

- **Saubere Adaptergrenze**: Verhindert das direkte Zugreifen auf interne `.E1`/`.D1`-Datenstrukturen in SubApp-Netzwerken.
- **Automatische Filter-Reinitialisierung bei TM-Änderung**: Ein Eintreffen von `TM.E1` löst intern `FT_PT1.RST` aus (setzt `init := FALSE`). Beim nächsten `AR_IN.E1`-Aufruf wird `out` mit dem aktuellen Messwert (`K * in`) reinitialisiert und der Zeitstempel aktualisiert, damit geänderte Filterzeiten ohne Sprungartefakte oder veraltete Seeding-Werte sauber wirksam werden.
- **Entlastung der Ereigniskette**: `E_D_FF_ANY` verhindert unnötiges Event-Spamming in nachgelagerten SubApps bei unveränderten Werten.
- **`ATM`-Socket-Konvention**: Zeitparameter werden nicht als nackte Variablen deklariert, sondern über `ATM`-Adapter-Sockets eingebunden (an der Instanziierungsstelle z. B. über `initval_ATM` zu speisen).

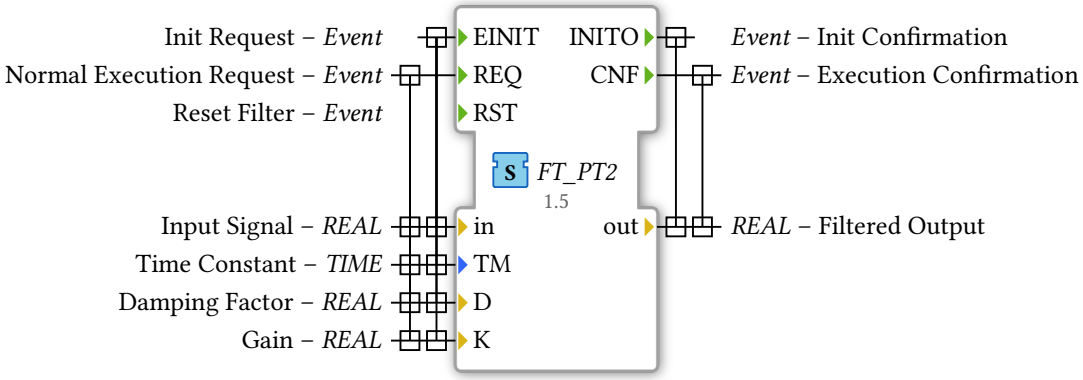
Anwendungsszenarien

- Glättung von analogen Sensorwerten (z. B. Druck, Temperatur, Drehzahl) in adapterbasierten SubApp-Architekturen.
- Rauschfilterung vor Schwellwert- oder Hysterese-Bausteinen (`AR_D_FF_HYS_TMIN`).

Siehe auch

- `FT_PT1` – Grundbaustein (OSCAT).
- `FT_PT2_AR` – AR-Adapter-Wrapper für PT2-Filter 2. Ordnung.
- `FT_DERIV_AR` – AR-Adapter-Wrapper für Differenzierer.

FT_PT2



2nd grade filter with programmable times

Interface

Event Inputs

Name	Comment	With
EINIT	Init Request	in, TM, D, K
REQ	Normal Execution Request	in, TM, D, K
RST	Reset Filter	

Event Outputs

Name	Comment	With
INITO	Init Confirmation	out
CNF	Execution Confirmation	out

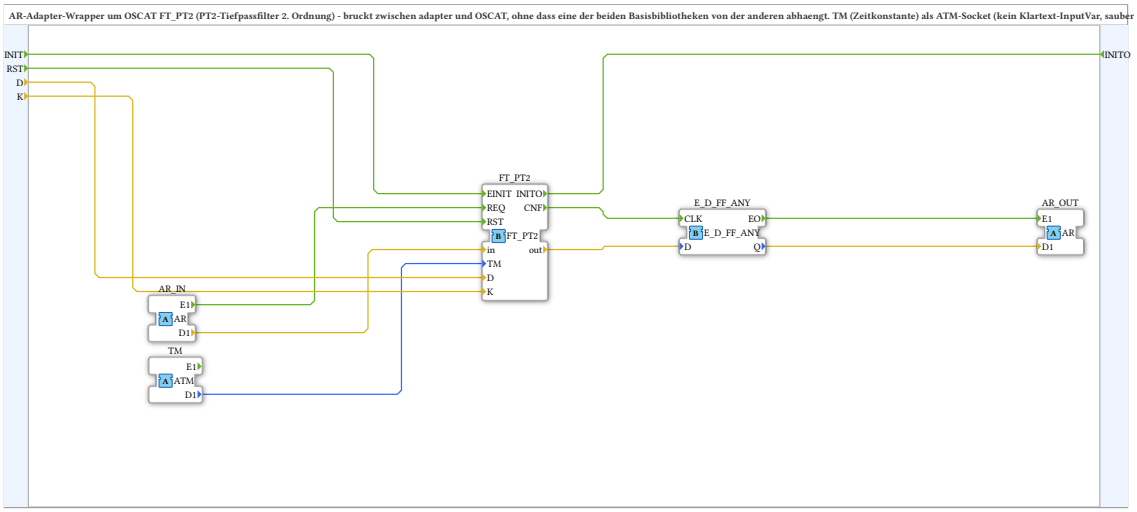
Input Vars

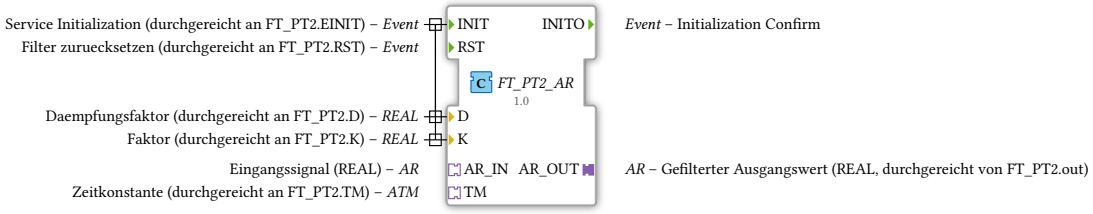
Name	Type	Initial Value	Comment
in	REAL		Input Signal
TM	TIME		Time Constant
D	REAL		Damping Factor
K	REAL	1.0	Gain

Output Vars

Name	Type	Comment
out	REAL	Filtered Output

FT_PT2_AR





Einleitung

`FT_PT2_AR` ist ein AR-Adapter-Wrapper um den OSCAT-Tiefpassfilterbaustein 2. Ordnung `FT_PT2`. Er ermöglicht die Filterung 2. Ordnung mit einstellbarer Zeitkonstante `TM`, Dämpfung `D` und Verstärkung `K` in rein adapterbasierten IEC 61499 Anwendungen.

Die Zeitkonstante `TM` wird über einen `ATM`-Adapter-Socket bereitgestellt (gemäß Section 13 des `iec61499-creator`-Skills), während Dämpfung `D` und Verstärkung `K` als gewöhnliche `InputVars` geführt werden. Ein internes `E_D_FF_ANY` D-Flipflop sorgt dafür, dass Ausgangsevent `AR_OUT.E1` beim ersten Zyklus bedingungslos und danach nur bei tatsächlichen Wertänderungen gesendet wird.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung
`INIT`	`EInit`	Service-Initialisierung, wird an `FT_PT2.INIT` durchgereicht
`RST`	`Event`	Setzt den Filterausgang über `FT_PT2.RST` zurück

Ereignis-Ausgänge

Name	Typ	Beschreibung
`INITO`	`EInit`	Initialisierungsbestätigung von `FT_PT2`

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung
`D`	`REAL`	`0.0`	Dämpfungsfaktor des PT2-Filters (z. B. 0.707 für Butterworth-Charakteristik)
`K`	`REAL`	`1.0`	Verstärkungsfaktor

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden. Die Ausgabe erfolgt ausschließlich über den Adapter-Plug `AR_OUT`.

Adapter

Schnittstelle	Richtung	Adaptertyp	Datentyp	Beschreibung
`AR_IN`	Socket	`AR`	`REAL`	Eingangssignal für die PT2-Filterung
`TM`	Socket	`ATM`	`TIME`	Zeitkonstante des Tiefpassfilters (`TM.D1` an `FT_PT2.TM`)
`AR_OUT`	Plug	`AR`	`REAL`	Gefilterter Ausgangswert (`FT_PT2.out` via `E_D_FF_ANY`)

Funktionsweise

Der Baustein bettet `FT_PT2` in ein FBNetzwerk ein:

1. **Signalverarbeitung**:
   Ein Ereignis auf `AR_IN.E1` löst `FT_PT2.REQ` aus. `AR_IN.D1` wird an `FT_PT2.in` weitergegeben.
2. **Parameterisierung**:
   `TM.D1` wird vom `ATM`-Socket eingelesen. `D` und `K` werden direkt als Parameter übergeben.
3. **Zustandsbehaftete Integration 2. Ordnung**:
   `FT_PT2` verarbeitet das Signal intern über zwei gekoppelte Integratoren (`INTEGRATE`).
4. **Änderungsfilterung (`E_D_FF_ANY`)**:
   `FT_PT2.CNF` steuert das interne `E_D_FF_ANY`. Das Flipflop gibt beim ersten Zyklus den berechneten Ausgangswert an `AR_OUT` weiter. In Folgezyklen werden `AR_OUT.E1` und `AR_OUT.D1` nur aktualisiert, wenn der berechnete Filterwert vom bisherigen Wert abweicht.
5. **Reset & Initialisierung**:
   `INIT` steuert `FT_PT2.INIT`. `RST` setzt die internen Speicher der beiden Integrationsstufen zurück.

Technische Besonderheiten

- **PT2-Charakteristik**: Bietet im Vergleich zum PT1-Filter eine stärkere Dämpfung höherfrequenter Störungen (12 dB/Oktave Abfall).
- **Integrierte Änderungsfilterung**: Vermeidet unnötige Adapter-Events in nachgelagerten SubApps.
- **`ATM`-Socket für Zeitparameter**: Konform zu den Projektdesignregeln für Zeit- und Konfigurationskonstanten.

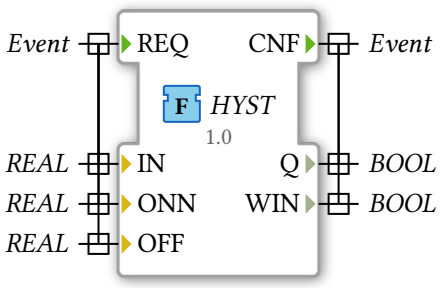
Anwendungsszenarien

- Filterung von hochfrequenten Störungen auf analogen Signalen (z. B. Kraft-, Druck- oder Fahrgeschwindigkeitssignalen).
- Schwingungsdämpfung in Regelkreisen.

Siehe auch

- `FT_PT2` – Grundbaustein (OSCAT).
- `FT_PT1_AR` – AR-Adapter-Wrapper für PT1-Filter 1. Ordnung.
- `FT_DERIV_AR` – AR-Adapter-Wrapper für Differenzierer.

HYST



🎧 Podcast

- Das Relais im Detail: Schaltverstärker, Schutz und die Geheimnisse von A1/A2, 85/86 und der Hysterese

Interface

Event Inputs

Name	Comment	With
REQ		IN, ONN, OFF

Event Outputs

Name	Comment	With
CNF		Q, WIN

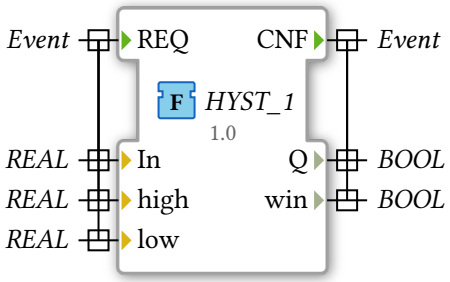
Input Vars

Name	Type	Comment
IN	REAL	
ONN	REAL	
OFF	REAL	

Output Vars

Name	Type	Comment
Q	BOOL	
WIN	BOOL	

HYST_1



Hystereses function

this hysteresis function switches the output high if the input signal reaches above high and will switch to low when the input falls back below low value. a separate output mid is set if the input stays between low and high value.

Interface

Event Inputs

Name	Comment	With
REQ		In, high, low

Event Outputs

Name	Comment	With

CNF		Q, win

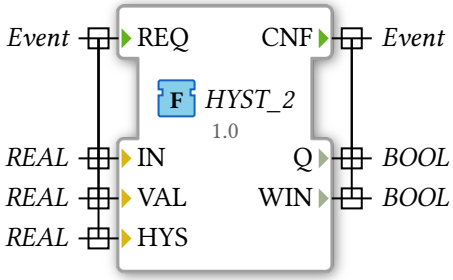
Input Vars

Name	Type	Comment
In	REAL	
high	REAL	
low	REAL	

Output Vars

Name	Type	Comment
Q	BOOL	
win	BOOL	

HYST_2



Hystereses function

this hysteresis function switches the output high if the input signal reaches above val + hys/2 and will switch to low when the input falls back below val - hys/2 value. a separate output mid is set if the input stays between low and high value.

Interface

Event Inputs

Name	Comment	With
REQ		IN, VAL, HYS

Event Outputs

Name	Comment	With
CNF		Q, WIN

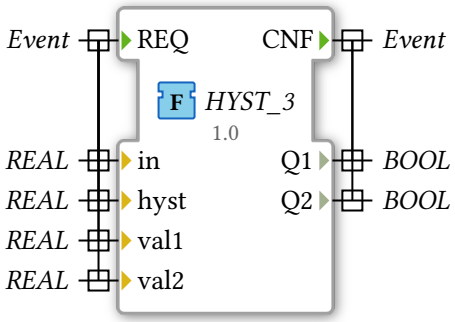
Input Vars

Name	Type	Comment
IN	REAL	
VAL	REAL	
HYS	REAL	

Output Vars

Name	Type	Comment
Q	BOOL	
WIN	BOOL	

HYST_3



Hystereses function

this is a double hysteresis function. Out1 follows a hysteresis function defined by val1and hyst, while out 2 follows val2 and hyst. if the input signal is between the two hysteresis switches (val1 and val2) then non of the outputs is active.

Interface

Event Inputs

Name	Comment	With
REQ		in, hyst, val1, val2

Event Outputs

Name	Comment	With
CNF		Q1, Q2

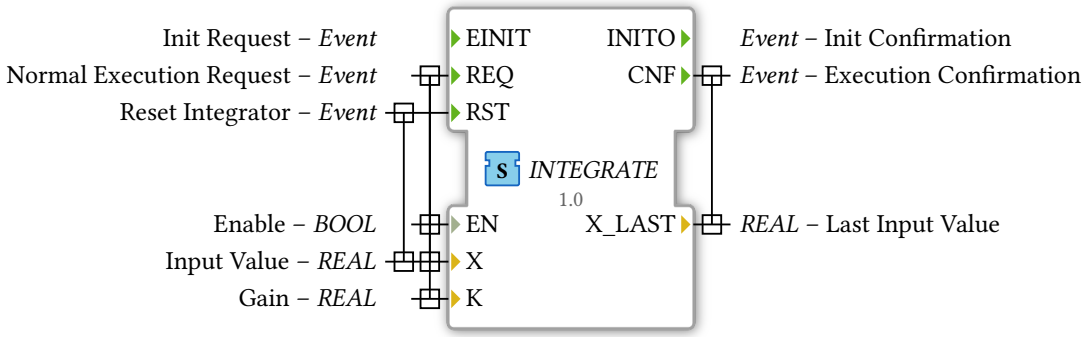
Input Vars

Name	Type	Comment
in	REAL	
hyst	REAL	
val1	REAL	
val2	REAL	

Output Vars

Name	Type	Comment
Q1	BOOL	
Q2	BOOL	

INTEGRATE



Integrator with I/O for out

Interface

Event Inputs

Name	Comment	With
EINIT	Init Request	
REQ	Normal Execution Request	EN, X, K, Y
RST	Reset Integrator	X

Event Outputs

Name	Comment	With
INITO	Init Confirmation	
CNF	Execution Confirmation	X_LAST, Y

Input Vars

Name	Type	Initial Value	Comment
EN	BOOL	TRUE	Enable
X	REAL		Input Value
K	REAL	1.0	Gain

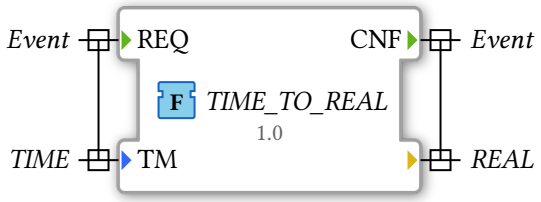
Output Vars

Name	Type	Comment
X_LAST	REAL	Last Input Value

InOut Vars

Name	Type	Comment
Y	REAL	Output Value

TIME_TO_REAL



Einleitung

`TIME_TO_REAL` ist eine projekteigene Hilfsfunktion (`FunctionType`), die eine Zeitdauer vom Typ `TIME` in einen Fließkommawert (`REAL`) umrechnet. Das Ergebnis entspricht der Zeitdauer in **Sekunden**.

Diese Funktion wird innerhalb von OSCAT-Regelungs- und Filterbausteinen (wie `FT_PT1`) verwendet, um Zeitangaben aus der SPS-Umgebung für mathematische Differential- und Integrationsberechnungen aufzubereiten.

Schnittstellenstruktur

Eingänge

Name	Typ	Beschreibung
`IN`	`TIME`	Eingangs-Zeitdauer

Ausgänge

Name	Typ	Beschreibung
(Return)	`REAL`	Berechnete Zeitdauer in Sekunden

Funktionsweise

Die Funktion rechnet den internen Millisekunden-Wert des `TIME`-Eingangs in Sekunden um:

$$\text{Return} = \text{TIME_TO_UDINT}\ (\text{IN}) \cdot 1.0 \times 10^{-3} \tag{2.3}$$

Beispiel:

- `IN = T#1s` → `Return = 1.0`
- `IN = T#500ms` → `Return = 0.5`
- `IN = T#2m30s` → `Return = 150.0`

Technische Besonderheiten

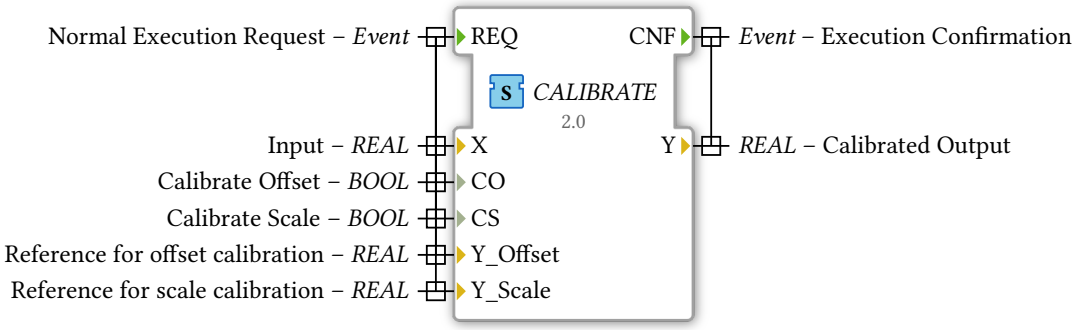
- **Projektspezifische Erweiterung**: `TIME_TO_REAL` ist keine Standard-Funktion der OSCAT-Bibliothek, sondern ein lokaler Konvertierungshelfer für IEC 61499.
- **Skalierungsgenauigkeit**: Verwendet explizite Fließkommaskalierung (1.0×10^{-3}), um Ganzzahl-Abrundungsfehler bei der Division zu vermeiden.

Siehe auch

- `FT_PT1` – Tiefpassfilter 1. Ordnung (verwendet `TIME_TO_REAL`).
- `FT_PT2` – Tiefpassfilter 2. Ordnung.

measurements

CALIBRATE



• – ◦ • – ◦ • – ◦ •

Einleitung

`CALIBRATE` führt eine Zwei-Punkt-Kalibrierung (Offset & Skalierung) eines analogen Eingangssignals durch: `Y = (X + OFFSET) * SCALE`. Die Kalibrierung wird nicht über eigene Ereignisse, sondern über die booleschen Eingänge `CO`/`CS` ausgelöst, die bei jedem `REQ` mitgeprüft werden. Für eine ereignisgesteuerte Variante siehe E_CALIBRATE, Details zum Verfahren finden Sie unter Kalibrierverfahren Zwei- und Dreipunkt.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normale Ausführungsanforderung, liefert `X`, `CO`, `CS`, `Y_Offset`, `Y_Scale`, `OFFSET`, `SCALE`. Berechnet bei jedem Aufruf `Y` neu und führt dabei ggf. eine Kalibrierung durch (siehe Funktionsweise).

Ereignis-Ausgänge

- **CNF**: Bestätigt die Ausführung, liefert `Y`, `OFFSET` und `SCALE`.

Daten-Eingänge

- **X** (REAL): Roheingangswert.
- **CO** (BOOL): Löst bei `TRUE` eine Offset-Kalibrierung aus.
- **CS** (BOOL): Löst bei `TRUE` eine Skalierungs-Kalibrierung aus (nur sinnvoll, wenn zuvor `CO` ausgeführt wurde).
- **Y_Offset** (REAL): Zielwert für `Y` am unteren Referenzpunkt (bei Offset-Kalibrierung).
- **Y_Scale** (REAL): Zielwert für `Y` am oberen Referenzpunkt (bei Skalierungs-Kalibrierung).

Daten-Ausgänge

- **Y** (REAL): Kalibrierter Ausgangswert.

Ein-/Ausgabevariablen (InOut)

- **OFFSET** (REAL, Startwert `0.0`): Gespeicherter Offset-Wert – wird bei `CO` neu berechnet und bleibt darüber hinaus persistent im Aufrufer erhalten.
- **SCALE** (REAL, Startwert `1.0`): Gespeicherter Skalierungsfaktor – wird bei `CS` neu berechnet und bleibt darüber hinaus persistent im Aufrufer erhalten.

Funktionsweise

Bei jedem `REQ` prüft `CALIBRATE` zuerst `CO` und `CS` (als `ELSIF`-Kette, also höchstens eine Kalibrierung pro Aufruf):

- Ist `CO = TRUE`: `OFFSET := Y_Offset - X` – verschiebt `X` so, dass bei der aktuellen Referenz `Y = Y_Offset` gilt (korrekt nur, wenn `SCALE = 1`).
- Sonst, ist `CS = TRUE`: `SCALE := Y_Scale / (X + OFFSET)` – skaliert so, dass bei der aktuellen Referenz `Y = Y_Scale` gilt. Setzt voraus, dass `OFFSET` bereits über `CO` bestimmt wurde.

Anschließend wird in jedem Fall `Y := (X + OFFSET) * SCALE` berechnet und über `CNF` ausgegeben.

Ablauf einer Zwei-Punkt-Kalibrierung:

1. Niedrige Referenz anlegen, `Y_Offset` auf den gewünschten Ausgabewert setzen, `CO = TRUE` – ergibt `OFFSET`.
2. Hohe Referenz anlegen, `Y_Scale` auf den gewünschten Ausgabewert setzen, `CS = TRUE` – ergibt `SCALE`.

Beispiel (4-20-mA-Drucksensor über logiBUS, normiert auf `0.0 .. 1.0`, gewünschter Ausgabebereich `0.0 .. 500.0`):

Schritt	Aktion	Ergebnis
1	4 mA anlegen (`X=0.0`), `Y_Offset=0.0`, `CO=TRUE`	`OFFSET = 0`
2	20 mA anlegen (`X=1.0`), `Y_Scale=500.0`, `CS=TRUE`	`SCALE = 500`

Ergebnis: `Y = (X + 0) * 500 = X * 500 = 0..500`.

Technische Besonderheiten

- **Reihenfolge nicht erzwungen**: `CO` muss vor `CS` ausgeführt werden, da `CS` den bereits ermittelten `OFFSET` verwendet. Als `SimpleFB` mit nur einem `ECState` liegt die korrekte Reihenfolge in der Verantwortung des Aufrufers.
- **Y nach CO nur korrekt bei SCALE = 1**: Die Formel `OFFSET := Y_Offset - X` liefert nach der Offset-Kalibrierung nur dann exakt `Y = Y_Offset`, wenn `SCALE` noch seinen Ausgangswert `1.0` hat.
- **Auslösung über BOOL statt Event**: Anders als bei den `E_CALIBRATE*`-Varianten läuft die Kalibrierung hier über die Dateneingänge `CO`/`CS`, die bei jedem `REQ` neu ausgewertet werden – kein eigenes Kalibrierungsereignis nötig, aber auch kein separates Bestätigungsereignis pro Kalibrierschritt.

Zustandsübersicht

`CALIBRATE` ist ein `SimpleFB` mit einem einzigen `ECState` (`REQ`): Jeder Aufruf führt unmittelbar den `REQ`-Algorithmus aus und feuert `CNF`. Es gibt keine Zustandsübergänge und keine ECC im eigentlichen Sinn – die „Kalibrierphase“ ergibt sich rein aus dem Wert von `CO`/`CS` zum Zeitpunkt des Aufrufs.

Anwendungsszenarien

- Skalierung analoger Sensoren (Druck, Temperatur, Füllstand) mit Zwei-Punkt-Referenzkalibrierung
- Systeme, in denen die Kalibrierauslösung bereits als boolesches Signal vorliegt (z. B. Taster/Schalter statt separater Ereignisquelle)
- Einfache Anwendungen, bei denen eine versehentlich falsche Reihenfolge (`CS` vor `CO`) kein Risiko darstellt

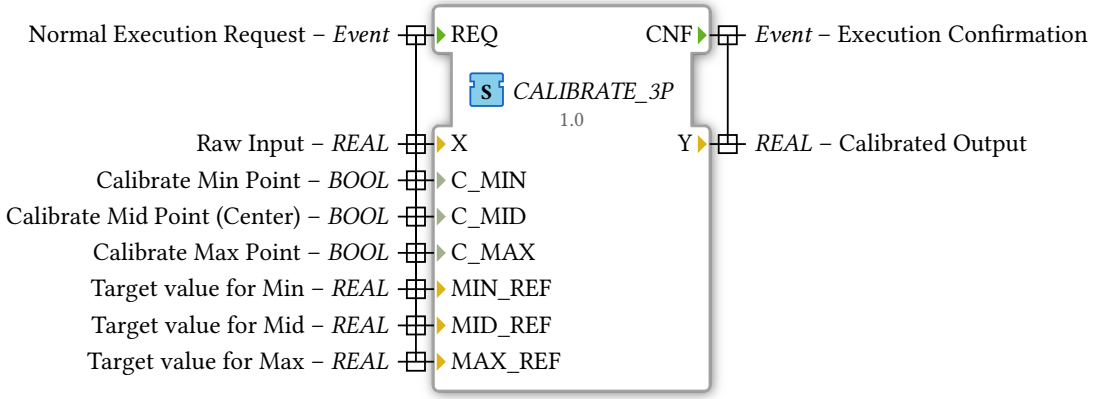
⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit E_CALIBRATE, das dieselbe Zwei-Punkt-Kalibrierung ereignisgesteuert (`EICO`/`EICS` statt `CO`/`CS`) durchführt, Weiterführende Informationen zu den Kalibrierverfahren finden Sie unter Kalibrierverfahren Zwei- und Dreipunkt. Für eine Drei-Punkt-Kalibrierung (z. B. Joysticks mit Mittelstellung) siehe CALIBRATE_3P.

Fazit

`CALIBRATE` bietet eine einfache, boolean-gesteuerte Zwei-Punkt-Kalibrierung für analoge Signale und eignet sich für Anwendungen, in denen die korrekte Reihenfolge von Offset- und Skalierungskalibrierung vom Aufrufer sichergestellt wird.

CALIBRATE_3P



• – ◦ • – ◦ • – ◦ •

Einleitung

`CALIBRATE_3P` ist eine Drei-Punkt-Kalibrierung für Roheingänge mit Mittelstellung, wie sie typischerweise bei Joysticks auftritt. Statt einer linearen Skalierung wie CALIBRATE interpoliert er stückweise linear zwischen drei gespeicherten Referenzpunkten (Min, Mid, Max) und kompensiert damit auch eine Mittelstellung, die nicht exakt in der Mitte des Rohbereichs liegt.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normale Ausführungsanforderung, liefert `X`, `C_MIN`, `C_MID`, `C_MAX`, `MIN_REF`, `MID_REF`, `MAX_REF`, `X_MIN`, `X_MID`, `X_MAX`. Berechnet bei jedem Aufruf `Y` neu und führt dabei ggf. eine Kalibrierung durch.

Ereignis-Ausgänge

- **CNF**: Bestätigt die Ausführung, liefert `Y`, `X_MIN`, `X_MID`, `X_MAX`.

Daten-Eingänge

- **X** (REAL): Roheingangswert.
- **C_MIN** (BOOL): Löst bei `TRUE` die Kalibrierung des Minimalpunkts aus.
- **C_MID** (BOOL): Löst bei `TRUE` die Kalibrierung des Mittelpunkts aus.
- **C_MAX** (BOOL): Löst bei `TRUE` die Kalibrierung des Maximalpunkts aus.
- **MIN_REF** (REAL, Startwert `0.0`): Zielausgabewert für den Minimalpunkt.
- **MID_REF** (REAL, Startwert `50.0`): Zielausgabewert für den Mittelpunkt.
- **MAX_REF** (REAL, Startwert `100.0`): Zielausgabewert für den Maximalpunkt.

Daten-Ausgänge

- **Y** (REAL): Kalibrierter, interpolierter und auf `MIN_REF..MAX_REF` begrenzter Ausgangswert.

Ein-/Ausgabevariablen (InOut)

- **X_MIN** (REAL, Startwert `0.0`): Gespeicherter Rohwert am Minimalpunkt.
- **X_MID** (REAL, Startwert `50.0`): Gespeicherter Rohwert am Mittelpunkt.
- **X_MAX** (REAL, Startwert `100.0`): Gespeicherter Rohwert am Maximalpunkt.

Funktionsweise

Bei jedem `REQ` prüft `CALIBRATE_3P` zunächst `C_MIN`, `C_MID` und `C_MAX` (als `ELSIF`-Kette, höchstens ein Kalibrierpunkt pro Aufruf) und speichert bei gesetztem Flag den aktuellen Rohwert `X` im jeweiligen `X_MIN`/`X_MID`/`X_MAX`.

Anschließend wird `Y` stückweise linear interpoliert:

- Für `X < X_MID`: Interpolation zwischen `(X_MIN, MIN_REF)` und `(X_MID, MID_REF)`.
- Für `X >= X_MID`: Interpolation zwischen `(X_MID, MID_REF)` und `(X_MAX, MAX_REF)`.
- Das Ergebnis wird auf den Bereich `MIN_REF..MAX_REF` begrenzt (Clipping).

Die drei Kalibrierpunkte können in **beliebiger Reihenfolge** und beliebig oft einzeln wiederholt werden – pro REQ-Aufruf wird jedoch höchstens einer davon aktualisiert.

Beispiel (Joystick mit Mittelpunkt-Drift, Rohbereich `0..1000`, gewünschter Ausgabebereich `-100..0..100`):

Schritt	Aktion	Ergebnis
1	Joystick auf **Minimum**, `C_MIN=TRUE`	`X_MIN = 50` (Drift gegenüber ideal `0`)
2	Joystick auf **Mitte**, `C_MID=TRUE`	`X_MID = 520` (Drift gegenüber ideal `500`)
3	Joystick auf **Maximum**, `C_MAX=TRUE`	`X_MAX = 980` (Drift gegenüber ideal `1000`)

Ergebnis: `Y` wird zwischen `MIN_REF=-100`, `MID_REF=0` und `MAX_REF=100` interpoliert und begrenzt – die tatsächliche (verschobene) Mittelstellung des Joysticks liefert exakt `Y=0`.

Technische Besonderheiten

- **Kompensation der Mittelstellung**: Anders als bei einer reinen Zwei-Punkt-Kalibrierung (siehe CALIBRATE) wird die Mittelstellung explizit erfasst – wichtig bei Joysticks, deren mechanische Ruhelage nicht exakt in der Mitte des elektrischen Rohbereichs liegt.
- **Ausgabe stets begrenzt**: `Y` wird nach der Interpolation auf `MIN_REF..MAX_REF` geklippt, auch wenn `X` außerhalb der kalibrierten Rohwerte liegt.
- **Degenerierte Kalibrierpunkte**: Liegt `X_MID` nicht oberhalb von `X_MIN` (bzw. `X_MAX` nicht oberhalb von `X_MID`), liefert die betroffene Interpolation konstant `MIN_REF` bzw. `MID_REF`, statt durch Null zu teilen.
- **Reihenfolge nicht erzwungen**: Wie bei `CALIBRATE` liegt es als `SimpleFB` in der Verantwortung des Aufrufers, sinnvolle Kalibrierwerte zu liefern. Für eine ereignisgesteuerte Variante siehe E_CALIBRATE_3P.

Zustandsübersicht

`CALIBRATE_3P` ist ein `SimpleFB` mit einem einzigen `ECState` (`REQ`): Jeder Aufruf führt unmittelbar den `REQ`-Algorithmus aus und feuert `CNF`. Es gibt keine Zustandsübergänge – welcher Kalibrierpunkt (falls überhaupt) aktualisiert wird, ergibt sich rein aus `C_MIN`/`C_MID`/`C_MAX` zum Zeitpunkt des Aufrufs.

Anwendungsszenarien

- Kalibrierung von Joysticks und ähnlichen Bedienelementen mit mechanischer Mittelstellung
- Roheingänge mit nichtlinearem oder nicht symmetrischem Verhalten um den Nullpunkt
- Anwendungen, bei denen einzelne Kalibrierpunkte unabhängig voneinander nachjustiert werden müssen, ohne die anderen zu beeinflussen

⚖️ Vergleich mit ähnlichen Bausteinen

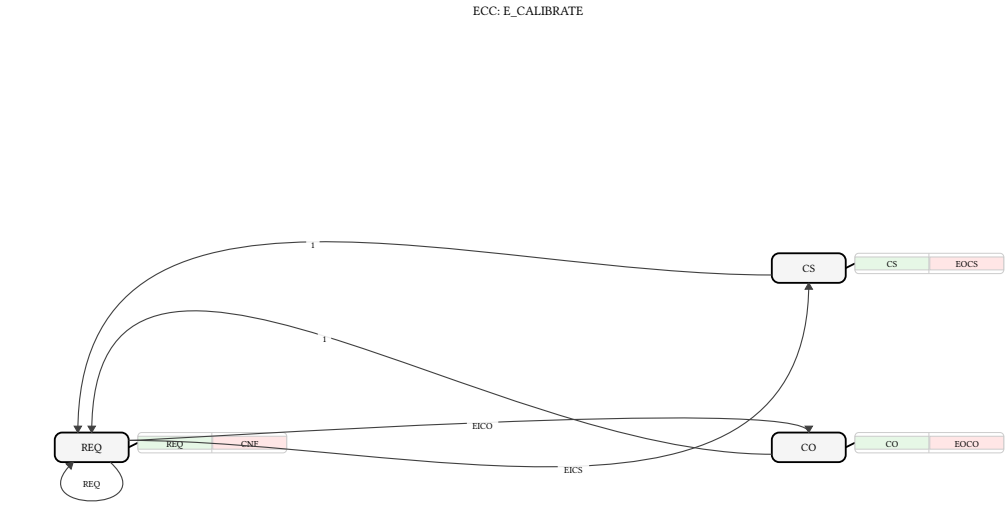
Details zum Dreipunkt-Kalibrierverfahren finden Sie unter Kalibrierverfahren Zwei- und Dreipunkt.

Vergleich mit CALIBRATE, das dieselbe boolean-gesteuerte Auslösung nutzt, jedoch nur eine lineare Zwei-Punkt-Kalibrierung ohne Mittelstellung bietet, sowie mit E_CALIBRATE_3P, das dieselbe Drei-Punkt-Logik ereignisgesteuert (`EI_MIN`/`EI_MID`/`EI_MAX`) mit eigenen Bestätigungsereignissen pro Punkt bereitstellt.

Fazit

`CALIBRATE_3P` eignet sich überall dort, wo eine einfache Zwei-Punkt-Kalibrierung nicht ausreicht, weil der Rohwertbereich um einen (ggf. verschobenen) Mittelpunkt nicht symmetrisch ist – allen voran bei Joysticks und ähnlichen Bedienelementen.

E_CALIBRATE



• – ◦ • – ◦ • – ◦ •

Einleitung

`E_CALIBRATE` ist die ereignisgesteuerte Variante von CALIBRATE: Die Zwei-Punkt-Kalibrierung (`Y = (X + OFFSET) * SCALE`) wird nicht über boolesche Datenwerte, sondern über die eigenen Ereignisse `EICO`/`EICS` ausgelöst, mit jeweils eigenem Bestätigungsereignis. Details zum Kalibrierverfahren finden Sie unter Kalibrierverfahren Zwei- und Dreipunkt.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normale Ausführungsanforderung – berechnet `Y`. Liefert `X`, `Y_Offset`, `Y_Scale`, `OFFSET`, `SCALE`.
- **EICO**: Kalibriert den Offset (`OFFSET := Y_Offset - X`). Liefert `X`, `Y_Offset`.
- **EICS**: Kalibriert die Skalierung (`SCALE := Y_Scale / (X + OFFSET)`). Liefert `X`, `Y_Scale`, `OFFSET`.

Ereignis-Ausgänge

- **CNF**: Bestätigt `REQ`, liefert `Y`.
- **EOCO**: Bestätigt die Offset-Kalibrierung, liefert `OFFSET`.
- **EOCS**: Bestätigt die Skalierungs-Kalibrierung, liefert `SCALE`.

Daten-Eingänge

- **X** (REAL): Roheingangswert.
- **Y_Offset** (REAL): Zielausgabewert am unteren Referenzpunkt.
- **Y_Scale** (REAL): Zielausgabewert am oberen Referenzpunkt.

Daten-Ausgänge

- **Y** (REAL): Kalibrierter Ausgangswert.

Ein-/Ausgabevariablen (InOut)

- **OFFSET** (REAL, Startwert `0.0`): Gespeicherter Offset-Wert.
- **SCALE** (REAL, Startwert `1.0`): Gespeicherter Skalierungsfaktor.

Funktionsweise

`E_CALIBRATE` besitzt drei ECC-Zustände, die alle direkt vom Zustand `REQ` aus erreichbar sind:

- **REQ** (Startzustand): Bei `REQ`-Ereignis wird `Y := (X + OFFSET) * SCALE` berechnet und `CNF` gefeuert; der Baustein bleibt in `REQ`.
- **CO**: Bei `EICO` wechselt der Baustein nach `CO`, führt `OFFSET := Y_Offset - X` aus, feuert `EOCO` und kehrt sofort zu `REQ` zurück.
- **CS**: Bei `EICS` wechselt der Baustein nach `CS`, führt `SCALE := Y_Scale / (X + OFFSET)` aus (nur wenn `X + OFFSET <> 0`), feuert `EOCS` und kehrt sofort zu `REQ` zurück.

Da sowohl `EICO` als auch `EICS` jederzeit aus `REQ` heraus erreichbar sind, erzwingt die ECC keine Reihenfolge – `EICS` kann auch vor `EICO` ausgelöst werden, liefert dann aber ein inkonsistentes Ergebnis, da `SCALE` auf einem noch nicht kalibrierten `OFFSET` basiert.

Beispiel (4-20-mA-Drucksensor über logiBUS, normiert auf `0.0..1.0`, gewünschter Ausgabebereich `0.0..500.0`):

Schritt	Aktion	Ergebnis
1	4 mA anlegen (`X=0.0`), `Y_Offset=0.0`, `EICO` feuern	`OFFSET = 0`
2	20 mA anlegen (`X=1.0`), `Y_Scale=500.0`, `EICS` feuern	`SCALE = 500`

Ergebnis: `Y = (X + 0) * 500 = X * 500 = 0..500`.

Technische Besonderheiten

- **Y nach CO nur korrekt bei SCALE = 1**: Wie bei `CALIBRATE` liefert `OFFSET := Y_Offset - X` nach der Offset-Kalibrierung nur dann exakt `Y = Y_Offset`, wenn `SCALE` noch `1.0` ist.
- **Keine Reihenfolge-Erzwingung**: `EICO` und `EICS` sind beide direkt aus `REQ` erreichbar – die ECC unterscheidet nicht, ob bereits kalibriert wurde.
- **REQ bleibt jederzeit verfügbar**: Normalbetrieb (`REQ`) ist unabhängig vom Kalibrierstatus jederzeit möglich, auch zwischen `EICO` und `EICS`.

Zustandsübersicht

```
REQ --EICO--> CO --1--> REQ     (Offset-Kalibrierung)
REQ --EICS--> CS --1--> REQ     (Skalierungs-Kalibrierung)
REQ --REQ---> REQ --1--> REQ    (Normalbetrieb)
```

Alle drei Zustände sind Durchgangszustände: Der Baustein kehrt nach jeder Aktion sofort zu `REQ` zurück.

Anwendungsszenarien

- Zwei-Punkt-Kalibrierung analoger Sensoren in Systemen, die Kalibrierauslösung bereits als Ereignis bereitstellen (z. B. per Knopfdruck-Event statt Datenwert)
- Anwendungen, in denen Offset- und Skalierungskalibrierung jeweils eine eigene, sofortige Bestätigung (`EOCO`/`EOCS`) benötigen
- Systeme mit gelegentlicher Nachkalibrierung des Offsets, ohne die Skalierung neu zu bestimmen

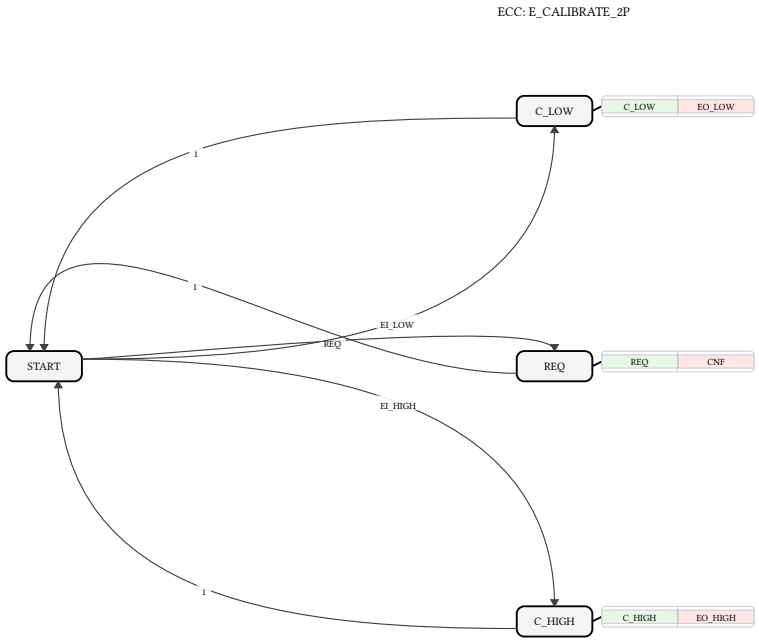
⚖️ Vergleich mit ähnlichen Bausteinen

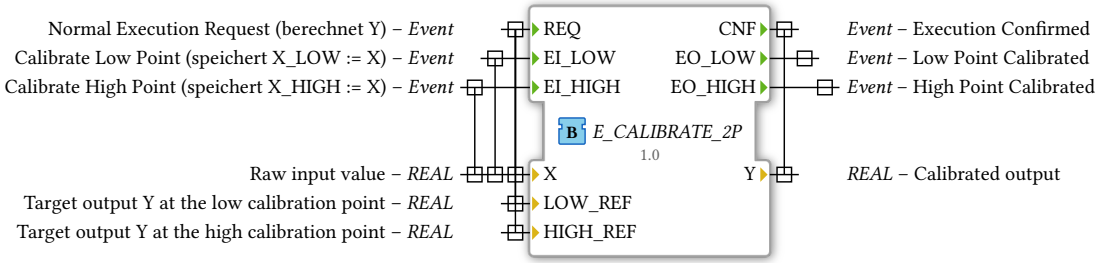
Vergleich mit CALIBRATE, das dieselbe Formel booleangesteuert statt ereignisgesteuert anwendet, Weitere Details zum Verfahren finden Sie unter Kalibrierverfahren Zwei- und Dreipunkt.

Fazit

`E_CALIBRATE` bietet eine ereignisgesteuerte Zwei-Punkt-Kalibrierung mit eigenen Bestätigungsereignissen pro Kalibrierschritt, überlässt die korrekte Reihenfolge (Offset vor Skalierung) aber weiterhin dem Aufrufer.

E_CALIBRATE_2P





• – ◦ • – ◦ • – ◦ •

Einleitung

`E_CALIBRATE_2P` ist eine ereignisgesteuerte Zwei-Punkt-Kalibrierung (Low/High) mit unabhängig gespeicherten Punkten. Er ersetzt E_CALIBRATE (`E_CALIBRATE_SQ`-Verhalten): Die beiden Punkte werden vollkommen unabhängig voneinander gespeichert und erst beim Lesen (`REQ`) zu einer Geraden kombiniert. Es gibt daher keine erzwungene Reihenfolge `EI_LOW`/`EI_HIGH`.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normale Ausführungsanforderung, berechnet `Y`. Liefert `X`, `LOW_REF`, `HIGH_REF`, `X_LOW`, `X_HIGH`.
- **EI_LOW**: Kalibriert den unteren Punkt (`X_LOW := X`). Liefert `X`.
- **EI_HIGH**: Kalibriert den oberen Punkt (`X_HIGH := X`). Liefert `X`.

Ereignis-Ausgänge

- **CNF**: Bestätigt `REQ`, liefert `Y`.
- **EO_LOW**: Unterer Punkt kalibriert, liefert `X_LOW`.
- **EO_HIGH**: Oberer Punkt kalibriert, liefert `X_HIGH`.

Daten-Eingänge

- **X** (REAL): Roheingangswert.
- **LOW_REF** (REAL): Zielausgabewert `Y` am unteren Kalibrierpunkt.

- **HIGH_REF** (REAL): Zielausgabewert `Y` am oberen Kalibrierpunkt.

Daten-Ausgänge

- **Y** (REAL): Kalibrierter Ausgangswert.

Ein-/Ausgabevariablen (InOut)

- **X_LOW** (REAL, Startwert `0.0`): Gespeicherter Rohwert am unteren Kalibrierpunkt.
- **X_HIGH** (REAL, Startwert `1.0`): Gespeicherter Rohwert am oberen Kalibrierpunkt.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Formel: `Y = LOW_REF + (X - X_LOW) * (HIGH_REF - LOW_REF) / (X_HIGH - X_LOW)`
Der Baustein besitzt einen Startzustand `START` sowie drei direkt daraus erreichbare Aktionszustände:

- **REQ**: Berechnet `Y` live per linearer Interpolation zwischen den beiden gespeicherten Punkten (ohne Begrenzung, wie bei `E_CALIBRATE_SQ`) und feuert `CNF`. Ist `X_HIGH - X_LOW = 0`, gilt `Y := LOW_REF`.
- **C_LOW**: Speichert den aktuellen Rohwert in `X_LOW` und feuert `EO_LOW`.
- **C_HIGH**: Speichert den aktuellen Rohwert in `X_HIGH` und feuert `EO_HIGH`.

Jeder Aktionszustand kehrt sofort zu `START` zurück.
Kalibrierung (beliebige Reihenfolge):

1. **Low (`EI_LOW`)**: Unteren Referenzwert anlegen, `LOW_REF` auf den gewünschten Ausgang setzen, `EI_LOW` auslösen ⟶ `X_LOW := X`
2. **High (`EI_HIGH`)**: Oberen Referenzwert anlegen, `HIGH_REF` auf den gewünschten Ausgang setzen, `EI_HIGH` auslösen ⟶ `X_HIGH := X`

Jeder Punkt kann später einzeln neu kalibriert werden, ohne den anderen zu wiederholen.

Technische Besonderheiten

- **Unabhängige Punkte statt inkrementeller OFFSET/SCALE-Berechnung:** `E_CALIBRATE_SQ` berechnet OFFSET/SCALE schrittweise (EICO setzt OFFSET aus einem aktuellen SCALE, EICS berechnet beide aus intern gemerkten Werten) und ist erst nach mindestens einem EICO mathematisch gültig. Das erforderte eine per ECC erzwungene Reihenfolge, um stillschweigend falsche Offsets zu vermeiden. `E_CALIBRATE_2P` speichert `X_LOW`/`X_HIGH` wie `E_CALIBRATE_3P` seine Punkte, ohne Datenabhängigkeit zwischen `EI_LOW` und `EI_HIGH`.
- Keine Begrenzung von `Y` auf `LOW_REF..HIGH_REF` (anders als `E_CALIBRATE_3P`).
- Entartete Kalibrierpunkte (`X_HIGH = X_LOW`) liefern `LOW_REF` statt einer Division durch Null.

Zustandsübersicht

```
START --REQ-----> REQ    --1--> START   (Normalbetrieb)
START --EI_LOW--> C_LOW  --1--> START   (unteren Punkt kalibrieren)
START --EI_HIGH-> C_HIGH --1--> START   (oberen Punkt kalibrieren)
```

Anwendungsszenarien

- Zwei-Punkt-Kalibrierung von Analogeingängen (z. B. Sensoren), wenn die Kalibrierauslösung bereits als Ereignis vorliegt
- Nachjustage nur eines Punktes

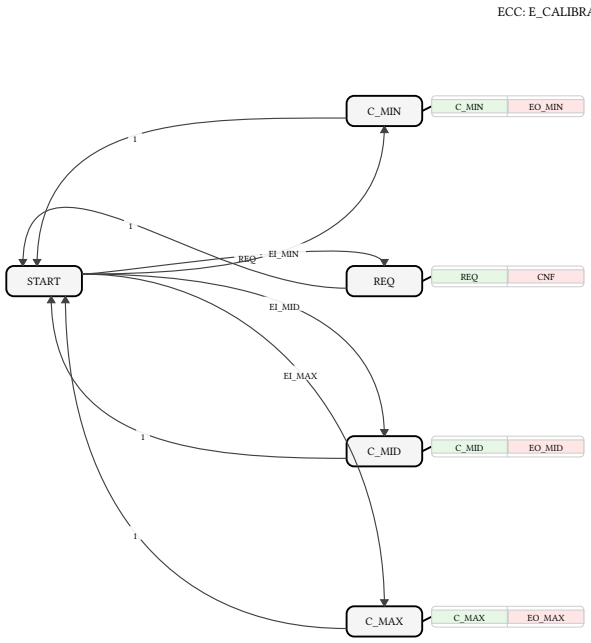
⚖️ Vergleich mit ähnlichen Bausteinen

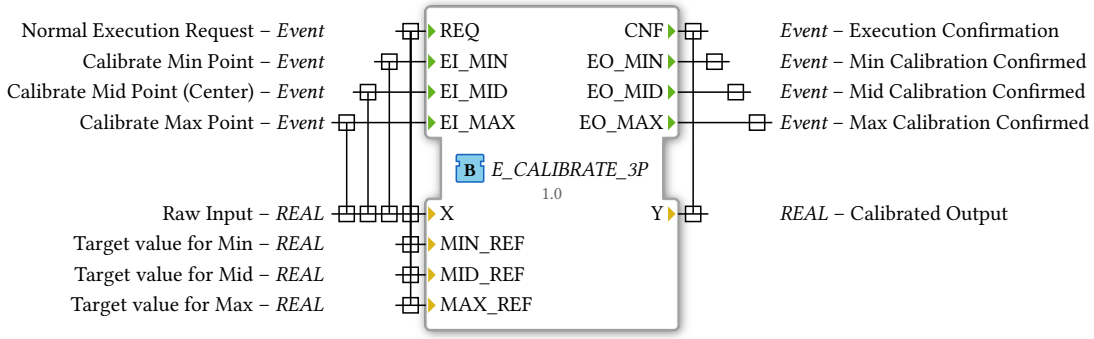
- E_CALIBRATE: Variante mit `EICO`/`EICS` (`OFFSET`/`SCALE`), bei der `EICS` nur nach `EICO` ein konsistentes Ergebnis liefert
- E_CALIBRATE_3P: Drei-Punkt-Variante mit stückweiser Interpolation und Begrenzung

Fazit

`E_CALIBRATE_2P` bietet eine ereignisgesteuerte Zwei-Punkt-Kalibrierung mit unabhängig ansteuerbaren Kalibrierpunkten und ohne erzwungene Reihenfolge.

E_CALIBRATE_3P





• – ○ • – ○ • – ○ •

Einleitung

`E_CALIBRATE_3P` ist die ereignisgesteuerte Variante von CALIBRATE_3P: Die Drei-Punkt-Kalibrierung mit stückweiser linearer Interpolation (Min/Mid/Max) wird über die eigenen Ereignisse `EI_MIN`/`EI_MID`/`EI_MAX` ausgelöst, jeweils mit eigenem Bestätigungsereignis.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normale Ausführungsanforderung – berechnet `Y`. Liefert `X`, `MIN_REF`, `MID_REF`, `MAX_REF`, `X_MIN`, `X_MID`, `X_MAX`.
- **EI_MIN**: Kalibriert den Minimalpunkt (`X_MIN := X`). Liefert `X`.
- **EI_MID**: Kalibriert den Mittelpunkt (`X_MID := X`). Liefert `X`.
- **EI_MAX**: Kalibriert den Maximalpunkt (`X_MAX := X`). Liefert `X`.

Ereignis-Ausgänge

- **CNF**: Bestätigt `REQ`, liefert `Y`.
- **EO_MIN**: Bestätigt die Minimalpunkt-Kalibrierung, liefert `X_MIN`.
- **EO_MID**: Bestätigt die Mittelpunkt-Kalibrierung, liefert `X_MID`.
- **EO_MAX**: Bestätigt die Maximalpunkt-Kalibrierung, liefert `X_MAX`.

Daten-Eingänge

- **X** (REAL): Roheingangswert.
- **MIN_REF** (REAL, Startwert `0.0`): Zielausgabewert für den Minimalpunkt.
- **MID_REF** (REAL, Startwert `50.0`): Zielausgabewert für den Mittelpunkt.
- **MAX_REF** (REAL, Startwert `100.0`): Zielausgabewert für den Maximalpunkt.

Daten-Ausgänge

- **Y** (REAL): Kalibrierter, interpolierter und auf `MIN_REF..MAX_REF` begrenzter Ausgangswert.

Ein-/Ausgabevariablen (InOut)

- **X_MIN** (REAL, Startwert `0.0`): Gespeicherter Rohwert am Minimalpunkt.
- **X_MID** (REAL, Startwert `50.0`): Gespeicherter Rohwert am Mittelpunkt.
- **X_MAX** (REAL, Startwert `100.0`): Gespeicherter Rohwert am Maximalpunkt.

Funktionsweise

`E_CALIBRATE_3P` besitzt einen Startzustand `START` sowie vier direkt daraus erreichbare Aktionszustände:

- **REQ**: Interpoliert `Y` stückweise linear zwischen den gespeicherten Punkten (wie bei CALIBRATE_3P) und feuert `CNF`.
- **C_MIN/C_MID/C_MAX**: Speichern den aktuellen Rohwert `X` im jeweiligen `X_MIN/X_MID/X_MAX` und feuern das zugehörige Bestätigungsereignis (`EO_MIN/EO_MID/EO_MAX`).

Jeder dieser vier Zustände kehrt unmittelbar zu `START` zurück, von wo aus alle vier Ereignisse wieder gleichberechtigt erreichbar sind – die Kalibrierpunkte können also in beliebiger Reihenfolge und beliebig oft einzeln angefahren werden.

Beispiel (Joystick mit Mittelpunkt-Drift, Rohbereich `0..1000`, gewünschter Ausgabebereich `-100..0..100`):

Schritt	Aktion	Ergebnis
1	Joystick auf **Minimum**, `EI_MIN` feuern	`X_MIN = 50` (Drift gegenüber ideal `0`)
2	Joystick auf **Mitte**, `EI_MID` feuern	`X_MID = 520` (Drift gegenüber ideal `500`)
3	Joystick auf **Maximum**, `EI_MAX` feuern	`X_MAX = 980` (Drift gegenüber ideal `1000`)

Ergebnis: `Y` wird zwischen `MIN_REF=-100`, `MID_REF=0` und `MAX_REF=100` interpoliert und begrenzt.

Technische Besonderheiten

- **Vier gleichberechtigte Aktionen aus einem gemeinsamen Startzustand**: Es gibt keine Abhängigkeit zwischen den Kalibrierschritten – jeder Punkt kann unabhängig und in beliebiger Reihenfolge neu gesetzt werden.
- **Ausgabe stets begrenzt**: `Y` wird nach der Interpolation auf `MIN_REF..MAX_REF` geklippt.
- **Degenerierte Kalibrierpunkte**: Wie bei `CALIBRATE_3P` liefert eine Interpolation mit `X_MID <= X_MIN` bzw. `X_MAX <= X_MID` konstant `MIN_REF` bzw. `MID_REF` statt einer Division durch Null.

Zustandsübersicht

```
START --REQ-----> REQ    --1--> START   (Normalbetrieb)
START --EI_MIN--> C_MIN  --1--> START   (Minimalpunkt kalibrieren)
START --EI_MID--> C_MID  --1--> START   (Mittelpunkt kalibrieren)
START --EI_MAX--> C_MAX  --1--> START   (Maximalpunkt kalibrieren)
```

Alle vier Kalibrierereignisse sind unabhängig voneinander aus `START` erreichbar – keine erzwungene Reihenfolge.

Anwendungsszenarien

- Drei-Punkt-Kalibrierung von Joysticks in Systemen, die Kalibrierauslösung bereits als Ereignis bereitstellen
- Anwendungen, in denen jeder Kalibrierpunkt eine eigene, sofortige Bestätigung benötigt (z. B. zur UI-Rückmeldung „Punkt X kalibriert“)
- Nachjustage einzelner Punkte (z. B. nur `EI_MID` nach Drift), ohne die anderen beiden neu kalibrieren zu müssen

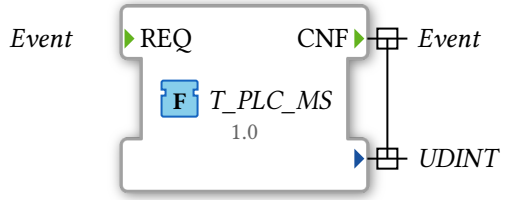
⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit CALIBRATE_3P, das dieselbe Drei-Punkt-Logik booleangesteuert statt ereignisgesteuert anwendet, sowie mit E_CALIBRATE, das eine lineare Zwei-Punkt- statt Drei-Punkt-Kalibrierung durchführt. Details siehe Kalibrierverfahren Zwei- und Dreipunkt.

Fazit

`E_CALIBRATE_3P` bietet eine ereignisgesteuerte Drei-Punkt-Kalibrierung mit unabhängig ansteuerbaren, eigenständig bestätigten Kalibrierpunkten und eignet sich besonders für Joysticks und ähnliche Bedienelemente mit Mittelstellung.

T_PLC_MS



T_PLC_MS reads the internal PLC timer and return the time

Interface

Event Inputs

Name	Comment	With
REQ		

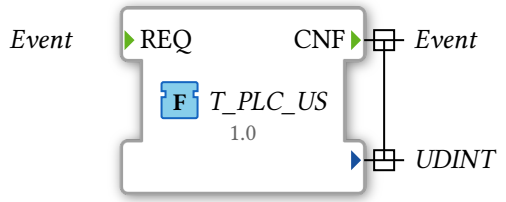
Event Outputs

Name	Comment	With
CNF		

Output Vars

Name	Type	Comment
	UDINT	

T_PLC_US



T_PLC_US reads the internal PLC timer and return the time

Interface

Event Inputs

Name	Comment	With
REQ		

Event Outputs

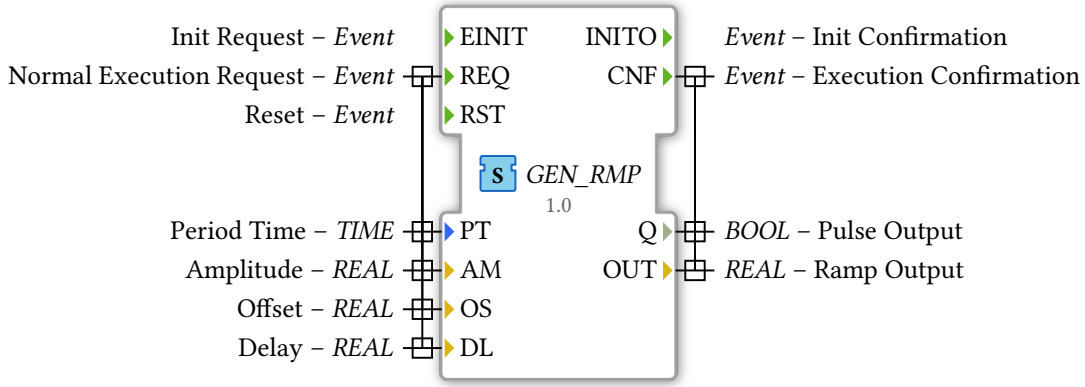
Name	Comment	With
CNF		

Output Vars

Name	Type	Comment
	UDINT	

signal_generators

GEN_RMP



this signal generator generates a ramp wave output

this signal generator generates a ramp wave output. The ramp wave signal is defined by period time (PT), amplitude (AM), offset (OS) and a specific delay for the output signal (DL). The Output waveform will have its minimum peak at OS and its maximum peak at AM + OS. The delay input can delay a signal up to PT, this can be useful to synchronize different generators and generate interleaving signals. in addition to the analog output Out there is a second boolean output Q with is true for one cycle when the ramp starts.

Interface

Event Inputs

Name	Comment	With
EINIT	Init Request	
REQ	Normal Execution Request	PT, AM, OS, DL
RST	Reset	

Event Outputs

Name	Comment	With
INITO	Init Confirmation	
CNF	Execution Confirmation	Q, OUT

Input Vars

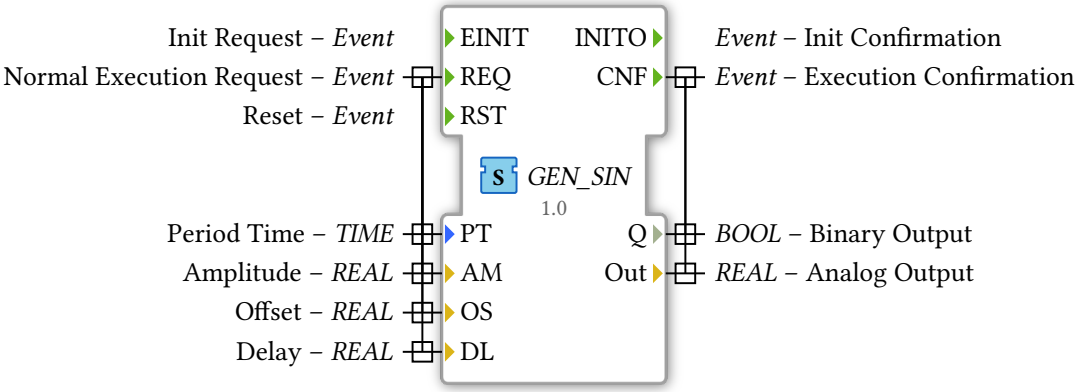
Name	Type	Initial Value	Comment
PT	TIME	T#1s	Period Time
AM	REAL	1.0	Amplitude

OS	REAL		Offset
DL	REAL		Delay

Output Vars

Name	Type	Comment
Q	BOOL	Pulse Output
OUT	REAL	Ramp Output

GEN_SIN



this signal generator generates a sine wave output

this signal generator generates a sine wave output. The sine wave signal is defined by period time (PT), amplitude (AM), offset (OS) and a specific delay for the output signal (DL). The Output waveform will have its max peak at AM/2 + OS and its minimum peak at -AM/2 + OS. The delay input can delay a signal up to PT, this can be useful to synchronize different generators and generate interleaving signals. A Cos wave can be generated accordingly. in addition to a analog output Out there is a second boolean output Q with the corresponding binary signal.

Interface

Event Inputs

Name	Comment	With
EINIT	Init Request	
REQ	Normal Execution Request	PT, AM, OS, DL
RST	Reset	

Event Outputs

Name	Comment	With
INITO	Init Confirmation	
CNF	Execution Confirmation	Q, Out

Input Vars

Name	Type	Initial Value	Comment
PT	TIME		Period Time
AM	REAL	1.0	Amplitude

OS	REAL		Offset
DL	REAL		Delay

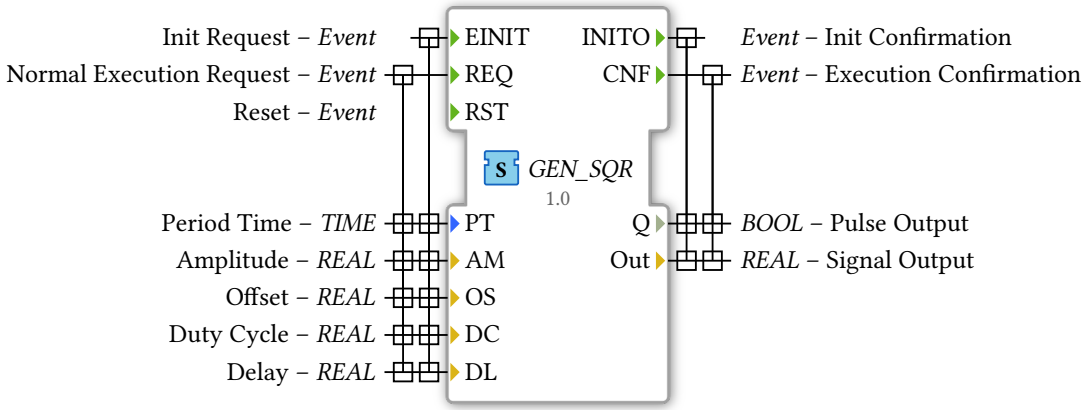
Output Vars

Name	Type	Comment
Q	BOOL	Binary Output
Out	REAL	Analog Output

🛠️ Zugehörige Übungen

- SIN_TO_PCAN_Callback_DWord
- SIN_TO_PCAN_Callback_Byte

GEN_SQR



this signal generator generates a square wave output

this signal generator generates a square wave output. The square wave signal is defined by period time (PT), amplitude (AM), offset (OS), duty cycle (DC) and a specific delay for the output signal (DL). The Output waveform will switch between AM/2 + OS and -AM/2 + OS. The DC input specifies ther duty cycle, DC = 0 means output is low at all times and 1 means output is high at all times. The delay input can delay a signal up to PT, this can be useful to synchronize different generators and generate interleaving signals. in addition to a analog output Out there is a second boolean output Q.

Interface

Event Inputs

Name	Comment	With
EINIT	Init Request	PT, AM, OS, DC, DL
REQ	Normal Execution Request	PT, AM, OS, DC, DL
RST	Reset	

Event Outputs

Name	Comment	With
INITO	Init Confirmation	Q, Out
CNF	Execution Confirmation	Q, Out

Input Vars

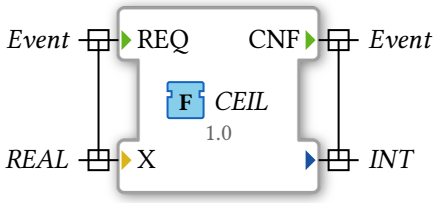
Name	Type	Initial Value	Comment
PT	TIME		Period Time
AM	REAL	1.0	Amplitude
OS	REAL		Offset
DC	REAL	0.5	Duty Cycle
DL	REAL		Delay

Output Vars

Name	Type	Comment
Q	BOOL	Pulse Output
Out	REAL	Signal Output

Mathematical

CEIL



• – ◦ • – ◦ • – ◦ •

Einleitung

Die Funktion `CEIL` ist eine Rundungsfunktion (portiert aus `OSCAT_basic`, `CEIL.TcPOU`), die die kleinste ganze Zahl zurückgibt, die größer oder gleich `X` ist. Das Ergebnis wird auf den `INT`-Bereich begrenzt.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Startet die Berechnung. Verbundene Daten: `X`

Ereignis-Ausgänge

- `CNF`: Signalisiert die Berechnung und die Ausgabe des Ergebnisses.

Daten-Eingänge

- `X` (`REAL`): Zu rundender Wert

Daten-Ausgänge

- (kein Name) (`INT`): Aufgerundeter Wert, zugleich der Rückgabewert der Funktion

Adapter

Diese Funktion verwendet keine Adapter.

Funktionsweise

1. Ist `X >= 32767.0`, ist das Ergebnis `32767`.
2. Ist `X <= -32768.0`, ist das Ergebnis `-32768`.
3. Andernfalls wird `X` mit `REAL_TO_INT` in eine ganze Zahl umgewandelt. Ist das Ergebnis kleiner als `X`, wird 1 addiert.

Beispiele: `CEIL(3.14) = 4`, `CEIL(-3.14) = -3`.

Technische Besonderheiten

- Implementierung in Structured Text (ST) als `FUNCTION` ohne internen Zustand
- Teil des Pakets `OSCAT::Basic::POUs::Mathematical`
- Die Bereichsprüfung schützt vor einem `INT`-Überlauf (Portierung vom 2026-09-17).
- `REAL_TO_INT` wird statt `TRUNC` verwendet (Kompatibilität, Revision 1.1 des OSCAT-Originals).

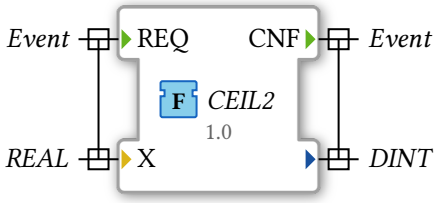
Anwendungsszenarien

- Aufrunden von Zahlenwerten, z. B. für Anzeige- oder Zählwerte

Fazit

`CEIL` liefert die nächstgrößere ganze Zahl zu einem `REAL`-Wert als `INT` und begrenzt das Ergebnis auf den `INT`-Wertebereich.

CEIL2



• – ◦ • – ◦ • – ◦ •

Einleitung

`CEIL2` ist eine Rundungsfunktion aus der OSCAT-Basic-Bibliothek: Sie liefert die kleinste ganze Zahl, die größer oder gleich `X` ist (Aufrunden). Gegenüber `CEIL` ist das Ergebnis ein `DINT`, die Funktion arbeitet also auch außerhalb des `INT`-Bereichs. Beispiele: `CEIL2(3.14) = 4`, `CEIL2(-3.14) = -3`. Portiert aus `OSCAT_basic CEIL2.TcPOU`.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Berechnungsanforderung (assoziiert mit `X`)

Ereignis-Ausgänge

- **CNF**: Berechnungsbestätigung (assoziiert mit dem Funktionsergebnis)

Daten-Eingänge

- **X** (REAL): zu rundender Wert

Daten-Ausgänge

- **(Funktionsergebnis)** (DINT): kleinste ganze Zahl `>= X`, auf den `DINT`-Bereich begrenzt

Adapter

Es werden keine Adapter verwendet.

Funktionsweise

Die Funktion ist in Structured Text implementiert:

1. Ist `X >= 2147483647.0`, wird `2147483647` zurückgegeben.
2. Ist `X <= -2147483648.0`, wird `-2147483648` zurückgegeben.
3. Sonst wird `CEIL2 := REAL_TO_DINT(X)` berechnet; ist `DINT_TO_REAL(CEIL2) < X`, wird um 1 erhöht (`CEIL2 := CEIL2 + 1`).

Technische Besonderheiten

- Die Bereichsprüfung schützt vor `DINT`-Überlauf: Werte außerhalb des `DINT`-Bereichs werden auf die Grenzwerte begrenzt

- Reine Funktion ohne Zustand; als Function-Baustein mit `REQ`/`CNF` in das Ereignis-Netzwerk eingebunden
- Der Ausgang besitzt keinen eigenen Namen, sondern ist der Rückgabewert der Funktion

Anwendungsszenarien

- Aufrunden eines Messwerts oder einer berechneten Größe auf die nächste ganze Zahl, auch bei großen Beträgen

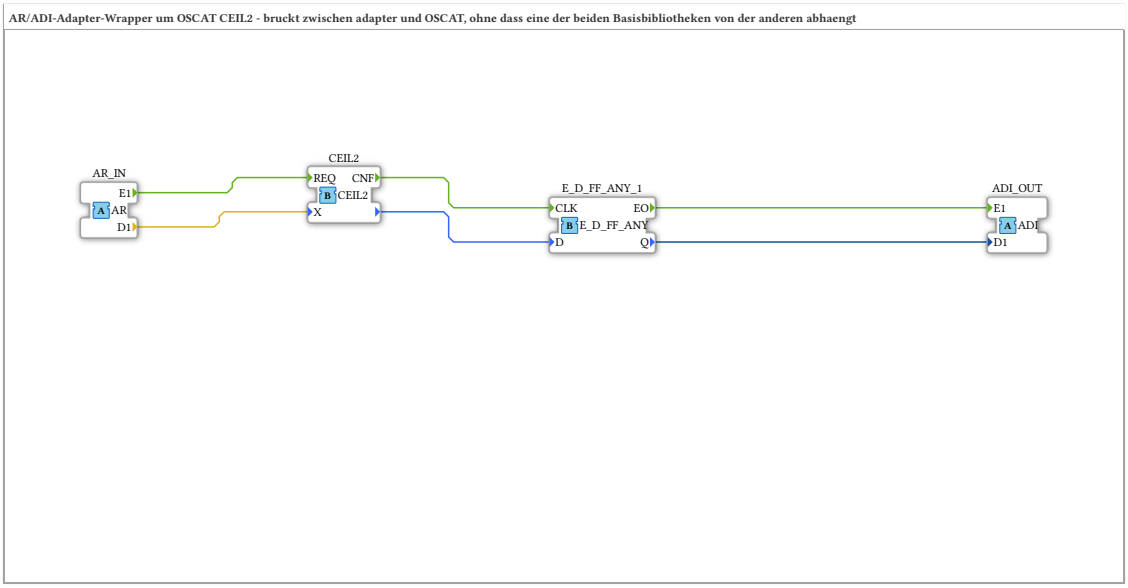
⚖️ Vergleich mit ähnlichen Bausteinen

`FLOOR2` ist das Gegenstück zum Abrunden (größte ganze Zahl `<= X`).

Fazit

`CEIL2` rundet einen `REAL`-Wert auf die nächste ganze Zahl auf und liefert das Ergebnis als `DINT` mit Schutz vor Überlauf.

CEIL2_AR

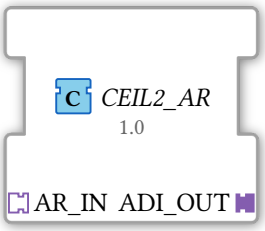


• – ○ • – ○ • – ○ •

Einleitung

`CEIL2_AR` ist ein Adapter-Wrapper um `OSCAT::Basic::POUs::Mathematical::CEIL2` (DINT-Variante von CEIL für Werte außerhalb des INT-Bereichs). Er verbindet den `AR`-Adapter mit der OSCAT-Funktion `CEIL2` und gibt das Ergebnis als `ADI`-Adapter aus (kleinste ganze Zahl größer oder gleich X (DINT)). Der Wrapper überbrückt Adapter- und OSCAT-Bibliothek, ohne dass eine der beiden Basisbibliotheken von der anderen abhängt.

Eingangssignal (REAL) – *AR*



ADI – Kleinste ganze Zahl >= X (DINT)

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge – Ereignisse kommen über den Adapter `AR_IN`.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge – Ereignisse gehen über den Adapter `ADI_OUT`.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

- **AR_IN** (Socket, `adapter::types::unidirectional::AR`): Eingangssignal (`REAL`)
- **ADI_OUT** (Plug, `adapter::types::unidirectional::ADI`): kleinste ganze Zahl größer oder gleich X (DINT)

Verwendete Funktionsbausteine (FBs)

- **`CEIL2`** (`OSCAT::Basic::POUs::Mathematical::CEIL2`): Rundungsfunktion, siehe CEIL2
- **`E_D_FF_ANY_1`** (`iec61499::events::E_D_FF_ANY`): Änderungsfilter

Funktionsweise

- `AR_IN.E1` ⟶ `CEIL2.REQ`; `AR_IN.D1` ⟶ `CEIL2.X`
- `CEIL2.CNF` ⟶ `E_D_FF_ANY_1.CLK`; Rückgabewert von `CEIL2` ⟶ `E_D_FF_ANY_1.D`
- `E_D_FF_ANY_1.EO` ⟶ `ADI_OUT.E1`; `E_D_FF_ANY_1.Q` ⟶ `ADI_OUT.D1`

Technische Besonderheiten

- `E_D_FF_ANY_1` (Standardbibliothek `events-3.0.0`) filtert Änderungen: `ADI_OUT.E1` feuert nur, wenn sich das Ergebnis seit dem letzten Aufruf tatsächlich geändert hat. Gleiches Muster wie bei den Wrappern `FT_DERIV_AR`/`CTRL_PI_AR`.
- **Warum ein eigener Wrapper?** `CEIL2` ist eine reine OSCAT-`Function` ohne Adapter-Schnittstelle. `adapter` soll nicht von `OSCAT` abhängen (und umgekehrt); die Brücke liegt daher im dritten Bibliotheksprojekt `OSCAT_adapter`, das von beiden abhängt.

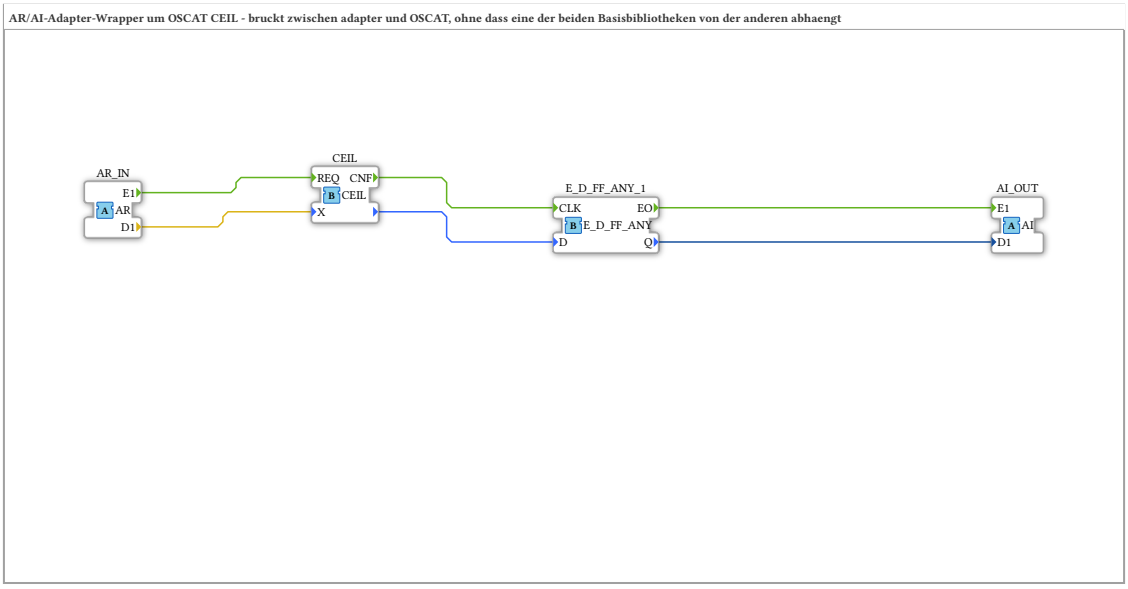
Anwendungsszenarien

- Runden eines `AR`-Signals in einer Adapter-Kette auf eine ganze Zahl (`DINT`)

Zusammenfassung

`CEIL2_AR` rundet einen `AR`-Wert mit `CEIL2` und gibt das Ergebnis nur bei Änderung als `ADI`-Adapter aus.

CEIL_AR

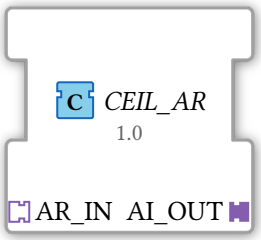


• – ◦ • – ◦ • – ◦ •

Einleitung

`CEIL_AR` ist ein Adapter-Wrapper um `OSCAT::Basic::POUs::Mathematical::CEIL`. Er verbindet den `AR`-Adapter mit der OSCAT-Funktion `CEIL` und gibt das Ergebnis als `AI`-Adapter aus (kleinste ganze Zahl größer oder gleich X (INT)). Der Wrapper überbrückt Adapter- und OSCAT-Bibliothek, ohne dass eine der beiden Basisbibliotheken von der anderen abhängt.

Eingangssignal (REAL) – *AR*



AI – Kleinste ganze Zahl >= X (INT)

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge – Ereignisse kommen über den Adapter `AR_IN`.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge – Ereignisse gehen über den Adapter `AI_OUT`.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

- **AR_IN** (Socket, `adapter::types::unidirectional::AR`): Eingangssignal (`REAL`)
- **AI_OUT** (Plug, `adapter::types::unidirectional::AI`): kleinste ganze Zahl größer oder gleich X (INT)

Verwendete Funktionsbausteine (FBs)

- **`CEIL`** (`OSCAT::Basic::POUs::Mathematical::CEIL`): Rundungsfunktion, siehe CEIL
- **`E_D_FF_ANY_1`** (`iec61499::events::E_D_FF_ANY`): Änderungsfilter

Funktionsweise

- `AR_IN.E1` ⟶ `CEIL.REQ`; `AR_IN.D1` ⟶ `CEIL.X`
- `CEIL.CNF` ⟶ `E_D_FF_ANY_1.CLK`; Rückgabewert von `CEIL` ⟶ `E_D_FF_ANY_1.D`
- `E_D_FF_ANY_1.EO` ⟶ `AI_OUT.E1`; `E_D_FF_ANY_1.Q` ⟶ `AI_OUT.D1`

Technische Besonderheiten

- `E_D_FF_ANY_1` (Standardbibliothek `events-3.0.0`) filtert Änderungen: `AI_OUT.E1` feuert nur, wenn sich das Ergebnis seit dem letzten Aufruf tatsächlich geändert hat. Gleiches Muster wie bei den Wrappern `FT_DERIV_AR`/`CTRL_PI_AR`.
- **Warum ein eigener Wrapper?** `CEIL` ist eine reine OSCAT-`Function` ohne Adapter-Schnittstelle. `adapter` soll nicht von `OSCAT` abhängen (und umgekehrt); die Brücke liegt daher im dritten Bibliotheksprojekt `OSCAT_adapter`, das von beiden abhängt.

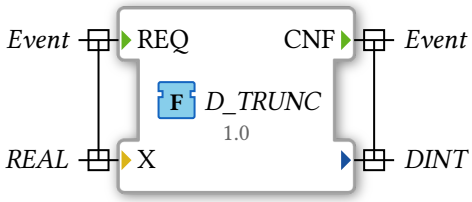
Anwendungsszenarien

- Runden eines `AR`-Signals in einer Adapter-Kette auf eine ganze Zahl (`INT`)

Zusammenfassung

`CEIL_AR` rundet einen `AR`-Wert mit `CEIL` und gibt das Ergebnis nur bei Änderung als `AI`-Adapter aus.

D_TRUNC



d_trunc truncates a real to a dint 1.5 will be 1 and −1.5 will be −1

Interface

Event Inputs

Name	Comment	With
REQ		X

Event Outputs

Name	Comment	With
CNF		

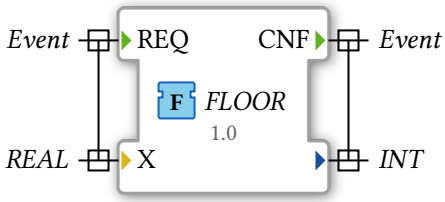
Input Vars

Name	Type	Comment
X	REAL	

Output Vars

Name	Type	Comment
	DINT	

FLOOR



🎧 Podcast

- Unlocking Universal Automation: The IEC 61499 Revolution from Factory Floors to the Seas

Interface

Event Inputs

Name	Comment	With
REQ		X

Event Outputs

Name	Comment	With

CNF		

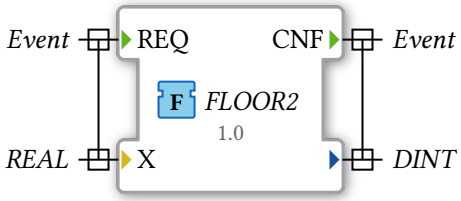
Input Vars

Name	Type	Comment
X	REAL	

Output Vars

Name	Type	Comment
	INT	

FLOOR2



This is a rounding function which returns the biggest possible integer which is less or equal to X.

Interface

Event Inputs

Name	Comment	With
REQ		X

Event Outputs

Name	Comment	With
CNF		

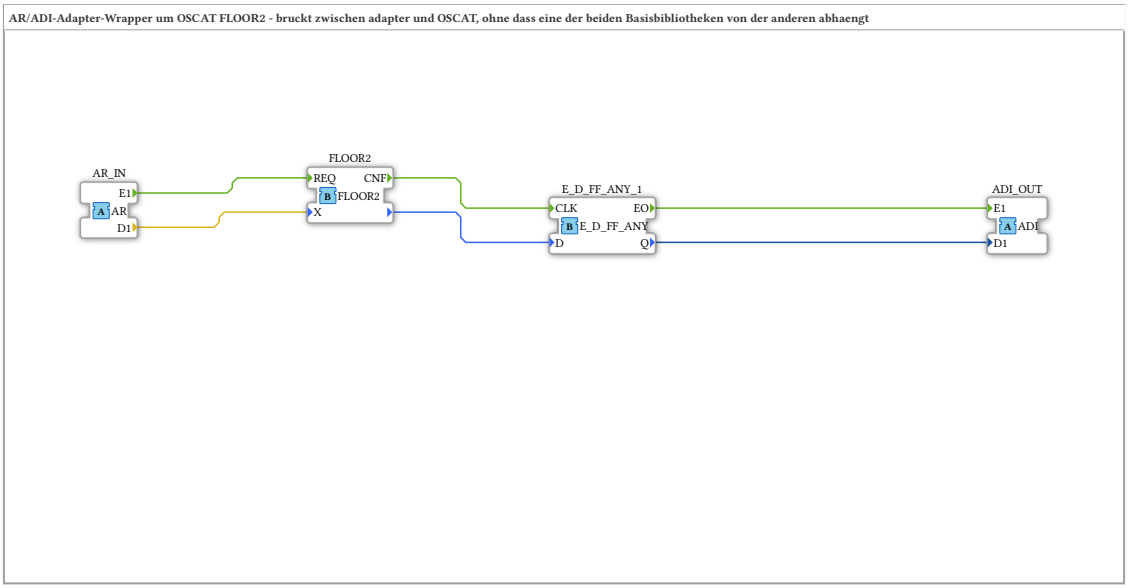
Input Vars

Name	Type	Comment
X	REAL	

Output Vars

Name	Type	Comment
	DINT	

FLOOR2_AR

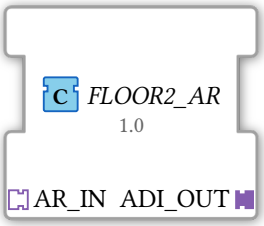


• – ○ • – ○ • – ○ •

Einleitung

`FLOOR2_AR` ist ein AR/ADI-Adapter-Wrapper um die OSCAT-Funktion `OSCAT::Basic::POUs::Mathematical::FLOOR2` (größte ganze Zahl `<=` `X`, `DINT`-Variante von `FLOOR` für Werte außerhalb des `INT`-Bereichs). Er überbrückt zwischen Adaptern und OSCAT, ohne dass eine der beiden Basisbibliotheken von der anderen abhängt.

Eingangssignal (REAL) – *AR*



ADI – Groesste ganze Zahl <= X (DINT)

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **AR_IN** (Socket, `adapter::types::unidirectional::AR`): Eingangssignal (REAL)
- **ADI_OUT** (Plug, `adapter::types::unidirectional::ADI`): größte ganze Zahl `<=` `X` (DINT)

Funktionsweise

- `AR_IN.E1` ⟶ `FLOOR2.REQ` und `AR_IN.D1` ⟶ `FLOOR2.X`: Jedes neue Eingangssignal startet die Berechnung.
- `FLOOR2.CNF` ⟶ `E_D_FF_ANY_1.CLK`; das Funktionsergebnis geht an `E_D_FF_ANY_1.D`.

- `E_D_FF_ANY_1` (`iec61499::events::E_D_FF_ANY`) filtert Änderungen: `E_D_FF_ANY_1.EO` ⟶ `ADI_OUT.E1` und `E_D_FF_ANY_1.Q` ⟶ `ADI_OUT.D1`, aber nur, wenn sich das Ergebnis seit dem letzten Aufruf tatsächlich geändert hat.

Technische Besonderheiten

- Änderungsfilter: `ADI_OUT.E1` feuert nur bei tatsächlicher Wertänderung (gleiches Muster wie bei `CTRL_PI_AR`/`FT_DERIV_AR`)
- `FLOOR2` ist eine reine OSCAT-Funktion ohne Adapter-Schnittstelle; `adapter` soll nicht von `OSCAT` abhängen (und umgekehrt), die Brücke liegt daher im dritten Bibliotheksprojekt `OSCAT_adapter`

Anwendungsszenarien

- Abrunden eines adapterbasierten REAL-Werts auf eine ganze Zahl (DINT) in einem Adapter-Netzwerk

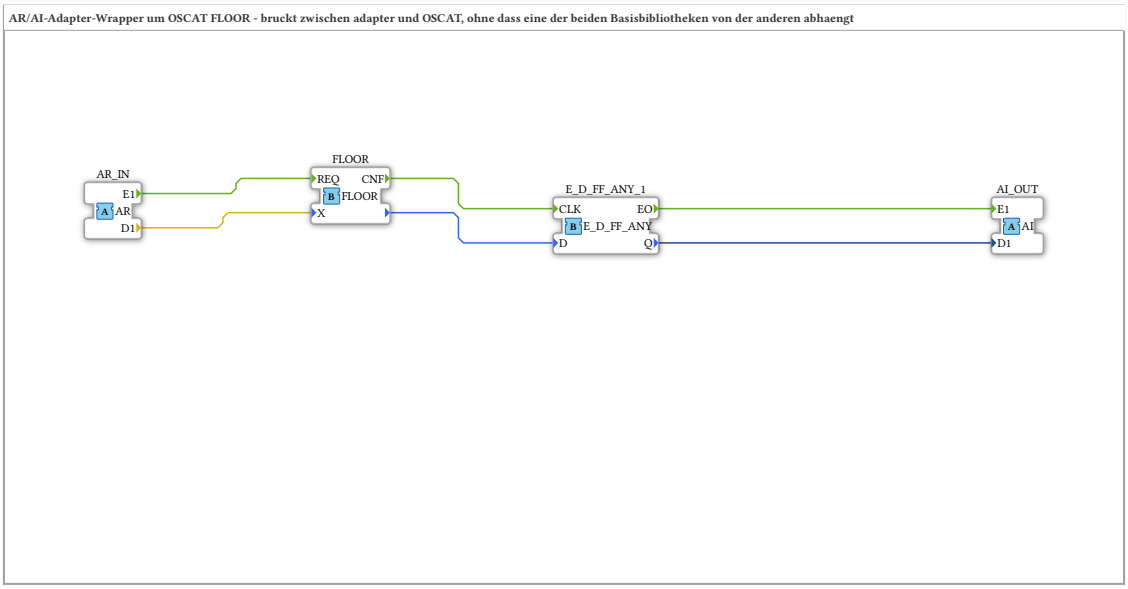
⚖️ Vergleich mit ähnlichen Bausteinen

`ROUND_AR` rundet auf Nachkommastellen (REAL-Ausgang); `FLOOR2_AR` liefert die abgerundete ganze Zahl als DINT.

Fazit

`FLOOR2_AR` stellt `FLOOR2` als Adapter-Baustein von `AR` nach `ADI` bereit und gibt das Ergebnis nur bei Änderung aus.

FLOOR_AR

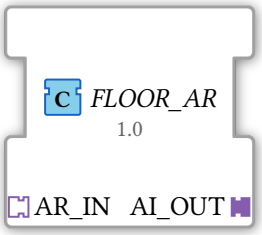


• – ◦ • – ◦ • – ◦ •

Einleitung

`FLOOR_AR` ist ein Adapter-Wrapper um `OSCAT::Basic::POUs::Mathematical::FLOOR`. Er verbindet den `AR`-Adapter mit der OSCAT-Funktion `FLOOR` und gibt das Ergebnis als `AI`-Adapter aus (größte ganze Zahl kleiner oder gleich X (INT)). Der Wrapper überbrückt Adapter- und OSCAT-Bibliothek, ohne dass eine der beiden Basisbibliotheken von der anderen abhängt.

Eingangssignal (REAL) – *AR*



AI – Groesste ganze Zahl <= X (INT)

Schnittstellenstruktur

Ereignis-Eingänge
Keine direkten Ereignis-Eingänge – Ereignisse kommen über den Adapter `AR_IN`.

Ereignis-Ausgänge
Keine direkten Ereignis-Ausgänge – Ereignisse gehen über den Adapter `AI_OUT`.

Daten-Eingänge
Keine.

Daten-Ausgänge
Keine.

Adapter
- **AR_IN** (Socket, `adapter::types::unidirectional::AR`): Eingangssignal (`REAL`)
- **AI_OUT** (Plug, `adapter::types::unidirectional::AI`): größte ganze Zahl kleiner oder gleich X (INT)

Verwendete Funktionsbausteine (FBs)
- **`FLOOR`** (`OSCAT::Basic::POUs::Mathematical::FLOOR`): Rundungsfunktion, siehe FLOOR
- **`E_D_FF_ANY_1`** (`iec61499::events::E_D_FF_ANY`): Änderungsfilter

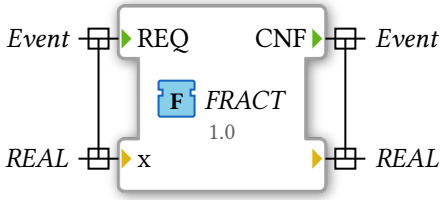
Funktionsweise
- `AR_IN.E1` ⟶ `FLOOR.REQ`; `AR_IN.D1` ⟶ `FLOOR.X`
- `FLOOR.CNF` ⟶ `E_D_FF_ANY_1.CLK`; Rückgabewert von `FLOOR` ⟶ `E_D_FF_ANY_1.D`
- `E_D_FF_ANY_1.EO` ⟶ `AI_OUT.E1`; `E_D_FF_ANY_1.Q` ⟶ `AI_OUT.D1`

Technische Besonderheiten
- `E_D_FF_ANY_1` (Standardbibliothek `events-3.0.0`) filtert Änderungen: `AI_OUT.E1` feuert nur, wenn sich das Ergebnis seit dem letzten Aufruf tatsächlich geändert hat. Gleiches Muster wie bei den Wrappern `FT_DERIV_AR`/`CTRL_PI_AR`.
- **Warum ein eigener Wrapper?** `FLOOR` ist eine reine OSCAT-`Function` ohne Adapter-Schnittstelle. `adapter` soll nicht von `OSCAT` abhängen (und umgekehrt); die Brücke liegt daher im dritten Bibliotheksprojekt `OSCAT_adapter`, das von beiden abhängt.

Anwendungsszenarien
- Runden eines `AR`-Signals in einer Adapter-Kette auf eine ganze Zahl (`INT`)

Zusammenfassung
`FLOOR_AR` rundet einen `AR`-Wert mit `FLOOR` und gibt das Ergebnis nur bei Änderung als `AI`-Adapter aus.

FRACT



this function returns the fraction of a real number

Interface

Event Inputs

Name	Comment	With
REQ		x

Event Outputs

Name	Comment	With
CNF		

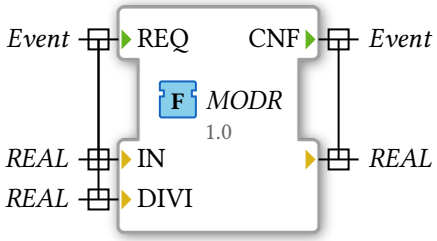
Input Vars

Name	Type	Comment
x	REAL	

Output Vars

Name	Type	Comment
	REAL	

MODR



this is a modulo function for real numbers

Interface

Event Inputs

Name	Comment	With
REQ		IN, DIVI

Event Outputs

Name	Comment	With
CNF		

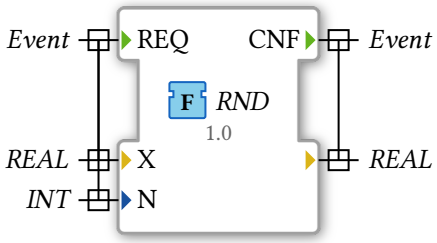
Input Vars

Name	Type	Comment
IN	REAL	
DIVI	REAL	

Output Vars

Name	Type	Comment
	REAL	

RND



• – ◦ • – ◦ • – ◦ •

Einleitung

Die Funktion `RND` rundet eine `REAL`-Zahl auf `N` **signifikante** Stellen (nicht auf Nachkommastellen). Sie ist aus `OSCAT_basic` portiert. Anders als `ROUND` bezeichnet `N` hier die Gesamtzahl der signifikanten Stellen.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Berechnung. Verbundene Daten: `X`, `N`.

Ereignis-Ausgänge

- **CNF**: Signalisiert die Berechnung und die Ausgabe des Ergebnisses.

Daten-Eingänge

- **X** (`REAL`): Zu rundender Wert
- **N** (`INT`): Anzahl der signifikanten Stellen

Daten-Ausgänge

- **(unbenannt)** (`REAL`): Gerundeter Wert, zugleich Rückgabewert der Funktion

Adapter

Diese Funktion verwendet keine Adapter.

Funktionsweise

1. Ist `X = 0.0`, ist das Ergebnis `0.0`.
2. Andernfalls wird der Faktor `M = 10^(LIMIT(1, N, 8) - CEIL(LOG(ABS(X))))` berechnet; der Exponent wird auf `-37..8` begrenzt.
3. Ist `ABS(X * M) >= 2147483647.0`, wird `X` unverändert zurückgegeben (Schutz vor `DINT`-Überlauf).
4. Sonst ist das Ergebnis `DINT_TO_REAL(REAL_TO_DINT(X * M)) / M`.

Technische Besonderheiten

- Implementierung in Structured Text (ST) als `FUNCTION` ohne internen Zustand, Paket `OSCAT::Basic::POUs::Mathematical`
- `N` wird auf `1..8` begrenzt
- Portierung vom 2026-09-17: Begrenzung von `N` (1..8), des Exponenten (-37..8) und eine Bereichsprüfung schützen vor `DINT`-Überlauf
- Originalversion 1.1 (4. Apr. 2008, hm): Typumwandlung ergänzt, um Warnungen unter CODESYS 3.0 zu vermeiden

Anwendungsszenarien

- Runden von Messwerten auf eine feste Anzahl signifikanter Stellen, unabhängig von der Größenordnung

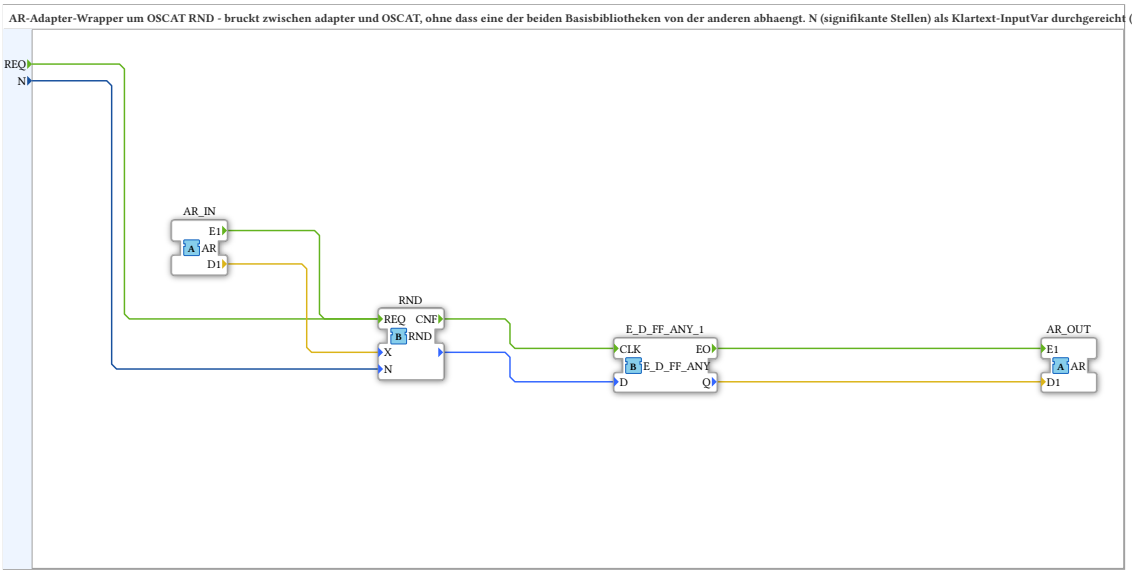
⚖️ Vergleich mit ähnlichen Bausteinen

- `ROUND` rundet auf `N` Nachkommastellen, `RND` auf `N` signifikante Stellen.

Fazit

RND rundet eine REAL-Zahl auf N signifikante Stellen und schützt dabei vor DINT-Überlauf.

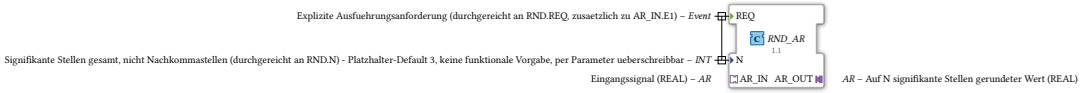
RND_AR



• – ○ • – ○ • – ○ •

Einleitung

RND_AR ist ein AR-Adapter-Wrapper um die OSCAT-Funktion OSCAT::Basic::POUs::Mathematical::RND (rundet X auf N **signifikante Stellen**, nicht auf Nachkommastellen). Er überbrückt zwischen Adaptern und OSCAT, ohne dass eine der beiden Basisbibliotheken von der anderen abhängt. N wird als einfache Eingangsvariable (Default 3) durchgereicht.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: explizite Ausführungsanforderung (durchgereicht an RND.REQ, zusätzlich zu AR_IN.E1)
  - **Verbundene Daten**: N

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

- **N** (INT): Signifikante Stellen gesamt, nicht Nachkommastellen (durchgereicht an RND.N); Platzhalter-Default 3, keine funktionale Vorgabe, per Parameter überschreibbar

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **AR_IN** (Socket, adapter::types::unidirectional::AR): Eingangssignal (REAL)
- **AR_OUT** (Plug, adapter::types::unidirectional::AR): auf N signifikante Stellen gerundeter Wert (REAL)

Verwendete Funktionsbausteine (FBs)

- **RND** (OSCAT::Basic::POUs::Mathematical::RND): Rundungsfunktion, siehe RND
- **E_D_FF_ANY_1** (iec61499::events::E_D_FF_ANY): Änderungsfilter

Funktionsweise

- AR_IN.E1 ⟶ RND.REQ und AR_IN.D1 ⟶ RND.X; N ⟶ RND.N.
- REQ führt ebenfalls auf RND.REQ, parallel zu AR_IN.E1.
- RND.CNF ⟶ E_D_FF_ANY_1.CLK; das Funktionsergebnis geht an E_D_FF_ANY_1.D.
- E_D_FF_ANY_1.EO ⟶ AR_OUT.E1 und E_D_FF_ANY_1.Q ⟶ AR_OUT.D1, aber nur, wenn sich das gerundete Ergebnis seit dem letzten Aufruf tatsächlich geändert hat.

Technische Besonderheiten

- Das explizite REQ mit With N wurde in Version 1.1 (2026-09-20) als Bugfix ergänzt: Zuvor wurde N nur per Datenverbindung an RND durchgereicht und einzig über AR_IN.E1 ausgelöst; da der Wrapper kein Ereignis-Eingang hatte, gab es keine With-Verknüpfung (nach IEC 61499 werden ungebundene Variablen nicht zuverlässig gelesen). REQ erlaubt manuellen oder periodischen Aufruf zusätzlich zum automatischen Auslösen über AR_IN.E1
- Änderungsfilter: AR_OUT.E1 feuert nur bei tatsächlicher Änderung des gerundeten Werts (gleiches Muster wie bei CTRL_PI_AR/FT_DERIV_AR, dort mit E_D_FF_ANY_3/_4 für mehrere Werte gleichzeitig)
- RND ist eine reine OSCAT-Funktion ohne Adapter-Schnittstelle; adapter soll nicht von OSCAT abhängen (und umgekehrt), die Brücke liegt daher im dritten Bibliotheksprojekt OSCAT_adapter
- Unterschied zu ROUND_AR: RND_AR rundet auf signifikante Stellen, ROUND_AR auf Nachkommastellen

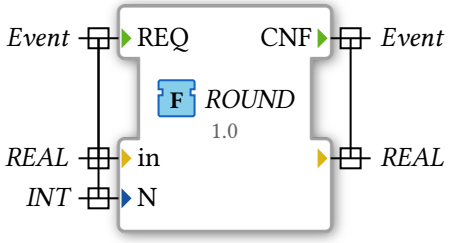
Anwendungsszenarien

- Runden eines AR-Signals auf eine feste Anzahl signifikanter Stellen in einer Adapter-Kette, z. B. für Anzeigewerte

Zusammenfassung

RND_AR rundet einen AR-Wert mit RND auf N signifikante Stellen und gibt das Ergebnis nur bei Änderung als AR-Adapter aus.

ROUND



• – ◦ • – ◦ • – ◦ •

Einleitung

Die Funktion ROUND rundet einen REAL-Wert auf N Nachkommastellen (0 bis 8). Sie wurde aus OSCAT_basic portiert.

Schnittstellenstruktur

Ereignis-Eingänge

- REQ: Startet die Berechnung. Verbundene Daten: in, N

Ereignis-Ausgänge

- CNF: Signalisiert die Berechnung und die Ausgabe des Ergebnisses.

Daten-Eingänge

- in (REAL): Zu rundender Wert
- N (INT): Anzahl der Nachkommastellen (wird auf 0 bis 8 begrenzt)

Daten-Ausgänge

- (kein Name) (REAL): Gerundeter Wert, zugleich der Rückgabewert der Funktion

Adapter

Diese Funktion verwendet keine Adapter.

Funktionsweise

1. Der Faktor `X := EXPT(10.0, LIMIT(0, N, 8))` wird bestimmt.
2. Ist `ABS(in * X) >= 2147483647.0` (DINT-Bereich), wird `in` unverändert zurückgegeben.
3. Andernfalls ist das Ergebnis `DINT_TO_REAL(REAL_TO_DINT(in * X)) / X`.

Technische Besonderheiten

- Implementierung in Structured Text (ST) als `FUNCTION` mit lokaler Variable `X`, ohne internen Zustand
- Teil des Pakets `OSCAT::Basic::POUs::Mathematical`
- `EXPT(10.0, LIMIT(0, N, 8))` ersetzt die globalen OSCAT-Konstanten `decades`; die Bereichsprüfung schützt vor einem `DINT`-Überlauf (Portierung vom 2026-09-17).
- `REAL_TO_DINT` statt `TRUNC` korrigiert den Fehler bei negativen Zahlen (OSCAT-Revision 1.3).
- `N` wird auf 0 bis 8 begrenzt.

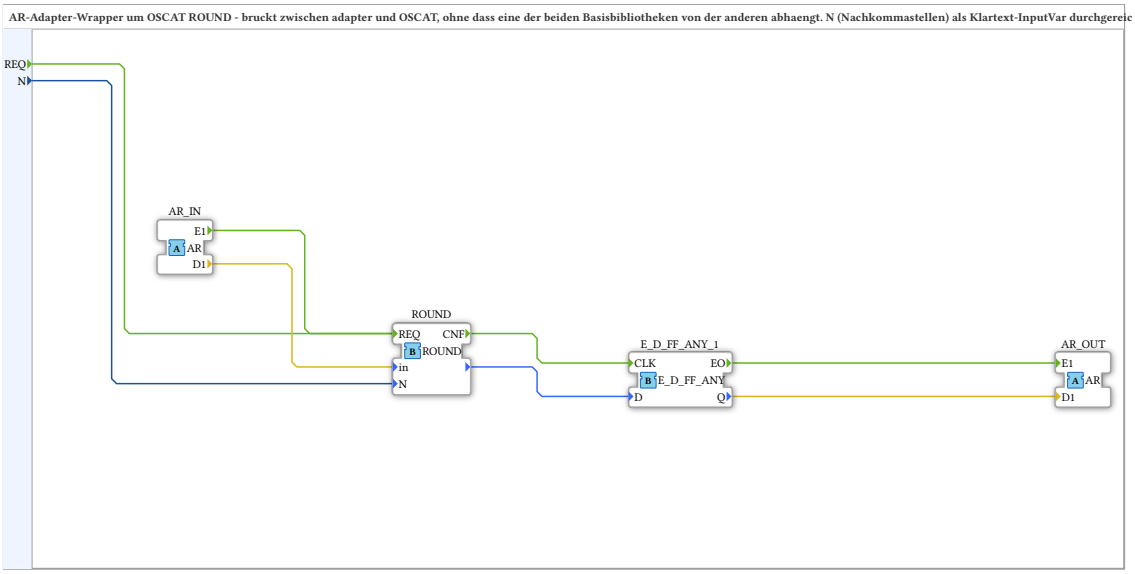
Anwendungsszenarien

- Runden von Messwerten für Anzeige oder Vergleich auf eine feste Anzahl Nachkommastellen

Fazit

`ROUND` rundet einen `REAL`-Wert auf bis zu 8 Nachkommastellen und gibt bei zu großen Werten den Eingang unverändert zurück.

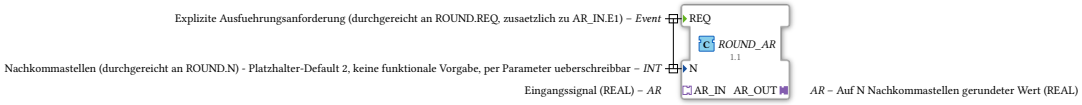
ROUND_AR



• — ○ • — ○ • — ○ •

Einleitung

`ROUND_AR` ist ein AR-Adapter-Wrapper um die OSCAT-Funktion `OSCAT::Basic::POUs::Mathematical::ROUND` (rundet `in` auf `N` Nachkommastellen, 0 bis 8 Stellen). Er überbrückt zwischen Adaptern und OSCAT, ohne dass eine der beiden Basisbibliotheken von der anderen abhängt. `N` wird als einfache Eingangsvariable (Default 2) durchgereicht.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: explizite Ausführungsanforderung (durchgereicht an `ROUND.REQ`, zusätzlich zu `AR_IN.E1`)

- **Verbundene Daten**: N

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

- **N** (INT): Anzahl der Nachkommastellen (durchgereicht an ROUND.N); Platzhalter-Default 2, per Parameter überschreibbar

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **AR_IN** (Socket, adapter::types::unidirectional::AR): Eingangssignal (REAL)
- **AR_OUT** (Plug, adapter::types::unidirectional::AR): auf N Nachkommastellen gerundeter Wert (REAL)

Funktionsweise

- AR_IN.E1 ⟶ ROUND.REQ und AR_IN.D1 ⟶ ROUND.in; N ⟶ ROUND.N.
- REQ führt ebenfalls auf ROUND.REQ, parallel zu AR_IN.E1.
- ROUND.CNF ⟶ E_D_FF_ANY_1.CLK; das Funktionsergebnis geht an E_D_FF_ANY_1.D.
- E_D_FF_ANY_1 (iec61499::events::E_D_FF_ANY) filtert Änderungen: E_D_FF_ANY_1.EO ⟶ AR_OUT.E1 und E_D_FF_ANY_1.Q ⟶ AR_OUT.D1, aber nur, wenn sich das gerundete Ergebnis seit dem letzten Aufruf tatsächlich geändert hat.

Technische Besonderheiten

- Das explizite REQ mit With N wurde ergänzt, weil N sonst nur per Datenverbindung ausgelesen würde; nach IEC 61499 werden nicht über With gebundene Variablen nicht zuverlässig gelesen. REQ erlaubt manuellen oder periodischen Aufruf zusätzlich zum automatischen Auslösen über AR_IN.E1
- Änderungsfilter: AR_OUT.E1 feuert nur bei tatsächlicher Änderung des gerundeten Werts (gleiches Muster wie bei CTRL_PI_AR/FT_DERIV_AR, dort mit E_D_FF_ANY_3/_4)
- ROUND ist eine reine OSCAT-Funktion ohne Adapter-Schnittstelle; die Brücke liegt im dritten Bibliotheksprojekt OSCAT_adapter

Anwendungsszenarien

- Runden angezeigter oder weitergegebener Messwerte auf eine feste Anzahl Nachkommastellen in Adapter-Netzwerken

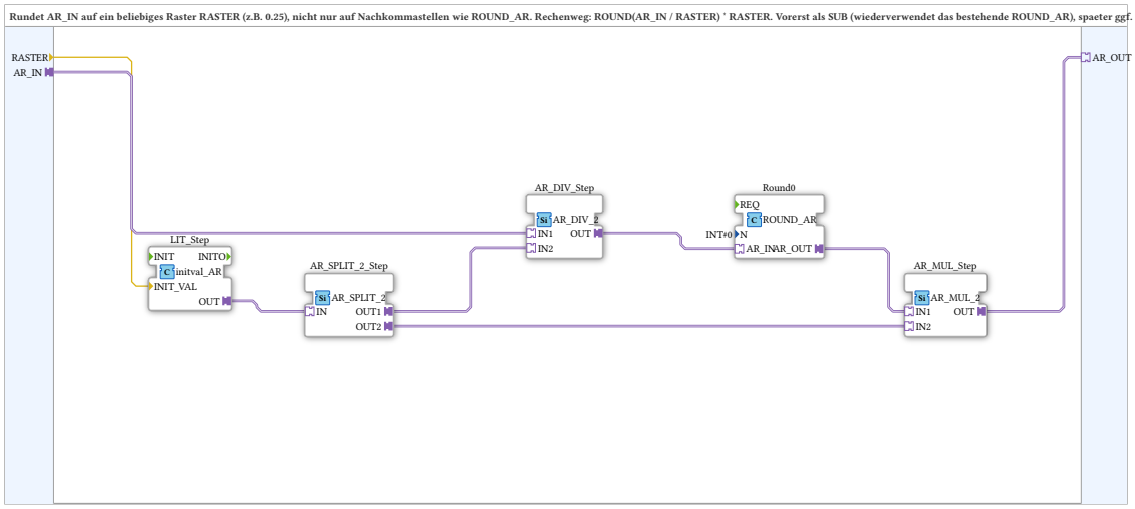
⚖️ Vergleich mit ähnlichen Bausteinen

ROUND_STEP_AR rundet auf ein beliebiges Raster (z. B. 0.25) statt nur auf Nachkommastellen und verwendet intern ROUND_AR (ROUND(AR_IN / RASTER) * RASTER). FLOOR2_AR liefert dagegen eine abgerundete ganze Zahl als DINT.

Fazit

ROUND_AR stellt ROUND als Adapter-Baustein von AR nach AR bereit, rundet auf N Nachkommastellen und gibt das Ergebnis nur bei Änderung aus.

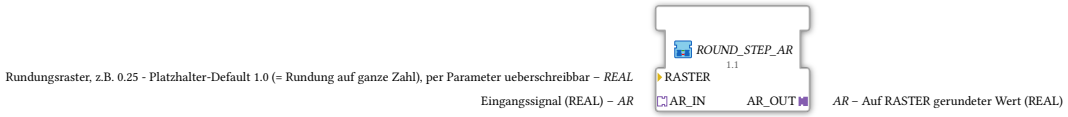
ROUND_STEP_AR



• — ○ • — ○ • — ○ •

Einleitung

`ROUND_STEP_AR` rundet `AR_IN` auf ein **beliebiges Raster** `RASTER` (z. B. 0.25), nicht nur auf Nachkommastellen wie `ROUND_AR`. Rechenweg: `ROUND(AR_IN / RASTER) * RASTER`. Der Baustein ist vorerst eine SubApp, die das bestehende `ROUND_AR` wiederverwendet; später ggf. als eigener Basic-Baustein.



Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

- **RASTER** (`REAL`): Rundungsraster, z. B. `0.25`. Platzhalter-Default `1.0` (= Rundung auf ganze Zahl), per Parameter überschreibbar.

Daten-Ausgänge

Keine.

Adapter

- **AR_IN** (Socket, `adapter::types::unidirectional::AR`): Eingangssignal (`REAL`)
- **AR_OUT** (Plug, `adapter::types::unidirectional::AR`): auf `RASTER` gerundeter Wert (`REAL`)

Verwendete Funktionsbausteine (FBs)

- **`LIT_Step`** (`adapter::types::unidirectional::AR::initval::initval_AR`): erzeugt aus `RASTER` einen konstanten `AR`-Wert
- **`AR_SPLIT_2_Step`** (`adapter::events::unidirectional::AR_SPLIT_2`): fächert `RASTER` auf Division und Multiplikation auf (Adapter-Fanout ist strikt 1:1)
- **`AR_DIV_Step`** (`adapter::iec61131::arithmetic::AR_DIV_2`): `AR_IN / RASTER`
- **`Round0`** (`OSCAT_adapter::Mathematical::ROUND_AR`, `N = 0`): rundet den Quotienten auf eine ganze Zahl
- **`AR_MUL_Step`** (`adapter::iec61131::arithmetic::AR_MUL_2`): multipliziert zurück mit `RASTER`

Funktionsweise

- `RASTER` ⟶ `LIT_Step.INIT_VAL`; `LIT_Step.OUT` ⟶ `AR_SPLIT_2_Step.IN`
- `AR_IN` ⟶ `AR_DIV_Step.IN1`; `AR_SPLIT_2_Step.OUT1` ⟶ `AR_DIV_Step.IN2`
- `AR_DIV_Step.OUT` ⟶ `Round0.AR_IN`
- `Round0.AR_OUT` ⟶ `AR_MUL_Step.IN1`; `AR_SPLIT_2_Step.OUT2` ⟶ `AR_MUL_Step.IN2`
- `AR_MUL_Step.OUT` ⟶ `AR_OUT`

Technische Besonderheiten

- Rundung auf ein Raster statt auf Nachkommastellen
- `RASTER` wird per Parameter bei der Instanziierung gesetzt
- Beispiel: `RASTER = 0.25` rundet auf Viertel

Anwendungsszenarien

- Rundung von Sollwerten oder Anzeigewerten auf Schrittweiten (z. B. 0.25, 0.5, 5)

⚖️ Vergleich mit ähnlichen Bausteinen

- `ROUND_AR` rundet nur auf `N` Nachkommastellen und wird hier intern verwendet.

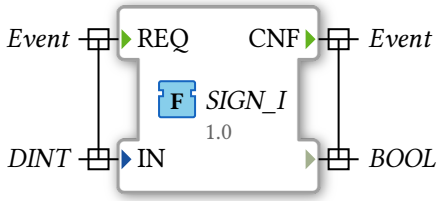
Zusammenfassung

`ROUND_STEP_AR` rundet ein `AR`-Signal auf ein frei wählbares Raster.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SIGN_I



this function return true if the integer input is negative

Interface

Event Inputs

Name	Comment	With
REQ		IN

Event Outputs

Name	Comment	With
CNF		

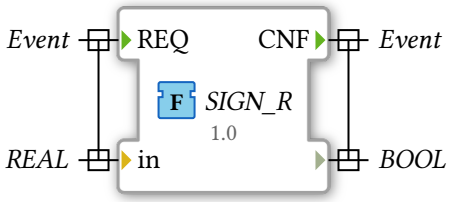
Input Vars

Name	Type	Comment
IN	DINT	

Output Vars

Name	Type	Comment
	BOOL	

SIGN_R



this function return true if the real input is negative

Interface

Event Inputs

Name	Comment	With
REQ		in

Event Outputs

Name	Comment	With
CNF		

Input Vars

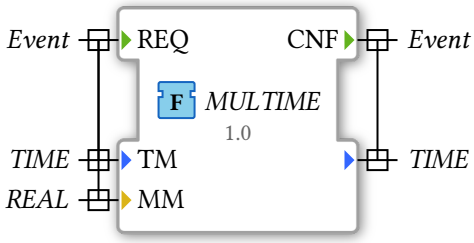
Name	Type	Comment
in	REAL	

Output Vars

Name	Type	Comment
	BOOL	

Time_Date

MULTIME



multiplies a time by a real number and returns a time

Interface

Event Inputs

Name	Comment	With
REQ		TM, MM

Event Outputs

Name	Comment	With

CNF		

Input Vars

Name	Type	Comment
TM	TIME	
MM	REAL	

Output Vars

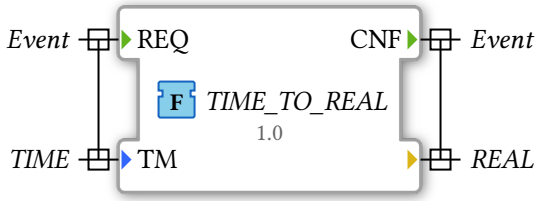
Name	Type	Comment
	TIME	

🛠️ Zugehörige Übungen

- Uebung_020c2

conversion

TIME_TO_REAL



Converts TIME to REAL

Interface

Event Inputs

Name	Comment	With
REQ		TM

Event Outputs

Name	Comment	With
CNF		

Input Vars

Name	Type	Comment
TM	TIME	

Output Vars

Name	Type	Comment
	REAL	

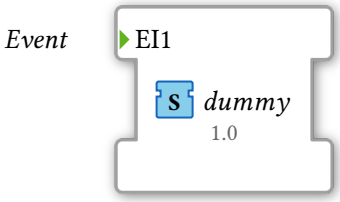
2.1.8.2 Building

🎧 Podcast

- Building Tomorrow's Factories: Bridging OT and IT with IEC 61499

POUs

dummy



does nothing

Interface

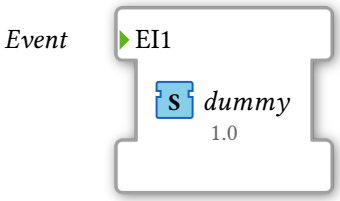
Event Inputs

Name	Comment	With
EI1		

2.1.8.3 Network

POUs

dummy



does nothing

Interface

Event Inputs

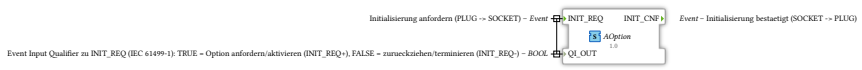
Name	Comment	With
EI1		

2.1.9 Option

2.1.9.1 AOption

Plug (Adapter-Ausgang)

Socket (Adapter-Eingang)



• – ○ • – ○ • – ○ •

Einleitung

Der Adapter `AOption` (Bibliothek `Option`) ist ein Optionsmanagement-Adapter: Er überträgt einen INIT-Request/Confirm-Handshake mit dem Event-Input-Qualifier `QI` (IEC 61499-1). Die Plug-Seite (Anforderer) kann damit eine Option bei der Socket-Seite (Optionsgruppe) anfordern/aktivieren (`QI=TRUE`) oder zurückziehen/terminieren (`QI=FALSE`); der Socket quittiert mit `INIT_CNF`.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Kommentar
`INIT_CNF`	Initialisierung bestätigt (SOCKET -> PLUG).

Ereignis-Ausgänge

Ereignis	Kommentar
`INIT_REQ`	Initialisierung anfordern (PLUG -> SOCKET). Trägt die Daten `QI_OUT`.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Variable	Typ	Kommentar
`QI_OUT`	BOOL	Event Input Qualifier zu `INIT_REQ` (IEC 61499-1): `TRUE` = Option anfordern/aktivieren (INIT_REQ+), `FALSE` = zurückziehen/terminieren (INIT_REQ-).

(Die Angaben gelten aus Sicht der Plug-Seite; auf der Socket-Seite sind die Richtungen gespiegelt.)

Funktionsweise

Der Plug sendet `INIT_REQ` zusammen mit `QI_OUT`. Der Socket wertet `QI_OUT` aus (Optionsgruppe aktivieren bzw. deaktivieren) und antwortet nach Abschluss mit `INIT_CNF`. Der Adapter selbst enthält keine Logik, er definiert nur die Schnittstelle. Zum Anschließen an eigene Netzwerke dienen die Wandler `AOption_EVENT_TO_X` (hält den Plug) und `AOption_X_TO_EVENT` (hält den Socket).

Technische Besonderheiten

- **Handshake**: Jede Anforderung wird mit `INIT_CNF` quittiert (Request/Confirm).
- **Datenbindung**: `QI_OUT` ist mit dem Ereignis `INIT_REQ` über `With` verbunden.
- **Kein Zustandsautomat**: Der Adapter ist ein reiner Schnittstellentyp.

Anwendungsszenarien

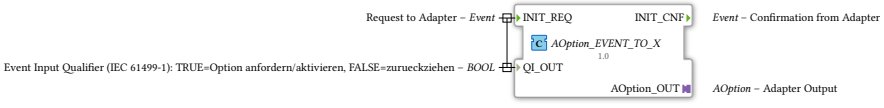
- Ein-/Ausschalten optionaler Funktionsgruppen (z. B. Gruppen mit E/A-Bausteinen) über einen Auswahlbaustein wie `AOption_AD_DEMUX_2`.

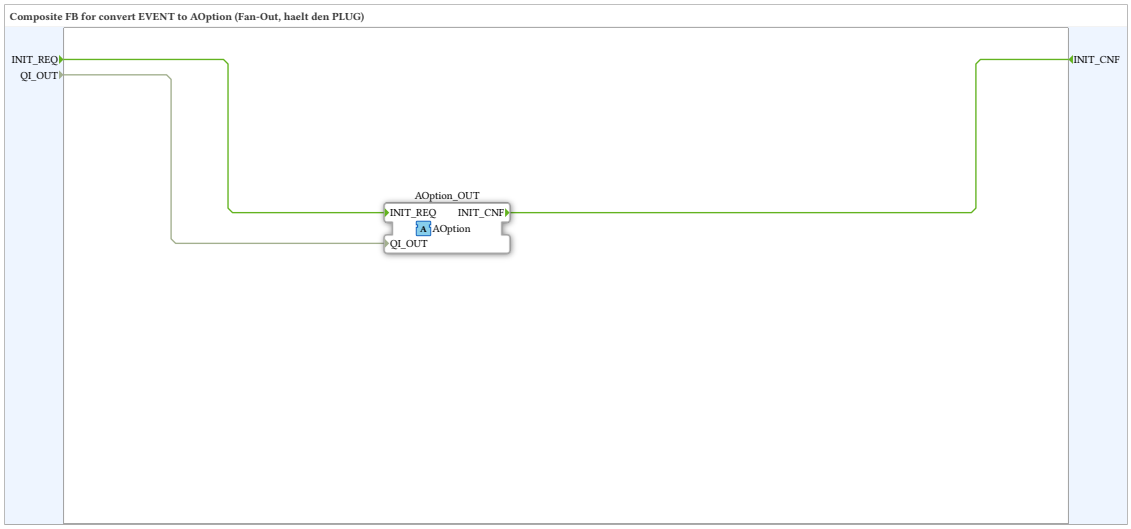
Fazit

`AOption` bietet einen schlanken INIT_REQ/INIT_CNF-Handshake mit `QI` zum Aktivieren und Deaktivieren optionaler Teile einer Anwendung.

2.1.9.2 conversion

AOption_EVENT_TO_X





• – ○ • – ○ • – ○ •

Einleitung

`AOption_EVENT_TO_X` ist ein Composite-FB, der Ereignisse und Daten auf den Options-Adapter `AOption` abbildet (Fan-Out, hält den PLUG).

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar
`INIT_REQ`	EInit	Anforderung an den Adapter. Trägt die Daten `QI_OUT`.

Ereignis-Ausgänge

Ereignis	Typ	Kommentar
`INIT_CNF`	EInit	Bestätigung vom Adapter.

Daten-Eingänge

Variable	Typ	Kommentar
`QI_OUT`	BOOL	Event Input Qualifier (IEC 61499-1): `TRUE` = Option anfordern/aktivieren, `FALSE` = zurückziehen.

Daten-Ausgänge

Keine vorhanden.

Adapter

Adapter	Typ	Richtung	Kommentar
`AOption_OUT`	`Option::AOption`	Plug	Adapter-Ausgang.

Funktionsweise

Das interne Netzwerk besteht nur aus Verbindungen (keine weiteren FBs):

- `INIT_REQ` -> `AOption_OUT.INIT_REQ`
- `AOption_OUT.INIT_CNF` -> `INIT_CNF`
- `QI_OUT` -> `AOption_OUT.QI_OUT`

Eine Anforderung wird also unverändert an den Adapter weitergereicht, die Bestätigung des Sockets kommt als `INIT_CNF` zurück.

Technische Besonderheiten

- Reines Durchreichen ohne Logik.
- Gegenstück ist `AOption_X_TO_EVENT`, der den SOCKET hält.

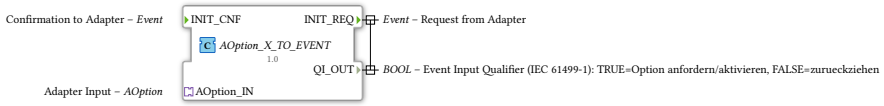
Anwendungsszenarien

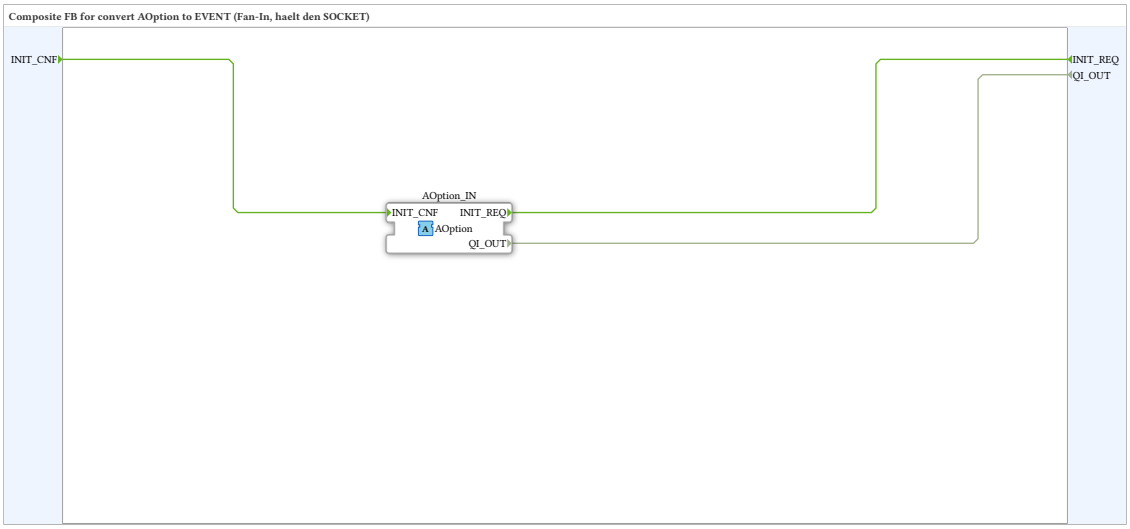
- Ein Anforderer-Netzwerk stellt über diesen Baustein eine Option bei einer Optionsgruppe ein.

Fazit

`AOption_EVENT_TO_X` macht den Plug des `AOption`-Adapters als normale Ereignis-/Datenschnittstelle (`INIT_REQ`, `QI_OUT`, `INIT_CNF`) zugänglich.

AOption_X_TO_EVENT





• – ○ • – ○ • – ○ •

Einleitung

`AOption_X_TO_EVENT` ist ein Composite-FB, der den Options-Adapter `AOption` in Ereignisse und Daten umsetzt (Fan-In, hält den SOCKET). Er wird in Optionsgruppen eingesetzt, um die Anforderung (`INIT_REQ`, `QI_OUT`) zu empfangen und zu quittieren.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar
`INIT_CNF`	EInit	Bestätigung an den Adapter.

Ereignis-Ausgänge

Ereignis	Typ	Kommentar
`INIT_REQ`	EInit	Anforderung vom Adapter. Trägt die Daten `QI_OUT`.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Variable	Typ	Kommentar
`QI_OUT`	BOOL	Event Input Qualifier (IEC 61499-1): `TRUE` = Option anfordern/aktivieren, `FALSE` = zurückziehen.

Adapter

Adapter	Typ	Richtung	Kommentar
`AOption_IN`	`Option::AOption`	Socket	Adapter-Eingang.

Funktionsweise

Das interne Netzwerk besteht nur aus Verbindungen (keine weiteren FBs):

- `AOption_IN.INIT_REQ` -> `INIT_REQ`
- `INIT_CNF` -> `AOption_IN.INIT_CNF`
- `AOption_IN.QI_OUT` -> `QI_OUT`

Die Anforderung des Plugs erscheint als Ereignis `INIT_REQ` mit `QI_OUT`; sobald die Gruppe initialisiert ist, wird `INIT_CNF` an den Baustein gegeben und zum Plug zurückgemeldet.

Technische Besonderheiten

- Reines Durchreichen ohne Logik.
- Gegenstück ist `AOption_EVENT_TO_X`, der den PLUG hält.
- Die SubApps `OptionalGroup_001` und `DummyGroup` verwenden ihn.

Anwendungsszenarien

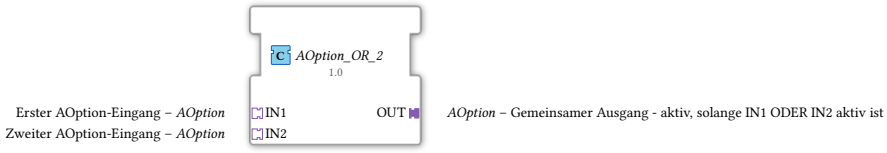
- Optionsgruppe: `QI_OUT` aktiviert/deaktiviert die Gruppe, `INIT_CNF` quittiert die Aktivierung.

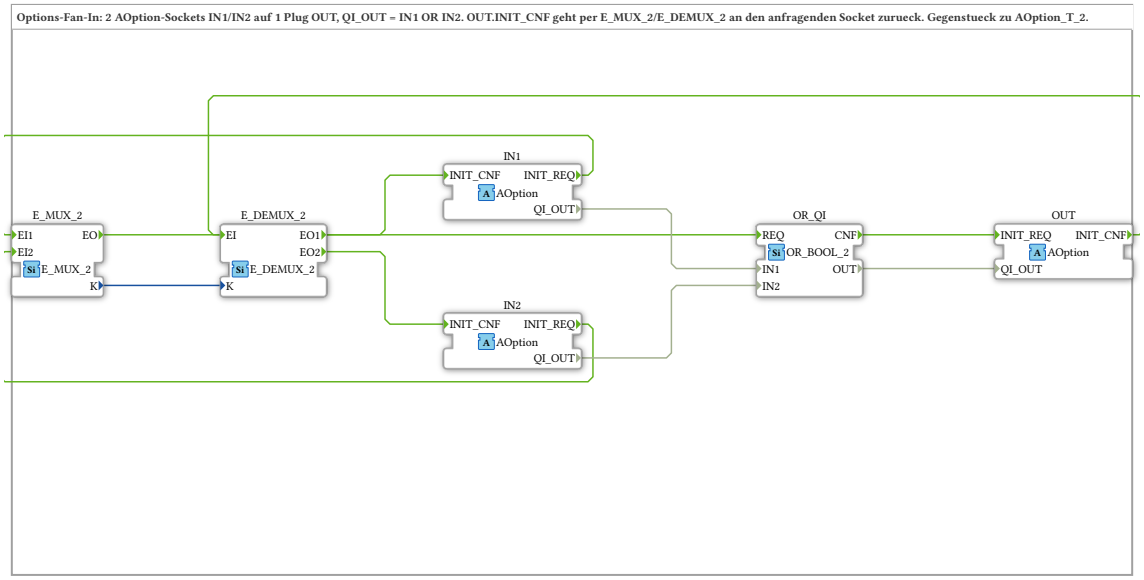
Fazit

`AOption_X_TO_EVENT` macht den Socket des `AOption`-Adapters als normale Ereignis-/Datenschnittstelle (`INIT_REQ`, `QI_OUT`, `INIT_CNF`) zugänglich.

2.1.9.3 Merging

AOption_OR_2





• – ◦ • – ◦ • – ◦ •

Einleitung

`AOption_OR_2` ist ein Composite-FB, der als **Fan-In** (ODER-Verknüpfung) für den Options-Adapter `AOption` dient: Zwei Optionsanforderungen an den Sockets `IN1` und `IN2` werden zu einer einzigen Anforderung am Plug `OUT` zusammengeführt. `OUT` ist aktiv, solange `IN1` **oder** `IN2` aktiv ist. Der Baustein ist das Gegenstück zu `AOption_T_2`.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden (die Ereignisse laufen über die Adapter).

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

Adapter	Typ	Richtung	Kommentar
`IN1`	`Option::AOption`	Socket	Erster AOption-Eingang.
`IN2`	`Option::AOption`	Socket	Zweiter AOption-Eingang.
`OUT`	`Option::AOption`	Plug	Gemeinsamer Ausgang - aktiv, solange `IN1` ODER `IN2` aktiv ist.

Funktionsweise

Das interne Netzwerk enthält drei FBs: `E_MUX_2` (`iec61499::events::E_MUX_2`), `OR_QI` (`iec61131::booleanOperators::OR_BOOL_2`) und `E_DEMUX_2` (`iec61499::events::E_DEMUX_2`).

Ereignisse:

- `IN1.INIT_REQ` -> `E_MUX_2.EI1` (`K = 0`) und `IN2.INIT_REQ` -> `E_MUX_2.EI2` (`K = 1`)
- `E_MUX_2.EO` -> `OR_QI.REQ`
- `OR_QI.CNF` -> `OUT.INIT_REQ`

- `OUT.INIT_CNF` -> `E_DEMUX_2.EI`
- `E_DEMUX_2.EO1` -> `IN1.INIT_CNF` und `E_DEMUX_2.EO2` -> `IN2.INIT_CNF`

Daten:

- `IN1.QI_OUT` -> `OR_QI.IN1`, `IN2.QI_OUT` -> `OR_QI.IN2`
- `OR_QI.OUT` -> `OUT.QI_OUT`
- `E_MUX_2.K` -> `E_DEMUX_2.K`

`E_MUX_2` merkt sich in `K`, über welchen Socket die Anforderung kam, und stößt erst danach die ODER-Verknüpfung an. Deren Ergebnis geht als `INIT_REQ` mit `QI_OUT` an `OUT`. Die Bestätigung `OUT.INIT_CNF` leitet `E_DEMUX_2` anhand von `K` an genau den Socket zurück, der angefragt hat.

Technische Besonderheiten

- **Bestätigung erst nach OUT:** Ein Eingang wird erst bestätigt, wenn die Gruppe am `OUT` fertig ist. Ein vorgeschalteter DEMUX oder `AOption_T_2` sieht die Kette also erst als fertig, wenn sie es tatsächlich ist.
- **Reihenfolge per Verdrahtung:** Weil `OR_QI` über `E_MUX_2.EO` angestoßen wird, ist `K` immer gesetzt, bevor `OUT.INIT_REQ` rausgeht.
- **Nur eine offene Anforderung:** `K` fasst genau einen Absender. Kommt die zweite Anforderung, bevor `OUT.INIT_CNF` zurück ist, geht die Zuordnung verloren. Hinter einem DEMUX (Break-before-Make) passiert das nicht; für unabhängige Quellen bräuchte es einen BasicFB mit ECC.
- **Ein DEMUX-Ausgang nur einmal:** Soll derselbe Ausgang zusätzlich eine eigene Gruppe speisen, wird vorher ein `AOption_T_2` eingesetzt.

Anwendungsszenarien

- Eine gemeinsame Optionsgruppe, die bei mehreren Optionen (`K = 2` oder `K = 3`) aktiv sein soll, aber bei `K = 1` (`DummyGroup`) nicht.

Zusammenfassung

`AOption_OR_2` führt zwei `AOption`-Anforderungen zu einer zusammen (`OUT` aktiv, wenn `IN1` oder `IN2` aktiv ist) und gibt die Bestätigung über `E_MUX_2`/`E_DEMUX_2` an den anfragenden Socket zurück.

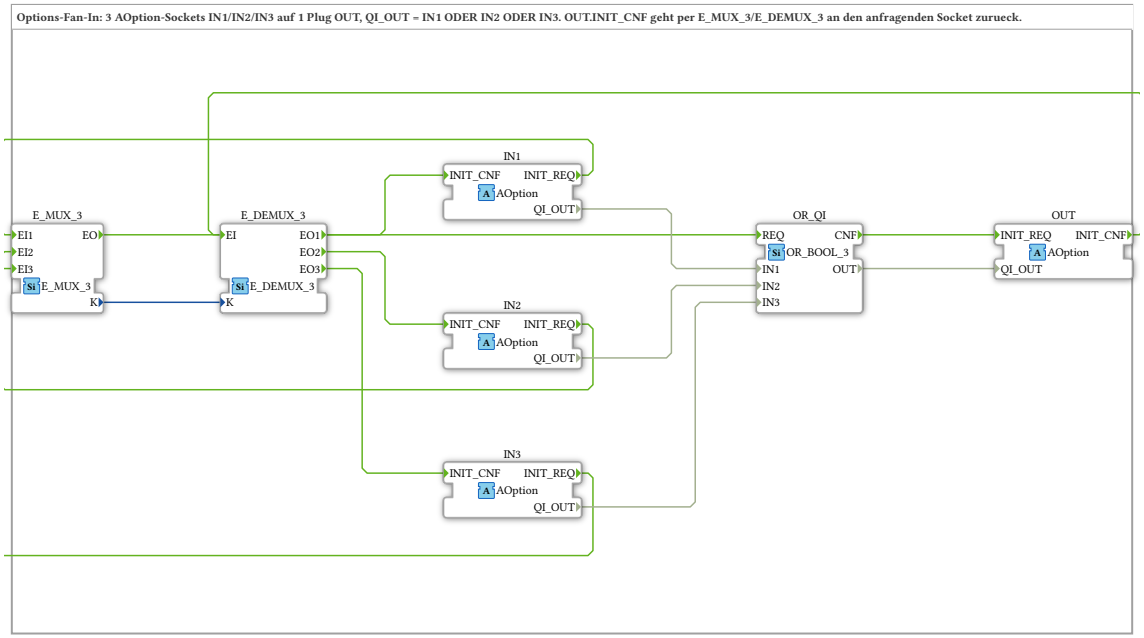
🛠️ Zugehörige Übungen

- Uebung_094b3e_AX
- Uebung_094b3f_AX

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AOption_OR_3



• – ○ • – ○ • – ○ •

Einleitung

`AOption_OR_3` ist ein Composite-FB, der als **Fan-In** (ODER-Verknüpfung) für den Options-Adapter `AOption` dient: Drei Optionsanforderungen an den Sockets `IN1`, `IN2` und `IN3` werden zu einer einzigen Anforderung am Plug `OUT` zusammengeführt. `OUT` ist aktiv, solange `IN1` **oder** `IN2` **oder** `IN3` aktiv ist. Variante von `AOption_OR_2` mit 3 Eingängen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden (die Ereignisse laufen über die Adapter).

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

Adapter	Typ	Richtung	Kommentar
`IN1`	`Option::AOption`	Socket	Erster AOption-Eingang.
`IN2`	`Option::AOption`	Socket	Zweiter AOption-Eingang.
`IN3`	`Option::AOption`	Socket	Dritter AOption-Eingang.
`OUT`	`Option::AOption`	Plug	Gemeinsamer Ausgang - aktiv, solange `IN1` ODER `IN2` aktiv ist.

Funktionsweise

Das interne Netzwerk enthält drei FBs: `E_MUX_3` (`iec61499::events::E_MUX_3`), `OR_QI` (`iec61131::booleanOperators::OR_BOOL_3`) und `E_DEMUX_3` (`iec61499::events::E_DEMUX_3`).

Ereignisse:

- IN1.INIT_REQ -> E_MUX_3.EI1 (K = 0), IN2.INIT_REQ -> E_MUX_3.EI2 (K = 1) und IN3.INIT_REQ -> E_MUX_3.EI3 (K = 2)
- E_MUX_3.EO -> OR_QI.REQ
- OR_QI.CNF -> OUT.INIT_REQ
- OUT.INIT_CNF -> E_DEMUX_3.EI
- E_DEMUX_3.EO1 -> IN1.INIT_CNF, E_DEMUX_3.EO2 -> IN2.INIT_CNF und E_DEMUX_3.EO3 -> IN3.INIT_CNF

Daten:

- IN1.QI_OUT -> OR_QI.IN1, IN2.QI_OUT -> OR_QI.IN2, IN3.QI_OUT -> OR_QI.IN3
- OR_QI.OUT -> OUT.QI_OUT
- E_MUX_3.K -> E_DEMUX_3.K

E_MUX_3 merkt sich in K, über welchen Socket die Anforderung kam, und stößt erst danach die ODER-Verknüpfung an. Deren Ergebnis geht als INIT_REQ mit QI_OUT an OUT. Die Bestätigung OUT.INIT_CNF leitet E_DEMUX_3 anhand von K an genau den Socket zurück, der angefragt hat.

Technische Besonderheiten

- **Bestätigung erst nach OUT:** Ein Eingang wird erst bestätigt, wenn die Gruppe am OUT fertig ist. Ein vorgeschalteter DEMUX oder AOption_T_2 sieht die Kette also erst als fertig, wenn sie es tatsächlich ist.
- **Reihenfolge per Verdrahtung:** Weil OR_QI über E_MUX_3.EO angestoßen wird, ist K immer gesetzt, bevor OUT.INIT_REQ rausgeht.
- **Nur eine offene Anforderung:** K fasst genau einen Absender. Kommt die zweite Anforderung, bevor OUT.INIT_CNF zurück ist, geht die Zuordnung verloren. Hinter einem DEMUX (Break-before-Make) passiert das nicht; für unabhängige Quellen bräuchte es einen BasicFB mit ECC.
- **Ein DEMUX-Ausgang nur einmal:** Soll derselbe Ausgang zusätzlich eine eigene Gruppe speisen, wird vorher ein AOption_T_2 eingesetzt.

Anwendungsszenarien

- Eine gemeinsame Optionsgruppe, die bei mehreren Optionen aktiv sein soll, ohne AOption_OR_2 zu kaskadieren.

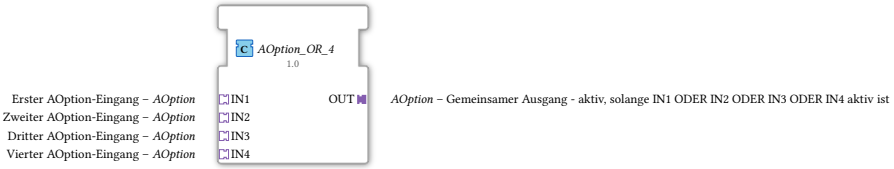
Zusammenfassung

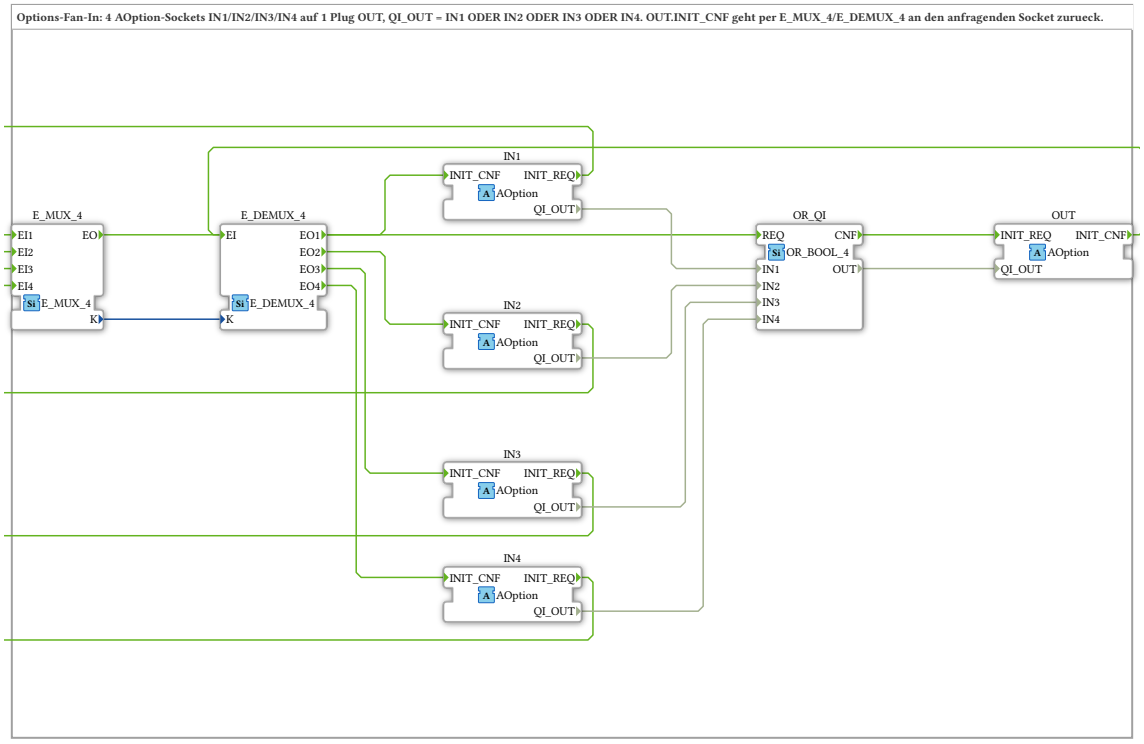
AOption_OR_3 führt drei AOption-Anforderungen zu einer zusammen (OUT aktiv, wenn IN1, IN2 oder IN3 aktiv ist) und gibt die Bestätigung über E_MUX_3/E_DEMUX_3 an den anfragenden Socket zurück.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AOption_OR_4





• – ○ • – ○ • – ○ •

Einleitung

`AOption_OR_4` ist ein Composite-FB, der als **Fan-In** (ODER-Verknüpfung) für den Options-Adapter `AOption` dient: Vier Optionsanforderungen an den Sockets `IN1`, `IN2`, `IN3` und `IN4` werden zu einer einzigen Anforderung am Plug `OUT` zusammengeführt. `OUT` ist aktiv, solange `IN1` **oder** `IN2` **oder** `IN3` **oder** `IN4` aktiv ist. Variante von `AOption_OR_2` mit 4 Eingängen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden (die Ereignisse laufen über die Adapter).

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

Adapter	Typ	Richtung	Kommentar
`IN1`	`Option::AOption`	Socket	Erster AOption-Eingang.
`IN2`	`Option::AOption`	Socket	Zweiter AOption-Eingang.
`IN3`	`Option::AOption`	Socket	Dritter AOption-Eingang.
`IN4`	`Option::AOption`	Socket	Vierter AOption-Eingang.
`OUT`	`Option::AOption`	Plug	Gemeinsamer Ausgang - aktiv, solange `IN1` ODER `IN2` aktiv ist.

Funktionsweise

Das interne Netzwerk enthält drei FBs: `E_MUX_4` (`iec61499::events::E_MUX_4`), `OR_QI` (`iec61131::booleanOperators::OR_BOOL_4`) und `E_DEMUX_4` (`iec61499::events::E_DEMUX_4`).

Ereignisse:

- `IN1.INIT_REQ` -> `E_MUX_4.EI1` (`K = 0`), `IN2.INIT_REQ` -> `E_MUX_4.EI2` (`K = 1`), `IN3.INIT_REQ` -> `E_MUX_4.EI3` (`K = 2`) und `IN4.INIT_REQ` -> `E_MUX_4.EI4` (`K = 3`)
- `E_MUX_4.EO` -> `OR_QI.REQ`
- `OR_QI.CNF` -> `OUT.INIT_REQ`
- `OUT.INIT_CNF` -> `E_DEMUX_4.EI`
- `E_DEMUX_4.EO1` -> `IN1.INIT_CNF`, `E_DEMUX_4.EO2` -> `IN2.INIT_CNF`, `E_DEMUX_4.EO3` -> `IN3.INIT_CNF` und `E_DEMUX_4.EO4` -> `IN4.INIT_CNF`

Daten:

- `IN1.QI_OUT` -> `OR_QI.IN1`, `IN2.QI_OUT` -> `OR_QI.IN2`, `IN3.QI_OUT` -> `OR_QI.IN3`, `IN4.QI_OUT` -> `OR_QI.IN4`
- `OR_QI.OUT` -> `OUT.QI_OUT`
- `E_MUX_4.K` -> `E_DEMUX_4.K`

`E_MUX_4` merkt sich in `K`, über welchen Socket die Anforderung kam, und stößt erst danach die ODER-Verknüpfung an. Deren Ergebnis geht als `INIT_REQ` mit `QI_OUT` an `OUT`. Die Bestätigung `OUT.INIT_CNF` leitet `E_DEMUX_4` anhand von `K` an genau den Socket zurück, der angefragt hat.

Technische Besonderheiten

- **Bestätigung erst nach OUT:** Ein Eingang wird erst bestätigt, wenn die Gruppe am `OUT` fertig ist. Ein vorgeschalteter DEMUX oder `AOption_T_2` sieht die Kette also erst als fertig, wenn sie es tatsächlich ist.
- **Reihenfolge per Verdrahtung:** Weil `OR_QI` über `E_MUX_4.EO` angestoßen wird, ist `K` immer gesetzt, bevor `OUT.INIT_REQ` rausgeht.
- **Nur eine offene Anforderung:** `K` fasst genau einen Absender. Kommt die zweite Anforderung, bevor `OUT.INIT_CNF` zurück ist, geht die Zuordnung verloren. Hinter einem DEMUX (Break-before-Make) passiert das nicht; für unabhängige Quellen bräuchte es einen BasicFB mit ECC.
- **Ein DEMUX-Ausgang nur einmal:** Soll derselbe Ausgang zusätzlich eine eigene Gruppe speisen, wird vorher ein `AOption_T_2` eingesetzt.

Anwendungsszenarien

- Eine gemeinsame Optionsgruppe, die bei mehreren Optionen aktiv sein soll, ohne `AOption_OR_2` zu kaskadieren.

Zusammenfassung

`AOption_OR_4` führt vier `AOption`-Anforderungen zu einer zusammen (`OUT` aktiv, wenn `IN1`, `IN2`, `IN3` oder `IN4` aktiv ist) und gibt die Bestätigung über `E_MUX_4`/`E_DEMUX_4` an den anfragenden Socket zurück.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.1.9.4 selection

AOption_AD_DEMUX_1

• – ○ • – ○ • – ○ •

Einleitung

`AOption_AD_DEMUX_1` ist ein Bitmasken-Demultiplexer (natives Generic-FB `GEN_AOption_AD_DEMUX`, kein ECC und kein Algorithmus in diesem Typ) für `N=1` Options-Ausgänge vom Typ `Option::AOption`. Über den bidirektionalen Bitmasken-Adapter `K` (`adapter::types::bidirectional::AD2`, DWORD) werden beliebig viele der 1 Ausgänge gleichzeitig aktiviert (Multi-Select).

Schnittstellenstruktur

Ereignis-Eingänge / -Ausgänge

Keine direkten Ereignisanschlüsse; die Ereignisse laufen über die Adapter.

Daten-Eingänge / -Ausgänge

Keine direkten Datenanschlüsse; die Daten laufen über die Adapter.

Adapter

Adapter	Typ	Richtung	Kommentar
`OUT1`	`Option::AOption`	Plug	Option 1, aktiv wenn Bit 0 von `K` gesetzt ist (Wert 1).
`K`	`adapter::types::bidirectional::AD2`	Socket	Bitmaske (Bit0=OUT1, Bit1=OUT2, ...), bidirektional: `EO1`/`DO1` = neuer Bitmasken-Wunsch rein, `EI1`/`DI1` = Quittung mit der übernommenen Bitmaske zurück.

Funktionsweise

Bitzuordnung (Multi-Select):

- `Bit 0` (Wert 1): `OUT1`

Beispiel: `K = 1` aktiviert `OUT1`, `K = 0` deaktiviert es. Es können beliebig viele bzw. alle Ausgänge gleichzeitig aktiv sein.

Umschaltverhalten: Nur Bits, die sich gegenüber dem vorherigen `K`-Wert ändern, lösen am zugehörigen Ausgang ein `INIT_REQ` aus (0->1: `QI=TRUE`, aktivieren; 1->0: `QI=FALSE`, deaktivieren). Unveränderte Bits werden nicht erneut getriggert; jeder Ausgang wird pro eintreffendem `K`-Wert höchstens einmal getriggert.

Busy-Gating über den K-Adapter: Solange noch auf `INIT_CNF` von mindestens einem durch den letzten `K`-Wechsel betroffenen Ausgang gewartet wird, wird jeder weitere über `K` (Plug -> Socket, `EO1`/`DO1`) eintreffende Bitmasken-Wunsch ignoriert; er wird weder gepuffert noch nachgeholt. Sobald alle ausgelösten `INIT_CNF` eingetroffen sind, meldet der Baustein dies rückwärts über `K` (Socket -> Plug, `EI1`/`DI1`) mit der übernommenen Bitmaske zurück (nicht mit dem Zustand der Ausgänge; auch Bits außerhalb der vorhandenen Ausgänge werden unverändert quittiert). Erst danach werden wieder neue `K`-Wünsche angenommen.

Technische Besonderheiten

- **Multi-Select** im Gegensatz zum Single-Select von `AOption_AUI_DEMUX_1`.
- **Natives Generic-FB**: Das Verhalten ist in der Laufzeitumgebung implementiert, nicht als ECC oder Algorithmus.
- **Quittierung**: `K` wird erst nach Eintreffen aller `INIT_CNF` zurückgemeldet; zwischenzeitliche Wünsche gehen verloren.

Anwendungsszenarien

- Gleichzeitiges Zu- und Abschalten mehrerer optionaler Funktionsgruppen (z. B. `OptionalGroup_001`) über eine Bitmaske.

Fazit

`AOption_AD_DEMUX_1` schaltet per Bitmaske `K` beliebige Kombinationen von 1 Optionen und meldet die übernommene Bitmaske erst nach Abschluss aller Aktivierungen zurück.

AOption_AD_DEMUX_2

• — ○ • — ○ • — ○ •

Einleitung

AOption_AD_DEMUX_2 ist ein Bitmasken-Demultiplexer (natives Generic-FB GEN_AOption_AD_DEMUX, kein ECC und kein Algorithmus in diesem Typ) für N=2 Options-Ausgänge vom Typ Option::AOption. Über den bidirektionalen Bitmasken-Adapter K (adapter::types::bidirectional::AD2, DWORD) werden beliebig viele der 2 Ausgänge gleichzeitig aktiviert (Multi-Select).

Schnittstellenstruktur

Ereignis-Eingänge / -Ausgänge

Keine direkten Ereignisanschlüsse; die Ereignisse laufen über die Adapter.

Daten-Eingänge / -Ausgänge

Keine direkten Datenanschlüsse; die Daten laufen über die Adapter.

Adapter

Adapter	Typ	Richtung	Kommentar
OUT1	Option::AOption	Plug	Option 1, aktiv wenn Bit 0 von K gesetzt ist (Wert 1).
OUT2	Option::AOption	Plug	Option 2, aktiv wenn Bit 1 von K gesetzt ist (Wert 2).
K	adapter::types::bidirectional::AD2	Socket	Bitmaske (Bit0=OUT1, Bit1=OUT2, ...), bidirektional: EO1/DO1 = neuer Bitmasken-Wunsch rein, EI1/DI1 = Quittung mit der übernommenen Bitmaske zurück.

Funktionsweise

Bitzuordnung (Multi-Select):

- Bit 0 (Wert 1): OUT1
- Bit 1 (Wert 2): OUT2

Beispiel: K = 3 (Bit 0 und Bit 1 gesetzt) aktiviert OUT1 und OUT2; alle anderen Ausgänge bleiben aus. Es können beliebig viele bzw. alle Ausgänge gleichzeitig aktiv sein.

Umschaltverhalten: Nur Bits, die sich gegenüber dem vorherigen K-Wert ändern, lösen am zugehörigen Ausgang ein INIT_REQ aus (0->1: QI=TRUE, aktivieren; 1->0: QI=FALSE, deaktivieren). Unveränderte Bits werden nicht erneut getriggert; jeder Ausgang wird pro eintreffendem K-Wert höchstens einmal getriggert.

Busy-Gating über den K-Adapter: Solange noch auf INIT_CNF von mindestens einem durch den letzten K-Wechsel betroffenen Ausgang gewartet wird, wird jeder weitere über K (Plug -> Socket, EO1/DO1) eintreffende Bitmasken-Wunsch ignoriert; er wird weder gepuffert noch nachgeholt. Sobald alle ausgelösten INIT_CNF eingetroffen sind, meldet der Baustein dies rückwärts über K (Socket -> Plug, EI1/DI1) mit der übernommenen Bitmaske zurück (nicht mit dem Zustand der Ausgänge; auch Bits außerhalb der vorhandenen Ausgänge werden unverändert quittiert). Erst danach werden wieder neue K-Wünsche angenommen.

Technische Besonderheiten

- **Multi-Select** im Gegensatz zum Single-Select von AOption_AUI_DEMUX_2.
- **Natives Generic-FB**: Das Verhalten ist in der Laufzeitumgebung implementiert, nicht als ECC oder Algorithmus.
- **Quittierung**: K wird erst nach Eintreffen aller INIT_CNF zurückgemeldet; zwischenzeitliche Wünsche gehen verloren.

Anwendungsszenarien

- Gleichzeitiges Zu- und Abschalten mehrerer optionaler Funktionsgruppen (z. B. OptionalGroup_001) über eine Bitmaske.

Fazit

`AOption_AD_DEMUX_2` schaltet per Bitmaske `K` beliebige Kombinationen von 2 Optionen und meldet die übernommene Bitmaske erst nach Abschluss aller Aktivierungen zurück.

🛠️ Zugehörige Übungen

- Uebung_094b5_AX
- Uebung_094b5b_AX

AOption_AD_DEMUX_3

• – ○ • – ○ • – ○ •

Einleitung

`AOption_AD_DEMUX_3` ist ein Bitmasken-Demultiplexer (natives Generic-FB `GEN_AOption_AD_DEMUX`, kein ECC und kein Algorithmus in diesem Typ) für `N=3` Options-Ausgänge vom Typ `Option::AOption`. Über den bidirektionalen Bitmasken-Adapter `K` (`adapter::types::bidirectional::AD2`, DWORD) werden beliebig viele der 3 Ausgänge gleichzeitig aktiviert (Multi-Select).

Schnittstellenstruktur

Ereignis-Eingänge / -Ausgänge

Keine direkten Ereignisanschlüsse; die Ereignisse laufen über die Adapter.

Daten-Eingänge / -Ausgänge

Keine direkten Datenanschlüsse; die Daten laufen über die Adapter.

Adapter

Adapter	Typ	Richtung	Kommentar
`OUT1`	`Option::AOption`	Plug	Option 1, aktiv wenn Bit 0 von `K` gesetzt ist (Wert 1).
`OUT2`	`Option::AOption`	Plug	Option 2, aktiv wenn Bit 1 von `K` gesetzt ist (Wert 2).
`OUT3`	`Option::AOption`	Plug	Option 3, aktiv wenn Bit 2 von `K` gesetzt ist (Wert 4).
`K`	`adapter::types::bidirectional::AD2`	Socket	Bitmaske (Bit0=OUT1, Bit1=OUT2, ...), bidirektional: `EO1/DO1` = neuer Bitmasken-Wunsch rein, `EI1/DI1` = Quittung mit der übernommenen Bitmaske zurück.

Funktionsweise

Bitzuordnung (Multi-Select):

- `Bit 0` (Wert 1): `OUT1`
- `Bit 1` (Wert 2): `OUT2`
- `Bit 2` (Wert 4): `OUT3`

Beispiel: `K = 3` (Bit 0 und Bit 1 gesetzt) aktiviert `OUT1` und `OUT2`; alle anderen Ausgänge bleiben aus. Es können beliebig viele bzw. alle Ausgänge gleichzeitig aktiv sein.

Umschaltverhalten: Nur Bits, die sich gegenüber dem vorherigen `K`-Wert ändern, lösen am zugehörigen Ausgang ein `INIT_REQ` aus (0->1: `QI=TRUE`, aktivieren; 1->0: `QI=FALSE`, deaktivieren). Unveränderte Bits werden nicht erneut getriggert; jeder Ausgang wird pro eintreffendem `K`-Wert höchstens einmal getriggert.

Busy-Gating über den K-Adapter: Solange noch auf `INIT_CNF` von mindestens einem durch den letzten `K`-Wechsel betroffenen Ausgang gewartet wird, wird jeder weitere über `K` (Plug -> Socket, `EO1`/`DO1`) eintreffende Bitmasken-Wunsch ignoriert; er wird weder gepuffert noch nachgeholt. Sobald alle ausgelösten `INIT_CNF` eingetroffen sind, meldet der Baustein dies rückwärts über `K` (Socket -> Plug, `EI1`/`DI1`) mit der übernommenen Bitmaske zurück (nicht mit dem Zustand der Ausgänge; auch Bits außerhalb der vorhandenen Ausgänge werden unverändert quittiert). Erst danach werden wieder neue `K`-Wünsche angenommen.

Technische Besonderheiten

- **Multi-Select** im Gegensatz zum Single-Select von `AOption_AUI_DEMUX_3`.
- **Natives Generic-FB**: Das Verhalten ist in der Laufzeitumgebung implementiert, nicht als ECC oder Algorithmus.
- **Quittierung**: `K` wird erst nach Eintreffen aller `INIT_CNF` zurückgemeldet; zwischenzeitliche Wünsche gehen verloren.

Anwendungsszenarien

- Gleichzeitiges Zu- und Abschalten mehrerer optionaler Funktionsgruppen (z. B. `OptionalGroup_001`) über eine Bitmaske.

Fazit

`AOption_AD_DEMUX_3` schaltet per Bitmaske `K` beliebige Kombinationen von 3 Optionen und meldet die übernommene Bitmaske erst nach Abschluss aller Aktivierungen zurück.

AOption_AD_DEMUX_4

• – ○ • – ○ • – ○ •

Einleitung

`AOption_AD_DEMUX_4` ist ein Bitmasken-Demultiplexer (natives Generic-FB `GEN_AOption_AD_DEMUX`, kein ECC und kein Algorithmus in diesem Typ) für `N=4` Options-Ausgänge vom Typ `Option::AOption`. Über den bidirektionalen Bitmasken-Adapter `K` (`adapter::types::bidirectional::AD2`, DWORD) werden beliebig viele der 4 Ausgänge gleichzeitig aktiviert (Multi-Select).

Schnittstellenstruktur

Ereignis-Eingänge / -Ausgänge

Keine direkten Ereignisanschlüsse; die Ereignisse laufen über die Adapter.

Daten-Eingänge / -Ausgänge

Keine direkten Datenanschlüsse; die Daten laufen über die Adapter.

Adapter

Adapter	Typ	Richtung	Kommentar
`OUT1`	`Option::AOption`	Plug	Option 1, aktiv wenn Bit 0 von `K` gesetzt ist (Wert 1).
`OUT2`	`Option::AOption`	Plug	Option 2, aktiv wenn Bit 1 von `K` gesetzt ist (Wert 2).
`OUT3`	`Option::AOption`	Plug	Option 3, aktiv wenn Bit 2 von `K` gesetzt ist (Wert 4).
`OUT4`	`Option::AOption`	Plug	Option 4, aktiv wenn Bit 3 von `K` gesetzt ist (Wert 8).

K	adapter::types::bidirectional::AD2	Socket	Bitmaske (Bit0=OUT1, Bit1=OUT2, ...), bidirektional: EO1/DO1 = neuer Bitmasken-Wunsch rein, EI1/DI1 = Quittung mit der übernommenen Bitmaske zurück.

Funktionsweise

Bitzuordnung (Multi-Select):

- Bit 0 (Wert 1): OUT1
- Bit 1 (Wert 2): OUT2
- Bit 2 (Wert 4): OUT3
- Bit 3 (Wert 8): OUT4

Beispiel: K = 3 (Bit 0 und Bit 1 gesetzt) aktiviert OUT1 und OUT2; alle anderen Ausgänge bleiben aus. Es können beliebig viele bzw. alle Ausgänge gleichzeitig aktiv sein.

Umschaltverhalten: Nur Bits, die sich gegenüber dem vorherigen K-Wert ändern, lösen am zugehörigen Ausgang ein INIT_REQ aus (0->1: QI=TRUE, aktivieren; 1->0: QI=FALSE, deaktivieren). Unveränderte Bits werden nicht erneut getriggert; jeder Ausgang wird pro eintreffendem K-Wert höchstens einmal getriggert.

Busy-Gating über den K-Adapter: Solange noch auf INIT_CNF von mindestens einem durch den letzten K-Wechsel betroffenen Ausgang gewartet wird, wird jeder weitere über K (Plug -> Socket, EO1/DO1) eintreffende Bitmasken-Wunsch ignoriert; er wird weder gepuffert noch nachgeholt. Sobald alle ausgelösten INIT_CNF eingetroffen sind, meldet der Baustein dies rückwärts über K (Socket -> Plug, EI1/DI1) mit der übernommenen Bitmaske zurück (nicht mit dem Zustand der Ausgänge; auch Bits außerhalb der vorhandenen Ausgänge werden unverändert quittiert). Erst danach werden wieder neue K-Wünsche angenommen.

Technische Besonderheiten

- **Multi-Select** im Gegensatz zum Single-Select von AOption_AUI_DEMUX_4.
- **Natives Generic-FB**: Das Verhalten ist in der Laufzeitumgebung implementiert, nicht als ECC oder Algorithmus.
- **Quittierung**: K wird erst nach Eintreffen aller INIT_CNF zurückgemeldet; zwischenzeitliche Wünsche gehen verloren.

Anwendungsszenarien

- Gleichzeitiges Zu- und Abschalten mehrerer optionaler Funktionsgruppen (z. B. OptionalGroup_001) über eine Bitmaske.

Fazit

AOption_AD_DEMUX_4 schaltet per Bitmaske K beliebige Kombinationen von 4 Optionen und meldet die übernommene Bitmaske erst nach Abschluss aller Aktivierungen zurück.

AOption_AD_DEMUX_5

• — ○ • — ○ • — ○ •

Einleitung

AOption_AD_DEMUX_5 ist ein Bitmasken-Demultiplexer (natives Generic-FB GEN_AOption_AD_DEMUX, kein ECC und kein Algorithmus in diesem Typ) für N=5 Options-Ausgänge vom Typ Option::AOption. Über den bidirektionalen Bitmasken-Adapter K (adapter::types::bidirectional::AD2, DWORD) werden beliebig viele der 5 Ausgänge gleichzeitig aktiviert (Multi-Select).

Schnittstellenstruktur

Ereignis-Eingänge / -Ausgänge

Keine direkten Ereignisanschlüsse; die Ereignisse laufen über die Adapter.

Daten-Eingänge / -Ausgänge

Keine direkten Datenanschlüsse; die Daten laufen über die Adapter.

Adapter

Adapter	Typ	Richtung	Kommentar
OUT1	Option::AOption	Plug	Option 1, aktiv wenn Bit 0 von K gesetzt ist (Wert 1).
OUT2	Option::AOption	Plug	Option 2, aktiv wenn Bit 1 von K gesetzt ist (Wert 2).
OUT3	Option::AOption	Plug	Option 3, aktiv wenn Bit 2 von K gesetzt ist (Wert 4).
OUT4	Option::AOption	Plug	Option 4, aktiv wenn Bit 3 von K gesetzt ist (Wert 8).
OUT5	Option::AOption	Plug	Option 5, aktiv wenn Bit 4 von K gesetzt ist (Wert 16).
K	adapter::types::bidirectional::AD2	Socket	Bitmaske (Bit0=OUT1, Bit1=OUT2, ...), bidirektional: EO1/DO1 = neuer Bitmasken-Wunsch rein, EI1/DI1 = Quittung mit der übernommenen Bitmaske zurück.

Funktionsweise

Bitzuordnung (Multi-Select):

- Bit 0 (Wert 1): OUT1
- Bit 1 (Wert 2): OUT2
- Bit 2 (Wert 4): OUT3
- Bit 3 (Wert 8): OUT4
- Bit 4 (Wert 16): OUT5

Beispiel: K = 3 (Bit 0 und Bit 1 gesetzt) aktiviert OUT1 und OUT2; alle anderen Ausgänge bleiben aus. Es können beliebig viele bzw. alle Ausgänge gleichzeitig aktiv sein.

Umschaltverhalten: Nur Bits, die sich gegenüber dem vorherigen K-Wert ändern, lösen am zugehörigen Ausgang ein INIT_REQ aus (0->1: QI=TRUE, aktivieren; 1->0: QI=FALSE, deaktivieren). Unveränderte Bits werden nicht erneut getriggert; jeder Ausgang wird pro eintreffendem K-Wert höchstens einmal getriggert.

Busy-Gating über den K-Adapter: Solange noch auf INIT_CNF von mindestens einem durch den letzten K-Wechsel betroffenen Ausgang gewartet wird, wird jeder weitere über K (Plug -> Socket, EO1/DO1) eintreffende Bitmasken-Wunsch ignoriert; er wird weder gepuffert noch nachgeholt. Sobald alle ausgelösten INIT_CNF eingetroffen sind, meldet der Baustein dies rückwärts über K (Socket -> Plug, EI1/DI1) mit der übernommenen Bitmaske zurück (nicht mit dem Zustand der Ausgänge; auch Bits außerhalb der vorhandenen Ausgänge werden unverändert quittiert). Erst danach werden wieder neue K-Wünsche angenommen.

Technische Besonderheiten

- **Multi-Select** im Gegensatz zum Single-Select von AOption_AUI_DEMUX_5.
- **Natives Generic-FB**: Das Verhalten ist in der Laufzeitumgebung implementiert, nicht als ECC oder Algorithmus.
- **Quittierung**: K wird erst nach Eintreffen aller INIT_CNF zurückgemeldet; zwischenzeitliche Wünsche gehen verloren.

Anwendungsszenarien

- Gleichzeitiges Zu- und Abschalten mehrerer optionaler Funktionsgruppen (z. B. `OptionalGroup_001`) über eine Bitmaske.

Fazit

`AOption_AD_DEMUX_5` schaltet per Bitmaske `K` beliebige Kombinationen von 5 Optionen und meldet die übernommene Bitmaske erst nach Abschluss aller Aktivierungen zurück.

AOption_AD_DEMUX_6

• — ○ • — ○ • — ○ •

Einleitung

`AOption_AD_DEMUX_6` ist ein Bitmasken-Demultiplexer (natives Generic-FB `GEN_AOption_AD_DEMUX`, kein ECC und kein Algorithmus in diesem Typ) für `N=6` Options-Ausgänge vom Typ `Option::AOption`. Über den bidirektionalen Bitmasken-Adapter `K` (`adapter::types::bidirectional::AD2`, DWORD) werden beliebig viele der 6 Ausgänge gleichzeitig aktiviert (Multi-Select).

Schnittstellenstruktur

Ereignis-Eingänge / -Ausgänge

Keine direkten Ereignisanschlüsse; die Ereignisse laufen über die Adapter.

Daten-Eingänge / -Ausgänge

Keine direkten Datenanschlüsse; die Daten laufen über die Adapter.

Adapter

Adapter	Typ	Richtung	Kommentar
`OUT1`	`Option::AOption`	Plug	Option 1, aktiv wenn Bit 0 von `K` gesetzt ist (Wert 1).
`OUT2`	`Option::AOption`	Plug	Option 2, aktiv wenn Bit 1 von `K` gesetzt ist (Wert 2).
`OUT3`	`Option::AOption`	Plug	Option 3, aktiv wenn Bit 2 von `K` gesetzt ist (Wert 4).
`OUT4`	`Option::AOption`	Plug	Option 4, aktiv wenn Bit 3 von `K` gesetzt ist (Wert 8).
`OUT5`	`Option::AOption`	Plug	Option 5, aktiv wenn Bit 4 von `K` gesetzt ist (Wert 16).
`OUT6`	`Option::AOption`	Plug	Option 6, aktiv wenn Bit 5 von `K` gesetzt ist (Wert 32).
`K`	`adapter::types::bidirectional::AD2`	Socket	Bitmaske (Bit0=OUT1, Bit1=OUT2, ...), bidirektional: `EO1/DO1` = neuer Bitmasken-Wunsch rein, `EI1/DI1` = Quittung mit der übernommenen Bitmaske zurück.

Funktionsweise

Bitzuordnung (Multi-Select):

- `Bit 0` (Wert 1): `OUT1`

- `Bit 1` (Wert 2): `OUT2`
- `Bit 2` (Wert 4): `OUT3`
- `Bit 3` (Wert 8): `OUT4`
- `Bit 4` (Wert 16): `OUT5`
- `Bit 5` (Wert 32): `OUT6`

Beispiel: `K = 3` (Bit 0 und Bit 1 gesetzt) aktiviert `OUT1` und `OUT2`; alle anderen Ausgänge bleiben aus. Es können beliebig viele bzw. alle Ausgänge gleichzeitig aktiv sein.

Umschaltverhalten: Nur Bits, die sich gegenüber dem vorherigen `K`-Wert ändern, lösen am zugehörigen Ausgang ein `INIT_REQ` aus (0->1: `QI=TRUE`, aktivieren; 1->0: `QI=FALSE`, deaktivieren). Unveränderte Bits werden nicht erneut getriggert; jeder Ausgang wird pro eintreffendem `K`-Wert höchstens einmal getriggert.

Busy-Gating über den K-Adapter: Solange noch auf `INIT_CNF` von mindestens einem durch den letzten `K`-Wechsel betroffenen Ausgang gewartet wird, wird jeder weitere über `K` (Plug -> Socket, `EO1`/`DO1`) eintreffende Bitmasken-Wunsch ignoriert; er wird weder gepuffert noch nachgeholt. Sobald alle ausgelösten `INIT_CNF` eingetroffen sind, meldet der Baustein dies rückwärts über `K` (Socket -> Plug, `EI1`/`DI1`) mit der übernommenen Bitmaske zurück (nicht mit dem Zustand der Ausgänge; auch Bits außerhalb der vorhandenen Ausgänge werden unverändert quittiert). Erst danach werden wieder neue `K`-Wünsche angenommen.

Technische Besonderheiten

- **Multi-Select** im Gegensatz zum Single-Select von `AOption_AUI_DEMUX_6`.
- **Natives Generic-FB**: Das Verhalten ist in der Laufzeitumgebung implementiert, nicht als ECC oder Algorithmus.
- **Quittierung**: `K` wird erst nach Eintreffen aller `INIT_CNF` zurückgemeldet; zwischenzeitliche Wünsche gehen verloren.

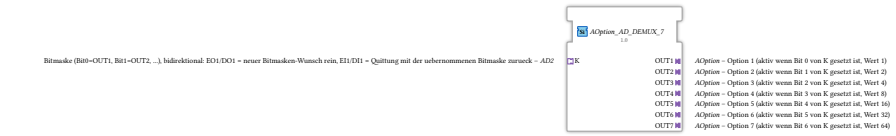
Anwendungsszenarien

- Gleichzeitiges Zu- und Abschalten mehrerer optionaler Funktionsgruppen (z. B. `OptionalGroup_001`) über eine Bitmaske.

Fazit

`AOption_AD_DEMUX_6` schaltet per Bitmaske `K` beliebige Kombinationen von 6 Optionen und meldet die übernommene Bitmaske erst nach Abschluss aller Aktivierungen zurück.

AOption_AD_DEMUX_7



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock AOption_AD_DEMUX_7 ist ein Demultiplexer (natives Generic-FB, kein ECC und kein Algorithmus in diesem Typ) für 7 Options-Ausgange vom Typ `Option::AOption`. Die Auswahl erfolgt über eine **Bitmaske** am bidirektionalen Adapter `K` (`adapter::types::bidirectional::AD2`, DWORD). Es handelt sich um eine Multi-Select-Variante: beliebig viele Ausgänge können gleichzeitig aktiv sein.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Kommunikation läuft ausschließlich über die Adapter.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **K** (Socket, `adapter::types::bidirectional::AD2`): Bitmaske (Bit 0 = OUT1, Bit 1 = OUT2, ...). Bidirektional: `EO1`/`DO1` = neuer Bitmasken-Wunsch kommt herein, `EI1`/`DI1` = Quittung mit der übernommenen Bitmaske geht zurück
- **OUT1** (Plug, `Option::AOption`): Option 1, aktiv wenn Bit 0 von K gesetzt ist (Wert 1)
- **OUT2** (Plug, `Option::AOption`): Option 2, aktiv wenn Bit 1 von K gesetzt ist (Wert 2)
- **OUT3** (Plug, `Option::AOption`): Option 3, aktiv wenn Bit 2 von K gesetzt ist (Wert 4)
- **OUT4** (Plug, `Option::AOption`): Option 4, aktiv wenn Bit 3 von K gesetzt ist (Wert 8)
- **OUT5** (Plug, `Option::AOption`): Option 5, aktiv wenn Bit 4 von K gesetzt ist (Wert 16)
- **OUT6** (Plug, `Option::AOption`): Option 6, aktiv wenn Bit 5 von K gesetzt ist (Wert 32)
- **OUT7** (Plug, `Option::AOption`): Option 7, aktiv wenn Bit 6 von K gesetzt ist (Wert 64)

Jeder Ausgang `OUTx` verwendet den Adapter `Option::AOption`: das Ereignis `INIT_REQ` (zusammen mit `QI_OUT`) läuft in Richtung Plug ⟶ Socket, die Bestätigung `INIT_CNF` in Gegenrichtung.

Funktionsweise

Bitzuordnung (Multi-Select):

- Bit 0 (Wert 1) – `OUT1`
- Bit 1 (Wert 2) – `OUT2`
- Bit 2 (Wert 4) – `OUT3`
- Bit 3 (Wert 8) – `OUT4`
- Bit 4 (Wert 16) – `OUT5`
- Bit 5 (Wert 32) – `OUT6`
- Bit 6 (Wert 64) – `OUT7`

Beispiel: `K = 3` (Bit 0 und Bit 1 gesetzt) aktiviert `OUT1` und `OUT2`, alle anderen Ausgänge sind aus. Es können beliebig viele bzw. alle Ausgänge gleichzeitig aktiv sein.

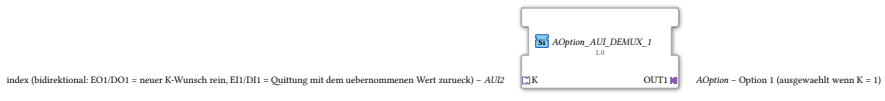
Umschaltverhalten: Nur Bits, die sich gegenüber dem vorherigen `K`-Wert ändern, lösen am zugehörigen Ausgang ein `INIT_REQ` aus (0⟶1: `QI=TRUE`, aktivieren; 1⟶0: `QI=FALSE`, deaktivieren). Unveränderte Bits werden nicht erneut getriggert – jeder Ausgang wird pro eintreffendem `K`-Wert maximal einmal getriggert.

Busy-Gating über den bidirektionalen K-Adapter: Solange noch auf `INIT_CNF` von mindestens einem durch den letzten K-Wechsel betroffenen Ausgang gewartet wird, wird jeder weitere über `K` (Richtung Plug ⟶ Socket, `EO1`/`DO1`) eintreffende Bitmasken-Wunsch ignoriert – er wird weder gepuffert noch nachgeholt. Sind alle ausgelösten `INIT_CNF` eingetroffen, meldet der Baustein dies rückwärts über `K` (Richtung Socket ⟶ Plug, `EI1`/`DI1`) mit der übernommenen Bitmaske zurück (nicht mit dem Zustand der Ausgänge: auch Bits außerhalb der vorhandenen Ausgänge werden unverändert quittiert). Erst danach werden wieder neue K-Wünsche angenommen.

⚖️ Vergleich mit ähnlichen Bausteinen

AOption_AUI_DEMUX_7 ist die Single-Select-Variante mit Index (`AUI2`) statt Bitmaske: dort ist höchstens ein Ausgang gleichzeitig aktiv.

AOption_AUI_DEMUX_1



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock AOption_AUI_DEMUX_1 ist ein Demultiplexer (natives Generic-FB, kein ECC und kein Algorithmus in diesem Typ) für 1 Options-Ausgang vom Typ `Option::AOption`. Die Auswahl erfolgt über einen **Index** am bidirektionalen Adapter `K` (`adapter::types::bidirectional::AUI2`, UINT). Es handelt sich um eine Single-Select-Variante: höchstens ein Ausgang ist gleichzeitig aktiv.

Schnittstellenstruktur

Ereignis-Eingänge
Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Kommunikation läuft ausschließlich über die Adapter.

Ereignis-Ausgänge
Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge
Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge
Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter
- **K** (Socket, `adapter::types::bidirectional::AUI2`): Index. Bidirektional: `EO1`/`DO1` = neuer K-Wunsch kommt herein, `EI1`/`DI1` = Quittung mit dem übernommenen Wert geht zurück
- **OUT1** (Plug, `Option::AOption`): Option 1, ausgewählt wenn K = 1

Jeder Ausgang `OUTx` verwendet den Adapter `Option::AOption`: das Ereignis `INIT_REQ` (zusammen mit `QI_OUT`) läuft in Richtung Plug ⟶ Socket, die Bestätigung `INIT_CNF` in Gegenrichtung.

Funktionsweise
Indexzuordnung:

- `K = 0` – alle Ausgänge AUS
- `K = 1` – `OUT1` AN, alle anderen AUS
- jeder andere Wert (> 1) – alle Ausgänge AUS

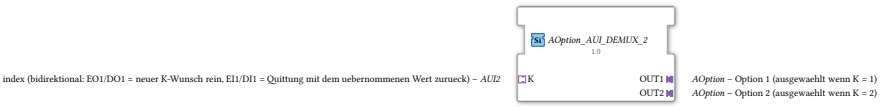
Umschaltverhalten (Break-before-Make): Wechselt `K` von einem aktiven Ausgang auf einen anderen, wird zuerst der bisher aktive Ausgang deaktiviert (`INIT_REQ` mit `QI=FALSE`). Erst wenn von diesem Ausgang `INIT_CNF` eingetroffen ist, wird der neu ausgewählte Ausgang aktiviert (`INIT_REQ` mit `QI=TRUE`). Es ist also nie mehr als eine Option gleichzeitig aktiv. Pro eintreffendem `K`-Wert wird jeder Ausgang maximal einmal getriggert; nicht betroffene Ausgänge werden nicht angefasst.

Busy-Gating über den bidirektionalen K-Adapter: Solange ein Umschaltvorgang läuft (Warten auf `INIT_CNF` vom bisherigen und/oder neuen Ausgang), wird jeder weitere über `K` (Richtung Plug ⟶ Socket, `EO1`/`DO1`) eintreffende Indexwunsch ignoriert – er wird weder gepuffert noch nachgeholt. Nach Abschluss meldet der Baustein dies rückwärts über `K` (Richtung Socket ⟶ Plug, `EI1`/`DI1`) mit dem übernommenen K-Wert zurück (nicht mit dem Zustand der Ausgänge: auch ein ungültiger Wert wird unverändert quittiert). Erst danach werden wieder neue Indexwünsche angenommen. Es zählt immer der Stand nach Abschluss des letzten tatsächlich bearbeiteten Wechsels.

⚖️ Vergleich mit ähnlichen Bausteinen
AOption_AD_DEMUX_1 ist die Multi-Select-Variante mit Bitmaske (`AD2`): dort können mehrere Ausgänge gleichzeitig aktiv sein.

🛠️ Zugehörige Übungen
- Uebung_094c2_AX
- Uebung_094c2b_AX
- Uebung_094c_AX

AOption_AUI_DEMUX_2



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock AOption_AUI_DEMUX_2 ist ein Demultiplexer (natives Generic-FB, kein ECC und kein Algorithmus in diesem Typ) für 2 Options-Ausgange vom Typ `Option::AOption`. Die Auswahl erfolgt über einen **Index** am bidirektionalen Adapter `K` (`adapter::types::bidirectional::AUI2`, UINT). Es handelt sich um eine Single-Select-Variante: höchstens ein Ausgang ist gleichzeitig aktiv.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Kommunikation läuft ausschließlich über die Adapter.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **K** (Socket, `adapter::types::bidirectional::AUI2`): Index. Bidirektional: `EO1`/`DO1` = neuer K-Wunsch kommt herein, `EI1`/`DI1` = Quittung mit dem übernommenen Wert geht zurück
- **OUT1** (Plug, `Option::AOption`): Option 1, ausgewählt wenn K = 1
- **OUT2** (Plug, `Option::AOption`): Option 2, ausgewählt wenn K = 2

Jeder Ausgang `OUTx` verwendet den Adapter `Option::AOption`: das Ereignis `INIT_REQ` (zusammen mit `QI_OUT`) läuft in Richtung Plug ⟶ Socket, die Bestätigung `INIT_CNF` in Gegenrichtung.

Funktionsweise

Indexzuordnung:

- `K = 0` – alle Ausgänge AUS
- `K = 1` – `OUT1` AN, alle anderen AUS
- `K = 2` – `OUT2` AN, alle anderen AUS
- jeder andere Wert (> 2) – alle Ausgänge AUS

Umschaltverhalten (Break-before-Make): Wechselt `K` von einem aktiven Ausgang auf einen anderen, wird zuerst der bisher aktive Ausgang deaktiviert (`INIT_REQ` mit `QI=FALSE`). Erst wenn von diesem Ausgang `INIT_CNF` eingetroffen ist, wird der neu ausgewählte Ausgang aktiviert (`INIT_REQ` mit `QI=TRUE`). Es ist also nie mehr als eine Option gleichzeitig aktiv. Pro eintreffendem `K`-Wert wird jeder Ausgang maximal einmal getriggert; nicht betroffene Ausgänge werden nicht angefasst.

Busy-Gating über den bidirektionalen K-Adapter: Solange ein Umschaltvorgang läuft (Warten auf `INIT_CNF` vom bisherigen und/oder neuen Ausgang), wird jeder weitere über `K` (Richtung Plug ⟶ Socket, `EO1`/`DO1`) eintreffende Indexwunsch ignoriert – er wird weder gepuffert noch nachgeholt. Nach Abschluss meldet der Baustein dies rückwärts über `K` (Richtung Socket ⟶ Plug, `EI1`/`DI1`) mit dem übernommenen K-Wert zurück (nicht mit dem Zustand der Ausgänge: auch ein ungültiger Wert wird unverändert quittiert). Erst danach werden wieder neue Indexwünsche angenommen. Es zählt immer der Stand nach Abschluss des letzten tatsächlich bearbeiteten Wechsels.

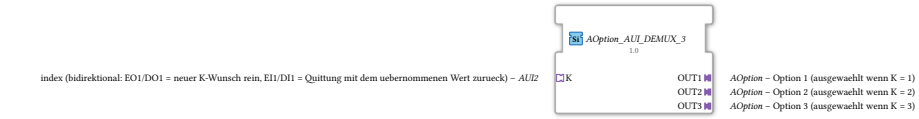
⚖️ Vergleich mit ähnlichen Bausteinen

AOption_AD_DEMUX_2 ist die Multi-Select-Variante mit Bitmaske (`AD2`): dort können mehrere Ausgänge gleichzeitig aktiv sein.

🛠️ Zugehörige Übungen

- Uebung_094b4_AX
- Uebung_094b4b_AX
- Uebung_094b_AX

AOption_AUI_DEMUX_3



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock AOption_AUI_DEMUX_3 ist ein Demultiplexer (natives Generic-FB, kein ECC und kein Algorithmus in diesem Typ) für 3 Options-Ausgange vom Typ `Option::AOption`. Die Auswahl erfolgt über einen **Index** am bidirektionalen Adapter `K` (`adapter::types::bidirectional::AUI2`, UINT). Es handelt sich um eine Single-Select-Variante: höchstens ein Ausgang ist gleichzeitig aktiv.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Kommunikation läuft ausschließlich über die Adapter.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **K** (Socket, `adapter::types::bidirectional::AUI2`): Index. Bidirektional: `EO1`/`DO1` = neuer K-Wunsch kommt herein, `EI1`/`DI1` = Quittung mit dem übernommenen Wert geht zurück
- **OUT1** (Plug, `Option::AOption`): Option 1, ausgewählt wenn K = 1
- **OUT2** (Plug, `Option::AOption`): Option 2, ausgewählt wenn K = 2
- **OUT3** (Plug, `Option::AOption`): Option 3, ausgewählt wenn K = 3

Jeder Ausgang `OUTx` verwendet den Adapter `Option::AOption`: das Ereignis `INIT_REQ` (zusammen mit `QI_OUT`) läuft in Richtung Plug → Socket, die Bestätigung `INIT_CNF` in Gegenrichtung.

Funktionsweise

Indexzuordnung:

- `K = 0` – alle Ausgänge AUS
- `K = 1` – `OUT1` AN, alle anderen AUS
- `K = 2` – `OUT2` AN, alle anderen AUS
- `K = 3` – `OUT3` AN, alle anderen AUS
- jeder andere Wert (> 3) – alle Ausgänge AUS

Umschaltverhalten (Break-before-Make): Wechselt `K` von einem aktiven Ausgang auf einen anderen, wird zuerst der bisher aktive Ausgang deaktiviert (`INIT_REQ` mit `QI=FALSE`). Erst wenn von diesem Ausgang `INIT_CNF` eingetroffen ist, wird der neu ausgewählte Ausgang aktiviert (`INIT_REQ` mit `QI=TRUE`). Es ist also nie mehr als eine Option gleichzeitig aktiv. Pro eintreffendem `K`-Wert wird jeder Ausgang maximal einmal getriggert; nicht betroffene Ausgänge werden nicht angefasst.

Busy-Gating über den bidirektionalen K-Adapter: Solange ein Umschaltvorgang läuft (Warten auf `INIT_CNF` vom bisherigen und/oder neuen Ausgang), wird jeder weitere über `K` (Richtung Plug → Socket, `EO1`/`DO1`) eintreffende Indexwunsch ignoriert – er wird weder gepuffert noch nachgeholt. Nach Abschluss meldet der Baustein dies rückwärts über `K` (Richtung Socket → Plug, `EI1`/`DI1`) mit dem übernommenen K-Wert zurück (nicht mit dem Zustand der Ausgänge: auch ein ungültiger Wert wird unverändert quittiert). Erst danach werden wieder neue Indexwünsche angenommen. Es zählt immer der Stand nach Abschluss des letzten tatsächlich bearbeiteten Wechsels.

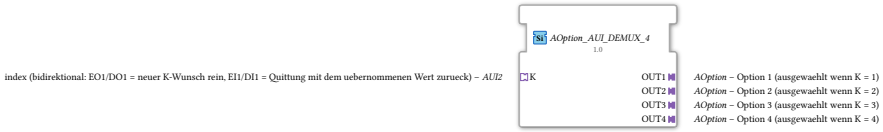
⚖️ Vergleich mit ähnlichen Bausteinen

AOption_AD_DEMUX_3 ist die Multi-Select-Variante mit Bitmaske (`AD2`): dort können mehrere Ausgänge gleichzeitig aktiv sein.

🛠️ Zugehörige Übungen

- Uebung_094b2_AX
- Uebung_094b3_AX
- Uebung_094b3b_AX

AOption_AUI_DEMUX_4



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock AOption_AUI_DEMUX_4 ist ein Demultiplexer (natives Generic-FB, kein ECC und kein Algorithmus in diesem Typ) für 4 Options-Ausgange vom Typ `Option::AOption`. Die Auswahl erfolgt über einen **Index** am bidirektionalen Adapter `K` (`adapter::types::bidirectional::AUI2`, UINT). Es handelt sich um eine Single-Select-Variante: höchstens ein Ausgang ist gleichzeitig aktiv.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Kommunikation läuft ausschließlich über die Adapter.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **K** (Socket, `adapter::types::bidirectional::AUI2`): Index. Bidirektional: `EO1`/`DO1` = neuer K-Wunsch kommt herein, `EI1`/`DI1` = Quittung mit dem übernommenen Wert geht zurück
- **OUT1** (Plug, `Option::AOption`): Option 1, ausgewählt wenn K = 1
- **OUT2** (Plug, `Option::AOption`): Option 2, ausgewählt wenn K = 2
- **OUT3** (Plug, `Option::AOption`): Option 3, ausgewählt wenn K = 3
- **OUT4** (Plug, `Option::AOption`): Option 4, ausgewählt wenn K = 4

Jeder Ausgang `OUTx` verwendet den Adapter `Option::AOption`: das Ereignis `INIT_REQ` (zusammen mit `QI_OUT`) läuft in Richtung Plug ⟶ Socket, die Bestätigung `INIT_CNF` in Gegenrichtung.

Funktionsweise

Indexzuordnung:

- `K = 0` – alle Ausgänge AUS
- `K = 1` – `OUT1` AN, alle anderen AUS
- `K = 2` – `OUT2` AN, alle anderen AUS
- `K = 3` – `OUT3` AN, alle anderen AUS
- `K = 4` – `OUT4` AN, alle anderen AUS
- jeder andere Wert (> 4) – alle Ausgänge AUS

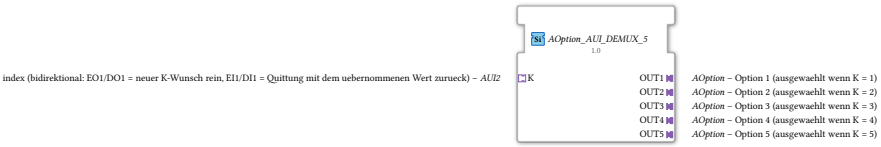
Umschaltverhalten (Break-before-Make): Wechselt K von einem aktiven Ausgang auf einen anderen, wird zuerst der bisher aktive Ausgang deaktiviert (INIT_REQ mit QI=FALSE). Erst wenn von diesem Ausgang INIT_CNF eingetroffen ist, wird der neu ausgewählte Ausgang aktiviert (INIT_REQ mit QI=TRUE). Es ist also nie mehr als eine Option gleichzeitig aktiv. Pro eintreffendem K-Wert wird jeder Ausgang maximal einmal getriggert; nicht betroffene Ausgänge werden nicht angefasst.

Busy-Gating über den bidirektionalen K-Adapter: Solange ein Umschaltvorgang läuft (Warten auf INIT_CNF vom bisherigen und/oder neuen Ausgang), wird jeder weitere über K (Richtung Plug ⟶ Socket, EO1/DO1) eintreffende Indexwunsch ignoriert – er wird weder gepuffert noch nachgeholt. Nach Abschluss meldet der Baustein dies rückwärts über K (Richtung Socket ⟶ Plug, EI1/DI1) mit dem übernommenen K-Wert zurück (nicht mit dem Zustand der Ausgänge: auch ein ungültiger Wert wird unverändert quittiert). Erst danach werden wieder neue Indexwünsche angenommen. Es zählt immer der Stand nach Abschluss des letzten tatsächlich bearbeiteten Wechsels.

⚖️ Vergleich mit ähnlichen Bausteinen

AOption_AD_DEMUX_4 ist die Multi-Select-Variante mit Bitmaske (AD2): dort können mehrere Ausgänge gleichzeitig aktiv sein.

AOption_AUI_DEMUX_5



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock AOption_AUI_DEMUX_5 ist ein Demultiplexer (natives Generic-FB, kein ECC und kein Algorithmus in diesem Typ) für 5 Options-Ausgange vom Typ Option::AOption. Die Auswahl erfolgt über einen **Index** am bidirektionalen Adapter K (adapter::types::bidirectional::AUI2, UINT). Es handelt sich um eine Single-Select-Variante: höchstens ein Ausgang ist gleichzeitig aktiv.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Kommunikation läuft ausschließlich über die Adapter.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **K** (Socket, adapter::types::bidirectional::AUI2): Index. Bidirektional: EO1/DO1 = neuer K-Wunsch kommt herein, EI1/DI1 = Quittung mit dem übernommenen Wert geht zurück
- **OUT1** (Plug, Option::AOption): Option 1, ausgewählt wenn K = 1
- **OUT2** (Plug, Option::AOption): Option 2, ausgewählt wenn K = 2
- **OUT3** (Plug, Option::AOption): Option 3, ausgewählt wenn K = 3
- **OUT4** (Plug, Option::AOption): Option 4, ausgewählt wenn K = 4
- **OUT5** (Plug, Option::AOption): Option 5, ausgewählt wenn K = 5

Jeder Ausgang OUTx verwendet den Adapter Option::AOption: das Ereignis INIT_REQ (zusammen mit QI_OUT) läuft in Richtung Plug ⟶ Socket, die Bestätigung INIT_CNF in Gegenrichtung.

Funktionsweise

Indexzuordnung:

- `K = 0` – alle Ausgänge AUS
- `K = 1` – `OUT1` AN, alle anderen AUS
- `K = 2` – `OUT2` AN, alle anderen AUS
- `K = 3` – `OUT3` AN, alle anderen AUS
- `K = 4` – `OUT4` AN, alle anderen AUS
- `K = 5` – `OUT5` AN, alle anderen AUS
- jeder andere Wert (> 5) – alle Ausgänge AUS

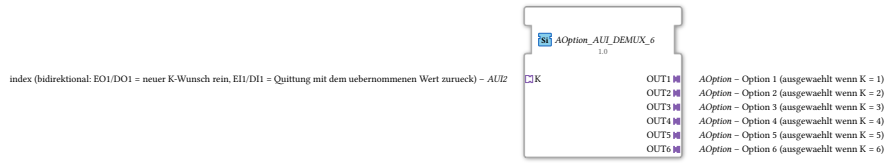
Umschaltverhalten (Break-before-Make): Wechselt `K` von einem aktiven Ausgang auf einen anderen, wird zuerst der bisher aktive Ausgang deaktiviert (`INIT_REQ` mit `QI=FALSE`). Erst wenn von diesem Ausgang `INIT_CNF` eingetroffen ist, wird der neu ausgewählte Ausgang aktiviert (`INIT_REQ` mit `QI=TRUE`). Es ist also nie mehr als eine Option gleichzeitig aktiv. Pro eintreffendem `K`-Wert wird jeder Ausgang maximal einmal getriggert; nicht betroffene Ausgänge werden nicht angefasst.

Busy-Gating über den bidirektionalen K-Adapter: Solange ein Umschaltvorgang läuft (Warten auf `INIT_CNF` vom bisherigen und/oder neuen Ausgang), wird jeder weitere über `K` (Richtung Plug ⟶ Socket, `EO1`/`DO1`) eintreffende Indexwunsch ignoriert – er wird weder gepuffert noch nachgeholt. Nach Abschluss meldet der Baustein dies rückwärts über `K` (Richtung Socket ⟶ Plug, `EI1`/`DI1`) mit dem übernommenen K-Wert zurück (nicht mit dem Zustand der Ausgänge: auch ein ungültiger Wert wird unverändert quittiert). Erst danach werden wieder neue Indexwünsche angenommen. Es zählt immer der Stand nach Abschluss des letzten tatsächlich bearbeiteten Wechsels.

⚖️ Vergleich mit ähnlichen Bausteinen

AOption_AD_DEMUX_5 ist die Multi-Select-Variante mit Bitmaske (`AD2`): dort können mehrere Ausgänge gleichzeitig aktiv sein.

AOption_AUI_DEMUX_6



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock AOption_AUI_DEMUX_6 ist ein Demultiplexer (natives Generic-FB, kein ECC und kein Algorithmus in diesem Typ) für 6 Options-Ausgange vom Typ `Option::AOption`. Die Auswahl erfolgt über einen **Index** am bidirektionalen Adapter `K` (`adapter::types::bidirectional::AUI2`, UINT). Es handelt sich um eine Single-Select-Variante: höchstens ein Ausgang ist gleichzeitig aktiv.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Kommunikation läuft ausschließlich über die Adapter.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **K** (Socket, `adapter::types::bidirectional::AUI2`): Index. Bidirektional: `EO1`/`DO1` = neuer K-Wunsch kommt herein, `EI1`/`DI1` = Quittung mit dem übernommenen Wert geht zurück
- **OUT1** (Plug, `Option::AOption`): Option 1, ausgewählt wenn K = 1
- **OUT2** (Plug, `Option::AOption`): Option 2, ausgewählt wenn K = 2
- **OUT3** (Plug, `Option::AOption`): Option 3, ausgewählt wenn K = 3
- **OUT4** (Plug, `Option::AOption`): Option 4, ausgewählt wenn K = 4
- **OUT5** (Plug, `Option::AOption`): Option 5, ausgewählt wenn K = 5
- **OUT6** (Plug, `Option::AOption`): Option 6, ausgewählt wenn K = 6

Jeder Ausgang `OUTx` verwendet den Adapter `Option::AOption`: das Ereignis `INIT_REQ` (zusammen mit `QI_OUT`) läuft in Richtung Plug ⟶ Socket, die Bestätigung `INIT_CNF` in Gegenrichtung.

Funktionsweise

Indexzuordnung:

- `K = 0` – alle Ausgänge AUS
- `K = 1` – `OUT1` AN, alle anderen AUS
- `K = 2` – `OUT2` AN, alle anderen AUS
- `K = 3` – `OUT3` AN, alle anderen AUS
- `K = 4` – `OUT4` AN, alle anderen AUS
- `K = 5` – `OUT5` AN, alle anderen AUS
- `K = 6` – `OUT6` AN, alle anderen AUS
- jeder andere Wert (> 6) – alle Ausgänge AUS

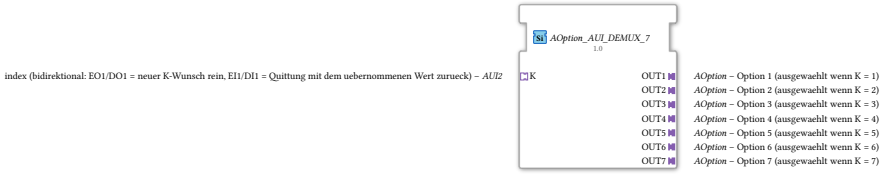
Umschaltverhalten (Break-before-Make): Wechselt `K` von einem aktiven Ausgang auf einen anderen, wird zuerst der bisher aktive Ausgang deaktiviert (`INIT_REQ` mit `QI=FALSE`). Erst wenn von diesem Ausgang `INIT_CNF` eingetroffen ist, wird der neu ausgewählte Ausgang aktiviert (`INIT_REQ` mit `QI=TRUE`). Es ist also nie mehr als eine Option gleichzeitig aktiv. Pro eintreffendem `K`-Wert wird jeder Ausgang maximal einmal getriggert; nicht betroffene Ausgänge werden nicht angefasst.

Busy-Gating über den bidirektionalen K-Adapter: Solange ein Umschaltvorgang läuft (Warten auf `INIT_CNF` vom bisherigen und/oder neuen Ausgang), wird jeder weitere über `K` (Richtung Plug ⟶ Socket, `EO1`/`DO1`) eintreffende Indexwunsch ignoriert – er wird weder gepuffert noch nachgeholt. Nach Abschluss meldet der Baustein dies rückwärts über `K` (Richtung Socket ⟶ Plug, `EI1`/`DI1`) mit dem übernommenen K-Wert zurück (nicht mit dem Zustand der Ausgänge: auch ein ungültiger Wert wird unverändert quittiert). Erst danach werden wieder neue Indexwünsche angenommen. Es zählt immer der Stand nach Abschluss des letzten tatsächlich bearbeiteten Wechsels.

⚖️ Vergleich mit ähnlichen Bausteinen

AOption_AD_DEMUX_6 ist die Multi-Select-Variante mit Bitmaske (`AD2`): dort können mehrere Ausgänge gleichzeitig aktiv sein.

AOption_AUI_DEMUX_7



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock AOption_AUI_DEMUX_7 ist ein Demultiplexer (natives Generic-FB, kein ECC und kein Algorithmus in diesem Typ) für 7 Options-Ausgange vom Typ `Option::AOption`. Die Auswahl erfolgt über einen **Index** am bidirektionalen Adapter `K` (`adapter::types::bidirectional::AUI2`, UINT). Es handelt sich um eine Single-Select-Variante: höchstens ein Ausgang ist gleichzeitig aktiv.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Kommunikation läuft ausschließlich über die Adapter.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **K** (Socket, `adapter::types::bidirectional::AUI2`): Index. Bidirektional: `EO1`/`DO1` = neuer K-Wunsch kommt herein, `EI1`/`DI1` = Quittung mit dem übernommenen Wert geht zurück
- **OUT1** (Plug, `Option::AOption`): Option 1, ausgewählt wenn K = 1
- **OUT2** (Plug, `Option::AOption`): Option 2, ausgewählt wenn K = 2
- **OUT3** (Plug, `Option::AOption`): Option 3, ausgewählt wenn K = 3
- **OUT4** (Plug, `Option::AOption`): Option 4, ausgewählt wenn K = 4
- **OUT5** (Plug, `Option::AOption`): Option 5, ausgewählt wenn K = 5
- **OUT6** (Plug, `Option::AOption`): Option 6, ausgewählt wenn K = 6
- **OUT7** (Plug, `Option::AOption`): Option 7, ausgewählt wenn K = 7

Jeder Ausgang `OUTx` verwendet den Adapter `Option::AOption`: das Ereignis `INIT_REQ` (zusammen mit `QI_OUT`) läuft in Richtung Plug ⟶ Socket, die Bestätigung `INIT_CNF` in Gegenrichtung.

Funktionsweise

Indexzuordnung:

- `K = 0` – alle Ausgänge AUS
- `K = 1` – `OUT1` AN, alle anderen AUS
- `K = 2` – `OUT2` AN, alle anderen AUS
- `K = 3` – `OUT3` AN, alle anderen AUS
- `K = 4` – `OUT4` AN, alle anderen AUS
- `K = 5` – `OUT5` AN, alle anderen AUS
- `K = 6` – `OUT6` AN, alle anderen AUS
- `K = 7` – `OUT7` AN, alle anderen AUS
- jeder andere Wert (> 7) – alle Ausgänge AUS

Umschaltverhalten (Break-before-Make): Wechselt `K` von einem aktiven Ausgang auf einen anderen, wird zuerst der bisher aktive Ausgang deaktiviert (`INIT_REQ` mit `QI=FALSE`). Erst wenn von diesem Ausgang `INIT_CNF` eingetroffen ist, wird der neu ausgewählte Ausgang aktiviert (`INIT_REQ` mit `QI=TRUE`). Es ist also nie mehr als eine Option gleichzeitig aktiv. Pro eintreffendem `K`-Wert wird jeder Ausgang maximal einmal getriggert; nicht betroffene Ausgänge werden nicht angefasst.

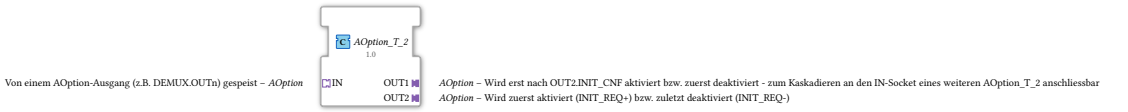
Busy-Gating über den bidirektionalen K-Adapter: Solange ein Umschaltvorgang läuft (Warten auf `INIT_CNF` vom bisherigen und/oder neuen Ausgang), wird jeder weitere über `K` (Richtung Plug ⟶ Socket, `EO1`/`DO1`) eintreffende Indexwunsch ignoriert – er wird weder gepuffert noch nachgeholt. Nach Abschluss meldet der Baustein dies rückwärts über `K` (Richtung Socket ⟶ Plug, `EI1`/`DI1`) mit dem übernommenen K-Wert zurück (nicht mit dem Zustand der Ausgänge: auch ein ungültiger Wert wird unverändert quittiert). Erst danach werden wieder neue Indexwünsche angenommen. Es zählt immer der Stand nach Abschluss des letzten tatsächlich bearbeiteten Wechsels.

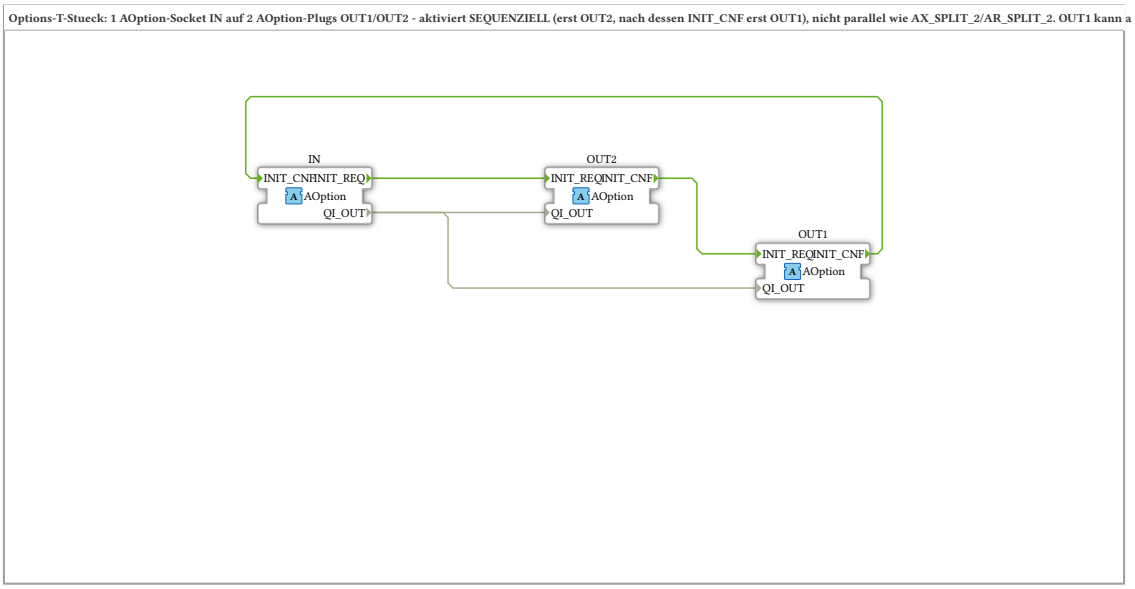
⚖️ Vergleich mit ähnlichen Bausteinen

AOption_AD_DEMUX_7 ist die Multi-Select-Variante mit Bitmaske (`AD2`): dort können mehrere Ausgänge gleichzeitig aktiv sein.

2.1.9.5 splitting

AOption_T_2





• – ○ • – ○ • – ○ •

Einleitung

`AOption_T_2` ist ein Composite-FB, der als sequenzielles **T-Stück** für den Options-Adapter `AOption` dient: Er nimmt über den Socket `IN` eine einzelne Optionsanforderung entgegen und aktiviert daraus zwei nachgeschaltete Options-Verbraucher (Plugs `OUT2` und `OUT1`) **nacheinander**, nicht parallel. Erst wird `OUT2` angesprochen, und erst nach dessen `INIT_CNF` folgt `OUT1`. `IN` wird erst bestätigt, wenn beide Ausgänge bestätigt haben.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden (die Ereignisse laufen über die Adapter).

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

Adapter	Typ	Richtung	Kommentar
`IN`	`Option::AOption`	Socket	Von einem AOption-Ausgang (z. B. `DEMUX.OUTn`) gespeist.
`OUT2`	`Option::AOption`	Plug	Wird zuerst aktiviert (`INIT_REQ` mit `QI_OUT=TRUE`) bzw. zuletzt deaktiviert (`QI_OUT=FALSE`).

OUT1	Option::AOption	Plug	Wird erst nach OUT2.INIT_CNF aktiviert bzw. zuerst deaktiviert; kann zum Kaskadieren an den Socket IN eines weiteren AOption_T_2 angeschlossen werden.

Funktionsweise

Das interne Netzwerk besteht nur aus Verbindungen (keine weiteren FBs, keine Algorithmen, kein ECC):
Ereignisse:

- IN.INIT_REQ -> OUT2.INIT_REQ
- OUT2.INIT_CNF -> OUT1.INIT_REQ
- OUT1.INIT_CNF -> IN.INIT_CNF

Daten:

- IN.QI_OUT -> OUT2.QI_OUT
- IN.QI_OUT -> OUT1.QI_OUT

Eine Anforderung an IN löst also die Kette IN.INIT_REQ -> OUT2 -> (wartet auf OUT2.INIT_CNF) -> OUT1 -> (wartet auf OUT1.INIT_CNF) -> IN.INIT_CNF aus. Die Deaktivierung läuft über dieselbe Verdrahtung in derselben Reihenfolge; QI_OUT trägt dann FALSE statt TRUE. Die Ereigniskette selbst unterscheidet nicht zwischen Aktivierung und Deaktivierung.

Technische Besonderheiten

- **Sequenziell statt parallel:** Die *_SPLIT_2-Bausteine (z. B. AX_SPLIT_2, AR_SPLIT_2) geben ein Ereignis bzw. einen Wert parallel an mehrere Ausgänge weiter. AOption_T_2 serialisiert die Aktivierung bewusst, damit eine Gruppe mit Abhängigkeit zu einer vorher aktivierten Gruppe erst startet, wenn diese tatsächlich fertig ist.
- **Kaskadierbar:** OUT1 kann an den IN-Socket eines weiteren AOption_T_2 angeschlossen werden, so lassen sich beliebig viele Gruppen streng nacheinander hochfahren, ohne dass die einzelne Optionsgruppe selbst einen Weiterleitungs-Plug mitbringen muss (vgl. den Daisy-Chain-Stil in OptionalGroup_001a).
- **Composite statt ECC-BasicFB:** Eine eigene Zustandslogik ist nicht nötig, die reine Verkettung der Ereignisse erzwingt bereits die Reihenfolge.

Anwendungsszenarien

- Eine Optionsanforderung (z. B. aus einem AOption-Demultiplexer) soll zwei Optionsgruppen nacheinander aktivieren, wobei die zweite erst nach Abschluss der ersten starten darf.
- Mehrere kaskadierte AOption_T_2 für längere Ketten von Optionsgruppen.

Zusammenfassung

AOption_T_2 verteilt eine AOption-Anforderung ohne eigene Logik sequenziell auf zwei Ausgänge (OUT2, dann OUT1) und bestätigt IN erst nach beiden Bestätigungen.

🛠️ Zugehörige Übungen

- Uebung_094b3d_AX

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.1.10 SafeArithmetic

Saturierende (klemmende) Arithmetik-Funktionsbausteine, entwickelt unter EPL-2.0-Lizenz von HR Agrartechnik GmbH. Motiviert durch einen echten Hardware-Befund (Uebung_011b3): normale IEC-61131-3-Arithmetik läuft bei Integer-Typen bei Über-/Unterlauf still über und erzeugt bei REAL/LREAL INF/NaN — beides gefährlich für reale Messwerte. Jeder Baustein dieser Bibliothek klemmt das Ergebnis stattdessen auf den gültigen Bereich und meldet die Grenzwertverletzung über einen LIMIT_HIT-Ausgang.

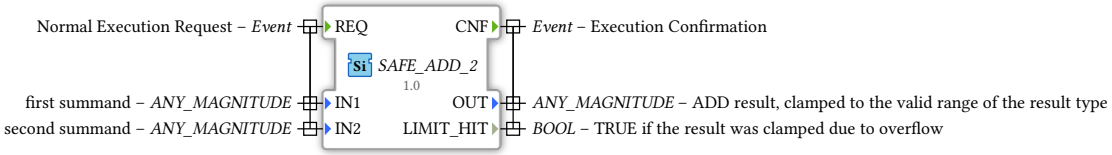
- Arithmetik-Funktionen

2.1.10.1 Arithmetik-Funktionen

Hier finden Sie die saturierenden (klemmenden) Arithmetik-Funktionsbausteine der SafeArithmetic-Bibliothek — sicherheitsorientierter Drop-in-Ersatz für die normalen Arithmetik-Bausteine der Standardbibliothek, die einen Über-/Unterlauf bzw. eine Division durch Null über `LIMIT_HIT` melden, statt still zu überlaufen (Integer) bzw. `INF`/`NaN` zu erzeugen (REAL/LREAL).

- SAFE_ADD_2
- SAFE_ADD_3
- SAFE_ADD_4
- SAFE_DIV
- SAFE_MUL_2
- SAFE_MUL_3
- SAFE_MUL_4
- SAFE_SUB

SAFE_ADD_2



• – ◦ • – ◦ • – ◦ •

Einleitung

SAFE_ADD_2 ist Teil der **SafeArithmetic**-Bibliothek (`SafeArithmetic::arithmetic`, EPL-2.0, HR Agrartechnik GmbH). Es ist ein generischer Funktionsbaustein für Addition mit 2 Eingängen, der sich wie das Standard-`ADD_2` verhält, aber einen Overflow nie unbemerkt durchlässt: bei Overflow wird das Ergebnis auf den gültigen Bereich des Ergebnistyps **geklemmt** (saturiert), und `LIMIT_HIT` zeigt an, dass geklemmt wurde.

Diese Bibliothek existiert, weil normale IEC-61131-3-Arithmetik-Funktionsbausteine (wie `ADD_2` selbst, bei Integer-Typen) bei Overflow den in IEC 61131-3 spezifizierten Zweierkomplement-Wraparound berechnen — laut Norm korrekt, aber gefährlich für reale Messwerte. `Uebung_011b3` zeigt das auf echter Hardware: das normale `F_SUB` berechnet `UDINT#1 - UDINT#12 = UDINT#4294967285`, ein stiller Unterlauf-Wraparound.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Normale Ausführungsanforderung

Ereignis-Ausgänge

- `CNF`: Ausführungsbestätigung

Daten-Eingänge

- `IN1` (ANY_MAGNITUDE): erster Summand
- `IN2` (ANY_MAGNITUDE): zweiter Summand

Daten-Ausgänge

- `OUT` (ANY_MAGNITUDE): ADD-Ergebnis, geklemmt auf den gültigen Bereich des Ergebnistyps
- `LIMIT_HIT` (BOOL): TRUE, wenn das Ergebnis wegen Overflow geklemmt wurde

Funktionsweise

1. **Berechnungsanforderung**: `REQ`-Ereignis mit Werten für IN1, IN2.
2. **Berechnung**: berechnet `IN1 + IN2` von links nach rechts, Schritt für Schritt paarweise. Jeder Schritt wird auf Overflow geprüft; überläuft ein Schritt, wird sein Ergebnis auf den gültigen Bereich des Ergebnistyps geklemmt und `LIMIT_HIT` gesetzt (bleibt gesetzt — einmal von irgendeinem Schritt gesetzt, bleibt es für den Rest der Faltung gesetzt).
3. **Ergebnisausgabe**: `CNF`-Ereignis mit berechnetem `OUT` und `LIMIT_HIT`.

Overflow-/Sättigungsverhalten

- **Signed Integer**-Typen: bei Overflow Klemmen auf `numeric_limits::max()` bzw. `::min()`, Richtung abhängig von den Vorzeichen der Operanden des überlaufenden Schritts.
- **Unsigned Integer**-Typen: bei Overflow Klemmen auf `numeric_limits::max()`.
- **REAL/LREAL**: ein nicht-endliches Zwischen- oder Endergebnis wird immer erkannt, unabhängig davon, ob die *Eingänge* dieses Schritts bereits nicht-endlich waren — `+INF`/`-INF` werden auf `numeric_limits::max()`/`::lowest()` geklemmt, `NaN` (z.B. wenn ein vorheriger Schritt bereits nicht-endlich wurde und die Kombination `NaN` ergibt) wird zu einer deterministischen `0`. `LIMIT_HIT` wird in jedem Fall gesetzt.
- **Inkompatible Typen** (z.B. Mischen eines Magnitude-Typs mit einem nicht verwandten Typ, für den kein `ADD` definiert ist): der betroffene Schritt wird übersprungen (als Fehler geloggt), `OUT` behält den bis dahin akkumulierten Wert der Faltung, und `LIMIT_HIT` wird für diesen Fall **nicht** gesetzt — es ist ein Typfehler, kein numerisches Grenzwert-Ereignis.

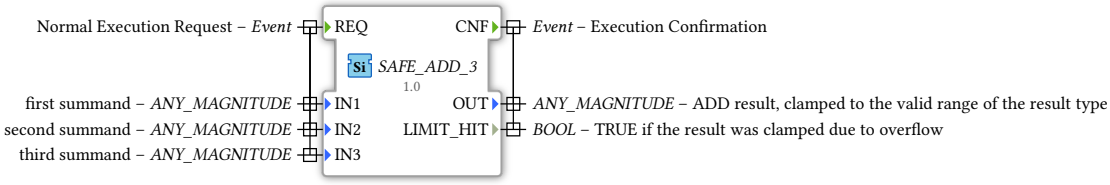
Verwandte Funktionsbausteine

- ADD_2 — das normale, nicht-saturierende Standard-Pendant (stiller Wraparound/`INF` bei Overflow statt Klemmen und Melden).
- Uebung_011b3 — der Hardware-Befund, der diese Bibliothek motiviert hat.

Fazit

SAFE_ADD_2 bietet dieselbe Addition mit 2 Eingängen wie `ADD_2`, aber mit einem expliziten, bereichsbewussten Grenzwert-Signal statt eines stillen Wraparounds (Integer) bzw. eines still verfälschten Ergebnisses (Fließkomma) — auf Kosten der oben dokumentierten Arität-≥-3/signed-Verkettungs-Einschränkung, die eine bewusste Design-Entscheidung ist, kein Versehen.

SAFE_ADD_3



• – ◦ • – ◦ • – ◦ •

Einleitung

SAFE_ADD_3 ist Teil der **SafeArithmetic**-Bibliothek (`SafeArithmetic::arithmetic`, EPL-2.0, HR Agrartechnik GmbH). Es ist ein generischer Funktionsbaustein für Addition mit 3 Eingängen, der sich wie das Standard-`ADD_3` verhält, aber einen Overflow nie unbemerkt durchlässt: bei Overflow wird das Ergebnis auf den gültigen Bereich des Ergebnistyps **geklemmt** (saturiert), und `LIMIT_HIT` zeigt an, dass geklemmt wurde.

Diese Bibliothek existiert, weil normale IEC-61131-3-Arithmetik-Funktionsbausteine (wie `ADD_3` selbst, bei Integer-Typen) bei Overflow den in IEC 61131-3 spezifizierten Zweierkomplement-Wraparound berechnen — laut Norm korrekt, aber gefährlich für reale Messwerte. `Uebung_011b3` zeigt das auf echter Hardware: das normale `F_SUB` berechnet `UDINT#1 - UDINT#12 = UDINT#4294967285`, ein stiller Unterlauf-Wraparound.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Normale Ausführungsanforderung

Ereignis-Ausgänge

- `CNF`: Ausführungsbestätigung

Daten-Eingänge

- `IN1` (ANY_MAGNITUDE): erster Summand
- `IN2` (ANY_MAGNITUDE): zweiter Summand
- `IN3` (ANY_MAGNITUDE): dritter Summand

Daten-Ausgänge

- `OUT` (ANY_MAGNITUDE): ADD-Ergebnis, geklemmt auf den gültigen Bereich des Ergebnistyps
- `LIMIT_HIT` (BOOL): TRUE, wenn das Ergebnis wegen Overflow geklemmt wurde

Funktionsweise

1. **Berechnungsanforderung**: `REQ`-Ereignis mit Werten für IN1, IN2, IN3.
2. **Berechnung**: berechnet `IN1 + IN2 + IN3` von links nach rechts, Schritt für Schritt paarweise. Jeder Schritt wird auf Overflow geprüft; überläuft ein Schritt, wird sein Ergebnis auf den gültigen Bereich des Ergebnistyps geklemmt und `LIMIT_HIT` gesetzt (bleibt gesetzt — einmal von irgendeinem Schritt gesetzt, bleibt es für den Rest der Faltung gesetzt).
3. **Ergebnisausgabe**: `CNF`-Ereignis mit berechnetem `OUT` und `LIMIT_HIT`.

Overflow-/Sättigungsverhalten

- **Signed Integer**-Typen: bei Overflow Klemmen auf `numeric_limits::max()` bzw. `::min()`, Richtung abhängig von den Vorzeichen der Operanden des überlaufenden Schritts.
- **Unsigned Integer**-Typen: bei Overflow Klemmen auf `numeric_limits::max()`.
- **REAL/LREAL**: ein nicht-endliches Zwischen- oder Endergebnis wird immer erkannt, unabhängig davon, ob die *Eingänge* dieses Schritts bereits nicht-endlich waren — `+INF`/`-INF` werden auf `numeric_limits::max()`/`::lowest()` geklemmt, `NaN` (z.B. wenn ein vorheriger Schritt bereits nicht-endlich wurde und die Kombination `NaN` ergibt) wird zu einer deterministischen `0`. `LIMIT_HIT` wird in jedem Fall gesetzt.
- **Inkompatible Typen** (z.B. Mischen eines Magnitude-Typs mit einem nicht verwandten Typ, für den kein `ADD` definiert ist): der betroffene Schritt wird übersprungen (als Fehler geloggt), `OUT` behält den bis dahin akkumulierten Wert der Faltung, und `LIMIT_HIT` wird für diesen Fall **nicht** gesetzt — es ist ein Typfehler, kein numerisches Grenzwert-Ereignis.

Bekannte Einschränkung bei verketteter (Arität ≥ 3) Addition mit SIGNED-Typen

`SAFE_ADD_3` wird als strikte, von links nach rechts gefaltete Kette einzelner Zweier-Operationen ausgewertet: `((IN1 + IN2) + IN3) + ...`, wobei nach jedem Schritt geklemmt wird. Bei **signed** Integer-Typen (und REAL/LREAL, die kein unsigned kennen) bedeutet das: Ein vorübergehender Zwischen-Overflow, den ein späterer Eingang eigentlich wieder in den gültigen Bereich zurückholen würde, wird dauerhaft in `OUT` festgeschrieben — inklusive eines `LIMIT_HIT`, das die wahre Gesamtsumme nicht mehr widerspiegelt.

Beispiel (INT, IN1=20000, IN2=20000, IN3=−30000): das wahre Ergebnis wäre `10000`, locker innerhalb des INT-Bereichs, aber die Faltung klemmt bereits nach dem zweiten Schritt und erholt sich nicht mehr.

Das ist **kein** Bug, der still gefixt wird — es ist dokumentiertes, beabsichtigtes Verhalten: die Faltung wertet als reale Kette von Operationen mit fester Breite aus, genau wie das normale (nicht-sichere) `ADD_3` es auch tut, und meldet eine echte Zwischen-Grenzwertverletzung statt die Berechnung heimlich umzusortieren.

Unsigned-Ketten jeder Arität sind davon nicht betroffen und melden immer einen echten Overflow: bei der Addition kann ein später hinzukommender, nicht-negativer Term eine bereits übergelaufene Zwischensumme nie wieder in den gültigen Bereich zurückholen (die Summe kann nur wachsen).

`SAFE_ADD_2` (nur eine einzelne Operation, keine Verkettung) meldet unabhängig vom Vorzeichen immer einen echten, korrekten Overflow — die Mehrdeutigkeit existiert nur ab Arität 3.

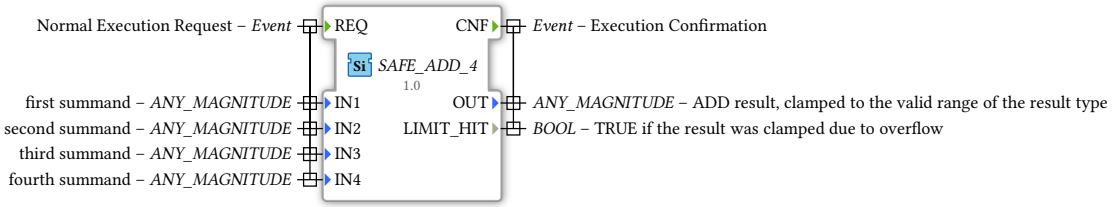
Verwandte Funktionsbausteine

- ADD_3 — das normale, nicht-saturierende Standard-Pendant (stiller Wraparound/`INF` bei Overflow statt Klemmen und Melden).
- Uebung_011b3 — der Hardware-Befund, der diese Bibliothek motiviert hat.

Fazit

SAFE_ADD_3 bietet dieselbe Addition mit 3 Eingängen wie `ADD_3`, aber mit einem expliziten, bereichsbewussten Grenzwert-Signal statt eines stillen Wraparounds (Integer) bzw. eines still verfälschten Ergebnisses (Fließkomma) — auf Kosten der oben dokumentierten Arität-≥-3/signed-Verkettungs-Einschränkung, die eine bewusste Design-Entscheidung ist, kein Versehen.

SAFE_ADD_4



• – ◦ • – ◦ • – ◦ •

Einleitung

SAFE_ADD_4 ist Teil der **SafeArithmetic**-Bibliothek (`SafeArithmetic::arithmetic`, EPL-2.0, HR Agrartechnik GmbH). Es ist ein generischer Funktionsbaustein für Addition mit 4 Eingängen, der sich wie das Standard-`ADD_4` verhält, aber einen Overflow nie unbemerkt durchlässt: bei Overflow wird das Ergebnis auf den gültigen Bereich des Ergebnistyps **geklemmt** (saturiert), und `LIMIT_HIT` zeigt an, dass geklemmt wurde.

Diese Bibliothek existiert, weil normale IEC-61131-3-Arithmetik-Funktionsbausteine (wie `ADD_4` selbst, bei Integer-Typen) bei Overflow den in IEC 61131-3 spezifizierten Zweierkomplement-Wraparound berechnen — laut Norm korrekt, aber gefährlich für reale Messwerte. `Uebung_011b3` zeigt das auf echter Hardware: das normale `F_SUB` berechnet `UDINT#1 - UDINT#12 = UDINT#4294967285`, ein stiller Unterlauf-Wraparound.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Normale Ausführungsanforderung

Ereignis-Ausgänge

- `CNF`: Ausführungsbestätigung

Daten-Eingänge

- `IN1` (ANY_MAGNITUDE): erster Summand
- `IN2` (ANY_MAGNITUDE): zweiter Summand
- `IN3` (ANY_MAGNITUDE): dritter Summand
- `IN4` (ANY_MAGNITUDE): vierter Summand

Daten-Ausgänge

- `OUT` (ANY_MAGNITUDE): ADD-Ergebnis, geklemmt auf den gültigen Bereich des Ergebnistyps
- `LIMIT_HIT` (BOOL): TRUE, wenn das Ergebnis wegen Overflow geklemmt wurde

Funktionsweise

1. **Berechnungsanforderung**: `REQ`-Ereignis mit Werten für IN1, IN2, IN3, IN4.
2. **Berechnung**: berechnet `IN1 + IN2 + IN3 + IN4` von links nach rechts, Schritt für Schritt paarweise. Jeder Schritt wird auf Overflow geprüft; überläuft ein Schritt, wird sein Ergebnis auf den gültigen Bereich des Ergebnistyps geklemmt und `LIMIT_HIT` gesetzt (bleibt gesetzt — einmal von irgendeinem Schritt gesetzt, bleibt es für den Rest der Faltung gesetzt).
3. **Ergebnisausgabe**: `CNF`-Ereignis mit berechnetem `OUT` und `LIMIT_HIT`.

Overflow-/Sättigungsverhalten

- **Signed Integer**-Typen: bei Overflow Klemmen auf `numeric_limits::max()` bzw. `::min()`, Richtung abhängig von den Vorzeichen der Operanden des überlaufenden Schritts.
- **Unsigned Integer**-Typen: bei Overflow Klemmen auf `numeric_limits::max()`.
- **REAL/LREAL**: ein nicht-endliches Zwischen- oder Endergebnis wird immer erkannt, unabhängig davon, ob die *Eingänge* dieses Schritts bereits nicht-endlich waren — `+INF`/`-INF` werden auf `numeric_limits::max()`/`::lowest()` geklemmt, `NaN` (z.B. wenn ein vorheriger Schritt bereits nicht-endlich wurde und die Kombination `NaN` ergibt) wird zu einer deterministischen `0`. `LIMIT_HIT` wird in jedem Fall gesetzt.

- **Inkompatible Typen** (z.B. Mischen eines Magnitude-Typs mit einem nicht verwandten Typ, für den kein `ADD` definiert ist): der betroffene Schritt wird übersprungen (als Fehler geloggt), `OUT` behält den bis dahin akkumulierten Wert der Faltung, und `LIMIT_HIT` wird für diesen Fall **nicht** gesetzt — es ist ein Typfehler, kein numerisches Grenzwert-Ereignis.

Bekannte Einschränkung bei verketteter (Arität ≥ 3) Addition mit SIGNED-Typen

`SAFE_ADD_4` wird als strikte, von links nach rechts gefaltete Kette einzelner Zweier-Operationen ausgewertet: `((IN1 + IN2) + IN3) + ...`, wobei nach jedem Schritt geklemmt wird. Bei **signed** Integer-Typen (und REAL/LREAL, die kein unsigned kennen) bedeutet das: Ein vorübergehender Zwischen-Overflow, den ein späterer Eingang eigentlich wieder in den gültigen Bereich zurückholen würde, wird dauerhaft in `OUT` festgeschrieben — inklusive eines `LIMIT_HIT`, das die wahre Gesamtsumme nicht mehr widerspiegelt.

Beispiel (INT, IN1=20000, IN2=20000, IN3=−30000, IN4=0): das wahre Ergebnis wäre `10000`, locker innerhalb des INT-Bereichs, aber die Faltung klemmt bereits nach dem zweiten Schritt und erholt sich nicht mehr.

Das ist **kein** Bug, der still gefixt wird — es ist dokumentiertes, beabsichtigtes Verhalten: die Faltung wertet als reale Kette von Operationen mit fester Breite aus, genau wie das normale (nicht-sichere) `ADD_4` es auch tut, und meldet eine echte Zwischen-Grenzwertverletzung statt die Berechnung heimlich umzusortieren.

Unsigned-Ketten jeder Arität sind davon nicht betroffen und melden immer einen echten Overflow: bei der Addition kann ein später hinzukommender, nicht-negativer Term eine bereits übergelaufene Zwischensumme nie wieder in den gültigen Bereich zurückholen (die Summe kann nur wachsen).

`SAFE_ADD_2` (nur eine einzelne Operation, keine Verkettung) meldet unabhängig vom Vorzeichen immer einen echten, korrekten Overflow — die Mehrdeutigkeit existiert nur ab Arität 3.

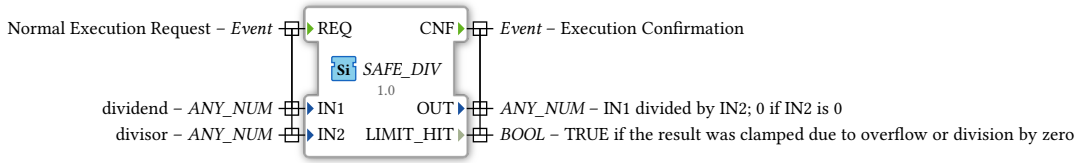
Verwandte Funktionsbausteine

- ADD_4 — das normale, nicht-saturierende Standard-Pendant (stiller Wraparound/`INF` bei Overflow statt Klemmen und Melden).
- Uebung_011b3 — der Hardware-Befund, der diese Bibliothek motiviert hat.

Fazit

SAFE_ADD_4 bietet dieselbe Addition mit 4 Eingängen wie `ADD_4`, aber mit einem expliziten, bereichsbewussten Grenzwert-Signal statt eines stillen Wraparounds (Integer) bzw. eines still verfälschten Ergebnisses (Fließkomma) — auf Kosten der oben dokumentierten Arität-≥-3/signed-Verkettungs-Einschränkung, die eine bewusste Design-Entscheidung ist, kein Versehen.

SAFE_DIV



• – ◦ • – ◦ • – ◦ •

Einleitung

SAFE_DIV ist Teil der **SafeArithmetic**-Bibliothek (`SafeArithmetic::arithmetic`, EPL-2.0, HR Agrartechnik GmbH). Es ist ein fester Divisions-Funktionsbaustein mit 2 Eingängen, der sich wie das Standard-`F_DIV` verhält, aber niemals abstürzt, `INF`/`NaN` erzeugt oder einen Overflow unbemerkt durchlässt: Division durch Null und Overflow werden als **Klemm**-Bedingung behandelt, und `LIMIT_HIT` zeigt an, dass geklemmt wurde.

Wie `SAFE_SUB` ist auch `SAFE_DIV` **nicht** generisch in der Arität: Division ist weder assoziativ noch kommutativ, sodass eine „verkettete“ Division keine wohldefinierte Bedeutung hätte, wie es bei erweiterbarem ADD/MUL der Fall ist.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Normale Ausführungsanforderung

Ereignis-Ausgänge

- `CNF`: Ausführungsbestätigung

Daten-Eingänge

- `IN1` (ANY_NUM): Dividend
- `IN2` (ANY_NUM): Divisor

Daten-Ausgänge

- `OUT` (ANY_NUM): IN1 dividiert durch IN2; 0, wenn IN2 gleich 0 ist
- `LIMIT_HIT` (BOOL): TRUE, wenn das Ergebnis wegen Overflow oder Division durch Null geklemmt wurde

Funktionsweise

1. **Berechnungsanforderung**: `REQ`-Ereignis mit Werten für IN1 und IN2.
2. **Division**: berechnet `IN1 / IN2`.
3. **Ergebnisausgabe**: `CNF`-Ereignis mit berechnetem `OUT` und `LIMIT_HIT`.

Overflow-/Division-durch-Null-Verhalten

- **Beliebiger Integer**-Typ, `IN2 == 0`: klemmt auf `0` mit `LIMIT_HIT = TRUE`, statt abzustürzen (Integer-Division durch Null ist in reinem C++ undefiniertes Verhalten und würde `F_DIV` zum Absturz bringen).
- **Signed Integer**-Typen: der eine verbleibende Fall, der überlaufen kann, `INT_MIN / -1`, klemmt auf `numeric_limits::max()`.
- **REAL/LREAL**, `IN2 == 0.0`: klemmt auf `0` mit `LIMIT_HIT = TRUE`, statt IEEE-754 `+INF`/`-INF`/`NaN` zu erzeugen (`0.0 / 0.0`).
- **REAL/LREAL**, `IN2 != 0.0`: ein nicht-endliches Zwischen- oder Endergebnis wird immer erkannt, unabhängig davon, ob die *Eingänge* bereits nicht-endlich waren — `+INF`/`-INF` werden auf `numeric_limits::max()`/`::lowest()` geklemmt, `NaN` wird zu einer deterministischen `0`. `LIMIT_HIT` wird in jedem Fall gesetzt.
- **Inkompatible Typen** (z.B. Mischen eines numerischen Typs mit einem nicht verwandten Typ, für den keine Division definiert ist): das Ergebnis dieser Operation ist undefiniert (als Fehler geloggt), `OUT` bleibt unverändert, und `LIMIT_HIT` wird für diesen Fall **nicht** gesetzt — es ist ein Typfehler, kein numerisches Grenzwert-Ereignis.

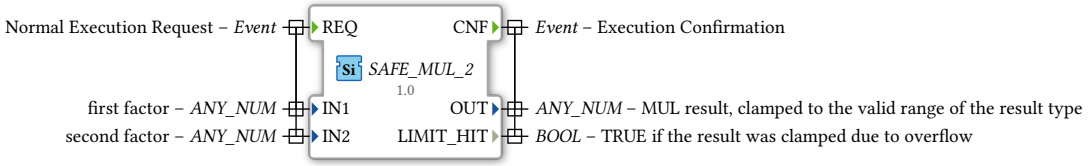
Verwandte Funktionsbausteine

- F_DIV — das normale, nicht-saturierende Standard-Pendant (undefiniertes Verhalten bei Integer-Division durch Null, `INF`/`NaN` bei Fließkomma-Division durch/von Null, statt Klemmen und Melden).
- Uebung_011b7 — eine Division-durch-Null-Demo mit `SAFE_DIV`, `LIMIT_HIT` auf einem zweiten Ausgang verdrahtet, damit das Klemmen auf der Hardware sichtbar wird.

Fazit

SAFE_DIV bietet dieselbe Division wie `F_DIV`, aber mit einem expliziten, bereichsbewussten Grenzwert-Signal statt undefiniertem Verhalten (Integer-Division durch Null) oder einem still verfälschten Ergebnis (Fließkomma-Overflow/`NaN`).

SAFE_MUL_2



• – ◦ • – ◦ • – ◦ •

Einleitung

SAFE_MUL_2 ist Teil der **SafeArithmetic**-Bibliothek (`SafeArithmetic::arithmetic`, EPL-2.0, HR Agrartechnik GmbH). Es ist ein generischer Funktionsbaustein für Multiplikation mit 2 Eingängen, der sich wie das Standard-`F_MUL` verhält, aber einen Overflow nie unbemerkt durchlässt: bei Overflow wird

das Ergebnis auf den gültigen Bereich des Ergebnistyps **geklemmt** (saturiert), und `LIMIT_HIT` zeigt an, dass geklemmt wurde.

Diese Bibliothek existiert, weil normale IEC-61131-3-Arithmetik-Funktionsbausteine (wie `F_MUL` selbst, bei Integer-Typen) bei Overflow den in IEC 61131-3 spezifizierten Zweierkomplement-Wraparound berechnen — laut Norm korrekt, aber gefährlich für reale Messwerte. `Uebung_011b3` zeigt das auf echter Hardware: das normale `F_SUB` berechnet `UDINT#1 - UDINT#12 = UDINT#4294967285`, ein stiller Unterlauf-Wraparound.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Normale Ausführungsanforderung

Ereignis-Ausgänge

- `CNF`: Ausführungsbestätigung

Daten-Eingänge

- `IN1` (ANY_NUM): erster Faktor
- `IN2` (ANY_NUM): zweiter Faktor

Daten-Ausgänge

- `OUT` (ANY_NUM): MUL-Ergebnis, geklemmt auf den gültigen Bereich des Ergebnistyps
- `LIMIT_HIT` (BOOL): TRUE, wenn das Ergebnis wegen Overflow geklemmt wurde

Funktionsweise

1. **Berechnungsanforderung**: `REQ`-Ereignis mit Werten für IN1, IN2.
2. **Berechnung**: berechnet `IN1 * IN2` von links nach rechts, Schritt für Schritt paarweise. Jeder Schritt wird auf Overflow geprüft; überläuft ein Schritt, wird sein Ergebnis auf den gültigen Bereich des Ergebnistyps geklemmt und `LIMIT_HIT` gesetzt (bleibt gesetzt — einmal von irgendeinem Schritt gesetzt, bleibt es für den Rest der Faltung gesetzt).
3. **Ergebnisausgabe**: `CNF`-Ereignis mit berechnetem `OUT` und `LIMIT_HIT`.

Overflow-/Sättigungsverhalten

- **Signed Integer**-Typen: bei Overflow Klemmen auf `numeric_limits::max()` bzw. `::min()`, Richtung abhängig von den Vorzeichen der Operanden des überlaufenden Schritts.
- **Unsigned Integer**-Typen: bei Overflow Klemmen auf `numeric_limits::max()`.
- **REAL/LREAL**: ein nicht-endliches Zwischen- oder Endergebnis wird immer erkannt, unabhängig davon, ob die *Eingänge* dieses Schritts bereits nicht-endlich waren — `+INF`/`-INF` werden auf `numeric_limits::max()`/`::lowest()` geklemmt, `NaN` (z.B. wenn ein vorheriger Schritt bereits nicht-endlich wurde und die Kombination `NaN` ergibt) wird zu einer deterministischen `0`. `LIMIT_HIT` wird in jedem Fall gesetzt.
- **Inkompatible Typen** (z.B. Mischen eines Magnitude-Typs mit einem nicht verwandten Typ, für den kein `MUL` definiert ist): der betroffene Schritt wird übersprungen (als Fehler geloggt), `OUT` behält den bis dahin akkumulierten Wert der Faltung, und `LIMIT_HIT` wird für diesen Fall **nicht** gesetzt — es ist ein Typfehler, kein numerisches Grenzwert-Ereignis.

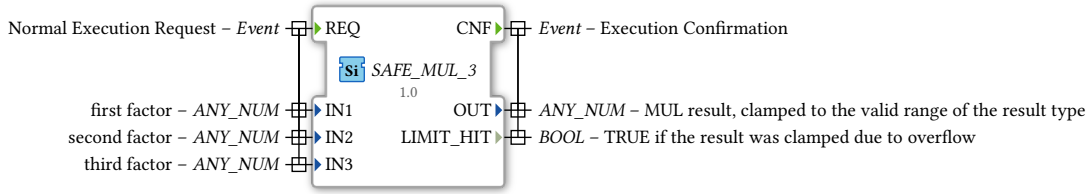
Verwandte Funktionsbausteine

- F_MUL — das normale, nicht-saturierende Standard-Pendant (stiller Wraparound/`INF` bei Overflow statt Klemmen und Melden).
- Uebung_011b3 — der Hardware-Befund, der diese Bibliothek motiviert hat.

Fazit

SAFE_MUL_2 bietet dieselbe Multiplikation mit 2 Eingängen wie `F_MUL`, aber mit einem expliziten, bereichsbewussten Grenzwert-Signal statt eines stillen Wraparounds (Integer) bzw. eines still verfälschten Ergebnisses (Fließkomma) — auf Kosten der oben dokumentierten Arität-≥-3/signed-Verkettungs-Einschränkung, die eine bewusste Design-Entscheidung ist, kein Versehen.

SAFE_MUL_3



• – ◦ • – ◦ • – ◦ •

Einleitung

SAFE_MUL_3 ist Teil der **SafeArithmetic**-Bibliothek (`SafeArithmetic::arithmetic`, EPL-2.0, HR Agrartechnik GmbH). Es ist ein generischer Funktionsbaustein für Multiplikation mit 3 Eingängen, der das feste 2-Eingänge-`F_MUL` der Standardbibliothek auf 3 Eingänge erweitert (die Standardbibliothek hat noch kein generisches, mehrstelliges `MUL_n`, anders als `ADD_2/3/4`), und einen Overflow nie unbemerkt durchlässt: bei Overflow wird das Ergebnis auf den gültigen Bereich des Ergebnistyps **geklemmt** (saturiert), und `LIMIT_HIT` zeigt an, dass geklemmt wurde.

Diese Bibliothek existiert, weil normale IEC-61131-3-Arithmetik-Funktionsbausteine (wie `F_MUL` selbst, bei Integer-Typen) bei Overflow den in IEC 61131-3 spezifizierten Zweierkomplement-Wraparound berechnen — laut Norm korrekt, aber gefährlich für reale Messwerte. `Uebung_011b3` zeigt das auf echter Hardware: das normale `F_SUB` berechnet `UDINT#1 - UDINT#12 = UDINT#4294967285`, ein stiller Unterlauf-Wraparound.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Normale Ausführungsanforderung

Ereignis-Ausgänge

- `CNF`: Ausführungsbestätigung

Daten-Eingänge

- `IN1` (ANY_NUM): erster Faktor
- `IN2` (ANY_NUM): zweiter Faktor
- `IN3` (ANY_NUM): dritter Faktor

Daten-Ausgänge

- `OUT` (ANY_NUM): MUL-Ergebnis, geklemmt auf den gültigen Bereich des Ergebnistyps
- `LIMIT_HIT` (BOOL): TRUE, wenn das Ergebnis wegen Overflow geklemmt wurde

Funktionsweise

1. **Berechnungsanforderung**: `REQ`-Ereignis mit Werten für IN1, IN2, IN3.
2. **Berechnung**: berechnet `IN1 * IN2 * IN3` von links nach rechts, Schritt für Schritt paarweise. Jeder Schritt wird auf Overflow geprüft; überläuft ein Schritt, wird sein Ergebnis auf den gültigen Bereich des Ergebnistyps geklemmt und `LIMIT_HIT` gesetzt (bleibt gesetzt — einmal von irgendeinem Schritt gesetzt, bleibt es für den Rest der Faltung gesetzt).
3. **Ergebnisausgabe**: `CNF`-Ereignis mit berechnetem `OUT` und `LIMIT_HIT`.

Overflow-/Sättigungsverhalten

- **Signed Integer**-Typen: bei Overflow Klemmen auf `numeric_limits::max()` bzw. `::min()`, Richtung abhängig von den Vorzeichen der Operanden des überlaufenden Schritts.
- **Unsigned Integer**-Typen: bei Overflow Klemmen auf `numeric_limits::max()`.
- **REAL/LREAL**: ein nicht-endliches Zwischen- oder Endergebnis wird immer erkannt, unabhängig davon, ob die *Eingänge* dieses Schritts bereits nicht-endlich waren — `+INF`/`-INF` werden auf `numeric_limits::max()`/`::lowest()` geklemmt, `NaN` (z.B. wenn ein vorheriger Schritt bereits nicht-endlich wurde und die Kombination `NaN` ergibt) wird zu einer deterministischen `0`. `LIMIT_HIT` wird in jedem Fall gesetzt.

- **Inkompatible Typen** (z.B. Mischen eines Magnitude-Typs mit einem nicht verwandten Typ, für den kein `MUL` definiert ist): der betroffene Schritt wird übersprungen (als Fehler geloggt), `OUT` behält den bis dahin akkumulierten Wert der Faltung, und `LIMIT_HIT` wird für diesen Fall **nicht** gesetzt — es ist ein Typfehler, kein numerisches Grenzwert-Ereignis.

Bekannte Einschränkung bei verketteter (Arität ≥ 3) Multiplikation mit SIGNED-Typen

`SAFE_MUL_3` wird als strikte, von links nach rechts gefaltete Kette einzelner Zweier-Operationen ausgewertet: `((IN1 * IN2) * IN3) * ...`, wobei nach jedem Schritt geklemmt wird. Bei **signed** Integer-Typen (und REAL/LREAL, die kein unsigned kennen) bedeutet das: Ein vorübergehender Zwischen-Overflow, den ein späterer Eingang eigentlich wieder in den gültigen Bereich zurückholen würde, wird dauerhaft in `OUT` festgeschrieben — inklusive eines `LIMIT_HIT`, das die wahre Gesamtsumme nicht mehr widerspiegelt.

Beispiel (INT, IN1=40000, IN2=40000, IN3=0): das wahre Ergebnis wäre `0`, locker innerhalb des INT-Bereichs, aber die Faltung klemmt bereits nach dem zweiten Schritt und erholt sich nicht mehr.

Das ist **kein** Bug, der still gefixt wird — es ist dokumentiertes, beabsichtigtes Verhalten: die Faltung wertet als reale Kette von Operationen mit fester Breite aus, genau wie das normale (nicht-sichere) `F_MUL` es auch tut, und meldet eine echte Zwischen-Grenzwertverletzung statt die Berechnung heimlich umzusortieren.

Unsigned-Ketten jeder Arität sind davon nicht betroffen und melden immer einen echten Overflow: ein späterer Faktor ist entweder 0 (das Ergebnis kollabiert dann sauber auf 0, unabhängig von vorherigem Klemmen) oder ≥ 1 (das Produkt bleibt monoton nicht-fallend) — bei unsigned kann sich das Zwischenklemmen daher nie von der wahren Gesamtsumme unterscheiden.

`SAFE_MUL_2` (nur eine einzelne Operation, keine Verkettung) meldet unabhängig vom Vorzeichen immer einen echten, korrekten Overflow — die Mehrdeutigkeit existiert nur ab Arität 3.

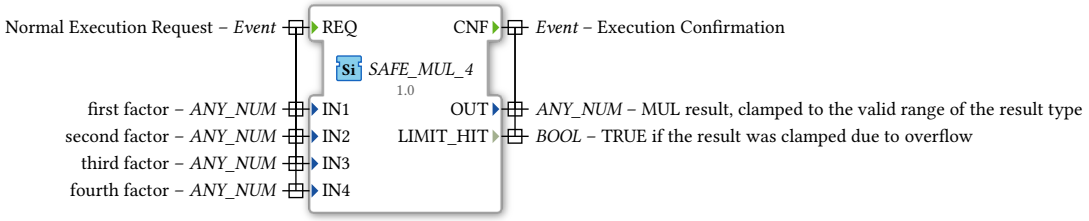
Verwandte Funktionsbausteine

- F_MUL — der feste 2-Eingänge- Standardbaustein, den dieses Muster erweitert (stiller Wraparound/`INF` bei Overflow statt Klemmen und Melden; ein 3-Eingänge-Standardpendant existiert nicht).
- Uebung_011b3 — der Hardware-Befund, der diese Bibliothek motiviert hat.

Fazit

SAFE_MUL_3 erweitert die Multiplikation von `F_MUL` auf 3 Eingänge, mit einem expliziten, bereichsbewussten Grenzwert-Signal statt eines stillen Wraparounds (Integer) bzw. eines still verfälschten Ergebnisses (Fließkomma) — auf Kosten der oben dokumentierten Arität-≥-3/signed-Verkettungs-Einschränkung, die eine bewusste Design-Entscheidung ist, kein Versehen.

SAFE_MUL_4



• – ○ • – ○ • – ○ •

Einleitung

SAFE_MUL_4 ist Teil der **SafeArithmetic**-Bibliothek (`SafeArithmetic::arithmetic`, EPL-2.0, HR Agrartechnik GmbH). Es ist ein generischer Funktionsbaustein für Multiplikation mit 4 Eingängen, der das feste 2-Eingänge-`F_MUL` der Standardbibliothek auf 4 Eingänge erweitert (die Standardbibliothek hat noch kein generisches, mehrstelliges `MUL_n`, anders als `ADD_2/3/4`), und einen Overflow nie unbemerkt durchlässt: bei Overflow wird das Ergebnis auf den gültigen Bereich des Ergebnistyps **geklemmt** (saturiert), und `LIMIT_HIT` zeigt an, dass geklemmt wurde.

Diese Bibliothek existiert, weil normale IEC-61131-3-Arithmetik-Funktionsbausteine (wie `F_MUL` selbst, bei Integer-Typen) bei Overflow den in IEC 61131-3 spezifizierten Zweierkomplement-Wraparound berechnen — laut Norm korrekt, aber gefährlich für reale Messwerte. `Uebung_011b3` zeigt das auf echter Hardware: das normale `F_SUB` berechnet `UDINT#1 - UDINT#12 = UDINT#4294967285`, ein stiller Unterlauf-Wraparound.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Normale Ausführungsanforderung

Ereignis-Ausgänge

- `CNF`: Ausführungsbestätigung

Daten-Eingänge

- `IN1` (ANY_NUM): erster Faktor
- `IN2` (ANY_NUM): zweiter Faktor
- `IN3` (ANY_NUM): dritter Faktor
- `IN4` (ANY_NUM): vierter Faktor

Daten-Ausgänge

- `OUT` (ANY_NUM): MUL-Ergebnis, geklemmt auf den gültigen Bereich des Ergebnistyps
- `LIMIT_HIT` (BOOL): TRUE, wenn das Ergebnis wegen Overflow geklemmt wurde

Funktionsweise

1. **Berechnungsanforderung**: `REQ`-Ereignis mit Werten für IN1, IN2, IN3, IN4.
2. **Berechnung**: berechnet `IN1 * IN2 * IN3 * IN4` von links nach rechts, Schritt für Schritt paarweise. Jeder Schritt wird auf Overflow geprüft; überläuft ein Schritt, wird sein Ergebnis auf den gültigen Bereich des Ergebnistyps geklemmt und `LIMIT_HIT` gesetzt (bleibt gesetzt — einmal von irgendeinem Schritt gesetzt, bleibt es für den Rest der Faltung gesetzt).
3. **Ergebnisausgabe**: `CNF`-Ereignis mit berechnetem `OUT` und `LIMIT_HIT`.

Overflow-/Sättigungsverhalten

- **Signed Integer**-Typen: bei Overflow Klemmen auf `numeric_limits::max()` bzw. `::min()`, Richtung abhängig von den Vorzeichen der Operanden des überlaufenden Schritts.
- **Unsigned Integer**-Typen: bei Overflow Klemmen auf `numeric_limits::max()`.
- **REAL/LREAL**: ein nicht-endliches Zwischen- oder Endergebnis wird immer erkannt, unabhängig davon, ob die *Eingänge* dieses Schritts bereits nicht-endlich waren — `+INF`/`-INF` werden auf `numeric_limits::max()`/`::lowest()` geklemmt, `NaN` (z.B. wenn ein vorheriger Schritt bereits nicht-endlich wurde und die Kombination `NaN` ergibt) wird zu einer deterministischen `0`. `LIMIT_HIT` wird in jedem Fall gesetzt.
- **Inkompatible Typen** (z.B. Mischen eines Magnitude-Typs mit einem nicht verwandten Typ, für den kein `MUL` definiert ist): der betroffene Schritt wird übersprungen (als Fehler geloggt), `OUT` behält den bis dahin akkumulierten Wert der Faltung, und `LIMIT_HIT` wird für diesen Fall **nicht** gesetzt — es ist ein Typfehler, kein numerisches Grenzwert-Ereignis.

Bekannte Einschränkung bei verketteter (Arität ≥ 3) Multiplikation mit SIGNED-Typen

`SAFE_MUL_4` wird als strikte, von links nach rechts gefaltete Kette einzelner Zweier-Operationen ausgewertet: `((IN1 * IN2) * IN3) * ...`, wobei nach jedem Schritt geklemmt wird. Bei **signed** Integer-Typen (und REAL/LREAL, die kein unsigned kennen) bedeutet das: Ein vorübergehender Zwischen-Overflow, den ein späterer Eingang eigentlich wieder in den gültigen Bereich zurückholen würde, wird dauerhaft in `OUT` festgeschrieben — inklusive eines `LIMIT_HIT`, das die wahre Gesamtsumme nicht mehr widerspiegelt.

Beispiel (INT, IN1=40000, IN2=40000, IN3=0, IN4=1): das wahre Ergebnis wäre `0`, locker innerhalb des INT-Bereichs, aber die Faltung klemmt bereits nach dem zweiten Schritt und erholt sich nicht mehr.

Das ist **kein** Bug, der still gefixt wird — es ist dokumentiertes, beabsichtigtes Verhalten: die Faltung wertet als reale Kette von Operationen mit fester Breite aus, genau wie das normale (nicht-sichere) `F_MUL` es auch tut, und meldet eine echte Zwischen-Grenzwertverletzung statt die Berechnung heimlich umzusortieren.

Unsigned-Ketten jeder Arität sind davon nicht betroffen und melden immer einen echten Overflow: ein späterer Faktor ist entweder 0 (das Ergebnis kollabiert dann sauber auf 0, unabhängig von vorherigem Klemmen) oder ≥ 1 (das Produkt bleibt monoton nicht-fallend) — bei unsigned kann sich das Zwischenklemmen daher nie von der wahren Gesamtsumme unterscheiden.

`SAFE_MUL_2` (nur eine einzelne Operation, keine Verkettung) meldet unabhängig vom Vorzeichen immer einen echten, korrekten Overflow — die Mehrdeutigkeit existiert nur ab Arität 3.

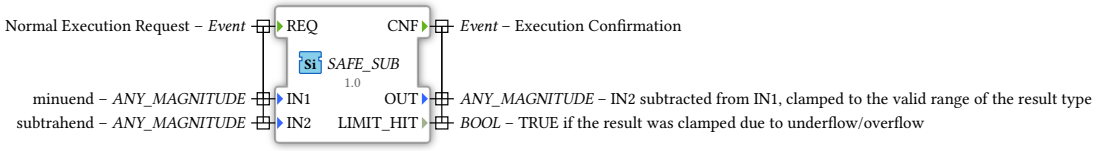
Verwandte Funktionsbausteine

- F_MUL — der feste 2-Eingänge- Standardbaustein, den dieses Muster erweitert (stiller Wraparound/`INF` bei Overflow statt Klemmen und Melden; ein 4-Eingänge-Standardpendant existiert nicht).
- Uebung_011b3 — der Hardware-Befund, der diese Bibliothek motiviert hat.

Fazit

SAFE_MUL_4 erweitert die Multiplikation von `F_MUL` auf 4 Eingänge, mit einem expliziten, bereichsbewussten Grenzwert-Signal statt eines stillen Wraparounds (Integer) bzw. eines still verfälschten Ergebnisses (Fließkomma) — auf Kosten der oben dokumentierten Arität-≥-3/signed-Verkettungs-Einschränkung, die eine bewusste Design-Entscheidung ist, kein Versehen.

SAFE_SUB



• – ◦ • – ◦ • – ◦ •

Einleitung

SAFE_SUB ist Teil der **SafeArithmetic**-Bibliothek (`SafeArithmetic::arithmetic`, EPL-2.0, HR Agrartechnik GmbH). Es ist ein fester Subtraktions-Funktionsbaustein mit 2 Eingängen, der sich wie das Standard-`F_SUB` verhält, aber einen Unter-/Überlauf nie unbemerkt durchlässt: bei einer Grenzwertverletzung wird das Ergebnis auf den gültigen Bereich des Ergebnistyps **geklemmt** (saturiert), und `LIMIT_HIT` zeigt an, dass geklemmt wurde.

Das ist der Baustein, der die ganze Bibliothek ausgelöst hat: `Uebung_011b3` zeigt auf echter ESP32-P4-Hardware, dass das normale `F_SUB UDINT#1 - UDINT#12 = UDINT#4294967285` berechnet — genau der in IEC 61131-3 spezifizierte Zweierkomplement-Wraparound für unsigned-Typen, aber gefährlich für reale Messwerte (Sollwertdifferenzen, Restweg-Berechnungen). `SAFE_SUB` berechnet dieselbe Subtraktion stattdessen als `UDINT#0` mit `LIMIT_HIT = TRUE`.

Anders als `SAFE_ADD_2/3/4/SAFE_MUL_2/3/4` ist `SAFE_SUB` **nicht** generisch in der Arität: Subtraktion ist weder assoziativ noch kommutativ, sodass ein N-stelliges, „erweiterbares“ `SAFE_SUB_n` keine wohldefinierte Bedeutung hätte, wie es bei erweiterbarem ADD/MUL der Fall ist.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Normale Ausführungsanforderung

Ereignis-Ausgänge

- `CNF`: Ausführungsbestätigung

Daten-Eingänge

- `IN1` (ANY_MAGNITUDE): Minuend
- `IN2` (ANY_MAGNITUDE): Subtrahend

Daten-Ausgänge

- `OUT` (ANY_MAGNITUDE): IN2 von IN1 subtrahiert, geklemmt auf den gültigen Bereich des Ergebnistyps
- `LIMIT_HIT` (BOOL): TRUE, wenn das Ergebnis wegen Unter-/Überlauf geklemmt wurde

Funktionsweise

1. **Berechnungsanforderung**: `REQ`-Ereignis mit Werten für IN1 und IN2.
2. **Subtraktion**: berechnet `IN1 - IN2`.
3. **Ergebnisausgabe**: `CNF`-Ereignis mit berechnetem `OUT` und `LIMIT_HIT`.

Über-/Unterlaufverhalten

- **Signed Integer**-Typen: bei Über-/Unterlauf Klemmen auf `numeric_limits::min()` bzw. `::max()`, Richtung abhängig vom Vorzeichen von IN2.

- **Unsigned Integer**-Typen: Subtraktion kann nur unterlaufen (unter 0 gehen), daher wird immer auf `0` geklemmt — genau der `Uebung_011b3`-Fall: `UDINT#1 - UDINT#12` klemmt auf `UDINT#0` mit `LIMIT_HIT = TRUE`, statt auf `UDINT#4294967285` überzulaufen.
- **REAL/LREAL**: ein nicht-endliches Zwischen- oder Endergebnis wird immer erkannt, unabhängig davon, ob die *Eingänge* bereits nicht-endlich waren — `+INF`/`-INF` werden auf `numeric_limits::max()`/`::lowest()` geklemmt, `NaN` (z.B. `INF - INF`) wird zu einer deterministischen `0`. `LIMIT_HIT` wird in jedem Fall gesetzt.
- **Inkompatible Typen** (z.B. Mischen eines Magnitude-Typs mit einem nicht verwandten Typ, für den keine Subtraktion definiert ist): das Ergebnis dieser Operation ist undefiniert (als Fehler geloggt), `OUT` bleibt unverändert, und `LIMIT_HIT` wird für diesen Fall **nicht** gesetzt — es ist ein Typfehler, kein numerisches Grenzwert-Ereignis.

Verwandte Funktionsbausteine

- F_SUB — das normale, nicht-saturierende Standard-Pendant (stiller Wraparound/`INF` bei Overflow statt Klemmen und Melden).
- Uebung_011b3 — der Hardware-Befund, der diese Bibliothek motiviert hat.
- Uebung_011b5 — dasselbe `1 - 12`-Szenario, erneut ausgeführt mit `SAFE_SUB`, mit `LIMIT_HIT` auf einem zweiten Ausgang verdrahtet, damit das Klemmen auf der Hardware sichtbar wird.

Fazit

SAFE_SUB bietet dieselbe Subtraktion wie `F_SUB`, aber mit einem expliziten, bereichsbewussten Grenzwert-Signal statt eines stillen Wraparounds (Integer) bzw. eines still verfälschten Ergebnisses (Fließkomma) — und schließt damit direkt die Lücke, die `Uebung_011b3` in der Standardbibliothek gefunden hat.

2.1.11 adapter

2.1.11.1 🎧 Podcast

- Code-Renovierung mit AX-Adaptern: Wie Eclipse 4diac™ durch Signal-Bündelung Komplexität besiegt
- Von 1400 Fehlern zum sauberen Code: Die Migration der „Getreidehacke“ auf Eclipse 4diac™ 3.0 und die Macht der AX-Adapter

---

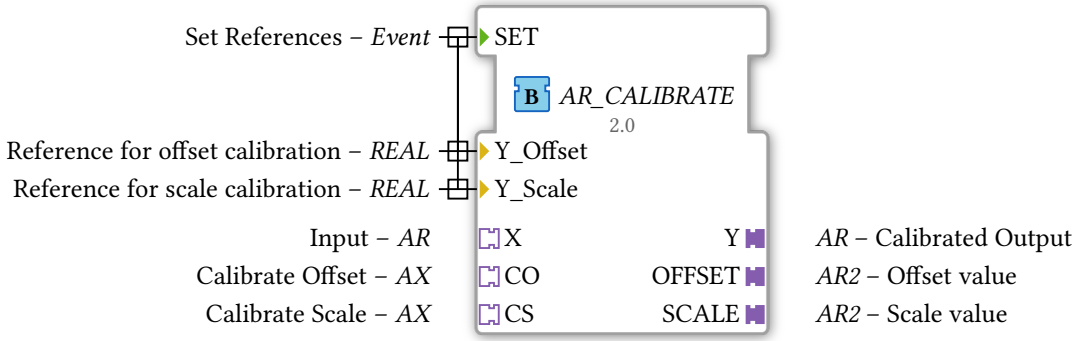
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

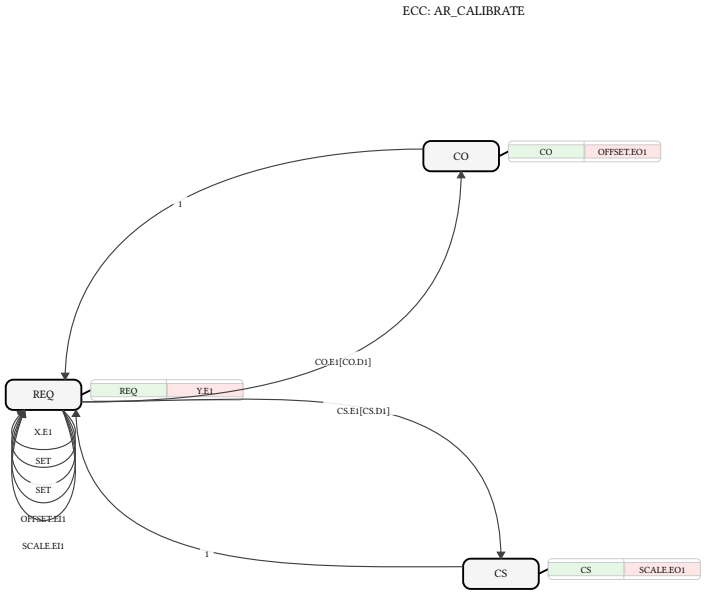
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.1.11.2 Engineering

measurements

AR_CALIBRATE





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `AR_CALIBRATE` dient der Offset- und Skalenkalibrierung eines analogen Eingangssignals, das über einen Adapter bereitgestellt wird. Er ermöglicht eine zweistufige Kalibrierung: Zunächst wird der Offset durch Vergleich mit einem Referenzwert bei aktiver Kalibrierung (`CO`) ermittelt, anschließend die Skalierung über eine zweite Referenz (`CS`). Das kalibrierte Ausgangssignal wird kontinuierlich berechnet.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Mit	Beschreibung
`SET`	`EInit`	`Y_Offset`, `Y_Scale`	Setzt die Referenzwerte für Offset- und Skalenkalibrierung
`CO`	`Event`		Atomarer Trigger für die Offset-Kalibrierung (liest aktuellen Rohwert von `X`)
`CS`	`Event`		Atomarer Trigger für die Skalen-Kalibrierung (liest aktuellen Rohwert von `X`)

Ereignis-Ausgänge

Der Funktionsblock besitzt keine eigenen Ereignis-Ausgänge. Ereignisse werden jedoch über die angeschlossenen Adapter gesendet (z. B. `Y.E1`, `OFFSET.EO1`, `SCALE.EO1`).

Daten-Eingänge

Name	Typ	Beschreibung
`Y_Offset`	`REAL`	Referenzwert für die Offset-Kalibrierung
`Y_Scale`	`REAL`	Referenzwert für die Skalen-Kalibrierung

Daten-Ausgänge

Der FB hat keine direkten Daten-Ausgänge. Die berechneten Werte werden über die Adapter `Y`, `OFFSET` und `SCALE` zur Verfügung gestellt.

Adapter

Plugs (bereitstellende Schnittstellen):

Adapter	Typ	Beschreibung
`Y`	`adapter::types::unidirectional::AR`	Kalibrierter Ausgang (Daten + Ereignis)
`OFFSET`	`adapter::types::bidirectional::AR2`	Liefert den berechneten Offsetwert (bidirektional)
`SCALE`	`adapter::types::bidirectional::AR2`	Liefert den berechneten Skalenfaktor (bidirektional)

Sockets (nutzende Schnittstellen):

Adapter	Typ	Beschreibung
`X`	`adapter::types::unidirectional::AR`	Analoger Eingangswert (unidirektional)

Funktionsweise

Der FB arbeitet mit drei Basisalgorithmen, die in den Zuständen `REQ`, `CO` und `CS` ausgeführt werden:

- **REQ** (Normalbetrieb):
  `Y.D1 := (X.D1 + OFFSET.DI1) * SCALE.DI1`
  Der Ausgangswert wird aus dem Eingang, dem aktuellen Offset und dem Skalenfaktor berechnet.
- **CO** (Offset-Kalibrierung):
  `OFFSET.DO1 := Y_Offset - X.D1`
  Der Offset wird als Differenz zwischen dem Referenzwert `Y_Offset` und dem aktuellen Eingangswert ermittelt.
- **CS** (Skalen-Kalibrierung):
  `SCALE.DO1 := Y_Scale / (X.D1 + OFFSET.DI1)`
  Der Skalenfaktor ergibt sich aus dem Referenzwert `Y_Scale` geteilt durch den um den Offset korrigierten Eingangswert.

Ablauf der Kalibrierung:

1. Im Zustand `REQ` wird der FB durch das Eintreffen des Ereignisses `CO` in den Zustand `CO` überführt. Dort wird der Offset berechnet und über den Adapter `OFFSET` ausgegeben. Anschließend kehrt der FB sofort nach `REQ` zurück.
2. Analog erfolgt die Skalenkalibrierung über das Ereignis `CS` und den Zustand `CS`.
3. Das Ereignis `SET` aktualisiert die Referenzwerte `Y_Offset` und `Y_Scale`, ohne den Kalibrierungszustand zu verlassen.
4. Wiederholte Messungen (über `X.E1`) aktualisieren den Ausgang `Y` mit den aktuellen Kalibrierparametern.

Die Übergänge zwischen den Zuständen sind wie folgt definiert:

- `REQ` ⟶ `REQ`: bei `X.E1`, `SET`, `OFFSET.EI1` oder `SCALE.EI1`
- `REQ` ⟶ `CO`: wenn Ereignis `CO` empfangen wird
- `REQ` ⟶ `CS`: wenn Ereignis `CS` empfangen wird
- `CO` ⟶ `REQ`: immer (Bedingung 1)
- `CS` ⟶ `REQ`: immer (Bedingung 1)

Technische Besonderheiten

- **Atomare Ereignisauslösung**: Die Kalibrierbefehle `CO` und `CS` sind als direkte Ereigniseingänge ausgeführt. Dadurch entfallen bisherige Umwege über AX-Adapter und Flankenwächter-Guards.
- **Bidirektionale Kalibrieradapter**: Die Adapter `OFFSET` und `SCALE` sind bidirektional ausgelegt (`AR2`), sodass sie sowohl vom Kalibrier-FB beschrieben als auch von externen Bausteinen gelesen werden können.

- **Zweistufige Kalibrierung**: Offset und Skalierung werden nacheinander kalibriert. Der Skalenfaktor verwendet bereits den ermittelten Offset, um eine korrekte lineare Korrektur zu gewährleisten.
- **Keine eigene Ereignisausgabe**: Der FB erzeugt keine eigenen Ereignisse, sondern löst über die Adapter-Ereignisse aus (z. B. `Y.E1` nach jeder Berechnung).

Zustandsübersicht

Zustand	Beschreibung	Aktion
REQ	Normalbetrieb – Berechnung des kalibrierten Ausgangs	Führt Algorithmus `REQ` aus, sendet Ereignis auf `Y.E1`
CO	Offset-Kalibrierung	Führt Algorithmus `CO` aus, sendet Ereignis auf `OFFSET.EO1`
CS	Skalen-Kalibrierung	Führt Algorithmus `CS` aus, sendet Ereignis auf `SCALE.EO1`

Anwendungsszenarien

- **Industrielle Messwertverarbeitung**: Kalibrierung von Drucksensoren, Temperaturfühlern oder anderen analogen Gebern, bei denen Offset und Verstärkung nachjustiert werden müssen.
- **Einmessung von Messketten**: Nach dem Anschluss eines Sensors kann durch Anlegen eines bekannten Nullpunkts (Offset) und eines bekannten Messwerts (Skalierung) die gesamte Kette linearisiert werden.
- **Automatische Nachkalibrierung**: Durch gezieltes Auslösen der Ereigniseingänge `CO` und `CS` können Drift-Effekte kompensiert werden.

Vergleich mit ähnlichen Bausteinen

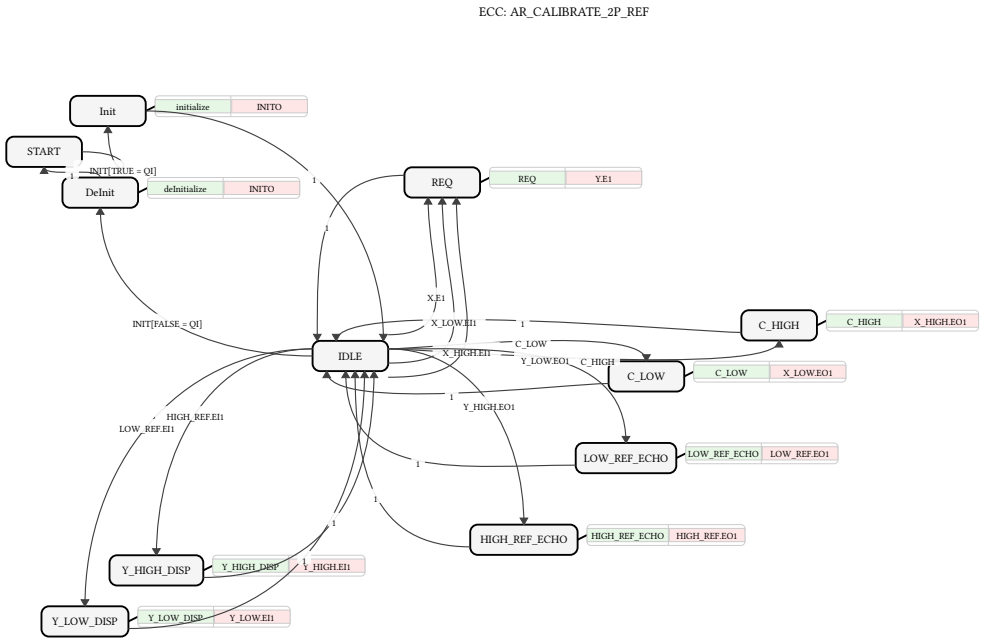
Details zum Kalibrierverfahren finden Sie unter Kalibrierverfahren Zwei- und Dreipunkt.

Einfache Skalierungsbausteine (z. B. `SCALE`) bieten nur eine feste Multiplikation und Addition, ohne automatische Kalibrierung. Der `AR_CALIBRATE` hingegen integriert die gesamte Kalibrierlogik – inklusive Referenzvorgabe und ereignisgesteuerter Aktivierung – und eignet sich daher für dynamische Umgebungen, in denen Parameter während des Betriebs angepasst werden müssen.

Fazit

Der Funktionsblock `AR_CALIBRATE` stellt eine robuste Lösung zur Offset- und Skalenkalibrierung analoger Signale dar. Dank der atomaren Ereigniseingänge und der klaren Zustandsmaschine lässt er sich leicht in bestehende Automatisierungslösungen integrieren und ermöglicht eine präzise, wiederholbare Messwertkorrektur.

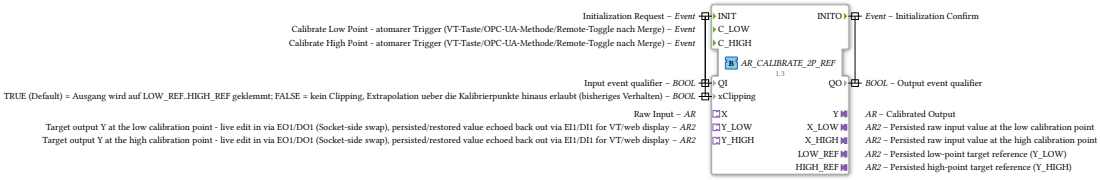
AR_CALIBRATE_2P_REF



• — ○ • — ○ • — ○ •

Einleitung

`AR_CALIBRATE_2P_REF` ist eine adapterbasierte 2-Punkt-Kalibrierung (Low/High). Beide Punkte werden unabhängig voneinander gespeichert, es gibt keine Reihenfolgeabhängigkeit wie bei `AR_CALIBRATE_SQ_REF`. Die Referenzziele (`Y_LOW`/`Y_HIGH`) sind lebende `AR2`-Sockets und werden über zwei weitere `AR2`-Plugs (`LOW_REF`/`HIGH_REF`) persistiert. Die Interpolation erfolgt erst live bei jedem `X.E1`.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierung bzw. Deinitialisierung. Verbundene Daten: `QI`, `xClipping`.
- `C_LOW`: Atomarer Trigger zum Kalibrieren des Low-Punkts (VT-Taste, OPC-UA-Methode oder Remote-Toggle nach Merge).
- `C_HIGH`: Atomarer Trigger zum Kalibrieren des High-Punkts.

Ereignis-Ausgänge

- `INIT0`: Initialisierungsbestätigung. Verbundene Daten: `QO`.

Daten-Eingänge

Name	Datentyp	Vorgabewert	Kommentar
`QI`	`BOOL`		Eingangs-Qualifier (TRUE = aktivieren, FALSE = deaktivieren).

xClipping	BOOL	TRUE	TRUE: Ausgang wird auf LOW_REF..HIGH_REF geklemmt; FALSE: kein Clipping, Extrapolation über die Kalibrierpunkte hinaus erlaubt.

Daten-Ausgänge

- QO (BOOL): Ausgangs-Qualifier.

Adapter

Richtung	Name	Adaptertyp	Kommentar
Plug (Ausgang)	Y	adapter::types::unidirectional::AR1	Kalibrierter Ausgang.
Plug (Ausgang)	X_LOW	adapter::types::bidirectional::AR2	Gespeicherter Rohwert am Low-Kalibrierpunkt.
Plug (Ausgang)	X_HIGH	adapter::types::bidirectional::AR2	Gespeicherter Rohwert am High-Kalibrierpunkt.
Plug (Ausgang)	LOW_REF	adapter::types::bidirectional::AR2	Gespeichertes Low-Ziel (Y_LOW).
Plug (Ausgang)	HIGH_REF	adapter::types::bidirectional::AR2	Gespeichertes High-Ziel (Y_HIGH).
Socket (Eingang)	X	adapter::types::unidirectional::AR1	Rohwert.
Socket (Eingang)	Y_LOW	adapter::types::bidirectional::AR2	Ziel-Ausgangswert am Low-Punkt: Live-Eingabe über EO1/DO1, der gespeicherte bzw. wiederhergestellte Wert wird über EI1/DI1 für VT/Web zurückgespiegelt.
Socket (Eingang)	Y_HIGH	adapter::types::bidirectional::AR2	Wie Y_LOW, für den High-Punkt.

Funktionsweise

Der Baustein ist ein Basis-FB. Im Zustand IDLE reagiert er auf folgende Ereignisse:

Ereignis	Zustand	Aktion
X.EI1, X_LOW.EI1, X_HIGH.EI1	REQ	Berechnet den kalibrierten Wert und gibt Y.EI1 aus.
C_LOW	C_LOW	X_LOW.DO1 := X.D1, Ausgabe X_LOW.EO1.
C_HIGH	C_HIGH	X_HIGH.DO1 := X.D1, Ausgabe X_HIGH.EO1.
Y_LOW.EO1	LOW_REF_ECHO	LOW_REF.DO1 := Y_LOW.DO1, Ausgabe LOW_REF.EO1 (persistieren).
Y_HIGH.EO1	HIGH_REF_ECHO	HIGH_REF.DO1 := Y_HIGH.DO1, Ausgabe HIGH_REF.EO1.
LOW_REF.EI1	Y_LOW_DISP	Y_LOW.DI1 := LOW_REF.DI1, Ausgabe Y_LOW.EI1 (Anzeige nach Boot und jeder Änderung).
HIGH_REF.EI1	Y_HIGH_DISP	Y_HIGH.DI1 := HIGH_REF.DI1, Ausgabe Y_HIGH.EI1.

Alle Aktionszustände kehren unmittelbar nach IDLE zurück. Die Initialisierung läuft über START → Init (INIT mit QI = TRUE, QO := QI, INITO) → IDLE; INIT mit QI = FALSE führt aus IDLE über DeInit (QO := FALSE, INITO) zurück nach START.

Berechnung im Zustand REQ:

1. QO := QI. Nur bei QI = TRUE wird gerechnet.

2. Intern gelatchte Flags: `xLowCal` bzw. `xHighCal` werden TRUE, sobald `X_LOW.DI1` bzw. `X_HIGH.DI1` vom Default `0.0` abweicht.
3. Sind beide Punkte kalibriert: `Y.D1 = LOW_REF + (X - X_LOW) * (HIGH_REF - LOW_REF) / (X_HIGH - X_LOW)`; bei `X_HIGH = X_LOW` wird `LOW_REF` ausgegeben.
4. Ist nur Low kalibriert, wird `LOW_REF` ausgegeben; ist nur High kalibriert, `HIGH_REF`; ist nichts kalibriert, `LOW_REF`.
5. Clipping auf `[LOW_REF, HIGH_REF]` nur, wenn `xClipping` TRUE ist und beide Punkte kalibriert sind.

Technische Besonderheiten

- **Keine Reihenfolgeabhängigkeit:** `C_LOW` und `C_HIGH` speichern ihren Rohwert unabhängig, jeder Punkt kann jederzeit einzeln neu kalibriert werden.
- **Teilkalibrierung:** Solange nur ein Punkt kalibriert ist, wird nicht gegen den noch auf `0.0` stehenden anderen Punkt interpoliert, sondern der bekannte REF-Wert ausgegeben.
- **Persistenz:** `X_LOW`, `X_HIGH`, `LOW_REF` und `HIGH_REF` sind bidirektionale `AR2`-Plugs; die dauerhafte Speicherung übernimmt ein nachgeschalteter Baustein (z. B. `INI_AR2/NVS_AR2`). Beim Boot zurückgelesene Werte erscheinen über `LOW_REF.EI1/HIGH_REF.EI1` wieder an `Y_LOW/Y_HIGH`.
- **Atomare Trigger:** `C_LOW/C_HIGH` sind reine Ereigniseingänge, kein `AX`-Adapter mit Flankenwächter.

Anwendungsszenarien

- Kalibrierung eines Analogsensors an zwei Referenzpunkten (z. B. Null- und Endwert) mit VT-Eingabe der Zielwerte.
- Nachträgliches Nachtrimmen eines gedrifteten Nullpunkts, ohne den High-Punkt erneut zu kalibrieren.

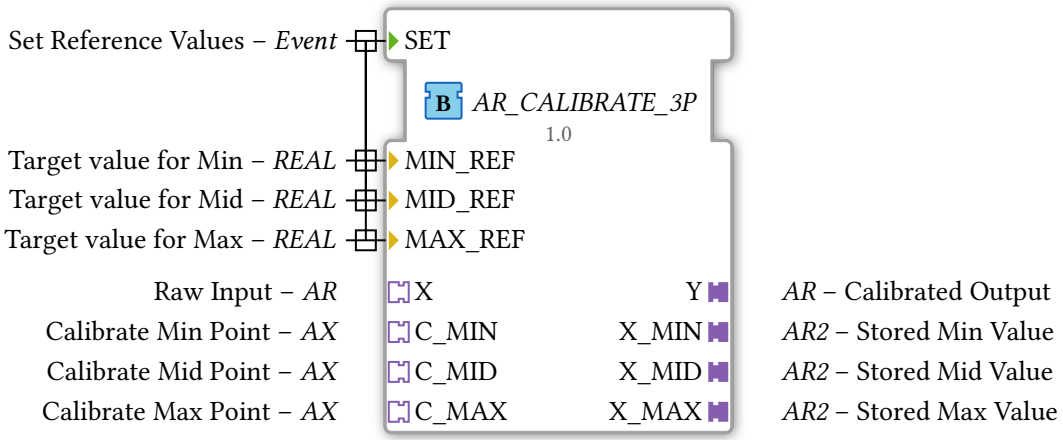
Zusammenfassung

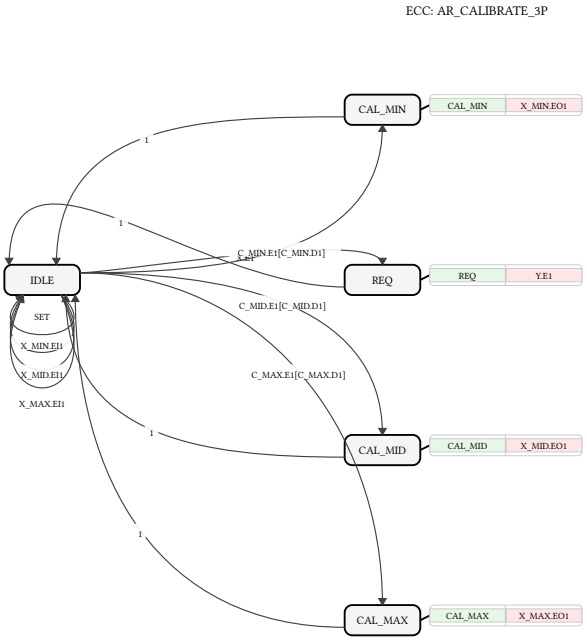
`AR_CALIBRATE_2P_REF` kalibriert ein `AR`-Signal über zwei unabhängig gespeicherte Rohwerte und zwei live editierbare, persistierbare Referenzziele und gibt das linear interpolierte (optional geklemmte) Ergebnis über `Y` aus.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AR_CALIBRATE_3P





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AR_CALIBRATE_3P** ermöglicht eine 3-Punkt-Kalibrierung eines analogen Eingangssignals unter Verwendung von Adaptern. Er ist speziell für Joysticks ausgelegt, die einen Mittenversatz (Center-Drift) aufweisen, und korrigiert diesen durch Linearisierung zwischen drei Referenzpunkten: Minimum, Mittelwert und Maximum. Die Kalibrierungspunkte werden gespeichert und können bei Bedarf neu eingestellt werden.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Mit	Kommentar
`SET`	`Event`	`MIN_REF`, `MID_REF`, `MAX_REF`	Setzt die Referenzwerte für die Kalibrierungskennlinie. Löst keine Berechnung aus, sondern legt nur die Zielausgabewerte fest.
`C_MIN`	`Event`		Atomarer Trigger zum Kalibrieren des Minimalpunkts (liest den aktuellen Rohwert von `X`).
`C_MID`	`Event`		Atomarer Trigger zum Kalibrieren des Mittelpunkts (liest den aktuellen Rohwert von `X`).
`C_MAX`	`Event`		Atomarer Trigger zum Kalibrieren des Maximalpunkts (liest den aktuellen Rohwert von `X`).

Ereignis-Ausgänge

Keine expliziten Ereignisausgänge vorhanden. Die Ausgabe erfolgt ausschließlich über den Adapter **Y**.

Daten-Eingänge

Name	Datentyp	Vorgabewert	Kommentar
`MIN_REF`	`REAL`	`0.0`	Zielwert für den kleinsten Eingangswert (Min).

MID_REF	REAL	50.0	Zielwert für den Mittelpunkt (Mid).
MAX_REF	REAL	100.0	Zielwert für den größten Eingangswert (Max).

Daten-Ausgänge

Keine direkten Datenausgänge – alle Ausgaben werden über die **Plugs** (Ausgangsadapter) bereitgestellt.

Adapter

Richtung	Name	Adaptertyp	Kommentar
Plug (Ausgang)	Y	adapter::types::unidirectional::AR	Kalibrierter Ausgabewert (Analogwert plus Ereignis).
Plug (Ausgang)	X_MIN	adapter::types::bidirectional::AR2	Gespeicherter Minimalwert (vom Rohwert).
Plug (Ausgang)	X_MID	adapter::types::bidirectional::AR2	Gespeicherter Mittelpunkt (vom Rohwert).
Plug (Ausgang)	X_MAX	adapter::types::bidirectional::AR2	Gespeicherter Maximalwert (vom Rohwert).
Socket (Eingang)	X	adapter::types::unidirectional::AR	Rohwert vom Sensor (Analogwert plus Ereignis).

Funktionsweise

Die Kalibrierung basiert auf einer stückweisen linearen Interpolation zwischen drei gespeicherten Rohwerten (X_MIN, X_MID, X_MAX) und den zugehörigen Referenzwerten (MIN_REF, MID_REF, MAX_REF).

1. **Kalibrierung der Punkte:**
   Ein Ereignis an einem der Kalibrierungseingänge (C_MIN, C_MID, C_MAX) speichert den gerade anliegenden Rohwert (X.D1) in den entsprechenden gespeicherten Wert (X_MIN.DO1, X_MID.DO1, X_MAX.DO1).
2. **Berechnung des kalibrierten Werts:**
   Sobald ein Ereignis vom Rohwert-Adapter (X.E1) eintrifft, wird der Funktionsblock aktiv und führt den Algorithmus **REQ** aus. Dabei wird der Rohwert X.D1 linear abgebildet:
   - Liegt der Rohwert unterhalb des gespeicherten Mittelpunkts X_MID.DI1, wird der untere Zweig der Kennlinie verwendet:
     `Y.D1 = MIN_REF + (X.D1 – X_MIN.DI1) * (MID_REF – MIN_REF) / (X_MID.DI1 – X_MIN.DI1)`
     Falls die Intervalle ungültig sind (Division durch Null oder negative Spannweite), wird auf MIN_REF zurückgegriffen.
   - Liegt der Rohwert oberhalb oder gleich X_MID.DI1, wird der obere Zweig berechnet:
     `Y.D1 = MID_REF + (X.D1 – X_MID.DI1) * (MAX_REF – MID_REF) / (X_MAX.DI1 – X_MID.DI1)`
     Auch hier wird bei ungültigen Intervallen MID_REF ausgegeben.
3. **Begrenzung (Clipping):**
   Der berechnete Ausgabewert wird auf das Intervall `[MIN_REF, MAX_REF]` begrenzt, um physikalisch sinnvolle Ergebnisse sicherzustellen.
4. **Ausgabe:**
   Der kalibrierte Wert wird über den Adapter Y (Ereignis Y.E1 und Data Y.D1) ausgegeben.

Technische Besonderheiten

- **Bidirektionale Adapter für Kalibrierungspunkte:** Die gespeicherten Rohwerte (X_MIN, X_MID, X_MAX) sind bidirektionale Adapter vom Typ AR2. Sie können sowohl beschrieben (während der Kalibrierung) als auch gelesen (während der Berechnung) werden. Dadurch stehen die Kalibrierungspunkte für nachfolgende Berechnungen zur Verfügung (eine dauerhafte Speicherung über Reinitialisierung oder Stromausfall hinaus erfordert ein nachgeschaltetes Speichermodul wie INI/NVS).
- **Atomare Ereignisauslösung:** Die Kalibrierung der drei Punkte erfolgt direkt über spezifische Ereigniseingänge (C_MIN, C_MID, C_MAX) ohne vorgeschaltete AX-Adapter oder Flankenwächter-Guards.

- **Schutz vor ungültigen Intervallen:** Die Algorithmen prüfen, ob die Spannweiten der gespeicherten Rohwerte positiv sind. Falls nicht (z. B. bei noch nicht kalibriertem Zustand), werden sichere Standardwerte ausgegeben.
- **Keine Selbstkalibrierung:** Der FB speichert keine Historie – das System muss die Kalibrierungspunkte explizit durch Auslösen der Kalibrierereignisse setzen.

Zustandsübersicht

Zustand	Beschreibung
IDLE	Wartezustand. Transitionen: Bei `SET` → IDLE; bei `X_MIN.EI1`, `X_MID.EI1`, `X_MAX.EI1` → IDLE; bei `C_MIN` → CAL_MIN; bei `C_MID` → CAL_MID; bei `C_MAX` → CAL_MAX; bei `X.E1` → REQ.
REQ	Berechnung des kalibrierten Ausgangswerts. Nach Ausführung sofort zurück zu IDLE.
CAL_MIN	Speichert den aktuellen Rohwert als Minimum (`X_MIN.DO1 := X.D1`). Kehrt automatisch zu IDLE zurück.
CAL_MID	Speichert den aktuellen Rohwert als Mittelpunkt (`X_MID.DO1 := X.D1`). Kehrt automatisch zu IDLE zurück.
CAL_MAX	Speichert den aktuellen Rohwert als Maximum (`X_MAX.DO1 := X.D1`). Kehrt automatisch zu IDLE zurück.

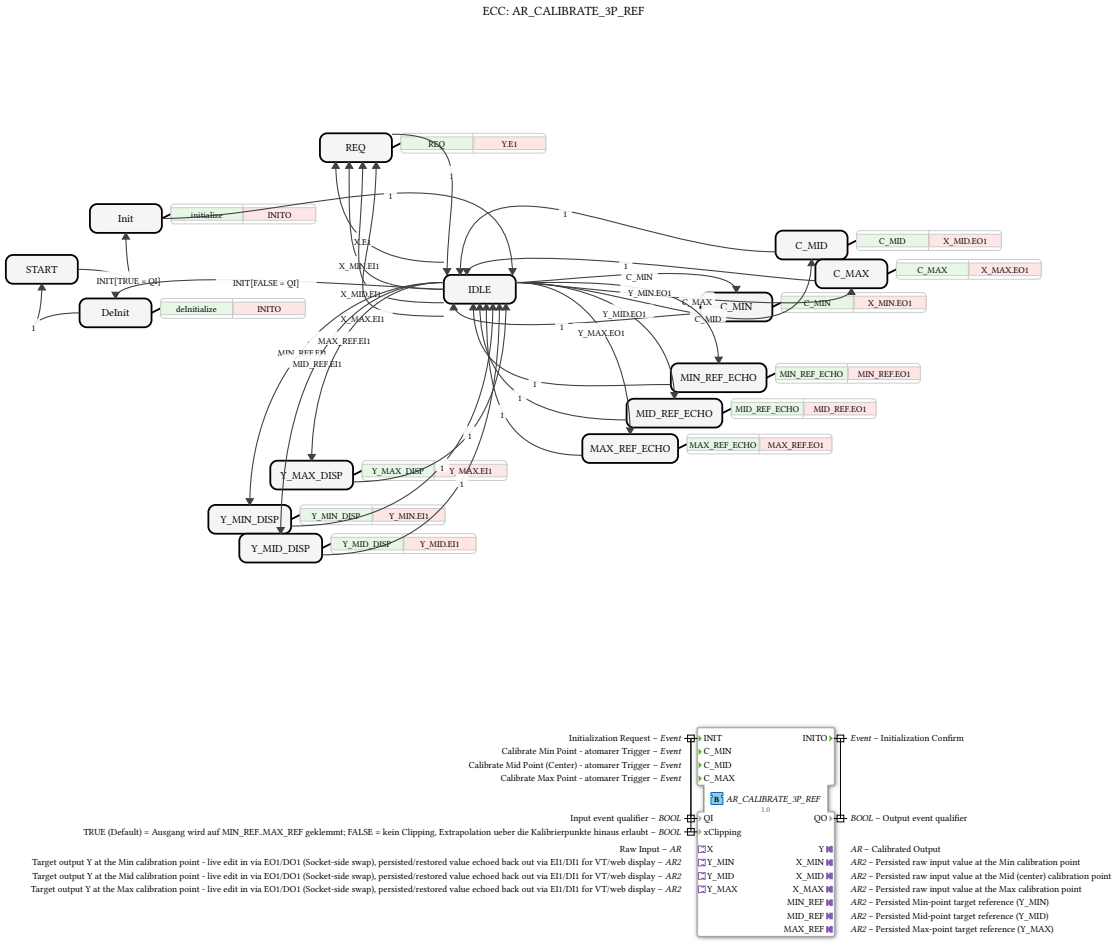
Übergangsbedingungen:

- `X.E1` → Start der Berechnung
- `C_MIN` → Kalibrierung des Minimalpunkts
- `C_MID` → Kalibrierung des Mittelpunkts
- `C_MAX` → Kalibrierung des Maximalpunkts
- `SET`, `X_MIN.EI1`, `X_MID.EI1`, `X_MAX.EI1` → Keine Zustandsänderung (bleibt in IDLE)

Anwendungsszenarien

- **Joystick-Kalibrierung:** Ein Joystick mit analogem Ausgang (z. B. 0-10 V) weist bauteilbedingte Abweichungen in der Mitte und an den Endanschlägen auf. Der Bediener fährt den Joystick an die drei Positionen (Min, Mitte, Max) und löst über Taster die Kalibrierungsereignisse aus. Danach liefert `Y` einen linearisierten, auf die gewünschten Zielwerte normierten Wert.
- **Analoges Potentiometer:** Ein Schleifpotentiometer mit Abnutzungserscheinungen kann durch 3-Punkt-Kalibrierung korrigiert werden.

AR_CALIBRATE_3P_REF



• – ○ • – ○ • – ○ •

Einleitung

`AR_CALIBRATE_3P_REF` ist eine adapterbasierte Drei-Punkt-Kalibrierung für Joysticks und Sensoren mit Mittelstellung (Min, Mid, Max), in freier Reihenfolge wie AR_CALIBRATE_3P. Die Referenz-Zielwerte (`Y_MIN`/`Y_MID`/`Y_MAX`) sind **lebende `AR2`-Sockets** und werden über drei zusätzliche `AR2`-Plugs (`MIN_REF`/`MID_REF`/`MAX_REF`) persistiert. Es gilt dasselbe Live-Echo-Muster wie bei `AR_CALIBRATE_SQ_REF` und AR_CALIBRATE_2P_REF.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung. Verknüpft mit `QI`, `xClipping`.
- **C_MIN**: Minimalpunkt kalibrieren, atomarer Trigger.
- **C_MID**: Mittelpunkt (Center) kalibrieren, atomarer Trigger.
- **C_MAX**: Maximalpunkt kalibrieren, atomarer Trigger.

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung. Verknüpft mit `QO`.

Daten-Eingänge

- **QI** (`BOOL`): Ereignis-Eingangsqualifizierer
- **xClipping** (`BOOL`, Default `TRUE`): `TRUE` = Ausgang wird auf `MIN_REF..MAX_REF` geklemmt; `FALSE` = kein Clipping, Extrapolation über die Kalibrierpunkte hinaus erlaubt

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifizierer

Adapter

Plugs:

- **Y** (adapter::types::unidirectional::AR): kalibrierter Ausgang
- **X_MIN**, **X_MID**, **X_MAX** (adapter::types::bidirectional::AR2): persistierter Rohwert am Min-/Mid-/Max-Kalibrierpunkt
- **MIN_REF**, **MID_REF**, **MAX_REF** (adapter::types::bidirectional::AR2): persistierte Zielreferenz (Y_MIN/Y_MID/Y_MAX)

Sockets:

- **X** (adapter::types::unidirectional::AR): Roheingang
- **Y_MIN**, **Y_MID**, **Y_MAX** (adapter::types::bidirectional::AR2): Zielausgabewert Y am Min-/Mid-/Max-Punkt. Live-Eingabe über EO1/DO1 (Socket-seitiger Swap); der persistierte/wiederhergestellte Wert wird über EI1/DI1 für VT-/Web-Anzeige zurückgespiegelt.

Funktionsweise

Der Baustein startet im Zustand START; INIT mit QI = TRUE führt über Init nach IDLE (QO := QI), INIT mit QI = FALSE über DeInit zurück nach START (QO := FALSE). Aus IDLE sind folgende Aktionszustände erreichbar, die sofort nach IDLE zurückkehren:

- **REQ** (bei X.E1 oder einem Ereignis X_MIN.EI1/X_MID.EI1/X_MAX.EI1): berechnet Y und feuert Y.E1
- **C_MIN**/**C_MID**/**C_MAX**: speichern den Roheingang X.D1 in X_MIN.DO1/X_MID.DO1/X_MAX.DO1 und feuern X_MIN.EO1/X_MID.EO1/X_MAX.EO1
- **MIN_REF_ECHO**/**MID_REF_ECHO**/**MAX_REF_ECHO** (bei Y_MIN.EO1/Y_MID.EO1/Y_MAX.EO1): übertragen die Live-Referenz Y_xxx.DO1 nach xxx_REF.DO1 (Persistenz)
- **Y_MIN_DISP**/**Y_MID_DISP**/**Y_MAX_DISP** (bei MIN_REF.EI1/MID_REF.EI1/MAX_REF.EI1): spiegeln die persistierte/wiederhergestellte Referenz nach Y_xxx.DI1 zurück (nach Boot und nach jeder Änderung)

Formel: stückweise lineare Interpolation zwischen (X_MIN, MIN_REF), (X_MID, MID_REF) und (X_MAX, MAX_REF), auf MIN_REF..MAX_REF geklemmt, sofern xClipping = TRUE.

Teilkalibrierung (bevor alle drei Punkte gesetzt sind): Jeder Punkt wird intern verfolgt (xMinCal/xMidCal/xMaxCal, gelatcht auf TRUE, sobald sein Rohwert vom INI-Default 0.0 abweicht; bleibt über Neustarts korrekt gesetzt, da Boot-Restore den persistierten Wert zurückliefert). Bis alle drei kalibriert sind, fällt REQ sauber zurück, statt auf einem noch unkalibrierten X_MID.DI1 (bei 0.0) zu pivotieren, was den MIN-Zweig dauerhaft unerreichbar machen würde:

- Nur ein Punkt kalibriert: Y = dessen eigener REF-Wert (keine Interpolation möglich)
- Zwei Punkte kalibriert: Zwei-Punkt-Gerade zwischen genau diesen beiden, der dritte wird ignoriert
- Alle drei kalibriert: volle stückweise Interpolation
- Keiner kalibriert: Y := MID_REF

Geklemmt wird nur, wenn xClipping gesetzt ist und sowohl MIN als auch MAX kalibriert sind; andernfalls würde die noch unkalibrierte Grenze mit dem INI-Default 0.0 den Teilkalibrierungs-Fallback verfälschen. MID ist dafür irrelevant.

Kalibrierung (Drei-Punkt, freie Reihenfolge):

1. **Min (C_MIN)**: Minimum anfahren, Y_MIN setzen, C_MIN auslösen ⟶ X_MIN := X
2. **Mid (C_MID)**: Mitte anfahren, Y_MID setzen, C_MID auslösen ⟶ X_MID := X
3. **Max (C_MAX)**: Maximum anfahren, Y_MAX setzen, C_MAX auslösen ⟶ X_MAX := X

Technische Besonderheiten

- **Freie Reihenfolge** wie bei AR_CALIBRATE_3P: Jeder Punkt wird unabhängig gespeichert und kann einzeln neu kalibriert werden (z. B. nur die Mitte nachtrimmen), ohne Datenabhängigkeit zwischen den Kalibrieraktionen.
- **Referenzen als Live-Sockets:** Y_MIN/Y_MID/Y_MAX sind bidirektionale AR2-Sockets, gespeist von einem externen VT-/Web-Leser, nach außen über die AR2-Plugs MIN_REF/MID_REF/MAX_REF für die Persistenz gespiegelt (INI_AR2/NVS_AR2) und über MIN_REF.EI1/MID_REF.EI1/MAX_REF.EI1 zurückgespielt, sodass

ein nach dem Boot wiederhergestellter Wert am Bildschirm erscheint, ohne dass der Bediener ihn erneut eingeben muss.
- Entartete Punkte (`X_MID <= X_MIN` bzw. `X_MAX <= X_MID`) liefern den jeweiligen REF-Wert statt einer Division durch Null.

Zustandsübersicht

```
START --INIT[QI]--> Init --1--> IDLE
IDLE --INIT[!QI]--> DeInit --1--> START
IDLE --X.E1 / X_xxx.EI1--> REQ --1--> IDLE
IDLE --C_MIN / C_MID / C_MAX--> C_xxx --1--> IDLE
IDLE --Y_xxx.EO1--> xxx_REF_ECHO --1--> IDLE
IDLE --xxx_REF.EI1--> Y_xxx_DISP --1--> IDLE
```

Anwendungsszenarien

- Lenkwinkelsensor: Min = Vollanschlag links, Mid = geradeaus, Max = Vollanschlag rechts. `MIN_REF`/`MID_REF`/`MAX_REF` enthalten die gewünschten kalibrierten Winkelwerte (z. B. −30,0 / 0,0 / 30,0 Grad), `X_MIN`/`X_MID`/`X_MAX` die tatsächlich gemessenen Rohwerte.
- Joysticks mit Mittelstellung und persistierten, am VT editierbaren Referenzwerten

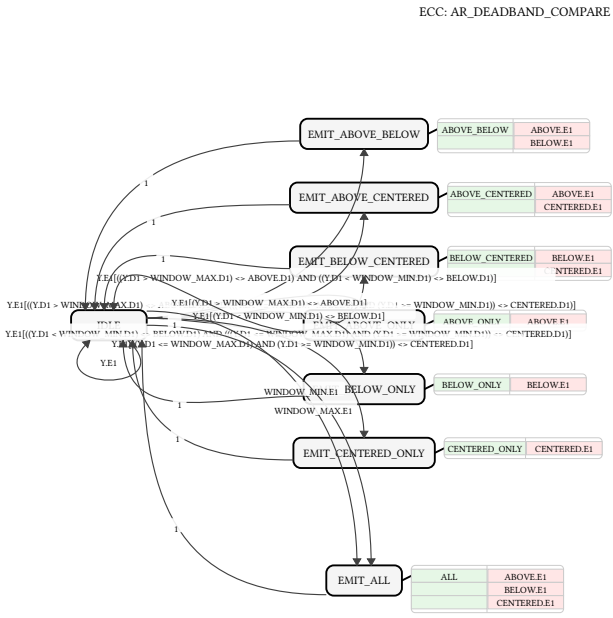
⚖️ Vergleich mit ähnlichen Bausteinen

- AR_CALIBRATE_3P: Drei-Punkt-Kalibrierung ohne `*_REF`-Sockets
- AR_CALIBRATE_2P_REF: Zwei-Punkt-Variante mit denselben Live-Referenzen

Fazit

`AR_CALIBRATE_3P_REF` kalibriert drei Punkte in freier Reihenfolge, hält Rohwerte und Referenzen persistiert über `AR2`-Adapter und fällt bei Teilkalibrierung sauber zurück.

AR_DEADBAND_COMPARE

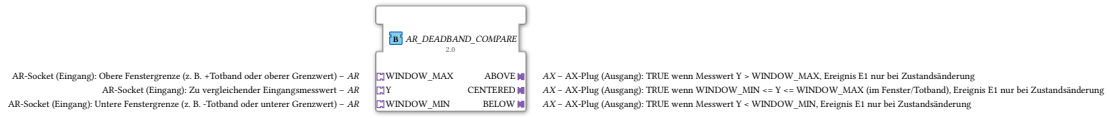


• – ○ • – ○ • – ○ •

Einleitung

`AR_DEADBAND_COMPARE` vergleicht einen `REAL`-Messwert `Y` (über einen `AR`-Socket) mit einem Fenster `[WINDOW_MIN, WINDOW_MAX]` und meldet das Ergebnis über drei `AX`-Plugs: `ABOVE` (`Y > WINDOW_MAX`), `BELOW` (`Y < WINDOW_MIN`) und `CENTERED` (dazwischen). Für ein symmetrisches Totband um Null setzt man

`WINDOW_MIN = -Totband` und `WINDOW_MAX = +Totband` (z. B. Lenkwinkel-Zentrierung); der Baustein ist genauso für beliebige Fenster nutzbar (z. B. Farbgrenzen).



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge – Ereignisse kommen über die Adapter-Sockets (`Y.E1`, `WINDOW_MIN.E1`, `WINDOW_MAX.E1`).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge – Ereignisse werden über die Adapter-Plugs gesendet (`ABOVE.E1`, `BELOW.E1`, `CENTERED.E1`).

Daten-Eingänge

Keine direkten Daten-Eingänge (Werte kommen über die Sockets, Datenport `D1`).

Daten-Ausgänge

Keine direkten Daten-Ausgänge (Werte werden über die Plugs ausgegeben, Datenport `D1`).

Adapter

Sockets (Eingänge):

Adapter	Typ	Beschreibung
`Y`	`adapter::types::unidirectional::AR`	Zu vergleichender Eingangsmesswert
`WINDOW_MIN`	`adapter::types::unidirectional::AR`	Untere Fenstergrenze (z. B. -Totband oder unterer Grenzwert)
`WINDOW_MAX`	`adapter::types::unidirectional::AR`	Obere Fenstergrenze (z. B. +Totband oder oberer Grenzwert)

Plugs (Ausgänge):

Adapter	Typ	Beschreibung
`ABOVE`	`adapter::types::unidirectional::AX`	TRUE, wenn `Y > WINDOW_MAX`; Ereignis `E1` nur bei Zustandsänderung
`CENTERED`	`adapter::types::unidirectional::AX`	TRUE, wenn `WINDOW_MIN <= Y <= WINDOW_MAX` (im Fenster/Totband); Ereignis `E1` nur bei Zustandsänderung
`BELOW`	`adapter::types::unidirectional::AX`	TRUE, wenn `Y < WINDOW_MIN`; Ereignis `E1` nur bei Zustandsänderung

Funktionsweise

Der Baustein ist ein Basis-FB mit einem Ruhezustand `IDLE` und sieben Sende-Zuständen, die jeweils nach ihrer Aktion sofort (Bedingung `1`) nach `IDLE` zurückkehren.

- **Y-Änderung (`Y.E1`)**: Die Transitionen vergleichen den neu berechneten Wert von `ABOVE`, `BELOW` und `CENTERED` mit dem jeweils gespeicherten Plug-Wert (`<>`). Nur die Ausgänge, deren boolescher Wert sich tatsächlich ändert, werden neu berechnet und senden ihr `E1`:
  - `EMIT_ABOVE_BELOW`: `ABOVE` und `BELOW` ändern sich
  - `EMIT_ABOVE_CENTERED`: `ABOVE` und `CENTERED` ändern sich
  - `EMIT_BELOW_CENTERED`: `BELOW` und `CENTERED` ändern sich
  - `EMIT_ABOVE_ONLY`, `EMIT_BELOW_ONLY`, `EMIT_CENTERED_ONLY`: nur der jeweilige Ausgang ändert sich
  - Ändert sich keiner der Werte, geht der Baustein mit `Y.E1` direkt `IDLE` ⟶ `IDLE` ohne Ausgangsereignis.

- **Fenstergrenzen-Änderung (`WINDOW_MIN.E1` oder `WINDOW_MAX.E1`)**: Zustand `EMIT_ALL` berechnet alle drei Ausgänge neu und sendet `ABOVE.E1`, `BELOW.E1` und `CENTERED.E1` (in der Praxis meist nur einmal beim Start).

Berechnungen (Algorithmen in ST):
```
ABOVE.D1    := Y.D1 > WINDOW_MAX.D1;
BELOW.D1    := Y.D1 < WINDOW_MIN.D1;
CENTERED.D1 := (Y.D1 <= WINDOW_MAX.D1) AND (Y.D1 >= WINDOW_MIN.D1);
```

Technische Besonderheiten

- **Ereignis-Effizienz**: Bei `Y`-Änderung feuert jeder Plug nur dann, wenn sich sein eigener Wert tatsächlich ändert (nicht alle drei bei jedem Update)
- **Fenstergrenzen**: Bei Änderung von `WINDOW_MIN`/`WINDOW_MAX` werden der Einfachheit halber immer alle drei Ausgänge neu gesetzt
- **Grenzwerte**: Die Grenzen selbst gehören zu `CENTERED` (<=/>=); `ABOVE` und `BELOW` sind strikt (>/<)
- Ein fester Wert für eine Grenze kann über `adapter::types::unidirectional::AR::initval::initval_AR` bereitgestellt werden
- Versionshistorie: 1.0 (initial, Achslenkung-Mittelpunkt-Automatik), 1.1 (DEADBAND als AR-Socket), 2.0 (verallgemeinert auf beliebiges Fenster `WINDOW_MIN`/`WINDOW_MAX`)

Zustandsübersicht

Zustand	Beschreibung	Aktion
IDLE	Wartet auf Ereignisse an `Y.E1`, `WINDOW_MIN.E1` oder `WINDOW_MAX.E1`	–
EMIT_ABOVE_BELOW	ABOVE und BELOW aktualisieren	`ABOVE.E1`, `BELOW.E1`
EMIT_ABOVE_CENTERED	ABOVE und CENTERED aktualisieren	`ABOVE.E1`, `CENTERED.E1`
EMIT_BELOW_CENTERED	BELOW und CENTERED aktualisieren	`BELOW.E1`, `CENTERED.E1`
EMIT_ABOVE_ONLY	Nur ABOVE aktualisieren	`ABOVE.E1`
EMIT_BELOW_ONLY	Nur BELOW aktualisieren	`BELOW.E1`
EMIT_CENTERED_ONLY	Nur CENTERED aktualisieren	`CENTERED.E1`
EMIT_ALL	Alle drei aktualisieren (Fenstergrenzen geändert)	`ABOVE.E1`, `BELOW.E1`, `CENTERED.E1`

Anwendungsszenarien

- **Lenkwinkel-Mittelpunktzentrierung**: Prüfen, ob ein Lenksensor innerhalb des Totbandes um 0° liegt (`WINDOW_MIN = -Totband`, `WINDOW_MAX = +Totband`)
- **Fenster-/Hysterese-Überwachung**: Auswertung von Grenzwert- oder Farbbereichen (z. B. grün/rot in Visualisierungen)

Fazit

`AR_DEADBAND_COMPARE` liefert für einen `AR`-Messwert die drei Zustände oberhalb, unterhalb und innerhalb eines Fensters als `AX`-Signale und erzeugt Ereignisse nur bei tatsächlicher Zustandsänderung.

Splitting

Vorzeichenbehaftete Aufteilungsbausteine (signed splitting)

Diese Bibliothek enthält Bausteine zum Aufteilen vorzeichenbehafteter Signale (`SINT`, `INT`, `DINT`, `LINT`, `REAL`, `LREAL`) in zwei positive Betragskomponenten (`NEG_MAG` und `POS_MAG`).

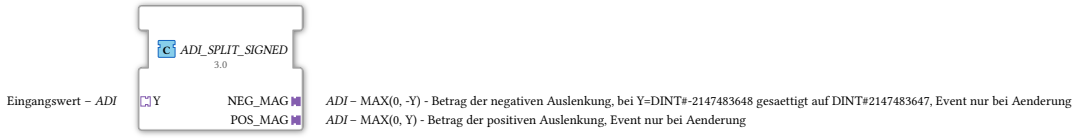
Basic FBs (Reine Berechnung)

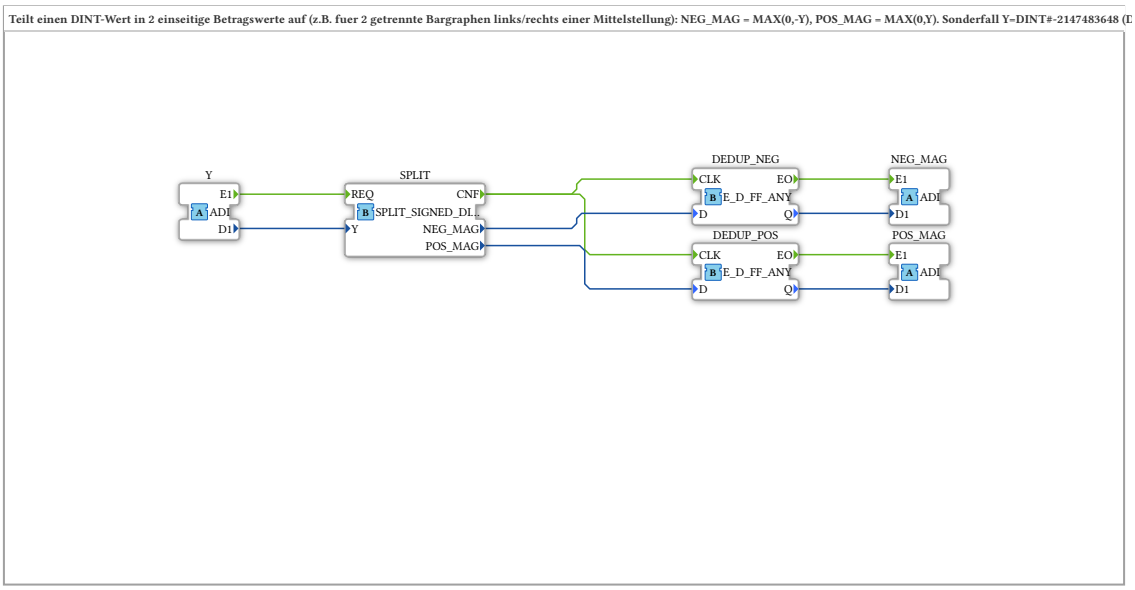
- SPLIT_SIGNED_SINT – Aufteilung für `SINT`
- SPLIT_SIGNED_INT – Aufteilung für `INT`
- SPLIT_SIGNED_DINT – Aufteilung für `DINT`
- SPLIT_SIGNED_LINT – Aufteilung für `LINT`
- SPLIT_SIGNED_REAL – Aufteilung für `REAL`
- SPLIT_SIGNED_LREAL – Aufteilung für `LREAL`

Adapter-Wrapper Composite FBs (Mit D-Flip-Flop Event-Filterung)

- AS_SPLIT_SIGNED – Adapter-Wrapper für `SINT` (`AS`)
- AI_SPLIT_SIGNED – Adapter-Wrapper für `INT` (`AI`)
- ADI_SPLIT_SIGNED – Adapter-Wrapper für `DINT` (`ADI`)
- ALI_SPLIT_SIGNED – Adapter-Wrapper für `LINT` (`ALI`)
- AR_SPLIT_SIGNED – Adapter-Wrapper für `REAL` (`AR`)
- ALR_SPLIT_SIGNED – Adapter-Wrapper für `LREAL` (`ALR`)

ADI_SPLIT_SIGNED





• – ○ • – ○ • – ○ •

Einleitung

Der **ADI_SPLIT_SIGNED** ist ein Adapter-Wrapper-Funktionsbaustein (Composite FB) zum Aufteilen eines vorzeichenbehafteten `DINT`-Messwerts in zwei einseitige Betragskomponenten über unidirektionale Adapter (`adapter::types::unidirectional::ADI`):

- `NEG_MAG`: Betrag der negativen Auslenkung (`MAX(0, -Y)`)
- `POS_MAG`: Betrag der positiven Auslenkung (`MAX(0, Y)`)

Der Baustein kapselt den Berechnungs-FB **SPLIT_SIGNED_DINT** sowie zwei D-Flip-Flop-Bausteine vom Typ **E_D_FF_ANY**, um das initiale Ereignis beim ersten `CLK` und danach Ereignisse auf den Plugs `NEG_MAG` und `POS_MAG` **nur bei tatsächlicher Wertänderung** der jeweiligen Seite auszulösen.

Schnittstellenstruktur

Adapter-Sockets (Eingang)

Name	Typ	Kommentar
Y	adapter::types::unidirectional::ADI	Vorzeichenbehafteter Eingangswert (ADI-Adapter-Socket)

Adapter-Plugs (Ausgang)

Name	Typ	Kommentar
NEG_MAG	adapter::types::unidirectional::ADI	Betrag der negativen Auslenkung (MAX(0, -Y)), initiales Ereignis beim 1. Aufruf, danach nur bei Wertänderung
POS_MAG	adapter::types::unidirectional::ADI	Betrag der positiven Auslenkung (MAX(0, Y)), initiales Ereignis beim 1. Aufruf, danach nur bei Wertänderung

Funktionsweise

Intern besteht das Composite-Netzwerk aus drei Bausteinen:

1. **SPLIT (SPLIT_SIGNED_DINT)**: Berechnet NEG_MAG und POS_MAG aus Y.D1 bei jedem Y.E1-Ereignis.
2. **DEDUP_NEG (E_D_FF_ANY)**: Ein ereignisgesteuertes D-Flip-Flop, das beim ersten CLK-Ereignis das initiale Ausgangsereignis auslöst und danach CLK nur dann an EO weiterleitet und Q := D aktualisiert, wenn der neue Wert D ungleich dem bisherigen Ausgangswert Q ist (D <> Q).
3. **DEDUP_POS (E_D_FF_ANY)**: Ein zweites D-Flip-Flop derselben Bauart, das das initiale Ereignis beim 1. Aufruf ausgibt und spätere Ereignisse auf POS_MAG.E1 ebenfalls nur bei einer echten Datenänderung (POS_MAG.D1 <> POS_MAG.Q) sendet.

```
Y (ADI-Adapter-Socket)
 ├──> SPLIT (SPLIT_SIGNED_DINT)
       ├──> NEG_MAG ──> DEDUP_NEG (E_D_FF_ANY D-Flip-Flop) ──> NEG_MAG (ADI-Adapter-
Plug)
       └──> POS_MAG ──> DEDUP_POS (E_D_FF_ANY D-Flip-Flop) ──> POS_MAG (ADI-Adapter-
Plug)
```

Technische Besonderheiten

- **Adapterbasierte Architektur:** Vollständig kompatibel mit der unidirektionalen Adapterfamilie adapter::types::unidirectional::ADI.
- **Ereignis-Filterung per D-Flip-Flop:** Durch die E_D_FF_ANY Bausteine wird sichergestellt, dass an NEG_MAG.E1 bzw. POS_MAG.E1 das initiale Ausgangsereignis beim ersten CLK gesendet wird und spätere Ereignisse per Definition nur dann erzeugt werden, wenn sich der neue Wert D vom bisherigen Zustand Q unterscheidet.
- **Überlaufsicherheit:** Vererbt die Sättigungslogik von SPLIT_SIGNED_DINT für Zweierkomplement-Ganzzahlgrenzen.

Anwendungsszenarien

- Anbindung von bipolaren Gebern (z. B. Lenkwinkelsensor, Neigungssensor, Joystick) an zwei getrennte VT-Bargraphen.
- Adapterbasierte Ansteuerung von Zylinderantrieben (Ausfahren/Einfahren).

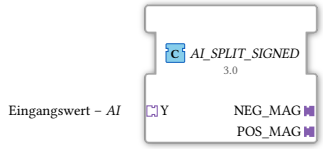
Vergleich mit ähnlichen Bausteinen

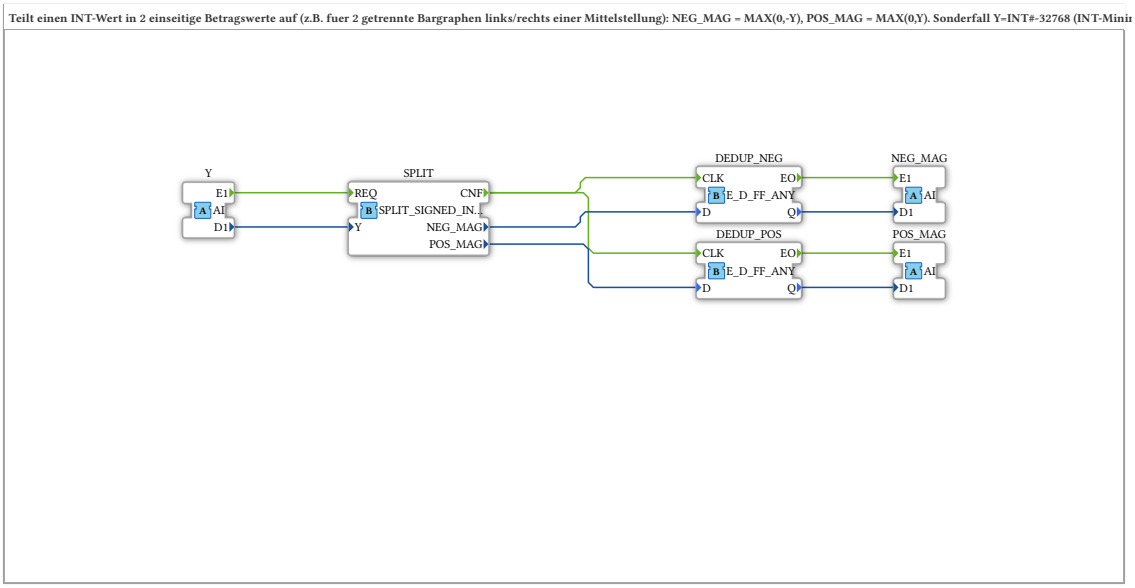
- **ADI_SPLIT_SIGNED**: Composite FB mit Adapter-Schnittstellen und D-Flip-Flop Event-Filterung.
- **SPLIT_SIGNED_DINT**: Unterlagerter Basic FB für reine Signalberechnung ohne Adapter.

Fazit

ADI_SPLIT_SIGNED bietet eine elegante, adapterbasierte und durch D-Flip-Flops ereigniseffiziente Lösung zur Vorzeichen-Signalaufteilung in IEC 61499 Anwendungen.

AI_SPLIT_SIGNED





• – ○ • – ○ • – ○ •

Einleitung

Der **AI_SPLIT_SIGNED** ist ein Adapter-Wrapper-Funktionsbaustein (Composite FB) zum Aufteilen eines vorzeichenbehafteten `INT`-Messwerts in zwei einseitige Betragskomponenten über unidirektionale Adapter (`adapter::types::unidirectional::AI`):

- `NEG_MAG`: Betrag der negativen Auslenkung (`MAX(0, -Y)`)
- `POS_MAG`: Betrag der positiven Auslenkung (`MAX(0, Y)`)

Der Baustein kapselt den Berechnungs-FB **SPLIT_SIGNED_INT** sowie zwei D-Flip-Flop-Bausteine vom Typ **E_D_FF_ANY**, um das initiale Ereignis beim ersten `CLK` und danach Ereignisse auf den Plugs `NEG_MAG` und `POS_MAG` **nur bei tatsächlicher Wertänderung** der jeweiligen Seite auszulösen.

Schnittstellenstruktur

Adapter-Sockets (Eingang)

Name	Typ	Kommentar
`Y`	`adapter::types::unidirectional::AI`	Vorzeichenbehafteter Eingangswert (AI-Adapter-Socket)

Adapter-Plugs (Ausgang)

Name	Typ	Kommentar
`NEG_MAG`	`adapter::types::unidirectional::AI`	Betrag der negativen Auslenkung (`MAX(0, -Y)`), initiales Ereignis beim 1. Aufruf, danach nur bei Wertänderung
`POS_MAG`	`adapter::types::unidirectional::AI`	Betrag der positiven Auslenkung (`MAX(0, Y)`), initiales Ereignis beim 1. Aufruf, danach nur bei Wertänderung

Funktionsweise

Intern besteht das Composite-Netzwerk aus drei Bausteinen:

1. **SPLIT (SPLIT_SIGNED_INT)**: Berechnet `NEG_MAG` und `POS_MAG` aus `Y.D1` bei jedem `Y.E1`-Ereignis.
2. **DEDUP_NEG (E_D_FF_ANY)**: Ein ereignisgesteuertes D-Flip-Flop, das beim ersten `CLK`-Ereignis das initiale Ausgangsereignis auslöst und danach `CLK` nur dann an `EO` weiterleitet und `Q := D` aktualisiert, wenn der neue Wert `D` ungleich dem bisherigen Ausgangswert `Q` ist (`D <> Q`).
3. **DEDUP_POS (E_D_FF_ANY)**: Ein zweites D-Flip-Flop derselben Bauart, das das initiale Ereignis beim 1. Aufruf ausgibt und spätere Ereignisse auf `POS_MAG.E1` ebenfalls nur bei einer echten Datenänderung (`POS_MAG.D1 <> POS_MAG.Q`) sendet.

```
Y (AI-Adapter-Socket)
 ├──> SPLIT (SPLIT_SIGNED_INT)
       ├──> NEG_MAG ──> DEDUP_NEG (E_D_FF_ANY D-Flip-Flop) ──> NEG_MAG (AI-Adapter-Plug)
       └──> POS_MAG ──> DEDUP_POS (E_D_FF_ANY D-Flip-Flop) ──> POS_MAG (AI-Adapter-Plug)
```

Technische Besonderheiten

- **Adapterbasierte Architektur:** Vollständig kompatibel mit der unidirektionalen Adapterfamilie `adapter::types::unidirectional::AI`.
- **Ereignis-Filterung per D-Flip-Flop:** Durch die `E_D_FF_ANY` Bausteine wird sichergestellt, dass an `NEG_MAG.E1` bzw. `POS_MAG.E1` das initiale Ausgangsereignis beim ersten `CLK` gesendet wird und spätere Ereignisse per Definition nur dann erzeugt werden, wenn sich der neue Wert `D` vom bisherigen Zustand `Q` unterscheidet.
- **Überlaufsicherheit:** Vererbt die Sättigungslogik von `SPLIT_SIGNED_INT` für Zweierkomplement-Ganzzahlgrenzen.

Anwendungsszenarien

- Anbindung von bipolaren Gebern (z. B. Lenkwinkelsensor, Neigungssensor, Joystick) an zwei getrennte VT-Bargraphen.
- Adapterbasierte Ansteuerung von Zylinderantrieben (Ausfahren/Einfahren).

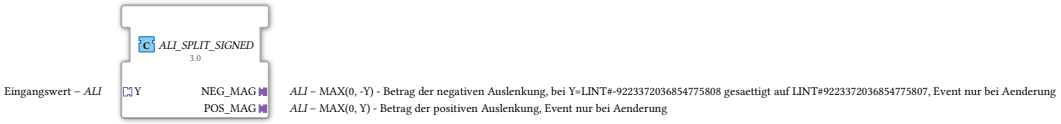
Vergleich mit ähnlichen Bausteinen

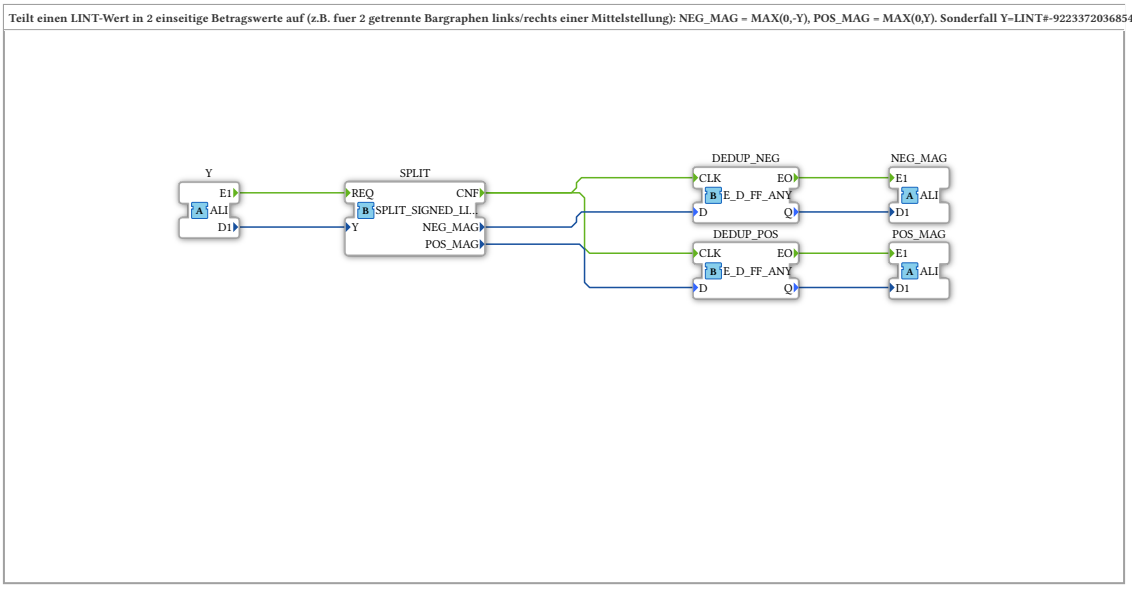
- **AI_SPLIT_SIGNED**: Composite FB mit Adapter-Schnittstellen und D-Flip-Flop Event-Filterung.
- **SPLIT_SIGNED_INT**: Unterlagerter Basic FB für reine Signalberechnung ohne Adapter.

Fazit

AI_SPLIT_SIGNED bietet eine elegante, adapterbasierte und durch D-Flip-Flops ereigniseffiziente Lösung zur Vorzeichen-Signalaufteilung in IEC 61499 Anwendungen.

ALI_SPLIT_SIGNED





• – ○ • – ○ • – ○ •

Einleitung

Der **ALI_SPLIT_SIGNED** ist ein Adapter-Wrapper-Funktionsbaustein (Composite FB) zum Aufteilen eines vorzeichenbehafteten `LINT`-Messwerts in zwei einseitige Betragskomponenten über unidirektionale Adapter (`adapter::types::unidirectional::ALI`):

- `NEG_MAG`: Betrag der negativen Auslenkung (`MAX(0, -Y)`)
- `POS_MAG`: Betrag der positiven Auslenkung (`MAX(0, Y)`)

Der Baustein kapselt den Berechnungs-FB **SPLIT_SIGNED_LINT** sowie zwei D-Flip-Flop-Bausteine vom Typ **E_D_FF_ANY**, um das initiale Ereignis beim ersten `CLK` und danach Ereignisse auf den Plugs `NEG_MAG` und `POS_MAG` **nur bei tatsächlicher Wertänderung** der jeweiligen Seite auszulösen.

Schnittstellenstruktur

Adapter-Sockets (Eingang)

Name	Typ	Kommentar
`Y`	`adapter::types::unidirectional::ALI`	Vorzeichenbehafteter Eingangswert (ALI-Adapter-Socket)

Adapter-Plugs (Ausgang)

Name	Typ	Kommentar
`NEG_MAG`	`adapter::types::unidirectional::ALI`	Betrag der negativen Auslenkung (`MAX(0, -Y)`), initiales Ereignis beim 1. Aufruf, danach nur bei Wertänderung
`POS_MAG`	`adapter::types::unidirectional::ALI`	Betrag der positiven Auslenkung (`MAX(0, Y)`), initiales Ereignis beim 1. Aufruf, danach nur bei Wertänderung

Funktionsweise

Intern besteht das Composite-Netzwerk aus drei Bausteinen:

1. **SPLIT (SPLIT_SIGNED_LINT)**: Berechnet `NEG_MAG` und `POS_MAG` aus `Y.D1` bei jedem `Y.E1`-Ereignis.
2. **DEDUP_NEG (E_D_FF_ANY)**: Ein ereignisgesteuertes D-Flip-Flop, das beim ersten `CLK`-Ereignis das initiale Ausgangsereignis auslöst und danach `CLK` nur dann an `EO` weiterleitet und `Q := D` aktualisiert, wenn der neue Wert `D` ungleich dem bisherigen Ausgangswert `Q` ist (`D <> Q`).

3. **DEDUP_POS (E_D_FF_ANY)**: Ein zweites D-Flip-Flop derselben Bauart, das das initiale Ereignis beim 1. Aufruf ausgibt und spätere Ereignisse auf `POS_MAG.E1` ebenfalls nur bei einer echten Datenänderung (`POS_MAG.D1 <> POS_MAG.Q`) sendet.

```
Y (ALI-Adapter-Socket)
 ├──> SPLIT (SPLIT_SIGNED_LINT)
       ├──> NEG_MAG ──> DEDUP_NEG (E_D_FF_ANY D-Flip-Flop) ──> NEG_MAG (ALI-Adapter-
Plug)
       └──> POS_MAG ──> DEDUP_POS (E_D_FF_ANY D-Flip-Flop) ──> POS_MAG (ALI-Adapter-
Plug)
```

Technische Besonderheiten

- **Adapterbasierte Architektur:** Vollständig kompatibel mit der unidirektionalen Adapterfamilie `adapter::types::unidirectional::ALI`.
- **Ereignis-Filterung per D-Flip-Flop:** Durch die `E_D_FF_ANY` Bausteine wird sichergestellt, dass an `NEG_MAG.E1` bzw. `POS_MAG.E1` das initiale Ausgangsereignis beim ersten `CLK` gesendet wird und spätere Ereignisse per Definition nur dann erzeugt werden, wenn sich der neue Wert `D` vom bisherigen Zustand `Q` unterscheidet.
- **Überlaufsicherheit:** Vererbt die Sättigungslogik von `SPLIT_SIGNED_LINT` für Zweierkomplement-Ganzzahlgrenzen.

Anwendungsszenarien

- Anbindung von bipolaren Gebern (z. B. Lenkwinkelsensor, Neigungssensor, Joystick) an zwei getrennte VT-Bargraphen.
- Adapterbasierte Ansteuerung von Zylinderantrieben (Ausfahren/Einfahren).

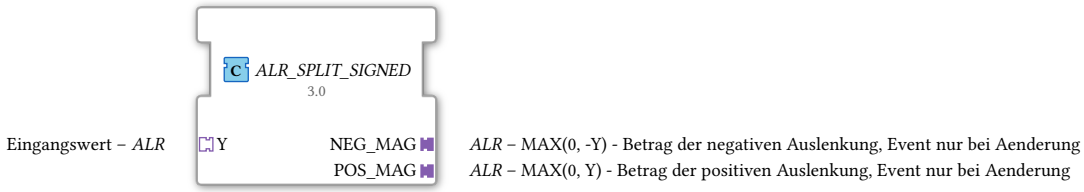
Vergleich mit ähnlichen Bausteinen

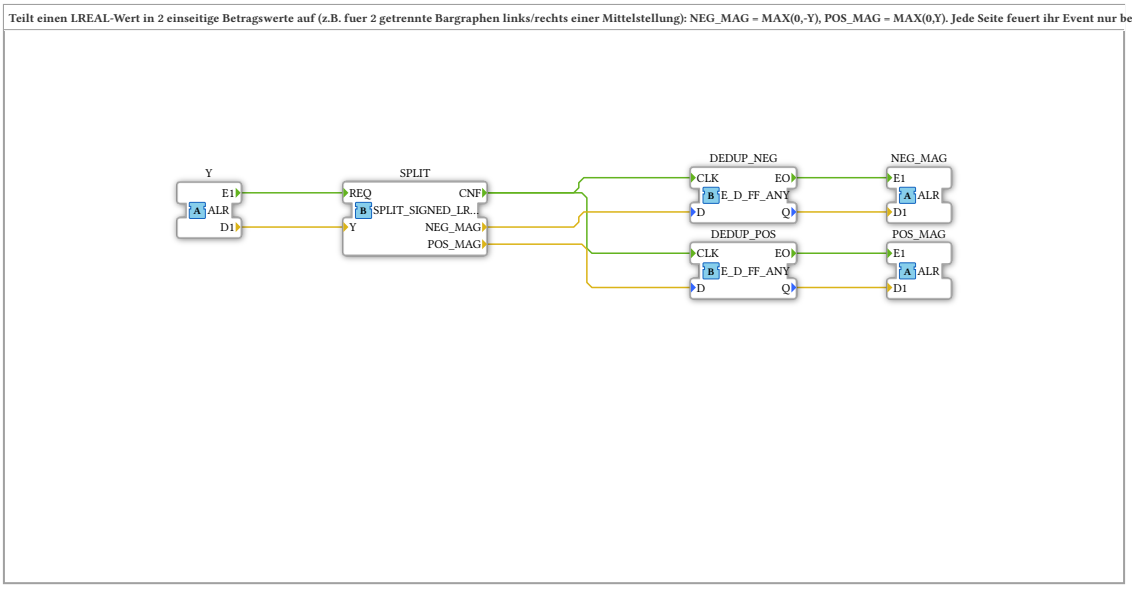
- **ALI_SPLIT_SIGNED**: Composite FB mit Adapter-Schnittstellen und D-Flip-Flop Event-Filterung.
- **SPLIT_SIGNED_LINT**: Unterlagerter Basic FB für reine Signalberechnung ohne Adapter.

Fazit

ALI_SPLIT_SIGNED bietet eine elegante, adapterbasierte und durch D-Flip-Flops ereigniseffiziente Lösung zur Vorzeichen-Signalaufteilung in IEC 61499 Anwendungen.

ALR_SPLIT_SIGNED





• – ○ • – ○ • – ○ •

Einleitung

Der **ALR_SPLIT_SIGNED** ist ein Adapter-Wrapper-Funktionsbaustein (Composite FB) zum Aufteilen eines vorzeichenbehafteten `LREAL`-Messwerts in zwei einseitige Betragskomponenten über unidirektionale Adapter (`adapter::types::unidirectional::ALR`):

- `NEG_MAG`: Betrag der negativen Auslenkung (`MAX(0, -Y)`)
- `POS_MAG`: Betrag der positiven Auslenkung (`MAX(0, Y)`)

Der Baustein kapselt den Berechnungs-FB **SPLIT_SIGNED_LREAL** sowie zwei D-Flip-Flop-Bausteine vom Typ **E_D_FF_ANY**, um das initiale Ereignis beim ersten `CLK` und danach Ereignisse auf den Plugs `NEG_MAG` und `POS_MAG` **nur bei tatsächlicher Wertänderung** der jeweiligen Seite auszulösen.

Schnittstellenstruktur

Adapter-Sockets (Eingang)

Name	Typ	Kommentar
`Y`	`adapter::types::unidirectional::ALR`	Vorzeichenbehafteter Eingangswert (ALR-Adapter-Socket)

Adapter-Plugs (Ausgang)

Name	Typ	Kommentar
`NEG_MAG`	`adapter::types::unidirectional::ALR`	Betrag der negativen Auslenkung (`MAX(0, -Y)`), initiales Ereignis beim 1. Aufruf, danach nur bei Wertänderung
`POS_MAG`	`adapter::types::unidirectional::ALR`	Betrag der positiven Auslenkung (`MAX(0, Y)`), initiales Ereignis beim 1. Aufruf, danach nur bei Wertänderung

Funktionsweise

Intern besteht das Composite-Netzwerk aus drei Bausteinen:

1. **SPLIT (SPLIT_SIGNED_LREAL)**: Berechnet `NEG_MAG` und `POS_MAG` aus `Y.D1` bei jedem `Y.E1`-Ereignis.
2. **DEDUP_NEG (E_D_FF_ANY)**: Ein ereignisgesteuertes D-Flip-Flop, das beim ersten `CLK`-Ereignis das initiale Ausgangsereignis auslöst und danach `CLK` nur dann an `EO` weiterleitet und `Q := D` aktualisiert, wenn der neue Wert `D` ungleich dem bisherigen Ausgangswert `Q` ist (`D <> Q`).

3. **DEDUP_POS (E_D_FF_ANY)**: Ein zweites D-Flip-Flop derselben Bauart, das das initiale Ereignis beim 1. Aufruf ausgibt und spätere Ereignisse auf `POS_MAG.E1` ebenfalls nur bei einer echten Datenänderung (`POS_MAG.D1 <> POS_MAG.Q`) sendet.

```
Y (ALR-Adapter-Socket)
 ├──> SPLIT (SPLIT_SIGNED_LREAL)
       ├──> NEG_MAG ──> DEDUP_NEG (E_D_FF_ANY D-Flip-Flop) ──> NEG_MAG (ALR-Adapter-
Plug)
       └──> POS_MAG ──> DEDUP_POS (E_D_FF_ANY D-Flip-Flop) ──> POS_MAG (ALR-Adapter-
Plug)
```

Technische Besonderheiten

- **Adapterbasierte Architektur:** Vollständig kompatibel mit der unidirektionalen Adapterfamilie `adapter::types::unidirectional::ALR`.
- **Ereignis-Filterung per D-Flip-Flop:** Durch die `E_D_FF_ANY` Bausteine wird sichergestellt, dass an `NEG_MAG.E1` bzw. `POS_MAG.E1` das initiale Ausgangsereignis beim ersten `CLK` gesendet wird und spätere Ereignisse per Definition nur dann erzeugt werden, wenn sich der neue Wert `D` vom bisherigen Zustand `Q` unterscheidet.
- **Überlaufsicherheit:** Vererbt die Sättigungslogik von `SPLIT_SIGNED_LREAL` für Zweierkomplement-Ganzzahlgrenzen.

Anwendungsszenarien

- Anbindung von bipolaren Gebern (z. B. Lenkwinkelsensor, Neigungssensor, Joystick) an zwei getrennte VT-Bargraphen.
- Adapterbasierte Ansteuerung von Zylinderantrieben (Ausfahren/Einfahren).

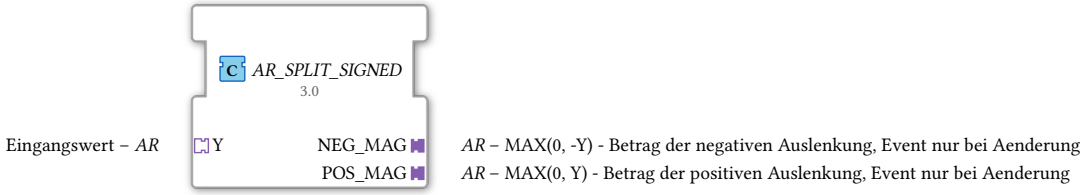
Vergleich mit ähnlichen Bausteinen

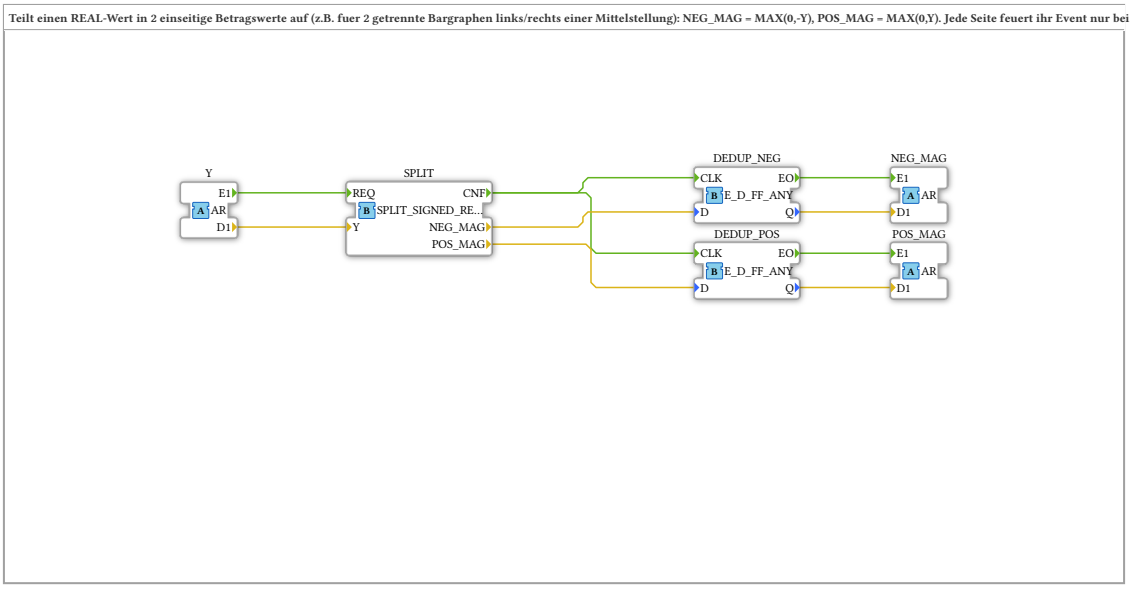
- **ALR_SPLIT_SIGNED**: Composite FB mit Adapter-Schnittstellen und D-Flip-Flop Event-Filterung.
- **SPLIT_SIGNED_LREAL**: Unterlagerter Basic FB für reine Signalberechnung ohne Adapter.

Fazit

ALR_SPLIT_SIGNED bietet eine elegante, adapterbasierte und durch D-Flip-Flops ereigniseffiziente Lösung zur Vorzeichen-Signalaufteilung in IEC 61499 Anwendungen.

AR_SPLIT_SIGNED





• – ○ • – ○ • – ○ •

Einleitung

Der **AR_SPLIT_SIGNED** ist ein Adapter-Wrapper-Funktionsbaustein (Composite FB) zum Aufteilen eines vorzeichenbehafteten `REAL`-Messwerts in zwei einseitige Betragskomponenten über unidirektionale Adapter (`adapter::types::unidirectional::AR`):

- `NEG_MAG`: Betrag der negativen Auslenkung (`MAX(0, -Y)`)
- `POS_MAG`: Betrag der positiven Auslenkung (`MAX(0, Y)`)

Der Baustein kapselt den Berechnungs-FB **SPLIT_SIGNED_REAL** sowie zwei D-Flip-Flop-Bausteine vom Typ **E_D_FF_ANY**, um das initiale Ereignis beim ersten `CLK` und danach Ereignisse auf den Plugs `NEG_MAG` und `POS_MAG` **nur bei tatsächlicher Wertänderung** der jeweiligen Seite auszulösen.

Schnittstellenstruktur

Adapter-Sockets (Eingang)

Name	Typ	Kommentar
`Y`	`adapter::types::unidirectional::AR`	Vorzeichenbehafteter Eingangswert (AR-Adapter-Socket)

Adapter-Plugs (Ausgang)

Name	Typ	Kommentar
`NEG_MAG`	`adapter::types::unidirectional::AR`	Betrag der negativen Auslenkung (`MAX(0, -Y)`), initiales Ereignis beim 1. Aufruf, danach nur bei Wertänderung
`POS_MAG`	`adapter::types::unidirectional::AR`	Betrag der positiven Auslenkung (`MAX(0, Y)`), initiales Ereignis beim 1. Aufruf, danach nur bei Wertänderung

Funktionsweise

Intern besteht das Composite-Netzwerk aus drei Bausteinen:

1. **SPLIT (SPLIT_SIGNED_REAL)**: Berechnet `NEG_MAG` und `POS_MAG` aus `Y.D1` bei jedem `Y.E1`-Ereignis.
2. **DEDUP_NEG (E_D_FF_ANY)**: Ein ereignisgesteuertes D-Flip-Flop, das beim ersten `CLK`-Ereignis das initiale Ausgangsereignis auslöst und danach `CLK` nur dann an `EO` weiterleitet und `Q := D` aktualisiert, wenn der neue Wert `D` ungleich dem bisherigen Ausgangswert `Q` ist (`D <> Q`).

3. **DEDUP_POS (E_D_FF_ANY)**: Ein zweites D-Flip-Flop derselben Bauart, das das initiale Ereignis beim 1. Aufruf ausgibt und spätere Ereignisse auf `POS_MAG.E1` ebenfalls nur bei einer echten Datenänderung (`POS_MAG.D1 <> POS_MAG.Q`) sendet.

```
Y (AR-Adapter-Socket)
 ├──> SPLIT (SPLIT_SIGNED_REAL)
       ├──> NEG_MAG ──> DEDUP_NEG (E_D_FF_ANY D-Flip-Flop) ──> NEG_MAG (AR-Adapter-Plug)
       └──> POS_MAG ──> DEDUP_POS (E_D_FF_ANY D-Flip-Flop) ──> POS_MAG (AR-Adapter-Plug)
```

Technische Besonderheiten

- **Adapterbasierte Architektur:** Vollständig kompatibel mit der unidirektionalen Adapterfamilie `adapter::types::unidirectional::AR`.
- **Ereignis-Filterung per D-Flip-Flop:** Durch die `E_D_FF_ANY` Bausteine wird sichergestellt, dass an `NEG_MAG.E1` bzw. `POS_MAG.E1` das initiale Ausgangsereignis beim ersten `CLK` gesendet wird und spätere Ereignisse per Definition nur dann erzeugt werden, wenn sich der neue Wert `D` vom bisherigen Zustand `Q` unterscheidet.
- **Überlaufsicherheit:** Vererbt die Sättigungslogik von `SPLIT_SIGNED_REAL` für Zweierkomplement-Ganzzahlgrenzen.

Anwendungsszenarien

- Anbindung von bipolaren Gebern (z. B. Lenkwinkelsensor, Neigungssensor, Joystick) an zwei getrennte VT-Bargraphen.
- Adapterbasierte Ansteuerung von Zylinderantrieben (Ausfahren/Einfahren).

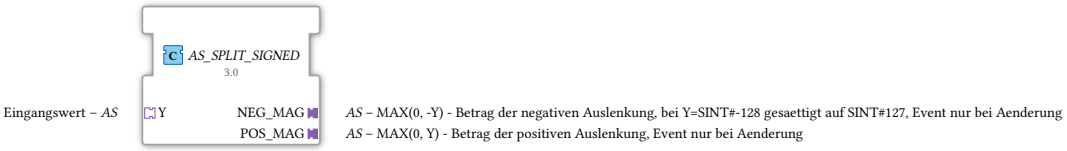
Vergleich mit ähnlichen Bausteinen

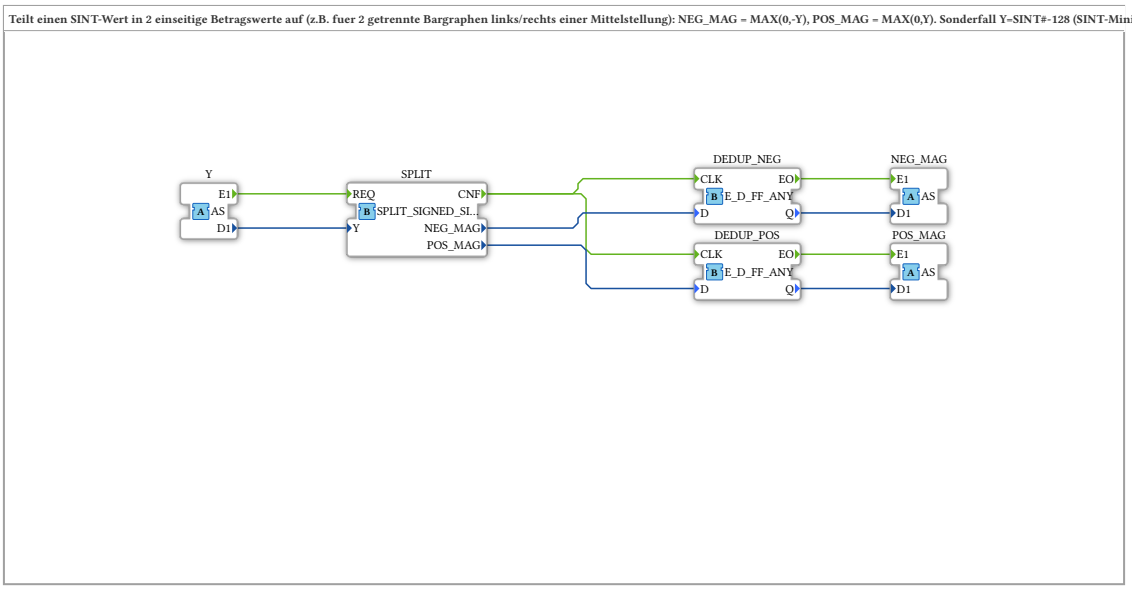
- **AR_SPLIT_SIGNED**: Composite FB mit Adapter-Schnittstellen und D-Flip-Flop Event-Filterung.
- **SPLIT_SIGNED_REAL**: Unterlagerter Basic FB für reine Signalberechnung ohne Adapter.

Fazit

AR_SPLIT_SIGNED bietet eine elegante, adapterbasierte und durch D-Flip-Flops ereigniseffiziente Lösung zur Vorzeichen-Signalaufteilung in IEC 61499 Anwendungen.

AS_SPLIT_SIGNED





• – ○ • – ○ • – ○ •

Einleitung

Der **AS_SPLIT_SIGNED** ist ein Adapter-Wrapper-Funktionsbaustein (Composite FB) zum Aufteilen eines vorzeichenbehafteten SINT-Messwerts in zwei einseitige Betragskomponenten über unidirektionale Adapter (adapter::types::unidirectional::AS):

- NEG_MAG: Betrag der negativen Auslenkung (MAX(0, -Y))
- POS_MAG: Betrag der positiven Auslenkung (MAX(0, Y))

Der Baustein kapselt den Berechnungs-FB **SPLIT_SIGNED_SINT** sowie zwei D-Flip-Flop-Bausteine vom Typ **E_D_FF_ANY**, um das initiale Ereignis beim ersten CLK und danach Ereignisse auf den Plugs NEG_MAG und POS_MAG **nur bei tatsächlicher Wertänderung** der jeweiligen Seite auszulösen.

Schnittstellenstruktur

Adapter-Sockets (Eingang)

Name	Typ	Kommentar
Y	adapter::types::unidirectional::AS	Vorzeichenbehafteter Eingangswert (AS-Adapter-Socket)

Adapter-Plugs (Ausgang)

Name	Typ	Kommentar
NEG_MAG	adapter::types::unidirectional::AS	Betrag der negativen Auslenkung (MAX(0, -Y)), initiales Ereignis beim 1. Aufruf, danach nur bei Wertänderung
POS_MAG	adapter::types::unidirectional::AS	Betrag der positiven Auslenkung (MAX(0, Y)), initiales Ereignis beim 1. Aufruf, danach nur bei Wertänderung

Funktionsweise

Intern besteht das Composite-Netzwerk aus drei Bausteinen:

1. **SPLIT (SPLIT_SIGNED_SINT)**: Berechnet NEG_MAG und POS_MAG aus Y.D1 bei jedem Y.E1-Ereignis.
2. **DEDUP_NEG (E_D_FF_ANY)**: Ein ereignisgesteuertes D-Flip-Flop, das beim ersten CLK-Ereignis das initiale Ausgangsereignis auslöst und danach CLK nur dann an E0 weiterleitet und Q := D aktualisiert, wenn der neue Wert D ungleich dem bisherigen Ausgangswert Q ist (D <> Q).
3. **DEDUP_POS (E_D_FF_ANY)**: Ein zweites D-Flip-Flop derselben Bauart, das das initiale Ereignis beim 1. Aufruf ausgibt und spätere Ereignisse auf POS_MAG.E1 ebenfalls nur bei einer echten Datenänderung (POS_MAG.D1 <> POS_MAG.Q) sendet.

```
Y (AS-Adapter-Socket)
 ├──> SPLIT (SPLIT_SIGNED_SINT)
       ├──> NEG_MAG ──> DEDUP_NEG (E_D_FF_ANY D-Flip-Flop) ──> NEG_MAG (AS-Adapter-Plug)
       └──> POS_MAG ──> DEDUP_POS (E_D_FF_ANY D-Flip-Flop) ──> POS_MAG (AS-Adapter-Plug)
```

Technische Besonderheiten

- **Adapterbasierte Architektur:** Vollständig kompatibel mit der unidirektionalen Adapterfamilie adapter::types::unidirectional::AS.
- **Ereignis-Filterung per D-Flip-Flop:** Durch die E_D_FF_ANY Bausteine wird sichergestellt, dass an NEG_MAG.E1 bzw. POS_MAG.E1 das initiale Ausgangsereignis beim ersten CLK gesendet wird und spätere Ereignisse per Definition nur dann erzeugt werden, wenn sich der neue Wert D vom bisherigen Zustand Q unterscheidet.
- **Überlaufsicherheit:** Vererbt die Sättigungslogik von SPLIT_SIGNED_SINT für Zweierkomplement-Ganzzahlgrenzen.

Anwendungsszenarien

- Anbindung von bipolaren Gebern (z. B. Lenkwinkelsensor, Neigungssensor, Joystick) an zwei getrennte VT-Bargraphen.
- Adapterbasierte Ansteuerung von Zylinderantrieben (Ausfahren/Einfahren).

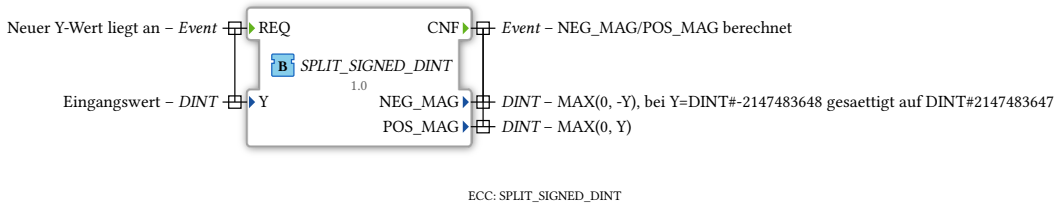
Vergleich mit ähnlichen Bausteinen

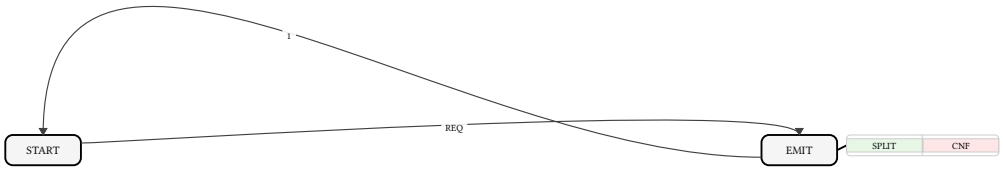
- **AS_SPLIT_SIGNED**: Composite FB mit Adapter-Schnittstellen und D-Flip-Flop Event-Filterung.
- **SPLIT_SIGNED_SINT**: Unterlagerter Basic FB für reine Signalberechnung ohne Adapter.

Fazit

AS_SPLIT_SIGNED bietet eine elegante, adapterbasierte und durch D-Flip-Flops ereigniseffiziente Lösung zur Vorzeichen-Signalaufteilung in IEC 61499 Anwendungen.

SPLIT_SIGNED_DINT





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **SPLIT_SIGNED_DINT** teilt einen vorzeichenbehafteten `DINT`-Eingangswert `Y` in zwei einseitige Betragswerte auf:

- `NEG_MAG = MAX(0, -Y)` (Betrag der negativen Auslenkung)
- `POS_MAG = MAX(0, Y)` (Betrag der positiven Auslenkung)

Er dient als primärer Berechnungsbaustein für gerichtete Anzeigen (z. B. getrennte Balkenanzeigen / Bargraphen für Links- und Rechtsauslenkung aus einer Mittelstellung) oder getrennte Ansteuerungen (z. B. für Heben/Senken oder Vorwärts/Rückwärts).

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
`REQ`	`Event`	Ausführung der Aufteilung anfordern (mit `Y`)

Ereignis-Ausgänge

Name	Typ	Kommentar
CNF	Event	Bestätigung der Berechnung (mit NEG_MAG, POS_MAG)

Daten-Eingänge

Name	Typ	Kommentar
Y	DINT	Vorzeichenbehafteter Eingangsmesswert

Daten-Ausgänge

Name	Typ	Kommentar
NEG_MAG	DINT	MAX(0, -Y) – Betrag der negativen Auslenkung
POS_MAG	DINT	MAX(0, Y) – Betrag der positiven Auslenkung

Funktionsweise

Beim Eintreffen des Ereignisses REQ führt der Baustein folgende Aufteilungslogik aus:

```
IF Y = DINT#-2147483648 THEN
    NEG_MAG := DINT#2147483647;
ELSE
    NEG_MAG := MAX(DINT#0, -Y);
END_IF;
POS_MAG := MAX(DINT#0, Y);
```

Der Baustein feuert bei jedem REQ-Ereignis die Bestätigung CNF. Eine Ereignis-Filterung bei gleichbleibenden Werten findet auf dieser Basisebene nicht statt (diese wird im Adapter-Wrapper **ADI_SPLIT_SIGNED** über **E_D_FF_ANY** D-Flip-Flops realisiert).

Überlaufbehandlung & Sättigung

Bei Zweierkomplement-Ganzzahltypen ist der Betrag des negativen Minimums (-2147483648) um 1 größer als das darstellbare positive Maximum (2147483647). Eine mathematische Negation von DINT#-2147483648 würde ohne Sonderbehandlung zu einem Ganzzahlüberlauf und damit zu einem fehlerhaften Vorzeichenwechsel führen.

Der **SPLIT_SIGNED_DINT** fängt diesen Sonderfall explizit ab:

- Wenn Y = DINT#-2147483648, wird NEG_MAG auf den maximal möglichen positiven Wert DINT#2147483647 gesättigt.
- Regelungstechnischer Vorteil: Sättigung am Messbereichsende statt Vorzeichenumkehr oder Überlauf.

Anwendungsszenarien

- Aufteilung von Vorzeichen-Messwerten für getrennte Visualisierungselemente (z. B. Links/Rechts-Bargraph).
- Ansteuerung von zwei unidirektionalen Aktoren (z. B. Ventil Heben vs. Ventil Senken) aus einem bipolaren Sollwert.
- Vorverarbeitung in Regelkreisen mit richtungsabhängigen Totbändern oder Kennlinien.

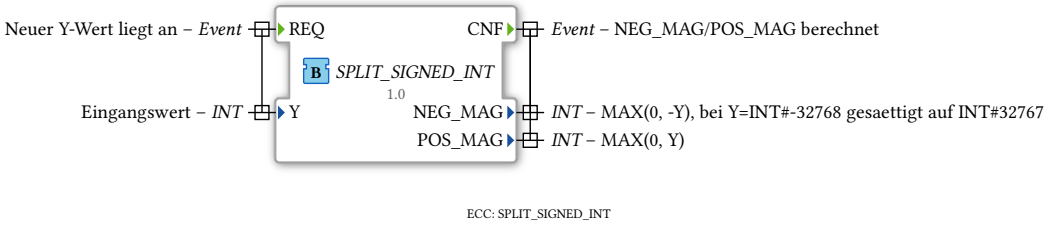
Vergleich mit ähnlichen Bausteinen

- **SPLIT_SIGNED_DINT**: Reine Value/Event-Berechnung für DINT (Basic FB).
- **ADI_SPLIT_SIGNED**: Composite-FB Wrapper mit Adapter-Schnittstellen (adapter::types::unidirectional::ADI) und automatischer Ereignis-Filterung per D-Flip-Flop (E_D_FF_ANY) je Ausgangsseite.

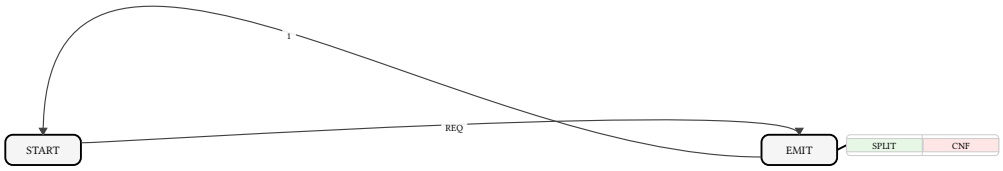
Fazit

SPLIT_SIGNED_DINT bietet eine robuste, überlaufsichere Aufteilung von DINT-Werten in einseitige Betragskomponenten.

SPLIT_SIGNED_INT



ECC: SPLIT_SIGNED_INT



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **SPLIT_SIGNED_INT** teilt einen vorzeichenbehafteten `INT`-Eingangswert `Y` in zwei einseitige Betragswerte auf:

- `NEG_MAG = MAX(0, -Y)` (Betrag der negativen Auslenkung)
- `POS_MAG = MAX(0, Y)` (Betrag der positiven Auslenkung)

Er dient als primärer Berechnungsbaustein für gerichtete Anzeigen (z. B. getrennte Balkenanzeigen / Bargraphen für Links- und Rechtsauslenkung aus einer Mittelstellung) oder getrennte Ansteuerungen (z. B. für Heben/Senken oder Vorwärts/Rückwärts).

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
`REQ`	`Event`	Ausführung der Aufteilung anfordern (mit `Y`)

Ereignis-Ausgänge

Name	Typ	Kommentar
`CNF`	`Event`	Bestätigung der Berechnung (mit `NEG_MAG`, `POS_MAG`)

Daten-Eingänge

Name	Typ	Kommentar
Y	INT	Vorzeichenbehafteter Eingangsmesswert

Daten-Ausgänge

Name	Typ	Kommentar
NEG_MAG	INT	MAX(0, -Y) – Betrag der negativen Auslenkung
POS_MAG	INT	MAX(0, Y) – Betrag der positiven Auslenkung

Funktionsweise

Beim Eintreffen des Ereignisses REQ führt der Baustein folgende Aufteilungslogik aus:

```
IF Y = INT#-32768 THEN
    NEG_MAG := INT#32767;
ELSE
    NEG_MAG := MAX(INT#0, -Y);
END_IF;
POS_MAG := MAX(INT#0, Y);
```

Der Baustein feuert bei jedem REQ-Ereignis die Bestätigung CNF. Eine Ereignis-Filterung bei gleichbleibenden Werten findet auf dieser Basisebene nicht statt (diese wird im Adapter-Wrapper **AI_SPLIT_SIGNED** über **E_D_FF_ANY** D-Flip-Flops realisiert).

Überlaufbehandlung & Sättigung

Bei Zweierkomplement-Ganzzahltypen ist der Betrag des negativen Minimums (-32768) um 1 größer als das darstellbare positive Maximum (32767). Eine mathematische Negation von INT#-32768 würde ohne Sonderbehandlung zu einem Ganzzahlüberlauf und damit zu einem fehlerhaften Vorzeichenwechsel führen.

Der **SPLIT_SIGNED_INT** fängt diesen Sonderfall explizit ab:

- Wenn Y = INT#-32768, wird NEG_MAG auf den maximal möglichen positiven Wert INT#32767 gesättigt.
- Regelungstechnischer Vorteil: Sättigung am Messbereichsende statt Vorzeichenumkehr oder Überlauf.

Anwendungsszenarien

- Aufteilung von Vorzeichen-Messwerten für getrennte Visualisierungselemente (z. B. Links/Rechts-Bargraph).
- Ansteuerung von zwei unidirektionalen Aktoren (z. B. Ventil Heben vs. Ventil Senken) aus einem bipolaren Sollwert.
- Vorverarbeitung in Regelkreisen mit richtungsabhängigen Totbändern oder Kennlinien.

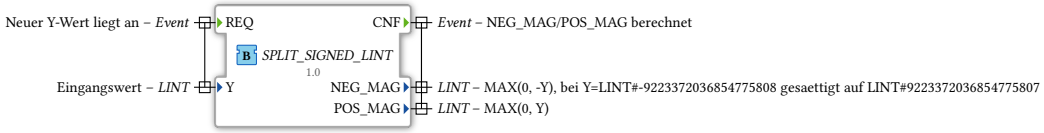
Vergleich mit ähnlichen Bausteinen

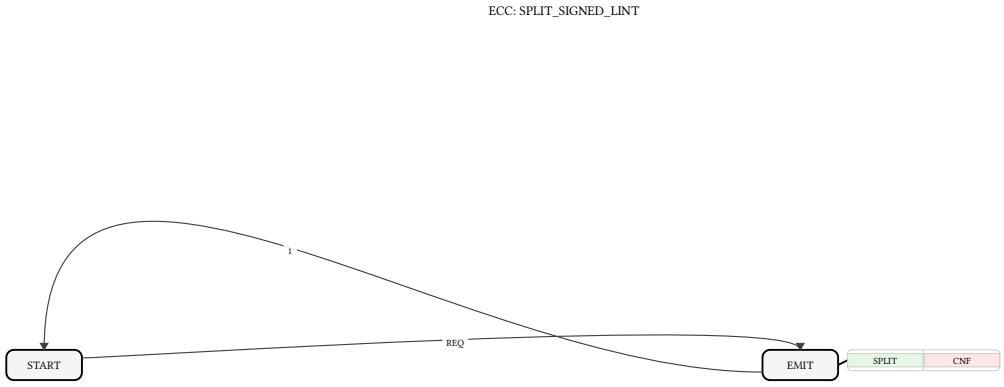
- **SPLIT_SIGNED_INT**: Reine Value/Event-Berechnung für INT (Basic FB).
- **AI_SPLIT_SIGNED**: Composite-FB Wrapper mit Adapter-Schnittstellen (adapter::types::unidirectional::AI) und automatischer Ereignis-Filterung per D-Flip-Flop (E_D_FF_ANY) je Ausgangsseite.

Fazit

SPLIT_SIGNED_INT bietet eine robuste, überlaufsichere Aufteilung von INT-Werten in einseitige Betragskomponenten.

SPLIT_SIGNED_LINT





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **SPLIT_SIGNED_LINT** teilt einen vorzeichenbehafteten `LINT`-Eingangswert `Y` in zwei einseitige Betragswerte auf:

- `NEG_MAG = MAX(0, -Y)` (Betrag der negativen Auslenkung)
- `POS_MAG = MAX(0, Y)` (Betrag der positiven Auslenkung)

Er dient als primärer Berechnungsbaustein für gerichtete Anzeigen (z. B. getrennte Balkenanzeigen / Bargraphen für Links- und Rechtsauslenkung aus einer Mittelstellung) oder getrennte Ansteuerungen (z. B. für Heben/Senken oder Vorwärts/Rückwärts).

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
`REQ`	`Event`	Ausführung der Aufteilung anfordern (mit `Y`)

Ereignis-Ausgänge

Name	Typ	Kommentar
`CNF`	`Event`	Bestätigung der Berechnung (mit `NEG_MAG`, `POS_MAG`)

Daten-Eingänge

Name	Typ	Kommentar
`Y`	`LINT`	Vorzeichenbehafteter Eingangsmesswert

Daten-Ausgänge

Name	Typ	Kommentar

NEG_MAG	LINT	MAX(0, -Y) – Betrag der negativen Auslenkung
POS_MAG	LINT	MAX(0, Y) – Betrag der positiven Auslenkung

Funktionsweise

Beim Eintreffen des Ereignisses REQ führt der Baustein folgende Aufteilungslogik aus:

```
IF Y = LINT#-9223372036854775808 THEN
    NEG_MAG := LINT#9223372036854775807;
ELSE
    NEG_MAG := MAX(LINT#0, -Y);
END_IF;
POS_MAG := MAX(LINT#0, Y);
```

Der Baustein feuert bei jedem REQ-Ereignis die Bestätigung CNF. Eine Ereignis-Filterung bei gleichbleibenden Werten findet auf dieser Basisebene nicht statt (diese wird im Adapter-Wrapper **ALI_SPLIT_SIGNED** über **E_D_FF_ANY** D-Flip-Flops realisiert).

Überlaufbehandlung & Sättigung

Bei Zweierkomplement-Ganzzahltypen ist der Betrag des negativen Minimums (-9223372036854775808) um 1 größer als das darstellbare positive Maximum (9223372036854775807). Eine mathematische Negation von LINT#-9223372036854775808 würde ohne Sonderbehandlung zu einem Ganzzahlüberlauf und damit zu einem fehlerhaften Vorzeichenwechsel führen.

Der **SPLIT_SIGNED_LINT** fängt diesen Sonderfall explizit ab:

- Wenn Y = LINT#-9223372036854775808, wird NEG_MAG auf den maximal möglichen positiven Wert LINT#9223372036854775807 gesättigt.
- Regelungstechnischer Vorteil: Sättigung am Messbereichsende statt Vorzeichenumkehr oder Überlauf.

Anwendungsszenarien

- Aufteilung von Vorzeichen-Messwerten für getrennte Visualisierungselemente (z. B. Links/Rechts-Bargraph).
- Ansteuerung von zwei unidirektionalen Aktoren (z. B. Ventil Heben vs. Ventil Senken) aus einem bipolaren Sollwert.
- Vorverarbeitung in Regelkreisen mit richtungsabhängigen Totbändern oder Kennlinien.

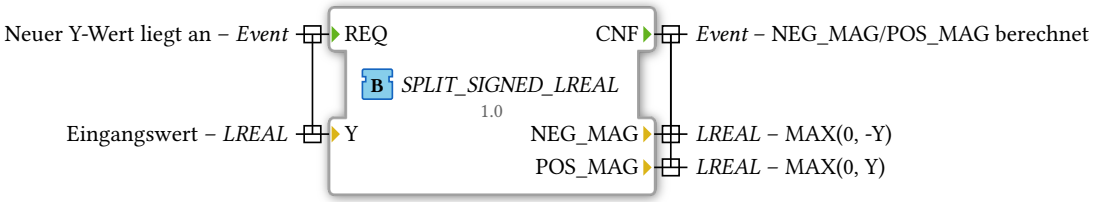
Vergleich mit ähnlichen Bausteinen

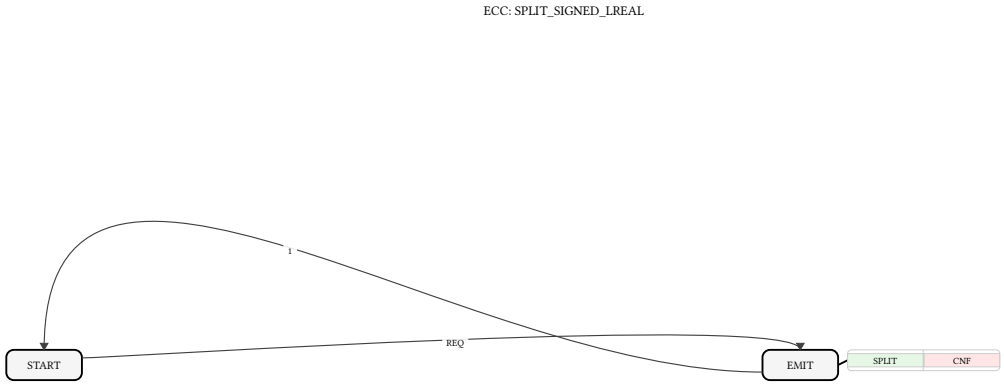
- **SPLIT_SIGNED_LINT**: Reine Value/Event-Berechnung für LINT (Basic FB).
- **ALI_SPLIT_SIGNED**: Composite-FB Wrapper mit Adapter-Schnittstellen (adapter::types::unidirectional::ALI) und automatischer Ereignis-Filterung per D-Flip-Flop (E_D_FF_ANY) je Ausgangsseite.

Fazit

SPLIT_SIGNED_LINT bietet eine robuste, überlaufsichere Aufteilung von LINT-Werten in einseitige Betragskomponenten.

SPLIT_SIGNED_LREAL





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **SPLIT_SIGNED_LREAL** teilt einen vorzeichenbehafteten `LREAL`-Eingangswert `Y` in zwei einseitige Betragswerte auf:

- `NEG_MAG = MAX(0, -Y)` (Betrag der negativen Auslenkung)
- `POS_MAG = MAX(0, Y)` (Betrag der positiven Auslenkung)

Er dient als primärer Berechnungsbaustein für gerichtete Anzeigen (z. B. getrennte Balkenanzeigen / Bargraphen für Links- und Rechtsauslenkung aus einer Mittelstellung) oder getrennte Ansteuerungen (z. B. für Heben/Senken oder Vorwärts/Rückwärts).

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
`REQ`	`Event`	Ausführung der Aufteilung anfordern (mit `Y`)

Ereignis-Ausgänge

Name	Typ	Kommentar
`CNF`	`Event`	Bestätigung der Berechnung (mit `NEG_MAG`, `POS_MAG`)

Daten-Eingänge

Name	Typ	Kommentar
`Y`	`LREAL`	Vorzeichenbehafteter Eingangsmesswert

Daten-Ausgänge

Name	Typ	Kommentar

NEG_MAG	LREAL	MAX(0, -Y) – Betrag der negativen Auslenkung
POS_MAG	LREAL	MAX(0, Y) – Betrag der positiven Auslenkung

Funktionsweise

Beim Eintreffen des Ereignisses REQ führt der Baustein folgende Aufteilungslogik aus:

```
NEG_MAG := MAX(LREAL#0.0, -Y);
POS_MAG := MAX(LREAL#0.0, Y);
```

Der Baustein feuert bei jedem REQ-Ereignis die Bestätigung CNF. Eine Ereignis-Filterung bei gleichbleibenden Werten findet auf dieser Basisebene nicht statt (diese wird im Adapter-Wrapper **ALR_SPLIT_SIGNED** über **E_D_FF_ANY** D-Flip-Flops realisiert).

Gleitkomma-Verhalten

Da LREAL ein Gleitkommatyp ist, tritt das Asymmetrie-Problem von Zweierkomplement-Ganzzahlen nicht auf. Die Negation -Y ist im gesamten Wertebereich ohne Überlauf darstellbar.

Anwendungsszenarien

- Aufteilung von Vorzeichen-Messwerten für getrennte Visualisierungselemente (z. B. Links/Rechts-Bargraph).
- Ansteuerung von zwei unidirektionalen Aktoren (z. B. Ventil Heben vs. Ventil Senken) aus einem bipolaren Sollwert.
- Vorverarbeitung in Regelkreisen mit richtungsabhängigen Totbändern oder Kennlinien.

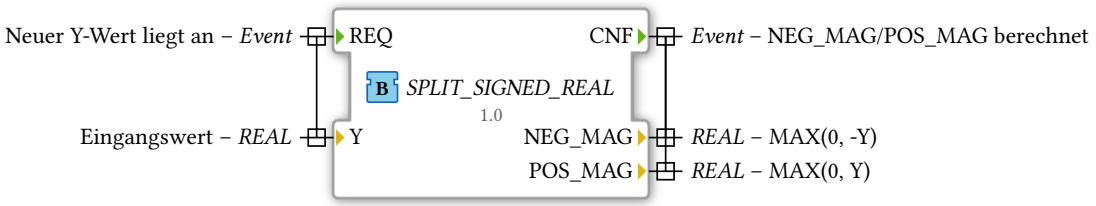
Vergleich mit ähnlichen Bausteinen

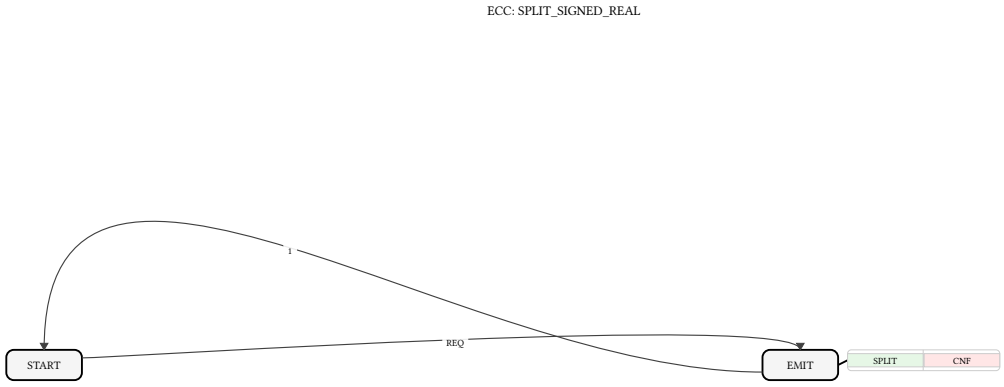
- **SPLIT_SIGNED_LREAL**: Reine Value/Event-Berechnung für LREAL (Basic FB).
- **ALR_SPLIT_SIGNED**: Composite-FB Wrapper mit Adapter-Schnittstellen (adapter::types::unidirectional::ALR) und automatischer Ereignis-Filterung per D-Flip-Flop (E_D_FF_ANY) je Ausgangsseite.

Fazit

SPLIT_SIGNED_LREAL bietet eine robuste, überlaufsichere Aufteilung von LREAL-Werten in einseitige Betragskomponenten.

SPLIT_SIGNED_REAL





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **SPLIT_SIGNED_REAL** teilt einen vorzeichenbehafteten `REAL`-Eingangswert `Y` in zwei einseitige Betragswerte auf:

- `NEG_MAG = MAX(0, -Y)` (Betrag der negativen Auslenkung)
- `POS_MAG = MAX(0, Y)` (Betrag der positiven Auslenkung)

Er dient als primärer Berechnungsbaustein für gerichtete Anzeigen (z. B. getrennte Balkenanzeigen / Bargraphen für Links- und Rechtsauslenkung aus einer Mittelstellung) oder getrennte Ansteuerungen (z. B. für Heben/Senken oder Vorwärts/Rückwärts).

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
`REQ`	`Event`	Ausführung der Aufteilung anfordern (mit `Y`)

Ereignis-Ausgänge

Name	Typ	Kommentar
`CNF`	`Event`	Bestätigung der Berechnung (mit `NEG_MAG`, `POS_MAG`)

Daten-Eingänge

Name	Typ	Kommentar
`Y`	`REAL`	Vorzeichenbehafteter Eingangsmesswert

Daten-Ausgänge

Name	Typ	Kommentar

NEG_MAG	REAL	MAX(0, -Y) – Betrag der negativen Auslenkung
POS_MAG	REAL	MAX(0, Y) – Betrag der positiven Auslenkung

Funktionsweise

Beim Eintreffen des Ereignisses REQ führt der Baustein folgende Aufteilungslogik aus:

```
NEG_MAG := MAX(REAL#0.0, -Y);
POS_MAG := MAX(REAL#0.0, Y);
```

Der Baustein feuert bei jedem REQ-Ereignis die Bestätigung CNF. Eine Ereignis-Filterung bei gleichbleibenden Werten findet auf dieser Basisebene nicht statt (diese wird im Adapter-Wrapper **AR_SPLIT_SIGNED** über **E_D_FF_ANY** D-Flip-Flops realisiert).

Gleitkomma-Verhalten

Da REAL ein Gleitkommatyp ist, tritt das Asymmetrie-Problem von Zweierkomplement-Ganzzahlen nicht auf. Die Negation -Y ist im gesamten Wertebereich ohne Überlauf darstellbar.

Anwendungsszenarien

- Aufteilung von Vorzeichen-Messwerten für getrennte Visualisierungselemente (z. B. Links/Rechts-Bargraph).
- Ansteuerung von zwei unidirektionalen Aktoren (z. B. Ventil Heben vs. Ventil Senken) aus einem bipolaren Sollwert.
- Vorverarbeitung in Regelkreisen mit richtungsabhängigen Totbändern oder Kennlinien.

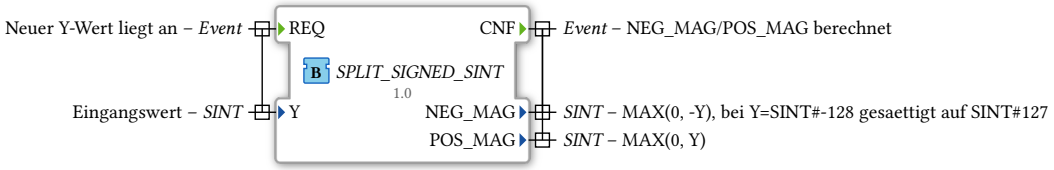
Vergleich mit ähnlichen Bausteinen

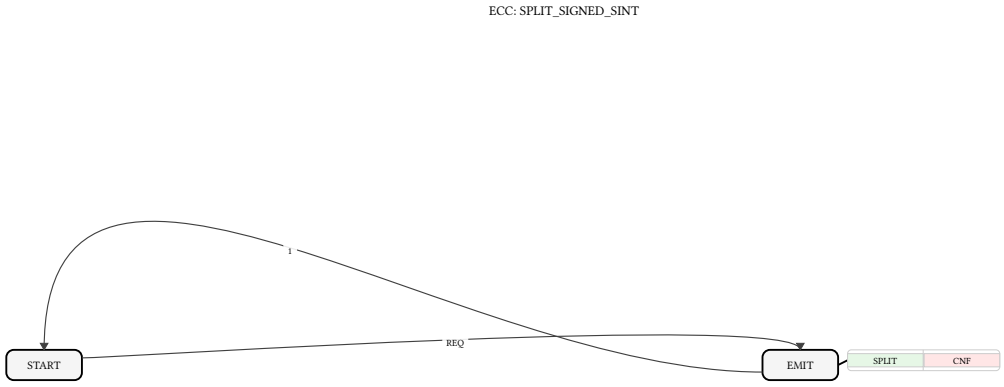
- **SPLIT_SIGNED_REAL**: Reine Value/Event-Berechnung für REAL (Basic FB).
- **AR_SPLIT_SIGNED**: Composite-FB Wrapper mit Adapter-Schnittstellen (adapter::types::unidirectional::AR) und automatischer Ereignis-Filterung per D-Flip-Flop (E_D_FF_ANY) je Ausgangsseite.

Fazit

SPLIT_SIGNED_REAL bietet eine robuste, überlaufsichere Aufteilung von REAL-Werten in einseitige Betragskomponenten.

SPLIT_SIGNED_SINT





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **SPLIT_SIGNED_SINT** teilt einen vorzeichenbehafteten `SINT`-Eingangswert `Y` in zwei einseitige Betragswerte auf:

- `NEG_MAG = MAX(0, -Y)` (Betrag der negativen Auslenkung)
- `POS_MAG = MAX(0, Y)` (Betrag der positiven Auslenkung)

Er dient als primärer Berechnungsbaustein für gerichtete Anzeigen (z. B. getrennte Balkenanzeigen / Bargraphen für Links- und Rechtsauslenkung aus einer Mittelstellung) oder getrennte Ansteuerungen (z. B. für Heben/Senken oder Vorwärts/Rückwärts).

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
`REQ`	`Event`	Ausführung der Aufteilung anfordern (mit `Y`)

Ereignis-Ausgänge

Name	Typ	Kommentar
`CNF`	`Event`	Bestätigung der Berechnung (mit `NEG_MAG`, `POS_MAG`)

Daten-Eingänge

Name	Typ	Kommentar
`Y`	`SINT`	Vorzeichenbehafteter Eingangsmesswert

Daten-Ausgänge

Name	Typ	Kommentar

NEG_MAG	SINT	MAX(0, -Y) – Betrag der negativen Auslenkung
POS_MAG	SINT	MAX(0, Y) – Betrag der positiven Auslenkung

Funktionsweise

Beim Eintreffen des Ereignisses REQ führt der Baustein folgende Aufteilungslogik aus:

```
IF Y = SINT#-128 THEN
    NEG_MAG := SINT#127;
ELSE
    NEG_MAG := MAX(SINT#0, -Y);
END_IF;
POS_MAG := MAX(SINT#0, Y);
```

Der Baustein feuert bei jedem REQ-Ereignis die Bestätigung CNF. Eine Ereignis-Filterung bei gleichbleibenden Werten findet auf dieser Basisebene nicht statt (diese wird im Adapter-Wrapper **AS_SPLIT_SIGNED** über **E_D_FF_ANY** D-Flip-Flops realisiert).

Überlaufbehandlung & Sättigung

Bei Zweierkomplement-Ganzzahltypen ist der Betrag des negativen Minimums (-128) um 1 größer als das darstellbare positive Maximum (127). Eine mathematische Negation von SINT#-128 würde ohne Sonderbehandlung zu einem Ganzzahlüberlauf und damit zu einem fehlerhaften Vorzeichenwechsel führen.

Der **SPLIT_SIGNED_SINT** fängt diesen Sonderfall explizit ab:

- Wenn Y = SINT#-128, wird NEG_MAG auf den maximal möglichen positiven Wert SINT#127 gesättigt.
- Regelungstechnischer Vorteil: Sättigung am Messbereichsende statt Vorzeichenumkehr oder Überlauf.

Anwendungsszenarien

- Aufteilung von Vorzeichen-Messwerten für getrennte Visualisierungselemente (z. B. Links/Rechts-Bargraph).
- Ansteuerung von zwei unidirektionalen Aktoren (z. B. Ventil Heben vs. Ventil Senken) aus einem bipolaren Sollwert.
- Vorverarbeitung in Regelkreisen mit richtungsabhängigen Totbändern oder Kennlinien.

Vergleich mit ähnlichen Bausteinen

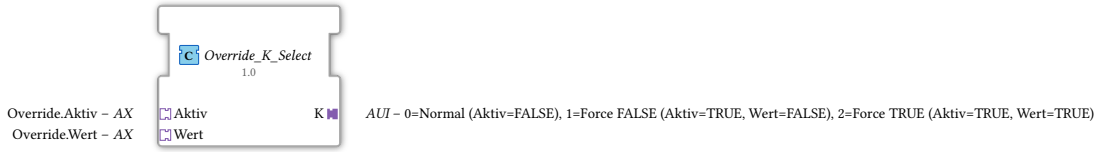
- **SPLIT_SIGNED_SINT**: Reine Value/Event-Berechnung für SINT (Basic FB).
- **AS_SPLIT_SIGNED**: Composite-FB Wrapper mit Adapter-Schnittstellen (adapter::types::unidirectional::AS) und automatischer Ereignis-Filterung per D-Flip-Flop (E_D_FF_ANY) je Ausgangsseite.

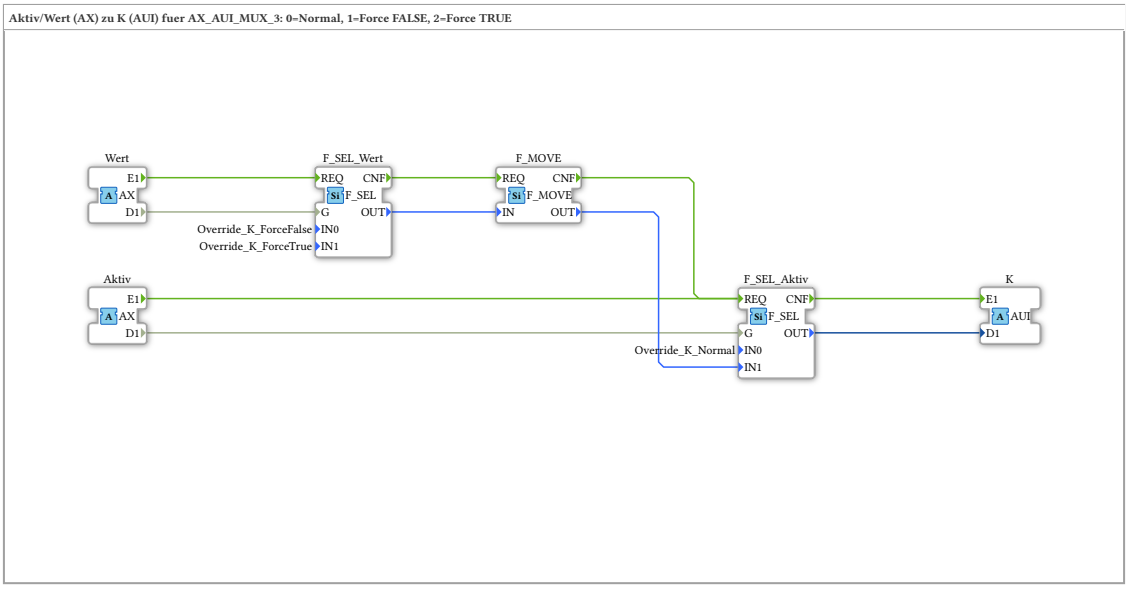
Fazit

SPLIT_SIGNED_SINT bietet eine robuste, überlaufsichere Aufteilung von SINT-Werten in einseitige Betragskomponenten.

2.1.11.3 OverrideK

Override_K_Select





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `Override_K_Select` wandelt ein Paar boolescher Override-Signale (`Aktiv`/`Wert`, je als AX-Adapter) in einen einzelnen `UINT`-Selektor `K` (als AUI-Adapter) um, wie ihn ein `AX_AUI_MUX_3` erwartet. Er kapselt damit das im Projekt wiederkehrende Muster „ein Sensorwert kann per RW-Override entweder auf `FALSE` oder `TRUE` erzwungen werden“ (siehe `Sensor.Leermelder.Override.*` in `docs/datenpunkte.md` der Getreideannahme-Anwendung) in einem einzigen, wiederverwendbaren Baustein, statt die Umrechnung an jeder Verwendungsstelle erneut zu verdrahten.

Schnittstellenstruktur

Ereignis-Eingänge

Keine eigenen Ereignis-Eingänge. Ereignisse laufen ausschließlich über die Adapter-Sockets `Aktiv`/`Wert`.

Ereignis-Ausgänge

Keine eigenen Ereignis-Ausgänge. Das Ergebnis wird über den Adapter-Plug `K` ausgegeben.

Daten-Eingänge

Keine eigenen Daten-Eingänge. Alle Daten werden über die Adapter-Sockets übertragen.

Daten-Ausgänge

Keine eigenen Daten-Ausgänge. Das Ergebnis wird über den Adapter-Plug `K` ausgegeben.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	`Aktiv`	`adapter::types::unidirectional::AX`	`Override.Aktiv`
Socket (Eingang)	`Wert`	`adapter::types::unidirectional::AX`	`Override.Wert`
Plug (Ausgang)	`K`	`adapter::types::unidirectional::AUI`	0=Normal (`Aktiv`=FALSE), 1=Force FALSE (`Aktiv`=TRUE, `Wert`=FALSE), 2=Force TRUE (`Aktiv`=TRUE, `Wert`=TRUE)

Funktionsweise

Intern arbeitet der Baustein mit zwei verketteten, generischen `F_SEL`-Bausteinen (Standard- IEC-61131-3-Selektor `OUT := G ? IN1 : IN0`) statt mit eigener ST-Logik:

1. **F_SEL_Wert**: `G := Wert.D1`. Liefert `Override_K_ForceFalse` (1) wenn `Wert = FALSE`, sonst `Override_K_ForceTrue` (2). Ausgelöst durch `Wert.E1`.
2. **F_MOVE** (Attribut `DataType="UINT"`): Übernimmt `F_SEL_Wert.OUT`, ausgelöst durch `F_SEL_Wert.CNF`. Dient als Typ-Anker für das generische `ANY`-Ergebnis von `F_SEL` (siehe Technische Besonderheiten) und stößt anschließend `F_SEL_Aktiv` erneut an.
3. **F_SEL_Aktiv**: `G := Aktiv.D1`. Liefert `Override_K_Normal` (0) wenn `Aktiv = FALSE`, sonst den zuletzt von `F_MOVE` gehaltenen Wert (1 oder 2). Ausgelöst sowohl direkt durch `Aktiv.E1` als auch indirekt über `F_MOVE.CNF`, sodass eine Änderung von `Wert` auch bei unverändertem `Aktiv` sofort ein neues `K` erzeugt.
4. `F_SEL_Aktiv.OUT`/`F_SEL_Aktiv.CNF` werden direkt auf die Datenausgangsvariable `K.D1` bzw. das Ereignis `K.E1` des eigenen Plugs geschrieben — ohne eine zusätzliche `_TO_AUI`-Konvertierungs- FB dazwischenzuschalten (siehe Technische Besonderheiten).

Technische Besonderheiten

- **Direktes Schreiben auf die eigenen Plug-Pins.** Statt eine `UINT_TO_AUI`-artige Konvertierungs-FB zwischenzuschalten, verbindet der Baustein `F_SEL_Aktiv.OUT`/`.CNF` direkt mit `K.D1`/`K.E1` — den vom `AUI`-Adaptertyp selbst deklarierten Event-/Datenvariablen des eigenen Plugs. Dasselbe Prinzip nutzen auch const_AX_TRUE und const_AX_FALSE für ihren `OUT`-Plug.
- **`F_MOVE` als Typ-Anker für `F_SEL`s generisches `ANY`-Ergebnis.** `F_SEL.OUT` ist generisch (`ANY`); ohne einen nachgeschalteten `F_MOVE` mit festem `DataType`-Attribut lässt sich der Wert nicht sauber als `UINT` an den zweiten `F_SEL`-Aufruf weiterreichen.
- **Die K-Werte sind benannte Konstanten, keine Literale.** `Override_K_Normal`/`Override_K_ForceFalse`/`Override_K_ForceTrue` kommen aus dem globalen Konstantenpaket Override_K (`adapter::OverrideK::const`), nicht als `UINT#0`/`UINT#1`/ `UINT#2`-Literale direkt in den `F_SEL`-Parametern.
- **Kein eigenes `REQ`/`CNF`.** Der Baustein reagiert ausschließlich auf die Ereignisse der Sockets selbst (`Aktiv.E1`, `Wert.E1`) — es gibt keinen externen Trigger-Eingang.

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine (Composite-FB ohne ECC); sein Verhalten ist rein kombinatorisch, neu berechnet bei jeder Änderung von `Aktiv` oder `Wert`.

Aktiv	Wert	K
FALSE	—	0 (Normal)
TRUE	FALSE	1 (Force FALSE)
TRUE	TRUE	2 (Force TRUE)

Anwendungsszenarien

- **Sensor-Override mit RW-OPC-UA-Knoten**: Ein per `AX_SUBSCRIBE_1` abonnenntes `Aktiv`/`Wert`-Paar (z. B. `Sensor.Leermelder.Override.Aktiv`/`.Wert`) direkt in den `K`-Eingang eines `AX_AUI_MUX_3` speisen, dessen drei Eingänge Normalwert, `FALSE` und `TRUE` bereitstellen.
- **Diagnose-/Testseiten**: Jeder boolesche Sensor, der über eine Weboberfläche testweise auf einen festen Wert erzwungen werden soll, kann dasselbe `Aktiv`/`Wert`/`K`-Muster wiederverwenden.

⚖️ Vergleich mit ähnlichen Bausteinen

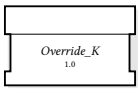
- **AX_AUI_MUX_3**: der eigentliche Multiplexer, den `Override_K_Select` mit seinem `K`-Ausgang ansteuert. `Override_K_Select` berechnet nur den Selektor, wählt aber selbst keinen Wert aus.
- **const_AX_TRUE** / **const_AX_FALSE**: liefern die festen `IN2`/`IN3`-Werte (Force FALSE/TRUE), die der von `Override_K_Select` berechnete Selektor am `AX_AUI_MUX_3` auswählt.
- **Override_K**: das globale Konstantenpaket mit den drei benannten K-Werten, die dieser Baustein intern verwendet.

Fazit

`Override_K_Select` bündelt die Umrechnung eines Aktiv/Wert-Override-Paars in einen MUX-Selektor in einem einzigen, wiederverwendbaren Baustein, gebaut ausschließlich aus Standard-Bausteinen (`F_SEL`, `F_MOVE`) ohne eigene ST-Algorithmen. Kombiniert mit `AX_AUI_MUX_3` und den `const_AX_TRUE`/`const_AX_FALSE`-Konstanten ergibt sich damit das komplette, wiederverwendbare Muster für einen per OPC-UA erzwingbaren booleschen Sensorwert.

const

Override_K



• – ◦ • – ◦ • – ◦ •

Einleitung

Das globale Konstantenpaket `Override_K` stellt die drei benannten `UINT`-Werte bereit, mit denen Override_K_Select seinen `K`-Ausgang für `AX_AUI_MUX_3` befüllt. Statt `UINT#0`/`UINT#1`/`UINT#2` als Literale in den `F_SEL`-Parametern zu verdrahten, referenziert der Baustein diese drei Konstanten über ihren sprechenden Namen. Das Paket ist im Namensraum `adapter::OverrideK::const` definiert.

Schnittstellenstruktur

Da es sich um ein Paket mit globalen Konstanten handelt, besitzt es keine ereignis- oder datenflussbasierten Schnittstellen im herkömmlichen Sinne eines Funktionsbausteins.

Ereignis-Eingänge

Nicht vorhanden.

Ereignis-Ausgänge

Nicht vorhanden.

Daten-Eingänge

Nicht vorhanden.

Daten-Ausgänge

Nicht vorhanden.

Adapter

Nicht vorhanden.

Funktionsweise

Die Konstanten werden beim Kompilieren durch ihre fest definierten Werte ersetzt:

- `Override_K_Normal`: `UINT#0` — K-Wert für „kein Override“ (`AX_AUI_MUX_3.IN1`, Normalwert).
- `Override_K_ForceFalse`: `UINT#1` — K-Wert für „Override auf FALSE erzwungen“ (`AX_AUI_MUX_3.IN2`).
- `Override_K_ForceTrue`: `UINT#2` — K-Wert für „Override auf TRUE erzwungen“ (`AX_AUI_MUX_3.IN3`).

Technische Besonderheiten

- **Direkte Kopplung an `AX_AUI_MUX_3`-Eingangsreihenfolge.** Die Zahlenwerte 0/1/2 entsprechen exakt der Eingangsnummerierung von `AX_AUI_MUX_3` (`IN1`/`IN2`/`IN3`) — eine Änderung dieser Reihenfolge in Override_K_Select müsste hier mit gepflegt werden.
- **Nur von `Override_K_Select` verwendet.** Anders als z. B. `OPC_UA_Endpoints` ist dieses Paket eng an einen einzigen Baustein gekoppelt, nicht an mehrere unabhängige Verwendungsstellen.

Zustandsübersicht

Da es sich um statische, konstante Werte handelt, existiert kein Zustandsautomat.

Anwendungsszenarien

- Referenzierung in den `IN0`/`IN1-Parameter`-Werten der beiden `F_SEL`-Bausteine in Override_K_Select.
- Vergleichswerte, falls ein Konsument den rohen `K`-Wert eines `AX_AUI_MUX_3` auswerten möchte, statt nur den gemuxten Ausgangswert zu verwenden.

⚖️ Vergleich mit ähnlichen Bausteinen

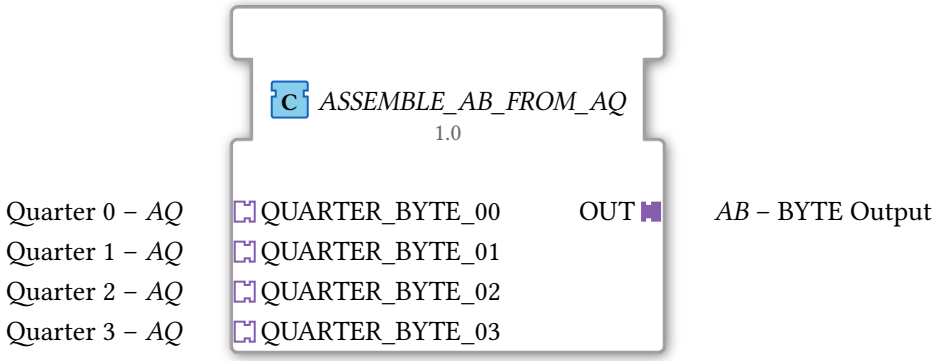
- **IDs** (ISOBUS-UT-Bibliothek): folgt demselben GCF- Muster (benannte Konstanten statt Literale), deckt aber ISOBUS-Objekt-IDs ab statt eines MUX-Selektors.

Fazit

`Override_K` ist ein kleines, eng gekoppeltes Konstantenpaket, das die drei K-Werte von `Override_K_Select` benennt und damit lesbarer und wartbarer macht als verstreute `UINT#0/1/2`- Literale.

2.1.11.4 assembling

ASSEMBLE_AB_FROM_AQ



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASSEMBLE_AB_FROM_AQ** dient der Zusammenführung von vier **AQ**-Quartal-Adaptern (Quarter Byte) zu einem einzelnen **AB**-Byte-Adapter. Er kapselt die Logik, aus vier eingehenden 4-Bit-Werten ein vollständiges Byte zu assemblieren und über eine standardisierte Adapter-Schnittstelle bereitzustellen. Der Baustein ist speziell für die Verwendung in verteilten Automatisierungssystemen nach IEC 61499 ausgelegt.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine dedizierten Ereignis-Eingänge auf der obersten Ebene.
Alle eingehenden Ereignisse werden über die **Adapter-Sockets** übertragen. Jeder Applikationsadapter vom Typ `AQ` verfügt über einen Ereignisausgang `E1`, der intern mit dem REQ-Eingang des Assemblierungsbausteins verbunden ist.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Byte-Adapters. Wird aktiv, sobald ein neues Byte vollständig assembliert und über den internen Flipflop getaktet wurde.

Daten-Eingänge

Die Daten werden über die **Adapter-Sockets** in Form von 4-Bit-Werten (Nibbles) eingelesen:

- **QUARTER_BYTE_00.D1**: Nibble für die niederwertigsten Bits (Bits 0..3)
- **QUARTER_BYTE_01.D1**: Nibble für Bits 4..7
- **QUARTER_BYTE_02.D1**: Nibble für Bits 8..11
- **QUARTER_BYTE_03.D1**: Nibble für Bits 12..15

(Hinweis: Die tatsächliche Bit-Reihenfolge kann je nach Implementierung des internen `ASSEMBLE_BYTE_FROM_QUARTERS`-Bausteins variieren; üblicherweise erfolgt die Zusammenführung in aufsteigender Reihenfolge.)

Daten-Ausgänge

- **OUT.D1**: Ausgang des Byte-Adapters. Liefert das fertig zusammengesetzte Byte (8 Bit) als ganzzahligen Wert.

Adapter

Typ	Richtung	Name	Beschreibung
`adapter::types::unidirectional::AQ`	Socket (Eingang)	`QUARTER_BYTE_00` .. `QUARTER_BYTE_03`	Vier Quartal-Adapter, die jeweils ein Nibble und ein Ereignis liefern.
`adapter::types::unidirectional::PAB`	Plug (Ausgang)	`OUT`	Byte-Adapter, der das zusammengesetzte Byte und ein Ereignis bereitstellt.

Funktionsweise

Der FB arbeitet vollständig ereignisgesteuert. Sobald eines der vier angeschlossenen AQ-Adapter ein Ereignis auf seinem `E1`-Ausgang sendet, wird dieses an den internen Baustein `ASSEMBLE_BYTE_FROM_QUARTERS` weitergeleitet (Ereignis `REQ`). Gleichzeitig werden die aktuellen Nibble-Werte von allen vier Quartal-Adapter-Eingängen über Data Connections an den Assemblierungsbaustein übergeben.

Der Assemblierungsbaustein setzt die vier Nibble intern zu einem 8-Bit-Wert zusammen. Nach erfolgreicher Verarbeitung sendet er ein Bestätigungsereignis `CNF` an den flankengesteuerten D-Flipflop (`E_D_FF_ANY`). Der Flipflop übernimmt den Datenwert (vom Assemblierungsausgang) bei steigender Flanke und gibt ihn an den Datenausgang `OUT.D1` weiter. Gleichzeitig wird das Ausgangsereignis `OUT.E1` generiert.

Somit erfolgt die Ausgabe immer taktsynchron: Erst wenn ein vollständiges Byte berechnet wurde, wird es durchgeschaltet. Mehrfache Ereignisse auf verschiedenen Quartal-Eingängen führen zu wiederholten Berechnungen, wobei jeweils alle vier aktuellen Nibble-Werte neu verarbeitet werden.

Technische Besonderheiten

- **Interne Kaskadierung**: Der FB verwendet zwei interne Bausteine – `ASSEMBLE_BYTE_FROM_QUARTERS` für die Bit-Kombination und `E_D_FF_ANY` zur flankengesteuerten Ausgabe.
- **Ereignissynchronisation**: Jedes eingehende Ereignis triggert die Assemblierung; es ist nicht erforderlich, dass alle vier Adapter gleichzeitig ein Ereignis liefern – der Baustein arbeitet mit den jeweils aktuellen Daten.
- **Adapter-Schnittstelle**: Die Ein- und Ausgabe erfolgt ausschließlich über unidirektionale Adapter, was die Wiederverwendung in Applikationsnetzwerken vereinfacht.
- **Keine Zustandsspeicherung im FB selbst**: Der FB ist ein reiner Netzwerkbaustein, der Logik vollständig delegiert.

Zustandsübersicht

Der Baustein besitzt keinen eigenen Zustandsautomaten. Die interne Logik wird durch die Zustände von `ASSEMBLE_BYTE_FROM_QUARTERS` und `E_D_FF_ANY` bestimmt:

1. **Warten auf Eingangsereignis**: Nach Initialisierung oder nach einer Verarbeitung wartet der FB auf ein Ereignis von einem der AQ-Adapter.
2. **Assemblierung**: Bei `REQ` werden die Nibble zusammengesetzt; der D-Flipflop wartet auf das `CNF`-Ereignis.
3. **Ausgabe**: Nach Taktung des Flipflops wird das Ergebnis an den OUT-Adapter gesendet. Danach kehrt der FB in den Wartezustand zurück.

Anwendungsszenarien

- **Zusammenführung von Busdaten**: Wenn vier getrennte 4-Bit-Leitungen (z. B. von Sensoren oder Datenquellen) zu einem Byte zusammengefasst werden müssen.
- **Adapter-basierte Datenintegration**: In Systemen, die auf IEC 61499-Adapterarchitektur setzen, kann dieser FB als generischer „Byte-Assembler“ in der Netzwerkebene dienen.
- **Protokollwandlung**: Von einem Viertel-Byte-Protokoll auf ein Byte-Protokoll, z. B. in der seriellen Kommunikation.
- **Test- und Simulationsumgebungen**: Zum einfachen Zusammenschalten von Testadaptern.

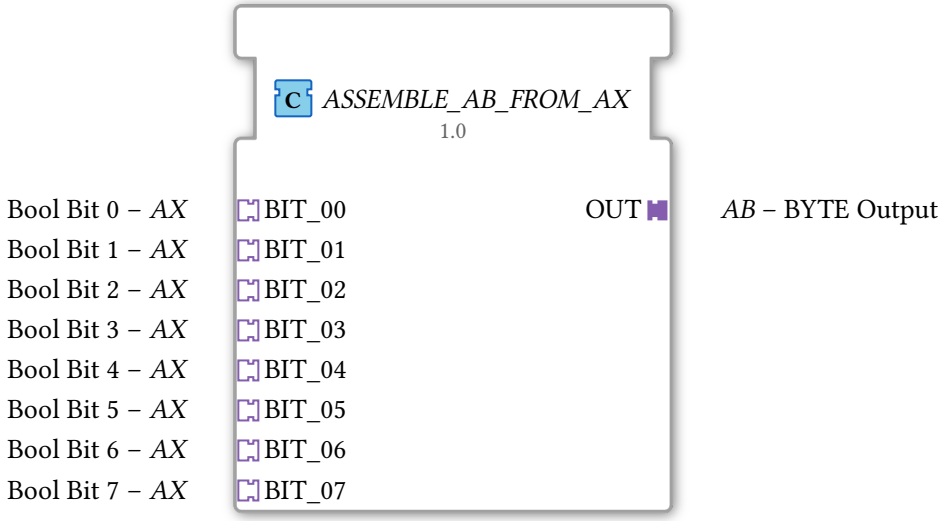
Vergleich mit ähnlichen Bausteinen

- **ASSEMBLE_BYTE_FROM_QUARTERS** (direkt): Dieser Baustein arbeitet ohne Adapter – er erwartet vier separate Daten- und Ereigniseingänge. Der vorliegende FB kapselt diese Schnittstelle in Adapter, was die Modularität erhöht.
- **ARRAY_TO_BYTE** oder ähnliche Funktionsbausteine: Oft auf Basis von Arrays implementiert; hier spezifisch für genau vier Nibble und mit Adapterunterstützung.
- **Eigener Adapter-basierter Assembler**: Falls benötigt, kann `ASSEMBLE_AB_FROM_AQ` durch Anpassung der internen Bausteine leicht für andere Datenbreiten modifiziert werden.

Fazit

Der Funktionsblock **ASSEMBLE_AB_FROM_AQ** bietet eine elegante, adapterbasierte Lösung zur Zusammenführung von vier Viertel-Byte-Datenströmen zu einem vollständigen Byte. Dank der internen Kaskadierung aus Assemblierungslogik und flankengesteuerter Ausgabe ist das Verhalten deterministisch und gut in ereignisgesteuerte Systeme integrierbar. Er eignet sich besonders für modulare Automatisierungsprojekte, die auf IEC 61499-Adapterarchitektur aufbauen und eine saubere Trennung von Daten- und Ereignisflüssen erfordern.

ASSEMBLE_AB_FROM_AX



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASSEMBLE_AB_FROM_AX** fasst acht boolesche Einzelsignale, die über AX-Adapter (unidirektional, Bool) bereitgestellt werden, zu einem Byte zusammen und gibt dieses über einen AB-Adapter (unidirektional, BYTE) aus. Er kapselt die Logik zur Byte-Erzeugung und stellt eine modulare, adapterbasierte Schnittstelle für die Verarbeitung von 8 Bit zur Verfügung.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Ereignisse werden ausschließlich über die Adapter-Sockets empfangen.

Ereignis-Ausgänge

Keine. Ereignisse werden ausschließlich über den Adapter-Plug ausgegeben.

Daten-Eingänge

Keine. Alle Daten werden über die Adapter-Sockets übertragen.

Daten-Ausgänge

Keine. Alle Daten werden über den Adapter-Plug ausgegeben.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	BIT_00	adapter::types::unidirectional::AX	Boolescher Wert für Bit 0 (niederwertigstes Bit)
	BIT_01	adapter::types::unidirectional::AX	Boolescher Wert für Bit 1
	BIT_02	adapter::types::unidirectional::AX	Boolescher Wert für Bit 2
	BIT_03	adapter::types::unidirectional::AX	Boolescher Wert für Bit 3
	BIT_04	adapter::types::unidirectional::AX	Boolescher Wert für Bit 4
	BIT_05	adapter::types::unidirectional::AX	Boolescher Wert für Bit 5
	BIT_06	adapter::types::unidirectional::AX	Boolescher Wert für Bit 6
	BIT_07	adapter::types::unidirectional::AX	Boolescher Wert für Bit 7 (höchstwertiges Bit)
Plug (Ausgang)	OUT	adapter::types::unidirectional::AB	Zusammengesetztes Byte (BYTE)

Jeder AX-Adapter stellt über seinen Daten-Eingang `D1` den Bool-Wert und über seinen Ereignis-Eingang `E1` das zugehörige Ereignis bereit. Der AB-Adapter liefert über seinen Daten-Ausgang `D1` das Byte und über den Ereignis-Ausgang `E1` das Bestätigungsereignis.

Funktionsweise

Der Funktionsblock ist als zusammengesetzter Baustein realisiert und besteht intern aus:

1. **ASSEMBLE_BYTE_FROM_BOOLS** – Nimmt die acht Bool-Werte (von `BIT_00` bis `BIT_07`) entgegen und setzt daraus ein Byte zusammen.
2. **E_D_FF_ANY** – Ein D-Flipflop, das den berechneten Byte-Wert puffert und erst bei einer positiven Flanke am Takteingang weitergibt.

Ablauf:

- Sobald an **einem** der AX-Adapter (z. B. `BIT_00`) ein Ereignis eintrifft, wird dieses an den `REQ`-Eingang des internen Bausteins **ASSEMBLE_BYTE_FROM_BOOLS** weitergeleitet.
- Der interne Baustein berechnet aus den aktuellen Bool-Werten aller acht Adapter das Byte und legt es an seinem Datenausgang an.
- Nach Abschluss der Berechnung sendet `ASSEMBLE_BYTE_FROM_BOOLS` ein `CNF`-Ereignis, das den Takteingang (`CLK`) des D-Flipflops **E_D_FF_ANY** triggert.
- Das Flipflop übernimmt den aktuellen Byte-Wert und gibt ihn an seinem Ausgang `Q` aus.
- Gleichzeitig wird das Ereignis `EO` des Flipflops an den Ereignis-Ausgang `OUT.E1` des AB-Adapters weitergegeben.

Dadurch wird sichergestellt, dass das Ausgangs-Byte nur dann aktualisiert wird, wenn sich ein Eingangs-Bit geändert hat, und dass die Ausgabe stabil und synchronisiert erfolgt.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle** – Der Baustein verwendet ausschließlich Adapter (`AX`/`AB`) anstelle einzelner Ereignis- und Daten-Ports. Dies ermöglicht eine einfache Kapselung und Wiederverwendung in modularen Entwürfen.
- **Internes D-Flipflop** – Das Flipflop verhindert Zwischenzustände und gibt das fertige Byte erst frei, nachdem die Berechnung abgeschlossen ist. Es fungiert zudem als Puffer, falls mehrere Eingangsereignisse kurz hintereinander eintreffen.
- **Effiziente Ereignissteuerung** – Jedes Ereignis an einem der acht AX-Sockets löst eine Neuberechnung aus. Überflüssige Aktualisierungen werden vermieden, da die Ausgabe nur nach einem Takt erfolgt.

Zustandsübersicht

Der Funktionsblock besitzt keine eigene Zustandsmaschine; er ist als reines Netzwerk aus zwei Unterbausteinen aufgebaut. Sein Verhalten wird vollständig durch die interne Logik von **ASSEMBLE_BYTE_FROM_BOOLS** und **E_D_FF_ANY** bestimmt.

Anwendungsszenarien

- **Zusammenfassen von 8 digitalen Sensoren** – z. B. Endschalter, Lichtschranken oder Binäreingänge einer SPS, deren Zustände als Byte übertragen werden sollen.
- **Bit-parallele Datenübertragung** – Umwandlung eines 8-Bit-parallelen Signals in ein serielles Byte für einen weiteren Baustein (z. B. über Adapter-Kopplung).
- **Modulare Automatisierungsfunktionen** – Einbindung in Hierarchien, bei denen mehrere `ASSEMBLE_AB_FROM_AX`-Blöcke verwendet werden, um größere Datenwörter (z. B. WORD, DWORD) zusammenzusetzen.

Vergleich mit ähnlichen Bausteinen

Baustein	Beschreibung	Unterschied
`ASSEMBLE_BYTE_FROM_BOOLS`	Interner Baustein, der aus 8 Bool-Eingängen ein Byte erzeugt (ohne Adapter)	`ASSEMBLE_AB_FROM_AX` kapselt diesen Baustein und verwendet Adapter für die Anbindung.
`eclipse4diac::utils::assembling::ASSEMBLE_BYTE_FROM_BOOLS`	Gleiche Funktion, aber mit direkten Ereignis-/Datenports	`ASSEMBLE_AB_FROM_AX` bietet eine adapterbasierte Schnittstelle und fügt ein D-Flipflop zur Synchronisation hinzu.
Selbst erstellter Byte-Assembler	Beliebig realisierbar, z. B. mit ST-Algorithmus	Adapter `AX`/`AB` sind vordefinierte Standard-Typen in 4diac, die Wiederverwendbarkeit und Austauschbarkeit fördern.

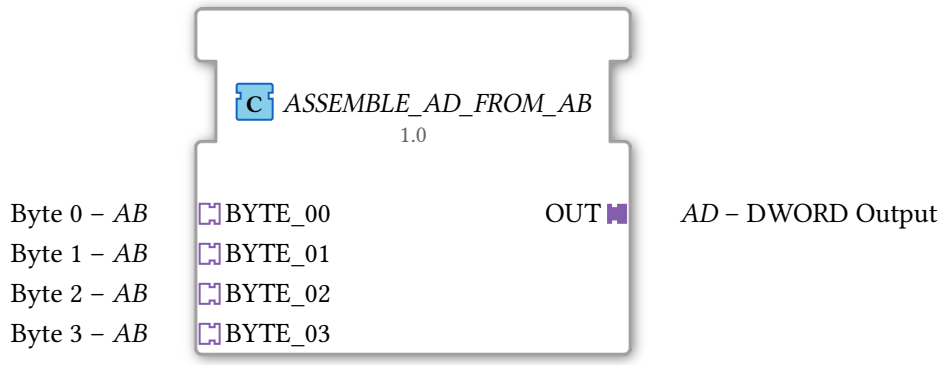
Fazit

Der Funktionsblock **ASSEMBLE_AB_FROM_AX** ist ein praktischer, adapterbasierter Helfer, um acht boolesche Signale in ein Byte zu konvertieren. Durch die Integration eines D-Flipflops arbeitet er zuverlässig und vermeidet inkonsistente Zwischenzustände. Sein modulares Design erleichtert die Wiederverwendung in größeren Projektstrukturen und macht ihn zu einer nützlichen Komponente in der Automatisierungstechnik mit 4diac.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ASSEMBLE_AD_FROM_AB



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASSEMBLE_AD_FROM_AB** dient dazu, vier einzelne Byte-Werte, die über unidirektionale **AB**-Adapter (Byte) hereinkommen, zu einem 32-Bit **DWORD**-Wert zusammenzufügen und über einen unidirektionalen **AD**-Adapter auszugeben. Die Kombination der Bytes erfolgt in der Reihenfolge BYTE_00 (niederwertigstes Byte) bis BYTE_03 (höchstwertiges Byte). Der Baustein verwendet

dazu einen internen Zusammenfüger und einen flankengesteuerten Flip-Flop, um die Ausgabe erst nach vollständigem Abschluss der Zusammenführung zu aktualisieren.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine direkten Ereignis-Eingänge. Ereignisse werden über die angeschlossenen **Socket-Adapter** (BYTE_00 … BYTE_03) empfangen.

Ereignis-Ausgänge

Es gibt keinen direkten Ereignis-Ausgang. Das Ausgangsereignis wird über den **Plug-Adapter** (OUT) bereitgestellt.

Daten-Eingänge

Es gibt keine direkten Daten-Eingänge. Alle Daten werden über die **Socket-Adapter** (BYTE_00 … BYTE_03) entgegengenommen.

Daten-Ausgänge

Es gibt keinen direkten Daten-Ausgang. Der zusammengeführte DWORD-Wert wird über den **Plug-Adapter** (OUT) ausgegeben.

Adapter

Name	Typ	Richtung	Beschreibung
BYTE_00	`adapter::types::unidirectional::AB`	Socket	Byte 0 (niederwertigstes Byte)
BYTE_01	`adapter::types::unidirectional::AB`	Socket	Byte 1
BYTE_02	`adapter::types::unidirectional::AB`	Socket	Byte 2
BYTE_03	`adapter::types::unidirectional::AB`	Socket	Byte 3 (höchstwertiges Byte)
OUT	`adapter::types::unidirectional::AD`	Plug	32-Bit DWORD-Ausgang

Funktionsweise

Der Baustein arbeitet vollständig im Inneren eines Subapplikationsnetzwerks:

1. Ein Ereignis an einem der vier Socket-Adapter (BYTE_00 … BYTE_03) löst über die jeweilige `E1`-Leitung den **REQ**-Eingang des internen `ASSEMBLE_DWORD_FROM_BYTES`-Bausteins aus.
2. Dieser Baustein liest die aktuellen Werte an seinen Daten-Eingängen (`BYTE_00` … `BYTE_03`) ein, setzt sie zu einem 32-Bit-Wert zusammen und signalisiert dies mit **CNF**.
3. Das **CNF**-Ereignis triggert den **CLK**-Eingang des internen `E_D_FF_ANY`-Flip-Flops. Das Flip-Flop übernimmt den am **D**-Eingang anliegenden Datenwert (das zusammengefügte DWORD).
4. Der Ausgang **Q** des Flip-Flops wird permanent auf den **D1**-Datenausgang des Plug-Adapters **OUT** gelegt.
5. Gleichzeitig wird mit dem Flip-Flop-Ausgangssignal das Ereignis **E1** des Plug-Adapters ausgelöst, so dass die empfangende Seite über den neuen Wert informiert wird.

Auf diese Weise wird sichergestellt, dass der Ausgang nur dann aktualisiert wird, wenn ein Ereignis an mindestens einem Socket-Adapter eingetroffen ist und die Zusammenführung abgeschlossen wurde. Der letzte zusammengeführte Wert bleibt solange erhalten, bis ein neuer Vorgang abgeschlossen ist.

Technische Besonderheiten

- **Ereignissynchronisation**: Alle vier Socket-Ereignisse werden auf einen gemeinsamen `REQ` des Zusammenführungsbausteins gelegt. Dadurch wird die Verarbeitung bei jedem eingehenden Byte-Ereignis neu angestoßen – es ist nicht notwendig, dass alle vier Bytes gleichzeitig aktualisiert werden.
- **Zwischenspeicherung**: Der eingebaute `E_D_FF_ANY` verhindert eine Ausgabe während der Zusammenführungsphase. Erst wenn das Ergebnis stabil vorliegt, wird es an den Plug weitergegeben.
- **Adapter-basiert**: Der Baustein besitzt keine direkten Ein-/Ausgänge, sondern kommuniziert ausschließlich über standardisierte unidirektionale Adapter (`AB` und `AD`). Dies erlaubt eine flexible Einbindung in existierende Adapter-Schnittstellen.

Zustandsübersicht

Der Baustein selbst besitzt keinen expliziten Zustandsautomaten. Das Verhalten wird durch die internen Bausteine bestimmt:

- **Warten auf Ereignis**: Es liegt kein neues Ereignis an den Sockets an. Der Ausgangswert ist statisch.
- **Summieren & Aktualisieren**: Ein Ereignis trifft ein → Zusammenführung wird ausgelöst → Flip-Flop übernimmt neuen Wert → Ausgang wird aktualisiert.
- **Warten auf nächstes Ereignis**: Nach erfolgter Aktualisierung bleibt der Zustand bis zum nächsten Socket-Ereignis erhalten.

Anwendungsszenarien

- **Datenstromvereinigung**: Zusammenführen von vier Byte-orientierten Sensor- oder Kommunikationsdaten zu einem 32-Bit-Wert (z. B. für eine Feldbus-Schnittstelle).
- **Protokoll-Wrapper**: Kombination von einzelnen Bytes aus unterschiedlichen Quellen zu einem vollständigen Datenwort, das anschließend über einen AD-Adapter an eine übergeordnete Logik weitergegeben wird.
- **Test- und Simulationsumgebungen**: Einsatz als Konverter zwischen bestehenden Byte-Adapter-Komponenten und DWORD-Adapter-Komponenten.

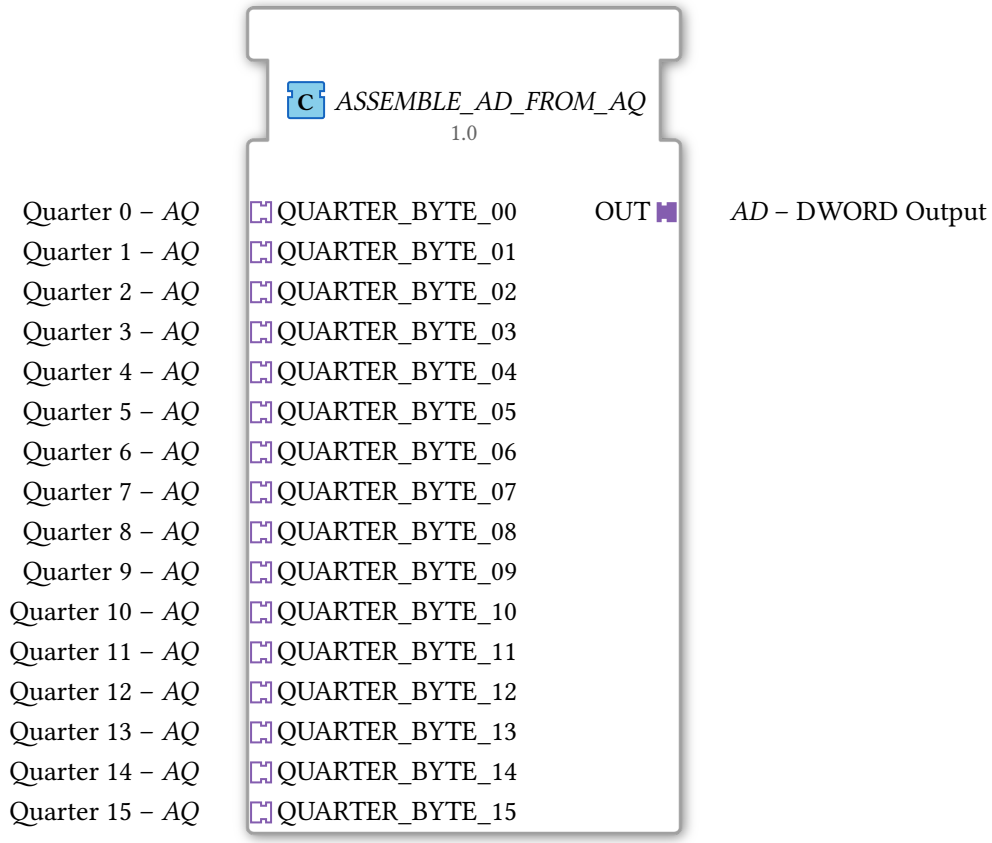
Vergleich mit ähnlichen Bausteinen

- **Direkter Datentyp-Konverter** (z. B. `BYTE_TO_DWORD`): Diese arbeiten oft mit einfachen Daten-Ein-/Ausgängen ohne Ereignisprotokoll. Der `ASSEMBLE_AD_FROM_AB` hingegen integriert die Ereignissteuerung und ist auf die Adapter-Schnittstelle zugeschnitten.
- **Manuelle Verkettung von Adaptern**: Statt vier separaten `AB`-Adaptern und einem manuell aufgebauten Netzwerk bietet dieser Baustein eine vorgefertigte und getestete Lösung, die weniger Verdrahtungsaufwand erfordert.

Fazit

Der **ASSEMBLE_AD_FROM_AB** ist ein spezialisierter, adapter-basierter Funktionsblock zur Zusammenführung von vier Byte-Werten zu einem DWORD. Durch die interne Verwendung eines Zusammenfügers und eines Flip-Flops wird eine zuverlässige, ereignisgesteuerte Aktualisierung des Ausgangswerts erreicht. Er eignet sich ideal für Systeme, die auf standardisierten unidirektionalen Adaptern aufbauen und eine saubere, flackerschutzfreie Datenweitergabe benötigen.

ASSEMBLE_AD_FROM_AQ



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ASSEMBLE_AD_FROM_AQ` dient dazu, sechzehn separate `AQ`-Adapter (Quarter) zu einem einzigen `AD`-Ausgangsadapter (DWORD) zusammenzusetzen. Die Bezeichnung „Quarter“ deutet darauf hin, dass jeder `AQ`-Adapter einen Teil eines 32-Bit-Wortes repräsentiert – konkret ein Byte (8 Bit). Der Baustein fasst diese 16 Bytes zu einem vollständigen DWORD zusammen und stellt es über einen `AD`-Plug bereit. Er eignet sich vor allem für Anwendungen, bei denen Daten in kleineren Einheiten eintreffen und erst auf der Empfängerseite zu einem größeren Datentyp vereinigt werden müssen.

Schnittstellenstruktur

Der Funktionsblock besitzt keine eigenständigen Ereignis- oder Dateneingänge/-ausgänge. Die gesamte Ein- und Ausgabe erfolgt ausschließlich über Adapter.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung

Sockets (Eingänge)	QUARTER_BYTE_00 … QUARTER_BYTE_15	`adapter::types::unidirectional_AD`	16 identische Adapter, die jeweils einen 8-Bit-Datenwert (Quarter/Byte) sowie ein zugehöriges Ereignis (`E1`) liefern.
Plug (Ausgang)	OUT	`adapter::types::unidirectional_AD`	Ausgangsadapter, der das aus den 16 Quarter-Werten zusammengesetzte DWORD (`D1`) und ein Ereignis (`E1`) bereitstellt.

Funktionsweise

Der Baustein arbeitet vollständig ereignisgesteuert:

1. Sobald einer der 16 `QUARTER_BYTE`-Sockets ein Ereignis (`E1`) empfängt, wird dieses an den internen Funktionsblock `ASSEMBLE_DWORD_FROM_QUARTERS` (Typ `eclipse4diac::utils::assembling::ASSEMBLE_DWORD_FROM_QUARTERS`) weitergeleitet.
2. Der interne Baustein kombiniert die aktuellen Daten aller 16 Quarter-Werte zu einem 32-Bit-DWORD (bit-0…7 = Quarter 0, bit-8…15 = Quarter 1, … bit-120…127 = Quarter 15).
3. Das Ergebnis wird einem flankengesteuerten D-Flip-Flop (`E_D_FF_ANY`) übergeben.
4. Der Flip-Flop gibt bei der nächsten steigenden Flanke seines Takteingangs (`CLK`) das zusammengesetzte DWORD an den `OUT`-Adapter aus und erzeugt ein Ereignis (`E0`), das wiederum das Ausgangsereignis `E1` des `OUT`-Adapters auslöst.

Durch die Zwischenschaltung des Flip-Flops wird sichergestellt, dass der Ausgang nur dann aktualisiert wird, wenn die Daten vollständig zusammengesetzt und stabil sind – selbst wenn mehrere Quarter-Ereignisse quasi gleichzeitig eintreffen.

Technische Besonderheiten

- **Keine eigenen Ein-/Ausgänge:** Der Baustein kommuniziert ausschließlich über Adapter, was eine saubere Kapselung und Wiederverwendbarkeit in unterschiedlichen Projektkontexten ermöglicht.
- **Synchrone Aktualisierung:** Der D-Flip-Flop verhindert Dateninkonsistenzen, falls mehrere Quarter ihre Werte nahezu zeitgleich ändern. Der Ausgang ändert sich erst, nachdem alle Eingänge ausgewertet wurden.
- **Autonome Triggerung:** Jedes Quarter-Ereignis (egal von welchem Socket) löst eine Neuberechnung aus. Der Baustein arbeitet immer mit den aktuell anliegenden Werten aller 16 Quarter-Daten.
- **Interne Indirektion:** Die tatsächliche Assemblierung erfolgt durch einen spezialisierten Unterbaustein (`ASSEMBLE_DWORD_FROM_QUARTERS`), was den Entwurf modular und wartbar hält.

Zustandsübersicht

Der Funktionsblock selbst besitzt keinen eigenen Zustandsautomaten. Er verhält sich wie eine kombinatorische Logik mit nachgeschalteter Synchronisation. Der innere Flip-Flop `E_D_FF_ANY` verfügt jedoch über einen internen Zustand (den gespeicherten DWORD-Wert). Dieser Zustand wird nur beim Eintreffen eines Ereignisses an `CLK` geändert.

Im Ruhezustand (kein Ereignis an einem Quarter-Socket) bleibt der Ausgangswert des `OUT`-Adapters unverändert.

Anwendungsszenarien

- **Verteiltes Messsystem:** Mehrere Sensoren liefern je ein Byte; diese werden in einer Zentraleinheit zum Gesamt-DWORD kombiniert.
- **Datenpaket-Rekonstruktion:** Zerlegte Kommunikationsframes (z. B. 16-Byte-Nutzdaten) werden wieder zusammengesetzt.
- **Mediatorenstrukturen:** Als Bestandteil einer Adapter-Kette zur Umwandlung von Datenbreiten in IEC 61499-Anwendungen.
- **Synthese von Steuerwörtern:** Mehrere 8-Bit-Steuerkanäle werden zu einem 32-Bit-Befehlswort vereinigt.

Vergleich mit ähnlichen Bausteinen

Baustein	Anzahl Eingänge	Ausgang	Synchronisation	Eigenschaften
`ASSEMBLE_AD_FROM_AQ`	16 (AQ-Adapter)	1 (AD-Adapter)	Ja (Flip-Flop)	Für ereignisgesteuerte Byte-Sammlung.
`ASSEMBLE_DWORD_FROM_QUARTERS` (intern)	16 Daten-Eingänge (keine Adapter)	1 DWORD-Ausgang	Nein	Reine Daten-Assemblierung ohne Ereignissynchronisation.
Einfacher `MERGE`-Baustein (fiktiv)	2 … n Daten-Eingänge	1 Daten-Ausgang	Oft nein	Nur Daten-Verknüpfung, keine Adapter-Struktur.

Der vorliegende Baustein zeichnet sich durch die Kapselung der Adapter-Schnittstellen und die integrierte Synchronisation aus, wodurch er besonders für heterogene, ereignisgetriebene Umgebungen geeignet ist.

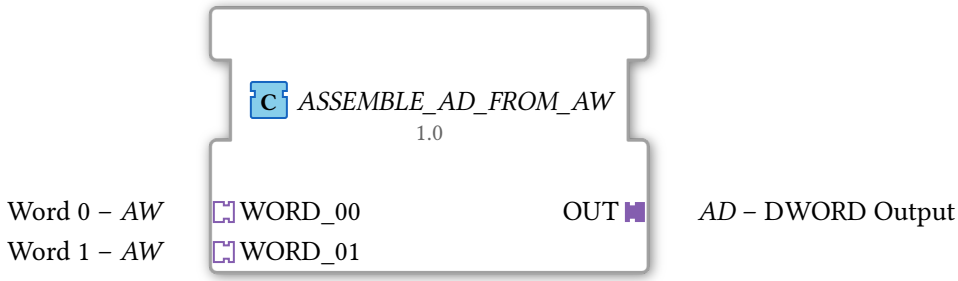
Fazit

`ASSEMBLE_AD_FROM_AQ` ist ein spezialisierter Adapter-Baustein zur Zusammenführung von 16 Byte-breiten Quarter-Daten in ein DWORD. Seine rein adapterbasierte Schnittstelle macht ihn flexibel einsetzbar, während die interne Synchronisation durch einen Flip-Flop die Datenkonsistenz bei asynchronen Ereignissen gewährleistet. Er eignet sich hervorragend für den Einsatz in modularen IEC 61499-Anwendungen, bei denen Daten in kleineren Einheiten ankommen und erst auf Empfängerseite zu einem vollständigen Wort zusammengesetzt werden müssen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ASSEMBLE_AD_FROM_AW



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASSEMBLE_AD_FROM_AW** dient der Kombination zweier Wort-Werte (WORD) aus je einem unidirektionalen AW-Adapter zu einem Doppelwort (DWORD), das über einen unidirektionalen AD-Adapter ausgegeben wird. Der Baustein kapselt die logische Verknüpfung von zwei 16-Bit-Eingängen zu einem 32-Bit-Ausgang und speichert das Ergebnis über ein flankengesteuertes D-Flipflop zwischen.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine direkt sichtbaren Ereignis-Eingänge. Die Ereignissteuerung erfolgt indirekt über die eingehenden Adapter **WORD_00** und **WORD_01**: Jeder dieser Adapter liefert über seinen impliziten Ereignisausgang (`E1`) ein Signal, das den internen Verarbeitungsablauf anstößt.

Ereignis-Ausgänge

Auch hier existieren keine direkten Ereignis-Ausgänge. Der Ausgangsadapter **OUT** stellt über seinen impliziten Ereigniseingang (`E1`) das Ausgangsereignis zur Verfügung, das nach Abschluss der Verarbeitung und Übernahme des Datenwertes ausgelöst wird.

Daten-Eingänge

Die Daten werden über die beiden Socket-Adapter eingelesen:

Adapter	Typ	Beschreibung
`WORD_00`	`adapter::types::unidirectional::AW`	Erstes 16-Bit-Wort (niederwertiger Anteil des Doppelworts)
`WORD_01`	`adapter::types::unidirectional::AW`	Zweites 16-Bit-Wort (höherwertiger Anteil des Doppelworts)

Jeder dieser Adapter stellt einen Datenausgang (`D1`) mit dem eigentlichen WORD-Wert bereit.

Daten-Ausgänge

Der Ausgang erfolgt über einen Plug-Adapter:

Adapter	Typ	Beschreibung
`OUT`	`adapter::types::unidirectional::AD`	Zusammengesetztes 32-Bit-Doppelwort (DWORD)

Der Adapter `OUT` besitzt einen Dateneingang (`D1`), der intern mit dem gespeicherten Ergebnis verbunden ist.

Adapter

Der FB stellt zwei eingehende Adapter (Socket) und einen ausgehenden Adapter (Plug) zur Verfügung, die alle vom Typ *unidirectional* sind:

- **WORD_00**, **WORD_01**: Liefern jeweils ein WORD und ein zugehöriges Ereignis.
- **OUT**: Nimmt ein DWORD entgegen und gibt es mit einem Ereignis weiter.

Funktionsweise

1. Sobald eines der beiden Eingangsereignisse (`WORD_00.E1` oder `WORD_01.E1`) eintrifft, wird der interne Funktionsblock `ASSEMBLE_DWORD_FROM_WORDS` über seinen Ereigniseingang `REQ` aktiviert.
2. Dieser interne Baustein setzt aus den beiden eingehenden 16-Bit-Werten (`WORD_00.D1` und `WORD_01.D1`) das 32-Bit-Doppelwort zusammen. Dabei wird `WORD_00` als niederwertiges und `WORD_01` als höherwertiges Wort interpretiert.
3. Nach Abschluss der Berechnung signalisiert `ASSEMBLE_DWORD_FROM_WORDS` mit dem Ereignis `CNF` den Takt eines flankengesteuerten D-Flipflops (`E_D_FF_ANY`).
4. Das Flipflop übernimmt den berechneten DWORD-Wert an seinem Dateneingang `D` und gibt ihn an seinem Ausgang `Q` weiter.
5. Der gespeicherte Wert wird über die Datenverbindung an den Adapter `OUT.D1` geleitet. Gleichzeitig wird über das Ausgangsereignis des Flipflops (`EO`) der Adapter-Ereigniseingang `OUT.E1` bedient.

Auf diese Weise wird sichergestellt, dass der Ausgangswert nur bei tatsächlichen Änderungen oder gültigen neuen Eingängen aktualisiert wird und bis zum nächsten Ereignis stabil bleibt.

Technische Besonderheiten

- Der Baustein verwendet einen internen Funktionsblock (`ASSEMBLE_DWORD_FROM_WORDS`) zur reinen Wort-Kombination und ein flankengesteuertes D-Flipflop (`E_D_FF_ANY`) zur Ausgangsspeicherung.
- Die Ereignisverarbeitung erfolgt asynchron: Jedes eingehende Ereignis an einem der beiden AW-Adapter löst eine neue Berechnung und eine anschließende Ausgabe aus. Es wird kein Zustand der Eingangsereignisse gespeichert – das Flipflop übernimmt immer den zuletzt berechneten DWORD.
- Die Zusammensetzung der Wörter erfolgt hardwarenah: WORD_00 = niederwertiges 16-Bit-Wort (Bits 0-15), WORD_01 = höherwertiges 16-Bit-Wort (Bits 16-31).

Zustandsübersicht

Der FB selbst besitzt keinen expliziten Zustandsautomaten. Der interne Ablauf lässt sich aber durch die Zustände des D-Flipflops charakterisieren:

Zustand	Beschreibung
Warten auf Ereignis	Das Flipflop hält den zuletzt berechneten Wert; es liegt kein neues Eingangsereignis an.
Berechnung aktiv	Ein Ereignis von WORD_00 oder WORD_01 triggert die Zusammenführung und das Flipflop wird getaktet.
Ausgabe aktiv	Nach dem Takt wird der neue Wert an `OUT` weitergegeben und das Ausgangsereignis wird gesendet.

Der Wechsel der Zustände erfolgt streng durch die Ereigniskette.

Anwendungsszenarien

- **Datenformat-Wandlung**: Zusammenführen zweier 16-Bit-Sensorwerte (z. B. aus zwei getrennten Analog-Eingabemodulen) zu einem 32-Bit-Datenwort für eine übergeordnete Steuerung.
- **Adapter-Integration**: Einbinden von AW-basierten Komponenten (z. B. Wort-Adapter von Bussystemen) in Systeme, die einen AD-Adapter (DWORD) erwarten.
- **Schnittstellenanpassung**: Vereinfachung der Verdrahtung, wenn zwei logisch zusammengehörige 16-Bit-Kanäle zu einem Doppelwort zusammengefasst werden müssen.

Vergleich mit ähnlichen Bausteinen

- **ASSEMBLE_DWORD_FROM_WORDS** (interner FB): Bietet die gleiche Datenzusammenführung, jedoch ohne Adapter-Schnittstellen und ohne Ausgangsspeicherung. `ASSEMBLE_AD_FROM_AW` erweitert diesen um die Adapter-Kapselung und das D-Flipflop.
- **SPLIT_AD_TO_AW**: Führt die umgekehrte Operation (Aufteilen eines DWORD in zwei Wörter) aus und stellt die Werte über AW-Adapter bereit.
- **MUX_WORDS_TO_DWORD**: Ein alternativer Baustein zur Wort-Kombination, der meist ohne Adapter und ohne flankengesteuerte Speicherung auskommt.

Fazit

`ASSEMBLE_AD_FROM_AW` ist ein spezialisierter Funktionsblock zur sauberen Kapselung der Wort-zu-Doppelwort-Konvertierung in einer Adapter-basierten Umgebung. Die Kombination aus reinem Rechenwerk und flankengesteuerter Speicherung gewährleistet stabile Ausgangswerte bei wechselnden Eingangsereignissen. Der Baustein eignet sich besonders für modulare Automatisierungslösungen, die auf unidirektionale Adapter-Schnittstellen setzen.

ASSEMBLE_AD_FROM_AX

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASSEMBLE_AD_FROM_AX** dient dazu, bis zu 32 boolesche Einzelsignale, die über AX-Adapter (Typ: `adapter::types::unidirectional::AX`) bereitgestellt werden, zu einem 32-Bit-Doppelwort (DWORD) zusammenzufassen und über einen AD-Adapter (Typ: `adapter::types::unidirectional::AD`) auszugeben. Dies ermöglicht eine kompakte Übertragung mehrerer diskreter Binärsignale über eine einzige Datenverbindung.

Schnittstellenstruktur

Der FB besitzt ausschließlich **Adapter-Schnittstellen** (Sockets und Plugs). Direkte Ereignis- oder Daten-Ein-/Ausgänge auf der obersten Ebene sind nicht vorhanden.

Ereignis-Eingänge

Keine expliziten Ereignis-Eingänge. Die Ereignisse werden implizit über die AX-Adapter empfangen:

- Jeder der 32 AX-Adapter (`BIT_00` … `BIT_31`) stellt einen Ereignisausgang (`E1`) bereit, der bei einer Wertänderung des zugehörigen BOOL-Signals aktiviert wird.

Ereignis-Ausgänge

Keine expliziten Ereignis-Ausgänge. Der AD-Ausgangsadapter (`OUT`) löst ein Ereignis (`E1`) aus, sobald das zusammengesetzte DWORD einen neuen Wert annimmt.

Daten-Eingänge

Keine expliziten Daten-Eingänge. Die booleschen Eingangswerte werden über die Datenports (`D1`) der AX-Adapter bezogen:

- `BIT_00.D1` … `BIT_31.D1`: jeweils ein **BOOL** (Bit 0 … Bit 31 des resultierenden DWORDs).

Daten-Ausgänge

Keine expliziten Daten-Ausgänge. Das zusammengesetzte DWORD wird über den Datenport (`D1`) des AD-Adapter ausgegeben:

- `OUT.D1`: **DWORD** (das aus den 32 BOOL-Eingängen zusammengesetzte 32-Bit-Wort).

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AX`	`BIT_00` … `BIT_31`	Socket (Eingang)	32 boolesche Einzelsignale, jeweils mit eigenem Ereignis (Datenänderung)
`adapter::types::unidirectional::AD`	`OUT`	Plug (Ausgang)	Ausgabe des zusammengesetzten Doppelworts mit Aktualisierungsereignis

Funktionsweise

1. **Ereignisempfang**: Sobald einer der 32 AX-Adapter eine Änderung seines booleschen Werts meldet (Ereignis `E1`), wird das Ereignis an den internen Baustein `ASSEMBLE_DWORD_FROM_BOOLS.REQ` weitergeleitet.
2. **Datenzusammenstellung**: Der interne Baustein `ASSEMBLE_DWORD_FROM_BOOLS` kombiniert alle 32 BOOL-Werte (von `BIT_00.D1` bis `BIT_31.D1`) zu einem einzigen DWORD. Bit 0 entspricht `BIT_00`, Bit 1 `BIT_01` usw.
3. **Speicherung und Ausgabe**: Das resultierende DWORD wird an den nächsten internen Baustein `E_D_FF_ANY` übergeben. Dieses flankengesteuerte Flipflop speichert den Wert und gibt ihn an seinem Ausgang `Q` aus. Gleichzeitig erzeugt es ein Ereignis (`EO`), das über den AD-Adapter (`OUT.E1`) nach außen signalisiert wird, dass ein neuer DWORD-Wert vorliegt.
4. **Pufferung**: Durch das Flipflop wird sichergestellt, dass der Ausgangswert nur bei tatsächlichen Änderungen aktualisiert wird und nicht bei jedem einzelnen Bit-Ereignis mehrmals hintereinander den gleichen Wert ausgibt.

Technische Besonderheiten

- **Adapter-basierte Schnittstelle**: Der FB nutzt ausschließlich Adapter (AX, AD) und keine direkten Ein-/Ausgangspins. Dies erlaubt eine flexible Kopplung mit anderen Adaptern in einer serviceorientierten Architektur.
- **Ereignis-Synchronisation**: Der integrierte `E_D_FF_ANY` (E-D-Flipflop mit beliebiger Flanke) fungiert als Entprell- und Synchronisationsstufe. Er verhindert, dass bei gleichzeitiger Änderung mehrerer Bits mehrere Ausgabeereignisse generiert werden – der Ausgang wird nur einmal pro Änderungszyklus aktualisiert.

- **Bit-Reihenfolge**: Bit 0 (LSB) entspricht Adapter `BIT_00`, Bit 31 (MSB) entspricht Adapter `BIT_31`. Eine einheitliche Zuordnung ist bei der Verschaltung zu beachten.
- **Typ-Hash**: Der FB enthält ein Attribut `eclipse4diac::core::TypeHash`, das zur Validierung der Bausteindefinition dient.

Zustandsübersicht

Der FB selbst besitzt keinen expliziten Zustandsautomaten. Das interne `E_D_FF_ANY` realisiert einen einfachen Speicherzustand:

- **Zustand 0**: Flipflop-Ausgang `Q` enthält den zuletzt geladenen DWORD-Wert.
- **Zustandsübergang**: Bei einem Ereignis von `ASSEMBLE_DWORD_FROM_BOOLS.CNF` wird der neue DWORD-Wert in das Flipflop übernommen und der Ausgang aktualisiert.

Anwendungsszenarien

- **Zusammenfassen digitaler Eingänge**: In einer Steuerung werden 32 diskrete Sensoren (z. B. Endschalter, Lichtschranken) über AX-Adapter eingelesen. Der FB fasst deren Zustände in einem DWORD zusammen, das über einen Feldbus oder eine andere Schnittstelle als kompaktes Datenwort übertragen werden kann.
- **Parallel-Seriell-Wandlung**: Vorbereitung von parallelen Binärdaten für eine serielle Übertragung, bei der das DWORD als einzelnes Telegramm gesendet wird.
- **Statusabfrage**: Ein zentraler FB fragt regelmäßig den Ausgangsadapter ab und erhält den aktuellen Gesamtstatus aller 32 Binärstellen auf einmal.

Vergleich mit ähnlichen Bausteinen

- **ASSEMBLE_DWORD_FROM_BOOLS**: Dieser Baustein hat direkte BOOL-Eingänge und einen DWORD-Ausgang, aber keine Adapter-Schnittstelle. `ASSEMBLE_AD_FROM_AX` kapselt diese Logik in eine Adapter-basierte Komponente und fügt eine Flipflop-Speicherung hinzu.
- **AD_TO_AX_SPLITTER**: Der umgekehrte Vorgang – Zerlegung eines DWORDs in einzelne BOOL-Signale – wird durch einen entsprechenden Splitter-Baustein ermöglicht.
- **Direkte Bit-Manipulation**: In manchen Umgebungen könnte man die Bits auch über logische Verknüpfungen zusammensetzen. Der hier beschriebene FB bietet jedoch eine standardisierte, wiederverwendbare und ereignisgesteuerte Lösung.

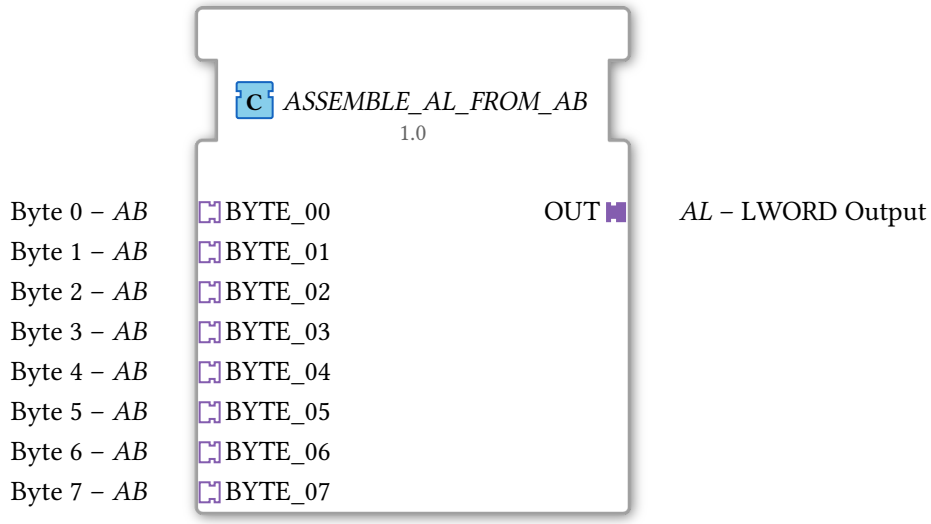
Fazit

Der `ASSEMBLE_AD_FROM_AX` ist ein nützlicher Baustein, um eine Vielzahl binärer Signale effizient in ein Datenwort zu bündeln. Die Verwendung von Adaptern erlaubt eine lose Kopplung und erleichtert die Integration in serviceorientierte Automatisierungsarchitekturen. Die integrierte Flipflop-Synchronisation vermeidet unnötige Ausgabeereignisse und sorgt für einen stabilen, aktualisierten Gesamtwert. Durch seine klare Struktur eignet er sich besonders für Anwendungen, in denen viele diskrete Signale zentral erfasst und weiterverarbeitet werden müssen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ASSEMBLE_AL_FROM_AB



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ASSEMBLE_AL_FROM_AB` fasst acht einzelne Byte-Informationen aus Adaptern des Typs `AB` (unidirektional) zu einem zusammenhängenden 64-Bit-Datenwort (LWORD) zusammen. Das Ergebnis wird über einen Ausgangsadapter des Typs `AL` ausgegeben. Der Baustein dient als zentrale Sammelstelle für Byte-orientierte Daten und ermöglicht eine modulare und übersichtliche Datenstrukturierung.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignissteuerung erfolgt ausschließlich über die angeschlossenen Adapter.

Ereignis-Ausgänge

Keine. Das Ausgangsereignis wird über den Adapter `OUT` bereitgestellt.

Daten-Eingänge

Keine. Die Dateneingänge werden über die Byte-Adapter eingespeist.

Daten-Ausgänge

Keine. Das zusammengesetzte LWORD wird über den Adapter `OUT` ausgegeben.

Adapter

Sockets (Eingangsadapter)

Name	Datentyp	Beschreibung
`BYTE_00`	`adapter::types::unidirectional::AB`	Byte 0 (niederwertigstes Byte)
`BYTE_01`	`adapter::types::unidirectional::AB`	Byte 1
`BYTE_02`	`adapter::types::unidirectional::AB`	Byte 2
`BYTE_03`	`adapter::types::unidirectional::AB`	Byte 3
`BYTE_04`	`adapter::types::unidirectional::AB`	Byte 4
`BYTE_05`	`adapter::types::unidirectional::AB`	Byte 5
`BYTE_06`	`adapter::types::unidirectional::AB`	Byte 6
`BYTE_07`	`adapter::types::unidirectional::AD`	Byte 7 (höchstwertiges Byte)

Jeder Byte-Adapter stellt ein Ereignis (`E1`) und ein Datenbyte (`D1`) bereit.

Plugs (Ausgangsadapter)

Name	Datentyp	Beschreibung
OUT	adapter::types::unidirectional::AL	Ausgang: zusammengesetztes LWORD; Ereignisausgang über E1, Datenwert über D1

Funktionsweise

Der Baustein ist als Composite-FB realisiert und enthält zwei interne Funktionsblöcke: ASSEMBLE_LWORD_FROM_BYTES und E_D_FF_ANY.

1. **Datensammlung**: Sobald einer der acht Byte-Adapter (BYTE_00 … BYTE_07) ein Ereignis an seinem Ausgang E1 liefert, wird dieses Ereignis an den Eingang REQ von ASSEMBLE_LWORD_FROM_BYTES weitergeleitet. Gleichzeitig werden die zugehörigen Byte-Werte von jedem Adapter an die entsprechenden Dateneingänge des Assembler-Blocks geführt (BYTE_00 … BYTE_07).
2. **Zusammenfügung**: Der interne Baustein ASSEMBLE_LWORD_FROM_BYTES setzt die acht Bytes in der Reihenfolge BYTE_00 bis BYTE_07 zu einem 64-Bit-Wort (LWORD) zusammen. BYTE_00 ist das niederwertigste Byte, BYTE_07 das höchstwertige. Nach Abschluss der Operation wird ein Bestätigungsereignis (CNF) ausgegeben und das zusammengesetzte LWORD an seinem Datenausgang bereitgestellt.
3. **Pufferung und Ausgabe**: Das Ereignis CNF triggert den flankengesteuerten Flip-Flop E_D_FF_ANY. Dieser übernimmt den aktuellen LWORD-Wert und speichert ihn zwischen. Bei jeder positiven Flanke am Takteingang (CLK) wird der Wert am Datenausgang Q aktualisiert. Der Ausgang Q ist mit dem Datenausgang D1 des Adapter-Plugs OUT verbunden. Das gespeicherte Ereignis EO des Flip-Flops wird als Ausgangsereignis E1 des Adapter-Plugs OUT bereitgestellt.

Somit wird bei jeder Aktualisierung eines der acht Byte-Eingänge der gesamte LWORD neu berechnet und über den Ausgangsadapter ausgegeben.

Technische Besonderheiten

- **Adapter-basierte Schnittstellen**: Der Baustein verwendet ausschließlich Adapter für die Ein- und Ausgabe. Dies ermöglicht eine flexible und lose Kopplung mit anderen Komponenten, die denselben Adaptertyp nutzen.
- **Komposit-Architektur**: Die interne Struktur ist aus bewährten Standardbausteinen aufgebaut (ASSEMBLE_LWORD_FROM_BYTES und E_D_FF_ANY). Dadurch wird die Komplexität verborgen und die Wartbarkeit erhöht.
- **Pufferung durch Flip-Flop**: Der E_D_FF_ANY puffert den assemblierten Wert und gibt ihn erst nach Abschluss der Assemblierung weiter. Dies vermeidet inkonsistente Ausgaben während der Aktualisierung einzelner Bytes und stellt sicher, dass immer ein vollständiger Datenwert anliegt.
- **Ereignissynchronisation**: Alle Byte-Adaptersignale werden auf einen gemeinsamen REQ-Eingang geführt. Dadurch wird jede Änderung eines beliebigen Bytes sofort verarbeitet. Die Zusammenführung aller Bytes erfolgt jedoch stets auf Basis der aktuell anliegenden Werte.

Zustandsübersicht

Da es sich um einen Composite-FB handelt, besitzt er keinen eigenen Zustandsautomaten. Der innerliegende E_D_FF_ANY besitzt einen internen Zustand: den gespeicherten LWORD-Wert. Dieser ändert sich nur bei positiven Flanken am Takteingang, also nach jeder erfolgreichen Assemblierung. Der Zustand bleibt erhalten, bis eine erneute Assemblierung ausgelöst wird.

Anwendungsszenarien

- **Datenaggregation in Sensornetzwerken**: Mehrere Byte-orientierte Sensoren (z. B. Temperatur, Druck, Feuchte) werden über separate Adapter angebunden. Der Baustein setzt sie zu einem 64-Bit-Datentelegramm zusammen, das an eine zentrale Steuerung weitergeleitet wird.
- **Protokollkonvertierung**: Ein Byte-Stream, der über mehrere Adapter eingeht (z. B. serielle Schnittstelle mit 8 Nutzbytes), wird in ein einzelnes LWORD umgewandelt, das für weitere Verarbeitungsschritte einfacher zu handhaben ist.
- **Modulare E/A-Erweiterungen**: In einer modularen Automatisierungslösung können acht digitale/analoge Byte-Eingänge (z. B. über I/O-Module) zu einer gemeinsamen 64-Bit-Variablen zusammengefasst werden.

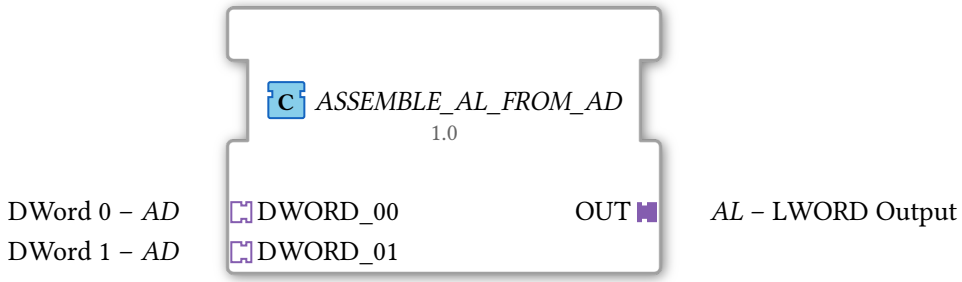
Vergleich mit ähnlichen Bausteinen

- **`ARRAY_TO_LWORD`-Bausteine**: Diese Bausteine arbeiten meist mit Arrays als Eingang und haben eine andere Schnittstelle (keine Adapter). Der `ASSEMBLE_AL_FROM_AB` bietet eine adapterbasierte, ereignisgesteuerte Alternative, die besser in modulare, typisierte Adapterstrukturen passt.
- **Multiplexer (MUX)**: Ein Multiplexer wählt einen von mehreren Eingängen aus, während dieser Baustein alle Eingänge gleichzeitig zu einem größeren Datenwort kombiniert.
- **Direkte Bitverkettung**: Ohne diesen Baustein müsste die Byte-Zusammenführung manuell durch bitweise Verschiebung und Oder-Verknüpfung erfolgen. Der `ASSEMBLE_AL_FROM_AB` kapselt diese Logik und reduziert den Entwicklungs- und Testaufwand.

Fazit

Der `ASSEMBLE_AL_FROM_AB` ist ein spezialisierter Funktionsblock für die modulare und ereignisgesteuerte Zusammenführung von acht Byte-Daten zu einem LWORD. Durch seine ausschließliche Verwendung von Adaptern eignet er sich hervorragend für den Einsatz in adapterbasierten Architekturen und fördert eine klare Trennung von Datenquellen und -senken. Die integrierte Pufferung über den Flip-Flop und die Auslösung über jedes eingehende Ereignis stellen sicher, dass stets ein konsistenter, vollständiger Datenwert ausgegeben wird. Der Baustein vereinfacht die Datenaggregation erheblich und trägt zu einer wartbaren und wiederverwendbaren Steuerungslogik bei.

ASSEMBLE_AL_FROM_AD



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASSEMBLE_AL_FROM_AD** dient dazu, zwei über AD-Adapter (DWORD) bereitgestellte 32-Bit-Werte zu einem 64-Bit-LWORD zusammenzuführen und diesen über einen AL-Adapter auszugeben. Er kapselt die Kombinationslogik und stellt eine gepufferte, ereignisgesteuerte Schnittstelle für 64-Bit-Daten bereit.

Schnittstellenstruktur

Ereignis-Eingänge

- **DWORD_00.E1** – Ereignis vom ersten AD-Adapter, das die Kombination anstößt.
- **DWORD_01.E1** – Ereignis vom zweiten AD-Adapter, das ebenfalls die Kombination anstößt.

Ereignis-Ausgänge

- **OUT.E1** – Ereignis am AL-Adapter, das signalisiert, dass ein neuer LWORD-Wert am Ausgang anliegt.

Daten-Eingänge

- **DWORD_00.D1** – Erster 32-Bit-DWORD (niederwertiger Teil).
- **DWORD_01.D1** – Zweiter 32-Bit-DWORD (höherwertiger Teil).

Daten-Ausgänge

- **OUT.D1** – Zusammengesetzter 64-Bit-LWORD.

Adapter

- **DWORD_00** (Socket, Typ `adapter::types::unidirectional::AD`) – Eingangsadapter für den ersten DWORD.

- **DWORD_01** (Socket, Typ `adapter::types::unidirectional::AD`) – Eingangsadapter für den zweiten DWORD.
- **OUT** (Plug, Typ `adapter::types::unidirectional::AL`) – Ausgangsadapter für den resultierenden LWORD.

Funktionsweise

Der FB arbeitet intern als FBNetzwerk aus zwei Unterbausteinen:

1. **ASSEMBLE_LWORD_FROM_DWORDS** – Kombiniert die beiden DWORD-Eingänge `DWORD_00` und `DWORD_01` zu einem LWORD. Der Baustein wird durch das Ereignis `REQ` getriggert, das von **beiden** AD-Adaptern (über `DWORD_00.E1` und `DWORD_01.E1`) gesendet werden kann. Nach Abschluss der Kombination wird das Ereignis `CNF` ausgegeben.
2. **E_D_FF_ANY** – Ein beliebiger D-Flipflop, der den kombinierten LWORD puffert. Das `CNF`-Ereignis taktet den Flipflop (`CLK`), wobei der kombinierte Wert am Dateneingang `D` anliegt. Der gespeicherte Wert erscheint am Ausgang `Q` und wird über `OUT.D1` weitergegeben. Gleichzeitig wird das Flipflop-Ausgangsereignis `EO` als `OUT.E1` ausgelöst.

Somit wird der LWORD nur dann aktualisiert, wenn einer der beiden AD-Adapter ein Ereignis sendet – das Flipflop verhindert mehrfache oder ungewollte Ausgaben bei unveränderten Eingängen.

Technische Besonderheiten

- **Ereignisgesteuerte Verarbeitung** – Die Kombination wird nur bei eingehenden Ereignissen der AD-Adapter ausgelöst, was unnötige Rechenlast vermeidet.
- **Gepufferter Ausgang** – Der D-Flipflop stellt sicher, dass ein einmal berechneter LWORD bis zur nächsten Aktualisierung stabil am Ausgang anliegt.
- **Unabhängigkeit der Eingänge** – Jeder der beiden DWORD-Adapter kann jederzeit ein Ereignis senden; die Kombination wird dann mit den aktuell anliegenden Werten durchgeführt.
- **Keine eigenen Ereignis-Eingänge auf oberster Ebene** – Der FB reagiert ausschließlich über die Adapter-Schnittstellen.

Zustandsübersicht

Der Baustein verwendet keinen expliziten ECC, sondern ein internes FBNetzwerk. Der funktionale Ablauf ist:

1. **Warten** auf ein Ereignis von `DWORD_00.E1` oder `DWORD_01.E1`.
2. **Kombinieren** – ASSEMBLE_LWORD_FROM_DWORDS verarbeitet die aktuellen DWORD-Werte und erzeugt den LWORD.
3. **Puffern** – Der Flipflop übernimmt den LWORD und gibt ihn über `OUT.D1` aus.
4. **Signalisieren** – Das Ereignis `OUT.E1` wird ausgelöst.
5. Zurück zum Wartezustand.

Eine gleichzeitige Anforderung beider Eingänge führt nur zu einer Verarbeitung, da beide Ereignisse denselben `REQ` des Kombinationsbausteins triggern.

Anwendungsszenarien

- **64-Bit-Datenverarbeitung** – Wenn ein System mit 32-Bit-Schnittstellen arbeitet und große Werte (z. B. hochauflösende Zeitstempel, Zählerstände, physikalische Messwerte) in einem LWORD zusammengefasst werden müssen.
- **Protokollumsetzung** – In Feldbussen oder Steuerungen, die Daten in separaten DWORD-Segmenten liefern und eine 64-Bit-Weiterverarbeitung erfordern.
- **Sichere Datenkombination** – Der gepufferte Ausgang verhindert Zwischenwerte bei asynchron ankommenden DWORDs.

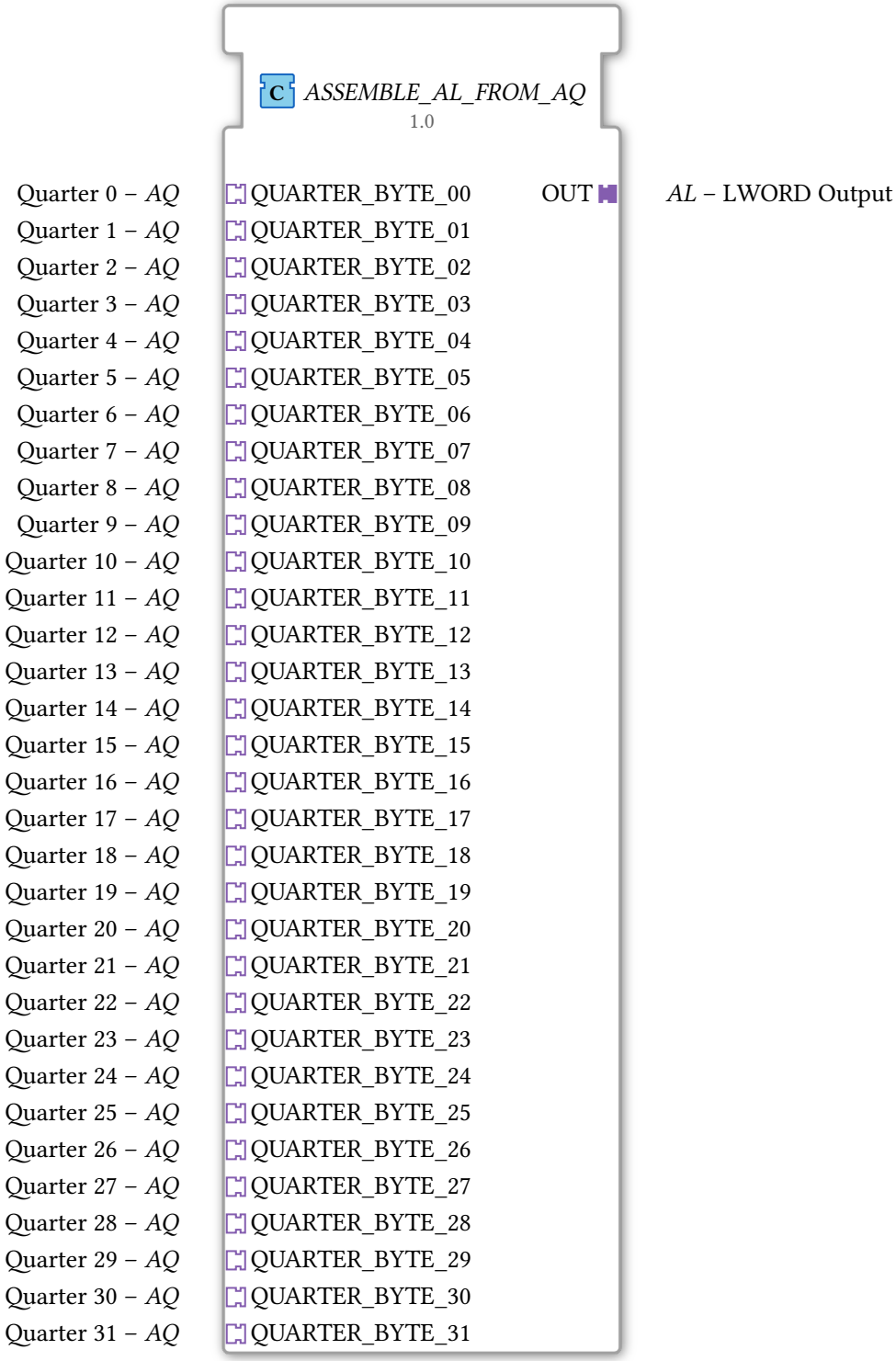
Vergleich mit ähnlichen Bausteinen

- **ASSEMBLE_DWORD_FROM_WORDS** – Kombiniert zwei WORDs zu einem DWORD; analoges Prinzip, aber für andere Wortbreiten.
- **ASSEMBLE_LWORD_FROM_DWORDS** – Der interne Baustein, der die reine Kombination ohne Pufferung durchführt; ASSEMBLE_AL_FROM_AD fügt die ereignisgesteuerte Pufferung und die Adapter-Schnittstelle hinzu.
- **Direkte Verwendung eines Typkonverters** – Wäre möglich, aber ohne Ereignissynchronisation; der vorliegende FB bietet eine saubere, industrialisierte Lösung mit Adaptern.

Fazit

Der **ASSEMBLE_AL_FROM_AD** Funktionsblock ist ein spezialisierter, aber nützlicher Baustein für die Zusammenführung zweier DWORDs zu einem LWORD. Er kombiniert ereignisgesteuerte Verarbeitung mit einer stabilen Ausgangspufferung und ist ideal für Steuerungssysteme, die 64-Bit-Daten über standardisierte Adapter austauschen. Die Verwendung von Adaptern ermöglicht eine einfache Einbindung in existierende Modellstrukturen.

ASSEMBLE_AL_FROM_AQ



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ASSEMBLE_AL_FROM_AQ` dient dazu, 32 unidirektionale AQ-Adapater („Quarter“) zu einem einzelnen AL-Adapater (LWORD) zusammenzufassen. Er ist als reiner Kompositionsbaustein ohne eigene Ereignis- oder Datenports auf der obersten Ebene realisiert und nutzt interne Sub-Funktionsblöcke, um die eingehenden Quarter-Daten zu einem 64-Bit-Wort zu kombinieren und ausgangsseitig zu stabilisieren.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt auf seiner obersten Ebene **keine** eigenen Ereignis-Eingänge. Die Ereignissteuerung erfolgt ausschließlich über die integrierten AQ-Adapater (Sockets) und die interne Verschaltung.

Ereignis-Ausgänge

Es sind **keine** ereignisausgangsseitigen Ports vorhanden. Das Ausgangssignal wird über den AL-Plug-Adapater (E1) ereignisgesteuert nach außen gegeben.

Daten-Eingänge

Es existieren **keine** direkt zugänglichen Daten-Eingänge. Die Daten werden über die 32 AQ-Adapater (Socket) in den Baustein eingebracht.

Daten-Ausgänge

Es existieren **keine** direkt zugänglichen Daten-Ausgänge. Das zusammengesetzte LWORD wird über den AL-Plug-Adapater (D1) ausgegeben.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AQ`	`QUARTER_BYTE_00` … `QUARTER_BYTE_31`	Socket (Eingang)	32 gleichartige Adapter, die jeweils einen 2-Bit-Wert („Quarter“) bereitstellen. Jeder Socket verfügt über einen Ereignisausgang (E1) und einen Datenausgang (D1).
`adapter::types::unidirectional::AL`	`OUT`	Plug (Ausgang)	Ausgangsadapter, der ein 64-Bit-LWORD (D1) sowie ein zugehöriges Ereignis (E1) weitergibt.

Funktionsweise

1. **Datenaufnahme** – Jeder der 32 `QUARTER_BYTE_xx`-Sockets liefert bei einem eingehenden Ereignis (E1) seinen Datenwert (D1).
2. **Zusammenstellung** – Das Ereignis wird an den internen Sub-Funktionsblock `ASSEMBLE_LWORD_FROM_QUARTERS` weitergeleitet (alle 32 Ereignisse sind mit demselben `REQ`-Eingang verbunden). Gleichzeitig werden alle 32 Quarter-Datenwerte an die entsprechenden Eingänge dieses Sub-Blocks geführt.
3. **Interne Verarbeitung** – `ASSEMBLE_LWORD_FROM_QUARTERS` kombiniert die 32 2-Bit-Quarter zu einem 64-Bit-LWORD. Sein Ausgangssignal wird anschließend über ein flankengesteuertes D-Flip-Flop (`E_D_FF_ANY`) geleitet.
4. **Ausgangsstabilisierung** – Das Flip-Flop wird durch das Abschlussereignis (`CNF`) von `ASSEMBLE_LWORD_FROM_QUARTERS` getaktet. Es hält den kombinierten Wert so lange, bis ein neuer Zusammenstellungszyklus abgeschlossen ist. Der Ausgang des Flip-Flops speist den Datenport `OUT.D1` des AL-Plug-Adapters, und das zugehörige Ereignis `OUT.E1` wird nach dem Takt ausgelöst.

Technische Besonderheiten

- **Reine Adapter-Komposition** – Der Baustein hat auf der obersten Ebene keine herkömmlichen Ein-/Ausgangsports, sondern nutzt ausschließlich Adapter. Dies ermöglicht eine modulare Kapselung und einfache Wiederverwendung in verschiedenen Kontexten.
- **Interne Datenaggregation** – Die 32 Quarter-Werte (je 2 Bit) werden zu einem 64-Bit-LWORD zusammengefügt. Das entspricht der vollständigen Ausnutzung der LWORD-Kapazität.
- **Flankengesteuerte Ausgangssynchronisation** – Das D-Flip-Flop verhindert Flackern oder unvollständige Datenwörter am Ausgang, indem es den Wert erst nach Abschluss der Zusammenstellung und mit steigender Taktflanke weitergibt.
- **Keine eigene Zustandslogik** – Der Baustein enthält keinen ECC (Execution Control Chart); die Logik ergibt sich vollständig aus der Verschaltung der Sub-Blöcke.

Zustandsübersicht

Der Baustein besitzt keinen eigenen Zustandsautomaten. Die Zustandslogik wird durch die internen Blöcke `ASSEMBLE_LWORD_FROM_QUARTERS` (datengesteuert) und `E_D_FF_ANY` (ereignisgesteuert) implementiert.

Anwendungsszenarien

- **Zusammenführung von Bit-Teilwörtern** – In der industriellen Steuerungstechnik müssen häufig Daten aus mehreren Sensoren oder Subsystemen (z. B. 32 Schaltzustände, die jeweils 2 Bit codieren) zu einem zusammenhängenden Datenwort vereint werden.
- **Abfrage paralleler Datenquellen** – Wenn 32 unabhängige Module jeweils ein 2-Bit-Signal über Adapter liefern, kann `ASSEMBLE_AL_FROM_AQ` diese in einem Schritt zu einem 64-Bit-Wort zusammenfassen und ereignisgesteuert bereitstellen.
- **Plattform-unabhängige Adapter-Schnittstellen** – Dank der reinen Adapternutzung eignet sich der Baustein für den Einsatz in heterogenen Systemen, bei denen die Datentypen durch die Adapterdefinitionen abstrahiert werden.

Vergleich mit ähnlichen Bausteinen

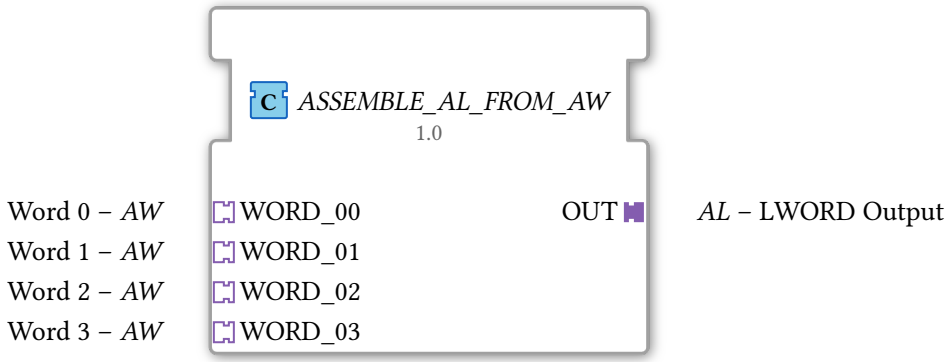
Baustein	Funktionsprinzip	Anzahl Eingänge	Ausgangstyp	Besonderheit
`ASSEMBLE_AL_FROM_AQ`	Adapter-basierte Zusammenstellung	32 × 2-Bit (AQ)	1 × LWORD (AL)	Flankengesteuerte Ausgabe, reine Komposition
`ASSEMBLE_LWORD_FROM_QUARTERS` (intern)	Daten-orientierte Kombination	32 × 2-Bit (direkt)	LWORD (Daten)	Keine Adapter, kein Flip-Flop
Klassischer Multiplexer (z. B. MUX)	Auswahl eines Eingangs über Steuerleitung	n Eingänge, 1 Auswahl	Einfacher Datentyp	Erfordert dezidierte Adresssignale

`ASSEMBLE_AL_FROM_AQ` bietet im Vergleich zu einem Multiplexer den Vorteil, dass *alle* Quarter-Werte parallel und ohne Selektionslogik zu einem vollständigen Wort kombiniert werden. Der zusätzliche Flip-Flop sorgt für eine saubere, ereignisgesteuerte Ausgabe.

Fazit

Der Funktionsblock `ASSEMBLE_AL_FROM_AQ` stellt eine elegante und modulare Lösung dar, um 32 2-Bit-Quarter-Daten über Adapter zu einem 64-Bit-LWORD zusammenzusetzen. Die strikte Trennung von Daten- und Ereignispfaden sowie die interne Nutzung eines flankengesteuerten Flip-Flops gewährleisten eine zuverlässige und deterministische Ausgangssignalbildung. Aufgrund seiner reinen Kompositionsstruktur eignet er sich besonders für den Einsatz in IEC-61499-Systemen, die auf adapterbasierte Schnittstellen setzen.

ASSEMBLE_AL_FROM_AW



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ASSEMBLE_AL_FROM_AW` kombiniert die Daten von vier **AW (WORD)**-Adapter-Schnittstellen zu einem **AL (LWORD)**-Ausgangswert. Er dient dazu, mehrere 16-Bit-Wörter zu einem 64-Bit-Wort zusammenzusetzen und über eine getaktete Ausgabe bereitzustellen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine – die Ereignissteuerung erfolgt ausschließlich über die angeschlossenen Adapter (siehe Abschnitt Adapter).

Ereignis-Ausgänge

Keine – die Ereignisausgabe erfolgt über den OUT-Adapter (`OUT.E1`).

Daten-Eingänge

Keine direkten Daten-Eingänge – die zu verarbeitenden Daten werden über die vier AW-Sockets bezogen.

Daten-Ausgänge

Keine direkten Daten-Ausgänge – das Ergebnis wird über den OUT-Adapter ausgegeben.

Adapter

Name	Typ	Kommentar
Sockets		
`WORD_00`	`adapter::types::unidirectional::AW`	Word 0 (niederwertigstes 16-Bit-Wort)
`WORD_01`	`adapter::types::unidirectional::AW`	Word 1
`WORD_02`	`adapter::types::unidirectional::AW`	Word 2
`WORD_03`	`adapter::types::unidirectional::AW`	Word 3 (höchstwertiges 16-Bit-Wort)
Plugs		
`OUT`	`adapter::types::unidirectional::AL`	LWORD-Ausgang (64-Bit)

Funktionsweise

Der Baustein nutzt ein internes Netzwerk aus zwei Funktionsbausteinen:

1. **`ASSEMBLE_LWORD_FROM_WORDS`** – übernimmt die logische Kombination der vier 16-Bit-Wörter (`WORD_00` bis `WORD_03`) zu einem 64-Bit-LWORD.
2. **`E_D_FF_ANY`** – ein taktflankengesteuertes D-Flip-Flop, das den zusammengesetzten Wert nur dann auf den Ausgangsadapter `OUT` überträgt, wenn eine steigende Flanke am Takteingang anliegt.

Die Ereignisse der vier Eingangsadapter (`WORD_00.E1` … `WORD_03.E1`) sind alle mit dem `REQ`-Ereigniseingang von `ASSEMBLE_LWORD_FROM_WORDS` verbunden. Sobald ein Ereignis an einem der Eingangsadapter eintrifft, wird die Kombination neu berechnet. Nach Abschluss (`CNF`) wird der `E_D_FF_ANY` getaktet, der den berechneten Wert in seinen internen Zustand übernimmt und über `OUT.D1` ausgibt. Das Ereignis am Ausgangsadapter (`OUT.E1`) signalisiert die Verfügbarkeit des neuen Werts.

Technische Besonderheiten

- **Taktflankengesteuerte Ausgabe** – Die Verwendung des `E_D_FF_ANY` verhindert ungewollte Zwischenwerte und stellt sicher, dass nur vollständig zusammengesetzte LWORDs nach außen gegeben werden.
- **Beliebige Reihenfolge der Eingangsereignisse** – Da alle vier Sockets auf denselben `REQ`-Eingang zusammengeführt sind, löst jedes eingehende Ereignis eine Neuberechnung aus. Der Baustein reagiert also auf jedes Teilwort, aktualisiert aber den Ausgang erst nach der vollständigen Zusammensetzung.
- **Adapterbasierte Schnittstelle** – Die Verwendung von unidirektionalen Adaptern (AW und AL) ermöglicht eine lose Kopplung und Wiederverwendung in verschiedenen Kontexten.

Zustandsübersicht

Der Baustein besitzt kein eigenes ECC (Execution Control Chart), sondern besteht aus einer kombinatorischen Schaltung mit einem nachgeschalteten Flip-Flop. Die interne Zustandslogik wird vollständig durch die beiden Unterbausteine `ASSEMBLE_LWORD_FROM_WORDS` und `E_D_FF_ANY` realisiert.

Anwendungsszenarien

- **Industrielle Datenerfassung** – Zusammenführen von vier 16-Bit-Sensorwerten (z. B. Temperatur, Druck, Durchfluss, Füllstand) zu einem einzigen 64-Bit-Datenpaket.
- **Kommunikationsschnittstellen** – Aufbereitung von getrennt eintreffenden 16-Bit-Datenblöcken für eine 64-Bit-Schnittstelle (z. B. Modbus-Register ⟶ OPC UA).
- **Bit-parallele Verarbeitung** – Kombinieren von vier Registern eines Mikrocontrollers zu einem Großregister für erweiterte Logikanwendungen.

Vergleich mit ähnlichen Bausteinen

Baustein	Beschreibung
`ASSEMBLE_AL_FROM_BYTES`	Setzt mehrere 8-Bit-Werte zu einem LWORD zusammen
`ASSEMBLE_AL_FROM_DWORDS`	Kombiniert zwei 32-Bit-Doppelwörter zu einem 64-Bit-Wort
`ASSEMBLE_AL_FROM_AW` (dieser)	Kombiniert vier 16-Bit-Wörter zu einem LWORD

Alle diese Bausteine folgen dem gleichen Prinzip, unterscheiden sich jedoch in der Wortbreite der Eingangsdaten und der Anzahl der benötigten Sockets.

Fazit

`ASSEMBLE_AL_FROM_AW` ist ein spezialisierter Funktionsbaustein zum Zusammenfügen von vier 16-Bit-Wörtern zu einem 64-Bit-Wort. Die getaktete Ausgabe über ein D-Flip-Flop sorgt für konsistente Datenweitergabe. Dank der adapterbasierten Schnittstelle fügt er sich nahtlos in modulare 4diac-Projekte ein und erleichtert die Handhabung von Wort-zu-Langwort-Konvertierungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 MCU vs. MPU Vergleichsguide auf ms-muc-docs.de

ASSEMBLE_AL_FROM_AX

(Bild des FB wird hier in der IDE dargestellt – zeigt einen Funktionsblock mit 64 Sockets (links) und einem Plug (rechts))

ASSEMBLE_AL_FROM_AX
1.0
Bool Bit 0 – AX
BIT_00
OUT
AL – LWORD Output
Bool Bit 1 – AX
BIT_01
Bool Bit 2 – AX
BIT_02
Bool Bit 3 – AX
BIT_03
Bool Bit 4 – AX
BIT_04
Bool Bit 5 – AX
BIT_05
Bool Bit 6 – AX
BIT_06
Bool Bit 7 – AX
BIT_07
Bool Bit 8 – AX
BIT_08
Bool Bit 9 – AX
BIT_09
Bool Bit 10 – AX
BIT_10
Bool Bit 11 – AX
BIT_11
Bool Bit 12 – AX
BIT_12
Bool Bit 13 – AX
BIT_13
Bool Bit 14 – AX
BIT_14
Bool Bit 15 – AX
BIT_15
Bool Bit 16 – AX
BIT_16
Bool Bit 17 – AX
BIT_17
Bool Bit 18 – AX
BIT_18
Bool Bit 19 – AX
BIT_19
Bool Bit 20 – AX
BIT_20
Bool Bit 21 – AX
BIT_21
Bool Bit 22 – AX
BIT_22
Bool Bit 23 – AX
BIT_23
Bool Bit 24 – AX
BIT_24
Bool Bit 25 – AX
BIT_25
Bool Bit 26 – AX
BIT_26
Bool Bit 27 – AX
BIT_27
Bool Bit 28 – AX
BIT_28
Bool Bit 29 – AX
BIT_29
Bool Bit 30 – AX
BIT_30
Bool Bit 31 – AX
BIT_31
Bool Bit 32 – AX
BIT_32
Bool Bit 33 – AX
BIT_33
Bool Bit 34 – AX
BIT_34
Bool Bit 35 – AX
BIT_35
Bool Bit 36 – AX
BIT_36
Bool Bit 37 – AX
BIT_37
Bool Bit 38 – AX
BIT_38
Bool Bit 39 – AX
BIT_39
Bool Bit 40 – AX
BIT_40
Bool Bit 41 – AX
BIT_41
Bool Bit 42 – AX
BIT_42
Bool Bit 43 – AX
BIT_43
Bool Bit 44 – AX
BIT_44
Bool Bit 45 – AX
BIT_45
Bool Bit 46 – AX
BIT_46
Bool Bit 47 – AX
BIT_47
Bool Bit 48 – AX
BIT_48
Bool Bit 49 – AX
BIT_49
Bool Bit 50 – AX
BIT_50
Bool Bit 51 – AX
BIT_51
Bool Bit 52 – AX
BIT_52
Bool Bit 53 – AX
BIT_53
Bool Bit 54 – AX
BIT_54
Bool Bit 55 – AX
BIT_55
Bool Bit 56 – AX
BIT_56
Bool Bit 57 – AX
BIT_57
Bool Bit 58 – AX
BIT_58
Bool Bit 59 – AX
BIT_59
Bool Bit 60 – AX
BIT_60
Bool Bit 61 – AX
BIT_61
Bool Bit 62 – AX
BIT_62
Bool Bit 63 – AX
BIT_63

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASSEMBLE_AL_FROM_AX** dient dazu, bis zu 64 boolesche Signale von **AX-Adaptern** (Typ: `adapter::types::unidirectional::AX`) zu einem einzigen **LWORD-Wert** zusammenzufassen und über einen **AL-Adapter** (Typ: `adapter::types::unidirectional::AL`) auszugeben. Der Baustein kapselt die Logik zur Bit-Kombination und stellt das Ergebnis stabil zur Verfügung, sobald eines der Eingangsbits ein Ereignis liefert.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine klassischen Ereignis-Eingänge (EVENT). Die Ereignissteuerung erfolgt indirekt über die **AX-Adapter (Sockets)**. Jeder Socket `BIT_00` … `BIT_63` kann ein auslösendes Ereignis (E1) empfangen.

Name	Typ	Beschreibung
`BIT_00` … `BIT_63`	AX-Adapter	Ereignis von einem externen Bool-Quelle, das die Zusammenstellung des LWORD auslöst

Ereignis-Ausgänge

Keine klassischen Ereignis-Ausgänge. Die Ausgabe des zusammengesetzten LWORD erfolgt ereignisgesteuert über den **AL-Adapter (Plug)** `OUT`.

Name	Typ	Beschreibung
`OUT`	AL-Adapter	Stellt das zusammengesetzte LWORD bereit, wenn das interne Flip-Flop getaktet wurde

Daten-Eingänge

Jeder AX-Adapter führt einen booleschen Datenwert (D1). Diese werden als Eingangsbits für das LWORD verwendet.

Name	Typ	Beschreibung
`BIT_00.D1` … `BIT_63.D1`	BOOL (über Adapter)	Boolesches Signal für Bit 0 … Bit 63

Daten-Ausgänge

Das zusammengesetzte Ergebnis wird als 64-Bit-Wort über den AL-Adapter ausgegeben.

Name	Typ	Beschreibung
`OUT.D1`	LWORD (über Adapter)	Kombiniertes LWORD aus den 64 booleschen Eingängen

Adapter

Typ	Richtung	Name	Beschreibung
`adapter::types::unidirectional::AX`	Socket (Eingang)	`BIT_00` … `BIT_63`	Boolesche Eingangsadapter
`adapter::types::unidirectional::AL`	Plug (Ausgang)	`OUT`	LWORD-Ausgangsadapter

Funktionsweise

1. **Ereignisempfang**: Sobald ein Ereignis an einem der 64 AX-Adapter (Socket) eintrifft (an dessen E1), wird ein internes REQ-Ereignis an den Funktionsblock `ASSEMBLE_LWORD_FROM_BOOLS` gesendet.
2. **Bit-Zusammenstellung**: Der interne FB `ASSEMBLE_LWORD_FROM_BOOLS` übernimmt die 64 booleschen Werte von den Datenanschlüssen `BIT_00.D1` … `BIT_63.D1` und setzt daraus einen LWORD-Wert zusammen. Jeder Eingang wird dabei einem entsprechenden Bit (0 für BIT_00, 1 für BIT_01, … 63 für BIT_63) zugeordnet.
3. **Zwischenspeicherung (Flip-Flop)**: Das zusammengestellte LWORD wird an den Daten-Eingang eines **E_D_FF_ANY** (Flip-Flop) übergeben. Das Ergebnis-Ereignis (CNF) von

`ASSEMBLE_LWORD_FROM_BOOLS` taktet das Flip-Flop. Dadurch wird der aktuelle LWORD-Wert am Ausgang Q des Flip-Flops übernommen.

4. **Ausgabe**: Der gespeicherte Wert wird über den AL-Plug `OUT` als LWORD bereitgestellt. Gleichzeitig wird ein Ereignis am Ausgang des Flip-Flops (EO) erzeugt, das den angeschlossenen AL-Adapter-Eingang (E1) auslöst.

Die Logik arbeitet **ereignisgesteuert**: Nur bei einer Änderung eines beliebigen Eingangsbits (über das Ereignis) wird der Ausgang aktualisiert. Dadurch wird Rechenleistung gespart und eine stabile Ausgabe gewährleistet.

Technische Besonderheiten

- **Vollständige Bit-Zuordnung**: Alle 64 Bits eines LWORD werden aus einzelnen booleschen Adaptern bezogen. Es werden genau 64 Sockets benötigt – keine Lücken.
- **Ereignis-Synchronisation**: Ein Ereignis von *einem* beliebigen Bit-Adapter löst die komplette Neuberechnung des LWORD aus. Es ist nicht nötig, alle Eingänge gleichzeitig zu aktualisieren.
- **Flip-Flop-Pufferung**: Der zusammengesetzte Wert wird zwischengespeichert, sodass der Ausgang auch dann stabil bleibt, wenn die Eingänge während der Verarbeitung schwanken.
- **Adapterbasierte Schnittstelle**: Die Ein- und Ausgänge sind als standardisierte 4diac-Adapter realisiert, wodurch eine einfache Verkettung mit anderen Adapter-basierten Bausteinen möglich ist.
- **Packages**: Der FB ist im Package `adapter::assembling` organisiert.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten. Die interne Logik arbeitet als Kombination von zwei Komponenten:

- **ASSEMBLE_LWORD_FROM_BOOLS**: Zustandslos – führt bei jedem REQ die Zusammenstellung aus.
- **E_D_FF_ANY**: Besitzt zwei Zustände: gespeicherter Wert (Q) und aktueller Daten-Eingang (D). Der Zustand wechselt bei jeder positiven Flanke am CLK-Eingang.

Ein ereignisgesteuertes Verhalten:

- **Warten auf Ereignis**: Keine Berechnung, Ausgang bleibt unverändert.
- **Ereignis eingetroffen**: Zusammenstellung + Takten des Flip-Flops → Ausgang wird aktualisiert.

Anwendungsszenarien

- **Parallel-Digital-Eingänge**: Zusammenfassen von 64 digitalen Sensoren oder Schaltern in ein maschinenlesbares LWORD für einfache Weiterverarbeitung.
- **Bit-orientierte Steuerungen**: In Anwendungen, bei denen viele binäre Statusmeldungen (z. B. Fehlerbits, Positionsrückmeldungen) gebündelt werden müssen.
- **Kommunikation über Feldbus**: Ein LWORD kann effizient über Profinet, EtherCAT o. ä. übertragen werden; dieser FB wandelt die einzelnen Bits in das kompakte Datenformat um.
- **Adapter-basierte Modul-Bibliotheken**: In modularen Steuerungsarchitekturen, die auf standardisierten Adaptern aufbauen, kann dieser Baustein als universeller „Bit-Sammler“ dienen.

Vergleich mit ähnlichen Bausteinen

Baustein	Anzahl Eingänge	Ausgangstyp	Besonderheit
ASSEMBLE_AL_FROM_AX	64 BOOL (AX)	LWORD (AL)	Adapter-basiert mit Flip-Flop-Pufferung
`ASSEMBLE_DWORD_FROM_BOOLS`	32 BOOL	DWORD	Klassische Daten-Ein-/Ausgänge, ohne Adapter
`ASSEMBLE_WORD_FROM_BOOLS`	16 BOOL	WORD	Wie oben, für 16-Bit-Wort
`DISASSEMBLE_AL_TO_AX`	1 LWORD (AL)	64 AX	Gegenstück: Zerlegt ein LWORD in einzelne BOOL-Adapter

Der wesentliche Unterschied liegt in der **Adapter-Schnittstelle** und der **Zwischenspeicherung** durch das Flip-Flop, was für robustes Verhalten in ereignisgesteuerten Umgebungen sorgt.

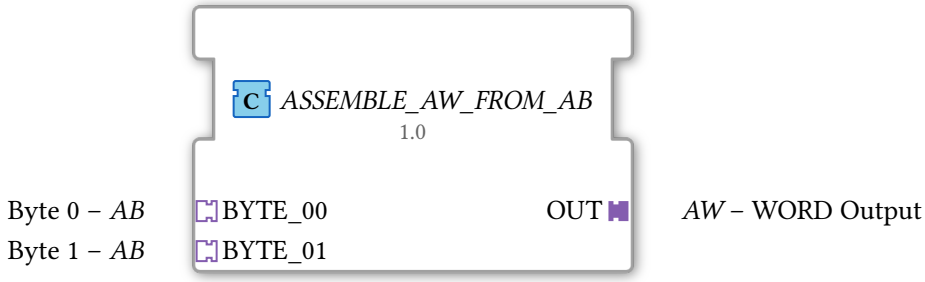
Fazit

Der Funktionsblock **ASSEMBLE_AL_FROM_AX** ist ein leistungsfähiges Werkzeug, um aus 64 dezentralen booleschen Signalen ein kompaktes 64-Bit-LWORD zu erzeugen. Durch die adapterbasierte Anbindung und die interne Flip-Flop-Pufferung eignet er sich besonders für saubere, ereignisgesteuerte und modulare Automatisierungslösungen. Die vollständige Abdeckung aller Bits ohne Skalierungsprobleme macht ihn zu einer universellen Lösung für Anwendungen, die viele digitale Eingänge bündeln müssen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ASSEMBLE_AW_FROM_AB



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **ASSEMBLE_AW_FROM_AB** dient dazu, zwei unidirektionale Byte-Adapter (Typ `AB`) zu einem unidirektionalen Word-Adapter (Typ `AW`) zusammenzufügen. Er kombiniert die Daten zweier BYTE-Eingänge zu einem WORD-Ausgang, wobei die Ereignislogik sicherstellt, dass die Ausgabe nur bei aktualisierter Information erfolgt.

Schnittstellenstruktur

Der FB besitzt keine direkten Ereignis- oder Datenein-/ausgänge, sondern kommuniziert ausschließlich über Adapter-Schnittstellen:

Adapter (Plugs – Ausgang)

Name	Typ	Richtung	Beschreibung
OUT	`adapter::types::unidirectional::AW`	Plug	WORD-Ausgabe (16 Bit)

Adapter (Sockets – Eingänge)

Name	Typ	Richtung	Beschreibung
BYTE_00	`adapter::types::unidirectional::AB`	Socket	Byte 0 (niedrigwertiges Byte)
BYTE_01	`adapter::types::unidirectional::AB`	Socket	Byte 1 (höherwertiges Byte)

Hinweis: Die Adapter vom Typ `AB` und `AW` sind unidirektional ausgeführt. Jeder Adapter enthält die Signale `E1` (Event-Ausgang des Sockets/Plug – intern verbunden) und `D1` (Datensignal).

Funktionsweise

Der FB setzt intern auf zwei Subbausteinen auf:

1. **`ASSEMBLE_WORD_FROM_BYTES`** – ein vordefinierter Assembler, der zwei Eingangsbytes zu einem WORD zusammenfasst.
2. **`E_D_FF_ANY`** – ein D-Flipflop, das den kombinierten Wert zwischenspeichert und bei einem Taktereignis weitergibt.

Ablauf:

- Jeder eingehende Event (`E1`) eines BYTE-Adapters (BYTE_00 oder BYTE_01) triggert den Assembler (`REQ`).
- Der Assembler kombiniert die aktuellen Daten beider BYTE-Eingänge zu einem WORD.
  Dabei gilt:
  `OUT.WORD = (BYTE_01.D1 << 8) | BYTE_00.D1`
- Nach erfolgreicher Kombination sendet der Assembler einen Bestätigungs-Event (`CNF`).
- Dieser Event taktet das D-Flipflop (`CLK`), das den aktuellen WORD-Wert übernimmt.
- Der Flipflop-Ausgang (`Q`) wird kontinuierlich an den OUT-Adapter (`D1`) weitergegeben.
- Der Ausgangs-Event (`E0`) des Flipflops aktiviert `OUT.E1`, sodass der empfangende Baustein über die aktualisierte WORD-Ausgabe informiert wird.

Wichtige Eigenschaft:
Da beide BYTE-Events auf den gleichen `REQ` des Assemblers führen, wird bei jedem Ereignis eines der beiden Eingänge die Assemblierung neu durchgeführt. Der Ausgangswert wird also stets aus den gerade anliegenden Byte-Werten gebildet. Das D-Flipflop sorgt für einen stabilen Ausgang, bis der nächste Assemblierungs-Event eintrifft.

Technische Besonderheiten
- **Reine Adapter-Kommunikation:** Der FB besitzt keine herkömmlichen Ein-/Ausgangsvariablen; die gesamte Daten- und Ereignisübertragung erfolgt über unidirektionale Adapter.
- **Ereignis-Synchronisation:** Der einschrittige Assemblierungsprozess vermeidet Wettlaufsituationen, da das Flipflop die Ausgabe erst nach erfolgreicher Berechnung aktualisiert.
- **Wiederverwendbarkeit:** Der FB kann in jede Umgebung eingebettet werden, die unidirektionale `AB`/`AW`-Adapter unterstützt.

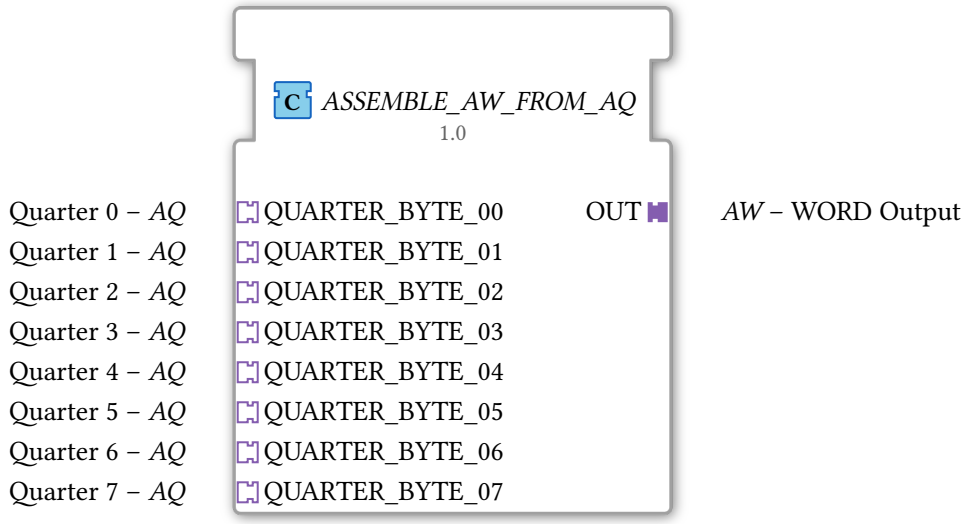
Zustandsübersicht
Der FB besitzt kein explizites Zustandsdiagramm, da die interne Logik rein ereignisgesteuert arbeitet. Im Wesentlichen lassen sich folgende Betriebszustände unterscheiden:
- **Bereit:** Es liegt kein Ereignis an; der Ausgang hält den zuletzt gespeicherten WORD-Wert.
- **Assemblieren:** Ein eingehender Event eines BYTE-Adapters startet die Berechnung.
- **Ausgeben:** Nach erfolgreicher Assemblierung wird der neue WORD-Wert durch das Flipflop übernommen und der Ausgangs-Event ausgelöst.

Anwendungsszenarien
- **Protokoll-Konvertierung:** Zusammenführen zweier serieller Byte-Ströme zu einem WORD-Datenwort für einen nachfolgenden Verarbeitungsbaustein.
- **Sensorfusion:** Kombination von zwei 8-Bit-Sensordaten (z. B. Temperatur und Luftfeuchte) zu einem 16-Bit-Wert.
- **Hardware-Ansteuerung:** Erzeugen eines 16-Bit-Ausgangssignals aus zwei getrennten 8-Bit-Registerwerten.

Vergleich mit ähnlichen Bausteinen
- **`ASSEMBLE_WORD_FROM_BYTES`** – reiner Assembler ohne Speicher. Er erwartet diskrete Daten-Eingänge und gibt das Ergebnis sofort aus (vorausgesetzt, Ereignis und Daten werden synchron bereitgestellt).
- **`SPLIT_BYTE_FROM_WORD`** – inverse Funktion (WORD -> zwei BYTEs); symmetrisch aufgebaut, nutzt ebenfalls unidirektionale Adapter.
- **Eigener FB mit Speicher:** Der vorliegende Baustein integriert das Flipflop, sodass die Ausgabe stabil bleibt, bis neue Daten anliegen – im Gegensatz zu Bausteinen, die bei jedem Event neu berechnen, aber keine Zwischenspeicherung bieten.

Fazit
Der FB **ASSEMBLE_AW_FROM_AB** bietet eine zuverlässige und elegante Möglichkeit, zwei unidirektionale BYTE-Adapter zu einem WORD-Ausgang zu kombinieren. Durch die Kombination aus Assemble- und Flipflop-Logik entsteht ein stabiler Ausgang, der nur bei tatsächlicher Datenänderung aktualisiert wird. Seine reine Adapter-Schnittstelle macht ihn flexibel einsetzbar in modularen Steuerungsarchitekturen nach IEC 61499.

ASSEMBLE_AW_FROM_AQ



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ASSEMBLE_AW_FROM_AQ` dient dazu, acht einzelne **AQ (Quarter)**-Adapter zu einem zusammenhängenden **AW (Word)**-Adapter zu kombinieren. Ein Quarter repräsentiert jeweils ein 4-Bit-Datenpaket (halbes Byte). Der Baustein setzt diese acht Quarter-Daten – insgesamt 32 Bit – zu einem vollständigen Wort (WORD, 32 Bit) zusammen und stellt es über einen einzigen AW-Ausgangsadapter bereit. Typische Anwendungen sind die Rekonstruktion breiter Datenbusse aus seriell oder partiell ankommenden Bruchstücken.

Schnittstellenstruktur

Der FB besitzt ausschließlich **Adapter-Schnittstellen**: Es gibt keine direkten Ereignis- oder Dateneingänge/-ausgänge auf der obersten Ebene. Die gesamte Kommunikation erfolgt über die angeschlossenen Adapter.

Ereignis-Eingänge

- Keine direkten Ereigniseingänge.
- Ereignisse werden über die Sockets der **AQ-Adapter** empfangen (jeder AQ-Adapter stellt einen Ereignisausgang `E1` bereit).

Ereignis-Ausgänge

- Keine direkten Ereignisausgänge.
- Das Ausgangsereignis wird über den **AW-Adapter** (Plug `OUT`) als Ereignis `E1` zur Verfügung gestellt.

Daten-Eingänge

- Keine direkten Dateneingänge.
- Die zu verarbeitenden Daten werden über die **AQ-Adapter** als Dateneingang `D1` (jeweils 4 Byte) eingelesen.

Daten-Ausgänge

- Keine direkten Datenausgänge.
- Das zusammengesetzte Wort (4 Byte) wird über den **AW-Adapter** als Datenausgang `D1` ausgegeben.

Adapter

Name	Typ	Richtung	Kommentar
`OUT`	adapter::AW	Plug	Wort-Ausgang (4 Byte)
`QUARTER_BYTE_00`	adapter::AQ	Socket	Quarter 0 (niederwertigstes)

QUARTER_BYTE_01	adapter::AQ	Socket	Quarter 1
QUARTER_BYTE_02	adapter::AQ	Socket	Quarter 2
QUARTER_BYTE_03	adapter::AQ	Socket	Quarter 3
QUARTER_BYTE_04	adapter::AQ	Socket	Quarter 4
QUARTER_BYTE_05	adapter::AQ	Socket	Quarter 5
QUARTER_BYTE_06	adapter::AQ	Socket	Quarter 6
QUARTER_BYTE_07	adapter::AQ	Socket	Quarter 7 (höchstwertigstes)

Funktionsweise

Der FB enthält intern zwei Funktionsblöcke:

1. **ASSEMBLE_WORD_FROM_QUARTERS**: Ein spezialisierter Assembler, der die Daten aller acht Quarter-Eingänge (jeweils 4 Byte) in der Reihenfolge 00 (LSB) bis 07 (MSB) zu einem 32-Bit-Wort zusammenfügt.
2. **E_D_FF_ANY**: Ein flankengesteuertes D-Flipflop, das das zusammengesetzte Wort zwischenspeichert und am Ausgang Q bereitstellt.

Der Ablauf:

- Sobald ein angeschlossener AQ-Adapter ein Ereignis auf seinem Ausgang E1 sendet, wird dieses Ereignis an den REQ-Eingang des Assemblers weitergeleitet. Alle acht AQ-Adapter sind mit dem gleichen REQ verbunden, sodass jedes eingehende Ereignis (unabhängig vom Absender) eine neue Verarbeitung anstößt.
- Der Assembler verarbeitet die aktuellen Daten aller acht Quarter und legt das Ergebnis am Datenausgang an.
- Nach Abschluss der Berechnung sendet der Assembler ein Bestätigungsereignis (CNF), das den Takteingang (CLK) des D-Flipflops triggert.
- Das Flipflop übernimmt den berechneten Wert und gibt ihn stabil an Q weiter. Über den OUT-Adapter wird dann sowohl das Ereignis E1 als auch der Datenwert D1 auf der Ausgangsseite verfügbar.

Durch diese Struktur wird sichergestellt, dass das Ausgangswort erst dann aktualisiert wird, wenn eine vollständige Berechnung durchgeführt wurde – unabhängig davon, welcher Quarter als letztes aktualisiert wurde.

Technische Besonderheiten

- **Zwischenspeicherung mit Flipflop**: Der Einsatz eines D-Flipflops vermeidet Glitches oder unvollständige Datenweitergabe, falls mehrere Quarter in schneller Folge aktualisiert werden. Der Ausgang ändert sich nur dann, wenn der Assembler einen neuen, gültigen Wert berechnet hat.
- **Adapter-basierte Schnittstelle**: Alle Ein- und Ausgaben erfolgen über Adapter, was den FB besonders geeignet für modulare, typisierte Datenflussarchitekturen in 4diac macht.
- **Synchroner Ablauf**: Die Verarbeitung wird durch jedes eingehende Ereignis eines Quarters angestoßen, erzeugt jedoch nur dann ein Ausgangsereignis, wenn die Berechnung abgeschlossen ist.

Zustandsübersicht

Der FB besitzt kein eigenes ECC (Execution Control Chart) – die interne Logik wird vollständig durch die enthaltenen Bausteine ASSEMBLE_WORD_FROM_QUARTERS und E_D_FF_ANY realisiert. Der Baustein verhält sich daher wie eine kombinatorische Schaltung mit einer flankengesteuerten Speicherstufe.

Anwendungsszenarien

- **Rekonstruktion breiter Datenpfade**: In der Automatisierungstechnik werden Daten oft in kleinen Paketen („Nibbles“) übertragen, z.B. über mehrere parallele E/A-Module. Der FB setzt diese wieder zu einer vollständigen 32-Bit-Größe zusammen.
- **Seriell-parallele Umsetzung**: Wenn ein Sensor oder Aktor seine Daten in 4-Bit-Quarten liefert (z.B. über asynchrone Schnittstellen), kann der FB die ankommenden Bruchstücke zwischenspeichern und zu einem konsistenten Gesamtwert vereinen.
- **Datenkonsolidierung**: In hierarchischen Steuerungssystemen, in denen mehrere Unterbausteine jeweils einen Quartierwert liefern, kann dieser FB die Werte zentral zusammenführen.

Vergleich mit ähnlichen Bausteinen

- **`ASSEMBLE_BYTE_FROM_NIBBLES`**: Kombiniert zwei Halbbytes zu einem Byte (8 Bit). Der vorliegende FB arbeitet auf der nächsthöheren Ebene mit 8 Quartieren (32 Bit).
- **Einfache Datenkonverter**: Konverter wie `WORD_TO_BYTE` oder `BYTE_TO_NIBBLE` arbeiten meist ohne Zwischenspeicher und Ereignissteuerung. `ASSEMBLE_AW_FROM_AQ` bietet eine ereignisgesteuerte, stabile Weiterleitung mit optionale Speicherung.

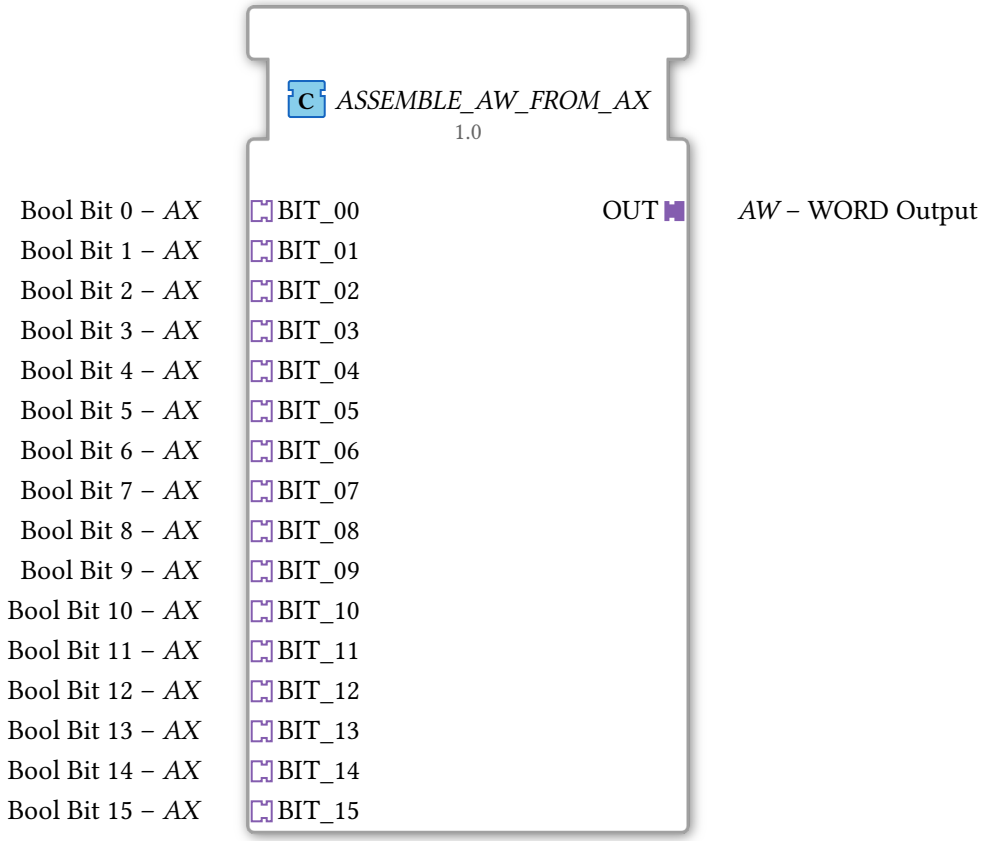
Fazit

Der Funktionsblock `ASSEMBLE_AW_FROM_AQ` stellt eine robuste und flexible Lösung dar, um aus acht unabhängigen Quarter-Daten (je 4 Bit) ein konsistentes 32-Bit-Wort zu erzeugen. Dank der adapterbasierten Schnittstellen und des integrierten D-Flipflops eignet er sich besonders für Echtzeitanwendungen, in denen Datenpakete asynchron eintreffen und zuverlässig zu einem Gesamtwert vereint werden müssen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ASSEMBLE_AW_FROM_AX



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ASSEMBLE_AW_FROM_AX` fasst 16 boolesche Einzelsignale, die von separaten AX-Adaptern bereitgestellt werden, zu einem 16-Bit-Wort (WORD) zusammen und gibt dieses über einen AW-Adapter aus. Er ermöglicht die Konvertierung einer diskreten digitalen Signalgruppe in einen einheitlichen Datenwort-Wert für die weitere Verarbeitung.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge. Die Ereignisse werden über die 16 AX-Adapter (Sockets) empfangen: Jeder AX-Adapter löst bei einer Datenänderung ein Ereignis (`E1`) aus, das die Verarbeitung anstößt.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge. Das zusammengesetzte Wort wird über den AW-Adapter (Plug) ausgegeben. Der Ausgangsadapter gibt den Datenwert frei, sobald er ein Ereignis (`E1`) vom internen Flip-Flop erhält.

Daten-Eingänge

Keine direkten Daten-Eingänge. Die 16 Einzelwerte (BOOL) werden über die AX-Adapter (Datenausgang `D1`) eingelesen.

Daten-Ausgänge

Keine direkten Daten-Ausgänge. Das resultierende 16-Bit-Wort (WORD) wird über den AW-Adapter (Dateneingang `D1`) ausgegeben.

Adapter

Typ	Name	Beschreibung
Socket (Eingang)	`BIT_00`	AX-Adapter, Bool Bit 0
Socket (Eingang)	`BIT_01`	AX-Adapter, Bool Bit 1
Socket (Eingang)	`BIT_02`	AX-Adapter, Bool Bit 2
Socket (Eingang)	`BIT_03`	AX-Adapter, Bool Bit 3
Socket (Eingang)	`BIT_04`	AX-Adapter, Bool Bit 4
Socket (Eingang)	`BIT_05`	AX-Adapter, Bool Bit 5
Socket (Eingang)	`BIT_06`	AX-Adapter, Bool Bit 6
Socket (Eingang)	`BIT_07`	AX-Adapter, Bool Bit 7
Socket (Eingang)	`BIT_08`	AX-Adapter, Bool Bit 8
Socket (Eingang)	`BIT_09`	AX-Adapter, Bool Bit 9
Socket (Eingang)	`BIT_10`	AX-Adapter, Bool Bit 10
Socket (Eingang)	`BIT_11`	AX-Adapter, Bool Bit 11
Socket (Eingang)	`BIT_12`	AX-Adapter, Bool Bit 12
Socket (Eingang)	`BIT_13`	AX-Adapter, Bool Bit 13
Socket (Eingang)	`BIT_14`	AX-Adapter, Bool Bit 14
Socket (Eingang)	`BIT_15`	AX-Adapter, Bool Bit 15
Plug (Ausgang)	`OUT`	AW-Adapter, WORD Output

Funktionsweise

1. **Eingangssignalerfassung**: Jeder der 16 AX-Adapter (`BIT_00` bis `BIT_15`) stellt einen booleschen Wert über seinen Datenkanal (`D1`) bereit. Sobald sich der Wert eines dieser Adapter ändert, sendet dieser ein Ereignis über seinen Ausgang `E1`.
2. **Zusammenführung**: Die Ereignisse aller 16 Adapter werden auf den Eingang `REQ` eines internen Bausteins `ASSEMBLE_WORD_FROM_BOOLS` geführt. Dieser liest die aktuellen Bool-Werte aller 16 Kanäle ein und setzt sie zu einem 16-Bit-Wort (WORD) zusammen.

3. **Synchronisation**: Das fertige Wort wird an einen flankengesteuerten D-Flip-Flop (`E_D_FF_ANY`) übergeben. Der Flip-Flop übernimmt den Wert an seinem D-Eingang, sobald er ein Taktereignis (vom Ausgang `CNF` des Zusammenführungsbausteins) empfängt.
4. **Ausgabe**: Nach erfolgter Übernahme gibt der Flip-Flop einen Takt an seinem Ausgang `EO` aus und legt das gespeicherte Wort an seinem Q-Ausgang an. Dieses Ereignis triggert den AW-Adapter `OUT`, der das WORD auf seinem Datenkanal (`D1`) zur Verfügung stellt.

Technische Besonderheiten

- **Verwendung eines Flip-Flops**: Die Ausgabe erfolgt nur nach einem vollständigen Zyklus der Zusammenführung. Dadurch werden inkonsistente oder flimmernde Daten vermieden – der Ausgang bleibt stabil, bis ein neues Ereignis eintrifft.
- **Kein Zustandsautomat**: Der Baustein ist rein ereignisgesteuert und besitzt keinen eigenen Ablaufzustand; das Verhalten wird durch die internen Bausteine `ASSEMBLE_WORD_FROM_BOOLS` und `E_D_FF_ANY` bestimmt.
- **Adapter-Schnittstelle**: Sowohl die Eingänge als auch der Ausgang sind als Adapter realisiert. Dies ermöglicht eine flexible Wiederverwendung und Kapselung der Signaltypen.

Zustandsübersicht

Der Funktionsblock enthält keinen expliziten Zustandsautomaten. Das interne D-Flip-Flop `E_D_FF_ANY` besitzt zwei Zustände (`Q = 0` oder `Q = 1`), die den aktuellen Datenwert speichern. Der Ausgang des Flip-Flops wird nur bei einer positiven Flanke am Takteingang aktualisiert.

Anwendungsszenarien

- **Bündelung diskreter Signale**: In der Automatisierungstechnik werden oft 16 einzelne digitale Sensoren oder Schalter (z. B. Endschalter, Taster) benötigt. Der Baustein fasst diese zu einem Wort zusammen, das über einen Feldbus oder als Eingang für eine übergeordnete Steuerung dient.
- **Datenvorbereitung für Kommunikation**: Vor der Übertragung über einen Netzwerkadapter (z. B. PROFINET, EtherCAT) müssen mehrere Binärsignale in einem Datenwort gepackt werden.
- **Signalregister**: Der Baustein kann als einfaches 16-Bit-Register verwendet werden, das den aktuellen Zustand aller Eingänge zwischenspeichert und nur bei Änderungen aktualisiert.

Vergleich mit ähnlichen Bausteinen

- **Klassische Bool-zu-WORD-Bausteine**: Standard-Bausteine kombinieren oft Bool-Eingänge direkt zu einem ganzzahligen Typ, jedoch ohne Adapter-Schnittstelle. `ASSEMBLE_AW_FROM_AX` nutzt Adapter, was eine klar getrennte Signal- und Ereignisübertragung ermöglicht und die Wiederverwendung in modularen Architekturen vereinfacht.
- **Adapter-basierte Alternativen**: Ähnliche Bausteine existieren für andere Wortgrößen (z. B. BYTE, DWORD) oder mit zusätzlicher Filtermöglichkeit. Der vorliegende Baustein beschränkt sich auf das Notwendigste und setzt auf einen Flip-Flop-Ausgang für saubere Synchronisation.

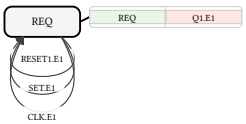
Fazit

Der Funktionsblock `ASSEMBLE_AW_FROM_AX` bietet eine saubere, synchronisierte Möglichkeit, 16 boolesche Signale (über AX-Adapter) zu einem 16-Bit-Wort (über AW-Adapter) zusammenzuführen. Die Kombination aus Zusammenführungssbaustein und flankengesteuertem Flip-Flop gewährleistet eine stabile Ausgabe und eignet sich besonders für den Einsatz in zeitkritischen oder signalverarbeitenden Umgebungen.

2.1.11.5 bistableElements

AX_FB_RS_T_FF

ECC: AX_FB_RS_T_FF



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **AX_FB_RS_T_FF** realisiert ein rücksetzdominantes bistabiles Kippglied (Reset-Dominant Latch) mit zusätzlicher Toggle-Funktion. Der Baustein kommuniziert über standardisierte Adapter-Schnittstellen und ermöglicht eine flexible Anbindung an andere Komponenten.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge/-ausgänge auf FB-Ebene. Die gesamte Kommunikation erfolgt über **Adapter**, die wiederum eigene Ereignis- und Datenkanäle bereitstellen.

Ereignis-Eingänge

Die folgenden Adapter-Ereignisse lösen die Verarbeitung aus:

- **SET.E1** – Ereignis zum Setzen des Ausgangs
- **RESET1.E1** – Ereignis zum Zurücksetzen des Ausgangs (Priorität)
- **CLK.E1** – Ereignis für den Takt, der bei steigender Flanke einen Toggle auslöst

Ereignis-Ausgänge

- **Q1.E1** – Ereignis, das nach jeder Ausführung des Algorithmus gesendet wird (unabhängig vom Ergebnis)

Daten-Eingänge

Über die Adapter werden folgende Datenwerte empfangen:

- **SET.D1** (BOOL) – Setz-Eingang: `TRUE` ⟶ setzt Q1 auf `TRUE`
- **RESET1.D1** (BOOL) – Reset-Eingang: `TRUE` ⟶ setzt Q1 auf `FALSE` (hat Priorität vor SET)
- **CLK.D1** (BOOL) – Takteingang: Flankenwechsel von `FALSE` nach `TRUE` löst Toggle aus

Daten-Ausgänge

- **Q1.D1** (BOOL) – Ausgangswert des Flipflops

Adapter

Adapter	Typ	Rolle	Kommentar
`SET`	adapter::types::unidirectional::AX	Eingang	Setzen
`RESET1`	adapter::types::unidirectional::AX	Eingang	Zurücksetzen
`CLK`	adapter::types::unidirectional::AX	Eingang	Takt
`Q1`	adapter::types::unidirectional::AX	Ausgang	Ausgabe

Alle Adapter sind vom Typ `AX` (unidirektional) und stellen jeweils einen Ereigniskanal (`.E1`) sowie einen Datenkanal (`.D1`) zur Verfügung.

Funktionsweise

Der Baustein arbeitet nach folgendem deterministischen Algorithmus, der bei jedem eingehenden Ereignis (SET, RESET1 oder CLK) ausgeführt wird:

1. **Priorität: Reset dominierend**
   Ist `RESET1.D1 = TRUE`, wird `Q1.D1` auf `FALSE` gesetzt.
2. **Andernfalls: Set Vorrang**
   Ist nur `SET.D1 = TRUE` (und `RESET1.D1 = FALSE`), wird `Q1.D1` auf `TRUE` gesetzt.
3. **Sonst: Toggle bei steigender Flanke**
   Liegen weder RESET1 noch SET an, wird geprüft, ob `CLK.D1 = TRUE` und im vorherigen Zyklus `FALSE` war (steigende Flanke). In diesem Fall wird der Ausgang `Q1` invertiert (`NOT Q1.D1`).
4. **Flankenerkennung**
   Die interne Variable `EDGE` speichert den letzten Wert von `CLK.D1`. Nach jedem Durchlauf wird sie aktualisiert (`EDGE := CLK.D1`). Dadurch wird die steigende Flanke erkannt, sobald `CLK.D1` von `FALSE` auf `TRUE` wechselt.

Nach der Ausführung des Algorithmus wird stets das Ereignis `Q1.E1` gesendet.

Technische Besonderheiten

- **Adapterbasierte Kommunikation**: Der Baustein verwendet ausschließlich Adapter für Ein- und Ausgabe, was eine lose Kopplung und Wiederverwendbarkeit in unterschiedlichen Kontexten erleichtert.
- **Rücksetzdominanz**: Der Reset-Eingang hat Vorrang vor dem Set-Eingang. Dies entspricht dem typischen Verhalten eines RS-Flipflops mit Priorität für den Reset.
- **Flankengetriggerter Toggle**: Das Toggle erfolgt nur bei einer steigenden Flanke des Taktsignals, nicht bei statischem High-Pegel. Die interne Flankenerkennung verhindert mehrfaches Umschalten bei anhaltendem `TRUE` am Takt.
- **Ein-Zustands-ECC**: Der Funktionsbaustein besitzt nur einen Zustand (`REQ`). Alle Transitionen führen zurück zu diesem Zustand, was einer kontinuierlichen Verarbeitung jedes eingehenden Ereignisses ohne zusätzliche Zustandslogik entspricht.

Zustandsübersicht

Der Baustein ist als **Basic FB** mit folgendem ECC realisiert:

Zustand	Aktion	Ausgabe
`REQ`	Führe Algorithmus `REQ` aus	`Q1.E1` (nach Algorithmus)

Übergänge:

- Von `REQ` nach `REQ` bei **RESET1.E1**
- Von `REQ` nach `REQ` bei **SET.E1**
- Von `REQ` nach `REQ` bei **CLK.E1**

Es gibt keine weiteren Zustände. Der Algorithmus wird immer im gleichen Kontext ausgeführt.

Anwendungsszenarien

- **Sicherheitssteuerungen**: Ein rücksetzdominantes Verhalten ist nützlich, wenn ein Fehlerzustand (Reset) höchste Priorität haben muss, z.B. in Not-Aus-Schaltungen.
- **Toggle-Funktion mit Takt**: Umschalten zwischen zwei Zuständen bei jedem Taktimpuls, z.B. für Blinklicht oder Impulszähler.
- **Hybride Schaltungen**: Kombination aus Set/Reset und Umschaltfunktion in einer Komponente, was Platz und Logik spart.

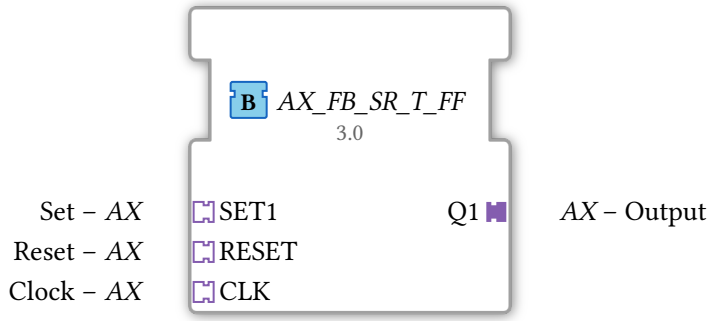
Vergleich mit ähnlichen Bausteinen

- **RS-Flipflop (reset-dominant)**: Verhält sich wie ein herkömmliches RS-Flipflop mit Reset-Vorrang, bietet aber keinen Toggle.
- **T-Flipflop**: Kann nur toggeln, besitzt keine separaten Set-/Reset-Eingänge.
- **JK-Flipflop**: Bietet Set, Reset und Toggle, jedoch mit anderer Prioritätslogik (kein expliziter Reset-Dominanz).
- **AX_FB_RS_T_FF** vereint Reset-Dominanz und Flanken-getriggerten Toggle in einem Baustein, nutzt Adapter für plattformunabhängige Anbindung.

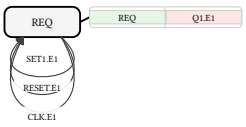
Fazit

Der **AX_FB_RS_T_FF** ist ein vielseitiger, adapterbasierter Funktionsbaustein für bistabile Schaltungen mit rücksetzdominanter Logik und integrierter Toggle-Funktion. Durch die Verwendung von Adaptern lässt er sich einfach in modulare Automatisierungssysteme integrieren. Die Flankenerkennung und die klare Priorität der Eingänge machen ihn robust und zuverlässig für typische Steuerungsaufgaben.

AX_FB_SR_T_FF



ECC: AX_FB_SR_T_FF



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AX_FB_SR_T_FF** realisiert ein bistabiles, setzdominantes Latch (SR-Flipflop) kombiniert mit einer Toggle-Funktion. Die gesamte Kommunikation erfolgt über Adapter-Schnittstellen, die sowohl Ereignis- als auch Datensignale bündeln.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge/-ausgänge, sondern ausschließlich Adapter. Diese stellen jeweils ein Ereignis (`E1`) und einen Datenwert (`D1`) vom Typ `BOOL` bereit.

Ereignis-Eingänge

Keine direkten Ereigniseingänge. Die folgenden Adapter stellen über ihr Ereignis `E1` den Triggermechanismus bereit:

- **SET1** – Setzereignis
- **RESET** – Rücksetzereignis
- **CLK** – Taktereignis (für Toggle)

Ereignis-Ausgänge

Keine direkten Ereignisausgänge. Der Adapter **Q1** stellt über sein Ereignis `E1` die Bestätigung der Ausgangsänderung bereit.

Daten-Eingänge

Die drei Eingangsadapter liefern jeweils einen Datenwert `D1` (BOOL):

- **SET1.D1** – Setzsignal (aktiv bei TRUE)
- **RESET.D1** – Rücksetzsignal (aktiv bei TRUE)
- **CLK.D1** – Taktsignal (positive Flanke löst Toggle aus)

Daten-Ausgänge

Der Ausgangsadapter **Q1** gibt über **Q1.D1** den aktuellen Zustand des Latch als BOOL-Wert aus.

Adapter

Adapter	Typ	Richtung	Beschreibung
SET1	adapter::types::unidirectional::AX	Socket (Eingang)	Setz-Eingang
RESET	adapter::types::unidirectional::AX	Socket (Eingang)	Rücksetz-Eingang
CLK	adapter::types::unidirectional::AX	Socket (Eingang)	Takt-Eingang für Toggle
Q1	adapter::types::unidirectional::AX	Plug (Ausgang)	Ausgang des Latch

Funktionsweise

Der Baustein arbeitet nach folgendem prioritätsgesteuerten Algorithmus:

1. **Setzdominanz**: Wenn `SET1.D1 = TRUE` ist, wird der Ausgang `Q1.D1` auf `TRUE` gesetzt.
2. **Rücksetzen**: Sonst, wenn `RESET.D1 = TRUE` ist, wird `Q1.D1` auf `FALSE` gesetzt.
3. **Toggle**: Falls weder Setzen noch Rücksetzen aktiv sind und am Takteingang `CLK.D1` eine **steigende Flanke** (Übergang von FALSE auf TRUE) erkannt wird, wird der aktuelle Ausgangswert umgeschaltet (`Q1.D1 := NOT Q1.D1`).

Die Flankenerkennung erfolgt durch die interne Variable `EDGE`: Sie speichert den vorherigen Wert von `CLK.D1`. Der Algorithmus prüft, ob `CLK.D1 = TRUE` und `EDGE = FALSE` ist. Nach der Berechnung wird `EDGE` auf den aktuellen `CLK.D1` gesetzt.

Technische Besonderheiten

- **Reine Adapter-Schnittstelle**: Der Baustein nutzt keine klassischen Ereignis-/Datenports, sondern kapselt die Signale in Adaptern. Dies ermöglicht eine flexible Wiederverwendung und Kapselung von Schnittstellen.
- **Set-Dominant-Verhalten**: Ein gleichzeitiges Setzen und Rücksetzen führt immer zum Setzen des Ausgangs.
- **Toggle mit positiver Flanke**: Die Toggle-Funktion reagiert nur auf steigende Flanken des Taktsignals, nicht auf statische Pegel.
- **Einziger ECC-Zustand**: Das ECC besteht nur aus dem Zustand `REQ`. Jedes eingehende Ereignis (`SET1.E1`, `RESET.E1` oder `CLK.E1`) führt sofort zur erneuten Ausführung des Algorithmus.

Zustandsübersicht

Der Funktionsblock besitzt nur einen Zustand:

- **REQ** – Wartet auf eingehende Ereignisse. Bei jedem Ereignis wird der Algorithmus `REQ` durchlaufen. Alle Transitionen führen zurück zu `REQ`.

Intern wird die Variable `EDGE` zur Flankenerkennung verwendet; sie ist Teil des internen Zustands, aber nicht als ECC-Zustand sichtbar.

Anwendungsszenarien

- **Steuerung mit Vorrang**: Wenn ein Setzimpuls immer Vorrang vor einem Rücksetzimpuls haben soll, z. B. bei Sicherheitsschaltungen.
- **Kombinierte Set/Reset- und Toggle-Funktion**: Für Anwendungen, bei denen ein Ausgang sowohl durch externe Signale gesetzt/rückgesetzt als auch durch einen Taster umgeschaltet werden kann (z. B. manuelle Bedienung mit Vorrang).
- **Adapter-basierte Systeme**: In einer modularen Umgebung, in der Schnittstellen über Adapter standardisiert sind (z. B. AX-Adaptersystem).

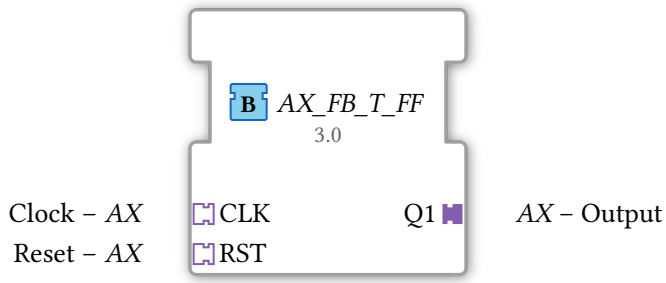
Vergleich mit ähnlichen Bausteinen

- **SR-Flipflop (setzdominant)** – reine Set/Reset-Funktion ohne Toggle. Der vorliegende Baustein erweitert dies um die Toggle-Möglichkeit.
- **T-Flipflop** – nur Toggle (kein Setzen/Rücksetzen). AX_FB_SR_T_FF bietet zusätzliche Setz- und Rücksetzpriorität.
- **Resetdominante SR-Flipflops** – bei Konflikten wird zurückgesetzt. Dieser Baustein verhält sich gegenteilig.

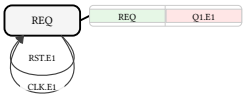
Fazit

Der **AX_FB_SR_T_FF** vereint ein setzdominantes SR-Latch mit einer flankengetriggerten Toggle-Funktion in einem kompakten Baustein. Die ausschließliche Verwendung von Adaptern macht ihn ideal für modulare, adapterbasierte Automatisierungslösungen. Die klare Priorisierung und die Flankenerkennung bieten ein vorhersagbares und robustes Schaltverhalten.

AX_FB_T_FF



ECC: AX_FB_T_FF



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AX_FB_T_FF** realisiert ein taktflankengesteuertes Toggle-Flipflop (T-FF). Er schaltet seinen Ausgangszustand bei jeder aktiven Taktflanke um und kann asynchron zurückgesetzt werden. Die Ein- und Ausgänge erfolgen ausschließlich über Adapter des Typs `AX`, die sowohl ein Ereignis als auch einen booleschen Datenwert übertragen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Datenschnittstellen. Die gesamte Kommunikation erfolgt über **Adapter** (Plugs und Sockets).

Ereignis-Eingänge

Keine (Ereignisse werden über die Adapter `CLK` und `RST` empfangen).

Ereignis-Ausgänge

Keine (Ereignisse werden über den Adapter `Q1` gesendet).

Daten-Eingänge

Keine (Daten werden über die Adapter `CLK` und `RST` empfangen).

Daten-Ausgänge

Keine (Daten werden über den Adapter `Q1` gesendet).

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	`CLK`	`adapter::types::unidirectional::AX`	Taktsignal – bei jedem eingehenden Ereignis (E1) wird der Bool-Wert (D1) als Taktpegel ausgewertet.
Socket (Eingang)	`RST`	`adapter::types::unidirectional::AX`	Reset – bei einem eingehenden Ereignis (E1) wird der Ausgang unabhängig vom Takt auf FALSE gesetzt (asynchroner Reset).
Plug (Ausgang)	`Q1`	`adapter::types::unidirectional::AX`	Ausgang – bei jedem Taktwechsel oder Reset wird ein Ereignis (E1) ausgelöst und der aktuelle Bool-Wert (D1) gesendet.

Die Adapter vom Typ `unidirectional::AX` besitzen intern:

- Ein Ereignis (E1)
- Ein Datum (D1) vom Typ `BOOL`

Funktionsweise

Der Baustein arbeitet als **Toggle-Flipflop mit positiver Flankenerkennung**. Er besitzt einen internen Speicher `EDGE`, der den letzten Taktpegel speichert.

Algorithmus (im Zustand REQ):

1. Wenn ein Reset-Ereignis an `RST.E1` eintrifft:
   - Ausgang `Q1.D1` wird auf `FALSE` gesetzt.
   - Ausgangsereignis `Q1.E1` wird ausgelöst.
2. Andernfalls, wenn ein Takt-Ereignis an `CLK.E1` eintrifft:
   - Prüfen, ob der aktuelle Taktpegel `CLK.D1 = TRUE` und der vorherige Pegel `EDGE = FALSE` ist (d.h. positive Flanke).
   - Wenn ja: `Q1.D1` wird invertiert (`NOT Q1.D1`).
   - Unabhängig von der Flanke wird `EDGE := CLK.D1` gesetzt (Pegelmerker).
   - Ausgangsereignis `Q1.E1` wird ausgelöst.
3. Bleibt der Ausgang ohne Änderung, wenn `RST.E1` oder `CLK.E1` nicht eintreffen, wird kein Ereignis gesendet.

Technische Besonderheiten

- **Adapter basierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter, was die modulare Kopplung mit anderen Bausteinen desselben Typs erleichtert.
- **Flankenerkennung**: Die interne Variable `EDGE` realisiert eine einfache positive Flankenerkennung. Ein Tastwechsel von 0⟶1 löst das Toggeln aus; ein konstanter High-Pegel löst nur einmal aus.
- **Asynchroner Reset**: Der Reset hat Vorrang vor der Taktflanke – er setzt den Ausgang sofort zurück, auch wenn gleichzeitig ein Takt-Ereignis ansteht.
- **Nur ein ECC-Zustand**: Der gesamte Ablauf spielt sich im Zustand `REQ` ab. Beide Transitionen (`CLK.E1` und `RST.E1`) führen zurück zu diesem Zustand.

Zustandsübersicht

Der ECC besteht aus einem einzigen Zustand `REQ`. Jeder eingehende Ereignisimpuls (über `CLK.E1` oder `RST.E1`) führt zur Ausführung des Algorithmus `REQ` und einem sofortigen Ausgangsereignis auf `Q1.E1`.

aktueller Zustand	eingehendes Ereignis	nächster Zustand	ausgeführte Aktion
REQ	`RST.E1`	REQ	RESET: Q1.D1 = FALSE
REQ	`CLK.E1`	REQ	Toggle bei steigender Flanke und aktualisieren von EDGE

Keine weiteren Zustände oder Verweilzeiten.

Anwendungsszenarien

- **Taktgesteuerter Zustandswechsel**: Umschalten eines Ausgangs bei jeder steigenden Flanke eines Taktsignals (z.B. für Frequenzteiler oder Zähler).
- **Tastenentprellung**: Kombiniert mit einem Entprell-Baustein kann das Flipflop einen stabilen Zustandswechsel bei jedem Tastendruck erzeugen.
- **Signalumschaltung**: Ein- und Ausschalten eines Signals durch wiederholte Impulse.
- **Steuerung in Automatisierungssystemen**: Z.B. Wechsel zwischen zwei Betriebsmodi.

Vergleich mit ähnlichen Bausteinen

- **Standard T-FF (z.B. `F_TRIG`/`R_TRIG`)**: Diese Bausteine detektieren nur Flanken, toggeln aber nicht den Ausgang. `AX_FB_T_FF` vereint Flankenerkennung und Toggle-Funktion.
- **SR-Flipflop (Set/Reset)**: Im Gegensatz zum SR-FF hat der T-FF nur einen Reset-Eingang und schaltet bei jedem Takt um, anstatt durch separate Set- und Reset-Signale gesteuert zu werden.
- **Adapterbasierte Varianten**: Andere T-FF in 4diac nutzen oft direkte Ein-/Ausgänge. Dieser Baustein ermöglicht eine lose Kopplung über Adapter, was die Wiederverwendbarkeit erhöht.

Fazit

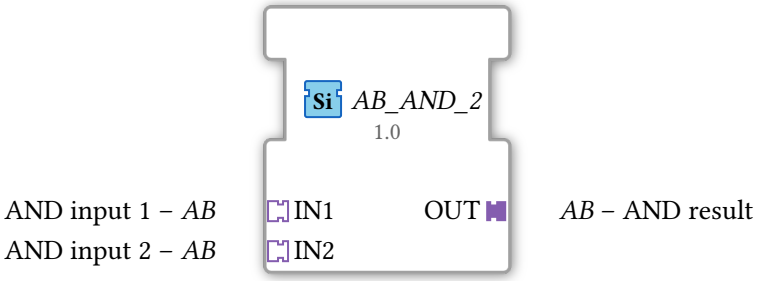
Der `AX_FB_T_FF` ist ein kompakter und flexibler Funktionsbaustein, der ein Toggle-Flipflop mit positiver Flankenerkennung und asynchronem Reset realisiert. Durch die reine Adapter-Schnittstelle lässt er sich besonders gut in modularen, adapterbasierten Entwürfen einsetzen, bei denen Daten- und Ereignisübertragung über einen einheitlichen Kanal erfolgen. Die einfache Logik und der minimale Zustandsautomat machen ihn zuverlässig und leicht verständlich.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.1.11.6 bitwiseOperators

AB_AND_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_AND_2** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 2 Eingangswerten vom Typ `BYTE` (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_AND`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 2 Eingangsadapter (`IN1` … `IN2`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_AND_2 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_AND_2`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_AND_2` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet `AB_AND_2` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AB_AND_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

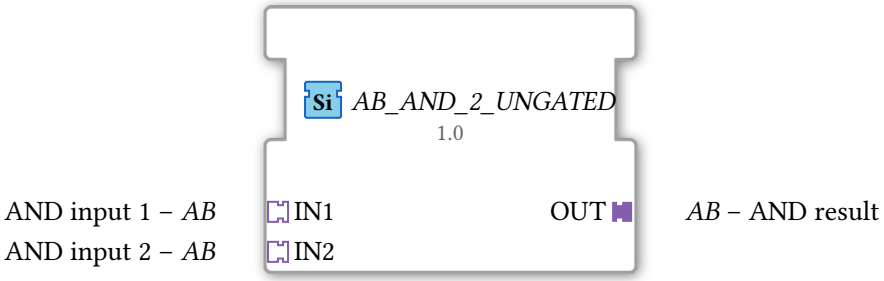
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AB_AND_2** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_AND_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AB_AND_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_AND_2_UNGATED** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 2 Eingangswerten vom Typ `BYTE` (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_AND`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 2 Eingangsadapter (`IN1` … `IN2`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_AND_2_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_AND_2`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_AND_2_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet `AB_AND_2_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AB_AND_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

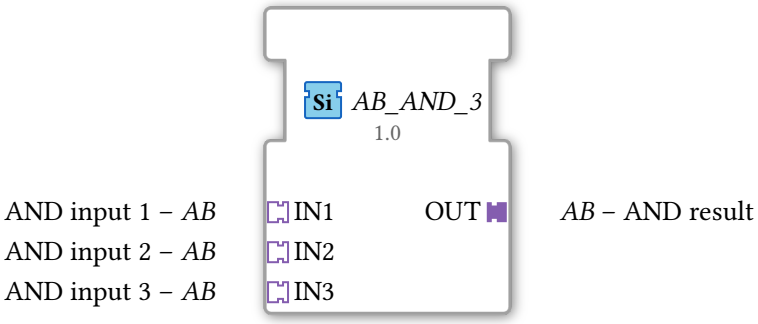
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AB_AND_2_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_AND_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_AND_3** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 3 Eingangswerten vom Typ `BYTE` (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_AND`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: UND-Eingang 3 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` … `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.

- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_AND_3 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_AND_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_AND_3` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet `AB_AND_3` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AB_AND_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

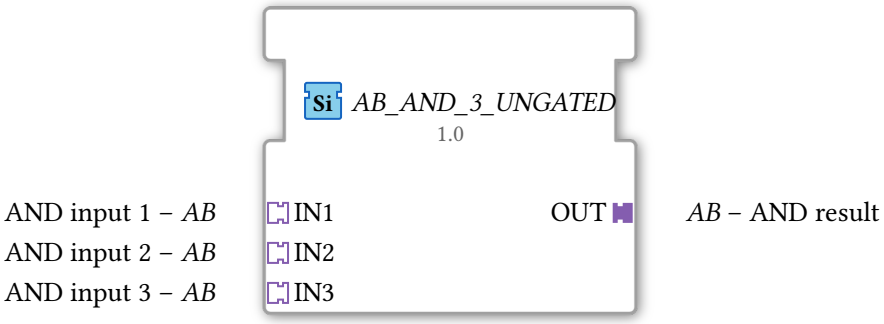
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AB_AND_3** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_AND_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AB_AND_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_AND_3_UNGATED** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 3 Eingangswerten vom Typ `BYTE` (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_AND`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: UND-Eingang 3 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` … `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_AND_3_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.

- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_AND_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_AND_3_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet `AB_AND_3_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AB_AND_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

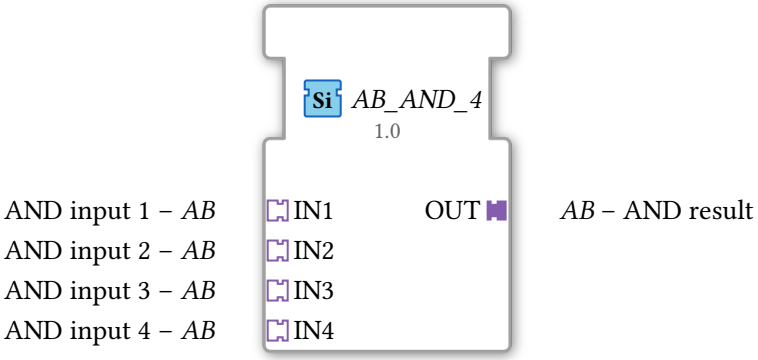
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AB_AND_3_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_AND_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_AND_4** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 4 Eingangswerten vom Typ `BYTE` (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_AND`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: UND-Eingang 3 (Typ: adapter::types::unidirectional::AB)
- **IN4**: UND-Eingang 4 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 4 Eingangsadapter (`IN1` … `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_AND_4 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_AND_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_AND_4` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet `AB_AND_4` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AB_AND_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

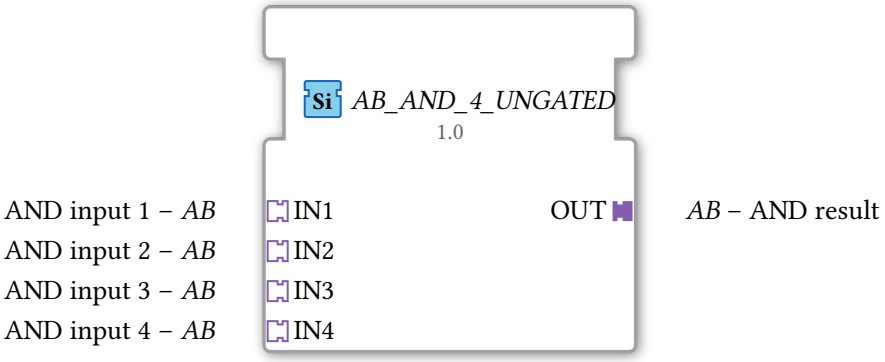
Fazit

Der **AB_AND_4** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar

in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_AND_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AB_AND_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_AND_4_UNGATED** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 4 Eingangswerten vom Typ `BYTE` (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_AND`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: UND-Eingang 3 (Typ: adapter::types::unidirectional::AB)
- **IN4**: UND-Eingang 4 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 4 Eingangsadapter (`IN1` … `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_AND_4_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_AND_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_AND_4_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet `AB_AND_4_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AB_AND_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

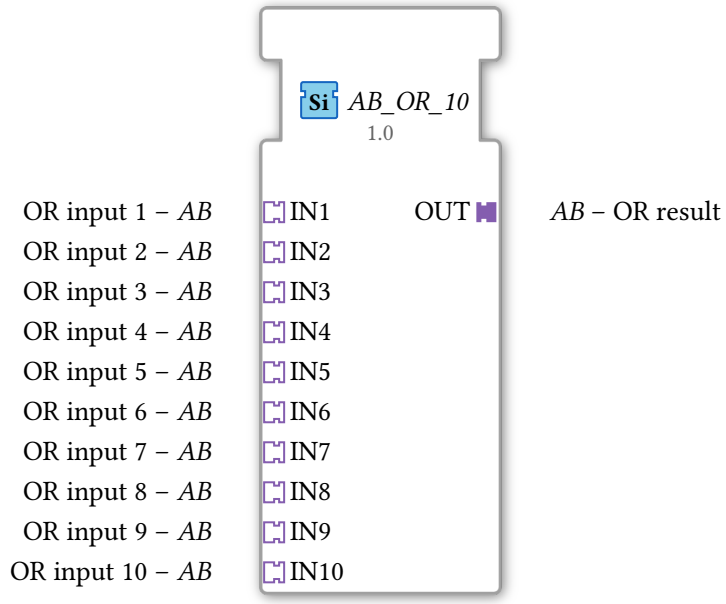
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AB_AND_4_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_OR_10



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_OR_10** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 10 Eingangswerten vom Typ `BYTE` (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AB)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AB)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AB)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AB)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AB)
- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AB)
- **IN9**: ODER-Eingang 9 (Typ: adapter::types::unidirectional::AB)
- **IN10**: ODER-Eingang 10 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 10 Eingangsadapter (`IN1` … `IN10`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 10 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_OR_10 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_10`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_OR_10` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AB_OR_10` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AB_OR_10_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

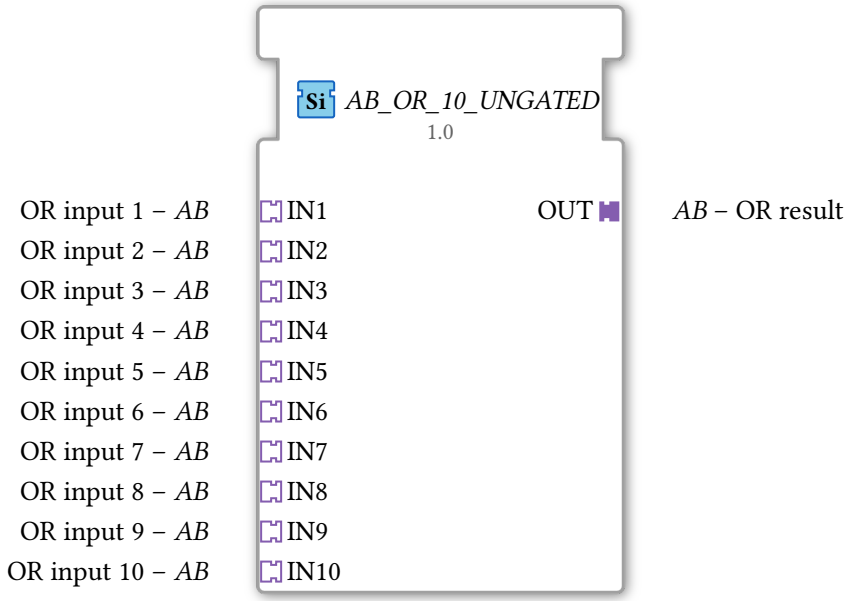
Fazit

Der **AB_OR_10** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_OR_10_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AB_OR_10`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht

gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_OR_10_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 10 Eingangswerten vom Typ `BYTE` (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AB)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AB)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AB)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AB)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AB)
- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AB)
- **IN9**: ODER-Eingang 9 (Typ: adapter::types::unidirectional::AB)
- **IN10**: ODER-Eingang 10 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 10 Eingangsadapter (`IN1` … `IN10`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 10 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_OR_10_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_10`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_OR_10_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AB_OR_10_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AB_OR_10`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

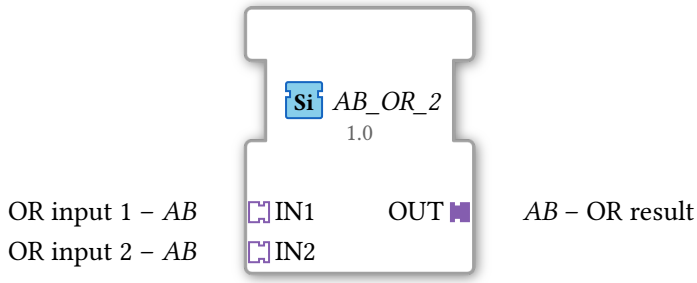
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AB_OR_10_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_OR_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_OR_2** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 2 Eingangswerten vom Typ `BYTE` (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 2 Eingangsadapter (`IN1` ... `IN2`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_OR_2 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_2`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_OR_2` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AB_OR_2` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AB_OR_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

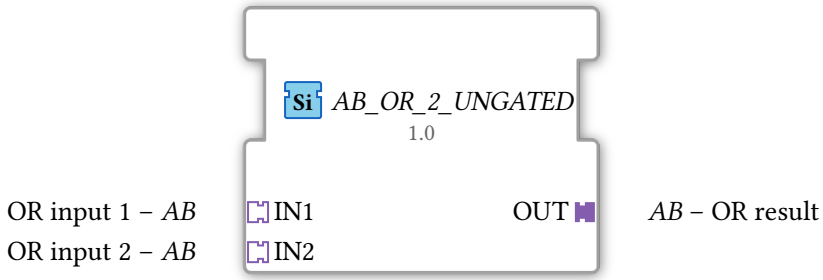
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AB_OR_2** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_OR_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AB_OR_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_OR_2_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 2 Eingangswerten vom Typ `BYTE` (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung

einzelner Wahrheitswerte (wie bei den AX_OR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise
Sobald an einem der 2 Eingangsadapter (IN1 … IN2) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter OUT. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf OUT gehaltenen Wert unterscheidet, wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (GEN_AB_OR) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen AX_OR-Bausteinen wird hier jedes Bit des BYTE-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ unidirectional::AB – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_OR_2_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ BYTE zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer BYTE-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen
Im Gegensatz zu AX_OR_2, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet AB_OR_2_UNGATED auf dem vollständigen Bitmuster eines BYTE-Werts. Verglichen mit dem Standard-Baustein OR_2 verwen-

det AB_OR_2_UNGATED Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- AB_OR_2: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

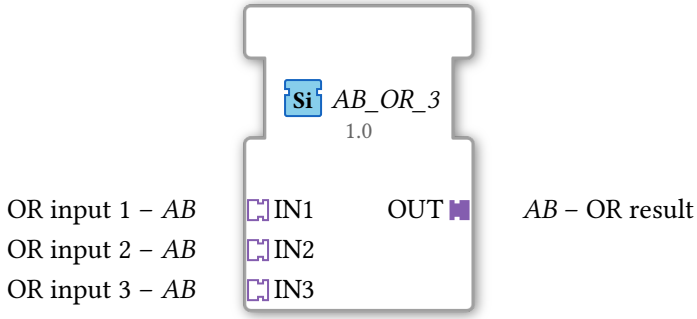
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AB_OR_2_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für BYTE-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_OR_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_OR_3** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 3 Eingangswerten vom Typ BYTE (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_OR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` … `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_OR_3 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_OR_3` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AB_OR_3` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AB_OR_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

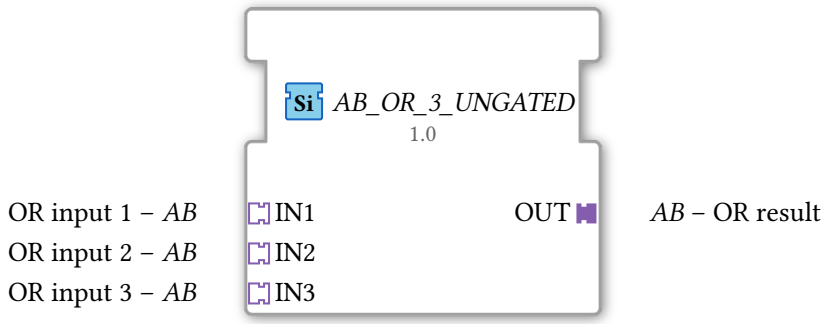
Fazit

Der **AB_OR_3** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_OR_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AB_OR_3. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht

gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_OR_3_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 3 Eingangswerten vom Typ `BYTE` (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` … `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.

- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_OR_3_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_OR_3_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AB_OR_3_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AB_OR_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

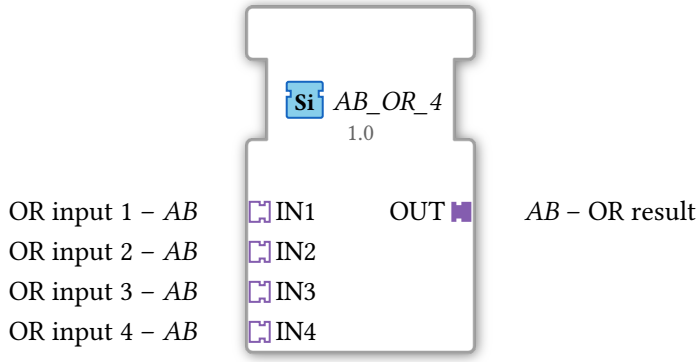
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AB_OR_3_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_OR_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_OR_4** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 4 Eingangswerten vom Typ `BYTE` (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AB)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise
Sobald an einem der 4 Eingangsadapter (`IN1` … `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten
- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_OR_4 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien
- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen
Im Gegensatz zu `AX_OR_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_OR_4` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet

AB_OR_4 Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AB_OR_4_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

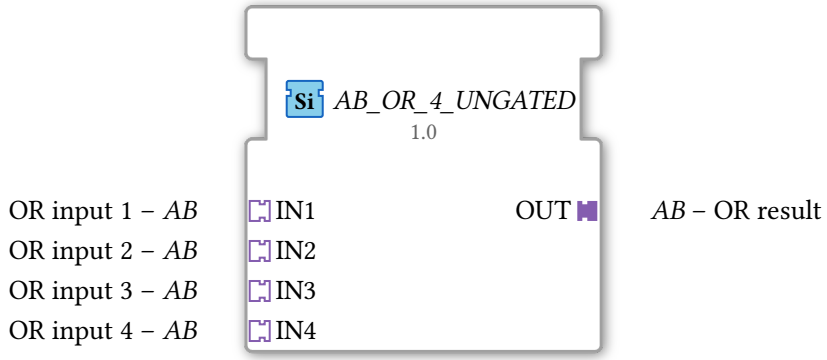
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AB_OR_4** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für BYTE-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_OR_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AB_OR_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_OR_4_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 4 Eingangswerten vom Typ BYTE (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_OR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AB)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 4 Eingangsadapter (`IN1` ... `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_OR_4_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_OR_4_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AB_OR_4_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AB_OR_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

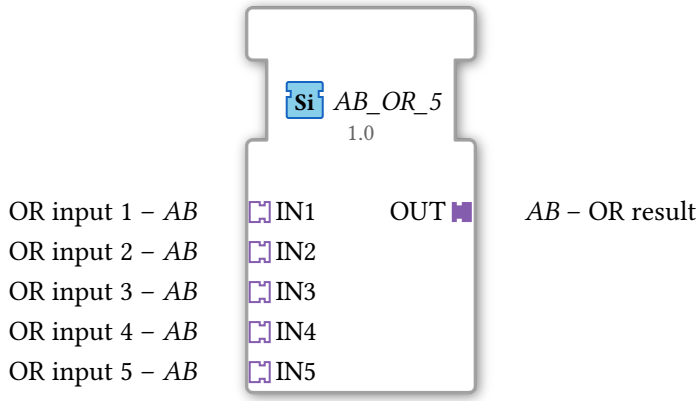
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AB_OR_4_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_OR_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_OR_5** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 5 Eingangswerten vom Typ `BYTE` (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AB)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AB)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 5 Eingangsadapter (`IN1` ... `IN5`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 5 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-

Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf OUT gehaltenen Wert unterscheidet, wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (GEN_AB_OR) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen AX_OR-Bausteinen wird hier jedes Bit des BYTE-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ unidirectional::AB – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_OR_5 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ BYTE zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer BYTE-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu AX_OR_5, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet AB_OR_5 auf dem vollständigen Bitmuster eines BYTE-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet AB_OR_5 Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- AB_OR_5_UNGATED: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

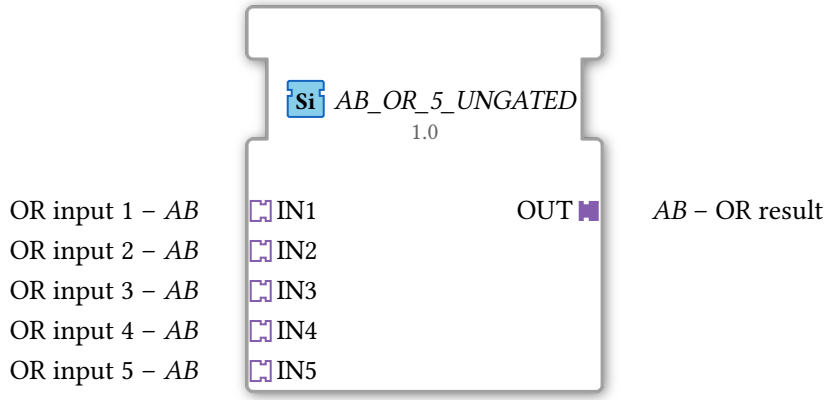
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AB_OR_5** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für BYTE-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_OR_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AB_OR_5. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_OR_5_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 5 Eingangswerten vom Typ `BYTE` (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AB)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AB)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 5 Eingangsadapter (`IN1` … `IN5`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 5 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_OR_5_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_5`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_OR_5_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AB_OR_5_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AB_OR_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

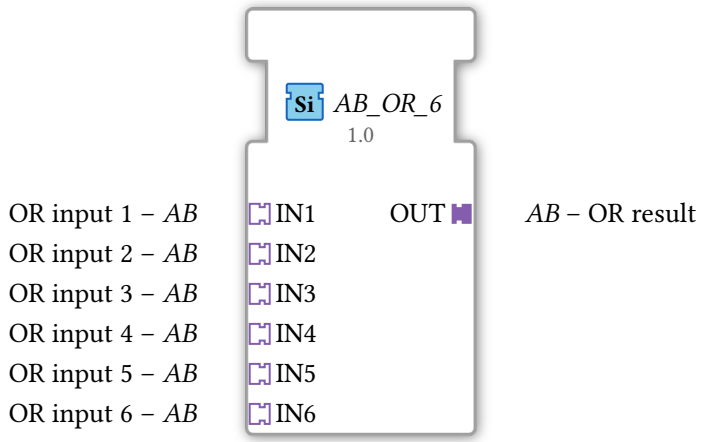
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AB_OR_5_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_OR_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_OR_6** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 6 Eingangswerten vom Typ `BYTE` (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AB)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AB)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AB)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 6 Eingangsadapter (`IN1` … `IN6`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 6 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_OR_6 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.

- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_6`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_OR_6` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AB_OR_6` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AB_OR_6_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

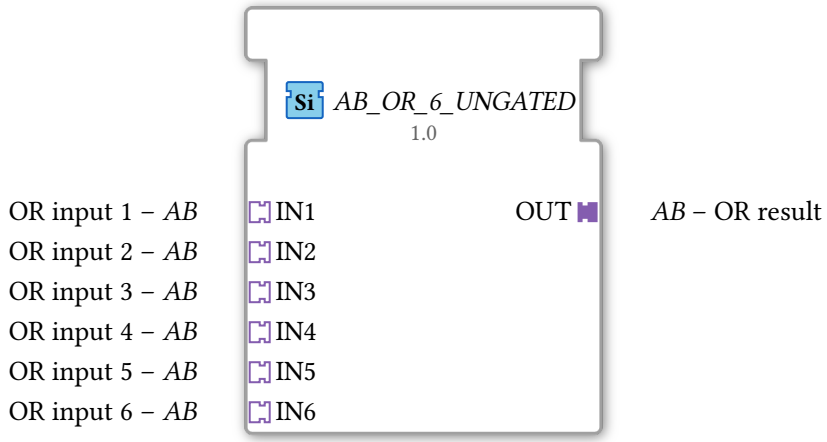
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AB_OR_6** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_OR_6_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AB_OR_6`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_OR_6_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 6 Eingangswerten vom Typ `BYTE` (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AB)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AB)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AB)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise
Sobald an einem der 6 Eingangsadapter (`IN1` … `IN6`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 6 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_OR_6_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu AX_OR_6, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet AB_OR_6_UNGATED auf dem vollständigen Bitmuster eines BYTE-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet AB_OR_6_UNGATED Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AB_OR_6**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

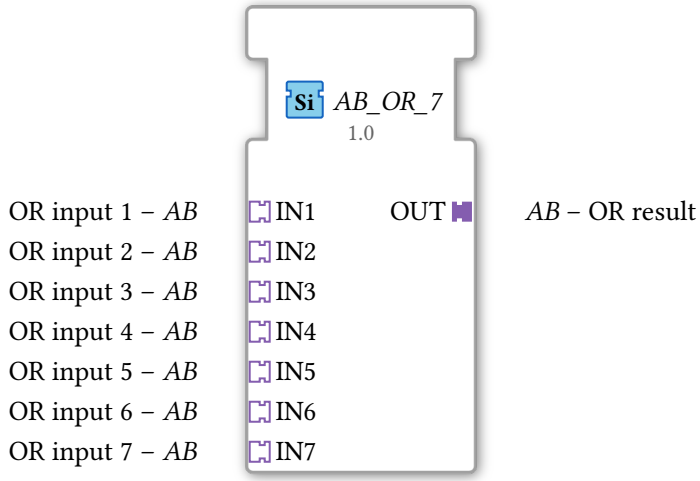
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AB_OR_6_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für BYTE-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_OR_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_OR_7** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 7 Eingangswerten vom Typ BYTE (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_OR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AB)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AB)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AB)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AB)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 7 Eingangsadapter (`IN1` … `IN7`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 7 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_OR_7 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_7`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_OR_7` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AB_OR_7` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AB_OR_7_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

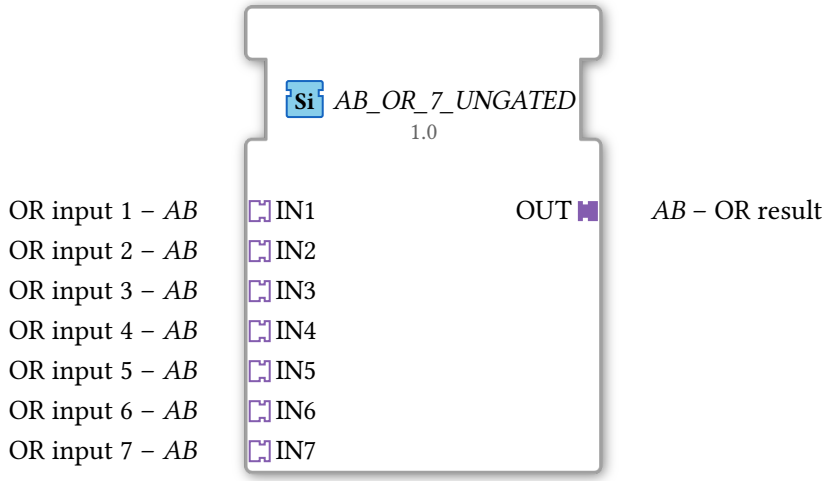
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AB_OR_7** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für BYTE-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_OR_7_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AB_OR_7. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_OR_7_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 7 Eingangswerten vom Typ BYTE (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_OR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AB)

- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AB)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AB)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AB)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AB)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 7 Eingangsadapter (`IN1` ... `IN7`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 7 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_OR_7_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_7`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_OR_7_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AB_OR_7_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AB_OR_7`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

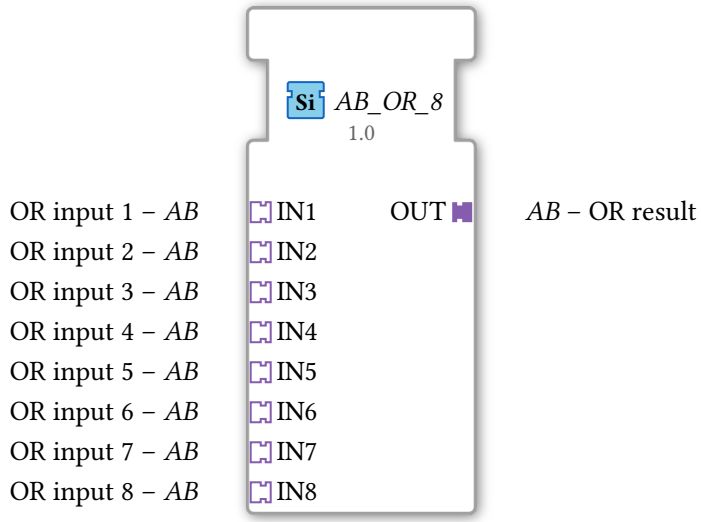
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AB_OR_7_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_OR_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_OR_8** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 8 Eingangswerten vom Typ `BYTE` (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AB)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AB)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AB)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AB)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AB)
- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 8 Eingangsadapter (`IN1` … `IN8`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 8 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`.

Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_OR_8 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_8`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_OR_8` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AB_OR_8` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AB_OR_8_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

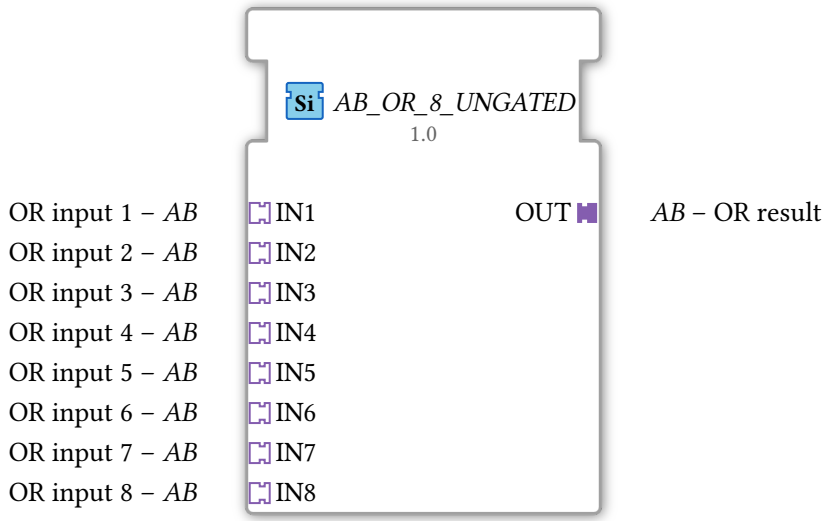
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AB_OR_8** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_OR_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AB_OR_8. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_OR_8_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 8 Eingangswerten vom Typ `BYTE` (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AB)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AB)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AB)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AB)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AB)
- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 8 Eingangsadapter (`IN1` … `IN8`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 8 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-

Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_OR_8_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_8`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_OR_8_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AB_OR_8_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AB_OR_8`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

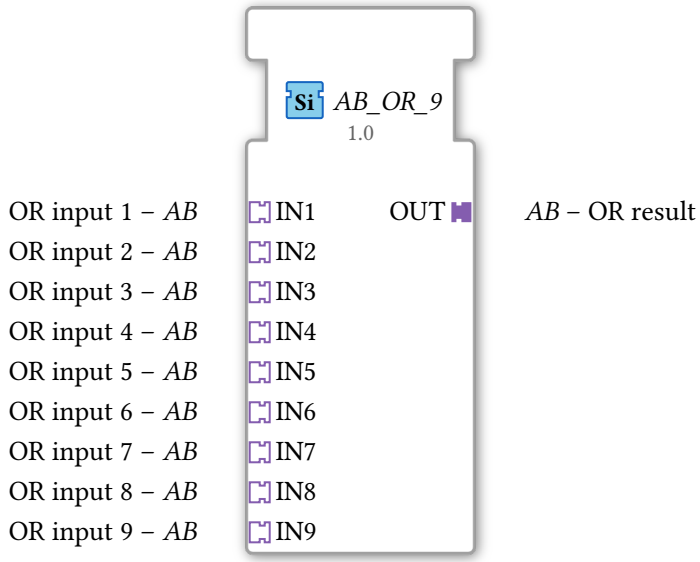
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AB_OR_8_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_OR_9



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_OR_9** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 9 Eingangswerten vom Typ `BYTE` (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AB)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AB)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AB)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AB)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AB)
- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AB)
- **IN9**: ODER-Eingang 9 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 9 Eingangsadapter (`IN1` … `IN9`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 9 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_OR_9 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_9`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_OR_9` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AB_OR_9` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AB_OR_9_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

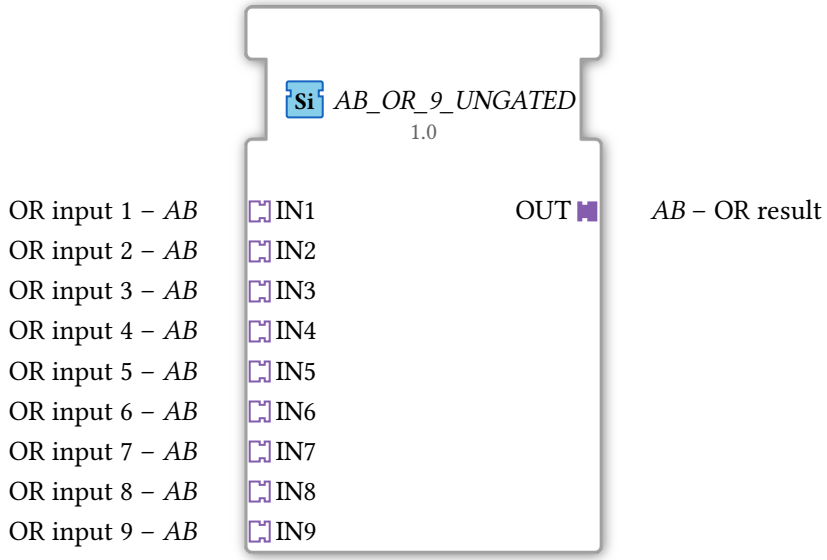
Fazit

Der **AB_OR_9** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_OR_9_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AB_OR_9. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht

gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_OR_9_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 9 Eingangswerten vom Typ `BYTE` (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AB)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AB)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AB)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AB)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AB)
- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AB)
- **IN9**: ODER-Eingang 9 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 9 Eingangsadapter (`IN1` … `IN9`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 9 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_OR_9_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_9`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_OR_9_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AB_OR_9_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AB_OR_9`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

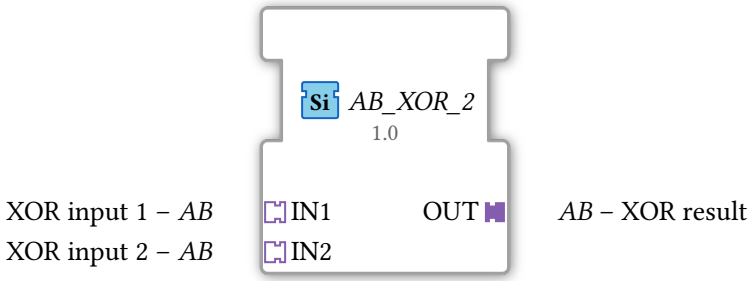
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AB_OR_9_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_XOR_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_XOR_2** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 2 Eingangswerten vom Typ `BYTE` (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 2 Eingangsadapter (`IN1` ... `IN2`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_XOR_2 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_2`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_XOR_2` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AB_XOR_2` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AB_XOR_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

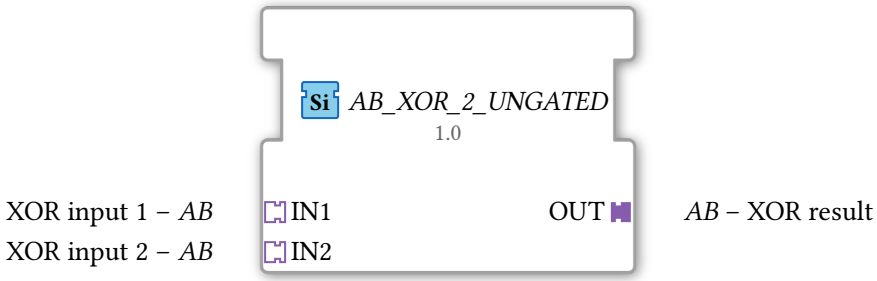
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AB_XOR_2** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_XOR_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AB_XOR_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_XOR_2_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 2 Eingangswerten vom Typ `BYTE` (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen

Verknüpfung einzelner Wahrheitswerte (wie bei den AX_XOR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise
Sobald an einem der 2 Eingangsadapter (IN1 ... IN2) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter OUT. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf OUT gehaltenen Wert unterscheidet, wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (GEN_AB_XOR) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen AX_XOR-Bausteinen wird hier jedes Bit des BYTE-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ unidirectional::AB – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_XOR_2_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ BYTE zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer BYTE-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen
Im Gegensatz zu AX_XOR_2, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet AB_XOR_2_UNGATED auf dem vollständigen Bitmuster eines BYTE-Werts. Verglichen mit dem Standard-Baustein XOR_2

verwendet AB_XOR_2_UNGATED Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AB_XOR_2**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

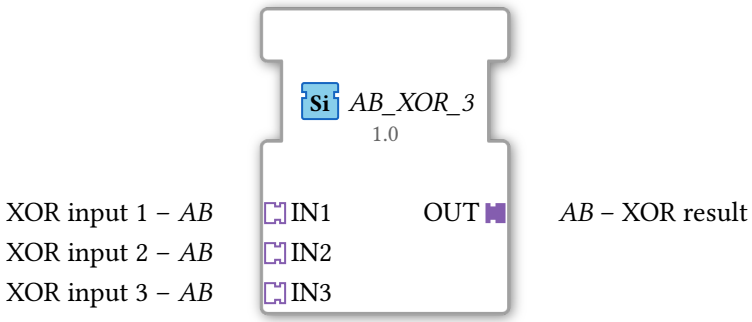
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AB_XOR_2_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für BYTE-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_XOR_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_XOR_3** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 3 Eingangswerten vom Typ BYTE (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_XOR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` … `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_XOR_3 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_XOR_3` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AB_XOR_3` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AB_XOR_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

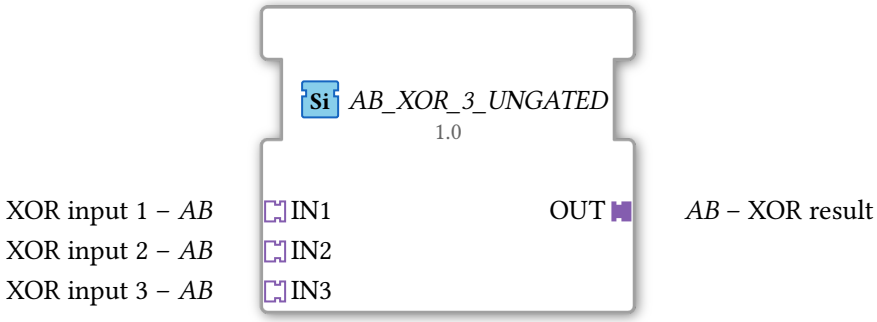
Fazit

Der **AB_XOR_3** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_XOR_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AB_XOR_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht

gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_XOR_3_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 3 Eingangswerten vom Typ `BYTE` (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` … `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.

- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_XOR_3_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_XOR_3_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AB_XOR_3_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AB_XOR_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

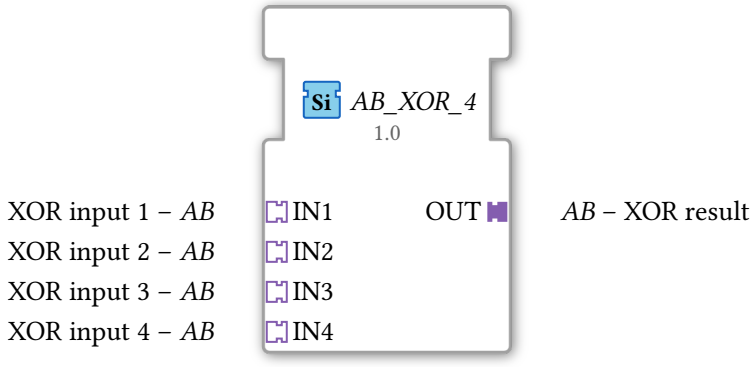
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AB_XOR_3_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_XOR_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_XOR_4** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 4 Eingangswerten vom Typ `BYTE` (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AB)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise
Sobald an einem der 4 Eingangsadapter (`IN1` … `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten
- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_XOR_4 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien
- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen
Im Gegensatz zu `AX_XOR_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_XOR_4` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet

AB_XOR_4 Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AB_XOR_4_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

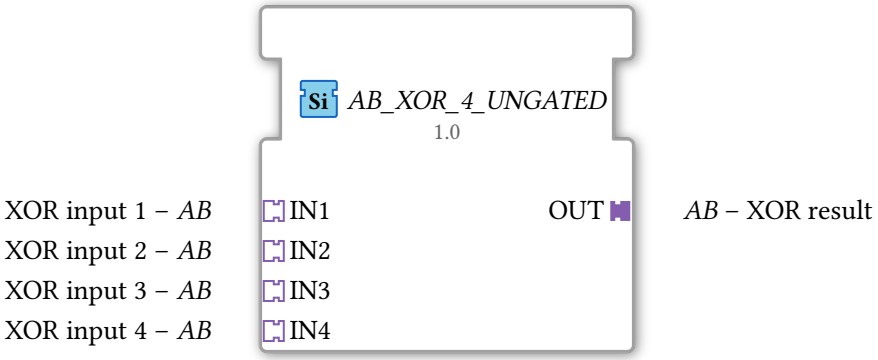
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AB_XOR_4** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für BYTE-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_XOR_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AB_XOR_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_XOR_4_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 4 Eingangswerten vom Typ BYTE (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_XOR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AB)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 4 Eingangsadapter (`IN1` ... `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_XOR_4_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_XOR_4_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AB_XOR_4_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AB_XOR_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

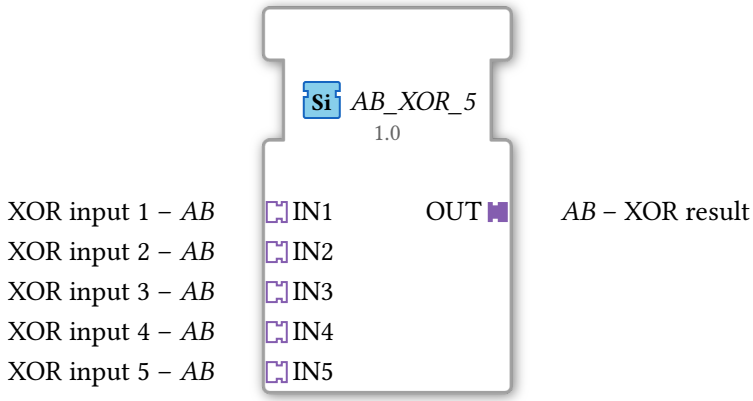
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AB_XOR_4_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_XOR_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_XOR_5** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 5 Eingangswerten vom Typ `BYTE` (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AB)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AB)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 5 Eingangsadapter (`IN1` ... `IN5`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 5 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0`

(Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_XOR_5 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_5`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_XOR_5` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AB_XOR_5` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AB_XOR_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

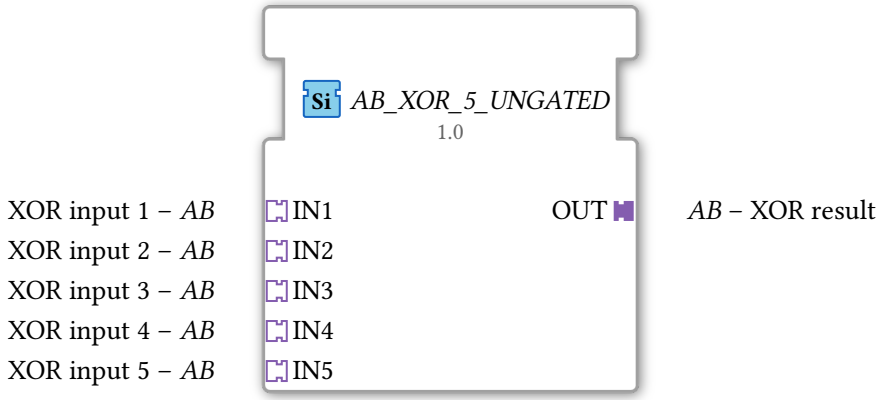
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AB_XOR_5** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_XOR_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AB_XOR_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_XOR_5_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 5 Eingangswerten vom Typ `BYTE` (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AB)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AB)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 5 Eingangsadapter (`IN1` … `IN5`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 5 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_XOR_5_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_5`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_XOR_5_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AB_XOR_5_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AB_XOR_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

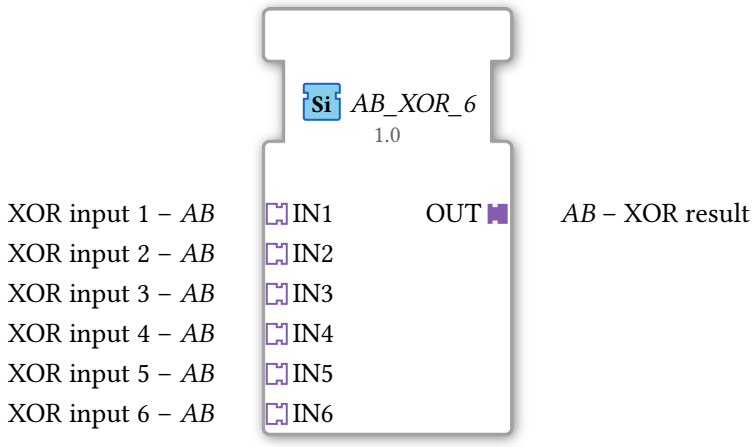
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AB_XOR_5_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_XOR_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_XOR_6** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 6 Eingangswerten vom Typ `BYTE` (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AB)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AB)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AB)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 6 Eingangsadapter (`IN1` … `IN6`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 6 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_XOR_6 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.

- **Signalaggregation**: Zusammenführen mehrerer BYTE-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu AX_XOR_6, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet AB_XOR_6 auf dem vollständigen Bitmuster eines BYTE-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet AB_XOR_6 Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- AB_XOR_6_UNGATED: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

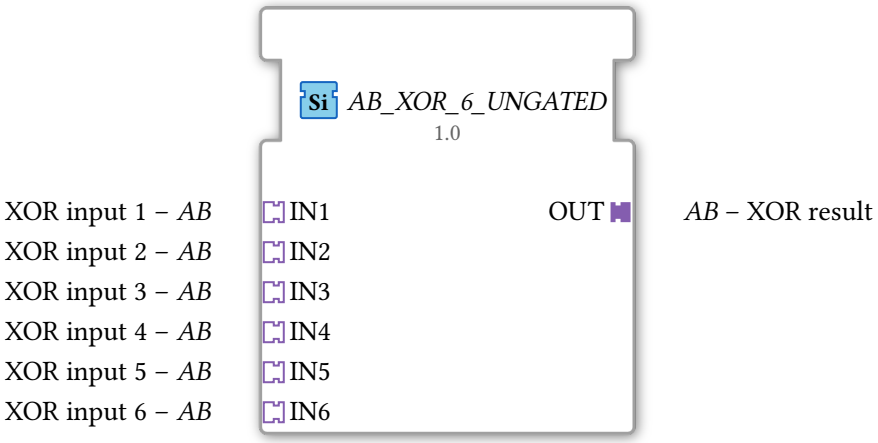
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AB_XOR_6** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für BYTE-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_XOR_6_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AB_XOR_6. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_XOR_6_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 6 Eingangswerten vom Typ BYTE (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_XOR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AB)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AB)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AB)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise
Sobald an einem der 6 Eingangsadapter (`IN1` … `IN6`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 6 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_XOR_6_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu AX_XOR_6, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet AB_XOR_6_UNGATED auf dem vollständigen Bitmuster eines BYTE-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet AB_XOR_6_UNGATED Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AB_XOR_6**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

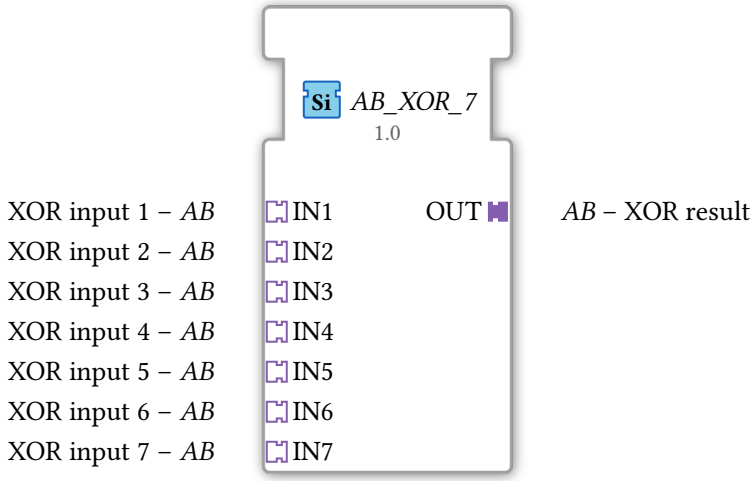
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AB_XOR_6_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für BYTE-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_XOR_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_XOR_7** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 7 Eingangswerten vom Typ BYTE (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_XOR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AB)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AB)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AB)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AB)
- **IN7**: XOR (exklusives ODER)-Eingang 7 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 7 Eingangsadapter (`IN1` ... `IN7`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 7 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_XOR_7 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_7`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_XOR_7` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AB_XOR_7` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AB_XOR_7_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

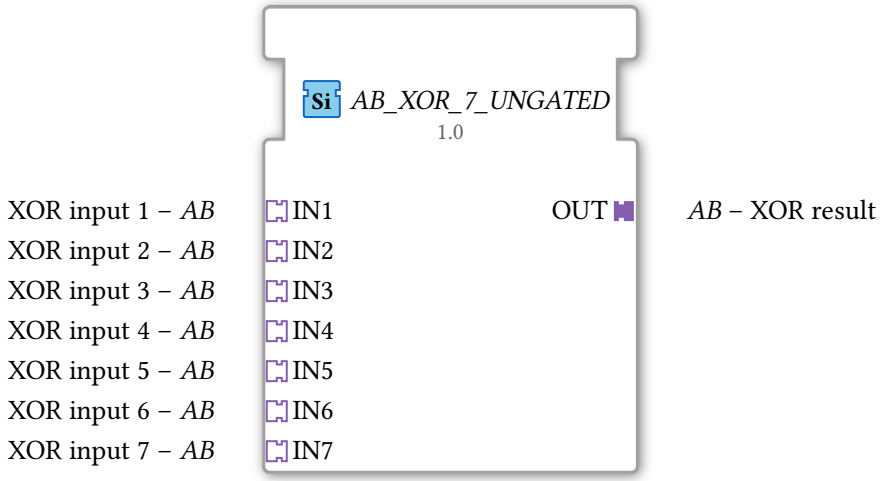
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AB_XOR_7** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für BYTE-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_XOR_7_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AB_XOR_7. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_XOR_7_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 7 Eingangswerten vom Typ BYTE (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_XOR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AB)

- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AB)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AB)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AB)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AB)
- **IN7**: XOR (exklusives ODER)-Eingang 7 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 7 Eingangsadapter (`IN1` ... `IN7`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 7 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_XOR_7_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_7`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_XOR_7_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AB_XOR_7_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AB_XOR_7`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

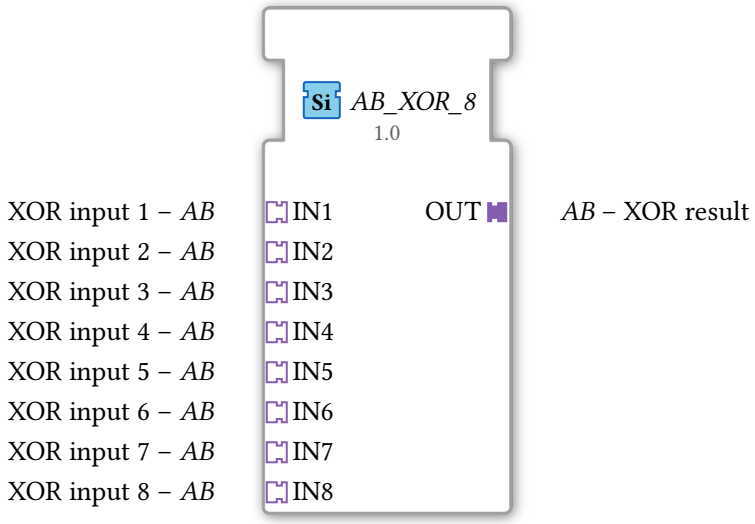
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AB_XOR_7_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_XOR_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_XOR_8** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 8 Eingangswerten vom Typ `BYTE` (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AB)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AB)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AB)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AB)
- **IN7**: XOR (exklusives ODER)-Eingang 7 (Typ: adapter::types::unidirectional::AB)
- **IN8**: XOR (exklusives ODER)-Eingang 8 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 8 Eingangsadapter (`IN1` ... `IN8`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 8 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den

Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_XOR_8 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_8`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_XOR_8` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AB_XOR_8` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AB_XOR_8_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

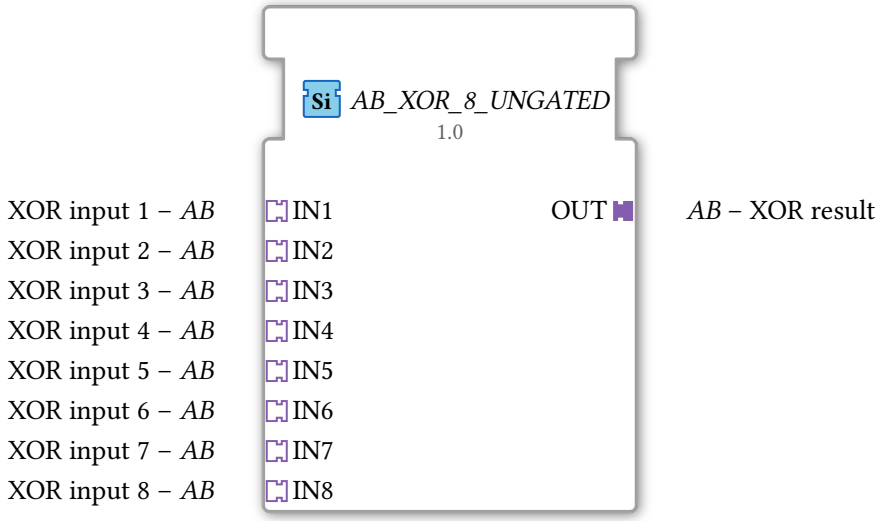
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AB_XOR_8** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AB_XOR_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AB_XOR_8`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_XOR_8_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 8 Eingangswerten vom Typ `BYTE` (8-Bit-Bitmuster (Byte)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AB)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AB)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AB)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AB)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AB)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AB)
- **IN7**: XOR (exklusives ODER)-Eingang 7 (Typ: adapter::types::unidirectional::AB)
- **IN8**: XOR (exklusives ODER)-Eingang 8 (Typ: adapter::types::unidirectional::AB)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AB)

Funktionsweise

Sobald an einem der 8 Eingangsadapter (`IN1` ... `IN8`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 8 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0

(Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AB` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AB_XOR_8_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_8`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AB_XOR_8_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AB_XOR_8_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AB_XOR_8`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

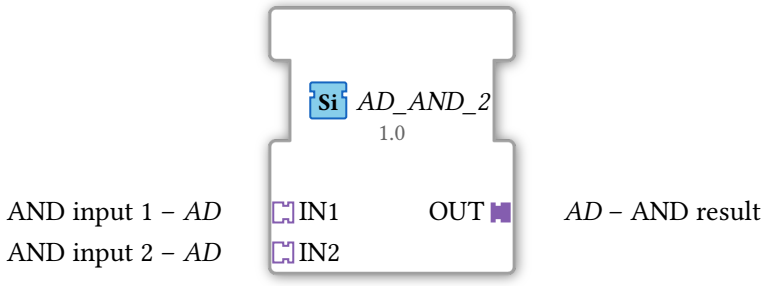
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AB_XOR_8_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_AND_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_AND_2** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 2 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_AND`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 2 Eingangsadapter (`IN1` … `IN2`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_AND_2 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu AX_AND_2, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet AD_AND_2 auf dem vollständigen Bitmuster eines DWORD-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet AD_AND_2 Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AD_AND_2_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

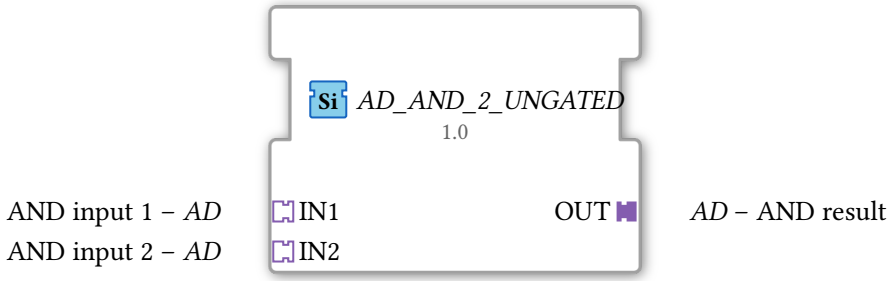
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AD_AND_2** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für DWORD-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_AND_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AD_AND_2. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_AND_2_UNGATED** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 2 Eingangswerten vom Typ DWORD (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_AND-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 2 Eingangsadapter (`IN1` ... `IN2`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_AND_2_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_AND_2`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_AND_2_UNGATED` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet `AD_AND_2_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AD_AND_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

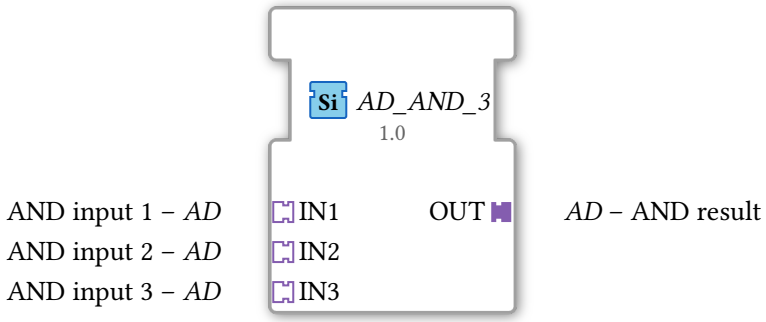
Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AD_AND_2_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn

vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_AND_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_AND_3** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 3 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_AND`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: UND-Eingang 3 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` … `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.

- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_AND_3 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_AND_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_AND_3` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet `AD_AND_3` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AD_AND_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

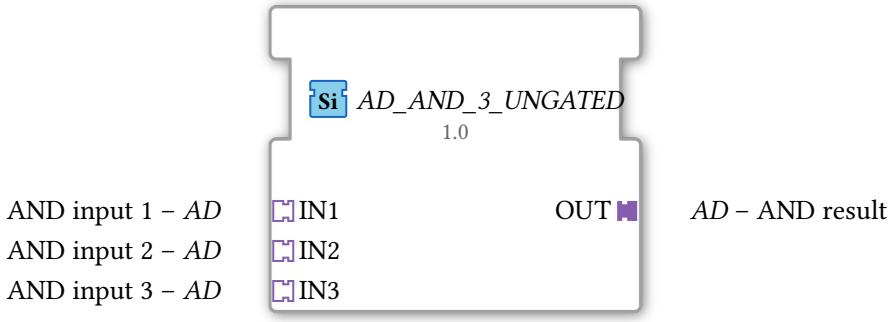
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AD_AND_3** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_AND_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AD_AND_3. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_AND_3_UNGATED** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 3 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_AND`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: UND-Eingang 3 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` … `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.

- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_AND_3_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_AND_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_AND_3_UNGATED` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet `AD_AND_3_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AD_AND_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

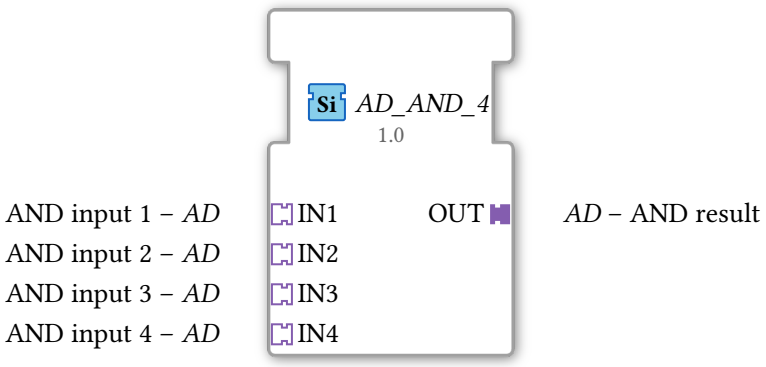
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AD_AND_3_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_AND_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_AND_4** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 4 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_AND`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: UND-Eingang 3 (Typ: adapter::types::unidirectional::AD)
- **IN4**: UND-Eingang 4 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise
Sobald an einem der 4 Eingangsadapter (`IN1` … `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten
- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_AND_4 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien
- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen
Im Gegensatz zu `AX_AND_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_AND_4` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet

AD_AND_4 Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AD_AND_4_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

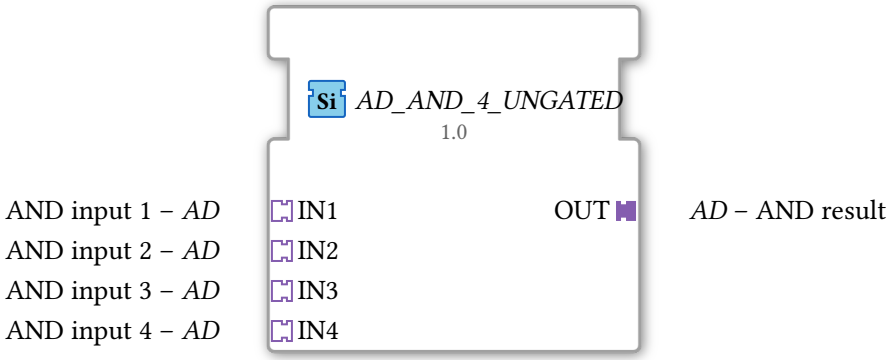
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AD_AND_4** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für DWORD-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_AND_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AD_AND_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_AND_4_UNGATED** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 4 Eingangswerten vom Typ DWORD (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_AND-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: UND-Eingang 3 (Typ: adapter::types::unidirectional::AD)
- **IN4**: UND-Eingang 4 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 4 Eingangsadapter (`IN1` ... `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_AND_4_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_AND_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_AND_4_UNGATED` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet `AD_AND_4_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AD_AND_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

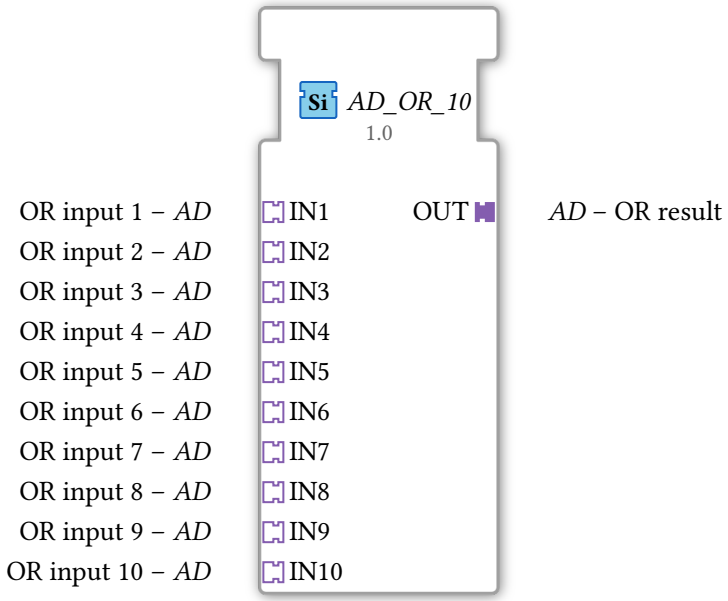
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AD_AND_4_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_OR_10



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_OR_10** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 10 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AD)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AD)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AD)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AD)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AD)

- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AD)
- **IN9**: ODER-Eingang 9 (Typ: adapter::types::unidirectional::AD)
- **IN10**: ODER-Eingang 10 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 10 Eingangsadapter (`IN1` … `IN10`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 10 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_OR_10 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_10`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_OR_10` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AD_OR_10` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AD_OR_10_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

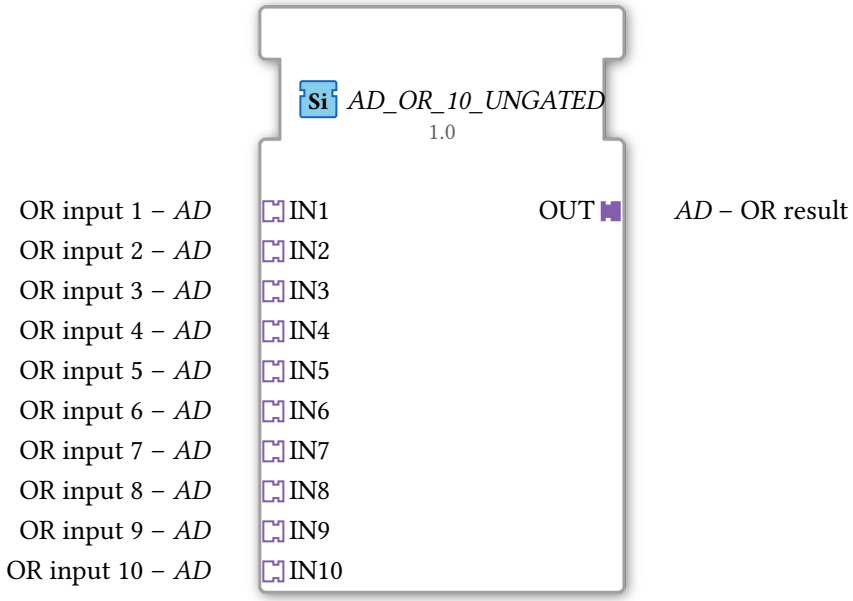
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AD_OR_10** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_OR_10_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `AD_OR_10`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_OR_10_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 10 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AD)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AD)

- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AD)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AD)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AD)
- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AD)
- **IN9**: ODER-Eingang 9 (Typ: adapter::types::unidirectional::AD)
- **IN10**: ODER-Eingang 10 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 10 Eingangsadapter (`IN1` … `IN10`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 10 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_OR_10_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_10`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_OR_10_UNGATED` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AD_OR_10_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AD_OR_10`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

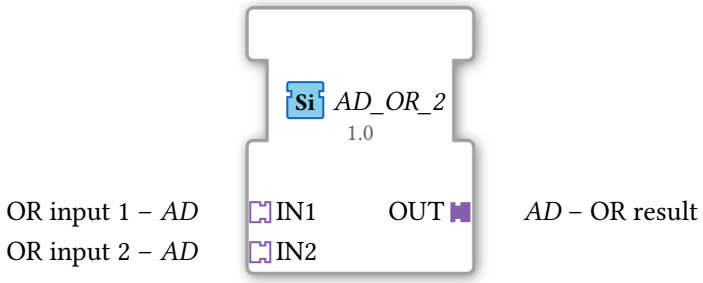
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AD_OR_10_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_OR_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_OR_2** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 2 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 2 Eingangsadapter (`IN1` … `IN2`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_OR_2 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_2`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_OR_2` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AD_OR_2` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AD_OR_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

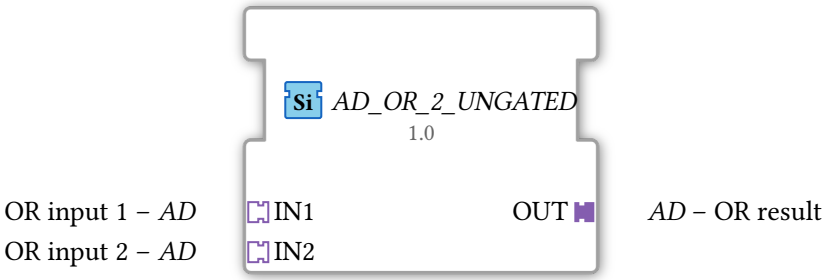
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AD_OR_2** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_OR_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AD_OR_2. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_OR_2_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 2 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Ver-

knüpfung einzelner Wahrheitswerte (wie bei den AX_OR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise
Sobald an einem der 2 Eingangsadapter (IN1 … IN2) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter OUT. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf OUT gehaltenen Wert unterscheidet, wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (GEN_AD_OR) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen AX_OR-Bausteinen wird hier jedes Bit des DWORD-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ unidirectional::AD – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_OR_2_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ DWORD zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer DWORD-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen
Im Gegensatz zu AX_OR_2, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet AD_OR_2_UNGATED auf dem vollständigen Bitmuster eines DWORD-Werts. Verglichen mit dem Standard-Baustein OR_2 ver-

wendet AD_OR_2_UNGATED Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- AD_OR_2: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

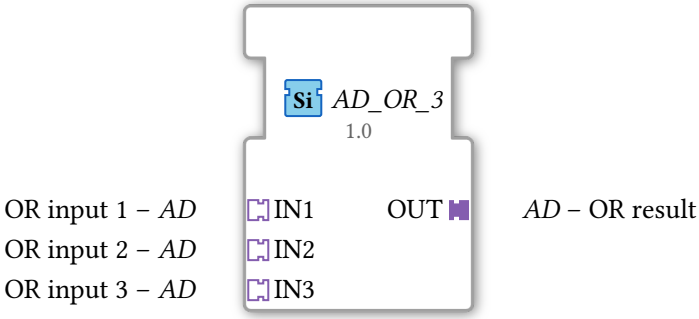
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AD_OR_2_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für DWORD-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_OR_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_OR_3** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 3 Eingangswerten vom Typ DWORD (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_OR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` … `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_OR_3 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_OR_3` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AD_OR_3` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AD_OR_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

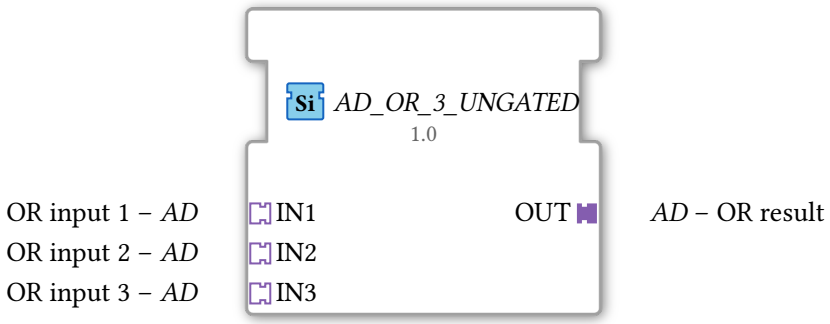
Fazit

Der **AD_OR_3** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_OR_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AD_OR_3. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht

gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_OR_3_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 3 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` … `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.

- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_OR_3_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_OR_3_UNGATED` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AD_OR_3_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AD_OR_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

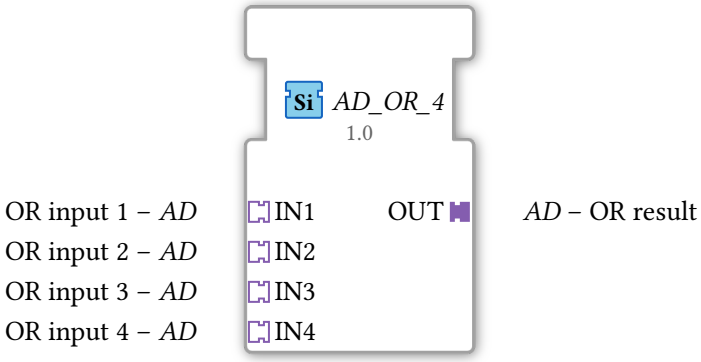
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AD_OR_3_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_OR_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_OR_4** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 4 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AD)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise
Sobald an einem der 4 Eingangsadapter (`IN1` … `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten
- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_OR_4 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien
- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen
Im Gegensatz zu `AX_OR_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_OR_4` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet

AD_OR_4 Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AD_OR_4_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

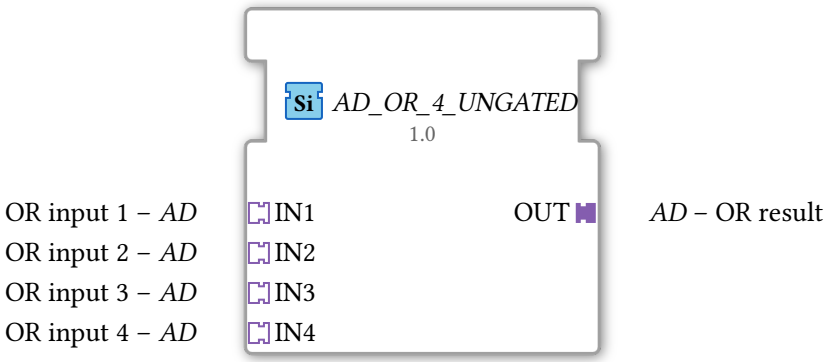
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AD_OR_4** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für DWORD-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_OR_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AD_OR_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_OR_4_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 4 Eingangswerten vom Typ DWORD (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_OR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AD)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 4 Eingangsadapter (`IN1` ... `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_OR_4_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_OR_4_UNGATED` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AD_OR_4_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AD_OR_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

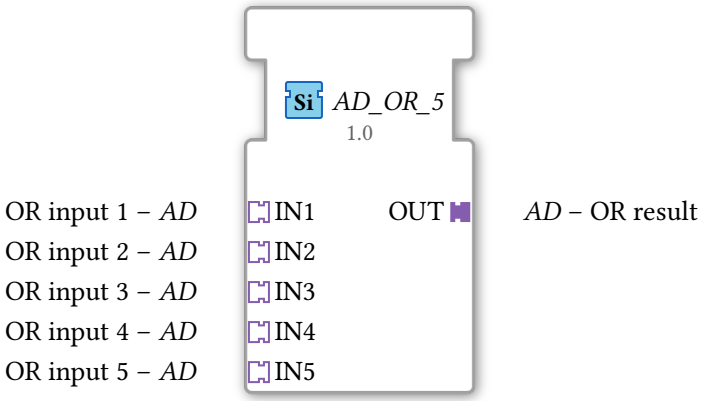
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AD_OR_4_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_OR_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_OR_5** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 5 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AD)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AD)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 5 Eingangsadapter (`IN1` ... `IN5`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 5 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-

Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_OR_5 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_5`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_OR_5` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AD_OR_5` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AD_OR_5_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

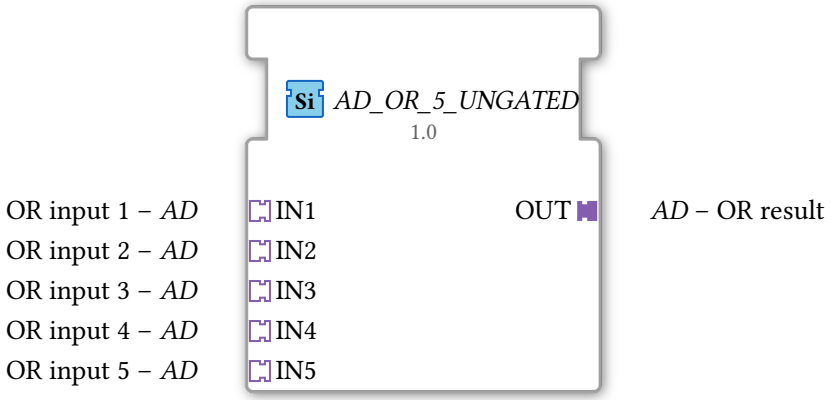
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AD_OR_5** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_OR_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AD_OR_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_OR_5_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 5 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AD)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AD)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 5 Eingangsadapter (`IN1` … `IN5`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 5 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_OR_5_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_5`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_OR_5_UNGATED` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AD_OR_5_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AD_OR_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

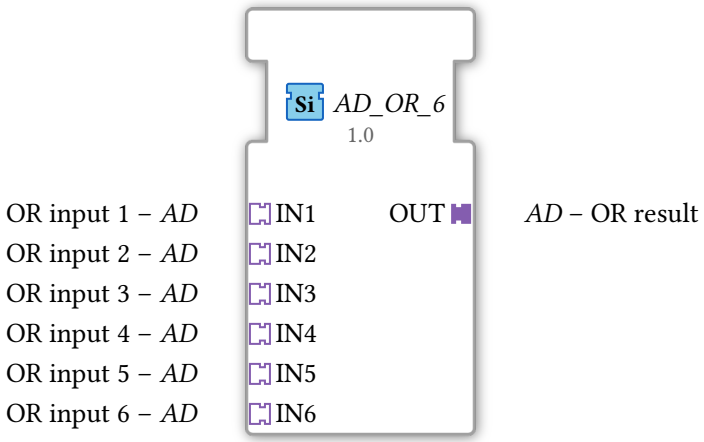
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AD_OR_5_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_OR_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_OR_6** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 6 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AD)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AD)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AD)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 6 Eingangsadapter (`IN1` … `IN6`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 6 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_OR_6 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.

- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_6`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_OR_6` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AD_OR_6` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AD_OR_6_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

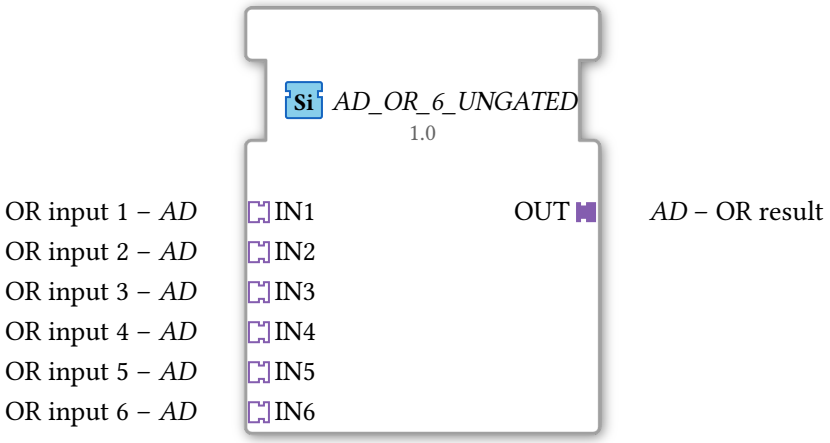
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AD_OR_6** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_OR_6_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AD_OR_6`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_OR_6_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 6 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AD)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AD)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AD)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise
Sobald an einem der 6 Eingangsadapter (`IN1` … `IN6`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 6 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_OR_6_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_6`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_OR_6_UNGATED` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AD_OR_6_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AD_OR_6`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

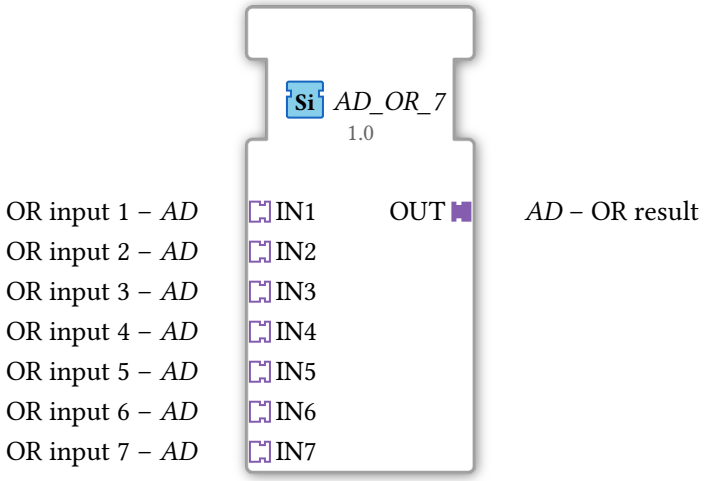
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AD_OR_6_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_OR_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_OR_7** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 7 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AD)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AD)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AD)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AD)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 7 Eingangsadapter (`IN1` … `IN7`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 7 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_OR_7 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_7`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_OR_7` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AD_OR_7` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AD_OR_7_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

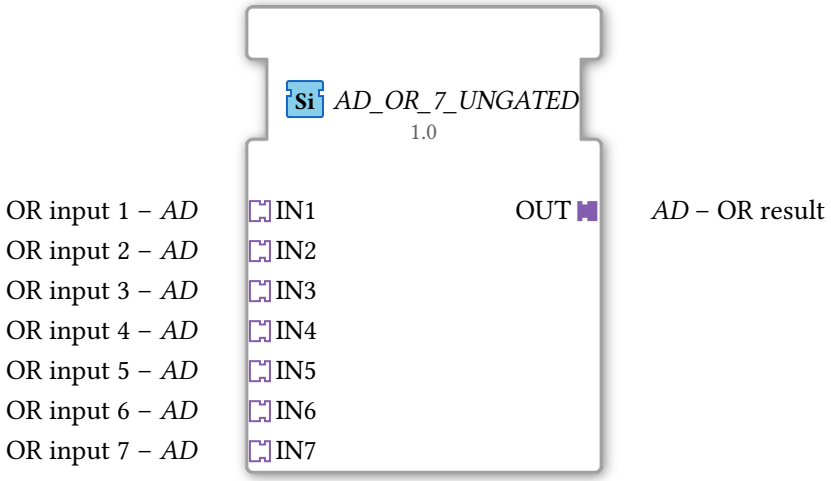
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AD_OR_7** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_OR_7_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AD_OR_7`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_OR_7_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 7 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AD)

- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AD)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AD)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AD)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AD)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 7 Eingangsadapter (`IN1` … `IN7`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 7 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_OR_7_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_7`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_OR_7_UNGATED` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AD_OR_7_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AD_OR_7`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

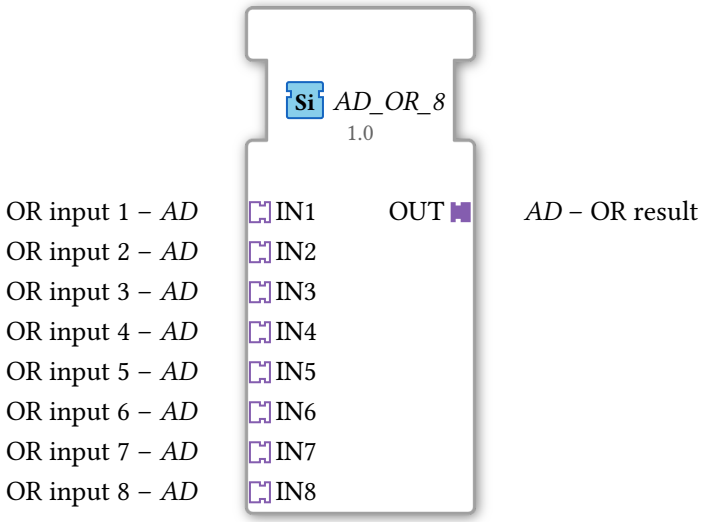
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AD_OR_7_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_OR_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_OR_8** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 8 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AD)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AD)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AD)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AD)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AD)
- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 8 Eingangsadapter (`IN1` … `IN8`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 8 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`.

Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_OR_8 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_8`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_OR_8` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AD_OR_8` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AD_OR_8_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

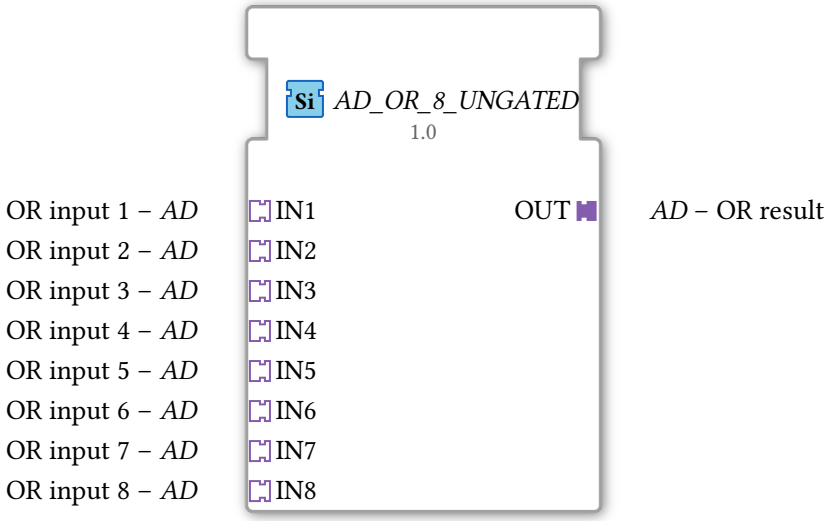
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AD_OR_8** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_OR_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AD_OR_8`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_OR_8_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 8 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AD)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AD)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AD)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AD)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AD)
- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 8 Eingangsadapter (`IN1` … `IN8`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 8 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-

Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_OR_8_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_8`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_OR_8_UNGATED` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AD_OR_8_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AD_OR_8`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

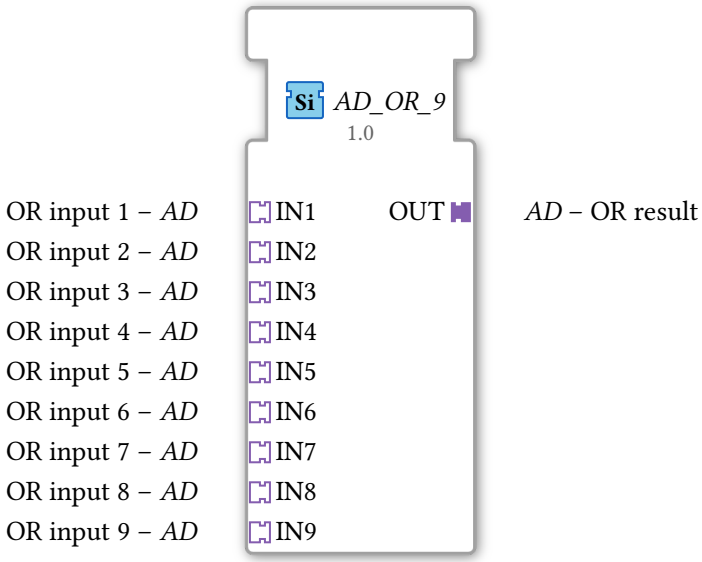
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AD_OR_8_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_OR_9



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_OR_9** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 9 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AD)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AD)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AD)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AD)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AD)
- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AD)
- **IN9**: ODER-Eingang 9 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 9 Eingangsadapter (`IN1` … `IN9`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 9 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_OR_9 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_9`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_OR_9` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AD_OR_9` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AD_OR_9_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

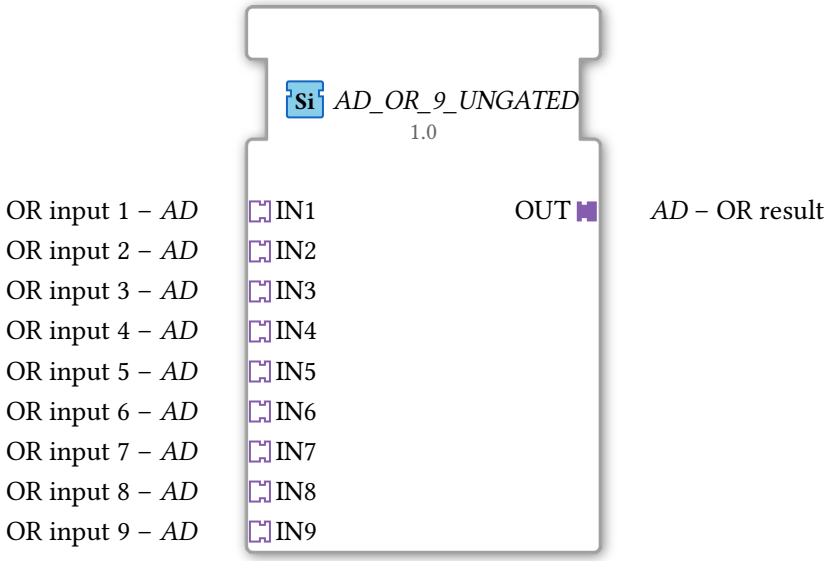
Fazit

Der **AD_OR_9** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_OR_9_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AD_OR_9. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht

gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_OR_9_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 9 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AD)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AD)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AD)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AD)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AD)
- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AD)
- **IN9**: ODER-Eingang 9 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 9 Eingangsadapter (`IN1` … `IN9`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 9 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_OR_9_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_9`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_OR_9_UNGATED` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AD_OR_9_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AD_OR_9`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

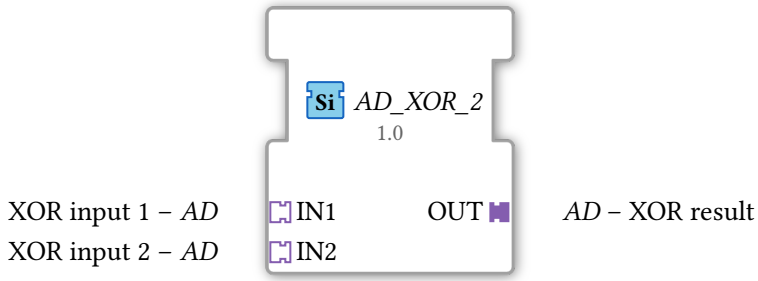
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AD_OR_9_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_XOR_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_XOR_2** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 2 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 2 Eingangsadapter (`IN1` ... `IN2`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_XOR_2 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_2`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_XOR_2` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AD_XOR_2` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AD_XOR_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

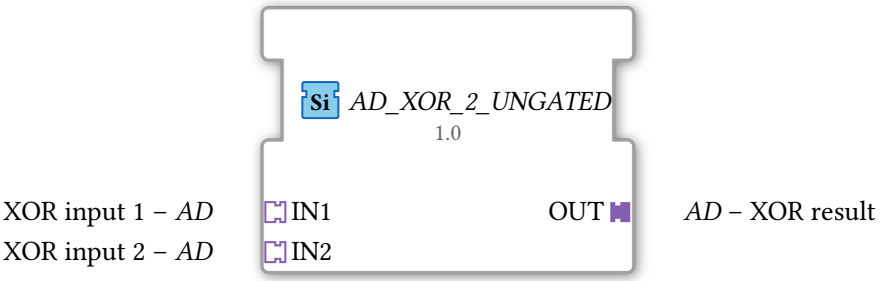
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AD_XOR_2** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_XOR_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AD_XOR_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_XOR_2_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 2 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur

booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_XOR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise
Sobald an einem der 2 Eingangsadapter (IN1 ... IN2) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter OUT. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf OUT gehaltenen Wert unterscheidet, wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (GEN_AD_XOR) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen AX_XOR-Bausteinen wird hier jedes Bit des DWORD-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ unidirectional::AD – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_XOR_2_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ DWORD zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer DWORD-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen
Im Gegensatz zu AX_XOR_2, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet AD_XOR_2_UNGATED auf dem vollständigen Bitmuster eines DWORD-Werts. Verglichen mit dem Standard-Baustein XOR_2

verwendet AD_XOR_2_UNGATED Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AD_XOR_2**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

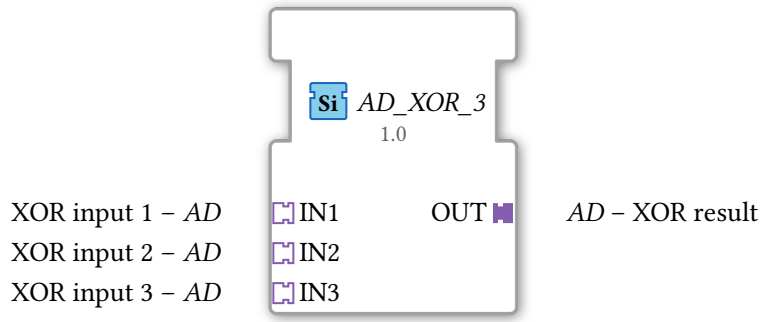
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AD_XOR_2_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für DWORD-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_XOR_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_XOR_3** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 3 Eingangswerten vom Typ DWORD (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_XOR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` … `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_XOR_3 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_XOR_3` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AD_XOR_3` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AD_XOR_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

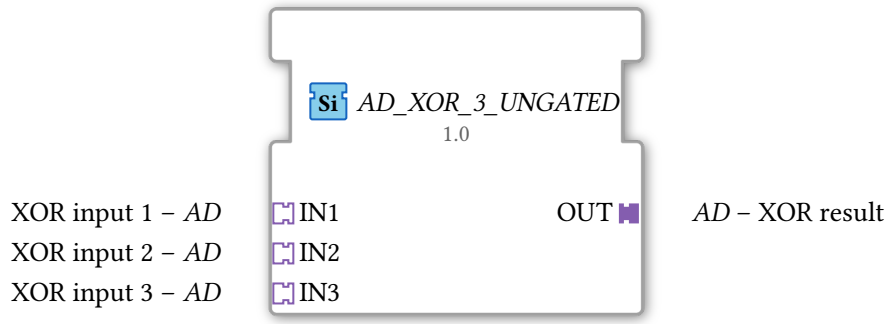
Fazit

Der **AD_XOR_3** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_XOR_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AD_XOR_3. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht

gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_XOR_3_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 3 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` … `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.

- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_XOR_3_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_XOR_3_UNGATED` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AD_XOR_3_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AD_XOR_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

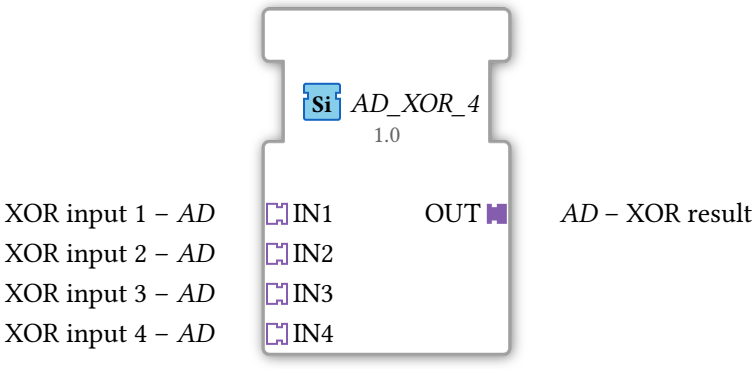
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AD_XOR_3_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_XOR_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_XOR_4** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 4 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AD)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise
Sobald an einem der 4 Eingangsadapter (`IN1` … `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_XOR_4 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen
Im Gegensatz zu `AX_XOR_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_XOR_4` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet

AD_XOR_4 Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AD_XOR_4_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

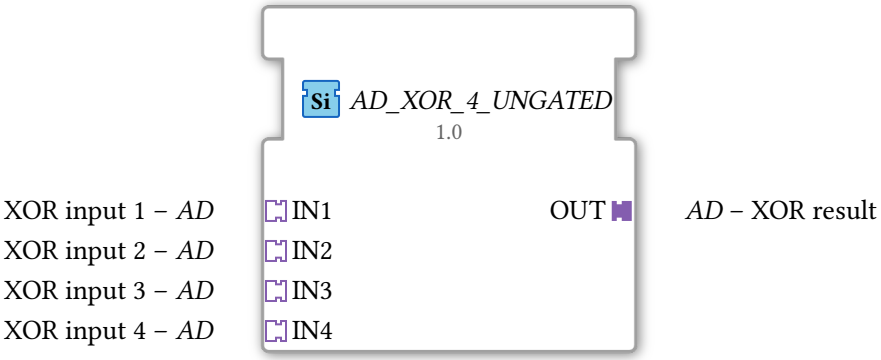
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AD_XOR_4** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für DWORD-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_XOR_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AD_XOR_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_XOR_4_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 4 Eingangswerten vom Typ DWORD (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_XOR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AD)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 4 Eingangsadapter (`IN1` ... `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_XOR_4_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_XOR_4_UNGATED` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AD_XOR_4_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AD_XOR_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

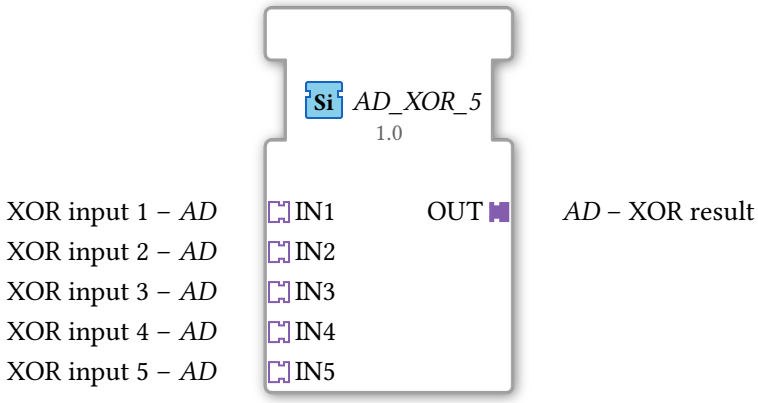
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AD_XOR_4_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_XOR_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_XOR_5** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 5 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AD)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AD)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 5 Eingangsadapter (`IN1` ... `IN5`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 5 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0`

(Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_XOR_5 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_5`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_XOR_5` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AD_XOR_5` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AD_XOR_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

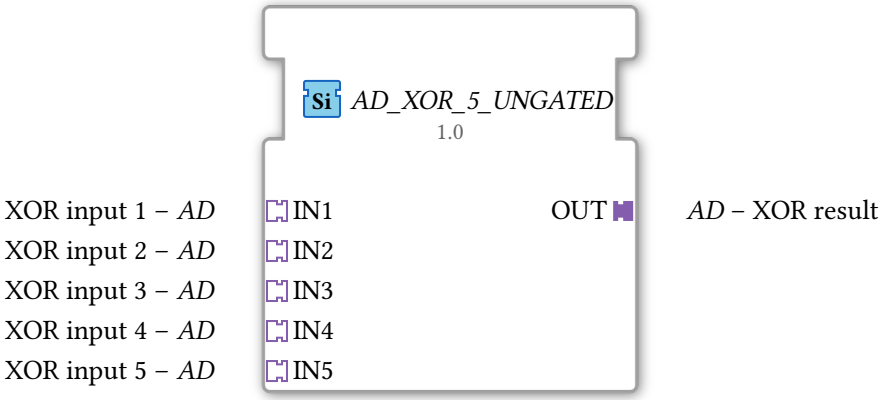
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AD_XOR_5** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_XOR_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AD_XOR_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_XOR_5_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 5 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AD)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AD)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 5 Eingangsadapter (`IN1` … `IN5`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 5 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_XOR_5_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ **Vergleich mit ähnlichen Bausteinen**

Im Gegensatz zu `AX_XOR_5`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_XOR_5_UNGATED` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AD_XOR_5_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AD_XOR_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

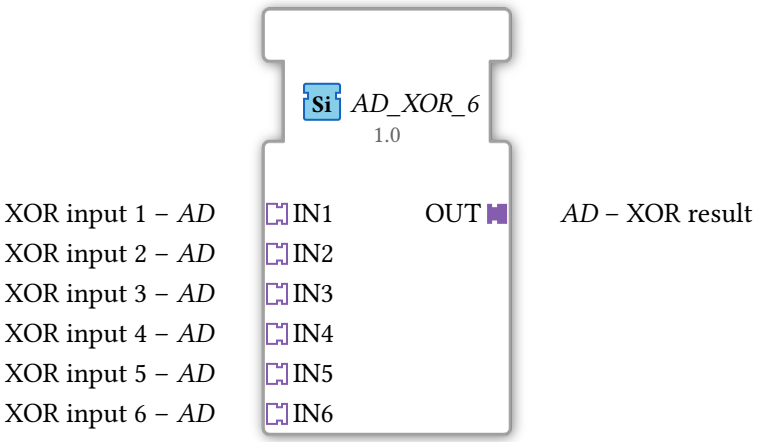
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AD_XOR_5_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_XOR_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_XOR_6** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 6 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AD)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AD)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AD)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 6 Eingangsadapter (`IN1` … `IN6`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 6 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_XOR_6 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.

- **Signalaggregation**: Zusammenführen mehrerer DWORD-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu AX_XOR_6, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet AD_XOR_6 auf dem vollständigen Bitmuster eines DWORD-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet AD_XOR_6 Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- AD_XOR_6_UNGATED: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

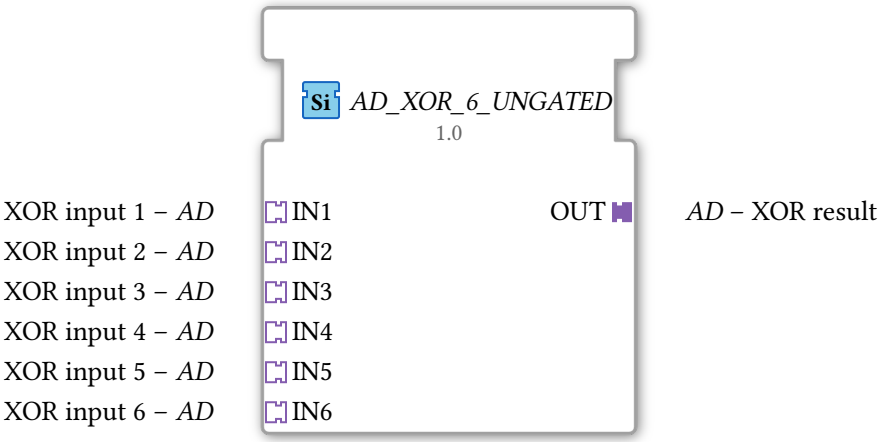
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AD_XOR_6** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für DWORD-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_XOR_6_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AD_XOR_6. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_XOR_6_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 6 Eingangswerten vom Typ DWORD (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_XOR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AD)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AD)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AD)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise
Sobald an einem der 6 Eingangsadapter (`IN1` … `IN6`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 6 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_XOR_6_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu AX_XOR_6, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet AD_XOR_6_UNGATED auf dem vollständigen Bitmuster eines DWORD-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet AD_XOR_6_UNGATED Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AD_XOR_6**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

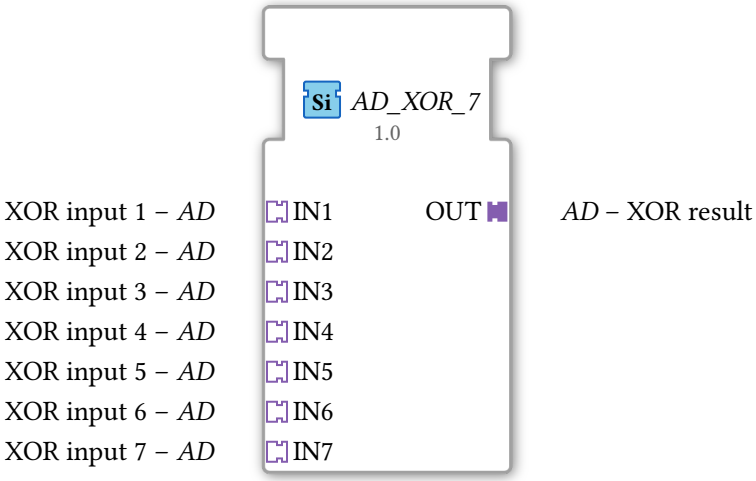
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AD_XOR_6_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für DWORD-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_XOR_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_XOR_7** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 7 Eingangswerten vom Typ DWORD (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_XOR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AD)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AD)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AD)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AD)
- **IN7**: XOR (exklusives ODER)-Eingang 7 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 7 Eingangsadapter (`IN1` ... `IN7`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 7 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_XOR_7 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_7`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_XOR_7` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AD_XOR_7` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AD_XOR_7_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

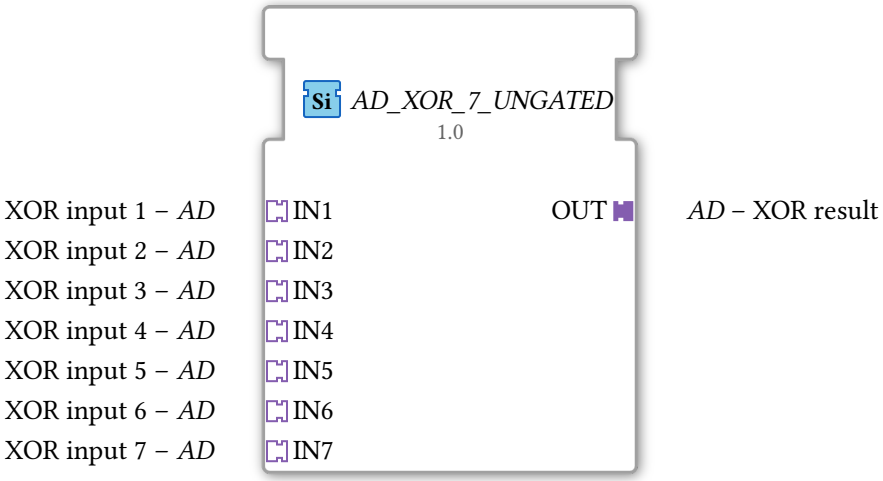
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AD_XOR_7** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_XOR_7_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AD_XOR_7`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_XOR_7_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 7 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AD)

- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AD)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AD)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AD)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AD)
- **IN7**: XOR (exklusives ODER)-Eingang 7 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 7 Eingangsadapter (`IN1` ... `IN7`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 7 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_XOR_7_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_7`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_XOR_7_UNGATED` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AD_XOR_7_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AD_XOR_7`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

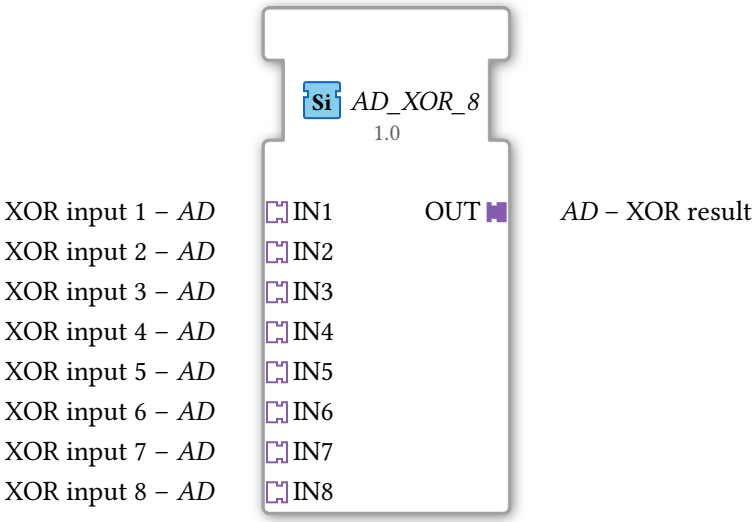
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AD_XOR_7_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_XOR_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_XOR_8** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 8 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AD)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AD)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AD)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AD)
- **IN7**: XOR (exklusives ODER)-Eingang 7 (Typ: adapter::types::unidirectional::AD)
- **IN8**: XOR (exklusives ODER)-Eingang 8 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 8 Eingangsadapter (`IN1` ... `IN8`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 8 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den

Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_XOR_8 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_8`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_XOR_8` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AD_XOR_8` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AD_XOR_8_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

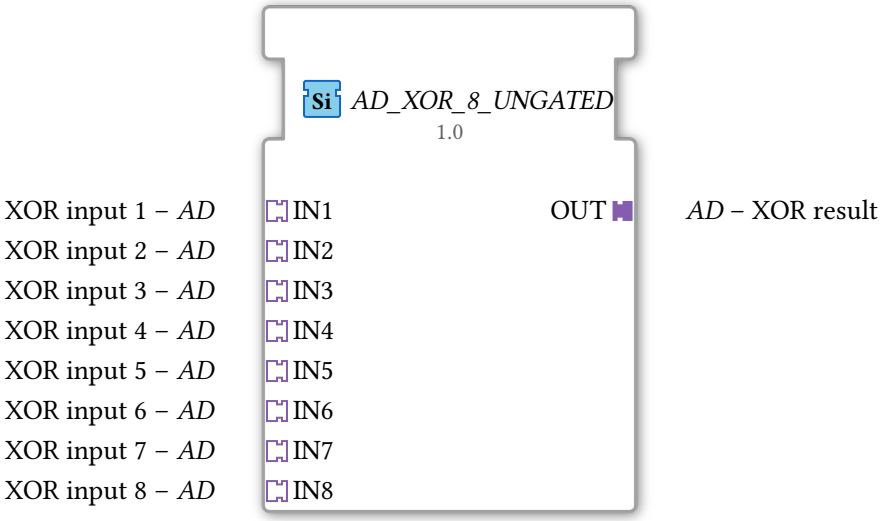
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AD_XOR_8** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AD_XOR_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AD_XOR_8`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_XOR_8_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 8 Eingangswerten vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AD)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AD)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AD)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AD)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AD)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AD)
- **IN7**: XOR (exklusives ODER)-Eingang 7 (Typ: adapter::types::unidirectional::AD)
- **IN8**: XOR (exklusives ODER)-Eingang 8 (Typ: adapter::types::unidirectional::AD)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AD)

Funktionsweise

Sobald an einem der 8 Eingangsadapter (`IN1` … `IN8`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 8 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0

(Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AD_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `DWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AD` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AD_XOR_8_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `DWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `DWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_8`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AD_XOR_8_UNGATED` auf dem vollständigen Bitmuster eines `DWORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AD_XOR_8_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AD_XOR_8`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

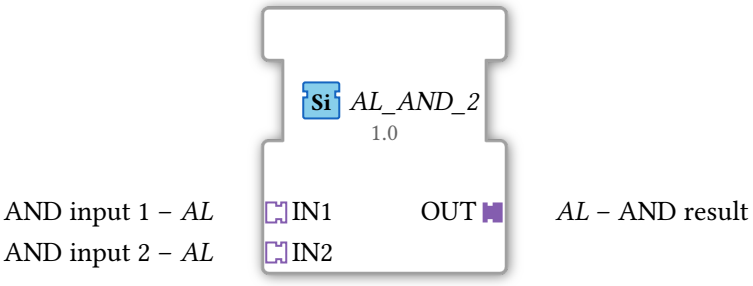
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AD_XOR_8_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `DWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_AND_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_AND_2** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 2 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_AND`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 2 Eingangsadapter (`IN1` … `IN2`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_AND_2 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu AX_AND_2, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet AL_AND_2 auf dem vollständigen Bitmuster eines LWORD-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet AL_AND_2 Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AL_AND_2_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

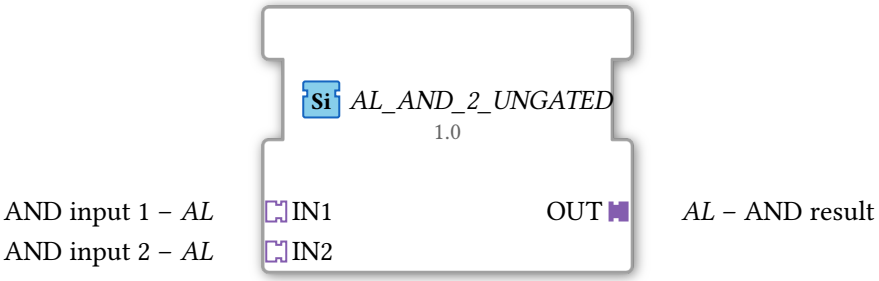
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AL_AND_2** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für LWORD-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_AND_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AL_AND_2. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_AND_2_UNGATED** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 2 Eingangswerten vom Typ LWORD (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_AND-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 2 Eingangsadapter (`IN1` ... `IN2`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_AND_2_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_AND_2`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_AND_2_UNGATED` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet `AL_AND_2_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AL_AND_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

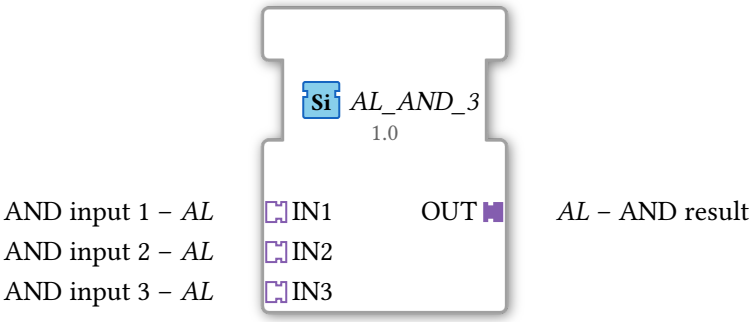
Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AL_AND_2_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn

vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_AND_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_AND_3** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 3 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_AND`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: UND-Eingang 3 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` ... `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.

- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_AND_3 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_AND_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_AND_3` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet `AL_AND_3` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AL_AND_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

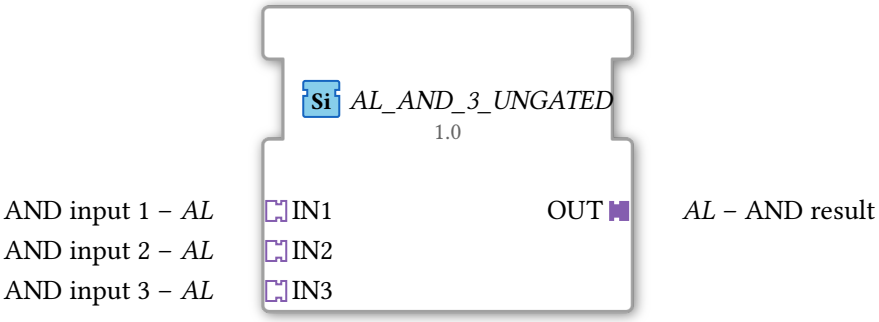
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AL_AND_3** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_AND_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AL_AND_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_AND_3_UNGATED** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 3 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_AND`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: UND-Eingang 3 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` … `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.

- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_AND_3_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_AND_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_AND_3_UNGATED` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet `AL_AND_3_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AL_AND_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

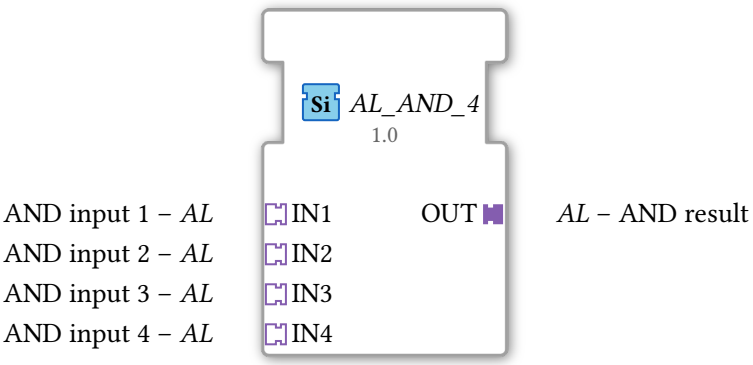
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AL_AND_3_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_AND_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_AND_4** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 4 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_AND`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: UND-Eingang 3 (Typ: adapter::types::unidirectional::AL)
- **IN4**: UND-Eingang 4 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise
Sobald an einem der 4 Eingangsadapter (`IN1` … `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten
- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_AND_4 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien
- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen
Im Gegensatz zu `AX_AND_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_AND_4` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet

AL_AND_4 Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AL_AND_4_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

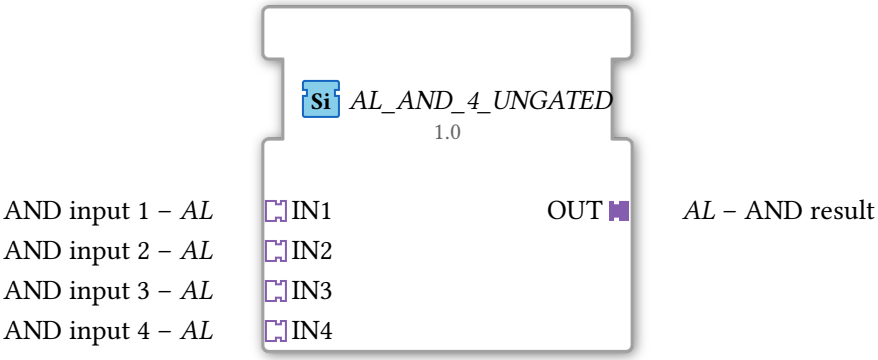
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AL_AND_4** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für LWORD-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_AND_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AL_AND_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_AND_4_UNGATED** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 4 Eingangswerten vom Typ LWORD (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_AND-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: UND-Eingang 3 (Typ: adapter::types::unidirectional::AL)
- **IN4**: UND-Eingang 4 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 4 Eingangsadapter (`IN1` … `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_AND_4_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_AND_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_AND_4_UNGATED` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet `AL_AND_4_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AL_AND_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

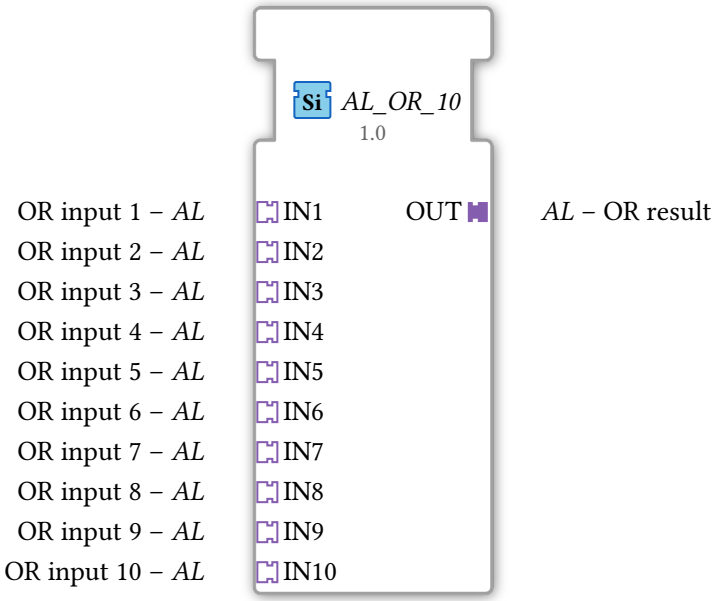
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AL_AND_4_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_OR_10



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_OR_10** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 10 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AL)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AL)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AL)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AL)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AL)

- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AL)
- **IN9**: ODER-Eingang 9 (Typ: adapter::types::unidirectional::AL)
- **IN10**: ODER-Eingang 10 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 10 Eingangsadapter (`IN1` ... `IN10`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 10 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_OR_10 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_10`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_OR_10` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AL_OR_10` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AL_OR_10_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

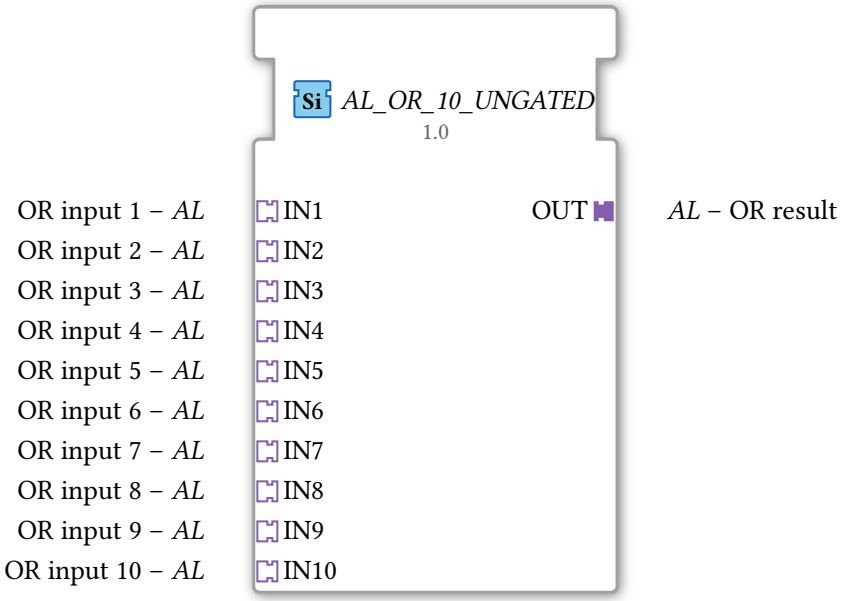
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AL_OR_10** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_OR_10_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AL_OR_10`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_OR_10_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 10 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AL)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AL)

- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AL)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AL)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AL)
- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AL)
- **IN9**: ODER-Eingang 9 (Typ: adapter::types::unidirectional::AL)
- **IN10**: ODER-Eingang 10 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 10 Eingangsadapter (`IN1` … `IN10`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 10 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_OR_10_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_10`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_OR_10_UNGATED` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AL_OR_10_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AL_OR_10`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

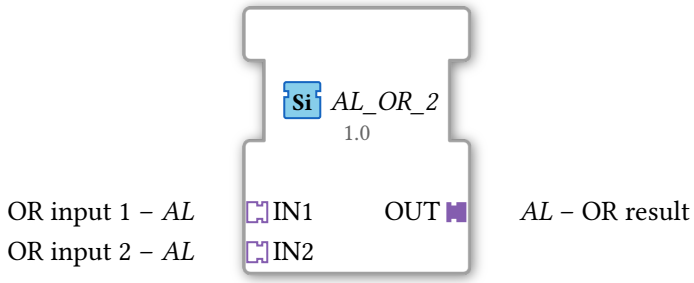
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AL_OR_10_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_OR_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_OR_2** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 2 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 2 Eingangsadapter (`IN1` … `IN2`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_OR_2 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_2`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_OR_2` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AL_OR_2` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AL_OR_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

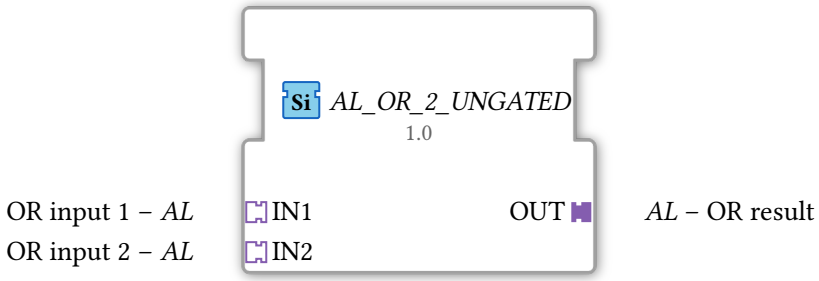
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AL_OR_2** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_OR_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AL_OR_2. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_OR_2_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 2 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüp-

fung einzelner Wahrheitswerte (wie bei den AX_OR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise
Sobald an einem der 2 Eingangsadapter (IN1 … IN2) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter OUT. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf OUT gehaltenen Wert unterscheidet, wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (GEN_AL_OR) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen AX_OR-Bausteinen wird hier jedes Bit des LWORD-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ unidirectional::AL – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_OR_2_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ LWORD zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer LWORD-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen
Im Gegensatz zu AX_OR_2, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet AL_OR_2_UNGATED auf dem vollständigen Bitmuster eines LWORD-Werts. Verglichen mit dem Standard-Baustein OR_2 ver-

wendet AL_OR_2_UNGATED Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- AL_OR_2: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

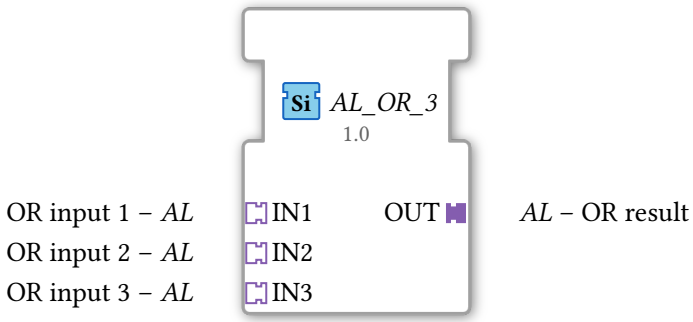
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AL_OR_2_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für LWORD-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_OR_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_OR_3** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 3 Eingangswerten vom Typ LWORD (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_OR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` … `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_OR_3 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_OR_3` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AL_OR_3` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AL_OR_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

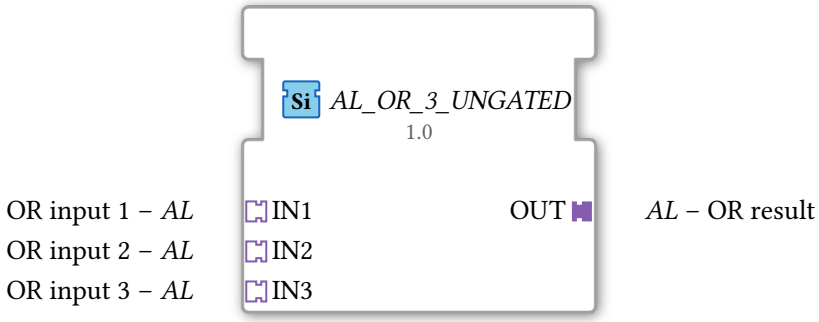
Fazit

Der **AL_OR_3** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_OR_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AL_OR_3. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht

gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_OR_3_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 3 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` … `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.

- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_OR_3_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_OR_3_UNGATED` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AL_OR_3_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AL_OR_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

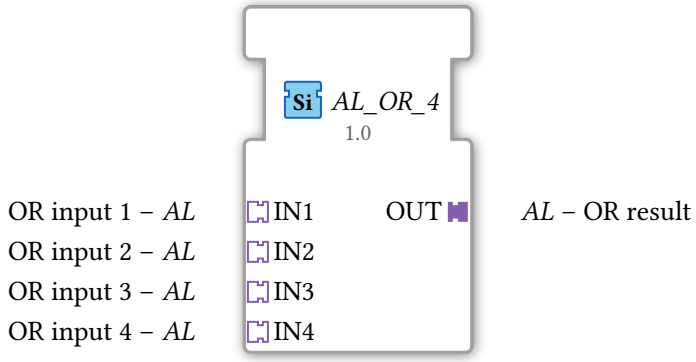
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AL_OR_3_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_OR_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_OR_4** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 4 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AL)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise
Sobald an einem der 4 Eingangsadapter (`IN1` … `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten
- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_OR_4 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien
- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen
Im Gegensatz zu `AX_OR_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_OR_4` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet

AL_OR_4 Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AL_OR_4_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

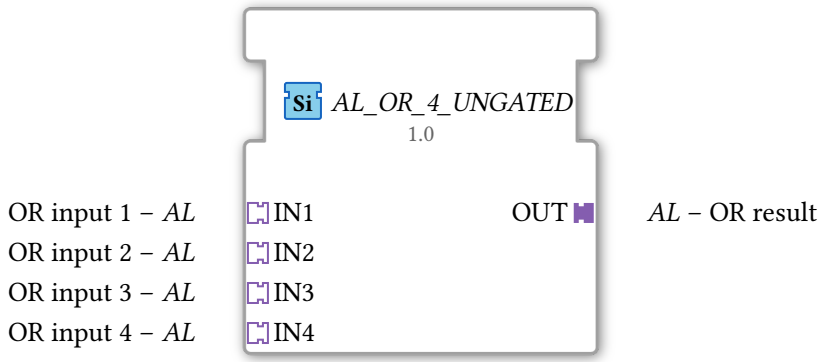
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AL_OR_4** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für LWORD-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_OR_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AL_OR_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_OR_4_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 4 Eingangswerten vom Typ LWORD (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_OR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AL)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 4 Eingangsadapter (`IN1` ... `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_OR_4_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_OR_4_UNGATED` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AL_OR_4_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AL_OR_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

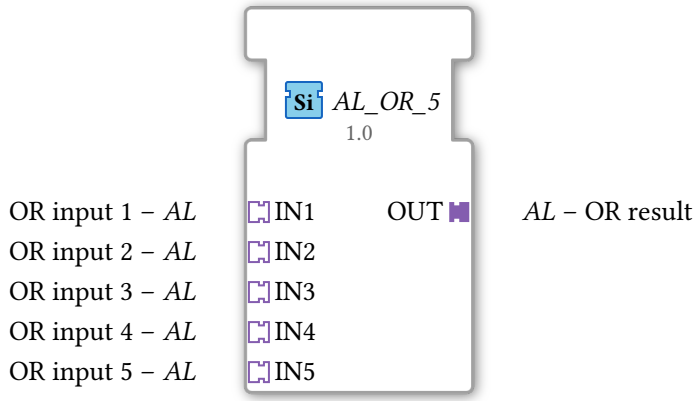
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AL_OR_4_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_OR_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_OR_5** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 5 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AL)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AL)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 5 Eingangsadapter (`IN1` ... `IN5`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 5 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-

Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_OR_5 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_5`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_OR_5` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AL_OR_5` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AL_OR_5_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

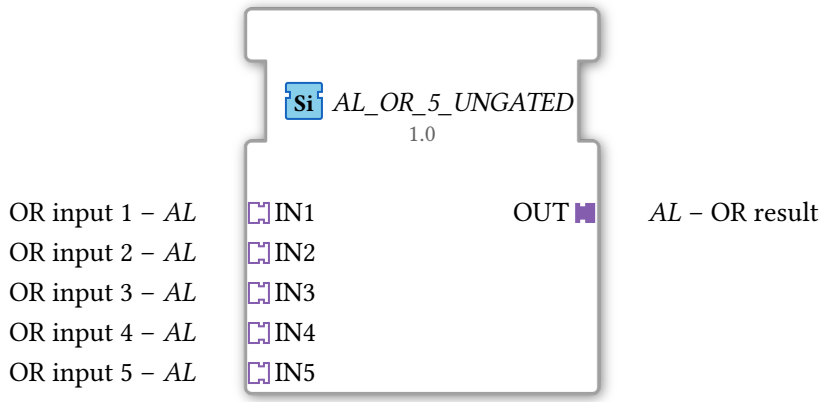
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AL_OR_5** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_OR_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AL_OR_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_OR_5_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 5 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AL)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AL)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 5 Eingangsadapter (`IN1` … `IN5`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 5 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_OR_5_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_5`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_OR_5_UNGATED` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AL_OR_5_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AL_OR_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

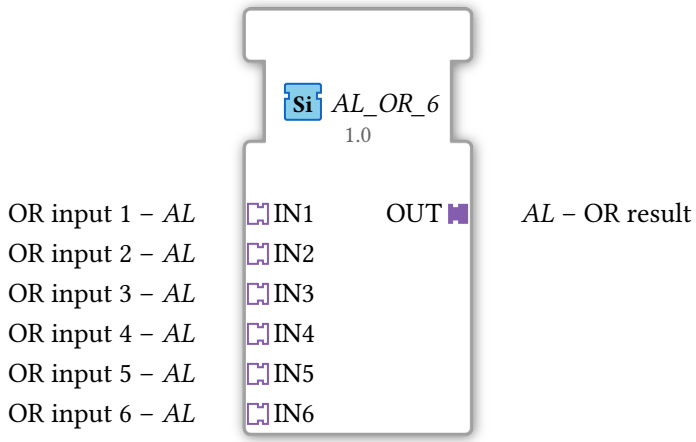
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AL_OR_5_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_OR_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_OR_6** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 6 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AL)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AL)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AL)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 6 Eingangsadapter (`IN1` … `IN6`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 6 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_OR_6 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.

- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_6`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_OR_6` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AL_OR_6` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AL_OR_6_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

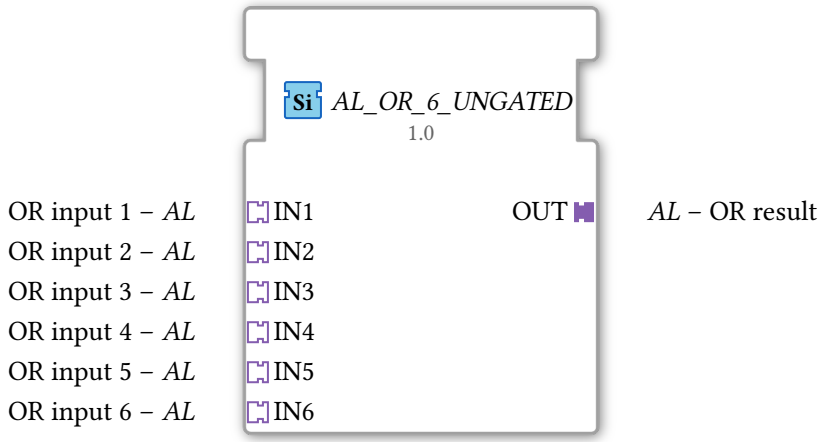
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AL_OR_6** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_OR_6_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AL_OR_6`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_OR_6_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 6 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AL)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AL)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AL)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise
Sobald an einem der 6 Eingangsadapter (`IN1` … `IN6`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 6 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_OR_6_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu AX_OR_6, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet AL_OR_6_UNGATED auf dem vollständigen Bitmuster eines LWORD-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet AL_OR_6_UNGATED Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- AL_OR_6: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

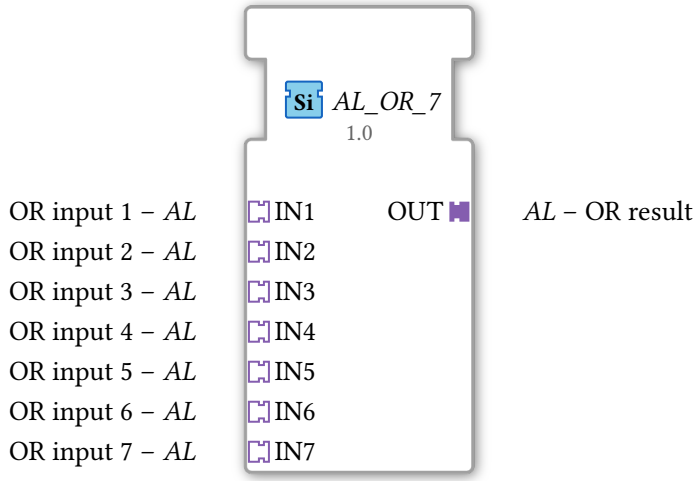
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AL_OR_6_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für LWORD-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_OR_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_OR_7** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 7 Eingangswerten vom Typ LWORD (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_OR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AL)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AL)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AL)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AL)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 7 Eingangsadapter (`IN1` ... `IN7`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 7 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_OR_7 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_7`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_OR_7` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AL_OR_7` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AL_OR_7_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

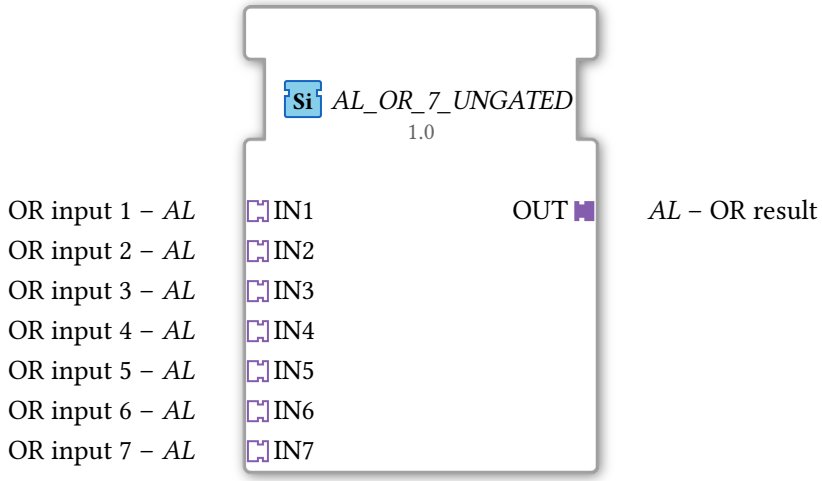
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AL_OR_7** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_OR_7_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AL_OR_7`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_OR_7_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 7 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AL)

- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AL)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AL)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AL)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AL)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 7 Eingangsadapter (`IN1` … `IN7`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 7 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_OR_7_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_7`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_OR_7_UNGATED` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AL_OR_7_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AL_OR_7`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

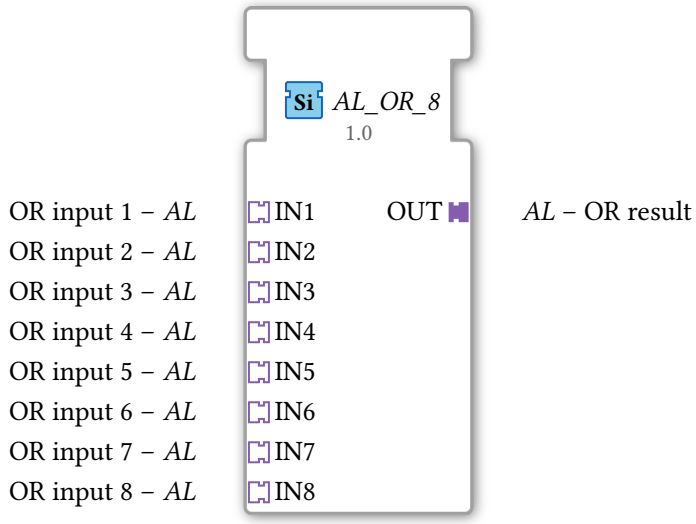
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AL_OR_7_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_OR_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_OR_8** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 8 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AL)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AL)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AL)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AL)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AL)
- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 8 Eingangsadapter (`IN1` … `IN8`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 8 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`.

Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_OR_8 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_8`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_OR_8` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AL_OR_8` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AL_OR_8_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

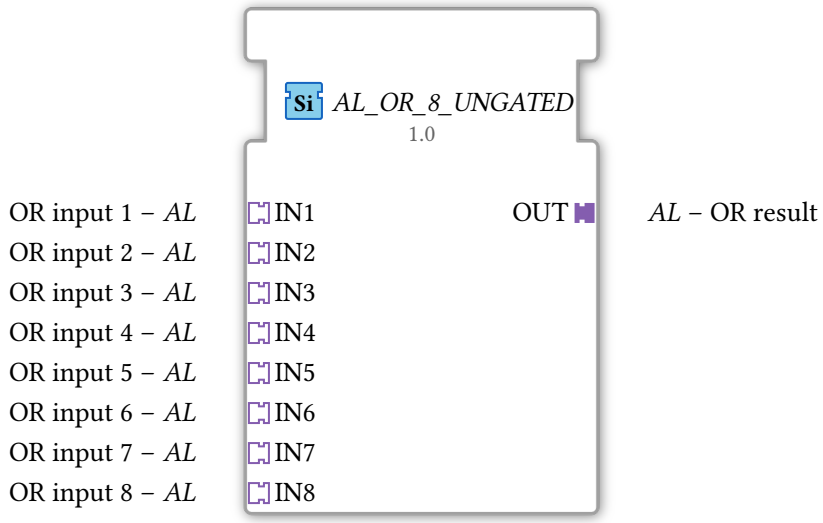
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AL_OR_8** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_OR_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AL_OR_8`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_OR_8_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 8 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AL)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AL)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AL)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AL)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AL)
- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 8 Eingangsadapter (`IN1` … `IN8`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 8 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-

Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_OR_8_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_8`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_OR_8_UNGATED` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AL_OR_8_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AL_OR_8`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

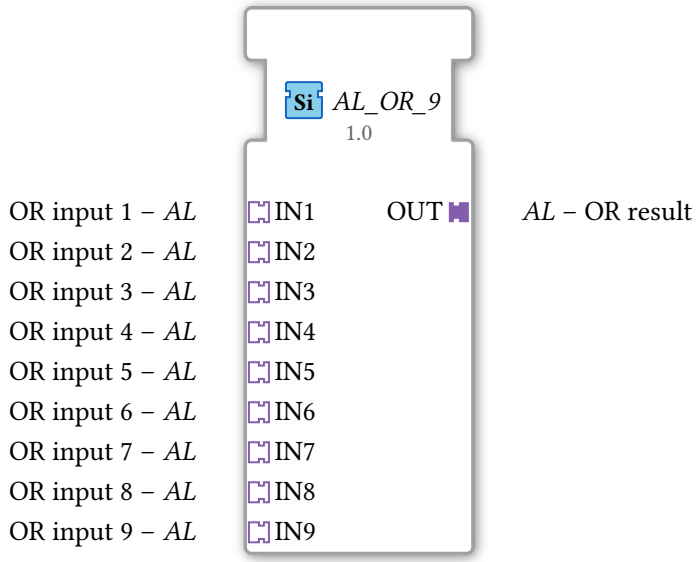
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AL_OR_8_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_OR_9



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_OR_9** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 9 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AL)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AL)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AL)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AL)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AL)
- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AL)
- **IN9**: ODER-Eingang 9 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 9 Eingangsadapter (`IN1` … `IN9`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 9 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_OR_9 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_9`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_OR_9` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AL_OR_9` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AL_OR_9_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

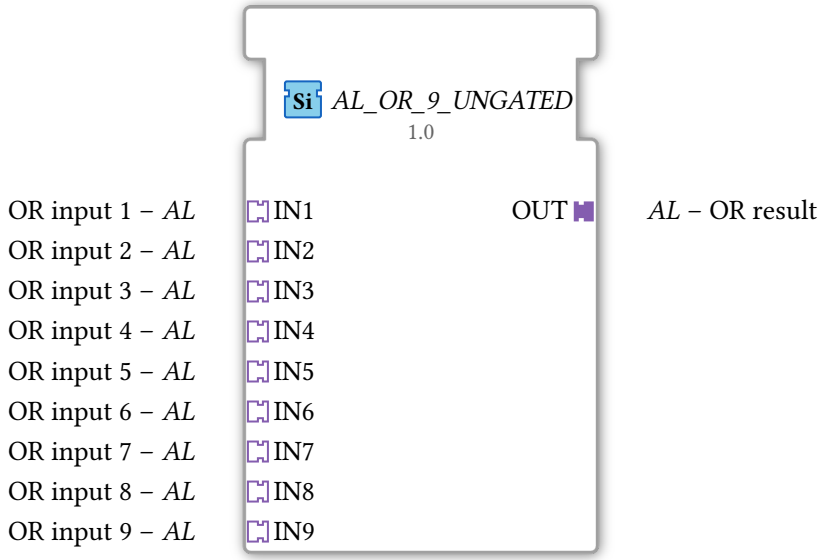
Fazit

Der **AL_OR_9** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_OR_9_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AL_OR_9. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht

gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_OR_9_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 9 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AL)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AL)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AL)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AL)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AL)
- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AL)
- **IN9**: ODER-Eingang 9 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 9 Eingangsadapter (`IN1` … `IN9`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 9 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_OR_9_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_9`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_OR_9_UNGATED` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AL_OR_9_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AL_OR_9`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

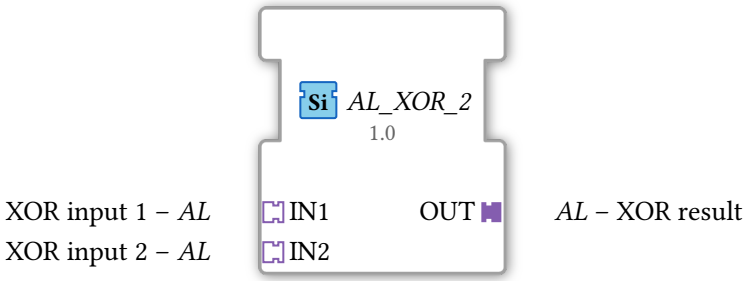
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AL_OR_9_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_XOR_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_XOR_2** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 2 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 2 Eingangsadapter (`IN1` ... `IN2`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_XOR_2 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_2`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_XOR_2` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AL_XOR_2` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AL_XOR_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

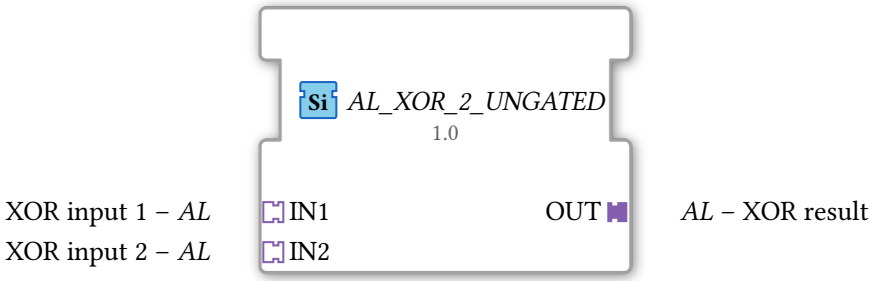
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AL_XOR_2** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_XOR_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AL_XOR_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_XOR_2_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 2 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur

booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_XOR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise
Sobald an einem der 2 Eingangsadapter (IN1 ... IN2) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter OUT. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf OUT gehaltenen Wert unterscheidet, wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (GEN_AL_XOR) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen AX_XOR-Bausteinen wird hier jedes Bit des LWORD-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ unidirectional::AL – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_XOR_2_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ LWORD zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer LWORD-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen
Im Gegensatz zu AX_XOR_2, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet AL_XOR_2_UNGATED auf dem vollständigen Bitmuster eines LWORD-Werts. Verglichen mit dem Standard-Baustein XOR_2

verwendet AL_XOR_2_UNGATED Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AL_XOR_2**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

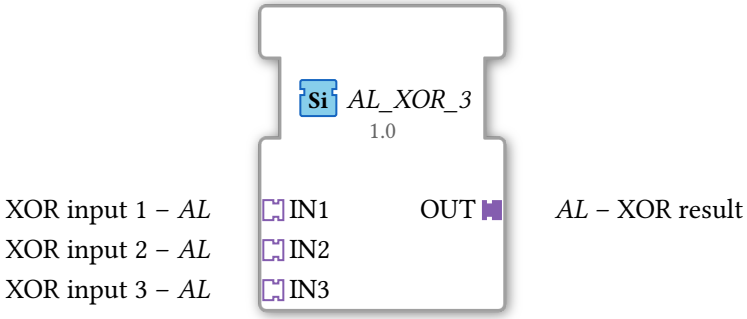
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AL_XOR_2_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für LWORD-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_XOR_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_XOR_3** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 3 Eingangswerten vom Typ LWORD (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_XOR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` … `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_XOR_3 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_XOR_3` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AL_XOR_3` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AL_XOR_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

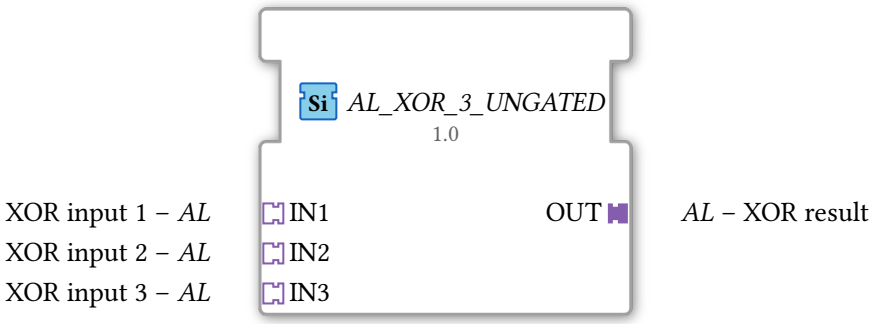
Fazit

Der **AL_XOR_3** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_XOR_3_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von AL_XOR_3. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht

gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_XOR_3_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 3 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` … `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.

- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_XOR_3_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_XOR_3_UNGATED` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AL_XOR_3_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AL_XOR_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

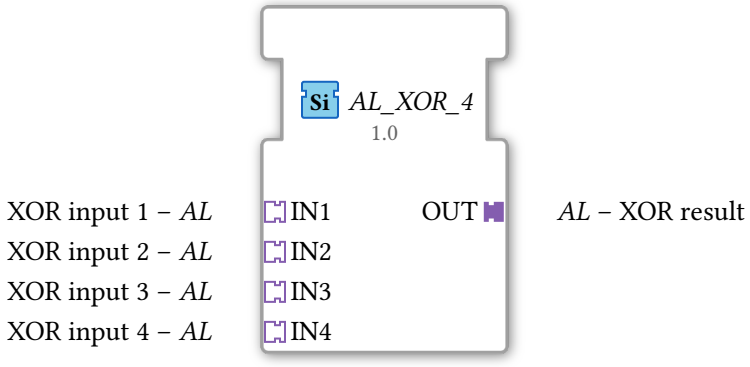
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AL_XOR_3_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_XOR_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_XOR_4** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 4 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AL)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise
Sobald an einem der 4 Eingangsadapter (`IN1` … `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten
- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_XOR_4 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien
- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen
Im Gegensatz zu `AX_XOR_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_XOR_4` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet

AL_XOR_4 Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AL_XOR_4_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

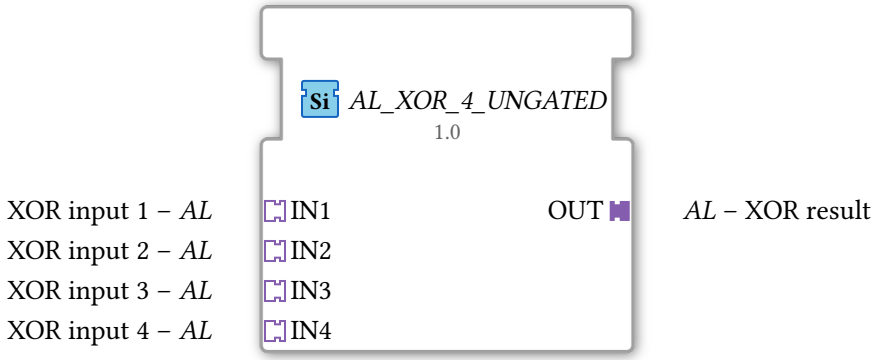
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AL_XOR_4** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für LWORD-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_XOR_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AL_XOR_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_XOR_4_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 4 Eingangswerten vom Typ LWORD (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_XOR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AL)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 4 Eingangsadapter (`IN1` ... `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_XOR_4_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_XOR_4_UNGATED` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AL_XOR_4_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AL_XOR_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

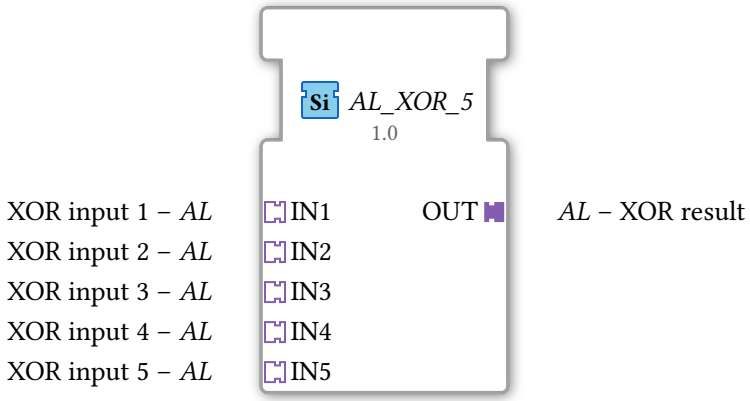
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AL_XOR_4_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_XOR_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_XOR_5** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 5 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AL)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AL)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 5 Eingangsadapter (`IN1` ... `IN5`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 5 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0

(Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_XOR_5 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_5`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_XOR_5` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AL_XOR_5` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AL_XOR_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

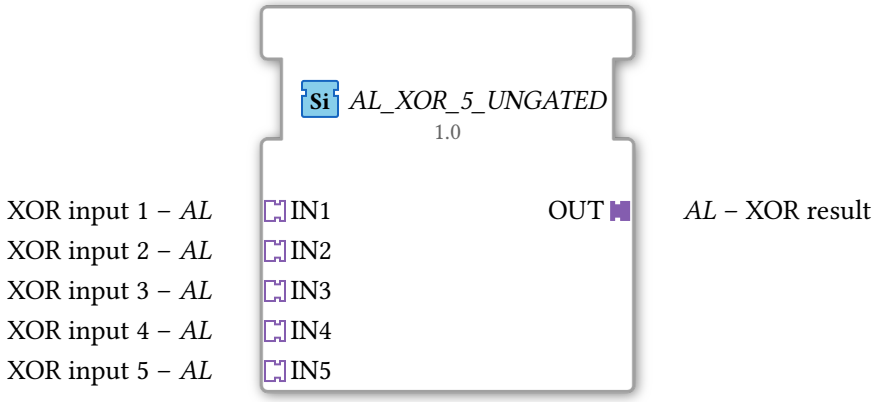
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AL_XOR_5** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_XOR_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AL_XOR_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_XOR_5_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 5 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AL)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AL)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 5 Eingangsadapter (`IN1` … `IN5`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 5 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_XOR_5_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_5`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_XOR_5_UNGATED` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AL_XOR_5_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AL_XOR_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

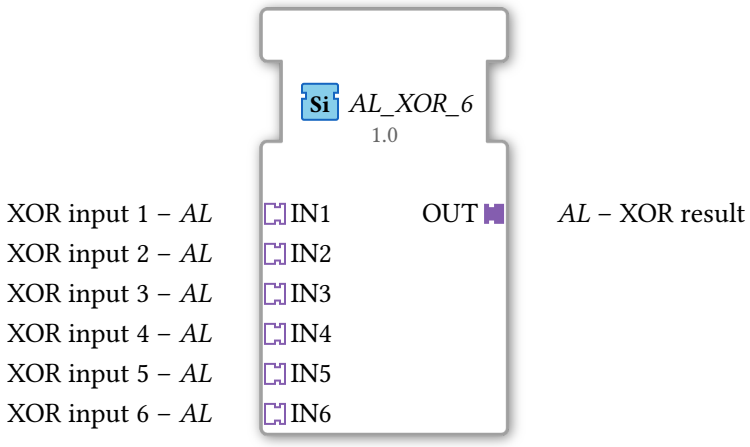
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AL_XOR_5_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_XOR_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_XOR_6** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 6 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AL)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AL)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AL)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 6 Eingangsadapter (`IN1` … `IN6`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 6 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_XOR_6 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.

- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_6`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_XOR_6` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AL_XOR_6` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AL_XOR_6_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

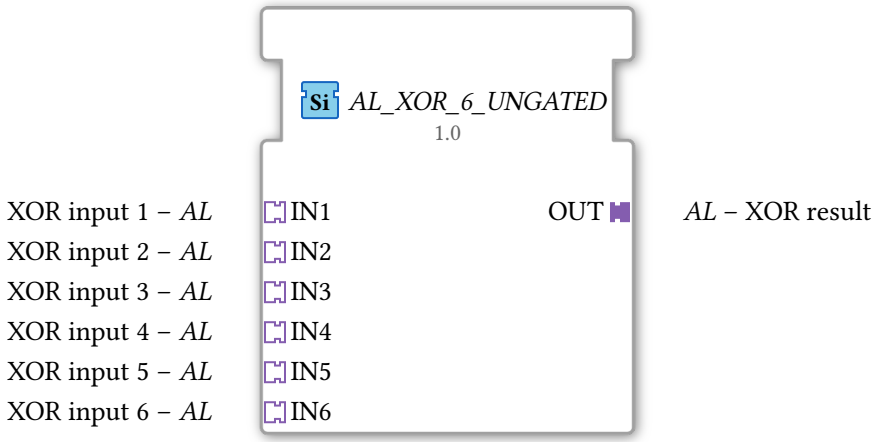
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AL_XOR_6** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_XOR_6_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AL_XOR_6`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_XOR_6_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 6 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AL)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AL)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AL)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise
Sobald an einem der 6 Eingangsadapter (`IN1` … `IN6`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 6 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_XOR_6_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu AX_XOR_6, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet AL_XOR_6_UNGATED auf dem vollständigen Bitmuster eines LWORD-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet AL_XOR_6_UNGATED Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AL_XOR_6**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

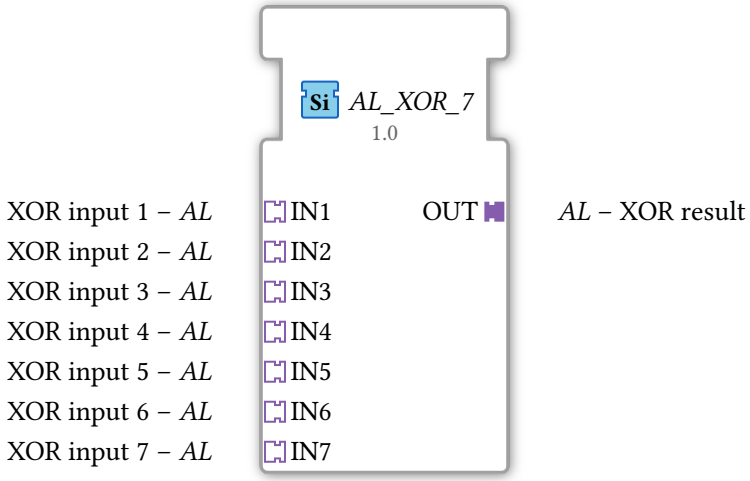
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AL_XOR_6_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für LWORD-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_XOR_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_XOR_7** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 7 Eingangswerten vom Typ LWORD (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_XOR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AL)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AL)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AL)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AL)
- **IN7**: XOR (exklusives ODER)-Eingang 7 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 7 Eingangsadapter (`IN1` … `IN7`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 7 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_XOR_7 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_7`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_XOR_7` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AL_XOR_7` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AL_XOR_7_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

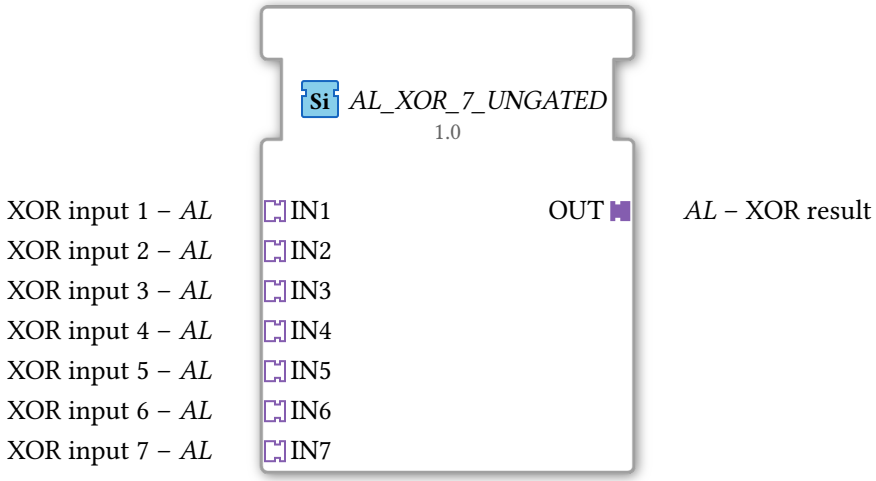
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AL_XOR_7** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_XOR_7_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AL_XOR_7`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_XOR_7_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 7 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AL)

- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AL)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AL)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AL)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AL)
- **IN7**: XOR (exklusives ODER)-Eingang 7 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 7 Eingangsadapter (`IN1` ... `IN7`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 7 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_XOR_7_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_7`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_XOR_7_UNGATED` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AL_XOR_7_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AL_XOR_7`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

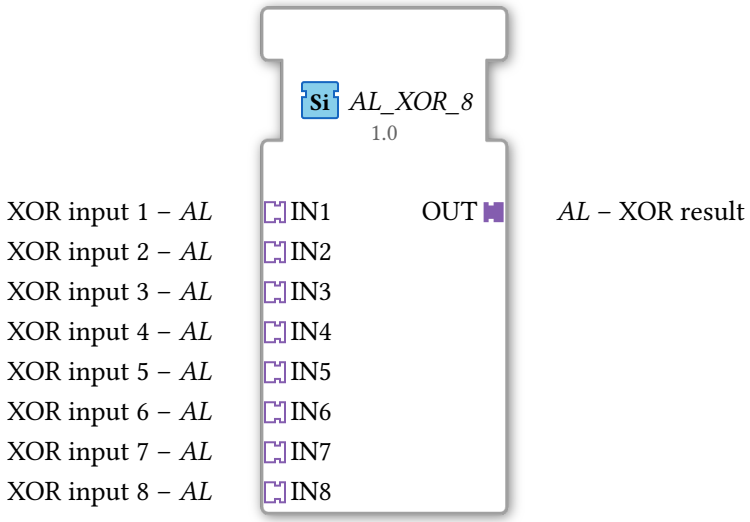
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AL_XOR_7_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_XOR_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_XOR_8** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 8 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AL)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AL)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AL)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AL)
- **IN7**: XOR (exklusives ODER)-Eingang 7 (Typ: adapter::types::unidirectional::AL)
- **IN8**: XOR (exklusives ODER)-Eingang 8 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 8 Eingangsadapter (`IN1` ... `IN8`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 8 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den

Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_XOR_8 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_8`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_XOR_8` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AL_XOR_8` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AL_XOR_8_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

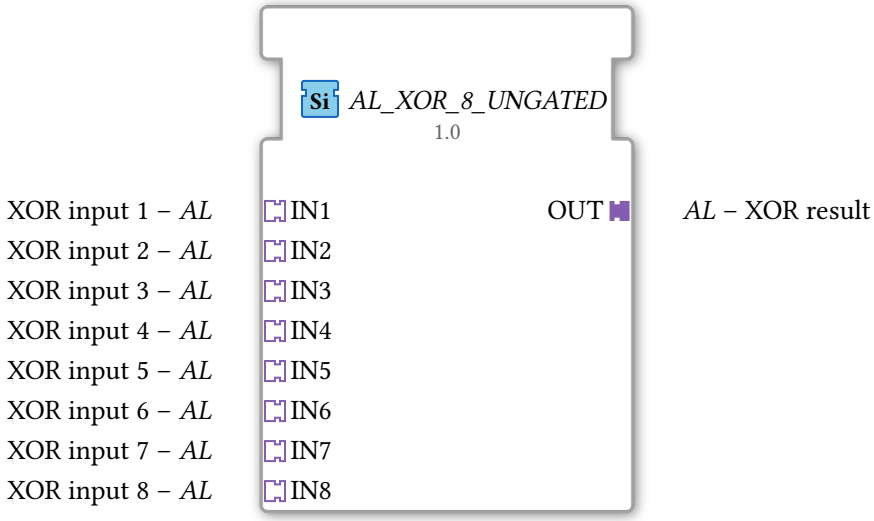
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AL_XOR_8** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AL_XOR_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AL_XOR_8. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AL_XOR_8_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 8 Eingangswerten vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AL)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AL)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AL)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AL)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AL)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AL)
- **IN7**: XOR (exklusives ODER)-Eingang 7 (Typ: adapter::types::unidirectional::AL)
- **IN8**: XOR (exklusives ODER)-Eingang 8 (Typ: adapter::types::unidirectional::AL)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AL)

Funktionsweise

Sobald an einem der 8 Eingangsadapter (`IN1` ... `IN8`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 8 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0

(Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AL_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `LWORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AL` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AL_XOR_8_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `LWORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `LWORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_8`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AL_XOR_8_UNGATED` auf dem vollständigen Bitmuster eines `LWORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AL_XOR_8_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AL_XOR_8`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

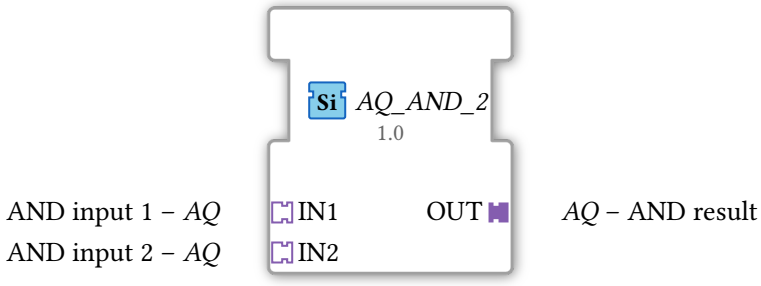
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AL_XOR_8_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `LWORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_AND_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_AND_2** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 2 Eingangswerten vom Typ `BYTE` (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_AND`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 2 Eingangsadapter (`IN1` … `IN2`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_AND_2 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu AX_AND_2, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet AQ_AND_2 auf dem vollständigen Bitmuster eines BYTE-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet AQ_AND_2 Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AQ_AND_2_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

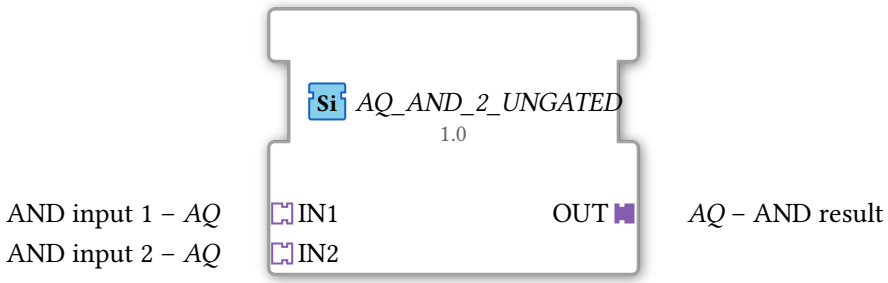
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AQ_AND_2** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für BYTE-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_AND_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AQ_AND_2. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_AND_2_UNGATED** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 2 Eingangswerten vom Typ BYTE (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_AND-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 2 Eingangsadapter (`IN1` ... `IN2`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_AND_2_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_AND_2`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AQ_AND_2_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet `AQ_AND_2_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AQ_AND_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

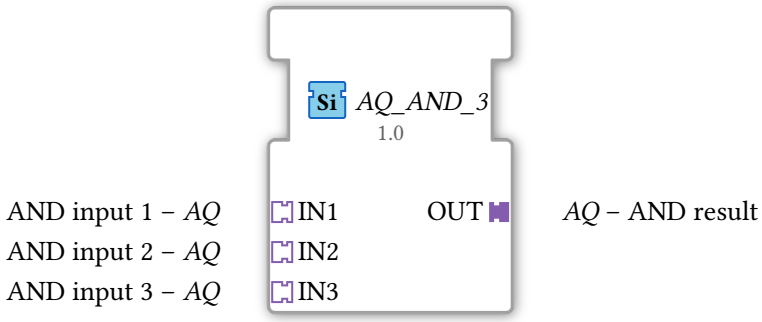
Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AQ_AND_2_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig

einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_AND_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_AND_3** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 3 Eingangswerten vom Typ `BYTE` (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_AND`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: UND-Eingang 3 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` … `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.

- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_AND_3 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_AND_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AQ_AND_3` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet `AQ_AND_3` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AQ_AND_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

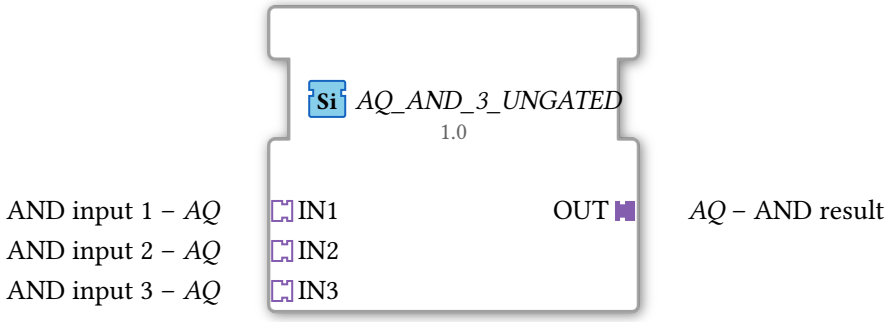
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AQ_AND_3** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_AND_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AQ_AND_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_AND_3_UNGATED** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 3 Eingangswerten vom Typ `BYTE` (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_AND`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: UND-Eingang 3 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` … `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.

- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_AND_3_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_AND_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AQ_AND_3_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet `AQ_AND_3_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AQ_AND_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

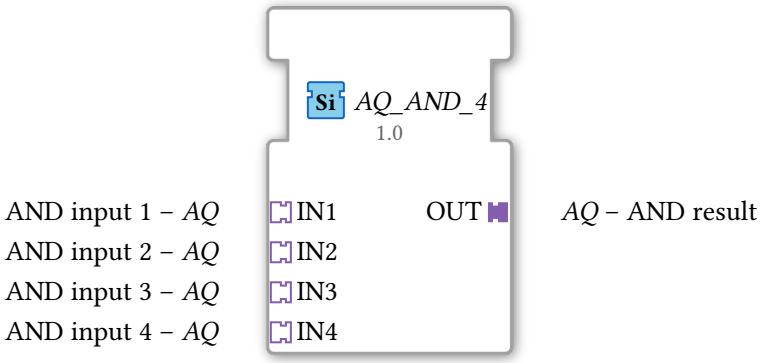
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AQ_AND_3_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_AND_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_AND_4** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 4 Eingangswerten vom Typ `BYTE` (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_AND`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: UND-Eingang 3 (Typ: adapter::types::unidirectional::AQ)
- **IN4**: UND-Eingang 4 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise
Sobald an einem der 4 Eingangsadapter (`IN1` … `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten
- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_AND_4 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien
- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen
Im Gegensatz zu `AX_AND_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AQ_AND_4` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet

AQ_AND_4 Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AQ_AND_4_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

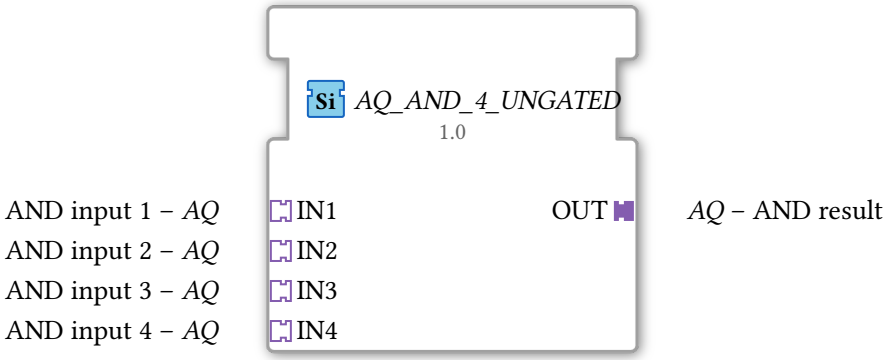
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AQ_AND_4** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für BYTE-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_AND_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AQ_AND_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_AND_4_UNGATED** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 4 Eingangswerten vom Typ BYTE (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_AND-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: UND-Eingang 3 (Typ: adapter::types::unidirectional::AQ)
- **IN4**: UND-Eingang 4 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 4 Eingangsadapter (`IN1` ... `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_AND_4_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_AND_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AQ_AND_4_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet `AQ_AND_4_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AQ_AND_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

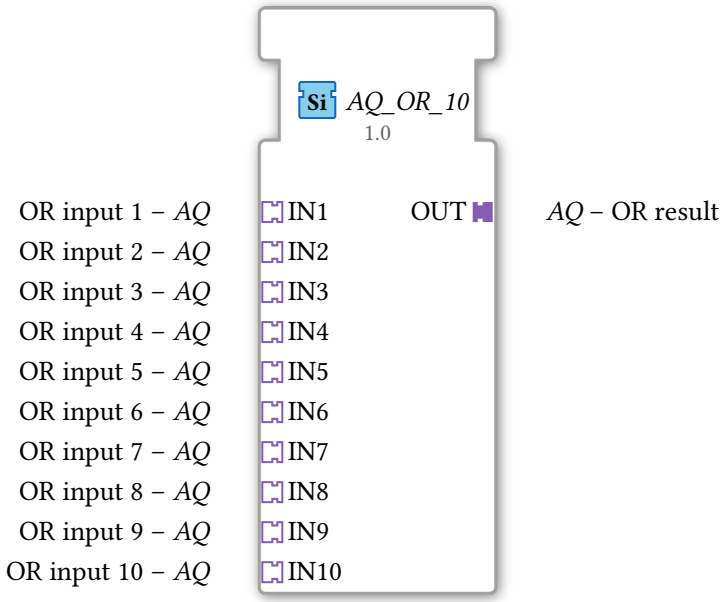
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AQ_AND_4_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für BYTE-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_OR_10



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_OR_10** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 10 Eingangswerten vom Typ BYTE (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_OR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AQ)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AQ)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AQ)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AQ)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AQ)

- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AQ)
- **IN9**: ODER-Eingang 9 (Typ: adapter::types::unidirectional::AQ)
- **IN10**: ODER-Eingang 10 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 10 Eingangsadapter (`IN1` … `IN10`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 10 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_OR_10 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_10`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AQ_OR_10` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AQ_OR_10` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AQ_OR_10_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AQ_OR_10** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_OR_10_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `AQ_OR_10`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_OR_10_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 10 Eingangswerten vom Typ `BYTE` (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AQ)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AQ)

- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AQ)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AQ)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AQ)
- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AQ)
- **IN9**: ODER-Eingang 9 (Typ: adapter::types::unidirectional::AQ)
- **IN10**: ODER-Eingang 10 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 10 Eingangsadapter (`IN1` … `IN10`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 10 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_OR_10_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_10`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AQ_OR_10_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AQ_OR_10_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AQ_OR_10`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

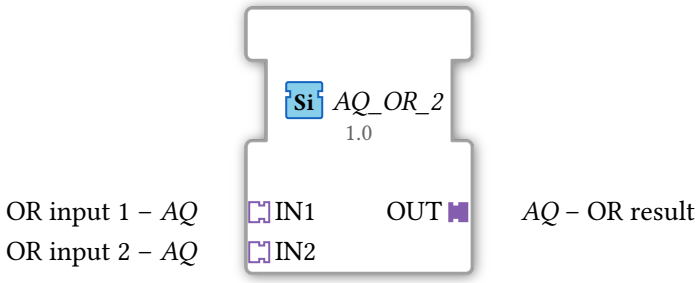
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AQ_OR_10_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_OR_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_OR_2** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 2 Eingangswerten vom Typ `BYTE` (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 2 Eingangsadapter (`IN1` ... `IN2`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_OR_2 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_2`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AQ_OR_2` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AQ_OR_2` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AQ_OR_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

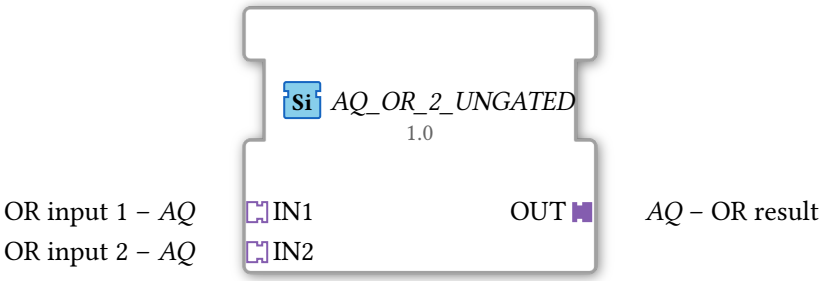
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AQ_OR_2** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_OR_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AQ_OR_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_OR_2_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 2 Eingangswerten vom Typ `BYTE` (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur boole-

schen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_OR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise
Sobald an einem der 2 Eingangsadapter (IN1 ... IN2) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter OUT. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf OUT gehaltenen Wert unterscheidet, wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (GEN_AQ_OR) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen AX_OR-Bausteinen wird hier jedes Bit des BYTE-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ unidirectional::AQ – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_OR_2_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ BYTE zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer BYTE-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ **Vergleich mit ähnlichen Bausteinen**
Im Gegensatz zu AX_OR_2, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet AQ_OR_2_UNGATED auf dem vollständigen Bitmuster eines BYTE-Werts. Verglichen mit dem Standard-Baustein OR_2 verwen-

det AQ_OR_2_UNGATED Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- AQ_OR_2: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

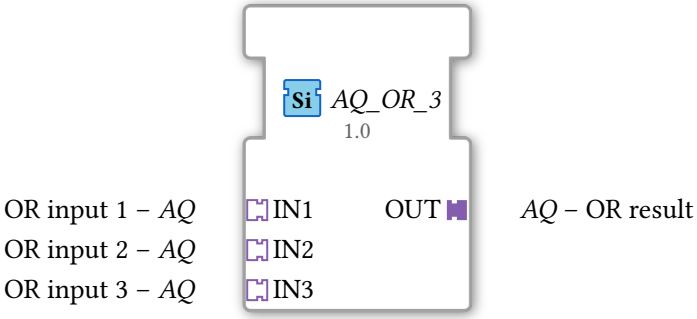
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AQ_OR_2_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für BYTE-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_OR_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_OR_3** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 3 Eingangswerten vom Typ BYTE (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_OR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` … `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_OR_3 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AQ_OR_3` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AQ_OR_3` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AQ_OR_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

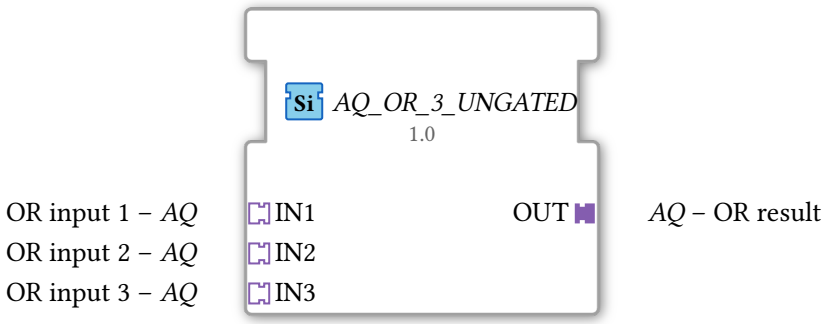
Fazit

Der **AQ_OR_3** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_OR_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AQ_OR_3. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht

gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_OR_3_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 3 Eingangswerten vom Typ `BYTE` (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` … `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.

- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_OR_3_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AQ_OR_3_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AQ_OR_3_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AQ_OR_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

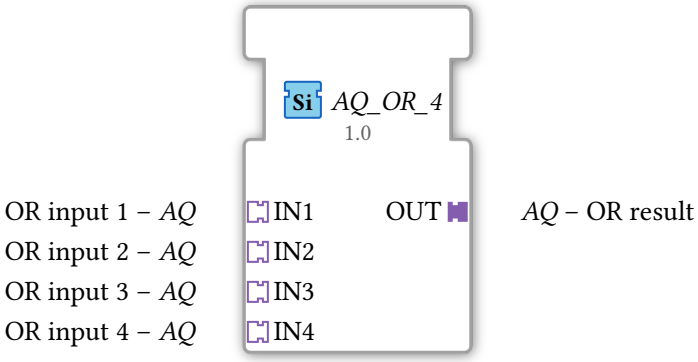
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AQ_OR_3_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_OR_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_OR_4** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 4 Eingangswerten vom Typ `BYTE` (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AQ)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise
Sobald an einem der 4 Eingangsadapter (`IN1` … `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten
- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_OR_4 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien
- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen
Im Gegensatz zu `AX_OR_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AQ_OR_4` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet

AQ_OR_4 Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AQ_OR_4_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

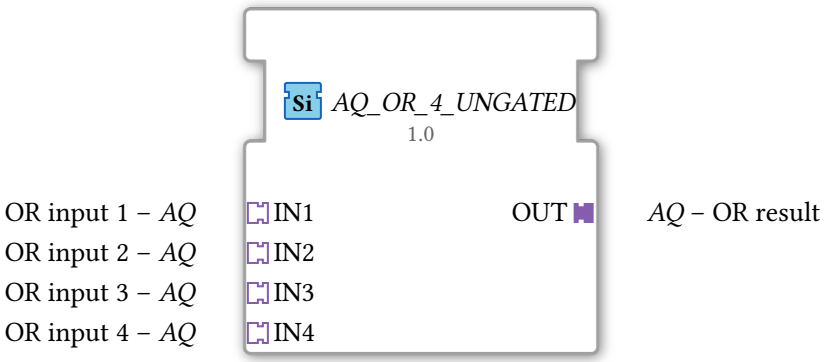
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AQ_OR_4** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für BYTE-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_OR_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AQ_OR_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_OR_4_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 4 Eingangswerten vom Typ BYTE (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_OR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AQ)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 4 Eingangsadapter (`IN1` ... `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_OR_4_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AQ_OR_4_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AQ_OR_4_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AQ_OR_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

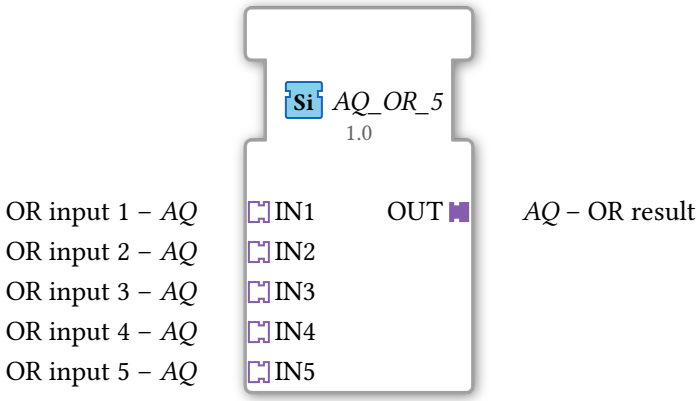
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AQ_OR_4_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_OR_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_OR_5** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 5 Eingangswerten vom Typ `BYTE` (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AQ)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AQ)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 5 Eingangsadapter (`IN1` ... `IN5`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 5 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-

Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf OUT gehaltenen Wert unterscheidet, wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (GEN_AQ_OR) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen AX_OR-Bausteinen wird hier jedes Bit des BYTE-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ unidirectional::AQ – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_OR_5 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ BYTE zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer BYTE-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu AX_OR_5, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet AQ_OR_5 auf dem vollständigen Bitmuster eines BYTE-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet AQ_OR_5 Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AQ_OR_5_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

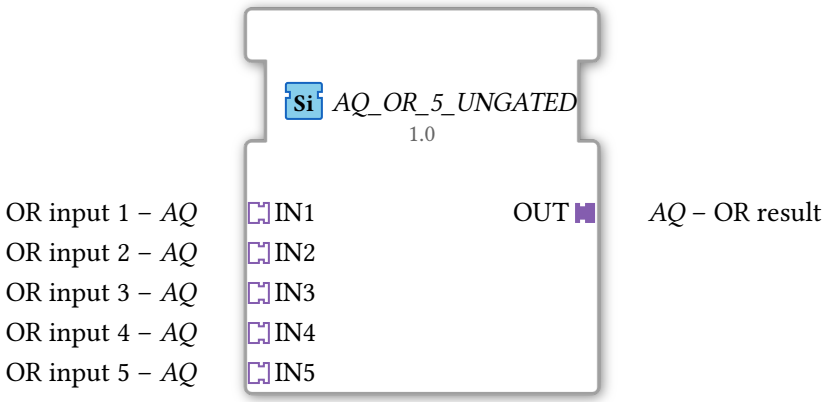
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AQ_OR_5** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für BYTE-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_OR_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AQ_OR_5. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_OR_5_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 5 Eingangswerten vom Typ `BYTE` (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AQ)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AQ)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 5 Eingangsadapter (`IN1` … `IN5`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 5 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_OR_5_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_5`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AQ_OR_5_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AQ_OR_5_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AQ_OR_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

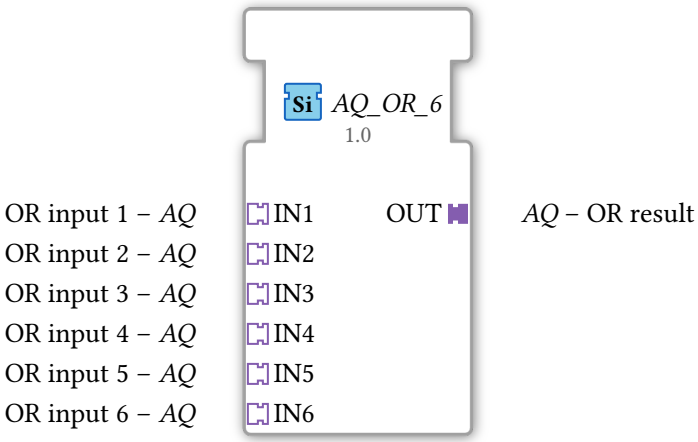
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AQ_OR_5_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_OR_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_OR_6** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 6 Eingangswerten vom Typ `BYTE` (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AQ)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AQ)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AQ)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 6 Eingangsadapter (`IN1` … `IN6`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 6 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_OR_6 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.

- **Signalaggregation**: Zusammenführen mehrerer BYTE-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu AX_OR_6, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet AQ_OR_6 auf dem vollständigen Bitmuster eines BYTE-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet AQ_OR_6 Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AQ_OR_6_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

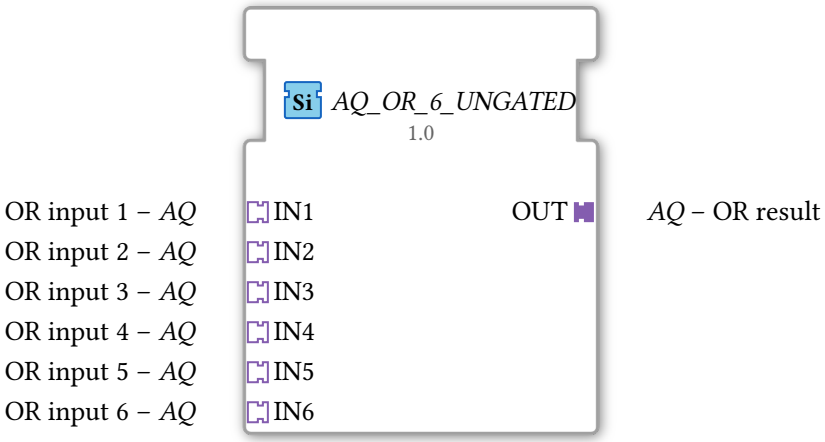
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AQ_OR_6** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für BYTE-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_OR_6_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AQ_OR_6. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_OR_6_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 6 Eingangswerten vom Typ BYTE (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_OR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AQ)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AQ)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AQ)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise
Sobald an einem der 6 Eingangsadapter (`IN1` … `IN6`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 6 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_OR_6_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu AX_OR_6, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet AQ_OR_6_UNGATED auf dem vollständigen Bitmuster eines BYTE-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet AQ_OR_6_UNGATED Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- AQ_OR_6: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

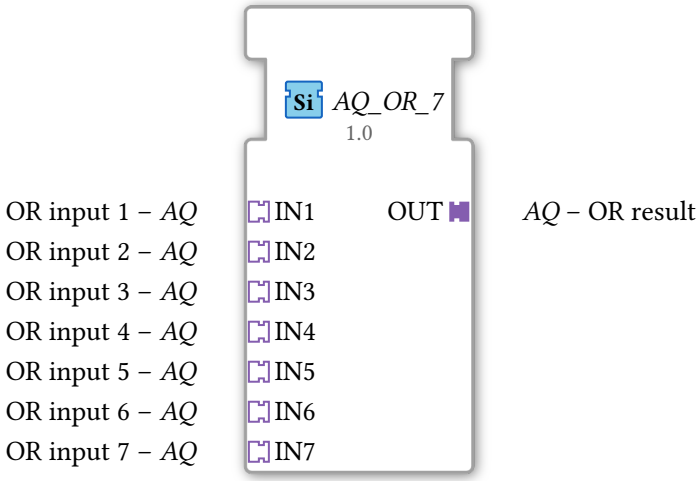
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AQ_OR_6_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für BYTE-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_OR_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_OR_7** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 7 Eingangswerten vom Typ BYTE (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_OR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AQ)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AQ)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AQ)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AQ)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 7 Eingangsadapter (`IN1` ... `IN7`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 7 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_OR_7 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_7`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AQ_OR_7` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AQ_OR_7` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AQ_OR_7_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

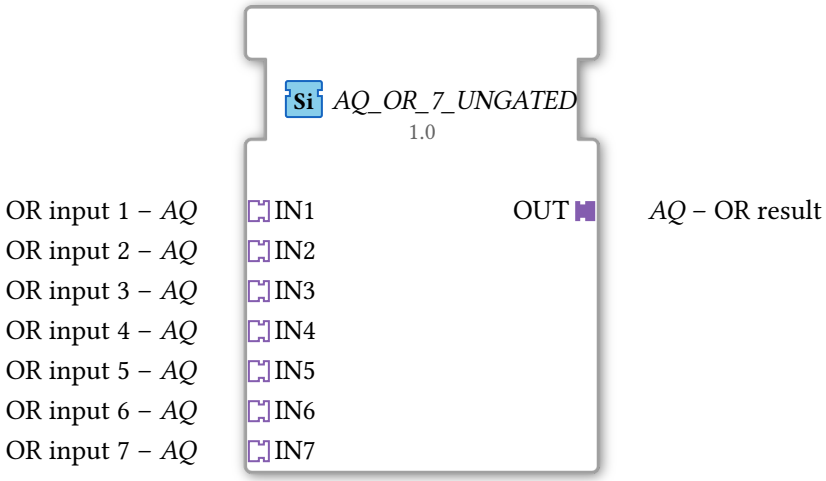
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AQ_OR_7** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für BYTE-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_OR_7_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AQ_OR_7. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_OR_7_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 7 Eingangswerten vom Typ BYTE (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_OR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AQ)

- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AQ)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AQ)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AQ)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AQ)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 7 Eingangsadapter (`IN1` ... `IN7`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 7 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_OR_7_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_7`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AQ_OR_7_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AQ_OR_7_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AQ_OR_7`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

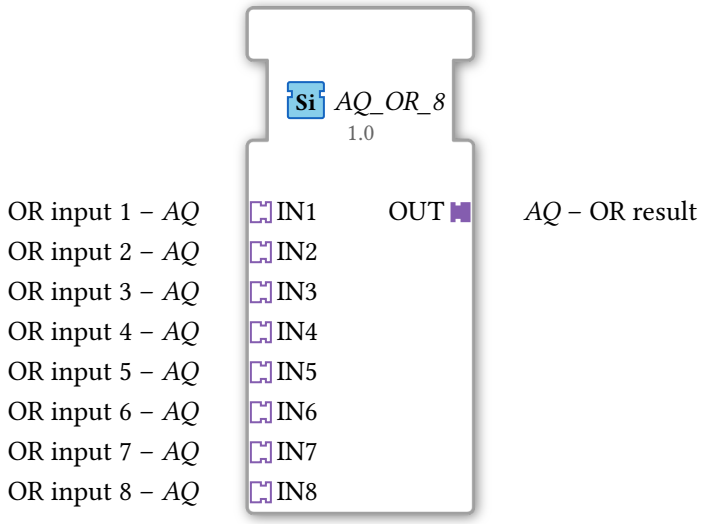
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AQ_OR_7_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_OR_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_OR_8** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 8 Eingangswerten vom Typ `BYTE` (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AQ)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AQ)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AQ)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AQ)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AQ)
- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 8 Eingangsadapter (`IN1` ... `IN8`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 8 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`.

Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf OUT gehaltenen Wert unterscheidet, wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (GEN_AQ_OR) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen AX_OR-Bausteinen wird hier jedes Bit des BYTE-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ unidirectional::AQ – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_OR_8 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ BYTE zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer BYTE-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu AX_OR_8, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet AQ_OR_8 auf dem vollständigen Bitmuster eines BYTE-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet AQ_OR_8 Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AQ_OR_8_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

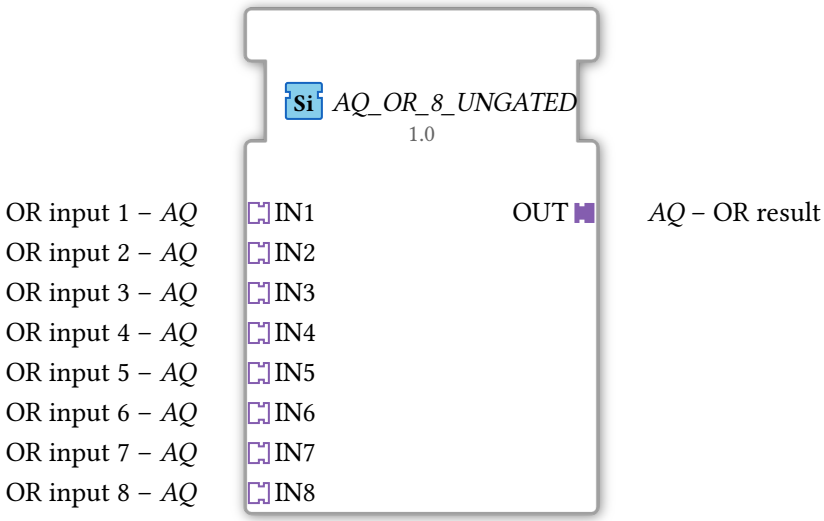
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AQ_OR_8** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für BYTE-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_OR_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AQ_OR_8. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_OR_8_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 8 Eingangswerten vom Typ `BYTE` (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AQ)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AQ)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AQ)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AQ)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AQ)
- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 8 Eingangsadapter (`IN1` … `IN8`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 8 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-

Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_OR_8_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_8`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AQ_OR_8_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AQ_OR_8_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AQ_OR_8`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

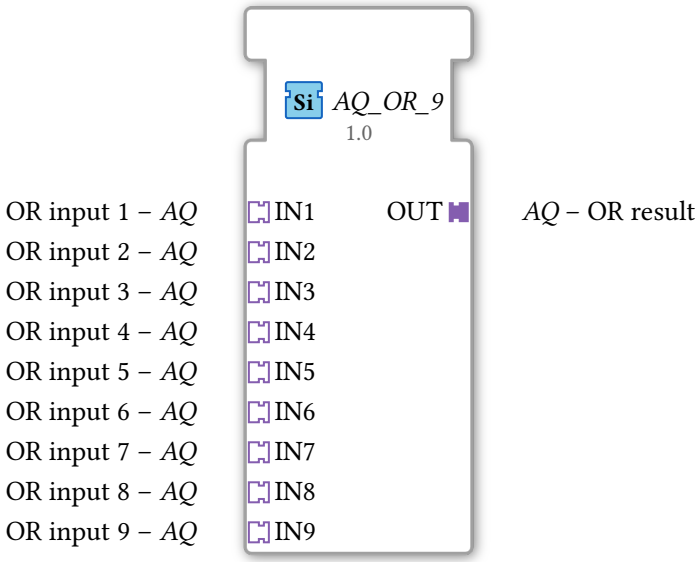
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AQ_OR_8_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_OR_9



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_OR_9** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 9 Eingangswerten vom Typ `BYTE` (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AQ)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AQ)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AQ)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AQ)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AQ)
- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AQ)
- **IN9**: ODER-Eingang 9 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 9 Eingangsadapter (`IN1` … `IN9`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 9 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_OR_9 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_9`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AQ_OR_9` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AQ_OR_9` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AQ_OR_9_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

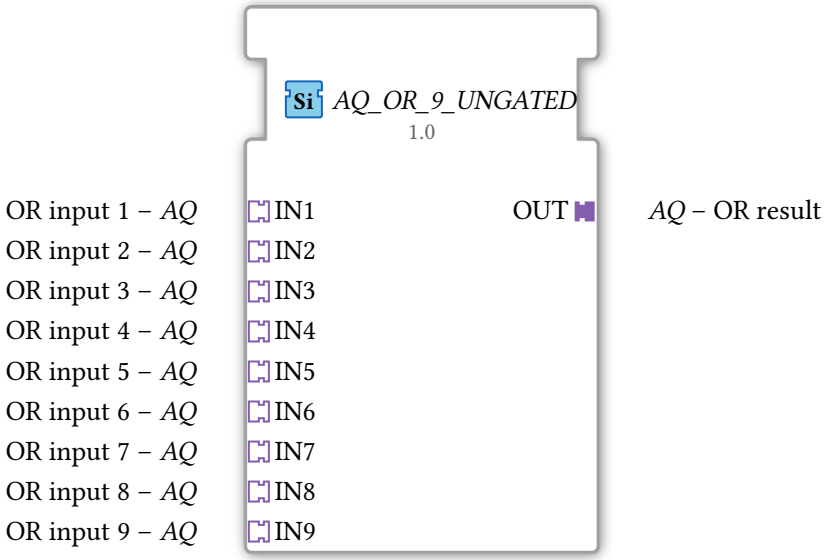
Fazit

Der **AQ_OR_9** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_OR_9_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AQ_OR_9. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht

gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_OR_9_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 9 Eingangswerten vom Typ `BYTE` (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AQ)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AQ)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AQ)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AQ)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AQ)
- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AQ)
- **IN9**: ODER-Eingang 9 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 9 Eingangsadapter (`IN1` … `IN9`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 9 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_OR_9_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_9`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AQ_OR_9_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AQ_OR_9_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AQ_OR_9`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

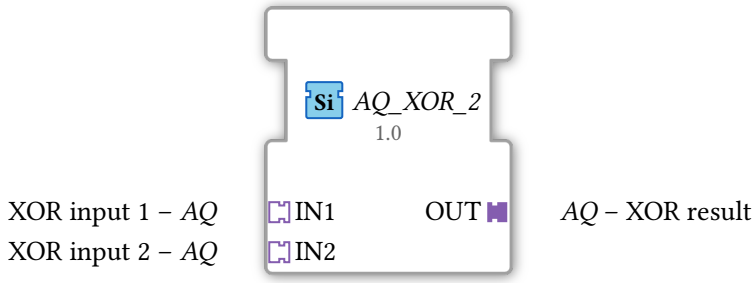
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AQ_OR_9_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_XOR_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_XOR_2** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 2 Eingangswerten vom Typ `BYTE` (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 2 Eingangsadapter (`IN1` ... `IN2`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_XOR_2 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_2`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AQ_XOR_2` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AQ_XOR_2` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AQ_XOR_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

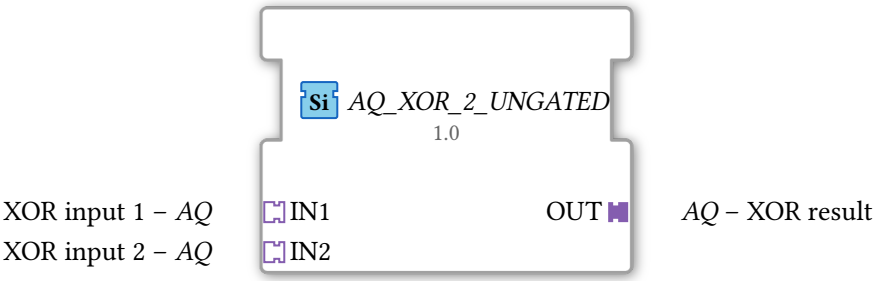
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AQ_XOR_2** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_XOR_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AQ_XOR_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_XOR_2_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 2 Eingangswerten vom Typ `BYTE` (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im

Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_XOR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise
Sobald an einem der 2 Eingangsadapter (IN1 ... IN2) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter OUT. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf OUT gehaltenen Wert unterscheidet, wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (GEN_AQ_XOR) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen AX_XOR-Bausteinen wird hier jedes Bit des BYTE-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ unidirectional::AQ – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_XOR_2_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ BYTE zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer BYTE-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen
Im Gegensatz zu AX_XOR_2, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet AQ_XOR_2_UNGATED auf dem vollständigen Bitmuster eines BYTE-Werts. Verglichen mit dem Standard-Baustein XOR_2

verwendet AQ_XOR_2_UNGATED Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- AQ_XOR_2: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

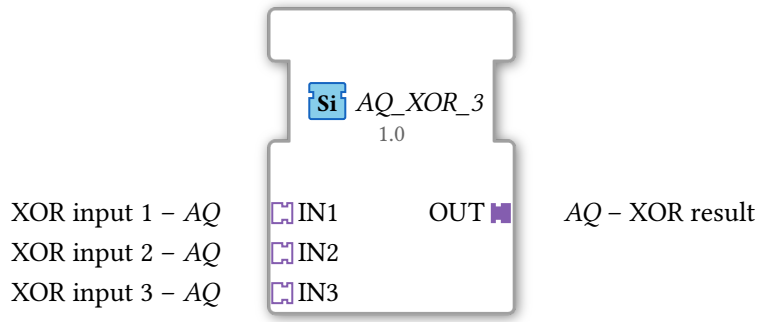
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AQ_XOR_2_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für BYTE-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_XOR_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_XOR_3** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 3 Eingangswerten vom Typ BYTE (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_XOR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` … `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_XOR_3 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AQ_XOR_3` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AQ_XOR_3` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AQ_XOR_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

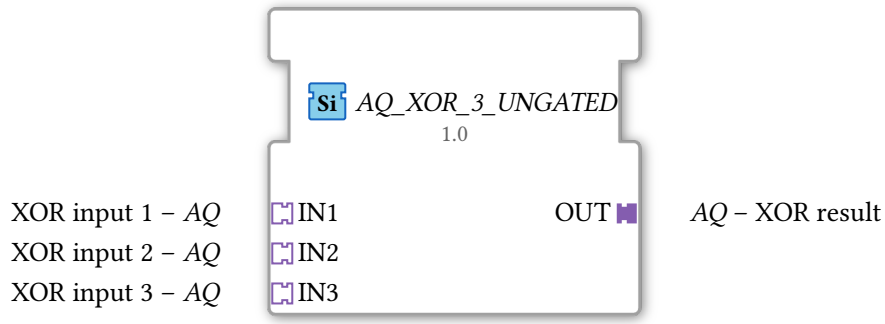
Fazit

Der **AQ_XOR_3** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_XOR_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AQ_XOR_3. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht

gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_XOR_3_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 3 Eingangswerten vom Typ `BYTE` (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` … `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.

- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_XOR_3_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AQ_XOR_3_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AQ_XOR_3_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AQ_XOR_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

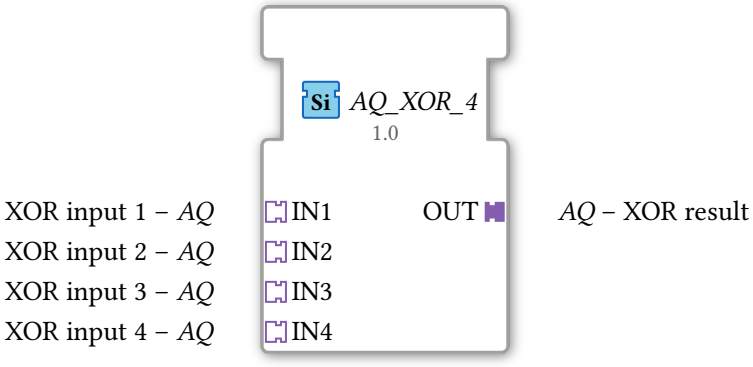
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AQ_XOR_3_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_XOR_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_XOR_4** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 4 Eingangswerten vom Typ `BYTE` (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AQ)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise
Sobald an einem der 4 Eingangsadapter (`IN1` … `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten
- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_XOR_4 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien
- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen
Im Gegensatz zu `AX_XOR_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AQ_XOR_4` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet

AQ_XOR_4 Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AQ_XOR_4_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

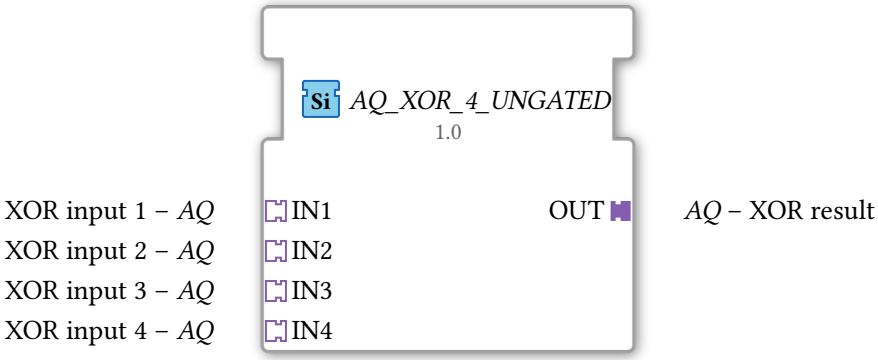
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AQ_XOR_4** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für BYTE-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_XOR_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AQ_XOR_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_XOR_4_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 4 Eingangswerten vom Typ BYTE (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_XOR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AQ)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 4 Eingangsadapter (`IN1` ... `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_XOR_4_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AQ_XOR_4_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AQ_XOR_4_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AQ_XOR_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AQ_XOR_4_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_XOR_5

• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_XOR_5** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 5 Eingangswerten vom Typ `BYTE` (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AQ)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AQ)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 5 Eingangsadapter (`IN1` ... `IN5`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 5 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0`

(Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_XOR_5 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_5`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AQ_XOR_5` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AQ_XOR_5` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AQ_XOR_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

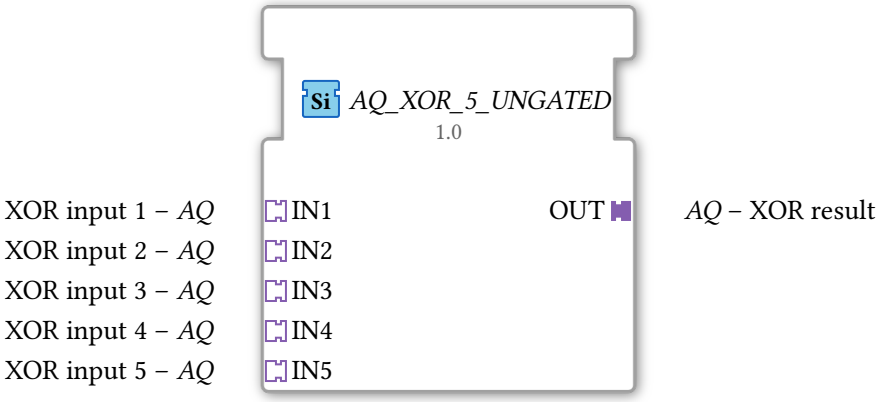
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AQ_XOR_5** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_XOR_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AQ_XOR_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_XOR_5_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 5 Eingangswerten vom Typ `BYTE` (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AQ)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AQ)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 5 Eingangsadapter (`IN1` … `IN5`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 5 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_XOR_5_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_5`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AQ_XOR_5_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AQ_XOR_5_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AQ_XOR_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

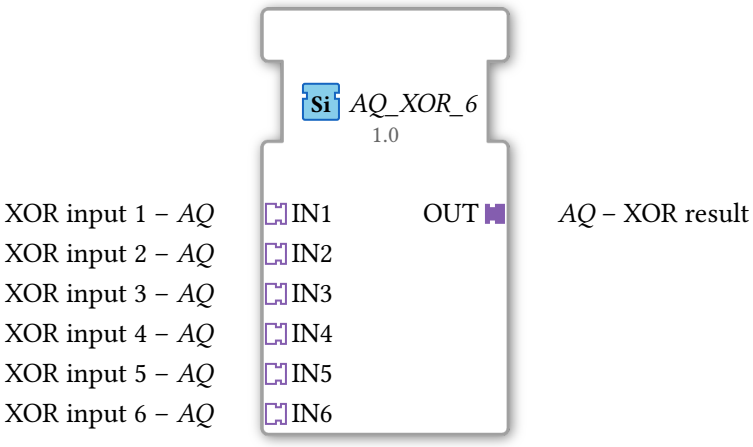
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AQ_XOR_5_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_XOR_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_XOR_6** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 6 Eingangswerten vom Typ `BYTE` (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AQ)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AQ)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AQ)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 6 Eingangsadapter (`IN1` … `IN6`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 6 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_XOR_6 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.

- **Signalaggregation**: Zusammenführen mehrerer BYTE-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu AX_XOR_6, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet AQ_XOR_6 auf dem vollständigen Bitmuster eines BYTE-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet AQ_XOR_6 Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- AQ_XOR_6_UNGATED: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

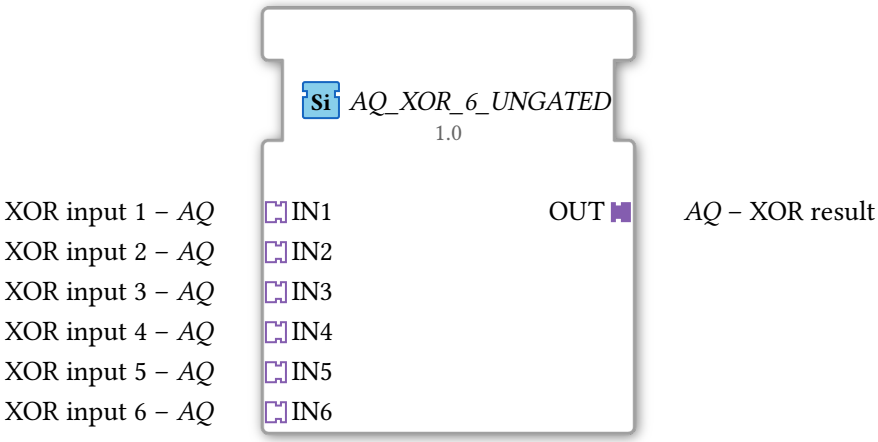
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AQ_XOR_6** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für BYTE-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_XOR_6_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AQ_XOR_6. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_XOR_6_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 6 Eingangswerten vom Typ BYTE (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_XOR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AQ)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AQ)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AQ)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise
Sobald an einem der 6 Eingangsadapter (`IN1` … `IN6`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 6 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_XOR_6_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu AX_XOR_6, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet AQ_XOR_6_UNGATED auf dem vollständigen Bitmuster eines BYTE-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet AQ_XOR_6_UNGATED Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AQ_XOR_6**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

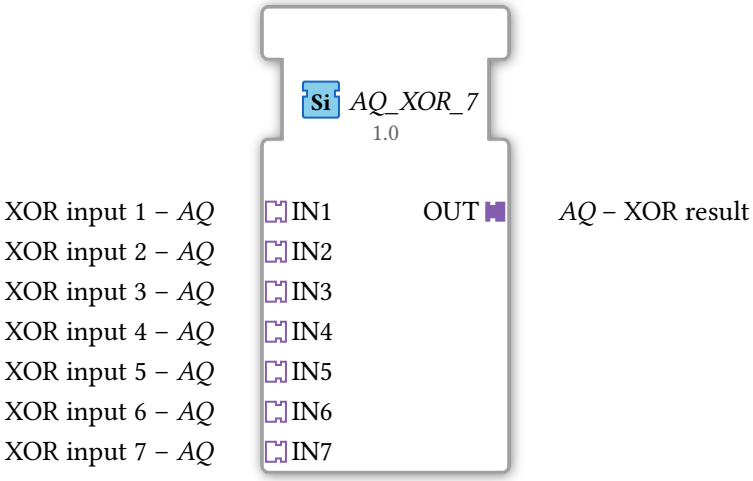
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AQ_XOR_6_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für BYTE-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_XOR_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_XOR_7** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 7 Eingangswerten vom Typ BYTE (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_XOR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AQ)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AQ)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AQ)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AQ)
- **IN7**: XOR (exklusives ODER)-Eingang 7 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 7 Eingangsadapter (`IN1` ... `IN7`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 7 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_XOR_7 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_7`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AQ_XOR_7` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AQ_XOR_7` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AQ_XOR_7_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

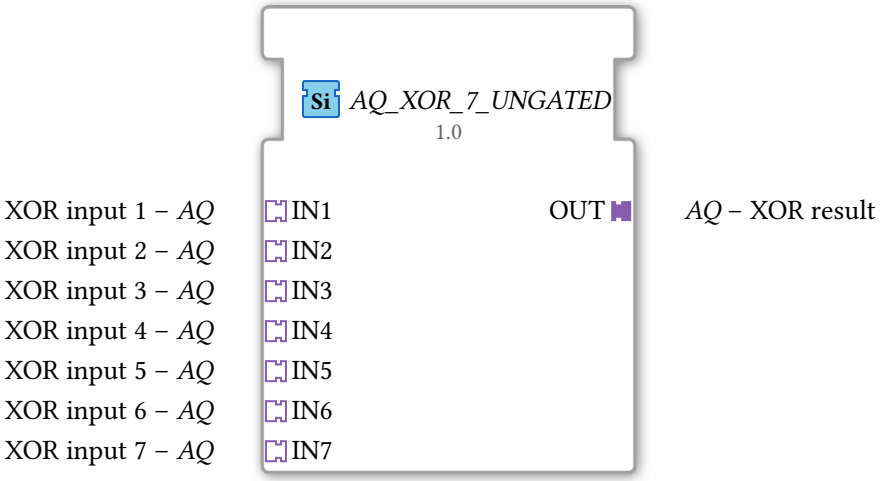
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AQ_XOR_7** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für BYTE-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_XOR_7_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AQ_XOR_7. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_XOR_7_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 7 Eingangswerten vom Typ BYTE (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_XOR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AQ)

- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AQ)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AQ)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AQ)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AQ)
- **IN7**: XOR (exklusives ODER)-Eingang 7 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 7 Eingangsadapter (`IN1` ... `IN7`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 7 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_XOR_7_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_7`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AQ_XOR_7_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AQ_XOR_7_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AQ_XOR_7`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

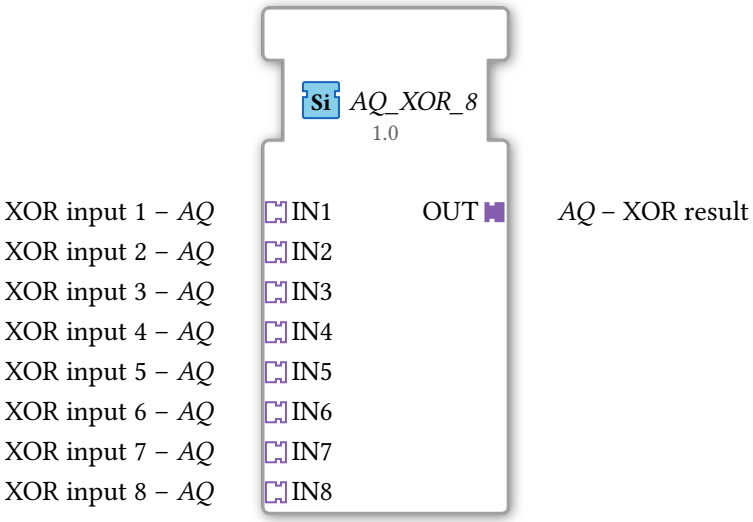
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AQ_XOR_7_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_XOR_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_XOR_8** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 8 Eingangswerten vom Typ `BYTE` (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AQ)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AQ)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AQ)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AQ)
- **IN7**: XOR (exklusives ODER)-Eingang 7 (Typ: adapter::types::unidirectional::AQ)
- **IN8**: XOR (exklusives ODER)-Eingang 8 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 8 Eingangsadapter (`IN1` ... `IN8`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 8 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den

Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_XOR_8 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_8`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AQ_XOR_8` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AQ_XOR_8` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AQ_XOR_8_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

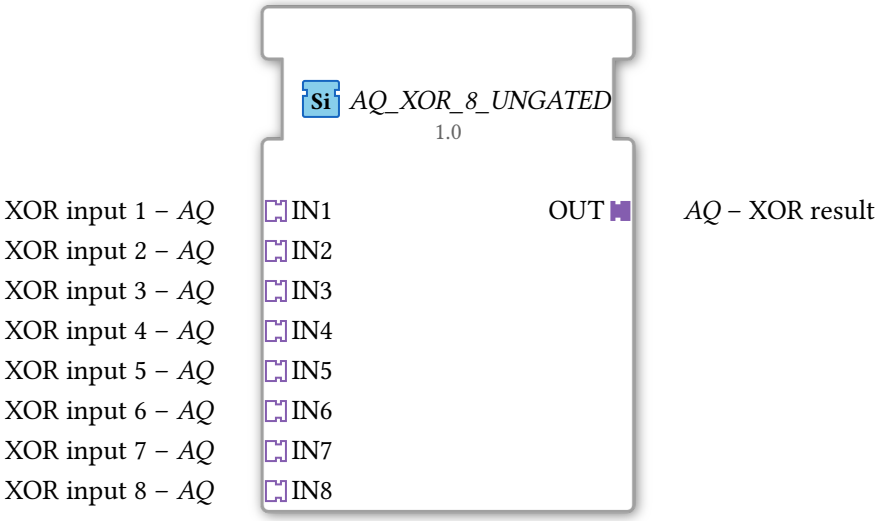
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AQ_XOR_8** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AQ_XOR_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AQ_XOR_8`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AQ_XOR_8_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 8 Eingangswerten vom Typ `BYTE` (2-Bit-Wert (Viertel-Byte, als Byte übertragen)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AQ)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AQ)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AQ)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AQ)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AQ)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AQ)
- **IN7**: XOR (exklusives ODER)-Eingang 7 (Typ: adapter::types::unidirectional::AQ)
- **IN8**: XOR (exklusives ODER)-Eingang 8 (Typ: adapter::types::unidirectional::AQ)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AQ)

Funktionsweise

Sobald an einem der 8 Eingangsadapter (`IN1` … `IN8`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 8 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0

(Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AQ_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `BYTE`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AQ` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AQ_XOR_8_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `BYTE` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `BYTE`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_8`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AQ_XOR_8_UNGATED` auf dem vollständigen Bitmuster eines `BYTE`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AQ_XOR_8_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AQ_XOR_8`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

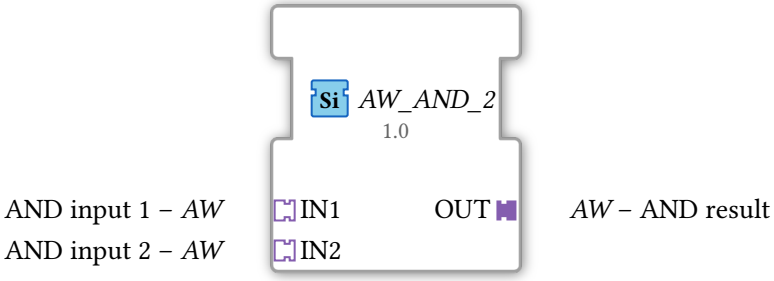
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AQ_XOR_8_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `BYTE`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_AND_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_AND_2** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 2 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_AND`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 2 Eingangsadapter (`IN1` … `IN2`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_AND_2 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu AX_AND_2, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet AW_AND_2 auf dem vollständigen Bitmuster eines WORD-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet AW_AND_2 Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AW_AND_2_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

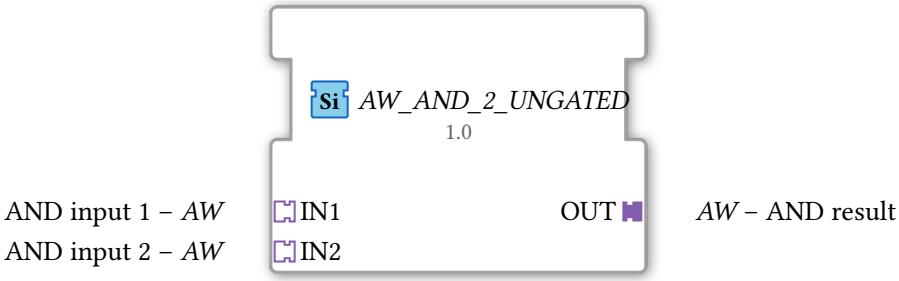
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AW_AND_2** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für WORD-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_AND_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AW_AND_2. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_AND_2_UNGATED** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 2 Eingangswerten vom Typ WORD (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_AND-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 2 Eingangsadapter (`IN1` ... `IN2`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_AND_2_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_AND_2`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_AND_2_UNGATED` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet `AW_AND_2_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AW_AND_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

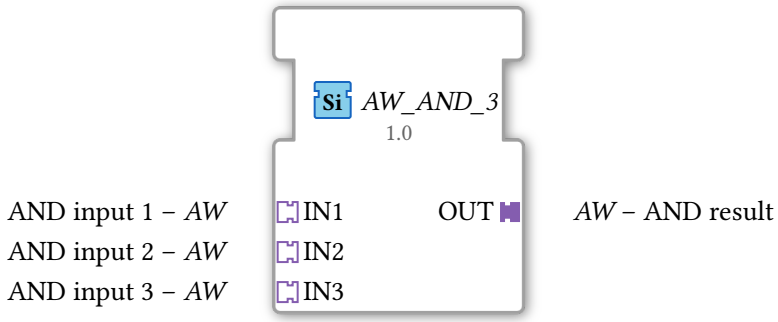
Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AW_AND_2_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig

einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_AND_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_AND_3** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 3 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_AND`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: UND-Eingang 3 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` … `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.

- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_AND_3 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_AND_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_AND_3` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet `AW_AND_3` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AW_AND_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

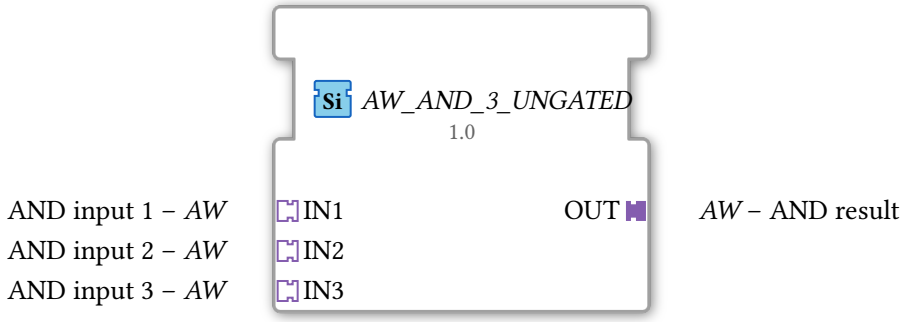
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AW_AND_3** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_AND_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AW_AND_3. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_AND_3_UNGATED** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 3 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_AND`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: UND-Eingang 3 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` … `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.

- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_AND_3_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_AND_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_AND_3_UNGATED` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet `AW_AND_3_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AW_AND_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

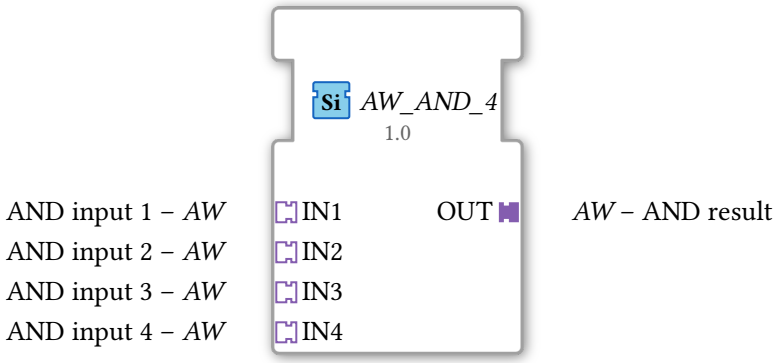
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AW_AND_3_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_AND_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_AND_4** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 4 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_AND`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: UND-Eingang 3 (Typ: adapter::types::unidirectional::AW)
- **IN4**: UND-Eingang 4 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise
Sobald an einem der 4 Eingangsadapter (`IN1` … `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten
- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_AND_4 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien
- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen
Im Gegensatz zu `AX_AND_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_AND_4` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet

AW_AND_4 Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AW_AND_4_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

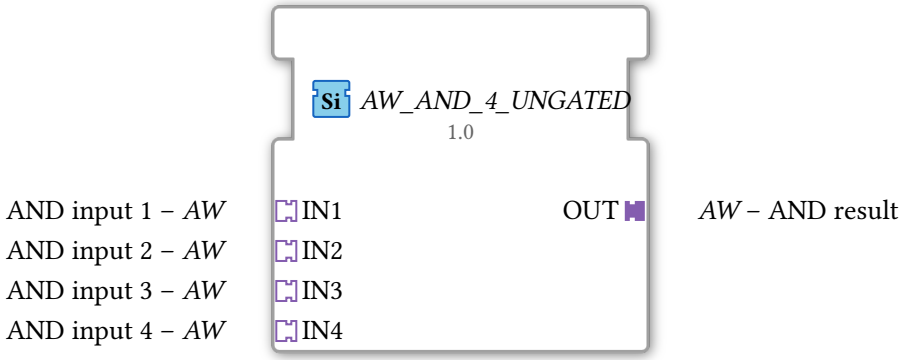
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AW_AND_4** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für WORD-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_AND_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AW_AND_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_AND_4_UNGATED** ist ein generischer Funktionsblock zur bitweisen UND-Verknüpfung von 4 Eingangswerten vom Typ WORD (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_AND-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: UND-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: UND-Eingang 3 (Typ: adapter::types::unidirectional::AW)
- **IN4**: UND-Eingang 4 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 4 Eingangsadapter (`IN1` ... `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **UND** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gesetzt (Einselement der UND-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_AND`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 4 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_AND`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_AND_4_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame UND-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_AND_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_AND_4_UNGATED` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein AND_2 verwendet `AW_AND_4_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AW_AND_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

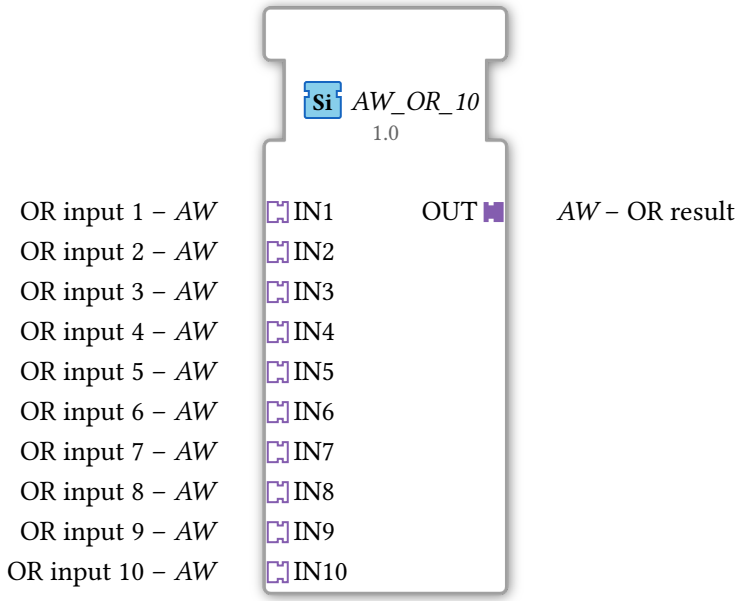
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AW_AND_4_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen UND-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_OR_10



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_OR_10** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 10 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AW)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AW)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AW)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AW)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AW)

- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AW)
- **IN9**: ODER-Eingang 9 (Typ: adapter::types::unidirectional::AW)
- **IN10**: ODER-Eingang 10 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 10 Eingangsadapter (`IN1` … `IN10`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 10 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_OR_10 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_10`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_OR_10` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AW_OR_10` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AW_OR_10_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AW_OR_10** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_OR_10_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `AW_OR_10`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_OR_10_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 10 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AW)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AW)

- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AW)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AW)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AW)
- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AW)
- **IN9**: ODER-Eingang 9 (Typ: adapter::types::unidirectional::AW)
- **IN10**: ODER-Eingang 10 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 10 Eingangsadapter (`IN1` … `IN10`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 10 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_OR_10_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_10`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_OR_10_UNGATED` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AW_OR_10_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AW_OR_10`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

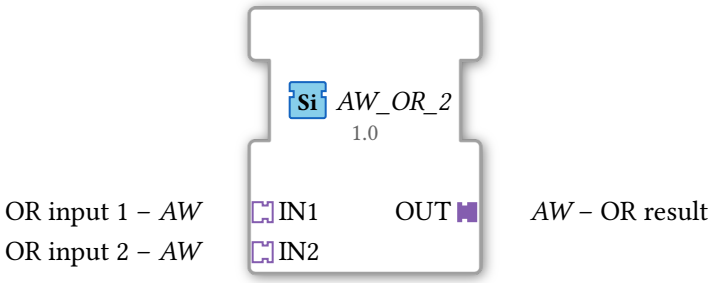
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AW_OR_10_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_OR_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_OR_2** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 2 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 2 Eingangsadapter (`IN1` … `IN2`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_OR_2 keine internen Zuständen. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_2`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_OR_2` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AW_OR_2` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AW_OR_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

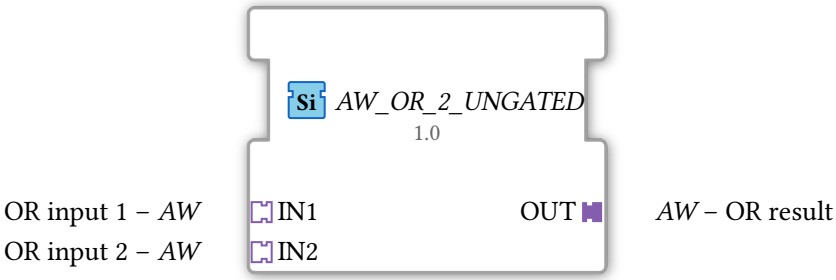
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AW_OR_2** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_OR_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AW_OR_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_OR_2_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 2 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung

einzelner Wahrheitswerte (wie bei den AX_OR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise
Sobald an einem der 2 Eingangsadapter (IN1 … IN2) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter OUT. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf OUT gehaltenen Wert unterscheidet, wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (GEN_AW_OR) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen AX_OR-Bausteinen wird hier jedes Bit des WORD-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ unidirectional::AW – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_OR_2_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ WORD zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer WORD-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ **Vergleich mit ähnlichen Bausteinen**
Im Gegensatz zu AX_OR_2, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet AW_OR_2_UNGATED auf dem vollständigen Bitmuster eines WORD-Werts. Verglichen mit dem Standard-Baustein OR_2 verwen-

det AW_OR_2_UNGATED Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- AW_OR_2: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

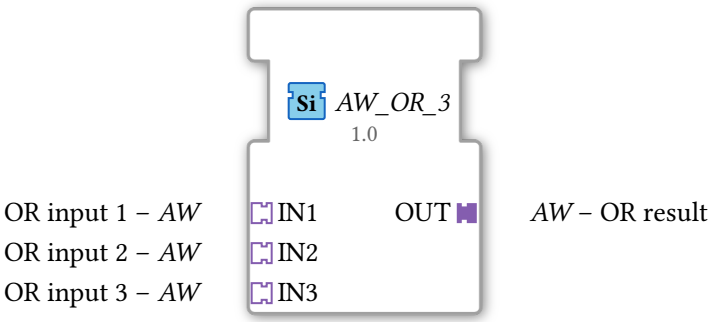
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AW_OR_2_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für WORD-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_OR_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_OR_3** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 3 Eingangswerten vom Typ WORD (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_OR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` … `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_OR_3 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_OR_3` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AW_OR_3` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AW_OR_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

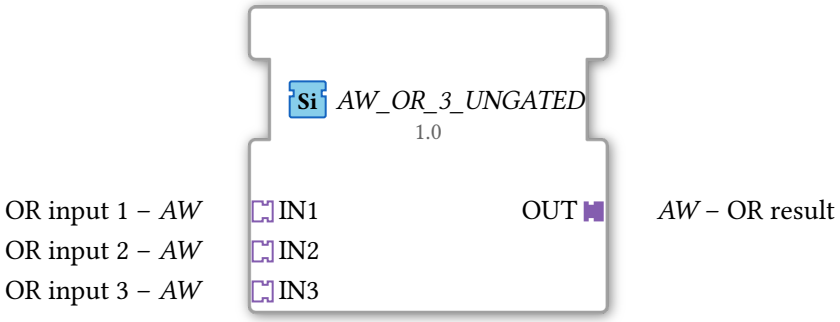
Fazit

Der **AW_OR_3** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_OR_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AW_OR_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht

gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_OR_3_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 3 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` … `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.

- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_OR_3_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_OR_3_UNGATED` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AW_OR_3_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AW_OR_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

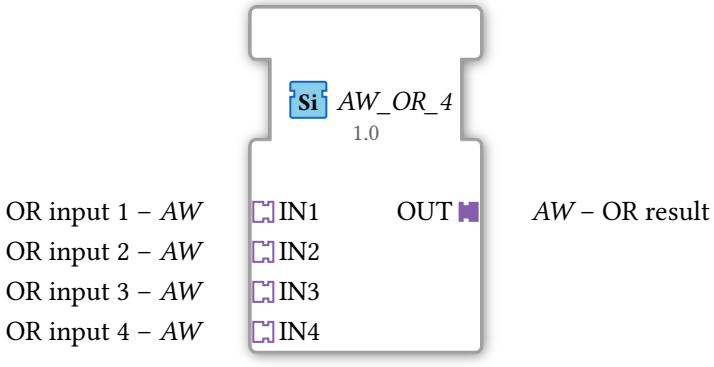
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AW_OR_3_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_OR_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_OR_4** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 4 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AW)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise
Sobald an einem der 4 Eingangsadapter (`IN1` … `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten
- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_OR_4 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien
- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen
Im Gegensatz zu `AX_OR_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_OR_4` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet

AW_OR_4 Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AW_OR_4_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

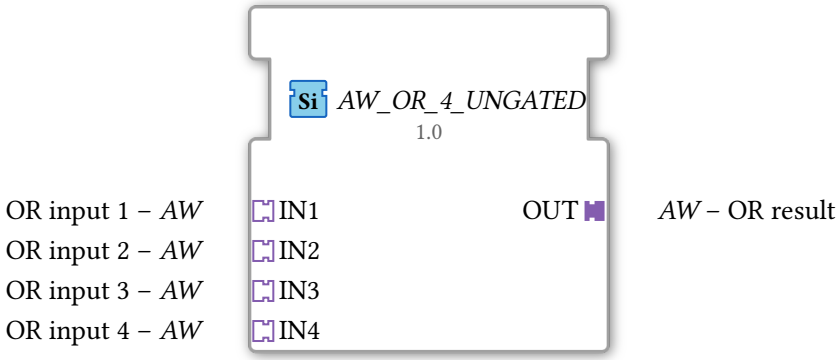
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AW_OR_4** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für WORD-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_OR_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AW_OR_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_OR_4_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 4 Eingangswerten vom Typ WORD (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_OR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AW)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 4 Eingangsadapter (`IN1` ... `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_OR_4_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_OR_4_UNGATED` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AW_OR_4_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AW_OR_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

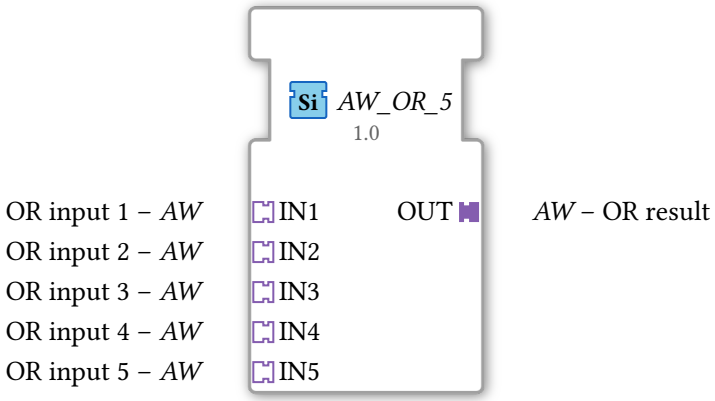
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AW_OR_4_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_OR_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_OR_5** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 5 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AW)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AW)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 5 Eingangsadapter (`IN1` ... `IN5`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 5 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-

Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_OR_5 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_5`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_OR_5` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AW_OR_5` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AW_OR_5_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

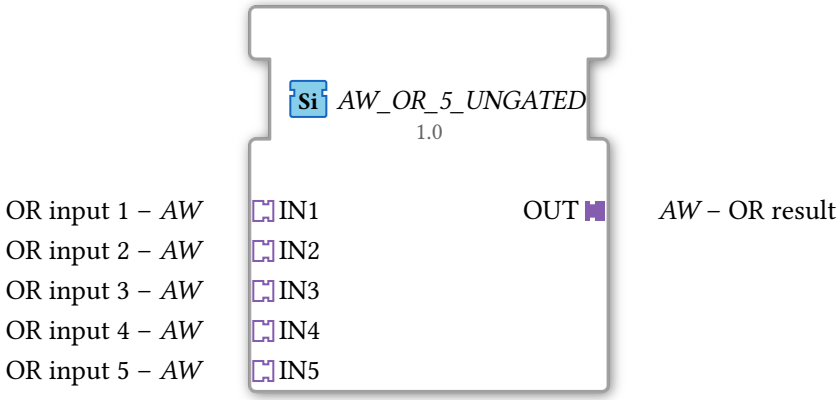
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AW_OR_5** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_OR_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AW_OR_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_OR_5_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 5 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AW)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AW)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 5 Eingangsadapter (`IN1` … `IN5`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 5 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_OR_5_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_5`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_OR_5_UNGATED` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AW_OR_5_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AW_OR_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

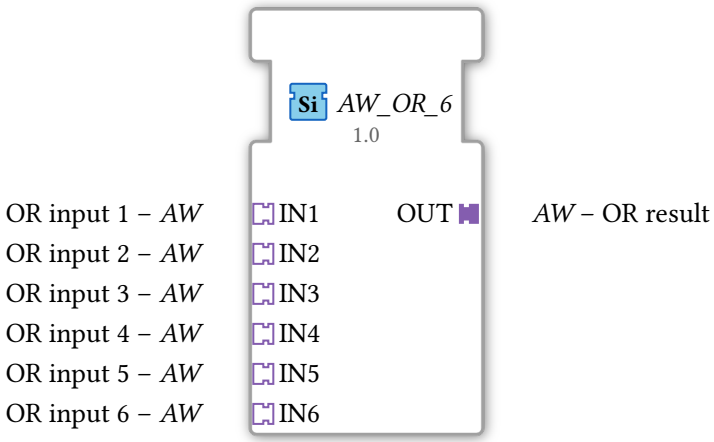
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AW_OR_5_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_OR_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_OR_6** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 6 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AW)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AW)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AW)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 6 Eingangsadapter (`IN1` … `IN6`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 6 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_OR_6 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.

- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_6`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_OR_6` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AW_OR_6` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AW_OR_6_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

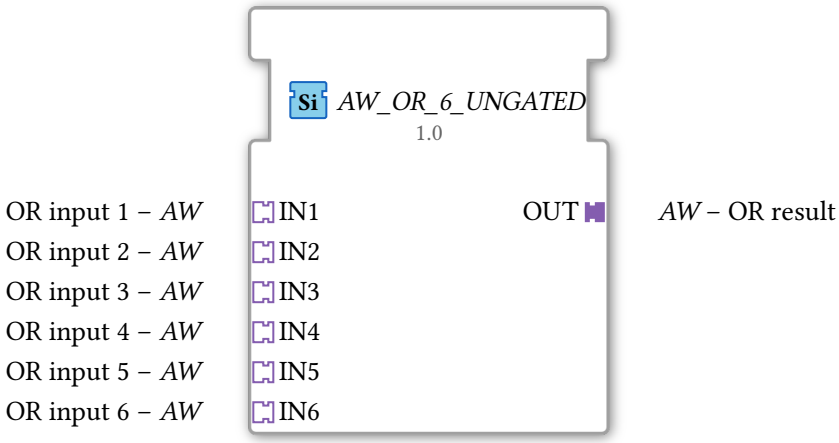
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AW_OR_6** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_OR_6_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AW_OR_6`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_OR_6_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 6 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AW)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AW)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AW)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise
Sobald an einem der 6 Eingangsadapter (`IN1` … `IN6`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 6 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_OR_6_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu AX_OR_6, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet AW_OR_6_UNGATED auf dem vollständigen Bitmuster eines WORD-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet AW_OR_6_UNGATED Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- AW_OR_6: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

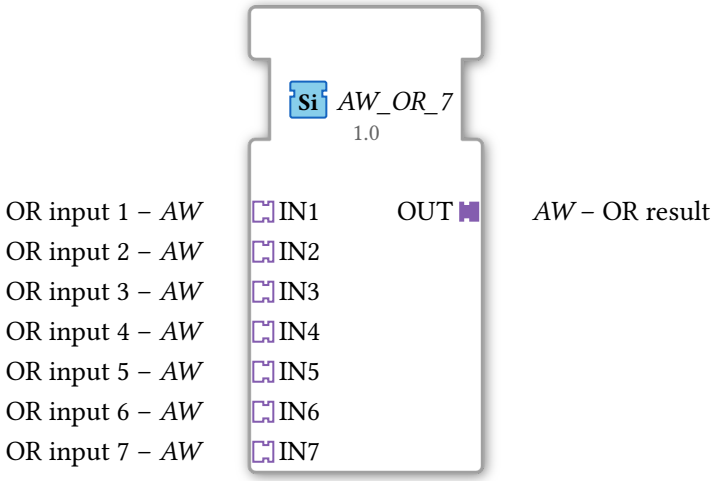
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AW_OR_6_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für WORD-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_OR_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_OR_7** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 7 Eingangswerten vom Typ WORD (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_OR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AW)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AW)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AW)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AW)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 7 Eingangsadapter (`IN1` ... `IN7`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 7 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_OR_7 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_7`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_OR_7` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AW_OR_7` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AW_OR_7_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

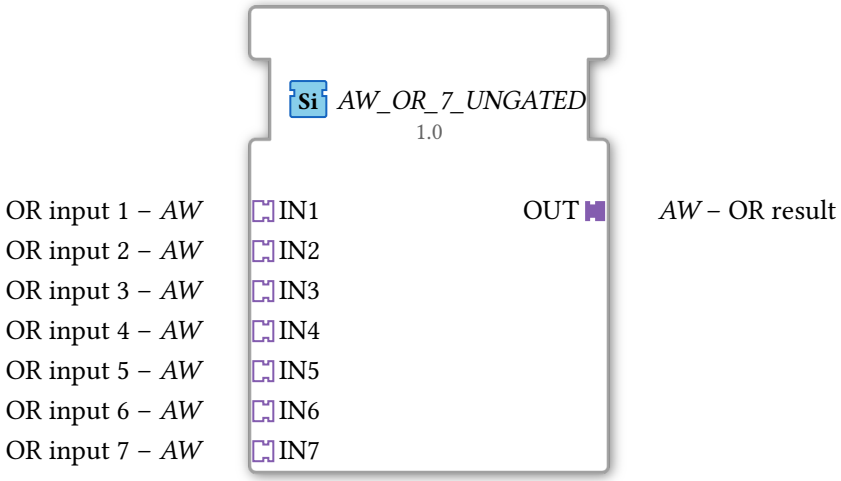
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AW_OR_7** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_OR_7_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AW_OR_7`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_OR_7_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 7 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AW)

- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AW)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AW)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AW)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AW)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 7 Eingangsadapter (`IN1` … `IN7`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 7 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_OR_7_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_7`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_OR_7_UNGATED` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AW_OR_7_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AW_OR_7`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

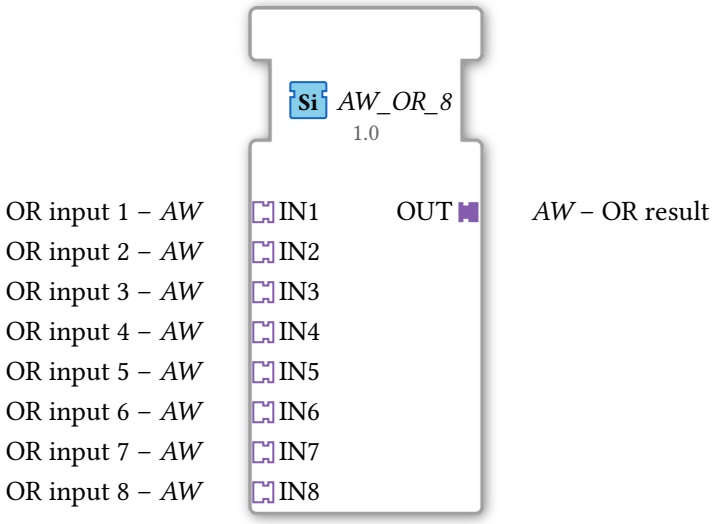
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AW_OR_7_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_OR_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_OR_8** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 8 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AW)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AW)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AW)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AW)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AW)
- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 8 Eingangsadapter (`IN1` … `IN8`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 8 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`.

Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_OR_8 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_8`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_OR_8` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AW_OR_8` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AW_OR_8_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

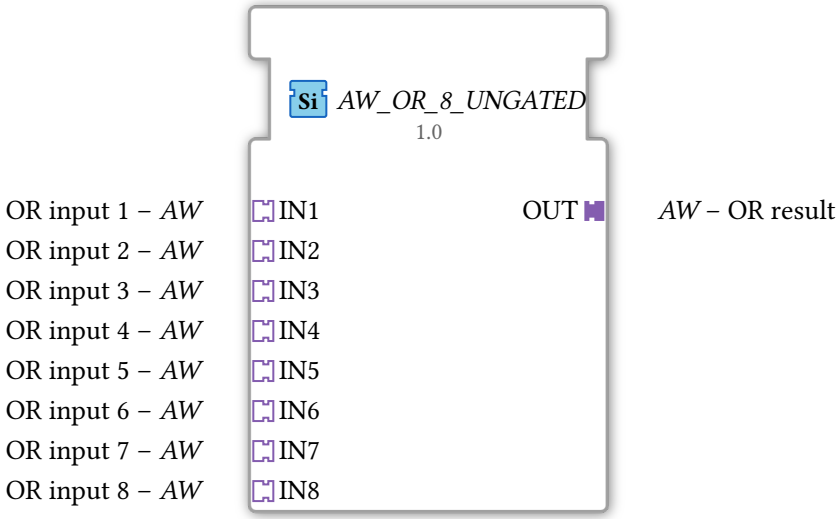
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AW_OR_8** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_OR_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AW_OR_8`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_OR_8_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 8 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AW)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AW)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AW)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AW)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AW)
- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 8 Eingangsadapter (`IN1` … `IN8`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 8 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-

Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_OR_8_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_8`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_OR_8_UNGATED` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AW_OR_8_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AW_OR_8`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

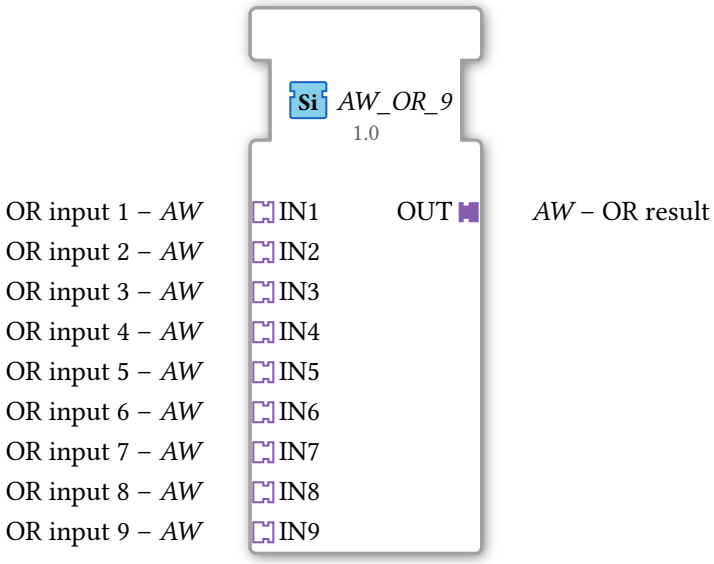
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AW_OR_8_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_OR_9



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_OR_9** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 9 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AW)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AW)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AW)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AW)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AW)
- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AW)
- **IN9**: ODER-Eingang 9 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 9 Eingangsadapter (`IN1` … `IN9`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 9 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_OR_9 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_9`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_OR_9` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AW_OR_9` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AW_OR_9_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

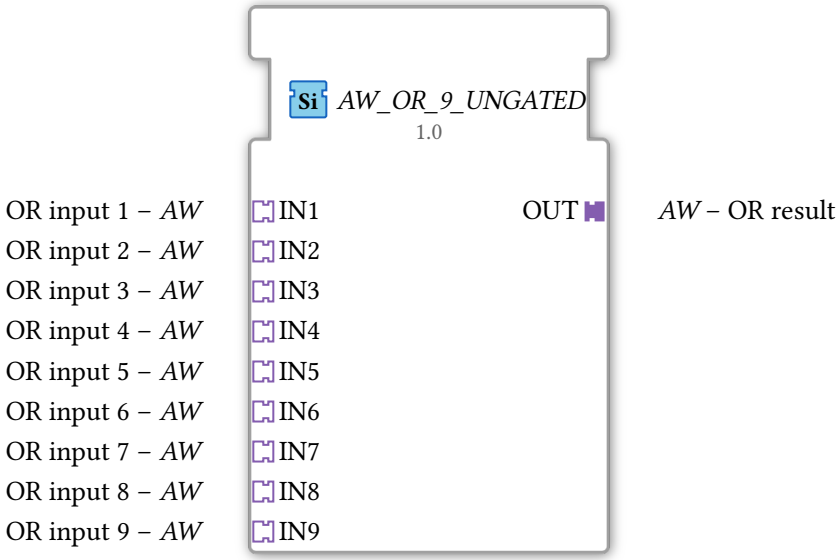
Fazit

Der **AW_OR_9** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_OR_9_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AW_OR_9`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht

gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_OR_9_UNGATED** ist ein generischer Funktionsblock zur bitweisen ODER-Verknüpfung von 9 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_OR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AW)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AW)
- **IN5**: ODER-Eingang 5 (Typ: adapter::types::unidirectional::AW)
- **IN6**: ODER-Eingang 6 (Typ: adapter::types::unidirectional::AW)
- **IN7**: ODER-Eingang 7 (Typ: adapter::types::unidirectional::AW)
- **IN8**: ODER-Eingang 8 (Typ: adapter::types::unidirectional::AW)
- **IN9**: ODER-Eingang 9 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 9 Eingangsadapter (`IN1` … `IN9`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 9 Eingänge bitweise mit **ODER** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der ODER-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_OR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 10 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_OR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_OR_9_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame ODER-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_OR_9`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_OR_9_UNGATED` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein OR_2 verwendet `AW_OR_9_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AW_OR_9`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

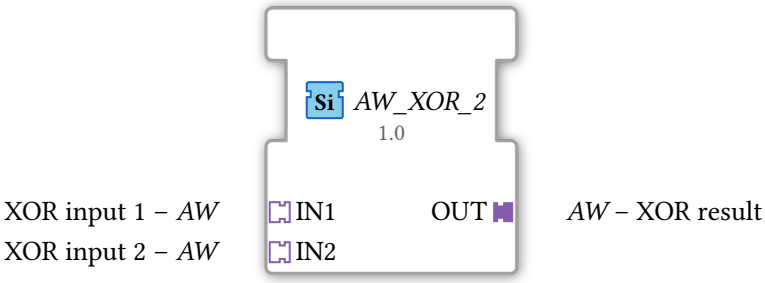
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AW_OR_9_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen ODER-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_XOR_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_XOR_2** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 2 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 2 Eingangsadapter (`IN1` ... `IN2`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_XOR_2 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_2`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_XOR_2` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AW_XOR_2` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AW_XOR_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

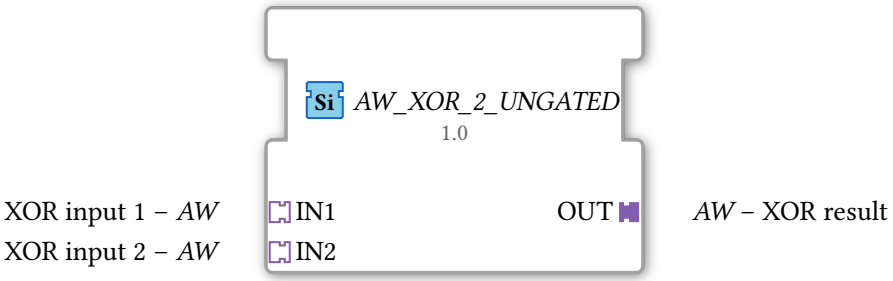
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AW_XOR_2** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_XOR_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AW_XOR_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_XOR_2_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 2 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur boole-

schen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_XOR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise
Sobald an einem der 2 Eingangsadapter (IN1 ... IN2) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 2 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter OUT. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf OUT gehaltenen Wert unterscheidet, wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (GEN_AW_XOR) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen AX_XOR-Bausteinen wird hier jedes Bit des WORD-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ unidirectional::AW – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_XOR_2_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ WORD zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer WORD-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen
Im Gegensatz zu AX_XOR_2, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet AW_XOR_2_UNGATED auf dem vollständigen Bitmuster eines WORD-Werts. Verglichen mit dem Standard-Baustein XOR_2

verwendet AW_XOR_2_UNGATED Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AW_XOR_2**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

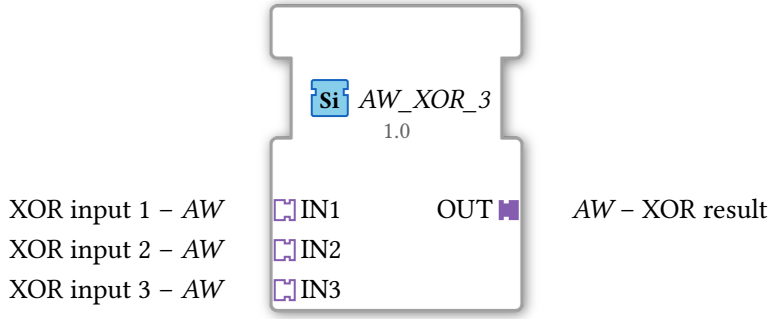
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AW_XOR_2_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für WORD-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_XOR_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_XOR_3** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 3 Eingangswerten vom Typ WORD (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_XOR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` … `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_XOR_3 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_XOR_3` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AW_XOR_3` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AW_XOR_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

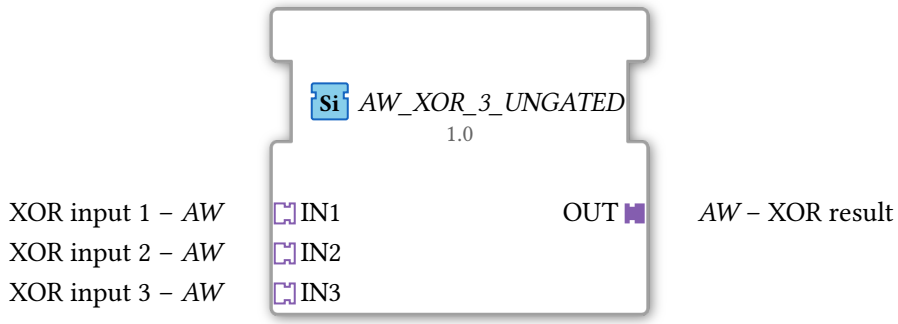
Fazit

Der **AW_XOR_3** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_XOR_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AW_XOR_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht

gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_XOR_3_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 3 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 3 Eingangsadapter (`IN1` … `IN3`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 3 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.

- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_XOR_3_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_3`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_XOR_3_UNGATED` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AW_XOR_3_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AW_XOR_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

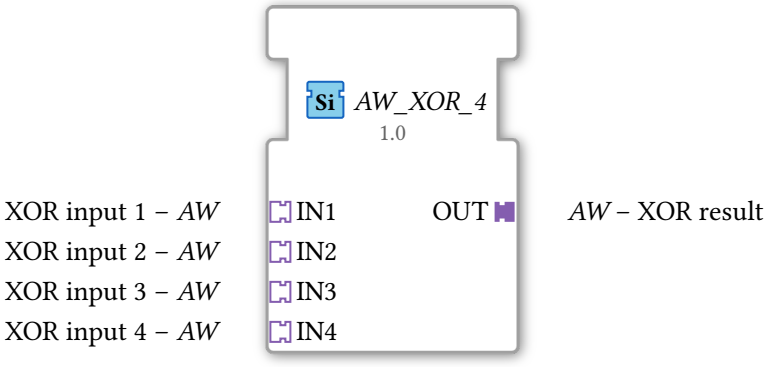
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AW_XOR_3_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_XOR_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_XOR_4** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 4 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AW)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise
Sobald an einem der 4 Eingangsadapter (`IN1` … `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_XOR_4 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ **Vergleich mit ähnlichen Bausteinen**
Im Gegensatz zu `AX_XOR_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_XOR_4` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet

AW_XOR_4 Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **AW_XOR_4_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

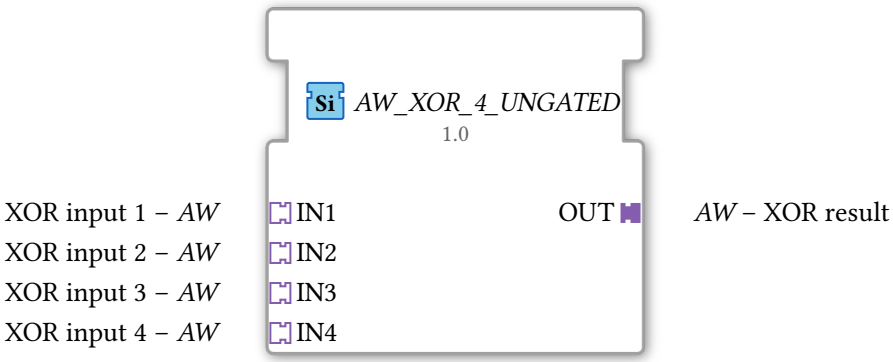
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AW_XOR_4** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für WORD-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_XOR_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AW_XOR_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_XOR_4_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 4 Eingangswerten vom Typ WORD (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den AX_XOR-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AW)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 4 Eingangsadapter (`IN1` ... `IN4`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 4 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_XOR_4_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_4`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_XOR_4_UNGATED` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AW_XOR_4_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AW_XOR_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

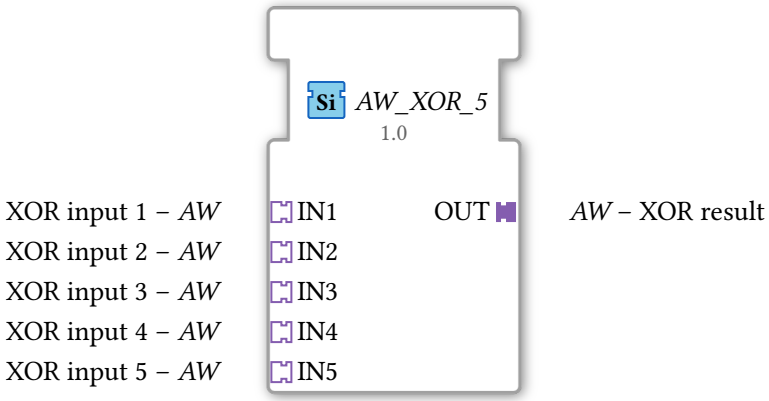
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AW_XOR_4_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_XOR_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_XOR_5** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 5 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AW)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AW)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 5 Eingangsadapter (`IN1` ... `IN5`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 5 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0

(Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_XOR_5 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_5`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_XOR_5` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AW_XOR_5` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AW_XOR_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

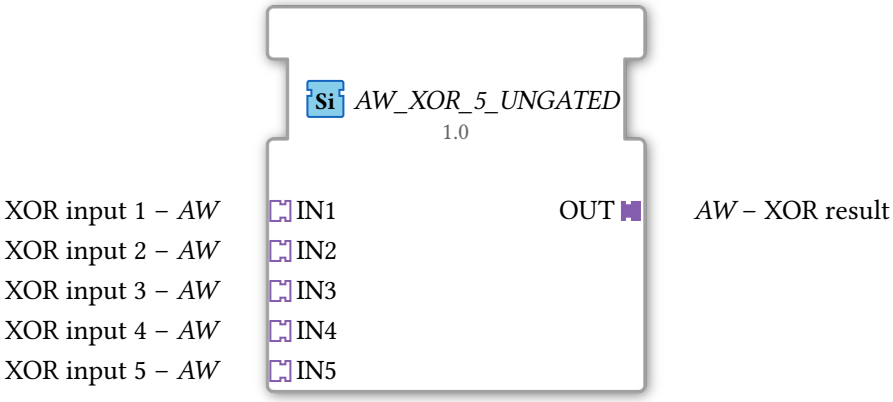
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AW_XOR_5** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_XOR_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AW_XOR_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_XOR_5_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 5 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AW)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AW)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 5 Eingangsadapter (`IN1` … `IN5`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 5 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_XOR_5_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_5`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_XOR_5_UNGATED` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AW_XOR_5_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AW_XOR_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

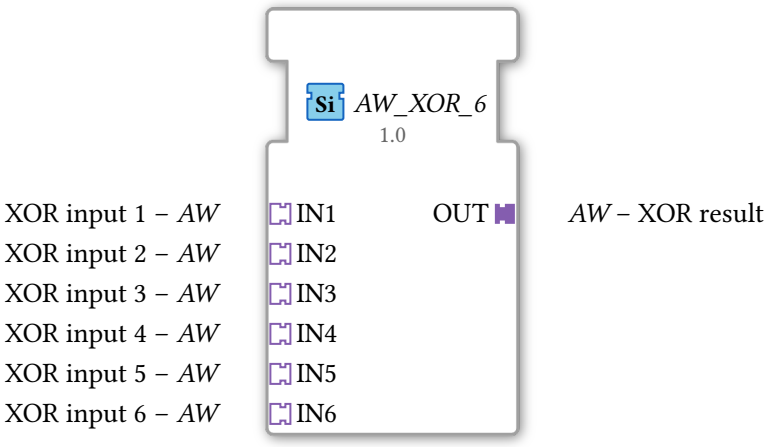
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AW_XOR_5_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_XOR_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_XOR_6** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 6 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AW)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AW)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AW)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 6 Eingangsadapter (`IN1` … `IN6`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 6 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_XOR_6 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.

- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_6`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_XOR_6` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AW_XOR_6` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AW_XOR_6_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

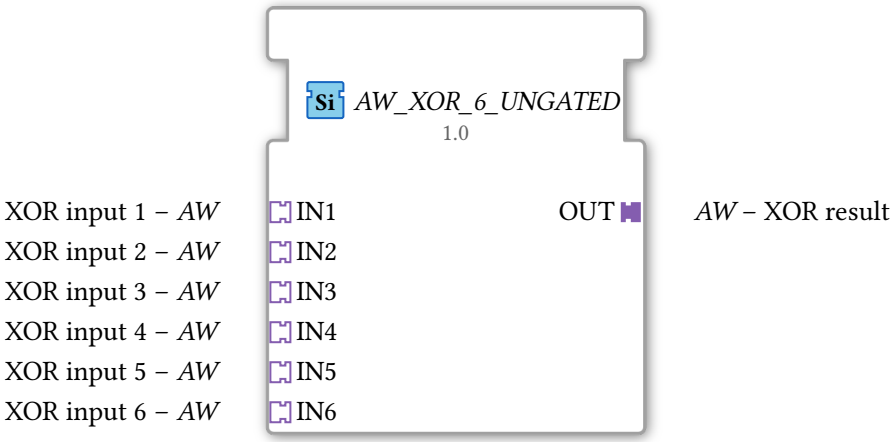
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AW_XOR_6** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_XOR_6_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AW_XOR_6`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_XOR_6_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 6 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AW)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AW)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AW)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise
Sobald an einem der 6 Eingangsadapter (`IN1` … `IN6`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 6 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_XOR_6_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_6`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_XOR_6_UNGATED` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AW_XOR_6_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AW_XOR_6`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

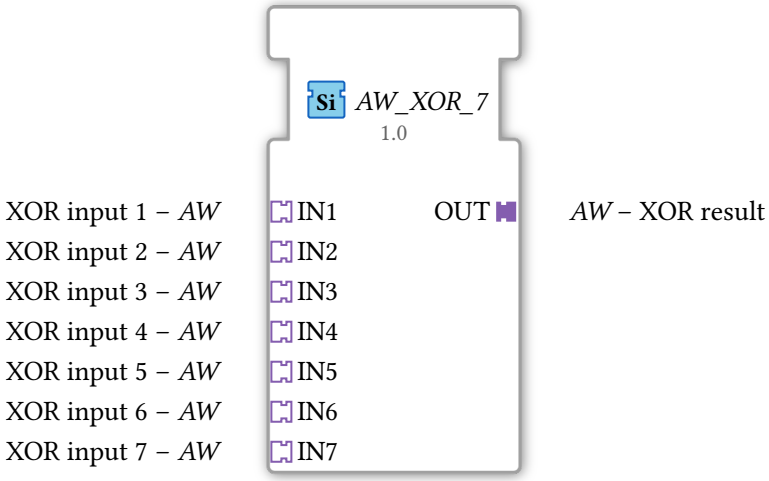
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AW_XOR_6_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_XOR_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_XOR_7** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 7 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AW)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AW)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AW)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AW)
- **IN7**: XOR (exklusives ODER)-Eingang 7 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 7 Eingangsadapter (`IN1` ... `IN7`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 7 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_XOR_7 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_7`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_XOR_7` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AW_XOR_7` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AW_XOR_7_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

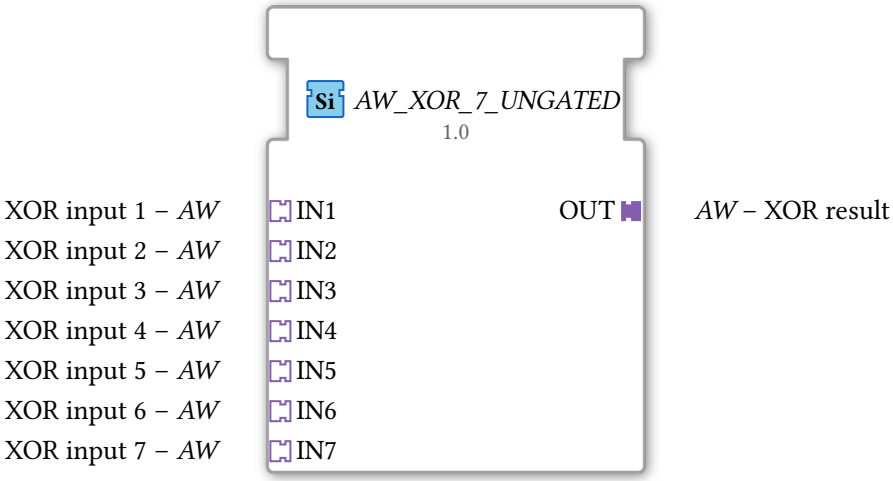
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AW_XOR_7** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_XOR_7_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AW_XOR_7`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_XOR_7_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 7 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AW)

- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AW)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AW)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AW)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AW)
- **IN7**: XOR (exklusives ODER)-Eingang 7 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 7 Eingangsadapter (`IN1` ... `IN7`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 7 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0 (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_XOR_7_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_7`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_XOR_7_UNGATED` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AW_XOR_7_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AW_XOR_7`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

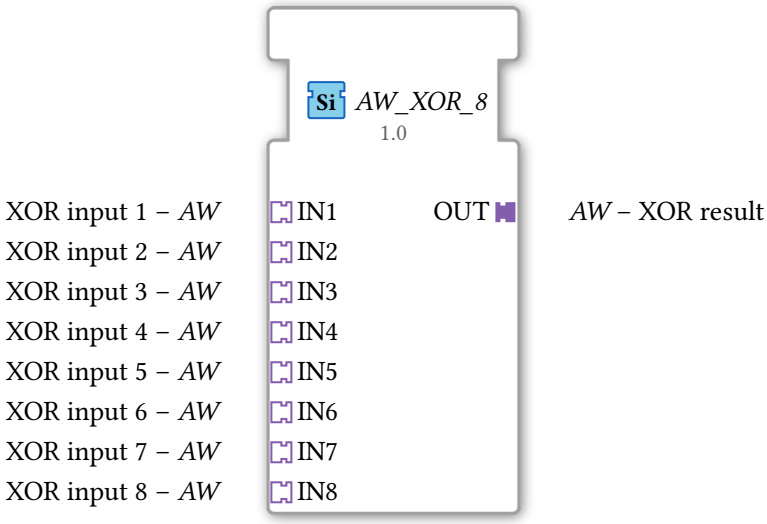
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AW_XOR_7_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_XOR_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_XOR_8** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 8 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AW)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AW)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AW)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AW)
- **IN7**: XOR (exklusives ODER)-Eingang 7 (Typ: adapter::types::unidirectional::AW)
- **IN8**: XOR (exklusives ODER)-Eingang 8 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 8 Eingangsadapter (`IN1` ... `IN8`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 8 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den

Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / `0` (Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_XOR_8 keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_8`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_XOR_8` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AW_XOR_8` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- **`AW_XOR_8_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

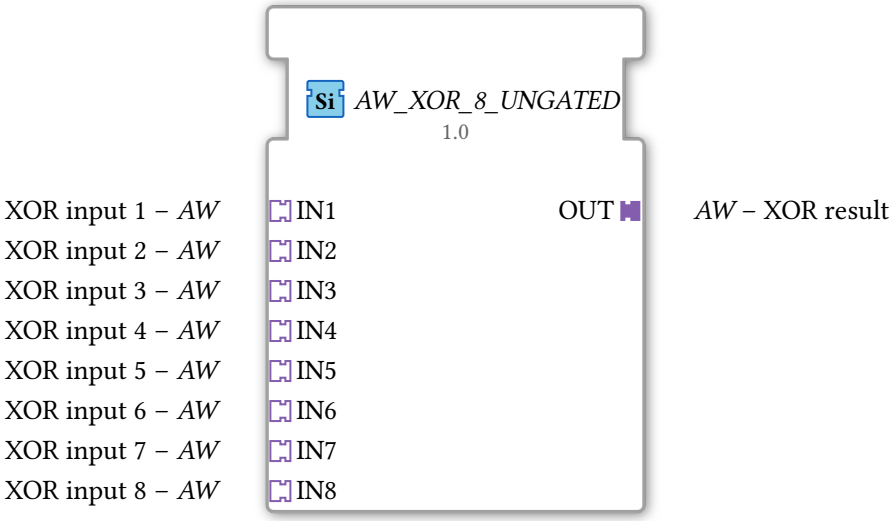
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AW_XOR_8** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

AW_XOR_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AW_XOR_8`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AW_XOR_8_UNGATED** ist ein generischer Funktionsblock zur bitweisen XOR (exklusives ODER)-Verknüpfung von 8 Eingangswerten vom Typ `WORD` (16-Bit-Bitmuster (Wort)). Im Gegensatz zur booleschen Verknüpfung einzelner Wahrheitswerte (wie bei den `AX_XOR`-Bausteinen) wird hier jedes einzelne Bit des Datenworts unabhängig verknüpft.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1**: XOR (exklusives ODER)-Eingang 1 (Typ: adapter::types::unidirectional::AW)
- **IN2**: XOR (exklusives ODER)-Eingang 2 (Typ: adapter::types::unidirectional::AW)
- **IN3**: XOR (exklusives ODER)-Eingang 3 (Typ: adapter::types::unidirectional::AW)
- **IN4**: XOR (exklusives ODER)-Eingang 4 (Typ: adapter::types::unidirectional::AW)
- **IN5**: XOR (exklusives ODER)-Eingang 5 (Typ: adapter::types::unidirectional::AW)
- **IN6**: XOR (exklusives ODER)-Eingang 6 (Typ: adapter::types::unidirectional::AW)
- **IN7**: XOR (exklusives ODER)-Eingang 7 (Typ: adapter::types::unidirectional::AW)
- **IN8**: XOR (exklusives ODER)-Eingang 8 (Typ: adapter::types::unidirectional::AW)

Ausgangsadapter:

- **OUT**: XOR (exklusives ODER)-Ergebnis (Typ: adapter::types::unidirectional::AW)

Funktionsweise

Sobald an einem der 8 Eingangsadapter (`IN1` … `IN8`) ein Ereignis eintrifft, verknüpft der Baustein die Bitmuster aller 8 Eingänge bitweise mit **XOR (exklusives ODER)** und schreibt das Ergebnis auf den Ausgangsadapter `OUT`. Als Startwert der Verknüpfung dient das neutrale Element (alle Bits gelöscht / 0

(Nullelement der XOR-Verknüpfung)), sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird.

Nur wenn sich das neu berechnete Ergebnis vom aktuell auf `OUT` gehaltenen Wert unterscheidet, wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AW_XOR`) definiert und deckt über den GenericClassName-Mechanismus alle Aritäten (2 bis 8 Eingänge) derselben Grundlogik ab.
- **Bitweise Verknüpfung**: Anders als bei den booleschen `AX_XOR`-Bausteinen wird hier jedes Bit des `WORD`-Datenworts einzeln verknüpft, nicht nur ein einzelner Wahrheitswert.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `unidirectional::AW` – die Daten fließen nur vom Socket zum Plug.
- **Normkonformität**: Der Baustein implementiert die Verknüpfung gemäß IEC 61499-2 / IEC 61131-3.

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AW_XOR_8_UNGATED keine internen Zustände. Die Ausgabe wird bei jedem eingehenden Ereignis direkt aus den aktuellen Eingangswerten neu berechnet.

Anwendungsszenarien

- **Bitmasken-Verknüpfung**: Kombinieren mehrerer Statusregister oder Flag-Bytes vom Typ `WORD` zu einem Gesamtergebnis.
- **Signalaggregation**: Zusammenführen mehrerer `WORD`-Datenquellen (z. B. aus verschiedenen Modulen) über eine gemeinsame XOR (exklusives ODER)-Verknüpfung.
- **Diagnose- und Statusauswertung**: Prüfen von Bitmustern auf gemeinsame oder unterschiedliche gesetzte Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AX_XOR_8`, der einzelne boolesche Wahrheitswerte verknüpft, arbeitet `AW_XOR_8_UNGATED` auf dem vollständigen Bitmuster eines `WORD`-Werts. Verglichen mit dem Standard-Baustein XOR_2 verwendet `AW_XOR_8_UNGATED` Adapter-basierte Schnittstellen anstelle direkter Daten-/Ereignisein-/ausgänge, was eine flexiblere Integration in Adapter-basierte Systemarchitekturen ermöglicht.

- `AW_XOR_8`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

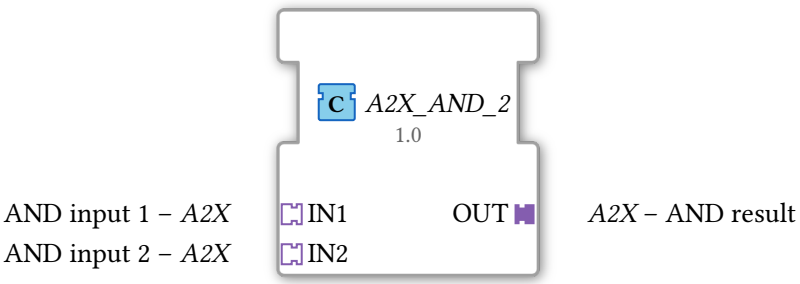
Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

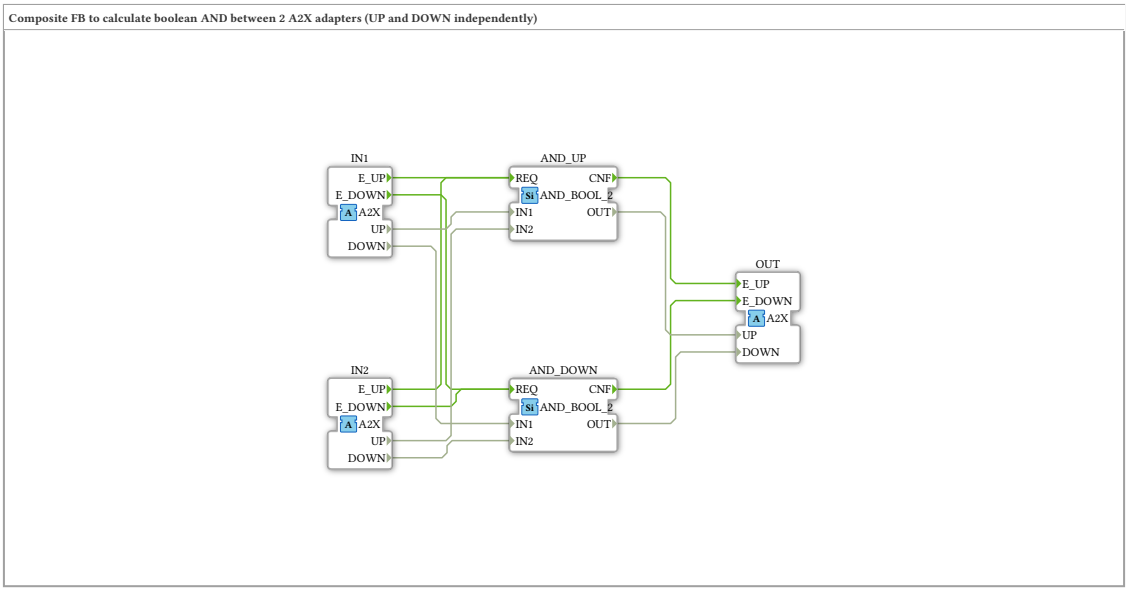
Fazit

Der **AW_XOR_8_UNGATED** bietet eine zuverlässige, generische Implementierung der bitweisen XOR (exklusives ODER)-Funktion für `WORD`-Werte mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden und mehrere Bitmuster desselben Datentyps kombinieren müssen.

2.1.11.7 booleanOperators

A2X_AND_2





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock A2X_AND_2 berechnet die logische UND-Verknüpfung zweier A2X-Adapter – getrennt für den UP- und den DOWN-Kanal. Im Gegensatz zu den einkanaligen, generischen `AX_AND`-Bausteinen ist A2X_AND_2 ein handgebauter Composite-Funktionsblock, der zwei Standard-Bausteine `AND_BOOL_2` intern verdrahtet, einen je Kanal.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Ereignisse laufen über die Adapter `IN1`/`IN2`.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge – die Ereignisse laufen über den Adapter `OUT`.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

Eingangsadapter:

- **IN1**: UND-Eingang 1 (Typ: `adapter::types::unidirectional::A2X`)
- **IN2**: UND-Eingang 2 (Typ: `adapter::types::unidirectional::A2X`)

Ausgangsadapter:

- **OUT**: UND-Ergebnis (Typ: `adapter::types::unidirectional::A2X`)

Funktionsweise

Der Baustein enthält zwei interne Instanzen von `iec61131::booleanOperators::AND_BOOL_2`: `AND_UP` verknüpft die UP-Kanäle von `IN1` und `IN2`, `AND_DOWN` die DOWN-Kanäle. Trifft an `IN1` oder `IN2` ein UP-Ereignis (`E_UP`) ein, wird `AND_UP.REQ` ausgelöst, das Ergebnis von `IN1.UP AND IN2.UP` berechnet und über `OUT.E_UP`/`OUT.UP` ausgegeben. Für den DOWN-Kanal läuft dieselbe Logik unabhängig über `AND_DOWN`. Da ein Ereignisziel – anders als ein Datenziel – mehrere Quellen haben darf, dürfen sowohl `IN1.E_UP` als

auch IN2.E_UP in AND_UP.REQ münden; das Ergebnis wird bei jedem der beiden auslösenden Ereignisse neu berechnet.

Technische Besonderheiten

- Echter Composite-Baustein (kein generischer GenericClassName-Mechanismus wie bei AX_AND_2), aufgebaut aus zwei Standard-AND_BOOL_2-Instanzen
- Zwei unabhängige Kanäle (UP/DOWN), jeweils mit eigenem Ereignispaar – kein gemeinsamer Zustand zwischen den Kanälen
- Da IEC 61499 mehrere Quellen auf ein Ereignisziel erlaubt, aber nicht auf ein Datenziel, wird pro Kanal genau ein Logikbaustein zwischengeschaltet, statt die beiden Eingangsdaten direkt auf eine gemeinsame Variable zu verdrahten

Zustandsübersicht

Der Baustein ist ein kombinatorischer Logikbaustein ohne internen Zustand; jedes eintreffende Ereignis berechnet das Ergebnis direkt neu:

- IN1.E_UP, IN2.E_UP → AND_UP.REQ; IN1.UP, IN2.UP → AND_UP.IN1/IN2; AND_UP.CNF → OUT.E_UP; AND_UP.OUT → OUT.UP
- IN1.E_DOWN, IN2.E_DOWN → AND_DOWN.REQ; IN1.DOWN, IN2.DOWN → AND_DOWN.IN1/IN2; AND_DOWN.CNF → OUT.E_DOWN; AND_DOWN.OUT → OUT.DOWN

Anwendungsszenarien

- Sicherheitslogik, bei der zwei UP/DOWN-Signalgeber gleichzeitig aktiv sein müssen, bevor ein Aktor freigegeben wird
- Verriegelungen zwischen zwei redundanten A2X-Signalquellen
- Kombinatorische Logik in Steuerungen, die auf A2X-Adaptern statt auf einfachen BOOL-Signalen basieren

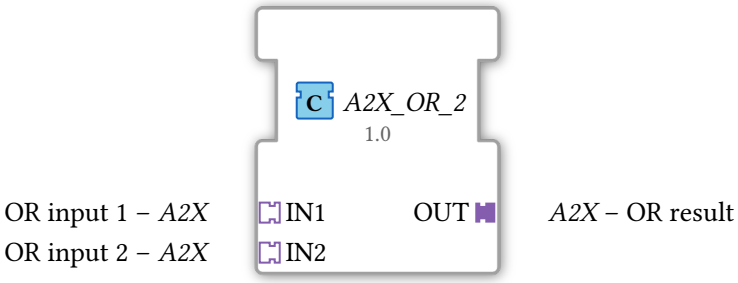
⚖️ Vergleich mit ähnlichen Bausteinen

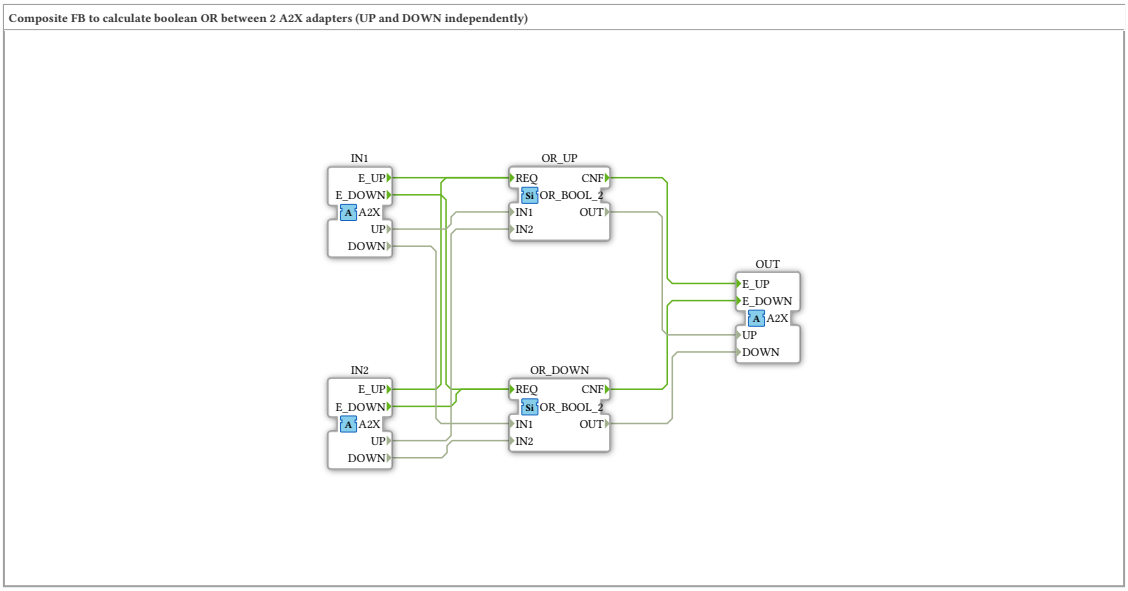
A2X_OR_2 und A2X_XOR_2 sind baugleich, verwenden aber OR_BOOL_2 bzw. XOR_BOOL_2 statt AND_BOOL_2. Der einkanalige Vorgänger AX_AND_2 verknüpft nur ein einzelnes Bool-Signal und ist – anders als A2X_AND_2 – als generischer Baustein (GEN_AX_AND) implementiert. Im Unterschied zu AB_AND_2, das ein komplettes BYTE-Bitmuster verknüpft, verarbeitet A2X_AND_2 einzelne boolesche Wahrheitswerte je Kanal.

Fazit

A2X_AND_2 bringt die logische UND-Verknüpfung auf die zweikanalige A2X-Welt: zwei unabhängige, ereignisgesteuerte UND-Verknüpfungen für UP und DOWN, sauber getrennt und ohne Mehrfachschreiber auf einer gemeinsamen Variable.

A2X_OR_2





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock A2X_OR_2 berechnet die logische ODER-Verknüpfung zweier A2X-Adapter – getrennt für den UP- und den DOWN-Kanal. Wie A2X_AND_2 ist er ein handgebauter Composite-Funktionsblock, der zwei Standard-Bausteine `OR_BOOL_2` intern verdrahtet, einen je Kanal.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Ereignisse laufen über die Adapter `IN1`/`IN2`.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge – die Ereignisse laufen über den Adapter `OUT`.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

Eingangsadapter:

- **IN1**: ODER-Eingang 1 (Typ: `adapter::types::unidirectional::A2X`)
- **IN2**: ODER-Eingang 2 (Typ: `adapter::types::unidirectional::A2X`)

Ausgangsadapter:

- **OUT**: ODER-Ergebnis (Typ: `adapter::types::unidirectional::A2X`)

Funktionsweise

Der Baustein enthält zwei interne Instanzen von `iec61131::booleanOperators::OR_BOOL_2`: `OR_UP` verknüpft die UP-Kanäle von `IN1` und `IN2`, `OR_DOWN` die DOWN-Kanäle. Trifft an `IN1` oder `IN2` ein UP-Ereignis (`E_UP`) ein, wird `OR_UP.REQ` ausgelöst, das Ergebnis von `IN1.UP OR IN2.UP` berechnet und über `OUT.E_UP`/`OUT.UP` ausgegeben. Für den DOWN-Kanal läuft dieselbe Logik unabhängig über `OR_DOWN`.

Technische Besonderheiten

- Echter Composite-Baustein (kein generischer `GenericClassName`-Mechanismus), aufgebaut aus zwei Standard-`OR_BOOL_2`-Instanzen
- Zwei unabhängige Kanäle (UP/DOWN), jeweils mit eigenem Ereignispaar – kein gemeinsamer Zustand zwischen den Kanälen
- Da IEC 61499 mehrere Quellen auf ein Ereignisziel erlaubt, aber nicht auf ein Datenziel, wird pro Kanal genau ein Logikbaustein zwischengeschaltet, statt die beiden Eingangsdaten direkt auf eine gemeinsame Variable zu verdrahten

Zustandsübersicht

Der Baustein ist ein kombinatorischer Logikbaustein ohne internen Zustand; jedes eintreffende Ereignis berechnet das Ergebnis direkt neu:

- IN1.E_UP, IN2.E_UP → OR_UP.REQ; IN1.UP, IN2.UP → OR_UP.IN1/IN2; OR_UP.CNF → OUT.E_UP; OR_UP.OUT → OUT.UP
- IN1.E_DOWN, IN2.E_DOWN → OR_DOWN.REQ; IN1.DOWN, IN2.DOWN → OR_DOWN.IN1/IN2; OR_DOWN.CNF → OUT.E_DOWN; OR_DOWN.OUT → OUT.DOWN

Anwendungsszenarien

- Zusammenführen zweier redundanter oder alternativer UP/DOWN-Signalquellen zu einem gemeinsamen Ergebnis
- Sammelmeldungen, bei denen das Auslösen eines von mehreren A2X-Signalgebern genügt
- Kombinatorische Logik in Steuerungen, die auf A2X-Adaptern statt auf einfachen BOOL-Signalen basieren

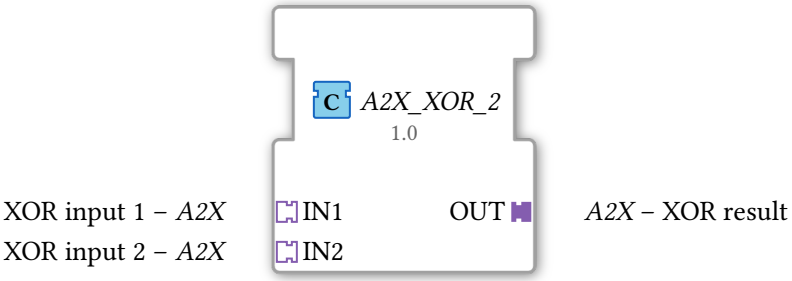
⚖️ Vergleich mit ähnlichen Bausteinen

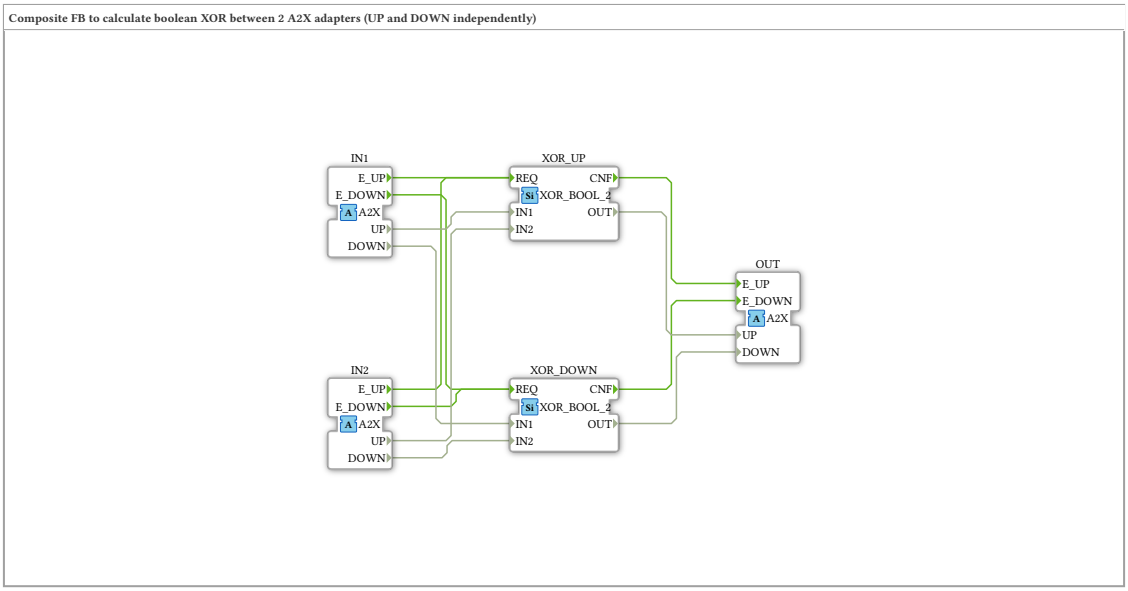
A2X_AND_2 und A2X_XOR_2 sind baugleich, verwenden aber `AND_BOOL_2` bzw. `XOR_BOOL_2` statt `OR_BOOL_2`. Der einkanalige Vorgänger AX_OR_2 verknüpft nur ein einzelnes Bool-Signal und ist – anders als A2X_OR_2 – als generischer Baustein implementiert. Im Unterschied zu bitweisen Bausteinen wie AB_AND_2 verarbeitet A2X_OR_2 einzelne boolesche Wahrheitswerte je Kanal.

Fazit

A2X_OR_2 bringt die logische ODER-Verknüpfung auf die zweikanalige A2X-Welt: zwei unabhängige, ereignisgesteuerte ODER-Verknüpfungen für UP und DOWN, sauber getrennt und ohne Mehrfachschreiber auf einer gemeinsamen Variable.

A2X_XOR_2





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock A2X_XOR_2 berechnet die logische EXKLUSIV-ODER-Verknüpfung zweier A2X-Adapter – getrennt für den UP- und den DOWN-Kanal. Wie A2X_AND_2 und A2X_OR_2 ist er ein handgebauter Composite-Funktionsblock, der zwei Standard-Bausteine `XOR_BOOL_2` intern verdrahtet, einen je Kanal.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Ereignisse laufen über die Adapter `IN1`/`IN2`.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge – die Ereignisse laufen über den Adapter `OUT`.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

Eingangsadapter:

- **IN1**: XOR-Eingang 1 (Typ: `adapter::types::unidirectional::A2X`)
- **IN2**: XOR-Eingang 2 (Typ: `adapter::types::unidirectional::A2X`)

Ausgangsadapter:

- **OUT**: XOR-Ergebnis (Typ: `adapter::types::unidirectional::A2X`)

Funktionsweise

Der Baustein enthält zwei interne Instanzen von `iec61131::booleanOperators::XOR_BOOL_2`: `XOR_UP` verknüpft die UP-Kanäle von `IN1` und `IN2`, `XOR_DOWN` die DOWN-Kanäle. Trifft an `IN1` oder `IN2` ein UP-Ereignis (`E_UP`) ein, wird `XOR_UP.REQ` ausgelöst, das Ergebnis von `IN1.UP XOR IN2.UP` berechnet und über `OUT.E_UP`/`OUT.UP` ausgegeben. Für den DOWN-Kanal läuft dieselbe Logik unabhängig über `XOR_DOWN`.

Technische Besonderheiten

- Echter Composite-Baustein (kein generischer `GenericClassName`-Mechanismus), aufgebaut aus zwei Standard-`XOR_BOOL_2`-Instanzen
- Zwei unabhängige Kanäle (UP/DOWN), jeweils mit eigenem Ereignispaar – kein gemeinsamer Zustand zwischen den Kanälen
- Da IEC 61499 mehrere Quellen auf ein Ereignisziel erlaubt, aber nicht auf ein Datenziel, wird pro Kanal genau ein Logikbaustein zwischengeschaltet, statt die beiden Eingangsdaten direkt auf eine gemeinsame Variable zu verdrahten

Zustandsübersicht

Der Baustein ist ein kombinatorischer Logikbaustein ohne internen Zustand; jedes eintreffende Ereignis berechnet das Ergebnis direkt neu:

- IN1.E_UP, IN2.E_UP → XOR_UP.REQ; IN1.UP, IN2.UP → XOR_UP.IN1/IN2; XOR_UP.CNF → OUT.E_UP; XOR_UP.OUT → OUT.UP
- IN1.E_DOWN, IN2.E_DOWN → XOR_DOWN.REQ; IN1.DOWN, IN2.DOWN → XOR_DOWN.IN1/IN2; XOR_DOWN.CNF → OUT.E_DOWN; XOR_DOWN.OUT → OUT.DOWN

Anwendungsszenarien

- Erkennen von Widersprüchen zwischen zwei UP/DOWN-Signalquellen, die eigentlich synchron laufen sollten
- Umschalt- oder Wechsellogik, bei der genau ein Signal aktiv sein darf
- Kombinatorische Logik in Steuerungen, die auf A2X-Adaptern statt auf einfachen BOOL-Signalen basieren

⚖️ Vergleich mit ähnlichen Bausteinen

A2X_AND_2 und A2X_OR_2 sind baugleich, verwenden aber `AND_BOOL_2` bzw. `OR_BOOL_2` statt `XOR_BOOL_2`. Der einkanalige Vorgänger AX_XOR_2 verknüpft nur ein einzelnes Bool-Signal. Im Unterschied zu bitweisen Bausteinen wie AB_AND_2 verarbeitet A2X_XOR_2 einzelne boolesche Wahrheitswerte je Kanal.

Fazit

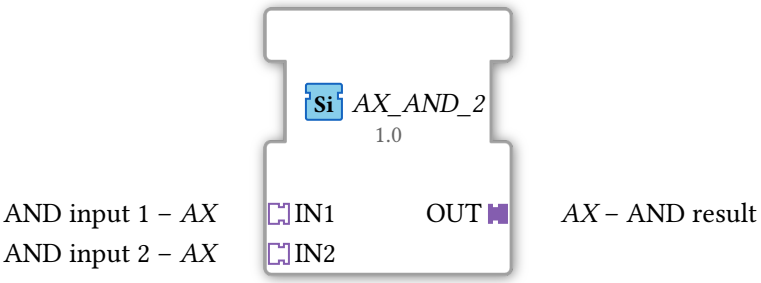
A2X_XOR_2 bringt die logische EXKLUSIV-ODER-Verknüpfung auf die zweikanalige A2X-Welt: zwei unabhängige, ereignisgesteuerte XOR-Verknüpfungen für UP und DOWN, sauber getrennt und ohne Mehrfachschreiber auf einer gemeinsamen Variable.

AX_AND_2

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_AND_2 ist ein generischer Funktionsblock zur Berechnung der logischen UND-Verknüpfung. Der Baustein verarbeitet zwei boolesche Eingangssignale und gibt das logische UND-Ergebnis aus.



Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1** - UND-Eingang 1 (Adapter-Typ: adapter::types::unidirectional::AX)
- **IN2** - UND-Eingang 2 (Adapter-Typ: adapter::types::unidirectional::AX)

Ausgangsadapter:

- **OUT** - UND-Ergebnis (Adapter-Typ: adapter::types::unidirectional::AX)

Funktionsweise

Der Funktionsblock führt eine logische UND-Verknüpfung der beiden Eingangssignale IN1 und IN2 durch. Das Ergebnis wird über den Ausgangsadapter OUT ausgegeben. Die Verarbeitung erfolgt unidirektional über die Adapterschnittstellen.

Technische Besonderheiten

- Generischer Funktionsblock mit spezifischem Klassennamen ‚GEN_AX_AND'
- Verwendet unidirektionale Adapter für die Signalübertragung
- Implementiert gemäß IEC 61499-2 Standard

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AX_AND_2 keine internen Zustände. Die Ausgabe wird direkt aus den aktuellen Eingangswerten berechnet.

Anwendungsszenarien

- Sicherheitskritische Steuerungen, bei denen zwei Bedingungen gleichzeitig erfüllt sein müssen
- Verknüpfung von Sensorsignalen in industriellen Automatisierungsanlagen
- Logische Verknüpfungen in Steuerungssystemen
- Sicherheitsabschaltungen mit Mehrfachbedingungen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu standardmäßigen UND-Bausteinen verwendet AX_AND_2 Adapter-basierte Schnittstellen anstelle von direkten Daten- und Ereignisein-/ausgängen. Dies ermöglicht eine flexiblere Integration in Adapter-basierte Systemarchitekturen.

Vergleich AND_2

- `AX_AND_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

🛠️ Zugehörige Übungen

- Uebung_002a_AX
- Uebung_002b3_AX
- Uebung_006a3_AX

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der AX_AND_2 bietet eine zuverlässige und standardkonforme Implementierung der logischen UND-Funktion mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in verschiedenen Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden.

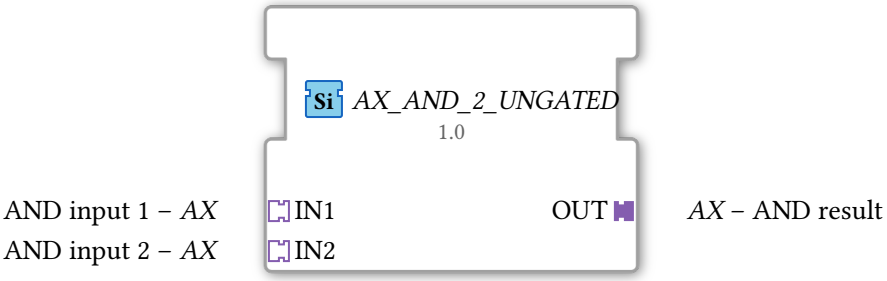
AX_AND_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AX_AND_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_AND_2_UNGATED ist ein generischer Funktionsblock zur Berechnung der logischen UND-Verknüpfung. Der Baustein verarbeitet zwei boolesche Eingangssignale und gibt das logische UND-Ergebnis aus.



Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1** - UND-Eingang 1 (Adapter-Typ: adapter::types::unidirectional::AX)
- **IN2** - UND-Eingang 2 (Adapter-Typ: adapter::types::unidirectional::AX)

Ausgangsadapter:

- **OUT** - UND-Ergebnis (Adapter-Typ: adapter::types::unidirectional::AX)

Funktionsweise

Der Funktionsblock führt eine logische UND-Verknüpfung der beiden Eingangssignale IN1 und IN2 durch. Das Ergebnis wird über den Ausgangsadapter OUT ausgegeben. Die Verarbeitung erfolgt unidirektional über die Adapterschnittstellen.

Technische Besonderheiten

- Generischer Funktionsblock mit spezifischem Klassennamen ‚GEN_AX_AND'
- Verwendet unidirektionale Adapter für die Signalübertragung
- Implementiert gemäß IEC 61499-2 Standard

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AX_AND_2_UNGATED keine internen Zustände. Die Ausgabe wird direkt aus den aktuellen Eingangswerten berechnet.

Anwendungsszenarien

- Sicherheitskritische Steuerungen, bei denen zwei Bedingungen gleichzeitig erfüllt sein müssen
- Verknüpfung von Sensorsignalen in industriellen Automatisierungsanlagen
- Logische Verknüpfungen in Steuerungssystemen
- Sicherheitsabschaltungen mit Mehrfachbedingungen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu standardmäßigen UND-Bausteinen verwendet AX_AND_2_UNGATED Adapter-basierte Schnittstellen anstelle von direkten Daten- und Ereignisein-/ausgängen. Dies ermöglicht eine flexiblere Integration in Adapter-basierte Systemarchitekturen.

Vergleich AND_2

- **`AX_AND_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

🛠️ Zugehörige Übungen

- Uebung_002a_AX
- Uebung_002b3_AX
- Uebung_006a3_AX

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

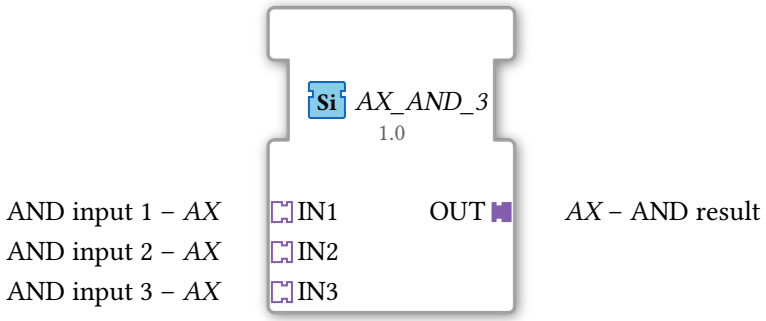
Der AX_AND_2_UNGATED bietet eine zuverlässige und standardkonforme Implementierung der logischen UND-Funktion mit Adapter-basierten Schnittstellen. Seine generische Natur macht ihn vielseitig einsetzbar in verschiedenen Automatisierungsprojekten, die nach IEC 61499-Standard entwickelt werden.

AX_AND_3

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_AND_3 ist ein generischer Funktionsblock zur Berechnung einer logischen UND-Verknüpfung mit drei Eingängen. Der Baustein führt eine boolesche AND-Operation auf drei unabhängigen Eingangssignalen durch und gibt das Ergebnis über einen Adapter-Ausgang aus.



Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter (Sockets):

- **IN1** - AND-Eingang 1 (Typ: adapter::types::unidirectional::AX)
- **IN2** - AND-Eingang 2 (Typ: adapter::types::unidirectional::AX)
- **IN3** - AND-Eingang 3 (Typ: adapter::types::unidirectional::AX)

Ausgangsadapter (Plugs):

- **OUT** - AND-Ergebnis (Typ: adapter::types::unidirectional::AX)

Funktionsweise
Der AX_AND_3-Block führt eine logische UND-Verknüpfung der drei Eingangssignale durch. Das Ausgangssignal ist nur dann aktiv (TRUE), wenn alle drei Eingangssignale gleichzeitig aktiv (TRUE) sind. Die Verarbeitung erfolgt kontinuierlich basierend auf den aktuellen Eingangswerten.

Technische Besonderheiten
- Verwendet unidirektionale Adapter für die Signalübertragung
- Implementiert als generischer Funktionsblock mit der generischen Klasse ‚GEN_AX_AND'
- Arbeitet mit AX-Typ Adaptern für standardisierte Signalübertragung
- Keine Ereignissteuerung - kontinuierliche Auswertung

Zustandsübersicht
Der Funktionsblock besitzt keinen internen Zustandsautomaten, da er als kombinatorische Logikschaltung arbeitet. Die Ausgabe wird direkt aus der aktuellen Kombination der Eingangswerte berechnet.

Anwendungsszenarien
- Sicherheitssteuerungen, bei denen mehrere Bedingungen gleichzeitig erfüllt sein müssen
- Verknüpfung von Sensor-Signalen in industriellen Steuerungen
- Logische Verarbeitung in Automatisierungssystemen
- Überwachungssysteme mit Mehrfachbedingungen

⚖️ Vergleich mit ähnlichen Bausteinen
Im Vergleich zu Standard-AND-Blöcken bietet AX_AND_3:

- Drei Eingänge statt der typischen zwei Eingänge
- Adapter-basierte Schnittstelle statt direkter Daten-Ein-/Ausgänge
- Spezifische AX-Typ-Kompatibilität
- Unidirektionale Signalübertragung

Vergleich mit AND_3

- `AX_AND_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

🛠️ Zugehörige Übungen
- Uebung_002a6_AX

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der AX_AND_3 ist ein spezialisierter Logikbaustein für Anwendungen, die eine dreifache UND-Verknüpfung mit standardisierten AX-Adaptern erfordern. Seine Adapter-basierte Architektur ermöglicht eine einfache Integration in bestehende Steuerungssysteme und bietet eine zuverlässige Lösung für komplexe logische Verknüpfungen.

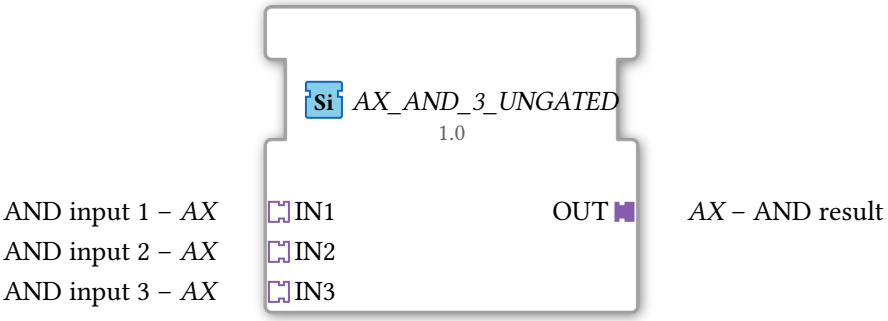
AX_AND_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AX_AND_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_AND_3_UNGATED ist ein generischer Funktionsblock zur Berechnung einer logischen UND-Verknüpfung mit drei Eingängen. Der Baustein führt eine boolesche AND-Operation auf drei unabhängigen Eingangssignalen durch und gibt das Ergebnis über einen Adapter-Ausgang aus.



Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter (Sockets):

- **IN1** - AND-Eingang 1 (Typ: adapter::types::unidirectional::AX)
- **IN2** - AND-Eingang 2 (Typ: adapter::types::unidirectional::AX)

- **IN3** - AND-Eingang 3 (Typ: adapter::types::unidirectional::AX)

Ausgangsadapter (Plugs):

- **OUT** - AND-Ergebnis (Typ: adapter::types::unidirectional::AX)

Funktionsweise

Der AX_AND_3_UNGATED-Block führt eine logische UND-Verknüpfung der drei Eingangssignale durch. Das Ausgangssignal ist nur dann aktiv (TRUE), wenn alle drei Eingangssignale gleichzeitig aktiv (TRUE) sind. Die Verarbeitung erfolgt kontinuierlich basierend auf den aktuellen Eingangswerten.

Technische Besonderheiten

- Verwendet unidirektionale Adapter für die Signalübertragung
- Implementiert als generischer Funktionsblock mit der generischen Klasse ‚GEN_AX_AND'
- Arbeitet mit AX-Typ Adaptern für standardisierte Signalübertragung
- Keine Ereignissteuerung - kontinuierliche Auswertung

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustandsautomaten, da er als kombinatorische Logikschaltung arbeitet. Die Ausgabe wird direkt aus der aktuellen Kombination der Eingangswerte berechnet.

Anwendungsszenarien

- Sicherheitssteuerungen, bei denen mehrere Bedingungen gleichzeitig erfüllt sein müssen
- Verknüpfung von Sensor-Signalen in industriellen Steuerungen
- Logische Verarbeitung in Automatisierungssystemen
- Überwachungssysteme mit Mehrfachbedingungen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-AND-Blöcken bietet AX_AND_3_UNGATED:

- Drei Eingänge statt der typischen zwei Eingänge
- Adapter-basierte Schnittstelle statt direkter Daten-Ein-/Ausgänge
- Spezifische AX-Typ-Kompatibilität
- Unidirektionale Signalübertragung

Vergleich mit AND_3

- **`AX_AND_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

🛠️ Zugehörige Übungen

- Uebung_002a6_AX

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

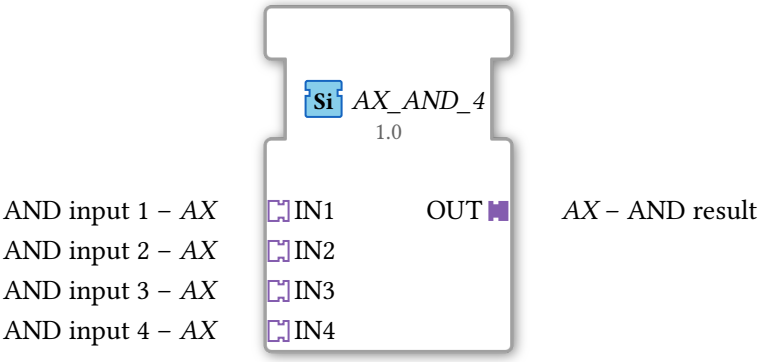
Der AX_AND_3_UNGATED ist ein spezialisierter Logikbaustein für Anwendungen, die eine dreifache UND-Verknüpfung mit standardisierten AX-Adaptern erfordern. Seine Adapter-basierte Architektur ermöglicht eine einfache Integration in bestehende Steuerungssysteme und bietet eine zuverlässige Lösung für komplexe logische Verknüpfungen.

AX_AND_4

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_AND_4 Funktionsblock ist ein generischer Baustein zur Berechnung einer logischen UND-Verknüpfung mit vier Eingängen. Er dient zur Verarbeitung von booleschen Signalen in Automatisierungssystemen und ermöglicht die Kombination mehrerer Eingangssignale zu einem gemeinsamen Ausgangssignal.



Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Plug-Adapter:

- **OUT**: AND-Ergebnis (Adapter-Typ: adapter::types::unidirectional::AX)

Socket-Adapter:

- **IN1**: UND-Eingang 1 (Adapter-Typ: adapter::types::unidirectional::AX)
- **IN2**: UND-Eingang 2 (Adapter-Typ: adapter::types::unidirectional::AX)
- **IN3**: UND-Eingang 3 (Adapter-Typ: adapter::types::unidirectional::AX)
- **IN4**: UND-Eingang 4 (Adapter-Typ: adapter::types::unidirectional::AX)

Funktionsweise
Der Funktionsblock berechnet eine logische UND-Verknüpfung über vier Eingänge. Das Ausgangssignal ist nur dann aktiv (TRUE), wenn alle vier Eingangssignale gleichzeitig aktiv (TRUE) sind. Sobald mindestens einer der vier Eingänge inaktiv (FALSE) ist, wird auch der Ausgang inaktiv.

Technische Besonderheiten

- Generischer Funktionsblock mit vordefiniertem Klassennamen ‚GEN_AX_AND‘
- Verwendet unidirektionale Adapter für die Signalübertragung
- Implementiert als Teil des Packages „adapter::booleanOperators“
- Unterstützt vier unabhängige Eingangssignale

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AX_AND_4 keine internen Zustände. Die Ausgabe wird ausschließlich durch die aktuellen Eingangswerte bestimmt.

Anwendungsszenarien

- Sicherheitssteuerungen, bei denen mehrere Bedingungen gleichzeitig erfüllt sein müssen
- Überwachung von Mehrfachsensoren
- Verknüpfung von Freigabesignalen in Prozesssteuerungen
- Logische Verknüpfung von Steuerbefehlen

Vergleich mit AND_4

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen UND-Bausteinen mit weniger Eingängen bietet AX_AND_4 die Möglichkeit, vier Signale direkt zu verknüpfen, ohne zusätzliche Verkettungen. Gegenüber ODER-Verknüpfungen stellt die UND-Verknüpfung strengere Anforderungen an die Aktivierung.

- `AX_AND_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der AX_AND_4 Funktionsblock bietet eine effiziente Lösung für die Verknüpfung von vier booleschen Eingangssignalen mittels logischer UND-Operation. Durch die Verwendung von Adaptern ermöglicht er eine flexible Integration in verschiedene Steuerungssysteme und eignet sich besonders für Anwendungen, bei denen mehrere Sicherheits- oder Freigabebedingungen gleichzeitig erfüllt sein müssen.

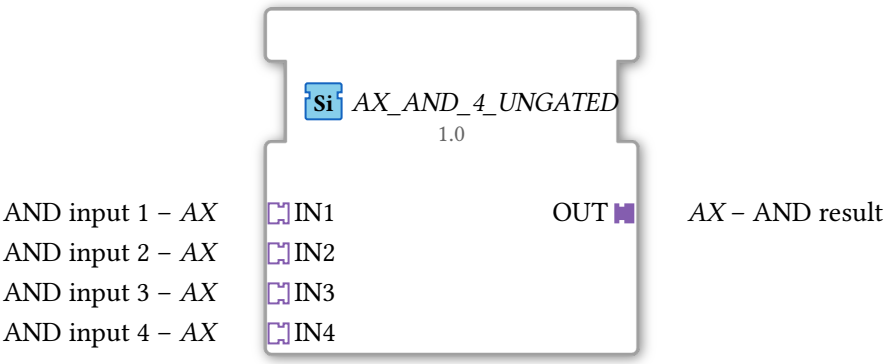
AX_AND_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AX_AND_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_AND_4_UNGATED Funktionsblock ist ein generischer Baustein zur Berechnung einer logischen UND-Verknüpfung mit vier Eingängen. Er dient zur Verarbeitung von booleschen Signalen in Automatisierungssystemen und ermöglicht die Kombination mehrerer Eingangssignale zu einem gemeinsamen Ausgangssignal.



Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Plug-Adapter:

- **OUT**: AND-Ergebnis (Adapter-Typ: adapter::types::unidirectional::AX)

Socket-Adapter:

- **IN1**: UND-Eingang 1 (Adapter-Typ: adapter::types::unidirectional::AX)
- **IN2**: UND-Eingang 2 (Adapter-Typ: adapter::types::unidirectional::AX)
- **IN3**: UND-Eingang 3 (Adapter-Typ: adapter::types::unidirectional::AX)
- **IN4**: UND-Eingang 4 (Adapter-Typ: adapter::types::unidirectional::AX)

Funktionsweise

Der Funktionsblock berechnet eine logische UND-Verknüpfung über vier Eingänge. Das Ausgangssignal ist nur dann aktiv (TRUE), wenn alle vier Eingangssignale gleichzeitig aktiv (TRUE) sind. Sobald mindestens einer der vier Eingänge inaktiv (FALSE) ist, wird auch der Ausgang inaktiv.

Technische Besonderheiten

- Generischer Funktionsblock mit vordefiniertem Klassennamen ‚GEN_AX_AND'
- Verwendet unidirektionale Adapter für die Signalübertragung
- Implementiert als Teil des Packages „adapter::booleanOperators"
- Unterstützt vier unabhängige Eingangssignale

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AX_AND_4_UNGATED keine internen Zustände. Die Ausgabe wird ausschließlich durch die aktuellen Eingangswerte bestimmt.

Anwendungsszenarien

- Sicherheitssteuerungen, bei denen mehrere Bedingungen gleichzeitig erfüllt sein müssen
- Überwachung von Mehrfachsensoren
- Verknüpfung von Freigabesignalen in Prozesssteuerungen
- Logische Verknüpfung von Steuerbefehlen

Vergleich mit AND_4

⚖️ **Vergleich mit ähnlichen Bausteinen**

Im Vergleich zu einfachen UND-Bausteinen mit weniger Eingängen bietet AX_AND_4_UNGATED die Möglichkeit, vier Signale direkt zu verknüpfen, ohne zusätzliche Verkettungen. Gegenüber ODER-Verknüpfungen stellt die UND-Verknüpfung strengere Anforderungen an die Aktivierung.

- **`AX_AND_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

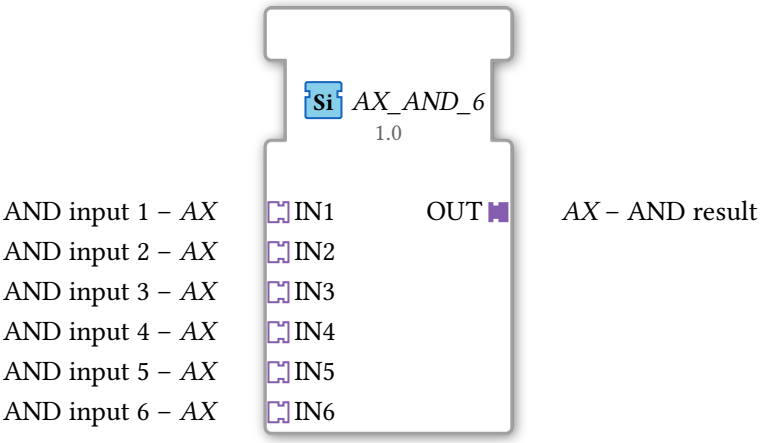
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der AX_AND_4_UNGATED Funktionsblock bietet eine effiziente Lösung für die Verknüpfung von vier booleschen Eingangssignalen mittels logischer UND-Operation. Durch die Verwendung von Adaptern ermöglicht er eine flexible Integration in verschiedene Steuerungssysteme und eignet sich besonders für Anwendungen, bei denen mehrere Sicherheits- oder Freigabebedingungen gleichzeitig erfüllt sein müssen.

AX_AND_6



Einleitung

Der `AX_AND_6` Funktionsblock ist ein generischer Baustein zur Berechnung einer logischen UND-Verknüpfung mit sechs unidirektionalen AX-Adapter-Eingängen. Er dient zur Bündelung und Sammelüberwachung von bis zu 6 Eingangssignalen (z. B. Not-Halt-Ketten, Freigabe- oder Statussignalen) zu einem gemeinsamen Adapter-Ausgangssignal.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Plug-Adapter (Ausgang):

- **OUT** – UND-Ergebnis (Adaptertyp: `adapter::types::unidirectional::AX`)

Socket-Adapter (Eingänge):

- **IN1** – UND-Eingang 1 (Adaptertyp: `adapter::types::unidirectional::AX`)
- **IN2** – UND-Eingang 2 (Adaptertyp: `adapter::types::unidirectional::AX`)
- **IN3** – UND-Eingang 3 (Adaptertyp: `adapter::types::unidirectional::AX`)
- **IN4** – UND-Eingang 4 (Adaptertyp: `adapter::types::unidirectional::AX`)
- **IN5** – UND-Eingang 5 (Adaptertyp: `adapter::types::unidirectional::AX`)
- **IN6** – UND-Eingang 6 (Adaptertyp: `adapter::types::unidirectional::AX`)

Funktionsweise

Der Funktionsblock wertet die logische UND-Funktion aller 6 AX-Eingänge aus. Der Ausgang `OUT.D1` ist nur dann `TRUE`, wenn alle sechs Sockets `IN1.D1` bis `IN6.D1` gleichzeitig `TRUE` sind. Sobald mindestens ein Eingang `FALSE` ist, schaltet der Ausgang auf `FALSE`.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event `OUT.E1` nur gesendet, wenn sich der neu berechnete Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt das

logische Ergebnis unverändert, wird kein Adapter-Event gesendet – dies schützt nachgelagerte SubApp-Ketten vor überflüssigem Event-Spamming.

Technische Besonderheiten

- **Generischer Funktionsblock**: Verwendet den System-Klassennamen `GEN_AX_AND` im Package `adapter::booleanOperators`.
- **Reine Adapter-Schnittstelle**: Ermöglicht eine direkte Anbindung ohne manuelle Daten-/Ereignis-sentkopplung.

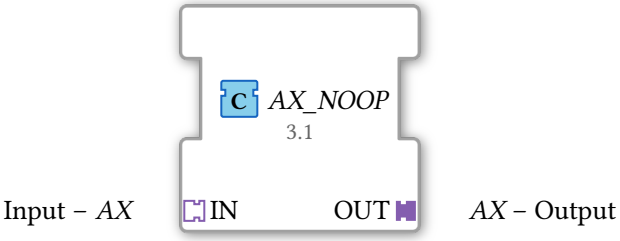
Anwendungsszenarien

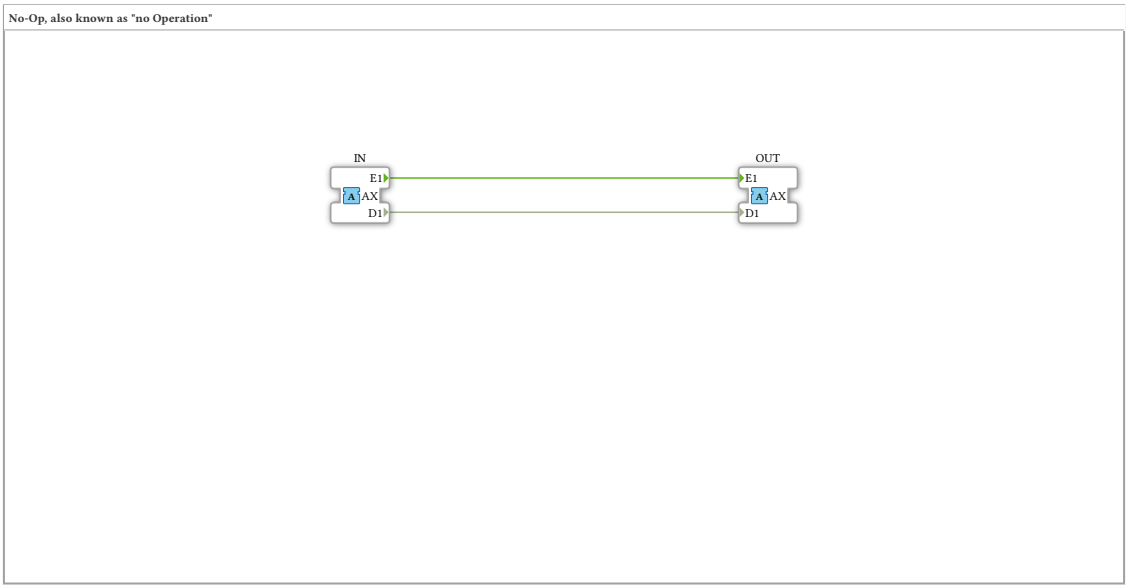
- Sammelüberwachung von bis zu 6 Not-Halt- oder Schutzschalterkanälen (z. B. Not-Halt STG1 I1–I6 auf Diagnoseseiten).
- Bündelung mehrerer Sicherheits- und Freigabekriterien.

Siehe auch

- `AX_AND_2` – 2-fach UND-Verknüpfung.
- `AX_AND_3` – 3-fach UND-Verknüpfung.
- `AX_AND_4` – 4-fach UND-Verknüpfung.

AX_NOOP





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AX_NOOP** (No Operation) dient als reine Durchleitungsfunktion. Er verbindet einen eingehenden Adapter (`IN`) direkt mit einem ausgehenden Adapter (`OUT`), ohne Daten zu verändern oder Ereignisse zu verarbeiten. Der FB ist als Basisbaustein für Test-, Debugging- und Platzhalterzwecke konzipiert.

Schnittstellenstruktur

Ereignis-Eingänge
Keine direkten Ereignis-Eingänge vorhanden. Ereignisse werden über den Adapter `IN` empfangen.

Ereignis-Ausgänge
Keine direkten Ereignis-Ausgänge vorhanden. Ereignisse werden über den Adapter `OUT` weitergegeben.

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden. Daten werden über den Adapter `IN` empfangen.

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden. Daten werden über den Adapter `OUT` weitergegeben.

Adapter

Typ	Name	Richtung	Kommentar
`adapter::types::unidirectional::AX`	IN	Socket (Eingang)	Empfängt Ereignisse und Daten
`adapter::types::unidirectional::AX`	OUT	Plug (Ausgang)	Gibt Ereignisse und Daten unverändert weiter

Funktionsweise
Der Baustein stellt eine Punkt-zu-Punkt-Verbindung zwischen dem Eingangsadapter `IN` und dem Ausgangsadapter `OUT` her. Intern werden folgende Verbindungen realisiert:

- **Ereignisverbindung**: `IN.E1` ⟶ `OUT.E1`
- **Datenverbindung**: `IN.D1` ⟶ `OUT.D1`

Jedes eingehende Ereignis an `IN.E1` wird sofort an `OUT.E1` weitergeleitet. Jeder Datenwert an `IN.D1` wird unverändert an `OUT.D1` durchgereicht. Es findet keine Logik, Verzögerung oder Zustandsänderung statt.

Technische Besonderheiten
- Reine Durchschaltung ohne Verarbeitung.
- Verwendet den Adaptertyp `AX` für die Kommunikation.
- Keine veränderbaren Parameter oder Konfigurationen.
- Ideal geeignet als Platzhalter oder zum Debugging von Adapterverbindungen in der 4diac-IDE.

Zustandsübersicht
Der Baustein besitzt keine eigene Zustandsmaschine. Sein Verhalten ist statisch und entspricht einer direkten Verdrahtung.

Anwendungsszenarien
- **Testumgebungen**: Ersetzen eines komplexen FB während der Testphase, um die Verdrahtung zu prüfen.
- **Debugging**: Einschleusen eines Mess- oder Trace-Punkts in einen Adapterpfad.
- **Platzhalter**: Vorübergehendes Einfügen bei der schrittweisen Implementierung eines größeren Systems.
- **Adapter-Transparenz**: Überprüfen, ob ein Adaptertyp korrekt verbunden ist, ohne Funktionslogik.

Vergleich mit ähnlichen Bausteinen

Baustein	Beschreibung	Unterschied
`E_NOP`	Ereignis-NOP mit direkten Ereignis-/Daten-Ein-/Ausgängen	Verwendet Standard-I/O statt Adapter
`MUX`	Auswahl zwischen mehreren Datenquellen	Bietet Auswahl, kein reiner Durchgriff
`BYE`	Explizite End-of-Service-Markierung	Besitzt spezielle Semantik, kein Durchgriff

AX_NOOP	Reine Adapter-Durchleitung	Speziell für den Adaptertyp `AX` ausgelegt

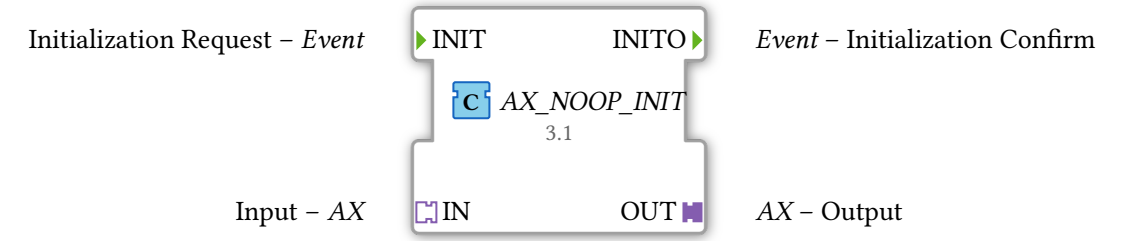
Fazit

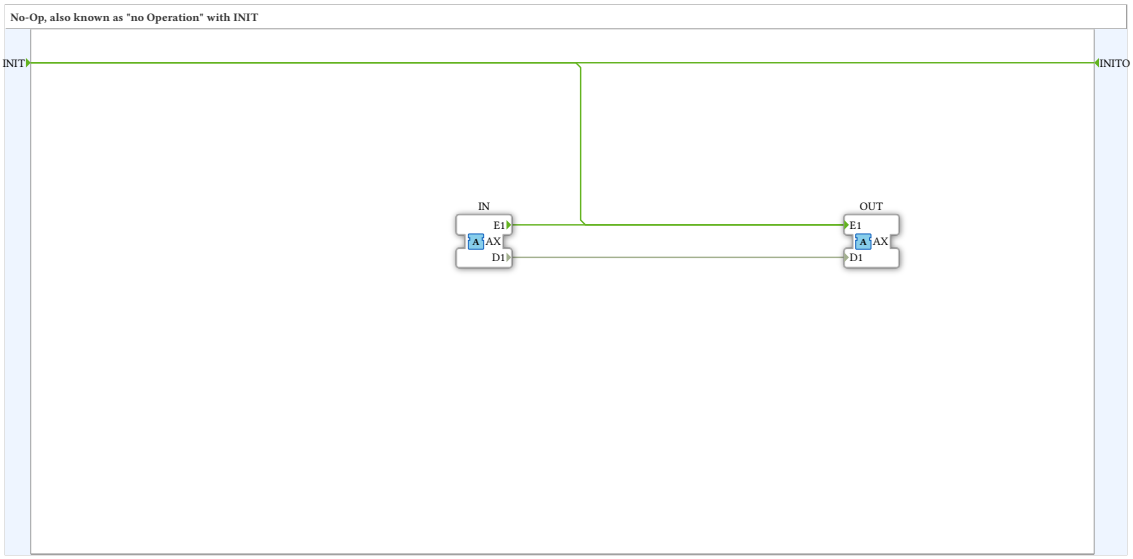
Der **AX_NOOP** ist ein minimalistischer, aber nützlicher Baustein für die Arbeit mit Adapterschnittstellen. Dank seiner einfachen Durchleitungsfunktion eignet er sich hervorragend zum Testen, Debuggen und als Platzhalter in der 4diac-IDE. Trotz fehlender eigener Logik erfüllt er eine wichtige Rolle beim schrittweisen Aufbau und der Verifikation von verbindungsintensiven Anwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AX_NOOP_INIT





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AX_NOOP_INIT` ist ein No-Operation-Baustein, der eingehende Ereignisse und Daten von einer Adapter-Schnittstelle unverändert an eine Ausgangs-Schnittstelle weiterleitet. Zusätzlich besitzt er eine Initialisierungsfunktion, die beim Eintreffen des INIT-Ereignisses sowohl die Bestätigung INITO als auch das Ausgangsereignis des Adapters auslöst. Er dient als einfacher Durchschleif- oder Platzhalter-Baustein in der Automatisierungstechnik.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (Typ: `EInit`): Initialisierungsanforderung. Löst beim Eintreffen die Ereignisausgänge `INITO` und `OUT.E1` aus.

Ereignis-Ausgänge

- **INITO** (Typ: `EInit`): Initialisierungsbestätigung. Wird direkt durch `INIT` ausgelöst.

Daten-Eingänge

Der Baustein besitzt keine separaten Dateneingänge auf der obersten Ebene. Sämtliche Daten werden über den eingehenden Adapter (`IN`) transportiert.

Daten-Ausgänge

Der Baustein besitzt keine separaten Datenausgänge auf der obersten Ebene. Sämtliche Daten werden über den ausgehenden Adapter (`OUT`) transportiert.

Adapter

- **IN** (Socket): Eingangsadapter vom Typ `adapter::types::unidirectional::AX`. Stellt eine Ereignisschnittstelle `E1` und eine Datenschnittstelle `D1` zur Verfügung.
- **OUT** (Plug): Ausgangsadapter vom Typ `adapter::types::unidirectional::AX`. Stellt eine Ereignisschnittstelle `E1` und eine Datenschnittstelle `D1` zur Verfügung.

Funktionsweise

Der Baustein arbeitet als reiner Durchlaufschieber:

1. **Ereignisweiterleitung**:
   - Das INIT-Ereignis wird nach `INIT0` und nach `OUT.E1` weitergeleitet.
   - Das Ereignis `IN.E1` wird direkt zu `OUT.E1` durchgeschliffen.
2. **Datenweiterleitung**:
   - Der Wert von `IN.D1` wird unverändert auf `OUT.D1` kopiert.

Es erfolgt keinerlei Verarbeitung oder Veränderung der Daten. Der Baustein ist zustandslos und führt lediglich die Signal- und Datenpfade zusammen.

Technische Besonderheiten

- Der Baustein kombiniert die reine Durchschleiffunktion eines No-Operation-Bausteins mit einer initialisierungsgesteuerten Ausgabe auf den Ausgangsadapter.
- Die Verwendung von Adaptern (plug/socket) ermöglicht eine saubere Kapselung und Wiederverwendbarkeit der Schnittstellendefinition.
- Es existieren keine internen Zustände oder Zeitverhalten – die Reaktion erfolgt unmittelbar bei Ereigniseingang.

Zustandsübersicht

Der `AX_NOOP_INIT` besitzt keine eigenen Zustände. Er reagiert sofort auf jedes eingehende Ereignis und leitet es samt den zugehörigen Daten weiter. Eine Zustandsmaschine ist nicht vorhanden.

Anwendungsszenarien

- **Initialisierungskaskaden**: Wenn mehrere Bausteine in einer Kette initialisiert werden müssen, kann `AX_NOOP_INIT` als Verbindungsglied dienen, das eine INIT-Anforderung sowohl bestätigt als auch an den nachfolgenden Baustein weitergibt.
- **Test- und Simulationsumgebungen**: Als Platzhalter, um Schnittstellen zu überbrücken und Funktionalitäten ohne tatsächliche Logik zu testen.
- **Adapter-Durchschleifung**: Wenn ein Adapter-Signal unverändert von einer Quelle zu einem Ziel geführt werden soll, ohne zusätzliche Verarbeitung.

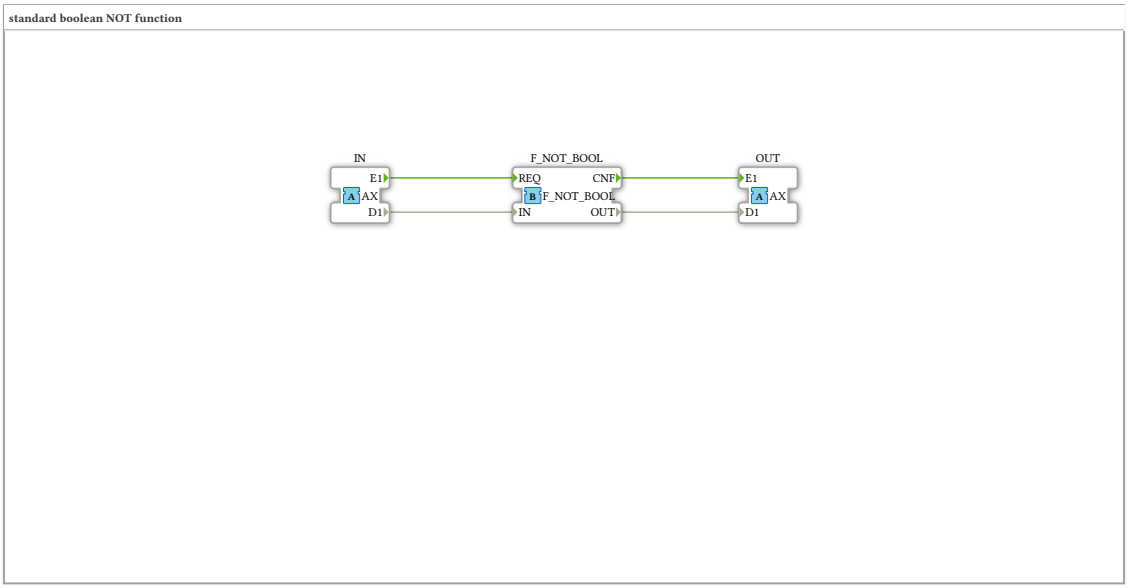
Vergleich mit ähnlichen Bausteinen

Ein reiner `AX_NOOP` (ohne INIT) würde nur die IN/OUT-Ereignisse und -Daten durchschleifen, aber keine separate Initialisierungsbestätigung (`INIT0`) generieren. Der `AX_NOOP_INIT` erweitert diese Funktion um die INIT-Behandlung, sodass er in Initialisierungsabläufen direkt eingesetzt werden kann.

Fazit

Der `AX_NOOP_INIT` ist ein einfacher, aber nützlicher Funktionsblock für die Durchleitung von Ereignissen und Daten auf Adapterebene. Seine INIT-Funktionalität macht ihn besonders in Initialisierungsketten oder als Test-Ersatzbaustein wertvoll. Durch seine Zustandslosigkeit ist er leicht verständlich und ressourcenschonend.

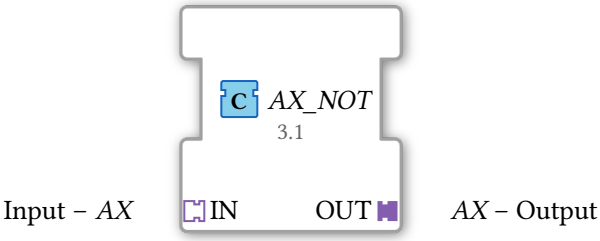
AX_NOT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_NOT Funktionsblock ist ein standardisierter boolescher NOT-Operator, der gemäß der Norm IEC 61131-3 implementiert wurde. Dieser Baustein führt eine logische Negation (NOT-Operation) auf den Eingangswerten durch und gibt das invertierte Ergebnis aus.



Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **IN** (Socket): Eingangsadapter vom Typ `adapter::types::unidirectional::AX` - dient zur Übertragung der Eingangsdaten
- **OUT** (Plug): Ausgangsadapter vom Typ `adapter::types::unidirectional::AX` - dient zur Ausgabe der verarbeiteten Daten

Funktionsweise

Der AX_NOT Funktionsblock führt eine logische Negation auf den über den IN-Adapter empfangenen booleschen Werten durch. Das Ergebnis der NOT-Operation wird über den OUT-Adapter ausgegeben. Die Verarbeitung erfolgt bidirektional zwischen Anwendungs- und Ressourcenebene.

Technische Besonderheiten

- Implementiert gemäß IEC 61131-3 Standard für boolesche Funktionen
- Verwendet unidirektionale AX-Adapter für die Datenübertragung
- Arbeitet mit dem Package `adapter::booleanOperators`
- Unterstützt TypeHash-Funktionalität für Typidentifikation

Zustandsübersicht

Der Funktionsblock arbeitet zustandslos und führt die NOT-Operation direkt auf den eingehenden Daten durch, ohne internen Zustand zu speichern.

Anwendungsszenarien

- Logische Invertierung von booleschen Signalen in Steuerungsanwendungen
- Signalumkehr in Automatisierungssystemen
- Verwendung in kombinatorischen Schaltungen
- Integration in größere Steuerungslogiken zur Signalverarbeitung

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen booleschen Operatoren wie AND oder OR führt AX_NOT ausschließlich die Negationsoperation durch. Die Verwendung von Adaptern anstelle direkter Ein-/Ausgänge ermöglicht eine flexiblere Integration in komplexere Systemarchitekturen.

Vergleich mit F_NOT

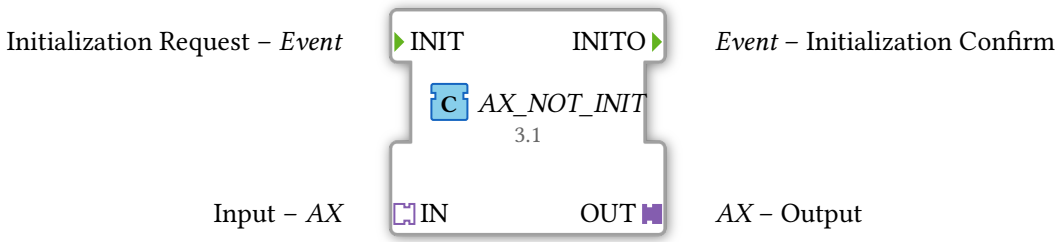
🛠️ Zugehörige Übungen

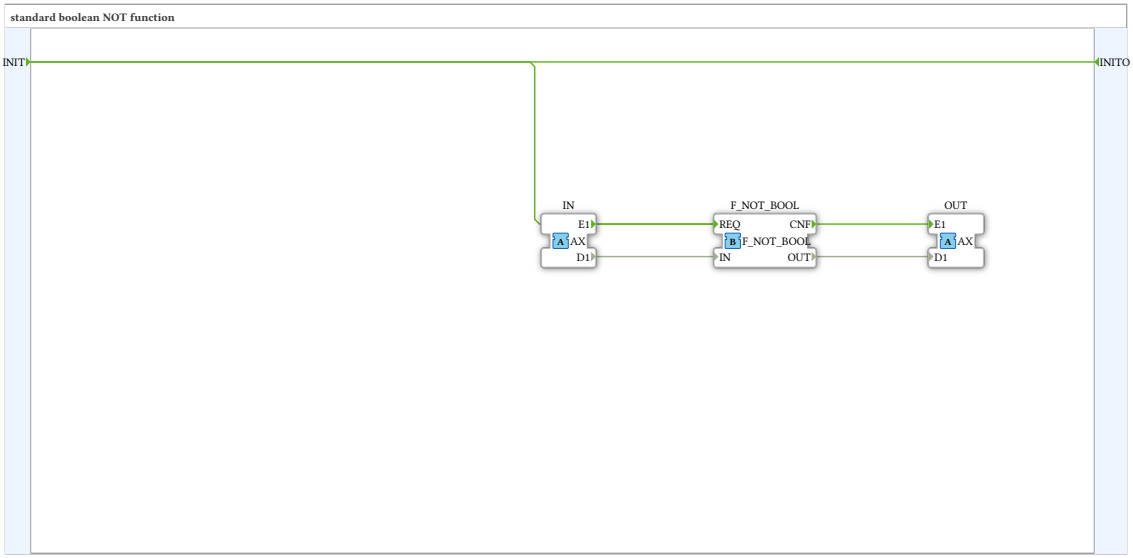
- Uebung_006a3_sub_AX

Fazit

Der AX_NOT Funktionsblock bietet eine zuverlässige und standardkonforme Implementierung der booleschen NOT-Operation. Durch die Adapter-basierte Schnittstellengestaltung eignet er sich besonders für modulare Systemdesigns und ermöglicht eine einfache Integration in bestehende Automatisierungslösungen.

AX_NOT_INIT





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AX_NOT_INIT** realisiert eine boolesche Negation (NOT-Funktion) auf Basis eines Adaptertyps. Im Gegensatz zu klassischen NOT-Bausteinen arbeitet er über eine standardisierte Adapterschnittstelle und ist speziell für die Verwendung in modularen Automatisierungssystemen konzipiert. Die Initialisierung erfolgt über den Ereigniseingang INIT, der die erste Berechnung auslöst.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Initialisierungsanforderung

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Initialisierungsbestätigung

Daten-Eingänge

Der Baustein besitzt keine direkten Dateneingänge. Die Eingangsdaten werden über den Adapter **IN** bezogen.

Daten-Ausgänge

Der Baustein besitzt keine direkten Datenausgänge. Die Ausgangsdaten werden über den Adapter **OUT** bereitgestellt.

Adapter

Name	Typ	Kommentar	Richtung
IN	adapter::types::unidirectional::AX	Eingangswert	Socket
OUT	adapter::types::unidirectional::AX	Ausgangswert (negiert)	Plug

Funktionsweise

Der Baustein führt eine bitweise Negation des über den **IN**-Adapter bereitgestellten Wertes durch und gibt das Ergebnis über den **OUT**-Adapter aus. Die Verarbeitung wird durch ein Ereignis am **IN.E1** (Ereignis aus dem IN-Adapter) oder durch das INIT-Ereignis gestartet. Interne Schritte:

1. Ein Ereignis an **INIT** oder **IN.E1** aktiviert den eingebetteten Funktionsblock **F_NOT_BOOL**.
2. **F_NOT_BOOL** holt den Wert von **IN.D1**, invertiert ihn und schreibt das Ergebnis nach **OUT.D1**.
3. Nach Abschluss wird ein Ereignis über **OUT.E1** (an den OUT-Adapter) sowie **INITO** ausgegeben.

Wird das INIT-Ereignis empfangen, so werden **INITO** und **OUT.E1** sofort ausgelöst – die erste Negation erfolgt ohne vorheriges Eingangsereignis.

Technische Besonderheiten

- **Adapterbasierte Kommunikation**: Der Baustein verwendet ausschließlich unidirektionale Adapter vom Typ `AX`, wodurch die Daten- und Ereignisschnittstellen modular gekapselt werden. Dies erleichtert den Austausch zwischen Komponenten mit standardisierten Protokollen.
- **Initialisierungsverhalten**: Der INIT-Eingang bewirkt eine sofortige Erstberechnung und gibt ein Bestätigungsereignis aus, was für initiale Wertsetzungen oder Reset-Abläufe nützlich ist.
- **Wiederverwendung**: Intern kommt der IEC-61131-3-Funktionsblock `F_NOT_BOOL` zum Einsatz, der eine einfache Negation auf booleschen Werten realisiert.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten – er reagiert ereignisgesteuert. Vereinfachter Ablauf:

```
Ereignis (INIT oder IN.E1)
  → Erfasse IN.D1
  → Berechne NOT
  → Setze OUT.D1
  → Sende INITO und/oder OUT.E1
```

Der Baustein ist **nicht zustandsbehaftet**, d.h. jede neue Anforderung wird unabhängig von vorherigen Werten verarbeitet.

Anwendungsszenarien

- **Signalnegation in modularen Systemen**: Wenn Ein- und Ausgänge über Adapter definiert sind, kann der Baustein als universeller NOT-Gatter eingesetzt werden, z.B. zur Invertierung von Aktor- oder Sensorwerten.
- **Initialisierungslogik**: Über den INIT-Eingang kann sichergestellt werden, dass der Ausgang beim Start einen definierten negierten Wert liefert.
- **Adapter-Kompatibilität**: In Systemen mit dem `AX`-Adapterprotokoll dient der Baustein als funktionaler Grundbaustein für logische Verknüpfungen.

Vergleich mit ähnlichen Bausteinen

- **F_NOT_BOOL**: Standard IEC-61131-3 NOT-Funktion ohne Adapter. **AX_NOT_INIT** kapselt diese Funktionalität in eine Adapter-Schnittstelle und fügt ein INIT-Verhalten hinzu.
- **AX_AND / AX_OR**: Analoge Adapterbausteine für UND bzw. ODER. **AX_NOT_INIT** ist spezifisch für die Negation und besitzt nur einen Eingang.
- **F_TRIG / R_TRIG**: Flankenerkennung, die ebenfalls Ereignisse nutzt, aber keine logische Negation durchführt.

Fazit

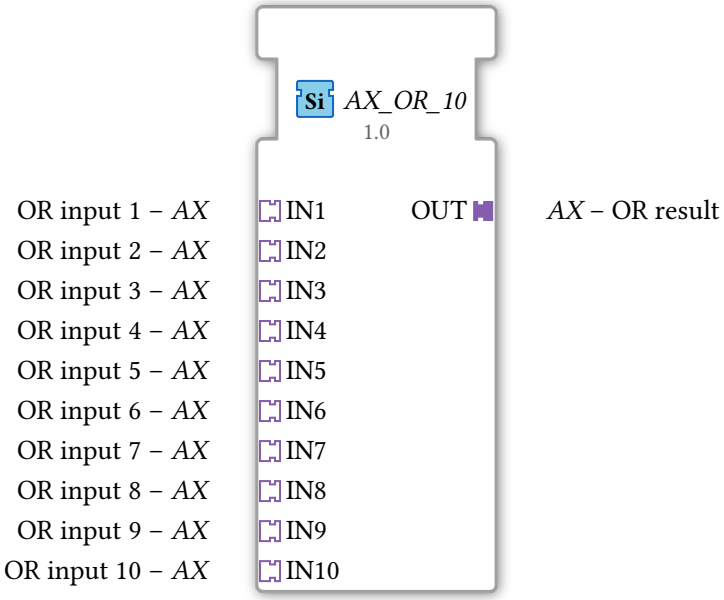
Der **AX_NOT_INIT**-Funktionsblock bietet eine effiziente und standardisierte Möglichkeit, boolesche Werte über den `AX`-Adapter zu negieren. Durch die Integration einer Initialisierungslogik eignet er sich besonders für Systeme, die beim Start einen definierten Ausgangszustand benötigen. Er ergänzt die Familie adapterbasierter Logikbausteine und vereinfacht die modulare Strukturierung von Automatisierungslösungen.

AX_OR_10

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_OR_10 ist ein generischer Funktionsblock zur Berechnung der logischen ODER-Verknüpfung von 10 Eingängen.



Schnittstellenstruktur

Adapter

Eingangsadapter:

- **IN1** bis **IN10** (adapter::types::unidirectional::AX)

Ausgangsadapter:

- **OUT** (adapter::types::unidirectional::AX)

Funktionsweise

Der Funktionsblock führt eine logische ODER-Verknüpfung der 10 Eingangssignale durch.

Technische Besonderheiten

- Generischer Funktionsblock mit spezifischem Klassennamen ‚GEN_AX_OR'
- Verwendet unidirektionale Adapter.

Zustandsübersicht

Kombinatorischer Logikbaustein ohne Zustände.

Anwendungsszenarien

Logische Verknüpfungen mit vielen Eingängen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **AX_OR_2…9**: Varianten mit weniger Eingängen.
- `AX_OR_10_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Adapter-basierter ODER-Baustein mit 10 Eingängen.

AX_OR_10_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AX_OR_10`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_OR_10_UNGATED ist ein generischer Funktionsblock zur Berechnung der logischen ODER-Verknüpfung von 10 Eingängen.

Schnittstellenstruktur

Adapter

Eingangsadapter:

- **IN1** bis **IN10** (adapter::types::unidirectional::AX)

Ausgangsadapter:

- **OUT** (adapter::types::unidirectional::AX)

Funktionsweise

Der Funktionsblock führt eine logische ODER-Verknüpfung der 10 Eingangssignale durch.

Technische Besonderheiten

- Generischer Funktionsblock mit spezifischem Klassennamen ‚GEN_AX_OR'
- Verwendet unidirektionale Adapter.

Zustandsübersicht

Kombinatorischer Logikbaustein ohne Zustände.

Anwendungsszenarien

Logische Verknüpfungen mit vielen Eingängen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **AX_OR_2...9**: Varianten mit weniger Eingängen.
- **AX_OR_10**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

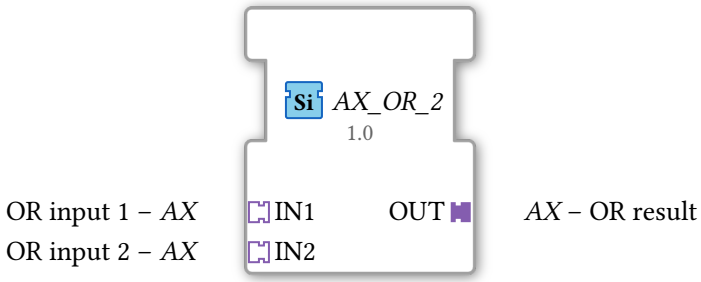
Adapter-basierter ODER-Baustein mit 10 Eingängen.

AX_OR_2

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_OR_2 Funktionsblock ist ein generischer Baustein zur Berechnung der booleschen ODER-Verknüpfung. Er verarbeitet zwei Eingangssignale und gibt das logische ODER-Ergebnis aus. Der Baustein ist speziell für die Verwendung in Automatisierungssystemen konzipiert.



Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1** - ODER Eingang 1 (Adapter-Typ: adapter::types::unidirectional::AX)
- **IN2** - ODER Eingang 2 (Adapter-Typ: adapter::types::unidirectional::AX)

Ausgangsadapter:

- **OUT** - ODER Ergebnis (Adapter-Typ: adapter::types::unidirectional::AX)

Funktionsweise

Der AX_OR_2 Funktionsblock führt eine logische ODER-Verknüpfung zwischen den beiden Eingangssignalen IN1 und IN2 durch. Das Ergebnis wird über den Ausgangsadapter OUT ausgegeben. Die Verarbeitung erfolgt kontinuierlich, sobald Eingangssignale verfügbar sind.

Technische Besonderheiten

- Generischer Funktionsblock mit der Klasse ‚GEN_AX_OR‘
- Verwendet unidirektionale Adapter für die Signalübertragung
- Implementiert als Teil des Packages „adapter::booleanOperators“
- Unterstützt die Eclipse 4diac Standard-Spezifikation 61499-2

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt AX_OR_2 keinen internen Zustand. Die Ausgabe wird ausschließlich von den aktuellen Eingangswerten bestimmt.

Anwendungsszenarien

- Sicherheitskreise mit redundanten Sensoren
- Parallele Steuerungsbefehle
- Überwachungssysteme mit mehreren Eingangssignalen
- Logische Verknüpfungen in Steuerungsalgorithmen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen ODER-Bausteinen zeichnet sich AX_OR_2 durch seine Adapter-basierte Schnittstelle aus, die eine flexible Integration in verschiedene Systemarchitekturen ermöglicht. Die unidirektionalen Adapter gewährleisten eine klare Signalflussrichtung.

Vergleich mit OR_2

- **`AX_OR_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

🛠️ Zugehörige Übungen

- Uebung_002a3_AX
- Uebung_002b3_AX
- Uebung_035a2_AX
- Uebung_035a3_AX
- Uebung_160_AX
- Uebung_160b2_AX
- Uebung_160b_AX

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der AX_OR_2 Funktionsblock bietet eine zuverlässige und einfach zu integrierende Lösung für boolesche ODER-Verknüpfungen in Automatisierungssystemen. Seine Adapter-basierte Architektur ermöglicht eine flexible Verwendung in verschiedenen Steuerungskontexten.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

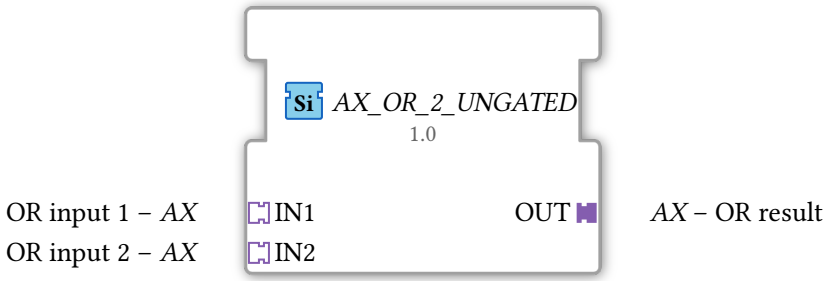
AX_OR_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AX_OR_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_OR_2_UNGATED Funktionsblock ist ein generischer Baustein zur Berechnung der booleschen ODER-Verknüpfung. Er verarbeitet zwei Eingangssignale und gibt das logische ODER-Ergebnis aus. Der Baustein ist speziell für die Verwendung in Automatisierungssystemen konzipiert.



Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1** - ODER Eingang 1 (Adapter-Typ: adapter::types::unidirectional::AX)
- **IN2** - ODER Eingang 2 (Adapter-Typ: adapter::types::unidirectional::AX)

Ausgangsadapter:

- **OUT** - ODER Ergebnis (Adapter-Typ: adapter::types::unidirectional::AX)

Funktionsweise

Der AX_OR_2_UNGATED Funktionsblock führt eine logische ODER-Verknüpfung zwischen den beiden Eingangssignalen IN1 und IN2 durch. Das Ergebnis wird über den Ausgangsadapter OUT ausgegeben. Die Verarbeitung erfolgt kontinuierlich, sobald Eingangssignale verfügbar sind.

Technische Besonderheiten

- Generischer Funktionsblock mit der Klasse ‚GEN_AX_OR‘
- Verwendet unidirektionale Adapter für die Signalübertragung
- Implementiert als Teil des Packages „adapter::booleanOperators“
- Unterstützt die Eclipse 4diac Standard-Spezifikation 61499-2

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt AX_OR_2_UNGATED keinen internen Zustand. Die Ausgabe wird ausschließlich von den aktuellen Eingangswerten bestimmt.

Anwendungsszenarien

- Sicherheitskreise mit redundanten Sensoren
- Parallele Steuerungsbefehle
- Überwachungssysteme mit mehreren Eingangssignalen
- Logische Verknüpfungen in Steuerungsalgorithmen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen ODER-Bausteinen zeichnet sich AX_OR_2_UNGATED durch seine Adapter-basierte Schnittstelle aus, die eine flexible Integration in verschiedene Systemarchitekturen ermöglicht. Die unidirektionalen Adapter gewährleisten eine klare Signalflussrichtung.

Vergleich mit OR_2

- **AX_OR_2**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

🛠️ Zugehörige Übungen

- Uebung_002a3_AX
- Uebung_002b3_AX
- Uebung_035a2_AX
- Uebung_035a3_AX
- Uebung_160_AX
- Uebung_160b2_AX
- Uebung_160b_AX

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der AX_OR_2_UNGATED Funktionsblock bietet eine zuverlässige und einfach zu integrierende Lösung für boolesche ODER-Verknüpfungen in Automatisierungssystemen. Seine Adapter-basierte Architektur ermöglicht eine flexible Verwendung in verschiedenen Steuerungskontexten.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

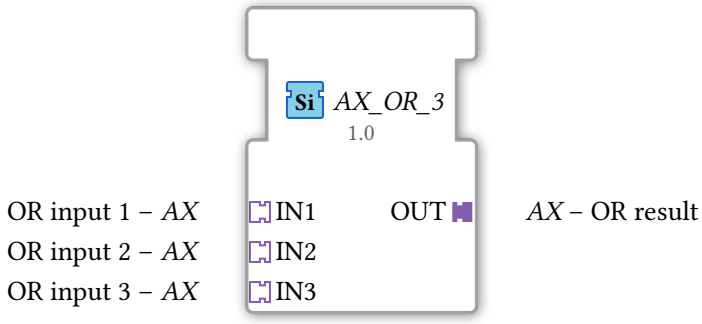
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AX_OR_3

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_OR_3 Funktionsblock ist ein generischer Baustein zur Berechnung der logischen ODER-Verknüpfung mit drei Eingängen. Er dient zur Verarbeitung von booleschen Signalen in Steuerungsanwendungen und ermöglicht die flexible Kombination mehrerer Eingangssignale zu einem Ausgangssignal.



Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereigniseingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignisausgänge vorhanden.

Daten-Eingänge

Keine direkten Dateneingänge vorhanden.

Daten-Ausgänge

Keine direkten Datenausgänge vorhanden.

Adapter

Eingangsadapter:

- **IN1**: OR-Eingang 1 (Typ: adapter::types::unidirectional::AX)
- **IN2**: OR-Eingang 2 (Typ: adapter::types::unidirectional::AX)
- **IN3**: OR-Eingang 3 (Typ: adapter::types::unidirectional::AX)

Ausgangsadapter:

- **OUT**: OR-Ergebnis (Typ: adapter::types::unidirectional::AX)

Funktionsweise

Der Funktionsblock berechnet kontinuierlich die logische ODER-Verknüpfung der drei Eingangssignale. Das Ausgangssignal ist wahr (TRUE), wenn mindestens einer der drei Eingänge wahr ist. Nur wenn alle drei Eingänge falsch (FALSE) sind, wird der Ausgang ebenfalls falsch.

Die logische Funktion lässt sich wie folgt darstellen: OUT = IN1 OR IN2 OR IN3

Technische Besonderheiten

- Verwendet unidirektionale Adapter für die Signalübertragung
- Implementiert als generischer Funktionsblock mit der Klasse ‚GEN_AX_OR'
- Arbeitet ohne Ereignissteuerung, berechnet kontinuierlich
- Optimiert für die Verwendung in Adapter-basierten Architekturen

Zustandsübersicht

Da es sich um einen kombinatorischen Baustein ohne Speicherfunktion handelt, besitzt AX_OR_3 keine internen Zustände. Die Ausgabe wird ausschließlich durch die aktuellen Eingangswerte bestimmt.

Anwendungsszenarien

- Sicherheitskreise mit mehreren Not-Aus-Tastern
- Überwachungssysteme mit mehreren Sensoren
- Steuerungslogik mit alternativen Aktivierungsbedingungen
- Verknüpfung von Statusmeldungen aus verschiedenen Quellen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-ODER-Bausteinen bietet AX_OR_3 den Vorteil von drei Eingängen in einem einzigen Baustein, was die Verdrahtung vereinfacht. Gegenüber Bausteinen mit variabler Eingangsanzahl ist AX_OR_3 spezifisch optimiert für den dreifachen ODER-Einsatz.

Vergleich mit OR_3

- `AX_OR_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

🛠️ Zugehörige Übungen

- Uebung_002a5_AX
- Uebung_002a5b_AX

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der AX_OR_3 Funktionsblock stellt eine effiziente Lösung für ODER-Verknüpfungen mit drei Eingängen dar. Durch die Adapter-basierte Schnittstelle ermöglicht er eine flexible Integration in verschiedene

Steuerungsarchitekturen und eignet sich besonders für Anwendungen, bei denen mehrere Bedingungen alternativ erfüllt sein können.

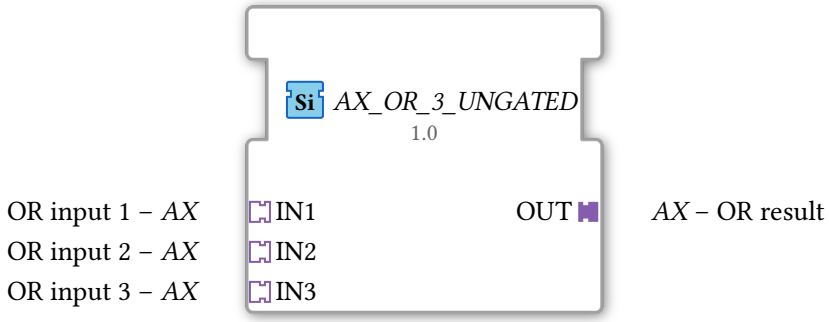
AX_OR_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AX_OR_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_OR_3_UNGATED Funktionsblock ist ein generischer Baustein zur Berechnung der logischen ODER-Verknüpfung mit drei Eingängen. Er dient zur Verarbeitung von booleschen Signalen in Steuerungsanwendungen und ermöglicht die flexible Kombination mehrerer Eingangssignale zu einem Ausgangssignal.



Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereigniseingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignisausgänge vorhanden.

Daten-Eingänge

Keine direkten Dateneingänge vorhanden.

Daten-Ausgänge

Keine direkten Datenausgänge vorhanden.

Adapter

Eingangsadapter:

- **IN1**: OR-Eingang 1 (Typ: adapter::types::unidirectional::AX)
- **IN2**: OR-Eingang 2 (Typ: adapter::types::unidirectional::AX)
- **IN3**: OR-Eingang 3 (Typ: adapter::types::unidirectional::AX)

Ausgangsadapter:

- **OUT**: OR-Ergebnis (Typ: adapter::types::unidirectional::AX)

Funktionsweise

Der Funktionsblock berechnet kontinuierlich die logische ODER-Verknüpfung der drei Eingangssignale. Das Ausgangssignal ist wahr (TRUE), wenn mindestens einer der drei Eingänge wahr ist. Nur wenn alle drei Eingänge falsch (FALSE) sind, wird der Ausgang ebenfalls falsch.

Die logische Funktion lässt sich wie folgt darstellen: OUT = IN1 OR IN2 OR IN3

Technische Besonderheiten

- Verwendet unidirektionale Adapter für die Signalübertragung
- Implementiert als generischer Funktionsblock mit der Klasse ‚GEN_AX_OR'
- Arbeitet ohne Ereignissteuerung, berechnet kontinuierlich
- Optimiert für die Verwendung in Adapter-basierten Architekturen

Zustandsübersicht

Da es sich um einen kombinatorischen Baustein ohne Speicherfunktion handelt, besitzt AX_OR_3_UNGATED keine internen Zustände. Die Ausgabe wird ausschließlich durch die aktuellen Eingangswerte bestimmt.

Anwendungsszenarien

- Sicherheitskreise mit mehreren Not-Aus-Tastern
- Überwachungssysteme mit mehreren Sensoren
- Steuerungslogik mit alternativen Aktivierungsbedingungen
- Verknüpfung von Statusmeldungen aus verschiedenen Quellen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-ODER-Bausteinen bietet AX_OR_3_UNGATED den Vorteil von drei Eingängen in einem einzigen Baustein, was die Verdrahtung vereinfacht. Gegenüber Bausteinen mit variabler Eingangsanzahl ist AX_OR_3_UNGATED spezifisch optimiert für den dreifachen ODER-Einsatz.

Vergleich mit OR_3

- **`AX_OR_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

🛠️ Zugehörige Übungen

- Uebung_002a5_AX
- Uebung_002a5b_AX

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

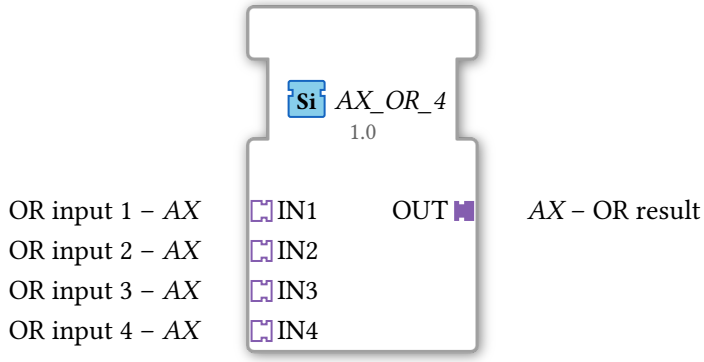
Der AX_OR_3_UNGATED Funktionsblock stellt eine effiziente Lösung für ODER-Verknüpfungen mit drei Eingängen dar. Durch die Adapter-basierte Schnittstelle ermöglicht er eine flexible Integration in verschiedene Steuerungsarchitekturen und eignet sich besonders für Anwendungen, bei denen mehrere Bedingungen alternativ erfüllt sein können.

AX_OR_4

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_OR_4 ist ein generischer Funktionsblock zur Berechnung einer logischen ODER-Verknüpfung mit vier Eingängen. Der Baustein dient zur Verarbeitung von booleschen Signalen und gibt das Ergebnis der ODER-Operation über einen Adapter-Ausgang aus.



Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden.

Adapter
Eingehende Adapter (Sockets):

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AX)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AX)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AX)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AX)

Ausgehende Adapter (Plugs):

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AX)

Funktionsweise
Der Funktionsblock berechnet kontinuierlich die logische ODER-Verknüpfung der vier Eingangssignale. Das Ergebnis wird über den ausgehenden Adapter OUT ausgegeben. Die ODER-Operation liefert ein logisches TRUE, wenn mindestens einer der vier Eingänge TRUE ist. Nur wenn alle vier Eingänge FALSE sind, wird das Ergebnis FALSE.

Technische Besonderheiten

- Verwendet unidirektionale Adapter für die Kommunikation
- Implementiert als generischer Funktionsblock mit der Klasse ‚GEN_AX_OR'
- Arbeitet mit dem Adapter-Typ AX für boolesche Werte
- Keine Ereignissteuerung - arbeitet kontinuierlich

Zustandsübersicht
Da es sich um einen kombinatorischen Baustein ohne Ereignissteuerung handelt, besitzt der AX_OR_4 keine Zustandsmaschine. Die Ausgabe wird kontinuierlich basierend auf den aktuellen Eingangswerten berechnet.

Anwendungsszenarien

- Sicherheitskreise mit mehreren Not-Aus-Tastern
- Überwachungssysteme mit mehreren Sensoren
- Steuerungslogik mit parallelen Bedingungen

- Alarmierungsysteme mit mehreren Auslösern

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren ODER-Bausteinen bietet AX_OR_4 den Vorteil von vier Eingängen in einem einzigen Baustein, was die Verdrahtung vereinfacht und Platz spart. Gegenüber Bausteinen mit Ereignissteuerung arbeitet AX_OR_4 kontinuierlich ohne explizite Auslöseereignisse.

Vergleich mit OR_4

- `AX_OR_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der AX_OR_4 ist ein effizienter und kompakter Baustein für logische ODER-Verknüpfungen mit vier Eingängen. Seine Adapter-basierte Schnittstelle ermöglicht eine flexible Integration in größere Steuerungssysteme, während die kontinuierliche Arbeitsweise eine sofortige Reaktion auf Eingangsänderungen gewährleistet.

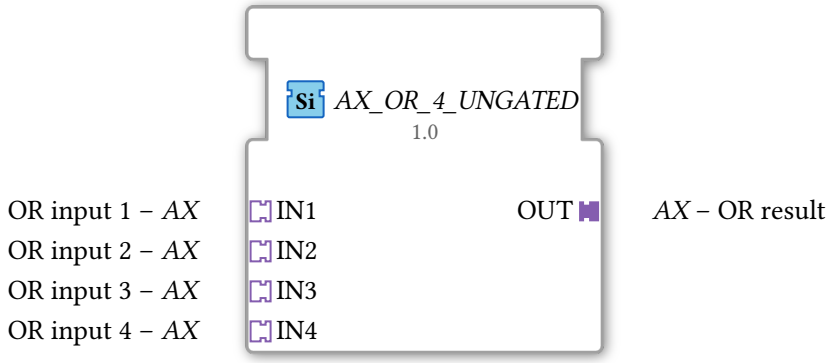
AX_OR_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AX_OR_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_OR_4_UNGATED ist ein generischer Funktionsblock zur Berechnung einer logischen ODER-Verknüpfung mit vier Eingängen. Der Baustein dient zur Verarbeitung von booleschen Signalen und gibt das Ergebnis der ODER-Operation über einen Adapter-Ausgang aus.



Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Eingehende Adapter (Sockets):

- **IN1**: ODER-Eingang 1 (Typ: adapter::types::unidirectional::AX)
- **IN2**: ODER-Eingang 2 (Typ: adapter::types::unidirectional::AX)
- **IN3**: ODER-Eingang 3 (Typ: adapter::types::unidirectional::AX)
- **IN4**: ODER-Eingang 4 (Typ: adapter::types::unidirectional::AX)

Ausgehende Adapter (Plugs):

- **OUT**: ODER-Ergebnis (Typ: adapter::types::unidirectional::AX)

Funktionsweise

Der Funktionsblock berechnet kontinuierlich die logische ODER-Verknüpfung der vier Eingangssignale. Das Ergebnis wird über den ausgehenden Adapter OUT ausgegeben. Die ODER-Operation liefert ein logisches TRUE, wenn mindestens einer der vier Eingänge TRUE ist. Nur wenn alle vier Eingänge FALSE sind, wird das Ergebnis FALSE.

Technische Besonderheiten

- Verwendet unidirektionale Adapter für die Kommunikation
- Implementiert als generischer Funktionsblock mit der Klasse ‚GEN_AX_OR'
- Arbeitet mit dem Adapter-Typ AX für boolesche Werte
- Keine Ereignissteuerung - arbeitet kontinuierlich

Zustandsübersicht

Da es sich um einen kombinatorischen Baustein ohne Ereignissteuerung handelt, besitzt der AX_OR_4_UNGATED keine Zustandsmaschine. Die Ausgabe wird kontinuierlich basierend auf den aktuellen Eingangswerten berechnet.

Anwendungsszenarien

- Sicherheitskreise mit mehreren Not-Aus-Tastern
- Überwachungssysteme mit mehreren Sensoren
- Steuerungslogik mit parallelen Bedingungen
- Alarmierungsysteme mit mehreren Auslösern

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren ODER-Bausteinen bietet AX_OR_4_UNGATED den Vorteil von vier Eingängen in einem einzigen Baustein, was die Verdrahtung vereinfacht und Platz spart. Gegenüber Bausteinen mit Ereignissteuerung arbeitet AX_OR_4_UNGATED kontinuierlich ohne explizite Auslöseereignisse.

Vergleich mit OR_4

- **`AX_OR_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

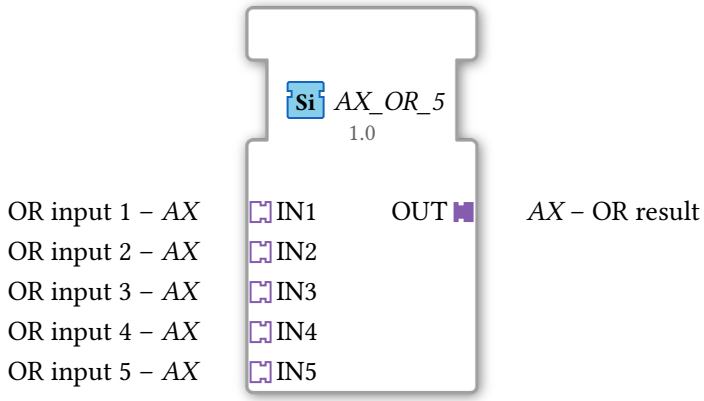
Der AX_OR_4_UNGATED ist ein effizienter und kompakter Baustein für logische ODER-Verknüpfungen mit vier Eingängen. Seine Adapter-basierte Schnittstelle ermöglicht eine flexible Integration in größere Steuerungssysteme, während die kontinuierliche Arbeitsweise eine sofortige Reaktion auf Eingangsänderungen gewährleistet.

AX_OR_5

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_OR_5 Funktionsblock ist ein generischer Baustein zur Berechnung der booleschen ODER-Verknüpfung mit fünf Eingängen. Er dient zur logischen Verarbeitung von Signalen in Automatisierungssystemen und ermöglicht die Kombination mehrerer Eingangssignale zu einem Ausgangssignal.



Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Plug-Adapter:

- **OUT**: Unidirektionaler Adapter für das ODER-Ergebnis

Socket-Adapter:

- **IN1**: Unidirektionaler Adapter für ODER-Eingang 1
- **IN2**: Unidirektionaler Adapter für ODER-Eingang 2
- **IN3**: Unidirektionaler Adapter für ODER-Eingang 3
- **IN4**: Unidirektionaler Adapter für ODER-Eingang 4
- **IN5**: Unidirektionaler Adapter für ODER-Eingang 5

Funktionsweise

Der Funktionsblock berechnet die logische ODER-Verknüpfung der fünf Eingangssignale. Das Ausgangssignal ist TRUE, wenn mindestens einer der fünf Eingänge TRUE ist. Sind alle Eingänge FALSE, wird das Ausgangssignal ebenfalls FALSE. Die Verarbeitung erfolgt über Adapter-Schnittstellen, wobei alle Ein- und Ausgänge unidirektional konfiguriert sind.

Technische Besonderheiten

- Generischer Funktionsblock mit fünf Eingängen
- Verwendung von unidirektionalen Adaptern für alle Schnittstellen
- Implementiert als ODER-Verknüpfung mit fünf Operanden
- Keine Ereignissteuerung, kontinuierliche Signalverarbeitung

Zustandsübersicht

Der Baustein besitzt keinen internen Zustand und arbeitet stateless. Die Ausgabe wird kontinuierlich basierend auf den aktuellen Eingangswerten berechnet.

Anwendungsszenarien

- Sicherheitskreise mit mehreren Not-Aus-Tastern
- Überwachungssysteme mit mehreren Sensoren
- Steuerungslogik mit parallelen Bedingungen
- Alarmierungssysteme mit mehreren Auslösern

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-ODER-Bausteinen mit weniger Eingängen bietet AX_OR_5 die Möglichkeit, fünf Signale gleichzeitig zu verarbeiten, was die Verdrahtung vereinfacht und Platz spart. Gegenüber kaskadierten ODER-Verknüpfungen bietet dieser Baustein eine integrierte Lösung.

Vergleich mit OR_5

- `AX_OR_5_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der AX_OR_5 Funktionsblock stellt eine effiziente Lösung für ODER-Verknüpfungen mit fünf Eingängen dar. Durch die Verwendung von Adaptern ermöglicht er eine flexible Integration in verschiedene Systemarchitekturen und eignet sich besonders für Anwendungen, bei denen mehrere Signale logisch verknüpft werden müssen.

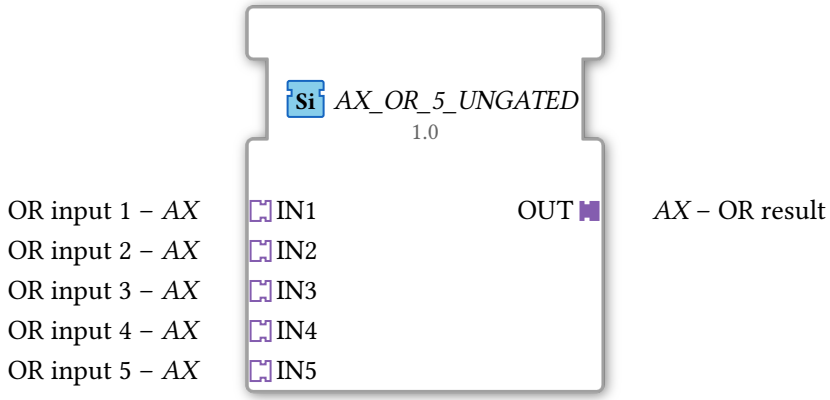
AX_OR_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AX_OR_5. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_OR_5_UNGATED Funktionsblock ist ein generischer Baustein zur Berechnung der booleschen ODER-Verknüpfung mit fünf Eingängen. Er dient zur logischen Verarbeitung von Signalen in Automatisierungssystemen und ermöglicht die Kombination mehrerer Eingangssignale zu einem Ausgangssignal.



Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden.

Adapter
Plug-Adapter:

- **OUT**: Unidirektionaler Adapter für das ODER-Ergebnis

Socket-Adapter:

- **IN1**: Unidirektionaler Adapter für ODER-Eingang 1
- **IN2**: Unidirektionaler Adapter für ODER-Eingang 2
- **IN3**: Unidirektionaler Adapter für ODER-Eingang 3
- **IN4**: Unidirektionaler Adapter für ODER-Eingang 4
- **IN5**: Unidirektionaler Adapter für ODER-Eingang 5

Funktionsweise
Der Funktionsblock berechnet die logische ODER-Verknüpfung der fünf Eingangssignale. Das Ausgangssignal ist TRUE, wenn mindestens einer der fünf Eingänge TRUE ist. Sind alle Eingänge FALSE, wird das Ausgangssignal ebenfalls FALSE. Die Verarbeitung erfolgt über Adapter-Schnittstellen, wobei alle Ein- und Ausgänge unidirektional konfiguriert sind.

Technische Besonderheiten

- Generischer Funktionsblock mit fünf Eingängen
- Verwendung von unidirektionalen Adaptern für alle Schnittstellen
- Implementiert als ODER-Verknüpfung mit fünf Operanden
- Keine Ereignissteuerung, kontinuierliche Signalverarbeitung

Zustandsübersicht
Der Baustein besitzt keinen internen Zustand und arbeitet stateless. Die Ausgabe wird kontinuierlich basierend auf den aktuellen Eingangswerten berechnet.

Anwendungsszenarien

- Sicherheitskreise mit mehreren Not-Aus-Tastern
- Überwachungssysteme mit mehreren Sensoren

- Steuerungslogik mit parallelen Bedingungen
- Alarmierungssysteme mit mehreren Auslösern

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-ODER-Bausteinen mit weniger Eingängen bietet AX_OR_5_UNGATED die Möglichkeit, fünf Signale gleichzeitig zu verarbeiten, was die Verdrahtung vereinfacht und Platz spart. Gegenüber kaskadierten ODER-Verknüpfungen bietet dieser Baustein eine integrierte Lösung.

Vergleich mit OR_5

- `AX_OR_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

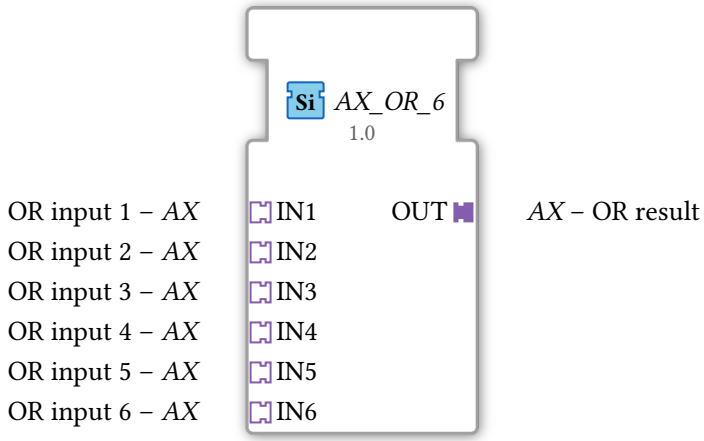
Der AX_OR_5_UNGATED Funktionsblock stellt eine effiziente Lösung für ODER-Verknüpfungen mit fünf Eingängen dar. Durch die Verwendung von Adaptern ermöglicht er eine flexible Integration in verschiedene Systemarchitekturen und eignet sich besonders für Anwendungen, bei denen mehrere Signale logisch verknüpft werden müssen.

AX_OR_6

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_OR_6 ist ein generischer Funktionsblock zur Berechnung einer logischen ODER-Verknüpfung mit sechs Eingängen. Der Baustein dient zur Verarbeitung von booleschen Signalen in Automatisierungssystemen und gibt das Ergebnis der ODER-Operation über einen Adapterausgang aus.



Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereigniseingänge vorhanden

Ereignis-Ausgänge

Keine Ereignisausgänge vorhanden

Daten-Eingänge

Keine Dateneingänge vorhanden

Daten-Ausgänge

Keine Datenausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1** - ODER-Eingang 1 (Typ: adapter::types::unidirectional::AX)
- **IN2** - ODER-Eingang 2 (Typ: adapter::types::unidirectional::AX)
- **IN3** - ODER-Eingang 3 (Typ: adapter::types::unidirectional::AX)
- **IN4** - ODER-Eingang 4 (Typ: adapter::types::unidirectional::AX)
- **IN5** - ODER-Eingang 5 (Typ: adapter::types::unidirectional::AX)
- **IN6** - ODER-Eingang 6 (Typ: adapter::types::unidirectional::AX)

Ausgangsadapter:

- **OUT** - ODER-Ergebnis (Typ: adapter::types::unidirectional::AX)

Funktionsweise

Der Funktionsblock berechnet kontinuierlich die logische ODER-Verknüpfung aller sechs Eingangssignale. Das Ausgangssignal ist TRUE (wahr), wenn mindestens einer der sechs Eingänge TRUE ist. Nur wenn alle sechs Eingänge FALSE (falsch) sind, wird der Ausgang ebenfalls FALSE.

Technische Besonderheiten

- Generischer Funktionsblock mit der Klasse ‚GEN_AX_OR'
- Verwendet unidirektionale AX-Adapter für die Signalübertragung
- Sechs unabhängige Eingänge für flexible Anwendungen
- Echtzeitfähige Verarbeitung ohne Ereignissteuerung

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand und arbeitet stateless. Die Ausgabe wird kontinuierlich basierend auf den aktuellen Eingangswerten berechnet.

Anwendungsszenarien

- Sicherheitskreise mit mehreren Not-Aus-Tastern
- Überwachungssysteme mit mehreren Sensoren
- Steuerungslogik mit alternativen Aktivierungsbedingungen
- Verknüpfung mehrerer Zustandsmeldungen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren ODER-Bausteinen mit weniger Eingängen bietet AX_OR_6 eine höhere Flexibilität durch sechs separate Eingänge. Die ausschließliche Verwendung von Adaptern statt konventioneller Daten-Ein-/Ausgänge ermöglicht eine modulare Systemarchitektur.

Vergleich mit OR_6

- `AX_OR_6_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der AX_OR_6 ist ein spezialisierter Logikbaustein für Anwendungen, die eine ODER-Verknüpfung mit mehreren Eingängen erfordern. Durch die Adapter-basierte Schnittstelle eignet er sich besonders für modulare Systemdesigns und ermöglicht eine klare Trennung der Signalwege.

AX_OR_6_UNGATED

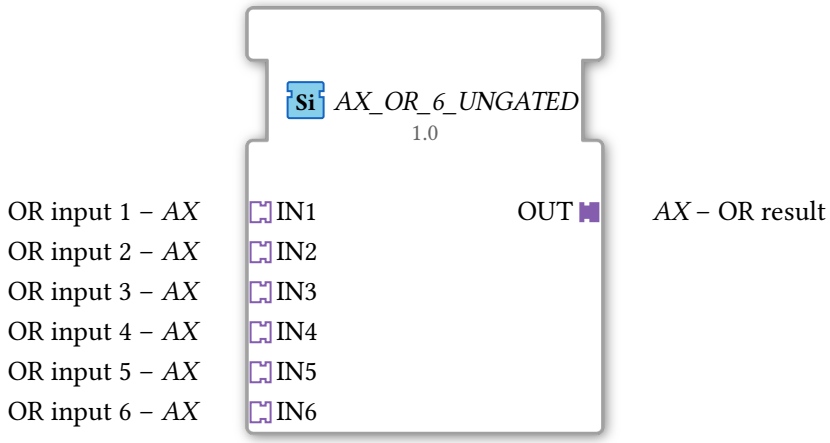
> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AX_OR_6`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz

unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_OR_6_UNGATED ist ein generischer Funktionsblock zur Berechnung einer logischen ODER-Verknüpfung mit sechs Eingängen. Der Baustein dient zur Verarbeitung von booleschen Signalen in Automatisierungssystemen und gibt das Ergebnis der ODER-Operation über einen Adapterausgang aus.



Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereigniseingänge vorhanden

Ereignis-Ausgänge

Keine Ereignisausgänge vorhanden

Daten-Eingänge

Keine Dateneingänge vorhanden

Daten-Ausgänge

Keine Datenausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1** - ODER-Eingang 1 (Typ: adapter::types::unidirectional::AX)
- **IN2** - ODER-Eingang 2 (Typ: adapter::types::unidirectional::AX)
- **IN3** - ODER-Eingang 3 (Typ: adapter::types::unidirectional::AX)
- **IN4** - ODER-Eingang 4 (Typ: adapter::types::unidirectional::AX)
- **IN5** - ODER-Eingang 5 (Typ: adapter::types::unidirectional::AX)
- **IN6** - ODER-Eingang 6 (Typ: adapter::types::unidirectional::AX)

Ausgangsadapter:

- **OUT** - ODER-Ergebnis (Typ: adapter::types::unidirectional::AX)

Funktionsweise

Der Funktionsblock berechnet kontinuierlich die logische ODER-Verknüpfung aller sechs Eingangssignale. Das Ausgangssignal ist TRUE (wahr), wenn mindestens einer der sechs Eingänge TRUE ist. Nur wenn alle sechs Eingänge FALSE (falsch) sind, wird der Ausgang ebenfalls FALSE.

Technische Besonderheiten

- Generischer Funktionsblock mit der Klasse ‚GEN_AX_OR‘
- Verwendet unidirektionale AX-Adapter für die Signalübertragung
- Sechs unabhängige Eingänge für flexible Anwendungen
- Echtzeitfähige Verarbeitung ohne Ereignissteuerung

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand und arbeitet stateless. Die Ausgabe wird kontinuierlich basierend auf den aktuellen Eingangswerten berechnet.

Anwendungsszenarien

- Sicherheitskreise mit mehreren Not-Aus-Tastern
- Überwachungssysteme mit mehreren Sensoren
- Steuerungslogik mit alternativen Aktivierungsbedingungen
- Verknüpfung mehrerer Zustandsmeldungen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren ODER-Bausteinen mit weniger Eingängen bietet AX_OR_6_UNGATED eine höhere Flexibilität durch sechs separate Eingänge. Die ausschließliche Verwendung von Adaptern statt konventioneller Daten-Ein-/Ausgänge ermöglicht eine modulare Systemarchitektur.

Vergleich mit OR_6

- **`AX_OR_6`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

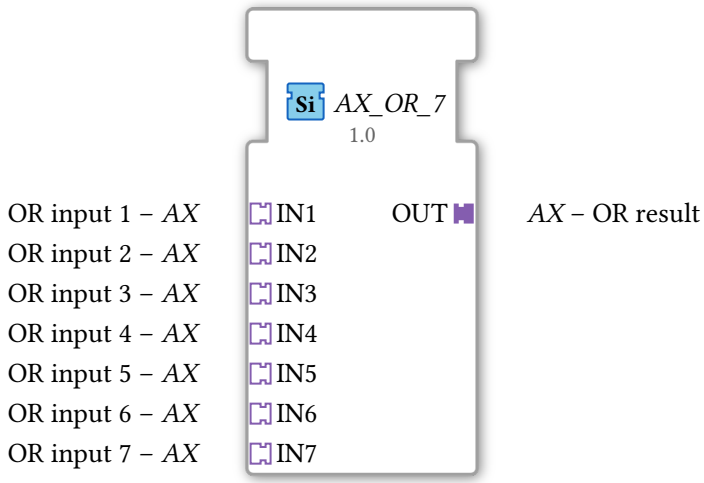
Der AX_OR_6_UNGATED ist ein spezialisierter Logikbaustein für Anwendungen, die eine ODER-Verknüpfung mit mehreren Eingängen erfordern. Durch die Adapter-basierte Schnittstelle eignet er sich besonders für modulare Systemdesigns und ermöglicht eine klare Trennung der Signalwege.

AX_OR_7

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_OR_7 Funktionsblock ist ein generischer Baustein zur Berechnung der booleschen ODER-Verknüpfung. Er verfügt über sieben Eingänge und berechnet das logische ODER aller Eingangssignale.



Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden.

Adapter
Eingangsadapter:

- **IN1** - ODER Eingang 1
- **IN2** - ODER Eingang 2
- **IN3** - ODER Eingang 3
- **IN4** - ODER Eingang 4
- **IN5** - ODER Eingang 5
- **IN6** - ODER Eingang 6
- **IN7** - ODER Eingang 7

Ausgangsadapter:

- **OUT** - ODER Ergebnis

Alle Adapter verwenden den Typ `adapter::types::unidirectional::AX`.

Funktionsweise
Der Funktionsblock berechnet kontinuierlich die boolesche ODER-Verknüpfung aller sieben Eingangssignale. Das Ergebnis wird über den Ausgangsadapter OUT ausgegeben. Das Ausgangssignal ist TRUE, wenn mindestens einer der sieben Eingänge TRUE ist.

Technische Besonderheiten
- Generischer Funktionsblock mit der Klasse ‚GEN_AX_OR'
- Verwendet unidirektionale AX-Adapter für alle Schnittstellen
- Keine Ereignissteuerung - arbeitet kontinuierlich
- Sieben unabhängige Eingänge für maximale Flexibilität

Zustandsübersicht
Da es sich um einen rein kombinatorischen Baustein ohne Ereignissteuerung handelt, besitzt der AX_OR_7 keine Zustandsmaschine. Die Ausgabe wird direkt aus den aktuellen Eingangswerten berechnet.

Anwendungsszenarien
- Sicherheitskreise mit mehreren Not-Aus-Tastern
- Mehrfach-Bedienstationen in Maschinensteuerungen
- Überwachungssysteme mit mehreren Sensoren
- Verknüpfung mehrerer Freigabesignale

⚖️ Vergleich mit ähnlichen Bausteinen
Im Vergleich zu Standard-ODER-Bausteinen bietet AX_OR_7 die Besonderheit von sieben Eingängen, was die Verkabelung vereinfacht, da weniger Bausteine in Reihe geschaltet werden müssen. Die Verwendung von Adaptern statt direkter Datenanschlüsse ermöglicht eine flexiblere Integration in verschiedene Systemarchitekturen.

Vergleich mit OR_7

- `AX_OR_7_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der AX_OR_7 ist ein spezialisierter ODER-Baustein für Anwendungen mit vielen Eingangssignalen. Durch die sieben Eingänge und die Adapter-basierte Schnittstelle eignet er sich besonders für komplexe Steuerungsaufgaben, bei denen mehrere Signale logisch verknüpft werden müssen.

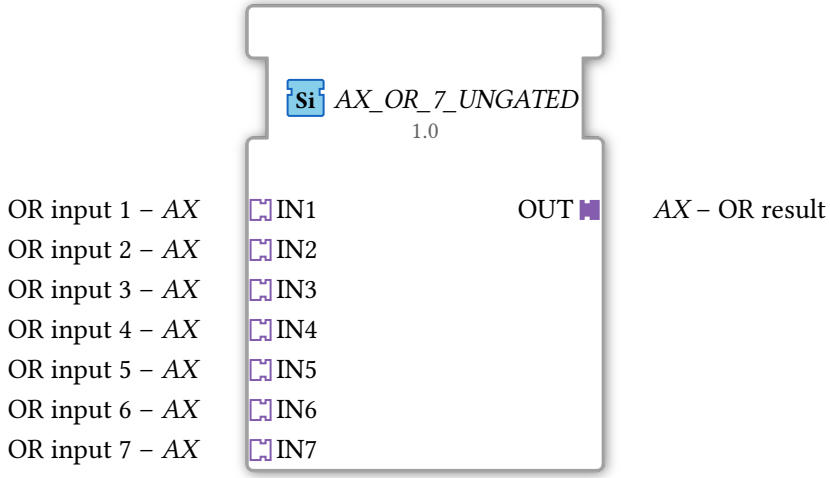
AX_OR_7_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AX_OR_7`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_OR_7_UNGATED Funktionsblock ist ein generischer Baustein zur Berechnung der booleschen ODER-Verknüpfung. Er verfügt über sieben Eingänge und berechnet das logische ODER aller Eingangssignale.



Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Eingangsadapter:

- **IN1** - ODER Eingang 1
- **IN2** - ODER Eingang 2
- **IN3** - ODER Eingang 3
- **IN4** - ODER Eingang 4
- **IN5** - ODER Eingang 5
- **IN6** - ODER Eingang 6
- **IN7** - ODER Eingang 7

Ausgangsadapter:

- **OUT** - ODER Ergebnis

Alle Adapter verwenden den Typ `adapter::types::unidirectional::AX`.

Funktionsweise

Der Funktionsblock berechnet kontinuierlich die boolesche ODER-Verknüpfung aller sieben Eingangssignale. Das Ergebnis wird über den Ausgangsadapter OUT ausgegeben. Das Ausgangssignal ist TRUE, wenn mindestens einer der sieben Eingänge TRUE ist.

Technische Besonderheiten

- Generischer Funktionsblock mit der Klasse ‚GEN_AX_OR'
- Verwendet unidirektionale AX-Adapter für alle Schnittstellen
- Keine Ereignissteuerung - arbeitet kontinuierlich
- Sieben unabhängige Eingänge für maximale Flexibilität

Zustandsübersicht

Da es sich um einen rein kombinatorischen Baustein ohne Ereignissteuerung handelt, besitzt der AX_OR_7_UNGATED keine Zustandsmaschine. Die Ausgabe wird direkt aus den aktuellen Eingangswerten berechnet.

Anwendungsszenarien

- Sicherheitskreise mit mehreren Not-Aus-Tastern
- Mehrfach-Bedienstationen in Maschinensteuerungen
- Überwachungssysteme mit mehreren Sensoren
- Verknüpfung mehrerer Freigabesignale

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-ODER-Bausteinen bietet AX_OR_7_UNGATED die Besonderheit von sieben Eingängen, was die Verkabelung vereinfacht, da weniger Bausteine in Reihe geschaltet werden müssen. Die Verwendung von Adaptern statt direkter Datenanschlüsse ermöglicht eine flexiblere Integration in verschiedene Systemarchitekturen.

Vergleich mit OR_7

- **`AX_OR_7`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

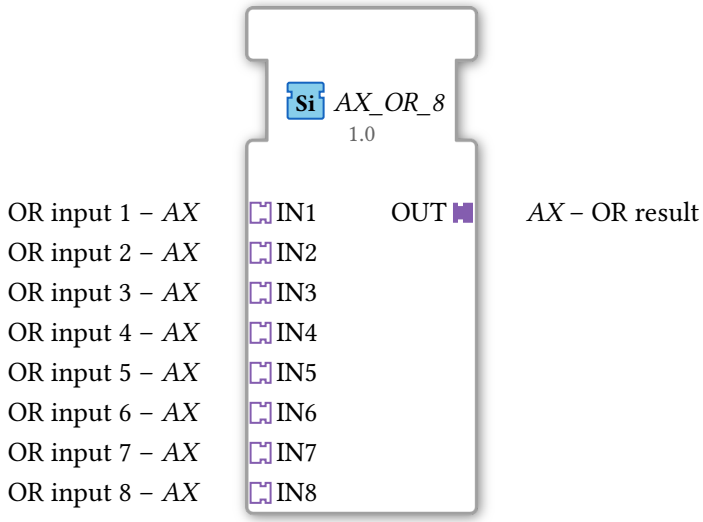
Der AX_OR_7_UNGATED ist ein spezialisierter ODER-Baustein für Anwendungen mit vielen Eingangssignalen. Durch die sieben Eingänge und die Adapter-basierte Schnittstelle eignet er sich besonders für komplexe Steuerungsaufgaben, bei denen mehrere Signale logisch verknüpft werden müssen.

AX_OR_8

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_OR_8 Funktionsblock ist ein generischer Baustein zur Berechnung einer logischen ODER-Verknüpfung mit acht Eingängen. Er dient zur Verarbeitung von booleschen Signalen in Steuerungsanwendungen und ermöglicht die Kombination mehrerer Eingangssignale zu einem gemeinsamen Ausgangssignal.



Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN1** - OR input 1 (unidirektionaler AX-Adapter)
- **IN2** - OR input 2 (unidirektionaler AX-Adapter)
- **IN3** - OR input 3 (unidirektionaler AX-Adapter)
- **IN4** - OR input 4 (unidirektionaler AX-Adapter)
- **IN5** - OR input 5 (unidirektionaler AX-Adapter)
- **IN6** - OR input 6 (unidirektionaler AX-Adapter)
- **IN7** - OR input 7 (unidirektionaler AX-Adapter)
- **IN8** - OR input 8 (unidirektionaler AX-Adapter)

Ausgangsadapter:

- **OUT** - OR result (unidirektionaler AX-Adapter)

Funktionsweise

Der AX_OR_8 Block führt eine logische ODER-Verknüpfung über alle acht Eingänge durch. Das Ausgangssignal wird auf TRUE gesetzt, wenn mindestens einer der acht Eingänge TRUE ist. Nur wenn alle Eingänge FALSE sind, wird der Ausgang ebenfalls FALSE.

Die Funktionsweise entspricht der booleschen Gleichung: OUT = IN1 ∨ IN2 ∨ IN3 ∨ IN4 ∨ IN5 ∨ IN6 ∨ IN7 ∨ IN8

Technische Besonderheiten

- Generischer Funktionsblock mit der Klassennamen-Kennung ‚GEN_AX_OR‘
- Verwendet unidirektionale AX-Adapter für alle Schnittstellen
- Keine internen Zustände oder Speicherfunktionen
- Sofortige Verarbeitung ohne Verzögerung
- Teil des Pakets „adapter::booleanOperators“

Zustandsübersicht

Der Funktionsblock ist zustandslos und führt bei jeder Änderung an den Eingängen sofort die ODER-Berechnung durch. Es werden keine internen Zustände gespeichert.

Anwendungsszenarien

- Sicherheitskreise mit mehreren Not-Aus-Tastern
- Überwachungssysteme mit mehreren Sensoren
- Steuerungslogik mit parallelen Bedingungen
- Alarmierungssysteme mit mehreren Auslösern
- Verknüpfung mehrerer Freigabesignale

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen ODER-Bausteinen mit weniger Eingängen bietet AX_OR_8 die Möglichkeit, bis zu acht Signale direkt zu verknüpfen, ohne zusätzliche Verkettung mehrerer Bausteine. Die Verwendung von Adaptern statt direkter Datenanschlüsse ermöglicht eine flexiblere Integration in verschiedene Systemarchitekturen.

Vergleich mit OR_8

- `AX_OR_8_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der AX_OR_8 Funktionsblock stellt eine effiziente Lösung für komplexe ODER-Verknüpfungen mit bis zu acht Eingängen dar. Durch die Verwendung von Standard-Adaptern und die generische Implementierung eignet er sich besonders für wiederverwendbare Steuerungslogik in industriellen Automatisierungsanwendungen.

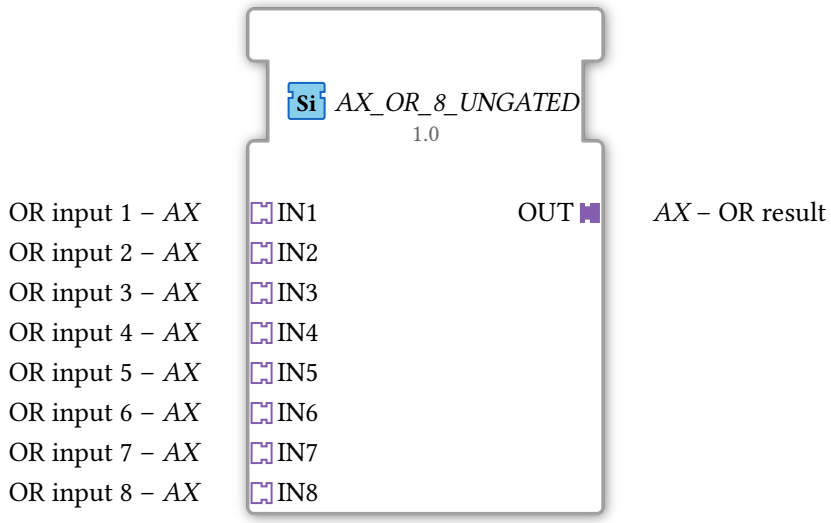
AX_OR_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AX_OR_8`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_OR_8_UNGATED Funktionsblock ist ein generischer Baustein zur Berechnung einer logischen ODER-Verknüpfung mit acht Eingängen. Er dient zur Verarbeitung von booleschen Signalen in Steuerungsanwendungen und ermöglicht die Kombination mehrerer Eingangssignale zu einem gemeinsamen Ausgangssignal.



Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN1** - OR input 1 (unidirektionaler AX-Adapter)
- **IN2** - OR input 2 (unidirektionaler AX-Adapter)
- **IN3** - OR input 3 (unidirektionaler AX-Adapter)
- **IN4** - OR input 4 (unidirektionaler AX-Adapter)
- **IN5** - OR input 5 (unidirektionaler AX-Adapter)
- **IN6** - OR input 6 (unidirektionaler AX-Adapter)
- **IN7** - OR input 7 (unidirektionaler AX-Adapter)
- **IN8** - OR input 8 (unidirektionaler AX-Adapter)

Ausgangsadapter:

- **OUT** - OR result (unidirektionaler AX-Adapter)

Funktionsweise
Der AX_OR_8_UNGATED Block führt eine logische ODER-Verknüpfung über alle acht Eingänge durch. Das Ausgangssignal wird auf TRUE gesetzt, wenn mindestens einer der acht Eingänge TRUE ist. Nur wenn alle Eingänge FALSE sind, wird der Ausgang ebenfalls FALSE.

Die Funktionsweise entspricht der booleschen Gleichung: OUT = IN1 ∨ IN2 ∨ IN3 ∨ IN4 ∨ IN5 ∨ IN6 ∨ IN7 ∨ IN8

Technische Besonderheiten

- Generischer Funktionsblock mit der Klassennamen-Kennung ‚GEN_AX_OR‘
- Verwendet unidirektionale AX-Adapter für alle Schnittstellen
- Keine internen Zustände oder Speicherfunktionen
- Sofortige Verarbeitung ohne Verzögerung

- Teil des Pakets „adapter::booleanOperators“

Zustandsübersicht

Der Funktionsblock ist zustandslos und führt bei jeder Änderung an den Eingängen sofort die ODER-Berechnung durch. Es werden keine internen Zustände gespeichert.

Anwendungsszenarien

- Sicherheitskreise mit mehreren Not-Aus-Tastern
- Überwachungssysteme mit mehreren Sensoren
- Steuerungslogik mit parallelen Bedingungen
- Alarmierungssysteme mit mehreren Auslösern
- Verknüpfung mehrerer Freigabesignale

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen ODER-Bausteinen mit weniger Eingängen bietet AX_OR_8_UNGATED die Möglichkeit, bis zu acht Signale direkt zu verknüpfen, ohne zusätzliche Verkettung mehrerer Bausteine. Die Verwendung von Adaptern statt direkter Datenanschlüsse ermöglicht eine flexiblere Integration in verschiedene Systemarchitekturen.

Vergleich mit OR_8

- `AX_OR_8`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

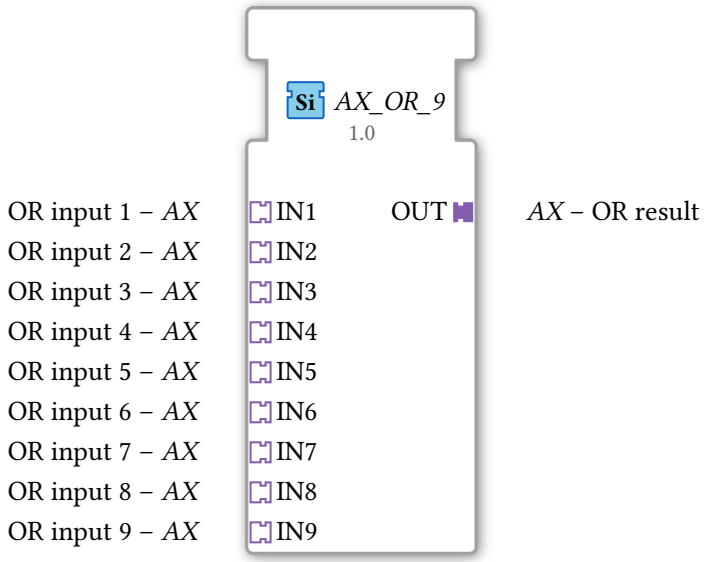
Der AX_OR_8_UNGATED Funktionsblock stellt eine effiziente Lösung für komplexe ODER-Verknüpfungen mit bis zu acht Eingängen dar. Durch die Verwendung von Standard-Adaptern und die generische Implementierung eignet er sich besonders für wiederverwendbare Steuerungslogik in industriellen Automatisierungsanwendungen.

AX_OR_9

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_OR_9 ist ein generischer Funktionsblock zur Berechnung der booleschen ODER-Verknüpfung mit neun Eingängen. Der Baustein dient zur logischen Verarbeitung von Signalen in Automatisierungssystemen und gibt das Ergebnis der ODER-Operation aller Eingangssignale aus.



Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Sockets (Eingänge):

- **IN1** - ODER Eingang 1
- **IN2** - ODER Eingang 2
- **IN3** - ODER Eingang 3
- **IN4** - ODER Eingang 4
- **IN5** - ODER Eingang 5
- **IN6** - ODER Eingang 6
- **IN7** - ODER Eingang 7
- **IN8** - ODER Eingang 8
- **IN9** - ODER Eingang 9

Plugs (Ausgang):

- **OUT** - ODER Ergebnis

Funktionsweise
Der Funktionsblock berechnet kontinuierlich die boolesche ODER-Verknüpfung aller neun Eingangssignale. Das Ausgangssignal ist TRUE (1), wenn mindestens einer der neun Eingänge TRUE (1) ist. Nur wenn alle Eingänge FALSE (0) sind, wird der Ausgang ebenfalls FALSE (0).

Technische Besonderheiten

- Generischer Funktionsblock mit fest konfigurierter Anzahl von 9 Eingängen
- Verwendet unidirektionale AX-Adapter für Ein- und Ausgänge
- Keine Ereignissteuerung - arbeitet kontinuierlich
- Implementiert als generischer Baustein mit spezifischem Klassennamen

Zustandsübersicht
Der Funktionsblock besitzt keinen internen Zustand und arbeitet stateless. Die Ausgabe wird ausschließlich auf Basis der aktuellen Eingangswerte berechnet.

Anwendungsszenarien

- Sicherheitskreise mit mehreren Not-Aus-Tastern
- Überwachungssysteme mit mehreren Sensoren
- Steuerungslogik mit redundanten Eingangssignalen
- Alarmierungssysteme mit mehreren Auslösern

⚖️ Vergleich mit ähnlichen Bausteinen
Im Vergleich zu Standard-ODER-Bausteinen mit weniger Eingängen bietet AX_OR_9 die Möglichkeit, bis zu neun Signale gleichzeitig zu verarbeiten, was die Verdrahtung komplexerer Logiken vereinfacht. Gegenüber konfigurierbaren ODER-Bausteinen hat dieser FB eine feste Anzahl an Eingängen, was bei bekannten Anforderungen die Konfiguration vereinfacht.
Vergleich mit OR_9

- `AX_OR_9_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der AX_OR_9 ist ein spezialisierter ODER-Baustein für Anwendungen, die genau neun Eingangssignale logisch verknüpfen müssen. Durch die feste Eingangsanzahl und die einfache Funktionsweise eignet er sich besonders für klar definierte Steuerungsaufgaben mit redundanten Eingangssignalen.

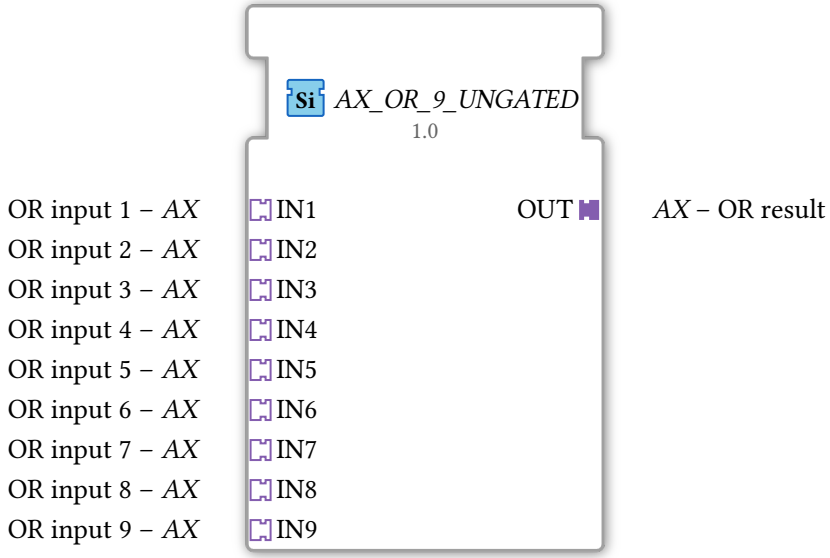
AX_OR_9_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AX_OR_9`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_OR_9_UNGATED ist ein generischer Funktionsblock zur Berechnung der booleschen ODER-Verknüpfung mit neun Eingängen. Der Baustein dient zur logischen Verarbeitung von Signalen in Automatisierungssystemen und gibt das Ergebnis der ODER-Operation aller Eingangssignale aus.



Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Sockets (Eingänge):

- **IN1** - ODER Eingang 1
- **IN2** - ODER Eingang 2
- **IN3** - ODER Eingang 3
- **IN4** - ODER Eingang 4
- **IN5** - ODER Eingang 5
- **IN6** - ODER Eingang 6
- **IN7** - ODER Eingang 7
- **IN8** - ODER Eingang 8
- **IN9** - ODER Eingang 9

Plugs (Ausgang):

- **OUT** - ODER Ergebnis

Funktionsweise

Der Funktionsblock berechnet kontinuierlich die boolesche ODER-Verknüpfung aller neun Eingangssignale. Das Ausgangssignal ist TRUE (1), wenn mindestens einer der neun Eingänge TRUE (1) ist. Nur wenn alle Eingänge FALSE (0) sind, wird der Ausgang ebenfalls FALSE (0).

Technische Besonderheiten

- Generischer Funktionsblock mit fest konfigurierter Anzahl von 9 Eingängen
- Verwendet unidirektionale AX-Adapter für Ein- und Ausgänge
- Keine Ereignissteuerung - arbeitet kontinuierlich
- Implementiert als generischer Baustein mit spezifischem Klassennamen

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand und arbeitet stateless. Die Ausgabe wird ausschließlich auf Basis der aktuellen Eingangswerte berechnet.

Anwendungsszenarien

- Sicherheitskreise mit mehreren Not-Aus-Tastern
- Überwachungssysteme mit mehreren Sensoren
- Steuerungslogik mit redundanten Eingangssignalen
- Alarmierungssysteme mit mehreren Auslösern

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-ODER-Bausteinen mit weniger Eingängen bietet AX_OR_9_UNGATED die Möglichkeit, bis zu neun Signale gleichzeitig zu verarbeiten, was die Verdrahtung komplexerer Logiken vereinfacht. Gegenüber konfigurierbaren ODER-Bausteinen hat dieser FB eine feste Anzahl an Eingängen, was bei bekannten Anforderungen die Konfiguration vereinfacht.

Vergleich mit OR_9

- `AX_OR_9`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

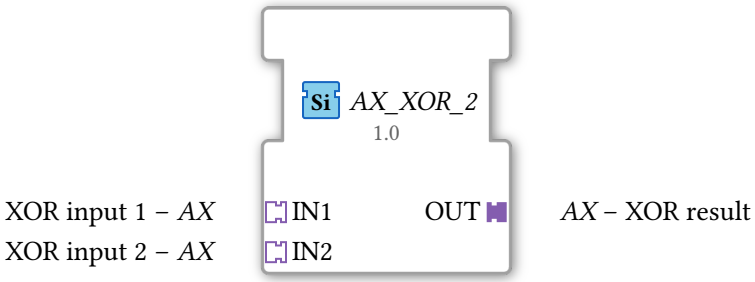
Der AX_OR_9_UNGATED ist ein spezialisierter ODER-Baustein für Anwendungen, die genau neun Eingangssignale logisch verknüpfen müssen. Durch die feste Eingangsanzahl und die einfache Funktionsweise eignet er sich besonders für klar definierte Steuerungsaufgaben mit redundanten Eingangssignalen.

AX_XOR_2

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_XOR_2 Funktionsblock ist ein generischer Baustein zur Berechnung der booleschen XOR-Operation (exklusives ODER). Er verarbeitet zwei Eingangssignale und gibt das logische XOR-Ergebnis aus.



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereigniseingänge vorhanden - die Verarbeitung erfolgt über Adapter-Schnittstellen.

Ereignis-Ausgänge

Keine direkten Ereignisausgänge vorhanden - die Ausgabe erfolgt über Adapter-Schnittstellen.

Daten-Eingänge

Keine direkten Dateneingänge vorhanden - die Eingabe erfolgt über Adapter-Schnittstellen.

Daten-Ausgänge

Keine direkten Datenausgänge vorhanden - die Ausgabe erfolgt über Adapter-Schnittstellen.

Adapter

- **IN1** (Socket): XOR-Eingang 1 - Empfängt das erste Eingangssignal
- **IN2** (Socket): XOR-Eingang 2 - Empfängt das zweite Eingangssignal
- **OUT** (Plug): XOR-Ergebnis - Gibt das berechnete XOR-Ergebnis aus

Alle Adapter verwenden den Typ `adapter::types::unidirectional::AX` für unidirektionale Kommunikation.

Funktionsweise

Der Funktionsblock berechnet kontinuierlich die XOR-Operation zwischen den beiden Eingangssignalen IN1 und IN2. Die XOR-Funktion gibt genau dann ein wahr-Signal (TRUE) aus, wenn genau einer der beiden Eingänge wahr ist (TRUE). Sind beide Eingänge gleich (beide TRUE oder beide FALSE), wird ein falsch-Signal (FALSE) ausgegeben.

Technische Besonderheiten

- Generischer Funktionsblock mit der Klasse `GEN_AX_XOR`
- Verwendet unidirektionale Adapter für Ein- und Ausgaben
- Implementiert gemäß IEC 61499-2 Standard
- Teil des Pakets `adapter::booleanOperators`

Zustandsübersicht

Der Baustein besitzt einen kontinuierlichen Betriebszustand und berechnet permanent das XOR-Ergebnis basierend auf den aktuellen Eingangswerten.

Anwendungsszenarien

- Logische Schaltungen in Steuerungssystemen
- Vergleich von zwei binären Signalen
- Erkennung von Signalunterschieden

- Sicherheitsrelevante Schaltungen
- Digitale Signalverarbeitung

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen logischen Operatoren wie AND oder OR bietet XOR die Besonderheit, dass nur bei unterschiedlichen Eingangswerten ein TRUE-Signal erzeugt wird. Dies macht ihn besonders geeignet für Vergleichs- und Differenzerkennungsaufgaben.

Vergleich mit XOR_2

- `AX_XOR_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der AX_XOR_2 Funktionsblock stellt eine effiziente und zuverlässige Lösung für XOR-Operationen in 4diac-basierten Steuerungssystemen dar. Durch die Verwendung von Adapter-Schnittstellen ermöglicht er eine flexible Integration in verschiedene Systemarchitekturen und eignet sich besonders für Anwendungen, bei denen Signalunterschiede erkannt werden müssen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

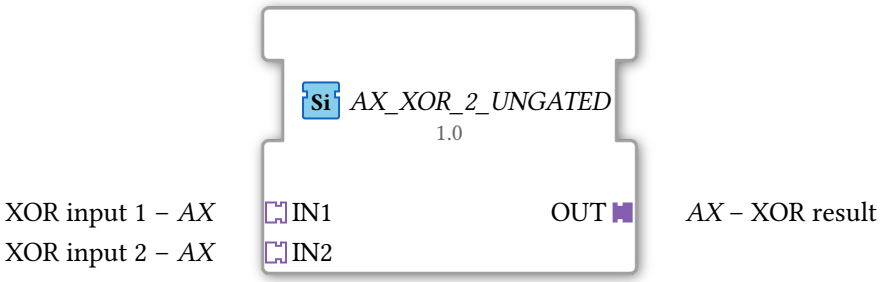
AX_XOR_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AX_XOR_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_XOR_2_UNGATED Funktionsblock ist ein generischer Baustein zur Berechnung der booleschen XOR-Operation (exklusives ODER). Er verarbeitet zwei Eingangssignale und gibt das logische XOR-Ergebnis aus.



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereigniseingänge vorhanden - die Verarbeitung erfolgt über Adapter-Schnittstellen.

Ereignis-Ausgänge

Keine direkten Ereignisausgänge vorhanden - die Ausgabe erfolgt über Adapter-Schnittstellen.

Daten-Eingänge

Keine direkten Dateneingänge vorhanden - die Eingabe erfolgt über Adapter-Schnittstellen.

Daten-Ausgänge

Keine direkten Datenausgänge vorhanden - die Ausgabe erfolgt über Adapter-Schnittstellen.

Adapter

- **IN1** (Socket): XOR-Eingang 1 - Empfängt das erste Eingangssignal
- **IN2** (Socket): XOR-Eingang 2 - Empfängt das zweite Eingangssignal
- **OUT** (Plug): XOR-Ergebnis - Gibt das berechnete XOR-Ergebnis aus

Alle Adapter verwenden den Typ `adapter::types::unidirectional::AX` für unidirektionale Kommunikation.

Funktionsweise

Der Funktionsblock berechnet kontinuierlich die XOR-Operation zwischen den beiden Eingangssignalen IN1 und IN2. Die XOR-Funktion gibt genau dann ein wahr-Signal (TRUE) aus, wenn genau einer der beiden Eingänge wahr ist (TRUE). Sind beide Eingänge gleich (beide TRUE oder beide FALSE), wird ein falsch-Signal (FALSE) ausgegeben.

Technische Besonderheiten

- Generischer Funktionsblock mit der Klasse `GEN_AX_XOR`
- Verwendet unidirektionale Adapter für Ein- und Ausgaben
- Implementiert gemäß IEC 61499-2 Standard
- Teil des Pakets `adapter::booleanOperators`

Zustandsübersicht

Der Baustein besitzt einen kontinuierlichen Betriebszustand und berechnet permanent das XOR-Ergebnis basierend auf den aktuellen Eingangswerten.

Anwendungsszenarien

- Logische Schaltungen in Steuerungssystemen
- Vergleich von zwei binären Signalen
- Erkennung von Signalunterschieden
- Sicherheitsrelevante Schaltungen
- Digitale Signalverarbeitung

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen logischen Operatoren wie AND oder OR bietet XOR die Besonderheit, dass nur bei unterschiedlichen Eingangswerten ein TRUE-Signal erzeugt wird. Dies macht ihn besonders geeignet für Vergleichs- und Differenzerkennungsaufgaben.

Vergleich mit XOR_2

- **`AX_XOR_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der AX_XOR_2_UNGATED Funktionsblock stellt eine effiziente und zuverlässige Lösung für XOR-Operationen in 4diac-basierten Steuerungssystemen dar. Durch die Verwendung von Adapter-Schnittstellen ermöglicht er eine flexible Integration in verschiedene Systemarchitekturen und eignet sich besonders für Anwendungen, bei denen Signalunterschiede erkannt werden müssen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

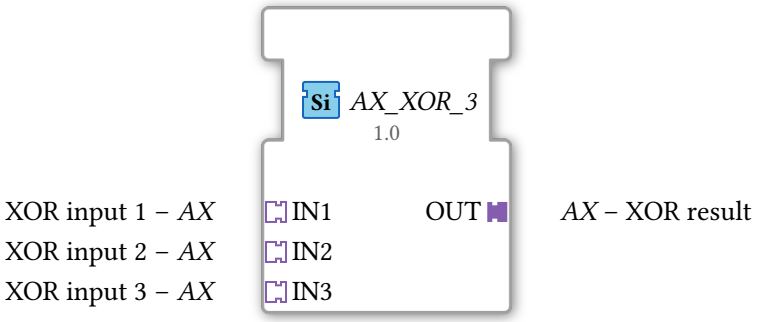
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AX_XOR_3

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_XOR_3 Funktionsblock ist ein generischer Baustein zur Berechnung der booleschen XOR-Verknüpfung mit drei Eingängen. Er implementiert die exklusive ODER-Operation für drei binäre Eingangssignale und gibt das entsprechende Ergebnis aus.



Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Sockets (Eingänge):

- **IN1**: XOR-Eingang 1 (Adapter-Typ: unidirectional::AX)
- **IN2**: XOR-Eingang 2 (Adapter-Typ: unidirectional::AX)
- **IN3**: XOR-Eingang 3 (Adapter-Typ: unidirectional::AX)

Plugs (Ausgänge):

- **OUT**: XOR-Ergebnis (Adapter-Typ: unidirectional::AX)

Funktionsweise

Der Funktionsblock berechnet die XOR-Verknüpfung für drei Eingangssignale. Die XOR-Operation gibt genau dann ein „wahr“-Signal (true) aus, wenn eine ungerade Anzahl der Eingänge „wahr“ ist. Bei drei Eingängen bedeutet dies:

- Ergebnis ist true, wenn genau einer oder alle drei Eingänge true sind
- Ergebnis ist false, wenn keine oder genau zwei Eingänge true sind

Technische Besonderheiten

- Generischer Funktionsblock mit der Klassennamen-Kennung ‚GEN_AX_XOR‘
- Verwendet unidirektionale Adapter für die Kommunikation
- Implementiert nach IEC 61499-2 Standard
- Keine internen Zustände oder Speicherung erforderlich

Zustandsübersicht

Der Baustein besitzt keine internen Zustände und arbeitet statisch. Die Ausgabe wird ausschließlich auf Basis der aktuellen Eingangswerte berechnet.

Anwendungsszenarien

- Schaltnetze mit drei Eingängen
- Paritätsprüfungen
- Sicherheitskritische Steuerungen, wo ungerade Aktivierung erforderlich ist
- Logische Verknüpfungen in Automatisierungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-XOR-Bausteinen mit zwei Eingängen bietet AX_XOR_3 die Erweiterung auf drei Eingänge. Während ein 2-Eingang-XOR nur bei ungleichen Eingängen true liefert, berücksichtigt der 3-Eingang-XOR die Parität aller drei Signale.

Vergleich mit XOR_3

- **`AX_XOR_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

🛠️ Zugehörige Übungen

- Uebung_002a7_AX

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der AX_XOR_3 Funktionsblock bietet eine zuverlässige und effiziente Lösung für dreifache XOR-Verknüpfungen in IEC 61499-basierten Steuerungssystemen. Durch die Verwendung von Adaptern ermöglicht er eine flexible Integration in verschiedene Systemarchitekturen und ist besonders für Anwendungen geeignet, die eine Paritätsprüfung oder spezielle logische Bedingungen mit drei Eingangssignalen erfordern.

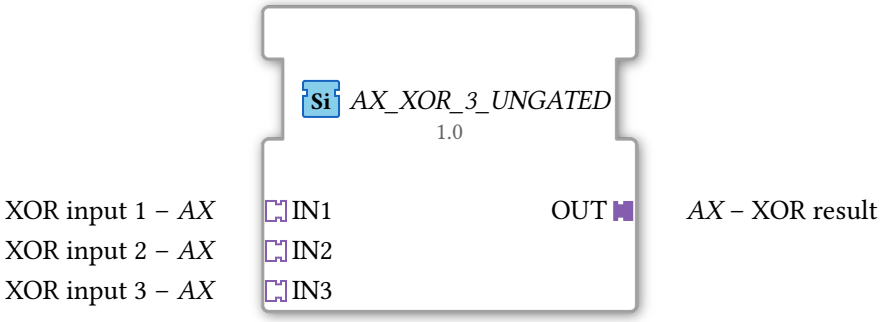
AX_XOR_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AX_XOR_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_XOR_3_UNGATED Funktionsblock ist ein generischer Baustein zur Berechnung der booleschen XOR-Verknüpfung mit drei Eingängen. Er implementiert die exklusive ODER-Operation für drei binäre Eingangssignale und gibt das entsprechende Ergebnis aus.



Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Sockets (Eingänge):

- **IN1**: XOR-Eingang 1 (Adapter-Typ: unidirectional::AX)
- **IN2**: XOR-Eingang 2 (Adapter-Typ: unidirectional::AX)
- **IN3**: XOR-Eingang 3 (Adapter-Typ: unidirectional::AX)

Plugs (Ausgänge):

- **OUT**: XOR-Ergebnis (Adapter-Typ: unidirectional::AX)

Funktionsweise
Der Funktionsblock berechnet die XOR-Verknüpfung für drei Eingangssignale. Die XOR-Operation gibt genau dann ein „wahr“-Signal (true) aus, wenn eine ungerade Anzahl der Eingänge „wahr“ ist. Bei drei Eingängen bedeutet dies:

- Ergebnis ist true, wenn genau einer oder alle drei Eingänge true sind
- Ergebnis ist false, wenn keine oder genau zwei Eingänge true sind

Technische Besonderheiten

- Generischer Funktionsblock mit der Klassennamen-Kennung ‚GEN_AX_XOR‘
- Verwendet unidirektionale Adapter für die Kommunikation
- Implementiert nach IEC 61499-2 Standard
- Keine internen Zustände oder Speicherung erforderlich

Zustandsübersicht
Der Baustein besitzt keine internen Zustände und arbeitet statisch. Die Ausgabe wird ausschließlich auf Basis der aktuellen Eingangswerte berechnet.

Anwendungsszenarien

- Schaltnetze mit drei Eingängen
- Paritätsprüfungen
- Sicherheitskritische Steuerungen, wo ungerade Aktivierung erforderlich ist
- Logische Verknüpfungen in Automatisierungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-XOR-Bausteinen mit zwei Eingängen bietet AX_XOR_3_UNGATED die Erweiterung auf drei Eingänge. Während ein 2-Eingang-XOR nur bei ungleichen Eingängen true liefert, berücksichtigt der 3-Eingang-XOR die Parität aller drei Signale.

Vergleich mit XOR_3

- **AX_XOR_3**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

🛠️ Zugehörige Übungen

- Uebung_002a7_AX

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

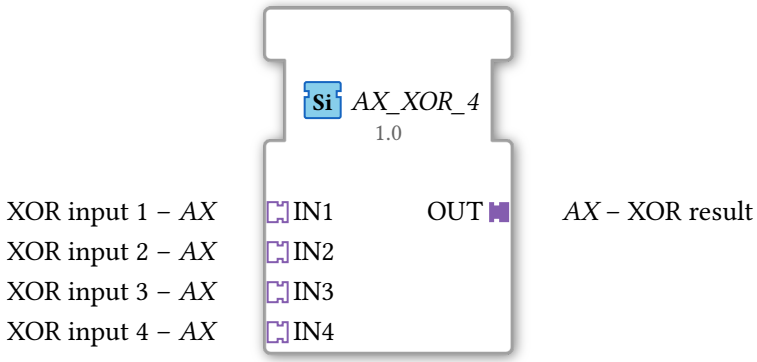
Der AX_XOR_3_UNGATED Funktionsblock bietet eine zuverlässige und effiziente Lösung für dreifache XOR-Verknüpfungen in IEC 61499-basierten Steuerungssystemen. Durch die Verwendung von Adaptern ermöglicht er eine flexible Integration in verschiedene Systemarchitekturen und ist besonders für Anwendungen geeignet, die eine Paritätsprüfung oder spezielle logische Bedingungen mit drei Eingangssignalen erfordern.

AX_XOR_4

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_XOR_4 Funktionsblock ist ein generischer Baustein zur Berechnung der booleschen XOR-Verknüpfung mit vier Eingängen. Er implementiert die exklusive ODER-Operation für bis zu vier verschiedene Eingangssignale.



Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Plug-Adapter:

- **OUT**: Unidirektionaler Adapter für das XOR-Ergebnis

Socket-Adapter:

- **IN1**: Unidirektionaler Adapter für XOR-Eingang 1
- **IN2**: Unidirektionaler Adapter für XOR-Eingang 2
- **IN3**: Unidirektionaler Adapter für XOR-Eingang 3
- **IN4**: Unidirektionaler Adapter für XOR-Eingang 4

Funktionsweise

Der Funktionsblock berechnet die XOR-Verknüpfung über vier Eingänge. Die XOR-Operation liefert genau dann ein „wahr"-Signal, wenn eine ungerade Anzahl der Eingänge aktiv ist. Bei vier Eingängen bedeutet dies:

- Ergebnis = 1, wenn 1 oder 3 Eingänge aktiv sind
- Ergebnis = 0, wenn 0, 2 oder 4 Eingänge aktiv sind

Die Kommunikation erfolgt ausschließlich über die definierten Adapter-Schnittstellen.

Technische Besonderheiten

- Generischer Funktionsblock mit der Klasse ‚GEN_AX_XOR'
- Verwendet unidirektionale Adapter für alle Ein- und Ausgänge
- Keine direkten Ereignis- oder Datenschnittstellen
- Vollständige Adapter-basierte Architektur

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AX_XOR_4 keine internen Zustände. Die Ausgabe wird ausschließlich von den aktuellen Eingangswerten bestimmt.

Anwendungsszenarien

- Parity-Check in Steuerungssystemen
- Ungerade-Paritätsprüfung
- Sicherheitskritische Schaltungen mit Mehrfacheingängen
- Verteilte Logikoperationen in Automatisierungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-XOR-Bausteinen bietet AX_XOR_4:

- Vier Eingänge statt der typischen zwei Eingänge
- Rein adapterbasierte Schnittstelle
- Spezialisierung auf unidirektionale AX-Adapter
- Keine Ereignissteuerung, sondern kontinuierliche Operation

Vergleich mit XOR_4

- `AX_XOR_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der AX_XOR_4 Funktionsblock stellt eine spezialisierte Lösung für XOR-Operationen mit vier Eingängen in adapterbasierten Systemarchitekturen dar. Seine rein adapterbasierte Schnittstelle macht ihn besonders geeignet für modulare Systemdesigns, bei denen die Kommunikation über standardisierte Adapter erfolgt.

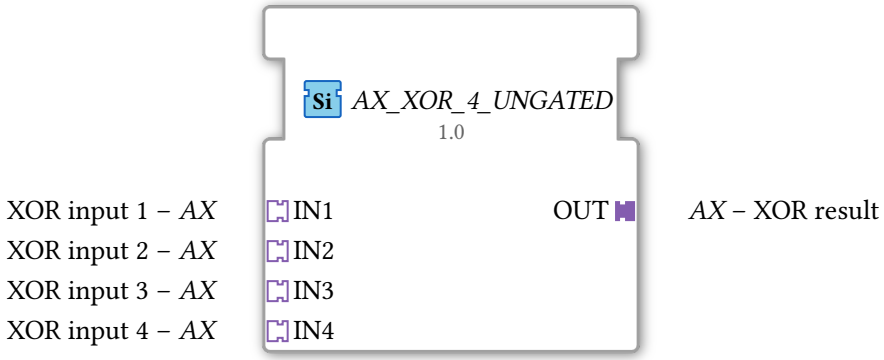
AX_XOR_4_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von AX_XOR_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_XOR_4_UNGATED Funktionsblock ist ein generischer Baustein zur Berechnung der booleschen XOR-Verknüpfung mit vier Eingängen. Er implementiert die exklusive ODER-Operation für bis zu vier verschiedene Eingangssignale.



Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Plug-Adapter:

- **OUT**: Unidirektionaler Adapter für das XOR-Ergebnis

Socket-Adapter:

- **IN1**: Unidirektionaler Adapter für XOR-Eingang 1
- **IN2**: Unidirektionaler Adapter für XOR-Eingang 2
- **IN3**: Unidirektionaler Adapter für XOR-Eingang 3
- **IN4**: Unidirektionaler Adapter für XOR-Eingang 4

Funktionsweise

Der Funktionsblock berechnet die XOR-Verknüpfung über vier Eingänge. Die XOR-Operation liefert genau dann ein „wahr"-Signal, wenn eine ungerade Anzahl der Eingänge aktiv ist. Bei vier Eingängen bedeutet dies:

- Ergebnis = 1, wenn 1 oder 3 Eingänge aktiv sind

- Ergebnis = 0, wenn 0, 2 oder 4 Eingänge aktiv sind

Die Kommunikation erfolgt ausschließlich über die definierten Adapter-Schnittstellen.

Technische Besonderheiten

- Generischer Funktionsblock mit der Klasse ‚GEN_AX_XOR'
- Verwendet unidirektionale Adapter für alle Ein- und Ausgänge
- Keine direkten Ereignis- oder Datenschnittstellen
- Vollständige Adapter-basierte Architektur

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AX_XOR_4_UNGATED keine internen Zustände. Die Ausgabe wird ausschließlich von den aktuellen Eingangswerten bestimmt.

Anwendungsszenarien

- Parity-Check in Steuerungssystemen
- Ungerade-Paritätsprüfung
- Sicherheitskritische Schaltungen mit Mehrfacheingängen
- Verteilte Logikoperationen in Automatisierungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-XOR-Bausteinen bietet AX_XOR_4_UNGATED:

- Vier Eingänge statt der typischen zwei Eingänge
- Rein adapterbasierte Schnittstelle
- Spezialisierung auf unidirektionale AX-Adapter
- Keine Ereignissteuerung, sondern kontinuierliche Operation

Vergleich mit XOR_4

- **`AX_XOR_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

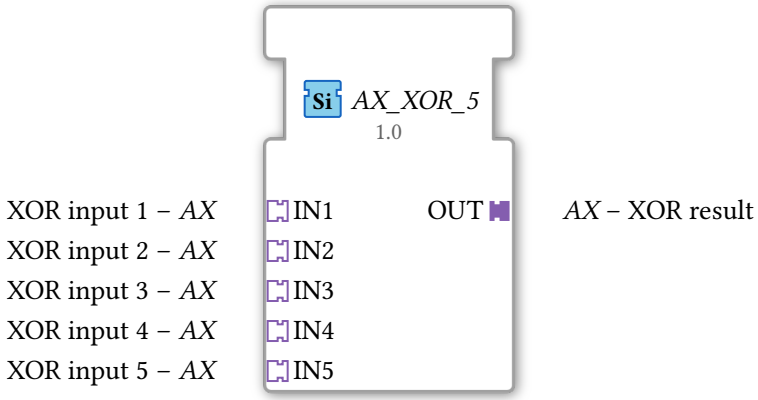
Der AX_XOR_4_UNGATED Funktionsblock stellt eine spezialisierte Lösung für XOR-Operationen mit vier Eingängen in adapterbasierten Systemarchitekturen dar. Seine rein adapterbasierte Schnittstelle macht ihn besonders geeignet für modulare Systemdesigns, bei denen die Kommunikation über standardisierte Adapter erfolgt.

AX_XOR_5

• – ○ • – ○ • – ○ •

Einleitung

Der AX_XOR_5 ist ein generischer Funktionsblock zur Berechnung der booleschen XOR-Verknüpfung mit fünf Eingängen. Der Baustein implementiert die exklusive ODER-Operation für bis zu fünf Eingangssignale und gibt das Ergebnis über einen Adapter-Ausgang aus.



Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Eingangsadapter:

- **IN1**: XOR-Eingang 1 (unidirektionaler AX-Adapter)
- **IN2**: XOR-Eingang 2 (unidirektionaler AX-Adapter)
- **IN3**: XOR-Eingang 3 (unidirektionaler AX-Adapter)
- **IN4**: XOR-Eingang 4 (unidirektionaler AX-Adapter)
- **IN5**: XOR-Eingang 5 (unidirektionaler AX-Adapter)

Ausgangsadapter:

- **OUT**: XOR-Ergebnis (unidirektionaler AX-Adapter)

Funktionsweise

Der Funktionsblock berechnet die XOR-Verknüpfung über alle fünf Eingänge. Die XOR-Operation (exklusives ODER) liefert genau dann ein „wahr“-Signal, wenn eine ungerade Anzahl der Eingänge aktiv ist. Bei einer geraden Anzahl aktiver Eingänge wird „falsch“ ausgegeben.

Die mathematische Formel lautet:

```
OUT = IN1 XOR IN2 XOR IN3 XOR IN4 XOR IN5
```

Technische Besonderheiten

- Verwendet unidirektionale AX-Adapter für alle Schnittstellen
- Implementiert als generischer Funktionsblock mit der Bezeichnung ‚GEN_AX_XOR‘
- Unterstützt bis zu fünf unabhängige Eingangssignale
- Keine Ereignissteuerung - arbeitet kontinuierlich basierend auf den Adapter-Eingängen

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein ohne Speicherfunktion handelt, besitzt der AX_XOR_5 keine internen Zustände. Die Ausgabe wird ausschließlich durch die aktuellen Eingangswerte bestimmt.

Anwendungsszenarien

- Paritätsprüfung in digitalen Schaltungen
- Fehlererkennung in Sicherheitssystemen
- Steuerungslogik mit Mehrfachbedingungen
- Verteilte Entscheidungsfindung in Automatisierungssystemen
- Sicherheitskritische Anwendungen mit redundanten Sensoren

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-XOR-Bausteinen mit weniger Eingängen bietet AX_XOR_5 die Möglichkeit, bis zu fünf Signale gleichzeitig zu verarbeiten. Während einfache XOR-Blöcke typischerweise nur zwei Eingänge haben, ermöglicht dieser Baustein komplexere logische Verknüpfungen ohne zusätzliche Verschachtelung.

Vergleich mit XOR_5

- **`AX_XOR_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der AX_XOR_5 Funktionsblock stellt eine flexible Lösung für XOR-Operationen mit mehreren Eingängen dar. Seine Adapter-basierte Schnittstelle ermöglicht eine einfache Integration in bestehende Steuerungssysteme, während die generische Implementierung Wiederverwendbarkeit und Erweiterbarkeit gewährleistet.

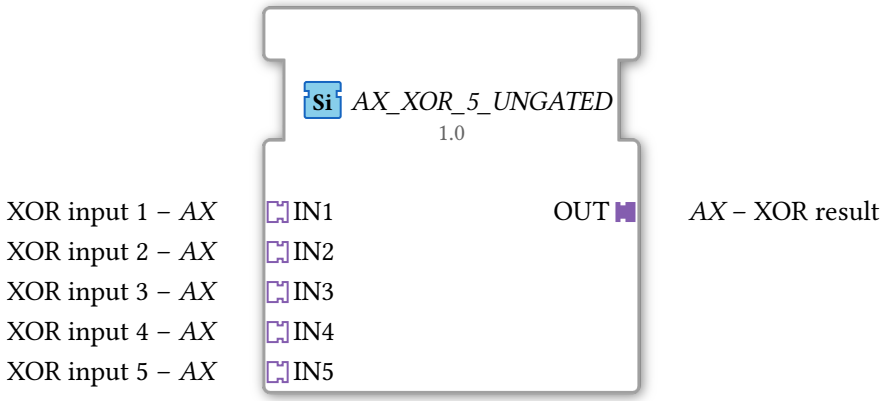
AX_XOR_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AX_XOR_5. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_XOR_5_UNGATED ist ein generischer Funktionsblock zur Berechnung der booleschen XOR-Verknüpfung mit fünf Eingängen. Der Baustein implementiert die exklusive ODER-Operation für bis zu fünf Eingangssignale und gibt das Ergebnis über einen Adapter-Ausgang aus.



Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden.

Adapter
Eingangsadapter:

- **IN1**: XOR-Eingang 1 (unidirektionaler AX-Adapter)
- **IN2**: XOR-Eingang 2 (unidirektionaler AX-Adapter)
- **IN3**: XOR-Eingang 3 (unidirektionaler AX-Adapter)
- **IN4**: XOR-Eingang 4 (unidirektionaler AX-Adapter)
- **IN5**: XOR-Eingang 5 (unidirektionaler AX-Adapter)

Ausgangsadapter:

- **OUT**: XOR-Ergebnis (unidirektionaler AX-Adapter)

Funktionsweise
Der Funktionsblock berechnet die XOR-Verknüpfung über alle fünf Eingänge. Die XOR-Operation (exklusives ODER) liefert genau dann ein „wahr“-Signal, wenn eine ungerade Anzahl der Eingänge aktiv ist. Bei einer geraden Anzahl aktiver Eingänge wird „falsch“ ausgegeben.

Die mathematische Formel lautet:

```
OUT = IN1 XOR IN2 XOR IN3 XOR IN4 XOR IN5
```

Technische Besonderheiten
- Verwendet unidirektionale AX-Adapter für alle Schnittstellen
- Implementiert als generischer Funktionsblock mit der Bezeichnung ‚GEN_AX_XOR‘
- Unterstützt bis zu fünf unabhängige Eingangssignale
- Keine Ereignissteuerung - arbeitet kontinuierlich basierend auf den Adapter-Eingängen

Zustandsübersicht
Da es sich um einen kombinatorischen Logikbaustein ohne Speicherfunktion handelt, besitzt der AX_XOR_5_UNGATED keine internen Zustände. Die Ausgabe wird ausschließlich durch die aktuellen Eingangswerte bestimmt.

Anwendungsszenarien
- Paritätsprüfung in digitalen Schaltungen
- Fehlererkennung in Sicherheitssystemen
- Steuerungslogik mit Mehrfachbedingungen
- Verteilte Entscheidungsfindung in Automatisierungssystemen
- Sicherheitskritische Anwendungen mit redundanten Sensoren

⚖️ Vergleich mit ähnlichen Bausteinen
Im Vergleich zu Standard-XOR-Bausteinen mit weniger Eingängen bietet AX_XOR_5_UNGATED die Möglichkeit, bis zu fünf Signale gleichzeitig zu verarbeiten. Während einfache XOR-Blöcke typischerweise nur zwei Eingänge haben, ermöglicht dieser Baustein komplexere logische Verknüpfungen ohne zusätzliche Verschachtelung.

Vergleich mit XOR_5

- **AX_XOR_5**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

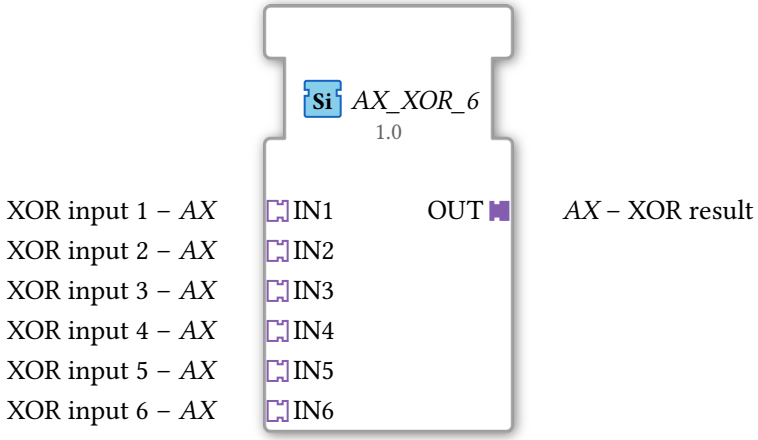
Der AX_XOR_5_UNGATED Funktionsblock stellt eine flexible Lösung für XOR-Operationen mit mehreren Eingängen dar. Seine Adapter-basierte Schnittstelle ermöglicht eine einfache Integration in bestehende Steuerungssysteme, während die generische Implementierung Wiederverwendbarkeit und Erweiterbarkeit gewährleistet.

AX_XOR_6

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_XOR_6 ist ein generischer Funktionsblock zur Berechnung der booleschen XOR-Verknüpfung mit sechs Eingängen. Der Baustein implementiert die exklusive ODER-Operation für bis zu sechs binäre Eingangssignale und gibt das Ergebnis über einen Adapter-Ausgang aus.



Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Plug-Adapter:

- **OUT** (Typ: adapter::types::unidirectional::AX) - XOR-Ergebnis

Socket-Adapter:

- **IN1** (Typ: adapter::types::unidirectional::AX) - XOR-Eingang 1
- **IN2** (Typ: adapter::types::unidirectional::AX) - XOR-Eingang 2
- **IN3** (Typ: adapter::types::unidirectional::AX) - XOR-Eingang 3
- **IN4** (Typ: adapter::types::unidirectional::AX) - XOR-Eingang 4
- **IN5** (Typ: adapter::types::unidirectional::AX) - XOR-Eingang 5

- **IN6** (Typ: adapter::types::unidirectional::AX) - XOR-Eingang 6

Funktionsweise

Der Funktionsblock berechnet die XOR-Verknüpfung aller sechs Eingänge. Die XOR-Operation liefert genau dann ein „wahr"-Signal (TRUE), wenn eine ungerade Anzahl der Eingänge auf „wahr" gesetzt ist. Bei einer geraden Anzahl von „wahr"-Eingängen wird „falsch" (FALSE) ausgegeben.

Technische Besonderheiten

- Generischer Funktionsblock mit der generischen Klasse ‚GEN_AX_XOR'
- Verwendet unidirektionale AX-Adapter für die Kommunikation
- Keine Ereignissteuerung - arbeitet kontinuierlich basierend auf den Eingangswerten
- Unterstützt sechs unabhängige Eingänge für komplexe XOR-Berechnungen

Zustandsübersicht

Da es sich um einen rein kombinatorischen Baustein ohne Ereignissteuerung handelt, besitzt der AX_XOR_6 keine internen Zustände. Die Ausgabe wird ausschließlich durch die aktuellen Eingangswerte bestimmt.

Anwendungsszenarien

- Paritätsprüfung in Datenübertragungssystemen
- Sicherheitskritische Steuerungen mit Mehrfachredundanz
- Fehlererkennung in binären Signalketten
- Verschlüsselungsalgorithmen mit XOR-Operationen
- Steuerungssysteme mit Mehrfacheingangs-Logik

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-XOR-Bausteinen mit weniger Eingängen bietet der AX_XOR_6 die Möglichkeit, bis zu sechs Signale gleichzeitig zu verarbeiten. Während einfache XOR-Blöcke typischerweise nur zwei Eingänge haben, ermöglicht dieser Baustein komplexere logische Verknüpfungen ohne zusätzliche Verschachtelung mehrerer Blöcke.

Vergleich mit XOR_6

- `AX_XOR_6_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der AX_XOR_6 ist ein spezialisierter Funktionsblock für anspruchsvolle XOR-Operationen mit mehreren Eingängen. Durch die Adapter-basierte Schnittstelle lässt er sich gut in bestehende Steuerungssysteme integrieren und bietet eine effiziente Lösung für komplexe boolesche Verknüpfungen ohne zusätzliche Verkabelungskomplexität.

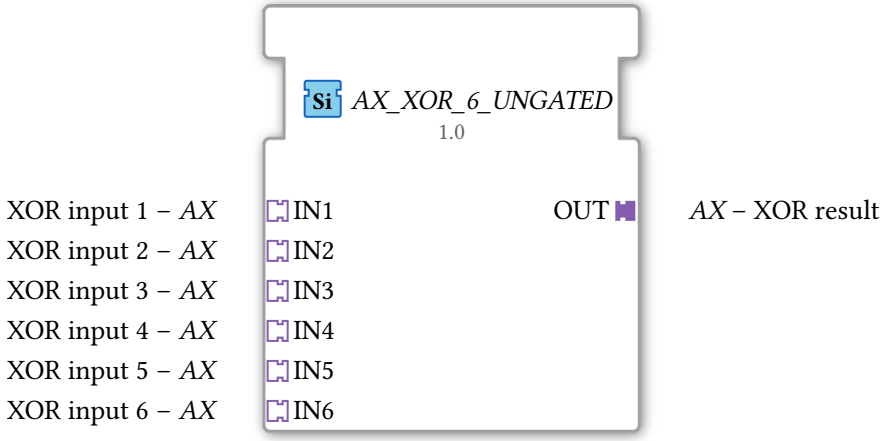
AX_XOR_6_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AX_XOR_6`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_XOR_6_UNGATED ist ein generischer Funktionsblock zur Berechnung der booleschen XOR-Verknüpfung mit sechs Eingängen. Der Baustein implementiert die exklusive ODER-Operation für bis zu sechs binäre Eingangssignale und gibt das Ergebnis über einen Adapter-Ausgang aus.



Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Plug-Adapter:

- **OUT** (Typ: adapter::types::unidirectional::AX) - XOR-Ergebnis

Socket-Adapter:

- **IN1** (Typ: adapter::types::unidirectional::AX) - XOR-Eingang 1
- **IN2** (Typ: adapter::types::unidirectional::AX) - XOR-Eingang 2
- **IN3** (Typ: adapter::types::unidirectional::AX) - XOR-Eingang 3
- **IN4** (Typ: adapter::types::unidirectional::AX) - XOR-Eingang 4
- **IN5** (Typ: adapter::types::unidirectional::AX) - XOR-Eingang 5
- **IN6** (Typ: adapter::types::unidirectional::AX) - XOR-Eingang 6

Funktionsweise

Der Funktionsblock berechnet die XOR-Verknüpfung aller sechs Eingänge. Die XOR-Operation liefert genau dann ein „wahr“-Signal (TRUE), wenn eine ungerade Anzahl der Eingänge auf „wahr“ gesetzt ist. Bei einer geraden Anzahl von „wahr“-Eingängen wird „falsch“ (FALSE) ausgegeben.

Technische Besonderheiten

- Generischer Funktionsblock mit der generischen Klasse ‚GEN_AX_XOR‘
- Verwendet unidirektionale AX-Adapter für die Kommunikation
- Keine Ereignissteuerung - arbeitet kontinuierlich basierend auf den Eingangswerten
- Unterstützt sechs unabhängige Eingänge für komplexe XOR-Berechnungen

Zustandsübersicht

Da es sich um einen rein kombinatorischen Baustein ohne Ereignissteuerung handelt, besitzt der AX_XOR_6_UNGATED keine internen Zustände. Die Ausgabe wird ausschließlich durch die aktuellen Eingangswerte bestimmt.

Anwendungsszenarien

- Paritätsprüfung in Datenübertragungssystemen
- Sicherheitskritische Steuerungen mit Mehrfachredundanz
- Fehlererkennung in binären Signalketten
- Verschlüsselungsalgorithmen mit XOR-Operationen
- Steuerungssysteme mit Mehrfacheingangs-Logik

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-XOR-Bausteinen mit weniger Eingängen bietet der AX_XOR_6_UNGATED die Möglichkeit, bis zu sechs Signale gleichzeitig zu verarbeiten. Während einfache XOR-Blöcke typischerweise nur zwei Eingänge haben, ermöglicht dieser Baustein komplexere logische Verknüpfungen ohne zusätzliche Verschachtelung mehrerer Blöcke.

Vergleich mit XOR_6

- **`AX_XOR_6`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

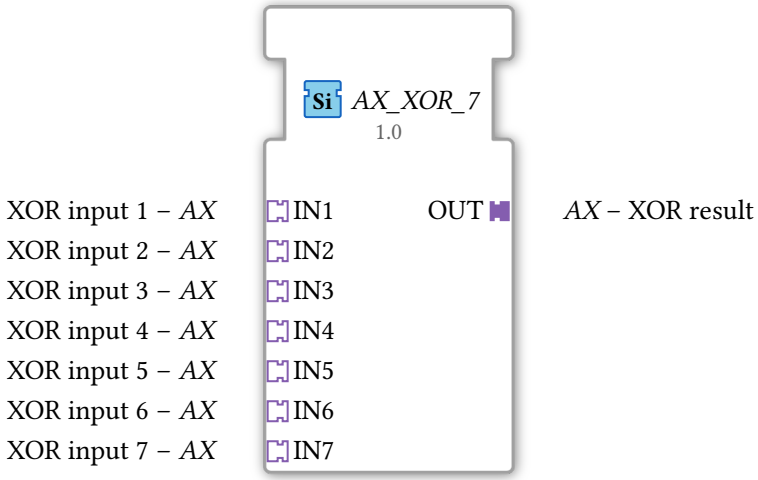
Der AX_XOR_6_UNGATED ist ein spezialisierter Funktionsblock für anspruchsvolle XOR-Operationen mit mehreren Eingängen. Durch die Adapter-basierte Schnittstelle lässt er sich gut in bestehende Steuerungssysteme integrieren und bietet eine effiziente Lösung für komplexe boolesche Verknüpfungen ohne zusätzliche Verkabelungskomplexität.

AX_XOR_7

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_XOR_7 Funktionsblock ist ein generischer Baustein zur Berechnung der booleschen XOR-Verknüpfung mit sieben Eingängen. Er ermöglicht die Verarbeitung von bis zu sieben Eingangssignalen und gibt das Ergebnis der XOR-Operation aus.



Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereigniseingänge vorhanden.

Ereignis-Ausgänge
Keine Ereignisausgänge vorhanden.

Daten-Eingänge
Keine direkten Dateneingänge vorhanden.

Daten-Ausgänge
Keine direkten Datenausgänge vorhanden.

Adapter
Plug-Adapter:

- `OUT` - XOR-Ergebnis (Adapter-Typ: `adapter::types::unidirectional::AX`)

Socket-Adapter:

- `IN1` - XOR-Eingang 1 (Adapter-Typ: `adapter::types::unidirectional::AX`)
- `IN2` - XOR-Eingang 2 (Adapter-Typ: `adapter::types::unidirectional::AX`)
- `IN3` - XOR-Eingang 3 (Adapter-Typ: `adapter::types::unidirectional::AX`)
- `IN4` - XOR-Eingang 4 (Adapter-Typ: `adapter::types::unidirectional::AX`)
- `IN5` - XOR-Eingang 5 (Adapter-Typ: `adapter::types::unidirectional::AX`)
- `IN6` - XOR-Eingang 6 (Adapter-Typ: `adapter::types::unidirectional::AX`)
- `IN7` - XOR-Eingang 7 (Adapter-Typ: `adapter::types::unidirectional::AX`)

Funktionsweise
Der Funktionsblock berechnet die XOR-Verknüpfung über alle sieben Eingänge. Die XOR-Operation liefert genau dann ein „wahr“-Signal (TRUE), wenn eine ungerade Anzahl der Eingänge auf „wahr“ gesetzt ist. Bei einer geraden Anzahl von „wahr“-Eingängen wird „falsch“ (FALSE) ausgegeben.

Technische Besonderheiten
- Generischer Funktionsblock mit der generischen Klassenbezeichnung `GEN_AX_XOR`
- Verwendet unidirektionale AX-Adapter für Ein- und Ausgänge
- Unterstützt genau sieben Eingänge für die XOR-Berechnung
- Implementiert als Adapter-basierte Lösung ohne direkte Ereignis- oder Datenschnittstellen

Zustandsübersicht
Der Funktionsblock besitzt keinen internen Zustand und arbeitet stateless. Die Ausgabe wird ausschließlich auf Basis der aktuellen Eingangswerte berechnet.

Anwendungsszenarien
- Paritätsprüfung in Datenübertragungssystemen
- Sicherheitskritische Steuerungen mit Mehrheitsentscheidung
- Fehlererkennung in binären Signalketten
- Logische Verknüpfungen in industriellen Automatisierungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen
Im Vergleich zu Standard-XOR-Bausteinen mit weniger Eingängen bietet AX_XOR_7 die Möglichkeit, XOR-Operationen mit bis zu sieben Eingängen durchzuführen. Während einfache XOR-Blöcke typischerweise nur zwei Eingänge verarbeiten, ermöglicht dieser Baustein komplexere Verknüpfungslogiken in einem einzigen Block.
Vergleich mit XOR_7

- **`AX_XOR_7_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der AX_XOR_7 Funktionsblock stellt eine effiziente Lösung für XOR-Operationen mit mehreren Eingängen dar. Durch die Adapter-basierte Architektur ermöglicht er eine flexible Integration in verschiedene Steuerungssysteme und eignet sich besonders für Anwendungen, die eine Prüfung auf ungerade Parität über mehrere Signale erfordern.

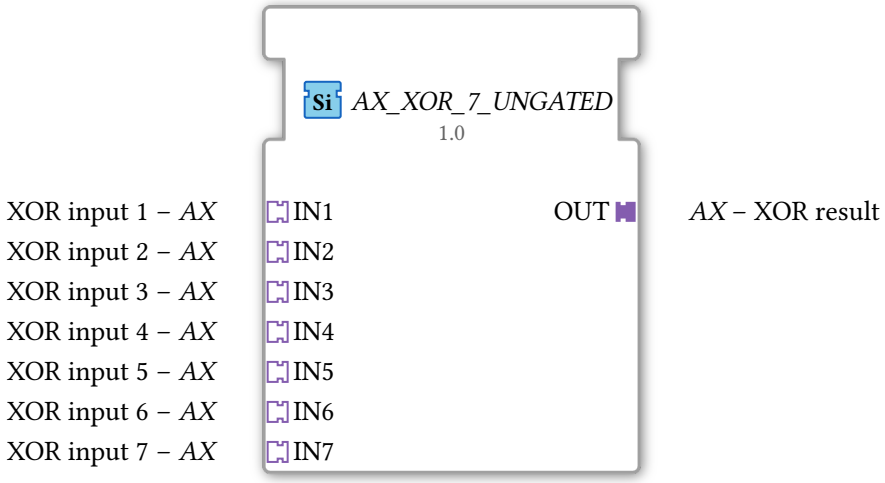
AX_XOR_7_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AX_XOR_7`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_XOR_7_UNGATED Funktionsblock ist ein generischer Baustein zur Berechnung der booleschen XOR-Verknüpfung mit sieben Eingängen. Er ermöglicht die Verarbeitung von bis zu sieben Eingangssignalen und gibt das Ergebnis der XOR-Operation aus.



Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereigniseingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignisausgänge vorhanden.

Daten-Eingänge

Keine direkten Dateneingänge vorhanden.

Daten-Ausgänge

Keine direkten Datenausgänge vorhanden.

Adapter

Plug-Adapter:

- `OUT` - XOR-Ergebnis (Adapter-Typ: `adapter::types::unidirectional::AX`)

Socket-Adapter:

- `IN1` - XOR-Eingang 1 (Adapter-Typ: `adapter::types::unidirectional::AX`)
- `IN2` - XOR-Eingang 2 (Adapter-Typ: `adapter::types::unidirectional::AX`)
- `IN3` - XOR-Eingang 3 (Adapter-Typ: `adapter::types::unidirectional::AX`)
- `IN4` - XOR-Eingang 4 (Adapter-Typ: `adapter::types::unidirectional::AX`)
- `IN5` - XOR-Eingang 5 (Adapter-Typ: `adapter::types::unidirectional::AX`)
- `IN6` - XOR-Eingang 6 (Adapter-Typ: `adapter::types::unidirectional::AX`)
- `IN7` - XOR-Eingang 7 (Adapter-Typ: `adapter::types::unidirectional::AX`)

Funktionsweise

Der Funktionsblock berechnet die XOR-Verknüpfung über alle sieben Eingänge. Die XOR-Operation liefert genau dann ein „wahr“-Signal (TRUE), wenn eine ungerade Anzahl der Eingänge auf „wahr“ gesetzt ist. Bei einer geraden Anzahl von „wahr“-Eingängen wird „falsch“ (FALSE) ausgegeben.

Technische Besonderheiten

- Generischer Funktionsblock mit der generischen Klassenbezeichnung `GEN_AX_XOR`
- Verwendet unidirektionale AX-Adapter für Ein- und Ausgänge
- Unterstützt genau sieben Eingänge für die XOR-Berechnung
- Implementiert als Adapter-basierte Lösung ohne direkte Ereignis- oder Datenschnittstellen

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand und arbeitet stateless. Die Ausgabe wird ausschließlich auf Basis der aktuellen Eingangswerte berechnet.

Anwendungsszenarien

- Paritätsprüfung in Datenübertragungssystemen
- Sicherheitskritische Steuerungen mit Mehrheitsentscheidung
- Fehlererkennung in binären Signalketten
- Logische Verknüpfungen in industriellen Automatisierungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-XOR-Bausteinen mit weniger Eingängen bietet AX_XOR_7_UNGATED die Möglichkeit, XOR-Operationen mit bis zu sieben Eingängen durchzuführen. Während einfache XOR-Blöcke typischerweise nur zwei Eingänge verarbeiten, ermöglicht dieser Baustein komplexere Verknüpfungslogiken in einem einzigen Block.

Vergleich mit XOR_7

- **`AX_XOR_7`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

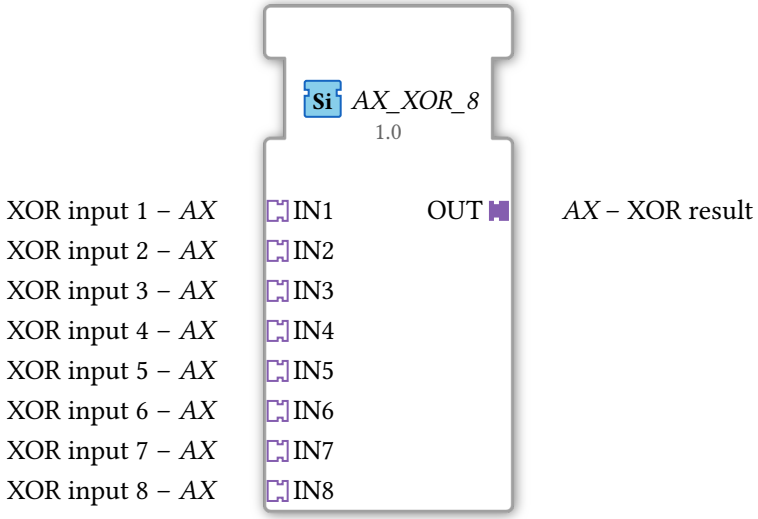
Der AX_XOR_7_UNGATED Funktionsblock stellt eine effiziente Lösung für XOR-Operationen mit mehreren Eingängen dar. Durch die Adapter-basierte Architektur ermöglicht er eine flexible Integration in verschiedene Steuerungssysteme und eignet sich besonders für Anwendungen, die eine Prüfung auf ungerade Parität über mehrere Signale erfordern.

AX_XOR_8

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_XOR_8 Funktionsblock ist ein generischer Baustein zur Berechnung der booleschen XOR-Verknüpfung mit 8 Eingängen. Er ermöglicht die Verarbeitung von logischen Signalen nach dem Exklusiv-ODER-Prinzip und ist für den Einsatz in Steuerungsanwendungen konzipiert.



Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Plug-Adapter:

- **OUT**: Unidirektionaler Adapter für das XOR-Ergebnis

Socket-Adapter:

- **IN1**: Unidirektionaler Adapter für XOR-Eingang 1
- **IN2**: Unidirektionaler Adapter für XOR-Eingang 2
- **IN3**: Unidirektionaler Adapter für XOR-Eingang 3
- **IN4**: Unidirektionaler Adapter für XOR-Eingang 4
- **IN5**: Unidirektionaler Adapter für XOR-Eingang 5
- **IN6**: Unidirektionaler Adapter für XOR-Eingang 6
- **IN7**: Unidirektionaler Adapter für XOR-Eingang 7
- **IN8**: Unidirektionaler Adapter für XOR-Eingang 8

Funktionsweise

Der Funktionsblock berechnet die XOR-Verknüpfung über alle 8 Eingänge. Die XOR-Operation (Exklusiv-ODER) liefert genau dann ein TRUE-Signal, wenn eine ungerade Anzahl der Eingänge TRUE ist. Bei einer geraden Anzahl von TRUE-Eingängen wird FALSE ausgegeben.

Technische Besonderheiten

- Generischer Funktionsblock mit der Klasse ‚GEN_AX_XOR‘
- Verwendet unidirektionale Adapter für alle Ein- und Ausgänge

- Keine direkten Ereignis- oder Datenschnittstellen
- Adapter-basierte Architektur für flexible Anbindung

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AX_XOR_8 keinen internen Zustand. Die Ausgabe wird ausschließlich von den aktuellen Eingangswerten bestimmt.

Anwendungsszenarien

- Paritätsprüfung in Datenübertragungssystemen
- Sicherheitskritische Steuerungen mit Mehrfachredundanz
- Logische Verknüpfungen in Automatisierungssystemen
- Fehlererkennung in binären Signalketten

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen XOR-Bausteinen mit weniger Eingängen bietet AX_XOR_8 die Möglichkeit, bis zu 8 Signale gleichzeitig zu verknüpfen. Die ausschließliche Verwendung von Adaptern statt direkter Daten- und Ereigniseingänge ermöglicht eine höhere Flexibilität in komplexen Systemarchitekturen.

Vergleich mit XOR_8

- **`AX_XOR_8_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der AX_XOR_8 ist ein spezialisierter Logikbaustein für anspruchsvolle XOR-Operationen mit mehreren Eingängen. Seine Adapter-basierte Schnittstelle macht ihn besonders geeignet für modulare Systemdesigns, bei denen flexible Verbindungen zwischen Funktionsblöcken erforderlich sind.

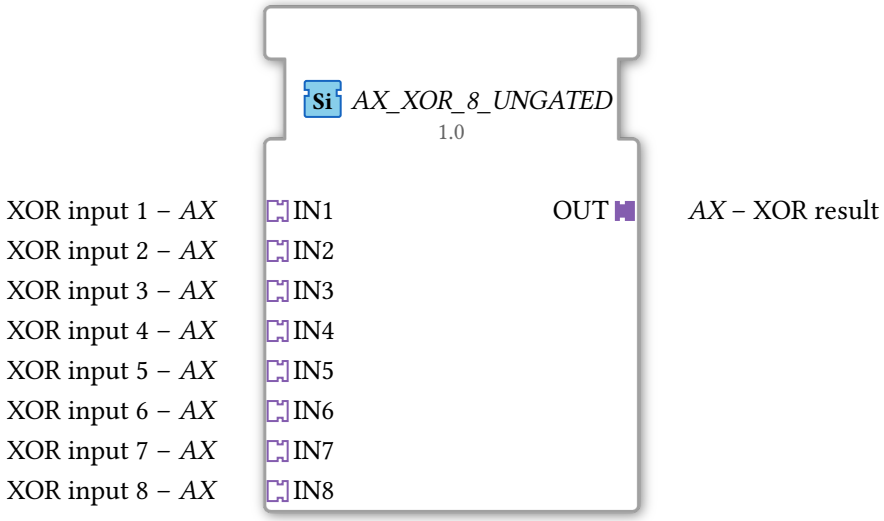
AX_XOR_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AX_XOR_8. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_XOR_8_UNGATED Funktionsblock ist ein generischer Baustein zur Berechnung der booleschen XOR-Verknüpfung mit 8 Eingängen. Er ermöglicht die Verarbeitung von logischen Signalen nach dem Exklusiv-ODER-Prinzip und ist für den Einsatz in Steuerungsanwendungen konzipiert.



Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Plug-Adapter:

- **OUT**: Unidirektionaler Adapter für das XOR-Ergebnis

Socket-Adapter:

- **IN1**: Unidirektionaler Adapter für XOR-Eingang 1
- **IN2**: Unidirektionaler Adapter für XOR-Eingang 2
- **IN3**: Unidirektionaler Adapter für XOR-Eingang 3
- **IN4**: Unidirektionaler Adapter für XOR-Eingang 4
- **IN5**: Unidirektionaler Adapter für XOR-Eingang 5
- **IN6**: Unidirektionaler Adapter für XOR-Eingang 6
- **IN7**: Unidirektionaler Adapter für XOR-Eingang 7
- **IN8**: Unidirektionaler Adapter für XOR-Eingang 8

Funktionsweise
Der Funktionsblock berechnet die XOR-Verknüpfung über alle 8 Eingänge. Die XOR-Operation (Exklusiv-ODER) liefert genau dann ein TRUE-Signal, wenn eine ungerade Anzahl der Eingänge TRUE ist. Bei einer geraden Anzahl von TRUE-Eingängen wird FALSE ausgegeben.

Technische Besonderheiten

- Generischer Funktionsblock mit der Klasse ‚GEN_AX_XOR‘
- Verwendet unidirektionale Adapter für alle Ein- und Ausgänge
- Keine direkten Ereignis- oder Datenschnittstellen
- Adapter-basierte Architektur für flexible Anbindung

Zustandsübersicht

Da es sich um einen kombinatorischen Logikbaustein handelt, besitzt der AX_XOR_8_UNGATED keinen internen Zustand. Die Ausgabe wird ausschließlich von den aktuellen Eingangswerten bestimmt.

Anwendungsszenarien

- Paritätsprüfung in Datenübertragungssystemen
- Sicherheitskritische Steuerungen mit Mehrfachredundanz
- Logische Verknüpfungen in Automatisierungssystemen
- Fehlererkennung in binären Signalketten

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen XOR-Bausteinen mit weniger Eingängen bietet AX_XOR_8_UNGATED die Möglichkeit, bis zu 8 Signale gleichzeitig zu verknüpfen. Die ausschließliche Verwendung von Adaptern statt direkter Daten- und Ereigniseingänge ermöglicht eine höhere Flexibilität in komplexen Systemarchitekturen.

Vergleich mit XOR_8

- **AX_XOR_8**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

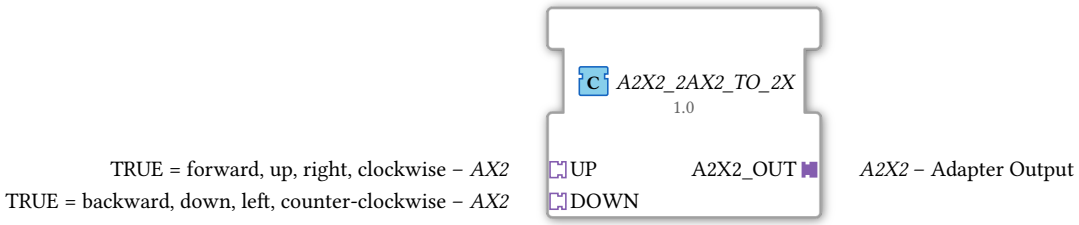
Der AX_XOR_8_UNGATED ist ein spezialisierter Logikbaustein für anspruchsvolle XOR-Operationen mit mehreren Eingängen. Seine Adapter-basierte Schnittstelle macht ihn besonders geeignet für modulare Systemdesigns, bei denen flexible Verbindungen zwischen Funktionsblöcken erforderlich sind.

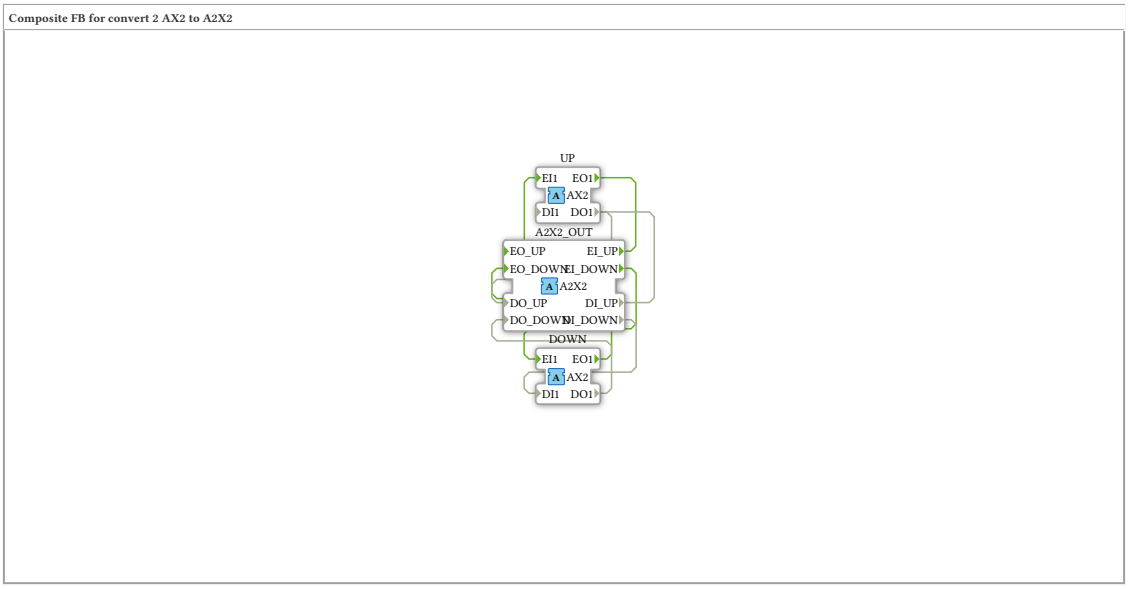
2.1.11.8 conversion

bidirectional

BOOL

A2X2_2AX2_TO_2X





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock A2X2_2AX2_TO_2X setzt einen A2X2-Plug aus zwei AX2-Sockets zusammen – einem für den UP-Kanal, einem für den DOWN-Kanal. Da AX2 selbst schon bidirektional ist (1 Ereignis/1 Bool je Richtung), genügt pro Kanal ein einziger AX2, um beide Richtungen von A2X2 abzudecken.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Kommunikation läuft ausschließlich über die Adapter.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **A2X2_OUT** (Plug): zusammengesetzter Ausgang vom Typ `adapter::types::bidirectional::A2X2`
- **UP** (Socket): Kanal UP, vom Typ `adapter::types::bidirectional::AX2` – TRUE = vorwärts, hoch, rechts, im Uhrzeigersinn
- **DOWN** (Socket): Kanal DOWN, vom Typ `adapter::types::bidirectional::AX2` – TRUE = rückwärts, runter, links, gegen den Uhrzeigersinn

Funktionsweise

Für jeden Kanal wird die komplette bidirektionale Verdrahtung zwischen A2X2 und dem zugehörigen AX2-Socket hergestellt: Was `A2X2_OUT` auf seiner Anfrage-Seite empfängt (`EI_UP`/`DI_UP`), wird an `UP.EI1`/`UP.DI1` weitergereicht; was `UP` auf seiner Indikations-Seite liefert (`EO1`/`DO1`), erscheint an `A2X2_OUT.EO_UP`/`DO_UP`. Für DOWN gilt dieselbe Verdrahtung mit dem `DOWN`-Socket. Beide Kanäle sind vollständig unabhängig.

Technische Besonderheiten

- Nutzt die Bidirektionalität von AX2 direkt aus – jeder Kanal braucht nur einen einzigen Adapter, nicht zwei

- Reine Verdrahtung ohne Logik oder Zustand
- Jede Zielvariable hat genau einen Schreiber, keine Mehrfach-Datenverbindungen

Zustandsübersicht

Der Baustein ist zustandslos:

- A2X2_OUT.EI_UP ⟶ UP.EI1, A2X2_OUT.DI_UP ⟶ UP.DI1
- UP.EO1 ⟶ A2X2_OUT.EO_UP, UP.DO1 ⟶ A2X2_OUT.DO_UP
- A2X2_OUT.EI_DOWN ⟶ DOWN.EI1, A2X2_OUT.DI_DOWN ⟶ DOWN.DI1
- DOWN.EO1 ⟶ A2X2_OUT.EO_DOWN, DOWN.DO1 ⟶ A2X2_OUT.DO_DOWN

Anwendungsszenarien

- Aufbau eines A2X2-Endgeräts aus zwei bereits vorhandenen, unabhängigen AX2-Kanälen
- Systeme, in denen UP und DOWN aus historischen Gründen getrennt als AX2 verdrahtet sind, aber nach außen als ein A2X2 auftreten sollen
- Modulare Zusammensetzung größerer Adapter aus kleineren, wiederverwendbaren Bausteinen

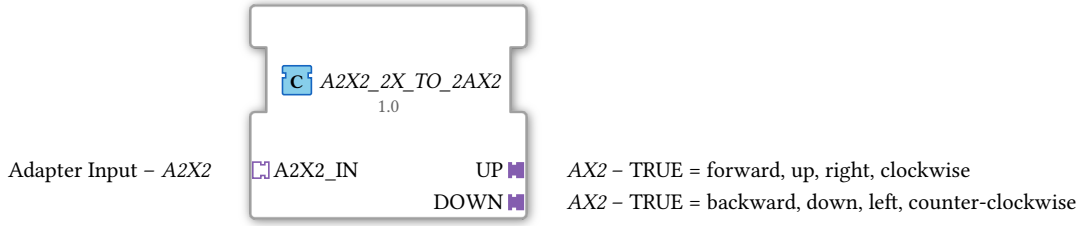
⚖️ Vergleich mit ähnlichen Bausteinen

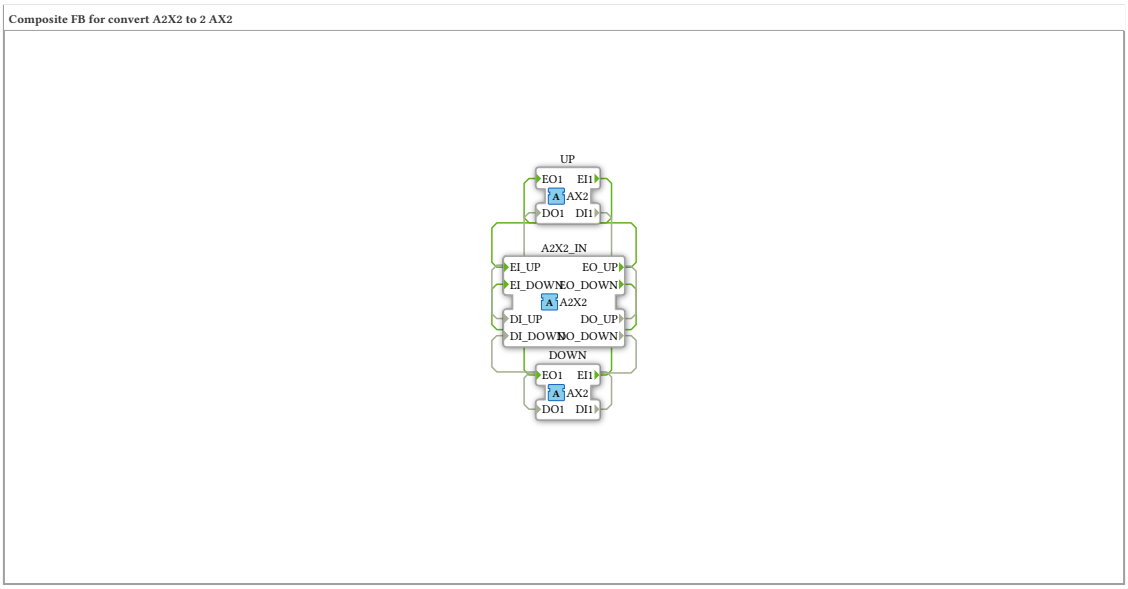
Das Gegenstück A2X2_2X_TO_2AX2 zerlegt statt zu komponieren. Für dieselbe Aufgabe gibt es mit A2X2_4AX_TO_2X eine Alternative, die statt zwei bidirektionalen AX2 vier unidirektionale AX verwendet (zwei je Kanal, je einer pro Richtung) – sinnvoll, wenn nur unidirektionale AX-Infrastruktur zur Verfügung steht. Der unidirektionale Vorgänger A2X_2AX_TO_2X komponiert analog aus zwei einfachen AX-Adaptern.

Fazit

A2X2_2AX2_TO_2X ist der effizienteste Weg, ein A2X2 aus zwei vorhandenen AX2-Kanälen zu bauen, da beide Adapter bereits bidirektional sind und keine zusätzliche Logik nötig ist.

A2X2_2X_TO_2AX2





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock A2X2_2X_TO_2AX2 ist die Umkehrung von A2X2_2AX2_TO_2X: Er zerlegt einen A2X2-Socket in zwei AX2-Plugs – einen für UP, einen für DOWN.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Kommunikation läuft ausschließlich über die Adapter.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **A2X2_IN** (Socket): zu zerlegender Eingang vom Typ `adapter::types::bidirectional::A2X2`
- **UP** (Plug): Kanal UP, vom Typ `adapter::types::bidirectional::AX2` – TRUE = vorwärts, hoch, rechts, im Uhrzeigersinn
- **DOWN** (Plug): Kanal DOWN, vom Typ `adapter::types::bidirectional::AX2` – TRUE = rückwärts, runter, links, gegen den Uhrzeigersinn

Funktionsweise

Für jeden Kanal wird die komplette bidirektionale Verdrahtung zwischen dem A2X2-Socket und dem zugehörigen AX2-Plug hergestellt: Was `A2X2_IN` auf seiner Indikations-Seite empfängt (`EO_UP`/`DO_UP`), wird an `UP.EO1`/`UP.DO1` weitergegeben; was am `UP`-Plug auf dessen Anfrage-Seite eintrifft (`EI1`/`DI1`), wird an `A2X2_IN.EI_UP`/`DI_UP` zurückgemeldet. Für DOWN gilt dieselbe Verdrahtung mit dem `DOWN`-Plug. Beide Kanäle sind vollständig unabhängig.

Technische Besonderheiten

- Nutzt die Bidirektionalität von AX2 direkt aus – jeder Kanal braucht nur einen einzigen Adapter, nicht zwei
- Reine Verdrahtung ohne Logik oder Zustand
- Jede Zielvariable hat genau einen Schreiber, keine Mehrfach-Datenverbindungen

Zustandsübersicht

Der Baustein ist zustandslos:

- A2X2_IN.EO_UP ⟶ UP.EO1, A2X2_IN.DO_UP ⟶ UP.DO1
- UP.EI1 ⟶ A2X2_IN.EI_UP, UP.DI1 ⟶ A2X2_IN.DI_UP
- A2X2_IN.EO_DOWN ⟶ DOWN.EO1, A2X2_IN.DO_DOWN ⟶ DOWN.DO1
- DOWN.EI1 ⟶ A2X2_IN.EI_DOWN, DOWN.DI1 ⟶ A2X2_IN.DI_DOWN

Anwendungsszenarien

- Aufsplitten eines A2X2-Bussignals in zwei unabhängig weiterverarbeitbare AX2-Kanäle
- Anbindung vorhandener AX2-basierter Teilsysteme an ein zentrales A2X2
- Testaufbauten, die jeden Kanal einzeln beobachten oder steuern müssen

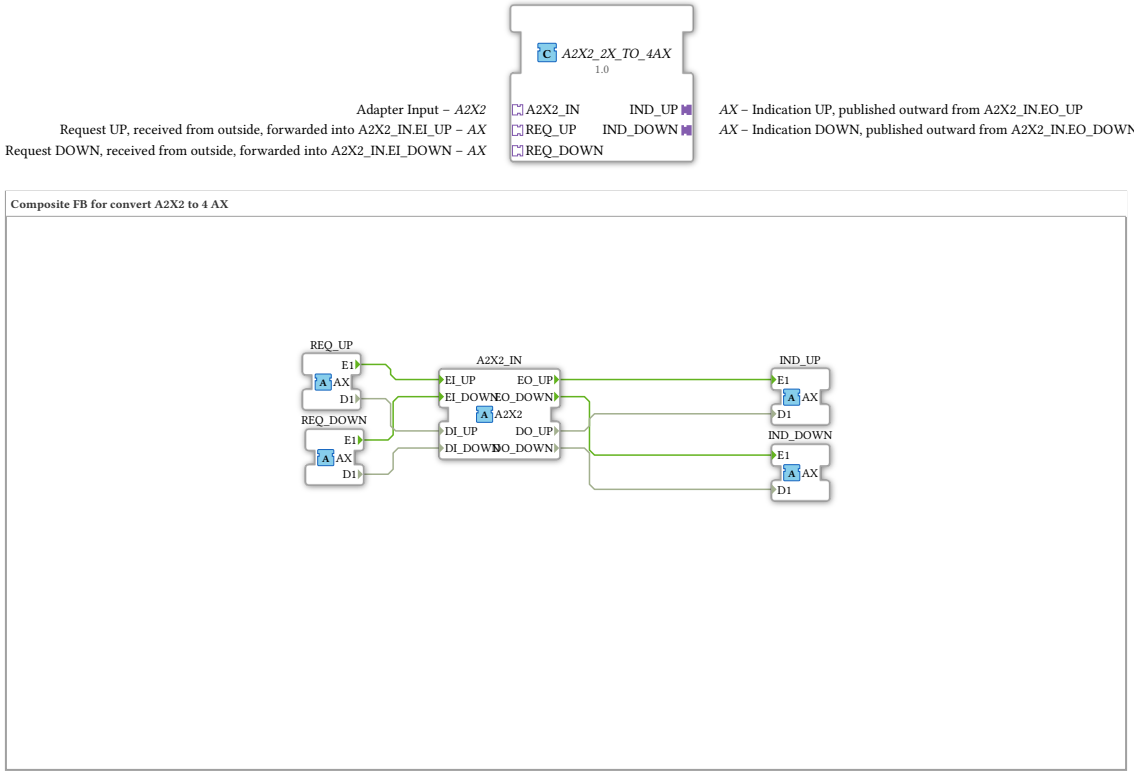
⚖️ Vergleich mit ähnlichen Bausteinen

Das Gegenstück A2X2_2AX2_TO_2X komponiert statt zu zerlegen. Für dieselbe Aufgabe gibt es mit A2X2_4AX_TO_2X / A2X2_2X_TO_4AX eine Alternative mit vier unidirektionalen AX-Adaptern statt zwei bidirektionalen AX2. Der unidirektionale Vorgänger A2X_2X_TO_2AX zerlegt analog in zwei einfache AX-Adapter.

Fazit

A2X2_2X_TO_2AX2 ist der effizienteste Weg, ein A2X2-Signal in zwei unabhängige AX2-Kanäle aufzuteilen, da beide Adapter bereits bidirektional sind und keine zusätzliche Logik nötig ist.

A2X2_2X_TO_4AX



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock A2X2_2X_TO_4AX zerlegt einen A2X2-Socket in **vier** unidirektionale AX-Adapter. Da A2X2 zwei Kanäle (UP/DOWN) hat und jeder davon selbst bidirektional ist, braucht man pro Kanal zwei unidirektionale AX, um beide Richtungen abzudecken – eins für die Indikations-Richtung, eins für die Anfrage-Richtung. 2 Kanäle × 2 Richtungen = 4 AX.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Kommunikation läuft ausschließlich über die Adapter.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **A2X2_IN** (Socket): zu zerlegender Eingang vom Typ `adapter::types::bidirectional::A2X2`
- **REQ_UP** (Socket): Anfrage UP, von außen empfangen und an `A2X2_IN.EI_UP` weitergereicht, Typ `adapter::types::unidirectional::AX`

- **REQ_DOWN** (Socket): Anfrage DOWN, von außen empfangen und an `A2X2_IN.EI_DOWN` weitergereicht, Typ `adapter::types::unidirectional::AX`
- **IND_UP** (Plug): Indikation UP, aus `A2X2_IN.EO_UP` nach außen veröffentlicht, Typ `adapter::types::unidirectional::AX`
- **IND_DOWN** (Plug): Indikation DOWN, aus `A2X2_IN.EO_DOWN` nach außen veröffentlicht, Typ `adapter::types::unidirectional::AX`

Funktionsweise

Was der A2X2-Socket auf seiner Indikations-Seite empfängt (`EO_UP`/`DO_UP` bzw. `EO_DOWN`/`DO_DOWN`), wird unverändert über die unidirektionalen Plugs `IND_UP`/`IND_DOWN` veröffentlicht. Umgekehrt wird alles, was an den unidirektionalen Sockets `REQ_UP`/`REQ_DOWN` von außen ankommt, an die Anfrage-Seite des A2X2-Sockets (`EI_UP`/`DI_UP` bzw. `EI_DOWN`/`DI_DOWN`) weitergereicht. Damit stehen beide Richtungen jedes Kanals als eigener, rein unidirektionaler AX-Adapter zur Verfügung.

Technische Besonderheiten

- Vier separate AX-Adapter statt zwei AX2, weil AX kein `EventInputs`/`EventOutputs`-Paar in einem Adapter vereint – jede Richtung braucht ihren eigenen Adapter
- Reine Verdrahtung ohne Logik oder Zustand
- Jede Zielvariable hat genau einen Schreiber, keine Mehrfach-Datenverbindungen

Zustandsübersicht

Der Baustein ist zustandslos:

- A2X2_IN.EO_UP ⟶ IND_UP.E1, A2X2_IN.DO_UP ⟶ IND_UP.D1
- A2X2_IN.EO_DOWN ⟶ IND_DOWN.E1, A2X2_IN.DO_DOWN ⟶ IND_DOWN.D1
- REQ_UP.E1 ⟶ A2X2_IN.EI_UP, REQ_UP.D1 ⟶ A2X2_IN.DI_UP
- REQ_DOWN.E1 ⟶ A2X2_IN.EI_DOWN, REQ_DOWN.D1 ⟶ A2X2_IN.DI_DOWN

Anwendungsszenarien

- Anbindung eines A2X2-Signals an eine Infrastruktur, die nur unidirektionale AX-Adapter kennt (z. B. ältere Teilsysteme)
- Getrennte Verarbeitung von Anfrage- und Indikations-Richtung, etwa zur unabhängigen Protokollierung oder Filterung
- Systeme, in denen bidirektionale AX2-Adapter nicht verfügbar oder nicht gewünscht sind

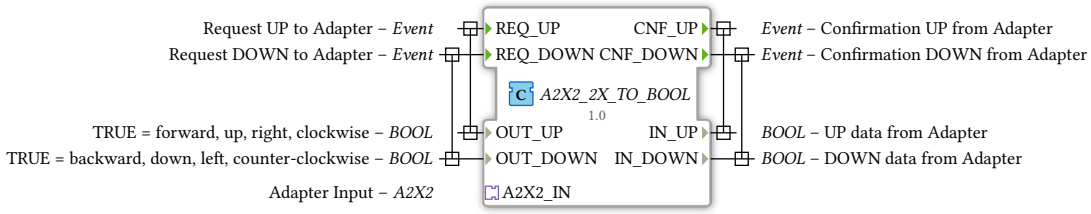
⚖️ Vergleich mit ähnlichen Bausteinen

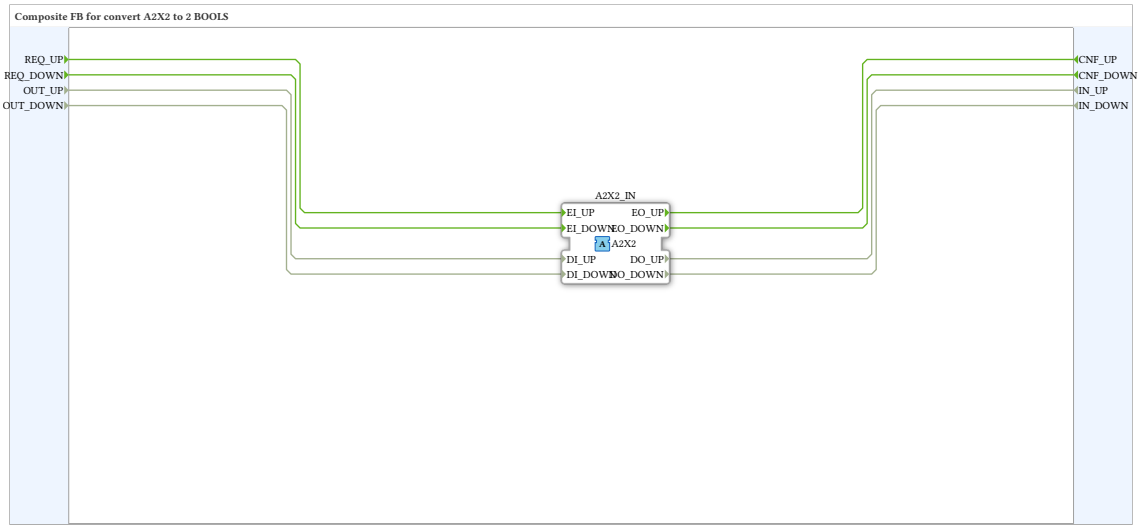
Für dieselbe Aufgabe (A2X2 zerlegen) gibt es mit A2X2_2X_TO_2AX2 eine einfachere Alternative, die nur zwei bidirektionale AX2 statt vier unidirektionaler AX benötigt – wo AX2-Infrastruktur zur Verfügung steht, ist das die schlankere Lösung. Das Gegenstück zu diesem Baustein ist A2X2_4AX_TO_2X, das aus vier AX wieder ein A2X2 zusammensetzt.

Fazit

A2X2_2X_TO_4AX ist die richtige Wahl, wenn ein A2X2-Signal in eine rein unidirektionale AX-Landschaft integriert werden muss – auf Kosten von doppelt so vielen Adaptern wie bei der AX2-basierten Alternative.

A2X2_2X_TO_BOOL





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock A2X2_2X_TO_BOOL ist das Socket-Gegenstück zu A2X2_BOOL_TO_2X: Er übersetzt das bidirektionale A2X2-Adapterformat zurück in zwei einfache BOOL-Kanäle (UP und DOWN), indem er einen A2X2-Socket bereitstellt.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ_UP**: Anfrage-Ereignis für den UP-Kanal, liefert `OUT_UP`
- **REQ_DOWN**: Anfrage-Ereignis für den DOWN-Kanal, liefert `OUT_DOWN`

Ereignis-Ausgänge

- **CNF_UP**: Bestätigungs-Ereignis für den UP-Kanal, liefert `IN_UP`
- **CNF_DOWN**: Bestätigungs-Ereignis für den DOWN-Kanal, liefert `IN_DOWN`

Daten-Eingänge

- **OUT_UP**: BOOL, TRUE = vorwärts, hoch, rechts, im Uhrzeigersinn
- **OUT_DOWN**: BOOL, TRUE = rückwärts, runter, links, gegen den Uhrzeigersinn

Daten-Ausgänge

- **IN_UP**: BOOL, UP-Daten vom Adapter
- **IN_DOWN**: BOOL, DOWN-Daten vom Adapter

Adapter

- **A2X2_IN** (Socket): Adapter-Eingang vom Typ `adapter::types::bidirectional::A2X2`

Funktionsweise

Empfängt der Socket `A2X2_IN` auf seiner Indikations-Seite Ereignisse (`EO_UP`/`DO_UP` bzw. `EO_DOWN`/`DO_DOWN`), werden diese unmittelbar als `CNF_UP`/`IN_UP` bzw. `CNF_DOWN`/`IN_DOWN` nach außen gemeldet. Trifft umgekehrt ein `REQ_UP`- oder `REQ_DOWN`-Ereignis ein, wird der zugehörige `OUT_UP`/`OUT_DOWN`-Wert über den Socket (`EI_UP`/`DI_UP` bzw. `EI_DOWN`/`DI_DOWN`) nach außen gesendet. Beide Kanäle laufen komplett unabhängig voneinander.

Technische Besonderheiten

- Reine 1:1-Durchleitung, kein interner Zustand, keine Gatter oder Logikbausteine nötig
- Nutzt einen A2X2-Socket, d. h. dieser Baustein ist das Gegenstück, an das ein A2X2-Plug angeschlossen wird
- Identische Schnittstelle zu A2X2_BOOL_TO_2X, nur die Adapter-Rolle (Socket statt Plug) unterscheidet sich

Zustandsübersicht

Der Baustein ist zustandslos, jede Verbindung wirkt sofort und direkt:

- A2X2_IN.EO_UP ⟶ CNF_UP, A2X2_IN.DO_UP ⟶ IN_UP
- A2X2_IN.EO_DOWN ⟶ CNF_DOWN, A2X2_IN.DO_DOWN ⟶ IN_DOWN
- REQ_UP ⟶ A2X2_IN.EI_UP, OUT_UP ⟶ A2X2_IN.DI_UP
- REQ_DOWN ⟶ A2X2_IN.EI_DOWN, OUT_DOWN ⟶ A2X2_IN.DI_DOWN

Anwendungsszenarien

- Auslesen zweier UP/DOWN-Kanäle eines A2X2-Busteilnehmers als einfache BOOL-Signale
- Integration eines A2X2-Endgeräts in eine klassisch verdrahtete Steuerung
- Testbausteine, die als Gegenstelle für A2X2_BOOL_TO_2X dienen

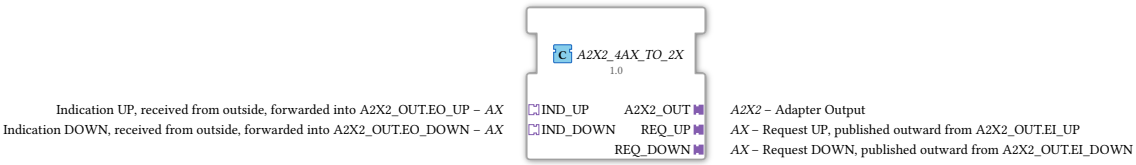
⚖️ Vergleich mit ähnlichen Bausteinen

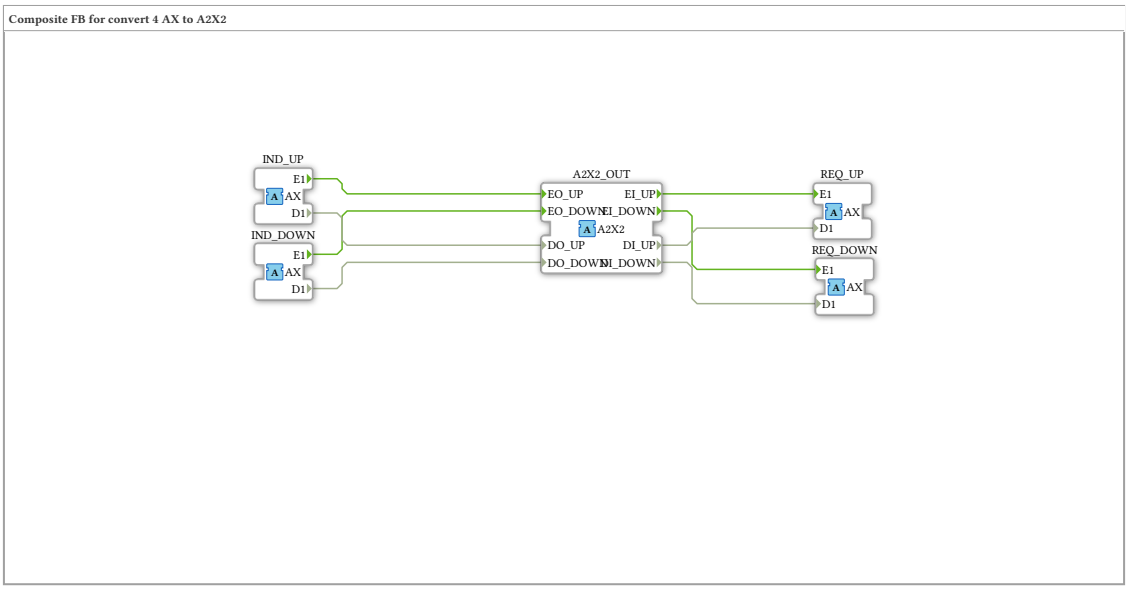
Das Plug-Gegenstück ist A2X2_BOOL_TO_2X. Für den unidirektionalen Fall existiert A2X_2X_TO_BOOL. Der einkanalige bidirektionale Vorgänger ist AX2_X_TO_BOOL.

Fazit

A2X2_2X_TO_BOOL ist das direkte Gegenstück zu A2X2_BOOL_TO_2X und macht ein A2X2-Bussignal als zwei einfache BOOL-Kanäle für die restliche Anwendung nutzbar.

A2X2_4AX_TO_2X





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock A2X2_4AX_TO_2X ist die Umkehrung von A2X2_2X_TO_4AX: Er setzt aus **vier** unidirektionalen AX-Adaptern einen A2X2-Plug zusammen. Zwei Kanäle (UP/DOWN) × zwei Richtungen (Indikation/Anfrage) = vier AX.

Schnittstellenstruktur

Ereignis-Eingänge
Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Kommunikation läuft ausschließlich über die Adapter.

Ereignis-Ausgänge
Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge
Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge
Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter
- **A2X2_OUT** (Plug): zusammengesetzter Ausgang vom Typ `adapter::types::bidirectional::A2X2`
- **REQ_UP** (Plug): Anfrage UP, aus `A2X2_OUT.EI_UP` nach außen veröffentlicht, Typ `adapter::types::unidirectional::AX`
- **REQ_DOWN** (Plug): Anfrage DOWN, aus `A2X2_OUT.EI_DOWN` nach außen veröffentlicht, Typ `adapter::types::unidirectional::AX`
- **IND_UP** (Socket): Indikation UP, von außen empfangen und an `A2X2_OUT.EO_UP` weitergereicht, Typ `adapter::types::unidirectional::AX`
- **IND_DOWN** (Socket): Indikation DOWN, von außen empfangen und an `A2X2_OUT.EO_DOWN` weitergereicht, Typ `adapter::types::unidirectional::AX`

Funktionsweise
Was an den unidirektionalen Sockets `IND_UP`/`IND_DOWN` von außen ankommt, wird auf die Indikations-Seite des A2X2-Plugs (`EO_UP`/`DO_UP` bzw. `EO_DOWN`/`DO_DOWN`) durchgereicht. Umgekehrt wird alles, was der A2X2-Plug auf seiner Anfrage-Seite empfängt (`EI_UP`/`DI_UP` bzw. `EI_DOWN`/`DI_DOWN`), über die unidirektionalen Plugs `REQ_UP`/`REQ_DOWN` veröffentlicht.

Technische Besonderheiten
- Vier separate AX-Adapter statt zwei AX2, weil AX kein `EventInputs`/`EventOutputs`-Paar in einem Adapter vereint – jede Richtung braucht ihren eigenen Adapter
- Reine Verdrahtung ohne Logik oder Zustand
- Jede Zielvariable hat genau einen Schreiber, keine Mehrfach-Datenverbindungen

Zustandsübersicht
Der Baustein ist zustandslos:

- IND_UP.E1 ⟶ A2X2_OUT.EO_UP, IND_UP.D1 ⟶ A2X2_OUT.DO_UP
- IND_DOWN.E1 ⟶ A2X2_OUT.EO_DOWN, IND_DOWN.D1 ⟶ A2X2_OUT.DO_DOWN
- A2X2_OUT.EI_UP ⟶ REQ_UP.E1, A2X2_OUT.DI_UP ⟶ REQ_UP.D1
- A2X2_OUT.EI_DOWN ⟶ REQ_DOWN.E1, A2X2_OUT.DI_DOWN ⟶ REQ_DOWN.D1

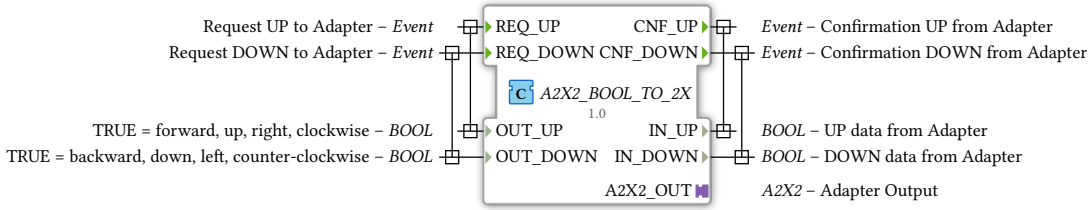
Anwendungsszenarien
- Aufbau eines A2X2-Endgeräts aus vier bereits vorhandenen, unidirektionalen AX-Signalen
- Migrationsszenarien, in denen eine bestehende AX-Infrastruktur schrittweise auf A2X2 umgestellt wird
- Systeme, in denen bidirektionale AX2-Adapter nicht verfügbar oder nicht gewünscht sind

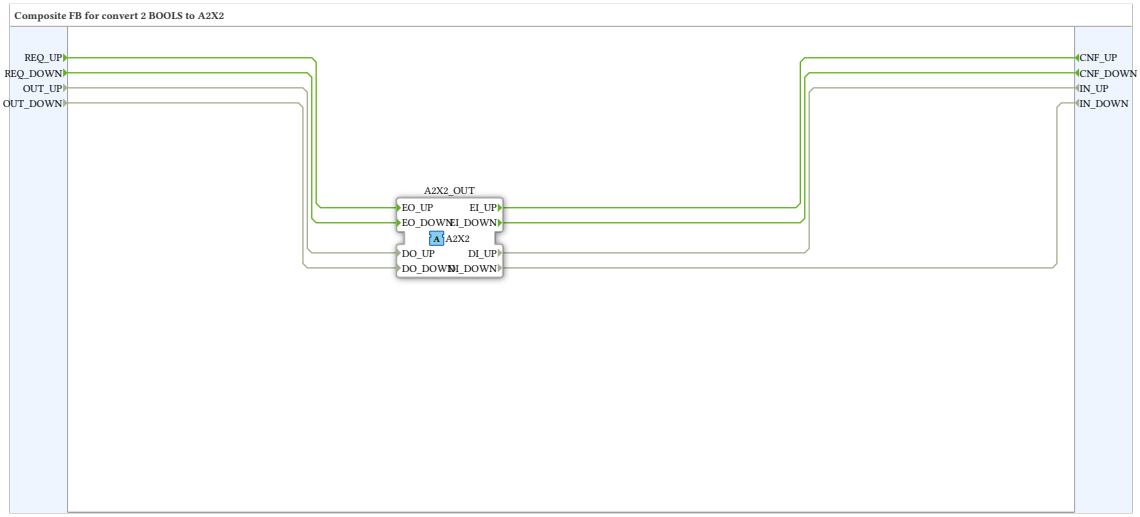
⚖️ Vergleich mit ähnlichen Bausteinen
Für dieselbe Aufgabe (A2X2 komponieren) gibt es mit A2X2_2AX2_TO_2X eine einfachere Alternative, die nur zwei bidirektionale AX2 statt vier unidirektionaler AX benötigt – wo AX2-Infrastruktur zur Verfügung steht, ist das die schlankere Lösung. Das Gegenstück zu diesem Baustein ist A2X2_2X_TO_4AX, das ein A2X2 wieder in vier AX zerlegt.

Fazit

A2X2_4AX_TO_2X ist die richtige Wahl, wenn ein A2X2-Endgerät aus rein unidirektionaler AX-Infrastruktur aufgebaut werden muss – auf Kosten von doppelt so vielen Adaptern wie bei der AX2-basierten Alternative.

A2X2_BOOL_TO_2X





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock A2X2_BOOL_TO_2X ist ein Composite-Funktionsblock, der zwei einfache BOOL-Kanäle (UP und DOWN) in das bidirektionale A2X2-Adapterformat übersetzt. Er stellt dazu einen A2X2-Plug bereit und bildet jeden der beiden Kanäle unabhängig über ein eigenes Anfrage-/Bestätigungs-Ereignispaar ab.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ_UP**: Anfrage-Ereignis für den UP-Kanal, liefert `OUT_UP`
- **REQ_DOWN**: Anfrage-Ereignis für den DOWN-Kanal, liefert `OUT_DOWN`

Ereignis-Ausgänge

- **CNF_UP**: Bestätigungs-Ereignis für den UP-Kanal, liefert `IN_UP`
- **CNF_DOWN**: Bestätigungs-Ereignis für den DOWN-Kanal, liefert `IN_DOWN`

Daten-Eingänge

- **OUT_UP**: BOOL, TRUE = vorwärts, hoch, rechts, im Uhrzeigersinn
- **OUT_DOWN**: BOOL, TRUE = rückwärts, runter, links, gegen den Uhrzeigersinn

Daten-Ausgänge

- **IN_UP**: BOOL, UP-Daten vom Adapter
- **IN_DOWN**: BOOL, DOWN-Daten vom Adapter

Adapter

- **A2X2_OUT** (Plug): Adapter-Ausgang vom Typ `adapter::types::bidirectional::A2X2`

Funktionsweise

Trifft ein `REQ_UP`-Ereignis ein, wird der aktuelle Wert von `OUT_UP` über `A2X2_OUT.EO_UP/A2X2_OUT.DO_UP` gesendet; analog für `REQ_DOWN/OUT_DOWN` über `EO_DOWN/DO_DOWN`. Umgekehrt wird alles, was der Adapter auf seiner Bestätigungs-Seite empfängt (`A2X2_OUT.EI_UP/A2X2_OUT.EI_DOWN` mit `DI_UP/DI_DOWN`), unverändert als `CNF_UP/IN_UP` bzw. `CNF_DOWN/IN_DOWN` nach außen weitergereicht. Beide Kanäle laufen komplett unabhängig voneinander.

Technische Besonderheiten

- Reine 1:1-Durchleitung, kein interner Zustand, keine Gatter oder Logikbausteine nötig
- Jeder Kanal (UP/DOWN) hat sein eigenes Ereignispaar – es gibt keine geteilte Variable, die von zwei Quellen gleichzeitig beschrieben werden müsste
- Nutzt den A2X2-Plug, d. h. dieser Baustein tritt als „Endgerät“ auf, an das ein A2X2-Socket angeschlossen wird

Zustandsübersicht

Der Baustein ist zustandslos, jede Verbindung wirkt sofort und direkt:

- REQ_UP ⟶ A2X2_OUT.EO_UP, OUT_UP ⟶ A2X2_OUT.DO_UP
- REQ_DOWN ⟶ A2X2_OUT.EO_DOWN, OUT_DOWN ⟶ A2X2_OUT.DO_DOWN
- A2X2_OUT.EI_UP ⟶ CNF_UP, A2X2_OUT.DI_UP ⟶ IN_UP
- A2X2_OUT.EI_DOWN ⟶ CNF_DOWN, A2X2_OUT.DI_DOWN ⟶ IN_DOWN

Anwendungsszenarien

- Anschluss zweier einfacher BOOL-Signale (z. B. zweier Taster oder Endschalter für UP/DOWN) an ein A2X2-Bussystem
- Testbausteine, die A2X2-Verkehr ohne echte Hardware simulieren
- Brücke zwischen klassischer BOOL-Verdrahtung und Adapter-basierten Subnetzen

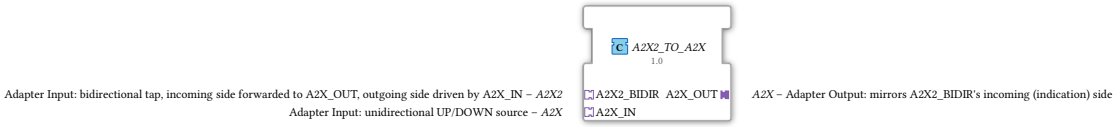
⚖️ Vergleich mit ähnlichen Bausteinen

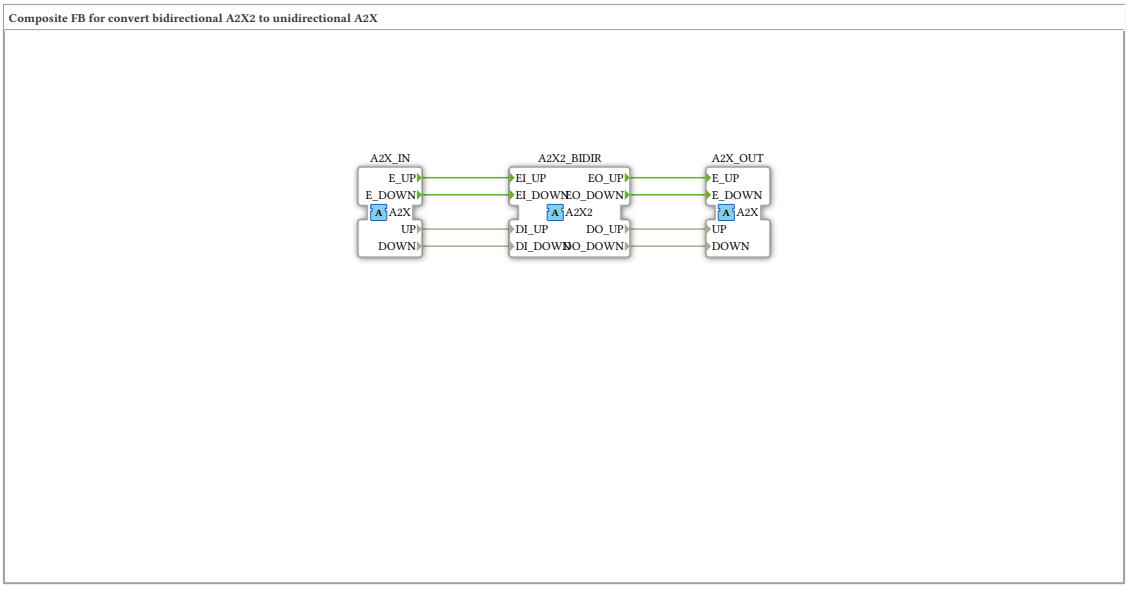
Das Socket-Gegenstück A2X2_2X_TO_BOOL hat dieselbe Schnittstelle, verwendet aber einen A2X2-Socket statt eines Plugs. Für den unidirektionalen Fall existiert mit A2X_BOOL_TO_2X ein Baustein mit identischer Grundidee, aber ohne Anfrage-Seite (nur `E_UP/E_DOWN`). Der einkanalige bidirektionale Vorgänger ist AX2_BOOL_TO_X, von dem dieser Baustein das Muster auf zwei unabhängige Kanäle verdoppelt.

Fazit

A2X2_BOOL_TO_2X ist die einfachste Möglichkeit, plain-BOOL-Signale mit einem A2X2-Adapter zu verbinden – ohne Zustand, ohne Logik, ein direkter 1:1-Durchgriff pro Kanal.

A2X2_TO_A2X





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock A2X2_TO_A2X ist das Spiegelbild von A2X_TO_A2X2: Hier steht der bidirektionale A2X2-Abzweig als Socket auf der linken (Eingangs-)Seite, ergänzt um einen zweiten unidirektionalen A2X-Socket und einen unidirektionalen A2X-Plug.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Kommunikation läuft ausschließlich über die Adapter.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **A2X2_BIDIR** (Socket): bidirektionaler Abzweig, dessen eingehende Seite an `A2X_OUT` weitergereicht wird und dessen ausgehende Seite von `A2X_IN` gespeist wird, Typ `adapter::types::bidirectional::A2X2`
- **A2X_IN** (Socket): unidirektionale UP/DOWN-Quelle, Typ `adapter::types::unidirectional::A2X`
- **A2X_OUT** (Plug): spiegelt die eingehende (Indikations-)Seite von `A2X2_BIDIR` als unidirektionales Signal, Typ `adapter::types::unidirectional::A2X`

Funktionsweise

Der Baustein arbeitet in zwei Richtungen gleichzeitig: Was der Socket `A2X2_BIDIR` auf seiner Indikations-Seite empfängt (`EO_UP`/`DO_UP`, `EO_DOWN`/`DO_DOWN` – also was ein an A2X2_BIDIR angeschlossener bidirektionaler Teilnehmer meldet), wird unverändert als eigenständiges unidirektionales Signal über den Plug `A2X_OUT` ausgegeben. Umgekehrt wird alles, was am Socket `A2X_IN` eintrifft (`E_UP`/`UP`, `E_DOWN`/`DOWN`), auf die Anfrage-Seite von `A2X2_BIDIR` (`EI_UP`/`DI_UP`, `EI_DOWN`/`DI_DOWN`) veröffentlicht – A2X2_BIDIR sendet also dasselbe UP/DOWN-Signal an den angeschlossenen bidirektionalen Teilnehmer weiter.

Technische Besonderheiten

- Drei statt zwei Adapter: Die vom Tap empfangene und die separat eingespeiste Signalrichtung werden bewusst getrennt gehalten, statt sie zu vermischen
- Reine Verdrahtung ohne Logik oder Zustand, jede Zielvariable hat genau einen Schreiber
- `A2X_OUT` gibt nur wieder, was über `A2X2_BIDIR` von einem angeschlossenen bidirektionalen Teilnehmer gemeldet wird – ohne einen solchen Teilnehmer bleibt `A2X_OUT` inaktiv

Zustandsübersicht

Der Baustein ist zustandslos:

- A2X2_BIDIR.EO_UP ⟶ A2X_OUT.E_UP, A2X2_BIDIR.DO_UP ⟶ A2X_OUT.UP
- A2X2_BIDIR.EO_DOWN ⟶ A2X_OUT.E_DOWN, A2X2_BIDIR.DO_DOWN ⟶ A2X_OUT.DOWN
- A2X_IN.E_UP ⟶ A2X2_BIDIR.EI_UP, A2X_IN.UP ⟶ A2X2_BIDIR.DI_UP
- A2X_IN.E_DOWN ⟶ A2X2_BIDIR.EI_DOWN, A2X_IN.DOWN ⟶ A2X2_BIDIR.DI_DOWN

Anwendungsszenarien

- Auslesen des Status eines bidirektionalen A2X2-Teilnehmers als einfaches unidirektionales A2X-Signal, kombiniert mit dem Einspeisen eines separaten unidirektionalen Kommandosignals
- Systeme, in denen ein bidirektionaler Teilnehmer überwacht werden soll, ohne die restliche Steuerung auf A2X2 umzustellen
- Brücke zwischen einer bidirektionalen A2X2-Zone und einer unidirektionalen A2X-Zone

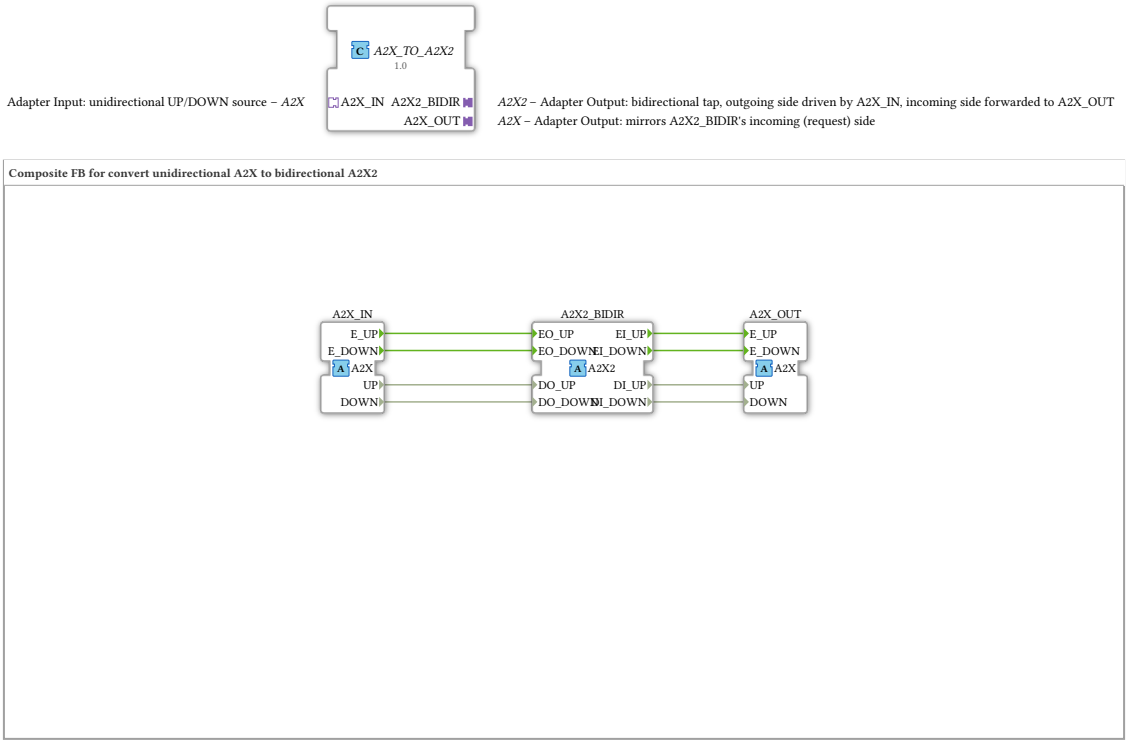
⚖ Vergleich mit ähnlichen Bausteinen

A2X_TO_A2X2 ist das Spiegelbild dieses Bausteins – dort steht `A2X2_BIDIR` als Plug auf der rechten Seite. Für eine reine, nicht abzweigende 1:1-Umsetzung zwischen den einkanaligen Adaptern siehe die Konvertierungs-Bausteine zwischen AX und AX2 in derselben Bibliothek.

Fazit

A2X2_TO_A2X ist kein einfacher Konverter, sondern ein Abzweigpunkt: Es macht den Zustand eines bidirektionalen A2X2-Teilnehmers als eigenständiges unidirektionales A2X-Signal sichtbar und speist gleichzeitig ein separates unidirektionales Signal in die bidirektionale Seite ein.

A2X_TO_A2X2



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock A2X_TO_A2X2 hängt einen bidirektionalen A2X2-Abzweig („Tap“) in eine unidirektionale A2X-Kette ein. Anders als eine einfache 1:1-Konvertierung nutzt er dafür **drei** Adapter: Ein eingehendes unidirektionales A2X-Signal wird sowohl auf die bidirektionale A2X2-Schnittstelle veröffentlicht als auch – was A2X2 auf seiner Anfrage-Seite zurückliefert – als eigenständiges unidirektionales A2X-Signal wieder ausgegeben.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Kommunikation läuft ausschließlich über die Adapter.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **A2X_IN** (Socket): unidirektionale UP/DOWN-Quelle, Typ `adapter::types::unidirectional::A2X`
- **A2X2_BIDIR** (Plug): bidirektionaler Abzweig, dessen ausgehende Seite von `A2X_IN` gespeist wird und dessen eingehende Seite an `A2X_OUT` weitergereicht wird, Typ `adapter::types::bidirectional::A2X2`
- **A2X_OUT** (Plug): spiegelt die eingehende (Anfrage-)Seite von `A2X2_BIDIR` als unidirektionales Signal, Typ `adapter::types::unidirectional::A2X`

Funktionsweise

Der Baustein arbeitet in zwei Richtungen gleichzeitig: Was am Socket `A2X_IN` eintrifft (`E_UP`/`UP`, `E_DOWN`/`DOWN`), wird direkt auf die ausgehende Seite von `A2X2_BIDIR` (`EO_UP`/`DO_UP`, `EO_DOWN`/`DO_DOWN`) veröffentlicht – A2X2_BIDIR meldet also nach außen dasselbe UP/DOWN-Signal, das über A2X_IN hereinkommt. Umgekehrt wird alles, was `A2X2_BIDIR` auf seiner Anfrage-Seite empfängt (`EI_UP`/`DI_UP`, `EI_DOWN`/`DI_DOWN` – also was ein an A2X2_BIDIR angeschlossener bidirektionaler Teilnehmer „zurücksendet“), unverändert als eigenständiges unidirektionales Signal über den Plug `A2X_OUT` ausgegeben. A2X2_BIDIR fungiert damit als Abzweigpunkt, der die unidirektionale Kette nicht unterbricht, sondern um eine bidirektionale Beobachtungs-/Einspeisemöglichkeit erweitert.

Technische Besonderheiten

- Drei statt zwei Adapter: Die eingehende und die vom Tap zurückkommende Signalrichtung werden bewusst getrennt gehalten, statt sie zu vermischen
- Reine Verdrahtung ohne Logik oder Zustand, jede Zielvariable hat genau einen Schreiber
- `A2X_OUT` gibt nur wieder, was über `A2X2_BIDIR` von einem angeschlossenen bidirektionalen Teilnehmer zurückkommt – ohne einen solchen Teilnehmer bleibt `A2X_OUT` inaktiv

Zustandsübersicht

Der Baustein ist zustandslos:

- A2X_IN.E_UP ⟶ A2X2_BIDIR.EO_UP, A2X_IN.UP ⟶ A2X2_BIDIR.DO_UP
- A2X_IN.E_DOWN ⟶ A2X2_BIDIR.EO_DOWN, A2X_IN.DOWN ⟶ A2X2_BIDIR.DO_DOWN
- A2X2_BIDIR.EI_UP ⟶ A2X_OUT.E_UP, A2X2_BIDIR.DI_UP ⟶ A2X_OUT.UP
- A2X2_BIDIR.EI_DOWN ⟶ A2X_OUT.E_DOWN, A2X2_BIDIR.DI_DOWN ⟶ A2X_OUT.DOWN

Anwendungsszenarien

- Nachträgliches Einhängen eines bidirektionalen Diagnose- oder Steuerpunkts in eine bestehende unidirektionale A2X-Signalkette, ohne diese zu unterbrechen
- Systeme, in denen ein bidirektionaler Teilnehmer (z. B. ein Bedienpanel) sowohl den Status eines unidirektionalen Signalgebers anzeigen als auch selbst Kommandos zurücksenden soll
- Brücke zwischen einer unidirektionalen A2X-Zone und einer bidirektionalen A2X2-Zone

⚖️ Vergleich mit ähnlichen Bausteinen

A2X2_TO_A2X ist das Spiegelbild dieses Bausteins – dort steht `A2X2_BIDIR` als Socket auf der linken Seite. Für eine reine, nicht abzweigende 1:1-Umsetzung zwischen den einkanaligen Adaptern siehe die Konvertierungs-Bausteine zwischen AX und AX2 in derselben Bibliothek.

Fazit

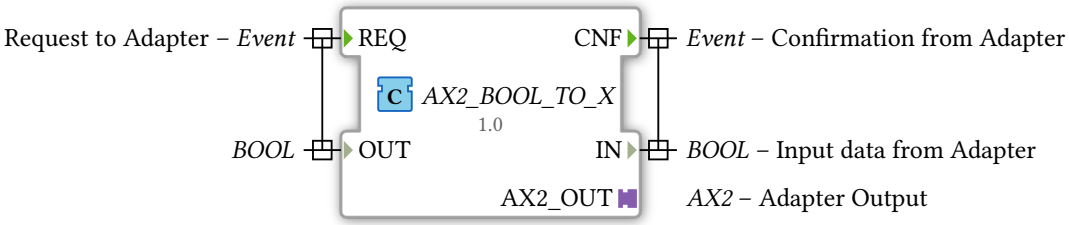
A2X_TO_A2X2 ist kein einfacher Konverter, sondern ein Abzweigpunkt: Es macht ein unidirektionales A2X-Signal an einem bidirektionalen A2X2-Punkt sichtbar, ohne die ursprüngliche unidirektionale Signalkette zu unterbrechen, und gibt gleichzeitig das dort empfangene Anfrage-Signal als eigenständiges A2X-Signal weiter.

AX2_BOOL_TO_X

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AX2_BOOL_TO_X dient als Composite-Funktionsblock zur Konvertierung von BOOL-Werten in das AX2-Adapterformat. Er ermöglicht die bidirektionale Kommunikation zwischen einfachen BOOL-Daten und dem spezifischen AX2-Adapter-Interface.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Anfrage-Ereignis zur Aktivierung des Adapters
  - Verknüpft mit Datenausgang OUT
  - Y-Position: 46

Ereignis-Ausgänge

- **CNF**: Bestätigungs-Ereignis vom Adapter
  - Verknüpft mit Dateneingang IN
  - Y-Position: 46

Daten-Eingänge

- **OUT**: BOOL-Dateneingang für die Ausgaberichtung
  - Y-Position: 31

Daten-Ausgänge

- **IN**: BOOL-Datenausgang für die Eingaberichtung
  - Empfängt Daten vom Adapter
  - Y-Position: 47

Adapter

- **AX2_OUT**: AX2-Adapter für bidirektionale Kommunikation
  - Adapter-Typ: adapter::types::bidirectional::AX2
  - Position: x=−3900, y=−500

Funktionsweise

Der Funktionsblock fungiert als Brücke zwischen BOOL-Daten und dem AX2-Adapter. Bei Eintreffen eines REQ-Ereignisses wird der OUT-Wert über den AX2_OUT-Adapter gesendet. Gleichzeitig können über denselben Adapter eingehende Daten empfangen und als IN-Ausgang mit einem CNF-Ereignis bereitgestellt werden.

Technische Besonderheiten

- Implementiert als Composite-Funktionsblock
- Unterstützt bidirektionale Datenübertragung
- Verwendet spezifische AX2-Adapter-Schnittstelle
- Automatische Ereignis- und Datenweiterleitung zwischen Hauptschnittstelle und Adapter

Zustandsübersicht

Der Funktionsblock arbeitet zustandslos und leitet Ereignisse und Daten direkt zwischen den Schnittstellen weiter:

- REQ ⟶ AX2_OUT.EO1
- AX2_OUT.EI1 ⟶ CNF
- OUT ⟶ AX2_OUT.DO1
- AX2_OUT.DI1 ⟶ IN

Anwendungsszenarien

- Integration von BOOL-Signalen in AX2-basierte Systeme
- Protokollumsetzung zwischen verschiedenen Automatisierungsebenen
- Bidirektionale Datenaustausch-Szenarien
- Adapter-basierte Systemintegration

⚖️ Vergleich mit ähnlichen Bausteinen

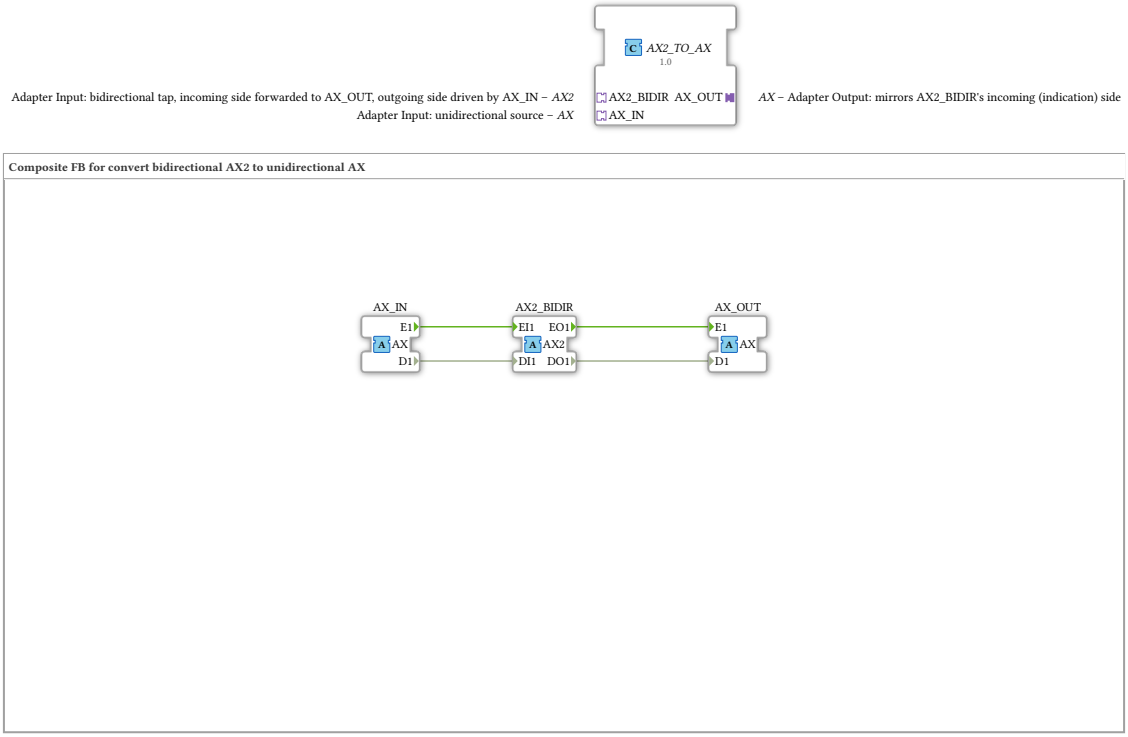
Im Vergleich zu einfachen Konvertierungsbausteinen bietet AX2_BOOL_TO_X:

- Bidirektionale Funktionalität
- Integrierte Adapter-Schnittstelle
- Komplette Ereignisbehandlung
- Höhere Abstraktionsebene durch Composite-Design

Fazit

AX2_BOOL_TO_X ist ein spezialisierter Composite-Funktionsblock, der eine effiziente Brücke zwischen BOOL-Daten und dem AX2-Adapter-Protokoll bildet. Seine bidirektionale Architektur und integrierte Ereignisbehandlung machen ihn ideal für Systeme, die eine nahtlose Integration zwischen verschiedenen Protokollschichten erfordern.

AX2_TO_AX



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock AX2_TO_AX ist das Spiegelbild von AX_TO_AX2: Hier steht der bidirektionale AX2-Abzweig als Socket auf der linken (Eingangs-)Seite, ergänzt um einen zweiten unidirektionalen AX-Socket und einen unidirektionalen AX-Plug.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Kommunikation läuft ausschließlich über die Adapter.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **AX2_BIDIR** (Socket): bidirektionaler Abzweig, dessen eingehende Seite an `AX_OUT` weitergereicht wird und dessen ausgehende Seite von `AX_IN` gespeist wird, Typ `adapter::types::bidirectional::AX2`
- **AX_IN** (Socket): unidirektionale BOOL-Quelle, Typ `adapter::types::unidirectional::AX`
- **AX_OUT** (Plug): spiegelt die eingehende (Indikations-)Seite von `AX2_BIDIR` als unidirektionales Signal, Typ `adapter::types::unidirectional::AX`

Funktionsweise

Der Baustein arbeitet in zwei Richtungen gleichzeitig: Was der Socket `AX2_BIDIR` auf seiner Indikations-Seite empfängt (`EO1`/`DO1` – also was ein an AX2_BIDIR angeschlossener bidirektionaler Teilnehmer meldet), wird unverändert als eigenständiges unidirektionales Signal über den Plug `AX_OUT` ausgegeben.

Umgekehrt wird alles, was am Socket `AX_IN` eintrifft (`E1`/`D1`), auf die Anfrage-Seite von `AX2_BIDIR` (`EI1`/`DI1`) veröffentlicht – AX2_BIDIR sendet also dasselbe Signal an den angeschlossenen bidirektionalen Teilnehmer weiter.

Technische Besonderheiten

- Drei statt zwei Adapter: Die vom Tap empfangene und die separat eingespeiste Signalrichtung werden bewusst getrennt gehalten, statt sie zu vermischen
- Reine Verdrahtung ohne Logik oder Zustand, jede Zielvariable hat genau einen Schreiber
- `AX_OUT` gibt nur wieder, was über `AX2_BIDIR` von einem angeschlossenen bidirektionalen Teilnehmer gemeldet wird – ohne einen solchen Teilnehmer bleibt `AX_OUT` inaktiv

Zustandsübersicht

Der Baustein ist zustandslos:

- AX2_BIDIR.EO1 ⟶ AX_OUT.E1, AX2_BIDIR.DO1 ⟶ AX_OUT.D1
- AX_IN.E1 ⟶ AX2_BIDIR.EI1, AX_IN.D1 ⟶ AX2_BIDIR.DI1

Anwendungsszenarien

- Auslesen des Status eines bidirektionalen AX2-Teilnehmers als einfaches unidirektionales AX-Signal, kombiniert mit dem Einspeisen eines separaten unidirektionalen Kommandosignals
- Systeme, in denen ein bidirektionaler Teilnehmer überwacht werden soll, ohne die restliche Steuerung auf AX2 umzustellen
- Brücke zwischen einer bidirektionalen AX2-Zone und einer unidirektionalen AX-Zone

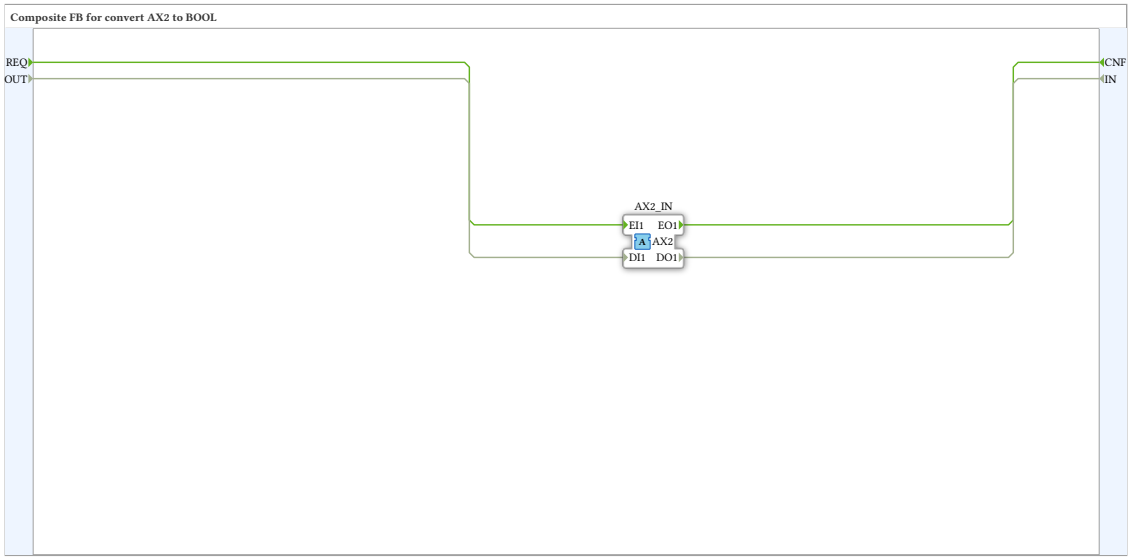
⚖️ Vergleich mit ähnlichen Bausteinen

AX_TO_AX2 ist das Spiegelbild dieses Bausteins – dort steht `AX2_BIDIR` als Plug auf der rechten Seite. Für eine reine, nicht abzweigende 1:1-Umsetzung zwischen AX2 und einem plain BOOL-Signal siehe AX2_X_TO_BOOL/AX2_BOOL_TO_X. Das Gegenstück für den 2-kanaligen A2X2/A2X ist A2X2_TO_A2X.

Fazit

AX2_TO_AX ist kein einfacher Konverter, sondern ein Abzweigpunkt: Es macht den Zustand eines bidirektionalen AX2-Teilnehmers als eigenständiges unidirektionales AX-Signal sichtbar und speist gleichzeitig ein separates unidirektionales Signal in die bidirektionale Seite ein.

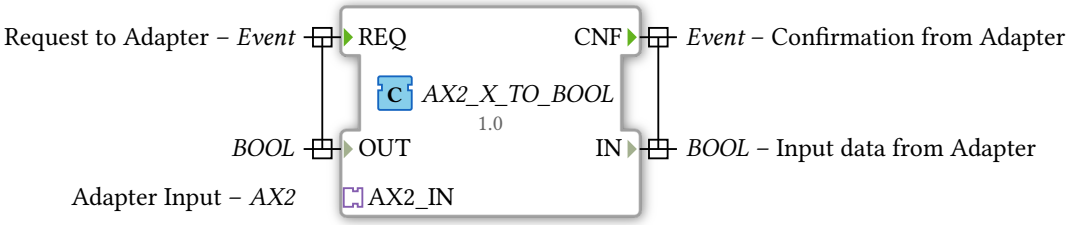
AX2_X_TO_BOOL



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock AX2_X_TO_BOOL ist ein Composite-Funktionsblock zur Konvertierung von AX2-Daten in BOOL-Werte. Er dient als bidirektionaler Adapter für die Datentypumwandlung zwischen verschiedenen Schnittstellen in Automatisierungssystemen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Request to Adapter): Startet die Konvertierungsanfrage an den Adapter

Ereignis-Ausgänge

- **CNF** (Confirmation from Adapter): Bestätigt den Abschluss der Konvertierung vom Adapter

Daten-Eingänge

- **OUT** (BOOL): Ausgabedaten für die Übergabe an den Adapter

Daten-Ausgänge

- **IN** (BOOL): Eingabedaten vom Adapter nach erfolgter Konvertierung

Adapter

- **AX2_IN** (adapter::types::bidirectional::AX2): Bidirektionaler AX2-Adapter für die Datentypkonvertierung

Funktionsweise

Der Funktionsblock empfängt über das REQ-Ereignis eine Konvertierungsanfrage und leitet diese an den AX2-Adapter weiter. Der Adapter führt die Umwandlung der AX2-Daten in BOOL-Werte durch. Nach erfolgreicher Konvertierung wird das CNF-Ereignis mit den konvertierten Daten am IN-Ausgang ausgegeben. Gleichzeitig können BOOL-Daten über den OUT-Eingang an den Adapter zurückgegeben werden.

Technische Besonderheiten

- Bidirektionale Datenübertragung zwischen AX2- und BOOL-Datentypen
- Composite-Funktionsblock mit integriertem Adapter
- Automatische Ereignisweiterleitung zwischen Schnittstellen
- Positionierte Elemente für optimierte Darstellung in der Entwicklungsumgebung

Zustandsübergänge

1. **Wartezustand**: Keine aktive Konvertierung
2. **Aktiver Zustand**: Bei REQ-Ereignis - Daten werden an Adapter weitergeleitet
3. **Bestätigungszustand**: Bei Abschluss der Konvertierung - CNF-Ereignis wird ausgelöst

Anwendungsszenarien

- Integration von AX2-Sensordaten in BOOL-basierte Steuerungssysteme
- Datentypkonvertierung in gemischten Automatisierungsumgebungen
- Bidirektionale Kommunikation zwischen verschiedenen Protokollen
- Adapterlösungen für bestehende Systeme mit unterschiedlichen Datentypen

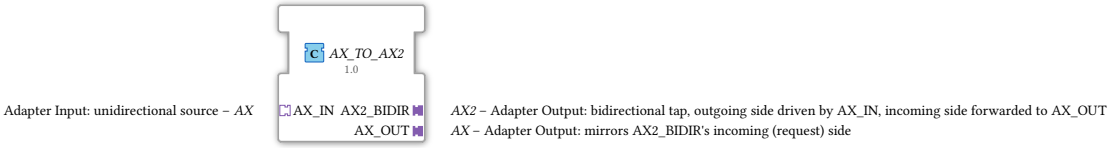
⚖️ Vergleich mit ähnlichen Bausteinen

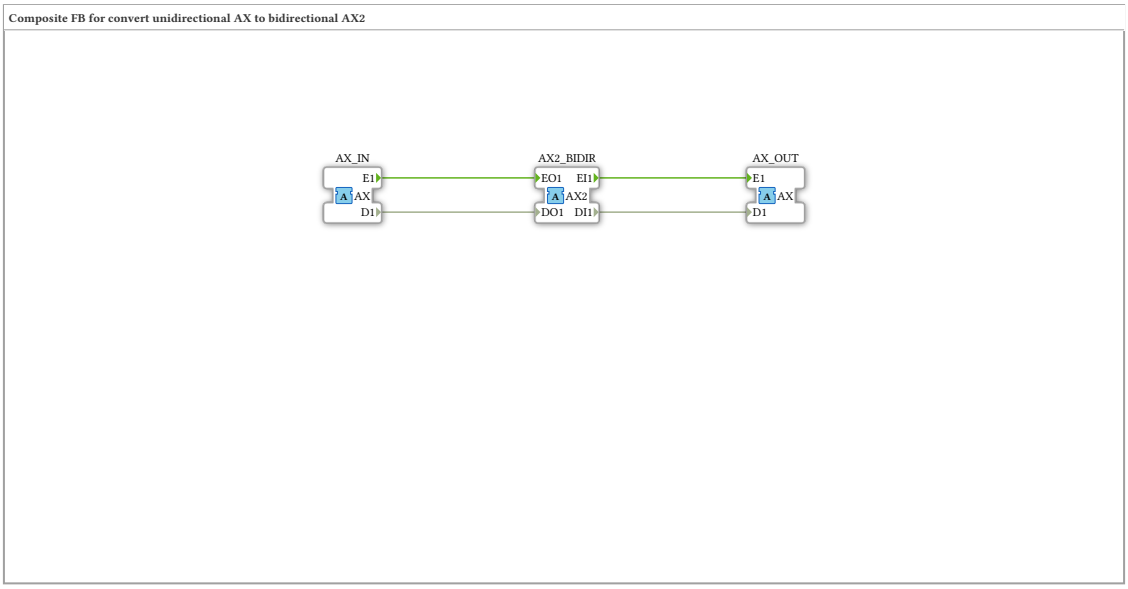
Im Vergleich zu einfachen Konvertierungsblöcken bietet AX2_X_TO_BOOL eine bidirektionale Funktionalität mit integriertem Adapter-Interface. Während einfache Konverter typischerweise nur in eine Richtung arbeiten, ermöglicht dieser Block den Datenaustausch in beide Richtungen.

Fazit

Der AX2_X_TO_BOOL Funktionsblock stellt eine robuste Lösung für die bidirektionale Konvertierung zwischen AX2- und BOOL-Datentypen dar. Durch die integrierte Adapter-Architektur und die klare Ereignissteuerung eignet er sich besonders für komplexe Automatisierungsanwendungen mit gemischten Datentypumgebungen.

AX_TO_AX2





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock AX_TO_AX2 hängt einen bidirektionalen AX2-Abzweig („Tap“) in eine unidirektionale AX-Kette ein. Anders als eine einfache 1:1-Konvertierung nutzt er dafür **drei** Adapter: Ein eingehendes unidirektionales AX-Signal wird sowohl auf die bidirektionale AX2-Schnittstelle veröffentlicht als auch – was AX2 auf seiner Anfrage-Seite zurückliefert – als eigenständiges unidirektionales AX-Signal wieder ausgegeben.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Kommunikation läuft ausschließlich über die Adapter.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **AX_IN** (Socket): unidirektionale BOOL-Quelle, Typ `adapter::types::unidirectional::AX`

- **AX2_BIDIR** (Plug): bidirektionaler Abzweig, dessen ausgehende Seite von `AX_IN` gespeist wird und dessen eingehende Seite an `AX_OUT` weitergereicht wird, Typ `adapter::types::bidirectional::AX2`
- **AX_OUT** (Plug): spiegelt die eingehende (Anfrage-)Seite von `AX2_BIDIR` als unidirektionales Signal, Typ `adapter::types::unidirectional::AX`

Funktionsweise

Der Baustein arbeitet in zwei Richtungen gleichzeitig: Was am Socket `AX_IN` eintrifft (`E1`/`D1`), wird direkt auf die ausgehende Seite von `AX2_BIDIR` (`EO1`/`DO1`) veröffentlicht – AX2_BIDIR meldet also nach außen dasselbe Signal, das über AX_IN hereinkommt. Umgekehrt wird alles, was `AX2_BIDIR` auf seiner Anfrage-Seite empfängt (`EI1`/`DI1` – also was ein an AX2_BIDIR angeschlossener bidirektionaler Teilnehmer „zurücksendet"), unverändert als eigenständiges unidirektionales Signal über den Plug `AX_OUT` ausgegeben. AX2_BIDIR fungiert damit als Abzweigpunkt, der die unidirektionale Kette nicht unterbricht, sondern um eine bidirektionale Beobachtungs-/Einspeisemöglichkeit erweitert.

Technische Besonderheiten

- Drei statt zwei Adapter: Die eingehende und die vom Tap zurückkommende Signalrichtung werden bewusst getrennt gehalten, statt sie zu vermischen
- Reine Verdrahtung ohne Logik oder Zustand, jede Zielvariable hat genau einen Schreiber
- `AX_OUT` gibt nur wieder, was über `AX2_BIDIR` von einem angeschlossenen bidirektionalen Teilnehmer zurückkommt – ohne einen solchen Teilnehmer bleibt `AX_OUT` inaktiv

Zustandsübersicht

Der Baustein ist zustandslos:

- AX_IN.E1 ⟶ AX2_BIDIR.EO1, AX_IN.D1 ⟶ AX2_BIDIR.DO1
- AX2_BIDIR.EI1 ⟶ AX_OUT.E1, AX2_BIDIR.DI1 ⟶ AX_OUT.D1

Anwendungsszenarien

- Nachträgliches Einhängen eines bidirektionalen Diagnose- oder Steuerpunkts in eine bestehende unidirektionale AX-Signalkette, ohne diese zu unterbrechen
- Systeme, in denen ein bidirektionaler Teilnehmer (z. B. ein Bedienpanel) sowohl den Status eines unidirektionalen Signalgebers anzeigen als auch selbst Kommandos zurücksenden soll
- Brücke zwischen einer unidirektionalen AX-Zone und einer bidirektionalen AX2-Zone

⚖️ Vergleich mit ähnlichen Bausteinen

AX2_TO_AX ist das Spiegelbild dieses Bausteins – dort steht `AX2_BIDIR` als Socket auf der linken Seite. Für eine reine, nicht abzweigende 1:1-Umsetzung zwischen einem plain BOOL-Signal und AX2 siehe AX2_BOOL_TO_X/AX2_X_TO_BOOL. Das Gegenstück für den 2-kanaligen A2X/A2X2 ist A2X_TO_A2X2.

Fazit

AX_TO_AX2 ist kein einfacher Konverter, sondern ein Abzweigpunkt: Es macht ein unidirektionales AX-Signal an einem bidirektionalen AX2-Punkt sichtbar, ohne die ursprüngliche unidirektionale Signalkette zu unterbrechen, und gibt gleichzeitig das dort empfangene Anfrage-Signal als eigenständiges AX-Signal weiter.

BOOL

🎧 Podcast

- George Boole: Wie ein Autodidakt mit 0 und 1 das Fundament unserer digitalen Welt legte

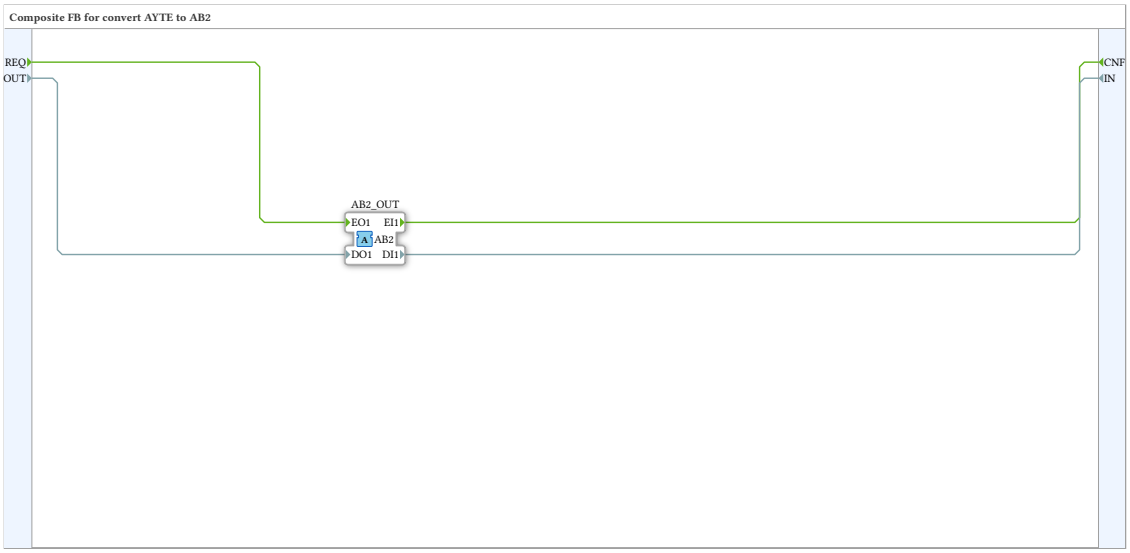
🛠️ Zugehörige Übungen

- Uebung_001_AX_b
- Uebung_002a2_AX
- Uebung_002a4
- Uebung_002a5b
- Uebung_002b3
- Uebung_004b_AX
- Uebung_006a3

- LinksRechts_T_FF_Event
- Uebung_035a2
- Uebung_035a3
- SoftkeyOverride_TO_QX_BG
- Uebung_055
- Uebung_056
- Bool8_TO_logiBUS_QX8
- Uebung_094a_AX
- Uebung_160

BYTE

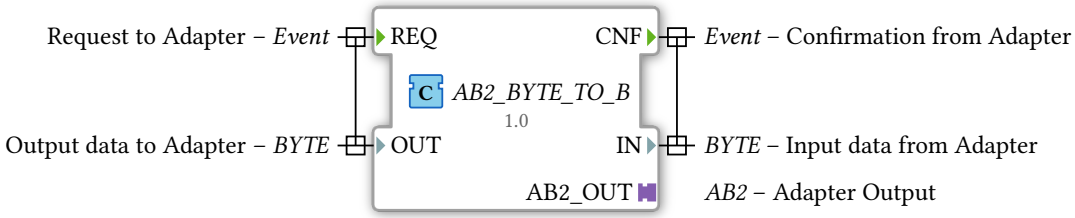
AB2_BYTE_TO_B



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AB2_BYTE_TO_B ist ein Composite-Funktionsblock, der der Konvertierung zwischen BYTE-Datentypen und dem AB2-Adapter dient. Er fungiert als bidirektionaler Konverter und ermöglicht die Kommunikation zwischen Systemkomponenten mit unterschiedlichen Schnittstellen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Request to Adapter): Startet die Konvertierung und Datenübertragung zum Adapter

Ereignis-Ausgänge

- **CNF** (Confirmation from Adapter): Signalisiert den erfolgreichen Abschluss der Konvertierung und Datenübernahme vom Adapter

Daten-Eingänge

- **OUT** (BYTE): Ausgabedaten, die zum Adapter übertragen werden sollen

Daten-Ausgänge

- **IN** (BYTE): Eingabedaten, die vom Adapter empfangen wurden

Adapter

- **AB2_OUT** (adapter::types::bidirectional::AB2): Bidirektionaler Adapter für die Datenkonvertierung

Funktionsweise

Der Funktionsblock empfängt über das REQ-Ereignis eine Konvertierungsanfrage und leitet die anliegenden BYTE-Daten (OUT) an den AB2-Adapter weiter. Der Adapter führt die eigentliche Konvertierung durch und sendet bei erfolgreichem Abschluss ein CNF-Ereignis zurück, zusammen mit den konvertierten BYTE-Daten (IN).

Technische Besonderheiten

- Composite-Funktionsblock mit integriertem Adapter
- Bidirektionale Datenübertragung
- Verwendet BYTE als primären Datentyp
- Positionierte Schnittstellen für optimierte Darstellung in der Entwicklungsumgebung

Zustandsübergänge

1. **Idle-Zustand**: Wartet auf REQ-Ereignis
2. **Verarbeitungszustand**: Bei REQ werden Daten an AB2_OUT weitergeleitet
3. **Bestätigungszustand**: Bei Rückmeldung vom Adapter wird CNF ausgelöst

Anwendungsszenarien

- Integration von BYTE-basierten Komponenten in AB2-kompatible Systeme
- Protokollkonvertierung in verteilten Automatisierungssystemen
- Bidirektionale Datenaustausch-Schnittstellen
- Adapter-basierte Systemintegration

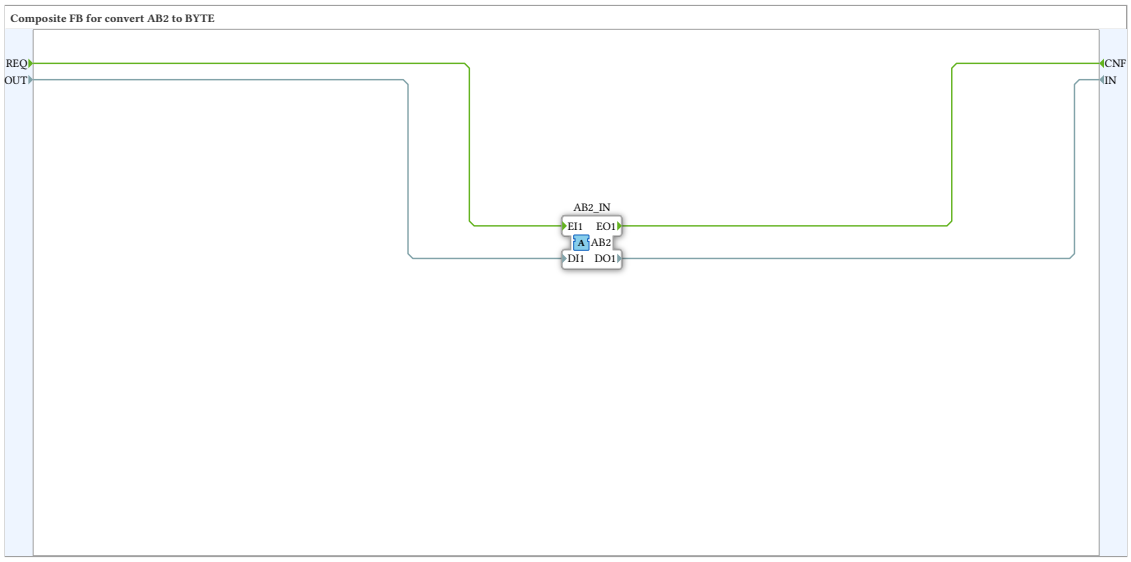
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen Konvertierungsblöcken bietet AB2_BYTE_TO_B eine bidirektionale Funktionalität mit Adapter-Integration. Während einfache Konverter typischerweise nur in eine Richtung arbeiten, ermöglicht dieser Block die Kommunikation in beide Richtungen über standardisierte Adapterschnittstellen.

Fazit

AB2_BYTE_TO_B ist ein spezialisierter Composite-Funktionsblock, der eine zuverlässige bidirektionale Konvertierung zwischen BYTE-Daten und dem AB2-Adapter-Standard bereitstellt. Seine strukturierte Schnittstelle und klare Ereignissteuerung machen ihn ideal für den Einsatz in komplexen Automatisierungssystemen mit standardisierten Kommunikationsprotokollen.

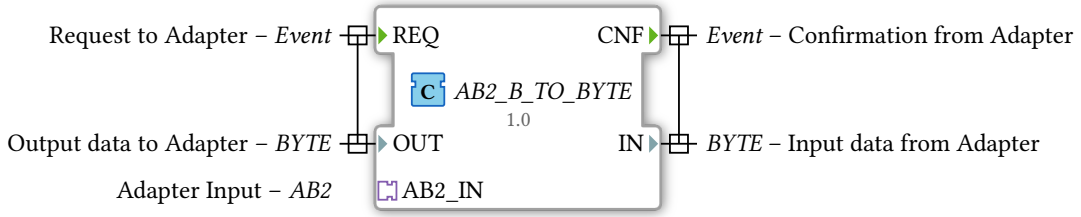
AB2_B_TO_BYTE



• – ○ • – ○ • – ○ •

Einleitung

Der AB2_B_TO_BYTE ist ein Composite-Funktionsblock, der zur bidirektionalen Konvertierung zwischen AB2- und BYTE-Datentypen dient. Der Baustein fungiert als Adapter-Konverter und ermöglicht die nahtlose Integration unterschiedlicher Datentypen in Automatisierungssystemen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Request to Adapter): Startet die Konvertierungsoperation und fordert die Datenausgabe an

Ereignis-Ausgänge

- **CNF** (Confirmation from Adapter): Signalisiert den erfolgreichen Abschluss der Konvertierungsoperation

Daten-Eingänge

- **OUT** (BYTE): Ausgabedaten, die an den Adapter gesendet werden sollen

Daten-Ausgänge

- **IN** (BYTE): Eingabedaten, die vom Adapter empfangen wurden

Adapter

- **AB2_IN** (adapter::types::bidirectional::AB2): Bidirektionaler AB2-Adapter für die Datentyp-Konvertierung

Funktionsweise

Der Funktionsblock arbeitet als bidirektionaler Konverter zwischen AB2- und BYTE-Datentypen. Bei Empfang eines REQ-Ereignisses werden die an OUT anliegenden BYTE-Daten über den AB2_IN-Adapter konvertiert. Nach erfolgreicher Konvertierung wird das CNF-Ereignis ausgelöst und die konvertierten Daten an IN ausgegeben.

Die interne Vernetzung zeigt folgende Signalflüsse:

- REQ → AB2_IN.EI1 (Adapter-Eingangsereignis)
- AB2_IN.EO1 → CNF (Adapter-Ausgangsereignis)
- OUT → AB2_IN.DI1 (Adapter-Dateneingang)
- AB2_IN.DO1 → IN (Adapter-Datenausgang)

Technische Besonderheiten

- Implementiert als Composite-Funktionsblock mit integriertem Adapter
- Unterstützt bidirektionale Datenkonvertierung
- Positionierungsattribute für grafische Darstellung in der Entwicklungsumgebung
- Eingebettet im Package „adapter::conversion::bidirectional“

Zustandsübersicht

Der Funktionsblock durchläuft folgende Zustände:

1. **Bereit**: Wartet auf REQ-Ereignis
2. **Konvertierung aktiv**: Verarbeitet Daten über AB2-Adapter
3. **Bestätigung**: Sendet CNF bei erfolgreichem Abschluss

Anwendungsszenarien

- Integration von AB2-Komponenten in BYTE-basierte Steuerungssysteme
- Datentyp-Konvertierung in heterogenen Automatisierungsnetzwerken
- Adapter-Lösungen für Legacy-Systeme mit AB2-Schnittstellen
- Protokollumsetzung zwischen verschiedenen Kommunikationsstandards

⚖️ Vergleich mit ähnlichen Bausteinen

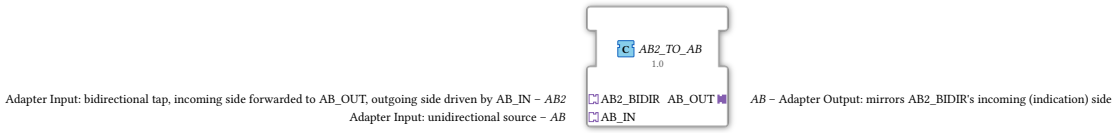
Im Vergleich zu einfachen Konvertierungsbausteinen bietet AB2_B_TO_BYTE:

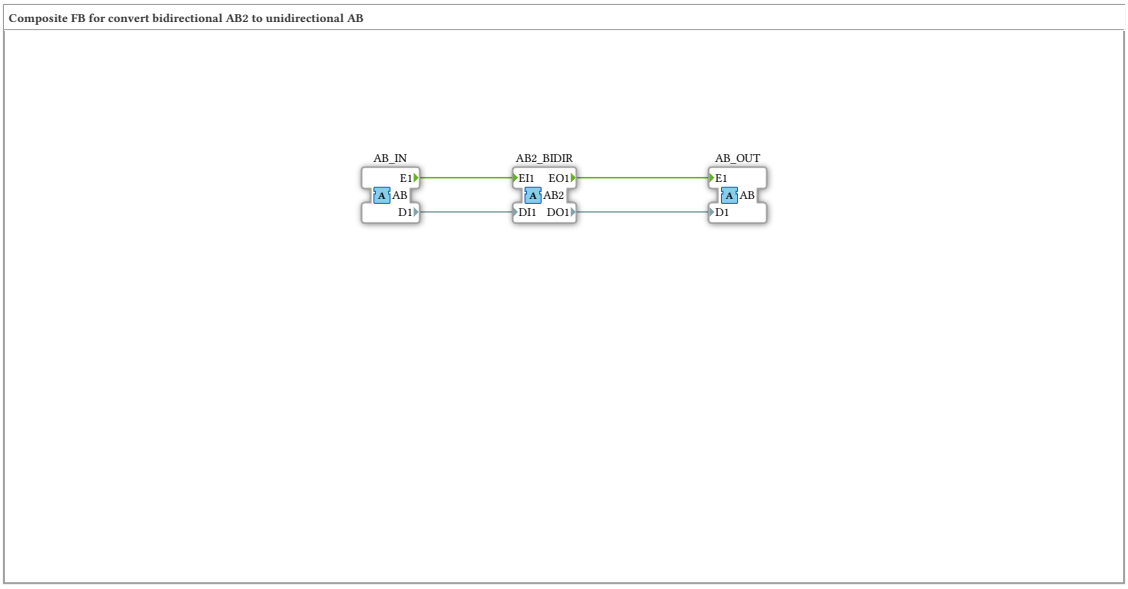
- Bidirektionale Funktionalität in einem einzigen Baustein
- Integrierte Adapter-Schnittstelle für AB2-Typen
- Composite-Architektur mit erweiterter Funktionalität
- Standardisierte Ereignisbehandlung mit REQ/CNF-Paar

Fazit

Der AB2_B_TO_BYTE Funktionsblock stellt eine robuste Lösung für die bidirektionale Konvertierung zwischen AB2- und BYTE-Datentypen dar. Seine Composite-Architektur mit integriertem Adapter ermöglicht eine einfache Integration in bestehende Automatisierungssysteme und bietet zuverlässige Datentyp-Kompatibilität.

AB2_TO_AB





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock AB2_TO_AB ist das Spiegelbild von AB_TO_AB2: Hier steht der bidirektionale AB2-Abzweig als Socket auf der linken (Eingangs-)Seite, ergänzt um einen zweiten unidirektionalen AB-Socket und einen unidirektionalen AB-Plug.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Kommunikation läuft ausschließlich über die Adapter.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **AB2_BIDIR** (Socket): bidirektionaler Abzweig, dessen eingehende Seite an `AB_OUT` weitergereicht wird und dessen ausgehende Seite von `AB_IN` gespeist wird, Typ `adapter::types::bidirectional::AB2`
- **AB_IN** (Socket): unidirektionale BYTE-Quelle, Typ `adapter::types::unidirectional::AB`
- **AB_OUT** (Plug): spiegelt die eingehende (Indikations-)Seite von `AB2_BIDIR` als unidirektionales Signal, Typ `adapter::types::unidirectional::AB`

Funktionsweise

Der Baustein arbeitet in zwei Richtungen gleichzeitig: Was der Socket `AB2_BIDIR` auf seiner Indikations-Seite empfängt (`EO1`/`DO1` – also was ein an AB2_BIDIR angeschlossener bidirektionaler Teilnehmer meldet), wird unverändert als eigenständiges unidirektionales Signal über den Plug `AB_OUT` ausgegeben. Umgekehrt wird alles, was am Socket `AB_IN` eintrifft (`E1`/`D1`), auf die Anfrage-Seite von `AB2_BIDIR` (`EI1`/`DI1`) veröffentlicht – AB2_BIDIR sendet also dasselbe Signal an den angeschlossenen bidirektionalen Teilnehmer weiter.

Technische Besonderheiten

- Drei statt zwei Adapter: Die vom Tap empfangene und die separat eingespeiste Signalrichtung werden bewusst getrennt gehalten, statt sie zu vermischen
- Reine Verdrahtung ohne Logik oder Zustand, jede Zielvariable hat genau einen Schreiber
- `AB_OUT` gibt nur wieder, was über `AB2_BIDIR` von einem angeschlossenen bidirektionalen Teilnehmer gemeldet wird – ohne einen solchen Teilnehmer bleibt `AB_OUT` inaktiv

Zustandsübersicht

Der Baustein ist zustandslos:

- AB2_BIDIR.EO1 ⟶ AB_OUT.E1, AB2_BIDIR.DO1 ⟶ AB_OUT.D1
- AB_IN.E1 ⟶ AB2_BIDIR.EI1, AB_IN.D1 ⟶ AB2_BIDIR.DI1

Anwendungsszenarien

- Auslesen des Status eines bidirektionalen AB2-Teilnehmers als einfaches unidirektionales AB-Signal, kombiniert mit dem Einspeisen eines separaten unidirektionalen Kommandosignals
- Systeme, in denen ein bidirektionaler Teilnehmer überwacht werden soll, ohne die restliche Steuerung auf AB2 umzustellen
- Brücke zwischen einer bidirektionalen AB2-Zone und einer unidirektionalen AB-Zone

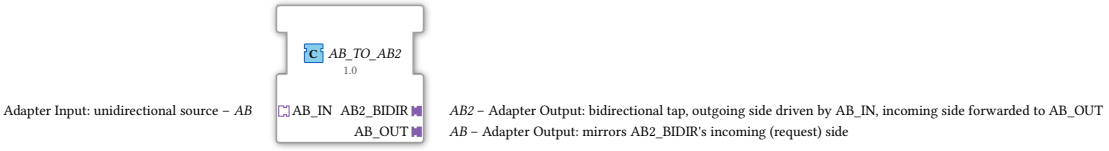
⚖️ Vergleich mit ähnlichen Bausteinen

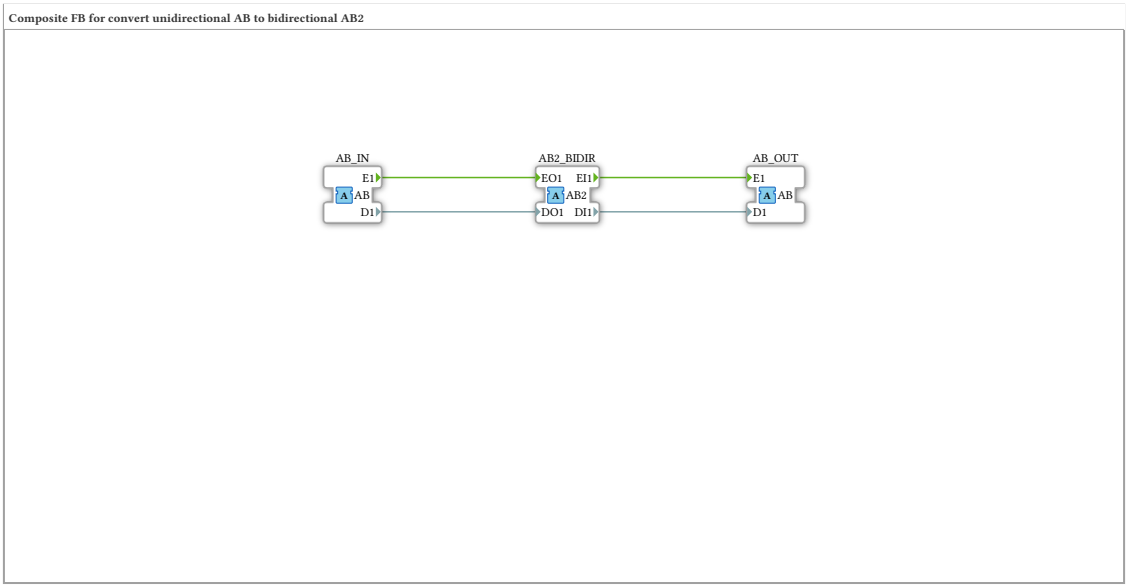
AB_TO_AB2 ist das Spiegelbild dieses Bausteins – dort steht `AB2_BIDIR` als Plug auf der rechten Seite. Für eine reine, nicht abzweigende 1:1-Umsetzung zwischen AB2 und einem plain BYTE-Signal siehe AB2_BYTE_TO_B/AB2_B_TO_BYTE.

Fazit

AB2_TO_AB ist kein einfacher Konverter, sondern ein Abzweigpunkt: Es macht den Zustand eines bidirektionalen AB2-Teilnehmers als eigenständiges unidirektionales AB-Signal sichtbar und speist gleichzeitig ein separates unidirektionales Signal in die bidirektionale Seite ein.

AB_TO_AB2





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock AB_TO_AB2 hängt einen bidirektionalen AB2-Abzweig („Tap“) in eine unidirektionale AB-Kette ein. Anders als eine einfache 1:1-Konvertierung nutzt er dafür **drei** Adapter: Ein eingehendes unidirektionales AB-Signal wird sowohl auf die bidirektionale AB2-Schnittstelle veröffentlicht als auch – was AB2 auf seiner Anfrage-Seite zurückliefert – als eigenständiges unidirektionales AB-Signal wieder ausgegeben.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Kommunikation läuft ausschließlich über die Adapter.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **AB_IN** (Socket): unidirektionale BYTE-Quelle, Typ `adapter::types::unidirectional::AB`
- **AB2_BIDIR** (Plug): bidirektionaler Abzweig, dessen ausgehende Seite von `AB_IN` gespeist wird und dessen eingehende Seite an `AB_OUT` weitergereicht wird, Typ `adapter::types::bidirectional::AB2`
- **AB_OUT** (Plug): spiegelt die eingehende (Anfrage-)Seite von `AB2_BIDIR` als unidirektionales Signal, Typ `adapter::types::unidirectional::AB`

Funktionsweise

Der Baustein arbeitet in zwei Richtungen gleichzeitig: Was am Socket `AB_IN` eintrifft (`E1`/`D1`), wird direkt auf die ausgehende Seite von `AB2_BIDIR` (`EO1`/`DO1`) veröffentlicht – AB2_BIDIR meldet also nach außen dasselbe Signal, das über AB_IN hereinkommt. Umgekehrt wird alles, was `AB2_BIDIR` auf seiner Anfrage-Seite empfängt (`EI1`/`DI1` – also was ein an AB2_BIDIR angeschlossener bidirektionaler Teilnehmer „zurücksendet“), unverändert als eigenständiges unidirektionales Signal über den Plug `AB_OUT` ausgegeben. AB2_BIDIR fungiert damit als Abzweigpunkt, der die unidirektionale Kette nicht unterbricht, sondern um eine bidirektionale Beobachtungs-/Einspeisemöglichkeit erweitert.

Technische Besonderheiten

- Drei statt zwei Adapter: Die eingehende und die vom Tap zurückkommende Signalrichtung werden bewusst getrennt gehalten, statt sie zu vermischen
- Reine Verdrahtung ohne Logik oder Zustand, jede Zielvariable hat genau einen Schreiber
- `AB_OUT` gibt nur wieder, was über `AB2_BIDIR` von einem angeschlossenen bidirektionalen Teilnehmer zurückkommt – ohne einen solchen Teilnehmer bleibt `AB_OUT` inaktiv

Zustandsübersicht

Der Baustein ist zustandslos:

- AB_IN.E1 ⟶ AB2_BIDIR.EO1, AB_IN.D1 ⟶ AB2_BIDIR.DO1
- AB2_BIDIR.EI1 ⟶ AB_OUT.E1, AB2_BIDIR.DI1 ⟶ AB_OUT.D1

Anwendungsszenarien

- Nachträgliches Einhängen eines bidirektionalen Diagnose- oder Steuerpunkts in eine bestehende unidirektionale AB-Signalkette, ohne diese zu unterbrechen
- Systeme, in denen ein bidirektionaler Teilnehmer (z. B. ein Bedienpanel) sowohl den Status eines unidirektionalen Byte-Signalgebers anzeigen als auch selbst Kommandos zurücksenden soll
- Brücke zwischen einer unidirektionalen AB-Zone und einer bidirektionalen AB2-Zone

⚖️ Vergleich mit ähnlichen Bausteinen

AB2_TO_AB ist das Spiegelbild dieses Bausteins – dort steht `AB2_BIDIR` als Socket auf der linken Seite. Für eine reine, nicht abzweigende 1:1-Umsetzung zwischen einem plain BYTE-Signal und AB2 siehe AB2_B_TO_BYTE/AB2_BYTE_TO_B.

Fazit

AB_TO_AB2 ist kein einfacher Konverter, sondern ein Abzweigpunkt: Es macht ein unidirektionales AB-Signal an einem bidirektionalen AB2-Punkt sichtbar, ohne die ursprüngliche unidirektionale Signalkette zu unterbrechen, und gibt gleichzeitig das dort empfangene Anfrage-Signal als eigenständiges AB-Signal weiter.

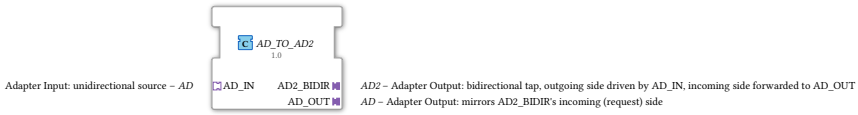
BYTE

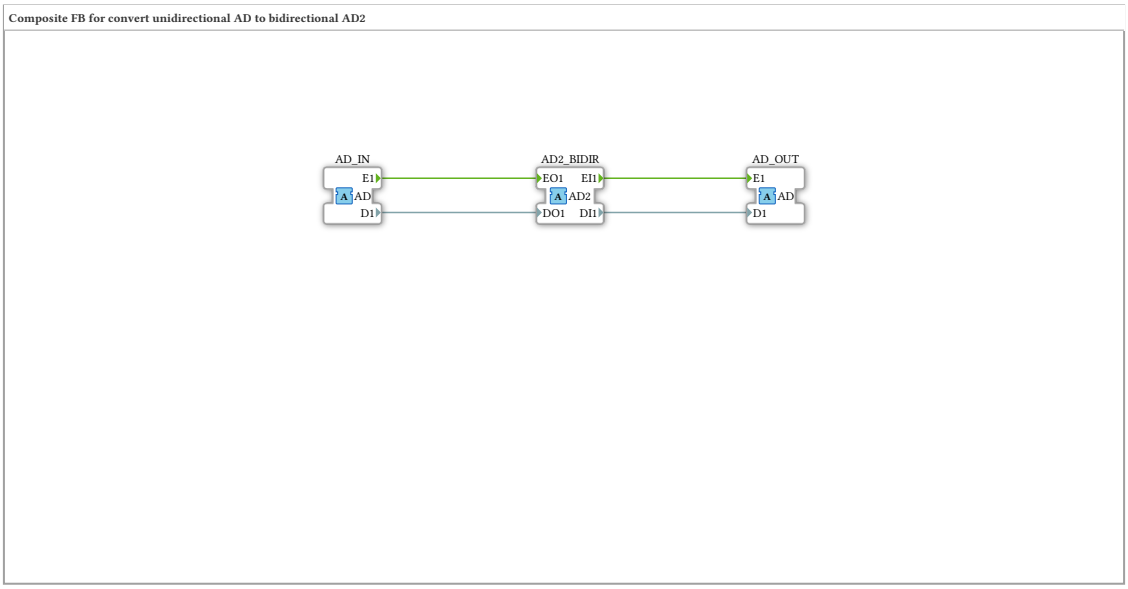
🛠️ Zugehörige Übungen

- SIN_TO_PCAN_Callback_Byte
- Uebung_128b
- Uebung_12x_sub

DWORD

AD_TO_AD2





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock AD_TO_AD2 hängt einen bidirektionalen AD2-Abzweig („Tap“) in eine unidirektionale AD-Kette ein. Anders als eine einfache 1:1-Konvertierung nutzt er dafür **drei** Adapter: Ein eingehendes unidirektionales AD-Signal wird sowohl auf die bidirektionale AD2-Schnittstelle veröffentlicht als auch – was AD2 auf seiner Anfrage-Seite zurückliefert – als eigenständiges unidirektionales AD-Signal wieder ausgegeben.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Kommunikation läuft ausschließlich über die Adapter.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **AD_IN** (Socket): unidirektionale DWORD-Quelle, Typ `adapter::types::unidirectional::AD`
- **AD2_BIDIR** (Plug): bidirektionaler Abzweig, dessen ausgehende Seite von `AD_IN` gespeist wird und dessen eingehende Seite an `AD_OUT` weitergereicht wird, Typ `adapter::types::bidirectional::AD2`
- **AD_OUT** (Plug): spiegelt die eingehende (Anfrage-)Seite von `AD2_BIDIR` als unidirektionales Signal, Typ `adapter::types::unidirectional::AD`

Funktionsweise

Der Baustein arbeitet in zwei Richtungen gleichzeitig: Was am Socket `AD_IN` eintrifft (`E1`/`D1`), wird direkt auf die ausgehende Seite von `AD2_BIDIR` (`EO1`/`DO1`) veröffentlicht – AD2_BIDIR meldet also nach außen dasselbe Signal, das über AD_IN hereinkommt. Umgekehrt wird alles, was `AD2_BIDIR` auf seiner Anfrage-Seite empfängt (`EI1`/`DI1` – also was ein an AD2_BIDIR angeschlossener bidirektionaler Teilnehmer „zurücksendet“), unverändert als eigenständiges unidirektionales Signal über den Plug `AD_OUT` ausgegeben. AD2_BIDIR fungiert damit als Abzweigpunkt, der die unidirektionale Kette nicht unterbricht, sondern um eine bidirektionale Beobachtungs-/Einspeisemöglichkeit erweitert.

Technische Besonderheiten

- Drei statt zwei Adapter: Die eingehende und die vom Tap zurückkommende Signalrichtung werden bewusst getrennt gehalten, statt sie zu vermischen
- Reine Verdrahtung ohne Logik oder Zustand, jede Zielvariable hat genau einen Schreiber
- `AD_OUT` gibt nur wieder, was über `AD2_BIDIR` von einem angeschlossenen bidirektionalen Teilnehmer zurückkommt – ohne einen solchen Teilnehmer bleibt `AD_OUT` inaktiv

Zustandsübersicht

Der Baustein ist zustandslos:

- AD_IN.E1 ⟶ AD2_BIDIR.EO1, AD_IN.D1 ⟶ AD2_BIDIR.DO1
- AD2_BIDIR.EI1 ⟶ AD_OUT.E1, AD2_BIDIR.DI1 ⟶ AD_OUT.D1

Anwendungsszenarien

- Nachträgliches Einhängen eines bidirektionalen Diagnose- oder Steuerpunkts in eine bestehende unidirektionale AD-Signalkette, ohne diese zu unterbrechen
- Systeme, in denen ein bidirektionaler Teilnehmer (z. B. ein Bedienpanel) sowohl den Status eines unidirektionalen Byte-Signalgebers anzeigen als auch selbst Kommandos zurücksenden soll
- Brücke zwischen einer unidirektionalen AD-Zone und einer bidirektionalen AD2-Zone

⚖️ Vergleich mit ähnlichen Bausteinen

AD_TO_AD2 gehört zur Konvertierungsfamilie der Abzweig-Bausteine (analog zu AB_TO_AB2 für BYTE): ein unidirektionales AD-Signal wird an einem bidirektionalen AD2-Punkt abgegriffen.

Fazit

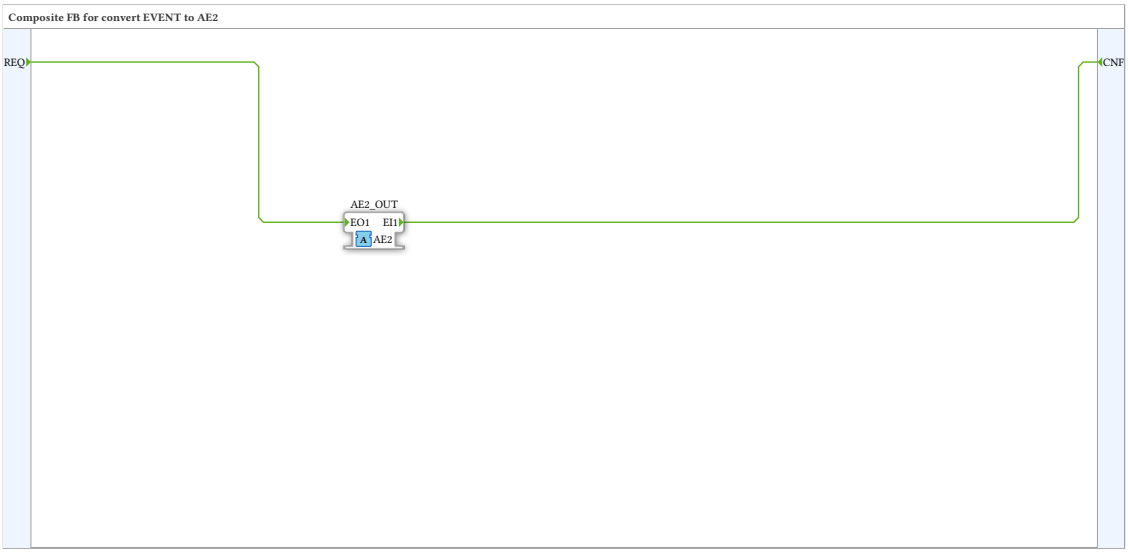
AD_TO_AD2 ist kein einfacher Konverter, sondern ein Abzweigpunkt: Es macht ein unidirektionales AD-Signal an einem bidirektionalen AD2-Punkt sichtbar, ohne die ursprüngliche unidirektionale Signalkette zu unterbrechen, und gibt gleichzeitig das dort empfangene Anfrage-Signal als eigenständiges AD-Signal weiter.

🛠️ Zugehörige Übungen

- Uebung_094b5_AX

EVENT

AE2_EVENT_TO_E



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AE2_EVENT_TO_E ist ein Composite-Funktionsblock, der zur Konvertierung von EVENT-Signalen in das AE2-Adapterformat dient. Er ermöglicht die bidirektionale Kommunikation zwischen Standard-Ereignissen und dem spezifischen AE2-Adapter.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Request to Adapter): Startet die Konvertierung und sendet eine Anfrage an den AE2-usgangsadapter

Ereignis-Ausgänge

- **CNF** (Confirmation from Adapter): Bestätigt den erfolgreichen Abschluss der Konvertierung und den Empfang der Antwort vom AE2-Adapter

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

- **AE2_OUT** (Adapter Output): Bidirektionaler AE2-Ausgangsadapter für die Kommunikation mit AE2-kompatiblen Systemen

Funktionsweise

Der Funktionsblock leitet eingehende REQ-Ereignisse direkt an den AE2_OUT-Adapter weiter (EO1-Port). Gleichzeitig werden eingehende Bestätigungen vom AE2-Adapter (EI1-Port) als CNF-Ereignis ausgegeben. Dies ermöglicht eine transparente Konvertierung zwischen dem Standard-EVENT-Format und dem spezifischen AE2-Adapterprotokoll.

Technische Besonderheiten

- Implementiert als Composite-Funktionsblock
- Unterstützt bidirektionale Kommunikation
- Verwendet den AE2-Adaptertyp aus dem adapter::types::bidirectional-Paket
- Positionierte Adapter-Schnittstelle für optimierte Visualisierung

Zustandsübersicht

Der Funktionsblock verfügt über einen einfachen Zustandsautomaten:

1. **Idle-Zustand**: Wartet auf REQ-Ereignis
2. **Request-Zustand**: Nach REQ-Eingabe, Sendung an AE2_OUT.EO1
3. **Confirmation-Zustand**: Nach Empfang von AE2_OUT.EI1, Ausgabe von CNF

Anwendungsszenarien

- Integration von EVENT-basierten Systemen in AE2-kompatible Umgebungen
- Protokollkonvertierung in verteilten Automatisierungssystemen
- Bidirektionale Kommunikationsbrücken zwischen verschiedenen Automatisierungskomponenten
- Adapter-basierte Systemintegration

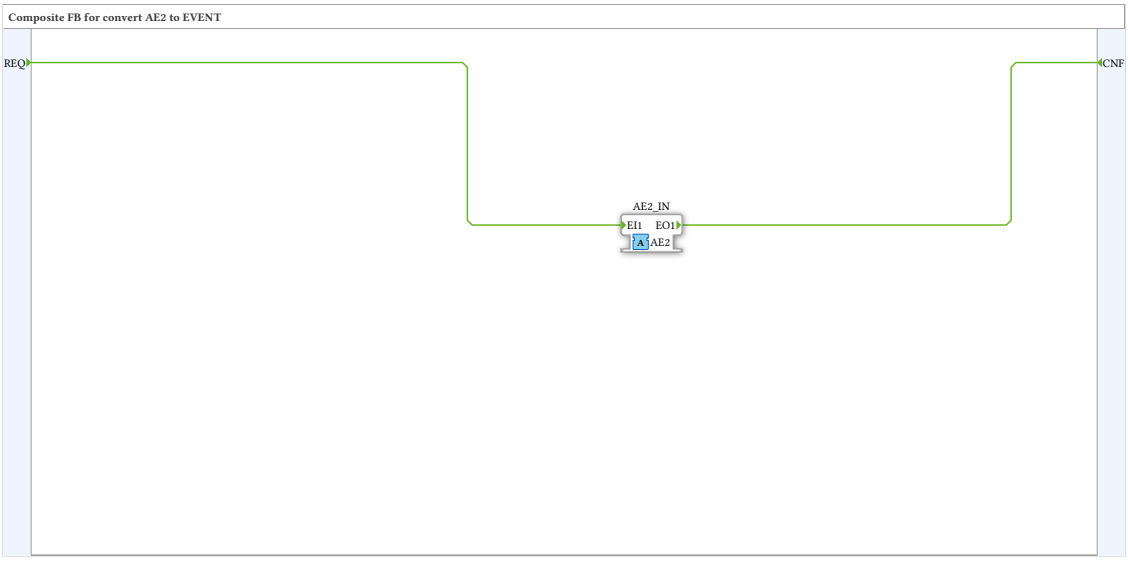
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen EVENT-Konvertern bietet AE2_EVENT_TO_E spezifische AE2-Adapterunterstützung mit bidirektionaler Kommunikation. Während Standard-Konverter oft nur unidirektional arbeiten, ermöglicht dieser Baustein vollständige Request-Response-Zyklen mit dem AE2-Protokoll.

Fazit

Der AE2_EVENT_TO_E Funktionsblock stellt eine zuverlässige Lösung für die Konvertierung zwischen Standard-EVENT-Signalen und dem AE2-Adapterprotokoll dar. Seine bidirektionale Funktionalität und einfache Integration machen ihn besonders geeignet für Systeme, die eine nahtlose Kommunikation zwischen verschiedenen Protokollwelten erfordern.

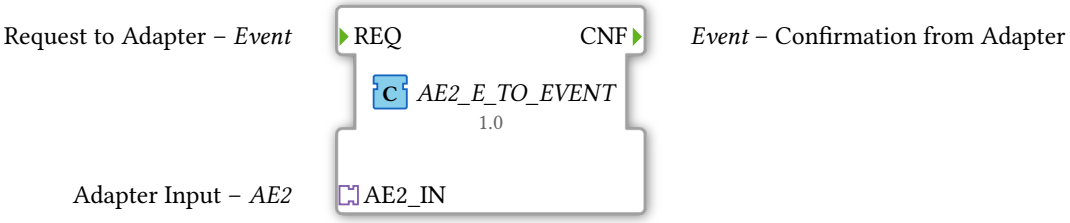
AE2_E_TO_EVENT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AE2_E_TO_EVENT ist ein Composite-Funktionsblock, der zur Konvertierung von AE2-Adapter-Signalen in Ereignisse dient. Er ermöglicht die bidirektionale Kommunikation zwischen Adapter-basierten Komponenten und ereignisgesteuerten Funktionsblöcken.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Request to Adapter): Startet die Konvertierung und sendet eine Anfrage an den Adapter

Ereignis-Ausgänge

- **CNF** (Confirmation from Adapter): Bestätigt den erfolgreichen Abschluss der Konvertierung

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

- **AE2_IN** (Adapter Input): Bidirektionaler AE2-Adapter-Eingang für die Kommunikation mit AE2-kompatiblen Komponenten

Funktionsweise

Der Funktionsblock arbeitet als einfacher Konverter zwischen dem AE2-Adapter-Protokoll und dem standardmäßigen 4diac-Ereignissystem. Bei Empfang eines REQ-Ereignisses wird eine Anfrage über den AE2-Adapter gesendet. Die Antwort des Adapters wird über das EO1-Signal empfangen und als CNF-Ereignis ausgegeben.

Technische Besonderheiten

- Implementiert als Composite-Funktionsblock
- Verwendet bidirektionale AE2-Adapter-Kommunikation
- Einfache Ereignisweiterleitung ohne Datenmanipulation
- Position des CNF-Ausgangs ist fest auf Y-Position 46 definiert

Zustandsübersicht

Der Funktionsblock verfügt über zwei Hauptzustände:

1. **Wartezustand**: Keine aktive Konvertierung
2. **Aktivzustand**: Konvertierung läuft (nach REQ-Eingang bis CNF-Ausgabe)

Anwendungsszenarien

- Integration von AE2-kompatiblen Geräten in 4diac-Steuerungen
- Bridge-Funktionalität zwischen verschiedenen Kommunikationsprotokollen
- Einfache Ereigniskonvertierung in Automatisierungssystemen
- Protokollumsetzung in verteilten Steuerungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Adapter-Konvertern zeichnet sich AE2_E_TO_EVENT durch seine spezifische Ausrichtung auf das AE2-Adapter-Protokoll aus. Während allgemeine Adapter-Konverter verschiedene Protokolle unterstützen, ist dieser Baustein speziell für die bidirektionale AE2-Kommunikation optimiert.

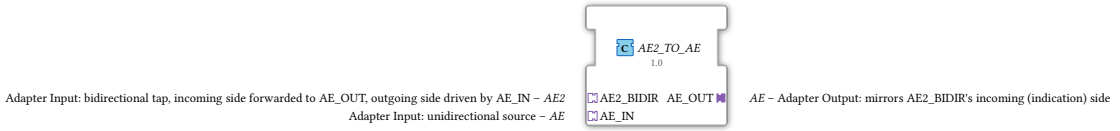
Fazit

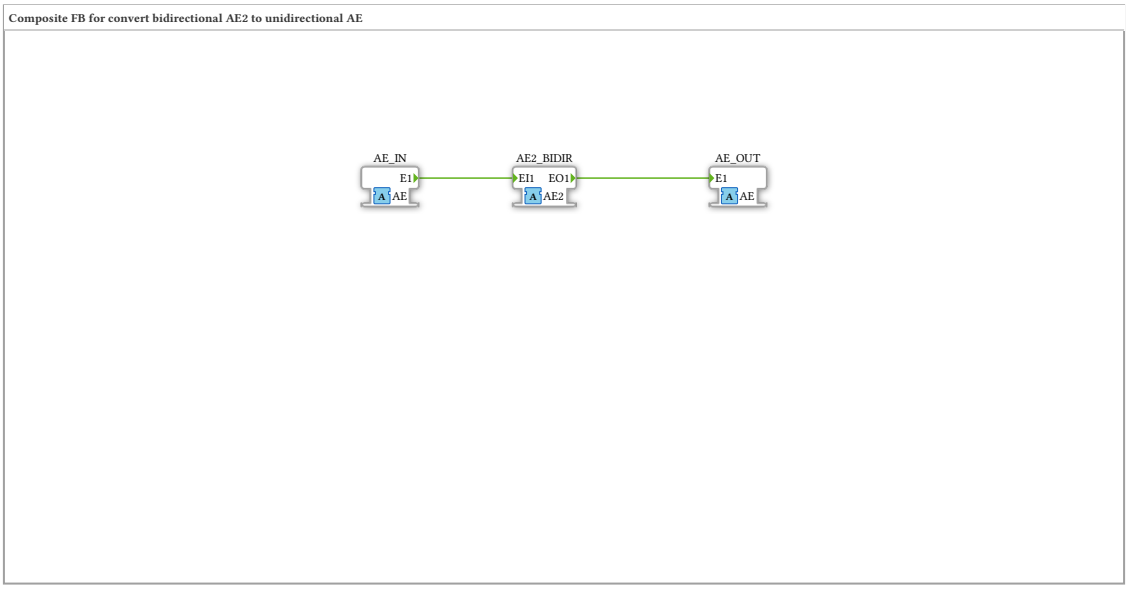
Der AE2_E_TO_EVENT Funktionsblock bietet eine zuverlässige und einfache Lösung zur Integration von AE2-Adaptern in ereignisgesteuerte 4diac-Systeme. Seine klare Schnittstellendefinition und direkte Funktionsweise machen ihn zu einem praktischen Werkzeug für Protokollkonvertierungen in industriellen Automatisierungsanwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AE2_TO_AE





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock AE2_TO_AE ist das Spiegelbild von AE_TO_AE2: Hier steht der bidirektionale AE2-Abzweig als Socket auf der linken (Eingangs-)Seite, ergänzt um einen zweiten unidirektionalen AE-Socket und einen unidirektionalen AE-Plug.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Kommunikation läuft ausschließlich über die Adapter.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine Daten-Ein- oder -Ausgänge – AE und AE2 transportieren ausschließlich Ereignisse, keine Nutzdaten.

Daten-Ausgänge

Der Funktionsblock verfügt über keine Daten-Ein- oder -Ausgänge – AE und AE2 transportieren ausschließlich Ereignisse, keine Nutzdaten.

Adapter

- **AE2_BIDIR** (Socket): bidirektionaler Abzweig, dessen eingehende Seite an `AE_OUT` weitergereicht wird und dessen ausgehende Seite von `AE_IN` gespeist wird, Typ `adapter::types::bidirectional::AE2`
- **AE_IN** (Socket): unidirektionale EVENT-Quelle, Typ `adapter::types::unidirectional::AE`
- **AE_OUT** (Plug): spiegelt die eingehende (Indikations-)Seite von `AE2_BIDIR` als unidirektionales Ereignis, Typ `adapter::types::unidirectional::AE`

Funktionsweise

Der Baustein arbeitet in zwei Richtungen gleichzeitig: Was der Socket `AE2_BIDIR` auf seiner Indikations-Seite empfängt (`EO1` – also was ein an AE2_BIDIR angeschlossener bidirektionaler Teilnehmer meldet), wird unverändert als eigenständiges unidirektionales Ereignis über den Plug `AE_OUT` ausgegeben. Umgekehrt wird alles, was am Socket `AE_IN` eintrifft (`E1`), auf die Anfrage-Seite von `AE2_BIDIR` (`EI1`) veröffentlicht – AE2_BIDIR sendet also dasselbe Ereignis an den angeschlossenen bidirektionalen Teilnehmer weiter. Da AE/AE2 reine Ereignisadapter ohne Nutzdaten sind, entfallen DataConnections vollständig.

Technische Besonderheiten

- Drei statt zwei Adapter: Die vom Tap empfangene und die separat eingespeiste Ereignisrichtung werden bewusst getrennt gehalten, statt sie zu vermischen
- Reine Verdrahtung ohne Logik oder Zustand, jede Zielvariable hat genau einen Schreiber
- Kein Datentransport: AE/AE2 tragen ausschließlich Ereignisse, daher besitzt dieser Baustein keine DataConnections
- `AE_OUT` gibt nur wieder, was über `AE2_BIDIR` von einem angeschlossenen bidirektionalen Teilnehmer gemeldet wird – ohne einen solchen Teilnehmer bleibt `AE_OUT` inaktiv

Zustandsübersicht

Der Baustein ist zustandslos:

- AE2_BIDIR.EO1 ⟶ AE_OUT.E1
- AE_IN.E1 ⟶ AE2_BIDIR.EI1

Anwendungsszenarien

- Auslesen eines bidirektionalen AE2-Teilnehmers als einfaches unidirektionales AE-Ereignis, kombiniert mit dem Einspeisen eines separaten unidirektionalen Ereignisses
- Systeme, in denen ein bidirektionaler Teilnehmer überwacht werden soll, ohne die restliche Steuerung auf AE2 umzustellen
- Brücke zwischen einer bidirektionalen AE2-Zone und einer unidirektionalen AE-Zone

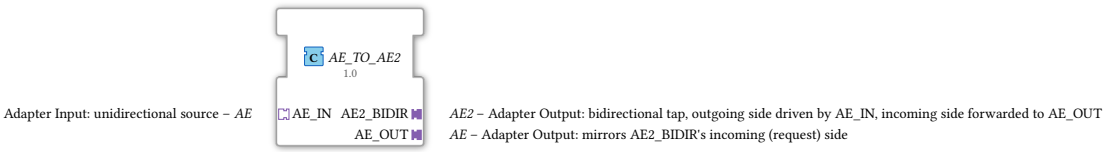
⚖️ Vergleich mit ähnlichen Bausteinen

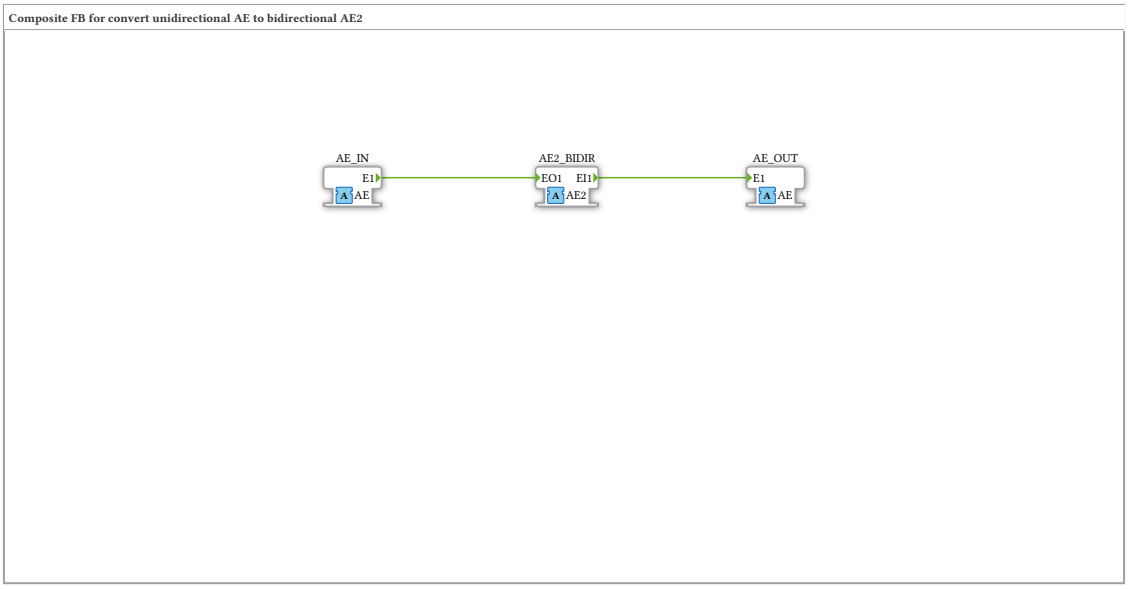
AE_TO_AE2 ist das Spiegelbild dieses Bausteins – dort steht `AE2_BIDIR` als Plug auf der rechten Seite. Für eine reine, nicht abzweigende 1:1-Umsetzung zwischen AE2 und einem plain EVENT-Signal siehe AE2_E_TO_EVENT/AE2_EVENT_TO_E. Für zwei gekoppelte Ereignisse (SET/RESET) siehe ASR2_TO_ASR.

Fazit

AE2_TO_AE ist kein einfacher Konverter, sondern ein Abzweigpunkt: Es macht den Zustand eines bidirektionalen AE2-Teilnehmers als eigenständiges unidirektionales AE-Ereignis sichtbar und speist gleichzeitig ein separates unidirektionales Ereignis in die bidirektionale Seite ein.

AE_TO_AE2





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock AE_TO_AE2 hängt einen bidirektionalen AE2-Abzweig („Tap“) in eine unidirektionale AE-Ereigniskette ein. Anders als eine einfache 1:1-Konvertierung nutzt er dafür **drei** Adapter: Ein eingehendes unidirektionales AE-Ereignis wird sowohl auf die bidirektionale AE2-Schnittstelle veröffentlicht als auch – was AE2 auf seiner Anfrage-Seite zurückliefert – als eigenständiges unidirektionales AE-Ereignis wieder ausgegeben.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Kommunikation läuft ausschließlich über die Adapter.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine Daten-Ein- oder -Ausgänge – AE und AE2 transportieren ausschließlich Ereignisse, keine Nutzdaten.

Daten-Ausgänge

Der Funktionsblock verfügt über keine Daten-Ein- oder -Ausgänge – AE und AE2 transportieren ausschließlich Ereignisse, keine Nutzdaten.

Adapter

- **AE_IN** (Socket): unidirektionale EVENT-Quelle, Typ `adapter::types::unidirectional::AE`
- **AE2_BIDIR** (Plug): bidirektionaler Abzweig, dessen ausgehende Seite von `AE_IN` gespeist wird und dessen eingehende Seite an `AE_OUT` weitergereicht wird, Typ `adapter::types::bidirectional::AE2`
- **AE_OUT** (Plug): spiegelt die eingehende (Anfrage-)Seite von `AE2_BIDIR` als unidirektionales Ereignis, Typ `adapter::types::unidirectional::AE`

Funktionsweise

Der Baustein arbeitet in zwei Richtungen gleichzeitig: Was am Socket `AE_IN` eintrifft (`E1`), wird direkt auf die ausgehende Seite von `AE2_BIDIR` (`EO1`) veröffentlicht – AE2_BIDIR meldet also nach außen dasselbe Ereignis, das über AE_IN hereinkommt. Umgekehrt wird alles, was `AE2_BIDIR` auf seiner Anfrage-Seite empfängt (`EI1` – also was ein an AE2_BIDIR angeschlossener bidirektionaler Teilnehmer „zurücksendet“),

unverändert als eigenständiges unidirektionales Ereignis über den Plug `AE_OUT` ausgegeben. Da AE/AE2 reine Ereignisadapter ohne Nutzdaten sind, entfallen DataConnections vollständig.

Technische Besonderheiten

- Drei statt zwei Adapter: Die eingehende und die vom Tap zurückkommende Ereignisrichtung werden bewusst getrennt gehalten, statt sie zu vermischen
- Reine Verdrahtung ohne Logik oder Zustand, jede Zielvariable hat genau einen Schreiber
- Kein Datentransport: AE/AE2 tragen ausschließlich Ereignisse, daher besitzt dieser Baustein keine DataConnections
- `AE_OUT` gibt nur wieder, was über `AE2_BIDIR` von einem angeschlossenen bidirektionalen Teilnehmer zurückkommt – ohne einen solchen Teilnehmer bleibt `AE_OUT` inaktiv

Zustandsübersicht

Der Baustein ist zustandslos:

- AE_IN.E1 ⟶ AE2_BIDIR.EO1
- AE2_BIDIR.EI1 ⟶ AE_OUT.E1

Anwendungsszenarien

- Nachträgliches Einhängen eines bidirektionalen Diagnose- oder Steuerpunkts in eine bestehende unidirektionale AE-Ereigniskette, ohne diese zu unterbrechen
- Systeme, in denen ein bidirektionaler Teilnehmer (z. B. ein Bedienpanel) sowohl ein unidirektionales Ereignis anzeigen als auch selbst ein Ereignis zurücksenden soll
- Brücke zwischen einer unidirektionalen AE-Zone und einer bidirektionalen AE2-Zone

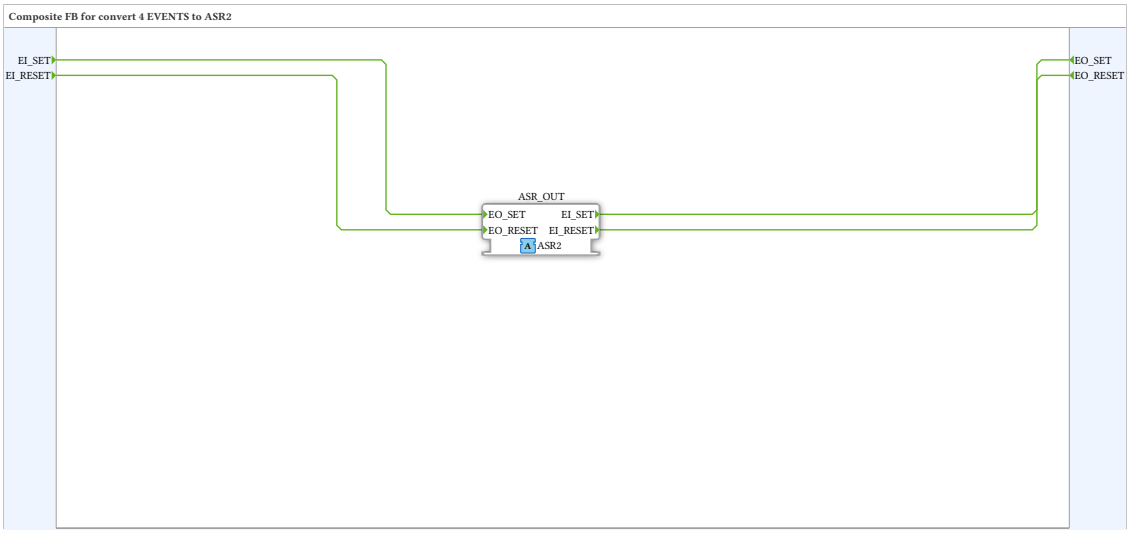
⚖️ Vergleich mit ähnlichen Bausteinen

AE2_TO_AE ist das Spiegelbild dieses Bausteins – dort steht `AE2_BIDIR` als Socket auf der linken Seite. Für eine reine, nicht abzweigende 1:1-Umsetzung zwischen einem plain EVENT-Signal und AE2 siehe AE2_EVENT_TO_E/AE2_E_TO_EVENT. Für zwei gekoppelte Ereignisse (SET/RESET) siehe ASR_TO_ASR2.

Fazit

AE_TO_AE2 ist kein einfacher Konverter, sondern ein Abzweigpunkt: Es macht ein unidirektionales AE-Ereignis an einem bidirektionalen AE2-Punkt sichtbar, ohne die ursprüngliche unidirektionale Ereigniskette zu unterbrechen, und gibt gleichzeitig das dort empfangene Anfrage-Ereignis als eigenständiges AE-Ereignis weiter.

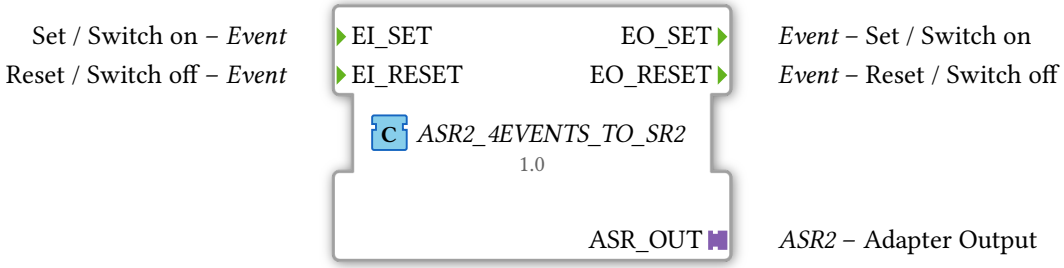
ASR2_4EVENTS_TO_SR2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock ASR2_4EVENTS_TO_SR2 ist ein Composite-Funktionsblock, der zur Konvertierung von 4 Ereignissen in das ASR2-Adapterformat dient. Er ermöglicht die bidirektionale Umwandlung zwischen einzelnen Set/Reset-Ereignissen und dem standardisierten ASR2-Adapter.



Schnittstellenstruktur

Ereignis-Eingänge

- **EI_SET**: Set-Ereignis (Einschalten)
- **EI_RESET**: Reset-Ereignis (Ausschalten)

Ereignis-Ausgänge

- **EO_SET**: Set-Ereignis (Einschalten)
- **EO_RESET**: Reset-Ereignis (Ausschalten)

Adapter

- **ASR_OUT**: ASR2-Adapter-Ausgang (bidirektional)

Funktionsweise

Der Funktionsblock fungiert als Brücke zwischen einzelnen Ereignissen und dem ASR2-Adapter. Er leitet eingehende Set- und Reset-Ereignisse an den ASR2-Adapter weiter und gibt gleichzeitig entsprechende Ereignisse vom ASR2-Adapter an die Ausgänge weiter. Dies ermöglicht eine nahtlose Integration von Einzelereignissen in Systeme, die das ASR2-Adapterformat verwenden.

Technische Besonderheiten

- Bidirektionale Signalübertragung
- Direkte Ereignisweiterleitung ohne Datenverarbeitung
- Kompatibel mit dem ASR2-Adapterstandard
- Composite-Funktionsblock-Implementierung

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand, da er ausschließlich als Durchschleif-Element fungiert. Alle Ereignisse werden direkt und ohne Verzögerung weitergeleitet.

Anwendungsszenarien

- Integration von Einzelereignis-Systemen in ASR2-basierte Architekturen
- Protokollkonvertierung zwischen verschiedenen Steuerungssystemen
- Adapter-Lösungen für bestehende Automatisierungssysteme
- Bidirektionale Kommunikation zwischen verschiedenen Ereignisformaten

⚖️ Vergleich mit ähnlichen Bausteinen

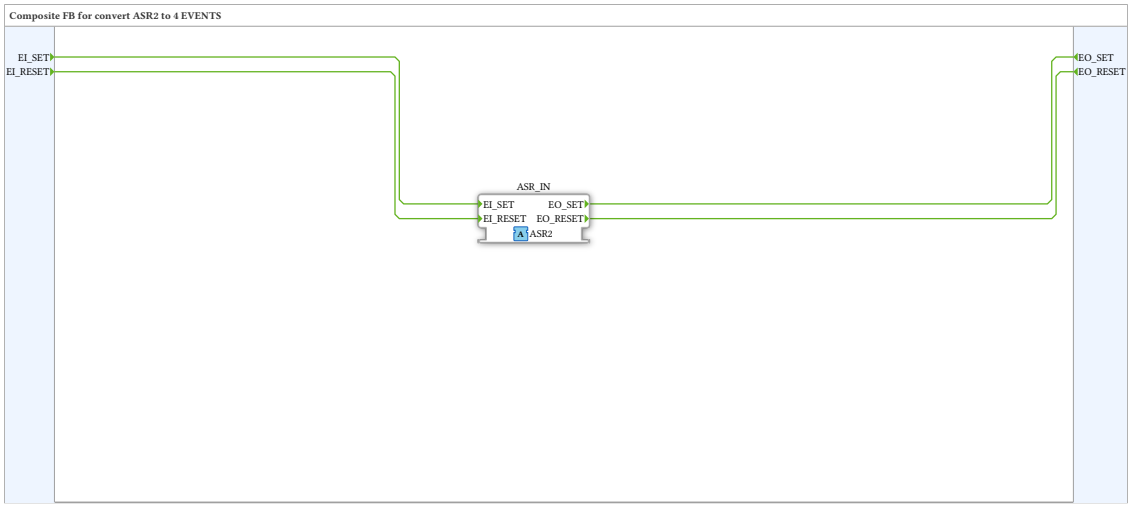
Im Vergleich zu einfachen Ereignis-Bausteinen bietet ASR2_4EVENTS_TO_SR2 spezifische Adapter-Funktionalität für das ASR2-Format. Während Standard-Bausteine typischerweise nur unidirektionale Signalverarbeitung bieten, ermöglicht dieser Baustein bidirektionale Kommunikation mit ASR2-kompatiblen Systemen.

Fazit

Der ASR2_4EVENTS_TO_SR2 Funktionsblock stellt eine effiziente Lösung für die Integration von Einzelereignissen in ASR2-basierte Systeme dar. Seine bidirektionale Funktionalität und direkte Ereignisweiter-

leitung machen ihn zu einem wertvollen Baustein für Systemintegration und Protokollkonvertierung in automatisierten Steuerungssystemen.

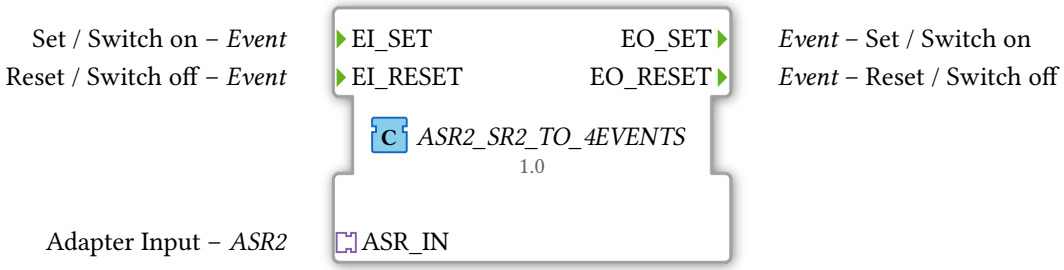
ASR2_SR2_TO_4EVENTS



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock ASR2_SR2_TO_4EVENTS ist ein Composite-Funktionsblock, der zur Bereitstellung der ASR2-Adaptersignale als vier diskrete Ereignis-Endpunkte dient. Er ermöglicht die bidirektionale Umwandlung zwischen ASR2-Adapter-Schnittstellen und diskreten Ereignis-Ein- und Ausgängen.



Schnittstellenstruktur

Ereignis-Eingänge

- **EI_SET**: Setzen / Einschalten
- **EI_RESET**: Rücksetzen / Ausschalten

Ereignis-Ausgänge

- **EO_SET**: Setzen / Einschalten
- **EO_RESET**: Rücksetzen / Ausschalten

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

- **ASR_IN**: Adapter-Eingang vom Typ `adapter::types::bidirectional::ASR2`

Funktionsweise

Der Funktionsblock fungiert als Brücke zwischen dem ASR2-Adapter und diskreten Ereignissen. Die Ereignis-Eingänge EI_SET und EI_RESET werden direkt an den ASR2-Adapter weitergeleitet, während die Ereignis-Ausgänge EO_SET und EO_RESET direkt vom ASR2-Adapter gespeist werden. Dies ermöglicht eine bidirektionale Kommunikation zwischen den beiden Schnittstellen.

Technische Besonderheiten

- Implementiert als Composite-Funktionsblock
- Verwendet bidirektionale Adapter-Kommunikation
- Enthält keine internen Datenverarbeitungslogiken, sondern dient primär als Signalweiterleitung
- Unterstützt die Eclipse 4diac Standardkonformität

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände, da er ausschließlich als Durchreichblock für Ereignisse fungiert. Alle eingehenden Ereignisse werden sofort an die entsprechende Schnittstelle weitergeleitet.

Anwendungsszenarien

- Integration von ASR2-kompatiblen Komponenten in 4diac-Systeme
- Konvertierung zwischen Adapter-basierten und ereignisbasierten Schnittstellen
- Bidirektionale Signalübertragung in Automatisierungssystemen
- Kopplung unterschiedlicher Kommunikationsprotokolle

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen Konverter-Blöcken bietet ASR2_SR2_TO_4EVENTS die Besonderheit der bidirektionalen Kommunikation und die spezifische Anpassung an ASR2-Adapter. Andere Konverter-Blöcke sind oft unidirektional oder unterstützen andere Adapter-Typen.

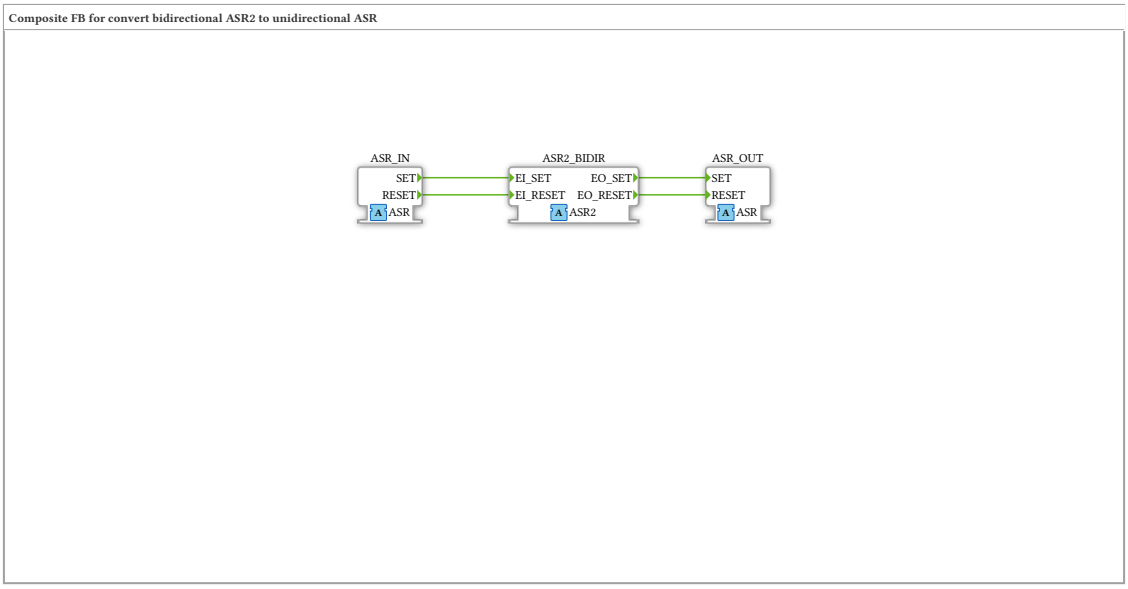
Fazit

Der ASR2_SR2_TO_4EVENTS Funktionsblock stellt eine effiziente Lösung für die Integration von ASR2-Schnittstellen in 4diac-basierte Steuerungssysteme dar. Durch seine einfache, aber effektive Durchreich-Architektur ermöglicht er eine nahtlose Kommunikation zwischen unterschiedlichen Schnittstellentypen ohne zusätzliche Verarbeitungslatenz.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ASR2_TO_ASR



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock ASR2_TO_ASR ist das Spiegelbild von ASR_TO_ASR2: Hier steht der bidirektionale ASR2-Abzweig als Socket auf der linken (Eingangs-)Seite, ergänzt um einen zweiten unidirektionalen ASR-Socket und einen unidirektionalen ASR-Plug.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Kommunikation läuft ausschließlich über die Adapter.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine Daten-Ein- oder -Ausgänge – ASR und ASR2 transportieren ausschließlich die Ereignisse SET/RESET, keine Nutzdaten.

Daten-Ausgänge

Der Funktionsblock verfügt über keine Daten-Ein- oder -Ausgänge – ASR und ASR2 transportieren ausschließlich die Ereignisse SET/RESET, keine Nutzdaten.

Adapter

- **ASR2_BIDIR** (Socket): bidirektionaler Abzweig, dessen eingehende Seite an `ASR_OUT` weitergereicht wird und dessen ausgehende Seite von `ASR_IN` gespeist wird, Typ `adapter::types::bidirectional::ASR2`
- **ASR_IN** (Socket): unidirektionale SET/RESET-Quelle, Typ `adapter::types::unidirectional::ASR`
- **ASR_OUT** (Plug): spiegelt die eingehende (Indikations-)Seite von `ASR2_BIDIR` als unidirektionale SET/RESET-Ereignisse, Typ `adapter::types::unidirectional::ASR`

Funktionsweise

Der Baustein arbeitet in zwei Richtungen gleichzeitig: Was der Socket `ASR2_BIDIR` auf seiner Indikations-Seite empfängt (`EO_SET`/`EO_RESET` – also was ein an ASR2_BIDIR angeschlossener bidirektionaler Teilnehmer meldet), wird unverändert als eigenständige unidirektionale Ereignisse über den Plug `ASR_OUT` (`SET`/`RESET`) ausgegeben. Umgekehrt wird alles, was am Socket `ASR_IN` eintrifft (`SET`/`RESET`), auf die Anfrage-Seite von `ASR2_BIDIR` (`EI_SET`/`EI_RESET`) veröffentlicht – ASR2_BIDIR sendet also dieselben

Ereignisse an den angeschlossenen bidirektionalen Teilnehmer weiter. Da ASR/ASR2 reine Ereignisadapter ohne Nutzdaten sind, entfallen DataConnections vollständig.

Technische Besonderheiten

- Drei statt zwei Adapter: Die vom Tap empfangene und die separat eingespeiste Ereignisrichtung werden bewusst getrennt gehalten, statt sie zu vermischen
- Zwei parallele, aber unabhängig geführte Ereignispaare (SET und RESET), keine Verrechnung oder Zusammenführung der beiden
- Reine Verdrahtung ohne Logik oder Zustand, jede Zielvariable hat genau einen Schreiber
- Kein Datentransport: ASR/ASR2 tragen ausschließlich SET/RESET-Ereignisse, daher besitzt dieser Baustein keine DataConnections
- `ASR_OUT` gibt nur wieder, was über `ASR2_BIDIR` von einem angeschlossenen bidirektionalen Teilnehmer gemeldet wird – ohne einen solchen Teilnehmer bleibt `ASR_OUT` inaktiv

Zustandsübersicht

Der Baustein ist zustandslos:

- ASR2_BIDIR.EO_SET ⟶ ASR_OUT.SET, ASR2_BIDIR.EO_RESET ⟶ ASR_OUT.RESET
- ASR_IN.SET ⟶ ASR2_BIDIR.EI_SET, ASR_IN.RESET ⟶ ASR2_BIDIR.EI_RESET

Anwendungsszenarien

- Auslesen eines bidirektionalen ASR2-Teilnehmers als einfaches unidirektionales ASR-SET/RESET-Paar, kombiniert mit dem Einspeisen eines separaten unidirektionalen SET/RESET-Paars
- Systeme, in denen ein bidirektionaler Teilnehmer überwacht werden soll, ohne die restliche Steuerung auf ASR2 umzustellen
- Brücke zwischen einer bidirektionalen ASR2-Zone und einer unidirektionalen ASR-Zone

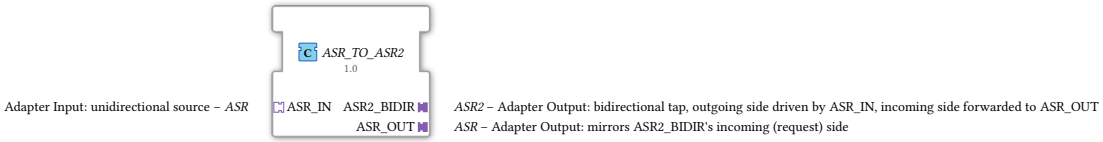
⚖️ Vergleich mit ähnlichen Bausteinen

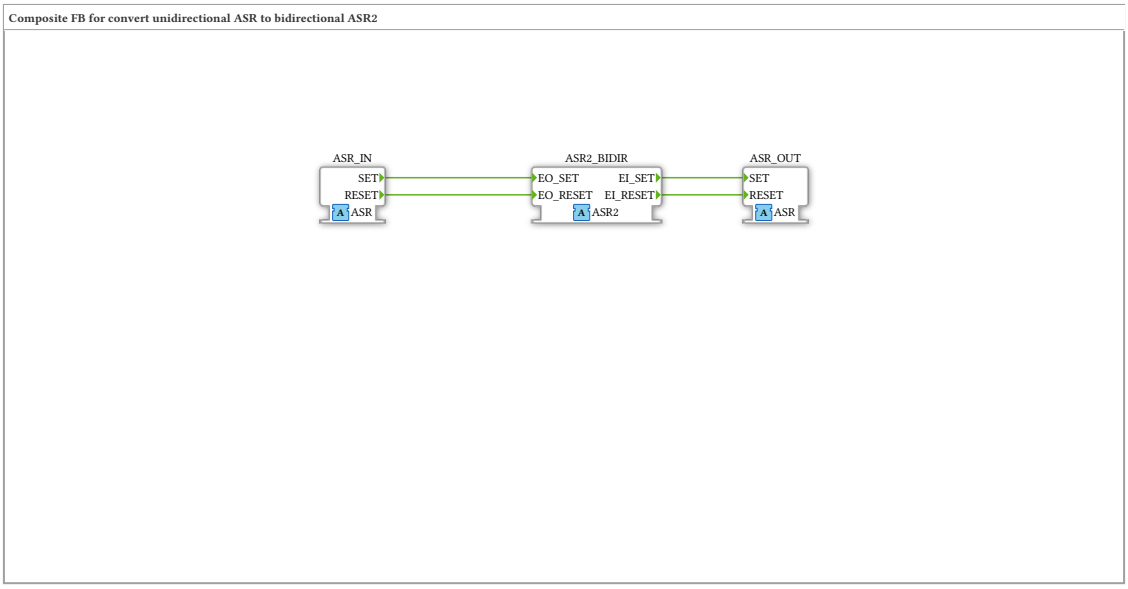
ASR_TO_ASR2 ist das Spiegelbild dieses Bausteins – dort steht `ASR2_BIDIR` als Plug auf der rechten Seite. Für eine reine, nicht abzweigende 1:1-Umsetzung zwischen ASR2 und zwei plain EVENT-Signalen siehe ASR2_SR2_TO_4EVENTS/ASR2_4EVENTS_TO_SR2. Für ein einzelnes Ereignis ohne SET/RESET-Kopplung siehe AE2_TO_AE.

Fazit

ASR2_TO_ASR ist kein einfacher Konverter, sondern ein Abzweigpunkt: Es macht den Zustand eines bidirektionalen ASR2-Teilnehmers als eigenständiges unidirektionales SET/RESET-Paar sichtbar und speist gleichzeitig ein separates unidirektionales SET/RESET-Paar in die bidirektionale Seite ein.

ASR_TO_ASR2





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock ASR_TO_ASR2 hängt einen bidirektionalen ASR2-Abzweig („Tap“) in eine unidirektionale ASR-Kette ein. ASR/ASR2 transportieren zwei gekoppelte Ereignisse, SET und RESET, statt eines einzelnen E1. Wie bei den anderen Bridge-Bausteinen nutzt ASR_TO_ASR2 dafür **drei** Adapter: Eingehende unidirektionale SET/RESET-Ereignisse werden sowohl auf die bidirektionale ASR2-Schnittstelle veröffentlicht als auch – was ASR2 auf seiner Anfrage-Seite zurückliefert – als eigenständige unidirektionale SET/RESET-Ereignisse wieder ausgegeben.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Kommunikation läuft ausschließlich über die Adapter.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine Daten-Ein- oder -Ausgänge – ASR und ASR2 transportieren ausschließlich die Ereignisse SET/RESET, keine Nutzdaten.

Daten-Ausgänge

Der Funktionsblock verfügt über keine Daten-Ein- oder -Ausgänge – ASR und ASR2 transportieren ausschließlich die Ereignisse SET/RESET, keine Nutzdaten.

Adapter

- **ASR_IN** (Socket): unidirektionale SET/RESET-Quelle, Typ `adapter::types::unidirectional::ASR`
- **ASR2_BIDIR** (Plug): bidirektionaler Abzweig, dessen ausgehende Seite von `ASR_IN` gespeist wird und dessen eingehende Seite an `ASR_OUT` weitergereicht wird, Typ `adapter::types::bidirectional::ASR2`
- **ASR_OUT** (Plug): spiegelt die eingehende (Anfrage-)Seite von `ASR2_BIDIR` als unidirektionale SET/RESET-Ereignisse, Typ `adapter::types::unidirectional::ASR`

Funktionsweise

Der Baustein arbeitet in zwei Richtungen gleichzeitig: Was am Socket `ASR_IN` eintrifft (`SET`/`RESET`), wird direkt auf die ausgehende Seite von `ASR2_BIDIR` (`EO_SET`/`EO_RESET`) veröffentlicht – ASR2_BIDIR meldet also nach außen dieselben Ereignisse, die über ASR_IN hereinkommen. Umgekehrt wird alles, was `ASR2_BIDIR` auf seiner Anfrage-Seite empfängt (`EI_SET`/`EI_RESET` – also was ein an ASR2_BIDIR

angeschlossener bidirektionaler Teilnehmer „zurücksendet“), unverändert als eigenständige unidirektionale Ereignisse über den Plug `ASR_OUT` (`SET`/`RESET`) ausgegeben. Da ASR/ASR2 reine Ereignisadapter ohne Nutzdaten sind, entfallen DataConnections vollständig.

Technische Besonderheiten

- Drei statt zwei Adapter: Die eingehende und die vom Tap zurückkommende Ereignisrichtung werden bewusst getrennt gehalten, statt sie zu vermischen
- Zwei parallele, aber unabhängig geführte Ereignispaare (SET und RESET), keine Verrechnung oder Zusammenführung der beiden
- Reine Verdrahtung ohne Logik oder Zustand, jede Zielvariable hat genau einen Schreiber
- Kein Datentransport: ASR/ASR2 tragen ausschließlich SET/RESET-Ereignisse, daher besitzt dieser Baustein keine DataConnections
- `ASR_OUT` gibt nur wieder, was über `ASR2_BIDIR` von einem angeschlossenen bidirektionalen Teilnehmer zurückkommt – ohne einen solchen Teilnehmer bleibt `ASR_OUT` inaktiv

Zustandsübersicht

Der Baustein ist zustandslos:

- ASR_IN.SET ⟶ ASR2_BIDIR.EO_SET, ASR_IN.RESET ⟶ ASR2_BIDIR.EO_RESET
- ASR2_BIDIR.EI_SET ⟶ ASR_OUT.SET, ASR2_BIDIR.EI_RESET ⟶ ASR_OUT.RESET

Anwendungsszenarien

- Nachträgliches Einhängen eines bidirektionalen Diagnose- oder Steuerpunkts in eine bestehende unidirektionale ASR-SET/RESET-Kette, ohne diese zu unterbrechen
- Systeme, in denen ein bidirektionaler Teilnehmer (z. B. ein Bedienpanel) sowohl SET/RESET-Zustände anzeigen als auch selbst SET/RESET-Ereignisse zurücksenden soll
- Brücke zwischen einer unidirektionalen ASR-Zone und einer bidirektionalen ASR2-Zone

⚖️ Vergleich mit ähnlichen Bausteinen

ASR2_TO_ASR ist das Spiegelbild dieses Bausteins – dort steht `ASR2_BIDIR` als Socket auf der linken Seite. Für eine reine, nicht abzweigende 1:1-Umsetzung zwischen zwei plain EVENT-Signalen und ASR2 siehe ASR2_4EVENTS_TO_SR2/ASR2_SR2_TO_4EVENTS. Für ein einzelnes Ereignis ohne SET/RESET-Kopplung siehe AE_TO_AE2.

Fazit

ASR_TO_ASR2 ist kein einfacher Konverter, sondern ein Abzweigpunkt: Es macht ein unidirektionales ASR-SET/RESET-Paar an einem bidirektionalen ASR2-Punkt sichtbar, ohne die ursprüngliche unidirektionale Kette zu unterbrechen, und gibt gleichzeitig das dort empfangene Anfrage-SET/RESET-Paar als eigenständiges ASR-Signal weiter.

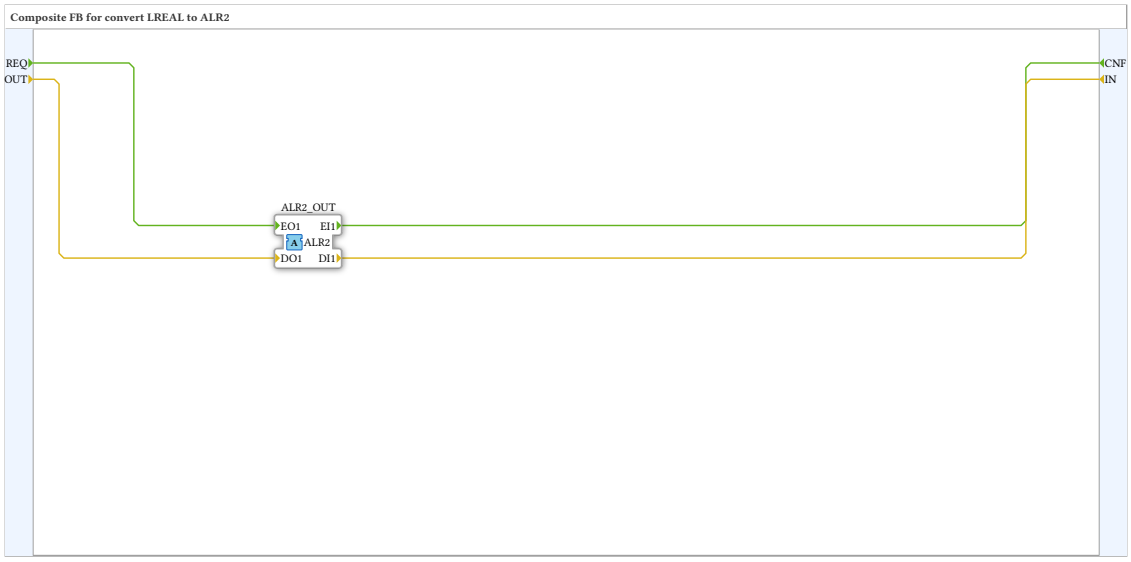
EVENT

🎧 Podcast

- E_TOF vs. FB_TOF: Der Event-Timer, der nicht zyklisch tickt – Revolution für Automatisierungssysteme?
- EventFBs nach IEC 61499: Legosteine der Automatisierung – So funktionieren Ereignis-Funktionsbausteine
- IEC 61499: Revolution der Automatisierung – Event-gesteuerte FBs und verteilte Systeme erklärt
- E_REND: Event Synchronization in IEC 61499
- The Unstoppable Counter: Why IEC 61499's ECTU Guarantees Safe, Event-Driven Control (and Never Overflows)

LREAL

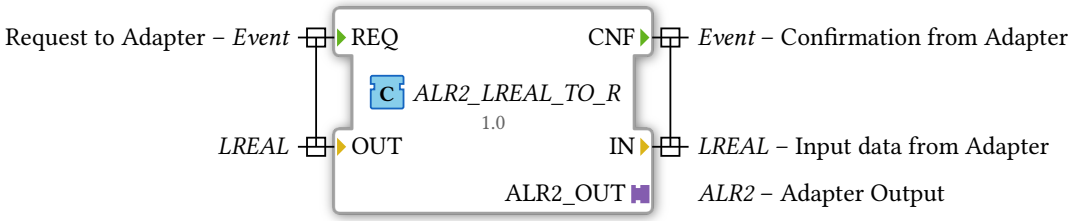
ALR2_LREAL_TO_R



• – ○ • – ○ • – ○ •

Einleitung

`ALR2_LREAL_TO_R` ist ein Composite-FB, der zwischen einem `LREAL`-Datenpaar (`OUT`/`IN`) und dem bidirektionalen Adapter `ALR2` vermittelt. Er vervollständigt die Konvertierungsfamilie analog zu `AR2`/`AB2`/`AE2`.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Anforderung an den Adapter. Verbundene Daten: `OUT`.

Ereignis-Ausgänge

- `CNF`: Bestätigung vom Adapter. Verbundene Daten: `IN`.

Daten-Eingänge

- `OUT` (`LREAL`): Wert, der zum Adapter gesendet wird.

Daten-Ausgänge

- `IN` (`LREAL`): Vom Adapter empfangener Wert.

Adapter

Richtung	Name	Adaptertyp	Kommentar
Plug (Ausgang)	`ALR2_OUT`	`adapter::types::bidirectional::ALR2`	Adapter-Ausgang.

Funktionsweise

Der Baustein enthält keine internen FBs, sondern nur Verbindungen:

- `REQ` ⟶ `ALR2_OUT.EO1` und `OUT` ⟶ `ALR2_OUT.DO1`: Ein `REQ`-Ereignis sendet den Wert `OUT` über den Adapter.
- `ALR2_OUT.EI1` ⟶ `CNF` und `ALR2_OUT.DI1` ⟶ `IN`: Ein über den Adapter empfangenes Ereignis wird als `CNF` ausgegeben, der empfangene Wert erscheint an `IN`.

Technische Besonderheiten

- Reine Durchreichung ohne Logik.
- Der Datentyp ist `LREAL` (64 Bit Gleitkomma).
- Die Benennung `OUT`/`IN` ist aus Sicht des Adapters gewählt: `OUT` geht zum Adapter hinaus, `IN` kommt vom Adapter herein.

Anwendungsszenarien

- Einen `LREAL`-Wert auf einen bidirektionalen `ALR2`-Adapter ausgeben und gleichzeitig dessen Rückkanal als Datenausgang auslesen.

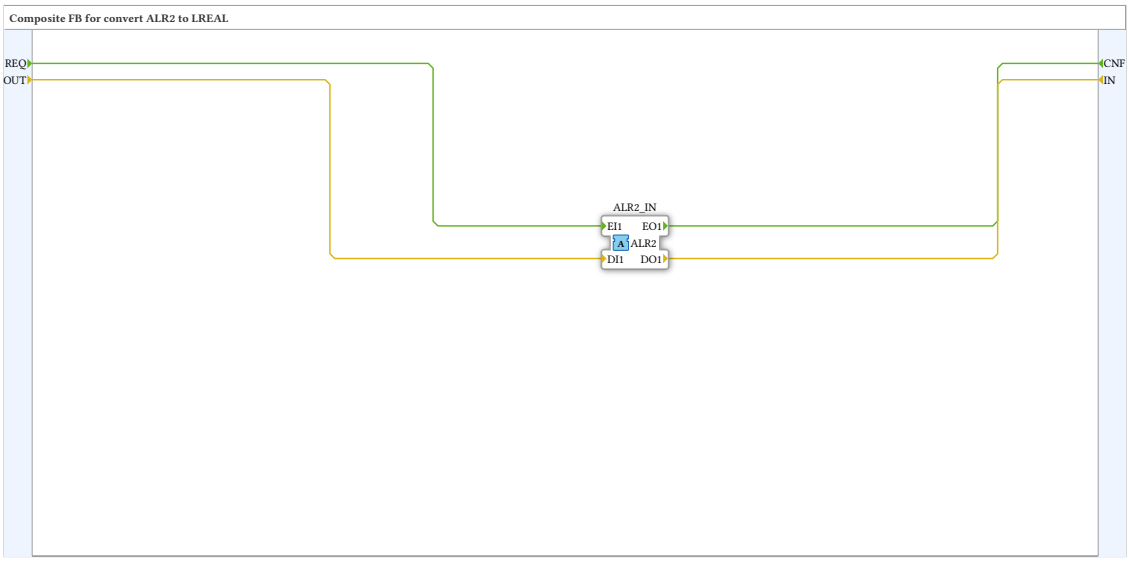
Zusammenfassung

`ALR2_LREAL_TO_R` verbindet Ereignisse und `LREAL`-Daten 1:1 mit dem bidirektionalen Adapter `ALR2`.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

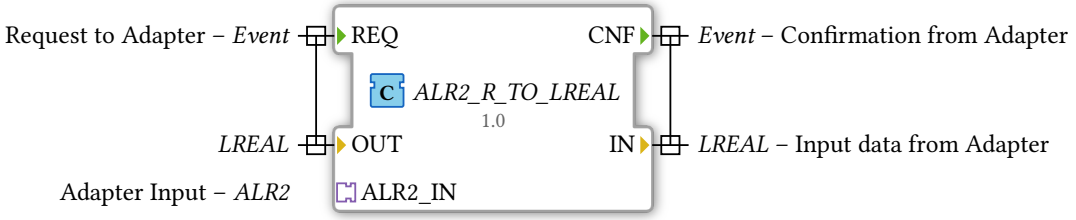
ALR2_R_TO_LREAL



• – ○ • – ○ • – ○ •

Einleitung

`ALR2_R_TO_LREAL` ist ein Composite-FB, der zwischen einem `LREAL`-Datenpaar (`OUT`/`IN`) und dem bidirektionalen Adapter `ALR2` vermittelt. Im Gegensatz zur Plug-Variante `ALR2_LREAL_TO_R` besitzt er einen Adapter-**Socket** (Eingang). Er vervollständigt die Konvertierungsfamilie analog zu `AR2`/`AB2`/`AE2`.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Anforderung an den Adapter. Verbundene Daten: `OUT`.

Ereignis-Ausgänge

- `CNF`: Bestätigung vom Adapter. Verbundene Daten: `IN`.

Daten-Eingänge

- `OUT` (`LREAL`): Wert, der zum Adapter gesendet wird.

Daten-Ausgänge

- `IN` (`LREAL`): Vom Adapter empfangener Wert.

Adapter

Richtung	Name	Adaptertyp	Kommentar
Socket (Eingang)	`ALR2_IN`	`adapter::types::bidirectional::ALR2`	Adapter-Eingang.

Funktionsweise

Der Baustein enthält keine internen FBs, sondern nur Verbindungen:

- `REQ` ⟶ `ALR2_IN.EI1` und `OUT` ⟶ `ALR2_IN.DI1`: Ein `REQ`-Ereignis sendet den Wert `OUT` über den Adapter.
- `ALR2_IN.EO1` ⟶ `CNF` und `ALR2_IN.DO1` ⟶ `IN`: Ein vom Adapter kommendes Ereignis wird als `CNF` ausgegeben, der empfangene Wert erscheint an `IN`.

Technische Besonderheiten

- Reine Durchreichung ohne Logik und ohne internen Zustand.
- Der Datentyp ist `LREAL` (64-Bit-Gleitkomma).
- Die Benennung `OUT`/`IN` ist aus Sicht des Adapters gewählt: `OUT` geht zum Adapter hinaus, `IN` kommt vom Adapter herein.

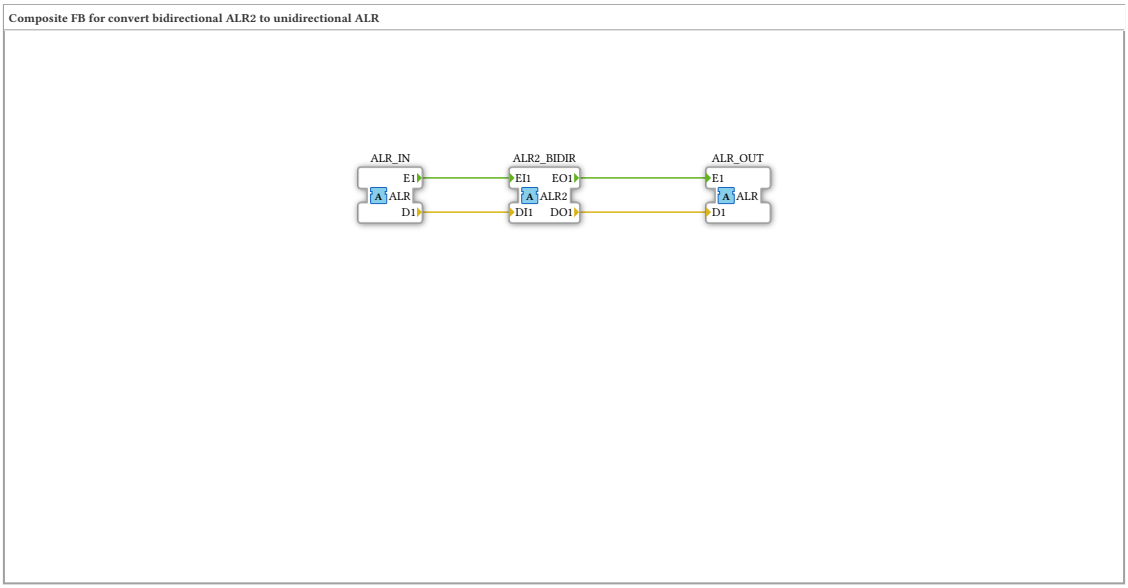
Anwendungsszenarien

- Einen `LREAL`-Wert auf einen bidirektionalen `ALR2`-Adapter ausgeben und gleichzeitig dessen Rückkanal als Datenausgang auslesen, wenn der Adapter als Socket vorliegt.

Zusammenfassung

`ALR2_R_TO_LREAL` verbindet Ereignisse und `LREAL`-Daten 1:1 mit dem bidirektionalen Adapter `ALR2` (Socket-Variante).

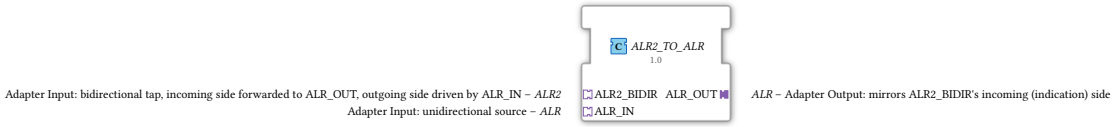
ALR2_TO_ALR



• – ○ • – ○ • – ○ •

Einleitung

`ALR2_TO_ALR` ist ein Composite-Funktionsblock, der einen bidirektionalen ALR2-Abzweig mit unidirektionalen ALR-Adaptern (LREAL) verbindet: Die eingehende Seite von `ALR2_BIDIR` wird als plain unidirektionales Signal über `ALR_OUT` ausgegeben, und `ALR_IN` wird auf die ausgehende Seite von `ALR2_BIDIR` veröffentlicht.



Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Kommunikation läuft ausschließlich über die Adapter.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **ALR2_BIDIR** (Socket): bidirektionaler Abzweig, dessen eingehende Seite an `ALR_OUT` weitergereicht wird und dessen ausgehende Seite von `ALR_IN` gespeist wird, Typ `adapter::types::bidirectional::ALR2`
- **ALR_IN** (Socket): unidirektionale Quelle, Typ `adapter::types::unidirectional::ALR`
- **ALR_OUT** (Plug): spiegelt die eingehende (Indikations-)Seite von `ALR2_BIDIR`, Typ `adapter::types::unidirectional::ALR`

Funktionsweise

Der Baustein besteht nur aus vier Verbindungen:

- Ereignis: `ALR2_BIDIR.EO1` ⟶ `ALR_OUT.E1`; Daten: `ALR2_BIDIR.DO1` ⟶ `ALR_OUT.D1`
- Ereignis: `ALR_IN.E1` ⟶ `ALR2_BIDIR.EI1`; Daten: `ALR_IN.D1` ⟶ `ALR2_BIDIR.DI1`

Was ein an `ALR2_BIDIR` angeschlossener bidirektionaler Teilnehmer meldet, erscheint also unverändert an `ALR_OUT`; was an `ALR_IN` eintrifft, wird an den bidirektionalen Teilnehmer weitergegeben.

Technische Besonderheiten

- Reine Verdrahtung ohne Logik oder Zustand
- Jede Zielvariable hat genau einen Schreiber
- Ohne angeschlossenen bidirektionalen Teilnehmer bleibt `ALR_OUT` inaktiv

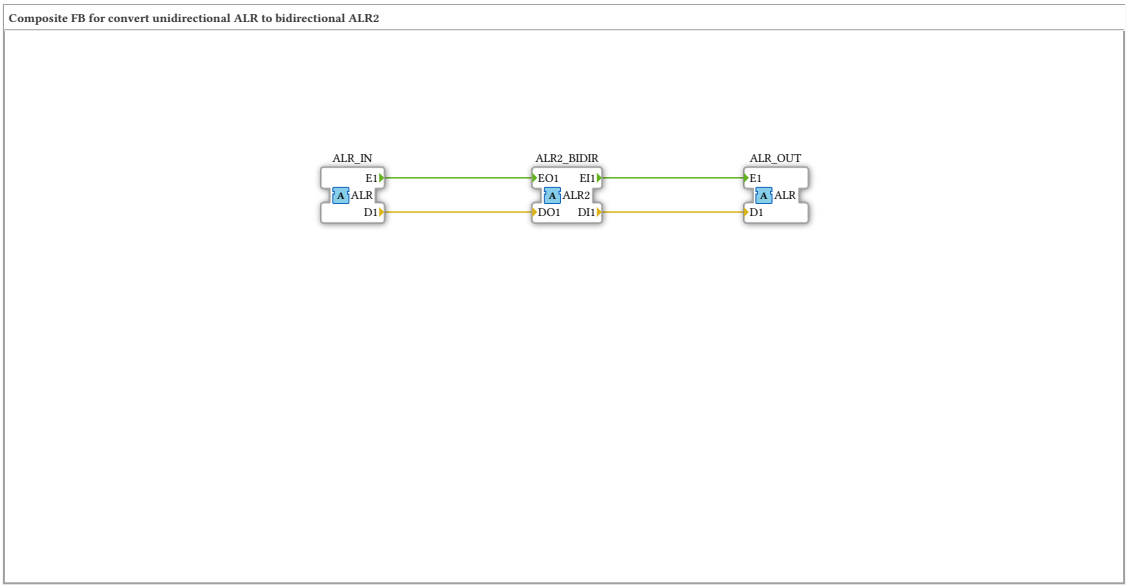
Anwendungsszenarien

- Auslesen eines bidirektionalen ALR2-Teilnehmers als einfaches unidirektionales ALR-Signal bei gleichzeitigem Einspeisen eines separaten unidirektionalen Signals
- Brücke zwischen einer bidirektionalen ALR2-Zone und einer unidirektionalen ALR-Zone

Fazit

`ALR2_TO_ALR` ist ein Abzweigpunkt ohne Logik: Er macht die eingehende Seite eines bidirektionalen ALR2-Teilnehmers als unidirektionales ALR-Signal sichtbar und speist gleichzeitig ein unidirektionales Signal in die ausgehende Seite ein.

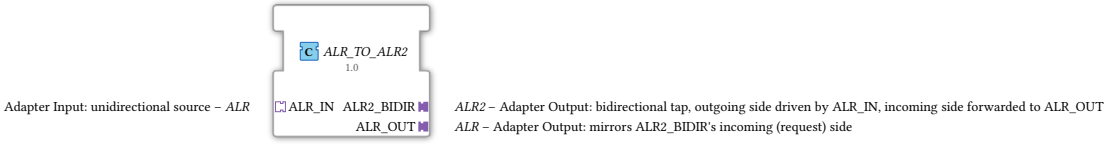
ALR_TO_ALR2



• – ◦ • – ◦ • – ◦ •

Einleitung

`ALR_TO_ALR2` ist ein Composite-Funktionsblock zur Konvertierung eines **unidirektionalen** `ALR`-Adapters in einen **bidirektionalen** `ALR2`-Adapter. `ALR_IN` wird auf die ausgehende Seite (`EO1`/`DO1`) von `ALR2_BIDIR` geschrieben; was `ALR2_BIDIR` auf seiner eingehenden Seite (`EI1`/`DI1`) empfängt, wird als einfaches unidirektionales Signal über `ALR_OUT` wieder ausgegeben.



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden – Ereignisse kommen über die Adapter-Sockets/-Plugs

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

- **ALR_IN** (Socket): Unidirektionaler Adapter vom Typ `adapter::types::unidirectional::ALR` – unidirektionale Quelle
- **ALR2_BIDIR** (Plug): Bidirektionaler Adapter vom Typ `adapter::types::bidirectional::ALR2` – ausgehende Seite wird von `ALR_IN` getrieben, eingehende Seite wird an `ALR_OUT` weitergeleitet
- **ALR_OUT** (Plug): Unidirektionaler Adapter vom Typ `adapter::types::unidirectional::ALR` – spiegelt die eingehende (Request-)Seite von `ALR2_BIDIR`

Funktionsweise

1. `ALR_IN.E1` wird an `ALR2_BIDIR.EO1` und `ALR_IN.D1` an `ALR2_BIDIR.DO1` weitergeleitet.
2. `ALR2_BIDIR.EI1` wird an `ALR_OUT.E1` und `ALR2_BIDIR.DI1` an `ALR_OUT.D1` weitergeleitet.

Technische Besonderheiten

- Reine Ereignis-/Datenverbindungen (`FBNetwork`), keine eigene Logik und kein interner Zustand
- Teil der Konvertierungsfamilie (analog zu `AR2`/`AB2`/`AE2`); der Datentyp der Daten ist `LREAL`

Anwendungsszenarien

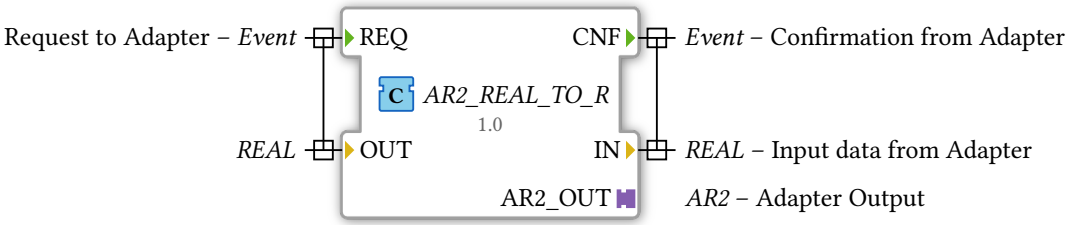
- Anbindung unidirektionaler `ALR`-Quellen und -Senken an einen bidirektionalen `ALR2`-Adapter

Fazit

`ALR_TO_ALR2` fasst einen unidirektionalen `ALR`-Eingang und -Ausgang zu einem bidirektionalen `ALR2`-Adapter zusammen.

REAL

AR2_REAL_TO_R



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `AR2_REAL_TO_R` ist ein zusammengesetzter Baustein (Composite FB) zur Konvertierung eines REAL-Wertes in ein AR2-Format. Er kapselt einen bidirektionalen Adapter des Typs `adapter::types::bidirectional::AR2` und fungiert als transparente Schnittstelle zwischen einer einfachen REAL-Variable und dem komplexeren AR2-Datentyp. Die Konvertierungslogik liegt vollständig im verwendeten Adapter, während der FB lediglich die Signal- und Datenweiterleitung übernimmt.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Kommentar
`REQ`	Anforderung zur Verarbeitung über den Adapter (löst Adapterausgang aus)

Ereignis-Ausgänge

Ereignis	Kommentar
`CNF`	Bestätigung nach Abschluss der Adapterverarbeitung

Daten-Eingänge

Variable	Typ	Kommentar
`OUT`	REAL	Ausgangsdaten, die an den Adapter gesendet werden (tatsächlich: Eingangswert für die Konvertierung)

Daten-Ausgänge

Variable	Typ	Kommentar
`IN`	REAL	Vom Adapter zurückgegebene Eingangsdaten (tatsächlich: konvertierter Wert)

Adapter

Adapter	Typ	Kommentar
`AR2_OUT`	`adapter::types::bidirectional::AR2`	Bidirektionaler Schnittstellenadapter für AR2-Konvertierung

Funktionsweise

Der FB arbeitet als reiner Weiterleitungsbaustein:

1. Ein eingehendes Ereignis am **Ereignis-Eingang `REQ`** wird direkt an den Ereignisausgang `EO1` des Adapters `AR2_OUT` weitergeleitet.
2. Der aktuelle Wert des **Daten-Eingangs `OUT`** wird gleichzeitig an den Datenausgang `DO1` des Adapters gesendet.
3. Der Adapter führt intern die Konvertierung vom REAL- in das AR2-Format durch (und ggf. zurück) und sendet ein Bestätigungs-Ereignis über seinen Ereigniseingang `EI1` zurück.
4. Dieses Ereignis wird an den **Ereignis-Ausgang `CNF`** des FBs weitergegeben.
5. Der vom Adapter bereitgestellte Wert an dessen Dateneingang `DI1` wird auf den **Daten-Ausgang `IN`** des FBs übertragen.

Die tatsächliche Umrechnung (REAL ↔ AR2) erfolgt ausschließlich im Adapter – der FB selbst führt keine arithmetischen Operationen durch.

Technische Besonderheiten

- **Zusammengesetzter Baustein**: Die Funktionalität wird durch die innere Verschaltung eines externen Adaptertyps realisiert. Der FB ist daher abhängig von der Implementierung des Adapters `adapter::types::bidirectional::AR2`.
- **Bidirektionale Datenrichtung**: Die Namensgebung der Variablen (`OUT` als Eingang, `IN` als Ausgang) kann verwirren. Tatsächlich fließen die Daten wie folgt:
  - REAL-Wert ⟶ Adapter ⟶ AR2-Objekt (Hinrichtung)
  - AR2-Objekt ⟶ Adapter ⟶ REAL-Wert (Rückrichtung)
    Der FB stellt beide Richtungen über die Ereignissteuerung `REQ` / `CNF` bereit.
- **Keine Zustandslogik**: Der FB besitzt keinen eigenen Zustandsautomaten – alle Abläufe sind ereignisgesteuert und direkt.

Zustandsübersicht

Da der FB keine interne Zustandsmaschine besitzt, liegt keine klassische Zustandsübersicht vor. Der Ablauf ist rein sequenziell pro Ereignis:

1. Warten auf `REQ`
2. Senden an Adapter
3. Warten auf Adapter-Rückmeldung
4. Ausgabe von `CNF` und aktualisiertem `IN`

Anwendungsszenarien

- **Schnittstelle zu externen Systemen**: Wenn ein externes Gerät oder ein anderer Funktionsblock Daten im AR2-Format erwartet, der Anwendercode jedoch mit REAL-Werten arbeitet.
- **Datenvorverarbeitung**: Einbindung in eine Kette von Konvertierungsbausteinen, um REAL-Daten in ein spezifisches binäres oder strukturiertes Format (AR2) zu überführen.
- **Test und Simulation**: Einsatz in Testumgebungen, um Adapterfunktionalität ohne tiefere Einblicke in den Adapter zu prüfen.

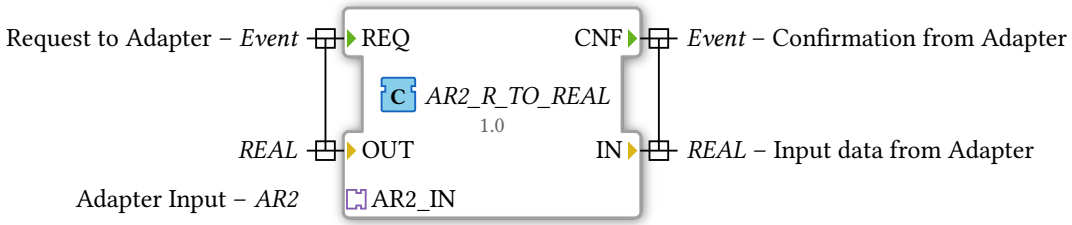
Vergleich mit ähnlichen Bausteinen

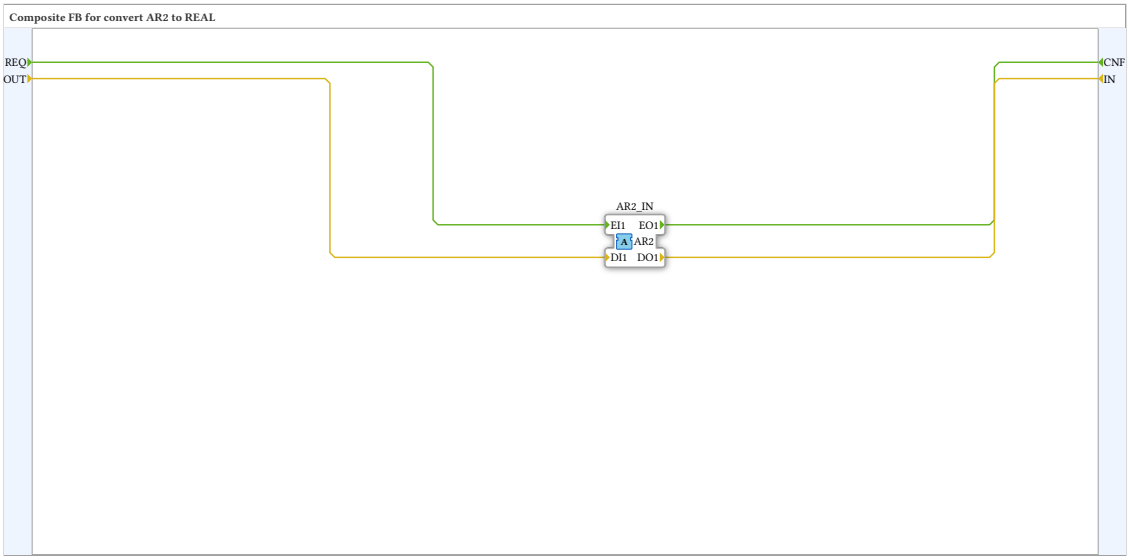
- **AR2_REAL_TO_R vs. einfache Typkonvertierung (REAL_TO_INT etc.)**:
  Einfache Konverter arbeiten direkt auf elementaren Datentypen. `AR2_REAL_TO_R` hingegen verwendet einen Adapter, der eine komplexe Umwandlung (z. B. in ein Array oder eine Struktur) kapselt.
- **AR2_REAL_TO_R vs. direkte Adapter-Nutzung**:
  Der FB vereinfacht die Handhabung, indem er die Signalverdrahtung (Ereignis- und Datenverbindungen) vornimmt. Der Anwender muss nur noch den FB instanziieren und mit seinem Code verbinden.
- **AR2_REAL_TO_R vs. CONVERT-Adapter-Bausteine**:
  Ähnliche Bausteine (z. B. aus der IEC 61499-Bibliothek) bieten meist generische Konvertierung. Dieser FB ist spezifisch für die Kopplung von REAL und AR2.

Fazit

Der Funktionsblock `AR2_REAL_TO_R` bietet eine einfache, ereignisgesteuerte Schnittstelle zur Nutzung eines bidirektionalen AR2-Adapters. Er ist ideal für alle Anwendungen, die eine Konvertierung zwischen dem Standard-Datentyp REAL und einem projektspezifischen AR2-Format benötigen. Durch seine Zusammensetzung bleibt die Komplexität im Adapter verborgen, und der Baustein kann wie ein normaler FB in Anwendungen integriert werden.

AR2_R_TO_REAL





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AR2_R_TO_REAL** ist ein Composite-Baustein, der als Brücke zwischen einer AR2-Adapter-Schnittstelle und REAL-Werten dient. Er ermöglicht die bidirektionale Datenübertragung und Konvertierung, wobei die eigentliche Umwandlung durch einen externen Adapter vom Typ `adapter::types::bidirectional::AR2` realisiert wird.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung
`REQ`	Startet die Datenübertragung an den Adapter. Der Datenwert `OUT` wird gesendet.

Ereignis-Ausgänge

Ereignis	Beschreibung
`CNF`	Quittiert die Rückmeldung des Adapters. Der konvertierte Wert liegt als `IN` an.

Daten-Eingänge

Variable	Typ	Beschreibung
OUT	REAL	Datenwert, der an den Adapter übergeben wird.

Daten-Ausgänge

Variable	Typ	Beschreibung
IN	REAL	Vom Adapter zurückgelieferter konvertierter Wert.

Adapter

Name	Typ	Richtung	Beschreibung
AR2_IN	adapter::types::bidirectional::AR2	Socket	Bidirektionaler Adapter für die AR2-Kommunikation und Konvertierung.

Funktionsweise

Der Baustein fungiert als reine Vermittlungsschicht:

1. Ein eingehendes Ereignis REQ löst die Übergabe des Wertes OUT (REAL) an den verbundenen Adapter aus (über AR2_IN.DI1).
2. Der Adapter führt die erforderliche Konvertierung durch (z. B. AR2 ↔ REAL) und sendet ein Bestätigungsereignis an seinen Ausgang EO1.
3. Dieses Ereignis wird als CNF nach außen geleitet, und der konvertierte Wert wird über AR2_IN.DO1 als IN (REAL) bereitgestellt.

Die tatsächliche Konvertierungsrichtung wird durch die Implementierung des angeschlossenen Adapters bestimmt. Der Name des Bausteins deutet auf eine primäre Umwandlung von AR2 nach REAL hin, die bidirektionale Adapter-Schnittstelle erlaubt jedoch auch die umgekehrte Richtung.

Technische Besonderheiten

- **Composite-Baustein**: Enthält keine eigene Logik, sondern kombiniert lediglich einen Adapter-Socket mit Ereignis- und Datenverbindungen.
- **Bidirektionaler Adapter**: Die Schnittstelle AR2_IN muss einen Adapter vom Typ adapter::types::bidirectional::AR2 bereitstellen, der die eigentliche Konvertierung implementiert.
- **Einfache Kopplung**: Der Baustein trennt die REAL-Welt von der AR2-Welt und ermöglicht so eine modulare Integration.

Zustandsübersicht

Der Funktionsblock besitzt keine eigenen internen Zustände. Sein Verhalten wird vollständig durch die externen Ereignisse REQ und die Rückmeldung des Adapters gesteuert.

Anwendungsszenarien

- **Anbindung von AR2-basierten Sensoren/Aktoren** an ein Steuerungssystem, das mit REAL-Werten arbeitet.
- **Bidirektionale Datenbrücke** in gemischten Automatisierungsumgebungen, in denen AR2- und REAL-Protokolle aufeinandertreffen.
- **Test- und Simulationsaufbauten**, in denen AR2-Daten durch einfach handhabbare REAL-Werte ersetzt werden sollen.

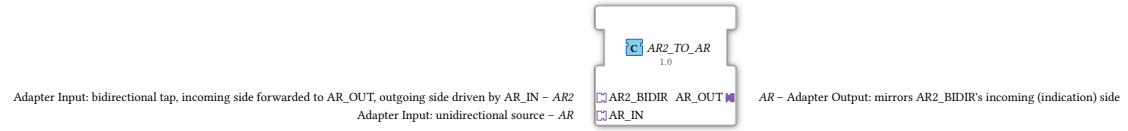
Vergleich mit ähnlichen Bausteinen

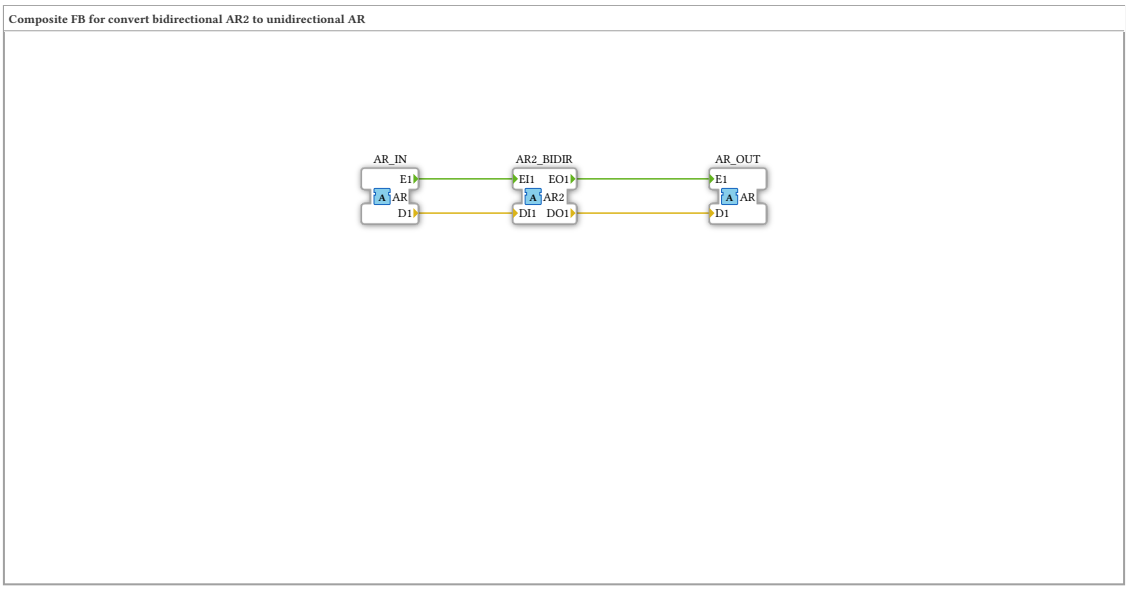
Während direkte Konvertierungsbausteine (z. B. AR2_TO_REAL) die Umwandlung fest verdrahten, erlaubt dieser Composite-Baustein die flexible Ankopplung eines externen Adapters. Dadurch wird die Konvertierungslogik entkoppelt und kann je nach Anwendungsfall ausgetauscht werden, ohne die restliche Konfiguration zu ändern.

Fazit

Der **AR2_R_TO_REAL** ist ein nützlicher Brückenbaustein für Systeme, die sowohl AR2- als auch REAL-Daten verarbeiten müssen. Durch die Nutzung eines bidirektionalen Adapters wird eine klare Trennung zwischen Schnittstelle und Konvertierungslogik erreicht, was Wiederverwendbarkeit und Wartbarkeit verbessert.

AR2_TO_AR





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock AR2_TO_AR ist das Spiegelbild von AR_TO_AR2: Hier steht der bidirektionale AR2-Abzweig als Socket auf der linken (Eingangs-)Seite, ergänzt um einen zweiten unidirektionalen AR-Socket und einen unidirektionalen AR-Plug.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Kommunikation läuft ausschließlich über die Adapter.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **AR2_BIDIR** (Socket): bidirektionaler Abzweig, dessen eingehende Seite an `AR_OUT` weitergereicht wird und dessen ausgehende Seite von `AR_IN` gespeist wird, Typ `adapter::types::bidirectional::AR2`
- **AR_IN** (Socket): unidirektionale REAL-Quelle, Typ `adapter::types::unidirectional::AR`

- **AR_OUT** (Plug): spiegelt die eingehende (Indikations-)Seite von `AR2_BIDIR` als unidirektionales Signal, Typ `adapter::types::unidirectional::AR`

Funktionsweise

Der Baustein arbeitet in zwei Richtungen gleichzeitig: Was der Socket `AR2_BIDIR` auf seiner Indikations-Seite empfängt (`EO1`/`DO1` – also was ein an AR2_BIDIR angeschlossener bidirektionaler Teilnehmer meldet), wird unverändert als eigenständiges unidirektionales Signal über den Plug `AR_OUT` ausgegeben. Umgekehrt wird alles, was am Socket `AR_IN` eintrifft (`E1`/`D1`), auf die Anfrage-Seite von `AR2_BIDIR` (`EI1`/`DI1`) veröffentlicht – AR2_BIDIR sendet also dasselbe Signal an den angeschlossenen bidirektionalen Teilnehmer weiter.

Technische Besonderheiten

- Drei statt zwei Adapter: Die vom Tap empfangene und die separat eingespeiste Signalrichtung werden bewusst getrennt gehalten, statt sie zu vermischen
- Reine Verdrahtung ohne Logik oder Zustand, jede Zielvariable hat genau einen Schreiber
- `AR_OUT` gibt nur wieder, was über `AR2_BIDIR` von einem angeschlossenen bidirektionalen Teilnehmer gemeldet wird – ohne einen solchen Teilnehmer bleibt `AR_OUT` inaktiv

Zustandsübersicht

Der Baustein ist zustandslos:

- AR2_BIDIR.EO1 ⟶ AR_OUT.E1, AR2_BIDIR.DO1 ⟶ AR_OUT.D1
- AR_IN.E1 ⟶ AR2_BIDIR.EI1, AR_IN.D1 ⟶ AR2_BIDIR.DI1

Anwendungsszenarien

- Auslesen des Status eines bidirektionalen AR2-Teilnehmers als einfaches unidirektionales AR-Signal, kombiniert mit dem Einspeisen eines separaten unidirektionalen Sollwertsignals
- Systeme, in denen ein bidirektionaler Teilnehmer überwacht werden soll, ohne die restliche Steuerung auf AR2 umzustellen
- Brücke zwischen einer bidirektionalen AR2-Zone und einer unidirektionalen AR-Zone

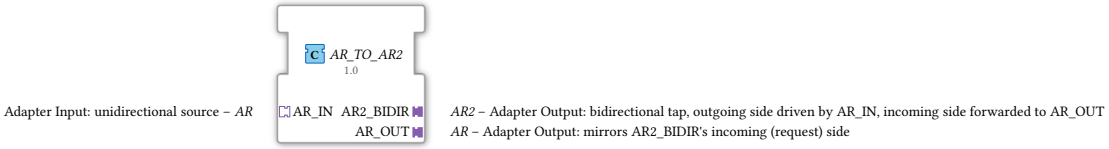
⚖️ Vergleich mit ähnlichen Bausteinen

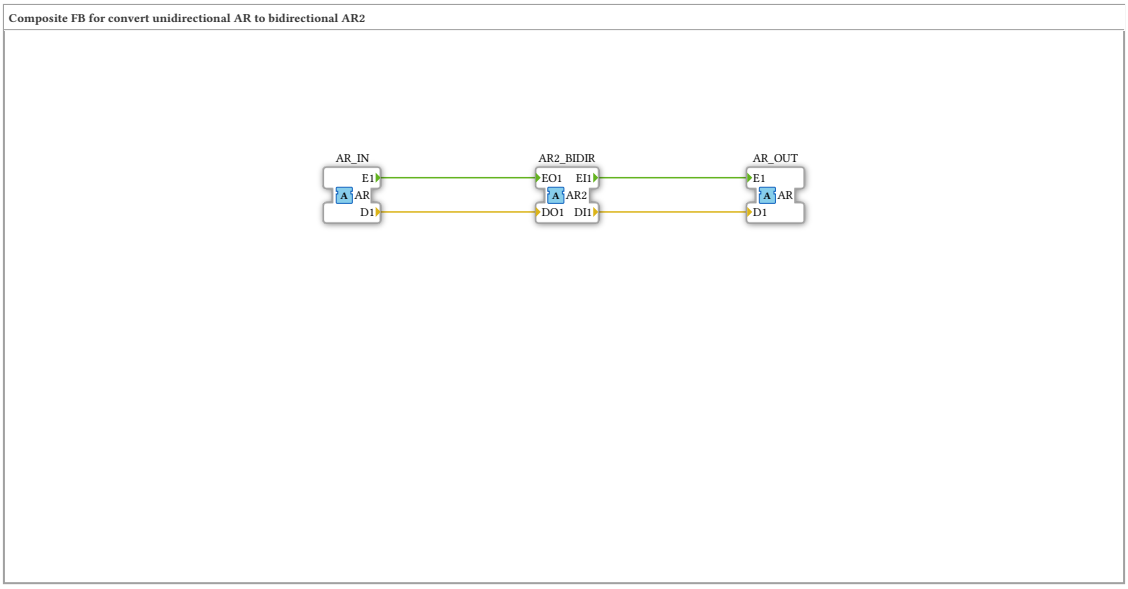
AR_TO_AR2 ist das Spiegelbild dieses Bausteins – dort steht `AR2_BIDIR` als Plug auf der rechten Seite. Für eine reine, nicht abzweigende 1:1-Umsetzung zwischen AR2 und einem plain REAL-Signal siehe AR2_REAL_TO_R/AR2_R_TO_REAL.

Fazit

AR2_TO_AR ist kein einfacher Konverter, sondern ein Abzweigpunkt: Es macht den Zustand eines bidirektionalen AR2-Teilnehmers als eigenständiges unidirektionales AR-Signal sichtbar und speist gleichzeitig ein separates unidirektionales Signal in die bidirektionale Seite ein.

AR_TO_AR2





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock AR_TO_AR2 hängt einen bidirektionalen AR2-Abzweig („Tap“) in eine unidirektionale AR-Kette ein. Anders als eine einfache 1:1-Konvertierung nutzt er dafür **drei** Adapter: Ein eingehendes unidirektionales AR-Signal wird sowohl auf die bidirektionale AR2-Schnittstelle veröffentlicht als auch – was AR2 auf seiner Anfrage-Seite zurückliefert – als eigenständiges unidirektionales AR-Signal wieder ausgegeben.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Kommunikation läuft ausschließlich über die Adapter.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **AR_IN** (Socket): unidirektionale REAL-Quelle, Typ `adapter::types::unidirectional::AR`
- **AR2_BIDIR** (Plug): bidirektionaler Abzweig, dessen ausgehende Seite von `AR_IN` gespeist wird und dessen eingehende Seite an `AR_OUT` weitergereicht wird, Typ `adapter::types::bidirectional::AR2`
- **AR_OUT** (Plug): spiegelt die eingehende (Anfrage-)Seite von `AR2_BIDIR` als unidirektionales Signal, Typ `adapter::types::unidirectional::AR`

Funktionsweise

Der Baustein arbeitet in zwei Richtungen gleichzeitig: Was am Socket `AR_IN` eintrifft (`E1`/`D1`), wird direkt auf die ausgehende Seite von `AR2_BIDIR` (`EO1`/`DO1`) veröffentlicht – AR2_BIDIR meldet also nach außen dasselbe Signal, das über AR_IN hereinkommt. Umgekehrt wird alles, was `AR2_BIDIR` auf seiner Anfrage-Seite empfängt (`EI1`/`DI1` – also was ein an AR2_BIDIR angeschlossener bidirektionaler Teilnehmer „zurücksendet“), unverändert als eigenständiges unidirektionales Signal über den Plug `AR_OUT` ausgegeben. AR2_BIDIR fungiert damit als Abzweigpunkt, der die unidirektionale Kette nicht unterbricht, sondern um eine bidirektionale Beobachtungs-/Einspeisemöglichkeit erweitert.

Technische Besonderheiten

- Drei statt zwei Adapter: Die eingehende und die vom Tap zurückkommende Signalrichtung werden bewusst getrennt gehalten, statt sie zu vermischen
- Reine Verdrahtung ohne Logik oder Zustand, jede Zielvariable hat genau einen Schreiber
- `AR_OUT` gibt nur wieder, was über `AR2_BIDIR` von einem angeschlossenen bidirektionalen Teilnehmer zurückkommt – ohne einen solchen Teilnehmer bleibt `AR_OUT` inaktiv

Zustandsübersicht

Der Baustein ist zustandslos:

- AR_IN.E1 ⟶ AR2_BIDIR.EO1, AR_IN.D1 ⟶ AR2_BIDIR.DO1
- AR2_BIDIR.EI1 ⟶ AR_OUT.E1, AR2_BIDIR.DI1 ⟶ AR_OUT.D1

Anwendungsszenarien

- Nachträgliches Einhängen eines bidirektionalen Diagnose- oder Steuerpunkts in eine bestehende unidirektionale AR-Signalkette, ohne diese zu unterbrechen
- Systeme, in denen ein bidirektionaler Teilnehmer (z. B. ein Bedienpanel) sowohl den Status eines unidirektionalen Messwertgebers anzeigen als auch selbst Sollwerte zurücksenden soll
- Brücke zwischen einer unidirektionalen AR-Zone und einer bidirektionalen AR2-Zone

⚖️ Vergleich mit ähnlichen Bausteinen

AR2_TO_AR ist das Spiegelbild dieses Bausteins – dort steht `AR2_BIDIR` als Socket auf der linken Seite. Für eine reine, nicht abzweigende 1:1-Umsetzung zwischen einem plain REAL-Signal und AR2 siehe AR2_R_TO_REAL/AR2_REAL_TO_R.

Fazit

AR_TO_AR2 ist kein einfacher Konverter, sondern ein Abzweigpunkt: Es macht ein unidirektionales AR-Signal an einem bidirektionalen AR2-Punkt sichtbar, ohne die ursprüngliche unidirektionale Signalkette zu unterbrechen, und gibt gleichzeitig das dort empfangene Anfrage-Signal als eigenständiges AR-Signal weiter.

SINT

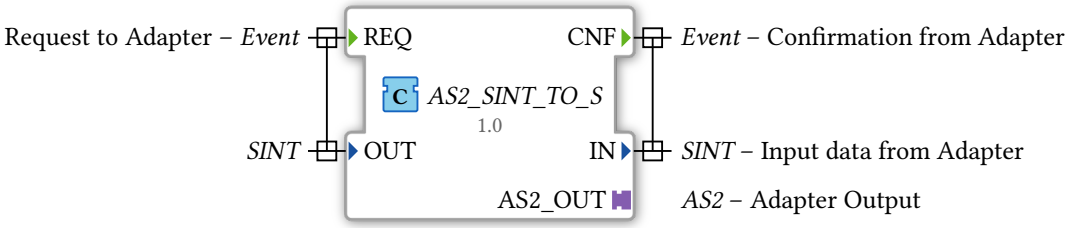
AS2_SINT_TO_S

• – ◦ • – ◦ • – ◦ •

Einleitung

`AS2_SINT_TO_S` ist ein Composite-Funktionsblock, der zwischen klassischen Ereignis-/Datenanschlüssen vom Typ `SINT` und einem **bidirektionalen** `AS2`-Adapter vermittelt. Der Eingang `OUT` wird bei `REQ` auf

die ausgehende Seite des Adapter-Plugs `AS2_OUT` geschrieben; der über die Rückrichtung des Adapters gemeldete Wert erscheint am Ausgang `IN` und wird mit `CNF` bestätigt.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Anforderung an den Adapter. Verbundene Daten: `OUT`

Ereignis-Ausgänge

- **CNF**: Bestätigung vom Adapter. Verbundene Daten: `IN`

Daten-Eingänge

- **OUT** (`SINT`): Daten, die an den Adapter gesendet werden

Daten-Ausgänge

- **IN** (`SINT`): Vom Adapter empfangene Daten

Adapter

- **AS2_OUT** (Plug): Bidirektionaler Adapter vom Typ `adapter::types::bidirectional::AS2` (Adapter-Ausgang)

Funktionsweise

1. `REQ` wird an `AS2_OUT.EO1` weitergeleitet, `OUT` an `AS2_OUT.DO1`.
2. `AS2_OUT.EI1` wird an `CNF` weitergeleitet, `AS2_OUT.DI1` an `IN`.

Technische Besonderheiten

- Reine Ereignis-/Datenverbindungen (`FBNetwork`), keine eigene Logik und kein interner Zustand
- Teil der Konvertierungsfamilie (analog zu `AR2`/`AB2`/`AE2`)

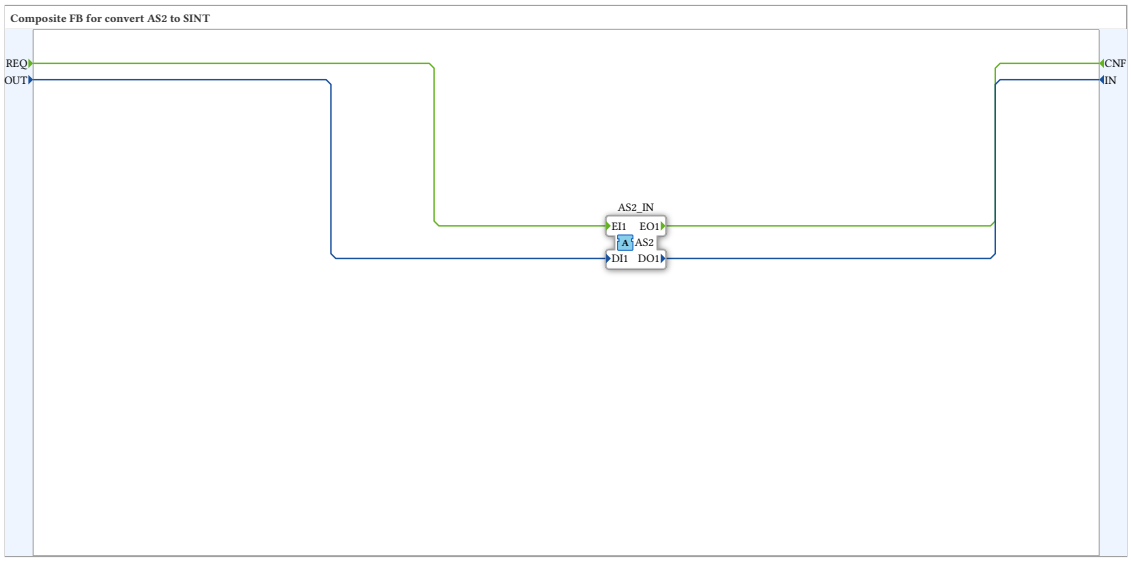
Anwendungsszenarien

- Anbindung von `SINT`-Werten aus klassischen Funktionsbaustein-Netzen an einen bidirektionalen `AS2`-Adapter

Fazit

`AS2_SINT_TO_S` überführt `SINT`-Ereignis-/Datenanschlüsse 1:1 in die beiden Richtungen eines `AS2`-Adapters.

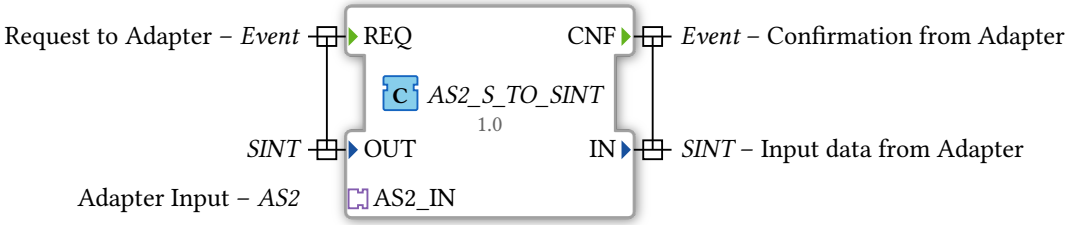
AS2_S_TO_SINT



• – ◦ • – ◦ • – ◦ •

Einleitung

`AS2_S_TO_SINT` ist ein Composite-FB, der zwischen einem `SINT`-Datenpaar (`OUT`/`IN`) und dem bidirektionalen Adapter `AS2` vermittelt. Im Gegensatz zur Plug-Variante `AS2_SINT_TO_R` besitzt er einen Adapter-**Socket** (Eingang). Er vervollständigt die Konvertierungsfamilie analog zu `AR2`/`AB2`/`AE2`.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Anforderung an den Adapter. Verbundene Daten: `OUT`.

Ereignis-Ausgänge

- `CNF`: Bestätigung vom Adapter. Verbundene Daten: `IN`.

Daten-Eingänge

- `OUT` (`SINT`): Wert, der zum Adapter gesendet wird.

Daten-Ausgänge

- `IN` (`SINT`): Vom Adapter empfangener Wert.

Adapter

Richtung	Name	Adaptertyp	Kommentar
Socket (Eingang)	`AS2_IN`	`adapter::types::bidirectional::AS2`	Adapter-Eingang.

Funktionsweise

Der Baustein enthält keine internen FBs, sondern nur Verbindungen:

- `REQ` ⟶ `AS2_IN.EI1` und `OUT` ⟶ `AS2_IN.DI1`: Ein `REQ`-Ereignis sendet den Wert `OUT` über den Adapter.

- `AS2_IN.EO1` ⟶ `CNF` und `AS2_IN.DO1` ⟶ `IN`: Ein vom Adapter kommendes Ereignis wird als `CNF` ausgegeben, der empfangene Wert erscheint an `IN`.

Technische Besonderheiten

- Reine Durchreichung ohne Logik und ohne internen Zustand.
- Der Datentyp ist `SINT` (vorzeichenbehaftete 8-Bit-Ganzzahl).
- Die Benennung `OUT`/`IN` ist aus Sicht des Adapters gewählt: `OUT` geht zum Adapter hinaus, `IN` kommt vom Adapter herein.

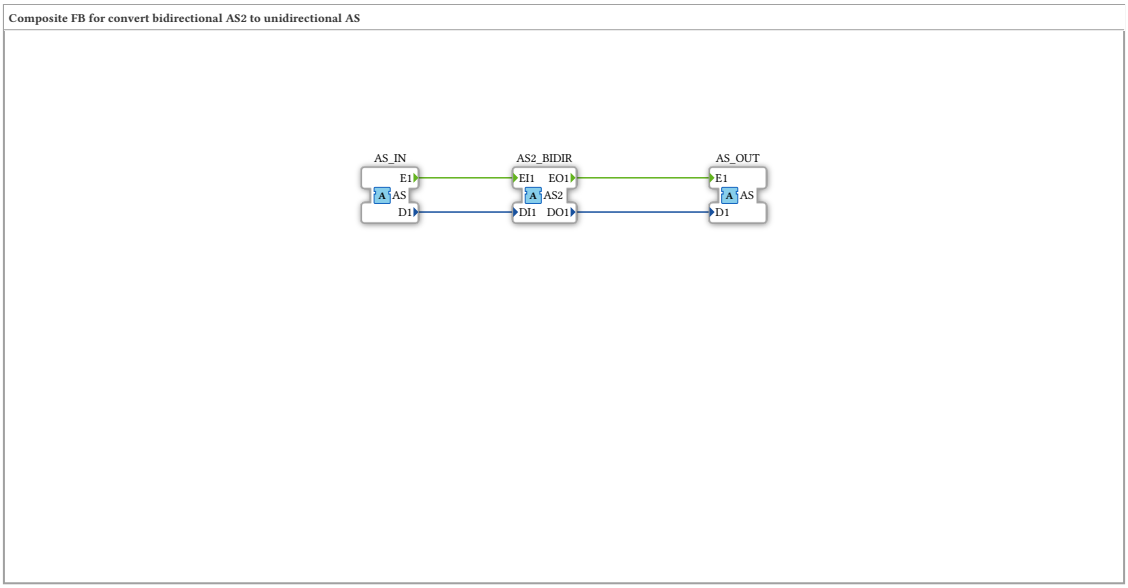
Anwendungsszenarien

- Einen `SINT`-Wert auf einen bidirektionalen `AS2`-Adapter ausgeben und gleichzeitig dessen Rückkanal als Datenausgang auslesen, wenn der Adapter als Socket vorliegt.

Zusammenfassung

`AS2_S_TO_SINT` verbindet Ereignisse und `SINT`-Daten 1:1 mit dem bidirektionalen Adapter `AS2` (Socket-Variante).

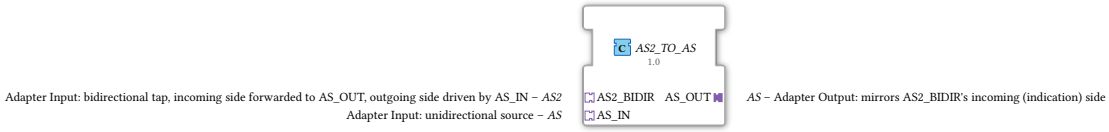
AS2_TO_AS



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock AS2_TO_AS ist das Spiegelbild von AS_TO_AS2: Hier steht der bidirektionale AS2-Abzweig als Socket auf der linken (Eingangs-)Seite, ergänzt um einen zweiten unidirektionalen AS-Socket und einen unidirektionalen AS-Plug.



Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Kommunikation läuft ausschließlich über die Adapter.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **AS2_BIDIR** (Socket): bidirektionaler Abzweig, dessen eingehende Seite an `AS_OUT` weitergereicht wird und dessen ausgehende Seite von `AS_IN` gespeist wird, Typ `adapter::types::bidirectional::AS2`
- **AS_IN** (Socket): unidirektionale SINT-Quelle, Typ `adapter::types::unidirectional::AS`
- **AS_OUT** (Plug): spiegelt die eingehende (Indikations-)Seite von `AS2_BIDIR` als unidirektionales Signal, Typ `adapter::types::unidirectional::AS`

Funktionsweise

Der Baustein arbeitet in zwei Richtungen gleichzeitig: Was der Socket `AS2_BIDIR` auf seiner Indikations-Seite empfängt (`EO1`/`DO1` – also was ein an AS2_BIDIR angeschlossener bidirektionaler Teilnehmer meldet), wird unverändert als eigenständiges unidirektionales Signal über den Plug `AS_OUT` ausgegeben. Umgekehrt wird alles, was am Socket `AS_IN` eintrifft (`E1`/`D1`), auf die Anfrage-Seite von `AS2_BIDIR` (`EI1`/`DI1`) veröffentlicht – AS2_BIDIR sendet also dasselbe Signal an den angeschlossenen bidirektionalen Teilnehmer weiter.

Technische Besonderheiten

- Drei statt zwei Adapter: Die vom Tap empfangene und die separat eingespeiste Signalrichtung werden bewusst getrennt gehalten, statt sie zu vermischen
- Reine Verdrahtung ohne Logik oder Zustand, jede Zielvariable hat genau einen Schreiber
- `AS_OUT` gibt nur wieder, was über `AS2_BIDIR` von einem angeschlossenen bidirektionalen Teilnehmer gemeldet wird – ohne einen solchen Teilnehmer bleibt `AS_OUT` inaktiv

Zustandsübersicht

Der Baustein ist zustandslos:

- AS2_BIDIR.EO1 ⟶ AS_OUT.E1, AS2_BIDIR.DO1 ⟶ AS_OUT.D1
- AS_IN.E1 ⟶ AS2_BIDIR.EI1, AS_IN.D1 ⟶ AS2_BIDIR.DI1

Anwendungsszenarien

- Auslesen des Status eines bidirektionalen AS2-Teilnehmers als einfaches unidirektionales AS-Signal, kombiniert mit dem Einspeisen eines separaten unidirektionalen Kommandosignals
- Systeme, in denen ein bidirektionaler Teilnehmer überwacht werden soll, ohne die restliche Steuerung auf AS2 umzustellen
- Brücke zwischen einer bidirektionalen AS2-Zone und einer unidirektionalen AS-Zone

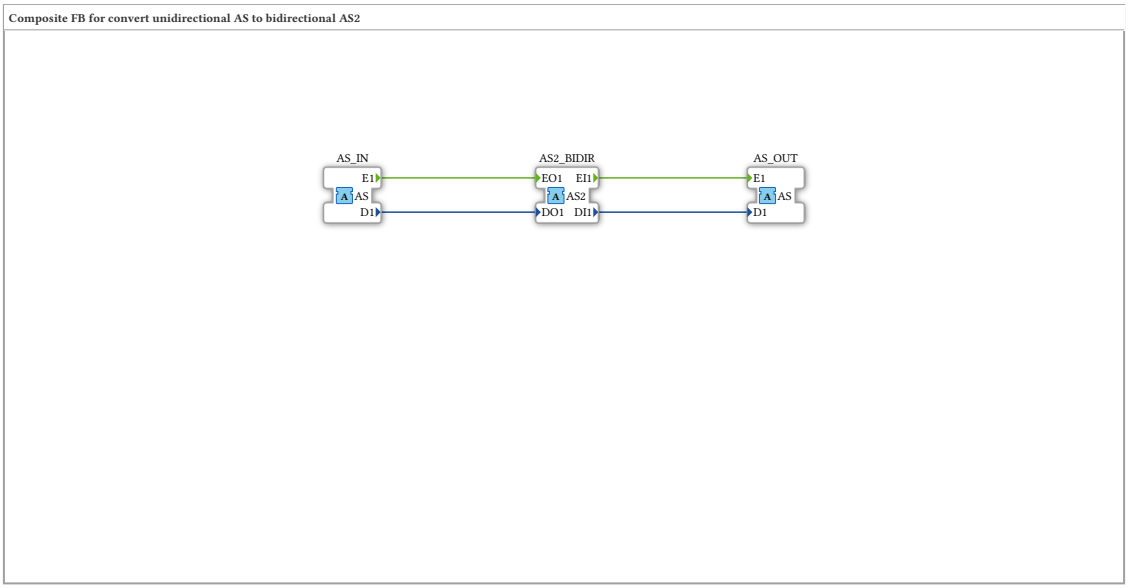
⚖️ Vergleich mit ähnlichen Bausteinen

AS_TO_AS2 ist das Spiegelbild dieses Bausteins – dort steht `AS2_BIDIR` als Plug auf der rechten Seite.

Fazit

AS2_TO_AS ist kein einfacher Konverter, sondern ein Abzweigpunkt: Es macht den Zustand eines bidirektionalen AS2-Teilnehmers als eigenständiges unidirektionales AS-Signal sichtbar und speist gleichzeitig ein separates unidirektionales Signal in die bidirektionale Seite ein.

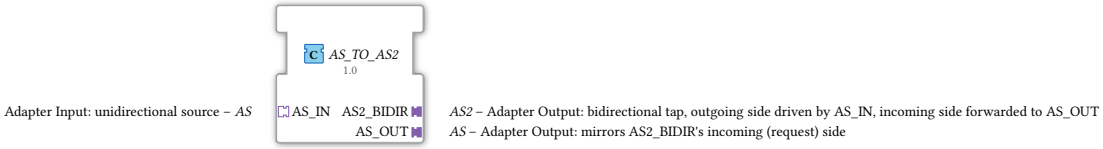
AS_TO_AS2



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `AS_TO_AS2` hängt einen bidirektionalen `AS2`-Abzweig („Tap“) in eine unidirektionale `AS`-Kette (SINT) ein. Er nutzt dafür **drei** Adapter: Ein eingehendes unidirektionales AS-Signal wird auf die bidirektionale AS2-Schnittstelle veröffentlicht, und was AS2 auf seiner Anfrage-Seite zurückliefert, wird als eigenständiges unidirektionales AS-Signal wieder ausgegeben.



Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **AS_IN** (Socket): unidirektionale SINT-Quelle, Typ `adapter::types::unidirectional::AS`
- **AS2_BIDIR** (Plug): bidirektionaler Abzweig, dessen ausgehende Seite von `AS_IN` gespeist wird und dessen eingehende Seite an `AS_OUT` weitergereicht wird, Typ `adapter::types::bidirectional::AS2`
- **AS_OUT** (Plug): spiegelt die eingehende (Anfrage-)Seite von `AS2_BIDIR` als unidirektionales Signal, Typ `adapter::types::unidirectional::AS`

Funktionsweise

Der Baustein arbeitet in zwei Richtungen gleichzeitig: Was am Socket `AS_IN` eintrifft (`E1`/`D1`), wird direkt auf die ausgehende Seite von `AS2_BIDIR` (`EO1`/`DO1`) veröffentlicht. Umgekehrt wird alles, was `AS2_BIDIR`

auf seiner Anfrage-Seite empfängt (EI1/DI1), unverändert als eigenständiges unidirektionales Signal über den Plug AS_OUT ausgegeben. AS2_BIDIR fungiert damit als Abzweigpunkt, der die unidirektionale Kette nicht unterbricht, sondern um eine bidirektionale Beobachtungs-/Einspeisemöglichkeit erweitert.

Technische Besonderheiten

- Drei statt zwei Adapter: Die eingehende und die vom Tap zurückkommende Signalrichtung werden bewusst getrennt gehalten
- Reine Verdrahtung ohne Logik oder Zustand, jede Zielvariable hat genau einen Schreiber
- AS_OUT gibt nur wieder, was über AS2_BIDIR von einem angeschlossenen bidirektionalen Teilnehmer zurückkommt; ohne einen solchen Teilnehmer bleibt AS_OUT inaktiv

Zustandsübersicht

Der Baustein ist zustandslos:

- AS_IN.E1 ⟶ AS2_BIDIR.EO1, AS_IN.D1 ⟶ AS2_BIDIR.DO1
- AS2_BIDIR.EI1 ⟶ AS_OUT.E1, AS2_BIDIR.DI1 ⟶ AS_OUT.D1

Anwendungsszenarien

- Nachträgliches Einhängen eines bidirektionalen Diagnose- oder Steuerpunkts in eine bestehende unidirektionale AS-Signalkette, ohne diese zu unterbrechen
- Brücke zwischen einer unidirektionalen AS-Zone und einer bidirektionalen AS2-Zone

⚖️ Vergleich mit ähnlichen Bausteinen

Das Gegenstück in umgekehrter Richtung ist der Baustein AS2_TO_AS (dort steht AS2_BIDIR als Socket auf der linken Seite). Die Bausteine dieser Konvertierungsfamilie existieren analog für die Adapter AR/AR2, AB/AB2 und AE/AE2.

Fazit

AS_TO_AS2 ist kein einfacher Konverter, sondern ein Abzweigpunkt: Er macht ein unidirektionales AS-Signal an einem bidirektionalen AS2-Punkt sichtbar, ohne die ursprüngliche Signalkette zu unterbrechen, und gibt das dort empfangene Anfrage-Signal als eigenständiges AS-Signal weiter.

STRING

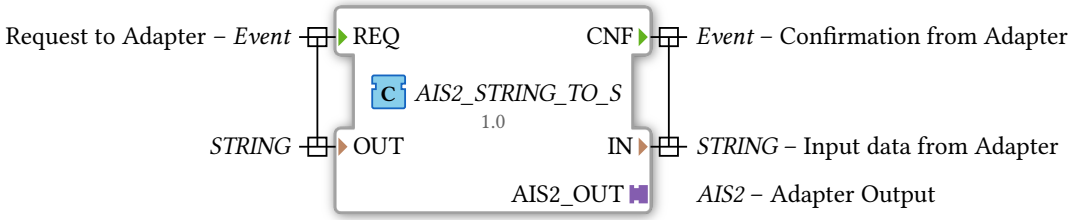
AIS2_STRING_TO_S

• – ◦ • – ◦ • – ◦ •

Einleitung

AIS2_STRING_TO_S ist ein Composite-Funktionsblock, der einen bidirektionalen AIS2-Adapter (Plug AIS2_OUT) mit einem klassischen Ereignis-/Daten-Interface vom Typ STRING verbindet. Das Ereignis REQ

mit dem Dateneingang `OUT` wird auf den Adapter geschrieben, Ereignis und Wert aus dem Adapter erscheinen als `CNF` mit dem Datenausgang `IN`. Der Baustein gehört zur Konvertierungsfamilie (analog zu `AR2`/`AB2`/`AE2`).



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Anforderung an den Adapter (zugeordnetes Datum: `OUT`)

Ereignis-Ausgänge

- **CNF**: Bestätigung vom Adapter (zugeordnetes Datum: `IN`)

Daten-Eingänge

- **OUT** (`STRING`): Wert, der zum Adapter gesendet wird

Daten-Ausgänge

- **IN** (`STRING`): Vom Adapter empfangener Wert (Input data from Adapter)

Adapter

- **AIS2_OUT** (Plug): Bidirektionaler Adapter vom Typ `adapter::types::bidirectional::AIS2` (Adapter-Ausgang)

Funktionsweise

1. **Senden**: `REQ` wird auf `AIS2_OUT.EO1` und `OUT` auf `AIS2_OUT.DO1` geleitet.
2. **Empfangen**: `AIS2_OUT.EI1` wird auf `CNF` und `AIS2_OUT.DI1` auf `IN` geleitet.

Technische Besonderheiten

- Reine Ereignis-/Datenverbindungen (`FBNetwork`), keine eigene Logik und kein interner Zustand
- Der Datentyp ist auf beiden Seiten `STRING`, es findet keine Wertumrechnung statt
- Version 1.0 (2026-09-12): Konvertierungsfamilie vervollständigt

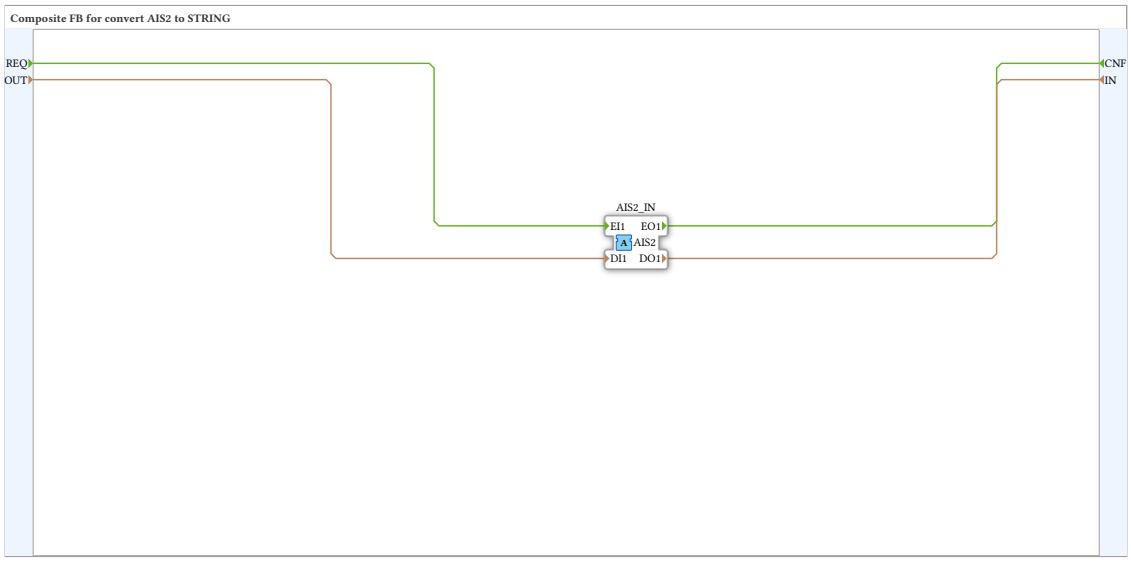
Anwendungsszenarien

- Anbindung eines Bausteins mit einfachem `REQ`/`CNF`-Interface an eine bidirektionale `AIS2`-Adapterverbindung
- Brücke zwischen Ereignis-/Datenports und Adapter bei `STRING`-Werten

Fazit

`AIS2_STRING_TO_S` übersetzt ein einfaches `STRING`-Ereignis-/Daten-Interface 1:1 in einen bidirektionalen `AIS2`-Adapter und zurück.

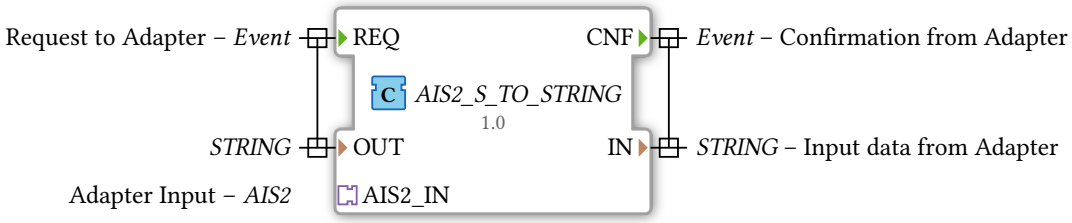
AIS2_S_TO_STRING



• – ○ • – ○ • – ○ •

Einleitung

`AIS2_S_TO_STRING` ist ein Composite-Funktionsblock, der den bidirektionalen Adapter AIS2 auf einfache Ereignis- und STRING-Datenports abbildet. Er gehört zur Konvertierungsfamilie analog zu AR2/AB2/AE2.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Request to Adapter): Sendet die Daten `OUT` über den Adapter (assoziiert mit `OUT`)

Ereignis-Ausgänge

- **CNF** (Confirmation from Adapter): Wird ausgelöst, wenn der Adapter ein Ereignis auf seiner eingehenden Seite liefert (assoziiert mit `IN`)

Daten-Eingänge

- **OUT** (STRING): Daten, die an den Adapter gesendet werden

Daten-Ausgänge

- **IN** (STRING): Vom Adapter empfangene Daten

Adapter

- **AIS2_IN** (Socket): Adapter vom Typ `adapter::types::bidirectional::AIS2`

Funktionsweise

Der Baustein besteht nur aus Verdrahtung ohne eigene Logik:

- `REQ` ⟶ `AIS2_IN.EI1` und `OUT` ⟶ `AIS2_IN.DI1`: Der Anfrage-Ereignisstrom samt Daten wird auf die ausgehende Seite des Adapters veröffentlicht.
- `AIS2_IN.EO1` ⟶ `CNF` und `AIS2_IN.DO1` ⟶ `IN`: Was auf der eingehenden Seite des Adapters gemeldet wird, erscheint als Ereignis `CNF` mit den Daten `IN`.

Technische Besonderheiten

- Zustandsloser Composite-Baustein ohne Algorithmen
- Jede Zielvariable hat genau einen Schreiber
- Die Adapterdaten werden unverändert durchgereicht, es findet keine Wertumwandlung statt

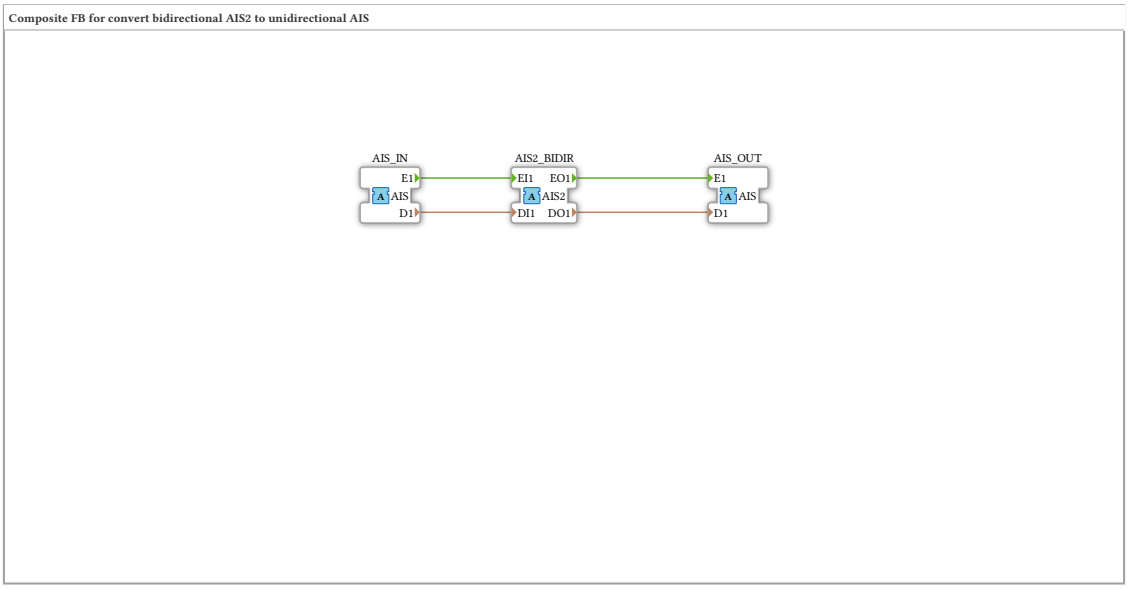
Anwendungsszenarien

- Anbindung klassischer Ereignis-/Datenports an einen bidirektionalen AIS2-Adapter
- Einbindung von Bausteinen mit STRING-Ports in adapterbasierte Netzwerke

Fazit

`AIS2_S_TO_STRING` setzt den bidirektionalen AIS2-Adapter ohne Logik auf die Ports `REQ`/`OUT` und `CNF`/`IN` um.

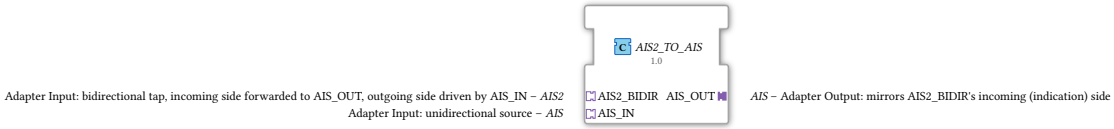
AIS2_TO_AIS



• – ○ • – ○ • – ○ •

Einleitung

`AIS2_TO_AIS` ist ein Composite-Funktionsblock zur Konvertierung eines **bidirektionalen** `AIS2`-Adapters in einen **unidirektionalen** `AIS`-Adapter. Die eingehende Seite (`EO1`/`DO1`) des bidirektionalen Adapters wird als einfaches unidirektionales Signal über `AIS_OUT` ausgegeben; über `AIS_IN` wird ein Wert auf die ausgehende Seite (`EI1`/`DI1`) des bidirektionalen Adapters geschrieben.



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden – Ereignisse kommen über die Adapter-Sockets/-Plugs

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

- **AIS2_BIDIR** (Socket): Bidirektionaler Adapter vom Typ `adapter::types::bidirectional::AIS2` – die eingehende Seite wird an `AIS_OUT` weitergeleitet, die ausgehende Seite wird von `AIS_IN` getrieben
- **AIS_IN** (Socket): Unidirektionaler Adapter vom Typ `adapter::types::unidirectional::AIS` – unidirektionale Quelle
- **AIS_OUT** (Plug): Unidirektionaler Adapter vom Typ `adapter::types::unidirectional::AIS` – spiegelt die eingehende (Indikations-)Seite von `AIS2_BIDIR`

Funktionsweise

1. `AIS2_BIDIR.EO1` wird an `AIS_OUT.E1` und `AIS2_BIDIR.DO1` an `AIS_OUT.D1` weitergeleitet.
2. `AIS_IN.E1` wird an `AIS2_BIDIR.EI1` und `AIS_IN.D1` an `AIS2_BIDIR.DI1` weitergeleitet.

Technische Besonderheiten

- Reine Ereignis-/Datenverbindungen (`FBNetwork`), keine eigene Logik und kein interner Zustand
- Teil der Konvertierungsfamilie (analog zu `AR2`/`AB2`/`AE2`); der Datentyp der Daten ist `STRING`

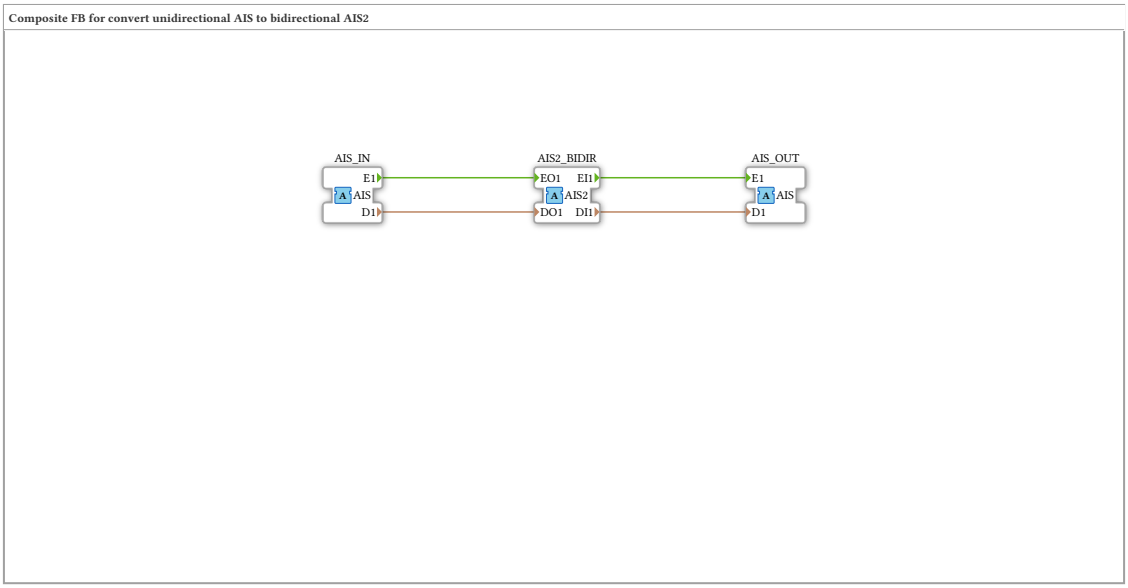
Anwendungsszenarien

- Anbindung eines unidirektionalen `AIS`-Verbrauchers bzw. -Erzeugers an einen bidirektionalen `AIS2`-Adapter

Fazit

`AIS2_TO_AIS` trennt die beiden Richtungen eines bidirektionalen `AIS2`-Adapters in einen unidirektionalen Ausgang (`AIS_OUT`) und einen unidirektionalen Eingang (`AIS_IN`).

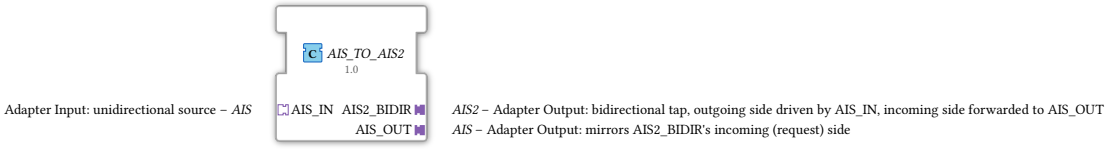
AIS_TO_AIS2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AIS_TO_AIS2` hängt einen bidirektionalen `AIS2`-Abzweig („Tap“) in eine unidirektionale `AIS`-Kette (STRING) ein. Er nutzt dafür **drei** Adapter: Ein eingehendes unidirektionales AIS-Signal wird auf die bidirektionale AIS2-Schnittstelle veröffentlicht, und was AIS2 auf seiner Anfrage-Seite zurückliefert, wird als eigenständiges unidirektionales AIS-Signal wieder ausgegeben.



Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **AIS_IN** (Socket): unidirektionale STRING-Quelle, Typ `adapter::types::unidirectional::AIS`
- **AIS2_BIDIR** (Plug): bidirektionaler Abzweig, dessen ausgehende Seite von `AIS_IN` gespeist wird und dessen eingehende Seite an `AIS_OUT` weitergereicht wird, Typ `adapter::types::bidirectional::AIS2`
- **AIS_OUT** (Plug): spiegelt die eingehende (Anfrage-)Seite von `AIS2_BIDIR` als unidirektionales Signal, Typ `adapter::types::unidirectional::AIS`

Funktionsweise

Der Baustein arbeitet in zwei Richtungen gleichzeitig: Was am Socket `AIS_IN` eintrifft (`E1`/`D1`), wird direkt auf die ausgehende Seite von `AIS2_BIDIR` (`EO1`/`DO1`) veröffentlicht. Umgekehrt wird alles, was `AIS2_BIDIR` auf seiner Anfrage-Seite empfängt (`EI1`/`DI1`), unverändert als eigenständiges unidirektionales Signal über den Plug `AIS_OUT` ausgegeben. `AIS2_BIDIR` fungiert damit als Abzweigpunkt, der die unidirektionale Kette nicht unterbricht, sondern um eine bidirektionale Beobachtungs-/Einspeisemöglichkeit erweitert.

Technische Besonderheiten

- Drei statt zwei Adapter: Die eingehende und die vom Tap zurückkommende Signalrichtung werden bewusst getrennt gehalten
- Reine Verdrahtung ohne Logik oder Zustand, jede Zielvariable hat genau einen Schreiber
- `AIS_OUT` gibt nur wieder, was über `AIS2_BIDIR` von einem angeschlossenen bidirektionalen Teilnehmer zurückkommt; ohne einen solchen Teilnehmer bleibt `AIS_OUT` inaktiv

Zustandsübersicht

Der Baustein ist zustandslos:

- AIS_IN.E1 ⟶ AIS2_BIDIR.EO1, AIS_IN.D1 ⟶ AIS2_BIDIR.DO1
- AIS2_BIDIR.EI1 ⟶ AIS_OUT.E1, AIS2_BIDIR.DI1 ⟶ AIS_OUT.D1

Anwendungsszenarien

- Nachträgliches Einhängen eines bidirektionalen Diagnose- oder Steuerpunkts in eine bestehende unidirektionale AIS-Signalkette, ohne diese zu unterbrechen
- Brücke zwischen einer unidirektionalen AIS-Zone und einer bidirektionalen AIS2-Zone

⚖️ Vergleich mit ähnlichen Bausteinen

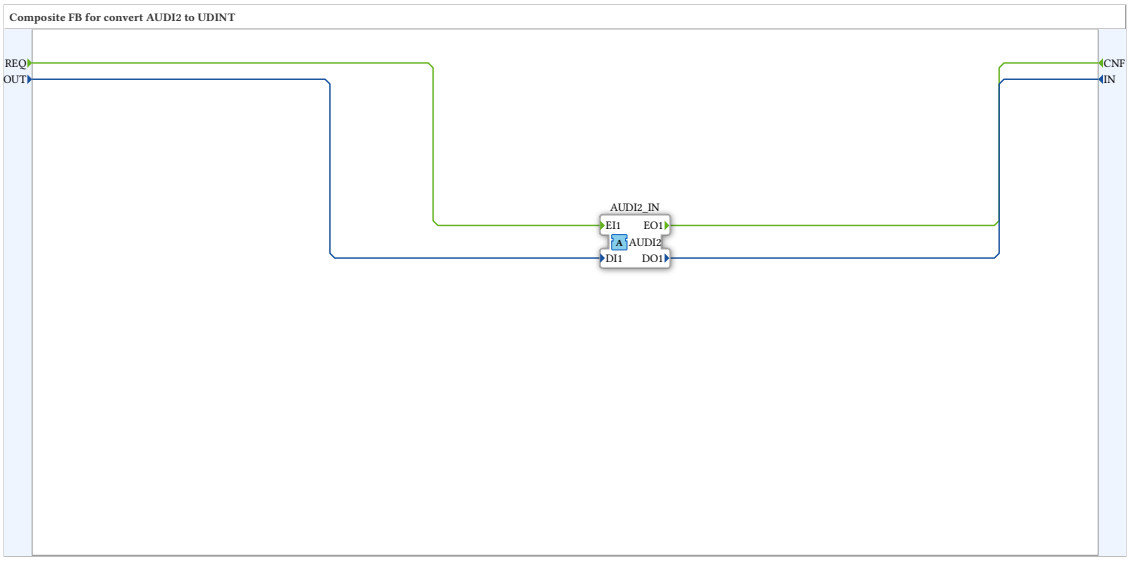
Das Gegenstück in umgekehrter Richtung ist der Baustein `AIS2_TO_AIS` (dort steht `AIS2_BIDIR` als Socket auf der linken Seite). Die Bausteine dieser Konvertierungsfamilie existieren analog für die Adapter `AR`/`AR2`, `AB`/`AB2` und `AE`/`AE2`.

Fazit

`AIS_TO_AIS2` ist kein einfacher Konverter, sondern ein Abzweigpunkt: Er macht ein unidirektionales AIS-Signal an einem bidirektionalen AIS2-Punkt sichtbar, ohne die ursprüngliche Signalkette zu unterbrechen, und gibt das dort empfangene Anfrage-Signal als eigenständiges AIS-Signal weiter.

UDINT

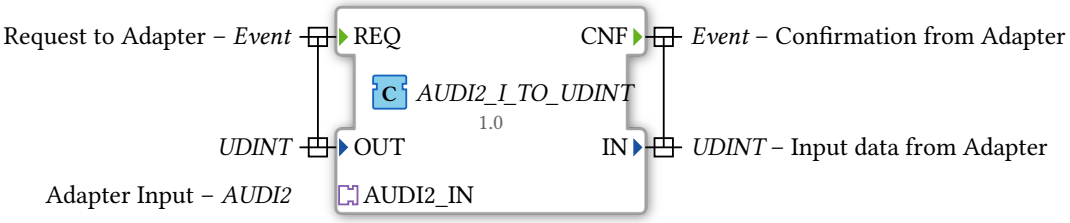
AUDI2_I_TO_UDINT



• – ○ • – ○ • – ○ •

Einleitung

`AUDI2_I_TO_UDINT` ist ein Composite-Funktionsblock, der den bidirektionalen Adapter AUDI2 auf einfache Ereignis- und UDINT-Datenports abbildet. Er gehört zur Konvertierungsfamilie analog zu AR2/AB2/AE2.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Request to Adapter): Sendet die Daten `OUT` über den Adapter (assoziiert mit `OUT`)

Ereignis-Ausgänge

- **CNF** (Confirmation from Adapter): Wird ausgelöst, wenn der Adapter ein Ereignis auf seiner eingehenden Seite liefert (assoziiert mit `IN`)

Daten-Eingänge

- **OUT** (UDINT): Daten, die an den Adapter gesendet werden

Daten-Ausgänge

- **IN** (UDINT): Vom Adapter empfangene Daten

Adapter

- **AUDI2_IN** (Socket): Adapter vom Typ `adapter::types::bidirectional::AUDI2`

Funktionsweise

Der Baustein besteht nur aus Verdrahtung ohne eigene Logik:

- `REQ` ⟶ `AUDI2_IN.EI1` und `OUT` ⟶ `AUDI2_IN.DI1`: Der Anfrage-Ereignisstrom samt Daten wird auf die ausgehende Seite des Adapters veröffentlicht.
- `AUDI2_IN.EO1` ⟶ `CNF` und `AUDI2_IN.DO1` ⟶ `IN`: Was auf der eingehenden Seite des Adapters gemeldet wird, erscheint als Ereignis `CNF` mit den Daten `IN`.

Technische Besonderheiten

- Zustandsloser Composite-Baustein ohne Algorithmen
- Jede Zielvariable hat genau einen Schreiber
- Die Adapterdaten werden unverändert durchgereicht, es findet keine Wertumwandlung statt

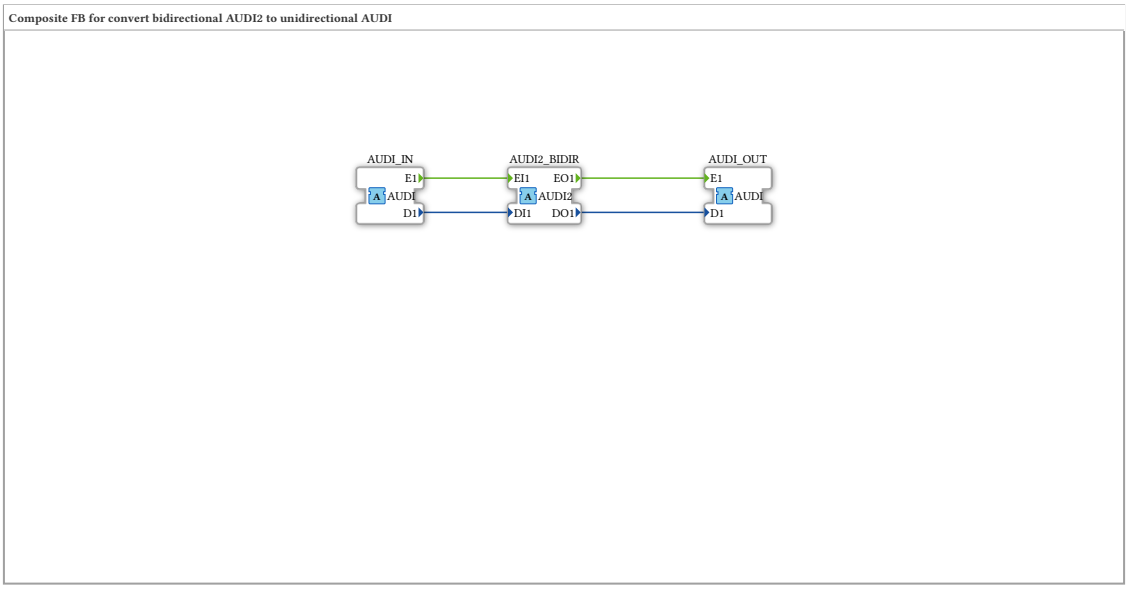
Anwendungsszenarien

- Anbindung klassischer Ereignis-/Datenports an einen bidirektionalen AUDI2-Adapter
- Einbindung von Bausteinen mit UDINT-Ports in adapterbasierte Netzwerke

Fazit

`AUDI2_I_TO_UDINT` setzt den bidirektionalen AUDI2-Adapter ohne Logik auf die Ports `REQ`/`OUT` und `CNF`/`IN` um.

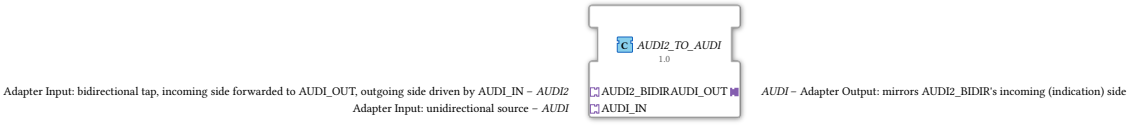
AUDI2_TO_AUDI



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock AUDI2_TO_AUDI ist das Spiegelbild von AUDI_TO_AUDI2: Hier steht der bidirektionale AUDI2-Abzweig als Socket auf der linken (Eingangs-)Seite, ergänzt um einen zweiten unidirektionalen AUDI-Socket und einen unidirektionalen AUDI-Plug.



Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge – die Kommunikation läuft ausschließlich über die Adapter.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **AUDI2_BIDIR** (Socket): bidirektionaler Abzweig, dessen eingehende Seite an `AUDI_OUT` weitergereicht wird und dessen ausgehende Seite von `AUDI_IN` gespeist wird, Typ `adapter::types::bidirectional::AUDI2`
- **AUDI_IN** (Socket): unidirektionale UDINT-Quelle, Typ `adapter::types::unidirectional::AUDI`
- **AUDI_OUT** (Plug): spiegelt die eingehende (Indikations-)Seite von `AUDI2_BIDIR` als unidirektionales Signal, Typ `adapter::types::unidirectional::AUDI`

Funktionsweise

Der Baustein arbeitet in zwei Richtungen gleichzeitig: Was der Socket `AUDI2_BIDIR` auf seiner Indikations-Seite empfängt (`EO1`/`DO1` – also was ein an AUDI2_BIDIR angeschlossener bidirektionaler Teilnehmer meldet), wird unverändert als eigenständiges unidirektionales Signal über den Plug `AUDI_OUT` ausgegeben. Umgekehrt wird alles, was am Socket `AUDI_IN` eintrifft (`E1`/`D1`), auf die Anfrage-Seite von `AUDI2_BIDIR` (`EI1`/`DI1`) veröffentlicht – AUDI2_BIDIR sendet also dasselbe Signal an den angeschlossenen bidirektionalen Teilnehmer weiter.

Technische Besonderheiten

- Drei statt zwei Adapter: Die vom Tap empfangene und die separat eingespeiste Signalrichtung werden bewusst getrennt gehalten, statt sie zu vermischen
- Reine Verdrahtung ohne Logik oder Zustand, jede Zielvariable hat genau einen Schreiber
- `AUDI_OUT` gibt nur wieder, was über `AUDI2_BIDIR` von einem angeschlossenen bidirektionalen Teilnehmer gemeldet wird – ohne einen solchen Teilnehmer bleibt `AUDI_OUT` inaktiv

Zustandsübersicht

Der Baustein ist zustandslos:

- AUDI2_BIDIR.EO1 ⟶ AUDI_OUT.E1, AUDI2_BIDIR.DO1 ⟶ AUDI_OUT.D1
- AUDI_IN.E1 ⟶ AUDI2_BIDIR.EI1, AUDI_IN.D1 ⟶ AUDI2_BIDIR.DI1

Anwendungsszenarien

- Auslesen des Status eines bidirektionalen AUDI2-Teilnehmers als einfaches unidirektionales AUDI-Signal, kombiniert mit dem Einspeisen eines separaten unidirektionalen Kommandosignals
- Systeme, in denen ein bidirektionaler Teilnehmer überwacht werden soll, ohne die restliche Steuerung auf AUDI2 umzustellen
- Brücke zwischen einer bidirektionalen AUDI2-Zone und einer unidirektionalen AUDI-Zone

⚖️ Vergleich mit ähnlichen Bausteinen

AUDI_TO_AUDI2 ist das Spiegelbild dieses Bausteins – dort steht `AUDI2_BIDIR` als Plug auf der rechten Seite.

Fazit

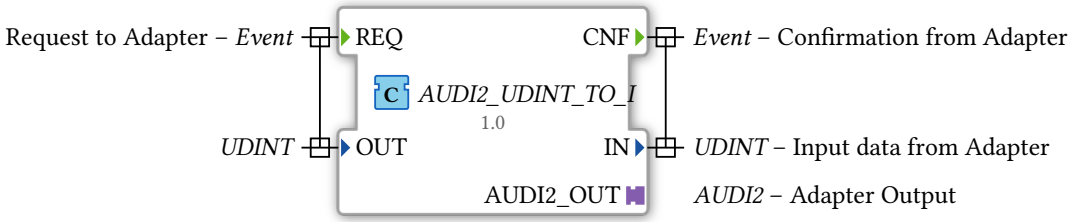
AUDI2_TO_AUDI ist kein einfacher Konverter, sondern ein Abzweigpunkt: Es macht den Zustand eines bidirektionalen AUDI2-Teilnehmers als eigenständiges unidirektionales AUDI-Signal sichtbar und speist gleichzeitig ein separates unidirektionales Signal in die bidirektionale Seite ein.

AUDI2_UDINT_TO_I

• – ◦ • – ◦ • – ◦ •

Einleitung

`AUDI2_UDINT_TO_I` ist ein Composite-Funktionsblock, der zwischen klassischen Ereignis-/Datenanschlüssen vom Typ `UDINT` und einem **bidirektionalen** `AUDI2`-Adapter vermittelt. Der Eingang `OUT` wird bei `REQ` auf die ausgehende Seite des Adapter-Plugs `AUDI2_OUT` geschrieben; der über die Rückrichtung des Adapters gemeldete Wert erscheint am Ausgang `IN` und wird mit `CNF` bestätigt.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Anforderung an den Adapter. Verbundene Daten: `OUT`

Ereignis-Ausgänge

- **CNF**: Bestätigung vom Adapter. Verbundene Daten: `IN`

Daten-Eingänge

- **OUT** (`UDINT`): Daten, die an den Adapter gesendet werden

Daten-Ausgänge

- **IN** (`UDINT`): Vom Adapter empfangene Daten

Adapter

- **AUDI2_OUT** (Plug): Bidirektionaler Adapter vom Typ `adapter::types::bidirectional::AUDI2` (Adapter-Ausgang)

Funktionsweise

1. `REQ` wird an `AUDI2_OUT.EO1` weitergeleitet, `OUT` an `AUDI2_OUT.DO1`.
2. `AUDI2_OUT.EI1` wird an `CNF` weitergeleitet, `AUDI2_OUT.DI1` an `IN`.

Technische Besonderheiten

- Reine Ereignis-/Datenverbindungen (`FBNetwork`), keine eigene Logik und kein interner Zustand
- Teil der Konvertierungsfamilie (analog zu `AR2`/`AB2`/`AE2`)

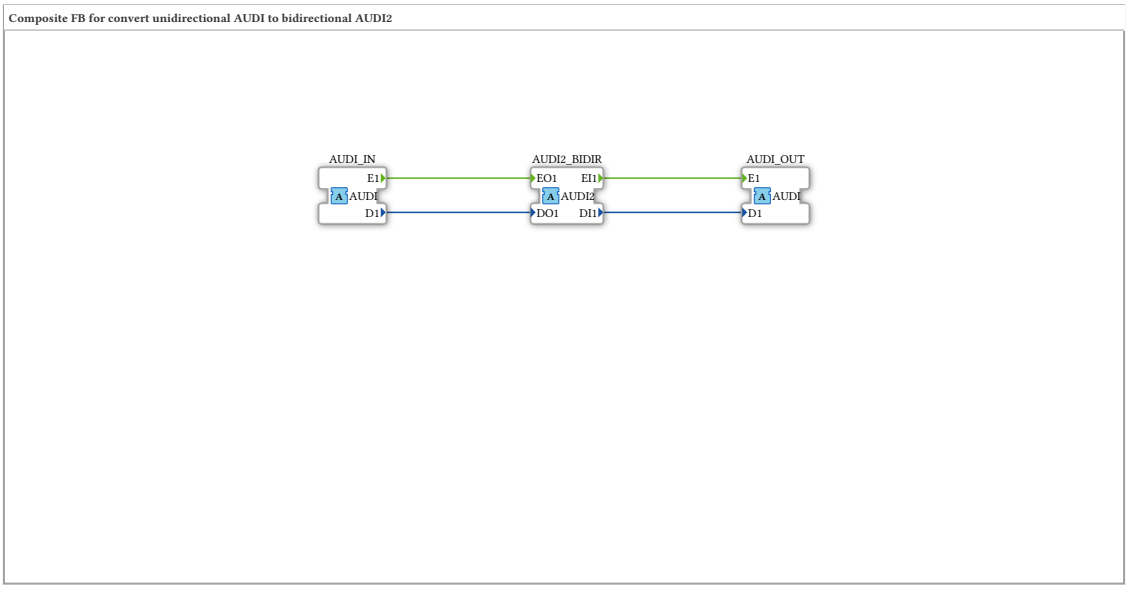
Anwendungsszenarien

- Anbindung von `UDINT`-Werten aus klassischen Funktionsbaustein-Netzen an einen bidirektionalen `AUDI2`-Adapter

Fazit

`AUDI2_UDINT_TO_I` überführt `UDINT`-Ereignis-/Datenanschlüsse 1:1 in die beiden Richtungen eines `AUDI2`-Adapters.

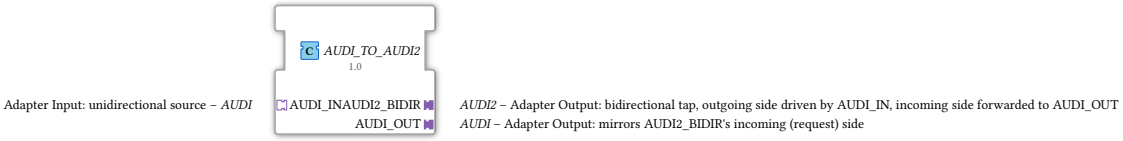
AUDI_TO_AUDI2



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `AUDI_TO_AUDI2` hängt einen bidirektionalen `AUDI2`-Abzweig („Tap“) in eine unidirektionale `AUDI`-Kette (UDINT) ein. Er nutzt dafür **drei** Adapter: Ein eingehendes unidirektionales AUDI-Signal wird auf die bidirektionale AUDI2-Schnittstelle veröffentlicht, und was AUDI2 auf seiner Anfrage-Seite zurückliefert, wird als eigenständiges unidirektionales AUDI-Signal wieder ausgegeben.



Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **AUDI_IN** (Socket): unidirektionale UDINT-Quelle, Typ `adapter::types::unidirectional::AUDI`
- **AUDI2_BIDIR** (Plug): bidirektionaler Abzweig, dessen ausgehende Seite von `AUDI_IN` gespeist wird und dessen eingehende Seite an `AUDI_OUT` weitergereicht wird, Typ `adapter::types::bidirectional::AUDI2`
- **AUDI_OUT** (Plug): spiegelt die eingehende (Anfrage-)Seite von `AUDI2_BIDIR` als unidirektionales Signal, Typ `adapter::types::unidirectional::AUDI`

Funktionsweise

Der Baustein arbeitet in zwei Richtungen gleichzeitig: Was am Socket `AUDI_IN` eintrifft (`E1`/`D1`), wird direkt auf die ausgehende Seite von `AUDI2_BIDIR` (`EO1`/`DO1`) veröffentlicht. Umgekehrt wird alles, was `AUDI2_BIDIR` auf seiner Anfrage-Seite empfängt (`EI1`/`DI1`), unverändert als eigenständiges unidirektionales Signal über den Plug `AUDI_OUT` ausgegeben. `AUDI2_BIDIR` fungiert damit als Abzweigpunkt, der die unidirektionale Kette nicht unterbricht, sondern um eine bidirektionale Beobachtungs-/Einspeisemöglichkeit erweitert.

Technische Besonderheiten

- Drei statt zwei Adapter: Die eingehende und die vom Tap zurückkommende Signalrichtung werden bewusst getrennt gehalten
- Reine Verdrahtung ohne Logik oder Zustand, jede Zielvariable hat genau einen Schreiber
- `AUDI_OUT` gibt nur wieder, was über `AUDI2_BIDIR` von einem angeschlossenen bidirektionalen Teilnehmer zurückkommt; ohne einen solchen Teilnehmer bleibt `AUDI_OUT` inaktiv

Zustandsübersicht

Der Baustein ist zustandslos:

- AUDI_IN.E1 ⟶ AUDI2_BIDIR.EO1, AUDI_IN.D1 ⟶ AUDI2_BIDIR.DO1
- AUDI2_BIDIR.EI1 ⟶ AUDI_OUT.E1, AUDI2_BIDIR.DI1 ⟶ AUDI_OUT.D1

Anwendungsszenarien

- Nachträgliches Einhängen eines bidirektionalen Diagnose- oder Steuerpunkts in eine bestehende unidirektionale AUDI-Signalkette, ohne diese zu unterbrechen
- Brücke zwischen einer unidirektionalen AUDI-Zone und einer bidirektionalen AUDI2-Zone

⚖️ Vergleich mit ähnlichen Bausteinen

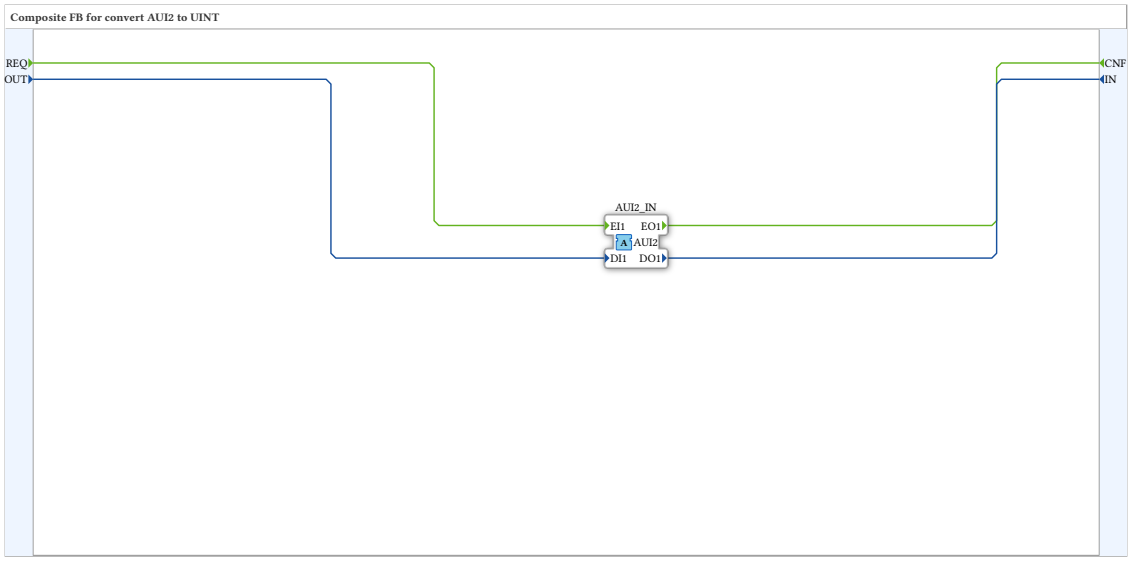
Das Gegenstück in umgekehrter Richtung ist der Baustein `AUDI2_TO_AUDI` (dort steht `AUDI2_BIDIR` als Socket auf der linken Seite). Die Bausteine dieser Konvertierungsfamilie existieren analog für die Adapter `AR`/`AR2`, `AB`/`AB2` und `AE`/`AE2`.

Fazit

`AUDI_TO_AUDI2` ist kein einfacher Konverter, sondern ein Abzweigpunkt: Er macht ein unidirektionales AUDI-Signal an einem bidirektionalen AUDI2-Punkt sichtbar, ohne die ursprüngliche Signalkette zu unterbrechen, und gibt das dort empfangene Anfrage-Signal als eigenständiges AUDI-Signal weiter.

UINT

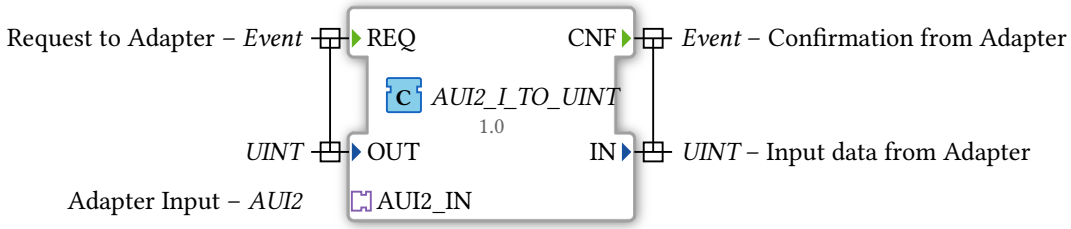
AUI2_I_TO_UINT



• – ◦ • – ◦ • – ◦ •

Einleitung

`AUI2_I_TO_UINT` ist ein Composite-FB, der zwischen einem `UINT`-Datenpaar (`OUT`/`IN`) und dem bidirektionalen Adapter `AUI2` vermittelt. Im Gegensatz zur Plug-Variante `AUI2_UINT_TO_R` besitzt er einen Adapter-**Socket** (Eingang). Er vervollständigt die Konvertierungsfamilie analog zu `AR2`/`AB2`/`AE2`.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Anforderung an den Adapter. Verbundene Daten: `OUT`.

Ereignis-Ausgänge

- `CNF`: Bestätigung vom Adapter. Verbundene Daten: `IN`.

Daten-Eingänge

- `OUT` (`UINT`): Wert, der zum Adapter gesendet wird.

Daten-Ausgänge

- `IN` (`UINT`): Vom Adapter empfangener Wert.

Adapter

Richtung	Name	Adaptertyp	Kommentar
Socket (Eingang)	`AUI2_IN`	`adapter::types::bidirectional::AUI2`	Adapter-Eingang.

Funktionsweise

Der Baustein enthält keine internen FBs, sondern nur Verbindungen:

- `REQ` ⟶ `AUI2_IN.EI1` und `OUT` ⟶ `AUI2_IN.DI1`: Ein `REQ`-Ereignis sendet den Wert `OUT` über den Adapter.
- `AUI2_IN.EO1` ⟶ `CNF` und `AUI2_IN.DO1` ⟶ `IN`: Ein vom Adapter kommendes Ereignis wird als `CNF` ausgegeben, der empfangene Wert erscheint an `IN`.

Technische Besonderheiten

- Reine Durchreichung ohne Logik und ohne internen Zustand.
- Der Datentyp ist `UINT` (vorzeichenlose 16-Bit-Ganzzahl).
- Die Benennung `OUT`/`IN` ist aus Sicht des Adapters gewählt: `OUT` geht zum Adapter hinaus, `IN` kommt vom Adapter herein.

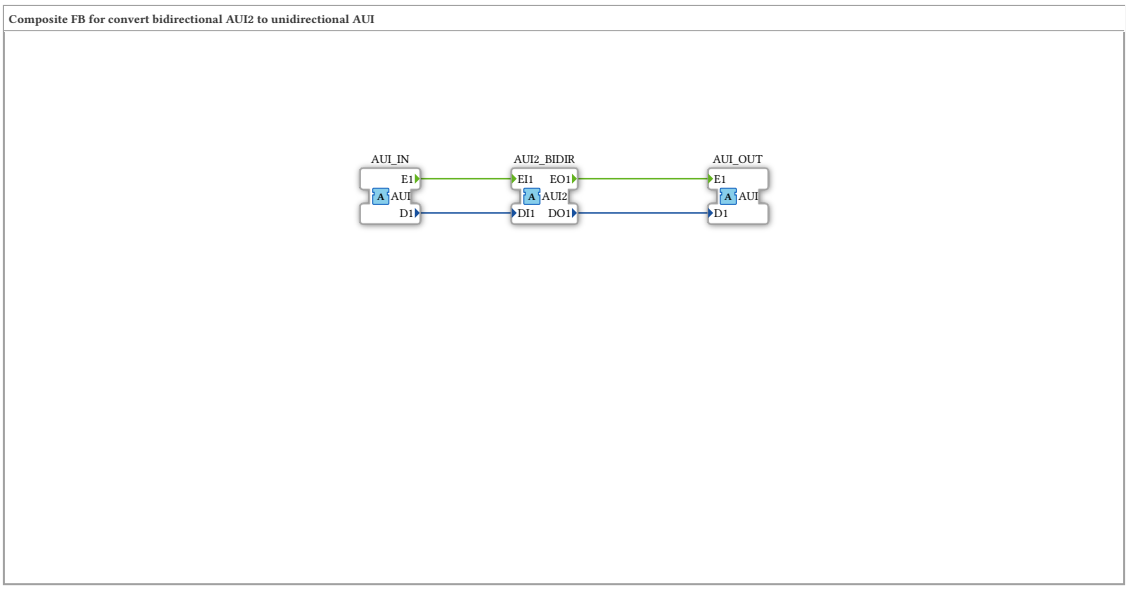
Anwendungsszenarien

- Einen `UINT`-Wert auf einen bidirektionalen `AUI2`-Adapter ausgeben und gleichzeitig dessen Rückkanal als Datenausgang auslesen, wenn der Adapter als Socket vorliegt.

Zusammenfassung

`AUI2_I_TO_UINT` verbindet Ereignisse und `UINT`-Daten 1:1 mit dem bidirektionalen Adapter `AUI2` (Socket-Variante).

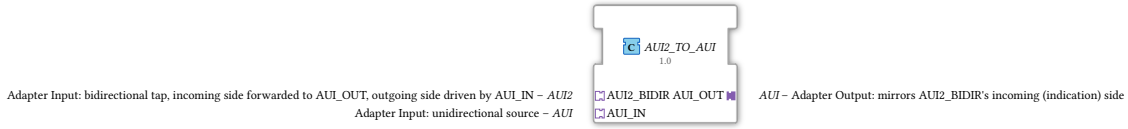
AUI2_TO_AUI



• – ○ • – ○ • – ○ •

Einleitung

`AUI2_TO_AUI` ist ein Composite-Funktionsblock zur Konvertierung eines **bidirektionalen** `AUI2`-Adapters in einen **unidirektionalen** `AUI`-Adapter. Die eingehende Seite (`EO1`/`DO1`) des bidirektionalen Adapters wird als einfaches unidirektionales Signal über `AUI_OUT` ausgegeben; über `AUI_IN` wird ein Wert auf die ausgehende Seite (`EI1`/`DI1`) des bidirektionalen Adapters geschrieben.



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden – Ereignisse kommen über die Adapter-Sockets/-Plugs

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

- **AUI2_BIDIR** (Socket): Bidirektionaler Adapter vom Typ `adapter::types::bidirectional::AUI2` – die eingehende Seite wird an `AUI_OUT` weitergeleitet, die ausgehende Seite wird von `AUI_IN` getrieben
- **AUI_IN** (Socket): Unidirektionaler Adapter vom Typ `adapter::types::unidirectional::AUI` – unidirektionale Quelle
- **AUI_OUT** (Plug): Unidirektionaler Adapter vom Typ `adapter::types::unidirectional::AUI` – spiegelt die eingehende (Indikations-)Seite von `AUI2_BIDIR`

Funktionsweise

1. `AUI2_BIDIR.EO1` wird an `AUI_OUT.E1` und `AUI2_BIDIR.DO1` an `AUI_OUT.D1` weitergeleitet.
2. `AUI_IN.E1` wird an `AUI2_BIDIR.EI1` und `AUI_IN.D1` an `AUI2_BIDIR.DI1` weitergeleitet.

Technische Besonderheiten

- Reine Ereignis-/Datenverbindungen (`FBNetwork`), keine eigene Logik und kein interner Zustand
- Teil der Konvertierungsfamilie (analog zu `AR2`/`AB2`/`AE2`)

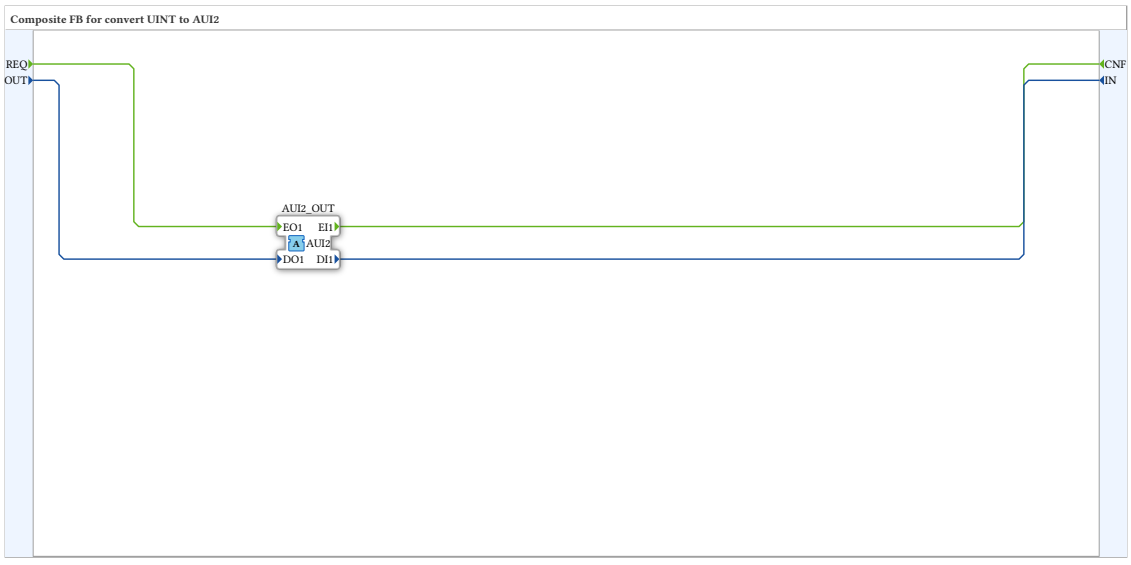
Anwendungsszenarien

- Anbindung eines unidirektionalen `AUI`-Verbrauchers bzw. -Erzeugers an einen bidirektionalen `AUI2`-Adapter

Fazit

`AUI2_TO_AUI` trennt die beiden Richtungen eines bidirektionalen `AUI2`-Adapters in einen unidirektionalen Ausgang (`AUI_OUT`) und einen unidirektionalen Eingang (`AUI_IN`).

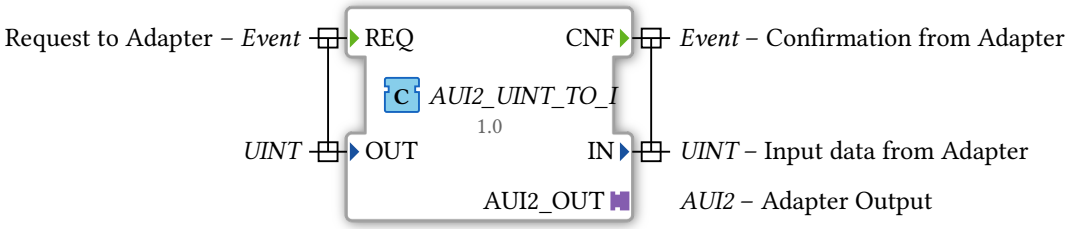
AUI2_UINT_TO_I



• – ◦ • – ◦ • – ◦ •

Einleitung

`AUI2_UINT_TO_I` ist ein Composite-Funktionsblock, der die einfachen Ereignis- und `UINT`-Datenports `REQ`/`OUT` und `CNF`/`IN` auf den bidirektionalen Adapter `AUI2` (als Plug) abbildet. Er gehört zur Konvertierungsfamilie analog zu AR2/AB2/AE2.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Request to Adapter): sendet die Daten `OUT` über den Adapter (assoziiert mit `OUT`)

Ereignis-Ausgänge

- **CNF** (Confirmation from Adapter): wird ausgelöst, wenn der Adapter ein Ereignis auf seiner eingehenden Seite liefert (assoziiert mit `IN`)

Daten-Eingänge

- **OUT** (UINT): Daten, die an den Adapter gesendet werden

Daten-Ausgänge

- **IN** (UINT): vom Adapter empfangene Daten

Adapter

- **AUI2_OUT** (Plug): Adapter vom Typ `adapter::types::bidirectional::AUI2`

Funktionsweise

Der Baustein besteht nur aus Verdrahtung ohne eigene Logik:

- `REQ` ⟶ `AUI2_OUT.EO1` und `OUT` ⟶ `AUI2_OUT.DO1`: Der Anfrage-Ereignisstrom samt Daten wird auf die ausgehende Seite des Adapters veröffentlicht.
- `AUI2_OUT.EI1` ⟶ `CNF` und `AUI2_OUT.DI1` ⟶ `IN`: Was auf der eingehenden Seite des Adapters gemeldet wird, erscheint als Ereignis `CNF` mit den Daten `IN`.

Technische Besonderheiten

- Zustandsloser Composite-Baustein ohne Algorithmen
- Jede Zielvariable hat genau einen Schreiber
- Die Adapterdaten werden unverändert durchgereicht, es findet keine Wertumwandlung statt
- Der Adapter wird hier als Plug geführt, der Baustein steht also am Anfang einer AUI2-Verbindung

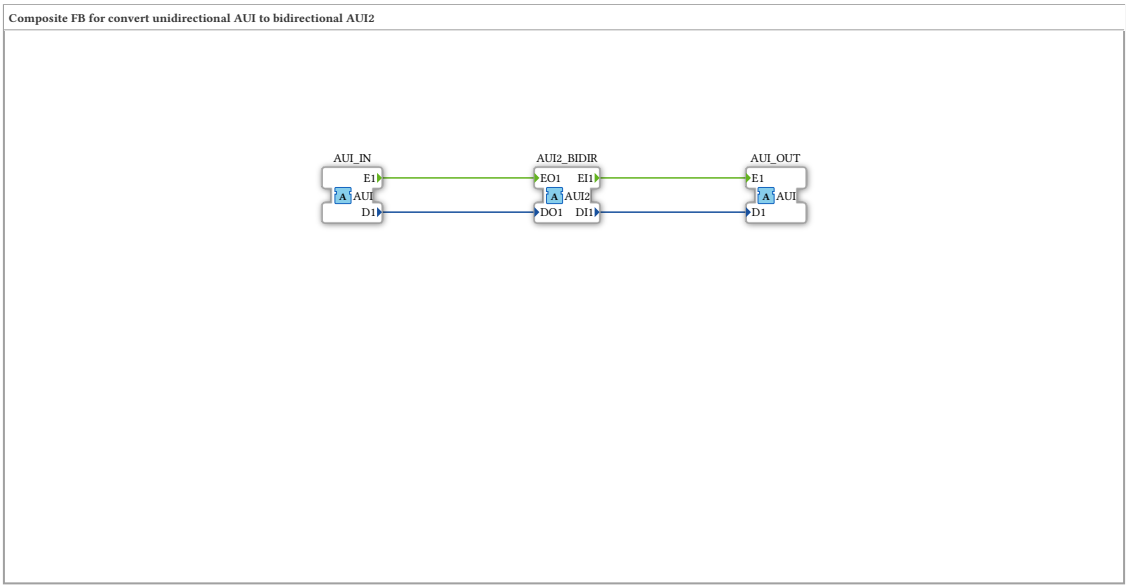
Anwendungsszenarien

- Anbindung klassischer Ereignis-/Datenports an einen bidirektionalen AUI2-Adapter
- Einbindung von Bausteinen mit `UINT`-Ports in adapterbasierte Netzwerke

Fazit

`AUI2_UINT_TO_I` setzt die Ports `REQ`/`OUT` und `CNF`/`IN` ohne Logik auf den bidirektionalen AUI2-Adapter um.

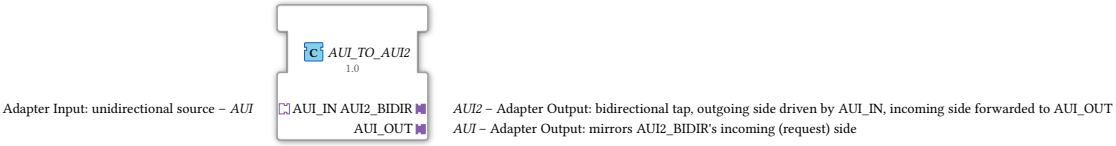
AUI_TO_AUI2



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `AUI_TO_AUI2` hängt einen bidirektionalen `AUI2`-Abzweig („Tap“) in eine unidirektionale `AUI`-Kette (UINT) ein. Er nutzt dafür **drei** Adapter: Ein eingehendes unidirektionales AUI-Signal wird auf die bidirektionale AUI2-Schnittstelle veröffentlicht, und was AUI2 auf seiner Anfrage-Seite zurückliefert, wird als eigenständiges unidirektionales AUI-Signal wieder ausgegeben.



Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **AUI_IN** (Socket): unidirektionale UINT-Quelle, Typ `adapter::types::unidirectional::AUI`
- **AUI2_BIDIR** (Plug): bidirektionaler Abzweig, dessen ausgehende Seite von `AUI_IN` gespeist wird und dessen eingehende Seite an `AUI_OUT` weitergereicht wird, Typ `adapter::types::bidirectional::AUI2`
- **AUI_OUT** (Plug): spiegelt die eingehende (Anfrage-)Seite von `AUI2_BIDIR` als unidirektionales Signal, Typ `adapter::types::unidirectional::AUI`

Funktionsweise

Der Baustein arbeitet in zwei Richtungen gleichzeitig: Was am Socket `AUI_IN` eintrifft (`E1`/`D1`), wird direkt auf die ausgehende Seite von `AUI2_BIDIR` (`EO1`/`DO1`) veröffentlicht. Umgekehrt wird alles, was `AUI2_BIDIR` auf seiner Anfrage-Seite empfängt (`EI1`/`DI1`), unverändert als eigenständiges unidirektionales Signal über

den Plug AUI_OUT ausgegeben. AUI2_BIDIR fungiert damit als Abzweigpunkt, der die unidirektionale Kette nicht unterbricht, sondern um eine bidirektionale Beobachtungs-/Einspeisemöglichkeit erweitert.

Technische Besonderheiten

- Drei statt zwei Adapter: Die eingehende und die vom Tap zurückkommende Signalrichtung werden bewusst getrennt gehalten
- Reine Verdrahtung ohne Logik oder Zustand, jede Zielvariable hat genau einen Schreiber
- AUI_OUT gibt nur wieder, was über AUI2_BIDIR von einem angeschlossenen bidirektionalen Teilnehmer zurückkommt; ohne einen solchen Teilnehmer bleibt AUI_OUT inaktiv

Zustandsübersicht

Der Baustein ist zustandslos:

- AUI_IN.E1 ⟶ AUI2_BIDIR.EO1, AUI_IN.D1 ⟶ AUI2_BIDIR.DO1
- AUI2_BIDIR.EI1 ⟶ AUI_OUT.E1, AUI2_BIDIR.DI1 ⟶ AUI_OUT.D1

Anwendungsszenarien

- Nachträgliches Einhängen eines bidirektionalen Diagnose- oder Steuerpunkts in eine bestehende unidirektionale AUI-Signalkette, ohne diese zu unterbrechen
- Brücke zwischen einer unidirektionalen AUI-Zone und einer bidirektionalen AUI2-Zone

⚖️ Vergleich mit ähnlichen Bausteinen

Das Gegenstück in umgekehrter Richtung ist der Baustein AUI2_TO_AUI (dort steht AUI2_BIDIR als Socket auf der linken Seite). Die Bausteine dieser Konvertierungsfamilie existieren analog für die Adapter AR/AR2, AB/AB2 und AE/AE2.

Fazit

AUI_TO_AUI2 ist kein einfacher Konverter, sondern ein Abzweigpunkt: Er macht ein unidirektionales AUI-Signal an einem bidirektionalen AUI2-Punkt sichtbar, ohne die ursprüngliche Signalkette zu unterbrechen, und gibt das dort empfangene Anfrage-Signal als eigenständiges AUI-Signal weiter.

unidirectional conversion

- 📖 Numerisch vs. bitweise: Die Konvertierungs-Falle in FORTE

Numerisch vs. bitweise: Die Konvertierungs-Falle in FORTE

• – ◦ • – ◦ • – ◦ •

Einleitung

Alle Bausteine in diesem Ordner (AB_TO_AR, AD_TO_ADI, AI_TO_AR, usw.) sowie die zugrunde liegenden Standard-Funktionsblöcke iec61131::conversion::F_X_TO_Y (Teil der 4diac-IDE-Standardbibliothek, nicht in diesem Repository vendored) wandeln einen Wert von einem IEC-61131-Datentyp in einen anderen um. Für einen Teil dieser Kombinationen ist das eine **echte Zahlenwert-Umwandlung** für einen anderen Teil ist es eine **reine Bit-Reinterpretation**, bei der der Zahlenwert bewusst ignoriert und stattdessen das rohe Bitmuster übernommen wird. Wer den Unterschied nicht kennt, produziert leicht einen stillen, schwer zu findenden Bug — siehe AD_TO_AR_TODO.md im Quell-Repository für den konkreten Fall, der diese Seite ausgelöst hat.

Die vier Typkategorien

IEC 61131-3 unterscheidet vier relevante Kategorien:

Kategorie	Typen	Bedeutung
ANY_BIT	BOOL, BYTE, WORD, DWORD, LWORD	reine Bitmuster ohne eigene Zahlenwert-Semantik
ANY_INT (vorzeichenbehaftet)	SINT, INT, DINT, LINT	Ganzzahlen mit Vorzeichen
ANY_INT (vorzeichenlos)	USINT, UINT, UDINT, ULINT	Ganzzahlen ohne Vorzeichen
ANY_REAL	REAL, LREAL	IEEE754-Gleitkommazahlen

Die Adapter-Präfixe in diesem Ordner entsprechen: AB=BYTE, AW=WORD, AD=DWORD, AL=LWORD, AX=BOOL, AS=SINT, AI=INT, ADI=DINT, ALI=LINT, AUS=USINT, AUI=UINT, AUDI=UDINT, AULI=ULINT, AR=REAL, ALR=LREAL.

Die Umwandlungsmatrix

Verifiziert im FORTE-Kern (`core/include/forte/datatypes/forte_any.h`, `CIEC_ANY::cast<U,T>`, sowie `forte_real.cpp`/`forte_lreal.cpp`, `CIEC_REAL::castRealData`) — dieselbe Logik hinter jedem `F_X_TO_Y`-Baustein und jedem Adapter-Wrapper in diesem Ordner:

Quelle ↓ Ziel →	→ BOOL	→ ANY_BIT (außer BOOL)	→ ANY_INT	→ ANY_REAL
ANY_BIT (außer BOOL)	LSB-Test	Bitmuster-Übertragung (strukturell, kein Zahlenwert)	Bit-Reinterpretation — bei einem **breiteren** Ziel bleibt der Zahlenwert erhalten (Nullerweiterung); bei **gleicher Breite** bleibt zwar das Bitmuster erhalten, aber ein vorzeichenbehaftetes Ziel kann den Zahlenwert ändern (z. B. wird `0xFF` als `BYTE` zu `-1` als `SINT`, nicht zu `255`); ein **schmaleres** Ziel schneidet die oberen Bits ab	⚠️ **Bit-Reinterpretation — KEIN Zahlenwert!** IEEE754-Fehlinterpretation
BOOL	Identität	Numerisch `0`/`1`	numerisch (0/1)	numerisch (0.0/1.0) — Sonderfall, siehe unten
ANY_INT	Boolesche Konvertierung (`!= 0`)	Bitmuster-Übertragung (erwartetes Verhalten für ein Bit-String-Ziel)	numerisch (Vorzeichenerweiterung/Nullerweiterung sicher, Verengung kann abschneiden)	**numerisch** (korrekter Cast)
ANY_REAL	Boolesche Konvertierung (`!= 0.0`)	Serialisierung via `F_REAL_TO_DWORD`)	Bit-Extraktion (beabsichtigt, z. B.	numerisch (Rundung, `llrint`) numerisch (Auf-/Abrunden der Genauigkeit)

Die auffälligste Falle ist die Zelle **ANY_BIT (außer BOOL) → ANY_REAL** (rot markiert): `BYTE`/`WORD`/`DWORD`/`LWORD` als Quelle einer Umwandlung nach `REAL`/`LREAL`. In dieser Bibliothek betrifft das konkret zwei Bausteine:

Eine weitere, oben in der Matrix bereits vermerkte Falle betrifft `ANY_BIT` (außer BOOL) → `ANY_INT` bei gleicher Bitbreite: Das Bitmuster bleibt erhalten, aber ein vorzeichenbehaftetes Ziel kann den Zahlenwert ändern (z. B. wird `0xFF` als `BYTE` zu `-1` als `SINT`). Anders als bei `ANY_BIT`→`ANY_REAL` bleibt der resultierende Wert dabei aber im plausiblen Zahlenbereich und wird nicht durch eine IEEE754-Fehlinterpretation komplett sinnlos — der Fehler fällt beim Debuggen daher meist schneller auf.

- `AD_TO_AR` (DWORD→REAL) — sicherer Ersatz: `AD_TO_AR_NUM`
- `AL_TO_ALR` (LWORD→LREAL) — sicherer Ersatz: `AL_TO_AULI` + `AULI_TO_ALR`

(Es gibt in dieser Bibliothek keine AB_TO_AR/AB_TO_ALR/AW_TO_AR/AW_TO_ALR-Paare — nur breitenpassende Bit↔Real-Kombinationen wurden als Adapter angeboten.)

Warum ist BOOL→REAL sicher, obwohl BOOL zu ANY_BIT gehört? FORTE behandelt BOOL beim Cast nach REAL als expliziten Sonderfall (`case e_BOOL: setTFLOAT(...)` statt der generischen Bit-String-Kopie) — der einzige Ausnahmefall in der Matrix.

Warum ist ANY_BIT→ANY_REAL überhaupt so implementiert?

Nicht aus Willkür: Für Bit-String-Ziele (ANY_BIT→ANY_BIT) und für den Rückweg ANY_REAL→ANY_BIT ist die Bit-Reinterpretation genau das gewünschte, dokumentierte Verhalten — z. B. um das IEEE754-Bitmuster eines REAL für die Übertragung in ein DWORD zu packen (F_REAL_TO_DWORD) und später mit AD_TO_AR wieder auszupacken. Die Krux ist die **Verwechslungsgefahr**: derselbe Mechanismus wird fälschlich auch dann verwendet, wenn eigentlich ein roher Zähler- oder Analogwert (kein serialisiertes Bitmuster) numerisch nach REAL soll.

Praktische Faustregel

- **Kommt der Wert aus einem F_REAL_TO_X/F_X_TO_REAL-Roundtrip oder einer Feldbus-/Protokoll-Deserialisierung, bei der explizit Bitmuster übertragen werden?** → Bit-Reinterpretation ist korrekt (AD_TO_AR, AL_TO_ALR).
- **Ist der Wert ein roher Zähler-, Analog- oder sonstiger Ganzzahlwert, der als derselbe Zahlenwert in REAL vorliegen soll?** → Numerische Variante verwenden (AD_TO_AR_NUM, oder allgemein: zweistufig über den passenden ANY_INT-Zwischentyp, z. B. AD_TO_AUDI → AUDI_TO_AR).
- **Alle anderen Zellen der Matrix** sind entweder eindeutig numerisch oder eindeutig strukturell (Bit-Operation) — ohne die versteckte Fehlinterpretations-Gefahr der ANY_BIT→ANY_REAL-Zelle. Jeder Baustein in diesem Ordner trägt einen kurzen Hinweis dazu in seinem Abschnitt „Technische Besonderheiten“.

Betrifft auch die Standard-Konvertierungsbausteine

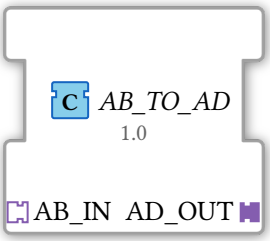
Die hier dokumentierten Adapter-Bausteine (AD_TO_AR etc.) sind dünne Wrapper um die Standard-4diac-Funktionsblöcke `iec61131::conversion::F_X_TO_Y` (z. B. F_DWORD_TO_REAL). Diese Standardbausteine sind Teil der 4diac-IDE-Distribution selbst (nicht in diesem Repository vendored) und unterliegen exakt derselben Matrix — F_DWORD_TO_REAL, F_WORD_TO_REAL, F_BYTE_TO_LREAL usw. sind ebenso Bit-Reinterpretationen, während F_DINT_TO_REAL, F_UDINT_TO_REAL usw. numerisch korrekt sind.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

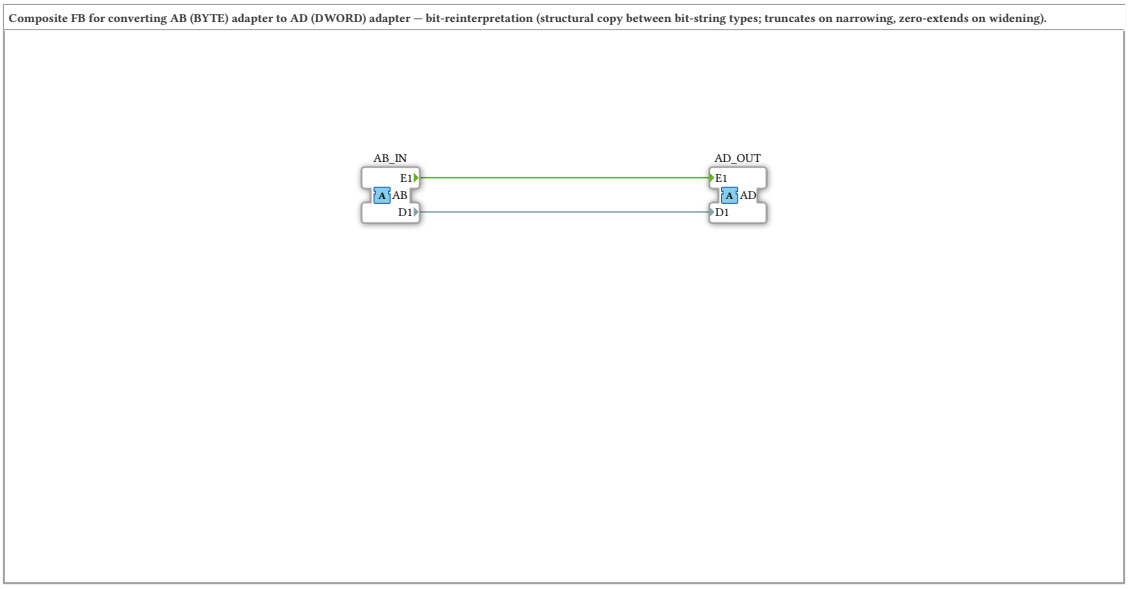
AB_AD

AB_TO_AD



BYTE Adapter Input – *AB*

AD – DWORD Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AB_TO_AD** ist ein zusammengesetzter Baustein (Composite FB), der einen BYTE-Adapter (Typ `AB`) in einen DWORD-Adapter (Typ `AD`) umwandelt. Er dient als Schnittstellen-Konverter in einer unidirektionalen Kommunikation und ermöglicht die einfache Anbindung von Komponenten, die auf unterschiedlichen Datenwortbreiten basieren.

Schnittstellenstruktur

Ereignis-Eingänge

- **AB_IN.E1** – Ereigniseingang vom BYTE-Adapter. Wird direkt an den DWORD-Adapter weitergeleitet.

Ereignis-Ausgänge

- **AD_OUT.E1** – Ereignisausgang zum DWORD-Adapter. Entspricht dem weitergeleiteten Ereignis von **AB_IN.E1**.

Daten-Eingänge

- **AB_IN.D1** – Dateneingang vom BYTE-Adapter (1 Byte Datenbreite). Wird direkt an den DWORD-Adapter weitergegeben.

Daten-Ausgänge

- **AD_OUT.D1** – Datenausgang zum DWORD-Adapter (4 Byte Datenbreite). Enthält denselben Wert wie der Eingang **AB_IN.D1**, jedoch im Format eines DWORD (Double Word).

Adapter

- **AB_IN** (Socket) – Unidirektionaler BYTE-Adapter (Typ `AB`), der als Eingang dient.
- **AD_OUT** (Plug) – Unidirektionaler DWORD-Adapter (Typ `AD`), der als Ausgang dient.

Funktionsweise

Der Baustein verbindet die Schnittstellen eines BYTE-Adapters (Eingang) mit denen eines DWORD-Adapters (Ausgang). Jedes eingehende Ereignis (**E1**) wird ohne Verzögerung oder Bearbeitung an den Ausgang weitergeleitet. Ebenso wird der eingehende Datenwert (**D1**) unverändert, aber als 4-Byte-DWORD, an den Ausgang übergeben. Der Baustein besitzt keine eigene Logik; er fungiert ausschließlich als Durchreiche („Passthrough“) zwischen den beiden Adaptertypen.

Technische Besonderheiten

- **Bit-Reinterpretation** (strukturelle Kopie zwischen Bit-String-Typen; schneidet bei Verengung ab, erweitert mit Nullen bei Erweiterung).

- **Unidirektionalität:** Der Baustein unterstützt nur eine Datenflussrichtung vom BYTE-Adapter zum DWORD-Adapter. Eine Rückwärtsübertragung ist nicht vorgesehen.
- **Datenbreitenanpassung:** Der eingehende BYTE-Wert (1 Byte) wird in einen DWORD-Wert (4 Byte) eingebettet. Das niederwertigste Byte des DWORD entspricht dem eingehenden BYTE; die höherwertigen Bytes werden auf Null gesetzt (bzw. bleiben je nach Implementierung des Zieladapters erhalten).
- **Einfache Kopplung:** Der FB ist ein reiner Verbindungsbaustein im FBNetwork und benötigt keine eigene Zustandsmaschine oder Algorithmen.

Zustandsübersicht

Der Baustein verfügt über keine eigene Zustandslogik, da er ausschließlich als Verkabelung zwischen den Adaptern fungiert. Sein Verhalten ist ereignisgesteuert: Jedes Ereignis am Eingang wird sofort zum Ausgang durchgereicht.

Anwendungsszenarien

- **Systemintegration:** Wenn ein Subsystem mit BYTE-Adaptern arbeitet, ein nachfolgendes Subsystem jedoch DWORD-Adapter erwartet, kann `AB_TO_AD` als Konverter zwischengeschaltet werden.
- **Datenaggregation:** Mehrere solcher Bausteine können verwendet werden, um einzelne Bytes zu einem DWORD zusammenzuführen, falls die Quelladapter jeweils nur ein Byte liefern.
- **Protokollanpassung:** In Feldbus- oder Sensoranbindungen, bei denen die Datenbreite zwischen Komponenten wechselt, dient der Baustein als einfache Anpassungsschicht.

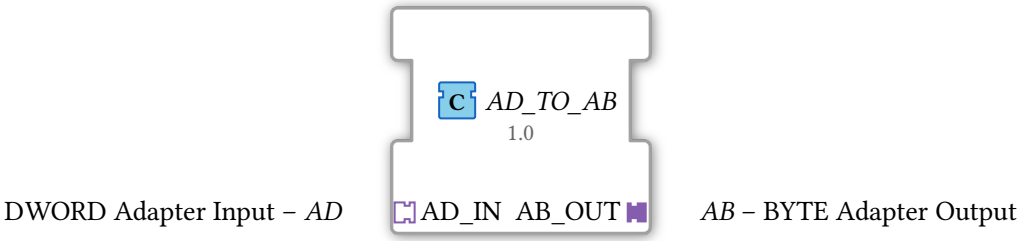
Vergleich mit ähnlichen Bausteinen

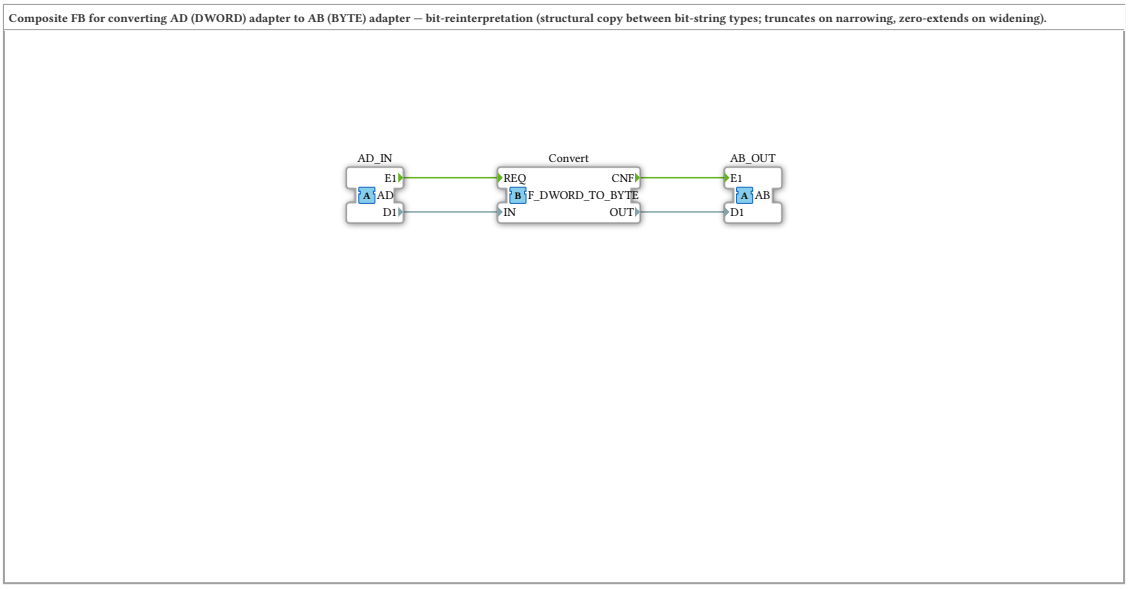
Es existieren analoge Konverterbausteine für andere Datenbreiten, z. B. `AB_TO_WORD` (BYTE nach WORD) oder `WORD_TO_DWORD`. Der vorliegende Baustein spezialisiert sich auf die Kombination `AB` ⟶ `AD` und ist daher ideal für den direkten Ersatz oder die Kopplung entsprechender Adapter. Im Gegensatz zu komplexeren Konvertern mit Datenmanipulation oder Puffern arbeitet er verzögerungsfrei und ohne zusätzliche Logik.

Fazit

`AB_TO_AD` ist ein schlanker, klar strukturierter Konverterbaustein, der die unidirektionale Umwandlung eines BYTE-Adapters in einen DWORD-Adapter ermöglicht. Durch seine einfache Durchreichung eignet er sich hervorragend für die schnelle Integration unterschiedlicher Schnittstellen in IEC 61499-basierten Steuerungssystemen, ohne zusätzliche funktionale Komplexität einzuführen.

AD_TO_AB





• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock **AD_TO_AB** ist ein Composite-Baustein zur unidirektionalen Konvertierung eines 32-Bit-DWORD-Adapters (AD) in einen 8-Bit-BYTE-Adapter (AB). Er dient als Brücke zwischen Komponenten, die auf unterschiedliche Datenbreiten ausgelegt sind, und kapselt die notwendige Typumwandlung gemäß IEC 61131-3.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine direkt sichtbaren Ereignis-Eingänge. Die Ereignissteuerung erfolgt über den Adapter **AD_IN**.

Ereignis-Ausgänge

Der Baustein besitzt keine direkt sichtbaren Ereignis-Ausgänge. Die Ereignisrückmeldung erfolgt über den Adapter **AB_OUT**.

Daten-Eingänge

Der Baustein besitzt keine direkt sichtbaren Daten-Eingänge. Die Bereitstellung des DWORD-Werts erfolgt über den Adapter **AD_IN**.

Daten-Ausgänge

Der Baustein besitzt keine direkt sichtbaren Daten-Ausgänge. Die Ausgabe des BYTE-Werts erfolgt über den Adapter **AB_OUT**.

Adapter

Adapter	Typ	Richtung	Beschreibung
AD_IN	`adapter::types::unidirectional::AD`	Socket (Eingang)	Nimmt ein DWORD (32-Bit) sowie ein zugehöriges Ereignis (hier `E1`) entgegen.
AB_OUT	`adapter::types::unidirectional::AB`	Plug (Ausgang)	Gibt ein konvertiertes BYTE (8-Bit) sowie ein quittierendes Ereignis (hier `E1`) aus.

Beide Adapter sind unidirektional ausgelegt. Die genaue Signatur der Adapter (Anzahl der Ereignis- und Datenkanäle) entspricht den in der 4diac-Bibliothek definierten Typen.

Funktionsweise

Der Baustein arbeitet rein ereignisgesteuert und kapselt die Konvertierung mithilfe des internen Basisbausteins **F_DWORD_TO_BYTE** aus der IEC 61131-1-Bibliothek.

1. Ein eingehendes Ereignis an **AD_IN.E1** triggert den Aufruf des internen Konvertierungsbausteins über dessen Eingang `REQ`.
2. Der Wert aus **AD_IN.D1** (DWORD) wird an den Dateneingang `IN` des Konverters weitergeleitet.
3. Nach erfolgter Wandlung signalisiert der Konverter mit `CNF` die Bereitschaft des Ergebnisses.
4. Dieses Ereignis wird an **AB_OUT.E1** weitergegeben, und der konvertierte BYTE-Wert (aus `OUT`) wird an **AB_OUT.D1** ausgegeben.

Die Umwandlung erfolgt gemäß IEC 61131-3: Die niederwertigen 8 Bit des eingehenden DWORD werden als BYTE interpretiert.

Technische Besonderheiten

- **Bit-Reinterpretation** (strukturelle Kopie zwischen Bit-String-Typen; schneidet bei Verengung ab, erweitert mit Nullen bei Erweiterung).
- Der Baustein ist als Composite realisiert, d.h. er enthält keine eigene Zustandsmaschine, sondern delegiert die gesamte Logik an den internen Basisbaustein.
- Weder Pufferung noch Fehlerbehandlung sind implementiert; die Datenübergabe erfolgt direkt synchron zum Ereignis.
- Der Baustein ist für den Einsatz in Systemen vorgesehen, die eine klare Trennung zwischen Adaptern für unterschiedliche Datenbreiten erfordern.

Zustandsübersicht

Da es sich um einen reinen Composite-Baustein ohne eigene Zustandsmaschine handelt, existiert kein expliziter Zustandsautomat. Sein Verhalten wird vollständig durch den internen **F_DWORD_TO_BYTE**-Baustein bestimmt, der einen einfachen Wartezustand (IDLE) und einen Verarbeitungszustand besitzt.

Anwendungsszenarien

- **Anbindung von Sensoren/Aktoren**: Ein Sensor liefert 32-Bit-Rohdaten über einen AD-Adapter, die nach Konvertierung in einen 8-Bit-Wert an einen nachgelagerten BYTE-Adapter weitergegeben werden.
- **Protokollwandlung**: In einer Kommunikationskette wird eine DWORD-Schnittstelle auf eine BYTE-Schnittstelle abgebildet, z. B. bei der Integration älterer Feldbusgeräte.
- **Reduzierung der Datenbreite**: Gezielte Extraktion des niederwertigen Bytes aus einem DWORD für nachfolgende Verarbeitungsschritte.

Vergleich mit ähnlichen Bausteinen

Baustein	Konvertierung	Richtung	Adapter-Typ
AD_TO_AB	DWORD ⟶ BYTE	unidirektional	AD ↔ AB
AD_TO_AW	DWORD ⟶ WORD	unidirektional	AD ↔ AW
AW_TO_AB	WORD ⟶ BYTE	unidirektional	AW ↔ AB

Der **AD_TO_AB** unterscheidet sich durch die spezifische Ziel-Datenbreite und die Verwendung der entsprechenden Adapter. Er ist speziell für Systeme ausgelegt, die mit den unidirektionalen Standardadaptern arbeiten.

Fazit

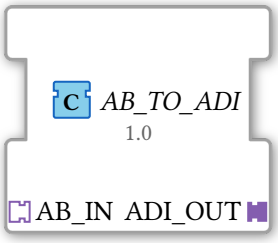
Der Composite-Baustein **AD_TO_AB** bietet eine einfache, ereignisgesteuerte Möglichkeit zur Konvertierung eines DWORD-Adapters in einen BYTE-Adapter. Er kapselt die Typumwandlung sauber und kann ohne eigene Zustandslogik in bestehende 4diac-Netzwerke integriert werden. Die Wiederverwendung des standardisierten **F_DWORD_TO_BYTE**-Bausteins garantiert korrektes Verhalten gemäß IEC 61131-3.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

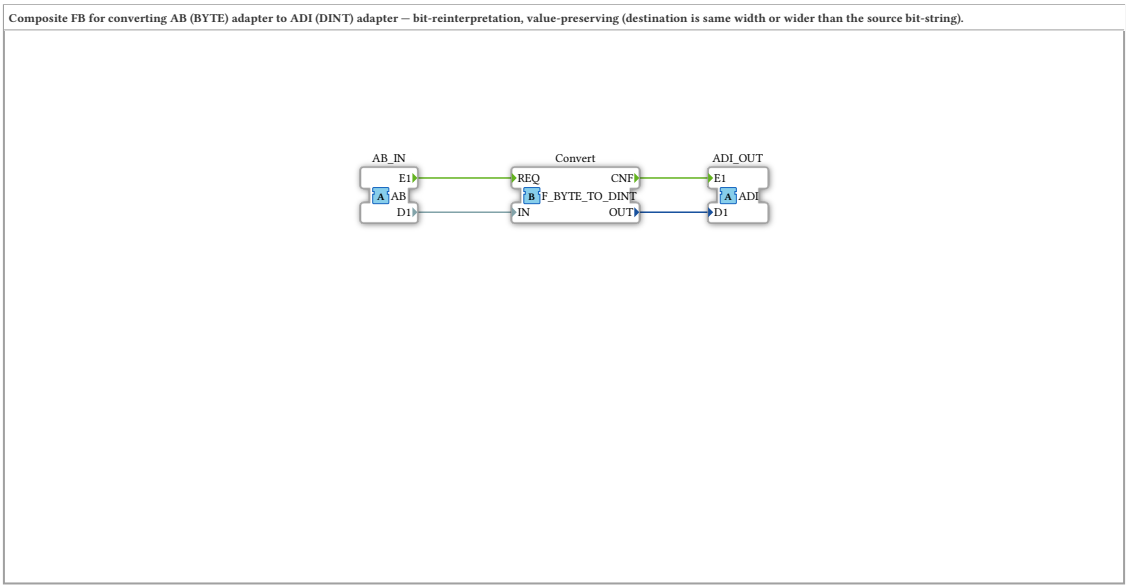
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AB_ADI

AB_TO_ADI



BYTE Adapter Input – *AB* — *ADI* – DINT Adapter Output



• – ◦ • – ◦ • – ◦ • –

Einleitung

Der Funktionsblock **AB_TO_ADI** ist ein zusammengesetzter Baustein (Composite FB), der einen eingehenden BYTE-Wert eines **AB**-Adapters in einen DINT-Wert umwandelt und diesen über einen **ADI**-Adapter ausgibt. Er dient als einfache, unidirektionale Konvertierungsschnittstelle zwischen diesen beiden Adaptertypen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine – die Ereignissteuerung erfolgt ausschließlich über den eingehenden Adapter (`AB_IN`).

Ereignis-Ausgänge

Keine – Ereignisse werden nur über den ausgehenden Adapter (`ADI_OUT`) weitergegeben.

Daten-Eingänge

Keine – die Daten werden über den `AB_IN`-Adapter bereitgestellt.

Daten-Ausgänge

Keine – die konvertierten Daten werden über den `ADI_OUT`-Adapter ausgegeben.

Adapter

Typ	Name	Richtung	Kommentar
Socket	`AB_IN`	Eingang	BYTE-Adapter (Eingang)

Plug	ADI_OUT	Ausgang	DINT-Adapter (Ausgang)

Funktionsweise

Der FB enthält intern einen einzelnen Konvertierungsbaustein F_BYTE_TO_DINT (aus der IEC-61131-Bibliothek). Das eingehende Ereignis E1 des Adapters AB_IN triggert die Konvertierung des Datenwerts D1 (BYTE). Der daraus resultierende DINT-Wert wird über OUT an den Datenausgang D1 des Adapters ADI_OUT weitergeleitet. Nach Abschluss der Konvertierung wird das Ereignis E1 des Ausgangsadapters ausgelöst.

Die Ansteuerung erfolgt rein datenflussgesteuert: Jedes Ereignis am Eingangsadapter bewirkt eine einmalige Umwandlung und Weiterleitung.

Technische Besonderheiten

- **Bit-Reinterpretation, werterhaltend** (Zielbreite ist gleich groß oder größer als der Quell-Bit-String).
- **Zusammengesetzter Baustein** – Keine eigene Zustandslogik, sondern alleinige Nutzung des internen FBs F_BYTE_TO_DINT.
- **Adapter-basiert** – Die Schnittstellen sind als Adapter definiert, was eine hohe Wiederverwendbarkeit und Austauschbarkeit mit anderen Bausteinen ermöglicht.
- **Lizenz** – Der Baustein ist unter der **Eclipse Public License 2.0** veröffentlicht.
- **Paket** – Der FB ist im Paket adapter::conversion::unidirectional organisiert.

Zustandsübersicht

Der Baustein besitzt keinen eigenen Zustandsautomaten. Die Funktionalität ergibt sich allein aus der inneren Verschaltung des Konvertierungsbausteins. Die Verarbeitung ist strikt ereignisgesteuert und kombinatorisch.

Anwendungsszenarien

- Umwandlung von BYTE-basierten Sensordaten (z. B. digitale Statuswerte) in ganzzahlige DINT-Werte für weiterverarbeitende Steuerungslogiken.
- Anbindung von Komponenten, die den **AB**-Adapter verwenden, an Systeme, die den **ADI**-Adapter erwarten.
- Einsatz in Adapter-basierten Kommunikationsarchitekturen, wo eine einheitliche Datenrepräsentation hergestellt werden muss.

Vergleich mit ähnlichen Bausteinen

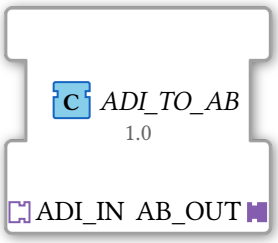
- **AB_TO_AI** – Konvertiert BYTE nach INT (16-Bit-Ganzzahl).
- **AB_TO_AREAL** – Konvertiert BYTE nach REAL.
- **AB_TO_ADI** – Spezifisch für BYTE nach DINT (32-Bit-Ganzzahl).

Alle diese Bausteine folgen dem gleichen Prinzip: Ein- und Ausgang sind als unidirektionale Adapter definiert, die intern eine Standard-Konvertierungsfunktion nutzen.

Fazit

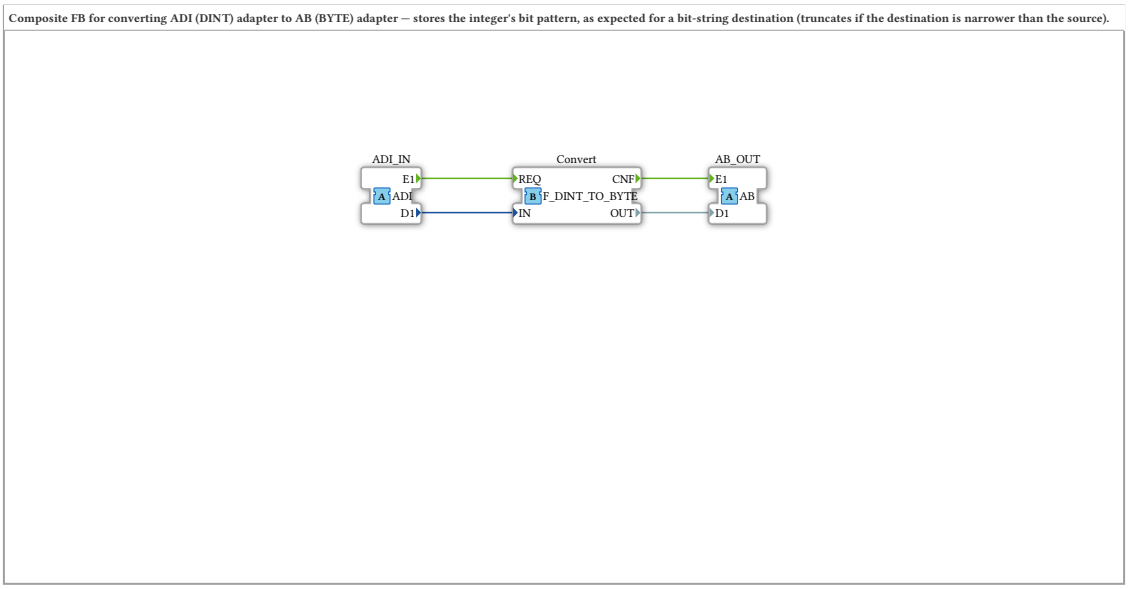
Der FB AB_TO_ADI stellt eine minimalistische und zuverlässige Lösung zur Umwandlung von BYTE- in DINT-Daten dar. Seine Adapter-Schnittstellen ermöglichen eine nahtlose Integration in modulare Steuerungssysteme, ohne dass die zugrundeliegende Konvertierungslogik neu implementiert werden muss. Er ist besonders geeignet für Anwendungen, die auf klare Trennung von Kommunikationsprotokollen und Datenformaten Wert legen.

ADI_TO_AB



DINT Adapter Input – *ADI*

AB – BYTE Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `ADI_TO_AB` dient als Komposit-Baustein zur Umwandlung eines 32-Bit-Integer-Wertes (DINT) in einen Byte-Wert (BYTE). Er kapselt die Konvertierung zwischen zwei unidirektionalen Adaptern: dem **ADI-Adapter** (Eingang) und dem **AB-Adapter** (Ausgang). Der Baustein wird eingesetzt, um Daten von einem DINT-Adapter auf einen BYTE-Adapter zu übertragen, ohne dass der Anwender die eigentliche Konvertierungslogik selbst implementieren muss.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine direkten Ereignis-Eingänge. Der Ereignis-Empfang erfolgt über den **Socket-Adapter ADI_IN**:

Adapter-Port	Typ	Beschreibung
ADI_IN.E1	Ereignis	Startet die Konvertierung am Eingang.

Ereignis-Ausgänge

Auch die Ereignis-Ausgabe erfolgt ausschließlich über den **Plug-Adapter AB_OUT**:

Adapter-Port	Typ	Beschreibung
AB_OUT.E1	Ereignis	Signalisiert, dass der Wert konvertiert und am Ausgang bereitsteht.

Daten-Eingänge

Die Daten werden über den ADI_IN-Adapter bereitgestellt:

Adapter-Port	Typ	Beschreibung

ADI_IN.D1	DINT	32-Bit-Integer-Eingangswert.

Daten-Ausgänge

Die konvertierten Daten werden über den AB_OUT-Adapter ausgegeben:

Adapter-Port	Typ	Beschreibung
AB_OUT.D1	BYTE	8-Bit-Byte-Ausgangswert.

Adapter

Name	Richtung	Typ des Adapters	Beschreibung
ADI_IN	Socket	`adapter::types::unidirectional::ADI`	DINT-Adapter als Eingang.
AB_OUT	Plug	`adapter::types::unidirectional::AB`	BYTE-Adapter als Ausgang.

Funktionsweise

Intern verwendet der Baustein den vordefinierten IEC-61131-3-Funktionsbaustein `F_DINT_TO_BYTE`, um die eigentliche Datentypkonvertierung durchzuführen. Die Ereignis- und Datenflüsse sind wie folgt:

1. Ein Ereignis am **ADI_IN.E1** triggert die Konvertierung.
2. Der Datenwert von **ADI_IN.D1** wird an den Eingang `IN` des internen `F_DINT_TO_BYTE`-Bausteins weitergeleitet.
3. Nach Abschluss der Konvertierung erzeugt der interne Baustein das Ereignis **CNF**.
4. Dieses Ereignis wird als **AB_OUT.E1** nach außen gegeben.
5. Gleichzeitig steht der konvertierte Wert am Ausgang `OUT` des internen Bausteins bereit und wird über **AB_OUT.D1** ausgegeben.

Die gesamte Logik ist in einem Composite-FB gekapselt, sodass der Anwender nur die Adapter verbinden muss.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Composite-FB**: Der Baustein besteht aus einem internen Netzwerk und besitzt keine eigene Zustandsmaschine (ECC). Er führt die Konvertierung streng ereignisgesteuert aus.
- **Verwendung von IEC-61131-3-Standard-Bausteinen**: Die Konvertierung erfolgt mit dem normierten Baustein `F_DINT_TO_BYTE`, was Portabilität und Einhaltung von Industriestandards gewährleistet.
- **Adapter-basierte Ein-/Ausgabe**: Alle Daten und Ereignisse werden über Adapter – und nicht über einzelne Ports – geführt. Dies vereinfacht die Anbindung an gleichartige Adapter-Schnittstellen in der 4diac-Umgebung.

Zustandsübersicht

Da es sich um einen Composite-FB ohne eigene Zustandsmaschine handelt, existieren keine expliziten Zustände. Der Baustein reagiert auf das eingehende Ereignis, konvertiert den Wert und gibt das Ergebnis aus. Nach der Ausgabe ist er sofort bereit für eine neue Konvertierung.

Anwendungsszenarien

- **Datenprotokoll-Anpassung**: Wenn ein Sensor oder eine Steuerung DINT-Werte liefert, das angeschlossene Gerät oder der Kommunikationskanal jedoch BYTE-Werte erwartet.
- **Adapter-Vermittlung**: In einer modularen Architektur, die den Austausch von Adaptern mit unterschiedlichen Datentypen ermöglicht, ohne die Verbindungslogik jedes Mal neu schreiben zu müssen.
- **Schnittstellen-Kopplung**: Zwischen Komponenten, die unidirektionale Adapter nutzen, z. B. in einer Pipeline-Struktur.

Vergleich mit ähnlichen Bausteinen

Es gibt zahlreiche Konvertierungsbausteine für direkte Datentypumwandlungen (z. B. `F_DINT_TO_INT`, `F_DINT_TO_WORD`). Der `ADI_TO_AB` unterscheidet sich dadurch, dass er keine einzelnen Daten- und

Ereignisports, sondern vollständige Adapter verwendet. Dadurch wird die Kapselung verbessert und die Wiederverwendbarkeit erhöht. Ein ähnliches Konzept existiert für andere Datentyp-Paare (z. B. `ADI_TO_AC` für DINT nach CHAR), die der gleichen Entwurfslogik folgen.

Fazit

Der Funktionsblock `ADI_TO_AB` bietet eine sauber gekapselte, adaptergestützte Konvertierung von DINT- nach BYTE-Werten. Er nutzt Standard-IEC-Bausteine und Ereignissteuerung, sodass er sich nahtlos in 4diac-Projekte integrieren lässt. Durch die Adapter-Schnittstellen ist er besonders für modulare und wiederverwendbare Steuerungsanwendungen geeignet.

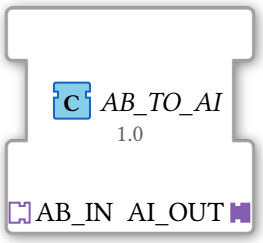
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

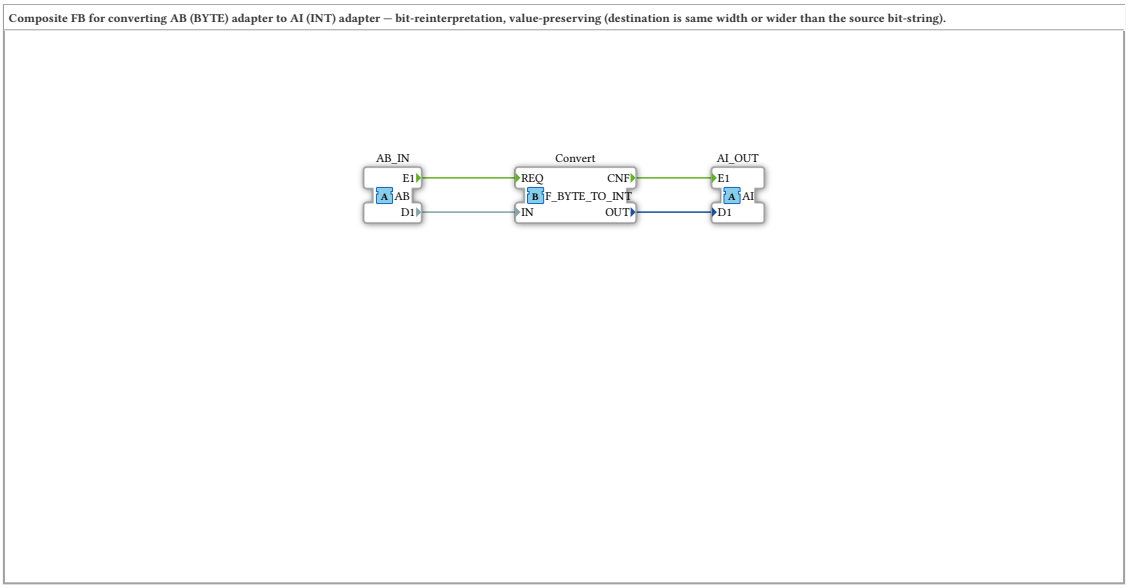
AB_AI

AB_TO_AI

BYTE Adapter Input – *AB*



AI – INT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der **AB_TO_AI** ist ein Composite-Funktionsblock zur Konvertierung eines **AB-Adapters (BYTE)** in einen **AI-Adapter (INT)**. Er ermöglicht die nahtlose Integration von BYTE-basierten Adapter-Schnittstellen in Systeme, die INT-Werte erwarten. Der Baustein nutzt intern den IEC-61131-Konvertierungsbaustein `F_BYTE_TO_INT`.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine direkten Ereignis-Eingänge. Das Ereignis wird über den angeschlossenen Adapter **AB_IN** (Socket) bereitgestellt:

- **AB_IN.E1** – Ereignis, das die Konvertierung eines eingehenden BYTE-Wertes auslöst.

Ereignis-Ausgänge

Der FB besitzt keine direkten Ereignis-Ausgänge. Das Ausgangsereignis wird über den angeschlossenen Adapter **AI_OUT** (Plug) gesendet:

- **AI_OUT.E1** – Ereignis, das signalisiert, dass der konvertierte INT-Wert am Ausgang anliegt.

Daten-Eingänge

Der FB besitzt keine direkten Daten-Eingänge. Der Eingangswert wird über den Adapter **AB_IN** (Socket) bereitgestellt:

- **AB_IN.D1** – BYTE-Wert (0…255), der in einen INT-Wert umgewandelt werden soll.

Daten-Ausgänge

Der FB besitzt keine direkten Daten-Ausgänge. Der Ausgangswert wird über den Adapter **AI_OUT** (Plug) bereitgestellt:

- **AI_OUT.D1** – INT-Wert, der das konvertierte BYTE darstellt.

Adapter

Bezeichnung	Typ	Richtung	Beschreibung
AB_IN	`adapter::types::unidirectional: AB`	Socket (Eingang)	BYTE-adapter, der den zu konvertierenden Wert sowie das Start-Ereignis liefert.
AI_OUT	`adapter::types::unidirectional: AI`	Plug (Ausgang)	INT-adapter, der den konvertierten Wert und das Bestätigungs-Ereignis bereitstellt.

Funktionsweise

1. Ein eingehendes Ereignis auf **AB_IN.E1** triggert die Verarbeitung.
2. Der von **AB_IN.D1** gelieferte BYTE-Wert wird an den internen Baustein `F_BYTE_TO_INT` übergeben.
3. `F_BYTE_TO_INT` führt die Typkonvertierung durch und stellt den INT-Wert an seinem Ausgang `OUT` bereit.
4. Nach erfolgreicher Konvertierung wird das Ereignis **AI_OUT.E1** gesendet, und der INT-Wert liegt an **AI_OUT.D1** an.

Die Konvertierung erfolgt ohne Wertebereichsverlust: BYTE (0…255) wird direkt auf INT abgebildet.

Technische Besonderheiten

- **Bit-Reinterpretation, werterhaltend** (Zielbreite ist gleich groß oder größer als der Quell-Bit-String).
- **Composite-FB:** Die Logik wird vollständig durch ein internes Netzwerk aus einem einzigen Konvertierungsbaustein realisiert.
- **Wiederverwendbar:** Der Baustein kann in Projekte eingebunden werden, die eine unidirektionale Adapter-Konvertierung benötigen.
- **Lizenz:** Der Baustein und die zugehörigen Materialien stehen unter der **Eclipse Public License 2.0**.
- **Paket:** `adapter::conversion::unidirectional`

Zustandsübersicht

Der FB besitzt keinen eigenen internen Zustandsautomaten. Das Verhalten wird ausschließlich durch die auslösenden Ereignisse und den eingebetteten Baustein `F_BYTE_TO_INT` bestimmt. Es gibt keine persistenten Zustände.

Anwendungsszenarien

- **Sensoranbindung:** Ein Sensor liefert Daten als BYTE über einen AB-Adapter; die nachgelagerte Verarbeitung erwartet INT-Werte über einen AI-Adapter.

- **Protokollanpassung:** In heterogenen Kommunikationspfaden müssen BYTE-kodierte Informationen in ein INT-basiertes Datenmodell überführt werden.
- **Adapter-Vermittlung:** Als Zwischenstück zwischen zwei Systemkomponenten, die unterschiedliche Adaptertypen verwenden.

Vergleich mit ähnlichen Bausteinen

Es existieren häufig direkte Konvertierungsbausteine wie `F_BYTE_TO_INT` (IEC-61131), die jedoch keine Adapter-Ereignissteuerung bieten. Der **AB_TO_AI** kapselt diese Konvertierung in einen vollständigen Adapter-zu-Adapter-Übergang mit Ereignisweitergabe. Ähnliche Bausteine für andere Datentypen (z. B. WORD zu DINT) könnten nach demselben Muster aufgebaut werden.

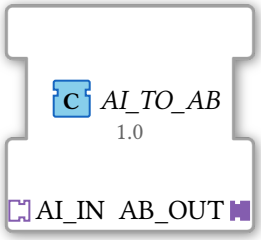
Fazit

Der **AB_TO_AI** ist ein nützlicher und einfacher Funktionsblock, der die Lücke zwischen BYTE- und INT-basierten Adapter-Schnittstellen schließt. Er erleichtert die Integration von Komponenten mit unterschiedlichen Datentypschnittstellen in der 4diac-IDE und folgt bewährten Entwurfsmustern der IEC-61499-Norm. Dank seiner Composite-Struktur ist er leicht erweiterbar und kann als Vorlage für ähnliche Adapter-Konvertierungen dienen.

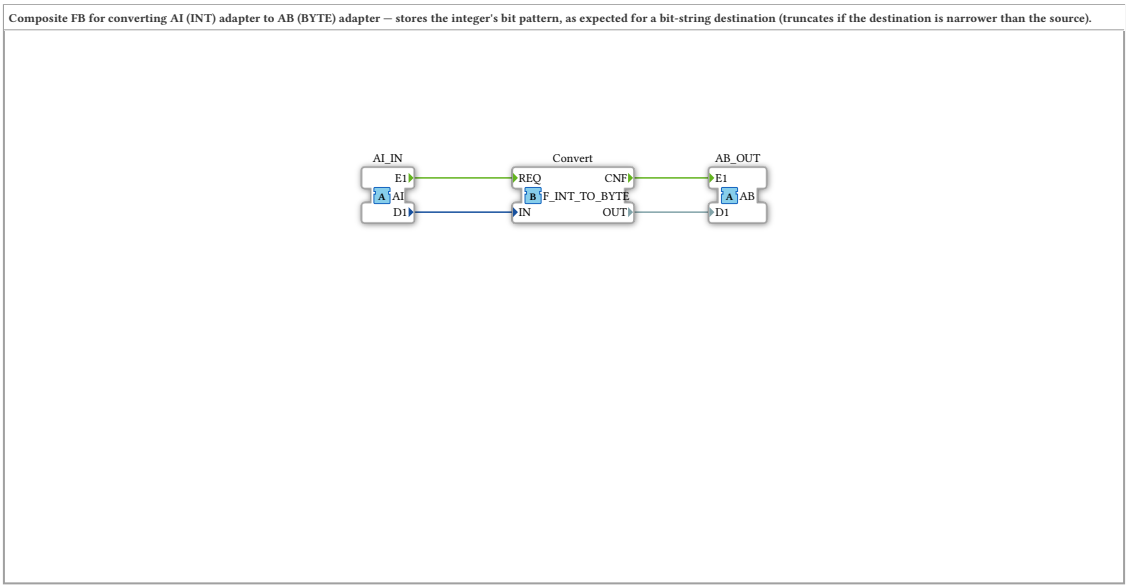
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AI_TO_AB



INT Adapter Input – *AI* — *AB* – BYTE Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Composite FB `AI_TO_AB` dient der Umwandlung eines **INT**-Werts, der über einen unidirektionalen **AI**-Adapter bereitgestellt wird, in einen **BYTE**-Wert und der Weiterleitung über einen unidirektionalen **AB**-Adapter. Die eigentliche Konvertierung erfolgt durch den intern genutzten FB `F_INT_TO_BYTE`. Der

Baustein ist Teil einer Adapter-Konvertierungsbibliothek und ermöglicht die nahtlose Integration von Komponenten mit unterschiedlichen Datentypen in IEC-61499-Systemen.

Schnittstellenstruktur

Ereignis-Eingänge
Keine eigenständigen Ereignis-Eingänge. Die ereignisgesteuerte Kommunikation erfolgt über den AI_IN-Adapter (siehe Adapter).

Ereignis-Ausgänge
Keine eigenständigen Ereignis-Ausgänge. Die ereignisgesteuerte Ausgabe erfolgt über den AB_OUT-Adapter (siehe Adapter).

Daten-Eingänge
Keine eigenständigen Daten-Eingänge. Der zu konvertierende INT-Wert wird über den AI_IN-Adapter bereitgestellt.

Daten-Ausgänge
Keine eigenständigen Daten-Ausgänge. Der konvertierte BYTE-Wert wird über den AB_OUT-Adapter ausgegeben.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	`AI_IN`	`adapter::types::unidirectional::AI`	Stellt den INT-Eingangswert (D1) sowie das auslösende Ereignis (E1) bereit.
Plug (Ausgang)	`AB_OUT`	`adapter::types::unidirectional::AB`	Gibt den konvertierten BYTE-Wert (D1) sowie das zugehörige Ereignis (E1) aus.

Funktionsweise
Der FB `AI_TO_AB` arbeitet als reiner Datenkonverter auf Basis einer Ereignissteuerung:

1. Am **AI_IN**-Adapter wird ein Ereignis **E1** empfangen.
2. Dieses Ereignis triggert den internen FB `Convert` vom Typ `F_INT_TO_BYTE` über dessen **REQ**-Eingang.
3. Gleichzeitig wird der INT-Datenwert von `AI_IN.D1` an den **IN**-Eingang von `Convert` weitergeleitet.
4. Nach Abschluss der Konvertierung erzeugt `Convert` ein Bestätigungsereignis (**CNF**).
5. Dieses Ereignis wird an **AB_OUT.E1** gesendet, und der konvertierte BYTE-Wert (`Convert.OUT`) wird an **AB_OUT.D1** übergeben.

Somit wird jeder eingehende INT-Wert durch ein Ereignis synchron in einen BYTE-Wert umgesetzt und über den Ausgangsadapter bereitgestellt.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Composite FB:** Der Baustein kapselt die Logik in einem Netzwerk aus einem einzelnen Konvertierungs-FB. Er ist selbst kein elementarer Baustein, sondern nutzt die Wiederverwendung von `F_INT_TO_BYTE`.
- **Adapterbasierte Schnittstelle:** Ereignisse und Daten werden ausschließlich über unidirektionale Adapter übertragen, wodurch eine lose Kopplung zwischen Quell- und Zielkomponenten erreicht wird.
- **Typkonvertierung:** Die eigentliche Umwandlung `INT -> BYTE` folgt der IEC 61131-3 Konvention (z.B. Wertebereich <0,255>; bei Überschreitung erfolgt ein Overflow). Dies ist abhängig von der Implementierung des verwendeten `F_INT_TO_BYTE`-Bausteins.
- **Keine interne Zustandsmaschine:** Der FB ist rein kombinatorisch mit ereignisgesteuerter Ausführung.

Zustandsübersicht

Da es sich um einen Composite FB handelt, besitzt `AI_TO_AB` keinen eigenen Zustandsautomaten. Der interne FB `F_INT_TO_BYTE` kann je nach Implementierung einen einfachen Zustandsautomaten (IDLE, EXECUTING) haben, der jedoch im Rahmen dieses Bausteins nicht sichtbar ist. Die Daten- und Ereignisflüsse sind deterministisch und werden durch das Zusammenspiel der Verbindungen gesteuert.

Anwendungsszenarien

- **Sensoranbindung:** Ein Sensor liefert INT-Werte, die von einer Steuerungseinheit als BYTE verarbeitet werden müssen (z.B. für einfache Ein-/Ausgabe-Karten).
- **Datenformatkonvertierung** in heterogenen IEC-61499-Systemen, um Komponenten mit unterschiedlichen Adaptertypen zu verbinden.
- **Adapter-Übersetzung** in Bibliotheken, die auf unidirektionale Kommunikation ausgelegt sind, z.B. bei der Verwendung von Feldbusprotokollen, die BYTE-orientierte Daten erwarten.

Vergleich mit ähnlichen Bausteinen

- **Direkte Konverter (z.B. `F_INT_TO_BYTE`):** Diese arbeiten auf Datenportebene und benötigen eine explizite Eventsteuerung. `AI_TO_AB` kapselt dies in einer adapterbasierten Schnittstelle.
- **Weitere Adapterkonverter:** Denkbar sind Bausteine wie `AI_TO_AW` (INT nach WORD) oder `AB_TO_AI` (BYTE nach INT). `AI_TO_AB` ist speziell auf die Kombination der beiden unidirektionalen Adapter `AI` und `AB` zugeschnitten.
- **Composite-Konverter vs. elementare Konverter:** Composite-FBs bieten eine höhere Abstraktion und Wiederverwendbarkeit, während elementare FBs mehr Kontrolle über die interne Logik erlauben.

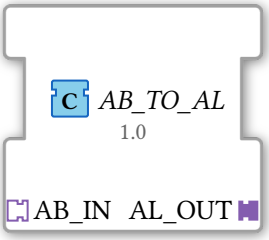
Fazit

Der Composite FB `AI_TO_AB` stellt eine effiziente, adapterbasierte Lösung zur Konvertierung von INT- zu BYTE-Werten dar. Er vereinfacht die Integration von Komponenten mit unterschiedlichen Datentypen in IEC-61499-Anwendungen, indem er die Konvertierungslogik in einer standardisierten, wiederverwendbaren Schnittstelle kapselt. Die ereignisgesteuerte Ausführung und die Nutzung eines etablierten Konvertierungsbausteins gewährleisten eine zuverlässige und deterministische Datenverarbeitung.

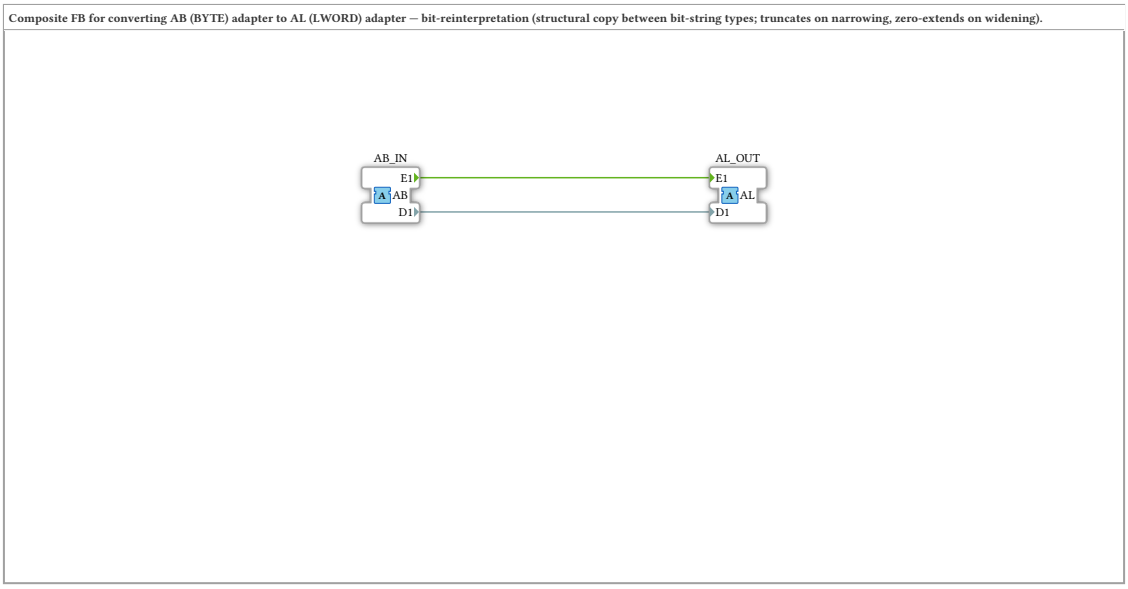
AB_AL

AB_TO_AL

BYTE Adapter Input – *AB*



AL – LWORD Adapter Output



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsbaustein `AB_TO_AL` ist ein Composite-Baustein zur Konvertierung eines unidirektionalen BYTE-Adapters (Typ `AB`) in einen unidirektionalen LWORD-Adapters (Typ `AL`). Er wird eingesetzt, um Daten von einem 8-Bit-Datenformat in ein 64-Bit-Datenformat zu überführen und dabei die Ereignissteuerung des Quelladapters auf den Zieladapter zu übertragen.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine direkten Ereigniseingänge. Der eingehende Ereignisfluss erfolgt indirekt über den Socket-Adapter `AB_IN`. Der Adapter `AB` stellt intern einen Ereigniseingang (`E1`) bereit, der mit dem Ereignisausgang des Zieladapters verbunden wird.

Ereignis-Ausgänge

Der Baustein besitzt keine direkten Ereignisausgänge. Der ausgehende Ereignisfluss erfolgt indirekt über den Plug-Adapter `AL_OUT`. Der Adapter `AL` stellt intern einen Ereignisausgang (`E1`) bereit, der durch die interne Verbindung aktiviert wird.

Daten-Eingänge

Name	Datentyp	Beschreibung
`AB_IN.D1`	BYTE	8-Bit-Eingangswert (über den Socket-Adapter)

Daten-Ausgänge

Name	Datentyp	Beschreibung
`AL_OUT.D1`	LWORD	64-Bit-Ausgangswert (über den Plug-Adapter)

Adapter

Name	Richtung	Adapter-Typ	Beschreibung
`AB_IN`	Socket	`adapter::types::unidirectional::AB`	Eingangsadapter, liefert BYTE-Daten und ein Ereignis
`AL_OUT`	Plug	`adapter::types::unidirectional::AL`	Ausgangsadapter, erwartet LWORD-Daten und gibt ein Ereignis weiter

Funktionsweise

Der Baustein ist als reine Verdrahtung (Composite) realisiert. Intern werden folgende Verbindungen hergestellt:

- Ereignis: `AB_IN.E1` ⟶ `AL_OUT.E1`
- Daten: `AB_IN.D1` ⟶ `AL_OUT.D1`

Sobald am Socket-Adapter `AB_IN` ein Ereignis eintrifft, wird der zugehörige BYTE-Wert (`D1`) an den Datenausgang `AL_OUT.D1` weitergegeben und gleichzeitig das Ereignis an `AL_OUT.E1` weitergeleitet. Dabei findet eine implizite Typkonvertierung von BYTE (8 Bit) nach LWORD (64 Bit) statt – die oberen 56 Bit werden mit Nullen aufgefüllt. Der Baustein besitzt keine eigene Zustandslogik oder Verzögerung.

Technische Besonderheiten

- **Bit-Reinterpretation** (strukturelle Kopie zwischen Bit-String-Typen; schneidet bei Verengung ab, erweitert mit Nullen bei Erweiterung).
- **Reiner Kompositbaustein**: Kein Algorithmus, keine Zustandsmaschine – die Funktionalität ergibt sich ausschließlich aus der internen Verdrahtung.
- **Unidirektionale Adapter**: Der Datenfluss ist nur von `AB_IN` zu `AL_OUT` möglich; eine Rückrichtung existiert nicht.
- **Automatische Typkonvertierung**: Die 4diac-IDE führt die Umwandlung von BYTE nach LWORD implizit durch (Big-Endian-Erweiterung mit Nullen).
- **Keine Parameter oder Konfiguration**: Der Baustein benötigt keine zusätzlichen Einstellungen.

Zustandsübersicht

Als Composite-Baustein ohne eigene Zustandsmaschine existiert kein definierter Zustand. Das Verhalten wird vollständig durch die eingehenden Ereignisse und Daten des Socket-Adapters bestimmt.

Anwendungsszenarien

- **Schnittstellenanpassung**: Ein Sensor liefert BYTE-Daten über einen AB-Adapter, das nachfolgende System erwartet jedoch LWORD-Daten über einen AL-Adapter.
- **Datenaufbereitung**: Vereinfachte Integration von 8-Bit-Geräten in 64-Bit-Steuerungsarchitekturen.
- **Protokollumsetzung**: Wenn Ereignisse und Daten über Adapter gekapselt werden, kann `AB_TO_AL` als universelle Konvertierungsstufe dienen.

Vergleich mit ähnlichen Bausteinen

Es existieren weitere Adapter-Konvertierungsbausteine wie `WORD_TO_DWORD`, `ARRAY_TO_STRUCT` oder spezifische Typwandler. `AB_TO_AL` zeichnet sich jedoch durch die vollständige Kapselung in Adapter aus – sowohl Ereignis als auch Daten werden gemeinsam konvertiert. Dies vereinfacht den Austausch kompletter Schnittstellen, ohne dass separate Data- und Event-Connections modelliert werden müssen.

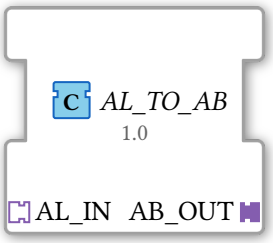
Fazit

`AB_TO_AL` ist ein einfacher, aber nützlicher Composite-Baustein zur Umwandlung eines BYTE-Adapters in einen LWORD-Adapter. Er reduziert den Modellierungsaufwand bei Typkonvertierungen in Adapter-basierten Netzwerken und stellt sicher, dass Ereignisse synchron zu den Daten weitergegeben werden. Aufgrund seiner klar definierten Schnittstelle eignet er sich besonders für modulare und wiederverwendbare Automatisierungslösungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

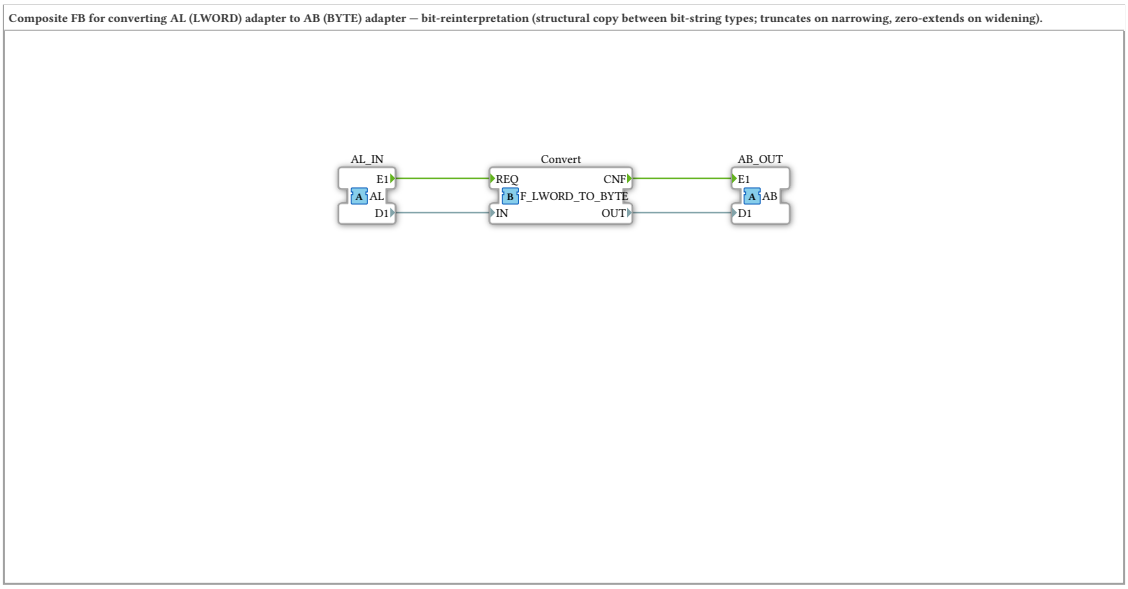
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AL_TO_AB



LWORD Adapter Input – *AL*

AB – BYTE Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AL_TO_AB** ist ein Composite-FB, der eine unidirektionale Konvertierung von einem LWORD-Adapter (AL) zu einem BYTE-Adapter (AB) ermöglicht. Er wird eingesetzt, um Daten eines LWORD-basierten Adapters in das Format eines BYTE-Adapters zu überführen, wobei die Konvertierung intern durch den IEC-61131-3-Baustein `F_LWORD_TO_BYTE` realisiert wird.

Schnittstellenstruktur

Adapter (Eingang)

Name	Typ	Beschreibung
AL_IN	`adapter::types::unidirectional::AL`	LWORD-Adapter-Socket. Über diesen Adapter werden die zu konvertierenden LWORD-Daten empfangen. Der Adapter stellt standardmäßig einen Ereignisausgang `E1` und einen Datenausgang `D1` (Datentyp LWORD) zur Verfügung.

Adapter (Ausgang)

Name	Typ	Beschreibung
AB_OUT	`adapter::types::unidirectional::AB`	BYTE-Adapter-Plug. Über diesen Adapter werden die konvertierten BYTE-Daten ausgegeben. Der Adapter besitzt einen Ereigniseingang `E1` und einen Dateneingang `D1` (Datentyp BYTE).

Hinweis: Der FB besitzt keine separaten Ereignis- oder Dateneingänge/-ausgänge; die gesamte Kommunikation erfolgt über die beiden Adapter-Schnittstellen.

Funktionsweise

1. Über den Socket `AL_IN` wird ein Ereignis (am Ereignisausgang `E1` des Adapters) empfangen, das gleichzeitig einen LWORD-Wert über den Datenausgang `D1` bereitstellt.
2. Dieses Ereignis triggert intern den Baustein `F_LWORD_TO_BYTE` (Typ: `iec61131::conversion::F_LWORD_TO_BYTE`), welcher die Umwandlung des LWORD-Wertes in einen BYTE-Wert vornimmt.
3. Nach erfolgreicher Konvertierung signalisiert `F_LWORD_TO_BYTE` den Abschluss über seinen Ereignisausgang `CNF`, der das Ereignis an den Plug `AB_OUT` weiterleitet.
4. Gleichzeitig wird der konvertierte BYTE-Wert über den Datenausgang `OUT` des Konverters an den Dateneingang `D1` von `AB_OUT` übergeben.
5. Der angeschlossene Empfänger am Plug `AB_OUT` erhält somit das Ereignis und den konvertierten Datenwert.

Technische Besonderheiten

- **Bit-Reinterpretation** (strukturelle Kopie zwischen Bit-String-Typen; schneidet bei Verengung ab, erweitert mit Nullen bei Erweiterung).
- **Einbindung von IEC-61131-3**: Die Konvertierung nutzt den standardisierten Baustein `F_LWORD_TO_BYTE` aus der IEC-61131-3-Bibliothek. Dadurch ist die Konvertierungslogik normkonform und portabel.
- **Composite-Struktur**: Der FB besteht ausschließlich aus einem internen Netzwerk, besitzt keinen eigenen Ablaufzustand (ECC) und arbeitet daher ereignisgesteuert ohne Verzögerung.
- **Unidirektionale Adapter**: Verwendet werden die unidirektionalen Adapter `AL` und `AB`, die eine klare Richtung der Daten- und Ereignisflüsse vorgeben.
- **Paketstruktur**: Der FB ist im Paket `adapter::conversion::unidirectional` organisiert, was die Wiederverwendung in modularen Projekten erleichtert.

Zustandsübersicht

Der Funktionsblock besitzt keinen eigenen Zustandsautomaten. Die gesamte Ablaufsteuerung erfolgt über die Ereigniskette des internen Netzwerks. Eine Zustandsüberwachung ist daher nicht erforderlich.

Anwendungsszenarien

- **Protokollanpassung**: Wenn in einem verteilten System ein Adapter, der LWORD-Daten liefert, durch einen Adapter ersetzt werden muss, der nur BYTE-Daten verarbeitet.
- **Systemintegration**: Kopplung von Komponenten, die unterschiedliche Adapterformate verwenden (z. B. AL-und AB-Adapter), ohne Änderungen an den originären Bausteinen vorzunehmen.
- **Datenreduktion**: Gezielte Umwandlung von 64-Bit-LWORD in 8-Bit-BYTE, wenn die höheren Bytes nicht benötigt werden oder eine schmalere Schnittstelle erforderlich ist.

Vergleich mit ähnlichen Bausteinen

Baustein	Funktion	Unterschied
`LWORD_TO_BYTE`	Reine Datentypkonvertierung ohne Adapter	Keine Adapter-Schnittstelle; direkte Daten- und Ereignisports
`AB_TO_AL`	Umgekehrte Konvertierung (BYTE → LWORD)	Gegensätzliche Richtung; ebenfalls Composite mit Adaptern
`WORD_TO_BYTE`	Konvertierung von WORD (16 Bit) zu BYTE	Anderer Datentyp; ähnliche Struktur, aber andere Adaptertypen

Der **AL_TO_AB** ist speziell auf die Verwendung der unidirektionalen Adapter `AL` und `AB` zugeschnitten und bietet eine saubere, wiederverwendbare Kapselung.

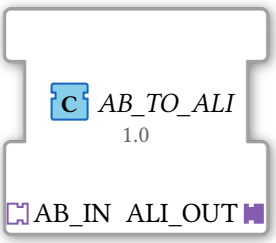
Fazit

Der Funktionsblock `AL_TO_AB` stellt eine einfache, aber effektive Lösung zur Umwandlung eines LWORD-Adapter-Signals in ein BYTE-Adapter-Signal dar. Durch die Verwendung eines standardisierten IEC-61131-3-Konverters und die konsequente Adapter-Schnittstelle ist er leicht in bestehende Systeme integrierbar und fördert einen modularen, wartbaren Entwurf.

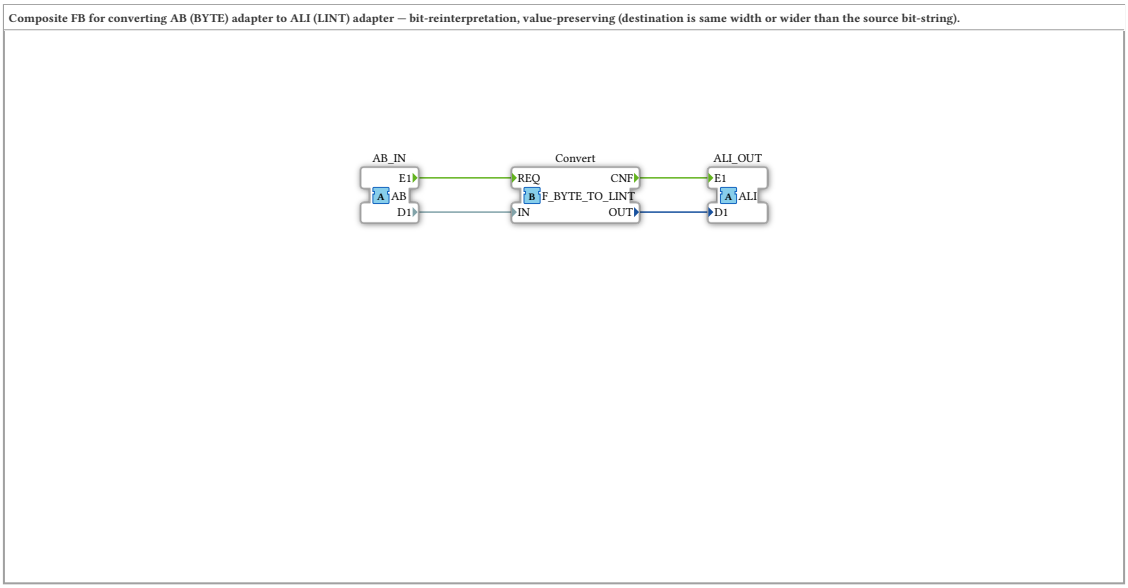
AB_ALI

AB_TO_ALI

BYTE Adapter Input – *AB*



ALI – LINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AB_TO_ALI` ist ein Composite-Baustein, der eine Konvertierung von einem BYTE-Adapter (`AB`) zu einem LINT-Adapter (`ALI`) realisiert. Er ermöglicht die nahtlose Integration zwischen Schnittstellen, die auf unterschiedlichen Datentypen basieren, und stellt die Datenübergabe über standardisierte Adapter bereit.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge des Funktionsblocks. Ereignisse werden über den Socket-Adapter `AB_IN` empfangen.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge des Funktionsblocks. Ereignisse werden über den Plug-Adapter `ALI_OUT` ausgegeben.

Daten-Eingänge

Keine direkten Daten-Eingänge des Funktionsblocks. Daten werden über den Socket-Adapter `AB_IN` empfangen.

Daten-Ausgänge

Keine direkten Daten-Ausgänge des Funktionsblocks. Daten werden über den Plug-Adapter `ALI_OUT` ausgegeben.

Adapter

- **Socket `AB_IN`** (Typ: `adapter::types::unidirectional::AB`) Empfängt ein BYTE-Signal (über den Datenport `D1`) sowie ein Ereignis (über den Ereignisport `E1`).
- **Plug `ALI_OUT`** (Typ: `adapter::types::unidirectional::ALI`) Sendet ein LINT-Signal (über den Datenport `D1`) sowie ein Ereignis (über den Ereignisport `E1`).

Funktionsweise

Der Funktionsblock arbeitet als reine Datenkonvertierungsschleife:

1. Ein eingehendes Ereignis am Socket `AB_IN` (Port `E1`) triggert den internen Konvertierungsbaustein `F_BYTE_TO_LINT`.
2. Der ebenfalls am Socket anliegende BYTE-Wert (`D1`) wird an den Eingang `IN` des Konverters übergeben.
3. Der Konverter wandelt den BYTE-Wert in einen LINT-Wert um und stellt ihn an seinem Ausgang `OUT` bereit.
4. Das Bestätigungsereignis des Konverters (`CNF`) wird an den Plug `ALI_OUT` (Port `E1`) weitergeleitet, und der konvertierte LINT-Wert wird auf dem Datenport `D1` des Plugs ausgegeben.

Technische Besonderheiten

- **Bit-Reinterpretation, werterhaltend** (Zielbreite ist gleich groß oder größer als der Quell-Bit-String).
- Es handelt sich um einen **Composite-Baustein**, der intern den Standard-Funktionsblock `iec61131::conversion::F_BYTE_TO_LINT` verwendet.
- Die Konvertierung erfolgt **ereignisgesteuert** – jede Änderung am Eingang löst eine Verarbeitung aus.
- Der Baustein ist für unidirektionale Adapter ausgelegt und unterstützt keine Rückwärtskonvertierung.
- Die Typinformationen der Adapter (`AB` und `ALI`) sind als unidirektionale Schnittstellen definiert und enthalten je einen Ereignis- und einen Datenport.

Zustandsübersicht

Der Funktionsblock besitzt keinen eigenen Zustandsautomaten. Die Konvertierung erfolgt **stateless** – bei jedem eingehenden Ereignis wird der aktuelle Wert sofort umgesetzt und ausgegeben. Es gibt keine Verzögerung, Pufferung oder interne Zustände.

Anwendungsszenarien

- **Integration von Feldbussystemen** oder Sensoren, die BYTE-Daten liefern, in eine Steuerungsumgebung, die LINT-Werte erwartet (z. B. zur weiteren Verarbeitung in Zählern oder langen Integer-Operationen).
- **Adapter-Konvertierung** in modularen Funktionsbaustein-Netzwerken, bei denen die Schnittstellen unterschiedliche Datentypen verwenden (z. B. in der Agrar- oder Automatisierungstechnik).
- **Brückenbildung** zwischen älteren BYTE-basierten Modulen und modernen LINT-basierten Komponenten.

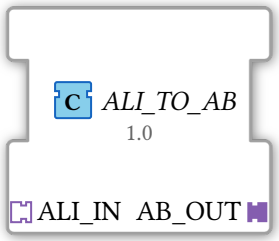
Vergleich mit ähnlichen Bausteinen

- `AB_TO_ALI` ist spezifisch für die Konvertierung von **BYTE** nach **LINT**. Es gibt analoge Bausteine für andere Datentyp-Kombinationen (z. B. `WORD_TO_DINT`, `BYTE_TO_INT`).
- Im Gegensatz zu generischen Konvertern oder Skalierungsbausteinen ist dieser FB auf eine feste Typumwandlung optimiert und benötigt keine Parametrierung.
- Der Einsatz von Adaptern anstelle direkter Ein-/Ausgangsports ermöglicht eine höhere Wiederverwendbarkeit und eine saubere Trennung der Schnittstellen.

Fazit

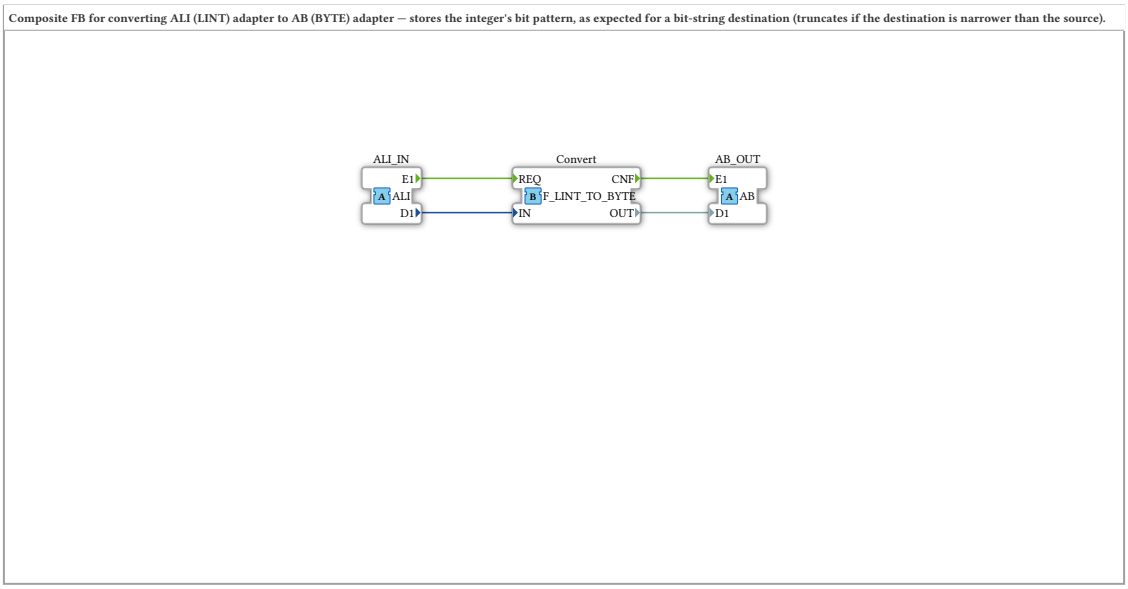
Der `AB_TO_ALI`-Funktionsblock bietet eine einfache und zuverlässige Möglichkeit, BYTE-Daten über einen Adapter in LINT-Daten umzuwandeln. Durch die Verwendung von Standard-Konvertierungsbausteinen und einer klaren ereignisgesteuerten Architektur eignet er sich ideal für modulare Automatisierungssysteme, die eine konsistente Datenkonvertierung zwischen verschiedenen Adaptertypen erfordern.

ALI_TO_AB



LINT Adapter Input – *ALI*

AB – BYTE Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ALI_TO_AB** ist ein zusammengesetzter Baustein (Composite FB) gemäß IEC 61499-2, der zur Umwandlung eines LINT-Werts (64-Bit Ganzzahl) in einen BYTE-Wert (8-Bit) dient. Die Konvertierung erfolgt über zwei unidirektionale Adapter-Schnittstellen: Der ALI_Empfänger (Socket) nimmt einen LINT-Wert entgegen, und der AB_Sender (Plug) gibt den konvertierten BYTE-Wert weiter. Intern wird der Konvertierungsbaustein `F_LINT_TO_BYTE` aus der IEC 61131-Bibliothek verwendet.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine direkten Ereignis-Eingänge. Ereignisse werden ausschließlich über den Adapter **ALI_IN** (Socket) empfangen.

Ereignis	Beschreibung
E1 (über ALI_IN)	Signal zur Auslösung der Konvertierung.

Ereignis-Ausgänge

Der FB besitzt keine direkten Ereignis-Ausgänge. Ereignisse werden ausschließlich über den Adapter **AB_OUT** (Plug) gesendet.

Ereignis	Beschreibung
E1 (über AB_OUT)	Quittung nach erfolgreicher Konvertierung.

Daten-Eingänge

Der FB besitzt keine direkten Daten-Eingänge. Daten werden ausschließlich über den Adapter **ALI_IN** (Socket) bereitgestellt.

Daten	Typ	Beschreibung
D1 (über ALI_IN)	LINT	64-Bit Ganzzahl, die konvertiert werden soll.

Daten-Ausgänge

Der FB besitzt keine direkten Daten-Ausgänge. Daten werden ausschließlich über den Adapter **AB_OUT** (Plug) ausgegeben.

Daten	Typ	Beschreibung
D1 (über AB_OUT)	BYTE	8-Bit konvertierter Wert.

Adapter

Adapter	Richtung	Typ	Beschreibung
ALI_IN	Socket (Eingang)	`adapter::types::unidirectional::ALI`	Nimmt den LINT-Wert sowie ein zugehöriges Ereignis entgegen.
AB_OUT	Plug (Ausgang)	`adapter::types::unidirectional::AB`	Gibt den konvertierten BYTE-Wert sowie ein Quittungsereignis aus.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis am **ALI_IN**-Adapter (E1).
2. Sobald das Ereignis eintrifft, wird der Datenwert D1 (LINT) an den internen Konvertierungsbaustein `F_LINT_TO_BYTE` übergeben und dort in einen BYTE-Wert umgewandelt.
3. Nach Abschluss der Konvertierung wird der BYTE-Wert über den **AB_OUT**-Adapter (D1) ausgegeben und das Quittungsereignis (E1) gesendet.

Die Umwandlung von LINT (64 Bit) nach BYTE (8 Bit) erfolgt durch Abschneiden der niederwertigsten 8 Bits. Höherwertige Bits gehen dabei verloren.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Typkonvertierung mit Werteverlust:** Da LINT 64 Bit breit ist, BYTE jedoch nur 8 Bit, werden nur die unteren 8 Bit des LINT-Werts übernommen. Ein Wertebereich von 0 … 255 wird korrekt abgebildet; größere Zahlen werden modulo 256 reduziert.
- **Reine Adapter-Schnittstelle:** Der Baustein besitzt keine direkten Ein-/Ausgänge, sondern kommuniziert ausschließlich über zwei unidirektionale Adapter. Dies ermöglicht eine lose Kopplung in adapterbasierten Architekturen.
- **Standardkonformität:** Der FB setzt auf den IEC 61131-Baustein `F_LINT_TO_BYTE` auf und ist daher in Umgebungen einsetzbar, die diese Bibliothek unterstützen.

Zustandsübersicht

Als zusammengesetzter Baustein besitzt **ALI_TO_AB** keinen eigenen internen Zustandsautomaten. Die Zustände des enthaltenen `F_LINT_TO_BYTE` (z. B. IDLE, BUSY) sind für den Anwender nicht sichtbar. Der FB verhält sich daher wie eine kombinatorische oder ereignisgesteuerte Funktion ohne explizite Zustandsmaske.

Anwendungsszenarien

- **Adapterbasierte Datenkonvertierung:** In Systemen, die über Adapter mit ALI- und AB-Schnittstellen verbunden sind, kann ALI_TO_AB als Brücke zwischen einem LINT-liefernden und einem BYTE-empfangenden Modul eingesetzt werden.
- **Reduzierung der Datenbreite:** Wenn ein 64-Bit-Wert (z. B. ein Zählerstand) auf ein 8-Bit-Signal abgebildet werden soll (z. B. ein einfacher Status-BYTE), ist dieser Baustein geeignet.
- **Protokollumsetzung:** In heterogenen Steuerungssystemen, in denen unterschiedliche Adaptertypen verwendet werden, ermöglicht ALI_TO_AB eine einfache Typumwandlung.

Vergleich mit ähnlichen Bausteinen

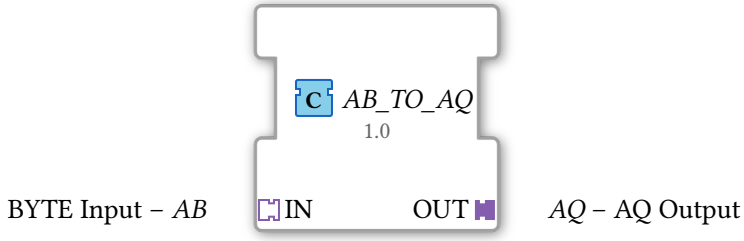
- **ALI_TO_Ax (LINT nach ANY):** Ein allgemeinerer Konverter, der LINT in einen beliebigen anderen Datentyp umwandelt. ALI_TO_AB ist auf BYTE spezialisiert und dadurch einfacher und direkter.
- **F_LINT_TO_BYTE:** Der zugrunde liegende IEC 61131-Baustein. ALI_TO_AB kapselt diesen in eine adapterbasierte Schnittstelle, sodass er in Umgebungen verwendet werden kann, die nur Adapterkommunikation unterstützen.
- **Direkte Adapterverbindung:** Ohne Konvertierung könnte ein ALI-Adapter nicht direkt mit einem AB-Adapter verbunden werden. ALI_TO_AB schließt diese Lücke.

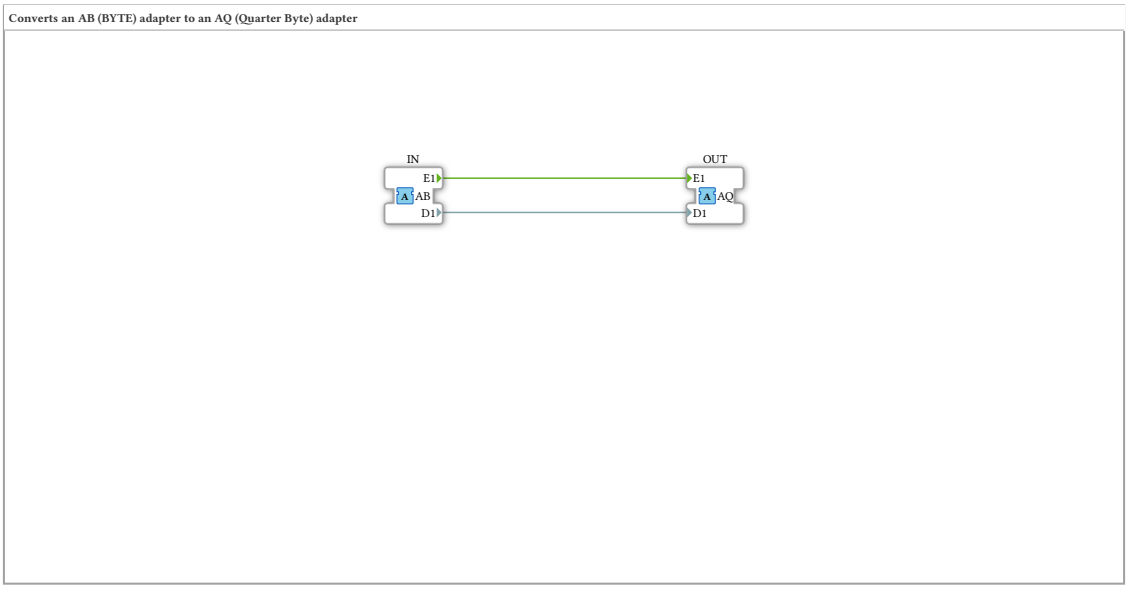
Fazit

ALI_TO_AB ist ein einfacher, aber nützlicher Konvertierungsbaustein für die adapterbasierte Kommunikation. Er ermöglicht die Umwandlung eines LINT-Werts in einen BYTE-Wert unter Verwendung eines standardisierten IEC 61131-Bausteins. Durch die vollständige Kapselung in zwei unidirektionale Adapter lässt er sich nahtlos in Adapter-Netzwerke integrieren, ohne dass der Anwender die interne Typumwandlung selbst implementieren muss. Der Werteverlust durch die reduzierte Datenbreite ist bei der Verwendung zu beachten.

AB_AQ

AB_TO_AQ





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AB_TO_AQ** dient der Konvertierung eines unidirektionalen Adapters vom Typ **AB (BYTE)** in einen Adapter vom Typ **AQ (Quarter Byte)**. Er übernimmt die direkte Durchleitung eines Ereignisses und eines Datenwerts von der Eingangsschnittstelle zur Ausgangsschnittstelle, wobei die Datenbreite implizit von 8 Bit auf 2 Bit reduziert wird.

Schnittstellenstruktur

Ereignis-Eingänge
- **IN.E1** (über Adapter IN): Ereigniseingang des BYTE-Adapters.

Ereignis-Ausgänge
- **OUT.E1** (über Adapter OUT): Ereignisausgang des Quarter-Byte-Adapters.

Daten-Eingänge
- **IN.D1** (über Adapter IN): Dateneingang des BYTE-Adapters (1 Byte).

Daten-Ausgänge
- **OUT.D1** (über Adapter OUT): Datenausgang des Quarter-Byte-Adapters (1 Quarter Byte).

Adapter

Adapter	Richtung	Typ	Beschreibung
IN	Socket	`adapter::types::unidirectional::AB`	BYTE-Eingangsadapter (8 Bit)
OUT	Plug	`adapter::types::unidirectional::AQ`	Quarter-Byte-Ausgangsadapter (2 Bit)

Funktionsweise
Der Baustein leitet ein eingehendes Ereignis (E1) und die zugehörigen Daten (D1) direkt vom IN-Adapter zum OUT-Adapter weiter. Dabei erfolgt keine interne Verarbeitung oder Zustandsänderung; die Daten werden durchschleifen. Die Konvertierung von einem 8-Bit- auf ein 2-Bit-Format wird durch die unterschiedlichen Adaptertypen bestimmt – üblicherweise werden die niederwertigsten Bits (LSB) des BYTE-Werts in das Quarter Byte übernommen, sofern nicht anders spezifiziert.

Technische Besonderheiten
- Reiner Durchschleifbaustein ohne eigene Zustandslogik (keine ECC).
- Ereignis- und Datenpfade sind direkt miteinander verbunden: `IN.E1 → OUT.E1` und `IN.D1 → OUT.D1`.
- Die Datenreduktion von 8 Bit auf 2 Bit erfolgt allein durch die Typanpassung; eine explizite Wandlung (z. B. Beschneidung oder Rundung) ist nicht im FB selbst implementiert.
- Der Baustein ist als unidirektionaler Adapter-Konverter ausgelegt und unterstützt keine bidirektionale Kommunikation.

Zustandsübersicht
Es existiert keine Zustandsmaschine. Der FB arbeitet rein kombinatorisch: Jedes Ereignis am Eingang wird sofort als Ereignis am Ausgang weitergeleitet, begleitet vom aktuellen Datenwert.

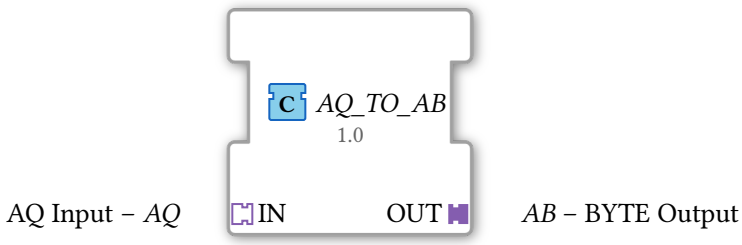
Anwendungsszenarien
- Anbindung einer Byte-orientierten Schnittstelle an eine Logik, die nur zwei Bit breite Daten verarbeitet (z. B. Statusbits oder Schaltsignale).
- Reduzierung der Datenbreite in einer Adapterkaskade, wenn die ursprüngliche Byte-Information auf die unteren zwei Bits komprimiert werden kann.
- Test- und Simulationsumgebungen, in denen unterschiedliche Adapterformate miteinander verbunden werden müssen.

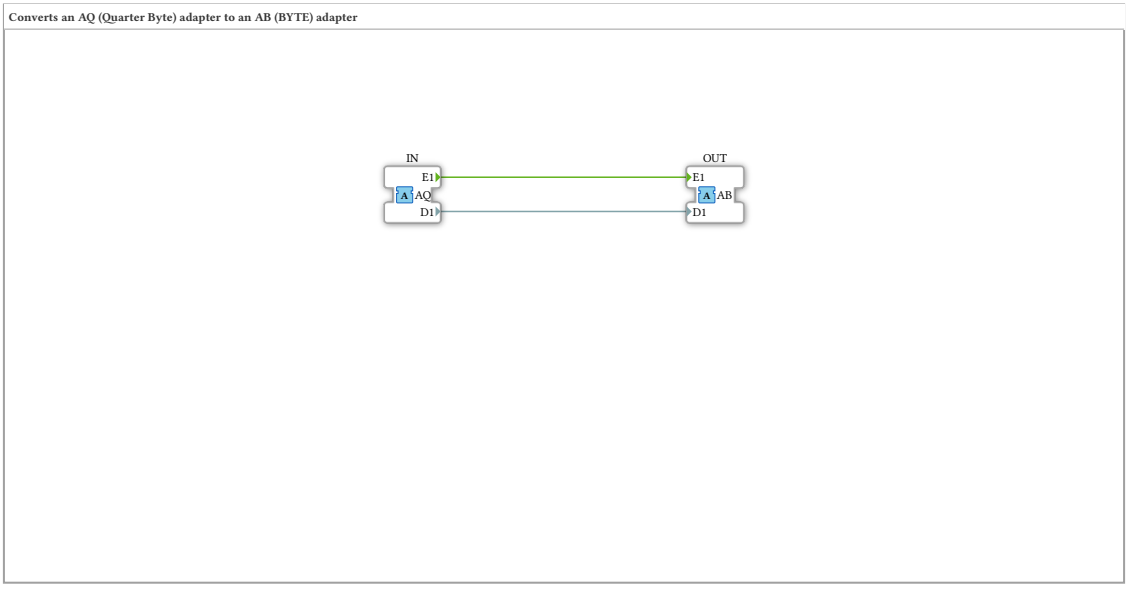
Vergleich mit ähnlichen Bausteinen
- **AB_TO_AQ** ähnelt anderen unidirektionalen Adapter-Konvertern wie z. B. `AB_TO_AB` (identische Typen) oder `AQ_TO_AB` (Erweiterung auf ein Byte). Der Unterschied liegt in der Datenbreite und der Art der impliziten Umwandlung.
- Im Gegensatz zu Wandlern mit expliziten Algorithmen (z. B. Skalierung oder Bitmaskierung) bietet dieser FB keine Konfigurationsmöglichkeiten – die Umwandlung ist fest vorgegeben.

Fazit
Der **AB_TO_AQ** Funktionsblock ist ein einfacher, aber nützlicher Adapter-Konverter, der eine direkte elektrische und logische Verbindung zwischen zwei unterschiedlichen unidirektionalen Adaptertypen

herstellt. Er eignet sich hervorragend für Fälle, in denen eine Byte-Information auf die wichtigsten zwei Bits reduziert weitergegeben werden soll, ohne zusätzliche Verarbeitungslast zu erzeugen.

AQ_TO_AB





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AQ_TO_AB` dient der Konvertierung eines AQ-Adapters (Quarter Byte) in einen AB-Adapter (BYTE). Er ermöglicht die nahtlose Verbindung von Komponenten, die unterschiedliche Adapter-Schnittstellen verwenden, indem er die Signale unverändert durchleitet. Der Baustein ist als reiner „Durchschleif"-FB realisiert, der keine Datenmanipulation vornimmt.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine expliziten Ereignis-Eingänge auf oberster Ebene. Die Ereignisübertragung erfolgt über die Adapter-Schnittstellen.

Ereignis-Ausgänge

Der FB besitzt keine expliziten Ereignis-Ausgänge auf oberster Ebene. Die Ereignisübertragung erfolgt über die Adapter-Schnittstellen.

Daten-Eingänge

Der FB besitzt keine expliziten Daten-Eingänge auf oberster Ebene. Die Datenübertragung erfolgt über die Adapter-Schnittstellen.

Daten-Ausgänge

Der FB besitzt keine expliziten Daten-Ausgänge auf oberster Ebene. Die Datenübertragung erfolgt über die Adapter-Schnittstellen.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AQ`	IN	Eingang (Socket)	AQ-Adapter als Eingangsschnittstelle für Quarter-Byte-Daten
`adapter::types::unidirectional::AB`	OUT	Ausgang (Plug)	AB-Adapter als Ausgangsschnittstelle für BYTE-Daten

Funktionsweise

Der Baustein verbindet intern die Ereignis- und Datenkanäle des Eingangsadapters `IN` direkt mit den entsprechenden Kanälen des Ausgangsadapters `OUT`. Konkret werden folgende Verbindungen hergestellt:

- Ereignisverbindung: `IN.E1` ⟶ `OUT.E1`
- Datenverbindung: `IN.D1` ⟶ `OUT.D1`

Somit werden alle ankommenden Ereignisse und Daten des AQ-Adapters unverändert an den AB-Adapter weitergegeben. Der FB selbst führt keine Verarbeitung oder Konvertierung der Nutzdaten durch – er fungiert ausschließlich als Schnittstellenadapter.

Technische Besonderheiten

- **Unidirektionaler Datenfluss:** Der FB unterstützt nur eine Richtung (AQ ⟶ AB).
- **Keine Zustandsänderung:** Da keinerlei Logik oder Speicher vorhanden ist, handelt es sich um einen rein kombinatorischen Baustein.
- **Einfaches Durchschleifen:** Die Implementierung beschränkt sich auf zwei direkte Verbindungen im FBNetwork.
- **Adapter-Konvertierung:** Der FB erlaubt die Integration von AQ-kompatiblen Komponenten in ein System, das AB-Adapter erwartet, ohne dass Anpassungen der angeschlossenen Bausteine nötig sind.

Zustandsübersicht

Der FB besitzt keinen Zustandsautomaten. Das Verhalten ist deterministisch: Jedes eingehende Ereignis triggert sofort das entsprechende Ausgangsereignis, und jeder eingehende Datenwert wird unverzögert zum Ausgang weitergeleitet.

Anwendungsszenarien

- **Systemintegration:** Verbindung eines Sensormoduls mit AQ-Schnittstelle an eine Steuerung, die AB-Adapter verwendet.
- **Protokollumsetzung:** Nutzung in Bibliotheken, um unterschiedliche Adaptertypen kompatibel zu machen.
- **Testumgebungen:** Einsatz als einfacher Adapter in Testaufbauten, um Schnittstellen anzupassen.

Vergleich mit ähnlichen Bausteinen

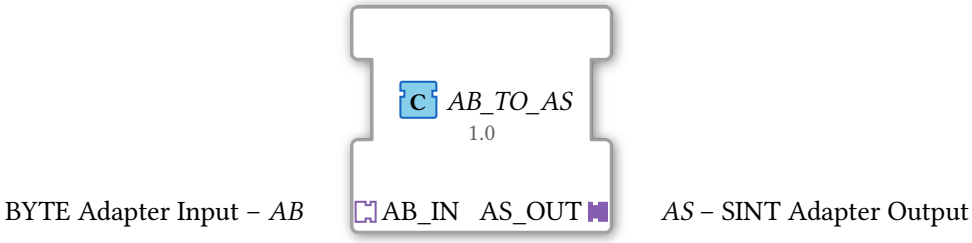
- **AB_TO_AQ:** Führt die umgekehrte Konvertierung (AB ⟶ AQ) durch.
- **AQ_TO_BYTE_EXT:** Ein hypothetischer FB, der zusätzlich Datenkonvertierungen (z. B. Skalierung, Formatierung) vornimmt, während `AQ_TO_AB` ausschließlich die Schnittstelle wechselt.
- **Generische Adapter-Wrapper:** Andere Plattformen bieten oft generische Konvertierungsbausteine, die jedoch komplexer konfiguriert werden müssen. `AQ_TO_AB` ist spezialisiert und dadurch sehr leichtgewichtig.

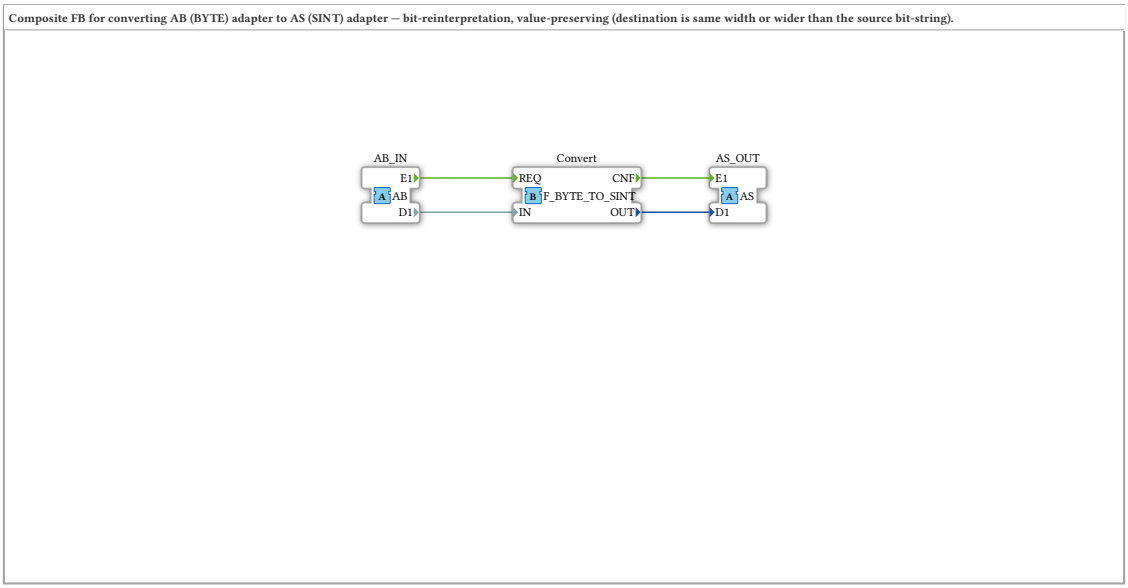
Fazit

Der Funktionsblock `AQ_TO_AB` bietet eine schlanke und effiziente Lösung zur Adapterkonvertierung von AQ auf AB. Durch seine einfache Durchschleif-Struktur ist er zuverlässig, ressourcenschonend und ideal für Situationen, in denen lediglich die Schnittstellenform geändert werden muss, ohne die Signale zu beeinflussen. Er eignet sich besonders für modulare Systeme mit unterschiedlichen Adapterstandards.

AB_AS

AB_TO_AS





• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AB_TO_AS** ist ein Composite-Funktionsbaustein (FB), der einen unidirektionalen AB-Adapter (BYTE) in einen unidirektionalen AS-Adapter (SINT) umwandelt. Er dient als konvertierende Brücke zwischen zwei verschiedenen Datentypen in einer IEC 61499-basierten Steuerungsanwendung.

Schnittstellenstruktur

Der FB besitzt keine eigenen Ereignis- oder Datenports auf der obersten Ebene. Die gesamte Kommunikation erfolgt über zwei Adapter-Schnittstellen:

Ereignis-Eingänge

- **E1** (über **AB_IN**, Socket, Typ `adapter::types::unidirectional::AB`) Dieses Ereignis wird vom angeschlossenen Quell-Adapter empfangen und löst die Konvertierung eines BYTE-Wertes in einen SINT-Wert aus.

Ereignis-Ausgänge

- **E1** (über **AS_OUT**, Plug, Typ `adapter::types::unidirectional::AS`) Dieses Ereignis wird nach erfolgreicher Konvertierung an den Ziel-Adapter gesendet, um den neuen SINT-Wert zu signalisieren.

Daten-Eingänge

- **D1** (BYTE, über **AB_IN**-Adapter) Enthält den zu konvertierenden BYTE-Wert. Er wird über das Ereignis **E1** am Eingang abgetastet.

Daten-Ausgänge

- **D1** (SINT, über **AS_OUT**-Adapter) Stellt den konvertierten SINT-Wert bereit. Er wird nach Abschluss der Konvertierung mit dem Ereignis **E1** am Ausgang gültig.

Adapter

Adapter	Richtung	Typ	Beschreibung
AB_IN	Socket (Eingang)	`adapter::types::unidirectional::AB`	Nimmt BYTE-Daten und ein Trigger-Ereignis vom Quellsystem entgegen.
AS_OUT	Plug (Ausgang)	`adapter::types::unidirectional::AS`	Stellt die konvertierten SINT-Daten mit einem Bestätigungsereignis für das Zielsystem bereit.

Funktionsweise

Der **AB_TO_AS**-FB enthält intern einen einzelnen Funktionsbaustein des Typs `iec61131::conversion::F_BYTE_TO_SINT`. Dieser Baustein führt die eigentliche Datentypkonvertierung von BYTE auf SINT durch.

1. Ein eingehendes Ereignis **AB_IN.E1** triggert den **F_BYTE_TO_SINT**-Baustein über dessen **REQ**-Eingang.
2. Gleichzeitig wird der Datenwert von **AB_IN.D1** (BYTE) auf den **IN**-Eingang des Konverters gelegt.
3. Nach der Verarbeitung signalisiert der Konverter sein **CNF**-Ereignis, das an den Adapter-Ausgang **AS_OUT.E1** weitergeleitet wird.
4. Der konvertierte SINT-Wert wird vom **OUT**-Ausgang des Konverters auf **AS_OUT.D1** übertragen.

Damit ist eine asynchrone, ereignisgesteuerte Übersetzung eines BYTE-Werts in einen SINT-Wert realisiert.

Technische Besonderheiten

- **Bit-Reinterpretation, werterhaltend** (Zielbreite ist gleich groß oder größer als der Quell-Bit-String).
- **Composite-FB**: Die Logik ist vollständig in der internen FBNetzwerk-Struktur gekapselt. Dies erleichtert Wartung und Wiederverwendung.
- **Lizenz**: Der Baustein wird unter der **Eclipse Public License 2.0** bereitgestellt (SPDX-Lizenzkennung: EPL-2.0).
- **Version**: 1.0, veröffentlicht 2026.
- **Abhängigkeiten**: Der interne Konverterbaustein `F_BYTE_TO_SINT` aus der Bibliothek `iec61131::conversion` wird vorausgesetzt.

Zustandsübersicht

Der **AB_TO_AS** besitzt keinen eigenen Zustandsautomaten. Die Verarbeitung erfolgt rein datenfluss- und ereignisgesteuert über den eingebetteten **F_BYTE_TO_SINT**-Baustein. Somit gibt es keine expliziten Zustände, lediglich die impliziten Abläufe: *Warten auf Eingabeereignis*, *Konvertierung aktiv* und *Ausgabeereignis gesendet*.

Anwendungsszenarien

- **Systemintegration**: Verbindung zweier Subsysteme, die unterschiedliche Adapter-Datentypen (BYTE vs. SINT) verwenden.
- **Datenformatumstellung**: Wenn ein bestehender Steuerungsalgorithmus SINT-Werte erwartet, die Quelle aber nur BYTE-Adapter bereitstellt.
- **Protokollanpassung**: In heterogenen Automatisierungsumgebungen, in denen Bausteine verschiedener Hersteller oder Bibliotheken kombiniert werden müssen.

Vergleich mit ähnlichen Bausteinen

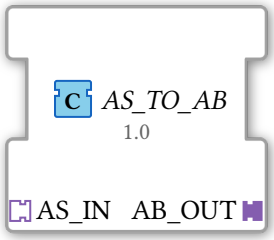
- **F_BYTE_TO_SINT** (direkter FB): Bietet dieselbe Konvertierungslogik, jedoch ohne Adapter-Kapselung. Erfordert manuelle Verkabelung von Ereignis- und Datenleitungen.
- **AB_TO_INT** oder **AB_TO_USINT**: Ähnliche Adapter-Konverter für andere Ziel-Datentypen. Der **AB_TO_AS** ist speziell auf SINT ausgelegt.
- **Eigene Implementierung**: Der Composite-Bauweise erlaubt eine einfache Erweiterung, z. B. durch Hinzufügen von Plausibilitätsprüfungen oder Skalierungen, ohne die Grundfunktion zu verändern.

Fazit

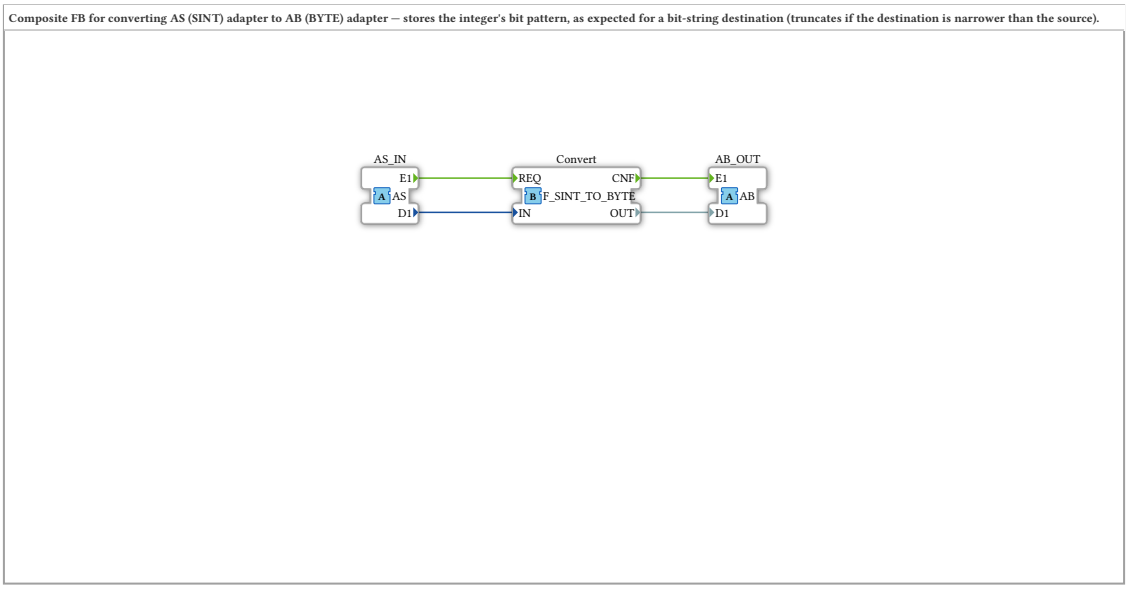
Der **AB_TO_AS** ist ein spezialisierter, aber einfach aufgebauter Composite-FB, der eine saubere und wiederverwendbare Lösung zur Adapter-basierten Datentypkonvertierung von BYTE zu SINT bietet. Durch die Nutzung eines standardisierten Konverterbausteins und die klare Ereignissteuerung fügt er sich nahtlos in IEC-61499-Netzwerke ein. Seine Lizenzierung unter EPL-2.0 fördert die offene Nutzung und Weiterentwicklung.

AS_TO_AB

SINT Adapter Input – *AS*



AB – BYTE Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AS_TO_AB** ist ein Composite-FB, der einen AS-Adapter (SINT) in einen AB-Adapter (BYTE) umwandelt. Er dient als Brücke zwischen unterschiedlichen Adapter-Typen und ermöglicht die Konvertierung des Datentyps innerhalb einer Adapter-basierten Kommunikationskette. Der Baustein nutzt intern die IEC-61131-Funktion **F_SINT_TO_BYTE** und ist als reiner Datenkonverter ohne Zustandslogik realisiert.

Schnittstellenstruktur

Da der FB über keine direkten Ereignis- oder Datenports verfügt, werden Ein- und Ausgänge über die folgenden Adapter bereitgestellt:

Ereignis-Eingänge

Port	Beschreibung
`AS_IN.E1`	Startet die Konvertierung eines SINT-Werts.

Ereignis-Ausgänge

Port	Beschreibung

AB_OUT.E1	Signalisiert den Abschluss der Konvertierung.

Daten-Eingänge

Port	Datentyp	Beschreibung
AS_IN.D1	SINT	Der zu konvertierende Wert (Signed Short Integer).

Daten-Ausgänge

Port	Datentyp	Beschreibung
AB_OUT.D1	BYTE	Der konvertierte Wert (Byte).

Adapter

Bezeichnung	Typ	Richtung	Kommentar
AS_IN	adapter::types::unidirectional::AS	Socket (Eingang)	SINT Adapter Eingang
AB_OUT	adapter::types::unidirectional::AB	Plug (Ausgang)	BYTE Adapter Ausgang

Funktionsweise

1. Ein eingehendes Ereignis auf AS_IN.E1 triggert die Verarbeitung.
2. Der aktuelle Wert von AS_IN.D1 (SINT) wird ausgelesen.
3. Die interne Instanz F_SINT_TO_BYTE führt die Umwandlung in einen BYTE-Wert durch.
4. Nach erfolgreicher Konvertierung wird ein Ereignis auf AB_OUT.E1 ausgegeben und der berechnete BYTE-Wert auf AB_OUT.D1 bereitgestellt.

Die Konvertierung erfolgt streng nach IEC-61131-Semantik: Der SINT-Wert wird bitweise auf den BYTE-Datentyp abgebildet, ohne Vorzeichen- oder Bereichsfragen (da SINT 8 Bit umfasst, identisch zu BYTE).

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Composite-FB**: Der Baustein kapselt die Konvertierungslogik und stellt sie als Adapter-Konverter bereit.
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangs-Adapter sind unidirektional ausgelegt – Daten fließen nur in eine Richtung.
- **Kein interner Zustand**: Der FB verhält sich wie eine reine Funktion; jede Aktivierung erzeugt unabhängig vom vorherigen Zustand das korrekte Ergebnis.
- **Abhängigkeit**: Der Baustein importiert die IEC-61131-Funktion F_SINT_TO_BYTE, die in der Zielumgebung verfügbar sein muss.

Zustandsübersicht

Der FB besitzt **kein eigenes Zustandsdiagramm** – er ist ereignisgesteuert und führt bei jeder Aktivierung sofort eine einmalige Konvertierung aus. Das interne Element F_SINT_TO_BYTE ist ebenfalls zustandslos.

Anwendungsszenarien

- **Adapter-Kopplung**: Wenn ein Sensor oder Aktor einen AS-Adapter (SINT) bereitstellt, das nachfolgende System aber einen AB-Adapter (BYTE) erwartet, wird dieser FB als Zwischenstück eingesetzt.
- **Protokollanpassung**: In SPS- oder Automatisierungssystemen müssen häufig unterschiedliche Datentypen zwischen Adaptern konvertiert werden – hier speziell SINT ↔ BYTE.
- **Modularisierung**: Der FB erlaubt es, Konvertierungslogik in wiederverwendbaren Adapter-Bausteinen zu kapseln, anstatt sie in jedem Anwendungs-FB neu zu implementieren.

Vergleich mit ähnlichen Bausteinen

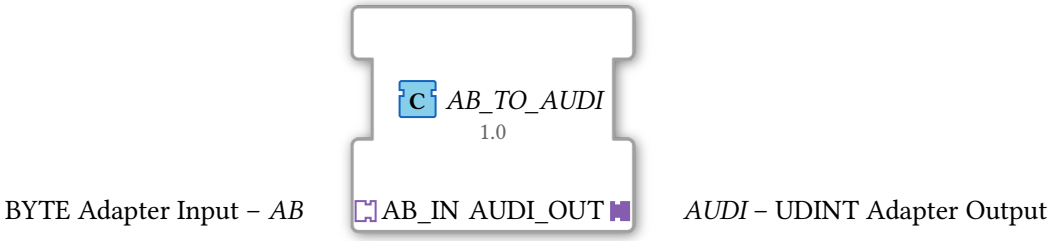
- **AS_TO_WORD / AS_TO_INT**: Konvertieren SINT in WORD oder INT; AS_TO_AB ist auf BYTE spezialisiert.
- **Direkte Verwendung von `F_SINT_TO_BYTE`**: Der Composite-FB bietet den Vorteil, dass die Konvertierung über Adapter-Ports erfolgt, was keine manuelle Verschaltung der Ereignis- und Datenflüsse benötigt.
- **Andere Adapter-Konverter**: Falls in der Bibliothek ähnliche Adapter-Konverter existieren (z.B. `AB_TO_AS`), ergänzt dieser FB die unidirektionale Konvertierung in die Gegenrichtung.

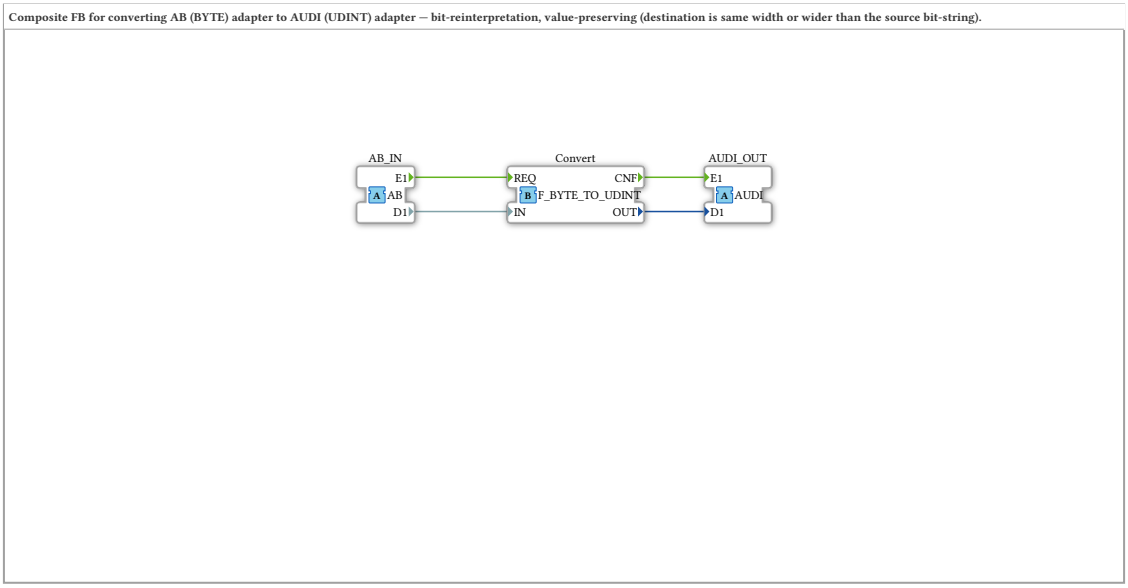
Fazit

Der **AS_TO_AB**-Funktionsblock ist ein einfacher, aber nützlicher Konverter für Adapter-basierte Architekturen. Er erlaubt die nahtlose Integration von SINT- und BYTE-kommunizierenden Komponenten, ohne dass der Anwender die zugrundeliegende Konvertierung selbst programmieren muss. Dank seiner Composite-Struktur bleibt die Logik klar und wartbar – ideal für den Einsatz in industriellen Steuerungssystemen nach IEC 61499.

AB_AUDI

AB_TO_AUDI





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AB_TO_AUDI** ist ein zusammengesetzter Baustein (Composite FB), der die Konvertierung eines AB-Adapters (Datentyp BYTE) in einen AUDI-Adapter (Datentyp UDINT) durchführt. Er dient als Schnittstelle zwischen unterschiedlichen Datenformaten in einer IEC 61499-basierten Automatisierungsumgebung.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Datenein-/ausgänge auf der obersten Ebene. Die Kommunikation erfolgt ausschließlich über zwei Adapter:

Ereignis-Eingänge

- Keine (die Ereignisse werden über den Socket-Adapter `AB_IN` bereitgestellt)

Ereignis-Ausgänge

- Keine (die Ereignisse werden über den Plug-Adapter `AUDI_OUT` ausgegeben)

Daten-Eingänge

- Keine (die Daten werden über den Socket-Adapter `AB_IN` bereitgestellt)

Daten-Ausgänge

- Keine (die Daten werden über den Plug-Adapter `AUDI_OUT` ausgegeben)

Adapter

- **`AB_IN`** (Socket) – Typ: `adapter::types::unidirectional::AB` Nimmt ein BYTE-Signal entgegen. Der Adapter stellt ein Ereignis `E1` und einen Datenwert `D1` (BYTE) bereit.
- **`AUDI_OUT`** (Plug) – Typ: `adapter::types::unidirectional::AUDI` Gibt das konvertierte UDINT-Signal aus. Der Adapter erwartet ein Ereignis `E1` und einen Datenwert `D1` (UDINT).

Funktionsweise

Der Baustein nutzt intern einen vordefinierten Konvertierungsbaustein `F_BYTE_TO_UDINT` aus der IEC 61131-Bibliothek. Der Ablauf:

1. Über den Socket `AB_IN` wird ein Ereignis `E1` empfangen, das den Start der Konvertierung auslöst.
2. Gleichzeitig wird der zugehörige BYTE-Datenwert aus `AB_IN.D1` an den Konverter `F_BYTE_TO_UDINT.IN` weitergeleitet.
3. Der Konverter berechnet den UDINT-Wert und gibt ihn an seinem Ausgang `OUT` aus.
4. Das Bestätigungsereignis `CNF` des Konverters wird an den Plug `AUDI_OUT.E1` gesendet.
5. Der konvertierte UDINT-Wert wird über `AUDI_OUT.D1` an den angeschlossenen Baustein weitergegeben.

Somit wird eine unidirektionale BYTE-zu-UDINT-Wandlung mit Ereignissteuerung realisiert.

Technische Besonderheiten

- **Bit-Reinterpretation, werterhaltend** (Zielbreite ist gleich groß oder größer als der Quell-Bit-String).
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional ausgelegt – Daten fließen nur in eine Richtung (BYTE → UDINT).
- **Synchroner Betrieb**: Die Konvertierung erfolgt streng ereignisgesteuert: Jedes eingehende Ereignis `E1` löst genau eine Umwandlung und ein ausgehendes Ereignis aus.
- **Standardkonvertierung**: Der interne Baustein `F_BYTE_TO_UDINT` ist ein etablierter IEC 61131-Baustein, sodass die Konvertierung normkonform und portabel ist.

Zustandsübersicht

Da es sich um einen rein zusammengesetzten Baustein ohne eigenen Zustandsautomaten handelt, ergibt sich die Zustandslogik aus dem Zusammenspiel der internen Verbindungen:

- **Bereit**: nach Initialisierung, warten auf Ereignis `AB_IN.E1`.
- **Konvertierung aktiv**: nach Empfang von `AB_IN.E1`, während der interne Konverter arbeitet.
- **Ausgabe bereit**: nach Bestätigung durch `Convert.CNF`, bis das Ereignis `AUDI_OUT.E1` und der zugehörige Datenwert übergeben werden.

Anwendungsszenarien

- **Protokollanpassung**: Wenn ein Sensor oder Aktor mit BYTE-Schnittstelle an einen Steuerungsteil angeschlossen werden soll, der UDINT erwartet (z.B. für Zählwerte, Frequenzen oder Zeitwerte).
- **Datenaufbereitung in Automatisierungsnetzwerken**: Einsatz in SPS- oder verteilten Systemen, um unterschiedliche Datenbreiten zu harmonisieren.
- **Adapterbrücke**: Als Zwischenglied zwischen vordefinierten Adaptertypen in einer IEC 61499-Applikation.

Vergleich mit ähnlichen Bausteinen

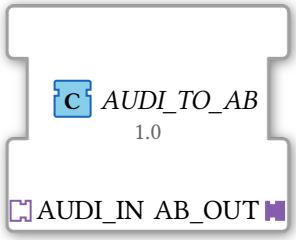
- **AB_TO_DINT**: Konvertiert BYTE nach DINT (vorzeichenbehaftet) – hier wird nach UDINT (vorzeichenlos) umgewandelt.
- **Direkte Konverter** wie `F_BYTE_TO_UDINT`: Diese sind einfache Funktionsbausteine ohne Adapter-Schnittstelle. `AB_TO_AUDI` kapselt diesen Baustein in eine Adapterstruktur und integriert die Ereignissteuerung.

Fazit

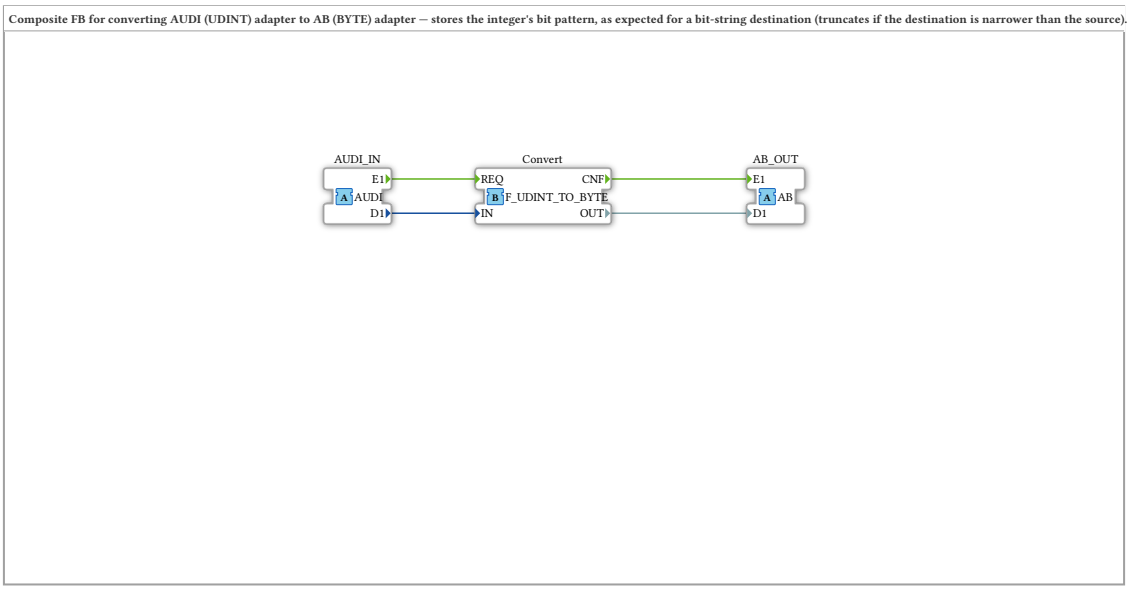
Der Funktionsblock `AB_TO_AUDI` stellt eine kompakte und wiederverwendbare Lösung zur Umwandlung von BYTE-Daten über einen AB-Adapter in UDINT-Daten über einen AUDI-Adapter dar. Durch die Verwendung standardisierter Konvertierung und Adapter wird eine hohe Kompatibilität und klare Schnittstellen in IEC 61499-Systemen erreicht.

AUDI_TO_AB

UDINT Adapter Input – *AUDI*



AB – BYTE Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AUDI_TO_AB** Funktionsblock ist ein zusammengesetzter Baustein (Composite FB), der eine UDINT-Schnittstelle (AUDI, 32-Bit unsigned Integer) in eine BYTE-Schnittstelle (AB, 8-Bit) umwandelt. Er kapselt die Konvertierungslogik und stellt zwei unidirektionale Adapter bereit: einen Socket für den AUDI-Eingang und einen Plug für den AB-Ausgang. Der Baustein wird typischerweise eingesetzt, um Datenkommunikation zwischen Systemen zu ermöglichen, die unterschiedliche Datentypen über Adapter verwenden.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung

AUDI_IN.E1	Ereignis (über Adapter)	Trigger für die Datenkonvertierung – ein eingehendes Ereignis startet die Umwandlung des aktuellen UDINT-Werts in einen BYTE-Wert.

Ereignis-Ausgänge

Name	Typ	Beschreibung
AB_OUT.E1	Ereignis (über Adapter)	Bestätigung der Konvertierung – nach erfolgreicher Umwandlung wird ein Ereignis ausgegeben, das dem nachgeschalteten System den neuen BYTE-Wert signalisiert.

Daten-Eingänge

Name	Typ	Beschreibung
AUDI_IN.D1	UDINT (32-Bit)	Der umzuwandelnde 32-Bit unsigned Integer-Wert, der über den AUDI-Adapter bereitgestellt wird.

Daten-Ausgänge

Name	Typ	Beschreibung
AB_OUT.D1	BYTE (8-Bit)	Der konvertierte 8-Bit Wert, der über den AB-Adapter ausgegeben wird.

Adapter

Name	Typ	Richtung	Beschreibung
AUDI_IN	adapter::types::unidirectional::AUDI	Socket (Eingang)	Unidirektionaler Adapter, der Ereignisse und Daten vom Typ UDINT bereitstellt.
AB_OUT	adapter::types::unidirectional::AB	Plug (Ausgang)	Unidirektionaler Adapter, der Ereignisse und Daten vom Typ BYTE ausgibt.

Funktionsweise

1. Ein externes System sendet ein Ereignis über den Adapter AUDI_IN (E1).
2. Dieses Ereignis triggert den internen Konvertierungsbaustein F_UDINT_TO_BYTE (aus der IEC 61131-Bibliothek) über dessen REQ-Eingang.
3. Gleichzeitig wird der aktuelle Datenwert von AUDI_IN.D1 an den IN-Eingang des Konvertierungsbausteins übergeben.
4. Der Konvertierungsbaustein wandelt den UDINT-Wert in einen BYTE-Wert um und legt das Ergebnis an seinem OUT-Ausgang an.
5. Nach Abschluss der Konvertierung sendet der F_UDINT_TO_BYTE ein Bestätigungsereignis (CNF), das an den AB_OUT-Adapter (E1) weitergeleitet wird.
6. Gleichzeitig wird der konvertierte BYTE-Wert an AB_OUT.D1 ausgegeben und steht dem empfangenden System zur Verfügung.

Die gesamte Verarbeitung erfolgt ereignisgesteuert und ohne Zwischenspeicherung – jedes eingehende Ereignis erzeugt genau ein ausgehendes Ereignis.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Composite-Struktur**: Der Baustein ist als Netzwerk von Unterbausteinen realisiert. Er nutzt den vordefinierten Konvertierungsbaustein F_UDINT_TO_BYTE aus der IEC 61131-Bibliothek, wodurch eine standardisierte und testbare Konvertierung gewährleistet wird.

- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional. Dies reduziert die Kopplung und ermöglicht eine einfache Integration in Datenflüsse, die nur eine Richtung benötigen.
- **Keine Zustandsspeicherung**: Der Baustein ist rein kombinatorisch (keine internen Zustände). Er reagiert auf jedes eingehende Ereignis sofort und gibt das Ergebnis ohne Verzögerung aus.
- **Kompatibilität**: Erfüllt den Standard IEC 61499-2 und kann in Umgebungen eingesetzt werden, die unidirektionale Adapter (AUDI/AB) unterstützen.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten. Er arbeitet ereignisgesteuert nach dem Prinzip „Eingangsereignis ⟶ Konvertierung ⟶ Ausgangsereignis“. Es gibt keine Warteschlangen oder Verzögerungen – die Verarbeitung ist atomar und erfolgt innerhalb eines einzigen Zyklus.

Anwendungsszenarien

- **Datenanpassung in Adapter-basierten Systemen**: Wenn eine Komponente oder ein Protokoll einen UDINT-Wert über einen AUDI-Adapter liefert, der nachgeschaltete Baustein aber einen BYTE-Wert über einen AB-Adapter erwartet, kann dieser Baustein zur nahtlosen Kopplung eingesetzt werden.
- **Übergang von 32-Bit auf 8-Bit Schnittstellen**: Beispielsweise in der Sensor-Aktor-Kommunikation, wo ein Sensor 32-Bit Rohdaten liefert, das Stellglied aber nur 8-Bit Kommandos verarbeitet.
- **Test- und Simulationsumgebungen**: Erlaubt das einfache Austauschen von Adaptern, ohne die gesamte Datenpfadlogik ändern zu müssen.

Vergleich mit ähnlichen Bausteinen

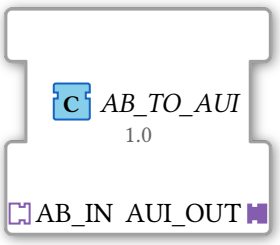
Baustein	Beschreibung	Unterschied zu AUDI_TO_AB
`F_UDINT_TO_BYTE`	Direkte Konvertierungsfunktion ohne Adapter	Erwartet Standard-Ereignis- und Daten-Schnittstellen; AUDI_TO_AB kapselt die Konvertierung in Adapter-Signaturen.
`UDINT_TO_BYTE` (Composite mit Standard-Ports)	Ähnliche Logik, aber mit SOCKET/PLUG-Schnittstelle	Verwendet möglicherweise andere Adapter-Typen oder bidirektionale Schnittstellen; AUDI_TO_AB ist auf unidirektionale Adapter spezialisiert.

Fazit

Der **AUDI_TO_AB** Funktionsblock bietet eine saubere, wiederverwendbare Lösung zur Typumwandlung zwischen UDINT- und BYTE-Werten unter Verwendung unidirektionaler Adapter. Seine kompakte Composite-Struktur, die auf einer standardisierten IEC-Konvertierungsfunktion aufbaut, macht ihn zuverlässig und einfach integrierbar. Durch die ereignisgesteuerte Architektur ohne Zustandsspeicherung eignet er sich besonders für zeitkritische Datenflüsse in der Automatisierungstechnik.

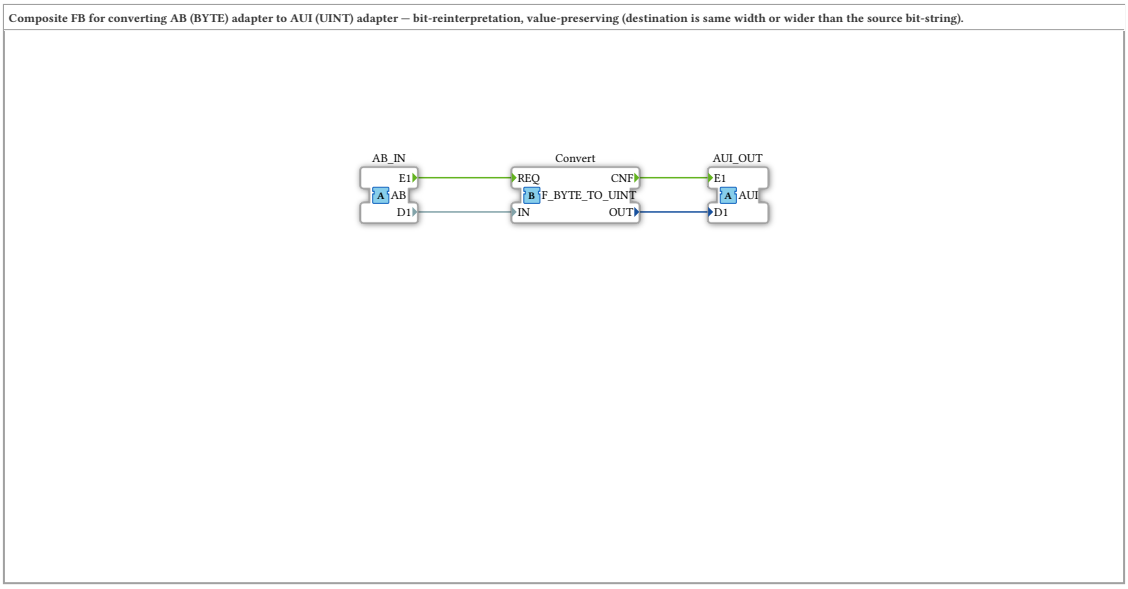
AB_AUI

AB_TO_AUI



BYTE Adapter Input – *AB*

AUI – UINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AB_TO_AUI** dient als Adapter-Konverter zwischen einem AB-Adapter (BYTE-basiert) und einem AUI-Adapter (UINT-basiert). Er realisiert eine unidirektionale Konvertierung von BYTE- auf UINT-Datentyp. Der Baustein ist als Composite-FB ausgeführt und nutzt intern die IEC 61131-3 Konvertierungsfunktion `F_BYTE_TO_UINT`. Er ist speziell für die Anbindung von Sensoren oder Aktoren vorgesehen, die ihre Daten im BYTE-Format liefern, jedoch in einer UINT-basierten Umgebung verarbeitet werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine direkten Ereignis-Eingänge auf der obersten Ebene. Die Ereignissteuerung erfolgt über die angeschlossenen Adapter.

Ereignis-Ausgänge

Der FB besitzt keine direkten Ereignis-Ausgänge auf der obersten Ebene. Die Ereignisweitergabe erfolgt über die angeschlossenen Adapter.

Daten-Eingänge

Der FB besitzt keine direkten Daten-Eingänge auf der obersten Ebene. Die Datenaufnahme erfolgt über den angeschlossenen AB-Adapter.

Daten-Ausgänge

Der FB besitzt keine direkten Daten-Ausgänge auf der obersten Ebene. Die Datenabgabe erfolgt über den angeschlossenen AUI-Adapter.

Adapter

Richtung	Name	Typ des Adapters	Beschreibung
Socket	AB_IN	`adapter::types::unidirectional::AB`	Eingangsadapter für BYTE-Daten (Quelladapter)
Plug	AUI_OUT	`adapter::types::unidirectional::AUI`	Ausgangsadapter für UINT-Daten (Zieladapter)

Hinweis: Die Adapter vom Typ `AB` und `AUI` besitzen jeweils einen Ereignis-Eingang (`E1`), einen Ereignis-Ausgang (`E1`), einen Daten-Eingang (`D1` vom Typ BYTE bzw. UINT) sowie einen Daten-Ausgang (`D1` vom Typ BYTE bzw. UINT). Die genauen Details sind den entsprechenden Adapterdefinitionen zu entnehmen.

Funktionsweise

Der Baustein arbeitet als reiner Konverter: Sobald am Socket `AB_IN` ein Ereignis anliegt (typischerweise über den Ereignis-Eingang des Adapters), wird dieses an den internen FB `F_BYTE_TO_UINT` weitergeleitet. Gleichzeitig wird der vom Adapter bereitgestellte BYTE-Wert (über den Daten-Eingang `D1` des AB-Adapters) an den Konvertierungs-FB übergeben. Der Konvertierungs-FB wandelt den BYTE-Wert in einen gleichwertigen UINT-Wert um. Nach erfolgreicher Konvertierung wird ein Ereignis über den Ereignis-Ausgang des internen FB ausgelöst, das an den Plug `AUI_OUT` weitergegeben wird. Der konvertierte UINT-Wert wird über den Daten-Ausgang des Konvertierungs-FB an den Daten-Eingang des AUI-Adapters übertragen.

Die folgende Abbildung zeigt die interne Verkabelung:

- Ereignisverbindung: `AB_IN.E1` ⟶ `Convert.REQ` ⟶ `Convert.CNF` ⟶ `AUI_OUT.E1`
- Datenverbindung: `AB_IN.D1` ⟶ `Convert.IN` ⟶ `Convert.OUT` ⟶ `AUI_OUT.D1`

Der gesamte Vorgang erfolgt streng sequenziell und ereignisgesteuert.

Technische Besonderheiten

- **Bit-Reinterpretation, werterhaltend** (Zielbreite ist gleich groß oder größer als der Quell-Bit-String).
- **Unidirektionaler Betrieb:** Der FB unterstützt nur die Datenflussrichtung von AB (BYTE) zu AUI (UINT). Eine Rückwandlung ist mit diesem Baustein nicht möglich.
- **Verwendung der IEC 61131-3 Konvertierung:** Die Konvertierung erfolgt über den standardisierten FB `F_BYTE_TO_UINT` aus der IEC 61131-Bibliothek. Dies gewährleistet eine zuverlässige und portable Wandlung.
- **Adapter-basierte Schnittstelle:** Der FB ist bewusst als Adapter-Konverter ausgelegt und nicht als eigenständiger FB mit direkten Ein-/Ausgängen. Dies ermöglicht eine einfache Integration in bestehende Adapter-basierte Kommunikationsstrukturen.
- **Keine Zustandsspeicherung:** Der FB ist stateless, d.h. er speichert keine Zwischenwerte. Jeder Konvertierungsvorgang ist unabhängig von vorhergehenden.

Zustandsübersicht

Der FB besitzt keine eigenen Zustandsautomaten. Das Verhalten wird durch den internen FB `F_BYTE_TO_UINT` (ein einfacher Funktionsbaustein ohne Zustände) und die Ereignisverkettung bestimmt. Der FB ist daher zu jedem Zeitpunkt bereit, eine Konvertierung durchzuführen, sobald ein Ereignis am Eingangsadapter eintrifft.

Anwendungsszenarien

- **Sensoranbindung:** Ein Sensor liefert Messwerte im BYTE-Format (z.B. über einen AB-Adapter), während das Steuerungssystem ausschließlich UINT-Werte verarbeitet. Der AB_TO_AUI-FB wandelt die Werte entsprechend um.
- **Protokollanpassung:** In heterogenen Feldbussystemen kann der FB als einfacher Datentypkonverter zwischen unterschiedlichen Adaptertypen eingesetzt werden.
- **Test und Simulation:** Während der Entwicklung können mit diesem FB BYTE-Daten aus einer Testumgebung in UINT-Daten für die Zielapplikation umgewandelt werden.

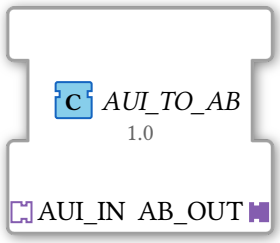
Vergleich mit ähnlichen Bausteinen

Es existieren weitere Adapter-Konverter wie z.B. `AB_TO_UDINT`, `BYTE_TO_WORD` oder direkte Konverter wie `F_BYTE_TO_SINT`. Der `AB_TO_AUI` fokussiert speziell auf die Adapter-Konvertierung unter Beibehaltung der Adapter-Struktur. Im Gegensatz zu reinen Funktionsbausteinen behält er die Adapter-Schnittstellen bei und ermöglicht so eine einfache Kapselung in Adapter-basierten Architekturen. Andere Bausteine wie `F_BYTE_TO_UINT` arbeiten auf Signalebene und benötigen separate Schnittstellen.

Fazit

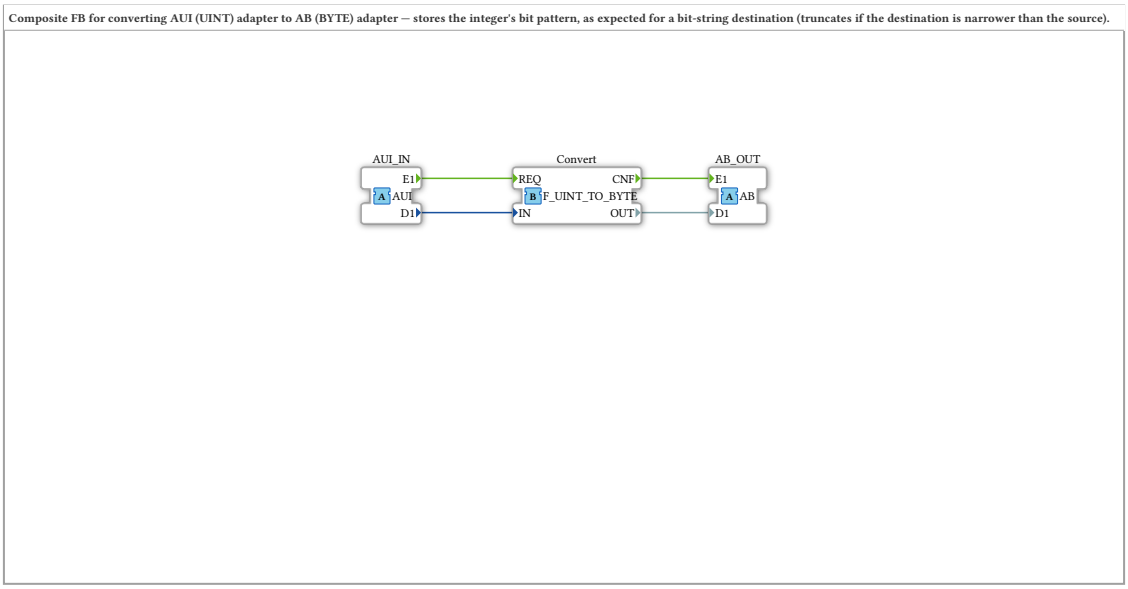
Der `AB_TO_AUI` ist ein kompakter und spezialisierter Composite-FB zur unidirektionalen Konvertierung von BYTE- auf UINT-Daten unter Verwendung standardisierter Adapter-Schnittstellen. Er erlaubt eine saubere Integration in adapterbasierte Systeme und reduziert den Aufwand für manuelle Datenkonvertierung. Durch den Einsatz des IEC 61131-3 Konvertierungsbausteins ist die Funktionalität zuverlässig und portabel. Der Baustein eignet sich besonders für Anwendungen, die auf eine lose Kopplung zwischen Datenquellen und -senken angewiesen sind.

AUI_TO_AB



UINT Adapter Input – *AUI*

AB – BYTE Adapter Output



Kein Bild verfügbar

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUI_TO_AB** ist ein Composite-Baustein, der eine AUI-Schnittstelle (UINT) in eine AB-Schnittstelle (BYTE) umwandelt. Er kapselt die Konvertierung mittels des internen IEC 61131-Bausteins `F_UINT_TO_BYTE` und stellt über Adapter einen einheitlichen unidirektionalen Datenfluss bereit. Entwickelt wurde dieser Baustein zur Verwendung in Automatisierungssystemen, die unterschiedliche Datentypen zwischen Adaptern übertragen müssen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Beschreibung	Datentyp
`AUI_IN.E1`	Ereignis über den AUI-Adapter (Socket), das die Konvertierung auslöst.	Ereignis

Ereignis-Ausgänge

Name	Beschreibung	Datentyp
`AB_OUT.E1`	Ereignis über den AB-Adapter (Plug), das nach erfolgreicher Konvertierung gesendet wird.	Ereignis

Daten-Eingänge

Name	Beschreibung	Datentyp
`AUI_IN.D1`	UINT-Wert, der über den AUI-Adapter empfangen und konvertiert wird.	UINT

Daten-Ausgänge

Name	Beschreibung	Datentyp
AB_OUT.D1	Konvertierter BYTE-Wert, der über den AB-Adapter ausgegeben wird.	BYTE

Adapter

Typ	Name	Richtung	Kommentar
adapter::types::unidirectional::AUI	AUI_IN	Socket	UINT-Adapter-Eingang
adapter::types::unidirectional::AB	AB_OUT	Plug	BYTE-Adapter-Ausgang

Funktionsweise

Der Baustein arbeitet als reiner Signalwandler:

1. Ein eingehendes Ereignis am **AUI_IN.E1** wird an den internen Funktionsblock Convert (Typ F_UINT_TO_BYTE) weitergeleitet.
2. Gleichzeitig wird der Datenwert **AUI_IN.D1** an den Eingang Convert.IN übergeben.
3. Der Konverter wandelt den UINT-Wert in einen BYTE-Wert um (Datenkürzung auf niederwertiges Byte).
4. Nach Abschluss wird das Ereignis **AB_OUT.E1** ausgelöst und der konvertierte Wert über **AB_OUT.D1** bereitgestellt.

Intern wird die Mapping-Logik durch einen einzigen IEC 61131-Baustein realisiert, daher ist die Latenz minimal und die Funktion deterministisch.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Composite-Struktur**: Der FB ist als Composite implementiert, d.h. er enthält ein internes Netzwerk mit einem einzigen Funktionsblock. Dies erleichtert die Wiederverwendung und Kapselung.
- **Adapter-Konzept**: Die Ein- und Ausgabe erfolgt ausschliesslich über typisierte Adapter (AUI und AB). Dies ermöglicht eine lose Kopplung und den einfachen Austausch von Kommunikationsprotokollen in der Applikation.
- **Lizenz**: Der Baustein steht unter der Eclipse Public License 2.0 (EPL-2.0). Der Quellcode ist offen und kann frei angepasst werden.
- **Abhängigkeit**: Er benötigt den IEC 61131-Baustein F_UINT_TO_BYTE, der im Paket iec61131::conversion enthalten sein muss.

Zustandsübersicht

Da der Baustein keine interne Zustandsmaschine besitzt (Composite ohne eigenes ECC), ist der Zustand rein durch den internen Funktionsblock F_UINT_TO_BYTE bestimmt. Dieser arbeitet ereignisgesteuert und besitzt keinen persistenten Zustand. Nach jedem Ereignis kehrt der Baustein in den Ruhezustand zurück.

Anwendungsszenarien

- **Datenanpassung**: Wenn ein Sensor Daten im UINT-Format über einen AUI-Adapter liefert, das nachgeschaltete System jedoch einen BYTE-Wert über einen AB-Adapter erwartet.
- **Systemintegration**: Als Vermittler zwischen unterschiedlichen Adapter-Typen in heterogenen Automatisierungsnetzwerken.
- **Protokollanpassung**: In Szenarien, bei denen bestehende Bibliotheken nur bestimmte Adapter-Typen unterstützen und eine Typumwandlung erforderlich ist.

Vergleich mit ähnlichen Bausteinen

- **Direkter F_UINT_TO_BYTE**: Dieser IEC-Baustein wird auch intern verwendet, arbeitet aber ohne Adapter – er benötigt direkte Daten- und Ereignisports. AUI_TO_AB kapselt diese Schnittstelle in Adapterform.
- **Andere Adapter-Konverter**: Es existieren ähnliche Bausteine wie AB_TO_AUI oder ADI_TO_AB für andere Typkombinationen. Sie folgen dem gleichen Pattern und sind austauschbar.

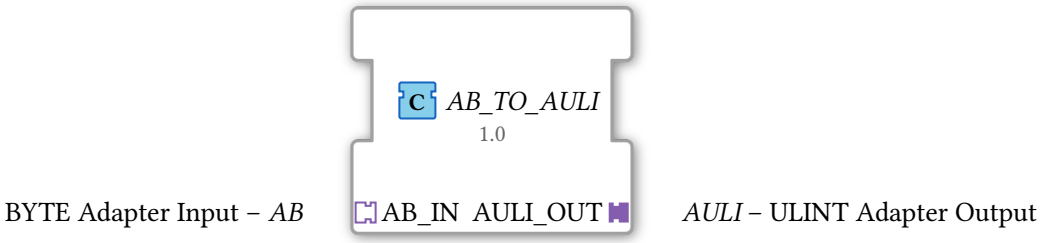
Der entscheidende Vorteil von `AUI_TO_AB` ist die vollständige Integration in adapterbasierte Architekturen – ohne dass die Applikation die interne Konvertierungslogik kennen muss.

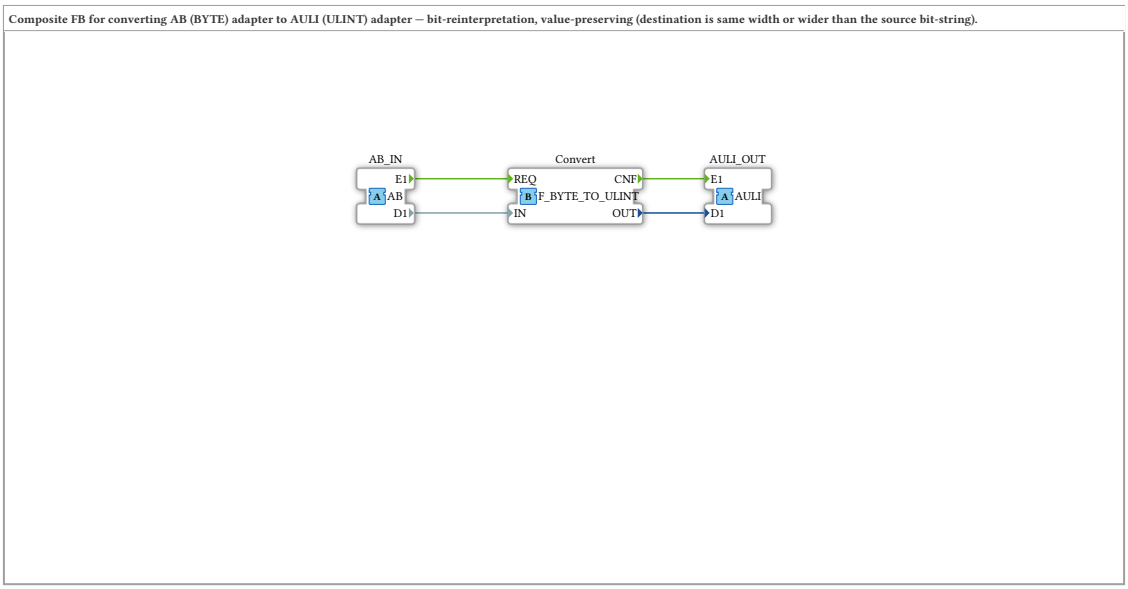
Fazit

Der Funktionsblock **AUI_TO_AB** ist ein einfacher, aber nützlicher Composite-Baustein zur Typumwandlung zwischen zwei unidirektionalen Adaptern. Seine klare Struktur und die Verwendung eines standardisierten Konverters machen ihn zuverlässig und leicht verständlich. Für alle Anwendungen, die eine saubere Adapter-Trennung erfordern, stellt er eine elegante Lösung dar.

AB_AULI

AB_TO_AULI





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AB_TO_AULI** ist ein zusammengesetzter Baustein (Composite FB), der einen unidirektionalen Adapter vom Typ `AB` (BYTE) in einen unidirektionalen Adapter vom Typ `AULI` (ULINT) umwandelt. Er dient als Konverter zwischen unterschiedlichen Datentypen in der Adapterkommunikation nach IEC 61499-2. Der FB nutzt intern den standardisierten Konvertierungsbaustein `F_BYTE_TO_ULINT` aus der IEC 61131-Bibliothek.

Schnittstellenstruktur

Ereignis-Eingänge

- **AB_IN.E1** – Ereigniseingang des AB-Adapter-Sockets. Wird durch das eingehende Adapterprotokoll ausgelöst und startet die Konvertierung.

Ereignis-Ausgänge

- **AULI_OUT.E1** – Ereignisausgang des AULI-Adapter-Plugs. Signalisiert den Abschluss der Konvertierung an den nachgeschalteten Adapterpfad.

Daten-Eingänge

- **AB_IN.D1** (Datentyp: BYTE) – Empfängt den zu konvertierenden Wert über den AB-Adapter.

Daten-Ausgänge

- **AULI_OUT.D1** (Datentyp: ULINT) – Stellt den konvertierten ULINT-Wert über den AULI-Adapter bereit.

Adapter

- **Socket „AB_IN“** – Typ: `adapter::types::unidirectional::AB` Unidirektionaler Eingangsadapter für den Datentyp BYTE.
- **Plug „AULI_OUT“** – Typ: `adapter::types::unidirectional::AULI` Unidirektionaler Ausgangsadapter für den Datentyp ULINT.

Funktionsweise

Der FB wartet auf ein Ereignis am Eingangsadapter `AB_IN`. Sobald das Ereignis eintrifft, wird der an `AB_IN.D1` anliegende BYTE-Wert an den internen Konvertierungsbaustein `F_BYTE_TO_ULINT` weitergeleitet und dort in einen ULINT-Wert umgewandelt. Nach Abschluss der Konvertierung wird der resultierende Wert an `AULI_OUT.D1` ausgegeben und das Abschlussereignis an `AULI_OUT.E1` gesendet. Die gesamte Verarbeitung erfolgt ereignisgesteuert und ohne Zustandsautomaten.

Technische Besonderheiten

- **Bit-Reinterpretation, werterhaltend** (Zielbreite ist gleich groß oder größer als der Quell-Bit-String).
- Der Baustein implementiert keine eigenen Zustände oder Algorithmen – er ist eine reine Verschaltung eines Standard-Konvertierungs-FBs.
- Die Konvertierung erfolgt nach den Regeln der IEC 61131-3: Der BYTE-Wert wird als 8-Bit-Darstellung eines ULINT interpretiert und in eine vorzeichenlose 64-Bit-Ganzzahl umgewandelt.
- Die Adapter basieren auf einem unidirektionalen Kommunikationsmuster, d. h. Daten und Ereignisse fließen nur in eine Richtung (vom Socket zum Plug).
- Der FB ist nicht zustandsbehaftet; jede Anforderung wird unabhängig verarbeitet.

Zustandsübersicht

Der Baustein besitzt keinen eigenen Zustandsautomaten. Die Abarbeitung folgt einem einfachen ereignisgesteuerten Ablauf:

1. Warten auf Eingangsereignis (`AB_IN.E1`).
2. Konvertierung des Datenwerts.
3. Ausgabe des Ergebnisses und Auslösen des Ausgangsereignisses (`AULI_OUT.E1`).

Anwendungsszenarien

- **Anbindung von BYTE-basierten Sensoren** an eine Steuerung, die ULINT als einheitlichen Datentyp für längere Messwerte verwendet.
- **Datenformatwandlung** inAdapterketten, wenn ein Protokoll auf BYTE-Ebene kommuniziert, aber die Anwendung einen ULINT-Wert erwartet.
- **Test- und Debugging-Aufbauten**, bei denen ein BYTE-Testadapter in eine ULINT-Schnittstelle umgesetzt werden soll.

Vergleich mit ähnlichen Bausteinen

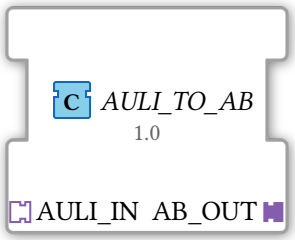
- **Direkte Konvertierung ohne Adapter** (z. B. `F_BYTE_TO_ULINT`): Diese Bausteine arbeiten auf Datenebene, nicht auf Adapterebene. Der `AB_TO_AULI` kapselt die Konvertierung inklusive der Adapterschnittstellen.
- **Adapter-Konverter für andere Datentypen** (z. B. `AB_TO_AUDINT`): Diese sind strukturell ähnlich, unterscheiden sich jedoch im Ziel-Datentyp und gegebenenfalls in der verwendeten Konvertierungsfunktion.
- **Bidirektionale Adapter-Konverter**: Falls erforderlich, müssten zwei unidirektionale Konverter (z. B. `AB_TO_AULI` und `AULI_TO_AB`) kombiniert werden.

Fazit

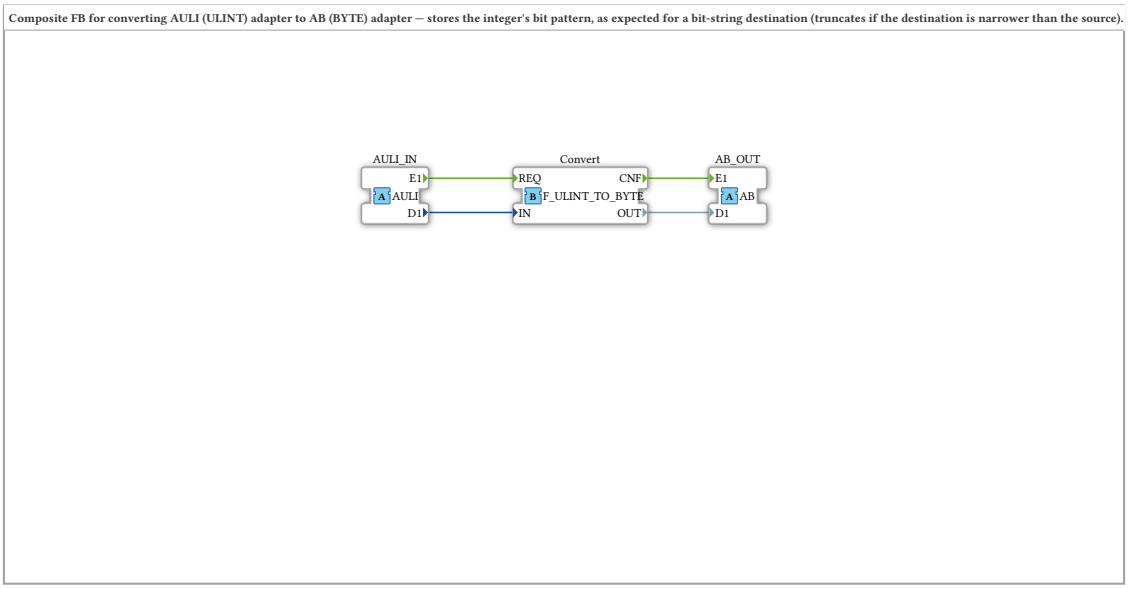
Der FB `AB_TO_AULI` bietet eine einfache, standardkonforme Möglichkeit, einen BYTE-Adapter in einen ULINT-Adapter umzuwandeln. Durch die Verwendung eines etablierten IEC 61131-Konvertierungsbausteins ist die Konvertierung zuverlässig und leicht nachvollziehbar. Der Baustein ist ideal für Adapterbasierte Kommunikationsketten, die einen Datentypwechsel auf Adapterebene erfordern.

AULI_TO_AB

ULINT Adapter Input – *AULI*



AB – BYTE Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `AULI_TO_AB` ist ein Composite-Baustein, der einen AULI-Adapter (ULINT-Datentyp) in einen AB-Adapter (BYTE-Datentyp) umwandelt. Er ermöglicht die nahtlose Verbindung von Komponenten, die unterschiedliche Adapter-Schnittstellen verwenden, indem die eingehenden Daten konvertiert und als kompatibler Ausgang bereitgestellt werden.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge/-ausgänge. Die gesamte Ein- und Ausgabe erfolgt über Adapter.

Ereignis-Eingänge

Keine (Ereignisse werden über den AULI-Adapter empfangen).

Ereignis-Ausgänge

Keine (Ereignisse werden über den AB-Adapter gesendet).

Daten-Eingänge

Keine (Daten werden über den AULI-Adapter empfangen).

Daten-Ausgänge

Keine (Daten werden über den AB-Adapter gesendet).

Adapter

Name	Typ	Richtung	Kommentar
AULI_IN	adapter::types::unidirectional::ASocket	Socket (Eingangs-Adapter)	ULINT-Adapter, der die zu konvertierenden Werte und das Auslöse-Ereignis liefert
AB_OUT	adapter::types::unidirectional::APlug	Plug (Ausgangs-Adapter)	BYTE-Adapter, der die konvertierten Werte und das Bestätigungs-Ereignis bereitstellt

Funktionsweise

Der Baustein nutzt intern den IEC-61131-Baustein `F_ULINT_TO_BYTE` für die Konvertierung von ULINT nach BYTE. Die Abläufe sind:

1. **Eingang**: Am Socket `AULI_IN` wird ein Ereignis (E1) empfangen, das die Konvertierung anstößt. Gleichzeitig liegt der Datenwert (D1) als ULINT an.
2. **Verarbeitung**: Das Ereignis und die Daten werden an den internen Baustein `F_ULINT_TO_BYTE` weitergeleitet (Ereignis `REQ`, Daten `IN`). Dieser führt die Typumwandlung durch.
3. **Ausgang**: Nach erfolgreicher Konvertierung signalisiert der interne Baustein ein Ereignis (`CNF`) und liefert das Ergebnis (`OUT`) als BYTE. Beide werden auf den Plug `AB_OUT` übertragen (Ereignis E1, Daten D1).

Der gesamte Ablauf erfolgt synchron innerhalb eines Ausführungszyklus des Composite-Bausteins.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Composite-Baustein**: Die Logik wird vollständig durch das interne Netzwerk aus vordefinierten Bausteinen realisiert – es existiert kein eigenes ECC (Execution Control Chart).
- **Lizenz und Copyright**: Der Baustein ist unter der Eclipse Public License 2.0 (EPL-2.0) lizenziert und unterliegt dem Urheberrecht eines Drittanbieters.
- **Abhängigkeiten**: Er setzt den IEC-61131-Baustein `F_ULINT_TO_BYTE` aus der Bibliothek `iec61131::conversion` voraus.
- **Paketstruktur**: Der Baustein gehört zum Paket `adapter::conversion::unidirectional`.

Zustandsübersicht

Da der Baustein kein eigenes ECC besitzt, existieren keine expliziten Zustände. Der interne Ablauf ist strikt durch die Ereigniskette `AULI_IN.E1 → Convert.REQ → Convert.CNF → AB_OUT.E1` festgelegt.

Anwendungsszenarien

- Einbindung eines ULINT-liefernden Sensors (z. B. hochauflösender Entfernungsmesser) in ein Steuerungssystem, das auf BYTE-Adapter-Schnittstellen basiert.
- Nachrüstung bestehender Anlagen, bei denen Komponenten mit unterschiedlichen Adaptertypen (ULINT vs. BYTE) zusammengeführt werden müssen.
- Prototypenentwicklung, bei der vorübergehend ULINT-Werte über einen BYTE-Kanal verarbeitet werden (z. B. Simulation oder Test).

Vergleich mit ähnlichen Bausteinen

Es existieren analoge Konvertierungsbausteine für andere Datentypen (z. B. `AULI_TO_xx` oder `xx_TO_AB`). Der `AULI_TO_AB` ist speziell für die Umwandlung von ULINT in BYTE optimiert und nutzt die zugehörigen Adapter-Typen. Im Gegensatz zu generischen Funktionsbausteinen vermeidet er die manuelle Verkabelung von Einzel-Ereignissen und -Daten und bietet eine gekapselte, adapterbasierte Schnittstelle.

Fazit

Der `AULI_TO_AB` ist ein kompakter, zuverlässiger Composite-Baustein zur Konvertierung zwischen zwei gängigen Adapter-Typen in der 4diac-IDE. Er reduziert den Verkabelungsaufwand und vereinfacht die

Integration heterogener Komponenten. Durch die Verwendung des standardisierten IEC-61131-Konvertierungsbausteins bleibt die Umwandlung typsicher und effizient.

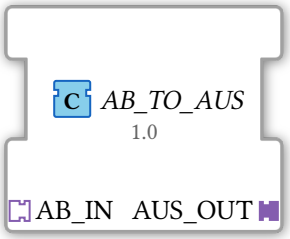
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

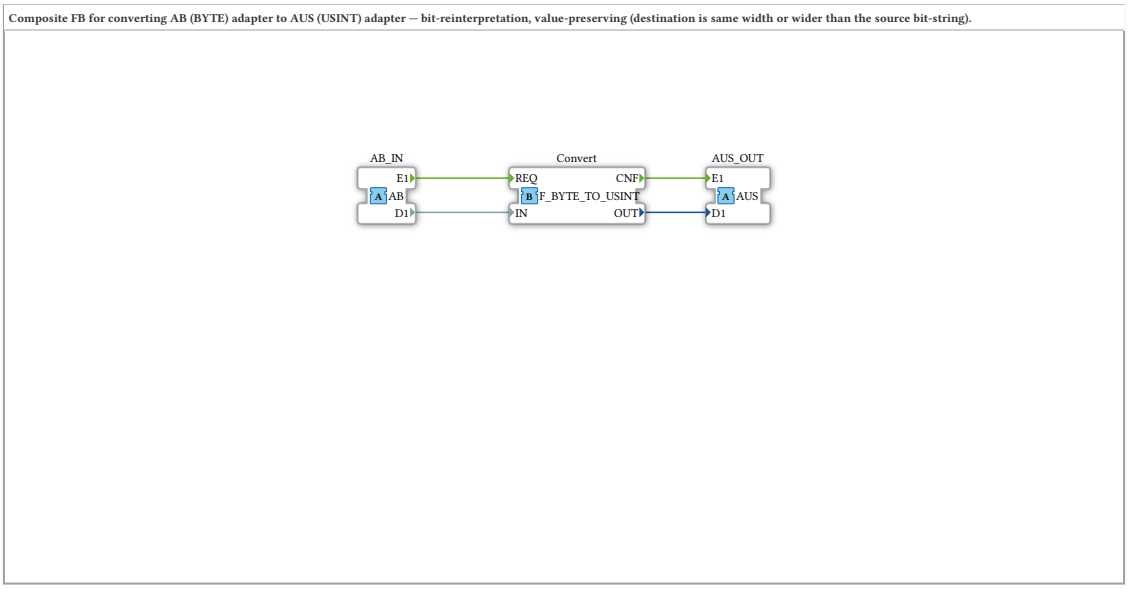
AB_AUS

AB_TO_AUS

BYTE Adapter Input – *AB*



AUS – USINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AB_TO_AUS** ist ein Composite-Baustein, der eine unidirektionale Konvertierung von Daten vom **AB-Adapter** (BYTE) zum **AUS-Adapter** (USINT) vornimmt. Er kapselt die notwendige Typumwandlung und Ereignisweitergabe und ermöglicht so die nahtlose Integration von Komponenten, die auf unterschiedlichen Datentypen basieren.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt auf oberster Ebene keine separaten Ereignis-Eingänge. Das auslösende Ereignis wird über den Socket-Adapter **AB_IN** bereitgestellt (dort als `E1` verfügbar).

Ereignis-Ausgänge

Es gibt keine direkten Ereignis-Ausgänge auf der Ebene des Composite-FBs. Das resultierende Ereignis wird über den Plug-Adapter **AUS_OUT** (dort als `E1`) ausgegeben.

Daten-Eingänge

Auf oberster Ebene sind keine Daten-Eingänge definiert. Die Eingangsdaten (BYTE) werden über den Socket-Adapter **AB_IN** (dort als `D1`) entgegengenommen.

Daten-Ausgänge

Es existieren keine direkten Daten-Ausgänge. Die konvertierten Daten (USINT) werden über den Plug-Adapter **AUS_OUT** (dort als `D1`) bereitgestellt.

Adapter

Typ	Name	Richtung	Beschreibung
Socket	**AB_IN**	Eingang	Unidirektionaler Adapter vom Typ `adapter::types::unidirectional::AB`. Liefert ein BYTE (`D1`) sowie ein Ereignis (`E1`).
Plug	**AUS_OUT**	Ausgang	Unidirektionaler Adapter vom Typ `adapter::types::unidirectional::AUS`. Stellt ein USINT (`D1`) sowie ein Ereignis (`E1`) zur Verfügung.

Funktionsweise

1. Ein Ereignis an `AB_IN.E1` löst die Verarbeitung aus.
2. Der Wert von `AB_IN.D1` (BYTE) wird an den internen Konverter **F_BYTE_TO_USINT** übergeben.
3. Der Konverter wandelt das BYTE in einen USINT-Wert um.
4. Das Ergebnis wird auf `AUS_OUT.D1` (USINT) ausgegeben.
5. Gleichzeitig wird ein Ereignis auf `AUS_OUT.E1` erzeugt, um die nachfolgende Verarbeitung zu signalisieren.

Die gesamte Konvertierung erfolgt ereignisgesteuert und in einem Schritt.

Technische Besonderheiten

- **Bit-Reinterpretation, werterhaltend** (Zielbreite ist gleich groß oder größer als der Quell-Bit-String).
- **Unidirektionale Adapterkopplung**: Der FB verbindet zwei unidirektionale Adapter (AB → AUS) und stellt sicher, dass die Datenflussrichtung klar definiert ist.
- **Wiederverwendbare IEC-Konvertierung**: Intern wird der standardisierte Baustein `iec61131::conversion::F_BYTE_TO_USINT` verwendet.
- **Composite-Architektur**: Die Konvertierungslogik ist gekapselt und kann leicht in andere Projekte eingebunden oder durch alternative Konverter ersetzt werden.
- **Kein interner Zustand**: Der FB besitzt keinen eigenen Zustandsautomaten; er leitet Ereignisse und Daten direkt weiter.

Zustandsübersicht

Der **AB_TO_AUS**-Baustein besitzt keinen eigenen Zustandsautomaten. Die Funktionsweise entspricht einer reinen, ereignisgesteuerten Datenumsetzung. Der interne Konverter `F_BYTE_TO_USINT` arbeitet ebenfalls ohne Zustandspeicherung.

Anwendungsszenarien

- **Systemintegration**: Verbindung eines Geräts, das Messwerte als BYTE über den AB-Adapter bereitstellt, mit einer Steuerung, die USINT-Daten über den AUS-Adapter erwartet.
- **Protokollanpassung**: Umwandlung von BYTE-kodierten Befehlen in USINT-kodierte Werte für die Weiterverarbeitung in Bibliotheken, die ausschließlich USINT unterstützen.
- **Modulare Wiederverwendung**: Kapselung der Konvertierung als eigenständigen Baustein, um die Übersichtlichkeit in großen Automatisierungsprojekten zu erhöhen.

Vergleich mit ähnlichen Bausteinen

Baustein	Beschreibung
F_BYTE_TO_USINT	Einfacher IEC-Konverter ohne Adapteranbindung. Muss manuell mit Ereignis- und Datenverbindungen verdrahtet werden.

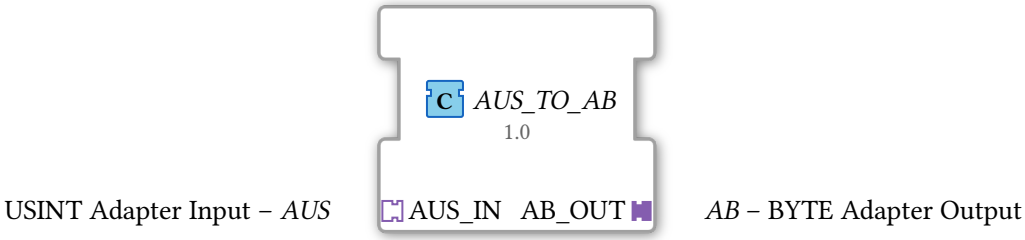
AB_TO_AUS	Composite-FB, der die gesamte Adapterkonvertierung inklusive Ereignisweitergabe kapselt und dadurch eine saubere, abstrahierte Schnittstelle bietet.
AUS_TO_AB (hypothetisch)	Reverse-Conversion von USINT zu BYTE – spiegelbildliche Funktionalität für die Gegenrichtung.

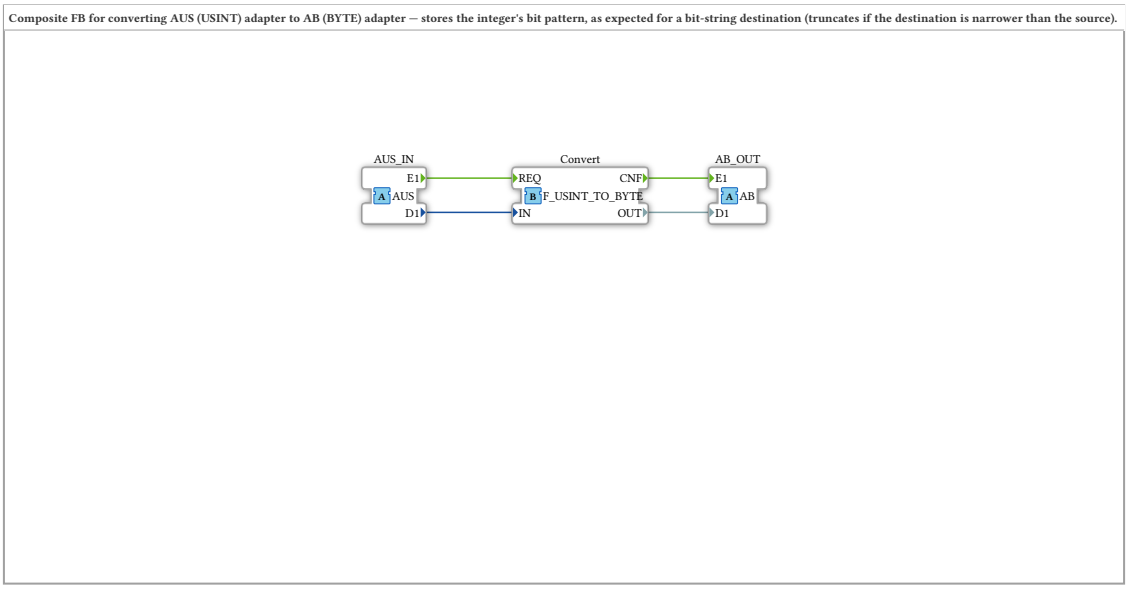
Der **AB_TO_AUS** hebt sich durch seine einfache Handhabung und die direkte Adapterkompatibilität hervor.

Fazit

Der **AB_TO_AUS**-Funktionsblock bietet eine elegante und standardisierte Möglichkeit, BYTE-Daten aus einem AB-Adapter in USINT-Daten für einen AUS-Adapter umzuwandeln. Durch die Kapselung der Konvertierungslogik in einem Composite-Baustein wird die Wiederverwendbarkeit gefördert und die Systemintegration vereinfacht. Er eignet sich besonders für modulare Automatisierungslösungen auf Basis von IEC 61499.

AUS_TO_AB





• – ○ • – ○ • – ○ •

Einleitung

Der **AUS_TO_AB** ist ein Composite-Funktionsblock zur Konvertierung eines unidirektionalen **AUS-Adapters** (USINT-Datentyp) in einen **AB-Adapter** (BYTE-Datentyp). Er dient als Brücke zwischen unterschiedlichen Adapter-Schnittstellen in Automatisierungssystemen und ermöglicht die nahtlose Integration von Komponenten, die auf verschiedenen Datentypen basieren. Der Baustein nutzt intern den Standardkonvertierungsblock *F_USINT_TO_BYTE* aus der IEC 61131-Bibliothek.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine eigenen Ereignis-Eingangsports. Ereignisse werden über den **AUS_IN**-Adapter (Socket) empfangen:

- **AUS_IN.E1** – Auslöser für die Datenkonvertierung und Weitergabe an den Ausgang.

Ereignis-Ausgänge

Ereignisse werden über den **AB_OUT**-Adapter (Plug) ausgegeben:

- **AB_OUT.E1** – Bestätigung, dass der konvertierte Wert am Ausgang anliegt.

Daten-Eingänge

Daten werden über den **AUS_IN**-Adapter bereitgestellt:

- **AUS_IN.D1** (USINT) – Der zu konvertierende 8-Bit-vorzeichenlose Integerwert.

Daten-Ausgänge

Daten werden über den **AB_OUT**-Adapter weitergegeben:

- **AB_OUT.D1** (BYTE) – Der konvertierte Byte-Wert.

Adapter

Name	Typ	Richtung	Beschreibung
AUS_IN	`adapter::types::unidirectional::AUS`	Socket	Eingangsadapter für USINT-Daten und Ereignisse.
AB_OUT	`adapter::types::unidirectional::AB`	Plug	Ausgangsadapter für BYTE-Daten und Ereignisse.

Funktionsweise

Der **AUS_TO_AB** ist als reines Composite-FB ohne eigene Logik realisiert. Er verbindet den AUS_IN- und AB_OUT-Adapter über den internen Baustein **F_USINT_TO_BYTE**:

1. Ein eingehendes Ereignis auf **AUS_IN.E1** triggert den Konvertierungsbaustein **Convert** (`F_USINT_TO_BYTE`) über dessen **REQ**-Eingang.
2. Gleichzeitig wird der Datenwert von **AUS_IN.D1** an den **Convert.IN**-Eingang übergeben.
3. Nach erfolgter Konvertierung (USINT ⟶ BYTE) bestätigt **Convert** mit einem Ereignis auf **CNF**.
4. Dieses Ereignis wird direkt an **AB_OUT.E1** weitergeleitet.
5. Der konvertierte Byte-Wert von **Convert.OUT** wird an **AB_OUT.D1** übergeben.

Damit ist eine synchronisierte, ereignisgesteuerte Datenweitergabe gewährleistet.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Composite-FB**: Enthält keine eigene Zustandsmaschine; die gesamte Logik wird durch den internen FB `F_USINT_TO_BYTE` abgebildet.
- **Unidirektionale Adapter**: Sowohl der eingehende als auch der ausgehende Adapter sind unidirektional – die Datenflüsse erfolgen ausschließlich in eine Richtung.
- **Standardkonformität**: Der Baustein basiert auf IEC 61499-2 und nutzt die IEC 61131-Konvertierungsfunktion `F_USINT_TO_BYTE`. Dadurch ist er in kompatiblen Laufzeitumgebungen (z. B. 4diac-FORTE) sofort einsetzbar.
- **Lizenz**: Veröffentlicht unter der Eclipse Public License 2.0 (EPL-2.0). Copyright (c) 2026 HR Agrartechnik GmbH.

Zustandsübersicht

Da es sich um einen Composite-FB ohne eigene ECC (Execution Control Chart) handelt, besteht kein expliziter Zustandsautomat. Das Verhalten ist rein ereignisgesteuert und wird durch die interne Verschaltung der Einzelfunktionsblöcke bestimmt. Nach dem Start liegt jederzeit eine Bereitschaft vor – ein eingehendes Ereignis wird direkt verarbeitet.

Anwendungsszenarien

- **Systemintegration** in der Landtechnik oder anderen Automatisierungsumgebungen, in denen ein **AUS-Adapter** (USINT) an einen **AB-Adapter** (BYTE) angebunden werden muss.
- **Nachrüstung älterer Komponenten**, die einen BYTE-Ausgang liefern, mit modernen USINT-basierten Sensoren oder Aktoren.
- **Prototyping und Testaufbauten**, bei denen schnell zwischen verschiedenen Adaptertypen gewechselt werden soll.
- **Datenaufbereitung** in Kommunikationsknoten, die unterschiedliche Adapterprotokolle vereinheitlichen.

Vergleich mit ähnlichen Bausteinen

- **F_USINT_TO_BYTE** – direkter Funktionsblock für die reine Datentypkonvertierung, jedoch ohne Adapter-Einbindung. **AUS_TO_AB** fügt die Adapter- und Ereignislogik hinzu.
- **BYTE_TO_USINT** – inverse Umwandlung (BYTE → USINT), falls die umgekehrte Adapterrichtung benötigt wird (nicht im Lieferumfang dieses Bausteins).
- **Generische Adapter-Konverter** – komplexere Blöcke, die mehrere Datentypen gleichzeitig umsetzen. **AUS_TO_AB** ist auf den spezifischen Anwendungsfall optimiert und daher schlanker.

Fazit

Der **AUS_TO_AB** ist ein nützlicher und leichter Baustein zur Umwandlung eines **AUS-Adapters (USINT)** in einen **AB-Adapter (BYTE)**. Seine Composite-Struktur minimiert Implementierungsaufwand, während die Verwendung eines standardisierten Konvertierungsblocks die Zuverlässigkeit sichert. Der Baustein eignet sich besonders für die schnelle Integration heterogener Adapter-Schnittstellen in IEC 61499-basierten Steuerungssystemen.

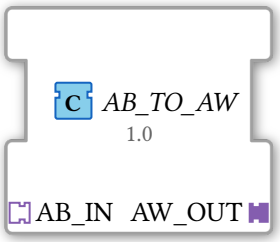
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

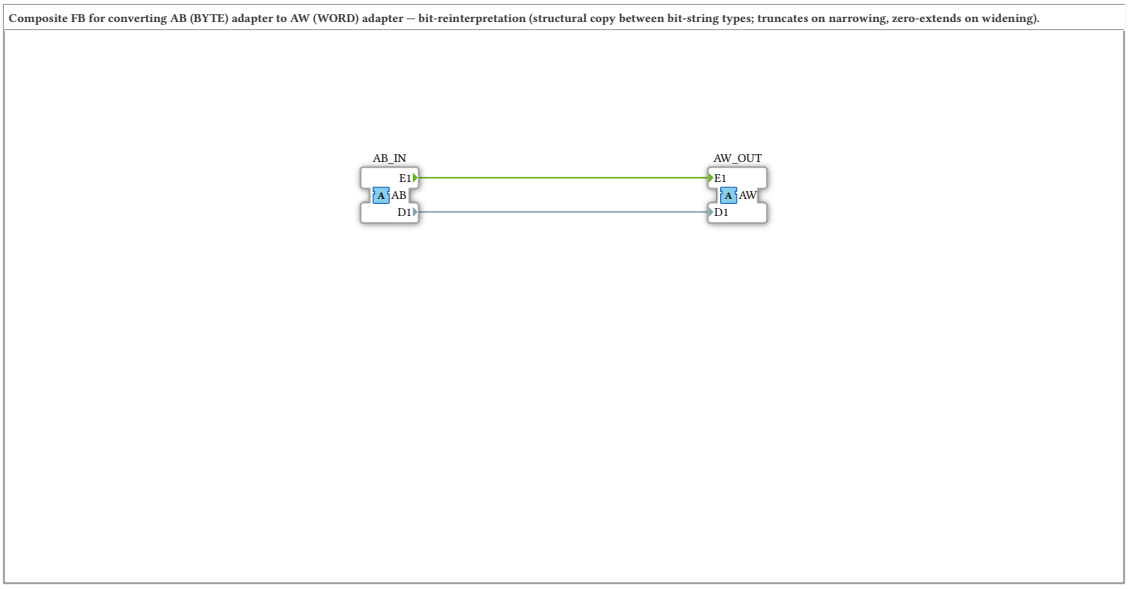
AB_AW

AB_TO_AW

BYTE Adapter Input – *AB*



AW – WORD Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **AB_TO_AW** ist ein Composite-Baustein, der eine unidirektionale Konvertierung von einem AB-Adapter (BYTE-basiert) zu einem AW-Adapter (WORD-basiert) vornimmt. Er dient als Schnittstellenumsetzer, wenn ein Systemteil Daten im BYTE-Format liefert, der nachgeschaltete Teil jedoch ein WORD-Format erwartet.

Schnittstellenstruktur

Der Baustein besitzt keine klassischen Ereignis- oder Dateneingänge/-ausgänge auf der obersten Ebene, sondern verwendet ausschließlich Adapter-Schnittstellen.

Ereignis-Eingänge

- Keine direkten Ereignis-Eingänge vorhanden. Ereignisse werden über den **AB_IN**-Adapter empfangen.

Ereignis-Ausgänge

- Keine direkten Ereignis-Ausgänge vorhanden. Ereignisse werden über den **AW_OUT**-Adapter gesendet.

Daten-Eingänge

- Keine direkten Daten-Eingänge vorhanden. Daten werden über den **AB_IN**-Adapter empfangen.

Daten-Ausgänge

- Keine direkten Daten-Ausgänge vorhanden. Daten werden über den **AW_OUT**-Adapter gesendet.

Adapter

Richtung	Name	Typ	Kommentar
Eingang (Socket)	**AB_IN**	`adapter::types::unidirectional::AB`	BYTE-Adapter-Eingang, liefert ein BYTE-Datum und ein Ereignis
Ausgang (Plug)	**AW_OUT**	`adapter::types::unidirectional::AW`	WORD-Adapter-Ausgang, stellt ein WORD-Datum und ein Ereignis bereit

Funktionsweise

Der Baustein leitet die eingehenden Signale des **AB_IN**-Adapters direkt an den **AW_OUT**-Adapter weiter:

- Das Ereignis **E1** des **AB_IN** wird mit dem Ereignis **E1** des **AW_OUT** verbunden.
- Das Datum **D1** (BYTE) des **AB_IN** wird mit dem Datum **D1** (WORD) des **AW_OUT** verbunden.

Da es sich um eine reine Durchleitung handelt, erfolgt keine Datenumrechnung im Sinne einer Wertänderung; die BYTE-Daten werden als WORD-Daten interpretiert (implizite Typanpassung durch die Datenverbindung). Die tatsächliche Konvertierung (z. B. Erweiterung des Wertebereichs) hängt von der Implementierung der zugrunde liegenden Adapter ab.

Technische Besonderheiten

- **Bit-Reinterpretation** (strukturelle Kopie zwischen Bit-String-Typen; schneidet bei Verengung ab, erweitert mit Nullen bei Erweiterung).
- Der Baustein ist als **Composite-Baustein** realisiert, d. h. sein Verhalten wird vollständig durch das interne Netzwerk (FBNetwork) definiert.
- Die Konvertierung ist unidirektional: Nur vom AB-Adapter zum AW-Adapter.
- Die Verbindung erfolgt auf Ereignis- und Datenebene ohne Zwischenspeicherung oder Synchronisation.
- Der Baustein ist Teil der Paketstruktur `adapter::conversion::unidirectional`.

Zustandsübersicht

Der Baustein besitzt **keinen eigenen Zustandsautomaten**. Das Verhalten ist rein kombinatorisch: Jedes eingehende Ereignis wird sofort auf den Ausgang weitergeleitet, ebenso wie die Daten.

Anwendungsszenarien

- **Anbindung von Sensoren/Aktoren**: Ein Modul liefert einen BYTE-Wert (z. B. über einen AB-Adapter), nachgeschaltete Logik erwartet jedoch WORD-Format.
- **Protokollanpassung**: In heterogenen Systemen, in denen unterschiedliche Adaptertypen aufeinandertreffen.
- **Systemintegration**: Wenn eine bestehende Ablaufsteuerung auf BYTE-Basis in eine WORD-basierte Umgebung eingebunden werden muss.

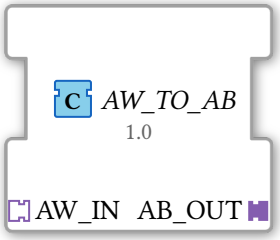
Vergleich mit ähnlichen Bausteinen

- **WORD_TO_BYTE**: Ggf. existiert ein analoger Baustein für die umgekehrte Richtung (WORD → BYTE).
- **Direkte Adapterverbindung**: Ohne diesen Baustein müsste der Programmierer die Ereignis- und Datenverknüpfung manuell im Netzwerk vornehmen; **AB_TO_AW** kapselt diese Verbindung und erhöht die Wiederverwendbarkeit.
- **Typkonverter**: Reine Datenkonverter (z. B. `BYTE_TO_WORD`) arbeiten oft ohne Ereignisweiterleitung; hier wird auch das Ereignis durchgeschliffen.

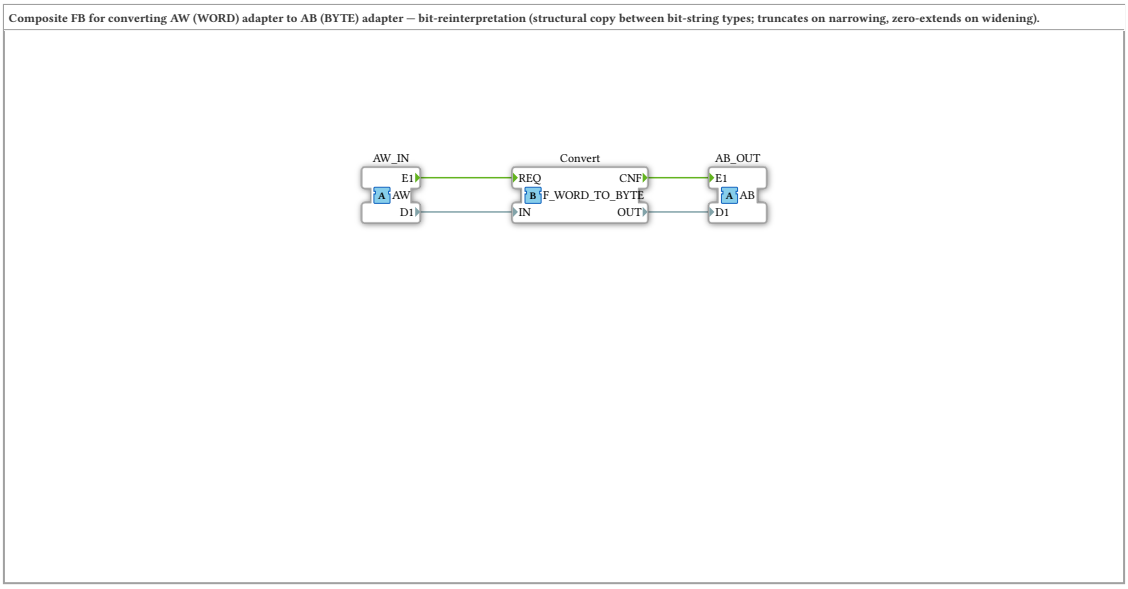
Fazit

Der **AB_TO_AW** ist ein einfacher, aber nützlicher Schnittstellenbaustein zur unidirektionalen Konvertierung eines BYTE-Adapters in einen WORD-Adapter. Durch die Kapselung der Verbindungslogik wird die Übersichtlichkeit im Applikationsnetzwerk erhöht und die Wiederverwendung standardisierter Konvertierungsmuster gefördert.

AW_TO_AB



WORD Adapter Input – *AW* — *AB* – BYTE Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein `AW_TO_AB` ist ein Composite-Baustein, der einen unidirektionalen WORD-Adapter (`AW`) in einen unidirektionalen BYTE-Adapter (`AB`) umwandelt. Er dient als Schnittstellenkonverter zwischen Komponenten, die auf unterschiedlichen Datentypen basieren. Die Konvertierung erfolgt durch den integrierten Standardfunktionsbaustein `F_WORD_TO_BYTE`.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine eigenen Ereignis-Eingänge. Stattdessen wird das Triggersignal über den Adapter-Eingang `AW_IN` (Ereignis `E1`) bereitgestellt.

Ereignis-Ausgänge

Der Baustein besitzt keine eigenen Ereignis-Ausgänge. Das Ergebnis wird über den Adapter-Ausgang `AB_OUT` (Ereignis `E1`) signalisiert.

Daten-Eingänge

Der Baustein besitzt keine eigenen Daten-Eingänge. Der zu konvertierende WORD-Wert wird über den Adapter-Eingang `AW_IN` (Daten `D1`) bereitgestellt.

Daten-Ausgänge

Der Baustein besitzt keine eigenen Daten-Ausgänge. Das konvertierte BYTE-Ergebnis wird über den Adapter-Ausgang `AB_OUT` (Daten `D1`) ausgegeben.

Adapter

Name	Richtung	Typ	Beschreibung
`AW_IN`	Socket (Eingang)	`adapter::types::unidirectional::AW`	WORD-Adapter mit einem Ereignis `E1` und einem Datenwert `D1` (WORD)
`AB_OUT`	Plug (Ausgang)	`adapter::types::unidirectional::AB`	BYTE-Adapter mit einem Ereignis `E1` und einem Datenwert `D1` (BYTE)

Funktionsweise

Der Baustein realisiert eine reine Datenkonvertierung von WORD (16 Bit) zu BYTE (8 Bit). Der Ablauf ist:

1. Ein Ereignis am Adapter-Eingang `AW_IN.E1` löst den internen Funktionsbaustein `F_WORD_TO_BYTE` aus.
2. Der Datenwert `AW_IN.D1` (WORD) wird als Eingang an `F_WORD_TO_BYTE.IN` übergeben.
3. Nach der Konvertierung wird das Ergebnis über `F_WORD_TO_BYTE.OUT` an den Adapter-Ausgang `AB_OUT.D1` weitergeleitet.
4. Der Ausgang `AB_OUT.E1` wird aktiviert, um die Fertigstellung zu signalisieren.

Die Konvertierung gemäß IEC 61131-3 liefert standardmäßig das niederwertigste Byte (Low-Byte) des WORD-Wertes.

Technische Besonderheiten

- **Bit-Reinterpretation** (strukturelle Kopie zwischen Bit-String-Typen; schneidet bei Verengung ab, erweitert mit Nullen bei Erweiterung).
- **Unidirektionale Adapter**: Der Baustein arbeitet nur in einer Datenflussrichtung – vom Socket zum Plug.
- **Integrierte Konvertierung**: Die eigentliche Umwandlung erfolgt durch den etablierten Baustein `F_WORD_TO_BYTE`, sodass keine eigene Logik implementiert werden muss.
- **Datenverlust**: Da ein WORD (16 Bit) auf ein BYTE (8 Bit) abgebildet wird, gehen die oberen 8 Bit verloren. Dies muss bei der Anwendung berücksichtigt werden.
- **Paketstruktur**: Der Baustein ist im Package `adapter::conversion::unidirectional` eingeordnet und nutzt das Import `iec61131::conversion::F_WORD_TO_BYTE`.

Zustandsübersicht

Der Composite-Baustein besitzt keinen eigenen Zustandsautomaten. Die Funktionsweise ist rein ereignisgesteuert:

- **Idle**: Warten auf ein Ereignis am Socket `AW_IN`.
- **Konvertierung aktiv**: Nach Empfang eines Ereignisses wird der interne FB `F_WORD_TO_BYTE` ausgeführt (kombinatorische Logik ohne Wartezyklen).
- **Ausgabe**: Sobald die Konvertierung abgeschlossen ist, wird das Ereignis am Plug `AB_OUT` und der neue Datenwert ausgegeben.

Anwendungsszenarien

- **Protokollanpassung**: Wenn ein Sensor einen WORD-Wert liefert, der Aktor aber nur BYTE-fähige Adapter unterstützt.
- **Schnittstellenreduktion**: In hierarchischen Steuerungssystemen, bei denen aus Performance- oder Speichergründen auf BYTE-Größen zurückgegriffen wird.
- **Adapter-Brücke**: Als Zwischenstück in einer 4diac-IDE-Applikation, um Komponenten mit inkompatiblen Adaptertypen zu verbinden.

Vergleich mit ähnlichen Bausteinen

- **`F_WORD_TO_BYTE`**: Ein reiner Datenkonvertierungsbaustein ohne Adapter-Integration. `AW_TO_AB` kapselt diesen Baustein und stellt eine standardisierte Adapter-Schnittstelle bereit.
- **`AW_TO_AW` (identischer Typ)**: Nicht sinnvoll; der vorliegende Baustein ist speziell für die Typumwandlung konzipiert.
- **Manuelle Konvertierung**: Ohne diesen Baustein müsste der Entwickler im Netzwerk manuell den `F_WORD_TO_BYTE` einfügen und die Adapterverbindungen selbst verdrahten – der Composite-Baustein vereinfacht dies.

Fazit

`AW_TO_AB` ist ein einfacher, aber nützlicher Composite-Baustein zur Umwandlung eines WORD-Adapters in einen BYTE-Adapter. Er reduziert den Verdrahtungsaufwand in 4diac-Applikationen und gewährleistet eine konsistente Konvertierung durch Wiederverwendung des Standardbausteins `F_WORD_TO_BYTE`. Anwender sollten sich des möglichen Datenverlusts durch die Reduzierung der Bitbreite bewusst sein.

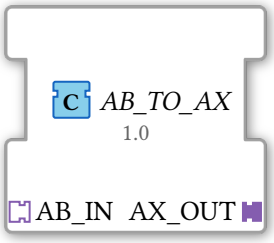
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

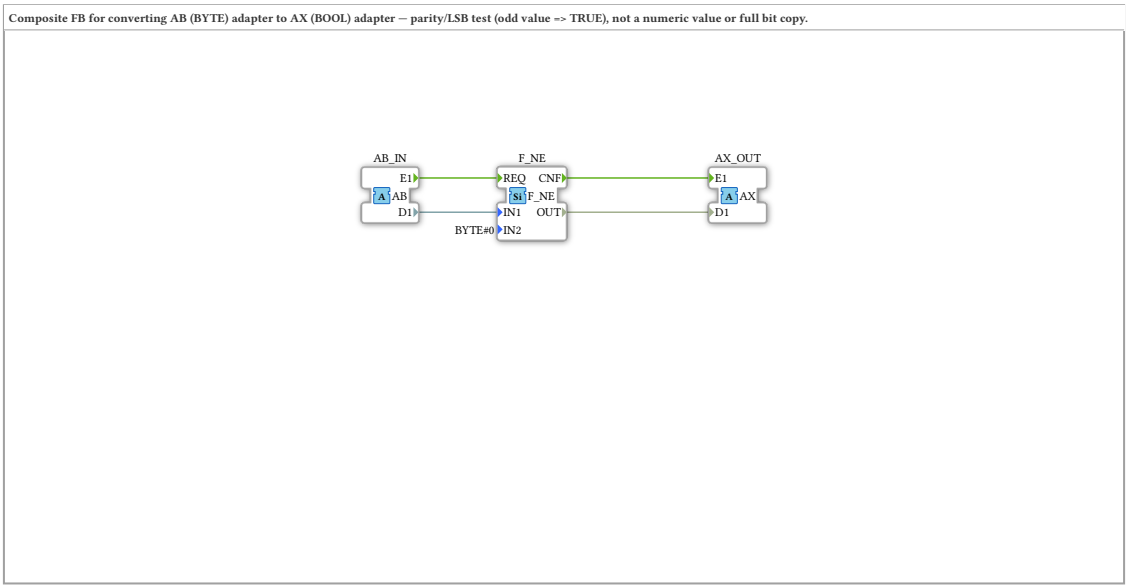
AB_AX

AB_TO_AX

BYTE Adapter Input – *AB*



AX – BOOL Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AB_TO_AX** ist ein zusammengesetzter Baustein (Composite FB) zur Umwandlung eines BYTE-Signals (über den AB-Adapter) in ein BOOL-Signal (über den AX-Adapter). Er dient als Konverter zwischen zwei unterschiedlichen Adaptertypen der unidirektionalen Adapterfamilie.

Schnittstellenstruktur

Adapter (Eingang)

- **AB_IN** (Typ: `adapter::types::unidirectional::AB`): Nimmt einen BYTE-Wert sowie ein zugehöriges Ereignis entgegen. Der Adapter stellt die Daten über die internen Kanäle D1 (BYTE) und E1 (Ereignis) bereit.

Adapter (Ausgang)

- **AX_OUT** (Typ: `adapter::types::unidirectional::AX`): Gibt einen BOOL-Wert sowie ein auslösendes Ereignis aus. Der Adapter liefert das Ergebnis über D1 (BOOL) und E1 (Ereignis).

Funktionsweise

Der Baustein verwendet intern einen `F_NE`-Vergleichsbaustein (ungleich). Das eingehende BYTE-Signal (AB_IN.D1) wird mit dem konstanten Wert `BYTE#0` verglichen.

- Wenn der BYTE-Wert ungleich 0 ist, liefert der Vergleich `TRUE`.
- Wenn der BYTE-Wert gleich 0 ist, liefert der Vergleich `FALSE`.

Der Ablauf wird durch das Ereignis `AB_IN.E1` gestartet. Dieses triggert den `F_NE`-Baustein. Nach Abschluss des Vergleichs wird das Ereignis `AX_OUT.E1` ausgelöst und das Ergebnis (BOOL) über `AX_OUT.D1` ausgegeben.

Technische Besonderheiten

- **Paritäts-/LSB-Test** (ungerader Wert ⟹ TRUE), kein Zahlenwert und keine vollständige Bit-Kopie.
- Der Baustein ist als Composite realisiert, d.h. seine Funktionalität ergibt sich aus der Verschaltung von Standard-FBs (hier `F_NE`).
- Er ist für die Verwendung in der 4diac-IDE mit der Spezifikation IEC 61499-2 ausgelegt.
- Die Adapter `AB` und `AX` sind unidirektional, sodass die Daten- und Ereignisflüsse klar getrennt sind.
- Der interne Vergleich erfolgt binär: Jeder von Null verschiedene BYTE-Wert wird als `TRUE` interpretiert.

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Die gesamte Verarbeitung erfolgt ereignisgesteuert über die internen Verbindungen. Der `F_NE`-Baustein arbeitet deterministisch ohne Zwischenzustände.

Anwendungsszenarien

- **Signalkonvertierung**: Ein digitaler Sensor liefert einen Byte-Wert (z.B. Zählerstand oder Status). AB_TO_AX wandelt diesen in ein boolsches Signal um, das z.B. eine Alarmierung auslöst, sobald der Wert ungleich Null ist.
- **Adapterbrücke**: In einer Steuerungskomponente wird ein AB-Adapter verwendet, die nachgeschaltete Logik erwartet jedoch einen AX-Adapter. Der Baustein fungiert als einfacher Übersetzer.
- **Schwellwertersetzung**: Ersetzt einen expliziten Vergleichsbaustein durch einen kompakten Adapter-Konverter.

Vergleich mit ähnlichen Bausteinen

Andere Konverterbausteine wie `BOOL_TO_BYTE` oder `BYTE_TO_BOOL` existieren als einfache Datenumwandler, arbeiten jedoch meist auf Datenebene ohne Adapter. AB_TO_AX integriert die Adapterschnittstelle und ermöglicht so eine nahtlose Einbindung in adapterbasierte Architekturen. Im Gegensatz zu einem allgemeinen Vergleichsbaustein (`F_NE`) kapselt dieser FB die spezifische Konvertierungslogik und bietet eine aufgeräumte Schnittstelle.

Fazit

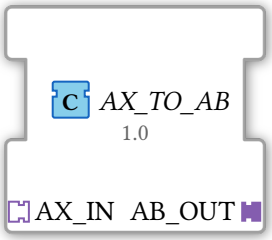
Der **AB_TO_AX**-Baustein bietet eine saubere und einfache Methode, um ein BYTE-Signal aus einem AB-Adapter in ein BOOL-Signal für einen AX-Adapter zu wandeln. Durch die Nutzung des Standardvergleichs `Ungleich Null` ist er besonders für binäre Schwellwertentscheidungen geeignet und vereinfacht die Wiederverwendung in adapterorientierten Steuerungsprojekten.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

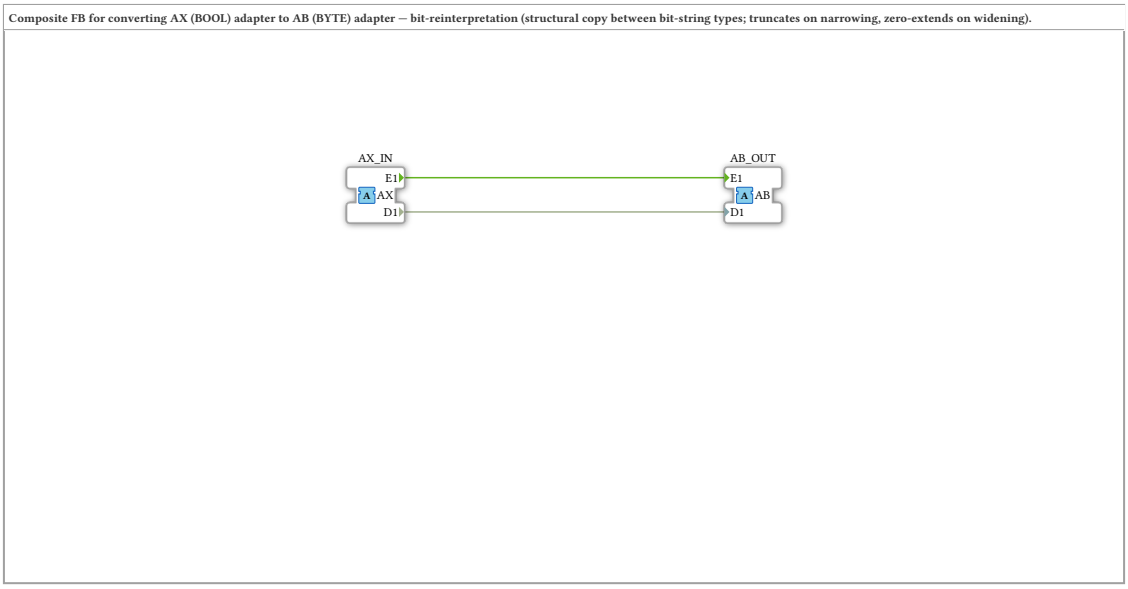
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AX_TO_AB

BOOL Adapter Input – *AX*



AB – BYTE Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Composite-Funktionsblock `AX_TO_AB` dient zur Umwandlung eines BOOL-Adaptersignals vom Typ `AX` in ein BYTE-Adaptersignal vom Typ `AB`. Er kapselt die Typkonvertierung und das Durchreichen von Ereignissen in einer wiederverwendbaren Einheit.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Beschreibung
`E1` (aus `AX_IN`)	Ereigniseingang, der über den BOOL-Adapter empfangen und unverändert an den Ausgang weitergeleitet wird.

Ereignis-Ausgänge

Name	Beschreibung
`E1` (aus `AB_OUT`)	Ereignisausgang, der das durchgereichte Ereignis vom Eingang bereitstellt.

Daten-Eingänge

Name	Datentyp	Beschreibung
`D1` (aus `AX_IN`)	`BOOL`	Das zu konvertierende boolesche Signal (TRUE oder FALSE).

Daten-Ausgänge

Name	Datentyp	Beschreibung
`D1` (aus `AB_OUT`)	`BYTE`	Das konvertierte Signal: `TRUE` wird zu `1`, `FALSE` zu `0`.

Adapter

Typ	Richtung	Bezeichner	Kommentar
`adapter::types::unidirectional::AX`	Socket (Eingang)	`AX_IN`	BOOL-Adapter, der die Eingangsdaten und Ereignisse liefert.

adapter::types::unidirectional::	AB	Plug (Ausgang)	AB_OUT	BYTE-Adapter, der die konvertierten Daten und Ereignisse ausgibt.

Funktionsweise

Der Funktionsblock besteht intern nur aus Verbindungen, die den Ereignis- und Datenpfad vom Socket AX_IN zum Plug AB_OUT durchschalten.

- Das Ereignis E1 wird ohne Verzögerung oder Veränderung weitergereicht.
- Der Datenwert D1 wird vom Typ BOOL in den Typ BYTE gewandelt. Dabei wird der boolesche Wert TRUE in das Byte 1 und FALSE in das Byte 0 umgesetzt. Die Konvertierung erfolgt implizit durch die Laufzeitumgebung von 4diac.

Da es sich um einen Composite-FB handelt, werden alle internen Verbindungen zur Entwurfszeit festgelegt; es gibt keine dynamische Logik oder Zustandsautomaten.

Technische Besonderheiten

- **Bit-Reinterpretation** (strukturelle Kopie zwischen Bit-String-Typen; schneidet bei Verengung ab, erweitert mit Nullen bei Erweiterung).
- **Composite-Typ**: Der Baustein ist ein reiner Netzwerk-FB ohne eigene Algorithmen. Die Konvertierung wird durch die Verbindung zweier unterschiedlicher Adaptertypen erreicht.
- **Typkonvertierung**: Die Umwandlung von BOOL zu BYTE ist standardkonform und nutzt die implizite Cast-Funktion der 4diac-IDE. Dadurch entfällt der Einsatz eines separaten Konvertierungsbausteins.
- **Versionierung**: Der FB ist mit Version 1.0 vom 17.02.2026 versehen und für die Nutzung im Package adapter::conversion::unidirectional vorgesehen.

Zustandsübersicht

Der Baustein besitzt keine internen Zustände – er arbeitet rein kombinatorisch. Die Ausgangswerte sind zu jedem Zeitpunkt eine direkte Funktion der Eingangswerte.

Anwendungsszenarien

- **Anbindung von Sensoren/Aktoren**: Wenn ein Gerät oder eine Steuerung als BOOL-Signal arbeitet, das Zielsystem aber ein BYTE-Adapterformat erwartet, kann AX_TO_AB als Vermittler eingesetzt werden.
- **Adapterbrücken**: In heterogenen Automatisierungsnetzwerken, in denen unterschiedliche Adapterprotokolle verwendet werden, erlaubt der FB eine einfache und typsichere Verbindung.
- **Test und Simulation**: Der FB kann als einfacher Konverter in Testumgebungen genutzt werden, um Signale zwischen verschiedenen Logikschichten zu übersetzen.

Vergleich mit ähnlichen Bausteinen

Es existieren spezielle Konvertierungs-FBs wie BOOL_TO_BYTE oder BYTE_TO_BOOL, die jedoch meist als Basis-FBs (Basic Function Blocks) mit einem Algorithmus realisiert sind. Der AX_TO_AB ist dagegen ein Composite-FB auf Adapterebene und kapselt die gesamte (Ereignis- und Daten-) Kommunikation. Er ist dann sinnvoll, wenn ganze Adapterschnittstellen umgesetzt werden müssen, nicht nur einzelne Datenpunkte.

Fazit

Der Composite-FB AX_TO_AB bietet eine kompakte und elegante Lösung zur Umwandlung eines BOOL-Adapter in einen BYTE-Adapter. Durch die implizite Typkonvertierung und die direkte Ereignisdurchleitung reduziert er den manuellen Anpassungsaufwand und erhöht die Wiederverwendbarkeit in modularen Automatisierungsprojekten.

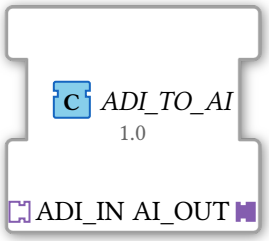
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

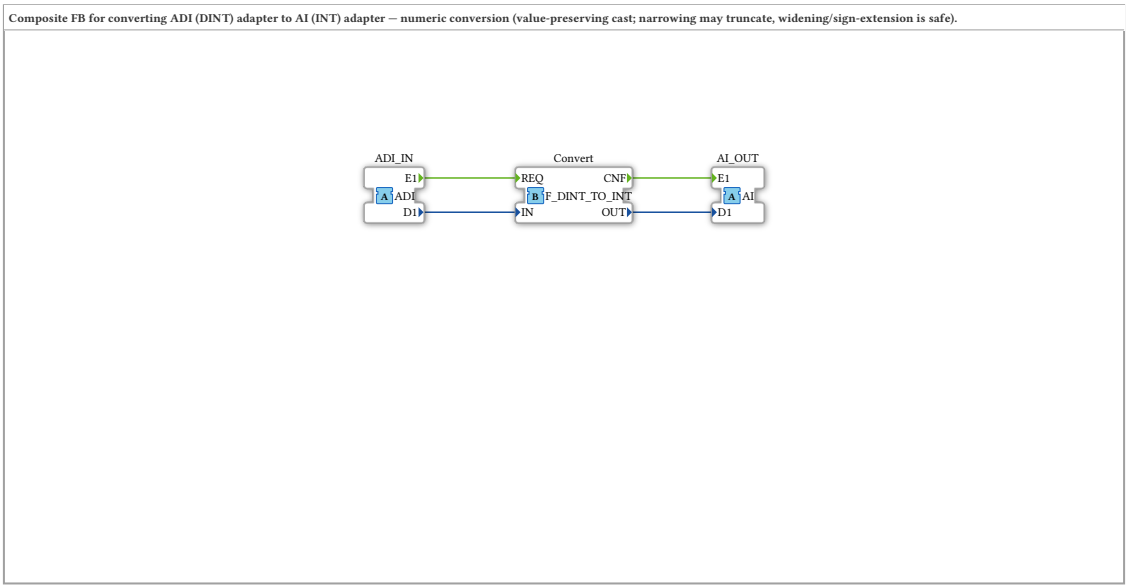
ADI_AI

ADI_TO_AI

DINT Adapter Input – *ADI*



AI – INT Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **ADI_TO_AI** ist ein Composite FB, der einen DINT-Adapter (ADI) in einen INT-Adapter (AI) umwandelt. Er dient als unidirektionale Konvertierungsschnittstelle zwischen zwei Adapter-Typen und nutzt intern den Baustein `F_DINT_TO_INT` zur Typumwandlung der Datenwerte.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine separaten Ereignis-Eingänge. Die ereignisgesteuerte Verarbeitung erfolgt über den **Socket ADI_IN**:

- **ADI_IN.E1** – Ereignis zur Anforderung der Konvertierung (wird intern an den `Convert.REQ` weitergeleitet).

Ereignis-Ausgänge

Der FB besitzt keine separaten Ereignis-Ausgänge. Die Ergebnis-Rückmeldung erfolgt über den **Plug AI_OUT**:

- **AI_OUT.E1** – Ereignis, das nach erfolgreicher Konvertierung ausgelöst wird (entspricht `Convert.CNF`).

Daten-Eingänge

Der FB besitzt keine separaten Daten-Eingänge. Die Eingangsdaten werden über den **Socket ADI_IN** bereitgestellt:

- **ADI_IN.D1** – DINT-Wert (32-Bit-Ganzzahl), der in einen INT-Wert umgewandelt werden soll.

Daten-Ausgänge

Der FB besitzt keine separaten Daten-Ausgänge. Die Ergebnisdaten werden über den **Plug AI_OUT** ausgegeben:

- **AI_OUT.D1** – INT-Wert (16-Bit-Ganzzahl), das Ergebnis der Konvertierung.

Adapter

- **Socket**: `ADI_IN` (Typ `adapter::types::unidirectional::ADI`) – DINT-Adapter-Eingang.
- **Plug**: `AI_OUT` (Typ `adapter::types::unidirectional::AI`) – INT-Adapter-Ausgang.

Funktionsweise

Der Baustein verbindet die Adapter-Schnittstellen direkt mit einer internen Instanz des Konvertierungsbausteins `F_DINT_TO_INT`:

1. Ein Ereignis an `ADI_IN.E1` triggert den `Convert.REQ`.
2. Der Wert von `ADI_IN.D1` wird an `Convert.IN` übergeben.
3. Die Konvertierung `DINT → INT` wird durchgeführt.
4. Nach Abschluss signalisiert `Convert.CNF` das Ereignis an `AI_OUT.E1` und der konvertierte Wert von `Convert.OUT` wird an `AI_OUT.D1` weitergeleitet.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Keine eigene Zustandsmaschine**: Der FB ist ein reines Composite (Netzwerk) ohne ECC. Das Verhalten wird ausschließlich durch die interne Verbindung und den verwendeten Konvertierungsbaustein bestimmt.
- **Typkonvertierung**: Die Umwandlung von DINT nach INT erfolgt gemäß IEC 61131-3 (Überlauf bei Wertebereichsüberschreitung wird nicht abgefangen; INT-Bereich: −32768 … 32767).
- **Paket**: Der FB ist im Paket `adapter::conversion::unidirectional` organisiert.

Zustandsübersicht

Der FB besitzt keine explizite Zustandsmaschine. Das Verhalten ist rein datenflussgesteuert: Auf jedes eingehende Ereignis erfolgt genau eine Konvertierung und eine Ausgabe.

Anwendungsszenarien

- **Integration von DINT-basierten Komponenten** in eine Umgebung, die ausschließlich INT-Adapter unterstützt (z. B. ältere Steuerungen oder standardisierte Schnittstellen).
- **Typanpassung** in heterogenen Automatisierungssystemen, bei denen unterschiedliche Datenbreiten verwendet werden.

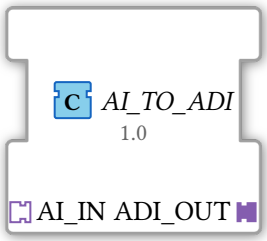
Vergleich mit ähnlichen Bausteinen

- **AI_TO_ADI**: Die umgekehrte Konvertierung (INT → DINT). Während `ADI_TO_AI` den Wertebereich verkleinert (potenzieller Datenverlust), erweitert `AI_TO_ADI` den Wertebereich ohne Informationsverlust.
- **Direkte Adapter-Kopplung**: Ohne Konvertierung müssten die Adapter-Typen exakt übereinstimmen. `ADI_TO_AI` ermöglicht die Kommunikation zwischen inkompatiblen Typen.

Fazit

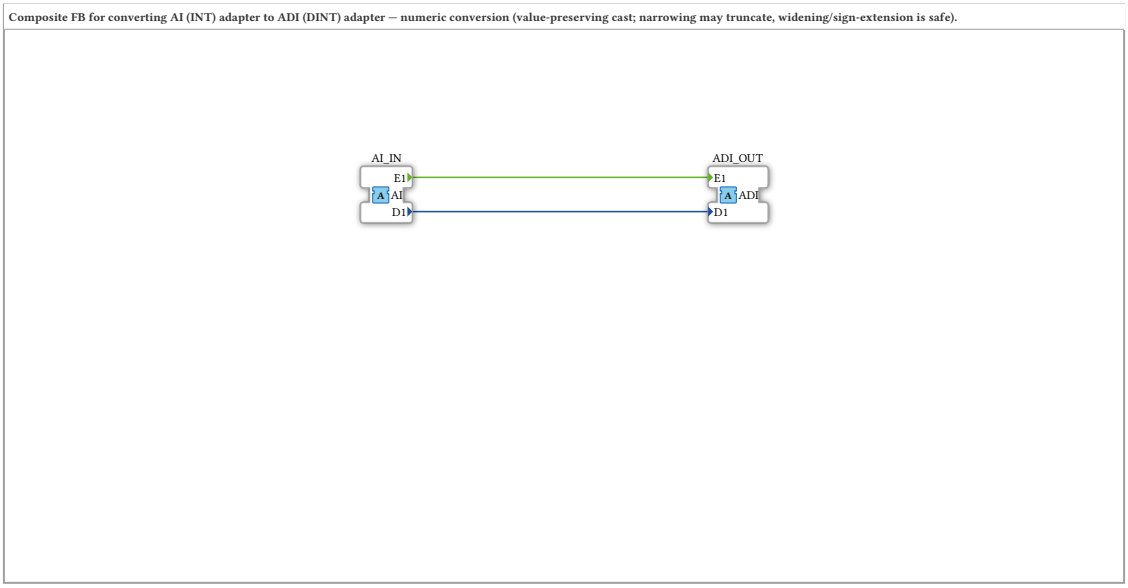
Der Baustein `ADI_TO_AI` ist ein einfacher, aber nützlicher Konverter, der die Lücke zwischen DINT- und INT-Adapter-Schnittstellen schließt. Durch seine Composite-Struktur ist er leicht verständlich und wartbar. Die Verwendung des standardisierten `F_DINT_TO_INT` stellt die korrekte Umwandlung sicher.

AI_TO_ADI



INT Adapter Input – *AI*

ADI – DINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein AI_TO_ADI ist ein Composite-Funktionsblock, der einen AI-Adapter (Integer-Werte, Typ INT) in einen ADI-Adapter (Double-Integer-Werte, Typ DINT) umwandelt. Er dient als Brücke zwischen Komponenten, die unterschiedliche Adapter-Schnittstellen verwenden.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine eigenen Ereignis-Eingänge. Das Ereignis wird über den eingehenden Adapter AI_IN bereitgestellt:

- **AI_IN.E1** (Ereignis): Auslöser für die Weiterleitung des Datenwerts und des Ereignisses an den Ausgang.

Ereignis-Ausgänge

Der FB besitzt keine eigenen Ereignis-Ausgänge. Das Ereignis wird über den ausgehenden Adapter ADI_OUT ausgegeben:

- **ADI_OUT.E1** (Ereignis): Wird nach erfolgreicher Datenweitergabe ausgelöst.

Daten-Eingänge

Der FB besitzt keine eigenen Daten-Eingänge. Die Daten werden über den eingehenden Adapter AI_IN bereitgestellt:

- **AI_IN.D1** (INT): Der umzuwandelnde Integer-Wert.

Daten-Ausgänge

Der FB besitzt keine eigenen Daten-Ausgänge. Die Daten werden über den ausgehenden Adapter ADI_OUT ausgegeben:

- **ADI_OUT.D1** (DINT): Der konvertierte Double-Integer-Wert.

Adapter

Name	Richtung	Typ	Beschreibung
AI_IN	Socket	adapter::types::unidirectional::AI	Eingangsadapter für Integer-Werte (INT)
ADI_OUT	Plug	adapter::types::unidirectional::ADI	Ausgangsadapter für Double-Integer-Werte (DINT)

Funktionsweise

Der Baustein arbeitet als reine Weiterleitung: Sobald am eingehenden Adapter `AI_IN` das Ereignis `E1` anliegt, wird der aktuelle Wert von `AI_IN.D1` gelesen und unverändert an `ADI_OUT.D1` weitergegeben. Anschließend wird das Ereignis `E1` am ausgehenden Adapter `ADI_OUT` ausgelöst. Die Typkonvertierung von `INT` nach `DINT` erfolgt automatisch durch die Adapterbindung; es wird keine explizite Konvertierungslogik benötigt.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Implizite Typkonvertierung:** Der FB führt die Umwandlung von 16-Bit Integer (INT) auf 32-Bit Double Integer (DINT) ohne Datenverlust durch, da der Ziel-Datentyp größer ist.
- **Composite-Implementierung:** Der Baustein ist als Composite-FB realisiert, d.h. er enthält ein internes Netzwerk, das die beiden Adapter direkt miteinander verbindet. Er kann wie ein primitiver FB in Steuerungsprojekten eingesetzt werden.
- **Unidirektionale Kommunikation:** Beide Adapter sind unidirektional ausgelegt – die Daten fließen nur vom Eingang zum Ausgang.
- **Keine Latenz oder Pufferung:** Die Weiterleitung erfolgt ereignisgesteuert und ohne zusätzliche Verzögerung.

Zustandsübersicht

Der AI_TO_ADI besitzt keinen internen Zustandsautomaten (ECC). Er verhält sich passiv und leitet jedes eingehende Ereignis und die zugehörigen Daten sofort weiter. Es gibt keine internen Zustände, Wartezeiten oder Speicherung von Werten.

Anwendungsszenarien

- **Adapter-Brücke:** Wenn ein Sensormodul einen AI-Adapter (INT) bereitstellt, ein nachfolgendes Verarbeitungsmodul jedoch einen ADI-Adapter (DINT) erwartet, kann der AI_TO_ADI zwischengeschaltet werden.
- **Systemintegration:** In heterogenen Automatisierungssystemen, in denen unterschiedliche Datentypen für Adapter-Schnittstellen verwendet werden, ermöglicht dieser Baustein eine einfache Anpassung ohne Änderung der ursprünglichen Komponenten.
- **Prototyping:** Schnelle Kopplung von Funktionsbausteinen aus verschiedenen Bibliotheken, die auf unterschiedlichen Adaptertypen basieren.

Vergleich mit ähnlichen Bausteinen

Baustein	Typkonvertierung	Schnittstelle	Anwendungsfall
AI_TO_ADI	INT -> DINT	Adapter (Socket/Plug)	Brücke zwischen adapterbasierten Komponenten
INT_TO_DINT	INT -> DINT	Direkte Eingänge/Ausgänge (z.B. DI1, DO1)	Einfache Datenkonvertierung ohne Adapter
CONV_ANY_TO_ANY	Generisch	Flexible Ein-/Ausgänge	Universelle Konvertierung (oft aufwändiger)

Während Bausteine wie `INT_TO_DINT` eine direkte Datenkonvertierung mit eigenen Ein- und Ausgängen bieten, integriert der AI_TO_ADI die Konvertierung nahtlos in eine Adapter-basierte Architektur und erleichtert so die Wiederverwendung vorhandener Adapterdefinitionen.

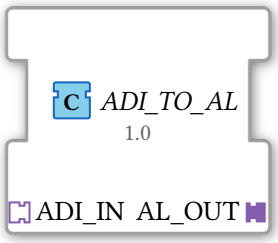
Fazit

Der AI_TO_ADI ist ein einfacher, aber wertvoller Funktionsblock für die Umwandlung von Integer- auf Double-Integer-Adapter-Schnittstellen. Durch seine Composite-Struktur und die implizite Typkonvertierung lässt er sich ohne zusätzliche Programmierlogik in adapterbasierte Steuerungssysteme einfügen und vereinfacht die Integration von Komponenten mit unterschiedlichen Datenbreiten.

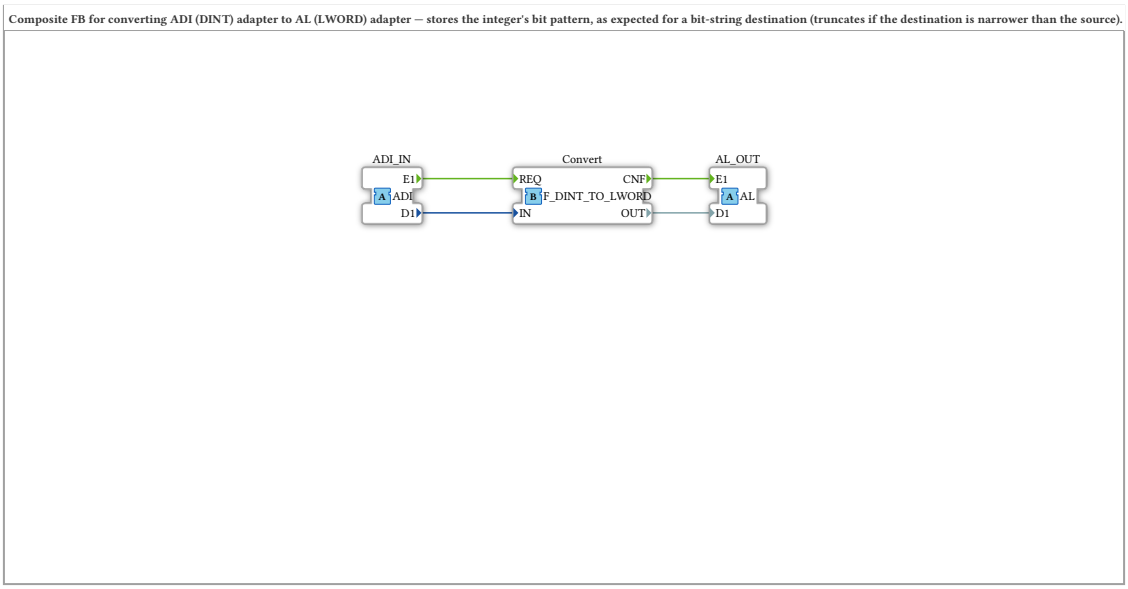
ADI_AL

ADI_TO_AL

DINT Adapter Input – *ADI*



AL – LWORD Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `ADI_TO_AL` dient als Composite-Baustein zur Konvertierung eines ADI-Adapters (DINT) in einen AL-Adapter (LWORD). Er kapselt eine Typumwandlung von einem 32-Bit-Ganzzahlwert in ein 64-Bit-Wort und stellt die Verbindung über unidirektionale Adapter bereit. Der Baustein ist ideal für Systeme, in denen eine unterschiedliche Datentypdarstellung zwischen Komponenten überbrückt werden muss.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Datenein-/-ausgänge. Die gesamte Kommunikation erfolgt über zwei Adapter-Schnittstellen:

Ereignis-Eingänge

- **Keine direkten Ereignis-Eingänge** Das Ereignis wird über den ADI-Adapter `ADI_IN.E1` empfangen.

Ereignis-Ausgänge

- **Keine direkten Ereignis-Ausgänge** Das ausgehende Ereignis wird über den AL-Adapter `AL_OUT.E1` gesendet.

Daten-Eingänge

- **Keine direkten Daten-Eingänge** Die Eingangsdaten werden über den ADI-Adapter `ADI_IN.D1` (Datentyp: DINT) geliefert.

Daten-Ausgänge

- **Keine direkten Daten-Ausgänge** Die Ausgangsdaten werden über den AL-Adapter `AL_OUT.D1` (Datentyp: LWORD) bereitgestellt.

Adapter

Bezeichnung	Typ	Richtung	Kommentar
`ADI_IN`	`adapter::types::unidirectional::ADI`	Socket (Eingang)	Liefert einen DINT-Wert und steuert die Verarbeitung über Ereignis `E1`.
`AL_OUT`	`adapter::types::unidirectional::AL`	Plug (Ausgang)	Gibt den konvertierten LWORD-Wert aus und quittiert über Ereignis `E1`.

Funktionsweise

Der Baustein implementiert eine einfache Typumwandlungskette:

1. Ein über den ADI-Adapter `ADI_IN` eingehendes Ereignis (`E1`) startet die Konvertierung.
2. Der interne Funktionsblock `F_DINT_TO_LWORD` wandelt den eingehenden DINT-Wert (`ADI_IN.D1`) in einen LWORD-Wert um.
3. Nach erfolgreicher Konvertierung wird das Ergebnis als LWORD über den AL-Adapter (`AL_OUT.D1`) ausgegeben und ein Ereignis (`E1`) gesendet.

Die Kommunikation zwischen den Adaptern und dem Konvertierungsbaustein erfolgt asynchron über Ereignis- und Datenverbindungen im Netzwerk des Composite-Blocks.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Unidirektionale Adapter** – Der Baustein verwendet ausschließlich unidirektionale Adapter (Socket für Eingang, Plug für Ausgang), die eine klare Datenflussrichtung vorgeben.
- **Interne Konvertierung** – Die eigentliche Typumwandlung wird durch den Standardbaustein `F_DINT_TO_LWORD` aus der IEC-61131-Bibliothek realisiert, sodass keine eigene Implementierung notwendig ist.
- **Lizenz** – Der Baustein ist unter der Eclipse Public License 2.0 (EPL-2.0) lizenziert und kann frei verwendet, modifiziert und weitergegeben werden.
- **Keine Zustände** – Der Composite-Baustein besitzt keine eigene Zustandslogik; die gesamte Steuerung erfolgt über die Ereignisverkettung.

Zustandsübersicht

Der Baustein hat keine eigene Zustandsmaschine. Sein Verhalten ist rein reaktiv: Auf ein eingehendes Ereignis am ADI-Adapter wird unverzögert das Ergebnis am AL-Adapter ausgegeben. Interne Verzögerungen sind nicht definiert.

Anwendungsszenarien

- **Datenbrücke zwischen unterschiedlichen Bussystemen** – Wenn ein Sensor oder eine Steuerung einen DINT-Wert liefert, aber eine nachfolgende Komponente einen LWORD-Wert erwartet (z. B. bei 64-Bit-Aktoren oder Feldbus-Schnittstellen).

- **Middleware-Anpassung** – Einsatz in heterogenen Automatisierungssystemen, um Adapter-Typen ohne Änderung der ursprünglichen Komponenten zu harmonisieren.
- **Test- und Simulationsumgebungen** – Einfaches Umwandeln von Datentypen für Debugging- oder Ersatzverbindungen.

Vergleich mit ähnlichen Bausteinen

- **Einzelkonvertierungs-FBs** – Bausteine wie `DINT_TO_LWORD` (Direktkonvertierung ohne Adapter) bieten eine kompaktere Schnittstelle, erfordern aber eigene Ereignis- und Datenleitungen.
- **Universelle Adapter-Konverter** – Es existieren generische Konverter, die beliebige Datentypen via Adapter umwandeln. `ADI_TO_AL` ist speziell auf die Kombination DINT→LWORD optimiert und daher einfacher zu verstehen und zu warten.
- **Bidirektionale Adapter** – Im Gegensatz zu bidirektionalen Varianten beschränkt sich dieser Baustein auf eine Datenflussrichtung, was die Fehleranfälligkeit reduziert.

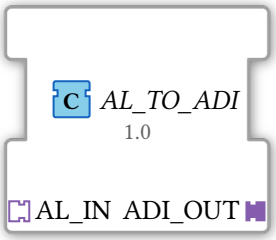
Fazit

Der `ADI_TO_AL`-Baustein bietet eine saubere, auf wenige Elemente reduzierte Lösung zur Typumwandlung von DINT nach LWORD über unidirektionale Adapter. Er ist leicht in bestehende 4diac-Projekte integrierbar, benötigt keine eigenen Zustandsmodellierungen und profitiert von der Wiederverwendbarkeit des internen Standard-Konvertierungsbausteins. Damit eignet er sich hervorragend für alle Anwendungen, bei denen eine schnelle und zuverlässige Datenanpassung zwischen unterschiedlichen Adapter-Typen erforderlich ist.

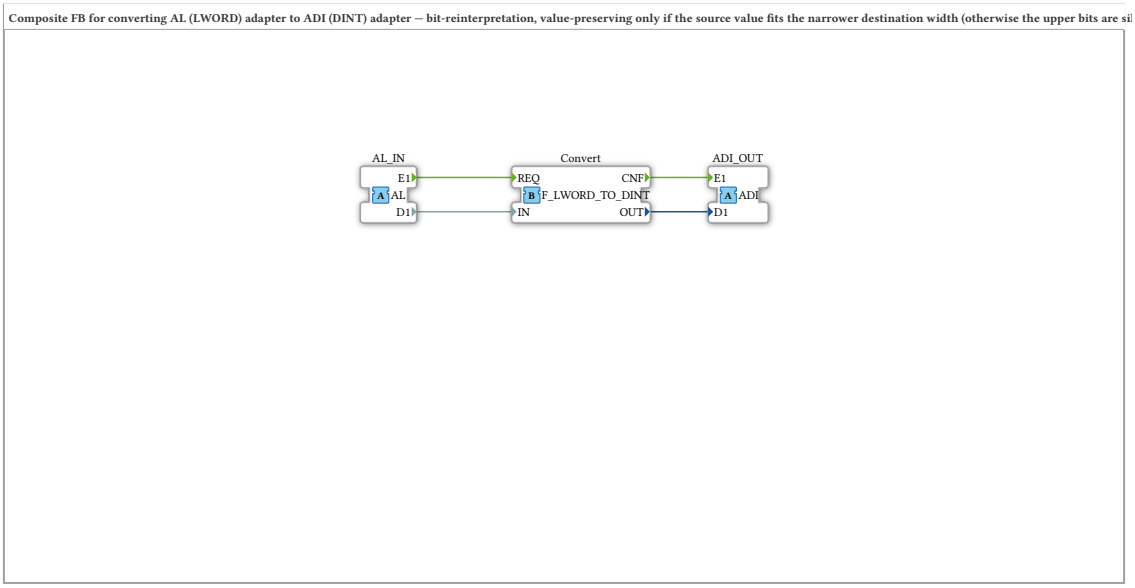
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AL_TO_ADI



LWORD Adapter Input – *AL* — *ADI* – DINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Composite-Funktionsblock `AL_TO_ADI` dient der standardisierten Umwandlung eines unidirektionalen LWORD-Adapters (`AL`) in einen unidirektionalen DINT-Adapter (`ADI`). Er ermöglicht die nahtlose Integration von Komponenten, die mit unterschiedlichen Datentypen in einer 4diac-Adapterlandschaft arbeiten.

Schnittstellenstruktur

Da der FB als Composite realisiert ist, besitzt er keine eigenen Ereignis- oder Datenschnittstellen außerhalb der Adapter. Alle Ein- und Ausgaben erfolgen über die integrierten Adapter.

Ereignis-Eingänge

Keine eigenen Ereignis-Eingänge. Das Ereignis zur Auslösung der Konvertierung wird über den Socket `AL_IN` (dort über das Ereignis `E1`) empfangen.

Ereignis-Ausgänge

Keine eigenen Ereignis-Ausgänge. Das Bestätigungsereignis nach abgeschlossener Konvertierung wird über den Plug `ADI_OUT` (dort über das Ereignis `E1`) gesendet.

Daten-Eingänge

Keine eigenen Daten-Eingänge. Der zu konvertierende LWORD-Wert wird über den Socket `AL_IN` (dort über das Datum `D1`) bereitgestellt.

Daten-Ausgänge

Keine eigenen Daten-Ausgänge. Der konvertierte DINT-Wert wird über den Plug `ADI_OUT` (dort über das Datum `D1`) ausgegeben.

Adapter

- **AL_IN (Socket)** Typ: `adapter::types::unidirectional::AL` Aufnahme eines unidirektionalen LWORD-Adapters. Stellt das Auslöseereignis (`E1`) und den LWORD-Eingabewert (`D1`) bereit.
- **ADI_OUT (Plug)** Typ: `adapter::types::unidirectional::ADI` Bereitstellung eines unidirektionalen DINT-Adapters. Liefert das Bestätigungsereignis (`E1`) und den konvertierten DINT-Ausgabewert (`D1`).

Funktionsweise

Der FB ist ein Composite-Baustein, der intern die IEC-61131-Standardfunktion `F_LWORD_TO_DINT` verwendet. Der Ablauf ist:

1. Ein Ereignis am Socket `AL_IN.E1` aktiviert den internen Konverter (`Convert.REQ`).
2. Der Konverter liest den LWORD-Wert von `AL_IN.D1`, wandelt ihn in einen DINT um und legt das Ergebnis an `Convert.OUT` an.
3. Nach erfolgreicher Konvertierung wird ein Ereignis von `Convert.CNF` an den Plug `ADI_OUT.E1` weitergeleitet.
4. Gleichzeitig wird der konvertierte DINT-Wert von `Convert.OUT` an `ADI_OUT.D1` übergeben.

Damit wird eine vollständige und zeitlich korrekte Umwandlung von einem LWORD-Adapter in einen DINT-Adapter realisiert.

Technische Besonderheiten

- **Bit-Reinterpretation**, werterhaltend nur, wenn der Quellwert in die schmalere Zielbreite passt (sonst werden die oberen Bits stillschweigend verworfen).
- Der FB nutzt die IEC-61131-3 Funktion `F_LWORD_TO_DINT` – eine standardisierte und zuverlässige Typkonvertierung.
- Es handelt sich um eine reine **unidirektionale** Konvertierung: Daten und Ereignisse fließen nur vom Socket (`AL_IN`) zum Plug (`ADI_OUT`).
- Der FB ist als Composite modelliert, was eine klare Trennung von Konvertierungslogik und Adapterschnittstellen ermöglicht und die Wiederverwendung fördert.
- Bei der Umwandlung von LWORD (64 Bit) nach DINT (32 Bit) können Werte außerhalb des DINT-Bereichs zu Überläufen oder Begrenzungen führen – dies ist durch die IEC-Funktion definiert.

Zustandsübersicht

Der FB besitzt keine eigene Zustandsmaschine (ECC), da er als Composite ohne sequentielle Logik realisiert ist. Die Funktionalität wird rein durch die ereignisgesteuerte Verkettung der internen Elemente abgebildet. Ein einmaliger Konvertierungsdurchlauf erfolgt bei jedem eingehenden Ereignis am `AL_IN`.

Anwendungsszenarien

- **Sensor-Anbindung:** Ein Sensor liefert LWORD-Daten (z. B. ein Zeitsignal oder eine große Zählvariable), die in einer Steuerung als DINT weiterverarbeitet werden müssen.
- **Systemintegration:** Brückenschlag zwischen Adapter-basierten Subsystemen, die unterschiedliche Datentypen verwenden – ohne dass der Anwender manuelle Konvertierungsbausteine einfügen muss.
- **Bibliothekserweiterung:** Einsatz als Teil einer Sammlung von unidirektionalen Typkonverter-Adaptern für verschiedene IEC-61131-3 Datentypen.

Vergleich mit ähnlichen Bausteinen

Ähnliche Konverter-Adapter in 4diac existieren etwa für `AL_TO_UDINT`, `BYTE_TO_DINT` oder `LWORD_TO_REAL`. Der `AL_TO_ADI` ist speziell auf die Kombination LWORD ⟶ DINT ausgelegt. Im Gegensatz zu einfachen Funktionsbausteinen, die nur Daten konvertieren, bietet er die vollständige Adapter-Kopplung mit Ereignissteuerung. Gegenüber bidirektionalen Adaptern ist er auf die Datenflussrichtung von AL zu ADI festgelegt. Ein möglicher Datenverlust durch die Verkleinerung der Bitbreite (64 ⟶ 32 Bit) muss bei der Auswahl berücksichtigt werden.

Fazit

Der Composite-FB `AL_TO_ADI` stellt eine saubere und standardisierte Lösung zur Konvertierung eines unidirektionalen LWORD-Adapters auf einen DINT-Adapter dar. Durch die Nutzung der IEC-61131-Funktion und die Einbettung in das 4diac-Adapterkonzept erleichtert er die Integration heterogener Systemkomponenten erheblich. Er ist wartungsfreundlich, wiederverwendbar und folgt bewährten Entwurfsmustern der industriellen Automatisierung.

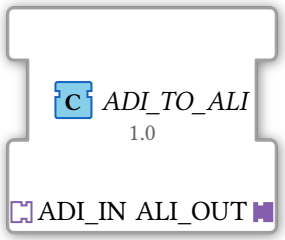
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

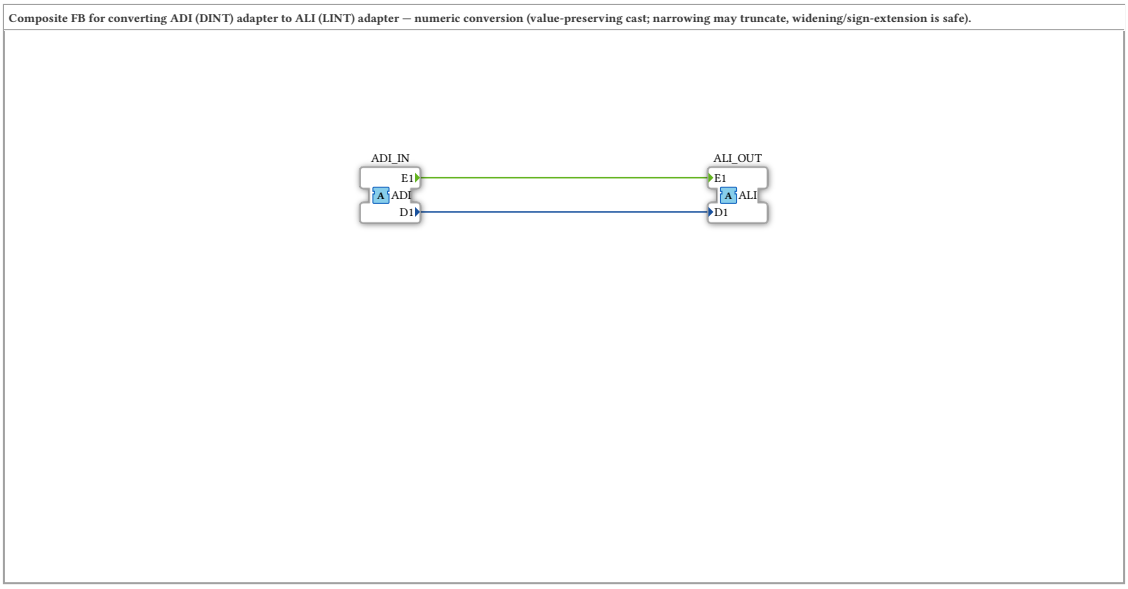
ADI_ALI

ADI_TO_ALI

DINT Adapter Input – *ADI*



ALI – LINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ADI_TO_ALI** ist ein Composite-Baustein, der einen unidirektionalen ADI-Adapter (Datentyp DINT) in einen unidirektionalen ALI-Adapter (Datentyp LINT) umwandelt. Er dient der Anpassung der Datenbreite von 32-Bit auf 64-Bit und ermöglicht so die nahtlose Integration von Komponenten unterschiedlicher Integer-Typen in Automatisierungssystemen nach IEC 61499-2. Der Baustein ist unter der Eclipse Public License 2.0 verfügbar.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

- **ADI_IN** (Socket) – Typ: `adapter::types::unidirectional::ADI` Nimmt einen DINT-Wert (32-Bit) sowie ein zugehöriges Ereignis entgegen.
- **ALI_OUT** (Plug) – Typ: `adapter::types::unidirectional::ALI` Gibt den konvertierten LINT-Wert (64-Bit) sowie das weitergeleitete Ereignis aus.

Funktionsweise

Der Baustein realisiert eine direkte Durchschleifung:

- Das Ereignis `E1` vom ADI_IN-Socket wird ohne Verzögerung an den ALI_OUT-Plug weitergegeben.
- Gleichzeitig wird der Datenwert `D1` (DINT) übertragen; dabei erfolgt eine implizite Typumwandlung von 32-Bit Integer (DINT) auf 64-Bit Integer (LINT).
- Es findet keine Zwischenspeicherung oder Verarbeitungslogik statt – jede ankommende Ereignis-Daten-Kombination wird sofort an die Ausgabeseite weitergeleitet.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Kein interner Zustand** – Der Baustein ist vollständig kombinatorisch und besitzt kein Gedächtnis.
- **Automatische Typkonvertierung** durch die Datenverbindung; der Wertebereich von DINT wird von LINT vollständig abgedeckt, sodass keine Informationsverluste auftreten.
- **Adapterbasierte Kommunikation** gemäß IEC 61499-2 ermöglicht lose Kopplung und Wiederverwendbarkeit.
- Der Baustein ist als Composite ausgeführt, d. h. sein Verhalten wird allein durch das interne FBNetzwerk (eine Ereignis- und eine Datenverbindung) definiert.

Zustandsübersicht

Der Baustein besitzt keinen Zustandsautomaten. Die Funktionalität beschränkt sich auf die direkte Weiterleitung der Adapter-Signale.

Anwendungsszenarien

- **Systemmigration** von 32-Bit auf 64-Bit Datenverarbeitung, z. B. bei der Modernisierung von Steuerungen.
- **Integration älterer Komponenten**, die DINT-Werte liefern, in neue Module, die LINT-Schnittstellen voraussetzen.
- **Adapter-Brücke** in heterogenen Netzwerken, um unterschiedliche Integer-Typen zwischen Funktionsblöcken anzupassen.

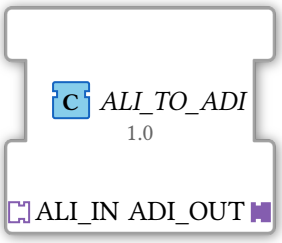
Vergleich mit ähnlichen Bausteinen

Einfache Typkonverter (z. B. `DINT_TO_LINT`) arbeiten meist auf Datenebene ohne Ereignisweitergabe. Der `ADI_TO_ALI`-Baustein hingegen kapselt Ereignis und Daten in einer Adapter-Schnittstelle und eignet sich daher besonders für modulare, ereignisgesteuerte Architekturen. Gegenüber einer manuellen Konvertierung im Netzwerk reduziert er die Komplexität und erhöht die Lesbarkeit des Systems.

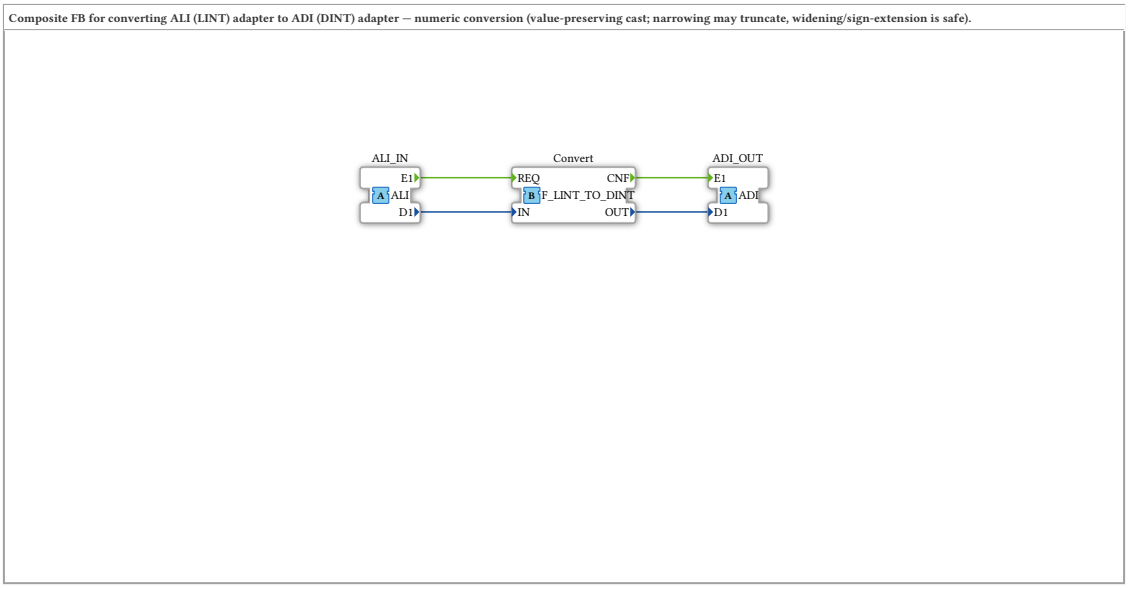
Fazit

`ADI_TO_ALI` ist ein minimaler, aber effektiver Baustein zur Anpassung der Datenbreite zwischen DINT- und LINT-Adaptern. Seine einfache Durchschleifungslogik und die automatische Typumwandlung machen ihn zu einem nützlichen Werkzeug für die Überbrückung von Integer-Typen in IEC-61499-Systemen. Die Einhaltung des Standards und die Lizenz unter EPL 2.0 gewährleisten offene Verwendbarkeit und Interoperabilität.

ALI_TO_ADI



LINT Adapter Input – *ALI* *ADI* – DINT Adapter Output



• — ○ • — ○ • — ○ •

Einleitung

Dieser Composite-Funktionsblock dient der Umwandlung eines **ALI**-Adaptertyps (LINT, 64-Bit-Integer) in einen **ADI**-Adaptertyp (DINT, 32-Bit-Integer). Er ermöglicht die nahtlose Integration von Komponenten, die unterschiedliche Adapter-Formate verwenden, indem er die Datenkonvertierung kapselt und über Ereignissteuerung synchronisiert.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die Steuerung erfolgt über den eingehenden Adapter **ALI_IN**.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Ausgabe wird über den ausgehenden Adapter **ADI_OUT** getaktet.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden. Die Eingangsdaten werden über den Adapter **ALI_IN** (vom Typ ALI) bereitgestellt.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden. Die umgewandelten Daten werden über den Adapter **ADI_OUT** (vom Typ ADI) ausgegeben.

Adapter

Name	Typ	Richtung	Beschreibung
ALI_IN	`adapter::types::unidirectional::ALI`	Socket (Eingang)	Liefert den LINT-Wert sowie ein zugehöriges Ereignis (E1).
ADI_OUT	`adapter::types::unidirectional::ADI`	Plug (Ausgang)	Gibt den konvertierten DINT-Wert sowie ein Bestätigungsereignis (E1) aus.

Funktionsweise

Der Baustein enthält im Inneren die Instanz `Convert` vom Typ `iec61131::conversion::F_LINT_TO_DINT`. Die Ereignis- und Datenverbindungen sind wie folgt geschaltet:

- Das Ereignis `ALI_IN.E1` triggert die Umwandlung über `Convert.REQ`.
- Der Datenwert `ALI_IN.D1` wird an `Convert.IN` übergeben.
- Nach erfolgreicher Konvertierung signalisiert `Convert.CNF` das Ereignis `ADI_OUT.E1`.
- Der Ergebniswert `Convert.OUT` wird an `ADI_OUT.D1` weitergeleitet.

Somit wird bei jedem Ereignis am Eingangsadapter der LINT-Wert in einen DINT-Wert umgerechnet und synchron über den Ausgangsadapter bereitgestellt.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-Baustein**: Der FB kapselt die Konvertierung in einem wiederverwendbaren Adapter-zu-Adapter-Wandler.
- **Datentypumwandlung**: Es findet eine 64-Bit- zu 32-Bit-Integer-Konvertierung statt. Bei Werten außerhalb des DINT-Bereichs (-2^{31} … $2^{31}-1$) kommt es zu einem Überlauf oder einer Begrenzung (implementierungsabhängig).
- **Ereignissynchronisation**: Die Umwandlung erfolgt streng getaktet durch die Ereigniskette; ein neuer Wert wird nur nach einem vorherigen Ereignis verarbeitet.

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine (ECC). Seine gesamte Logik wird durch den internen Funktionsblock `F_LINT_TO_DINT` realisiert, der als einfacher Umwandlungsbaustein ohne Zustandsverhalten arbeitet.

Anwendungsszenarien

- Anbindung einer Komponente, die einen 64-Bit-Zähler oder Zeitstempel (LINT) liefert, an ein System, das nur 32-Bit-DINT-Adapter unterstützt.
- Migration von Altsystemen mit 32-Bit-Arithmetik auf neuere 64-Bit-Sensoren, ohne die gesamte Adapterlandschaft zu ändern.
- Zwischenschicht in einer Hierarchie, in der unterschiedliche Integer-Breiten aufeinandertreffen.

Vergleich mit ähnlichen Bausteinen

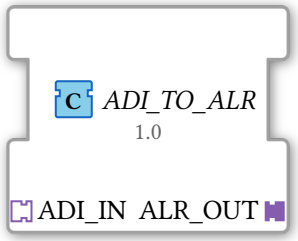
Es existieren separate Konvertierungsbausteine wie `F_LINT_TO_DINT` für direkte Datenumwandlungen. Der `ALI_TO_ADI`-Baustein hebt sich dadurch ab, dass er die Konvertierung vollständig in die Adapter-Schnittstelle integriert. Statt mehrerer diskreter Funktionsblöcke und manueller Adapterverkabelung wird die gesamte Logik in einem wiederverwendbaren Composite-Baustein zusammengefasst, der die Ereignis- und Datenströme automatisch behandelt.

Fazit

Der `ALI_TO_ADI`-Funktionsblock bietet eine saubere, gekapselte Lösung für die Umwandlung von LINT- in DINT-Adapter. Durch die Verwendung eines Composite-Bausteins reduziert er die Komplexität in der Applikation und sorgt für eine standardisierte, ereignisgesteuerte Konvertierung zwischen unterschiedlichen Integer-Breiten in der IEC 61499-Umgebung.

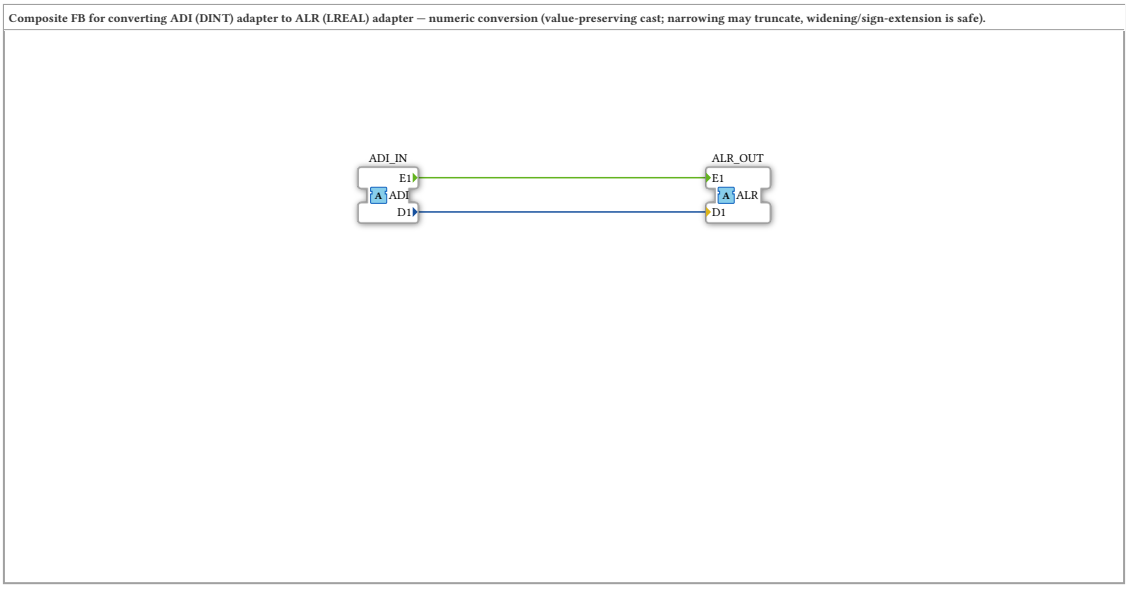
ADI_ALR

ADI_TO_ALR



DINT Adapter Input – *ADI*

ALR – LREAL Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ADI_TO_ALR** ist ein zusammengesetzter Baustein (Composite FB), der einen ADI-Adapter (Datentyp DINT) in einen ALR-Adapter (Datentyp LREAL) umwandelt. Er dient als unidirektionaler Konverter zwischen zwei standardisierten Adapter-Schnittstellen und ermöglicht die nahtlose Integration von ganzzahligen Werten in Systeme, die Gleitkommazahlen benötigen.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine dedizierten Ereignis-Eingänge vorhanden. Die Ereignisweiterleitung erfolgt über den Adapter **ADI_IN** (siehe Adapter-Abschnitt).

Ereignis-Ausgänge

- Keine dedizierten Ereignis-Ausgänge vorhanden. Das Ereignis wird über den Adapter **ALR_OUT** ausgegeben (siehe Adapter-Abschnitt).

Daten-Eingänge

- Keine dedizierten Daten-Eingänge vorhanden. Der zu konvertierende Wert wird über den Adapter **ADI_IN** bereitgestellt (siehe Adapter-Abschnitt).

Daten-Ausgänge

- Keine dedizierten Daten-Ausgänge vorhanden. Der konvertierte Wert wird über den Adapter **ALR_OUT** ausgegeben (siehe Adapter-Abschnitt).

Adapter

Name	Typ	Richtung	Beschreibung
ADI_IN	`adapter::types::unidirectional::ADI`	Socket	Eingang: DINT-Wert und Ereignis
ALR_OUT	`adapter::types::unidirectional::ALR`	Plug	Ausgang: LREAL-Wert und Ereignis

Hinweis: Die Adapter implizieren jeweils ein Ereignis- und ein Datensignal (`E1` und `D1`).

Funktionsweise

Der Baustein führt intern eine direkte Durchleitung durch:

- **Ereignis:** Das eingehende Ereignis `ADI_IN.E1` wird unverzögert auf den Ausgang `ALR_OUT.E1` übertragen.

- **Daten:** Der ganzzahlige Wert `ADI_IN.D1` (Datentyp DINT) wird ohne Skalierung oder Umrechnung in den Datentyp LREAL konvertiert und als `ALR_OUT.D1` ausgegeben. Die Typumwandlung erfolgt implizit durch die Laufzeitumgebung des IEC 61499-Systems.

Es findet keine weitere Verarbeitung (Filterung, Begrenzung, Verzögerung) statt.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-Baustein:** Die Konvertierung wird durch einen einfachen „Wire-Through“ im FBNetzwerk realisiert – keine Zustandsmaschine, keine eigenen Algorithmen.
- **Implizite Typumwandlung:** DINT ⟶ LREAL ist eine verlustfreie Umwandlung (LREAL bietet 64-Bit Genauigkeit).
- **Unidirektional:** Der Datenfluss erfolgt ausschließlich vom Socket zum Plug.
- **Wiederverwendbarkeit:** Durch die Kapselung als Adapter-Konverter kann der Baustein in beliebigen Systemen eingesetzt werden, die auf die entsprechenden Adapter-Schnittstellen angewiesen sind.

Zustandsübersicht

Der Baustein besitzt **keine Zustandsmaschine**. Er ist rein kombinatorisch (ereignisgesteuert) und führt bei jedem eingehenden Ereignis sofort die Durchleitung aus.

Anwendungsszenarien

- **Zählerwerte als Gleitkommazahlen:** Ein ganzzahliger Zähler (z.B. Stückzahlen, Impulse) soll in einer Steuerung als LREAL-Wert weiterverarbeitet werden.
- **Schnittstellenanpassung:** Systemkomponenten, die bereits mit ADI-Adaptern arbeiten, können nahtlos an Komponenten angebunden werden, die ALR-Adapter erwarten.
- **Prototypische Entwicklung:** Einfache Umwandlung ohne zusätzliche Logik, um schnell verschiedene Adapterprotokolle zu verknüpfen.

Vergleich mit ähnlichen Bausteinen

Baustein	Typumwandlung	Ereignisverhalten
ADI_TO_ALR	DINT ⟶ LREAL	Direkte Durchleitung ohne Verzögerung
`INT_TO_REAL` (4diac)	INT ⟶ REAL	Eigenständiger FB mit eigenem Ereignis-Interface
individuelle Konverter	beliebig	oft mit Zustandsmaschinen oder Parametrierung

Der vorliegende Baustein zeichnet sich durch seine **einfache Adapter-orientierte Struktur** aus, die eine direkte Kopplung in Adapter-Netzwerken ermöglicht, ohne separate Daten- und Ereignis-Kanäle definieren zu müssen.

Fazit

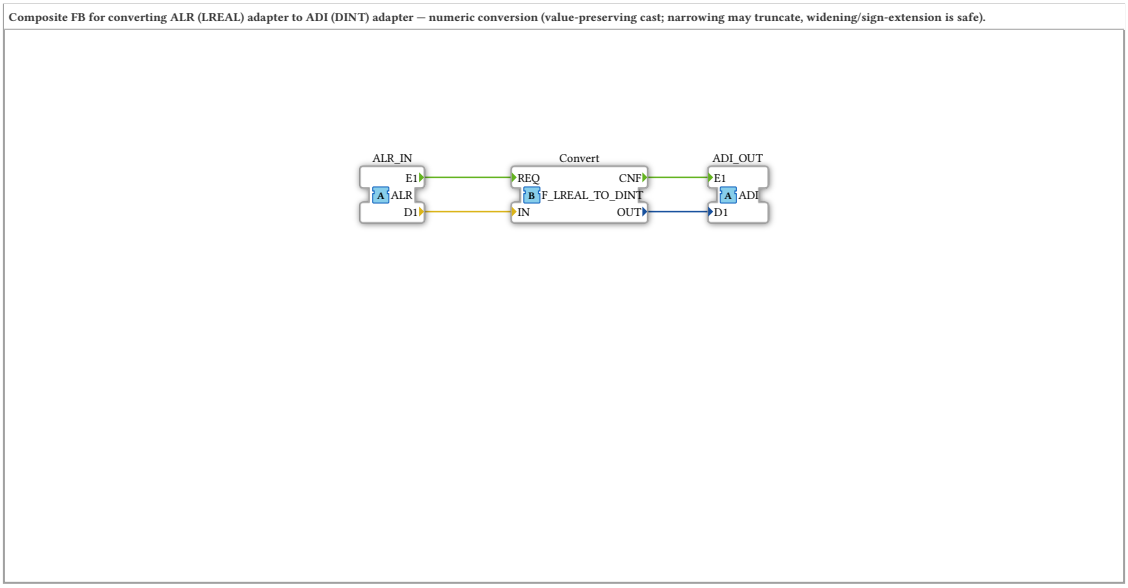
ADI_TO_ALR ist ein minimaler, aber nützlicher Konverter für die unidirektionale Übertragung von DINT-Werten in das LREAL-Format. Er ist robust, schnell und erfordert keinerlei Konfiguration. Dank seiner Composite-Struktur lässt er sich leicht in bestehende Applikationen integrieren, die auf Adapter-Schnittstellen basieren.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ALR_TO_ADI

LREAL Adapter Input – *ALR* — *ADI* – DINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ALR_TO_ADI** ist ein Composite-Baustein, der eine Konvertierung zwischen zwei unterschiedlichen Adapter-Typen ermöglicht. Er übersetzt einen **ALR-Adapter** (der Daten vom Typ LREAL liefert) in einen **ADI-Adapter** (der Daten vom Typ DINT bereitstellt). Eingesetzt wird er in Umgebungen, die auf dem IEC 61499-Standard basieren, um die Kommunikation zwischen Komponenten mit unterschiedlichen Datentyp-Schnittstellen zu vereinheitlichen.

Schnittstellenstruktur

Der Baustein besitzt keine eigenständigen Ereignis- oder Daten-Ein-/Ausgänge auf der obersten Ebene. Die gesamte Kommunikation erfolgt über die beiden Adapter-Schnittstellen.

Ereignis-Eingänge

Der FB hat keine direkten Ereignis-Eingänge. Ereignisse werden über den **ALR_IN**-Socket empfangen.

Ereignis-Ausgänge

Der FB hat keine direkten Ereignis-Ausgänge. Ereignisse werden über den **ADI_OUT**-Plug gesendet.

Daten-Eingänge

Der FB hat keine direkten Daten-Eingänge. Daten werden über den **ALR_IN**-Socket empfangen.

Daten-Ausgänge

Der FB hat keine direkten Daten-Ausgänge. Daten werden über den **ADI_OUT**-Plug gesendet.

Adapter

Adapter	Richtung	Typ	Beschreibung

ALR_IN	Socket	adapter::types::unidirectional::ALR	Liefert ein Ereignis (E1) und einen Datenwert (D1) vom Typ LREAL.
ADI_OUT	Plug	adapter::types::unidirectional::ADI	Erwartet ein Ereignis (E1) und gibt einen Datenwert (D1) vom Typ DINT aus.

Funktionsweise

Der Baustein ist als Composite realisiert und enthält intern einen einzelnen Konvertierungs-FB: `iec61131::conversion::F_LREAL_TO_DINT`. Die Verschaltung erfolgt wie folgt:

1. **Ereignisweiterleitung**: Ein am Socket **ALR_IN** eintreffendes Ereignis (E1) wird direkt an den `REQ`-Eingang des Konvertierungs-FBs weitergegeben.
2. **Datenübergabe**: Der Datenwert **D1** des **ALR_IN**-Adapters wird an den `IN`-Eingang des Konvertierungs-FBs übergeben.
3. **Konvertierung**: Der FB `F_LREAL_TO_DINT` wandelt den LREAL-Wert in einen DINT-Wert um.
4. **Rückmeldung**: Das Bestätigungsereignis (CNF) des Konvertierungs-FBs löst das Ereignis **E1** am Plug **ADI_OUT** aus.
5. **Datenausgabe**: Der konvertierte DINT-Wert vom Ausgang `OUT` des Konvertierungs-FBs wird an den Datenausgang **D1** des **ADI_OUT**-Adapters übergeben.

Somit entsteht eine synchronisierte, ereignisgesteuerte Umwandlung: Jeder Eingangsimpuls erzeugt einen Ausgangsimpuls mit dem konvertierten Datenwert.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-Baustein**: Die Implementierung nutzt einen internen Standard-FB aus der IEC 61131-Bibliothek, was Wartbarkeit und Wiederverwendung fördert.
- **Unidirektionalität**: Die Adapter sind als unidirektional deklariert, d.h. die Daten fließen nur in eine Richtung (ALR → ADI).
- **Paketstruktur**: Der Baustein ist im Paket `adapter::conversion::unidirectional` organisiert, was eine klare Einordnung in eine Bibliothek ermöglicht.
- **Keine Zustandsspeicherung**: Die Konvertierung erfolgt rein kombinatorisch mit ereignisgesteuerter Triggerung – es wird kein interner Zustand gehalten.

Zustandsübersicht

Da der FB kein eigenes Zustandsdiagramm besitzt, beschränkt sich die Zustandslogik auf die des internen Konvertierungs-FBs. Im Wesentlichen gibt es zwei Phasen:

- **Warten**: Der FB wartet auf ein Ereignis am ALR_IN-Socket.
- **Konvertieren**: Nach Empfang des Ereignisses wird die Konvertierung durchgeführt und das Ergebnis am ADI_OUT-Socket ausgegeben.

Es gibt keine Verzögerungen oder Fehlerbehandlung – die Konvertierung erfolgt sofort und erzeugt stets einen gültigen DINT-Wert (bei Überlauf oder ungültigen Werten verhält sich der Konvertierungs-FB gemäß IEC 61131).

Anwendungsszenarien

- **Systemintegration**: Ein Systemteil liefert Daten als LREAL (z.B. Gleitkomma-Messwerte), ein anderer benötigt ganzzahlige DINT-Werte (z.B. für Zählregister oder Indexberechnungen).
- **Adapter-Anpassung**: Wenn eine Komponente nur ALR-Adapter unterstützt, aber eine andere Komponente ADI-Adapter erwartet, kann dieser FB als Vermittler dienen.
- **Protokollumsetzung**: In eigenständigen Subsystemen, die über Adapter verbunden sind, ermöglicht der Baustein eine saubere Typanpassung ohne Änderung der ursprünglichen Adapterdefintion.

Vergleich mit ähnlichen Bausteinen

- **F_LREAL_TO_DINT (direkter FB)**: Dieser FB erledigt die reine Konvertierung, benötigt aber eigene Ereignis- und Datenschnittstellen und keine Adapter. `ALR_TO_ADI` kapselt diesen FB, um ihn kompatibel zu Adapter-basierten Verbindungen zu machen.
- **ALR_TO_DINT (hypothetisch)**: Ein direkter Baustein mit eigener Ein-/Ausgangsschnittstelle (Ereignis + Daten) würde keine Adapter verwenden. Der vorliegende FB ist speziell für die Verwendung in einem Adapter-Framework konzipiert.
- **Umgekehrte Konvertierung (ADI_TO_ALR)**: Analog dazu existiert vermutlich ein Baustein für die Gegenrichtung (DINT ⟶ LREAL), der ebenfalls auf demselben Konvertierungsprinzip aufbaut.

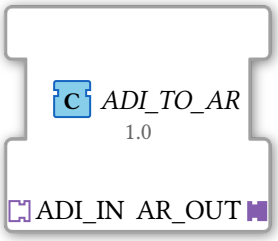
Fazit

Der Composite-FB `ALR_TO_ADI` bietet eine elegante Lösung, um zwischen Adapter-Typen mit unterschiedlichen Datentypen zu vermitteln. Durch die Wiederverwendung eines standardisierten Konvertierungsbausteins bleibt die Implementierung schlank und zuverlässig. Der FB eignet sich besonders für modulare, adapterbasierte Systeme, in denen Typkonflikte zwischen Komponenten aufgelöst werden müssen.

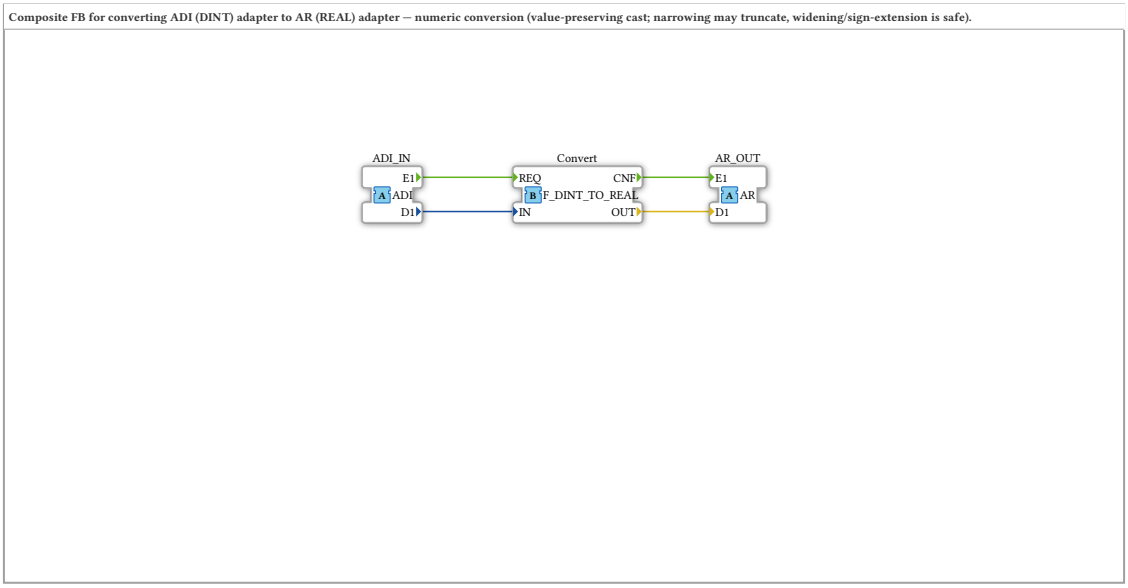
ADI_AR

ADI_TO_AR

DINT Adapter Input – *ADI*



AR – REAL Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ADI_TO_AR** ist ein Composite-Baustein, der einen DINT-Adapter (ADI) in einen REAL-Adapter (AR) umwandelt. Er dient als Vermittler zwischen Komponenten, die über unidirektionale Adapter kommunizieren, und führt dabei eine Datentypkonvertierung von Ganzzahl (DINT) auf Gleitkommazahl (REAL) durch.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkt sichtbaren Ereignis-Eingänge. Die Ereignissteuerung erfolgt indirekt über den Adapter **ADI_IN** (Ereignis E1).

Ereignis-Ausgänge

- Keine direkt sichtbaren Ereignis-Ausgänge. Die Ereignissignalisierung erfolgt indirekt über den Adapter **AR_OUT** (Ereignis E1).

Daten-Eingänge

- Keine direkt sichtbaren Daten-Eingänge. Der Eingabedatenwert wird über den Adapter **ADI_IN** (Datenpunkt D1 vom Typ DINT) bereitgestellt.

Daten-Ausgänge

- Keine direkt sichtbaren Daten-Ausgänge. Der Ausgabedatenwert wird über den Adapter **AR_OUT** (Datenpunkt D1 vom Typ REAL) ausgegeben.

Adapter

Adapter	Rolle	Typ	Beschreibung
ADI_IN	Socket	adapter::types::unidirectional::ADI	Unidirektionaler DINT-Adapter als Eingangsschnittstelle.
AR_OUT	Plug	adapter::types::unidirectional::AR	Unidirektionaler REAL-Adapter als Ausgangsschnittstelle.

Funktionsweise

Der Baustein enthält intern einen einzelnen Funktionsbaustein `F_DINT_TO_REAL` aus der IEC 61131-3 Standardbibliothek. Die Ablaufsteuerung erfolgt rein ereignisgesteuert:

1. Ein Ereignis am Eingangsadapter **ADI_IN.E1** triggert den Eingang **REQ** des internen Konvertierungsbausteins.
2. Der Datenwert **ADI_IN.D1** (DINT) wird an den Eingang **IN** des Konverters übergeben.
3. Bei Abschluss der Konvertierung signalisiert der Ausgang **CNF** des Konverters ein Ereignis.
4. Dieses Ereignis wird an den Ausgangsadapter **AR_OUT.E1** weitergeleitet.
5. Gleichzeitig wird der konvertierte REAL-Wert (Ausgang **OUT**) an den Datenpunkt **AR_OUT.D1** übergeben.

Der gesamte Vorgang erfolgt synchron und ohne Zwischenspeicherung – jede erfolgreiche Konvertierung löst sofort ein Ausgangsereignis aus.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Typkonvertierung nach IEC 61131-3**: Der interne Baustein `F_DINT_TO_REAL` garantiert eine standardkonforme Umwandlung von 32-Bit-Ganzzahlen in Gleitkommazahlen (einfache Genauigkeit).
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional ausgelegt. Eine Rückwärtskommunikation ist nicht vorgesehen, was die Implementierung vereinfacht und den Datenfluss eindeutig festlegt.
- **Composite-Struktur**: Der Baustein besteht aus einem Netzwerk, das vollständig im FB selbst gekapselt ist. Er eignet sich daher besonders gut für modulare, wiederverwendbare Schnittstellen.
- **Keine Zustände**: Es existiert kein Zustandsautomat – die Funktionalität ist rein kombinatorisch mit einem ereignisgesteuerten Trigger.

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Der Ablauf lässt sich als einfacher Schritt beschreiben:

Schritt	Aktion
1	Warten auf Ereignis an **ADI_IN.E1**
2	Konvertierung des Datenwerts von DINT nach REAL
3	Ausgabe des konvertierten Werts an **AR_OUT.D1** und Ereignis an **AR_OUT.E1**
4	Rückkehr zu Schritt 1

Anwendungsszenarien

- **Sensorintegration**: Ein Sensor liefert ganzzahlige Messwerte (z. B. Drehzahl, Zählerstand) über einen ADI-Adapter. Für eine nachgelagerte Regelung oder Visualisierung, die REAL-Werte erwartet, wird dieser Baustein als Konverter eingesetzt.
- **Schnittstellenanpassung**: In heterogenen Steuerungssystemen, in denen verschiedene Komponenten unterschiedliche Datentypen verwenden, ermöglicht der Baustein eine nahtlose Integration ohne Änderung der vorhandenen Adapterdefinitionen.
- **Typsichere Datenverarbeitung**: Durch die explizite Konvertierung werden Laufzeitfehler vermieden, die bei impliziten Typumwandlungen in IEC 61499 auftreten können.

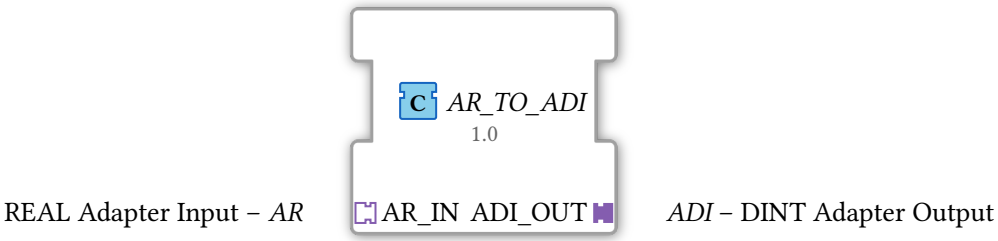
Vergleich mit ähnlichen Bausteinen

- **F_DINT_TO_REAL**: Ein direkter Aufruf dieses Standardbausteins würde denselben Zweck erfüllen, erfordert jedoch eine manuelle Verschaltung von Ereignis- und Datenverbindungen. Der Adapter-basierte Ansatz vereinfacht die Wiederverwendung und verbessert die Lesbarkeit des Netzwerks.
- **ADI_TO_AR** im Gegensatz zu anderen Konvertierungsbausteinen (z. B. REAL_TO_DINT): Dieser Baustein ist speziell für die unidirektionale Adapter-Kommunikation ausgelegt und bietet keine weiteren Konvertierungsrichtungen an.
- **Composite-Bausteine ohne Adapter**: Andere Implementierungen nutzen direkte Daten- und Ereignis-Ports, was die Flexibilität beim Austausch von Komponenten einschränkt. Die Adapter-Schnittstelle erlaubt ein einfaches Plug-and-Play in vordefinierten Architekturen.

Fazit

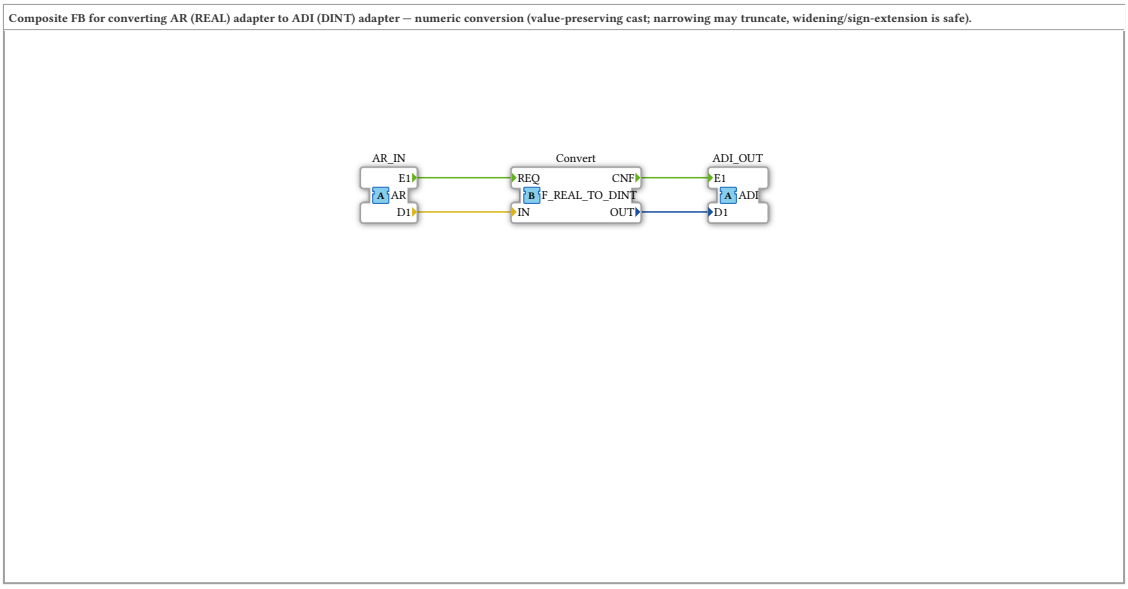
Der **ADI_TO_AR**-Funktionsblock stellt eine elegante und standardkonforme Lösung zur Umwandlung von DINT- in REAL-Werte dar, die vollständig in die Adapter-basierte Kommunikationsstruktur der IEC 61499 eingebettet ist. Durch die Kapselung des Konvertierungsvorgangs in einem wiederverwendbaren Composite-Baustein wird die Entwicklung modularer, typsicherer und gut dokumentierter Automatisierungssysteme unterstützt.

AR_TO_ADI



REAL Adapter Input – *AR*

ADI – DINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **AR_TO_ADI** ist ein Composite-Baustein, der einen REAL-Adapter (AR) in einen DINT-Adapter (ADI) umwandelt. Er wird verwendet, um zwischen Schnittstellen mit unterschiedlichen Datentypen zu vermitteln, wobei die Konvertierung ereignisgesteuert abläuft.

Schnittstellenstruktur

Die Schnittstelle des Bausteins besteht ausschließlich aus Adaptern, die sowohl Ereignisse als auch Daten bündeln. Es sind keine separaten Ein- oder Ausgänge vorhanden.

Ereignis-Eingänge

- **AR_IN.E1** – Ereigniseingang über den AR-Adapter Löst die Konvertierung des anliegenden REAL-Wertes aus.

Ereignis-Ausgänge

- **ADI_OUT.E1** – Ereignisausgang über den ADI-Adapter Wird nach erfolgreicher Konvertierung aktiviert.

Daten-Eingänge

- **AR_IN.D1** (Datentyp REAL) – Der zu konvertierende Gleitkommawert.

Daten-Ausgänge

- **ADI_OUT.D1** (Datentyp DINT) – Der konvertierte Ganzzahlwert.

Adapter

Adapter	Name	Richtung	Beschreibung
AR	AR_IN	Socket (Eingang)	Unidirektionaler REAL-Adapter, der das Eingangssignal bereitstellt.
ADI	ADI_OUT	Plug (Ausgang)	Unidirektionaler DINT-Adapter, der das konvertierte Signal weitergibt.

Funktionsweise

Der Baustein ist als Composite-FB realisiert. Im Inneren befindet sich eine Instanz des IEC-61131-Standardbausteins `F_REAL_TO_DINT`. Sobald am Ereigniseingang **AR_IN.E1** ein Ereignis eintrifft, wird der über **AR_IN.D1** anliegende REAL-Wert an den internen Konverter übergeben. Nach abgeschlossener Konvertierung wird das Ergebnis an **ADI_OUT.D1** weitergeleitet und das Ereignis **ADI_OUT.E1** ausgelöst.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- Der Baustein verwendet den Datentyp-Konverter `F_REAL_TO_DINT` aus der IEC-61131-Bibliothek.
- Es handelt sich um einen Composite-FB, der keine eigene Zustandslogik besitzt, sondern die Funktionalität durch den internen Baustein abbildet.
- Die Schnittstelle ist rein adapterbasiert, sodass eine einfache Einbindung in bestehende Adapter-Netzwerke möglich ist.

Zustandsübersicht

Da der FB keine Zustandsmaschine enthält, ergibt sich folgender einfacher Ablauf:

1. **Warten** – Der FB ist bereit, ein Ereignis auf **AR_IN.E1** zu empfangen.
2. **Konvertierung** – Sobald ein Ereignis eintrifft, wird der REAL-Wert in DINT umgewandelt.
3. **Ausgabe** – Nach Abschluss der Konvertierung wird das Ergebnis ausgegeben und das Ausgangsereignis gesendet.Danach kehrt der FB in den Wartezustand zurück.

Anwendungsszenarien

- Anpassung einer Sensorik, die REAL-Werte liefert, an eine Steuerung, die DINT-Werte erwartet.
- Brücke zwischen zwei Adapter-basierten Komponenten mit unterschiedlichen Datentypen in einem IEC-61499-System.
- Nutzung in Datenvorverarbeitungsketten, bei denen Gleitkommazahlen in Ganzzahlen umgerechnet werden müssen.

Vergleich mit ähnlichen Bausteinen

- **`F_REAL_TO_DINT`** – Dieser Baustein führt die eigentliche Konvertierung durch, erwartet aber separate Ereignis- und Datenanschlüsse. **AR_TO_ADI** kapselt diese Funktionalität in einer adapterbasierten Schnittstelle und vereinfacht so die Einbindung in Adapter-Netzwerke.
- **Andere Adapter-Konverter** – Ähnliche Bausteine existieren für andere Datentypkombinationen (z.B. AR_TO_AINT, ADI_TO_AR). Alle folgen dem gleichen Muster: Verwendung eines internen Konvertierungs-FBs, eingebettet in eine einheitliche Adapter-Hülle.

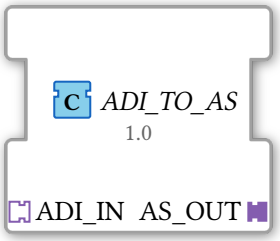
Fazit

AR_TO_ADI ist ein kompakter, wiederverwendbarer Composite-FB, der eine saubere Umwandlung von REAL- auf DINT-Adapter ermöglicht. Dank seiner einfachen, ereignisgesteuerten Struktur und der Nutzung standardisierter Konvertierungsbausteine eignet er sich hervorragend für Adapter-basierte Entwürfe in industriellen Steuerungssystemen.

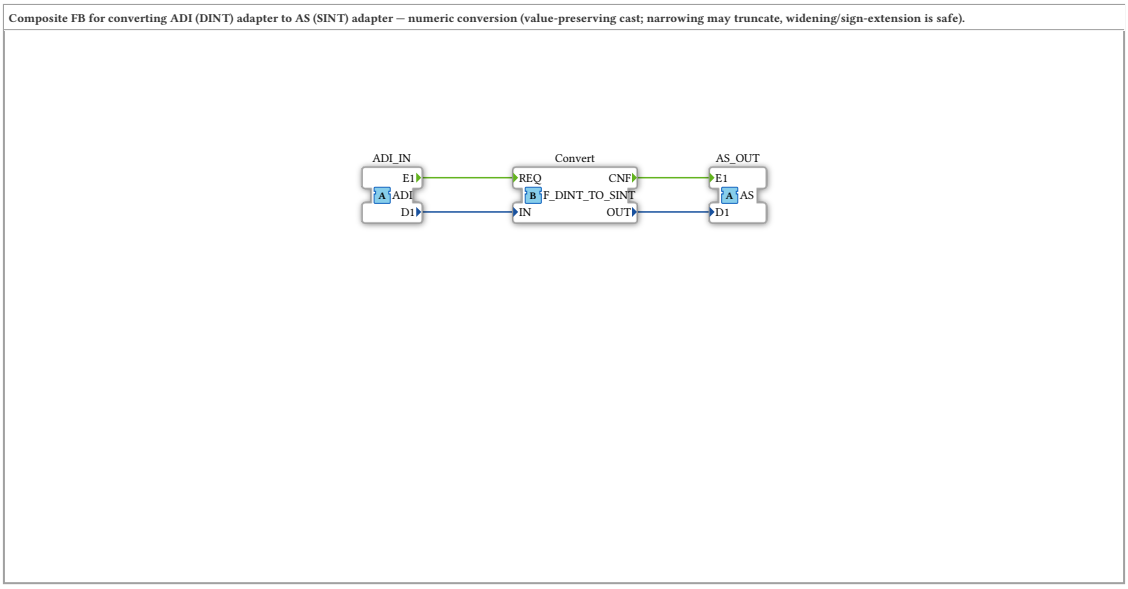
ADI_AS

ADI_TO_AS

DINT Adapter Input – *ADI*



AS – SINT Adapter Output



• — ○ • — ○ • — ○ •

Einleitung

Der Composite-Funktionsblock **ADI_TO_AS** dient der Konvertierung eines **DINT-Adapter-Signals** (Typ `ADI`) in ein **SINT-Adapter-Signal** (Typ `AS`). Er wird eingesetzt, wenn ein Daten- und Ereignisaustausch über unidirektionale Adapter erfolgen soll und eine Anpassung des Datentyps von DINT (32-Bit Signed Integer) auf SINT (8-Bit Signed Integer) erforderlich ist.

Schnittstellenstruktur

Ereignis-Eingänge

Keine separaten Ereignis-Eingänge – die Ereignisansteuerung erfolgt über den integrierten **ADI-IN-Adapter** (Socket).

Ereignis-Ausgänge

Keine separaten Ereignis-Ausgänge – die Ereignisweiterleitung erfolgt über den integrierten **AS-OUT-Adapter** (Plug).

Daten-Eingänge

Keine separaten Daten-Eingänge – der Datenwert wird über den **ADI-IN-Adapter** bereitgestellt.

Daten-Ausgänge

Keine separaten Daten-Ausgänge – der konvertierte Wert wird über den **AS-OUT-Adapter** ausgegeben.

Adapter

Adapter	Richtung	Typ	Kommentar
ADI_IN	Socket (Eingang)	`adapter::types::unidirectional::ADI`	DINT-Adapter-Eingang
AS_OUT	Plug (Ausgang)	`adapter::types::unidirectional::AS`	SINT-Adapter-Ausgang

Der **ADI-IN-Adapter** stellt eine Ereignisschnittstelle `E1` und eine Datenschnittstelle `D1` (Typ DINT) bereit. Der **AS-OUT-Adapter** bietet entsprechend `E1` und `D1` (Typ SINT).

Funktionsweise

Im Inneren des Composite-Bausteins wird ein einzelner Funktionsblock `Convert` vom Typ `iec61131::conversion::F_DINT_TO_SINT` verwendet. Die Signalverbindungen sind:

1. Ein eingehendes Ereignis am **ADI-IN-Adapter** (`E1`) triggert den **Convert-FB** über dessen Ereigniseingang `REQ`.

2. Gleichzeitig wird der DINT-Wert vom Adapter-Datenport `ADI_IN.D1` an den **Convert-Eingang** `IN` übergeben.
3. Nach Abschluss der Konvertierung erzeugt **Convert** ein Ereignis an seinem Ausgang `CNF`.
4. Dieses Ereignis wird direkt an den **AS-OUT-Adapter** (`E1`) weitergeleitet.
5. Der konvertierte SINT-Wert liegt am **Convert-Ausgang** `OUT` an und wird an den Adapter-Datenport `AS_OUT.D1` übergeben.

Damit ist eine vollständige, ereignisgesteuerte Umwandlung von DINT nach SINT über die Adapter-Schnittstellen realisiert.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Lizenz und Herkunft:** Der Baustein unterliegt der **Eclipse Public License 2.0** und stammt von der **HR Agrartechnik GmbH** (Version 1.0, Autor: Franz Höpfinger, Stand: 2026-02-17).
- **Abhängigkeit:** Er nutzt den Bibliotheksbaustein `iec61131::conversion::F_DINT_TO_SINT` aus dem IEC-61131-Konvertierungspaket.
- **Composite-Struktur:** Es werden keine eigenen Zustände oder Algorithmen definiert – die gesamte Logik wird durch das innere Netzwerk ausgeführt.
- **Paketzuordnung:** Der Baustein ist im Paket `adapter::conversion::unidirectional` organisiert.

Zustandsübersicht

Als Composite-Funktionsblock besitzt **ADI_TO_AS** keine eigene Zustandsmaschine. Das Verhalten ergibt sich ausschließlich aus der Abarbeitung des eingebetteten FBs `F_DINT_TO_SINT` und den fest verdrahteten Verbindungen.

Anwendungsszenarien

- **Adapter-Typkonvertierung:** Wenn eine Komponente (z. B. ein Sensor oder Aktor) über einen unidirektionalen DINT-Adapter kommuniziert, das Zielsystem jedoch einen SINT-Adapter erwartet.
- **Datenbereichsreduktion:** Anwendungen, bei denen ein 32-Bit-Wert in einen 8-Bit-Wert umgewandelt werden muss, z. B. zur Ansteuerung einfacher Peripherie mit begrenzter Auflösung.
- **Schnittstellenanpassung:** Einbindung in eine Toolchain, die auf das unidirektionale Adapter-Konzept von 4diac aufbaut und eine saubere Trennung von Ereignis- und Datenflüssen erfordert.

Vergleich mit ähnlichen Bausteinen

Es existieren weitere Adapter-Konvertierungsbausteine wie z. B. `ADI_TO_AI` (DINT → INT) oder `ADI_TO_ADI` (identische Durchleitung). Der **ADI_TO_AS** unterscheidet sich durch die spezifische Zieltypumsetzung auf SINT. Gemeinsam ist allen die einheitliche Adapter-Schnittstelle und die ereignisgesteuerte Verarbeitung mittels eines enthaltenen Konvertierungs-FBs.

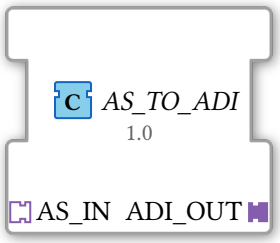
Fazit

Der **ADI_TO_AS** ist ein kompakter, fertig vorgefertigter Composite-Baustein, der die Umwandlung eines DINT-Adapter-Signals in ein SINT-Adapter-Signal zuverlässig und einfach ermöglicht. Er ist ideal für alle Anwendungen, die eine saubere Typkonvertierung im Rahmen des 4diac-Adapterkonzepts benötigen, ohne dass separate Konvertierungslogik manuell implementiert werden muss. Dank der Lizenz unter EPL-2.0 kann er in eigenen Projekten frei genutzt und angepasst werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

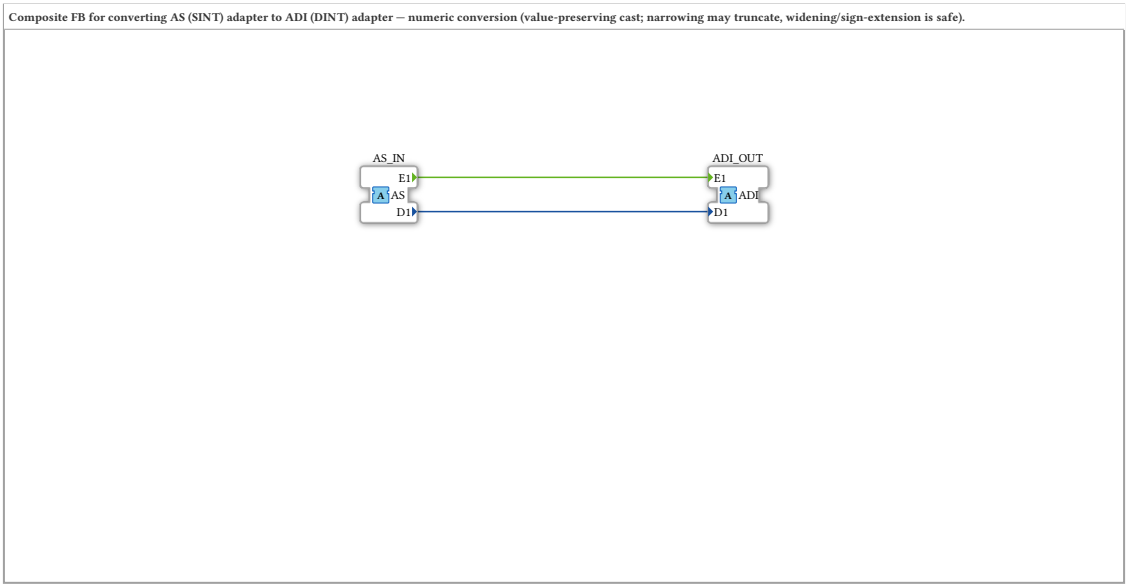
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AS_TO_ADI



SINT Adapter Input – *AS*

ADI – DINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AS_TO_ADI** wandelt einen SINT-Adapter (AS) in einen DINT-Adapter (ADI) um. Es handelt sich um einen Composite-Baustein, der eine unidirektionale, typkonvertierende Verbindung zwischen den beiden unterschiedlichen Datenadaptern herstellt.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Typ	Richtung	Name	Beschreibung
Socket	Eingang	AS_IN	Adapter vom Typ `AS` (SINT, 8-Bit signed integer) – nimmt das SINT-Signal auf.
Plug	Ausgang	ADI_OUT	Adapter vom Typ `ADI` (DINT, 32-Bit signed integer) – gibt das umgewandelte DINT-Signal weiter.

Funktionsweise

Der Baustein verknüpft intern die Ereignis- und Datenleitungen der beiden Adapter direkt miteinander:

- Das eingehende Ereignis `E1` des Sockets `AS_IN` wird unverändert auf den Ausgangsplug `ADI_OUT` an dessen Ereignis `E1` übertragen.
- Der Datenwert `D1` aus dem SINT-Adapter (`AS_IN.D1`) wird auf den Datenport `D1` des DINT-Adapters (`ADI_OUT.D1`) weitergeleitet.

Da die Datentypen unterschiedlich sind (SINT → DINT), erfolgt eine implizite Werterhaltung: Der SINT-Wert (Wertebereich –128 … 127) wird in einen vorzeichenbehafteten 32-Bit-Wert umkopiert (Vorzeichenerweiterung). Der Baustein selbst führt keine eigene Konvertierungslogik aus, sondern vertraut auf die durch die Adapter-Schnittstellen definierte Typkonvertierung.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-Baustein**: Der FB besitzt kein eigenes Verhaltensmodell (ECC), sondern nur ein internes FBNetzwerk, das die Adapter verbindet.
- **Typkonvertierung**: Die Konvertierung von SINT auf DINT erfolgt systemseitig durch die Laufzeitumgebung (Typkompatibilität bei Datenverbindungen).
- **Paket**: Der FB ist im Paket `adapter::conversion::unidirectional` eingeordnet.
- **Vollständig vorgegeben**: Die interne Verdrahtung ist fest; die Adapter müssen jeweils über einen Ereignis- und einen Datenport (`E1`, `D1`) verfügen.

Zustandsübersicht

Da es sich um einen Composite-Baustein ohne eigene Zustandsmaschine handelt, gibt es keine Zustände oder Schritte. Der FB arbeitet rein datenflusssynchron – jedes eingehende Ereignis löst sofort die Weiterleitung aus.

Anwendungsszenarien

- **Sensoranbindung**: Ein SINT-Adapter (z. B. von einem 8-Bit-Sensor) soll an eine Steuerung angeschlossen werden, die als Schnittstelle einen DINT-Adapter erwartet.
- **Systemintegration**: In Anlagen, in denen unterschiedliche Genauigkeitsstufen (8-Bit vs. 32-Bit) aufeinandertreffen, erlaubt der Baustein eine einfache, standardisierte Umwandlung.
- **Protokollanpassung**: Als Teil einer Adapter-Kaskade, um von einem einfachen Daten-Schnittstellenstandard auf einen leistungsfähigeren zu wechseln.

Vergleich mit ähnlichen Bausteinen

- **AS_TO_UDI** – Konvertiert SINT nach UINT (vorzeichenloser 16-Bit-Integer), andere Wertebereichsinterpretation.
- **ADI_TO_AS** – Rückwärtskonvertierung von DINT nach SINT (mit möglichen Überlaufrisiken).
- **DIRECT_LINK** – Reine Durchleitung ohne Typkonvertierung, nur bei identischen Adaptertypen verwendbar.

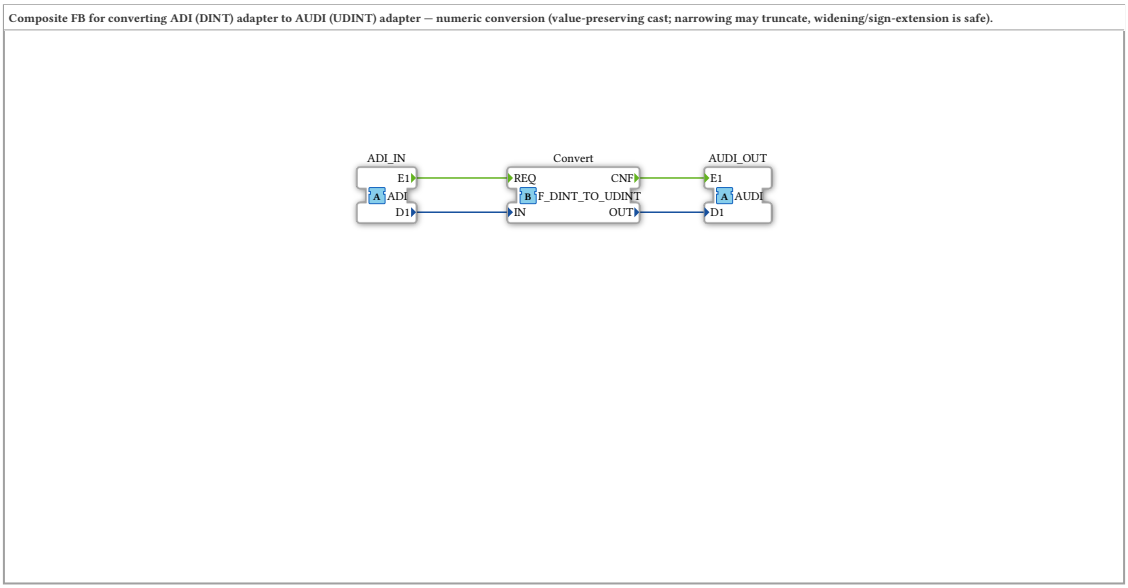
Im Gegensatz zu universellen Konvertern (z. B. mittels `MATH`-Bausteinen) bietet `AS_TO_ADI` eine spezialisierte, kompakte Lösung ohne zusätzliche Parametrierung.

Fazit

Der Funktionsblock **AS_TO_ADI** realisiert eine schnelle und zuverlässige Typanpassung zwischen SINT- und DINT-Adaptern. Durch die einfache, composite-basierte Struktur eignet er sich ideal für standardisierte Adapter-Wandlungen in der Automatisierungstechnik, bei denen eine unidirektionale 8-Bit → 32-Bit-Konvertierung benötigt wird.

ADI_AUDI

ADI_TO_AUDI



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock ADI_TO_AUDI dient der Konvertierung eines Adapters vom Typ ADI (DINT) in einen Adapter vom Typ AUDI (UDINT). Es handelt sich um einen zusammengesetzten Baustein (Composite FB), der einen internen Konvertierungsbaustein verwendet, um die Datentypen von signed 32-bit Integer (DINT) nach unsigned 32-bit Integer (UDINT) zu wandeln. Der Baustein ermöglicht die nahtlose Integration von Komponenten, die unterschiedliche Adaptertypen verwenden.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine direkten Ereignis-Eingänge. Ereignisse werden über den eingehenden Adapter **ADI_IN** bereitgestellt. Der Adapter ADI definiert einen Ereignis-Ausgang E1, der mit dem Ereignis-Eingang REQ des internen Konvertierungsbausteins verbunden ist.

Bezeichnung	Datentyp	Beschreibung
(über ADI_IN.E1)	EVENT	Auslöser für die Konvertierung

Ereignis-Ausgänge

Der Baustein besitzt keine direkten Ereignis-Ausgänge. Ereignisse werden über den ausgehenden Adapter **AUDI_OUT** weitergegeben. Der interne Konvertierungsbaustein signalisiert die Fertigstellung über CNF, das mit dem Ereignis-Eingang E1 des Adapters AUDI verbunden ist.

Bezeichnung	Datentyp	Beschreibung
(über AUDI_OUT.E1)	EVENT	Bestätigung der abgeschlossenen Konvertierung

Daten-Eingänge

Der Baustein besitzt keine direkten Daten-Eingänge. Der zu konvertierende DINT-Wert wird über den eingehenden Adapter **ADI_IN** (Ausgang D1) bereitgestellt.

Bezeichnung	Datentyp	Beschreibung
(über ADI_IN.D1)	DINT	Der als DINT vorliegende Wert, der in UDINT konvertiert werden soll

Daten-Ausgänge

Der Baustein besitzt keine direkten Daten-Ausgänge. Der konvertierte UDINT-Wert wird über den ausgehenden Adapter **AUDI_OUT** (Eingang D1) ausgegeben.

Bezeichnung	Datentyp	Beschreibung
(über AUDI_OUT.D1)	UDINT	Das Ergebnis der Konvertierung als UDINT

Adapter

Der Baustein stellt zwei Adapter-Schnittstellen bereit:

Name	Richtung	Typ	Beschreibung
ADI_IN	Socket	adi (unidirectional)	Eingangsadapter – liefert DINT-Daten und ein Ereignis zur Auslösung
AUDI_OUT	Plug	audi (unidirectional)	Ausgangsadapter – gibt konvertierte UDINT-Daten und ein Bestätigungsereignis aus

Die Adapter sind unidirektional, d.h. sie übertragen Daten und Ereignisse nur in eine Richtung.

Funktionsweise

Der Funktionsblock arbeitet als reine Hülle (Composite) und enthält einen einzelnen internen Baustein vom Typ `iec61131::conversion::F_DINT_TO_UDINT`. Sobald am Eingangsadapter ADI_IN ein Ereignis an E1 eintrifft, wird der interne Baustein über seinen REQ-Eingang getriggert. Dieser liest den an ADI_IN.D1 anliegenden DINT-Wert, konvertiert ihn in einen UDINT-Wert und legt das Ergebnis an seinem Ausgang OUT an. Gleichzeitig erzeugt er ein Bestätigungsereignis an CNF, das über den Ausgangsadapter AUDI_OUT.E1 weitergeleitet wird. Der konvertierte Wert steht dann am AUDI_OUT.D1 zur Verfügung.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- Der Baustein ist als Composite FB realisiert und enthält selbst keine eigene Logik. Die gesamte Funktionalität wird durch den internen Konvertierungsbaustein bereitgestellt.
- Die Adapter sind vom Typ „unidirectional“, d.h. die Daten- und Ereignisflüsse sind gerichtet.
- Der Baustein ist Teil der Adapter-Konvertierungsbibliothek und erleichtert die Anbindung von Komponenten mit unterschiedlichen Adapter-Datentypen.
- Die Datentypkonvertierung von DINT nach UDINT erfolgt typisiert; ein Überlauf bei negativen Werten ist zu beachten (z. B. wird −1 zu 4294967295).

Zustandsübersicht

Der Funktionsblock besitzt keine eigene Zustandsmaschine. Sein Verhalten ergibt sich ausschließlich aus dem Zusammenspiel des internen Bausteins `F_DINT_TO_UDINT` und der Ereignisverbindungen. Der interne Baustein hat eine einfache Zustandslogik: Im Ruhezustand wartet er auf ein REQ-Ereignis, führt dann die Konvertierung aus und sendet CNF. Es gibt keine weiteren Zustände.

Anwendungsszenarien

- **Adapterwechsel**: Wenn in einer Anwendung ein DINT-basierter Adapter (ADI) durch einen UDINT-basierten Adapter (AUDI) ersetzt werden muss, ohne die umgebende Logik zu ändern.
- **Systemintegration**: Verbindung von Komponenten, die unterschiedliche Integer-Typen verwenden (z. B. ein Sensor liefert DINT, ein Aktor erwartet UDINT).

- **Protokollanpassung**: In heterogenen Steuerungssystemen, wo Adaptertypen nicht direkt kompatibel sind.

Vergleich mit ähnlichen Bausteinen

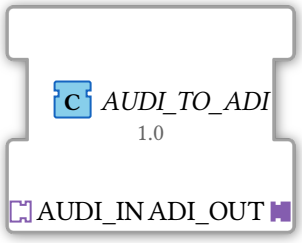
- **ADI_TO_AUDI** (dieser Baustein): Konvertiert DINT-Adapter zu UDINT-Adapter.
- **AUDI_TO_ADI**: Ein hypothetischer Gegenstück, das UDINT in DINT wandeln würde (existiert nicht in der gegebenen XML, aber denkbar).
- **Direkte Konvertierung mit F_DINT_TO_UDINT**: Statt den Adapter zu verwenden, könnte man den Konvertierungsbaustein direkt einsetzen, müsste dann aber die Adapterstruktur manuell nachbilden. Der ADI_TO_AUDI kapselt diese Anpassung und bietet eine saubere Schnittstelle.

Fazit

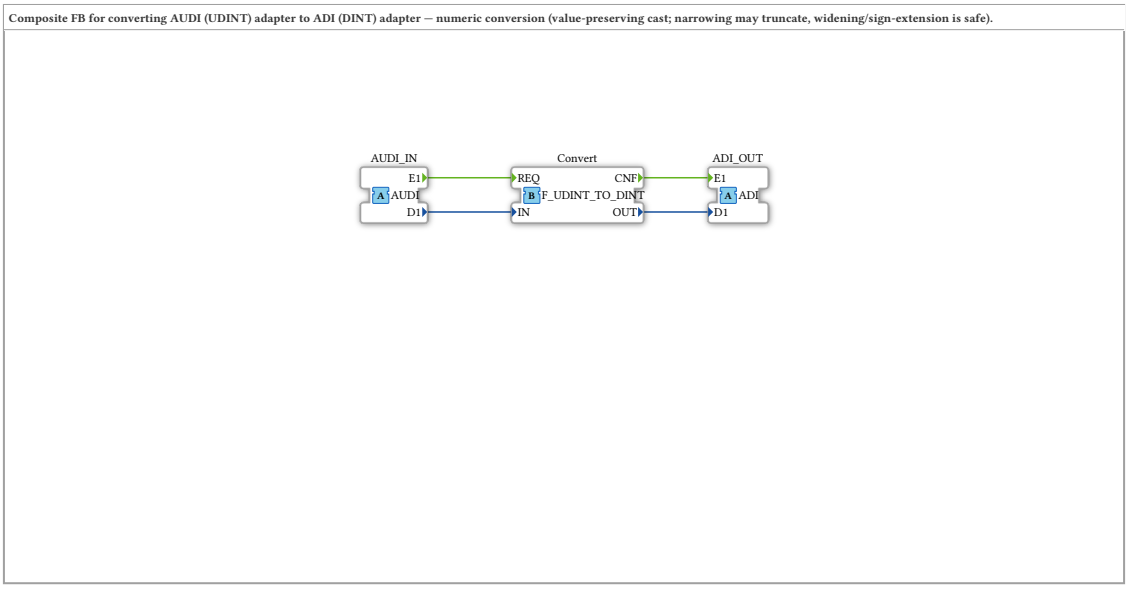
Der Funktionsblock ADI_TO_AUDI stellt eine einfache und effektive Lösung zur Konvertierung von DINT- auf UDINT-Adapter dar. Als Composite FB nutzt er einen bewährten Standard-Konvertierungsbaustein und bietet über seine Adapter eine klare, unidirektionale Schnittstelle. Er eignet sich besonders für die Systemintegration und Adapteranpassung in IEC 61499-basierten Automatisierungslösungen.

AUDI_TO_ADI

UDINT Adapter Input – *AUDI*



ADI – DINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUDI_TO_ADI** ist ein Composite-Baustein, der einen unidirektionalen AUDI-Adapter (UDINT) in einen ADI-Adapter (DINT) umwandelt. Er dient als Brücke zwischen Komponenten, die unterschiedliche Datentypen über Adapter-Schnittstellen verwenden. Die Konvertierung erfolgt durch den inneren Baustein `iec61131::conversion::F_UDINT_TO_DINT`.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
AUDI_IN.E1	EVENT	Ereignis-Eingang über den AUDI-Adapter; löst die Konvertierung aus

Ereignis-Ausgänge

Name	Typ	Kommentar
ADI_OUT.E1	EVENT	Ereignis-Ausgang über den ADI-Adapter; signalisiert abgeschlossene Konvertierung

Daten-Eingänge

Name	Typ	Kommentar
AUDI_IN.D1	UDINT	Unsignierter 32-Bit-Wert, der in einen vorzeichenbehafteten DINT gewandelt wird

Daten-Ausgänge

Name	Typ	Kommentar
ADI_OUT.D1	DINT	Konvertierter vorzeichenbehafteter 32-Bit-Wert

Adapter

Name	Typ	Richtung	Kommentar
AUDI_IN	adaper::types::unidirectional::AUDI	Socket (Eingang)	UDINT-Adapter-Schnittstelle
ADI_OUT	adaper::types::unidirectional::ADI	Plug (Ausgang)	DINT-Adapter-Schnittstelle

Funktionsweise

Der Baustein empfängt über den Socket `AUDI_IN` einen UDINT-Wert zusammen mit einem Ereignis am Eingang `E1`. Dieses Ereignis löst die Konvertierung des anliegenden UDINT-Wertes in einen DINT-Wert aus. Die Konvertierung wird intern durch den Funktionsbaustein `F_UDINT_TO_DINT` realisiert. Nach erfolgreicher Wandlung wird der konvertierte DINT-Wert über den Plug `ADI_OUT` bereitgestellt und ein Ereignis am Ausgang `E1` ausgesendet.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- Der Baustein ist ein Composite-FB und enthält keinen eigenen Algorithmus – die gesamte Logik wird an den eingebetteten `F_UDINT_TO_DINT` delegiert.
- Die Konvertierung folgt der IEC-61131-3-Definition von `UDINT_TO_DINT`, d. h. Werte außerhalb des DINT-Bereichs (−2 147 483 648 … 2 147 483 647) werden auf den nächstliegenden gültigen Wert begrenzt (Clamping).
- Die Ereignisweiterleitung erfolgt streng kausal: erst nach Abschluss der Konvertierung wird der Ausgangs-Event aktiviert.

Zustandsübersicht

Der Baustein besitzt keine eigenen internen Zustände. Sein Verhalten ist rein reaktiv: Auf ein Ereignis am Eingang folgt stets ein Ereignis am Ausgang – sofern die Daten gültig sind. Eine dedizierte Zustandsmaschine existiert nicht.

Anwendungsszenarien

- Anbindung eines UDINT-liefernden Sensors (z. B. Zähler oder Positionswert) an einen Aktor, der einen DINT-Adapter erwartet.
- Integration von SPS-Komponenten, die unterschiedliche Adapter-Datentypen verwenden, ohne dass der Anwender die Konvertierung manuell programmieren muss.
- Einheitliche Adapter-Schnittstelle in einer Steuerungsarchitektur, die teilweise mit signierten und teils mit unsignierten Werten arbeitet.

Vergleich mit ähnlichen Bausteinen

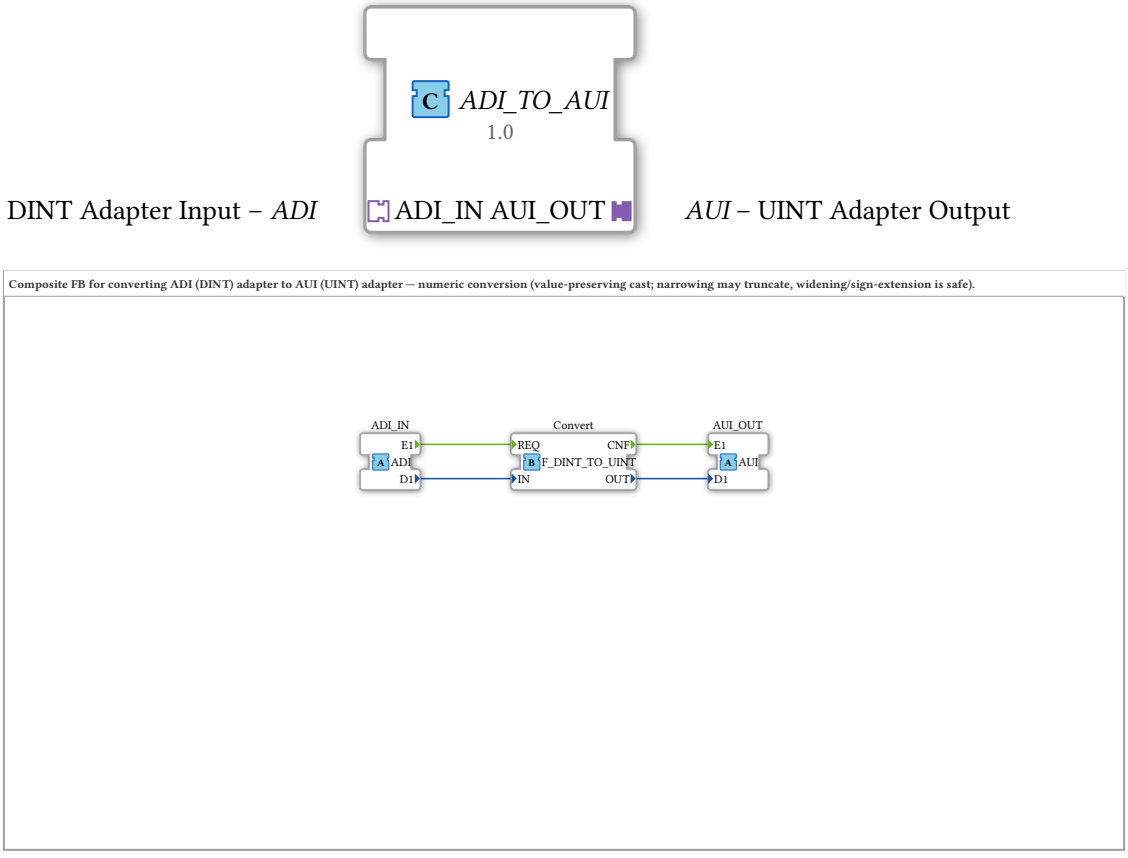
- **DINT_TO_AUDI**: Führt die umgekehrte Konvertierung (DINT ⟶ UDINT) durch.
- **F_UDINT_TO_DINT**: Direkte Verwendung des Konvertierungs-FBs ohne Adapter-Kapselung. Der vorliegende Baustein kapselt die Datentypwandlung in einen Adapter-Kontext und vereinfacht so die Verschaltung von Komponenten, die auf Adapter-Schnittstellen angewiesen sind.

Fazit

AUDI_TO_ADI bietet eine saubere, kompakte Lösung zur Umwandlung von UDINT-Adapter-Daten in DINT-Adapter-Daten. Durch die Verwendung eines standardisierten Konvertierungs-FBs ist die Funktion zuverlässig und leicht nachvollziehbar. Der Composite-Ansatz reduziert die Komplexität im übergeordneten System und erleichtert die Wiederverwendbarkeit der Adapter-Logik.

ADI_AUI

ADI_TO_AUI



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ADI_TO_AUI** ist ein Composite-FB (zusammengesetzter Baustein), der eine Schnittstellenkonvertierung zwischen dem ADI-Adapter (DINT, 32-Bit-Ganzzahl) und dem AUI-Adapter (UINT, 16-Bit-Ganzzahl) realisiert. Er ermöglicht die unidirektionale Übertragung eines Wertes von einem

DINT-Adapter an einen UINT-Adapter unter Verwendung eines standardisierten IEC 61131-Konvertierungsbausteins.

Schnittstellenstruktur

Ereignis-Eingänge
Keine – die Ereignissteuerung erfolgt ausschließlich über die Adapter.

Ereignis-Ausgänge
Keine.

Daten-Eingänge
Keine – alle Daten werden über den Socket-Adapter bereitgestellt.

Daten-Ausgänge
Keine – alle Daten werden über den Plug-Adapter ausgegeben.

Adapter

Typ	Name	Richtung	Beschreibung
Socket	`ADI_IN`	Eingang	ADI-Adapter (DINT) – liefert den zu konvertierenden Wert und ein Ereignis
Plug	`AUI_OUT`	Ausgang	AUI-Adapter (UINT) – stellt den konvertierten Wert und zugehöriges Ereignis bereit

Funktionsweise
Der Baustein besitzt keine eigenen Ein-/Ausgangs-Ereignisse oder Datenpunkte auf der obersten Ebene. Die gesamte Steuerung und Datenübertragung erfolgt über die beiden Adapter:

1. **Ereignisweiterleitung**: Das am Socket `ADI_IN` eingehende Ereignis `E1` wird direkt an den internen Baustein `F_DINT_TO_UINT` (Typ: `iec61131::conversion`) weitergeleitet, um eine Konvertierung zu triggern.
2. **Datenpfad**: Der über `ADI_IN.D1` bereitgestellte DINT-Wert wird an den Eingang `IN` des `F_DINT_TO_UINT` übergeben. Nach erfolgreicher Konvertierung liegt das Ergebnis (UINT) am Ausgang `OUT` an und wird auf den Datenausgang `D1` des Plug-Adapters `AUI_OUT` gelegt.
3. **Rückmeldung**: Der interne Baustein signalisiert das Ende der Konvertierung durch das Ereignis `CNF`. Dieses Ereignis wird auf das Ausgangsereignis `E1` des Plug-Adapters `AUI_OUT` übertragen – synchron zum bereitgestellten Datenwert.

Damit realisiert der FB eine **unidirektionale, ereignisgesteuerte Übertragung** eines DINT-Werts in einen UINT-Wert über die Adapterschnittstellen.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-FB**: Der Baustein kapselt die Konvertierungslogik vollständig und bietet eine aufgeräumte, wiederverwendbare Schnittstelle.
- **Verwendeter Kernbaustein**: `iec61131::conversion::F_DINT_TO_UINT` – eine standardisierte IEC 61131-Funktion zur Umwandlung von DINT in UINT (Wertebereichsanpassung: DINT auf UINT wird ohne Vorzeichen betrachtet, negative Werte werden entsprechend umgesetzt).
- **Adapter-Typen**: Die Adapter `ADI` und `AUI` sind als unidirektionale Adapter definiert (`adapter::types::unidirectional`). Sie stellen jeweils ein Ereignispaar (`E1`/`E1`) und einen Datenpunkt (`D1` im entsprechenden Datentyp) bereit.
- **Paket-Struktur**: Der FB ist im Paket `adapter::conversion::unidirectional` organisiert.
- **Lizenz**: Der Baustein unterliegt der Eclipse Public License 2.0 (EPL-2.0). Copyright liegt bei der HR Agrartechnik GmbH.

Zustandsübersicht

Als Composite-FB besitzt **ADI_TO_AUI** keinen eigenen internen Zustandsautomaten. Der Zustand wird ausschließlich durch den eingebetteten Baustein `F_DINT_TO_UINT` bestimmt, der nach Erhalt des `REQ`-Ereignisses die Konvertierung vornimmt und mit `CNF` abschließt. Der Baustein ist daher **zustandslos** im Sinne eines reinen Funktionsbausteins (keine speichernden Zustände außerhalb des Kernbausteins).

Anwendungsszenarien

- **Schnittstellenanpassung** zwischen Systemkomponenten, die unterschiedliche Datentypen für analoge oder digitale Werte verwenden (z. B. Sensoren mit DINT-Ausgabe und Aktoren mit UINT-Eingabe).
- **Datenkonsolidierung** in heterogenen IEC 61499-Netzwerken, in denen Adapter unterschiedlicher Datentypen aufeinandertreffen.
- **Prototypen und Migration**: Einfache Integration von Bausteinen, die ursprünglich mit DINT arbeiten, in eine Umgebung, die UINT erwartet – ohne manuelle Konvertierung in jedem Verbindungsschritt.

Vergleich mit ähnlichen Bausteinen

Direkt vergleichbare Adapter-Konvertierungsblöcke existieren für andere Datentypen (z. B. `BOOL_TO_UINT`, `INT_TO_DINT`). Der Unterschied von **ADI_TO_AUI** liegt in der spezifischen Kombination der Adapter-Schnittstellen (ADI ↔ AUI). Im Gegensatz zu einfachen Konvertierungs-FBs (`F_DINT_TO_UINT`) kapselt dieser Composite-FB sowohl die Adapter-Anbindung als auch die Ereignis- und Datensynchronisation in einem wiederverwendbaren Modul.

Ein alternatives Design könnte einen generischen Konvertierungsbaustein mit parametrierbaren Adaptertypen verwenden, jedoch bietet die dedizierte Lösung eine klarere Dokumentation und einfachere Handhabung in der 4diac-IDE.

Fazit

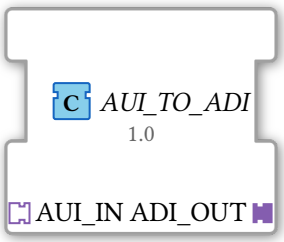
Der Funktionsblock **ADI_TO_AUI** bietet eine saubere, wartbare und standardisierte Möglichkeit, DINT-Daten über einen ADI-Adapter in ein UINT-Signal über einen AUI-Adapter umzuwandeln. Durch die Verwendung eines IEC 61131-Kernbausteins ist die Konvertierung normkonform und zuverlässig. Der Composite-Ansatz erhöht die Wiederverwendbarkeit und vereinfacht das Engineering in verteilten Automatisierungssystemen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

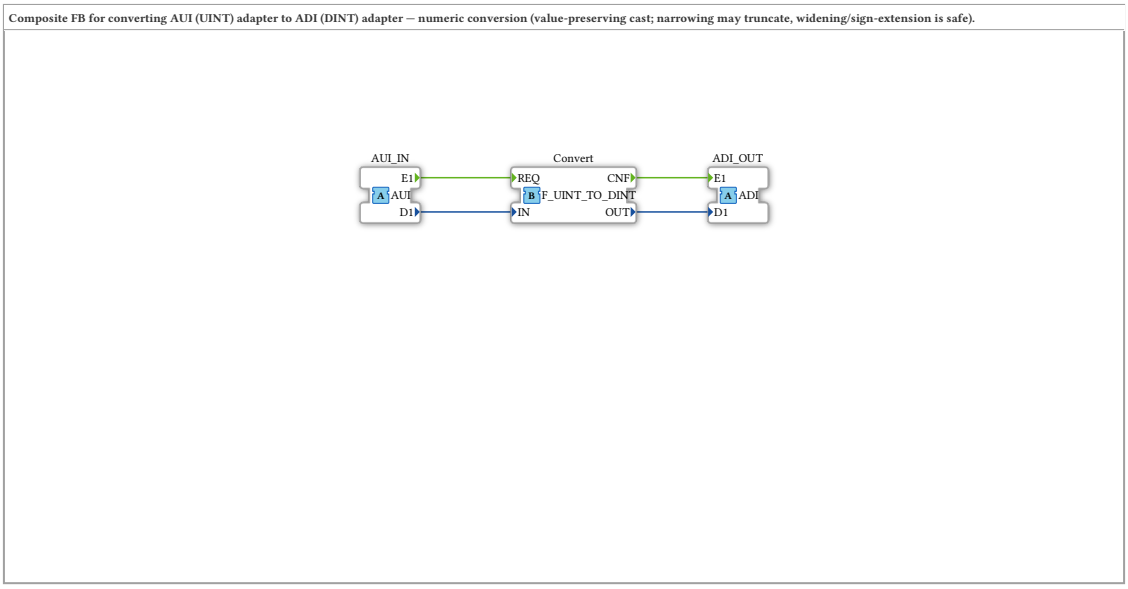
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUI_TO_ADI

UINT Adapter Input – *AUI*



ADI – DINT Adapter Output



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock AUI_TO_ADI ist ein Composite-Baustein, der einen unidirektionalen AUI-Adapter (Datentyp UINT) in einen unidirektionalen ADI-Adapter (Datentyp DINT) umwandelt. Er dient als standardisierte Schnittstelle zwischen Komponenten, die unterschiedliche Datentypen für analoge Ein- oder Ausgangswerte verwenden.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignissteuerung erfolgt über den angeschlossenen AUI-Adapter (Socket AUI_IN).

Ereignis-Ausgänge

Keine. Die Ereignisausgabe erfolgt über den ADI-Adapter (Plug ADI_OUT).

Daten-Eingänge

Keine. Die Dateneingabe erfolgt über den angeschlossenen AUI-Adapter.

Daten-Ausgänge

Keine. Die Datenausgabe erfolgt über den ADI-Adapter.

Adapter

Name	Richtung	Typ	Beschreibung
AUI_IN	Socket (Eingang)	`adapter::types::unidirectional::AUI`	UINT-Adapter zum Empfangen von Eingangsdaten und Ereignissen.
ADI_OUT	Plug (Ausgang)	`adapter::types::unidirectional::ADI`	DINT-Adapter zum Senden der konvertierten Daten und Ereignisse.

Funktionsweise

Der FB empfängt über den Socket AUI_IN ein Ereignis sowie einen UINT-Wert. Dieses Ereignis triggert den internen Konvertierungsbaustein `F_UINT_TO_DINT`, der den Wert vom Typ UINT in den Typ DINT umwandelt. Nach erfolgreicher Konvertierung wird ein Ereignis über den Plug ADI_OUT ausgegeben, zusammen mit dem konvertierten DINT-Wert. Die Konvertierung erfolgt entsprechend der IEC-61131-Norm und ist verlustfrei, da der Wertebereich von UINT (0…65535) vollständig in DINT abgebildet werden kann.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- Der FB ist als **Composite-Baustein** realisiert, d.h. seine Funktionalität basiert auf einem enthaltenen IEC-61131-Baustein (`F_UINT_TO_DINT`).
- Die Kommunikation erfolgt ausschließlich über **unidirektionale Adapter** – es werden keine eigenständigen Ereignis- oder Dateneingänge/-ausgänge auf oberster Ebene benötigt.
- Der interne Konverter arbeitet **ereignisgesteuert**: Jedes eingehende Ereignis am AUI_IN-Adapter löst eine Konvertierung aus.

Zustandsübersicht

Der FB besitzt keine eigene Zustandsmaschine. Die Zustandslogik wird vollständig durch den internen Baustein `F_UINT_TO_DINT` abgebildet, der bei jedem REQ-Ereignis eine einmalige Konvertierung durchführt und mit CNF quittiert.

Anwendungsszenarien

- **Schnittstellenanpassung**: Ein Sensor liefert über einen UINT-basierten AUI-Adapter Messwerte, die nachgeschaltete Logik benötigt jedoch DINT-Werte (z.B. für arithmetische Operationen oder Visualisierung).
- **Systemintegration**: In heterogenen Automatisierungssystemen, in denen unterschiedliche Komponenten verschiedene Adaptertypen verwenden, kann der Baustein als vermittelnde Schicht eingesetzt werden.
- **Migration**: Bestehende Anlagen mit UINT-Datenpfaden sollen schrittweise auf DINT umgestellt werden.

Vergleich mit ähnlichen Bausteinen

- **F_UINT_TO_DINT**: Direkter IEC-61131-Konverter, jedoch ohne Adapterkapselung. Erfordert separate Ereignis- und Datenleitungen.
- **AUI_TO_ABOOL**: Analoger Konverter für binäre Ausgaben (UINT → BOOL), nicht für numerische Typen.
- **ATO_TO_ADI**: Konvertiert ATO-Adapter (DWORD) nach ADI, andere Datenbreite.

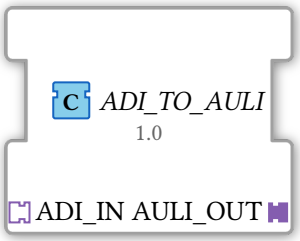
Der vorliegende Baustein hebt sich durch die vollständige Adapterintegration und die unidirektionale Auslegung ab, was den Einsatz in adapterbasierten Architekturen erleichtert.

Fazit

Der Funktionsblock `AUI_TO_ADI` bietet eine saubere, gekapselte Lösung zur Typkonvertierung von UINT auf DINT über standardisierte Adapter. Er reduziert den Verdrahtungsaufwand und erhöht die Wiederverwendbarkeit in modularen Automatisierungslösungen. Durch die Nutzung eines bewährten IEC-61131-Bausteins ist die Korrektheit der Konvertierung sichergestellt.

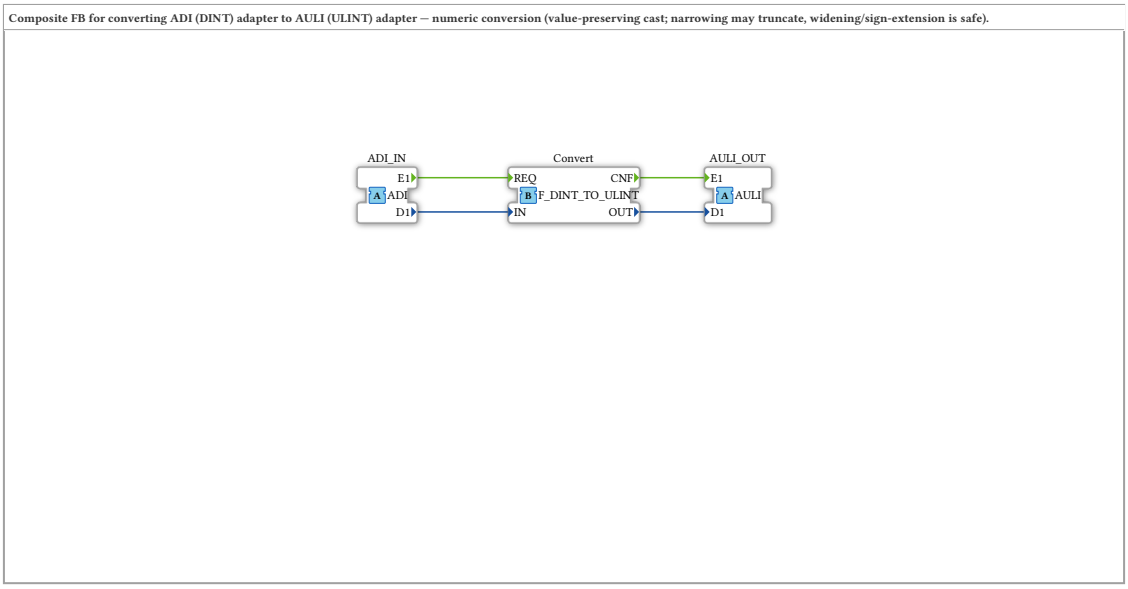
ADI_AULI

ADI_TO_AULI



DINT Adapter Input – *ADI*

AULI – ULINT Adapter Output



• — ○ • — ○ • — ○ •

Einleitung

Der Baustein **ADI_TO_AULI** ist ein Composite-Funktionsblock (FB) gemäß IEC 61499. Er dient der Umwandlung eines DINT-Wertes (vorzeichenbehaftete 32-Bit-Ganzzahl), der über einen ADI-Adapter (Input-Schnittstelle) eingeht, in einen ULINT-Wert (vorzeichenlose 64-Bit-Ganzzahl), der über einen AULI-Adapter (Output-Schnittstelle) ausgegeben wird. Intern wird der standardisierte Konvertierungsbaustein `F_DINT_TO_ULINT` aus der IEC-61131-Bibliothek verwendet. Der Baustein vereinfacht die Integration von Adaptern und die Typumwandlung in Automatisierungslösungen.

Schnittstellenstruktur

Ereignis-Eingänge

Bezeichnung	Typ	Beschreibung
ADI_IN.E1	Ereignis (implizit)	Auslöser für die Konvertierung; wird vom ADI-Adapter bereitgestellt.

Ereignis-Ausgänge

Bezeichnung	Typ	Beschreibung
AULI_OUT.E1	Ereignis (implizit)	Signalisiert den Abschluss der Konvertierung; wird an den AULI-Adapter weitergegeben.

Daten-Eingänge

Bezeichnung	Typ	Beschreibung
ADI_IN.D1	DINT	Eingangswert, der in ULINT konvertiert werden soll.

Daten-Ausgänge

Bezeichnung	Typ	Beschreibung
AULI_OUT.D1	ULINT	Konvertierter Ausgangswert (vorzeichenlos).

Adapter

Name	Typ (Schnittstelle)	Richtung	Beschreibung

ADI_IN	ADI (Socket)	Input	Nimmt den DINT-Wert über einen unidirektionalen ADI-Adapter entgegen.
AULI_OUT	AULI (Plug)	Output	Stellt den konvertierten ULINT-Wert über einen unidirektionalen AULI-Adapter bereit.

Funktionsweise

Der Baustein wartet auf ein Ereignis am Eingangsadapter `ADI_IN.E1`. Sobald dieses eintrifft, wird der aktuelle Datenwert `ADI_IN.D1` (Typ DINT) an den internen Konvertierungs-FB `F_DINT_TO_ULINT` übergeben. Nach erfolgreicher Umwandlung wird das Ergebnis (Typ ULINT) an den Ausgang `AULI_OUT.D1` weitergeleitet und ein Ereignis am Ausgang `AULI_OUT.E1` ausgelöst. Die gesamte Logik ist rein kombinatorisch – es gibt keine Verzögerungen oder Zustände außer der Signalverarbeitung des internen Bausteins.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Adapterbasierte Schnittstelle:** Der Baustein verwendet ausschließlich Adapter-Plugs und -Sockets für die Ein- und Ausgabe. Dies erlaubt eine saubere Kapselung und Wiederverwendung in modularen Steuerungsarchitekturen.
- **Unidirektionale Adapter:** Sowohl der ADI- als auch der AULI-Adapter sind als unidirektional deklariert und übertragen nur in eine Richtung (Eingang oder Ausgang).
- **Interne Konvertierung:** Der für die eigentliche Wandlung zuständige FB `F_DINT_TO_ULINT` ist dem Paket `iec61131::conversion` entnommen und stellt eine standardisierte, ablaufinvariante Funktion bereit.
- **Namensraum:** Der Composite-FB gehört zum Paket `adapter::conversion::unidirectional`, was seine Rolle als Adapter-Wrapper deutlich macht.

Zustandsübersicht

Als Composite-Baustein besitzt `ADI_TO_AULI` keine eigene Zustandsmaschine. Sein Verhalten wird vollständig durch die interne Verknüpfung der Ereignis- und Datenflüsse zwischen dem Socket `ADI_IN`, dem Konvertierungs-FB und dem Plug `AULI_OUT` bestimmt. Die Zustandslogik des internen Bausteins `F_DINT_TO_ULINT` ist für die Funktionsweise transparent.

Anwendungsszenarien

- **Übergang zwischen Steuerungssystemen:** Wenn ein System mit DINT-Signalen (z. B. SPS-Daten) mit einem anderen System kommunizieren muss, das ULINT-Werte erwartet (z. B. Visualisierung oder Langzeitarchivierung mit 64-Bit-Zeitstempeln).
- **Adapter-Brücke:** In einer modularen Automatisierungsumgebung, in der Adapter für unterschiedliche Datentypen definiert sind, dient der Baustein als einfaches „Übersetzungsglied" zwischen ADI- und AULI-Schnittstellen.
- **Prototypen oder Migration:** Schnelle Integration von Komponenten mit unterschiedlichen Integer-Darstellungen ohne anwendungsspezifische Konvertierungslogik.

Vergleich mit ähnlichen Bausteinen

- **Separater `F_DINT_TO_ULINT`:** Der hier verwendete interne FB ist direkt für die reine Typumwandlung zuständig, erfordert jedoch separate Ereignis- und Datenverbindungen. `ADI_TO_AULI` kapselt diese Verbindungen und die Adapter-Anbindung in einem einzigen Baustein.
- **Andere Adapter-Wrapper:** Es existieren ähnliche Composite-Bausteine für andere Typumwandlungen (z. B. `ADI_TO_AUDI`, `ASI_TO_AULI`), die nach demselben Muster aufgebaut sind. Sie unterscheiden sich lediglich durch die beteiligten Datentypen.
- **Direkte Typkonvertierung ohne Adapter:** Manche Systeme bieten Konvertierungsfunktionen als reine Funktionsbausteine ohne Adapter-Schicht. Der vorliegende Baustein ist speziell auf den Einsatz mit Adaptern ausgelegt.

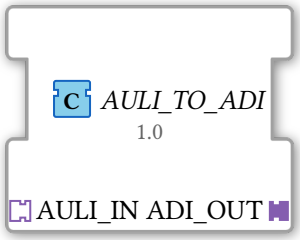
Fazit

Der `ADI_TO_AULI`-Baustein ist ein kompakter, adapterbasierter Wrapper zur Umwandlung von DINT- in ULINT-Werte. Durch die saubere Trennung von Adapter- und Konvertierungslogik erhöht er die Wie-

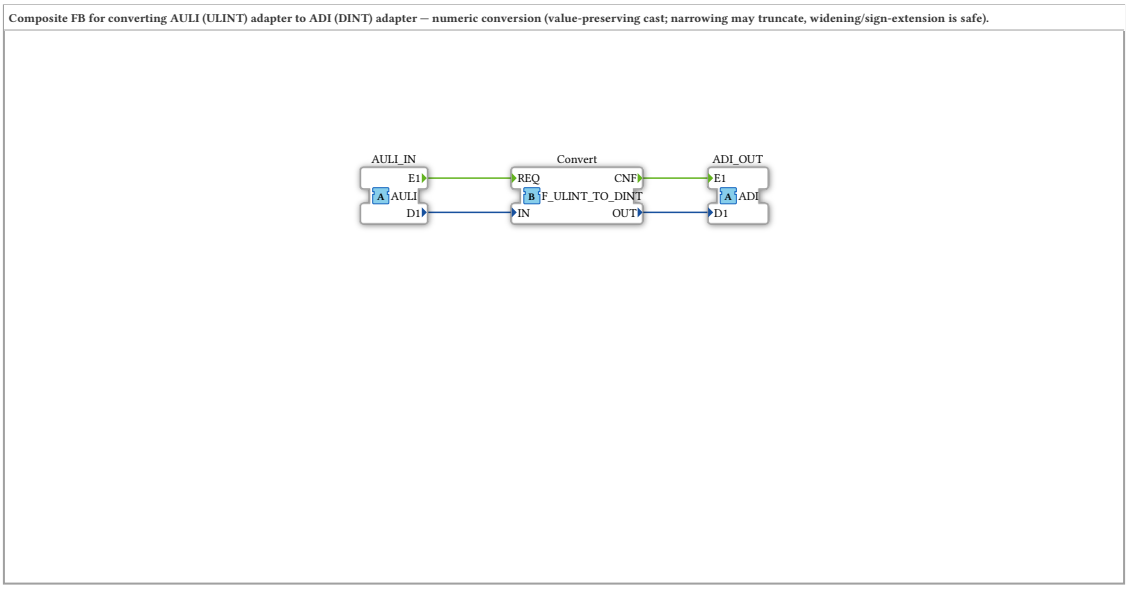
derverwendbarkeit und Lesbarkeit von Steuerungsprogrammen. Er eignet sich besonders für modulare Automatisierungslösungen, die auf unidirektionale Adapter setzen und eine standardisierte Typumwandlung benötigen.

AULI_TO_ADI

ULINT Adapter Input – *AULI*



ADI – DINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AULI_TO_ADI** ist ein zusammengesetzter Baustein zur Umwandlung eines **AULI**-Adapters (unsignierter 64-Bit-Wert) in einen **ADI**-Adapter (vorzeichenbehafteter 32-Bit-Wert). Er ermöglicht die einfache Konvertierung zwischen zwei unidirektionalen Adapterschnittstellen und nutzt intern eine IEC-61131-Typumwandlungsfunktion.

Schnittstellenstruktur

Der Baustein besitzt keine eigenen Ereignis- oder Dateneingänge/-ausgänge. Die gesamte Kommunikation erfolgt ausschließlich über die beiden Adapter.

Ereignis-Eingänge

Keine (ausschließlich über Adapter).

Ereignis-Ausgänge

Keine (ausschließlich über Adapter).

Daten-Eingänge

Keine (ausschließlich über Adapter).

Daten-Ausgänge

Keine (ausschließlich über Adapter).

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AULA`	`AULI_IN`	Socket / Eingang	Aufnahme eines ULINT-Werts über Ereignis- und Datenkanäle (E1, D1)
`adapter::types::unidirectional::ADA`	`ADI_OUT`	Plug / Ausgang	Bereitstellung des konvertierten DINT-Werts über Ereignis- und Datenkanäle (E1, D1)

Funktionsweise

Der Baustein arbeitet als reine Kaskade:

1. Ein Ereignis am Socket `AULI_IN.E1` löst die Datenübergabe von `AULI_IN.D1` (ULINT) aus.
2. Dieses Ereignis wird auf den Eingang `REQ` des internen Konvertierungsbausteins `F_ULINT_TO_DINT` (IEC-61131-Typumwandlung) übertragen.
3. Der Konverter wandelt den ULINT-Wert in einen DINT-Wert um und legt ihn an seinem Ausgang `OUT` an.
4. Nach Abschluss der Umwandlung signalisiert der Konverter mit `CNF` das Ereignis, das den Plug `ADI_OUT.E1` auslöst und gleichzeitig den konvertierten Wert auf `ADI_OUT.D1` bereitstellt.
5. Der angeschlossene Empfänger erhält so einen DINT-Wert, der dem ursprünglichen ULINT-Wert – soweit darstellbar (Wertebereich DINT: -2^{31} … 2^{31}-1) – entspricht.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Adapterbasiert**: Der FB implementiert keine direkten Ein-/Ausgangsvariablen, sondern nutzt ausschließlich Adapter-Sockets und -Plugs für lose Kopplung.
- **Typumwandlung**: Die Konvertierung erfolgt mittels des IEC-61131-Standardbausteins `F_ULINT_TO_DINT`. Dabei können ULINT-Werte außerhalb des DINT-Bereichs zu Überläufen oder unerwarteten Ergebnissen führen.
- **Kompakte Struktur**: Da keine Zustandslogik oder Verzweigungen vorhanden sind, arbeitet der Baustein deterministisch und ohne Verzögerung jenseits der reinen Konvertierungszeit.
- **Paket**: Der Baustein ist im Package `adapter::conversion::unidirectional` zusammengefasst.

Zustandsübersicht

Der FB besitzt keinen internen Zustandsautomaten. Er verhält sich wie eine reine funktionale Abbildung: Auf ein Eingangsereignis folgt unmittelbar ein Ausgangsereignis mit dem umgewandelten Wert.

Anwendungsszenarien

- **Schnittstellenanpassung**: Wenn ein Sensor oder eine Steuerung einen ULINT-Adapter liefert, das nachgelagerte System aber einen DINT-Adapter erwartet.
- **Datenreduktion**: Reduzierung der Bitbreite von 64 auf 32 Bit in Fällen, in denen die Auflösung von 32 Bit ausreicht.
- **Adapterkaskaden**: Verwendung als ein Glied in einer lückenlosen Umwandlungskette zwischen unterschiedlichen Datenadaptertypen.

Vergleich mit ähnlichen Bausteinen

- **AULI_TO_UDI**: Wandelt ULINT in UDINT (unsigned 32-Bit) – hier bleibt der Wertebereich positiv und Überläufe treten erst bei Werten > 2^{32}-1 auf.
- **DIRECT_CONVERT**: Ein generischer Baustein mit direktem Dateneingang und -ausgang, der keine Adapter verwendet – AULI_TO_ADI bietet dagegen lose Kopplung und standardisierte Adapterschnittstellen.
- **Manuelle Konvertierung**: Ohne diesen Baustein müsste die Umwandlung im Anwendungs-FB selbst implementiert werden – AULI_TO_ADI kapselt diese Aufgabe und fördert die Wiederverwendbarkeit.

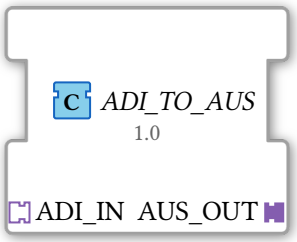
Fazit

AULI_TO_ADI ist ein spezialisierter, adapterbasierter Konvertierungsbaustein, der die Umwandlung von ULINT- auf DINT-Daten über unidirektionale Adapter effizient und standardisiert abwickelt. Durch die Nutzung eines IEC-61131-Konvertierungs-FBs ist die Umwandlung zuverlässig, und die reine Adapter-Schnittstelle ermöglicht eine flexible Integration in bestehende Systeme. Der Baustein eignet sich besonders für lose gekoppelte Architekturen, in denen Datenformate angepasst werden müssen, ohne die Anwendungslogik mit Typkonvertierung zu belasten.

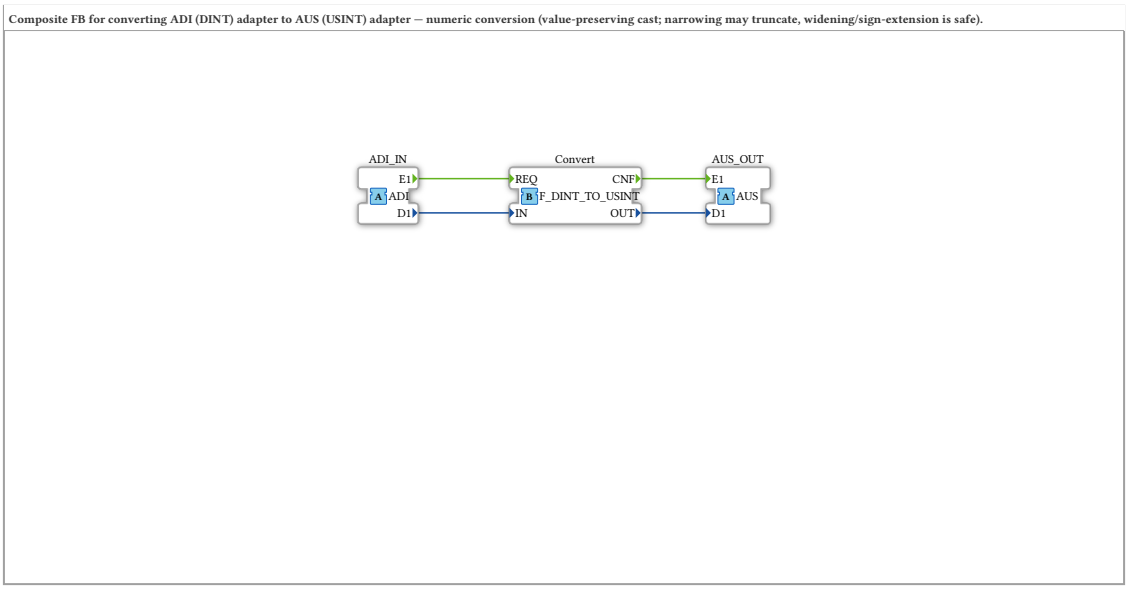
ADI_AUS

ADI_TO_AUS

DINT Adapter Input – *ADI*



AUS – USINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock ADI_TO_AUS dient als Adapter-Konverter von einem DINT-Adapter (ADI) zu einem USINT-Adapter (AUS). Er wandelt einen DINT-Wert in einen USINT-Wert um und ermöglicht die nahtlose Integration von Komponenten mit unterschiedlichen Datentypen in der 4diac-IDE.

Schnittstellenstruktur

Ereignis-Eingänge

- **E1** (über ADI_IN): Das Ereignis löst die Konvertierung aus.

Ereignis-Ausgänge

- **E1** (über AUS_OUT): Signalisiert den Abschluss der Konvertierung und die Bereitstellung des Ausgangswerts.

Daten-Eingänge

- **D1** (über ADI_IN): DINT-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **D1** (über AUS_OUT): USINT-Wert, das Ergebnis der Konvertierung.

Adapter

- **ADI_IN**: Socket (Eingang) vom Typ `adapter::types::unidirectional::ADI`. Stellt DINT-Daten und Ereignisse bereit.
- **AUS_OUT**: Plug (Ausgang) vom Typ `adapter::types::unidirectional::AUS`. Empfängt USINT-Daten und Ereignisse zur weiteren Verarbeitung.

Funktionsweise

Der Baustein ist als Composite FB realisiert. Bei einem Ereignis am Ereigniseingang E1 (über ADI_IN) wird der zugehörige DINT-Wert von Daten-Eingang D1 ausgelesen und dem internen Konvertierungsbaustein `F_DINT_TO_USINT` übergeben. Dieser wandelt den Wert nach IEC 61131-Standard um (z. B. durch Begrenzung auf den USINT-Bereich 0 – 255 oder Typumwandlung). Das Ergebnis wird über den Daten-Ausgang D1 ausgegeben und ein Ereignis am Ausgang E1 (über AUS_OUT) signalisiert die Verfügbarkeit des neuen Werts.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- Der FB verwendet einen IEC 61131-konformen Konvertierungsbaustein für die Typumwandlung von DINT zu USINT.
- Es handelt sich um einen Composite FB (verschachtelter Baustein), der die Logik in einem Netzwerk aus einem einzelnen Convert-FB abbildet.
- Die Kommunikation erfolgt ausschließlich über Adapter-Schnittstellen, was eine modulare Kapselung ermöglicht.
- Bei der Konvertierung können Werte außerhalb des USINT-Bereichs (0 – 255) abgeschnitten oder auf den maximalen bzw. minimalen Wert begrenzt werden (abhängig vom verwendeten Konvertierungsbaustein).

Zustandsübersicht

Da der Baustein keinen eigenen Zustandsautomaten besitzt, ist sein Verhalten rein ereignisgesteuert: Auf jedes ankommende Ereignis reagiert er mit einer sofortigen Konvertierung und Ausgabe. Es gibt keine internen Zustände oder Verzögerungen.

Anwendungsszenarien

- Anbindung eines Sensors, der DINT-Werte liefert, an eine Steuerung, die USINT erwartet.
- Integration von Adapter-basierten Komponenten mit unterschiedlichen Datentypen in einer 4diac-Applikation.
- Migration von bestehenden Systemen, bei denen Datenformate angepasst werden müssen.

Vergleich mit ähnlichen Bausteinen

- Ein direkter Konvertierungsbaustein wie `F_DINT_TO_USINT` führt dieselbe Umwandlung durch, arbeitet jedoch auf Standard-Ein-/Ausgängen. Der ADI_TO_AUS kapselt die Logik in Adapter-Schnittstellen und ist daher speziell für Adapter-basierte Kommunikationswege ausgelegt.
- Der Baustein unterscheidet sich von anderen Adapter-Konvertern durch die spezifische Typkombination DINT↔USINT.

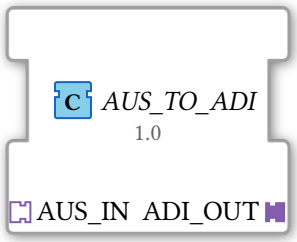
Fazit

Der ADI_TO_AUS ist ein praktischer Hilfsbaustein für die nahtlose Integration von DINT- und USINT-Adaptern in der 4diac-IDE. Seine einfache, ereignisgesteuerte Funktionsweise und die Kapselung als Composite FB machen ihn flexibel einsetzbar und gut wartbar. Er eignet sich besonders für Projekte, die eine klare Trennung zwischen Adapter-Schnittstellen und Konvertierungslogik erfordern.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

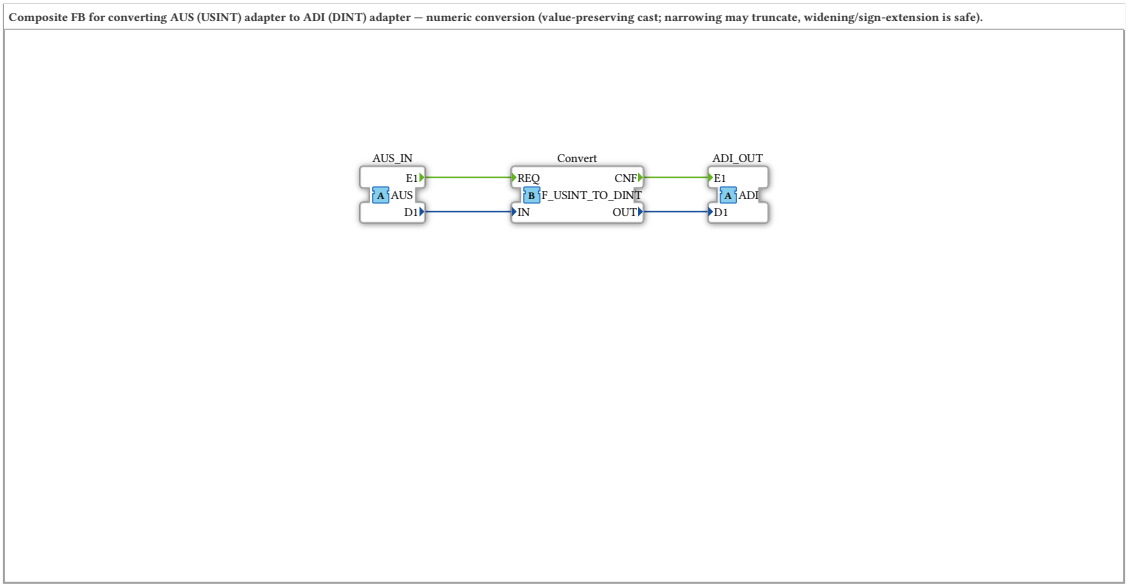
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUS_TO_ADI



USINT Adapter Input – *AUS*

ADI – DINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein AUS_TO_ADI ist ein zusammengesetzter (Composite) Funktionsbaustein, der eine unidirektionale Adapter-Konvertierung von `AUS` (Daten vom Typ `USINT`) nach `ADI` (Daten vom Typ `DINT`) realisiert. Er kapselt einen IEC 61131-Konvertierungsbaustein und ermöglicht so eine saubere Trennung von Adaptertypen in der 4diac-IDE.

Schnittstellenstruktur

Ereignis-Eingänge

Keine separaten Ereignis-Eingänge vorhanden. Die Ereignissteuerung erfolgt über den Socket-Adapter `AUS_IN` (siehe Abschnitt **Adapter**).

Ereignis-Ausgänge

Keine separaten Ereignis-Ausgänge vorhanden. Die Ereignisweiterleitung erfolgt über den Plug-Adapter `ADI_OUT` (siehe Abschnitt **Adapter**).

Daten-Eingänge

Keine separaten Daten-Eingänge vorhanden. Die Daten werden über den Socket-Adapter `AUS_IN` bereitgestellt.

Daten-Ausgänge

Keine separaten Daten-Ausgänge vorhanden. Die konvertierten Daten werden über den Plug-Adapter `ADI_OUT` ausgegeben.

Adapter

Name	Richtung	Typ	Beschreibung

AUS_IN	Socket (Eingang)	adapter::types::unidirectional::AUS	Unidirektionaler Adapter-Eingang, der einen USINT-Wert (und ein zugehöriges Ereignis) bereitstellt.
ADI_OUT	Plug (Ausgang)	adapter::types::unidirectional::ADI	Unidirektionaler Adapter-Ausgang, der den konvertierten DINT-Wert (und ein zugehöriges Ereignis) ausgibt.

Funktionsweise

Der Baustein arbeitet ereignisgesteuert:

1. Ein eingehendes Ereignis am Socket `AUS_IN.E1` triggert den internen Konvertierungsbaustein `Convert` (Typ `F_USINT_TO_DINT`).
2. Der Konvertierungsbaustein liest den Datenwert `AUS_IN.D1` (`USINT`) und wandelt ihn in einen `DINT`-Wert um.
3. Nach Abschluss der Konvertierung wird ein Ereignis am Ausgang `Convert.CNF` erzeugt.
4. Dieses Ereignis wird auf den Plug-Adapter `ADI_OUT.E1` weitergeleitet, und der konvertierte Wert wird über `ADI_OUT.D1` ausgegeben.

Die gesamte Logik ist als Datenfluss im enthaltenen FBNetzwerk abgebildet – es wird keine Zustandsmaschinenlogik innerhalb des Composite-Bausteins verwendet.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- Der Baustein verwendet den IEC 61131-Baustein `F_USINT_TO_DINT` (Typkonvertierung vorzeichenloser 8-Bit zu vorzeichenbehaftetem 32-Bit Integer).
- Es handelt sich um einen reinen **Composite-Baustein** – das Verhalten wird vollständig durch das interne FBNetzwerk definiert.
- Die Adapter sind **unidirektional** ausgelegt: Daten fließen nur vom Socket zum Plug.
- Durch die Kapselung der Konvertierung in einen eigenen Baustein können unterschiedliche Adaptertypen zusammengeschaltet werden, ohne dass die umgebende Applikation die Typen konvertieren muss.

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Das Verhalten wird direkt durch den inneren Konvertierungsbaustein und die Ereignisverbindungen gesteuert. Eine Fehlerbehandlung (z. B. Bereichsüberschreitung bei der Konvertierung) ist nicht explizit implementiert und müsste entweder im aufrufenden Kontext oder durch Erweiterung des Bausteins ergänzt werden.

Anwendungsszenarien

- **Adapter-Brücke**: Austausch von Daten zwischen Komponenten, die unterschiedliche Adaptertypen verwenden (z. B. Sensoren mit `USINT`-Werten und Steuerungen, die `DINT`-Werte erwarten).
- **Datenintegration**: Einbindung von Bausteinen aus IEC 61131-Bibliotheken in eine 4diac-Anwendung, die auf Adaptern basiert.
- **Typanpassung**: Schnelle Umwandlung von Datentypen in der Adapter-Ebene, ohne die Logik der angeschlossenen Bausteine ändern zu müssen.

Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu einem direkten Einsatz des Konvertierungsbausteins `F_USINT_TO_DINT` bietet `AUS_TO_ADI` eine **adapter-orientierte Schnittstelle**. Dadurch wird die Konvertierung vollständig transparent für den Anwender, der nur mit den Adaptern arbeitet.
- Ähnliche Adapter-Konvertierungen könnten für andere Datentypen existieren (z. B. `BOOL_TO_ADI`, `INT_TO_ADI`). Der vorliegende Baustein ist speziell für die Umwandlung von `USINT` nach `DINT` ausgelegt.

Fazit

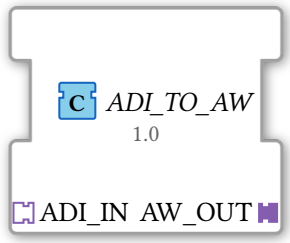
Der Composite-Baustein `AUS_TO_ADI` bietet eine einfache und saubere Möglichkeit, einen `AUS`-Adapter (USINT-basiert) in einen `ADI`-Adapter (DINT-basiert) zu konvertieren. Durch die interne Nutzung des IEC 61131-Konvertierungsbausteins ist die Umsetzung standardkonform und wiederverwendbar. Der Baustein eignet sich besonders für modulare Anwendungen, bei denen unterschiedliche Adaptertypen harmonisiert werden müssen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

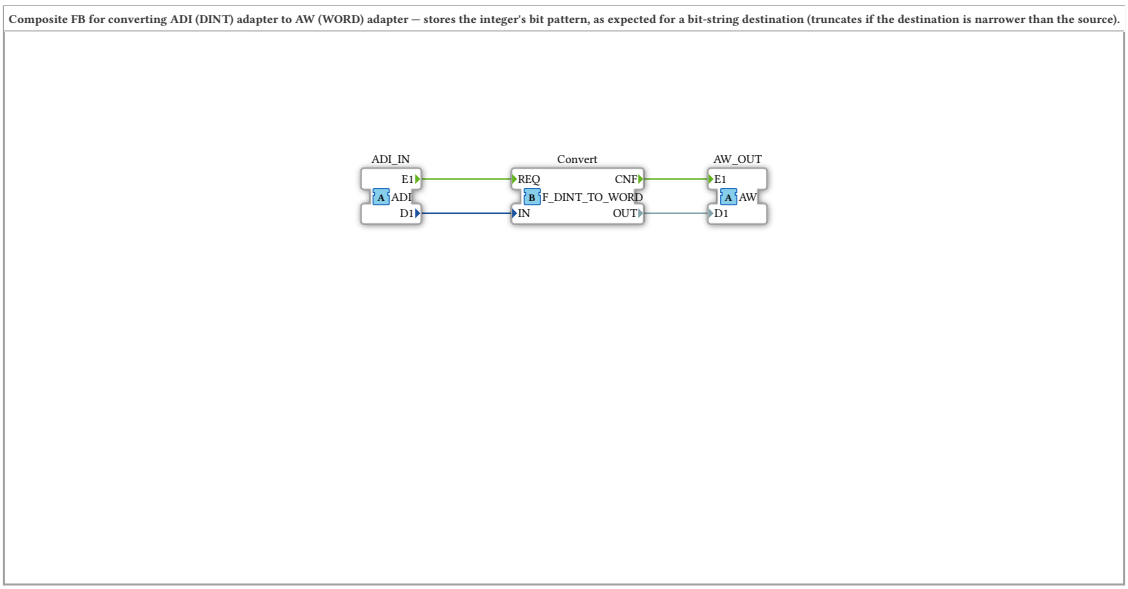
ADI_AW

ADI_TO_AW



DINT Adapter Input – *ADI*

AW – WORD Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ADI_TO_AW** ist ein zusammengesetzter Baustein, der die Umwandlung eines DINT-Wertes in einen WORD-Wert realisiert. Er fungiert als unidirektionaler Konverter zwischen einem **ADI**-Adapter (DINT-Eingang) und einem **AW**-Adapter (WORD-Ausgang). Der FB kapselt die Konvertierungslogik und stellt eine standardisierte Schnittstelle für den Datenaustausch auf Adapterebene bereit.

Schnittstellenstruktur

Der FB verfügt über keine direkten Ereignis- oder Datenein-/ausgänge. Die gesamte Kommunikation erfolgt über Adapter:

Adapter

Typ	Name	Richtung	Beschreibung

adapter::types::unidirectional::ADI	ADI_IN	Socket (Eingang)	DINT-Adapter, der den zu konvertierenden Wert bereitstellt
adapter::types::unidirectional::AW	AW_OUT	Plug (Ausgang)	WORD-Adapter, der den konvertierten Wert ausgibt

Die internen Ereignis- und Datenverbindungen des Composite-FBs ergeben sich aus den genutzten Adapter-Schnittstellen:

Ereignis-Eingänge (über ADI_IN)

Ereignis	Beschreibung
E1	Trigger für die Konvertierung des DINT-Wertes

Ereignis-Ausgänge (über AW_OUT)

Ereignis	Beschreibung
E1	Bestätigung der erfolgten Konvertierung und Ausgabe des WORD-Wertes

Daten-Eingänge (über ADI_IN)

Bezeichner	Typ	Beschreibung
D1	DINT	Der zu konvertierende Ganzzahlwert

Daten-Ausgänge (über AW_OUT)

Bezeichner	Typ	Beschreibung
D1	WORD	Der konvertierte Bitmuster-Wert (16 Bit)

Funktionsweise

Der FB arbeitet ereignisgesteuert:

1. Ein eingehendes Ereignis an ADI_IN.E1 wird intern an den Konvertierungsbaustein F_DINT_TO_WORD weitergeleitet (über dessen REQ-Eingang).
2. Gleichzeitig wird der Datenwert von ADI_IN.D1 als DINT an den IN-Eingang der Konvertierungsfunktion übergeben.
3. Der Baustein F_DINT_TO_WORD wandelt den DINT-Wert in einen WORD-Wert um (gemäß IEC 61131-3).
4. Nach Abschluss der Konvertierung wird das Ausgangsereignis (CNF) des Konverters abgesetzt.
5. Dieses Ereignis wird an AW_OUT.E1 weitergeleitet, und der konvertierte WORD-Wert (von OUT) wird an AW_OUT.D1 übergeben.

Somit erfolgt die gesamte Konvertierung in einem strengen Ereignis-/Daten-Fluss ohne zusätzliche Zustände.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Adapterbasierte Kapselung:** Der FB nutzt ausschließlich Adapter für Ein- und Ausgabe, was eine modulare und wiederverwendbare Anbindung in 4diac-Projekten ermöglicht.
- **Unidirektionale Konvertierung:** Es findet nur eine Datenwandlung vom Socket zum Plug statt; eine Rückwärtskonvertierung ist nicht vorgesehen.
- **Verwendung von IEC 61131-Bibliotheken:** Die Kernfunktion F_DINT_TO_WORD stammt aus dem iec61131::conversion-Paket, sodass die Konvertierung standardkonform und performant ist.
- **Keine eigenen Zustände:** Der FB besitzt kein internes Zustandsdiagramm; die Logik wird allein durch das Verhalten des enthaltenen Bausteins bestimmt.

Zustandsübersicht

Da es sich um einen rein strukturierten Composite-FB ohne eigene ECC (Execution Control Chart) handelt, existiert kein explizites Zustandsmodell. Das Verhalten ist rein datenstromgetrieben und entspricht dem des internen Konvertierungsbausteins (IDLE → REQ → CNF → IDLE).

Anwendungsszenarien

- **SPS-Kopplung:** Anbindung eines DINT-basierten Sensors (z. B. Encoder) an ein Subsystem, das WORD-Werte erwartet.
- **Datenformat-Konvertierung in Adapterketten:** Einsatz in einer 4diac-Anwendung, bei der verschiedene Adapter-Typen harmonisiert werden müssen.
- **Vereinheitlichung von Schnittstellen:** Umsetzung einer einheitlichen WORD-Schnittstelle für nachgelagerte Logikbausteine, während die Quelle im DINT-Format vorliegt.

Vergleich mit ähnlichen Bausteinen

Baustein	Funktion	Unterschied
`F_DINT_TO_WORD`	Direkte Konvertierung DINT → WORD	Keine Adapter-Kapselung; muss manuell in ein Netzwerk integriert werden
`ADI_TO_AW`	Adapterbasierte Kapselung derselben Konvertierung	Bietet sofort einsatzbereite Adapter-Schnittstellen; vereinfacht die Wiederverwendung
`WORD_TO_DINT`	Umgekehrte Konvertierung	Andere Richtung; eigener Adapter-Typ erforderlich

Der `ADI_TO_AW` ist speziell für Umgebungen optimiert, in denen Adapter als standardisierte Schnittstellen gefordert sind.

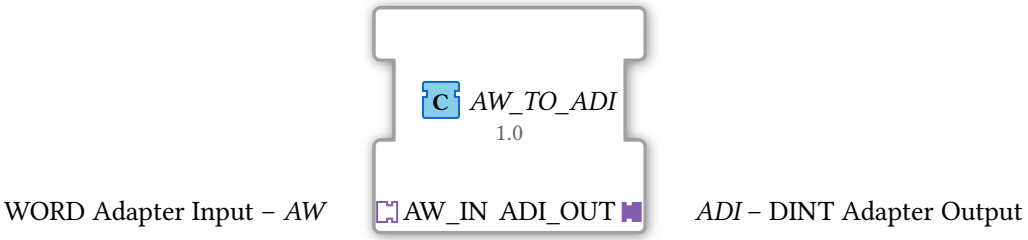
Fazit

Der Funktionsblock `ADI_TO_AW` stellt eine wiederverwendbare, adapterbasierte Lösung zur Konvertierung von `DINT` nach `WORD` dar. Er kapselt die Konvertierungslogik sauber und ermöglicht eine einfache Integration in 4diac-Architekturen, die auf dem Adapter-Konzept basieren. Durch die Verwendung von Standardbibliotheken ist die Funktion zuverlässig und spezifikationstreu. Der Baustein eignet sich besonders für Szenarien, bei denen unterschiedliche Datenformate über Adapterbrücken verbunden werden müssen.

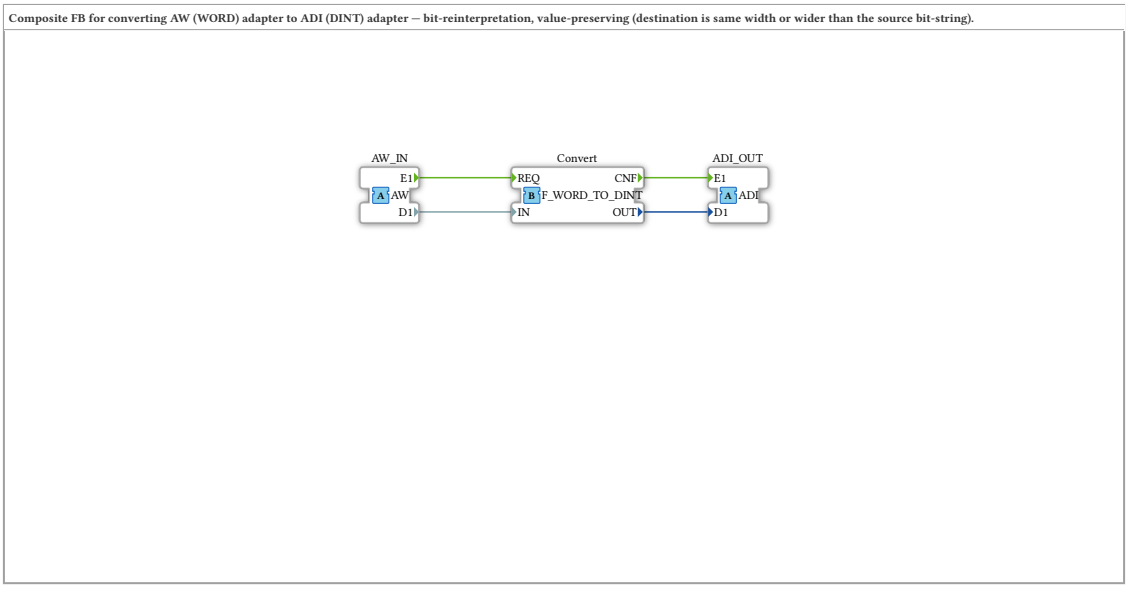
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AW_TO_ADI



WORD Adapter Input – *AW* — *ADI* – DINT Adapter Output



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock **AW_TO_ADI** ist ein Composite-Baustein, der einen unidirektionalen WORD-Adapter (AW) in einen unidirektionalen DINT-Adapter (ADI) umwandelt. Er ermöglicht die nahtlose Integration von Komponenten, die mit WORD-Daten arbeiten, in Systeme, die DINT-Werte erwarten. Intern verwendet der Baustein den Standard-Konvertierungsbaustein `F_WORD_TO_DINT` aus der IEC 61131-Bibliothek.

Schnittstellenstruktur

Ereignis-Eingänge

- **Keine direkten Ereignis-Eingänge** Der Baustein besitzt keine eigenständigen Ereignis-Eingänge. Das auslösende Ereignis wird über den Socket-Adapter `AW_IN` bereitgestellt.

Ereignis-Ausgänge

- **Keine direkten Ereignis-Ausgänge** Der Baustein besitzt keine eigenständigen Ereignis-Ausgänge. Das Ergebnis-Ereignis wird über den Plug-Adapter `ADI_OUT` weitergegeben.

Daten-Eingänge

- **Keine direkten Daten-Eingänge** Der zu konvertierende WORD-Wert wird über den Socket-Adapter `AW_IN` empfangen.

Daten-Ausgänge

- **Keine direkten Daten-Ausgänge** Der konvertierte DINT-Wert wird über den Plug-Adapter `ADI_OUT` ausgegeben.

Adapter

Name	Typ	Richtung	Beschreibung
`AW_IN`	`adapter::types::unidirectional::AW`	Socket (Eingang)	WORD-Adapter als Eingang. Liefert das Auslöseereignis (E1) und den WORD-Datenwert (D1).
`ADI_OUT`	`adapter::types::unidirectional::ADI`	Plug (Ausgang)	DINT-Adapter als Ausgang. Gibt das Bestätigungsereignis (E1) und den konvertierten DINT-Wert (D1) weiter.

Funktionsweise

1. Ein eingehendes Ereignis am Socket `AW_IN.E1` triggert den internen Baustein `F_WORD_TO_DINT` über dessen Ereigniseingang `REQ`.
2. Gleichzeitig wird der WORD-Datenwert von `AW_IN.D1` an den Dateneingang `IN` von `F_WORD_TO_DINT` übergeben.
3. Nach Abschluss der Konvertierung erzeugt `F_WORD_TO_DINT` ein Bestätigungsereignis an `CNF`.
4. Dieses Ereignis wird an den Plug `ADI_OUT.E1` weitergeleitet, während der konvertierte DINT-Wert von `F_WORD_TO_DINT.OUT` an `ADI_OUT.D1` übergeben wird.

Der gesamte Ablauf erfolgt damit synchron: Ein Impuls am WORD-Adapter führt zu einem Impuls am DINT-Adapter mit dem umgewandelten Wert.

Technische Besonderheiten

- **Bit-Reinterpretation, werterhaltend** (Zielbreite ist gleich groß oder größer als der Quell-Bit-String).
- **Reine Kopplung:** Der Baustein enthält keine eigene Logik, sondern verbindet lediglich die Adapter-Schnittstellen mit einem existierenden Konvertierungsbaustein.
- **Verwendung von IEC 61131-Standardbausteinen:** Die eigentliche Konvertierung erfolgt durch `F_WORD_TO_DINT`, der eine definierte und normierte Umrechnung bietet.
- **Unidirektionale Adapter:** Sowohl `AW_IN` als auch `ADI_OUT` sind als unidirektionale Adapter ausgelegt, d. h. sie transportieren Ereignis und Daten nur in eine Richtung.
- **Keine eigenen Zustände:** Als Composite-Baustein besitzt `AW_TO_ADI` keine eigenen internen Zustände – das Verhalten wird vollständig durch die Verschaltung der enthaltenen Elemente bestimmt.

Zustandsübersicht

Da es sich um einen Composite-Baustein ohne eigene Zustandsmaschine handelt, existiert keine separate Zustandsübersicht. Das Laufzeitverhalten wird durch den eingebetteten `F_WORD_TO_DINT` bestimmt, der nach einem Ereignis am Eingang einen Verarbeitungsschritt durchführt und das Ergebnis ausgibt.

Anwendungsszenarien

- **Agrartechnik:** Anbindung von Sensoren oder Aktoren, die WORD-Daten liefern (z. B. über Feldbusse mit 16-Bit-Werten), an Steuerungen, die DINT benötigen.
- **Migration alter Systeme:** Ersatz von WORD-basierten Schnittstellen durch moderne DINT-Komponenten, ohne die gesamte Architektur umzustellen.
- **Adapter-Kapselung:** Einbinden von Bibliotheken oder Modulen, die auf unterschiedliche Datentypen ausgelegt sind, durch einen einheitlichen Adapter-Konverter.

Vergleich mit ähnlichen Bausteinen

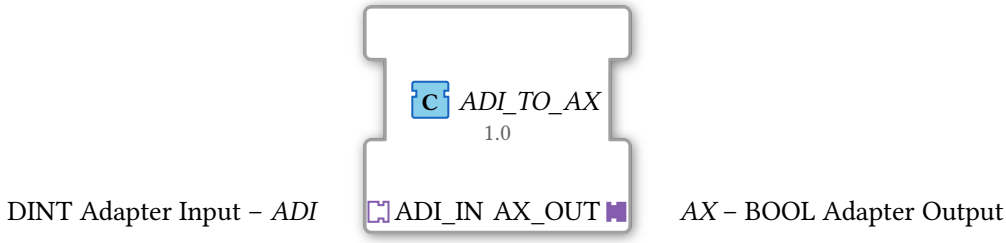
- **Direkte Konvertierung:** Im Gegensatz zu einem einfachen `F_WORD_TO_DINT` bietet `AW_TO_ADI` eine Adapter-basierte Schnittstelle, die eine lose Kopplung zwischen Komponenten ermöglicht.
- **Bidirektionale Konverter:** Es existieren Bausteine, die sowohl Hin- als auch Rückkonvertierung (z. B. WORD ↔ DINT) in einem Baustein vereinen. Der vorliegende Baustein ist bewusst unidirektional ausgelegt.
- **Typkonvertierung über flexible Adapter:** Andere Adapter-Konverter arbeiten mit generischen oder polymorphen Schnittstellen; dieser Baustein ist spezifisch auf WORD-zu-DINT festgelegt.

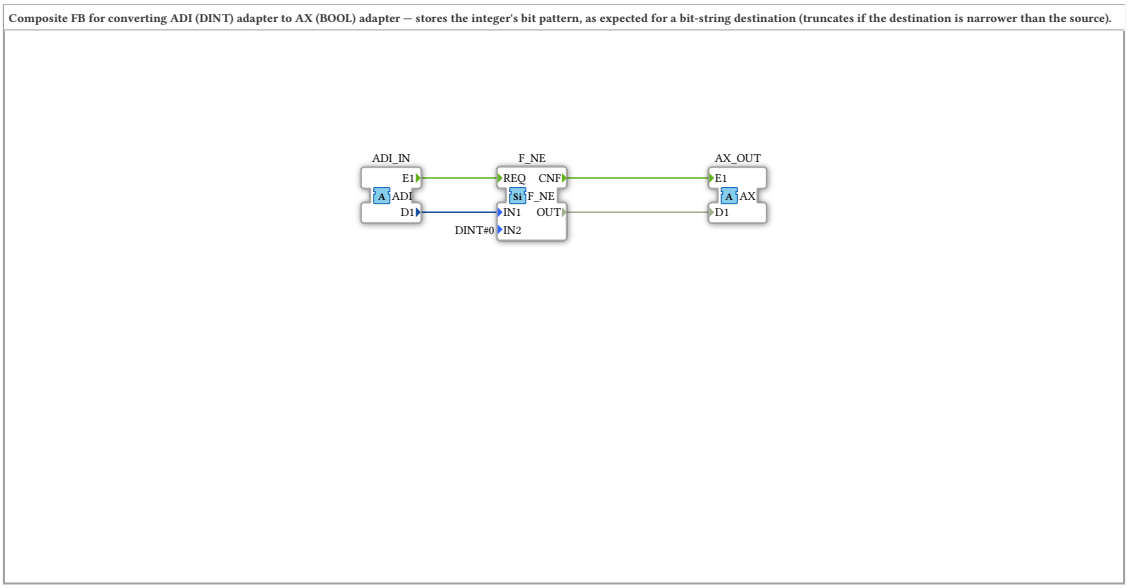
Fazit

Der `AW_TO_ADI` ist ein einfacher, aber effektiver Composite-Baustein zur Umwandlung eines WORD-Adapters in einen DINT-Adapter. Durch die Verwendung eines standardisierten Konvertierungsbausteins und klarer Adapter-Schnittstellen eignet er sich besonders für den Einsatz in modularen Automatisierungslösungen, bei denen Datentypanpassungen zwischen Komponenten erforderlich sind. Seine Struktur ist transparent, leicht nachvollziehbar und erlaubt eine schnelle Integration in bestehende Projekte.

ADI_AX

ADI_TO_AX





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ADI_TO_AX** ist ein Composite-Baustein, der eine einfache Signalumsetzung zwischen zwei Adaptertypen realisiert. Er wandelt einen über den **ADI_IN**-Socket empfangenen DINT-Wert (Typ: *adapter::types::unidirectional::ADI*) in einen BOOL-Wert um und gibt diesen über den **AX_OUT**-Plug (Typ: *adapter::types::unidirectional::AX*) aus. Die Umwandlung erfolgt durch einen Vergleich des empfangenen DINT-Werts mit dem konstanten Wert 0. Ist der Wert ungleich 0, wird der BOOL-Ausgang auf TRUE gesetzt; bei Gleichheit auf FALSE. Der Baustein eignet sich vor allem für die Anbindung von digitalen Signalquellen, die als Ganzzahlen codiert sind, an boolesche Schnittstellen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Beschreibung
(keine expliziten Ereignis-Eingänge)	Der FB nutzt ausschließlich die über den ADI_IN-Adapter angebundenen Ereignisse. Tatsächlich wird das Ereignis `E1` des ADI_IN-Adapters intern auf den Funktionsbaustein `F_NE` weitergeleitet.

Ereignis-Ausgänge

Name	Beschreibung

(keine expliziten Ereignis-Ausgänge)	Der Ausgang erfolgt über den AX_OUT-Adapter; das Ereignis `E1` des AX_OUT-Adapters wird nach erfolgreicher Verarbeitung des `F_NE`-Aufrufs gesendet.

Daten-Eingänge

Name	Typ	Beschreibung
(über ADI_IN-Adapter)	`DINT`	Der über den Adapter-Socket angebundene DINT-Wert (z. B. Messwert oder Statuscode) wird als Eingang `D1` des ADI_IN-Adapters empfangen.

Daten-Ausgänge

Name	Typ	Beschreibung
(über AX_OUT-Adapter)	`BOOL`	Der Ausgangswert wird als `D1` des AX_OUT-Adapters bereitgestellt. TRUE, wenn der empfangene DINT-Wert ungleich `0` ist; sonst FALSE.

Adapter

Adapter	Name	Richtung	Typ	Beschreibung
Socket	ADI_IN	Eingang	`adapter::types::unidirectional::ADI`	Nimmt einen DINT-Wert (mit zugehörigem Ereignis) entgegen.
Plug	AX_OUT	Ausgang	`adapter::types::unidirectional::AX`	Gibt den resultierenden BOOL-Wert (mit zugehörigem Ereignis) weiter.

Funktionsweise

Der Baustein besteht intern aus einem einzigen Funktionsbaustein **F_NE** des Typs `iec61131::comparison::F_NE`. Dieser implementiert die IEC-61131-Funktion „Ungleich“ (NE – Not Equal). Das Verhalten:

1. Ein eingehendes Ereignis **E1** des ADI_IN-Adapters wird an den **REQ**-Eingang des **F_NE** weitergeleitet.
2. Der Datenwert **D1** des ADI_IN-Adapters wird an den **IN1**-Eingang des **F_NE** gelegt. Der zweite Vergleichswert **IN2** ist fest auf den Wert `DINT#0` eingestellt.
3. Der **F_NE** vergleicht `IN1 != IN2`. Das Ergebnis (BOOL) wird am Ausgang **OUT** ausgegeben.
4. Nach Beendigung der Verarbeitung signalisiert **F_NE.CNF** die Fertigstellung. Dieses Ereignis wird an den **E1**-Eingang des AX_OUT-Adapters gesendet, wodurch der Ausgangswert (ebenfalls das **OUT**-Signal) an **D1** des AX_OUT-Adapters weitergegeben wird.

Damit realisiert der FB eine einfache „Schwellwertlogik“: Jeder von Null verschiedene DINT-Wert wird als logisch „Wahr“ interpretiert. Der Baustein arbeit dabei strikt ereignisgesteuert – jedes eingehende Ereignis löst genau eine Vergleiche- und Ausgabeaktion aus.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Adapter-Kopplung:** Der FB ist als Composite-Baustein konzipiert, der zwei unidirektionale Adapter miteinander verbindet. Er kann daher nahtlos in eine Adapter-basierte Kommunikationsarchitektur eingebunden werden.
- **Keine eigenen Zustände:** Da der FB keinerlei interne Zustandsvariablen oder Timer besitzt, ist sein Verhalten rein kombinatorisch und deterministisch.
- **Direkte Impedanz:** Der verwendete Baustein `F_NE` aus der IEC-61131-Bibliothek ist hochperformant und benötigt keine dynamische Speicherverwaltung.

- **Typsicherheit:** Die Konstante `DINT#0` wird explizit als DINT deklariert, sodass keine impliziten Typkonvertierungen auftreten.
- **Lizenzhinweis:** Der Baustein unterliegt der Eclipse Public License 2.0 (EPL-2.0), Copyright 2026 HR Agrartechnik GmbH.

Zustandsübersicht

Der FB besitzt keine eigenen Zustände, da er vollständig auf dem internen funktionalen Block **F_NE** aufbaut. Das Verhalten ist daher ohne Verzögerungen oder Speicherung auskommend. Der Ablauf ist:

- **Warten auf Ereignis:** Inaktiv, bis ein Ereignis über den ADI_IN-Adapter eintrifft.
- **Verarbeitung:** Nach Ereigniseingang wird der Vergleich ausgeführt; der Ausgang aktualisiert sich sofort. Nach dem Vergleich wird das Ausgangsereignis gesendet.

Anwendungsszenarien

- **Digitalisierung von analogen Werten:** In der Automatisierung werden häufig DINT-Werte (z. B. Zählerstände, Fehlercodes) als boolesche Signale interpretiert. Der FB eignet sich zur Umwandlung von „ungleich Null"-Bedingungen in binäre Signale.
- **Adapter-Brückenbildung:** Wenn eine Komponente nur einen ADI-Adapter bereitstellt, die nachgeschaltete Logik aber nur AX-Adapter akzeptiert, kann dieser FB als verlustfreier Konverter eingesetzt werden.
- **Einfache Schaltschwellen:** Mit minimalem Aufwand lässt sich feststellen, ob ein Wert von Null abweicht – nützlich z. B. für Überwachung von Zählimpulsen oder Grenzwertverletzungen.

Vergleich mit ähnlichen Bausteinen

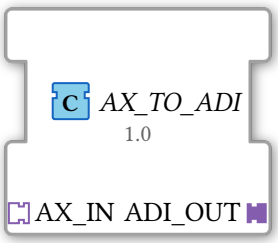
Baustein	Typ	Funktion
ADI_TO_AX (dieser FB)	Composite	DINT ≠ 0 ⟶ BOOL
`BOOL_TO_DINT` (Standard-Bibliothek)	Funktion	Direkte Konvertierung von BOOL zu DINT (z. B. TRUE ⟶ 1)
`F_NE` (IEC 61131)	Funktion	Allgemeiner Ungleich-Vergleich beliebiger Datentypen
Eigener Adapter-Konverter	–	Kann einfach durch andere Vergleichsfunktionen (F_EQ, F_GT, etc.) erweitert werden

Im Vergleich zu einer reinen BOOL-zu-DINT-Konvertierung bietet **ADI_TO_AX** den Vorteil, dass der DINT-Wert exakt beibehalten wird (z. B. 42 ⟶ TRUE), während eine boolesche Konvertierung nur zwischen TRUE/FALSE und 1/0 unterscheiden kann.

Fazit

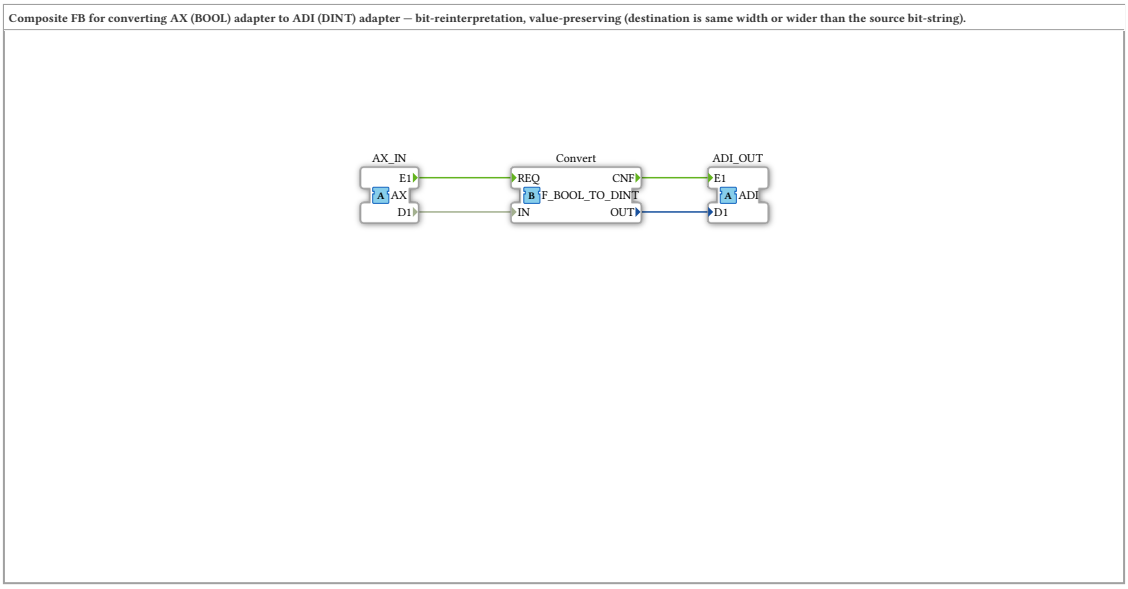
Der Funktionsblock **ADI_TO_AX** ist ein schlanker und robuster Adapter-Konverter für unidirektionale Schnittstellen. Er nutzt den bewährten IEC-61131-Vergleichsbaustein `F_NE` und setzt damit eine einfache, aber nützliche Logik um: „Jeder Wert ungleich Null ist wahr." Der FB eignet sich besonders für den Einsatz in modularen Automatisierungssystemen, bei denen unterschiedliche Adapter-Standards aufeinander abgestimmt werden müssen. Dank der Composite-Struktur kann er leicht in eigene Netzwerke integriert und bei Bedarf erweitert werden.

AX_TO_ADI



BOOL Adapter Input – *AX*

ADI – DINT Adapter Output



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock **AX_TO_ADI** ist ein zusammengesetzter Baustein (Composite FB), der einen AX-Adapter mit boolescher Schnittstelle (BOOL) in einen ADI-Adapter mit ganzzahliger Schnittstelle (DINT) umwandelt. Er dient als Konverter zwischen unidirektionalen Adaptern und ermöglicht die nahtlose Integration von binären Signalen in Systeme, die integerbasierte Adapter verwenden.

Schnittstellenstruktur

Da es sich um einen Composite FB handelt, werden alle Ein- und Ausgänge über die enthaltenen Adapter realisiert. Die folgende Aufstellung zeigt die logischen Schnittstellen.

Ereignis-Eingänge

Der Baustein besitzt keine direkten Ereignis-Eingänge. Die Ereignissteuerung erfolgt über den **AX_IN** Adapter:

- **E1** (BOOT): Ereigniseingang des AX-Adapter, der die Datenkonvertierung auslöst.

Ereignis-Ausgänge

Der Baustein besitzt keine direkten Ereignis-Ausgänge. Die Ereignisweiterleitung erfolgt über den **ADI_OUT** Adapter:

- **E1** (CNF): Ereignisausgang des ADI-Adapter, signalisiert die Fertigstellung der Konvertierung.

Daten-Eingänge

Der Baustein besitzt keine direkten Daten-Eingänge. Die Datenaufnahme erfolgt über den **AX_IN** Adapter:

- **D1** (BOOL): Boolescher Eingangswert, der in einen DINT-Wert umgewandelt wird.

Daten-Ausgänge

Der Baustein besitzt keine direkten Daten-Ausgänge. Die Datenausgabe erfolgt über den **ADI_OUT** Adapter:

- **D1** (DINT): Konvertierter ganzzahliger Ausgangswert (0 bei FALSE, 1 bei TRUE).

Adapter

- **AX_IN** (Socket): Empfängt einen AX-Adapter (Typ: `adapter::types::unidirectional::AX`) mit einem booleschen Daten- und Ereignisknal.
- **ADI_OUT** (Plug): Stellt einen ADI-Adapter (Typ: `adapter::types::unidirectional::ADI`) mit einem DINT-Daten- und Ereignisknal zur Verfügung.

Funktionsweise

Der Baustein nutzt intern den IEC 61131-3 Konverter `F_BOOL_TO_DINT`. Sobald am **AX_IN.E1** ein Ereignis eintrifft, wird der aktuelle Wert von **AX_IN.D1** ausgelesen und dem Konverter übergeben. Der Konverter setzt bei FALSE den Wert 0 und bei TRUE den Wert 1. Nach Abschluss der Konvertierung wird der Ergebniswert an **ADI_OUT.D1** weitergegeben und gleichzeitig ein Ereignis an **ADI_OUT.E1** gesendet.

Die Ereigniskette ist damit vollständig durchgeschliffen: `AX_IN.E1` → Konvertierung → `ADI_OUT.E1`.

Technische Besonderheiten

- **Bit-Reinterpretation, werterhaltend** (Zielbreite ist gleich groß oder größer als der Quell-Bit-String).
- **Reine Datenkonvertierung:** Der Baustein führt keine Zustandslogik oder zeitliche Verzögerung durch – jede Ereignisaktivierung am Eingang erzeugt sofort eine Ausgabe.
- **Verwendung von IEC 61131-Bibliotheken:** Die Konvertierung basiert auf dem standardisierten Funktionsbaustein `F_BOOL_TO_DINT` aus der IEC 61131-Konvertierungsbibliothek.
- **Unidirektionale Adapter:** Sowohl AX als auch ADI sind unidirektionale Adapter, d.h. sie unterstützen nur eine Richtung (hier Eingang bzw. Ausgang). Dies vereinfacht die Datenflusssteuerung.
- **Composite FB ohne eigenen Zustand:** Als Composite FB besitzt der Baustein keinen internen Zustandsautomaten; die gesamte Logik wird durch den enthaltenen Funktionsbaustein realisiert.

Zustandsübersicht

Als Composite FB hat **AX_TO_ADI** keinen eigenen Zustandsautomaten. Das Verhalten wird ausschließlich vom eingebetteten `F_BOOL_TO_DINT` bestimmt, der keine zustandsabhängige Logik besitzt. Die Ausgabe ist stets eine direkte Funktion des Eingangs zum Zeitpunkt der Ereignisaktivierung.

Anwendungsszenarien

- **Umwandlung von Binärsignalen:** Wenn ein Sensor oder Schalter ein boolesches Signal über einen AX-Adapter liefert, das in einem System mit ADI-Adaptern (DINT) weiterverarbeitet werden soll.
- **Gateway-Funktion:** Verbindung zwischen Teilsystemen, die unterschiedliche Adaptertypen verwenden, ohne die zugrundeliegende Logik zu ändern.
- **Test und Simulation:** Erzeugung eines definierten DINT-Werts aus einem booleschen Eingang für Testzwecke.

Vergleich mit ähnlichen Bausteinen

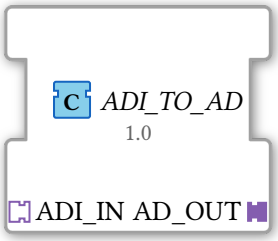
- **AX_TO_ADI (dieser FB):** Konvertiert BOOL → DINT über Adapter.
- **ADI_TO_AX (hypothetisch):** Würde DINT → BOOL konvertieren, z.B. durch Vergleich mit Null oder Schwellwert.
- **Direkter Befehl F_BOOL_TO_DINT:** Dieser Baustein ist eine gekapselte Version, die speziell für die Arbeit mit unidirektionalen Adaptern ausgelegt ist. Im Gegensatz zum Basisbaustein sind die Adapter-Schnittstellen bereits integriert.

Fazit

AX_TO_ADI ist ein einfacher, aber nützlicher Konverterbaustein, der boolesche Adaptersignale in integerbasierte Adaptersignale überführt. Er erweitert die Interoperabilität zwischen verschiedenen Komponenten eines Systems und ermöglicht eine saubere Trennung von Daten- und Ereignisflüssen. Aufgrund seiner geringen Komplexität ist er schnell einsetzbar und gut in größere Composite FB-Ketten integrierbar.

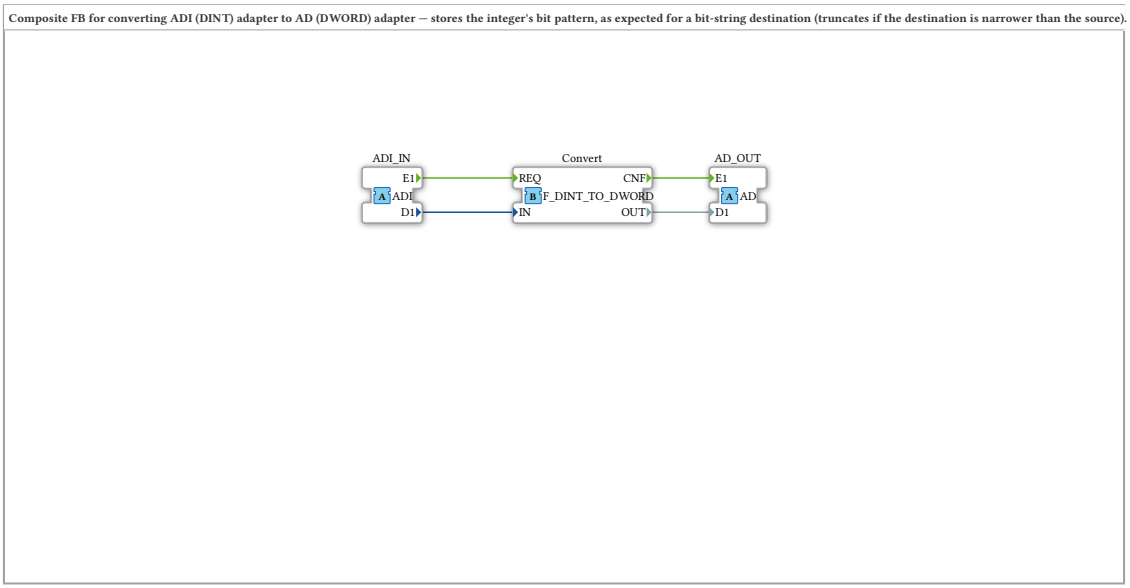
AD_ADI

ADI_TO_AD



DINT Adapter Input – *ADI*

AD – DWORD Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ADI_TO_AD** ist ein Composite-Baustein (zusammengesetzter FB), der eine Schnittstellenkonvertierung von einem **ADI**-Adapter (DINT-basiert) zu einem **AD**-Adapter (DWORD-basiert) realisiert. Er fungiert als unidirektionaler Wandler und ermöglicht die nahtlose Integration von Komponenten mit unterschiedlichen Datenadapter-Typen in der 4diac-IDE.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine eigenständigen Ereignis-Eingänge auf Ebene der Schnittstelle. Die Ereignissteuerung erfolgt ausschließlich über den angeschlossenen **Socket-Adapter ADI_IN**:

- **ADI_IN.E1** – Ereignis, das die Datenkonvertierung anstößt.

Ereignis-Ausgänge

Auch auf dieser Ebene gibt es keine separaten Ereignis-Ausgänge. Der Ausgang wird über den **Plug-Adapter AD_OUT** bereitgestellt:

- **AD_OUT.E1** – Ereignis, das nach erfolgreicher Konvertierung ausgelöst wird.

Daten-Eingänge

- **ADI_IN.D1** – Eingabe des zu konvertierenden DINT-Werts (Quelladapter).

Daten-Ausgänge

- **AD_OUT.D1** – Ausgabe des konvertierten DWORD-Werts (Zieladapter).

Adapter

- **ADI_IN** (Socket) – vom Typ `adapter::types::unidirectional::ADI` (DINT-Daten).
- **AD_OUT** (Plug) – vom Typ `adapter::types::unidirectional::AD` (DWORD-Daten).

Beide Adapter sind unidirektional ausgelegt (je ein Ereignis- und ein Datenkanal).

Funktionsweise

Der FB enthält einen internen Baustein `F_DINT_TO_DWORD` aus der IEC 61131-Bibliothek, der die eigentliche Typumwandlung vornimmt. Die Ablaufsteuerung ist wie folgt:

1. Ein eingehendes Ereignis am **ADI_IN.E1** triggert den **Convert.REQ**-Eingang des internen Konverters.
2. Der Konverter wandelt den über **ADI_IN.D1** anliegenden DINT-Wert in einen DWORD-Wert um.
3. Nach Abschluss der Konvertierung signalisiert **Convert.CNF** den erfolgreichen Abschluss und feuert das Ereignis am **AD_OUT.E1**, gleichzeitig wird der konvertierte Wert an **AD_OUT.D1** ausgegeben.

Damit wird eine transparente, ereignisgesteuerte Konvertierung zwischen den Adaptertypen realisiert.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Typkonvertierung**: Die eigentliche Umwandlung erfolgt über den standardisierten IEC 61131-Baustein `F_DINT_TO_DWORD`, der eine bitidentische Übertragung des Integer-Werts in den DWORD-Raum sicherstellt (keine logische Interpretation, nur Typanpassung).
- **Unidirektionalität**: Der FB ist strikt unidirektional aufgebaut und erwartet genau einen Eingangs- und einen Ausgangsadapter. Eine Rückwärtskonvertierung ist nicht vorgesehen.
- **Paketstruktur**: Der FB ist im Paket `adapter::conversion::unidirectional` organisiert und importiert die entsprechende IEC-Bibliothek.

Zustandsübersicht

Als Composite-Baustein besitzt **ADI_TO_AD** keine eigenen expliziten Zustände. Sein Verhalten wird vollständig durch den inneren Konverter bestimmt:

- **Idle**: Wartet auf ein Ereignis am ADI_IN.
- **Umwandlung aktiv**: Wenn `Convert.REQ` gesetzt ist.
- **Abschluss**: Nach `Convert.CNF` wird das Ergebnis am Ausgang bereitgestellt.

Anwendungsszenarien

- **Schnittstellenanpassung**: Wenn eine Komponente einen DINT-Adapter bereitstellt, aber ein nachgeschaltetes Modul einen DWORD-Adapter erwartet (z. B. bei der Integration von Steuerungen mit unterschiedlichen Datenbreiten).
- **Systemmodernisierung**: Alte DINT-basierte Sensoren oder Aktoren (über ADI) sollen mit neuen DWORD-fähigen Bussystemen (über AD) verbunden werden.
- **Prototypische Entwicklung**: Schnelles Umschalten zwischen Adaptervarianten ohne Änderung der Applikationslogik.

Vergleich mit ähnlichen Bausteinen

- **Direkte Konverter-FBs**: Im Gegensatz zu reinen Typkonvertern (wie `F_DINT_TO_DWORD`) arbeitet **ADI_TO_AD** auf Adapterebene und integriert die Ereignissteuerung sowie die Schnittstellenanpassung in einem Baustein.
- **Bidirektionale Adapter**: Es existieren auch Bausteine, die eine Konvertierung in beide Richtungen anbieten (z. B. ADI_TO_AD und AD_TO_ADI kombiniert). Der vorliegende FB ist auf die eine Richtung spezialisiert und daher schlanker.
- **Manuelle Verkabelung**: Ohne diesen FB müsste man die Adapter intern manuell verschalten und die Ereignissignale separat führen – der Composite-Baustein kapselt diese Komplexität.

Fazit

Der **ADI_TO_AD**-Funktionsblock ist ein nützlicher und übersichtlicher Composite-Baustein zur einfachen Anpassung zwischen DINT- und DWORD-basierten Adaptern. Er reduziert den manuellen Verschaltungsaufwand, nutzt bewährte IEC-Konvertierungslogik und fügt sich nahtlos in unidirektionale

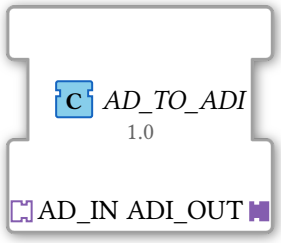
Datenflüsse ein. Ideal für Entwickler, die schnell und sauber zwischen Adaptertypen wechseln müssen, ohne tief in die interne Signalverarbeitung eingreifen zu wollen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

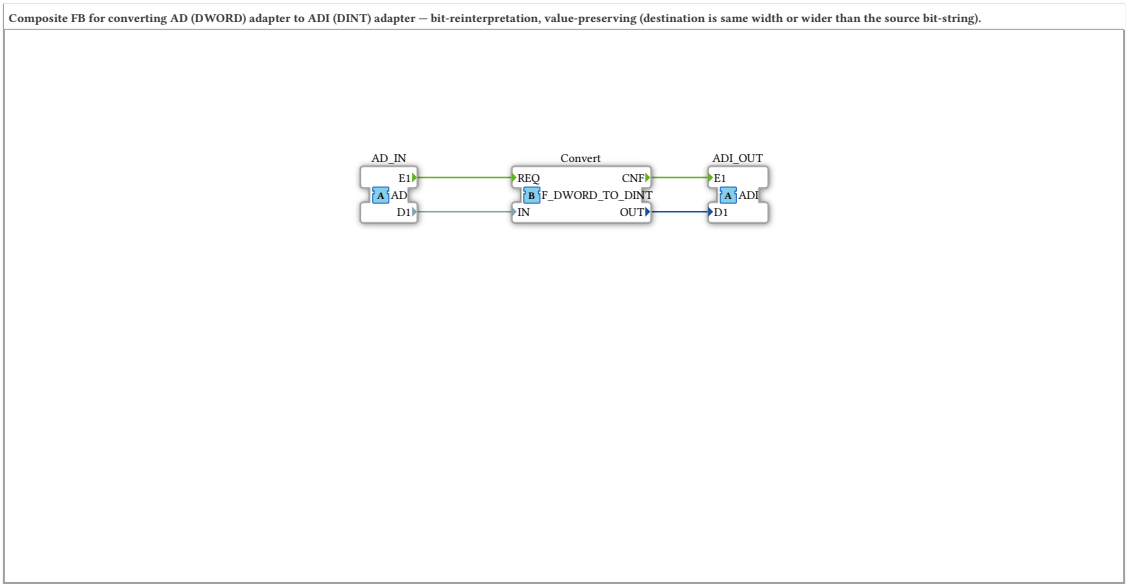
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AD_TO_ADI

DWORD Adapter Input – *AD*



ADI – DINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AD_TO_ADI** ist ein Composite-Baustein zur Umwandlung eines DWORD-Adapters (AD) in einen DINT-Adapters (ADI). Er ermöglicht die nahtlose Integration von Komponenten unterschiedlicher Datentypen in einer IEC 61499-Applikation, ohne dass der Anwender die Konvertierungslogik selbst implementieren muss.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge. Die Ereignissteuerung erfolgt über den Socket-Adapter `AD_IN`.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge. Die Ereignisausgabe erfolgt über den Plug-Adapter `ADI_OUT`.

Daten-Eingänge

Keine direkten Daten-Eingänge. Die Datenübergabe erfolgt über den Socket-Adapter `AD_IN`.

Daten-Ausgänge

Keine direkten Daten-Ausgänge. Die Datenweitergabe erfolgt über den Plug-Adapter `ADI_OUT`.

Adapter

Bezeichnung	Typ	Richtung	Kommentar
AD_IN	adapter::types::unidirectional::AD	Socket (Eingang)	DWORD Adapter Input – liefert ein DWORD-Datum und ein Ereignis
ADI_OUT	adapter::types::unidirectional::ADI	Plug (Ausgang)	DINT Adapter Output – liefert ein konvertiertes DINT-Datum und ein Ereignis

Funktionsweise

Der Baustein kapselt einen Aufruf der IEC-61131-3-Konvertierungsfunktion `F_DWORD_TO_DINT`. Der Ablauf ist:

1. Ein Ereignis am Socket `AD_IN.E1` triggert den Konvertierungs-FB `Convert` (Ereignis `REQ`).
2. Gleichzeitig wird das DWORD-Datum `AD_IN.D1` an den Eingang `Convert.IN` übergeben.
3. Der Konvertierungs-FB wandelt den Wert in einen DINT um und signalisiert die Fertigstellung über das Ereignis `CNF`.
4. Dieses Ereignis wird an den Plug `ADI_OUT.E1` weitergeleitet, und der konvertierte DINT-Wert (`Convert.OUT`) wird an `ADI_OUT.D1` gelegt.

Dadurch verhält sich der Composite-FB aus Sicht der Applikation wie ein direkter Adapter-Konverter ohne Nebenwirkungen.

Technische Besonderheiten

- **Bit-Reinterpretation, werterhaltend** (Zielbreite ist gleich groß oder größer als der Quell-Bit-String).
- **IEC-61131-3-Kompatibilität** – Die Konvertierung erfolgt via der standardisierten Bibliotheksfunktion `F_DWORD_TO_DINT`, die für gängige SPS-Systeme verfügbar ist.
- **Unidirektionale Adapter** – Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional ausgelegt; Rückmeldungen oder Gegenrichtungsdaten werden nicht unterstützt.
- **Keine Zustandsinformationen** – Der Composite selbst besitzt keine internen Zustandsvariablen; jede Konvertierung ist atomar und zeitlich unabhängig.
- **Autonomie** – Der Baustein kann ohne manuelle Einbindung weiterer FB in einer Netzwerkkonfiguration verwendet werden.

Zustandsübersicht

Der Composite-FB besitzt keinen eigenen Zustandsautomaten. Das Verhalten ergibt sich rein aus der sequenziellen Abarbeitung der internen Verbindungen:

- **Warten auf Ereignis** – keine Aktivität, solange kein Ereignis an `AD_IN.E1` anliegt.
- **Konvertierung** – nach Ereigniseingang erfolgt die einmalige Umwandlung und Ausgabe des Ergebnisses.
- **Abschluss** – der FB ist sofort wieder bereit für das nächste Ereignis.

Anwendungsszenarien

- **Gateway zwischen Systemen unterschiedlicher Datenbreite** – z. B. Anbindung eines DWORD-Sensorwertes an eine DINT-basierte Steuerungslogik.
- **Adapter-Brücke in IEC 61499-Architekturen** – wenn Komponenten über die standardisierten unidirektionalen Adapter (AD/ADI) kommunizieren müssen.
- **Kapselung von Plausibilitätsprüfungen** – der DWORD-Wert kann vor der Konvertierung in einer vorgelagerten Logik validiert werden, bevor er in den DINT-Bereich überführt wird.

Vergleich mit ähnlichen Bausteinen

- **AD_TO_ADI** ist spezialisiert auf die Konvertierung von DWORD nach DINT und folgt einem strengen unidirektionalen Adapter-Profil.
- Ähnliche Bausteine wie `AD_TO_AI` (DWORD nach INT) oder `ADI_TO_AD` (Rückkonvertierung) existieren für andere Datentypen, teilen aber das gleiche Entwurfsmuster.

- Im Gegensatz zu einem generischen Konvertierungs-FB (z. B. `F_DWORD_TO_DINT` allein) bietet der Composite die direkte Adapter-Einbindung, sodass der Anwender keine manuellen Adapter-Schnittstellen erstellen muss.

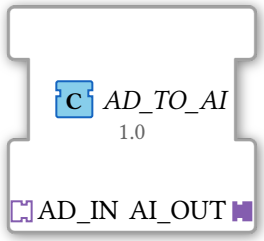
Fazit

Der Composite-FB **AD_TO_ADI** stellt eine einfache, robuste und standardkonforme Lösung zur Umwandlung von DWORD- in DINT-Adapterdaten bereit. Er reduziert den Konfigurationsaufwand innerhalb von IEC-61499-Netzwerken und erhöht die Wiederverwendbarkeit von Bausteinen aus unterschiedlichen Quellen. Dank der rein funktionalen Umsetzung ohne Zustandsprobleme eignet er sich besonders für zeitkritische und sicherheitsgerichtete Anwendungen.

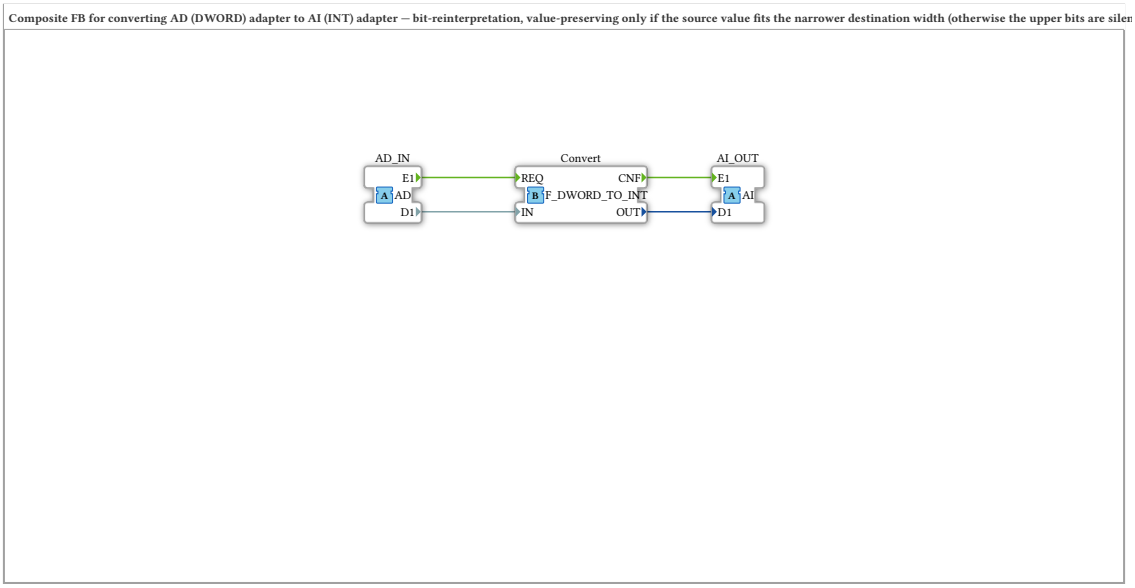
AD_AI

AD_TO_AI

DWORD Adapter Input – *AD*



AI – INT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **AD_TO_AI** dient als Konverter zwischen zwei unterschiedlichen Adapterschnittstellen: Er wandelt einen **AD-Adapter (DWORD)** in einen **AI-Adapter (INT)** um. Der Baustein wird als Composite FB realisiert und nutzt intern die IEC 61131-3 Funktion `F_DWORD_TO_INT`, um die eigentliche Typumwandlung durchzuführen. Ziel ist es, die Integration von Komponenten zu erleichtern, die auf verschiedenen Datentypen basieren, und dabei die ereignisgesteuerte Datenweitergabe innerhalb der 4diac-IDE zu gewährleisten.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine direkten Ereignis-Eingänge auf oberster Ebene. Die Ereignissteuerung erfolgt über die angeschlossenen Adapter:

- **AD_IN.E1** (Ereignis des Socket-Adapters) – löst die Konvertierung aus.

Ereignis-Ausgänge

Auch diese werden ausschließlich über die Adapter bereitgestellt:

- **AI_OUT.E1** (Ereignis des Plug-Adapters) – signalisiert die Fertigstellung der konvertierten Daten.

Daten-Eingänge

- **AD_IN.D1** (DWORD) – Der ursprüngliche Datenwert, der vom AD-Adapter geliefert wird.

Daten-Ausgänge

- **AI_OUT.D1** (INT) – Der konvertierte Wert, der über den AI-Adapter nach außen gegeben wird.

Adapter

Adaptertyp	Richtung	Typ	Beschreibung
Socket	Eingang	`adapter::types::unidirectional::AD`	Empfängt einen DWORD-Wert (über Ereignis E1 und Daten D1).
Plug	Ausgang	`adapter::types::unidirectional::AI`	Stellt den konvertierten INT-Wert bereit (Ereignis E1, Daten D1).

Funktionsweise

Der Baustein arbeitet ereignisgesteuert:

1. Ein Ereignis am Socket **AD_IN.E1** triggert die interne Instanz `Convert` (Typ `F_DWORD_TO_INT`) über deren Ereigniseingang **REQ**.
2. Gleichzeitig wird der Datenwert **AD_IN.D1** (DWORD) an den Dateneingang **IN** von `Convert` übergeben.
3. Die Funktion `F_DWORD_TO_INT` führt die Typumwandlung von DWORD nach INT durch.
4. Nach Abschluss der Berechnung sendet `Convert` ein Ereignis an ihrem Ausgang **CNF**, welches mit dem Plug-Ereignis **AI_OUT.E1** verbunden ist.
5. Parallel dazu wird der konvertierte INT-Wert vom Ausgang **OUT** von `Convert` an den Datenausgang **AI_OUT.D1** weitergeleitet.

Somit wird nach jedem eingehenden Ereignis am Socket automatisch ein konvertierter Datenwert auf dem Plug bereitgestellt.

Technische Besonderheiten

- **Bit-Reinterpretation**, werterhaltend nur, wenn der Quellwert in die schmalere Zielbreite passt (sonst werden die oberen Bits stillschweigend verworfen).
- **Komposition:** Der FB ist ein reiner Composite-Baustein – er enthält keine eigene Logik, sondern delegiert die Umwandlung an den Standardbaustein `F_DWORD_TO_INT` aus der IEC-61131-Bibliothek.
- **Datentypen:** DWORD ist ein 32-Bit-Wert, INT dagegen ein 16-Bit-Integer (vorzeichenbehaftet). Bei der Konvertierung können Datenverluste auftreten, wenn der DWORD-Wert außerhalb des INT-Bereichs (−32768 … 32767) liegt oder nicht in diesen Bereich passt. Der Anwender muss dies bei der Verwendung beachten.
- **Adapter-Kopplung:** Die Verbindung zwischen Socket und Plug erfolgt ausschließlich über die inneren Ereignis- und Datenflüsse. Die Adapter selbst sind vom Typ `unidirectional`, d. h. sie übertragen Daten nur in eine Richtung.
- **Paketstruktur:** Der Baustein wird im Paket `adapter::conversion::unidirectional` bereitgestellt und importiert die Konvertierungsfunktion aus `iec61131::conversion`.

Zustandsübersicht

Da es sich um einen Composite-FB ohne eigene Zustandsmaschine handelt, gibt es keine expliziten Zustände. Der Baustein verhält sich wie ein transparenter Konverter:

- **Idle:** Wartet auf ein Ereignis am Socket.
- **Processing:** Nach Ereigniseingang wird die Konvertierung gestartet; in dieser Phase sind keine weiteren Ereignisse aktiv.

- **Done:** Nach der Konvertierung wird das Ausgangsereignis ausgegeben; der Baustein kehrt in den Idle-Zustand zurück.

Der interne FB `F_DWORD_TO_INT` besitzt eine eigene, vom System definierte Zustandslogik, die jedoch durch die Kapselung im Composite-FB nicht sichtbar ist.

Anwendungsszenarien

- **Integration unterschiedlicher Adaptertypen:** Ein Sensor liefert Daten als DWORD über einen AD-Adapter, nachgeschaltete Logik erwartet jedoch INT-Werte über einen AI-Adapter. Der Baustein übernimmt die Typumwandlung nahtlos.
- **Heterogene Systemarchitekturen:** In Anlagen, in denen verschiedene Komponenten unterschiedliche Datentypen verwenden, ermöglicht `AD_TO_AI` eine einfache und standardisierte Anbindung.
- **Prototyping und Migration:** Beim Wechsel von einer DWORD-basierten Datenverarbeitung auf INT-basierte Komponenten kann der Baustein als temporärer oder dauerhafter Konverter eingesetzt werden.

Vergleich mit ähnlichen Bausteinen

Es existieren weitere Konverter-Bausteine wie `AD_TO_AI`, `AI_TO_AD`, `INT_TO_DWORD` usw. Der vorliegende Baustein fokussiert speziell auf die Adapterebene und arbeitet vollständig ereignisgesteuert. Im Gegensatz zu einfachen Daten-Konvertierungs-FBs (z. B. `F_DWORD_TO_INT`) bietet er eine vorgefertigte Adapterschnittstelle, die eine direkte Kopplung an Adapter-orientierte Architekturen ermöglicht, ohne dass zusätzliche Verbindungen von Hand gezeichnet werden müssen.

Andere ähnliche Composite-Bausteine unterscheiden sich hauptsächlich in der Richtung (z. B. `AI_TO_AD` für die Rückwandlung) oder in den verwendeten Datentypen.

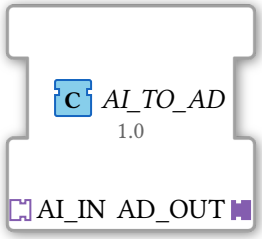
Fazit

Der Composite-FB **AD_TO_AI** ist ein nützliches Werkzeug zur Typanpassung zwischen DWORD- und INT-Adaptern. Er kapselt die Konvertierungslogik in einer sauberen, ereignisgesteuerten Schnittstelle und erleichtert so die Wiederverwendbarkeit und Integration in bestehende 4diac-Projekte. Seine einfache Struktur und die Verwendung eines bewährten Standardbausteins machen ihn zuverlässig und leicht verständlich. Anwender sollten jedoch stets die Grenzen des INT-Datentyps im Auge behalten, um unbeabsichtigte Datenverluste zu vermeiden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

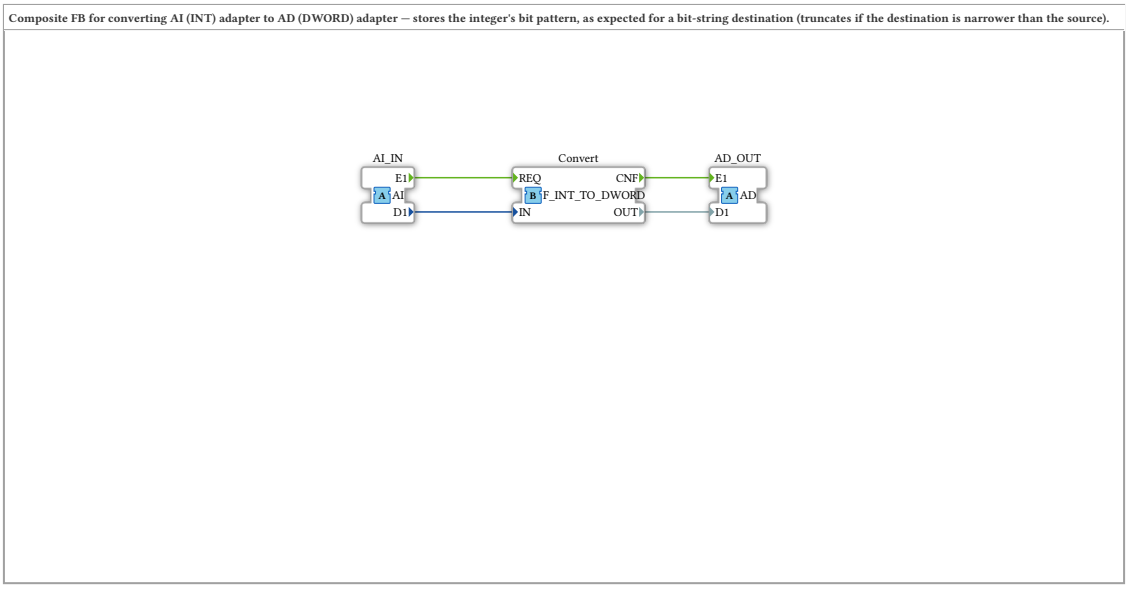
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AI_TO_AD



INT Adapter Input – *AI*

AD – DWORD Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Composite-Funktionsbaustein **AI_TO_AD** dient der Konvertierung eines INT-basierten Adapter-Interface (AI) in ein DWORD-basiertes Adapter-Interface (AD). Er kapselt die Typumwandlung von `INT` nach `DWORD` und ermöglicht so die nahtlose Integration unterschiedlicher Adapter-Typen in Steuerungsanwendungen. Der Baustein ist unidirektional ausgelegt und arbeitet ereignisgesteuert.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine direkten Ereignis-Eingänge. Die ereignisgesteuerte Auslösung erfolgt über den Adapter **AI_IN** (Socket).

- **E1 (über AI_IN)**: Auslöseereignis zur Konvertierung des anliegenden INTEGER-Wertes.

Ereignis-Ausgänge

Der Baustein besitzt keine direkten Ereignis-Ausgänge. Die Bestätigung der Konvertierung erfolgt über den Adapter **AD_OUT** (Plug).

- **E1 (über AD_OUT)**: Bestätigungsereignis nach erfolgreicher Umwandlung.

Daten-Eingänge

Der Baustein besitzt keine direkten Daten-Eingänge. Der zu konvertierende INTEGER-Wert wird über den Adapter **AI_IN** bereitgestellt.

- **D1 (über AI_IN)**: Eingangswert vom Typ `INT`.

Daten-Ausgänge

Der Baustein besitzt keine direkten Daten-Ausgänge. Der konvertierte DWORD-Wert wird über den Adapter **AD_OUT** ausgegeben.

- **D1 (über AD_OUT)**: Ausgangswert vom Typ `DWORD`.

Adapter

- **AI_IN** (Socket) – Eingangsadapter vom Typ `adapter::types::unidirectional::AI`. Er stellt einen INTEGER-Eingang (D1) sowie ein Auslöseereignis (E1) bereit.
- **AD_OUT** (Plug) – Ausgangsadapter vom Typ `adapter::types::unidirectional::AD`. Er stellt einen DWORD-Ausgang (D1) sowie ein Bestätigungsereignis (E1) bereit.

Funktionsweise

Der Baustein ist als Composite ausgeführt. Intern wird der IEC-61131-konforme Funktionsbaustein **F_INT_TO_DWORD** verwendet. Der Ablauf:

1. Ein Ereignis am Eingangsadapter **AI_IN.E1** löst die Datenübernahme aus.
2. Der Wert von **AI_IN.D1** (INT) wird an den Eingang `IN` von `F_INT_TO_DWORD` weitergeleitet.
3. Der interne Baustein führt die Typkonvertierung `INT → DWORD` durch.
4. Das Ergebnis liegt am Ausgang `OUT` von `F_INT_TO_DWORD` an und wird an **AD_OUT.D1** übergeben.
5. Gleichzeitig wird das Ereignis **AD_OUT.E1** ausgelöst, um die Verarbeitung im nachfolgenden Netzwerk zu signalisieren.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Composite-FB**: Die Funktionalität wird durch einen eingebetteten Baustein realisiert, sodass keine eigene Zustandsmaschine (ECC) erforderlich ist.
- **Unidirektional**: Der Datenfluss erfolgt ausschließlich vom Eingangs- zum Ausgangsadapter; eine Rückkopplung ist nicht vorgesehen.
- **Typkonvertierung**: Der Baustein verwendet den standardisierten IEC-61131-Baustein `F_INT_TO_DWORD` und gewährleistet damit Kompatibilität mit vielen Automatisierungssystemen.
- **Adapter-Kapselung**: Die Konvertierungslogik ist vollständig in die Adapter-Schnittstellen eingebettet, was die Wiederverwendung in unterschiedlichen Kontexten vereinfacht.

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Die interne Verarbeitung wird vollständig durch den eingebetteten Funktionsbaustein `F_INT_TO_DWORD` gesteuert, der eine einfache ereignisgesteuerte Datenverarbeitung ohne Zustände realisiert. Ein Fehlerverhalten oder Timeout sind nicht definiert.

Anwendungsszenarien

- **Adapter-Brücke**: Wenn ein Systemkomponente einen INTEGER-Adapter (AI) bereitstellt, die Zielkomponente jedoch einen DWORD-Adapter (AD) erwartet.
- **Datenwort-Erzeugung**: Umrechnung von Analogwerten (z. B. 16-Bit-Messwerte) in ein 32-Bit-Datenwort für Datenübertragung oder Speicherung.
- **Bibliotheks-Unabhängigkeit**: Kapselung der Konvertierungsfunktion, sodass Anwendungsentwickler nicht direkt auf IEC-61131-Bausteine zugreifen müssen.

Vergleich mit ähnlichen Bausteinen

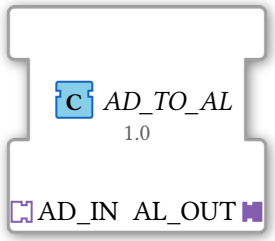
- **F_INT_TO_DWORD**: Die direkte IEC-61131-Funktion bietet die reine Typumwandlung ohne Adapter-Einbettung. `AI_TO_AD` fügt die Adapter-Schnittstellen hinzu und erleichtert so die Integration in adapterbasierte Architekturen.
- **DWORD_TO_INT (umgekehrt)**: Ein entsprechender Baustein für die Rückkonvertierung wäre denkbar, ist aber nicht Teil dieser Implementierung.
- **Eigene Adapter-Wrapper**: Andere Bausteine könnten ähnliche Konvertierungen für andere Datentypen (z. B. `REAL_TO_DWORD`) bereitstellen.

Fazit

AI_TO_AD ist ein kompakter, adapterorientierter Konvertierungsbaustein, der die Umwandlung von INTEGER- nach DWORD-Werten nahtlos in bestehende Steuerungsnetzwerke integriert. Durch die Verwendung standardisierter IEC-61131-Bausteine und die klare Schnittstellendefinition über Adapter eignet er sich besonders für modulare, wiederverwendbare Automatisierungslösungen.

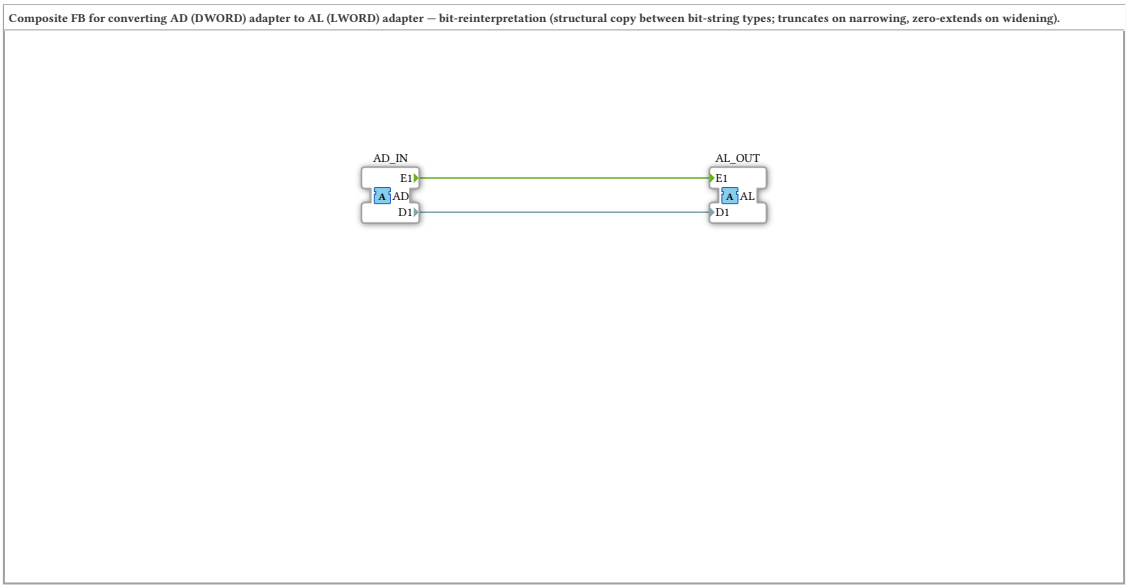
AD_AL

AD_TO_AL



DWORD Adapter Input – *AD*

AL – LWORD Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AD_TO_AL** ist ein Composite-Baustein, der eine unidirektionale Konvertierung eines AD-Adapters (DWORD) in einen AL-Adapter (LWORD) realisiert. Er dient als Brücke zwischen Subsystemen, die unterschiedliche Adaptertypen verwenden, jedoch dieselbe logische Verbindung benötigen. Der Baustein reicht Ereignis- und Datensignale eins zu eins durch und ermöglicht so die nahtlose Integration heterogener Komponenten.

Schnittstellenstruktur

Der Baustein besitzt ausschließlich Adapter-Schnittstellen; separate Ereignis- oder Daten-Ein-/Ausgänge sind nicht vorhanden.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Typ	Richtung	Name	Kommentar
Socket (Eingang)	Input	`AD_IN`	DWORD-Adapter-Eingang vom Typ `adapter::types::unidirectional::AD`
Plug (Ausgang)	Output	`AL_OUT`	LWORD-Adapter-Ausgang vom Typ `adapter::types::unidirectional::AL`

Funktionsweise

Der Baustein ist ein reiner Weiterleitungsbaustein. Innerhalb des Composite-FBs werden die Adapter-Kanäle direkt verbunden:

- **Ereignisverbindung:** `AD_IN.E1` ⟶ `AL_OUT.E1` Jedes eingehende Ereignis am AD-Adapter wird unverändert an den AL-Adapter weitergegeben.
- **Datenverbindung:** `AD_IN.D1` ⟶ `AL_OUT.D1` Der eingehende Datenwert (DWORD) wird ohne Transformation als LWORD an den Ausgang übergeben.

Die Typumwandlung erfolgt implizit durch die Kopplung der Adapter; eine explizite Datentyp-Konvertierung findet nicht statt. Die Laufzeitumgebung sorgt dafür, dass die Werte entsprechend der Adapterdefinition behandelt werden.

Technische Besonderheiten

- **Bit-Reinterpretation** (strukturelle Kopie zwischen Bit-String-Typen; schneidet bei Verengung ab, erweitert mit Nullen bei Erweiterung).
- **Composite-Baustein:** Der FB enthält keine eigene Logik (keine Zustandsmaschine, keine Algorithmen) und besteht ausschließlich aus internen Verbindungen.
- **Unidirektionalität:** Die Adapter sind unidirektional ausgelegt (nur eine Daten- und eine Ereignisrichtung).
- **Paketstruktur:** Der FB ist im Paket `adapter::conversion::unidirectional` abgelegt.
- **Typkonvertierung:** AD (DWORD) und AL (LWORD) sind als eigenständige Adaptertypen definiert; der FB ermöglicht deren Zusammenschaltung ohne manuelle Codierung.
- **Metadaten:** Der Baustein stammt von *HR Agrartechnik GmbH*, Version 1.0, erstellt am 17.02.2026.

Zustandsübersicht

Der Baustein besitzt keinen internen Zustandsautomaten. Der Zustand ist identisch mit dem Zustand der angeschlossenen Adapter. Das Durchreichen erfolgt ohne Verzögerung oder Logik.

Anwendungsszenarien

- **Systemintegration:** Verbindung von Komponenten, die unterschiedliche Adapterprotokolle nutzen (z. B. eine Steuerung mit DWORD-Schnittstelle und ein Buskoppler mit LWORD-Schnittstelle).
- **Testumgebungen:** Einfaches Einschleusen von Testdaten durch Austausch von Adaptern.
- **Modularisierung:** Trennung von Schnittstellendefinition und Datenflusslogik in großen IEC 61499-Projekten.

Vergleich mit ähnlichen Bausteinen

- `AL_TO_AD`: Führt die umgekehrte Konvertierung (LWORD ⟶ DWORD) durch.
- Direkte Adapter-Konvertierungsbausteine (z. B. `BYTE_TO_WORD`) ersetzen in der Regel keine Adapter, sondern wandeln elementare Datentypen um.
- Andere „Composite“-Bausteine im Paket `adapter::conversion` bieten ähnliche Weiterleitungen für andere Adaptertypen (z. B. `AI_TO_AO`, `DI_TO_DO`).

Im Gegensatz zu Bausteinen mit eigener Logik beschränkt sich `AD_TO_AL` auf das reine Durchreichen, was maximale Transparenz und niedrigste Latenz gewährleistet.

Fazit

Der Funktionsblock `AD_TO_AL` ist ein schlanker, aber wertvoller Helfer zur Adapter-Konvertierung in der 4diac-IDE. Er reduziert manuellen Verdrahtungsaufwand und erhöht die Wiederverwendbarkeit von

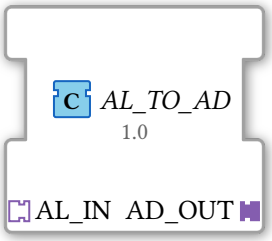
Komponenten, die auf unterschiedliche Adapterstandards setzen. Dank seiner einfachen Struktur ist er robust, performant und leicht verständlich – ideal für die typische Arbeit mit unidirektionalen Adapterflüssen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

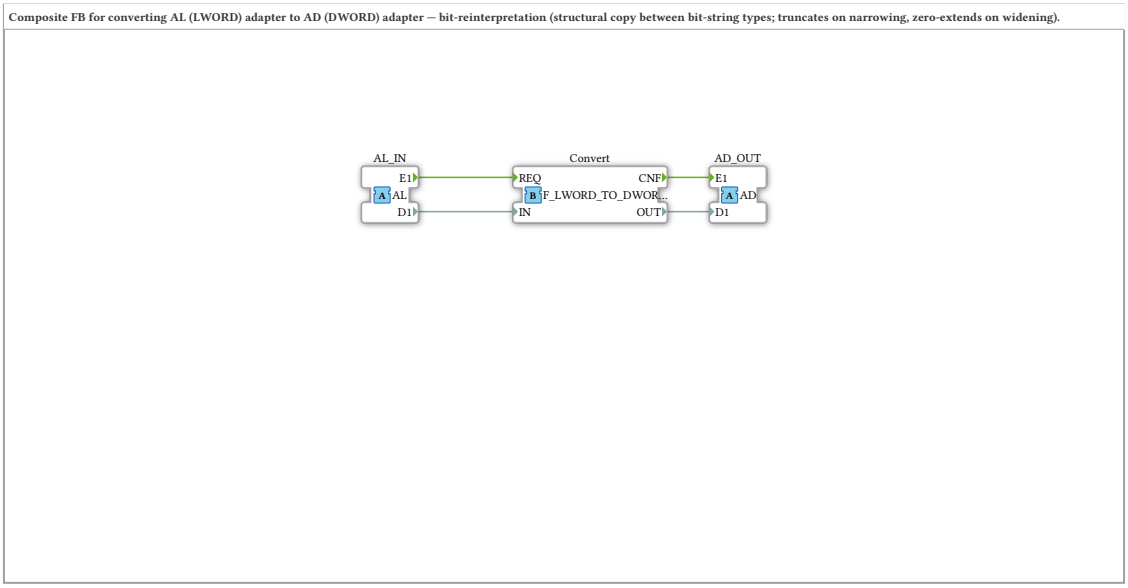
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AL_TO_AD

LWORD Adapter Input – *AL*



AD – DWORD Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AL_TO_AD** ist ein Composite-FB, der einen unidirektionalen LWORD-Adapter (AL) in einen unidirektionalen DWORD-Adapter (AD) umwandelt. Er ermöglicht die Datenkonvertierung zwischen zwei unterschiedlichen Adaptertypen innerhalb der 4diac-IDE, basierend auf der IEC 61499-Norm.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine (der FB besitzt keine eigenen Ereignis-Eingänge; die Ereignissteuerung erfolgt über den AL_IN-Socket)

Ereignis-Ausgänge

- Keine (der FB besitzt keine eigenen Ereignis-Ausgänge; die Ereignisweiterleitung erfolgt über den AD_OUT-Plug)

Daten-Eingänge

- Keine (Daten werden ausschließlich über den AL_IN-Socket bereitgestellt)

Daten-Ausgänge

- Keine (Daten werden ausschließlich über den AD_OUT-Plug ausgegeben)

Adapter

Typ	Name	Richtung	Beschreibung
Socket	AL_IN	Eingang	Unidirektionaler LWORD-Adapter mit einem Ereignis-Eingang (`E1`) und einem Daten-Eingang (`D1` vom Typ `LWORD`)
Plug	AD_OUT	Ausgang	Unidirektionaler DWORD-Adapter mit einem Ereignis-Ausgang (`E1`) und einem Daten-Ausgang (`D1` vom Typ `DWORD`)

Funktionsweise

Der FB nimmt über den **AL_IN**-Socket ein LWORD-Datenwort (`D1`) zusammen mit einem Ereignis (`E1`) entgegen. Das Ereignis durchläuft den internen Funktionsblock **F_LWORD_TO_DWORD**, der die Konvertierung gemäß IEC 61131-Standard durchführt. Nach erfolgreicher Umwandlung wird das Ergebnis als DWORD (`AD_OUT.D1`) über den Plug ausgegeben, begleitet von einem Ereignis (`AD_OUT.E1`).

Die interne Netzwerkstruktur besteht aus:

- Einem **F_LWORD_TO_DWORD**-Baustein (aus der Bibliothek `iec61131::conversion`)
- Ereignisverbindung: `AL_IN.E1` ⟶ `Convert.REQ` / `Convert.CNF` ⟶ `AD_OUT.E1`
- Datenverbindung: `AL_IN.D1` ⟶ `Convert.IN` / `Convert.OUT` ⟶ `AD_OUT.D1`

Technische Besonderheiten

- **Bit-Reinterpretation** (strukturelle Kopie zwischen Bit-String-Typen; schneidet bei Verengung ab, erweitert mit Nullen bei Erweiterung).
- **Standardkonformität**: Die Konvertierung erfolgt über den IEC 61131-konformen FB `F_LWORD_TO_DWORD`, der eine verlustfreie Umwandlung von 64-Bit zu 32-Bit (niedrigwertige 32 Bit) vorsieht.
- **Unidirektionale Adapter**: Sowohl Eingangs- als auch Ausgangsadapter sind unidirektional ausgelegt, was die Komplexität reduziert und eine klar definierte Datenflussrichtung gewährleistet.
- **Ereignisgesteuerte Kommunikation**: Die Konvertierung wird durch ein Ereignis am Eingangsadapter getriggert; das Ergebnis wird ebenfalls ereignisgesteuert weitergereicht.
- **Einbettung in Composite-FB**: Durch die Kapselung als Composite-Baustein kann der FB einfach in größere Netze integriert werden.

Zustandsübersicht

Der FB **AL_TO_AD** besitzt keinen eigenen Zustandsautomaten, da er als Composite-FB realisiert ist. Die Zustandslogik wird vollständig durch den internen FB **F_LWORD_TO_DWORD** abgebildet:

1. **Warten** – Kein Ereignis am Eingang vorhanden.
2. **Konvertieren** – Nach Empfang eines Ereignisses wird die Umwandlung der LWORD-Daten durchgeführt (ein Taktzyklus).
3. **Ausgeben** – Das DWORD-Ergebnis und ein Bestätigungsereignis werden am Ausgangsadapter bereitgestellt.

Anwendungsszenarien

- **Systemintegration** – Wenn eine Komponente (z.B. eine SPS) LWORD-Daten liefert, aber ein nachgeschaltetes Modul DWORD-Daten erwartet.
- **Adapterbrücke** – In heterogenen Netzwerken, in denen Geräte unterschiedliche Datentypen für denselben physikalischen Wert verwenden.
- **Datenaufbereitung** – Vorverarbeitung von 64-Bit-Werten, die nur auf 32-Bit-Basis weiterverarbeitet werden sollen (z.B. bei Messwerten oder Zählern).

Vergleich mit ähnlichen Bausteinen

- **WORD_TO_DWORD** – Konvertiert 16-Bit-Worte in 32-Bit-Doppelworte, jedoch auf einfachen Datentypen, nicht auf Adaptern.

- **Direkte Adapterkopplung** – Ohne Konvertierung müssten beide Systeme denselben Adaptertyp (entweder AL oder AD) verwenden, was oft nicht gegeben ist.
- **F_LWORD_TO_DWORD (isoliert)** – Dieser Baustein arbeitet nur auf Datenebene; die Adapteranbindung muss separat erfolgen. **AL_TO_AD** vereinfacht die Handhabung durch Kapselung von Adapterlogik und Konvertierung.

Fazit

Der **AL_TO_AD**-Funktionsblock stellt eine schlanke und standardkonforme Lösung zur Umwandlung von LWORD- in DWORD-Adapterdaten dar. Durch die Kombination von Adapter- und Konvertierungslogik in einem Composite-FB wird die Wiederverwendbarkeit erhöht und die Netzwerkerstellung vereinfacht. Er eignet sich besonders für die Integration unterschiedlicher IEC 61499-Komponenten und trägt zur Modularität industrieller Automatisierungslösungen bei.

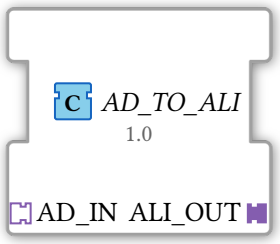
Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

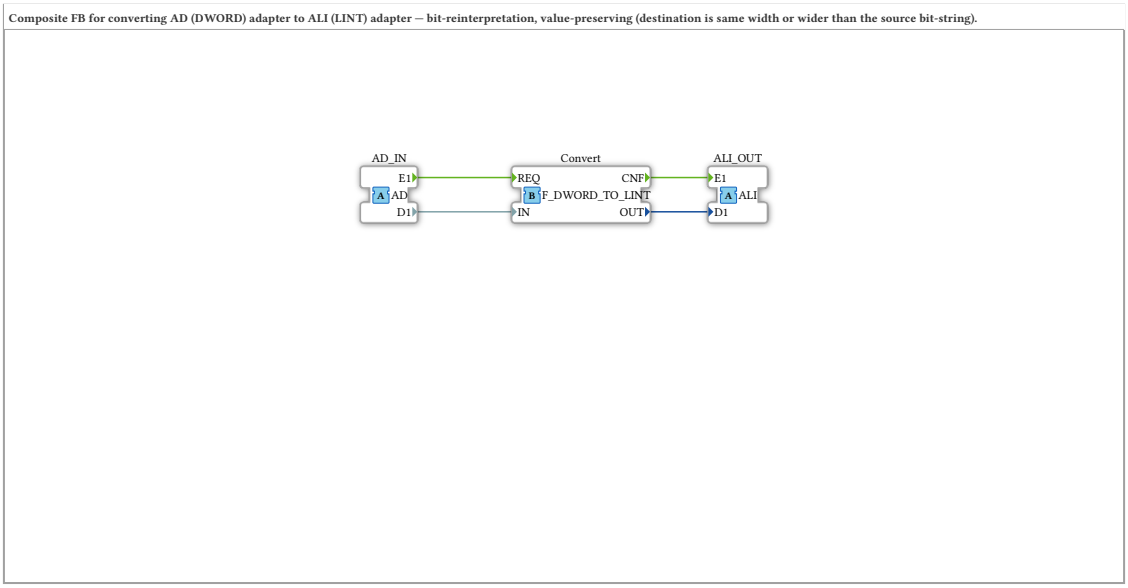
AD_ALI

AD_TO_ALI

DWORD Adapter Input – *AD*



ALI – LINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AD_TO_ALI** ist ein Composite-Funktionsbaustein zur unidirektionalen Umwandlung eines DWORD-Werts (über einen AD-Adapter) in einen LINT-Wert (über einen ALI-Adapter). Die Konvertierung erfolgt mittels des integrierten Standard-FBs `F_DWORD_TO_LINT`. Der Baustein ermöglicht eine einfache und standardisierte Anbindung von Komponenten, die unterschiedliche Datentypen über Adapterschnittstellen austauschen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine separaten Ereignis-Eingänge vorhanden. Die Ereignissteuerung erfolgt über den Adapter **AD_IN** (Ereignis *E1*).

Ereignis-Ausgänge

Keine separaten Ereignis-Ausgänge vorhanden. Ausgangsereignisse werden über den Adapter **ALI_OUT** (Ereignis *E1*) bereitgestellt.

Daten-Eingänge

Keine separaten Daten-Eingänge vorhanden. Die Eingangsdaten (DWORD) werden über den Adapter **AD_IN** (Data *D1*) bezogen.

Daten-Ausgänge

Keine separaten Daten-Ausgänge vorhanden. Die Ausgangsdaten (LINT) werden über den Adapter **ALI_OUT** (Data *D1*) ausgegeben.

Adapter

Adapter	Typ	Richtung	Beschreibung
AD_IN	`adapter::types::unidirectional::AD`	Socket	Empfängt einen DWORD-Wert (*D1*) sowie ein Eingangsereignis (*E1*).
ALI_OUT	`adapter::types::unidirectional::ALI`	Plug	Sendet den konvertierten LINT-Wert (*D1*) sowie ein Ausgangsereignis (*E1*).

Funktionsweise

Der Baustein wartet auf ein Ereignis am **AD_IN**-Adapter (Ereignis *E1*). Sobald dieses eintrifft, wird der aktuell anliegende DWORD-Wert (Data *D1*) aus dem Adapter gelesen und an den internen Baustein **Convert** (Typ `F_DWORD_TO_LINT`) übergeben. Dieser führt die Datentypkonvertierung durch. Nach erfolgreicher Konvertierung wird der resultierende LINT-Wert an den **ALI_OUT**-Adapter (Data *D1*) übergeben und ein Ereignis (*E1*) zur Bestätigung ausgelöst.

Technische Besonderheiten

- **Bit-Reinterpretation, werterhaltend** (Zielbreite ist gleich groß oder größer als der Quell-Bit-String).
- **Reine Composite-Logik**: Der FB enthält keine eigene Zustandsmaschine, sondern realisiert die Konvertierung ausschließlich über die Verschaltung des eingebetteten FBs `F_DWORD_TO_LINT`.
- **Standardkonformität**: Die verwendete Konvertierungsfunktion entspricht der IEC 61131-3 (Typkonvertierung `DWORD_TO_LINT`).
- **Synchronität**: Ereignis und Daten werden innerhalb eines einzigen Ausführungszyklus vom Eingang zum Ausgang weitergeleitet (E1 → REQ → CNF → E1).

Zustandsübersicht

Als Composite-Baustein besitzt **AD_TO_ALI** keine eigene Zustandsmaschine. Das Verhalten wird allein durch die innere Verschaltung und die Zustände des FBs `F_DWORD_TO_LINT` bestimmt. Eine grafische Zustandsübersicht entfällt.

Anwendungsszenarien

- **Industrielle Automatisierung**: Umwandlung von DWORD-basierten Sensordaten (z. B. Zählerstände) in LINT-Werte zur Weiterverarbeitung in Systemen mit 64-Bit-Integern.
- **Speicheradressierung**: Konvertierung von 32-Bit-Adressen (DWORD) in 64-Bit-Adressen (LINT) für große Speicherbereiche.
- **Datenintegration**: Anbindung von Komponenten, die unterschiedliche Adapterprotokolle nutzen, über eine standardisierte Konvertierungsschnittstelle.

Vergleich mit ähnlichen Bausteinen

Es existieren analoge Konvertierungsbausteine wie `AD_TO_ALI` für andere Datentypen (z. B. `AD_TO_ALI` für DWORD→LINT, `AI_TO_ALI` für INT→LINT). Der vorliegende FB deckt spezifisch die Umwandlung von **DWORD** nach **LINT** ab und ist auf diese Kombination optimiert. Er ersetzt nicht die allgemeine `F_DWORD_TO_LINT`-Funktion, sondern kapselt sie in einer adapterbasierten Schnittstelle.

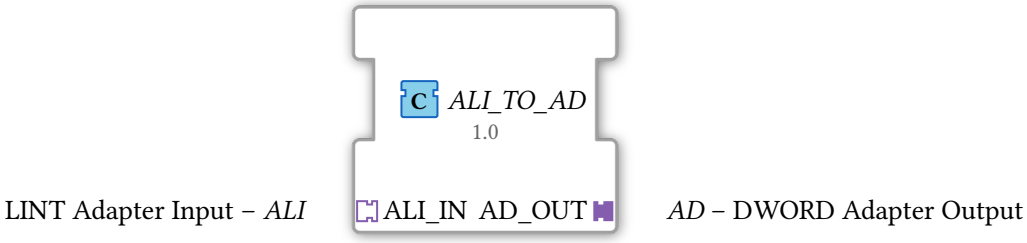
Fazit

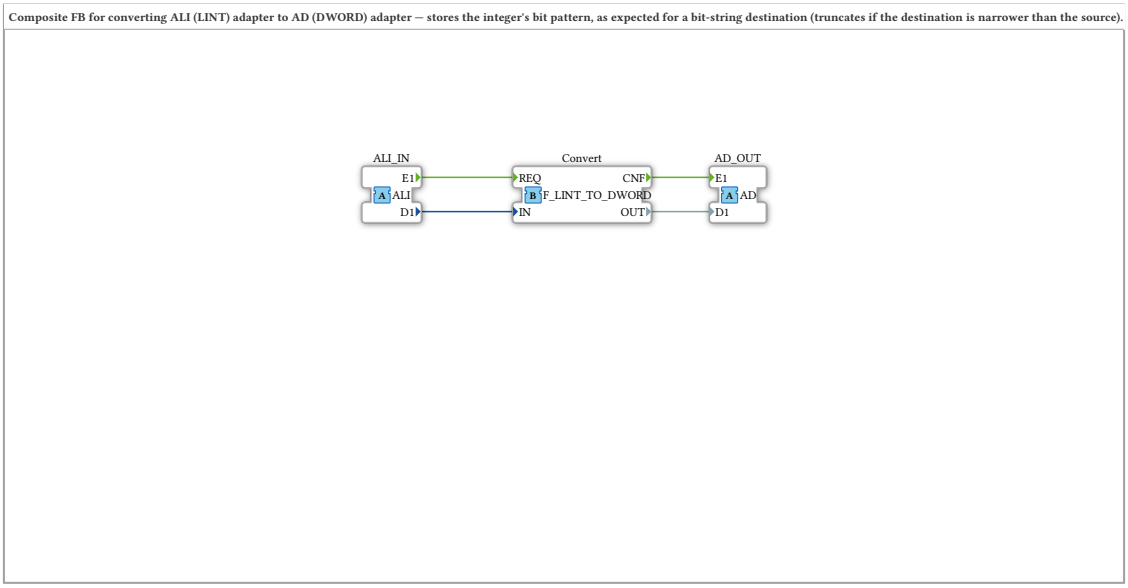
Der **AD_TO_ALI** bietet eine saubere, wiederverwendbare Lösung zur Konvertierung von DWORD-Adaptern auf LINT-Adapter. Durch die Composite-Struktur bleibt die Logik transparent und leicht in bestehende 4diac-Projekte integrierbar. Er eignet sich besonders für Szenarien, in denen eine einheitliche Adapterschnittstelle gefordert ist, aber unterschiedliche Datentypen zusammengeführt werden müssen.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ALI_TO_AD





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `ALI_TO_AD` ist ein Composite-Baustein zur Konvertierung einer LINT–Adapter-Schnittstelle (ALI) in eine DWORD–Adapter-Schnittstelle (AD). Er dient als unidirektionaler Übersetzer zwischen zwei unterschiedlichen Datentypen innerhalb einer adapterbasierten Kommunikationsstruktur.

Schnittstellenstruktur

Der Baustein besitzt keine eigenständigen Ereignis- oder Dateneingänge/-ausgänge. Die gesamte Kommunikation erfolgt ausschließlich über vordefinierte Adapter.

Ereignis-Eingänge

Keine. Die Ereignissteuerung erfolgt über den eingehenden Socket-Adapter `ALI_IN`.

Ereignis-Ausgänge

Keine. Die Ergebnisereignisse werden über den ausgehenden Plug-Adapter `AD_OUT` bereitgestellt.

Daten-Eingänge

Keine. Die Eingangsdaten werden über den Socket-Adapter `ALI_IN` übertragen.

Daten-Ausgänge

Keine. Die Ausgangsdaten werden über den Plug-Adapter `AD_OUT` übertragen.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	ALI_IN	`adapter::types::unidirectional::ALI`	Eingangsadapter vom Typ LINT
Plug	AD_OUT	`adapter::types::unidirectional::AD`	Ausgangsadapter vom Typ DWORD

Funktionsweise

Im Inneren des Bausteins wird eine einzelne Instanz des Funktionsblocks `iec61131::conversion::F_LINT_TO_DWORD` verwendet. Die Ereignis- und Datenverbindungen sind wie folgt verschaltet:

1. Ein Ereignis vom Socket `ALI_IN.E1` triggert den Eingang `REQ` des Konverters.
2. Der Konverter liest den LINT-Wert vom Eingang `ALI_IN.D1` ein und wandelt ihn in einen DWORD-Wert um.
3. Nach erfolgreicher Konvertierung erzeugt der Konverter das Bestätigungsereignis `CNF`, das auf den Plug `AD_OUT.E1` weitergeleitet wird.
4. Gleichzeitig wird der konvertierte DWORD-Wert vom Ausgang `Convert.OUT` auf den Datenausgang `AD_OUT.D1` übertragen.

Die gesamte Übersetzung erfolgt ereignisgesteuert: Ein eingehendes Ereignis löst die Konvertierung aus und das Ergebnis wird mit einem Ausgangsereignis quittiert.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- Der Baustein ist vollständig als Composite realisiert und benötigt keine eigene Laufzeitlogik.
- Der interne Konverter `F_LINT_TO_DWORD` ist ein IEC-61131-3-Baustein, der eine bitgenaue Konvertierung eines 64-Bit-Ganzzahlwerts (LINT) in einen 32-Bit-Doppelwortwert (DWORD) vornimmt.
- Die Adapter sind als unidirektional definiert, d. h. die Daten fließen ausschließlich vom Eingangs- zum Ausgangsadapter.
- Der Baustein ist für den Einsatz in Automatisierungssystemen konzipiert, die Adapter-Schnittstellen nach der Norm IEC 61499 nutzen.

Zustandsübersicht

Der Funktionsblock besitzt keinen eigenen expliziten Zustandsautomaten. Sein Verhalten wird vollständig durch die Ereignisweitergabe und die interne Logik des Konverterbausteins bestimmt. Aus Sicht des Anwenders verhält er sich wie eine transparente Datenwandlung, die bei jedem eingehenden Ereignis einmal ausgeführt wird.

Anwendungsszenarien

- **Integration heterogener Feldgeräte** – Wenn ein Gerät oder ein Teilnetzwerk Daten im LINT-Format (z. B. als Zählerwert oder Zeitstempel) bereitstellt, das Zielsystem aber DWORD-Werte erwartet.
- **Adapter-Brücke zwischen Subsystemen** – In einer verteilten Steuerung nach IEC 61499 kann `ALI_TO_AD` eingesetzt werden, um zwei Subsysteme mit unterschiedlichen Adapterdefinitionen zu verbinden.
- **Datenformat-Anpassung in Migrationsprojekten** – Bei der Modernisierung von Anlagen, bei denen alte Komponenten noch LINT-Adapter liefern, neue Komponenten jedoch DWORD-Adapter benötigen.

Vergleich mit ähnlichen Bausteinen

- **`INT_TO_DWORD`** – Wandelt einen Integer-Wert (16 Bit) in DWORD um, wird aber nicht auf Adapter-Ebene angeboten.
- **`DWORD_TO_LINT`** – Kehrt die hier beschriebene Konvertierung um und wandelt DWORD zurück nach LINT.
- **Eigene Kombinationsbausteine** – Statt eines fest verdrahteten Adapter-Converters könnte die Konvertierung auch manuell mit einem `F_LINT_TO_DWORD` und separaten Adaptern realisiert werden. Der Composite-Baustein bietet hier eine kompakte und wiederverwendbare Kapselung.

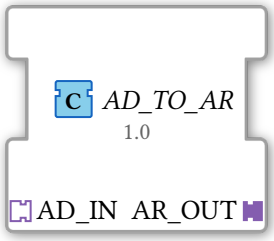
Fazit

`ALI_TO_AD` ist ein einfacher, aber nützlicher Composite-Baustein, der die unidirektionale Konvertierung eines LINT-Adapters in einen DWORD-Adapter ermöglicht. Er erlaubt eine saubere Trennung der Datentypen auf der Adapterebene und vereinfacht die Integration unterschiedlicher Systeme in einer IEC-61499-Umgebung. Die Verwendung eines standardisierten IEC-61131-Konverters gewährleistet eine definierte und reproduzierbare Datenwandlung.

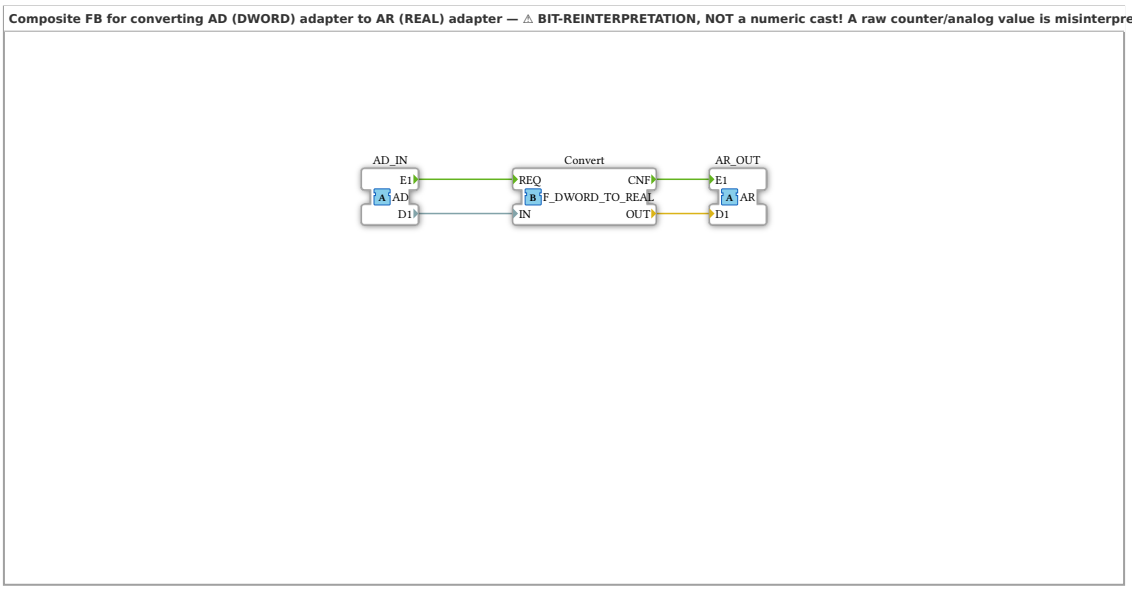
AD_AR

AD_TO_AR

DWORD Adapter Input – *AD*



AR – REAL Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AD_TO_AR** ist ein Composite-Baustein, der einen DWORD-Wert aus einem unidirektionalen AD-Adapter (Socket) auf einen REAL-Wert an einem AR-Adapter (Plug) abbildet.

⚠ **Achtung — keine numerische Wertumwandlung:** `AD_TO_AR` nutzt intern `F_DWORD_TO_REAL`, was im FORTE-Kern für Bit-String-Quelltypen (BYTE/WORD/DWORD/LWORD) eine reine IEEE754-**Bit-Reinterpretation** ist, keine Zahlenwert-Umwandlung. Ein roher Zähler- oder Analogwert wie `DWORD#2048` wird dadurch **nicht** zu `REAL#2048.0`, sondern zu einer bedeutungslosen

Zahl nahe Null. Für eine echte numerische DWORD→REAL-Umwandlung siehe AD_TO_AR_NUM (zweistufig über UDINT). AD_TO_AR ist nur dann korrekt, wenn AD_IN bereits ein Bitmuster ist, das als REAL interpretiert werden soll (z. B. das Ergebnis von F_REAL_TO_DWORD).

Schnittstellenstruktur

Der Baustein besitzt keine direkten (einzelnen) Ereignis- oder Daten-Ein-/Ausgänge. Die gesamte Kommunikation erfolgt über zwei Adapter-Schnittstellen:

Ereignis-Eingänge

- *Keine direkten Ereignis-Eingänge* – die Ereignissteuerung erfolgt über den eingehenden Adapter.

Ereignis-Ausgänge

- *Keine direkten Ereignis-Ausgänge* – die Ereignisausgabe erfolgt über den ausgehenden Adapter.

Daten-Eingänge

- *Keine direkten Daten-Eingänge* – die Datenaufnahme erfolgt über den eingehenden Adapter.

Daten-Ausgänge

- *Keine direkten Daten-Ausgänge* – die Datenausgabe erfolgt über den ausgehenden Adapter.

Adapter

Name	Typ	Richtung	Beschreibung
AD_IN	adapter::types::unidirectional::AD	Socket	Empfängt ein Ereignis (E1) und einen DWORD-Wert (D1) zur Konvertierung.
AR_OUT	adapter::types::unidirectional::AR	Plug	Sendet ein Ereignis (E1) mit dem konvertierten REAL-Wert (D1) aus.

Funktionsweise

Der Baustein arbeitet als ereignisgesteuerte Pipeline:

1. Ein eingehendes Ereignis am Socket **AD_IN.E1** triggert die Konvertierung.
2. Der Datenwert **AD_IN.D1** (DWORD) wird an den internen Funktionsblock F_DWORD_TO_REAL übergeben.
3. Nach erfolgreicher Konvertierung signalisiert der interne Baustein ein Ausgangsereignis (**CNF**).
4. Dieses Ereignis wird an den Plug **AR_OUT.E1** weitergeleitet und zeitgleich der konvertierte REAL-Wert über **AR_OUT.D1** ausgegeben.

F_DWORD_TO_REAL reinterpretiert dabei lediglich das 32-Bit-Muster des DWORD als IEEE754-REAL (Bit-Kopie), es findet **keine** Zahlenwert-Konvertierung statt.

Technische Besonderheiten

- ⚠️ **Bit-Reinterpretation statt Zahlenwert-Umwandlung**: Siehe Warnhinweis in der Einleitung. Betroffen sind alle Bit-String-Quelltypen (AB/BYTE, AW/WORD, AD/DWORD, AL/LWORD) beim Umwandeln in AR/REAL oder ALR/LREAL — der einzige weitere Fall in dieser Bibliothek ist AL_TO_ALR (LWORD→LREAL).
- **Composite-Baustein**: Die Konvertierungslogik ist vollständig in einem internen Netzwerk aus einem einzigen Funktionsblock realisiert.
- **Typkonvertierung über Adapter**: Der Baustein ermöglicht die Anbindung von Komponenten, die ausschließlich über Adapter-Schnittstellen kommunizieren, ohne dass zusätzliche manuelle Konvertierungen notwendig sind.
- **Standardkonformität**: Verwendet die IEC-61131-Bibliotheksfunktion F_DWORD_TO_REAL, dadurch portabel und gut getestet.
- **Unidirektionale Adapter**: Sowohl Eingangs- als auch Ausgangsadapter sind als unidirektional ausgeführt, d.h. es findet nur eine Datenflussrichtung statt.

Zustandsübersicht

Der Baustein besitzt keinen eigenen Zustandsautomaten. Die Verarbeitung erfolgt streng kausal: Ein Ereignis am Eingang löst sofort die Konvertierung und Ausgabe aus. Es gibt keine Verzögerungen oder Zwischenzustände.

Anwendungsszenarien

- **Bitmuster-Weitergabe**: `AD_IN` liefert bereits ein Bitmuster, das als REAL interpretiert werden soll (z. B. das Ergebnis eines `F_REAL_TO_DWORD` an anderer Stelle im Netzwerk, oder deserialisierte Float-Rohdaten aus einem Feldbus/Protokoll).
- **NICHT geeignet** für rohe Zähler-, Analog- oder sonstige Ganzzahlwerte, die als derselbe Zahlenwert in REAL vorliegen sollen — dafür `AD_TO_AR_NUM` verwenden.

Vergleich mit ähnlichen Bausteinen

- **`AD_TO_AR_NUM`** (numerisch, DWORD→UDINT→REAL): der sichere Ersatz, wenn tatsächlich ein Zahlenwert gemeint ist.
- **`AI_TO_AR`/`ADI_TO_AR`/`AUDI_TO_AR`** (INT/DINT/UDINT→REAL): diese sind bereits numerisch korrekt, da ihre Quelltypen `ANY_INT` (nicht `ANY_BIT`) sind — kein Trap.
- **`AL_TO_ALR`** (LWORD→LREAL): dieselbe Bit-Reinterpretations-Falle, nur mit den 64-Bit-Pendants.

Fazit

Der **AD_TO_AR**-Baustein bietet eine kompakte Lösung, um ein DWORD-Bitmuster über Adapter-Schnittstellen als REAL bereitzustellen — **aber keine numerische Wertumwandlung**. Für rohe Zähler- oder Analogwerte, die als derselbe Zahlenwert in REAL vorliegen sollen, `AD_TO_AR_NUM` verwenden.

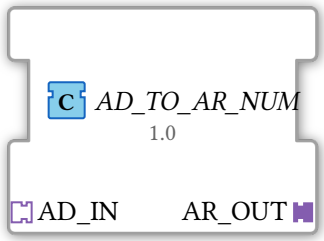
📖 Hintergrund

- Numerisch vs. bitweise: Die Konvertierungs-Falle in FORTE

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

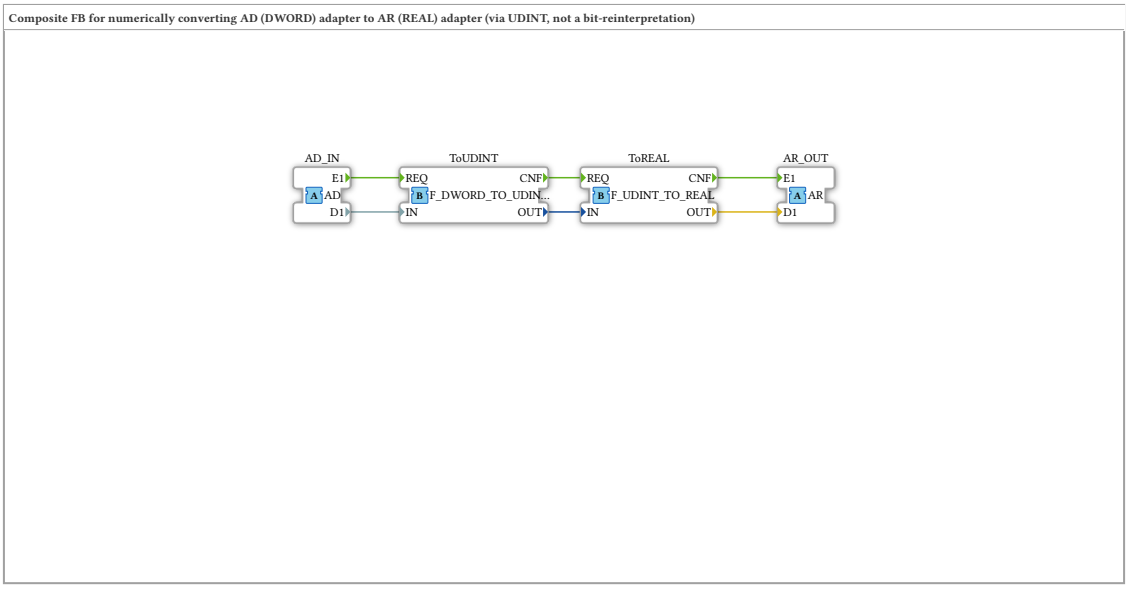
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AD_TO_AR_NUM



DWORD Adapter Input – *AD*

AR – REAL Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AD_TO_AR_NUM** ist die numerisch korrekte Alternative zu `AD_TO_AR`: Er wandelt einen DWORD-Wert aus einem unidirektionalen AD-Adapter (Socket) über einen internen UDINT-Zwischenschritt in einen REAL-Wert um, der über einen AR-Adapter (Plug) ausgegeben wird — eine **echte Wertumwandlung**, keine Bit-Reinterpretation.

Warum nicht einfach AD_TO_AR?

`AD_TO_AR` nutzt intern `F_DWORD_TO_REAL`, was im FORTE-Kern für Bit-String-Quelltypen (BYTE/WORD/DWORD/LWORD) eine reine IEEE754-Bit-Reinterpretation ist, keinen numerischen Cast. Ein roher Zähler- oder Analogwert wie `DWORD#2048` würde dadurch **nicht** zu `REAL#2048.0`, sondern zu einer bedeutungslosen Zahl nahe Null (siehe `AD_TO_AR` und `AD_TO_AR_TODO.md` für die Herleitung).

Schnittstellenstruktur

Der Baustein besitzt keine direkten (einzelnen) Ereignis- oder Daten-Ein-/Ausgänge. Die gesamte Kommunikation erfolgt über zwei Adapter-Schnittstellen:

Adapter

Name	Typ	Richtung	Beschreibung
AD_IN	`adapter::types::unidirectional::AD`	Socket	Empfängt ein Ereignis (E1) und einen DWORD-Rohwert (D1) zur Konvertierung.
AR_OUT	`adapter::types::unidirectional::AR`	Plug	Sendet ein Ereignis (E1) mit dem numerisch umgewandelten REAL-Wert (D1) aus.

Funktionsweise

Der Baustein arbeitet zweistufig statt in einem einzigen (gefährlichen) Schritt:

1. Ein eingehendes Ereignis am Socket **AD_IN.E1** triggert die Konvertierung.
2. Der Datenwert **AD_IN.D1** (DWORD) wird an `ToUDINT` (`F_DWORD_TO_UDINT`) übergeben — eine Bit-Reinterpretation DWORD⟶UDINT, die hier **gültig** ist, da beide Typen dieselbe 32-Bit-Darstellung eines vorzeichenlosen Integers teilen.
3. `ToUDINT.CNF` löst `ToREAL` (`F_UDINT_TO_REAL`) aus — ein **echter numerischer Cast**, der den UDINT-Zahlenwert korrekt in REAL überführt.
4. Das Ergebnis wird über den Plug **AR_OUT.E1**/**AR_OUT.D1** ausgegeben.

Technische Besonderheiten

- **Identische Schnittstelle wie `AD_TO_AR`**: `AD_TO_AR_NUM` ist ein Drop-in-Ersatz — gleiche Socket-/Plug-Typen (`AD`/`AR`), einfach austauschbar, sobald klar ist, dass ein Zahlenwert (nicht ein Bitmuster) gemeint ist.
- **Zweistufige Kette DWORD→UDINT→REAL**: Dieselbe Kette, die bereits in mehreren Übungen (`Uebung_028a_AR` u. a.) von Hand verdrahtet wird, hier als ein einziger Baustein.
- **Kein Genauigkeitsverlust im relevanten Wertebereich**: `UDINT` (32 bit, max. 4.294.967.295) wird verlustfrei in `REAL` (IEEE754 einfache Genauigkeit, 24-Bit-Mantisse) übertragen, solange der Wert innerhalb der exakt darstellbaren Ganzzahlen liegt (bis 2^{24} = 16.777.216) — für typische Analog-/Zählerrohwerte (z. B. 0–64255 oder 0–4095) unproblematisch.

Anwendungsszenarien

- **Analoge Rohwerte**: Ein Analogeingang oder Zähler liefert einen DWORD-Rohwert (z. B. 0–4095, 0–64255), der als tatsächlicher Zahlenwert in REAL weiterverarbeitet werden soll (Skalierung, Anzeige, OPC-UA-Publish).
- **Ersatz für die manuelle `AD_TO_AUDI` + `AUDI_TO_AR`-Kette**: Wo bisher zwei separate Adapter-Instanzen im SubApp-Netzwerk nötig waren, genügt jetzt eine.

Vergleich mit ähnlichen Bausteinen

- `AD_TO_AR`: die Bit-Reinterpretations-Variante — nur verwenden, wenn `AD_IN` bereits ein Bitmuster ist, das als REAL interpretiert werden soll.
- **`AI_TO_AR`/`ADI_TO_AR`/`AUDI_TO_AR`**: bereits numerisch korrekt, da ihre Quelltypen `ANY_INT` sind — falls der Rohwert bereits als INT/DINT/UDINT-Adapter vorliegt, ist keine `_NUM`-Variante nötig.

Fazit

AD_TO_AR_NUM schließt die Lücke zwischen dem naheliegenden, aber gefährlichen `AD_TO_AR` und der tatsächlich benötigten numerischen DWORD→REAL-Umwandlung — als einzelner, drop-in-kompatibler Baustein statt einer manuell verdrahteten Zwei-Baustein-Kette.

📖 **Hintergrund**

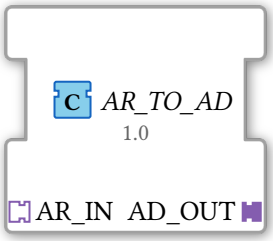
- Numerisch vs. bitweise: Die Konvertierungs-Falle in FORTE

🌐 **Passende Themen-Unterseiten auf ms-muc-docs.de**

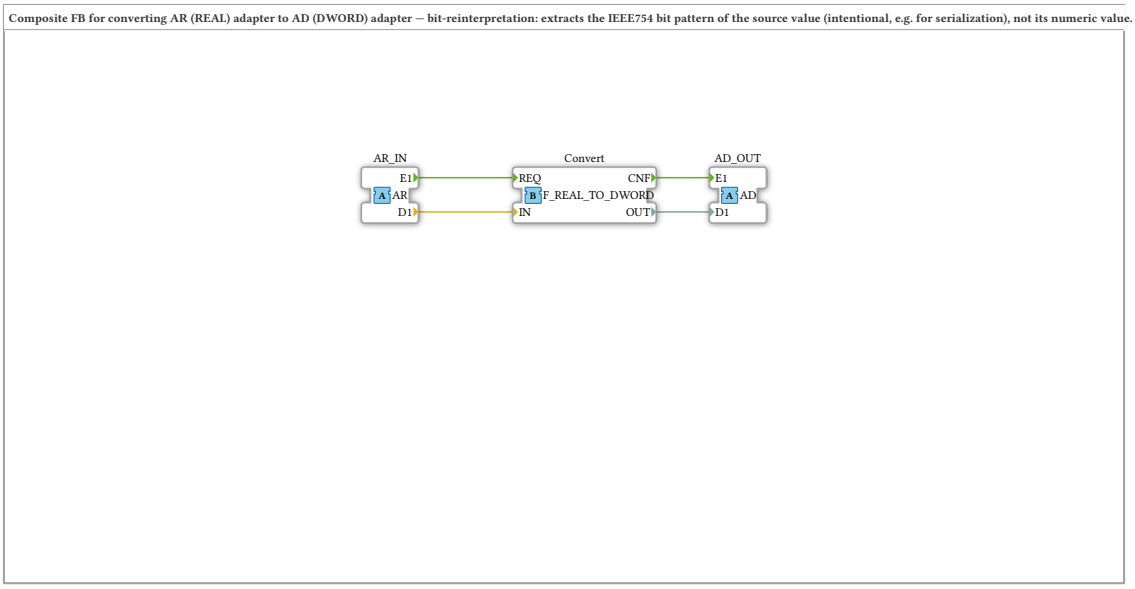
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AR_TO_AD

REAL Adapter Input – *AR*



AD – DWORD Adapter Output



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock `AR_TO_AD` dient als zusammengesetzter Baustein (Composite FB) zur Umwandlung eines REAL-Wertes, der über einen unidirektionalen AR-Adapter hereinkommt, in einen DWORD-Wert, der über einen unidirektionalen AD-Adapter ausgegeben wird. Die Konvertierung erfolgt nach IEC 61131-3 mittels des Bausteins `F_REAL_TO_DWORD`. Der FB kapselt die notwendigen Ereignis- und Datenverbindungen und erlaubt eine einfache Integration in bestehende Adapter-basierte Kommunikationsstrukturen.

Schnittstellenstruktur

Ereignis-Eingänge

Eingang	Typ	Beschreibung
`AR_IN.E1`	Adapter-Ereignis	Empfängt das Auslöseereignis vom AR-Adapter, welches die Konvertierung startet.

Ereignis-Ausgänge

Ausgang	Typ	Beschreibung
`AD_OUT.E1`	Adapter-Ereignis	Sendet ein Ereignis an den AD-Adapter, sobald der konvertierte DWORD-Wert anliegt.

Daten-Eingänge

Eingang	Typ	Beschreibung
`AR_IN.D1`	REAL	Der zu konvertierende REAL-Wert, der vom AR-Adapter bereitgestellt wird.

Daten-Ausgänge

Ausgang	Typ	Beschreibung
`AD_OUT.D1`	DWORD	Der konvertierte DWORD-Wert, der an den AD-Adapter übergeben wird.

Adapter

Der FB besitzt einen **Socket** vom Typ `adapter::types::unidirectional::AR` (mit der Bezeichnung `AR_IN`) und einen **Plug** vom Typ `adapter::types::unidirectional::AD` (mit der Bezeichnung `AD_OUT`).

Die tatsächlichen Ein- und Ausgangssignale sind über die Ereignis- und Daten-Pins dieser Adapter zugänglich.

Funktionsweise

1. Der FB wartet auf ein Ereignis am Ereignis-Eingang `AR_IN.E1`.
2. Gleichzeitig wird der aktuelle Wert am Daten-Eingang `AR_IN.D1` (REAL) übernommen.
3. Intern wird der Baustein `F_REAL_TO_DWORD` (aus der IEC 61131-3-Bibliothek) aufgerufen, der die Umwandlung vornimmt.
4. Nach erfolgreicher Konvertierung wird der resultierende DWORD-Wert an den Daten-Ausgang `AD_OUT.D1` gelegt.
5. Anschließend wird ein Ereignis am Ausgang `AD_OUT.E1` ausgegeben, um den angeschlossenen AD-Adapter zu benachrichtigen.

Der gesamte Vorgang erfolgt synchron in einem Schritt, d.h. der FB ist rein ereignisgesteuert und führt keine internen Zustände.

Technische Besonderheiten

- **Bit-Reinterpretation**: extrahiert das IEEE754-Bitmuster des Quellwerts (beabsichtigt, z. B. zur Serialisierung), nicht dessen Zahlenwert.
- **Composite FB:** Die Logik wird ausschließlich durch die Verknüpfung des internen `F_REAL_TO_DWORD`-Bausteins realisiert. Es gibt keine eigene Zustandsmaschine oder komplexe Algorithmen.
- **Adapter-Kopplung:** Der FB kapselt die Konvertierung auf Adapterebene, sodass er ohne zusätzliche Anpassungen in Adapter-basierte Architekturen (z.B. verteilte Systeme mit 4diac) eingefügt werden kann.
- **IEC 61131-3 Konformität:** Die eigentliche Umwandlung erfolgt durch einen genormten Baustein, was die Portabilität und Standardkonformität erhöht.
- **Paketname:** Der Baustein wird unter dem Paket `adapter::conversion::unidirectional` bereitgestellt und importiert `iec61131::conversion::F_REAL_TO_DWORD`.

Zustandsübersicht

Da es sich um einen kombinatorischen Composite-FB handelt, besitzt der Baustein keinen internen Zustandsautomaten. Die Funktionalität ist rein ereignisgesteuert und wird bei jedem Ereignis am Eingang einmalig durchlaufen.

Anwendungsszenarien

- **Datenprotokoll-Adaption:** Wenn ein REAL-Wert (z.B. ein Messwert) über ein Bussystem nur als DWORD (z.B. bei vielen Feldbusprotokollen) übertragen werden kann.
- **Adapterbrücke:** In Systemen, die sowohl REAL- als auch DWORD-Adapter verwenden, kann dieser FB als einfache Umwandlungseinheit zwischen zwei Adaptertypen dienen.
- **Wiederverwendbare Konvertierung:** Durch die Kapselung kann die Konvertierung mehrfach in verschiedenen Netzwerken eingesetzt werden, ohne dass jedes Mal die Verbindungen manuell hergestellt werden müssen.

Vergleich mit ähnlichen Bausteinen

- **F_REAL_TO_DWORD:** Der direkte Baustein aus IEC 61131-3 bietet dieselbe Konvertierung, erfordert jedoch manuelle Verdrahtung von Ereignis- und Datenleitungen. `AR_TO_AD` kapselt dies in einer Adapter-Schnittstelle, was die Integration in Adapter-Netzwerke vereinfacht.
- **AR_TO_SINT / AR_TO_INT:** Vergleichbare Konvertierungen existieren für andere ganzzahlige Datentypen, jedoch nicht als Adapter-Version. `AR_TO_AD` ist speziell für DWORD und Adapter konzipiert.
- **Selbstdefinierte Composite-FBs:** Prinzipiell könnte jeder Anwender eine ähnliche Kapselung erstellen. Der Baustein bietet jedoch eine standardisierte Lösung mit klarer Paketstruktur.

Fazit

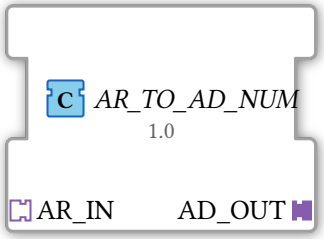
Der `AR_TO_AD`-Funktionsblock ist ein spezialisierter Composite-Baustein, der die Umwandlung eines REAL- auf einen DWORD-Wert auf Adapterebene ermöglicht. Er zeichnet sich durch seine einfache Handhabung, Standardkonformität und die nahtlose Integration in unidirektionale Adapter-Kommunikationen aus. Durch die Verwendung des bewährten IEC 61131-3-Bausteins `F_REAL_TO_DWORD` wird eine zuverlässige und portable Konvertierung gewährleistet.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

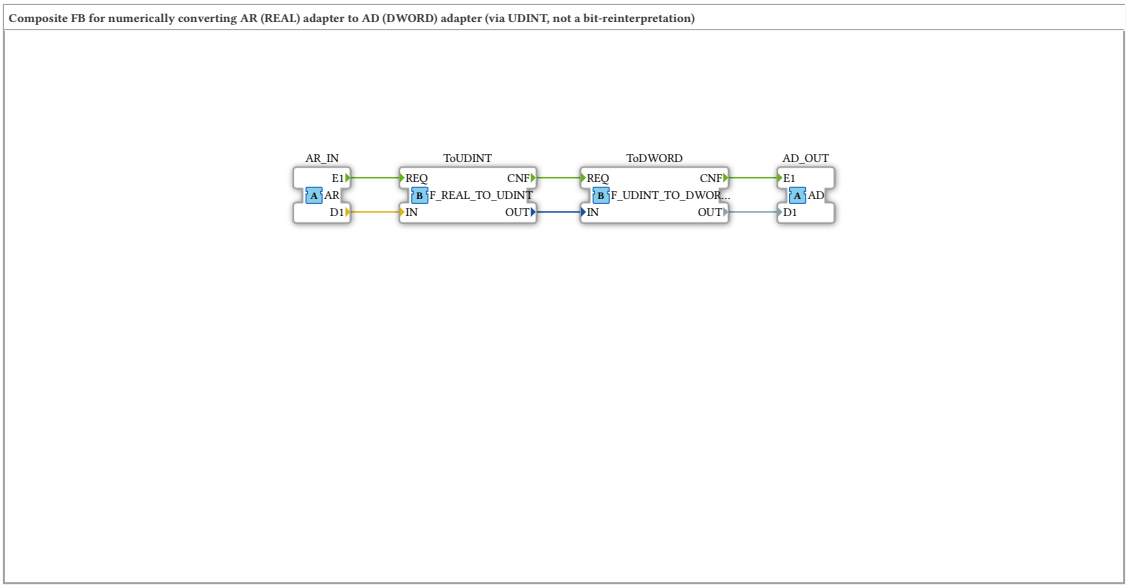
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AR_TO_AD_NUM

REAL Adapter Input – *AR*



AD – DWORD Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein `AR_TO_AD_NUM` ist ein Composite-FB, der eine numerisch korrekte Umwandlung eines REAL-Adapter-Signals (AR) in ein DWORD-Adapter-Signal (AD) durchführt. Im Gegensatz zu einer einfachen Bit-Reinterpretation wird der REAL-Wert über einen internen UDINT-Zwischenschritt in die entsprechende DWORD-Zahl konvertiert. Dadurch bleibt der numerische Wert erhalten – ein REAL-Wert von `50.0` wird zu `DWORD#50`, nicht zum rohen IEEE-754-Bitmuster.

Dieser Baustein ist als drop-in-kompatible Alternative zu `AR_TO_AD` gedacht und vermeidet dessen typische Fehlerquelle bei der Übergabe von Sollwerten (z. B. Prozentwerten oder PWM-Tastgraden) an Ausgänge, die einen direkten Zahlenwert erwarten.

Schnittstellenstruktur

Ereignis-Eingänge

- **E1** (vom Socket `AR_IN`): Wird ausgelöst, wenn der angeschlossene REAL-Adapter eine gültige Datenübertragung signalisiert. Dieses Ereignis startet die Konvertierungskette.

Ereignis-Ausgänge

- **E1** (vom Plug `AD_OUT`): Wird nach Abschluss der Konvertierung ausgegeben und signalisiert, dass der DWORD-Wert am Daten-Ausgang gültig ist.

Daten-Eingänge

- **D1** (vom Socket `AR_IN`): REAL-Eingangswert, der in einen DWORD-Wert umgewandelt werden soll.

Daten-Ausgänge

- **D1** (vom Plug `AD_OUT`): Konvertierter DWORD-Wert, der den numerischen Wert des Eingangs als vorzeichenlose 32-Bit-Ganzzahl enthält.

Adapter

- **Socket `AR_IN`**: Typ `adapter::types::unidirectional::AR` (REAL-Adapter) – stellt den Eingang für den zu konvertierenden Wert dar.
- **Plug `AD_OUT`**: Typ `adapter::types::unidirectional::AD` (DWORD-Adapter) – stellt den Ausgang für den konvertierten Wert dar.

Funktionsweise

Der Baustein arbeitet in zwei Schritten:

1. **Reale numerische Konvertierung REAL → UDINT**
   Der REAL-Eingangswert wird über den internen Funktionsbaustein `F_REAL_TO_UDINT` in eine vorzeichenlose 32-Bit-Ganzzahl (UDINT) umgewandelt. Dies ist ein echter numerischer Cast, kein Bit-Umschreiben.
2. **Bit-Identische Umwandlung UDINT → DWORD**
   Der UDINT-Wert wird über `F_UDINT_TO_DWORD` in einen DWORD-Wert interpretiert. Da UDINT und DWORD beide exakt 32-Bit breit sind und dieselbe Bit-Darstellung für vorzeichenlose Integer verwenden, ist dieser Schritt verlustfrei.

Die Ereigniskette ist: `AR_IN.E1` → `ToUDINT.REQ` → (nach Konvertierung) `ToUDINT.CNF` → `ToDWORD.REQ` → (nach Konvertierung) `ToDWORD.CNF` → `AD_OUT.E1`.

Die Datenkette: `AR_IN.D1` → `ToUDINT.IN` → `ToUDINT.OUT` → `ToDWORD.IN` → `ToDWORD.OUT` → `AD_OUT.D1`.

Damit wird der numerische Wert des REAL-Eingangs exakt als DWORD-Ausgang bereitgestellt.

Technische Besonderheiten

- **Vermeidung der Bit-Reinterpretationsfalle**: Der direkte Weg über `F_REAL_TO_DWORD` (wie in `AR_TO_AD`) würde das IEEE-754-Bitmuster des REAL-Werts erzeugen. `AR_TO_AD_NUM` umgeht dies durch den Zwischenschritt über UDINT.
- **Drop-in-Kompatibilität**: Der Baustein hat exakt dieselbe Signatur wie `AR_TO_AD`, sodass er einfach ausgetauscht werden kann.
- **Unidirektionale Adapter**: Sowohl Ein- als auch Ausgang sind als unidirektionale Adapter (nur Senden bzw. Empfangen) ausgelegt.
- **Kein interner Zustand**: Es findet keine Speicherung statt; die Konvertierung erfolgt rein ereignisgesteuert.

Zustandsübersicht

Der Baustein besitzt keinen expliziten internen Zustand. Er verhält sich wie eine reine Transformationskette: Nach einer Anforderung über `E1` am Eingang wird nach Durchlauf der beiden Konvertierungsstufen der Ergebniswert am Ausgang bereitgestellt. Es gibt keine Verzögerungen außer der reinen Verarbeitungszeit der enthaltenen Funktionsbausteine.

Anwendungsszenarien

- **PWM-Tastgrad-Ausgabe**: Wenn ein REAL-Sollwert (z. B. `50.0` für 50 %) direkt als PWM-Tastgrad an einen Ausgang wie `logiBUS_QDA_PWM` übergeben werden soll, erwartet dieser Ausgang einen DWORD-Zahlenwert. `AR_TO_AD_NUM` liefert genau diesen Wert.
- **Prozentwert-Weitergabe**: Ein prozentualer Sollwert (0…100) kann als DWORD interpretiert und an nachgelagerte Bausteine übergeben werden, ohne dass eine Bit-Konvertierung die Bedeutung verfälscht.
- **Ersatz für manuelle Verkettung**: Die früher übliche manuelle Verdrahtung über `AR_TO_AUDI` + `AUDI_TO_AD` wird durch diesen Baustein in einem einzigen Baustein gebündelt.

Vergleich mit ähnlichen Bausteinen

Baustein	Konvertierung	Ergebnis

AR_TO_AD	REAL ⟶ DWORD (direkt, bitweise)	IEEE-754-Bitmuster
AR_TO_AD_NUM	REAL ⟶ UDINT ⟶ DWORD (numerisch)	Zahlenwert als DWORD
AR_TO_AUDI + AUDI_TO_AD	REAL ⟶ UDINT (numerisch) + UDINT ⟶ DWORD (bitidentisch)	Zahlenwert als DWORD (manuell)

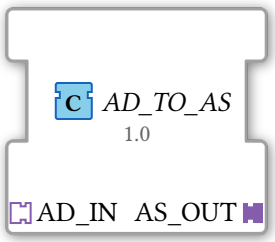
Während AR_TO_AD für die reine Bit-Serialisierung von REAL-Werten geeignet ist (z. B. zur Übertragung über einen Kommunikationskanal), ist AR_TO_AD_NUM für numerische Anwendungen konzipiert, bei denen der DWORD-Wert denselben Zahlenwert wie der REAL-Eingang repräsentieren soll.

Fazit

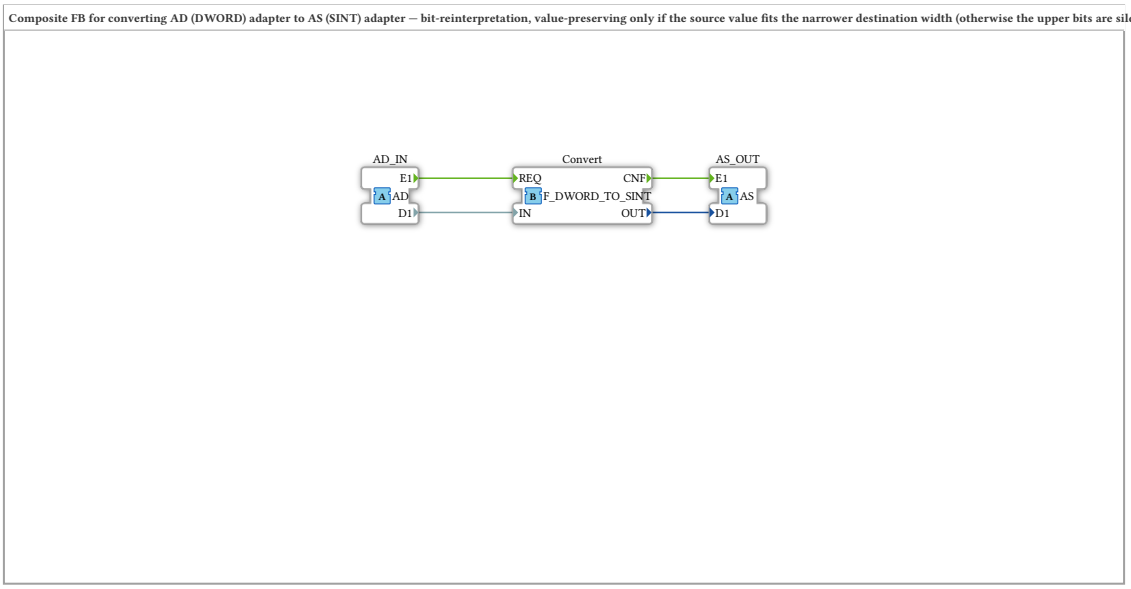
AR_TO_AD_NUM ist eine wichtige Ergänzung für Steuerungsanwendungen, die einen REAL-Sollwert als numerischen DWORD-Wert an Ausgabebausteine übergeben müssen. Durch die bewusste Umgehung der bitweisen Konvertierung vermeidet er häufig auftretende, schwer zu diagnostizierende Fehler und vereinfacht die Projektierung erheblich. Er ist die empfohlene Wahl, wenn der REAL-Wert als Zahl (z. B. Prozent- oder Tastgrad) interpretiert werden soll.

AD_AS

AD_TO_AS



DWORD Adapter Input – *AD* — *AS* – SINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AD_TO_AS** ist ein Composite-Baustein zur Umwandlung eines DWORD-Adapters (Typ AD) in einen SINT-Adapter (Typ AS). Er ermöglicht die Datenkonvertierung zwischen verschiedenen Adapter-Standards innerhalb der 4diac-IDE und wird in Version 1.0 bereitgestellt.

Die interne Konvertierung erfolgt über den IEC-61131-Baustein F_DWORD_TO_SINT.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereigniseingänge. Der Baustein empfängt Ereignisse ausschließlich über den Adapter-Socket **AD_IN** (Ereignisport E1 des AD-Adapters).

Ereignis-Ausgänge

Keine direkten Ereignisausgänge. Der Baustein sendet Ereignisse ausschließlich über den Adapter-Plug **AS_OUT** (Ereignisport E1 des AS-Adapters).

Daten-Eingänge

Keine direkten Dateneingänge. Der Baustein empfängt Daten ausschließlich über den Adapter-Socket **AD_IN** (Datenport D1 des AD-Adapters, Datentyp **DWORD**).

Daten-Ausgänge

Keine direkten Datenausgänge. Der Baustein sendet Daten ausschließlich über den Adapter-Plug **AS_OUT** (Datenport D1 des AS-Adapters, Datentyp **SINT**).

Adapter

- **Socket AD_IN**: Typ: `adapter::types::unidirectional::AD` Beschreibung: Empfängt einen DWORD-Wert sowie ein zugehöriges Ereignis.
- **Plug AS_OUT**: Typ: `adapter::types::unidirectional::AS` Beschreibung: Sendet einen SINT-Wert sowie ein zugehöriges Ereignis.

Funktionsweise

Der Funktionsblock arbeitet ereignisgesteuert:

1. Ein Ereignis am Socket **AD_IN.E1** löst die Verarbeitung aus.
2. Der interne Baustein `Convert` (Typ `F_DWORD_TO_SINT`) wird gestartet. Er erhält den DWORD-Wert von **AD_IN.D1** an seinem Eingang `IN` und führt die Umwandlung in einen SINT-Wert durch.
3. Nach Abschluss der Konvertierung signalisiert `Convert` über seinen Ausgang `CNF` ein Ereignis.
4. Dieses Ereignis wird an den Plug **AS_OUT.E1** weitergeleitet und gleichzeitig der konvertierte SINT-Wert von `Convert.OUT` an **AS_OUT.D1** übergeben.

Damit wird eine vollständige, unidirektionale Kopplung zwischen dem AD- und dem AS-Adapter realisiert.

Technische Besonderheiten

- **Bit-Reinterpretation**, werterhaltend nur, wenn der Quellwert in die schmalere Zielbreite passt (sonst werden die oberen Bits stillschweigend verworfen).
- **Wertebereichsverlust**: Die Konvertierung von DWORD (32 Bit, vorzeichenlos) zu SINT (8 Bit, vorzeichenbehaftet) kann zu Datenverlust führen. Es werden nur die unteren 8 Bit des DWORD verwendet; der Wertebereich von SINT liegt zwischen −128 und +127.
- **Verwendete Bibliothek**: Der Baustein greift auf die IEC-61131-Konvertierungsfunktion `F_DWORD_TO_SINT` zurück, die in der Paketgruppe `iec61131::conversion` definiert ist.
- **Unidirektionaler Datenfluss**: Der Funktionsblock ermöglicht nur eine Datenrichtung (von AD zu AS) und besitzt keine Rückkopplung.

Zustandsübersicht

Da es sich um einen Composite-Baustein handelt, ist kein expliziter interner Zustandsautomat vorhanden. Der Ablauf kann wie folgt als einfache Zustandsfolge beschrieben werden:

- **WARTEN**: Der Baustein wartet auf ein Ereignis am Socket AD_IN.
- **KONVERTIEREN**: Nach Ereigniseingang wird die Konvertierung `F_DWORD_TO_SINT` ausgeführt.
- **SENDEN**: Nach Abschluss der Konvertierung wird das Ausgangsereignis am Plug AS_OUT gesendet und der Datenwert weitergegeben.

Damit kehrt der Baustein wieder in den Zustand **WARTEN** zurück.

Anwendungsszenarien

- **Datenanpassung zwischen unterschiedlichen Adaptertypen**: Wenn ein Sendebaustein einen DWORD-Wert (z.B. einen 32-Bit-Zählerstand) über einen AD-Adapter bereitstellt, der Empfänger jedoch einen SINT-Wert über einen AS-Adapter erwartet, wird dieser FB zur Vermittlung eingesetzt.
- **Ereignisgesteuerte Signalaufbereitung**: In Steuerungsanwendungen, in denen Ereignisse asynchron zwischen Adaptern übertragen werden müssen, dient `AD_TO_AS` als Brücke zur Typumwandlung.

Vergleich mit ähnlichen Bausteinen

- **AD_TO_INT / AD_TO_BYTE**: Analoge Konvertierungsbausteine, die DWORD-Werte in andere ganzzahlige Typen (z.B. INT, BYTE) umwandeln.
- **Direkte Typumwandlung**: Im Unterschied zur Verwendung eines einfachen `F_DWORD_TO_SINT` als eingebetteten FB bietet `AD_TO_AS` eine vollständige Adapter-Kapselung, sodass die Schnittstellen ohne zusätzliche Verbindungslogik in ein 4diac-System integriert werden können.

Fazit

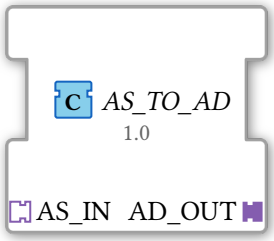
Der Funktionsblock **AD_TO_AS** stellt eine kompakte und standardisierte Lösung zur Konvertierung von DWORD- in SINT-Daten in einer ereignisgesteuerten Adapter-Umgebung dar. Durch die Verwendung der IEC-61131-Bibliotheksfunktion und die klare ereignisbasierte Ausführung ist er zuverlässig und einfach in bestehende Steuerungsprojekte integrierbar. Die unidirektionale Struktur begrenzt die Anwendung auf einen einfachen Datenfluss, was für viele Anwendungsfälle ausreichend ist.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

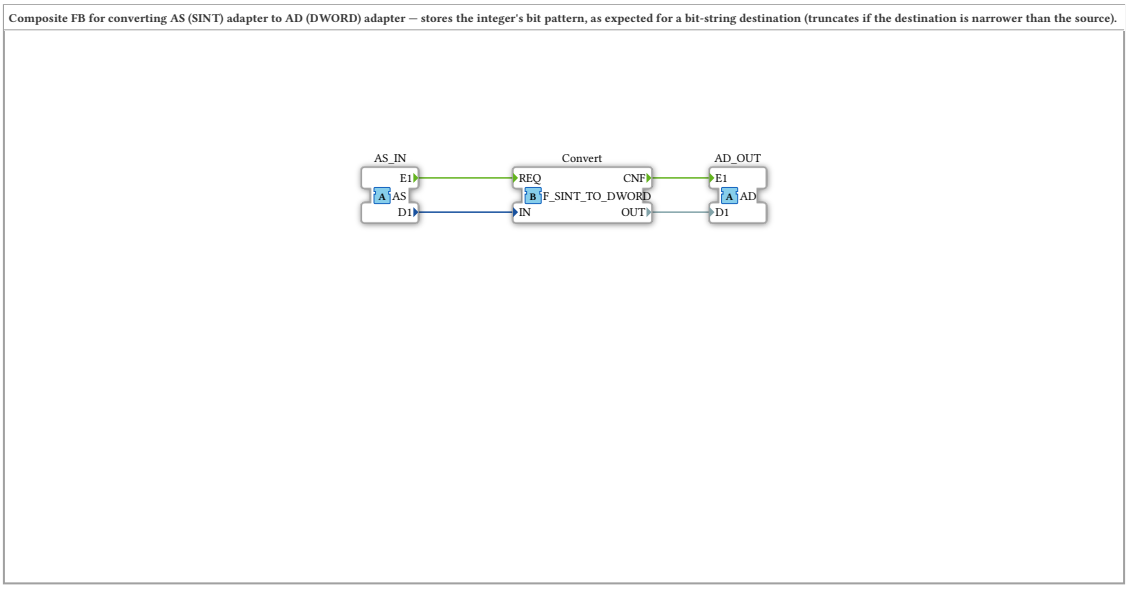
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AS_TO_AD

SINT Adapter Input – *AS*



AD – DWORD Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AS_TO_AD** ist ein zusammengesetzter Baustein (Composite FB), der eine SINT-Adapter-Schnittstelle (AS) in eine DWORD-Adapter-Schnittstelle (AD) umwandelt. Er dient als Konverter zwischen zwei unidirektionalen Adaptertypen und wird typischerweise eingesetzt, um Daten von einem SINT-basierten Signalweg an einen DWORD-basierten Signalweg anzupassen.

Schnittstellenstruktur

Der Baustein besitzt keine eigenen Ereignis- oder Datenein-/ausgänge, sondern stellt ausschließlich Adapter-Schnittstellen zur Verfügung.

Ereignis-Eingänge

- Keine direkten Ereigniseingänge – die Ereignissteuerung erfolgt über den eingehenden Adapter **AS_IN** (Socket). Der Adapter liefert über seinen Ereignisausgang **E1** das Startsignal für die Umwandlung.

Ereignis-Ausgänge

- Keine direkten Ereignisausgänge – die Ereignisausgabe erfolgt über den ausgehenden Adapter **AD_OUT** (Plug). Nach abgeschlossener Umwandlung wird das Ereignis **E1** am ausgehenden Adapter gesendet.

Daten-Eingänge

- Keine direkten Dateneingänge – der zu konvertierende SINT-Wert wird über den Adapter **AS_IN** (Socket) über dessen Datenausgang **D1** bereitgestellt.

Daten-Ausgänge

- Keine direkten Datenausgänge – der konvertierte DWORD-Wert wird über den Adapter **AD_OUT** (Plug) über dessen Dateneingang **D1** ausgegeben.

Adapter

Adaptername	Richtung	Typ	Beschreibung
AS_IN	Socket (Eingang)	adapter::types::unidirectional::AS	Eingangsadapter für den SINT-Wert (inkl. Ereignis E1)
AD_OUT	Plug (Ausgang)	adapter::types::unidirectional::AD	Ausgangsadapter für den konvertierten DWORD-Wert (inkl. Ereignis E1)

Funktionsweise

Der Baustein arbeitet als reine Durchleitung mit Datentypkonvertierung:

1. Über den Socket **AS_IN** wird ein Ereignis am Ausgang **E1** empfangen. Gleichzeitig liegt am Datenausgang **D1** des Sockets ein Wert vom Typ `SINT` an.
2. Das Ereignis wird an den internen Funktionsbaustein **Convert** (Typ `F_SINT_TO_DWORD`) weitergeleitet und triggert dessen Umwandlung.
3. Der `F_SINT_TO_DWORD`-Baustein wandelt den empfangenen `SINT`-Wert in einen `DWORD`-Wert um und gibt das Ergebnis an seinem Ausgang **OUT** aus.
4. Nach erfolgreicher Umwandlung wird das Ereignis **CNF** des internen Bausteins an den Plug **AD_OUT** übergeben, welcher daraufhin sein Ereignis **E1** auslöst. Gleichzeitig wird der konvertierte `DWORD`-Wert an den Dateneingang **D1** des Plugs weitergeleitet.

Die gesamte Verarbeitung erfolgt synchron und ohne Datenpufferung – jede Umwandlung wird unmittelbar durch ein eingehendes Ereignis gestartet und das Ergebnis sofort ausgegeben.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Unidirektionale Adapter**: Der Baustein nutzt ausschließlich unidirektionale Adapter (Socket für Eingang, Plug für Ausgang). Eine Rückwärtskommunikation ist nicht vorgesehen.

- **Wiederverwendbarer Standard-Baustein**: Die Konvertierung wird durch den IEC-61131-konformen Baustein `F_SINT_TO_DWORD` realisiert, der in der Bibliothek `iec61131::conversion` bereitsteht.
- **Composite-Architektur**: Der Baustein ist als Composite FB implementiert und erlaubt eine einfache Anpassung oder Wiederverwendung des internen Netzwerks.
- **Lizenz**: Der Baustein steht unter der Eclipse Public License 2.0 (EPL-2.0) und enthält einen entsprechenden Copyright-Hinweis.

Zustandsübersicht

Der Baustein besitzt keinen eigenen Zustandsautomaten. Seine Funktionsweise ist rein ereignisgesteuert und datenflussbasiert:

1. **Warten auf Ereignis** – Der interne FB `Convert` wartet auf ein `REQ`-Ereignis (bereitgestellt durch **AS_IN.E1**).
2. **Konvertierung** – Ein eingehendes Ereignis startet die Umwandlung des SINT-Werts in einen DWORD-Wert.
3. **Ausgabe** – Nach Abschluss wird das Ereignis **CNF** vom internen FB erzeugt und der konvertierte Wert an den Ausgangsadapter weitergegeben.

> **Hinweis:** Da keine Verzögerung oder Pufferung stattfindet, kann während der Konvertierung kein weiteres Ereignis bearbeitet werden.

Anwendungsszenarien

- **Anbindung von Sensoren**: Ein Sensor liefert Daten als SINT (z. B. 8-Bit-Werte), während die Steuerungseinheit ausschließlich DWORD-Werte verarbeitet.
- **Systemintegration**: Bestehende Kommunikationsschnittstellen auf SINT-Basis sollen an ein System angebunden werden, das DWORD-Adapter erwartet.
- **Protokollumsetzung**: In modularen Fertigungsanlagen werden verschiedene Adaptertypen verwendet, die durch solche Konverter miteinander verbunden werden können.

Vergleich mit ähnlichen Bausteinen

- **BYTE_TO_WORD, BYTE_TO_DWORD, WORD_TO_DWORD** – Entsprechende Konverter für andere Datentypen sind in der IEC-61131-Bibliothek vorhanden.
- **AS_TO_AD** unterscheidet sich durch die Verwendung von Adapter-Schnittstellen statt direkter Ein-/Ausgänge. Dadurch eignet er sich besonders für den modularen Aufbau von Adapterketten.
- **Direkte Konverter-FBs** wie `F_SINT_TO_DWORD` bieten eine einfachere Schnittstelle (Ein-/Ausgänge), während der Composite FB den Vorteil der Adapterkompatibilität hat.

Fazit

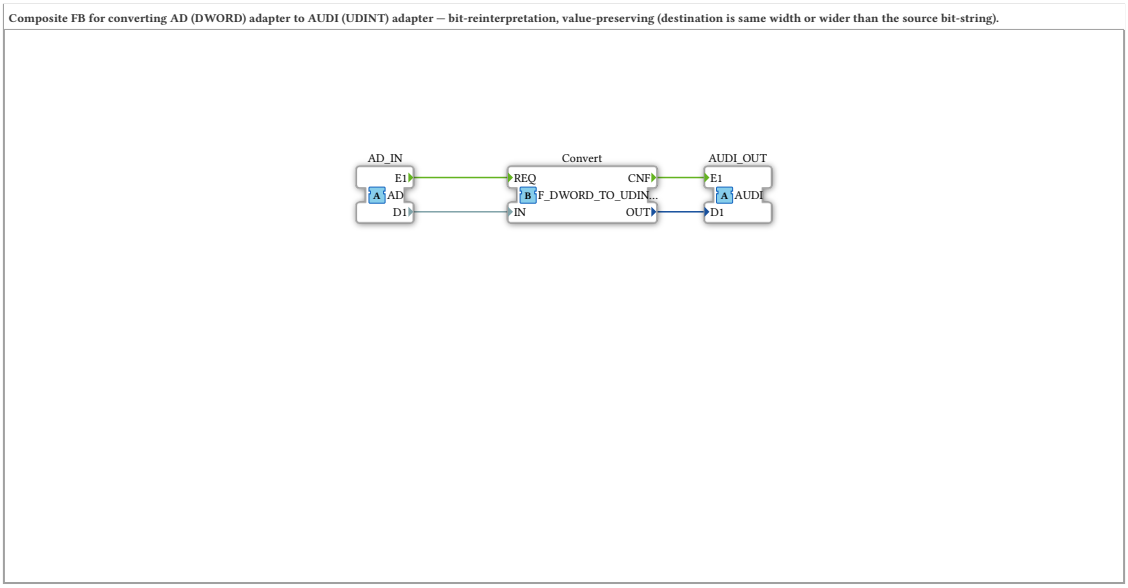
Der **AS_TO_AD**-Funktionsblock ist ein spezialisierter Adapter-Konverter, der eine nahtlose Integration eines SINT-basierten Datenpfades in eine DWORD-basierte Architektur ermöglicht. Durch die Nutzung des standardisierten Bausteins `F_SINT_TO_DWORD` und die Implementierung als Composite FB ist er zuverlässig, erweiterbar und einfach in bestehende 4diac-Projekte einzubinden. Seine unidirektionale Auslegung macht ihn ideal für Anwendungen, bei denen Daten nur in eine Richtung fließen müssen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AD_AUDI Conversion

AD_TO_AUDI

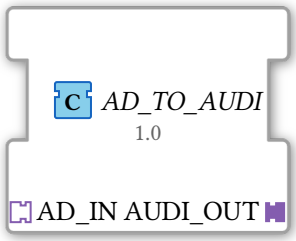


Einleitung

Der AD_TO_AUDI Funktionsblock ist ein spezialisierter Adapter-Konvertierungsbaustein. Er dient dazu, einen unidirektionalen **AD** (DWORD) Adapter-Socket in einen unidirektionalen **AUDI** (UDINT) Adapter-Plug umzuwandeln.

Intern nutzt der Baustein die Standard-Konvertierungsfunktion `F_DWORD_TO_UDINT`, um eine saubere Typwandlung zwischen dem Bit-String-Format (DWORD) und dem numerischen Format (UDINT) sicherzustellen.

DWORD Adapter Input – *AD*



AUDI – UDINT Adapter Output

Schnittstellenstruktur

Adapter

- **AD_IN** (Socket): Eingangsadapter vom Typ `AD` (DWORD).
- **AUDI_OUT** (Plug): Ausgangsadapter vom Typ `AUDI` (UDINT).

Funktionsweise

Sobald am `AD_IN` ein Ereignis (`E1`) zusammen mit Daten (`D1`) eintrifft, wird der DWORD-Wert intern in einen UDINT-Wert gewandelt. Dieser wird anschließend sofort über den `AUDI_OUT` Adapter (Ereignis `E1`, Daten `D1`) weitergegeben.

Technische Besonderheiten

- **Bit-Reinterpretation, werterhaltend** (Zielbreite ist gleich groß oder größer als der Quell-Bit-String).

✓ **Direkte Adapter-zu-Adapter Kopplung**. ✓ **Normgerechte Wandlung** durch Verwendung von `F_DWORD_TO_UDINT`. ✓ **Ereignisgesteuert**: Jede Aktualisierung am Eingang löst sofort eine Aktualisierung am Ausgang aus.

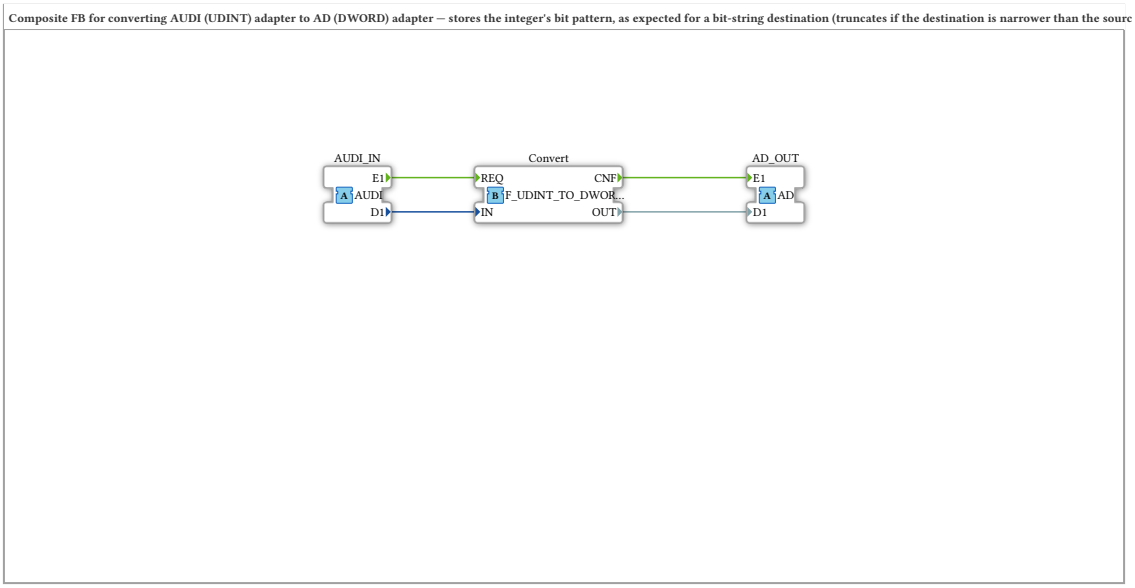
Anwendungsszenarien

- Anbindung von ISOBUS-Eingabebausteinen (die oft DWORD liefern) an numerische Verarbeitungslogik auf Adapter-Basis.
- Protokoll- und Typanpassung in komplexen Baustein-Netzwerken.

🛠️ Zugehörige Übungen

- Uebung_011_AUDI

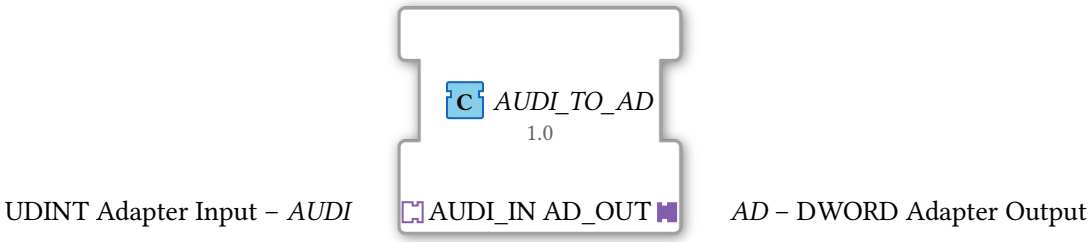
AUDI_TO_AD



Einleitung

Der AUDI_TO_AD Funktionsblock ist ein spezialisierter Adapter-Konvertierungsbaustein. Er dient dazu, einen unidirektionalen **AUDI** (UDINT) Adapter-Socket in einen unidirektionalen **AD** (DWORD) Adapter-Plug umzuwandeln.

Intern nutzt der Baustein die Standard-Konvertierungsfunktion `F_UDINT_TO_DWORD`, um eine saubere Typwandlung zwischen dem numerischen Format (UDINT) und dem Bit-String-Format (DWORD) sicherzustellen.



UDINT Adapter Input – *AUDI*

AD – DWORD Adapter Output

Schnittstellenstruktur

Adapter

- **AUDI_IN** (Socket): Eingangsadapter vom Typ `AUDI` (UDINT).
- **AD_OUT** (Plug): Ausgangsadapter vom Typ `AD` (DWORD).

Funktionsweise

Sobald am `AUDI_IN` ein Ereignis (`E1`) zusammen mit Daten (`D1`) eintrifft, wird der UDINT-Wert intern in einen DWORD-Wert gewandelt. Dieser wird anschließend sofort über den `AD_OUT` Adapter (Ereignis `E1`, Daten `D1`) weitergegeben.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).

✓ **Direkte Adapter-zu-Adapter Kopplung**. ✓ **Normgerechte Wandlung** durch Verwendung von `F_UDINT_TO_DWORD`. ✓ **Ereignisgesteuert**: Jede Aktualisierung am Eingang löst sofort eine Aktualisierung am Ausgang aus.

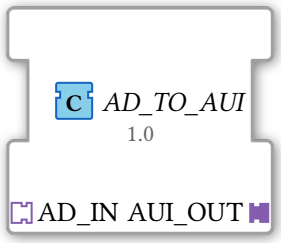
Anwendungsszenarien

- Rückführung von numerischen Berechnungsergebnissen an ISOBUS-Ausgabebausteine (z.B. Bargraph oder numerische Anzeige), die oft DWORD als Eingangsformat erwarten.
- Kompatibilitätsschicht in adapterbasierten Steuerungsarchitekturen.

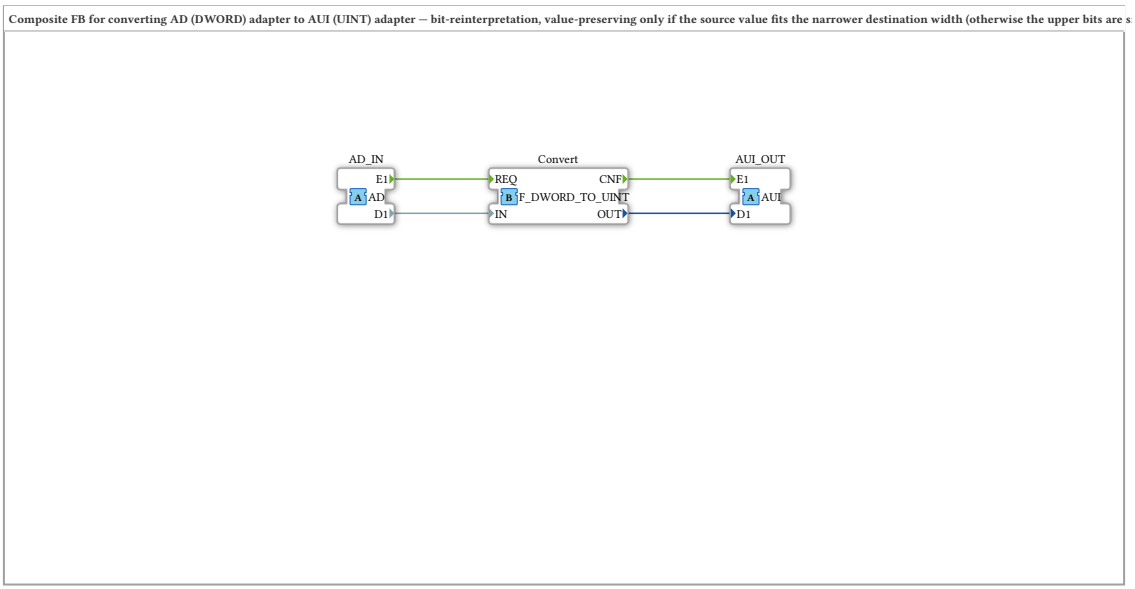
AD_AUI

AD_TO_AUI

DWORD Adapter Input – *AD*



AUI – UINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `AD_TO_AUI` ist ein Composite-Baustein, der einen AD-Adapter (DWORD-basiert) in einen AUI-Adapter (UINT-basiert) umwandelt. Er ermöglicht die nahtlose Integration von Komponenten, die mit unterschiedlichen Datentypen arbeiten, indem er die eingehenden DWORD-Werte in UINT-Werte konvertiert und die entsprechenden Ereignisse weiterleitet.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung

(keine eigenen)	-	Der FB besitzt keine eigenen Ereignis-Eingänge. Die Ereignissteuerung erfolgt über den Adapter-Socket `AD_IN`.

Ereignis-Ausgänge

Name	Typ	Beschreibung
(keine eigenen)	-	Der FB besitzt keine eigenen Ereignis-Ausgänge. Die Ereignisweiterleitung erfolgt über den Adapter-Plug `AUI_OUT`.

Daten-Eingänge

Name	Datentyp	Beschreibung
(keine eigenen)	-	Alle Daten werden über den Adapter-Socket `AD_IN` bereitgestellt.

Daten-Ausgänge

Name	Datentyp	Beschreibung
(keine eigenen)	-	Alle Daten werden über den Adapter-Plug `AUI_OUT` ausgegeben.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AD`	`AD_IN`	Socket (Eingang)	Nimmt DWORD-Werte und zugehörige Ereignisse entgegen.
`adapter::types::unidirectional::AUI`	`AUI_OUT`	Plug (Ausgang)	Gibt konvertierte UINT-Werte und zugehörige Ereignisse aus.

Der Adapter `AD_IN` stellt ein Ereignis `E1` und ein Daten-Element `D1` (Typ DWORD) bereit. Der Adapter `AUI_OUT` erwartet ein Ereignis `E1` und ein Daten-Element `D1` (Typ UINT). Die Zuordnung erfolgt wie in der Funktionsweise beschrieben.

Funktionsweise

Der FB enthält eine Instanz des Standard-Konvertierungsbausteins `iec61131::conversion::F_DWORD_TO_UINT`. Die interne Verschaltung:

1. **Ereignissteuerung**:
   - Das Ereignis `AD_IN.E1` wird direkt an den Konvertierungsbaustein (`Convert.REQ`) weitergeleitet.
   - Nach erfolgreicher Konvertierung signalisiert `Convert.CNF` das Ereignis an `AUI_OUT.E1`.
2. **Datenfluss**:
   - Der Datenwert `AD_IN.D1` (DWORD) wird an den Eingang `Convert.IN` übergeben.
   - Der Ausgang `Convert.OUT` liefert den umgewandelten Wert (UINT) an `AUI_OUT.D1`.

Der FB führt somit eine ereignisgesteuerte, unidirektionale Typkonvertierung von DWORD nach UINT durch. Die Wandlung erfolgt nach der IEC 61131-3 Standardfunktion `DWORD_TO_UINT`.

Technische Besonderheiten

- **Bit-Reinterpretation**, werterhaltend nur, wenn der Quellwert in die schmalere Zielbreite passt (sonst werden die oberen Bits stillschweigend verworfen).
- **Composite-Baustein**: Der FB kapselt einen bereits bestehenden Standard-Konvertierungsbaustein und zwei Adapter-Schnittstellen. Dies vereinfacht die Wiederverwendung in größeren Applikationen.
- **Datentyp-Konvertierung**: Die Umwandlung von `DWORD` (32-Bit, vorzeichenlos) in `UINT` (16-Bit, vorzeichenlos) kann je nach Wertbereich zu Informationsverlust führen (z. B. wenn der DWORD-Wert größer als 65535 ist). Der Anwender ist für die Wertebereichsüberwachung verantwortlich.

- **Unidirektionale Schnittstelle**: Sowohl der Eingangs- als auch der Ausgangs-Adapter sind unidirektional ausgelegt (nur eine Ereignis-/Datenrichtung).
- **Lizenz**: Der Baustein ist unter der Eclipse Public License 2.0 (EPL-2.0) verfügbar.

Zustandsübersicht

Der `AD_TO_AUI`-Baustein besitzt keinen eigenen internen Zustand. Das Verhalten ist rein kombinatorisch in Verbindung mit der ereignisgesteuerten Ausführung des enthaltenen Konvertierungsbausteins. Es gibt keine Initialisierungs- oder Fehlerzustände; die Konvertierung erfolgt bei jedem eingehenden Ereignis sofort.

Anwendungsszenarien

- **Schnittstellenanpassung**: Wenn ein vorhandener Funktionsblock oder ein Subsystem nur einen AUI-Adapter (UINT) unterstützt, aber die Datenquelle einen AD-Adapter (DWORD) liefert.
- **Datenreduktion**: Gezieltes Kürzen eines 32-Bit-Wertes auf 16 Bit für nachgelagerte Verarbeitungslogik.
- **Integration in bestehende Steuerungen**: Austausch von Adaptern zwischen verschiedenen Bussystemen oder Kommunikationsprotokollen, die unterschiedliche Datentypen verwenden.

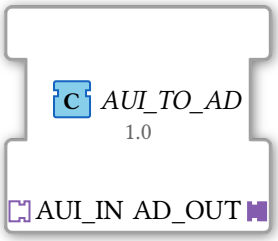
Vergleich mit ähnlichen Bausteinen

- **AUI_TO_AD**: Führt die umgekehrte Konvertierung (UINT nach DWORD) durch. Beide Bausteine sind symmetrisch aufgebaut und nutzen die entsprechenden Standard-Konvertierungsfunktionen.
- **Direkte Typumwandlung**: Eine manuelle Implementierung mit einem `F_DWORD_TO_UINT`-Baustein und Adaptern wäre möglich, aber der Composite-Baustein bietet eine gekapselte, wiederverwendbare Lösung.

Fazit

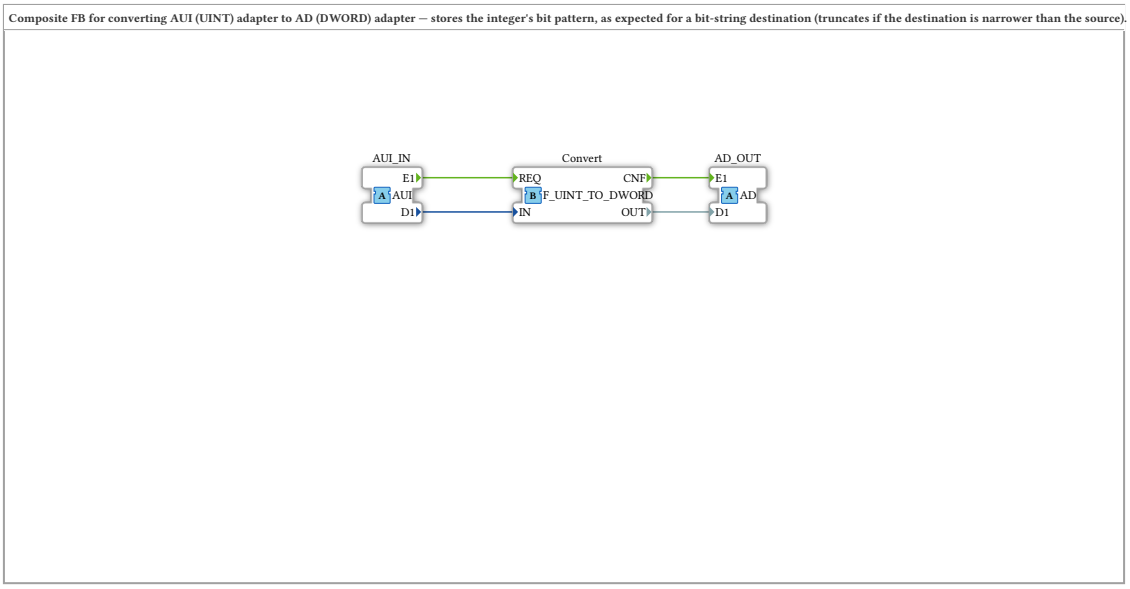
Der `AD_TO_AUI`-Baustein bietet eine einfache und zuverlässige Möglichkeit, zwischen zwei gängigen Adaptertypen zu konvertieren. Er reduziert den Entwicklungsaufwand, indem er die notwendige Verschaltung und Konvertierung kapselt und als wiederverwendbare Komponente bereitstellt. Trotz der potenziellen Wertebereichseinschränkung (DWORD → UINT) ist er ein nützliches Werkzeug für die Integration heterogener Systemteile in der Steuerungstechnik.

AUI_TO_AD



UINT Adapter Input – *AUI*

AD – DWORD Adapter Output



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock **AUI_TO_AD** ist ein zusammengesetzter Baustein (Composite FB), der eine Umwandlung eines AUI-Adapters (UINT-Datentyp) in einen AD-Adapter (DWORD-Datentyp) realisiert. Er dient als unidirektionale Konvertierungsschnittstelle innerhalb einer IEC 61499-basierten Steuerungsanwendung. Der Baustein ist als Kapselung des Standardkonverters `F_UINT_TO_DWORD` aus der Bibliothek `iec61131::conversion` implementiert und ermöglicht eine adapterbasierte Datenweitergabe.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Beschreibung
AUI_IN.E1	Ereignis vom AUI-Adapter, das die Konvertierung auslöst.

Ereignis-Ausgänge

Name	Beschreibung
AD_OUT.E1	Ereignis zum AD-Adapter, das die Verfügbarkeit des konvertierten Wertes signalisiert.

Daten-Eingänge

Name	Datentyp	Beschreibung
AUI_IN.D1	UINT	Eingangswert vom AUI-Adapter, der in einen DWORD-Wert umgewandelt wird.

Daten-Ausgänge

Name	Datentyp	Beschreibung
AD_OUT.D1	DWORD	Ausgangswert, der den konvertierten DWORD-Wert an den AD-Adapter weitergibt.

Adapter

Typ	Name	Rolle

adapter::types::unidirectional::AUI	AUI_IN	Socket – Eingangsadapter für den UINT-Wert und das zugehörige Ereignis.
adapter::types::unidirectional::AD	AD_OUT	Plug – Ausgangsadapter für den konvertierten DWORD-Wert und das Bestätigungsereignis.

Funktionsweise

Der Baustein **AUI_TO_AD** arbeitet als Kapselung des Funktionsbausteins F_UINT_TO_DWORD. Sobald ein Ereignis am Ereigniseingang AUI_IN.E1 eintrifft, wird der angeschlossene interne Konverter Convert aufgerufen. Dieser Konverter liest den Wert vom Dateneingang AUI_IN.D1 (Typ UINT) und wandelt ihn in einen DWORD-Wert um. Das Ergebnis wird am Datenausgang AD_OUT.D1 bereitgestellt. Gleichzeitig wird nach erfolgreicher Konvertierung ein Ereignis am Ausgang AD_OUT.E1 erzeugt, welches dem nachgeschalteten AD-Adapter die Fertigstellung signalisiert.

Die Umwandlung erfolgt nach der IEC 61131-3 Norm für die Funktion UINT_TO_DWORD. Der Wertebereich des UINT-Eingangs (0 bis 65535) wird auf den unteren 16 Bit des DWORD-Ausgangs abgebildet; die oberen 16 Bit werden mit Null aufgefüllt.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- Der Baustein ist als Composite FB realisiert und enthält keine eigene Zustandsautomaten (ECC). Das Verhalten wird vollständig durch den integrierten Konverter F_UINT_TO_DWORD bestimmt.
- Die Verbindung zwischen den Adaptern und dem internen Konverter erfolgt über Ereignis- und Datenverbindungen.
- Der Baustein ist für den unidirektionalen Datentransfer konzipiert – es gibt keine Rückwärtskonvertierung von AD zu AUI.
- Die Paketangabe (packageName) lautet adapter::conversion::unidirectional, was auf eine klare Strukturierung der Konvertierungsadapter hindeutet.

Zustandsübersicht

Als Composite FB besitzt **AUI_TO_AD** keine eigene Zustandsmaschine. Der Funktionsablauf ist rein ereignisgesteuert: Ein Ereignis am Eingang löst die Umwandlung aus, nach deren Abschluss ein Ereignis am Ausgang generiert wird. Zwischen diesen Ereignissen gibt es keine internen Zustände.

Anwendungsszenarien

- **Datenintegration**: Ein AUI-Adapter liefert einen UINT-Wert (z. B. aus einem Sensormodul), der in einem System benötigt wird, das ausschließlich DWORD-Adressen (z. B. über einen AD-Adapter) verarbeitet.
- **Protokollanpassung**: Wenn vorhandene Komponenten mit unterschiedlichen Datentypen arbeiten, kann dieser Baustein als einfacher Konverter zwischen den Adapterschnittstellen eingesetzt werden.
- **Modularisierung**: Innerhalb einer 4diac-Applikation können so standardisierte Konvertierungsbausteine als Adapter zusammengefasst werden, um die Wiederverwendbarkeit zu erhöhen.

Vergleich mit ähnlichen Bausteinen

Es existieren vergleichbare Konvertierungsbausteine für andere Datentypen, z. B. WORD_TO_DWORD, BYTE_TO_DWORD oder generische TO_DWORD-Funktionen. Der Besonderheit von **AUI_TO_AD** liegt in der Kapselung der Adapterprotokolle: Statt einer direkten Datenverbindung werden die Werte über Adapter bereitgestellt, was eine lose Kopplung zwischen den Komponenten ermöglicht. Im Gegensatz zu einem direkten Aufruf von F_UINT_TO_DWORD in einer Applikation bietet dieser FB eine standardisierte, wiederverwendbare Schnittstelle auf Adapterebene.

Fazit

Der Baustein **AUI_TO_AD** bietet eine saubere, auf Adapter basierende Lösung zur Umwandlung von UINT- in DWORD-Werte. Er ist einfach zu verwenden, erfordert keine eigene Programmierung und fügt sich nahtlos in die 4diac-Welt mit Ereignis- und Adapternetzwerken ein. Durch die Kapselung des Stan-

dardkonverters bleibt die Funktionalität robust und normkonform. Der Baustein eignet sich besonders für Anwendungen, bei denen unterschiedliche Adaptertypen harmonisiert werden müssen.

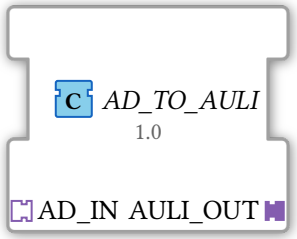
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

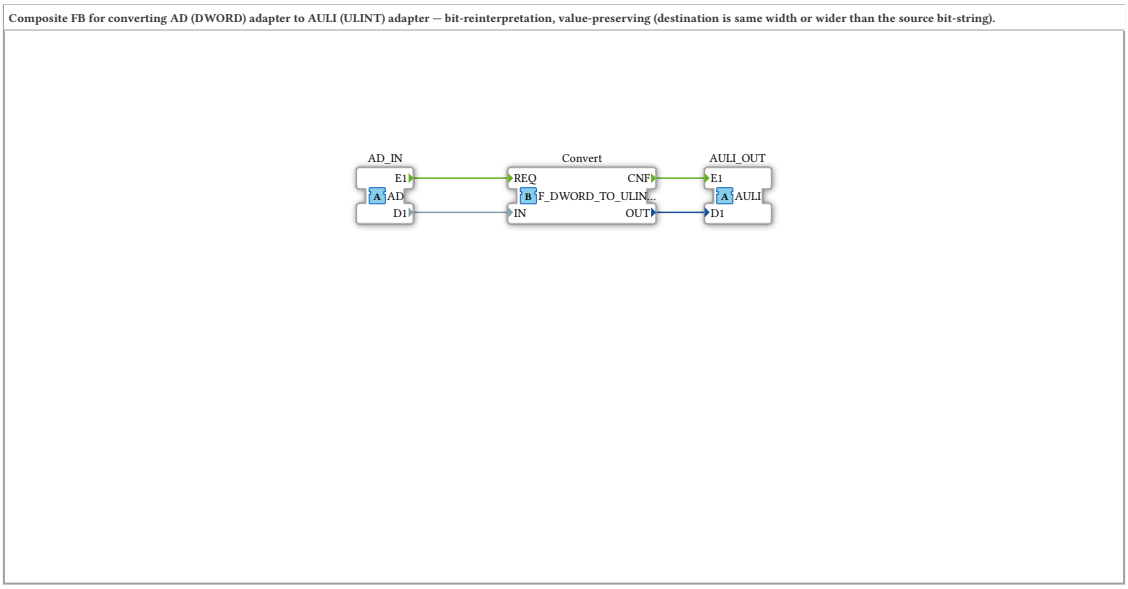
AD_AULI

AD_TO_AULI

DWORD Adapter Input – *AD*



AULI – ULINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein AD_TO_AULI ist ein Composite-FB zur Konvertierung eines DWORD-Adapters (AD) in einen ULINT-Adapter (AULI). Er verwendet die IEC 61131-Konvertierungsfunktion F_DWORD_TO_ULINT und kapselt die Ereignis- und Datenverbindungen für eine einfache adapterbasierte Umwandlung.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

- **AD_IN** (Socket): Typ AD (DWORD), unidirektionaler Eingangsadapter für das DWORD.
- **AULI_OUT** (Plug): Typ AULI (ULINT), unidirektionaler Ausgangsadapter für das konvertierte ULINT.

Funktionsweise

Der Baustein nimmt ein DWORD über den AD_IN-Adapter entgegen. Ein Ereignis am E1-Eingang von AD_IN triggert den REQ-Ereigniseingang des internen Konverters F_DWORD_TO_ULINT. Dieser konvertiert den Datenwert (AD_IN.D1) und signalisiert nach Abschluss über CNF. Das CNF-Ereignis löst dann das E1-Ereignis am AULI_OUT-Adapter aus, und der konvertierte Wert (ULINT) wird über AULI_OUT.D1 ausgegeben.

Technische Besonderheiten

- **Bit-Reinterpretation, werterhaltend** (Zielbreite ist gleich groß oder größer als der Quell-Bit-String).
- Der Baustein ist als Composite realisiert und nutzt die Bibliothek `iec61131::conversion::F_DWORD_TO_ULINT`.
- Es werden keine direkten Ereignis- oder Dateneingänge/-ausgänge verwendet; die gesamte Kommunikation erfolgt ausschließlich über Adapter.
- Die Konvertierung erfolgt nach IEC 61499-Ereignissteuerung: Das Ausgangsereignis wird erst nach Abschluss der Konvertierung gesendet.

Zustandsübersicht

Der Composite-Baustein selbst besitzt keine eigenen Zustände. Der integrierte Funktionsblock F_DWORD_TO_ULINT durchläuft folgende Zustände:

- **IDLE**: Warten auf ein REQ-Ereignis.
- **REQ**: Konvertierung wird durchgeführt.
- **CNF**: Konvertierung abgeschlossen, Ausgangsdaten werden aktualisiert.

Anwendungsszenarien

- Umwandlung von Messwerten oder Steuerdaten, die als DWORD (32 Bit) vorliegen, in einen ULINT (64 Bit) Adapter für nachfolgende Verarbeitung.
- Einsatz in modularen Automatisierungssystemen, die auf Adapter-Schnittstellen basieren.
- Brücke zwischen Komponenten mit unterschiedlichen Adapter-Typen.

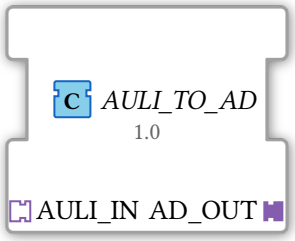
Vergleich mit ähnlichen Bausteinen

- Direkte Typumwandlungs-FBs wie `F_DWORD_TO_ULINT` arbeiten auf Datenebene, während AD_TO_AULI die Adapter-Schnittstellen kapselt.
- Ähnliche Composite-Bausteine existieren für andere Datentyp-Kombinationen (z. B. DWORD_TO_LWORD), wobei hier speziell die unidirektionalen Adapter AD und AULI verwendet werden.

Fazit

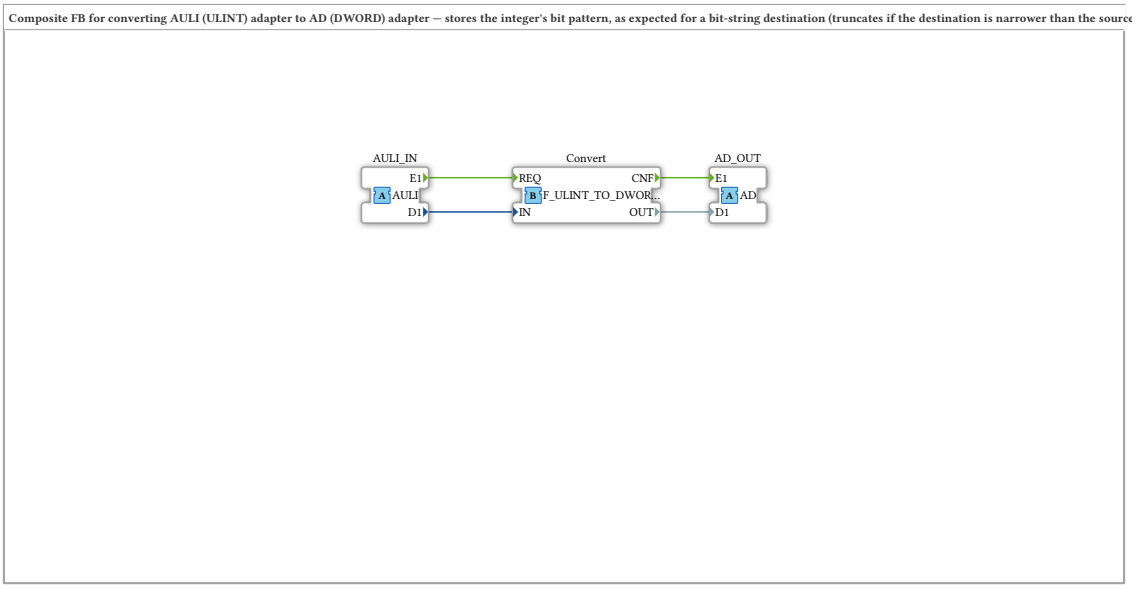
AD_TO_AULI bietet eine einfache und wiederverwendbare Lösung zur adapterbasierten Konvertierung von DWORD zu ULINT. Durch die Kapselung der Konvertierungslogik in einem Composite-FB wird die Wiederverwendbarkeit erhöht und die Netzwerkkomplexität reduziert.

AULI_TO_AD



ULINT Adapter Input – *AULI*

AD – DWORD Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AULI_TO_AD** ist ein Composite-Baustein, der einen unidirektionalen Adapter vom Typ **AULI** (ULINT) in einen unidirektionalen Adapter vom Typ **AD** (DWORD) umwandelt. Er dient als Brücke zwischen Komponenten, die Daten im ULINT-Format bereitstellen, und solchen, die ein DWORD-Format erwarten. Die Umwandlung erfolgt mithilfe der IEC 61131-Funktion `F_ULINT_TO_DWORD`.

Schnittstellenstruktur

Der Baustein besitzt keine eigenen Ereignis- oder Datenein-/ausgänge, sondern kommuniziert ausschließlich über zwei Adapter:

Ereignis-Eingänge

Adapter	Ereignis	Beschreibung
`AULI_IN`	`E1`	Startet die Konvertierung

Ereignis-Ausgänge

Adapter	Ereignis	Beschreibung
`AD_OUT`	`E1`	Signalisiert abgeschlossene Konvertierung

Daten-Eingänge

Adapter	Variable	Datentyp	Beschreibung
`AULI_IN`	`D1`	ULINT	Eingangswert für die Umwandlung

Daten-Ausgänge

Adapter	Variable	Datentyp	Beschreibung
`AD_OUT`	`D1`	DWORD	Konvertierter Ausgangswert

Adapter

Name	Richtung	Typ	Beschreibung
`AULI_IN`	Socket	`adapter::types::unidirectional::AULI`	Eingangsadapter (ULINT)

AD_OUT	Plug	adapter::types::unidirectional::AD	Ausgangsadapter (DWORD)

Funktionsweise

Der Baustein verbindet intern den Socket AULI_IN mit dem Funktionsbaustein F_ULINT_TO_DWORD und dessen Ausgang mit dem Plug AD_OUT. Die Ablaufsteuerung erfolgt über Ereignisse:

1. Ein eingehendes Ereignis an AULI_IN.E1 aktiviert den Eingang der Konvertierungsfunktion.
2. Der Datenwert AULI_IN.D1 (ULINT) wird an den Eingang IN der Funktion übergeben.
3. Nach der Umwandlung wird das Ergebnis (DWORD) am Ausgang OUT bereitgestellt.
4. Gleichzeitig wird ein Ereignis an AD_OUT.E1 ausgegeben, um den angeschlossenen Baustein über die neuen Daten zu informieren.

Der gesamte Vorgang ist ereignisgesteuert und erfolgt ohne jegliche Verzögerung oder Zustandsspeicherung.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Composite-Struktur**: Der Baustein ist als Netzwerk aus einem einzigen Funktionsbaustein (F_ULINT_TO_DWORD) realisiert. Alle Verbindungen sind fest verdrahtet.
- **Paket-Import**: Die verwendete Konvertierungsfunktion stammt aus dem Paket iec61131::conversion. Der Baustein ist im Paket adapter::conversion::unidirectional organisiert.
- **Keine Zustandshaltung**: Der Baustein ist rein kombinatorisch – jedes Eingangsereignis erzeugt sofort ein Ausgangsereignis mit dem umgewandelten Wert.
- **Typsicherheit**: Die Adapter-Schnittstellen sind strikt auf ULINT bzw. DWORD festgelegt, sodass Fehlverbindungen bereits zur Entwicklungszeit erkannt werden.

Zustandsübersicht

Da es sich um einen Composite-Baustein ohne eigene Algorithmen handelt, existiert keine explizite Zustandsmaschine. Der Gesamtzustand ergibt sich ausschließlich aus dem internen Zustand des verwendeten Funktionsbausteins F_ULINT_TO_DWORD, der nach jedem Ausführungszyklus in den Ruhezustand zurückkehrt. Im Normalbetrieb wechselt der Baustein zwischen:

- **Warten**: Auf ein Ereignis an AULI_IN.E1.
- **Konvertieren**: Kurze Ausführung von F_ULINT_TO_DWORD.
- **Ausgeben**: Senden des Ergebnisses und des Ereignisses an AD_OUT.E1.

Anwendungsszenarien

- **Datenanpassung in der Landtechnik**: Umwandlung von ULINT-Werten (z. B. von Sensoren oder Steuergeräten) in das von nachgelagerten Komponenten geforderte DWORD-Format.
- **Adapter-basierte Kommunikation**: Einsatz in Modulen, die ausschließlich über Adapter-Schnittstellen miteinander verbunden werden, ohne direkte Ein-/Ausgangspins.
- **Systemintegration**: Wenn ein bestehendes System ULINT-Daten liefert, ein anderes aber nur DWORD-Adapter akzeptiert, dient dieser Baustein als einfacher Konverter.

Vergleich mit ähnlichen Bausteinen

Baustein	Beschreibung
F_ULINT_TO_DWORD	Einfache Konvertierungsfunktion, jedoch ohne Adapter-Einbindung.
AULI_TO_AD	Gleiche Funktionalität, aber als Composite-Baustein mit Adapter-Schnittstellen.
Andere Adapter-Konverter	Es existieren Bausteine für verwandte Typen (z. B. LINT, DINT), die nach demselben Prinzip arbeiten.

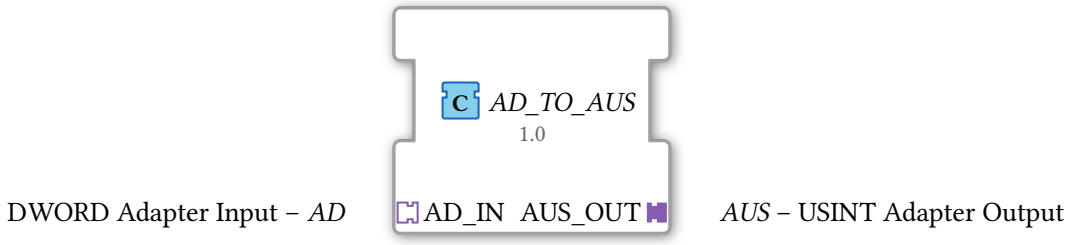
Der wesentliche Unterschied liegt in der Schnittstellenart: Während die reine Funktion direkte Daten- und Ereignispins verwendet, kapselt AULI_TO_AD die Konvertierung in eine adapterbasierte Lösung.

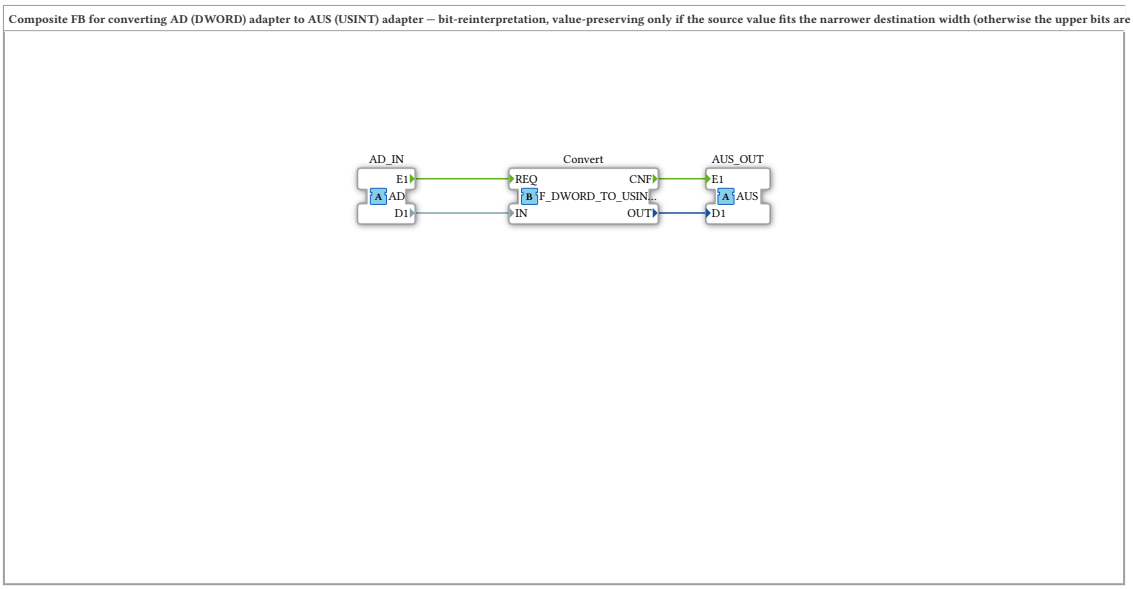
Fazit

Der Funktionsblock `AULI_TO_AD` bietet eine saubere, standardisierte Möglichkeit, ULINT-Daten über einen AULI-Adapter in das DWORD-Format eines AD-Adapters zu überführen. Dank seiner Composite-Natur ist er einfach einsetzbar, erfordert keine eigene Programmierung und fügt sich nahtlos in eine adapterorientierte Architektur ein. Er ist besonders nützlich in Umgebungen, in denen Komponenten über definierte Adapter-Schnittstellen kommunizieren.

AD_AUS

AD_TO_AUS





• – ○ • – ○ • – ○ •

Einleitung

Der **AD_TO_AUS** ist ein zusammengesetzter Funktionsblock (Composite FB), der einen DWORD-Adapter (AD) in einen USINT-Adapter (AUS) umwandelt. Er dient als unidirektionale Schnittstelle zur Konvertierung eines 32-Bit-Datenwerts in einen 8-Bit-Wert. Der Baustein kapselt die IEC-61131-Konvertierungsfunktion `F_DWORD_TO_USINT` und stellt die Adapter-Schnittstellen bereit, sodass er nahtlos in eine Adapter-basierte Kommunikationsstruktur eingebunden werden kann.

Schnittstellenstruktur

Ereignis-Eingänge

Über den Adapter-Eingang **AD_IN** wird das Ereignis **E1** empfangen. Es löst die Konvertierung des anliegenden DWORD-Werts aus.

Ereignis-Ausgänge

Über den Adapter-Ausgang **AUS_OUT** wird das Ereignis **E1** gesendet, sobald die Konvertierung abgeschlossen ist und der umgewandelte USINT-Wert bereitsteht.

Daten-Eingänge

Der Adapter-Eingang **AD_IN** liefert einen DWORD-Wert (32-Bit) über die Datenleitung **D1**.

Daten-Ausgänge

Der Adapter-Ausgang **AUS_OUT** stellt den konvertierten USINT-Wert (8-Bit) über die Datenleitung **D1** zur Verfügung.

Adapter

Adapter	Typ	Richtung	Beschreibung
AD_IN	`adapter::types::unidirectional::AD`	Eingang (Socket)	Nimmt DWORD-Daten und Steuerereignisse auf.
AUS_OUT	`adapter::types::unidirectional::AUS`	Ausgang (Plug)	Gibt USINT-Daten und Bestätigungsereignisse aus.

Funktionsweise

1. Der FB wartet auf ein Ereignis am Adapter-Eingang **AD_IN** (Signal **E1**).
2. Mit Eintreffen des Ereignisses wird der über **AD_IN.D1** anliegende DWORD-Wert an den internen FB `F_DWORD_TO_USINT` weitergeleitet und dessen `REQ`-Eingang aktiviert.
3. Der integrierte Konvertierungsbaustein führt die Typumwandlung von DWORD nach USINT durch (Typkonvertierung gemäß IEC 61131-3).
4. Nach Abschluss der Konvertierung signalisiert der interne FB das Ereignis `CNF`, welches an den Adapter-Ausgang **AUS_OUT** (Signal **E1**) weitergegeben wird. Gleichzeitig wird der konvertierte USINT-Wert über **AUS_OUT.D1** bereitgestellt.

Technische Besonderheiten

- **Bit-Reinterpretation**, werterhaltend nur, wenn der Quellwert in die schmalere Zielbreite passt (sonst werden die oberen Bits stillschweigend verworfen).
- **Adapter-Kopplung:** Der FB besitzt keine direkten Ereignis- oder Daten-Ein-/Ausgänge, sondern kommuniziert ausschließlich über Adapter-Schnittstellen. Dies ermöglicht eine lose Kopplung in einer modularen Steuerungsarchitektur.
- **Verwendete Bibliothek:** Die Konvertierung erfolgt mit dem standardisierten Baustein `iec61131::conversion::F_DWORD_TO_USINT`. Dadurch wird die IEC-61131-Konformität gewährleistet.
- **Paketname:** `adapter::conversion::unidirectional` (siehe CompilerInfo) ordnet den FB in eine spezifische Adapter-Konvertierungs-Bibliothek ein.

Zustandsübersicht

Der **AD_TO_AUS** ist ein reiner Funktionsblock ohne eigenen Zustandsautomaten. Er arbeitet streng kausal: Jedes Eingangsereignis führt genau zu einem Ausgangsereignis. Die Latenz ergibt sich aus der Ausführungszeit des internen Konvertierungsbausteins.

Anwendungsszenarien

- **Schnittstellenanpassung:** Wenn ein Sensor oder Aktor Daten im DWORD-Format (32 Bit) liefert, die Steuerung aber nur USINT-Werte (8 Bit) verarbeiten kann, wird der FB zwischengeschaltet.
- **Adapter-basierte Systeme:** In einer 4diac-Umgebung, die auf unidirektionale Adapter setzt, kann der FB nahtlos in Datenfluss-Pfade eingefügt werden, ohne die Ereignis-/Daten-Struktur der umgebenden Komponenten zu ändern.
- **Datenreduktion:** Gezieltes Abschneiden von höherwertigen Bytes, wenn nur der niederwertige Teil eines DWORDs relevant ist (z. B. bei Zählern oder einfachen Analogwerten).

Vergleich mit ähnlichen Bausteinen

- **AD_TO_AUS** ist spezifisch für die unidirektionale Umwandlung eines DWORD-Adapters in einen USINT-Adapter. Alternativ könnte man die IEC-61131-Funktion `F_DWORD_TO_USINT` direkt verwenden, müsste dann aber die Adapter-Kopplung selbst vornehmen.

- Andere Adapter-Konvertierungsbausteine (z. B. `AD_TO_BYTE` oder `AUS_TO_DWORD`) folgen dem gleichen Muster, arbeiten jedoch mit anderen Datentypen. Der vorliegende FB füllt die Lücke für die spezifische Kombination DWORD → USINT.

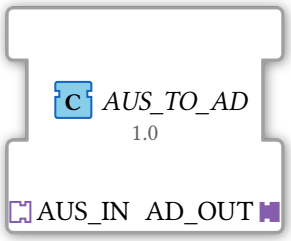
Fazit

Der **AD_TO_AUS** ist ein einfacher, aber nützlicher Composite-FB zur Typkonvertierung zwischen zwei unidirektionalen Adapter-Schnittstellen. Er reduziert den Implementierungsaufwand, indem er die Standardkonvertierung kapselt und eine saubere, ereignisgesteuerte Adapter-Kommunikation ermöglicht. Dank der klaren Struktur eignet er sich sowohl für den schnellen Entwurf als auch für den produktiven Einsatz in IEC-61499-basierten Steuerungen.

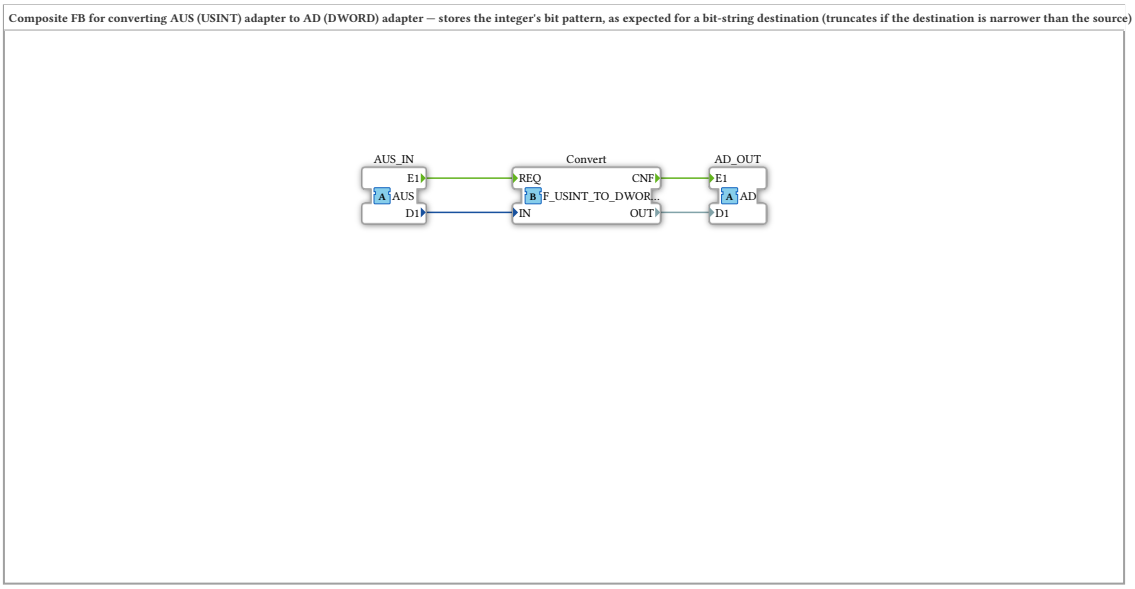
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUS_TO_AD



USINT Adapter Input – *AUS* — *AD* – DWORD Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **AUS_TO_AD** ist ein Composite-Baustein, der eine einfache Konvertierung von einem USINT-Adapter (Typ AUS) zu einem DWORD-Adapter (Typ AD) durchführt. Er kapselt die Umwandlung eines vorzeichenlosen 8-Bit-Wertes (USINT) in einen 32-Bit-DWORD-Wert und macht diese über standardisierte Adapter-Schnittstellen zugänglich.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung

E1	Ereignis (vom Socket AUS_IN)	Startet die Konvertierung des anliegenden USINT-Wertes. Der Eingang wird über den Socket AUS_IN bereitgestellt.

Ereignis-Ausgänge

Name	Typ	Beschreibung
E1	Ereignis (zum Plug AD_OUT)	Signalisiert die erfolgreiche Konvertierung und das Vorliegen des DWORD-Wertes am Ausgang. Der Ausgang wird über den Plug AD_OUT bereitgestellt.

Daten-Eingänge

Name	Datentyp	Beschreibung
D1	USINT (vom Socket AUS_IN)	Der zu konvertierende 8-Bit-Wert im Bereich 0…255.

Daten-Ausgänge

Name	Datentyp	Beschreibung
D1	DWORD (zum Plug AD_OUT)	Der konvertierte 32-Bit-Wert (direkte Typumsetzung des USINT in DWORD).

Adapter

Rolle	Name	Typ	Richtung	Beschreibung
Socket	AUS_IN	adapter::types::unidirectional::AUS	Eingang	Nimmt den USINT-Wert sowie das zugehörige Ereignis entgegen.
Plug	AD_OUT	adapter::types::unidirectional::AD	Ausgang	Stellt den DWORD-Wert sowie das Bestätigungsereignis bereit.

Funktionsweise

Der Baustein implementiert eine einfache, durchgängige Datenkonvertierung:

1. Ein eingehendes Ereignis am Adapter-Eingang AUS_IN.E1 triggert die interne Konvertierung.
2. Der angeschlossene USINT-Wert (AUS_IN.D1) wird über den standardisierten IEC-61131-Funktionsbaustein F_USINT_TO_DWORD in einen DWORD-Wert umgewandelt.
3. Nach erfolgreicher Konvertierung wird das Ausgangsereignis AD_OUT.E1 gesendet und der DWORD-Wert unter AD_OUT.D1 bereitgestellt.

Der Baustein arbeit ohne eigene Zustandslogik – er führt die Konvertierung pro Ereignis sofort aus (transparentes Verhalten).

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Composite-Bauweise:** Der Baustein besteht ausschließlich aus einem internen Netzwerk mit einem einzelnen Funktionsbaustein (F_USINT_TO_DWORD) sowie direkten Ereignis- und Datenverbindungen.
- **Wiederverwendung:** Die Konvertierung nutzt die IEC-61131-3-Standardfunktion F_USINT_TO_DWORD, die in der Bibliothek iec61131::conversion enthalten ist.
- **Adapter-Schnittstellen:** Die Schnittstellen sind als unidirektionale Adapter (adapter::types::unidirectional::AUS bzw. AD) realisiert. Jeder Adapter besteht aus genau einem Ereignis- und einem Datenkanal.
- **Kein Zustandsautomat:** Da es sich um einen reinen Composite-Baustein ohne eigenen ECC handelt, werden keine Zustände verwaltet.

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Das Verhalten ist rein datenflussgesteuert: Bei jedem eingehenden Ereignis wird genau eine Konvertierung durchgeführt und das Ereignis unverzögert weitergeleitet.

Anwendungsszenarien

- **Schnittstellenanpassung:** Wenn ein USINT-Wert (z. B. ein Byte-Sensor) in einem System genutzt werden muss, das intern nur DWORD-Werte verarbeitet.
- **Adapter-Brückenbildung:** Einsatz in modularen Steuerungsarchitekturen, in denen Adapter-Typen nicht direkt kompatibel sind und eine Konvertierung zwischen AUS- und AD-Adaptern erforderlich ist.
- **Prototyp-/Testumgebungen:** Schnelle Verbindung zweier Komponenten mit unterschiedlichen Adapter-Definitionen ohne eigene Konvertierungslogik.

Vergleich mit ähnlichen Bausteinen

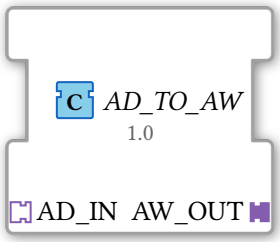
- **F_USINT_TO_DWORD:** Der zugrunde liegende Konvertierungsbaustein ist ein reiner Datenkonverter ohne Adapter-Schnittstelle. Er setzt direkte Daten- und Ereignisverbindungen voraus. `AUS_TO_AD` hingegen kapselt diese Funktion und bietet eine standardisierte Adapter-Schnittstelle an.
- **Andere Adapter-Konverter (z. B. BYTE_TO_DWORD, WORD_TO_DWORD):** Ähnliche Bausteine existieren für andere Datentypen, setzen aber meist andere Adapter-Typen voraus. `AUS_TO_AD` ist speziell für die Kombination der Adapter `AUS` (USINT) und `AD` (DWORD) optimiert.

Fazit

Der Funktionsbaustein `AUS_TO_AD` bietet eine einfache und robuste Möglichkeit, einen unidirektionalen USINT-Adapter in einen DWORD-Adapter umzuwandeln. Durch die Verwendung des standardisierten Konvertierungsbausteins und der Adapter-Schnittstellen lässt er sich nahtlos in IEC-61499-basierten Automatisierungssystemen einsetzen. Er reduziert den manuellen Aufwand für Typkonvertierungen und sorgt für eine saubere, wiederverwendbare Schnittstellenanpassung.

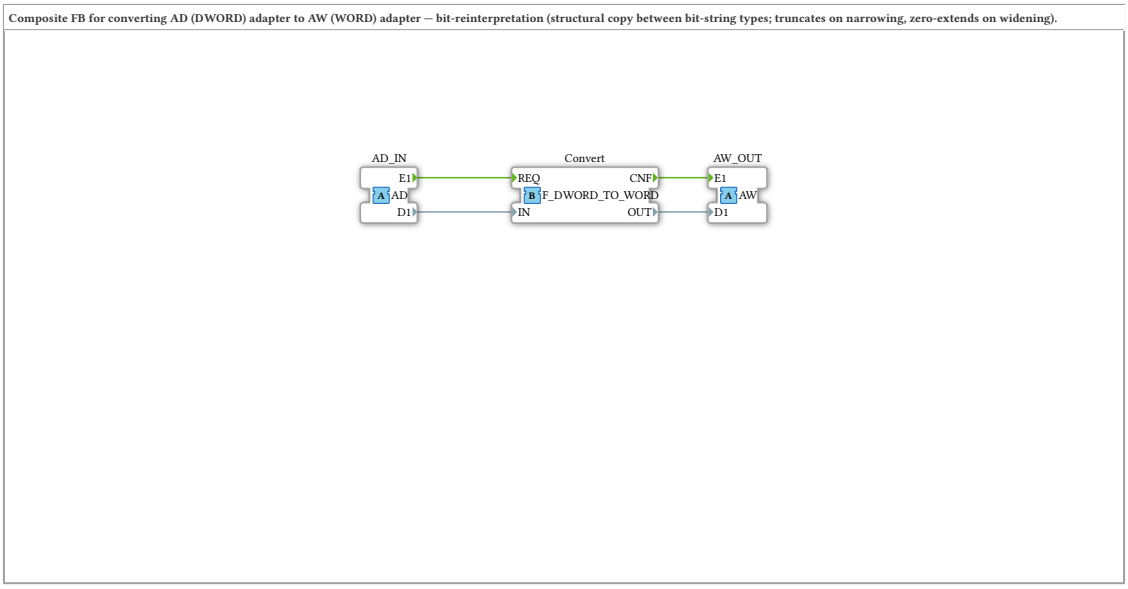
AD_AW

AD_TO_AW



DWORD Adapter Input – *AD*

AW – WORD Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AD_TO_AW** ist ein zusammengesetzter (Composite) FB, der einen DWORD-Adapter (Typ `AD`) in einen WORD-Adapter (Typ `AW`) umwandelt. Er kapselt die Konvertierung eines DWORD-Wertes in einen WORD-Wert und stellt die entsprechende Ereignis- und Datenweiterleitung bereit. Die Implementierung basiert auf einem internen Aufruf des Standard-Konvertierungsbausteins `F_DWORD_TO_WORD` aus der IEC 61131-Bibliothek.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ein-/Ausgangs-Events oder Datenports. Alle Signale werden über die angeschlossenen Adapter (Socket und Plug) geführt.

Ereignis-Eingänge

- **AD_IN.E1** (über Socket `AD_IN`): Dieses Ereignis triggert die Umwandlung des anliegenden DWORD-Wertes.

Ereignis-Ausgänge

- **AW_OUT.E1** (über Plug `AW_OUT`): Nach erfolgreicher Konvertierung wird dieses Ereignis gesendet.

Daten-Eingänge

- **AD_IN.D1** (über Socket `AD_IN`): Eingangsdaten vom Typ `DWORD`, die in einen WORD-Wert umgewandelt werden.

Daten-Ausgänge

- **AW_OUT.D1** (über Plug `AW_OUT`): Ausgangsdaten vom Typ `WORD`, die das Ergebnis der Konvertierung enthalten.

Adapter

Rolle	Name	Typ	Richtung	Kommentar
Socket	AD_IN	`adapter::types::unidirectional::AD`	Eingang (DWORD)	Liefert DWORD-Daten und Ereignis.
Plug	AW_OUT	`adapter::types::unidirectional::AW`	Ausgang (WORD)	Stellt WORD-Daten und Ereignis bereit.

Beide Adapter sind vom Typ *unidirectional*, d. h. die Kommunikation erfolgt in eine Richtung.

Funktionsweise

1. Sobald am Socket `AD_IN` das Ereignis `E1` eintrifft, wird der Wert von `AD_IN.D1` (DWORD) an den internen Konvertierungsbaustein `F_DWORD_TO_WORD` übergeben.
2. Der interne Baustein führt die Konvertierung durch und legt das Ergebnis (WORD) an seinem Ausgang `OUT` an.
3. Der konvertierte Wert wird sofort an den Plug `AW_OUT.D1` weitergeleitet.
4. Gleichzeitig wird das Ereignis `E1` am Plug `AW_OUT` ausgelöst, um die nachgeschaltete Logik zu informieren.

Die gesamte Verarbeitung erfolgt ereignisgesteuert und synchron.

Technische Besonderheiten

- **Bit-Reinterpretation** (strukturelle Kopie zwischen Bit-String-Typen; schneidet bei Verengung ab, erweitert mit Nullen bei Erweiterung).
- **Kapselung**: Der FB verwendet die standardisierte IEC-61131-Funktion `F_DWORD_TO_WORD`, die eine saubere und portierbare Konvertierung gewährleistet.
- **Adapter-basiert**: Ein- und Ausgänge sind ausschließlich über Adapter realisiert – es gibt keine separaten Event-/Datenports auf Bausteinebene.
- **Unidirektionalität**: Die verwendeten Adapter unterstützen nur eine Datenflussrichtung, was die Handhabung in einfachen Konvertierungsketten erleichtert.
- **Composite-Design**: Der FB besitzt kein eigenes ECC (Execution Control Chart); die Steuerung erfolgt vollständig durch den internen Baustein.

Zustandsübersicht

Als zusammengesetzter Baustein ohne eigenes ECC besitzt **AD_TO_AW** keine internen Zustände. Die Ausführung ist rein ereignisgetrieben und erfolgt atomar bei jedem eingehenden Ereignis am Socket.

Anwendungsszenarien

- **Adapter-Anpassung**: Wenn ein Steuerungssystem Daten über einen DWORD-Adapter bereitstellt, aber ein angeschlossenes Modul oder eine Komponente einen WORD-Adapter erwartet.
- **Datenreduktion**: Gezieltes Abschneiden eines DWORD-Wertes auf die unteren 16 Bit (WORD) für nachfolgende Verarbeitungsschritte.
- **Brückenbildung**: Einsatz in heterogenen Systemlandschaften, in denen unterschiedliche Adapter-Typen aufeinander treffen.

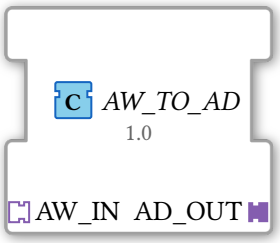
Vergleich mit ähnlichen Bausteinen

- **WORD_TO_DWORD**: Führt die entgegengesetzte Konvertierung (WORD → DWORD) durch und erweitert den Wert (in der Regel durch Nullen).
- **AD_TO_AD (z. B. AD_TO_AD mit anderen Typen)**: Andere Composite-Bausteine können ähnliche Adapter-Umwandlungen für weitere Datentypen (z. B. BYTE, INT, REAL) kapseln.
- **Direkte Nutzung von `F_DWORD_TO_WORD`**: Statt eines Composite-Bausteins könnte die Konvertierungsfunktion direkt in ein FB-Netzwerk eingebunden werden – der Composite bietet jedoch eine wiederverwendbare, gekapselte Lösung.

Fazit

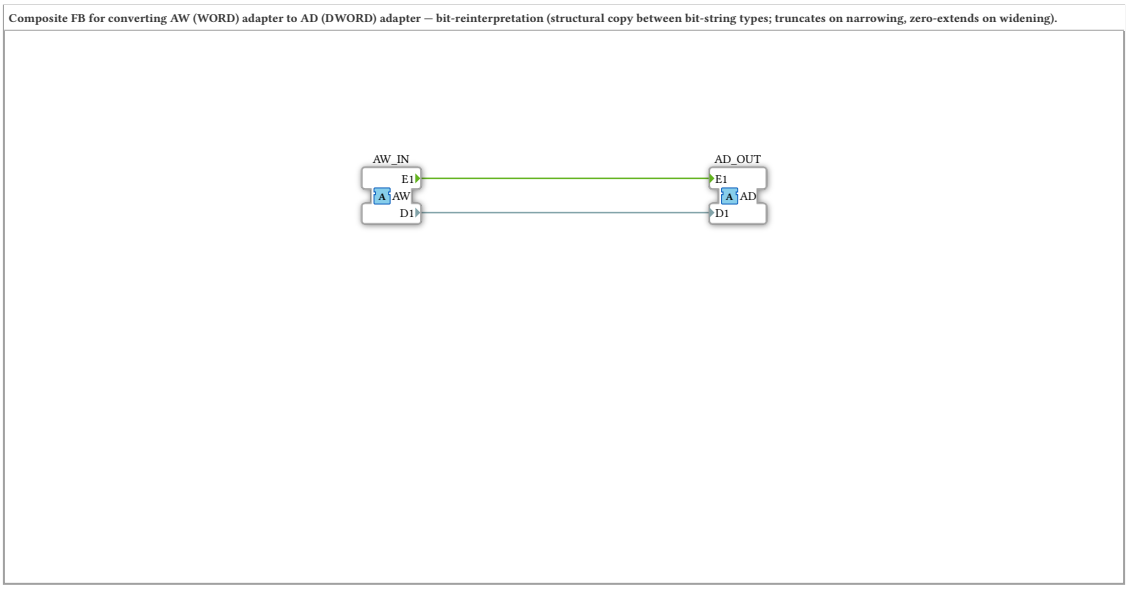
Der **AD_TO_AW**-Funktionsblock ist ein einfacher, aber nützlicher Composite-Baustein zur Umwandlung eines DWORD-Adapters in einen WORD-Adapter. Er nutzt eine standardisierte Konvertierungsfunktion, ist leicht verständlich und kann in vielen Adapter-basierten Automatisierungssystemen eingesetzt werden. Durch die Kapselung in einem eigenen FB wird die Netzwerktopologie übersichtlicher und die Wiederverwendbarkeit erhöht.

AW_TO_AD



WORD Adapter Input – *AW*

AD – DWORD Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AW_TO_AD** ist ein zusammengesetzter Baustein (Composite FB), der einen unidirektionalen WORD-Adapter (AW) in einen unidirektionalen DWORD-Adapter (AD) umwandelt. Er dient als Schnittstellenkonverter und ermöglicht die Anbindung von Komponenten, die unterschiedliche Adaptertypen verwenden. Der FB leitet Ereignisse und Daten direkt vom Eingangs- zum Ausgangsadapter weiter, wobei implizit eine Typkonvertierung von 16-Bit-WORD auf 32-Bit-DWORD erfolgt.

Schnittstellenstruktur

Ereignis-Eingänge

- **AW_IN.E1** – Ereigniseingang über den Socket `AW_IN`. Ein eingehendes Ereignis wird direkt an den Ausgang weitergeleitet.

Ereignis-Ausgänge

- **AD_OUT.E1** – Ereignisausgang über den Plug `AD_OUT`. Das vom Eingang durchgereichte Ereignis wird hier ausgegeben.

Daten-Eingänge

- **AW_IN.D1** – Dateneingang (Typ: WORD, 16 Bit) über den Socket `AW_IN`.

Daten-Ausgänge

- **AD_OUT.D1** – Datenausgang (Typ: DWORD, 32 Bit) über den Plug `AD_OUT`. Die empfangenen 16 Bit werden auf die unteren 16 Bit des DWORD abgebildet; die oberen 16 Bit werden mit Nullen aufgefüllt (implizite Konvertierung).

Adapter

Name	Typ	Richtung	Beschreibung
`AW_IN`	`adapter::types::unidirectional::AW`	Socket (Eingang)	WORD-Adapter, der Ereignis- und Dateneingänge bereitstellt
`AD_OUT`	`adapter::types::unidirectional::AD`	Plug (Ausgang)	DWORD-Adapter, der Ereignis- und Datenausgänge bereitstellt

Funktionsweise

Der Baustein ist als reine Weiterleitung konzipiert. Im internen FBNetzwerk sind die Ereignis- und Datenleitungen zwischen `AW_IN` und `AD_OUT` direkt verbunden:

- Ein Ereignis an `AW_IN.E1` wird sofort an `AD_OUT.E1` durchgereicht.
- Der 16-Bit-Datenwert von `AW_IN.D1` wird auf `AD_OUT.D1` übertragen. Dabei erfolgt eine implizite Erweiterung auf 32 Bit: Der WORD-Wert wird in die niederwertigen 16 Bit des DWORD kopiert, die höherwertigen Bit werden auf `0` gesetzt.

Es findet keine eigene Verarbeitung oder Zwischenspeicherung statt – der FB fungiert als einfacher Adapter-Konverter.

Technische Besonderheiten

- **Bit-Reinterpretation** (strukturelle Kopie zwischen Bit-String-Typen; schneidet bei Verengung ab, erweitert mit Nullen bei Erweiterung).
- **Composite FB ohne eigene Zustandslogik:** Der Baustein besitzt keinen ECC (Execution Control Chart) und arbeitet zustandslos. Alle Abläufe werden durch die direkten Verbindungen im FBNetzwerk bestimmt.
- **Unidirektionale Richtung:** Die Konvertierung erfolgt nur in eine Richtung (AW ⟶ AD). Für die umgekehrte Richtung wäre ein separater Baustein erforderlich.
- **Implizite Typkonvertierung:** Die Daten werden ohne explizite Konvertierungslogik von WORD nach DWORD gecastet. Falls eine andere Abbildung (z. B. Vorzeichenbehandlung) gewünscht wird, muss ein eigener Funktionsbaustein verwendet werden.
- **Paket & Lizenz:** Der Baustein ist im Paket `adapter::conversion::unidirectional` unter der Eclipse Public License 2.0 verfügbar.

Zustandsübersicht

Da der FB als Composite ohne ECC realisiert ist, existiert kein interner Zustandsautomat. Die gesamte Funktionalität erschöpft sich in der direkten Durchleitung von Ereignissen und Daten. Der FB ist daher **zustandslos** und reagiert sofort auf jedes eingehende Ereignis.

Anwendungsszenarien

- **Schnittstellenmigration:** Austausch einer Komponente, die einen WORD-Adapter erwartet, gegen eine Komponente mit DWORD-Adapter.
- **Systemintegration:** Anbindung eines bestehenden WORD-basierten Moduls an eine DWORD-basierte Steuerung.
- **Protokollanpassung:** Wenn ein Subsystem Daten im WORD-Format liefert, das Ziel jedoch einen DWORD-Eingang benötigt.
- **Test und Simulation:** Einfache Konvertierung für Testzwecke, ohne komplexe Logik implementieren zu müssen.

Vergleich mit ähnlichen Bausteinen

Baustein	Beschreibung
`WORD_TO_DWORD`	Reine Datenkonvertierung ohne Adapter-Schnittstelle
`AW_TO_AD`	Konvertiert komplette unidirektionale Adapter (Ereignis + Daten)
`AD_TO_AW` (nicht vorhanden)	Theoretischer Baustein für die umgekehrte Richtung

Im Gegensatz zu einem reinen Datenkonverter berücksichtigt `AW_TO_AD` auch die Ereignisweiterleitung und ist direkt in Adapter-basierte Architekturen integrierbar.

Fazit

`AW_TO_AD` ist ein einfacher, aber nützlicher Composite-Funktionsbaustein zur Adapter-Konvertierung von WORD nach DWORD. Er ermöglicht eine schnelle und saubere Anpassung von Schnittstellen in 4diac-Projekten, ohne dass eigene Konvertierungslogik geschrieben werden muss. Aufgrund seiner Zustandslosigkeit und der direkten Durchleitung eignet er sich besonders für unidirektionale Verbindungen in bestehenden Automatisierungslösungen.

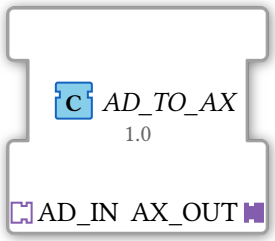
Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

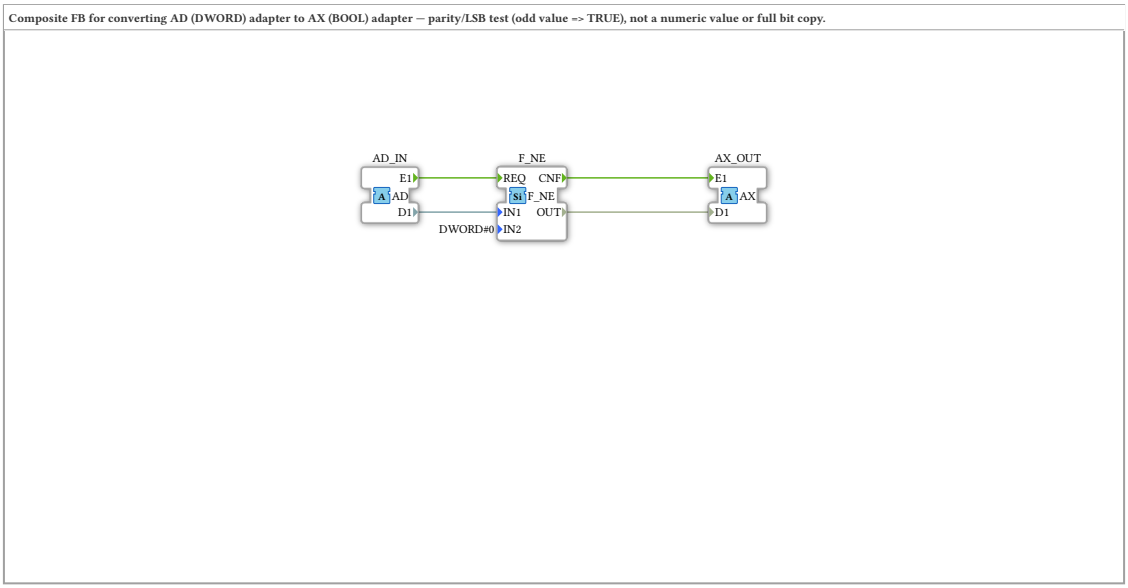
AD_AX

AD_TO_AX

DWORD Adapter Input – *AD*



AX – BOOL Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `AD_TO_AX` dient als Composite-FB zur Umwandlung eines AD-Adapters (Datenwort – DWORD) in einen AX-Adapter (Boolescher Wert – BOOL). Er kapselt einen Vergleichsmechanismus, der prüft, ob der anliegende DWORD-Wert ungleich Null ist, und das Ergebnis als BOOL an den Ausgangsadapter weiterleitet. Der Baustein vereinfacht die Adapterkommunikation in Steuerungssystemen, indem er die Signaltypen zwischen verschiedenen Komponenten umsetzt.

Schnittstellenstruktur

Ereignis-Eingänge

Keine eigenständigen Ereignis-Eingänge. Das Ereignis wird über den Socket-Adapter `AD_IN` bereitgestellt (siehe Abschnitt **Adapter**).

Ereignis-Ausgänge

Keine eigenständigen Ereignis-Ausgänge. Das Ergebnisereignis wird über den Plug-Adapter `AX_OUT` bereitgestellt (siehe Abschnitt **Adapter**).

Daten-Eingänge

Keine eigenständigen Daten-Eingänge. Die Eingangsdaten werden über den Socket-Adapter `AD_IN` bereitgestellt (siehe Abschnitt **Adapter**).

Daten-Ausgänge

Keine eigenständigen Daten-Ausgänge. Die Ausgangsdaten werden über den Plug-Adapter `AX_OUT` bereitgestellt (siehe Abschnitt **Adapter**).

Adapter

Adapter	Richtung	Typ	Kommentar
`AD_IN`	Socket	`adapter::types::unidirectional::AD`	DWORD-Eingangsadapter – liefert Ereignis und DWORD-Daten.
`AX_OUT`	Plug	`adapter::types::unidirectional::AX`	BOOL-Ausgangsadapter – gibt Ereignis und Boolesches Ergebnis aus.

Die Adapter folgen dem unidirektionalen Muster und stellen jeweils einen Ereignisausgang (`E1`) und einen Datenausgang (`D1`) zur Verfügung:

- **AD_IN.E1** ⟶ initiiert die Verarbeitung.
- **AD_IN.D1** ⟶ enthält den DWORD-Quellwert.
- **AX_OUT.E1** ⟶ signalisiert das Ende der Verarbeitung.
- **AX_OUT.D1** ⟶ enthält den Booleschen Ergebniswert.

Funktionsweise

Der FB `AD_TO_AX` ist als Composite FB realisiert. Er enthält eine Instanz des Vergleichsbausteins `F_NE` („ungleich“) aus der IEC 61131-Bibliothek.

1. Ein eingehendes Ereignis auf `AD_IN.E1` triggert den Eingang `F_NE.REQ`.
2. Der Baustein `F_NE` vergleicht den Wert von `AD_IN.D1` mit dem konstanten Wert `DWORD#0`.
3. Das Vergleichsergebnis (BOOL) wird am Ausgang `F_NE.OUT` bereitgestellt.
4. Über eine Datenverbindung gelangt dieses Ergebnis an `AX_OUT.D1`.
5. Gleichzeitig signalisiert `F_NE.CNF` das Ende der Berechnung und triggert `AX_OUT.E1`.

Somit wird aus einem DWORD-Wert ein BOOL-Wert abgeleitet:

- `DWORD = 0` ⟶ Ausgabe `FALSE`
- `DWORD ≠ 0` ⟶ Ausgabe `TRUE`

Technische Besonderheiten

- **Paritäts-/LSB-Test** (ungerader Wert ⟹ TRUE), kein Zahlenwert und keine vollständige Bit-Kopie.
- **Composite-FB**: Der Baustein besteht aus einem internen Netzwerk, das die eigentliche Logik abbildet – dies erleichtert die Wiederverwendung und Kapselung.
- **Verwendung von `F_NE`**: Der eingesetzte Vergleichsbaustein aus der IEC 61131-Bibliothek ist standardisiert und plattformunabhängig.
- **Adapterbasierte Kommunikation**: Statt einzelner Ein-/Ausgänge werden Adapter verwendet, was die strukturierte Verbindung zu anderen Adapter-kompatiblen Bausteinen ermöglicht.
- **Unidirektionale Adapter**: Sowohl Eingangs- als auch Ausgangsadapter sind unidirektional, d.h. sie transportieren Daten und Ereignisse nur in eine Richtung – dies vermeidet Rückkopplungen und vereinfacht das Design.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, da er ausschließlich aus einer logischen Verknüpfung besteht. Der interne Ablauf ist ereignisgesteuert:

- **Ruhezustand**: Warten auf ein Ereignis am Eingang `AD_IN.E1`.

- **Ausführung**: Nach dem Ereignis erfolgt die Berechnung durch `F_NE` und die Weiterleitung des Ergebnisses – dies geschieht in einem Schritt ohne Zwischenzustände.

Anwendungsszenarien

- **Adapterkonvertierung**: Ein System arbeitet mit DWORD-Adaptern, ein angeschlossenes Modul erwartet jedoch BOOL-Signale. Der `AD_TO_AX`-Baustein übernimmt die Umwandlung.
- **Schwellwertabfrage**: Prüfung, ob ein DWORD-Wert von Null abweicht (z.B. Aktivsignal, Zählerstand > 0).
- **Signalaufbereitung**: Vereinfacht die Integration von Sensoren oder Aktoren, deren Ausgang als DWORD codiert ist, in boolesche Steuerungslogiken.

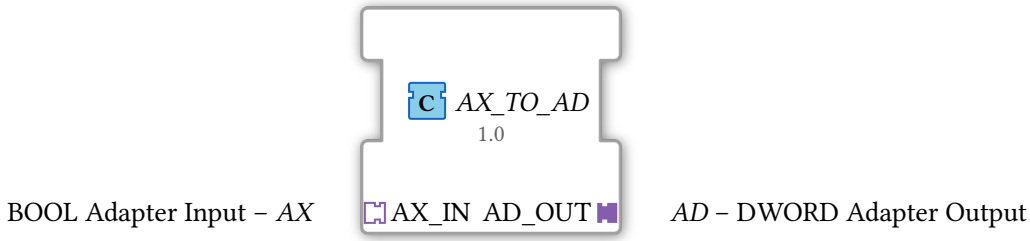
Vergleich mit ähnlichen Bausteinen

- **Direkter `DWORD_TO_BOOL`-Konverter**: Ein solcher FB würde typischerweise nur die Datenkonvertierung durchführen, aber keine Adapterstruktur und kein Ereignis-Handling bieten. `AD_TO_AX` hingegen kapselt sowohl Ereignis- als auch Datenpfade in Adaptern.
- **Allgemeine Adapterkonverter**: Es gibt ähnliche Bausteine für andere Datentypen (z.B. `AD_TO_AX` für WORD, INT etc.). Der vorliegende FB ist speziell auf die Umwandlung DWORD ⟶ BOOL ausgelegt.
- **Composite vs. Basic FB**: Als Composite-FB ist der Baustein leicht anpassbar, da das interne Netzwerk bei Bedarf modifiziert werden kann.

Fazit

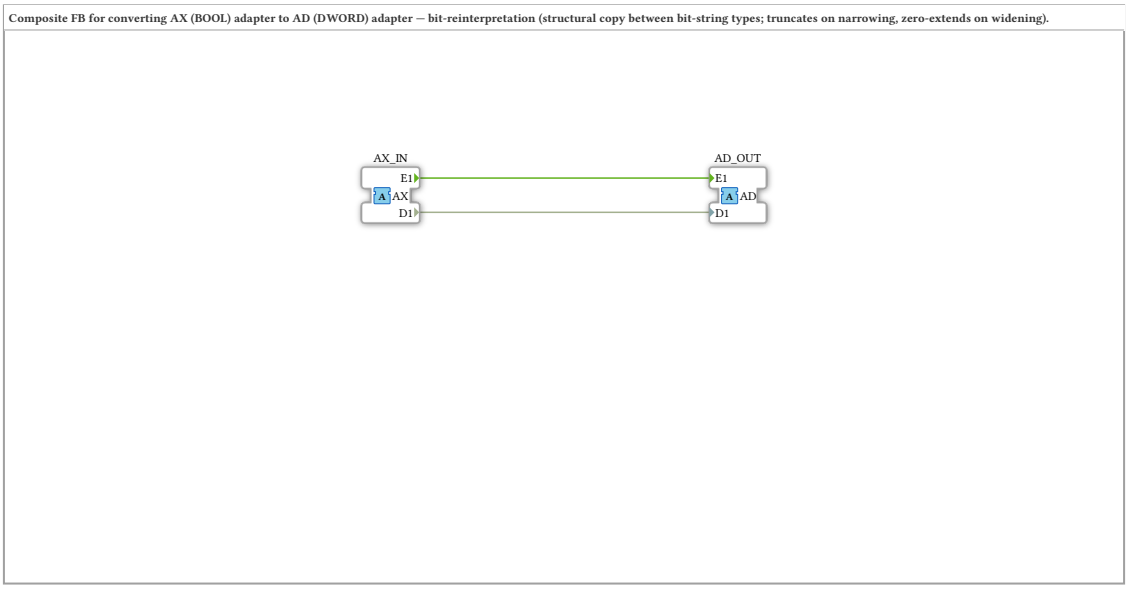
Der Funktionsblock `AD_TO_AX` bietet eine elegante und standardisierte Möglichkeit, einen DWORD-Adaptersignale in einen BOOL-Adapter umzuwandeln. Durch die Nutzung des etablierten Vergleichsbausteins `F_NE` ist die Logik robust und nachvollziehbar. Die Adapterstruktur erleichtert die Integration in modulare Automatisierungssysteme und fördert die Wiederverwendbarkeit. Der FB ist besonders geeignet für Szenarien, in denen eine ereignisgesteuerte Umwandlung von digitalen Werten in boolesche Zustände erforderlich ist.

AX_TO_AD



BOOL Adapter Input – *AX*

AD – DWORD Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Composite FB `AX_TO_AD` dient der Konvertierung eines Bool-Adapters (AX) in einen DWORD-Adapter (AD). Er leitet Ereignis- und Datensignale vom Eingangsadapter direkt an den Ausgangsadapter weiter und wandelt implizit den Datentyp von BOOL nach DWORD um.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine direkten Ereignis-Eingänge. Die Ereignissteuerung erfolgt über den Socket `AX_IN`.

Ereignis-Ausgänge

Es existieren keine direkten Ereignis-Ausgänge. Die Ereignisweitergabe erfolgt über den Plug `AD_OUT`.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenaufnahme erfolgt über den Socket `AX_IN`.

Daten-Ausgänge

Es gibt keine direkten Daten-Ausgänge. Die Datenausgabe wird über den Plug `AD_OUT` realisiert.

Adapter

- **AX_IN** (Socket): Adapter vom Typ `adapter::types::unidirectional::AX`. Stellt ein BOOL-Signal (D1) und ein Trigger-Ereignis (E1) bereit.
- **AD_OUT** (Plug): Adapter vom Typ `adapter::types::unidirectional::AD`. Erwartet ein DWORD-Signal (D1) und ein Trigger-Ereignis (E1).

Funktionsweise

Der FB verknüpft den Eingangsadapter `AX_IN` direkt mit dem Ausgangsadapter `AD_OUT`:

- **Ereignispfad**: Das Ereignis von `AX_IN.E1` wird unverändert an `AD_OUT.E1` weitergeleitet.
- **Datenpfad**: Der boolesche Wert von `AX_IN.D1` wird auf `AD_OUT.D1` übertragen. Dabei erfolgt eine implizite Typkonvertierung: `TRUE` wird zu `1` (DWORD), `FALSE` zu `0` (DWORD).

Der Baustein selbst enthält keine eigene Logik oder Zustandsautomaten – er fungiert als reiner Durchreiche-Konverter zwischen den beiden Adaptertypen.

Technische Besonderheiten

- **Bit-Reinterpretation** (strukturelle Kopie zwischen Bit-String-Typen; schneidet bei Verengung ab, erweitert mit Nullen bei Erweiterung).
- **Composite FB**: Die Implementierung besteht ausschließlich aus einer Verdrahtung im FBNetzwerk, ohne zusätzliche Algorithmen.
- **Implizite Datentypkonvertierung**: Der BOOL-Wert wird automatisch in einen DWORD-Wert umgewandelt (Bits 0..31: 0 oder 1).
- **Unidirektionale Adapter**: Sowohl Ein- als auch Ausgang verwenden unidirektionale Adapter, die nur eine Daten- und eine Ereignisverbindung unterstützen.

Zustandsübersicht

Der `AX_TO_AD` FB besitzt keinen eigenen internen Zustand. Sein Verhalten ist rein reaktiv: Ein eintreffendes Ereignis am Eingang wird sofort an den Ausgang weitergegeben, optional begleitet von der aktualisierten Datensignatur.

Anwendungsszenarien

- Integration eines digitalen Sensors (z.B. Schalter, Lichtschranke) in ein System, das einen DWORD-Adapter für die Datenaufnahme erwartet.
- Schrittweise Migration einer Anwendung von BOOL-basierten zu DWORD-basierten Adapter-Schnittstellen.
- Prototypische Verbindung zwischen unterschiedlichen Bibliotheken oder Komponenten mit inkompatiblen Adaptertypen.

Vergleich mit ähnlichen Bausteinen

- **BOOL_TO_DWORD** (Funktionsbaustein): Führt die reine Datentypkonvertierung durch, jedoch ohne Adapter-Kapselung.
- **AX_TO_AD** spezialisiert auf die Adapter-Schnittstellen und eignet sich daher besonders für den Austausch zwischen verschiedenen Kommunikationsmustern in 4diac.
- Ähnliche Bausteine existieren für andere Typkombinationen, z. B. `INT_TO_REAL_AD` oder `WORD_TO_BOOL_AD`.

Fazit

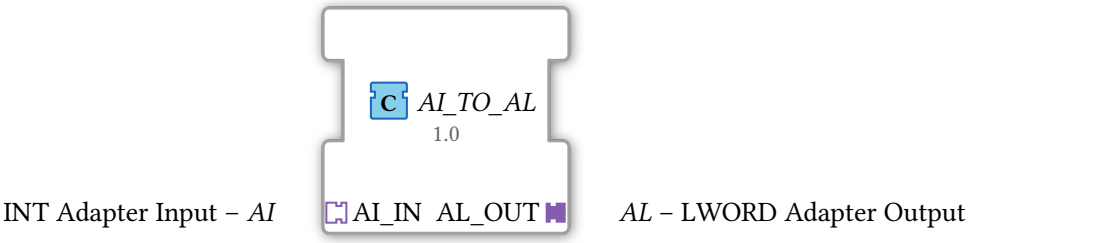
Der `AX_TO_AD` FB bietet eine einfache und saubere Möglichkeit, einen BOOL-Adapter in einen DWORD-Adapter umzuwandeln, ohne dass der Anwender selbst Konvertierungslogik schreiben muss. Er eignet sich ideal für die Kopplung unterschiedlicher Adapter-Welten in 4diac-Projekten.

Passende Themen-Unterseiten auf ms-muc-docs.de

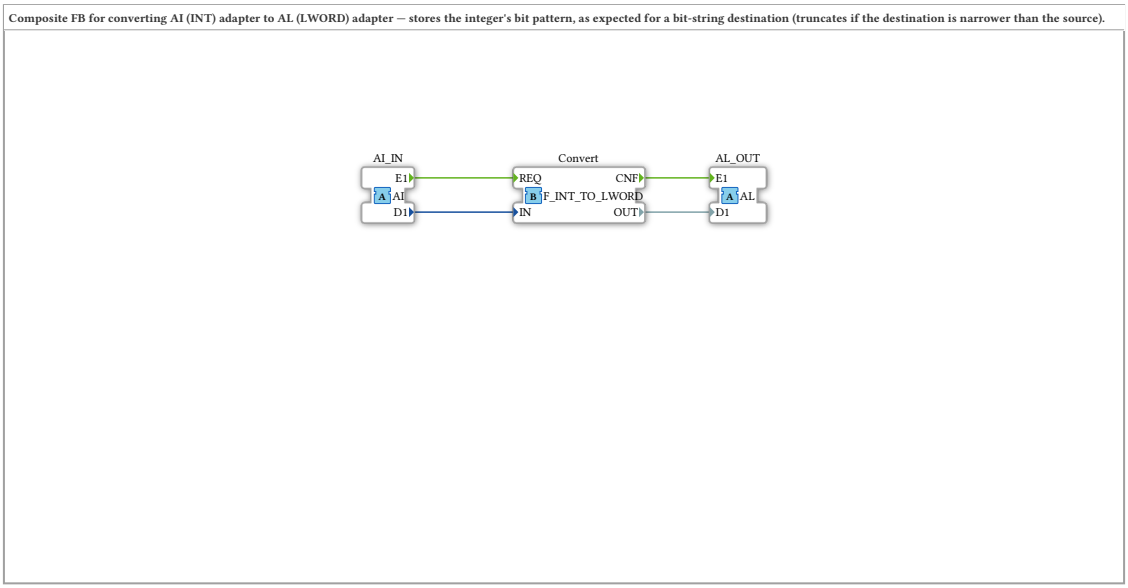
- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AI_AL

AI_TO_AL



INT Adapter Input – *AI* — *AL* – LWORD Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Composite-Funktionsbaustein **AI_TO_AL** dient zur Umwandlung eines INT-Werts, der über den unidirektionalen AI-Adapter (Typ `adapter::types::unidirectional::AI`) bereitgestellt wird, in einen LWORD-Wert, der über den unidirektionalen AL-Adapter (Typ `adapter::types::unidirectional::AL`) ausgegeben wird. Der Baustein kapselt die Konvertierungslogik und ermöglicht so eine saubere Trennung zwischen INT- und LWORD-basierten Adapter-Schnittstellen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine eigenständigen Ereignis-Eingänge am Baustein. Die Ereignissteuerung erfolgt ausschließlich über den integrierten Socket `AI_IN`, der das Ereignis `E1` als Trigger für die Konvertierung bereitstellt.

Ereignis-Ausgänge

Keine eigenständigen Ereignis-Ausgänge am Baustein. Das Ergebnisereignis der internen Konvertierung wird über den Plug `AL_OUT` als Ereignis `E1` ausgegeben.

Daten-Eingänge

Keine eigenständigen Daten-Eingänge am Baustein. Der zu konvertierende INT-Wert wird über den Socket `AI_IN` (Datenpunkt `D1`) in den Baustein eingelesen.

Daten-Ausgänge

Keine eigenständigen Daten-Ausgänge am Baustein. Der konvertierte LWORD-Wert wird über den Plug `AL_OUT` (Datenpunkt `D1`) ausgegeben.

Adapter

Bezeichnung	Typ	Richtung	Kommentar
`AI_IN`	`adapter::types::unidirectional::AI`	Socket (Input)	INT-Adapter-Eingang
`AL_OUT`	`adapter::types::unidirectional::AL`	Plug (Output)	LWORD-Adapter-Ausgang

Funktionsweise

1. Ein eingehendes Ereignis `E1` am Adapter `AI_IN` aktiviert die Datenkonvertierung.
2. Der interne Funktionsbaustein `F_INT_TO_LWORD` (aus der Bibliothek `iec61131::conversion`) wird über dessen Ereigniseingang `REQ` getriggert und wandelt den an `IN` anliegenden INT-Wert in einen LWORD-Wert um.
3. Nach Abschluss der Konvertierung signalisiert der Baustein dies über den Ereignisausgang `CNF`.
4. Dieses Ereignis wird an den Adapter `AL_OUT` als Ereignis `E1` weitergeleitet und gibt gleichzeitig den konvertierten LWORD-Wert über `D1` aus.

Die Datenpfade lauten:

- `AI_IN.D1` ⟶ `Convert.IN`
- `Convert.OUT` ⟶ `AL_OUT.D1`

Die Ereignispfade lauten:

- `AI_IN.E1` ⟶ `Convert.REQ`
- `Convert.CNF` ⟶ `AL_OUT.E1`

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Composite-Baustein:** Der `AI_TO_AL` ist als Composite implementiert und enthält keine eigene algorithmische Logik. Er delegiert die Konvertierung vollständig an den definierten Funktionsbaustein `F_INT_TO_LWORD`.
- **Standardkonvertierung:** Die verwendete Konvertierungsfunktion ist Teil der IEC 61131-Bibliothek und gewährleistet eine standardkonforme Typumwandlung von INT (16-Bit) nach LWORD (64-Bit).
- **Adapterbasiert:** Der Baustein arbeitet ausschließlich über die spezifizierten Adapter-Schnittstellen und kann daher nahtlos in eine adapterbasierte Architektur integriert werden.
- **Lizenz und Version:** Der Baustein steht unter der Eclipse Public License 2.0 (EPL-2.0) und wurde in Version 1.0 von der HR Agrartechnik GmbH veröffentlicht.
- **Erstellungsdatum:** 17. Februar 2026, Autor: Franz Höpfinger.

Zustandsübersicht

Als Composite-Baustein besitzt `AI_TO_AL` keinen eigenen internen Zustandsautomaten. Die gesamte Zustandslogik wird durch den integrierten Baustein `F_INT_TO_LWORD` gesteuert, der seinerseits eine einfache Anforderungs-Bestätigungs-Sequenz (REQ/CNF) ausführt. Der Baustein ist daher rein datenflussgetrieben und verfügt über keine persistenten Zustände.

Anwendungsszenarien

- **Buskopplung:** Wenn ein Sensor oder Aktor einen INT-Wert über einen AI-Adapter liefert, das nachgelagerte System aber einen LWORD-Wert über einen AL-Adapter erwartet (z. B. bei Anbindung an einen Feldbus mit 64-Bit-Datentypen).
- **Protokollanpassung:** In heterogenen Automatisierungssystemen, in denen verschiedene Adaptertypen (AI und AL) miteinander verbunden werden müssen.
- **Datenaufbereitung:** Vor der Weiterverarbeitung von INT-Daten in einer Steuerung, die intern mit LWORD arbeitet.

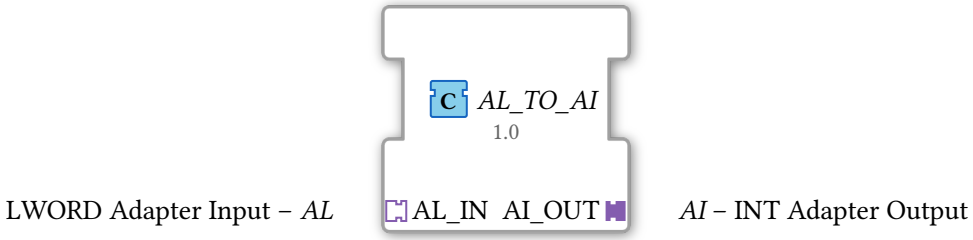
Vergleich mit ähnlichen Bausteinen

- **Andere Konvertierungsadapter:** Es existieren ähnliche Composite-Bausteine für andere Datentyp-Paare wie z. B. `WORD_TO_DWORD`, `BYTE_TO_WORD` oder `DINT_TO_LWORD`. Der `AI_TO_AL` ist auf die spezifische Kombination INT → LWORD ausgelegt.
- **Direkte Konvertierung:** Alternativ könnte die Konvertierung auch ohne Adapter-Kapselung direkt mit `F_INT_TO_LWORD` erfolgen. Der Vorteil des `AI_TO_AL` liegt in der standardisierten Adapter-Schnittstelle, die den Austausch und die Wiederverwendbarkeit in adapterbasierten Netzwerken erleichtert.

Fazit

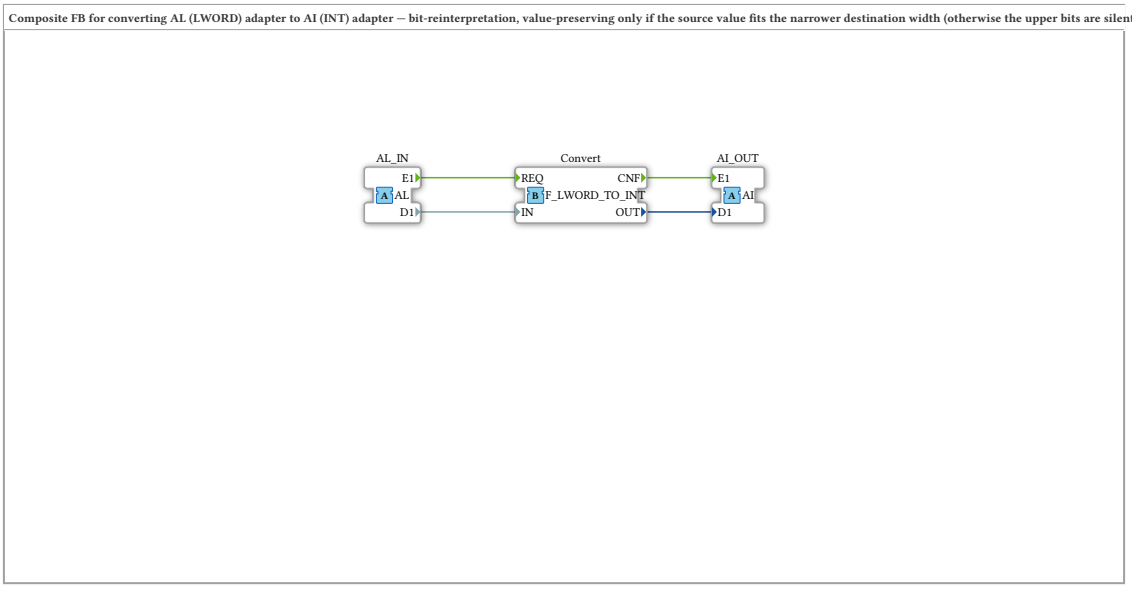
Der `AI_TO_AL`-Funktionsbaustein ist ein einfacher, aber wertvoller Adapter-Konverter, der die Lücke zwischen INT- und LWORD-basierten Adapter-Schnittstellen schließt. Durch die Kapselung der Konvertierungslogik in einem Composite-Baustein bleibt die Applikationslogik übersichtlich und die Wiederverwendbarkeit wird erhöht. Er eignet sich besonders für den Einsatz in modularen Automatisierungssystemen, die auf der unidirektionalen Adapter-Kommunikation basieren.

AL_TO_AI



LWORD Adapter Input – *AL*

AI – INT Adapter Output



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock AL_TO_AI ist ein Composite-Baustein, der einen unidirektionalen AL-Adapter (LWORD) in einen unidirektionalen AI-Adapter (INT) umwandelt. Er dient als Schnittstellenkonverter für adapterbasierte Kommunikation und ermöglicht eine einfache Integration zwischen Komponenten, die unterschiedliche Datentypen verwenden. Die Konvertierung erfolgt intern mittels des standardisierten IEC 61131-3-Funktionsbausteins F_LWORD_TO_INT.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine expliziten Ereignis-Eingänge. Die Ereignissteuerung erfolgt indirekt über den Socket-Adapter **AL_IN**, der das Ereignis **E1** bereitstellt.

Ereignis-Ausgänge

Der FB besitzt keine expliziten Ereignis-Ausgänge. Die Ereignissteuerung erfolgt indirekt über den Plug-Adapter **AI_OUT**, der das Ereignis **E1** bereitstellt.

Daten-Eingänge

Der FB besitzt keine expliziten Daten-Eingänge. Die aufzunehmenden Daten werden über den Socket-Adapter **AL_IN** als LWORD zur Verfügung gestellt.

Daten-Ausgänge

Der FB besitzt keine expliziten Daten-Ausgänge. Die konvertierten Daten werden über den Plug-Adapter **AI_OUT** als INT ausgegeben.

Adapter

Adapter	Richtung	Typ	Beschreibung
AL_IN	Socket (Eingang)	`adapter::types::unidirectional::AL`	Liefert ein LWORD (über Daten-Port D1) sowie ein Ereignis (E1).
AI_OUT	Plug (Ausgang)	`adapter::types::unidirectional::AI`	Stellt ein INT (über Daten-Port D1) sowie ein Ereignis (E1) bereit.

Funktionsweise

Der FB arbeitet als transparenter Konverter zwischen den beiden Adaptern. Sobald am Socket **AL_IN** das Ereignis **E1** eintrifft, wird der interne Funktionsbaustein **F_LWORD_TO_INT** mit dem übergebenen

LWORD-Wert (aus **AL_IN.D1**) getriggert. Der Konvertierungsbaustein wandelt den Wert in ein INT um und signalisiert die Fertigstellung über das Ereignis **CNF**. Dieses Ereignis wird direkt an den Plug **AI_OUT** weitergeleitet (E1), und das konvertierte INT (aus **F_LWORD_TO_INT.OUT**) wird auf **AI_OUT.D1** ausgegeben. Die gesamte Umwandlung erfolgt synchron innerhalb eines Ereignisdurchlaufs.

Technische Besonderheiten

- **Bit-Reinterpretation**, werterhaltend nur, wenn der Quellwert in die schmalere Zielbreite passt (sonst werden die oberen Bits stillschweigend verworfen).
- **Composite-Architektur**: Der FB verwendet intern einen standardisierten IEC 61131-3-Funktionsbaustein (`F_LWORD_TO_INT`), der plattformunabhängig implementiert werden kann.
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind vom Typ `unidirectional`, d. h. sie unterstützen nur eine Daten- und Ereignisrichtung (vom Socket zum Plug).
- **Keine Eigenlogik**: Der FB enthält keine Zustandslogik oder Sequenzen; er fungiert ausschließlich als Koppler und Konverter.
- **Typumwandlung**: Die Konvertierung von LWORD (64 Bit) nach INT (16 Bit) kann je nach Wertebereich zu Datenverlusten führen (Abschneiden höherwertiger Bits). Dies muss im Anwendungskontext berücksichtigt werden.

Zustandsübersicht

Der Composite-Baustein besitzt keine eigenen Zustände. Das Verhalten wird vollständig durch den internen **F_LWORD_TO_INT** bestimmt, der eine reine Funktion ohne Zustand ist. Die Ablaufsteuerung erfolgt ereignisgesteuert: Auf ein Eingangsereignis folgt unmittelbar ein Ausgangsereignis.

Anwendungsszenarien

- **Adapter-Brücke**: Verbindung einer Komponente, die LWORD über einen AL-Adapter bereitstellt, mit einer Komponente, die INT über einen AI-Adapter erwartet.
- **Systemintegration**: Einsatz in heterogenen Automatisierungssystemen, wenn unterschiedliche Adaptertypen aufeinandertreffen.
- **Protokollanpassung**: Datenkonvertierung in einer serviceorientierten Architektur (z. B. 4diac) zur Anpassung von Schnittstellen aus verschiedenen IEC 61499- oder IEC 61131-Bibliotheken.

Vergleich mit ähnlichen Bausteinen

- **AL_TO_AI** ist spezifisch für die Kombination unidirektionaler AL- und AI-Adapter. Andere Konverter wie `LWORD_TO_INT` arbeiten auf reiner Datenebene ohne Adapter-Kapselung.
- Im 4diac-Ökosystem gibt es weitere Adapter-Konverter (z. B. `AI_TO_AL`), die die umgekehrte Richtung abdecken. `AL_TO_AI` füllt die Lücke für die unidirektionale LWORD→INT-Umwandlung.
- Gegenüber einer direkten Verwendung von `F_LWORD_TO_INT` bietet der Composite-FB den Vorteil einer standardisierten Adapter-Schnittstelle, die eine einfache Plug-and-Play-Integration ermöglicht.

Fazit

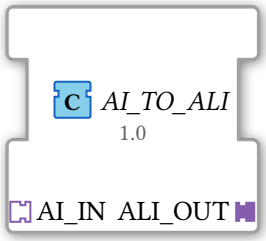
Der **AL_TO_AI**-Funktionsblock ist ein nützlicher, schlanker Konverter für den Wechsel zwischen unidirektionalen LWORD- und INT-Adaptern. Er kapselt die bewährte IEC 61131-3-Konvertierung in eine adaptergerechte Form und erleichtert so die Wiederverwendung und den Austausch von Komponenten in IEC 61499-basierten Systemen. Durch seine Ereignissteuerung fügt er sich nahtlos in ereignisgesteuerte Abläufe ein und erfordert keine zusätzliche Konfiguration.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

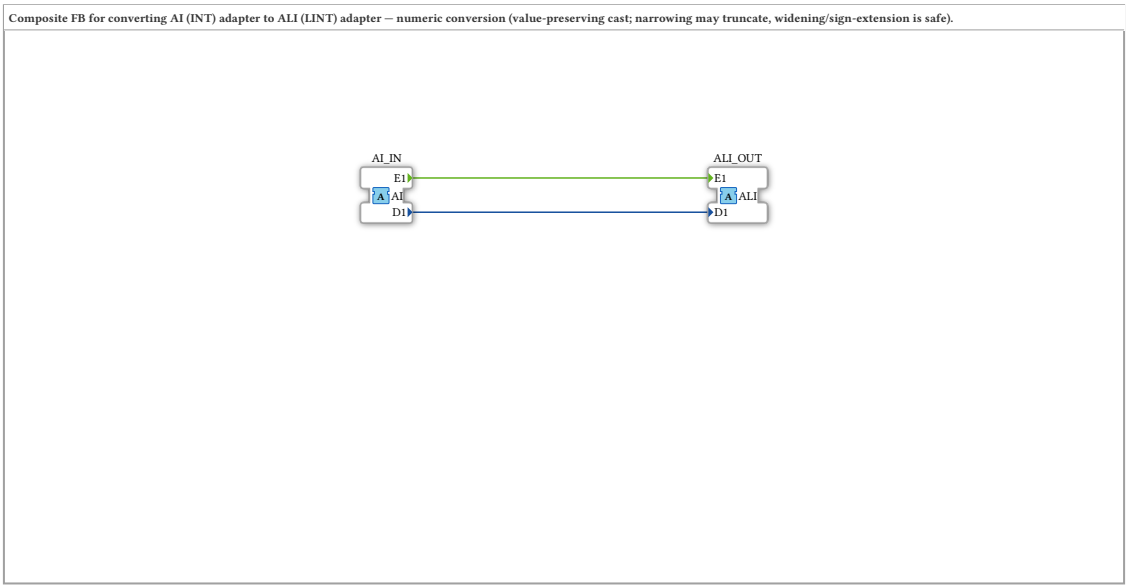
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AI_ALI

AI_TO_ALI



INT Adapter Input – *AI* *ALI* – LINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AI_TO_ALI ist ein Composite-Funktionsblock, der einen Eingangsadapter vom Typ AI (INT) in einen Ausgangsadapter vom Typ ALI (LINT) umwandelt. Er dient als einfache Konvertierungsschnittstelle zwischen unterschiedlichen Adaptertypen für die unidirektionale Kommunikation.

Schnittstellenstruktur

Ereignis-Eingänge

Keine eigenen Ereignis-Eingänge. Die Ereignisübertragung erfolgt über den Eingangsadapter AI_IN. Adapter AI_IN stellt den Ereignis-Eingang E1 bereit.

Ereignis-Ausgänge

Keine eigenen Ereignis-Ausgänge. Die Ereignisweitergabe erfolgt über den Ausgangsadapter ALI_OUT. Adapter ALI_OUT stellt den Ereignis-Ausgang E1 bereit.

Daten-Eingänge

Keine eigenen Daten-Eingänge. Die Datenübertragung erfolgt über den Eingangsadapter AI_IN. Adapter AI_IN stellt den Daten-Eingang D1 (Datentyp INT) bereit.

Daten-Ausgänge

Keine eigenen Daten-Ausgänge. Die Datenweitergabe erfolgt über den Ausgangsadapter ALI_OUT. Adapter ALI_OUT stellt den Daten-Ausgang D1 (Datentyp LINT) bereit.

Adapter

- **AI_IN** (Socket): Typ `adapter::types::unidirectional::AI` – INT-Eingangsschnittstelle.

- **ALI_OUT** (Plug): Typ `adapter::types::unidirectional::ALI` – LINT-Ausgangsschnittstelle.

Hinweis: Die eigentlichen Ereignis- und Datenschnittstellen werden durch die Adapter definiert. Der FB selbst besitzt keine eigenen Schnittstellen.

Funktionsweise

Der FB arbeitet als reine Durchreiche: Das Ereignis- und Datensignal vom Eingangsadapter AI_IN wird direkt an den Ausgangsadapter ALI_OUT weitergeleitet. Dabei wird der INT-Datenwert implizit in einen LINT-Wert konvertiert, da LINT einen größeren Wertebereich abdeckt. Die Umwandlung erfolgt ohne zusätzliche Logik oder Verarbeitung.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- Es handelt sich um einen Composite-FB, der keine eigene Funktionalität implementiert, sondern lediglich zwei Adapter verdrahtet.
- Die Konvertierung von INT nach LINT ist verlustfrei, da LINT (64-Bit) den gesamten Wertebereich von INT (32-Bit) abdeckt.
- Der FB ist für den Einsatz in unidirektionalen Kommunikationspfaden vorgesehen, bei denen eine Anpassung des Datentyps zwischen Adapterschnittstellen erforderlich ist.
- Die Verwendung von Adaptern ermöglicht eine lose Kopplung zwischen Komponenten.

Zustandsübersicht

Der FB besitzt keinen eigenen Zustandsautomaten, da die Funktionalität auf die reine Durchleitung von Ereignissen und Daten reduziert ist. Der interne Zustand ist nicht relevant; das Verhalten ist deterministisch und reagiert instantan auf eingehende Ereignisse.

Anwendungsszenarien

- **Schnittstellenanpassung:** Ein Modul liefert einen INT-Wert über einen AI-Adapter, ein nachfolgendes Modul erwartet jedoch einen LINT-Wert über einen ALI-Adapter.
- **Erweiterung von Wertebereichen:** Wenn ein bestehender INT-Datenpfad auf einen größeren Wertebereich umgestellt werden muss, kann dieser FB als Zwischenstück eingesetzt werden.
- **Adapter-Protokollkonvertierung:** In heterogenen Systemen, in denen verschiedene Adaptertypen verwendet werden, dient der FB als einfacher Konverter.

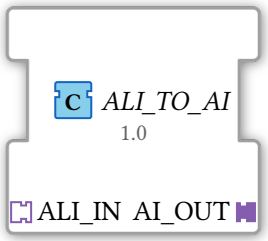
Vergleich mit ähnlichen Bausteinen

- **AI_TO_ALI** ist spezifisch für die Konvertierung von AI nach ALI. Andere Bausteine könnten z. B. AI_TO_AL (INT nach WORD) oder unterschiedliche Richtungen (ALI_TO_AI) realisieren.
- Im Vergleich zu einem generischen Typkonverter (z. B. mit Parametrierung) ist dieser FB spezialisiert und daher einfacher in der Anwendung, aber weniger flexibel.
- Composite-FBs wie dieser bieten eine klare Trennung der Verantwortlichkeiten und können in Bibliotheken für Standardkonvertierungen abgelegt werden.

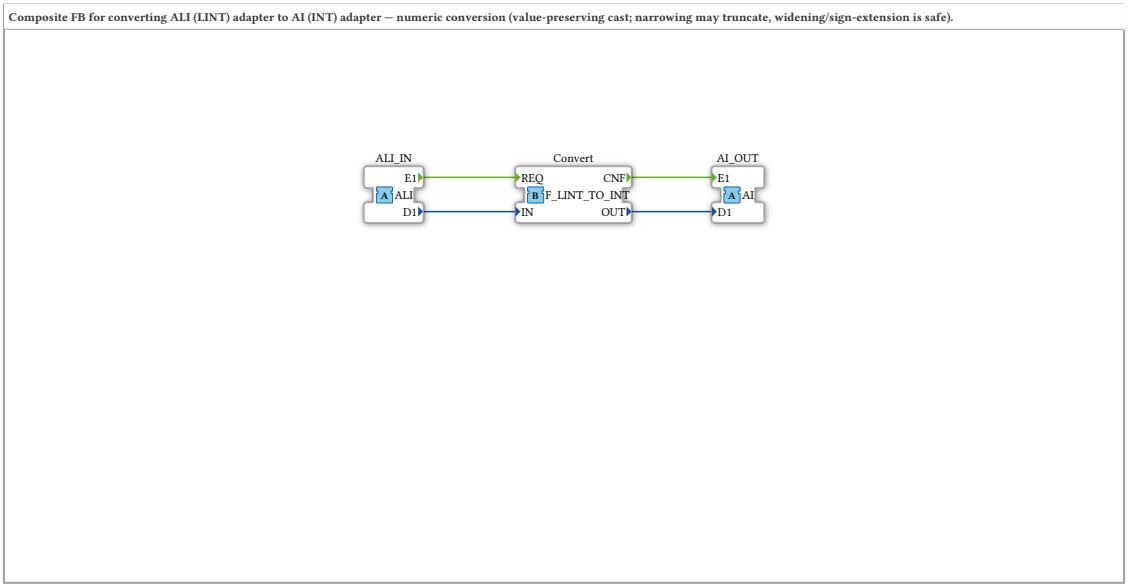
Fazit

Der AI_TO_ALI-Funktionsblock ist ein nützliches Werkzeug zur Adapterkonvertierung in IEC 61499-basierten Systemen. Er ermöglicht eine nahtlose Integration von Komponenten mit unterschiedlichen Datentypen auf der Adapterebene. Durch seine einfache, durchreichende Natur ist er zuverlässig und leicht zu verstehen. Er eignet sich besonders für Projekte, die auf das adapterbasierte unidirektionale Kommunikationsmodell setzen.

ALI_TO_AI



LINT Adapter Input – *ALI* *AI* – INT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ALI_TO_AI** ist ein zusammengesetzter Baustein (Composite FB), der einen **ALI-Adapter** (Eingang einer LINT-Wert-Schnittstelle) in einen **AI-Adapter** (Ausgang einer INT-Wert-Schnittstelle) umwandelt. Er dient dazu, Daten eines LINT-basierten Adapters in den INT-basierten Adaptertyp zu überführen, ohne dass der Anwender die Konvertierung selbst implementieren muss.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Datentyp	Beschreibung
ALI_IN	Ereignis	Steuersignal des ALI-Adapters, löst die Konvertierung aus

Ereignis-Ausgänge

Name	Datentyp	Beschreibung
AI_OUT	Ereignis	Bestätigungssignal des AI-Adapters nach erfolgreicher Konvertierung

Daten-Eingänge

Name	Datentyp	Beschreibung
ALI_IN.D1	LINT	Der umzuwandelnde LINT-Wert

Daten-Ausgänge

Name	Datentyp	Beschreibung
AI_OUT.D1	INT	Der konvertierte INT-Wert

Adapter

Richtung	Name	Typ	Beschreibung
Socket	ALI_IN	ALI (unidirectional)	Liefert LINT-Werte und Steuereignisse
Plug	AI_OUT	AI (unidirectional)	Gibt konvertierte INT-Werte und Ereignisse aus

Funktionsweise

Der Baustein arbeitet als reine Kopplung eines ALI-Sockets mit einem AI-Plug. Im Inneren wird der intern verfügbare Konvertierungsbaustein **F_LINT_TO_INT** verwendet.

1. Ein eingehendes Ereignis am **ALI_IN.E1** triggert den Konvertierungsbaustein.
2. Gleichzeitig wird der aktuelle Wert von **ALI_IN.D1** an den Eingang des Konverters übergeben.
3. Nach Abschluss der Konvertierung wird das Ergebnis an **AI_OUT.D1** weitergeleitet.
4. Ein Bestätigungsereignis wird über **AI_OUT.E1** ausgegeben.

Der gesamte Ablauf erfolgt atomar innerhalb eines Zyklus – Verzögerungen entstehen nur durch die Laufzeit des Konvertierungsbausteins.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Reine Adapterkopplung**: Der Baustein enthält keine eigene Zustandslogik oder Zeitsteuerung.
- **Verwendung von IEC-61131-Konvertierung**: Die eigentliche Typumwandlung erfolgt durch den standardisierten Baustein `F_LINT_TO_INT`.
- **Einfache Zusammensetzung**: Alle Verbindungen sind direkt und ohne zwischengeschaltete Logik.
- **Typ- und ereignisgetrieben**: Die Konvertierung wird nur bei einem Ereignis am ALI-Eingang ausgeführt.

Zustandsübersicht

Als zusammengesetzter Baustein ohne eigenes Zustandsdiagramm hat **ALI_TO_AI** keinen eigenen Zustandsautomaten. Der interne Konvertierungsbaustein `F_LINT_TO_INT` arbeitet nach dem einfachen Muster:

- **IDLE**: Warten auf ein Ereignis.
- **CONVERT**: Ausführen der Konvertierung und Ausgabe des Ergebnisses.

Anwendungsszenarien

- **Systemkopplung**: Wenn ein Sensormodul LINT-Werte über einen ALI-Adapter liefert, das Steuerungssystem jedoch INT-Werte über einen AI-Adapter erwartet.
- **Protokoll- oder Schnittstellenanpassung**: Integration von Komponenten unterschiedlicher Hersteller, die verschiedene Adaptertypen verwenden.
- **Bereinigung von Datenpfaden**: Vereinfachung des Signalflusses durch definierte Konvertierung an einer zentralen Stelle.

Vergleich mit ähnlichen Bausteinen

Baustein	Funktion	Unterschied
ALI_TO_AI	LINT → INT über Adapter	Speziell für unidirektionale ALI-/AI-Schnittstellen
LINT_TO_INT	Direkte Konvertierung LINT → INT	Keine Adapterintegration – reine Datenkonvertierung

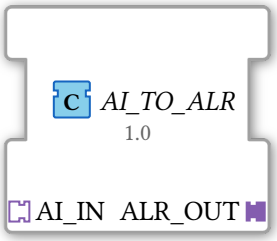
AI_TO_ALI	INT ⟶ LINT (umgekehrte Richtung)	Gleiche Struktur, aber entgegengesetzte Datenrichtung

Fazit

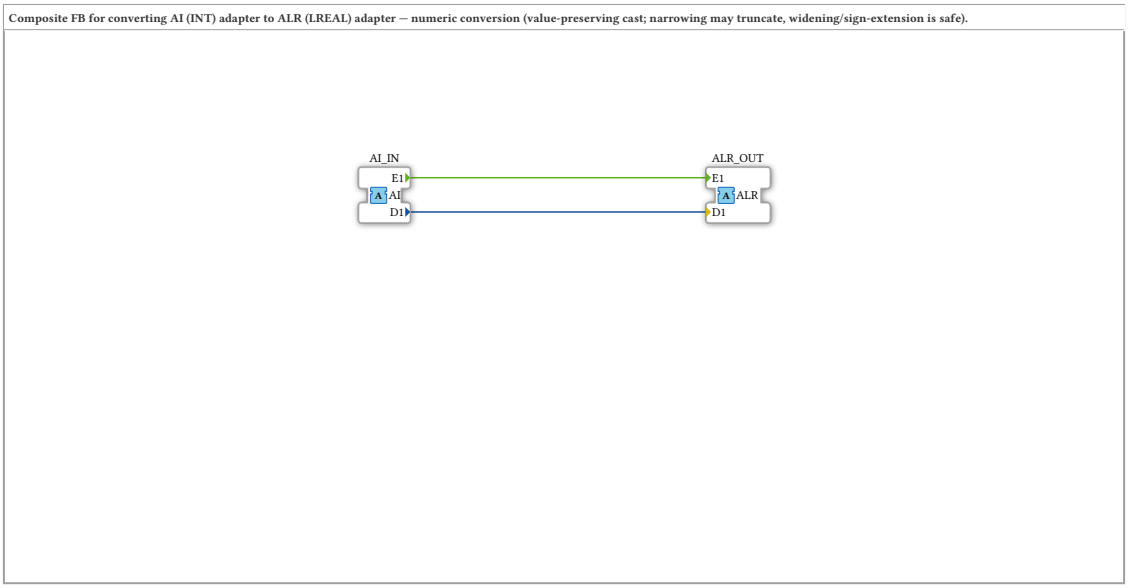
ALI_TO_AI ist ein kleiner, aber nützlicher zusammengesetzter Baustein, der die Lücke zwischen zwei gebräuchlichen IEC-61499-Adaptertypen schließt. Durch die Kapselung der Konvertierungslogik wird die Wiederverwendung vereinfacht und die Systemintegration beschleunigt. Der Baustein ist robust, da er auf einen standardisierten IEC-61131-Konverter zurückgreift, und benötigt keinerlei Konfiguration.

AI_ALR

AI_TO_ALR



INT Adapter Input – *AI* *ALR* – LREAL Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AI_TO_ALR** ist ein Composite-Baustein, der einen **AI (Adapter INT)** in einen **ALR (Adapter LREAL)** umwandelt. Er dient als reine Konvertierungsschnittstelle, indem er die Ereignis- und Datenkanäle des Eingangsadapters direkt auf den Ausgangsadapter durchschleift. Die eigentliche Typkonvertierung von Integer (INT) nach Long Real (LREAL) wird von den zugrunde liegenden Adapterdefinitionen übernommen.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine direkten Ereignis-Eingänge. Das Ereignis wird über den Socket-Adapter `AI_IN` empfangen:

- **AI_IN.E1** – Ereignis des INT-Adapters (Eingang).

Ereignis-Ausgänge

Auch hier gibt es keine direkten Ereignis-Ausgänge. Das Ereignis wird über den Plug-Adapter `ALR_OUT` gesendet:

- **ALR_OUT.E1** – Ereignis des LREAL-Adapters (Ausgang).

Daten-Eingänge

- **AI_IN.D1** – Datenwert vom Typ INT (über den Adapter).

Daten-Ausgänge

- **ALR_OUT.D1** – Datenwert vom Typ LREAL (über den Adapter).

Adapter

- **AI_IN** (Socket) – Adapter vom Typ `AI` (INT-Adapter), Eingangsschnittstelle.
- **ALR_OUT** (Plug) – Adapter vom Typ `ALR` (LREAL-Adapter), Ausgangsschnittstelle.

Funktionsweise

Der Baustein verbindet die Signale des Socket-Adapters `AI_IN` direkt mit dem Plug-Adapter `ALR_OUT`:

- **Ereignispfad:** Das an `AI_IN.E1` eingehende Ereignis wird unverändert an `ALR_OUT.E1` weitergeleitet.
- **Datenpfad:** Der an `AI_IN.D1` anliegende INT-Wert wird an `ALR_OUT.D1` übergeben. Die Adapter selbst implementieren die notwendige Typkonvertierung von INT nach LREAL.

Da es sich um einen Composite-Baustein ohne eigenen Algorithmus oder Zustandsautomaten handelt, erfolgt keine weitere Verarbeitung. Der Baustein fungiert als reine Routing- und Anpassungsschicht zwischen unterschiedlichen Adaptertypen.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Reine Durchleitstruktur:** Keine eigene Logik, keine Latenz oder Pufferung.
- **Abhängigkeit von Adapterdefinitionen:** Die korrekte Funktion setzt voraus, dass die Adapter `AI` und `ALR` gemäß dem 4diac-Standard definiert sind und die Konvertierung INT->LREAL enthalten.
- **Komposit-Baustein:** Einfach erweiterbar, z. B. durch zusätzliche Filter oder Skalierung im Netzwerk.
- **Paketstruktur:** Der Baustein liegt im Paket `adapter::conversion::unidirectional`.

Zustandsübersicht

Der Baustein besitzt kein eigenes Zustandsdiagramm (ECC). Die Signalverarbeitung erfolgt rein ereignisgesteuert über die durchgeschliffenen Adapter-Ereignisse. Ein Zustand ist daher nicht definiert.

Anwendungsszenarien

- **Systemintegration:** Verbindung eines INT-basierten Sensors (z. B. Digitalwert aus Zähler) mit einer Komponente, die LREAL-Werte erwartet (z. B. Regelalgorithmus).
- **Adapter-Brücke:** Einsatz in modularen Automatisierungssystemen, bei denen unterschiedliche Adaptertypen aufeinandertreffen.
- **Prototypenentwicklung:** Schnelle Kopplung von Test-Bausteinen unterschiedlicher Datenformate.

Vergleich mit ähnlichen Bausteinen

Während es in 4diac-Bibliotheken oft separate Konverter-Bausteine für Basisdatentypen gibt (z. B. `INT_TO_REAL`), kapselt `AI_TO_ALR` die Konvertierung auf Adapterebene. Der Vorteil liegt in der Wiederverwendbarkeit von Adapterschnittstellen und der einfachen Austauschbarkeit der Konvertierungslogik durch andere Adapter. Nachteil ist die Abhängigkeit von den konkreten Adapterdefinitionen.

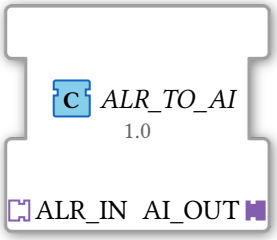
Fazit

Der Baustein **AI_TO_ALR** ist ein minimalistischer, aber effektiver Composite-FB zur Typanpassung zwischen INT- und LREAL-Adapter. Er reduziert den Verdrahtungsaufwand in komplexen Steuerungsnetzwerken und ermöglicht eine saubere Trennung von Schnittstellen und Konvertierungslogik. Für Anwendungen, die eine reine Durchleitung ohne zusätzliche Verarbeitung benötigen, stellt er eine ideale Lösung dar.

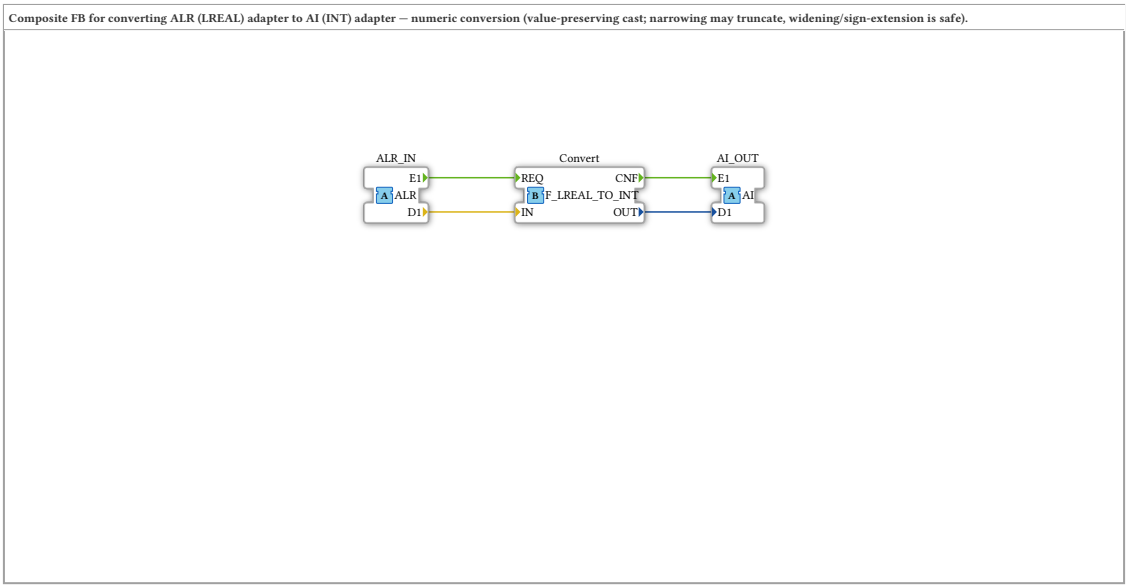
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ALR_TO_AI



LREAL Adapter Input – *ALR* — *AI* – INT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der **ALR_TO_AI** Funktionsblock ist ein Composite-Baustein, der einen Adapter des Typs ALR (LREAL-Daten) in einen Adapter des Typs AI (INT-Daten) umwandelt. Er wird eingesetzt, um Signale aus dem Bereich der landwirtschaftlichen Technik (z. B. Sensorwerte) auf ein standardisiertes integer-basiertes Adapterinterface zu bringen. Der Baustein kapselt die Konvertierung und Ereignisweiterleitung und erleichtert so die Integration in bestehende 4diac-Anwendungen.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine eigenständigen Ereignis-Eingänge auf der obersten Ebene. Die Ereignissteuerung erfolgt über den angeschlossenen Socket `ALR_IN`:

Eingang	Typ	Beschreibung
`ALR_IN`	ALR (Socket)	Adapter, der über sein Ereignis (`E1`) die Konvertierung anstößt

Ereignis-Ausgänge

Der FB besitzt keine eigenständigen Ereignis-Ausgänge auf der obersten Ebene. Die Ereignisrückmeldung erfolgt über den angeschlossenen Plug `AI_OUT`:

Ausgang	Typ	Beschreibung

AI_OUT	AI (Plug)	Adapter, der das konvertierte Ereignis (E1) nach erfolgreicher Umwandlung ausgibt

Daten-Eingänge

Über den Socket ALR_IN wird das **Daten-Eingangssignal** bereitgestellt:

Adapter-Eingang	Datentyp	Beschreibung
ALR_IN.D1	LREAL	Der umzuwandelnde Gleitkommawert

Daten-Ausgänge

Über den Plug AI_OUT wird das **Daten-Ausgangssignal** bereitgestellt:

Adapter-Ausgang	Datentyp	Beschreibung
AI_OUT.D1	INT	Der konvertierte Integer-Wert

Adapter

Der FB besitzt zwei Adapter-Schnittstellen:

- **Socket (Eingang)**: ALR_IN – Adapter für unidirektionale LREAL-Daten (Typ adapter::types::unidirectional::ALR)
- **Plug (Ausgang)**: AI_OUT – Adapter für unidirektionale INT-Daten (Typ adapter::types::unidirectional::AI)

Funktionsweise

Der **ALR_TO_AI** enthält einen internen Konvertierungsbaustein F_LREAL_TO_INT aus der IEC 61131-Bibliothek.

1. Wenn der Socket ALR_IN ein Ereignis an seinem Ausgang E1 sendet, wird dieses an den Eingang REQ des internen Konverters weitergeleitet.
2. Der Konverter wandelt den anliegenden LREAL-Wert (ALR_IN.D1) in einen INT-Wert um und gibt das Ergebnis über seinen Ausgang OUT sowie ein Bestätigungsereignis CNF aus.
3. Das CNF-Ereignis wird an den Plug AI_OUT.E1 weitergegeben, sodass der nachgeschaltete Baustein über die Datenverfügbarkeit informiert wird.
4. Parallel wird der konvertierte INT-Wert auf den Datenausgang AI_OUT.D1 gelegt.

Der gesamte Ablauf erfolgt synchron in einem Schritt.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- Der FB ist ein **Composite-Baustein** – er besitzt keine eigene Zustandsmaschine (ECC), sondern realisiert die Logik über ein internes Subnetzwerk.
- Die Konvertierung folgt der IEC 61131-3-Funktion LREAL_TO_INT: Nachkommastellen werden abgeschnitten (Abschneiden gegen Null), Über- oder Unterläufe sind implementierungsabhängig.
- Der Baustein ist als unidirektionale Adapter-Kopplung ausgelegt – es werden keine Rückkanäle unterstützt.

Zustandsübersicht

Als Composite-FB besitzt **ALR_TO_AI** kein eigenes Zustandsdiagramm. Der interne Konverter F_LREAL_TO_INT arbeitet ereignisgesteuert: Ein Ereignis am Eingang REQ löst die Konvertierung aus, der Ausgang CNF signalisiert das Ende der Operation. Der FB verhält sich wie ein Transparentbaustein für die Ereignis- und Datenweiterleitung.

Anwendungsszenarien

- **Sensorwertaufbereitung**: Umwandlung eines LREAL-Signals (z. B. von einem analogen Sensoradapter) in ein INT-Signal, das von einer SPS oder einer Steuerung mit integer-basierten Adaptern weiterverarbeitet wird.

- **Adapterbrücke**: Verbindung von Adaptertypen unterschiedlicher physikalischer Einheiten, wenn nur der Datentyp, nicht die Skalierung geändert werden muss.
- **Schnittstellenanpassung**: Einsatz in landwirtschaftlichen Steuerungen (z. B. HR Agrartechnik – allgemein), wo LREAL-Werte aus Sensoren auf ein CAN-basiertes INT-Adapterinterface abgebildet werden.

Vergleich mit ähnlichen Bausteinen

FB	Funktion	Unterschied
ALR_TO_AI	Konvertiert LREAL ⟶ INT über Adapter	Composite, keine eigene Zustandsmaschine; verwendet IEC 61131-Konverter
`F_LREAL_TO_INT`	Reine Datenkonvertierung ohne Adapter	Direkter FB, benötigt separate Adapter-Anbindung
`AI_TO_ALR`	Umgekehrte Richtung INT ⟶ LREAL	Anderer Datentyp und Konvertierungsrichtung

Der **ALR_TO_AI** vereinfacht die Wiederverwendung durch Kapselung der Adapterlogik und der Konvertierung in einem Baustein.

Fazit

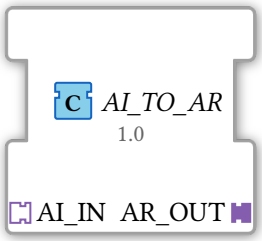
Der **ALR_TO_AI** ist ein praktischer Composite-FB zur standardisierten Umwandlung von LREAL- auf INT-basierte Adapterschnittstellen. Er erlaubt eine saubere Trennung von Datenkonvertierung und Adapterlogik und erleichtert die Integration in größere Automatisierungsprojekte. Dank der klaren Ereignisweiterleitung und der Nutzung einer IEC 61131-Basisfunktion ist er zuverlässig und einfach handhabbar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

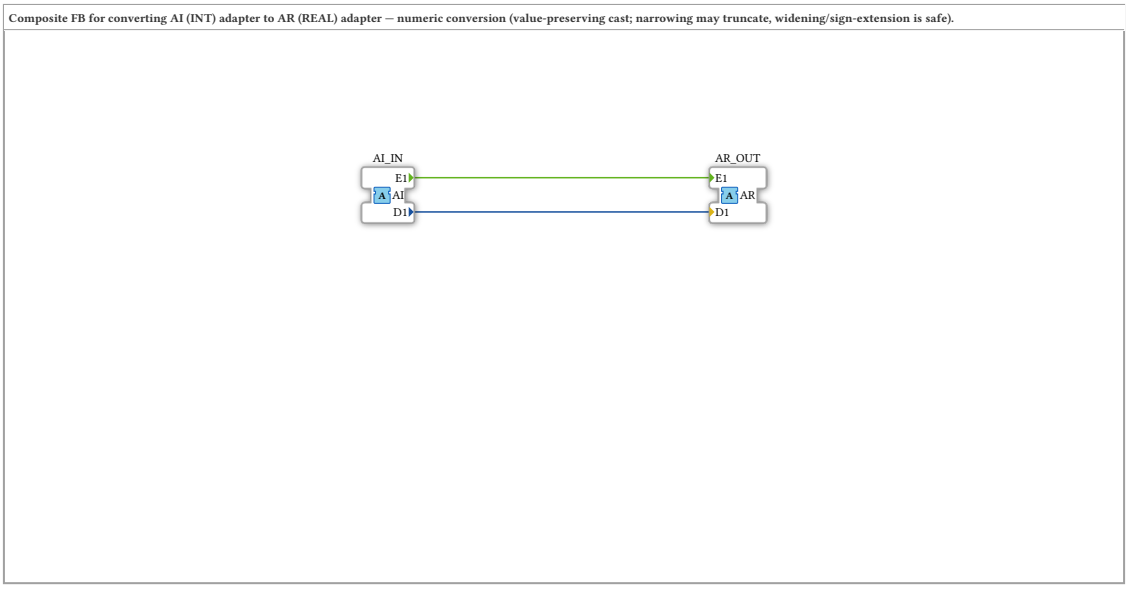
AI_AR

AI_TO_AR



INT Adapter Input – *AI*

AR – REAL Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AI_TO_AR** ist ein Composite-Baustein, der eine Schnittstellenumsetzung von einem **AI**-Adapter (Integer-Typ) auf einen **AR**-Adapter (Real-Typ) ermöglicht. Er dient als reine Durchleitungs- und Anpassungseinheit zwischen verschiedenen Adapterprotokollen in einem industriellen Steuerungssystem.

Schnittstellenstruktur

Da **AI_TO_AR** keine eigenständigen Ereignis- oder Datenschnittstellen besitzt, erfolgt die gesamte Kommunikation ausschließlich über die angeschlossenen Adapter.

Ereignis-Eingänge

– nicht vorhanden – Die Ereignissteuerung erfolgt über den eingehenden Adapter **AI_IN**.

Ereignis-Ausgänge

– nicht vorhanden – Die Ereignisweiterleitung erfolgt über den ausgehenden Adapter **AR_OUT**.

Daten-Eingänge

– nicht vorhanden – Die Daten werden über den **AI_IN**-Adapter als Integer-Wert bereitgestellt.

Daten-Ausgänge

– nicht vorhanden – Die Daten werden über den **AR_OUT**-Adapter als Real-Wert ausgegeben.

Adapter

Name	Typ	Richtung	Beschreibung
AI_IN	`adapter::types::unidirectional::AI`	Socket	Integer-Adapter (Eingang)
AR_OUT	`adapter::types::unidirectional::AR`	Plug	Real-Adapter (Ausgang)

Funktionsweise

Im FBNetzwerk des Composite-Bausteins sind die Ereignis- und Datenleitungen der beiden Adapter direkt miteinander verbunden:

- Das Ereignis **E1** des AI_IN-Adapters wird auf das Ereignis **E1** des AR_OUT-Adapters geleitet.
- Der Datenwert **D1** (Typ INT) des AI_IN-Adapters wird auf den Datenwert **D1** (Typ REAL) des AR_OUT-Adapters übertragen.

Der Baustein führt selbst keine explizite Typkonvertierung durch; die Anpassung von Integer auf Real erfolgt entweder implizit durch die Laufzeitumgebung oder muss durch den angeschlossenen Kontext sichergestellt werden. Somit fungiert der Block als reine Durchschleife für Ereignisse und Daten zwischen den beiden Adapterschnittstellen.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- Der Baustein enthält keinen eigenen Event-oder Data-Interface; seine gesamte Kommunikation läuft über die Adapter.
- Es ist kein interner Zustandsautomat (ECC) vorhanden – der Block arbeitet rein ereignisgesteuert durch die Verbindungen.
- Durch die Verwendung von Adaptern wird eine lose Kopplung zwischen den Ein- und Ausgängen erreicht, was die Wiederverwendbarkeit und Austauschbarkeit der Schnittstellen erhöht.

Zustandsübersicht

Der Baustein besitzt keinen Zustandsautomaten. Die Verarbeitung erfolgt direkt ohne Verzögerung oder Zustandsabhängigkeit.

Anwendungsszenarien

- **Adapter-Konvertierung**: Einsatz in Anlagen, in denen ein Integer-basiertes Sensorsignal (AI) an eine Komponente übergeben werden muss, die ein Real-Signal (AR) erwartet.
- **Schnittstellenanpassung**: Nutzung in modularen Steuerungslösungen, um zwischen verschiedenen Adaptertypen zu vermitteln, ohne die eigentliche Datenverarbeitung zu beeinflussen.
- **Prototypen und Tests**: Schnelles Zusammenstecken von unterschiedlichen Adapter-Komponenten während der Entwicklungsphase.

Vergleich mit ähnlichen Bausteinen

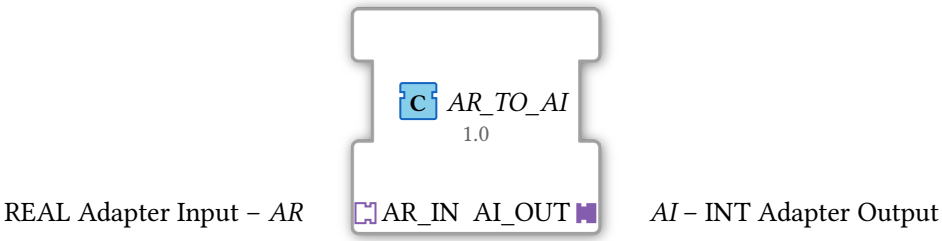
Baustein	Funktion	Besonderheit
AI_TO_AR	Durchleitung von INT nach REAL über Adapter	Keine eigene Logik, reine Adaptervermittlung
REAL_TO_INT	Konvertierung von REAL nach INT (i.d.R. mit Rundung)	Enthält oft einen eigenen Algorithmus
ADAPTER_COUPLER	Allgemeine Adapterkopplung ohne Typanpassung	Erfordert identische Datentypen

Im Gegensatz zu dedizierten Konvertern nimmt **AI_TO_AR** keine aktive Wertumwandlung vor, sondern verlässt sich auf die Kompatibilität der angeschlossenen Adapter.

Fazit

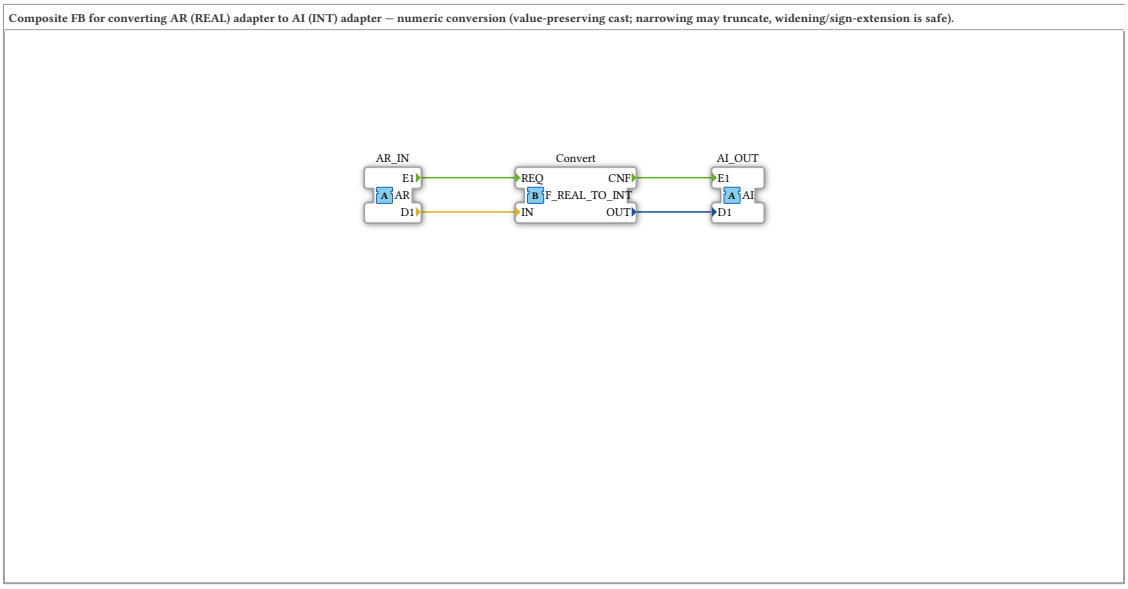
AI_TO_AR ist ein einfacher, aber effektiver Composite-Baustein zur Verbindung zweier unterschiedlicher Adapterschnittstellen (INT nach REAL). Er eignet sich besonders für die systemübergreifende Signalweitergabe in IEC 61499-basierten Steuerungen, bei denen die Logik der Typumsetzung außerhalb des Blocks liegt. Seine klare Struktur macht ihn zu einem nützlichen Werkzeug für modulare und flexible Automatisierungslösungen.

AR_TO_AI



REAL Adapter Input – *AR*

AI – INT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AR_TO_AI** ist ein Composite-Baustein, der einen unidirektionalen AR-Adapter (Eingang mit dem Datentyp REAL) in einen unidirektionalen AI-Adapter (Ausgang mit dem Datentyp INT) umwandelt. Er dient zur numerischen Typkonvertierung innerhalb von 4diac-Netzwerken und kapselt einen IEC-61131-Konvertierungsbaustein.

Schnittstellenstruktur

Ereignis-Eingänge

Der Ereignis-Eingang wird über den Socket **AR_IN** bereitgestellt. Das Ereignis **AR_IN.E1** löst die Konvertierung aus.

Ereignis-Ausgänge

Der Ereignis-Ausgang wird über den Plug **AI_OUT** bereitgestellt. Nach erfolgreicher Konvertierung wird das Ereignis **AI_OUT.E1** gesendet.

Daten-Eingänge

Der Daten-Eingang wird über den Socket **AR_IN** bereitgestellt. Der Eingangswert **AR_IN.D1** ist vom Typ REAL.

Daten-Ausgänge

Der Daten-Ausgang wird über den Plug **AI_OUT** bereitgestellt. Der Ausgangswert **AI_OUT.D1** ist vom Typ INT.

Adapter

Bezeichnung	Richtung	Typ	Bemerkung
AR_IN	Eingang	`adapter::types::unidirectional::AR`	REAL-Adapter (Eingang)
AI_OUT	Ausgang	`adapter::types::unidirectional::AI`	INT-Adapter (Ausgang)

Funktionsweise

Der FB ist als Composite realisiert und enthält einen einzelnen **F_REAL_TO_INT**-Baustein aus der IEC-61131-Bibliothek (`iec61131::conversion::F_REAL_TO_INT`). Sobald am Socket **AR_IN.E1** ein Ereignis eintrifft, wird der Wert von **AR_IN.D1** (REAL) an den Eingang `IN` des Konverters übergeben und dessen Verarbeitung gestartet (`REQ`). Nach Abschluss der Konvertierung signalisiert der Konverter

über `CNF`, dass das Ergebnis am Ausgang `OUT` (INT) anliegt. Dieses Ergebnis wird unverzüglich an den Plug-Ausgang **AI_OUT.D1** (INT) weitergeleitet und das Ereignis **AI_OUT.E1** ausgelöst.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- Der FB ist vollständig ereignisgesteuert; er besitzt keine eigene Zustandsmaschine.
- Die Konvertierung erfolgt laut IEC-61131-Standard: REAL wird auf INT abgebildet (Abschneiden der Nachkommastellen, entsprechend einer Typumwandlung).
- Der Baustein ist im Paket `adapter::conversion::unidirectional` abgelegt, was eine klare Trennung von anderen Konvertierungsbausteinen erlaubt.
- Durch die Verwendung von Adaptern wird die Kapselung und Wiederverwendbarkeit in 4diac-Projekten verbessert.

Zustandsübersicht

Der **AR_TO_AI** besitzt keine interne Zustandsmaschine. Die Funktionsweise ist rein kombinatorisch in Abhängigkeit vom eingehenden Ereignis:

- **Warten:** Es liegt kein Ereignis an AR_IN.E1 an.
- **Konvertieren:** Bei Ereignis wird der REAL-Wert unverzögert konvertiert und das Ergebnis als INT ausgegeben.

Anwendungsszenarien

- **Sensor-Anbindung:** Realwert-Sensoren (z.B. analoge Spannungsmessung) liefern REAL-Daten über einen AR-Adapter; nachgeschaltete Logik benötigt INT-Werte (z.B. für Zähler oder Indexberechnungen).
- **Protokoll-Konvertierung:** Wenn ein Subsystem Daten im REAL-Format sendet, das Zielsystem aber nur INT-Daten akzeptiert (z.B. bei Feldbus-Gateways).
- **Test- und Simulationsumgebungen:** Zur einfachen Wandlung zwischen REAL- und INT-Adaptern in Testaufbauten.

Vergleich mit ähnlichen Bausteinen

- **AI_TO_AR:** Führt die umgekehrte Konvertierung durch (INT ⟶ REAL).
- **F_REAL_TO_INT (direkt):** Der eingekapselte Basisbaustein ist für Punkt-zu-Punkt-Verbindungen ohne Adapter gedacht. Der AR_TO_AI bietet eine adapterbasierte Schnittstelle, die in Adapter-orientierten Netzwerken besser eingebunden werden kann.
- **Eigene Composite-Bausteine:** Andere Projekte könnten ähnliche Adapter-Konvertierungen für andere Datentypen (z.B. LREAL, DINT) erstellen.

Fazit

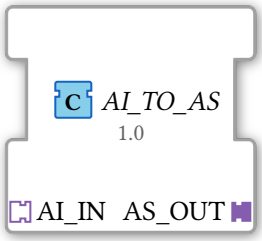
Der **AR_TO_AI** ist ein kompakter, adapterbasierter Konvertierungsbaustein, der die Umwandlung von REAL- nach INT-Daten in ereignisgesteuerten 4diac-Netzwerken ermöglicht. Durch die Kapselung des Standardbausteins `F_REAL_TO_INT` in eine Adapterschnittstelle wird die Integration in bestehende Adapter–Sockets-Architekturen vereinfacht und die Lesbarkeit des Gesamtsystems erhöht.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

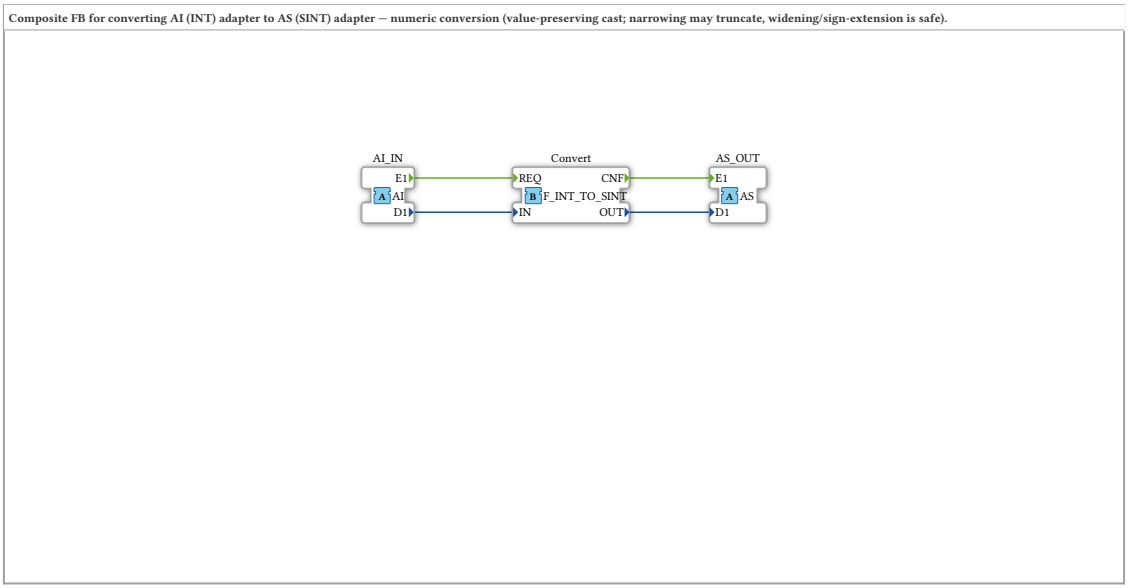
AI_AS

AI_TO_AS



INT Adapter Input – *AI*

AS – SINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AI_TO_AS** ist ein Composite-Baustein, der einen Adapter vom Typ `AI` (basierend auf `INT`) in einen Adapter vom Typ `AS` (basierend auf `SINT`) umwandelt. Er dient als Schnittstellenkonverter zwischen unterschiedlichen Datenformaten in der IEC 61499-Architektur.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine eigenen Ereignis-Eingänge. Die Ereignissteuerung erfolgt vollständig über den Adapter-Eingang `AI_IN` (siehe Abschnitt Adapter). Intern wird das Ereignis `E1` des Adapters verarbeitet.

Ereignis-Ausgänge

Der FB besitzt keine eigenen Ereignis-Ausgänge. Das Ergebnisereignis wird über den Adapter-Ausgang `AS_OUT` bereitgestellt (siehe Abschnitt Adapter). Nach erfolgreicher Konvertierung wird das Ereignis `E1` auf dem Ausgangsadapter ausgelöst.

Daten-Eingänge

Der FB besitzt keine eigenen Daten-Eingänge. Der zu konvertierende `INT`-Wert wird über den Adapter-Eingang `AI_IN` (Datenpunkt `D1`) eingelesen.

Daten-Ausgänge

Der FB besitzt keine eigenen Daten-Ausgänge. Der konvertierte `SINT`-Wert wird über den Adapter-Ausgang `AS_OUT` (Datenpunkt `D1`) ausgegeben.

Adapter

Adapter	Richtung	Typ	Beschreibung
AI_IN	Eingang (Socket)	adapter::types::unidirectional::AI	INT-basierter Adapter-Eingang; liefert das Ereignis E1 und den Datenwert D1 (Typ INT).
AS_OUT	Ausgang (Plug)	adapter::types::unidirectional::AS	SINT-basierter Adapter-Ausgang; empfängt das Ereignis E1 und gibt den konvertierten Datenwert D1 (Typ SINT) weiter.

Funktionsweise

Der Baustein führt eine **Typkonvertierung von INT nach SINT** durch. Sobald am Adapter-Eingang AI_IN ein Ereignis E1 eintrifft, wird der zugehörige Datenwert D1 (INT) ausgelesen. Dieser Wert wird dem internen Baustein F_INT_TO_SINT übergeben. Die Konvertierung erfolgt gemäß der IEC 61131-3-Definition; dabei wird der INT-Wert auf den Wertebereich von SINT (-128…127) begrenzt. Nach erfolgreicher Umwandlung wird am Adapter-Ausgang AS_OUT das Ereignis E1 erzeugt und der konvertierte SINT-Wert über D1 übergeben.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-Struktur**: Der FB ist als Composite realisiert, der auf den Standard-Baustein F_INT_TO_SINT zurückgreift. Dies ermöglicht eine einfache Wiederverwendung und klare Trennung der Konvertierungslogik.
- **Adapter-basierte Schnittstelle**: Die Ein- und Ausgänge sind als unidirektionale Adapter (AI und AS) ausgeführt, was eine flexible Anbindung an andere Komponenten mit kompatiblen Adapterschnittstellen erlaubt.
- **Keine Bereichsüberprüfung**: Der zugrunde liegende Baustein F_INT_TO_SINT führt keine explizite Fehlerbehandlung bei Wertebereichsüberschreitung durch. Liegt der INT-Wert außerhalb des SINT-Bereichs, kann es zu unerwarteten Ergebnissen oder Überläufen kommen (abhängig von der Zielplattform).

Zustandsübersicht

Der FB verfügt über keine eigene Zustandsmaschine. Die interne Logik folgt dem einfachen Ablauf des Bausteins F_INT_TO_SINT:

Zustand	Beschreibung
Idle	Warten auf ein Ereignis an AI_IN.E1.
Convert	Bei Eintreffen von E1 wird der Wert von AI_IN.D1 gelesen, konvertiert und an AS_OUT.D1 übergeben. Anschließend wird AS_OUT.E1 ausgelöst. Der FB kehrt sofort in den Idle-Zustand zurück.

Anwendungsszenarien

- **Datenformat-Anpassung**: Wenn eine vorliegende INT-Schnittstelle in ein System integriert werden soll, das ausschließlich SINT-Werte verarbeitet (z. B. ältere Steuerungen oder begrenzte Busprotokolle).
- **Adapter-Wechsel**: Ersetzen eines AI-Adapters durch einen AS-Adapter in einer bestehenden Komponentenstruktur, ohne die interne Logik anzupassen.
- **Prototypen/Testaufbauten**: Schnelle Umwandlung von Testsignalen zwischen unterschiedlichen Datenbreiten.

Vergleich mit ähnlichen Bausteinen

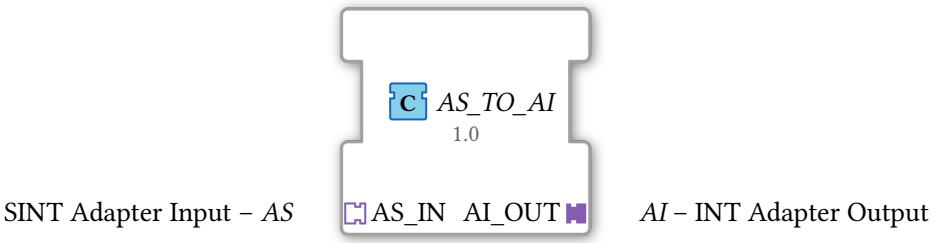
- **AI_TO_AS (INT → SINT)**: Speziell für die Konvertierung von INT zu SINT mit unidirektionalen Adaptern.

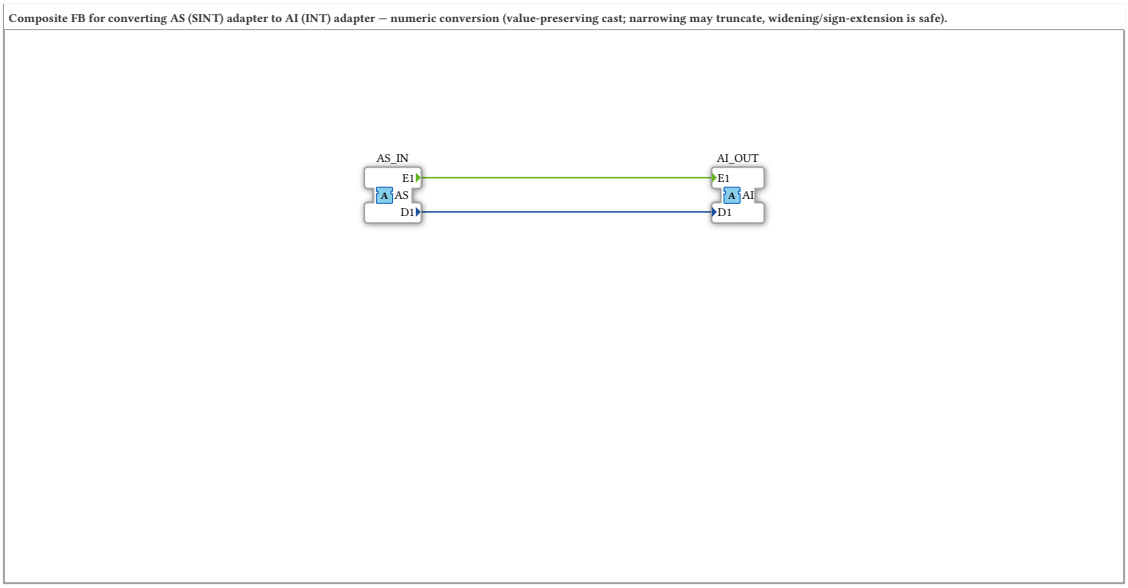
- **AY_TO_AS (DINT → SINT)**: Konvertiert DINT-Werte in SINT; prinzipiell gleicher Aufbau, aber anderer Datentyp.
- **Direkte Konvertierung ohne Adapter**: Bausteine wie `F_INT_TO_SINT` bieten die reine Typumwandlung, erfordern aber separate Ereignis- und Datenverbindungen. Der `AI_TO_AS` kapselt diese in einer Adapterlogik.

Fazit

Der `AI_TO_AS`-Baustein stellt eine einfache und kompakte Lösung zur Umwandlung eines INT-basierten Adapters in einen SINT-basierten Adapter dar. Durch die Composite-Struktur und die klare Schnittstellendefinition lässt er sich leicht in IEC 61499-Applikationen integrieren. Die fehlende Bereichsprüfung sollte bei sicherheitskritischen Anwendungen beachtet werden. Insgesamt ist er ein nützlicher Helfer für die Signalaufbereitung in heterogenen Automatisierungssystemen.

AS_TO_AI





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AS_TO_AI` ist ein Composite-Funktionsblock, der einen AS-Adapter (Datentyp SINT) in einen AI-Adapter (Datentyp INT) umwandelt. Er dient als einfacher Konverter, um Komponenten mit unterschiedlichen Adapter-Typen innerhalb der 4diac-IDE zu verbinden.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine

Ereignis-Ausgänge

- Keine

Daten-Eingänge

- Keine

Daten-Ausgänge

- Keine

Adapter

- **AS_IN** (Socket, Typ: `adapter::types::unidirectional::AS`) Erwartet einen SINT-Wert und ein zugehöriges Ereignis E1.
- **AI_OUT** (Plug, Typ: `adapter::types::unidirectional::AI`) Stellt einen INT-Wert und ein Ereignis E1 bereit.

Funktionsweise

Der FB leitet das Ereignis E1 vom AS_IN-Socket direkt zum AI_OUT-Plug weiter. Parallel dazu wird der Datenwert D1 vom AS_IN-Socket auf den AI_OUT-Plug übertragen. Dabei erfolgt eine implizite Typkonvertierung von SINT (8-Bit vorzeichenbehaftet) nach INT (16-Bit vorzeichenbehaftet). Es findet keine weitere Verarbeitung oder Zustandslogik statt.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- Die Typkonvertierung von SINT zu INT wird automatisch von der 4diac-Laufzeitumgebung durchgeführt, ohne dass zusätzliche Code-Blöcke erforderlich sind.
- Der Baustein ist als **Composite FB** realisiert und enthält kein eigenes ECC (Execution Control Chart).
- Er ist im Paket `adapter::conversion::unidirectional` abgelegt, was auf unidirektionale Adapter-Konvertierung hinweist.

Zustandsübersicht

Da es sich um einen Composite-Funktionsblock handelt, besitzt `AS_TO_AI` keine eigenen Zustände. Das Verhalten ergibt sich ausschließlich aus der Verdrahtung der internen Verbindungen.

Anwendungsszenarien

- Ein Sensor liefert Daten über einen AS-Adapter (SINT), aber die angeschlossene Verarbeitungskomponente erwartet einen AI-Adapter (INT).
- Migration von 8-Bit- zu 16-Bit-Datenpfaden, ohne die Quell- oder Zielkomponente zu ändern.
- Prototyping und Systemintegration, wenn unterschiedliche Adapterversionen aufeinandertreffen.

Vergleich mit ähnlichen Bausteinen

- **AI_TO_AS**: Führt die umgekehrte Konvertierung (INT → SINT) durch.
- **AI_TO_AD** o. Ä.: Konvertiert zwischen anderen ganzzahligen Datentypen (z. B. INT nach DINT).
- Im Gegensatz zu diesen Bausteinen arbeitet `AS_TO_AI` verlustfrei, da der Zieltyp (INT) den Quelltyp (SINT) vollständig abbilden kann.

Fazit

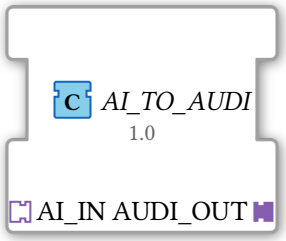
Der `AS_TO_AI`-Funktionsblock ist ein einfaches, aber effektives Werkzeug zur Sicherstellung der Adapter-Kompatibilität in 4diac-Anwendungen. Er übernimmt die automatische Typkonvertierung von SINT nach INT und erleichtert so die Integration heterogener Komponenten ohne manuelle Anpassung des Datenpfades.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

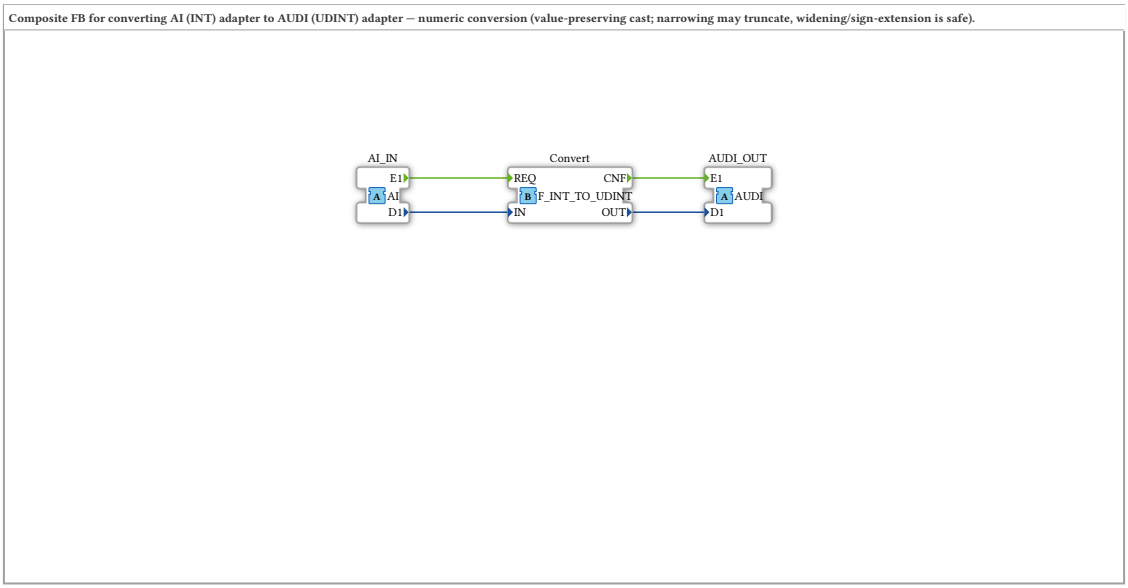
AI_AUDI

AI_TO_AUDI



INT Adapter Input – *AI*

AUDI – UDINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AI_TO_AUDI` ist ein Composite-Baustein, der einen **AI-Adapter** (Daten vom Typ `INT`) in einen **AUDI-Adapter** (Daten vom Typ `UDINT`) umwandelt. Er dient als unidirektionale Schnittstelle zwischen unterschiedlichen Adaptertypen und ermöglicht eine nahtlose Datenkonvertierung innerhalb der 4diac-IDE.

Schnittstellenstruktur

Der Baustein besitzt keine klassischen Ereignis- und Datenein-/ausgänge, sondern realisiert seine Ein- und Ausgaben über **Adapter-Schnittstellen**. Folgende Ports sind über die internen Adapter verfügbar:

Ereignis-Eingänge

Port	Adapter	Beschreibung
`E1`	`AI_IN`	Startet die Konvertierung des INT-Wertes.

Ereignis-Ausgänge

Port	Adapter	Beschreibung
`E1`	`AUDI_OUT`	Signalisiert Abschluss der Konvertierung.

Daten-Eingänge

Port	Adapter	Datentyp	Beschreibung

D1	AI_IN	INT	Zu konvertierender Integer-Wert.

Daten-Ausgänge

Port	Adapter	Datentyp	Beschreibung
D1	AUDI_OUT	UDINT	Konvertierter unsigned Integer.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	AI_IN	adapter::types::unidirectional::AI	Eingangsadapter (INT).
Plug	AUDI_OUT	adapter::types::unidirectional::AUDI	Ausgangsadapter (UDINT).

Funktionsweise

Der Baustein ist als Composite-FB realisiert. Intern nutzt er den Standard-Konvertierungsbaustein `iec61131::conversion::F_INT_TO_UDINT`.

Ablauf:

1. Am Socket `AI_IN` trifft ein Ereignis am Eingang `E1` ein, begleitet von einem INT-Wert an `D1`.
2. Dieses Ereignis wird an den internen Baustein `Convert` (Typ `F_INT_TO_UDINT`) weitergeleitet.
3. Der interne Baustein führt die Typkonvertierung von `INT` nach `UDINT` durch (gemäß IEC 61131-3).
4. Nach erfolgreicher Konvertierung wird ein Ereignis am Ausgang `CNF` des internen Bausteins erzeugt.
5. Dieses Ereignis wird an den Plug `AUDI_OUT` (Ereignisausgang `E1`) weitergegeben, gleichzeitig liegt der konvertierte UDINT-Wert am Datenausgang `D1` an.

Die Verbindungen im FBNetwork sind:

- `AI_IN.E1` ⟶ `Convert.REQ`
- `Convert.CNF` ⟶ `AUDI_OUT.E1`
- `AI_IN.D1` ⟶ `Convert.IN`
- `Convert.OUT` ⟶ `AUDI_OUT.D1`

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-Baustein:** Vereinfacht die Wiederverwendung und Kapselung der Konvertierungslogik.
- **Standardkonvertierung:** Verwendet den IEC 61131-3 Baustein `F_INT_TO_UDINT`, der eine saubere und portable Typumwandlung gewährleistet.
- **Unidirektionale Adapter:** Sowohl der Eingangs- als auch der Ausgangsadapter sind als unidirektionale Schnittstellen (Socket bzw. Plug) ausgeführt – die Daten fließen nur in eine Richtung.
- **Ereignisgesteuert:** Die Konvertierung wird durch ein Ereignis am Eingangsadapter ausgelöst und durch ein Ereignis am Ausgangsadapter quittiert (Handshake-Prinzip).

Zustandsübersicht

Der Baustein selbst besitzt keine eigene Zustandsmaschine, da er ausschließlich aus einer direkten Verkettung von Adapterports und dem internen Konvertierungsbaustein besteht. Der interne FB `F_INT_TO_UDINT` führt die Konvertierung in einem Schritt aus – eine Zustandsübersicht entfällt daher.

Anwendungsszenarien

- **Datenbrücke zwischen verschiedenen Adaptertypen:** Wenn ein übergeordnetes System oder eine Bibliothek Adapter vom Typ `AI` (mit INT-Daten) bereitstellt, aber nachgelagerte Komponenten einen `AUDI`-Adapter (mit UDINT-Daten) erwarten.
- **Signalanpassung in Automatisierungslösungen:** Zum Beispiel bei der Verarbeitung von Zählerwerten oder Positionsdaten, die als vorzeichenbehaftete 16-Bit-Werte ankommen, aber als vorzeichenlose 32-Bit-Werte weitergegeben werden müssen.

- **Modulare Adapterkonvertierung:** Einfacher Austausch oder Erweiterung der Konvertierungslogik durch Austausch des internen Bausteins.

Vergleich mit ähnlichen Bausteinen

Es existieren weitere Adapter-Konvertierungsbausteine in der Bibliothek, z. B.:

- `AI_TO_AI` – weiterleiten ohne Konvertierung
- `AUDI_TO_AI` – umgekehrte Richtung (UDINT → INT)
- Direkte Nutzung des `F_INT_TO_UDINT`-Bausteins mit manueller Adapterverdrahtung

Der Baustein `AI_TO_AUDI` kapselt nicht nur die Typkonvertierung, sondern auch die komplette Adapteranbindung, was den Entwurf übersichtlicher macht und Fehler durch falsche Verdrahtung vermeidet.

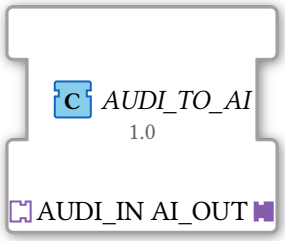
Fazit

Der Composite-Baustein `AI_TO_AUDI` bietet eine einfache, aber saubere Lösung zur Konvertierung eines INT-basierten Adapterprotokolls in ein UDINT-basiertes Protokoll. Durch die Verwendung des genormten IEC-Bausteins sowie die ereignisgesteuerte Kopplung ist er zuverlässig und leicht in größere 4diac-Projekte integrierbar.

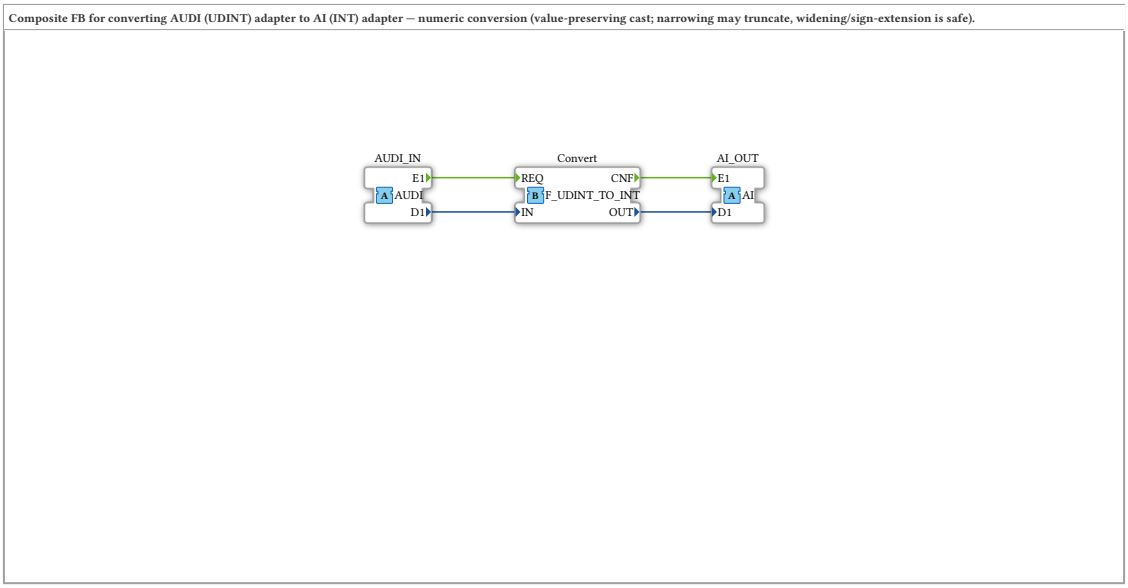
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUDI_TO_AI



UDINT Adapter Input – *AUDI* — *AI* – INT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Composite-Funktionsblock `AUDI_TO_AI` dient der Umwandlung eines unidirektionalen AUDI-Adapters (Datentyp `UDINT`) in einen unidirektionalen AI-Adapter (Datentyp `INT`). Er kapselt die Konvertierungslogik und ermöglicht eine einfache Integration in Adapter-basierte Kommunikationsstrukturen gemäß IEC 61499.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine direkten Ereignis-Eingänge. Die Ereignissteuerung erfolgt über den Socket `AUDI_IN` vom Typ `adapter::types::unidirectional::AUDI`. Dieser Adapter stellt einen Ereignis-Eingang `E1` bereit, der die Konvertierung anstößt.

Ereignis-Ausgänge

Der FB besitzt keine direkten Ereignis-Ausgänge. Die Ereignisausgabe erfolgt über den Plug `AI_OUT` vom Typ `adapter::types::unidirectional::AI`. Dieser Adapter stellt einen Ereignis-Ausgang `E1` bereit, der den Abschluss der Konvertierung signalisiert.

Daten-Eingänge

Keine direkten Daten-Eingänge. Der Socket `AUDI_IN` liefert über seinen Daten-Eingang `D1` den umzuwandelnden `UDINT`-Wert.

Daten-Ausgänge

Keine direkten Daten-Ausgänge. Der Plug `AI_OUT` gibt über seinen Daten-Ausgang `D1` den konvertierten `INT`-Wert aus.

Adapter

- **Socket `AUDI_IN`**: Typ `adapter::types::unidirectional::AUDI` – empfängt einen unidirektionalen Datenwert vom Typ `UDINT` sowie ein dazugehöriges Ereignis.
- **Plug `AI_OUT`**: Typ `adapter::types::unidirectional::AI` – sendet einen unidirektionalen Datenwert vom Typ `INT` sowie ein dazugehöriges Ereignis.

Funktionsweise

Der FB ist als Composite-Baustein realisiert. Intern wird der Socket `AUDI_IN` mit dem Standard-Konvertierungsbaustein `iec61131::conversion::F_UDINT_TO_INT` verbunden, dessen Ausgang wiederum mit dem Plug `AI_OUT` verknüpft ist. Wird über den Adapter-Eingang `AUDI_IN.E1` ein Ereignis empfangen, wird der zugehörige Datenwert `AUDI_IN.D1` an den Konvertierungsbaustein übergeben. Nach erfolgreicher Umwandlung von `UDINT` nach `INT` wird das Ereignis `Convert.CNF` erzeugt und über `AI_OUT.E1` ausgegeben. Gleichzeitig steht der konvertierte Wert an `AI_OUT.D1` zur Verfügung.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- Nutzt einen standardisierten IEC 61131-Konvertierungsbaustein, der plattformunabhängig implementiert werden kann.
- Die Umwandlung von `UDINT` (32 Bit ohne Vorzeichen) nach `INT` (16 Bit mit Vorzeichen) ist potenziell verlustbehaftet. Bei Wertebereichsüberschreitung kann je nach Implementierung des Konvertierungsbausteins ein Überlauf oder eine Begrenzung auftreten.
- Die Verwendung unidirektionaler Adapter stellt einen klaren Datenfluss sicher und erleichtert die Integration in modulare Systeme.
- Der FB besitzt keine eigene Zustandsmaschine; die gesamte Logik wird ereignisgesteuert durch die interne Verdrahtung ausgeführt.

Zustandsübersicht

Der Baustein verfügt über keinen internen Zustandsautomaten. Die Funktionalität ergibt sich ausschließlich aus der kausalen Verkettung von Ereignis- und Datenverbindungen. Es existiert lediglich ein aktiver Zustand, in dem eine eintreffende Ereignis-Daten-Kombination verarbeitet und die Ausgabe erzeugt wird.

Anwendungsszenarien

- Anbindung eines Sensors, der Messwerte als `UDINT` liefert, an ein Steuerungssystem, das `INT`-Werte erwartet (z. B. SPS mit 16-Bit-Analogeingängen).
- Integration in PROFINET- oder EtherCAT-Anwendungen, bei denen Adapter zur Typanpassung zwischen unterschiedlichen Datenformaten benötigt werden.

- Wiederverwendbare Kapselung der Konvertierung in einer Adapter-basierten Bibliothek für industrielle Automatisierungslösungen.

Vergleich mit ähnlichen Bausteinen

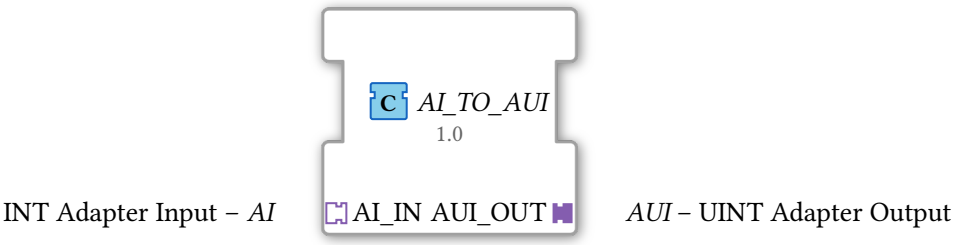
- Ein direkter Konvertierungsbaustein wie `F_UDINT_TO_INT` erfordert separate Ereignis- und Datenverbindungen und ist nicht für den direkten Einsatz in Adapter-Schnittstellen ausgelegt.
- `AUDI_TO_AI` kapselt die gesamte Adapterkonvertierung und vereinfacht so die Wiederverwendung in Architekturen, die auf unidirektionalen Adaptern basieren.
- Analoge Bausteine für andere Datentypen (z. B. `REAL_TO_INT` oder `DINT_TO_INT`) folgen demselben Entwurfsmuster und können nach dem gleichen Schema realisiert werden.

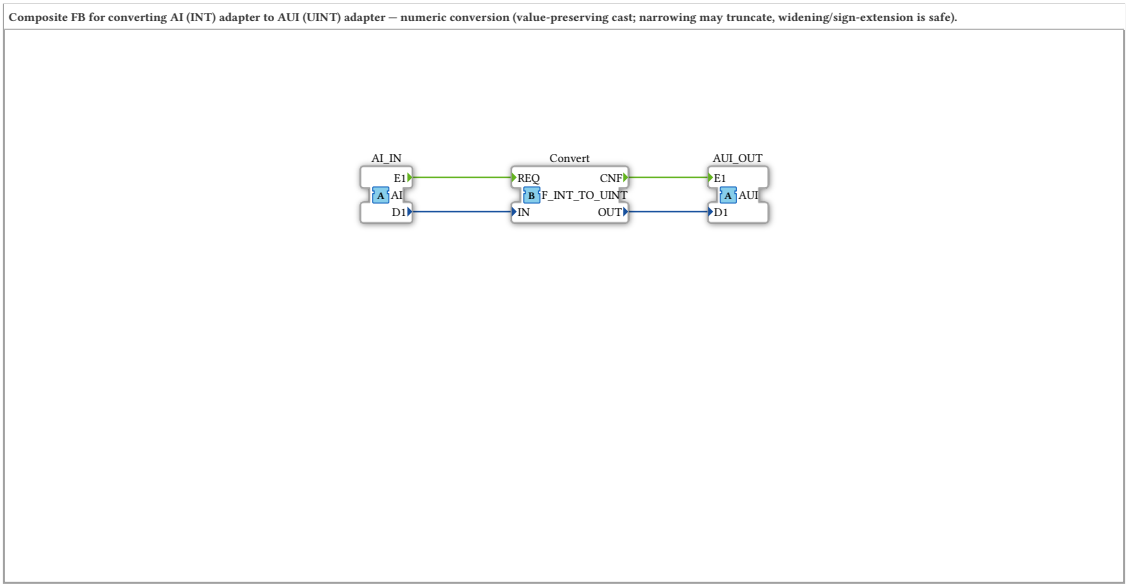
Fazit

Der Composite-Baustein `AUDI_TO_AI` bietet eine saubere und wiederverwendbare Lösung zur verlustbehafteten Umwandlung von `UDINT`- in `INT`-Werte unter Verwendung unidirektionaler Adapter. Er vereinfacht die Integration in IEC 61499-Systeme, die auf Adapterkommunikation setzen, und kapselt die Konvertierungslogik in einem einzigen, klar definierten Funktionsblock.

AI_AUI

AI_TO_AUI





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AI_TO_AUI** ist ein Composite-Baustein, der einen INT-Adapter (AI) in einen UINT-Adapter (AUI) umwandelt. Er dient als Schnittstellenkonverter in Systemen, die auf unidirektionale Adapter-Kommunikation setzen. Im Inneren nutzt er den Baustein `F_INT_TO_UINT` aus der IEC 61131-Bibliothek, um den Integer-Wert von `AI_IN` in einen vorzeichenlosen Integer zu konvertieren und über `AUI_OUT` bereitzustellen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine separaten Ereignis-Eingänge vorhanden. Die Ereignissteuerung erfolgt ausschließlich über die Adapter (`AI_IN.E1` für Anforderung, `AUI_OUT.E1` für Bestätigung).

Ereignis-Ausgänge

Keine separaten Ereignis-Ausgänge vorhanden. Die Ereignisrückmeldung wird über den Adapter `AUI_OUT.E1` bereitgestellt.

Daten-Eingänge

Keine separaten Daten-Eingänge vorhanden. Der zu konvertierende Wert wird über den Adapter `AI_IN.D1` (Typ INT) angenommen.

Daten-Ausgänge

Keine separaten Daten-Ausgänge vorhanden. Der konvertierte Wert wird über den Adapter `AUI_OUT.D1` (Typ UINT) bereitgestellt.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	`AI_IN`	`adapter::types::unidirectional::AI`	INT-Adapter-Eingang. Liefert das Ereignis `E1` und den Datenwert `D1` (INT).
Plug (Ausgang)	`AUI_OUT`	`adapter::types::unidirectional::AUI`	UINT-Adapter-Ausgang. Stellt das Ereignis `E1` und den Datenwert `D1` (UINT) bereit.

Funktionsweise

Der FB arbeitet als einfache Pipeline:

1. Ein Ereignis wird über `AI_IN.E1` empfangen.
2. Dieses Ereignis triggert den internen Konvertier-Baustein `F_INT_TO_UINT` über dessen `REQ`-Eingang.
3. Der Wert von `AI_IN.D1` (INT) wird an `Convert.IN` weitergeleitet.
4. Nach Abschluss der Konvertierung wird über `Convert.CNF` ein Ereignis an `AUI_OUT.E1` gesendet.
5. Der konvertierte Wert (UINT) steht an `AUI_OUT.D1` zur Verfügung.

Der Baustein ist rein kausal – er reagiert auf ein eingehendes Ereignis und gibt nach der Umwandlung ein ausgehendes Ereignis aus.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Adapter-basierte Kommunikation**: Statt klassischer Ein-/Ausgangspins werden die Schnittstellen über Adapter (`AI` und `AUI`) realisiert. Dies ermöglicht eine lose Kopplung in modularen Systemen.
- **Unidirektionaler Datenfluss**: Beide Adapter sind unidirektional, d. h. sie transportieren Daten und Ereignisse nur in eine Richtung (Ein- oder Ausgang).
- **Paketierung**: Der Baustein befindet sich im Paket `adapter::conversion::unidirectional`, was seine Rolle als Konverter zwischen Adapter-Typen kennzeichnet.
- **Typumwandlung**: Die Konvertierung von `INT` in `UINT` erfolgt mittels des IEC-61131-Bausteins `F_INT_TO_UINT`, der eine Wert-Erhaltung bei positiven Zahlen gewährleistet. Negative Werte werden in den entsprechenden UINT-Raum (z. B. Zweierkomplement) umgesetzt.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten. Er durchläuft intern nur den Ablauf: Ereignis empfangen → Konvertierung auslösen → Ereignis senden. Es gibt keine verzögerten Reaktionen oder dauerhaften Zustände.

Anwendungsszenarien

- **Sensor-Schnittstellenanpassung**: Ein Sensor liefert Messwerte als `INT` (z. B. Temperatur oder Druck) über einen AI-Adapter; die nachfolgende Steuerung erwartet `UINT` über einen AUI-Adapter.
- **Systemintegration**: In heterogenen Systemen, in denen unterschiedliche Adapter-Typen verwendet werden, ermöglicht `AI_TO_AUI` die nahtlose Verbindung von Komponenten.
- **Datenaufbereitung**: Vor der Weiterverarbeitung in einer SPS, die nur vorzeichenlose Werte unterstützt (z. B. für Zähler oder Skalierung).

Vergleich mit ähnlichen Bausteinen

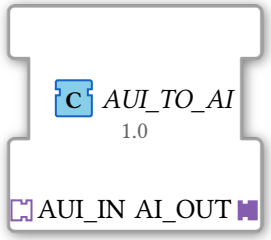
- **`AI_TO_AUI` vs. direkte Konvertierung mit `F_INT_TO_UINT`**: Der Composite-Baustein kapselt die Adapter-Schnittstellen und macht die Anwendung für den Entwickler einfacher – es müssen keine einzelnen FBs mit Signalverbindungen verdrahtet werden.
- **Ähnliche Adapter-Konverter**: In der Bibliothek existieren ggf. weitere Bausteine wie `AUI_TO_AI` (Rückkonvertierung) oder Konverter für andere Datentypen (z. B. `REAL_TO_INT`). `AI_TO_AUI` ist speziell auf die Kombination von INT- und UINT-Adapter zugeschnitten.

Fazit

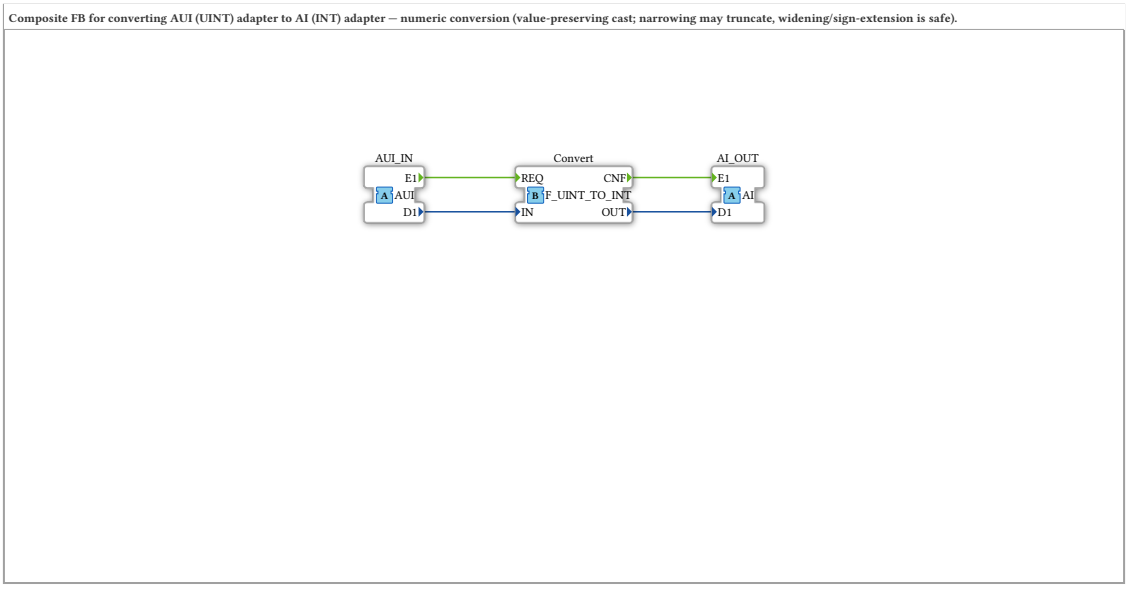
Der Baustein `AI_TO_AUI` bietet eine kompakte, wiederverwendbare Lösung zur Umwandlung eines INT-Adapter-Datenstroms in einen UINT-Adapter-Datenstrom. Er reduziert die Komplexität der Verschaltung und stellt sicher, dass die Datenkonvertierung korrekt und ereignisgesteuert erfolgt. Dank der Adapter-Schnittstellen fügt er sich nahtlos in modulare, unidirektionale Architekturen ein.

AUI_TO_AI

UINT Adapter Input – *AUI*



AI – INT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Composite FB zur Umwandlung eines AUI- (UINT) Adapters in einen AI- (INT) Adapter. Er kapselt die Konvertierung mittels des Bausteins `F_UINT_TO_INT` und stellt eine einheitliche Adapter-Schnittstelle bereit.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge auf FB-Ebene. Das Ereignis wird über den **Socket AUI_IN** (Ereignis `E1`) empfangen.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge auf FB-Ebene. Das Ereignis wird über den **Plug AI_OUT** (Ereignis `E1`) ausgegeben.

Daten-Eingänge

Keine direkten Daten-Eingänge auf FB-Ebene. Der UINT-Wert wird über den **Socket AUI_IN** (Datenpunkt `D1`) bereitgestellt.

Daten-Ausgänge

Keine direkten Daten-Ausgänge auf FB-Ebene. Der INT-Wert wird über den **Plug AI_OUT** (Datenpunkt `D1`) ausgegeben.

Adapter

- **Socket AUI_IN** (Typ: `adapter::types::unidirectional::AUI`) – Empfängt UINT-Daten und ein Ereignis.
- **Plug AI_OUT** (Typ: `adapter::types::unidirectional::AI`) – Sendet INT-Daten und ein Ereignis.

Funktionsweise

Der Baustein verbindet den AUI-Adapter-Socket über ein FBNetwork mit einem Konvertierungs-FB vom Typ `F_UINT_TO_INT`:

1. Das Ereignis `E1` des AUI_IN-Adapters triggert den Konvertierungs-FB am **REQ**-Eingang.
2. Der Konvertierungs-FB liest den UINT-Wert vom Datenpunkt `AUI_IN.D1` und konvertiert ihn nach INT.
3. Nach der Konvertierung sendet der FB ein Bestätigungsereignis über **CNF**, das das Ereignis `E1` des AI_OUT-Adapters auslöst.
4. Der konvertierte INT-Wert wird über den Datenpunkt `AI_OUT.D1` an den AI-Adapter übergeben.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- Composite-FB ohne eigene Zustandslogik (kein ECC).
- Verwendet den Standard-Baustein `iec61131::conversion::F_UINT_TO_INT` für die Typkonvertierung.
- Enthält Lizenzhinweise gemäß Eclipse Public License 2.0.

Zustandsübersicht

Der Baustein besitzt keinen internen Zustandsautomaten. Die Steuerung erfolgt rein durch die Ereignisverkettung im FBNetwork.

Anwendungsszenarien

- Integration einer Komponente, die einen AUI-Adapter (UINT) liefert, in ein System, das einen AI-Adapter (INT) erwartet.
- Kapselung der Konvertierung, um die Wiederverwendung zu erleichtern und die Netzwerkstruktur übersichtlich zu halten.

Vergleich mit ähnlichen Bausteinen

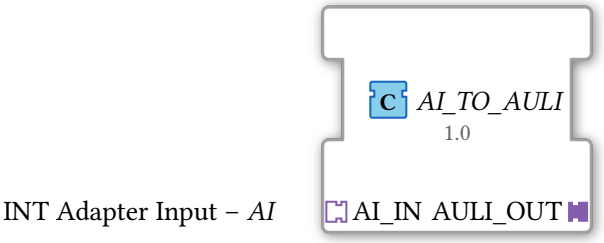
- **AUI_TO_AI** wandelt UINT → INT.
- Ein analoger Baustein **AI_TO_AUI** würde die umgekehrte Konvertierung (INT → UINT) bereitstellen.
- Im Gegensatz zu direkten Typkonvertierungs-FBs (wie `F_UINT_TO_INT`) arbeitet dieser Baustein auf Adapterebene und fügt sich nahtlos in IEC 61499-Adapterkonzepte ein.

Fazit

Der **AUI_TO_AI**-Baustein bietet eine einfache und gekapselte Lösung, um AUI-Adapter in AI-Adapter zu konvertieren. Er erleichtert die Interoperabilität zwischen Komponenten mit unterschiedlichen Datentypen und trägt zu einer sauberen, modularen Systemarchitektur bei.

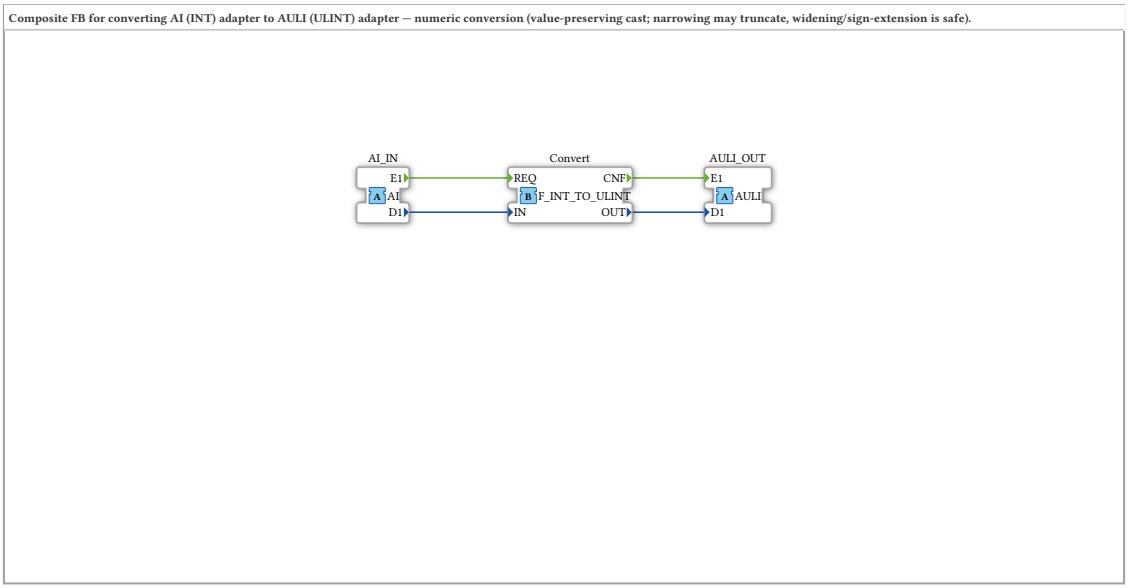
AI_AULI

AI_TO_AULI



INT Adapter Input – *AI*

AULI – ULINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AI_TO_AULI** dient als Composite-Baustein zur Konvertierung eines unidirektionalen Adapters vom Typ `AI` (basierend auf dem Datentyp `INT`) in einen unidirektionalen Adapter vom Typ `AULI` (basierend auf `ULINT`). Er ermöglicht die nahtlose Integration von Komponenten, die unterschiedliche ganzzahlige Datentypen verwenden, ohne dass der Anwender die Konvertierungslogik selbst implementieren muss.

Schnittstellenstruktur

Der Baustein besitzt keine klassischen Ereignis- oder Datenein-/ausgänge, sondern ausschließlich Adapter-Schnittstellen (Socket & Plug), über die sowohl Ereignisse als auch Daten übertragen werden.

Ereignis-Eingänge

Der Ereigniseingang wird über den Socket-Adapter `AI_IN` bereitgestellt.

Name	Adapter	Kommentar
`E1`	`AI_IN`	Ereignis, das die Konvertierung auslöst

Ereignis-Ausgänge

Der Ereignisausgang wird über den Plug-Adapter `AULI_OUT` bereitgestellt.

Name	Adapter	Kommentar
E1	AULI_OUT	Ereignis nach erfolgreicher Konvertierung

Daten-Eingänge

Der Dateneingang wird über den Socket-Adapter AI_IN bereitgestellt.

Name	Adapter	Typ	Kommentar
D1	AI_IN	INT	Eingangsdaten (16-Bit Integer)

Daten-Ausgänge

Der Datenausgang wird über den Plug-Adapter AULI_OUT bereitgestellt.

Name	Adapter	Typ	Kommentar
D1	AULI_OUT	ULINT	Ausgangsdaten (64-Bit vorzeichenloser Integer)

Adapter

- **Socket** AI_IN : Unidirektionaler Adapter mit den Schnittstellen E1 (Ereignis) und D1 (INT).
- **Plug** AULI_OUT : Unidirektionaler Adapter mit den Schnittstellen E1 (Ereignis) und D1 (ULINT).

Funktionsweise

Der Baustein realisiert eine einfache Typkonvertierung von INT nach ULINT. Intern wird der IEC 61131-Baustein F_INT_TO_ULINT verwendet. Die Ablaufsteuerung erfolgt ereignisgesteuert:

1. Ein eingehendes Ereignis am Socket AI_IN.E1 triggert den Eingang REQ des Konvertierungsbausteins.
2. Der Datenwert AI_IN.D1 (INT) wird an den Eingang IN des Konvertierungsbausteins übergeben.
3. Die Konvertierung wird ausgeführt und das Ergebnis (ULINT) am Ausgang OUT bereitgestellt.
4. Nach Abschluss der Konvertierung wird ein Ereignis am Ausgang CNF des Konvertierungsbausteins generiert, das an den Plug AULI_OUT.E1 weitergeleitet wird.
5. Parallel dazu wird der konvertierte Wert an den Plug AULI_OUT.D1 gesendet.

Dadurch wird eine vollständige, synchronisierte Übertragung von Ereignis und Daten gewährleistet.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Adapterbasierte Kommunikation:** Der Baustein kapselt die Konvertierungslogik in einem Composite, sodass die angeschlossenen Komponenten nur die jeweiligen Adapter-Schnittstellen kennen müssen.
- **Unidirektionaler Datentransfer:** Sowohl AI als auch AULI sind als unidirektionale Adapter ausgelegt; eine Rückrichtung ist nicht vorgesehen.
- **Typkonvertierung:** Der interne Baustein F_INT_TO_ULINT führt eine erweiternde Umwandlung von 16-Bit Integer (INT) zu 64-Bit vorzeichenlosem Integer (ULINT) durch. Negative INT-Werte werden dabei gemäß dem Zweierkomplement in große positive ULINT-Werte umgewandelt (z. B. -1 ⟶ 2^64-1).
- **Keine Eigenzustände:** Der Baustein besitzt keine eigene Zustandsmaschine, sondern leitet Ereignisse und Daten direkt weiter.

Zustandsübersicht

Da der Baustein als reines Composite ohne eigene zustandsbehaftete Logik realisiert ist, existiert keine explizite Zustandsmaschine. Das Verhalten wird allein durch die Kopplung der internen Bausteine bestimmt. Die Funktionalität ist rein kombinatorisch und ereignisgetriggert.

Anwendungsszenarien

- **Adapter-Wandler:** Einbinden von Sensoren oder Aktoren, die als AI-Adapter (INT) bereitgestellt werden, in ein System, das AULI-Adapter (ULINT) erwartet.

- **Systemintegration:** Verwenden in SPS- oder Automatisierungsarchitekturen, in denen unterschiedliche Datentypen normiert werden müssen.
- **Protokollanpassung:** Konvertieren von 16-Bit Zählerwerten in 64-Bit Zählerwerte für erweiterte Reichweite.

Vergleich mit ähnlichen Bausteinen

Baustein	Quelltyp	Zieltyp	Adapterbasiert	Eigene Zustände
`F_INT_TO_ULINT`	`INT`	`ULINT`	Nein (reine FB)	Nein
`AI_TO_AULI`	`AI`	`AULI`	Ja (Adapter)	Nein
`ANY_TO_ANY`	variabel	variabel	möglich	je nach Implementierung

Während der Basisbaustein `F_INT_TO_ULINT` nur die Datenkonvertierung vornimmt, kapselt `AI_TO_AULI` diese zusammen mit der Ereignissteuerung in einer adapterkompatiblen Schnittstelle. Dadurch wird die Wiederverwendbarkeit in adapterbasierten Systemen erhöht.

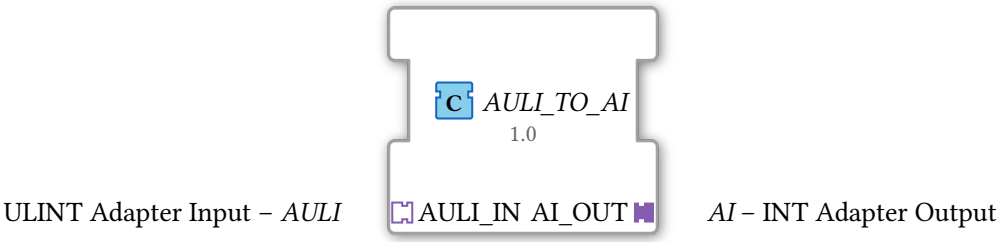
Fazit

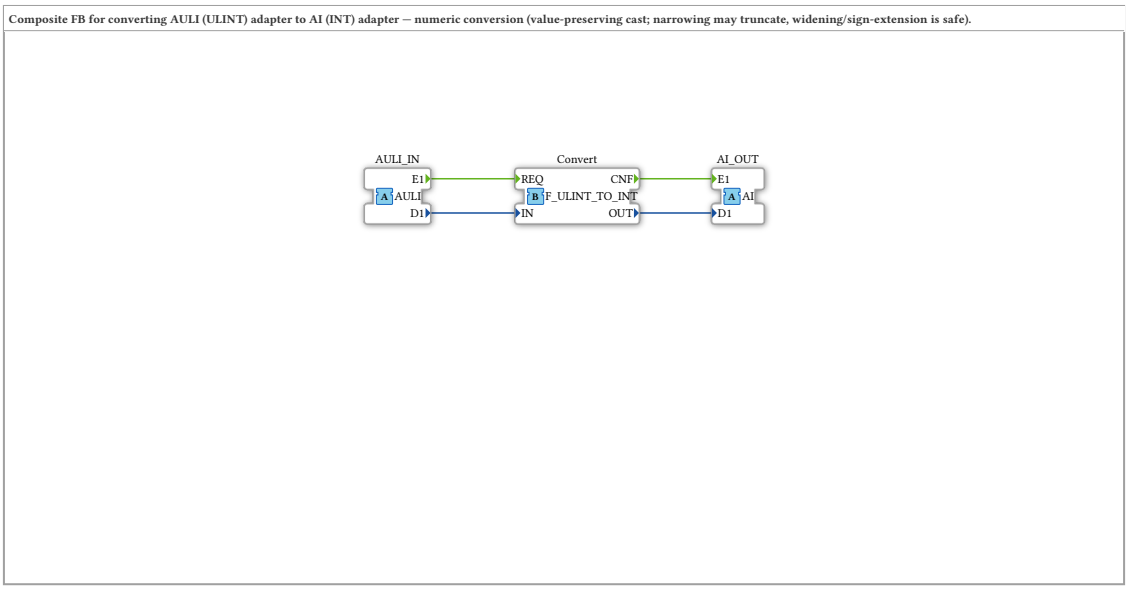
Der Funktionsblock `AI_TO_AULI` bietet eine saubere, adapterbasierte Lösung zur Konvertierung von `INT`- nach `ULINT`-Daten. Er reduziert den manuellen Aufwand und die Fehleranfälligkeit bei der Typanpassung und lässt sich nahtlos in bestehende 4diac-Projekte integrieren. Dank seiner einfachen, ereignisgetriggerten Struktur ist er besonders für unidirektionale Datenflüsse geeignet.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AULI_TO_AI





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AULI_TO_AI** ist ein zusammengesetzter Baustein (Composite FB), der einen eingehenden Adapter vom Typ **AULI** (ULINT-basiert) in einen ausgehenden Adapter vom Typ **AI** (INT-basiert) konvertiert. Er dient als Schnittstellenwandler zwischen Komponenten, die unterschiedliche Datentypen für analoge Werte verwenden.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein selbst besitzt keine direkten Ereignis-Eingänge. Die Ereignissteuerung erfolgt über die angeschlossenen Adapter.

Ereignis-Ausgänge

Auch auf dieser Ebene sind keine dedizierten Ereignis-Ausgänge vorhanden; die Ausgabeereignisse werden über den AI_OUT-Adapter weitergegeben.

Daten-Eingänge

- Keine direkten Dateneingänge auf Blockebene. Die Daten gelangen über den Socket `AULI_IN` in den Baustein.

Daten-Ausgänge

- Keine direkten Datenausgänge auf Blockebene. Die konvertierten Daten werden über den Plug `AI_OUT` ausgegeben.

Adapter

Name	Richtung	Typ	Beschreibung
`AULI_IN`	Socket	`adapter::types::unidirectional::AULI`	ULINT-Eingangsadapter (analoger Wert)
`AI_OUT`	Plug	`adapter::types::unidirectional::AI`	INT-Ausgangsadapter (konvertierter Wert)

Funktionsweise

Der Baustein verwendet intern den vordefinierten Funktionsblock `F_ULINT_TO_INT` aus der IEC 61131-Bibliothek. Die Funktionsweise ist:

1. Ein Ereignis vom Socket `AULI_IN.E1` triggert die Konvertierung des Datenwerts `AULI_IN.D1` (ULINT) über den internen FB `Convert`.
2. Der FB `Convert` führt die Typumwandlung von `ULINT` nach `INT` aus.
3. Nach erfolgreicher Konvertierung wird das Ausgangsereignis `Convert.CNF` ausgelöst, welches das Ereignis `AI_OUT.E1` aktiviert.
4. Der konvertierte Datenwert (`Convert.OUT`) wird an `AI_OUT.D1` weitergeleitet.

Damit wird eine nahtlose, ereignisgesteuerte Umwandlung von AULI- auf AI-Adapter realisiert.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Komposition:** Der Baustein ist als Composite FB realisiert, d.h. er kapselt die Logik in einem übersichtlichen Baustein und nutzt einen standardisierten Konvertierungsbaustein.
- **Unidirektionale Adapter:** Sowohl `AULI_IN` als auch `AI_OUT` sind unidirektionale Adapter, die nur eine Datenflussrichtung unterstützen.
- **Ereignisgesteuert:** Der Konvertierungsprozess startet nur bei einem eingehenden Ereignis, was eine effiziente Verarbeitung ermöglicht.
- **Lizenzhinweis:** Der Baustein unterliegt der Eclipse Public License 2.0.

Zustandsübersicht

Da der Baustein rein ereignisgesteuert arbeitet und keine internen Zustände speichert, gibt es keine explizite Zustandsmaschine. Er verhält sich wie eine funktionale Transformation: Auf jedes Ereignis am Eingang erfolgt die Konvertierung und ein Ausgangsereignis.

Interner Ablauf (vereinfacht):

- **Idle:** Warten auf Ereignis von `AULI_IN`.
- **Processing:** Konvertierung läuft (instantan, da synchroner FB).
- **Done:** Ausgangsereignis wird gesendet, Rückkehr zu Idle.

Anwendungsszenarien

- **Systemintegration:** Verbindung von Komponenten, die unterschiedliche Datentypen für analoge Signale verwenden (z.B. ein Sensor liefert ULINT, ein Aktuator erwartet INT).
- **Adapter-Kaskaden:** Erweiterung von Protokollumsetzungen in Automatisierungsprojekten.
- **Datentypanpassung:** Wenn ein Feldbus- oder Gateway-Modul nur ULINT-Werte liefert, das Steuerungssystem jedoch INT benötigt.

Vergleich mit ähnlichen Bausteinen

Baustein	Eingabetyp	Ausgabetyp	Beschreibung
`AULI_TO_AI`	AULI (ULINT)	AI (INT)	Konvertiert ganze Adapter-Schnittstellen.
`F_ULINT_TO_INT`	ULINT	INT	Nur Datenkonvertierung, keine Adapter.
`AI_TO_AULI` (ggf.)	AI (INT)	AULI (ULINT)	Umgekehrte Richtung (nicht vorhanden).

Der wesentliche Unterschied zu reinen Datenkonvertierungsbausteinen liegt in der Adapter-Ein-/Ausgabe, die eine vollständige Schnittstellenumsetzung ermöglicht.

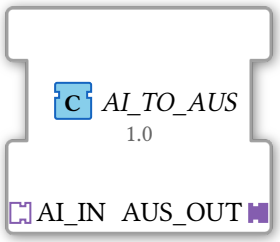
Fazit

Der Baustein `AULI_TO_AI` bietet eine elegante und standardisierte Möglichkeit, adapterspezifische Typkonvertierungen zwischen ULINT- und INT-basierten analogen Schnittstellen durchzuführen. Durch die Kapselung der Konvertierungslogik in einem Composite FB wird die Wiederverwendbarkeit und Übersichtlichkeit erhöht. Er eignet sich besonders für modulare Automatisierungsarchitekturen, bei denen unterschiedliche Protokolle oder Datentypen aufeinandertreffen.

AI_AUS

AI_TO_AUS

INT Adapter Input – *AI*



AUS – USINT Adapter Output

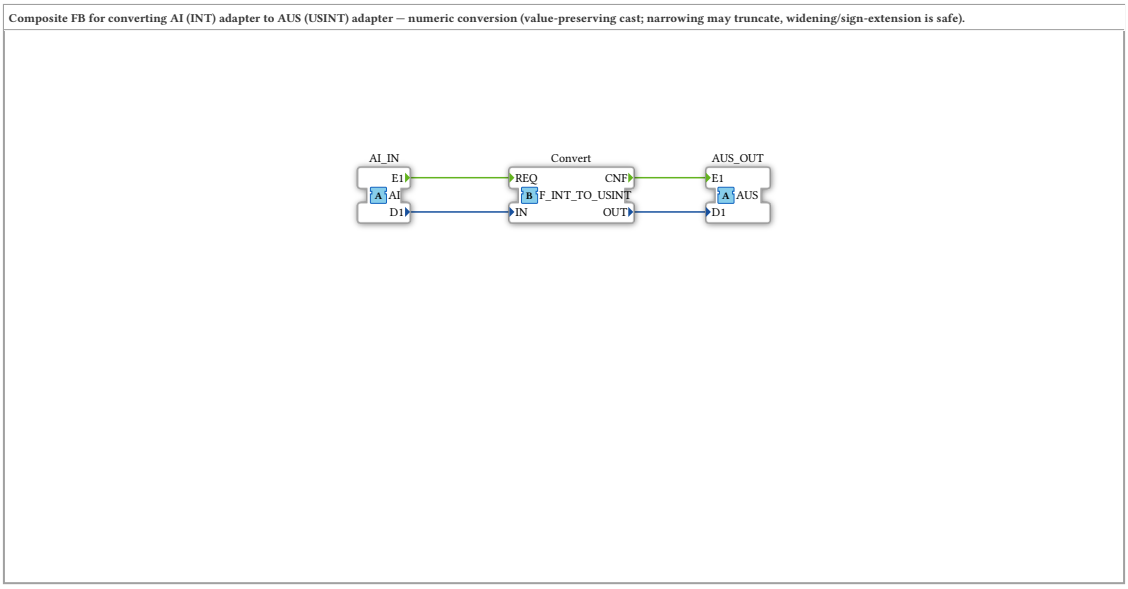


Bild des Funktionsblocks – nicht in der XML enthalten, daher kein automatisch generiertes Bild vorhanden.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AI_TO_AUS** ist ein Composite-Baustein, der einen **AI-Adapter** (Datenstrom vom Typ `INT`) in einen **AUS-Adapter** (Datenstrom vom Typ `USINT`) umwandelt. Er dient als Brücke zwischen Komponenten, die unterschiedliche Datentypen für analoge Ein- und Ausgangssignale verwenden. Die Implementierung nutzt den internen Baustein `F_INT_TO_USINT` aus der Bibliothek `iec61131::conversion`.

Schnittstellenstruktur

Der Baustein besitzt keine eigenen separaten Ereignis- oder Datenports, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- Keine direkten Ereigniseingänge. Die Ereignissteuerung erfolgt indirekt über den Adapter `AI_IN` (Socket) – hierüber wird das Ereignis `E1` empfangen.

Ereignis-Ausgänge

- Keine direkten Ereignisausgänge. Das konvertierte Ereignis wird über den Adapter `AUS_OUT` (Plug) als `E1` ausgegeben.

Daten-Eingänge

- Keine direkten Dateneingänge. Der zu konvertierende Datenwert (`INT`) wird über den Adapter `AI_IN` (Socket) als `D1` bezogen.

Daten-Ausgänge

- Keine direkten Datenausgänge. Der konvertierte Wert (`USINT`) wird über den Adapter `AUS_OUT` (Plug) als `D1` bereitgestellt.

Adapter

- **`AI_IN`** (Socket) Typ: `adapter::types::unidirectional::AI` Beschreibung: Nimmt einen `INT`-Wert und ein zugehöriges Ereignis auf.
- **`AUS_OUT`** (Plug) Typ: `adapter::types::unidirectional::AUS` Beschreibung: Gibt den konvertierten `USINT`-Wert mit einem Ereignis aus.

Funktionsweise

1. Ein eingehendes Ereignis am Adapter `AI_IN.E1` triggert den internen Funktionsblock `Convert` (`F_INT_TO_USINT`) über dessen Ereigniseingang `REQ`.

2. Gleichzeitig wird der Datenwert `AI_IN.D1` (Typ `INT`) an den Eingang `IN` des Konvertierungsbausteins übergeben.
3. Der Baustein `F_INT_TO_USINT` führt die Typkonvertierung von `INT` nach `USINT` durch.
4. Nach Abschluss der Konvertierung sendet `Convert` ein Bestätigungsereignis (`CNF`) an den Ausgangsadapter, und der konvertierte Wert (`OUT`) wird an `AUS_OUT.D1` weitergeleitet.
5. Das Ereignis `AUS_OUT.E1` signalisiert der nachfolgenden Komponente, dass ein neuer Datenwert am Adapter anliegt.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Verwendete Bibliothek:** Der Baustein greift auf den IEC-61131-Standardbaustein `F_INT_TO_USINT` zurück, der eine typische Konvertierung mit möglicher Bereichsbegrenzung oder Überlaufverhalten implementiert.
- **Adapter-Kapselung:** Der Composite-Baustein abstrahiert die Konvertierung auf Adapterebene, sodass in der Netzwerkkonfiguration nur die beiden passenden Adapter verbunden werden müssen.
- **Datenbereichskonflikt:** `INT` umfasst Werte von −32.768 bis +32.767, während `USINT` nur 0…255 darstellen kann. Bei Eingabewerten außerhalb des USINT-Bereichs hängt das Verhalten vom `F_INT_TO_USINT` ab (z. B. Sättigung auf 0 oder 255, oder modularer Überlauf). Die genaue Abbildungsvorschrift ist im konkreten Zielsystem zu prüfen.

Zustandsübersicht

Da der Baustein ausschließlich aus einer sequenziellen Datenverarbeitung besteht (wiederholte Konvertierung bei jedem Ereignis), besitzt er keine eigenen Zustandsautomaten. Die Zustände des internen Bausteins `F_INT_TO_USINT` sind für die Funktion nicht sichtbar; der Composite-Baustein arbeitet ereignisgesteuert und ohne interne Zustandsspeicherung.

Anwendungsszenarien

- **Sensor-Aktor-Kopplung:** Ein Sensor liefert Messwerte als `INT` (z. B. Temperatur −10…50 °C skaliert auf Ganzzahlen), während ein nachgeschalteter Aktor eine Steuergröße als `USINT` (z. B. 0…100 %) erwartet. Der Baustein übernimmt die erforderliche Typumwandlung.
- **Adapter-Kompatibilisierung:** In einer 4diac-Applikation sind zwei Module über unterschiedliche Adapter (`AI` und `AUS`) definiert. Der Composite-FB ermöglicht die direkte Verbindung ohne zusätzliche Konvertierungslogik im Hauptnetzwerk.

Vergleich mit ähnlichen Bausteinen

- **Direkte Konverter (`F_INT_TO_USINT`):** Stellt die reine Typkonvertierung auf Datenebene bereit, erfordert aber separate Ereignis- und Datenverbindungen. `AI_TO_AUS` erweitert diesen um die Adapter-Anbindung und vereinfacht so den Netzwerkentwurf.
- **Andere Adapter-Konverter (`AI_TO_...`, `..._TO_AUS`):** Je nach benötigten Adaptertypen sind entsprechende Composite-Bausteine denkbar. `AI_TO_AUS` deckt die spezifische Kombination `INT` → `USINT` ab.

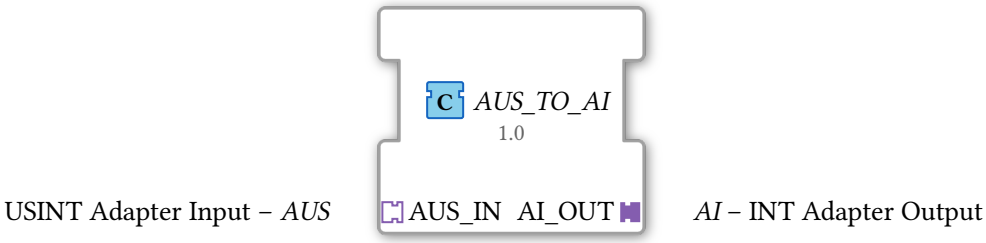
Fazit

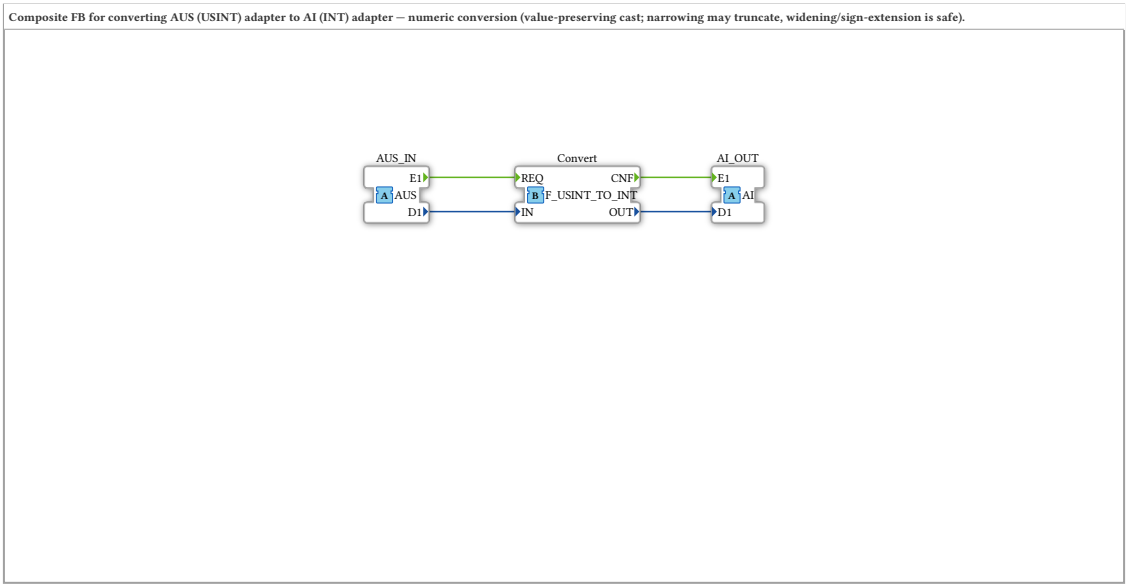
Der Funktionsblock **AI_TO_AUS** ist ein kompakter Composite-Baustein zur einfachen Adapter-Konvertierung von `INT` nach `USINT`. Er kapselt den IEC-61131-Konverter und stellt die Daten- und Ereignisflüsse über standardisierte Adapter bereit. Dadurch wird die Integration von Komponenten mit unterschiedlichen analogen Schnittstellen in 4diac-Applikationen erheblich erleichtert. Bei der Verwendung sollte der mögliche Datentypüberlauf beachtet werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUS_TO_AI





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `AUS_TO_AI` ist ein Composite-Baustein, der einen Adapter des Typs `AUS` (USINT-basiert) in einen Adapter des Typs `AI` (INT-basiert) umwandelt. Er dient als unidirektionale Schnittstelle zwischen Systemteilen, die unterschiedliche Datentypen für analoge Eingänge verwenden. Der Baustein kapselt die Konvertierungslogik und erleichtert die Integration von Komponenten, die auf `USINT`-Adaptern basieren, in Umgebungen, die `INT`-Adapter erwarten.

Schnittstellenstruktur

Der FB besitzt **keine direkten Ereignis- oder Dateneingänge bzw. -ausgänge**. Die gesamte Signalkommunikation erfolgt über die beiden eingebauten Adapter:

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Adapter	Richtung	Typ	Beschreibung

AUS_IN	Socket	adapter::types::unidirectional::AUS	USINT-basierter Adaptereingang (Quelle des zu konvertierenden Werts)
AI_OUT	Plug	adapter::types::unidirectional::AI	INT-basierter Adapterausgang (Ziel des konvertierten Werts)

Der Socket AUS_IN empfängt ein Ereignissignal und einen USINT-Datenwert. Der Plug AI_OUT gibt nach erfolgter Konvertierung ein Ereignissignal und den dazugehörigen INT-Wert aus.

Funktionsweise

Der Baustein arbeitet rein kombinatorisch auf Basis einer internen Instanz des IEC-61131-Funktionsbausteins F_USINT_TO_INT. Die Ablaufsteuerung wird über die Ereignisverbindungen der Adapter realisiert:

1. **Ereignis E1 am AUS_IN-Adapter** wird auf den **REQ-Eingang** des Konvertierungsbausteins Convert geleitet.
2. Der Konvertierungsbaustein führt die Wandlung des an seinem **IN**-Eingang anliegenden USINT-Werts in einen INT-Wert durch und signalisiert die Fertigstellung mit dem Ereignis **CNF**.
3. Das **CNF**-Ereignis wird an den **E1-Eingang des AI_OUT-Adapter**s weitergegeben, gleichzeitig wird der konvertierte INT-Wert an den **D1-Datenausgang** des AI_OUT-Adapter gelegt.

Somit entsteht eine kausale Kette: Ein eingehendes Ereignis am AUS_IN-Adapter löst die Umwandlung aus und führt zur Ausgabe eines entsprechenden Ereignisses und des konvertierten Datenwerts am AI_OUT-Adapter.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Reine Adapterkopplung** – Der Baustein besitzt keine eigenen Ein-/Ausgangsvariablen, sondern definiert nur Adapter-Schnittstellen (Socket/Plug). Dies folgt dem Konzept der adapterbasierten Kommunikation in IEC 61499.
- **Unidirektionale Konvertierung** – Die Richtung ist festgelegt: von USINT nach INT. Eine Rückwärtskonvertierung ist nicht vorgesehen.
- **Nutzung eines Standard-IEC-61131-Bausteins** – Die eigentliche Typkonvertierung erfolgt durch den etablierten FB F_USINT_TO_INT, der bereits in vielen Laufzeitumgebungen verfügbar ist.
- **Kein interner Zustand** – Der Composite-Baustein ist zustandslos; die Konvertierung wird bei jedem eingehenden Ereignis neu ausgelöst.

Zustandsübersicht

Da der FB keine eigene Zustandsmaschine besitzt, existieren nur die impliziten Zustände des eingebetteten F_USINT_TO_INT. In der Praxis verhält sich der Baustein wie eine **kombinatorische Funktion**: Nach dem Eingang eines Ereignisses steht das Ausgangssignal zum nächsten Ausführungszyklus zur Verfügung. Ein definierter Fehlerzustand liegt nicht vor – die Konvertierung erfolgt gemäß der Spezifikation des IEC-Bausteins.

Anwendungsszenarien

- **Migration von USINT-basierter Hardware auf INT-basierte Steuerungen** – Ein System, das ursprünglich mit 8-Bit-Werten (USINT) arbeitet, kann über diesen Adapter an Komponenten angebunden werden, die 16-Bit-Analogwerte (INT) erwarten.
- **Kopplung unterschiedlicher Subsysteme** – In einer modularen Anlage nutzt ein Sensoradapter USINT, während die übergeordnete Steuerungseinheit INT-Schnittstellen bereitstellt.
- **Prototypen und Testumgebungen** – Schnelle Integration von Komponenten mit abweichenden Datentypen ohne separate Konvertierungslogik in der Anwendungsebene.

Vergleich mit ähnlichen Bausteinen

- **Direkte Konvertierungsbausteine** – FBs wie F_USINT_TO_INT oder F_INT_TO_USINT bieten die reine Typumwandlung, erfordern aber eine manuelle Verkabelung von Ereignissen und Daten. AUS_TO_AI kapselt dies in einem Adapter-to-Adapter-Konzept, wodurch die Integration in adapterbasierte Architekturen vereinfacht wird.

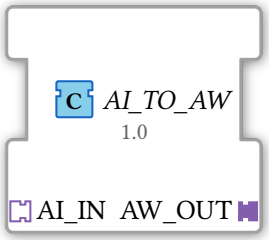
- **Andere Adapter-Converter** – Denkbar sind symmetrische Bausteine wie `AI_TO_AUS` (INT nach USINT) oder bidirektionale Varianten. `AUS_TO_AI` ist spezifisch für die unidirektionale Umwandlung von `AUS` nach `AI` ausgelegt.

Fazit

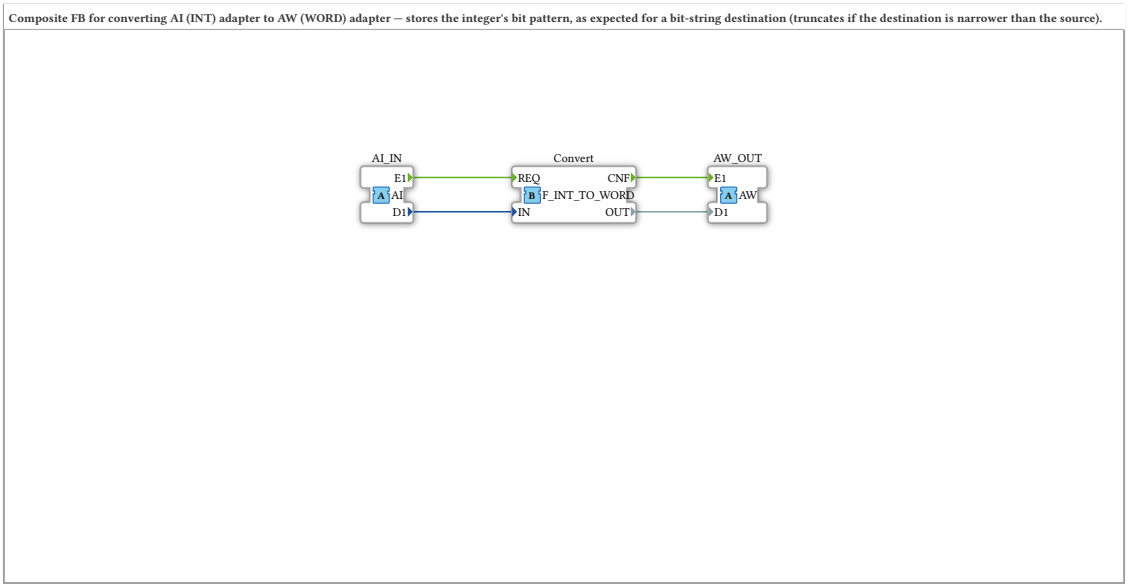
Der Composite-Baustein `AUS_TO_AI` bietet eine saubere, adapterbasierte Lösung zur Konvertierung eines USINT-Werts in einen INT-Wert. Durch die Verkapselung des Standard-Bausteins `F_USINT_TO_INT` und die klare Ereignis-/Datenflusssteuerung eignet er sich hervorragend für die Integration heterogener Komponenten in IEC-61499-Systemen. Der Baustein ist wartungsarm, zustandslos und folgt den etablierten Konventionen der Adapter-Schnittstellen.

AI_AW

AI_TO_AW



INT Adapter Input – *AI* *AW* – WORD Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AI_TO_AW** ist ein Composite-Baustein zur Umsetzung eines INT-Adapters (AI) in einen WORD-Adapter (AW). Er wird verwendet, um die Schnittstelle eines AI-Adapters (Typ INT) auf einen AW-Adapter (Typ WORD) zu konvertieren, damit die Daten und Ereignisse nahtlos zwischen unterschiedlichen Adapter-Typen weitergegeben werden können.

Schnittstellenstruktur

Ereignis-Eingänge

- **AI_IN.E1** (über Socket AI_IN): Ereigniseingang, der den Konvertierungsvorgang auslöst.

Ereignis-Ausgänge

- **AW_OUT.E1** (über Plug AW_OUT): Ereignisausgang, der nach erfolgreicher Konvertierung aktiviert wird.

Daten-Eingänge

- **AI_IN.D1** (über Socket AI_IN): Dateninput vom Typ **INT**, der den zu konvertierenden Wert bereitstellt.

Daten-Ausgänge

- **AW_OUT.D1** (über Plug AW_OUT): Datenoutput vom Typ **WORD**, der das konvertierte Ergebnis liefert.

Adapter

- **AI_IN** (Socket): Erwartet einen Adapter vom Typ `adapter::types::unidirectional::AI` (INT-basiert).
- **AW_OUT** (Plug): Stellt einen Adapter vom Typ `adapter::types::unidirectional::AW` (WORD-basiert) bereit.

Funktionsweise

Der Baustein arbeitet als Kapselung eines einfachen Typkonvertierungs-Funktionsblocks (`F_INT_TO_WORD`) innerhalb einer Composite-Struktur. Der Ablauf ist wie folgt:

1. Ein Ereignis am Eingang **AI_IN.E1** wird an den internen Baustein `F_INT_TO_WORD` weitergeleitet.
2. Gleichzeitig wird der Wert von **AI_IN.D1** (INT) an den Eingang `IN` des Konvertierungsblocks übergeben.
3. Der interne Baustein führt die Konvertierung von INT nach WORD durch und legt das Ergebnis an seinem Ausgang `OUT` an.
4. Nach Abschluss der Konvertierung wird das Ereignis **AW_OUT.E1** gesendet und der konvertierte Wert (WORD) an **AW_OUT.D1** ausgegeben.

Somit wird die gesamte Adapter-Konvertierung durch ein einzelnes Ereignis gesteuert und die Daten werden automatisch umgesetzt.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Composite-Baustein**: Die Logik ist durch einen internen Subbaustein realisiert, was eine klare Trennung von Konvertierungslogik und Schnittstellenadaptern ermöglicht.
- **Wiederverwendbarkeit**: Der Baustein basiert auf dem Standardbaustein `iec61131::conversion::F_INT_TO_WORD` und kann daher in jeder IEC 61499-konformen Umgebung eingesetzt werden.
- **Lizenz**: Der Baustein ist unter der Eclipse Public License 2.0 (EPL-2.0) veröffentlicht.
- **Namensschema**: Der Baustein befindet sich im Paket `adapter::conversion::unidirectional`, was auf seine Funktion als unidirektionale Adapterkonvertierung hinweist.

Zustandsübersicht

Da es sich um einen Composite-Baustein ohne eigene Zustandsmaschine handelt, wird das Zustandsverhalten vollständig durch den internen `F_INT_TO_WORD`-Baustein bestimmt. Im Wesentlichen gibt es zwei Zustände:

- **Idle**: Warten auf ein Ereignis am Eingang **AI_IN.E1**.
- **Converting**: Nach Empfang des Ereignisses wird die Konvertierung ausgeführt. Dabei sind keine Verzögerungen oder Wartezeiten vorgesehen – die Ausgabe erfolgt unmittelbar im selben Zyklus.

Anwendungsszenarien

- **Systemintegration**: Wenn ein Subsystem oder eine Komponente Daten über einen INT-Adapter (AI) bereitstellt, aber die Zielkomponente einen WORD-Adapter (AW) erwartet.
- **Protokollanpassung**: In Anlagensteuerungen, bei denen Sensordaten als INT geliefert werden, aber die Kommunikationsschnittstelle auf WORD-Basis arbeitet.
- **Typkonvertierung in Adapterketten**: Der Baustein kann als Teil einer Adapterkette verwendet werden, um unterschiedliche Adapter-Typen zu verbinden, ohne dass manuelle Konvertierung notwendig ist.

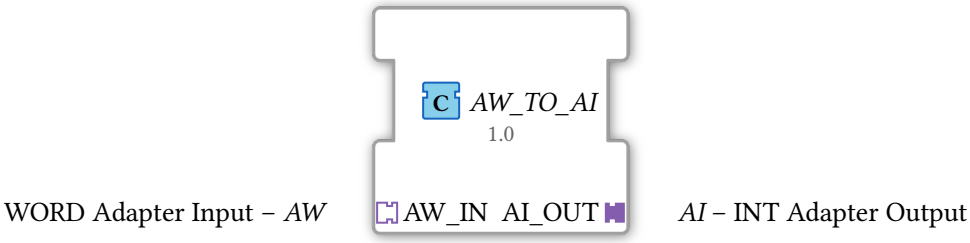
Vergleich mit ähnlichen Bausteinen

Es existieren weitere Adapter-Konvertierungsbausteine wie z.B. `WORD_TO_INT` oder `REAL_TO_DINT` – jedoch fokussiert sich `AI_TO_AW` speziell auf die Adapter-Umwandlung von INT (AI) nach WORD (AW). Im Gegensatz zu einfachen Typkonvertierungsbausteinen (z.B. `F_INT_TO_WORD`) integriert dieser Baustein die Adapterschnittstellen direkt, sodass der Entwickler keine separate Adapter-Instanzierung vornehmen muss.

Fazit

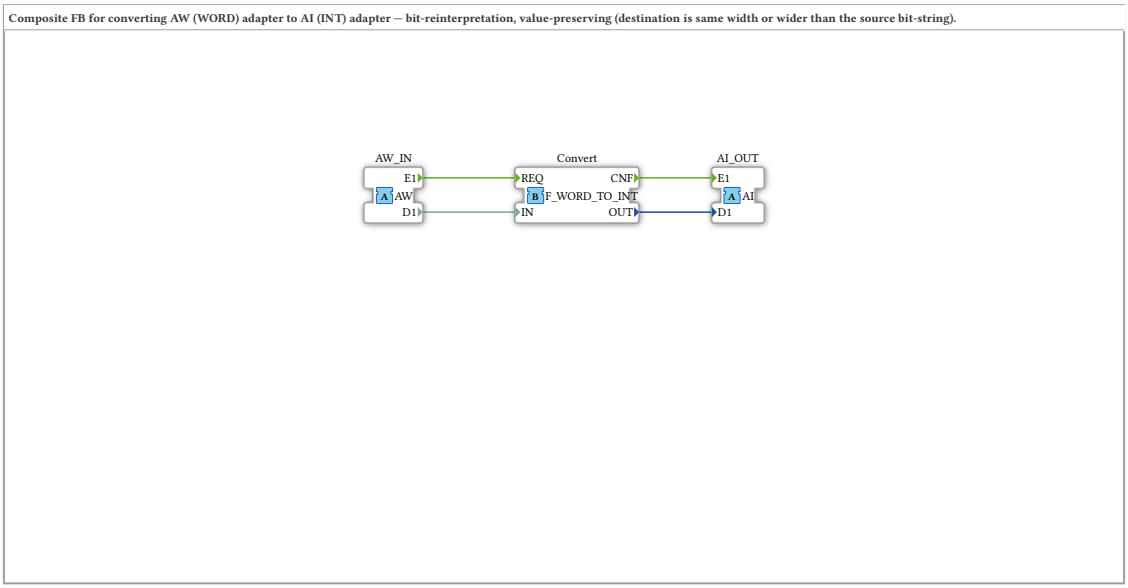
Der **AI_TO_AW**-Funktionsblock bietet eine einfache und robuste Lösung zur Konvertierung eines INT-Adapters in einen WORD-Adapter. Durch die Composite-Struktur bleibt die Logik überschaubar und erweiterbar. Der Baustein eignet sich besonders für Szenarien, in denen unterschiedliche Adapter-Typen in einer IEC 61499-Applikation harmonisiert werden müssen, und reduziert den Implementierungsaufwand für derartige Konvertierungen erheblich.

AW_TO_AI



WORD Adapter Input – *AW*

AI – INT Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AW_TO_AI** ist ein Composite FB, der einen unidirektionalen **WORD-Adapter (AW)** in einen unidirektionalen **INT-Adapter (AI)** konvertiert. Er kapselt die Typumwandlung von 16-Bit Wort- in Integer-Werte und ermöglicht so die nahtlose Integration von Komponenten, die unterschiedliche Adaptertypen verwenden.

Schnittstellenstruktur

Ereignis-Eingänge

Über den angeschlossenen AW-Socket werden folgende Ereignisse empfangen:

Name	Beschreibung

E1 (AW_IN.E1)	Trigger für die Konvertierung des anliegenden WORD-Wertes

Ereignis-Ausgänge

Über den AI-Plug werden folgende Ereignisse gesendet:

Name	Beschreibung
E1 (AI_OUT.E1)	Bestätigt die erfolgreiche Konvertierung und stellt den INT-Wert bereit

Daten-Eingänge

Eingang über den AW-Socket:

Name	Datentyp	Beschreibung
D1 (AW_IN.D1)	WORD	Der zu konvertierende 16-Bit Wortwert

Daten-Ausgänge

Ausgang über den AI-Plug:

Name	Datentyp	Beschreibung
D1 (AI_OUT.D1)	INT	Der konvertierte Integer-Wert (16-Bit vorzeichenbehaftet)

Adapter

Richtung	Adapter	Typ	Beschreibung
Socket (Eingang)	AW_IN	`adapter::types::unidirectional::AW`	Empfängt WORD-Werte und zugehörige Ereignisse
Plug (Ausgang)	AI_OUT	`adapter::types::unidirectional::AI`	Sendet INT-Werte und zugehörige Ereignisse

Funktionsweise

Der FB enthält eine Instanz des Standard-Konvertierungsbausteins `F_WORD_TO_INT` aus der IEC-61131-Bibliothek. Das interne Netzwerk ist wie folgt verdrahtet:

1. Ein eingehendes Ereignis **AW_IN.E1** triggert den Eingang `REQ` des Konvertierungsbausteins.
2. Gleichzeitig wird der Datenwert **AW_IN.D1** auf den Eingang `IN` des Bausteins gelegt.
3. Der Baustein wandelt den WORD-Wert in einen INT-Wert um und gibt ihn an seinem Ausgang `OUT` aus.
4. Das vom Baustein erzeugte Bestätigungsereignis `CNF` wird direkt an **AI_OUT.E1** weitergeleitet.
5. Der konvertierte Integer-Wert (Ausgang `OUT`) wird auf **AI_OUT.D1** geschrieben.

Die gesamte Verarbeitung erfolgt streng ereignisgesteuert: Nur bei einem neuen Ereignis am Eingang findet eine Umwandlung statt.

Technische Besonderheiten

- **Bit-Reinterpretation, werterhaltend** (Zielbreite ist gleich groß oder größer als der Quell-Bit-String).
- **Reine Datenkonvertierung:** Es werden keine Zustände gespeichert oder zusätzliche Logik ausgeführt.
- **Standardbibliothek:** Der verwendete `F_WORD_TO_INT` ist ein etablierter Konvertierungsbaustein und stellt eine korrekte Abbildung des WORD-Wertes (0…65535) auf den INT-Wert (-32768…32767) sicher.
- **Unidirektional:** Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional ausgelegt, d.h. sie übertragen Ereignis- und Datenflüsse nur in eine Richtung (Eingang → Ausgang).
- **Kein Datenverlust durch Typanpassung:** Die Bitbreite bleibt identisch (16 Bit), lediglich die Interpretation als vorzeichenlos (WORD) bzw. vorzeichenbehaftet (INT) ändert sich.

Zustandsübersicht

Der FB ist ein **Composite FB** und besitzt keine eigene Zustandsmaschine (ECC). Das Verhalten wird vollständig durch die Verdrahtung des internen Bausteins bestimmt. Es existieren nur die impliziten Zustände:

- **Idle:** Warten auf ein Ereignis am Eingang.
- **Konvertierung aktiv:** Nach Eingang von `AW_IN.E1` bis zum Eintreffen von `Convert.CNF`.

Anwendungsszenarien

- **Adapter-Konvertierung in heterogenen Systemen:** Ein Steuerungsmodul arbeitet mit INT-Werten, eine angeschlossene Sensorik liefert jedoch WORD-Daten über einen AW-Adapter.
- **Bibliothekseinbindung:** Vorhandene Funktionsblöcke, die nur einen AI-Adapter akzeptieren, sollen mit Komponenten verbunden werden, die einen AW-Adapter bereitstellen.
- **Prototypsysteme:** Schnelle Kopplung verschiedener Bausteine während der Entwicklung, ohne die zugrundeliegenden Adapterdefinitionen zu ändern.

Vergleich mit ähnlichen Bausteinen

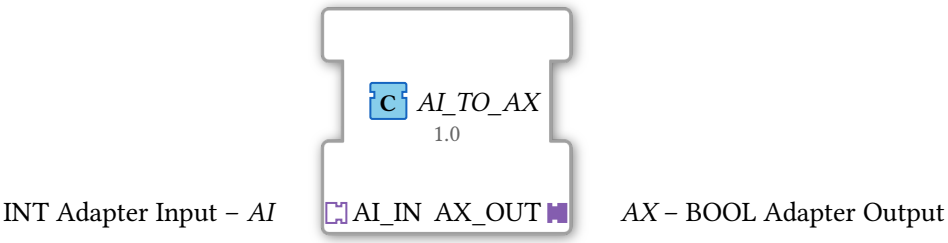
- **AW_TO_AI** konvertiert **WORD → INT** (unidirektional).
- Vergleichbare Konverter aus derselben Bibliothek (`iec61131::conversion`) decken andere Typumwandlungen ab, z.B. `F_INT_TO_WORD` oder `F_WORD_TO_DINT`. Diese sind als traditionelle Funktionsblöcke ohne Adapter verfügbar.
- Falls eine bidirektionale Konversion oder andere Adapter-Typen benötigt werden, sind ähnliche Composite-Bausteine mit entsprechenden Umwandlungsblöcken denkbar (z.B. `AI_TO_AW`).

Fazit

Der Composite FB **AW_TO_AI** stellt eine einfache und zuverlässige Möglichkeit dar, einen unidirektionalen WORD-Adapter in einen INT-Adapter umzuwandeln. Durch die Verwendung des IEC-61131-Standardbausteins bleibt die Konvertierung transparent und effizient. Der Baustein ist besonders dann nützlich, wenn unterschiedliche Adapter-Typen in einem System harmonisiert werden müssen, ohne die ursprünglichen Komponenten zu verändern.

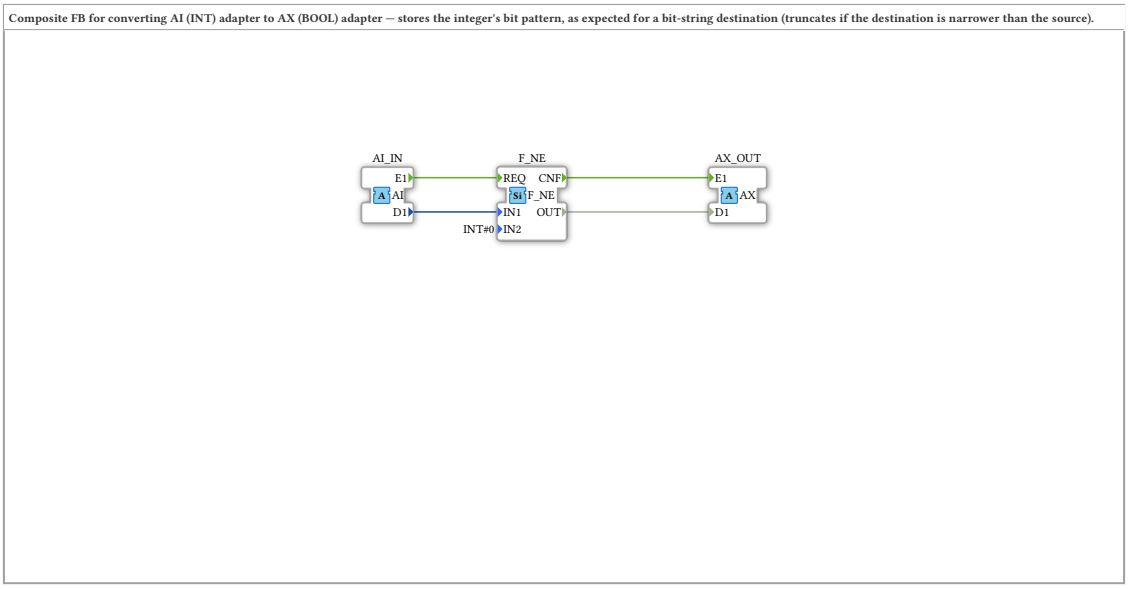
AI_AX

AI_TO_AX



INT Adapter Input – *AI*

AX – BOOL Adapter Output



• — ○ • — ○ • — ○ •

Einleitung

Der Baustein `AI_TO_AX` ist ein Composite-Funktionsblock (Typ-Adapter), der eine unidirektionale Schnittstelle eines INT-Adapters (`AI`) in eine entsprechende BOOL-Schnittstelle eines Adapters (`AX`) umwandelt. Er dient dazu, analoge Integer-Signale (z. B. aus Sensoren) in binäre Boolesche Werte zu überführen, die beispielsweise für Schaltsignale oder Alarmmeldungen genutzt werden können.

Schnittstellenstruktur

Der Baustein besitzt lediglich Adapter-Anschlüsse (Socket und Plug), über die Ereignisse und Daten fließen. Die nachfolgende Tabelle fasst die logischen Ein-/Ausgänge zusammen, die sich aus den verbundenen Adaptern ergeben.

Ereignis-Eingänge

Bezeichnung	Datentyp	Beschreibung
`AI_IN.E1`	EVENT	Eingangsereignis, das die Konvertierung anstößt.

Ereignis-Ausgänge

Bezeichnung	Datentyp	Beschreibung
`AX_OUT.E1`	EVENT	Ausgangsereignis, das nach abgeschlossener Konvertierung gesendet wird.

Daten-Eingänge

Bezeichnung	Datentyp	Beschreibung
`AI_IN.D1`	INT	Analoger Integer-Eingangswert (z. B. 0…32767).

Daten-Ausgänge

Bezeichnung	Datentyp	Beschreibung
`AX_OUT.D1`	BOOL	Boolescher Ausgangswert: `TRUE` wenn der Eingangswert ungleich 0 ist, sonst `FALSE`.

Adapter

Typ	Name	Richtung
AI (Eingangs-Adapter)	AI_IN	Socket
AX (Ausgangs-Adapter)	AX_OUT	Plug

Funktionsweise

Der Baustein nutzt intern den IEC-61131-Vergleichsbaustein F_NE (Not Equal). Sobald am Ereigniseingang AI_IN.E1 ein Ereignis eintrifft, wird F_NE.REQ getriggert. Dieser vergleicht den über AI_IN.D1 anliegenden INT-Wert mit dem konstanten Wert INT#0.

- Ist der Eingangswert **ungleich 0**, so liefert F_NE.OUT den Wert TRUE.
- Ist der Eingangswert **gleich 0**, so liefert F_NE.OUT den Wert FALSE.

Das Ergebnis wird an den Datenausgang AX_OUT.D1 weitergegeben und gleichzeitig das Ereignis AX_OUT.E1 ausgelöst, um nachgelagerte Bausteine zu benachrichtigen.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Composite-Baustein:** Der AI_TO_AX besteht aus verschalteten Grundbausteinen, wodurch er als wiederverwendbare Komponente in verschiedenen Projekten eingesetzt werden kann.
- **Verwendung von IEC-61131-Bausteinen:** Die Vergleichslogik wird durch den standardisierten Baustein F_NE realisiert, was Portabilität und Einhaltung der Norm unterstützt.
- **Kein eigenständiger Zustand:** Der Baustein ist zustandslos – jeder Ereignis-Impuls erzeugt eine aktuelle Berechnung ohne Speicherung von Zwischenwerten.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Endlichen Automaten. Die interne Logik ist rein kombinatorisch: Nach Eintreffen eines Ereignisses werden die Daten sofort verarbeitet und das Ergebnis ausgegeben.

Anwendungsszenarien

- **Analog-Digital-Umwandlung auf Signalebene:** Ein analoger Messwert (z. B. Drucksensor, Temperaturfühler) wird in ein „Ein/Aus“-Signal umgesetzt – z. B. Aktivierung einer Pumpe bei Drucküberschreitung.
- **Schwellwertentscheidung:** Der Baustein kann als einfache Ungleich-Null-Erkennung eingesetzt werden, um z. B. die Anwesenheit eines Signals oder einen aktivierten Zählerstand zu signalisieren.
- **Adapter-Kopplung:** In Systemen, die unterschiedliche Adapterschnittstellen nutzen, ermöglicht AI_TO_AX die nahtlose Verbindung zwischen INT- und BOOL-basierten Komponenten.

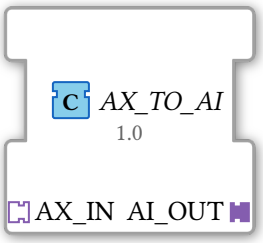
Vergleich mit ähnlichen Bausteinen

- **Direkte Vergleichsbausteine (z. B. F_EQ, F_GT):** Diese liefern Boolesche Ergebnisse für beliebige Vergleiche, benötigen aber zwei Eingänge und sind nicht als Adapter-Konverter ausgelegt.
- **Eigene Adapter-Wrapper:** Entwickler könnten selbst eine Kapselung vornehmen; der AI_TO_AX bietet jedoch eine standardisierte, sofort einsetzbare Lösung mit geringerem Entwicklungsaufwand.
- **Typ-Konverter ohne Ereignissteuerung:** Manche Konverter arbeiten kontinuierlich ohne Ereignis-Auslöser; der AI_TO_AX hingegen reagiert nur auf Ereignisse und ist daher besser in ereignisgesteuerte Architekturen integrierbar.

Fazit

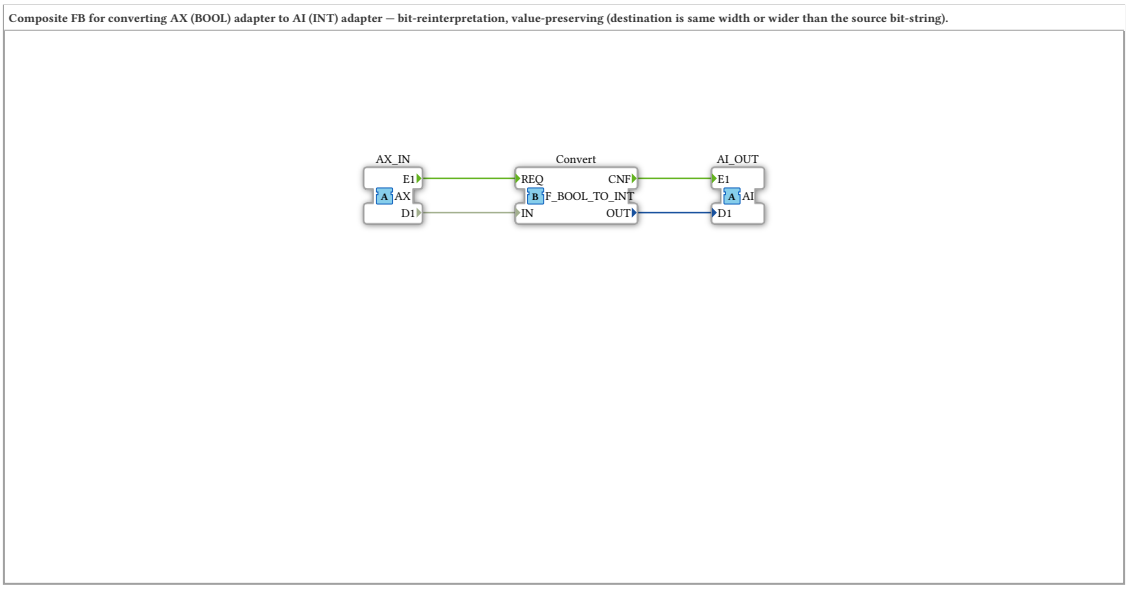
Der AI_TO_AX ist ein praktischer Composite-Baustein, der die Umwandlung von INT-Adapter-Signalen in BOOL-Adapter-Signale vereinfacht. Er eignet sich hervorragend für Anwendungen, in denen analoge Messwerte als binäre Zustände interpretiert werden müssen, und fügt sich dank der ereignisgesteuerten Verarbeitung nahtlos in IEC-61499-basierte Automatisierungssysteme ein.

AX_TO_AI



BOOL Adapter Input – *AX*

AI – INT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AX_TO_AI** ist ein zusammengesetzter Baustein, der einen BOOL-Adapter (AX) in einen INT-Adapter (AI) umwandelt. Er dient als Brücke zwischen Schnittstellen, die boolesche Signale (TRUE / FALSE) verarbeiten, und solchen, die ganzzahlige Werte erwarten. Die Konvertierung erfolgt über einen internen Aufruf des Bausteins `F_BOOL_TO_INT` aus der IEC 61131 Bibliothek.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge. Der Ereignis-Eingang des Bausteins wird über den Socket `AX_IN` (E1) realisiert.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge. Der Ereignis-Ausgang des Bausteins wird über den Plug `AI_OUT` (E1) realisiert.

Daten-Eingänge

Keine direkten Daten-Eingänge. Der Daten-Eingang des Bausteins wird über den Socket `AX_IN` (D1) realisiert (BOOL).

Daten-Ausgänge

Keine direkten Daten-Ausgänge. Der Daten-Ausgang des Bausteins wird über den Plug `AI_OUT` (D1) realisiert (INT).

Adapter

Richtung	Name	Typ (Adapter)	Erläuterung

Socket	AX_IN	AX	Boolescher Eingangsadapter (BOOL)
Plug	AI_OUT	AI	Integer-Ausgangsadapter (INT)

Der Adapter AX stellt einen Ereignis-Eingang E1 und einen Daten-Eingang D1 (BOOL) zur Verfügung. Der Adapter AI stellt einen Ereignis-Ausgang E1 und einen Daten-Ausgang D1 (INT) zur Verfügung.

Funktionsweise

Der Baustein arbeitet als sogenannter *Composite FB*, d.h. er nutzt intern einen anderen Funktionsblock zur Realisierung seiner Logik.

- Am Socket AX_IN wird ein BOOL-Signal über den Daten-Eingang D1 empfangen.
- Ein eingehendes Ereignis an AX_IN.E1 löst den internen Baustein Convert (Typ: F_BOOL_TO_INT) über dessen REQ-Eingang aus.
- Der F_BOOL_TO_INT-Baustein wandelt den BOOL-Wert am Eingang IN in einen INT-Wert (0 für FALSE, 1 für TRUE) um.
- Das Ergebnis wird am Ausgang OUT bereitgestellt und an den Plug AI_OUT.D1 weitergeleitet.
- Nach Abschluss der Umwandlung sendet der interne Baustein ein Bestätigungsereignis (CNF), welches über AI_OUT.E1 als Ausgangsereignis ausgegeben wird.

Die Abarbeitung erfolgt streng sequentiell: erst die boolesche Eingabe verarbeiten, dann die Integer-Ausgabe liefern.

Technische Besonderheiten

- **Bit-Reinterpretation, werterhaltend** (Zielbreite ist gleich groß oder größer als der Quell-Bit-String).
- **Adapterbasierte Kommunikation:** Der Baustein verwendet ausschließlich Adapter-Schnittstellen (Socket und Plug) und keine klassischen Ein-/Ausgangsports. Dies ermöglicht eine lose Kopplung zwischen Komponenten und eine einfache Wiederverwendung in verschiedenen Umgebungen.
- **Unidirektionaler Datenaustausch:** Die verwendeten Adapter AX und AI sind als unidirektionale Typen definiert, d.h. die Daten fließen nur in eine Richtung – vom Socket zum Plug.
- **Externe Bibliothek:** Für die Konvertierung wird der Baustein F_BOOL_TO_INT aus der Bibliothek iec61131::conversion importiert. Dadurch müssen keine eigenen Konvertierungslogiken implementiert werden.
- **Kein eigener Zustandsautomat:** Da es sich um einen Composite-FB handelt, besitzt er kein eigenes ECC (Execution Control Chart). Die Zustandslogik wird vollständig durch den internen Baustein abgebildet.

Zustandsübersicht

Der Baustein verfügt über keinen eigenen Zustandsautomaten. Die Ausführung erfolgt ereignisgesteuert:

1. **Warten:** Am Socket AX_IN wird auf ein Ereignis gewartet.
2. **Konvertieren:** Nach Empfang eines Ereignisses wird der interne FB F_BOOL_TO_INT ausgeführt und das Ergebnis an den Plug AI_OUT weitergegeben.
3. **Abgeschlossen:** Ein Bestätigungsereignis wird über AI_OUT.E1 gesendet; der Baustein kehrt in den Wartezustand zurück.

Dieses Verhalten wird implizit durch die Ereignisverkabelung im Netzwerk gesteuert.

Anwendungsszenarien

- **Signalumsetzung in der Automatisierungstechnik:** Wenn ein Sensor oder eine Steuerung boolesche Werte (z.B. „Tür offen/geschlossen“) liefert, ein nachgeschaltetes Modul aber ganzzahlige Werte (0/1) erwartet, kann dieser Baustein die Anpassung übernehmen.
- **Adapter-Integration:** Er dient als Konverter zwischen verschiedenen Adaptertypen, um Komponenten unterschiedlicher Hersteller oder Bibliotheken miteinander zu verbinden.
- **Protokollanpassung:** In heterogenen Systemen können boolesche Signale mit Hilfe dieses Bausteins in einheitliche Integer-Formate umgewandelt werden.

Vergleich mit ähnlichen Bausteinen

Baustein	Funktion	Unterschied zu AX_TO_AI

`DI_TO_AI`	Wandelt digitalen Input in INT-Adapter um	Benötigt separate Ein-/Ausgangsports; keine Adapter-Schnittstelle
`AX_TO_DINT`	BOOL-Adapter ⟶ DINT-Adapter	Liefert einen 32-Bit-Integer; hier wird 16-Bit INT verwendet
`F_BOOL_TO_INT`	Standard-Konvertierung (BOOL⟶INT)	Kein Adapter-Wrapper; nur als interner Baustein verfügbar

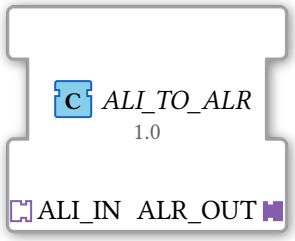
Im Gegensatz zu diesen Alternativen bietet AX_TO_AI eine adressierbare, ereignisgesteuerte Adapter-Schnittstelle und erleichtert die modulare Verschaltung in IEC 61499-Projekten.

Fazit

Der Funktionsblock `AX_TO_AI` stellt eine einfache, aber effektive Lösung zur Umwandlung boolescher Adapter-Signale in Integer-Adapter-Signale dar. Er kapselt die Logik in einem wiederverwendbaren Composite-Baustein, nutzt Standardbibliotheken und ermöglicht eine saubere, adapterbasierte Kommunikation. Der Baustein eignet sich besonders für modulare Automatisierungslösungen, bei denen unterschiedliche Schnittstellentypen aufeinander abgestimmt werden müssen.

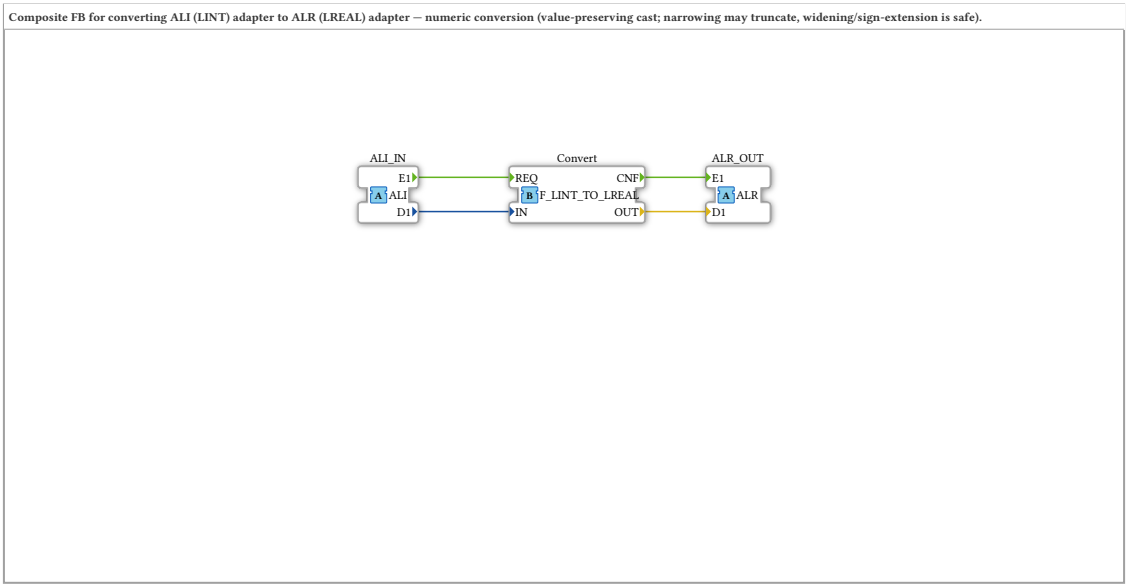
ALI_ALR

ALI_TO_ALR



LINT Adapter Input – *ALI*

ALR – LREAL Adapter Output



(Bild nicht verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ALI_TO_ALR** ist ein Composite-Baustein, der die Konvertierung eines **ALI-Adapters** (LINT-Datentyp) in einen **ALR-Adapter** (LREAL-Datentyp) realisiert. Er dient als Brücke zwischen unterschiedlichen Adapter-Schnittstellen und ermöglicht eine nahtlose Datenüberführung von 64-Bit-Ganzzahlen in 64-Bit-Gleitkommazahlen.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine separaten Ereignis-Eingänge. Die ereignisgesteuerte Verarbeitung erfolgt über den **Socket** `ALI_IN`, dessen Ereignis `E1` die Konvertierung auslöst.

Ereignis-Ausgänge

Es sind keine eigenständigen Ereignis-Ausgänge vorhanden. Das Ergebnisereignis wird über den **Plug** `ALR_OUT` auf dessen Ereignis `E1` ausgegeben.

Daten-Eingänge

Der FB verfügt über keine direkten Daten-Eingänge. Die zu konvertierenden LINT-Daten werden über den **Socket** `ALI_IN` (Datenpunkt `D1`) bereitgestellt.

Daten-Ausgänge

Analog gibt es keine separaten Daten-Ausgänge. Der konvertierte LREAL-Wert wird über den **Plug** `ALR_OUT` (Datenpunkt `D1`) ausgegeben.

Adapter

Typ	Name	Richtung	Beschreibung
Socket	`ALI_IN`	Eingang	Adapter vom Typ `adapter::types::unidirectional::ALI` (LINT)
Plug	`ALR_OUT`	Ausgang	Adapter vom Typ `adapter::types::unidirectional::ALR` (LREAL)

Funktionsweise

Der FB arbeitet als reine Weiterleitung und Konvertierung:

1. Ein Ereignis am **Socket** `ALI_IN.E1` triggert den internen Funktionsblock `Convert` (Typ `F_LINT_TO_LREAL`).
2. Gleichzeitig wird der LINT-Wert von `ALI_IN.D1` an den Eingang `Convert.IN` übergeben.
3. Der Konvertierungsblock `Convert` wandelt den Wert gemäß IEC 61131-Konvertierungslogik in einen LREAL-Wert um.
4. Nach Abschluss der Konvertierung erzeugt `Convert` das Ereignis `CNF`, das an den **Plug** `ALR_OUT.E1` weitergeleitet wird.
5. Zeitgleich wird der konvertierte LREAL-Wert von `Convert.OUT` an `ALR_OUT.D1` übergeben.

Die gesamte Verarbeitung erfolgt ereignisgesteuert und ohne eigene Zustandslogik.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-Baustein**: Die Logik wird ausschließlich durch die Verschaltung des standardisierten Konvertierungsbausteins `F_LINT_TO_LREAL` realisiert – kein eigener Algorithmus.
- **Adapter-orientierte Schnittstelle**: Der FB kapselt die Konvertierung vollständig in die Adapter-Schnittstellen, sodass er in Umgebungen mit einheitlichen Adapter-Typen (unidirektional) eingesetzt werden kann.
- **Kein Zustandsautomat**: Aufgrund der reinen Datenfluss-Struktur gibt es keinen internen Zustandsautomaten.

Zustandsübersicht

Der FB besitzt keinen eigenen Zustandsautomaten. Die Abarbeitung folgt ausschließlich der Ereigniskette des inneren Netzwerks:

- Idle → Ereignis an `ALI_IN.E1` → Konvertierung läuft → Ergebnisereignis an `ALR_OUT.E1` → zurück in Idle.

Anwendungsszenarien

- Integration von Komponenten, die LINT-Werte liefern (z. B. Zähler, Positionsgeber), in Systeme, die LREAL erwarten (z. B. Regelungsalgorithmen mit Gleitkommaberechnung).
- Umstellung von Integer-basierten auf Fließkomma-basierte Adapter-Schnittstellen ohne Änderung der übergeordneten Architektur.
- Zwischenschicht in modularen Steuerungsprojekten, in denen unterschiedliche Datentypen miteinander verbunden werden müssen.

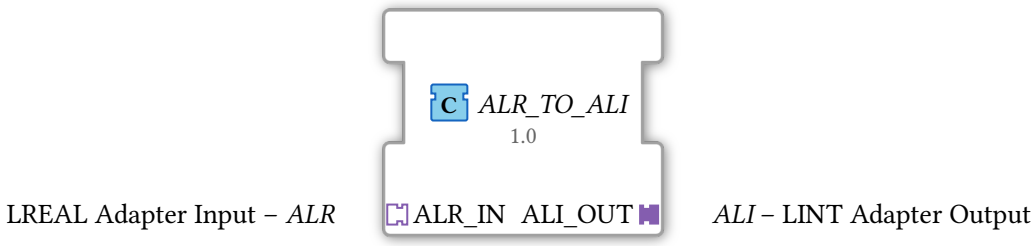
Vergleich mit ähnlichen Bausteinen

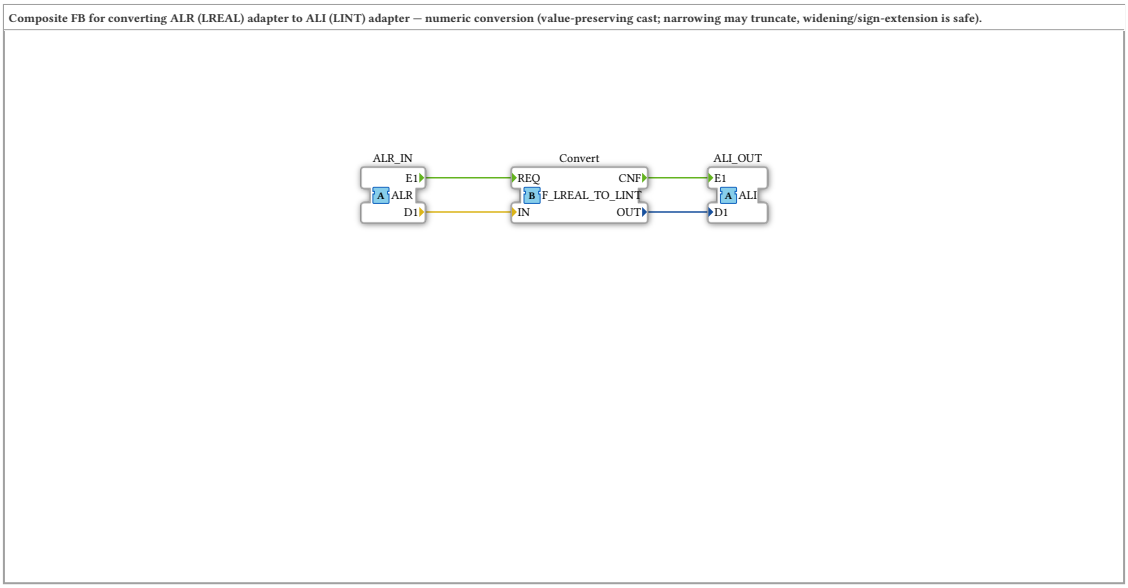
In der Bibliothek existieren analoge Konverter für andere Datentypen (z. B. `LINT_TO_REAL`, `DINT_TO_REAL`). `ALI_TO_ALR` ist die Adapter-spezifische Variante, die die Konvertierung in die standardisierten unidirektionalen Adapter `ALI` und `ALR` einbettet. Sie erlaubt einen einfachen Austausch zwischen verschiedenen Adapter-Typen, ohne dass die aufrufende Umgebung die Details der Konvertierung kennen muss.

Fazit

Der FB `ALI_TO_ALR` bietet eine saubere, wartbare und standardkonforme Lösung zur Umwandlung eines LINT-Adapters in einen LREAL-Adapter. Durch die Nutzung eines bewährten IEC 61131-Konvertierungsbausteins bleibt die Implementierung zuverlässig und effizient. Er eignet sich besonders für modulare Steuerungsarchitekturen, in denen Datentyp- und Adapter-Kompatibilität gefordert ist.

ALR_TO_ALI





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **ALR_TO_ALI** ist ein Composite-Baustein, der eine Adapter-Konvertierung von **ALR (LREAL)** nach **ALI (LINT)** realisiert. Er ermöglicht die einfache und strukturierte Integration einer LREAL-basierten Adapter-Schnittstelle in eine LINT-basierte Umgebung, ohne dass die Konvertierungslogik selbst implementiert werden muss. Der Baustein ist für den Einsatz in IEC 61499 basierten Industriesteuerungen konzipiert und nutzt den internen Funktionbaustein `F_LREAL_TO_LINT` nach IEC 61131-3.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine expliziten Ereignis-Eingänge. Das auslösende Ereignis wird über den eingehenden Adapter **ALR_IN** (Typ `adapter::types::unidirectional::ALR`) bereitgestellt:

- **ALR_IN.E1** – Eingangsereignis, das die Konvertierung startet.

Ereignis-Ausgänge

Der FB besitzt keine expliziten Ereignis-Ausgänge. Das Bestätigungsereignis wird über den ausgehenden Adapter **ALI_OUT** (Typ `adapter::types::unidirectional::ALI`) bereitgestellt:

- **ALI_OUT.E1** – Ausgangsereignis, das die Fertigstellung der Konvertierung signalisiert.

Daten-Eingänge

Der FB besitzt keinen expliziten Daten-Eingang. Der Eingangswert wird über den Adapter **ALR_IN** zur Verfügung gestellt:

- **ALR_IN.D1** – Wert vom Typ LREAL, der in LINT konvertiert werden soll.

Daten-Ausgänge

Der FB besitzt keinen expliziten Daten-Ausgang. Das Ergebnis wird über den Adapter **ALI_OUT** ausgegeben:

- **ALI_OUT.D1** – konvertierter Wert vom Typ LINT.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	ALR_IN	`adapter::types::unidirectional::ALR`	Unidirektionaler Adapter für LREAL-Daten und Ereignisse. Enthält Ereignis-Port E1 und Daten-Port D1 (LREAL).
Plug (Ausgang)	ALI_OUT	`adapter::types::unidirectional::ALI`	Unidirektionaler Adapter für LINT-Daten und Ereignisse. Enthält Ereignis-Port E1 und Daten-Port D1 (LINT).

Funktionsweise

Der FB arbeitet als reiner „Wrapper“ für die Typkonvertierung von LREAL nach LINT. Das Ablaufdiagramm ist wie folgt:

1. Ein Ereignis (E1) auf dem Eingangsadapter **ALR_IN** triggert den internen Baustein **Convert** (Typ `iec61131::conversion::F_LREAL_TO_LINT`).
2. Der Datenwert von **ALR_IN.D1** (LREAL) wird an den Eingang **Convert.IN** übergeben.
3. Der interne Baustein führt die Konvertierung durch und liefert das Ergebnis an **Convert.OUT**.
4. Das Ereignis am Ausgang **Convert.CNF** wird an **ALI_OUT.E1** weitergeleitet, sodass der ausgehende Adapter die fertige Konvertierung signalisiert.
5. Der konvertierte LINT-Wert steht an **ALI_OUT.D1** zur Verfügung.

Die Aufrufsteuerung erfolgt über die Ereignisverkettung (E1 → REQ → CNF → E1), sodass die Konvertierung blockierend innerhalb eines Ausführungszyklus erfolgt.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-Struktur**: Der Baustein ist als Composite (Netzwerk) realisiert und nutzt intern den Standardbaustein `F_LREAL_TO_LINT` aus der IEC 61131-3-Bibliothek.

- **Adapterkapselung**: Durch die Adaptertechnik können die konkreten LREAL- und LINT-Signale über standardisierte Schnittstellen (ALR, ALI) angebunden werden, was eine hohe Wiederverwendbarkeit und Austauschbarkeit ermöglicht.
- **Lizenz**: Der Quelltext steht unter der Eclipse Public License 2.0 (EPL-2.0).
- **Version**: 1.0, veröffentlicht am 17.02.2026 durch HR Agrartechnik GmbH, Autor: Franz Höpfinger.
- **Paketstruktur**: `adapter::conversion::unidirectional`.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten, da er als Netzwerk von einem einzigen internen FB realisiert wird. Das Verhalten ist rein ereignisgesteuert:

- **Idle**: Warten auf ein Ereignis am Eingangsadapter (ALR_IN.E1).
- **Processing**: Nach Eintreffen des Ereignisses wird der interne Konvertierungsbaustein aktiviert.
- **Done**: Nach Abschluss (CNF) wird das Ausgangsereignis (ALI_OUT.E1) gesetzt und der Baustein kehrt in den Idle-Zustand zurück.

Ein expliziter Fehlerzustand existiert nicht, da der Konvertierungsbaustein `F_LREAL_TO_LINT` keine Fehlerbehandlung vorsieht.

Anwendungsszenarien

- **Migration**: Austausch eines ALI-basierten (LINT) Systems gegen ein ALR-basiertes (LREAL) System oder umgekehrt, ohne Änderung der übergeordneten Logik.
- **Adapter-Brücke**: Wenn eine Komponente nur LREAL-Werte liefert, der nachfolgende Adapter aber LINT erwartet, kann dieser FB dazwischen geschaltet werden.
- **Standardisierte Konvertierung**: In Bibliotheken oder Frameworks, die eine einheitliche Umsetzung von Typkonvertierungen über Adapter fordern.

Vergleich mit ähnlichen Bausteinen

In der IEC 61499- und IEC 61131-Welt existieren zahlreiche Konvertierungsbausteine:

- **F_LREAL_TO_LINT** (direkt): Einzelner Baustein, der einen einfachen Wert konvertiert, aber keine Adapter-Schnittstelle bietet.
- **ALR_TO_ALI** (dieser FB): Kapselt den direkten Baustein in eine Adapter-Struktur, sodass die Anbindung über einheitliche Adapter-Ports erfolgt.
- **Weitere Typkonvertierungen**: Ähnlich aufgebaute Bausteine für z.B. REAL → DINT oder LREAL → DWORD können nach demselben Muster erstellt werden.

Der entscheidende Vorteil von **ALR_TO_ALI** ist die Wiederverwendbarkeit im Adapterkontext und die saubere Trennung von Ereignis und Daten.

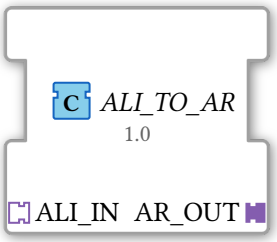
Fazit

Der Funktionsbaustein **ALR_TO_ALI** ist ein nützlicher Composite-Baustein, der eine saubere und standardisierte Adapterkonvertierung von LREAL nach LINT ermöglicht. Er erbt die Robustheit des IEC 61131-3-Konvertierungsbausteins und integriert ihn nahtlos in die adapterbasierte Kommunikationsstruktur. Durch seine offene Lizenz und die klare Schnittstelle eignet er sich gut für den Einsatz in modularen Automatisierungslösungen.

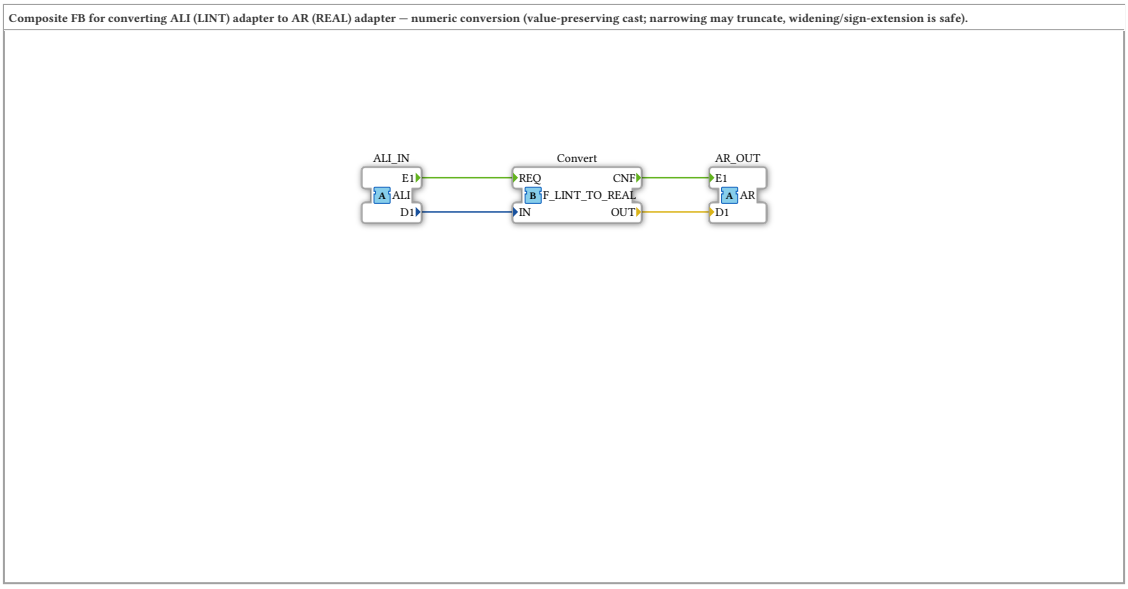
ALI_AR

ALI_TO_AR

LINT Adapter Input – *ALI*



AR – REAL Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ALI_TO_AR** ist ein zusammengesetzter Baustein (Composite FB), der einen LINT-Adapter (ALI) in einen REAL-Adapter (AR) umwandelt. Er dient als universelle Konvertierungsschnittstelle, um ganzzahlige 64-Bit-Werte aus der IEC-61131-Welt in Gleitkommawerte für die weitere Verarbeitung in Steuerungsanwendungen bereitzustellen. Der Baustein ist unter der Eclipse Public License 2.0 lizenziert und wird von der HR Agrartechnik GmbH bereitgestellt.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Datenports. Die gesamte Kommunikation erfolgt über zwei vordefinierte unidirektionale Adapter:

Ereignis-Eingänge

– Keine (die Ereignisschnittstelle wird über den Adapter **ALI_IN** bereitgestellt)

Ereignis-Ausgänge

– Keine (die Ereignisschnittstelle wird über den Adapter **AR_OUT** bereitgestellt)

Daten-Eingänge

– Keine (die Datenschnittstelle wird über den Adapter **ALI_IN** bereitgestellt)

Daten-Ausgänge

– Keine (die Datenschnittstelle wird über den Adapter **AR_OUT** bereitgestellt)

Adapter

Richtung	Name	Typ	Beschreibung
Socket	ALI_IN	`adapter::types::unidirectional::ALI`	LINT-Eingabeadapter (64-Bit Integer)
Plug	AR_OUT	`adapter::types::unidirectional::AR`	REAL-Ausgabeadapter (Gleitkomma)

Die Adapter enthalten jeweils einen Ereigniskanal (E1) und einen Datenkanal (D1), die über die interne Vernetzung des Bausteins verbunden sind.

Funktionsweise

Der Baustein realisiert eine einfache datengetriebene Konvertierung:

1. Über den Socket **ALI_IN** trifft ein Ereignis am Ereignis-Eingang **E1** ein – gleichzeitig wird der zugehörige LINT-Wert am Daten-Eingang **D1** bereitgestellt.

2. Dieses Ereignis wird an den internen Baustein `iec61131::conversion::F_LINT_TO_REAL` weitergeleitet (Ereignis **REQ**).
3. Der interne Baustein wandelt den LINT-Wert in einen REAL-Wert um und signalisiert die Fertigstellung über den Ereignis-Ausgang **CNF**.
4. Der konvertierte REAL-Wert wird über die Datenverbindung an den Daten-Eingang des Plug-Adapters **AR_OUT** (D1) übergeben.
5. Gleichzeitig wird das Ereignis **E1** des Adapters **AR_OUT** ausgelöst, sodass die angeschlossene Logik den neuen REAL-Wert verarbeiten kann.

Die gesamte Umwandlung erfolgt ohne Zustandsspeicherung – jede Anforderung wird sofort bearbeitet.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-Baustein**: Der ALI_TO_AR ist ein zusammengesetzter Funktionsblock, der ausschließlich aus dem internen Standardbaustein `F_LINT_TO_REAL` besteht.
- **Bibliothekseinbindung**: Die Konvertierung nutzt die IEC-61131-Bibliothek `iec61131::conversion::F_LINT_TO_REAL`, die als Import deklariert ist.
- **Paketstruktur**: Der Baustein ist im Paket `adapter::conversion::unidirectional` organisiert.
- **Lizenz**: Der Quellcode steht unter der Eclipse Public License 2.0 (EPL-2.0).

Zustandsübersicht

Der Baustein besitzt keine eigenen Zustände. Er arbeitet rein ereignisgesteuert und datenflussorientiert. Es gibt keine initialisierungs- oder fehlerbezogenen Zustände – jede ankommende Anfrage wird direkt verarbeitet.

Anwendungsszenarien

- **Zählerauswertung**: Ein LINT-Zähler (z. B. Impulszähler) liefert eine große Ganzzahl, die für eine Regelung in einen REAL-Wert umgerechnet werden muss.
- **Datenvorverarbeitung**: In einer SPS-Applikation liegen Messwerte als LINT vor (z. B. aus einem Encoder), die für weitere Berechnungen oder Visualisierung in REAL konvertiert werden.
- **Adapter-Brücke**: Der Baustein dient als standardisierter Konverter zwischen zwei unterschiedlichen Adaptertypen in einer geräteübergreifenden Kommunikationsinfrastruktur.

Vergleich mit ähnlichen Bausteinen

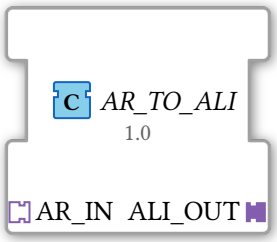
- **ALI_TO_ARI**: Wandelt LINT in ein ARRAY-von-REAL um – für Mehrfachwerte oder Puffer.
- **LREAL_TO_AR**: Konvertiert LREAL (80-Bit-Gleitkomma) in REAL – für Verluste durch geringere Genauigkeit.
- **Direkte Typkonvertierung** in IEC-61131-Funktionen (z. B. `LINT_TO_REAL`) – ohne Adapter-Kapselung, erfordert manuelle Verdrahtung von Ereignissen.

Der ALI_TO_AR bietet gegenüber reinen Funktionsaufrufen den Vorteil der standardisierten Adapter-Schnittstelle, die in modularen Systemen mit plug-&-play-Fähigkeit verwendet werden kann.

Fazit

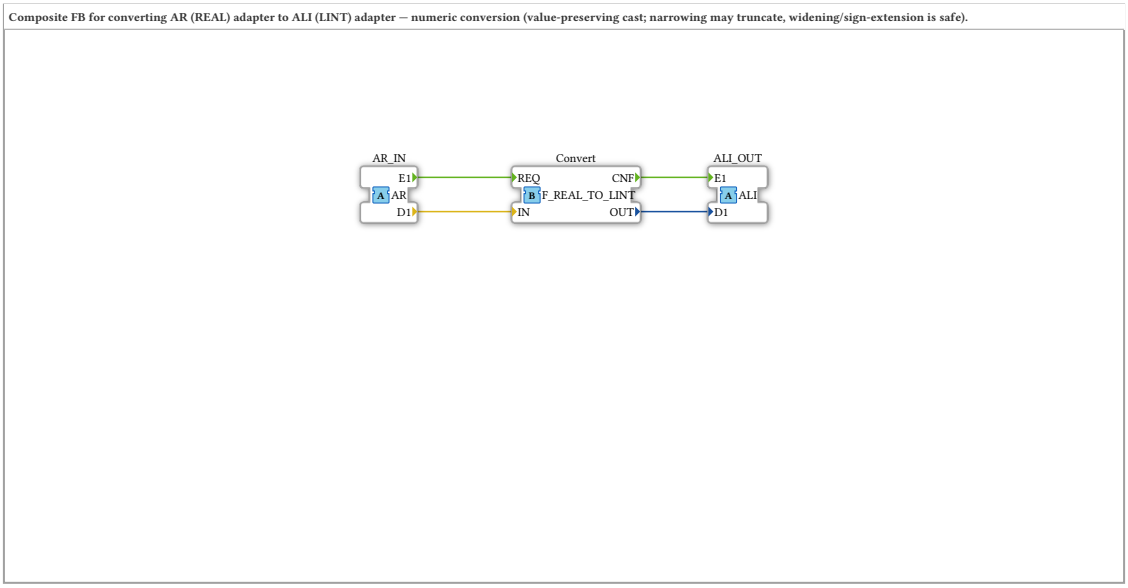
Der **ALI_TO_AR** ist ein kompakter, wiederverwendbarer Funktionsblock zur Konvertierung von LINT- in REAL-Werte über eine Adapter-Schnittstelle. Seine einfache interne Struktur und die klare Ereignis-Daten-Kopplung machen ihn zu einem unverzichtbaren Baustein für jede Applikation, die ganzzahlige Sensordaten in Gleitkommawerte für Regelungen oder Visualisierungen umsetzen muss. Die Lizenzierung unter EPL-2.0 ermöglicht eine uneingeschränkte Nutzung in offenen und kommerziellen Projekten.

AR_TO_ALI



REAL Adapter Input – *AR*

ALI – LINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Composite Funktionsblock `AR_TO_ALI` dient der Umwandlung eines REAL-Adapters (`AR-IN`) in einen LINT-Adapters (`ALI-OUT`). Er ermöglicht die nahtlose Integration von Gleitkommawerten aus dem Adapter `AR` in Applikationen, die ganzzahlige 64-Bit-Werte über den Adapter `ALI` erwarten. Die Konvertierung erfolgt über einen internen Baustein `F_REAL_TO_LINT` nach IEC 61131.

Schnittstellenstruktur

Der FB besitzt keine eigenständigen Ereignis- oder Dateneingänge/-ausgänge auf Blockebene. Die gesamte Signalübertragung erfolgt über Adapter-Sockets bzw.-Plugs.

Ereignis-Eingänge

Keine eigenständigen Ereignis-Eingänge. Das Ereignis wird über den Socket `AR_IN` (Schnittstelle `E1`) empfangen.

Ereignis-Ausgänge

Keine eigenständigen Ereignis-Ausgänge. Das Ereignis wird über den Plug `ALI_OUT` (Schnittstelle `E1`) gesendet.

Daten-Eingänge

Keine eigenständigen Daten-Eingänge. Die Daten werden über den Socket `AR_IN` (Schnittstelle `D1`) empfangen.

Daten-Ausgänge

Keine eigenständigen Daten-Ausgänge. Die Daten werden über den Plug `ALI_OUT` (Schnittstelle `D1`) gesendet.

Adapter

Richtung	Bezeichnung	Typ	Kompatibler Adaptertyp	Beschreibung
Socket	AR_IN	adapter::types::unidirectional:AR	AR-Adapter (REAL)	Aufnehmender Adapter für den REAL-Wert.
Plug	ALI_OUT	adapter::types::unidirectional:ALI	ALI-Adapter (LINT)	Bereitstellender Adapter für den konvertierten LINT-Wert.

Funktionsweise

Im inneren Netzwerk wird der FB `iec61131::conversion::F_REAL_TO_LINT` instanziiert. Sobald am Socket `AR_IN` ein Ereignis anliegt (über die Schnittstelle `E1`), wird der interne Konvertierungsbaustein mit der dafür vorgesehenen Ereignisverbindung `AR_IN.E1 -> Convert.REQ` angestoßen. Gleichzeitig wird der Datenwert `AR_IN.D1` an den Eingang `Convert.IN` übergeben. Nach erfolgter Umrechnung (REAL → LINT) bestätigt der Konverter mit dem Ereignis `Convert.CNF`. Dieses Ereignis wird an den Plug `ALI_OUT.E1` weitergeleitet, und der konvertierte Wert `Convert.OUT` wird auf `ALI_OUT.D1` ausgegeben.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Reine Adapter-Kopplung**: Der FB verwendet keine direkten Ereignis- oder Dateneingänge, sondern kommuniziert ausschließlich über die standardisierten unidirektionalen Adapter `AR` und `ALI`. Dadurch lässt er sich leicht in bestehende Adapter-Strukturen einfügen.
- **IEC 61131-Konformität**: Der eingesetzte Konvertierungsbaustein `F_REAL_TO_LINT` entspricht der IEC-Norm und stellt eine standardisierte, deterministische Umwandlung sicher.
- **Laufzeitverhalten**: Die Konvertierung erfolgt ereignisgesteuert und ohne zusätzliche Verzögerung; die Ausgangsgröße liegt im selben Ausführungszyklus an, in dem das Eingangsereignis empfangen wird.

Zustandsübersicht

Der Composite FB selbst besitzt keinen eigenen Zustandsautomaten. Sein Verhalten wird vollständig durch die interne Verschaltung bestimmt. Es existieren folgende logische Zustände:

- **Idle**: Kein Ereignis am Eingang. Der Ausgang `ALI_OUT` bleibt unverändert.
- **Converting**: Ein Ereignis am Socket `AR_IN` wird empfangen. Der interne Konverter wird aktiviert, und die Konvertierung läuft.
- **Completed**: Die Konvertierung ist abgeschlossen; das Ausgangsereignis wird gesendet und der neue Datenwert liegt am Plug an.

Anwendungsszenarien

- **Sensor-Aktor-Kopplung**: Ein REAL-Adapter (z. B. von einem analogen Sensor) muss an ein Steuermodul angebunden werden, das ausschließlich LINT-Werte über den ALI-Adapter verarbeitet.
- **Protokoll-Wandlung**: In heterogenen Automatisierungsnetzwerken, in denen Datenformate konvertiert werden müssen, um unterschiedliche Adaptertypen miteinander zu verbinden.
- **Test- und Simulationsumgebungen**: Ersetzen eines ALI-Adapters durch einen simulierten REAL-Adapter und Rückkonvertierung für Testzwecke.

Vergleich mit ähnlichen Bausteinen

FB	Konvertierung	Schnittstelle	Anmerkung
`AR_TO_ALI`	REAL → LINT	AR / ALI-Adapter	Speziell für Adapter-kopplung
`F_REAL_TO_LINT`	REAL → LINT	Eigenständige E/A (REQ/CNF, IN/OUT)	Direkter Konverter ohne Adapter

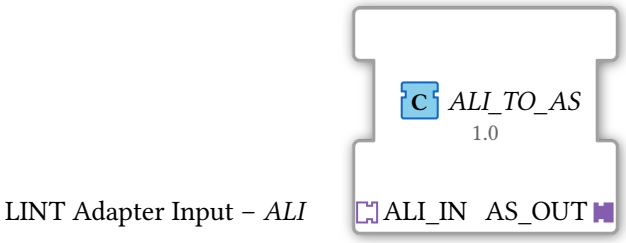
`AR_TO_ALI` umschließt `F_REAL_TO_LINT` und fügt die Adapter-Anschlüsse hinzu. Der Vorteil liegt in der einfachen adapternetzwerktauglichen Anbindung. Ein direkter Konverter ohne Adapter wäre in Adapter-Strukturen nicht einsetzbar.			

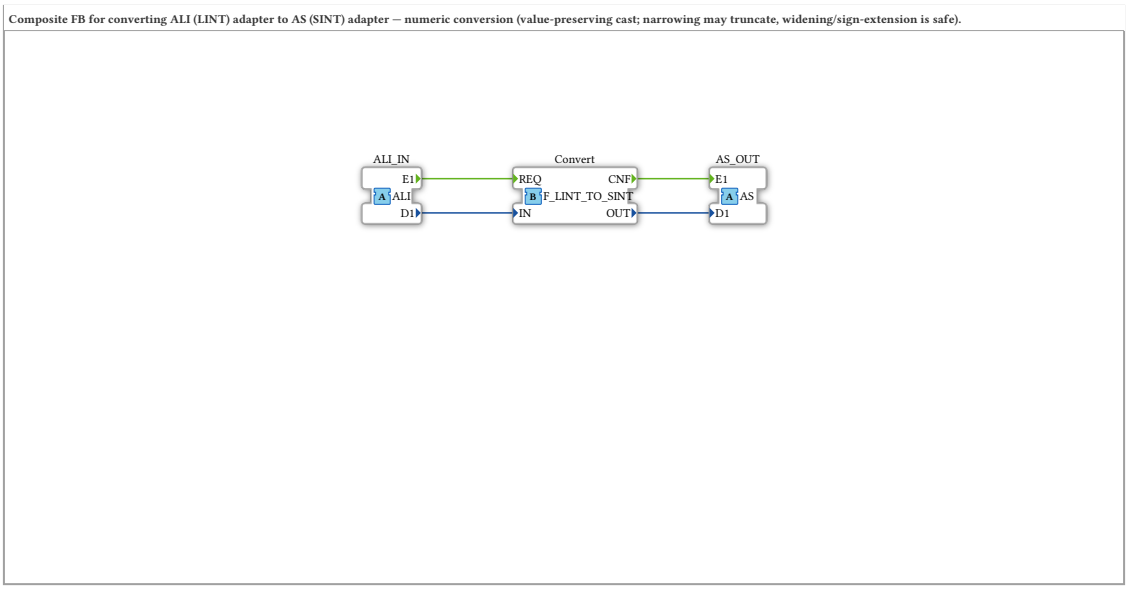
Fazit

`AR_TO_ALI` ist ein kompakter, zweckgebundener Composite-FB, der die Konvertierung von REAL- nach LINT-Werten über die standardisierten unidirektionalen Adapter `AR` und `ALI` ermöglicht. Durch die saubere Kapselung und die Verwendung eines IEC-konformen Kernbausteins wird eine zuverlässige und wiederverwendbare Lösung für die industrielle Automatisierung bereitgestellt. Der FB eignet sich besonders für Szenarien, in denen eine saubere Trennung zwischen Datenformat und Adapter-Schnittstelle erforderlich ist.

ALI_AS

ALI_TO_AS





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ALI_TO_AS** ist ein zusammengesetzter (Composite) Baustein, der einen ALI-Adapter (Datentyp LINT) in einen AS-Adapter (Datentyp SINT) umwandelt. Er kapselt die Konvertierung eines 64-Bit-Integerwerts in einen 8-Bit-Integerwert und ermöglicht so die nahtlose Integration zwischen unterschiedlichen Adapter-Schnittstellen in der 4diac-IDE.

Schnittstellenstruktur

Ereignis-Eingänge
Keine. Der FB besitzt keine separaten Ereignis-Eingänge; die Ereignissteuerung erfolgt über die angeschlossenen Adapter.

Ereignis-Ausgänge
Keine. Ereignisse werden ausschließlich über den AS-Adapter weitergeleitet.

Daten-Eingänge
Keine. Das zu konvertierende LINT-Signal wird über den ALI-Adapter eingelesen.

Daten-Ausgänge
Keine. Das konvertierte SINT-Signal wird über den AS-Adapter ausgegeben.

Adapter

Name	Typ	Richtung	Beschreibung
ALI_IN	`adapter::types::unidirectional::ALI`	Socket (Eingang)	Aufnahme eines LINT-Werts (64-Bit) über das Ereignis E1 und Datum D1.
AS_OUT	`adapter::types::unidirectional::AS`	Plug (Ausgang)	Bereitstellung des konvertierten SINT-Werts (8-Bit) über Ereignis E1 und Datum D1.

Funktionsweise
Der Baustein arbeitet nach dem Prinzip eines reinen Datenflusses ohne eigenen Zustandsautomaten.

1. Über den **ALI_IN**-Adapter wird das Ereignis **E1** empfangen und der zugehörige LINT-Wert über das Datum **D1** bereitgestellt.
2. Dieses Ereignis triggert den internen Funktionsblock **F_LINT_TO_SINT**, der die Konvertierung durchführt.
3. Nach Abschluss der Konvertierung wird das Ereignis **CNF** des Konvertierungsbausteins aktiviert, welches wiederum das Ausgangsereignis **E1** des **AS_OUT**-Adapters auslöst und den konvertierten SINT-Wert über das Datum **D1** weitergibt.

Die Wandlung entspricht der IEC-61131-Funktion `LINT_TO_SINT`. Dabei kommt es bei Werten außerhalb des SINT-Bereichs (-128…127) zu einem Überlauf – der Baustein führt keine zusätzliche Bereichsprüfung durch.

Technische Besonderheiten
- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-Struktur**: Der FB setzt sich ausschließlich aus einem einzigen Konvertierungsbaustein (`F_LINT_TO_SINT`) zusammen.
- **Unidirektionale Adapter**: Sowohl Eingangs- als auch Ausgangsschnittstelle sind als unidirektionale Adapter definiert – Daten fließen nur in eine Richtung.
- **Kein Zustandsautomat**: Aufgrund der einfachen Transformation wird auf einen internen Zustandsautomaten verzichtet; die Logik erschöpft sich in der Ereignis-Daten-Verkettung.

Zustandsübersicht
Der Baustein besitzt **keinen** internen Zustandsautomaten. Er arbeitet rein daten-/ereignisgesteuert und führt bei jedem eingehenden Ereignis eine sofortige Konvertierung durch.

Anwendungsszenarien
- **Integration von LINT-Datenquellen** (z.B. aus hochauflösenden Zählern, Zeitstempeln) in Systeme, die nur SINT-Werte verarbeiten (z.B. einfache Aktoransteuerungen).

- **Brücken zwischen verschiedenen Adapter-Typen** innerhalb einer 4diac-Applikation, ohne dass manuelle Konvertierungsbausteine verschaltet werden müssen.

Vergleich mit ähnlichen Bausteinen

- **ALI_TO_INT**: Wandelt LINT nach INT (16-Bit).
- **LINT_TO_DINT** o.ä.: Direkte Typumwandlungen ohne Adapter-Kapselung. Der **ALI_TO_AS** zeichnet sich durch seine spezifische Adapter-Schnittstelle aus; er vereinfacht die Wiederverwendung vorgefertigter Adapter-Kopplungen in Modulen.

Fazit

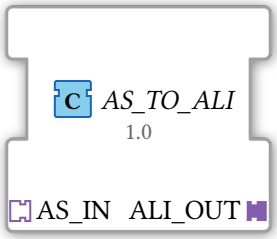
Der **ALI_TO_AS** ist ein spezialisierter Komposit-Baustein zur verlustbehafteten Konvertierung eines LINT- in einen SINT-Wert, eingebettet in eine unidirektionale Adapter-Struktur. Er bietet eine saubere, wiederverwendbare Kapselung der Wandlungslogik und erleichtert die Anbindung unterschiedlichster Datenquellen und -senken innerhalb der 4diac-Entwicklungsumgebung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

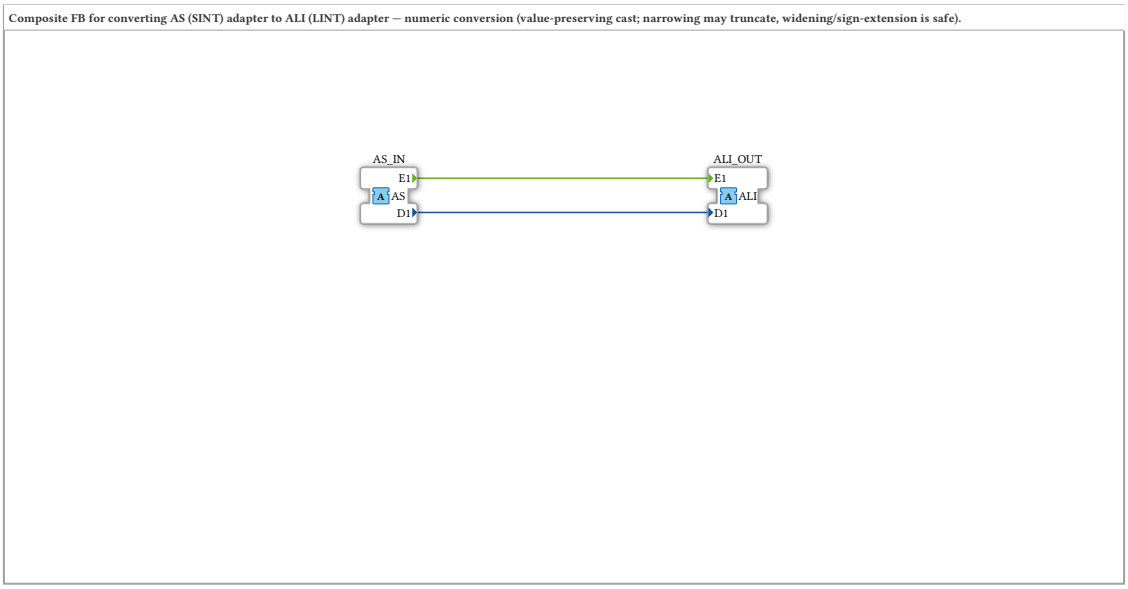
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AS_TO_ALI

SINT Adapter Input – *AS*



ALI – LINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der **AS_TO_ALI** Funktionsblock ist ein Composite-Baustein zur Umwandlung eines SINT-Adapter-Eingangs (AS) in einen LINT-Adapter-Ausgang (ALI). Er dient als Brücke zwischen Komponenten, die unterschiedliche ganzzahlige Datentypen verwenden, und ermöglicht eine einfache Integration von SINT-basierten Signalen in LINT-basierte Systeme.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine direkt sichtbaren Ereignis-Eingänge. Die Ereignissteuerung erfolgt über den Socket-Adapter **AS_IN**, der ein Ereignis **E1** bereitstellt.

Ereignis-Ausgänge

Der FB besitzt keine direkt sichtbaren Ereignis-Ausgänge. Die Ereignisausgabe erfolgt über den Plug-Adapter **ALI_OUT**, der ein Ereignis **E1** weitergibt.

Daten-Eingänge

Keine direkten Daten-Eingänge. Der Datenwert **D1** wird über den Socket-Adapter **AS_IN** (Datentyp SINT) eingelesen.

Daten-Ausgänge

Keine direkten Daten-Ausgänge. Der konvertierte Datenwert **D1** wird über den Plug-Adapter **ALI_OUT** (Datentyp LINT) ausgegeben.

Adapter

- **AS_IN** (Socket, Typ: `adapter::types::unidirectional::AS`) – SINT-Adapter-Eingang mit Ereignis E1 und Daten D1 (SINT).
- **ALI_OUT** (Plug, Typ: `adapter::types::unidirectional::ALI`) – LINT-Adapter-Ausgang mit Ereignis E1 und Daten D1 (LINT).

Funktionsweise

Der AS_TO_ALI FB ist als Composite-Baustein realisiert. Bei Eintreffen eines Ereignisses am **AS_IN.E1** wird dieses direkt an **ALI_OUT.E1** weitergeleitet. Gleichzeitig wird der eingehende SINT-Datenwert **AS_IN.D1** in einen LINT-Wert konvertiert und an **ALI_OUT.D1** übergeben. Die Umwandlung erfolgt verlustfrei, da der Wertebereich von LINT den von SINT vollständig abdeckt.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-Struktur** – keine eigene Zustandslogik, reine Weiterleitung und Konvertierung.
- **Synchronisierte Ereignis- und Datenpfade** – die Datenweitergabe erfolgt nur nach einem Ereignis.
- **Implizite Typkonvertierung** – die Wandlung von SINT nach LINT geschieht automatisch im Netzwerk des Composite-Blocks.

Zustandsübersicht

Der FB besitzt keine eigenen Zustände. Die Funktionalität beschränkt sich auf die ereignisgesteuerte Datenweitergabe mit Typkonvertierung.

Anwendungsszenarien

- Anbindung eines SINT-basierten Sensors an ein LINT-basiertes Steuerungssystem.
- Migration bestehender SINT-Komponenten auf eine LINT-Infrastruktur.
- Heterogene Systeme, in denen unterschiedliche ganzzahlige Datentypen konvertiert werden müssen.

Vergleich mit ähnlichen Bausteinen

Ähnliche Konverter existieren für andere Datentypen (z.B. BOOL_TO_BYTE, INT_TO_DINT). Der AS_TO_ALI FB ist speziell für die Umwandlung von SINT zu LINT ausgelegt und als Adapter-Baustein konzipiert, sodass er direkt in Adapter-basierte Kommunikationsstrukturen eingebunden werden kann.

Fazit

Der AS_TO_ALI FB bietet eine einfache, zuverlässige und effiziente Lösung zur Konvertierung von SINT- auf LINT-Daten. Durch seine Composite-Struktur ist er ohne zusätzliche Programmierung in vorhandene 4diac-Projekte integrierbar und erleichtert die Verwendung unterschiedlicher Datentypen in Automatisierungssystemen.

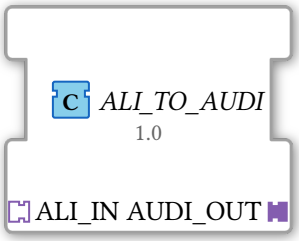
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

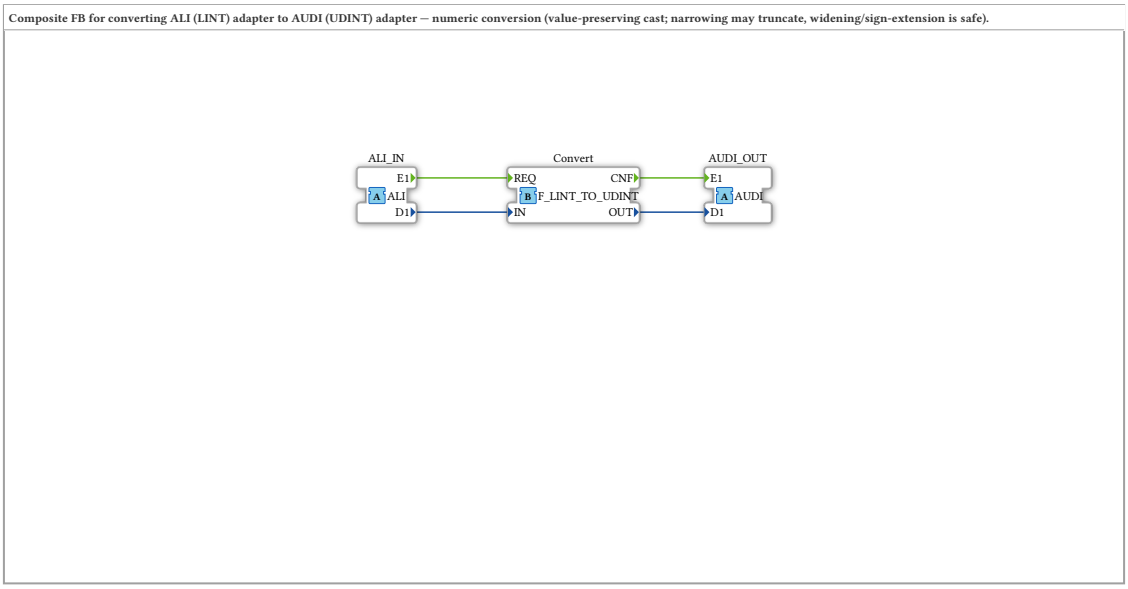
ALI_AUDI

ALI_TO_AUDI

LINT Adapter Input – *ALI*



AUDI – UDINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ALI_TO_AUDI** ist ein Composite Baustein, der einen ALI-Adapter (LINT) in einen AUDI-Adapter (UDINT) umwandelt. Er dient als unidirektionaler Konverter zwischen zwei unterschiedlichen Datentyp-Adaptern für die IEC 61499-Profilschnittstelle. Die Konvertierung erfolgt über einen internen Baustein *F_LINT_TO_UDINT* aus der IEC 61131-Bibliothek.

Schnittstellenstruktur

Der FB besitzt keine direkten Ereignis- oder Datenein-/ausgänge. Die gesamte Kommunikation erfolgt über zwei Adapter-Schnittstellen.

Ereignis-Eingänge

Keine eigenen Ereignis-Eingänge vorhanden. Das Ereignis wird über den ALI-Adapter empfangen.

Ereignis-Ausgänge

Keine eigenen Ereignis-Ausgänge vorhanden. Das Ergebnisereignis wird über den AUDI-Adapter gesendet.

Daten-Eingänge

Keine eigenen Daten-Eingänge vorhanden. Der Eingabedatenwert wird über den ALI-Adapter bereitgestellt.

Daten-Ausgänge

Keine eigenen Daten-Ausgänge vorhanden. Der konvertierte Datenwert wird über den AUDI-Adapter ausgegeben.

Adapter

Adapter	Typ	Richtung	Beschreibung
ALI_IN	`adapter::types::unidirectional::ALI`	Socket (Eingang)	Liefert den LINT-Wert (64-Bit signed) und das zugehörige Ereignis.
AUDI_OUT	`adapter::types::unidirectional::AUDI`	Plug (Ausgang)	Stellt den konvertierten UDINT-Wert (32-Bit unsigned) und das Bestätigungsereignis bereit.

Funktionsweise

Sobald der ALI_IN-Adapter ein Ereignis (E1) empfängt, wird der daran gebundene LINT-Datenwert (D1) ausgelesen. Dieser Datenwert wird an den internen FB *F_LINT_TO_UDINT* übergeben, der ihn in einen UDINT-Wert umwandelt. Nach erfolgreicher Konvertierung sendet der interne FB ein Bestätigungsereignis (CNF), das über den AUDI_OUT-Adapter (E1) weitergegeben wird. Gleichzeitig wird der konvertierte UDINT-Wert (OUT) an den AUDI_OUT-Adapter (D1) übergeben.

Die Konvertierung von LINT (64-Bit signed) nach UDINT (32-Bit unsigned) erfolgt ohne Vorzeichen und kann bei Werten außerhalb des UDINT-Bereichs (0 … 4.294.967.295) zu Überläufen oder Informationsverlust führen.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite Funktionsblock**: Der FB besteht aus einem internen Netzwerk mit einem einzigen Konvertierungsbaustein.
- **Unidirektionaler Adapter-zu-Adapter-Konverter**: Die Daten fließen nur von ALI_IN nach AUDI_OUT.
- **Keine Zustandslogik**: Der FB besitzt keinen internen Zustandsautomaten; die Konvertierung erfolgt bei jedem Ereignis deterministisch.

Zustandsübersicht

Der FB hat keinen eigenen Zustandsautomaten. Der Zustand wird allein durch den internen Baustein *F_LINT_TO_UDINT* bestimmt, der bei jedem REQ-Ereignis eine einmalige Konvertierung durchführt und sofort das Ergebnis liefert.

Anwendungsszenarien

- **Systemintegration**: Wenn ein Systemteil (z. B. Sensor, Antrieb) einen ALI-Adapter (LINT) bereitstellt, ein anderer Teil jedoch einen AUDI-Adapter (UDINT) erwartet.
- **Protokollanpassung**: In heterogenen Steuerungsumgebungen, in denen unterschiedliche IEC 61499-Adaterprofile für Ganzzahlen verwendet werden.
- **Migration**: Austausch von 64-Bit LINT-Werten gegen 32-Bit UDINT-Werte, sofern die Wertebereichsgrenzen eingehalten werden.

Vergleich mit ähnlichen Bausteinen

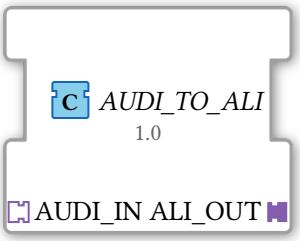
- **F_LINT_TO_UDINT (direkt)**: Dieser Baustein führt die reine Datenkonvertierung ohne Adapter durch. *ALI_TO_AUDI* kapselt diese Funktion in einer adapterbasierten Schnittstelle.
- **ALI_TO_xxx/Konverter**: Andere Adapter-Konverter (z. B. ALI_TO_BOOL, ALI_TO_REAL) verwenden analoge Mechanismen, unterscheiden sich jedoch in Zieltyp und Wertebereich.

Fazit

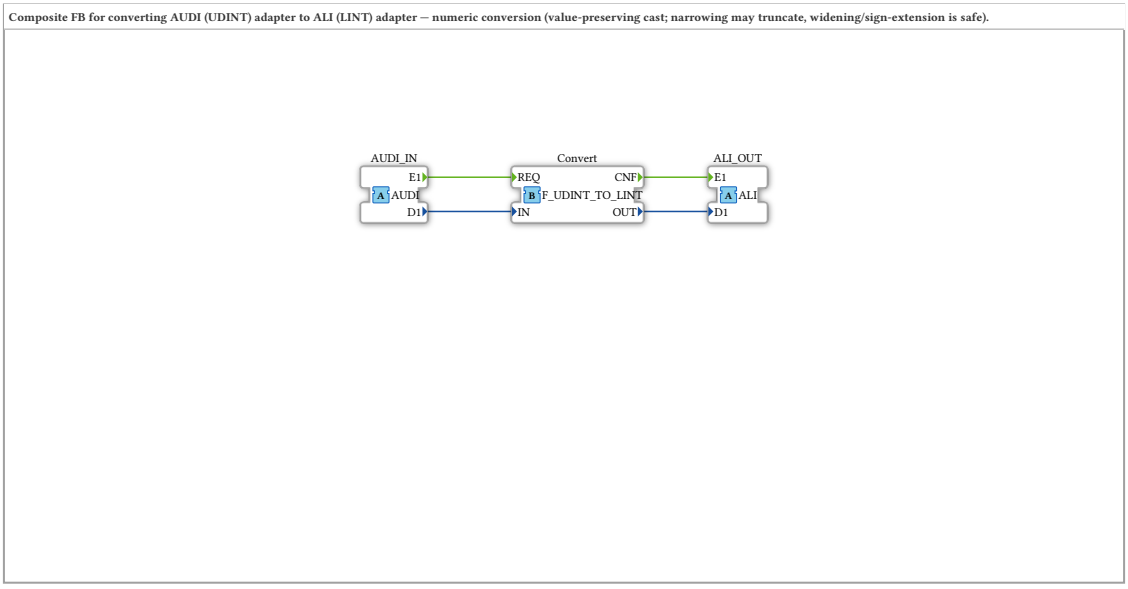
Der *ALI_TO_AUDI* Funktionsblock ist ein schlanker Composite-Baustein zur einfachen Adapterkonvertierung von LINT nach UDINT. Er nutzt einen vorhandenen IEC 61131-Baustein und ergänzt ihn um

eine standardisierte IEC 61499-Adapter-Schnittstelle. Dadurch wird die Integration in bestehende Systeme erleichtert, ohne dass die zugrundeliegende Konvertierungslogik neu implementiert werden muss.

AUDI_TO_ALI



UDINT Adapter Input – *AUDI* — *ALI* – LINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUDI_TO_ALI** ist ein Composite-Baustein, der eine Brücke zwischen einem **AUDI**-Adapter (vom Typ *UDINT*) und einem **ALI**-Adapter (vom Typ *LINT*) bereitstellt. Er wandelt eingehende UDINT-Werte über eine standardisierte Konvertierungsfunktion in LINT-Werte um und leitet sie an den Ausgangsadapter weiter. Der Baustein wird typischerweise in Anwendungen eingesetzt, die unterschiedliche Adress- oder Integertypen innerhalb eines 4diac-Adapterkonzepts verbinden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Beschreibung
keine eigenen Ereignis-Eingänge	Die Ereignissteuerung erfolgt über den internen Socket AUDI_IN.E1.

Ereignis-Ausgänge

Name	Beschreibung
keine eigenen Ereignis-Ausgänge	Die Ereignisweitergabe erfolgt über den internen Plug ALI_OUT.E1.

Daten-Eingänge

Name	Datentyp	Beschreibung

keine eigenen Daten-Eingänge	–	Der Datenwert wird vom Socket AUDI_IN.D1 geliefert.

Daten-Ausgänge

Name	Datentyp	Beschreibung
keine eigenen Daten-Ausgänge	–	Der konvertierte Wert wird über den Plug ALI_OUT.D1 bereitgestellt.

Adapter

Typ	Name	Richtung	Beschreibung
AUDI (unidirektional)	`AUDI_IN`	Socket (Eingang)	UDINT-Datenadapter, der einen Wert über das Ereignis E1 und das Datum D1 bereitstellt.
ALI (unidirektional)	`ALI_OUT`	Plug (Ausgang)	LINT-Datenadapter, der den konvertierten Wert über das Ereignis E1 und das Datum D1 ausgibt.

Funktionsweise

Der Baustein arbeitet als reine **Typkonvertierungsbrücke** für Adapter:

1. **Ereignisempfang**: Ein Ereignis auf dem Socket `AUDI_IN.E1` aktiviert den internen Konvertierungsbaustein `F_UDINT_TO_LINT` über dessen Ereigniseingang `REQ`.
2. **Datenkonvertierung**: Der aktuelle UDINT-Wert von `AUDI_IN.D1` wird auf den Dateneingang `IN` des Konverters gelegt. `F_UDINT_TO_LINT` führt die IEC-61131-konforme Umwandlung von einer vorzeichenlosen 32-Bit-Ganzzahl (UDINT) in eine vorzeichenbehaftete 64-Bit-Ganzzahl (LINT) durch.
3. **Ausgabe**: Nach erfolgreicher Konvertierung signalisiert der Konverter mit `CNF`, dass der Ausgangswert gültig ist. Dieses Ereignis wird direkt an den Plug `ALI_OUT.E1` weitergegeben, und der konvertierte LINT-Wert von `Convert.OUT` wird auf `ALI_OUT.D1` gelegt.

Die gesamte Verarbeitung ist **ereignisgesteuert** und erfolgt innerhalb eines einzigen Ausführungsschrittes (ohne zusätzliche Verzögerung).

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-Baustein** – er enthält keine eigene Logik, sondern delegiert die Konvertierung vollständig an den IEC-61131-Baustein `F_UDINT_TO_LINT`.
- **Adapter-orientierte Schnittstelle** – alle Ein- und Ausgaben erfolgen über standardisierte unidirektionale Adapter (`AUDI` und `ALI`), was eine einfache Integration in bestehende Adapter-Netzwerke ermöglicht.
- **Lizenz & Copyright** – Der Baustein unterliegt der **Eclipse Public License 2.0** und wurde von **HR Agrartechnik GmbH** entwickelt (Stand 2026).
- **Typsicherheit** – die Konvertierung von UDINT (unsigned 32 Bit) nach LINT (signed 64 Bit) ist verlustfrei, da der Wert einfach vorzeichenerweitert wird.
- **Paketname** – im CompilerInfo ist der Baustein unter `adapter::conversion::unidirectional` organisiert.

Zustandsübersicht

Als Composite-Baustein besitzt `AUDI_TO_ALI` keine eigenen Zustände. Die Zustandslogik liegt innerhalb des integrierten `F_UDINT_TO_LINT`-Bausteins, der nach jedem `REQ`-Ereignis in einen Verarbeitungszustand übergeht und mit `CNF` quittiert.

Anwendungsszenarien

- **Übergang zwischen unterschiedlichen Automatisierungsprotokollen** – wenn ein System UDINT-Werte über AUDI-Adapter liefert, das Zielsystem aber LINT-Werte über ALI-Adapter erwartet.
- **Datenanpassung in Adapter-Hierarchien** – Umleitung von 32-Bit-Zählerwerten (z. B. Drehzahl, Impulszähler) auf einen 64-Bit-Bus.
- **Migration** – Altanlagen mit UDINT-Bereichen können ohne Änderung der Adapterstruktur an neue 64-Bit-Schnittstellen angebunden werden.
- **Testumgebungen** – einfaches Einschleifen einer Typkonvertierung zwischen zwei Adaptern, ohne separate Konvertierungsbausteine im Netzwerk.

Vergleich mit ähnlichen Bausteinen

Baustein	Typ	Besonderheit
`F_UDINT_TO_LINT`	reiner Konverter	keine Adapter, direkte Daten-E/A; kein Ereignis-Handling über Adapter, erfordert eigene Ansteuerung.
`AUDI_TO_ALI` (dieser)	Adapter-Konverter	kapselt die Konvertierung in einer adapterbasierten Schnittstelle; Ereignisse und Daten werden automatisch über die Adapter weitergeleitet.
`ALI_TO_AUDI` (hypothetisch)	Adapter-Konverter	würde LINT rückwärts nach UDINT wandeln (potenziell verlustbehaftet).

Der **Hauptunterschied** zu einer direkten Verwendung von `F_UDINT_TO_LINT` ist die nahtlose Einbindung in Adapter-Netzwerke und die Vermeidung zusätzlicher Leitungen für Ereignis- und Datenverbindungen.

Fazit

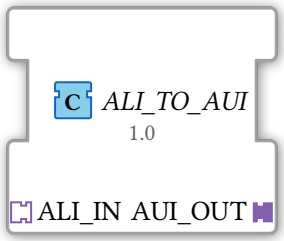
Der **AUDI_TO_ALI**-Funktionsblock ist ein praktischer Adapter-Konverter, der die Lücke zwischen einem UDINT-basierten AUDI-Adapter und einem LINT-basierten ALI-Adapter schließt. Durch die interne Verwendung des standardisierten IEC-61131-Konverters `F_UDINT_TO_LINT` ist die Umwandlung effizient und typsicher. Der Baustein eignet sich besonders für modulare Architekturen, bei denen unterschiedliche Adressbreiten aufeinander treffen, und vereinfacht die Wiederverwendung von Adapter-Schnittstellen in heterogenen Automatisierungsumgebungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

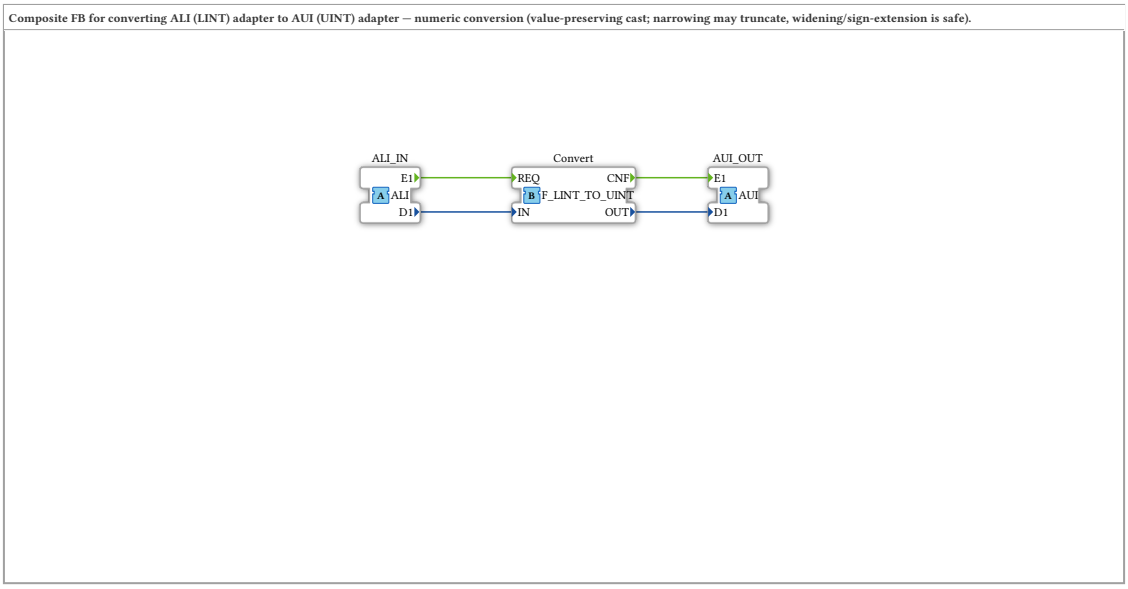
ALI_AUI

ALI_TO_AUI



LINT Adapter Input – *ALI*

AUI – UINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **ALI_TO_AUI** ist ein zusammengesetzter Baustein (Composite FB) zur Umwandlung eines ALI-Adapters (LINT-Datentyp) in einen AUI-Adapter (UINT-Datentyp). Er erlaubt die nahtlose Integration von Komponenten mit unterschiedlichen Adapterdefinitionen in einer 4diac-Umgebung. Der Baustein basiert auf dem internen Konvertierungsbaustein `F_LINT_TO_UINT` aus der IEC-61131-Bibliothek.

Schnittstellenstruktur

Ereignis-Eingänge

Nicht vorhanden – die Ereignissteuerung erfolgt über die Adapter.

Ereignis-Ausgänge

Nicht vorhanden – die Ereignissteuerung erfolgt über die Adapter.

Daten-Eingänge

Nicht vorhanden – die Datenübergabe erfolgt über die Adapter.

Daten-Ausgänge

Nicht vorhanden – die Datenübergabe erfolgt über die Adapter.

Adapter

Typ	Name	Kommentar
Socket	`ALI_IN`	Eingangsadapter vom Typ `adapter::types::unidirectional::ALI` – liefert LINT-Wert und assoziiertes Ereignis.
Plug	`AUI_OUT`	Ausgangsadapter vom Typ `adapter::types::unidirectional::AUI` – stellt den umgewandelten UINT-Wert und zugehöriges Ereignis bereit.

Funktionsweise

Der Baustein verbindet intern den Socket `ALI_IN` mit dem Eingang des Konvertierungsbausteins `F_LINT_TO_UINT`.

- Über den Ereignisausgang `E1` von `ALI_IN` wird der `REQ`-Eingang des Konverters angesteuert.
- Der Datenwert `D1` von `ALI_IN` (LINT) wird auf den `IN`-Eingang des Konverters gelegt.
- Nach erfolgreicher Konvertierung signalisiert der Konverter mit `CNF` das Ereignis an den Plug `AUI_OUT`, und der umgewandelte Wert (UINT) wird auf dessen `D1`-Ausgang geschrieben.

Der gesamte Ablauf ist ereignisgesteuert: Ein eingehendes Ereignis am ALI-Adapter löst die Umwandlung aus und erzeugt ein entsprechendes Ausgangsereignis am AUI-Adapter.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Datentypkonvertierung:** Der interne Baustein wandelt einen 64-Bit signed Integer (LINT) in einen 16-Bit unsigned Integer (UINT) um. Da der Wertebereich von UINT (0…65535) deutlich kleiner ist als der von LINT, kommt es bei Werten außerhalb dieses Bereichs zu einem Überlauf/Verlust.
- **Abhängigkeit:** Der Baustein benötigt die Bibliothek `iec61131::conversion` (insbesondere den Baustein `F_LINT_TO_UINT`).
- **Adapterreine Schnittstelle:** Der gesamte Datenaustausch erfolgt ausschließlich über Adapter, was eine modulare und wiederverwendbare Einbindung in größere Netzwerke ermöglicht.

Zustandsübersicht

Der Baustein selbst besitzt keinen eigenen Zustandsautomaten. Die Zustandslogik wird vollständig durch den internen Baustein `F_LINT_TO_UINT` abgebildet. Dieser verfügt typischerweise über die Zustände **IDLE** (wartet auf REQ) und **BUSY** (führt Konvertierung durch). Nach Abschluss wird das Ergebnis bereitgestellt und der CNF-Ereignisausgang aktiviert.

Anwendungsszenarien

- **Adapterbrücke:** Ein Sensor, der Daten im ALI-Format (LINT) bereitstellt, soll an einen Aktor angeschlossen werden, der einen AUI-Eingang (UINT) erwartet.
- **Systemintegration:** In heterogenen Automatisierungssystemen, in denen unterschiedliche Adapterdefinitionen verwendet werden, dient der Baustein als einfacher Konverter ohne manuelle Programmierung.
- **Prototypenbau:** Schnelles Ersetzen von Adaptern während der Entwicklungsphase, ohne die grundlegende Datenflusslogik ändern zu müssen.

Vergleich mit ähnlichen Bausteinen

Es existieren analoge Konverterbausteine wie `ALI_TO_UDI` (LINT ⟶ UDINT) oder `DINT_TO_UINT`. Der vorliegende Baustein spezialisiert sich auf die Kombination der unidirektionalen Adapter ALI und AUI. Im Gegensatz zu einer direkten Verbindung der Adapter über Datentypkonvertierung in der Netzwerkschicht bietet dieser Composite FB eine kompakte, wiederverwendbare Lösung mit klar definierten Ereignissignalen.

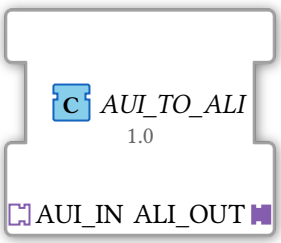
Fazit

Der ALI_TO_AUI-Baustein ist eine praktische und leicht verständliche Komponente zur Umwandlung zwischen ALI- und AUI-Adaptern. Seine Composite-Struktur kapselt die notwendige Datenkonvertierung und Ereignisweitergabe, sodass er sich nahtlos in bestehende 4diac-Projekte einfügt. Die einfache Handhabung und die klare Trennung der Schnittstellen machen ihn zu einem nützlichen Werkzeug bei der Adapter-Integration.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

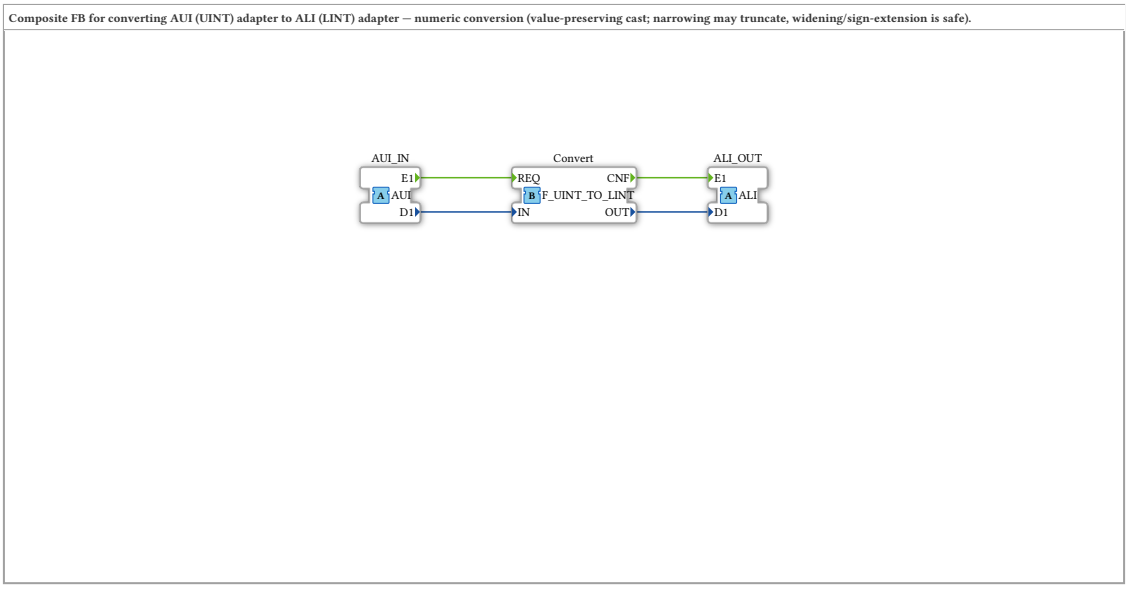
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUI_TO_ALI



UINT Adapter Input – *AUI*

ALI – LINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock AUI_TO_ALI ist ein Composite FB, der einen AUI-Adapter (UINT) in einen ALI-Adapter (LINT) umwandelt. Er nutzt intern den IEC 61131-Baustein F_UINT_TO_LINT zur Konvertierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **AUI_IN.E1** (Ereignis über Socket AUI_IN): Startet die Konvertierung, wenn ein Ereignis am Socket anliegt (über den AUI-Adapter).

Ereignis-Ausgänge

- **ALI_OUT.E1** (Ereignis über Plug ALI_OUT): Wird nach abgeschlossener Konvertierung ausgegeben (über den ALI-Adapter).

Daten-Eingänge

- **AUI_IN.D1** (Daten über Socket AUI_IN): UINT-Wert, der in einen LINT-Wert konvertiert werden soll.

Daten-Ausgänge

- **ALI_OUT.D1** (Daten über Plug ALI_OUT): Konvertierter LINT-Wert.

Adapter

- **Socket AUI_IN** (Typ: adapter::types::unidirectional::AUI): Empfängt UINT-Daten und zugehöriges Ereignis.
- **Plug ALI_OUT** (Typ: adapter::types::unidirectional::ALI): Sendet konvertierte LINT-Daten und zugehöriges Ereignis.

Funktionsweise

Der FB wartet auf ein Ereignis am Eingangsadapter AUI_IN (E1). Sobald dieses eintrifft, wird der anliegende UINT-Wert (D1) an den internen Konvertierungsbaustein F_UINT_TO_LINT weitergeleitet. Die Konvertierung erfolgt gemäß IEC 61131-Standard, wobei der UINT-Wert ohne Vorzeichen in einen LINT-Wert (64-Bit-Ganzzahl mit Vorzeichen) umgewandelt wird. Nach Abschluss der Konvertierung wird das Ausgangsereignis am ALI_OUT-Adapter (E1) ausgelöst und der konvertierte LINT-Wert an ALI_OUT.D1 übergeben.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).

- Der FB ist als Composite-Typ realisiert und kapselt die Konvertierungslogik in einem wiederverwendbaren Adapter-Interface.
- Verwendet den standardisierten Baustein F_UINT_TO_LINT aus der IEC 61131-Bibliothek.
- Lizenz: Eclipse Public License 2.0 (EPL-2.0).
- Entwickelt von HR Agrartechnik GmbH (Version 1.0, 2026-02-17).

Zustandsübersicht

Der FB besitzt keine eigene Zustandsmaschine, da er als Composite passiv die Zustände des internen Bausteins abbildet. Der interne F_UINT_TO_LINT arbeitet nach einem einfachen Schema: Warten auf das REQ-Ereignis, Verarbeitung der Konvertierung, Auslösen des CNF-Ereignisses.

Anwendungsszenarien

- Integration von einem System, das UINT-Werte über einen AUI-Adapter liefert, in ein System, das LINT-Werte über einen ALI-Adapter erwartet.
- Datenbrücke zwischen unterschiedlichen Adapter-Typen in einer verteilten Steuerung.
- Erweiterung von 4diac-Anwendungen, wenn eine Typkonvertierung zwischen UINT und LINT benötigt wird.

Vergleich mit ähnlichen Bausteinen

Es existieren analoge Adapter-Konvertierungsbausteine für andere Datentypen (z. B. UINT_TO_DINT, UINT_TO_REAL usw.). Der AUI_TO_ALI ist speziell für die Kombination der unidirektionalen Adapter AUI und ALI ausgelegt und ermöglicht eine einheitliche Adapter-Schnittstelle ohne zusätzliche manuelle Konvertierung.

Fazit

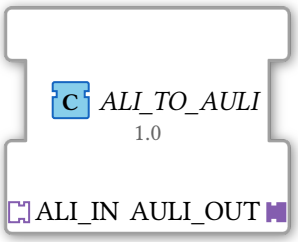
Der Composite FB AUI_TO_ALI bietet eine einfache und zuverlässige Möglichkeit, UINT-Daten über AUI-Adapter in LINT-Daten über ALI-Adapter zu konvertieren. Er ist gut gekapselt, nutzt Standardbausteine und kann leicht in 4diac-Projekte integriert werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

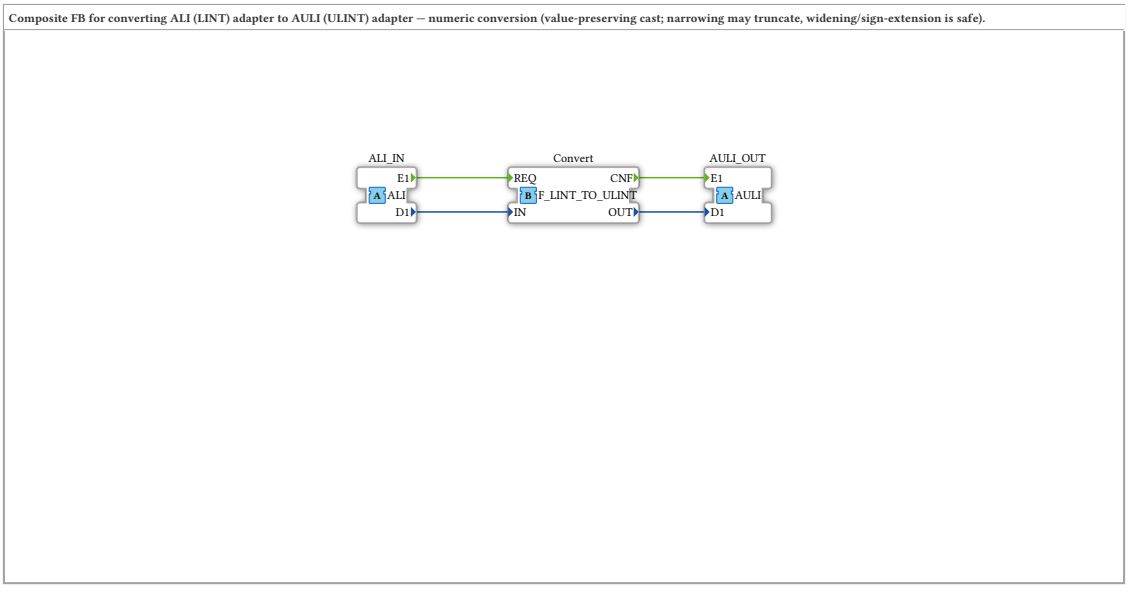
ALI_AULI

ALI_TO_AULI



LINT Adapter Input – *ALI*

AULI – ULINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `ALI_TO_AULI` ist ein Composite-Baustein, der eine bidirektionale Adapterkonvertierung zwischen dem LINT-Adapter (ALI) und dem ULINT-Adapter (AULI) realisiert. Er dient als standardisierte Schnittstelle, um einen mit Vorzeichen behafteten 64-Bit-Ganzzahlwert (LINT) in einen vorzeichenlosen 64-Bit-Ganzzahlwert (ULINT) umzuwandeln. Der Baustein kapselt den IEC-61131-3-Konvertierungsbaustein `F_LINT_TO_ULINT` und vereinfacht die Einbindung in adapterbasierte Architekturen.

Schnittstellenstruktur

Der Baustein besitzt keine eigenen Ereignis- oder Datenschnittstellen, sondern lediglich zwei Adapteranschlüsse (Socket und Plug). Die innere Logik erfolgt durch den eingebetteten Unterbaustein und die direkte Verbindung der Adaptersignale.

Ereignis-Eingänge

– Keine (Ereignisse werden über den ALI-Adapter empfangen)

Ereignis-Ausgänge

– Keine (Ereignisse werden über den AULI-Adapter gesendet)

Daten-Eingänge

– Keine (Daten werden über den ALI-Adapter empfangen)

Daten-Ausgänge

– Keine (Daten werden über den AULI-Adapter ausgegeben)

Adapter

Adaptertyp	Richtung	Name	Kommentar
`adapter::types::unidirectional::ALI`	Socket (Eingang)	`ALI_IN`	LINT-Adapter-Eingang (liefert Ereignis und einen LINT-Wert)
`adapter::types::unidirectional::AULI`	Plug (Ausgang)	`AULI_OUT`	ULINT-Adapter-Ausgang (gibt Ereignis und einen ULINT-Wert weiter)

Funktionsweise

Der Baustein arbeitet als reine Durchreiche-Konvertierung:

1. Ein eingehendes Ereignis am Socket `ALI_IN.E1` triggert den internen Baustein `Convert` (`F_LINT_TO_ULINT`) über dessen Ereigniseingang `REQ`.
2. Gleichzeitig wird der vom Socket bereitgestellte LINT-Wert (`ALI_IN.D1`) an den Dateneingang `IN` von `Convert` übergeben.
3. Nach erfolgreicher Konvertierung signalisiert `Convert` mit einem Ereignis an seinem Ausgang `CNF`, welches direkt an den Plug `AULI_OUT.E1` weitergeleitet wird.
4. Der konvertierte ULINT-Wert (`Convert.OUT`) wird parallel an den Dateneingang `D1` des AULI-Adapters übergeben.

Somit wird eine vollständige, ereignisgesteuerte und datensynchrone Umwandlung ohne zusätzliche Verzögerung oder Logik realisiert.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Verwendeter Unterbaustein**: `iec61131::conversion::F_LINT_TO_ULINT` – standardkonforme Typkonvertierung gemäß IEC 61131-3.
- **Adapter-Kopplung**: Der Baustein setzt auf das unidirektionale Adapter-Konzept von 4diac und ermöglicht die lose Kopplung zwischen verschiedenen Datentypschnittstellen.
- **Keine Zustandshaltung**: Als Composite-Baustein besitzt `ALI_TO_AULI` keinen eigenen internen Zustand; alle Entscheidungen werden durch den Unterbaustein getroffen.
- **Plattformunabhängigkeit**: Der Baustein kann in jeder 4diac-Umgebung eingesetzt werden, die die IEC-61131-Konvertierungsbausteine bereitstellt.

Zustandsübersicht

Der Baustein selbst führt keinen Zustandsautomaten. Sämtliche Zustandsübergänge werden vom internen `F_LINT_TO_ULINT` gesteuert, der nach einem Ereignis am Eingang `REQ` in den Zustand „konvertiere“ wechselt und nach Abschluss ein Bestätigungsereignis ausgibt. Da diese Zustände nicht nach außen sichtbar sind, wird auf eine detaillierte Darstellung verzichtet.

Anwendungsszenarien

- **Systemintegration**: Anbindung eines älteren Steuerungssystems, das LINT-Werte (vorzeichenbehaftet) verwendet, an eine moderne Komponente, die nur ULINT-Werte (vorzeichenlos) verarbeiten kann.
- **Datenkonvertierung in Adapterketten**: Einbettung zwischen zwei Adaptern, z. B. in einer Datenvorverarbeitungskette, die eine einheitliche ULINT-Schnittstelle erfordert.
- **Migration**: Ersatz von LINT-basierten Bausteinen durch ULINT-Äquivalente, ohne die gesamte Adapterinfrastruktur ändern zu müssen.

Vergleich mit ähnlichen Bausteinen

In der Bibliothek existieren analoge Konverter für andere Datentypen, z. B. `DINT_TO_UDINT` oder `REAL_TO_LREAL`. `ALI_TO_AULI` unterscheidet sich durch die spezifische Adapter-Schnittstelle und die Kapselung des Konvertierungsbausteins in einem Composite. Während die reinen IEC-Bausteine einzelne Daten- und Ereignisschnittstellen besitzen, ermöglicht dieser Baustein eine direkte Adapter-zu-Adapter-Konvertierung, was den Entwurf modulärer und austauschbarer Komponenten erleichtert.

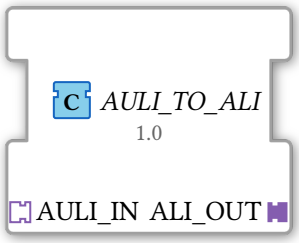
Fazit

Der `ALI_TO_AULI`-Baustein bietet eine einfache und zuverlässige Möglichkeit, LINT-Adapter in ULINT-Adapter umzuwandeln. Er kapselt die Standardkonvertierung sauber in einer adapterbasierten Architektur und trägt so zur Wiederverwendbarkeit und Interoperabilität von 4diac-Anwendungen bei. Durch die Verwendung des bewährten IEC-Bausteins ist die Konvertierung korrekt und plattformunabhängig.

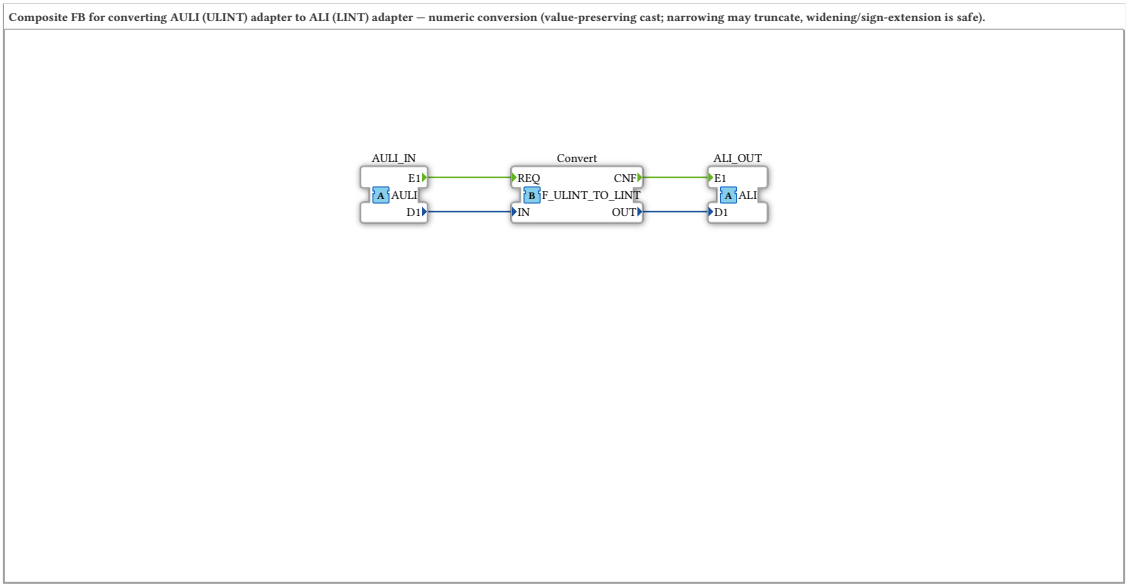
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AULI_TO_ALI



ULINT Adapter Input – *AULI* *ALI* – LINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AULI_TO_ALI** ist ein Composite-Baustein, der eine Adapter-Konvertierung von **AULI** (ULINT-Adapter) nach **ALI** (LINT-Adapter) durchführt. Er dient dazu, einen vorzeichenlosen 64-Bit-Ganzzahlwert (ULINT) in einen vorzeichenbehafteten 64-Bit-Ganzzahlwert (LINT) zu wandeln und über die entsprechenden Adapter-Schnittstellen weiterzuleiten.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignisse werden über den eingehenden Adapter **AULI_IN** empfangen:

- `E1` – Startet die Konvertierung des anliegenden ULINT-Wertes.

Ereignis-Ausgänge

Ereignisse werden über den ausgehenden Adapter **ALI_OUT** gesendet:

- `E1` – Quittiert die abgeschlossene Konvertierung und signalisiert, dass der LINT-Wert am Ausgang anliegt.

Daten-Eingänge

Daten werden über den eingehenden Adapter **AULI_IN** bereitgestellt:

- `D1` (Datentyp: ULINT) – Der zu konvertierende vorzeichenlose 64-Bit-Wert.

Daten-Ausgänge

Daten werden über den ausgehenden Adapter **ALI_OUT** ausgegeben:

- `D1` (Datentyp: LINT) – Der konvertierte vorzeichenbehaftete 64-Bit-Wert.

Adapter

- **AULI_IN** (Socket) – Unidirektionaler Adapter vom Typ `adapter::types::unidirectional::AULI` zur Aufnahme von ULINT Werten und zugehörigen Ereignissen.
- **ALI_OUT** (Plug) – Unidirektionaler Adapter vom Typ `adapter::types::unidirectional::ALI` zur Ausgabe von LINT Werten und zugehörigen Ereignissen.

Funktionsweise

Der Baustein verwendet intern den IEC-61131-Konvertierungsbaustein `F_ULINT_TO_LINT`. Sobald am **AULI_IN**-Adapter das Ereignis `E1` eintrifft, wird der an `D1` anliegende ULINT-Wert gelesen und an den Konverter übergeben. Nach erfolgter Wandlung wird der resultierende LINT-Wert an den **ALI_OUT**-Adapter weitergegeben und gleichzeitig das Ereignis `E1` an dessen Ausgang gesendet. Die Daten- und Ereignisflüsse sind strikt synchronisiert: Eine Konvertierung erfolgt nur auf Anforderung über das Ereignis `E1`.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Adapterbasierte Kapselung**: Der Baustein kapselt die Konvertierungslogik vollständig hinter standardisierten unidirektionalen Adaptern (AULI/ALI), sodass er leicht in bestehende Adapter-basierten Datenflüsse integriert werden kann.
- **Composite-Aufbau**: Da es sich um einen Composite-FB handelt, wird keine eigene Zustandsmaschine benötigt; die Logik wird durch die Verknüpfung des Unterbausteins `F_ULINT_TO_LINT` realisiert.
- **Typensicherheit**: Die Wandlung von ULINT nach LINT ist wertebereichserhaltend, solange der ULINT-Wert nicht größer als `LINT#9223372036854775807` (maximaler LINT) ist. Bei Überschreitung kommt es zu einem undefinierten Verhalten (keine Fehlerbehandlung im Baustein).

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Er reagiert ereignisgesteuert:

- **Warten auf Ereignis `E1`**: Keine Aktivität.
- **Während der Konvertierung**: Der interne FB `F_ULINT_TO_LINT` wird ausgeführt (die Dauer ist taktzyklenabhängig).
- **Ausgabe nach `CNF`**: Das Ereignis `E1` am Ausgang wird gesendet.

Anwendungsszenarien

- **Datentypanpassung in Steuerungssystemen** wenn ein Sensor oder eine Quelle ULINT-Werte liefert, das nachfolgende System aber LINT-Werte (vorzeichenbehaftet) erwartet.
- **Adapter-Brücke** zwischen Komponenten, die ausschließlich über AULI- bzw. ALI-Adapter kommunizieren, z. B. in modularen Automatisierungsarchitekturen.
- **Einbindung in geräteübergreifende Datenflüsse** ohne Änderung der eigentlichen Applikationslogik.

Vergleich mit ähnlichen Bausteinen

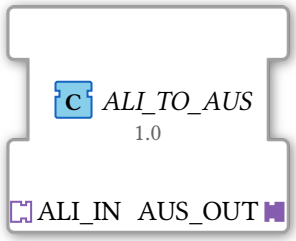
Ähnliche Adapter-Konvertierungen existieren für andere Datentypen, z. B. `UINT_TO_INT` oder `UDINT_TO_DINT`. Der **AULI_TO_ALI** spezialisiert sich auf die 64-Bit-Ganzzahlen ULINT/LINT und verwendet die gleiche Adapter-Schnittstelle (unidirektional), sodass er nahtlos in bestehende Adapter-Verschaltungen eingefügt werden kann.

Fazit

Der **AULI_TO_ALI**-Baustein ist eine einfache, aber effektive Lösung zur Wandlung von ULINT- in LINT-Werte unter Verwendung standardisierter Adapter. Er vereinfacht die Integration unterschiedlicher Datentypen in unidirektionale Kommunikationspfade und eignet sich besonders für modulare IEC-61499-Anwendungen.

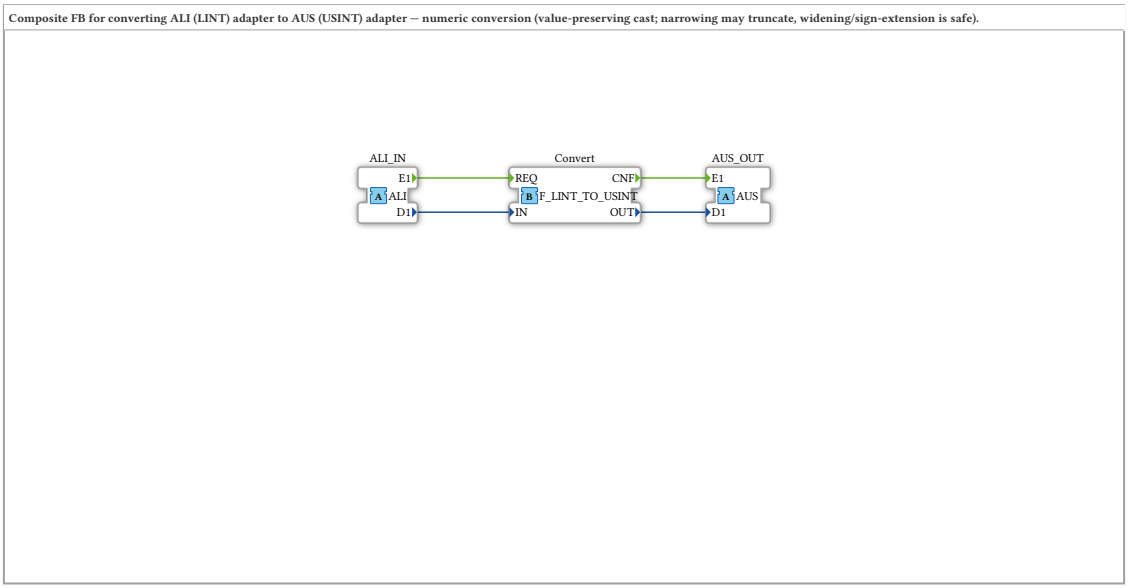
ALI_AUS

ALI_TO_AUS



LINT Adapter Input – *ALI*

AUS – USINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **ALI_TO_AUS** ist ein zusammengesetzter (Composite) Funktionsbaustein nach IEC 61499, der zur Umwandlung eines **ALI-Adapters** (Daten vom Typ **LINT**, Long Integer) in einen **AUS-Adapter** (Daten vom Typ **USINT**, Unsigned Short Integer) dient. Der Baustein kapselt die notwendige Konvertierungslogik, um eine einfache und typensichere Verbindung zwischen unterschiedlich typisierten Adapter-Schnittstellen zu ermöglichen.

Schnittstellenstruktur

Ereignis-Eingänge

- **ALI_IN.E1** (über Adapter-Socket) – löst die Konvertierung aus, sobald ein Ereignis von der angeschlossenen ALI-Quelle ankommt.

Ereignis-Ausgänge

- **AUS_OUT.E1** (über Adapter-Plug) – signalisiert den Abschluss der Konvertierung an die nachgeschaltete AUS-Senke.

Daten-Eingänge

- **ALI_IN.D1** (über Adapter-Socket) – der zu konvertierende LINT-Wert (Long Integer).

Daten-Ausgänge

- **AUS_OUT.D1** (über Adapter-Plug) – der konvertierte USINT-Wert (Unsigned Short Integer).

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	ALI_IN	adapter::types::unidirectional::ALI	Empfängt den LINT-Datenwert und das zugehörige Ereignis
Plug (Ausgang)	AUS_OUT	adapter::types::unidirectional::AUS	Stellt den konvertierten USINT-Wert und das Bestätigungsereignis bereit

Funktionsweise

Der Baustein enthält intern eine Instanz des Funktionsbausteins **F_LINT_TO_USINT** aus der IEC 61131-Konvertierungsbibliothek. Der Ablauf ist:

1. Ein Ereignis **ALI_IN.E1** triggert die Datenübernahme.
2. Der an **ALI_IN.D1** anliegende LINT-Wert wird an den Eingang **IN** des internen Konverters weitergeleitet.
3. Der interne Baustein führt die Typumwandlung von LINT nach USINT durch (Beachtung von Wertebereichsgrenzen).
4. Nach erfolgreicher Konvertierung erzeugt der interne Baustein ein **CNF**-Ereignis, das als **AUS_OUT.E1** ausgegeben wird.
5. Das konvertierte Ergebnis gelangt von **OUT** des internen Bausteins an **AUS_OUT.D1**.

Die Kopplung von Ereignis- und Datenflüssen erfolgt vollständig über das interne Netzwerk; der Anwender sieht nur die beiden Adapter-Schnittstellen.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Wiederverwendbarer Konverter**: Der Baustein setzt auf den standardisierten **F_LINT_TO_USINT** (iec61131::conversion), sodass die Konvertierungslogik nicht neu implementiert werden muss.
- **Adapter-basiert**: Die gesamte Schnittstelle ist als Uni-Directional-Adapter realisiert, vereinfacht den Austausch und die Wiederverwendung in größeren Systemen.
- **Lizenz**: Der Baustein wird unter der **Eclipse Public License 2.0** bereitgestellt (EPL-2.0).

Zustandsübersicht

Da es sich um einen Composite-Baustein handelt, existiert keine explizite Zustandsmaschine auf oberster Ebene. Der interne Konverter **F_LINT_TO_USINT** arbeitet ereignisgesteuert und führt nach jedem **REQ**-Ereignis genau einen Konvertierungsschritt durch. Solange kein Ereignis anliegt, bleibt der Baustein inaktiv.

Anwendungsszenarien

- **Schnittstellenanpassung**: Verbindung von Komponenten, die mit **LINT** (z. B. hochauflösende Zähler) arbeiten, mit solchen, die **USINT** (z. B. einfache Aktorsteuerungen) erwarten.
- **Systemintegration**: Wenn ein vorgegebener Adapter-Typ (ALI) vorhanden ist und ein anderer Adapter-Typ (AUS) benötigt wird – z. B. beim Zusammenschluss von Modulen verschiedener Hersteller.
- **Protokoll-Wandlung**: Einsatz in Datenvorverarbeitungsketten, bei denen lange Integerwerte auf einen kleineren Bereich abgebildet werden müssen.

Vergleich mit ähnlichen Bausteinen

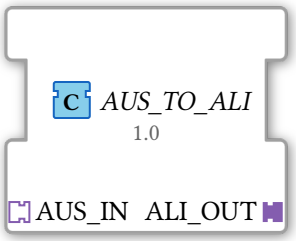
Es existieren zahlreiche datentyp-spezifische Konvertierungsbausteine (z. B. LINT_TO_INT, LINT_TO_DINT, USINT_TO_LINT). Der **ALI_TO_AUS** hebt sich hervor, indem er nicht nur die reine Typumwandlung kapselt, sondern diese direkt in eine **Adapter-Schnittstelle** einbettet. Dies reduziert den Verdrahtungsaufwand auf der Blockebene und erhöht die Lesbarkeit des Gesamtentwurfs.

Fazit

Der **ALI_TO_AUS** ist ein praktischer und robuster Baustein zur Umwandlung von LINT-Adapter-Daten in USINT-Adapter-Daten. Durch die Nutzung eines standardisierten Konverters und die klare

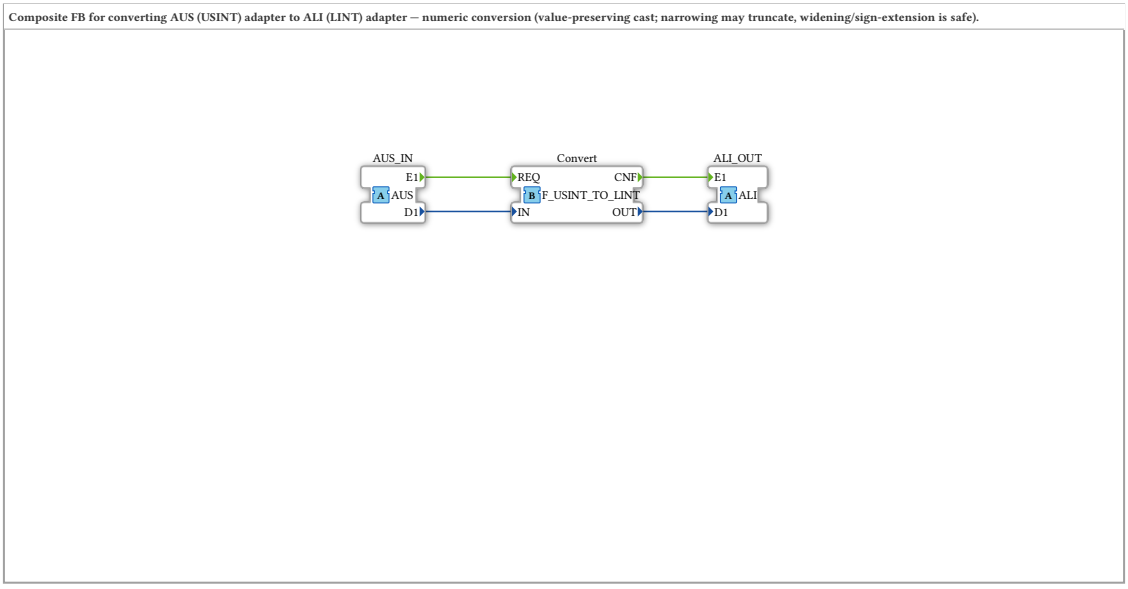
Adapter-Kapselung wird eine wiederverwendbare, typensichere und wartungsfreundliche Lösung für unterschiedliche Steuerungssysteme bereitgestellt.

AUS_TO_ALI



USINT Adapter Input – *AUS*

ALI – LINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AUS_TO_ALI` ist ein zusammengesetzter Baustein (Composite FB), der einen AUS-Adapter (USINT-Werte) in einen ALI-Adapter (LINT-Werte) umwandelt. Er kapselt den Konvertierungsvorgang und stellt eine unidirektionale Schnittstelle zwischen diesen beiden Adaptertypen bereit. Intern wird der Funktionsblock `F_USINT_TO_LINT` aus der IEC 61131-Bibliothek verwendet.

Schnittstellenstruktur

Ereignis-Eingänge

Keine eigenen Ereignis-Eingänge. Die Ereignissteuerung erfolgt über den `AUS_IN`-Adapter (Socket).

Ereignis-Ausgänge

Keine eigenen Ereignis-Ausgänge. Die Ereignisausgabe erfolgt über den `ALI_OUT`-Adapter (Plug).

Daten-Eingänge

Keine eigenen Daten-Eingänge. Die Datenaufnahme erfolgt über den `AUS_IN`-Adapter.

Daten-Ausgänge

Keine eigenen Daten-Ausgänge. Die Datenausgabe erfolgt über den `ALI_OUT`-Adapter.

Adapter

- **AUS_IN** (Socket) – Empfängt einen AUS-Adapter mit unidirektionalem USINT-Datentyp. Enthält:
  - Ereignis-Eingang `E1`
  - Daten-Eingang `D1` (USINT)

- **ALI_OUT** (Plug) – Sendet einen ALI-Adapter mit unidirektionalem LINT-Datentyp. Enthält:
  - Ereignis-Ausgang `E1`
  - Daten-Ausgang `D1` (LINT)

Funktionsweise

Der Baustein arbeitet ereignisgesteuert:

1. Ein Ereignis am `E1`-Eingang des `AUS_IN`-Adapters initiiert die Verarbeitung.
2. Der aktuelle Wert von `AUS_IN.D1` (USINT) wird an den internen Baustein `Convert` (`F_USINT_TO_LINT`) übergeben.
3. Nach der Konvertierung wird das Ergebnis (LINT) über `ALI_OUT.D1` ausgegeben und gleichzeitig ein Ereignis am `E1`-Ausgang des `ALI_OUT`-Adapters gesendet.

Die Typkonvertierung erfolgt gemäß der IEC 61131-Definition von `F_USINT_TO_LINT`: USINT (0..255) wird ohne Vorzeichenverlust in LINT (64-Bit) abgebildet.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-Baustein**: Die Konvertierungslogik ist gekapselt und wiederverwendbar.
- **Adapter-basiert**: Die Ein-/Ausgabe erfolgt ausschließlich über Adapter (`AUS` und `ALI`), was die Integration in adapterorientierte Architekturen erleichtert.
- **Unidirektional**: Der Baustein unterstützt nur eine Datenflussrichtung (AUS → ALI).
- **Bibliotheksimport**: Er verwendet den Baustein `iec61131::conversion::F_USINT_TO_LINT` aus der IEC 61131-Bibliothek.
- **Paketstruktur**: Der Baustein ist im Package `adapter::conversion::unidirectional` organisiert.

Zustandsübersicht

Der `AUS_TO_ALI`-Baustein besitzt keine eigenen Zustandsautomaten. Der interne Ablauf ist rein ereignisgesteuert und zustandslos:

- **Warte**: auf ein Ereignis vom `AUS_IN.E1`
- **Konvertiere**: sofortige Umwandlung des USINT-Werts in LINT
- **Ausgabe**: unverzögertes Senden des Ergebnisses über `ALI_OUT`

Jede Konvertierung ist unabhängig von vorherigen Aufrufen.

Anwendungsszenarien

- **Schnittstellenanpassung**: Ein System, das Daten als AUS-Adapter (USINT) bereitstellt, muss an einen Empfänger angebunden werden, der einen ALI-Adapter (LINT) erwartet.
- **Datenaufbereitung**: Umwandlung von Sensordaten oder Stellgrößen mit geringer Auflösung (USINT) in einen breiteren Wertebereich (LINT) für weiterführende Berechnungen.
- **Adapter-Kaskadierung**: Als Teil einer Kette von Adapterkonvertierungen, z. B. in einer modularen Steuerungsarchitektur.

Vergleich mit ähnlichen Bausteinen

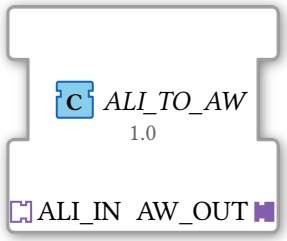
- **Direkte Verwendung von `F_USINT_TO_LINT`**: Dieser Baustein arbeitet auf Datenebene ohne Adapter. Der `AUS_TO_ALI` kapselt zusätzlich die Adapterlogik und ist daher besser für adapterbasierte Kommunikation geeignet.
- **Weitere Adapterkonvertierungen**: Für andere Typkombinationen (z. B. `INT_TO_DINT`, `BYTE_TO_WORD`) existieren analoge Composite-Bausteine, die demselben Entwurfsmuster folgen.

Fazit

Der `AUS_TO_ALI`-Baustein ist eine einfache, aber effektive Lösung zur unidirektionalen Konvertierung zwischen USINT- und LINT-Adaptern. Er erhöht die Wiederverwendbarkeit und Klarheit in Systemen, die auf Adapter-Schnittstellen setzen, und vermeidet die manuelle Verkabelung von Konvertierungslogik.

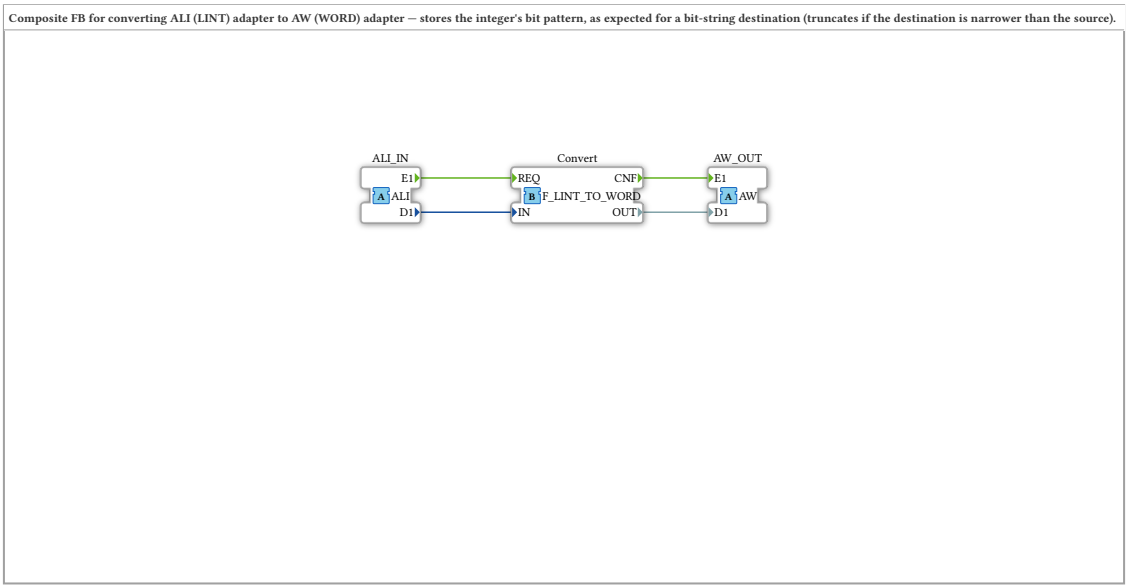
ALI_AW

ALI_TO_AW



LINT Adapter Input – *ALI*

AW – WORD Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ALI_TO_AW** ist ein zusammengesetzter Baustein (Composite FB), der einen LINT-Wert (ALI-Adapter) in einen WORD-Wert (AW-Adapter) umwandelt. Er kapselt die Konvertierungslogik des IEC-61131-3-Funktionsbausteins `F_LINT_TO_WORD` in einer adapterbasierten Schnittstelle. Der Baustein ist für den Einsatz in unidirektionalen Adapter-Kommunikationspfaden vorgesehen und ermöglicht eine saubere Trennung von Datenformat und Anwendung.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Datenein-/ausgänge auf FB-Ebene. Die Kommunikation erfolgt ausschließlich über zwei Adapter (einen Socket und einen Plug).

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Das eingehende Ereignis wird über den Adapter `ALI_IN.E1` bereitgestellt.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Das ausgehende Ereignis wird über den Adapter `AW_OUT.E1` bereitgestellt.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden. Der LINT-Wert wird über den Adapter `ALI_IN.D1` bezogen.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden. Der WORD-Wert wird über den Adapter `AW_OUT.D1` ausgegeben.

Adapter

Adapter-Name	Typ	Richtung	Kommentar
`ALI_IN`	`adapter::types::unidirectional::ALI`	Socket (Eingang)	LINT-Adapter-Eingang; liefert Ereignis auf `E1` und LINT-Daten auf `D1`
`AW_OUT`	`adapter::types::unidirectional::AW`	Plug (Ausgang)	WORD-Adapter-Ausgang; sendet Ereignis auf `E1` und WORD-Daten auf `D1`

Funktionsweise

1. Der Baustein erwartet über den Socket `ALI_IN` ein Ereignis (an `E1`) sowie einen zugehörigen LINT-Datenwert (an `D1`).
2. Das Ereignis wird an den internen Konvertierungsbaustein `F_LINT_TO_WORD` (Typ `iec61131::conversion::F_LINT_TO_WORD`) weitergeleitet (Verbindung `ALI_IN.E1` ⟶ `Convert.REQ`). Gleichzeitig wird der LINT-Wert an den Dateneingang `IN` des Konverters übergeben (Verbindung `ALI_IN.D1` ⟶ `Convert.IN`).
3. Der interne Baustein führt die Wandlung durch und signalisiert die Fertigstellung über seinen Ausgang `CNF`.
4. Dieses Ereignis wird an den Plug `AW_OUT.E1` weitergegeben. Der konvertierte WORD-Wert (Ausgang `OUT` des Konverters) wird parallel an den Dateneingang `AW_OUT.D1` angelegt.

Somit stellt der FB die vom Eingangsadapter bereitgestellten Daten in konvertierter Form am Ausgangsadapter zur Verfügung, gesteuert durch die Ereigniskette.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **IEC-61131-3-Konvertierung**: Die eigentliche Umwandlung erfolgt mittels des standardisierten Funktionsbausteins `F_LINT_TO_WORD`, der eine verlustbehaftete Konvertierung von 64-Bit (LINT) auf 16-Bit (WORD) durchführt (Abschneiden oder Bereichsprüfung abhängig von der Implementierung).
- **Adapterbasierte Schnittstelle**: Der Baustein ist vollständig in die Adaptertechnologie von 4diac eingebettet und eignet sich für unidirektionale Kommunikationspfade (package `adapter::conversion::unidirectional`).
- **Versionierung**: Version 1.0, bereitgestellt unter der Eclipse Public License 2.0 (EPL-2.0).
- **Keine eigenen Zustände**: Der FB delegiert die Zustandslogik vollständig an den inneren `F_LINT_TO_WORD`.

Zustandsübersicht

Da es sich um einen zusammengesetzten Baustein handelt, besitzt `ALI_TO_AW` keinen eigenen Zustandsautomaten. Der interne Konverter `F_LINT_TO_WORD` realisiert typischerweise folgende implizite Zustände:

Zustand	Beschreibung
IDLE	Warten auf ein Ereignis am Eingang `ALI_IN.E1`.
CONVERT	Nach Empfang von `REQ` wird die Konvertierung durchgeführt.
DONE	Nach Abschluss der Konvertierung wird `CNF` aktiviert und der Ergebniswert am Ausgang bereitgestellt.

Anschließend kehrt der interne Baustein in den IDLE-Zustand zurück.

Anwendungsszenarien

- **Steuerungstechnik**: Umwandlung eines 64-Bit Zählerstands (z. B. von einem Encoder) in einen 16-Bit Wert für eine einfache Anzeige oder eine Wort-basierte Schnittstelle.
- **Datenkonsolidierung**: Einsatz in Adapter-Ketten, wenn ein nachfolgender Baustein nur WORD-Eingänge akzeptiert, die Quelle aber LINT-Daten liefert.

- **Test- und Simulationsumgebungen**: Erzeugung von WORD-Daten aus LINT-Testwerten zur Überprüfung von Adapter-Kommunikationspfaden.

Vergleich mit ähnlichen Bausteinen

- **LINT_TO_DWORD / LINT_TO_BYTE**: Andere Konvertierungen mit unterschiedlicher Zielbreite; `ALI_TO_AW` ist speziell auf die Kombination der unidirektionalen Adapter `ALI` und `AW` zugeschnitten.
- **Direkter Einsatz von F_LINT_TO_WORD**: `ALI_TO_AW` bietet eine höhere Abstraktionsebene durch Adapter-Schnittstellen und erleichtert die Einbindung in adapterbasierte Architekturen, während der reine Funktionsbaustein nur über Ereignis- und Datenschnittstellen verfügt.

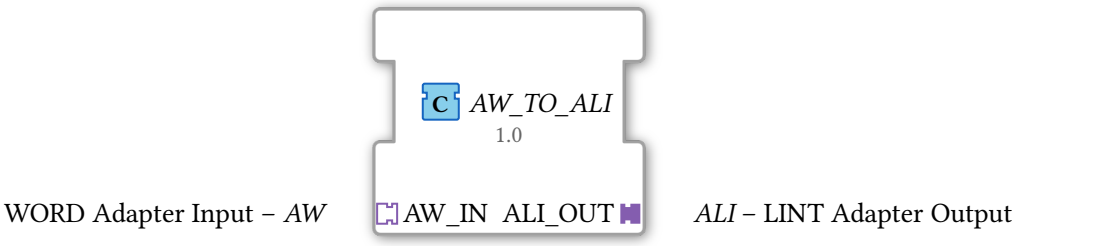
Fazit

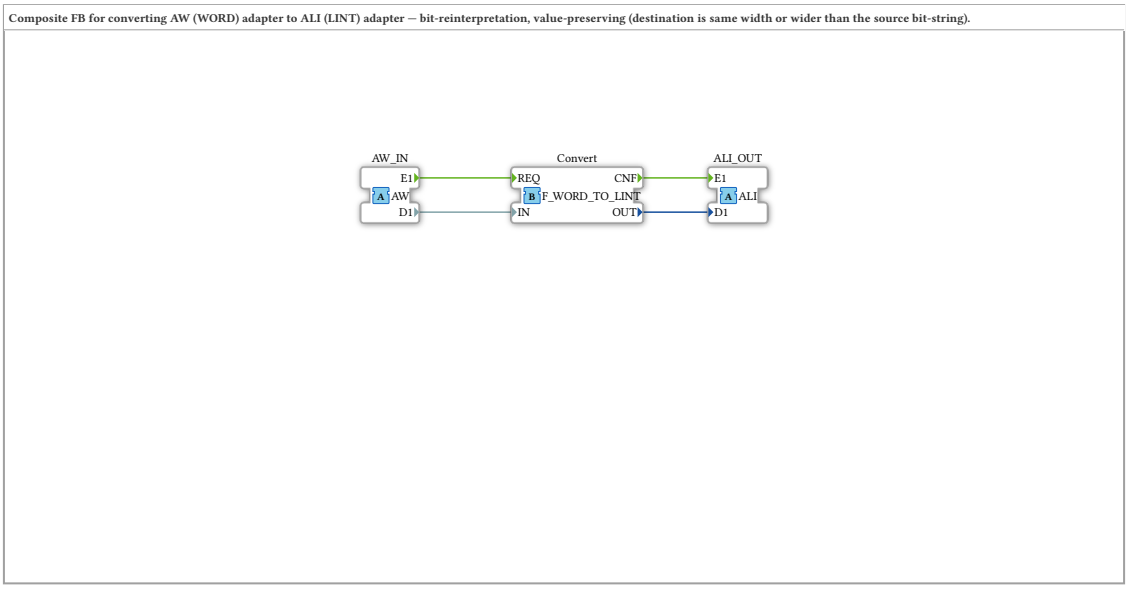
Der Funktionsblock `ALI_TO_AW` ist ein praktischer und wiederverwendbarer Baustein zur Konvertierung von LINT- auf WORD-Adaptern. Er kapselt die typische IEC-61131-3-Wandlung in einer adapterfreundlichen Hülle und vereinfacht so die Integration in bestehende oder neue 4diac-Projekte. Seine klare unidirektionale Schnittstelle und die Einhaltung des Eclipse-Public-License-Modells machen ihn zu einer soliden Komponente für Automatisierungslösungen, die unterschiedliche Datenbreiten verbinden müssen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AW_TO_ALI





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AW_TO_ALI** ist ein zusammengesetzter (Composite) Baustein, der einen unidirektionalen Adapter vom Typ **AW** (WORD) in einen Adapter vom Typ **ALI** (LINT) umwandelt. Er wird eingesetzt, um Daten zwischen verschiedenen Adapterschnittstellen zu konvertieren, ohne dass der Anwender die eigentliche Konvertierungslogik implementieren muss. Der Baustein nutzt intern den

IEC 61131-Funktionsbaustein `F_WORD_TO_LINT` und stellt die typische Ereignis-/Datensteuerung über die mitgelieferten Adapter-Schnittstellen bereit.

Schnittstellenstruktur

Ereignis-Eingänge
Über den **Socket AW_IN** (Typ: `adapter::types::unidirectional::AW`) wird ein Ereignis vom Typ **E1** hereingeführt. Dieses Ereignis löst die Konvertierung des anliegenden Datenwerts aus.

Ereignis-Ausgänge
Über den **Plug ALI_OUT** (Typ: `adapter::types::unidirectional::ALI`) wird nach erfolgreicher Konvertierung ein Ereignis **E1** ausgegeben. Es signalisiert, dass der konvertierte Wert am Datenausgang bereitsteht.

Daten-Eingänge
Eingehende Daten werden über den **Socket AW_IN** geführt:

- **D1** (Datentyp: WORD) – der zu konvertierende 16-Bit-Wert.

Daten-Ausgänge
Die konvertierten Daten werden über den **Plug ALI_OUT** ausgegeben:

- **D1** (Datentyp: LINT) – der konvertierte 64-Bit-Wert (Long Integer).

Adapter
Der Baustein besitzt zwei Adapter-Schnittstellen:

- **Socket AW_IN** – Unidirektionaler Eingangsadapter (Typ AW), der die Verbindung zu einem vorherigen Baustein herstellt, der WORD-Daten liefert.
- **Plug ALI_OUT** – Unidirektionaler Ausgangsadapter (Typ ALI), der die konvertierten LINT-Daten an nachfolgende Bausteine weitergibt.

Beide Adapter sind unidirektional, d. h. sie übertragen Ereignisse und Daten nur in eine Richtung (Eingang → Ausgang).

Funktionsweise
Der Ablauf innerhalb des Composite-Bausteins ist streng sequentiell:

1. Ein Ereignis **E1** am Socket **AW_IN** wird an den internen Funktionsbaustein **F_WORD_TO_LINT** (Instanzname *Convert*) weitergeleitet und aktiviert dessen **REQ-Eingang**.
2. Gleichzeitig wird der eingehende Datenwert **D1** (WORD) an den **IN-Eingang** des Konverters gelegt.
3. Der Konverter `F_WORD_TO_LINT` führt die Typumwandlung von WORD nach LINT durch und stellt das Ergebnis an seinem **OUT-Ausgang** bereit.
4. Nach Abschluss der Konvertierung erzeugt der Konverter ein **CNF-Ereignis**, das an den **E1-Eingang** des Ausgangsadapters **ALI_OUT** weitergegeben wird.
5. Gleichzeitig wird der konvertierte Wert (LINT) vom **OUT-Ausgang** des Konverters auf den Datenausgang **D1** des **ALI_OUT-Adapters** gelegt.

Damit ist die Datenkonvertierung zu einem sicheren, ereignisgesteuerten Zeitpunkt abgeschlossen.

Technische Besonderheiten

- **Bit-Reinterpretation, werterhaltend** (Zielbreite ist gleich groß oder größer als der Quell-Bit-String).
- **Composite-Baustein** – Die Konvertierungslogik ist vollständig in einem inneren Netzwerk gekapselt, das nur aus einem einzigen Konvertierungs-FB besteht. Der Baustein besitzt keinen eigenen ECC-Zustandsautomaten.
- **Adapter-basierte Schnittstelle** – Statt einzelner Ereignis-/Daten-Ein-/Ausgänge werden Adapter verwendet. Dies ermöglicht eine modulare, wieder verwendbare Verbindung in Adapter-basierten Steuerungsarchitekturen.
- **Paket-Struktur** – Der Baustein ist im Paket `adapter::conversion::unidirectional` abgelegt und verwendet den Konverter `iec61131::conversion::F_WORD_TO_LINT` aus der IEC-61131-Konvertierungsbibliothek.

- **Unidirektionale Richtung** – Daten fließen nur vom Eingangsadapter zum Ausgangsadapter; Rückwärtskommunikation ist nicht vorgesehen.
- **Lizenz** – Der Quellcode unterliegt der Eclipse Public License 2.0 (EPL-2.0), entwickelt von der HR Agrartechnik GmbH.

Zustandsübersicht

Da es sich um einen zusammengesetzten Baustein ohne eigenen Zustandsautomaten (ECC) handelt, gibt es keine definierten Zustände. Die Funktionalität wird vollständig durch den eingebetteten Funktionsbaustein `F_WORD_TO_LINT` gesteuert, der eine einmalige, ereignisgesteuerte Umwandlung durchführt. Der Baustein ist immer dann bereit, wenn ein Ereignis am Eingangsadapter eintrifft; nach der Konvertierung wartet er auf das nächste Ereignis.

Anwendungsszenarien

- **Datenkonvertierung in Adapterketten** – Wenn ein Steuerungssystem auf unidirektionalen Adaptern basiert und eine Komponente WORD-Daten liefert, eine nachfolgende Komponente jedoch LINT-Daten erwartet, kann AW_TO_ALI als Zwischenglied eingefügt werden.
- **Anbindung von Feldgeräten mit unterschiedlichen Bitbreiten** – Z. B. die Integration eines Sensors, der 16-Bit-Messwerte (WORD) liefert, in eine Logik, die mit 64-Bit-Werten (LINT) arbeitet (etwa für hochauflösende Zähler oder Zeitstempel).
- **Typanpassung in Bibliotheken** – Erweiterung einer bestehenden Adapterbibliothek um komfortable Konvertierungsbausteine, um die Kompatibilität zwischen verschiedenen Datenformaten zu gewährleisten.

Vergleich mit ähnlichen Bausteinen

Baustein	Eingang(sadapter)	Ausgang(sadapter)	Konvertierung	Besonderheiten
AW_TO_ALI	AW (WORD)	ALI (LINT)	WORD ⟶ LINT	Unidirektional, Composite
`F_WORD_TO_LINT` (IEC 61131)	Einzelner REQ-Eingang, IN (WORD)	Einzelner CNF-Ausgang, OUT (LINT)	WORD ⟶ LINT	Standard-FB, kein Adapter
`ALI_TO_AW` (hypothetisch)	ALI (LINT)	AW (WORD)	LINT ⟶ WORD	Umgekehrte Richtung

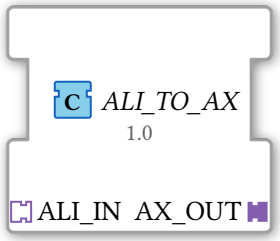
Der Vorteil von AW_TO_ALI liegt in der nahtlosen Integration in adapterbasierte Systeme, während der reine IEC-Baustein `F_WORD_TO_LINT` separat verdrahtet werden müsste. Bausteine mit Adapter-Schnittstellen vereinfachen den Steuerungsentwurf, da sie standardisierte Stecker (Plugs/Sockets) bieten.

Fazit

AW_TO_ALI ist ein kompakter, adapter-basierter Konvertierungsbaustein für die Umwandlung von WORD- in LINT-Daten. Durch die Nutzung des bewährten IEC-61131-Konverters `F_WORD_TO_LINT` und die Kapselung in eine Composite-Struktur bietet er eine saubere, ereignisgesteuerte Schnittstelle für unidirektionale Adapteranwendungen. Der Baustein eignet sich besonders für den Einsatz in modularen Automatisierungssystemen, in denen Daten zwischen Komponenten mit unterschiedlichen Datenbreiten ausgetauscht werden müssen.

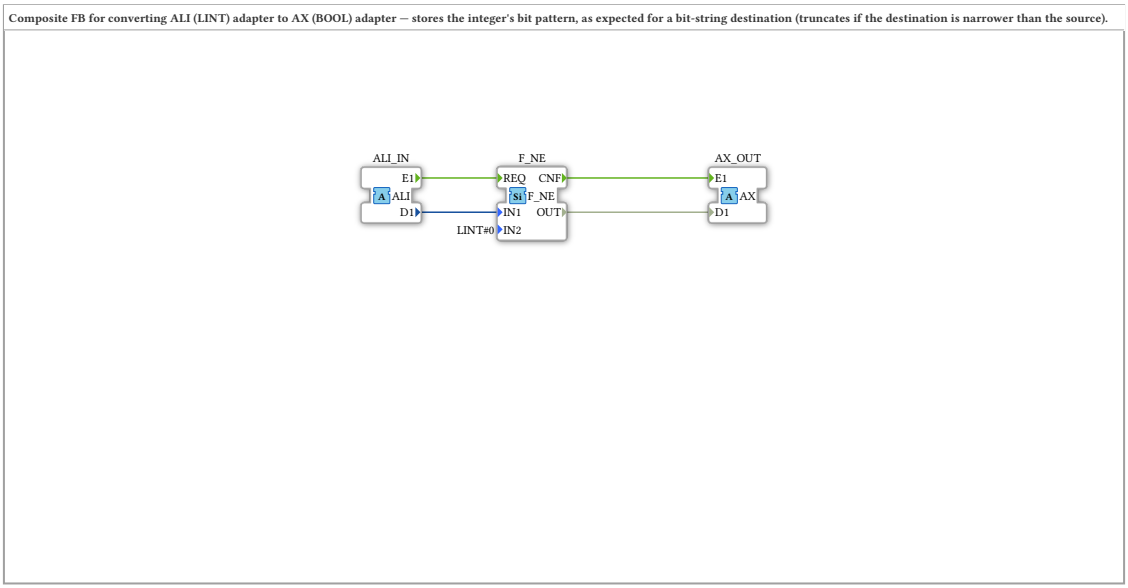
ALI_AX

ALI_TO_AX



LINT Adapter Input – *ALI*

AX – BOOL Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ALI_TO_AX** ist ein zusammengesetzter (Composite) Baustein, der einen Adapter vom Typ **ALI** (enthält einen LINT-Wert) in einen Adapter vom Typ **AX** (enthält einen BOOL-Wert) umwandelt. Die Konvertierung erfolgt nach der Regel: Ein LINT-Wert ungleich 0 wird als BOOL-Wert **TRUE** interpretiert, ein Wert gleich 0 als **FALSE**. Der Baustein dient als bidirektionale Brücke zwischen Adaptern, die auf unterschiedlichen Datentypen basieren.

Schnittstellenstruktur

Ereignis-Eingänge

- **ALI_IN.E1** – Eingangsereignis vom ALI-Adapter. Lösen die Verarbeitung aus.

Ereignis-Ausgänge

- **AX_OUT.E1** – Ausgangsereignis zum AX-Adapter. Wird nach erfolgreicher Verarbeitung gesendet.

Daten-Eingänge

(Keine direkten Daten-Eingänge. Die Daten werden über den ALI-Adapter bereitgestellt:)

- **ALI_IN.D1** – LINT-Eingangswert über den Adapter.

Daten-Ausgänge

(Keine direkten Daten-Ausgänge. Die Daten werden über den AX-Adapter ausgegeben:)

- **AX_OUT.D1** – BOOL-Ausgangswert über den Adapter.

Adapter

Adapter	Richtung	Typ	Beschreibung
ALI_IN	Socket (Eingang)	`adapter::types::unidirectional::ALI`	Eingangsadapter mit einem LINT-Wert.
AX_OUT	Plug (Ausgang)	`adapter::types::unidirectional::AX`	Ausgangsadapter mit einem BOOL-Wert.

Funktionsweise

Der Baustein verwendet intern den Vergleichsbaustein **F_NE** („Not Equal“) aus der IEC 61131-Bibliothek. Der Ablauf ist:

1. Über den Socket `ALI_IN` wird ein LINT-Wert empfangen.
2. Das Ereignis `ALI_IN.E1` triggert den Baustein `F_NE` über dessen `REQ`-Eingang.
3. Im `F_NE` wird der Eingangswert (`IN1`) mit dem konstanten Wert `LINT#0` (`IN2`) verglichen.
4. Das Ergebnis `OUT` ist ein BOOL:
   - **TRUE**, wenn `IN1 ≠ 0`
   - **FALSE**, wenn `IN1 = 0`
5. Nach Abschluss des Vergleichs sendet `F_NE` ein `CNF`-Ereignis, das den Ausgangsadapter `AX_OUT` über dessen `E1` triggert.
6. Gleichzeitig wird der BOOL-Wert auf `AX_OUT.D1` ausgegeben.

Somit wird ein beliebiger LINT-Wert (auch negative Zahlen) in ein binäres Signal umgesetzt.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Reine Logik-Konvertierung**: Es findet keine Skalierung oder Schwellwertbildung statt. Jeder von Null verschiedene Wert wird als `TRUE` interpretiert.
- **Nutzung von Composite-FB**: Der Baustein kapselt die Adapter-Konvertierung in einer wiederverwendbaren Einheit und erlaubt einfaches Einbinden in größere Netzwerke.
- **Ereignisgesteuert**: Die Verarbeitung erfolgt nur bei einem eingehenden Ereignis (`ALI_IN.E1`). Der Ausgang wird nur nach einer erfolgreichen Verarbeitung aktualisiert.
- **Lizenz**: Der Baustein steht unter der **Eclipse Public License 2.0** (EPL-2.0).

Zustandsübersicht

Da es sich um einen Composite-Baustein ohne eigene Zustandsmaschine handelt, ergibt sich die Zustandslogik aus der internen Verkettung:

1. **Warten auf Ereignis**: Der Baustein ist inaktiv, bis `ALI_IN.E1` eintrifft.
2. **Vergleich läuft**: Nach Empfang des Ereignisses wird der `F_NE`-Baustein ausgeführt.
3. **Ergebnis ausgeben**: Nach Abschluss des Vergleichs wird das Ergebnis auf `AX_OUT.D1` gesetzt und `AX_OUT.E1` ausgelöst.

Es gibt keine Schleifen, zeitlichen Verzögerungen oder Fehlerzustände – die Funktion ist deterministisch und fehlerfrei.

Anwendungsszenarien

- **Signalkonvertierung in Steuerungssystemen**: Wenn ein LINT-basiertes Protokoll (z. B. Zählerstände, Encoder) in eine einfache binäre Information (z. B. „Aktiv“, „Präsenz“) umgewandelt werden muss.
- **Adapter-Brücken**: Einsatz in Systemen, die heterogene Adaptertypen verwenden, um die Kommunikation zwischen Komponenten mit unterschiedlichen Datentypen zu ermöglichen.
- **Testumgebungen**: Schnelle Umwandlung von numerischen Werten in boolesche Signale für Debugging oder Simulation.

Vergleich mit ähnlichen Bausteinen

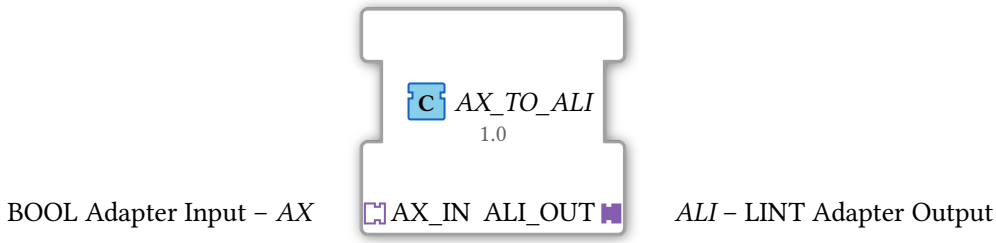
- **Direkter Compare-Baustein**: Statt `F_NE` könnte auch ein `F_EQ` (equal) mit anschließender Negation verwendet werden – funktional identisch.

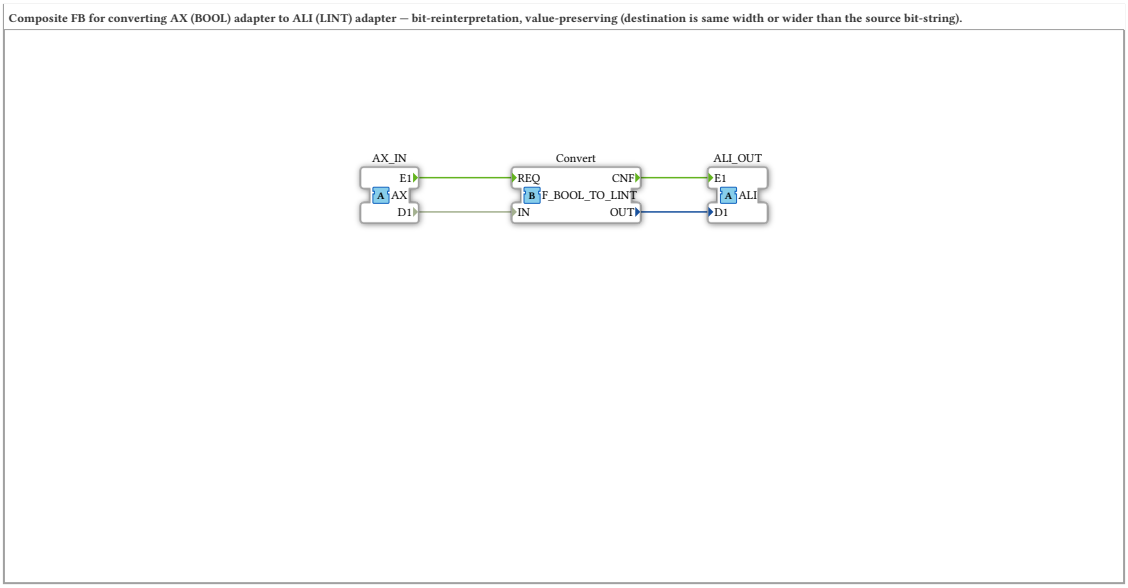
- **Schwellwert-Bausteine (z. B. Greater_than)**: Würden einen expliziten Schwellwert erfordern; `ALI_TO_AX` entscheidet hart nach Null.
- **Typkonvertierungen (z. B. LINT_TO_BOOL)**: Einige Bibliotheken bieten primitive Konverter – diese arbeiten jedoch meist nicht auf Adapter-Ebene. `ALI_TO_AX` ist speziell für die Adapter-Schnittstellen zugeschnitten.

Fazit

Der Baustein **ALI_TO_AX** bietet eine einfache und zuverlässige Methode, um einen LINT-Adapterwert in einen BOOL-Adapterwert zu übersetzen. Dank des klaren Logikaufbaus und der ereignisgesteuerten Architektur ist er sowohl für den Produktiveinsatz als auch für schnelle Prototypen geeignet. Die Verwendung des Composite-Konzepts fördert die Modularität und Wiederverwendbarkeit in IEC 61499-basierten Steuerungsanwendungen.

AX_TO_ALI





• – ○ • – ○ • – ○ •

Einleitung

Der **AX_TO_ALI** ist ein zusammengesetzter Funktionsblock (Composite FB), der einen unidirektionalen AX-Adapter (BOOL) in einen unidirektionalen ALI-Adapter (LINT) umwandelt. Er wurde entwickelt, um BOOL-Werte, die über einen AX-Socket empfangen werden, in LINT-Werte zu konvertieren und über einen ALI-Plug auszugeben. Der Baustein nutzt intern die IEC-61131-Konvertierungsfunktion `F_BOOL_TO_LINT`.

Schnittstellenstruktur

Der AX_TO_ALI besitzt keine direkten Ereignis- oder Dateneingänge/-ausgänge. Die gesamte Kommunikation erfolgt ausschließlich über Adapter.

Ereignis-Eingänge

Keine separaten Ereignis-Eingänge. Ereignisse werden indirekt über den AX-Adapter-Socket empfangen.

Ereignis-Ausgänge

Keine separaten Ereignis-Ausgänge. Ereignisse werden indirekt über den ALI-Adapter-Plug ausgegeben.

Daten-Eingänge

Keine separaten Daten-Eingänge. Daten werden indirekt über den AX-Adapter-Socket empfangen.

Daten-Ausgänge

Keine separaten Daten-Ausgänge. Daten werden indirekt über den ALI-Adapter-Plug ausgegeben.

Adapter

Adapter	Richtung	Typ	Beschreibung
AX_IN	Socket	adapter::types::unidirectional::AX	BOOL-Adapter-Eingang: Empfängt ein BOOL-Signal über den unidirektionalen AX-Adapter.
ALI_OUT	Plug	adapter::types::unidirectional::ALI	LINT-Adapter-Ausgang: Gibt den konvertierten LINT-Wert über den unidirektionalen ALI-Adapter aus.

Funktionsweise

Der Baustein arbeitet ereignisgesteuert und führt bei jedem empfangenen Ereignis über den AX-Adapter-Socket folgende Schritte aus:

1. **Ereignisempfang**: Ein Ereignis E1 des AX-Adapters wird an den internen Konvertierungsbaustein F_BOOL_TO_LINT übermittelt (Ereignis REQ).
2. **Datenübernahme**: Der aktuelle BOOL-Wert D1 des AX-Adapters wird an den Eingang IN der Konvertierungsfunktion weitergegeben.
3. **Konvertierung**: Der interne Baustein wandelt den BOOL-Wert in einen LINT-Wert um (FALSE ⟶ 0, TRUE ⟶ 1).
4. **Datenausgabe**: Das konvertierte LINT-Ergebnis wird über den Datenausgang OUT an den Daten-Eingang D1 des ALI-Adapters gelegt.
5. **Ereignisausgabe**: Nach erfolgreicher Konvertierung wird ein Bestätigungsereignis CNF an den ALI-Adapter-Ausgang E1 gesendet.

Damit wird eine synchrone, ereignisgesteuerte Umsetzung von BOOL in LINT realisiert.

Technische Besonderheiten

- **Bit-Reinterpretation, werterhaltend** (Zielbreite ist gleich groß oder größer als der Quell-Bit-String).
- **Lizenz**: Der Baustein ist unter der **Eclipse Public License 2.0 (EPL-2.0)** veröffentlicht. Copyright © 2026 HR Agrartechnik GmbH.
- **Autor & Version**: Entwickelt von Franz Höpfinger, Version 1.0, Stand 2026-02-17.
- **Adapter-Paket**: Verwendet Adapter aus dem Paket adapter::types::unidirectional, die für eine einseitige Daten- und Ereignisübertragung ausgelegt sind.
- **Interne Realisierung**: Die Konvertierung erfolgt über den IEC-61131-Baustein F_BOOL_TO_LINT aus der Bibliothek iec61131::conversion. Dieser Baustein arbeitet deterministisch und benötigt keine Zwischenspeicher.
- **Hash-Attribut**: Der Baustein enthält ein leeres Typ-Hash-Attribut (eclipse4diac::core::TypeHash).

Zustandsübersicht

Der AX_TO_ALI besitzt keinen eigenen Zustandsautomaten. Er leitet Ereignisse und Daten direkt an den internen Konvertierungsbaustein weiter und gibt das Ergebnis ohne Verzögerung aus. Da der interne Baustein F_BOOL_TO_LINT eine reine Funktionslogik ohne Zustandshaltung ist, verhält sich der gesamte Composite FB stateless. Nach jedem Ereignis am AX-Adapter wird sofort die entsprechende Ausgabe am ALI-Adapter erzeugt.

Anwendungsszenarien

- **Typkonvertierung in Adapterketten**: Wenn in einer IEC-61499-Applikation ein BOOL-Wert über einen unidirektionalen AX-Adapter ankommt, aber nachfolgende Bausteine einen LINT-Wert erwarten (z. B. für Zählfunktionen oder Analogwertverarbeitung).
- **Schnittstellenanpassung**: Anbindung binärer Sensoren (BOOL) an Steuerungslogiken, die mit ganzzahligen LINT-Werten arbeiten.
- **Ereignisgesteuerte Datenumsetzung**: Einfache und schnelle Umwandlung ohne Zwischenspeicher, ideal für zeitkritische Pfade.

Vergleich mit ähnlichen Bausteinen

In der Bibliothek `iec61131::conversion` existieren zahlreiche Funktionsbausteine zur Typkonvertierung, z. B. `F_BOOL_TO_DINT`, `F_BOOL_TO_INT` oder `F_BOOL_TO_USINT`. Der AX_TO_ALI unterscheidet sich dadurch, dass er bereits vollständig in eine Adapterhülle eingebettet ist und die Konvertierung zwischen den unidirektionalen Adaptern AX und ALI ermöglicht. Er erspart dem Entwickler die manuelle Verschaltung von Socket/Plug mit einem separaten Konvertierungsbaustein und sorgt für eine saubere, wiederverwendbare Kapselung.

Fazit

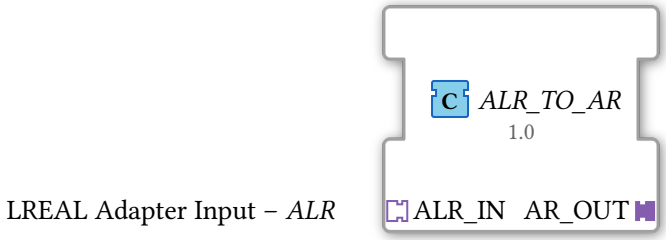
Der AX_TO_ALI ist ein kompakter, lizenzkonformer Composite-FB zur zuverlässigen Umwandlung eines BOOL- in einen LINT-Wert über unidirektionale Adapter. Durch seine einfache Ereignissteuerung und die Verwendung eines etablierten IEC-Bausteins bietet er eine robuste Lösung für typische Konvertierungsaufgaben in IEC-61499-Systemen. Die saubere Trennung von Ein- und Ausgang über Adapter erleichtert die Integration in modulare Steuerungsarchitekturen.

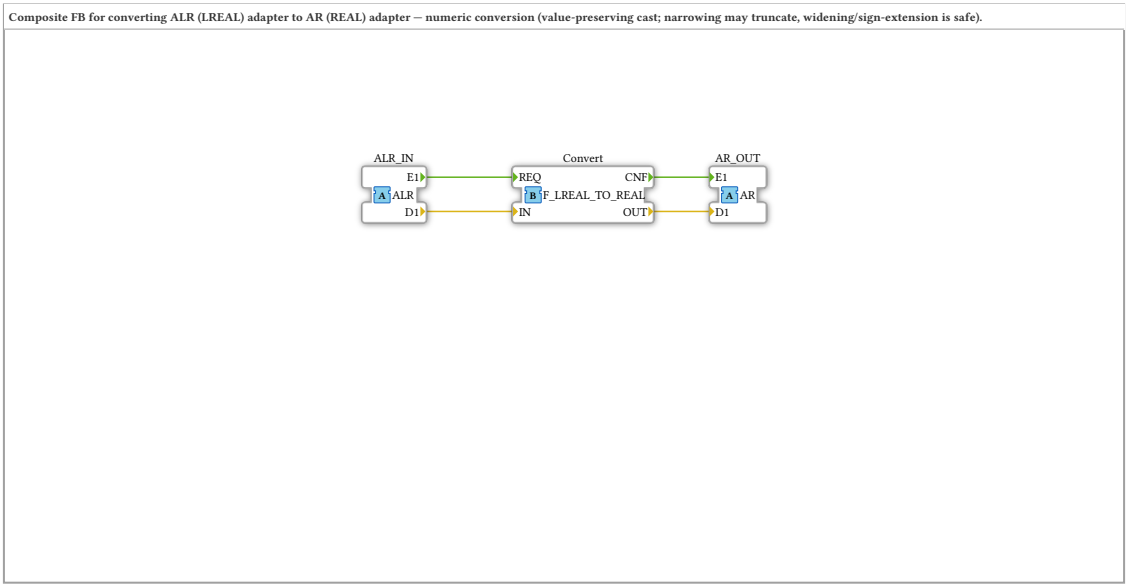
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ALR_AR

ALR_TO_AR





Kein Bild verfügbar

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ALR_TO_AR** ist ein Composite-Baustein, der einen LREAL-Adapter (ALR) in einen REAL-Adapter (AR) umwandelt. Er dient als Bindeglied zwischen Steuerungskomponenten, die mit unterschiedlichen Gleitkomma-Genauigkeiten arbeiten, und ermöglicht eine saubere, ereignisgesteuerte Datenkonvertierung innerhalb eines IEC 61499-Netzwerks.

Schnittstellenstruktur

Ereignis-Eingänge

- **E1** (über Socket `ALR_IN`): Startet die Konvertierung des anliegenden LREAL-Wertes.

Ereignis-Ausgänge

- **E1** (über Plug `AR_OUT`): Signalisiert, dass der konvertierte REAL-Wert am Ausgang gültig ist.

Daten-Eingänge

- **D1** (über Socket `ALR_IN`): LREAL-Eingangswert (64-Bit Gleitkomma).

Daten-Ausgänge

- **D1** (über Plug `AR_OUT`): REAL-Ausgangswert (32-Bit Gleitkomma).

Adapter

- **ALR_IN** (Socket): Adapter vom Typ `adapter::types::unidirectional::ALR` – nimmt LREAL-Daten entgegen.
- **AR_OUT** (Plug): Adapter vom Typ `adapter::types::unidirectional::AR` – stellt REAL-Daten bereit.

Funktionsweise

1. Ein Ereignis am Eingang **E1** (des ALR_IN-Sockets) triggert den internen Baustein `F_LREAL_TO_REAL`.
2. Gleichzeitig wird der LREAL-Wert von **D1** (ALR_IN) direkt an den Konvertierungsbaustein übergeben.
3. Der Konvertierungsbaustein wandelt den 64-Bit-Wert in einen 32-Bit-REAL-Wert um.
4. Nach Abschluss der Konvertierung wird ein Ereignis am Ausgang **E1** (AR_OUT) erzeugt und der REAL-Wert an **D1** (AR_OUT) weitergegeben.

Die Übertragung erfolgt streng sequenziell: erst Ereignis, dann Datenfluss – die interne Verbindung gewährleistet eine zuverlässige Synchronisation.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-Baustein**: Der ALR_TO_AR kapselt die Konvertierungslogik und vereinfacht die Wiederverwendung in verschiedenen Projekten.
- **IEC 61131-Funktion**: Die eigentliche Typumwandlung basiert auf der standardisierten Bibliotheksfunktion `F_LREAL_TO_REAL`.
- **Adapter-Typen**: Beide Adapter sind unidirektional ausgelegt und erlauben eine klare Trennung von Ein- und Ausgabeseite.
- **Keine Zustandsmaschine**: Der Baustein arbeitet rein ereignisgesteuert ohne internen Zustandsspeicher.

Zustandsübersicht

Da es sich um einen Composite-Baustein ohne eigene Zustandsautomaten handelt, gibt es keine expliziten Zustände. Der interne Ablauf ist deterministisch:

- **Bereit / Idle** – wartet auf ein Eingangsereignis.
- **Konvertierung aktiv** – nach Ereigniseingang bis zum Erzeugen des Ausgangsereignisses (dauert i. d. R. einen Zyklus).

Die Latenz ist minimal und durch die Ausführungsreihenfolge des zugrunde liegenden IEC-61499-Laufzeitsystems bestimmt.

Anwendungsszenarien

- **Sensor-Integration**: Ein Feldgerät liefert LREAL-Messwerte, nachgeschaltete Logik erwartet jedoch REAL-Werte (z. B. SPS-Kommunikation).
- **Adapter-Kopplung**: Verbindung zweier Module, die unterschiedliche Adapter-Schnittstellen verwenden, ohne dass der Entwickler die Konvertierung manuell programmieren muss.
- **Datenkonsistenz**: Erzwingt eine saubere Trennung von 64-Bit- und 32-Bit-Datenpfaden und vermeidet implizite Typumwandlungen.

Vergleich mit ähnlichen Bausteinen

- **F_LREAL_TO_REAL** (direkt): Bietet die gleiche Konvertierung, erfordert jedoch separate Ereignis- und Datenverbindungen. ALR_TO_AR kapselt diese Verbindungen und stellt eine einheitliche Adapter-Schnittstelle bereit.
- **LINT_TO_INT** (Adapter-Variante): Analoge Logik für ganzzahlige Typen mit unterschiedlichen Bitbreiten. Der prinzipielle Aufbau ist identisch, jedoch auf andere Datentypen zugeschnitten.
- **Eigene Composite-Konverter**: ALR_TO_AR reduziert den Aufwand für wiederkehrende Adapter-Konvertierungen und erhöht die Lesbarkeit des Netzwerks.

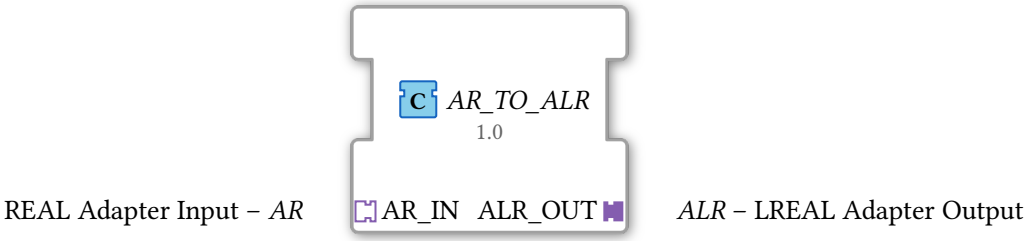
Fazit

Der **ALR_TO_AR** ist ein einfacher, aber effektiver Composite-Baustein, der die Lücke zwischen LREAL- und REAL-Adapter-Schnittstellen schließt. Er bietet eine saubere, ereignisgesteuerte Konvertierung ohne zusätzlichen Ballast und lässt sich leicht in existierende 4diac-Projekte einbinden. Dank der Verwendung von Standard-Bibliotheksfunktionen bleibt die Umwandlung performant und zuverlässig.

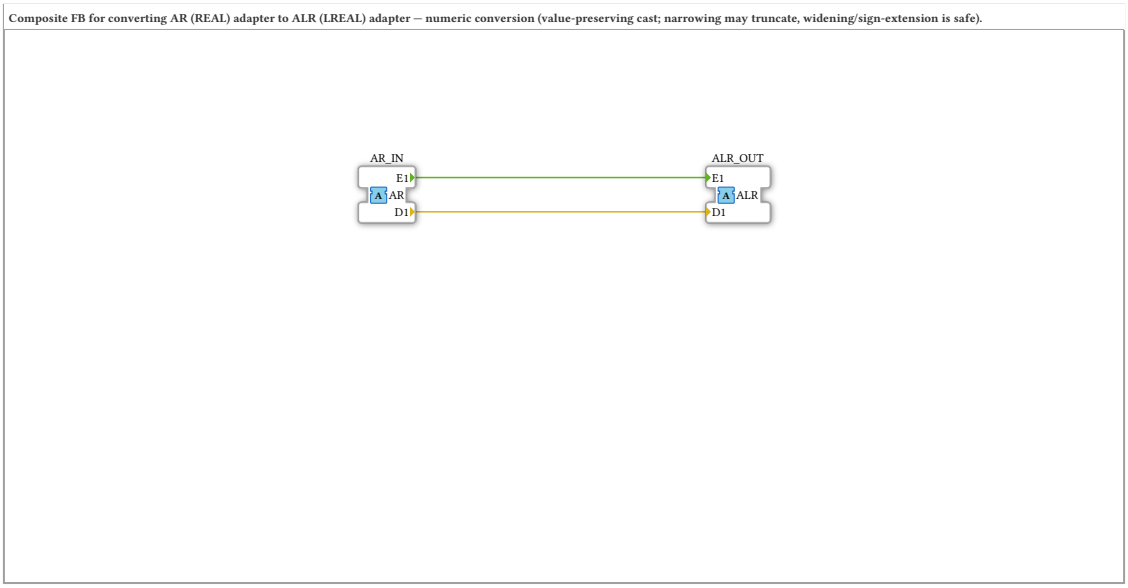
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AR_TO_ALR



REAL Adapter Input – *AR* — *ALR* – LREAL Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Composite-Funktionsblock zur Konvertierung eines AR-Adapters (REAL) in einen ALR-Adapter (LREAL). Er dient der Anpassung von Datenformaten in IEC-61499-2-Umgebungen und ist unter der Eclipse Public License 2.0 lizenziert.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Beschreibung
E1 (von AR_IN)	Ereignis zur Auslösung der Datenübergabe

Ereignis-Ausgänge

Name	Beschreibung
E1 (von ALR_OUT)	Ereignis, das nach erfolgreicher Konvertierung ausgegeben wird

Daten-Eingänge

Name	Datentyp	Beschreibung
D1 (von AR_IN)	REAL	32-Bit Gleitkommazahl

Daten-Ausgänge

Name	Datentyp	Beschreibung
D1 (von ALR_OUT)	LREAL	64-Bit Gleitkommazahl

Adapter

- **ALR_OUT** (Plug): Typ `adapter::types::unidirectional::ALR` – LREAL-Ausgangsadapter
- **AR_IN** (Socket): Typ `adapter::types::unidirectional::AR` – REAL-Eingangsadapter

Funktionsweise

Der Baustein verbindet die Adapterschnittstellen direkt. Ein eingehendes Ereignis an `AR_IN.E1` wird sofort an `ALR_OUT.E1` weitergeleitet. Gleichzeitig wird der Datenwert `AR_IN.D1` (REAL) nach `ALR_OUT.D1` (LREAL) kopiert, wobei die Laufzeitumgebung eine implizite Typkonvertierung durchführt. Es existiert kein Zustandsautomat; die Funktion ist rein kombinatorisch.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- Composite-FB ohne eigene Logik (kein ECC)
- Verwendung unidirektionaler Adapter gemäß IEC 61499-2
- Automatische Typumwandlung durch die 4diac-Laufzeit (z. B. 4diac-ide)
- Paket: `adapter::conversion::unidirectional`

Zustandsübersicht

Keine Zustände vorhanden – der FB besitzt keinen Ausführungs-Zustandsautomaten.

Anwendungsszenarien

- Anschluss eines REAL-liefernden Sensors an eine Komponente, die LREAL erwartet
- Migration bestehender REAL-Adapter auf LREAL mit höherer Genauigkeit
- Schnittstellenanpassung in heterogenen Automatisierungssystemen

Vergleich mit ähnlichen Bausteinen

Ähnliche Bausteine existieren für andere Typkonvertierungen (z. B. `LREAL_TO_REAL`, `INT_TO_REAL`). Der vorliegende Block ist speziell auf die unidirektionalen Adapter `AR` und `ALR` zugeschnitten und erfordert keine eigene Implementierungslogik.

Fazit

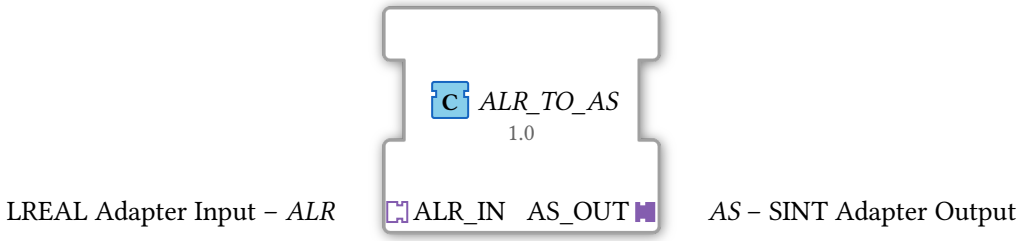
Der `AR_TO_ALR`-Baustein ermöglicht eine einfache und effiziente Umwandlung von REAL- zu LREAL-Adapterdaten. Als Composite-FB realisiert, nutzt er die direkte Verbindung der Adapterschnittstellen und bietet so hohe Performance bei minimaler Latenz.

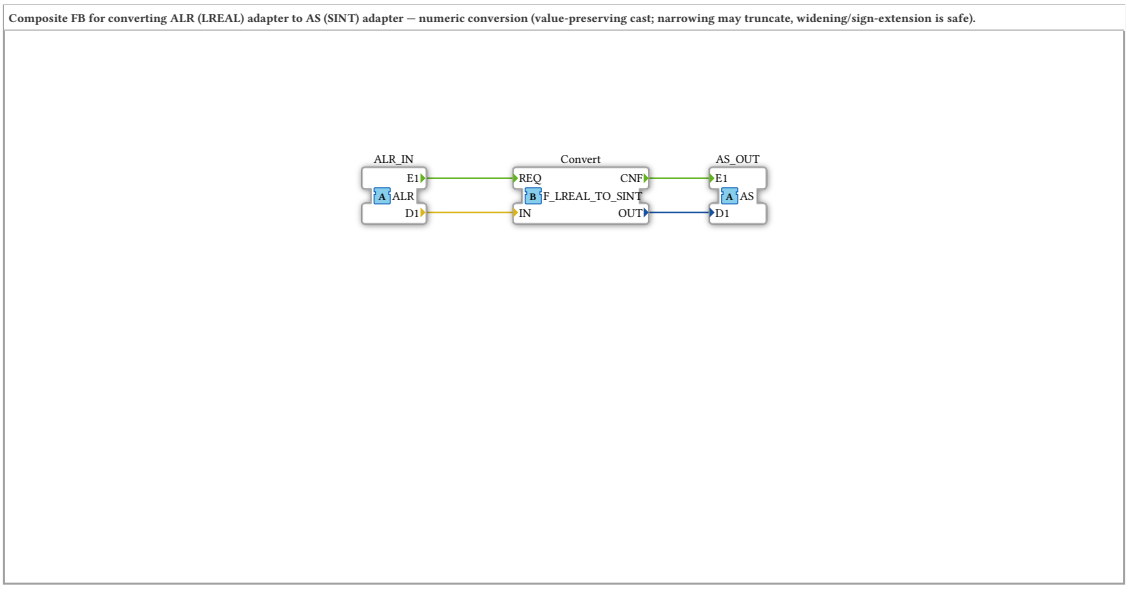
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ALR_AS

ALR_TO_AS





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ALR_TO_AS** ist ein zusammengesetzter (Composite) Baustein, der einen **LREAL-Adapter** (ALR) in einen **SINT-Adapter** (AS) umwandelt. Er kapselt die Konvertierung eines Gleitkommawertes (LREAL) in einen vorzeichenbehafteten 8-Bit-Ganzzahlwert (SINT) und stellt die Ergebnisdaten über den SINT-Adapter bereit. Der Baustein wird typischerweise in der Automatisierungstechnik eingesetzt, wenn Daten zwischen unterschiedlichen Adapter-Schnittstellen vermittelt werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine separaten Ereignis-Eingänge. Die Steuerung der Konvertierung erfolgt über das Ereignis des angeschlossenen **ALR-Adapter-Sockets** (`ALR_IN.E1`).

Ereignis-Ausgänge

Der Baustein besitzt keine separaten Ereignis-Ausgänge. Der Abschluss der Konvertierung wird über das Ereignis des **AS-Adapter-Plugs** (`AS_OUT.E1`) signalisiert.

Daten-Eingänge

Der Baustein besitzt keine separaten Daten-Eingänge. Die Eingangsdaten (LREAL-Wert) werden über den **ALR-Adapter** (`ALR_IN.D1`) bereitgestellt.

Daten-Ausgänge

Der Baustein besitzt keine separaten Daten-Ausgänge. Die Ausgangsdaten (SINT-Wert) werden über den **AS-Adapter** (`AS_OUT.D1`) zur Verfügung gestellt.

Adapter

Adapter	Richtung	Typ	Beschreibung
`ALR_IN`	Socket	`adapter::types::unidirectional::ALR`	LREAL-Adapter-Eingang: liefert den zu konvertierenden LREAL-Wert sowie ein Auslöseereignis (`E1`).
`AS_OUT`	Plug	`adapter::types::unidirectional::AS`	SINT-Adapter-Ausgang: stellt den konvertierten SINT-Wert sowie ein Bestätigungsereignis (`E1`) bereit.

Funktionsweise

Der Baustein ist als Composite FB realisiert. Er enthält intern einen Funktionsblock `F_LREAL_TO_SINT` aus der IEC 61131-Bibliothek. Die Verbindungen innerhalb des Netzwerks bewirken:

1. **Ereignisgesteuerte Aktivierung**: Ein eingehendes Ereignis am Adapter `ALR_IN` (`E1`) triggert den Konvertierungs-FB (`Convert.REQ`).
2. **Datenkonvertierung**: Der LREAL-Wert von `ALR_IN.D1` wird an den Eingang `Convert.IN` weitergeleitet. Der FB wandelt diesen Wert in einen SINT-Wert um (Rundung gemäß IEC 61131-Regeln).
3. **Ergebnisausgabe**: Nach erfolgreicher Konvertierung signalisiert `Convert.CNF` ein Ereignis an den Ausgangsadapter `AS_OUT` (`E1`), und der konvertierte SINT-Wert wird an `AS_OUT.D1` übergeben.

Damit wird eine synchrone, ereignisgesteuerte Umwandlung eines LREAL-Werts in einen SINT-Wert über Adapter-Schnittstellen ermöglicht.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-Struktur**: Der Baustein kapselt die Konvertierungslogik vollständig in einem Netzwerk, sodass die Implementierung transparent und leicht erweiterbar ist.
- **Adapter-basiert**: Statt einzelner Ereignis- und Datenports werden standardisierte uni-direktionale Adapter (`ALR` und `AS`) verwendet. Dies vereinfacht die Integration in bestehende Adapter-basierten Architekturen.
- **Wiederverwendung**: Der interne FB `F_LREAL_TO_SINT` ist ein bewährter IEC-Baustein und stellt die korrekte Konvertierung nach Norm sicher.
- **Keine eigene Zustandsmaschine**: Da es sich um einen Composite FB handelt, existiert kein eigenständiger Zustandsautomat; die Steuerung erfolgt allein durch die verbundenen Adapter-Ereignisse.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten. Sein Verhalten wird durch den internen FB `F_LREAL_TO_SINT` und die Ereignisverbindungen bestimmt. Dieser interne FB durchläuft einen einfachen **IDLE → CONVERTING → DONE**-Zyklus, der über die Ereignisse `REQ` und `CNF` gesteuert wird. Für den Anwender ist der Baustein daher transparent als **ereignisgesteuerter Konverter** nutzbar.

Anwendungsszenarien

- **Datenbrücke zwischen Systemen**: Wenn ein Steuerungsteil LREAL-Werte über einen `ALR`-Adapter liefert, ein nachgeschaltetes Modul aber SINT-Werte über einen `AS`-Adapter erwartet.
- **Einheitliche Adapter-Schnittstellen**: Vereinfachung des Systemdesigns, indem Konvertierungen in eigenständige Bausteine ausgelagert werden.
- **SPS-Anbindung**: Ankopplung von Sensoren oder Aktoren, die mit LREAL-Genauigkeit rechnen, an eine SINT-basierte Buskommunikation.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einer direkten Verdrahtung eines `F_LREAL_TO_SINT`-Bausteins mit einzelnen Ereignis-/Datenports bietet der `ALR_TO_AS`-Baustein eine **höhere Abstraktion durch Adapter**. Dies erleichtert den Austausch von Daten zwischen Komponenten, die bereits auf Adapter-Schnittstellen ausgelegt sind. Ein ähnlicher Baustein könnte `ALR_TO_ADI` (Umwandlung nach INT) oder `ALR_TO_UDI` (Umwandlung nach UDINT) sein, welche analog aufgebaut sind, aber andere Zieldatentypen verwenden.

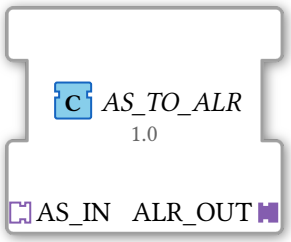
Fazit

Der `ALR_TO_AS`-Funktionsblock ist eine saubere, adapterbasierte Lösung zur Umwandlung von LREAL- in SINT-Werte in der 4diac-IDE. Durch die Composite-Struktur und die Verwendung standardisierter IEC-Konvertierungsbausteine gewährleistet er eine zuverlässige, ereignisgesteuerte Datenkonvertierung und erleichtert die Integration in modulare Automatisierungssysteme.

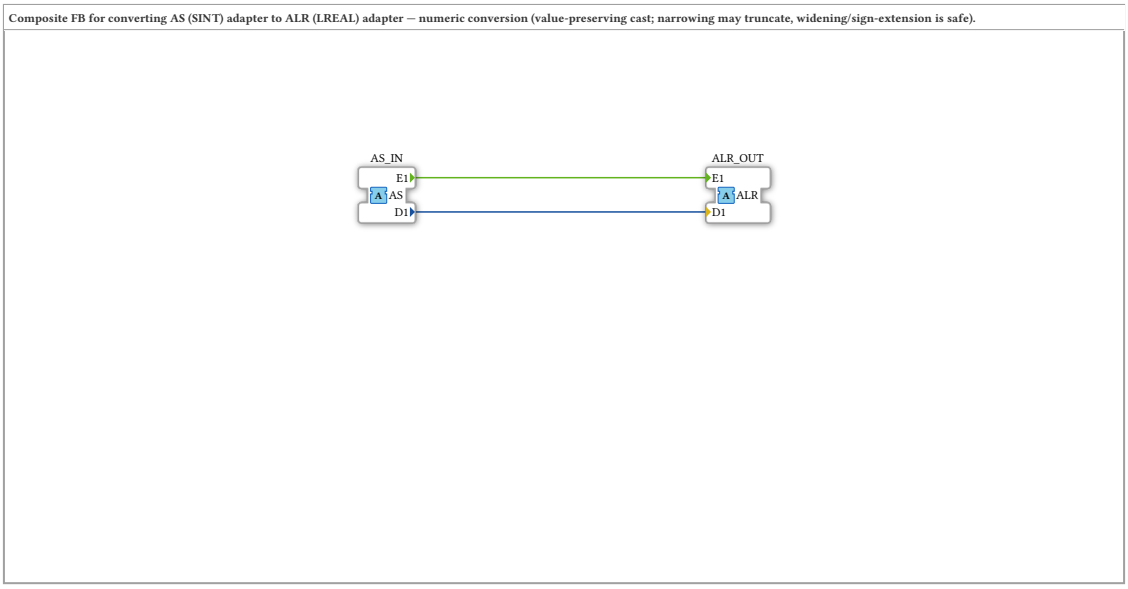
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AS_TO_ALR



SINT Adapter Input – *AS* — *ALR* – LREAL Adapter Output



Kein Bild verfügbar.

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AS_TO_ALR** ist ein zusammengesetzter Baustein (Composite FB), der einen **AS-Adapter** (Datentyp SINT) in einen **ALR-Adapter** (Datentyp LREAL) umwandelt. Er dient als einfache Schnittstellen-Konvertierung zwischen diesen beiden unidirektionalen Adaptertypen und ist im Paket `adapter::conversion::unidirectional` abgelegt.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Kommentar
E1	Ereignis vom AS-Adapter (Socket AS_IN)

Ereignis-Ausgänge

Name	Kommentar
E1	Ereignis zum ALR-Adapter (Plug ALR_OUT)

Daten-Eingänge

Name	Datentyp	Kommentar
D1	SINT	Datenwert vom AS-Adapter (Socket AS_IN)

Daten-Ausgänge

Name	Datentyp	Kommentar
D1	LREAL	Datenwert zum ALR-Adapter (Plug ALR_OUT)

Adapter

Richtung	Name	Typ	Kommentar
Socket	AS_IN	`adapter::types::unidirectional::AS`	Eingangs-Adapter (SINT)

Plug	ALR_OUT	`adapter::types::unidirectional::ALR`	Ausgangs-Adapter (LREAL)

Funktionsweise

Der **AS_TO_ALR**-Baustein leitet die eingehenden Ereignisse und Datenwerte direkt und ohne weitere Verarbeitung an die Ausgangsseite weiter.

- Das Ereignis **E1** am Socket `AS_IN` wird unverzögert an den Plug `ALR_OUT` übergeben.
- Der Datenwert **D1** vom Typ SINT wird an den Ausgang gleichen Namens weitergeleitet. Die Umwandlung von SINT nach LREAL erfolgt implizit durch die Laufzeitumgebung – der Funktionsblock selbst führt keine arithmetische Operation aus.

Der Baustein besitzt keine eigene Zustandsmaschine (ECC) und keine internen Algorithmen; seine gesamte Funktionalität wird durch die enthaltenen Verbindungen realisiert.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Komposit-Struktur**: Der FB besteht ausschließlich aus einem internen Netzwerk (FBNetwork) mit direkten Ereignis- und Datenverbindungen zwischen den Adaptern.
- **Implicite Typkonvertierung**: Die Datenkonvertierung von SINT nach LREAL wird von der IEC-61499-Laufzeitumgebung sichergestellt. Eine explizite Typanpassung ist nicht erforderlich.
- **Paketierung**: Der Baustein ist im Paket `adapter::conversion::unidirectional` organisiert und bietet eine saubere Trennung zwischen Adapter- und Konverterlogik.

Zustandsübersicht

Der **AS_TO_ALR**-Baustein besitzt keine eigene Zustandsmaschine (kein ECC-Diagramm). Sein Verhalten ist rein durch die direkte Verschaltung der Ein- und Ausgänge definiert. Ein definierter Start- oder Endzustand existiert nicht.

Anwendungsszenarien

- **Sensor-Anbindung**: Ein Sensor liefert Daten im SINT-Format (z. B. ein 8-Bit-Zähler), die über einen AS-Adapter in eine Umgebung mit LREAL-Schnittstelle (z. B. Analog-Adapter) eingebunden werden sollen.
- **Protokoll-Konvertierung**: In heterogenen Automatisierungssystemen, in denen unterschiedliche Komponenten Adapter verschiedener Typen erwarten, dient **AS_TO_ALR** als einfacher Umsetzer.
- **Schnittstellen-Migration**: Bei der Umstellung von SINT- auf LREAL-basierte Adapter kann dieser Baustein als temporäre Brücke verwendet werden.

Vergleich mit ähnlichen Bausteinen

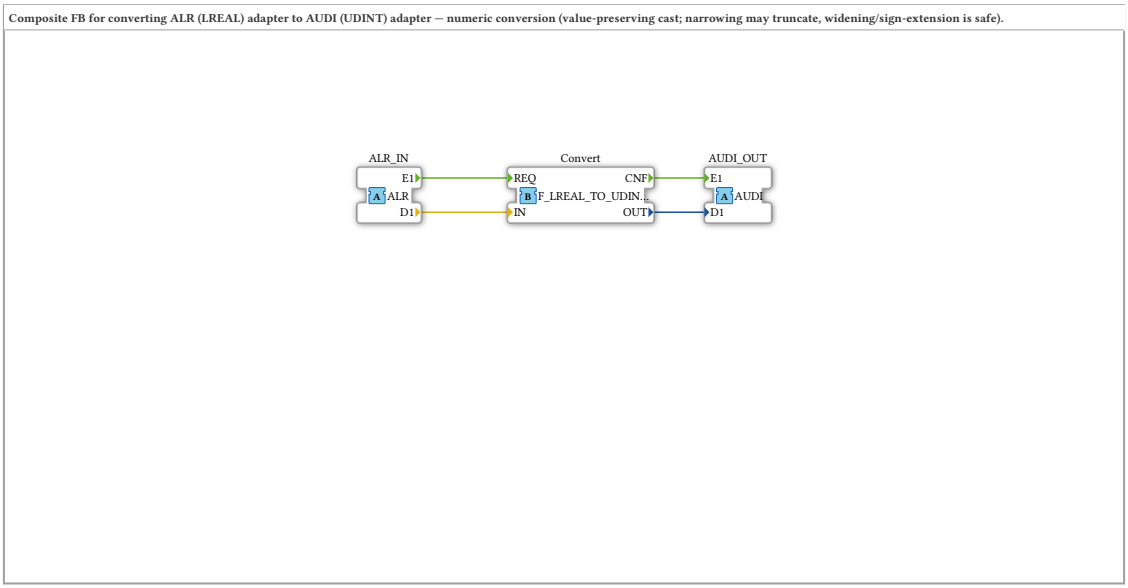
- **AS_TO_AR**: Wandelt einen AS-Adapter (SINT) in einen AR-Adapter (REAL) um – ähnliche Struktur, jedoch mit kleinerem Wertebereich (REAL statt LREAL).
- **ALR_TO_AS**: Die inverse Konvertierung (LREAL → SINT), die ggf. eine explizite Bereichsprüfung oder Rundung erfordert.
- **AS_TO_ALR** zeichnet sich durch den größeren Wertebereich des Ausgangs (LREAL) aus, was eine verlustfreie Übertragung von SINT-Werten ermöglicht.

Fazit

Der **AS_TO_ALR** ist ein minimaler, aber nützlicher Konverterbaustein für die einfache Ankopplung von AS-Adaptern an ALR-Adapter. Seine Composite-Architektur und die implizite Typkonvertierung machen ihn leicht verständlich und effizient in der Anwendung.

ALR_AUDI

ALR_TO_AUDI



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock ALR_TO_AUDI ist ein zusammengesetzter Baustein (Composite FB), der einen **ALR-Adapter** (LREAL) in einen **AUDI-Adapter** (UDINT) umwandelt. Er dient als Schnittstelle zwischen einer analogen LREAL-Quelle und einer digitalen UDINT-Senke in unidirektionalen Kommunikationspfaden.

Schnittstellenstruktur

Ereignis-Eingänge

- `ALR_IN.E1` – Ereigniseingang des ALR-Adapters; löst die Konvertierung aus.

Ereignis-Ausgänge

- `AUDI_OUT.E1` – Ereignisausgang des AUDI-Adapters; signalisiert die Fertigstellung der Konvertierung.

Daten-Eingänge

- `ALR_IN.D1` – LREAL-Eingangswert, der in einen UDINT-Wert umgewandelt wird.

Daten-Ausgänge

- `AUDI_OUT.D1` – UDINT-Ausgangswert, Ergebnis der Konvertierung.

Adapter

- **Socket** `ALR_IN` vom Typ `adapter::types::unidirectional::ALR` – Eingangsschnittstelle für LREAL-Werte mit Ereignis.
- **Plug** `AUDI_OUT` vom Typ `adapter::types::unidirectional::AUDI` – Ausgangsschnittstelle für UDINT-Werte mit Ereignis.

Funktionsweise

Der Baustein nutzt intern den Konvertierungs-FB `iec61131::conversion::F_LREAL_TO_UDINT`. Der Ablauf ist wie folgt:

1. Ein Ereignis auf `ALR_IN.E1` wird an den Eingang `REQ` des internen Konverters weitergeleitet.
2. Gleichzeitig wird der Datenwert von `ALR_IN.D1` an den Eingang `IN` des Konverters übergeben.
3. Nach erfolgreicher Wandlung sendet der Konverter ein Bestätigungsereignis (`CNF`) an `AUDI_OUT.E1`.
4. Der konvertierte UDINT-Wert steht am Ausgang `OUT` des Konverters und wird an `AUDI_OUT.D1` weitergegeben.

Somit erfolgt die Umwandlung streng ereignisgesteuert und datensynchron.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Adapterbasierte Kapselung** – Daten und Ereignisse werden über standardisierte Adapter vom Typ `unidirectional` ausgetauscht, was eine saubere Trennung von Schnittstellen und Implementierung ermöglicht.
- **Unidirektionale Kommunikation** – Der Baustein unterstützt nur eine Datenflussrichtung (LREAL → UDINT). Rückmeldungen oder Fehlerstatus sind nicht vorgesehen.
- **Typumwandlung** – Die Konvertierung von LREAL (64-Bit-Gleitkomma) nach UDINT (32-Bit-Ganzzahl ohne Vorzeichen) kann zu Rundungs- oder Überlaufverlusten führen, wird aber gemäß IEC 61131-Standard durchgeführt.

Zustandsübersicht

Der ALR_TO_AUDI besitzt keine eigene interne Zustandsmaschine. Sein Verhalten wird vollständig durch den enthaltenen Funktionsblock `F_LREAL_TO_UDINT` bestimmt, der als einfache, ereignisgesteuerte Wandlungsinstanz arbeitet.

Anwendungsszenarien

- **Analog-Digital-Wandlung in der Agrarsteuerung** – Umwandlung von LREAL-Sensorwerten (z. B. Druck, Füllstand) in ganzzahlige UDINT-Werte für Anzeigen oder digitale Regelkreise.
- **Schnittstellenanpassung** – Verbindung von Komponenten, die auf unterschiedliche Adapter-Typen setzen (z. B. ALR aus einem Messmodul und AUDI in einer Steuerung).
- **Protokollumsetzung** – Einfache Konvertierung innerhalb von Adapterketten, ohne eigene Logik implementieren zu müssen.

Vergleich mit ähnlichen Bausteinen

- **LREAL_TO_DINT** – Wandelt in einen vorzeichenbehafteten 32-Bit-Integer. ALR_TO_AUDI nutzt dagegen die vorzeichenlose Variante UDINT.
- **REAL_TO_UDINT** – Arbeitet mit 32-Bit-Gleitkomma statt 64-Bit-LREAL, entsprechend geringere Genauigkeit.
- **LREAL_TO_LWORD** – Konvertiert in einen 64-Bit-Integer, jedoch mit anderem Adapter-Typ (LWORD statt UDINT).

Der ALR_TO_AUDI ist speziell auf die unidirektionale Adapter-Kopplung mit den genannten Typen ausgelegt.

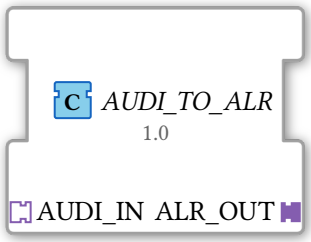
Fazit

Der ALR_TO_AUDI ist ein spezialisierter, aber dennoch einfacher Baustein zur Konvertierung von LREAL- in UDINT-Werte über Adapterschnittstellen. Dank seiner klaren Struktur und der Nutzung eines standardisierten IEC-Konverters lässt er sich leicht in bestehende 4diac-Projekte integrieren und trägt zur Interoperabilität unterschiedlicher Adapter-Typen bei.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

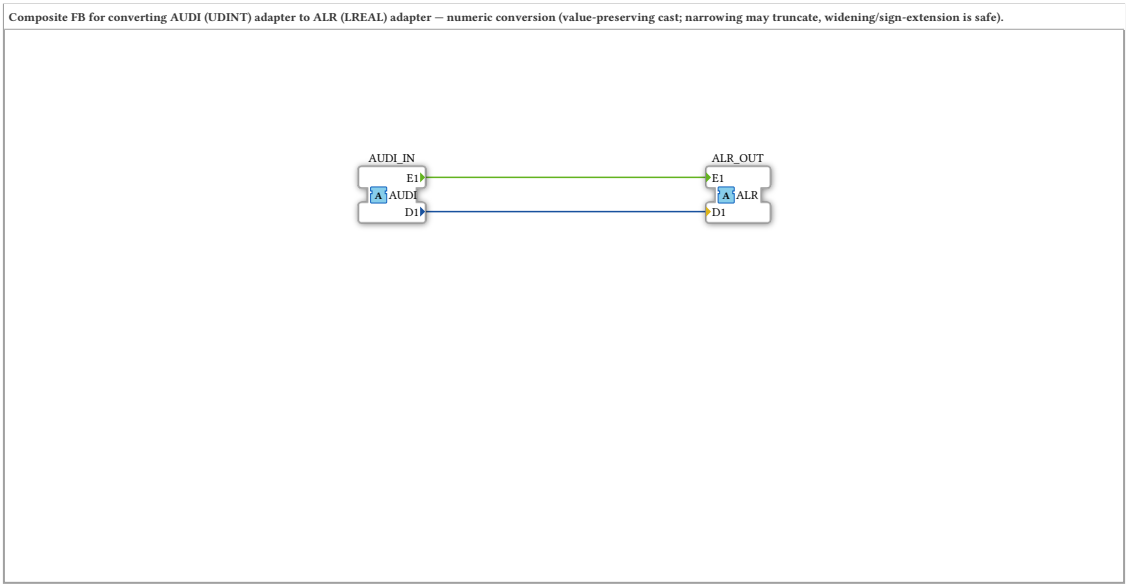
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUDI_TO_ALR



UDINT Adapter Input – *AUDI*

ALR – LREAL Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUDI_TO_ALR** ist ein Composite-Baustein, der einen unidirektionalen Adapter vom Typ **AUDI** (Datentyp `UDINT`) in einen Adapter vom Typ **ALR** (Datentyp `LREAL`) umwandelt. Er dient als einfaches Bindeglied zwischen zwei unterschiedlichen Adapter-Schnittstellen und ermöglicht so die Integration von Komponenten, die auf unterschiedlichen Datenformaten basieren.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Kommentar
`AUDI_IN.E1`	Ereignis-Eingang des AUDI-Adapters (durchgeschliffen zum ALR-Ausgang)

Ereignis-Ausgänge

Name	Kommentar
`ALR_OUT.E1`	Ereignis-Ausgang des ALR-Adapters (übernommen vom AUDI-Eingang)

Daten-Eingänge

Name	Datentyp	Kommentar
`AUDI_IN.D1`	`UDINT`	Daten-Eingang des AUDI-Adapters (wird direkt als LREAL weitergegeben)

Daten-Ausgänge

Name	Datentyp	Kommentar
ALR_OUT.D1	LREAL	Daten-Ausgang des ALR-Adapters (Wert vom AUDI-Eingang, typkonvertiert)

Adapter

Richtung	Name	Typ	Kommentar
Socket (Eingang)	AUDI_IN	Adapter::types::unidirectional::AUDI	Nimmt die UDINT-Daten und das zugehörige Ereignis entgegen
Plug (Ausgang)	ALR_OUT	Adapter::types::unidirectional::ALR	Stellt die konvertierten LREAL-Daten und das Ereignis bereit

Funktionsweise

Der Funktionsblock leitet das eingehende Ereignis `E1` vom Socket `AUDI_IN` direkt an den Plug `ALR_OUT` weiter. Gleichzeitig wird der Datenwert `D1` (Typ `UDINT`) unverändert auf den Ausgangsdatenkanal `D1` (Typ `LREAL`) übertragen. Die tatsächliche Typkonvertierung von `UDINT` nach `LREAL` erfolgt innerhalb der verwendeten Adapter-Schnittstellen; der Composite-Baustein selbst führt keine arithmetische oder logische Umformung durch, sondern fungiert als reine Verdrahtungshilfe.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Reine Adapter-Durchleitung:** Der Baustein enthält keinerlei eigene Logik oder Zustandsautomaten – er verbindet lediglich die gleichnamigen Kanäle der beiden Adapter.
- **Unidirektionale Kommunikation:** Sowohl der AUDI- als auch der ALR-Adapter unterstützen nur eine Datenflussrichtung, was den Einsatz in einfachen Sensor-/Aktor-Schnittstellen oder bei der Kopplung unterschiedlicher Protokolle erleichtert.
- **Keine Typkonvertierung im Baustein:** Die Wandlung von `UDINT` auf `LREAL` wird von der Infrastruktur der Adapter übernommen. Der Entwickler muss sicherstellen, dass die verwendeten Adapter-Implementierungen die nötige Konvertierung bereitstellen.

Zustandsübersicht

Als Composite-Baustein besitzt `AUDI_TO_ALR` keinen eigenen Zustandsautomaten. Sein Verhalten ist rein kombinatorisch: Ein eingehendes Ereignis wird sofort – ohne Verzögerung oder Bedingung – an den Ausgang weitergegeben.

Anwendungsszenarien

- **Integration von UDINT-basierten Sensoren** (z.B. Drehgeber mit 32-Bit-Zählern) in ein LREAL-verarbeitendes Regelungssystem.
- **Adapterbrücke** zwischen verschiedenen Bibliotheken oder Komponenten, die auf unterschiedlichen Datentypen basieren, aber einheitliche Adapter-Schnittstellen verwenden.
- **Prototypenentwicklung** und schnelle Verkabelung von Adaptern, ohne eine separate Konvertierungslogik erstellen zu müssen.

Vergleich mit ähnlichen Bausteinen

Im 4diac-Ökosystem existieren einfache Konvertierungsbausteine wie `INT_TO_REAL` oder `UDINT_TO_LREAL`, die meist als Basic- oder Service-FBs realisiert sind. Der vorliegende Composite-FB unterscheidet sich dadurch, dass er nicht den Datenwert selbst konvertiert, sondern zwei komplette Adapter miteinander verbindet. Er ist daher spezifisch für Umgebungen, die auf Adapter-Schnittstellen angewiesen sind, und nicht für reine Datenfluss-Konvertierung gedacht.

Fazit

`AUDI_TO_ALR` ist ein minimaler, aber nützlicher Composite-Baustein, um zwei unidirektionale Adapter mit unterschiedlichen Datentypen zu koppeln. Er reduziert den Verdrahtungsaufwand und macht die

Typanpassung explizit sichtbar. Der Baustein sollte immer dann eingesetzt werden, wenn eine saubere Trennung zwischen Adapter-Typen und eine einfache Durchleitung von Ereignis und Daten gewünscht ist.

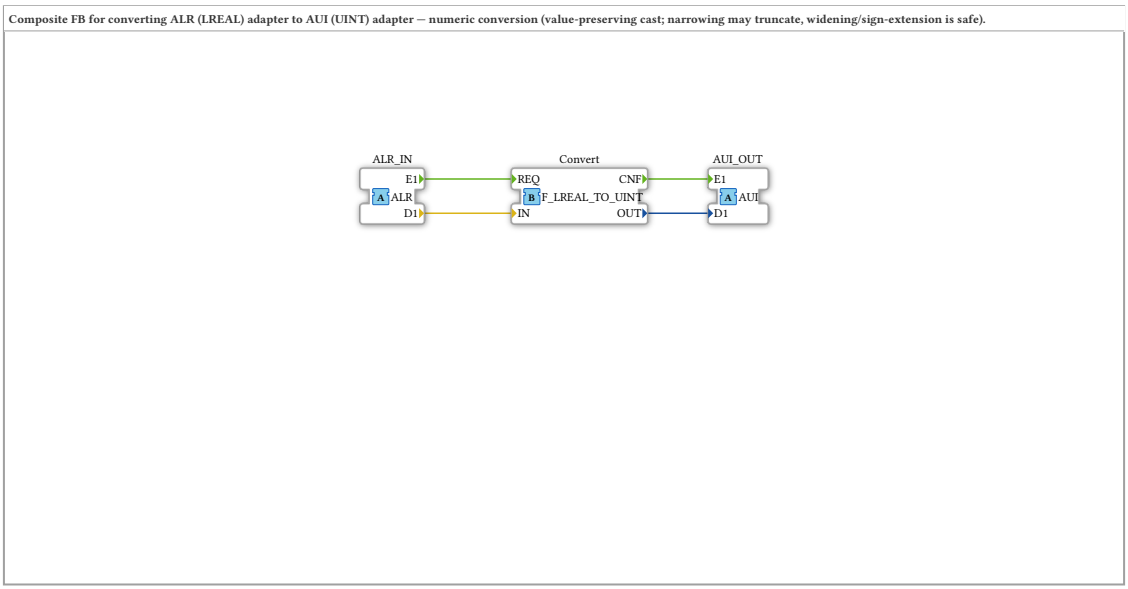
Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ALR_AUI

ALR_TO_AUI

LREAL Adapter Input – *ALR* — *AUI* – UINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ALR_TO_AUI** ist ein zusammengesetzter Baustein (Composite FB), der einen LREAL-Wert, der über einen **ALR**-Adapter (unidirektional) anliegt, in einen UINT-Wert umwandelt und über einen **AUI**-Adapter (unidirektional) bereitstellt. Die Konvertierung erfolgt mittels des internen IEC-61131-Bausteins `F_LREAL_TO_UINT`. Der Baustein kapselt die Adapterwandlung und bietet eine saubere Schnittstelle für den Austausch zwischen den beiden Adaptertypen.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine eigenen Ereignis-Eingänge. Ereignisse werden indirekt über den **ALR_IN**-Adapter empfangen:

- **ALR_IN.E1** – Ereignis zur Auslösung der Konvertierung.

Ereignis-Ausgänge

Der FB besitzt keine eigenen Ereignis-Ausgänge. Ereignisse werden über den **AUI_OUT**-Adapter gesendet:

- **AUI_OUT.E1** – Bestätigungsereignis nach erfolgter Konvertierung.

Daten-Eingänge

Die Daten werden ebenfalls über den **ALR_IN**-Adapter bereitgestellt:

- **ALR_IN.D1** – Eingangswert vom Typ **LREAL** (gemäß ALR-Adapter).

Daten-Ausgänge

Die konvertierten Daten werden über den **AUI_OUT**-Adapter ausgegeben:

- **AUI_OUT.D1** – Ausgangswert vom Typ **UINT** (gemäß AUI-Adapter).

Adapter

Bezeichnung	Typ	Richtung	Beschreibung
ALR_IN	`adapter::types::unidirectional::ALR`	Socket (Eingang)	Liefert den LREAL-Wert und das Trigger-Ereignis.
AUI_OUT	`adapter::types::unidirectional::AUI`	Plug (Ausgang)	Gibt den konvertierten UINT-Wert und das Bestätigungsereignis aus.

Funktionsweise

Der FB arbeitet als reine Übersetzungsschicht zwischen zwei Adaptern:

1. Ein eingehendes Ereignis am Socket **ALR_IN.E1** triggert den internen Konvertierungsbaustein `F_LREAL_TO_UINT` über dessen **REQ**-Eingang.
2. Gleichzeitig wird der Datenwert **ALR_IN.D1** an den **IN**-Eingang des Konvertierungsbausteins weitergeleitet.
3. Der interne Baustein führt die Typumwandlung von **LREAL** nach **UINT** aus und legt das Ergebnis an seinem **OUT**-Ausgang an.
4. Sobald die Konvertierung abgeschlossen ist, sendet der interne Baustein ein Bestätigungsereignis (**CNF**), welches an den Plug **AUI_OUT.E1** weitergegeben wird.
5. Gleichzeitig wird der konvertierte Wert (**OUT**) an den Datenausgang **AUI_OUT.D1** übergeben.

Der gesamte Ablauf ist ereignisgesteuert und erfolgt pro auslösendem Ereignis einmalig.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Adapter-Kapselung:** Der Baustein verbindet zwei unterschiedliche Adaptertypen (ALR und AUI) und ermöglicht so eine typsichere Kommunikation zwischen Komponenten, die diese Adapter verwenden.
- **Unidirektionale Adapter:** Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional ausgelegt, d.h. es gibt nur eine Daten- und Ereignisrichtung (Eingang → Ausgang).
- **Interne Konvertierung:** Die eigentliche Wandlung wird durch den standardisierten IEC-61131-Baustein `F_LREAL_TO_UINT` durchgeführt. Dadurch ist das Konvertierungsverhalten wohldefiniert und plattformunabhängig.
- **Keine Zuständigkeit:** Der Composite FB selbst besitzt keinen eigenen Zustandsautomaten. Das Verhalten wird vollständig durch den eingebetteten Baustein bestimmt.

Zustandsübersicht

Da es sich um einen zusammengesetzten Baustein ohne eigenen ECC (Execution Control Chart) handelt, existiert kein interner Zustandsautomat. Der Ablauf wird durch die Ereignisverknüpfungen gesteuert:

- **IDLE:** Warten auf ein Ereignis am Socket **ALR_IN.E1**.
- **KONVERTIEREN:** Nach Empfang des Ereignisses wird die Konvertierung ausgelöst; der interne Baustein durchläuft seinen eigenen Zustandsautomaten (z.B. REQ → CNF).
- **AUSGABE:** Nach Abschluss der Konvertierung wird das Ausgangsereignis **AUI_OUT.E1** sowie der konvertierte Wert ausgegeben.

Anwendungsszenarien

- **Anbindung von Sensoren/Aktoren:** Ein Sensor liefert einen LREAL-Wert (z.B. Temperatur, Druck), der über einen ALR-Adapter in eine SPS eingelesen wird. Für eine Anzeige oder Weitergabe an ein System, das nur Ganzzahlen (UINT) verarbeitet, wird der FB vorgeschaltet.
- **Adapter-Migration:** In bestehenden Anlagen sollen Adapter von ALR auf AUI umgestellt werden, ohne die angeschlossenen Komponenten zu ändern. Der Baustein fungiert als Zwischenstück.
- **Test- und Simulationsumgebungen:** Erzeugung von UINT-Werten aus LREAL-Quellen für Testzwecke.

Vergleich mit ähnlichen Bausteinen

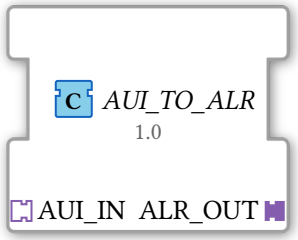
Baustein	Eingangstyp	Ausgangstyp	Beschreibung
ALR_TO_AUI	ALR (LREAL)	AUI (UINT)	Adapter-basierte Konvertierung LREAL→UINT.
`F_LREAL_TO_UINT`	LREAL	UINT	Reine Datenkonvertierung ohne Adapter.
`LREAL_TO_INT`	LREAL	INT	Konvertierung in einen vorzeichenbehafteten Integer.

Im Gegensatz zu den reinen Datenwandlern arbeitet **ALR_TO_AUI** auf Adapterebene, sodass sowohl die Daten als auch die zugehörigen Ereignisse über die Adapterschnittstellen ausgetauscht werden. Dies vereinfacht die Integration in adapterbasierte Architekturen.

Fazit

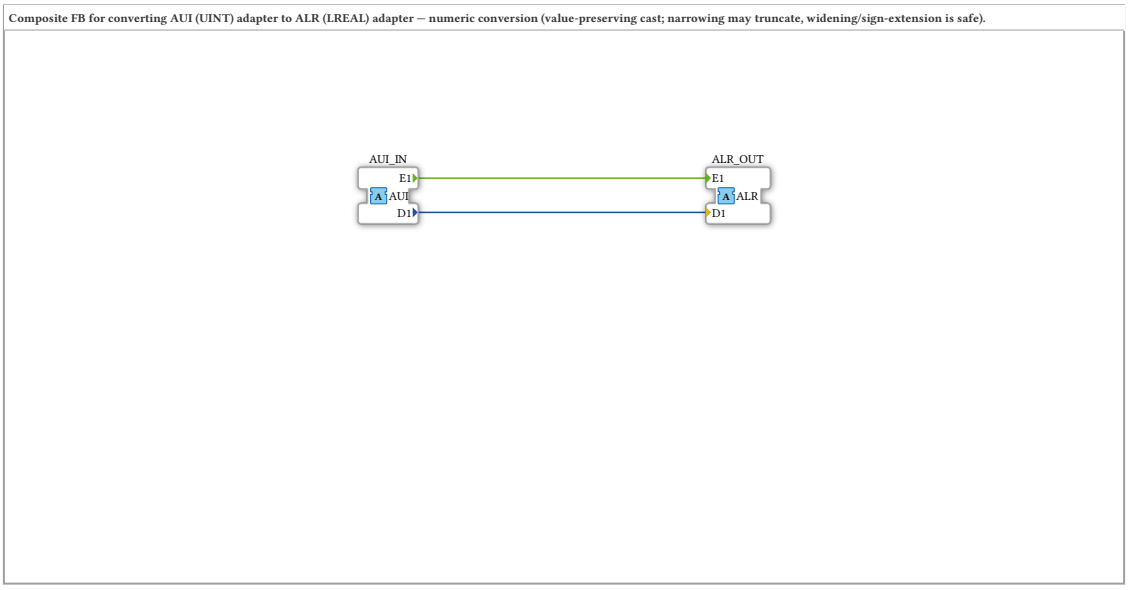
Der **ALR_TO_AUI**-Funktionsblock ist ein nützliches Werkzeug zur Umwandlung von LREAL-Werten aus einem ALR-Adapter in UINT-Werte eines AUI-Adapters. Durch die Kapselung der Konvertierung in einem Composite FB wird die Wiederverwendbarkeit erhöht und die Systemschnittstelle sauber definiert. Der Baustein eignet sich besonders für Anwendungen, in denen Adaptertypen harmonisiert oder angepasst werden müssen, ohne die interne Logik der angeschlossenen Komponenten zu verändern.

AUI_TO_ALR



UINT Adapter Input – *AUI*

ALR – LREAL Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AUI_TO_ALR ist ein zusammengesetzter Baustein (Composite), der eine Schnittstellenkonvertierung zwischen einem AUI-Adapter (Datentyp UINT) und einem ALR-Adapter (Datentyp LREAL) ermöglicht. Er dient als reine Durchleitung und überträgt sowohl Ereignis- als auch Datensignale vom Eingangs- zum Ausgangsadapter.

Schnittstellenstruktur

Ereignis-Eingänge

- **Ereignis über Socket AUI_IN (E1)** – Ereigniseingang vom AUI-Adapter. Wird intern direkt mit dem Ereignisausgang des ALR-Adapters verbunden.

Ereignis-Ausgänge

- **Ereignis über Plug ALR_OUT (E1)** – Ereignisausgang zum ALR-Adapter. Entspricht dem durchgeleiteten Ereignis vom Socket AUI_IN.

Daten-Eingänge

- **Datenwert über Socket AUI_IN (D1)** – Dateneingang vom AUI-Adapter (Typ UINT). Wird intern direkt mit dem Datenausgang des ALR-Adapters verbunden.

Daten-Ausgänge

- **Datenwert über Plug ALR_OUT (D1)** – Datenausgang zum ALR-Adapter (Typ LREAL). Entspricht dem durchgeleiteten Datenwert vom Socket AUI_IN.

Adapter

- **Socket: AUI_IN** – Quelle (Typ `adapter::types::unidirectional::AUI`) Nimmt die Eingangssignale (Ereignis + UINT-Daten) entgegen.
- **Plug: ALR_OUT** – Senke (Typ `adapter::types::unidirectional::ALR`) Stellt die Ausgangssignale (Ereignis + LREAL-Daten) bereit.

Funktionsweise

Der FB AUI_TO_ALR arbeitet als reine Koppelstelle. Ereignis und Daten werden vom Socket AUI_IN direkt auf den Plug ALR_OUT weitergeleitet. Die eigentliche Konvertierung von UINT nach LREAL wird nicht innerhalb dieses Bausteins durchgeführt, sondern erfolgt implizit durch die verwendeten Adapter-Typdefinitionen (unidirektionale Protokolle). Somit fungiert der Composite als standardisierte Schnittstelle, um zwei unterschiedliche Adapterprotokolle miteinander zu verbinden.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-Typ:** Der FB besitzt keinen eigenen Execution-Control-Chart (ECC), sondern besteht ausschließlich aus einer Netzwerkverbindung zwischen seinen beiden Adaptern.
- **Direkte Durchleitung:** Keine interne Logik, Verzögerungen oder Zustandsänderungen – der FB ist rein passiv.
- **Adapter-basiert:** Die verwendeten Adapter sind als unidirektionale Schnittstellen definiert, die je ein Ereignis und einen Datenwert transportieren.

Zustandsübersicht

Der Baustein hat keine eigenen Zustände, da er als Composite ohne ECC realisiert ist. Das Verhalten wird vollständig durch die verbundenen Adapter bestimmt.

Anwendungsszenarien

- **Protokollanpassung in der Agrartechnik:** Einsatz in Steuerungen, die zwischen verschiedenen Sensor- oder Aktorschnittstellen vermitteln müssen.
- **Nachrüstung oder Migration:** Ermöglicht den Einsatz eines ALR-kompatiblen Geräts an einem bestehenden AUI-Anschluss, ohne die eigentliche Konvertierung selbst implementieren zu müssen.

- **Modulare Verkabelung:** Als einfaches Verbindungsglied in einer Kette von Adapter-Bausteinen, z. B. in einer Feldbus- oder Kommunikationshierarchie.

Vergleich mit ähnlichen Bausteinen

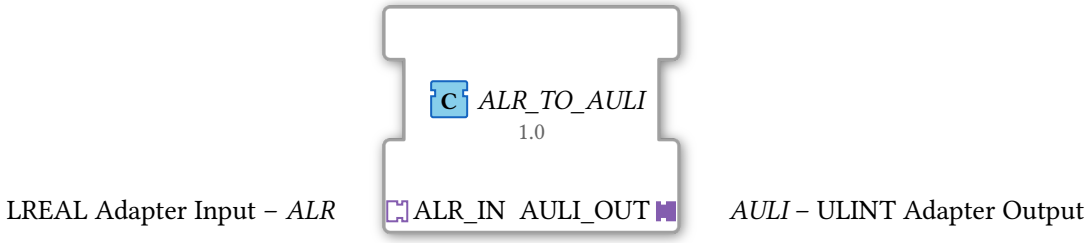
Ähnliche Konverterbausteine existieren für andere Adapterpaare (z. B. `AUI_TO_AREAL`, `AUI_TO_BOOL`). Im Gegensatz zu diesen übernimmt AUI_TO_ALR keine aktive Datenumformung, sondern verlässt sich auf die implizite Typwandlung durch die Adapterdefinition. Dies macht ihn besonders leichtgewichtig und geeignet für reine Durchleitungsaufgaben.

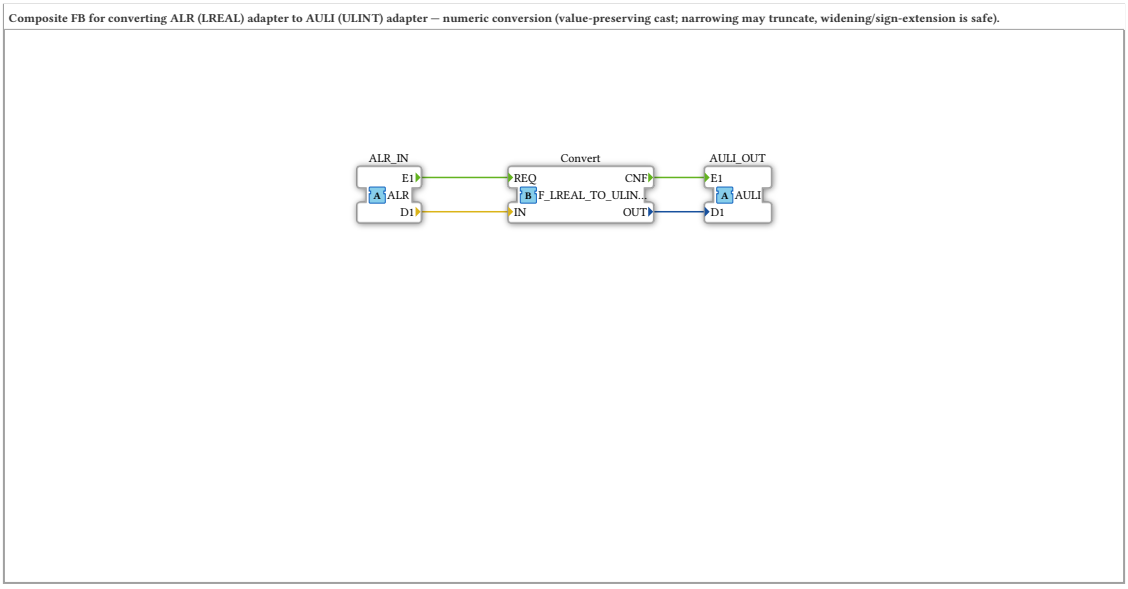
Fazit

Der AUI_TO_ALR ist ein minimaler, aber klar definierter Composite-FB, der die Verbindung zwischen einem AUI- und einem ALR-Adapter herstellt. Aufgrund seiner Einfachheit und Transparenz eignet er sich hervorragend für die strukturierte Verkabelung in IEC-61499-basierten Automatisierungssystemen, insbesondere in Umgebungen, in denen unterschiedliche Adapterprotokolle aufeinander abgestimmt werden müssen.

ALR_AULI

ALR_TO_AULI





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ALR_TO_AULI** ist ein **Composite FB** zur unidirektionalen Umwandlung eines **LREAL-Adapter-Signals** (Typ `ALR`) in ein **ULINT-Adapter-Signal** (Typ `AULI`). Durch die interne Verwendung des IEC-61131-Standard-FBs `F_LREAL_TO_ULINT` wird die Gleitkommazahl (LREAL) in eine vorzeichenlose Ganzzahl (ULINT) konvertiert und über einen Ausgangsadapter bereitgestellt. Der Baustein eignet sich besonders für die Integration von Gleitkomma-Datenquellen in Systeme, die eine ganzzahlige Adapter-Schnittstelle erwarten.

Schnittstellenstruktur

Der FB besitzt **keine direkten** Ereignis- oder Datenein-/-ausgänge. Die Kommunikation erfolgt ausschließlich über die beiden **Adapter** (Socket & Plug). Die nachfolgenden Tabellen beschreiben die über die Adapter bereitgestellten Signale.

Ereignis-Eingänge

Signal	Quelle	Datentyp	Beschreibung
E1	über Socket `ALR_IN`	(Ereignis)	Startet die Konvertierung eines eingehenden LREAL-Wertes.

Ereignis-Ausgänge

Signal	Ziel	Datentyp	Beschreibung
E1	über Plug `AULI_OUT`	(Ereignis)	Signalisiert, dass der konvertierte ULINT-Wert am Ausgang anliegt.

Daten-Eingänge

Signal	Quelle	Datentyp	Beschreibung
D1	über Socket `ALR_IN`	LREAL	Der umzuwandelnde Gleitkomma-Wert.

Daten-Ausgänge

Signal	Ziel	Datentyp	Beschreibung
D1	über Plug `AULI_OUT`	ULINT	Der konvertierte ganzzahlige Wert ohne Vorzeichen.

Adapter

Name	Typ	Richtung	Beschreibung
`ALR_IN`	`adapter::types::unidirectional::ALR`	Socket (Eingang)	Nimmt den LREAL-Wert sowie das dazugehörige Ereignis entgegen.
`AULI_OUT`	`adapter::types::unidirectional::AULI`	Plug (Ausgang)	Gibt den konvertierten ULINT-Wert mit einem Bestätigungsereignis aus.

Funktionsweise

Der FB arbeitet nach dem **Getriebe-Prinzip** eines Composite-Bausteins:

1. Ein eingehendes **Ereignis E1** am Socket `ALR_IN` aktiviert den internen FB `F_LREAL_TO_ULINT` (Typ `iec61131::conversion::F_LREAL_TO_ULINT`).
2. Gleichzeitig wird der am Socket `ALR_IN` anliegende **Datenwert D1** (LREAL) an den Eingang `IN` des internen FBs weitergeleitet.
3. Der interne FB führt sofort die Konvertierung von LREAL nach ULINT durch und gibt das Ergebnis an seinem Ausgang `OUT` aus.
4. Das Ereignis `CNF` des internen FBs triggert das **Ausgangsereignis E1** am Plug `AULI_OUT`.
5. Der konvertierte ULINT-Wert wird parallel an den Datenausgang `D1` des Plug `AULI_OUT` gelegt.

Die gesamte Wandlung erfolgt **ereignisgesteuert** und ohne eigene Zustandslogik.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Lizenz**: Der FB wird unter der **Eclipse Public License 2.0** bereitgestellt (siehe Copyright-Hinweis).

- **Standardkonformität**: Der intern verwendete Konvertierungsbaustein `F_LREAL_TO_ULINT` entspricht der IEC-61131-3-Norm.
- **Keine Zustandsmaschine**: Als Composite FB besitzt der Baustein keine eigene Zustandslogik – die Konvertierung ist rein kombinatorisch.
- **Unidirektionale Adapter**: Ein- und Ausgang sind getrennte Adaptertypen, die eine klare Signalrichtung vorgeben.

Zustandsübersicht

Da es sich um einen **Composite FB ohne eigene Zustandsmaschine** handelt, existiert keine explizite Zustandsübersicht. Die Funktionalität ist ereignisgetriggert und verhält sich wie eine **kombinatorische Funktion**: Ein Ereignis am Eingang führt nach einer internen Verarbeitung genau zu einem Ereignis am Ausgang.

Anwendungsszenarien

- **Anbindung von Gleitkomma-Sensoren** an eine Steuerung, die nur ganzzahlige Adapter-Signale verarbeitet (z. B. in der Agrartechnik).
- **Datenaufbereitung** in einer Adapter-basierten Kommunikationskette, wenn die Quelle LREAL liefert, das Ziel aber ULINT erwartet.
- **Ersatz von manuellen Typumwandlungen** in Systemen, die auf die standardisierten IEC-61131-Funktionen setzen.

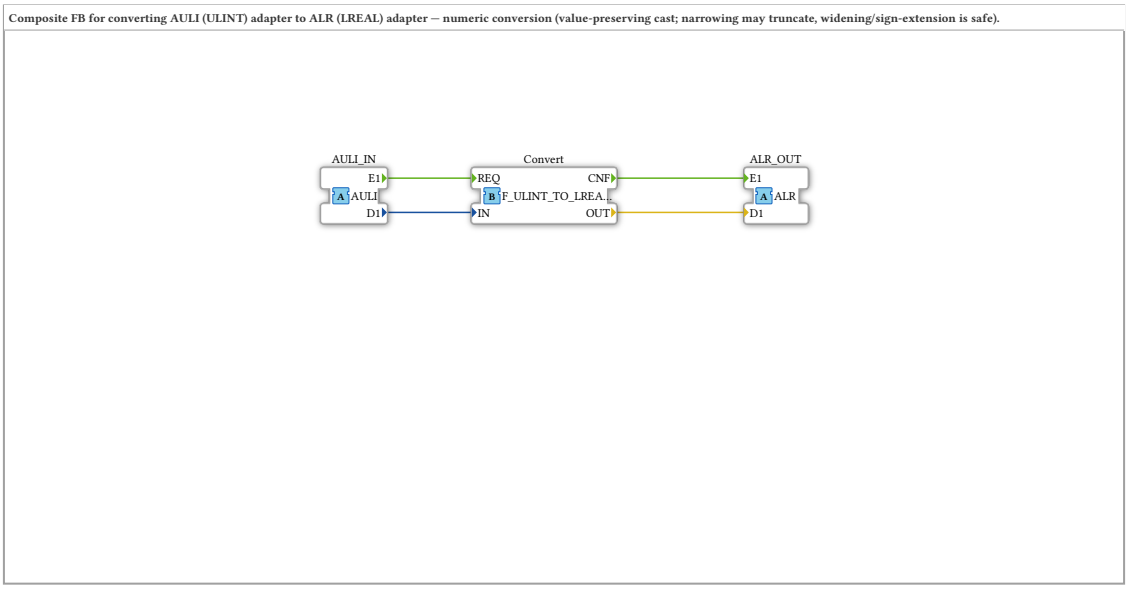
Vergleich mit ähnlichen Bausteinen

Andere Konvertierungsadapter (z. B. `REAL_TO_DINT`, `LREAL_TO_LINT`) folgen demselben Muster: Sie nutzen einen internen IEC-Funktionsbaustein und kapseln die Typumwandlung in einer Adapter-Schnittstelle. Der Vorteil des `ALR_TO_AULI` liegt in der **zielgerichteten Kombination aus LREAL und ULINT** – eine Typkombination, die in vielen Standard-Bibliotheken nicht als fertiger Adapter angeboten wird.

Fazit

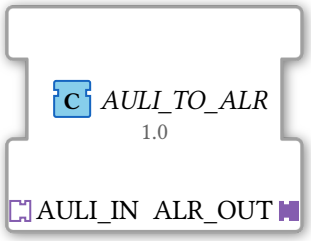
`ALR_TO_AULI` ist ein spezialisierter **Composite-Adapter-Baustein** zur verlustbehafteten Umwandlung von Gleitkomma- in Ganzzahlen. Dank der Verwendung eines normierten IEC-Bausteins ist die Konvertierung deterministisch und gut testbar. Der FB eignet sich hervorragend, um in heterogenen Automatisierungssystemen eine saubere Adapter-Schnittstelle zwischen LREAL- und ULINT-Domänen zu schaffen.

AULI_TO_ALR



Kein Bild vorhanden.

ULINT Adapter Input – *AULI*



ALR – LREAL Adapter Output

• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AULI_TO_ALR** ist ein zusammengesetzter Funktionsbaustein (Composite FB), der einen eingehenden AULI-Adapter (Daten vom Typ ULINT) in einen ausgehenden ALR-Adapter (Daten vom Typ LREAL) umwandelt. Er dient als standardisierte Schnittstelle zwischen Komponenten, die unterschiedliche Datentypen über Adapter kommunizieren.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine direkten Ereignis-Eingänge. Das eingehende Ereignis wird über den Socket-Adapter **AULI_IN** bereitgestellt:

- **AULI_IN.E1** (Ereignis) – Empfängt das Auslösesignal für die Umwandlung.

Ereignis-Ausgänge

Der Baustein besitzt keine direkten Ereignis-Ausgänge. Das ausgehende Ereignis wird über den Plug-Adapter **ALR_OUT** bereitgestellt:

- **ALR_OUT.E1** (Ereignis) – Signalisiert, dass die umgewandelten Daten am Datenausgang anliegen.

Daten-Eingänge

Direkte Daten-Eingänge sind nicht vorhanden. Die Eingangsdaten werden über den Socket-Adapter **AULI_IN** bereitgestellt:

- **AULI_IN.D1** (ULINT) – Der umzuwandelnde Wert im Format unsigned long integer.

Daten-Ausgänge

Direkte Daten-Ausgänge sind nicht vorhanden. Die Ausgangsdaten werden über den Plug-Adapter **ALR_OUT** bereitgestellt:

- **ALR_OUT.D1** (LREAL) – Der umgewandelte Wert im Format long real (64-Bit Gleitkommazahl).

Adapter

Der Baustein verwendet genau zwei Adapter:

- **Socket AULI_IN** (Typ: AULI aus `adapter::types::unidirectional`) – Empfängt die ULINT-Werte und das zugehörige Triggerereignis.
- **Plug ALR_OUT** (Typ: ALR aus `adapter::types::unidirectional`) – Stellt die konvertierten LREAL-Werte und ein Bestätigungsereignis zur Verfügung.

Funktionsweise

Der Baustein arbeitet nach dem Request-Bestätigungs-Prinzip:

1. Sobald am Eingangsadapter **AULI_IN** ein Ereignis an **E1** eintrifft, wird der interne Funktionsbaustein **F_ULINT_TO_LREAL** (aus der Bibliothek `iec61131::conversion`) aufgerufen.
2. Der Wert **AULI_IN.D1** wird dem Eingang **IN** des Konverters zugeführt.
3. Der Konverter führt die Umwandlung von ULINT nach LREAL durch (Ganzzahl zu Gleitkommazahl).
4. Nach Abschluss der Konvertierung wird das Ergebnis an **ALR_OUT.D1** weitergeleitet.
5. Gleichzeitig wird am Ausgangsadapter das Ereignis **ALR_OUT.E1** ausgelöst, um dem nachgeschalteten Baustein die Verarbeitung der Daten zu signalisieren.

Die gesamte Logik läuft innerhalb eines FB-Netzwerks ab und ist für den Anwender transparent.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Adapterbasierte Kommunikation:** Der Baustein kapselt die Typkonvertierung in einer Adapterschnittstelle, sodass er in Systemen mit einheitlichen Adaptertypen (z. B. AULI und ALR) nahtlos eingesetzt werden kann. Dadurch wird die Kompatibilität zwischen Komponenten erhöht.
- **Wiederverwendung vorhandener Bausteine:** Als Composite FB verwendet er den standardisierten Konvertierungsbaustein `F_ULINT_TO_LREAL`, was die Wartbarkeit und Korrektheit der Konvertierungslogik sicherstellt.
- **Kompatibilität mit IEC 61499-2:** Der Baustein erfüllt die Norm und kann in beliebigen 4diac-IDE Projekten verwendet werden.
- **Keine Zustandsmaschine:** Die Umwandlung erfolgt rein kombinatorisch unter Steuerung der Ereignisse – es gibt keine internen Zustände.

Zustandsübersicht

Der Baustein besitzt keine eigenständigen Zustände. Die Ablaufsteuerung wird durch das Ereignis an **AULI_IN.E1** angestoßen und endet mit dem Ereignis an **ALR_OUT.E1**. Während der Konvertierung befindet sich der Baustein im Zustand „warten auf Bestätigung des internen Funktionsbausteins“. Für den Anwender ist dies jedoch nicht sichtbar.

Anwendungsszenarien

- **Integration von Sensoren oder Aktoren,** die ihre internen Zählerwerte als ULINT liefern, in eine Steuerung, die für Berechnungen LREAL benötigt (z. B. Regelungsalgorithmen).
- **Adapterbrücke** zwischen zwei Subsystemen mit unterschiedlichen Adapterdefinitionen, um ohne Änderung der Schnittstellendefinitionen eine Typumwandlung zu ermöglichen.
- **Datenaufbereitung** in Kommunikationsprotokollen, bei denen empfangene Ganzzahlen in Gleitkommawerte für die weitere Verarbeitung umgerechnet werden müssen.

Vergleich mit ähnlichen Bausteinen

Statt eines speziellen Adapterkonverters könnte man auch direkt den Baustein `F_ULINT_TO_LREAL` verwenden und die Ereignisse/Daten manuell verbinden. Der Vorteil von **AULI_TO_ALR** liegt in der Kapselung der Adapterschnittstellen: Er erspart dem Entwickler das manuelle Verkabeln der Adapter-Ereignisse und -Daten und sorgt für eine saubere, wiederverwendbare Komponente. Gegenüber einer maßgeschneiderten Lösung bietet er eine standardisierte, dokumentierte Schnittstelle.

Fazit

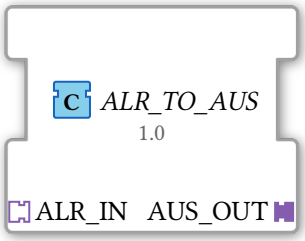
Der **AULI_TO_ALR** ist ein nützlicher, spezialisierter Funktionsbaustein für die Umwandlung von ULINT- in LREAL-Werte im Kontext von Adaptern. Er vereinfacht die Einbindung von Komponenten mit unterschiedlichen Datentypen in einheitliche Adapterstrukturen und erhöht die Wartbarkeit von IEC-61499-Systemen. Durch die Verwendung einer standardisierten Konvertierungsfunktion und seiner klaren, ereignisgesteuerten Arbeitsweise ist er zuverlässig und leicht verständlich.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

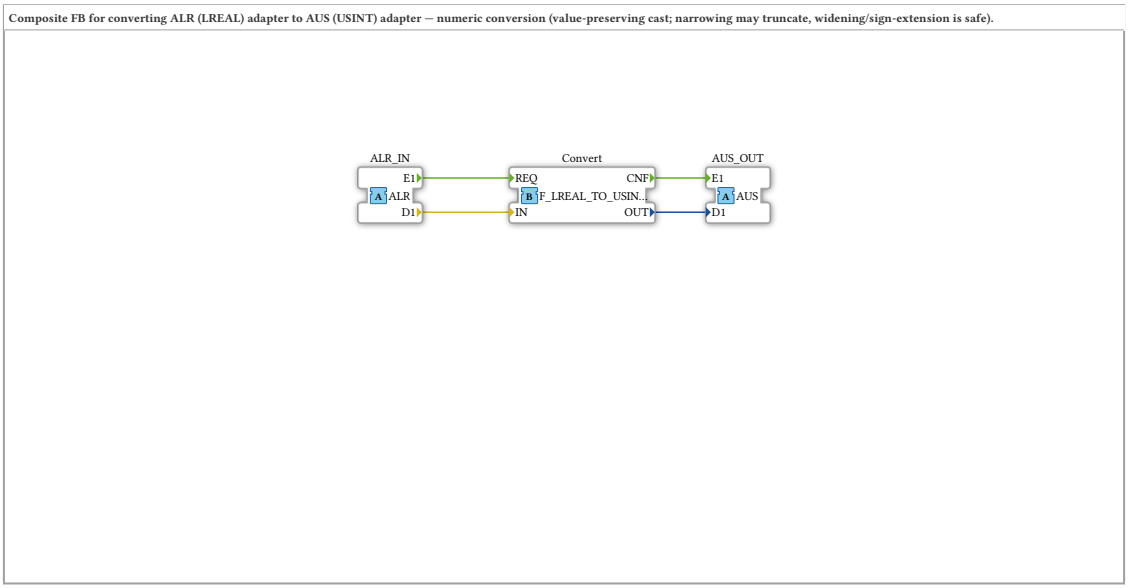
ALR_AUS

ALR_TO_AUS



LREAL Adapter Input – *ALR*

AUS – USINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Composite-Funktionsblock **ALR_TO_AUS** dient als universeller Konverter zwischen zwei unidirektionalen Adapterprotokollen. Er wandelt einen LREAL-Wert (64-Bit Gleitkomma) in einen USINT-Wert (8-Bit vorzeichenlose Ganzzahl) um. Der Block wurde für den Einsatz in IEC 61499-basierten Automatisierungssystemen entwickelt und nutzt intern den Standard-Funktionsblock `F_LREAL_TO_USINT` aus der Bibliothek `iec61131::conversion`.

Schnittstellenstruktur

Da es sich um einen Composite-Baustein handelt, werden alle Ein- und Ausgänge über Adapter realisiert.

Ereignis-Eingänge

- Keine direkten Ereignis-Eingänge auf der obersten Ebene. Der über den **Socket ALR_IN** angebundene Adapter besitzt jedoch den Ereignis-Eingang **E1**, der die Konvertierung auslöst.

Ereignis-Ausgänge

- Keine direkten Ereignis-Ausgänge auf der obersten Ebene. Der über den **Plug AUS_OUT** angebundene Adapter stellt den Ereignis-Ausgang **E1** bereit, der den Abschluss der Konvertierung signalisiert.

Daten-Eingänge

- Keine direkten Daten-Eingänge auf der obersten Ebene. Der Adapter **ALR_IN** führt den Daten-Eingang **D1** (Typ: LREAL), der den zu konvertierenden Gleitkommawert bereitstellt.

Daten-Ausgänge

- Keine direkten Daten-Ausgänge auf der obersten Ebene. Der Adapter **AUS_OUT** führt den Daten-Ausgang **D1** (Typ: USINT), der das konvertierte Ganzzahlergebnis ausgibt.

Adapter

Name	Typ	Richtung	Kommentar
ALR_IN	adapter::types::unidirectional::ALR	Socket	Unidirektionaler LREAL-Eingang
AUS_OUT	adapter::types::unidirectional::AUS	Plug	Unidirektionaler USINT-Ausgang

Funktionsweise

Der Funktionsblock arbeitet ereignisgesteuert:

1. Ein eingehendes Ereignis am Adapter-Eingang `ALR_IN.E1` wird intern mit dem **REQ**-Eingang des Konvertierungsbausteins `F_LREAL_TO_USINT` verbunden.
2. Gleichzeitig wird der Datenwert `ALR_IN.D1` an den Daten-Eingang `IN` des Konvertierungsbausteins übergeben.
3. Der Konverter führt die Umwandlung von LREAL nach USINT durch. Hierbei wird der Gleitkommawert gemäß IEC 61131-3 auf den nächstliegenden ganzzahligen USINT-Wert (0–255) gerundet oder abgeschnitten.
4. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` des Konverters erzeugt, das direkt an den Adapter-Ausgang `AUS_OUT.E1` weitergeleitet wird.
5. Das Ergebnis `OUT` des Konverters wird an den Daten-Ausgang `AUS_OUT.D1` übergeben.

Dadurch wird eine synchrone, datengekoppelte Umsetzung ohne zusätzliche Verzögerung durchgeführt.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-Aufbau**: Der Block kapselt die Konvertierungslogik in einer wiederverwendbaren Einheit und nutzt die standardisierten Adaptertypen `ALR` und `AUS`. Dadurch kann er nahtlos in bestehende unidirektionale Kommunikationsprotokolle eingebunden werden.
- **Standardkonforme Umwandlung**: Die eigentliche Konvertierung erfolgt durch den IEC 61131-Baustein `F_LREAL_TO_USINT`, der ein definiertes Verhalten (z.B. Runden oder Kappen) gemäß der Norm garantiert.
- **EPL-2.0 Lizenz**: Der Baustein ist unter der Eclipse Public License 2.0 veröffentlicht, was eine uneingeschränkte Nutzung in proprietären und Open-Source-Projekten ermöglicht.
- **Keine Zustandsmaschine**: Als Composite besitzt der Block keinen eigenen Zustandsautomaten; die interne Steuerung wird vollständig durch den verwendeten Konverterbaustein übernommen.

Zustandsübersicht

Der Baustein selbst besitzt keine explizite Zustandsmaschine. Der eingebettete Funktionsblock `F_LREAL_TO_USINT` realisiert typischerweise die Zustände **IDLE** und **BUSY** sowie **ERROR** bei dieser Umwandlung. Bei ungültigen Eingaben (z.B. Überlauf des USINT-Bereichs) kann der Konverter einen Fehler auslösen, der jedoch in dieser Version nicht gesondert an die Adapterausgänge weitergegeben wird.

Anwendungsszenarien

- **Sensorik / Aktorik-Anbindung**: Wenn ein Gleitkomma-Sensorwert (z.B. Druck, Temperatur) in ein 8-Bit-Signal für einen einfachen Aktor (z.B. Schalterstellung, Helligkeit) umgesetzt werden muss.
- **Protokollanpassung**: In Systemen, die zwischen verschiedenen Adaptertypen für die Datenübertragung vermitteln, wie z.B. in Feldbus-Kopplern oder Gateway-Funktionen.
- **Test und Simulation**: Zur einfachen Umwandlung von LREAL-Testdaten in USINT-Formate, um die Funktionsweise nachgeschalteter Bausteine zu prüfen.

Vergleich mit ähnlichen Bausteinen

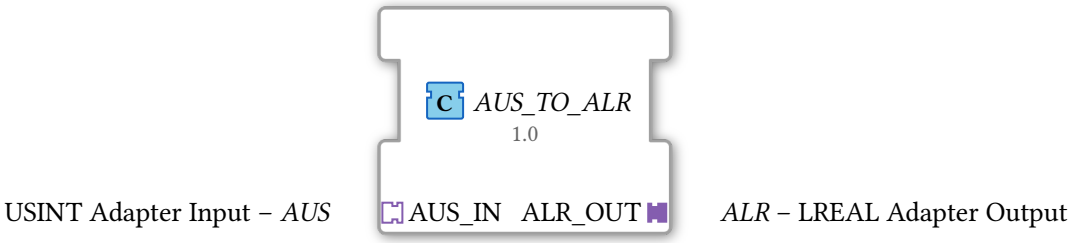
- **LREAL_TO_INT / LREAL_TO_DWORD**: Diese Bausteine liefern andere Ausgabetypen (INT, DWORD) und sind daher für Zielplattformen mit breiteren oder vorzeichenbehafteten Ganzzahlen geeignet.
- **REAL_TO_USINT**: Für 32-Bit Gleitkommazahlen (REAL) steht eine entsprechende Variante zur Verfügung. Der vorliegende Baustein nutzt jedoch den präziseren LREAL-Typ.

- **Skalierung mit arithmetischen Bausteinen**: Eine manuelle Umrechnung mittels `MUL` und `ADD` wäre möglich, aber weniger standardisiert und fehleranfällig. Der `ALR_TO_AUS`-Block bietet eine geprüfte, normenkonforme Lösung.

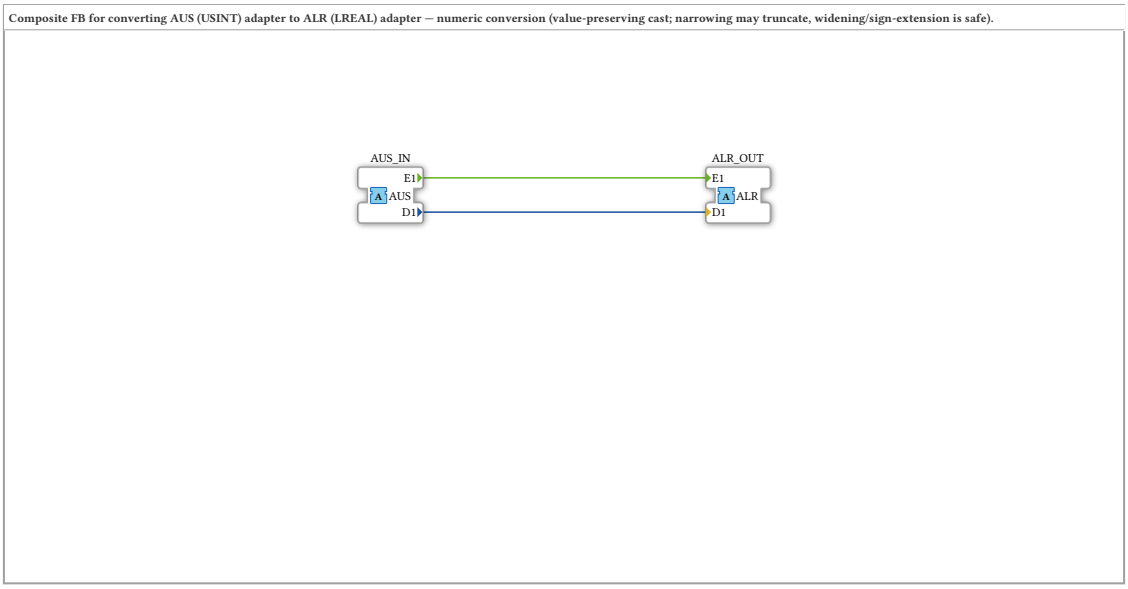
Fazit

Der Funktionsblock **ALR_TO_AUS** ist ein spezialisierter, aber unverzichtbarer Helfer für die Umwandlung von LREAL- in USINT-Daten in IEC 61499-Applikationen. Sein Composite-Design erlaubt eine saubere Trennung der Konvertierungslogik und eine einfache Integration über standardisierte Adapter. Er eignet sich besonders für Szenarien, in denen präzise Gleitkommawerte auf einen engen Ganzzahlbereich abgebildet werden müssen.

AUS_TO_ALR



USINT Adapter Input – *AUS* — *ALR* – LREAL Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUS_TO_ALR** ist ein Composite-Baustein, der einen **AUS-Adapter** (mit USINT-Daten) in einen **ALR-Adapter** (mit LREAL-Daten) umwandelt. Er dient als reine Weiterleitungsschnittstelle und führt keine komplexe Logik oder Konvertierungsberechnung durch.

Schnittstellenstruktur

Ereignis-Eingänge

Keine separaten Ereignis-Eingänge vorhanden. Die Ereignisweiterleitung erfolgt über den **AUS_IN**-Adapter.

Ereignis-Ausgänge

Keine separaten Ereignis-Ausgänge vorhanden. Die Ereignisweiterleitung erfolgt über den **ALR_OUT**-Adapter.

Daten-Eingänge

Keine separaten Daten-Eingänge vorhanden. Die Datenweiterleitung erfolgt über den **AUS_IN**-Adapter.

Daten-Ausgänge

Keine separaten Daten-Ausgänge vorhanden. Die Datenweiterleitung erfolgt über den **ALR_OUT**-Adapter.

Adapter

Rolle	Name	Typ	Kommentar
Socket (Eingang)	AUS_IN	`adapter::types::unidirectional::AUS`	USINT-Adapter-Eingang
Plug (Ausgang)	ALR_OUT	`adapter::types::unidirectional::ALR`	LREAL-Adapter-Ausgang

- Der **AUS_IN**-Adapter stellt die Schnittstelle für eingehende USINT-Daten inklusive eines zugehörigen Ereignisses bereit.
- Der **ALR_OUT**-Adapter stellt die Schnittstelle für ausgehende LREAL-Daten inklusive eines zugehörigen Ereignisses bereit.
- Im Inneren des Bausteins werden die Ereignisse (`E1`) und Daten (`D1`) direkt von der *AUS_IN*-Schnittstelle auf die *ALR_OUT*-Schnittstelle durchverbunden.

Funktionsweise

1. **Ereignisweiterleitung**: Das am **AUS_IN**-Adapter anliegende Ereignis (`E1`) wird unverändert an den **ALR_OUT**-Adapter (`E1`) weitergegeben.
2. **Datenweiterleitung**: Der Datenwert (`D1`) vom **AUS_IN**-Adapter wird direkt an den **ALR_OUT**-Adapter (`D1`) übergeben.
3. **Datentypumwandlung**: Die eigentliche Umwandlung von `USINT` (ganzzahliger 8-Bit-Wert) in `LREAL` (64-Bit-Gleitkommazahl) erfolgt **nicht innerhalb des FB** – hierfür ist der **AUS_TO_ALR**-Block lediglich eine strukturelle Hülle. Die Konvertierung muss entweder durch die angeschlossenen Adapter oder durch vorgelagerte Funktionsbausteine realisiert werden.

Der Baustein selbst enthält keine Berechnungslogik und keine Zustandsmaschine; er besteht ausschließlich aus Verbindungen zwischen den Adaptern.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-FB**: Der Baustein ist als Komposit (Composite Function Block) realisiert, dessen internes Netzwerk nur zwei Verbindungen enthält.
- **Keine eigene Logik**: Alle Funktionalität wird durch externe Bausteine oder die Adapter-Implementierung bereitgestellt.
- **Typkonformität**: Die Verbindung von `AUS.D1` (USINT) zu `ALR.D1` (LREAL) setzt voraus, dass die zugrundeliegende Adapter-Implementierung eine implizite Typumwandlung erlaubt. Bei Vorsicht ist eine explizite Konvertierung außerhalb des FB notwendig.

Zustandsübersicht

Der Baustein besitzt **keine eigene Zustandsmaschine**. Er arbeitet transparent: Sobald ein Ereignis am Eingangsadapter eintrifft, wird es sofort an den Ausgangsadapter weitergeleitet. Es gibt keine Verzögerungen, Speicherung oder interne Zustände.

Anwendungsszenarien

- **Anbindung heterogener Steuerungssysteme**: Ein Sensor liefert Werte als USINT über einen AUS-Adapter, während das nachfolgende Steuerungsmodul LREAL über einen ALR-Adapter erwartet.
- **Strukturelle Anpassung**: Der FB dient als reiner Umsetzungsadapter, wenn Protokoll-/Schnittstellen-Adaptertypen unterschiedliche Datenformate verwenden.
- **Test- und Simulationsumgebungen**: Einfaches Durchschleifen von Daten und Ereignissen zur Erprobung von Adapter-Kommunikation.

Vergleich mit ähnlichen Bausteinen

Merkmal	AUS_TO_ALR	Typische Konvertierungs-FBs (z.B. USINT_TO_LREAL)
Eingang / Ausgang	Adapter-Schnittstellen	Einzelne Daten-Eingänge/-Ausgänge
Ereignisverarbeitung	Durchleitung über Adapter	Eigene Ereignis-E/K-Paare
Implementierung	Composite (nur Verbindungen)	Ausführbare Algorithmen
Typumwandlung	Nicht im FB enthalten	Explizite Konvertierung durch Algorithmus

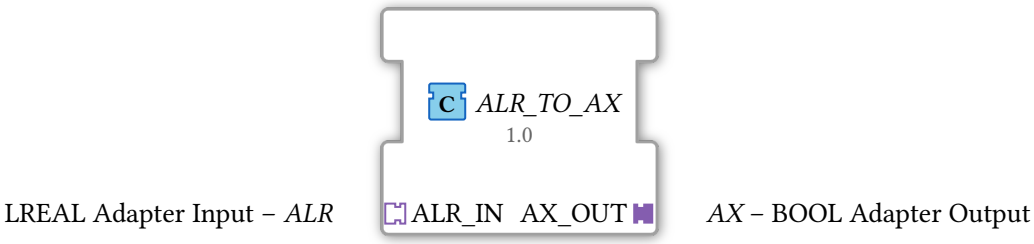
Während ein klassischer *USINT_TO_LREAL*-FB die Wertkonvertierung selbst vornimmt, bietet **AUS_TO_ALR** nur die Adapter-Anpassung. Für die eigentliche Datenumwandlung muss ein separater Funktionsbaustein zwischengeschaltet werden.

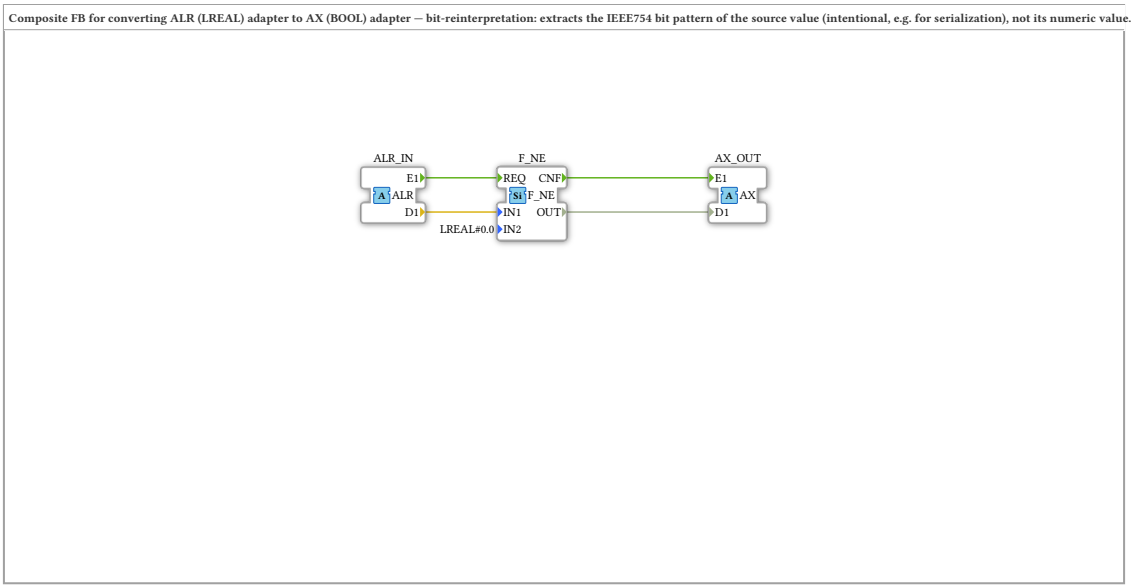
Fazit

Der **AUS_TO_ALR**-Funktionsblock ist ein einfacher, aber nützlicher Composite-Baustein zur strukturellen Adaption zwischen einem AUS- und einem ALR-Adapter. Er übernimmt keine Datenkonvertierung, sondern ermöglicht die nahtlose Weiterleitung von Ereignissen und Werten zwischen unterschiedlichen Adapter-Schnittstellen. Aufgrund seiner minimalistischen Ausführung eignet er sich besonders für Integrations- und Testaufgaben, bei denen die eigentliche Typumwandlung extern erfolgt.

ALR_AX

ALR_TO_AX





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ALR_TO_AX** ist ein Composite-Baustein zur Umwandlung eines LREAL-Werts (über den unidirektionalen ALR-Adapter) in einen BOOL-Wert (über den unidirektionalen AX-Adapter). Er wird typischerweise eingesetzt, um ein analoges Eingangssignal in ein digitales Ausgangssignal zu

konvertieren – beispielsweise zur Erkennung einer Schwellwertüberschreitung. Der Baustein basiert auf einem einfachen Null-Vergleich und löst das Ausgangsereignis aus, sobald ein gültiger Wert am Eingang anliegt.

Schnittstellenstruktur

Ereignis-Eingänge
- **ALR_IN.E1** – Ereignis vom ALR-Adapter; zeigt an, dass ein neuer LREAL-Wert bereitsteht.

Ereignis-Ausgänge
- **AX_OUT.E1** – Ereignis zum AX-Adapter; signalisiert, dass der konvertierte BOOL-Wert gültig ist.

Daten-Eingänge
- **ALR_IN.D1** – LREAL-Eingangswert (über den ALR-Adapter).

Daten-Ausgänge
- **AX_OUT.D1** – BOOL-Ausgangswert (über den AX-Adapter); ergibt sich aus dem Vergleich des Eingangswerts mit 0.0.

Adapter
- **ALR_IN** – Socket (Typ: *adapter::types::unidirectional::ALR*): unidirektionaler LREAL-Eingangsadapter.
- **AX_OUT** – Plug (Typ: *adapter::types::unidirectional::AX*): unidirektionaler BOOL-Ausgangsadapter.

Funktionsweise

Bei einem ankommenden Ereignis über **ALR_IN.E1** wird der Datenwert **ALR_IN.D1** (LREAL) an den internen IEC-61131-Baustein **F_NE** übergeben. Dieser vergleicht den Wert mit der Konstanten `LREAL#0.0` und erzeugt **TRUE**, falls der Eingang ungleich null ist, andernfalls **FALSE**. Das Ergebnis wird direkt an **AX_OUT.D1** weitergeleitet, und ein Bestätigungsereignis über **AX_OUT.E1** wird ausgelöst.

Der gesamte Ablauf erfolgt innerhalb eines einzigen Zyklus, da es sich um einen Composite-Block ohne eigenen Zustandsautomaten handelt.

Technische Besonderheiten
- **Bit-Reinterpretation**: extrahiert das IEEE754-Bitmuster des Quellwerts (beabsichtigt, z. B. zur Serialisierung), nicht dessen Zahlenwert.
- **Lizenz & Herkunft**: Der Baustein ist unter der Eclipse Public License 2.0 lizenziert und wurde mit der Versionsinformation 1.0 bereitgestellt.
- **Verwendete Bibliotheken**: Der Vergleichsbaustein *F_NE* stammt aus der IEC-61131-Bibliothek (*iec61131::comparison*).
- **Paketname**: Der FB gehört zum Paket *adapter::conversion::unidirectional*.
- **Kompatibilität**: Er setzt die Adapter-Spezifikation des 4diac-Frameworks voraus und ist für unidirektionale Adaptertypen ausgelegt.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten. Seine Logik ist rein kombinatorisch und ereignisgesteuert:

1. Warten auf Ereignis an **ALR_IN.E1**.
2. Datenverarbeitung (Vergleich mit 0.0).
3. Ausgabe des BOOL-Ergebnisses an **AX_OUT.D1** und Senden des Bestätigungsereignisses an **AX_OUT.E1**.

Anwendungsszenarien
- **Analog-Digital-Umwandlung**: Erkennung, ob ein analoger Messwert (z. B. Drucksensor, Temperatur) von null verschieden ist.
- **Schwellwert-Erkennung**: Durch einfache Modifikation (Austausch von *F_NE* gegen *F_GT* oder *F_LT*) kann eine beliebige Schwellwertlogik realisiert werden.
- **Adapter-Kopplung**: Einbinden von ALR-basierten Sensoren in eine AX-basierte Steuerungslogik, ohne dass komplexe Typkonvertierungen auf übergeordneter Ebene nötig sind.

Vergleich mit ähnlichen Bausteinen

- **LREAL_TO_BOOL**: Während ein dedizierter LREAL-to-BOOL-Baustein oft einen Schwellwert und Hysterese unterstützt, realisiert *ALR_TO_AX* die einfachste Form – null ≠ null ⟶ TRUE.
- **Vergleichsbausteine (F_NE, F_EQ usw.)**: Diese arbeiten direkt auf Daten, während *ALR_TO_AX* die vollständige Adapter-Integration und Ereignissteuerung übernimmt.
- **Adapter-Converter-Bausteine**: Im selben Paket finden sich weitere uni-/bidirektionale Konverter, die ähnliche Aufgaben für andere Typen (z. B. BOOL_TO_ALR) erfüllen.

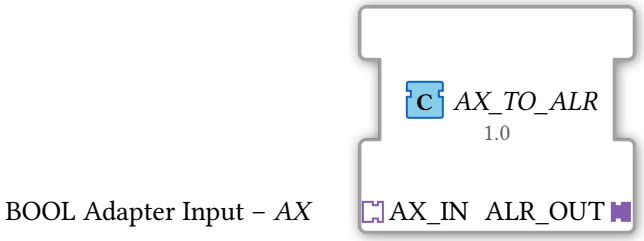
Fazit

Der **ALR_TO_AX**-Funktionsblock bietet eine schlanke, zuverlässige Möglichkeit, einen LREAL-Adapterausgang in einen BOOL-Adaptereingang umzusetzen. Dank seiner einfachen Struktur eignet er sich hervorragend für grundlegende Digitalisierungsaufgaben in IEC-61499-basierten Steuerungssystemen. Er ist leicht verständlich, gut erweiterbar und folgt den gängigen Entwurfsmustern für unidirektionale Adapterkonvertierung.

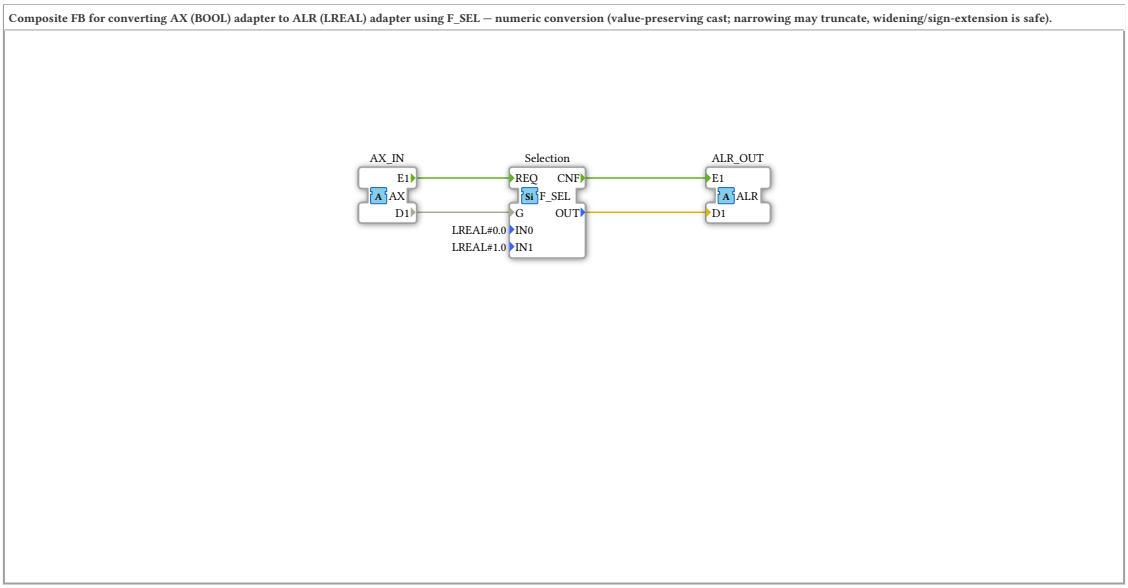
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AX_TO_ALR



BOOL Adapter Input – *AX* — *ALR* – LREAL Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AX_TO_ALR** dient zur Umwandlung eines BOOL-Adapter-Signals (AX) in ein LREAL-Adapter-Signal (ALR). Er setzt intern die IEC-61131-Standardfunktion `F_SEL` (Selection) ein, um aus einem booleschen Eingang einen diskreten LREAL-Wert zu generieren. Der Baustein ist als Composite FB realisiert und eignet sich für die einfache, adapterbasierte Konvertierung von digitalen Zuständen in numerische Werte.

Schnittstellenstruktur

Ereignis-Eingänge

Bezeichnung	Datentyp	Beschreibung
AX_IN.E1	EVENT	Ereignis vom angeschlossenen AX-Adapter; löst die Verarbeitung aus.

Ereignis-Ausgänge

Bezeichnung	Datentyp	Beschreibung
ALR_OUT.E1	EVENT	Bestätigungsereignis, nachdem der konvertierte Wert am ALR-Adapter ausgegeben wurde.

Daten-Eingänge

Bezeichnung	Datentyp	Beschreibung
AX_IN.D1	BOOL	Boolescher Wert, der in einen LREAL-Wert umgesetzt werden soll.

Daten-Ausgänge

Bezeichnung	Datentyp	Beschreibung
ALR_OUT.D1	LREAL	Ergebnis der Konvertierung: 0.0 bei FALSE, 1.0 bei TRUE.

Adapter

Richtung	Bezeichnung	Typ	Beschreibung
Socket (Eingang)	AX_IN	adapter::types::unidirectional::AX	Nimmt ein BOOL-Signal über den unidirektionalen AX-Adapter entgegen.
Plug (Ausgang)	ALR_OUT	adapter::types::unidirectional::ALR	Stellt das konvertierte LREAL-Signal über den unidirektionalen ALR-Adapter bereit.

Funktionsweise

Der FB arbeitet nach folgendem Schema:

1. Ein eingehendes Ereignis an AX_IN.E1 triggert die interne Verarbeitung.
2. Der boolesche Wert von AX_IN.D1 wird an den Gate-Eingang (G) der eingebetteten F_SEL-Funktion übergeben.
3. F_SEL wählt – je nach booleschem Wert – einen der beiden Konstantenwerte aus:
   - FALSE (G = 0) ⟶ Ausgabe von **0.0** (LREAL#0.0)
   - TRUE (G = 1) ⟶ Ausgabe von **1.0** (LREAL#1.0)
4. Das Ergebnis wird an ALR_OUT.D1 weitergeleitet und gleichzeitig das Bestätigungsereignis ALR_OUT.E1 ausgegeben.

Die Konvertierung erfolgt ereignisgesteuert: Jedes ankommende Ereignis bewirkt genau eine Umwandlung des aktuellen BOOL-Werts.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Adapter-Schnittstelle**: Der FB verwendet ausschließlich unidirektionale Adapter (Socket/Plug), wodurch er sich nahtlos in adapterbasierte Architekturen einfügt.
- **Interner Einsatz von F_SEL**: Die IEC-61131-Standardfunktion garantiert ein deterministisches und portables Verhalten.

- **Feste Schwellwerte**: Die Ausgangswerte sind auf `0.0` und `1.0` festgelegt. Eine Anpassung erfordert eine Modifikation des inneren Netzwerks.
- **Paketierung**: Der FB ist im Paket `adapter::conversion::unidirectional` abgelegt.
- **Lizenz**: Der Baustein ist unter der Eclipse Public License 2.0 verfügbar.

Zustandsübersicht

Da es sich um einen Composite FB ohne eigenen Ablaufzustand handelt, ergibt sich die Zustandslogik aus der Ereignis- und Datenverarbeitung des enthaltenen `F_SEL`-Bausteins. Es existieren zwei stabile Zustände:

- **Warten auf Ereignis**: Kein Ereignis an `AX_IN.E1` – keine Ausgabe.
- **Verarbeitung aktiv**: Ein Ereignis wird verarbeitet: der boolesche Wert wird gelesen, in einen LREAL-Wert umgewandelt und am Ausgang bereitgestellt; anschließend wird das Bestätigungsereignis gesendet.

Anwendungsszenarien

- **Digital-Analog-Umsetzung**: Ein digitaler Sensor (z. B. Schalter, Grenztaster) liefert einen BOOL-Wert, der in der Steuerung als LREAL-Signal (0.0/1.0) weiterverarbeitet werden soll.
- **Adapter-Integration**: Einbinden in bestehende Systeme, die bereits unidirektionale AX- und ALR-Adapter verwenden – z. B. in der Landtechnik oder der industriellen Automatisierung.
- **Skalierung von Zuständen**: Vereinfachte Umwandlung von Ein-/Aus-Signalen in numerische Größen für Berechnungen oder Visualisierungen.

Vergleich mit ähnlichen Bausteinen

Baustein	Eingangstyp	Ausgangstyp	Besonderheit
`AX_TO_ALR`	BOOL (Adapter)	LREAL (Adapter)	Adapter-basiert, feste Werte 0.0 / 1.0
`BOOL_TO_REAL`	BOOL (direkt)	REAL (direkt)	Konventioneller IEC-Baustein, Werte frei konfigurierbar
`SEL` / `F_SEL`	BOOL + 2x ANY	ANY	Allgemeine Auswahl ohne Adapter, Werte parameterisierbar

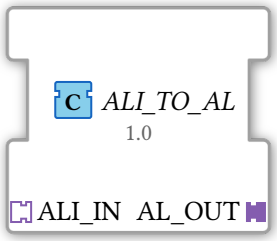
Gegenüber direkten Funktionsbausteinen bietet `AX_TO_ALR` den Vorteil der sauberen Adapter-Kapselung und der einfachen Wiederverwendung in adapterorientierten Projekten.

Fazit

Der FB `AX_TO_ALR` stellt eine kompakte und zuverlässige Lösung zur Umwandlung eines booleschen Adaptersignals in einen LREAL-Adapterwert dar. Durch den Einsatz der IEC-Normfunktion `F_SEL` ist das Verhalten deterministisch und portabel. Er eignet sich besonders für Anwendungen, die eine saubere Trennung von digitalen und analogen Signalpfaden über Adapter erfordern.

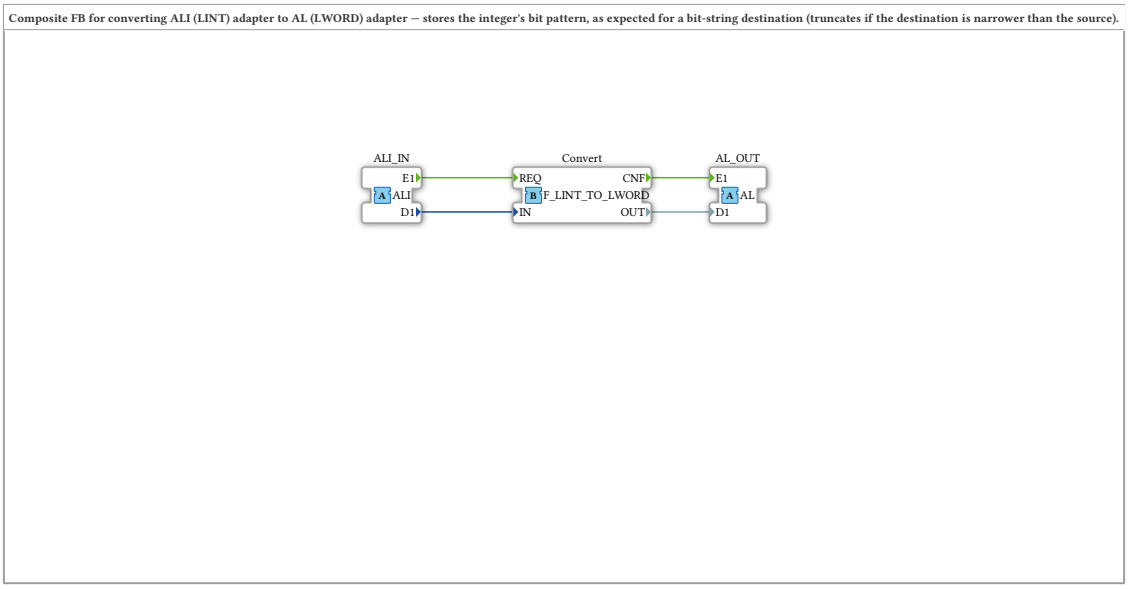
AL_ALI

ALI_TO_AL



LINT Adapter Input – *ALI*

AL – LWORD Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ALI_TO_AL** ist ein Composite-Baustein, der einen **ALI-Adapter** (Datentyp LINT) in einen **AL-Adapter** (Datentyp LWORD) umwandelt. Er dient als Schnittstellenkonverter zwischen unterschiedlichen Adaptertypen und ermöglicht die nahtlose Datenweitergabe in 4diac- bzw. IEC-61499-Systemen.

Schnittstellenstruktur

Ereignis-Eingänge

- **Keine direkten Ereigniseingänge** Ereignisse werden ausschließlich über den Socket-Adapter **ALI_IN** empfangen.

Ereignis-Ausgänge

- **Keine direkten Ereignisausgänge** Ausgangsereignisse werden ausschließlich über den Plug-Adapter **AL_OUT** gesendet.

Daten-Eingänge

- **Keine direkten Dateneingänge** Eingangsdaten (LINT) werden über den Socket-Adapter **ALI_IN** (Datenpunkt D1) bereitgestellt.

Daten-Ausgänge

- **Keine direkten Datenausgänge** Ausgangsdaten (LWORD) werden über den Plug-Adapter **AL_OUT** (Datenpunkt D1) abgegeben.

Adapter

Adapter	Richtung	Typ	Beschreibung
ALI_IN	Socket	`adapter::types::unidirectional::ALI`	Empfängt einen LINT-Wert sowie ein Ereignis.
AL_OUT	Plug	`adapter::types::unidirectional::AL`	Sendet einen LWORD-Wert sowie ein Ereignis.

Funktionsweise

Der Baustein arbeitet intern mit einem einzelnen Konvertierungs-FB `F_LINT_TO_LWORD`. Sobald am Socket-Adapter **ALI_IN** ein Ereignis (E1) eintrifft, wird dieses an den internen Konverter (`REQ`-Eingang)

weitergeleitet. Gleichzeitig wird der vom Adapter bereitgestellte LINT-Wert (D1) an den `IN`-Eingang des Konverters übergeben.

Der Konverter wandelt den LINT-Wert in einen LWORD-Wert um und signalisiert mit seinem `CNF`-Ausgang die Fertigstellung. Dieses Ereignis wird an den Plug-Adapter **AL_OUT** (E1) gesendet, zusammen mit dem konvertierten LWORD-Wert (OUT) auf dem Datenkanal D1.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Composite-FB**: Die eigentliche Typumwandlung erfolgt durch den eingebetteten Baustein `F_LINT_TO_LWORD` aus der IEC-61131-Standardbibliothek.
- **Unidirektionale Adapter**: Der Daten- und Ereignisfluss ist von ALI_IN zu AL_OUT festgelegt. Eine Rückwärtskommunikation ist nicht vorgesehen.
- **Keine Eigenzustände**: Der Baustein besitzt keinen eigenen Ablaufzustand (kein ECC), sondern leitet Ereignisse und Daten direkt weiter.

Zustandsübersicht

Der Funktionsblock verfügt über keine internen Zustände. Er reagiert streng ereignisgesteuert: Jedes eingehende Ereignis am Socket **ALI_IN** löst genau eine Konvertierung und eine Ausgabe am Plug **AL_OUT** aus.

Anwendungsszenarien

- **Adapterbrücke**: Einsatz in Systemen, die einen ALI-Adapter (LINT) verwenden, aber eine AL-Schnittstelle (LWORD) bereitstellen müssen.
- **Datenkonvertierung**: Umwandlung von LINT-Werten (z. B. lange Zählerstände) in LWORD-Werte (z. B. Bitmasken oder Speicheradressen) innerhalb einer Adapterkette.
- **Kompatibilität**: Nachrüstung älterer Komponenten, die LINT erwarten, in eine Umgebung, die LWORD über AL-Adapter nutzt.

Vergleich mit ähnlichen Bausteinen

- **AL_TO_ALI**: Führt die umgekehrte Konvertierung von LWORD nach LINT durch.
- **F_LINT_TO_LWORD**: Reiner Konvertierungs-FB ohne Adaptereinbindung; direkter Datenaustausch ohne Kapselung.
- **Andere Adapterkonverter** (z. B. AL_TO_AL): Gleicher Adaptertyp, keine Typumwandlung.

Der ALI_TO_AL-Baustein hebt sich durch seine einfache Plug-and-Play-Fähigkeit innerhalb eines Adapter-basierten Designs hervor, da er die gesamte Konvertierungslogik und Ereignissteuerung in einem Baustein kapselt.

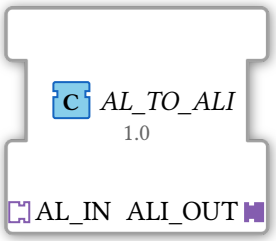
Fazit

Der Composite-FB **ALI_TO_AL** ermöglicht eine saubere und standardkonforme Umwandlung von LINT- auf LWORD-Daten über unidirektionale Adapter. Er ist ideal für den Einsatz in modularen Automatisierungssystemen, die unterschiedliche Adapterdatentypen integrieren müssen. Durch die Nutzung der bewährten IEC-61131-Konvertierungsfunktion ist eine zuverlässige und performante Ausführung gewährleistet.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

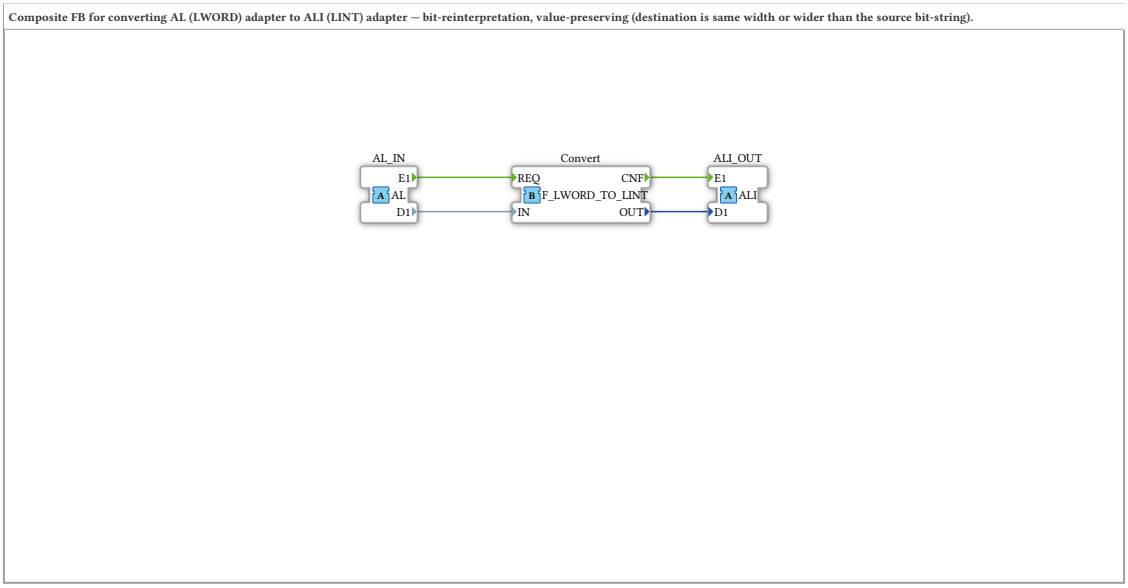
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AL_TO_ALI



LWORD Adapter Input – *AL*

ALI – LINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `AL_TO_ALI` ist ein zusammengesetzter Baustein (Composite FB) zur Umwandlung eines AL-Adapters (LWORD) in einen ALI-Adapter (LINT). Er dient als Schnittstelle zwischen einer LWORD-basierten und einer LINT-basierten Datenkommunikation innerhalb der Steuerungslogik. Der Baustein nutzt intern den Konvertierungsbaustein `F_LWORD_TO_LINT`, um die eigentliche Typumwandlung durchzuführen.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine eigenständigen Ereignis-Eingänge. Die Ereignissteuerung erfolgt über die angeschlossenen Adapter.

Ereignis-Ausgänge

Der Baustein besitzt keine eigenständigen Ereignis-Ausgänge. Die Ereignissteuerung erfolgt über die angeschlossenen Adapter.

Daten-Eingänge

Der Baustein besitzt keine eigenständigen Daten-Eingänge. Die Eingangsdaten werden über den Socket `AL_IN` bereitgestellt.

Daten-Ausgänge

Der Baustein besitzt keine eigenständigen Daten-Ausgänge. Die Ausgangsdaten werden über den Plug `ALI_OUT` bereitgestellt.

Adapter

Typ	Name	Richtung	Datentyp	Beschreibung
adapter::types::unidirectional::AL_IN		Socket (Eingang)	LWORD	LWORD-Adapter als Eingangsschnittstelle. Stellt die zu konvertierenden Daten bereit.
adapter::types::unidirectional::ALI_OUT		Plug (Ausgang)	LINT	LINT-Adapter als Ausgangsschnittstelle. Liefert die konvertierten Daten.

Funktionsweise

1. **Ereignisauslösung:** Der `AL_IN`-Adapter gibt über seinen Ereignisausgang `E1` ein Signal, sobald neue Daten anstehen.
2. **Konvertierung:** Dieses Ereignis triggert den internen FB `Convert` (`F_LWORD_TO_LINT`) an seinem Ereigniseingang `REQ`. Gleichzeitig wird der aktuelle Datenwert von `AL_IN.D1` (LWORD) an den Dateneingang `IN` des Konvertierungsbausteins übergeben.
3. **Ausgabe:** Nach erfolgreicher Umwandlung sendet `Convert` eine Bestätigung über seinen Ereignisausgang `CNF` an den `ALI_OUT`-Adapter (Ereigniseingang `E1`). Parallel dazu wird der konvertierte LINT-Wert am Ausgang `OUT` an den Dateneingang `ALI_OUT.D1` weitergeleitet.
4. **Ergebnis:** Der `ALI_OUT`-Adapter gibt nun die neuen LINT-Daten an den angeschlossenen FB weiter.

Technische Besonderheiten

- **Bit-Reinterpretation, werterhaltend** (Zielbreite ist gleich groß oder größer als der Quell-Bit-String).
- **Typkonvertierung:** Der Baustein wandelt einen LWORD (64-Bit-Bitfolge) in einen LINT (64-Bit-Ganzzahl mit Vorzeichen) um. Die Umwandlung erfolgt bit-identisch, d.h. die Binärdarstellung wird als vorzeichenbehaftete Ganzzahl interpretiert.
- **Komposition:** Es handelt sich um einen Composite-FB, der keine eigene Logik enthält, sondern auf den standardisierten Konvertierungsbaustein `iec61131::conversion::F_LWORD_TO_LINT` zurückgreift.
- **Adapterbasierte Schnittstelle:** Die Ein- und Ausgabe erfolgt ausschließlich über unidirektionale Adapter (`AL` und `ALI`), was eine modulare und typengeprüfte Verbindung in der 4diac-IDE ermöglicht.
- **Paketstruktur:** Der FB ist im Paket `adapter::conversion::unidirectional` abgelegt.

Zustandsübersicht

Da es sich um einen Composite-FB handelt, existiert kein eigener Zustandsautomat. Die Zustandslogik wird vollständig durch den intern genutzten Baustein `F_LWORD_TO_LINT` bestimmt. Dieser arbeitet ereignisgesteuert:

- **IDLE:** Warten auf ein Ereignis am `REQ`-Eingang.
- **PROCESSING:** Umwandlung des LWORD in LINT (dauert einen Schritt).
- **FINISHED:** Ausgabe des konvertierten Werts und Senden des `CNF`-Ereignisses.

Anwendungsszenarien

- **SPS-Datenkonvertierung:** Wenn ein Steuerungsprogramm LINT-Werte (z. B. Zählerstände oder Zeiten) über einen LWORD-basierten Adapter erhält und eine LINT-Schnittstelle benötigt wird.
- **Kommunikations-Adapter:** Einsatz als Zwischenglied in einer Adapterkette, um unterschiedliche Datentypen zwischen verschiedenen Hardware- oder Softwaremodulen zu übersetzen.
- **Protokollanpassung:** Beispielsweise Umwandlung von LWORD-rohdaten (z. B. aus einem Bussystem) in eine LINT-Repräsentation zur weiteren Verarbeitung in der Steuerungslogik.

Vergleich mit ähnlichen Bausteinen

- **AL_TO_AL (LWORD ↔ LWORD):** Reine Durchleitung ohne Konvertierung.
- **ALI_TO_AL (LINT ↔ LWORD):** Führt die umgekehrte Konvertierung durch.
- **F_LWORD_TO_LINT (direkter Konvertierungsbaustein):** Bietet die gleiche Funktionalität, jedoch ohne Adapter-Schnittstelle. `AL_TO_ALI` kapselt diesen Baustein und fügt die Adapter-Anbindung hinzu.

Fazit

Der `AL_TO_ALI`-Funktionsblock ist ein einfacher, aber wichtiger Konverter, der die Lücke zwischen LWORD-basierten und LINT-basierten Adapter-Schnittstellen schließt. Durch seine composite-Struktur bleibt er übersichtlich und kann flexibel in bestehende 4diac-Projekte integriert werden. Die Verwendung von Standard-Konvertierungsbausteinen gewährleistet eine korrekte und effiziente Typumwandlung.

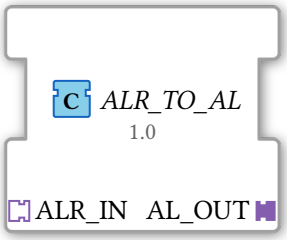
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

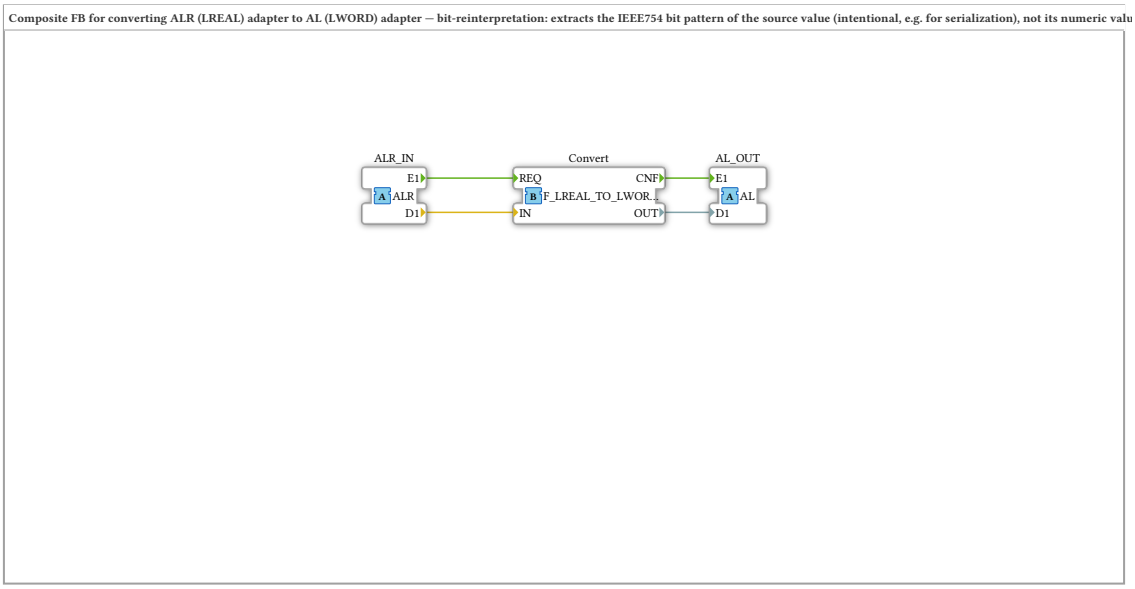
AL_ALR

ALR_TO_AL

LREAL Adapter Input – *ALR*



AL – LWORD Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ALR_TO_AL** ist ein Composite-Baustein (zusammengesetzter Baustein) gemäß IEC 61499-2. Er dient der Umwandlung eines **ALR-Adapters** (LREAL-Datentyp) in einen **AL-Adapter** (LWORD-Datentyp). Implementiert wird die Konvertierung über den internen Baustein `F_LREAL_TO_LWORD` aus der IEC 61131-3-Bibliothek.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock besitzt **keine eigenständigen Ereignis-Eingänge**. Ereignisse werden über den **Socket-Adapter** `ALR_IN` bereitgestellt:

Adapter	Ereignis / Signal	Beschreibung
ALR_IN	E1	Trigger-Ereignis zur Auslösung der Konvertierung

Ereignis-Ausgänge

Der Funktionsblock besitzt **keine eigenständigen Ereignis-Ausgänge**. Ereignisse werden über den **Plug-Adapter** `AL_OUT` ausgegeben:

Adapter	Ereignis / Signal	Beschreibung
AL_OUT	E1	Bestätigungsereignis nach abgeschlossener Konvertierung

Daten-Eingänge

Auch hier gibt es **keine direkten Dateneingänge**. Die Daten werden über den Socket-Adapter `ALR_IN` hereingeführt:

Adapter	Daten / Signal	Datentyp	Beschreibung
ALR_IN	D1	LREAL	Eingangswert im LREAL-Format

Daten-Ausgänge

Keine direkten Datenausgänge. Die Ausgabedaten erfolgen über den Plug-Adapter `AL_OUT`:

Adapter	Daten / Signal	Datentyp	Beschreibung
AL_OUT	D1	LWORD	Konvertierter Ausgangswert im LWORD-Format

Adapter

Der Baustein wird über zwei Adapter mit der Umgebung verbunden:

- **Socket** `ALR_IN`: Vom Typ `adapter::types::unidirectional::ALR` – bereitgestellt von einer vorhergehenden Komponente.
- **Plug** `AL_OUT`: Vom Typ `adapter::types::unidirectional::AL` – wird einer nachfolgenden Komponente angeboten.

Die Adapter übertragen sowohl Ereignisse als auch Daten über ihre jeweiligen Ports (E1, D1).

Funktionsweise

Bei Eintreffen eines Ereignisses **E1** am Socket `ALR_IN` wird der interne Baustein `F_LREAL_TO_LWORD` (Typ `iec61131::conversion::F_LREAL_TO_LWORD`) über seinen Ereigniseingang `REQ` aktiviert. Gleichzeitig wird der Datenwert `D1` des Adapter-Sockets an den Dateneingang `IN` des Konvertierungsbausteins übergeben. Nach erfolgreicher Umwandlung gibt der interne Baustein ein Bestätigungsereignis an seinem Ausgang `CNF` aus. Dieses Ereignis wird an den Ereigniseingang `E1` des Plug-Adapters `AL_OUT` weitergeleitet. Der konvertierte Wert (LWORD) wird über die Datenverbindung `Convert.OUT` ⟶ `AL_OUT.D1` an den Adapter-Plug übergeben.

Die Konvertierung selbst entspricht der IEC 61131-3-Funktion `LREAL_TO_LWORD`, die den bitweisen Aufbau eines LREAL-Wertes in einen LWORD-Wert abbildet (z. B. zur Übertragung von Gleitkommazahlen als Rohdaten).

Technische Besonderheiten

- **Bit-Reinterpretation**: extrahiert das IEEE754-Bitmuster des Quellwerts (beabsichtigt, z. B. zur Serialisierung), nicht dessen Zahlenwert.
- **Composite-Baustein**: Die Funktionalität wird durch eine interne Netzwerkschaltung realisiert, die den Baustein `F_LREAL_TO_LWORD` verwendet.
- **Paketierung**: Der Baustein ist im Paket `adapter::conversion::unidirectional` abgelegt, was auf seine Verwendung in unidirektionalen Adapterketten hinweist.
- **Import**: `iec61131::conversion::F_LREAL_TO_LWORD` muss im Projekt verfügbar sein.
- **Copyright**: Der Baustein wurde für die HR Agrartechnik GmbH entwickelt und steht unter der Eclipse Public License 2.0 (EPL-2.0).

Zustandsübersicht

Da es sich um einen **Composite-FB** ohne eigene Zustandsmaschine handelt, ergibt sich das Verhalten aus dem des internen Funktionsblocks `F_LREAL_TO_LWORD`. Dieser führt eine reine, stateless-Konvertierung

durch: Ein Ereignis löst die Berechnung aus, das Ergebnis wird sofort bereitgestellt. Der Baustein hat keinen internen Zustand, der über mehrere Aufrufe hinweg erhalten bleibt.

Anwendungsszenarien

- **Datenformat-Anpassung**: Wenn ein Systemteil Daten im ALR-Format (LREAL über Adapter) liefert, aber eine nachfolgende Komponente nur AL-Schnittstellen (LWORD über Adapter) versteht.
- **Kommunikation zwischen unterschiedlichen Steuerungsebenen**: Z. B. Übergabe von physikalischen Messwerten aus einer SPS (LREAL) an ein übergeordnetes Leitsystem, das LWORD als Rohdatenformat erwartet.
- **Test- und Simulationsaufbauten**: Erzeugung definierter LWORD-Werte aus LREAL-Quellen, ohne die eigentliche Adapterlogik zu ändern.

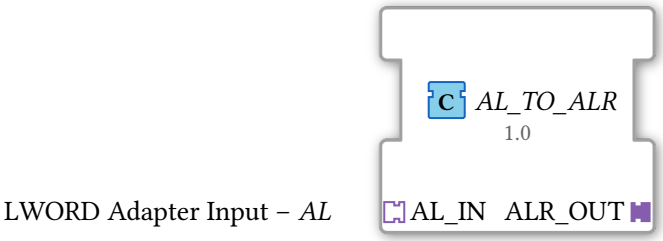
Vergleich mit ähnlichen Bausteinen

- **AL_TO_ALR**: Führt die entgegengesetzte Konvertierung (LWORD ⟶ LREAL) durch.
- **Direkte Konvertierungsfunktionen**: Die IEC 61131-3 bietet `LREAL_TO_LWORD` und `LWORD_TO_LREAL` als elementare Bausteine – `ALR_TO_AL` kapselt diese in einen Adapter-konformen Composite-Baustein, sodass er direkt in Adapterketten eingesetzt werden kann.

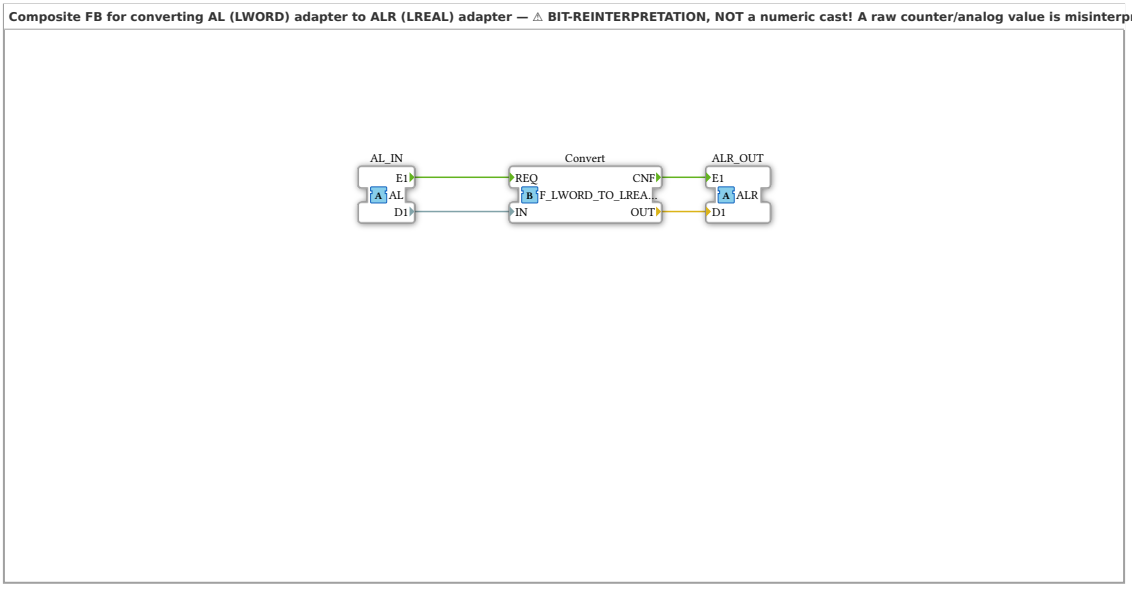
Fazit

Der Funktionsblock **ALR_TO_AL** ist ein spezialisierter Konvertierungsbaustein für die Umwandlung eines LREAL-Adapters in einen LWORD-Adapter. Seine Composite-Struktur ermöglicht eine saubere Integration in bestehende Adapter-basierte Architekturen, ohne dass der Anwender die zugrundeliegende Konvertierungsfunktion manuell verschalten muss. Die eindeutige Ereignissteuerung und die Verwendung standardisierter Datentypen machen ihn zu einer robusten Lösung für die Anpassung von Datenformaten in IEC 61499-Steuerungssystemen.

AL_TO_ALR



LWORD Adapter Input – *AL* *ALR* – LREAL Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AL_TO_ALR** ist ein zusammengesetzter Baustein (Composite FB), der einen unidirektionalen AL-Adapter (LWORD) auf einen unidirektionalen ALR-Adapter (LREAL) abbildet. Er nutzt intern den Baustein `F_LWORD_TO_LREAL`.

> ⚠️ **Achtung — keine numerische Wertumwandlung:** `LWORD` ist ein Bit-String-Typ (kein Integer mit Vorzeichen-Semantik), und `F_LWORD_TO_LREAL` reinterpretiert im FORTE-Kern lediglich das 64-Bit-Muster als IEEE754-`LREAL` (Bit-Kopie) — dieselbe Falle wie bei `AD_TO_AR` (DWORD→REAL), nur mit den 64-Bit-Pendants. Ein roher Zähler- oder Analogwert wird dadurch **nicht** zum entsprechenden LREAL-Zahlenwert, sondern zu einer bedeutungslosen Zahl. Für eine echte numerische Umwandlung: zweistufig über `AL_TO_AULI` (Bit-Reinterpretation LWORD→ULINT, hier gültig, da beide dieselbe 64-Bit-Darstellung eines vorzeichenlosen Integers teilen) gefolgt von `AULI_TO_ALR` (echter numerischer Cast) — dasselbe Muster wie `AD_TO_AR_NUM` für den 32-Bit-Fall.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine direkten Ereignis-Eingänge. Das für die Auslösung der Konvertierung erforderliche Ereignis wird über den Socket **AL_IN** (Typ `AL`) bereitgestellt. Dieses Ereignis (E1) wird intern mit dem `REQ`-Eingang des Konvertierungsbausteins verbunden.

Ereignis-Ausgänge

Der FB besitzt keine direkten Ereignis-Ausgänge. Das Abschlussereignis der Konvertierung wird über den Plug **ALR_OUT** (Typ `ALR`) ausgegeben. Das Ereignis (E1) des internen Bausteins (`CNF`) wird an den Ausgangsadapter weitergeleitet.

Daten-Eingänge

Der FB besitzt keine direkten Daten-Eingänge. Der zu konvertierende LWORD-Wert wird über den Socket **AL_IN** (Daten-Signal D1) bereitgestellt und an den `IN`-Eingang des Konvertierungsbausteins übergeben.

Daten-Ausgänge

Der FB besitzt keine direkten Daten-Ausgänge. Der konvertierte LREAL-Wert wird über den Plug **ALR_OUT** (Daten-Signal D1) ausgegeben; er stammt vom `OUT`-Ausgang des internen Bausteins.

Adapter

Adapter	Richtung	Typ	Beschreibung
AL_IN	Socket (Eingang)	`adapter::types::unidirectional::AL`	LWORD-Adapter, der das Eingangssignal (LWORD) und ein Auslöseereignis bereitstellt.
ALR_OUT	Plug (Ausgang)	`adapter::types::unidirectional::ALR`	LREAL-Adapter, der das konvertierte Signal (LREAL) und ein Bestätigungsereignis ausgibt.

Funktionsweise

`F_LWORD_TO_LREAL` reinterpretiert das 64-Bit-Muster des LWORD als IEEE754-`LREAL` (Bit-Kopie), es findet **keine** Zahlenwert-Konvertierung statt. Der Baustein arbeitet in einem einfachen ereignisgesteuerten Ablauf:

1. Ein eingehendes Ereignis am Socket **AL_IN.E1** wird an den `REQ`-Eingang des internen Funktionsblocks `F_LWORD_TO_LREAL` weitergeleitet.
2. Gleichzeitig wird der Datenwert von **AL_IN.D1** (LWORD) an den `IN`-Eingang des Konverters übergeben.
3. Der interne FB führt die Umwandlung `LWORD → LREAL` durch.
4. Nach Abschluss der Konvertierung wird das Ereignis `CNF` an den Plug **ALR_OUT.E1** gesendet.
5. Der konvertierte LREAL-Wert wird auf **ALR_OUT.D1** ausgegeben.

Die gesamte Verarbeitung erfolgt innerhalb eines Taktzyklus (keine Blockierung).

Technische Besonderheiten

- **Adapter-basierte Schnittstelle**: Der FB nutzt ausschließlich unidirektionale Adapter (`AL` und `ALR`). Dies ermöglicht eine lose Kopplung zwischen Sender und Empfänger sowie eine einfache Wiederverwendung in verschiedenen Systemarchitekturen.
- ⚠️ **Bit-Reinterpretation statt Zahlenwert-Umwandlung**: Siehe Warnhinweis in der Einleitung — betrifft alle Bit-String-Quelltypen beim Umwandeln in `AR`/REAL oder `ALR`/LREAL; der einzige weitere Fall in dieser Bibliothek ist `AD_TO_AR` (DWORD⟶REAL).
- **Durchgereichte Ereignissteuerung**: Der Baustein führt keine interne Zustandslogik aus; er leitet Ereignisse und Daten transparent weiter. Die Laufzeit des Aufrufs entspricht der Ausführungszeit des internen Konverters.

Zustandsübersicht

Der zusammengesetzte Baustein selbst besitzt **keine eigenen Zustände**. Die Verarbeitung wird vollständig durch den intern verwendeten Funktionsblock `F_LWORD_TO_LREAL` gesteuert, der folgendes minimales Zustandsverhalten aufweist:

- **IDLE**: Warten auf ein `REQ`-Ereignis.
- **CONVERT**: Ausführen der Konvertierung (unmittelbar nach `REQ`).
- **SEND**: Ausgeben des `CNF`-Ereignisses und des konvertierten Werts.

Da der innere FB in einem einzigen Ausführungsschritt arbeitet, sind die Zustände nicht von außen beobachtbar.

Anwendungsszenarien

- **Bitmuster-Weitergabe**: `AL_IN` liefert bereits ein Bitmuster, das als LREAL interpretiert werden soll (z. B. serialisierte Double-Rohdaten aus einem Feldbus/Protokoll).
- **NICHT geeignet** für rohe Zähler-, Analog- oder sonstige Ganzzahlwerte, die als derselbe Zahlenwert in LREAL vorliegen sollen — dafür `AL_TO_AULI` + `AULI_TO_ALR` verwenden.

Vergleich mit ähnlichen Bausteinen

- **`AL_TO_AULI` + `AULI_TO_ALR`** (numerisch, LWORD⟶ULINT⟶LREAL): der sichere Ersatz, wenn tatsächlich ein Zahlenwert gemeint ist.
- **`ALI_TO_ALR`/`AULI_TO_ALR`** (LINT/ULINT⟶LREAL): bereits numerisch korrekt, da ihre Quelltypen `ANY_INT` (nicht `ANY_BIT`) sind — kein Trap.
- **`AD_TO_AR`** (DWORD⟶REAL): dieselbe Bit-Reinterpretations-Falle, nur mit den 32-Bit-Pendants.

Fazit

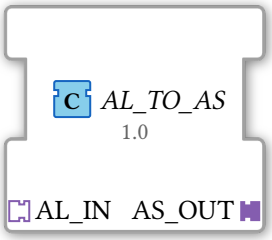
`AL_TO_ALR` bietet eine kompakte Lösung, um ein LWORD-Bitmuster über Adapter-Schnittstellen als LREAL bereitzustellen — **aber keine numerische Wertumwandlung**. Für rohe Zähler- oder Analogwerte, die als derselbe Zahlenwert in LREAL vorliegen sollen, `AL_TO_AULI` + `AULI_TO_ALR` verwenden.

📖 **Hintergrund**

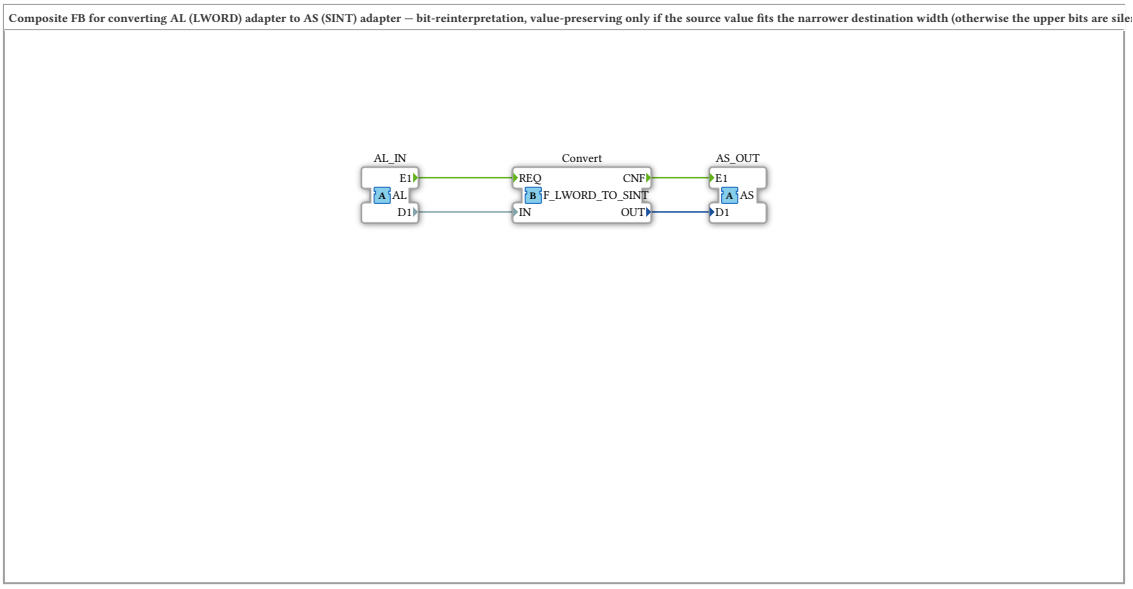
- Numerisch vs. bitweise: Die Konvertierungs-Falle in FORTE

AL_AS

AL_TO_AS



LWORD Adapter Input – *AL* — *AS* – SINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AL_TO_AS** ist ein Composite-Baustein, der einen unidirektionalen AL-Adapter (Daten vom Typ LWORD) in einen unidirektionalen AS-Adapter (Daten vom Typ SINT) umwandelt. Er ermöglicht die nahtlose Integration von Komponenten, die mit unterschiedlichen Datentypen arbeiten, indem er die Signale konvertiert und über die jeweiligen Adapter-Schnittstellen weiterreicht.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine

Ereignis-Ausgänge

- Keine

Daten-Eingänge

- Keine

Daten-Ausgänge

- Keine

Adapter

Typ	Name	Richtung	Beschreibung
Socket	AL_IN	Input	Unidirektionaler AL-Adapter, der LWORD-Daten bereitstellt.
Plug	AS_OUT	Output	Unidirektionaler AS-Adapter, der die konvertierten SINT-Daten ausgibt.

Funktionsweise

Der FB empfängt über den Socket `AL_IN` Ereignisse und LWORD-Daten. Intern wird ein Baustein vom Typ `F_LWORD_TO_SINT` (aus der IEC-61131-Bibliothek) verwendet, um die Eingangsdaten (LWORD) in den Ausgangsdatentyp SINT zu konvertieren. Die Ereignissteuerung erfolgt wie folgt:

- Ein eingehendes Ereignis am Socket `AL_IN.E1` löst den Eingang `REQ` des internen Konvertierungsbausteins aus.
- Nach erfolgreicher Konvertierung signalisiert der Ausgang `CNF` des Konverters ein Ereignis, das an den Plug `AS_OUT.E1` weitergegeben wird.

- Gleichzeitig wird der konvertierte Datenwert vom Ausgang `OUT` des Konverters an den Datenausgang `AS_OUT.D1` übergeben.

Der gesamte Ablauf ist ereignisgesteuert: Nur bei einem Ereignis am `AL_IN` wird die Konvertierung durchgeführt und ein Ereignis am `AS_OUT` ausgelöst.

Technische Besonderheiten

- **Bit-Reinterpretation**, werterhaltend nur, wenn der Quellwert in die schmalere Zielbreite passt (sonst werden die oberen Bits stillschweigend verworfen).
- Der FB ist als Composite implementiert und nutzt die vordefinierte Konvertierungsfunktion `F_LWORD_TO_SINT`, was eine hohe Wiederverwendbarkeit und Standardkonformität gewährleistet.
- Die Konvertierung erfolgt **unidirektional** nur von LWORD zu SINT; eine Rückrichtung ist nicht vorgesehen.
- Die Ereignisweiterleitung ist direkt: Ein Ereignis am Eingang führt zu genau einem Ereignis am Ausgang.
- Die Datenbreite reduziert sich von 64 Bit (LWORD) auf 8 Bit (SINT), daher sind nur Werte im Bereich $[-128, 127]$ darstellbar. Bei Überschreitung dieses Bereichs verhält sich der interne Konverter entsprechend der IEC-61131-Definition (z. B. Überlaufkennzeichnung oder Modulo-Verhalten).

Zustandsübersicht

Der FB besitzt keinen eigenen Zustandsautomaten. Sein Verhalten ist rein kombinatorisch mit ereignisgesteuerter Triggerung. Ein interner Zustand existiert nicht; die Konvertierung erfolgt immer auf Basis der aktuellen Eingangsdaten bei einem Ereignis.

Anwendungsszenarien

- **Systemintegration**: Wenn ein Softwaresystem oder ein Gerät LWORD-Schnittstellen (Adapter AL) verwendet, aber ein anderer Teil des Systems nur AS-Adapter (SINT) unterstützt, kann dieser FB als Brücke dienen.
- **Datenreduktion**: Gezieltes Umwandeln von 64-Bit-Daten in einen 8-Bit-Wert, z. B. für die Übergabe an einfache Aktoren oder Anzeigen, die nur SINT verarbeiten.
- **Protokollanpassung**: In heterogenen Automatisierungsnetzwerken, in denen unterschiedliche Adaptertypen (unidirektional) zum Einsatz kommen, vereinfacht dieser FB die Konvertierung auf einer einzigen logischen Ebene.

Vergleich mit ähnlichen Bausteinen

Baustein	Eingangstyp	Ausgangstyp	Besonderheit
AL_TO_AS	LWORD (Adapter)	SINT (Adapter)	Composite; verwendet `F_LWORD_TO_SINT`
`F_LWORD_TO_SINT`	LWORD (Daten)	SINT (Daten)	Reiner Datenkonverter, keine Adaptereinbindung
`F_LWORD_TO_DINT`	LWORD	DINT	Konvertierung zu 32-Bit-Integer
`F_SINT_TO_LWORD`	SINT	LWORD	Rückrichtung (sofern vorhanden)

Der **AL_TO_AS** hebt sich durch seine Adapter-zu-Adapter-Schnittstelle ab, die eine direkte Verbindung zwischen zwei unterschiedlichen Adaptertypen ohne zusätzliche manuelle Verkabelung von Einzelereignissen und Daten ermöglicht.

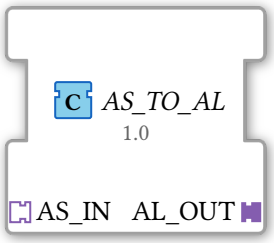
Fazit

Der **AL_TO_AS**-Funktionsblock ist ein praktischer, standardkonformer Konverterbaustein für unidirektionale Adapter. Er kapselt die notwendige Typumwandlung von LWORD nach SINT und vereinfacht die Integration in bestehende 4diac-Projekte. Aufgrund seiner Composite-Struktur und der Verwendung einer etablierten Konvertierungsfunktion ist er zuverlässig, leicht verständlich und erweiterbar. Er eignet sich besonders für Szenarien, in denen Datenbreiten reduziert oder Schnittstellen angepasst werden müssen.

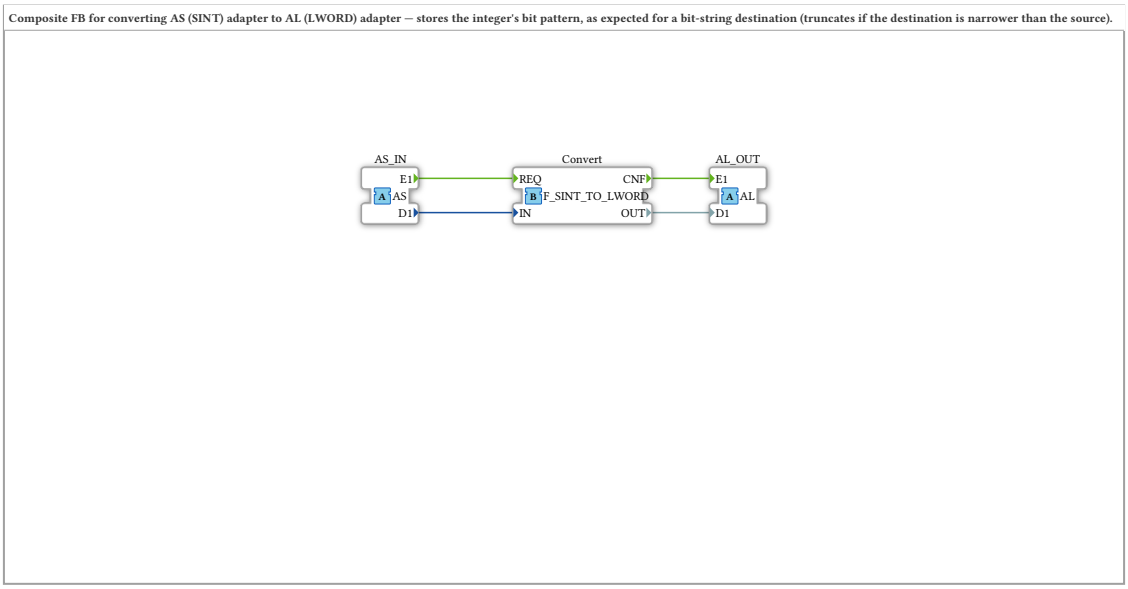
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AS_TO_AL



SINT Adapter Input – *AS* *AL* – LWORD Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **AS_TO_AL** ist ein Composite-Baustein, der einen SINT-Adapter (AS) in einen LWORD-Adapter (AL) umwandelt. Er dient als Brücke zwischen unterschiedlichen Adaptertypen in einer unidirektionalen Kommunikationsstrecke. Der Baustein kapselt die Konvertierungslogik und macht sie für den Anwender als zusammengesetzte Einheit nutzbar.

Schnittstellenstruktur

Der Baustein besitzt **keine direkten Ereignis- oder Datenschnittstellen** auf oberster Ebene. Die gesamte Ein- und Ausgabe erfolgt über Adapter.

Ereignis-Eingänge

- Keine (Ereignisse werden über den Socket-Adapter AS_IN hereingereicht)

Ereignis-Ausgänge

- Keine (Ereignisse werden über den Plug-Adapter AL_OUT ausgegeben)

Daten-Eingänge

- Keine (Daten werden über den Socket-Adapter AS_IN bereitgestellt)

Daten-Ausgänge

- Keine (Daten werden über den Plug-Adapter AL_OUT bereitgestellt)

Adapter

Name	Typ	Richtung	Beschreibung
AS_IN	adapter::types::unidirectional::AS	Socket	SINT-Adapter (Eingang)

AL_OUT	adapter::types::unidirectional::AL	Plug	LWORD-Adapter (Ausgang)

Funktionsweise

Der Baustein arbeitet intern mit dem IEC-61131-Konvertierungsbaustein **F_SINT_TO_LWORD**. Sobald ein Ereignis am Ereignisausgang **E1** des Socket-Adapters **AS_IN** eintrifft, wird dieses an den Eingang **REQ** des Konvertierungsbausteins weitergeleitet. Dieser wandelt den SINT-Datenwert von **AS_IN.D1** in einen LWORD-Wert um und gibt ihn an **AL_OUT.D1** aus. Nach erfolgreicher Umwandlung wird ein Bestätigungsereignis an den Ereigniseingang **E1** des Plug-Adapters **AL_OUT** gesendet.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Standardkonformität**: Entspricht IEC 61499-2.
- **Urheberrechte**: Der Baustein unterliegt der Eclipse Public License 2.0 (Copyright 2026 HR Agrartechnik GmbH).
- **Version**: 1.0, erstellt von Franz Höpfinger am 17.02.2026.
- **Paketstruktur**: Im Paket `adapter::conversion::unidirectional` organisiert.
- **Abhängigkeiten**: Nutzt den Baustein `iec61131::conversion::F_SINT_TO_LWORD`.

Zustandsübersicht

Da es sich um einen Composite-Baustein handelt, besitzt er keinen eigenen Zustandsautomaten. Die Zustandslogik wird durch den integrierten Baustein **F_SINT_TO_LWORD** abgebildet, der intern eine einfache Umwandlung ohne Verzögerung oder Fehlerzustände vornimmt. Die Ausführung erfolgt streng ereignisgesteuert.

Anwendungsszenarien

- **Adapterkonvertierung**: Verbindung von Komponenten, die einen SINT-Adapter bereitstellen, mit Komponenten, die einen LWORD-Adapter erwarten.
- **Datenanpassung**: Einsatz in heterogenen Automatisierungssystemen, in denen unterschiedliche Datentypen auf Adapterebene harmonisiert werden müssen.
- **Modulare Wiederverwendung**: Der Baustein kann in Bibliotheken für unidirektionale Konvertierungen eingebunden werden.

Vergleich mit ähnlichen Bausteinen

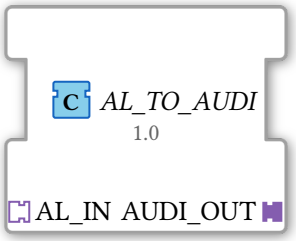
Ähnliche Adapter-Konvertierungsbausteine existieren für andere Datentypen (z. B. BOOL, INT, DWORD). Der **AS_TO_AL** ist speziell auf die Umwandlung von **SINT** (Signed Short Integer, 8 Bit) nach **LWORD** (Long Word, 64 Bit) ausgelegt. Im Gegensatz zu generischen Funktionsbausteinen wird die Konvertierung hier bereits auf Adapterebene bereitgestellt, was eine einfachere Verdrahtung im Netzwerk ermöglicht.

Fazit

Der Composite-Baustein **AS_TO_AL** bietet eine kompakte und standardkonforme Lösung zur Umwandlung eines SINT-Adapters in einen LWORD-Adapters. Durch die Kapselung der Konvertierungslogik wird die Übersichtlichkeit des Anwendungsnetzwerks erhöht und die Wiederverwendbarkeit gefördert. Der Baustein eignet sich besonders für modulare Automatisierungsprojekte, bei denen unterschiedliche Datentypen auf Adapterebene zusammengeführt werden müssen.

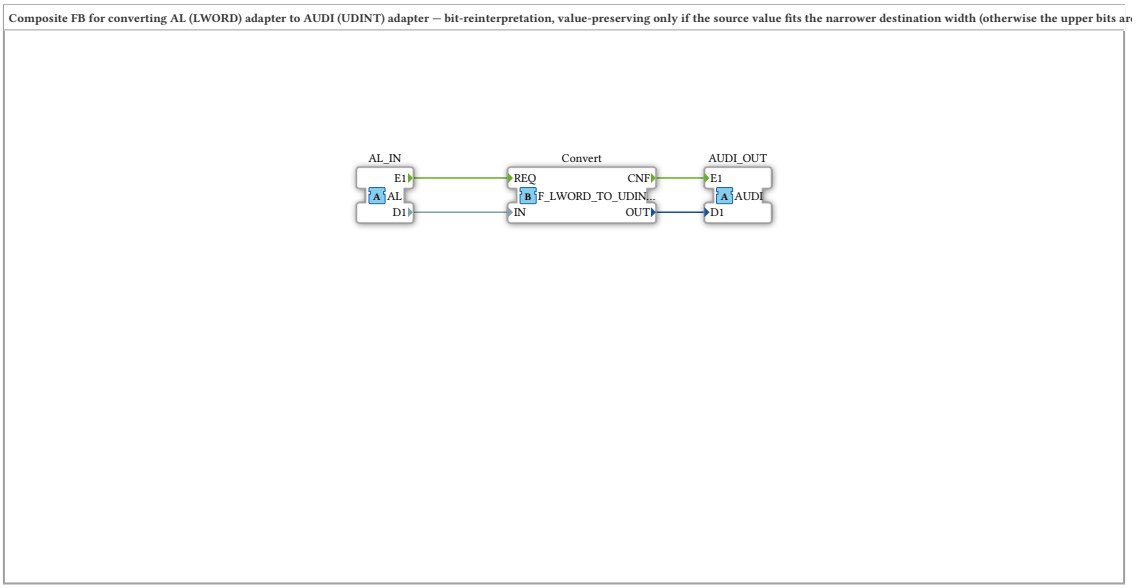
AL_AUDI

AL_TO_AUDI



LWORD Adapter Input – *AL*

AUDI – UDINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AL_TO_AUDI** ist ein Composite-Baustein, der einen LWORD-Adaptereingang (Typ `AL`) in einen UDINT-Adapterausgang (Typ `AUDI`) umwandelt. Er kapselt die IEC-61131-3-Konvertierungsfunktion `F_LWORD_TO_UDINT` und stellt eine saubere, adapterbasierte Schnittstelle für unidirektionale Datenflüsse bereit.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine direkten Ereignis-Eingänge. Die Ereignissteuerung erfolgt vollständig über den integrierten Socket-Adapter `AL_IN`.

Ereignis-Ausgänge

Der FB besitzt keine direkten Ereignis-Ausgänge. Ereignisse werden über den integrierten Plug-Adapter `AUDI_OUT` an die nachfolgende Logik weitergegeben.

Daten-Eingänge

Der FB besitzt keine direkten Daten-Eingänge. Die zu konvertierenden LWORD-Daten werden über den Socket-Adapter `AL_IN` empfangen.

Daten-Ausgänge

Der FB besitzt keine direkten Daten-Ausgänge. Die konvertierten UDINT-Daten werden über den Plug-Adapter `AUDI_OUT` ausgegeben.

Adapter

Adaptertyp	Name	Beschreibung
Socket	AL_IN	LWORD-Adaptereingang (Typ: adapter::types::unidirectional::AL)
Plug	AUDI_OUT	UDINT-Adapterausgang (Typ: adapter::types::unidirectional::AUDI)

Funktionsweise

1. Ein eingehendes Ereignis am Socket AL_IN (über dessen Ereignisausgang E1) triggert die Konvertierungsinstanz Convert (Typ F_LWORD_TO_UDINT).
2. Gleichzeitig wird der LWORD-Datenwert von AL_IN.D1 an den Eingang IN von Convert übergeben.
3. Die Konvertierungsfunktion wandelt den LWORD-Wert in einen UDINT-Wert um.
4. Nach erfolgreicher Konvertierung signalisiert Convert dies über seinen Ereignisausgang CNF, wodurch der Plug AUDI_OUT ein Ereignis an seinem Ereigniseingang E1 erhält.
5. Der umgewandelte UDINT-Wert wird vom Ausgang OUT von Convert an den Dateneingang D1 von AUDI_OUT weitergeleitet und steht damit am Ausgangsadapter zur Verfügung.

Technische Besonderheiten

- **Bit-Reinterpretation**, werterhaltend nur, wenn der Quellwert in die schmalere Zielbreite passt (sonst werden die oberen Bits stillschweigend verworfen).
- **Composite-Architektur**: Der Baustein besteht ausschließlich aus der internen Verwendung des Funktionsblocks F_LWORD_TO_UDINT und den Adaptern. Es ist keine eigene Zustandslogik oder zusätzliche Steuerung vorhanden.
- **Unidirektionale Adapter**: Sowohl AL_IN als auch AUDI_OUT sind unidirektionale Adapter (Richtung von Eingang zu Ausgang).
- **IEC-61131-3-Konformität**: Die Konvertierung nutzt eine standardisierte IEC-61131-3-Funktion, was die Wiederverwendbarkeit und Austauschbarkeit erhöht.

Zustandsübersicht

Der FB besitzt keine eigenen Zustände. Das Verhalten ist rein kombinatorisch und wird vollständig durch die interne Konvertierungsfunktion bestimmt – die Konvertierung erfolgt pro Ereignis ohne interne Speicherung.

Anwendungsszenarien

- **Systemintegration**: Verbindung von Komponenten, die unterschiedliche Datentypen für Adressen, Zähler oder Konfigurationswerte verwenden (z. B. LWORD für breite Adressen, UDINT für eingebettete Systeme).
- **Adapter-Architekturen**: Einsatz in modularen Automatisierungslösungen, bei denen über Adapter standardisierte Schnittstellen bereitgestellt werden sollen.
- **Datenaufbereitung**: Konvertierung von LWORD-Signalen aus Sensoren oder Bussystemen in UDINT für die Weiterverarbeitung in Steuerungen.

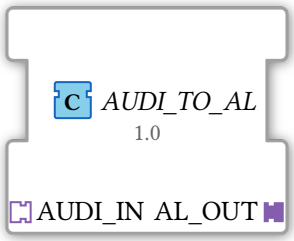
Vergleich mit ähnlichen Bausteinen

- **Einfache Typkonverter** (z. B. F_LWORD_TO_UDINT): Diese arbeiten direkt mit Ein-/Ausgangsvariablen, während AL_TO_AUDI die Konvertierung in eine Adapter-Schnittstelle kapselt und so eine höhere Abstraktionsebene bietet.
- **Andere Adapter-Konverter** (z. B. UDINT_TO_DINT): Ähnliche Konzepte, aber für andere Datentypen und Richtungen. AL_TO_AUDI ist spezifisch für die unidirektionale Wandlung von LWORD nach UDINT.

Fazit

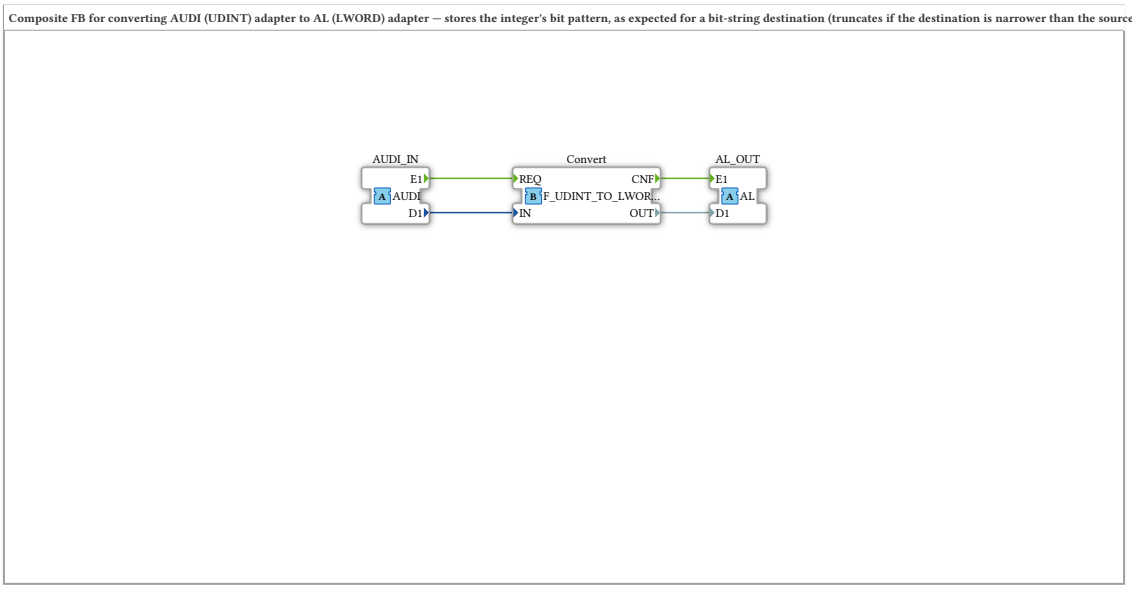
Der Funktionsblock AL_TO_AUDI ist ein kompakter und wiederverwendbarer Baustein zur Konvertierung von LWORD- auf UDINT-Daten über standardisierte Adapter. Er vereinfacht die Integration von Komponenten mit unterschiedlichen Datentypen und fördert eine modulare, adapterbasierte Architektur in der Automatisierungstechnik.

AUDI_TO_AL



UDINT Adapter Input – *AUDI*

AL – LWORD Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUDI_TO_AL** ist ein zusammengesetzter (Composite) Baustein, der die Konvertierung eines **AUDI**-Adapters (Datenformat UDINT) in einen **AL**-Adapter (Datenformat LWORD) realisiert. Er dient als Schnittstellenwandler zwischen unterschiedlichen Adaptertypen in Automatisierungssystemen, die auf dem Standard IEC 61499 basieren.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine eigenen Ereignis-Eingänge. Die ereignisgesteuerte Verarbeitung erfolgt über den **AUDI_IN**-Socket:

- **E1** : Ereignisseingang des AUDI-Adapters – löst die Konvertierung aus.

Ereignis-Ausgänge

Der Baustein besitzt keine eigenen Ereignis-Ausgänge. Die Ergebnisweitergabe erfolgt über den **AL_OUT**-Plug:

- **E1** : Ereignisausgang des AL-Adapters – signalisiert abgeschlossene Konvertierung.

Daten-Eingänge

Der Baustein besitzt keine eigenen Daten-Eingänge. Die zu konvertierenden Daten werden über den **AUDI_IN**-Socket bereitgestellt:

- **D1** : Dateneingang des AUDI-Adapters (Typ: UDINT) – der zu konvertierende Wert.

Daten-Ausgänge

Der Baustein besitzt keine eigenen Daten-Ausgänge. Das konvertierte Ergebnis wird über den **AL_OUT**-Plug bereitgestellt:

- **D1** : Datenausgang des AL-Adapters (Typ: LWORD) – der konvertierte Wert.

Adapter

Name	Richtung	Typ	Beschreibung
AUDI_IN	Socket	adapter::types::unidirectional::AUDI	UDINT-Adapter als Eingangsschnittstelle
AL_OUT	Plug	adapter::types::unidirectional::AL	LWORD-Adapter als Ausgangsschnittstelle

Funktionsweise

Der Baustein bildet eine reine Kapselung der IEC-61131-Funktion `F_UDINT_TO_LWORD`. Bei einem Ereignis am **AUDI_IN.E1** wird der interne Funktionsbaustein **Convert** (Typ `F_UDINT_TO_LWORD`) über dessen Ereigniseingang **REQ** aktiviert. Gleichzeitig wird der an **AUDI_IN.D1** anliegende UDINT-Wert an den Dateneingang **IN** von **Convert** übergeben. Nach Abschluss der Konvertierung signalisiert **Convert** über seinen Ereignisausgang **CNF** die Fertigstellung. Dieses Ereignis wird direkt an **AL_OUT.E1** weitergeleitet, und der konvertierte LWORD-Wert wird von **Convert.OUT** auf **AL_OUT.D1** ausgegeben.

Der gesamte Ablauf ist ereignisgesteuert und ohne zusätzliche Laufzeitlogik innerhalb des Composite-Bausteins.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- Der Baustein ist als **Composite-Funktionsblock** realisiert und verwendet ausschließlich die standardisierte IEC-61131-Konvertierungsfunktion `F_UDINT_TO_LWORD`.
- Die Konvertierung erfolgt direkt: **UDINT** (32-Bit unsigned integer) wird auf **LWORD** (64-Bit logical word) abgebildet. Hinweis: Bei der Umwandlung eines 32-Bit-Werts in ein 64-Bit-Wort werden die oberen 32 Bits auf Null gesetzt.
- Der Baustein ist unter der **Eclipse Public License 2.0** lizenziert.
- Die Schnittstellen (Adapter) sind unidirektional ausgelegt: Der Datenfluss erfolgt nur vom Socket zum Plug.

Zustandsübersicht

Da es sich um einen zusammengesetzten Baustein ohne eigene Zustandsmaschine handelt, besitzt **AUDI_TO_AL** keinen internen Zustandsautomaten. Das Verhalten wird vollständig durch den verschalteten Funktionbaustein `F_UDINT_TO_LWORD` bestimmt, der einen einfachen REQ/CNF-Handshake implementiert.

Anwendungsszenarien

- **Adapter-Konvertierung** in IEC-61499-Systemen, in denen ein existierender AUDI-Adapter (UDINT-basiert) mit einem AL-Adapter (LWORD-basiert) verbunden werden muss – z. B. zur Anbindung von Sensordaten an eine 64-Bit-Datenverarbeitungskette.
- **Schnittstellenanpassung** beim Austausch von Hardwarekomponenten oder bei der Migration von 32-Bit- auf 64-Bit-Automatisierungslösungen.
- **Prototypische Umsetzung** in der Landtechnik oder anderen Bereichen, in denen die Datenformate UDINT und LWORD gemischt auftreten.

Vergleich mit ähnlichen Bausteinen

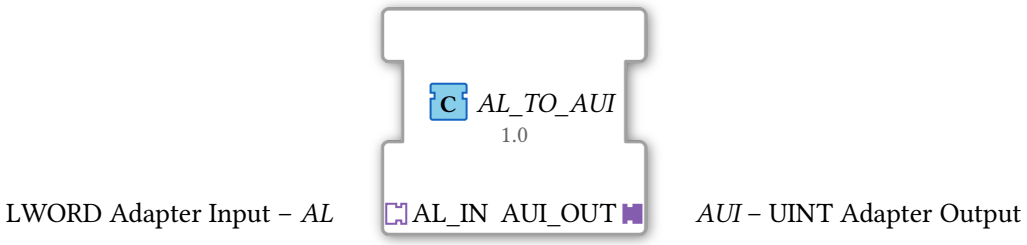
Im Gegensatz zu generischen Konvertern wie `UDINT_TO_LWORD` (einem atomaren Baustein) bietet **AUDI_TO_AL** eine vorgefertigte Adapter-Kopplung, die den Entwickler von der manuellen Verschaltung von Ereignis- und Datenleitungen befreit. Ähnliche Composite-Bausteine existieren z. B. für `UDINT_TO_DWORD` oder `LREAL_TO_LWORD`, jedoch spezifisch für andere Zieladapter.

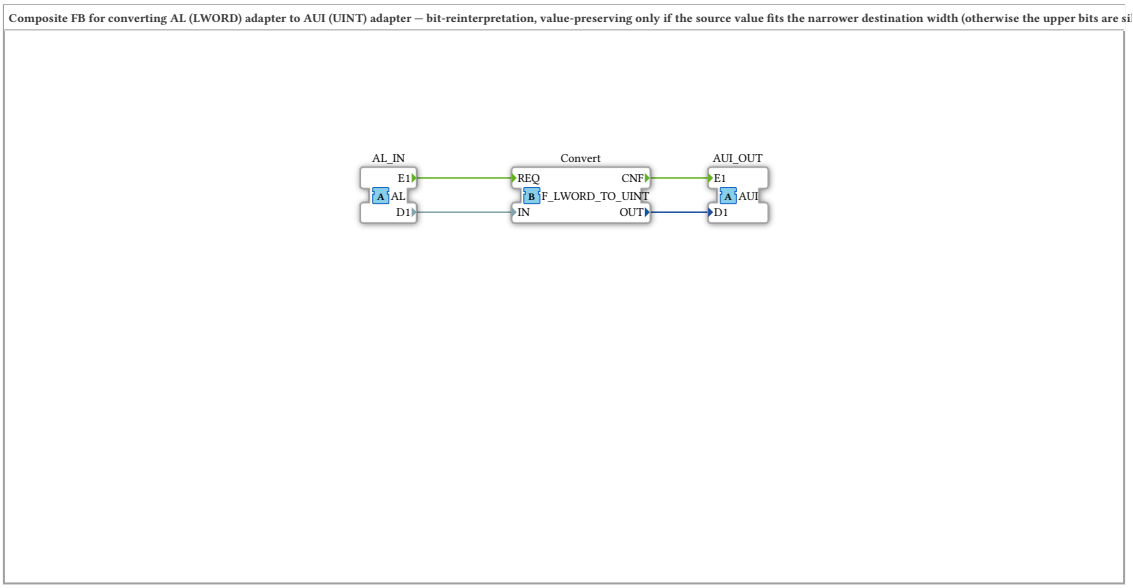
Fazit

Der Funktionsblock **AUDI_TO_AL** ist ein kompakter, wiederverwendbarer Baustein zur sauberen Trennung von Adaptertypen. Durch die Kapselung der Konvertierung in einem Composite vereinfacht er die Wartbarkeit und Lesbarkeit von Anwendungen, die auf unterschiedlichen Datenformatadaptern basieren.

AL_AUI

AL_TO_AUI





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AL_TO_AUI** ist ein Composite-Baustein, der einen LWORD-Adapter (AL) in einen UINT-Adapter (AUI) umwandelt. Er ermöglicht die nahtlose Integration von LWORD-basierten Schnittstellen in Systeme, die UINT-Daten erwarten, indem er die Konvertierung innerhalb einer standardisierten Adapterstruktur kapselt.

Schnittstellenstruktur

Ereignis-Eingänge

Keine eigenständigen Ereignis-Eingänge. Der Socket **AL_IN** liefert über seinen Ereignisausgang `E1` das auslösende Signal für die Konvertierung.

Ereignis-Ausgänge

Keine eigenständigen Ereignis-Ausgänge. Der Plug **AUI_OUT** stellt über seinen Ereigniseingang `E1` die Bestätigung der abgeschlossenen Konvertierung bereit.

Daten-Eingänge

Keine eigenständigen Daten-Eingänge. Der Socket **AL_IN** liefert über seinen Datenausgang `D1` den als `LWORD` zu konvertierenden Wert.

Daten-Ausgänge

Keine eigenständigen Daten-Ausgänge. Der Plug **AUI_OUT** stellt über seinen Dateneingang `D1` den konvertierten `UINT`-Wert zur Verfügung.

Adapter

Typ	Richtung	Name	Beschreibung
Socket	Eingang	**AL_IN**	Adapter vom Typ `adapter::types::unidirectional::AL` – liefert ein LWORD-Datum und ein Ereignis zur Auslösung der Konvertierung.
Plug	Ausgang	**AUI_OUT**	Adapter vom Typ `adapter::types::unidirectional::AUI` – empfängt das konvertierte UINT-Datum und gibt das Bestätigungsereignis weiter.

Funktionsweise

Der Baustein besteht aus einem internen Netzwerk mit einem einzelnen Konvertierungsbaustein `F_LWORD_TO_UINT` aus der IEC-61131-Bibliothek. Die Ablaufsteuerung erfolgt strikt ereignisgesteuert:

1. Ein Ereignis am Socket `AL_IN.E1` wird an den `REQ`-Eingang des Konvertierungsbausteins weitergeleitet.
2. Der Konvertierungsbaustein liest das Datum von `AL_IN.D1` (Typ `LWORD`) und wandelt es nach IEC-61131-Regeln in einen `UINT`-Wert um.
3. Nach Abschluss der Konvertierung erzeugt der Baustein ein Ereignis an seinem `CNF`-Ausgang, das direkt an den Plug `AUI_OUT.E1` übergeben wird.
4. Gleichzeitig wird der konvertierte Wert über den Datenausgang `OUT` an den Dateneingang `AUI_OUT.D1` übergeben.

Damit ist die gesamte Konvertierung in einem einzigen, unidirektionalen Durchlauf abgeschlossen.

Technische Besonderheiten

- **Bit-Reinterpretation**, werterhaltend nur, wenn der Quellwert in die schmalere Zielbreite passt (sonst werden die oberen Bits stillschweigend verworfen).
- **Kompositstruktur**: Der FB enthält keinen eigenen Zustandsautomaten, sondern delegiert die gesamte Funktionalität an den eingebetteten Konvertierungsbaustein.
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind vom Typ `unidirectional`. Es findet nur eine Vorwärtskonvertierung statt; Rückwärtsrichtung oder Statusrückmeldungen sind nicht vorgesehen.
- **Paketzuordnung**: Der FB befindet sich im Paket `adapter::conversion::unidirectional`, was auf seine Verwendung in modularen Adapterketten hinweist.
- **Wiederverwendbarkeit**: Der Baustein kann als universeller Umsetzer in beliebigen IEC-61499-Anwendungen eingesetzt werden, die eine LWORD-zu-UINT-Konvertierung über Adapter erfordern.

Zustandsübersicht

Da es sich um einen Composite-Baustein ohne eigene Zustandsmaschine handelt, ist die Zustandslogik vollständig durch den internen Baustein `F_LWORD_TO_UINT` bestimmt. Dieser arbeitet nach einem einfachen REQ/CNF-Schema:

- **Warten auf REQ**: Inaktiv, kein Ereignis am Eingang.
- **Konvertierung aktiv**: Nach Eintreffen von `REQ` wird die Umwandlung durchgeführt.
- **CNF ausgeben**: Nach Abschluss wird `CNF` gesetzt und die Ausgabedaten aktualisiert.

Eine detaillierte Zustandsbeschreibung des Konvertierungsbausteins ist der IEC-61131-Dokumentation zu entnehmen.

Anwendungsszenarien

- **Sensordatenanbindung**: Ein Sensor liefert Werte im LWORD-Format (z. B. 64-Bit-Zählerstände), die über einen AL-Adapter in das System eingespeist werden. Der `AL_TO_AUI`-Baustein wandelt diese in UINT (16-Bit) um, um sie an eine Steuerung mit UINT-Schnittstelle weiterzugeben.

- **Protokollkonvertierung**: In modularen Adapterketten dient der Baustein als Zwischenstufe, um von LWORD-basierten zu UINT-basierten Subsystemen zu wechseln, ohne die ursprüngliche Adapterlogik ändern zu müssen.
- **Datenreduktion**: Falls die oberen 48 Bit eines LWORD-Wertes nicht benötigt werden, kann der Baustein gezielt nur die unteren 16 Bit als UINT extrahieren (gemäß IEC-61131-Konvertierungsregeln).

Vergleich mit ähnlichen Bausteinen

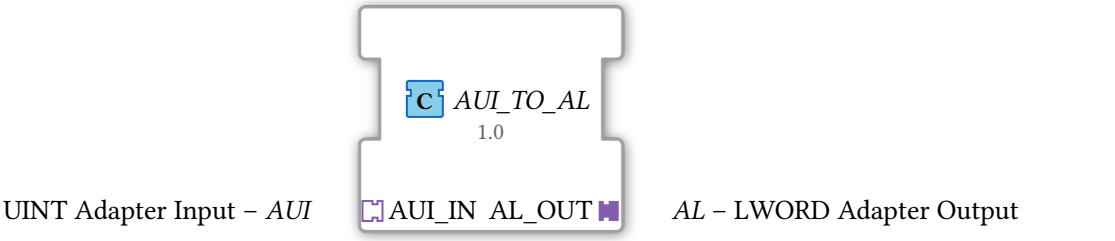
Baustein	Ausgangstyp	Besonderheit
`AL_TO_AUI`	UINT	Direkter Adapter-Konverter, unidirektional, Composite
`AL_TO_AUD`	DINT	Konvertiert LWORD in DINT (32-Bit)
`AUI_TO_AL`	LWORD	Rückwärtskonvertierung von UINT nach LWORD (nicht vorhanden, müsste separat implementiert werden)

Im Vergleich zu einer direkten Verwendung von `F_LWORD_TO_UINT` bietet `AL_TO_AUI` den Vorteil der einfachen Integration in Adapter-Schnittstellen, ohne dass manuell Ereignis- und Datenverbindungen hergestellt werden müssen.

Fazit

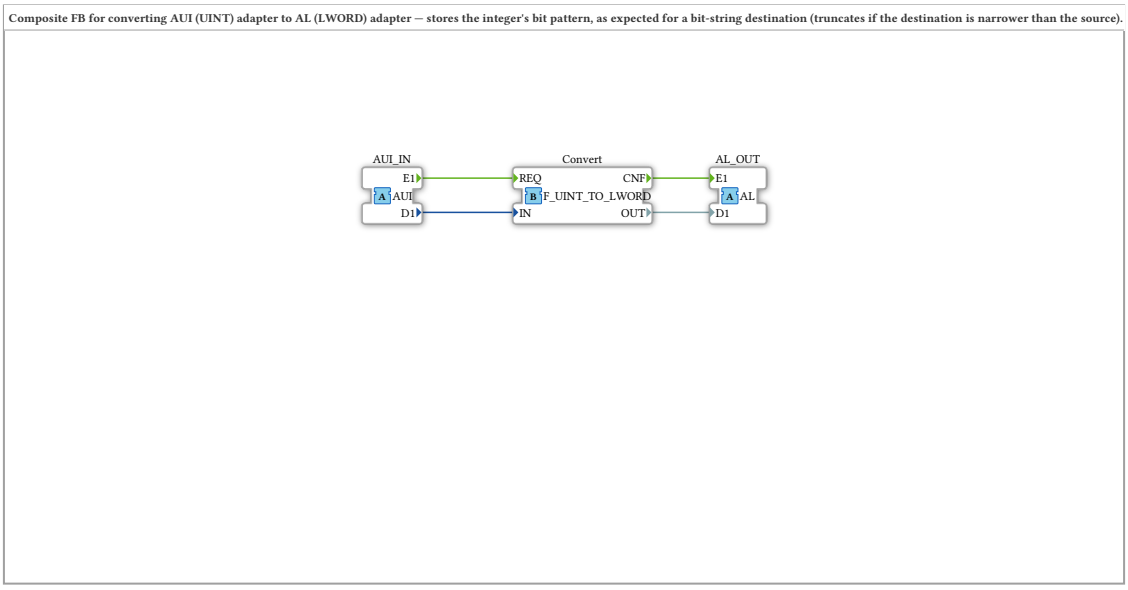
Der **AL_TO_AUI** ist ein komfortabler, standardisierter Adapter-Konverter für die Umwandlung von LWORD nach UINT. Dank seiner Composite-Struktur und der Nutzung bewährter IEC-61131-Konvertierungslogik lässt er sich nahtlos in modulare Adapterketten einfügen. Er eignet sich besonders für Anwendungen, bei denen heterogene Datentypen über einheitliche Adapterschnittstellen verbunden werden müssen.

AUI_TO_AL



UINT Adapter Input – *AUI*

AL – LWORD Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUI_TO_AL** ist ein zusammengesetzter (Composite) Funktionsblock, der die Konvertierung eines Adapters vom Typ AUI (basierend auf dem Datentyp UINT) in einen Adapter vom Typ AL (basierend auf dem Datentyp LWORD) ermöglicht. Er kapselt die standardisierte Konvertierungsfunktion `F_UINT_TO_LWORD` und stellt eine einfache, ereignisgesteuerte Schnittstelle für die unidirektionale Datenübertragung bereit.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine expliziten Einzelereignis-Eingänge, da die Ereignissteuerung über die angeschlossenen Adapter erfolgt. Das Ereignis **E1** des Sockets **AUI_IN** wird intern auf den Eingang `REQ` des Konvertierungsbausteins geleitet.

Ereignis-Ausgänge

Der Baustein gibt das Ereignis **E1** über den Plug **AL_OUT** aus, nachdem die Konvertierung abgeschlossen ist (Signal von `CNF` des internen Bausteins).

Daten-Eingänge

Der Daten-Eingang wird ausschließlich über den Socket bereitgestellt:

- **AUI_IN.D1** (Typ: UINT) – Der zu konvertierende Wert.

Daten-Ausgänge

Der Daten-Ausgang wird über den Plug bereitgestellt:

- **AL_OUT.D1** (Typ: LWORD) – Der konvertierte Wert (LWORD).

Adapter

Typ	Name	Richtung	Beschreibung
`AUI`	**AUI_IN**	Eingang (Socket)	Eingangsadapter für den UINT-Wert und das Trigger-Ereignis.
`AL`	**AL_OUT**	Ausgang (Plug)	Ausgangsadapter für den konvertierten LWORD-Wert und das Quittungsereignis.

Funktionsweise

Der Funktionsblock arbeitet ereignisgesteuert:

1. Ein eingehendes Ereignis am Socket **AUI_IN.E1** (ausgelöst durch den angeschlossenen Sender) aktiviert den internen Baustein `F_UINT_TO_LWORD` über dessen Ereigniseingang `REQ`.
2. Der Wert von **AUI_IN.D1** (UINT) wird an den Dateneingang `IN` des Konvertierungsbausteins übergeben.
3. Die Konvertierung erfolgt nach IEC 61131-Standard: Ein UINT-Wert wird in einen LWORD-Wert umgewandelt (die oberen 48 Bit werden mit Nullen aufgefüllt).
4. Nach Abschluss der Konvertierung wird das Ergebnis an **AL_OUT.D1** (LWORD) weitergegeben.
5. Zeitgleich wird das Quittungsereignis von `CNF` an **AL_OUT.E1** gesendet, um dem Empfänger die Verfügbarkeit des neuen Wertes zu signalisieren.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Zusammengesetzter Baustein (Composite FB)**: Die Implementierung verbirgt die interne Logik und erleichtert die Wiederverwendung in größeren Adapterketten.
- **Standardkonvertierung**: Verwendet die IEC 61131-konforme Funktion `F_UINT_TO_LWORD`, dadurch portabel und plattformunabhängig.
- **Unidirektionale Übertragung**: Die Adapter sind ausschließlich für eine Datenflussrichtung ausgelegt (kein Rückkanal).

- **Keine Zustandslogik**: Der Baustein besitzt keinen eigenen Zustandsautomaten; die Ablaufsteuerung erfolgt rein über die Ereignisverkettung.

Zustandsübersicht

Der Baustein **AUI_TO_AL** realisiert keine expliziten Zustände, da es sich um eine reine, ereignisgesteuerte Abbildung handelt. Es besteht folgender deterministischer Ablauf:

- **Warten**: Der Baustein wartet auf ein Ereignis am Eingangssocket.
- **Konvertieren**: Nach Ereigniseingang wird die Konvertierung ausgeführt.
- **Ausgeben**: Nach Abschluss wird das Ergebnis ausgegeben und der Baustein kehrt in den Wartezustand zurück.

Anwendungsszenarien

- **Protokollanpassung**: Wenn ein Datenlieferant UINT-Werte über einen AUI-Adapter bereitstellt, aber der Verbraucher einen AL-Adapter (LWORD) erwartet.
- **Systemintegration**: In verteilten Automatisierungssystemen mit unterschiedlichen Datentypen auf der Kommunikationsebene.
- **Adapterverkettung**: Als Zwischenglied in einer Adapterkaskade zur schrittweisen Typanpassung.

Vergleich mit ähnlichen Bausteinen

Im 4diac-Umfeld existieren weitere Konvertierungsadapter wie `AUI_TO_AB` (UINT nach BYTE) oder `AUI_TO_AW` (UINT nach WORD). **AUI_TO_AL** unterscheidet sich durch die Zielgröße LWORD (64 Bit), die für zeitkritische Anwendungen oder die Zusammenführung mehrerer Datenwörter in einen zusammenhängenden Wert genutzt wird. Es ersetzt keine universelle Typkonvertierung, sondern bietet eine dedizierte, ereignissynchrone Anpassung für die genannten Adaptertypen.

Fazit

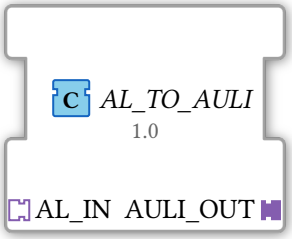
Der Funktionsblock **AUI_TO_AL** stellt eine saubere, standardisierte Lösung zur Umwandlung eines UINT-basierten Adapters (AUI) in einen LWORD-basierten Adapter (AL) dar. Er ist einfach zu handhaben, ereignisgesteuert und nutzt eine bewährte IEC 61131-Konvertierungsfunktion. Dank seines Composite-Charakters fügt er sich nahtlos in bestehende 4diac-Netzwerke ein und erleichtert die Integration unterschiedlicher Datenbreiten in der Automatisierungstechnik.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

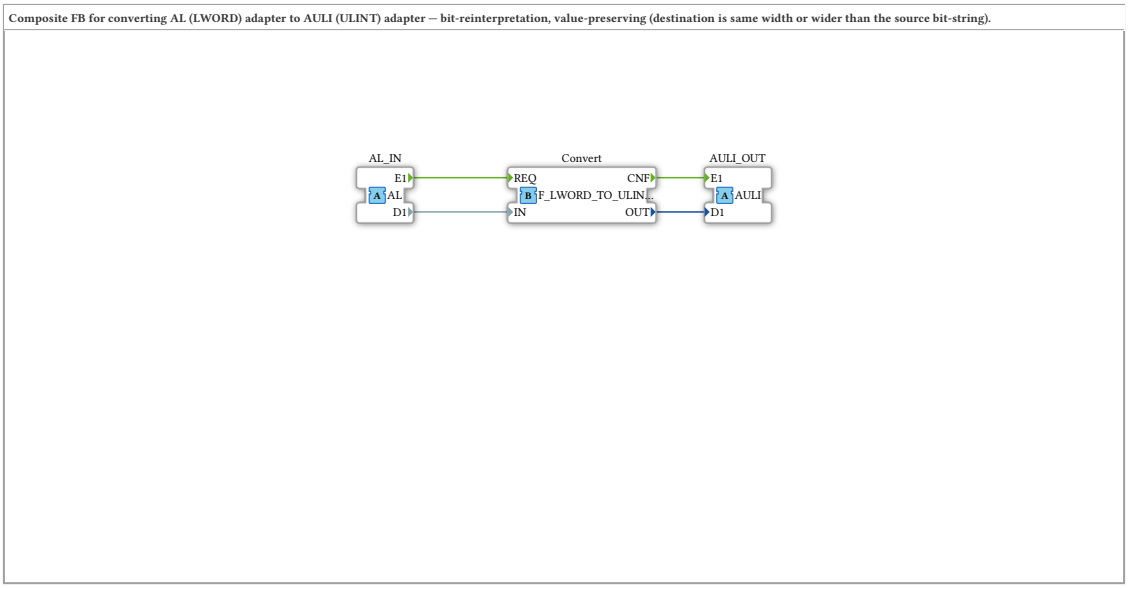
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AL_AULI

AL_TO_AULI



LWORD Adapter Input – *AL* — *AULI* – ULINT Adapter Output



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock **AL_TO_AULI** ist ein zusammengesetzter Baustein (Composite FB), der eine Datentypkonvertierung von einem **AL-Adapter** (Typ `LWORD`) in einen **AULI-Adapter** (Typ `ULINT`) vornimmt. Er dient als Brücke zwischen Komponenten, die mit unterschiedlichen Datentypen arbeiten, und ermöglicht eine nahtlose Integration in IEC 61499-basierten Systemen.

Schnittstellenstruktur

Ereignis-Eingänge

Über den Socket **AL_IN** (Adapter `AL`) wird das Ereignis **E1** empfangen. Dieses Ereignis triggert die Konvertierung des anliegenden LWORD-Werts.

Ereignis-Ausgänge

Über den Plug **AULI_OUT** (Adapter `AULI`) wird das Ereignis **E1** gesendet, sobald die Konvertierung abgeschlossen ist und der ULINT-Wert am Ausgang bereitsteht.

Daten-Eingänge

Name	Adapter-Port	Datentyp	Beschreibung
AL_IN	D1	LWORD	Eingangswert im LWORD-Format

Daten-Ausgänge

Name	Adapter-Port	Datentyp	Beschreibung
AULI_OUT	D1	ULINT	Konvertierter Wert im ULINT-Format

Adapter

Typ	Richtung	Name	Beschreibung
`AL`	Socket	AL_IN	Eingangsadapter mit LWORD-Daten
`AULI`	Plug	AULI_OUT	Ausgangsadapter mit ULINT-Daten

Funktionsweise

Der FB realisiert die Konvertierung durch einen internen Baustein `F_LWORD_TO_ULINT` aus der Bibliothek `iec61131::conversion`.

- Sobald am Eingangsadapter **AL_IN** ein Ereignis **E1** eintrifft, wird dieses an den Eingang **REQ** des internen Konvertierungsbausteins weitergeleitet.
- Gleichzeitig wird der Datenwert **D1** (LWORD) an den Eingang **IN** des Konverters übergeben.
- Nach erfolgreicher Konvertierung signalisiert der interne Baustein dies über seinen Ausgang **CNF**, welcher das Ereignis **E1** des Ausgangsadapters **AULI_OUT** auslöst.
- Der konvertierte ULINT-Wert steht dann am Datenausgang **D1** des Ausgangsadapters zur Verfügung.

Technische Besonderheiten

- **Bit-Reinterpretation, werterhaltend** (Zielbreite ist gleich groß oder größer als der Quell-Bit-String).
- **Lizenz:** Der Baustein wird unter der **Eclipse Public License 2.0** bereitgestellt.
- **Version:** 1.0 (Stand 2026-02-17)
- **Abhängigkeiten:** Er benötigt die Funktion `F_LWORD_TO_ULINT` aus der Bibliothek `iec61132::conversion`.
- **Portabilität:** Als Composite FB ist er plattformunabhängig und kann in jeder 4diac-IDE-Umgebung eingesetzt werden.
- **Typ-Hash:** Der Baustein besitzt ein Attribut `eclipse4diac::core::TypeHash` zur eindeutigen Identifikation (Wert leer).

Zustandsübersicht

Der FB **AL_TO_AULI** besitzt keine eigene Zustandsmaschine – er delegiert die gesamte Konvertierungslogik an den internen Baustein `F_LWORD_TO_ULINT`. Dadurch verhält er sich wie eine transparente Umwandlungsschicht, die nur bei einem eingehenden Ereignis aktiv wird und nach der Konvertierung wieder in den Ruhezustand zurückkehrt.

Anwendungsszenarien

- **Datentypanpassung** in SPS- oder Automatisierungssystemen, wenn ein LWORD-Signal (z. B. aus einem Kommunikationsmodul) in einen ULINT-Wert für eine nachfolgende Verarbeitung (z. B. Zähler oder Arithmetik) umgewandelt werden muss.
- **Adapter-Brücke** zwischen Funktionsblöcken, die ausschließlich mit dem AULI-Adapter arbeiten, und solchen, die den AL-Adapter verwenden.
- **Migration** von bestehenden LWORD-basierten Komponenten auf ULINT-basierte Systeme ohne Änderung der Signalkette.

Vergleich mit ähnlichen Bausteinen

- **F_LWORD_TO_ULINT (direkter Konverter):** Dieser reine Funktionsbaustein führt dieselbe Konvertierung durch, jedoch ohne Adapter-Schnittstellen. Er eignet sich für einfache Datenfluss-Modelle, während **AL_TO_AULI** speziell für den Einsatz in Adapter-basierten Architekturen optimiert ist.
- **Andere Adapter-Konverter (z. B. AL_TO_UDINT, AL_TO_DWORD):** Ähnliche Bausteine existieren für andere Zieltypen (UDINT, DWORD). Sie unterscheiden sich nur im Ausgabedatentyp, folgen aber dem gleichen Entwurfsmuster.

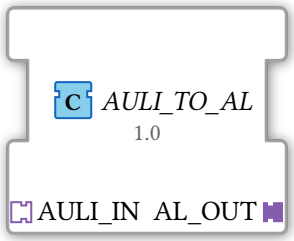
Fazit

Der **AL_TO_AULI**-Funktionsblock bietet eine einfache und robuste Möglichkeit, LWORD-Daten über Adapter in ULINT-Daten umzuwandeln. Durch die Kapselung der Konvertierungslogik in einem Composite FB bleibt die Schnittstelle klar und wiederverwendbar. Er eignet sich besonders für modulare IEC-61499-Projekte, in denen Adapter als standardisierte Kommunikationsschnittstellen eingesetzt werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

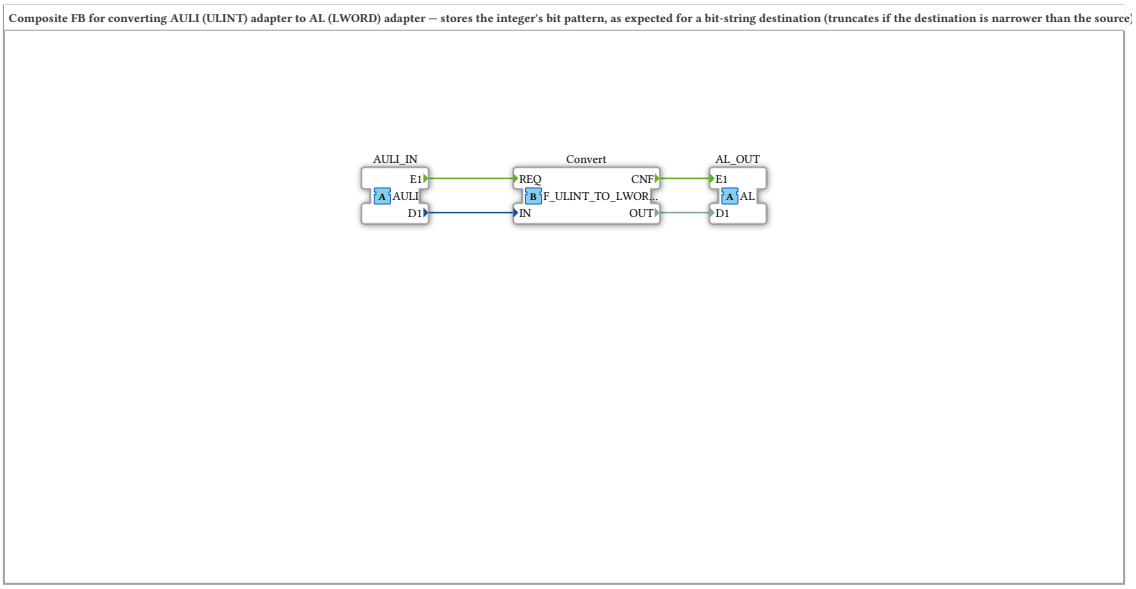
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AULI_TO_AL



ULINT Adapter Input – *AULI*

AL – LWORD Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein `AULI_TO_AL` ist ein Composite-FB zur Umwandlung eines AULI-Adapters (Datentyp ULINT) in einen AL-Adapter (Datentyp LWORD). Er nutzt intern die IEC-61131-Konvertierungsfunktion `F_ULINT_TO_LWORD` und ist für den Einsatz in unidirektionalen Adapter-Strukturen vorgesehen.

Schnittstellenstruktur

Ereignis-Eingänge

Bezeichnung	Beschreibung
AULI_IN.E1	Ereignisseingang über den Socket AULI_IN. Startet die Konvertierung.

Ereignis-Ausgänge

Bezeichnung	Beschreibung
AL_OUT.E1	Ereignisausgang über den Plug AL_OUT. Signalisiert den Abschluss der Konvertierung.

Daten-Eingänge

Bezeichnung	Datentyp	Beschreibung
AULI_IN.D1	ULINT	Eingangswert, der in LWORD konvertiert werden soll.

Daten-Ausgänge

Bezeichnung	Datentyp	Beschreibung
AL_OUT.D1	LWORD	Konvertierter Ausgangswert.

Adapter

Bezeichnung	Typ	Richtung	Beschreibung
AULI_IN	AULI (ULINT)	Socket	Unidirektionaler Eingangsadapter.
AL_OUT	AL (LWORD)	Plug	Unidirektionaler Ausgangsadapter.

Funktionsweise

Der Baustein arbeitet als reine Weiterleitung: Ein eingehendes Ereignis an `AULI_IN.E1` triggert den internen Funktionsbaustein `F_ULINT_TO_LWORD`, welcher den aktuellen Wert von `AULI_IN.D1` ausliest und in einen LWORD-Wert umrechnet. Nach Abschluss der Konvertierung wird der Ergebniswert an `AL_OUT.D1` ausgegeben und gleichzeitig ein Fertig-Ereignis an `AL_OUT.E1` gesendet. Die interne Logik beschränkt sich auf die Daten- und Ereignisverbindungen zwischen dem Socket und dem internen Konverter.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Composite-Implementierung**: Der Baustein enthält keine eigene Ablauflogik (ECC), sondern besteht ausschließlich aus einem verknüpften Funktionsbaustein der IEC-61131-Bibliothek.
- **Adapter-Architektur**: Sowohl Eingang als auch Ausgang sind als unidirektionale Adapter (Socket/Plug) ausgelegt, was eine lose Kopplung und einfache Wiederverwendung in verschiedenen Applikationen ermöglicht.
- **Typumwandlung**: Die Konvertierung von ULINT (vorzeichenloser 64-Bit-Integer) nach LWORD (64-Bit-Bitfeld) erfolgt nach IEC-61131-Norm.

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Die gesamte Funktionalität wird durch den internen FB `F_ULINT_TO_LWORD` realisiert, der als zustandslose Funktion arbeitet. Die Ereignissteuerung erfolgt ausschließlich über die in der Netzliste definierten Verbindungen.

Anwendungsszenarien

- Integration von Sensorsystemen, die ULINT-Daten über einen AULI-Adapter liefern, in Steuerungen, die LWORD-Daten über einen AL-Adapter erwarten.
- Datentyp-Migration oder Protokollanpassung in bestehenden Automatisierungslösungen.
- Einheitliche Adapterschnittstellen für Bibliotheken, die unterschiedliche physikalische Größen oder Adressräume abbilden.

Vergleich mit ähnlichen Bausteinen

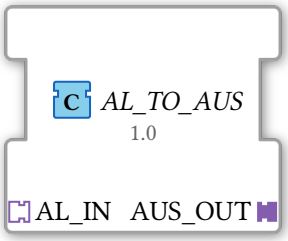
Im Gegensatz zu generischen Konverter-Bausteinen (z. B. `ULINT_TO_LWORD`) ist `AULI_TO_AL` speziell auf den Adapter-basierten Datenaustausch zugeschnitten. Ähnliche Adapter-Konverter existieren für andere Datentypen (z. B. `BOOL_TO_BYTE`, `INT_TO_DINT`), unterscheiden sich jedoch in den verwendeten Adapter-Typen und der internen Konvertierungsfunktion. Der vorliegende Baustein ergänzt die Bibliothek um eine saubere, typisierte Schnittstelle für ULINT↔LWORD.

Fazit

`AULI_TO_AL` ist ein einfacher, aber zuverlässiger Composite-FB zur Konvertierung zwischen AULI- und AL-Adaptern. Durch die direkte Nutzung der standardisierten IEC-61131-Funktion wird eine hohe Kompatibilität und Korrektheit der Umwandlung sichergestellt. Die Adapter-Schnittstellen ermöglichen eine modulare Einbindung in komplexe Automatisierungssysteme.

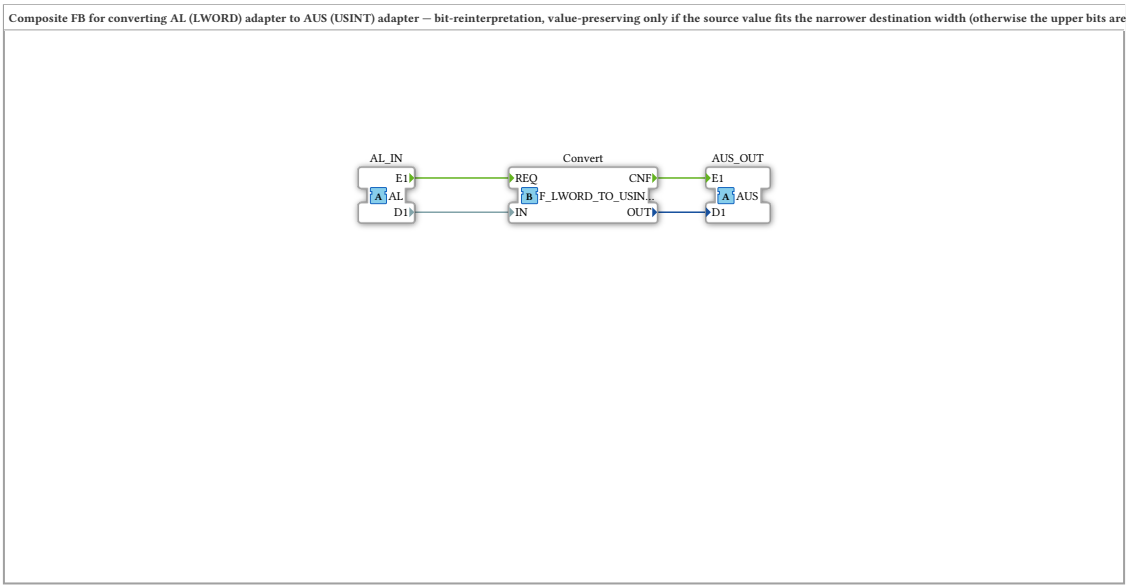
AL_AUS

AL_TO_AUS



LWORD Adapter Input – *AL*

AUS – USINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein AL_TO_AUS dient der Umwandlung eines AL-Adapters (LWORD) in einen AUS-Adapter (USINT). Er ist als Composite-FB realisiert und kapselt die Konvertierungslogik mithilfe des Bausteins F_LWORD_TO_USINT. Die Ein- und Ausgabe erfolgen über unidirektionale Adapter-Schnittstellen.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine direkten Ereignis-Eingänge. Stattdessen werden Ereignisse über den Adapter-Socket `AL_IN` empfangen. Der Socket stellt den Ereignis-Eingang `E1` bereit.

- **`AL_IN.E1`** (Ereignis): Auslöser für die Datenkonvertierung.

Ereignis-Ausgänge

Der FB besitzt keine direkten Ereignis-Ausgänge. Ereignisse werden über den Adapter-Plug `AUS_OUT` gesendet. Der Plug stellt den Ereignis-Ausgang `E1` bereit.

- **`AUS_OUT.E1`** (Ereignis): Signalisiert den Abschluss der Konvertierung und das Vorliegen des Ausgangswerts.

Daten-Eingänge

Die Daten werden über den Adapter-Socket `AL_IN` bereitgestellt.

- **`AL_IN.D1`** (LWORD): Der einzulesende LWORD-Wert, der in einen USINT-Wert umgewandelt werden soll.

Daten-Ausgänge

Die konvertierten Daten werden über den Adapter-Plug `AUS_OUT` ausgegeben.

- **`AUS_OUT.D1`** (USINT): Der konvertierte USINT-Wert.

Adapter

- **Socket `AL_IN`** (Typ: `adapter::types::unidirectional::AL`): Unidirektionaler LWORD-Eingangsadapter.
- **Plug `AUS_OUT`** (Typ: `adapter::types::unidirectional::AUS`): Unidirektionaler USINT-Ausgangsadapter.

Funktionsweise

Der Baustein wartet auf ein Ereignis am Eingangsadapter `AL_IN.E1`. Sobald dieses eintrifft, wird der an `AL_IN.D1` anliegende LWORD-Wert ausgelesen und an den internen Konvertierungsbaustein `F_LWORD_TO_USINT` übergeben. Dieser führt die Konvertierung in einen USINT-Wert durch. Nach erfolgreicher Konvertierung wird der Ergebniswert an den Ausgangsadapter `AUS_OUT.D1` weitergeleitet und gleichzeitig ein Ereignis an `AUS_OUT.E1` gesendet.

Die Verbindungen im Inneren sind:

- Vom Ereigniseingang des Sockets zum `REQ`-Eingang des Konverters.
- Vom `CNF`-Ausgang des Konverters zum Ereignisausgang des Plugs.
- Vom Dateneingang des Sockets zum `IN`-Eingang des Konverters.
- Vom `OUT`-Ausgang des Konverters zum Datenausgang des Plugs.

Technische Besonderheiten

- **Bit-Reinterpretation**, werterhaltend nur, wenn der Quellwert in die schmalere Zielbreite passt (sonst werden die oberen Bits stillschweigend verworfen).
- Der Baustein ist als Composite-FB realisiert, der auf die IEC-61131-Konvertierungsfunktion `F_LWORD_TO_USINT` zurückgreift.
- Die Adapter sind unidirektional ausgelegt, d.h. die Daten- und Ereignisflüsse verlaufen nur in eine Richtung.
- Der Konvertierungsbaustein stammt aus der Bibliothek `iec61131::conversion`.
- Durch die Kapselung in einem Composite-FB wird die Wiederverwendbarkeit und Austauschbarkeit der Konvertierungslogik erhöht.

Zustandsübersicht

Der Baustein hat keinen eigenen definierten Zustandsautomaten. Die interne Logik besteht aus einem einzigen Konvertierungsbaustein und festen Verbindungen. Der Ablauf ist rein ereignisgesteuert:

1. Warten auf Ereignis am Eingang (`AL_IN.E1`).
2. Konvertierung durchführen.
3. Ergebnis ausgeben und Ereignis am Ausgang (`AUS_OUT.E1`) setzen.

Anwendungsszenarien

- Anbindung von Sensoren oder Aktoren, die einen LWORD-Wert liefern, an ein System, das USINT erwartet.
- Protokollumsetzung in Automatisierungsanlagen, bei denen unterschiedliche Datentypen zwischen Adaptern konvertiert werden müssen.
- Einbettung in größere Composite-FBs, die mehrere Konvertierungsschritte kombinieren.

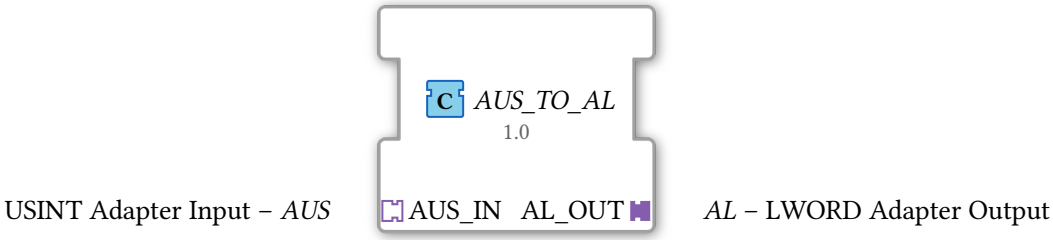
Vergleich mit ähnlichen Bausteinen

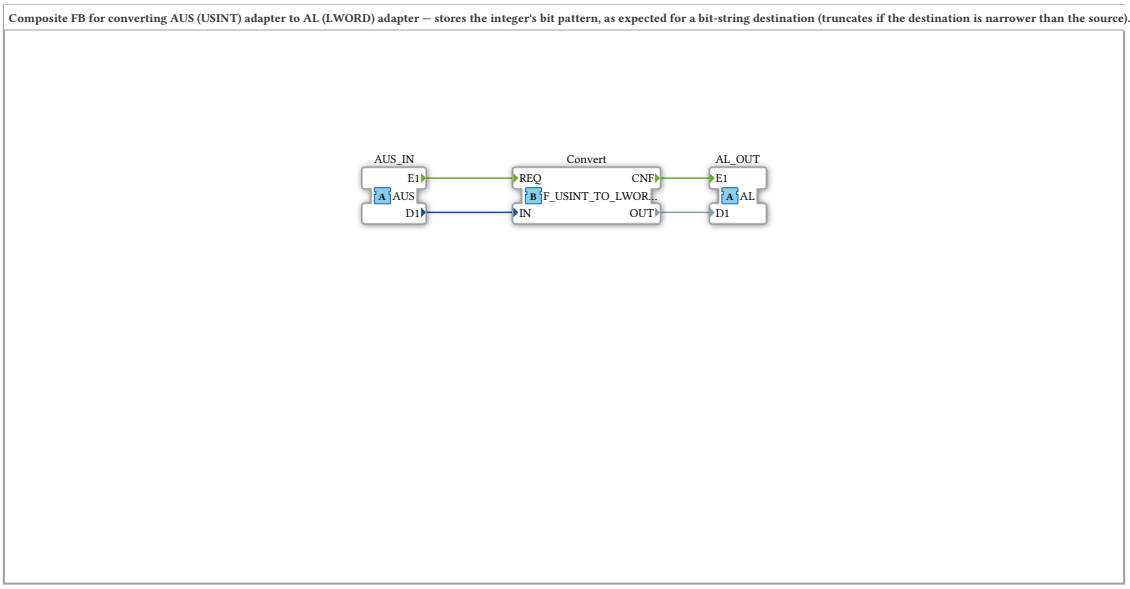
- **F_LWORD_TO_UINT**: Wandelt LWORD in UINT (16 Bit) um.
- **F_LWORD_TO_DWORD**: Wandelt LWORD in DWORD (32 Bit) um.
- **F_LWORD_TO_BYTE**: Wandelt LWORD in BYTE (8 Bit) um, liefert aber ein signed/unsigned? USINT ist unsigned 8 Bit.
- Der vorliegende Baustein füllt die Lücke für die direkte Umwandlung in den Datentyp USINT im Adapter-Kontext.

Fazit

Der Composite-FB AL_TO_AUS bietet eine einfache und saubere Lösung zur Umwandlung eines LWORD-Adapter-Eingangs in einen USINT-Adapter-Ausgang. Durch die Verwendung standardisierter Konvertierungsbausteine und die Kapselung in einem eigenen FB-Typ wird die Wartbarkeit und Lesbarkeit des Gesamtsystems verbessert. Er eignet sich besonders für Szenarien, in denen unidirektionale Adapter-Schnittstellen mit unterschiedlichen Datentypen verbunden werden müssen.

AUS_TO_AL





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUS_TO_AL** ist ein zusammengesetzter Baustein (Composite FB), der einen unidirektionalen **AUS**-Adapter (USINT-Datentyp) in einen unidirektionalen **AL**-Adapter (LWORD-Datentyp) umwandelt. Er dient als Brücke zwischen unterschiedlichen Datenformaten in einer 4diac-IDE-Umgebung, wobei die Konvertierung intern durch den IEC-61131-FB `F_USINT_TO_LWORD` realisiert wird. Der Baustein erleichtert die Integration von Komponenten, die auf verschiedenen Adaptertypen basieren.

Schnittstellenstruktur

Da es sich um einen Composite FB handelt, werden alle E/A-Signale über die beiden Adapter bereitgestellt. Der FB selbst besitzt keine eigenständigen Ereignis- oder Dateneingänge/-ausgänge.

Ereignis-Eingänge

- Keine eigenständigen Ereignis-Eingänge.

Ereignis-Ausgänge

- Keine eigenständigen Ereignis-Ausgänge.

Daten-Eingänge

- Keine eigenständigen Daten-Eingänge.

Daten-Ausgänge

- Keine eigenständigen Daten-Ausgänge.

Adapter

Bezeichnung	Typ	Richtung	Beschreibung
AUS_IN	`adapter::types::unidirectional::AUSS`	Socket (Eingang)	Empfängt den USINT-Wert sowie das zugehörige Ereignis. Der Adapter stellt intern einen Ereignis-Eingang `E1` und einen Daten-Eingang `D1` (USINT) bereit.
AL_OUT	`adapter::types::unidirectional:`	`AL` Plug (Ausgang)	Gibt den konvertierten LWORD-Wert aus. Der Adapter stellt intern einen Ereignis-Ausgang `E1` und einen Daten-Ausgang `D1` (LWORD) bereit.

Funktionsweise

1. Der Socket **AUS_IN** empfängt ein Ereignis an seinem internen Eingang `E1` und den zugehörigen USINT-Wert an `D1`.
2. Dieses Ereignis wird intern an den **REQ**-Eingang des Konvertierungs-FBs `F_USINT_TO_LWORD` weitergeleitet; der USINT-Wert wird an dessen **IN**-Eingang gelegt.
3. Der FB `F_USINT_TO_LWORD` führt die Umwandlung von USINT (8-Bit vorzeichenlos) zu LWORD (64-Bit vorzeichenlos) durch.
4. Nach Abschluss der Konvertierung wird am **CNF**-Ausgang von `F_USINT_TO_LWORD` ein Ereignis erzeugt, das an den internen Ereignis-Eingang des **AL_OUT**-Adapters (`E1`) gesendet wird. Gleichzeitig wird der konvertierte LWORD-Wert an den Daten-Ausgang `D1` des Adapters gelegt.
5. Der übergeordnete Baustein kann anschließend den LWORD-Wert über den Plug **AL_OUT** abgreifen und das Ereignis nutzen.

Die gesamte Abfolge ist streng kausal: Die Konvertierung wird nur dann ausgelöst, wenn am **AUS_IN** ein Ereignis eintrifft.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Composite FB**: Der Baustein kapselt die Konvertierungslogik in einem wiederverwendbaren Adapter-Paar.
- **Verwendete Bibliothek**: Die eigentliche Typumwandlung erfolgt durch `iec61131::conversion::F_USINT_TO_LWORD` aus der IEC-61131-Standardbibliothek.
- **Kein Zustand**: Der Baustein ist rein kombinatorisch mit ereignisgesteuerter Ausführung; er speichert keine internen Zustände.
- **Adapter-Unidirektionalität**: Sowohl AUS als auch AL sind als unidirektionale Adapter definiert, d.h. die Daten fließen nur in eine Richtung (Eingang → Ausgang).

Zustandsübersicht

Der Baustein besitzt kein eigenständiges Zustandsdiagramm. Sein Verhalten ist ausschließlich durch die Ereignis-/Datenflüsse des eingebetteten FBs `F_USINT_TO_LWORD` bestimmt:

- **Idle**: Warten auf ein Ereignis am **AUS_IN**.
- **Konvertierung aktiv**: Sobald ein Ereignis eintrifft, wird die Umwandlung durchgeführt. Nach Abschluss wird das Ausgangsereignis generiert und der Baustein kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- **Datenformat-Anpassung**: Ein Sensor oder eine Steuerung liefert Daten als USINT-Wert (z.B. Zählerstände), doch die nachgelagerte Komponente erwartet LWORD (z.B. für große Datenbereiche oder Zeitstempel).
- **Adapter-Brücken**: Wenn in einer 4diac-Anwendung Komponenten mit unterschiedlichen Adaptertypen verbunden werden müssen, ermöglicht dieser FB eine saubere, typensichere Umwandlung.
- **Unterstützung von Legacy-Systemen**: Altanlagen, die USINT-Daten ausgeben, können über diesen Baustein in moderne LWORD-Schnittstellen integriert werden.

Vergleich mit ähnlichen Bausteinen

- **F_USINT_TO_LWORD**: Ein reiner Funktionsbaustein (Basic FB) ohne Adapter-Kapselung. Er kann direkt in einem Netzwerk verwendet werden, erfordert aber die manuelle Verkabelung von Ereignissen und Daten.
- **AUS_TO_AL**: Bietet eine höhere Abstraktionsebene durch die Verwendung von Adaptern. Dadurch wird die Wiederverwendbarkeit und Austauschbarkeit von Komponenten verbessert, da die Adapter-Schnittstellen standardisiert sind.
- **Andere Typkonverter** (z.B. `INT_TO_DWORD`): Ähnliche Logik, jedoch für andere Datentypen; der vorliegende Baustein füllt die spezifische Lücke zwischen USINT und LWORD.

Fazit

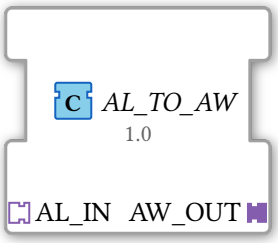
Der **AUS_TO_AL**-Funktionsblock ist ein praktischer Composite-FB zur Umwandlung von USINT- in LWORD-Daten, verpackt in ein adapterbasiertes Interface. Er nutzt die bewährte IEC-61131-Konvertierung und erlaubt eine saubere Integration in modulare 4diac-Anwendungen. Durch die Adapter-Kapselung bleibt die Schnittstelle klar definiert und der Baustein kann einfach in bestehende Kommunikationsstrukturen eingebunden werden.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

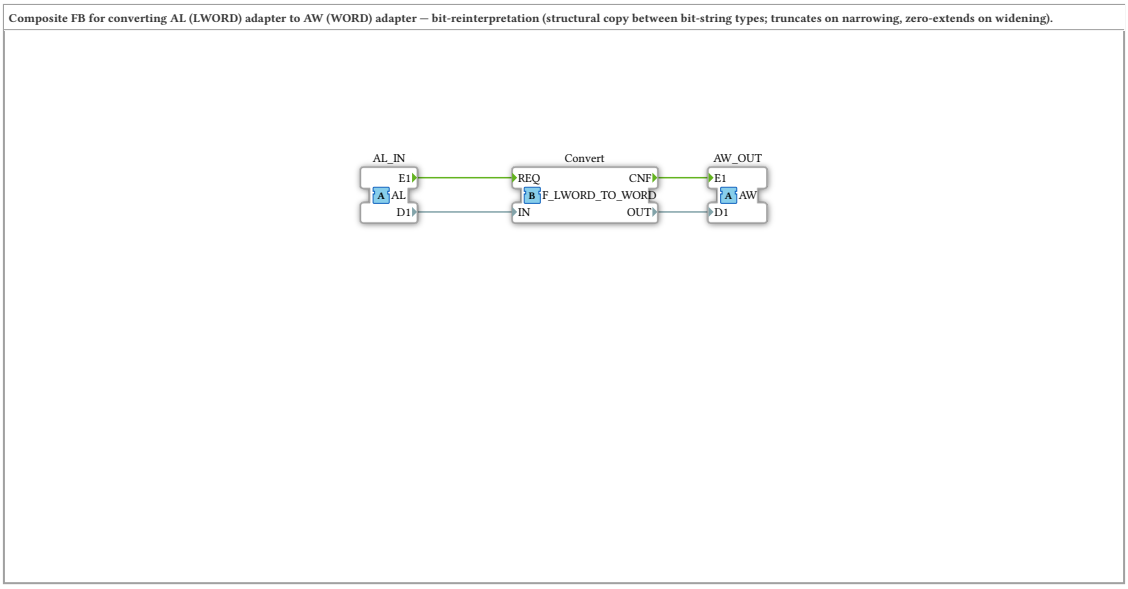
AL_AW

AL_TO_AW



LWORD Adapter Input – *AL*

AW – WORD Adapter Output



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock AL_TO_AW ist ein zusammengesetzter (Composite) Baustein, der einen LWORD-Eingangsadaper vom Typ AL (unidirektional) in einen WORD-Ausgangsadaper vom Typ AW (unidirektional) umwandelt. Er kapselt die Konvertierungsfunktion und stellt eine einfache Schnittstelle für die Umwandlung von 64-Bit-Daten auf 16-Bit-Daten in der 4diac-IDE bereit.

Schnittstellenstruktur

Der Baustein besitzt keine diskreten Ereignis- oder Datenports, sondern ausschließlich Adapter-Schnittstellen (Socket/Plug). Die nachfolgende Übersicht zeigt die verfügbaren Adapter:

Ereignis-Eingänge

Keine separaten Ereignis-Eingänge. Das Ereignissignal wird über den AL_IN-Adapter (Socket) geführt.

Ereignis-Ausgänge

Keine separaten Ereignis-Ausgänge. Das Ereignissignal wird über den AW_OUT-Adapter (Plug) ausgegeben.

Daten-Eingänge

Keine separaten Daten-Eingänge. Die Daten werden über den AL_IN-Adapter eingelesen.

Daten-Ausgänge

Keine separaten Daten-Ausgänge. Die Daten werden über den AW_OUT-Adapter ausgegeben.

Adapter

- **AL_IN** (Socket) – Typ: `adapter::types::unidirectional::AL` LWORD-Eingangsadaper; enthält je ein Ereignis- und Datensignal (E1, D1).
- **AW_OUT** (Plug) – Typ: `adapter::types::unidirectional::AW` WORD-Ausgangsadaper; enthält je ein Ereignis- und Datensignal (E1, D1).

Funktionsweise

Der Baustein arbeitet ereignisgesteuert:

1. Ein Ereignis am Socket **AL_IN** (E1) triggert den internen Konvertierungs-FB `F_LWORD_TO_WORD`.
2. Gleichzeitig wird das LWORD-Datum (D1) vom Eingangsadaper an den `IN`-Eingang des Konvertierungs-FB weitergeleitet.
3. Der Konvertierungs-FB führt die Umwandlung von LWORD (64-Bit) nach WORD (16-Bit) durch.

4. Nach erfolgreicher Konvertierung wird ein Ereignis am `CNF`-Ausgang des Konvertierungs-FB erzeugt, das an den Ausgangsadaper **AW_OUT** (E1) weitergegeben wird.
5. Das konvertierte WORD-Datum (16 Bit) wird am `OUT`-Ausgang des Konvertierungs-FB abgegriffen und an den Datenausgang des Adapters (D1) übergeben.

Die gesamte Logik besteht aus einer direkten Verbindungskette ohne Zwischenzustände oder Verzögerungen.

Technische Besonderheiten

- **Bit-Reinterpretation** (strukturelle Kopie zwischen Bit-String-Typen; schneidet bei Verengung ab, erweitert mit Nullen bei Erweiterung).
- **Verwendung standardisierter IEC 61131-Konvertierung** – Intern wird der Baustein `F_LWORD_TO_WORD` aus der IEC 61131-Bibliothek eingesetzt.
- **Adapterbasierte Schnittstelle** – Der Baustein ist vollständig in die Adapter-Konzepte von 4diac eingebettet und kann einfach in bestehende Systeme mit unidirektionalen AL- und AW-Adaptern integriert werden.
- **Keine Seiteneffekte** – Die Konvertierung erfolgt ohne Speicherung oder Zwischenpufferung; jedes eingehende Ereignis löst genau eine Ausgabe aus.
- **Kompatibilität** – Geeignet für Anwendungen, bei denen 64-Bit-LWORD-Datensignale auf 16-Bit-WORD-Signale reduziert werden müssen (z. B. bei der Anbindung älterer Steuerungen oder Feldbusse).

Zustandsübersicht

Der Baustein besitzt keine internen Zustände. Es handelt sich um eine rein kombinatorische Transformation ohne sequentielle Logik oder Zustandsautomaten.

Anwendungsszenarien

- **Protokollkonvertierung** – In Automatisierungssystemen, die unterschiedliche Datenwortbreiten verwenden (z. B. Umbau von 64-Bit-Messwerten auf 16-Bit-Ausgänge).
- **Schnittstellenanpassung** – Wenn ein System LWORD-Daten liefert, aber ein nachgeschalteter Adapter nur WORD-Daten verarbeitet.
- **Test- und Simulationsaufgaben** – Zum einfachen Umwandeln von Testdaten zwischen verschiedenen Adaptertypen.

Vergleich mit ähnlichen Bausteinen

- **F_LWORD_TO_WORD** (direkter Funktionsblock) – Bietet die gleiche Konvertierung, jedoch ohne Adapter-Kapselung. AL_TO_AW integriert diese Konvertierung direkt in eine adapterbasierte Umgebung.
- **Andere Adapter-Konvertierungen** – Es existieren ähnliche Bausteine für andere Datentyp-Paare (z. B. DWORD_TO_WORD). AL_TO_AW ist speziell für die Kombination LWORD↔WORD ausgelegt.
- **Selbst erstellte Composite-Bausteine** – Der Baustein zeigt, wie man durch Kapselung wiederkehrende Konvertierungsaufgaben effizient in der 4diac-IDE abbildet.

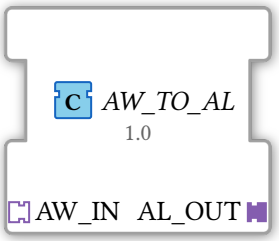
Fazit

AL_TO_AW ist ein kompakter und klar strukturierter Composite-Funktionsblock zur Umwandlung eines LWORD-Adapters in einen WORD-Adapter. Er nutzt die bewährte IEC-Konvertierungsfunktion, bietet eine adapterkonforme Schnittstelle und eignet sich hervorragend für die Integration in 4diac-Projekte, die eine saubere Trennung von Datenwortbreiten erfordern.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

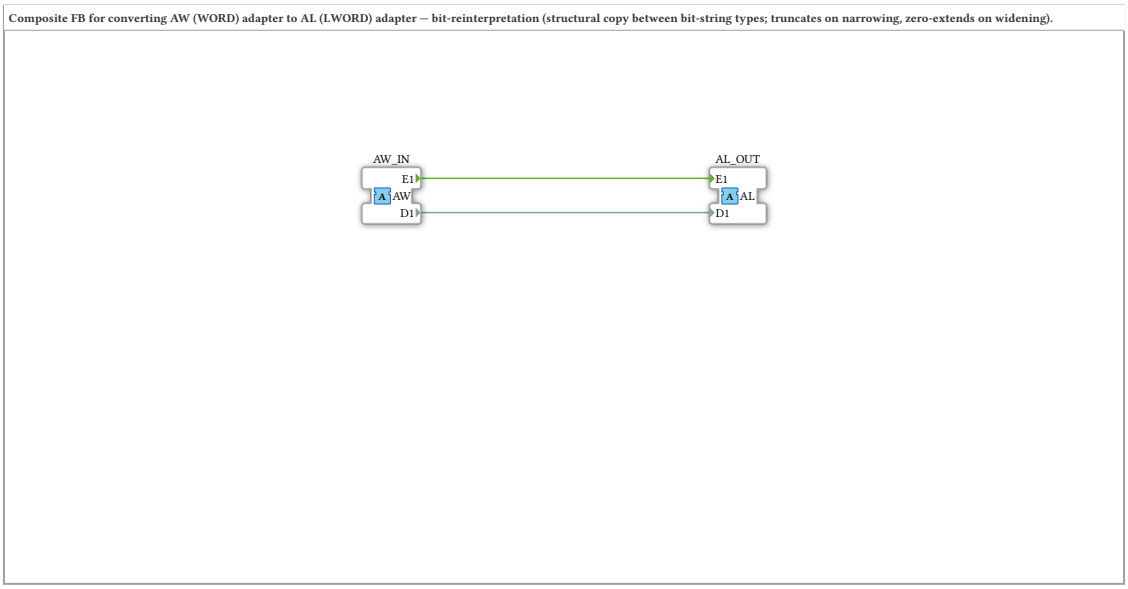
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AW_TO_AL



WORD Adapter Input – *AW*

AL – LWORD Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein AW_TO_AL ist ein zusammengesetzter Baustein zur Umwandlung eines WORD-Adapters (AW) in einen LWORD-Adapter (AL). Er wird verwendet, um Daten von einem 16-Bit-Format in ein 64-Bit-Format zu überführen, typischerweise in Steuerungs- und Automatisierungssystemen. Der Baustein leitet Ereignisse und Daten direkt von der Eingangsseite zur Ausgangsseite weiter und sorgt dabei für die Typanpassung.

Schnittstellenstruktur

Ereignis-Eingänge

Keine (Ereignisse werden ausschließlich über den Adapter AW_IN übertragen)

Ereignis-Ausgänge

Keine (Ereignisse werden ausschließlich über den Adapter AL_OUT ausgegeben)

Daten-Eingänge

Keine (Daten werden ausschließlich über den Adapter AW_IN empfangen)

Daten-Ausgänge

Keine (Daten werden ausschließlich über den Adapter AL_OUT gesendet)

Adapter

- **AW_IN** (Socket): Adapter vom Typ `adapter::types::unidirectional::AW` – WORD-Eingang (16 Bit) mit Ereignis- und Datenkanal (E1, D1).
- **AL_OUT** (Plug): Adapter vom Typ `adapter::types::unidirectional::AL` – LWORD-Ausgang (64 Bit) mit Ereignis- und Datenkanal (E1, D1).

Funktionsweise

Der Baustein realisiert eine einfache Durchschaltung: Das vom AW_IN-Adapter empfangene Ereignis (E1) wird direkt auf den Ereignisausgang AL_OUT.E1 weitergeleitet. Gleichzeitig wird das 16-Bit-Datensignal von AW_IN.D1 auf den 64-Bit-Datenausgang AL_OUT.D1 übertragen. Da der AL-Adapter LWORD erwartet, wird der WORD-Wert implizit in ein LWORD konvertiert (durch Null-Erweiterung oder explizite Typumwandlung gemäß der verwendeten Laufzeitumgebung). Der Baustein führt keine weiteren Operationen durch.

Technische Besonderheiten

- **Bit-Reinterpretation** (strukturelle Kopie zwischen Bit-String-Typen; schneidet bei Verengung ab, erweitert mit Nullen bei Erweiterung).
- Der Baustein ist als Composite-FB realisiert, d. h. er enthält ein internes Netzwerk, das die Adapter verbindet.
- Die Typkonvertierung von WORD zu LWORD erfolgt automatisch durch die Verkettung der Datenleitungen (implizite Typumwandlung).
- Ereignis- und Datenpfad sind synchron: Jedes eingehende Ereignis löst ein ausgehendes Ereignis aus, begleitet vom aktuellen Datenwert.
- Der Baustein ist unidirektional und unterstützt keine Rückwärtskommunikation.

Zustandsübersicht

Da der Baustein keine eigene Zustandslogik besitzt, sondern nur eine Durchleitung realisiert, existieren keine expliziten Zustände. Die Funktion ist rein kombinatorisch bzw. ereignisgetrieben ohne Speicherverhalten.

Anwendungsszenarien

- **Format-Anpassung**: Wenn ein Sensor oder eine Steuerung nur WORD-Daten (16 Bit) liefert, das nachgelagerte System aber LWORD-Daten (64 Bit) erwartet, kann dieser Baustein zur Anpassung eingesetzt werden.
- **Adapterbrücke**: In Kommunikationsketten, in denen unterschiedliche Adaptertypen verwendet werden, ermöglicht der Baustein die nahtlose Integration.
- **Protokollkonvertierung**: Als Teil einer größeren Adapter-Kette für die Umstellung von 16-Bit auf 64-Bit Datenbusse.

Vergleich mit ähnlichen Bausteinen

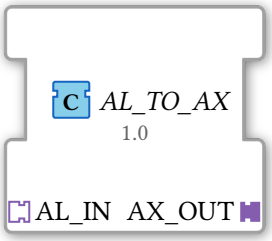
- **AW_TO_AW**: Identischer Typ, keine Konvertierung.
- **AL_TO_AW**: Rückwärtige Konvertierung von LWORD zu WORD (möglicherweise mit Datenverlust).
- **Direkte Verbindung ohne Konvertierung**: Wenn die Systeme denselben Adaptertyp verwenden, wird kein Konvertierungsbaustein benötigt.
- Der AW_TO_AL zeichnet sich durch seine Einfachheit und spezifische Typanpassung aus.

Fazit

Der AW_TO_AL-Funktionsbaustein bietet eine unkomplizierte Möglichkeit, WORD-Daten in ein LWORD-Format zu überführen. Er eignet sich besonders für Szenarien, in denen eine reine Typanpassung ohne zusätzliche Logik erforderlich ist. Die Implementierung als Composite-FB gewährleistet eine transparente und effiziente Durchleitung.

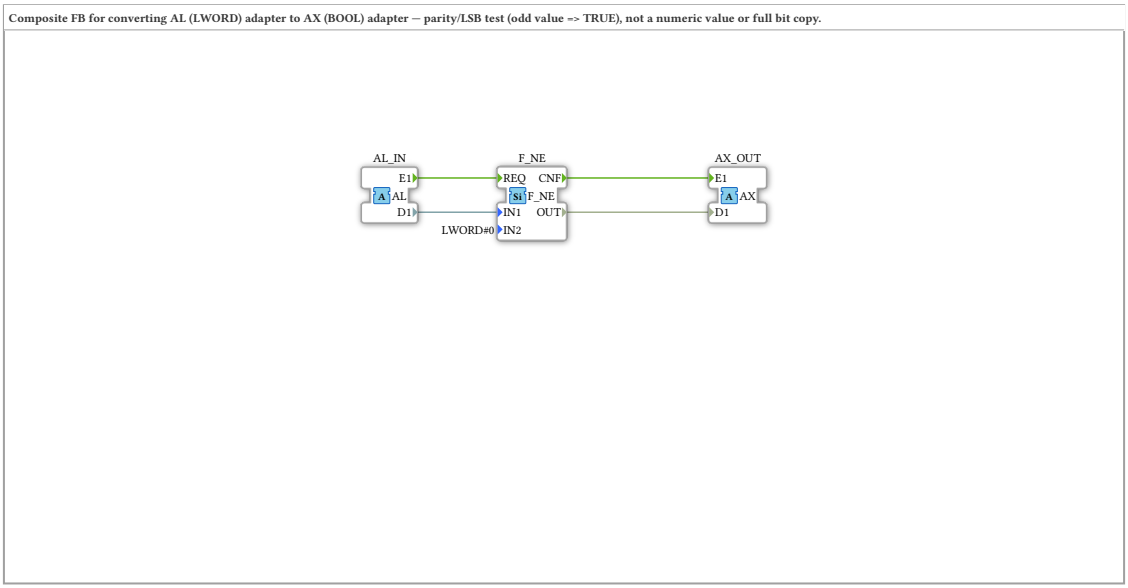
AL_AX

AL_TO_AX



LWORD Adapter Input – *AL*

AX – BOOL Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AL_TO_AX` ist ein Composite-Baustein, der eine unidirektionale Adapterkonvertierung von einem `AL`-Adapter (Datentyp `LWORD`) zu einem `AX`-Adapter (Datentyp `BOOL`) durchführt. Er ermöglicht die einfache Umwandlung eines LWORD-Signals in ein boolesches Signal, das dann typischerweise an nachfolgende BOOL-basierte Adapter oder Funktionsblöcke weitergegeben wird. Der Baustein prüft, ob der eingehende LWORD-Wert ungleich Null ist, und liefert als Ergebnis `TRUE`, wenn dies zutrifft.

Schnittstellenstruktur

Der Funktionsblock besitzt keine eigenen Ereignis- oder Datenein-/ausgänge, sondern verwendet ausschließlich Adapter-Schnittstellen:

- **Socket (Eingang):** `AL_IN` – Empfängt einen `AL`-Adapter (enthält ein LWORD und ein Ereignis).
- **Plug (Ausgang):** `AX_OUT` – Stellt einen `AX`-Adapter (enthält ein BOOL und ein Ereignis) bereit.

Die internen Verbindungen werden über die Adapter-Elemente abgewickelt:

Ereignis-Eingänge

(über den AL_IN-Adapter)

- **`AL_IN.E1`** – Ereigniseingang; löst die Konvertierung aus.

Ereignis-Ausgänge

(über den AX_OUT-Adapter)

- **`AX_OUT.E1`** – Ereignisausgang; signalisiert den Abschluss der Konvertierung.

Daten-Eingänge

(über den AL_IN-Adapter)

- **AL_IN.D1** (LWORD) – Der umzuwandelnde LWORD-Wert.

Daten-Ausgänge

(über den AX_OUT-Adapter)

- **AX_OUT.D1** (BOOL) – Das Ergebnis der Umwandlung: TRUE, wenn AL_IN.D1 ≠ 0; andernfalls FALSE.

Adapter

- **AL_IN** – Socket für den adapter::types::unidirectional::AL-Adapter (LWORD-basiert)
- **AX_OUT** – Plug für den adapter::types::unidirectional::AX-Adapter (BOOL-basiert)

Funktionsweise

Der Baustein arbeitet intern mit dem Funktionsblock F_NE („ungleich“) aus der IEC 61131-Bibliothek. Dieser vergleicht den eingehenden LWORD-Wert (AL_IN.D1) mit dem Konstantwert LWORD#0. Wenn die Werte ungleich sind, liefert F_NE.OUT den Wert TRUE, sonst FALSE.

Die ereignisgesteuerte Abfolge ist:

1. Ein Ereignis an AL_IN.E1 triggert den Eingang F_NE.REQ.
2. F_NE führt den Vergleich durch und legt das Ergebnis an F_NE.OUT an.
3. Nach Abschluss sendet F_NE ein Ereignis an F_NE.CNF, das mit AX_OUT.E1 verbunden ist.
4. Gleichzeitig wird das Ergebnis F_NE.OUT auf AX_OUT.D1 übertragen.

Somit wird bei jedem Ereignis am Eingang eine aktualisierte boolesche Ausgabe am Ausgang bereitgestellt.

Technische Besonderheiten

- **Paritäts-/LSB-Test** (ungerader Wert ⟹ TRUE), kein Zahlenwert und keine vollständige Bit-Kopie.
- **Typkonvertierung:** Der Baustein wandelt einen LWORD-Wert nicht bitweise, sondern als Ganzzahl in einen BOOL-Wert um. Jeder von Null verschiedene LWORD ergibt TRUE.
- **Adapterbasiert:** Durch die Verwendung der standardisierten unidirektionalen Adapter (AL und AX) ist der Baustein in beliebigen 4diac-Netzwerken einsetzbar, die diese Adaptertypen unterstützen.
- **Latenz:** Die Verarbeitung erfolgt synchron zum Eingangsereignis. Die Ausgabe wird im selben Ausführungszyklus aktualisiert.
- **Keine Zustandsspeicherung:** Der Baustein ist rein kombinatorisch; er speichert keine Werte zwischen zwei Aufrufen.

Zustandsübersicht

Der AL_TO_AX-Baustein besitzt keinen eigenen Zustandsautomaten. Sein Verhalten ist rein datenflussgesteuert durch das eingehende Ereignis. Eine Zustandsübersicht entfällt daher.

Anwendungsszenarien

- **Schwellwertdetektion:** Umwandlung eines LWORD-Sensorsignals (z. B. Zählerstand, Bitmaske) in ein boolesches Signal, das anzeigt, ob ein Wert ungleich Null anliegt.
- **Kopplung von LWORD- und BOOL-Adaptern:** In heterogenen Systemen, in denen ein Funktionsblock einen LWORD-basierten Adapter bereitstellt, der nächste Block jedoch einen BOOL-Adapter erwartet.
- **Fehlererkennung:** Ein LWORD-Fehlerregister (z. B. 0 = kein Fehler, ≠0 = Fehler) wird in ein einfaches BOOL-Signal „Fehler vorhanden“ übersetzt.

Vergleich mit ähnlichen Bausteinen

- **Direkte Konvertierer (z. B. LWORD_TO_BOOL):** Diese wandeln meist nur das niederwertigste Bit oder definieren ein festes Bit. Der AL_TO_AX-Baustein nutzt hingegen den Vergleich mit Null, sodass jedes beliebige gesetzte Bit oder ein von Null verschiedener Wert als TRUE interpretiert wird.
- **Adapter-Konverter:** Es gibt generische Adapterumsetzer, die mit verschiedenen Datentypen umgehen (z. B. ANY_TO_BOOL). Der spezialisierte AL_TO_AX-Baustein ist auf die genannte Adapterkombination zugeschnitten und erfordert keine externe Parametrierung.

Fazit

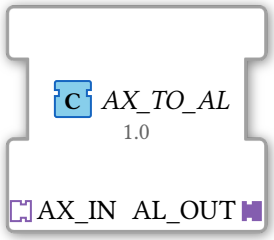
Der `AL_TO_AX`-Funktionsblock bietet eine einfache und zuverlässige Möglichkeit, einen unidirektionalen LWORD-basierten Adapter in einen BOOL-basierten Adapter zu konvertieren. Durch die Verwendung des Vergleichs mit Null ist die Interpretation eindeutig. Die Integration in bestehende 4diac-Netzwerke gestaltet sich dank der standardisierten Adapter-Schnittstellen unkompliziert. Der Baustein eignet sich besonders für schnelle, ereignisgesteuerte Umwandlungen ohne zusätzlichen Speicherbedarf und ist eine praktische Ergänzung für die Adapterkonvertierung in der industriellen Automatisierung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

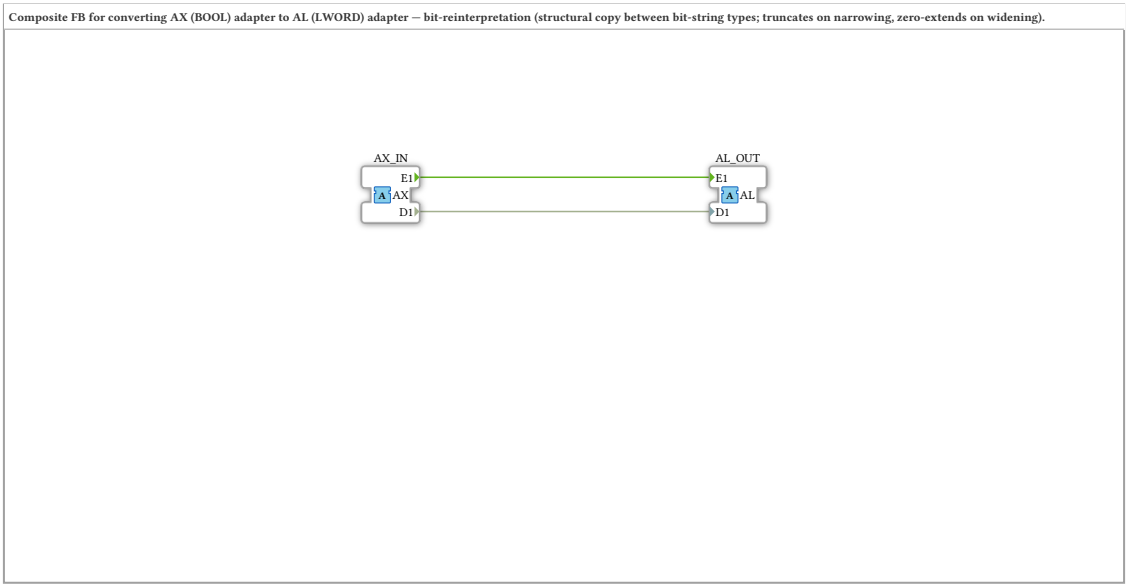
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AX_TO_AL

BOOL Adapter Input – *AX*



AL – LWORD Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AX_TO_AL** ist ein Composite-Baustein zur bidirektionalen Kopplung eines AX-Adapters (BOOL) mit einem AL-Adapter (LWORD). Er ermöglicht das Weiterleiten von Ereignissen und Daten zwischen diesen beiden Adapterschnittstellen, ohne eigene Logik zu implementieren. Der FB dient als einfache Konvertierungsbrücke und wird typischerweise in Automatisierungssystemen eingesetzt, um Komponenten mit unterschiedlichen Adapterformaten zu verbinden.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Adapter	Beschreibung
AX_IN.E1	AX (BOOL)	Ereigniseingang, der über den AX-Adapter angebunden wird.

Ereignis-Ausgänge

Name	Adapter	Beschreibung
AL_OUT.E1	AL (LWORD)	Ereignisausgang, der das eingehende Ereignis unverändert weitergibt.

Daten-Eingänge

Name	Adapter	Typ	Beschreibung
AX_IN.D1	AX (BOOL)	BOOL	Dateneingang für ein boolesches Signal.

Daten-Ausgänge

Name	Adapter	Typ	Beschreibung
AL_OUT.D1	AL (LWORD)	LWORD	Datenausgang, der das boolesche Signal als LWORD (64-Bit-Speicher) darstellt. Die Typkonvertierung erfolgt außerhalb des Bausteins.

Adapter

Typ	Name	Richtung
Socket	AX_IN	Eingang (BOOL)
Plug	AL_OUT	Ausgang (LWORD)

Funktionsweise

Der FB **AX_TO_AL** arbeitet als reiner Durchreiche-Baustein. Im FBNetzwerk werden die Ereignisverbindung `AX_IN.E1 → AL_OUT.E1` und die Datenverbindung `AX_IN.D1 → AL_OUT.D1` direkt hergestellt. Ein eingehendes Ereignis am AX-Socket löst sofort das entsprechende Ereignis am AL-Plug aus. Gleichzeitig wird der boolesche Wert von `AX_IN.D1` an `AL_OUT.D1` weitergegeben.

Der Baustein selbst führt keine Wertumwandlung durch; die Konvertierung des booleschen Signals in ein LWORD (z. B. 1 oder 0 in einem 64-Bit-Wort) wird von der zugrunde liegenden Adapter-Implementierung oder der Host-Umgebung bereitgestellt.

Technische Besonderheiten

- **Bit-Reinterpretation** (strukturelle Kopie zwischen Bit-String-Typen; schneidet bei Verengung ab, erweitert mit Nullen bei Erweiterung).
- **Composite-Architektur**: Der FB enthält kein internes ECC (Execution Control Chart) und keine eigene Logik. Er besteht lediglich aus zwei Adapterverbindungen und einfachen Durchleitungen.
- **Typzuordnung**: Die Zuordnung von BOOL zu LWORD erfolgt implizit. Der Anwender muss sicherstellen, dass die angeschlossenen Adapter die Konvertierung korrekt unterstützen.
- **Projektstruktur**: Der Baustein ist im Paket `adapter::conversion::unidirectional` abgelegt, was seine Funktion als unidirektionaler Konvertierungsadapter kennzeichnet.

Zustandsübersicht

Der FB besitzt keinen Zustandsautomaten und keine eigenen Zustände. Er arbeitet ereignisgesteuert und reagiert passiv auf Signale des AX-Adapters.

Anwendungsszenarien

- Einbinden einer alten Bool-Schnittstelle (AX) in ein System, das nur LWORD-Adapter (AL) unterstützt.
- Temporäre Brückenschaltung während der Migration von Feldbusprotokollen.
- Testumgebungen, in denen ein boolesches Signal als 64-Bit-Wert interpretiert werden soll.

Vergleich mit ähnlichen Bausteinen

Es existieren weitere Konvertierungsbausteine wie z. B. `AL_TO_AX` (Rückwandlung von LWORD nach BOOL) oder `BOOL_TO_LWORD` (direkte Umwandlung ohne Adapter-Kapselung). Im Gegensatz zu diesen

reinen Wertwandlern kapselt **AX_TO_AL** die gesamte Adapter-Schnittstelle, sodass der Anwender nur die Adapter verbinden muss.

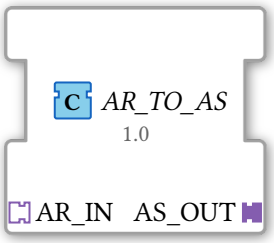
Fazit

Der Funktionsblock **AX_TO_AL** ist ein schlanker, aber nützlicher Adapter-Koppler. Er reduziert den Aufwand, wenn unterschiedliche Adapterformate in einem Automatisierungssystem zusammengeführt werden müssen, und erfordert keine eigenen Logik- oder Zustandsdefinitionen. Die einfache Struktur erleichtert das schnelle Einfügen in bestehende IEC-61499-Netzwerke.

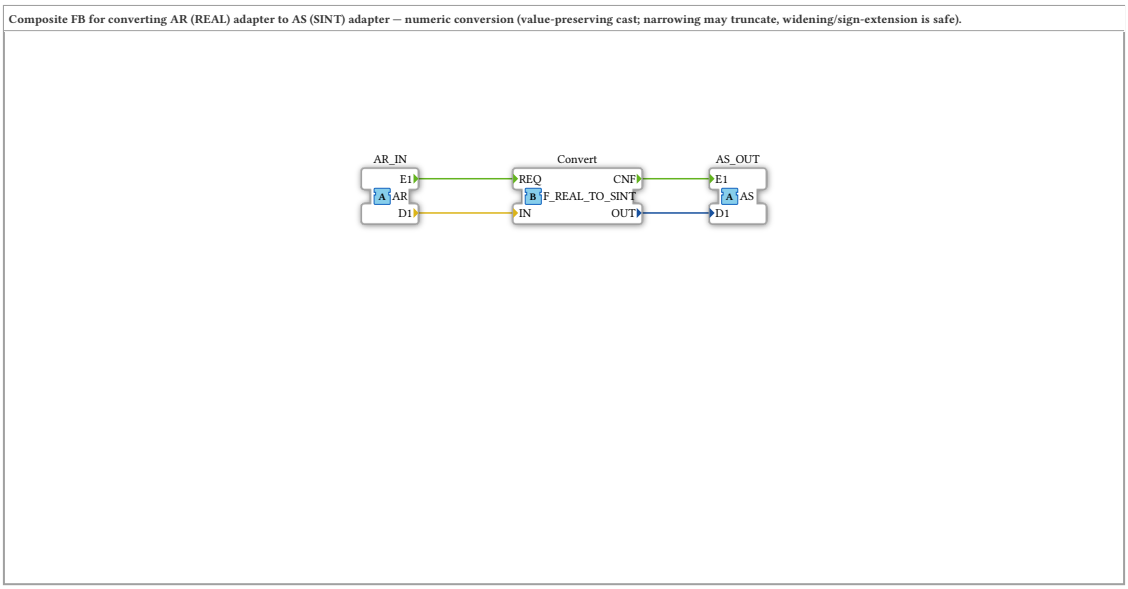
AR_AS

AR_TO_AS

REAL Adapter Input – *AR*



AS – SINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AR_TO_AS** ist ein Composite-Baustein, der einen **REAL-Adapter** (AR) in einen **SINT-Adapter** (AS) umwandelt. Er ermöglicht die einfache und standardisierte Konvertierung zwischen den Adaptertypen und kapselt die dafür notwendige Logik in einem wiederverwendbaren Baustein.

Schnittstellenstruktur

Der FB besitzt keine separaten Ereignis- oder Datenports der IEC 61499-Ebene. Die gesamte Kommunikation erfolgt über die integrierten Adapter. Die nachfolgende Tabelle beschreibt die verfügbaren Adapter.

Ereignis-Eingänge

Keine eigenen Ereignis-Eingänge vorhanden. Ereignisse werden über den **AS_OUT-Adapter** empfangen (siehe Adapter).

Ereignis-Ausgänge

Keine eigenen Ereignis-Ausgänge vorhanden. Ereignisse werden über den **AR_IN-Adapter** gesendet (siehe Adapter).

Daten-Eingänge

Keine eigenen Daten-Eingänge vorhanden. Daten werden über den **AS_OUT-Adapter** empfangen (siehe Adapter).

Daten-Ausgänge

Keine eigenen Daten-Ausgänge vorhanden. Daten werden über den **AR_IN-Adapter** gesendet (siehe Adapter).

Adapter

Adapter	Typ	Richtung	Beschreibung
AR_IN	adapter::types::unidirectional::AR	Socket (eingehend)	Unidirektionaler REAL-Adapter. Liefert einen REAL-Wert (D1) und ein zugehöriges Ereignis (E1).
AS_OUT	adapter::types::unidirectional::AS	Plug (ausgehend)	Unidirektionaler SINT-Adapter. Empfängt einen SINT-Wert (D1) und ein zugehöriges Ereignis (E1).

Funktionsweise

Der Baustein realisiert die Konvertierung wie folgt:

1. Ein eingehendes Ereignis auf **AR_IN.E1** triggert den internen Konvertierungsbaustein **F_REAL_TO_SINT**.
2. Der mit dem Ereignis verbundene REAL-Wert von **AR_IN.D1** wird an den Eingang `IN` des Konvertierungsbausteins weitergeleitet.
3. **F_REAL_TO_SINT** wandelt den REAL-Wert in einen SINT-Wert um und legt das Ergebnis an seinem Ausgang `OUT` an.
4. Nach Abschluss der Konvertierung wird ein Ereignis über **AS_OUT.E1** gesendet, zusammen mit dem konvertierten SINT-Wert auf **AS_OUT.D1**.

Es findet eine synchrone 1:1-Umsetzung statt – jede eingehende REAL-Anforderung erzeugt genau eine ausgehende SINT-Antwort.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Wiederverwendbarkeit**: Der Baustein basiert auf dem Standard-FB `F_REAL_TO_SINT` aus der IEC 61131-Bibliothek. Dadurch ist die Konvertierungslogik normgerecht und auf vielen Plattformen verfügbar.
- **Adapter-Schnittstelle**: Die Verwendung von Adaptern erlaubt eine lose Kopplung zwischen Funktionsblöcken und erleichtert den Austausch oder die Erweiterung der Schnittstelle.
- **Wertebereich**: Bei der Konvertierung von REAL (ca. $\pm 3{,}4 \cdot 10^{38}$) nach SINT ($-128 \ldots 127$) kann es zu Wertebereichsüberschreitungen kommen. Der interne FB `F_REAL_TO_SINT` verhält sich dabei gemäß der IEC 61131-Definition (Überlauf oder Verwendung des sättigenden Verhaltens). Der Anwender sollte die Eingangswerte entsprechend begrenzen.

Zustandsübersicht

Der **AR_TO_AS** besitzt keinen eigenen Zustandsautomaten. Da es sich um einen reinen Composite-Baustein handelt, der nur die Verbindung zwischen dem eingehenden und dem ausgehenden Adapter herstellt, gibt es kein internes Zustandsverhalten. Der Baustein arbeitet kombinatorisch auf Ereignisebene.

Anwendungsszenarien

- **Integration von REAL-basierten Bausteinen in SINT-Umgebungen**: Wenn eine Komponente einen REAL-Adapter bereitstellt, der nachfolgende Funktionsblock jedoch einen SINT-Adapter erwartet, kann **AR_TO_AS** als vermittelnder Adapter eingesetzt werden.

- **Typsichere Konvertierung in modularen Steuerungen**: Durch die Verwendung von Adaptern bleibt die Typisierung erhalten, und die Konvertierung ist explizit sichtbar im Netzwerk.
- **Erweiterung von Bibliotheken**: Der Baustein kann als Teil einer Adapter-Konvertierungsbibliothek verwendet werden, um den Datenaustausch zwischen unterschiedlichen numerischen Auflösungen zu standardisieren.

Vergleich mit ähnlichen Bausteinen

Es existieren analoge Konvertierungs-Adapter, z. B.:

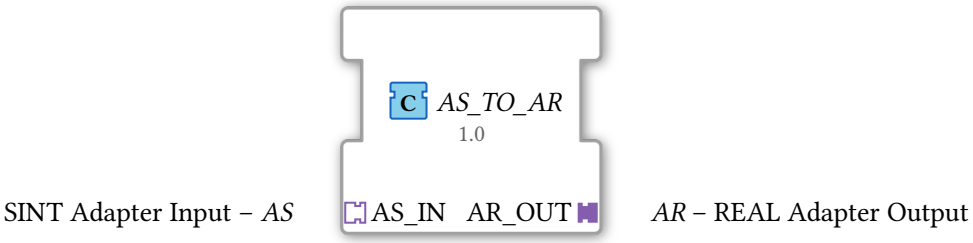
- **AR_TO_INT** – REAL ⟶ INT
- **AR_TO_UDINT** – REAL ⟶ UDINT
- **AR_TO_LREAL** – REAL ⟶ LREAL

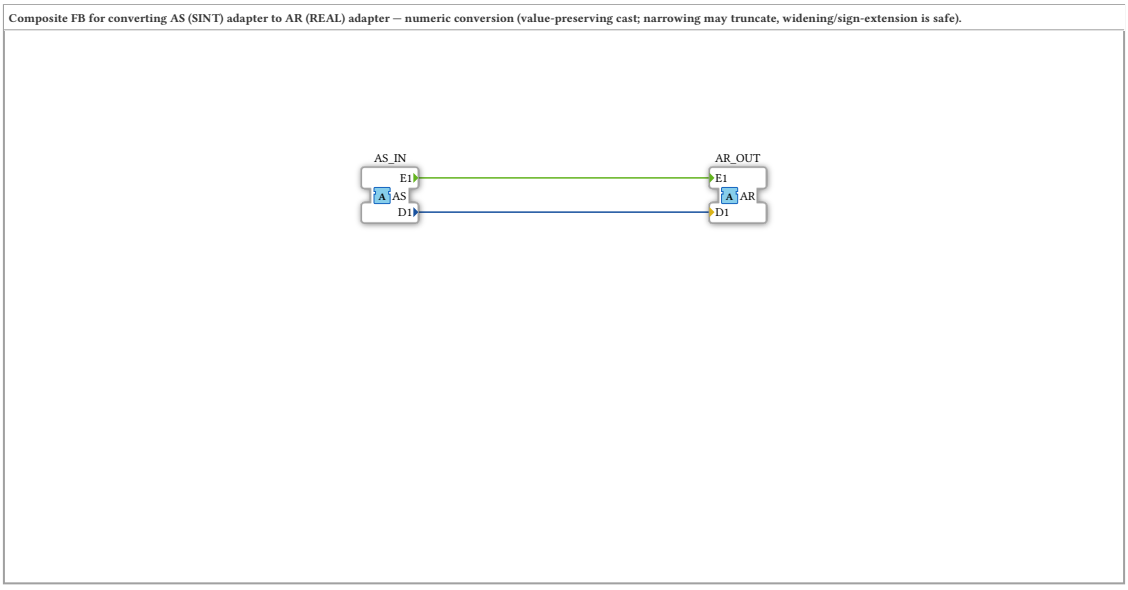
Gemeinsam ist allen die Verwendung des entsprechenden IEC 61131-Konvertierungsbausteins im Inneren. Der Unterschied liegt im Zieltyp und der Adapter-Definition. **AR_TO_AS** ist speziell für die Konvertierung in den schmalen, vorzeichenbehafteten 8-Bit-SINT-Typ ausgelegt.

Fazit

Der **AR_TO_AS** ist ein einfacher, aber effektiver Composite-Baustein zur Umwandlung eines REAL-Adapters in einen SINT-Adapter. Er kapselt die Konvertierungslogik sauber und ermöglicht eine typsichere, ereignisgesteuerte Kommunikation zwischen Funktionsblöcken mit unterschiedlichen numerischen Schnittstellen. Dank der Verwendung von Standard-Bibliotheken ist er zuverlässig und plattformunabhängig einsetzbar.

AS_TO_AR





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AS_TO_AR` ist ein Composite-Baustein, der einen unidirektionalen Adapter vom Typ **AS** (SINT) auf einen Adapter vom Typ **AR** (REAL) umsetzt. Er wird eingesetzt, um eine logische Verbin-

dung zwischen Komponenten herzustellen, die auf unterschiedlichen Datentypen basieren – speziell von einem vorzeichenbehafteten 8-Bit-Integer (SINT) zu einer Gleitkommazahl (REAL).

Schnittstellenstruktur

Ereignis-Eingänge
Der Baustein besitzt keine eigenständigen Ereignis-Eingänge. Ereignisse werden ausschließlich über den integrierten Adapter **AS_IN** empfangen.

- **AS_IN.E1** – Ereignis vom angeschlossenen SINT-Adapter.

Ereignis-Ausgänge
Ereignisse werden über den integrierten Adapter **AR_OUT** weitergegeben.

- **AR_OUT.E1** – Ereignis zum angeschlossenen REAL-Adapter.

Daten-Eingänge
Daten werden ausschließlich über den Adapter **AS_IN** bereitgestellt.

- **AS_IN.D1** – Datenwert vom Typ **SINT** (vorzeichenbehafteter 8-Bit-Integer).

Daten-Ausgänge
Daten werden ausschließlich über den Adapter **AR_OUT** ausgegeben.

- **AR_OUT.D1** – Datenwert vom Typ **REAL** (Gleitkommazahl).

Adapter

Typ	Name	Richtung	Kommentar
`adapter::types::unidirectional::AS`	**AS_IN**	Socket	Eingangsadapter für SINT-Daten
`adapter::types::unidirectional::AR`	**AR_OUT**	Plug	Ausgangsadapter für REAL-Daten

Funktionsweise
Der `AS_TO_AR`-Baustein verhält sich als reine Typkonvertierungs-Durchleitung. Im enthaltenen Funktionsnetzwerk werden das Ereignis `AS_IN.E1` direkt mit `AR_OUT.E1` und der Datenwert `AS_IN.D1` direkt mit `AR_OUT.D1` verbunden. Die eigentliche Umwandlung von **SINT** in **REAL** erfolgt implizit durch die Adapter-Definitionen des jeweiligen Typs. Der Baustein selbst führt keine zusätzliche Logik oder Berechnung aus.

Technische Besonderheiten
- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-Baustein**: Die Konvertierung wird allein durch die Verschaltung von Adaptern im Funktionsnetzwerk realisiert. Eine eigene Zustandsmaschine (ECC) ist nicht vorhanden.
- **Paket-Zugehörigkeit**: Der Baustein ist im Paket `adapter::conversion::unidirectional` eingeordnet.
- **Keine Datenhaltung**: Alle Werte werden instantan weitergeleitet; der Baustein besitzt kein internes Gedächtnis.
- **Implizite Typumwandlung**: Die Konvertierung von Ganzzahl (SINT) in Gleitkommazahl (REAL) findet in der Peripherie der Adapter-Typen statt.

Zustandsübersicht
Der `AS_TO_AR`-Baustein besitzt keinen Zustandsautomaten (ECC). Das Verhalten ist rein kombinatorisch und beschränkt sich auf die direkte Weiterleitung von Ereignissen und Daten.

Anwendungsszenarien
- Anbindung eines Sensors, der seine Messwerte als SINT-Adapter bereitstellt, an eine Verarbeitungskomponente, die REAL-Adapter erwartet.
- Übergang zwischen unterschiedlichen ‚4diac'-Adaptertypen in heterogenen Steuerungsprojekten.

- Vorbereitung von Werten für mathematische Operatoren oder Visualisierungskomponenten, die REAL-Eingänge benötigen.

Vergleich mit ähnlichen Bausteinen

- **AR_TO_AS** – Führt die umgekehrte Konvertierung von REAL nach SINT durch.
- **Allgemeine Typkonverter** (z. B. `INT_TO_REAL`) arbeiten auf Ebene einzelner Datenports, während `AS_TO_AR` ganze Adapterschnittstellen umsetzt.
- Der Baustein bietet keine einstellbaren Parameter (z. B. Rundungsmodi); eine spezifischere Konvertierung müsste über separate Funktionsbausteine realisiert werden.

Fazit

`AS_TO_AR` ist ein einfacher, aber nützlicher Composite-Baustein zur Adapter-Typkonvertierung von SINT auf REAL. Er ermöglicht die verlustfreie Integration von Komponenten mit unterschiedlichen Datentypen in durchgängigen Steuerungsabläufen und zeichnet sich durch geringe Komplexität und direkte Durchleitung aus.

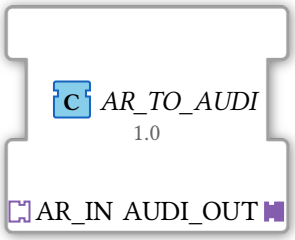
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

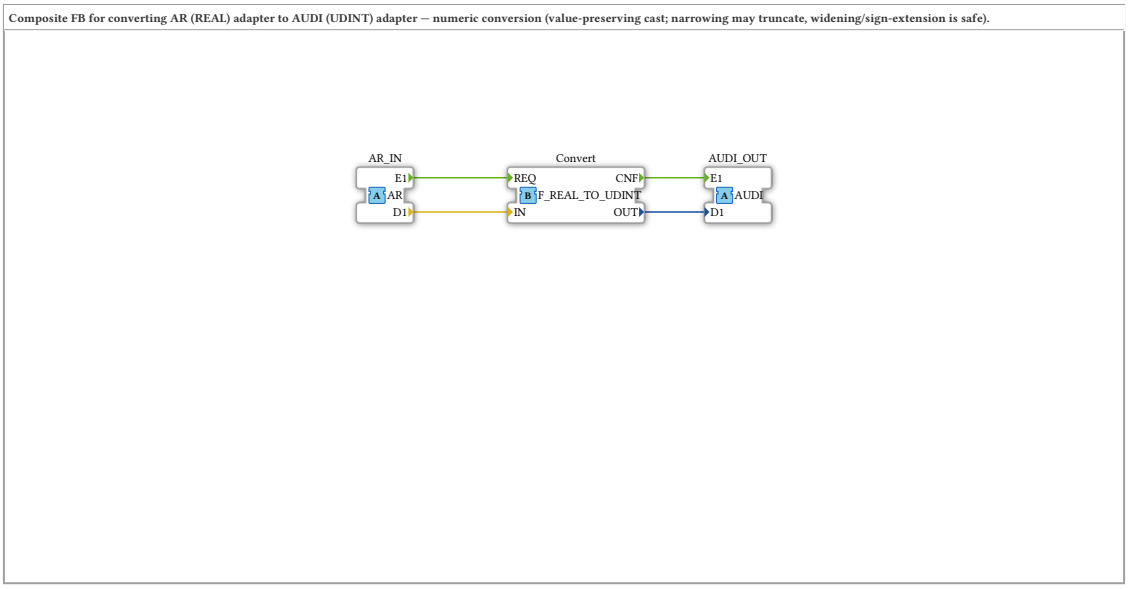
AR_AUDI

AR_TO_AUDI

REAL Adapter Input – *AR*



AUDI – UDINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `AR_TO_AUDI` ist ein zusammengesetzter Baustein (Composite FB), der einen REAL-Wert von einem unidirektionalen **AR**-Adapter entgegennimmt, in einen **UDINT**-Wert konvertiert und über einen **AUDI**-Adapter ausgibt. Er kapselt die Typkonvertierung `REAL_TO_UDINT` und vereinfacht die Einbindung in adapterbasierte Kommunikationsstrukturen.

Schnittstellenstruktur

Der Baustein besitzt keine eigenen Ereignis- oder Daten-Ein-/Ausgänge. Die gesamte Kommunikation erfolgt über zwei Adapter (Socket und Plug).

Adapter (Socket – Eingang)

Bezeichnung	Typ	Kommentar
AR_IN	adapter::types::unidirectional::AR	Unidirektionaler REAL-Adapter als Eingang

Der Adapter AR_IN liefert ein Ereignis E1 und einen Daten-Eingang D1 vom Typ REAL.

Adapter (Plug – Ausgang)

Bezeichnung	Typ	Kommentar
AUDI_OUT	adapter::types::unidirectional::AUDI	Unidirektionaler UDINT-Adapter als Ausgang

Der Adapter AUDI_OUT erwartet ein Ereignis E1 und einen Daten-Ausgang D1 vom Typ UDINT.

Funktionsweise

Der Baustein enthält intern eine Instanz des Konvertierungsbausteins F_REAL_TO_UDINT aus der Bibliothek iec61131::conversion. Die Signalverbindungen realisieren einen einfachen Datenfluss:

1. **Ereignis vom Eingangsadapter** Das Ereignis AR_IN.E1 triggert den Konvertierungsbaustein über dessen Ereigniseingang REQ.
2. **Datenkonvertierung** Gleichzeitig wird der REAL-Wert von AR_IN.D1 an den Dateneingang IN von F_REAL_TO_UDINT übergeben. Die Konvertierung erfolgt ohne Verzögerung innerhalb des Ereigniszyklus.
3. **Ausgangsereignis und Ergebnis** Nach erfolgreicher Konvertierung sendet F_REAL_TO_UDINT ein Bestätigungsereignis (CNF) an den Ausgangsadapter (AUDI_OUT.E1). Der konvertierte UDINT-Wert wird über den Datenausgang OUT an AUDI_OUT.D1 weitergeleitet.

Die Umsetzung ist synchron: Jedes eingehende Ereignis löst genau eine Ausgabe aus.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Zusammengesetzter Baustein** – Die Logik ist vollständig durch ein inneres Netzwerk realisiert; es existiert kein eigenständiger Algorithmus oder Zustandsautomat.
- **Unidirektionale Adapter** – Sowohl Eingangs- als auch Ausgangsschnittstelle sind unidirektional und übertragen nur einen Ereignis-/Datenkanal.
- **Verwendete Bibliothek** – Die Konvertierung erfolgt über den IEC-61131-Baustein F_REAL_TO_UDINT, der hardwareunabhängig und weit verbreitet ist.
- **Keine Fehlerbehandlung** – Der Baustein führt keine Bereichs- oder Typüberprüfung durch; die Konvertierung von REAL nach UDINT folgt den Standardregeln (Rundung, Begrenzung).

Zustandsübersicht

Da der Baustein keine eigene Zustandsmaschine besitzt, wird die Betriebsweise ausschließlich durch das Ereignisnetzwerk bestimmt:

- **Idle** – Kein Ereignis an AR_IN.E1 aktiv; Ausgangsadapter bleibt inaktiv.
- **Aktiv** – Eingehendes Ereignis löst sofort die Konvertierung aus und liefert ein Ausgangsereignis. Nach Abschluss kehrt der Baustein in den Ruhezustand zurück (kein interner Speicher).

Anwendungsszenarien

- **Brückenbildung** zwischen Systemteilen, die REAL-Werte liefern (z. B. Sensoren für Gleitkommazahlen) und Komponenten, die UDINT-Werte verarbeiten (z. B. Zähler, Indexverwaltung).

- **Adapter-basierte Kommunikation** in verteilten Automatisierungssystemen nach IEC 61499, wenn die Schnittstellen als unidirektionale Adapter definiert sind.
- **Typanpassung** in Datenvorverarbeitungspfaden, bevor Werte an speicherprogrammierbare Steuerungen (SPS) oder Visualisierungen übergeben werden.

Vergleich mit ähnlichen Bausteinen

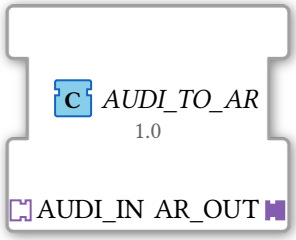
- **REAL_TO_DINT** – konvertiert REAL in vorzeichenbehaftete 32-Bit-Ganzzahl; hier wird stattdessen in vorzeichenlose Ganzzahl (UDINT) gewandelt.
- **AR_TO_xx-Bausteine** – Andere Varianten könnten z. B. in `AR_TO_BYTE` oder `AR_TO_DWORD` konvertieren, jedoch ohne den Adapter-Rahmen.
- **Direkter Konverter** – Der interne Baustein `F_REAL_TO_UDINT` lässt sich auch direkt ohne Adapter einbinden; `AR_TO_AUDI` bietet eine gekapselte, leicht austauschbare Schnittstelle.

Fazit

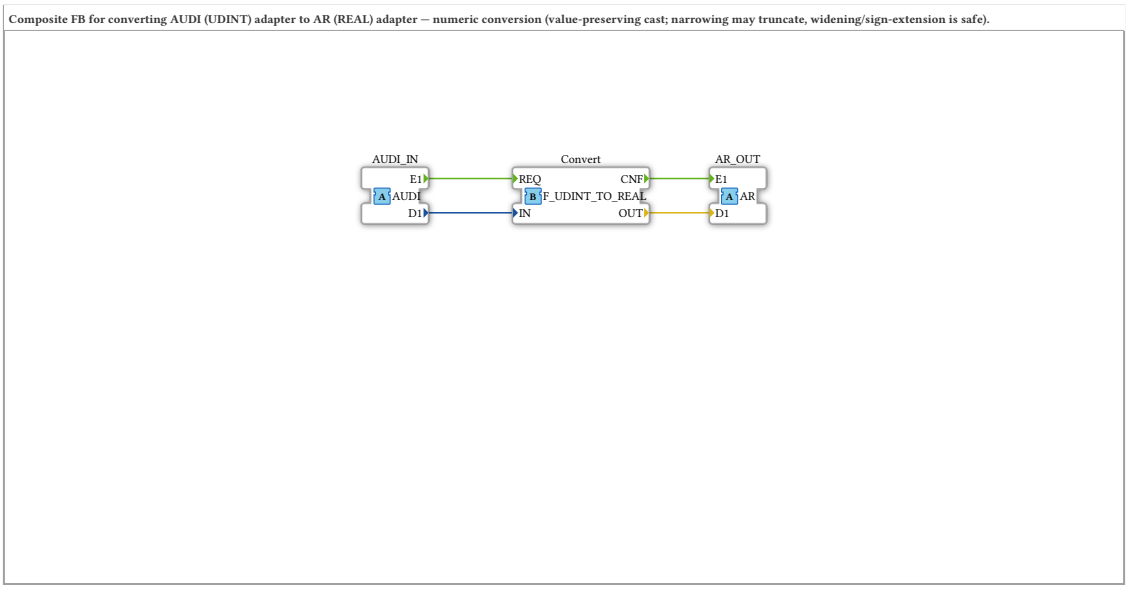
Der Baustein `AR_TO_AUDI` stellt eine sauber gekapselte, adapterbasierte Typkonvertierung von REAL nach UDINT bereit. Er reduziert den Verdrahtungsaufwand in modularen IEC-61499-Projekten und gewährleistet eine zuverlässige, ereignisgesteuerte Umsetzung. Durch die Verwendung standardisierter Konvertierungsbausteine ist die Lösung portabel und leicht verständlich.

AUDI_TO_AR

UDINT Adapter Input – *AUDI*



AR – REAL Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `AUDI_TO_AR` ist ein zusammengesetzter Baustein (Composite FB), der eine UDINT-große Ganzzahl, die über einen unidirektionalen AUDI-Adapter empfangen wird, in einen REAL-Wert umwandelt und über einen unidirektionalen AR-Adapter ausgibt. Er kapselt die Konvertierungslogik und ermöglicht so eine saubere Schnittstellenanpassung zwischen unterschiedlichen Adaptertypen.

Schnittstellenstruktur

Dieser FB besitzt keine direkten Ereignis- oder Datenein-/ausgänge. Die gesamte Kommunikation erfolgt ausschließlich über Adapter-Schnittstellen.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Name	Typ	Richtung	Beschreibung
`AUDI_IN`	`Adapter::types::unidirectional::AUDS`	Socket (Eingang)	Liefert einen UDINT-Wert sowie ein zugehöriges Ereignis. Der Adapter stellt intern einen Ereignisausgang `E1` und einen Datenausgang `D1` (Datentyp UDINT) bereit.
`AR_OUT`	`adapter::types::unidirectional::AR`	Plug (Ausgang)	Gibt den konvertierten REAL-Wert aus. Der Adapter besitzt intern einen Ereigniseingang `E1` und einen Dateneingang `D1` (Datentyp REAL).

Funktionsweise

1. Ein externes Signal triggert das Ereignis `AUDI_IN.E1`.
2. Dieses Ereignis wird an den internen Konvertierungsbaustein `F_UDINT_TO_REAL` (Typ `iec61131::conversion::F_UDINT_TO_REAL`) weitergeleitet (`Convert.REQ`).
3. Gleichzeitig wird der UDINT-Wert von `AUDI_IN.D1` an den Dateneingang `Convert.IN` übergeben.
4. Der Konvertierungsbaustein führt die Umwandlung von UDINT in REAL durch.
5. Nach Abschluss löst er das Ereignis `Convert.CNF` aus, welches an den Ereigniseingang `AR_OUT.E1` weitergegeben wird.
6. Der konvertierte REAL-Wert wird von `Convert.OUT` auf den Dateneingang `AR_OUT.D1` geschrieben und steht damit am AR-Adapter zur Verfügung.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- Die Umwandlung nutzt einen IEC-61131-3-Standardbaustein (`F_UDINT_TO_REAL`), der innerhalb des Composite FB instanziiert wird.
- Die Adapter sind als unidirektionale Schnittstellen ausgeführt, was eine klare Datenflussrichtung (vom AUDI- zum AR-Adapter) erzwingt.
- Der Composite FB erzeugt keinerlei eigene Zustandslogik, sondern delegiert die gesamte Verarbeitung an den internen Konvertierungsbaustein.

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Seine Funktionalität folgt dem einfachen Ablauf:

- **Warten auf Eingangsereignis** – solange `AUDI_IN.E1` nicht aktiv ist, bleibt der FB passiv.
- **Konvertierung** – nach Eintreffen von `AUDI_IN.E1` wird der integrierte `F_UDINT_TO_REAL` ausgeführt.

- **Ausgabe** – nach erfolgreicher Konvertierung wird `AR_OUT.E1` ausgelöst und der REAL-Wert bereitgestellt.

Anwendungsszenarien

- Anbindung eines UDINT-basierten Zählers (z. B. Impulszähler) an eine Steuerung, die REAL-Werte für Berechnungen oder Visualisierungen erwartet.
- Umwandlung von Adaptersignalen zwischen unterschiedlichen Datentypen in einer modularen, adapterbasierten Kommunikationsarchitektur.
- Als Baustein in einer Bibliothek für typsichere Adapter-Konvertierungen.

Vergleich mit ähnlichen Bausteinen

Dieser FB ist speziell auf die Kombination von AUDI- (UDINT) und AR- (REAL) Adaptern zugeschnitten. Vergleichbare Bausteine existieren für andere Datentypen, z. B.:

- `SINT_TO_AR` (SINT ⟶ REAL)
- `AUDI_TO_AI` (UDINT ⟶ INT)

Der Vorteil dieses Bausteins liegt in der klaren Kapselung und der Wiederverwendbarkeit innerhalb eines adapterbasierten Systemdesigns.

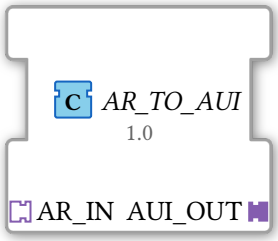
Fazit

Der `AUDI_TO_AR`-Funktionsblock bietet eine einfache, aber effektive Möglichkeit, UDINT-Daten über Adapterschnittstellen in REAL-Daten zu konvertieren. Durch die Verwendung eines IEC-Standardbausteins und die rein adapterbasierte Kommunikation ist er leicht verständlich, wartbar und in vielen Automatisierungsprojekten einsetzbar.

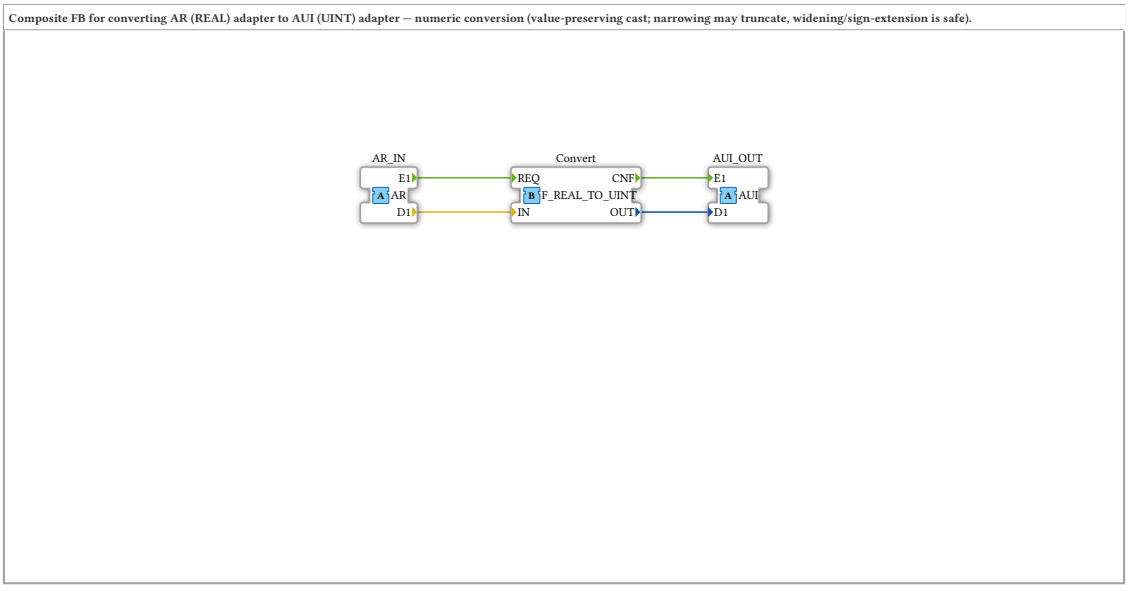
AR_AUI

AR_TO_AUI

REAL Adapter Input – *AR*



AUI – UINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AR_TO_AUI** ist ein zusammengesetzter Baustein (Composite FB), der eine einheitliche und bidirektionale Konvertierung von einem AR-Adapter (REAL) zu einem AUI-Adapter (UINT) realisiert. Er kapselt die Typumwandlung und vereinfacht die Kommunikation zwischen Komponenten, die unterschiedliche Datentypen über Adapter verwenden.

Schnittstellenstruktur

Ereignis-Eingänge

- **AR_IN.E1** – Trigger für die Konvertierung: Dieses Ereignis wird vom angeschlossenen AR-Adapter empfangen und startet die Umwandlung des REAL-Wertes.

Ereignis-Ausgänge

- **AUI_OUT.E1** – Abschluss der Konvertierung: Nach erfolgreicher Umwandlung wird dieses Ereignis an den AUI-Adapter gesendet.

Daten-Eingänge

- **AR_IN.D1** – REAL-Eingangswert: Der zu konvertierende Gleitkommawert (REAL), der über den AR-Adapter bereitgestellt wird.

Daten-Ausgänge

- **AUI_OUT.D1** – UINT-Ausgangswert: Das Ergebnis der Konvertierung (UINT) wird über diesen Datenausgang an den AUI-Adapter weitergegeben.

Adapter

- **AR_IN** – Socket vom Typ `AR` (unidirektionaler REAL-Adapter)
- **AUI_OUT** – Plug vom Typ `AUI` (unidirektionaler UINT-Adapter)

Die Adapter fassen jeweils ein Ereignis- und ein Datensignal zusammen, sodass der Baustein eine saubere, typsichere Schnittstelle bietet.

Funktionsweise

1. Ein eingehendes Ereignis auf **AR_IN.E1** triggert die Verarbeitung.
2. Der aktuelle Wert von **AR_IN.D1** (REAL) wird an den integrierten Konvertierungsbaustein **F_REAL_TO_UINT** übergeben.
3. Die Konvertierung wandelt den REAL-Wert in einen vorzeichenlosen Ganzzahlwert (UINT) um – gemäß IEC 61131-3 Rundungsregeln.
4. Nach Abschluss wird das Ergebnis auf **AUI_OUT.D1** ausgegeben und gleichzeitig das Ereignis **AUI_OUT.E1** gesendet.

Der gesamte Ablauf erfolgt ohne Zwischenspeicherung oder Verzögerung – jeder Impuls löst eine sofortige Umwandlung aus.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Stateless Design**: Der Baustein besitzt keinen internen Zustandsspeicher, jede Konvertierung ist unabhängig von vorherigen.
- **Einheitliche Adatperschnittstelle**: Durch die Verwendung von unidirektionalen Standardadaptern (AR und AUI) ist der FB leicht in verschiedene Architekturen integrierbar.
- **Keine Nebenwirkungen**: Es werden keine globalen Variablen oder Systemressourcen beeinflusst.

Zustandsübersicht

Der AR_TO_AUI ist ein reiner Funktionsbaustein ohne internen Zustandsautomaten (ECC). Die Abarbeitung erfolgt streng ereignisgesteuert und deterministisch. Somit entfällt eine separate Zustandsmaschine.

Anwendungsszenarien

- **Anbindung analoger Sensorik**: Wenn ein Sensor seinen Messwert als REAL über einen AR-Adapter liefert, das nachgeschaltete Steuerungssystem jedoch UINT erwartet.

- **Datenbrücke**: Vermittlung zwischen verschiedenen Kommunikationsmodulen oder Bibliotheken mit unterschiedlichen Adaptertypen.
- **Protokollumsetzung**: Beispielsweise in der Agrartechnik, wo oft REAL-Werte (z. B. Geschwindigkeit) in Ganzzahlen (z. B. für Zähler) umgerechnet werden müssen.

Vergleich mit ähnlichen Bausteinen

- **AR_TO_ADI**: Wandelt REAL in INT (vorzeichenbehaftet) um. AR_TO_AUI hingegen erzeugt vorzeichenlose Werte (UINT), was für bestimmte Anwendungen wie Zählerstände oder HMI-Indizes vorteilhaft ist.
- **Direkte Konvertierung mit F_REAL_TO_UINT**: Dieser einzelne Funktionsbaustein bietet die gleiche Logik, erfordert aber die manuelle Verbindung von Ereignissen und Daten. Der Composite FB kapselt diesen Aufwand und reduziert die Fehleranfälligkeit in komplexen Netzwerken.

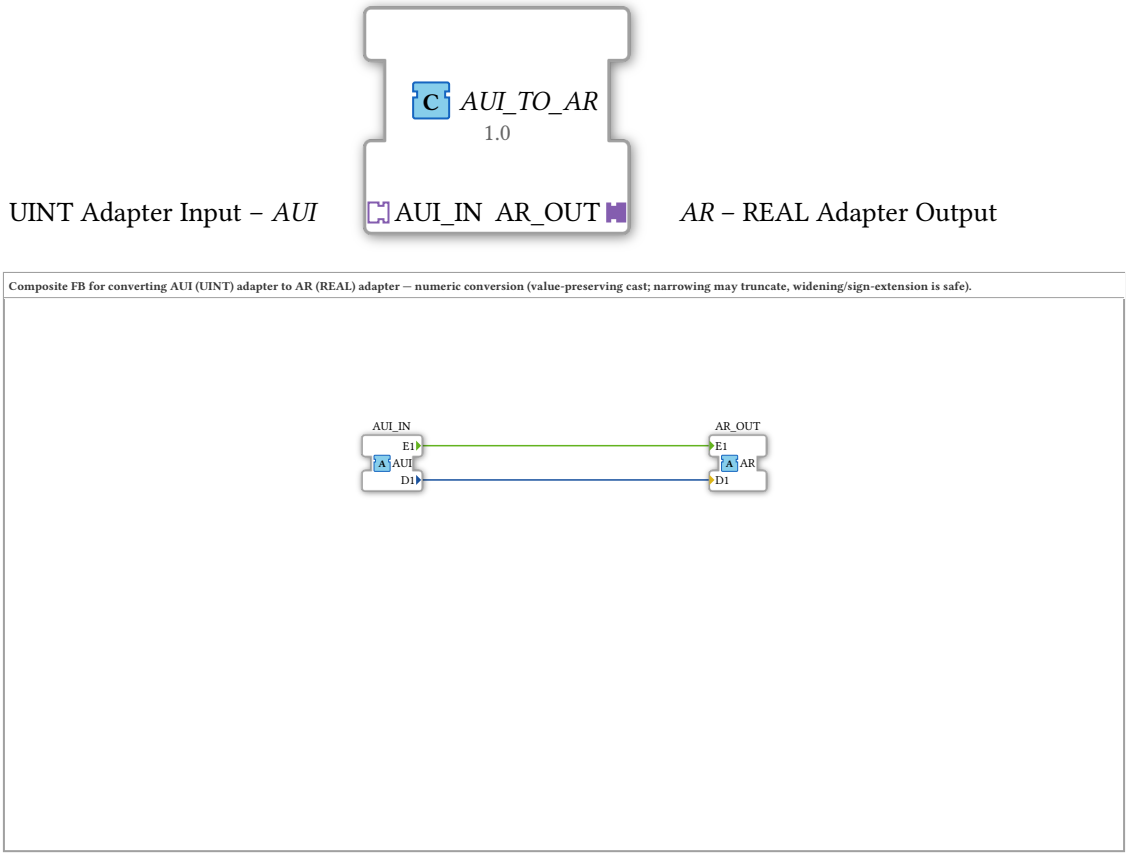
Fazit

Der **AR_TO_AUI** ist ein praktischer und zuverlässiger Baustein zur einfachen Adapterkonvertierung von REAL auf UINT. Seine klare Schnittstelle, das rein funktionale Verhalten und die Wiederverwendbarkeit machen ihn zu einem nützlichen Werkzeug im 4diac-Ökosystem. Er eignet sich besonders für Anwendungen, in denen eine saubere Trennung von Adaptertypen und eine einfache Integration in bestehende Projekte gefordert ist.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUI_TO_AR



UINT Adapter Input – *AUI* — *AR* – REAL Adapter Output

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUI_TO_AR** ist ein Composite-Baustein, der einen AUI-Adapter (UINT) in einen AR-Adapter (REAL) umwandelt. Er ermöglicht die nahtlose Integration eines unsigned Integer-Signals in Systeme, die einen REAL-Wert erwarten, indem er das Ereignis und den Datenwert 1:1 durchschleift.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keinen direkten Ereignis-Eingang, da die Ereignisse ausschließlich über den Input-Adapter **AUI_IN** bereitgestellt werden:

- **E1** (vom AUI_IN Adapter): Ereignis, das die Datenübertragung auslöst.

Ereignis-Ausgänge

Auch die Ereignis-Ausgänge werden über den Output-Adapter **AR_OUT** realisiert:

- **E1** (vom AR_OUT Adapter): Ereignis, das nach erfolgter Konvertierung ausgegeben wird.

Daten-Eingänge

Daten-Eingänge sind nur indirekt über den **AUI_IN**-Adapter vorhanden:

- **D1** (vom AUI_IN Adapter): zu konvertierender Wert als **UINT**.

Daten-Ausgänge

Daten-Ausgänge werden über den **AR_OUT**-Adapter zur Verfügung gestellt:

- **D1** (vom AR_OUT Adapter): konvertierter Wert als **REAL**.

Adapter

Bezeichnung	Typ	Richtung	Beschreibung
AUI_IN	`adapter::types::unidirectional::AUI`	Input (Socket)	Liefert ein UINT-Signal und ein zugehöriges Ereignis.
AR_OUT	`adapter::types::unidirectional::AR`	Output (Plug)	Gibt das konvertierte REAL-Signal und ein Ereignis aus.

Funktionsweise

Der Baustein **AUI_TO_AR** fungiert als reine Durchschleif-Komponente. Er verbindet 1:1:

- **AUI_IN.E1 → AR_OUT.E1**
- **AUI_IN.D1 → AR_OUT.D1**

Das bedeutet: Sobald ein Ereignis am AUI_IN-Eingang eintrifft, wird das zugehörige **UINT**-Datum ohne zusätzliche Verarbeitung oder Typumwandlung zum **AR_OUT**-Ausgang weitergeleitet. Die eigentliche Konvertierung von **UINT** nach **REAL** muss entweder durch den sendenden Adapter oder durch einen nachgeschalteten Baustein erfolgen – der **AUI_TO_AR** selbst führt keine arithmetische Wandlung durch. Er stellt lediglich sicher, dass die Ereignis- und Datenpfade zwischen den beiden unterschiedlichen Adapter-Typen kompatibel sind.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-Baustein:** Die Implementierung erfolgt rein über das Netzwerk aus Adapterverbindungen, ohne eigene Logik oder Zustandsautomaten.
- **Keine Typumwandlung:** Der **AUI_TO_AR** führt keine explizite Konvertierung des Datentyps durch. Er setzt voraus, dass der **AUI_IN**-Adapter bereits REAL-konforme Daten liefert oder dass die Typumwandlung extern erfolgt.
- **Unidirektionaler Adapter-Typ:** Beide Adapter sind unidirektional ausgelegt, d.h. sie übertragen Daten und Ereignisse nur in eine Richtung (von AUI_IN nach AR_OUT).

Zustandsübersicht

Da der Baustein als reines Verbindungsnetzwerk ohne eigenen Zustandsautomaten realisiert ist, existiert keine Zustandsmaschine. Die Funktionalität beschränkt sich auf die sofortige Weiterleitung eingehender Ereignisse und Daten.

Anwendungsszenarien

- **Systemintegration:** Einbindung von Sensoren oder Steuerungen, die ihre Messwerte als UINT über einen AUI-Adapter bereitstellen, in ein System, das REAL-Werte über AR-Adapter erwartet.
- **Protokollanpassung:** Wenn zwei Subsysteme unterschiedliche Adapter-Typen für die gleiche physikalische Größe verwenden, kann **AUI_TO_AR** als Adapter-Übersetzer dienen (vorausgesetzt, der UINT-Wert repräsentiert numerisch bereits den gewünschten REAL-Wert).
- **Erweiterung von Bibliotheken:** Als wiederverwendbares Baustein in einem Bibliotheksprojekt, um Adapter-Konvertierungen übersichtlich zu gestalten.

Vergleich mit ähnlichen Bausteinen

Baustein	Funktion	Unterschied
AUI_TO_AR	Adapter-Umschaltung UINT → REAL	Nur Durchschleif; keine Typumwandlung, erwartet kompatible Daten.
AR_TO_AUI	Adapter-Umschaltung REAL → UINT	Umgekehrte Richtung, ebenfalls reine Durchschleif.
CONV_UINT_TO_REAL	Typumwandlung von UINT nach REAL	Führt tatsächliche Konvertierung durch, benötigt aber direkte Daten-Ein-/Ausgänge statt Adapter.

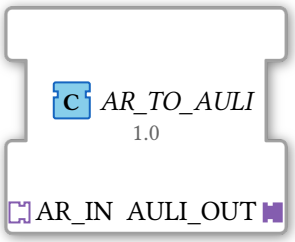
Fazit

Der **AUI_TO_AR** ist ein einfacher, aber nützlicher Composite-Baustein, der die Adapter-Typen **AUI** und **AR** miteinander verbindet, ohne die Daten zu verändern. Er eignet sich besonders für Systeme, in denen die Typkompatibilität bereits auf Adapter-Ebene gegeben ist oder eine nachgelagerte Konvertierung erfolgt. Seine schlanke Struktur macht ihn zu einem zuverlässigen Hilfsmittel für die Integration unterschiedlicher Adapter-Protokolle.

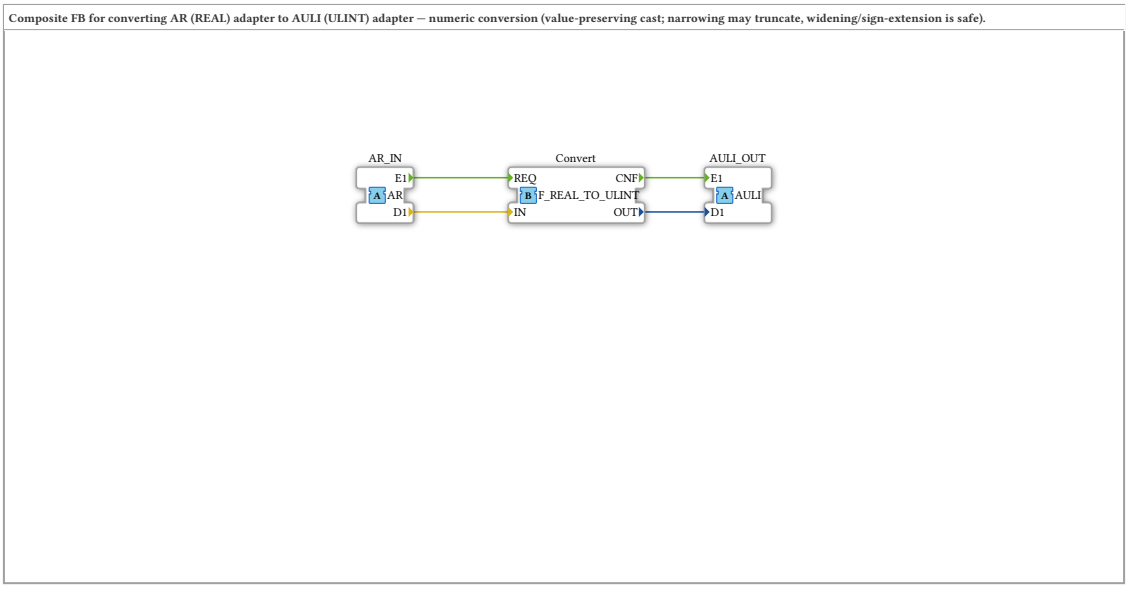
AR_AULI

AR_TO_AULI

REAL Adapter Input – *AR*



AULI – ULINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AR_TO_AULI** ist ein Composite-Baustein, der eine REAL-Eingangsadapter (AR) in eine ULINT-Ausgangsadapter (AULI) umwandelt. Er dient als unidirektionale Schnittstelle zwischen Komponenten, die mit dem Datentyp REAL arbeiten, und solchen, die ULINT (unsigned long integer) erwarten. Der Baustein kapselt die notwendige Konvertierung und stellt die Ereignis- und Datenweiterleitung automatisch sicher.

Schnittstellenstruktur

Ereignis-Eingänge

- **AR_IN.E1** – Eingangsereignis vom AR-Adapter; löst die Konvertierung aus.

Ereignis-Ausgänge

- **AULI_OUT.E1** – Ausgangsereignis an den AULI-Adapter; signalisiert, dass der konvertierte Wert bereitsteht.

Daten-Eingänge

- **AR_IN.D1** – REAL-Wert aus dem AR-Adapter, der konvertiert werden soll.

Daten-Ausgänge

- **AULI_OUT.D1** – ULINT-Wert, der aus dem REAL-Eingang konvertiert wurde.

Adapter

- **AR_IN** (Socket) – Unidirektionaler AR-Adapter (Typ: adapter::types::unidirectional::AR) als Eingang.
- **AULI_OUT** (Plug) – Unidirektionaler AULI-Adapter (Typ: adapter::types::unidirectional::AULI) als Ausgang.

Funktionsweise

Der Baustein ist als Composite FB realisiert und enthält intern einen einzigen Funktionsblock **F_REAL_TO_ULINT** aus der IEC-61131-Bibliothek. Die Arbeitsweise ist wie folgt:

1. Ein Ereignis am Socket **AR_IN.E1** löst die Ausführung des Konverters aus.
2. Der Datenwert **AR_IN.D1** wird an den Eingang **Convert.IN** weitergeleitet.
3. Der Konverter führt die Typumwandlung von REAL nach ULINT durch und stellt das Ergebnis am Ausgang **Convert.OUT** bereit.
4. Das Bestätigungsereignis **Convert.CNF** erzeugt ein Ereignis am Plug **AULI_OUT.E1**, das den angeschlossenen Baustein über den neuen Wert informiert.
5. Der konvertierte ULINT-Wert wird parallel auf **AULI_OUT.D1** ausgegeben.

Die Ereignis- und Datenverbindungen sind direkt und ohne zusätzliche Logik verdrahtet, sodass der Baustein eine reine Transformationshülle darstellt.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Unidirektionalität:** Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional ausgelegt. Es erfolgt keine Rückmeldung oder Bestätigung vom Ausgang zum Eingang.
- **Verwendeter Konverter:** Der eingebettete FB **F_REAL_TO_ULINT** implementiert die standardisierte IEC-61131-Konvertierung. Rundungs- und Überlaufverhalten entsprechen den IEC-Normvorgaben.
- **Ereignisgesteuerte Ausführung:** Die Konvertierung wird nur bei einem Ereignis am Eingang durchgeführt; ohne Ereignis bleibt der Ausgang unverändert.
- **Adapter-Typen:** Die Adapter basieren auf einer unidirektionalen Variante, d. h. sie übertragen genau ein Ereignis und einen Datenwert pro Richtung.

Zustandsübersicht

Als Composite FB besitzt **AR_TO_AULI** keine eigenen Zustände. Die interne Zustandslogik wird allein durch den eingebetteten **F_REAL_TO_ULINT** bestimmt, der in der Regel einen einfachen, ereignisgesteuerten Ablauf ohne Zustandsspeicherung realisiert (REQ → Verarbeitung → CNF). Der Baustein ist daher zustandslos und wartet nach jeder Konvertierung auf ein neues Eingangsereignis.

Anwendungsszenarien

- **Schnittstellenanpassung:** Wenn eine Komponente (z. B. ein Regelalgorithmus) REAL-Werte liefert, aber ein nachgeschaltetes Modul (z. B. eine Zähl- oder Indexlogik) ULINT-Eingänge erwartet.
- **Adapter-Bridge:** Verwendung in einer Tool-Chain, die unterschiedliche Datentypen über Adapter verbindet, ohne dass der Entwickler die Konvertierung manuell vornehmen muss.
- **Datenintegration:** Einbinden von REAL-basierten Sensoren oder Parametern in ein System, das intern mit ULINT arbeitet (z. B. für Status-Codes oder kumulierte Werte).

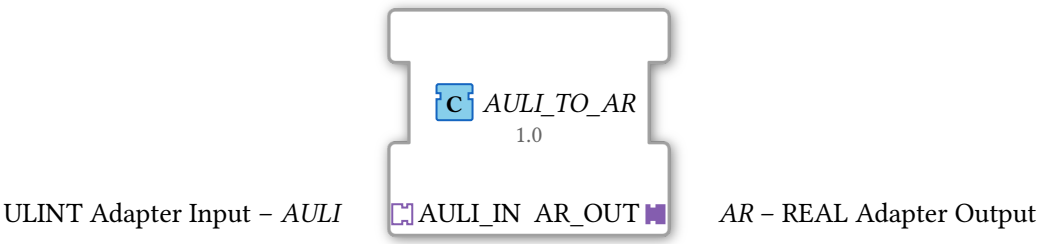
Vergleich mit ähnlichen Bausteinen

- **F_REAL_TO_ULINT** – Basisbaustein ohne Adapter; erfordert direkte Daten- und Ereignisverbindungen. **AR_TO_AULI** ist eine gekapselte Variante mit Adaptern, die die Einbindung in adapterbasierte Architekturen vereinfacht.
- **AR_TO_*-Bausteine für andere Zieltypen** – Es existieren entsprechende Composite-Bausteine für Konvertierungen von REAL nach LINT, DINT usw. Sie folgen dem gleichen Muster und nutzen jeweils den passenden IEC-Konverter.
- **Direkte Konvertierung im Netzwerk** – Alternativ könnte man den F_REAL_TO_ULINT direkt in ein FB-Netzwerk einbinden, was jedoch mehr manuelle Verdrahtung erfordert. **AR_TO_AULI** reduziert den Aufwand und verbessert die Wiederverwendbarkeit.

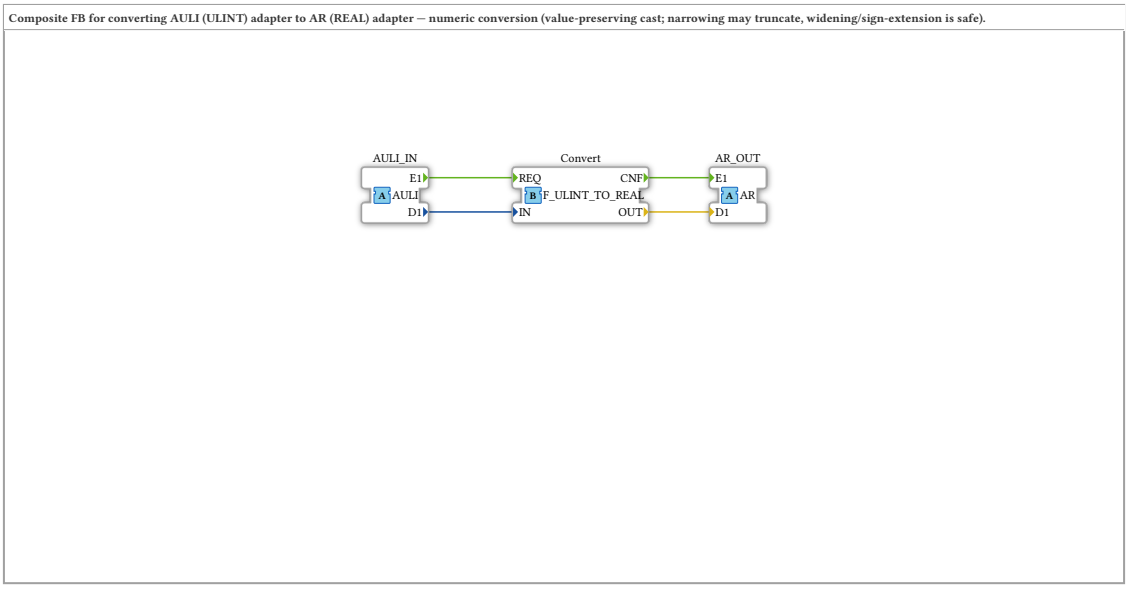
Fazit

Der **AR_TO_AULI**-Baustein ist eine einfache, aber nützliche Kapselung der REAL-nach-ULINT-Konvertierung in einem adapterbasierten Format. Er erweitert die IEC-61131-Konvertierung um eine plug-&-play-Schnittstelle für unidirektionale Adapter und ermöglicht eine saubere Trennung von Konvertierungslogik und Anwendungskommunikation. Der Baustein eignet sich besonders für modulare Systeme, die auf Adapterkonzepten aufbauen.

AULI_TO_AR



ULINT Adapter Input – *AULI* *AR* – REAL Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AULI_TO_AR** dient als konvertierender Composite-Baustein, der einen **AULI**-Adapter (ULINT) in einen **AR**-Adapter (REAL) umwandelt. Er ermöglicht die einfache Integration von Komponenten, die mit unterschiedlichen Datentypen arbeiten, und stellt die Typanpassung über standardisierte Adapter-Schnittstellen bereit.

Schnittstellenstruktur

Der FB besitzt keine direkten Ereignis- oder Datenein-/ausgänge, sondern kommuniziert ausschließlich über zwei Adapter-Schnittstellen.

Ereignis-Eingänge

- **AULI_IN.E1** (Ereignis vom Socket AULI_IN) – löst die Konvertierung des eingehenden ULINT-Wertes aus.

Ereignis-Ausgänge

- **AR_OUT.E1** (Ereignis zum Plug AR_OUT) – signalisiert, dass der konvertierte REAL-Wert bereitsteht.

Daten-Eingänge

- **AULI_IN.D1** (Daten vom Socket AULI_IN) – der zu konvertierende ULINT-Wert.

Daten-Ausgänge

- **AR_OUT.D1** (Daten zum Plug AR_OUT) – der konvertierte REAL-Wert.

Adapter

Name	Richtung	Typ	Beschreibung
AULI_IN	Socket	`adapter::types::unidirectional::AULI` (ULINT)	Eingangswert
AR_OUT	Plug	`adapter::types::unidirectional::AR` (REAL)	Ausgangswert (konvertiert)

Funktionsweise

Der FB ist als Composite-Baustein realisiert und enthält eine Instanz des Konvertierungs-FBs `F_ULINT_TO_REAL` aus der IEC-61131-Bibliothek.

Ablauf:

1. Ein Ereignis auf **AULI_IN.E1** triggert die Ausführung des internen Konvertierungsbausteins `Convert`.
2. Der aktuelle ULINT-Wert von **AULI_IN.D1** wird an den Eingang `Convert.IN` übergeben.
3. Der interne FB führt die Umwandlung von `ULINT` nach `REAL` durch und legt das Ergebnis an `Convert.OUT` ab.
4. Nach Abschluss der Konvertierung wird das Ereignis **AR_OUT.E1** ausgelöst, und der konvertierte REAL-Wert steht an **AR_OUT.D1** zur Verfügung.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-Struktur** – ermöglicht Wiederverwendung und Kapselung der Konvertierungslogik.
- **Adapter-basiert** – keine direkten Ein-/Ausgänge; die Kommunikation erfolgt über unidirektionale Adapter (Socket/Plug).
- **Verwendung der IEC-61131-Standardfunktion** `F_ULINT_TO_REAL` – garantiert plattformunabhängige und korrekte Typumwandlung.
- **Ereignisgesteuerte Ausführung** – die Konvertierung wird nur bei Bedarf durch ein Eingangsereignis angestoßen.
- **Lizenz** – Eclipse Public License 2.0 (siehe Quelltext).

Zustandsübersicht

Der FB besitzt **keine eigene Zustandsmaschine**. Die Konvertierung erfolgt rein funktional und ereignisgesteuert über den internen Baustein. Nach jedem Eingangsereignis wird einmalig umgewandelt und das Ergebnis ausgegeben.

Anwendungsszenarien

- **Adapter-Brücke** zwischen Komponenten, die ULINT liefern, und solchen, die REAL erwarten.
- **Datenaufbereitung** in Steuerungssystemen, wenn physikalische Größen (z. B. Zählerstände als ULINT) in Gleitkommawerte für Berechnungen umgewandelt werden müssen.
- **Modulare Vernetzung** in IEC-61499-Anwendungen, wo Adapter als standardisierte Schnittstellen zwischen Funktionsbausteinen dienen.

Vergleich mit ähnlichen Bausteinen

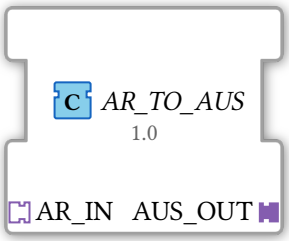
- **AULI_TO_LREAL** – analoge Konvertierung zu `LREAL` (Doppelgenauigkeit).
- **Direkte Konvertierungs-FBs** (z. B. `F_ULINT_TO_REAL`): Diese arbeiten auf Datenebene, erfordern aber manuelle Verdrahtung von Ereignissen und Daten. Der **AULI_TO_AR** kapselt diese Verdrahtung und bietet eine kompakte Adapter-Schnittstelle.
- **Andere Adapter-Konverter** – es existieren entsprechende Bausteine für andere Typkombinationen (z. B. `LINT_TO_AR`), die ähnlich aufgebaut sind.

Fazit

Der **AULI_TO_AR** ist ein nützlicher Composite-Baustein zur einfachen und standardisierten Typkonvertierung von ULINT nach REAL über Adapter-Schnittstellen. Dank der ereignisgesteuerten Ausführung und der Nutzung einer IEC-61131-Standardfunktion ist er zuverlässig, wiederverwendbar und fügt sich nahtlos in IEC-61499-Projekte ein.

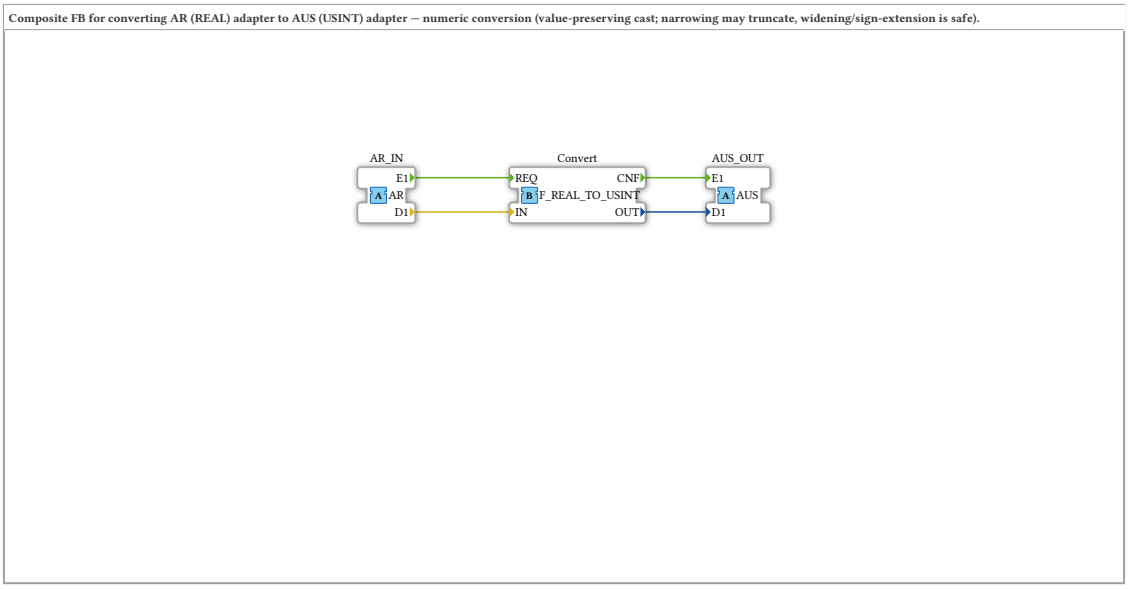
AR_AUS

AR_TO_AUS



REAL Adapter Input – *AR*

AUS – USINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock AR_TO_AUS ist ein Composite-Baustein zur Umwandlung eines REAL-Wertes in einen USINT-Wert über eine unidirektionale Adapterschnittstelle. Er kapselt den Konvertierungsprozess und stellt eine einheitliche Verbindung zwischen einem REAL-Adaptersocket und einem USINT-Adapterplug bereit. Der FB ist für den Einsatz in Automatisierungssystemen konzipiert, die eine Datentypanpassung erfordern.

Schnittstellenstruktur

Ereignis-Eingänge

- **AR_IN.E1** (über Socket `AR_IN`): Ereignis zum Starten der Konvertierung des anliegenden REAL-Wertes.

Ereignis-Ausgänge

- **AUS_OUT.E1** (über Plug `AUS_OUT`): Ereignis, das nach erfolgreicher Konvertierung und Ausgabe des USINT-Wertes ausgelöst wird.

Daten-Eingänge

- **AR_IN.D1** (über Socket `AR_IN`, Typ: REAL): Eingangswert, der in USINT konvertiert werden soll.

Daten-Ausgänge

- **AUS_OUT.D1** (über Plug `AUS_OUT`, Typ: USINT): Konvertierter Ausgangswert.

Adapter

- **Socket AR_IN** (Typ: `adapter::types::unidirectional::AR`): Unidirektionaler REAL-Eingangsadapter.
- **Plug AUS_OUT** (Typ: `adapter::types::unidirectional::AUS`): Unidirektionaler USINT-Ausgangsadapter.

Funktionsweise

Der FB AR_TO_AUS arbeitet als reiner Datenkonverter ohne eigene Zustandslogik. Intern wird der vordefinierte Baustein `F_REAL_TO_USINT` aus der IEC-61131-Standardbibliothek verwendet.

Ablauf:

1. Ein eingehendes Ereignis an `AR_IN.E1` triggert die Verarbeitung.
2. Der aktuelle Wert von `AR_IN.D1` (REAL) wird an den Eingang `IN` des internen Konverters übergeben.
3. Nach Abschluss der Konvertierung wird der Ergebniswert (USINT) an `AUS_OUT.D1` weitergeleitet.
4. Gleichzeitig wird das Ereignis `AUS_OUT.E1` ausgelöst, um nachgeschaltete Bausteine zu aktivieren.

Der gesamte Daten- und Ereignisfluss ist unidirektional vom Eingangs- zum Ausgangsadapter.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- Der FB ist als Composite implementiert und verwendet den Baustein `iec61131::conversion::F_REAL_TO_USINT` aus der Bibliothek `adapter::conversion::unidirectional`.
- Es findet keine Wertebereichsprüfung statt; die Konvertierung folgt den Regeln der IEC 61131 (REAL → USINT: Nachkommastellen werden abgeschnitten, Werte außerhalb des Bereichs 0…255 können zu undefiniertem Verhalten führen).
- Der FB besitzt keine eigene Zustandsmaschine und arbeitet ereignisgesteuert.
- Der Einsatz von Adaptern ermöglicht eine lose Kopplung der Schnittstellen und erleichtert die Wiederverwendung in verschiedenen Projektkontexten.

Zustandsübersicht

Der FB AR_TO_AUS besitzt keinen eigenen Zustandsautomaten. Das Laufzeitverhalten wird vollständig durch den internen Funktionsbaustein `F_REAL_TO_USINT` bestimmt, der standardmäßig zustandslos arbeitet. Die Ereignissteuerung erfolgt direkt von `AR_IN.E1` zu `AUS_OUT.E1` ohne Zwischenzustände.

Anwendungsszenarien

- **Sensor-Aktor-Kopplung**: Ein REAL-Wert (z. B. von einem analogen Sensor) muss als USINT an einen binären Aktor (z. B. Motorsteuerung mit 8-Bit-Wert) weitergegeben werden.
- **Schnittstellenanpassung**: Wenn ein Adapter vom Typ `AR` (REAL) mit einem Adapter vom Typ `AUS` (USINT) verbunden werden muss.
- **Datenvorverarbeitung**: Vor der Übergabe an Systeme, die nur USINT-Werte akzeptieren (z. B. Diagnose- oder Visualisierungskomponenten).

Vergleich mit ähnlichen Bausteinen

- **AR_TO_INT** / **AR_TO_BYTE**: Analoger Aufbau für die Konvertierung in andere Datentypen (INT, BYTE). Der vorliegende FB beschränkt sich auf die Wandlung von REAL in USINT.
- **Direkte Konverter-FBs** (z. B. `F_REAL_TO_USINT`): Diese bieten keine Adapterschnittstellen, sondern erfordern eine direkte Verdrahtung. AR_TO_AUS kapselt diesen Konverter und stellt eine ad-hoc nutzbare Adapterverbindung bereit.
- **Bidirektionale Konverter**: Im Gegensatz zu diesen arbeitet AR_TO_AUS strikt unidirektional, was die Klarheit des Datenflusses erhöht.

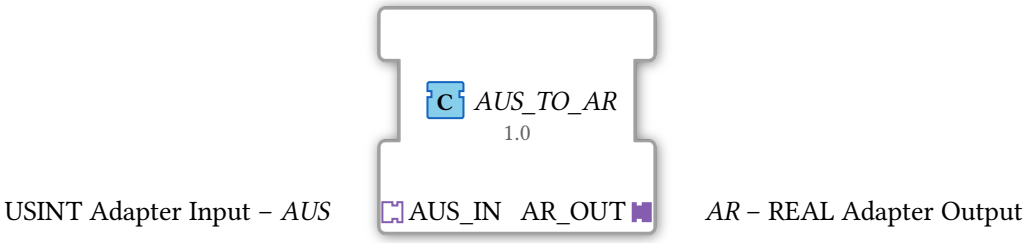
Fazit

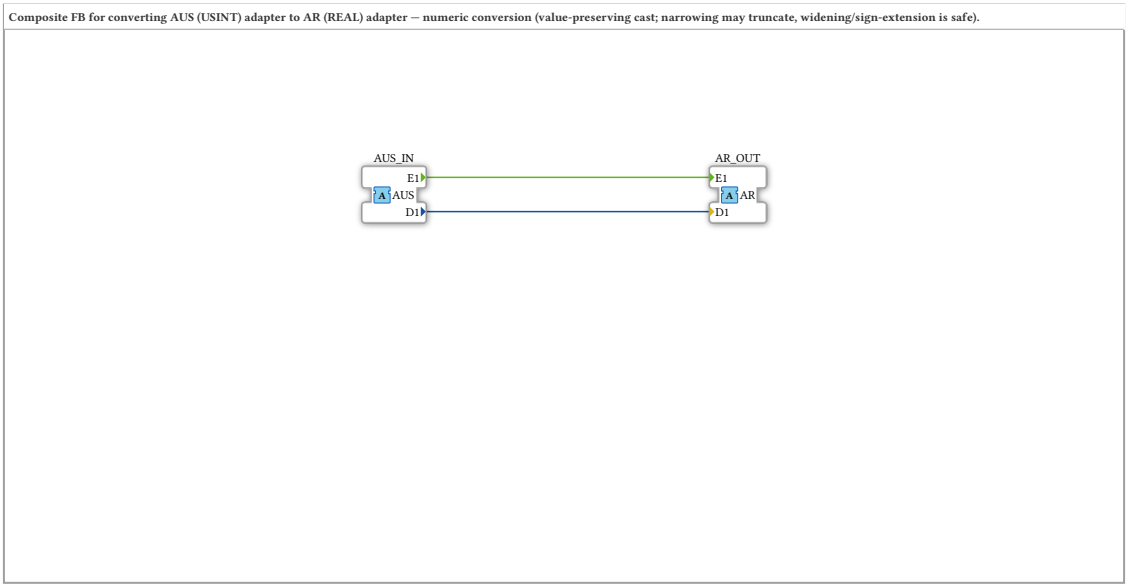
Der Funktionsblock AR_TO_AUS bietet eine einfache, modulare Lösung zur Konvertierung eines REAL-Adapters in einen USINT-Adapter. Durch die Verwendung eines standardisierten Konvertierungsbausteins und die Kapselung als Composite-FB kann er leicht in bestehende 4diac-Projekte integriert werden. Er eignet sich besonders für Szenarien, in denen eine klare Trennung von Schnittstellen und eine unidirektionale Datenübertragung erforderlich sind.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUS_TO_AR





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUS_TO_AR** ist ein Composite FB zur Umwandlung eines AUS-Adapter-Interfaces (basierend auf USINT) in ein AR-Adapter-Interface (basierend auf REAL). Laut Beschreibung dient er zur Konvertierung zwischen diesen beiden Adaptertypen. Die aktuelle Implementierung (Version 1.0) leitet Ereignis- und Datenwerte lediglich durch, ohne eine explizite Umrechnung vorzunehmen – die eigentliche Typwandlung muss entweder durch die Adapter-Definitionen selbst oder durch nachträgliche Erweiterungen realisiert werden.

Schnittstellenstruktur

Ereignis-Eingänge

Keine separaten Ereignis-Eingänge vorhanden. Ereignisse werden ausschließlich über den **AUS_IN** Adapter übertragen.

Ereignis-Ausgänge

Keine separaten Ereignis-Ausgänge vorhanden. Ereignisse werden ausschließlich über den **AR_OUT** Adapter ausgegeben.

Daten-Eingänge

Keine separaten Daten-Eingänge vorhanden. Daten werden ausschließlich über den **AUS_IN** Adapter übertragen.

Daten-Ausgänge

Keine separaten Daten-Ausgänge vorhanden. Daten werden ausschließlich über den **AR_OUT** Adapter ausgegeben.

Adapter

Typ	Name	Richtung	Kommentar
`adapter::types::unidirectional::AUS`	AUS_IN	Socket (Eingang)	USINT-basierter Adapter-Eingang
`adapter::types::unidirectional::AR`	AR_OUT	Plug (Ausgang)	REAL-basierter Adapter-Ausgang

Funktionsweise

Der Baustein verbindet die Ereignis- und Datenkanäle des AUS_IN-Adapters direkt mit denen des AR_OUT-Adapters:

- **Ereignisdurchleitung**: Das am Socket `AUS_IN.E1` ankommende Ereignis wird unverzögert an den Plug `AR_OUT.E1` weitergeleitet.
- **Datendurchleitung**: Der vom Socket `AUS_IN.D1` bereitgestellte Datenwert (Typ USINT) wird direkt an `AR_OUT.D1` (Typ REAL) übergeben.

Da keine Typkonvertierung oder sonstige Verarbeitung im Netzwerk abgebildet ist, erfolgt die Umwandlung von USINT nach REAL entweder implizit durch die verwendete 4diac-Laufzeitumgebung oder muss durch Hinzufügen eines Konvertierungsbausteins (z. B. `CONV`) innerhalb einer Erweiterung des Composite-FBs realisiert werden.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite ohne innere Logik**: Das FBNetzwerk besteht ausschließlich aus zwei direkten Verbindungen; es sind keine weiteren Funktionsblöcke oder Zustandsautomaten enthalten.
- **Adapterbasierte Schnittstelle**: Statt einzelner Ereignis-/Dateneingänge und -ausgänge wird die gesamte Kommunikation über standardisierte unidirektionale Adapter abgewickelt. Dies erleichtert die Wiederverwendung in modularen Adapter-Hierarchien.
- **Unvollständige Implementierung**: Der Name und der Kommentar des Bausteins suggerieren eine aktive Konvertierung, die in der vorliegenden Version (1.0) jedoch nicht umgesetzt ist. Dies kann zu Laufzeitfehlern führen, wenn die Laufzeitumgebung keine implizite Typumwandlung unterstützt.

Zustandsübersicht

Der Baustein besitzt keine Zustandsautomaten (ECC). Sein Verhalten ist rein kombinatorisch und auf die sofortige Durchleitung von Ereignissen und Daten beschränkt.

Anwendungsszenarien

- **Ersatz für ältere Systeme**: Ein bestehendes AUS-Adapter-Interface (USINT) muss an eine Komponente angeschlossen werden, die ein AR-Adapter-Interface (REAL) erwartet.
- **Platzhalter in der Entwicklungsphase**: Der FB dient als vorläufige Verbindung, bis eine eigentliche Konvertierungslogik implementiert wird.
- **Testumgebungen**: Schnelle Kopplung zweier Adaptertypen ohne Datenänderung für Simulationen oder Tests.

Vergleich mit ähnlichen Bausteinen

Typische Konvertierungsbausteine (z. B. `CONV` aus der Standardbibliothek) führen eine explizite Wertumrechnung mittels arithmetischer oder logischer Operationen durch. `AUS_TO_AR` hingegen delegiert die Umwandlung an die Adapterebene und übernimmt selbst keine Transformation. Es handelt sich eher um eine **strukturelle Anpassung** als um einen echten Konverter.

Fazit

Der Funktionsblock `AUS_TO_AR` ist ein einfacher Adapter-zu-Adapter-Through-Pass-Baustein. Seine Nützlichkeit hängt stark davon ab, ob das Zielsystem automatische Typkonvertierungen zwischen USINT und REAL unterstützt. Ist dies nicht der Fall, muss der FB um eine explizite Konvertierungslogik erweitert

werden. Er eignet sich daher als temporäre Lösung oder als Grundgerüst für eine vollständige Implementierung.

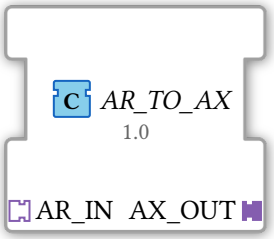
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

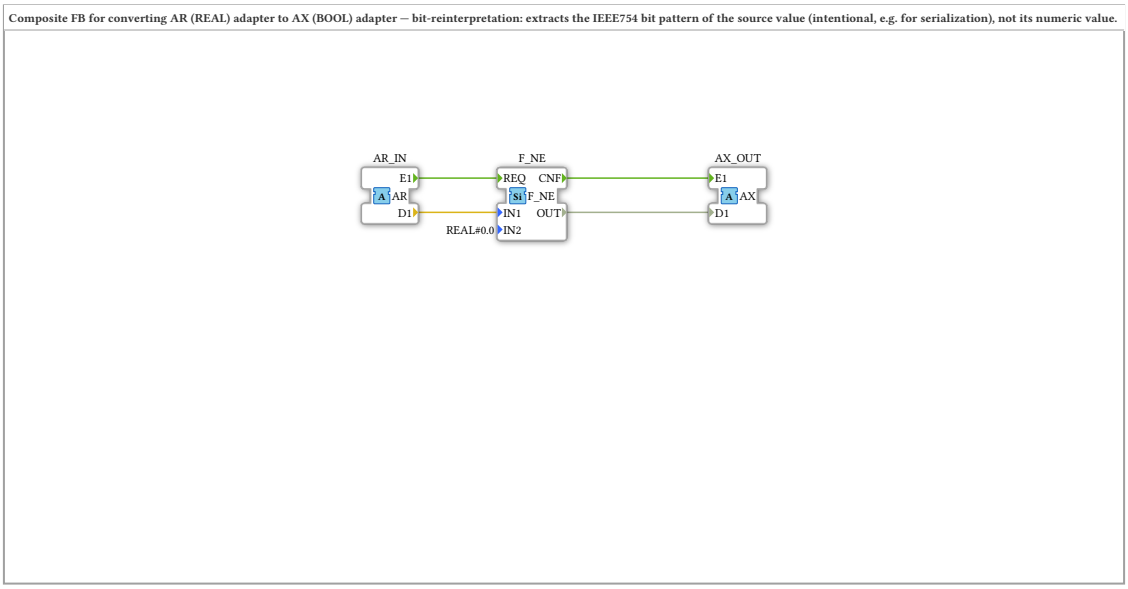
AR_AX

AR_TO_AX

REAL Adapter Input – *AR*



AX – BOOL Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AR_TO_AX wandelt einen REAL-Wert, der über einen AR-Adapter empfangen wird, in einen BOOL-Wert um und gibt diesen über einen AX-Adapter aus. Die Umwandlung erfolgt durch einen Vergleich des REAL-Werts mit 0,0. Ist der Wert ungleich Null, wird TRUE ausgegeben, ansonsten FALSE. Der Baustein dient als Brücke zwischen analogen und digitalen Adapterschnittstellen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge auf FB-Ebene. Der Ereignis-Eingang ist im AR-Adapter (`AR_IN.E1`) enthalten.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge auf FB-Ebene. Der Ereignis-Ausgang ist im AX-Adapter (`AX_OUT.E1`) enthalten.

Daten-Eingänge

Keine direkten Daten-Eingänge auf FB-Ebene. Der Daten-Eingang ist im AR-Adapter (`AR_IN.D1`) enthalten.

Daten-Ausgänge

Keine direkten Daten-Ausgänge auf FB-Ebene. Der Daten-Ausgang ist im AX-Adapter (`AX_OUT.D1`) enthalten.

Adapter

- **AR_IN** (Socket) – Adapter vom Typ `adapter::types::unidirectional::AR`. Nimmt einen REAL-Wert sowie ein Ereignis zum Start der Verarbeitung entgegen.
- **AX_OUT** (Plug) – Adapter vom Typ `adapter::types::unidirectional::AX`. Stellt den BOOL-Ergebniswert sowie ein Ereignis zur Bestätigung der Verarbeitung bereit.

Funktionsweise

Der Baustein verarbeitet eingehende REAL-Werte folgendermaßen:

1. Ein Ereignis an `AR_IN.E1` triggert den internen Vergleichsbaustein `F_NE` (ungleich).
2. Der über `AR_IN.D1` anliegende REAL-Wert wird mit dem konstanten Wert 0,0 verglichen.
3. Das Ergebnis des Vergleichs (BOOL) wird über `F_NE.OUT` an `AX_OUT.D1` weitergeleitet.
4. Ein Ereignis am Ausgang `F_NE.CNF` wird an `AX_OUT.E1` gesendet, um die Verfügbarkeit des BOOL-Werts zu signalisieren.

Der interne Baustein `F_NE` ist ein IEC 61131-3-konformer Vergleichsbaustein und gibt TRUE aus, wenn der Eingang `IN1` ungleich `IN2` (hier 0,0) ist.

Technische Besonderheiten

- **Bit-Reinterpretation**: extrahiert das IEEE754-Bitmuster des Quellwerts (beabsichtigt, z. B. zur Serialisierung), nicht dessen Zahlenwert.
- Verwendet einen Composite-FB-Ansatz, bei dem die Logik aus einem einzigen Vergleichsbaustein und Adapterverbindungen besteht.
- Alle Ein- und Ausgänge sind über standardisierte unidirektionale Adapter (`AR`, `AX`) realisiert, was eine modulare und austauschbare Schnittstellendefinition ermöglicht.
- Der Vergleichswert (0,0) ist fest im FB eingestellt und kann nicht zur Laufzeit geändert werden.
- Die Ausführung erfolgt ereignisgesteuert: Nur bei einem Ereignis an `AR_IN.E1` wird eine Umwandlung durchgeführt.

Zustandsübersicht

Da der FB als Composition (Netzwerk aus Basisbausteinen) implementiert ist, besitzt er keine eigene Zustandsmaschine. Die Funktion wird durch die sequenzielle Verarbeitung der Ereignis- und Datenflüsse im internen Netzwerk bestimmt:

- **Wartezustand**: Kein Ereignis an `AR_IN.E1`. Der Ausgangswert bleibt unverändert.
- **Verarbeitungszustand**: Ein Ereignis an `AR_IN.E1` löst den Vergleich aus und erzeugt ein Ereignis an `AX_OUT.E1`, sobald der BOOL-Wert bereitsteht.

Anwendungsszenarien

- **Schwellwertschalter**: Ein analoger Sensor (z.B. Drucksensor) liefert REAL-Werte. Der FB gibt TRUE aus, sobald der Wert von 0,0 abweicht (z.B. bei Überschreitung eines Schwellwerts, wenn der Nullpunkt kalibriert ist).
- **Digitalisierung von Analogsignalen**: Zur Anbindung analoger Adapter an digitale Logik in der Steuerungstechnik.
- **Einfache Plausibilitätsprüfung**: Prüfen, ob ein REAL-Wert ungleich Null (z.B. gültiger Messwert) vorliegt.

Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu einem generischen `AR_TO_BOOL`-Baustein, der direkte Daten ohne Adapter verwendet, ist AR_TO_AX speziell für die Verwendung mit Adapterschnittstellen ausgelegt. Ein Vergleichsbaustein mit einstellbarem Schwellwert (z.B. `F_GT`, `F_LT`) würde mehr Flexibilität bieten, aber auch zusätzlichen Konfigurationsaufwand erfordern. Der vorliegende Baustein bietet eine einfache, feste Umwandlung: ungleich Null ⟶ TRUE.

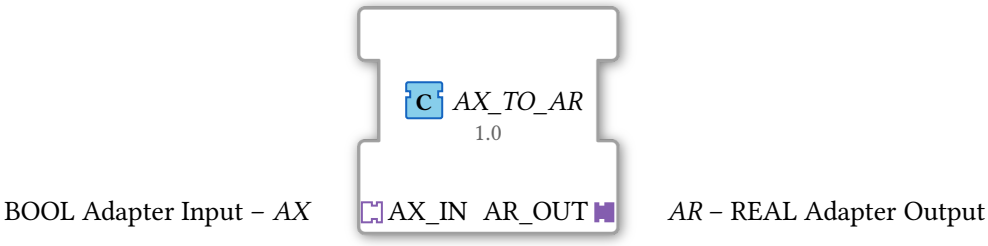
Fazit

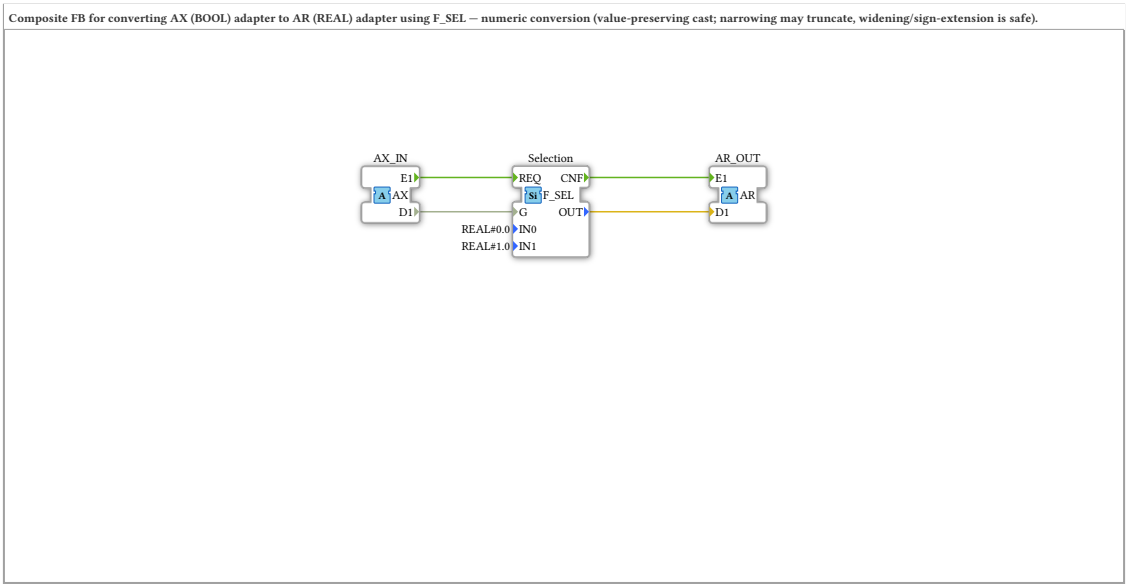
Der AR_TO_AX-Funktionsblock bietet eine kompakte Lösung zur Umwandlung eines REAL-Adapter-Werts in einen BOOL-Adapter-Wert mittels Nullvergleich. Durch die Verwendung von Adaptern und einem standardisierten Vergleichsbaustein ist er einfach in bestehende 4diac-Projekte integrierbar und eignet sich gut für schnelle, unidirektionale Signalkonvertierungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AX_TO_AR





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AX_TO_AR** ist ein Composite-Funktionsbaustein, der ein boolesches Signal aus einem AX-Adapter (Typ `adapter::types::unidirectional::AX`) in einen REAL-Wert (0.0 oder 1.0) umwandelt und über einen AR-Adapter (Typ `adapter::types::unidirectional::AR`) ausgibt. Die Konvertierung erfolgt mithilfe des IEC-61131-Bausteins **F_SEL**.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt auf FB-Ebene keine eigenen Ereignis-Eingänge. Der interne Ablauf wird ausschließlich über das Ereignis des AX_IN-Adapters gesteuert, das mit dem REQ-Eingang des F_SEL verbunden ist.

Ereignis-Ausgänge

Auf FB-Ebene sind keine Ereignis-Ausgänge vorhanden. Das Ausgangsereignis des F_SEL (CNF) wird direkt an den Ereignis-Eingang des AR_OUT-Adapters weitergeleitet.

Daten-Eingänge

Es existieren keine direkten Daten-Eingänge auf FB-Ebene. Der einzige Daten-Eingang wird über den AX_IN-Adapter bereitgestellt:

- **D1** (BOOL) – das zu konvertierende boolesche Signal.

Daten-Ausgänge

Es gibt keine direkten Daten-Ausgänge auf FB-Ebene. Der konvertierte Wert wird über den AR_OUT-Adapter ausgegeben:

- **D1** (REAL) – der resultierende REAL-Wert (0.0 oder 1.0).

Adapter

- **AX_IN** (Socket):
  - Typ: `adapter::types::unidirectional::AX`
  - Liefert ein boolesches Signal und einen Ereignisimpuls.
- **AR_OUT** (Plug):
  - Typ: `adapter::types::unidirectional::AR`
  - Gibt den umgewandelten REAL-Wert aus.

Funktionsweise

Der Baustein arbeitet ereignisgesteuert:

1. Ein eingehendes Ereignis am AX_IN-Adapter (E1) triggert den internen F_SEL-Baustein über dessen REQ-Eingang.
2. Gleichzeitig wird das boolesche Signal (AX_IN.D1) an den G-Eingang des F_SEL angelegt.
3. F_SEL wählt abhängig von G:
   - Bei `G = FALSE` => Ausgabe von **IN0** (Wert `REAL#0.0`).
   - Bei `G = TRUE` => Ausgabe von **IN1** (Wert `REAL#1.0`).
4. Das Ergebnis wird über den Ausgang OUT an den Daten-Eingang D1 des AR_OUT-Adapters weitergegeben.
5. Das Ereignis CNF des F_SEL löst das Ausgangsereignis E1 des AR_OUT-Adapters aus, sodass die angeschlossene Logik die neuen Daten übernehmen kann.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Wiederverwendung**: Der Baustein ist als Composite-FB realisiert und nutzt den standardisierten IEC-61131-Baustein `F_SEL` aus der Bibliothek `iec61131::selection`.
- **Adapter-Kopplung**: Die gesamte Kommunikation erfolgt über unidirektionale Adapter (`adapter::types::unidirectional::AX` und `adapter::types::unidirectional::AR`), was eine lose Kopplung und einfache Wiederverwendung in verschiedenen Kontexten ermöglicht.
- **Konfiguration**: Die Grenzwerte (IN0=0.0, IN1=1.0) sind fest vorgegeben und können bei Bedarf durch Anpassung der Parameter des internen F_SEL geändert werden.

Zustandsübersicht

Der Baustein besitzt keinen internen Zustandsautomaten. Er reagiert **ereignisgesteuert** und führt bei jedem Ereignis am AX_IN-Adapter eine einmalige Selektion durch. Es gibt keine speichernden Zustände – das Verhalten ist rein kombinatorisch.

Anwendungsszenarien

- Umwandlung eines binären Sensorsignals (z. B. Schalter, Grenzwertmelder) in einen numerischen Wert für die Weiterverarbeitung in REAL-basierten Regelkreisen oder Visualisierungen.
- Integration boolescher Adapter-Signale in Systeme, die durchgängig mit REAL-Werten arbeiten.
- Einfaches Mapping von digitalen Ein-/Ausgängen auf analoge Werte (0/1) in adapterbasierten Automatisierungslösungen.

Vergleich mit ähnlichen Bausteinen

- **Standard-Konvertierungsbausteine** wie `BOOL_TO_REAL` aus IEC-Bibliotheken bieten eine direkte Umwandlung auf Datenebene, jedoch ohne Adapter-Schnittstellen.
- **`SELECT`** (IEC 61131) kann ebenfalls ein boolesches Signal zur Auswahl zwischen zwei REAL-Werten nutzen, erfordert aber manuelle Verdrahtung.
- **AX_TO_AR** kapselt die komplette Adapter-Konvertierung in einem wiederverwendbaren Composite-FB und reduziert so den Verdrahtungsaufwand in adapternetzwerkbasierten 4diac-Anwendungen deutlich.

Fazit

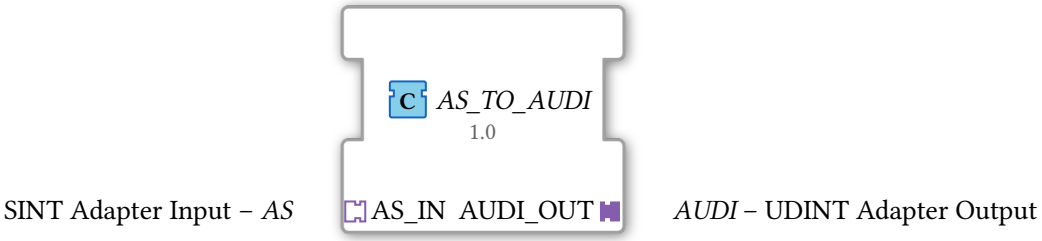
Der Baustein **AX_TO_AR** stellt eine kompakte und robuste Lösung zur Konvertierung boolescher Adapter-Signale in REAL-Werte dar. Durch die Nutzung von `F_SEL` und die Kapselung als Composite-FB ist er einfach zu integrieren, erweiterbar und ideal für adapterorientierte Automatisierungsarchitekturen geeignet.

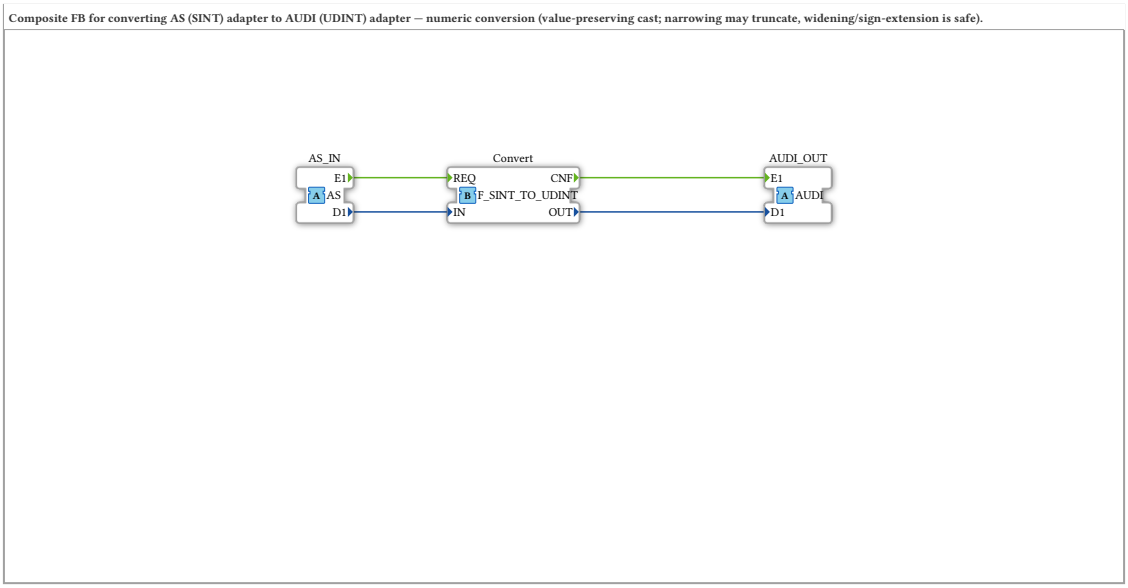
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AS_AUDI

AS_TO_AUDI





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock AS_TO_AUDI ist ein Composite-Baustein, der eine Konvertierung von einem SINT-Adapter (AS) zu einem UDINT-Adapter (AUDI) durchführt. Er dient als Brücke zwischen zwei unterschiedlichen Adaptertypen und ermöglicht die nahtlose Integration von Komponenten, die unterschiedliche Datentypen verwenden.

Schnittstellenstruktur

Ereignis-Eingänge
Nicht vorhanden. Die Ereignissteuerung erfolgt über den Adapter-Socket **AS_IN**.

Ereignis-Ausgänge
Nicht vorhanden. Die Ereignisausgabe erfolgt über den Adapter-Plug **AUDI_OUT**.

Daten-Eingänge
Nicht vorhanden. Die Datenaufnahme erfolgt über den Adapter-Socket **AS_IN**.

Daten-Ausgänge
Nicht vorhanden. Die Datenausgabe erfolgt über den Adapter-Plug **AUDI_OUT**.

Adapter

Name	Typ	Richtung	Beschreibung
AS_IN	`adapter::types::unidirectional::AS`	Socket (Eingang)	SINT-Adapter, der einen Eingangswert (SINT) und ein zugehöriges Ereignis bereitstellt.
AUDI_OUT	`adapter::types::unidirectional::AUDI`	Plug (Ausgang)	UDINT-Adapter, der den konvertierten Wert (UDINT) und ein Ereignis ausgibt.

Die Adapter enthalten jeweils ein Daten- und ein Ereigniselement:

- **AS_IN**: Daten `D1` (SINT), Ereignis `E1`.
- **AUDI_OUT**: Daten `D1` (UDINT), Ereignis `E1`.

Funktionsweise
Der Baustein verwendet intern die IEC 61131-Funktion `F_SINT_TO_UDINT`, um den eingehenden SINT-Wert in einen UDINT-Wert umzuwandeln. Die Ablaufsteuerung erfolgt ereignisgesteuert:

1. Ein Ereignis am Element `E1` des Sockets **AS_IN** (z. B. von einem vorgelagerten Baustein) wird zum Eingang `REQ` des internen Konvertierungsbausteins `Convert` weitergeleitet.
2. Der Konvertierungsbaustein liest den aktuellen Wert von `AS_IN.D1` und wandelt ihn in einen UDINT-Wert um.
3. Nach erfolgreicher Konvertierung wird das Ausgangsereignis `CNF` des Konverters ausgelöst, welches wiederum das Ereignis am Element `E1` des Plugs **AUDI_OUT** aktiviert.
4. Gleichzeitig wird der konvertierte Wert über `Convert.OUT` auf `AUDI_OUT.D1` geschrieben.

Damit wird sichergestellt, dass der konvertierte UDINT-Wert erst dann am Ausgang anliegt, wenn das zugehörige Ereignis signalisiert wird.

Technische Besonderheiten
- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-Pattern**: Der Baustein ist als Composite (zusammengesetzter Funktionsblock) realisiert, der einen einzelnen Konvertierungsbaustein kapselt. Dadurch entsteht eine wiederverwendbare und gekapselte Einheit.
- **Verwendung von IEC 61131-Bausteinen**: Die Konvertierung nutzt die standardisierte Funktion `F_SINT_TO_UDINT`, was Portabilität und Nachvollziehbarkeit fördert.
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional (keine Rückmeldung). Dies vereinfacht die Datenflussrichtung.

Zustandsübersicht
Der Composite-Baustein besitzt keine eigene Zustandsmaschine (ECC). Die gesamte Ablaufsteuerung wird durch die Ereignisverkettung des internen Netzwerks definiert. Somit gibt es keine definierten Zustände oder Zustandsübergänge.

Anwendungsszenarien

- **Systemintegration**: Wenn ein Subsystem Daten im SINT-Format über einen Adapter bereitstellt, aber ein nachfolgendes Subsystem UDINT-Daten über einen Adapter erwartet.
- **Protokollanpassung**: In heterogenen Automatisierungssystemen, in denen unterschiedliche Datentypen über Adapter ausgetauscht werden, kann dieser Baustein als universeller Konverter dienen.
- **Vereinfachung der Netzwerkstruktur**: Anstatt die Konvertierung direkt in den Applikationsbausteinen vorzunehmen, wird sie in einem eigenen Composite gekapselt, was die Wartbarkeit erhöht.

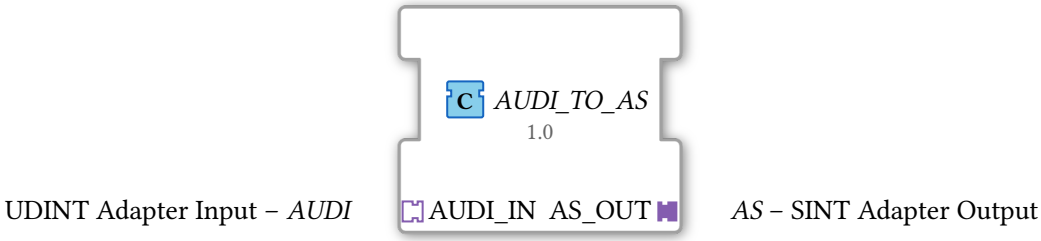
Vergleich mit ähnlichen Bausteinen

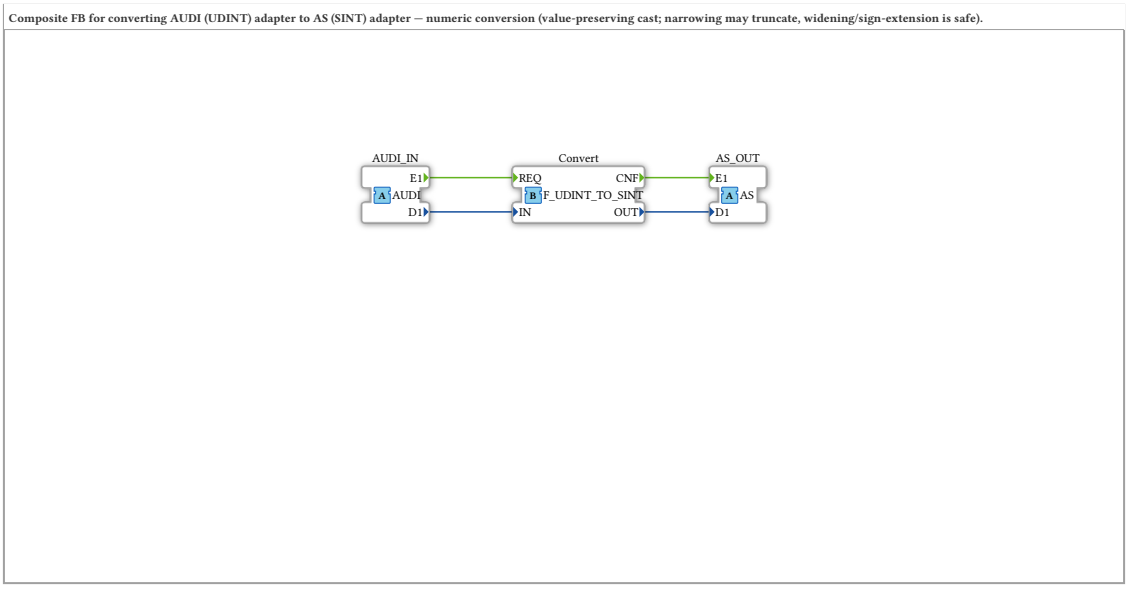
Ähnliche Bausteine existieren für andere Datentyp-Konvertierungen, z. B. `AS_TO_ABOOL` (SINT nach BOOL) oder `AUINT_TO_AUDI` (UINT nach UDINT). Der vorliegende Baustein unterscheidet sich durch die spezifische Kombination aus SINT-Eingang und UDINT-Ausgang. Er ergänzt die Bibliothek um eine weitere Konvertierungsmöglichkeit und folgt dem gleichen Entwurfsmuster (Composite mit einem internen IEC-61131-Konverter).

Fazit

Der Funktionsblock `AS_TO_AUDI` bietet eine elegante Lösung zur Umwandlung von SINT-Adapterdaten in UDINT-Adapterdaten. Durch die Kapselung in einem Composite-Baustein wird die Wiederverwendbarkeit und Übersichtlichkeit des Gesamtsystems verbessert. Die Verwendung standardisierter Konvertierungsfunktionen stellt sicher, dass die Konvertierung korrekt und effizient erfolgt.

AUDI_TO_AS





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUDI_TO_AS** ist ein Composite-Baustein, der einen AUDI-Adapter (Datentyp `UDINT`) in einen AS-Adapter (Datentyp `SINT`) umwandelt. Er dient als Brücke zwischen zwei unterschiedlichen Adapter-Schnittstellen und ermöglicht die verlustbehaftete Konvertierung eines vorzeichenlosen 32-Bit-Wertes in einen vorzeichenbehafteten 8-Bit-Wert. Die Umwandlung erfolgt ereignisgesteuert unter Verwendung der internen Konvertierungsfunktion `F_UDINT_TO_SINT`.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine eigenständigen Ereignis-Eingänge. Die Ereignissteuerung erfolgt über den **Socket-Adapter `AUDI_IN`**, der ein Ereignis `E1` bereitstellt.

Ereignis-Ausgänge

Der FB besitzt keine eigenständigen Ereignis-Ausgänge. Das Ergebnisereignis wird über den **Plug-Adapter `AS_OUT`** mit dem Ereignis `E1` ausgegeben.

Daten-Eingänge

Daten-Eingänge sind ausschließlich über den **Socket-Adapter `AUDI_IN`** verfügbar:

- **`D1`** (Datentyp `UDINT`): Der umzuwandelnde 32-Bit-Wert.

Daten-Ausgänge

Daten-Ausgänge werden über den **Plug-Adapter `AS_OUT`** bereitgestellt:

- **`D1`** (Datentyp `SINT`): Der konvertierte 8-Bit-Wert (Wertebereich −128…127).

> Hinweis: Ein `UDINT`-Wert größer als 127 wird durch die Konvertierung abgeschnitten oder interpretiert (abhängig von der Implementierung von `F_UDINT_TO_SINT`).

Adapter

Name	Typ	Richtung	Beschreibung
`AUDI_IN`	`adapter::types::unidirectional::AUDI`	Socket	Eingangsadapter (UDINT)
`AS_OUT`	`adapter::types::unidirectional::AS`	Plug	Ausgangsadapter (SINT)

Funktionsweise

Der Baustein arbeitet als transparente Konvertierungsstufe zwischen zwei unidirektionalen Adaptern. Ein eingehendes Ereignis auf `AUDI_IN.E1` triggert den internen Funktionsblock `F_UDINT_TO_SINT`, welcher den an `AUDI_IN.D1` anliegenden `UDINT`-Wert in einen `SINT`-Wert umrechnet. Nach Abschluss der Konvertierung wird ein Ereignis auf `AS_OUT.E1` ausgegeben und der konvertierte Wert auf `AS_OUT.D1` bereitgestellt.

Der interne Ablauf ist rein kausal und benötigt keine eigenen Zustände oder Timings – die Reaktion erfolgt unmittelbar nach dem Eingangsereignis.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Adapterbasierte Schnittstelle**: Der FB kommuniziert ausschließlich über spezielle Adapter, was eine lose Kopplung und Wiederverwendbarkeit in unterschiedlichen Umgebungen ermöglicht.
- **Datenverlust möglich**: Da `SINT` nur 8 Bit umfasst, gehen bei einem `UDINT`-Eingangswert über 127 Informationen verloren. Die genaue Abbildung (z. B. Beschneidung oder modulare Reduktion) hängt von der internen Konvertierungsfunktion ab.
- **Keine Eigenlogik**: Der Baustein besteht vollständig aus der Verschaltung des vorhandenen Konvertierungs-FBs und zweier Adapter – ein reiner Kompositionstyp.

Zustandsübersicht

Der FB besitzt keinen eigenen Zustandsautomaten. Er verhält sich zustandslos: Jedes eingehende Ereignis am Socket löst genau eine Ausgabe am Plug aus. Die Verarbeitung ist deterministisch und ereignissynchron.

Anwendungsszenarien

- **Protokollanpassung**: Wenn ein Subsystem Daten als `UDINT` über einen AUDI-Adapter liefert, das Zielsystem aber nur `SINT` über einen AS-Adapter akzeptiert (z. B. bei 8-Bit-Aktorsteuerungen).

- **Wertebereichsreduktion**: Explizite Einschränkung eines großen Zahlenraums auf einen kleineren, um Kompatibilität mit älteren oder ressourcenbeschränkten Komponenten herzustellen.
- **Test und Simulation**: Einsatz in Testumgebungen, in denen Adapter-Schnittstellen unterschiedlicher Datentypen aufeinander treffen.

Vergleich mit ähnlichen Bausteinen

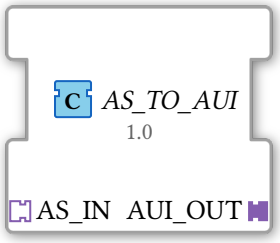
- **`UDINT_TO_SINT` (direkt)**: Der reine Konvertierungs-FB ohne Adaptereinbindung. `AUDI_TO_AS` kapselt diesen Baustein und fügt die spezifischen Adapter hinzu.
- Weitere Adapterkonverter (z. B. `DINT_TO_AS`, `AUDI_TO_INT`) bedienen andere Datentyp-Kombinationen. Der vorliegende Baustein ist speziell für die Kombination `AUDI` ↔ `AS` optimiert.

Fazit

Der Composite-FB `AUDI_TO_AS` stellt eine einfache, aber effektive Möglichkeit dar, einen `UDINT`-basierten Adapter in einen `SINT`-basierten Adapter zu überführen. Dank der klaren Ereignissteuerung und der Verwendung einer bewährten Konvertierungsfunktion eignet er sich für alle Szenarien, in denen eine verlustbehaftete Datentyp-Anpassung zwischen diesen beiden Adapter-Standards erforderlich ist.

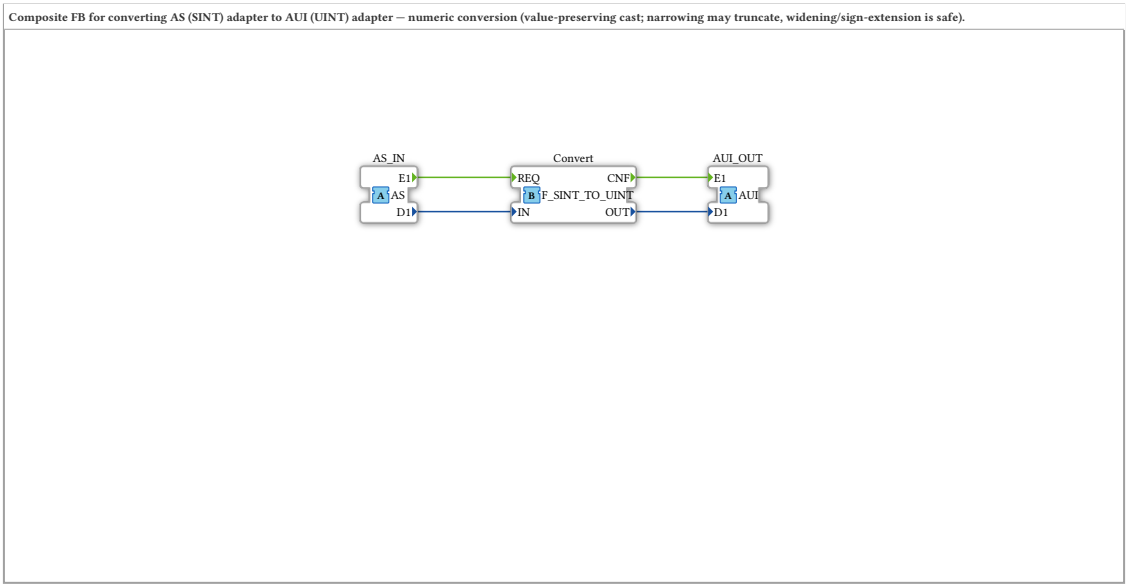
AS_AUI

AS_TO_AUI



SINT Adapter Input – *AS*

AUI – UINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AS_TO_AUI` ist ein zusammengesetzter Baustein (Composite FB), der die Konvertierung eines SINT-Adapter-Inputs (`AS`) in einen UINT-Adapter-Output (`AUI`) realisiert. Er kapselt die Typumwandlung und ermöglicht eine saubere, adapterbasierte Schnittstelle zwischen Komponenten, die unterschiedliche Datentypen verwenden.

Schnittstellenstruktur

Ereignis-Eingänge

Bezeichnung	Datentyp	Beschreibung
`E1` (über `AS_IN`)	EVENT	Startet die Konvertierung des anliegenden SINT-Werts.

Ereignis-Ausgänge

Bezeichnung	Datentyp	Beschreibung
`E1` (über `AUI_OUT`)	EVENT	Bestätigt die abgeschlossene Konvertierung und Ausgabe des UINT-Werts.

Daten-Eingänge

Bezeichnung	Datentyp	Beschreibung
`D1` (über `AS_IN`)	SINT	Der zu konvertierende vorzeichenbehaftete 8-Bit-Wert.

Daten-Ausgänge

Bezeichnung	Datentyp	Beschreibung
`D1` (über `AUI_OUT`)	UINT	Der konvertierte vorzeichenlose 16-Bit-Wert.

Adapter

Bezeichnung	Typ	Richtung	Beschreibung
`AS_IN`	`adapter::types::unidirectional::AS`	Socket (Eingang)	Stellt den SINT-Wert und das auslösende Ereignis bereit.
`AUI_OUT`	`adapter::types::unidirectional::AUI`	Plug (Ausgang)	Gibt den konvertierten UINT-Wert und das Bestätigungsereignis aus.

Funktionsweise

Der Baustein arbeitet ereignisgesteuert:

1. Wenn am `AS_IN`-Adapter das Ereignis `E1` eintrifft, wird der anliegende SINT-Datenwert (`D1`) an den internen Konvertierungsbaustein `F_SINT_TO_UINT` weitergeleitet.
2. Der `F_SINT_TO_UINT`-Baustein führt die Datentypkonvertierung gemäß IEC 61131 durch (SINT → UINT).
3. Sobald die Konvertierung abgeschlossen ist, wird der resultierende UINT-Wert an den `AUI_OUT`-Adapter übergeben und das Ereignis `E1` auf `AUI_OUT` ausgelöst.

Der gesamte Ablauf ist innerhalb eines Ereigniszyklus atomar: Ereignis rein, Daten wandeln, Ereignis raus.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- Der Baustein verwendet intern die IEC-61131-Konvertierungsfunktion `F_SINT_TO_UINT`, die aus der Bibliothek `iec61131::conversion` stammt.
- Die Adapter sind als **unidirektionale** Typen ausgelegt (AS für Eingang, AUI für Ausgang), was die Datenflussrichtung klar definiert und Missverständnisse vermeidet.
- Es gibt keine internen Zustände oder Zeitverhalten – die Konvertierung erfolgt rein kombinatorisch, gesteuert durch das Ereignis.
- Der Baustein ist unter der Eclipse Public License 2.0 lizenziert.

Zustandsübersicht

Da es sich um einen zusammengesetzten Baustein ohne eigene Zustandsmaschine handelt, existiert kein interner Zustandsautomat. Der Baustein ist vollständig ereignisgetrieben und führt bei jedem Auslösen die Konvertierung sofort aus.

Anwendungsszenarien

- **Anbindung unterschiedlicher Bussysteme**: Wenn ein Sensor (z. B. AS-Interface) SINT-Werte liefert, die Steuerung aber UINT-Werte erwartet, kann dieser Baustein die Übersetzung übernehmen.
- **Protokoll- oder Adapter-Kompatibilität**: Einsatz in heterogenen Automatisierungssystemen, in denen verschiedene Feldbusse oder Komponenten mit unterschiedlichen Datentypen arbeiten.
- **Datenaufbereitung**: Vorbereitung von SINT-Daten für eine weitere Verarbeitung, die ausschließlich UINT unterstützt.

Vergleich mit ähnlichen Bausteinen

- **Direkte Konvertierungs-FBs** (z. B. `F_SINT_TO_UINT`): Diese arbeiten auf Ebene einzelner Signale ohne Adapter-Kapselung. `AS_TO_AUI` integriert die Adapter-Schnittstelle und ist daher leichter in adapterbasierte Architekturen einbindbar.
- **Bidirektionale Adapter-Konverter**: Falls eine Umwandlung in beide Richtungen nötig ist, müsste ein separater Baustein (z. B. `AUI_TO_AS`) verwendet werden. `AS_TO_AUI` ist strikt unidirektional.

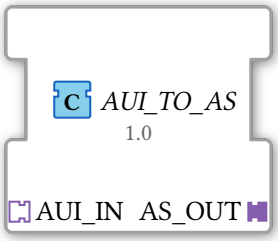
Fazit

Der `AS_TO_AUI`-Baustein bietet eine einfache, zuverlässige und standardisierte Methode, um SINT-Adapter-Daten in ein UINT-Adapter-Format zu konvertieren. Dank der Verwendung von IEC-61131-Bausteinen und unidirektionalen Adaptern ist er robust, leicht verständlich und gut in bestehende 4diac-Projekte integrierbar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

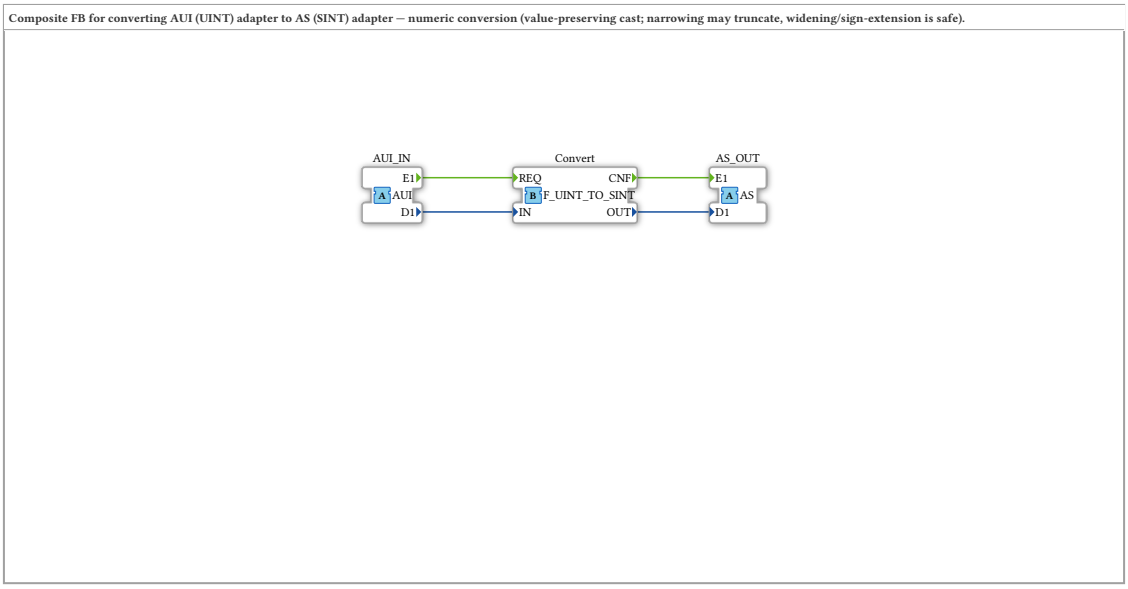
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUI_TO_AS



UINT Adapter Input – *AUI*

AS – SINT Adapter Output



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock `AUI_TO_AS` ist ein Composite Baustein, der die Konvertierung eines **AUI (UINT)**-Adapters in einen **AS (SINT)**-Adapter ermöglicht. Er kapselt die notwendige Signal- und Datenumwandlung in einer wiederverwendbaren Einheit und stellt die konvertierten Werte über einen einheitlichen Adapterausgang bereit.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt auf seiner obersten Ebene **keine direkten Ereignis-Eingänge**. Das auslösende Ereignis wird über den Socket `AUI_IN` (Adapter `adapter::types::unidirectional::AUI`) bereitgestellt. Dieses Ereignis (bezeichnet als `E1`) startet die Konvertierung intern.

Ereignis-Ausgänge

Auf der obersten Ebene sind **keine direkten Ereignis-Ausgänge** vorhanden. Das Abschlussereignis der Konvertierung wird über den Plug `AS_OUT` (Adapter `adapter::types::unidirectional::AS`) als Ereignis `E1` nach außen gegeben.

Daten-Eingänge

Es existieren **keine direkten Daten-Eingänge**. Die Eingangsdaten (Datentyp `UINT`) werden über den Socket `AUI_IN` (Kanal `D1`) in den Baustein eingespeist.

Daten-Ausgänge

Auch **keine direkten Daten-Ausgänge** sind vorhanden. Die konvertierten Daten (Datentyp `SINT`) verlassen den Baustein über den Plug `AS_OUT` (Kanal `D1`).

Adapter

Richtung	Name	Typ	Beschreibung
Socket	AUI_IN	`adapter::types::unidirectional::AUI`	Eingangsadapter (UINT-Daten)
Plug	AS_OUT	`adapter::types::unidirectional::AS`	Ausgangsadapter (SINT-Daten)

Funktionsweise

Der Baustein arbeitet als reine Kapselung eines Konvertierungsvorgangs:

1. Ein eingehendes Ereignis über `AUI_IN.E1` triggert den internen Funktionsblock `Convert` (vom Typ `iec61131::conversion::F_UINT_TO_SINT`).

2. Gleichzeitig wird der Datenwert von `AUI_IN.D1` (Typ `UINT`) an den Eingang `Convert.IN` weitergeleitet.
3. Nach Ausführung der Umwandlung sendet `Convert.CNF` ein Ereignis an `AS_OUT.E1`.
4. Der konvertierte Wert (`SINT`) wird von `Convert.OUT` auf `AS_OUT.D1` übertragen.

Dadurch wird eine durchgängige, ereignisgesteuerte Konvertierung von einem UINT-Adapter zu einem SINT-Adapter realisiert, ohne dass der Anwender die Konvertierungsdetails selbst implementieren muss.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-Baustein:** Alle Verbindungen sind innerhalb eines Netzwerks gekapselt. Die interne Logik bleibt für die aufrufende Umgebung verborgen.
- **Verwendung der Standard-Konvertierungsfunktion:** Die eigentliche Umwandlung erfolgt durch den IEC-61131-Baustein `F_UINT_TO_SINT`. Dieser ist extern referenziert und wird als Bibliotheksbaustein importiert.
- **Unidirektionale Adapter:** Sowohl Ein- als auch Ausgangsadapter sind unidirektional ausgelegt, d.h. die Daten fließen nur in eine Richtung (vom Eingang zum Ausgang).
- **Keine eigenen Zustände:** Da es sich um einen Composite-Baustein handelt, existiert kein eigenes ECC (Execution Control Chart). Das Verhalten ergibt sich ausschließlich aus den internen Verbindungen.

Zustandsübersicht

Der Baustein besitzt **keinen eigenen Zustandsautomaten**. Seine Funktionsweise ist rein datenfluss- und ereignisgesteuert durch die Vernetzung der enthaltenen Elemente. Es gibt keine definierten Zustände wie IDLE, BUSY oder ERROR.

Anwendungsszenarien

- **Adapter-Brückenbildung:** Wenn in einer Anwendung Adapter mit unterschiedlichen Datentypen (z.B. AUI mit `UINT` und AS mit `SINT`) aufeinandertreffen, kann dieser Baustein als Konverter zwischen den Adapter-Schnittstellen eingesetzt werden.
- **Systemintegration:** In heterogenen Steuerungssystemen, in denen Komponenten unterschiedliche Datentypen verwenden, erleichtert der Baustein die Integration ohne manuelle Typkonvertierung.
- **Wiederverwendung:** Der Composite-Baustein kann mehrfach in verschiedenen Projekten oder Netzwerken eingebunden werden, um immer die gleiche Konvertierungslogik bereitzustellen.

Vergleich mit ähnlichen Bausteinen

- **Andere Konvertierungs-Adapter:** Es existieren ähnliche Bausteine wie `SINT_TO_UINT_AS` oder generische Typkonverter. Der Unterschied liegt in der spezifischen Typkombination (UINT → SINT) und der Nutzung von Adaptern statt direkter Ereignis-/Daten-Eingänge.
- **Direkte Konvertierung ohne Adapter:** Statt eines Adapter-basierten Bausteins könnte man auch die Funktion `F_UINT_TO_SINT` direkt verwenden und die Adapter-Schnittstellen separat verdrahten. Der Composite-Baustein vereinfacht dies durch Kapselung.

Fazit

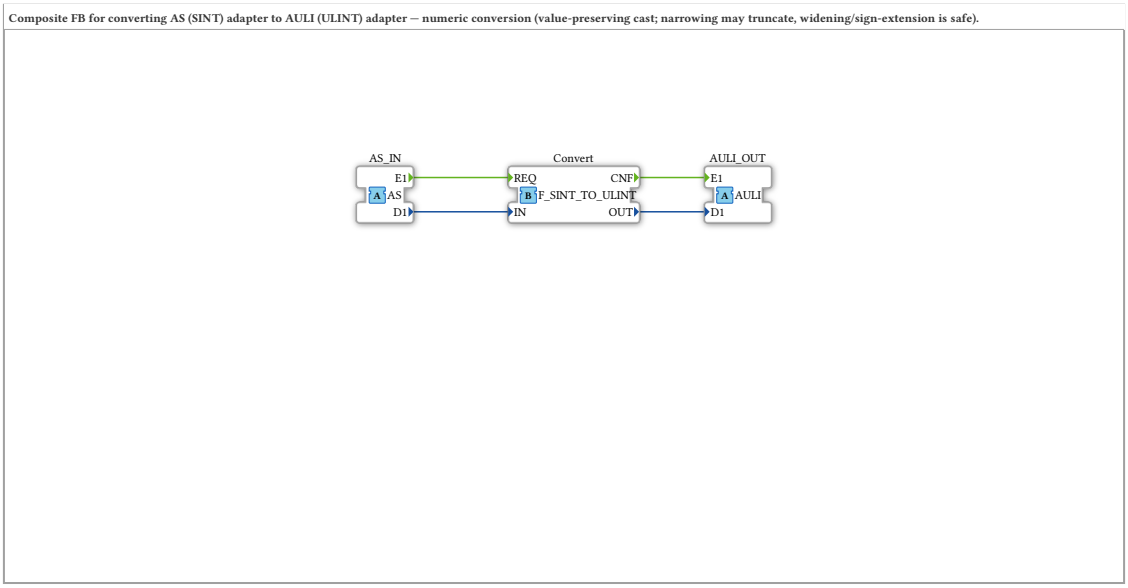
Der Funktionsblock `AUI_TO_AS` bietet eine einfache, gekapselte Lösung zur Umwandlung eines unidirektionalen UINT-Adapters in einen SINT-Adapter. Durch die Nutzung einer standardisierten IEC-61131-Konvertierungsfunktion und die klare Ereignissteuerung fügt er sich nahtlos in 4diac-IDE-Projekte ein. Er eignet sich besonders für Szenarien, in denen Adapter unterschiedlicher Datentypen verbunden werden müssen, ohne dass der Anwender die Konvertierungslogik selbst implementieren muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

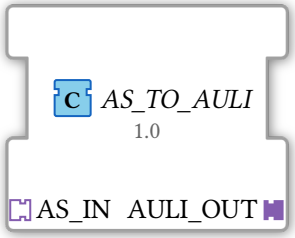
AS_AULI

AS_TO_AULI



Einleitung

SINT Adapter Input – *AS*



AULI – ULINT Adapter Output

Der zusammengesetzte Funktionsbaustein `AS_TO_AULI` dient der Umwandlung eines **AS (SINT)-Adapters** in einen **AULI (ULINT)-Adapter**. Er verwendet intern den IEC 61131-Umwandlungsbaustein `F_SINT_TO_ULINT`, um die eigentliche Typkonvertierung von SINT auf ULINT durchzuführen. Der Baustein ist als Teil einer unidirektionalen Adapter-Bibliothek für die Kommunikation zwischen unterschiedlichen Datentypen konzipiert.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt **keine direkten Ereignis-Eingänge**. Der Empfang von Ereignissen erfolgt über den **Socket-Adapter `AS_IN`**:

- **`AS_IN.E1`** – Ereignis-Eingang über den AS-Adapter, der die Datenumwandlung anstößt.

Ereignis-Ausgänge

Der Baustein besitzt **keine direkten Ereignis-Ausgänge**. Die Ausgabe von Ereignissen erfolgt über den **Plug-Adapter `AULI_OUT`**:

- **`AULI_OUT.E1`** – Ereignis-Ausgang über den AULI-Adapter, der nach abgeschlossener Konvertierung aktiviert wird.

Daten-Eingänge

Der Baustein besitzt **keine direkten Daten-Eingänge**. Die Dateneingabe erfolgt über den **Socket-Adapter `AS_IN`**:

- **`AS_IN.D1`** – Daten-Eingang vom Typ **SINT**, der den umzuwandelnden Wert bereitstellt.

Daten-Ausgänge

Der Baustein besitzt **keine direkten Daten-Ausgänge**. Die Datenausgabe erfolgt über den **Plug-Adapter `AULI_OUT`**:

- **`AULI_OUT.D1`** – Daten-Ausgang vom Typ **ULINT**, der den konvertierten Wert liefert.

Adapter

Adapter	Typ	Richtung	Beschreibung
`AS_IN`	`adapter::types::unidirectional::AS` (Socket)	Eingang	Empfängt SINT-Werte und Auslöseereignis `E1`.
`AULI_OUT`	`adapter::types::unidirectional::AULI` (Plug)	Ausgang	Sendet ULINT-Werte und Bestätigungsereignis `E1`.

Funktionsweise

1. Der Baustein erwartet ein Ereignis am Socket-Adapter `AS_IN.E1`. Gleichzeitig muss der Datenwert `AS_IN.D1` (Typ SINT) anliegen.
2. Das Ereignis wird an den internen Funktionsbaustein `F_SINT_TO_ULINT` (weitergeleitet als `REQ`).
3. Der interne Baustein führt die Konvertierung **SINT → ULINT** durch.
4. Nach erfolgreicher Umwandlung sendet `F_SINT_TO_ULINT` ein Bestätigungsereignis (`CNF`) an den Plug-Adapter `AULI_OUT.E1`.
5. Der konvertierte Wert (Typ ULINT) wird über `AULI_OUT.D1` ausgegeben.

Die Konvertierung erfolgt nach den Regeln der IEC 61131-3: Der SINT-Wert (8-Bit vorzeichenbehaftet) wird in einen vorzeichenlosen 64-Bit-Wert (ULINT) umgerechnet. Negative SINT-Werte führen dabei zu einer negativen ULINT-Repräsentation (Zweierkomplement-Umwandlung auf 64 Bit).

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Standardkonformität**: Der Baustein ist gemäß IEC 61499-2 spezifiziert.
- **Lizenz**: Eclipse Public License 2.0 (EPL-2.0).
- **Abhängigkeit**: Importiert den IEC 61131-Baustein `F_SINT_TO_ULINT` aus dem Paket `iec61131::conversion`.
- **Typsicherheit**: Die Adapter `AS` und `AULI` gewährleisten typsichere Verbindungen; eine direkte Verkabelung von SINT auf ULINT ohne Konvertierung ist nicht möglich.
- **Unidirektionaler Datenfluss**: Der Baustein arbeitet rein unidirektional – er empfängt Daten und Ereignisse und gibt konvertierte Daten und Bestätigungen aus.

Zustandsübersicht

Der Baustein besitzt **keine eigene Zustandsmaschine**. Er fungiert als reine Verschaltung des inneren `F_SINT_TO_ULINT`-Bausteins. Das Verhalten ist strikt ereignisgesteuert:

- **Warten** (Initialzustand) ⟶ Ereignis an `AS_IN.E1` ⟶ Übergang zu **Konvertierung**.
- **Konvertierung** ⟶ interner Baustein arbeitet asynchron ⟶ nach Beendigung ⟶ Ereignis an `AULI_OUT.E1` ⟶ Rückkehr zu **Warten**.

Eine Fehlerbehandlung ist nicht explizit modelliert; der innere Baustein gibt bei ungültigen Eingaben (z. B. außerhalb des Wertebereichs) ein definiertes Ergebnis (z. B. 0) aus.

Anwendungsszenarien

- **Einbindung in Adapterketten**: Wenn ein vorangeschalteter Funktionsbaustein einen SINT-Wert liefert, nachfolgende Komponenten jedoch nur ULINT verarbeiten, kann `AS_TO_AULI` als Zwischenadapter verwendet werden.
- **Sensor-Anbindung**: Sensoren mit 8-Bit-Ausgabe (z. B. Drehgeber, einfache Schalter) können an einen 64-Bit-Bus angebunden werden.

- **Legacy-Systeme**: Altanlagen mit SINT-Schnittstellen werden durch diesen Adapter kompatibel zu modernen ULINT-basierten Steuerungen.

Vergleich mit ähnlichen Bausteinen

Baustein	Zieltyp	Quelltyp	Besonderheit
`AS_TO_AULI`	ULINT	SINT	Konvertierung SINT ⟶ ULINT
`AS_TO_AUDI`	UDINT	SINT	Konvertierung SINT ⟶ UDINT (32 Bit)
`AS_TO_AWORD`	WORD	SINT	Konvertierung SINT ⟶ WORD (16 Bit)
`AULI_TO_AS`	SINT	ULINT	Rückkonvertierung ULINT ⟶ SINT (mit Überlaufbegrenzung)

Der vorliegende Baustein deckt die Konvertierung von einem 8-Bit-Vorzeichen-Typ auf einen 64-Bit-vorzeichenlosen Typ ab und eignet sich besonders für Anwendungen, die einen großen Wertebereich bei der Ausgabe benötigen.

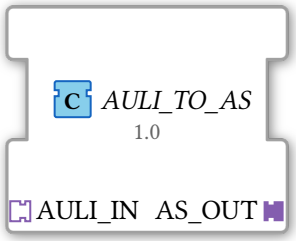
Fazit

`AS_TO_AULI` ist ein einfacher, aber wichtiger zusammengesetzter Baustein zur Typanpassung zwischen SINT- und ULINT-Adaptern. Er kapselt die IEC 61131-Konvertierung und ermöglicht eine typsichere, ereignisgesteuerte Integration in Automatisierungssysteme. Durch seine klare Schnittstellenstruktur und die Verwendung von Standardbibliotheken ist er wartbar, wiederverwendbar und leicht in bestehende 4diac-IDE-Projekte einbindbar.

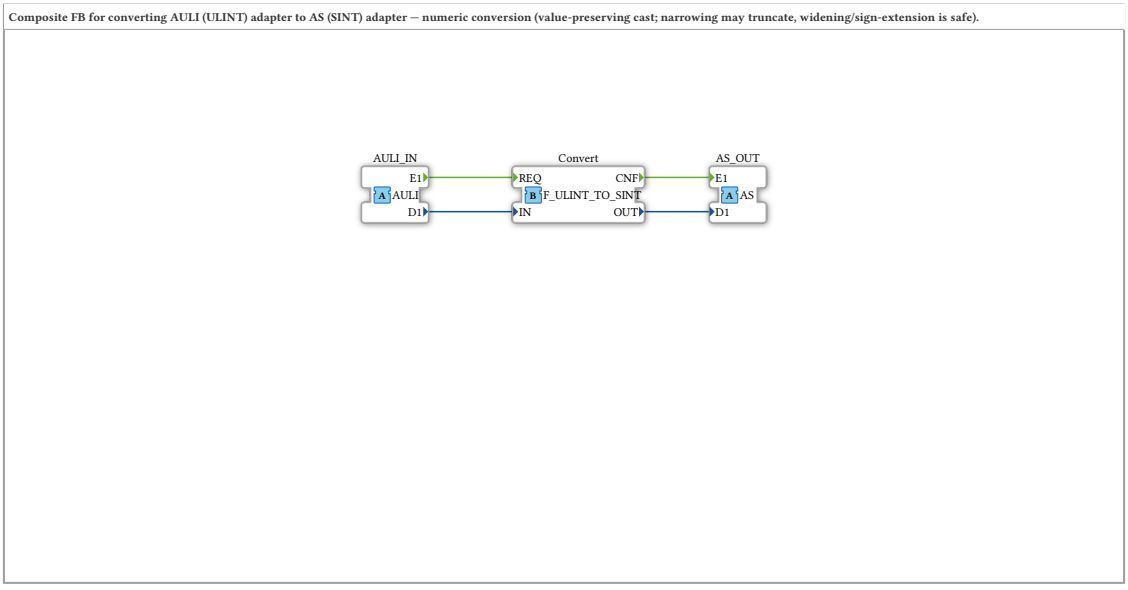
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AULI_TO_AS



ULINT Adapter Input – *AULI* *AS* – SINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AULI_TO_AS** ist ein Composite-Baustein zur Konvertierung eines unidirektionalen **AULI**-Adapters (ULINT) in einen unidirektionalen **AS**-Adapter (SINT). Er kapselt die Typumwandlung von vorzeichenlosen 64-Bit-Ganzzahlen in vorzeichenbehaftete 8-Bit-Ganzzahlen und stellt eine nahtlose Verbindung zwischen Komponenten mit unterschiedlichen Datentypanforderungen im 4diac-Framework bereit.

Schnittstellenstruktur

Der Baustein besitzt keine separaten Ereignis- oder Datenports auf der obersten Ebene. Die gesamte Kommunikation erfolgt ausschließlich über zwei Adapter.

Adapter

Adapter	Richtung	Typ	Beschreibung
`AULI_IN`	Socket (Eingang)	`adapter::types::unidirectional::AULI`	Bietet ein Ereignis `E1` und einen Datenwert `D1` vom Typ `ULINT`.
`AS_OUT`	Plug (Ausgang)	`adapter::types::unidirectional::AS`	Liefert ein Ereignis `E1` und einen Datenwert `D1` vom Typ `SINT`.

Ereignis-Eingänge (über Adapter)

- `AULI_IN.E1` – Startet die Konvertierung des anliegenden ULINT-Wertes.

Ereignis-Ausgänge (über Adapter)

- `AS_OUT.E1` – Bestätigt die Fertigstellung der Konvertierung und zeigt an, dass der neue SINT-Wert gültig ist.

Daten-Eingänge (über Adapter)

- `AULI_IN.D1` – ULINT-Wert, der umgewandelt werden soll (Wertebereich $0 \ldots 2^{24-1}$).

Daten-Ausgänge (über Adapter)

- `AS_OUT.D1` – Ergebnis der Konvertierung als SINT-Wert (Wertebereich $-128 \ldots 127$).

Funktionsweise

Der Baustein verwendet intern die IEC-61131-Konvertierungsfunktion `F_ULINT_TO_SINT`. Wird am Eingangsadapter das Ereignis `E1` empfangen, wird der aktuelle Wert von `AULI_IN.D1` ausgelesen und der Konvertierung unterzogen. Das Ergebnis wird an `AS_OUT.D1` übergeben und gleichzeitig das Ereignis `AS_OUT.E1` ausgelöst.

Die Verarbeitung ist rein ereignisgesteuert und erfolgt ohne Zwischenspeicherung interner Zustände.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Wertebereichsverlust:** Da `ULINT` einen sehr großen Wertevorrat ($0 \ldots 2^{64}-1$) abdeckt, während `SINT` nur $-128 \ldots 127$ darstellen kann, führt die Konvertierung bei Werte außerhalb des SINT-Bereichs zu einem Überlauf bzw. zur Sättigung (gemäß IEC-61131-Definition). Dies sollte bei der Anwendung bedacht werden.
- **Unidirektionale Adapter:** Der Baustein ist für Datenfluss in einer Richtung ausgelegt – vom Eingangs- zum Ausgangsadapter. Er kann nicht für bidirektionale Kommunikation verwendet werden.
- **Lizenz:** Der Baustein wird unter der Eclipse Public License 2.0 bereitgestellt.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten und keine internen Zustandsregister. Er verhält sich wie eine rein kombinatorische Funktion, die durch ein Ereignis getriggert wird. Es gibt keinen dauerhaften Speicher von Werten oder Verarbeitungszuständen.

Anwendungsszenarien

- **Sensor-Integration:** Ein Sensor liefert Daten im ULINT-Format, ein nachgeschaltetes Modul erwartet jedoch SINT-Werte (z. B. für eine einfache Anzeige oder Schwellwertlogik).
- **Protokollanpassung:** Inheterogene Automatisierungssysteme, bei denen Adapter unterschiedliche Datentypen verwenden, können mit diesem Baustein typkompatibel verbunden werden.
- **Datenreduktion:** Gezielte Umwandlung großer Zahlenbereiche in kleinere Formate zur Einsparung von Speicherplatz oder Busbandbreite (unter Inkaufnahme von Genauigkeit).

Vergleich mit ähnlichen Bausteinen

- **`AULI_TO_INT`** – Konvertiert nach `INT` (16-Bit, −32768 … 32767), größerer Wertebereich als SINT, aber dennoch verlustbehaftet bei großen Zahlen.
- **`AULI_TO_DINT`** – Konvertiert nach `DINT` (32-Bit, -2^{31} … $2^{31}-1$), deckt einen deutlich größeren Teil des ULINT-Bereichs ab und ist für viele Anwendungen die bessere Wahl.
- **`ULINT_TO_SINT` (direkt)** – Ein einfacher Funktionsbaustein ohne Adapter-Kapselung; `AULI_TO_AS` bietet denselben Dienst in einer adapterbasierten Umgebung.

Fazit

Der **AULI_TO_AS** ist ein schlanker, zweckgebundener Konverter für unidirektionale Adapter. Er ermöglicht die einfache Integration von Komponenten, die auf dem SINT-Datentyp basieren, in ein ULINT-dominiertes System. Aufgrund der großen Differenz der Wertebereiche muss der Anwender jedoch mögliche Überlauf- oder Sättigungseffekte berücksichtigen und gegebenenfalls auf breitere Datentypen ausweichen.

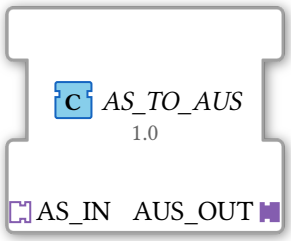
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

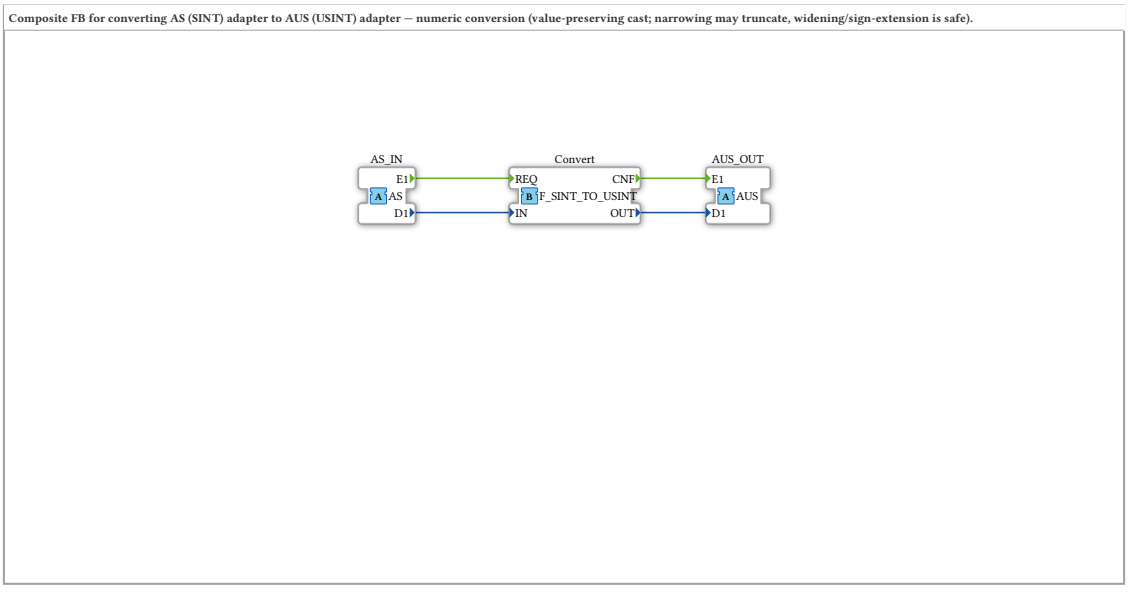
AS_AUS

AS_TO_AUS

SINT Adapter Input – *AS*



AUS – USINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AS_TO_AUS** ist ein Composite-Baustein, der einen AS-Adapter (Typ `adapter::types::unidirectional::AS`) mit einem AUS-Adapter (Typ `adapter::types::unidirectional::AUS`) verbindet und die darin enthaltenen Datenwerte von `SINT` (signed 8-Bit) nach `USINT` (unsigned 8-Bit) umwandelt. Die Konvertierung erfolgt durch den eingebetteten Standard-Baustein `F_SINT_TO_USINT` aus der IEC-61131-Bibliothek. Der FB ermöglicht eine einfache, adapterbasierte Datentypanpassung in industriellen Steuerungsanwendungen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge. Die Ereignissteuerung erfolgt ausschließlich über den Adapter `AS_IN` (Socket).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge. Die Ereignisweitergabe erfolgt ausschließlich über den Adapter `AUS_OUT` (Plug).

Daten-Eingänge

Keine direkten Daten-Eingänge. Die Datenaufnahme erfolgt ausschließlich über den Adapter `AS_IN`.

Daten-Ausgänge

Keine direkten Daten-Ausgänge. Die Datenausgabe erfolgt ausschließlich über den Adapter `AUS_OUT`.

Adapter

Adapter	Typ	Richtung	Beschreibung
`AS_IN`	`adapter::types::unidirectional::AS`	Socket (Eingang)	Nimmt einen SINT-Wert sowie ein Ereignis (E1) auf.
`AUS_OUT`	`adapter::types::unidirectional::AUS`	Plug (Ausgang)	Gibt den konvertierten USINT-Wert sowie ein Ereignis (E1) aus.

Funktionsweise

Der FB arbeitet als reiner Datenfluss-Konverter:

1. Ein externes Ereignis am Socket `AS_IN.E1` löst die Konvertierung aus.
2. Gleichzeitig wird der zugehörige Datenwert `AS_IN.D1` (SINT) an den Eingang `IN` des internen Bausteins `F_SINT_TO_USINT` übergeben.
3. Der Konvertierungsbaustein wandelt den SINT-Wert in den entsprechenden USINT-Wert um und legt ihn an seinem Ausgang `OUT` an.
4. Das Abschlussereignis (`CNF`) des Konverters triggert das Ereignis `AUS_OUT.E1` und übergibt den konvertierten Wert an `AUS_OUT.D1`.
5. Der AUS-Adapter stellt das Ergebnis der aufrufenden Umgebung zur Verfügung.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-Baustein**: Der FB enthält keine eigene Zustandslogik, sondern kombiniert den vorhandenen Standard-Konverter mit den vorgegebenen Adapter-Schnittstellen.
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional ausgelegt – die Daten und Ereignisse fließen nur in eine Richtung.
- **Typkonvertierung**: Die Wandlung von SINT nach USINT ist verlustfrei, da der Wertebereich von USINT (0…255) den positiven Teil von SINT (−128…127) vollständig abdeckt. Negative SINT-Werte werden jedoch wie folgt interpretiert: z. B. −1 wird zu 255 (Zweierkomplement). Dieses Verhalten entspricht der IEC-61131-Funktion `SINT_TO_USINT`.

Zustandsübersicht

Der FB besitzt keinen internen Zustandsautomaten (ECC). Die gesamte Ablaufsteuerung wird durch die Ereignisverkettung im Netzwerk realisiert.

Anwendungsszenarien

- **Sensorintegration**: Ein Sensor liefert einen SINT-Wert (z. B. Temperatur als signed Wert), aber das nachfolgende Modul erwartet USINT (z. B. für Anzeigen oder Logik mit unsigned Typen).
- **Systemkopplung**: Zwischen verschiedenen Steuerungskomponenten, die über Adapter kommunizieren, muss eine signierte in eine unsigned Darstellung umgewandelt werden.
- **Protokollanpassung**: In Feldbus- oder OPC-UA-Anbindungen, bei denen die Datenformate vorgegeben sind.

Vergleich mit ähnlichen Bausteinen

- **F_SINT_TO_USINT**: Dieser Standardbaustein führt die reine Typkonvertierung ohne Adapter durch. Der vorliegende FB kapselt ihn in einer Adapter-Schnittstelle, sodass er direkt in adapterbasierte Architekturen eingebunden werden kann.
- **Weitere Konverter** (z. B. `F_INT_TO_DINT`): Diese arbeiten analog, jedoch für andere Datentypen. Der Vorteil von `AS_TO_AUS` liegt in der spezifischen Adapter-Anpassung für die Typen AS und AUS.

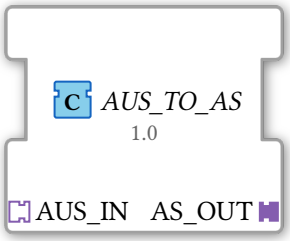
Fazit

Der Funktionsblock `AS_TO_AUS` ist ein praktischer Helfer zur Typumwandlung in adapterbasierten 4diac-Anwendungen. Er vereinfacht die Integration von Komponenten, die unterschiedliche vorzeichenbehaftete und vorzeichenlose 8-Bit-Datentypen verwenden, und ermöglicht eine saubere Trennung der Adapter-Schnittstellen. Durch die Verwendung des etablierten IEC-Konverters ist die Funktionalität zuverlässig und standardkonform.

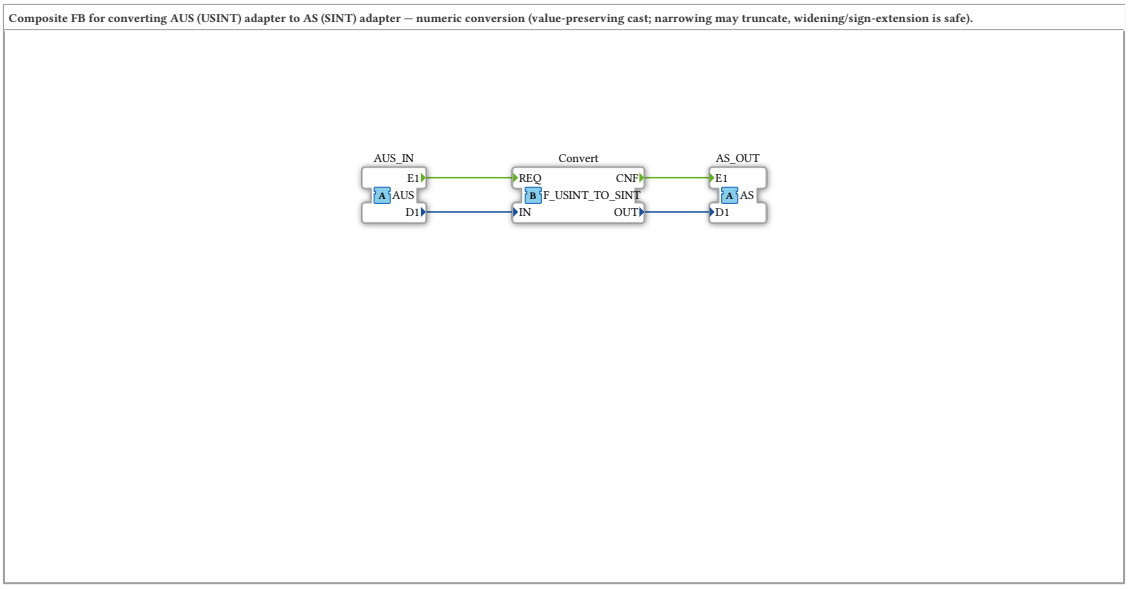
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUS_TO_AS



USINT Adapter Input – *AUS* — *AS* – SINT Adapter Output



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock **AUS_TO_AS** ist ein zusammengesetzter Baustein (Composite FB) zur Umwandlung eines unidirektionalen AUS-Adapters (basierend auf dem Datentyp USINT) in einen unidirektionalen AS-Adapter (basierend auf dem Datentyp SINT). Er dient als standardisierte Schnittstelle zwischen Komponenten, die unterschiedliche vorzeichenlose/vorzeichenbehaftete Ganzzahl-Typen verwenden.

Schnittstellenstruktur

Der FB verfügt über keine direkten Ereignis- oder Dateneingänge/-ausgänge, sondern ausschließlich über zwei Adapter-Schnittstellen:

Ereignis-Eingänge

Keine direkt sichtbaren Ereigniseingänge. Die Ereignissteuerung erfolgt über den Socket-Adapter `AUS_IN`.

Ereignis-Ausgänge

Keine direkt sichtbaren Ereignisausgänge. Die Ereignisrückmeldung erfolgt über den Plug-Adapter `AS_OUT`.

Daten-Eingänge

Keine direkt sichtbaren Dateneingänge. Die Daten werden über den Socket-Adapter `AUS_IN` bereitgestellt.

Daten-Ausgänge

Keine direkt sichtbaren Datenausgänge. Die Daten werden über den Plug-Adapter `AS_OUT` ausgegeben.

Adapter

Name	Typ	Richtung	Kommentar
`AUS_IN`	`adapter::types::unidirectional::AUS` (Socket)	Eingang	USINT-Adapter-Eingang
`AS_OUT`	`adapter::types::unidirectional::AS` (Plug)	Ausgang	SINT-Adapter-Ausgang

Die Schnittstelle der Adapter selbst enthält:

- `AUS_IN.E1` (Event), `AUS_IN.D1` (USINT)
- `AS_OUT.E1` (Event), `AS_OUT.D1` (SINT)

Funktionsweise

Der FB ist als Composite implementiert und enthält einen internen Funktionsblock `Convert` vom Typ `iec61131::conversion::F_USINT_TO_SINT`, der die eigentliche Wertekonvertierung durchführt.

1. **Datenerfassung:** Ein anliegender USINT-Wert wird über den Socket `AUS_IN.D1` bereitgestellt.
2. **Triggerung:** Ein Ereignis am Ausgang `AUS_IN.E1` löst den Eingang `REQ` des Konvertierungsbausteins aus.
3. **Konvertierung:** Der interne FB `F_USINT_TO_SINT` wandelt den USINT-Wert in einen entsprechenden SINT-Wert um (Wertebereich 0..255 ⟶ −128..127; bei Überschreitung erfolgt eine Bereichsbegrenzung gemäß IEC 61131).
4. **Rückmeldung:** Nach Abschluss der Konvertierung signalisiert der interne FB über seinen Ausgang `CNF` ein Ereignis an den Plug `AS_OUT.E1`.
5. **Datenweitergabe:** Der konvertierte SINT-Wert wird über `AS_OUT.D1` an den angeschlossenen Plug ausgegeben.

Die Verbindungen im FBNetzwerk sind:

- Ereignis: `AUS_IN.E1` ⟶ `Convert.REQ`
- Ereignis: `Convert.CNF` ⟶ `AS_OUT.E1`
- Daten: `AUS_IN.D1` ⟶ `Convert.IN`
- Daten: `Convert.OUT` ⟶ `AS_OUT.D1`

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Adapterbasiert:** Der Baustein fungiert als reiner Adapter-Konverter und kann nahtlos in bestehende adapterorientierte Systeme integriert werden.
- **Composite-Struktur:** Die Umwandlung erfolgt durch einen internen, standardisierten IEC-61131-Konverter, was eine zuverlässige und zertifizierte Konvertierung gewährleistet.
- **Unidirektionale Schnittstelle:** Die Daten- und Ereignisflüsse sind strikt von Eingang zu Ausgang gerichtet.
- **Kein expliziter Zustandsautomat:** Die Steuerung erfolgt rein ereignisgesteuert über die Adapter-Ereignisse. Es gibt keine internen Zustände.

Zustandsübersicht

Der FB besitzt keinen sichtbaren Zustandsautomaten. Er verhält sich wie ein transparentes Gateway, das bei jedem Ereignis am Eingang die Umwandlung auslöst und das Ergebnis am Ausgang bereitstellt.

Anwendungsszenarien

- **Integration von Komponenten** mit unterschiedlichen Datentypen, z. B. wenn ein Sensor USINT-Daten liefert, ein nachfolgender Regler aber SINT-Werte erwartet.
- **Adapter-Konvertierung** in modularen Steuerungssystemen, die auf unidirektionalen Adaptern basieren (z. B. nach IEC 61499).
- **Sicherheitskritische Systeme**, in denen die genaue Typumwandlung (ohne Überlauf) essenziell ist – der interne FB `F_USINT_TO_SINT` ist spezifikationskonform.

Vergleich mit ähnlichen Bausteinen

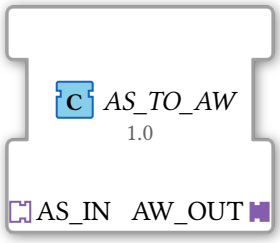
- **Einfache Konverter wie `F_USINT_TO_SINT`:** Wandeln nur skalare Werte, verwenden aber keine Adapter. `AUS_TO_AS` kapselt diese Konvertierung in einem adapterbasierten Baustein.
- **`INT_TO_DINT`-Adapter:** Andere Typkonvertierungen (z. B. INT nach DINT) folgen demselben Muster, unterscheiden sich jedoch in der Bitbreite und Vorzeichenbehandlung.
- **Bidirektionale Adapter:** Im Gegensatz zu bidirektionalen Varianten ist dieser FB nur für einen Datenfluss ausgelegt, was die Komplexität reduziert.

Fazit

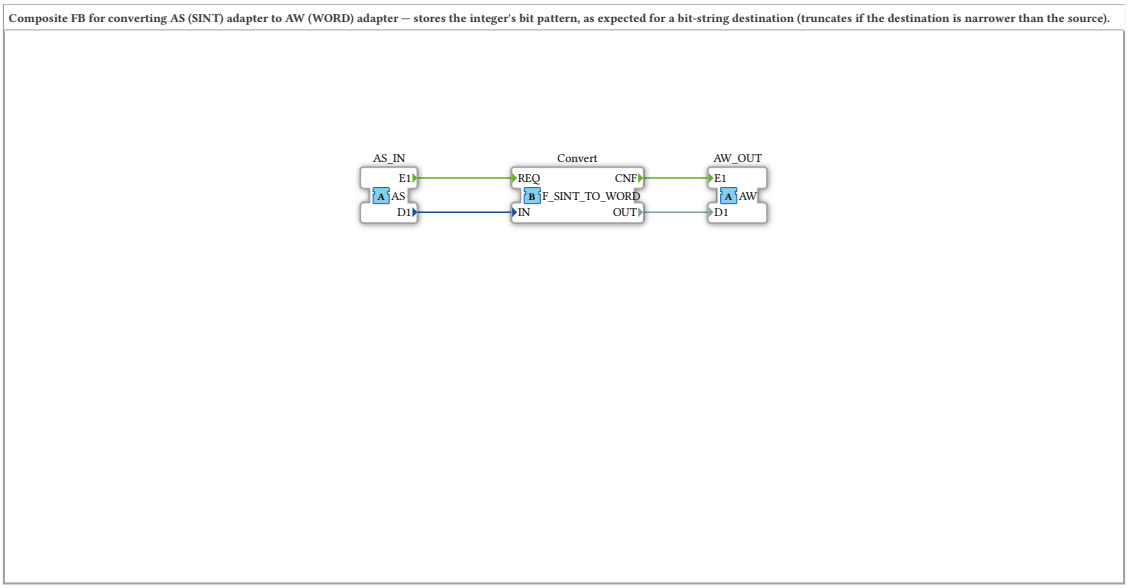
Der **AUS_TO_AS**-Funktionsblock bietet eine saubere, adapterbasierte Lösung zur Konvertierung von USINT- nach SINT-Werten. Dank seiner Composite-Struktur und der Nutzung eines standardisierten IEC-61131-Konverters ist er wartungsfreundlich, zuverlässig und leicht in industrielle Steuerungsnetzwerke integrierbar. Er eignet sich besonders für Systeme, die auf unidirektionale Adapter setzen und eine explizite Typumwandlung benötigen.

AS_AW

AS_TO_AW



SINT Adapter Input – *AS* *AW* – WORD Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AS_TO_AW** ist ein zusammengesetzter Baustein (Composite FB), der einen unidirektionalen **SINT**-Adapter (AS) in einen unidirektionalen **WORD**-Adapter (AW) umwandelt. Er dient als Brücke zwischen verschiedenen Adapter-Typen und ermöglicht die Weiterverarbeitung von SINT-Signalen als WORD-Daten in einer 4diac-Umgebung.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge. Der Baustein besitzt einen **Socket** vom Typ `AS`, der indirekt einen Ereignis-Eingang bereitstellt:

- **AS_IN.E1** – Ereignis-Eingang über den Socket; ein eingehendes Ereignis löst die Konvertierung aus.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge. Der Baustein besitzt einen **Plug** vom Typ `AW`, der indirekt einen Ereignis-Ausgang bereitstellt:

- **AW_OUT.E1** – Ereignis-Ausgang über den Plug; nach erfolgreicher Konvertierung wird ein Ereignis gesendet.

Daten-Eingänge

- **AS_IN.D1** (`SINT`) – Eingangsdatenwert des AS-Sockets; der zu konvertierende SINT-Wert.

Daten-Ausgänge

- **AW_OUT.D1** (`WORD`) – Ausgangsdatenwert des AW-Plugs; der konvertierte WORD-Wert.

Adapter

Typ	Name	Richtung	Kommentar
AS (unidirektional)	**AS_IN**	Socket (Eingang)	SINT Adapter Eingang
AW (unidirektional)	**AW_OUT**	Plug (Ausgang)	WORD Adapter Ausgang

Funktionsweise

Der Baustein enthält eine interne Instanz des Konvertierungsbausteins `iec61131::conversion::F_SINT_TO_WORD`. Die Verarbeitung erfolgt ereignisgesteuert:

1. Ein Ereignis am **AS_IN.E1** (Socket) triggert den internen Baustein über seinen `REQ`-Eingang.
2. Gleichzeitig wird der Wert von **AS_IN.D1** (`SINT`) an den `IN`-Eingang des Konverters weitergeleitet.
3. Der Konverter `F_SINT_TO_WORD` führt die Typumwandlung durch und signalisiert die Fertigstellung über seinen `CNF`-Ausgang.
4. Dieses Ereignis wird an **AW_OUT.E1** (Plug) weitergegeben, und der konvertierte `WORD`-Wert steht an **AW_OUT.D1** zur Verfügung.

Somit wird eine zuverlässige, einmalige Konvertierung bei jeder Ereignisankunft durchgeführt.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Composite FB**: Der Baustein nutzt die Composite-Funktionalität von 4diac, um einen vorhandenen Konvertierungsbaustein (`F_SINT_TO_WORD`) in eine Adapter-Hülle zu integrieren.
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional ausgelegt, d.h. nur eine Daten- und Ereignisrichtung wird unterstützt.
- **IEC 61131-3 Konvertierung**: Die eigentliche Typumwandlung erfolgt nach IEC 61131-3 Standard, was die Kompatibilität mit anderen Systemen erleichtert.
- **Copyright-Vermerk**: Der Baustein enthält einen Copyright-Hinweis (EPL 2.0), der die Nutzung und Weiterverbreitung regelt.

Zustandsübersicht

Da der Baustein aus einem internen FB mit REQ/CNF-Handshake besteht, lässt sich der Ablauf wie folgt beschreiben:

Zustand	Beschreibung
Idle	Warten auf ein Ereignis am AS_IN.E1
Busy	Konvertierung läuft (interner FB aktiv)
Done	Konvertierung abgeschlossen, Ereignis an AW_OUT.E1 gesendet

Der Baustein ist nicht zustandsbehaftet im Sinne eines eigenen Zustandsautomaten – der interne FB realisiert die Zustandslogik.

Anwendungsszenarien

- **Signalanpassung**: Ein Sensor liefert Daten als SINT (z.B. 8-Bit-Wert), das nachgeschaltete System erwartet jedoch WORD (16-Bit). Der AS_TO_AW übernimmt die Umwandlung.
- **Adapter-Integration**: In einem heterogenen Steuerungssystem müssen verschiedene Adapter-Typen miteinander verbunden werden. Der Baustein fungiert als „Übersetzer“ zwischen AS- und AW-Schnittstellen.
- **Test- und Simulationsumgebungen**: Zum Überbrücken von Adapter-Inkompatibilitäten während der Entwicklung oder beim Testen von Funktionsbausteinen.

Vergleich mit ähnlichen Bausteinen

Baustein	Funktion	Unterschied

AS_TO_AW	SINT ⟶ WORD	Spezifische Konvertierung für unidirektionale Adapter
Allgemeine Konverter (z.B. `F_SINT_TO_WORD`)	reine Datenkonvertierung ohne Adapter-Kapselung	AS_TO_AW bietet eine fertige Adapter-Schnittstelle und kann direkt in Adapter basierten Netzwerken verwendet werden
Andere Adapter-Konverter (z.B. `BOOL_TO_BYTE`)	andere Datentypen	AS_TO_AW ist auf SINT ↔ WORD spezialisiert

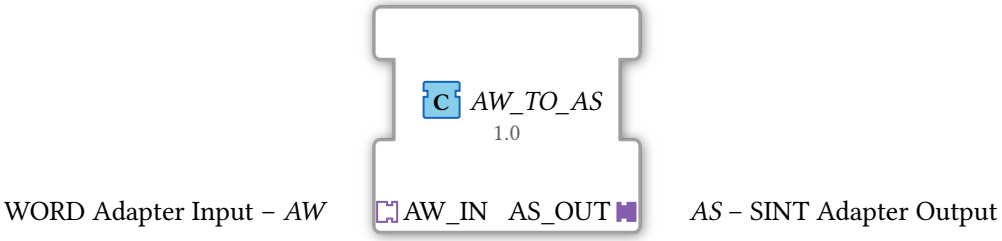
Fazit

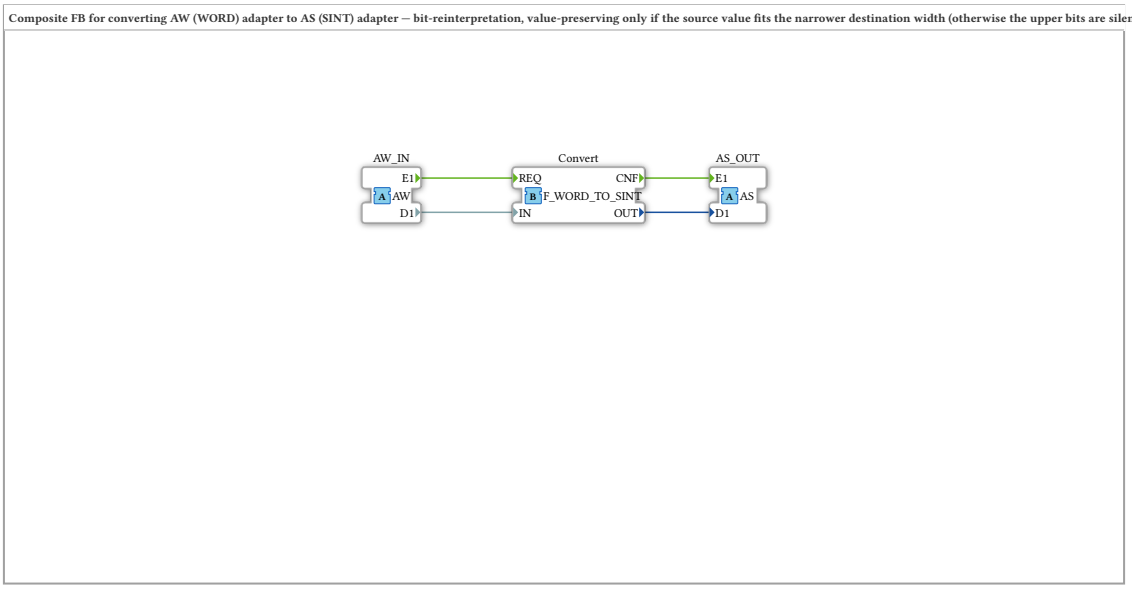
Der **AS_TO_AW** ist ein nützlicher, kompakter Konvertierungsbaustein, der die Umwandlung von einem SINT-Adapter (AS) zu einem WORD-Adapter (AW) nahtlos in die 4diac-IDE integriert. Durch die Verwendung des standardisierten IEC 61131-3 Konverters und die unidirektionale Adapter-Technologie ermöglicht er eine flexible und zuverlässige Signalumsetzung in Automatisierungsprojekten.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AW_TO_AS





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `AW_TO_AS` ist ein Composite-Baustein, der einen WORD-basierten Adapter (AW) in einen SINT-basierten Adapter (AS) umwandelt. Er kapselt die notwendige Typkonvertierung und ermöglicht so die einfache Integration unterschiedlicher Adapter-Schnittstellen in der Automatisierungstechnik.

Schnittstellenstruktur

Ereignis-Eingänge

- `AW_IN.E1` – Startet die Konvertierung des anliegenden Datenwerts.

Ereignis-Ausgänge

- `AS_OUT.E1` – Signalisiert, dass der konvertierte Wert am Datenausgang bereitsteht.

Daten-Eingänge

- `AW_IN.D1` (WORD) – Der 16-Bit-Wert, der in einen SINT-Wert umgewandelt werden soll.

Daten-Ausgänge

- `AS_OUT.D1` (SINT) – Der konvertierte 8-Bit-Wert (vorzeichenbehaftet, Wertebereich −128 … 127).

Adapter

- **Socket (Eingang)**: `AW_IN` vom Typ `adapter::types::unidirectional::AW`
- **Plug (Ausgang)**: `AS_OUT` vom Typ `adapter::types::unidirectional::AS`

Funktionsweise

Der Baustein ist als Composite realisiert und verwendet intern den IEC-61131-Funktionsbaustein `F_WORD_TO_SINT`. Ein eingehendes Ereignis an `AW_IN.E1` triggert die Wandlung: Der Datenwert von `AW_IN.D1` wird an den Eingang `IN` des internen Konverters übergeben. Nach erfolgreicher Umwandlung erzeugt dieser ein Bestätigungsereignis an `CNF`, das an den Ausgangsadapter `AS_OUT.E1` weitergeleitet wird. Gleichzeitig wird der konvertierte SINT-Wert an `AS_OUT.D1` ausgegeben.

Technische Besonderheiten

- **Bit-Reinterpretation**, werterhaltend nur, wenn der Quellwert in die schmalere Zielbreite passt (sonst werden die oberen Bits stillschweigend verworfen).
- Die Konvertierung folgt der IEC-61131-Funktion `WORD_TO_SINT`. Ein Überlauf tritt auf, wenn der eingehende WORD-Wert außerhalb des gültigen SINT-Bereichs (-128 … 127) liegt – dies muss in der Anwendung berücksichtigt werden.
- Der Baustein arbeitet rein ereignisgesteuert: Eine Wandlung erfolgt nur bei einem Ereignis am Eingang.
- Die Adapter sind unidirektional ausgelegt; eine Rückwandlung von SINT nach WORD ist nicht vorgesehen.

Zustandsübersicht

Da es sich um einen einfachen Composite-FB ohne eigenen Zustandsautomaten handelt, existiert keine explizite Zustandsmaschine. Der FB wartet auf ein Ereignis, führt die Konvertierung aus und gibt das Ergebnis synchron aus. Eine Zustandstabelle ist nicht erforderlich.

Anwendungsszenarien

- Anpassung von Sensordaten, die als WORD (z. B. über einen analogen Eingang) geliefert werden, an ein System, das SINT-Werte verarbeitet.
- Umwandlung von Feldbus-Datenformaten bei der Kopplung unterschiedlicher Automatisierungskomponenten.
- Einsatz in heterogenen Steuerungsumgebungen, in denen Adapter unterschiedliche Datentypen verwenden.

Vergleich mit ähnlichen Bausteinen

Analog aufgebaute Konverter wie `AW_TO_BY` (WORD → BYTE) oder `AINT_TO_AUDINT` (INT → UDINT) folgen demselben Prinzip. Der `AW_TO_AS`-Baustein hebt sich durch die spezifische Adapter-Kombination hervor, die eine direkte Einbindung in Adapter-basierte Architekturen ermöglicht. Im Gegensatz zu einfachen Typkonvertierungen auf Datenebene wird hier die ereignisgesteuerte Adapter-Schnittstelle genutzt, was die Wiederverwendbarkeit und Austauschbarkeit erhöht.

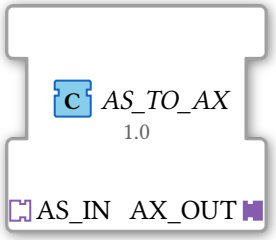
Fazit

Der Composite FB `AW_TO_AS` bietet eine kompakte und klare Lösung zur Konvertierung eines WORD-Adapters in einen SINT-Adapter. Die Kapselung der Konvertierung in einem eigenen Baustein verbessert

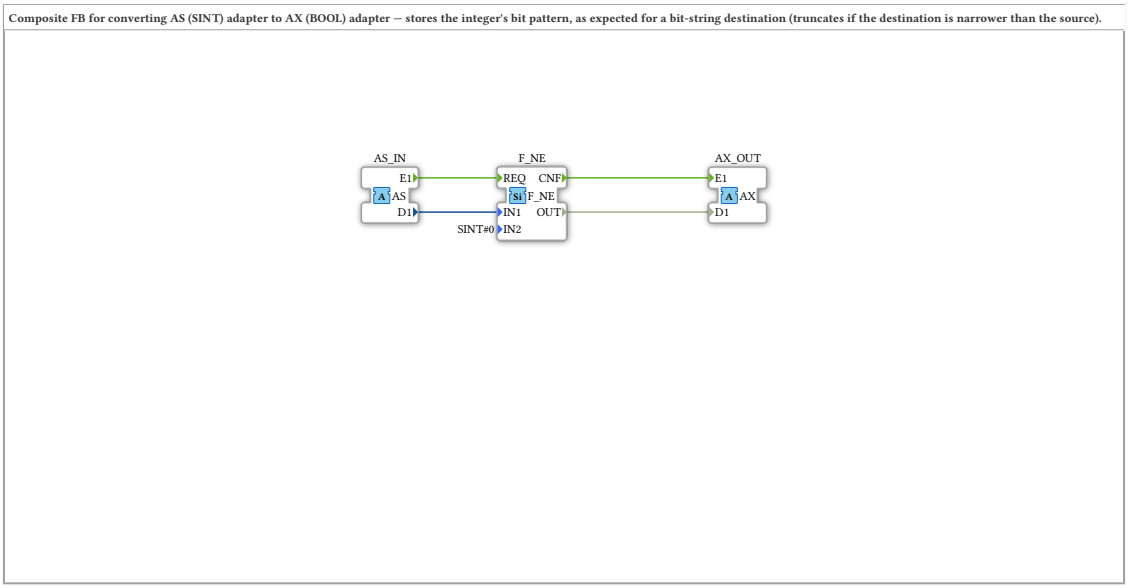
die Lesbarkeit und Wartbarkeit der Applikation. Der FB eignet sich besonders für Umgebungen, in denen mit unterschiedlichen Datentypen auf Adapterebene gearbeitet wird.

AS_AX

AS_TO_AX



SINT Adapter Input – *AS* *AX* – BOOL Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AS_TO_AX** ist ein Composite-Baustein, der einen SINT-Adapter (AS) in einen BOOL-Adapter (AX) umwandelt. Er prüft den ankommenden SINT-Wert auf Ungleichheit zu Null und gibt das Ergebnis als BOOL-Signal aus. Der Baustein eignet sich für die einfache Signalkonvertierung in einer IEC 61499-ANWENDUNG.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung
AS_IN.E1	Event	Empfängt das Triggersignal vom angeschlossenen AS-Adapter zur Verarbeitung der Daten.

Ereignis-Ausgänge

Name	Typ	Beschreibung
AX_OUT.E1	Event	Wird nach erfolgreicher Konvertierung gesendet; signalisiert, dass der BOOL-Wert am AX-Adapter anliegt.

Daten-Eingänge

Name	Typ	Beschreibung
AS_IN.D1	SINT	Empfängt den SINT-Wert, der auf Ungleichheit zu Null geprüft wird.

Daten-Ausgänge

Name	Typ	Beschreibung
AX_OUT.D1	BOOL	Gibt das Ergebnis des Vergleichs aus: `TRUE`, wenn der Eingangswert ungleich Null ist, sonst `FALSE`.

Adapter

Name	Typ	Richtung	Beschreibung
AS_IN	adapter::types::unidirectional::AS	Socket (Input)	SINT-Adapter für den Eingangswert und das zugehörige Ereignis.
AX_OUT	adapter::types::unidirectional::AX	Plug (Output)	BOOL-Adapter für den Ausgabewert und das zugehörige Ereignis.

Funktionsweise

Der Baustein nutzt intern den Funktionsblock `iec61131::comparison::F_NE` (Ungleich-Prüfung).

- Wird ein Ereignis am Eingang `AS_IN.E1` empfangen, wird die Verarbeitung von `F_NE` gestartet (Eingang `REQ`).
- `F_NE` vergleicht den über `AS_IN.D1` erhaltenen SINT-Wert mit dem konstanten Parameter `IN2` (fest auf `SINT#0` gesetzt).
- Das Ergebnis des Vergleichs (`OUT`) – `TRUE` bei Ungleichheit, `FALSE` bei Gleichheit – wird an den Datenausgang `AX_OUT.D1` weitergeleitet.
- Nach Abschluss der Berechnung wird über `F_NE.CNF` das Ereignis `AX_OUT.E1` ausgelöst, das den neuen BOOL-Wert am AX-Adapter signalisiert.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Typkonvertierung**: Der Baustein wandelt implizit einen SINT-Adapter in einen BOOL-Adapter um, ohne dass der Anwender direkte Typumwandlungslogik schreiben muss.
- **Verwendung von Standardbausteinen**: Intern kommt der bereits vorhandene IEC 61131-Baustein `F_NE` zum Einsatz, was Wiederverwendbarkeit und Testbarkeit erhöht.
- **Fester Vergleichswert**: Der Parameter `IN2` ist auf `SINT#0` voreingestellt – eine Änderung ist nur durch Anpassung des Composite-Bausteins möglich.

Zustandsübersicht

Da es sich um einen Composite-Baustein ohne eigene Zustandsmaschine handelt, beschränkt sich das Verhalten auf den Daten- und Ereignisfluss gemäß der internen Verschaltung. Die Zustände ergeben sich aus dem verarbeitenden FB `F_NE`:

- **Idle**: Es liegt kein Ereignis an.
- **Processing**: Ein Ereignis wird verarbeitet (Vergleich läuft).
- **Completed**: Der Vergleich ist abgeschlossen, das Ergebnis steht am Ausgang an.

Anwendungsszenarien

- **Schwellwertprüfung**: Umwandlung eines SINT-Zählerstands in ein BOOL-Signal, das anzeigt, ob der Wert von Null abweicht (z. B. „Motor läuft“).
- **Adapterbrücke**: Verbindung einer Komponente mit AS-Schnittstelle an eine Komponente mit AX-Schnittstelle, ohne manuelle Konvertierung.
- **Einfache Statusanzeige**: Signalweitergabe, ob ein Datensatz vorhanden ist (ungleich Null) oder nicht.

Vergleich mit ähnlichen Bausteinen

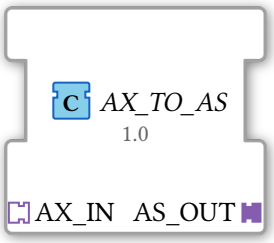
- **Direkte Konvertierungs-FBs** (z. B. `SINT_TO_BOOL`): Wandeln den Datentyp ohne Adapterlogik um, liefern aber kein ereignisgesteuertes Verhalten über Adapter.
- **Eigene Composite-Lösung**: Bietet Flexibilität, erfordert aber mehr Aufwand; `AS_TO_AX` kapselt die Konvertierung sauber und ist wiederverwendbar.
- **Adapter-Wrapper**: Andere Bausteine nutzen möglicherweise andere Vergleichsoperationen (z. B. `F_EQ`), hier wird spezifisch auf Ungleichheit zu Null geprüft.

Fazit

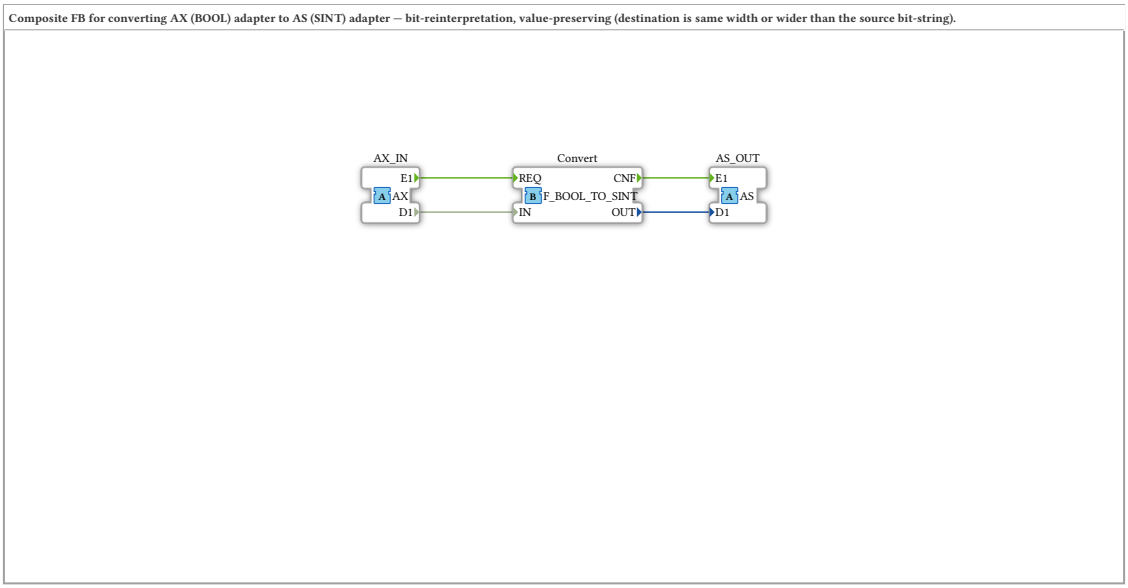
Der Baustein **AS_TO_AX** stellt eine kompakte und zuverlässige Lösung zur Umwandlung eines SINT-Adapter-Signals in ein BOOL-Adapter-Signal dar. Durch die Nutzung des IEC 61131-Standardbausteins `F_NE` ist das Verhalten deterministisch und leicht nachvollziehbar. Der FB eignet sich besonders für Anwendungen, bei denen eine einfache Null-Prüfung über Adaptergrenzen hinweg erforderlich ist.

AX_TO_AS

BOOL Adapter Input – *AX*



AS – SINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AX_TO_AS` ist ein Composite-FB, der einen BOOL-Adapter (Typ AX) in einen SINT-Adapter (Typ AS) umwandelt. Er kapselt die Konvertierung eines binären Signals in einen vorzeichenbehafteten 8-Bit-Wert und stellt diese über eine standardisierte Adapterschnittstelle bereit.

Schnittstellenstruktur

Ereignis-Eingänge

Keine separaten Ereignis-Eingänge; die Ereignissteuerung erfolgt über den AX_IN-Adapter (siehe Adapter).

Ereignis-Ausgänge

Keine separaten Ereignis-Ausgänge; die Ereignisausgabe erfolgt über den AS_OUT-Adapter (siehe Adapter).

Daten-Eingänge

Keine separaten Daten-Eingänge; die Datenübergabe erfolgt über den AX_IN-Adapter (siehe Adapter).

Daten-Ausgänge

Keine separaten Daten-Ausgänge; die Datenausgabe erfolgt über den AS_OUT-Adapter (siehe Adapter).

Adapter

- **AX_IN** (Socket): Typ `adapter::types::unidirectional::AX` (BOOL). Stellt das binäre Eingangssignal über den Ereignis-Port `E1` und den Daten-Port `D1` bereit.
- **AS_OUT** (Plug): Typ `adapter::types::unidirectional::AS` (SINT). Gibt das konvertierte SINT-Signal über den Ereignis-Port `E1` und den Daten-Port `D1` aus.

Funktionsweise

Der FB besitzt ein internes Netzwerk, das aus einer Instanz des Standardbausteins `F_BOOL_TO_SINT` besteht.

1. Ein eingehendes Ereignis am AX_IN-Adapter (`E1`) triggert die Konvertierung.
2. Der Datenwert von `AX_IN.D1` (BOOL) wird an den Eingang `IN` des Konvertierungsbausteins übergeben.
3. Der Konvertierungsbaustein wandelt den BOOL-Wert in einen SINT-Wert um (`TRUE` ⟶ 1, `FALSE` ⟶ 0).
4. Das Ergebnis wird am Ausgang `OUT` bereitgestellt und an den AS_OUT-Adapter (`D1`) weitergeleitet.
5. Ein Bestätigungsereignis wird über `AS_OUT.E1` ausgegeben.

Technische Besonderheiten

- **Bit-Reinterpretation, werterhaltend** (Zielbreite ist gleich groß oder größer als der Quell-Bit-String).
- **Composite-Struktur**: Der FB ist als Composition realisiert, wodurch die interne Logik transparent und wiederverwendbar ist.
- **Standard-Konvertierung**: Verwendet den genormten FB `iec61131::conversion::F_BOOL_TO_SINT` aus der IEC-61131-Bibliothek.
- **Adapter-Schnittstelle**: Die Ein- und Ausgabe erfolgt ausschließlich über unidirektionale Adapter (Sockets/Plugs), was eine modulare Verbindung in Adapter-basierten Architekturen ermöglicht.

Zustandsübersicht

Der FB besitzt keine explizit modellierten Zustände. Das Verhalten ist rein ereignisgesteuert: Jedes Ereignis am Eingangsadapter löst eine Konvertierung aus, nach deren Abschluss ein Ausgangsereignis gesendet wird.

Anwendungsszenarien

- **Binäre Sensoranbindung**: Ein binärer Sensor (z. B. Endschalter) wird über einen AX-Adapter eingelesen und als SINT-Wert an eine Steuerung weitergegeben.
- **Protokollanpassung**: In Systemen, die SINT-basierte Adapter erfordern, können vorhandene BOOL-Adapter ohne zusätzliche Codierung angebunden werden.
- **Testumgebungen**: Einfaches Umschalten zwischen BOOL- und SINT-Signalen zur Überprüfung von Adapter-Kommunikationswegen.

Vergleich mit ähnlichen Bausteinen

- **Direkte Konvertierungen (`F_BOOL_TO_SINT`)**: Diese arbeiten auf Datenebene ohne Adapter. Der `AX_TO_AS` kapselt die Konvertierung in eine adapterbasierte Schnittstelle und ist daher leichter in Adapter-Netzwerke integrierbar.
- **Andere Adapterkonverter (z. B. `AS_TO_AX`)**: Bilden die umgekehrte Richtung ab; `AX_TO_AS` ergänzt diese als Gegenstück.

Fazit

Der `AX_TO_AS` ist ein einfacher, aber nützlicher Composite-FB für die Umwandlung von BOOL-Adapter-Signalen in SINT-Adapter-Signale. Durch die Verwendung eines standardisierten Konvertierungsbausteins

und der Adapter-Technologie lässt er sich flexibel in bestehende 4diac-Projekte einfügen und trägt zur sauberen Trennung von Signaltypen bei.

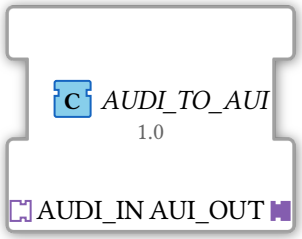
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

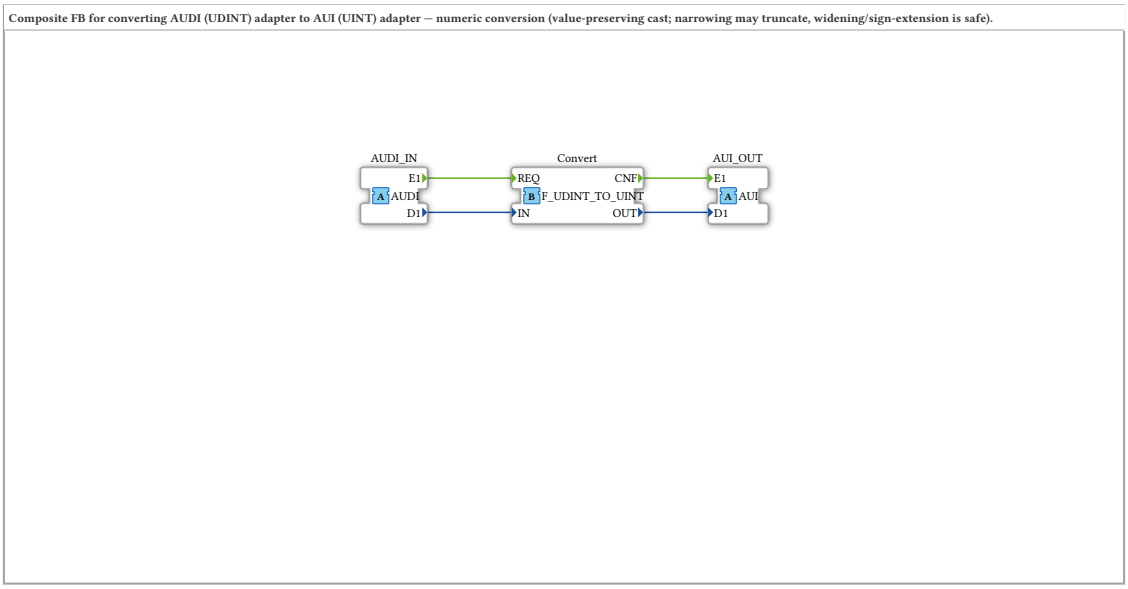
AUDI_AUI

AUDI_TO_AUI

UDINT Adapter Input – *AUDI*



AUI – UINT Adapter Output



Kein Bild vorhanden.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AUDI_TO_AUI` ist ein Composite-Baustein, der einen AUDI-Adapter (Datentyp UDINT) in einen AUI-Adapter (Datentyp UINT) umwandelt. Er dient als einfache Schnittstellenanpassung zwischen Komponenten, die unterschiedliche ganzzahlige Datentypen für ihre unidirektionalen Adapter verwenden. Der Baustein kapselt die Konvertierung und ermöglicht eine nahtlose Integration in IEC 61499-Netzwerke.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keinen direkten Ereignis-Eingang auf der oberen Ebene. Ereignisse werden über den Socket `AUDI_IN` empfangen:

- **E1** – Auslöser für die Konvertierung eines neuen Datenwerts.

Ereignis-Ausgänge

Der FB besitzt keinen direkten Ereignis-Ausgang auf der oberen Ebene. Ereignisse werden über den Plug `AUI_OUT` gesendet:

- **E1** – Bestätigung, dass der konvertierte Wert am Ausgang anliegt.

Daten-Eingänge

Der FB besitzt keinen direkten Daten-Eingang auf der oberen Ebene. Daten werden über den Socket `AUDI_IN` empfangen:

- **D1** (`UDINT`) – Der umzuwandelnde Eingangswert.

Daten-Ausgänge

Der FB besitzt keinen direkten Daten-Ausgang auf der oberen Ebene. Daten werden über den Plug `AUI_OUT` ausgegeben:

- **D1** (`UINT`) – Der konvertierte Ausgangswert.

Adapter

- **Socket `AUDI_IN`** – Unidirektionaler Eingangsadapter vom Typ `adapter::types::unidirectional::AUDI`. Er liefert ein Ereignis `E1` und einen Datenwert `D1` (Typ `UDINT`).
- **Plug `AUI_OUT`** – Unidirektionaler Ausgangsadapter vom Typ `adapter::types::unidirectional::AUI`. Er erwartet ein Ereignis `E1` und gibt einen Datenwert `D1` (Typ `UINT`) aus.

Funktionsweise

Der Baustein ist als Composite realisiert und enthält intern den Funktionsblock `iec61131::conversion::F_UDINT_TO_UINT`. Die Ablaufsteuerung erfolgt ereignisgesteuert:

1. Ein Ereignis am Socket `AUDI_IN.E1` triggert den Eingang `REQ` des internen Konvertierungsbausteins.
2. Gleichzeitig wird der Datenwert `AUDI_IN.D1` an den Eingang `IN` des Konverters übergeben.
3. Der Konverter führt die Wandlung von `UDINT` nach `UINT` durch.
4. Nach Abschluss sendet der Konverter ein Ereignis an seinem Ausgang `CNF`, welches auf den Plug `AUI_OUT.E1` weitergeleitet wird.
5. Der konvertierte Wert am Ausgang `OUT` des Konverters wird auf den Datenausgang `AUI_OUT.D1` gelegt.

Die gesamte Konvertierung erfolgt innerhalb eines einzigen Ereignisdurchlaufs (synchron).

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-Architektur:** Der FB ist als Composite entworfen, d. h. seine Funktionalität wird vollständig durch das interne Netzwerk bestimmt. Dies erleichtert die Wiederverwendung und Anpassung.
- **Standard-Konvertierungsbaustein:** Die eigentliche Typumwandlung verwendet den IEC 61131-Baustein `F_UDINT_TO_UINT`, der eine plattformunabhängige und getestete Implementierung bereitstellt.
- **Lizenz:** Der Baustein steht unter der Eclipse Public License 2.0 (EPL-2.0).
- **Version:** 1.0, erstellt am 17.02.2026.

Zustandsübersicht

Der FB besitzt keine eigenen Zustände. Da das Composite-Netzwerk nur aus einem einzigen funktionalen Baustein besteht, der seinerseits zustandslos arbeitet, verhält sich `AUDI_TO_AUI` ebenfalls zustandslos. Jeder Ereignisimpuls führt sofort zu einer einmaligen Konvertierung.

Anwendungsszenarien

- **Anbindung von Sensoren/Aktoren:** Ein Sensor liefert Werte als `UDINT` über einen AUDI-Adapter, während die nachfolgende Verarbeitungskette `UINT` über AUI erwartet.
- **Systemintegration:** In heterogenen Automatisierungssystemen treten häufig unterschiedliche Datentypen auf. Der Baustein dient als Adapter, um Komponenten verschiedener Hersteller oder Bibliotheken zu verbinden.
- **Prototypenentwicklung:** Einfache und schnelle Anpassung von Schnittstellen ohne manuelle Konvertierungslogik.

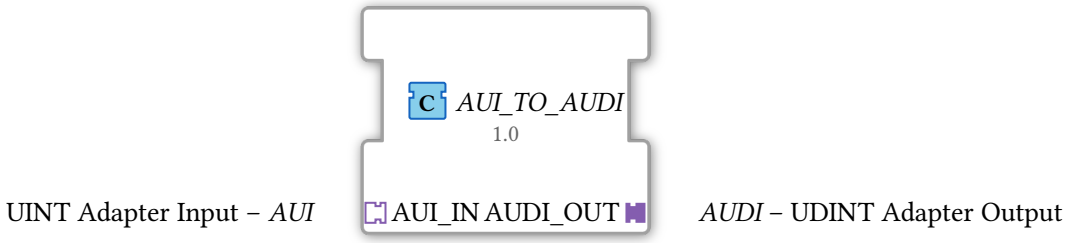
Vergleich mit ähnlichen Bausteinen

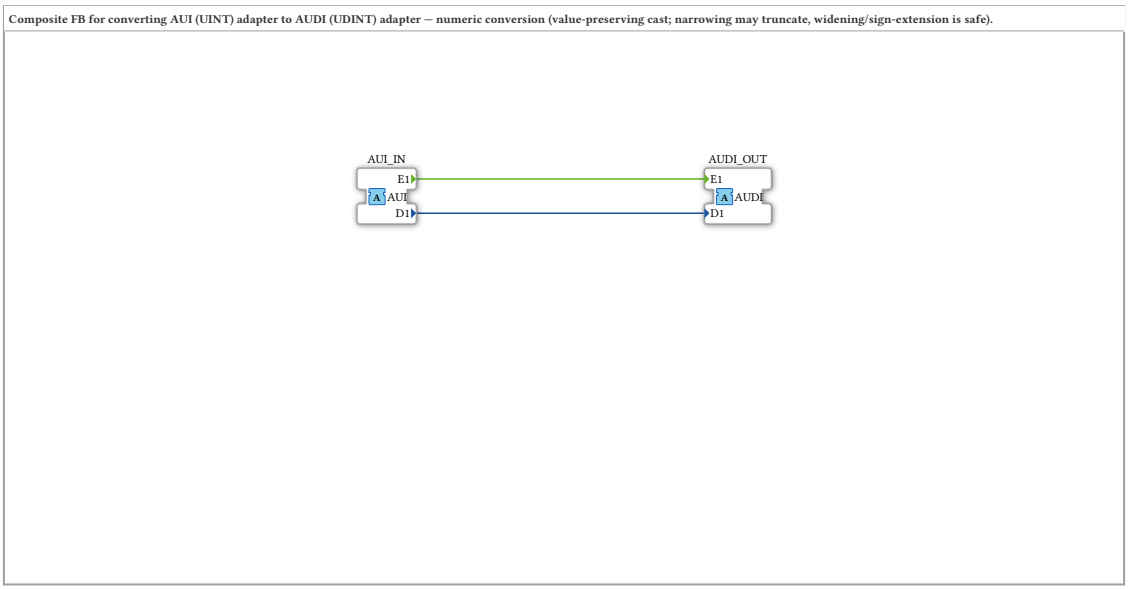
- **`AUDI_TO_XXX`-Bausteine:** Es existieren analoge Bausteine für andere Zieltypen (z. B. `AUDI_TO_DINT`, `AUDI_TO_SINT`). `AUDI_TO_AUI` ist spezifisch für die Umwandlung nach `UINT`.
- **Direkte Adapter-Kopplung:** Ohne Konvertierungsbaustein müsste die Typumwandlung in der Applikation selbst oder durch einen generischen Funktionsbaustein erfolgen. `AUDI_TO_AUI` bietet eine gekapselte, wiederverwendbare Lösung.

Fazit

Der `AUDI_TO_AUI`-Funktionsblock stellt eine kompakte und zuverlässige Möglichkeit dar, zwischen zwei unidirektionalen Adaptern mit unterschiedlichen Datentypen zu vermitteln. Durch die Composite-Struktur und die Nutzung eines standardisierten Konvertierungsbausteins ist er wartungsarm und einfach in bestehende IEC 61499-Projekte integrierbar.

AUI_TO_AUDI





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUI_TO_AUDI** ist ein zusammengesetzter (Composite) Baustein, der einen AUI-Adapter (UINT) in einen AUDI-Adapter (UDINT) umwandelt. Er dient dazu, Daten und Ereignisse von einem Adaptertyp auf einen anderen zu übertragen, wobei die zugrundeliegenden Datentypen unterschiedlich sind (UINT → UDINT). Die Umwandlung erfolgt durch einfaches Durchreichen der Signale, ohne zusätzliche Logik.

Schnittstellenstruktur

Ereignis-Eingänge

Keine eigenen Ereignis-Eingänge: Der Baustein besitzt keine direkten Ereignis-Eingänge. Ereignisse werden über den angeschlossenen Socket-Adapter **AUI_IN** empfangen.

Ereignis-Ausgänge

Keine eigenen Ereignis-Ausgänge: Der Baustein besitzt keine direkten Ereignis-Ausgänge. Ereignisse werden über den angeschlossenen Plug-Adapter **AUDI_OUT** gesendet.

Daten-Eingänge

Keine eigenen Daten-Eingänge: Der Baustein besitzt keine direkten Daten-Eingänge. Daten werden über den angeschlossenen Socket-Adapter **AUI_IN** empfangen.

Daten-Ausgänge

Keine eigenen Daten-Ausgänge: Der Baustein besitzt keine direkten Daten-Ausgänge. Daten werden über den angeschlossenen Plug-Adapter **AUDI_OUT** gesendet.

Adapter

Name	Typ	Richtung	Kommentar
AUI_IN	`adapter::types::unidirectional::AUI`	Socket (Eingang)	Empfängt UINT-Daten und zugehöriges Ereignis
AUDI_OUT	`adapter::types::unidirectional::AUDI`	Plug (Ausgang)	Sendet UDINT-Daten und zugehöriges Ereignis

Funktionsweise

Der Baustein verbindet intern die Schnittstellen des eingehenden AUI-Adapters mit denen des ausgehenden AUDI-Adapters:

- **Ereignisverbindung**: Das Ereignis `E1` des AUI-Adapters wird direkt an das Ereignis `E1` des AUDI-Adapters weitergeleitet.
- **Datenverbindung**: Der Datenwert `D1` des AUI-Adapters (Typ UINT) wird direkt an den Datenwert `D1` des AUDI-Adapters (Typ UDINT) übergeben. Dabei erfolgt eine implizite Typkonvertierung von UINT (16 Bit) nach UDINT (32 Bit).

Der Baustein führt keine eigene Logik oder Umrechnung durch; er fungiert als reine Durchreiche für Adapter mit unterschiedlichen Datentypen.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional, d.h. Daten und Ereignisse fließen nur in eine Richtung (vom Socket zum Plug).
- **Composite-FB**: Der Baustein ist als Composite (zusammengesetzter FB) realisiert, der die Adapterverbindungen im FBNetwork abbildet.
- **Implizite Typkonvertierung**: Die Konvertierung von UINT zu UDINT erfolgt automatisch im Rahmen der Datenverbindung, basierend auf den IEC 61499-Datentypregeln.
- **Paketstruktur**: Der Baustein gehört zum Paket `adapter::conversion::unidirectional`.

Zustandsübersicht

Der Baustein besitzt keine eigenen Zustände oder Zustandsautomaten. Das Verhalten ergibt sich ausschließlich aus der weiterleitenden Verbindung der Adapter-Schnittstellen.

Anwendungsszenarien

- **Adapteranpassung**: Wenn in einem System ein AUI-Adapter (UINT) verwendet wird, aber ein AUDI-Adapter (UDINT) erwartet wird, kann dieser Baustein als Konverter zwischengeschaltet werden.
- **Signalweiterleitung**: Einsatz in hierarchischen Systemen, wo ein unidirektionaler Datenfluss von einem kleineren auf einen größeren Datentyp abgebildet werden muss.
- **Prototypische Integration**: Schnelle Kopplung von Komponenten mit unterschiedlichen Adaptertypen ohne zusätzliche Logik.

Vergleich mit ähnlichen Bausteinen

- **AUI_TO_AUDI** führt eine reine Typkonvertierung durch, ohne weitere Verarbeitung.
- Ähnliche Bausteine wie `AUDI_TO_AUI` würden die umgekehrte Richtung (UDINT → UINT) abbilden, möglicherweise mit Datenüberlaufbehandlung.
- Andere Adapter-Konverter (z.B. `BOOL_TO_BYTE`) folgen dem gleichen Muster, unterscheiden sich jedoch in den verwendeten Datentypen und der Richtung.

Fazit

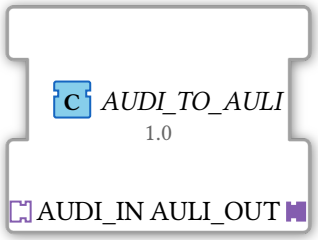
Der **AUI_TO_AUDI**-Baustein ist ein einfacher, aber nützlicher Composite-FB zur Konvertierung eines UINT-basierten AUI-Adapters in einen UDINT-basierten AUDI-Adapter. Er eignet sich besonders für Situationen, in denen eine unidirektionale Signalweiterleitung mit impliziter Datentypanpassung erforderlich ist, ohne dass eine eigene Verarbeitungslogik notwendig wird. Dank seiner klaren Struktur ist er leicht in bestehende 4diac-Projekte integrierbar.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUDI_AULI

AUDI_TO_AULI



UDINT Adapter Input – *AUDI* *AULI* – ULINT Adapter Output

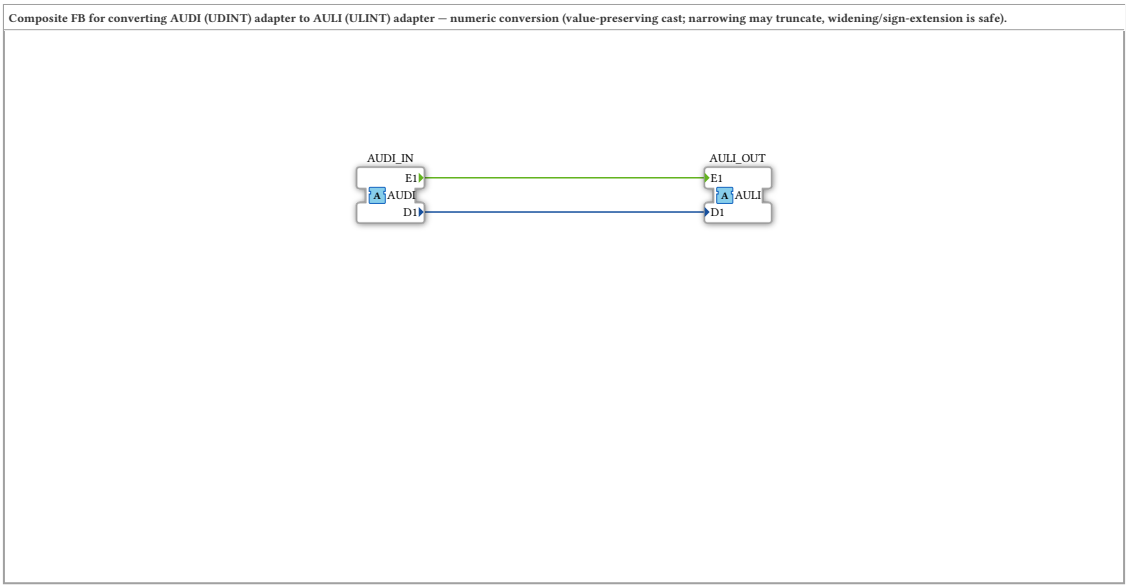


Bild nicht vorhanden

• – ◦ • – ◦ • – ◦ •

Einleitung

Dieser Composite FB konvertiert einen AUDI-Adapter (UDINT) in einen AULI-Adapter (ULINT). Er dient als Schnittstelle zwischen Komponenten, die unterschiedliche Adaptertypen für vorzeichenlose Ganzzahlen verwenden. Der Baustein steht unter der Eclipse Public License 2.0 (EPL 2.0) und wird von der HR Agrartechnik GmbH bereitgestellt.

Schnittstellenstruktur

Ereignis-Eingänge
Keine (Ereignisse werden über den AUDI_IN-Adapter empfangen).

Ereignis-Ausgänge
Keine (Ereignisse werden über den AULI_OUT-Adapter gesendet).

Daten-Eingänge
Keine (Daten werden über den AUDI_IN-Adapter empfangen).

Daten-Ausgänge
Keine (Daten werden über den AULI_OUT-Adapter gesendet).

Adapter
- **AUDI_IN** (Socket): Empfängt einen AUDI-Adapter (Typ: `adapter::types::unidirectional::AUDI`). Enthält einen Ereigniseingang (E1) und einen Dateneingang (D1) vom Typ UDINT.
- **AULI_OUT** (Plug): Sendet einen AULI-Adapter (Typ: `adapter::types::unidirectional::AULI`). Enthält einen Ereignisausgang (E1) und einen Datenausgang (D1) vom Typ ULINT.

Funktionsweise
Der FB leitet eingehende Ereignisse und Daten vom AUDI_IN-Adapter direkt zum AULI_OUT-Adapter weiter:

- Das Ereignis `E1` von `AUDI_IN` wird auf `E1` von `AULI_OUT` übertragen.
- Der Datenwert `D1` (UDINT) von `AUDI_IN` wird auf `D1` (ULINT) von `AULI_OUT` übertragen. Dabei wird der 32-Bit-UDINT-Wert implizit in den 64-Bit-ULINT-Wert umgewandelt (null-extension). Es findet keine zusätzliche Logik statt.

Technische Besonderheiten
- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite FB**: Enthält keine eigene Funktionsbausteinlogik oder Zustandsmaschine, sondern lediglich Verdrahtung zwischen zwei Adaptern.
- **Paket**: `adapter::conversion::unidirectional`
- **Version**: 1.0, erstellt am 2026-02-17.
- **Lizenz**: EPL 2.0.

Zustandsübersicht
Der FB besitzt keinen internen Zustand oder Algorithmus. Er arbeitet streng durchflussorientiert und reagiert auf jedes eingehende Ereignis sofort.

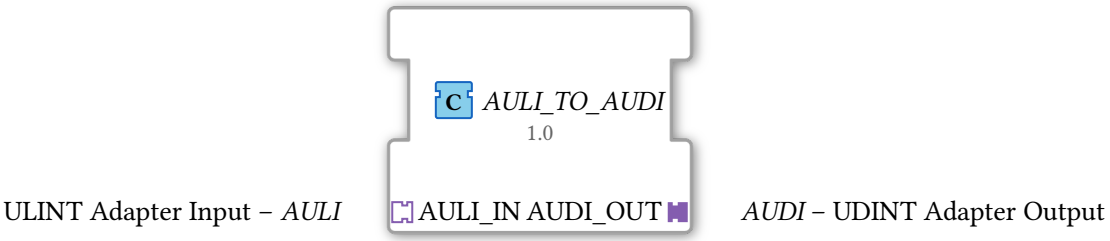
Anwendungsszenarien
- **Systemintegration**: Wenn ein Systemkomponente Daten über einen AUDI-Adapter (UDINT) bereitstellt, eine andere Komponente jedoch einen AULI-Adapter (ULINT) erwartet, fungiert dieser FB als Brücke.
- **Protokollanpassung**: Einsatz in Steuerungssystemen, die unterschiedliche Integer-Breiten für Busse oder Schnittstellen verwenden (z. B. 32-Bit zu 64-Bit).

Vergleich mit ähnlichen Bausteinen
- **Einfache Datentypkonverter** (z. B. `UDINT_TO_ULINT`): Diese wandeln nur Daten um, ohne Adapter. Der vorliegende FB arbeitet auf Adapterebene und leitet auch Ereignisse weiter.
- **Adapter-Transceiver**: Andere Bausteine könnten komplexere Umwandlungen (z. B. skalieren, offset) durchführen. Dieser FB ist minimal gehalten.

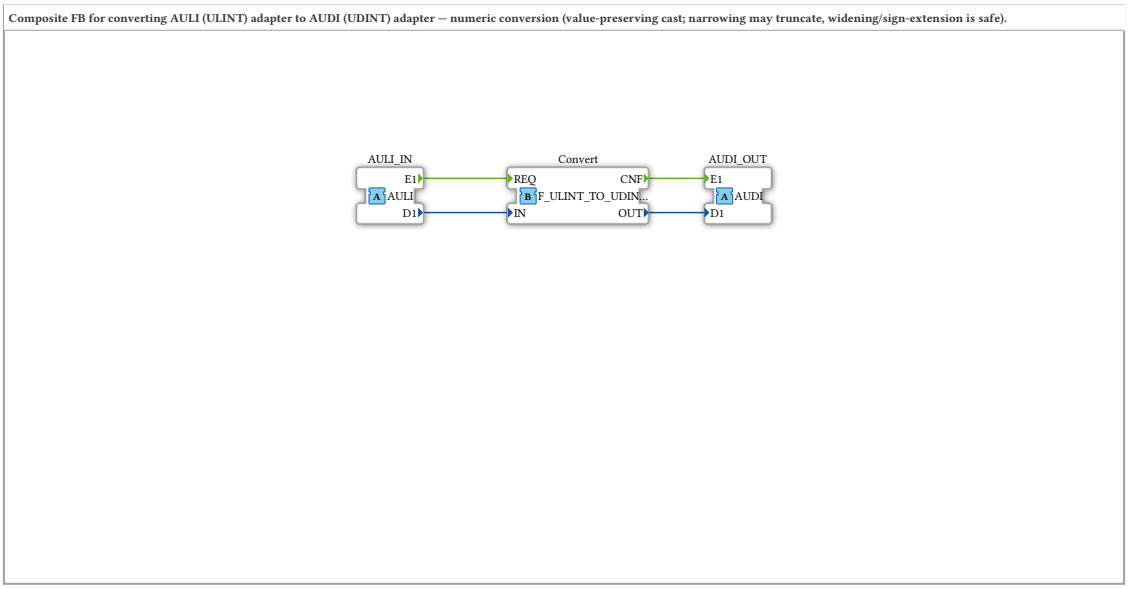
Fazit

`AUDI_TO_AULI` ist ein zweckmäßiger Composite FB zur verlustfreien Konvertierung von UDINT nach ULINT auf Adapterebene. Er ist einfach, leicht verständlich und ideal für saubere Schnittstellentrennung in modularen Systemen.

AULI_TO_AUDI



ULINT Adapter Input – *AULI* *AUDI* – UDINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AULI_TO_AUDI** ist ein Composite-Baustein, der die Konvertierung eines AULI-Adapters (Daten vom Typ ULINT) zu einem AUDI-Adapter (Daten vom Typ UDINT) realisiert. Er dient als unidirektionale Brücke zwischen zwei unterschiedlichen Adaptertypen und ermöglicht die nahtlose Integration von Komponenten, die auf verschiedenen Integer-Breiten basieren.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine separaten Ereignis-Eingänge. Das erforderliche Ereignis wird über den Socket-Adapter empfangen:

- **E1** (im Socket `AULI_IN`): Startet die Konvertierung eines ULINT-Wertes.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge vorhanden. Das Ereignis wird über den Plug-Adapter ausgegeben:

- **E1** (im Plug `AUDI_OUT`): Signalisiert den Abschluss der Konvertierung und die Bereitstellung des UDINT-Wertes.

Daten-Eingänge

Daten werden ausschließlich über den Socket-Adapter bereitgestellt:

- **D1** (im Socket `AULI_IN`): ULINT-Eingabewert (64-Bit unsigned integer).

Daten-Ausgänge

Daten werden über den Plug-Adapter ausgegeben:

- **D1** (im Plug `AUDI_OUT`): UDINT-Ausgabewert (32-Bit unsigned integer).

Adapter

- **Socket AULI_IN** Typ: `adapter::types::unidirectional::AULI` Empfängt den zu konvertierenden ULINT-Wert und das zugehörige Ereignis.
- **Plug AUDI_OUT** Typ: `adapter::types::unidirectional::AUDI` Stellt den konvertierten UDINT-Wert und das Abschluss-Ereignis bereit.

Funktionsweise

Der FB enthält ein internes Netzwerk mit einem einzigen Konvertierungsbaustein `F_ULINT_TO_UDINT` (aus der IEC 61131-Bibliothek).

1. Ein eingehendes Ereignis **E1** vom Socket `AULI_IN` triggert den Konvertierungsbaustein über dessen `REQ`-Event.
2. Gleichzeitig wird der ULINT-Wert von `AULI_IN.D1` an den Eingang `IN` des Konverters weitergeleitet.
3. Der Konverter wandelt den 64-Bit-Wert in einen 32-Bit-Wert um und legt das Ergebnis an seinem Ausgang `OUT` an.
4. Nach erfolgreicher Konvertierung sendet der Konverter ein Bestätigungsereignis (`CNF`), das als **E1** an den Plug `AUDI_OUT` weitergegeben wird.
5. Der umgewandelte UDINT-Wert wird an `AUDI_OUT.D1` ausgegeben.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-FB**: Der Baustein selbst besitzt keine eigene Logik, sondern nutzt einen standardisierten IEC 61131-Konverter.
- **Adapter-Konvertierung**: Ermöglicht die Verbindung von Komponenten, die auf unterschiedlichen Adapterdefinitionen basieren, ohne dass der Anwender die Konvertierung manuell vornehmen muss.
- **Unidirektionale Datenfluss**: Die Konvertierung erfolgt nur in eine Richtung (ULINT → UDINT); eine Rückkonvertierung ist nicht integriert.
- **Keine Datenbereichsüberprüfung**: Da UDINT nur 32 Bit umfasst, gehen bei ULINT-Werten > 2^{32}-1 die höherwertigen Bits verloren.

Zustandsübersicht

Der Composite-FB besitzt keine eigene Zustandsmaschine. Das Verhalten wird vollständig durch den internen Konvertierungsbaustein `F_ULINT_TO_UDINT` bestimmt, der typischerweise einen einfachen Warte-/Verarbeitungszustand implementiert. Nach dem Start (`REQ`) führt er die Konvertierung aus und quittiert diese mit `CNF`.

Anwendungsszenarien

- **Adapter-Brücken**: Wenn ein System einen AULI-Adapter (z. B. für große Zähler oder Zeitstempel) bereitstellt, eine nachgelagerte Komponente jedoch einen AUDI-Adapter erwartet (z. B. für kleinere Steuerwerte).
- **Protokoll- oder Schnittstellenanpassung**: In heterogenen Automatisierungsnetzwerken, in denen unterschiedliche Integer-Größen verwendet werden.
- **Datenreduktion**: Wenn nur die unteren 32 Bit einer 64-Bit-Größe relevant sind (z. B. bei bestimmten Sensorwerten).

Vergleich mit ähnlichen Bausteinen

- **Direkte Typumwandlungen** (z. B. `F_ULINT_TO_UDINT`): Diese arbeiten auf reinem Datenebene ohne Adapter. AULI_TO_AUDI kapselt die Umwandlung in eine adapterbasierte Schnittstelle, was den Austausch zwischen verschiedenen Adaptertypen erleichtert.

- **Bidirektionale Konverter**: Es existieren auch Bausteine für die Umkehrung (UDINT ⟶ ULINT). Der vorliegende FB ist auf eine Richtung spezialisiert.
- **Composite-Adapter**: Andere Composite-FBs können mehrere Konvertierungen oder zusätzliche Logik (z. B. Bereichsprüfung) enthalten – hier liegt der Fokus auf Einfachheit und Wiederverwendbarkeit.

Fazit

Der AULI_TO_AUDI ist ein kompakter, spezialisierter Composite-Baustein, der die Umwandlung eines AULI- (ULINT) in einen AUDI-Adapter (UDINT) sauber kapselt. Dank der Integration des standardisierten Konverters ist er zuverlässig und einfach in bestehende 4diac-Projekte einbindbar. Er eignet sich ideal für Szenarien, in denen unterschiedliche Adapterdefinitionen aufeinandertreffen, und trägt zur Modularität und Wiederverwendbarkeit von Funktionsbausteinen bei.

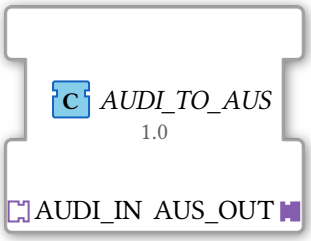
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

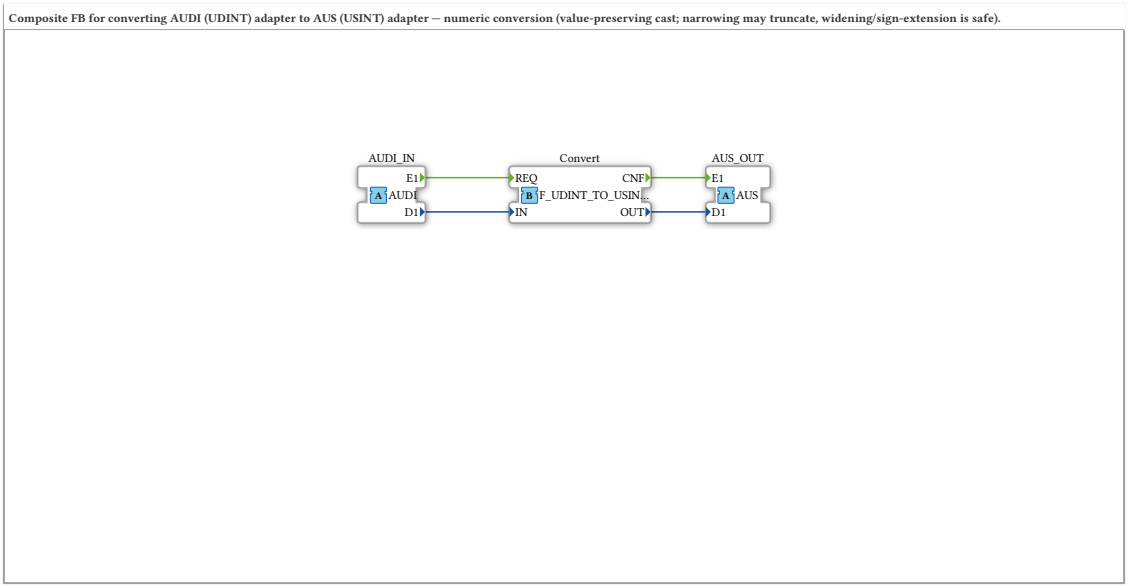
AUDI_AUS

AUDI_TO_AUS

UDINT Adapter Input – *AUDI*



AUS – USINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **AUDI_TO_AUS** dient als Komposit-Baustein zur Umwandlung eines UDINT-Wertes (über den AUDI-Adapter) in einen USINT-Wert (über den AUS-Adapter). Er kapselt die Konvertierung und erlaubt eine einfache Anbindung von Komponenten, die unterschiedliche Datentypen erwarten.

Schnittstellenstruktur

Ereignis-Eingänge

- **AUDI_IN.E1** (Ereignis vom AUDI-Adapter) – initiiert die Konvertierung des anliegenden Datenwertes.

Ereignis-Ausgänge

- **AUS_OUT.E1** (Ereignis zum AUS-Adapter) – signalisiert den Abschluss der Konvertierung und die Bereitstellung des Ergebnisses.

Daten-Eingänge

- **AUDI_IN.D1** (UDINT aus dem AUDI-Adapter) – der zu konvertierende 32-Bit-Wert.

Daten-Ausgänge

- **AUS_OUT.D1** (USINT zum AUS-Adapter) – der konvertierte 8-Bit-Wert.

Adapter

- **AUDI_IN** (Socket) – Adapter vom Typ `adapter::types::unidirectional::AUDI` (UDINT-Quelle).
- **AUS_OUT** (Plug) – Adapter vom Typ `adapter::types::unidirectional::AUS` (USINT-Senke).

Funktionsweise

Der Baustein ist als reines „Durchreichen“ implementiert. Im FBNetzwerk wird der eingehende Ereignisimpuls (`AUDI_IN.E1`) direkt an den internen Konvertierungsbaustein `F_UDINT_TO_USINT` weitergeleitet. Gleichzeitig wird der Datenwert `AUDI_IN.D1` an den Eingang `IN` des Konverters übergeben. Nach erfolgter Umwandlung sendet der Konverter das Bestätigungsereignis (`CNF`) an den Ausgangs-Adapter (`AUS_OUT.E1`) und das Ergebnis (`OUT`) an den Datenausgang (`AUS_OUT.D1`). Es findet keine Zwischenspeicherung oder weitere Logik statt.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- Verwendet den Standard-Konvertierungsbaustein `iec61131::conversion::F_UDINT_TO_USINT` aus der IEC-61131-Bibliothek.
- Reine Ereignis- und Datenweiterleitung – keine eigene Zustandsmaschine erforderlich.
- Die Konvertierung ist verlustbehaftet: Ein UDINT (32-Bit) wird auf einen USINT (8-Bit) abgebildet, d.h. höherwertige Bits gehen verloren (Werte > 255 werden abgeschnitten oder nach Herstellerdefinition behandelt).

Zustandsübersicht

Der Composite-Baustein besitzt keine eigenen Zustände; er ist vollständig ereignisgetrieben. Der interne Konverter `F_UDINT_TO_USINT` arbeitet nach dem Standard-Protokoll (REQ/CNF). Es existieren keine zusätzlichen Sperren oder Verzögerungen.

Anwendungsszenarien

- Anbindung eines UDINT-basierten Sensors an einen Aktor, der nur einen USINT-Wert akzeptiert.
- Umwandlung von Adressen oder Zählwerten, wenn nur der niederwertige Teil benötigt wird.
- Schnittstellenanpassung in einer gemischten IEC-61499 Umgebung, in der verschiedene Datentypen verwendet werden.

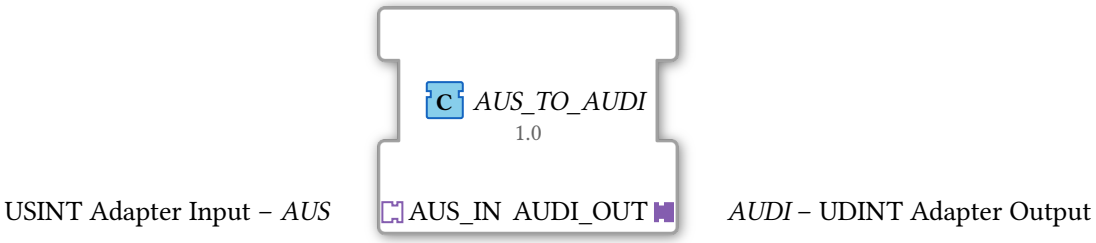
Vergleich mit ähnlichen Bausteinen

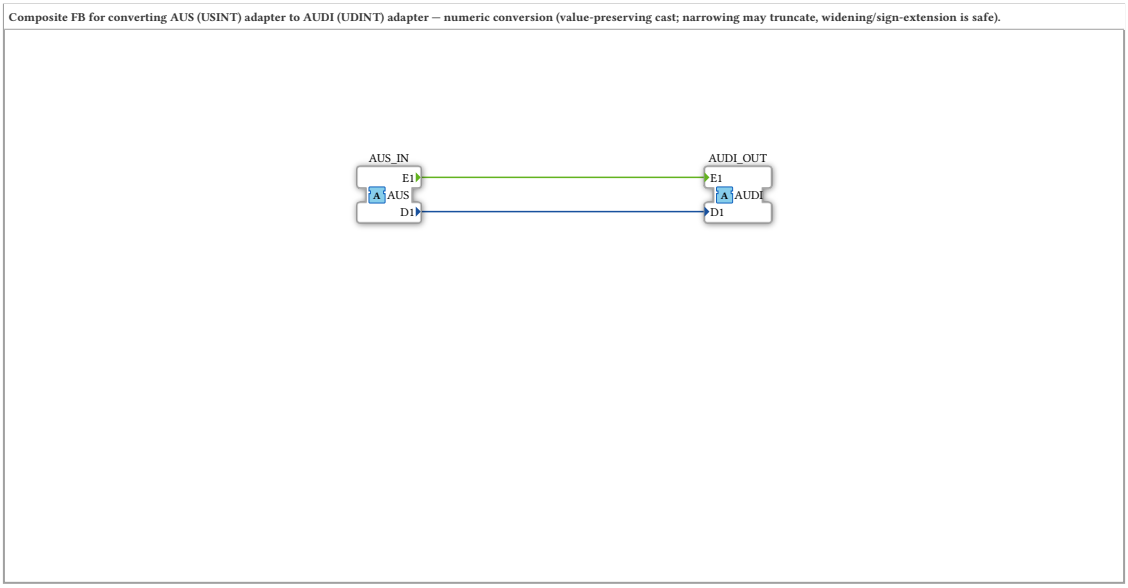
- **AUDI_TO_AUDI** oder **AUS_TO_AUS**: Würden keine Konvertierung durchführen, sondern nur eine reine Weiterleitung.
- **F_UDINT_TO_USINT** (direkt): Dieser Baustein führt die gleiche Konvertierung durch, bietet aber keine Adapter-Schnittstelle. `AUDI_TO_AUS` kapselt die Adapter-Typen und vereinfacht die Netzwerkstruktur.
- Bei Bedarf einer Bereichsüberprüfung oder Fehlerbehandlung müsste ein eigener Baustein erstellt werden; dieser Baustein führt keine Plausibilitätsprüfung durch.

Fazit

`AUDI_TO_AUS` ist ein einfacher, aber nützlicher Strip-Down-Baustein für die unidirektionale UDINT-zu-USINT-Konvertierung über Adapter. Er reduziert die Komplexität im FBNetzwerk, indem er die Adapter-Kopplung und die Konvertierungslogik in einem Block vereint. Aufgrund der fehlenden Fehlerbehandlung ist er nur für Anwendungen geeignet, bei denen Werte > 255 nicht vorkommen oder deren Verlust toleriert werden kann.

AUS_TO_AUDI





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Composite-Funktionsblock **AUS_TO_AUDI** dient der Umwandlung eines unidirektionalen Adapters vom Typ **AUS** (basierend auf USINT-Werten) in einen Adapter vom Typ **AUDI** (basierend auf UDINT-Werten). Er wird eingesetzt, um Daten zwischen zwei verschiedenen Adapter-Schnittstellen zu übersetzen, ohne dass die zugrunde liegende Datenlogik verändert wird. Der Baustein ist als reine Durchreich-Komponente realisiert, die Ereignisse und Daten direkt von der Eingangs- zur Ausgangsseite weiterleitet.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine eigenen Ereignis-Eingänge. Ereignisse werden über den angeschlossenen **Socket-Adapter** `AUS_IN` bereitgestellt.

Ereignis-Ausgänge

Der Baustein besitzt keine eigenen Ereignis-Ausgänge. Ereignisse werden über den angeschlossenen **Plug-Adapter** `AUDI_OUT` weitergegeben.

Daten-Eingänge

Der Baustein besitzt keine eigenen Daten-Eingänge. Daten werden über den **Socket-Adapter** `AUS_IN` entgegengenommen.

Daten-Ausgänge

Der Baustein besitzt keine eigenen Daten-Ausgänge. Daten werden über den **Plug-Adapter** `AUDI_OUT` ausgegeben.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AUS`	`AUS_IN`	Socket (Eingang)	Unidirektionaler Adapter, der Daten im USINT-Format (8 Bit) sowie ein Ereignis bereitstellt.
`adapter::types::unidirectional::AUDI`	`AUDI_OUT`	Plug (Ausgang)	Unidirektionaler Adapter, der Daten im UDINT-Format (32 Bit) sowie ein Ereignis erwartet.

Der interne Netzwerkaufbau verbindet das Ereignis `E1` und den Datenwert `D1` des Eingangsadapters direkt mit den gleichnamigen Schnittstellen des Ausgangsadapters.

Funktionsweise

Der Funktionsblock arbeitet als reine Durchreiche. Sobald am Socket `AUS_IN` das Ereignis `E1` eintritt, wird dieses unverzögert an den Plug `AUDI_OUT` als Ereignis `E1` weitergeleitet. Gleichzeitig wird der Datenwert `D1` (USINT) des Eingangsadapters auf den Datenwert `D1` (UDINT) des Ausgangsadapters übertragen. Eine Typumwandlung von USINT nach UDINT findet dabei implizit durch die Datenverbindung statt (Aufweitung von 8 Bit auf 32 Bit). Es erfolgt keine Validierung oder Wertebereichsprüfung.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Adapterbasierte Schnittstelle**: Der Baustein besitzt keine klassischen Ein-/Ausgangsvariablen, sondern verwendet ausschließlich die standardisierten Adapter-Typen `AUS` und `AUDI` aus dem Paket `adapter::types::unidirectional`.
- **Einfache Durchleitung**: Die Ereignis- und Datenverbindungen sind direkt und ohne zusätzliche Logik ausgeführt.
- **Typaufweitung**: Die Konvertierung von USINT (0–255) nach UDINT (0–4294967295) erfolgt verlustfrei, da der gesamte USINT-Wertebereich in den UDINT-Bereich abbildbar ist.
- **Keine Zustandsspeicherung**: Der Baustein ist zustandslos; jede Verarbeitung erfolgt in derselben Ausführungsinstanz.

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustandsautomaten. Die Funktionalität beschränkt sich auf eine direkte 1:1-Verbindung zwischen Eingangs- und Ausgangsadapter. Eine Zustandsbeschreibung ist daher nicht erforderlich.

Anwendungsszenarien

- **Adapterbrücke**: Einsatz in Systemen, die den AUS-Adapter (z. B. für einfache Byte-Werte) verwenden, aber eine AUDI-Schnittstelle (z. B. für 32-Bit-Zähler oder Adressen) erwarten.
- **Protokollumsetzung**: Verwendung innerhalb von Composite-FBs, um zwischen unterschiedlichen Adapterversionen einer Gerätefamilie zu vermitteln.
- **Datenaufbereitung**: Vorbereitung von USINT-Daten für Bausteine, die ausschließlich UDINT-Eingänge akzeptieren.

Vergleich mit ähnlichen Bausteinen

- **INT_TO_DINT**: Ein Standard-FB zur Konvertierung von INTEGER-Typen, jedoch mit eigenständigen Ein-/Ausgangs-Variablen. `AUS_TO_AUDI` arbeitet hingegen auf Adapterebene und ist für die Einbettung in adapterbasierte Architekturen optimiert.
- **AUS_TO_AUDI (einfach)**: Alternative Realisierungen könnten zusätzliche Überlaufprüfungen oder Filterfunktionen enthalten; dieser FB beschränkt sich auf die reine Durchleitung.

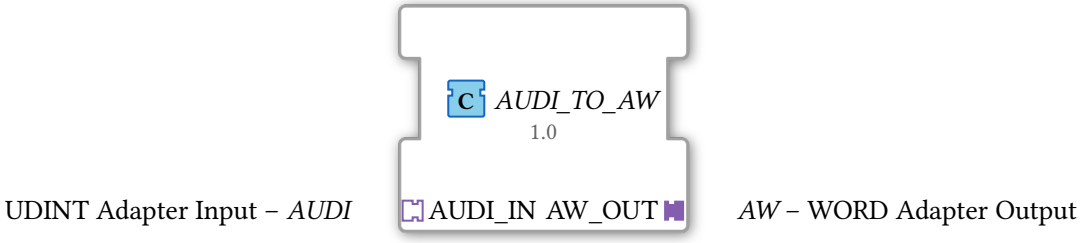
Fazit

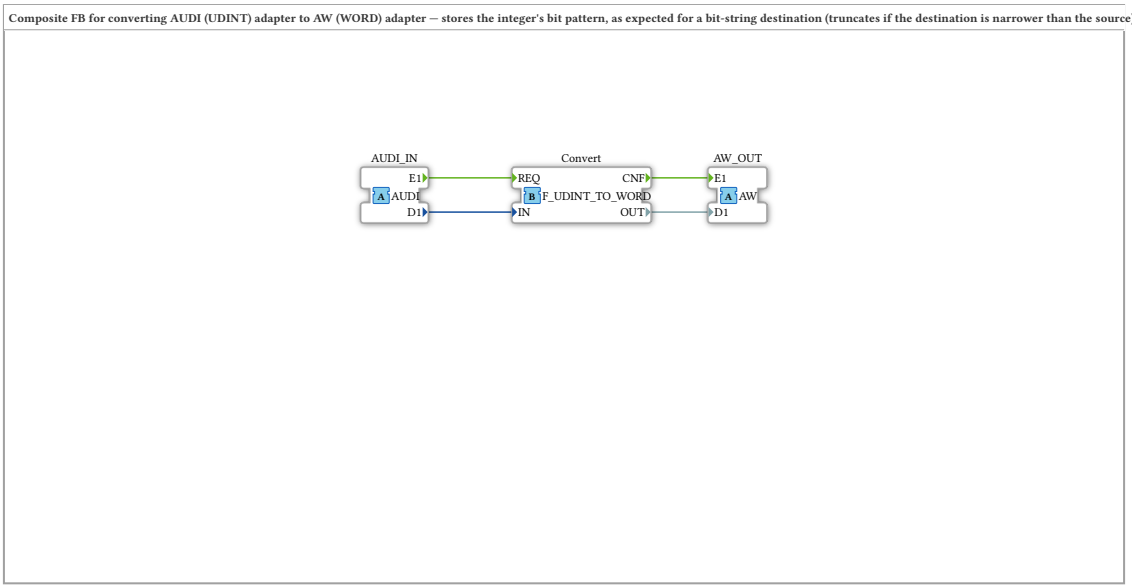
`AUS_TO_AUDI` ist ein kompakter, spezialisierter Composite-Funktionsblock zur verlustfreien Umwandlung zwischen den unidirektionalen Adaptern AUS (USINT) und AUDI (UDINT). Dank der direkten Durchleitung von Ereignis und Daten ohne Zusatzlogik eignet er sich ideal als Baustein in adapterbasierten

Steuerungsnetzwerken, bei denen eine einfache Typanpassung auf der Kommunikationsebene erforderlich ist.

AUDI_AW

AUDI_TO_AW





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUDI_TO_AW** ist ein Composite-Funktionsblock zur Konvertierung eines **AUDI-Adapter** (UDINT) in einen **AW-Adapter** (WORD). Er dient als unidirektionale Brücke zwischen unterschiedlichen Datentypen und ermöglicht die nahtlose Integration von Komponenten, die auf unterschiedlichen Adapter-Schnittstellen basieren.

Der Baustein kapselt die Konvertierungslogik und wird typischerweise in Automatisierungsprojekten nach IEC 61499 eingesetzt, die eine Anpassung von Datentypen zwischen Adaptern erfordern.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung
(über AUDI_IN.E1)	EVENT	Ereignis, das die Konvertierung anstößt (bereitgestellt durch den angeschlossenen AUDI-Adapter)

Ereignis-Ausgänge

Name	Typ	Beschreibung
(über AW_OUT.E1)	EVENT	Ausgangsereignis nach erfolgreicher Konvertierung (bereitgestellt durch den AW-Adapter)

Daten-Eingänge

Name	Typ	Beschreibung
(über AUDI_IN.D1)	UDINT	32-Bit unsigned Integer, der konvertiert werden soll (bereitgestellt durch den AUDI-Adapter)

Daten-Ausgänge

Name	Typ	Beschreibung
(über AW_OUT.D1)	WORD	16-Bit unsigned Integer (Ergebnis der Konvertierung, bereitgestellt durch den AW-Adapter)

Adapter

Name	Typ	Richtung	Beschreibung
AUDI_IN	adaper::types::unidirectional::AUDI	Socket (Eingang)	Empfängt das UDINT-Signal und das zugehörige Ereignis
AW_OUT	adaper::types::unidirectional::AW	Plug (Ausgang)	Gibt das konvertierte WORD-Signal und das Ausgangsereignis weiter

Funktionsweise

Der Funktionsblock arbeitet als **Composite FB**, der einen einzelnen IEC 61131-Konvertierungsbaustein `F_UDINT_TO_WORD` enthält. Die Verarbeitung erfolgt nach folgendem Ablauf:

1. Der Socket `AUDI_IN` empfängt ein Ereignis an seinem Ereigniseingang `E1` und den dazugehörigen Datenwert `D1` (UDINT).
2. Dieses Ereignis wird intern an den `REQ`-Eingang des Konvertierungsbausteins weitergeleitet, während der UDINT-Wert an den `IN`-Eingang übergeben wird.
3. Der Konvertierungsbaustein wandelt den UDINT-Wert in ein WORD um und meldet die Fertigstellung über seinen `CNF`-Ausgang.
4. Das `CNF`-Ereignis wird an den Ereignisausgang `E1` des Plug `AW_OUT` weitergegeben, und das konvertierte WORD wird an dessen Datenausgang `D1` ausgegeben.

Der gesamte Vorgang ist **ereignisgesteuert** und erfolgt innerhalb eines einzigen Ausführungsschrittes.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Unidirektionaler Datenfluss**: Der FB konvertiert ausschließlich von AUDI (UDINT) zu AW (WORD); eine Rückwärtskonvertierung ist nicht vorgesehen.
- **Composite-Architektur**: Die interne Logik ist vollständig durch einen einzelnen Standard-Konvertierungsbaustein realisiert, wodurch der FB leicht nachvollziehbar und wartbar ist.
- **Verwendung von IEC 61131-Bibliotheken**: Der Baustein `F_UDINT_TO_WORD` stammt aus der `iec61131::conversion`-Bibliothek, was eine hohe Portabilität und Standardkonformität gewährleistet.
- **Keine eigene Zustandsmaschine**: Der FB besitzt keinen ECC (Execution Control Chart) und arbeitet rein datenflussorientiert.

Zustandsübersicht

Der Funktionsblock besitzt **keinen Zustandsautomaten**. Die Verarbeitung erfolgt ereignisgesteuert und deterministisch gemäß der internen Verdrahtung. Es gibt keine internen Zustände oder Latenzen.

Anwendungsszenarien

- **Schnittstellenanpassung**: Ein System liefert Daten über einen AUDI-Adapter (UDINT), ein nachfolgender Funktionsblock erwartet jedoch einen AW-Adapter (WORD). Der AUDI_TO_AW wandelt die Daten ohne manuelle Programmierarbeit an.

- **Datenkonsolidierung**: In einer Steuerung müssen mehrere Sensordaten mit unterschiedlichen Adaptertypen zusammengeführt werden. Der Baustein dient als Konvertierungsglied.
- **Legacy-Integration**: Ältere Komponenten, die auf UDINT basieren, sollen in eine moderne WORD-basierte Architektur eingebunden werden.

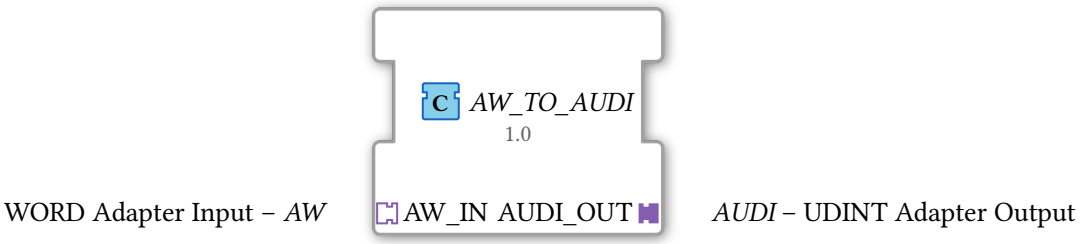
Vergleich mit ähnlichen Bausteinen

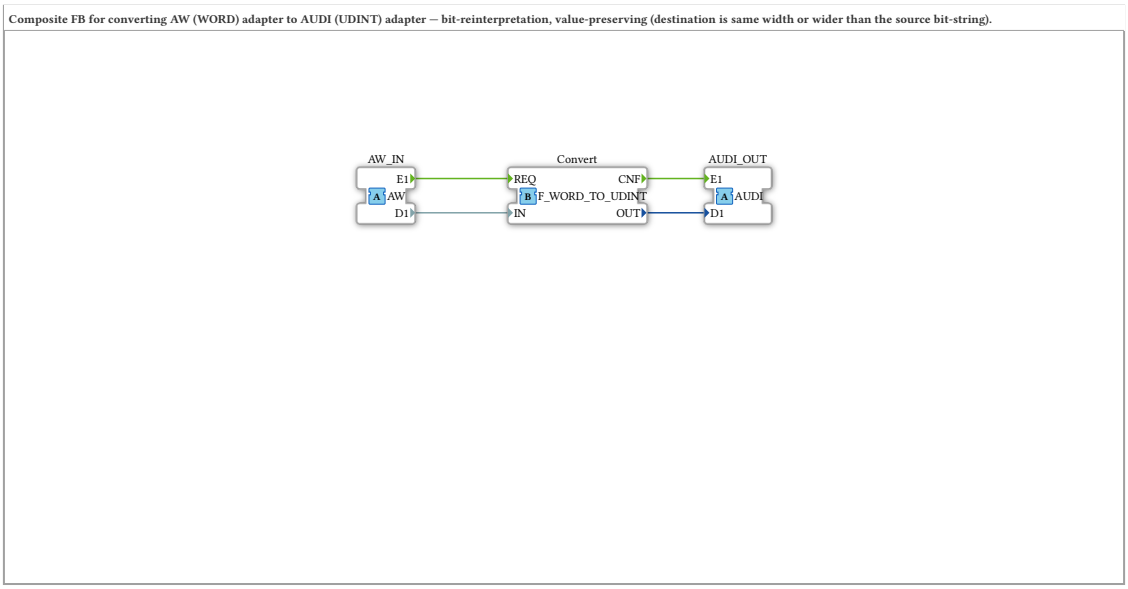
- **F_WORD_TO_UDINT** (analog, aber in umgekehrter Richtung): Konvertiert WORD nach UDINT und erwartet einen AW-Eingang sowie einen AUDI-Ausgang.
- **Allgemeine Konvertierungsbausteine** wie `F_UDINT_TO_DINT` oder `F_UDINT_TO_REAL` bieten ähnliche Funktionalität, arbeiten jedoch mit anderen Datentypen und ohne Adapter-Anbindung.
- **Eigene Adapter-Konverter** können durch Kombination mehrerer Konvertierungsbausteine und Adapter realisiert werden, sind jedoch weniger gekapselt als dieser Composite-FB.

Fazit

Der **AUDI_TO_AW** ist ein spezialisierter, aber essenzieller Baustein für die Adapter-basierte Kommunikation in IEC 61499-Systemen. Durch die Verwendung eines standardisierten Konvertierungsbausteins und die Kapselung in einem Composite-FB bietet er eine saubere, wiederverwendbare und testbare Lösung für die Umsetzung von UDINT- auf WORD-Daten innerhalb eines unidirektionalen Adapterpfades.

AW_TO_AUDI





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `AW_TO_AUDI` ist ein Composite-Baustein, der einen über einen WORD-Adapter (`AW`) eingehenden Datenwert in einen UDINT-Wert konvertiert und über einen UDINT-Adapter (`AUDI`) ausgibt. Die Konvertierung erfolgt mittels des intern genutzten Bausteins `F_WORD_TO_UDINT` aus der IEC-61131-Bibliothek. Der FB kapselt die Adapter-Schnittstellen und ermöglicht eine saubere Trennung zwischen verschiedenen Datenformaten in einem System.

Schnittstellenstruktur

Ereignis-Eingänge

- **AW_IN.E1** (über Socket `AW_IN`): Auslöser für die Konvertierung. Ein eingehendes Ereignis startet die Umwandlung des aktuellen WORD-Werts.
- **Eigene Ereignis-Eingänge**: Keine – der FB empfängt Ereignisse ausschließlich über den Adapter `AW_IN`.

Ereignis-Ausgänge

- **AUDI_OUT.E1** (über Plug `AUDI_OUT`): Quittierung nach erfolgter Konvertierung. Das ausgehende Ereignis signalisiert, dass ein neuer UDINT-Wert am Datenausgang anliegt.
- **Eigene Ereignis-Ausgänge**: Keine – der FB gibt Ereignisse ausschließlich über den Adapter `AUDI_OUT` weiter.

Daten-Eingänge

- **AW_IN.D1** (über Socket `AW_IN`): Der zu konvertierende WORD-Wert (16-Bit).
- **Eigene Daten-Eingänge**: Keine.

Daten-Ausgänge

- **AUDI_OUT.D1** (über Plug `AUDI_OUT`): Der konvertierte UDINT-Wert (32-Bit, vorzeichenlos).
- **Eigene Daten-Ausgänge**: Keine.

Adapter

- **Socket `AW_IN`**: Erwartet eine Instanz des Adapter-Typs `adapter::types::unidirectional::AW`. Dieser Adapter stellt die ereignis- und datenseitige Schnittstelle für den WORD-Eingang bereit.
- **Plug `AUDI_OUT`**: Stellt eine Instanz des Adapter-Typs `adapter::types::unidirectional::AUDI` bereit. Dieser Adapter gibt die ereignis- und datenseitige Schnittstelle für den UDINT-Ausgang nach außen.

Funktionsweise

1. Der FB wartet auf ein Ereignis an **AW_IN.E1**.
2. Mit dem Ereignis wird der aktuelle Wert von **AW_IN.D1** (WORD) an den eingebetteten Funktionsblock `F_WORD_TO_UDINT` übergeben.
3. Der `F_WORD_TO_UDINT` führt die Konvertierung durch und legt das Ergebnis (UDINT) an seinem Ausgang `OUT` an.
4. Sobald die Konvertierung abgeschlossen ist, wird ein Ereignis über `Convert.CNF` an **AUDI_OUT.E1** gesendet und der konvertierte Wert über **AUDI_OUT.D1** ausgegeben.

Datenfluss und Ereignissteuerung sind synchron: Jedes eingehende Ereignis erzeugt genau ein ausgehendes Ereignis mit dem aktualisierten Datenwert.

Technische Besonderheiten

- **Bit-Reinterpretation, werterhaltend** (Zielbreite ist gleich groß oder größer als der Quell-Bit-String).
- **Composite-FB**: Der Baustein ist ausschließlich aus einem internen Netzwerk zusammengesetzt und besitzt keine eigene Zustandslogik.
- **Adapter-Kapselung**: Die gesamte Kommunikation erfolgt über die standardisierten unidirektionalen Adapter `AW` und `AUDI`. Dadurch kann der FB in beliebige Systeme integriert werden, die diese Adapter-Schnittstellen unterstützen.
- **Wiederverwendung**: Die intern verwendete Konvertierungsfunktion `F_WORD_TO_UDINT` ist Teil der IEC-61131-Bibliothek und gewährleistet eine standardkonforme Typumwandlung.
- **Keine Nebenwirkungen**: Der FB führt keinerlei Zustandsänderungen oder Seiteneffekte außerhalb der eigentlichen Konvertierung durch.

Zustandsübersicht

Der FB `AW_TO_AUDI` besitzt keinen eigenen Zustandsautomaten. Die Ablaufsteuerung wird durch den internen Baustein `F_WORD_TO_UDINT` realisiert, der ein einfaches REQ/CNF-Verhalten aufweist:

- **Warten** (vor REQ): Kein Ereignis an `AW_IN.E1` ⟶ keine Aktion.
- **Verarbeitung** (nach REQ, vor CNF): Die Konvertierung läuft; der Ausgang wird nicht aktualisiert.
- **Fertig** (nach CNF): Neuer Wert liegt an `AUDI_OUT.D1` an und ein Ereignis an `AUDI_OUT.E1` wird gesendet.

Der Gesamt-FB spiegelt dieses Verhalten nach außen wider.

Anwendungsszenarien

- **Integration von Subsystemen mit unterschiedlichen Datentypen**: Ein Sensor liefert seine Werte als WORD (z.B. 16-Bit-Zähler) über einen AW-Adapter, während ein Steuerungsteil UDINT-Werte (32-Bit) über einen AUDI-Adapter erwartet.
- **Schnittstellenanpassung in modularen Automatisierungssystemen**: Nutzung in einer Komponente, die Adapter-basiert kommuniziert, aber die eigentliche Datenkonvertierung von WORD nach UDINT benötigt.
- **Bildung von Konvertierungskaskaden**: Zusammen mit anderen Bausteinen, die z.B. DINT- oder REAL-Adapter verwenden, können beliebige Datenformat-Anpassungen aufgebaut werden.

Vergleich mit ähnlichen Bausteinen

Baustein	Eingangsadapter	Ausgangsadapter	Konvertierung	Bemerkung
`AW_TO_AUDI`	`AW` (WORD)	`AUDI` (UDINT)	WORD ⟶ UDINT	Composite, kapselt `F_WORD_TO_UDINT`
Direkte Nutzung von `F_WORD_TO_UDINT`	Keine Adapter	Kein Adapter	WORD ⟶ UDINT	Standard-Funktionsbaustein ohne Adapter-Schnittstellen
`AW_TO_ADINT` (hypothetisch)	`AW` (WORD)	`ADINT` (DINT)	WORD ⟶ DINT	analoge Konvertierung mit Vorzeichen

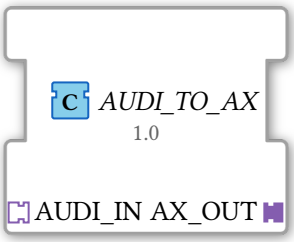
Der `AW_TO_AUDI` unterscheidet sich von der direkten Nutzung der Konvertierungsfunktion durch die Integration von Adaptern, was ihn für adapterbasierte Systeme sofort einsetzbar macht.

Fazit

Der Funktionsblock `AW_TO_AUDI` bietet eine einfache und saubere Möglichkeit, zwischen zwei unidirektionalen Adaptern der Typen WORD und UDINT zu konvertieren. Durch die Kapselung der IEC-Konvertierungsfunktion und die klare Schnittstellendefinition erhöht er die Wiederverwendbarkeit und erleichtert die Integration in bestehende Adapter-Architekturen. Er ist ideal für Anwendungen, in denen unterschiedliche Datenbreiten oder Formate über standardisierte Adapter ausgetauscht werden müssen.

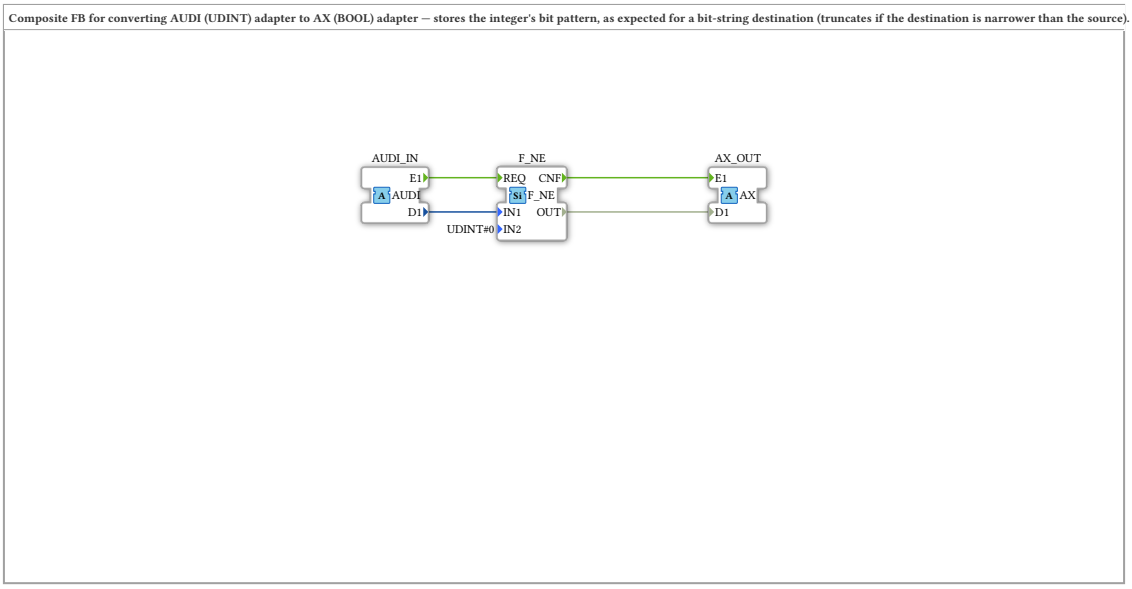
AUDI_AX

AUDI_TO_AX



UDINT Adapter Input – *AUDI*

AX – BOOL Adapter Output



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock `AUDI_TO_AX` ist ein Composite-FB, der einen **AUDI**-Adapter (UDINT-Daten) in einen **AX**-Adapter (BOOL-Daten) umwandelt. Er dient als unidirektionaler Konverter für Anwendungen, bei denen ein ganzzahliger Wert (z.B. aus einem Zähler) auf ein binäres Signal abgebildet werden soll – konkret wird geprüft, ob der Eingangswert ungleich Null ist. Der FB ist als reine Netzwerkverbindung aus IEC-61131-konformen Bausteinen realisiert.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine eigenen Ereignis-Eingänge. Das Ereignis zum Starten der Verarbeitung wird über den Socket-Adapter `AUDI_IN` bereitgestellt (siehe Abschnitt Adapter).

Ereignis-Ausgänge

Der FB besitzt keine eigenen Ereignis-Ausgänge. Das Ergebnisereignis wird über den Plug-Adapter `AX_OUT` nach außen gegeben (siehe Abschnitt Adapter).

Daten-Eingänge

Der FB besitzt keine eigenen Daten-Eingänge. Der zu verarbeitende UDINT-Wert wird über den Socket-Adapter `AUDI_IN` bereitgestellt (siehe Abschnitt Adapter).

Daten-Ausgänge

Der FB besitzt keine eigenen Daten-Ausgänge. Das Ergebnis (BOOL) wird über den Plug-Adapter `AX_OUT` nach außen gegeben (siehe Abschnitt Adapter).

Adapter

Name	Typ	Typ des Adapters	Richtung	Kommentar
`AUDI_IN`	`adapter::types::unidirectional::AUDI`	Socket	Eingang	Stellt den UDINT-Wert (D1) und das zugehörige Ereignis (E1) bereit
`AX_OUT`	`adapter::types::unidirectional::AX`	Plug	Ausgang	Liefert den BOOL-Wert (D1) und das Bestätigungsereignis (E1)

Funktionsweise

Der interne Ablauf ist in drei Schritte gegliedert:

1. **Ereignisempfang:** Das Ereignis `E1` des Socket-Adapters `AUDI_IN` triggert die Verarbeitung.
2. **Vergleich:** Der eingegangene UDINT-Wert (`AUDI_IN.D1`) wird im Baustein `F_NE` (IEC 61131-3: *ungleich*) mit dem konstanten Wert `UDINT#0` verglichen.
3. **Ergebnisausgabe:** Das Ergebnis des Vergleichs (`F_NE.OUT`) – ein BOOL – wird auf den Ausgangsdaten-Port `AX_OUT.D1` gelegt. Gleichzeitig wird das Ereignis `F_NE.CNF` an den Ereigniseingang `AX_OUT.E1` weitergeleitet, sodass der Ausgangsadapter das Ergebnis signalisiert.

Zusammenhang:

- Ist der Eingangswert ungleich 0 ⟶ Ausgabe `TRUE`
- Ist der Eingangswert gleich 0 ⟶ Ausgabe `FALSE`

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Verwendete Bibliothek:** Der Kern besteht aus dem IEC-61131-Baustein `F_NE` (Ungleich-Vergleich). Dieser wird aus der Bibliothek `iec61131::comparison::F_NE` importiert.
- **Kein eigener ECC:** Als Composite-FB besitzt `AUDI_TO_AX` keinen eigenen Ablaufzustand – die gesamte Logik ist rein datenflussgesteuert.
- **Unidirektionale Adapter:** Sowohl der Eingangs- als auch der Ausgangsadapter sind als *unidirectional* ausgelegt, d.h. sie unterstützen nur einen festgelegten Daten- und Ereignisfluss von der Quelle zum Senke.
- **Parametrierung:** Der Vergleichswert ist fest auf `UDINT#0` gesetzt und nicht nach außen parametrierbar.

Zustandsübersicht

Der FB besitzt keinen eigenen Zustandsautomaten. Das Verhalten wird vollständig durch die eingehenden Ereignisse bestimmt: Jedes Ereignis am Socket `AUDI_IN.E1` führt zu einem Durchlauf der internen Logik und einer Ausgabe am Plug `AX_OUT`.

Anwendungsszenarien

- **Binäre Schwellwertausgabe:** Ein Zählerwert (z.B. Anzahl erfasster Teile) soll als Schaltsignal (`TRUE` wenn vorhanden, `FALSE` wenn Null) an eine nachgeschaltete Steuerung weitergegeben werden.
- **Adapter-Konvertierung in heterogenen Systemen:** Wenn ein Sensor oder Subsystem einen UDINT-Wert über einen AUDI-Adapter liefert, das Zielsystem aber einen BOOL-Wert über einen AX-Adapter erwartet.
- **Vereinfachung von Netzwerken:** Der FB kapselt die Logik „Ungleich Null“ und reduziert dadurch die Komplexität in übergeordneten Funktionsplänen.

Vergleich mit ähnlichen Bausteinen

- **`AUDI_TO_BOOL` (hypothetisch):** Würde dieselbe Funktionalität bieten, aber ohne Adapter-Konzept – eventuell mit direkten Daten-Eingängen/-Ausgängen.
- **Direkter Vergleich `F_NE`:** Einzelner Baustein, der den Vergleich durchführt, jedoch ohne Adapter-Anpassung. Der Vorteil von `AUDI_TO_AX` liegt in der nahtlosen Integration in eine Adapter-basierte Architektur, wie sie in der 4diac-IDE üblich ist. Er vermeidet zusätzliche Konvertierungsschritte zwischen unterschiedlichen Schnittstellentypen.

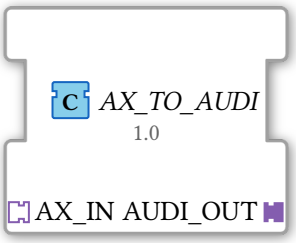
Fazit

Der Funktionsblock `AUDI_TO_AX` ist ein spezialisierter, aber sehr nützlicher Konverter für die unidirektionale Übertragung eines UDINT-Wertes auf ein BOOL-Signal. Durch die klare Kapselung der Vergleichslogik und die Verwendung von Standard-Adaptern vereinfacht er die Signalkette in IEC-61499-basierten Automatisierungssystemen. Er ist robust, einfach zu verstehen und benötigt keine aufwändige Konfiguration – ideal für häufige Aufgaben wie die binäre Auswertung von Zählerständen.

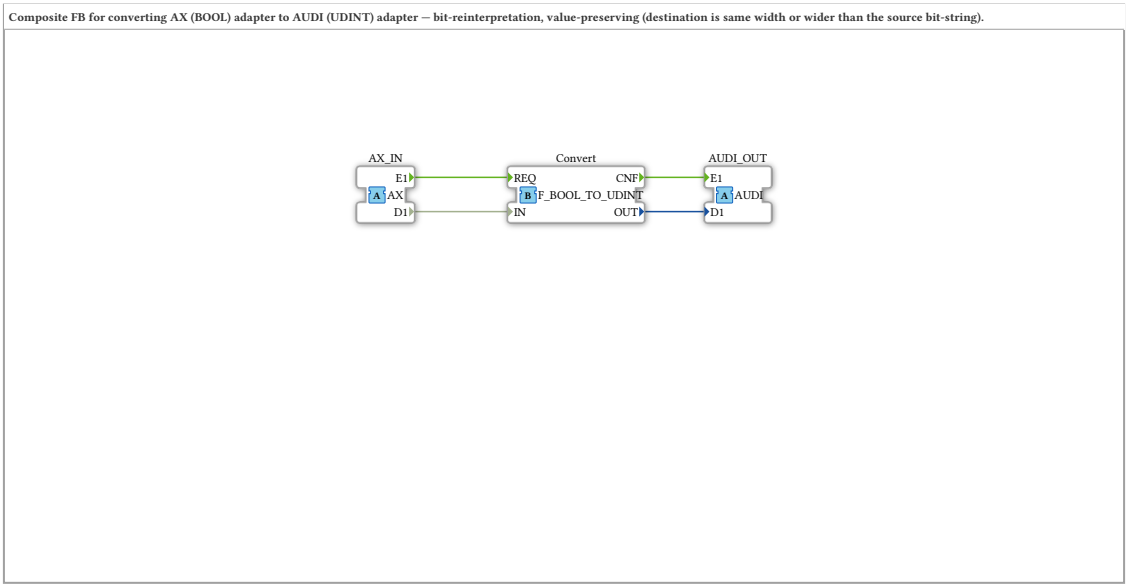
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AX_TO_AUDI



BOOL Adapter Input – *AX* — *AUDI* – UDINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AX_TO_AUDI** ist ein zusammengesetzter Baustein (Composite FB), der einen Booleschen Wert (BOOL), der über einen AX-Adapter bereitgestellt wird, in einen vorzeichenlosen Doppelwort-Wert (UDINT) umwandelt und über einen AUDI-Adapter ausgibt. Er wird eingesetzt, um Signale zwischen Adapter-Schnittstellen zu konvertieren, wobei die eigentliche Umwandlung durch einen internen IEC-61131-Konvertierungsbaustein erfolgt.

Schnittstellenstruktur

Ereignis-Eingänge

- **AX_IN.E1** (über Socket-Adapter AX_IN): Ereignis, das die Konvertierung anstößt.

Ereignis-Ausgänge

- **AUDI_OUT.E1** (über Plug-Adapter AUDI_OUT): Bestätigungsereignis nach erfolgreicher Konvertierung.

Daten-Eingänge

- **AX_IN.D1** (über Socket-Adapter AX_IN, Datentyp **BOOL**): Der umzuwandelnde Boolesche Wert.

Daten-Ausgänge

- **AUDI_OUT.D1** (über Plug-Adapter AUDI_OUT, Datentyp **UDINT**): Der konvertierte Ganzzahlwert (0 oder 1).

Adapter

- **AX_IN** – Socket-Adapter vom Typ `adapter::types::unidirectional::AX` Nimmt einen Booleschen Wert (und ein zugehöriges Ereignis) entgegen.

- **AUDI_OUT** – Plug-Adapter vom Typ `adapter::types::unidirectional::AUDI` Stellt den konvertierten UDINT-Wert (und ein quittierendes Ereignis) bereit.

Funktionsweise

Der Baustein arbeitet ereignisgesteuert:

1. Ein Ereignis am Ereignis-Eingang **AX_IN.E1** triggert den internen Konvertierungsbaustein `F_BOOL_TO_UDINT` über dessen Ereigniseingang **REQ**.
2. Der Datenwert **AX_IN.D1** (BOOL) wird an den Dateneingang **IN** des Konvertierungsbausteins weitergeleitet.
3. Der Konvertierungsbaustein wandelt den Booleschen Wert in einen UDINT um (FALSE → 0, TRUE → 1) und gibt das Ergebnis an seinem Ausgang **OUT** aus.
4. Das Ergebnis wird an den Datenausgang **AUDI_OUT.D1** übergeben.
5. Nach Abschluss der Umwandlung wird das quittierende Ereignis des Konvertierungsbausteins (**CNF**) an den Ereignis-Ausgang **AUDI_OUT.E1** weitergegeben.

Technische Besonderheiten

- **Bit-Reinterpretation, werterhaltend** (Zielbreite ist gleich groß oder größer als der Quell-Bit-String).
- **Zusammengesetzter Baustein (Composite FB)** nach IEC 61499-2.
- Verwendet den internen FB `iec61131::conversion::F_BOOL_TO_UDINT` für die eigentliche Konvertierung.
- Die Adapter-Schnittstellen (`AX` und `AUDI`) ermöglichen eine modulare und standardisierte Anbindung an andere Bausteine.
- Der Baustein ist quelloffen unter der Eclipse Public License 2.0 lizenziert.

Zustandsübersicht

Der Baustein besitzt keine eigenen Zustandsautomaten. Das Verhalten wird vollständig durch die sequenzielle Abarbeitung des internen Konvertierungsbausteins bestimmt:

- **Idle**: Warten auf ein Ereignis an **AX_IN.E1**.
- **Konvertierung aktiv**: Nach Empfang des Ereignisses wird die Umwandlung durchgeführt; während dieser Zeit werden keine weiteren Ereignisse verarbeitet (Single-Execution).
- **Abschluss**: Das Ausgangsereignis wird gesendet, danach kehrt der Baustein in den Idle-Zustand zurück.

Anwendungsszenarien

- Umwandlung eines digitalen Sensorsignals (z. B. Endschalter) in einen numerischen Wert, der in einer SPS-Steuerung weiterverarbeitet werden soll.
- Anpassung zwischen unterschiedlichen Adapter-Schnittstellen in IEC-61499-basierten Automatisierungssystemen.
- Einsatz in Systemen, in denen nur Boolesche Werte über eine AX-Schnittstelle verfügbar sind, aber eine UDINT-Schnittstelle (z. B. für Diagnose- oder Zählzwecke) benötigt wird.

Vergleich mit ähnlichen Bausteinen

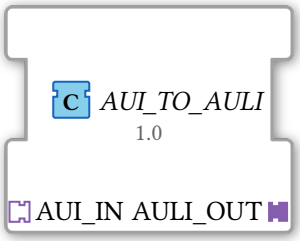
- **F_BOOL_TO_UDINT** (direkt): Führt die gleiche Konvertierung durch, erfordert aber separate Ereignis- und Datenverbindungen. **AX_TO_AUDI** kapselt diese Logik in einem Adapter-basierten Interface.
- **AX_TO_*-Bausteine**: Analog zu diesem Baustein, jedoch mit anderen Ziel-Datentypen (z. B. INT, DINT). Diesem Baustein liegt die UDINT-Variante zugrunde.
- **Adapter-Koppler**: Reine Durchleit-Bausteine ohne Konvertierung – im Gegensatz zu **AX_TO_AUDI**, das eine aktive Typumwandlung vornimmt.

Fazit

Der Funktionsblock **AX_TO_AUDI** stellt eine einfache, aber nützliche Brücke zwischen booleschen und ganzzahligen Adapter-Schnittstellen dar. Durch die Kapselung der Konvertierung in einem Composite-Baustein wird der Entwurf übersichtlicher und die Wiederverwendbarkeit erhöht. Er eignet sich besonders für Standard-Umwandlungsaufgaben in IEC-61499-Projekten, bei denen modulare Adapter-Schnittstellen verwendet werden.

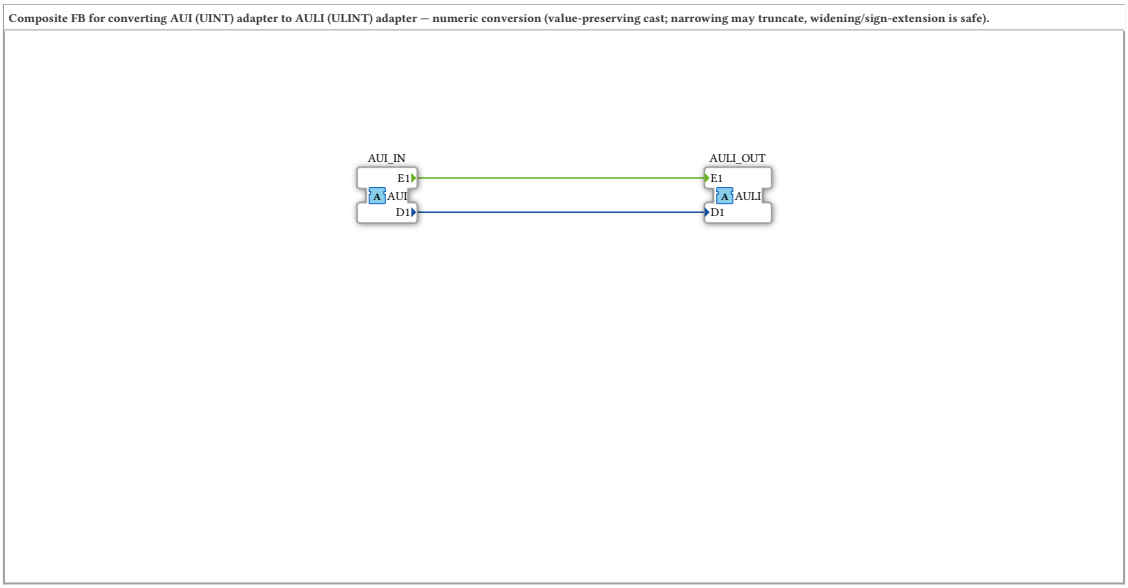
AUI_AULI

AUI_TO_AULI



UINT Adapter Input – *AUI*

AULI – ULINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AUI_TO_AULI` ist ein Composite-Baustein zur Umwandlung eines unidirektionalen AUI-Adapters (Datentyp `UINT`) in einen unidirektionalen AULI-Adapters (Datentyp `ULINT`). Er dient als einfacher Typkonverter zwischen den beiden Adapter-Schnittstellen, wobei sowohl das Ereignissignal als auch der Datenwert transparent weitergereicht werden. Der Baustein ermöglicht eine nahtlose Integration von Komponenten, die unterschiedliche Adressierungsbreiten verwenden.

Schnittstellenstruktur

Der Baustein besitzt keine eigenständigen Ereignis- oder Daten-Ein-/Ausgänge, sondern nutzt ausschließlich Adapter-Schnittstellen. Die nachfolgenden Abschnitte beschreiben die Signale, die über die verwendeten Adapter nach außen geführt werden.

Ereignis-Eingänge

Signal	Beschreibung
(E1 über AUI_IN)	Ereigniseingang des angeschlossenen AUI-Adapters (Socket). Ein eingehendes Ereignis wird direkt an den AULI-Adapter weitergeleitet.

Ereignis-Ausgänge

Signal	Beschreibung
(E1 über AULI_OUT)	Ereignisausgang des angeschlossenen AULI-Adapters (Plug). Das vom AUI-Adapter empfangene Ereignis wird hier ausgegeben.

Daten-Eingänge

Signal	Datentyp	Beschreibung
(D1 über AUI_IN)	`UINT` (16 Bit)	Dateneingang des AUI-Adapters. Der eingehende Wert wird als 16-Bit-Ganzzahl interpretiert.

Daten-Ausgänge

Signal	Datentyp	Beschreibung
(D1 über AULI_OUT)	`ULINT` (64 Bit)	Datenausgang des AULI-Adapters. Der vom AUI-Adapter übernommene Wert wird als 64-Bit-Ganzzahl ohne Vorzeichen ausgegeben.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AUI`	`AUI_IN`	Socket (Eingang)	Adapter für den Empfang von Ereignissen und Daten im Format `UINT`.
`adapter::types::unidirectional::AULI`	`AULI_OUT`	Plug (Ausgang)	Adapter für das Senden von Ereignissen und Daten im Format `ULINT`.

Funktionsweise

Der Baustein leitet eingehende Ereignisse und Daten vom Eingangsadapter `AUI_IN` unverändert zum Ausgangsadapter `AULI_OUT` weiter. Dabei wird der Datentyp `UINT` implizit nach `ULINT` konvertiert (Wert bleibt erhalten, lediglich die Bitbreite wird von 16 auf 64 Bit erweitert). Die Verbindung erfolgt im FBNetzwerk des Composite-Bausteins durch direkte Verdrahtung der jeweiligen Ereignis- und Datenkanäle (`E1` und `D1`). Es findet keine zusätzliche Logik oder Zustandsspeicherung statt.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Implizite Typkonvertierung:** Die Umwandlung von `UINT` nach `ULINT` erfolgt automatisch durch die 4diac-IDE und erfordert keinen expliziten Konvertierungsbaustein. Der Wertebereich von 0 bis 65535 (UINT) wird auf den entsprechenden ULINT-Bereich abgebildet.
- **Durchreicheverhalten:** Ereignisse und Daten werden ohne Verzögerung oder Pufferung übertragen. Das Ausgangsereignis tritt zeitgleich mit dem Eingangsereignis auf.
- **Unidirektionale Adapter:** Der Baustein unterstützt nur eine Datenflussrichtung (vom AUI-Eingang zum AULI-Ausgang). Eine Rückwärtskommunikation ist nicht vorgesehen.

Zustandsübersicht

Der Baustein besitzt keine eigenen Zustände, da er ausschließlich aus direkten Verbindungen besteht. Das Verhalten ist rein kombinatorisch: Ein Ereignis am Eingang erzeugt sofort ein Ereignis am Ausgang, und der Datenwert wird kontinuierlich übertragen.

Anwendungsszenarien

- **Systeme mit gemischter Adressierungsbreite:** Wenn ein Sensor oder eine Komponente Daten im Format `UINT` (z. B. 16-Bit-Zählerwert) liefert, aber eine nachgelagerte Steuerung oder ein Bus-Interface nur `ULINT` (64-Bit) akzeptiert.
- **Adapter-Kopplung:** Einsatz als Zwischenstück zwischen zwei unterschiedlichen Adaptertypen in einer modularen Steuerungsarchitektur, die auf dem 61499-Standard basiert.
- **Prototypenentwicklung:** Schnelle Integration von Bibliotheksbausteinen, die unterschiedliche Adapterschnittstellen verwenden, ohne manuelle Konvertierung.

Vergleich mit ähnlichen Bausteinen

- **INT_TO_LINT / UINT_TO_ULINT (direkte Datenkonverter):** Diese Bausteine arbeiten auf reinen Datenebenen ohne Ereignisweiterleitung und erfordern separate Ereignisverarbeitung. `AUI_TO_AULI` kombiniert Ereignis- und Datenkonvertierung in einem Adapter-Interface.
- **Adapter-Adapter-Connector:** Andere Composite-Bausteine, die zwei gleichartige Adapter verbinden (z. B. AUI_TO_AUI) – sie unterscheiden sich lediglich durch die fehlende Typumwandlung.
- **Explizite Konvertierungsbausteine mit mehreren Kanälen:** Komplexere Bausteine, die mehrere Signale gleichzeitig konvertieren, sind für einfache Punkt-zu-Punkt-Verbindungen wie diese überdimensioniert.

Fazit

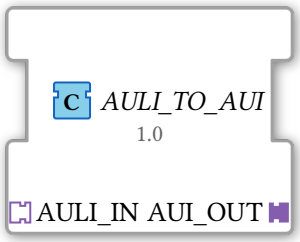
`AUI_TO_AULI` ist ein schlanker, spezialisierter Composite-Baustein zur Umwandlung eines AUI-Adapters in einen AULI-Adapter. Er vereinfacht die Typanpassung zwischen 16-Bit- und 64-Bit-Adressierung in IEC 61499-Systemen, indem er sowohl Ereignis- als auch Datenpfade automatisch konvertiert und weiterleitet. Dank der impliziten Typkonvertierung von `UINT` nach `ULINT` entfällt die manuelle Implementierung von Konvertierungslogik, was den Entwurf modularer und wiederverwendbarer Steuerungsanwendungen erleichtert.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

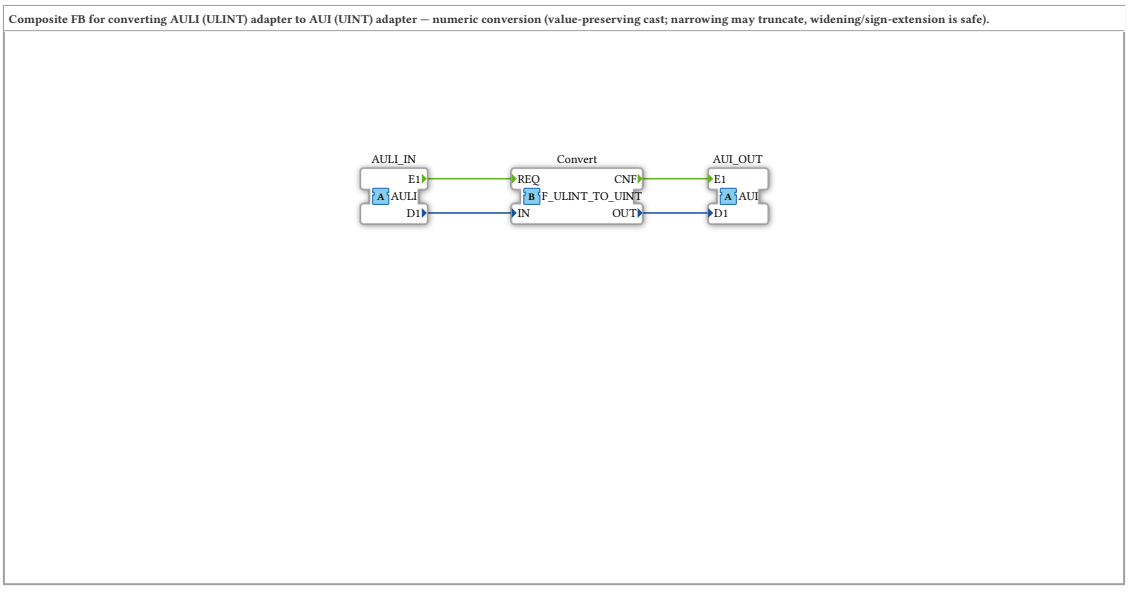
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AULI_TO_AUI

ULINT Adapter Input – *AULI*



AUI – UINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AULI_TO_AUI** ist ein zusammengesetzter Baustein (Composite FB), der einen **AULI**-Adapter (basiert auf dem Datentyp `ULINT`) in einen **AUI**-Adapter (basiert auf dem Datentyp `UINT`) umwandelt. Er wird eingesetzt, wenn eine Schnittstelle mit 64-Bit unsigned Integer (`ULINT`) auf eine 16-Bit

unsigned Integer (UINT) Schnittstelle abgebildet werden muss, z. B. bei der Anbindung unterschiedlicher Sensor- oder Aktorkomponenten in der Automatisierungstechnik.

Schnittstellenstruktur

Ereignis-Eingänge
Über den Adapter-Socket AULI_IN wird ein Ereignis vom Typ E1 bereitgestellt. Dieses wird intern an den Konvertierungsbaustein weitergeleitet.

Ereignis-Ausgänge
Über den Adapter-Plug AUI_OUT wird ein Ereignis vom Typ E1 ausgegeben, sobald die Konvertierung abgeschlossen ist.

Daten-Eingänge
Über den Adapter-Socket AULI_IN wird ein Datenwert vom Typ D1 (entspricht ULINT) entgegengenommen.

Daten-Ausgänge
Über den Adapter-Plug AUI_OUT wird ein Datenwert vom Typ D1 (entspricht UINT) ausgegeben.

Adapter

Name	Richtung	Typ
AULI_IN	Socket	adapter::types::unidirectional::AULI
AUI_OUT	Plug	adapter::types::unidirectional::AUI

Funktionsweise
Der Baustein arbeitet ereignisgesteuert:

1. Ein Ereignis am Eingang AULI_IN.E1 triggert den internen Funktionsblock F_ULINT_TO_UINT über dessen Eingang REQ.
2. Der Datenwert AULI_IN.D1 wird direkt als Eingang IN an den Konvertierungsbaustein übergeben.
3. Der Konvertierungsbaustein wandelt den ULINT-Wert in einen UINT-Wert um.
4. Nach Abschluss der Konvertierung signalisiert der Ausgang CNF des Konvertierungsbausteins ein Ereignis, welches an den Ausgang AUI_OUT.E1 weitergeleitet wird.
5. Gleichzeitig wird der konvertierte Wert über den Ausgang OUT des Konvertierungsbausteins an den Ausgang AUI_OUT.D1 übergeben.

Die Umwandlung erfolgt gemäß IEC 61131-Norm F_ULINT_TO_UINT. Bei Werten größer als 65535 (maximaler UINT-Wert) kann es zu einem Überlauf kommen, der applikationsseitig zu behandeln ist.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Adapterbasierte Kapselung**: Der Baustein erlaubt den nahtlosen Austausch von Adaptern in der 4diac-IDE, ohne die Logik des umgebenden Netzwerks zu ändern.
- **Wiederverwendung der IEC-Norm-Konvertierung**: Die eigentliche Typumwandlung wird durch den standardisierten IEC-Baustein F_ULINT_TO_UINT durchgeführt, was Portabilität und Korrektheit sicherstellt.
- **Keine Zustandsspeicherung**: Der Baustein ist rein kombinatorisch – nach jeder Ausführung liegt das aktuelle Ergebnis am Ausgang an.

Zustandsübersicht
Der Baustein besitzt keinen eigenen endlichen Automaten. Die interne Konvertierung erfolgt ereignisgesteuert und ohne interne Zustände. Die Laufzeit wird allein durch den unterlagerten F_ULINT_TO_UINT-Baustein bestimmt.

Anwendungsszenarien

- **Sensorintegration**: Ein 64-Bit-Sensor (z. B. hochauflösende Wegmessung) wird an eine Steuerung angeschlossen, die nur 16-Bit-Eingänge akzeptiert.
- **Kommunikations-Gateway**: Adapter-Schnittstellen zwischen Bussystemen (z. B. CAN-open mit 64-Bit-Daten auf PROFINET-I-Device mit 16-Bit-Daten).
- **Datenreduktion**: Gezielte Reduzierung der Auflösung für nachfolgende Verarbeitungsschritte (z. B. Anzeige oder einfache Schwellwertlogik).

Vergleich mit ähnlichen Bausteinen

Baustein	Eingangstyp	Ausgangstyp	Besonderheit
`AULI_TO_AUI`	`ULINT`	`UINT`	Adapter-Wrapper, ereignisgesteuert
`F_ULINT_TO_UINT`	`ULINT`	`UINT`	Reine Funktion, kein Adapter
`AUI_TO_AULI`	`UINT`	`ULINT`	Umgekehrte Richtung (nicht vorhanden)

Der vorliegende Baustein bietet im Gegensatz zur nackten IEC-Funktion eine adapterkonforme Ein- und Ausgangsschnittstelle und ermöglicht damit die einfache Einbindung in adapterbasierte Architekturen.

Fazit

Der **AULI_TO_AUI**-Baustein ist eine praktische Lösung, um 64-Bit-ULINT-Daten über einen einheitlichen Adapter-Mechanismus auf 16-Bit-UINT-Daten abzubilden. Er kombiniert die Robustheit der IEC-Konvertierungsfunktion mit der Flexibilität von 4diac-Adaptern und eignet sich besonders für heterogene Automatisierungssysteme, in denen unterschiedliche Datentypen aufeinandertreffen.

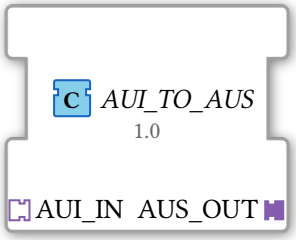
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

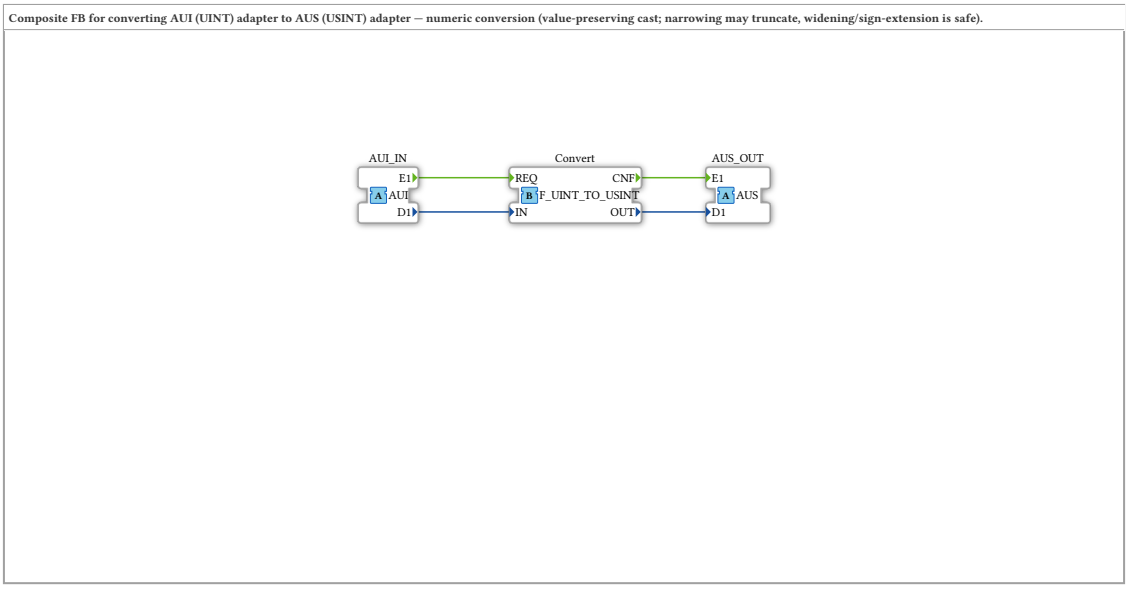
AUI_AUS

AUI_TO_AUS

UINT Adapter Input – *AUI*



AUS – USINT Adapter Output



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock **AUI_TO_AUS** dient als zusammengesetzter Baustein zur Umwandlung eines AUI-Adapters (Datentyp UINT) in einen AUS-Adapter (Datentyp USINT). Er wird typischerweise eingesetzt, um zwischen unterschiedlichen Adaptertypen in der 4diac-IDE zu vermitteln, wenn eine Größenreduktion des Wertebereichs oder eine Typanpassung erforderlich ist. Der Baustein kapselt die Konvertierungslogik und ermöglicht eine nahtlose Integration in bestehende Ablaufumgebungen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine separaten Ereignis-Eingänge. Die Ereignissteuerung erfolgt über die angeschlossenen Adapter.

Ereignis-Ausgänge

Keine separaten Ereignis-Ausgänge. Die Ereignisweitergabe erfolgt über die angeschlossenen Adapter.

Daten-Eingänge

Keine separaten Daten-Eingänge. Die Dateneingabe erfolgt über die angeschlossenen Adapter.

Daten-Ausgänge

Keine separaten Daten-Ausgänge. Die Datenausgabe erfolgt über die angeschlossenen Adapter.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	**AUI_IN**	adapter::types::unidirectional::AUI	UINT Eingangsadapter (Quelle)
Plug	**AUS_OUT**	adapter::types::unidirectional::AUS	USINT Ausgangsadapter (Ziel)

Der Socket `AUI_IN` nimmt den UINT-Wert (16-Bit ohne Vorzeichen) sowie das zugehörige Ereignissignal auf. Der Plug `AUS_OUT` gibt den konvertierten USINT-Wert (8-Bit ohne Vorzeichen) mit dem entsprechenden Ereignis aus.

Funktionsweise

Der FB arbeitet als einfaches Durchschalteelement mit Typkonvertierung:

1. **Ereignisempfang**: Ein eingehendes Ereignis am Socket `AUI_IN.E1` triggert die Verarbeitung.
2. **Datenübernahme**: Der zugehörige Datenwert `AUI_IN.D1` (Typ UINT) wird ausgelesen.
3. **Konvertierung**: Intern wird die Funktion `F_UINT_TO_USINT` aus der IEC-61131-Bibliothek aufgerufen. Diese wandelt den UINT-Wert in einen USINT-Wert um. Bei Überschreitung des USINT-Wertebereichs

(0…255) wird der Wert gemäß der zugrunde liegenden Implementierung beschnitten oder entsprechend behandelt (in der Regel auf 255 gekappt).
4. **Ausgabe**: Der konvertierte Wert wird auf `AUS_OUT.D1` (Typ USINT) bereitgestellt, und das Ausgangsereignis `AUS_OUT.E1` wird gesendet.

Die gesamte Kopplung erfolgt über die Ereignis- und Datenverbindungen im FBNetzwerk des Composite-Bausteins.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Bereichseinschränkung**: Die Konvertierung von UINT (0…65535) auf USINT (0…255) führt zwangsläufig zu einem Informationsverlust, sofern der Eingangswert größer als 255 ist. Anwender müssen sicherstellen, dass der Wertebereich des Eingangs den Erwartungen entspricht.
- **Keine Zustandsspeicherung**: Der FB ist zustandslos – jede Verarbeitung erfolgt unabhängig von vorherigen Umwandlungen.
- **Wiederverwendbarkeit**: Der Baustein ist als gekapselte Komponente ausgelegt und kann in verschiedenen Projekten eingesetzt werden, ohne dass die Konvertierungslogik erneut implementiert werden muss.
- **Kompatibilität**: Vorausgesetzt wird die Verfügbarkeit der Bibliothek `iec61131::conversion::F_UINT_TO_USINT`.

Zustandsübersicht

Der FB verfügt über keine explizite Zustandsmaschine. Der interne Ablauf kann jedoch als zweischrittig beschrieben werden:

- **Bereit (Idle)**: Der FB wartet auf ein eingehendes Ereignis am Socket.
- **Verarbeitung (Convert)**: Nach Ereigniseingang wird die Konvertierung durchgeführt und das Ausgangsereignis ausgelöst. Danach kehrt der FB sofort in den Bereit-Zustand zurück.

Eine detaillierte Zustandsgrafik ist nicht erforderlich, da der FB keine Verzögerungen oder mehrstufigen Schritte aufweist.

Anwendungsszenarien

- **Sensoranbindung**: Ein Sensor liefert Werte im UINT-Format, die an eine nachfolgende Logik mit USINT-Anforderung übergeben werden müssen.
- **Gerätekopplung**: Zwischen zwei Geräten oder Komponenten, die unterschiedliche Adaptertypen verwenden, wird eine nahtlose Kommunikation hergestellt.
- **Datenreduktion**: In Fällen, in denen die höhere Auflösung eines UINT nicht benötigt wird, kann die Konvertierung zu USINT Speicher- und Bandbreitenvorteile bringen.
- **Systemmigration**: Bei der Umstellung von älteren auf neuere Adapterprotokolle kann dieser FB als temporäre oder dauerhafte Brücke dienen.

Vergleich mit ähnlichen Bausteinen

Es existieren andere Konvertierungsbausteine in der 4diac-Bibliothek, z. B. `F_UINT_TO_USINT`, der direkt einen UINT-Eingang auf einen USINT-Ausgang abbildet. Der hier beschriebene **AUI_TO_AUS** unterscheidet sich dadurch, dass er Adapter (AUI und AUS) anstelle von elementaren Dateneingängen verwendet. Dadurch ist er in der Lage, die gesamte Schnittstelle inklusive Ereignissteuerung zu kapseln, was den Einsatz in komplexen, adapterbasierten Architekturen vereinfacht. Ein reiner Funktionsbaustein wie `F_UINT_TO_USINT` benötigt hingegen separate Ereignis- und Datenverbindungen.

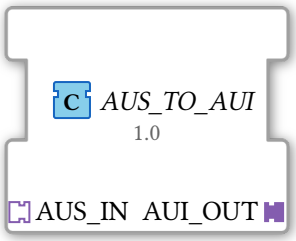
Fazit

Der **AUI_TO_AUS** ist ein kompakter und fokussierter Baustein zur Adapter-Konvertierung von UINT nach USINT. Seine composite-Struktur erleichtert die Integration in vorhandene 4diac-Projekte, reduziert den Verdrahtungsaufwand und erhöht die Übersichtlichkeit. Die einfache Funktionsweise und die klare Schnittstelle machen ihn zu einem nützlichen Werkzeug für alle Anwendungen, bei denen eine Typumwandlung auf Adapterebene erforderlich ist.

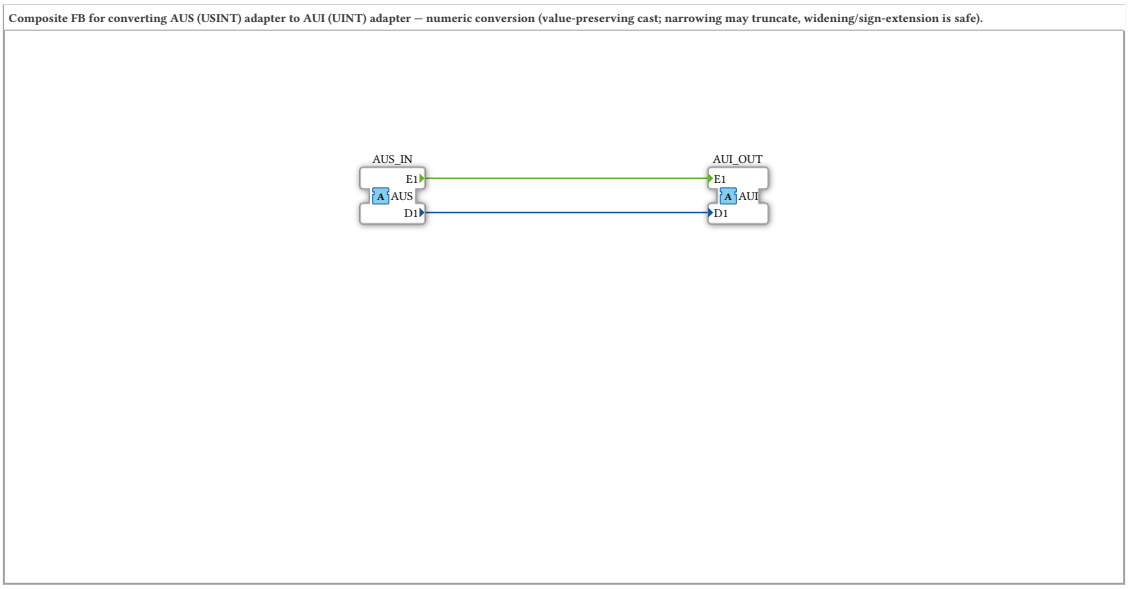
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUS_TO_AUI



USINT Adapter Input – *AUS* *AUI* – UINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUS_TO_AUI** ist ein Composite-Baustein, der eine Schnittstellenumsetzung von einem AUS-Adapter (USINT-Datentyp) zu einem AUI-Adapter (UINT-Datentyp) vornimmt. Er dient als einfache Durchleitung, bei der sowohl das Ereignis als auch der Datenwert direkt vom Eingangs- zum Ausgangsadapter übertragen werden. Die implizite Typkonvertierung von USINT (8-Bit vorzeichenlos) nach UINT (16-Bit vorzeichenlos) erfolgt automatisch durch die Laufzeitumgebung.

Schnittstellenstruktur

Ereignis-Eingänge

Bezeichnung	Datentyp	Beschreibung
E1	Event	Ereignis-Eingang des AUS-Adapters; wird unverändert an den Ausgang weitergeleitet.

Ereignis-Ausgänge

Bezeichnung	Datentyp	Beschreibung
E1	Event	Ereignis-Ausgang des AUI-Adapters; übernimmt das Ereignis vom Eingang.

Daten-Eingänge

Bezeichnung	Datentyp	Beschreibung
D1	USINT	Datenwert (8-Bit) vom AUS-Adapter; wird nach UINT konvertiert und weitergegeben.

Daten-Ausgänge

Bezeichnung	Datentyp	Beschreibung
D1	UINT	Datenwert (16-Bit) des AUI-Adapters; enthält den konvertierten USINT-Wert.

Adapter

Typ	Name	Richtung	Beschreibung
adapter::types::unidirectional::AUS	AUS_IN	Socket (Eingang)	AUS-Adapter mit USINT-Daten und einem Ereignis.
adapter::types::unidirectional::AUI	AUI_OUT	Plug (Ausgang)	AUI-Adapter mit UINT-Daten und einem Ereignis.

Funktionsweise

Der Baustein arbeitet als reine Durchleitung:

- Ein eingehendes Ereignis an `AUS_IN.E1` wird direkt an `AUI_OUT.E1` weitergegeben.
- Der eingehende Datenwert von `AUS_IN.D1` (USINT) wird ohne zusätzliche Logik an `AUI_OUT.D1` weitergeleitet. Die Typkonvertierung von USINT nach UINT wird automatisch von der 4diac-IDE bzw. der Zielplattform ausgeführt.

Es existieren keine internen Funktionsbausteine oder Algorithmen, die das Signal manipulieren.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-Baustein**: Enthält keine eigenen FB-Instanzen, sondern verbindet lediglich die Adapterports direkt miteinander.
- **Implizite Typkonvertierung**: Die Konvertierung von USINT zu UINT wird nicht explizit im FB abgebildet, sondern durch das Laufzeitsystem unterstützt. Dabei wird der 8-Bit-Wert in den niederwertigen Teil des 16-Bit-Worts übernommen.
- **Unidirektionale Adapter**: Sowohl AUS als auch AUI sind als unidirektionale Adapter definiert, d.h. die Datenflüsse sind nur in eine Richtung vorgesehen.

Zustandsübersicht

Der Baustein besitzt keine eigenen Zustände, da er keinerlei interne Logik oder Zustandsautomaten enthält. Er reagiert ausschließlich auf Eingangsereignisse und leitet diese unverzüglich weiter.

Anwendungsszenarien

- **Systemintegration**: Wenn ein Steuerungssystem (z.B. Feldgeräte) Daten im USINT-Format liefert, ein nachgelagertes Modul aber UINT erwartet.
- **Adapterumschaltung**: In einer SPS oder verteilten Steuerungsumgebung können Standardadapter unterschiedlicher Typen über solch einen Konvertierungsbaustein gekoppelt werden.
- **Prototyping**: Schnelle Anpassung von Schnittstellen während der Entwicklungsphase, ohne die zugrundeliegenden Adapter zu ändern.

Vergleich mit ähnlichen Bausteinen

- **INT_TO_UINT**: Konvertiert einen Integer-Datentyp in einen unsigned Integer – ebenfalls als Composite oder Basisbaustein realisierbar.

- **AUS_TO_AUI** ist spezifisch auf die Adaptertypen AUS (USINT) und AUI (UINT) zugeschnitten. Andere Bausteine, wie z.B. `SINT_TO_USINT`, verwenden analoge Durchleitungen für andere Datentypen.
- Im Gegensatz zu Bausteinen mit arithmetischer oder logischer Verarbeitung beschränkt sich dieser Baustein auf die reine Konvertierung und Ereignisweitergabe.

Fazit

Der **AUS_TO_AUI**-Baustein ist ein schlankes Werkzeug zur Anpassung von Adapterschnittstellen zwischen USINT und UINT. Seine einfache Durchleitungsstruktur macht ihn performant und leicht verständlich. Er eignet sich besonders für Integrationsaufgaben, bei denen bestehende Komponenten mit unterschiedlichen Datenbreiten verbunden werden müssen. Aufgrund fehlender interner Logik ist er robust und benötigt keine Parametrierung.

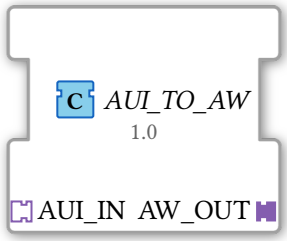
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

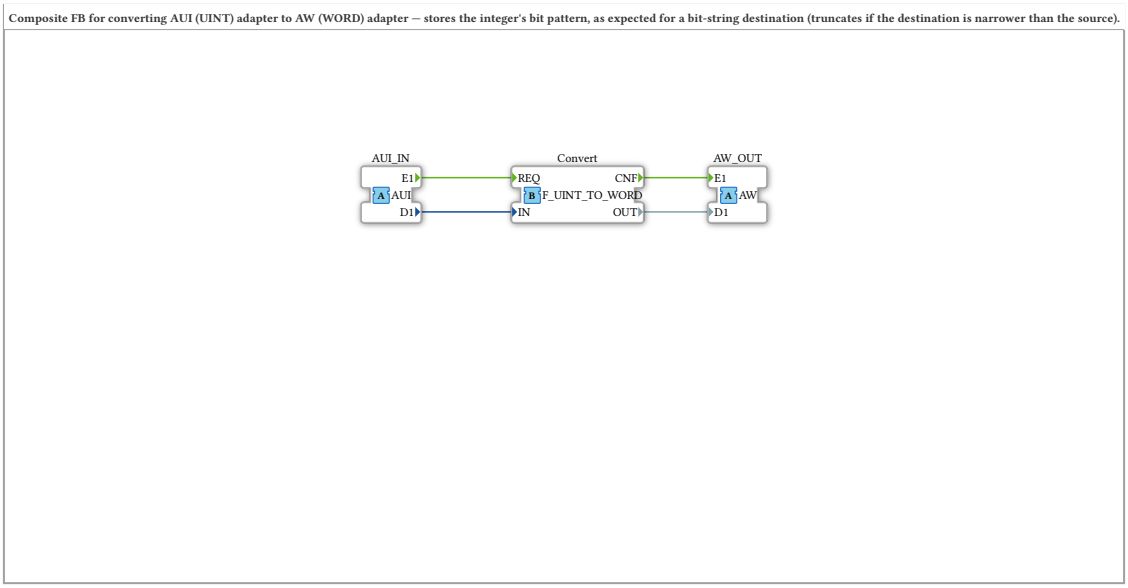
AUI_AW

AUI_TO_AW

UINT Adapter Input – *AUI*



AW – WORD Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `AUI_TO_AW` ist ein zusammengesetzter (Composite) Baustein, der einen AUI-Adapter (Datentyp UINT) in einen AW-Adapter (Datentyp WORD) umwandelt. Er kapselt die Typkonvertierung und stellt sie als einheitliche Adapterschnittstelle zur Verfügung, sodass Systemkomponenten mit unterschiedlichen Datentypen nahtlos miteinander verbunden werden können.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
`AUI_IN.E1`	Ereignis (über Adapter)	Trigger zur Konvertierung des eingehenden UINT-Wertes

Ereignis-Ausgänge

Name	Typ	Kommentar
`AW_OUT.E1`	Ereignis (über Adapter)	Bestätigung, dass der konvertierte WORD-Wert bereitgestellt wurde

Daten-Eingänge

Name	Datentyp	Kommentar
`AUI_IN.D1`	UINT (über Adapter)	Der zu konvertierende Eingangswert

Daten-Ausgänge

Name	Datentyp	Kommentar
`AW_OUT.D1`	WORD (über Adapter)	Der konvertierte Ausgangswert

Adapter

Richtung	Name	Typ	Kommentar
Socket (Eingang)	`AUI_IN`	`adapter::types::unidirectional::AUI`	UINT-Adapter als Quelle
Plug (Ausgang)	`AW_OUT`	`adapter::types::unidirectional::AW`	WORD-Adapter als Senke

Funktionsweise

Der Baustein reagiert auf ein Ereignis am Socket `AUI_IN.E1`. Daraufhin wird der aktuelle Wert von `AUI_IN.D1` (Datentyp UINT) an den internen Konvertierungsbaustein `F_UINT_TO_WORD` weitergeleitet. Dieser wandelt den Wert nach IEC 61131 in den Datentyp WORD um. Sobald die Konvertierung abgeschlossen ist, wird der Ergebniswert an den Plug `AW_OUT.D1` übergeben und ein Bestätigungsereignis auf `AW_OUT.E1` ausgelöst.

Das Zusammenspiel von Ereignis- und Datenfluss sorgt für eine sichere und deterministische Datenweitergabe.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Composite-Aufbau:** Der Baustein kapselt die komplette Umwandlungslogik in einem einzigen Funktionsblock, sodass in der Anwendung keine weiteren Konvertierungsschritte nötig sind.
- **Typkonvertierung:** Die Konvertierung erfolgt mittels des standardisierten IEC-61131-Bausteins `F_UINT_TO_WORD`. UINT und WORD sind bitkompatibel, sodass keine Datenverluste auftreten.
- **Adapter-Schnittstelle:** Durch die Nutzung von Adaptern wird eine lose Kopplung der Komponenten erreicht. Der Baustein kann direkt in Adapter-basierte Netzwerke eingefügt werden.
- **Keine Zustände:** Der Baustein besitzt kein eigenes Zustandsverhalten (ECC). Er arbeitet rein ereignisgesteuert und führt bei jedem Trigger eine sofortige Umwandlung durch.

Zustandsübersicht

Der Baustein enthält keinen eigenen Zustandsautomaten (ECC). Die Ablaufsteuerung erfolgt allein durch die Ereignisverbindungen im Composite-Netzwerk. Nach dem Start ist der Baustein sofort bereit, eingehende Ereignisse zu verarbeiten und die Konvertierung durchzuführen.

Anwendungsszenarien

- **Adapter-Brücke:** Wenn ein Systemteil nur über einen AUI-Adapter (UINT) kommuniziert, ein anderer Teil jedoch einen AW-Adapter (WORD) erwartet, kann `AUI_TO_AW` als Vermittler eingesetzt werden.
- **Daten-Typanpassung:** In Industrie 4.0- oder IIoT-Umgebungen, in denen Geräte unterschiedliche Datentypen verwenden, dient der Baustein zur Harmonisierung der Schnittstellen.
- **Modulare Verkettung:** Der Baustein lässt sich einfach in eine Kette von Adaptern einfügen, um die Kompatibilität zwischen Komponenten herzustellen.

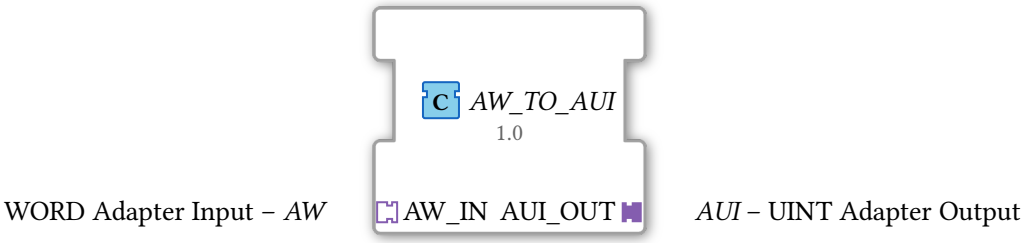
Vergleich mit ähnlichen Bausteinen

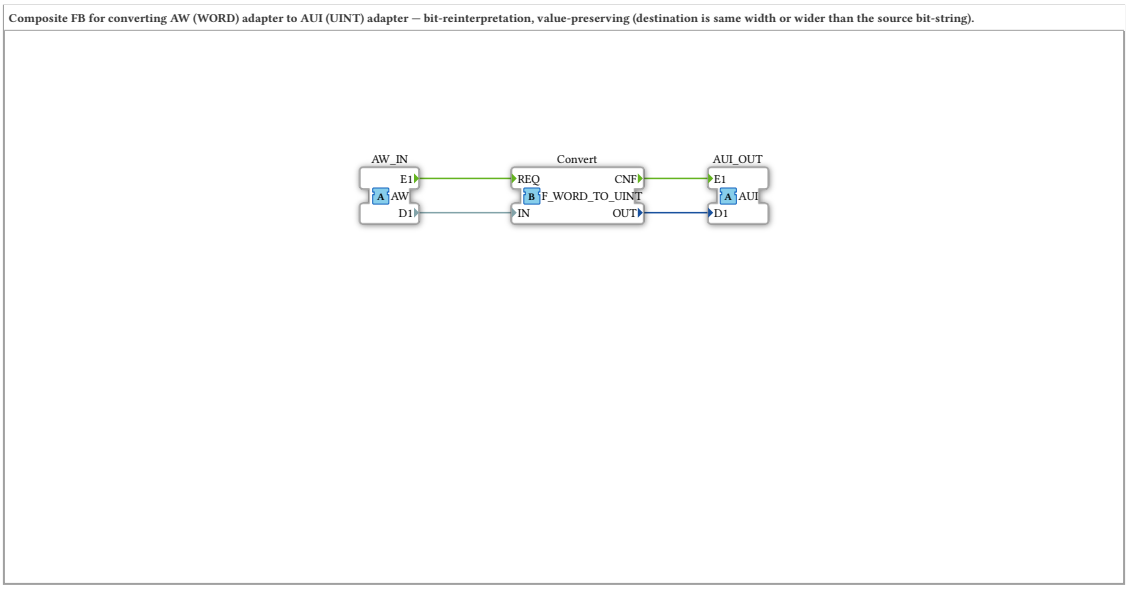
- **Direkte Nutzung von `F_UINT_TO_WORD`:** Statt des Composite-Bausteins könnte man direkt `F_UINT_TO_WORD` verwenden und die Adapter separat miteinander verbinden. Der Composite-Block vereinfacht die Verkabelung und erhöht die Wiederverwendbarkeit.
- **Analoge Konverter (z. B. `AWORD_TO_AUINT`):** Es existieren entsprechende Bausteine für die umgekehrte Richtung. `AUI_TO_AW` fokussiert auf die spezifische Umwandlung von UINT zu WORD und bietet eine klar benannte, zielgerichtete Lösung.

Fazit

Der Funktionsblock `AUI_TO_AW` stellt eine kompakte und robuste Lösung zur Umwandlung eines UINT-Adapter-Signals in ein WORD-Adapter-Signal dar. Durch die Kapselung der Standardkonvertierung und die Adapter-Schnittstelle wird die Integration in bestehende Systeme erleichtert und die Lesbarkeit des Netzwerks verbessert. Der Baustein ist ideal geeignet, um Datentypunterschiede zwischen Komponenten auszugleichen, ohne die Übersichtlichkeit der Anwendung zu beeinträchtigen.

AW_TO_AUI





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AW_TO_AUI** ist ein zusammengesetzter Baustein (Composite FB), der einen **AW (WORD)**-Adapter in einen **AUI (UINT)**-Adapter umwandelt. Er ermöglicht die nahtlose Anbindung von

Komponenten, die unterschiedliche Datentypen über unidirektionale Adapter nutzen. Intern verwendet er den Standard-Konvertierungsbaustein `F_WORD_TO_UINT` zur Typumwandlung.

Schnittstellenstruktur

Ereignis-Eingänge
Der FB besitzt keine eigenständigen Ereignis-Eingänge. Das Ereignis `E1` wird über den **Socket (Eingangsadapter) `AW_IN`** bereitgestellt und löst die Datenkonvertierung aus.

Ereignis-Ausgänge
Der FB besitzt keine eigenständigen Ereignis-Ausgänge. Das Ereignis `E1` wird über den **Plug (Ausgangsadapter) `AUI_OUT`** nach erfolgreicher Konvertierung ausgegeben.

Daten-Eingänge
Der FB besitzt keine eigenständigen Daten-Eingänge. Das Datum `D1` vom Typ `WORD` wird über den **Socket `AW_IN`** bereitgestellt.

Daten-Ausgänge
Der FB besitzt keine eigenständigen Daten-Ausgänge. Das Datum `D1` vom Typ `UINT` wird über den **Plug `AUI_OUT`** ausgegeben.

Adapter

Name	Richtung	Typ	Beschreibung
`AW_IN`	Eingang	`adapter::types::unidirectional::AW`	WORD-Adapter-Eingang (Socket)
`AUI_OUT`	Ausgang	`adapter::types::unidirectional::AUI`	UINT-Adapter-Ausgang (Plug)

Funktionsweise
Der Funktionsblock fungiert als reine Konvertierungsschicht zwischen zwei unterschiedlichen Adaptertypen. Intern ist ein einziger Funktionsbaustein `F_WORD_TO_UINT` eingebettet:

1. **Ereignissteuerung**:
   - Ein eingehendes Ereignis am Socket `AW_IN.E1` triggert den Konvertierungsvorgang (`Convert.REQ`).
   - Nach Abschluss der Konvertierung erzeugt der interne Baustein ein Bestätigungsereignis (`Convert.CNF`), das an den Plug `AUI_OUT.E1` weitergeleitet wird.
2. **Datenfluss**:
   - Der vom Socket `AW_IN.D1` bereitgestellte `WORD`-Wert wird an den Eingang `Convert.IN` übergeben.
   - Der Ausgang `Convert.OUT` liefert den konvertierten `UINT`-Wert, der an den Plug `AUI_OUT.D1` weitergegeben wird.

Die Umwandlung erfolgt nach der IEC 61131-3 Standardkonvertierung `WORD_TO_UINT`.

Technische Besonderheiten
- **Bit-Reinterpretation, werterhaltend** (Zielbreite ist gleich groß oder größer als der Quell-Bit-String).
- **Reine Composite-Struktur**: Der FB besitzt keine eigene Zustandsmaschine oder Logik, sondern delegiert die gesamte Funktionalität an den eingebetteten Standardbaustein.
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind als unidirektionale Typen ausgelegt, was eine einfache und gerichtete Datenweitergabe ermöglicht.
- **Wiederverwendbarkeit**: Der Baustein kann in Projekten eingesetzt werden, in denen eine Schnittstellenanpassung zwischen WORD- und UINT-basierten Komponenten erforderlich ist.

Zustandsübersicht
Da es sich um einen zusammengesetzten Baustein ohne eigene Zustandsmaschine handelt, gibt es keine internen Zustände. Der FB verhält sich strikt ereignisgesteuert: Auf jedes Ereignis am Eingang folgt – nach abgeschlossener Konvertierung – ein Ereignis am Ausgang.

Anwendungsszenarien

- **Systemintegration**: Verbindung einer Komponente, die Daten als `WORD` über einen AW-Adapter bereitstellt, mit einer anderen Komponente, die `UINT` über einen AUI-Adapter erwartet.
- **Protokollanpassung**: Einsatz in Middleware- oder Gateway-Funktionen, um unterschiedliche Datenformate innerhalb einer Steuerungsarchitektur zu harmonisieren.
- **Modulare Erweiterung**: Einfaches Einfügen in bestehende 4diac-Netzwerke, ohne dass Änderungen an den angrenzenden Bausteinen erforderlich sind.

Vergleich mit ähnlichen Bausteinen

Andere Konvertierungsbausteine (z. B. `F_WORD_TO_DWORD` oder `F_INT_TO_UINT`) arbeiten direkt auf Datenports. Der `AW_TO_AUI` hingegen ist speziell für die Adapter-basierte Kommunikation ausgelegt und kapselt sowohl die Typumwandlung als auch die Adapterlogik in einer Einheit. Dadurch wird die Handhabung von Adapter-Schnittstellen vereinfacht, und der Netzwerkentwurf bleibt übersichtlich.

Fazit

Der Funktionsblock `AW_TO_AUI` bietet eine kompakte und zuverlässige Lösung zur Adapterkonvertierung von `WORD` nach `UINT`. Durch die Verwendung eines standardisierten IEC-61131-Konvertierungsbausteins und der klaren Ereignissteuerung ist er leicht verständlich und wartbar. Er eignet sich besonders für Projekte, die eine saubere Trennung zwischen unterschiedlichen Datentypen auf Adapterebene erfordern.

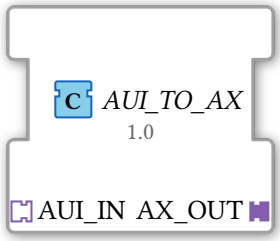
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

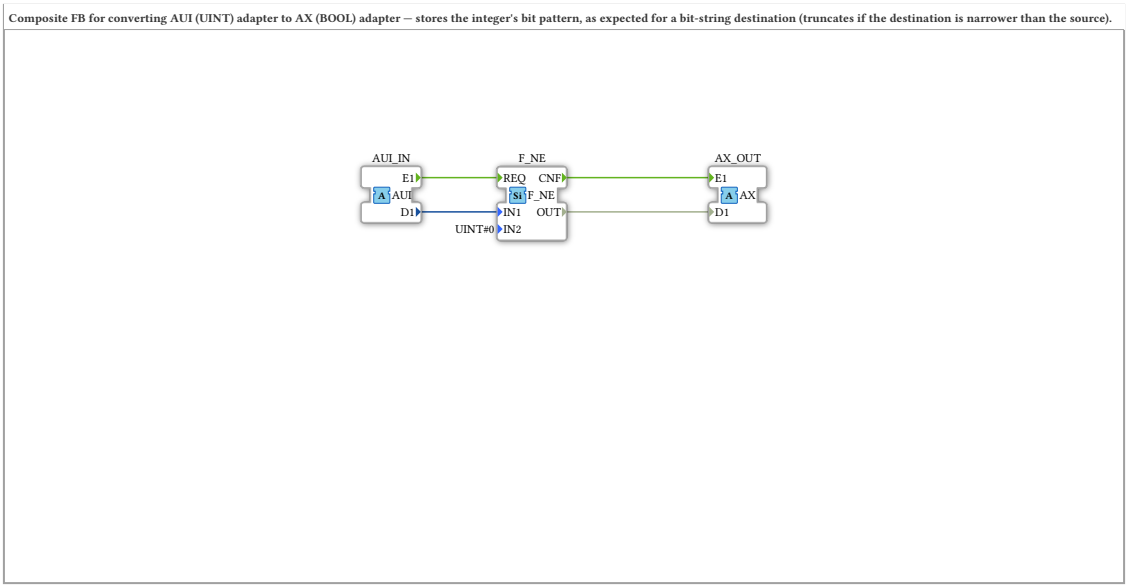
AUI_AX

AUI_TO_AX

UINT Adapter Input – *AUI*



AX – BOOL Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUI_TO_AX** ist ein Composite-Baustein, der einen AUI-Adapter (Typ *AUI* mit einem UINT-Wert) in einen AX-Adapter (Typ *AX* mit einem BOOL-Wert) umwandelt. Er dient als bidirektionale Konvertierungsschnittstelle zwischen zwei unterschiedlichen Adaptertypen und wird typischerweise eingesetzt, um digitale Vergleichsergebnisse für die Weiterverarbeitung in BOOL-basierten Adaptern bereitzustellen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge. Die Ereignissteuerung erfolgt ausschließlich über den Socket-Adapter **AUI_IN**.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge. Das Ergebnis-Event wird über den Plug-Adapter **AX_OUT** weitergegeben.

Daten-Eingänge

Keine direkten Daten-Eingänge. Der zu verarbeitende UINT-Wert wird über den Socket-Adapter **AUI_IN** bereitgestellt.

Daten-Ausgänge

Keine direkten Daten-Ausgänge. Das BOOL-Ergebnis wird über den Plug-Adapter **AX_OUT** ausgegeben.

Adapter

- **AUX_IN (Socket)** – Typ: *adapter::types::unidirectional::AUI*
  - Ereignis: E1 (auslösendes Ereignis für den Vergleich)
  - Daten: D1 (UINT-Wert, der mit 0 verglichen wird)
- **AX_OUT (Plug)** – Typ: *adapter::types::unidirectional::AX*
  - Ereignis: E1 (Bestätigungsereignis nach erfolgtem Vergleich)
  - Daten: D1 (BOOL-Ergebnis: TRUE wenn D1 ≠ 0, sonst FALSE)

Funktionsweise

Der Baustein enthält intern eine Instanz des Vergleichsbausteins *iec61131::comparison::F_NE* (Ungleichheit). Der Ablauf ist:

1. Ein Ereignis am Socket **AUI_IN.E1** triggert den Eingang **REQ** des F_NE-Bausteins.
2. Der vom Socket über **AUI_IN.D1** gelieferte UINT-Wert wird mit der Konstanten *UINT#0* verglichen.
3. Das Vergleichsergebnis (BOOL) wird am Ausgang **F_NE.OUT** bereitgestellt.
4. Das Ergebnis-Event **F_NE.CNF** wird an den Plug **AX_OUT.E1** weitergeleitet.
5. Das BOOL-Ergebnis wird gleichzeitig über die Datenverbindung an **AX_OUT.D1** übergeben.

Somit gilt:

- AUI_IN.D1 = 0 ⟶ AX_OUT.D1 = FALSE
- AUI_IN.D1 ≠ 0 ⟶ AX_OUT.D1 = TRUE

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Lizenz:** Eclipse Public License 2.0 (EPL-2.0)
- **Autor:** Franz Höpfinger, HR Agrartechnik GmbH
- **Paket:** `adapter::conversion::unidirectional`
- **Interner Baustein:** `iec61131::comparison::F_NE` mit Parameter `IN2 = UINT#0`
- Der Baustein ist ein reines Composite, enthält keine eigene Zustandsmaschine. Die gesamte Logik wird durch den eingebetteten F_NE realisiert.

Zustandsübersicht

Da es sich um einen Composite-Baustein ohne eigene ECC (Execution Control Chart) handelt, existiert keine explizite Zustandsmaschine. Das Verhalten wird vollständig durch den internen F_NE-Baustein bestimmt, der einen einfachen Vergleich ohne Zwischenspeicherung durchführt. Der Baustein ist somit zustandslos.

Anwendungsszenarien

- **Schwellwertdetektion:** Überführung eines analogen (UINT-)Sensorwerts in ein digitales Signal (BOOL), z. B. „Wert ungleich Null“ als Alarm- oder Steuersignal.
- **Adapter-Konvertierung:** Einsatz in Systemen, die unterschiedliche Adapterstandards (AUI ↔ AX) verwenden, um Kompatibilität zwischen Komponenten herzustellen.
- **Einfache Logikverkettung:** Verwendung als binäre Entscheidungsstufe in größeren Composite-Funktionsblöcken.

Vergleich mit ähnlichen Bausteinen

Baustein	Eingabetyp	Ausgabetyp	Funktion
AUI_TO_AX	AUI (UINT)	AX (BOOL)	Ungleichheit mit Null
AUI_TO_AX_INV (hypothetisch)	AUI (UINT)	AX (BOOL)	Gleichheit mit Null (invertiert)
CONV_UINT_TO_BOOL (direkt)	UINT (Data)	BOOL (Data)	Direkter Datenwandler, kein Adapter

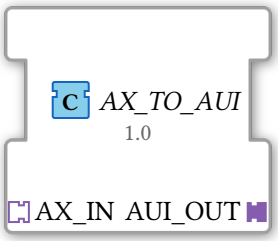
Der Baustein hebt sich durch die vollständige Adapterkonformität und die integrierte Ereignissteuerung von einfachen Datenkonvertern ab.

Fazit

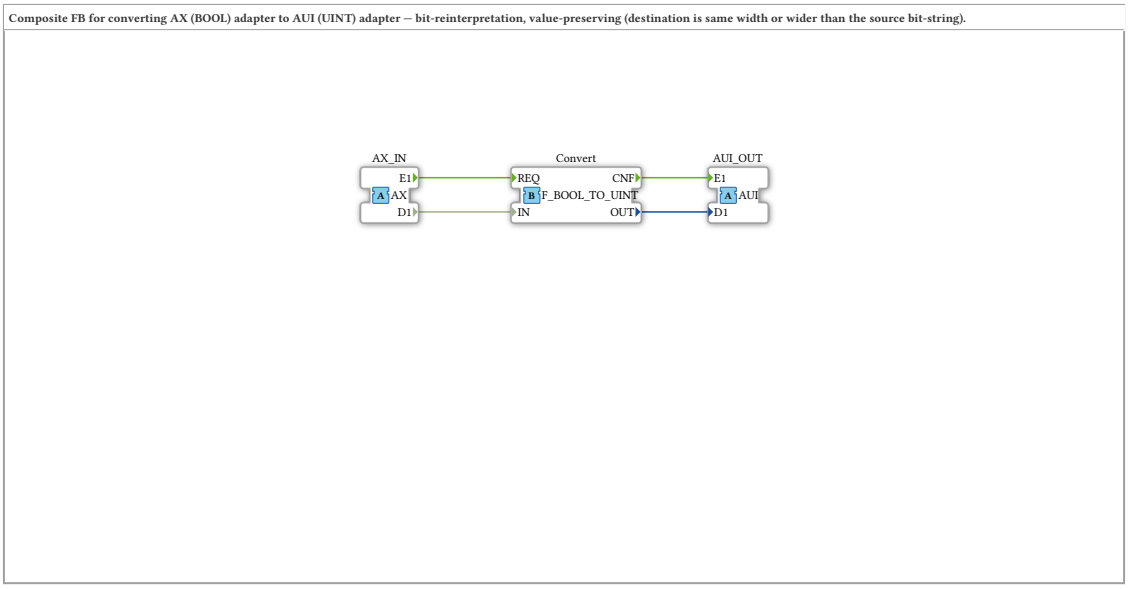
AUI_TO_AX ist ein kompakter, zweckorientierter Composite-FB zur Umwandlung eines UINT-basierten AUI-Adapters in einen BOOL-basierten AX-Adapter. Dank der Nutzung des standardisierten F_NE-Vergleichs ist die Funktion zuverlässig und leicht nachvollziehbar. Der Baustein eignet sich besonders für die Integration in Adapter-basierte Automatisierungsarchitekturen, bei denen ein einfacher Null-Vergleich auf die Signalebene gehoben werden muss.

AX_TO_AUI

BOOL Adapter Input – *AX*



AUI – UINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der AX_TO_AUI ist ein Composite Funktionsblock (FB), der einen BOOL-Wert aus einem AX-Adapter in einen UINT-Wert für einen AUI-Adapter umwandelt. Er kapselt die Konvertierungslogik und ermöglicht eine nahtlose Integration zwischen unidirektionalen Adaptern unterschiedlicher Datentypen. Die Umwandlung erfolgt ereignisgesteuert über einen internen IEC 61131 Konvertierungsbaustein.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Datenschnittstellen, sondern ausschließlich Adapter-Schnittstellen. Die nachfolgende Aufstellung beschreibt die über die Adapter bereitgestellten bzw. benötigten Signale.

Ereignis-Eingänge

- **AX_IN.E1** (Ereignis vom angeschlossenen AX-Adapter): Löst die Konvertierung des bereitgestellten BOOL-Werts aus.

Ereignis-Ausgänge

- **AUI_OUT.E1** (Ereignis zum angeschlossenen AUI-Adapter): Signalisiert, dass der umgewandelte UINT-Wert anliegt.

Daten-Eingänge

- **AX_IN.D1** (BOOL): Boolescher Eingangswert (z. B. 0 oder 1), der in einen UINT-Wert konvertiert werden soll.

Daten-Ausgänge

- **AUI_OUT.D1** (UINT): Ganzzahliger Ausgangswert (0 oder 1), der dem konvertierten BOOL-Eingang entspricht.

Adapter

- **AX_IN** (Socket) – Typ: `adapter::types::unidirectional::AX` (BOOL)
- **AUI_OUT** (Plug) – Typ: `adapter::types::unidirectional::AUI` (UINT)

Funktionsweise

Der FB arbeitet als einfache Pipeline: Ein eingehendes Ereignis am Socket AX_IN.E1 triggert den internen Funktionsblock `F_BOOL_TO_UINT`. Dieser liest den aktuellen Wert von AX_IN.D1, wandelt ihn in einen UINT-Wert um (BOOL FALSE → 0, BOOL TRUE → 1) und legt das Ergebnis an AUI_OUT.D1 an. Nach Abschluss wird ein Ereignis über AUI_OUT.E1 ausgegeben. Der gesamte Vorgang ist atomar und ereignisgesteuert, es existiert kein interner Zustand.

Technische Besonderheiten

- **Bit-Reinterpretation, werterhaltend** (Zielbreite ist gleich groß oder größer als der Quell-Bit-String).
- Der Baustein ist als Composite FB realisiert, d. h. er besteht aus einem Netzwerk untergeordneter FB-Instanzen (hier nur `F_BOOL_TO_UINT`).
- Die Umwandlung folgt der IEC 61131-3 Konvertierungsregel: BOOL FALSE = 0, BOOL TRUE = 1.
- Die Verwendung von Adaptern ermöglicht eine lose Kopplung zwischen Komponenten unterschiedlicher Domänen (BOOL-Steuerung und UINT-Kommunikation).
- Der FB besitzt keine eigene Zustandsmaschine; die Abarbeitung ist rein ereignisgesteuert.

Zustandsübersicht

Da es sich um einen Composite FB ohne eigenes ECC (Execution Control Chart) handelt, gibt es keine expliziten Zustände. Die Funktionalität reduziert sich auf die ereignisgesteuerte Ausführung des internen FBs.

Anwendungsszenarien

- **Steuerungsschnittstelle**: Ein BOOL-Signal (z. B. Schalter Ein/Aus) wird in einen numerischen Wert gewandelt, um es über einen AUI-Adapter an eine übergeordnete Steuerung zu senden.
- **Protokollanpassung**: Anbindung einer BOOL-basierten Sensorik an ein System, das UINT-Werte erwartet (z. B. zur Weiterverarbeitung in arithmetischen Blöcken).
- **Vereinheitlichung von Adapterschnittstellen**: Der FB dient als Adapter zwischen zwei unterschiedlichen unidirektionalen Adaptertypen und vermeidet Doppelentwicklungen.

Vergleich mit ähnlichen Bausteinen

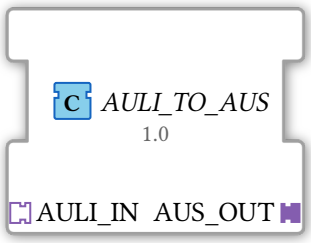
- **AX_TO_AUI** wandelt BOOL → UINT um.
- **AX_TO_ASI** würde BOOL → SINT konvertieren, **AX_TO_ADI** BOOL → DINT – alle folgen demselben Muster, jedoch mit unterschiedlichen Zieltypen.
- Im Gegensatz zu einem generischen `F_BOOL_TO_UINT` bietet dieser FB eine gekapselte Adapterschnittstelle und ist direkt in Adapter-basierte Architekturen integrierbar.

Fazit

Der AX_TO_AUI ist ein kleiner, spezialisierter Baustein zur Konvertierung von BOOL- in UINT-Daten über Adapterschnittstellen. Er erfüllt eine typische Integrationsaufgabe in IEC 61499-basierten Systemen, bei der unterschiedliche Datentypen über unidirektionale Adapter verbunden werden müssen. Seine Composite-Struktur macht die Implementierung transparent und erweiterbar.

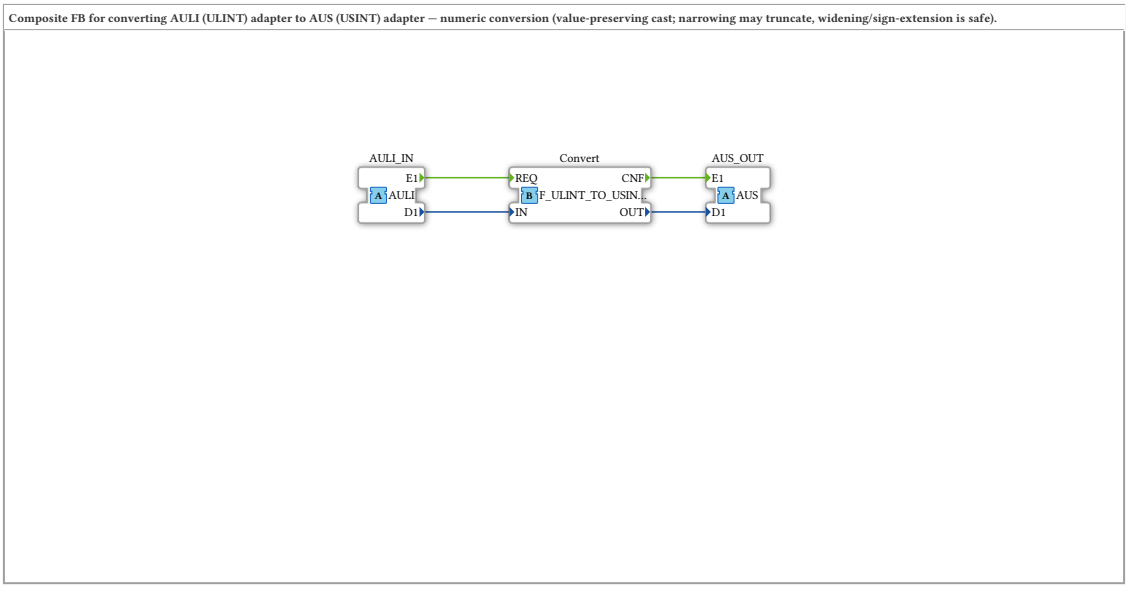
AULI_AUS

AULI_TO_AUS



ULINT Adapter Input – *AULI*

AUS – USINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AULI_TO_AUS** ist ein Composite-Baustein, der eine Konvertierung von einem **AULI-Adapter** (ulint, 64-Bit unsigned) zu einem **AUS-Adapter** (usint, 8-Bit unsigned) vornimmt. Er kapselt die IEC-61131-Funktion **F_ULINT_TO_USINT** in einer adapterbasierten Umgebung, sodass Daten über standardisierte Adapter ausgetauscht werden können.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Daten-Ein-/Ausgänge auf FB-Ebene. Die gesamte Kommunikation erfolgt über die beiden Adapter.

Ereignis-Eingänge

- über **Socket AULI_IN** (Typ AULI): Ereignis **E1** – initiiert die Datenübergabe und Konvertierung.

Ereignis-Ausgänge

- über **Plug AUS_OUT** (Typ AUS): Ereignis **E1** – wird ausgelöst, sobald der konvertierte Wert am Ausgang anliegt.

Daten-Eingänge

- über **Socket AULI_IN** (Typ AULI): Daten-Eingang **D1** (ULINT) – der zu konvertierende 64-Bit-Wert.

Daten-Ausgänge

- über **Plug AUS_OUT** (Typ AUS): Daten-Ausgang **D1** (USINT) – der konvertierte 8-Bit-Wert.

Adapter

Bezeichnung	Typ	Richtung	Beschreibung
AULI_IN	AULI	Socket	Eingangsadapter für ULINT-Daten
AUS_OUT	AUS	Plug	Ausgangsadapter für USINT-Daten

Funktionsweise

1. Über den Socket **AULI_IN** empfängt der FB ein Ereignis an **E1**. Gleichzeitig wird der zugehörige ULINT-Wert an **D1** bereitgestellt.
2. Dieses Ereignis wird an den internen Baustein **F_ULINT_TO_USINT** (Name: `Convert`) weitergeleitet (über die Ereignisverbindung `REQ`).
3. Der ULINT-Wert wird an den Dateneingang **IN** des Konverters übergeben.
4. Nach Abschluss der Konvertierung erzeugt der Konverter das Ereignis **CNF**.

5. Dieses Ereignis wird zum Plug **AUS_OUT** auf dessen Ereigniseingang **E1** übertragen, während der konvertierte USINT-Wert vom Ausgang **OUT** des Konverters an den Datenausgang **D1** des Plugs gelegt wird.
6. Der angeschlossene Empfänger kann nun das Ereignis und den USINT-Wert verarbeiten.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-Baustein** – Er enthält eine interne Instanz von **F_ULINT_TO_USINT** aus der IEC-61131-Bibliothek.
- **Adapterbasierte Kommunikation** – Keine direkten Daten/Events auf der obersten FB-Ebene; die Ein-/Ausgabe erfolgt ausschließlich über die mitgelieferten Adapter (AULI / AUS).
- **Datenverlust möglich** – Da der USINT-Datentyp nur 8 Bit umfasst, gehen Werte > 255 verloren (abgeschnitten oder wrap-around, je nach Implementierung des Konverters).
- **Keine eigene Zustandsmaschine** – Der FB ist ein reines Netzwerk aus bestehenden Bausteinen und besitzt keinen eigenen Zustandsautomaten.

Zustandsübersicht

Aufgrund der Composite-Natur und des Fehlens einer eigenen Zustandsmaschine entfällt eine Zustandsübersicht. Das Verhalten wird vollständig durch den inneren FB **F_ULINT_TO_USINT** bestimmt.

Anwendungsszenarien

- **Integration von ULINT-Datenquellen** (z. B. hochauflösende Zähler, Zeitstempel oder große Sensordaten) in ein Automatisierungssystem, das aufgrund von Speicher- oder Protokollbeschränkungen nur USINT-Werte verarbeiten kann.
- **Adapter-Konvertierung** in einer verteilten Anwendung, um zwischen zwei unterschiedlichen Adapter-Typen (AULI ↔ AUS) zu vermitteln, ohne die darüber liegende Logik zu ändern.
- **Prototypische Umstellungen** von 64-Bit auf 8-Bit Datenpfade während der Entwicklung.

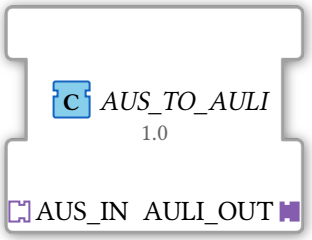
Vergleich mit ähnlichen Bausteinen

Ähnliche Bausteine wie **AULI_TO_AINT** (ULINT → AINT) oder **AULI_TO_AWORD** (ULINT → AWORD) realisieren analoge Konvertierungen für andere Ausgabetypen. Der vorliegende Baustein spezialisiert sich auf **USINT** – eine besonders speicherschonende Zielgröße. Im Gegensatz zu einer direkten Datenkonvertierung auf Applikationsebene bietet der adapterbasierte Ansatz eine strukturierte, wiederverwendbare Schnittstellenkapselung, die den Austausch von Datenströmen in standardisierten Industrieprotokollen erleichtert.

Fazit

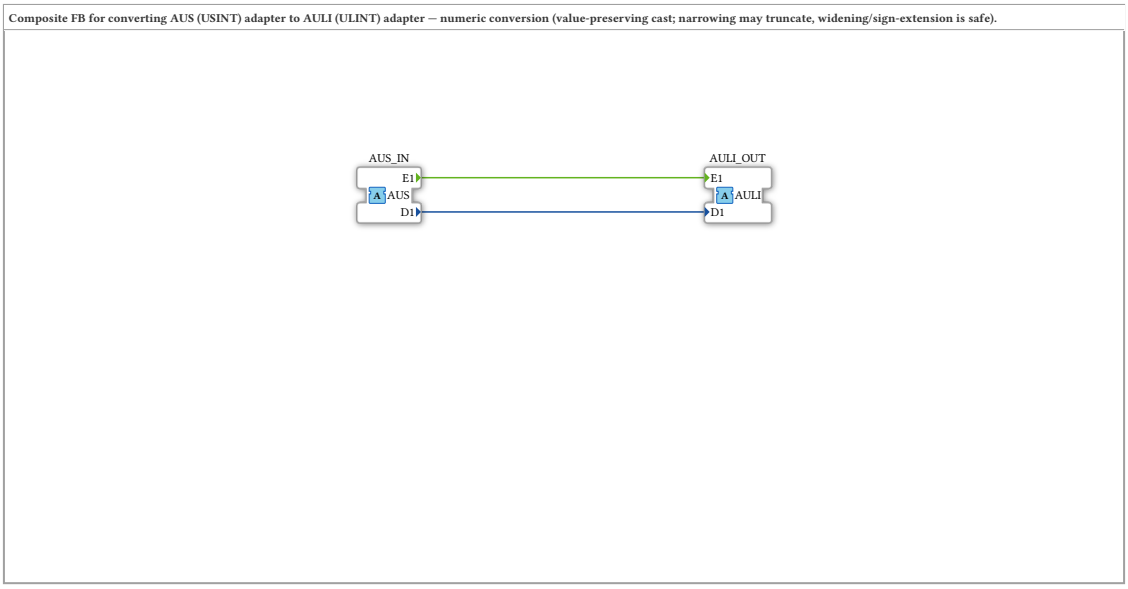
AULI_TO_AUS ist ein praktischer Composite-Baustein, der eine IEC-61131-Konvertierungsfunktion in eine adapterbasierte Umgebung einbettet. Er ermöglicht die transparente Umwandlung von ULINT- zu USINT-Werten über die entsprechenden Adapter und eignet sich insbesondere für Szenarien, in denen hohe Genauigkeit oder große Wertebereiche auf ein kleineres Datenformat reduziert werden müssen. Der Baustein ist einfach einsetzbar, erfordert aber Aufmerksamkeit bei möglichen Datenverlusten.

AUS_TO_AULI



USINT Adapter Input – *AUS*

AULI – ULINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `AUS_TO_AULI` ist ein Composite-Baustein zur Konvertierung eines AUS-Adapters (UNSIGNED SHORT INTEGER) in einen AULI-Adapter (UNSIGNED LONG INTEGER). Er ermöglicht die direkte Weiterleitung von Ereignissen und Daten zwischen diesen beiden unidirektionalen Adaptertypen, wobei eine implizite Typumwandlung von USINT nach ULINT stattfindet.

Schnittstellenstruktur

Ereignis-Eingänge

- **E1** (über `AUS_IN`): Ereigniseingang des angeschlossenen AUS-Adapters. Wird direkt an den Ausgang weitergeleitet.

Ereignis-Ausgänge

- **E1** (über `AULI_OUT`): Ereignisausgang des AULI-Adapters. Entspricht dem eingehenden Ereignis von `AUS_IN.E1`.

Daten-Eingänge

- **D1** (über `AUS_IN`): Dateneingang vom Typ **USINT** (AUS-Adapter). Der Wert wird an den Ausgang übertragen.

Daten-Ausgänge

- **D1** (über `AULI_OUT`): Datenausgang vom Typ **ULINT** (AULI-Adapter). Enthält den konvertierten Wert des Eingangs.

Adapter

- **AUS_IN** (Socket): Aufnahmeschnittstelle für einen AUS-Adapter (Datentyp USINT, Ereignis E1, Daten D1).
- **AULI_OUT** (Plug): Ausgabeschnittstelle für einen AULI-Adapter (Datentyp ULINT, Ereignis E1, Daten D1).

Funktionsweise

Der Baustein verbindet die Elemente des eingehenden AUS-Adapters direkt mit den korrespondierenden Elementen des ausgehenden AULI-Adapters:

- Ein eingehendes Ereignis am `AUS_IN.E1` wird unverzögert an `AULI_OUT.E1` weitergegeben.
- Der Datenwert `AUS_IN.D1` (USINT) wird ohne zusätzliche Logik auf `AULI_OUT.D1` (ULINT) abgebildet. Die Typkonvertierung erfolgt implizit durch die unterschiedlichen Adapterdefinitionen; ein USINT-Wert wird dabei in den entsprechenden ULINT-Wert erweitert.

Technische Besonderheiten

- **Numerische Umwandlung**: wertgetreue Zahlenwert-Konvertierung (Verengung kann abschneiden, Erweiterung/Vorzeichenerweiterung ist sicher).
- **Composite-Baustein**: Der FB enthält kein eigenes ECC (Execution Control Chart) und keine Zustandslogik, sondern besteht ausschließlich aus einer Ereignis- und Datenverbindung.
- **Implizite Typumwandlung**: Die Konvertierung von USINT nach ULINT wird durch die Adapterdefinitionen der Bibliothek `adapter::types::unidirectional` bereitgestellt. Eine explizite Typumwandlung im FB ist nicht erforderlich.
- **Unidirektionalität**: Der FB unterstützt nur eine Datenflussrichtung (von AUS zu AULI). Eine Rückwärtskonvertierung ist nicht vorgesehen.

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Er arbeitet rein ereignisgesteuert und kombinatorisch: Bei jedem eingehenden Ereignis wird der aktuelle Datenwert sofort weitergeleitet. Es werden keine internen Zustände gespeichert.

Anwendungsszenarien

- **Adapter-Integration**: Wenn ein bestehendes Modul (z. B. ein Sensor) einen AUS-Adapter ausgibt, aber ein nachfolgendes Modul (z. B. eine Steuerung) einen AULI-Adapter erwartet, kann dieser FB als konvertierende Zwischenschicht eingesetzt werden.
- **Systemerweiterung**: Einfaches Upgrade von USINT-basierten Schnittstellen auf ULINT ohne Änderung der umgebenden Logik.
- **Testumgebungen**: Als Platzhalter oder Brücke in Adapter-basierten Teststrukturen.

Vergleich mit ähnlichen Bausteinen

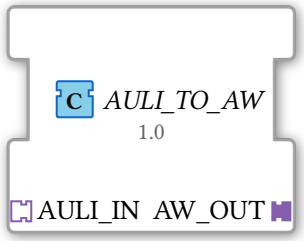
- **AUS_TO_AULI vs. generische Typkonverter**: Im Gegensatz zu einem universellen Konverter ist dieser FB speziell für die AUS/AULI-Adapter zugeschnitten und benötigt keine zusätzliche Parametrierung. Er ist kompakter und schneller in der Entwicklung.
- **AUS_TO_AULI vs. manuelle Adapter-Programmierung**: Statt selbstständig einen Adapter-zu-Adapter-Konverter zu programmieren, bietet dieser fertige Baustein eine sofort nutzbare, getestete Lösung.

Fazit

Der FB `AUS_TO_AULI` ist ein einfacher, aber effektiver Composite-Baustein zur Konvertierung eines AUS-Adapters (USINT) in einen AULI-Adapter (ULINT). Durch seine direkte Durchschaltung von Ereignissen und Daten sowie die implizite Typumwandlung stellt er eine saubere und wartbare Lösung für die Integration unterschiedlicher Adapter-Schnittstellen dar.

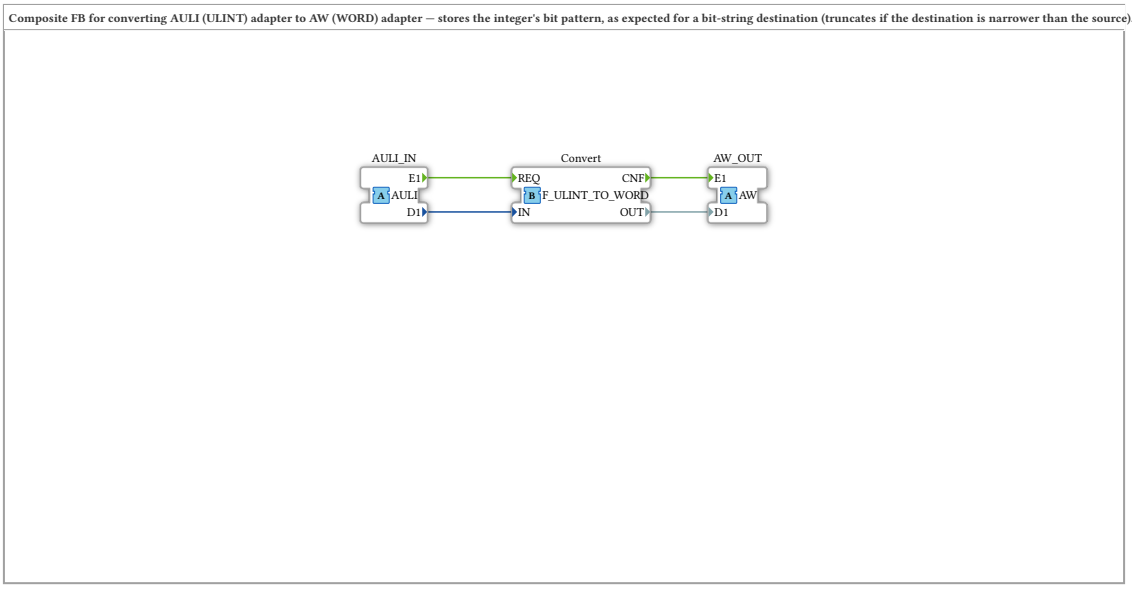
AULI_AW

AULI_TO_AW



ULINT Adapter Input – *AULI*

AW – WORD Adapter Output



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock **AULI_TO_AW** ist ein zusammengesetzter Baustein (Composite FB), der eine ULINT-Adapter-Schnittstelle (AULI) in eine WORD-Adapter-Schnittstelle (AW) umwandelt. Er wird dort eingesetzt, wo ein unidirektionaler Datentyp-Wechsel von vorzeichenloser 64-Bit-Ganzzahl auf 16-Bit-Wort erforderlich ist.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
E1 (über AULI_IN)	Event	Startet die Konvertierung

Ereignis-Ausgänge

Name	Typ	Kommentar
E1 (über AW_OUT)	Event	Signalisiert die Fertigstellung der Konvertierung

Daten-Eingänge

Name	Typ	Kommentar
D1 (über AULI_IN)	ULINT	ULINT-Eingangswert

Daten-Ausgänge

Name	Typ	Kommentar
D1 (über AW_OUT)	WORD	WORD-Ausgangswert

Adapter

Typ	Rolle	Name	Kommentar
adapter::types::unidirectional::AULI	Socket	AULI_IN	ULINT-Adapter Eingang
adapter::types::unidirectional::AW	Plug	AW_OUT	WORD-Adapter Ausgang

Funktionsweise

Der Baustein verwendet intern die IEC-61131-Funktion **F_ULINT_TO_WORD**, um den ankommenden ULINT-Wert (über den AULI_IN-Adapter) in einen WORD-Wert zu wandeln. Der Ablauf ist rein ereignisgesteuert:

1. Ein eingehendes Ereignis am AULI_IN-Adapter (E1) triggert die Konvertierung.
2. Der ULINT-Wert (D1) wird dem FB **F_ULINT_TO_WORD** übergeben.
3. Nach Abschluss der Wandlung wird das Ausgangsereignis am AW_OUT-Adapter (E1) gesendet, und der WORD-Wert steht am Ausgangsdatenpunkt (D1) zur Verfügung.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Unidirektionale Adapter**: Sowohl AULI_IN als auch AW_OUT sind unidirektionale Adapter, d.h. der Datenfluss erfolgt nur in eine Richtung.
- **Komposition**: Der Baustein ist aus einem einzigen integrierten Konvertierungs-FB aufgebaut, was ihn leicht nachvollziehbar und wartbar macht.
- **Typkonvertierung**: Die Wandlung von ULINT (64-Bit) nach WORD (16-Bit) führt bei Werten größer als 65535 zu einem Überlauf – der Anwender muss sicherstellen, dass der Eingangswert im Bereich von 0…65535 liegt.

Zustandsübersicht

Da es sich um einen reinen Composite-FB ohne eigene Zustandsmaschine handelt, gibt es keine internen Zustände. Die gesamte Logik wird durch den eingebetteten FB **F_ULINT_TO_WORD** realisiert, der eine einfache, ereignisgesteuerte Funktionsausführung besitzt.

Anwendungsszenarien

- **Schnittstellenanpassung**: Wenn ein AULI-Adapter (z.B. von einer Sensor-Einheit) in ein System eingebunden werden muss, das nur WORD-Adapter-Schnittstellen (AW) unterstützt.
- **Datenreduktion**: Reduzierung der Datenbreite von 64-Bit auf 16-Bit für nachfolgende Verarbeitungsschritte.
- **Legacy-Integration**: Einbinden älterer Komponenten, die nur mit 16-Bit-Daten arbeiten, in eine moderne Umgebung, die ULINT liefert.

Vergleich mit ähnlichen Bausteinen

Vergleichbare Bausteine sind z.B. **ULINT_TO_INT** oder **ULINT_TO_DWORD**, die andere Zielformate anbieten. **AULI_TO_AW** ist speziell auf die unidirektionalen Adapter-Typen der 4diac-Welt zugeschnitten. Im Unterschied zu einer einfachen Typkonvertierung übernimmt der Baustein auch die Ereignis- und Adapter-Verknüpfung, was die Integration in Adapter-basierte Architekturen erheblich vereinfacht.

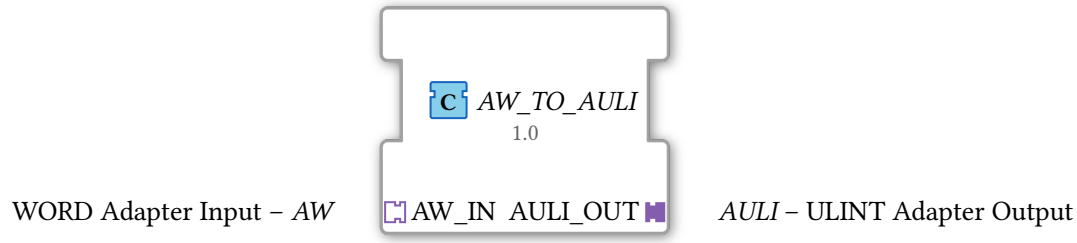
Fazit

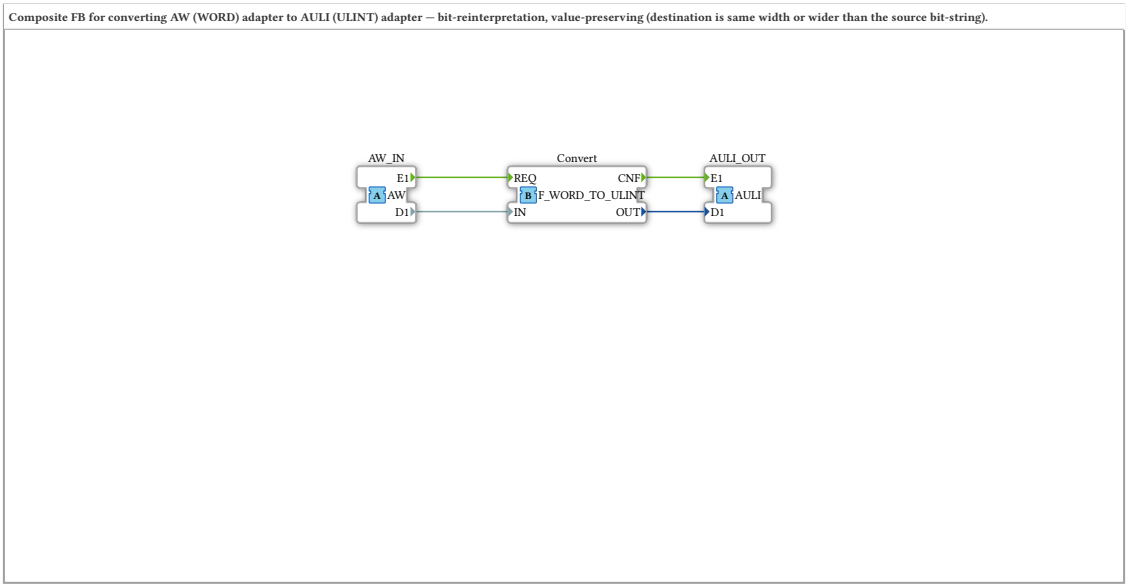
AULI_TO_AW ist ein praktischer, kompakter Konvertierungsbaustein, der die Lücke zwischen ULINT- und WORD-Adapter-Schnittstellen schließt. Durch die Kapselung der IEC-Funktion in einem Composite-FB wird der Anwender von Adapter-Details entlastet und erhält eine saubere, ereignisgesteuerte Transformationslösung. Der Baustein eignet sich besonders für Szenarien, in denen Datenbreiten reduziert und gleichzeitig die Adapter-Struktur beibehalten werden soll.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AW_TO_AULI





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Composite FB `AW_TO_AULI` dient der Umwandlung eines unidirektionalen Adapters vom Typ `AW` (Datenformat WORD) in einen Adapter vom Typ `AULI` (Datenformat ULINT). Die Konvertierung erfolgt intern mittels des Standardbausteins `F_WORD_TO_ULINT` aus der IEC 61131-Bibliothek.

Schnittstellenstruktur

Ereignis-Eingänge

- **AW_IN.E1** – Ereigniseingang über den Socket `AW_IN`. Löst die Konvertierung aus.

Ereignis-Ausgänge

- **AULI_OUT.E1** – Ereignisausgang über den Plug `AULI_OUT`. Signalisiert den Abschluss der Konvertierung.

Daten-Eingänge

- **AW_IN.D1** – Dateneingang vom Typ WORD (über Socket `AW_IN`).

Daten-Ausgänge

- **AULI_OUT.D1** – Datenausgang vom Typ ULINT (über Plug `AULI_OUT`).

Adapter

- **Socket:** `AW_IN` (Typ: `adapter::types::unidirectional::AW`)
- **Plug:** `AULI_OUT` (Typ: `adapter::types::unidirectional::AULI`)

Funktionsweise

Sobald ein Ereignis am Ereigniseingang `AW_IN.E1` eintrifft, wird der interne Baustein `F_WORD_TO_ULINT` aufgerufen. Dieser liest den aktuellen Wert des Dateneingangs `AW_IN.D1` (WORD) und wandelt ihn in

einen ULINT-Wert um. Nach erfolgreicher Konvertierung wird am Ereignisausgang `AULI_OUT.E1` ein Ereignis ausgegeben und der konvertierte Wert am Datenausgang `AULI_OUT.D1` bereitgestellt.

Technische Besonderheiten

- **Bit-Reinterpretation, werterhaltend** (Zielbreite ist gleich groß oder größer als der Quell-Bit-String).
- Der FB ist als Composite realisiert und kapselt die Konvertierungslogik vollständig.
- Die verwendete Konvertierungsfunktion `F_WORD_TO_ULINT` entspricht der IEC 61131-3 Norm.
- Die Schnittstellen basieren auf unidirektionalen Adaptern – Ereignis- und Datenfluss erfolgen nur in eine Richtung.
- Der FB besitzt keine Zustandsmaschine; die Funktionalität ist rein ereignisgesteuert.

Zustandsübersicht

Der FB enthält keine expliziten Zustände. Die Abarbeitung erfolgt linear: Ereignis ⟶ Konvertierung ⟶ Ausgabe.

Anwendungsszenarien

- Einbindung von Sensoren oder Aktoren, die WORD-Daten liefern, in Systeme die ULINT-Werte erwarten.
- Adapterbasierte Protokollkonvertierung in der Automatisierungstechnik (z.B. Landtechnik).
- Wiederverwendbare Kapselung der Standard-Konvertierungsfunktion für modulare Steuerungsprojekte.

Vergleich mit ähnlichen Bausteinen

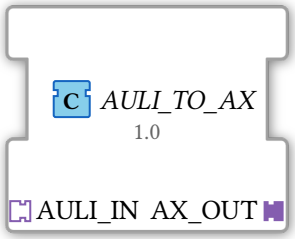
- **Direkte Verwendung von `F_WORD_TO_ULINT`:** Erfordert manuelle Verkabelung von Ereignis- und Datenleitungen. Der Composite FB `AW_TO_AULI` vereinfacht die Integration in adapterorientierte Architekturen.
- **Andere Konvertierungsadapter** (z.B. `AW_TO_ADINT` für DINT oder `AW_TO_ABOOL` für BOOL): Bieten analoge Konvertierungen für andere Ziel-Datentypen.

Fazit

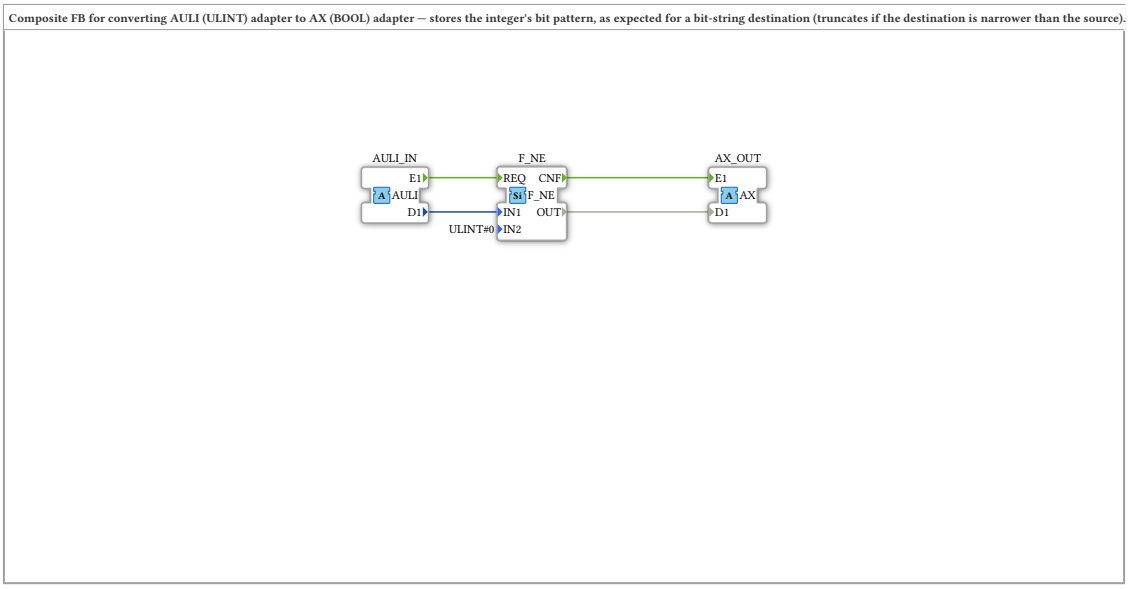
`AW_TO_AULI` stellt eine praktische und leicht einsetzbare Komponente zur Umwandlung von WORD- in ULINT-Daten dar. Die Verwendung von standardisierten Adapterschnittstellen und der IEC-Konvertierungsfunktion gewährleistet Kompatibilität und Zuverlässigkeit. Der FB eignet sich besonders für modulare, adapterbasierte Automatisierungslösungen.

AULI_AX

AULI_TO_AX



ULINT Adapter Input – *AULI* *AX* – BOOL Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AULI_TO_AX** dient als Composite-Baustein zur Umwandlung eines **AULI**-Adapters (Datentyp ULINT) in einen **AX**-Adapter (Datentyp BOOL). Er ermöglicht die Anbindung einer vorzeichenlosen Ganzzahl aus einem Adapter-Socket an einen booleschen Adapter-Plug, wobei der Wert 0 als FALSE und jeder andere Wert als TRUE interpretiert wird.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock besitzt keine direkten Ereignis-Eingänge. Das auslösende Ereignis wird über den Adapter **AULI_IN** bereitgestellt:

- **AULI_IN.E1**: Ereignis zur Verarbeitung des ULINT-Werts und Auslösung der Ausgabe auf dem AX-Adapter.

Ereignis-Ausgänge

Auch die Ereignis-Ausgabe erfolgt über den Adapter:

- **AX_OUT.E1**: Quittungsereignis, das nach erfolgreicher Umwandlung den neuen BOOL-Wert an den angeschlossenen Empfänger weitergibt.

Daten-Eingänge

Alle Dateneingänge sind Teil des Adapter-Sockets:

- **AULI_IN.D1** (ULINT): Der umzuwandelnde Eingangswert.

Daten-Ausgänge

Die Datenausgabe wird über den Adapter-Plug realisiert:

- **AX_OUT.D1** (BOOL): Ergebnis der Umwandlung – TRUE, wenn der Eingangswert ungleich 0 ist, sonst FALSE.

Adapter

- **AULI_IN** (Socket, Typ: `adapter::types::unidirectional::AULI`): Empfängt den ULINT-Wert und das zugehörige Ereignis.
- **AX_OUT** (Plug, Typ: `adapter::types::unidirectional::AX`): Stellt den konvertierten BOOL-Wert sowie das Ausgangsereignis bereit.

Funktionsweise

Der FB arbeitet rein ereignisgesteuert und nutzt intern den Vergleichsbaustein **F_NE** (IEC 61131-3, „ungleich“). Ein eingehendes Ereignis auf **AULI_IN.E1** startet die Ausführung des FB **F_NE**, indem sein Eingang **REQ** getriggert wird. Der Vergleichsbaustein prüft den an **IN1** anliegenden ULINT-Wert gegen den konstanten Wert **ULINT#0** (Parameter **IN2**). Das Ergebnis **OUT** (BOOL) wird an den Datenausgang **AX_OUT.D1** weitergeleitet. Sobald der Vergleich abgeschlossen ist, generiert **F_NE** ein Bestätigungsereignis auf **CNF**, welches an **AX_OUT.E1** weitergegeben wird, um dem angeschlossenen Empfänger die Verfügbarkeit des neuen Werts zu signalisieren.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Composite-FB**: Die Funktionalität wird durch eine interne Verschaltung von Standardbausteinen erreicht – hier nur ein einziger Vergleichsbaustein.
- **Adapter-Kapselung**: Ein- und Ausgänge sind ausschließlich über Adapter definiert, was eine einheitliche und modulare Anbindung in Adapter-basierten Architekturen ermöglicht.
- **Verwendung von IEC 61131-Bibliothek**: Der genutzte **F_NE**-Baustein stammt aus der IEC 61131-3-Bibliothek, was Kompatibilität und Wiederverwendbarkeit fördert.
- **Konstante Vergleichsreferenz**: Der Grenzwert ist fest auf 0 gesetzt, eine Parametrisierung ist nicht vorgesehen.

Zustandsübersicht

Als Composite-FB besitzt **AULI_TO_AX** keine eigenen Zustände im Sinne eines Zustandsautomaten. Der Ablauf wird rein durch die Ereignisverkettung gesteuert:

1. **Warten** auf Ereignis an **AULI_IN.E1**.
2. **Verarbeitung** durch **F_NE** (Vergleich) – quasistatisch, keine Verzögerung.
3. **Ausgabe** des Ergebnisses auf **AX_OUT** (Ereignis und Daten).

Anwendungsszenarien

- **Grenzwertlogik**: Wandelt einen ULINT-Zählerstand (z.B. Stückzahl, Füllstand) in ein einfaches Bool-Signal um, das anzeigt, ob der Wert von Null abweicht.
- **Adapter-Konvertierung**: Ersatz für eine direkte Typumwandlung in Adapter-basierten Steuerungsarchitekturen, wenn nur eine boolesche Weiterverarbeitung erforderlich ist.
- **Signalvereinfachung**: Reduziert komplexe numerische Informationen auf einen binären Zustand für Alarm- oder Statusmeldungen.

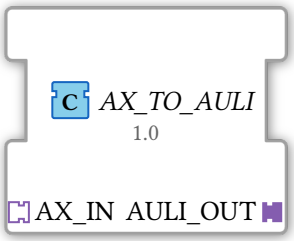
Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu allgemeinen Vergleichs-FBs (z.B. **GT**, **EQ**) ist **AULI_TO_AX** speziell auf die Adapter-Konvertierung ausgelegt und besitzt keine konfigurierbaren Vergleichsoperatoren oder variablen Schwellwerte. Ein direkter Vergleich mit 0 ist fest verdrahtet. Bausteine wie **ULINT_TO_BOOL** (ohne Adapter) erfordern eigene Ereignis- und Datenverbindungen, während dieser FB die gesamte Adapter-Handhabung integriert.

Fazit

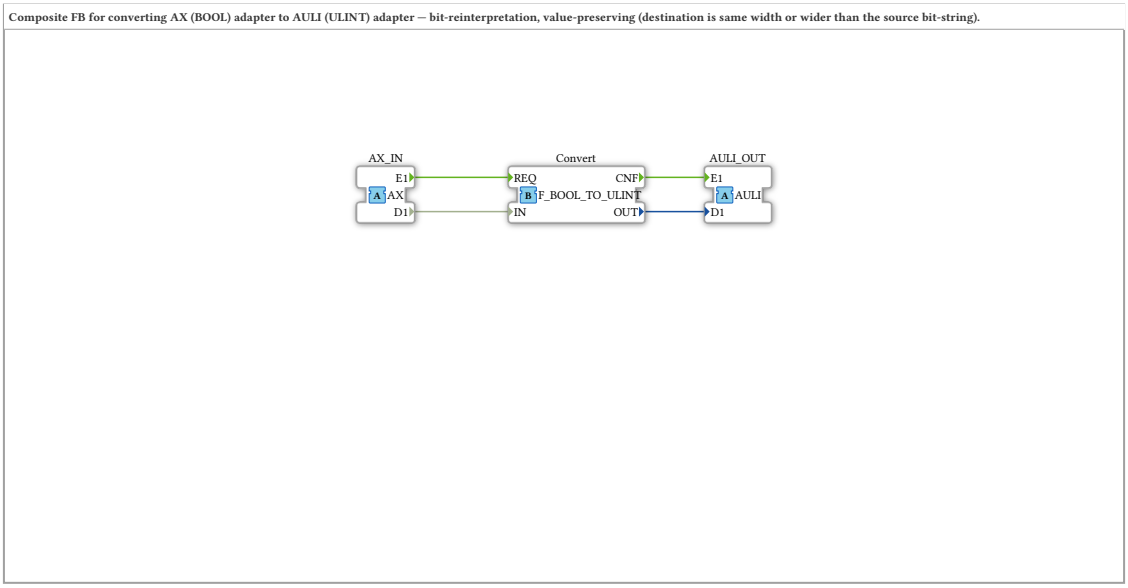
AULI_TO_AX ist ein kompakter und spezialisierter Composite-Baustein für die einfache Umwandlung eines ULINT-Adapter-Signals in ein boolesches Adapter-Signal. Er reduziert den Verdrahtungsaufwand in Adapter-basierten Projekten und sorgt für klare, wartbare Strukturen, wenn nur der Zustand „ungleich Null“ ausgewertet werden muss. Die Implementierung durch einen standardisierten Vergleichsbaustein gewährleistet zuverlässige und ressourcenschonende Ausführung.

AX_TO_AULI



BOOL Adapter Input – *AX*

AULI – ULINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AX_TO_AULI** ist ein Composite-Baustein zur Konvertierung eines BOOL-Adapter-Signals (Typ `AX`) in ein ULINT-Adapter-Signal (Typ `AULI`). Er kapselt die typsichere Umwandlung eines binären Werts in einen vorzeichenlosen 64-Bit-Integer und ermöglicht so die einfache Integration von BOOL-Schnittstellen in ULINT-basierte Steuerungslogiken. Der Baustein wird im Paket `adapter::conversion::unidirectional` bereitgestellt und basiert auf dem standardisierten Konvertierungsbaustein `F_BOOL_TO_ULINT` aus der IEC-61131-Bibliothek.

Schnittstellenstruktur

Ereignis-Eingänge

Keine eigenen Ereignis-Eingänge. Die Ereignissteuerung erfolgt über die angeschlossenen Adapter (siehe Abschnitt **Adapter**).

Ereignis-Ausgänge

Keine eigenen Ereignis-Ausgänge. Die Ergebnisereignisse werden über den Ausgangsadapter bereitgestellt (siehe Abschnitt **Adapter**).

Daten-Eingänge

Keine eigenen Daten-Eingänge. Die zu konvertierenden Daten werden über den Eingangsadapter empfangen (siehe Abschnitt **Adapter**).

Daten-Ausgänge

Keine eigenen Daten-Ausgänge. Das konvertierte Ergebnis wird über den Ausgangsadapter bereitgestellt (siehe Abschnitt **Adapter**).

Adapter

Richtung	Name	Typ	Beschreibung
Socket	AX_IN	adapter::types::unidirectional::AX	BOOL-Adapter-Eingang. Stellt über das Ereignis E1 und das Datum D1 (BOOL-Wert) die Eingangsdaten bereit.
Plug	AULI_OUTa	dapter::types::unidirectional::AULI	ULINT-Adapter-Ausgang. Gibt über das Ereignis E1 und das Datum D1 (ULINT-Wert) das Konvertierungsergebnis aus.

Funktionsweise

Der Baustein enthält ein internes Netzwerk, das aus einer einzigen Instanz des Konvertierungsbausteins F_BOOL_TO_ULINT besteht. Die Verbindungen sind wie folgt:

1. **Ereignisführung**:
   - Das Ereignis AX_IN.E1 (vom Eingangsadapter) wird an den Ereigniseingang Convert.REQ geleitet und startet die Konvertierung.
   - Nach Abschluss der Konvertierung wird das Ereignis Convert.CNF an den Ereignisausgang AULI_OUT.E1 weitergegeben, um abschließend den Ausgangsadapter zu aktualisieren.
2. **Datenfluss**:
   - Der boolesche Wert von AX_IN.D1 wird auf den Dateneingang Convert.IN gespiegelt.
   - Der konvertierte ULINT-Wert von Convert.OUT wird auf AULI_OUT.D1 übertragen.

Somit wird bei jeder steigenden Flanke des Eingangsereignisses der aktuelle BOOL-Wert in einen ULINT-Wert umgewandelt (BOOL ⟶ 0 ⟶ 0 ULINT, BOOL ⟶ 1 ⟶ 1 ULINT) und über den Ausgangsadapter bereitgestellt.

Technische Besonderheiten

- **Bit-Reinterpretation, werterhaltend** (Zielbreite ist gleich groß oder größer als der Quell-Bit-String).
- **Lizenz**: Der Quellcode steht unter der *Eclipse Public License 2.0* (EPL-2.0).
- **Urheber**: Entwickelt von **Franz Höpfinger** für die **HR Agrartechnik GmbH** (Version 1.0, Stand 2026-02-17).
- **Paketstruktur**: Der Baustein ist im Paket adapter::conversion::unidirectional organisiert.
- **Abhängigkeiten**: Er importiert den Baustein iec61131::conversion::F_BOOL_TO_ULINT.
- **Typsicherheit**: Durch die Verwendung von Adaptern wird eine strenge Typüberprüfung bereits zur Entwurfszeit gewährleistet.

Zustandsübersicht

Der Composite-Baustein besitzt keine eigene Zustandsmaschine. Die gesamte Ablaufsteuerung wird durch den internen Baustein F_BOOL_TO_ULINT realisiert, der einen einfachen ereignisgesteuerten Zustandsautomaten umsetzt:

- Im Ruhezustand wartet er auf ein REQ-Ereignis.
- Nach Eintreffen des Ereignisses wird die Konvertierung durchgeführt und unmittelbar der Ausgang mit CNF bestätigt.

Anwendungsszenarien

- **Integration von Binärsensoren** in Steuerungen, die ULINT-Werte erwarten (z. B. Zählerregister, Adressierungslogiken).
- **Adapter-Brückenbildung** in heterogenen Systemen, bei denen ein BOOL-Adapter (z. B. von einem Schalter) an einen ULINT-Adapter (z. B. einer Profil-/Parametrierschnittstelle) angepasst werden muss.
- **Typsichere Konvertierung** in Composite-Bausteinen, um die Handhabung unterschiedlicher Datentypen in einer Applikation zu vereinheitlichen.

Vergleich mit ähnlichen Bausteinen

- **F_BOOL_TO_ULINT**: Der zugrunde liegende Basisbaustein – er bietet die reine Konvertierung ohne Adapter-Kapselung.
- **AX_TO_UDINT**, **AX_TO_DINT**: Analog aufgebaute Composite-Bausteine für andere Zieldatentypen (UDINT, DINT).
- **Direkte Typumwandlung**: Im IEC-61131-Umfeld existieren zahlreiche Standard-Konvertierungsbausteine; der vorliegende spezialisiert sich auf die Adapter-Schnittstelle.

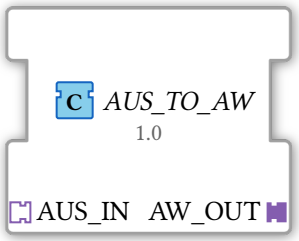
Fazit

Der Baustein **AX_TO_AULI** ist eine kompakte, typsichere und leicht wiederverwendbare Lösung zur Umwandlung eines BOOL-Adapters in einen ULINT-Adapter. Er vereinfacht die hierarchische Strukturierung von Applikationen und ermöglicht eine klare Trennung zwischen Datenquellen und -senken. Dank der Lizenzierung unter EPL-2.0 kann er frei in eigene Projekte integriert werden.

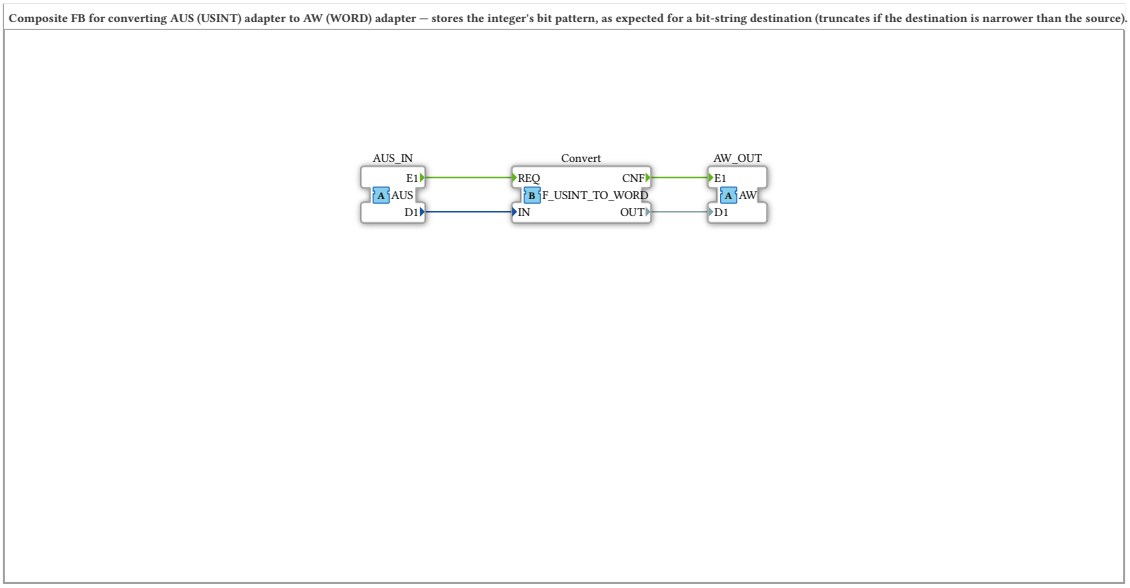
AUS_AW

AUS_TO_AW

USINT Adapter Input – *AUS*



AW – WORD Adapter Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUS_TO_AW** ist ein Composite-Baustein, der eine einfache und standardisierte Konvertierung eines **USINT**-Werts (0–255) in einen **WORD**-Wert (16 Bit) über Adapter-Schnittstellen ermöglicht. Er dient als Vermittler zwischen Systemen, die unterschiedliche Datentypen über unidirektionale Adapter bereitstellen oder erwarten.

Schnittstellenstruktur

Der Baustein selbst besitzt keine direkten Ein- oder Ausgänge. Die gesamte Kommunikation erfolgt über die beiden integrierten Adapter.

Ereignis-Eingänge

Keine direkten Ereigniseingänge vorhanden. Die Ereignissteuerung wird über den **AUS_IN**-Adapter realisiert. Dieser stellt einen Ereigniseingang **E1** bereit, dessen Aktivierung den Konvertierungsvorgang startet.

Ereignis-Ausgänge

Keine direkten Ereignisausgänge vorhanden. Ein Ereignis wird nach erfolgreicher Konvertierung über den **AW_OUT**-Adapter ausgegeben (Ereignisausgang **E1**).

Daten-Eingänge

Keine direkten Dateneingänge vorhanden. Der zu konvertierende USINT-Wert wird über den Dateneingang **D1** des **AUS_IN**-Adapters eingelesen (Typ: USINT).

Daten-Ausgänge

Keine direkten Datenausgänge vorhanden. Der konvertierte WORD-Wert wird über den Datenausgang **D1** des **AW_OUT**-Adapters ausgegeben (Typ: WORD).

Adapter

- **AUS_IN** (Socket): Unidirektionaler Adapter vom Typ `adapter::types::unidirectional::AUS`. Erwartet einen USINT-Wert und ein Ereignis zur Auslösung der Konvertierung.
- **AW_OUT** (Plug): Unidirektionaler Adapter vom Typ `adapter::types::unidirectional::AW`. Stellt nach erfolgter Konvertierung den WORD-Wert und ein Bestätigungsereignis bereit.

Funktionsweise

1. Ein externes Ereignis am **E1**-Eingang des **AUS_IN**-Adapters triggert die Ausführung.
2. Der interne Funktionsbaustein **F_USINT_TO_WORD** (aus der IEC-61131-Bibliothek) übernimmt den an **AUS_IN.D1** anliegenden USINT-Wert und konvertiert ihn in einen WORD-Wert.
3. Nach Abschluss der Konvertierung wird der WORD-Wert am **OUT**-Ausgang von **F_USINT_TO_WORD** an **AW_OUT.D1** weitergeleitet.
4. Gleichzeitig wird ein Ereignis am **CNF**-Ausgang von **F_USINT_TO_WORD** erzeugt, das über **AW_OUT.E1** nach außen signalisiert, dass der konvertierte Wert bereitsteht.

Technische Besonderheiten

- **Speichert das Bitmuster des Ganzzahlwerts**, wie für ein Bit-String-Ziel erwartet (schneidet ab, falls das Ziel schmaler ist als die Quelle).
- **Composite-Baustein**: Die Konvertierungslogik ist in einem eigenen Netzwerk gekapselt, was Wiederverwendung und Übersichtlichkeit fördert.
- **Unidirektionale Adapter**: Die Adapter **AUS** und **AW** sind als reine Daten- und Ereignispipelines ausgelegt – eine Rückwärtskommunikation ist nicht vorgesehen.
- **Standardkonvertierung**: Die eigentliche Typumwandlung basiert auf dem geprüften Baustein `F_USINT_TO_WORD` aus der IEC-61131-Bibliothek.
- **Keine eigene Zustandslogik**: Der Baustein delegiert die gesamte Steuerung an den internen Konverter und die Adapter-Schnittstellen.

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Der Ablauf ist ereignisgesteuert und erfolgt streng sequenziell:

- **Idle**: Warten auf ein Ereignis am AUS_IN-Adapter.
- **Konvertierung aktiv**: Interner **F_USINT_TO_WORD** wird ausgeführt. Nach Abschluss wechselt der Baustein automatisch zurück in den Idle-Zustand.
- **Ausgabe**: Das Bestätigungsereignis wird am AW_OUT-Adapter gesendet.

Anwendungsszenarien

- **Systemintegration**: Ein Sensor liefert Messwerte als USINT über einen unidirektionalen AUS-Adapter, die Steuerung erwartet jedoch WORD-Werte über einen AW-Adapter.
- **Datenaufbereitung**: In einer Adapter-basierten Kommunikationskette soll eine einfache Typanpassung ohne zusätzliche Logik erfolgen.

- **Prototypenentwicklung**: Schnelle Kopplung zweier Komponenten, die unterschiedliche Datentypen verwenden, ohne die bestehende Adapterstruktur zu verändern.

Vergleich mit ähnlichen Bausteinen

- **F_USINT_TO_WORD** (direkt): Bietet die reine Konvertierung ohne Adapter-Anbindung – geeignet für interne Flüsse innerhalb eines Geräts.
- **AUS_TO_AW**: Erweitert die Konvertierung um unidirektionale Adapter, wodurch eine lose Kopplung zwischen unabhängigen Systemen erreicht wird.
- **Andere Adapter-Konverter (z. B. INT_TO_DWORD)**: Folgen dem gleichen Muster, aber mit anderen Zieltypen. **AUS_TO_AW** ist speziell für die Paarung USINT ⟶ WORD optimiert.

Fazit

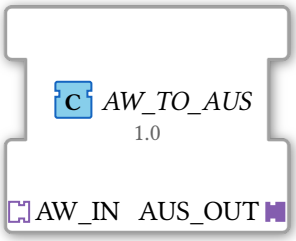
AUS_TO_AW ist ein kompakter, zuverlässiger Composite-Baustein, der die Konvertierung von USINT- zu WORD-Daten über unidirektionale Adapter standardisiert und in bestehende 4diac-Projekte nahtlos integriert werden kann. Er reduziert Entwicklungsaufwand und erhöht die Wiederverwendbarkeit in modularen Automatisierungssystemen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

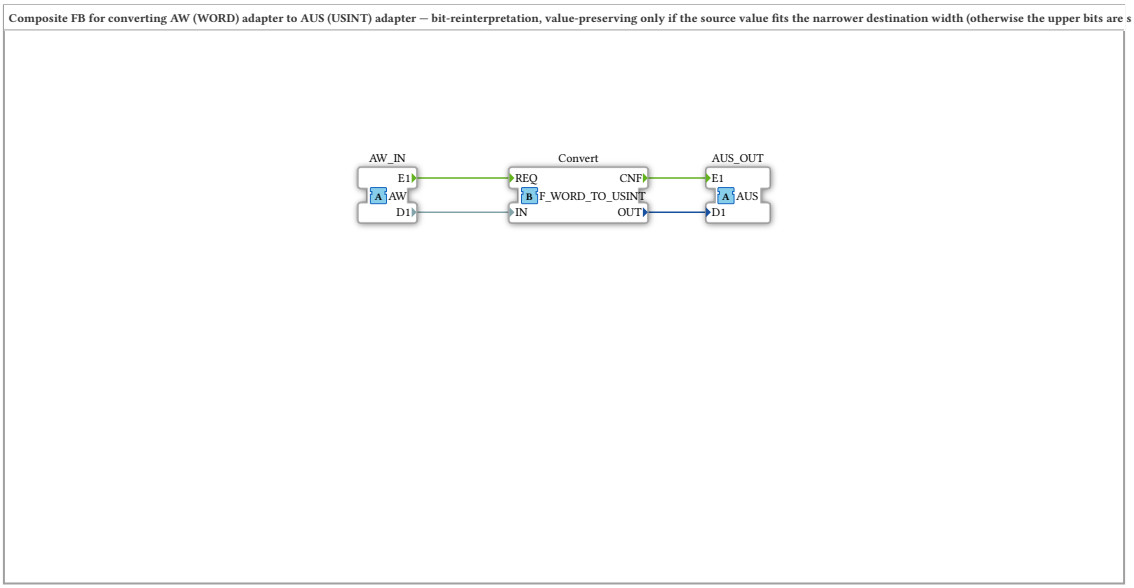
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AW_TO_AUS

WORD Adapter Input – *AW*



AUS – USINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Composite Funktionsblock `AW_TO_AUS` dient der Konvertierung eines `WORD`-Adapter-Signals in ein `USINT`-Adapter-Signal. Er kapselt die IEC 61131-3 Standardfunktion `F_WORD_TO_USINT` in einer Adapter-gekoppelten Umgebung und ermöglicht so eine nahtlose Integration zwischen unterschiedlichen Adaptertypen.

Schnittstellenstruktur

Der FB besitzt keine eigenständigen Ereignis- oder Datenports auf der Ebene der Funktionsblock-Schnittstelle. Die gesamte Kommunikation erfolgt ausschließlich über die beiden Adapter.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	`AW_IN`	`adapter::types::unidirectional::AW`	Erwartet einen `WORD`-Wert über den Datenport `D1` und ein Ereignis über `E1`.
Plug (Ausgang)	`AUS_OUT`	`adapter::types::unidirectional::AUS`	Liefert den konvertierten `USINT`-Wert über den Datenport `D1` und ein Bestätigungsereignis über `E1`.

Funktionsweise

1. Ein eingehendes Ereignis am Adapter-Socket `AW_IN.E1` triggert den internen Konvertierungsbaustein `Convert` (Typ `F_WORD_TO_USINT`) über dessen Ereigniseingang `REQ`.
2. Gleichzeitig wird der Datenwert von `AW_IN.D1` (vom Typ `WORD`) an den Dateneingang `Convert.IN` weitergeleitet.
3. Nach der Konvertierung wird das Ergebnis (`USINT`) am Datenausgang `Convert.OUT` bereitgestellt und an den Datenport `AUS_OUT.D1` übergeben.
4. Der Konvertierungsbaustein signalisiert die Fertigstellung über sein Ereignis `CNF`, welches direkt den Ereignisausgang `AUS_OUT.E1` aktiviert.

Dadurch entsteht eine synchrone, ereignisgesteuerte Umwandlung von einem `WORD`- zu einem `USINT`-Wert.

Technische Besonderheiten

- **Bit-Reinterpretation**, werterhaltend nur, wenn der Quellwert in die schmalere Zielbreite passt (sonst werden die oberen Bits stillschweigend verworfen).
- **Composite Pattern**: Der Baustein ist als Composite FB realisiert, der die vorhandene Standardfunktion `F_WORD_TO_USINT` kapselt.
- **Adapter-Kopplung**: Der FB vereinfacht die Integration in bestehende Systeme, die auf die Adapter `AW` und `AUS` angewiesen sind, indem er die Konvertierung in einem einzigen Block zusammenfasst.
- **Keine eigene Zustandslogik**: Die gesamte Logik wird delegiert; der Baustein selbst hat keine eigene Ablaufsteuerung (ECC) und verhält sich rein direktional.

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Der Ablauf wird vollständig durch die eingebettete Funktion `F_WORD_TO_USINT` und die Ereignisverbindungen bestimmt. Der FB ist daher als **stateless** zu betrachten.

Anwendungsszenarien

- **Adapter-Konvertierung**: Wenn in einer 4diac-Applikation ein Adapter vom Typ `AW` (mit `WORD`-Daten) vorliegt, aber ein nachfolgender Baustein einen Adapter vom Typ `AUS` (mit `USINT`-Daten) erwartet.

- **Protokollanpassung**: Vereinfacht die Anbindung von Komponenten, die unterschiedliche Datentypen über ihre Adapter-Schnittstellen verwenden.
- **Minimaler Konvertierungsaufwand**: Ideal, wenn keine zusätzliche Logik erforderlich ist und lediglich eine einfache Typumwandlung (z.B. aus einem 16-Bit-Wert in einen 8-Bit-Wert) durchgeführt werden muss.

Vergleich mit ähnlichen Bausteinen

- **Direkte Verwendung von `F_WORD_TO_USINT`**: Dieser Standardbaustein arbeitet auf Datenebene, benötigt jedoch manuelle Ereignis- und Datenverbindungen. `AW_TO_AUS` kapselt diese Verbindungen und bietet eine adapterbasierte Schnittstelle.
- **Andere Adapter-Konverter (z.B. `AW_TO_ABOOL`)**: Unterscheiden sich durch den Ziel-Datentyp. Das vorliegende Schema (Socket -> Konvertierungs-FB -> Plug) kann für beliebige Datentypen analog aufgebaut werden.

Fazit

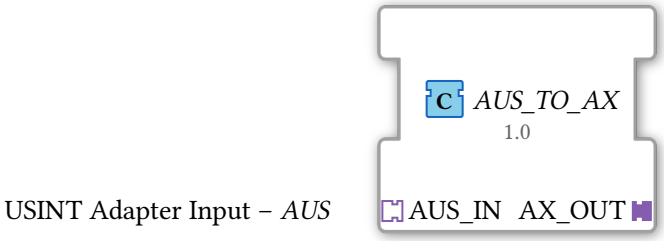
`AW_TO_AUS` ist ein einfacher, aber nützlicher Composite FB zur Konvertierung von `WORD`-Adapter-Signalen in `USINT`-Adapter-Signale. Er nutzt eine bewährte IEC 61131-3 Standardfunktion und bietet eine saubere, adapterbasierte Schnittstelle. Der Baustein erfordert keine eigene Konfiguration und fügt sich nahtlos in ereignisgesteuerte Abläufe ein. Er eignet sich besonders für Anwendungen, bei denen eine schnelle und zuverlässige Datentypanpassung ohne zusätzliche Logik gewünscht ist.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

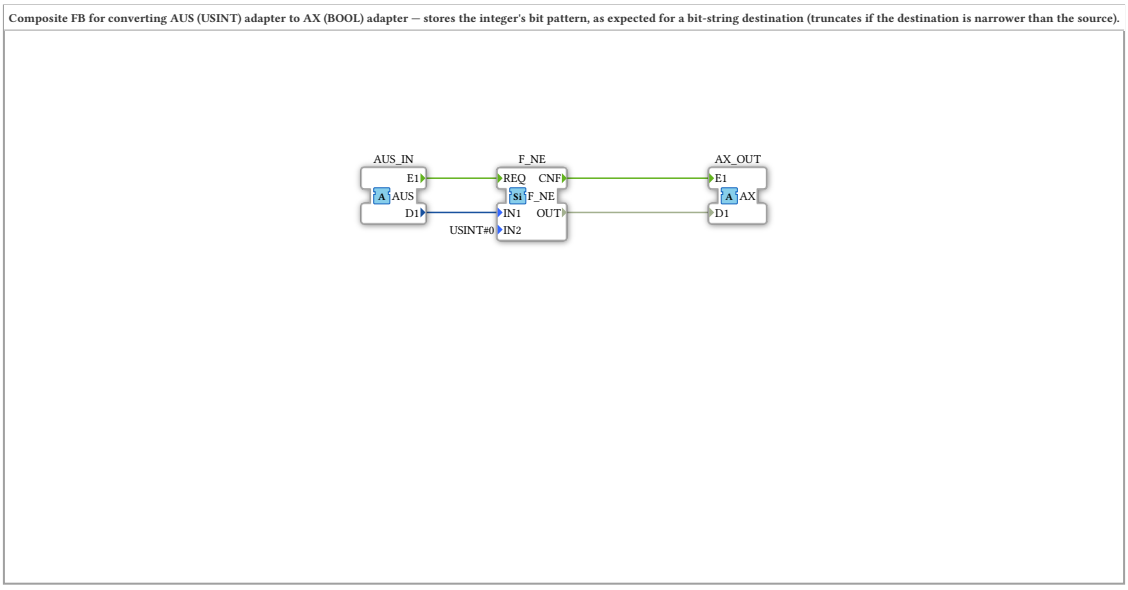
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUS_AX

AUS_TO_AX



USINT Adapter Input – *AUS* — *AX* – BOOL Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **AUS_TO_AX** ist ein Composite-FB zur Konvertierung eines **AUS**-Adapters (Daten vom Typ USINT) in einen **AX**-Adapter (Daten vom Typ BOOL). Er vergleicht den eingehenden USINT-Wert mit Null und gibt das Ergebnis als Boolesches Signal aus. Dies ermöglicht eine einfache Umsetzung von Zähler- oder Sensordaten in ein binäres Statusignal.

Schnittstellenstruktur

Der FB bietet keine direkten Ereignis- oder Datenanschlüsse, sondern ausschließlich Adapter-Schnittstellen.

Ereignis-Eingänge

Keine direkten Ereigniseingänge. Das Ereignissignal wird über den Adapter **AUS_IN.E1** bereitgestellt.

Ereignis-Ausgänge

Keine direkten Ereignisausgänge. Das Ereignissignal wird über den Adapter **AX_OUT.E1** ausgegeben.

Daten-Eingänge

Keine direkten Dateneingänge. Das Datensignal (USINT) wird über den Adapter **AUS_IN.D1** bereitgestellt.

Daten-Ausgänge

Keine direkten Datenausgänge. Das Datensignal (BOOL) wird über den Adapter **AX_OUT.D1** ausgegeben.

Adapter

Name	Richtung	Typ	Kommentar
AUS_IN	Socket	adapter::types::unidirectional::AUS	USINT-Eingangswert und Ereignis
AX_OUT	Plug	adapter::types::unidirectional::AX	BOOL-Ausgangswert und Ereignis

Funktionsweise

Der FB nutzt intern einen **F_NE**-Baustein (Vergleich auf Ungleichheit) aus der IEC-61131-Bibliothek.

- Ein ankommendes Ereignis **AUS_IN.E1** triggert den **REQ**-Eingang von **F_NE**.
- **F_NE** vergleicht den Datenwert **AUS_IN.D1** (USINT) mit dem konstanten Wert **USINT#0**.
- Das Vergleichsergebnis (BOOL: TRUE, wenn ungleich Null, sonst FALSE) wird an **AX_OUT.D1** weitergeleitet.
- Nach Abschluss der Verarbeitung erzeugt **F_NE** ein **CNF**-Ereignis, das auf **AX_OUT.E1** übertragen wird.

Technische Besonderheiten

- **Kein Bitmuster, sondern Ungleich-Null-Vergleich**: Das Ergebnis ist `TRUE` für jeden Wert ungleich 0, nicht das niederwertigste Bit. Der Kommentar im Quellbaustein („speichert das Bitmuster …, schneidet ab“) beschreibt ein anderes Verhalten und ist veraltet; maßgeblich ist das Netzwerk mit `F_NE`.
- Der FB ist als **Composite-FB** realisiert, d.h. seine Logik wird durch ein internes Netzwerk aus Standardbausteinen abgebildet.
- Er verwendet den **F_NE**-Vergleichsbaustein, der der Bibliothek **iec61131::comparison** entstammt.
- Die Paketorganisation lautet **adapter::conversion::unidirectional**, was die Verwendung als Adapter-Konverter kennzeichnet.
- Der FB besitzt keine eigenen Zustandsautomaten; die Verarbeitung erfolgt ausschließlich ereignisgesteuert über die internen Verbindungen.

Zustandsübersicht

Als Composite-FB besitzt **AUS_TO_AX** keinen eigenen Zustandsautomaten. Die gesamte Logik wird vom integrierten **F_NE** durchgeführt, das keine zustandsbehaftete Steuerung benötigt. Das Verhalten ist rein kombinatorisch mit ereignisgesteuerter Ausführung.

Anwendungsszenarien

- Umwandlung eines Zählerwerts (USINT) in ein Bool-Signal, das den Zustand „Wert ≠ 0“ anzeigt (z.B. „Füllstand vorhanden“ oder „Sensor aktiv“).
- Einsatz in der **Agrartechnik** – der FB stammt aus dem Umfeld der HR Agrartechnik GmbH – zur Verarbeitung von Sensordaten über Adapter-Schnittstellen.
- Vereinfachung der Verkabelung in 4diac-IDE-Projekten, wenn heterogene Adapter-Typen aufeinander abgestimmt werden müssen.

Vergleich mit ähnlichen Bausteinen

- Direkter Vergleich mit einem **NE**-Baustein: **AUS_TO_AX** erweitert die reine Vergleichsfunktion um eine Adapter-Konvertierung.
- Während ein **NE**-FB üblicherweise zwei einfache Dateneingänge (z.B. ANY) erwartet, arbeitet dieser Baustein ausschließlich mit vordefinierten Adaptern (AUS / AX) und versteckt die interne Logik.
- Er ist damit spezifisch für Systeme mit einheitlichem Adapter-Konzept (unidirektionale Schnittstellen) optimiert.

Fazit

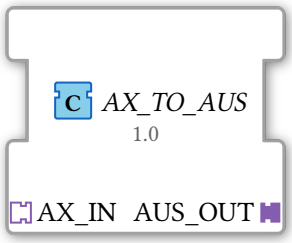
AUS_TO_AX ist ein einfacher, aber zweckmäßiger Adapter-Konverter, der einen USINT-Wert über den Vergleich mit Null in ein Boolesches Signal umsetzt. Durch die Kapselung als Composite-FB mit Adapteranschlüssen lässt er sich sauber in modulare 4diac-Projekte einbinden. Der FB bietet keine eigenen Zustände und ist aufgrund seiner geringen Komplexität zuverlässig und wartungsarm.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

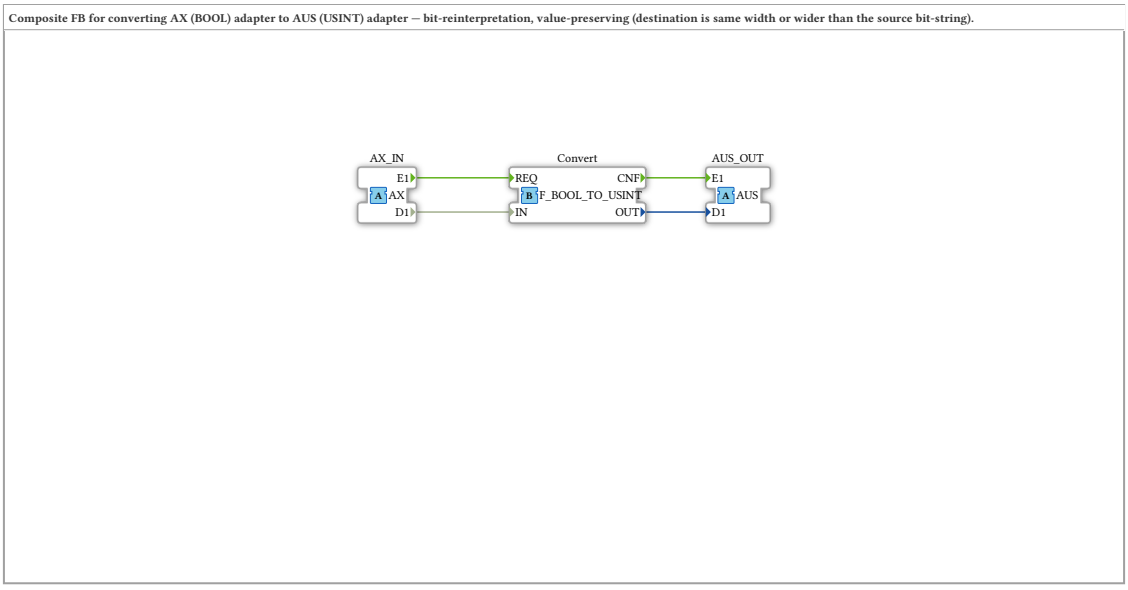
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AX_TO_AUS

BOOL Adapter Input – *AX*



AUS – USINT Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AX_TO_AUS** ist ein Composite-Baustein, der einen booleschen Eingangsadapter (Typ `AX`) in einen ganzzahligen Ausgangsadapter (Typ `AUS`) konvertiert. Er dient als Brücke zwischen Komponenten, die mit unterschiedlichen Datentypen arbeiten, und ermöglicht eine nahtlose Integration in Steuerungsanwendungen nach IEC 61499.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine direkten Ereignis-Eingänge, jedoch wird über den eingehenden Adapter `AX_IN` das Ereignis `E1` bereitgestellt. Dieses löst die Konvertierung aus.

Ereignis-Ausgänge

Auch hier gibt es keine direkten Ereignis-Ausgänge. Der ausgehende Adapter `AUS_OUT` stellt das Ereignis `E1` zur Verfügung, das nach erfolgreicher Konvertierung getriggert wird.

Daten-Eingänge

Die Daten werden über den Adapter `AX_IN` als boolescher Wert (`D1` vom Typ `BOOL`) entgegengenommen.

Daten-Ausgänge

Der konvertierte Wert wird über den Adapter `AUS_OUT` als vorzeichenloser 8-Bit-Ganzzahl (`D1` vom Typ `USINT`) ausgegeben.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	`AX_IN`	`adapter::types::unidirectional::AX`	Boolescher Eingangsadapter
Plug	`AUS_OUT`	`adapter::types::unidirectional::AUS`	USINT-Ausgangsadapter

Funktionsweise

Intern verwendet der Baustein den vordefinierten Konvertierungsbaustein `iec61131::conversion::F_BOOL_TO_USINT`. Der Ablauf ist wie folgt:

1. Ein eingehendes Ereignis am Adapter `AX_IN.E1` triggert den internen Baustein über dessen Eingang `REQ`.
2. Der boolesche Wert von `AX_IN.D1` wird an den Eingang `IN` des internen Bausteins weitergeleitet.
3. Der interne Baustein konvertiert den `BOOL`-Wert in einen `USINT`-Wert (`false` ⟶ 0, `true` ⟶ 1) und legt ihn an seinem Ausgang `OUT` an.
4. Das Bestätigungsereignis `CNF` des internen Bausteins wird an den Adapterausgang `AUS_OUT.E1` weitergeleitet.
5. Der konvertierte Wert vom Ausgang `OUT` wird an den Datenausgang `AUS_OUT.D1` übergeben.

Technische Besonderheiten

- **Bit-Reinterpretation, werterhaltend** (Zielbreite ist gleich groß oder größer als der Quell-Bit-String).
- **Composite-Baustein**: Der FB kapselt die Konvertierungslogik und stellt eine saubere, wiederverwendbare Schnittstelle über Adapter bereit.
- **Plug/Socket-Kopplung**: Der Baustein verwendet das Adapter-Pattern (Socket für Eingang, Plug für Ausgang), was eine flexible Verbindung in einer IEC-61499-Applikation ermöglicht.
- **Abhängigkeit**: Er importiert den Baustein `iec61131::conversion::F_BOOL_TO_USINT`, der Teil der IEC-61131-Konvertierungsbibliothek ist.

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Die gesamte Funktionalität wird durch den internen `F_BOOL_TO_USINT`-Baustein realisiert, der ereignisgesteuert arbeitet und keine internen Zustände speichert.

Anwendungsszenarien

- **Signalanpassung**: Wenn ein Sensor oder ein boolescher Adapter (z. B. Schalter, Grenztaster) an eine Komponente angeschlossen werden soll, die einen ganzzahligen Adapter erwartet.
- **Protokollumsetzung**: In heterogenen Systemen, in denen unterschiedliche Datenformate verwendet werden, kann dieser Baustein als einfacher Konverter dienen.
- **Test- und Simulationsumgebungen**: Um boolesche Werte in numerische Werte umzuwandeln, die von Analysewerkzeugen leichter verarbeitet werden können.

Vergleich mit ähnlichen Bausteinen

Es existieren zahlreiche Konvertierungsbausteine in der IEC-61499-Bibliothek, z. B. `F_BOOL_TO_INT`, `F_BOOL_TO_WORD` oder `F_BOOL_TO_BYTE`. Der Unterschied zu `AX_TO_AUS` liegt in der Adapter-Schnittstelle: Dieser Baustein kapselt die Konvertierung nicht nur auf Datenebene, sondern auch auf Ereignisebene in einem kompakten Adapter-Paar. Andere Bausteine arbeiten meist mit einfachen Ein-/Ausgangsdaten und erfordern zusätzliche Verdrahtung von Ereignissen.

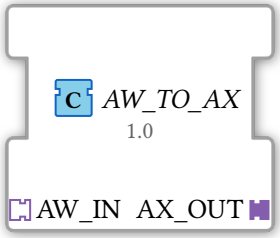
Fazit

Der Funktionsblock **AX_TO_AUS** ist ein praktischer Helfer, um boolesche Adapter-Signale in das USINT-Format zu wandeln. Durch den Einsatz von Adaptern fügt er sich nahtlos in die ereignisgesteuerte Architektur nach IEC 61499 ein und reduziert den Verdrahtungsaufwand erheblich. Er eignet sich besonders für Anwendungen, bei denen eine saubere Trennung zwischen boolescher und numerischer Domäne gewünscht ist.

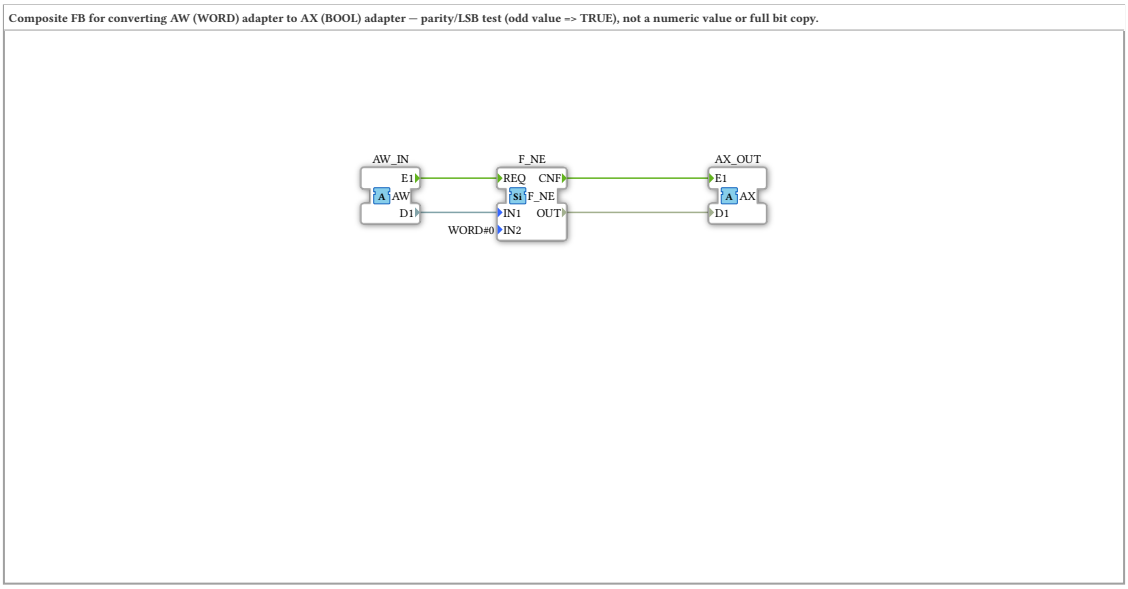
AW_AX

AW_TO_AX

WORD Adapter Input – *AW*



AX – BOOL Adapter Output



• – ○ • – ○ • – ○ •

Einleitung

Der Composite-Funktionsblock **AW_TO_AX** dient der Umwandlung eines **AW-Adapters** (WORD) in einen **AX-Adapter** (BOOL). Er prüft, ob der über den AW-Adapter eingehende 16-Bit-Wert ungleich Null ist und gibt das Ergebnis als boolesches Signal über den AX-Adapter aus.

Schnittstellenstruktur

Der Baustein besitzt keine klassischen Ereignis- oder Datenein-/ausgänge, sondern ausschließlich Adapter-Schnittstellen.

Ereignis-Eingänge

Keine separaten Ereignis-Eingänge. Ereignisse werden über den Socket-Adapter empfangen.

Ereignis-Ausgänge

Keine separaten Ereignis-Ausgänge. Ereignisse werden über den Plug-Adapter gesendet.

Daten-Eingänge

Keine separaten Daten-Eingänge. Daten werden über den Socket-Adapter empfangen.

Daten-Ausgänge

Keine separaten Daten-Ausgänge. Daten werden über den Plug-Adapter gesendet.

Adapter

Name	Typ	Richtung	Kommentar
AW_IN	adapter::types::unidirectional::AW	Socket	WORD-Adapter-Eingang
AX_OUT	adapter::types::unidirectional::AX	Plug	BOOL-Adapter-Ausgang

Der Socket **AW_IN** stellt die Ereignis- und Datenschnittstelle eines **WORD**-wertigen Adapters zur Verfügung. Der Plug **AX_OUT** stellt die Ereignis- und Datenschnittstelle eines **BOOL**-wertigen Adapters bereit.

Funktionsweise

Bei Eintreffen eines Ereignisses am **AW_IN.E1** wird der Wert von **AW_IN.D1** mit dem konstanten Wert **WORD#0** verglichen. Dazu wird intern der Baustein **F_NE** (ungleich-Vergleich aus IEC 61131) verwendet.

- Ist **AW_IN.D1** $\neq$ **0**, so wird ein Ereignis an **AX_OUT.E1** gesendet und der Datenwert **AX_OUT.D1** auf **TRUE** gesetzt.
- Andernfalls wird kein Ereignis am Ausgang generiert und der Datenwert bleibt auf **FALSE** (bzw. wird nicht geändert).

Das Verhalten ist ereignisgesteuert und folgt der 61499-Ausführungssemantik: Ein Ereignis am Eingang löst den Vergleich aus, dessen Ergebnis am Ausgang weitergegeben wird.

Technische Besonderheiten

- **Paritäts-/LSB-Test** (ungerader Wert $\Rightarrow$ TRUE), kein Zahlenwert und keine vollständige Bit-Kopie.
- Der Baustein ist als **Composite-Funktionsblock** realisiert, d.h. seine Funktionalität wird durch ein inneres Netzwerk aus einem einzelnen **F_NE**-Baustein abgebildet.
- Er verwendet einen **unidirektionalen Adapter** (AW und AX) gemäß der Adapterdefinition der 4diac-IDE.
- Die Vergleichslogik stammt aus der IEC 61131-Bibliothek (`iec61131::comparison::F_NE`).
- Es handelt sich um eine reine Signalumsetzung ohne Zustandsbehaftung.

Zustandsübersicht

Der Baustein besitzt **keinen internen Zustandsautomaten** (kein ECC). Die Ablaufsteuerung ergibt sich direkt aus dem inneren Netzwerk: Ein Ereignis am Eingangsadapter triggert den Vergleichsbaustein, der wiederum das Ergebnis an den Ausgangsadapter weitergibt.

Anwendungsszenarien

- **Signalaufbereitung:** Wandlung eines digitalen WORD-Werts (z.B. aus einem Sensor-Modul) in ein boolesches Signal, das anzeigt, ob der Wert von Null verschieden ist.
- **Adapter-Integration:** Einsatz in Systemen, die eine Adapter-basierte Kommunikation zwischen unterschiedlichen Datentypen erfordern (z.B. Verknüpfung einer WORD-basierten Buskomponente mit einer BOOL-basierten Steuerlogik).
- **Schwellwertprüfung:** Durch Ändern des Vergleichswerts (im internen Baustein **F_NE**) könnte der Funktionsblock auch auf andere Schwellwerte angepasst werden (z.B. „Wert > 100“).

Vergleich mit ähnlichen Bausteinen

- **WORD_TO_BOOL:** Einfache direkte Wandlung eines WORD-Datentyps in BOOL (z.B. Bit 0). Der **AW_TO_AX** arbeitet dagegen mit Adaptern und führt einen Vergleich mit Null durch – er reagiert auf das gesamte Word.
- **Ungleich-Vergleicher (F_NE):** Der **AW_TO_AX** kapselt den **F_NE**-Baustein und fügt die Adapter-Schnittstellen hinzu, sodass er in Adapter-basierten Architekturen verwendet werden kann.

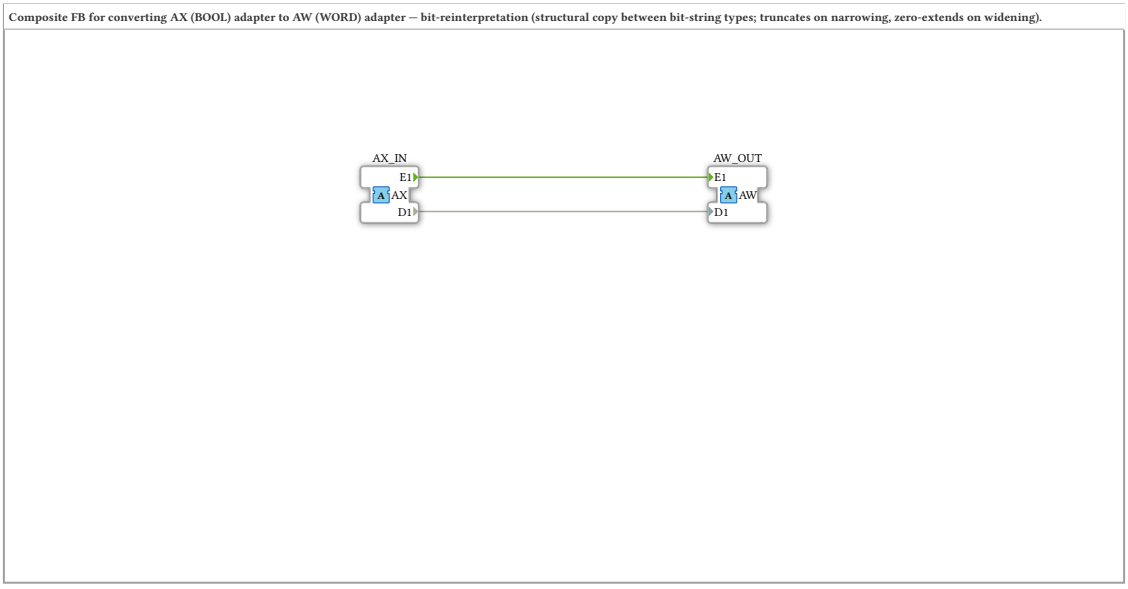
Fazit

Der **AW_TO_AX** ist ein kompakter, aber nützlicher Composite-Baustein zur Konvertierung von WORD-Adaptern in BOOL-Adapter. Dank seines einfachen Vergleichsprinzips eignet er sich hervorragend für die Integration in ereignisgesteuerte Automatisierungssysteme, die auf Adapter-Schnittstellen basieren. Die klare Trennung von Eingangs- und Ausgangsadaptern erleichtert den modularen Aufbau und die Wiederverwendbarkeit.

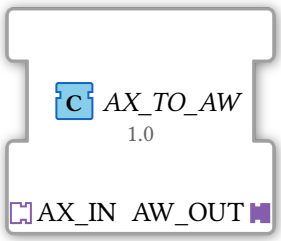
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AX_TO_AW



Kein Bild verfügbar.



BOOL Adapter Input – *AX* *AW* – WORD Adapter Output

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AX_TO_AW ist ein zusammengesetzter Baustein (Composite FB) zur Konvertierung eines unidirektionalen BOOL-Adapters (Typ AX) in einen unidirektionalen WORD-Adapter (Typ AW). Er dient als Schnittstellenwandler, der ein ankommendes Ereignis und einen BOOL-Datenwert direkt an den Ausgangsadapter weiterleitet, wobei der BOOL-Wert implizit in ein WORD umgewandelt wird.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine separaten Ereignis-Eingänge. Das Ereignis wird über den Socket-Adapter AX_IN (Typ AX) bereitgestellt (internes Ereignis E1).

Ereignis-Ausgänge

Der Baustein besitzt keine separaten Ereignis-Ausgänge. Das Ereignis wird über den Plug-Adapter AW_OUT (Typ AW) ausgegeben (internes Ereignis E1).

Daten-Eingänge

Der Baustein besitzt keine separaten Daten-Eingänge. Die Eingangsdaten (BOOL) werden über den Socket-Adapter AX_IN (Datenpunkt D1, Typ BOOL) bereitgestellt.

Daten-Ausgänge

Der Baustein besitzt keine separaten Daten-Ausgänge. Die Ausgangsdaten (WORD) werden über den Plug-Adapter AW_OUT (Datenpunkt D1, Typ WORD) ausgegeben.

Adapter

Richtung	Bezeichnung	Typ	Kommentar
Socket	AX_IN	adapter::types::unidirectional::AX	BOOL-Adapter-Eingang (bestehend aus Ereignis E1 und Daten D1:BOOL)
Plug	AW_OUT	adapter::types::unidirectional::AW	WORD-Adapter-Ausgang (bestehend aus Ereignis E1 und Daten D1:WORD)

Funktionsweise

Der Baustein leitet das vom Socket AX_IN empfangene Ereignis (E1) direkt an den Plug AW_OUT (E1) weiter. Gleichzeitig wird der eingehende BOOL-Wert (AX_IN.D1) ohne weitere Verarbeitung an den WORD-Ausgang (AW_OUT.D1) übergeben. Dabei findet eine implizite Typkonvertierung von BOOL nach WORD statt (übliche Abbildung: FALSE → 0, TRUE → 1). Es findet keine Logik, Verzögerung oder Zustandsänderung statt – der FB fungiert als reine Durchreiche.

Technische Besonderheiten

- **Bit-Reinterpretation** (strukturelle Kopie zwischen Bit-String-Typen; schneidet bei Verengung ab, erweitert mit Nullen bei Erweiterung).
- **Composite FB:** Der Baustein besitzt keinen eigenen Zustandsautomaten (ECC), sondern besteht ausschließlich aus einer Verdrahtung der enthaltenen Adapter-Schnittstellen.
- **Typkonvertierung:** Die Konvertierung von BOOL nach WORD erfolgt automatisch durch die Datenverbindung zwischen den Adaptern. Es ist keine explizite Konvertierungslogik implementiert.
- **Performance:** Durch die direkte Kopplung werden keine zusätzlichen Verarbeitungsschritte eingeführt, was eine minimale Latenz gewährleistet.
- **Wiederverwendbarkeit:** Der Baustein kann in Projekten eingesetzt werden, die Adapter unterschiedlicher Typen verbinden müssen, ohne dass eine separate Konvertierungslogik erforderlich ist.

Zustandsübersicht

Da es sich um einen Composite FB ohne eigenen ECC handelt, existieren keine expliziten Zustände. Der Baustein verhält sich wie eine reine Signalweiterleitung und besitzt kein eigenes Verhaltensmodell.

Anwendungsszenarien

- **Anbindung eines BOOL-basierten Sensors** an ein Steuerungssystem, das WORD-Schnittstellen erwartet (z. B. digitale Eingänge als Bitmasken).
- **Adapterschicht** in Bibliotheken, in denen Komponenten mit unterschiedlichen Adapter-Typen zusammengeführt werden müssen, ohne die Quellbausteine zu verändern.
- **Prototypische Entwicklung** zur schnellen Integration von BOOL- und WORD-Adaptern vor einer späteren Optimierung.

Vergleich mit ähnlichen Bausteinen

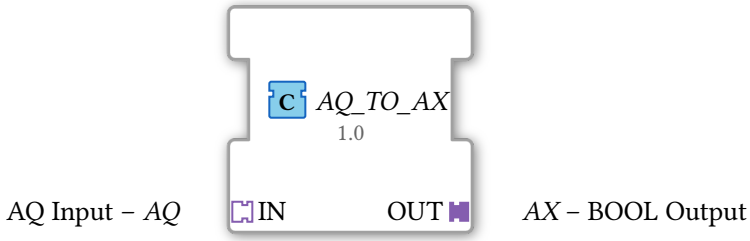
- **AW_TO_AX (theoretisch):** Wäre die inverse Konvertierung von WORD zu BOOL – würde eine Reduktion des Wertebereichs erfordern (z. B. nur Bit 0 wird ausgewertet).
- **TYPCONV-Bausteine:** In der Regel bieten IEC-61499-Bibliotheken explizite Typkonverter (z. B. `BOOL_TO_WORD`), die jedoch als normale Funktionsblöcke mit Ereignis- und Dateneingängen arbeiten und nicht als Adapter-Konverter.
- **Direkte Adapter-Kopplung:** Ohne `AX_TO_AW` müssten die Adapter-Typen identisch sein oder eine separate Konvertierung in der Netzwerkebene erfolgen.

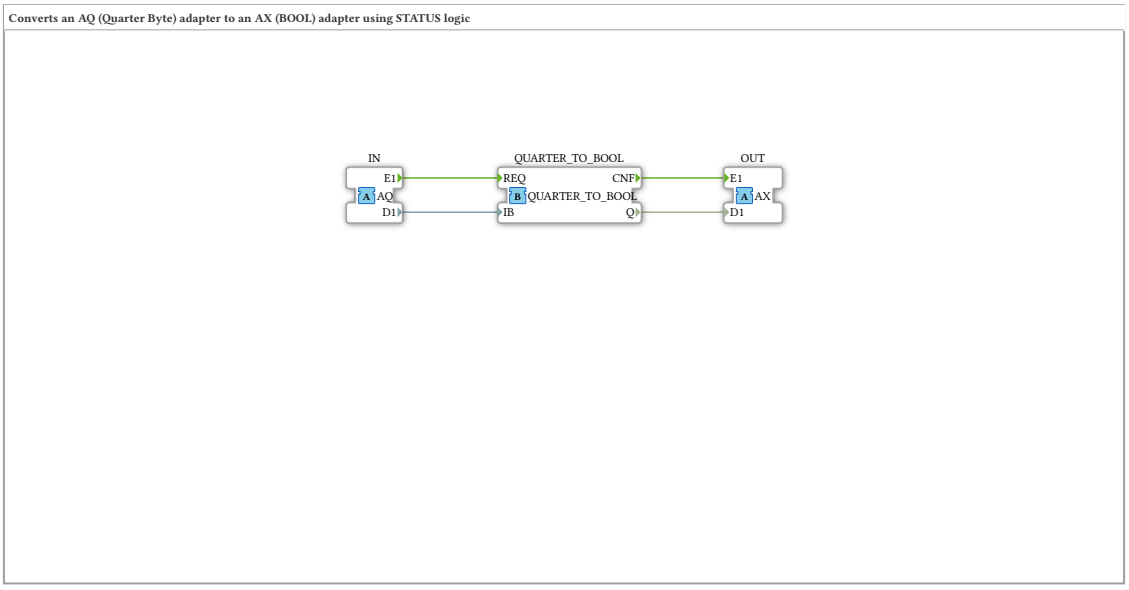
Fazit

Der Funktionsblock `AX_TO_AW` ist ein einfacher, aber nützlicher Konverter, der die Kopplung zwischen BOOL- und WORD-Adaptern ermöglicht. Dank der direkten Durchschleifung von Ereignis und Daten bei minimalem Overhead eignet er sich hervorragend für Schnittstellenanpassungen in verteilten Automatisierungssystemen. Seine Composite-Struktur macht ihn transparent, leicht verständlich und einfach in bestehende Projekte integrierbar.

AX_AQ

AQ_TO_AX





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock AQ_TO_AX dient der Konvertierung eines AQ-Adapters (Quarter Byte) in einen AX-Adapter (BOOL) unter Verwendung einer STATUS-Logik. Er ermöglicht die einfache Integration von AQ-basierten Signalen in Systeme, die ausschließlich mit AX-Schnittstellen arbeiten. Intern nutzt er den Baustein QUARTER_TO_BOOL, um die Datenkonvertierung durchzuführen.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkten Ereignis-Eingänge vorhanden. Ereignisse werden über den AQ-Adapter (Socket IN) empfangen.

Ereignis-Ausgänge

- Keine direkten Ereignis-Ausgänge vorhanden. Ereignisse werden über den AX-Adapter (Plug OUT) gesendet.

Daten-Eingänge

- Keine direkten Daten-Eingänge vorhanden. Daten werden über den AQ-Adapter (Socket IN) empfangen.

Daten-Ausgänge

- Keine direkten Daten-Ausgänge vorhanden. Daten werden über den AX-Adapter (Plug OUT) gesendet.

Adapter

- **IN (Socket)**: Typ adapter::types::unidirectional::AQ – AQ-Eingang (Quarter Byte)
- **OUT (Plug)**: Typ adapter::types::unidirectional::AX – BOOL-Ausgang (AX)

Funktionsweise

Der AQ_TO_AX Baustein besitzt keine eigenen Event- oder Daten-Ein-/Ausgänge, sondern realisiert die Konvertierung ausschließlich über zwei Adapter-Schnittstellen. Intern wird eine Instanz des Bausteins QUARTER_TO_BOOL eingesetzt:

1. **Ereignisweiterleitung**: Das Ereignis E1 des eingehenden AQ-Adapters wird mit dem REQ-Eingang von QUARTER_TO_BOOL verbunden. Dessen CNF-Ereignis wird auf dem E1-Ausgang des AX-Adapters ausgegeben.
2. **Datenweiterleitung**: Das Datenwort D1 des AQ-Adapters (Quarter Byte) wird an den IB-Eingang von QUARTER_TO_BOOL geleitet. Der konvertierte BOOL-Wert aus dem Ausgang Q von QUARTER_TO_BOOL wird auf den D1-Ausgang des AX-Adapters gegeben.

Die gesamte Ablaufsteuerung und Datenkonvertierung erfolgt somit transparent im Inneren des Bausteins.

Technische Besonderheiten

- Der Baustein fungiert als reiner Adapter-Konverter und besitzt keine eigenen Zustände oder Anwendungslogik; die gesamte Funktionalität wird durch den eingebetteten QUARTER_TO_BOOL bereitgestellt.
- Die Verwendung von Adaptern (Socket/Plug) erlaubt eine lose Kopplung in der 4diac-IDE und erleichtert den Austausch von Kommunikationsschnittstellen.
- Der Kommentar „using STATUS logic“ deutet darauf hin, dass die Konvertierung möglicherweise auf einer Status-Abfrage basiert (z.B. gültige/nicht gültige Daten), was im internen Baustein umgesetzt wird.

Zustandsübersicht

Der Baustein definiert keine eigenen Zustände. Die Zustandslogik liegt vollständig in dem internen Funktionsblock QUARTER_TO_BOOL.

Anwendungsszenarien

- **Schnittstellenanpassung**: Einbindung eines AQ-basierten Sensors oder Aktors in ein System, das ausschließlich AX-kompatible Adapter unterstützt.
- **Adapter-Transformation**: Einsatz in Hierarchien, in denen unterschiedliche Adapter-Typen aufeinander abgestimmt werden müssen, ohne die zugrundeliegende Datenlogik zu ändern.
- **Prototypen und Migration**: Schnelles Umschalten zwischen AQ- und AX-Varianten während der Entwicklungs- oder Testphase.

Vergleich mit ähnlichen Bausteinen

- **`QUARTER_TO_BOOL`**: Direkte Konvertierung eines Quarter-Byte-Wertes in einen BOOL-Wert mit eigenem Event-Interface; `AQ_TO_AX` kapselt diesen Baustein in einem Adapter-Kontext.
- **`BYTE_TO_BOOL` oder `WORD_TO_BOOL`**: Ähnliche Konvertern für andere Bit-Breiten, jedoch nicht speziell für die Adapter-Kopplung ausgelegt.
- **Direkte Kopplung von AQ und AX**: Ohne Konvertierung wäre eine Verbindung nicht möglich, da die Adapter-Typen inkompatibel sind.

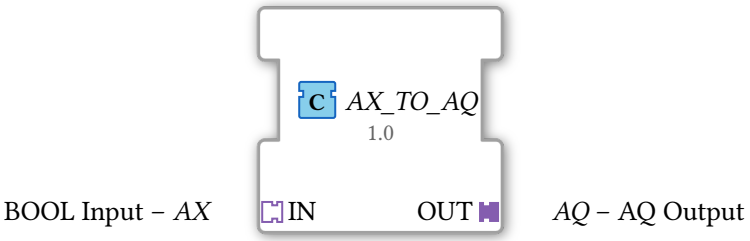
Fazit

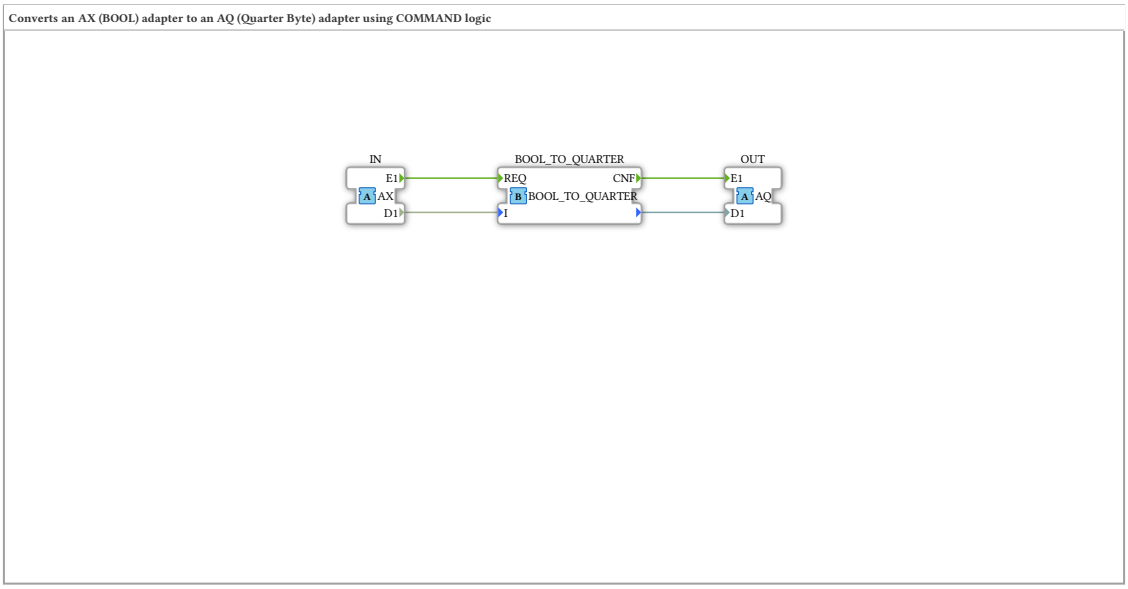
`AQ_TO_AX` ist ein schlanker, aber essentieller Adapter-Konverter, der die Lücke zwischen AQ- und AX-Schnittstellen schließt. Er vereinfacht die Integration von Quarter-Byte-Signalen in BOOL-basierte Steuerungen und trägt zur Wiederverwendbarkeit von Funktionsbausteinen in verschiedenen Adapter-Umgebungen bei.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AX_TO_AQ





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AX_TO_AQ** dient der Umwandlung eines booleschen Signals (AX-Adapter) in ein Quarter-Byte-Signal (AQ-Adapter). Die Umwandlung erfolgt auf Basis einer ereignisgesteuerten COMMAND-Logik, bei der ein eingehendes Ereignis die Konvertierung auslöst und das Ergebnis über ein Ausgangsereignis quittiert wird. Der Baustein kapselt die Konvertierungslogik in einem internen Sub-Funktionsblock und ermöglicht so eine saubere Trennung zwischen Adaptertypen.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung
`IN.E1`	Startet die Konvertierung des eingehenden Bool-Werts. Wird als Befehl (COMMAND) interpretiert.

Ereignis-Ausgänge

Ereignis	Beschreibung
`OUT.E1`	Bestätigt die abgeschlossene Konvertierung und signalisiert die Gültigkeit des Ausgangs-Quarter-Byte-Werts.

Daten-Eingänge

Daten	Typ	Beschreibung
`IN.D1`	BOOL (über AX-Adapter)	Boolescher Eingangswert, der in ein Quarter-Byte umgewandelt werden soll.

Daten-Ausgänge

Daten	Typ	Beschreibung
`OUT.D1`	Quarter Byte (über AQ-Adapter)	Ausgangswert – das umgewandelte Quarter-Byte (4-Bit-Darstellung).

Adapter

Adapter	Richtung	Typ	Beschreibung
`IN`	Socket	`adapter::types::unidirectional::AX`	Empfängt das boolesche Signal. Der AX-Adapter stellt einen unidirektionalen Bool-Kanal bereit.
`OUT`	Plug	`adapter::types::unidirectional::AQ`	Stellt das konvertierte Quarter-Byte-Signal bereit. Der AQ-Adapter ist ein unidirektionaler Quarter-Byte-Kanal.

Funktionsweise

1. Ein Ereignis an `IN.E1` triggert die interne Verarbeitung.
2. Der Sub-Funktionsblock `BOOL_TO_QUARTER` wird über seinen Ereigniseingang `REQ` aktiviert und wandelt den an `I` anliegenden Bool-Wert in ein Quarter-Byte um.
3. Nach Abschluss der Konvertierung meldet `BOOL_TO_QUARTER` dies über sein Ereignis `CNF`, das an `OUT.E1` durchgereicht wird.
4. Gleichzeitig wird der konvertierte Wert über die Datenverbindung vom Ausgang `QB` des Sub-Blocks an `OUT.D1` übergeben.

Der gesamte Prozess ist ereignisgesteuert und erfolgt ohne interne Zustandsspeicherung – jeder Befehl erzeugt eine vollständige Umwandlung.

Technische Besonderheiten

- **Interne Kapselung**: Die Konvertierungslogik wird vollständig durch den eingebetteten Funktionsblock `logiBUS::utils::quarter::BOOL_TO_QUARTER` realisiert. Dies vereinfacht Wartung und Austausch der Konvertierungsimplementierung.
- **Unidirektionalität**: Sowohl Eingangs- als auch Ausgangsadapter sind unidirektional ausgelegt; eine Rückkopplung oder bidirektionale Nutzung ist nicht vorgesehen.

- **Ereignissynchronisation**: Der Ausgangs-Ereignispuls `OUT.E1` garantiert, dass die Daten am Ausgang `OUT.D1` gültig sind, bevor der aufrufende Baustein weiterarbeitet.
- **Lizenz und Herkunft**: Der Baustein ist unter der EPL-2.0 lizenziert und im Paket `adapter::conversion::unidirectional` eingeordnet (siehe CompilerInfo).

Zustandsübersicht

Der Funktionsblock besitzt keine explizite Zustandsmaschine. Er arbeitet als reine Kombinationslogik mit ereignisgesteuerter Auslösung:

- **Ruhezustand**: Es liegt kein Ereignis an `IN.E1` an. Ausgänge bleiben unverändert.
- **Aktiv**: Ein Ereignis an `IN.E1` startet die Umwandlung. Nach Abschluss wird `OUT.E1` ausgelöst und der neue Wert an `OUT.D1` ausgegeben. Danach kehrt der Baustein in den Ruhezustand zurück.

Anwendungsszenarien

- **Übersetzung von Bussignalen**: In einem industriellen Bussystem werden boolesche Steuersignale (z. B. Bit-Werte von Sensoren) in kompakte 4-Bit-Datenwörter umgewandelt, um die Bandbreite zu reduzieren.
- **Bit-zu-Quarter-Byte-Wandler in Automatisierungsnetzwerken**: Wenn ein Subsystem nur Quarter-Byte-Werte verarbeiten kann, aber boolesche Eingänge anliegen.
- **Adapterbrücke**: Der Baustein fungiert als Zwischenstück zwischen einem AX-kompatiblen Sender und einem AQ-kompatiblen Empfänger in einer unidirektionalen Kette.

Vergleich mit ähnlichen Bausteinen

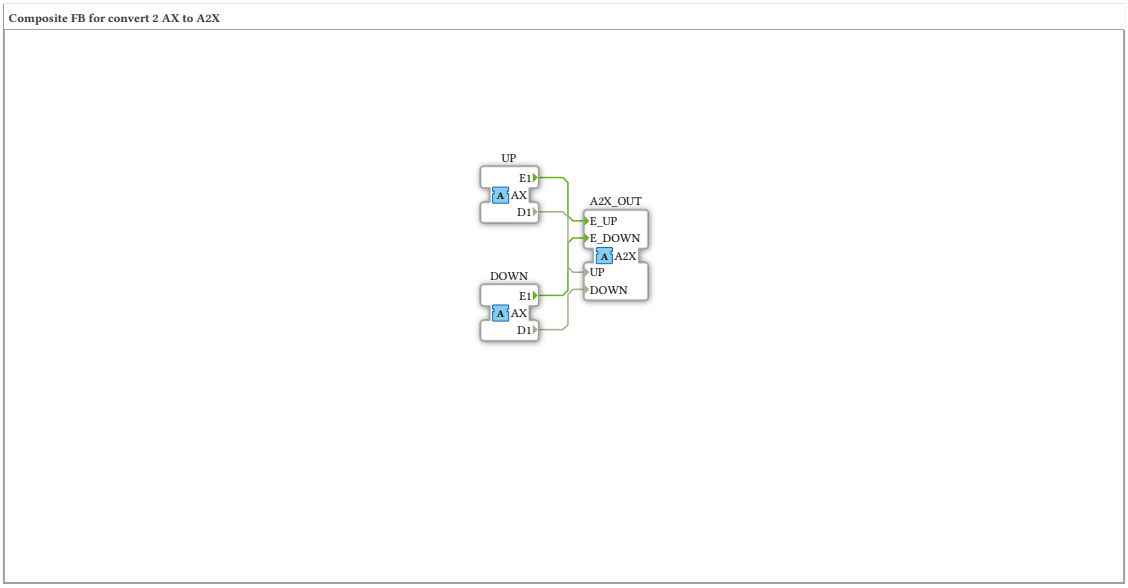
Baustein	Aufgabe	Unterschied
`BOOL_TO_BYTE`	Konvertiert BOOL in ein Byte (8-Bit)	Arbeitet mit 8-Bit- statt 4-Bit-Ausgabe. Benötigt andere Adaptertypen (z. B. `AB`).
`BOOL_TO_WORD`	Konvertiert BOOL in ein Word (16-Bit)	Liefert eine breitere Ausgabe, oft für analoge oder digitale Wort-Signale.
`AX_TO_AQ`	Konvertiert BOOL (AX) in Quarter Byte (AQ)	Speziell auf die unidirektionale Adapterkombination AX↔AQ zugeschnitten und erzeugt nur 4-Bit-Werte.

Fazit

Der **AX_TO_AQ**-Funktionsblock ist ein spezialisierter und schlanker Konverter für die Umwandlung boolescher Adaptersignale in das Quarter-Byte-Format. Durch die klare ereignisgesteuerte Struktur und die interne Wiederverwendung eines standardisierten Umwandlungsbausteins eignet er sich ideal für den Einsatz in Automatisierungs- und Bussystemen, in denen eine platzsparende Übermittlung boolescher Zustände erforderlich ist. Die einfache Schnittstelle und die Unidirektionalität erleichtern die Integration in bestehende Kommunikationsabläufe.

BOOL

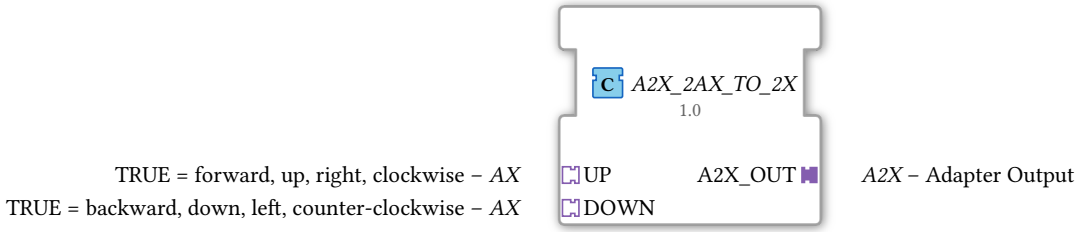
A2X_2AX_TO_2X



• – ◦ • – ◦ • – ◦ •

Einleitung

Der A2X_2AX_TO_2X ist ein Composite-Funktionsblock, der zur Konvertierung von zwei AX-Signalen in ein A2X-Signal dient. Dieser Baustein ermöglicht die Zusammenführung von zwei unidirektionalen AX-Adapter-Signalen in einen einzigen A2X-Adapter-Ausgang.



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Sockets (Eingänge):

- **UP**: AX-Adapter für positive Bewegungsrichtung (TRUE = vorwärts, oben, rechts, im Uhrzeigersinn)
- **DOWN**: AX-Adapter für negative Bewegungsrichtung (TRUE = rückwärts, unten, links, gegen den Uhrzeigersinn)

Plugs (Ausgänge):

- **A2X_OUT**: A2X-Adapter-Ausgang für die kombinierten Signale

Funktionsweise

Der Composite-Funktionsblock verbindet die beiden AX-Adapter-Eingänge (UP und DOWN) direkt mit dem A2X-Adapter-Ausgang. Dabei werden sowohl die Ereignis- als auch die Datensignale weitergeleitet:

- UP-Ereignisse und -Daten werden an E_UP und UP des A2X_OUT-Adapters weitergeleitet
- DOWN-Ereignisse und -Daten werden an E_DOWN und DOWN des A2X_OUT-Adapters weitergeleitet

Technische Besonderheiten

- Implementiert als Composite-FB ohne interne Logik
- Direkte Durchschaltung der Signale ohne Verzögerung
- Verwendet unidirektionale Adapter-Typen
- Keine Zustandsbehaftung oder interne Verarbeitung

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände, da es sich um eine reine Durchschaltung handelt.

Anwendungsszenarien

- Zusammenführung von zwei separaten Bewegungssteuerungssignalen
- Konvertierung von AX-basierten Steuerungssystemen zu A2X-Schnittstellen
- Integration in größere Steuerungssysteme mit A2X-Schnittstellenanforderungen
- Bidirektionale Bewegungserfassung in Automatisierungssystemen

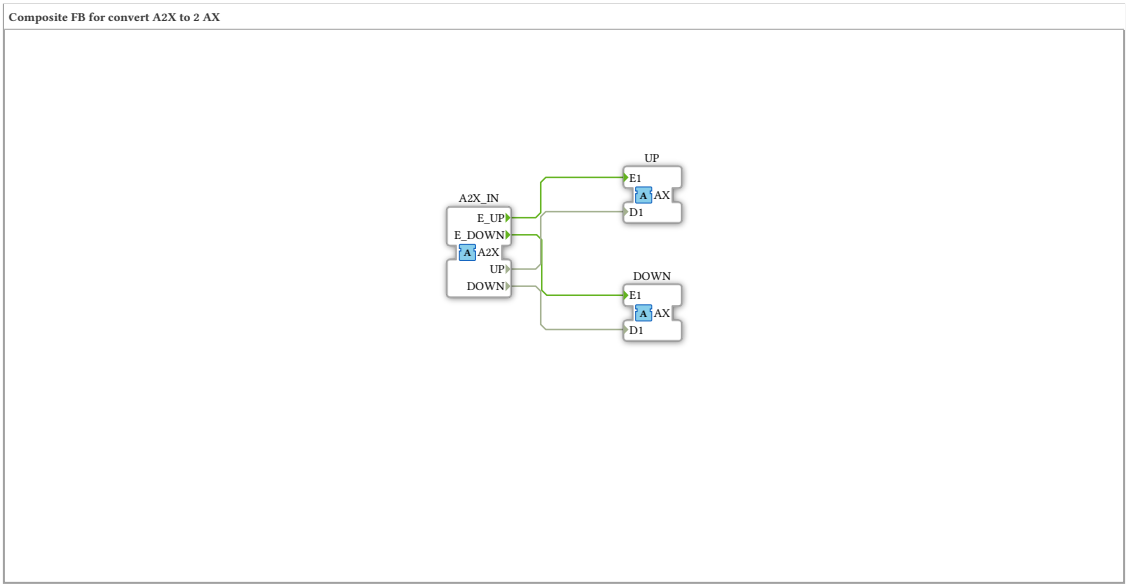
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen AX-Adaptern ermöglicht dieser Baustein die Kombination zweier entgegengesetzter Bewegungsrichtungen in einem einzigen A2X-Signal. Er ersetzt die manuelle Verkabelung von zwei AX-Adaptern zu einem A2X-Adapter.

Fazit

Der A2X_2AX_TO_2X ist ein einfacher aber effektiver Composite-Funktionsblock, der die Integration von AX-basierten Steuerungskomponenten in Systeme mit A2X-Schnittstellen vereinfacht. Durch die direkte Signalweiterleitung ohne zusätzliche Verarbeitung gewährleistet er eine zuverlässige und verlustfreie Signalübertragung.

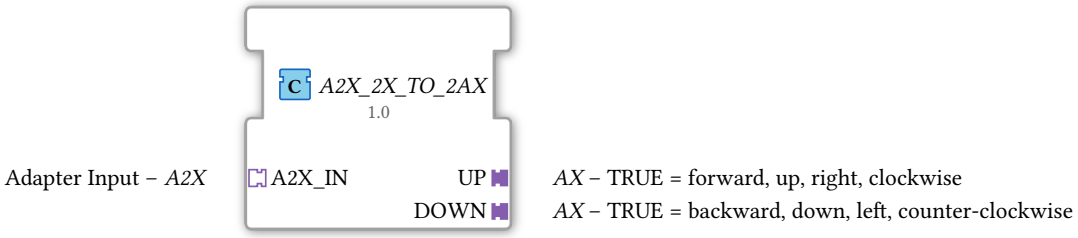
A2X_2X_TO_2AX



• – ○ • – ○ • – ○ •

Einleitung

Der A2X_2X_TO_2AX ist ein Composite-Funktionsblock, der zur Konvertierung von A2X-Signalen in zwei separate AX-Signale dient. Dieser Baustein ermöglicht die Umwandlung bidirektionaler Steuersignale in unidirektionale Bewegungsbefehle.



Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkten Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

- Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

- Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

- Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- `A2X_IN` (Socket) - Adapter-Eingang vom Typ `adapter::types::unidirectional::A2X`

Ausgangsadapter:

- `UP` (Plug) - Ausgang für positive Bewegungsrichtung (TRUE = vorwärts, oben, rechts, im Uhrzeigersinn)
- `DOWN` (Plug) - Ausgang für negative Bewegungsrichtung (TRUE = rückwärts, unten, links, gegen den Uhrzeigersinn)

Funktionsweise

Der Composite-Funktionsblock leitet die Ereignisse und Daten direkt vom A2X_IN-Adapter an die entsprechenden UP- und DOWN-Ausgangsadapter weiter:

- Das E_UP-Ereignis wird an UP.E1 weitergeleitet
- Das E_DOWN-Ereignis wird an DOWN.E1 weitergeleitet
- Die UP-Daten werden an UP.D1 weitergeleitet
- Die DOWN-Daten werden an DOWN.D1 weitergeleitet

Technische Besonderheiten

- Implementiert als Composite-FB ohne interne Logik
- Direkte Signalweiterleitung ohne Verzögerung
- Verwendet unidirektionale Adaptertypen für klare Signalflussrichtung

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand und arbeitet stateless. Alle Eingangssignale werden sofort an die entsprechenden Ausgänge weitergeleitet.

Anwendungsszenarien

- Umwandlung von bidirektionalen Antriebssignalen in separate Vorwärts/Rückwärts-Steuerungen
- Ansteuerung von Aktoren mit separaten Ein- und Ausfahrbefehlen
- Integration in Steuerungssysteme mit unterschiedlichen Signalformaten
- Drehzahlregelung mit separaten Rechts-/Linkslauf-Steuerungen

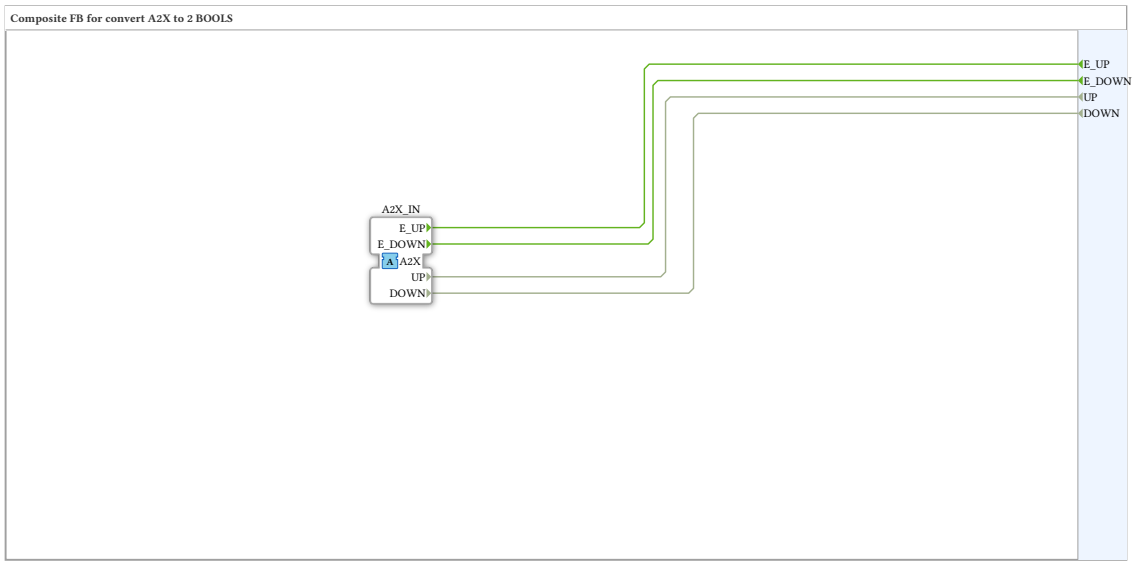
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen Konverter-Bausteinen bietet dieser Composite-FB eine strukturierte Lösung für die Aufteilung bidirektionaler Signale in zwei unabhängige unidirektionale Kanäle. Die Verwendung standardisierter Adaptertypen gewährleistet Kompatibilität innerhalb des Adapter-Frameworks.

Fazit

Der A2X_2X_TO_2AX Funktionsblock stellt eine effiziente und standardkonforme Lösung zur Signalumwandlung dar. Durch seine einfache, direkte Weiterleitungslogik und die Verwendung etablierter Adaptertypen eignet er sich ideal für die Integration in komplexere Steuerungssysteme, die eine Trennung von Bewegungsrichtungen erfordern.

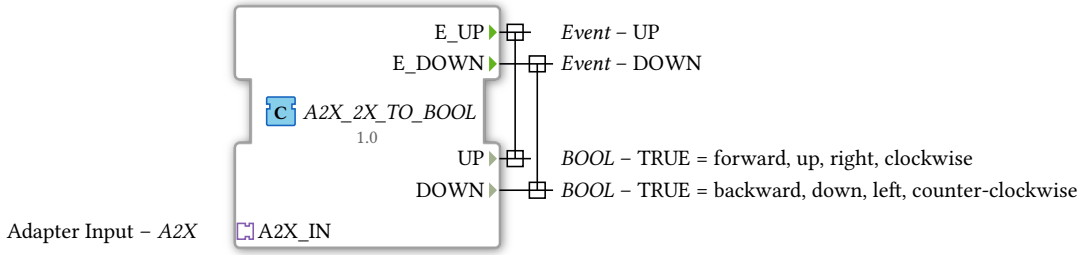
A2X_2X_TO_BOOL



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock A2X_2X_TO_BOOL ist ein Composite-Funktionsblock, der zur Konvertierung von A2X-Signalen in zwei separate BOOL-Signale dient. Er ermöglicht die Umwandlung von bidirektionalen Steuersignalen in einfache logische Ausgänge für verschiedene Anwendungsfälle.



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

- **E_UP**: Wird aktiviert, wenn eine Aufwärts-/Vorwärtsbewegung erkannt wird
- **E_DOWN**: Wird aktiviert, wenn eine Abwärts-/Rückwärtsbewegung erkannt wird

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

- **UP**: BOOL - TRUE für Vorwärts, Aufwärts, Rechts oder im Uhrzeigersinn
- **DOWN**: BOOL - TRUE für Rückwärts, Abwärts, Links oder gegen den Uhrzeigersinn

Adapter

- **A2X_IN**: Adapter-Eingang vom Typ A2X für unidirektionale Kommunikation

Funktionsweise

Der Funktionsblock empfängt Signale über den A2X-Adapter und leitet diese direkt an die entsprechenden Ausgänge weiter. Die Ereignisse und Datenwerte werden 1:1 vom Adapter an die Ausgangsschnittstellen durchgereicht, ohne zusätzliche Verarbeitung oder Verzögerung.

Technische Besonderheiten

- Composite-Funktionsblock ohne interne Logik
- Direkte Durchleitung der Adapter-Signale
- Keine Zustandsbehaftung oder Speicherung
- Echtzeitfähige Signalweiterleitung

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Er verhält sich rein kombinatorisch und leitet eingehende Signale sofort an die Ausgänge weiter.

Anwendungsszenarien

- Steuerung von Bewegungsrichtungen bei Aktoren
- Umwandlung von komplexeren Steuersignalen in einfache Richtungsbefehle
- Anbindung von A2X-kompatiblen Komponenten an BOOL-basierte Steuerungen
- Richtungserkennung in Antriebssystemen

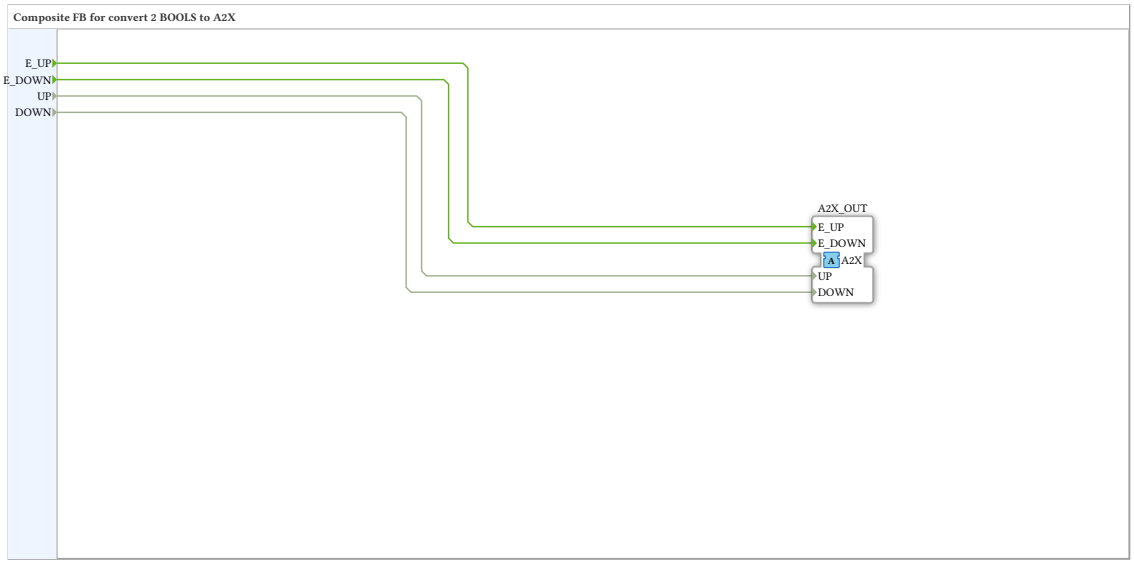
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Konvertierungsbausteinen bietet A2X_2X_TO_BOOL eine spezifische Lösung für die Umwandlung von A2X-Signalen in boolesche Richtungssignale. Andere Bausteine könnten zusätzliche Funktionen wie Filterung, Verzögerung oder komplexere Signalverarbeitung bieten.

Fazit

Der A2X_2X_TO_BOOL Funktionsblock stellt eine einfache und effiziente Lösung zur Konvertierung von A2X-Steuersignalen in boolesche Richtungsausgaben dar. Seine klare Schnittstellendefinition und direkte Signalweiterleitung machen ihn besonders geeignet für Anwendungen, bei denen eine unverfälschte Übertragung der Steuerinformationen erforderlich ist.

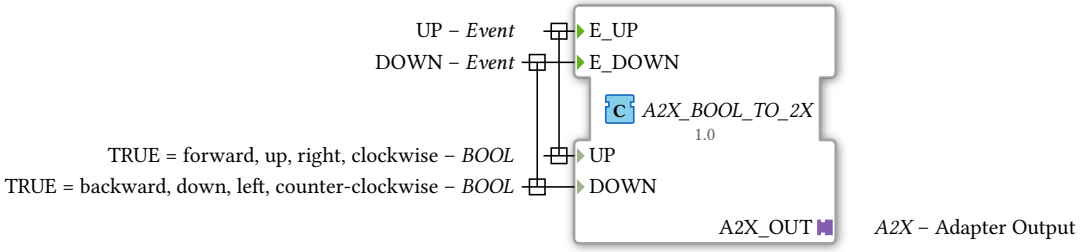
A2X_BOOL_TO_2X



• – ◦ • – ◦ • – ◦ •

Einleitung

Der A2X_BOOL_TO_2X ist ein zusammengesetzter Funktionsblock, der zwei BOOL-Werte in ein A2X-Ausgabeformat konvertiert. Dieser Baustein dient als Brücke zwischen einfachen booleschen Steuersignalen und dem standardisierten A2X-Adapterinterface.



Schnittstellenstruktur

Ereignis-Eingänge

- **E_UP**: Ereigniseingang für die Aufwärts-/Vorwärtsbewegung
- **E_DOWN**: Ereigniseingang für die Abwärts-/Rückwärtsbewegung

Ereignis-Ausgänge

Keine direkten Ereignisausgänge vorhanden

Daten-Eingänge

- **UP**: BOOL-Eingang für Aufwärts-/Vorwärtsrichtung (TRUE = vorwärts, oben, rechts, im Uhrzeigersinn)
- **DOWN**: BOOL-Eingang für Abwärts-/Rückwärtsrichtung (TRUE = rückwärts, unten, links, gegen den Uhrzeigersinn)

Daten-Ausgänge

Keine direkten Datenausgänge vorhanden

Adapter

- **A2X_OUT**: Unidirektionaler A2X-Ausgabeadapter für die Weiterleitung der konvertierten Signale

Funktionsweise

Der Funktionsblock empfängt zwei unabhängige boolesche Eingangssignale (UP und DOWN) zusammen mit den zugehörigen Ereigniseingängen. Diese werden direkt an den A2X-Ausgabeadapter weitergeleitet, wobei die Ereignis- und Datenverbindungen parallel verarbeitet werden.

Technische Besonderheiten

- Komplett unidirektionale Signalverarbeitung
- Direkte Durchschaltung ohne zusätzliche Logikverarbeitung
- Verwendet das standardisierte A2X-Adapterinterface
- Keine internen Zustände oder Verzögerungen

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände und arbeitet stateless. Jedes eingehende Ereignis wird sofort an den entsprechenden Adapterausgang weitergeleitet.

Anwendungsszenarien

- Steuerung von bidirektionalen Aktoren (z.B. Motoren, Zylinder)
- Interface zwischen einfachen Schaltlogiken und komplexeren Steuerungssystemen
- Konvertierung von Taster-/Schaltersignalen in standardisierte Bewegungssignale
- Antriebssteuerungen mit Vorwärts-/Rückwärtsfunktion

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen Konvertierungsbausteinen bietet A2X_BOOL_TO_2X die spezifische Kombination aus zwei booleschen Eingängen mit einem standardisierten A2X-Interface. Andere Bausteine könnten einzelne BOOL-zu-A2X-Konvertierungen durchführen, dieser Baustein kombiniert jedoch beide Richtungen in einer einzigen Komponente.

Fazit

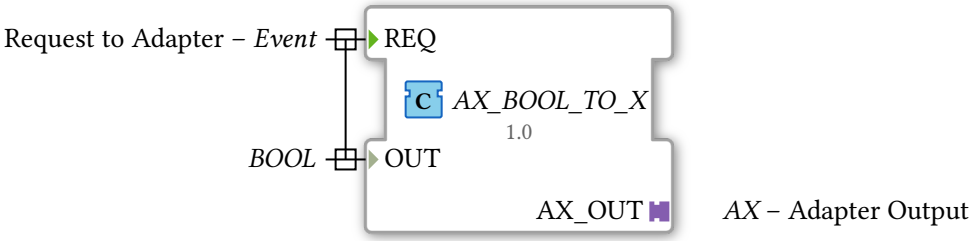
Der A2X_BOOL_TO_2X ist ein spezialisierter Interface-Baustein, der eine einfache und effiziente Schnittstelle zwischen grundlegenden booleschen Steuersignalen und dem standardisierten A2X-Protokoll bereitstellt. Seine klare Struktur und direkte Durchschaltung machen ihn zu einer zuverlässigen Lösung für bidirektionale Steuerungsanwendungen.

AX_BOOL_TO_X

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_BOOL_TO_X Funktionsblock ist ein Composite-Funktionsblock, der zur Konvertierung eines BOOL-Werts in das AX-Adapterformat dient. Er ermöglicht die einfache Umwandlung von booleschen Werten in ein standardisiertes Ausgabeformat für die weitere Verarbeitung in Automatisierungssystemen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Anfrage-Event zur Auslösung der Konvertierung

Ereignis-Ausgänge

- Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

- **OUT**: BOOL-Eingangswert, der konvertiert werden soll

Daten-Ausgänge

- Keine direkten Daten-Ausgänge vorhanden

Adapter

- **AX_OUT**: Unidirektionaler AX-Adapter-Ausgang für die konvertierten Daten

Funktionsweise

Der Funktionsblock empfängt einen booleschen Wert am Daten-Eingang OUT und wandelt diesen bei Eintreffen des REQ-Ereignisses in das AX-Adapterformat um. Die Konvertierung erfolgt automatisch und der konvertierte Wert wird über den AX_OUT-Adapter ausgegeben. Der Block fungiert als Brücke zwischen einfachen booleschen Signalen und dem standardisierten AX-Kommunikationsformat.

Technische Besonderheiten

- Composite-Funktionsblock mit interner Vernetzung
- Verwendet unidirektionalen AX-Adapter für die Ausgabe
- Automatische Weiterleitung von Ereignissen und Daten an den Adapter
- Einfache und direkte Signalumwandlung ohne zusätzliche Logik

Zustandsübersicht

Der Funktionsblock besitzt einen einfachen Zustandsautomaten:

1. **Wartezustand**: Wartet auf REQ-Ereignis
2. **Verarbeitungszustand**: Bei REQ-Ereignis wird der BOOL-Wert konvertiert und über den Adapter ausgegeben
3. Rückkehr in Wartezustand nach Abschluss der Konvertierung

Anwendungsszenarien

- Integration von einfachen Schaltzuständen in komplexere AX-basierte Systeme
- Schnittstellenanpassung zwischen verschiedenen Automatisierungskomponenten
- Protokollumwandlung in verteilten Steuerungssystemen
- Einbindung von Basis-Sensoren in AX-konforme Architekturen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Konvertierungsbausteinen zeichnet sich AX_BOOL_TO_X durch seine spezifische Ausrichtung auf das AX-Adapterformat aus. Während allgemeine Konvertierungsblöcke oft mehrere Ausgabeformate unterstützen, bietet dieser Block eine optimierte Lösung für die spezifische AX-Schnittstelle.

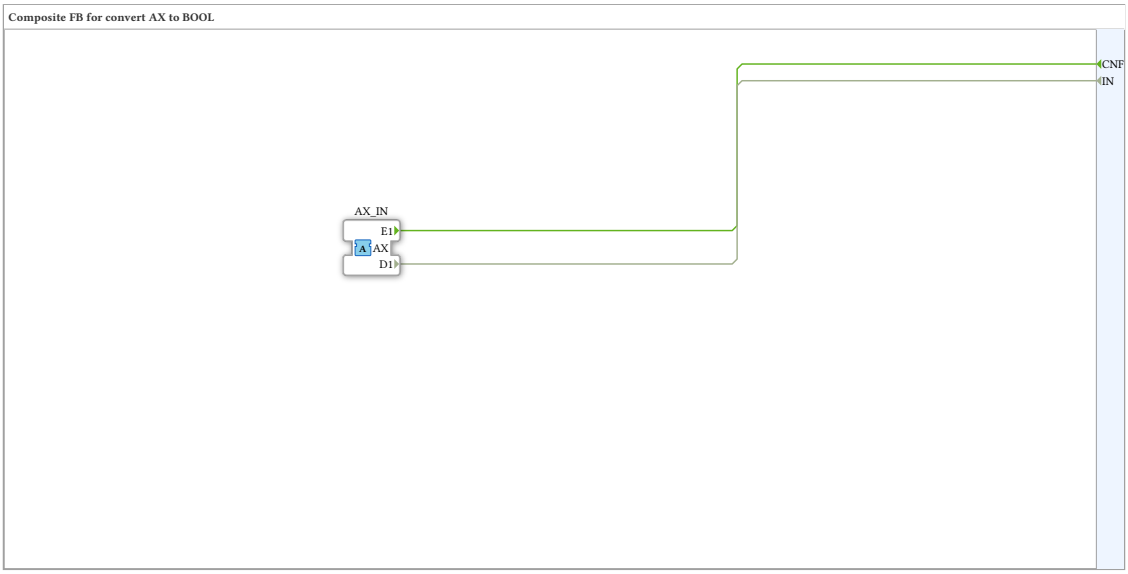
🛠️ Zugehörige Übungen

- Uebung_001_AX_b
- Uebung_002a2_AX
- Uebung_004b_AX
- Uebung_004b_AX_ASR
- Uebung_004b_AX_ASR_X

Fazit

AX_BOOL_TO_X ist ein spezialisierter Konvertierungsbaustein, der eine einfache und effiziente Schnittstelle zwischen booleschen Signalen und dem AX-Adapterformat bereitstellt. Seine klare Struktur und direkte Funktionsweise machen ihn zu einer zuverlässigen Lösung für Integrationsaufgaben in Automatisierungssystemen.

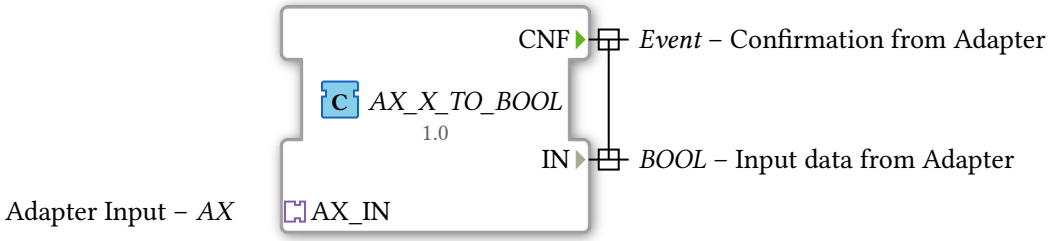
AX_X_TO_BOOL



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AX_X_TO_BOOL ist ein Composite-Funktionsblock, der für die Konvertierung von AX-Daten in den BOOL-Datentyp entwickelt wurde. Dieser Baustein dient als Adapter-Konverter und ermöglicht die unidirektionale Umwandlung von Daten zwischen unterschiedlichen Datentypen in 4diac-Systemen.



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

- **CNF** (Confirmation): Bestätigungsereignis, das vom Adapter ausgegeben wird.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

- **IN** (BOOL): Konvertierte Eingangsdaten vom Adapter.

Adapter

- **AX_IN**: Adapter-Eingang vom Typ AX für unidirektionale Datenübertragung.

Funktionsweise

Der Funktionsblock empfängt Daten über den AX_IN-Adapter und konvertiert diese in den BOOL-Datentyp. Bei erfolgreicher Konvertierung wird das Bestätigungsereignis CNF ausgelöst und die konvertierten

Daten am Ausgang IN bereitgestellt. Die interne Struktur verbindet direkt den Adapter-Ereignisausgang E1 mit dem CNF-Ausgang und den Datenausgang D1 des Adapters mit dem IN-Datenausgang.

Technische Besonderheiten

- Implementiert als Composite-Funktionsblock
- Verwendet unidirektionale Adapter-Kommunikation
- Direkte Durchschaltung ohne zusätzliche Verarbeitungslogik
- Einfache und effiziente Datenkonvertierung

Zustandsübersicht

Der Funktionsblock besitzt einen einfachen Zustand:

- **Bereit**: Wartet auf eingehende Daten vom Adapter
- Bei Empfang von Daten erfolgt sofortige Konvertierung und Ausgabe

Anwendungsszenarien

- Integration von AX-basierten Komponenten in BOOL-orientierte Steuerungen
- Adapter-Konvertierung in heterogenen Automatisierungssystemen
- Schnittstellenanpassung zwischen verschiedenen Gerätetypen
- Protokollumsetzung in verteilten Steuerungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Konvertierungsbausteinen zeichnet sich AX_X_TO_BOOL durch seine spezifische Fokussierung auf die AX-zu-BOOL-Konvertierung aus. Während universelle Konverter häufig komplexere Parametrierung erfordern, bietet dieser Baustein eine optimierte Lösung für diesen speziellen Anwendungsfall.

🛠️ Zugehörige Übungen

- Uebung_001_AX_b
- Uebung_002a2_AX
- Uebung_004b_AX
- Uebung_004b_AX_ASR
- Uebung_004b_AX_ASR_X
- Uebung_094a_AX

Fazit

AX_X_TO_BOOL ist ein spezialisierter Konvertierungsbaustein, der eine effiziente und zuverlässige Umwandlung von AX-Daten in BOOL-Werte ermöglicht. Seine einfache Struktur und direkte Funktionsweise machen ihn besonders geeignet für Systeme, in denen performante Datentyp-Konvertierung erforderlich ist.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Unidirectional BOOL Adapter Conversion

🎧 Podcast

- George Boole: Wie ein Autodidakt mit 0 und 1 das Fundament unserer digitalen Welt legte

🛠️ Zugehörige Übungen

- Uebung_001_AX_b
- Uebung_002a2_AX
- Uebung_002a4
- Uebung_002a5b
- Uebung_002b3
- Uebung_004b_AX
- Uebung_006a3
- LinksRechts_T_FF_Event

- Uebung_035a2
- Uebung_035a3
- SoftkeyOverride_TO_QX_BG
- Uebung_055
- Uebung_056
- Bool8_TO_logiBUS_QX8
- Uebung_094a_AX
- Uebung_160

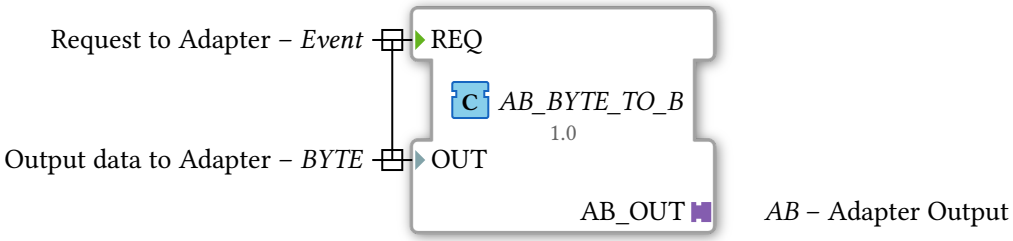
BYTE

AB_BYTE_TO_B

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AB_BYTE_TO_B ist ein Composite-Funktionsblock, der zur Konvertierung eines BYTE-Datentyps in das AB-Adapterformat dient. Er ermöglicht die unidirektionale Datenübertragung von Byte-Werten über ein standardisiertes Adapter-Interface.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Anforderungsereignis zur Auslösung der Adapter-Konvertierung

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

- **OUT**: BYTE-Eingangswert, der an den Adapter weitergeleitet wird

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

- **AB_OUT**: Unidirektionaler AB-Adapterausgang für die Datenausgabe

Funktionsweise

Der Funktionsblock empfängt über den REQ-Ereigniseingang eine Anforderung zur Datenkonvertierung. Gleichzeitig wird der Byte-Wert am Daten-Eingang OUT verarbeitet. Bei Eintreffen des REQ-Ereignisses wird der Byte-Wert über den AB_OUT-Adapter ausgegeben. Die Datenübertragung erfolgt unidirektional vom Funktionsblock zum angeschlossenen Adapter.

Technische Besonderheiten

- Implementiert als Composite-Funktionsblock mit interner Netzwerkstruktur
- Verwendet das unidirektionale AB-Adapter-Interface
- Direkte Ereignis- und Datenverbindungen zwischen Eingängen und Adapter
- Positionierungsattribute für grafische Darstellung in der Entwicklungsumgebung

Zustandsübersicht

Der Funktionsblock besitzt einen einfachen Zustandsautomaten:

1. **Wartezustand**: Keine aktive Verarbeitung
2. **Aktivierungszustand**: Bei REQ-Ereignis werden Daten an den Adapter weitergeleitet
3. **Rückkehr**: Automatische Rückkehr in Wartezustand nach Verarbeitung

Anwendungsszenarien

- Konvertierung von Byte-Daten für Adapter-basierte Kommunikation
- Integration in größere Steuerungssysteme mit standardisierten Adapter-Schnittstellen
- Datenaufbereitung für nachgelagerte Verarbeitungskomponenten
- Protokollumsetzung bei unterschiedlichen Datentypen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen Konvertierungsblöcken bietet AB_BYTE_TO_B den Vorteil der standardisierten Adapter-Schnittstelle, was eine bessere Wiederverwendbarkeit und Integration in komplexere Systemarchitekturen ermöglicht. Die unidirektionale Ausrichtung macht ihn speziell für Ausgabeszenarien geeignet.

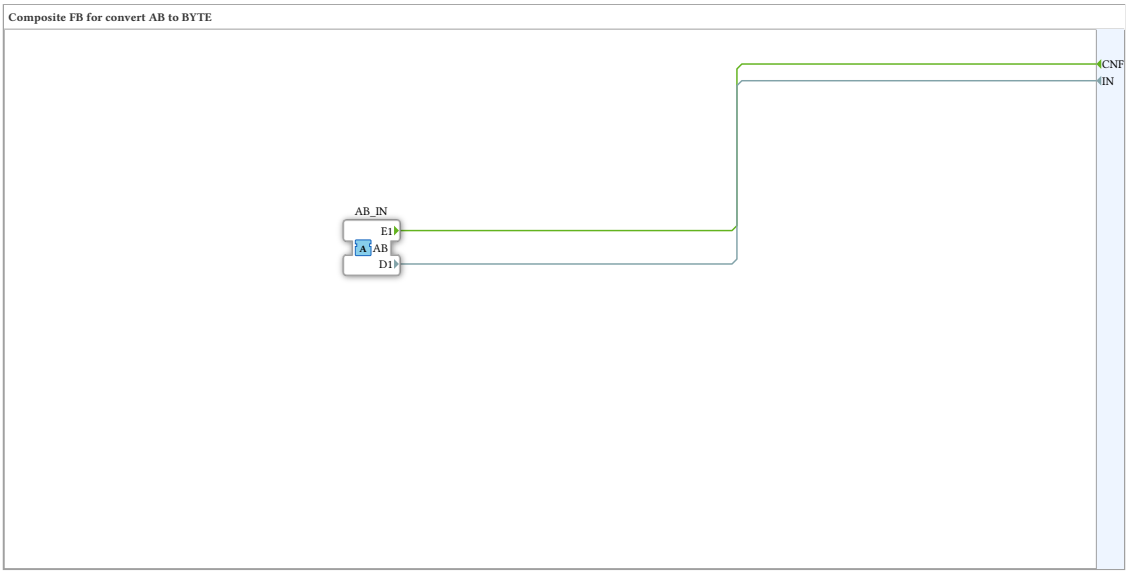
Fazit

AB_BYTE_TO_B ist ein spezialisierter Konvertierungsbaustein, der durch seine Adapter-basierte Architektur eine flexible und standardkonforme Integration in 4diac-basierte Steuerungssysteme ermöglicht. Seine einfache und robuste Funktionsweise macht ihn zu einer zuverlässigen Lösung für Byte-zu-Adapter-Konvertierungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

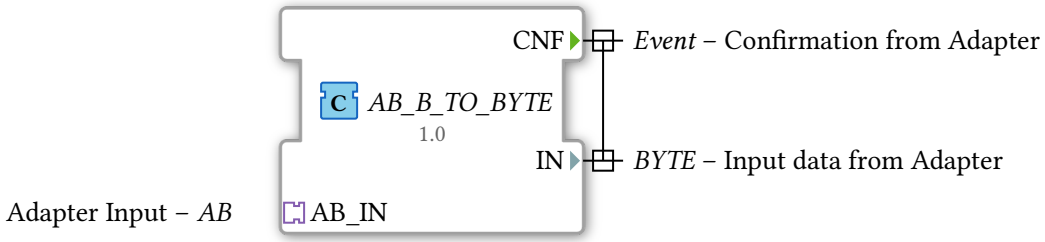
AB_B_TO_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AB_B_TO_BYTE ist ein Composite-Funktionsblock, der der Konvertierung von AB-Daten in das BYTE-Format dient. Er fungiert als Adapter-Konverter und ermöglicht die nahtlose Integration von AB-Datenquellen in Systeme, die BYTE-Daten erwarten.



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden - Ereignisse werden über den Adapter empfangen.

Ereignis-Ausgänge

- **CNF** (Confirmation): Bestätigungsereignis, das vom Adapter empfangen wird

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden - Daten werden über den Adapter empfangen.

Daten-Ausgänge

- **IN** (BYTE): Eingangsdaten, die vom Adapter konvertiert wurden

Adapter

- **AB_IN**: Adapter-Eingang vom Typ `adapter::types::unidirectional::AB` für den Empfang von AB-Daten

Funktionsweise

Der Funktionsblock empfängt AB-Daten über den Adapter-Eingang AB_IN und konvertiert diese in das BYTE-Format. Bei Empfang von Daten über den Adapter wird automatisch das Bestätigungsereignis CNF ausgelöst und die konvertierten Daten am Ausgang IN bereitgestellt.

Technische Besonderheiten

- Implementiert als Composite-FB mit interner Netzwerkstruktur
- Verwendet unidirektionale Adapter-Kommunikation
- Automatische Ereignisweiterleitung vom Adapter zum CNF-Ausgang
- Direkte Datenweiterleitung vom Adapter zum IN-Ausgang

Zustandsübersicht

Der Funktionsblock arbeitet zustandslos und reagiert sofort auf eingehende Adapter-Ereignisse. Bei jedem E1-Ereignis vom Adapter wird das CNF-Ereignis ausgelöst und die aktuellen D1-Daten des Adapters an IN ausgegeben.

Anwendungsszenarien

- Integration von AB-basierten Sensoren oder Aktoren in BYTE-orientierte Steuerungssysteme
- Protokollkonvertierung in industriellen Automatisierungsanlagen
- Adapter-basierte Datenumwandlung in verteilten Steuerungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen Konvertierungsbausteinen bietet AB_B_TO_BYTE den Vorteil der Adapter-Integration, was eine standardisierte und typsichere Schnittstelle ermöglicht. Andere Konvertierungsbausteine arbeiten typischerweise mit direkten Ein-/Ausgängen statt mit Adaptern.

Fazit

AB_B_TO_BYTE stellt eine elegante Lösung für die Konvertierung von AB- zu BYTE-Daten mittels Adapter-Schnittstellen dar. Die Composite-Implementierung gewährleistet eine robuste und zuverlässige Datenumwandlung, die besonders in komplexen Automatisierungssystemen mit standardisierten Kommunikationsschnittstellen vorteilhaft ist.

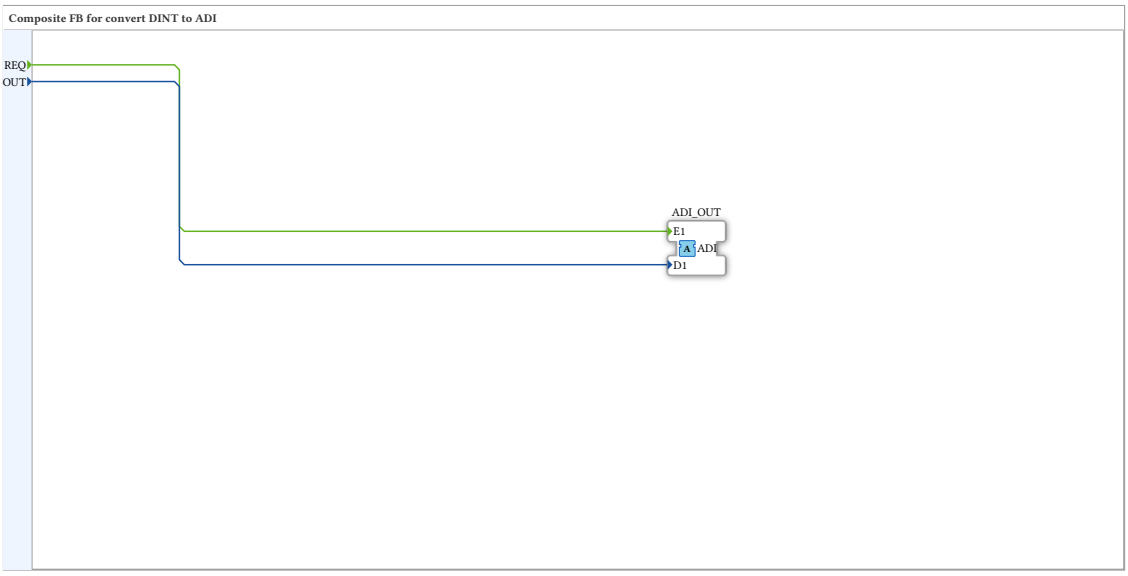
Unidirectional BYTE Adapter Conversion

🛠️ Zugehörige Übungen

- SIN_TO_PCAN_Callback_Byte
- Uebung_128b
- Uebung_12x_sub

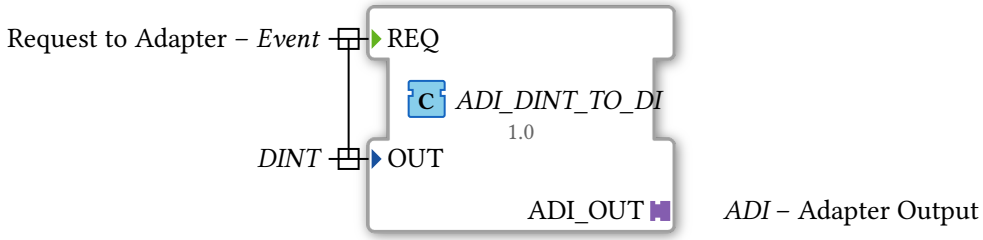
DINT

ADI_DINT_TO_DI



Einleitung

Der ADI_DINT_TO_DI Funktionsblock ist ein Composite-Funktionsblock, der zur Konvertierung eines DINT-Werts in das ADI-Adapterformat dient.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Anfrage-Event zur Auslösung der Konvertierung

Ereignis-Ausgänge

- Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

- **OUT**: DINT-Eingangswert, der konvertiert werden soll

Daten-Ausgänge

- Keine direkten Daten-Ausgänge vorhanden

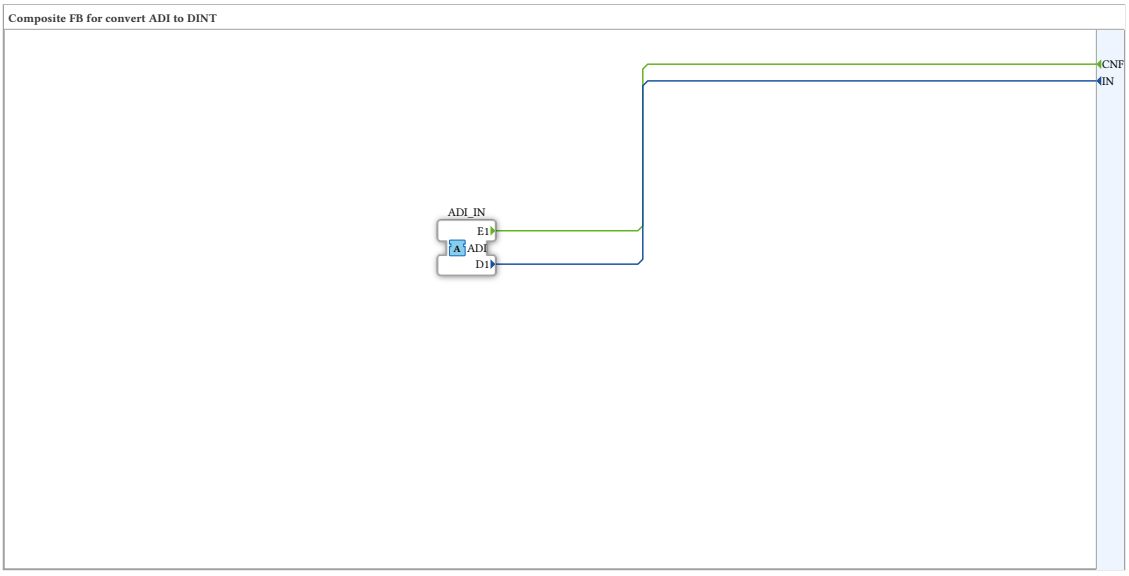
Adapter

- **ADI_OUT**: Plug vom Typ `ADI`

Funktionsweise

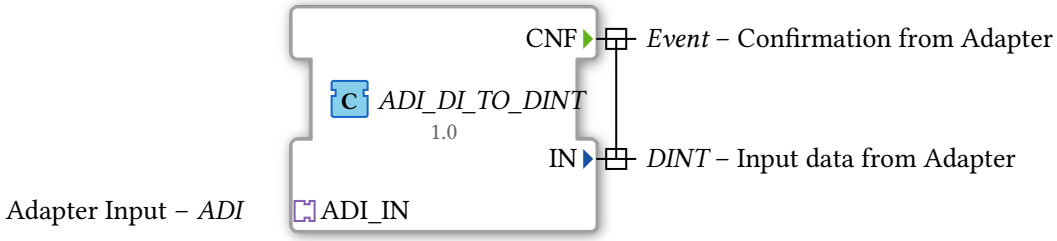
Der Funktionsblock empfängt einen DINT-Wert am Daten-Eingang OUT und wandelt diesen bei Eintreffen des REQ-Ereignisses in das ADI-Adapterformat um.

ADI_DI_TO_DINT



Einleitung

Der ADI_DI_TO_DINT Funktionsblock ist ein Composite-Funktionsblock, der zur Konvertierung des ADI-Adapterformats in einen DINT-Wert dient.



Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkten Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

- **CNF**: Bestätigungs-Event nach erfolgter Konvertierung

Daten-Eingänge

- Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

- **IN**: DINT-Ausgangswert der konvertierten Daten

Adapter

- **ADI_IN**: Socket vom Typ `ADI`

Funktionsweise

Der Funktionsblock empfängt Daten über den ADI-Adapter-Socket und stellt den enthaltenen DINT-Wert am Ausgang IN zur Verfügung, sobald ein Ereignis über den Adapter eintrifft (CNF).

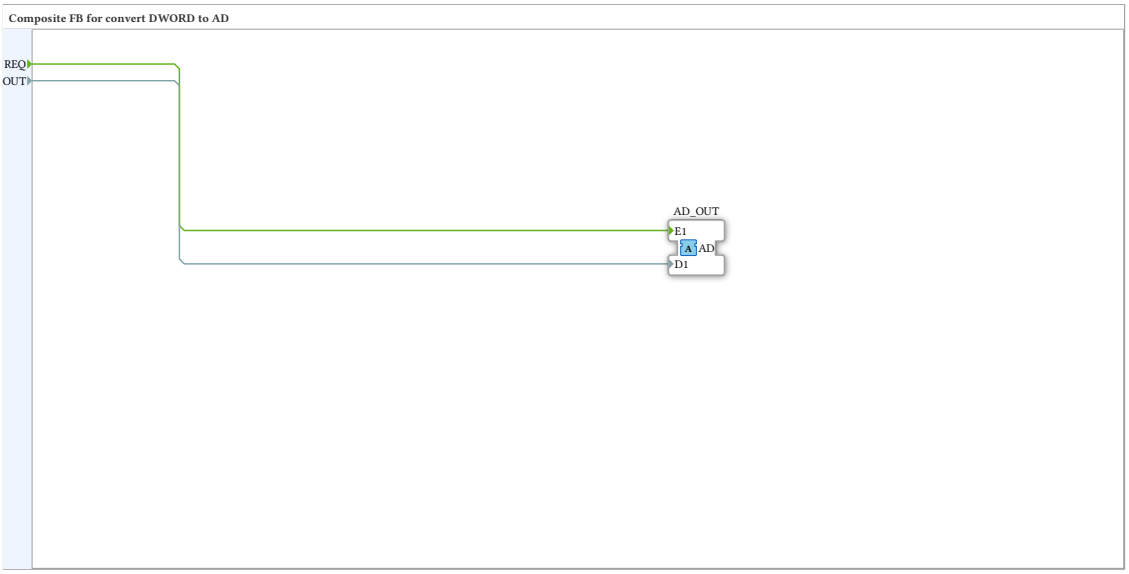
DINT Conversion

🎧 Podcast

- SINT, INT, DINT: Warum die Wahl des Datentyps über Effizienz und Fehler entscheidet

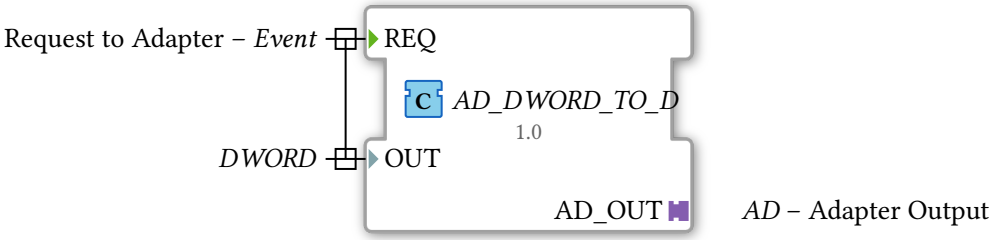
DWORD

AD_DWORD_TO_D



Einleitung

Der AD_DWORD_TO_D Funktionsblock ist ein Composite-Funktionsblock, der zur Konvertierung eines DWORD-Werts in das AD-Adapterformat dient.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Anfrage-Event zur Auslösung der Konvertierung

Ereignis-Ausgänge

- Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

- **OUT**: DWORD-Eingangswert, der konvertiert werden soll

Daten-Ausgänge

- Keine direkten Daten-Ausgänge vorhanden

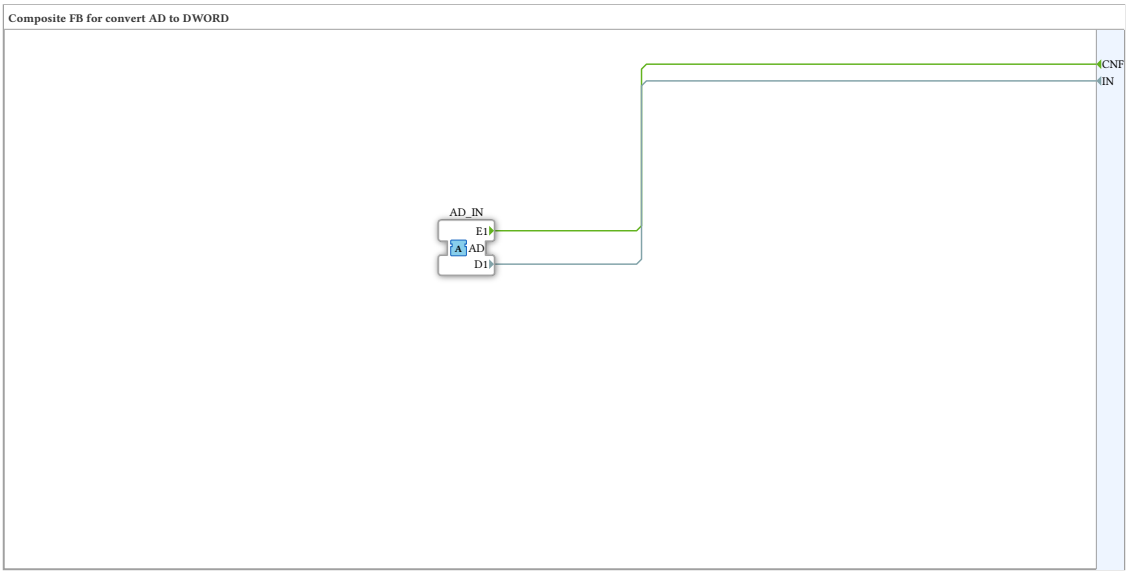
Adapter

- **AD_OUT**: Plug vom Typ `AD`

Funktionsweise

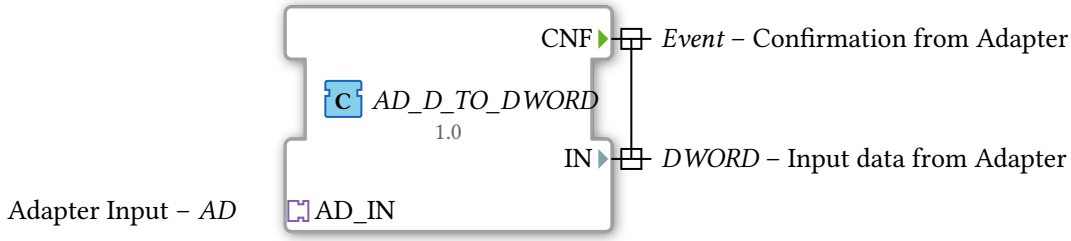
Der Funktionsblock empfängt einen DWORD-Wert am Daten-Eingang OUT und wandelt diesen bei Eintreffen des REQ-Ereignisses in das AD-Adapterformat um.

AD_D_TO_DWORD



Einleitung

Der AD_D_TO_DWORD Funktionsblock ist ein Composite-Funktionsblock, der zur Konvertierung des AD-Adapterformats in einen DWORD-Wert dient.



Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkten Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

- **CNF**: Bestätigungs-Event nach erfolgter Konvertierung

Daten-Eingänge

- Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

- **IN**: DWORD-Ausgangswert der konvertierten Daten

Adapter

- **AD_IN**: Socket vom Typ `AD`

Funktionsweise

Der Funktionsblock empfängt Daten über den AD-Adapter-Socket und stellt den enthaltenen DWORD-Wert am Ausgang IN zur Verfügung, sobald ein Ereignis über den Adapter eintrifft (CNF).

DWORD Conversion

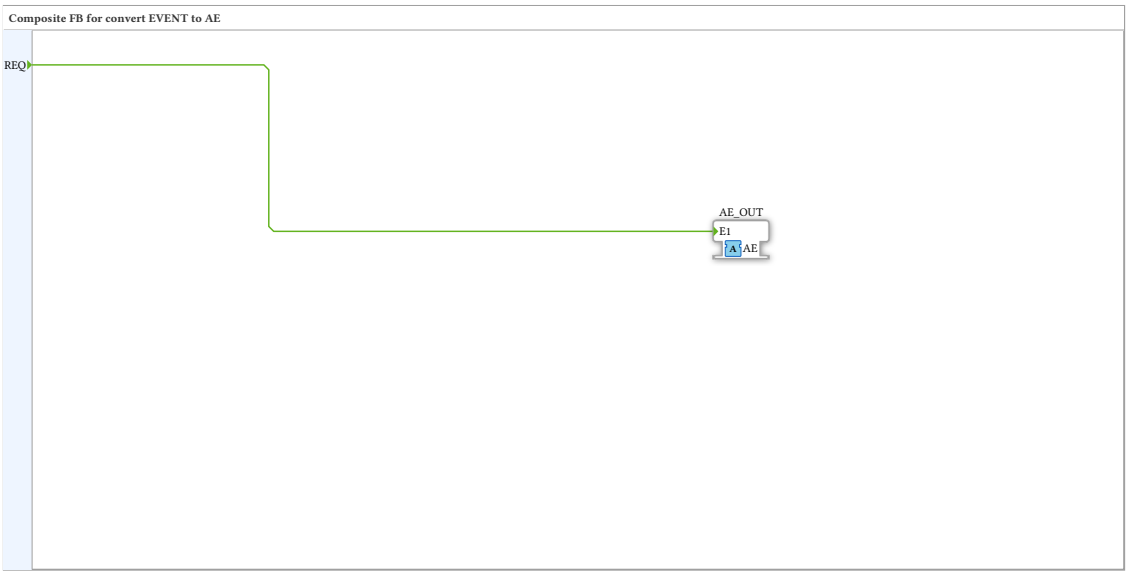
🛠️ Zugehörige Übungen

- Uebung_034b
- SIN_TO_PCAN_Callback_DWord
- Uebung_152
- Uebung_153

EVENT

- AE_E_TO_EVENT
- AE_EVENT_TO_E
- ASR_2EVENTS_TO_SR
- ASR_SR_TO_2EVENTS
- ASRT_3EVENTS_TO_SRT
- ASRT_SRT_TO_3EVENTS
- EVENT_Adapter_Conv

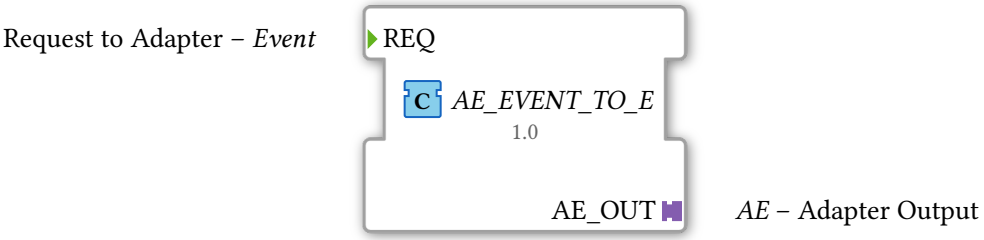
AE_EVENT_TO_E



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AE_EVENT_TO_E ist ein Composite-Funktionsblock, der zur Konvertierung von Ereignissen in AE-Adapter-Signale dient. Er ermöglicht die Umwandlung eines einfachen Ereignisses in ein Adapter-basiertes Ausgangssignal, was insbesondere bei der Integration verschiedener Kommunikationsschnittstellen nützlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Anfrage-Ereignis zur Auslösung der Adapter-Konvertierung

Ereignis-Ausgänge

- Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

- Keine Daten-Eingänge vorhanden

Daten-Ausgänge

- Keine Daten-Ausgänge vorhanden

Adapter

- **AE_OUT**: Unidirektionaler AE-Adapter-Ausgang für die Weiterleitung der konvertierten Ereignisse

Funktionsweise

Der Funktionsblock arbeitet als einfacher Konverter, der eingehende REQ-Ereignisse direkt an den AE_OUT-Adapter weiterleitet. Bei jedem Eintreffen eines REQ-Ereignisses wird das entsprechende E1-Ereignis am AE-Adapter-Ausgang ausgelöst. Es findet keine Datenverarbeitung oder -transformation statt, da der Block ausschließlich Ereignisse weiterleitet.

Technische Besonderheiten

- Composite-Funktionsblock mit einfacher Netzwerkstruktur
- Verwendet unidirektionale AE-Adapter-Schnittstelle
- Keine Datenverarbeitungskomponenten enthalten
- Direkte Ereignisweiterleitung ohne Verzögerung

Zustandsübersicht

Der Funktionsblock besitzt einen einfachen Zustand:

- **Bereit**: Wartet auf eingehende REQ-Ereignisse
- Bei Eintreffen von REQ wird sofort AE_OUT.E1 ausgelöst und der Block kehrt in den Bereitschaftszustand zurück

Anwendungsszenarien

- Integration von ereignisbasierten Systemen in Adapter-basierte Architekturen
- Kopplung unterschiedlicher Kommunikationsprotokolle
- Als Brücke zwischen einfachen Ereignissen und komplexeren Adapter-Schnittstellen
- In Systemen, die eine standardisierte AE-Schnittstelle erfordern

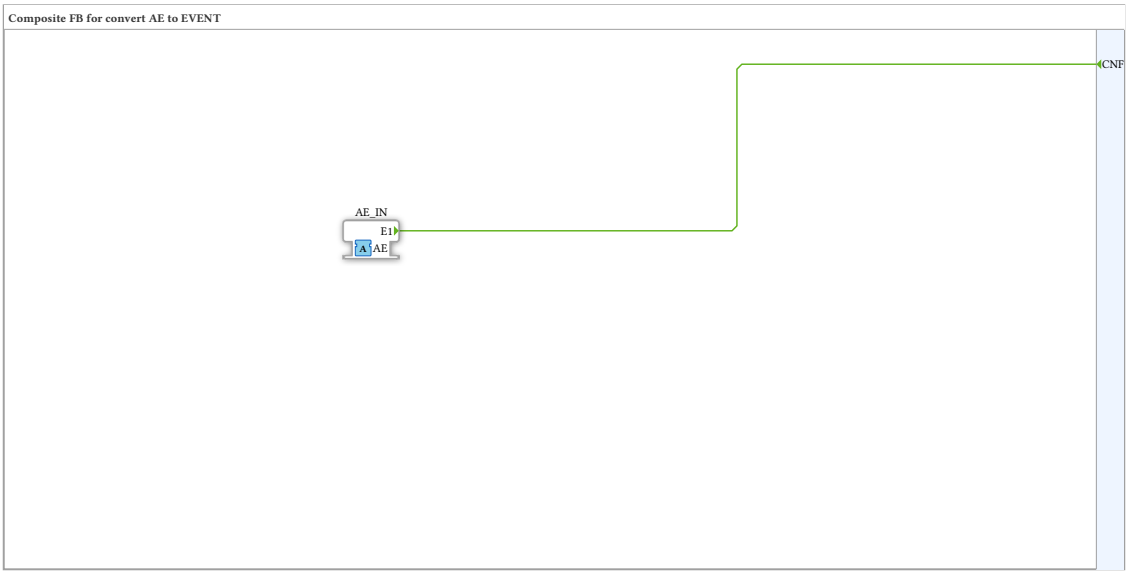
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Konvertierungsbausteinen zeichnet sich AE_EVENT_TO_E durch seine Einfachheit aus. Während andere Konverter oft Datenverarbeitung oder Protokollumsetzung durchführen, beschränkt sich dieser Block auf die reine Ereignisweiterleitung. Er ist speziell für die Integration in Adapter-basierte Systeme optimiert.

Fazit

AE_EVENT_TO_E ist ein spezialisierter Funktionsblock für die einfache Konvertierung von Ereignissen in AE-Adapter-Signale. Seine Stärke liegt in der minimalistischen Implementierung und der zuverlässigen Ereignisweiterleitung, was ihn ideal für Systemintegrationen macht, bei denen keine Datenverarbeitung erforderlich ist.

AE_E_TO_EVENT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AE_E_TO_EVENT ist ein Composite-Funktionsblock, der zur Konvertierung von Adapter-Ereignissen in Standard-Ereignisse dient. Er ermöglicht die Umwandlung von Adapter-Signalen in allgemein verwendbare Ereignisausgaben und dient als Brücke zwischen Adapter-Schnittstellen und standardisierten Event-basierten Kommunikationsstrukturen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereigniseingänge vorhanden

Ereignis-Ausgänge

- **CNF** - Bestätigungsereignis vom Adapter

Daten-Eingänge

Keine Dateneingänge vorhanden

Daten-Ausgänge

Keine Datenausgänge vorhanden

Adapter

- **AE_IN** - Adapter-Eingang vom Typ `adapter::types::unidirectional::AE`

Funktionsweise

Der Funktionsblock leitet das E1-Ereignis vom Adapter-Eingang AE_IN direkt an den CNF-Ausgang weiter. Es handelt sich um eine einfache Durchschleif-Funktionalität, bei der Adapter-Ereignisse in standardisierte Funktionsblock-Ereignisse umgewandelt werden.

Technische Besonderheiten

- Unidirektionale Signalverarbeitung
- Keine Datenverarbeitung oder -konvertierung
- Direkte Ereignisweiterleitung ohne Verzögerung
- Positionierung des CNF-Ausgangs bei Y-Position 46

Zustandsübersicht

Der Funktionsblock besitzt einen einfachen Zustand:

- **Bereit**: Wartet auf E1-Ereignis vom Adapter
- Bei Eintreffen von E1 wird sofort CNF ausgelöst

Anwendungsszenarien

- Integration von Adapter-Schnittstellen in bestehende Funktionsblock-Netzwerke
- Kopplung unterschiedlicher Kommunikationsprotokolle
- Erweiterung von Systemen mit speziellen Adapter-Anforderungen
- Protokollumsetzung in verteilten Automatisierungssystemen

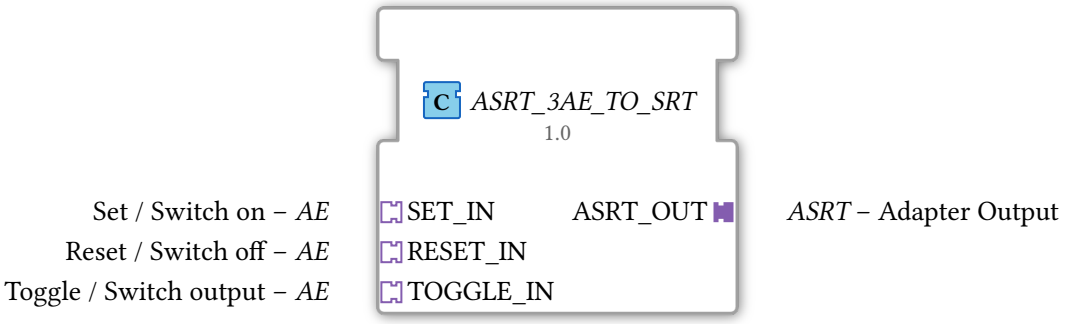
⚖️ Vergleich mit ähnlichen Bausteinen

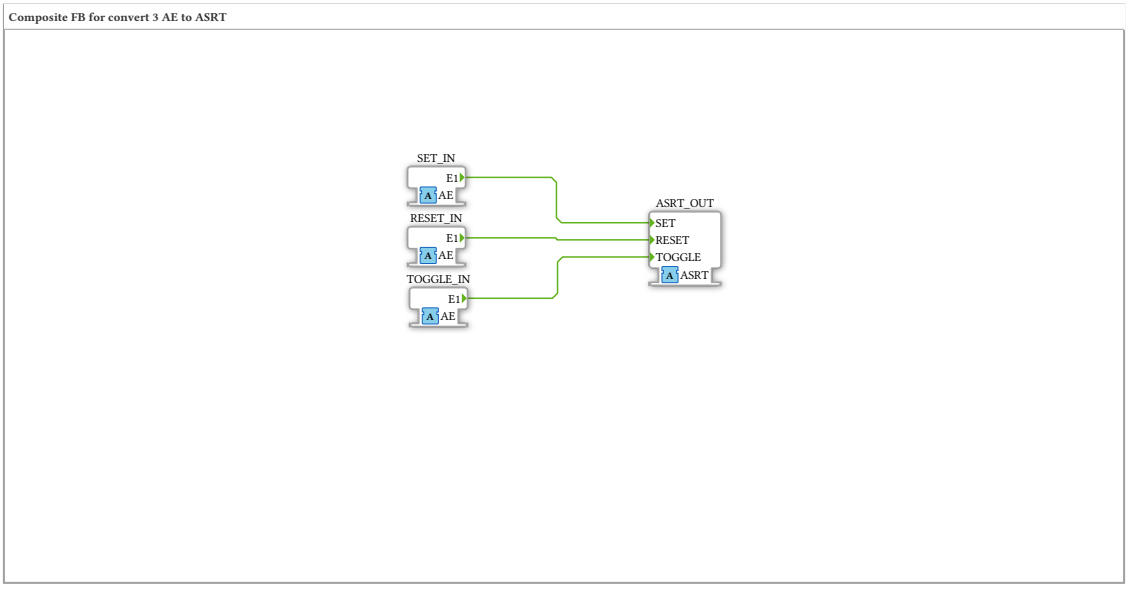
Im Vergleich zu anderen Konvertierungsbausteinen zeichnet sich AE_E_TO_EVENT durch seine Einfachheit aus. Während andere Konverter oft zusätzliche Datenverarbeitung oder Signalaufbereitung bieten, beschränkt sich dieser Baustein auf die reine Ereignisweiterleitung.

Fazit

Der AE_E_TO_EVENT Funktionsblock stellt eine schlanke und effiziente Lösung zur Adapter-Integration dar. Seine Einfachheit macht ihn zuverlässig und leicht verständlich, während er gleichzeitig die notwendige Brückenfunktion zwischen Adapter-Schnittstellen und standardisierten Event-Systemen erfüllt.

ASRT_3AE_TO_SRT





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ASRT_3AE_TO_SRT** ist ein Composite-FB, der drei separate unidirektionale AE-Adapter (reines Ereignis, keine Nutzdaten) zu einem gemeinsamen ASRT-Adapter (Set/Reset/Toggle-Ereignistripel) zusammenführt. Er ist die um `TOGGLE` erweiterte Variante von ASR_2AE_TO_SR und die adapterbasierte Entsprechung von ASRT_3EVENTS_TO_SRT.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignisübernahme erfolgt ausschließlich über die Adapter-Sockets.

Ereignis-Ausgänge

Keine. Die Ereignisweitergabe erfolgt ausschließlich über den Adapter-Plug.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket	`SET_IN`	`adapter::types::unidirectional::AE`	Set / Einschalten.

Socket	RESET_IN	adapter::types::unidirectional::AE	Reset / Ausschalten.
Socket	TOGGLE_IN	adapter::types::unidirectional::AE	Toggle / Ausgang umkehren.
Plug	ASRT_OUT	adapter::types::unidirectional::ASRT	Zusammengeführtes Set/Reset/Toggle-Signal.

Funktionsweise

Der Baustein ist ein reines FBNetwork aus drei Ereignisverbindungen ohne eigene Algorithmen: `SET_IN.E1` wird direkt auf `ASRT_OUT.SET` geführt, `RESET_IN.E1` auf `ASRT_OUT.RESET` und `TOGGLE_IN.E1` auf `ASRT_OUT.TOGGLE`. Ein Ereignis am jeweiligen AE-Socket löst unmittelbar das entsprechende Ereignis am ASRT-Plug aus.

Technische Besonderheiten

- **Reine Verdrahtung**: Composite-FB ohne ECC oder Algorithmen, ausschließlich drei direkte Ereignisverbindungen.
- **Adapterbasiert statt Event-basiert**: Im Gegensatz zu ASRT_3EVENTS_TO_SRT (klassische Ereigniseingänge) verwendet dieser Baustein AE-Adapter-Sockets, wodurch er sich nahtlos in adapterbasierte Netzwerke einfügt.

Zustandsübersicht

Der Baustein besitzt **keinen Zustandsautomaten**. Jedes Ereignis an `SET_IN`, `RESET_IN` oder `TOGGLE_IN` wird unmittelbar als gleichartiges Ereignis an `ASRT_OUT` weitergereicht.

Anwendungsszenarien

- **Zusammenführen dreier AE-Signalquellen** (z. B. aus ASRT_MERGE_2 oder direkt von Sensoren) zu einem einzigen ASRT-Adapter für nachgeschaltete Set/Reset/Toggle-Logik wie `AX_T_FF_SR`.
- **Adapterisierung** bestehender AE-basierter Netzwerke, um sie an ASRT-Schnittstellen anzubinden.

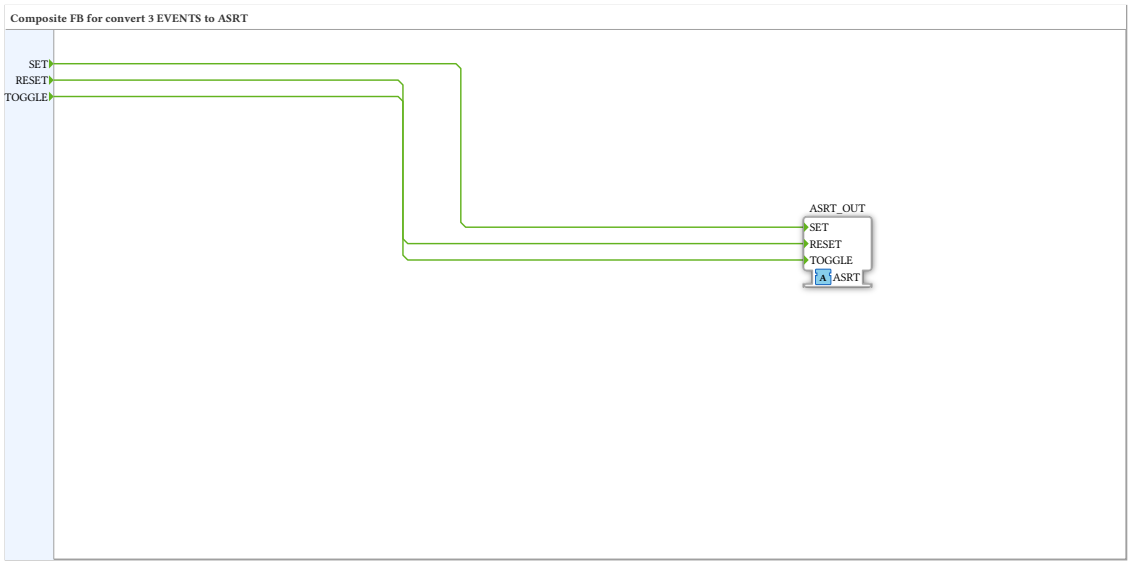
Vergleich mit ähnlichen Bausteinen

- **ASRT_SRT_TO_3AE**: die Umkehrrichtung – zerlegt ein ASRT-Signal wieder in drei AE-Adapter.
- **ASRT_3EVENTS_TO_SRT**: dieselbe Funktion mit klassischen Ereigniseingängen statt AE-Adapter-Sockets.
- **ASR_2AE_TO_SR**: die um `TOGGLE` reduzierte Variante für ASR.
- **ASRT_SR_AE_TO_SRT**: dieselbe Zusammenführung, jedoch mit `SET`/`RESET` bereits als fertigem ASR-Adapter statt zwei einzelnen AE-Sockets.

Fazit

`ASRT_3AE_TO_SRT` ist eine einfache, reine Verdrahtungslösung zur Zusammenführung dreier AE-Adapter zu einem ASRT-Adapter und eignet sich zur nahtlosen Integration von AE-basierten Ereignisquellen in ASRT-Schnittstellen.

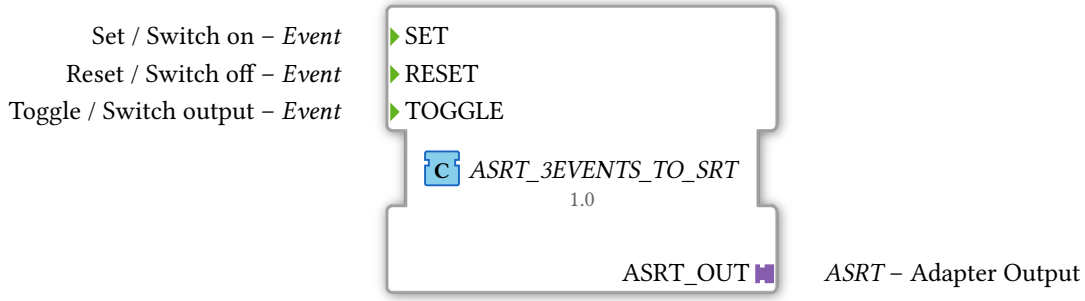
ASRT_3EVENTS_TO_SRT



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock ASRT_3EVENTS_TO_SRT ist ein Composite-Funktionsblock, der drei separate Ereignisse (Set/Reset/Toggle) in ein ASRT-Signal umwandelt. Er dient als Brücke zwischen ereignisbasierten Steuerungssignalen und dem ASRT-Adapterformat und ist die um TOGGLE erweiterte Variante von ASR_2EVENTS_TO_SR.



Schnittstellenstruktur

Ereignis-Eingänge

- **SET**: Setzt das Ausgangssignal (Einschalten)
- **RESET**: Rücksetzt das Ausgangssignal (Ausschalten)
- **TOGGLE**: Kehrt das Ausgangssignal um

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

- **ASRT_OUT**: Unidirektionaler ASRT-Adapter-Ausgang

Funktionsweise

Der Funktionsblock leitet die eingehenden SET-, RESET- und TOGGLE-Ereignisse direkt an den ASRT-Adapter weiter. Bei einem SET-Ereignis wird das entsprechende SET-Signal am ASRT-Adapter aktiviert, bei RESET das RESET-Signal und bei TOGGLE das TOGGLE-Signal.

Technische Besonderheiten

- Composite-Funktionsblock ohne interne Logikverarbeitung
- Direkte Durchleitung der Ereignisse an den Adapter
- Verwendet unidirektionale ASRT-Adapter-Technologie

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand. Er fungiert als reiner Signalweiterleiter zwischen den Ereigniseingängen und dem ASRT-Adapter.

Anwendungsszenarien

- Integration von ereignisbasierten Steuerungssystemen in ASRT-kompatible Umgebungen
- Verbindung zwischen verschiedenen Steuerungsebenen
- Protokollumsetzung in Automatisierungssystemen

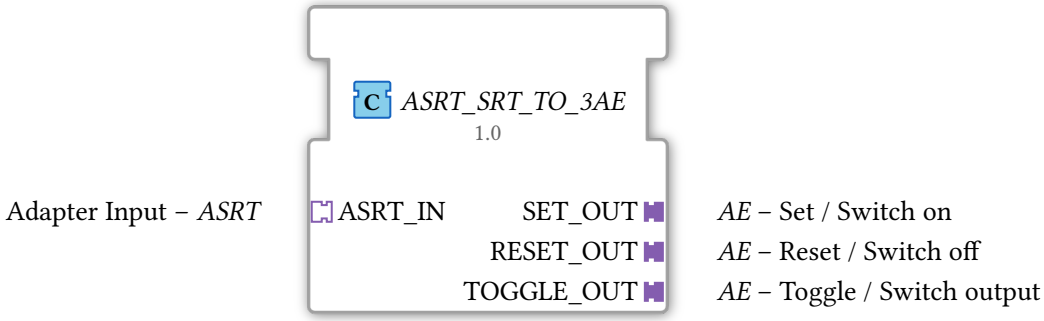
⚖️ Vergleich mit ähnlichen Bausteinen

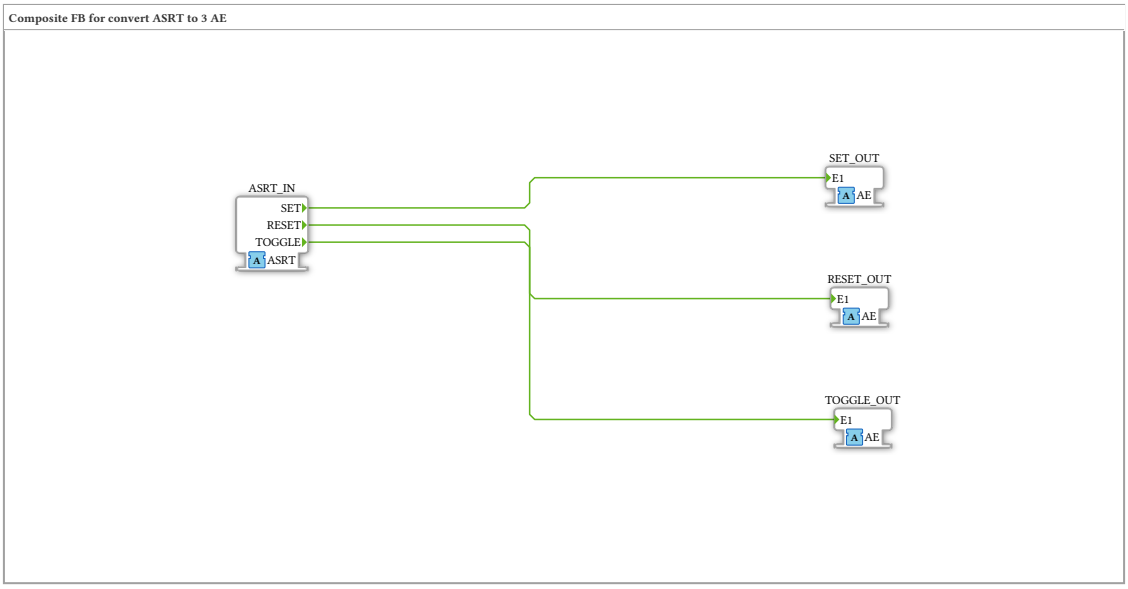
Im Vergleich zu ASR_2EVENTS_TO_SR bietet ASRT_3EVENTS_TO_SRT zusätzlich das TOGGLE-Ereignis und speist damit einen ASRT- statt einen ASR-Adapter.

Fazit

ASRT_3EVENTS_TO_SRT ist ein spezialisierter Funktionsblock für die Konvertierung von drei separaten Ereignissen (Set/Reset/Toggle) in das ASRT-Adapterformat. Seine einfache und direkte Funktionsweise macht ihn zu einer zuverlässigen Lösung für Systemintegrationen.

ASRT_SRT_TO_3AE





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ASRT_SRT_TO_3AE** ist ein Composite-FB, der einen unidirektionalen ASRT-Adapter (Set/Reset/Toggle-Ereignistripel) in drei separate AE-Adapter (reines Ereignis, keine Nutzdaten) zerlegt. Er ist die Umkehrung von ASRT_3AE_TO_SRT und die adapterbasierte Entsprechung von ASRT_SRT_TO_3EVENTS.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignisübernahme erfolgt ausschließlich über den Adapter-Socket.

Ereignis-Ausgänge

Keine. Die Ereignisweitergabe erfolgt ausschließlich über die Adapter-Plugs.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket	`ASRT_IN`	`adapter::types::unidirectional::ASRT`	Eingehendes Set/Reset/Toggle-Signal.
Plug	`SET_OUT`	`adapter::types::unidirectional::AE`	Set / Einschalten.
Plug	`RESET_OUT`	`adapter::types::unidirectional::AE`	Reset / Ausschalten.
Plug	`TOGGLE_OUT`	`adapter::types::unidirectional::AE`	Toggle / Ausgang umkehren.

Funktionsweise

Der Baustein ist ein reines FBNetwork aus drei Ereignisverbindungen ohne eigene Algorithmen: `ASRT_IN.SET` wird direkt auf `SET_OUT.E1` geführt, `ASRT_IN.RESET` auf `RESET_OUT.E1` und `ASRT_IN.TOGGLE` auf `TOGGLE_OUT.E1`. Ein SET-, RESET- oder TOGGLE-Ereignis am ASRT-Socket löst unmittelbar das entsprechende Ereignis am jeweiligen AE-Plug aus.

Technische Besonderheiten

- **Reine Verdrahtung**: Composite-FB ohne ECC oder Algorithmen, ausschließlich drei direkte Ereignisverbindungen.
- **Adapterbasiert statt Event-basiert**: Im Gegensatz zu ASRT_SRT_TO_3EVENTS (klassische Ereignisausgänge) liefert dieser Baustein AE-Adapter-Plugs, wodurch er sich nahtlos in adapterbasierte Netzwerke einfügt.

Zustandsübersicht

Der Baustein besitzt **keinen Zustandsautomaten**. Jedes `SET-`, `RESET-` oder `TOGGLE`-Ereignis an `ASRT_IN` wird unmittelbar an den jeweiligen AE-Plug weitergereicht.

Anwendungsszenarien

- **Auffächern eines ASRT-Signals** (z. B. von `AX_T_FF_SR`) in drei unabhängig weiterverarbeitbare AE-Ereignisse.
- **Adapterisierung** bestehender ASRT-basierter Netzwerke für Weiterverarbeitung mit reinen AE-Adaptern.

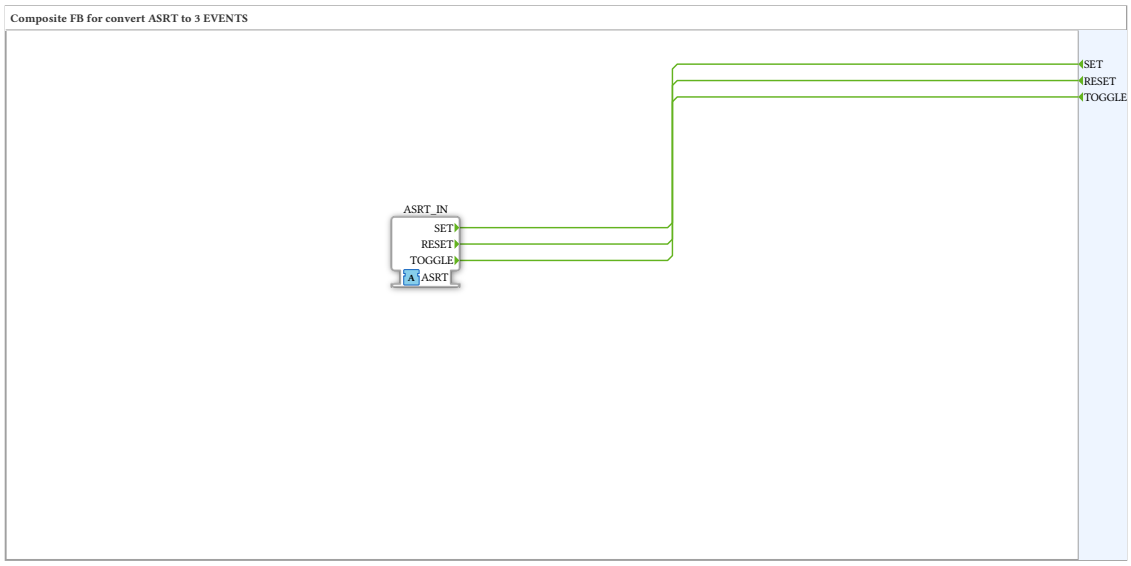
Vergleich mit ähnlichen Bausteinen

- **ASRT_3AE_TO_SRT**: die Umkehrrichtung – führt drei AE-Adapter zu einem ASRT-Signal zusammen.
- **ASRT_SRT_TO_3EVENTS**: dieselbe Funktion mit klassischen Ereignisausgängen statt AE-Adapter-Plugs.
- **ASR_SR_TO_2AE**: die um `TOGGLE` reduzierte Variante für ASR.
- **ASRT_SRT_TO_SR_AE**: dieselbe Zerlegung, jedoch mit `SET`/`RESET` als fertigem ASR-Adapter statt zwei einzelnen AE-Plugs.

Fazit

`ASRT_SRT_TO_3AE` ist eine einfache, reine Verdrahtungslösung zur Zerlegung eines ASRT-Adapters in drei AE-Adapter und eignet sich zur nahtlosen Integration von ASRT-Signalen in AE-basierte Netzwerke.

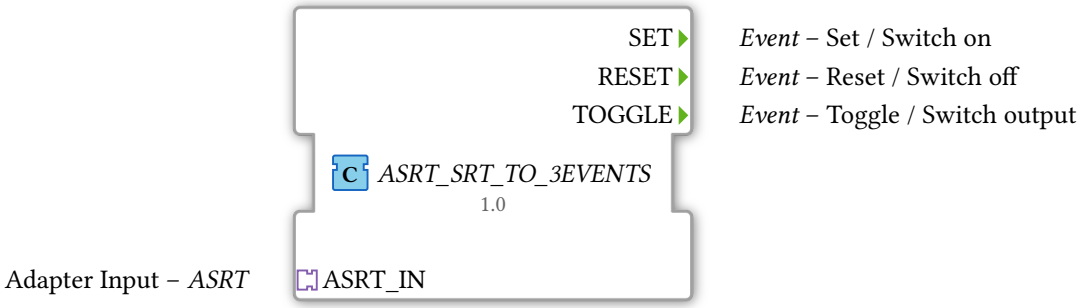
ASRT_SRT_TO_3EVENTS



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock ASRT_SRT_TO_3EVENTS ist ein Composite-Funktionsblock, der zur Konvertierung eines ASRT-Signals in drei separate Ereignisse dient. Er ermöglicht die Umwandlung eines unidirektionalen Adapter-Signals in unabhängige SET-, RESET- und TOGGLE-Ereignisse und ist die um TOGGLE erweiterte Variante von ASR_SR_TO_2EVENTS.



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereigniseingänge vorhanden.

Ereignis-Ausgänge

- **SET**: Setzen / Einschalten
- **RESET**: Rücksetzen / Ausschalten
- **TOGGLE**: Umkehren des Ausgangssignals

Daten-Eingänge

Keine Dateneingänge vorhanden.

Daten-Ausgänge

Keine Datenausgänge vorhanden.

Adapter

- **ASRT_IN**: Adapter-Eingang vom Typ `adapter::types::unidirectional::ASRT`

Funktionsweise

Der Funktionsblock leitet die SET-, RESET- und TOGGLE-Signale des ASRT-Adapters direkt an die entsprechenden Ereignisausgänge weiter. Dabei werden die eingehenden Adapterereignisse transparent an die Ausgänge des Funktionsblocks durchgeschleift.

Technische Besonderheiten

- Composite-Funktionsblock ohne interne Logikverarbeitung
- Direkte Verbindung zwischen Adapterereignissen und Ausgangsereignissen
- Keine Datenverarbeitung oder Zustandsverwaltung

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand und arbeitet stateless. Jedes eingehende Adapterereignis wird sofort an den entsprechenden Ausgang weitergeleitet.

Anwendungsszenarien

- Integration von ASRT-Adaptern in Ereignis-basierte Steuerungen
- Umwandlung unidirektionaler Adapter-Signale in unabhängige Einzelereignisse
- Schnittstellenanpassung zwischen verschiedenen Kommunikationsprotokollen

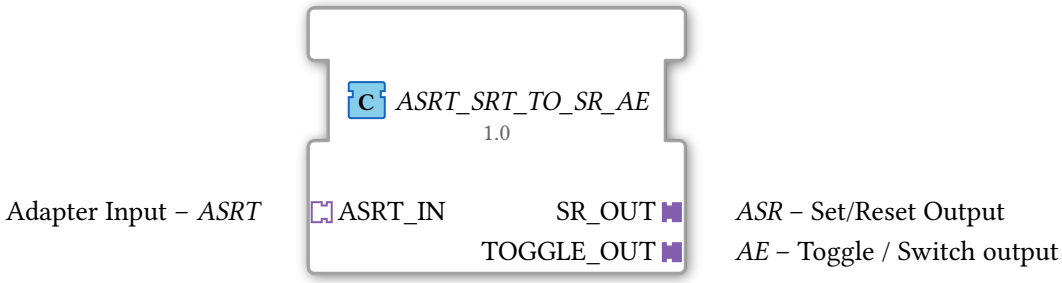
⚖️ Vergleich mit ähnlichen Bausteinen

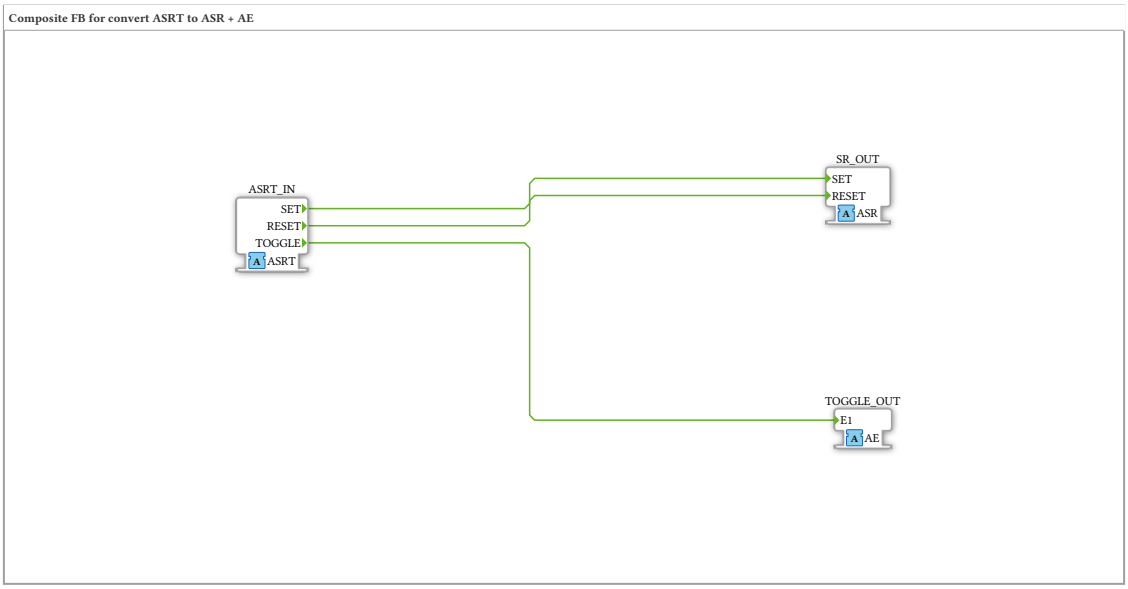
Im Vergleich zu ASR_SR_TO_2EVENTS verarbeitet dieser Composite-Funktionsblock zusätzlich das TOGGLE-Signal eines ASRT- statt eines ASR-Adapters.

Fazit

Der ASRT_SRT_TO_3EVENTS Funktionsblock stellt eine einfache und effiziente Lösung zur Konvertierung von ASRT-Adaptersignalen in separate SET-, RESET- und TOGGLE-Ereignisse dar. Seine schlanke Implementierung ohne interne Logik macht ihn zu einer zuverlässigen und performanten Komponente für Schnittstellenanpassungen.

ASRT_SRT_TO_SR_AE





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASRT_SRT_TO_SR_AE** ist ein Composite-FB, der einen unidirektionalen ASRT-Adapter (Set/Reset/Toggle) in einen ASR-Adapter (Set/Reset) und einen separaten AE-Adapter (Toggle, reines Ereignis) zerlegt. Er ist die Umkehrung von ASRT_SR_AE_TO_SRT und eine gemischte Variante von ASRT_SRT_TO_3AE.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignisübernahme erfolgt ausschließlich über den Adapter-Socket.

Ereignis-Ausgänge

Keine. Die Ereignisweitergabe erfolgt ausschließlich über die Adapter-Plugs.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket	`ASRT_IN`	`adapter::types::unidirectional::ASRT`	Eingehendes Set/Reset/Toggle-Signal.

Plug	SR_OUT	adapter::types::unidirectional::ASR	Set/Reset-Ausgang.
Plug	TOGGLE_OUT	adapter::types::unidirectional::AE	Toggle / Ausgang umkehren.

Funktionsweise

Der Baustein ist ein reines FBNetwork aus drei Ereignisverbindungen ohne eigene Algorithmen: `ASRT_IN.SET` wird direkt auf `SR_OUT.SET` geführt, `ASRT_IN.RESET` auf `SR_OUT.RESET` und `ASRT_IN.TOGGLE` auf `TOGGLE_OUT.E1`. Ein SET-, RESET- oder TOGGLE-Ereignis am ASRT-Socket löst unmittelbar das entsprechende Ereignis an `SR_OUT` bzw. `TOGGLE_OUT` aus.

Technische Besonderheiten

- **Reine Verdrahtung**: Composite-FB ohne ECC oder Algorithmen, ausschließlich drei direkte Ereignisverbindungen.
- **Gemischte Adapter-Ausgänge**: Liefert SET/RESET bereits als fertigen ASR-Adapter statt zweier einzelner AE-Plugs — sinnvoll, wenn die Folgelogik ohnehin einen ASR-Eingang erwartet (z. B. `ASR_AX_SR`) und nur TOGGLE separat behandelt werden soll.

Zustandsübersicht

Der Baustein besitzt **keinen Zustandsautomaten**. Jedes `SET`-, `RESET`- oder `TOGGLE`-Ereignis an `ASRT_IN` wird unmittelbar an `SR_OUT` bzw. `TOGGLE_OUT` weitergereicht.

Anwendungsszenarien

- **Anbindung an bestehende ASR-Schnittstellen**: Ein ASRT-Signal soll an einen Baustein weitergegeben werden, der nur SET/RESET als ASR-Adapter erwartet, während TOGGLE separat (z. B. für Diagnosezwecke) abgegriffen wird.
- **Vereinfachung nachgeschalteter Logik**, die Set/Reset ohnehin gebündelt als ASR-Adapter verarbeitet.

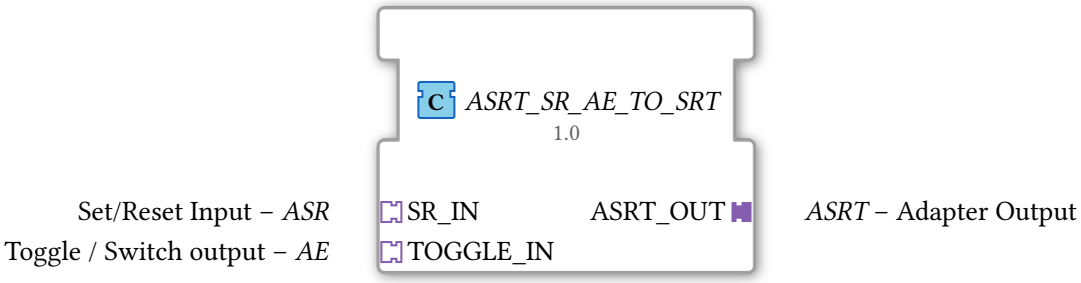
Vergleich mit ähnlichen Bausteinen

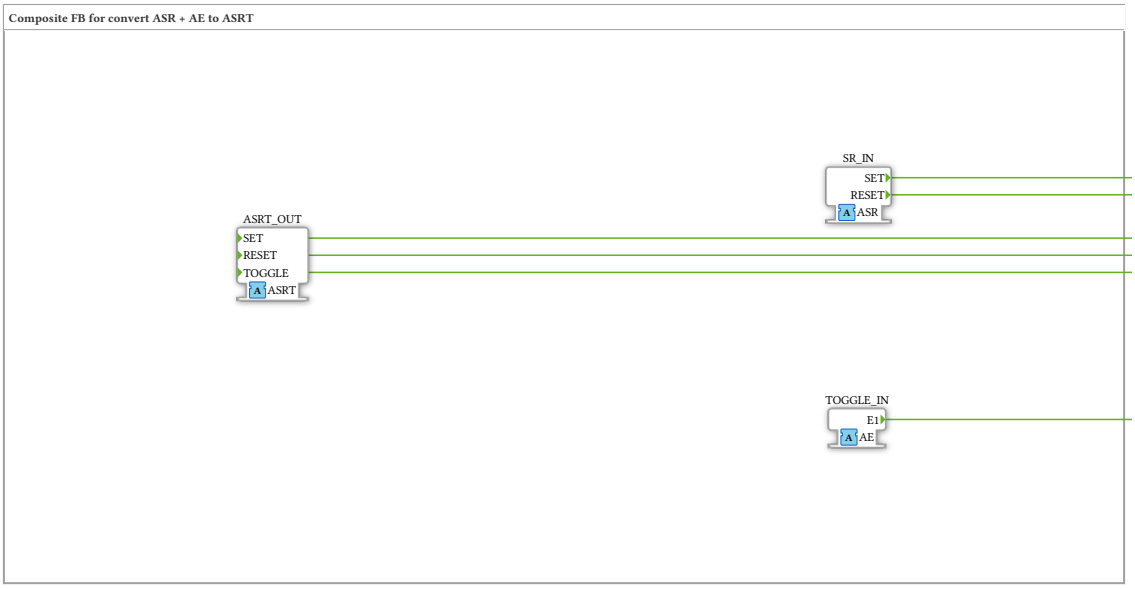
- ASRT_SR_AE_TO_SRT: die Umkehrrichtung – führt ASR (SET/RESET) und AE (TOGGLE) zu einem ASRT-Signal zusammen.
- ASRT_SRT_TO_3AE: dieselbe Zerlegung, jedoch mit drei einzelnen AE-Plugs statt einem fertigen ASR-Adapter für SET/RESET.
- ASR_SR_TO_2AE: die um `TOGGLE` reduzierte Variante für ASR.

Fazit

`ASRT_SRT_TO_SR_AE` ist eine einfache, reine Verdrahtungslösung, die ein ASRT-Signal in einen ASR-Adapter (SET/RESET) und ein separates Toggle-Ereignis auftrennt.

ASRT_SR_AE_TO_SRT





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ASRT_SR_AE_TO_SRT** ist ein Composite-FB, der einen unidirektionalen ASR-Adapter (Set/Reset) und einen separaten AE-Adapter (Toggle, reines Ereignis) zu einem gemeinsamen ASRT-Adapter (Set/Reset/Toggle) zusammenführt. Er ist eine gemischte Variante von ASRT_3AE_TO_SRT: Statt drei einzelner AE-Sockets für SET, RESET und TOGGLE nimmt er SET/RESET bereits als fertigen ASR-Adapter entgegen und ergänzt nur TOGGLE separat.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignisübernahme erfolgt ausschließlich über die Adapter-Sockets.

Ereignis-Ausgänge

Keine. Die Ereignisweitergabe erfolgt ausschließlich über den Adapter-Plug.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket	`SR_IN`	`adapter::types::unidirectional::ASR`	Eingehendes Set/Reset-Signal.
Socket	`TOGGLE_IN`	`adapter::types::unidirectional::AE`	Toggle / Ausgang umkehren.
Plug	`ASRT_OUT`	`adapter::types::unidirectional::ASRT`	Zusammengeführtes Set/Reset/Toggle-Signal.

Funktionsweise

Der Baustein ist ein reines FBNetwork aus drei Ereignisverbindungen ohne eigene Algorithmen: `SR_IN.SET` wird direkt auf `ASRT_OUT.SET` geführt, `SR_IN.RESET` auf `ASRT_OUT.RESET` und `TOGGLE_IN.E1` auf `ASRT_OUT.TOGGLE`. Ein SET- oder RESET-Ereignis am ASR-Socket bzw. ein Ereignis am AE-Socket löst unmittelbar das entsprechende Ereignis am ASRT-Plug aus.

Technische Besonderheiten

- **Reine Verdrahtung**: Composite-FB ohne ECC oder Algorithmen, ausschließlich drei direkte Ereignisverbindungen.
- **Gemischte Adapter-Eingänge**: Kombiniert einen bereits fertigen ASR-Adapter (SET/RESET) mit einem separaten AE-Adapter (TOGGLE) — sinnvoll, wenn SET/RESET bereits als ASR-Signal vorliegen (z. B. aus ASR_2AE_TO_SR oder einem bestehenden ASR-Netzwerkzweig) und nur TOGGLE zusätzlich ergänzt werden muss.

Zustandsübersicht

Der Baustein besitzt **keinen Zustandsautomaten**. Jedes Ereignis an `SR_IN` (SET/RESET) oder `TOGGLE_IN` wird unmittelbar als gleichartiges Ereignis an `ASRT_OUT` weitergereicht.

Anwendungsszenarien

- **Nachträgliches Ergänzen von TOGGLE**: Ein bestehendes ASR-Netzwerk soll um eine Toggle-Funktion erweitert werden, ohne SET/RESET neu zu verdrahten.
- **Wiederverwendung vorhandener ASR-Signalpfade** in Kombination mit einer separaten Toggle-Quelle, z. B. einem Taster über ASR_MERGE_2.

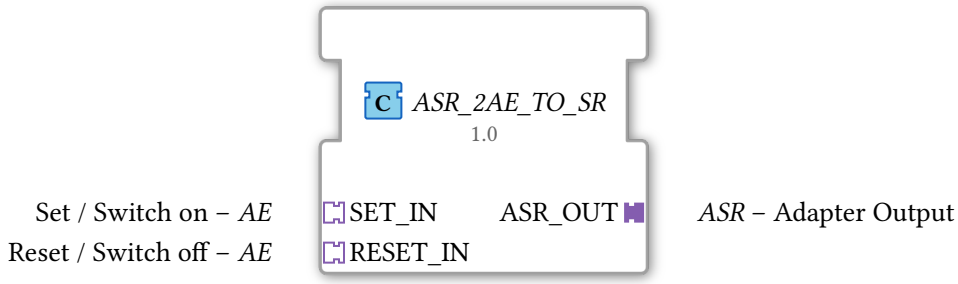
Vergleich mit ähnlichen Bausteinen

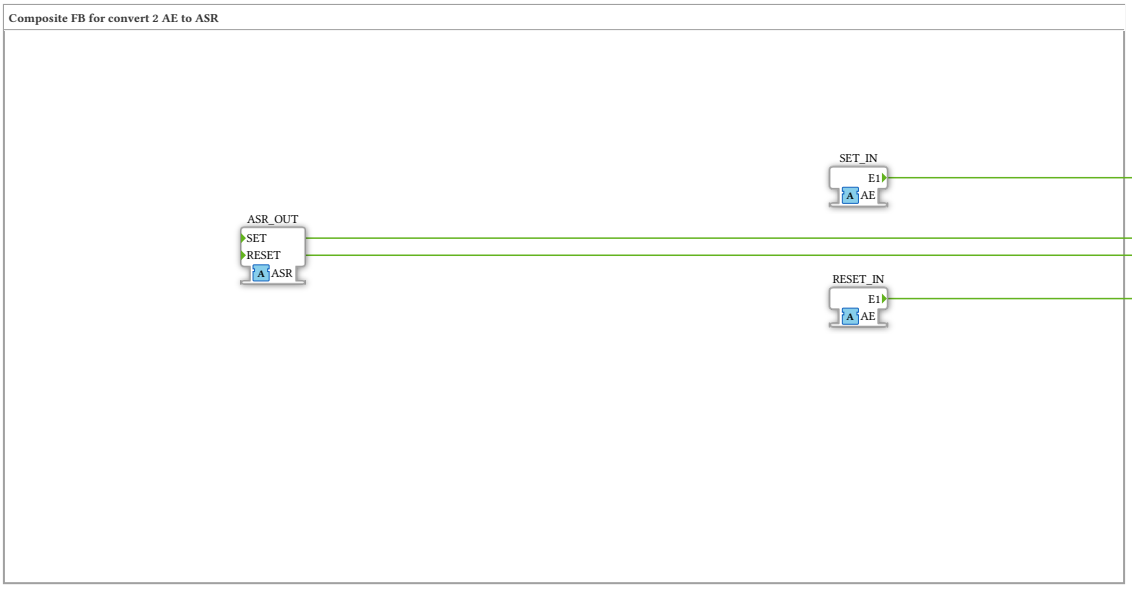
- **ASRT_SRT_TO_SR_AE**: die Umkehrrichtung – zerlegt ein ASRT-Signal wieder in ASR (SET/RESET) und AE (TOGGLE).
- **ASRT_3AE_TO_SRT**: dieselbe Zusammenführung, jedoch mit drei einzelnen AE-Sockets statt einem fertigen ASR-Adapter für SET/RESET.
- **ASR_2AE_TO_SR**: die um `TOGGLE` reduzierte Variante für ASR.

Fazit

`ASRT_SR_AE_TO_SRT` ist eine einfache, reine Verdrahtungslösung, die einen bestehenden ASR-Adapter um ein separates Toggle-Ereignis zu einem vollständigen ASRT-Adapter ergänzt.

ASR_2AE_TO_SR





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ASR_2AE_TO_SR** ist ein Composite-FB, der zwei separate unidirektionale AE-Adapter (reines Ereignis, keine Nutzdaten) zu einem gemeinsamen ASR-Adapter (Set/Reset-Ereignispaar) zusammenführt. Er ist die adapterbasierte Entsprechung von ASR_2EVENTS_TO_SR, verwendet jedoch AE-Adapter-Sockets statt klassischer Ereigniseingänge.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignisübernahme erfolgt ausschließlich über die Adapter-Sockets.

Ereignis-Ausgänge

Keine. Die Ereignisweitergabe erfolgt ausschließlich über den Adapter-Plug.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket	`SET_IN`	`adapter::types::unidirectional::AE`	Set / Einschalten.
Socket	`RESET_IN`	`adapter::types::unidirectional::AE`	Reset / Ausschalten.
Plug	`ASR_OUT`	`adapter::types::unidirectional::ASR`	Zusammengeführtes Set/Reset-Signal.

Funktionsweise

Der Baustein ist ein reines FBNetwork aus zwei Ereignisverbindungen ohne eigene Algorithmen: `SET_IN.E1` wird direkt auf `ASR_OUT.SET` geführt, `RESET_IN.E1` direkt auf `ASR_OUT.RESET`. Ein Ereignis am jeweiligen AE-Socket löst unmittelbar das entsprechende SET- bzw. RESET-Ereignis am ASR-Plug aus.

Technische Besonderheiten

- **Reine Verdrahtung**: Composite-FB ohne ECC oder Algorithmen, ausschließlich zwei direkte Ereignisverbindungen.

- **Adapterbasiert statt Event-basiert**: Im Gegensatz zu ASR_2EVENTS_TO_SR (klassische Ereigniseingänge SET/RESET) verwendet dieser Baustein AE-Adapter-Sockets, wodurch er sich nahtlos in adapterbasierte Netzwerke einfügt.

Zustandsübersicht

Der Baustein besitzt **keinen Zustandsautomaten**. Jedes Ereignis an SET_IN oder RESET_IN wird unmittelbar als SET bzw. RESET an ASR_OUT weitergereicht.

Anwendungsszenarien

- **Zusammenführen zweier AE-Signalquellen** (z. B. aus ASR_MERGE_2 oder direkt von einem Sensor) zu einem einzigen ASR-Adapter für nachgeschaltete Set/Reset-Logik.
- **Adapterisierung** bestehender AE-basierter Netzwerke, um sie an ASR-Schnittstellen (z. B. AX_SR) anzubinden.

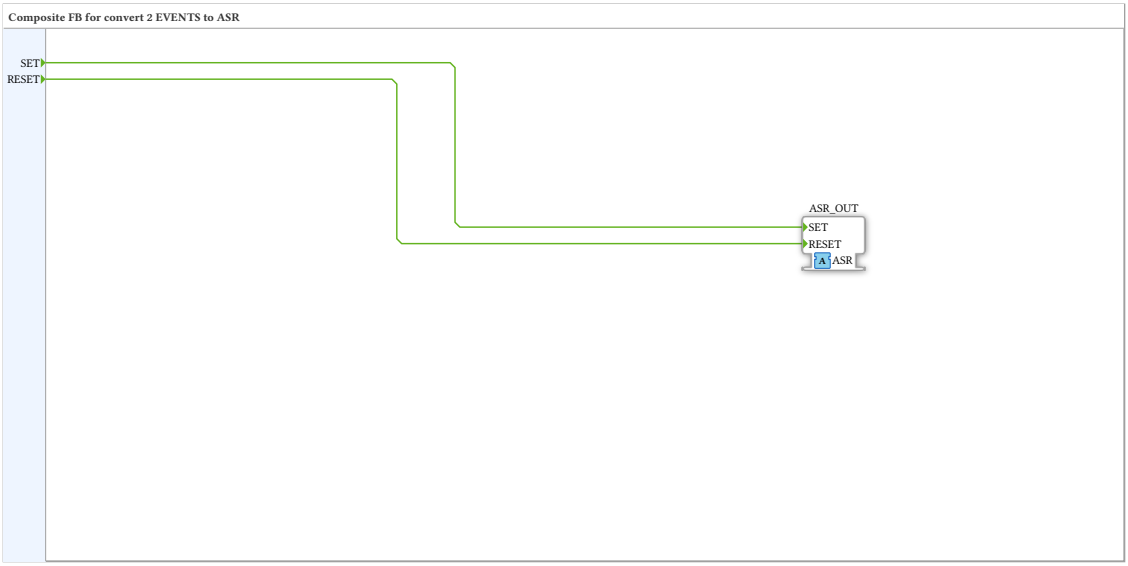
Vergleich mit ähnlichen Bausteinen

- **ASR_SR_TO_2AE**: die Umkehrrichtung – zerlegt ein ASR-Signal wieder in zwei AE-Adapter.
- **ASR_2EVENTS_TO_SR**: dieselbe Funktion mit klassischen Ereigniseingängen statt AE-Adapter-Sockets.
- **ASRT_3AE_TO_SRT**: die um TOGGLE erweiterte Variante für ASRT.

Fazit

ASR_2AE_TO_SR ist eine einfache, reine Verdrahtungslösung zur Zusammenführung zweier AE-Adapter zu einem ASR-Adapter und eignet sich zur nahtlosen Integration von AE-basierten Ereignisquellen in ASR-Schnittstellen.

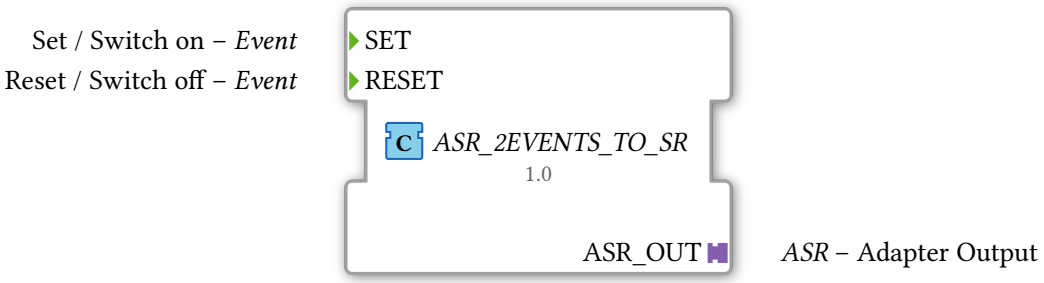
ASR_2EVENTS_TO_SR



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock ASR_2EVENTS_TO_SR ist ein Composite-Funktionsblock, der zwei separate Ereignisse in ein ASR-Signal (Set-Reset) umwandelt. Er dient als Brücke zwischen ereignisbasierten Steuerungssignalen und dem ASR-Adapterformat.



Schnittstellenstruktur

Ereignis-Eingänge

- **SET**: Setzt das Ausgangssignal (Einschalten)
- **RESET**: Rücksetzt das Ausgangssignal (Ausschalten)

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

- **ASR_OUT**: Unidirektionaler ASR-Adapter-Ausgang

Funktionsweise

Der Funktionsblock leitet die eingehenden SET- und RESET-Ereignisse direkt an den ASR-Adapter weiter. Bei einem SET-Ereignis wird das entsprechende SET-Signal am ASR-Adapter aktiviert, bei einem RESET-Ereignis das RESET-Signal.

Technische Besonderheiten

- Composite-Funktionsblock ohne interne Logikverarbeitung
- Direkte Durchleitung der Ereignisse an den Adapter
- Verwendet unidirektionale ASR-Adapter-Technologie

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand. Er fungiert als reiner Signalweiterleiter zwischen den Ereigniseingängen und dem ASR-Adapter.

Anwendungsszenarien

- Integration von ereignisbasierten Steuerungssystemen in ASR-kompatible Umgebungen
- Verbindung zwischen verschiedenen Steuerungsebenen
- Protokollumsetzung in Automatisierungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen Ereignis-Funktionsblöcken bietet ASR_2EVENTS_TO_SR eine spezifische Schnittstelle zum ASR-Adapter, während andere Blöcke möglicherweise direkte Datenausgänge verwenden.

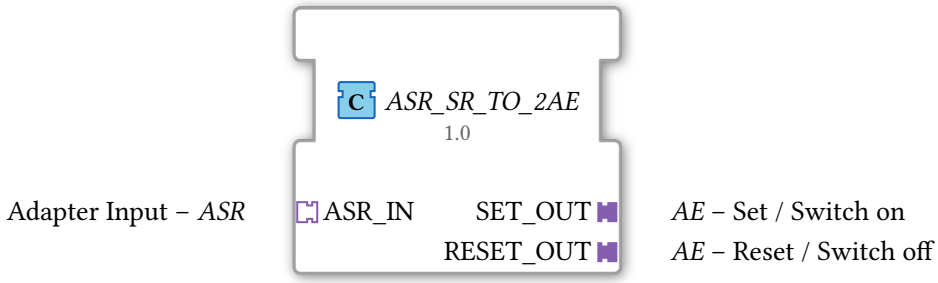
🛠️ Zugehörige Übungen

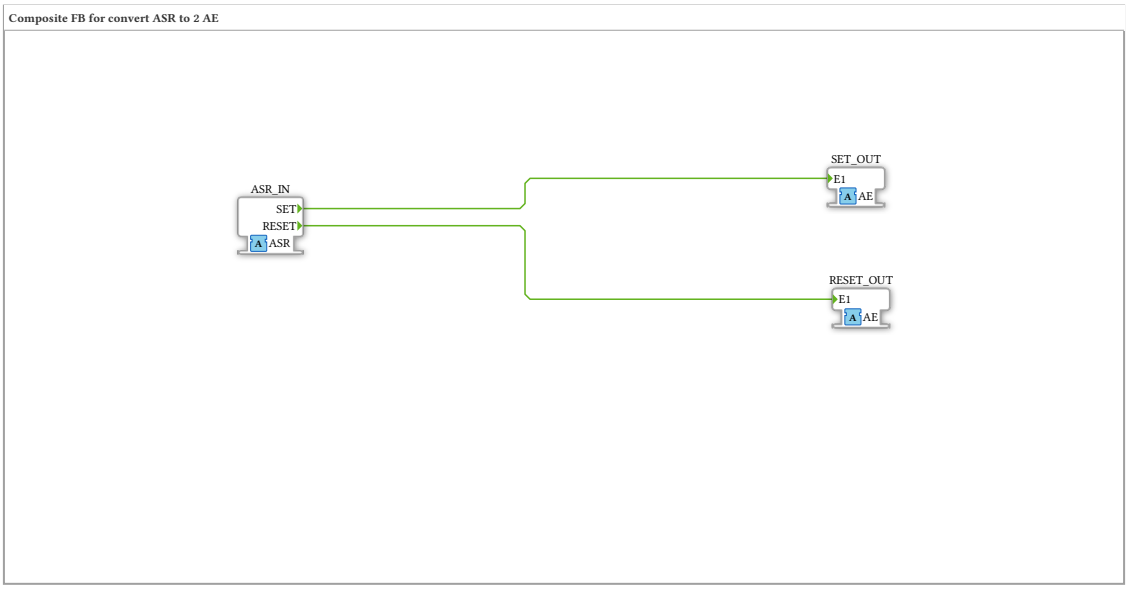
- Uebung_171_AX

Fazit

ASR_2EVENTS_TO_SR ist ein spezialisierter Funktionsblock für die Konvertierung von zwei separaten Ereignissen in das ASR-Adapterformat. Seine einfache und direkte Funktionsweise macht ihn zu einer zuverlässigen Lösung für Systemintegrationen.

ASR_SR_TO_2AE





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ASR_SR_TO_2AE** ist ein Composite-FB, der einen unidirektionalen ASR-Adapter (Set/Reset-Ereignispaar) in zwei separate AE-Adapter (reines Ereignis, keine Nutzdaten) zerlegt. Er ist die Umkehrung von ASR_2AE_TO_SR und die adapterbasierte Entsprechung von ASR_SR_TO_2EVENTS.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignisübernahme erfolgt ausschließlich über den Adapter-Socket.

Ereignis-Ausgänge

Keine. Die Ereignisweitergabe erfolgt ausschließlich über die Adapter-Plugs.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket	ASR_IN	adapter::types::unidirectional::ASR	Eingehendes Set/Reset-Signal.
Plug	SET_OUT	adapter::types::unidirectional::AE	Set / Einschalten.
Plug	RESET_OUT	adapter::types::unidirectional::AE	Reset / Ausschalten.

Funktionsweise

Der Baustein ist ein reines FBNetwork aus zwei Ereignisverbindungen ohne eigene Algorithmen: ASR_IN.SET wird direkt auf SET_OUT.E1 geführt, ASR_IN.RESET direkt auf RESET_OUT.E1. Ein SET- bzw. RESET-Ereignis am ASR-Socket löst unmittelbar das entsprechende Ereignis am jeweiligen AE-Plug aus.

Technische Besonderheiten

- **Reine Verdrahtung**: Composite-FB ohne ECC oder Algorithmen, ausschließlich zwei direkte Ereignisverbindungen.
- **Adapterbasiert statt Event-basiert**: Im Gegensatz zu ASR_SR_TO_2EVENTS (klassische Ereignisausgänge) liefert dieser Baustein AE-Adapter-Plugs, wodurch er sich nahtlos in adapterbasierte Netzwerke einfügt.

Zustandsübersicht

Der Baustein besitzt **keinen Zustandsautomaten**. Jedes SET- oder RESET-Ereignis an ASR_IN wird unmittelbar an SET_OUT bzw. RESET_OUT weitergereicht.

Anwendungsszenarien

- **Auffächern eines ASR-Signals** in zwei unabhängig weiterverarbeitbare AE-Ereignisse, z. B. um SET und RESET getrennt an unterschiedliche Folgebausteine zu leiten.
- **Adapterisierung** bestehender ASR-basierter Netzwerke für Weiterverarbeitung mit reinen AE-Adaptern.

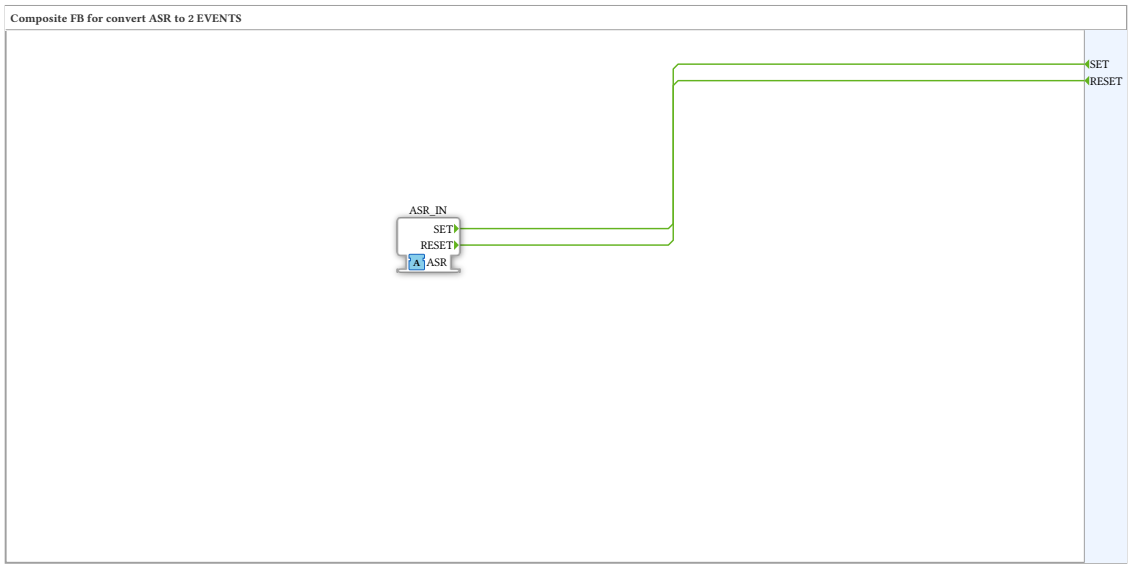
Vergleich mit ähnlichen Bausteinen

- **ASR_2AE_TO_SR**: die Umkehrrichtung – führt zwei AE-Adapter zu einem ASR-Signal zusammen.
- **ASR_SR_TO_2EVENTS**: dieselbe Funktion mit klassischen Ereignisausgängen statt AE-Adapter-Plugs.
- **ASRT_SRT_TO_3AE**: die um TOGGLE erweiterte Variante für ASRT.

Fazit

ASR_SR_TO_2AE ist eine einfache, reine Verdrahtungslösung zur Zerlegung eines ASR-Adapters in zwei AE-Adapter und eignet sich zur nahtlosen Integration von ASR-Signalen in AE-basierte Netzwerke.

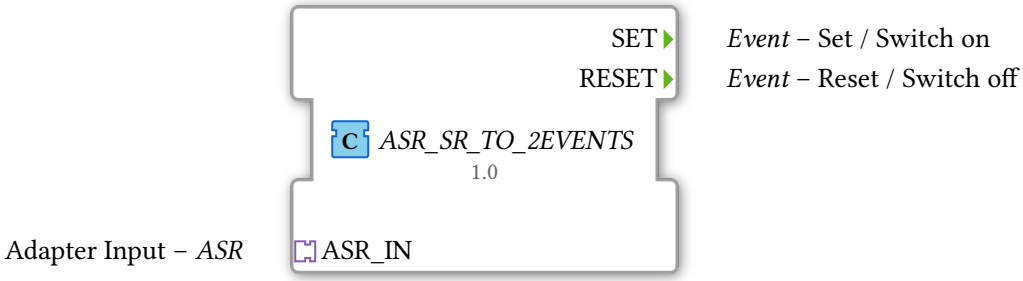
ASR_SR_TO_2EVENTS



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock ASR_SR_TO_2EVENTS ist ein Composite-Funktionsblock, der zur Konvertierung eines ASR-Signals in zwei separate Ereignisse dient. Er ermöglicht die Umwandlung eines bidirektionalen Adapter-Signals in unabhängige SET- und RESET-Ereignisse.



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereigniseingänge vorhanden.

Ereignis-Ausgänge

- **SET**: Setzen / Einschalten
- **RESET**: Rücksetzen / Ausschalten

Daten-Eingänge

Keine Dateneingänge vorhanden.

Daten-Ausgänge

Keine Datenausgänge vorhanden.

Adapter

- **ASR_IN**: Adapter-Eingang vom Typ `adapter::types::unidirectional::ASR`

Funktionsweise

Der Funktionsblock leitet die SET- und RESET-Signale des ASR-Adapters direkt an die entsprechenden Ereignisausgänge weiter. Dabei werden die eingehenden Adapterereignisse transparent an die Ausgänge des Funktionsblocks durchgeschleift.

Technische Besonderheiten

- Composite-Funktionsblock ohne interne Logikverarbeitung
- Direkte Verbindung zwischen Adapterereignissen und Ausgangsereignissen
- Keine Datenverarbeitung oder Zustandsverwaltung

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand und arbeitet stateless. Jedes eingehende Adapterereignis wird sofort an den entsprechenden Ausgang weitergeleitet.

Anwendungsszenarien

- Integration von ASR-Adaptern in Ereignis-basierte Steuerungen
- Umwandlung bidirektionaler Signale in unabhängige Einzelereignisse
- Schnittstellenanpassung zwischen verschiedenen Kommunikationsprotokollen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen Adapter-Blöcken bietet dieser Composite-Funktionsblock eine spezifische Konvertierung von ASR- zu Ereignissignalen, während generische Adapter eine allgemeinere Schnittstellenanpassung ermöglichen.

Fazit

Der ASR_SR_TO_2EVENTS Funktionsblock stellt eine einfache und effiziente Lösung zur Konvertierung von ASR-Adaptersignalen in separate SET- und RESET-Ereignisse dar. Seine schlanke Implementierung ohne interne Logik macht ihn zu einer zuverlässigen und performanten Komponente für Schnittstellenanpassungen.

Unidirectional EVENT Adapter Conversion

🎧 Podcast

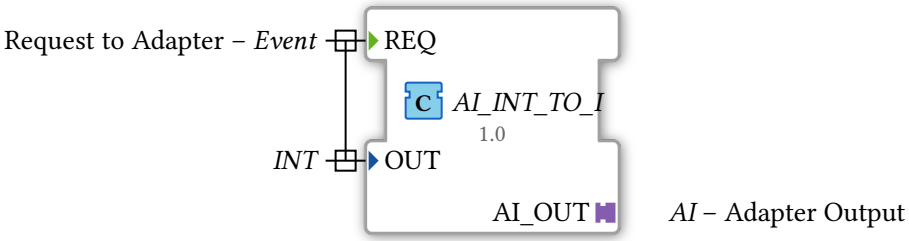
- E_TOF vs. FB_TOF: Der Event-Timer, der nicht zyklisch tickt – Revolution für Automatisierungssysteme?
- EventFBs nach IEC 61499: Legosteine der Automatisierung – So funktionieren Ereignis-Funktionsbausteine
- IEC 61499: Revolution der Automatisierung – Event-gesteuerte FBs und verteilte Systeme erklärt
- E_REND: Event Synchronization in IEC 61499
- The Unstoppable Counter: Why IEC 61499's ECTU Guarantees Safe, Event-Driven Control (and Never Overflows)

INT

AI_INT_TO_I

Einleitung

Der AI_INT_TO_I Funktionsblock ist ein Composite-Funktionsblock, der zur Konvertierung eines INT-Werts in das AI-Adapterformat dient.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Anfrage-Event zur Auslösung der Konvertierung

Ereignis-Ausgänge

- Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

- **OUT**: INT-Eingangswert, der konvertiert werden soll

Daten-Ausgänge

- Keine direkten Daten-Ausgänge vorhanden

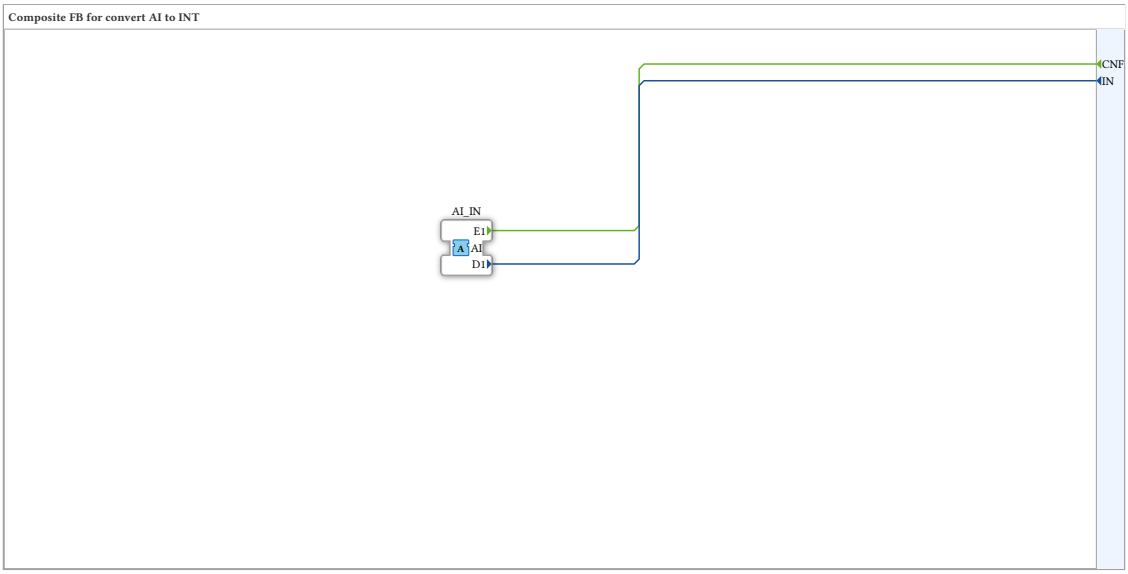
Adapter

- **AI_OUT**: Plug vom Typ `AI`

Funktionsweise

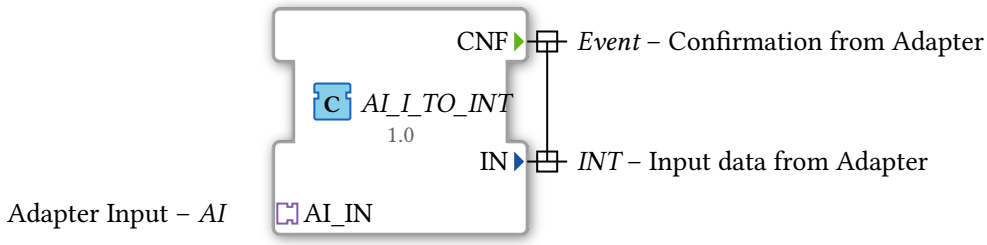
Der Funktionsblock empfängt einen INT-Wert am Daten-Eingang OUT und wandelt diesen bei Eintreffen des REQ-Ereignisses in das AI-Adapterformat um.

AI_I_TO_INT



Einleitung

Der AI_I_TO_INT Funktionsblock ist ein Composite-Funktionsblock, der zur Konvertierung des AI-Adapterformats in einen INT-Wert dient.



Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkten Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

- **CNF**: Bestätigungs-Event nach erfolgter Konvertierung

Daten-Eingänge

- Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

- **IN**: INT-Ausgangswert der konvertierten Daten

Adapter

- **AI_IN**: Socket vom Typ `AI`

Funktionsweise

Der Funktionsblock empfängt Daten über den AI-Adapter-Socket und stellt den enthaltenen INT-Wert am Ausgang IN zur Verfügung, sobald ein Ereignis über den Adapter eintrifft (CNF).

INT Conversion

🎧 Podcast

- Eclipse 4diac 3.0: ST-Interpreter, FBE und 7200 Commits – Der Turbo für verteilte Automatisierung

- Von der SPS zur verteilten Intelligenz: IEC 61499 für Industrie 4.0 verstehen
- Simplifying Industrial Control: Your Deep Dive into Eclipse 4diac and IEC 61499
- Ereignisse und IEC 61499: Der Startschuss für intelligente Systeme
- ERP-Systeme: Das digitale Gehirn – Chancen, Risiken und der Weg zur intelligenten Produktion

🛠️ Zugehörige Übungen

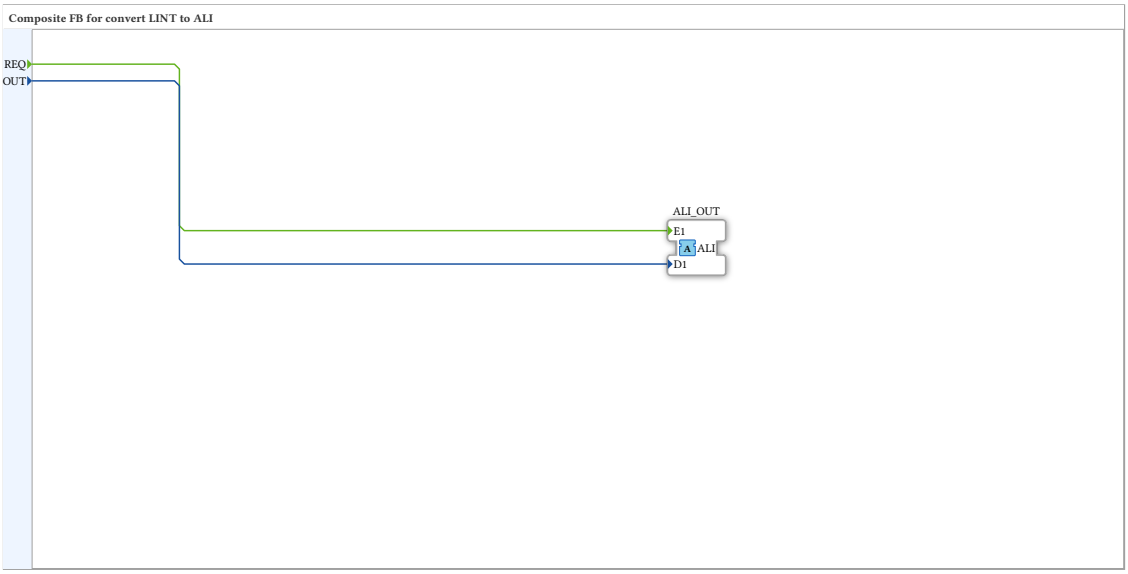
- SR_TO_QX_TRIG_EO
- SINT_TO_Q_NumericValue
- Softkey_T_FF_ILOCK_TO_QX_BG
- Bool8_TO_logiBUS_QX8

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

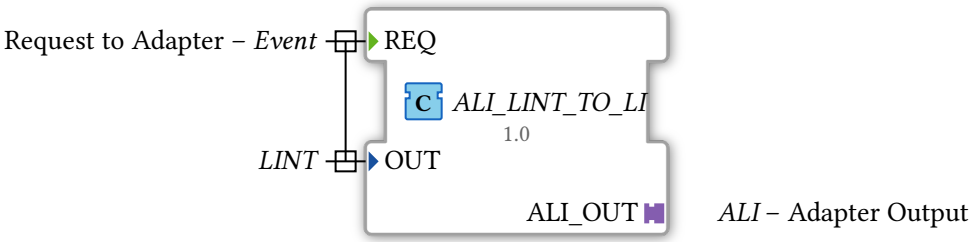
LINT

ALI_LINT_TO_LI



Einleitung

Der ALI_LINT_TO_LI Funktionsblock ist ein Composite-Funktionsblock, der zur Konvertierung eines LINT-Werts in das ALI-Adapterformat dient.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Anfrage-Event zur Auslösung der Konvertierung

Ereignis-Ausgänge

- Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

- **OUT**: LINT-Eingangswert, der konvertiert werden soll

Daten-Ausgänge

- Keine direkten Daten-Ausgänge vorhanden

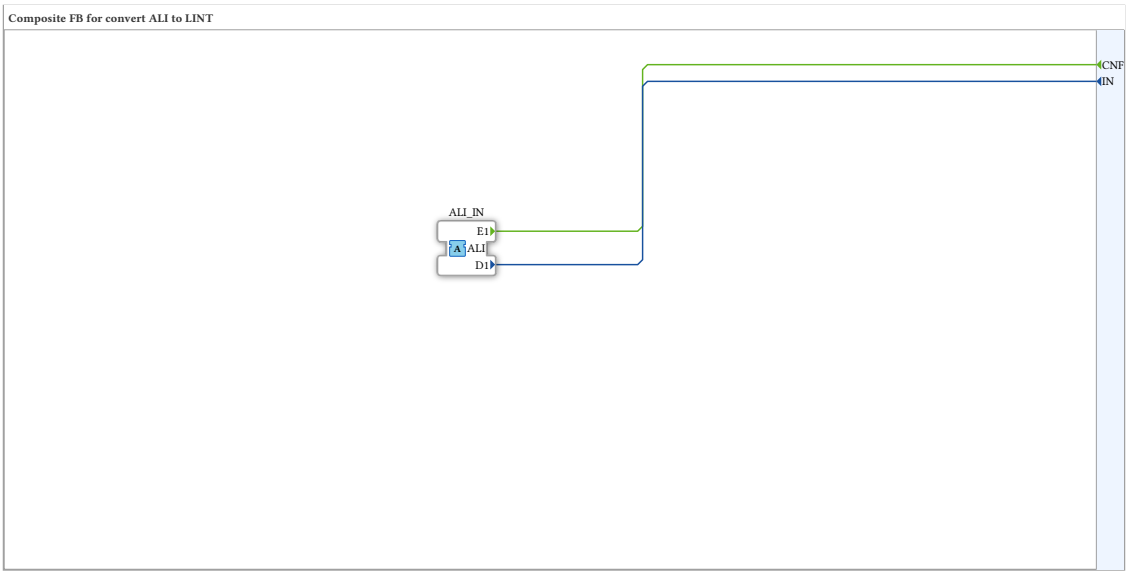
Adapter

- **ALI_OUT**: Plug vom Typ `ALI`

Funktionsweise

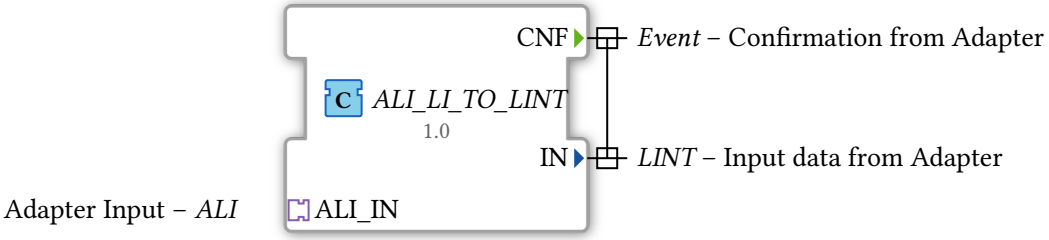
Der Funktionsblock empfängt einen LINT-Wert am Daten-Eingang OUT und wandelt diesen bei Eintreffen des REQ-Ereignisses in das ALI-Adapterformat um.

ALI_LI_TO_LINT



Einleitung

Der ALI_LI_TO_LINT Funktionsblock ist ein Composite-Funktionsblock, der zur Konvertierung des ALI-Adapterformats in einen LINT-Wert dient.



Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkten Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

- **CNF**: Bestätigungs-Event nach erfolgter Konvertierung

Daten-Eingänge

- Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

- **IN**: LINT-Ausgangswert der konvertierten Daten

Adapter

- **ALI_IN**: Socket vom Typ `ALI`

Funktionsweise

Der Funktionsblock empfängt Daten über den ALI-Adapter-Socket und stellt den enthaltenen LINT-Wert am Ausgang IN zur Verfügung, sobald ein Ereignis über den Adapter eintrifft (CNF).

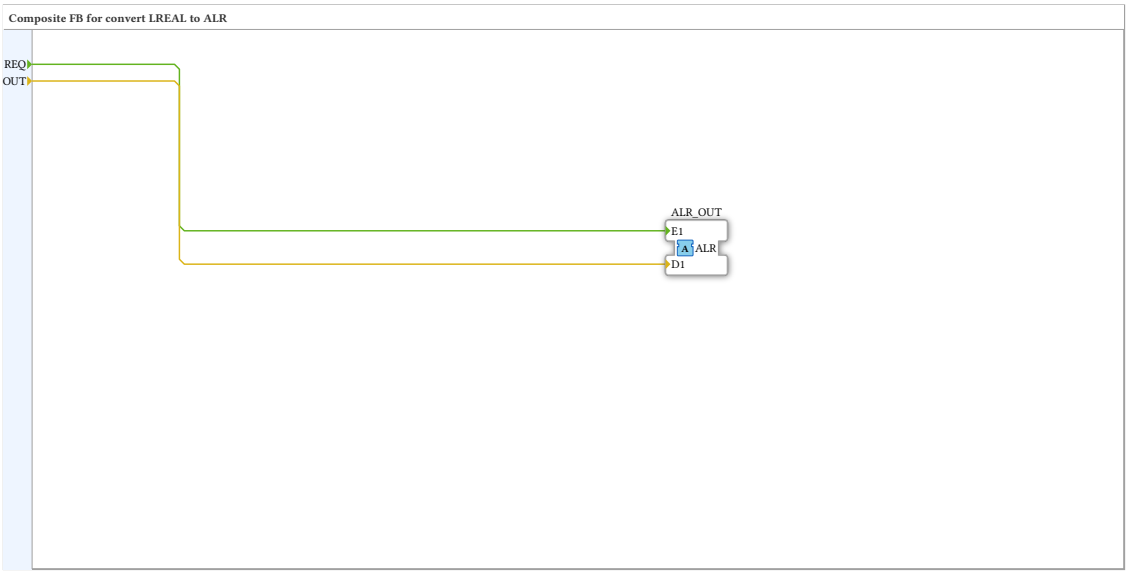
LINT Conversion

🛠️ Zugehörige Übungen

- Uebung_072c

LREAL

ALR_LREAL_TO_LR



Einleitung

Der ALR_LREAL_TO_LR Funktionsblock ist ein Composite-Funktionsblock, der zur Konvertierung eines LREAL-Werts in das ALR-Adapterformat dient.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Anfrage-Event zur Auslösung der Konvertierung

Ereignis-Ausgänge

- Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

- **OUT**: LREAL-Eingangswert, der konvertiert werden soll

Daten-Ausgänge

- Keine direkten Daten-Ausgänge vorhanden

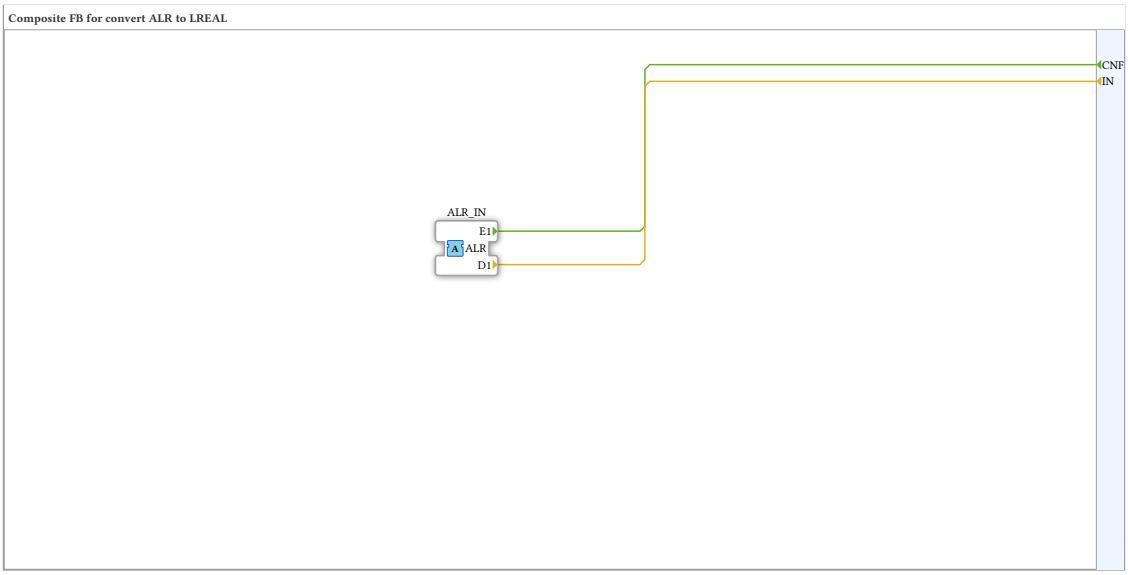
Adapter

- **ALR_OUT**: Plug vom Typ `ALR`

Funktionsweise

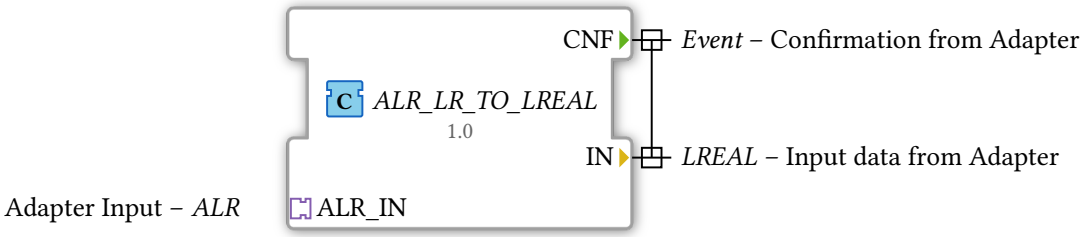
Der Funktionsblock empfängt einen LREAL-Wert am Daten-Eingang OUT und wandelt diesen bei Eintreffen des REQ-Ereignisses in das ALR-Adapterformat um.

ALR_LR_TO_LREAL



Einleitung

Der ALR_LR_TO_LREAL Funktionsblock ist ein Composite-Funktionsblock, der zur Konvertierung des ALR-Adapterformats in einen LREAL-Wert dient.



Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkten Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

- **CNF**: Bestätigungs-Event nach erfolgter Konvertierung

Daten-Eingänge

- Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

- **IN**: LREAL-Ausgangswert der konvertierten Daten

Adapter

- **ALR_IN**: Socket vom Typ `ALR`

Funktionsweise

Der Funktionsblock empfängt Daten über den ALR-Adapter-Socket und stellt den enthaltenen LREAL-Wert am Ausgang IN zur Verfügung, sobald ein Ereignis über den Adapter eintrifft (CNF).

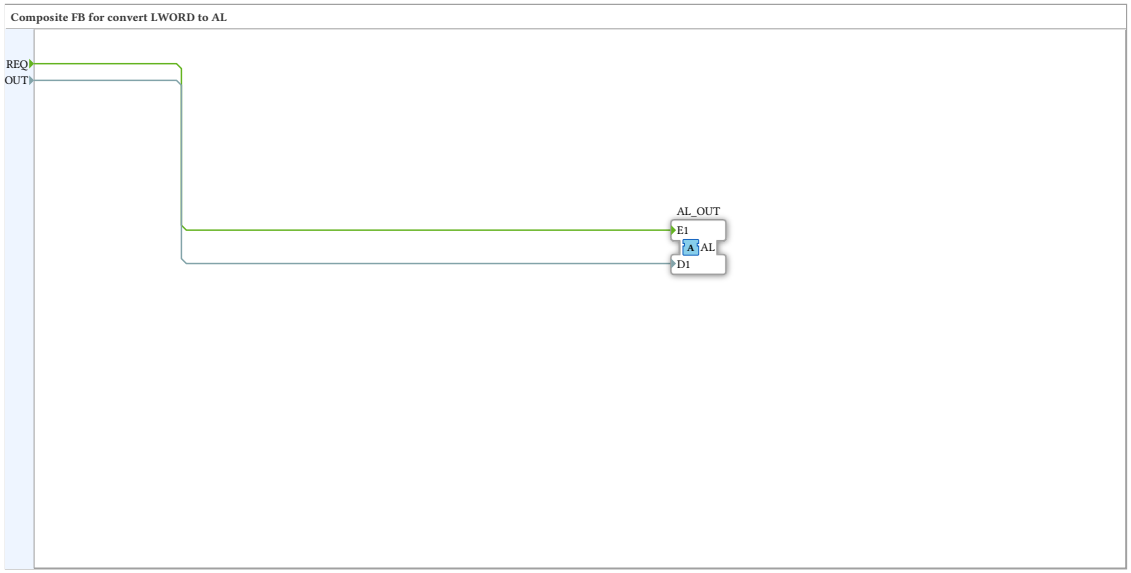
LREAL Conversion

🛠️ Zugehörige Übungen

- Uebung_003a0
- Uebung_003a0_AX
- Uebung_010c2
- Uebung_010c2_AX
- Uebung_035a
- Uebung_035a2
- Uebung_035a3

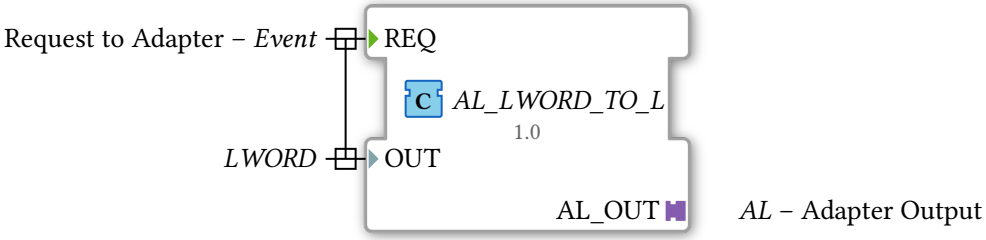
LWORD

AL_LWORD_TO_L



Einleitung

Der AL_LWORD_TO_L Funktionsblock ist ein Composite-Funktionsblock, der zur Konvertierung eines LWORD-Werts in das AL-Adapterformat dient.



Schnittstellenstruktur

Ereignis-Eingänge
- **REQ**: Anfrage-Event zur Auslösung der Konvertierung

Ereignis-Ausgänge
- Keine direkten Ereignis-Ausgänge vorhanden

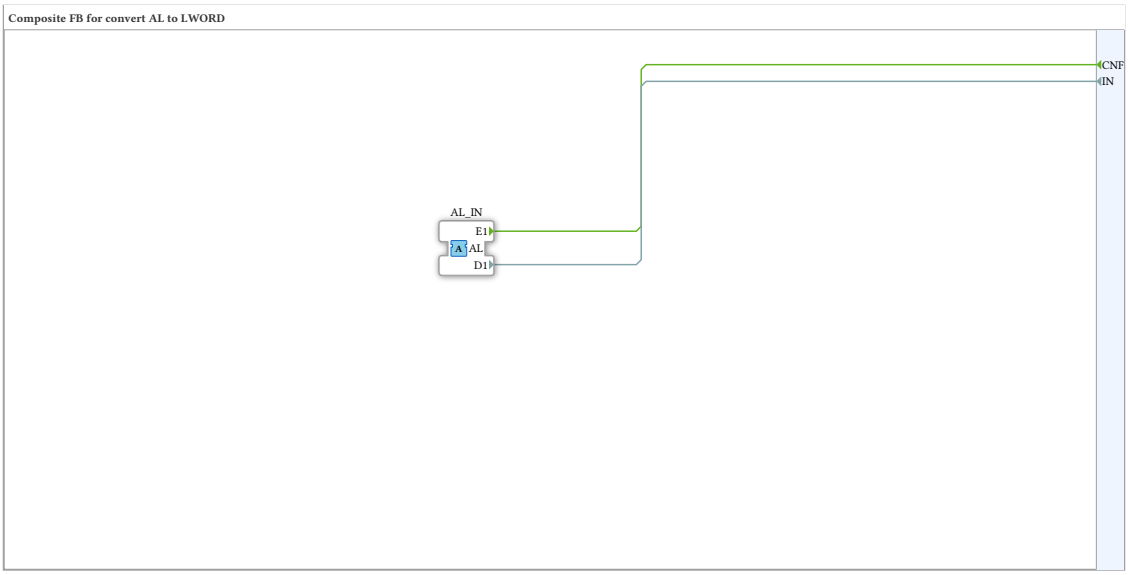
Daten-Eingänge
- **OUT**: LWORD-Eingangswert, der konvertiert werden soll

Daten-Ausgänge
- Keine direkten Daten-Ausgänge vorhanden

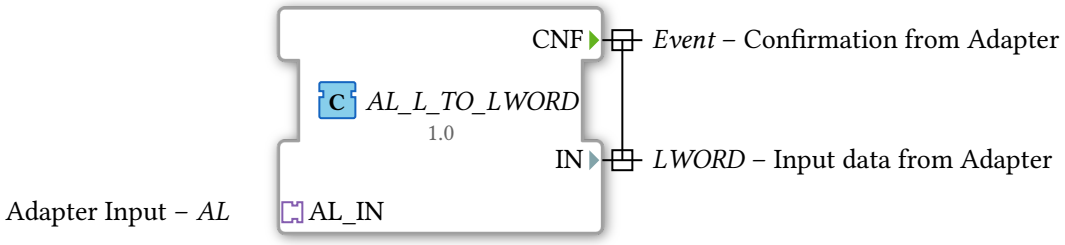
Adapter
- **AL_OUT**: Plug vom Typ `AL`

Funktionsweise
Der Funktionsblock empfängt einen LWORD-Wert am Daten-Eingang OUT und wandelt diesen bei Eintreffen des REQ-Ereignisses in das AL-Adapterformat um.

AL_L_TO_LWORD



Einleitung
Der AL_L_TO_LWORD Funktionsblock ist ein Composite-Funktionsblock, der zur Konvertierung des AL-Adapterformats in einen LWORD-Wert dient.



Schnittstellenstruktur

Ereignis-Eingänge
- Keine direkten Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

- **CNF**: Bestätigungs-Event nach erfolgter Konvertierung

Daten-Eingänge

- Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

- **IN**: LWORD-Ausgangswert der konvertierten Daten

Adapter

- **AL_IN**: Socket vom Typ `AL`

Funktionsweise

Der Funktionsblock empfängt Daten über den AL-Adapter-Socket und stellt den enthaltenen LWORD-Wert am Ausgang IN zur Verfügung, sobald ein Ereignis über den Adapter eintrifft (CNF).

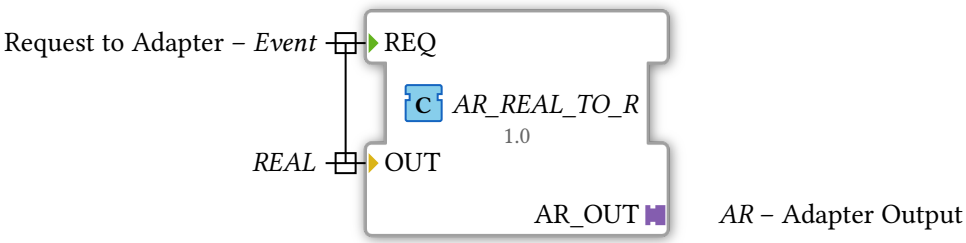
LWORD Conversion

REAL

AR_REAL_TO_R

Einleitung

Der AR_REAL_TO_R Funktionsblock ist ein Composite-Funktionsblock, der zur Konvertierung eines REAL-Werts in das AR-Adapterformat dient.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Anfrage-Event zur Auslösung der Konvertierung

Ereignis-Ausgänge

- Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

- **OUT**: REAL-Eingangswert, der konvertiert werden soll

Daten-Ausgänge

- Keine direkten Daten-Ausgänge vorhanden

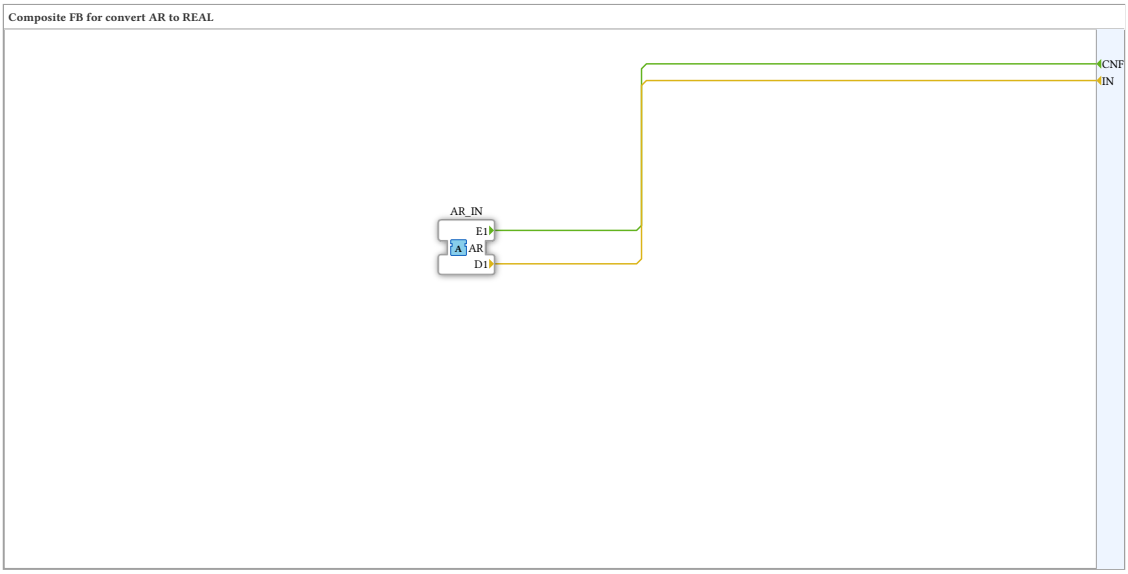
Adapter

- **AR_OUT**: Plug vom Typ `AR`

Funktionsweise

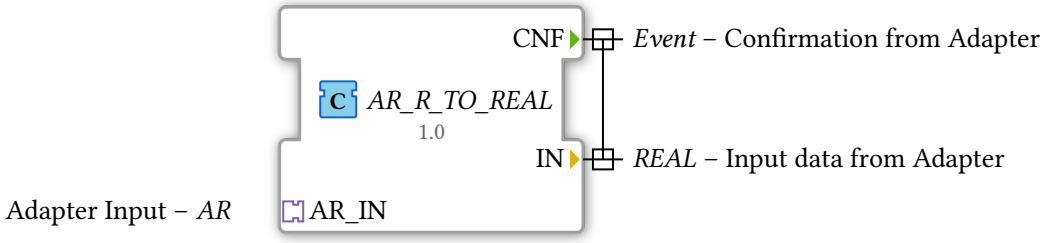
Der Funktionsblock empfängt einen REAL-Wert am Daten-Eingang OUT und wandelt diesen bei Eintreffen des REQ-Ereignisses in das AR-Adapterformat um.

AR_R_TO_REAL



Einleitung

Der AR_R_TO_REAL Funktionsblock ist ein Composite-Funktionsblock, der zur Konvertierung des AR-Adapterformats in einen REAL-Wert dient.



Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkten Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

- **CNF**: Bestätigungs-Event nach erfolgter Konvertierung

Daten-Eingänge

- Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

- **IN**: REAL-Ausgangswert der konvertierten Daten

Adapter

- **AR_IN**: Socket vom Typ `AR`

Funktionsweise

Der Funktionsblock empfängt Daten über den AR-Adapter-Socket und stellt den enthaltenen REAL-Wert am Ausgang IN zur Verfügung, sobald ein Ereignis über den Adapter eintrifft (CNF).

REAL Conversion

🎧 Podcast

- IEC 61499: Fabrikautomation im Umbruch – Von der Vision zur industriellen Realität?
- From „Get Away with Electronics“ to Essential: How Agricultural Tech Earned Trust and Delivered Real Relief

🛠️ Zugehörige Übungen

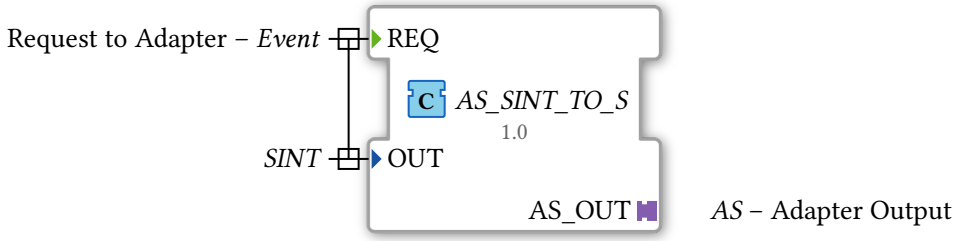
- Uebung_072c
- Uebung_151_AX
- Uebung_152
- Uebung_153

SINT

AS_SINT_TO_S

Einleitung

Der AS_SINT_TO_S Funktionsblock ist ein Composite-Funktionsblock, der zur Konvertierung eines SINT-Werts in das AS-Adapterformat dient.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Anfrage-Event zur Auslösung der Konvertierung

Ereignis-Ausgänge

- Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

- **OUT**: SINT-Eingangswert, der konvertiert werden soll

Daten-Ausgänge

- Keine direkten Daten-Ausgänge vorhanden

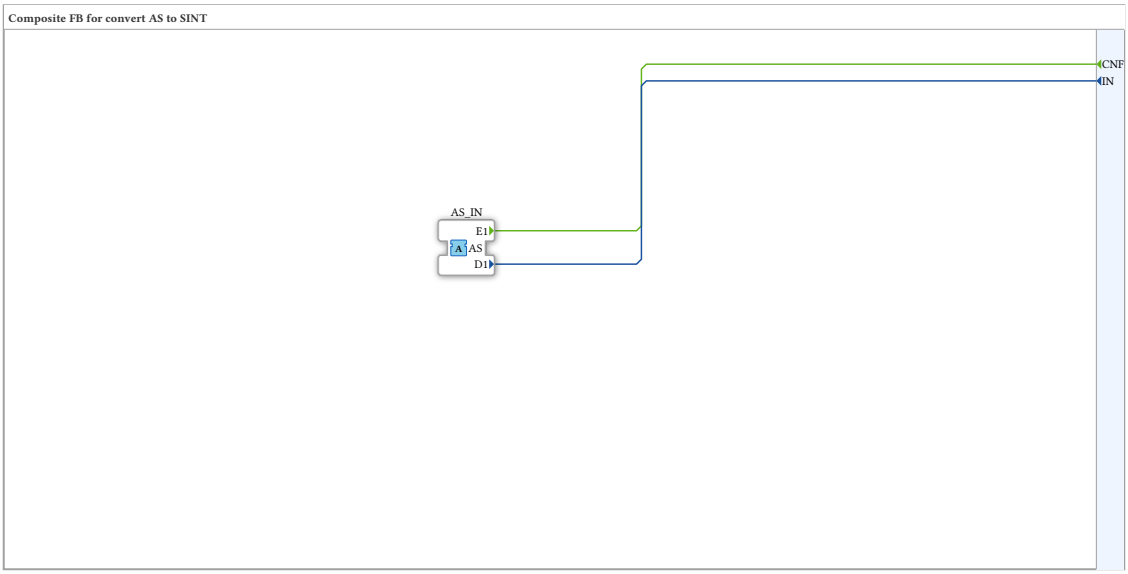
Adapter

- **AS_OUT**: Plug vom Typ `AS`

Funktionsweise

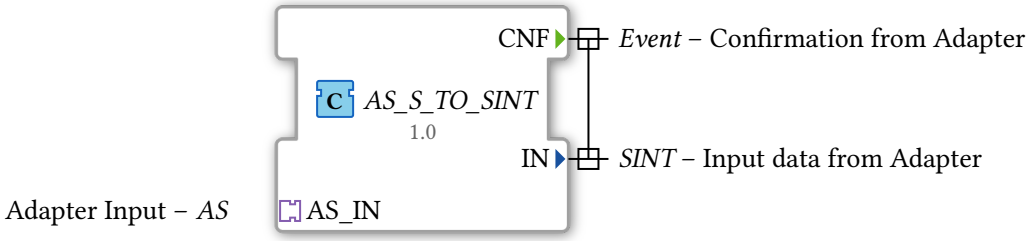
Der Funktionsblock empfängt einen SINT-Wert am Daten-Eingang OUT und wandelt diesen bei Eintreffen des REQ-Ereignisses in das AS-Adapterformat um.

AS_S_TO_SINT



Einleitung

Der AS_S_TO_SINT Funktionsblock ist ein Composite-Funktionsblock, der zur Konvertierung des AS-Adapterformats in einen SINT-Wert dient.



Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkten Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

- **CNF**: Bestätigungs-Event nach erfolgter Konvertierung

Daten-Eingänge

- Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

- **IN**: SINT-Ausgangswert der konvertierten Daten

Adapter

- **AS_IN**: Socket vom Typ `AS`

Funktionsweise

Der Funktionsblock empfängt Daten über den AS-Adapter-Socket und stellt den enthaltenen SINT-Wert am Ausgang IN zur Verfügung, sobald ein Ereignis über den Adapter eintrifft (CNF).

SINT Conversion

🎧 Podcast

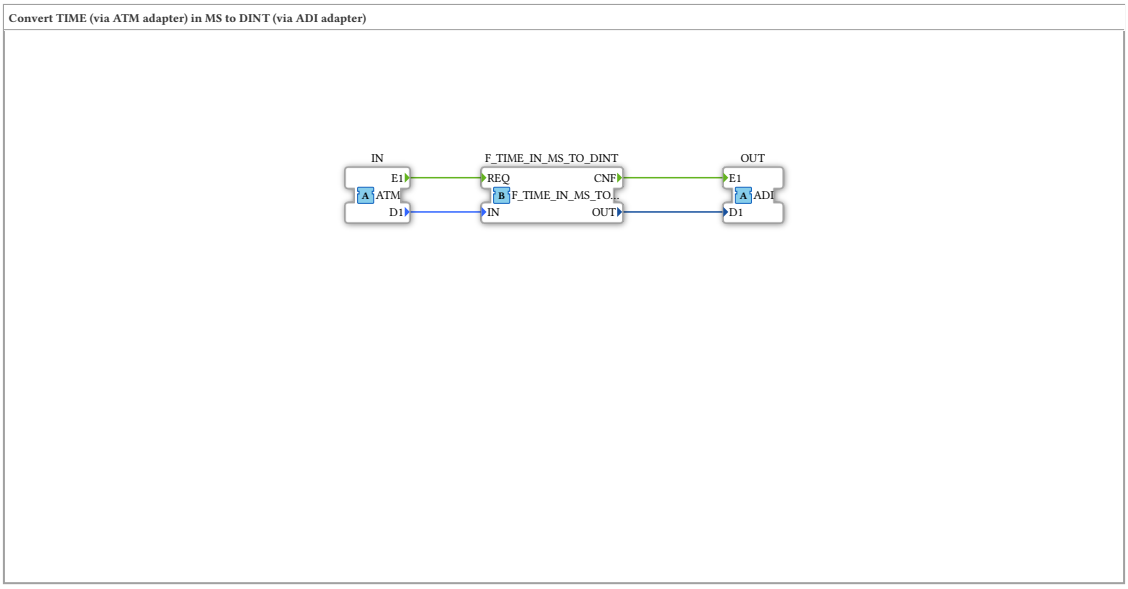
- SINT, INT, DINT: Warum die Wahl des Datentyps über Effizienz und Fehler entscheidet

🛠️ Zugehörige Übungen

- SINT_TO_Q_NumericValue

TIME

ATM_IN_MS_TO_ADI



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ATM_IN_MS_TO_ADI** ist ein Composite-Baustein, der einen `TIME`-Wert (übertragen über den ATM-Adapter) in Millisekunden auf einen `DINT`-Wert (32-Bit-Ganzzahl (vorzeichenbehaftet)) umrechnet und über den ADI-Adapter ausgibt. Er gehört zur Blockfamilie `ATM_IN_<EINHEIT>_TO_<ZIELTYP>`, die dieselbe Umrechnung für alle vier Zeiteinheiten (MS, NS, S, US) und alle fünf Zielganzzahl-/Gleitkommatypen bereitstellt.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkt sichtbaren Ereignis-Eingänge. Die Ereignissteuerung erfolgt indirekt über den Adapter **IN** (Ereignis E1).

Ereignis-Ausgänge

- Keine direkt sichtbaren Ereignis-Ausgänge. Die Ereignissignalisierung erfolgt indirekt über den Adapter **OUT** (Ereignis E1).

Daten-Eingänge

- Keine direkt sichtbaren Daten-Eingänge. Der Eingabewert wird über den Adapter **IN** (Datenpunkt D1 vom Typ `TIME`) bereitgestellt.

Daten-Ausgänge

- Keine direkt sichtbaren Daten-Ausgänge. Der Ausgabewert wird über den Adapter **OUT** (Datenpunkt D1 vom Typ `DINT`) bereitgestellt.

Adapter

Adapter	Rolle	Typ	Beschreibung
IN	Socket	adapter::types::unidirectional::ATM	Unidirektionaler `TIME`-Adapter als Eingangsschnittstelle.
OUT	Plug	adapter::types::unidirectional::ADI	Unidirektionaler `DINT`-Adapter als Ausgangsschnittstelle, Wert von `IN` in Millisekunden ausgedrückt.

Funktionsweise

Der Baustein enthält intern einen einzelnen Funktionsbaustein `F_TIME_IN_MS_TO_DINT` aus der IEC-61131-3-Standardbibliothek. Die Ablaufsteuerung erfolgt rein ereignisgesteuert:

1. Ein Ereignis am Eingangsadapter **IN.E1** triggert den Eingang **REQ** des internen Umrechnungsbausteins.
2. Der Datenwert **IN.D1** (`TIME`) wird an den Eingang **IN** von `F_TIME_IN_MS_TO_DINT` übergeben.
3. `F_TIME_IN_MS_TO_DINT` rechnet den Zeitwert in eine Millisekunden-Zahl vom Typ `DINT` um und signalisiert dies über seinen Ausgang **CNF**.
4. Dieses Ereignis wird an den Ausgangsadapter **OUT.E1** weitergeleitet.
5. Gleichzeitig wird der umgerechnete Wert (Ausgang **OUT** von `F_TIME_IN_MS_TO_DINT`) an den Datenpunkt **OUT.D1** übergeben.

Der gesamte Vorgang erfolgt synchron und ohne Zwischenspeicherung – jede erfolgreiche Umrechnung löst sofort ein Ausgangsereignis aus.

Technische Besonderheiten

- **Einheitenumrechnung nach IEC 61131-3**: Der interne Baustein `F_TIME_IN_MS_TO_DINT` interpretiert den intern in Nanosekunden gehaltenen `TIME`-Wert als Zahl in Millisekunden und liefert das Ergebnis als `DINT` (32-Bit-Ganzzahl (vorzeichenbehaftet)).
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional ausgelegt. Eine Rückwärtskommunikation ist nicht vorgesehen.
- **Composite-Struktur ohne eigene Logik**: Der Baustein besteht ausschließlich aus einer Daten- und einer Ereignisverbindung zum internen Umrechnungsbaustein; es gibt kein eigenes ECC.
- **Leerer TypeHash**: Wie bei allen generierten Konvertierungs-Composites in diesem Paket ist `eclipse4diac::core::TypeHash` leer (`''`).

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Der Ablauf lässt sich als einfacher Schritt beschreiben:

Schritt	Aktion
1	Warten auf Ereignis an **IN.E1**
2	Umrechnung des Datenwerts von `TIME` (Millisekunden) nach `DINT` über `F_TIME_IN_MS_TO_DINT`
3	Ausgabe des umgerechneten Werts an **OUT.D1** und Ereignis an **OUT.E1**
4	Rückkehr zu Schritt 1

Anwendungsszenarien

- **Anzeige/Weiterverarbeitung von Zeitdauern**: Eine über einen `ATM`-Adapter transportierte Zeitdauer (z. B. eine Zykluszeit oder Verzögerung) soll als Zahl in Millisekunden an einen Adapter-Socket vom Typ `ADI` weitergegeben werden, etwa zur Anzeige auf einem Panel oder zur Weiterverarbeitung in einer Berechnung.
- **Schnittstellenanpassung**: In Netzwerken, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen, ermöglicht der Baustein die nahtlose Umwandlung eines Zeit- in einen Zahlenadapter.

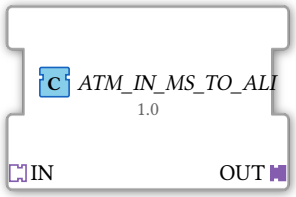
⚖️ Vergleich mit ähnlichen Bausteinen

- **Gleiche Einheit, andere Zieltypen**: ATM_IN_MS_TO_ALI, ATM_IN_MS_TO_ALR, ATM_IN_MS_TO_AUDI, ATM_IN_MS_TO_AULI
- **Gleicher Zieltyp, andere Einheiten**: ATM_IN_NS_TO_ADI, ATM_IN_S_TO_ADI, ATM_IN_US_TO_ADI
- **ATM_TIME_TO_TM** / **ATM_TM_TO_TIME**: reine Adapter-↔-Variablen-Durchreichung ohne Einheitenumrechnung, im Gegensatz zur numerischen Umrechnung von `ATM_IN_MS_TO_ADI`.

Fazit

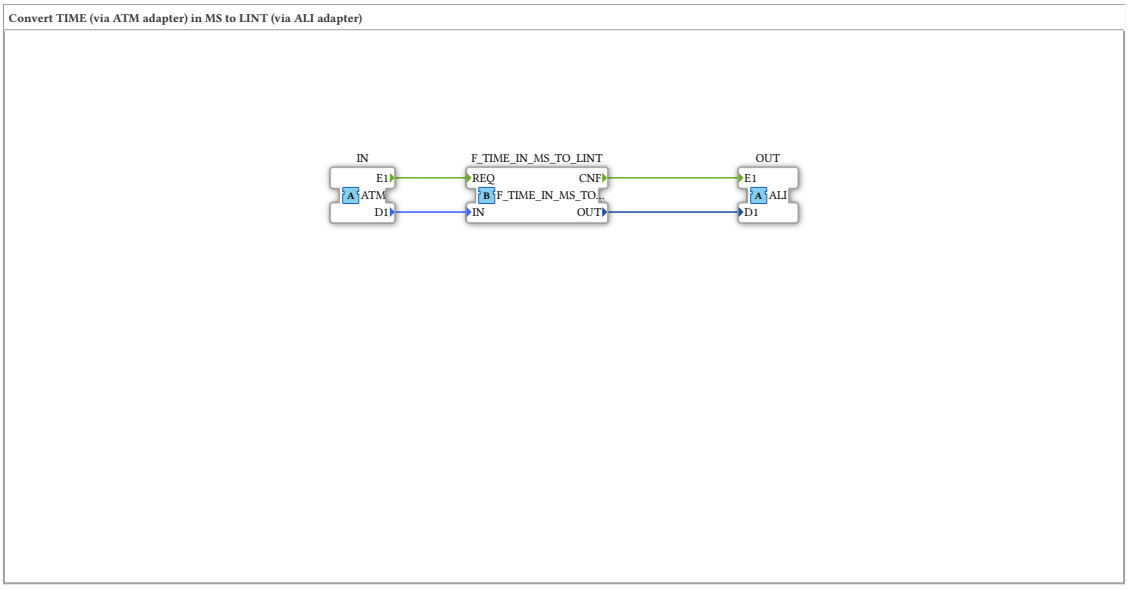
`ATM_IN_MS_TO_ADI` liefert eine standardkonforme, vollständig adapterbasierte Umrechnung eines `TIME`-Werts in Millisekunden auf einen `DINT`-Wert und fügt sich damit nahtlos in Netzwerke ein, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen.

ATM_IN_MS_TO_ALI



Time value to convert – *ATM*

ALI – TIME value of IN expressed in MS as LINT



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ATM_IN_MS_TO_ALI** ist ein Composite-Baustein, der einen `TIME`-Wert (übertragen über den ATM-Adapter) in Millisekunden auf einen `LINT`-Wert (64-Bit-Ganzzahl (vorzeichenbehaftet)) umrechnet und über den ALI-Adapter ausgibt. Er gehört zur Blockfamilie `ATM_IN_<EINHEIT>_TO_<ZIELTYP>`, die dieselbe Umrechnung für alle vier Zeiteinheiten (MS, NS, S, US) und alle fünf Zielganzzahl-/Gleitkommatypen bereitstellt.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkt sichtbaren Ereignis-Eingänge. Die Ereignissteuerung erfolgt indirekt über den Adapter **IN** (Ereignis E1).

Ereignis-Ausgänge

- Keine direkt sichtbaren Ereignis-Ausgänge. Die Ereignissignalisierung erfolgt indirekt über den Adapter **OUT** (Ereignis E1).

Daten-Eingänge

- Keine direkt sichtbaren Daten-Eingänge. Der Eingabewert wird über den Adapter **IN** (Datenpunkt D1 vom Typ `TIME`) bereitgestellt.

Daten-Ausgänge

- Keine direkt sichtbaren Daten-Ausgänge. Der Ausgabewert wird über den Adapter **OUT** (Datenpunkt D1 vom Typ `LINT`) bereitgestellt.

Adapter

Adapter	Rolle	Typ	Beschreibung
IN	Socket	adapter::types::unidirectional::ATM	Unidirektionaler `TIME`-Adapter als Eingangsschnittstelle.
OUT	Plug	adapter::types::unidirectional::ALI	Unidirektionaler `LINT`-Adapter als Ausgangsschnittstelle, Wert von `IN` in Millisekunden ausgedrückt.

Funktionsweise

Der Baustein enthält intern einen einzelnen Funktionsbaustein `F_TIME_IN_MS_TO_LINT` aus der IEC-61131-3-Standardbibliothek. Die Ablaufsteuerung erfolgt rein ereignisgesteuert:

1. Ein Ereignis am Eingangsadapter **IN.E1** triggert den Eingang **REQ** des internen Umrechnungsbausteins.
2. Der Datenwert **IN.D1** (`TIME`) wird an den Eingang **IN** von `F_TIME_IN_MS_TO_LINT` übergeben.
3. `F_TIME_IN_MS_TO_LINT` rechnet den Zeitwert in eine Millisekunden-Zahl vom Typ `LINT` um und signalisiert dies über seinen Ausgang **CNF**.
4. Dieses Ereignis wird an den Ausgangsadapter **OUT.E1** weitergeleitet.
5. Gleichzeitig wird der umgerechnete Wert (Ausgang **OUT** von `F_TIME_IN_MS_TO_LINT`) an den Datenpunkt **OUT.D1** übergeben.

Der gesamte Vorgang erfolgt synchron und ohne Zwischenspeicherung – jede erfolgreiche Umrechnung löst sofort ein Ausgangsereignis aus.

Technische Besonderheiten

- **Einheitenumrechnung nach IEC 61131-3**: Der interne Baustein `F_TIME_IN_MS_TO_LINT` interpretiert den intern in Nanosekunden gehaltenen `TIME`-Wert als Zahl in Millisekunden und liefert das Ergebnis als `LINT` (64-Bit-Ganzzahl (vorzeichenbehaftet)).
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional ausgelegt. Eine Rückwärtskommunikation ist nicht vorgesehen.
- **Composite-Struktur ohne eigene Logik**: Der Baustein besteht ausschließlich aus einer Daten- und einer Ereignisverbindung zum internen Umrechnungsbaustein; es gibt kein eigenes ECC.
- **Leerer TypeHash**: Wie bei allen generierten Konvertierungs-Composites in diesem Paket ist `eclipse4diac::core::TypeHash` leer (`''`).

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Der Ablauf lässt sich als einfacher Schritt beschreiben:

Schritt	Aktion
1	Warten auf Ereignis an **IN.E1**
2	Umrechnung des Datenwerts von `TIME` (Millisekunden) nach `LINT` über `F_TIME_IN_MS_TO_LINT`
3	Ausgabe des umgerechneten Werts an **OUT.D1** und Ereignis an **OUT.E1**
4	Rückkehr zu Schritt 1

Anwendungsszenarien

- **Anzeige/Weiterverarbeitung von Zeitdauern**: Eine über einen `ATM`-Adapter transportierte Zeitdauer (z. B. eine Zykluszeit oder Verzögerung) soll als Zahl in Millisekunden an einen Adapter-Socket vom Typ `ALI` weitergegeben werden, etwa zur Anzeige auf einem Panel oder zur Weiterverarbeitung in einer Berechnung.
- **Schnittstellenanpassung**: In Netzwerken, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen, ermöglicht der Baustein die nahtlose Umwandlung eines Zeit- in einen Zahlenadapter.

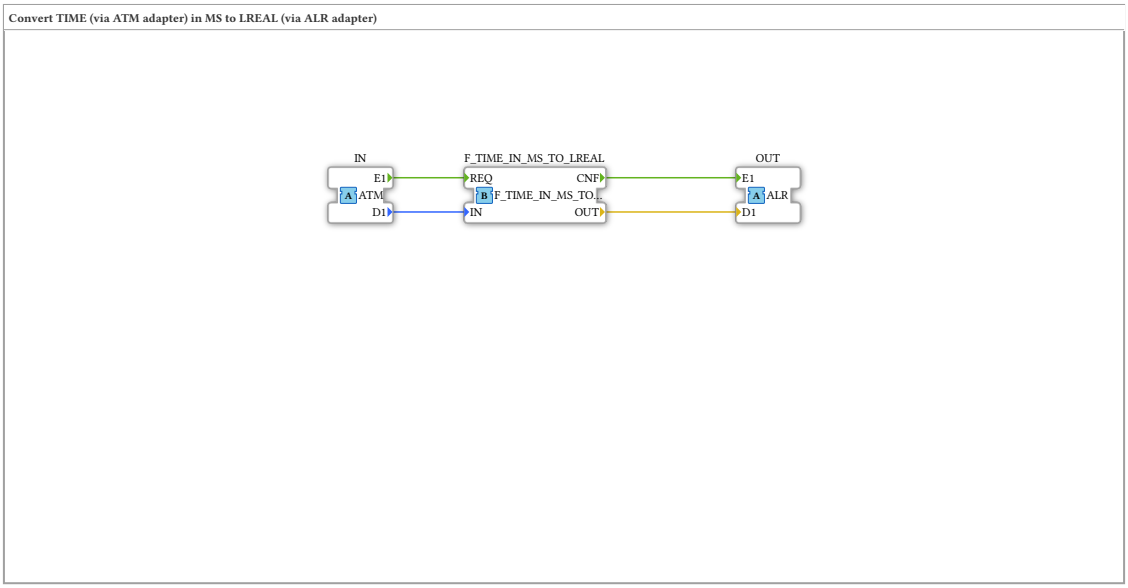
⚖️ Vergleich mit ähnlichen Bausteinen

- **Gleiche Einheit, andere Zieltypen**: ATM_IN_MS_TO_ADI, ATM_IN_MS_TO_ALR, ATM_IN_MS_TO_AUDI, ATM_IN_MS_TO_AULI
- **Gleicher Zieltyp, andere Einheiten**: ATM_IN_NS_TO_ALI, ATM_IN_S_TO_ALI, ATM_IN_US_TO_ALI
- **ATM_TIME_TO_TM / ATM_TM_TO_TIME**: reine Adapter-↔-Variablen-Durchreichung ohne Einheitenumrechnung, im Gegensatz zur numerischen Umrechnung von `ATM_IN_MS_TO_ALI`.

Fazit

`ATM_IN_MS_TO_ALI` liefert eine standardkonforme, vollständig adapterbasierte Umrechnung eines `TIME`-Werts in Millisekunden auf einen `LINT`-Wert und fügt sich damit nahtlos in Netzwerke ein, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen.

ATM_IN_MS_TO_ALR



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ATM_IN_MS_TO_ALR** ist ein Composite-Baustein, der einen `TIME`-Wert (übertragen über den ATM-Adapter) in Millisekunden auf einen `LREAL`-Wert (64-Bit-Gleitkommazahl) umrechnet und über den ALR-Adapter ausgibt. Er gehört zur Blockfamilie `ATM_IN_<EINHEIT>_TO_<ZIELTYP>`, die dieselbe Umrechnung für alle vier Zeiteinheiten (MS, NS, S, US) und alle fünf Zielganzzahl-/Gleitkommatypen bereitstellt.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkt sichtbaren Ereignis-Eingänge. Die Ereignissteuerung erfolgt indirekt über den Adapter **IN** (Ereignis E1).

Ereignis-Ausgänge

- Keine direkt sichtbaren Ereignis-Ausgänge. Die Ereignissignalisierung erfolgt indirekt über den Adapter **OUT** (Ereignis E1).

Daten-Eingänge

- Keine direkt sichtbaren Daten-Eingänge. Der Eingabewert wird über den Adapter **IN** (Datenpunkt D1 vom Typ `TIME`) bereitgestellt.

Daten-Ausgänge

- Keine direkt sichtbaren Daten-Ausgänge. Der Ausgabewert wird über den Adapter **OUT** (Datenpunkt D1 vom Typ `LREAL`) bereitgestellt.

Adapter

Adapter	Rolle	Typ	Beschreibung
IN	Socket	adapter::types::unidirectional::ATM	Unidirektionaler `TIME`-Adapter als Eingangsschnittstelle.
OUT	Plug	adapter::types::unidirectional::ALR	Unidirektionaler `LREAL`-Adapter als Ausgangsschnittstelle, Wert von `IN` in Millisekunden ausgedrückt.

Funktionsweise

Der Baustein enthält intern einen einzelnen Funktionsbaustein `F_TIME_IN_MS_TO_LREAL` aus der IEC-61131-3-Standardbibliothek. Die Ablaufsteuerung erfolgt rein ereignisgesteuert:

1. Ein Ereignis am Eingangsadapter **IN.E1** triggert den Eingang **REQ** des internen Umrechnungsbausteins.
2. Der Datenwert **IN.D1** (`TIME`) wird an den Eingang **IN** von `F_TIME_IN_MS_TO_LREAL` übergeben.
3. `F_TIME_IN_MS_TO_LREAL` rechnet den Zeitwert in eine Millisekunden-Zahl vom Typ `LREAL` um und signalisiert dies über seinen Ausgang **CNF**.
4. Dieses Ereignis wird an den Ausgangsadapter **OUT.E1** weitergeleitet.
5. Gleichzeitig wird der umgerechnete Wert (Ausgang **OUT** von `F_TIME_IN_MS_TO_LREAL`) an den Datenpunkt **OUT.D1** übergeben.

Der gesamte Vorgang erfolgt synchron und ohne Zwischenspeicherung – jede erfolgreiche Umrechnung löst sofort ein Ausgangsereignis aus.

Technische Besonderheiten

- **Einheitenumrechnung nach IEC 61131-3**: Der interne Baustein `F_TIME_IN_MS_TO_LREAL` interpretiert den intern in Nanosekunden gehaltenen `TIME`-Wert als Zahl in Millisekunden und liefert das Ergebnis als `LREAL` (64-Bit-Gleitkommazahl).
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional ausgelegt. Eine Rückwärtskommunikation ist nicht vorgesehen.
- **Composite-Struktur ohne eigene Logik**: Der Baustein besteht ausschließlich aus einer Daten- und einer Ereignisverbindung zum internen Umrechnungsbaustein; es gibt kein eigenes ECC.
- **Leerer TypeHash**: Wie bei allen generierten Konvertierungs-Composites in diesem Paket ist `eclipse4diac::core::TypeHash` leer (`''`).

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Der Ablauf lässt sich als einfacher Schritt beschreiben:

Schritt	Aktion
1	Warten auf Ereignis an **IN.E1**
2	Umrechnung des Datenwerts von `TIME` (Millisekunden) nach `LREAL` über `F_TIME_IN_MS_TO_LREAL`
3	Ausgabe des umgerechneten Werts an **OUT.D1** und Ereignis an **OUT.E1**
4	Rückkehr zu Schritt 1

Anwendungsszenarien

- **Anzeige/Weiterverarbeitung von Zeitdauern**: Eine über einen `ATM`-Adapter transportierte Zeitdauer (z. B. eine Zykluszeit oder Verzögerung) soll als Zahl in Millisekunden an einen Adapter-Socket vom Typ `ALR` weitergegeben werden, etwa zur Anzeige auf einem Panel oder zur Weiterverarbeitung in einer Berechnung.
- **Schnittstellenanpassung**: In Netzwerken, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen, ermöglicht der Baustein die nahtlose Umwandlung eines Zeit- in einen Zahlenadapter.

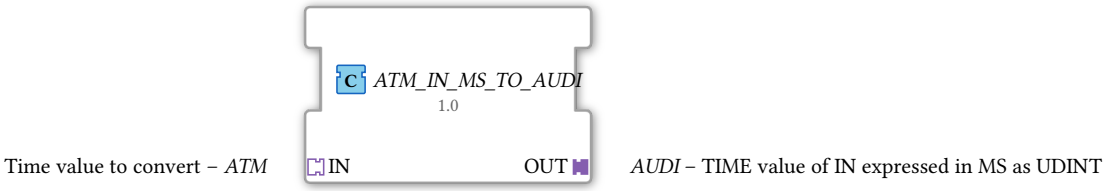
⚖️ Vergleich mit ähnlichen Bausteinen

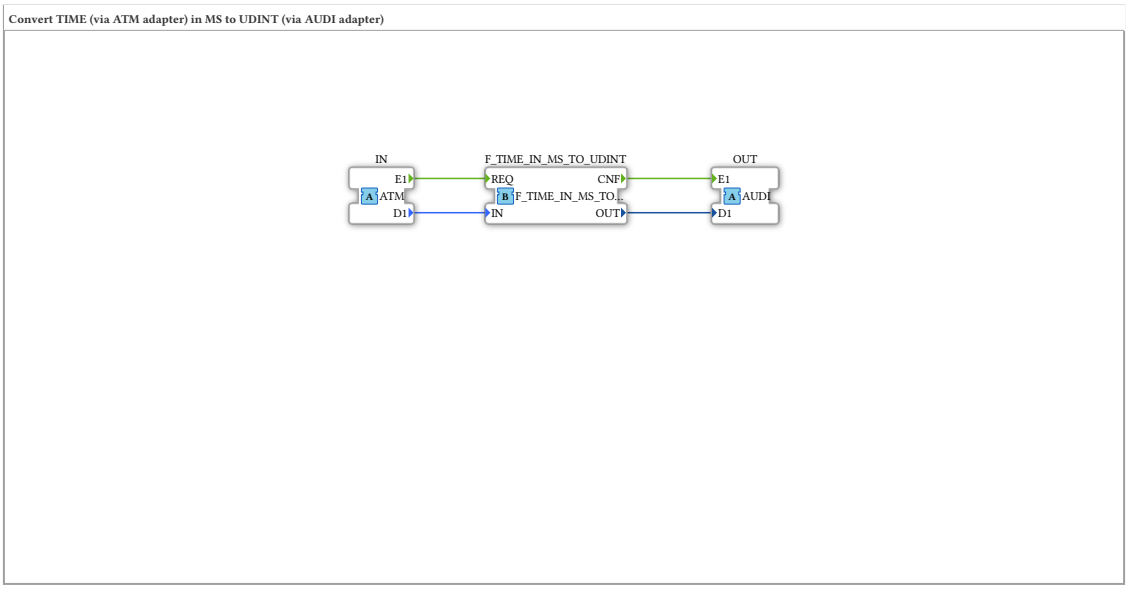
- **Gleiche Einheit, andere Zieltypen**: ATM_IN_MS_TO_ADI, ATM_IN_MS_TO_ALI, ATM_IN_MS_TO_AUDI, ATM_IN_MS_TO_AULI
- **Gleicher Zieltyp, andere Einheiten**: ATM_IN_NS_TO_ALR, ATM_IN_S_TO_ALR, ATM_IN_US_TO_ALR
- **ATM_TIME_TO_TM** / **ATM_TM_TO_TIME**: reine Adapter-↔-Variablen-Durchreichung ohne Einheitenumrechnung, im Gegensatz zur numerischen Umrechnung von `ATM_IN_MS_TO_ALR`.

Fazit

`ATM_IN_MS_TO_ALR` liefert eine standardkonforme, vollständig adapterbasierte Umrechnung eines `TIME`-Werts in Millisekunden auf einen `LREAL`-Wert und fügt sich damit nahtlos in Netzwerke ein, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen.

ATM_IN_MS_TO_AUDI





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ATM_IN_MS_TO_AUDI** ist ein Composite-Baustein, der einen `TIME`-Wert (übertragen über den ATM-Adapter) in Millisekunden auf einen `UDINT`-Wert (32-Bit-Ganzzahl (vorzeichenlos)) umrechnet und über den AUDI-Adapter ausgibt. Er gehört zur Blockfamilie `ATM_IN_<EINHEIT>_TO_<ZIELTYP>`, die dieselbe Umrechnung für alle vier Zeiteinheiten (MS, NS, S, US) und alle fünf Zielganzzahl-/Gleitkommatypen bereitstellt.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkt sichtbaren Ereignis-Eingänge. Die Ereignissteuerung erfolgt indirekt über den Adapter **IN** (Ereignis E1).

Ereignis-Ausgänge

- Keine direkt sichtbaren Ereignis-Ausgänge. Die Ereignissignalisierung erfolgt indirekt über den Adapter **OUT** (Ereignis E1).

Daten-Eingänge

- Keine direkt sichtbaren Daten-Eingänge. Der Eingabewert wird über den Adapter **IN** (Datenpunkt D1 vom Typ `TIME`) bereitgestellt.

Daten-Ausgänge

- Keine direkt sichtbaren Daten-Ausgänge. Der Ausgabewert wird über den Adapter **OUT** (Datenpunkt D1 vom Typ `UDINT`) bereitgestellt.

Adapter

Adapter	Rolle	Typ	Beschreibung
IN	Socket	adapter::types::unidirectional::ATM	Unidirektionaler `TIME`-Adapter als Eingangsschnittstelle.
OUT	Plug	adapter::types::unidirectional::AUDI	Unidirektionaler `UDINT`-Adapter als Ausgangsschnittstelle, Wert von `IN` in Millisekunden ausgedrückt.

Funktionsweise

Der Baustein enthält intern einen einzelnen Funktionsbaustein `F_TIME_IN_MS_TO_UDINT` aus der IEC-61131-3-Standardbibliothek. Die Ablaufsteuerung erfolgt rein ereignisgesteuert:

1. Ein Ereignis am Eingangsadapter **IN.E1** triggert den Eingang **REQ** des internen Umrechnungsbausteins.
2. Der Datenwert **IN.D1** (`TIME`) wird an den Eingang **IN** von `F_TIME_IN_MS_TO_UDINT` übergeben.
3. `F_TIME_IN_MS_TO_UDINT` rechnet den Zeitwert in eine Millisekunden-Zahl vom Typ `UDINT` um und signalisiert dies über seinen Ausgang **CNF**.
4. Dieses Ereignis wird an den Ausgangsadapter **OUT.E1** weitergeleitet.
5. Gleichzeitig wird der umgerechnete Wert (Ausgang **OUT** von `F_TIME_IN_MS_TO_UDINT`) an den Datenpunkt **OUT.D1** übergeben.

Der gesamte Vorgang erfolgt synchron und ohne Zwischenspeicherung – jede erfolgreiche Umrechnung löst sofort ein Ausgangsereignis aus.

Technische Besonderheiten

- **Einheitenumrechnung nach IEC 61131-3**: Der interne Baustein `F_TIME_IN_MS_TO_UDINT` interpretiert den intern in Nanosekunden gehaltenen `TIME`-Wert als Zahl in Millisekunden und liefert das Ergebnis als `UDINT` (32-Bit-Ganzzahl (vorzeichenlos)).
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional ausgelegt. Eine Rückwärtskommunikation ist nicht vorgesehen.
- **Composite-Struktur ohne eigene Logik**: Der Baustein besteht ausschließlich aus einer Daten- und einer Ereignisverbindung zum internen Umrechnungsbaustein; es gibt kein eigenes ECC.
- **Leerer TypeHash**: Wie bei allen generierten Konvertierungs-Composites in diesem Paket ist `eclipse4diac::core::TypeHash` leer (`''`).

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Der Ablauf lässt sich als einfacher Schritt beschreiben:

Schritt	Aktion

1	Warten auf Ereignis an **IN.E1**
2	Umrechnung des Datenwerts von `TIME` (Millisekunden) nach `UDINT` über `F_TIME_IN_MS_TO_UDINT`
3	Ausgabe des umgerechneten Werts an **OUT.D1** und Ereignis an **OUT.E1**
4	Rückkehr zu Schritt 1

Anwendungsszenarien

- **Anzeige/Weiterverarbeitung von Zeitdauern**: Eine über einen `ATM`-Adapter transportierte Zeitdauer (z. B. eine Zykluszeit oder Verzögerung) soll als Zahl in Millisekunden an einen Adapter-Socket vom Typ `AUDI` weitergegeben werden, etwa zur Anzeige auf einem Panel oder zur Weiterverarbeitung in einer Berechnung.
- **Schnittstellenanpassung**: In Netzwerken, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen, ermöglicht der Baustein die nahtlose Umwandlung eines Zeit- in einen Zahlenadapter.

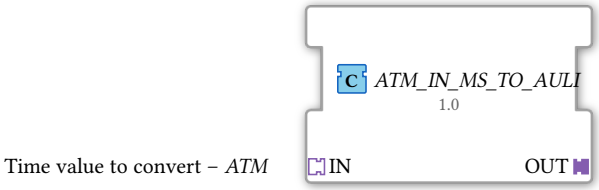
⚖️ **Vergleich mit ähnlichen Bausteinen**

- **Gleiche Einheit, andere Zieltypen**: ATM_IN_MS_TO_ADI, ATM_IN_MS_TO_ALI, ATM_IN_MS_TO_ALR, ATM_IN_MS_TO_AULI
- **Gleicher Zieltyp, andere Einheiten**: ATM_IN_NS_TO_AUDI, ATM_IN_S_TO_AUDI, ATM_IN_US_TO_AUDI
- **ATM_TIME_TO_TM / ATM_TM_TO_TIME**: reine Adapter-↔-Variablen-Durchreichung ohne Einheitenumrechnung, im Gegensatz zur numerischen Umrechnung von `ATM_IN_MS_TO_AUDI`.

Fazit

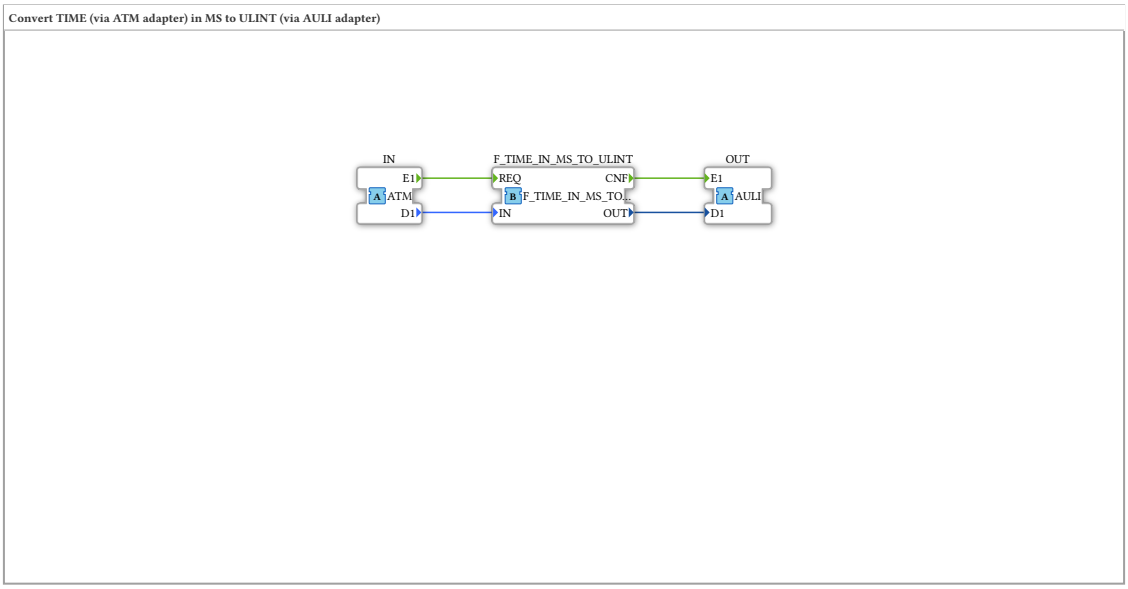
`ATM_IN_MS_TO_AUDI` liefert eine standardkonforme, vollständig adapterbasierte Umrechnung eines `TIME`-Werts in Millisekunden auf einen `UDINT`-Wert und fügt sich damit nahtlos in Netzwerke ein, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen.

ATM_IN_MS_TO_AULI



Time value to convert – *ATM*

AULI – TIME value of IN expressed in MS as ULINT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ATM_IN_MS_TO_AULI** ist ein Composite-Baustein, der einen `TIME`-Wert (übertragen über den ATM-Adapter) in Millisekunden auf einen `ULINT`-Wert (64-Bit-Ganzzahl (vorzeichenlos)) umrechnet und über den AULI-Adapter ausgibt. Er gehört zur Blockfamilie `ATM_IN_<EINHEIT>_TO_<ZIELTYP>`, die dieselbe Umrechnung für alle vier Zeiteinheiten (MS, NS, S, US) und alle fünf Zielganzzahl-/Gleitkommatypen bereitstellt.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkt sichtbaren Ereignis-Eingänge. Die Ereignissteuerung erfolgt indirekt über den Adapter **IN** (Ereignis E1).

Ereignis-Ausgänge

- Keine direkt sichtbaren Ereignis-Ausgänge. Die Ereignissignalisierung erfolgt indirekt über den Adapter **OUT** (Ereignis E1).

Daten-Eingänge

- Keine direkt sichtbaren Daten-Eingänge. Der Eingabewert wird über den Adapter **IN** (Datenpunkt D1 vom Typ `TIME`) bereitgestellt.

Daten-Ausgänge

- Keine direkt sichtbaren Daten-Ausgänge. Der Ausgabewert wird über den Adapter **OUT** (Datenpunkt D1 vom Typ `ULINT`) bereitgestellt.

Adapter

Adapter	Rolle	Typ	Beschreibung
IN	Socket	adapter::types::unidirectional::ATM	Unidirektionaler `TIME`-Adapter als Eingangsschnittstelle.
OUT	Plug	adapter::types::unidirectional::AULI	Unidirektionaler `ULINT`-Adapter als Ausgangsschnittstelle, Wert von `IN` in Millisekunden ausgedrückt.

Funktionsweise

Der Baustein enthält intern einen einzelnen Funktionsbaustein `F_TIME_IN_MS_TO_ULINT` aus der IEC-61131-3-Standardbibliothek. Die Ablaufsteuerung erfolgt rein ereignisgesteuert:

1. Ein Ereignis am Eingangsadapter **IN.E1** triggert den Eingang **REQ** des internen Umrechnungsbausteins.
2. Der Datenwert **IN.D1** (`TIME`) wird an den Eingang **IN** von `F_TIME_IN_MS_TO_ULINT` übergeben.
3. `F_TIME_IN_MS_TO_ULINT` rechnet den Zeitwert in eine Millisekunden-Zahl vom Typ `ULINT` um und signalisiert dies über seinen Ausgang **CNF**.
4. Dieses Ereignis wird an den Ausgangsadapter **OUT.E1** weitergeleitet.
5. Gleichzeitig wird der umgerechnete Wert (Ausgang **OUT** von `F_TIME_IN_MS_TO_ULINT`) an den Datenpunkt **OUT.D1** übergeben.

Der gesamte Vorgang erfolgt synchron und ohne Zwischenspeicherung – jede erfolgreiche Umrechnung löst sofort ein Ausgangsereignis aus.

Technische Besonderheiten

- **Einheitenumrechnung nach IEC 61131-3**: Der interne Baustein `F_TIME_IN_MS_TO_ULINT` interpretiert den intern in Nanosekunden gehaltenen `TIME`-Wert als Zahl in Millisekunden und liefert das Ergebnis als `ULINT` (64-Bit-Ganzzahl (vorzeichenlos)).
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional ausgelegt. Eine Rückwärtskommunikation ist nicht vorgesehen.

- **Composite-Struktur ohne eigene Logik**: Der Baustein besteht ausschließlich aus einer Daten- und einer Ereignisverbindung zum internen Umrechnungsbaustein; es gibt kein eigenes ECC.
- **Leerer TypeHash**: Wie bei allen generierten Konvertierungs-Composites in diesem Paket ist `eclipse4diac::core::TypeHash` leer (`''`).

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Der Ablauf lässt sich als einfacher Schritt beschreiben:

Schritt	Aktion
1	Warten auf Ereignis an **IN.E1**
2	Umrechnung des Datenwerts von `TIME` (Millisekunden) nach `ULINT` über `F_TIME_IN_MS_TO_ULINT`
3	Ausgabe des umgerechneten Werts an **OUT.D1** und Ereignis an **OUT.E1**
4	Rückkehr zu Schritt 1

Anwendungsszenarien

- **Anzeige/Weiterverarbeitung von Zeitdauern**: Eine über einen `ATM`-Adapter transportierte Zeitdauer (z. B. eine Zykluszeit oder Verzögerung) soll als Zahl in Millisekunden an einen Adapter-Socket vom Typ `AULI` weitergegeben werden, etwa zur Anzeige auf einem Panel oder zur Weiterverarbeitung in einer Berechnung.
- **Schnittstellenanpassung**: In Netzwerken, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen, ermöglicht der Baustein die nahtlose Umwandlung eines Zeit- in einen Zahlenadapter.

⚖️ Vergleich mit ähnlichen Bausteinen

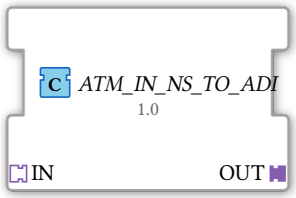
- **Gleiche Einheit, andere Zieltypen**: ATM_IN_MS_TO_ADI, ATM_IN_MS_TO_ALI, ATM_IN_MS_TO_ALR, ATM_IN_MS_TO_AUDI
- **Gleicher Zieltyp, andere Einheiten**: ATM_IN_NS_TO_AULI, ATM_IN_S_TO_AULI, ATM_IN_US_TO_AULI
- **ATM_TIME_TO_TM** / **ATM_TM_TO_TIME**: reine Adapter-↔-Variablen-Durchreichung ohne Einheitenumrechnung, im Gegensatz zur numerischen Umrechnung von `ATM_IN_MS_TO_AULI`.

Fazit

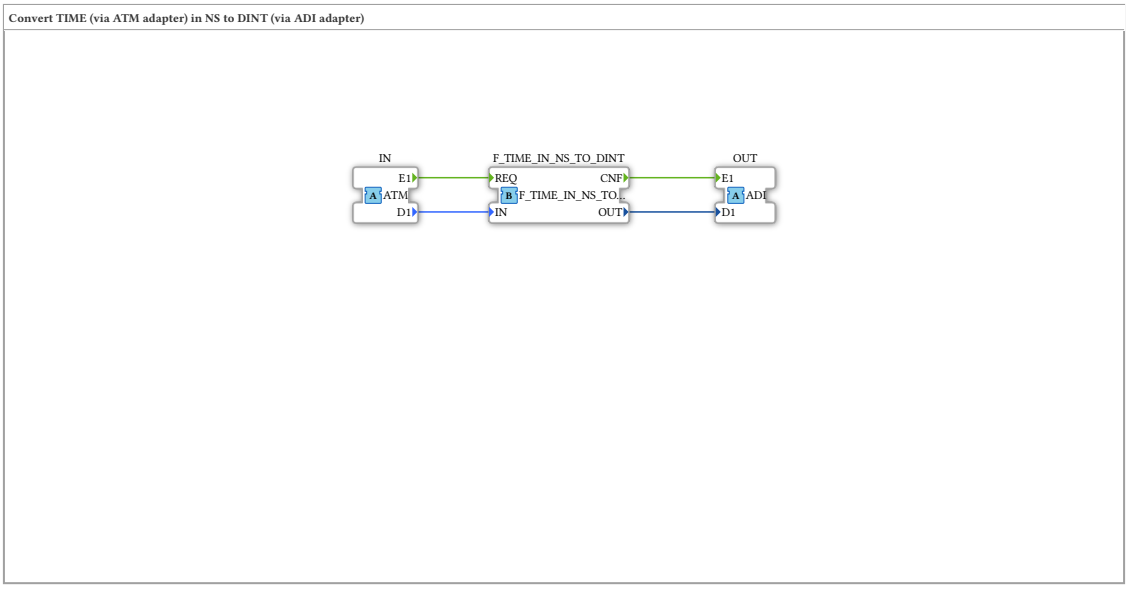
`ATM_IN_MS_TO_AULI` liefert eine standardkonforme, vollständig adapterbasierte Umrechnung eines `TIME`-Werts in Millisekunden auf einen `ULINT`-Wert und fügt sich damit nahtlos in Netzwerke ein, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen.

ATM_IN_NS_TO_ADI

Time value to convert – *ATM*



ADI – TIME value of IN expressed in NS as DINT



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ATM_IN_NS_TO_ADI** ist ein Composite-Baustein, der einen `TIME`-Wert (übertragen über den ATM-Adapter) in Nanosekunden auf einen `DINT`-Wert (32-Bit-Ganzzahl (vorzeichenbehaftet)) umrechnet und über den ADI-Adapter ausgibt. Er gehört zur Blockfamilie `ATM_IN_<EINHEIT>_TO_<ZIELTYP>`, die dieselbe Umrechnung für alle vier Zeiteinheiten (MS, NS, S, US) und alle fünf Zielganzzahl-/Gleitkommatypen bereitstellt.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkt sichtbaren Ereignis-Eingänge. Die Ereignissteuerung erfolgt indirekt über den Adapter **IN** (Ereignis E1).

Ereignis-Ausgänge

- Keine direkt sichtbaren Ereignis-Ausgänge. Die Ereignissignalisierung erfolgt indirekt über den Adapter **OUT** (Ereignis E1).

Daten-Eingänge

- Keine direkt sichtbaren Daten-Eingänge. Der Eingabewert wird über den Adapter **IN** (Datenpunkt D1 vom Typ `TIME`) bereitgestellt.

Daten-Ausgänge

- Keine direkt sichtbaren Daten-Ausgänge. Der Ausgabewert wird über den Adapter **OUT** (Datenpunkt D1 vom Typ `DINT`) bereitgestellt.

Adapter

Adapter	Rolle	Typ	Beschreibung
IN	Socket	adapter::types::unidirectional::ATM	Unidirektionaler `TIME`-Adapter als Eingangsschnittstelle.
OUT	Plug	adapter::types::unidirectional::ADI	Unidirektionaler `DINT`-Adapter als Ausgangsschnittstelle, Wert von `IN` in Nanosekunden ausgedrückt.

Funktionsweise

Der Baustein enthält intern einen einzelnen Funktionsbaustein `F_TIME_IN_NS_TO_DINT` aus der IEC-61131-3-Standardbibliothek. Die Ablaufsteuerung erfolgt rein ereignisgesteuert:

1. Ein Ereignis am Eingangsadapter **IN.E1** triggert den Eingang **REQ** des internen Umrechnungsbausteins.
2. Der Datenwert **IN.D1** (`TIME`) wird an den Eingang **IN** von `F_TIME_IN_NS_TO_DINT` übergeben.
3. `F_TIME_IN_NS_TO_DINT` rechnet den Zeitwert in eine Nanosekunden-Zahl vom Typ `DINT` um und signalisiert dies über seinen Ausgang **CNF**.
4. Dieses Ereignis wird an den Ausgangsadapter **OUT.E1** weitergeleitet.
5. Gleichzeitig wird der umgerechnete Wert (Ausgang **OUT** von `F_TIME_IN_NS_TO_DINT`) an den Datenpunkt **OUT.D1** übergeben.

Der gesamte Vorgang erfolgt synchron und ohne Zwischenspeicherung – jede erfolgreiche Umrechnung löst sofort ein Ausgangsereignis aus.

Technische Besonderheiten

- **Einheitenumrechnung nach IEC 61131-3**: Der interne Baustein `F_TIME_IN_NS_TO_DINT` interpretiert den intern in Nanosekunden gehaltenen `TIME`-Wert als Zahl in Nanosekunden und liefert das Ergebnis als `DINT` (32-Bit-Ganzzahl (vorzeichenbehaftet)).
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional ausgelegt. Eine Rückwärtskommunikation ist nicht vorgesehen.
- **Composite-Struktur ohne eigene Logik**: Der Baustein besteht ausschließlich aus einer Daten- und einer Ereignisverbindung zum internen Umrechnungsbaustein; es gibt kein eigenes ECC.
- **Leerer TypeHash**: Wie bei allen generierten Konvertierungs-Composites in diesem Paket ist `eclipse4diac::core::TypeHash` leer (`''`).

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Der Ablauf lässt sich als einfacher Schritt beschreiben:

Schritt	Aktion
1	Warten auf Ereignis an **IN.E1**
2	Umrechnung des Datenwerts von `TIME` (Nanosekunden) nach `DINT` über `F_TIME_IN_NS_TO_DINT`
3	Ausgabe des umgerechneten Werts an **OUT.D1** und Ereignis an **OUT.E1**
4	Rückkehr zu Schritt 1

Anwendungsszenarien

- **Anzeige/Weiterverarbeitung von Zeitdauern**: Eine über einen `ATM`-Adapter transportierte Zeitdauer (z. B. eine Zykluszeit oder Verzögerung) soll als Zahl in Nanosekunden an einen Adapter-Socket vom Typ `ADI` weitergegeben werden, etwa zur Anzeige auf einem Panel oder zur Weiterverarbeitung in einer Berechnung.
- **Schnittstellenanpassung**: In Netzwerken, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen, ermöglicht der Baustein die nahtlose Umwandlung eines Zeit- in einen Zahlenadapter.

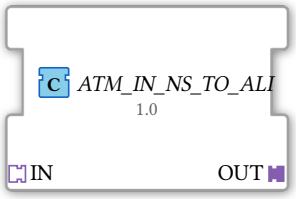
⚖️ Vergleich mit ähnlichen Bausteinen

- **Gleiche Einheit, andere Zieltypen**: ATM_IN_NS_TO_ALI, ATM_IN_NS_TO_ALR, ATM_IN_NS_TO_AUDI, ATM_IN_NS_TO_AULI
- **Gleicher Zieltyp, andere Einheiten**: ATM_IN_MS_TO_ADI, ATM_IN_S_TO_ADI, ATM_IN_US_TO_ADI
- **ATM_TIME_TO_TM** / **ATM_TM_TO_TIME**: reine Adapter-↔-Variablen-Durchreichung ohne Einheitenumrechnung, im Gegensatz zur numerischen Umrechnung von `ATM_IN_NS_TO_ADI`.

Fazit

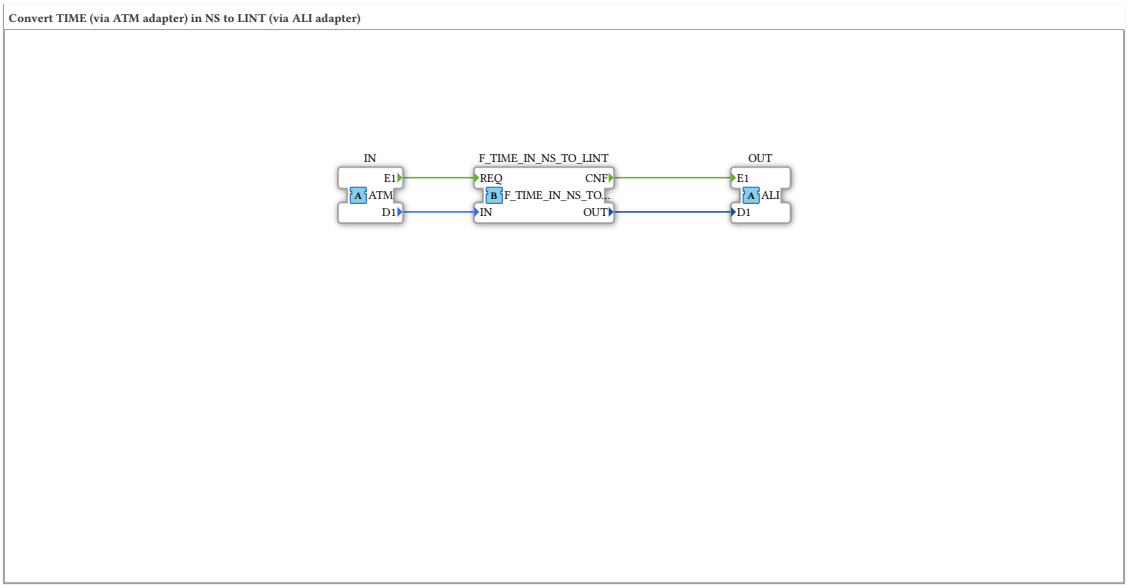
`ATM_IN_NS_TO_ADI` liefert eine standardkonforme, vollständig adapterbasierte Umrechnung eines `TIME`-Werts in Nanosekunden auf einen `DINT`-Wert und fügt sich damit nahtlos in Netzwerke ein, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen.

ATM_IN_NS_TO_ALI



Time value to convert – *ATM*

ALI – TIME value of IN expressed in NS as LINT



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ATM_IN_NS_TO_ALI** ist ein Composite-Baustein, der einen `TIME`-Wert (übertragen über den ATM-Adapter) in Nanosekunden auf einen `LINT`-Wert (64-Bit-Ganzzahl (vorzeichenbehaftet)) umrechnet und über den ALI-Adapter ausgibt. Er gehört zur Blockfamilie `ATM_IN_<EINHEIT>_TO_<ZIELTYP>`, die dieselbe Umrechnung für alle vier Zeiteinheiten (MS, NS, S, US) und alle fünf Zielganzzahl-/Gleitkommatypen bereitstellt.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkt sichtbaren Ereignis-Eingänge. Die Ereignissteuerung erfolgt indirekt über den Adapter **IN** (Ereignis E1).

Ereignis-Ausgänge

- Keine direkt sichtbaren Ereignis-Ausgänge. Die Ereignissignalisierung erfolgt indirekt über den Adapter **OUT** (Ereignis E1).

Daten-Eingänge

- Keine direkt sichtbaren Daten-Eingänge. Der Eingabewert wird über den Adapter **IN** (Datenpunkt D1 vom Typ `TIME`) bereitgestellt.

Daten-Ausgänge

- Keine direkt sichtbaren Daten-Ausgänge. Der Ausgabewert wird über den Adapter **OUT** (Datenpunkt D1 vom Typ `LINT`) bereitgestellt.

Adapter

Adapter	Rolle	Typ	Beschreibung
IN	Socket	adapter::types::unidirectional::ATM	Unidirektionaler TIME-Adapter als Eingangsschnittstelle.
OUT	Plug	adapter::types::unidirectional::ALI	Unidirektionaler LINT-Adapter als Ausgangsschnittstelle, Wert von IN in Nanosekunden ausgedrückt.

Funktionsweise

Der Baustein enthält intern einen einzelnen Funktionsbaustein F_TIME_IN_NS_TO_LINT aus der IEC-61131-3-Standardbibliothek. Die Ablaufsteuerung erfolgt rein ereignisgesteuert:

1. Ein Ereignis am Eingangsadapter **IN.E1** triggert den Eingang **REQ** des internen Umrechnungsbausteins.
2. Der Datenwert **IN.D1** (TIME) wird an den Eingang **IN** von F_TIME_IN_NS_TO_LINT übergeben.
3. F_TIME_IN_NS_TO_LINT rechnet den Zeitwert in eine Nanosekunden-Zahl vom Typ LINT um und signalisiert dies über seinen Ausgang **CNF**.
4. Dieses Ereignis wird an den Ausgangsadapter **OUT.E1** weitergeleitet.
5. Gleichzeitig wird der umgerechnete Wert (Ausgang **OUT** von F_TIME_IN_NS_TO_LINT) an den Datenpunkt **OUT.D1** übergeben.

Der gesamte Vorgang erfolgt synchron und ohne Zwischenspeicherung – jede erfolgreiche Umrechnung löst sofort ein Ausgangsereignis aus.

Technische Besonderheiten

- **Einheitenumrechnung nach IEC 61131-3**: Der interne Baustein F_TIME_IN_NS_TO_LINT interpretiert den intern in Nanosekunden gehaltenen TIME-Wert als Zahl in Nanosekunden und liefert das Ergebnis als LINT (64-Bit-Ganzzahl (vorzeichenbehaftet)).
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional ausgelegt. Eine Rückwärtskommunikation ist nicht vorgesehen.
- **Composite-Struktur ohne eigene Logik**: Der Baustein besteht ausschließlich aus einer Daten- und einer Ereignisverbindung zum internen Umrechnungsbaustein; es gibt kein eigenes ECC.
- **Leerer TypeHash**: Wie bei allen generierten Konvertierungs-Composites in diesem Paket ist eclipse4diac::core::TypeHash leer ('').

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Der Ablauf lässt sich als einfacher Schritt beschreiben:

Schritt	Aktion
1	Warten auf Ereignis an **IN.E1**
2	Umrechnung des Datenwerts von TIME (Nanosekunden) nach LINT über F_TIME_IN_NS_TO_LINT
3	Ausgabe des umgerechneten Werts an **OUT.D1** und Ereignis an **OUT.E1**
4	Rückkehr zu Schritt 1

Anwendungsszenarien

- **Anzeige/Weiterverarbeitung von Zeitdauern**: Eine über einen ATM-Adapter transportierte Zeitdauer (z. B. eine Zykluszeit oder Verzögerung) soll als Zahl in Nanosekunden an einen Adapter-Socket vom Typ ALI weitergegeben werden, etwa zur Anzeige auf einem Panel oder zur Weiterverarbeitung in einer Berechnung.
- **Schnittstellenanpassung**: In Netzwerken, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen, ermöglicht der Baustein die nahtlose Umwandlung eines Zeit- in einen Zahlenadapter.

⚖️ Vergleich mit ähnlichen Bausteinen

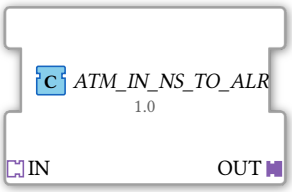
- **Gleiche Einheit, andere Zieltypen**: ATM_IN_NS_TO_ADI, ATM_IN_NS_TO_ALR, ATM_IN_NS_TO_AUDI, ATM_IN_NS_TO_AULI
- **Gleicher Zieltyp, andere Einheiten**: ATM_IN_MS_TO_ALI, ATM_IN_S_TO_ALI, ATM_IN_US_TO_ALI
- **ATM_TIME_TO_TM / ATM_TM_TO_TIME**: reine Adapter-↔-Variablen-Durchreichung ohne Einheitenumrechnung, im Gegensatz zur numerischen Umrechnung von `ATM_IN_NS_TO_ALI`.

Fazit

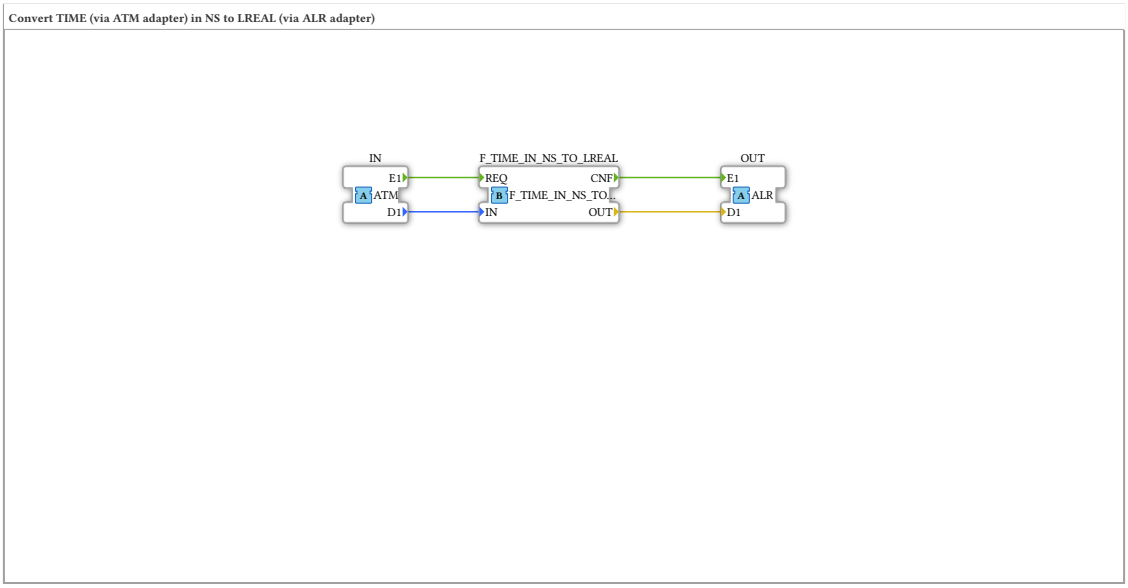
`ATM_IN_NS_TO_ALI` liefert eine standardkonforme, vollständig adapterbasierte Umrechnung eines `TIME`-Werts in Nanosekunden auf einen `LINT`-Wert und fügt sich damit nahtlos in Netzwerke ein, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen.

ATM_IN_NS_TO_ALR

Time value to convert – *ATM*



ALR – TIME value of IN expressed in NS as LREAL



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ATM_IN_NS_TO_ALR** ist ein Composite-Baustein, der einen `TIME`-Wert (übertragen über den ATM-Adapter) in Nanosekunden auf einen `LREAL`-Wert (64-Bit-Gleitkommazahl) umrechnet und über den ALR-Adapter ausgibt. Er gehört zur Blockfamilie `ATM_IN_<EINHEIT>_TO_<ZIELTYP>`, die dieselbe Umrechnung für alle vier Zeiteinheiten (MS, NS, S, US) und alle fünf Zielganzzahl-/Gleitkommatypen bereitstellt.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkt sichtbaren Ereignis-Eingänge. Die Ereignissteuerung erfolgt indirekt über den Adapter **IN** (Ereignis E1).

Ereignis-Ausgänge

- Keine direkt sichtbaren Ereignis-Ausgänge. Die Ereignissignalisierung erfolgt indirekt über den Adapter **OUT** (Ereignis E1).

Daten-Eingänge

- Keine direkt sichtbaren Daten-Eingänge. Der Eingabewert wird über den Adapter **IN** (Datenpunkt D1 vom Typ `TIME`) bereitgestellt.

Daten-Ausgänge

- Keine direkt sichtbaren Daten-Ausgänge. Der Ausgabewert wird über den Adapter **OUT** (Datenpunkt D1 vom Typ `LREAL`) bereitgestellt.

Adapter

Adapter	Rolle	Typ	Beschreibung
IN	Socket	adapter::types::unidirectional::ATM	Unidirektionaler `TIME`-Adapter als Eingangsschnittstelle.
OUT	Plug	adapter::types::unidirectional::ALR	Unidirektionaler `LREAL`-Adapter als Ausgangsschnittstelle, Wert von `IN` in Nanosekunden ausgedrückt.

Funktionsweise

Der Baustein enthält intern einen einzelnen Funktionsbaustein `F_TIME_IN_NS_TO_LREAL` aus der IEC-61131-3-Standardbibliothek. Die Ablaufsteuerung erfolgt rein ereignisgesteuert:

1. Ein Ereignis am Eingangsadapter **IN.E1** triggert den Eingang **REQ** des internen Umrechnungsbausteins.
2. Der Datenwert **IN.D1** (`TIME`) wird an den Eingang **IN** von `F_TIME_IN_NS_TO_LREAL` übergeben.
3. `F_TIME_IN_NS_TO_LREAL` rechnet den Zeitwert in eine Nanosekunden-Zahl vom Typ `LREAL` um und signalisiert dies über seinen Ausgang **CNF**.
4. Dieses Ereignis wird an den Ausgangsadapter **OUT.E1** weitergeleitet.
5. Gleichzeitig wird der umgerechnete Wert (Ausgang **OUT** von `F_TIME_IN_NS_TO_LREAL`) an den Datenpunkt **OUT.D1** übergeben.

Der gesamte Vorgang erfolgt synchron und ohne Zwischenspeicherung – jede erfolgreiche Umrechnung löst sofort ein Ausgangsereignis aus.

Technische Besonderheiten

- **Einheitenumrechnung nach IEC 61131-3**: Der interne Baustein `F_TIME_IN_NS_TO_LREAL` interpretiert den intern in Nanosekunden gehaltenen `TIME`-Wert als Zahl in Nanosekunden und liefert das Ergebnis als `LREAL` (64-Bit-Gleitkommazahl).
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional ausgelegt. Eine Rückwärtskommunikation ist nicht vorgesehen.
- **Composite-Struktur ohne eigene Logik**: Der Baustein besteht ausschließlich aus einer Daten- und einer Ereignisverbindung zum internen Umrechnungsbaustein; es gibt kein eigenes ECC.
- **Leerer TypeHash**: Wie bei allen generierten Konvertierungs-Composites in diesem Paket ist `eclipse4diac::core::TypeHash` leer (`''`).

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Der Ablauf lässt sich als einfacher Schritt beschreiben:

Schritt	Aktion
1	Warten auf Ereignis an **IN.E1**
2	Umrechnung des Datenwerts von `TIME` (Nanosekunden) nach `LREAL` über `F_TIME_IN_NS_TO_LREAL`
3	Ausgabe des umgerechneten Werts an **OUT.D1** und Ereignis an **OUT.E1**
4	Rückkehr zu Schritt 1

Anwendungsszenarien

- **Anzeige/Weiterverarbeitung von Zeitdauern**: Eine über einen `ATM`-Adapter transportierte Zeitdauer (z. B. eine Zykluszeit oder Verzögerung) soll als Zahl in Nanosekunden an einen Adapter-Socket vom Typ `ALR` weitergegeben werden, etwa zur Anzeige auf einem Panel oder zur Weiterverarbeitung in einer Berechnung.
- **Schnittstellenanpassung**: In Netzwerken, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen, ermöglicht der Baustein die nahtlose Umwandlung eines Zeit- in einen Zahlenadapter.

⚖️ Vergleich mit ähnlichen Bausteinen

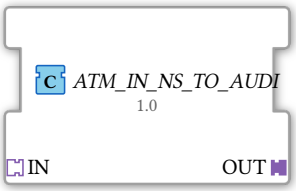
- **Gleiche Einheit, andere Zieltypen**: ATM_IN_NS_TO_ADI, ATM_IN_NS_TO_ALI, ATM_IN_NS_TO_AUDI, ATM_IN_NS_TO_AULI
- **Gleicher Zieltyp, andere Einheiten**: ATM_IN_MS_TO_ALR, ATM_IN_S_TO_ALR, ATM_IN_US_TO_ALR
- **ATM_TIME_TO_TM** / **ATM_TM_TO_TIME**: reine Adapter-↔-Variablen-Durchreichung ohne Einheitenumrechnung, im Gegensatz zur numerischen Umrechnung von `ATM_IN_NS_TO_ALR`.

Fazit

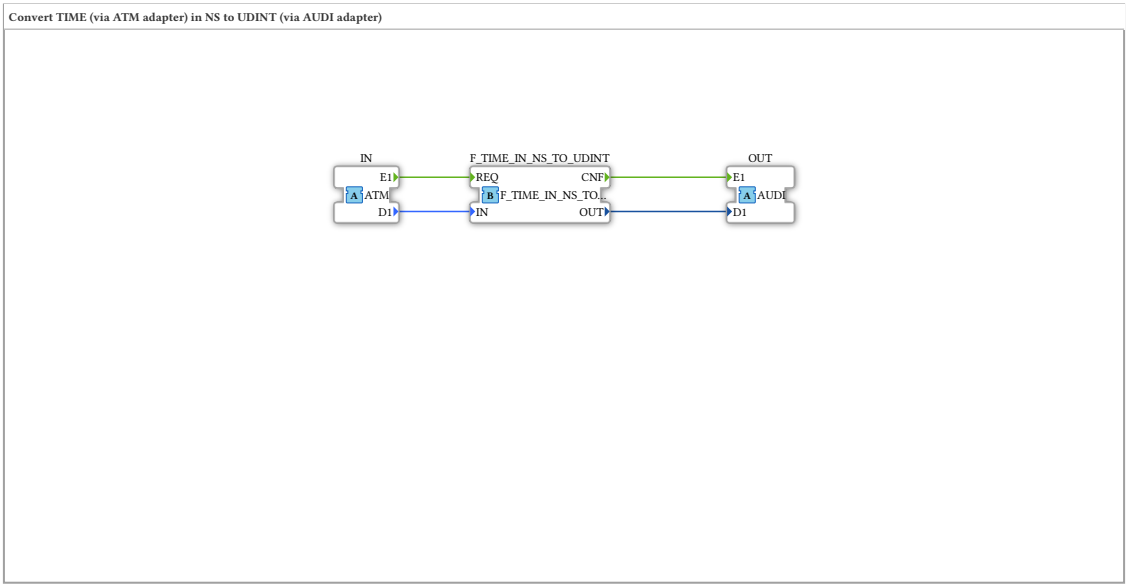
`ATM_IN_NS_TO_ALR` liefert eine standardkonforme, vollständig adapterbasierte Umrechnung eines `TIME`-Werts in Nanosekunden auf einen `LREAL`-Wert und fügt sich damit nahtlos in Netzwerke ein, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen.

ATM_IN_NS_TO_AUDI

Time value to convert – *ATM*



AUDI – TIME value of IN expressed in NS as UDINT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ATM_IN_NS_TO_AUDI** ist ein Composite-Baustein, der einen `TIME`-Wert (übertragen über den ATM-Adapter) in Nanosekunden auf einen `UDINT`-Wert (32-Bit-Ganzzahl (vorzeichenlos)) umrechnet und über den AUDI-Adapter ausgibt. Er gehört zur Blockfamilie `ATM_IN_<EINHEIT>_TO_<ZIELTYP>`, die dieselbe Umrechnung für alle vier Zeiteinheiten (MS, NS, S, US) und alle fünf Zielganzzahl-/Gleitkommatypen bereitstellt.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkt sichtbaren Ereignis-Eingänge. Die Ereignissteuerung erfolgt indirekt über den Adapter **IN** (Ereignis E1).

Ereignis-Ausgänge

- Keine direkt sichtbaren Ereignis-Ausgänge. Die Ereignissignalisierung erfolgt indirekt über den Adapter **OUT** (Ereignis E1).

Daten-Eingänge

- Keine direkt sichtbaren Daten-Eingänge. Der Eingabewert wird über den Adapter **IN** (Datenpunkt D1 vom Typ `TIME`) bereitgestellt.

Daten-Ausgänge

- Keine direkt sichtbaren Daten-Ausgänge. Der Ausgabewert wird über den Adapter **OUT** (Datenpunkt D1 vom Typ `UDINT`) bereitgestellt.

Adapter

Adapter	Rolle	Typ	Beschreibung
IN	Socket	adapter::types::unidirectional::ATM	Unidirektionaler `TIME`-Adapter als Eingangsschnittstelle.
OUT	Plug	adapter::types::unidirectional::AUDI	Unidirektionaler `UDINT`-Adapter als Ausgangsschnittstelle, Wert von `IN` in Nanosekunden ausgedrückt.

Funktionsweise

Der Baustein enthält intern einen einzelnen Funktionsbaustein `F_TIME_IN_NS_TO_UDINT` aus der IEC-61131-3-Standardbibliothek. Die Ablaufsteuerung erfolgt rein ereignisgesteuert:

1. Ein Ereignis am Eingangsadapter **IN.E1** triggert den Eingang **REQ** des internen Umrechnungsbausteins.
2. Der Datenwert **IN.D1** (`TIME`) wird an den Eingang **IN** von `F_TIME_IN_NS_TO_UDINT` übergeben.
3. `F_TIME_IN_NS_TO_UDINT` rechnet den Zeitwert in eine Nanosekunden-Zahl vom Typ `UDINT` um und signalisiert dies über seinen Ausgang **CNF**.
4. Dieses Ereignis wird an den Ausgangsadapter **OUT.E1** weitergeleitet.
5. Gleichzeitig wird der umgerechnete Wert (Ausgang **OUT** von `F_TIME_IN_NS_TO_UDINT`) an den Datenpunkt **OUT.D1** übergeben.

Der gesamte Vorgang erfolgt synchron und ohne Zwischenspeicherung – jede erfolgreiche Umrechnung löst sofort ein Ausgangsereignis aus.

Technische Besonderheiten

- **Einheitenumrechnung nach IEC 61131-3**: Der interne Baustein `F_TIME_IN_NS_TO_UDINT` interpretiert den intern in Nanosekunden gehaltenen `TIME`-Wert als Zahl in Nanosekunden und liefert das Ergebnis als `UDINT` (32-Bit-Ganzzahl (vorzeichenlos)).
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional ausgelegt. Eine Rückwärtskommunikation ist nicht vorgesehen.
- **Composite-Struktur ohne eigene Logik**: Der Baustein besteht ausschließlich aus einer Daten- und einer Ereignisverbindung zum internen Umrechnungsbaustein; es gibt kein eigenes ECC.
- **Leerer TypeHash**: Wie bei allen generierten Konvertierungs-Composites in diesem Paket ist `eclipse4diac::core::TypeHash` leer (`''`).

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Der Ablauf lässt sich als einfacher Schritt beschreiben:

Schritt	Aktion

1	Warten auf Ereignis an **IN.E1**
2	Umrechnung des Datenwerts von `TIME` (Nanosekunden) nach `UDINT` über `F_TIME_IN_NS_TO_UDINT`
3	Ausgabe des umgerechneten Werts an **OUT.D1** und Ereignis an **OUT.E1**
4	Rückkehr zu Schritt 1

Anwendungsszenarien

- **Anzeige/Weiterverarbeitung von Zeitdauern**: Eine über einen `ATM`-Adapter transportierte Zeitdauer (z. B. eine Zykluszeit oder Verzögerung) soll als Zahl in Nanosekunden an einen Adapter-Socket vom Typ `AUDI` weitergegeben werden, etwa zur Anzeige auf einem Panel oder zur Weiterverarbeitung in einer Berechnung.
- **Schnittstellenanpassung**: In Netzwerken, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen, ermöglicht der Baustein die nahtlose Umwandlung eines Zeit- in einen Zahlenadapter.

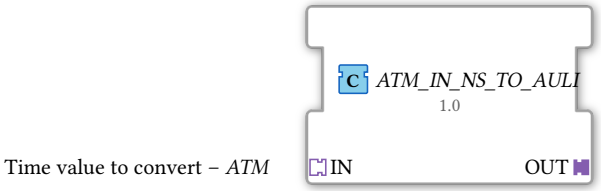
⚖️ Vergleich mit ähnlichen Bausteinen

- **Gleiche Einheit, andere Zieltypen**: ATM_IN_NS_TO_ADI, ATM_IN_NS_TO_ALI, ATM_IN_NS_TO_ALR, ATM_IN_NS_TO_AULI
- **Gleicher Zieltyp, andere Einheiten**: ATM_IN_MS_TO_AUDI, ATM_IN_S_TO_AUDI, ATM_IN_US_TO_AUDI
- **ATM_TIME_TO_TM / ATM_TM_TO_TIME**: reine Adapter-↔-Variablen-Durchreichung ohne Einheitenumrechnung, im Gegensatz zur numerischen Umrechnung von `ATM_IN_NS_TO_AUDI`.

Fazit

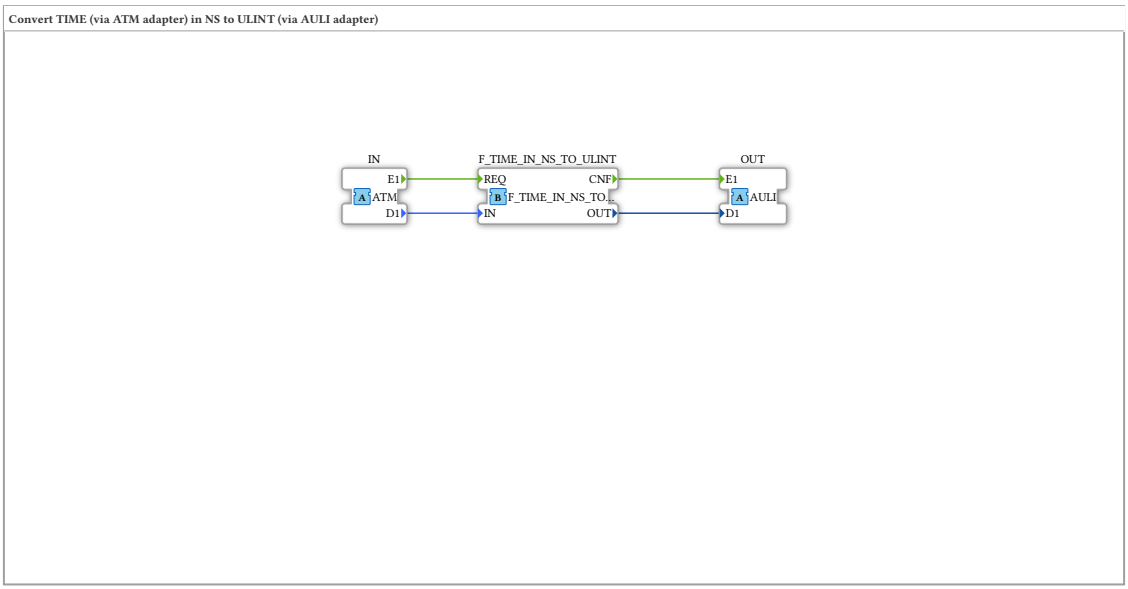
`ATM_IN_NS_TO_AUDI` liefert eine standardkonforme, vollständig adapterbasierte Umrechnung eines `TIME`-Werts in Nanosekunden auf einen `UDINT`-Wert und fügt sich damit nahtlos in Netzwerke ein, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen.

ATM_IN_NS_TO_AULI



Time value to convert – *ATM*

AULI – TIME value of IN expressed in NS as ULINT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ATM_IN_NS_TO_AULI** ist ein Composite-Baustein, der einen `TIME`-Wert (übertragen über den ATM-Adapter) in Nanosekunden auf einen `ULINT`-Wert (64-Bit-Ganzzahl (vorzeichenlos)) umrechnet und über den AULI-Adapter ausgibt. Er gehört zur Blockfamilie `ATM_IN_<EINHEIT>_TO_<ZIELTYP>`, die dieselbe Umrechnung für alle vier Zeiteinheiten (MS, NS, S, US) und alle fünf Zielganzzahl-/Gleitkommatypen bereitstellt.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkt sichtbaren Ereignis-Eingänge. Die Ereignissteuerung erfolgt indirekt über den Adapter **IN** (Ereignis E1).

Ereignis-Ausgänge

- Keine direkt sichtbaren Ereignis-Ausgänge. Die Ereignissignalisierung erfolgt indirekt über den Adapter **OUT** (Ereignis E1).

Daten-Eingänge

- Keine direkt sichtbaren Daten-Eingänge. Der Eingabewert wird über den Adapter **IN** (Datenpunkt D1 vom Typ `TIME`) bereitgestellt.

Daten-Ausgänge

- Keine direkt sichtbaren Daten-Ausgänge. Der Ausgabewert wird über den Adapter **OUT** (Datenpunkt D1 vom Typ `ULINT`) bereitgestellt.

Adapter

Adapter	Rolle	Typ	Beschreibung
IN	Socket	adapter::types::unidirectional::ATM	Unidirektionaler `TIME`-Adapter als Eingangsschnittstelle.
OUT	Plug	adapter::types::unidirectional::AULI	Unidirektionaler `ULINT`-Adapter als Ausgangsschnittstelle, Wert von `IN` in Nanosekunden ausgedrückt.

Funktionsweise

Der Baustein enthält intern einen einzelnen Funktionsbaustein `F_TIME_IN_NS_TO_ULINT` aus der IEC-61131-3-Standardbibliothek. Die Ablaufsteuerung erfolgt rein ereignisgesteuert:

1. Ein Ereignis am Eingangsadapter **IN.E1** triggert den Eingang **REQ** des internen Umrechnungsbausteins.
2. Der Datenwert **IN.D1** (`TIME`) wird an den Eingang **IN** von `F_TIME_IN_NS_TO_ULINT` übergeben.
3. `F_TIME_IN_NS_TO_ULINT` rechnet den Zeitwert in eine Nanosekunden-Zahl vom Typ `ULINT` um und signalisiert dies über seinen Ausgang **CNF**.
4. Dieses Ereignis wird an den Ausgangsadapter **OUT.E1** weitergeleitet.
5. Gleichzeitig wird der umgerechnete Wert (Ausgang **OUT** von `F_TIME_IN_NS_TO_ULINT`) an den Datenpunkt **OUT.D1** übergeben.

Der gesamte Vorgang erfolgt synchron und ohne Zwischenspeicherung – jede erfolgreiche Umrechnung löst sofort ein Ausgangsereignis aus.

Technische Besonderheiten

- **Einheitenumrechnung nach IEC 61131-3**: Der interne Baustein `F_TIME_IN_NS_TO_ULINT` interpretiert den intern in Nanosekunden gehaltenen `TIME`-Wert als Zahl in Nanosekunden und liefert das Ergebnis als `ULINT` (64-Bit-Ganzzahl (vorzeichenlos)).
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional ausgelegt. Eine Rückwärtskommunikation ist nicht vorgesehen.

- **Composite-Struktur ohne eigene Logik**: Der Baustein besteht ausschließlich aus einer Daten- und einer Ereignisverbindung zum internen Umrechnungsbaustein; es gibt kein eigenes ECC.
- **Leerer TypeHash**: Wie bei allen generierten Konvertierungs-Composites in diesem Paket ist `eclipse4diac::core::TypeHash` leer (`''`).

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Der Ablauf lässt sich als einfacher Schritt beschreiben:

Schritt	Aktion
1	Warten auf Ereignis an **IN.E1**
2	Umrechnung des Datenwerts von `TIME` (Nanosekunden) nach `ULINT` über `F_TIME_IN_NS_TO_ULINT`
3	Ausgabe des umgerechneten Werts an **OUT.D1** und Ereignis an **OUT.E1**
4	Rückkehr zu Schritt 1

Anwendungsszenarien

- **Anzeige/Weiterverarbeitung von Zeitdauern**: Eine über einen `ATM`-Adapter transportierte Zeitdauer (z. B. eine Zykluszeit oder Verzögerung) soll als Zahl in Nanosekunden an einen Adapter-Socket vom Typ `AULI` weitergegeben werden, etwa zur Anzeige auf einem Panel oder zur Weiterverarbeitung in einer Berechnung.
- **Schnittstellenanpassung**: In Netzwerken, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen, ermöglicht der Baustein die nahtlose Umwandlung eines Zeit- in einen Zahlenadapter.

⚖️ Vergleich mit ähnlichen Bausteinen

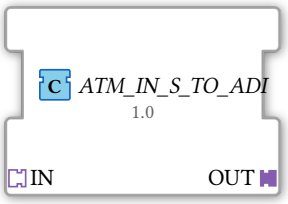
- **Gleiche Einheit, andere Zieltypen**: ATM_IN_NS_TO_ADI, ATM_IN_NS_TO_ALI, ATM_IN_NS_TO_ALR, ATM_IN_NS_TO_AUDI
- **Gleicher Zieltyp, andere Einheiten**: ATM_IN_MS_TO_AULI, ATM_IN_S_TO_AULI, ATM_IN_US_TO_AULI
- **ATM_TIME_TO_TM** / **ATM_TM_TO_TIME**: reine Adapter-↔-Variablen-Durchreichung ohne Einheitenumrechnung, im Gegensatz zur numerischen Umrechnung von `ATM_IN_NS_TO_AULI`.

Fazit

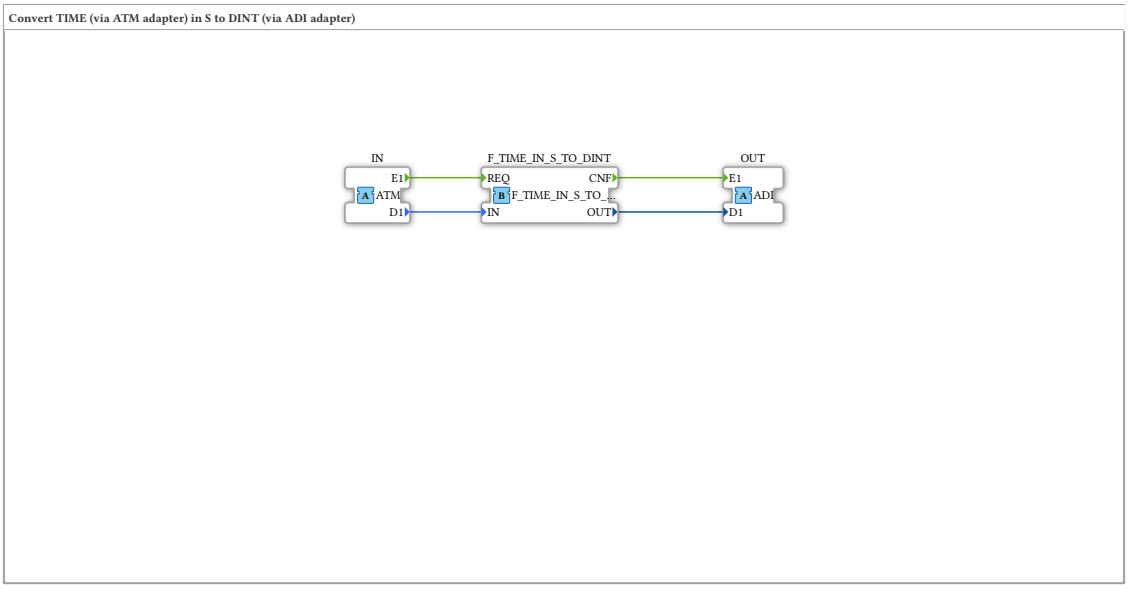
`ATM_IN_NS_TO_AULI` liefert eine standardkonforme, vollständig adapterbasierte Umrechnung eines `TIME`-Werts in Nanosekunden auf einen `ULINT`-Wert und fügt sich damit nahtlos in Netzwerke ein, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen.

ATM_IN_S_TO_ADI

Time value to convert – *ATM*



ADI – TIME value of IN expressed in S as DINT



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ATM_IN_S_TO_ADI** ist ein Composite-Baustein, der einen `TIME`-Wert (übertragen über den ATM-Adapter) in Sekunden auf einen `DINT`-Wert (32-Bit-Ganzzahl (vorzeichenbehaftet)) umrechnet und über den ADI-Adapter ausgibt. Er gehört zur Blockfamilie `ATM_IN_<EINHEIT>_TO_<ZIELTYP>`, die dieselbe Umrechnung für alle vier Zeiteinheiten (MS, NS, S, US) und alle fünf Zielganzzahl-/Gleitkommatypen bereitstellt.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkt sichtbaren Ereignis-Eingänge. Die Ereignissteuerung erfolgt indirekt über den Adapter **IN** (Ereignis E1).

Ereignis-Ausgänge

- Keine direkt sichtbaren Ereignis-Ausgänge. Die Ereignissignalisierung erfolgt indirekt über den Adapter **OUT** (Ereignis E1).

Daten-Eingänge

- Keine direkt sichtbaren Daten-Eingänge. Der Eingabewert wird über den Adapter **IN** (Datenpunkt D1 vom Typ `TIME`) bereitgestellt.

Daten-Ausgänge

- Keine direkt sichtbaren Daten-Ausgänge. Der Ausgabewert wird über den Adapter **OUT** (Datenpunkt D1 vom Typ `DINT`) bereitgestellt.

Adapter

Adapter	Rolle	Typ	Beschreibung
IN	Socket	adapter::types::unidirectional::ATM	Unidirektionaler `TIME`-Adapter als Eingangsschnittstelle.
OUT	Plug	adapter::types::unidirectional::ADI	Unidirektionaler `DINT`-Adapter als Ausgangsschnittstelle, Wert von `IN` in Sekunden ausgedrückt.

Funktionsweise

Der Baustein enthält intern einen einzelnen Funktionsbaustein `F_TIME_IN_S_TO_DINT` aus der IEC-61131-3-Standardbibliothek. Die Ablaufsteuerung erfolgt rein ereignisgesteuert:

1. Ein Ereignis am Eingangsadapter **IN.E1** triggert den Eingang **REQ** des internen Umrechnungsbausteins.
2. Der Datenwert **IN.D1** (`TIME`) wird an den Eingang **IN** von `F_TIME_IN_S_TO_DINT` übergeben.
3. `F_TIME_IN_S_TO_DINT` rechnet den Zeitwert in eine Sekunden-Zahl vom Typ `DINT` um und signalisiert dies über seinen Ausgang **CNF**.
4. Dieses Ereignis wird an den Ausgangsadapter **OUT.E1** weitergeleitet.
5. Gleichzeitig wird der umgerechnete Wert (Ausgang **OUT** von `F_TIME_IN_S_TO_DINT`) an den Datenpunkt **OUT.D1** übergeben.

Der gesamte Vorgang erfolgt synchron und ohne Zwischenspeicherung – jede erfolgreiche Umrechnung löst sofort ein Ausgangsereignis aus.

Technische Besonderheiten

- **Einheitenumrechnung nach IEC 61131-3**: Der interne Baustein `F_TIME_IN_S_TO_DINT` interpretiert den intern in Nanosekunden gehaltenen `TIME`-Wert als Zahl in Sekunden und liefert das Ergebnis als `DINT` (32-Bit-Ganzzahl (vorzeichenbehaftet)).
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional ausgelegt. Eine Rückwärtskommunikation ist nicht vorgesehen.
- **Composite-Struktur ohne eigene Logik**: Der Baustein besteht ausschließlich aus einer Daten- und einer Ereignisverbindung zum internen Umrechnungsbaustein; es gibt kein eigenes ECC.
- **Leerer TypeHash**: Wie bei allen generierten Konvertierungs-Composites in diesem Paket ist `eclipse4diac::core::TypeHash` leer (`''`).

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Der Ablauf lässt sich als einfacher Schritt beschreiben:

Schritt	Aktion
1	Warten auf Ereignis an **IN.E1**
2	Umrechnung des Datenwerts von `TIME` (Sekunden) nach `DINT` über `F_TIME_IN_S_TO_DINT`
3	Ausgabe des umgerechneten Werts an **OUT.D1** und Ereignis an **OUT.E1**
4	Rückkehr zu Schritt 1

Anwendungsszenarien

- **Anzeige/Weiterverarbeitung von Zeitdauern**: Eine über einen `ATM`-Adapter transportierte Zeitdauer (z. B. eine Zykluszeit oder Verzögerung) soll als Zahl in Sekunden an einen Adapter-Socket vom Typ `ADI` weitergegeben werden, etwa zur Anzeige auf einem Panel oder zur Weiterverarbeitung in einer Berechnung.
- **Schnittstellenanpassung**: In Netzwerken, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen, ermöglicht der Baustein die nahtlose Umwandlung eines Zeit- in einen Zahlenadapter.

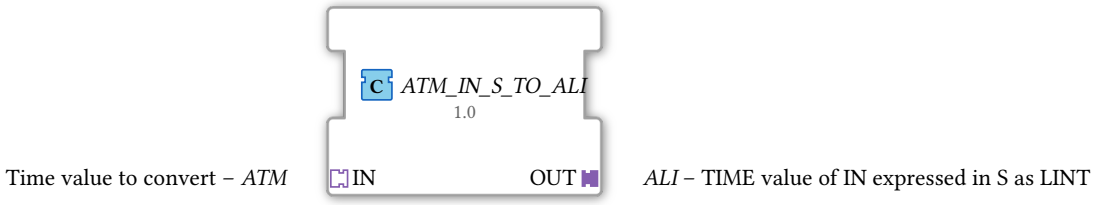
⚖️ Vergleich mit ähnlichen Bausteinen

- **Gleiche Einheit, andere Zieltypen**: ATM_IN_S_TO_ALI, ATM_IN_S_TO_ALR, ATM_IN_S_TO_AUDI, ATM_IN_S_TO_AULI
- **Gleicher Zieltyp, andere Einheiten**: ATM_IN_MS_TO_ADI, ATM_IN_NS_TO_ADI, ATM_IN_US_TO_ADI
- **ATM_TIME_TO_TM** / **ATM_TM_TO_TIME**: reine Adapter-↔-Variablen-Durchreichung ohne Einheitenumrechnung, im Gegensatz zur numerischen Umrechnung von `ATM_IN_S_TO_ADI`.

Fazit

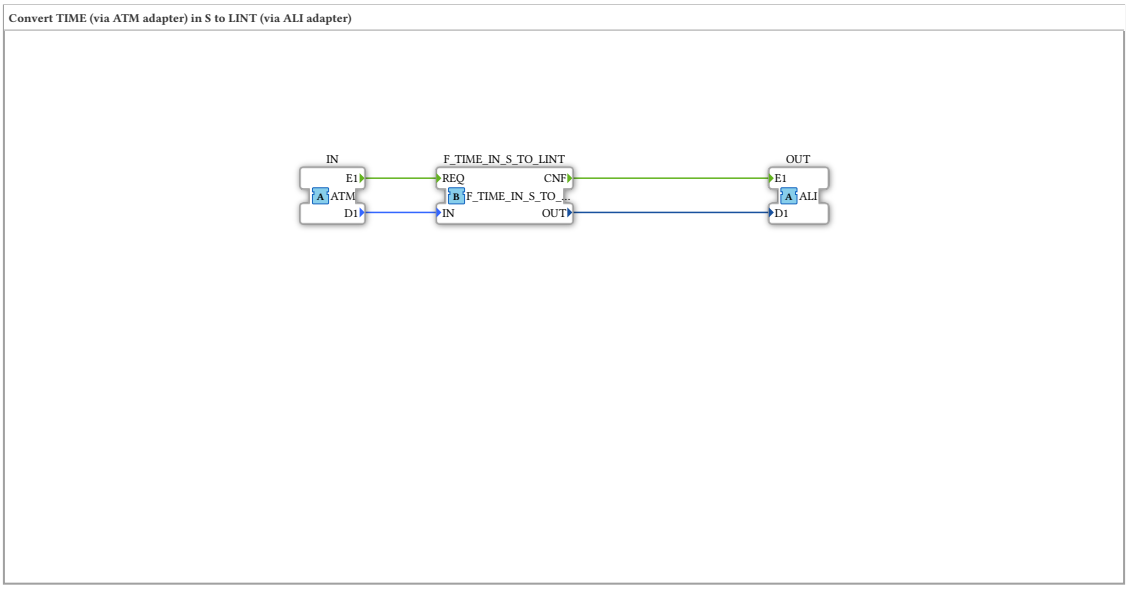
`ATM_IN_S_TO_ADI` liefert eine standardkonforme, vollständig adapterbasierte Umrechnung eines `TIME`-Werts in Sekunden auf einen `DINT`-Wert und fügt sich damit nahtlos in Netzwerke ein, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen.

ATM_IN_S_TO_ALI



Time value to convert – *ATM*

ALI – TIME value of IN expressed in S as LINT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ATM_IN_S_TO_ALI** ist ein Composite-Baustein, der einen `TIME`-Wert (übertragen über den ATM-Adapter) in Sekunden auf einen `LINT`-Wert (64-Bit-Ganzzahl (vorzeichenbehaftet)) umrechnet und über den ALI-Adapter ausgibt. Er gehört zur Blockfamilie `ATM_IN_<EINHEIT>_TO_<ZIELTYP>`, die dieselbe Umrechnung für alle vier Zeiteinheiten (MS, NS, S, US) und alle fünf Zielganzzahl-/Gleitkommatypen bereitstellt.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkt sichtbaren Ereignis-Eingänge. Die Ereignissteuerung erfolgt indirekt über den Adapter **IN** (Ereignis E1).

Ereignis-Ausgänge

- Keine direkt sichtbaren Ereignis-Ausgänge. Die Ereignissignalisierung erfolgt indirekt über den Adapter **OUT** (Ereignis E1).

Daten-Eingänge

- Keine direkt sichtbaren Daten-Eingänge. Der Eingabewert wird über den Adapter **IN** (Datenpunkt D1 vom Typ `TIME`) bereitgestellt.

Daten-Ausgänge

- Keine direkt sichtbaren Daten-Ausgänge. Der Ausgabewert wird über den Adapter **OUT** (Datenpunkt D1 vom Typ `LINT`) bereitgestellt.

Adapter

Adapter	Rolle	Typ	Beschreibung
IN	Socket	adapter::types::unidirectional::ATM	Unidirektionaler `TIME`-Adapter als Eingangsschnittstelle.
OUT	Plug	adapter::types::unidirectional::ALI	Unidirektionaler `LINT`-Adapter als Ausgangsschnittstelle, Wert von `IN` in Sekunden ausgedrückt.

Funktionsweise

Der Baustein enthält intern einen einzelnen Funktionsbaustein `F_TIME_IN_S_TO_LINT` aus der IEC-61131-3-Standardbibliothek. Die Ablaufsteuerung erfolgt rein ereignisgesteuert:

1. Ein Ereignis am Eingangsadapter **IN.E1** triggert den Eingang **REQ** des internen Umrechnungsbausteins.
2. Der Datenwert **IN.D1** (`TIME`) wird an den Eingang **IN** von `F_TIME_IN_S_TO_LINT` übergeben.
3. `F_TIME_IN_S_TO_LINT` rechnet den Zeitwert in eine Sekunden-Zahl vom Typ `LINT` um und signalisiert dies über seinen Ausgang **CNF**.
4. Dieses Ereignis wird an den Ausgangsadapter **OUT.E1** weitergeleitet.
5. Gleichzeitig wird der umgerechnete Wert (Ausgang **OUT** von `F_TIME_IN_S_TO_LINT`) an den Datenpunkt **OUT.D1** übergeben.

Der gesamte Vorgang erfolgt synchron und ohne Zwischenspeicherung – jede erfolgreiche Umrechnung löst sofort ein Ausgangsereignis aus.

Technische Besonderheiten

- **Einheitenumrechnung nach IEC 61131-3**: Der interne Baustein `F_TIME_IN_S_TO_LINT` interpretiert den intern in Nanosekunden gehaltenen `TIME`-Wert als Zahl in Sekunden und liefert das Ergebnis als `LINT` (64-Bit-Ganzzahl (vorzeichenbehaftet)).
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional ausgelegt. Eine Rückwärtskommunikation ist nicht vorgesehen.
- **Composite-Struktur ohne eigene Logik**: Der Baustein besteht ausschließlich aus einer Daten- und einer Ereignisverbindung zum internen Umrechnungsbaustein; es gibt kein eigenes ECC.
- **Leerer TypeHash**: Wie bei allen generierten Konvertierungs-Composites in diesem Paket ist `eclipse4diac::core::TypeHash` leer (`''`).

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Der Ablauf lässt sich als einfacher Schritt beschreiben:

Schritt	Aktion
1	Warten auf Ereignis an **IN.E1**
2	Umrechnung des Datenwerts von `TIME` (Sekunden) nach `LINT` über `F_TIME_IN_S_TO_LINT`
3	Ausgabe des umgerechneten Werts an **OUT.D1** und Ereignis an **OUT.E1**
4	Rückkehr zu Schritt 1

Anwendungsszenarien

- **Anzeige/Weiterverarbeitung von Zeitdauern**: Eine über einen `ATM`-Adapter transportierte Zeitdauer (z. B. eine Zykluszeit oder Verzögerung) soll als Zahl in Sekunden an einen Adapter-Socket vom Typ `ALI` weitergegeben werden, etwa zur Anzeige auf einem Panel oder zur Weiterverarbeitung in einer Berechnung.

- **Schnittstellenanpassung**: In Netzwerken, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen, ermöglicht der Baustein die nahtlose Umwandlung eines Zeit- in einen Zahlenadapter.

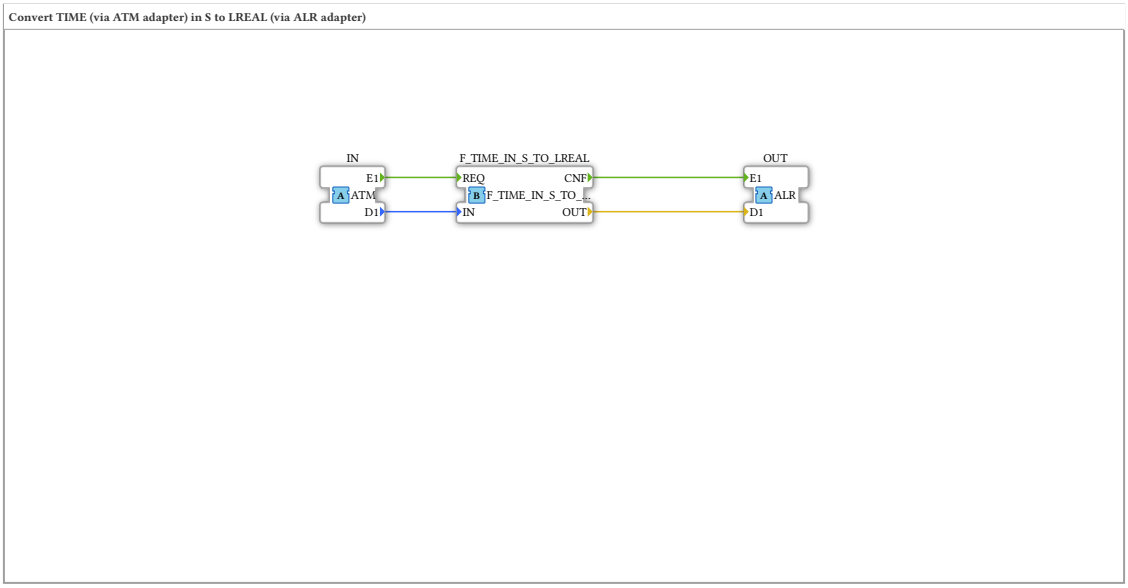
⚖️ Vergleich mit ähnlichen Bausteinen

- **Gleiche Einheit, andere Zieltypen**: ATM_IN_S_TO_ADI, ATM_IN_S_TO_ALR, ATM_IN_S_TO_AUDI, ATM_IN_S_TO_AULI
- **Gleicher Zieltyp, andere Einheiten**: ATM_IN_MS_TO_ALI, ATM_IN_NS_TO_ALI, ATM_IN_US_TO_ALI
- **ATM_TIME_TO_TM** / **ATM_TM_TO_TIME**: reine Adapter-↔-Variablen-Durchreichung ohne Einheitenumrechnung, im Gegensatz zur numerischen Umrechnung von `ATM_IN_S_TO_ALI`.

Fazit

`ATM_IN_S_TO_ALI` liefert eine standardkonforme, vollständig adapterbasierte Umrechnung eines `TIME`-Werts in Sekunden auf einen `LINT`-Wert und fügt sich damit nahtlos in Netzwerke ein, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen.

ATM_IN_S_TO_ALR



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ATM_IN_S_TO_ALR** ist ein Composite-Baustein, der einen `TIME`-Wert (übertragen über den ATM-Adapter) in Sekunden auf einen `LREAL`-Wert (64-Bit-Gleitkommazahl) umrechnet und über den ALR-Adapter ausgibt. Er gehört zur Blockfamilie `ATM_IN_<EINHEIT>_TO_<ZIELTYP>`, die dieselbe Umrechnung für alle vier Zeiteinheiten (MS, NS, S, US) und alle fünf Zielganzzahl-/Gleitkommatypen bereitstellt.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkt sichtbaren Ereignis-Eingänge. Die Ereignissteuerung erfolgt indirekt über den Adapter **IN** (Ereignis E1).

Ereignis-Ausgänge

- Keine direkt sichtbaren Ereignis-Ausgänge. Die Ereignissignalisierung erfolgt indirekt über den Adapter **OUT** (Ereignis E1).

Daten-Eingänge

- Keine direkt sichtbaren Daten-Eingänge. Der Eingabewert wird über den Adapter **IN** (Datenpunkt D1 vom Typ `TIME`) bereitgestellt.

Daten-Ausgänge

- Keine direkt sichtbaren Daten-Ausgänge. Der Ausgabewert wird über den Adapter **OUT** (Datenpunkt D1 vom Typ `LREAL`) bereitgestellt.

Adapter

Adapter	Rolle	Typ	Beschreibung
IN	Socket	adapter::types::unidirectional::ATM	Unidirektionaler `TIME`-Adapter als Eingangsschnittstelle.
OUT	Plug	adapter::types::unidirectional::ALR	Unidirektionaler `LREAL`-Adapter als Ausgangsschnittstelle, Wert von `IN` in Sekunden ausgedrückt.

Funktionsweise

Der Baustein enthält intern einen einzelnen Funktionsbaustein `F_TIME_IN_S_TO_LREAL` aus der IEC-61131-3-Standardbibliothek. Die Ablaufsteuerung erfolgt rein ereignisgesteuert:

1. Ein Ereignis am Eingangsadapter **IN.E1** triggert den Eingang **REQ** des internen Umrechnungsbausteins.
2. Der Datenwert **IN.D1** (`TIME`) wird an den Eingang **IN** von `F_TIME_IN_S_TO_LREAL` übergeben.
3. `F_TIME_IN_S_TO_LREAL` rechnet den Zeitwert in eine Sekunden-Zahl vom Typ `LREAL` um und signalisiert dies über seinen Ausgang **CNF**.
4. Dieses Ereignis wird an den Ausgangsadapter **OUT.E1** weitergeleitet.
5. Gleichzeitig wird der umgerechnete Wert (Ausgang **OUT** von `F_TIME_IN_S_TO_LREAL`) an den Datenpunkt **OUT.D1** übergeben.

Der gesamte Vorgang erfolgt synchron und ohne Zwischenspeicherung – jede erfolgreiche Umrechnung löst sofort ein Ausgangsereignis aus.

Technische Besonderheiten

- **Einheitenumrechnung nach IEC 61131-3**: Der interne Baustein `F_TIME_IN_S_TO_LREAL` interpretiert den intern in Nanosekunden gehaltenen `TIME`-Wert als Zahl in Sekunden und liefert das Ergebnis als `LREAL` (64-Bit-Gleitkommazahl).
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional ausgelegt. Eine Rückwärtskommunikation ist nicht vorgesehen.
- **Composite-Struktur ohne eigene Logik**: Der Baustein besteht ausschließlich aus einer Daten- und einer Ereignisverbindung zum internen Umrechnungsbaustein; es gibt kein eigenes ECC.
- **Leerer TypeHash**: Wie bei allen generierten Konvertierungs-Composites in diesem Paket ist `eclipse4diac::core::TypeHash` leer (`''`).

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Der Ablauf lässt sich als einfacher Schritt beschreiben:

Schritt	Aktion
1	Warten auf Ereignis an **IN.E1**
2	Umrechnung des Datenwerts von `TIME` (Sekunden) nach `LREAL` über `F_TIME_IN_S_TO_LREAL`
3	Ausgabe des umgerechneten Werts an **OUT.D1** und Ereignis an **OUT.E1**

4	Rückkehr zu Schritt 1

Anwendungsszenarien

- **Anzeige/Weiterverarbeitung von Zeitdauern**: Eine über einen `ATM`-Adapter transportierte Zeitdauer (z. B. eine Zykluszeit oder Verzögerung) soll als Zahl in Sekunden an einen Adapter-Socket vom Typ `ALR` weitergegeben werden, etwa zur Anzeige auf einem Panel oder zur Weiterverarbeitung in einer Berechnung.
- **Schnittstellenanpassung**: In Netzwerken, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen, ermöglicht der Baustein die nahtlose Umwandlung eines Zeit- in einen Zahlenadapter.

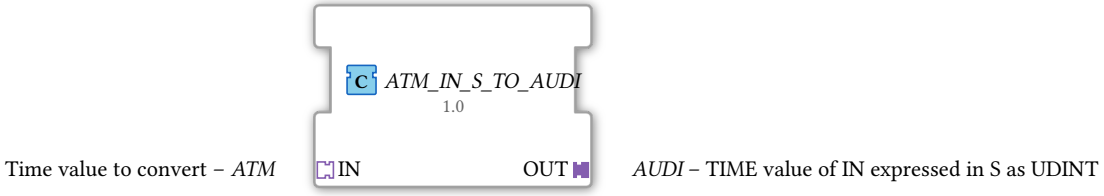
⚖️ Vergleich mit ähnlichen Bausteinen

- **Gleiche Einheit, andere Zieltypen**: ATM_IN_S_TO_ADI, ATM_IN_S_TO_ALI, ATM_IN_S_TO_AUDI, ATM_IN_S_TO_AULI
- **Gleicher Zieltyp, andere Einheiten**: ATM_IN_MS_TO_ALR, ATM_IN_NS_TO_ALR, ATM_IN_US_TO_ALR
- **ATM_TIME_TO_TM** / **ATM_TM_TO_TIME**: reine Adapter-↔-Variablen-Durchreichung ohne Einheitenumrechnung, im Gegensatz zur numerischen Umrechnung von `ATM_IN_S_TO_ALR`.

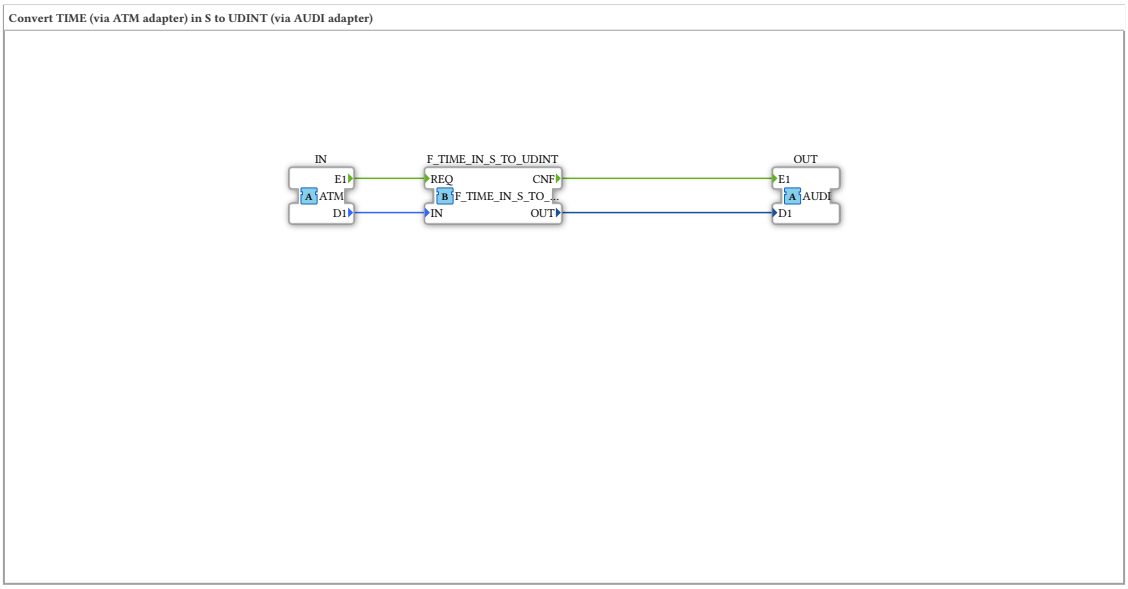
Fazit

`ATM_IN_S_TO_ALR` liefert eine standardkonforme, vollständig adapterbasierte Umrechnung eines `TIME`-Werts in Sekunden auf einen `LREAL`-Wert und fügt sich damit nahtlos in Netzwerke ein, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen.

ATM_IN_S_TO_AUDI



Time value to convert – *ATM* — *AUDI* – TIME value of IN expressed in S as UDINT



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ATM_IN_S_TO_AUDI** ist ein Composite-Baustein, der einen `TIME`-Wert (übertragen über den ATM-Adapter) in Sekunden auf einen `UDINT`-Wert (32-Bit-Ganzzahl (vorzeichenlos)) umrechnet

und über den AUDI-Adapter ausgibt. Er gehört zur Blockfamilie `ATM_IN_<EINHEIT>_TO_<ZIELTYP>`, die dieselbe Umrechnung für alle vier Zeiteinheiten (MS, NS, S, US) und alle fünf Zielganzzahl-/Gleitkommatypen bereitstellt.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkt sichtbaren Ereignis-Eingänge. Die Ereignissteuerung erfolgt indirekt über den Adapter **IN** (Ereignis E1).

Ereignis-Ausgänge

- Keine direkt sichtbaren Ereignis-Ausgänge. Die Ereignissignalisierung erfolgt indirekt über den Adapter **OUT** (Ereignis E1).

Daten-Eingänge

- Keine direkt sichtbaren Daten-Eingänge. Der Eingabewert wird über den Adapter **IN** (Datenpunkt D1 vom Typ `TIME`) bereitgestellt.

Daten-Ausgänge

- Keine direkt sichtbaren Daten-Ausgänge. Der Ausgabewert wird über den Adapter **OUT** (Datenpunkt D1 vom Typ `UDINT`) bereitgestellt.

Adapter

Adapter	Rolle	Typ	Beschreibung
IN	Socket	adapter::types::unidirectional::ATM	Unidirektionaler `TIME`-Adapter als Eingangsschnittstelle.
OUT	Plug	adapter::types::unidirectional::AUDI	Unidirektionaler `UDINT`-Adapter als Ausgangsschnittstelle, Wert von `IN` in Sekunden ausgedrückt.

Funktionsweise

Der Baustein enthält intern einen einzelnen Funktionsbaustein `F_TIME_IN_S_TO_UDINT` aus der IEC-61131-3-Standardbibliothek. Die Ablaufsteuerung erfolgt rein ereignisgesteuert:

1. Ein Ereignis am Eingangsadapter **IN.E1** triggert den Eingang **REQ** des internen Umrechnungsbausteins.
2. Der Datenwert **IN.D1** (`TIME`) wird an den Eingang **IN** von `F_TIME_IN_S_TO_UDINT` übergeben.
3. `F_TIME_IN_S_TO_UDINT` rechnet den Zeitwert in eine Sekunden-Zahl vom Typ `UDINT` um und signalisiert dies über seinen Ausgang **CNF**.
4. Dieses Ereignis wird an den Ausgangsadapter **OUT.E1** weitergeleitet.
5. Gleichzeitig wird der umgerechnete Wert (Ausgang **OUT** von `F_TIME_IN_S_TO_UDINT`) an den Datenpunkt **OUT.D1** übergeben.

Der gesamte Vorgang erfolgt synchron und ohne Zwischenspeicherung – jede erfolgreiche Umrechnung löst sofort ein Ausgangsereignis aus.

Technische Besonderheiten

- **Einheitenumrechnung nach IEC 61131-3**: Der interne Baustein `F_TIME_IN_S_TO_UDINT` interpretiert den intern in Nanosekunden gehaltenen `TIME`-Wert als Zahl in Sekunden und liefert das Ergebnis als `UDINT` (32-Bit-Ganzzahl (vorzeichenlos)).
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional ausgelegt. Eine Rückwärtskommunikation ist nicht vorgesehen.
- **Composite-Struktur ohne eigene Logik**: Der Baustein besteht ausschließlich aus einer Daten- und einer Ereignisverbindung zum internen Umrechnungsbaustein; es gibt kein eigenes ECC.
- **Leerer TypeHash**: Wie bei allen generierten Konvertierungs-Composites in diesem Paket ist `eclipse4diac::core::TypeHash` leer (`''`).

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Der Ablauf lässt sich als einfacher Schritt beschreiben:

Schritt	Aktion
1	Warten auf Ereignis an **IN.E1**
2	Umrechnung des Datenwerts von `TIME` (Sekunden) nach `UDINT` über `F_TIME_IN_S_TO_UDINT`
3	Ausgabe des umgerechneten Werts an **OUT.D1** und Ereignis an **OUT.E1**
4	Rückkehr zu Schritt 1

Anwendungsszenarien

- **Anzeige/Weiterverarbeitung von Zeitdauern**: Eine über einen `ATM`-Adapter transportierte Zeitdauer (z. B. eine Zykluszeit oder Verzögerung) soll als Zahl in Sekunden an einen Adapter-Socket vom Typ `AUDI` weitergegeben werden, etwa zur Anzeige auf einem Panel oder zur Weiterverarbeitung in einer Berechnung.
- **Schnittstellenanpassung**: In Netzwerken, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen, ermöglicht der Baustein die nahtlose Umwandlung eines Zeit- in einen Zahlenadapter.

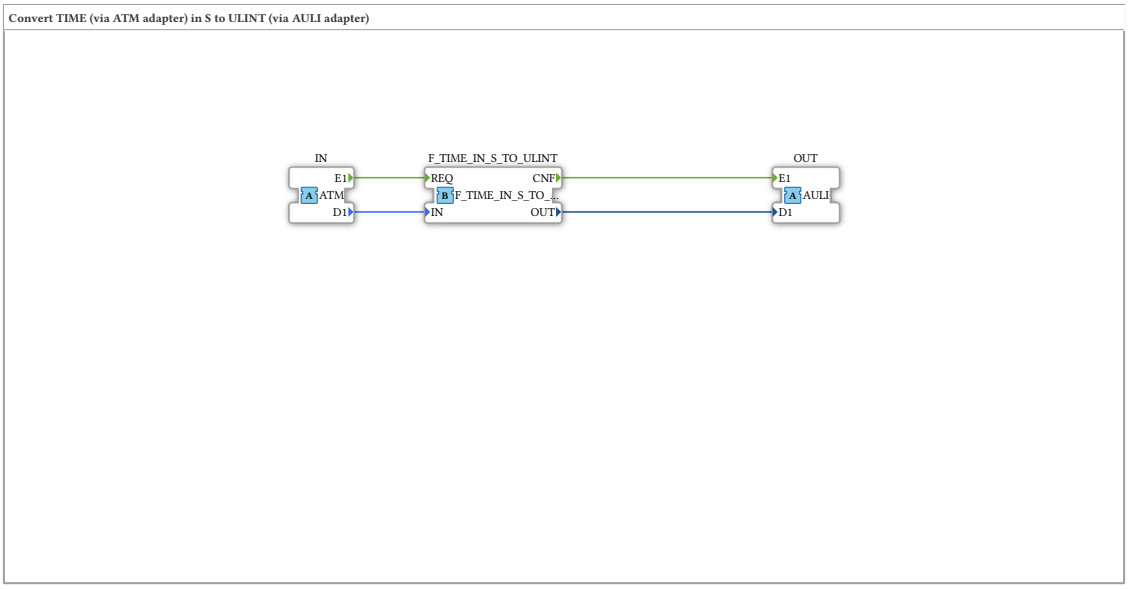
⚖️ Vergleich mit ähnlichen Bausteinen

- **Gleiche Einheit, andere Zieltypen**: ATM_IN_S_TO_ADI, ATM_IN_S_TO_ALI, ATM_IN_S_TO_ALR, ATM_IN_S_TO_AULI
- **Gleicher Zieltyp, andere Einheiten**: ATM_IN_MS_TO_AUDI, ATM_IN_NS_TO_AUDI, ATM_IN_US_TO_AUDI
- **ATM_TIME_TO_TM / ATM_TM_TO_TIME**: reine Adapter-↔-Variablen-Durchreichung ohne Einheitenumrechnung, im Gegensatz zur numerischen Umrechnung von `ATM_IN_S_TO_AUDI`.

Fazit

`ATM_IN_S_TO_AUDI` liefert eine standardkonforme, vollständig adapterbasierte Umrechnung eines `TIME`-Werts in Sekunden auf einen `UDINT`-Wert und fügt sich damit nahtlos in Netzwerke ein, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen.

ATM_IN_S_TO_AULI



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ATM_IN_S_TO_AULI** ist ein Composite-Baustein, der einen `TIME`-Wert (übertragen über den ATM-Adapter) in Sekunden auf einen `ULINT`-Wert (64-Bit-Ganzzahl (vorzeichenlos)) umrechnet und über den AULI-Adapter ausgibt. Er gehört zur Blockfamilie `ATM_IN_<EINHEIT>_TO_<ZIELTYP>`, die dieselbe Umrechnung für alle vier Zeiteinheiten (MS, NS, S, US) und alle fünf Zielganzzahl-/Gleitkommatypen bereitstellt.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkt sichtbaren Ereignis-Eingänge. Die Ereignissteuerung erfolgt indirekt über den Adapter **IN** (Ereignis E1).

Ereignis-Ausgänge

- Keine direkt sichtbaren Ereignis-Ausgänge. Die Ereignissignalisierung erfolgt indirekt über den Adapter **OUT** (Ereignis E1).

Daten-Eingänge

- Keine direkt sichtbaren Daten-Eingänge. Der Eingabewert wird über den Adapter **IN** (Datenpunkt D1 vom Typ `TIME`) bereitgestellt.

Daten-Ausgänge

- Keine direkt sichtbaren Daten-Ausgänge. Der Ausgabewert wird über den Adapter **OUT** (Datenpunkt D1 vom Typ `ULINT`) bereitgestellt.

Adapter

Adapter	Rolle	Typ	Beschreibung
IN	Socket	adapter::types::unidirectional::ATM	Unidirektionaler `TIME`-Adapter als Eingangsschnittstelle.
OUT	Plug	adapter::types::unidirectional::AULI	Unidirektionaler `ULINT`-Adapter als Ausgangsschnittstelle, Wert von `IN` in Sekunden ausgedrückt.

Funktionsweise

Der Baustein enthält intern einen einzelnen Funktionsbaustein `F_TIME_IN_S_TO_ULINT` aus der IEC-61131-3-Standardbibliothek. Die Ablaufsteuerung erfolgt rein ereignisgesteuert:

1. Ein Ereignis am Eingangsadapter **IN.E1** triggert den Eingang **REQ** des internen Umrechnungsbausteins.
2. Der Datenwert **IN.D1** (`TIME`) wird an den Eingang **IN** von `F_TIME_IN_S_TO_ULINT` übergeben.
3. `F_TIME_IN_S_TO_ULINT` rechnet den Zeitwert in eine Sekunden-Zahl vom Typ `ULINT` um und signalisiert dies über seinen Ausgang **CNF**.
4. Dieses Ereignis wird an den Ausgangsadapter **OUT.E1** weitergeleitet.
5. Gleichzeitig wird der umgerechnete Wert (Ausgang **OUT** von `F_TIME_IN_S_TO_ULINT`) an den Datenpunkt **OUT.D1** übergeben.

Der gesamte Vorgang erfolgt synchron und ohne Zwischenspeicherung – jede erfolgreiche Umrechnung löst sofort ein Ausgangsereignis aus.

Technische Besonderheiten

- **Einheitenumrechnung nach IEC 61131-3**: Der interne Baustein `F_TIME_IN_S_TO_ULINT` interpretiert den intern in Nanosekunden gehaltenen `TIME`-Wert als Zahl in Sekunden und liefert das Ergebnis als `ULINT` (64-Bit-Ganzzahl (vorzeichenlos)).
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional ausgelegt. Eine Rückwärtskommunikation ist nicht vorgesehen.
- **Composite-Struktur ohne eigene Logik**: Der Baustein besteht ausschließlich aus einer Daten- und einer Ereignisverbindung zum internen Umrechnungsbaustein; es gibt kein eigenes ECC.
- **Leerer TypeHash**: Wie bei allen generierten Konvertierungs-Composites in diesem Paket ist `eclipse4diac::core::TypeHash` leer (`''`).

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Der Ablauf lässt sich als einfacher Schritt beschreiben:

Schritt	Aktion
1	Warten auf Ereignis an **IN.E1**
2	Umrechnung des Datenwerts von `TIME` (Sekunden) nach `ULINT` über `F_TIME_IN_S_TO_ULINT`
3	Ausgabe des umgerechneten Werts an **OUT.D1** und Ereignis an **OUT.E1**
4	Rückkehr zu Schritt 1

Anwendungsszenarien

- **Anzeige/Weiterverarbeitung von Zeitdauern**: Eine über einen `ATM`-Adapter transportierte Zeitdauer (z. B. eine Zykluszeit oder Verzögerung) soll als Zahl in Sekunden an einen Adapter-Socket vom Typ `AULI` weitergegeben werden, etwa zur Anzeige auf einem Panel oder zur Weiterverarbeitung in einer Berechnung.
- **Schnittstellenanpassung**: In Netzwerken, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen, ermöglicht der Baustein die nahtlose Umwandlung eines Zeit- in einen Zahlenadapter.

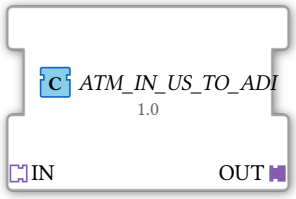
⚖️ Vergleich mit ähnlichen Bausteinen

- **Gleiche Einheit, andere Zieltypen**: ATM_IN_S_TO_ADI, ATM_IN_S_TO_ALI, ATM_IN_S_TO_ALR, ATM_IN_S_TO_AUDI
- **Gleicher Zieltyp, andere Einheiten**: ATM_IN_MS_TO_AULI, ATM_IN_NS_TO_AULI, ATM_IN_US_TO_AULI
- **ATM_TIME_TO_TM** / **ATM_TM_TO_TIME**: reine Adapter-↔-Variablen-Durchreichung ohne Einheitenumrechnung, im Gegensatz zur numerischen Umrechnung von `ATM_IN_S_TO_AULI`.

Fazit

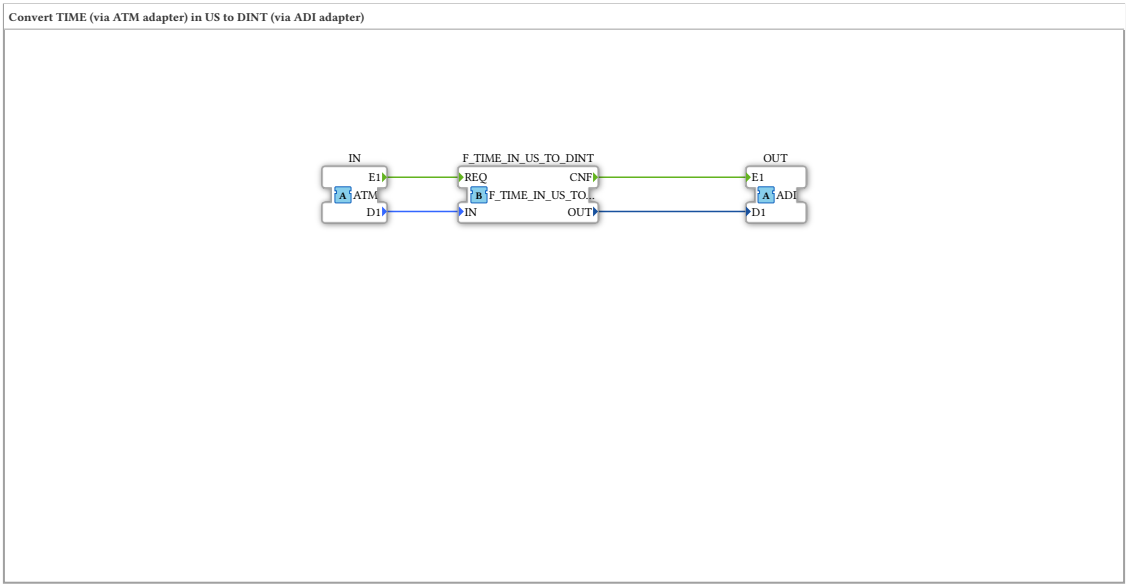
`ATM_IN_S_TO_AULI` liefert eine standardkonforme, vollständig adapterbasierte Umrechnung eines `TIME`-Werts in Sekunden auf einen `ULINT`-Wert und fügt sich damit nahtlos in Netzwerke ein, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen.

ATM_IN_US_TO_ADI



Time value to convert – *ATM*

ADI – TIME value of IN expressed in US as DINT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ATM_IN_US_TO_ADI** ist ein Composite-Baustein, der einen `TIME`-Wert (übertragen über den ATM-Adapter) in Mikrosekunden auf einen `DINT`-Wert (32-Bit-Ganzzahl (vorzeichenbehaftet)) umrechnet und über den ADI-Adapter ausgibt. Er gehört zur Blockfamilie `ATM_IN_<EINHEIT>_TO_<ZIELTYP>`, die dieselbe Umrechnung für alle vier Zeiteinheiten (MS, NS, S, US) und alle fünf Zielganzzahl-/Gleitkommatypen bereitstellt.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkt sichtbaren Ereignis-Eingänge. Die Ereignissteuerung erfolgt indirekt über den Adapter **IN** (Ereignis E1).

Ereignis-Ausgänge

- Keine direkt sichtbaren Ereignis-Ausgänge. Die Ereignissignalisierung erfolgt indirekt über den Adapter **OUT** (Ereignis E1).

Daten-Eingänge

- Keine direkt sichtbaren Daten-Eingänge. Der Eingabewert wird über den Adapter **IN** (Datenpunkt D1 vom Typ `TIME`) bereitgestellt.

Daten-Ausgänge

- Keine direkt sichtbaren Daten-Ausgänge. Der Ausgabewert wird über den Adapter **OUT** (Datenpunkt D1 vom Typ `DINT`) bereitgestellt.

Adapter

Adapter	Rolle	Typ	Beschreibung
IN	Socket	adapter::types::unidirectional::ATM	Unidirektionaler TIME-Adapter als Eingangsschnittstelle.
OUT	Plug	adapter::types::unidirectional::ADI	Unidirektionaler DINT-Adapter als Ausgangsschnittstelle, Wert von IN in Mikrosekunden ausgedrückt.

Funktionsweise

Der Baustein enthält intern einen einzelnen Funktionsbaustein F_TIME_IN_US_TO_DINT aus der IEC-61131-3-Standardbibliothek. Die Ablaufsteuerung erfolgt rein ereignisgesteuert:

1. Ein Ereignis am Eingangsadapter **IN.E1** triggert den Eingang **REQ** des internen Umrechnungsbausteins.
2. Der Datenwert **IN.D1** (TIME) wird an den Eingang **IN** von F_TIME_IN_US_TO_DINT übergeben.
3. F_TIME_IN_US_TO_DINT rechnet den Zeitwert in eine Mikrosekunden-Zahl vom Typ DINT um und signalisiert dies über seinen Ausgang **CNF**.
4. Dieses Ereignis wird an den Ausgangsadapter **OUT.E1** weitergeleitet.
5. Gleichzeitig wird der umgerechnete Wert (Ausgang **OUT** von F_TIME_IN_US_TO_DINT) an den Datenpunkt **OUT.D1** übergeben.

Der gesamte Vorgang erfolgt synchron und ohne Zwischenspeicherung – jede erfolgreiche Umrechnung löst sofort ein Ausgangsereignis aus.

Technische Besonderheiten

- **Einheitenumrechnung nach IEC 61131-3**: Der interne Baustein F_TIME_IN_US_TO_DINT interpretiert den intern in Nanosekunden gehaltenen TIME-Wert als Zahl in Mikrosekunden und liefert das Ergebnis als DINT (32-Bit-Ganzzahl (vorzeichenbehaftet)).
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional ausgelegt. Eine Rückwärtskommunikation ist nicht vorgesehen.
- **Composite-Struktur ohne eigene Logik**: Der Baustein besteht ausschließlich aus einer Daten- und einer Ereignisverbindung zum internen Umrechnungsbaustein; es gibt kein eigenes ECC.
- **Leerer TypeHash**: Wie bei allen generierten Konvertierungs-Composites in diesem Paket ist eclipse4diac::core::TypeHash leer ('').

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Der Ablauf lässt sich als einfacher Schritt beschreiben:

Schritt	Aktion
1	Warten auf Ereignis an **IN.E1**
2	Umrechnung des Datenwerts von TIME (Mikrosekunden) nach DINT über F_TIME_IN_US_TO_DINT
3	Ausgabe des umgerechneten Werts an **OUT.D1** und Ereignis an **OUT.E1**
4	Rückkehr zu Schritt 1

Anwendungsszenarien

- **Anzeige/Weiterverarbeitung von Zeitdauern**: Eine über einen ATM-Adapter transportierte Zeitdauer (z. B. eine Zykluszeit oder Verzögerung) soll als Zahl in Mikrosekunden an einen Adapter-Socket vom Typ ADI weitergegeben werden, etwa zur Anzeige auf einem Panel oder zur Weiterverarbeitung in einer Berechnung.
- **Schnittstellenanpassung**: In Netzwerken, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen, ermöglicht der Baustein die nahtlose Umwandlung eines Zeit- in einen Zahlenadapter.

⚖️ Vergleich mit ähnlichen Bausteinen

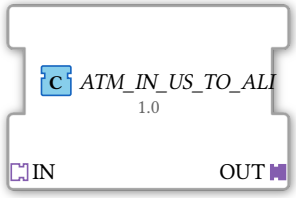
- **Gleiche Einheit, andere Zieltypen**: ATM_IN_US_TO_ALI, ATM_IN_US_TO_ALR, ATM_IN_US_TO_AUDI, ATM_IN_US_TO_AULI
- **Gleicher Zieltyp, andere Einheiten**: ATM_IN_MS_TO_ADI, ATM_IN_NS_TO_ADI, ATM_IN_S_TO_ADI
- **ATM_TIME_TO_TM / ATM_TM_TO_TIME**: reine Adapter-↔-Variablen-Durchreichung ohne Einheitenumrechnung, im Gegensatz zur numerischen Umrechnung von `ATM_IN_US_TO_ADI`.

Fazit

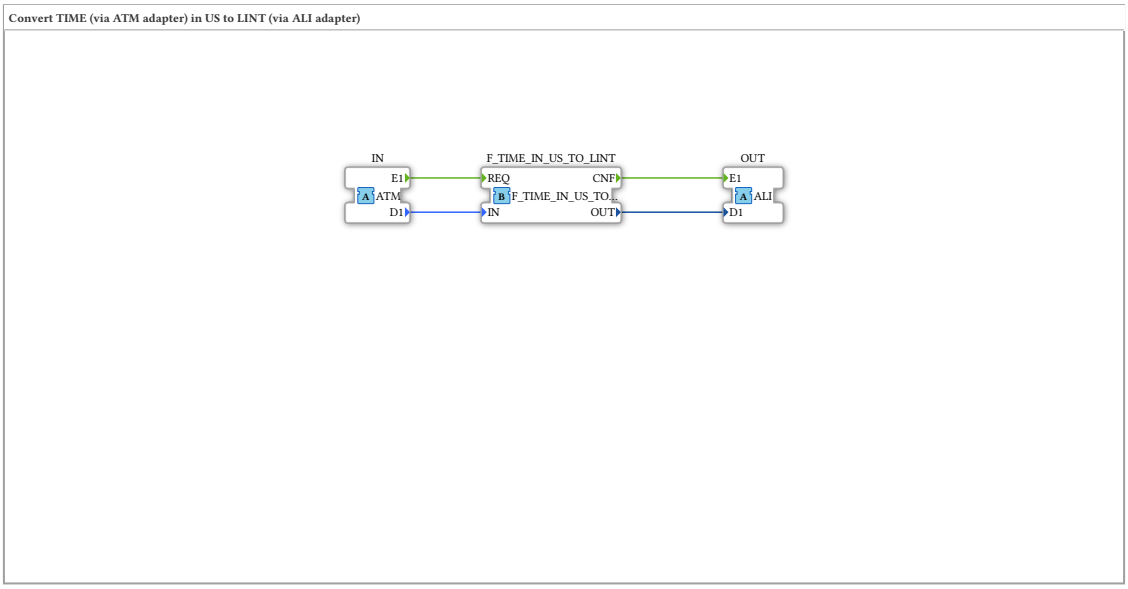
`ATM_IN_US_TO_ADI` liefert eine standardkonforme, vollständig adapterbasierte Umrechnung eines `TIME`-Werts in Mikrosekunden auf einen `DINT`-Wert und fügt sich damit nahtlos in Netzwerke ein, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen.

ATM_IN_US_TO_ALI

Time value to convert – *ATM*



ALI – TIME value of IN expressed in US as LINT



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ATM_IN_US_TO_ALI** ist ein Composite-Baustein, der einen `TIME`-Wert (übertragen über den ATM-Adapter) in Mikrosekunden auf einen `LINT`-Wert (64-Bit-Ganzzahl (vorzeichenbehaftet)) umrechnet und über den ALI-Adapter ausgibt. Er gehört zur Blockfamilie `ATM_IN_<EINHEIT>_TO_<ZIELTYP>`, die dieselbe Umrechnung für alle vier Zeiteinheiten (MS, NS, S, US) und alle fünf Zielganzzahl-/Gleitkommatypen bereitstellt.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkt sichtbaren Ereignis-Eingänge. Die Ereignissteuerung erfolgt indirekt über den Adapter **IN** (Ereignis E1).

Ereignis-Ausgänge

- Keine direkt sichtbaren Ereignis-Ausgänge. Die Ereignissignalisierung erfolgt indirekt über den Adapter **OUT** (Ereignis E1).

Daten-Eingänge

- Keine direkt sichtbaren Daten-Eingänge. Der Eingabewert wird über den Adapter **IN** (Datenpunkt D1 vom Typ `TIME`) bereitgestellt.

Daten-Ausgänge

- Keine direkt sichtbaren Daten-Ausgänge. Der Ausgabewert wird über den Adapter **OUT** (Datenpunkt D1 vom Typ `LINT`) bereitgestellt.

Adapter

Adapter	Rolle	Typ	Beschreibung
IN	Socket	adapter::types::unidirectional::ATM	Unidirektionaler `TIME`-Adapter als Eingangsschnittstelle.
OUT	Plug	adapter::types::unidirectional::ALI	Unidirektionaler `LINT`-Adapter als Ausgangsschnittstelle, Wert von `IN` in Mikrosekunden ausgedrückt.

Funktionsweise

Der Baustein enthält intern einen einzelnen Funktionsbaustein `F_TIME_IN_US_TO_LINT` aus der IEC-61131-3-Standardbibliothek. Die Ablaufsteuerung erfolgt rein ereignisgesteuert:

1. Ein Ereignis am Eingangsadapter **IN.E1** triggert den Eingang **REQ** des internen Umrechnungsbausteins.
2. Der Datenwert **IN.D1** (`TIME`) wird an den Eingang **IN** von `F_TIME_IN_US_TO_LINT` übergeben.
3. `F_TIME_IN_US_TO_LINT` rechnet den Zeitwert in eine Mikrosekunden-Zahl vom Typ `LINT` um und signalisiert dies über seinen Ausgang **CNF**.
4. Dieses Ereignis wird an den Ausgangsadapter **OUT.E1** weitergeleitet.
5. Gleichzeitig wird der umgerechnete Wert (Ausgang **OUT** von `F_TIME_IN_US_TO_LINT`) an den Datenpunkt **OUT.D1** übergeben.

Der gesamte Vorgang erfolgt synchron und ohne Zwischenspeicherung – jede erfolgreiche Umrechnung löst sofort ein Ausgangsereignis aus.

Technische Besonderheiten

- **Einheitenumrechnung nach IEC 61131-3**: Der interne Baustein `F_TIME_IN_US_TO_LINT` interpretiert den intern in Nanosekunden gehaltenen `TIME`-Wert als Zahl in Mikrosekunden und liefert das Ergebnis als `LINT` (64-Bit-Ganzzahl (vorzeichenbehaftet)).
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional ausgelegt. Eine Rückwärtskommunikation ist nicht vorgesehen.
- **Composite-Struktur ohne eigene Logik**: Der Baustein besteht ausschließlich aus einer Daten- und einer Ereignisverbindung zum internen Umrechnungsbaustein; es gibt kein eigenes ECC.
- **Leerer TypeHash**: Wie bei allen generierten Konvertierungs-Composites in diesem Paket ist `eclipse4diac::core::TypeHash` leer (`''`).

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Der Ablauf lässt sich als einfacher Schritt beschreiben:

Schritt	Aktion
1	Warten auf Ereignis an **IN.E1**
2	Umrechnung des Datenwerts von `TIME` (Mikrosekunden) nach `LINT` über `F_TIME_IN_US_TO_LINT`
3	Ausgabe des umgerechneten Werts an **OUT.D1** und Ereignis an **OUT.E1**

4	Rückkehr zu Schritt 1

Anwendungsszenarien

- **Anzeige/Weiterverarbeitung von Zeitdauern**: Eine über einen `ATM`-Adapter transportierte Zeitdauer (z. B. eine Zykluszeit oder Verzögerung) soll als Zahl in Mikrosekunden an einen Adapter-Socket vom Typ `ALI` weitergegeben werden, etwa zur Anzeige auf einem Panel oder zur Weiterverarbeitung in einer Berechnung.
- **Schnittstellenanpassung**: In Netzwerken, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen, ermöglicht der Baustein die nahtlose Umwandlung eines Zeit- in einen Zahlenadapter.

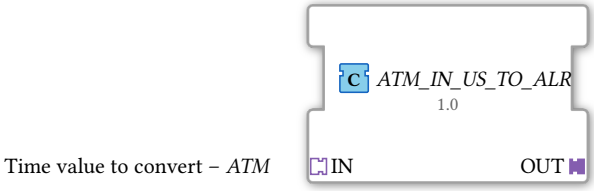
⚖️ Vergleich mit ähnlichen Bausteinen

- **Gleiche Einheit, andere Zieltypen**: ATM_IN_US_TO_ADI, ATM_IN_US_TO_ALR, ATM_IN_US_TO_AUDI, ATM_IN_US_TO_AULI
- **Gleicher Zieltyp, andere Einheiten**: ATM_IN_MS_TO_ALI, ATM_IN_NS_TO_ALI, ATM_IN_S_TO_ALI
- **ATM_TIME_TO_TM** / **ATM_TM_TO_TIME**: reine Adapter-↔-Variablen-Durchreichung ohne Einheitenumrechnung, im Gegensatz zur numerischen Umrechnung von `ATM_IN_US_TO_ALI`.

Fazit

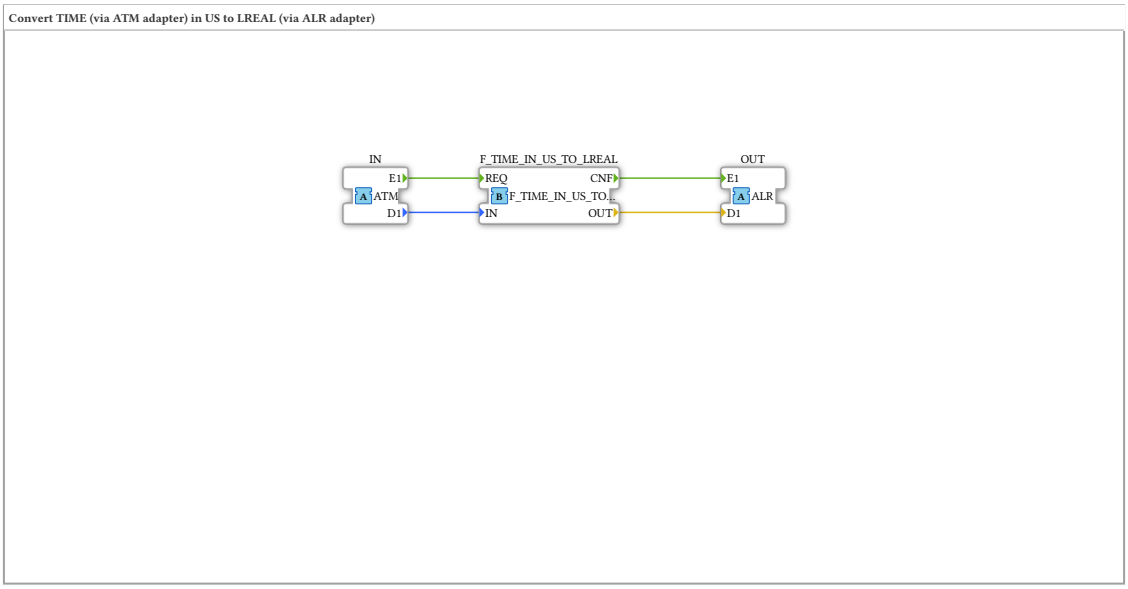
`ATM_IN_US_TO_ALI` liefert eine standardkonforme, vollständig adapterbasierte Umrechnung eines `TIME`-Werts in Mikrosekunden auf einen `LINT`-Wert und fügt sich damit nahtlos in Netzwerke ein, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen.

ATM_IN_US_TO_ALR



Time value to convert – *ATM*

ALR – TIME value of IN expressed in US as LREAL



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ATM_IN_US_TO_ALR** ist ein Composite-Baustein, der einen `TIME`-Wert (übertragen über den ATM-Adapter) in Mikrosekunden auf einen `LREAL`-Wert (64-Bit-Gleitkommazahl) umrechnet und über den ALR-Adapter ausgibt. Er gehört zur Blockfamilie `ATM_IN_<EINHEIT>_TO_<ZIELTYP>`, die

dieselbe Umrechnung für alle vier Zeiteinheiten (MS, NS, S, US) und alle fünf Zielganzzahl-/Gleitkommatypen bereitstellt.

Schnittstellenstruktur

Ereignis-Eingänge
- Keine direkt sichtbaren Ereignis-Eingänge. Die Ereignissteuerung erfolgt indirekt über den Adapter **IN** (Ereignis E1).

Ereignis-Ausgänge
- Keine direkt sichtbaren Ereignis-Ausgänge. Die Ereignissignalisierung erfolgt indirekt über den Adapter **OUT** (Ereignis E1).

Daten-Eingänge
- Keine direkt sichtbaren Daten-Eingänge. Der Eingabewert wird über den Adapter **IN** (Datenpunkt D1 vom Typ `TIME`) bereitgestellt.

Daten-Ausgänge
- Keine direkt sichtbaren Daten-Ausgänge. Der Ausgabewert wird über den Adapter **OUT** (Datenpunkt D1 vom Typ `LREAL`) bereitgestellt.

Adapter

Adapter	Rolle	Typ	Beschreibung
IN	Socket	adapter::types::unidirectional::ATM	Unidirektionaler `TIME`-Adapter als Eingangsschnittstelle.
OUT	Plug	adapter::types::unidirectional::ALR	Unidirektionaler `LREAL`-Adapter als Ausgangsschnittstelle, Wert von `IN` in Mikrosekunden ausgedrückt.

Funktionsweise

Der Baustein enthält intern einen einzelnen Funktionsbaustein `F_TIME_IN_US_TO_LREAL` aus der IEC-61131-3-Standardbibliothek. Die Ablaufsteuerung erfolgt rein ereignisgesteuert:

1. Ein Ereignis am Eingangsadapter **IN.E1** triggert den Eingang **REQ** des internen Umrechnungsbausteins.
2. Der Datenwert **IN.D1** (`TIME`) wird an den Eingang **IN** von `F_TIME_IN_US_TO_LREAL` übergeben.
3. `F_TIME_IN_US_TO_LREAL` rechnet den Zeitwert in eine Mikrosekunden-Zahl vom Typ `LREAL` um und signalisiert dies über seinen Ausgang **CNF**.
4. Dieses Ereignis wird an den Ausgangsadapter **OUT.E1** weitergeleitet.
5. Gleichzeitig wird der umgerechnete Wert (Ausgang **OUT** von `F_TIME_IN_US_TO_LREAL`) an den Datenpunkt **OUT.D1** übergeben.

Der gesamte Vorgang erfolgt synchron und ohne Zwischenspeicherung – jede erfolgreiche Umrechnung löst sofort ein Ausgangsereignis aus.

Technische Besonderheiten
- **Einheitenumrechnung nach IEC 61131-3**: Der interne Baustein `F_TIME_IN_US_TO_LREAL` interpretiert den intern in Nanosekunden gehaltenen `TIME`-Wert als Zahl in Mikrosekunden und liefert das Ergebnis als `LREAL` (64-Bit-Gleitkommazahl).
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional ausgelegt. Eine Rückwärtskommunikation ist nicht vorgesehen.
- **Composite-Struktur ohne eigene Logik**: Der Baustein besteht ausschließlich aus einer Daten- und einer Ereignisverbindung zum internen Umrechnungsbaustein; es gibt kein eigenes ECC.
- **Leerer TypeHash**: Wie bei allen generierten Konvertierungs-Composites in diesem Paket ist `eclipse4diac::core::TypeHash` leer (`''`).

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Der Ablauf lässt sich als einfacher Schritt beschreiben:

Schritt	Aktion
1	Warten auf Ereignis an **IN.E1**
2	Umrechnung des Datenwerts von `TIME` (Mikrosekunden) nach `LREAL` über `F_TIME_IN_US_TO_LREAL`
3	Ausgabe des umgerechneten Werts an **OUT.D1** und Ereignis an **OUT.E1**
4	Rückkehr zu Schritt 1

Anwendungsszenarien

- **Anzeige/Weiterverarbeitung von Zeitdauern**: Eine über einen `ATM`-Adapter transportierte Zeitdauer (z. B. eine Zykluszeit oder Verzögerung) soll als Zahl in Mikrosekunden an einen Adapter-Socket vom Typ `ALR` weitergegeben werden, etwa zur Anzeige auf einem Panel oder zur Weiterverarbeitung in einer Berechnung.
- **Schnittstellenanpassung**: In Netzwerken, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen, ermöglicht der Baustein die nahtlose Umwandlung eines Zeit- in einen Zahlenadapter.

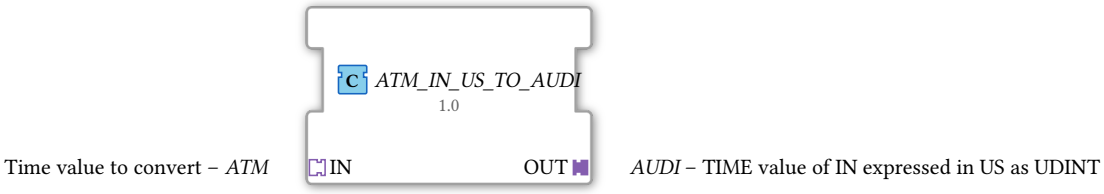
⚖️ Vergleich mit ähnlichen Bausteinen

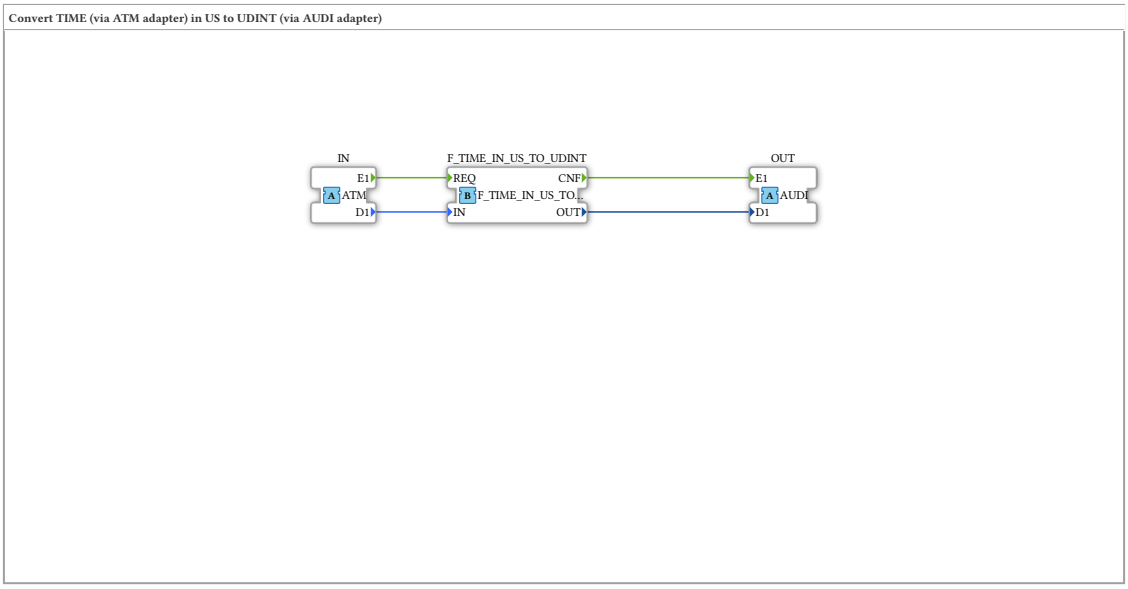
- **Gleiche Einheit, andere Zieltypen**: ATM_IN_US_TO_ADI, ATM_IN_US_TO_ALI, ATM_IN_US_TO_AUDI, ATM_IN_US_TO_AULI
- **Gleicher Zieltyp, andere Einheiten**: ATM_IN_MS_TO_ALR, ATM_IN_NS_TO_ALR, ATM_IN_S_TO_ALR
- **ATM_TIME_TO_TM** / **ATM_TM_TO_TIME**: reine Adapter-↔-Variablen-Durchreichung ohne Einheitenumrechnung, im Gegensatz zur numerischen Umrechnung von `ATM_IN_US_TO_ALR`.

Fazit

`ATM_IN_US_TO_ALR` liefert eine standardkonforme, vollständig adapterbasierte Umrechnung eines `TIME`-Werts in Mikrosekunden auf einen `LREAL`-Wert und fügt sich damit nahtlos in Netzwerke ein, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen.

ATM_IN_US_TO_AUDI





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ATM_IN_US_TO_AUDI** ist ein Composite-Baustein, der einen `TIME`-Wert (übertragen über den ATM-Adapter) in Mikrosekunden auf einen `UDINT`-Wert (32-Bit-Ganzzahl (vorzeichenlos)) umrechnet und über den AUDI-Adapter ausgibt. Er gehört zur Blockfamilie `ATM_IN_<EINHEIT>_TO_<ZIELTYP>`, die dieselbe Umrechnung für alle vier Zeiteinheiten (MS, NS, S, US) und alle fünf Zielganzzahl-/Gleitkommatypen bereitstellt.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkt sichtbaren Ereignis-Eingänge. Die Ereignissteuerung erfolgt indirekt über den Adapter **IN** (Ereignis E1).

Ereignis-Ausgänge

- Keine direkt sichtbaren Ereignis-Ausgänge. Die Ereignissignalisierung erfolgt indirekt über den Adapter **OUT** (Ereignis E1).

Daten-Eingänge

- Keine direkt sichtbaren Daten-Eingänge. Der Eingabewert wird über den Adapter **IN** (Datenpunkt D1 vom Typ `TIME`) bereitgestellt.

Daten-Ausgänge

- Keine direkt sichtbaren Daten-Ausgänge. Der Ausgabewert wird über den Adapter **OUT** (Datenpunkt D1 vom Typ `UDINT`) bereitgestellt.

Adapter

Adapter	Rolle	Typ	Beschreibung
IN	Socket	adapter::types::unidirectional::ATM	Unidirektionaler `TIME`-Adapter als Eingangsschnittstelle.
OUT	Plug	adapter::types::unidirectional::AUDI	Unidirektionaler `UDINT`-Adapter als Ausgangsschnittstelle, Wert von `IN` in Mikrosekunden ausgedrückt.

Funktionsweise

Der Baustein enthält intern einen einzelnen Funktionsbaustein `F_TIME_IN_US_TO_UDINT` aus der IEC-61131-3-Standardbibliothek. Die Ablaufsteuerung erfolgt rein ereignisgesteuert:

1. Ein Ereignis am Eingangsadapter **IN.E1** triggert den Eingang **REQ** des internen Umrechnungsbausteins.
2. Der Datenwert **IN.D1** (`TIME`) wird an den Eingang **IN** von `F_TIME_IN_US_TO_UDINT` übergeben.
3. `F_TIME_IN_US_TO_UDINT` rechnet den Zeitwert in eine Mikrosekunden-Zahl vom Typ `UDINT` um und signalisiert dies über seinen Ausgang **CNF**.
4. Dieses Ereignis wird an den Ausgangsadapter **OUT.E1** weitergeleitet.
5. Gleichzeitig wird der umgerechnete Wert (Ausgang **OUT** von `F_TIME_IN_US_TO_UDINT`) an den Datenpunkt **OUT.D1** übergeben.

Der gesamte Vorgang erfolgt synchron und ohne Zwischenspeicherung – jede erfolgreiche Umrechnung löst sofort ein Ausgangsereignis aus.

Technische Besonderheiten

- **Einheitenumrechnung nach IEC 61131-3**: Der interne Baustein `F_TIME_IN_US_TO_UDINT` interpretiert den intern in Nanosekunden gehaltenen `TIME`-Wert als Zahl in Mikrosekunden und liefert das Ergebnis als `UDINT` (32-Bit-Ganzzahl (vorzeichenlos)).
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional ausgelegt. Eine Rückwärtskommunikation ist nicht vorgesehen.
- **Composite-Struktur ohne eigene Logik**: Der Baustein besteht ausschließlich aus einer Daten- und einer Ereignisverbindung zum internen Umrechnungsbaustein; es gibt kein eigenes ECC.
- **Leerer TypeHash**: Wie bei allen generierten Konvertierungs-Composites in diesem Paket ist `eclipse4diac::core::TypeHash` leer (`''`).

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Der Ablauf lässt sich als einfacher Schritt beschreiben:

Schritt	Aktion
1	Warten auf Ereignis an **IN.E1**
2	Umrechnung des Datenwerts von `TIME` (Mikrosekunden) nach `UDINT` über `F_TIME_IN_US_TO_UDINT`
3	Ausgabe des umgerechneten Werts an **OUT.D1** und Ereignis an **OUT.E1**
4	Rückkehr zu Schritt 1

Anwendungsszenarien

- **Anzeige/Weiterverarbeitung von Zeitdauern**: Eine über einen `ATM`-Adapter transportierte Zeitdauer (z. B. eine Zykluszeit oder Verzögerung) soll als Zahl in Mikrosekunden an einen Adapter-Socket vom Typ `AUDI` weitergegeben werden, etwa zur Anzeige auf einem Panel oder zur Weiterverarbeitung in einer Berechnung.
- **Schnittstellenanpassung**: In Netzwerken, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen, ermöglicht der Baustein die nahtlose Umwandlung eines Zeit- in einen Zahlenadapter.

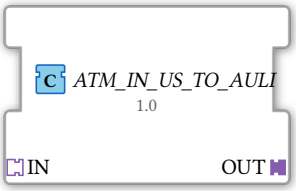
⚖️ Vergleich mit ähnlichen Bausteinen

- **Gleiche Einheit, andere Zieltypen**: ATM_IN_US_TO_ADI, ATM_IN_US_TO_ALI, ATM_IN_US_TO_ALR, ATM_IN_US_TO_AULI
- **Gleicher Zieltyp, andere Einheiten**: ATM_IN_MS_TO_AUDI, ATM_IN_NS_TO_AUDI, ATM_IN_S_TO_AUDI
- **ATM_TIME_TO_TM** / **ATM_TM_TO_TIME**: reine Adapter-↔-Variablen-Durchreichung ohne Einheitenumrechnung, im Gegensatz zur numerischen Umrechnung von `ATM_IN_US_TO_AUDI`.

Fazit

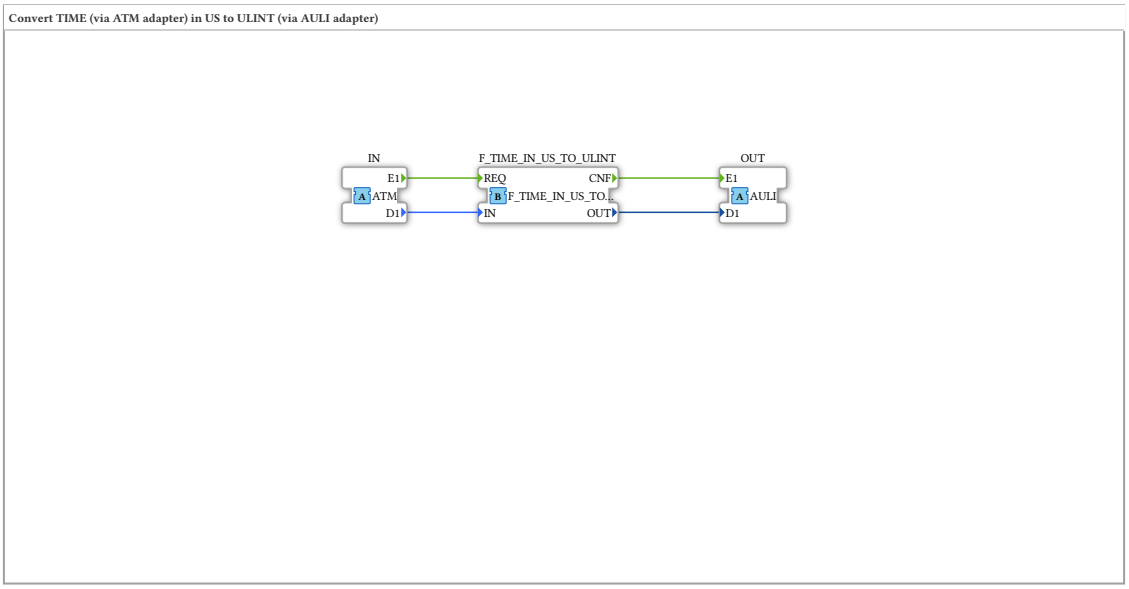
`ATM_IN_US_TO_AUDI` liefert eine standardkonforme, vollständig adapterbasierte Umrechnung eines `TIME`-Werts in Mikrosekunden auf einen `UDINT`-Wert und fügt sich damit nahtlos in Netzwerke ein, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen.

ATM_IN_US_TO_AULI



Time value to convert – *ATM*

AULI – TIME value of IN expressed in US as ULINT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ATM_IN_US_TO_AULI** ist ein Composite-Baustein, der einen `TIME`-Wert (übertragen über den ATM-Adapter) in Mikrosekunden auf einen `ULINT`-Wert (64-Bit-Ganzzahl (vorzeichenlos)) umrechnet und über den AULI-Adapter ausgibt. Er gehört zur Blockfamilie `ATM_IN_<EINHEIT>_TO_<ZIELTYP>`, die dieselbe Umrechnung für alle vier Zeiteinheiten (MS, NS, S, US) und alle fünf Zielganzzahl-/Gleitkommatypen bereitstellt.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkt sichtbaren Ereignis-Eingänge. Die Ereignissteuerung erfolgt indirekt über den Adapter **IN** (Ereignis E1).

Ereignis-Ausgänge

- Keine direkt sichtbaren Ereignis-Ausgänge. Die Ereignissignalisierung erfolgt indirekt über den Adapter **OUT** (Ereignis E1).

Daten-Eingänge

- Keine direkt sichtbaren Daten-Eingänge. Der Eingabewert wird über den Adapter **IN** (Datenpunkt D1 vom Typ `TIME`) bereitgestellt.

Daten-Ausgänge

- Keine direkt sichtbaren Daten-Ausgänge. Der Ausgabewert wird über den Adapter **OUT** (Datenpunkt D1 vom Typ `ULINT`) bereitgestellt.

Adapter

Adapter	Rolle	Typ	Beschreibung
IN	Socket	adapter::types::unidirectional::ATM	Unidirektionaler `TIME`-Adapter als Eingangsschnittstelle.
OUT	Plug	adapter::types::unidirectional::AULI	Unidirektionaler `ULINT`-Adapter als Ausgangsschnittstelle, Wert von `IN` in Mikrosekunden ausgedrückt.

Funktionsweise

Der Baustein enthält intern einen einzelnen Funktionsbaustein `F_TIME_IN_US_TO_ULINT` aus der IEC-61131-3-Standardbibliothek. Die Ablaufsteuerung erfolgt rein ereignisgesteuert:

1. Ein Ereignis am Eingangsadapter **IN.E1** triggert den Eingang **REQ** des internen Umrechnungsbausteins.
2. Der Datenwert **IN.D1** (`TIME`) wird an den Eingang **IN** von `F_TIME_IN_US_TO_ULINT` übergeben.
3. `F_TIME_IN_US_TO_ULINT` rechnet den Zeitwert in eine Mikrosekunden-Zahl vom Typ `ULINT` um und signalisiert dies über seinen Ausgang **CNF**.
4. Dieses Ereignis wird an den Ausgangsadapter **OUT.E1** weitergeleitet.
5. Gleichzeitig wird der umgerechnete Wert (Ausgang **OUT** von `F_TIME_IN_US_TO_ULINT`) an den Datenpunkt **OUT.D1** übergeben.

Der gesamte Vorgang erfolgt synchron und ohne Zwischenspeicherung – jede erfolgreiche Umrechnung löst sofort ein Ausgangsereignis aus.

Technische Besonderheiten

- **Einheitenumrechnung nach IEC 61131-3**: Der interne Baustein `F_TIME_IN_US_TO_ULINT` interpretiert den intern in Nanosekunden gehaltenen `TIME`-Wert als Zahl in Mikrosekunden und liefert das Ergebnis als `ULINT` (64-Bit-Ganzzahl (vorzeichenlos)).
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch der Ausgangsadapter sind unidirektional ausgelegt. Eine Rückwärtskommunikation ist nicht vorgesehen.
- **Composite-Struktur ohne eigene Logik**: Der Baustein besteht ausschließlich aus einer Daten- und einer Ereignisverbindung zum internen Umrechnungsbaustein; es gibt kein eigenes ECC.
- **Leerer TypeHash**: Wie bei allen generierten Konvertierungs-Composites in diesem Paket ist `eclipse4diac::core::TypeHash` leer (`''`).

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Der Ablauf lässt sich als einfacher Schritt beschreiben:

Schritt	Aktion
1	Warten auf Ereignis an **IN.E1**
2	Umrechnung des Datenwerts von `TIME` (Mikrosekunden) nach `ULINT` über `F_TIME_IN_US_TO_ULINT`
3	Ausgabe des umgerechneten Werts an **OUT.D1** und Ereignis an **OUT.E1**
4	Rückkehr zu Schritt 1

Anwendungsszenarien

- **Anzeige/Weiterverarbeitung von Zeitdauern**: Eine über einen `ATM`-Adapter transportierte Zeitdauer (z. B. eine Zykluszeit oder Verzögerung) soll als Zahl in Mikrosekunden an einen Adapter-Socket vom Typ `AULI` weitergegeben werden, etwa zur Anzeige auf einem Panel oder zur Weiterverarbeitung in einer Berechnung.
- **Schnittstellenanpassung**: In Netzwerken, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen, ermöglicht der Baustein die nahtlose Umwandlung eines Zeit- in einen Zahlenadapter.

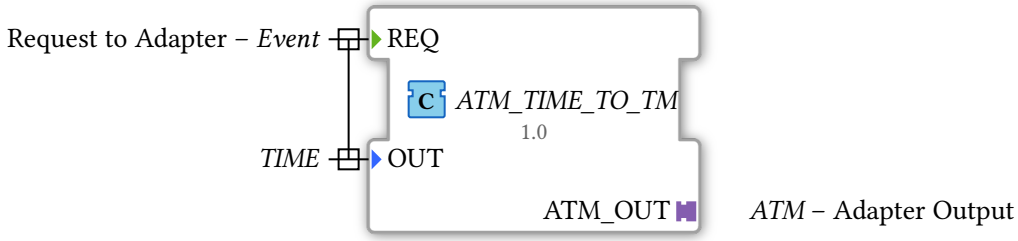
⚖️ **Vergleich mit ähnlichen Bausteinen**

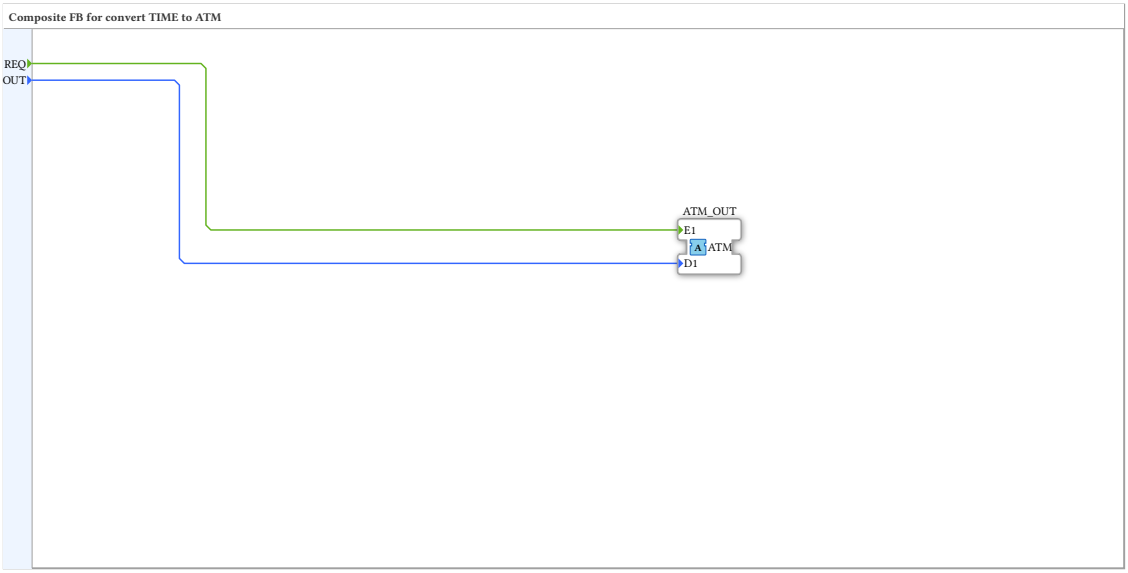
- **Gleiche Einheit, andere Zieltypen**: ATM_IN_US_TO_ADI, ATM_IN_US_TO_ALI, ATM_IN_US_TO_ALR, ATM_IN_US_TO_AUDI
- **Gleicher Zieltyp, andere Einheiten**: ATM_IN_MS_TO_AULI, ATM_IN_NS_TO_AULI, ATM_IN_S_TO_AULI
- **ATM_TIME_TO_TM / ATM_TM_TO_TIME**: reine Adapter-↔-Variablen-Durchreichung ohne Einheitenumrechnung, im Gegensatz zur numerischen Umrechnung von `ATM_IN_US_TO_AULI`.

Fazit

`ATM_IN_US_TO_AULI` liefert eine standardkonforme, vollständig adapterbasierte Umrechnung eines `TIME`-Werts in Mikrosekunden auf einen `ULINT`-Wert und fügt sich damit nahtlos in Netzwerke ein, die konsequent auf Adapterverbindungen statt klassischer Datenverdrahtung setzen.

ATM_TIME_TO_TM





• – ○ • – ○ • – ○ •

Einleitung

`ATM_TIME_TO_TM` ist ein Composite-Funktionsblock, der einen einfachen `TIME`-Datenwert in das ATM-Adapterformat umwandelt. Er folgt exakt demselben Muster wie ADI_DINT_TO_DI (DINT→ADI) und AUI_UINT_TO_UI (UINT→AUI): eine einzelne Datenverbindung und eine einzelne Ereignisverbindung, direkt auf den Plug durchgereicht — keine eigene Umrechnungslogik.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Anfrage-Event zur Auslösung der Konvertierung; trägt den Wert von `OUT` (`With Var="OUT"`).

Ereignis-Ausgänge

Keine eigenen Ereignis-Ausgänge; das Ereignis läuft ausschließlich über den Adapter-Plug `ATM_OUT`.

Daten-Eingänge

- **OUT** (TIME): Der zu konvertierende Zeitwert.

Daten-Ausgänge

Keine eigenen Daten-Ausgänge; der Wert wird ausschließlich über den Adapter-Plug `ATM_OUT` bereitgestellt.

Adapter

- **ATM_OUT** (Plug, Typ `adapter::types::unidirectional::ATM`): gibt bei `REQ` den Wert von `OUT` über sein `E1`/`D1`-Adapterereignis aus.

Funktionsweise

Bei Eintreffen des Ereignisses `REQ` wird dieses direkt an `ATM_OUT.E1` weitergereicht, während der aktuelle Wert von `OUT` gleichzeitig auf `ATM_OUT.D1` gelegt wird. Der Plug `ATM_OUT` feuert damit sein Adapterereignis mit dem übergebenen Zeitwert — ohne Zwischenspeicherung, ohne Change-Detection, bei jedem `REQ` erneut.

Technische Besonderheiten

- **Reine Verdrahtungslogik**, kein eigenes ECC, keine ST-Algorithmen — ein Composite-FB, der genau eine Daten- und eine Ereignisverbindung herstellt.
- **Kein `REAL_TO_TIME` in den Standardbibliotheken.** Dieser Baustein wandelt keinen numerischen Wert in `TIME` um, sondern nimmt einen bereits fertigen `TIME`-Wert entgegen und reicht ihn nur als Adapter weiter. Die eigentliche Umrechnung (z. B. von REAL-Sekunden) erfolgt vorher, typisch über `iec61131::arithmetic::F_MULTIME` (`T#1s * REAL#Sekunden`) — siehe den `Override_Timer`- SubApp (`FBs::sys`, Getreideannahme-Anwendungscode, nicht Teil dieser Bibliotheks-Doku) für ein konkretes Anwendungsbeispiel.
- **Kein Event bei unverändertem Wert wird unterdrückt** — anders als z. B. bei AX_ATM_FB_TONs `ET`-Ausgang gibt es hier keine `E_D_FF`/`E_D_FF_ANY`-Pufferung; jedes `REQ` erzeugt ein neues Adapterereignis.

Zustandsübersicht

Der Baustein ist zustandslos: Jedes `REQ`-Ereignis führt unmittelbar zur Ausgabe des aktuellen `OUT`-Werts über `ATM_OUT`.

Anwendungsszenarien

- **Sockets füttern, die einen `ATM`-Adapter statt einer einfachen `TIME`-Variable erwarten**, z. B. `PT` von AX_ATM_FB_TON/ AX_ATM_FB_TOF/ AX_ATM_FB_TP.
- Letztes Glied einer Umrechnungskette von REAL-Sekunden zu einem adapterbasierten Zeitwert (`F_MULTIME` ⟶ `ATM_TIME_TO_TM`), wie im `Override_Timer`-SubApp (`FBs::sys`).

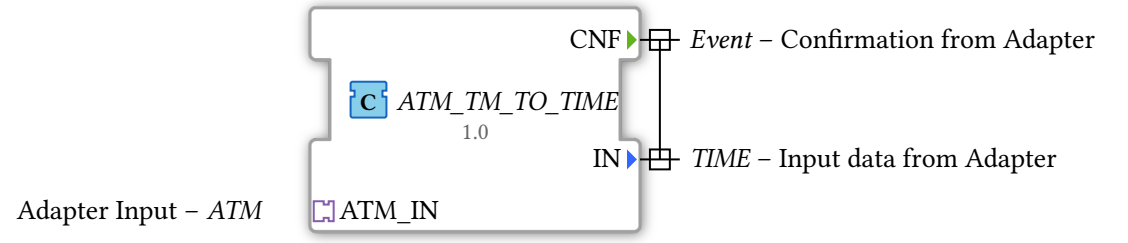
⚖️ Vergleich mit ähnlichen Bausteinen

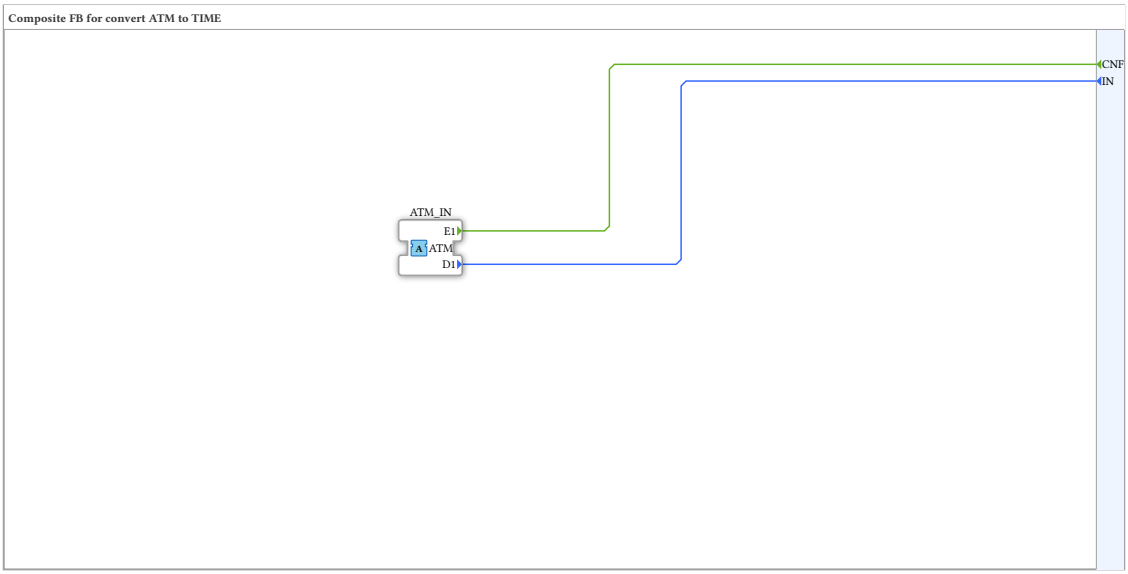
- **ATM_TM_TO_TIME**: die Gegenrichtung — nimmt einen `ATM`-Adapter-Socket entgegen und stellt den Wert als einfache `TIME`-Datenvariable bereit.
- **ADI_DINT_TO_DI** / **AUI_UINT_TO_UI** / **AR_REAL_TO_R**: dasselbe „Datenwert ⟶ Adapter-Plug“-Muster für DINT, UINT bzw. REAL statt TIME.

Fazit

`ATM_TIME_TO_TM` ist die einfachste Möglichkeit, einen `TIME`-Datenwert als `ATM`-Adapter-Plug bereitzustellen, und schließt damit die Lücke zwischen numerischen Zeitberechnungen (z. B. über `F_MULTIME`) und adapterbasierten Zeit-Sockets wie `PT` bei `AX_ATM_FB_TON/TOF/TP`.

ATM_TM_TO_TIME





• – ◦ • – ◦ • – ◦ •

Einleitung

`ATM_TM_TO_TIME` ist die Gegenrichtung zu ATM_TIME_TO_TM: ein Composite- Funktionsblock, der einen ATM-Adapter-Socket entgegennimmt und dessen Wert als einfache `TIME`-Datenvariable bereitstellt. Er folgt exakt demselben Muster wie AUI_UI_TO_UINT (AUI→UINT): eine einzelne Datenverbindung und eine einzelne Ereignisverbindung, direkt vom Socket abgegriffen — keine eigene Umrechnungslogik.

Schnittstellenstruktur

Ereignis-Eingänge

Keine eigenen Ereignis-Eingänge; das Ereignis kommt ausschließlich über den Adapter-Socket `ATM_IN`.

Ereignis-Ausgänge

- **CNF**: Bestätigungs-Event, feuert wenn der Adapter-Socket `ATM_IN` sein Ereignis auslöst; trägt den Wert von `IN` (`With Var="IN"`).

Daten-Eingänge

Keine eigenen Daten-Eingänge; der Wert kommt ausschließlich über den Adapter-Socket `ATM_IN`.

Daten-Ausgänge

- **IN** (TIME): Der vom Adapter empfangene Zeitwert.

Adapter

- **ATM_IN** (Socket, Typ `adapter::types::unidirectional::ATM`): empfängt den Zeitwert vom verbundenen Plug.

Funktionsweise

Sobald der verbundene Plug ein Ereignis auf `ATM_IN` auslöst, wird `ATM_IN.E1` direkt an `CNF` weitergereicht, während `ATM_IN.D1` gleichzeitig auf die Datenausgangsvariable `IN` gelegt wird. Der Baustein macht damit

den Wert eines eingehenden ATM-Adapters für nachgeschaltete Bausteine nutzbar, die eine einfache TIME-Variable statt eines Adapter-Sockets erwarten (z. B. OSCAT::Basic::POUs::conversion::TIME_TO_REAL, dessen TM-Eingang eine reine Datenvariable ist).

Technische Besonderheiten

- **Reine Verdrahtungslogik**, kein eigenes ECC, keine ST-Algorithmen — ein Composite-FB, der genau eine Daten- und eine Ereignisverbindung herstellt.
- **Notwendig, weil Plugs nicht direkt in Funktionen mit einfachen Datenvariablen einspeisen können.** Ein Adapter-Plug (z. B. der ET-Ausgang von AX_ATM_FB_TON) kann nur an einen Socket angeschlossen werden, nicht direkt an eine InputVar einer Funktion. ATM_TM_TO_TIME schließt diese Lücke.
- **Kein Event bei unverändertem Wert wird unterdrückt** — der Baustein selbst puffert nichts; ob ein Event tatsächlich nur bei Wertänderung kommt, hängt vom vorgeschalteten Plug ab (z. B. hat AX_ATM_FB_TON.ET bereits eine eigene E_D_FF_ANY-Change-Detection).

Zustandsübersicht

Der Baustein ist zustandslos: Jedes Ereignis am Socket ATM_IN führt unmittelbar zur Ausgabe des aktuellen Werts über IN/CNF.

Anwendungsszenarien

- **Einen Adapter-Plug lesbar machen für Funktionen mit einfachen Datenvariablen**, z. B. den ET-Ausgang von AX_ATM_FB_TON/ AX_ATM_FB_TOF/ AX_ATM_FB_TP an OSCAT::Basic::POUs::conversion::TIME_TO_REAL weiterreichen.
- Erstes Glied einer Rückrechnungskette von einem adapterbasierten Zeitwert zu REAL-Sekunden (ATM_TM_TO_TIME → TIME_TO_REAL → F_DIV), wie im Override_Timer-SubApp (FBs::sys, Getreideannahme-Anwendungscode, nicht Teil dieser Bibliotheks-Doku).

⚖️ Vergleich mit ähnlichen Bausteinen

- **ATM_TIME_TO_TM**: die Gegenrichtung — nimmt eine einfache TIME- Datenvariable entgegen und stellt sie als ATM-Adapter-Plug bereit.
- **AUI_UI_TO_UINT / ADI_DI_TO_DINT**: dasselbe „Adapter-Socket → Datenwert"-Muster für UINT bzw. DINT statt TIME.

Fazit

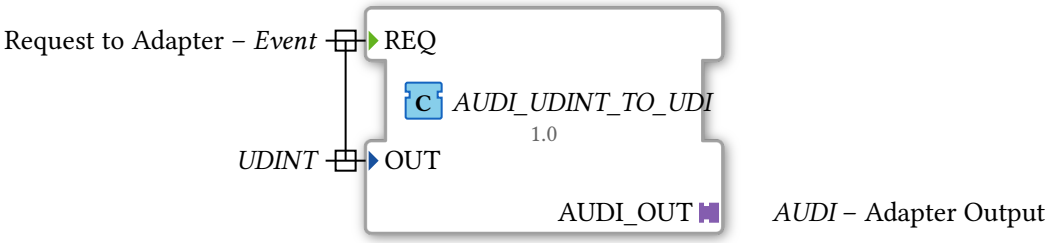
ATM_TM_TO_TIME ist die einfachste Möglichkeit, den Wert eines ATM-Adapter-Plugs als normale TIME-Variable weiterzuverarbeiten, und schließt damit die Lücke zwischen adapterbasierten Zeit-Plugs (z. B. ET bei AX_ATM_FB_TON/TOF/TP) und Funktionen, die nur einfache Datenvariablen entgegennehmen.

UDINT

AUDI_UDINT_TO_UDI

Einleitung

Der AUDI_UDINT_TO_UDI Funktionsblock ist ein Composite-Funktionsblock, der zur Konvertierung eines UDINT-Werts in das AUDI-Adapterformat dient.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Anfrage-Event zur Auslösung der Konvertierung

Ereignis-Ausgänge

- Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

- **OUT**: UDINT-Eingangswert, der konvertiert werden soll

Daten-Ausgänge

- Keine direkten Daten-Ausgänge vorhanden

Adapter

- **AUDI_OUT**: Plug vom Typ `AUDI`

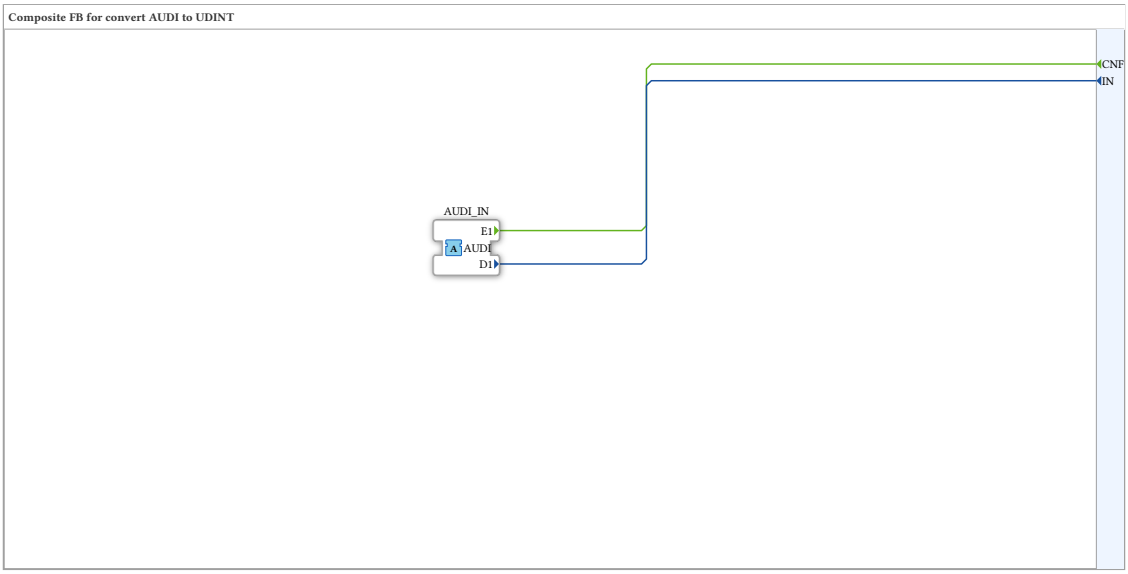
Funktionsweise

Der Funktionsblock empfängt einen UDINT-Wert am Daten-Eingang OUT und wandelt diesen bei Eintreffen des REQ-Ereignisses in das AUDI-Adapterformat um.

🛠️ Zugehörige Übungen

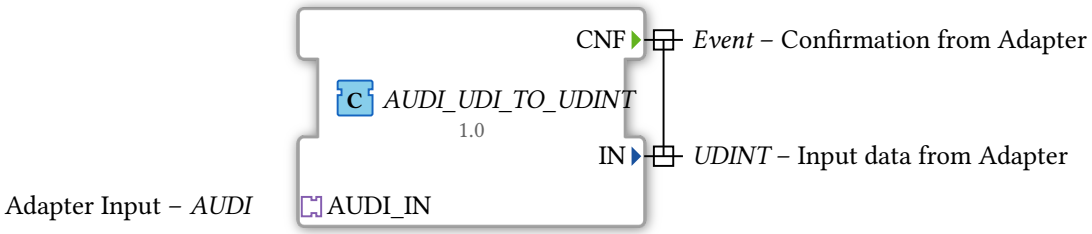
- Uebung_083_AX

AUDI_UDI_TO_UDINT



Einleitung

Der AUDI_UDI_TO_UDINT Funktionsblock ist ein Composite-Funktionsblock, der zur Konvertierung des AUDI-Adapterformats in einen UDINT-Wert dient.



Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkten Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

- **CNF**: Bestätigungs-Event nach erfolgter Konvertierung

Daten-Eingänge

- Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

- **IN**: UDINT-Ausgangswert der konvertierten Daten

Adapter

- **AUDI_IN**: Socket vom Typ `AUDI`

Funktionsweise

Der Funktionsblock empfängt Daten über den AUDI-Adapter-Socket und stellt den enthaltenen UDINT-Wert am Ausgang IN zur Verfügung, sobald ein Ereignis über den Adapter eintrifft (CNF).

UDINT Conversion

🛠️ Zugehörige Übungen

- Uebung_009

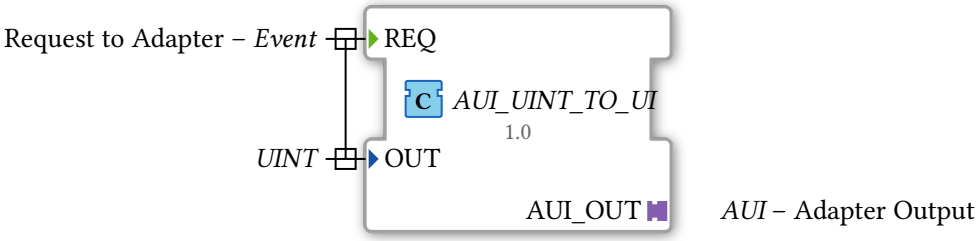
- Uebung_009_AX
- Uebung_009a
- Uebung_011
- Uebung_011a
- Uebung_011a2
- Uebung_012
- NumericValue_ID_TO_NVS
- Uebung_012b
- Uebung_012c
- Uebung_015
- Uebung_015a
- Uebung_020c2_sub
- Uebung_028
- Uebung_034
- Uebung_034a1_Q1
- Uebung_034a1_Q2
- Uebung_034a1_Q4
- Uebung_034b
- Uebung_072c
- Uebung_083
- Uebung_083_AX
- Uebung_103
- Uebung_111
- Uebung_151_AX
- Uebung_152
- Uebung_153

UINT

AUI_UINT_TO_UI

Einleitung

Der AUI_UINT_TO_UI Funktionsblock ist ein Composite-Funktionsblock, der zur Konvertierung eines UINT-Werts in das AUI-Adapterformat dient.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Anfrage-Event zur Auslösung der Konvertierung

Ereignis-Ausgänge

- Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

- **OUT**: UINT-Eingangswert, der konvertiert werden soll

Daten-Ausgänge

- Keine direkten Daten-Ausgänge vorhanden

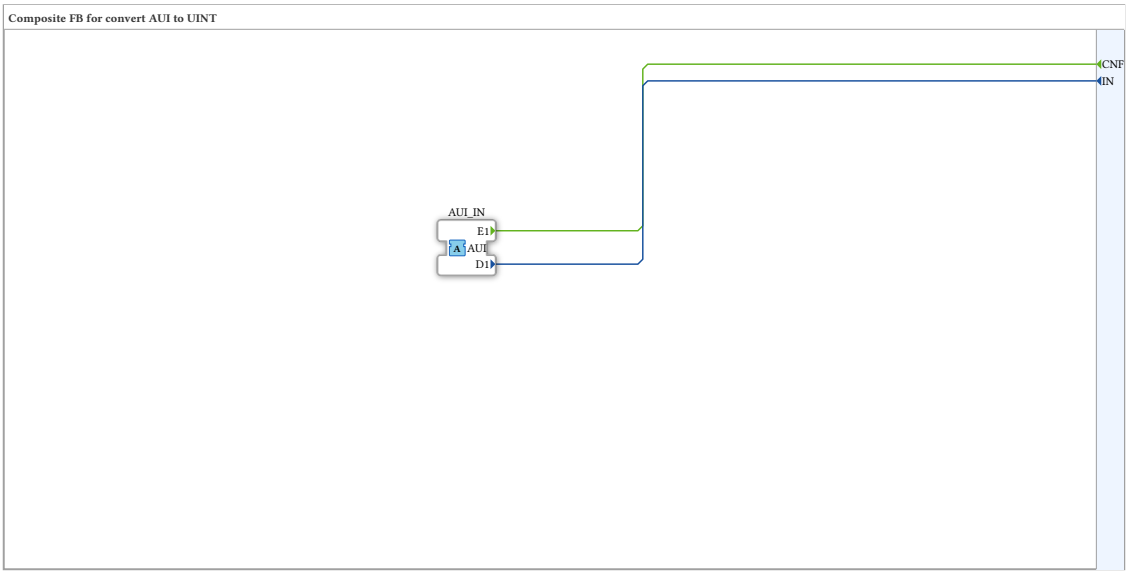
Adapter

- **AUI_OUT**: Plug vom Typ `AUI`

Funktionsweise

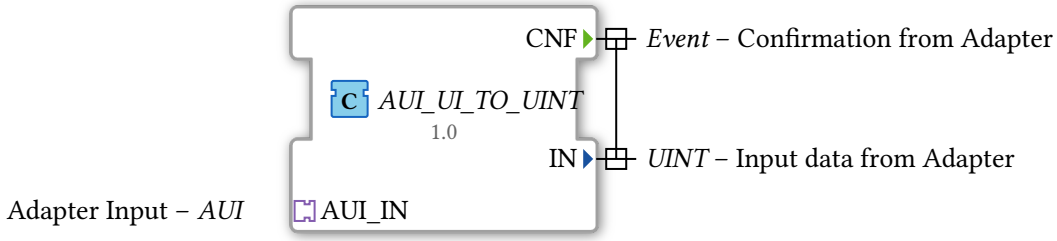
Der Funktionsblock empfängt einen UINT-Wert am Daten-Eingang OUT und wandelt diesen bei Eintreffen des REQ-Ereignisses in das AUI-Adapterformat um.

AUI_UI_TO_UINT



Einleitung

Der AUI_UI_TO_UINT Funktionsblock ist ein Composite-Funktionsblock, der zur Konvertierung des AUI-Adapterformats in einen UINT-Wert dient.



Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkten Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

- **CNF**: Bestätigungs-Event nach erfolgter Konvertierung

Daten-Eingänge

- Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

- **IN**: UINT-Ausgangswert der konvertierten Daten

Adapter

- **AUI_IN**: Socket vom Typ `AUI`

Funktionsweise

Der Funktionsblock empfängt Daten über den AUI-Adapter-Socket und stellt den enthaltenen UINT-Wert am Ausgang IN zur Verfügung, sobald ein Ereignis über den Adapter eintrifft (CNF).

UINT Conversion

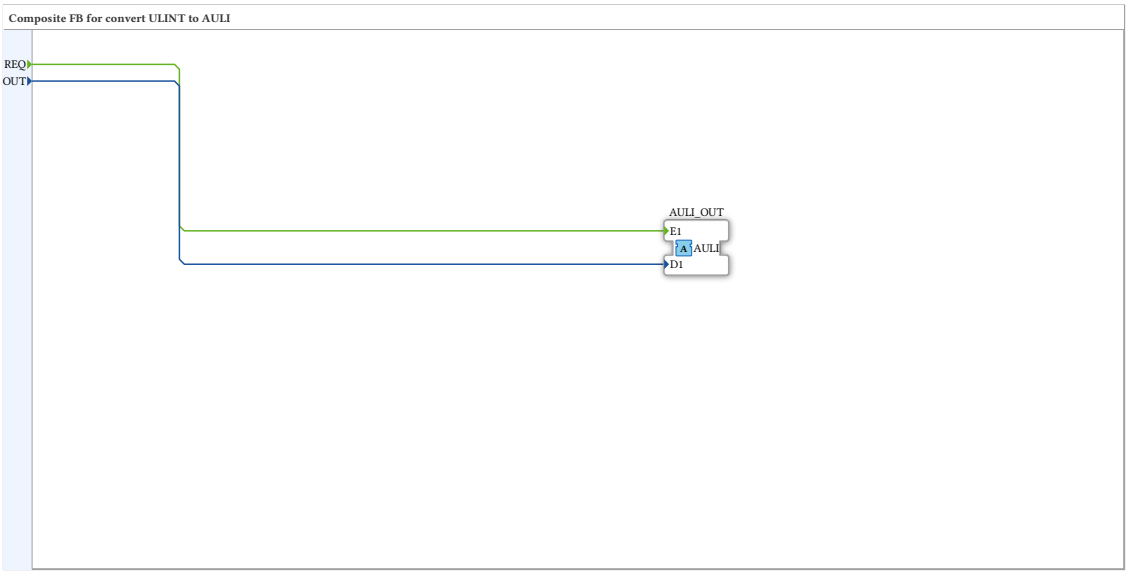
🛠️ Zugehörige Übungen

- Uebung_006c
- Uebung_010b4_sub_AX
- Uebung_010b5_sub_AX
- Uebung_010c2
- Uebung_010c2_AX
- AuxIXA_TO_logiBUS_QXA_BG
- NumericValue_ID_TO_NVS
- Uebung_020c2_sub
- IX_TO_logiBUS_LED_strip_QX
- Uebung_035
- Uebung_036
- Uebung_037
- Uebung_038
- Uebung_038_AX
- SINT_TO_Q_NumericValue
- SoftkeyOverride_TO_QX_BG
- Softkey_T_FF_ILOCK_TO_QX_BG
- Uebung_040
- Uebung_040_AX
- Uebung_041
- Uebung_087
- Uebung_087a1
- Uebung_090a1
- Uebung_090a1_AX
- Uebung_090a2

- Uebung_090a2_AX
- Uebung_103

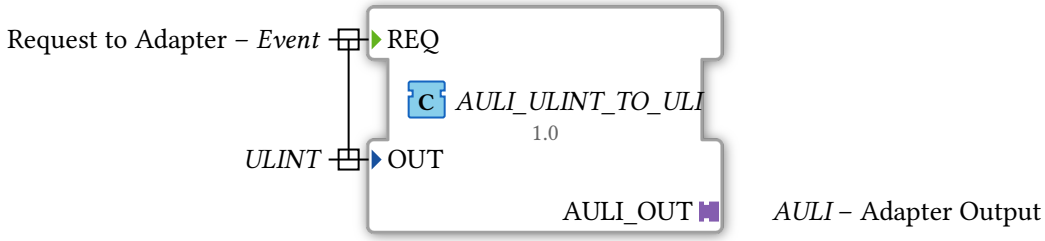
ULINT

AULI_ULINT_TO_ULI



Einleitung

Der AULI_ULINT_TO_ULI Funktionsblock ist ein Composite-Funktionsblock, der zur Konvertierung eines ULINT-Werts in das AULI-Adapterformat dient.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Anfrage-Event zur Auslösung der Konvertierung

Ereignis-Ausgänge

- Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

- **OUT**: ULINT-Eingangswert, der konvertiert werden soll

Daten-Ausgänge

- Keine direkten Daten-Ausgänge vorhanden

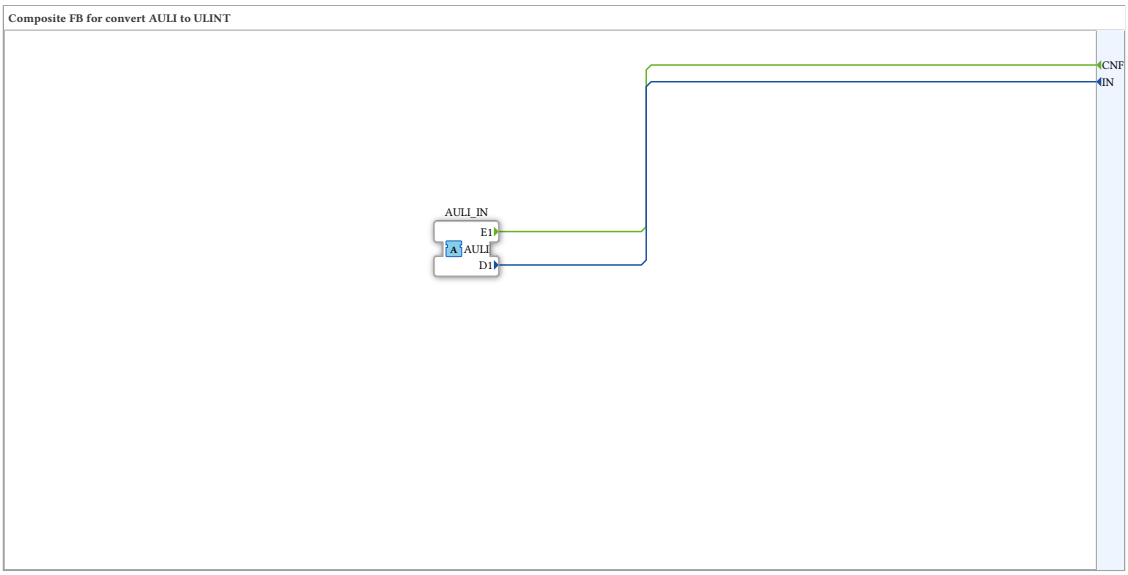
Adapter

- **AULI_OUT**: Plug vom Typ `AULI`

Funktionsweise

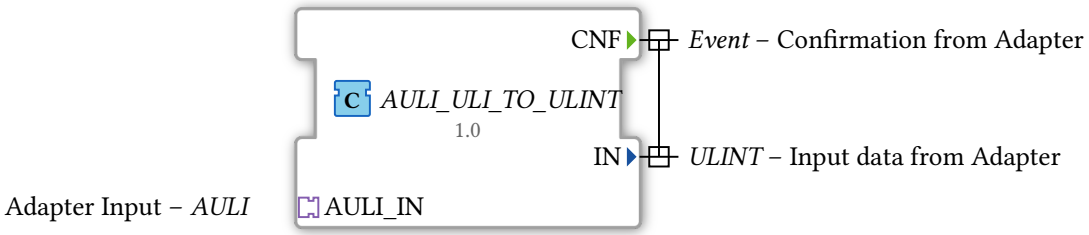
Der Funktionsblock empfängt einen ULINT-Wert am Daten-Eingang OUT und wandelt diesen bei Eintreffen des REQ-Ereignisses in das AULI-Adapterformat um.

AULI_ULI_TO_ULINT



Einleitung

Der AULI_ULI_TO_ULINT Funktionsblock ist ein Composite-Funktionsblock, der zur Konvertierung des AULI-Adapterformats in einen ULINT-Wert dient.



Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkten Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

- **CNF**: Bestätigungs-Event nach erfolgter Konvertierung

Daten-Eingänge

- Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

- **IN**: ULINT-Ausgangswert der konvertierten Daten

Adapter

- **AULI_IN**: Socket vom Typ `AULI`

Funktionsweise

Der Funktionsblock empfängt Daten über den AULI-Adapter-Socket und stellt den enthaltenen ULINT-Wert am Ausgang IN zur Verfügung, sobald ein Ereignis über den Adapter eintrifft (CNF).

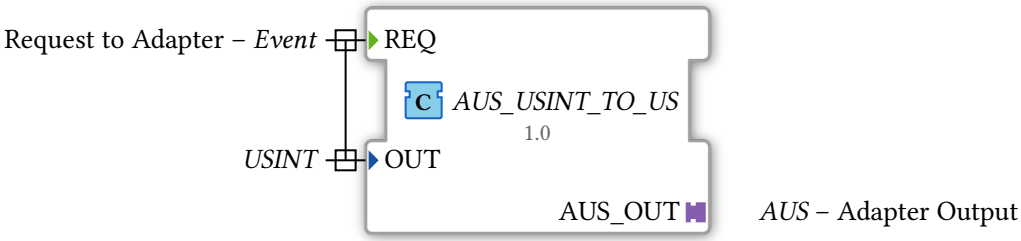
ULINT Conversion

USINT

AUS_USINT_TO_US

Einleitung

Der AUS_USINT_TO_US Funktionsblock ist ein Composite-Funktionsblock, der zur Konvertierung eines USINT-Werts in das AUS-Adapterformat dient.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Anfrage-Event zur Auslösung der Konvertierung

Ereignis-Ausgänge

- Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

- **OUT**: USINT-Eingangswert, der konvertiert werden soll

Daten-Ausgänge

- Keine direkten Daten-Ausgänge vorhanden

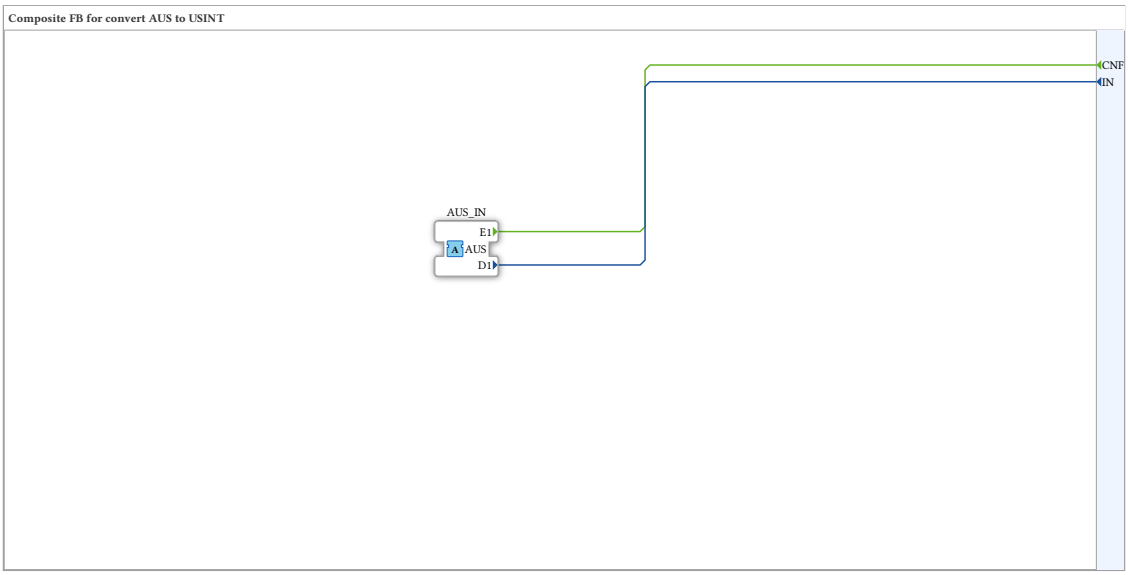
Adapter

- **AUS_OUT**: Plug vom Typ `AUS`

Funktionsweise

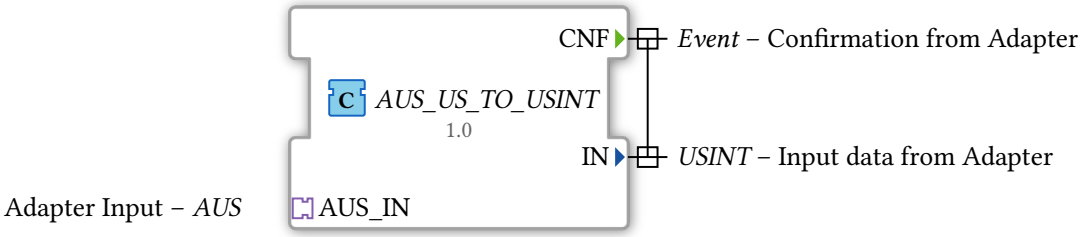
Der Funktionsblock empfängt einen USINT-Wert am Daten-Eingang OUT und wandelt diesen bei Eintreffen des REQ-Ereignisses in das AUS-Adapterformat um.

AUS_US_TO_USINT



Einleitung

Der AUS_US_TO_USINT Funktionsblock ist ein Composite-Funktionsblock, der zur Konvertierung des AUS-Adapterformats in einen USINT-Wert dient.



Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkten Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

- **CNF**: Bestätigungs-Event nach erfolgter Konvertierung

Daten-Eingänge

- Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

- **IN**: USINT-Ausgangswert der konvertierten Daten

Adapter

- **AUS_IN**: Socket vom Typ `AUS`

Funktionsweise

Der Funktionsblock empfängt Daten über den AUS-Adapter-Socket und stellt den enthaltenen USINT-Wert am Ausgang IN zur Verfügung, sobald ein Ereignis über den Adapter eintrifft (CNF).

USINT Conversion

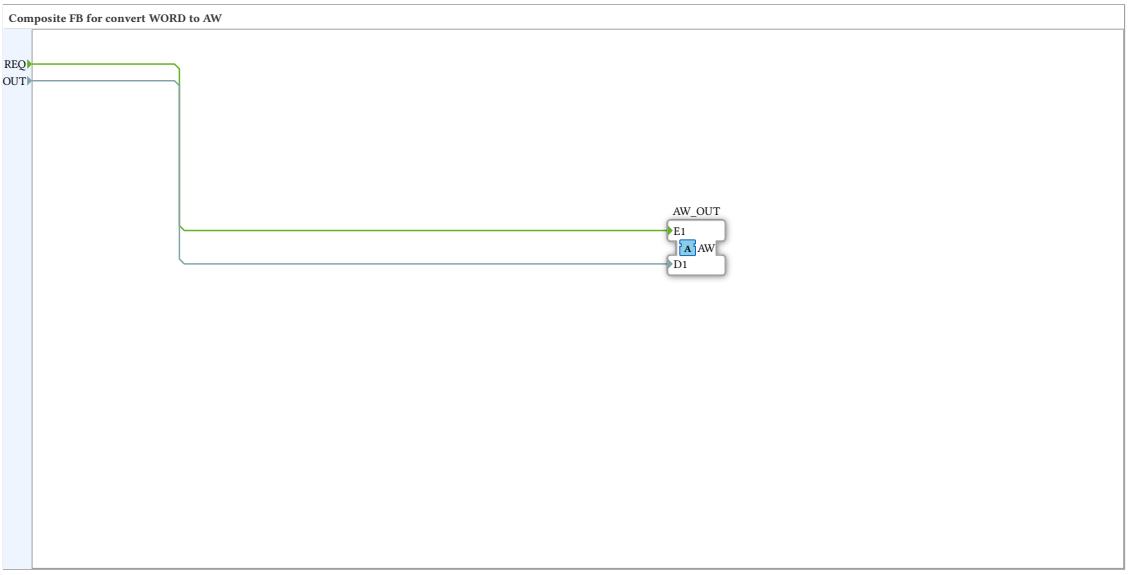
🛠️ Zugehörige Übungen

- Funk_IX_TO_DataPanel_MI_QX

- Funk_IXA_TO_DataPanel_MI_QXA
- IX_TO_Aux_QX
- IXA_TO_Aux_QXA
- IX_TO_logiBUS_LED_strip_QX
- SIN_TO_PCAN_Callback_Byte

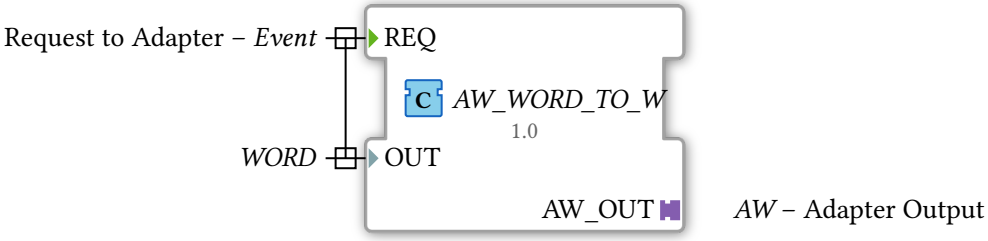
WORD

AW_WORD_TO_W



Einleitung

Der AW_WORD_TO_W Funktionsblock ist ein Composite-Funktionsblock, der zur Konvertierung eines WORD-Werts in das AW-Adapterformat dient.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Anfrage-Event zur Auslösung der Konvertierung

Ereignis-Ausgänge

- Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

- **OUT**: WORD-Eingangswert, der konvertiert werden soll

Daten-Ausgänge

- Keine direkten Daten-Ausgänge vorhanden

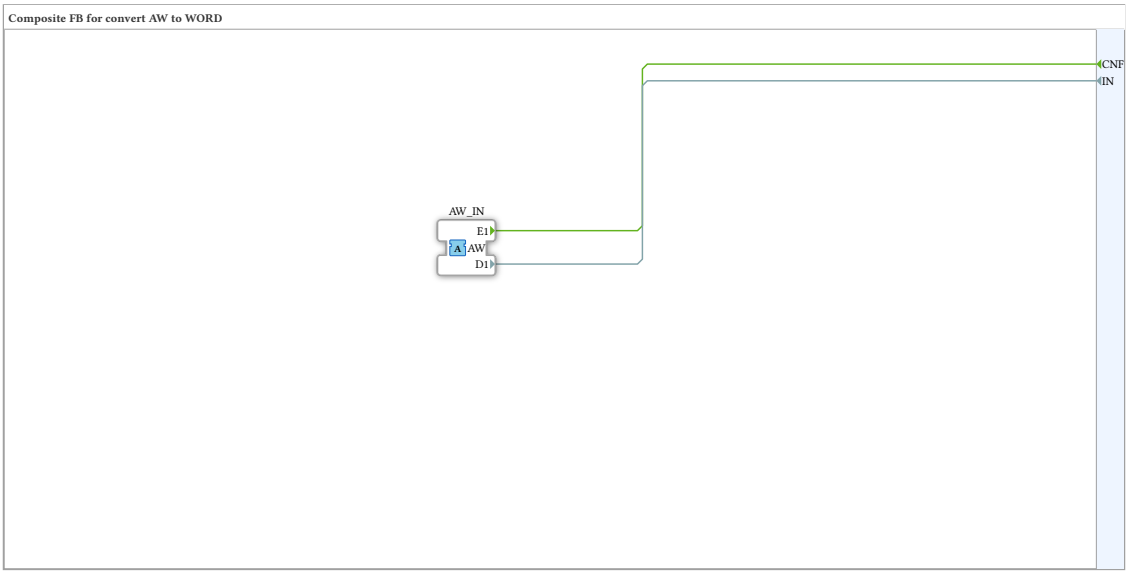
Adapter

- **AW_OUT**: Plug vom Typ `AW`

Funktionsweise

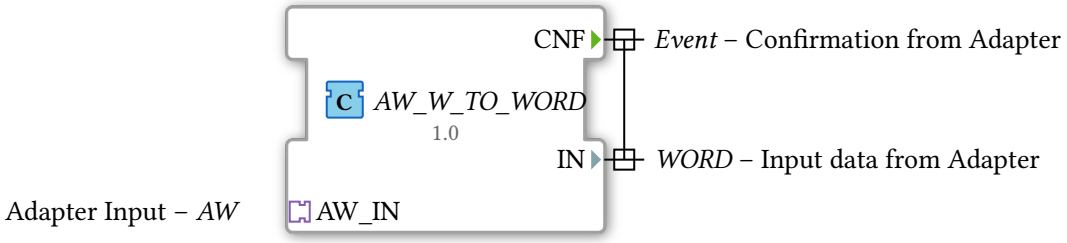
Der Funktionsblock empfängt einen WORD-Wert am Daten-Eingang OUT und wandelt diesen bei Eintreffen des REQ-Ereignisses in das AW-Adapterformat um.

AW_W_TO_WORD



Einleitung

Der AW_W_TO_WORD Funktionsblock ist ein Composite-Funktionsblock, der zur Konvertierung des AW-Adapterformats in einen WORD-Wert dient.



Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkten Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

- **CNF**: Bestätigungs-Event nach erfolgter Konvertierung

Daten-Eingänge

- Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

- **IN**: WORD-Ausgangswert der konvertierten Daten

Adapter

- **AW_IN**: Socket vom Typ `AW`

Funktionsweise

Der Funktionsblock empfängt Daten über den AW-Adapter-Socket und stellt den enthaltenen WORD-Wert am Ausgang IN zur Verfügung, sobald ein Ereignis über den Adapter eintrifft (CNF).

WORD Conversion

2.1.11.9 events

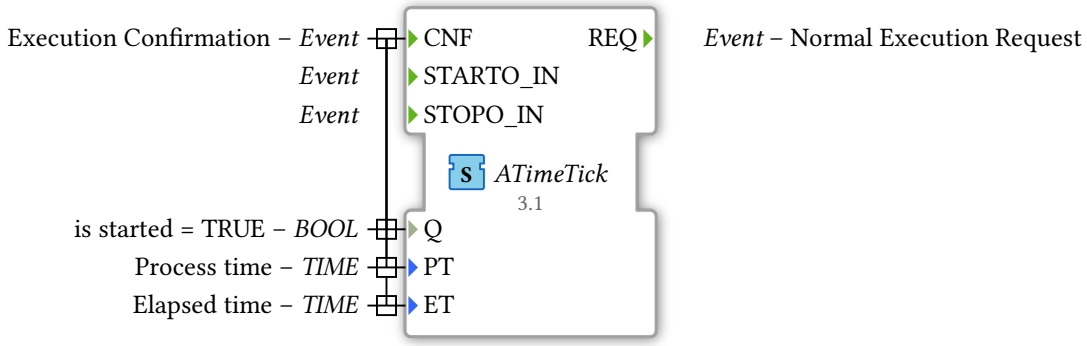
🎧 Podcast

- Taster & Events: GPIO-Grundlagen für Land- und Baumaschinen Mechatroniker

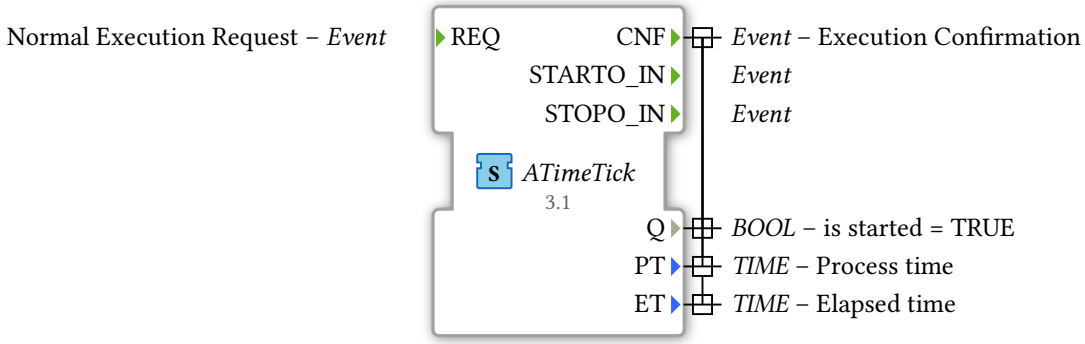
---

TimeOut

ATimeTick
Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Adapter `ATimeTick` stellt eine standardisierte Schnittstelle für einen Zeitüberwachungsdienst (Time-out-Service) bereit, der sich an den Grunddefinitionen der ROOM-Methodik orientiert. Er ermöglicht die lose Kopplung zwischen einer steuernden Komponente und einer Zeitgeber- oder Überwachungsinstanz. Die Interaktion erfolgt über Ereignisse und Zeitparameter, wobei der Adapter sowohl als Plug- als auch als Socket-Typ eingesetzt werden kann.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Kommentar
`CNF`	Bestätigung der Ausführung (Execution Confirmation), trägt die Daten `Q`, `ET` und `PT`.
`STARTO_IN`	Startanforderung von außen (ohne Parameter).
`STOPO_IN`	Stoppanforderung von außen (ohne Parameter).

Ereignis-Ausgänge

Ereignis	Kommentar
REQ	Normale Ausführungsanforderung (Normal Execution Request).

Daten-Eingänge

Variable	Typ	Kommentar
Q	BOOL	Zeigt an, ob der Timer aktiv ist (TRUE = gestartet).
PT	TIME	Prozesszeit (Sollwert für die Zeitüberwachung).
ET	TIME	Verstrichene Zeit (Istwert).

Daten-Ausgänge
Keine vorhanden.

Adapter
Der Adapter definiert zwei Service-Sequenzen:

- **Timeout**: Über die Plug-Seite wird das Ereignis START mit Parameter TD gesendet und auf der Socket-Seite weitergegeben. Anschließend wird von der Socket-Seite das Ereignis TimeOut empfangen und auf der Plug-Seite weitergeleitet.
- **NormalOperation**: Die Plug-Seite sendet Start (mit TD) und STOP (ohne Parameter) an die Socket-Seite.

Die Zuordnung der Ereignisse zu den Service-Sequenzen zeigt die erwartete Kommunikationsreihenfolge. Der Adapter unterstützt damit sowohl das Starten eines Timers als auch dessen Abbruch sowie die Rückmeldung eines Timeouts.

Funktionsweise
Der ATimeTick-Adapter dient als Vermittler zwischen einer Anwendungskomponente und einer Zeitüberwachungslogik. Die Steuerung kann über die Plug-Seite den Timer mit einer vorgegebenen Zeit (TD bzw. PT) starten. Die Zeitüberwachung (Socket-Seite) bestätigt den Start und liefert nach Ablauf einen Timeout-Ereignis zurück. Durch die ereignisgesteuerte Kommunikation wird die Zeitlogik von der eigentlichen Anwendung entkoppelt.

Die Daten Q, ET und PT werden über das CNF-Ereignis übertragen, das die Bestätigung der Ausführung darstellt. Über Q wird der aktive Zustand signalisiert, ET enthält die verstrichene Zeit und PT die eingestellte Prozesszeit. Damit stehen der Anwendung alle relevanten Informationen zur Verfügung.

Technische Besonderheiten

- **Ereignis- und Datenkopplung**: Beim CNF-Ereignis sind die Daten Q, ET und PT mitgebunden, was eine effiziente und synchrone Übertragung ermöglicht.
- **Plug/Socket-Prinzip**: Der Adapter kann wahlweise als Plug oder Socket verwendet werden, wodurch eine flexible Verschaltung in unterschiedlichen Kontexten möglich ist.
- **Service-Sequenzen**: Die definierten Sequenzen Timeout und NormalOperation beschreiben den erwarteten Nachrichtenaustausch und unterstützen die Verifikation und Implementierung.
- **Keine Datenausgänge**: Der Adapter kommuniziert primär über Ereignisse, Daten werden nur als Eingänge für die Bestätigung verwendet.

Zustandsübersicht
Der Adapter selbst besitzt keinen internen Zustandsautomaten, da er rein als Schnittstellentyp fungiert. Die möglichen Abläufe können jedoch aus den Service-Sequenzen abgeleitet werden:

- **Startzustand**: Über START/Start wird der Timer initialisiert.
- **Laufzustand**: Nach dem Start ist der Timer aktiv (Q = TRUE), die Zeit läuft.
- **Stoppzustand**: Über STOP wird der Timer vorzeitig beendet.
- **Timeout-Zustand**: Nach Ablauf der Zeit wird TimeOut ausgelöst, und die Anwendung erhält das Ereignis.

Die Zustandsübergänge sind ereignisgesteuert und hängen von der implementierenden Logik ab.

Anwendungsszenarien

- **Zeitüberwachung in industriellen Steuerungen**: Einsatz in SPS-Programmen zur Überwachung von Prozesszeiten, z.B. für Last- oder Laufzeitüberwachung.
- **Wartungsintervalle**: Überwachung von Wartungszeiten oder Bearbeitungsfristen in Automatisierungssystemen.
- **Ereignisgesteuerte Zeitgeber**: In verteilten Systemen kann der Adapter verwendet werden, um Zeitgeberfunktionen über verschiedene Komponenten hinweg zu synchronisieren.

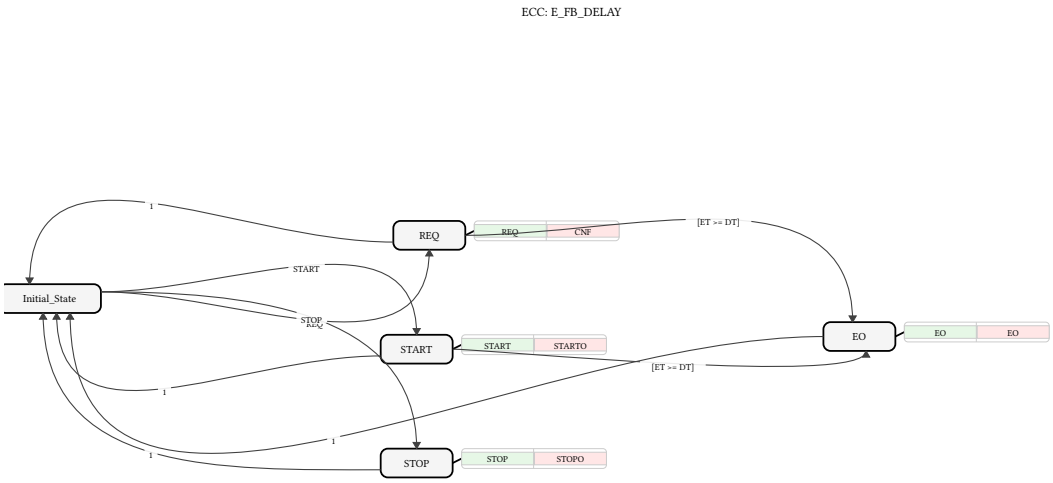
Vergleich mit ähnlichen Bausteinen

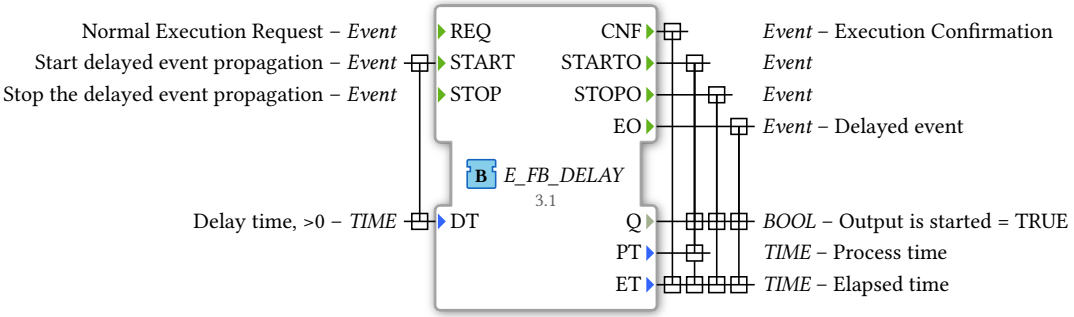
Im Vergleich zu einem klassischen `TON`-Funktionsblock (Timer ON-Delay) bietet der `ATimeTick`-Adapter eine isolierte Schnittstelle, die vom eigentlichen Zeitverhalten entkoppelt ist. Während ein FB die Zeitlogik direkt implementiert, definiert der Adapter nur die Kommunikationsprotokolle. Dadurch eignet er sich für modulare und wiederverwendbare Architekturen, bei denen die Zeitlogik austauschbar sein soll.

Fazit

Der `ATimeTick`-Adapter ist eine elegante Lösung zur Integration von Zeitüberwachungsdiensten in ereignisbasierte Automatisierungssysteme. Durch die klare Trennung von Schnittstelle und Implementierung ermöglicht er eine flexible und wartungsfreundliche Systemarchitektur. Die definierten Service-Sequenzen erleichtern die korrekte Einbindung und Kommunikation, insbesondere in Umgebungen mit verteilten Komponenten.

E_FB_DELAY





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **E_FB_DELAY** realisiert eine verzögerte Ereignisweiterleitung. Ein eingehendes Ereignis (z. B. ein Startsignal) wird nach Ablauf einer konfigurierbaren Zeitverzögerung als Ausgangsereignis ausgegeben. Der Block ähnelt funktional einem Einschaltverzögerer (TON) aus der IEC-61131-Welt, ist jedoch für die ereignisgesteuerte Verarbeitung in 4diac-Anwendungen optimiert. Neben der Verzögerungsfunktion bietet er einen zyklischen Abfrage-Mechanismus, um die abgelaufene Zeit aktuell zu halten.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar
REQ	Event	Normaler Ausführungsaufruf; aktualisiert ET, solange der Timer aktiv ist.
START	Event	Startet die verzögerte Ereignisweitergabe.
STOP	Event	Stoppt die Verzögerung und setzt den Timer zurück.

Ereignis-Ausgänge

Ereignis	Typ	Kommentar
CNF	Event	Bestätigung für den REQ-Aufruf (mit ET, REST).
STARTO	Event	Wird bei erfolgreichem Start ausgegeben.
STOPO	Event	Wird bei erfolgreichem Stopp ausgegeben.
EO	Event	Verzögertes Ereignis; wird ausgelöst, wenn die Verzögerungszeit abgelaufen ist.

Daten-Eingänge

Variable	Typ	Kommentar
DT	TIME	Verzögerungszeit, muss größer als 0 sein.

Daten-Ausgänge

Variable	Typ	Kommentar
Q	BOOL	Zeigt an, ob der Timer aktiv ist (TRUE = gestartet).
PT	TIME	Vorgabezeit (SOLL); wird bei START auf DT gesetzt.
ET	TIME	Abgelaufene Zeit seit dem letzten Start.
REST	TIME	Restzeit = PT - ET.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock verwendet eine interne Zeitmessung auf Basis von NOW_MONOTONIC(), die eine monotone Uhr ohne Systemzeit-Springe bereitstellt. Der Ablauf gliedert sich in drei Phasen:

1. **Start** – Durch ein START-Ereignis wird die Startzeit aufgezeichnet, Q auf TRUE gesetzt und ET auf 0 zurückgesetzt. Das Ereignis STARTO wird ausgegeben.
2. **Zyklische Abfrage** – Solange Q = TRUE, muss das REQ-Ereignis regelmäßig (z. B. von einem E_CYCLE-Baustein) aufgerufen werden. Bei jedem REQ wird ET als Differenz zwischen aktueller Zeit und Startzeit berechnet. Überschreitet ET die eingestellte Verzögerungszeit DT, wird das Ereignis EO ausgelöst.

3. **Abschluss oder Stopp** – Beim Auslösen von `EO` werden `Q` auf `FALSE` und `ET` auf 0 gesetzt. Ein `STOP`-Ereignis kann die Verzögerung jederzeit abbrechen; auch hier werden `Q` und `ET` zurückgesetzt und `STOPO` wird ausgegeben.

Nach einem `START` kann nur ein einziger `EO`-Ausgang ausgelöst werden. Erst ein erneutes `START` aktiviert den Timer wieder.

Technische Besonderheiten

- **Zeitbasis**: Verwendet `NOW_MONOTONIC()` – eine monotone Uhr, die unabhängig von Systemzeit-Anpassungen ist.
- **Keine eigene Zeitbasis**: Der FB besitzt keinen eigenen Timer-Interrupt, sondern muss über das `REQ`-Ereignis zyklisch aufgerufen werden. Dadurch ist die Aktualität von `ET` abhängig von der Aufruffrequenz.
- **Verzögerungsauslösung** erfolgt über die Bedingung `[ET >= DT]` im ECC. Da nach der Auslösung von `EO` `ET` auf 0 gesetzt wird, kann ein späteres `REQ` nicht erneut ein `EO` auslösen, solange kein neuer `START` erfolgt.
- **`PT`** wird bei `START` auf `DT` gesetzt (Vorgabezeit); `REST` wird bei `START` auf `PT` gesetzt, bei `REQ` als `PT - ET` nachgeführt und bei `STOP`/`EO` auf 0 zurückgesetzt.

Zustandsübersicht

Der FB besitzt die folgenden Zustände im Basic-FB-Modell:

Zustand	Beschreibung
`Initial_State`	Ruhezustand; wartet auf `START`, `STOP` oder `REQ`.
`REQ`	Verarbeitet zyklische Abfragen; aktualisiert `ET` und gibt `CNF` aus.
`START`	Reagiert auf `START`, setzt Startzeit, aktiviert Timer und gibt `STARTO` aus.
`STOP`	Reagiert auf `STOP`, deaktiviert Timer und gibt `STOPO` aus.
`EO`	Endzustand der Verzögerung; setzt `Q` und `ET` zurück und gibt `EO` aus.

Übergänge:

- `Initial_State` ⟶ `REQ` bei `REQ`-Ereignis
- `Initial_State` ⟶ `START` bei `START`-Ereignis
- `Initial_State` ⟶ `STOP` bei `STOP`-Ereignis
- `REQ` ⟶ `EO` falls `[ET >= DT]`
- `REQ` ⟶ `Initial_State` sonst (immer)
- `START` ⟶ `EO` falls `[ET >= DT]` (sofort wenn DT=0)
- `START` ⟶ `Initial_State` sonst
- `STOP` ⟶ `Initial_State` (immer)
- `EO` ⟶ `Initial_State` (immer)

Anwendungsszenarien

- **Verzögerte Signalweiterleitung**: Ein Startsignal soll erst nach einer bestimmten Zeit an einen nachfolgenden Funktionsblock weitergegeben werden – z. B. zum Warten auf ein physikalisches System.
- **Zeitüberwachung**: Überwachung, ob ein Ereignis innerhalb einer vorgegebenen Zeitspanne eintritt; bei Überschreitung wird über `EO` ein Alarm ausgelöst.
- **Schutz- und Sicherheitslogik**: Zeitverzögerungen vor dem Auslösen kritischer Aktionen.
- **Test und Simulation**: Erzeugung von zeitversetzten Ereignissen für Testabläufe.

Vergleich mit ähnlichen Bausteinen

Baustein	Eigenschaften
E_FB_DELAY	Ereignisgesteuert, benötigt externe Taktung über `REQ`, `ET` wird nur bei Aufruf aktualisiert, Keine Selbsttaktung.
TON (IEC 61131)	Selbsttaktend, integrierte Zeitbasis, arbeitet kontinuierlich ohne externen Aufruf, Ausgang `Q` bleibt aktiv solange Eingang `IN` = TRUE.

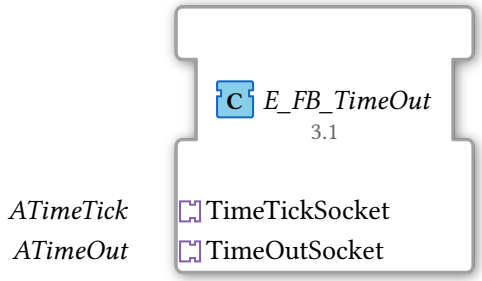
E_CYCLE + E_DELAY	Kombination aus zyklischem Taktgeber und Verzögerung; separate Bausteine, aber weniger flexibel in der Ereignisbehandlung.

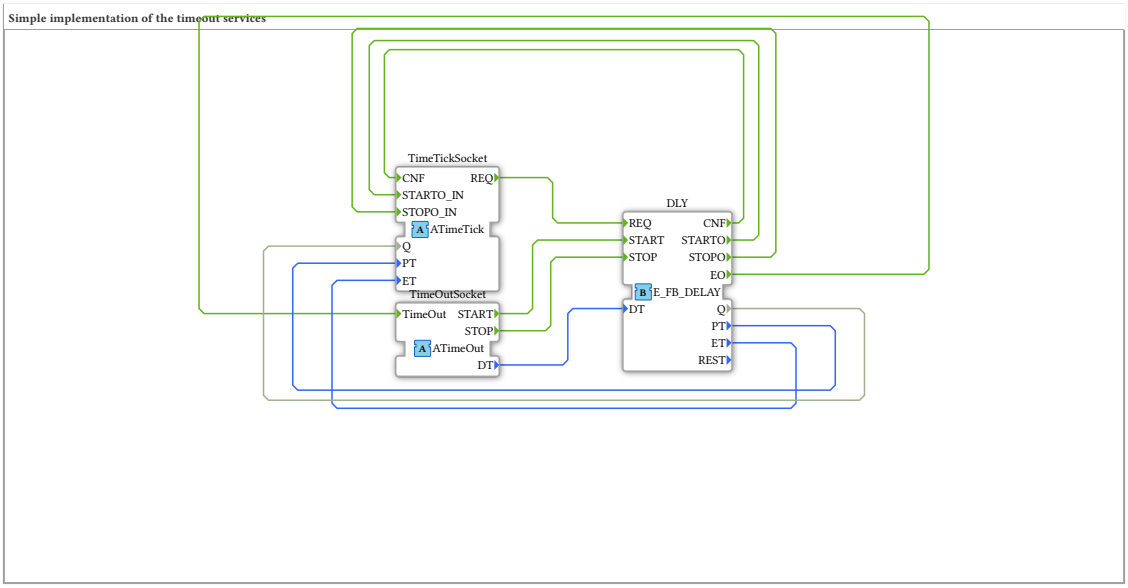
Der `E_FB_DELAY` bietet den Vorteil, dass er vollständig in eine ereignisbasierte 4diac-Architektur integriert ist und eine einfache Abfrage der abgelaufenen Zeit über `ET` erlaubt. Allerdings muss die Aufruffrequenz hoch genug sein, um eine präzise Verzögerung zu gewährleisten.

Fazit

Der **E_FB_DELAY** ist ein kompakter und flexibler Funktionsblock zur Realisierung von Zeitverzögerungen in ereignisgesteuerten Anwendungen. Durch seine klare Schnittstelle und die Verwendung einer monotonen Uhr eignet er sich besonders für industrielle Steuerungsszenarien, in denen eine zuverlässige, zyklische Zeitmessung erforderlich ist. Die Notwendigkeit eines externen `REQ`-Signals stellt eine Beeinträchtigung dar, die durch den Entwickler berücksichtigt werden muss, aber gleichzeitig eine präzise Kontrolle über die Aktualisierungsrate der Zeitmessung ermöglicht.

E_FB_TimeOut





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein `E_FB_TimeOut` bietet eine einfache Implementierung von Timeout-Diensten auf Basis eines internen Verzögerungsbausteins (`E_FB_DELAY`). Er ist als zusammengesetzter FB (Composite) konzipiert und verbindet eine Zeitgeber-Schnittstelle (`TimeTickSocket`) mit einer Timeout-Schnittstelle (`TimeOutSocket`). Die Funktionalität entspricht einem TON-Baustein (Timer ON-Delay) mit zyklischer Abfrage über Ticks.

Schnittstellenstruktur

Der FB besitzt **keine direkten Ereignis- oder Datenein-/ausgänge**. Alle externen Anbindungen erfolgen ausschließlich über zwei **Adapter-Sockets**. Die tatsächlichen Signale werden über die jeweiligen Adaptertypen definiert.

Ereignis-Eingänge

- Nicht vorhanden (es werden keine direkten Ereignisse angebunden).

Ereignis-Ausgänge

- Nicht vorhanden (es werden keine direkten Ereignisse ausgegeben).

Daten-Eingänge

- Nicht vorhanden (es werden keine direkten Daten angebunden).

Daten-Ausgänge

- Nicht vorhanden (es werden keine direkten Daten ausgegeben).

Adapter

Socket	Typ	Beschreibung
`TimeTickSocket`	`adapter::events::TimeOut::ATimeTick`	Adapter für die zyklische Zeitabfrage (Tick-Schnittstelle). Stellt Ereignisse wie `REQ`, `CNF`, `STARTO_IN`, `STOPO_IN` sowie Daten `ET`, `Q`, `PT` bereit.
`TimeOutSocket`	`iec61499::events::ATimeOut`	Adapter für den eigentlichen Timeout-Dienst. Bietet Ereignisse `START`, `STOP`, `TimeOut` sowie Daten `DT`, `ET`, `Q`, `PT`.

- **`ATimeTick`** (Zeitgeber): Empfängt `REQ` (Anforderung für einen Tick), liefert `CNF` (Bestätigung) und signalisiert Start/Stopp über `STARTO_IN`/`STOPO_IN`. Daten umfassen die verstrichene Zeit `ET`, das Ausgangssignal `Q` und die voreingestellte Zeit `PT`.
- **`ATimeOut`** (Timeout-Dienst): Startet und stoppt den Timeout über `START`/`STOP`, generiert das Ereignis `TimeOut` nach Ablauf. Daten umfassen die Zielzeit `DT`, die verstrichene Zeit `ET`, den Ausgangsstatus `Q` und die Zeitvorgabe `PT`.

Funktionsweise

Der FB realisiert einen timerbasierten Timeout wie folgt:

- **Ablauf**: Über `TimeOutSocket.START` wird der interne `E_FB_DELAY` gestartet. Die Sollzeit wird über `TimeOutSocket.DT` an den Delay-Baustein übergeben.
- **Zeitbasis**: Der Zeitgeber wird über `TimeTickSocket.REQ` mit einem zyklischen Impuls versorgt. Jeder dieser Ticks veranlasst den Delay-Baustein, seine interne Zeit zu aktualisieren und zu prüfen, ob die vorgegebene Zeit erreicht ist.
- **Abschluss**: Sobald die Zeit abgelaufen ist, erzeugt der Delay-Baustein ein Ereignis am Ausgang `EO`. Dieses wird als `TimeOut` über den `TimeOutSocket` ausgegeben.
- **Stopp**: Über `TimeOutSocket.STOP` wird der Delay-Baustein angehalten. Entsprechende Rückmeldungen über `CNF`, `STARTO_IN`, `STOPO_IN` und die Daten `ET`, `Q`, `PT` werden über den jeweils passenden Adapter in beide Richtungen kommuniziert.

Die Verbindungen im internen Netzwerk spiegeln exakt diese Kommunikation wider:

- `TimeOutSocket.START` ⟶ `DLY.START`
- `TimeOutSocket.STOP` ⟶ `DLY.STOP`
- `DLY.EO` ⟶ `TimeOutSocket.TimeOut`
- `TimeTickSocket.REQ` ⟶ `DLY.REQ`
- `DLY.CNF` ⟶ `TimeTickSocket.CNF`
- `DLY.STARTO` ⟶ `TimeTickSocket.STARTO_IN`
- `DLY.STOPO` ⟶ `TimeTickSocket.STOPO_IN`

Daten:

- `TimeOutSocket.DT` ⟶ `DLY.DT`
- `DLY.ET` ⟶ `TimeTickSocket.ET`
- `DLY.Q` ⟶ `TimeTickSocket.Q`
- `DLY.PT` ⟶ `TimeTickSocket.PT`

Technische Besonderheiten

- **Adapterbasiert**: Die gesamte Funktionalität ist in Adapterstrukturen gekapselt, wodurch eine hohe Wiederverwendbarkeit und modulare Einbindung in größere Systeme erreicht wird.
- **Zyklische Zeitbasis**: Der Timeout wird nicht über einen eigenständigen Hardware-Timer, sondern über externe Ticks (z. B. von einem zyklischen Taktgeber) realisiert. Dies erlaubt eine präzise Synchronisation mit anderen zyklischen Prozessen.
- **Kompatibilität**: Laut Versionshinweisen wurde der FB in Anlehnung an das Standard-TON-Verhalten (IEC 61131-3) entwickelt, jedoch mit einem ereignisgesteuerten Adapterinterface.
- **Flexible Parametrierung**: Die Zeitvorgabe (`PT`) und die Zielzeit (`DT`) werden über die Adapterdaten übergeben; dadurch ist der FB dynamisch konfigurierbar.

Zustandsübersicht

Der `E_FB_TimeOut` besitzt keine eigenen expliziten Zustände, da die Logik vollständig im internen `E_FB_DELAY` implementiert ist. Das Verhalten entspricht einem klassischen Timer-Zustandsautomaten:

- **Idle**: Kein Timeout aktiv.
- **Counting**: Nach `START` läuft der Zähler, solange `STOP` nicht gesetzt wurde.
- **Expired**: Nach Erreichen der eingestellten Zeit wird `TimeOut` generiert, der Zähler bleibt solange aktiv, bis `STOP` oder ein erneuter `START` (Reset) erfolgt.

Die Zustands-Rückmeldungen (`Q`, `ET`) werden über die Adapter an die angeschlossenen Applikationen kommuniziert.

Anwendungsszenarien

- **Überwachung von Kommunikationsprotokollen**: Einsatz als Kommunikationstimeout, z. B. um auf fehlende Antworten in Bussystemen zu reagieren.
- **Prozesssteuerung**: Absicherung von zeitkritischen Schritten in Fertigungsanlagen.
- **Test- und Diagnosesysteme**: Erzeugung definierter Verzögerungen oder Timeout-Auslösungen in Testumgebungen.
- **Energiemanagement**: Abschaltung von Komponenten nach einer festgelegten Inaktivitätszeit.

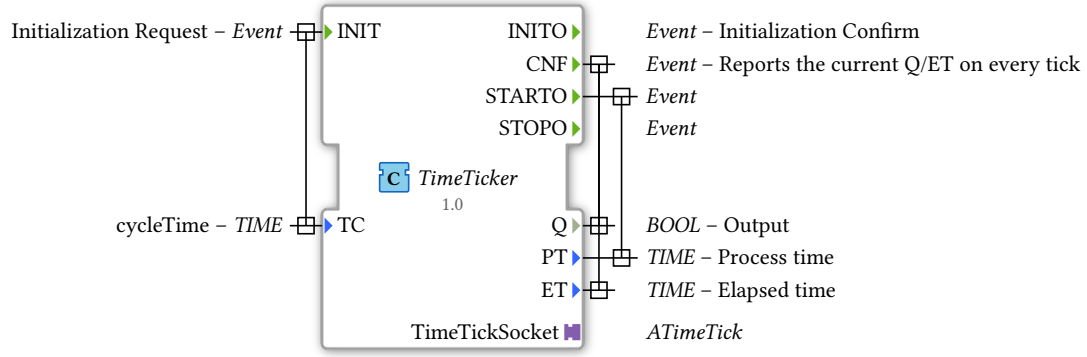
Vergleich mit ähnlichen Bausteinen

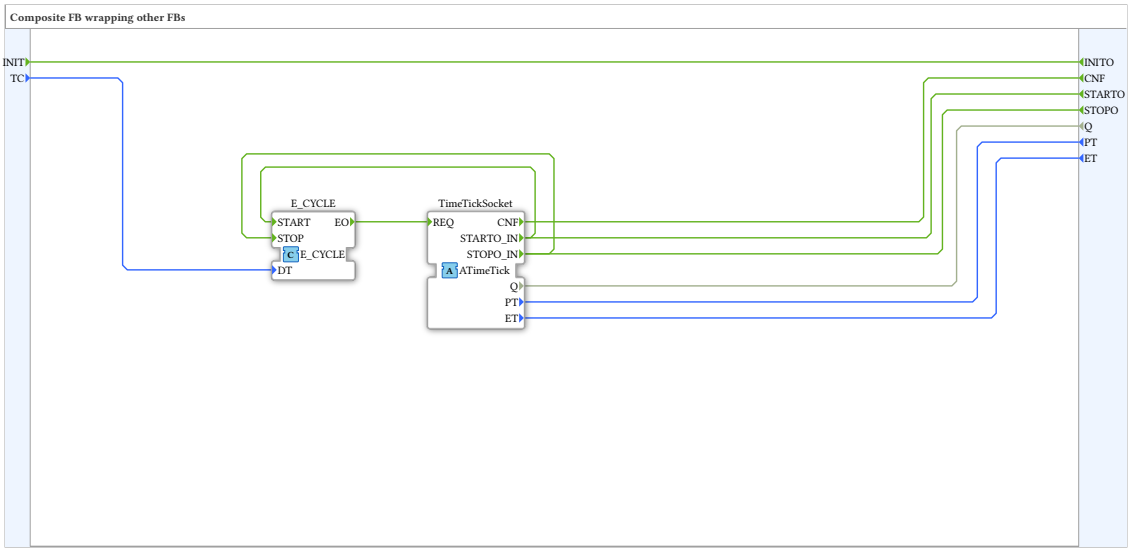
- **Standard-TON (IEC 61131-3)**: Basiert auf einer festen, internen Zeitbasis; im Gegensatz dazu verwendet `E_FB_TimeOut` eine externe Tick-Schnittstelle, was eine flexiblere Integration in ereignisgesteuerte Systeme ermöglicht.
- **Andere Timeout-FBs (z. B. `E_FB_Timer`)**: Diese verwenden oft interne Timer-Ressourcen; `E_FB_TimeOut` delegiert die Zeitmessung an einen internen Delay-FB und nutzt Adapter, wodurch eine saubere Trennung von Logik und Zeitgeber erfolgt.
- **Adapter-basierte Timeouts**: Im Gegensatz zu reinen Funktionsblöcken mit direkten Ein-/Ausgängen bietet dieser FB eine standardisierte, wiederverwendbare Schnittstelle, die sich leicht in existierende Adapter-Netzwerke einfügt.

Fazit

Der `E_FB_TimeOut` ist ein kompakter, adapterorientierter Baustein, der einen zuverlässigen Timeout-Mechanismus auf Basis eines zyklischen Zeitgebers realisiert. Durch die klare Trennung von Zeitgeber- und Timeout-Schnittstelle eignet er sich besonders für modulare Systeme, in denen eine flexible und erweiterbare Zeitsteuerung benötigt wird. Die Implementierung folgt bewährten IEC-61499-Strukturen und kann in verschiedenen Anwendungskontexten ohne großen Aufwand integriert werden.

TimeTicker





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **TimeTicker** ist ein zusammengesetzter Baustein (Composite FB), der einen zyklischen Zeitgeber realisiert. Er basiert auf dem Standardbaustein `E_CYCLE` und erweitert dessen Funktionalität durch eine Adapter-Schnittstelle zur Kommunikation mit anderen Bausteinen. Der TimeTicker erzeugt periodische Ereignisse (Ticks) und stellt dabei Informationen über die aktuelle Zykluszeit, die vergangene Zeit und den Prozesszustand bereit. Er eignet sich für Anwendungen, in denen eine zeitgesteuerte Ablaufsteuerung mit Rückmeldung über den aktuellen Zustand benötigt wird, beispielsweise in der Prozessautomatisierung oder bei zeitbasierten Steuerungsaufgaben.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar
`INIT`	EInit	Initialisierungsanforderung. Mit `TC` verbunden.

Ereignis-Ausgänge

Ereignis	Typ	Kommentar
`INITO`	EInit	Bestätigung der Initialisierung.
`CNF`	Event	Meldet bei jedem Tick den aktuellen Wert von `Q` und `ET`.
`STARTO`	Event	Ausgangssignal zum Starten des Tickers (wird vom Adapter weitergegeben).

STOPO	Event	Ausgangssignal zum Stoppen des Tickers (wird vom Adapter weitergegeben).

Daten-Eingänge

Name	Typ	Initialwert	Kommentar
TC	TIME	T#200ms	Zykluszeit (cycle time).

Daten-Ausgänge

Name	Typ	Kommentar
Q	BOOL	Ausgangszustand (z. B. aktiv/inaktiv).
PT	TIME	Prozesszeit (process time).
ET	TIME	Verstrichene Zeit (elapsed time).

Adapter

Typ	Name	Richtung
adapter::events::TimeOut::ATimeTick	TimeTickSocket	Plug (Stecker)

Der Adapter TimeTickSocket dient als Schnittstelle zu einem übergeordneten Zeitgeber-Kommunikationsprotokoll. Er empfängt Start-/Stopp-Ereignisse vom Adapter und sendet Tick-Bestätigungen sowie Zeitdaten an den angeschlossenen Adapter.

Funktionsweise

Der TimeTicker nutzt intern den Baustein E_CYCLE, der einen periodischen Zyklus erzeugt. Die Funktion im Detail:

1. **Initialisierung**: Beim Ereignis INIT wird der Baustein initialisiert und das Ereignis INITO sofort ausgelöst, um die Bereitschaft zu bestätigen.
2. **Start/Stopp über Adapter**: Die Ereignisse STARTO_IN und STOPO_IN vom Adapter (über TimeTickSocket) werden direkt an den internen E_CYCLE weitergeleitet, um den Zyklus zu starten bzw. zu stoppen. Gleichzeitig werden diese Ereignisse als STARTO bzw. STOPO an den Ausgang gegeben, damit angeschlossene Bausteine den Zustandswechsel ebenfalls mitbekommen.
3. **Tick-Erzeugung**: Der E_CYCLE erzeugt bei jedem Ablauf des Zyklus (konfiguriert über TC) das Ereignis EO. Dieses wird über den Adapter (TimeTickSocket.REQ) nach außen gesendet (z. B. an einen anderen Baustein).
4. **Rückmeldung**: Nach jedem Tick kommt vom Adapter das Ereignis CNF (Confirmation), das die aktuellen Werte von Q, ET und PT mit sich führt. Dieses Ereignis wird als CNF am Ausgang des TimeTickers nach außen gegeben. Die Daten Q, ET und PT werden aus den vom Adapter gelieferten Werten übernommen und an den jeweiligen Datenausgängen bereitgestellt.
5. **Datenfluss**: Die Zykluszeit TC wird direkt als DT (Delta-Time) an den E_CYCLE übergeben. Die Ausgabedaten Q, PT und ET stammen aus dem Adapter (TimeTickSocket.Q, TimeTickSocket.PT, TimeTickSocket.ET) und werden an die entsprechenden Ausgangs-Pins weitergeleitet.

Somit fungiert der TimeTicker als eine Art Wrapper, der die Steuerung eines zyklischen Zeitgebers über einen Adapter ermöglicht und die dabei anfallenden Zeitdaten nach außen transparent macht.

Technische Besonderheiten

- **Adapterbasiert**: Die Kommunikation mit externen Bausteinen erfolgt ausschließlich über den Adapter TimeTickSocket. Dadurch ist der TimeTicker klar vom eigentlichen Zeitgeber-Mechanismus entkoppelt.
- **Durchreichung von Ereignissen**: Start- und Stopp-Signale werden sowohl an den internen Zyklus als auch direkt an die Ausgänge weitergegeben – dies erlaubt eine Synchronisation mehrerer Bausteine.
- **Initialisierung ohne Verzögerung**: Das INIT-Ereignis wird sofort mit INITO beantwortet, ohne dass ein interner Zustandswechsel abgewartet wird.
- **Nutzung von E_CYCLE**: Die eigentliche Zeitsteuerung basiert auf dem Standard-FB E_CYCLE aus der IEC 61499 Bibliothek, was eine bewährte und robuste Implementierung garantiert.

- **Zeitwerte**: Die Ausgabevariablen sind vom Typ `TIME` und erlauben eine präzise Verarbeitung im Mikrosekundenbereich.

Zustandsübersicht

Der TimeTicker besitzt keinen expliziten internen Zustandsautomaten, da er als Composite FB hauptsächlich die Zustände des internen `E_CYCLE` widerspiegelt. Man kann jedoch folgende logische Zustände ableiten:

- **Initialisiert**: Nach `INIT`/`INITO` ist der Baustein bereit.
- **Gestoppt**: Solange kein Start-Signal empfangen wurde oder nach einem Stopp-Signal, erzeugt der Ticker keine Ticks.
- **Laufend**: Nach Erhalt eines Start-Signals (über Adapter) läuft der Zyklus und erzeugt periodisch `EO`/`CNF`-Ereignisse.

Die Übergänge zwischen diesen Zuständen werden durch die Ereignisse `STARTO_IN` (Start) und `STOPO_IN` (Stopp) gesteuert, die über den Adapter eingehen.

Anwendungsszenarien

- **Zeitbasierte Ablaufsteuerung**: Einsatz in Fertigungsprozessen, wo periodische Aktionen ausgelöst werden müssen (z. B. Sensorabfragen, Datenlogging).
- **Kommunikation mit übergeordneten Systemen**: Der Adapter ermöglicht die Anbindung an einen übergeordneten Zeitgeber-Baustein oder eine Steuerung, die den TimeTicker fernsteuern kann.
- **Zeitmessung und Überwachung**: Mit den Ausgaben `PT` (Prozesszeit) und `ET` (verstrichene Zeit) können Laufzeiten von Teilprozessen gemessen und überwacht werden.
- **Test und Simulation**: In Testumgebungen kann der TimeTicker verwendet werden, um zeitabhängige Testfälle zu generieren.

Vergleich mit ähnlichen Bausteinen

- **E_CYCLE**: Der direkte Basisbaustein erzeugt periodische Ereignisse, besitzt aber keine Adapter-Schnittstelle und gibt keine Zeitinformationen aus. Der TimeTicker erweitert diese Funktionalität.
- **E_TIME** (falls vorhanden): Ein Baustein, der absolute Zeiten liefert, aber nicht für zyklische Ticks gedacht ist.
- **E_SWITCH**: Schaltet zwischen Eingängen um, hat aber keinen Zeitbezug.
- **Andere Ticker-Implementierungen**: Es gibt ähnliche Bausteine, die oft eigene Zeitbasis verwenden. Der Vorteil des TimeTickers liegt in der standardisierten `E_CYCLE`-Integration und der Adapter-Kopplung, die eine nahtlose Einbindung in bestehende IEC-61499-Systeme ermöglicht.

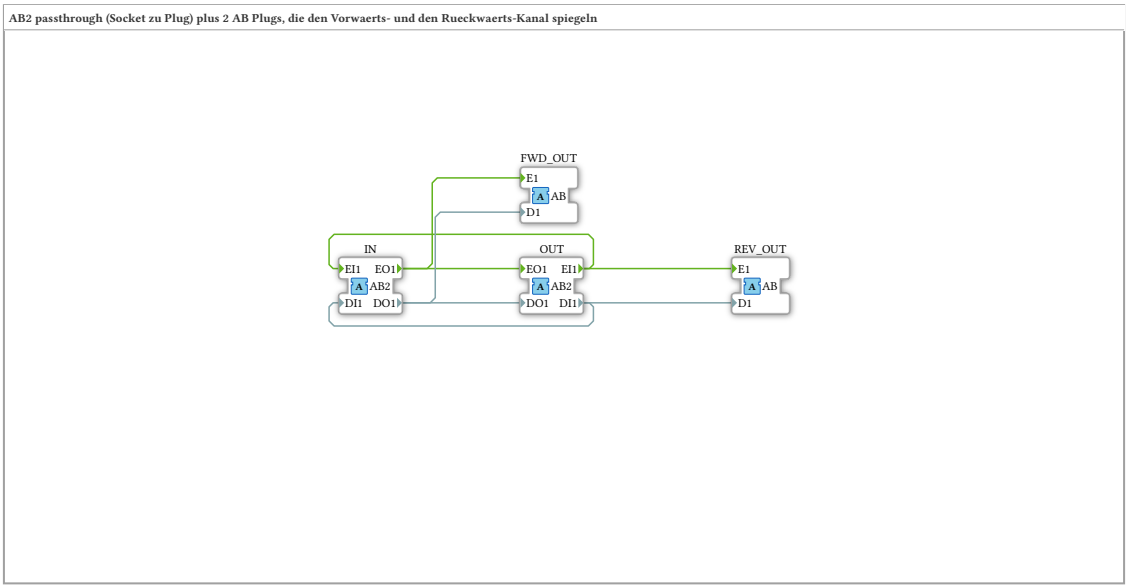
Fazit

Der TimeTicker ist ein nützlicher, zusammengesetzter Funktionsblock, der die Erzeugung periodischer Ereignisse mit einer erweiterten Schnittstelle für Adapter-Kommunikation kombiniert. Er bietet eine saubere Trennung zwischen Zeitsteuerung und Anwendung, indem er über den Adapter eine flexible Ein- und Auskopplung von Steuersignalen und Zeitdaten ermöglicht. Durch die Verwendung des bewährten `E_CYCLE`-Bausteins ist die Zuverlässigkeit gegeben, und die Bereitstellung von `Q`, `PT` und `ET` erlaubt eine umfassende Überwachung der zeitlichen Abläufe. Der Baustein eignet sich daher besonders für den Einsatz in komplexen automatisierten Systemen, die eine präzise und rückgekoppelte Zeitsteuerung benötigen.

bidirectional

- AE2_SPLIT_MERGE
- AE2_SPLIT_MERGE_FORWARD
- AE_AX_AX_SPLIT
- AE_AX_T_FF
- ASR_AX_AX_SPLIT
- ASR_AX_SR_2
- ASRT_AX_AX_SPLIT
- ASRT_AX_T_FF_SR_2

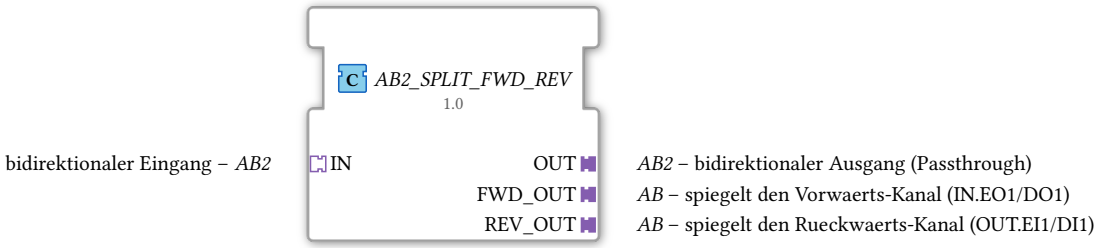
AB2_SPLIT_FWD_REV



• – ○ • – ○ • – ○ •

Einleitung

`AB2_SPLIT_FWD_REV` ist ein Composite-Funktionsblock, der einen bidirektionalen `AB2`-Adapter vom Socket `IN` unverändert zum Plug `OUT` durchreicht (Passthrough) und zusätzlich den Vorwärts- und den Rückwärts-Kanal auf zwei getrennte, unidirektionale `AB`-Plugs spiegelt. So können beide Richtungen einer `AB2`-Verbindung an weiteren Stellen mitgelesen werden, ohne die Verbindung selbst zu verändern.



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden – Ereignisse kommen über die Adapter-Sockets/-Plugs

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

- **IN** (Socket): Bidirektionaler Adapter vom Typ `adapter::types::bidirectional::AB2` (bidirektionaler Eingang)
- **OUT** (Plug): Bidirektionaler Adapter vom Typ `adapter::types::bidirectional::AB2` (bidirektionaler Ausgang, Passthrough)
- **FWD_OUT** (Plug): Unidirektionaler Adapter vom Typ `adapter::types::unidirectional::AB`, spiegelt den Vorwärts-Kanal (`IN.EO1/DO1`)

- **REV_OUT** (Plug): Unidirektionaler Adapter vom Typ `adapter::types::unidirectional::AB`, spiegelt den Rückwärts-Kanal (`OUT.EI1`/`DI1`)

Funktionsweise

1. **Vorwärts (Indikation)**: `IN.EO1`/`IN.DO1` wird unverändert an `OUT.EO1`/`OUT.DO1` weitergeleitet und zusätzlich an `FWD_OUT.E1`/`FWD_OUT.D1` gespiegelt.
2. **Rückwärts (Request)**: `OUT.EI1`/`OUT.DI1` wird unverändert an `IN.EI1`/`IN.DI1` weitergeleitet und zusätzlich an `REV_OUT.E1`/`REV_OUT.D1` gespiegelt.

Technische Besonderheiten

- Reine Ereignis-/Datenverbindungen (`FBNetwork`), keine eigene Logik und kein interner Zustand
- Jeder Ereignis-Ausgang des Passthrough speist zwei Ziele (Fan-out), die Daten laufen parallel zu den Ereignissen
- Die beiden Spiegel-Plugs sind unidirektional (`AB`) und liefern daher nur Lesezugriff auf den jeweiligen Kanal

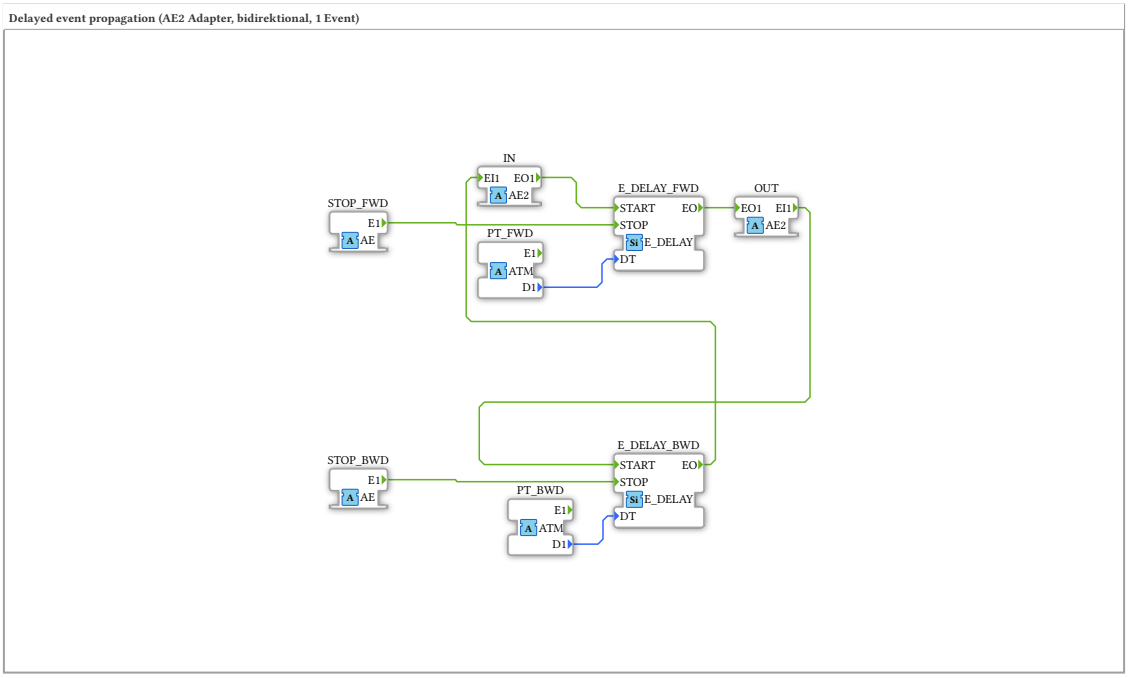
Anwendungsszenarien

- Mitlesen (Monitoring/Diagnose) beider Richtungen einer bidirektionalen `AB2`-Verbindung
- Aufteilen einer `AB2`-Verbindung in einen separat auswertbaren Vorwärts- und Rückwärtsstrom, ohne die Hauptverbindung zu unterbrechen

Fazit

`AB2_SPLIT_FWD_REV` reicht eine bidirektionale `AB2`-Verbindung transparent durch und stellt Vorwärts- und Rückwärts-Kanal zusätzlich als getrennte `AB`-Adapter zur Verfügung.

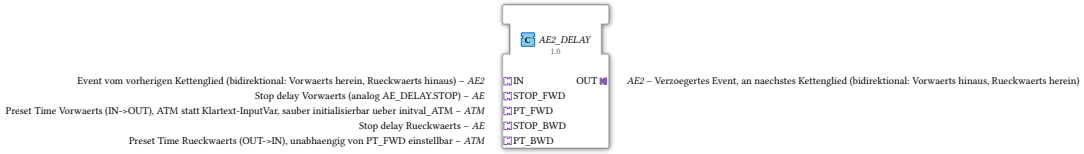
AE2_DELAY



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock `AE2_DELAY` verzögert Ereignisse eines bidirektionalen `AE2`-Adapters. Weil AE2 in beide Richtungen Ereignisse führt (`EO1` vorwärts, `EI1` rückwärts), besitzt der Baustein zwei voneinander unabhängige Verzögerungsstrecken: eine für die Vorwärts-Richtung (`IN` ⟶ `OUT`) mit der Zeit `PT_FWD` und eine für die Rückwärts-Richtung (`OUT` ⟶ `IN`) mit der Zeit `PT_BWD`. Jede Richtung kann über einen eigenen `STOP`-Socket abgebrochen werden.



Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **IN** (Socket): Ereignis vom vorherigen Kettenglied (vorwärts herein, rückwärts hinaus), Typ `adapter::types::bidirectional::AE2`
- **OUT** (Plug): verzögertes Ereignis an das nächste Kettenglied (vorwärts hinaus, rückwärts herein), Typ `adapter::types::bidirectional::AE2`
- **STOP_FWD** (Socket): bricht die laufende Vorwärts-Verzögerung ab, Typ `adapter::types::unidirectional::AE`
- **PT_FWD** (Socket): Verzögerungszeit (Preset Time) Vorwärts, Typ `adapter::types::unidirectional::ATM`
- **STOP_BWD** (Socket): bricht die laufende Rückwärts-Verzögerung ab, Typ `adapter::types::unidirectional::AE`
- **PT_BWD** (Socket): Verzögerungszeit Rückwärts, unabhängig von `PT_FWD` einstellbar, Typ `adapter::types::unidirectional::ATM`

Funktionsweise

Das interne Netzwerk besteht aus zwei Instanzen von `E_DELAY`:

- **Vorwärts** (`E_DELAY_FWD`): `IN.EO1` ⟶ `E_DELAY_FWD.START`, `STOP_FWD.E1` ⟶ `E_DELAY_FWD.STOP`, `PT_FWD.D1` ⟶ `E_DELAY_FWD.DT`. Nach Ablauf der Zeit wird `E_DELAY_FWD.EO` auf `OUT.EO1` ausgegeben.
- **Rückwärts** (`E_DELAY_BWD`): `OUT.EI1` ⟶ `E_DELAY_BWD.START`, `STOP_BWD.E1` ⟶ `E_DELAY_BWD.STOP`, `PT_BWD.D1` ⟶ `E_DELAY_BWD.DT`. Nach Ablauf der Zeit wird `E_DELAY_BWD.EO` auf `IN.EI1` ausgegeben.

Die Daten des Adapters werden nicht verarbeitet; es werden ausschließlich die Ereignisse `EO1`/`EI1` weitergereicht.

Technische Besonderheiten

- Zwei unabhängige Verzögerungsstrecken (je Richtung eine `E_DELAY`-Instanz) mit eigener Zeit und eigenem `STOP`
- Die Zeiten werden über `ATM`-Sockets statt über Klartext-Eingangsvariablen geliefert und lassen sich dadurch sauber über `initval_ATM` (mit `INIT`/`INITO`) initialisieren
- Der Socket für die Zeit heißt `PT` (Preset Time) und nicht `DT`, weil `DT` in IEC 61131-3 ein reserviertes Schlüsselwort (`DATE_AND_TIME`) ist
- Der Baustein selbst ist zustandslos; der Zustand (laufender Timer) liegt in den beiden `E_DELAY`-Instanzen

Anwendungsszenarien

- Zeitversetztes Weiterreichen von Ereignissen in einer AE2-Kette, z. B. gestaffeltes Einschalten und Ausschalten mit unterschiedlichen Zeiten
- Entkopplung von Anforderung und Rückmeldung in bidirektionalen Ereignisketten

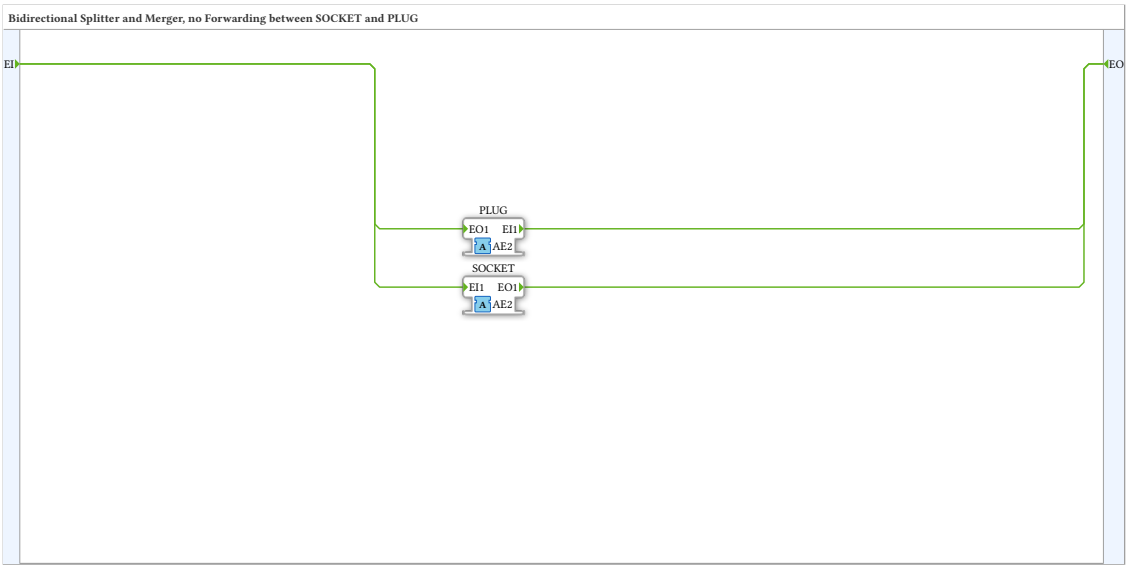
⚖️ Vergleich mit ähnlichen Bausteinen

Für Ketten mit getrennt verzögertem Set- und Reset-Zweig gibt es `ASR2_DELAY`, dessen Variante mit fest vorbelegten Zeiten `ASR2_DELAY_SUB` ist. `AE2_DELAY` verzögert dagegen genau ein Ereignis je Richtung.

Fazit

`AE2_DELAY` ist ein reiner Verdrahtungsbaustein aus zwei `E_DELAY`-Instanzen, der Ereignisse eines AE2-Adapters in beiden Richtungen mit je einer eigenen, per `ATM`-Socket einstellbaren Zeit verzögert und je Richtung stoppbar ist.

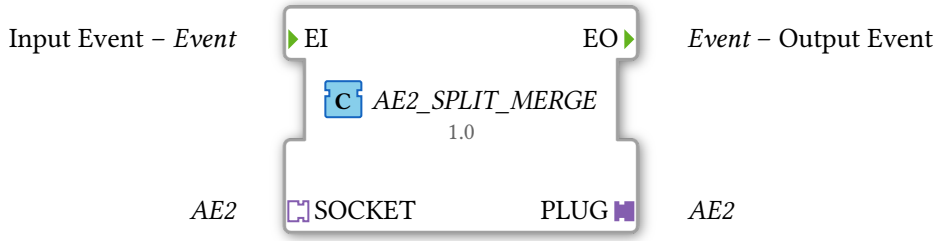
AE2_SPLIT_MERGE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AE2_SPLIT_MERGE ist ein bidirektionaler Splitter- und Merger-Funktionsblock, der Ereignisse in beide Richtungen verarbeiten kann. Der Baustein ermöglicht die Verteilung und Zusammenführung von Ereignissen über Adapterschnittstellen, wobei keine direkte Weiterleitung zwischen SOCKET und PLUG erfolgt.



Schnittstellenstruktur

Ereignis-Eingänge

- **EI**: Haupt-Ereigniseingang für eingehende Ereignisse

Ereignis-Ausgänge

- **EO**: Haupt-Ereignisausgang für ausgehende Ereignisse

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

- **PLUG**: Adapter-Schnittstelle vom Typ „adapter::types::bidirectional::AE2“
- **SOCKET**: Adapter-Schnittstelle vom Typ „adapter::types::bidirectional::AE2“

Funktionsweise

Der Funktionsblock arbeitet nach folgenden Prinzipien:

1. **Ereignis-Verteilung**: Ereignisse am Eingang EI werden gleichzeitig an beide Adapterschnittstellen (PLUG und SOCKET) weitergeleitet
2. **Ereignis-Zusammenführung**: Ereignisse von beiden Adapterschnittstellen (PLUG.EI1 und SOCKET.EO1) werden am Ausgang EO zusammengeführt
3. **Bidirektionale Isolation**: Es findet keine direkte Kommunikation zwischen den Adapterschnittstellen PLUG und SOCKET statt

Technische Besonderheiten

- Reine Ereignisverarbeitung ohne Datentransfer
- Symmetrische Behandlung beider Adapterrichtungen
- Isolierte Signalwege zwischen den Adapterschnittstellen
- Echtzeitfähige Ereignisverteilung

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand und arbeitet stateless. Alle eingehenden Ereignisse werden sofort verarbeitet und weitergeleitet.

Anwendungsszenarien

- Verteilung von Steuerereignissen an mehrere Subsysteme
- Zusammenführung von Statusereignissen aus verschiedenen Quellen
- Bidirektionale Kommunikation in verteilten Steuerungssystemen
- Event-basierte Architekturen mit Adapter-Pattern

⚖️ Vergleich mit ähnlichen Bausteinen

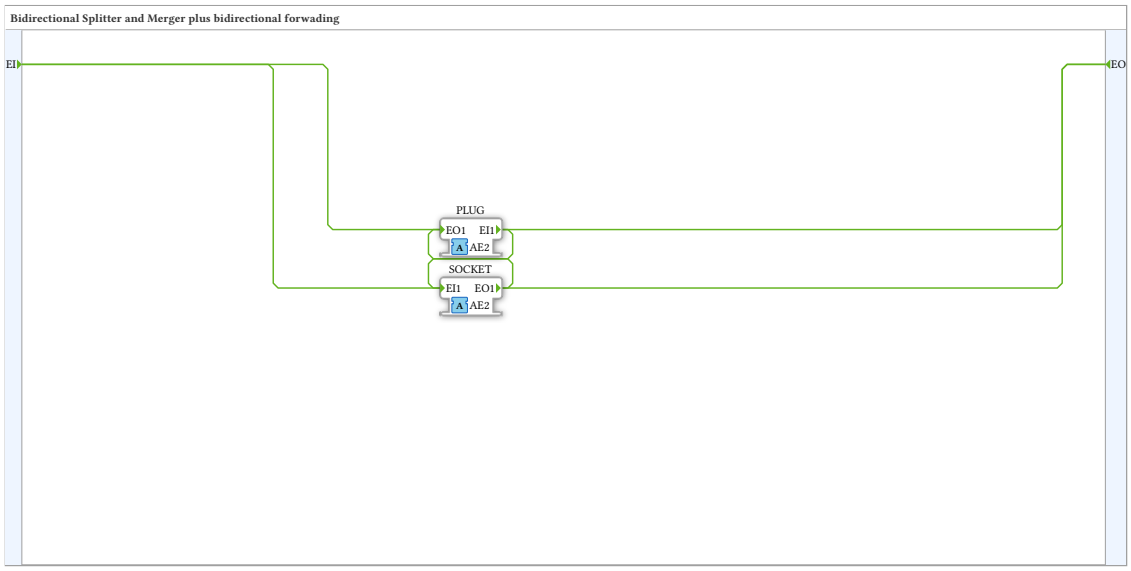
Im Vergleich zu einfachen Splitter- oder Merger-Blöcken bietet AE2_SPLIT_MERGE:

- Bidirektionale Funktionalität in einem einzigen Block
- Adapter-basierte Schnittstellen für standardisierte Kommunikation
- Isolierte Signalwege zwischen verschiedenen Adapterrichtungen

Fazit

Der AE2_SPLIT_MERGE ist ein spezialisierter Funktionsblock für komplexe Ereignisverteilungs- und Zusammenführungsszenarien in bidirektionalen Kommunikationsarchitekturen. Seine Adapter-basierte Struktur ermöglicht flexible und standardisierte Integration in größere Steuerungssysteme.

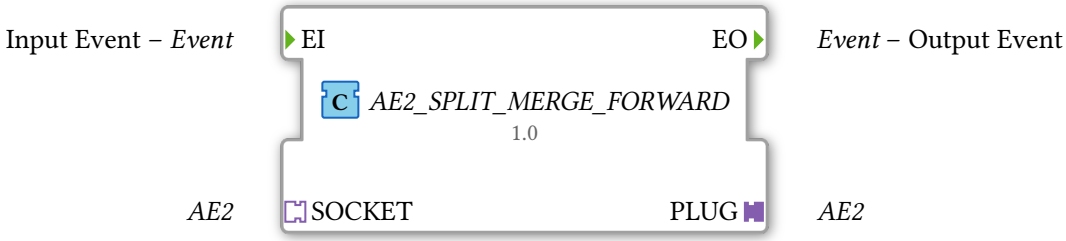
AE2_SPLIT_MERGE_FORWARD



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AE2_SPLIT_MERGE_FORWARD ist ein bidirektionaler Splitter und Merger mit bidirektionaler Weiterleitung. Er dient als universeller Verteiler und Sammler für Ereignisse in 4diac-Systemen und ermöglicht die flexible Verteilung von Ereignissen über verschiedene Schnittstellen.



Schnittstellenstruktur

Ereignis-Eingänge

- **EI**: Eingangsereignis (Input Event)

Ereignis-Ausgänge

- **EO**: Ausgangsereignis (Output Event)

Adapter

- **PLUG**: Bidirektionaler Adapter vom Typ AE2
- **SOCKET**: Bidirektionaler Adapter vom Typ AE2

Funktionsweise

Der Funktionsblock arbeitet nach folgenden Prinzipien:

1. **Ereignisverteilung**: Jedes am EI-Eingang eingehende Ereignis wird gleichzeitig an beide Adapter (PLUG und SOCKET) weitergeleitet.
2. **Ereigniszusammenführung**: Ereignisse, die über einen der beiden Adapter (PLUG oder SOCKET) eingehen, werden am EO-Ausgang ausgegeben.
3. **Bidirektionale Weiterleitung**: Zusätzlich werden Ereignisse zwischen den beiden Adaptern bidirektional weitergeleitet, sodass PLUG- und SOCKET-Ereignisse untereinander ausgetauscht werden können.

Technische Besonderheiten

- Verwendet den bidirektionalen AE2-Adapter-Typ
- Ermöglicht parallele Ereignisverteilung
- Bietet vollständige bidirektionale Kommunikation zwischen allen Schnittstellen
- Keine Datenverbindungen - rein ereignisbasiert

Zustandsübersicht

Der Funktionsblock verfügt über keine internen Zustände und arbeitet rein kombinatorisch. Alle Ereignisverarbeitungen erfolgen sofort bei Eintreffen eines Ereignisses.

Anwendungsszenarien

- Verteiler für Steuerereignisse in komplexen Automatisierungsnetzwerken
- Schnittstellenadapter zwischen verschiedenen Systemkomponenten
- Ereignis-Routing in modularen Steuerungsarchitekturen
- Bidirektionale Kommunikationsbrücken zwischen Funktionsblöcken

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen Splitter-Blöcken bietet AE2_SPLIT_MERGE_FORWARD zusätzlich:

- Bidirektionale Funktionalität
- Adapter-basierte Schnittstellen
- Ereignis-Zusammenführung
- Direkte Kommunikation zwischen Adaptern

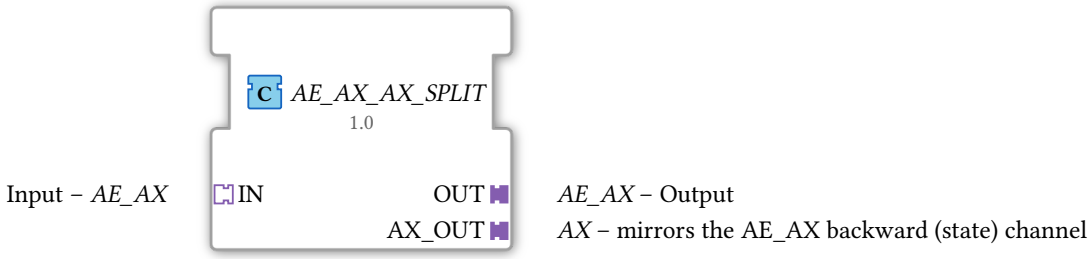
Fazit

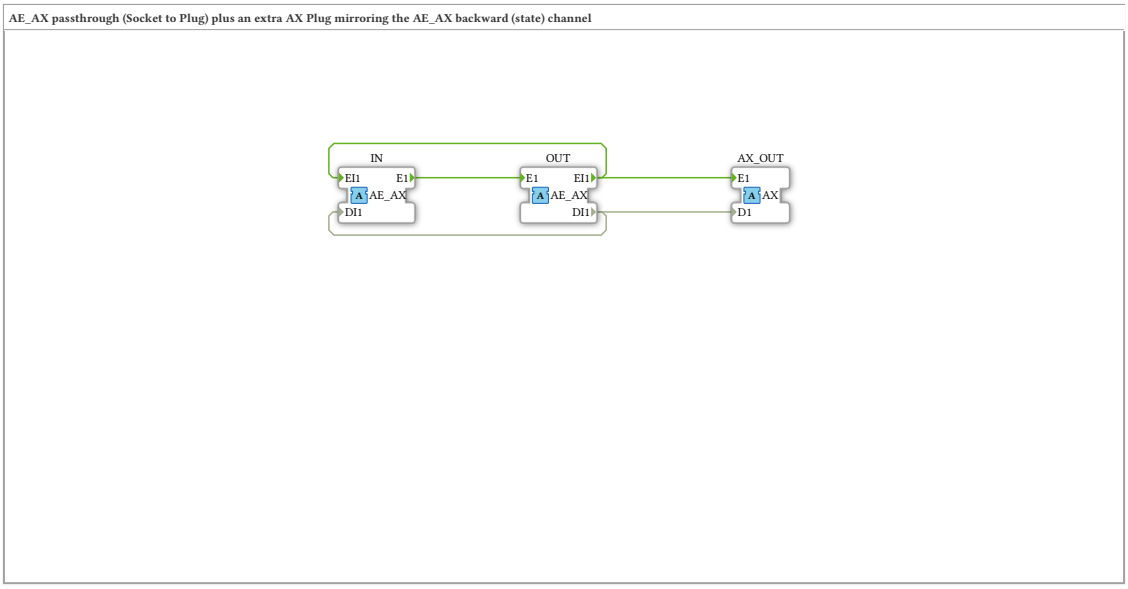
Der AE2_SPLIT_MERGE_FORWARD ist ein vielseitiger Funktionsblock für komplexe Ereignisverteilungs- und Sammelaufgaben in 4diac-Systemen. Seine bidirektionale Natur und die Adapter-Schnittstellen machen ihn besonders geeignet für modulare und erweiterbare Automatisierungslösungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AE_AX_AX_SPLIT





• – ○ • – ○ • – ○ •

Einleitung

Der AE_AX_AX_SPLIT ist ein Composite-Funktionsblock, der ein eingehendes AE_AX-Ereignis unverändert von seinem Socket `IN` an den Plug `OUT` durchreicht und zusätzlich den auf dem Rückkanal von `OUT` gemeldeten Zustand (Event + Bool) über einen dritten, unidirektionalen `AX_OUT`-Plug nach außen spiegelt. Damit lässt sich der Zustand einer nachgeschalteten Kette parallel an einer zweiten Stelle abgreifen, ohne die eigentliche AE_AX-Verbindung zu unterbrechen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden – Ereignisse kommen über die Adapter-Sockets/-Plugs

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

- **IN**: Bidirektionaler Adapter-Socket vom Typ `adapter::types::bidirectional::AE_AX` (Eingang)
- **OUT**: Bidirektionaler Adapter-Plug vom Typ `adapter::types::bidirectional::AE_AX` (Ausgang)
- **AX_OUT**: Unidirektionaler Adapter-Plug vom Typ `adapter::types::unidirectional::AX`, spiegelt den Rückkanal (Zustand) von AE_AX

Funktionsweise

1. Jedes an `IN.E1` eintreffende Ereignis wird unverändert an `OUT.E1` weitergeleitet – die Vorwärtsrichtung wird 1:1 durchgeschleift.
2. Das Rückkanal-Ereignis `OUT.EI1` (samt zugehörigem Datum `OUT.DI1`) wird sowohl an `IN.EI1`/`IN.DI1` zurückgemeldet als auch gleichzeitig an `AX_OUT.E1`/`AX_OUT.D1` weitergegeben.
3. Dadurch ist der über `OUT` gemeldete Zustand an zwei Stellen sichtbar: am ursprünglichen Socket `IN` (wie es ein reiner Passthrough auch täte) und zusätzlich isoliert am `AX_OUT`-Plug.

Technische Besonderheiten

- Reine Ereignis-/Datenverbindungen (`FBNetwork`), keine eigene Logik oder Zustandsverwaltung

- Vorwärtsrichtung (Socket ⟶ Plug) ist ein einfacher 1:1-Passthrough
- Rückkanal wird verdoppelt statt aufgeteilt: beide Ziele (`IN` und `AX_OUT`) erhalten dieselbe Information

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand und arbeitet stateless. Jedes eingehende Ereignis wird sofort weitergeleitet bzw. gespiegelt.

Anwendungsszenarien

- Zusätzliches, isoliertes Abgreifen des AE_AX-Rückkanals (z. B. für eine separate Anzeige oder Protokollierung) ohne die eigentliche Verbindung zwischen `IN` und `OUT` zu verändern
- Aufbau von Diagnose-/Monitoring-Pfaden in AE_AX-basierten Steuerungsnetzwerken

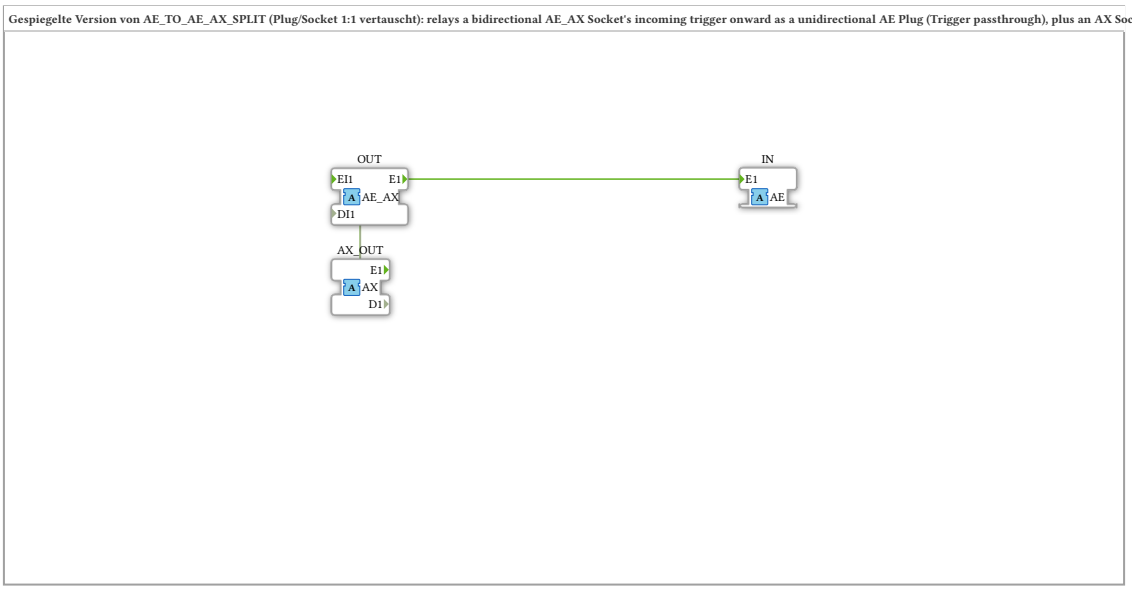
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu AE2_SPLIT_MERGE, das Vorwärts- und Rückwärtsrichtung zwischen zwei getrennten, isolierten Adaptern verteilt bzw. zusammenführt, ist der AE_AX_AX_SPLIT ein reiner Passthrough zwischen `IN` und `OUT` mit einem zusätzlichen Mirror-Ausgang für den Rückkanal – die ASR_AX- und ASRT_AX-Varianten ASR_AX_AX_SPLIT und ASRT_AX_AX_SPLIT folgen demselben Muster mit zwei bzw. drei Vorwärts-Ereignissen.

Fazit

Der AE_AX_AX_SPLIT ermöglicht es, den AX-Rückkanal eines AE_AX-Adapterpaares zusätzlich an einer zweiten Stelle im Netzwerk verfügbar zu machen, ohne die eigentliche Signalkette zu unterbrechen oder zu verändern.

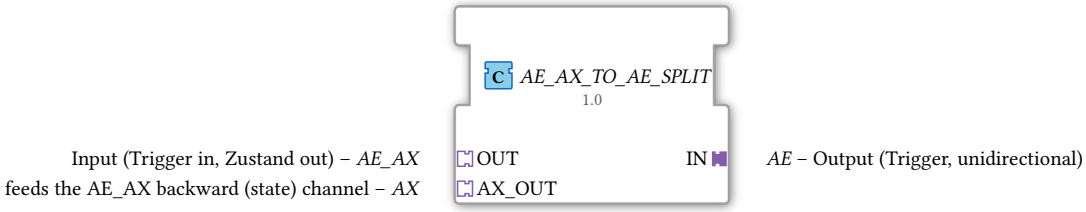
AE_AX_TO_AE_SPLIT



• – ○ • – ○ • – ○ •

Einleitung

`AE_AX_TO_AE_SPLIT` ist die gespiegelte Version von `AE_TO_AE_AX_SPLIT` (Plug und Socket 1:1 vertauscht). Der Baustein leitet das eingehende Trigger-Ereignis eines bidirektionalen `AE_AX`-Sockets als unidirektionalen `AE`-Plug weiter (Trigger-Durchreichung) und speist über einen zusätzlichen `AX`-Socket einen extern gelieferten Wert in den Rückkanal (Zustand) des `AE_AX`-Sockets ein.



Schnittstellenstruktur

Ereignis-Eingänge / -Ausgänge

Keine eigenen Ereignis- oder Datenanschlüsse; alle Ereignisse und Daten laufen über die Adapter.

Adapter

Richtung	Name	Adaptertyp	Kommentar
Plug (Ausgang)	`IN`	`adapter::types::unidirectional::AE`	Ausgang (Trigger, unidirektional).
Socket (Eingang)	`OUT`	`adapter::types::bidirectional::AE_AX`	Eingang (Trigger rein, Zustand raus).
Socket (Eingang)	`AX_OUT`	`adapter::types::unidirectional::AX`	Speist den Rückkanal (Zustand) des `AE_AX` ein.

Funktionsweise

Der Baustein ist ein Composite-FB ohne eigene Funktionsbausteine, er besteht nur aus Adapter-Verbindungen:

- **Ereignisverbindungen:** `OUT.E1` ⟶ `IN.E1` (das am `AE_AX`-Socket eingehende Ereignis wird unverändert am `AE`-Plug ausgegeben); `AX_OUT.E1` ⟶ `OUT.EI1` (das am `AX`-Socket empfangene Ereignis wird als Rückkanal-Ereignis über den `AE_AX`-Socket ausgegeben).
- **Datenverbindung:** `AX_OUT.D1` ⟶ `OUT.DI1` (der am `AX`-Socket empfangene Wert wird als Rückkanal-Datum `DI1` des `AE_AX`-Sockets ausgegeben).

Technische Besonderheiten

- Reine Verdrahtung, keine Logik und keine Verzögerung.
- Die Namen `IN`/`OUT` sind aus Sicht der Quelle `AE_TO_AE_AX_SPLIT` übernommen und deshalb vertauscht: `OUT` ist hier ein Socket, `IN` ein Plug.
- Die Richtung der Trigger bleibt unidirektional (`AE`), der Rückkanal wird vollständig vom `AX`-Socket gespeist.

Anwendungsszenarien

- Einen bidirektionalen `AE_AX`-Anschluss in einen reinen Trigger-Pfad (`AE`) und einen separat gespeisten Zustandspfad (`AX`) aufspalten.

Zusammenfassung

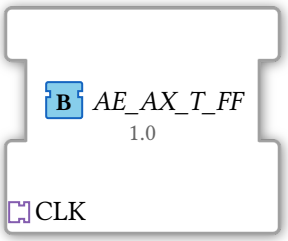
`AE_AX_TO_AE_SPLIT` trennt einen `AE_AX`-Socket in einen weitergereichten `AE`-Trigger und einen über `AX` gespeisten Rückkanal.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

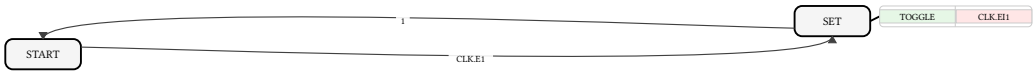
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AE_AX_T_FF

Clock (in), Zustand (out) – *AE_AX*



ECC: AE_AX_T_FF



• – ○ • – ○ • – ○ •

Einleitung

Der AE_AX_T_FF ist ein ereignisgesteuertes Toggle-Flipflop mit einem einzigen bidirektionalen `AE_AX`-Socket: Das Takt-Ereignis kommt am selben Adapter an, über den auch der neue Zustand zurückgemeldet wird – im Gegensatz zu AX_T_FF, der einen separaten CLK-EventInput und einen eigenen AX-Plug für den Zustand verwendet.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden – das Takt-Ereignis kommt über den Adapter-Socket `CLK`

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Dateneingänge vorhanden

Daten-Ausgänge

Keine Datenausgänge vorhanden

Adapter

- **CLK**: Bidirektionaler Adapter-Socket vom Typ `adapter::types::bidirectional::AE_AX` – Takt (Eingang), Zustand (Ausgang) über denselben Adapter

Funktionsweise

Der AE_AX_T_FF verfügt über zwei Betriebszustände:

- **START**: Initialzustand
- **SET**: Zustand nach dem Toggeln

Bei jedem an `CLK.E1` eintreffenden Ereignis wechselt der Funktionsblock von START nach SET und feuert dabei den Algorithmus `TOGGLE`, der `CLK.DI1 := NOT CLK.DI1` ausführt und damit über `CLK.EI1` den invertierten Zustand zurück über denselben Adapter meldet. Aus dem SET-Zustand kehrt der Baustein unbedingt (Bedingung `1`) sofort wieder nach START zurück, sodass jedes weitere `CLK.E1`-Ereignis erneut denselben TOGGLE-Übergang auslöst.

Technische Besonderheiten

- Bidirektionale Kommunikation über einen einzigen Adapter-Socket statt getrennter EventInput + AX-Plug
- Der ECC-Zustand SET ist nur ein Durchgangszustand für die Initialisierung des Ausgabe-Ereignisses; die eigentliche Toggle-Semantik steckt in der Algorithmuszuweisung `CLK.DI1 := NOT CLK.DI1`
- Direkter Ersatz für AX_T_FF in durchgängig adapterbasierten (AE_AX-)Anwendungen

Zustandsübersicht

```
START (Initialzustand)
    │
    └── CLK.E1 ─→ SET (CLK.DI1 := NOT CLK.DI1, meldet Zustand über CLK.EI1)
                      │
                      └── (unbedingt) ─→ START
```

Anwendungsszenarien

- Umschalten eines Zustands (z. B. Ein/Aus) über eine einzige bidirektionale Adapterverbindung, ohne separaten Zustandsausgang verdrahten zu müssen
- Ersatz für AX_T_FF in Netzwerken, die durchgängig AE_AX-Adapter verwenden

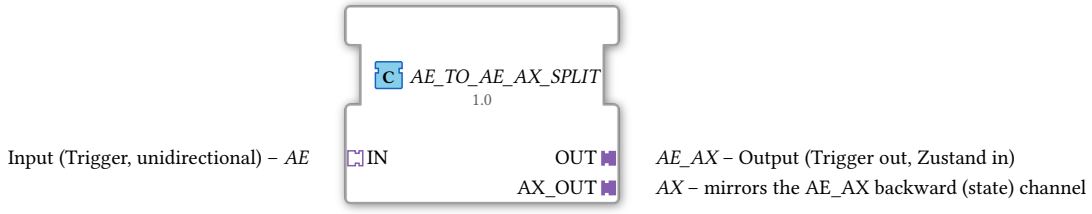
⚖️ Vergleich mit ähnlichen Bausteinen

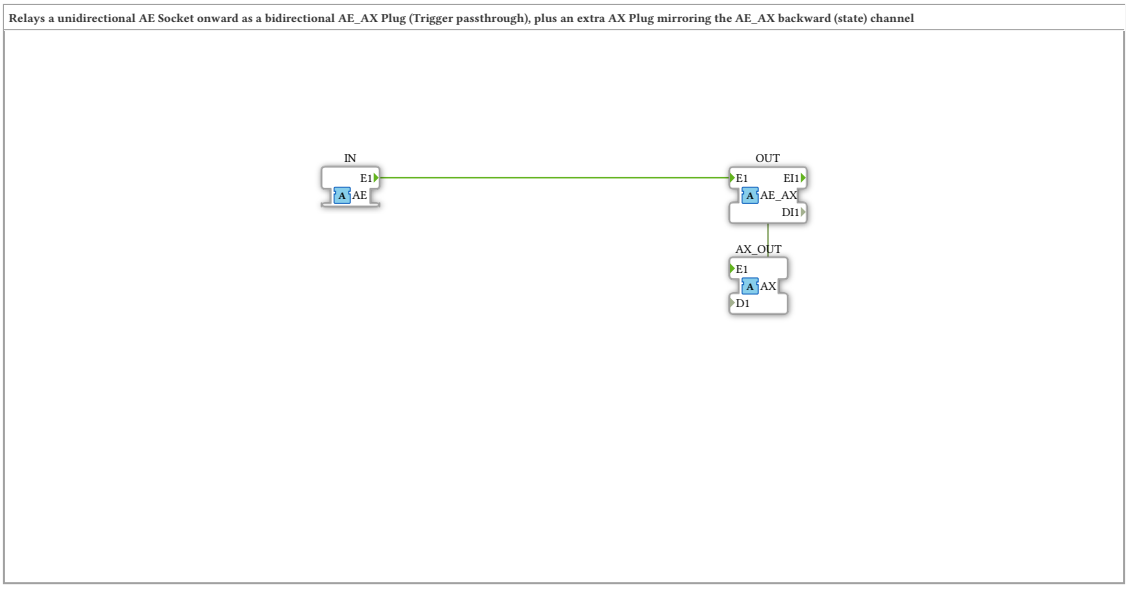
Im Vergleich zu AX_T_FF, das einen separaten `CLK`-EventInput und einen eigenen AX-Plug für den Zustand nutzt, bündelt AE_AX_T_FF beide Richtungen in einem einzigen AE_AX-Socket.

Fazit

Der AE_AX_T_FF überträgt das bewährte Toggle-Flipflop-Muster in die bidirektionale Adapterwelt: Takt-Eingang und Zustands-Rückmeldung teilen sich denselben AE_AX-Socket, was die Verdrahtung in adapterbasierten Netzwerken vereinfacht.

AE_TO_AE_AX_SPLIT





• – ○ • – ○ • – ○ •

Einleitung

Der AE_TO_AE_AX_SPLIT ist ein Composite-Funktionsblock, der ein eingehendes, **unidirektionales** AE-Ereignis an seinem Socket `IN` in ein **bidirektionales** AE_AX-Signal am Plug `OUT` umwandelt, und zusätzlich den auf dem Rückkanal von `OUT` gemeldeten Zustand (Event + Bool) über einen dritten, unidirektionalen `AX_OUT`-Plug nach außen spiegelt. Anders als beim reinen Passthrough `AE_AX_AX_SPLIT` hat `IN` hier keinen eigenen Rückkanal - der Rückkanal kann also nicht an `IN` zurückgemeldet werden und läuft ausschließlich über `AX_OUT` nach außen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden – Ereignisse kommen über die Adapter-Sockets/-Plugs

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

- **IN**: Unidirektionaler Adapter-Socket vom Typ `adapter::types::unidirectional::AE` (Eingang, kein Rückkanal)
- **OUT**: Bidirektionaler Adapter-Plug vom Typ `adapter::types::bidirectional::AE_AX` (Ausgang, mit Rückkanal)
- **AX_OUT**: Unidirektionaler Adapter-Plug vom Typ `adapter::types::unidirectional::AX`, spiegelt den Rückkanal (Zustand) von AE_AX nach außen

Funktionsweise

1. Jedes an `IN.E1` eintreffende Ereignis wird unverändert an `OUT.E1` weitergeleitet.
2. Das Rückkanal-Ereignis `OUT.EI1` (samt zugehörigem Datum `OUT.DI1`), das die nachgeschaltete Gegenstelle über `OUT` sendet, wird ausschließlich an `AX_OUT.E1`/`AX_OUT.D1` weitergegeben - eine Rückmeldung an `IN` ist technisch unmöglich, da `IN` als unidirektionaler AE-Adapter keinen Rückkanal besitzt.
3. `AX_OUT` ist damit der einzige Ort, an dem der von `OUT` gemeldete Zustand sichtbar wird.

Technische Besonderheiten

- Reine Ereignis-/Datenverbindungen (`FBNetwork`), keine eigene Logik oder Zustandsverwaltung
- Vorwärtsrichtung (Socket → Plug) ist ein einfacher 1:1-Passthrough
- Wandelt einen unidirektionalen Adapter (`IN`) in einen bidirektionalen (`OUT`) um, ohne selbst einen Rückkanal am Eingang zu benötigen
- Der Rückkanal wird nicht verdoppelt (wie bei AE_AX_AX_SPLIT), sondern einzig über `AX_OUT` exponiert

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand und arbeitet stateless. Jedes eingehende Ereignis wird sofort weitergeleitet bzw. gespiegelt.

Anwendungsszenarien

- Anschluss eines bestehenden, rein unidirektionalen AE-Signalgebers an eine Kette, die einen bidirektionalen AE_AX-Adapter erwartet
- Sichtbarmachen des Zustands der nachgeschalteten AE_AX-Gegenstelle über `AX_OUT`, obwohl die ursprüngliche Quelle (`IN`) selbst keinen Rückkanal unterstützt

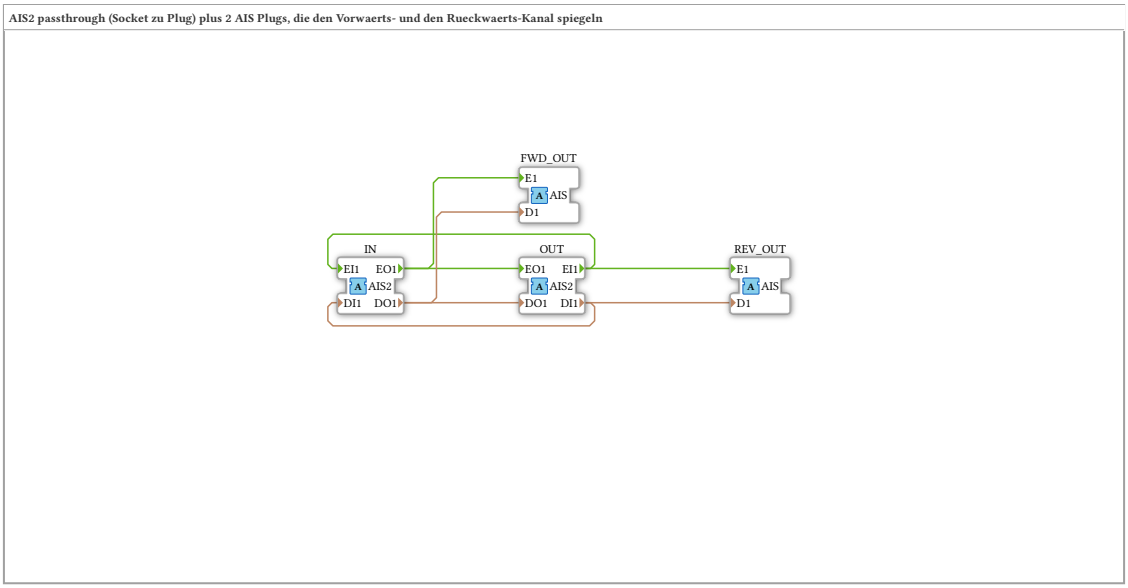
⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu `AE_AX_AX_SPLIT`, das zwischen zwei bereits bidirektionalen Adaptern vermittelt und den Rückkanal zusätzlich an `IN` zurückspiegelt, wandelt AE_TO_AE_AX_SPLIT einen unidirektionalen Eingang in einen bidirektionalen Ausgang um - der Rückkanal kann daher nur über `AX_OUT` beobachtet werden. Die Set/Reset- bzw. Set/Reset/Toggle-Varianten `ASR_TO_ASR_AX_SPLIT` und `ASRT_TO_ASRT_AX_SPLIT` folgen demselben Muster mit zwei bzw. drei Vorwärts-Ereignissen statt einem einzelnen.

Fazit

Der AE_TO_AE_AX_SPLIT ermöglicht den Anschluss eines rein unidirektionalen AE-Signalgebers an eine bidirektionale AE_AX-Kette und macht deren Rückkanal isoliert über `AX_OUT` verfügbar, ohne dass am ursprünglichen Eingang selbst eine Rückmeldung möglich sein muss.

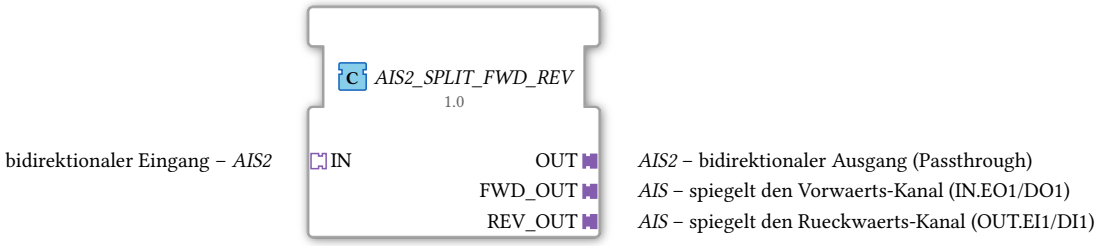
AIS2_SPLIT_FWD_REV



• – ○ • – ○ • – ○ •

Einleitung

`AIS2_SPLIT_FWD_REV` ist ein Composite-Funktionsblock, der einen bidirektionalen `AIS2`-Adapter vom Socket `IN` unverändert zum Plug `OUT` durchreicht (Passthrough) und zusätzlich den Vorwärts- und den Rückwärts-Kanal auf zwei getrennte, unidirektionale `AIS`-Plugs spiegelt. So können beide Richtungen einer `AIS2`-Verbindung an weiteren Stellen mitgelesen werden, ohne die Verbindung selbst zu verändern.



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden – Ereignisse kommen über die Adapter-Sockets/-Plugs

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

- **IN** (Socket): Bidirektionaler Adapter vom Typ `adapter::types::bidirectional::AIS2` (bidirektionaler Eingang)
- **OUT** (Plug): Bidirektionaler Adapter vom Typ `adapter::types::bidirectional::AIS2` (bidirektionaler Ausgang, Passthrough)
- **FWD_OUT** (Plug): Unidirektionaler Adapter vom Typ `adapter::types::unidirectional::AIS`, spiegelt den Vorwärts-Kanal (`IN.EO1/DO1`)
- **REV_OUT** (Plug): Unidirektionaler Adapter vom Typ `adapter::types::unidirectional::AIS`, spiegelt den Rückwärts-Kanal (`OUT.EI1/DI1`)

Funktionsweise

1. **Vorwärts (Indikation)**: `IN.EO1/IN.DO1` wird unverändert an `OUT.EO1/OUT.DO1` weitergeleitet und zusätzlich an `FWD_OUT.E1/FWD_OUT.D1` gespiegelt.
2. **Rückwärts (Request)**: `OUT.EI1/OUT.DI1` wird unverändert an `IN.EI1/IN.DI1` weitergeleitet und zusätzlich an `REV_OUT.E1/REV_OUT.D1` gespiegelt.

Technische Besonderheiten

- Reine Ereignis-/Datenverbindungen (`FBNetwork`), keine eigene Logik und kein interner Zustand
- Jeder Ereignis-Ausgang des Passthrough speist zwei Ziele (Fan-out), die Daten laufen parallel zu den Ereignissen
- Die beiden Spiegel-Plugs sind unidirektional (`AIS`) und liefern daher nur Lesezugriff auf den jeweiligen Kanal

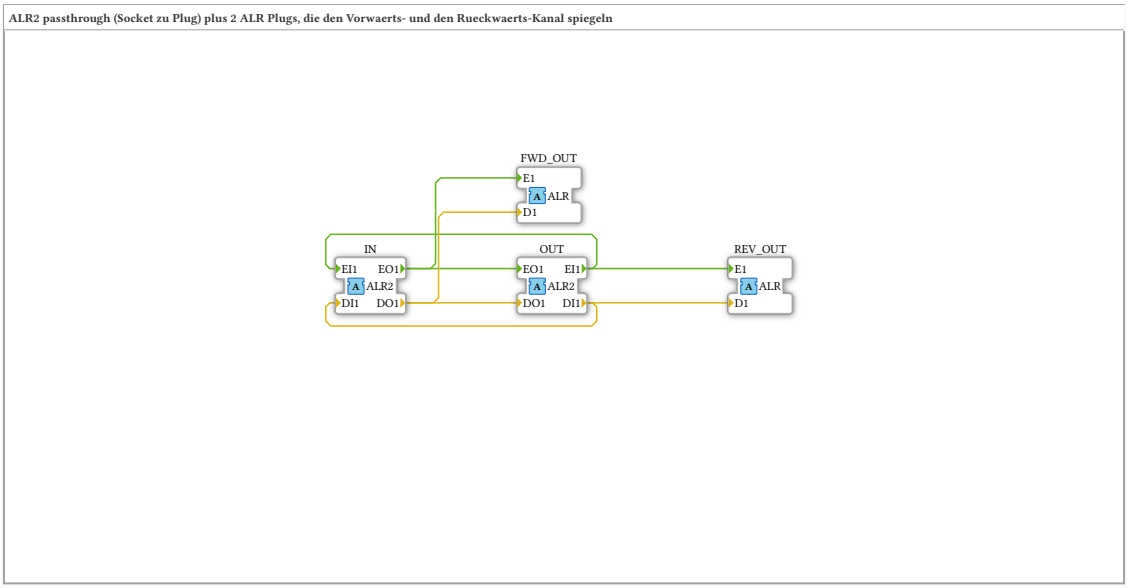
Anwendungsszenarien

- Mitlesen (Monitoring/Diagnose) beider Richtungen einer bidirektionalen `AIS2`-Verbindung
- Aufteilen einer `AIS2`-Verbindung in einen separat auswertbaren Vorwärts- und Rückwärtsstrom, ohne die Hauptverbindung zu unterbrechen

Fazit

`AIS2_SPLIT_FWD_REV` reicht eine bidirektionale `AIS2`-Verbindung transparent durch und stellt Vorwärts- und Rückwärts-Kanal zusätzlich als getrennte `AIS`-Adapter zur Verfügung.

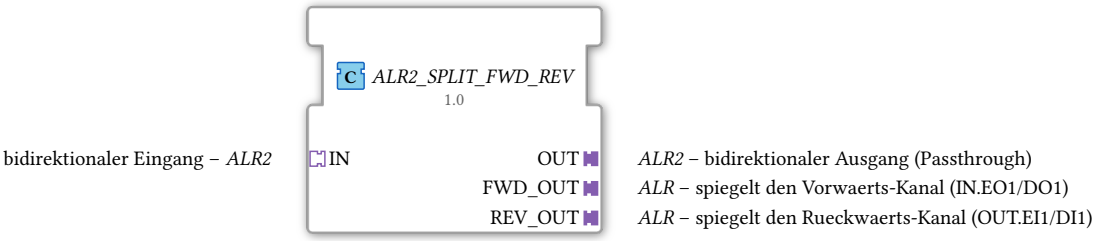
ALR2_SPLIT_FWD_REV



• – ○ • – ○ • – ○ •

Einleitung

`ALR2_SPLIT_FWD_REV` ist ein Composite-Funktionsblock, der einen bidirektionalen `ALR2`-Adapter vom Socket `IN` unverändert zum Plug `OUT` durchreicht (Passthrough) und zusätzlich den Vorwärts- und den Rückwärts-Kanal auf zwei getrennte, unidirektionale `ALR`-Plugs spiegelt. So können beide Richtungen einer `ALR2`-Verbindung an weiteren Stellen mitgelesen werden, ohne die Verbindung selbst zu verändern.



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden – Ereignisse kommen über die Adapter-Sockets/-Plugs

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

- **IN** (Socket): Bidirektionaler Adapter vom Typ `adapter::types::bidirectional::ALR2` (bidirektionaler Eingang)
- **OUT** (Plug): Bidirektionaler Adapter vom Typ `adapter::types::bidirectional::ALR2` (bidirektionaler Ausgang, Passthrough)
- **FWD_OUT** (Plug): Unidirektionaler Adapter vom Typ `adapter::types::unidirectional::ALR`, spiegelt den Vorwärts-Kanal (`IN.EO1/DO1`)

- **REV_OUT** (Plug): Unidirektionaler Adapter vom Typ `adapter::types::unidirectional::ALR`, spiegelt den Rückwärts-Kanal (`OUT.EI1/DI1`)

Funktionsweise

1. **Vorwärts (Indikation)**: `IN.EO1/IN.DO1` wird unverändert an `OUT.EO1/OUT.DO1` weitergeleitet und zusätzlich an `FWD_OUT.E1/FWD_OUT.D1` gespiegelt.
2. **Rückwärts (Request)**: `OUT.EI1/OUT.DI1` wird unverändert an `IN.EI1/IN.DI1` weitergeleitet und zusätzlich an `REV_OUT.E1/REV_OUT.D1` gespiegelt.

Technische Besonderheiten

- Reine Ereignis-/Datenverbindungen (`FBNetwork`), keine eigene Logik und kein interner Zustand
- Jeder Ereignis-Ausgang des Passthrough speist zwei Ziele (Fan-out), die Daten laufen parallel zu den Ereignissen
- Die beiden Spiegel-Plugs sind unidirektional (`ALR`) und liefern daher nur Lesezugriff auf den jeweiligen Kanal

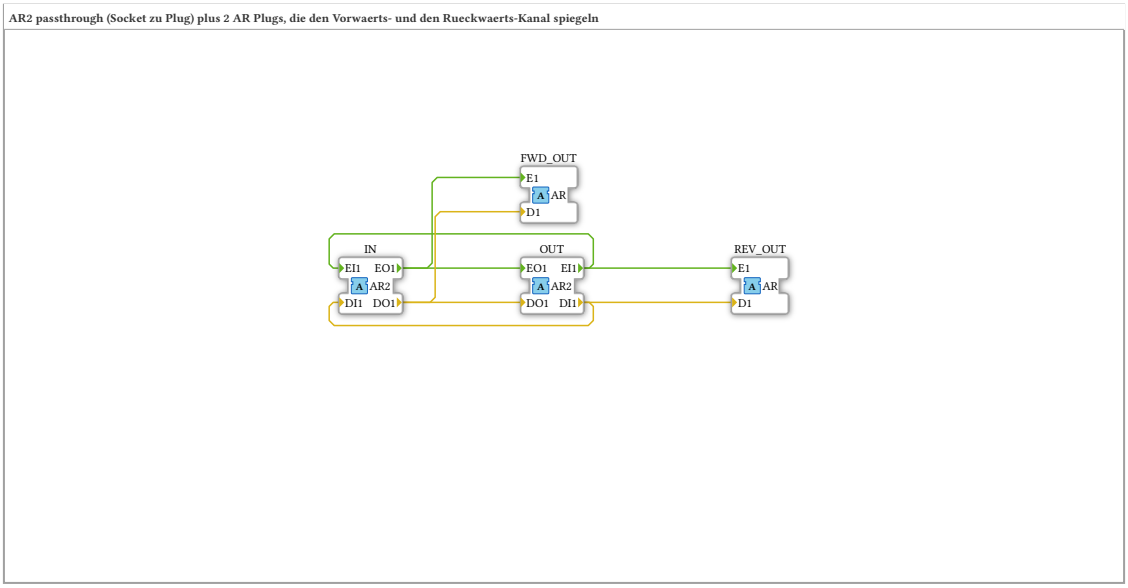
Anwendungsszenarien

- Mitlesen (Monitoring/Diagnose) beider Richtungen einer bidirektionalen `ALR2`-Verbindung
- Aufteilen einer `ALR2`-Verbindung in einen separat auswertbaren Vorwärts- und Rückwärtsstrom, ohne die Hauptverbindung zu unterbrechen

Fazit

`ALR2_SPLIT_FWD_REV` reicht eine bidirektionale `ALR2`-Verbindung transparent durch und stellt Vorwärts- und Rückwärts-Kanal zusätzlich als getrennte `ALR`-Adapter zur Verfügung.

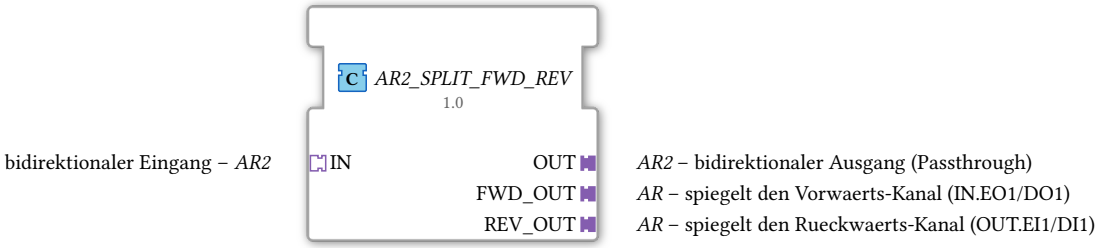
AR2_SPLIT_FWD_REV



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AR2_SPLIT_FWD_REV` führt einen bidirektionalen AR2-Adapter unverändert von `IN` (Socket) nach `OUT` (Plug) durch und spiegelt dabei zusätzlich beide Kanäle auf je einen unidirektionalen AR-Plug: den Vorwärts-Kanal (Indikation, `IN.EO1/DO1`) auf `FWD_OUT` und den Rückwärts-Kanal (Request, `OUT.EI1/DI1`) auf `REV_OUT`. Damit lassen sich beide Richtungen einer bidirektionalen AR2-Verbindung getrennt als einfache AR-Signale abgreifen.



Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **IN** (Socket): bidirektionaler Eingang, Typ `adapter::types::bidirectional::AR2`
- **OUT** (Plug): bidirektionaler Ausgang (Passthrough), Typ `adapter::types::bidirectional::AR2`
- **FWD_OUT** (Plug): spiegelt den Vorwärts-Kanal (`IN.EO1`/`DO1`), Typ `adapter::types::unidirectional::AR`
- **REV_OUT** (Plug): spiegelt den Rückwärts-Kanal (`OUT.EI1`/`DI1`), Typ `adapter::types::unidirectional::AR`

Funktionsweise

Der Baustein besteht nur aus Verdrahtung:

- **Vorwärts** (Indikation): `IN.EO1` ⟶ `OUT.EO1` und zusätzlich `IN.EO1` ⟶ `FWD_OUT.E1`; `IN.DO1` ⟶ `OUT.DO1` und zusätzlich `IN.DO1` ⟶ `FWD_OUT.D1`.
- **Rückwärts** (Request): `OUT.EI1` ⟶ `IN.EI1` und zusätzlich `OUT.EI1` ⟶ `REV_OUT.E1`; `OUT.DI1` ⟶ `IN.DI1` und zusätzlich `OUT.DI1` ⟶ `REV_OUT.D1`.

Beide Richtungen laufen also unverändert durch, und jede wird zusätzlich als einfaches unidirektionales AR-Signal ausgegeben.

Technische Besonderheiten

- Reine Verdrahtung ohne Logik oder Zustand
- Der Passthrough `IN` ⟶ `OUT` bleibt vollständig erhalten; `FWD_OUT` und `REV_OUT` sind reine Beobachtungs-Abzweige
- Die Quellen werden jeweils auf zwei Ziele verzweigt (Ereignis und Daten); das Verhalten entspricht dem eines Splits auf beide Zweige

Anwendungsszenarien

- Mitlesen der Vorwärts- und der Rückwärts-Richtung einer AR2-Verbindung (z. B. für Anzeige, Logging oder Diagnose), ohne die Verbindung selbst zu verändern
- Getrennte Weiterverarbeitung von Indikation und Request als einfache AR-Signale

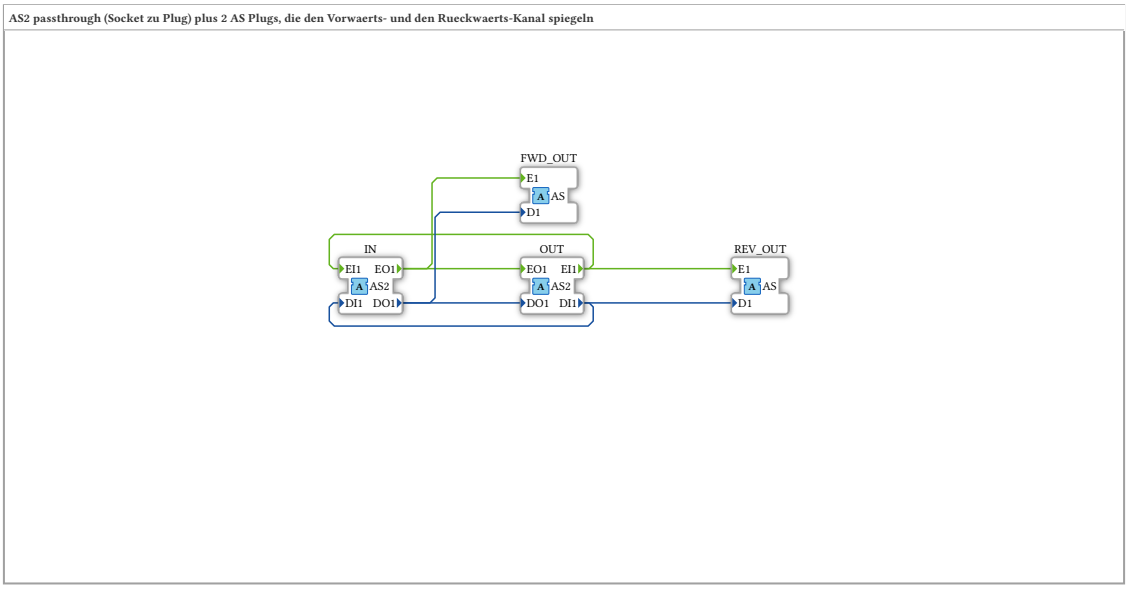
⚖️ Vergleich mit ähnlichen Bausteinen

Dieselbe Struktur existiert für weitere bidirektionale Adapter, z. B. `AR2_SPLIT_FWD_REV`, `AS2_SPLIT_FWD_REV`, `AX2_SPLIT_FWD_REV`, `AIS2_SPLIT_FWD_REV` und `ALR2_SPLIT_FWD_REV`; sie unterscheiden sich nur im Datentyp des Adapters.

Fazit

`AR2_SPLIT_FWD_REV` ist ein zustandsloser Passthrough für AR2, der Vorwärts- und Rückwärts-Kanal zusätzlich auf zwei unidirektionale AR-Plugs spiegelt.

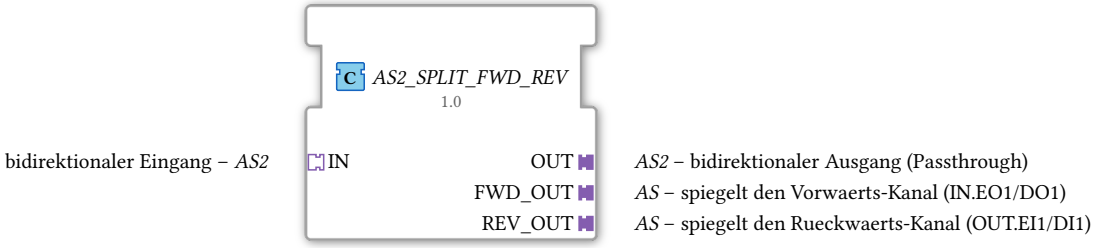
AS2_SPLIT_FWD_REV



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AS2_SPLIT_FWD_REV` führt einen bidirektionalen `AS2`-Adapter unverändert von `IN` (Socket) nach `OUT` (Plug) durch und spiegelt dabei zusätzlich beide Kanäle auf je einen unidirektionalen `AS`-Plug: den Vorwärts-Kanal (Indikation, `IN.EO1`/`DO1`) auf `FWD_OUT` und den Rückwärts-Kanal (Request, `OUT.EI1`/`DI1`) auf `REV_OUT`. Damit lassen sich beide Richtungen einer bidirektionalen AS2-Verbindung getrennt als einfache AS-Signale abgreifen.



Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **IN** (Socket): bidirektionaler Eingang, Typ `adapter::types::bidirectional::AS2`

- **OUT** (Plug): bidirektionaler Ausgang (Passthrough), Typ `adapter::types::bidirectional::AS2`
- **FWD_OUT** (Plug): spiegelt den Vorwärts-Kanal (`IN.EO1`/`DO1`), Typ `adapter::types::unidirectional::AS`
- **REV_OUT** (Plug): spiegelt den Rückwärts-Kanal (`OUT.EI1`/`DI1`), Typ `adapter::types::unidirectional::AS`

Funktionsweise

Der Baustein besteht nur aus Verdrahtung:

- **Vorwärts** (Indikation): `IN.EO1` ⟶ `OUT.EO1` und zusätzlich `IN.EO1` ⟶ `FWD_OUT.E1`; `IN.DO1` ⟶ `OUT.DO1` und zusätzlich `IN.DO1` ⟶ `FWD_OUT.D1`.
- **Rückwärts** (Request): `OUT.EI1` ⟶ `IN.EI1` und zusätzlich `OUT.EI1` ⟶ `REV_OUT.E1`; `OUT.DI1` ⟶ `IN.DI1` und zusätzlich `OUT.DI1` ⟶ `REV_OUT.D1`.

Beide Richtungen laufen also unverändert durch, und jede wird zusätzlich als einfaches unidirektionales AS-Signal ausgegeben.

Technische Besonderheiten

- Reine Verdrahtung ohne Logik oder Zustand
- Der Passthrough `IN` ⟶ `OUT` bleibt vollständig erhalten; `FWD_OUT` und `REV_OUT` sind reine Beobachtungs-Abzweige
- Die Quellen werden jeweils auf zwei Ziele verzweigt (Ereignis und Daten); das Verhalten entspricht dem eines Splits auf beide Zweige

Anwendungsszenarien

- Mitlesen der Vorwärts- und der Rückwärts-Richtung einer AS2-Verbindung (z. B. für Anzeige, Logging oder Diagnose), ohne die Verbindung selbst zu verändern
- Getrennte Weiterverarbeitung von Indikation und Request als einfache AS-Signale

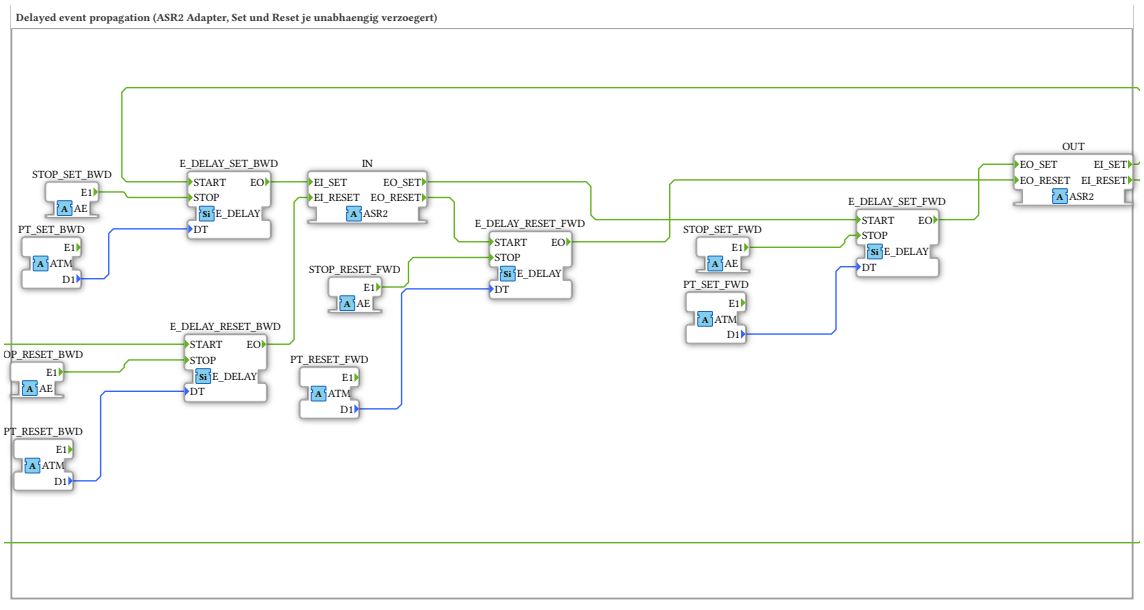
⚖️ Vergleich mit ähnlichen Bausteinen

Dieselbe Struktur existiert für weitere bidirektionale Adapter, z. B. `AR2_SPLIT_FWD_REV`, `AS2_SPLIT_FWD_REV`, `AX2_SPLIT_FWD_REV`, `AIS2_SPLIT_FWD_REV` und `ALR2_SPLIT_FWD_REV`; sie unterscheiden sich nur im Datentyp des Adapters.

Fazit

`AS2_SPLIT_FWD_REV` ist ein zustandsloser Passthrough für AS2, der Vorwärts- und Rückwärts-Kanal zusätzlich auf zwei unidirektionale AS-Plugs spiegelt.

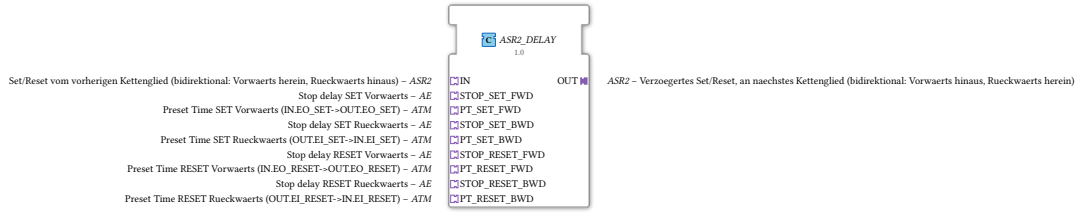
ASR2_DELAY



• — ○ • — ○ • — ○ •

Einleitung

`ASR2_DELAY` ist ein Composite-Funktionsblock, der die Set-/Reset-Ereignisse eines bidirektionalen `ASR2`-Adapters **verzögert** weiterreicht (z. B. für gestaffeltes Ein-/Ausschalten mehrerer Bänder mit `AX_T_FF_SR_SEQ`). Da `ASR2` echt bidirektional ist (`EI_SET/EI_RESET/EO_SET/EO_RESET`), werden **vier unabhängige** `E_DELAY`-Instanzen verwendet: Set und Reset je in Vorwärts- und Rückwärts-Richtung, jeweils mit eigener Zeit und eigenem Stop-Eingang.



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden – Ereignisse kommen über die Adapter-Sockets/-Plugs

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Daten-Eingänge vorhanden – die Zeiten kommen über `ATM`-Sockets

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

- **IN** (Socket): `adapter::types::bidirectional::ASR2` – Set/Reset vom vorherigen Kettenglied (vorwärts herein, rückwärts hinaus)
- **OUT** (Plug): `adapter::types::bidirectional::ASR2` – verzögertes Set/Reset an das nächste Kettenglied (vorwärts hinaus, rückwärts herein)
- **PT_SET_FWD** (Socket, `adapter::types::unidirectional::ATM`): Verzögerungszeit SET vorwärts (`IN.EO_SET` ⟶ `OUT.EO_SET`)
- **PT_SET_BWD** (Socket, `ATM`): Verzögerungszeit SET rückwärts (`OUT.EI_SET` ⟶ `IN.EI_SET`)
- **PT_RESET_FWD** (Socket, `ATM`): Verzögerungszeit RESET vorwärts (`IN.EO_RESET` ⟶ `OUT.EO_RESET`)
- **PT_RESET_BWD** (Socket, `ATM`): Verzögerungszeit RESET rückwärts (`OUT.EI_RESET` ⟶ `IN.EI_RESET`)
- **STOP_SET_FWD**, **STOP_SET_BWD**, **STOP_RESET_FWD**, **STOP_RESET_BWD** (Sockets, `adapter::types::unidirectional::AE`): Stoppen der jeweiligen Verzögerung

Verwendete Funktionsbausteine (FBs)

- **E_DELAY_SET_FWD**, **E_DELAY_SET_BWD**, **E_DELAY_RESET_FWD**, **E_DELAY_RESET_BWD**: je `iec61499::events::E_DELAY`

Funktionsweise

Pfad	START	STOP	Zeit (`DT`)	Ausgang
SET vorwärts	`IN.EO_SET`	`STOP_SET_FWD.E1`	`PT_SET_FWD.D1`	`OUT.EO_SET`
SET rückwärts	`OUT.EI_SET`	`STOP_SET_BWD.E1`	`PT_SET_BWD.D1`	`IN.EI_SET`
RESET vorwärts	`IN.EO_RESET`	`STOP_RESET_FWD.E1`	`PT_RESET_FWD.D1`	`OUT.EO_RESET`
RESET rückwärts	`OUT.EI_RESET`	`STOP_RESET_BWD.E1`	`PT_RESET_BWD.D1`	`IN.EI_RESET`

Jedes eintreffende Ereignis startet die zugehörige `E_DELAY`-Instanz; nach Ablauf der Zeit wird das Ereignis am Ausgang ausgelöst, sofern die Verzögerung nicht vorher über den Stop-Socket gestoppt wurde.

Technische Besonderheiten

- Die vier Zeiten sind über `ATM`-Sockets bereitgestellt und damit unabhängig voneinander einstellbar sowie über `initval_ATM` mit `INIT`/`INIT0` initialisierbar (Version 1.1/1.2). Die Bezeichner tragen das Präfix `PT` statt `DT`, da `DT` ein reserviertes Schlüsselwort ist.
- Seit Version 1.2 werden beide Richtungen verzögert (vorher nur vorwärts).

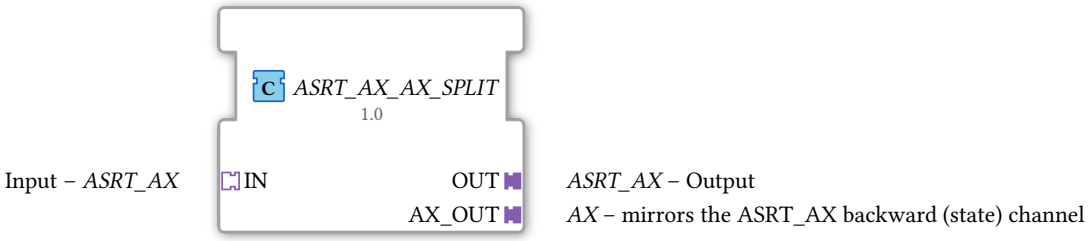
Anwendungsszenarien

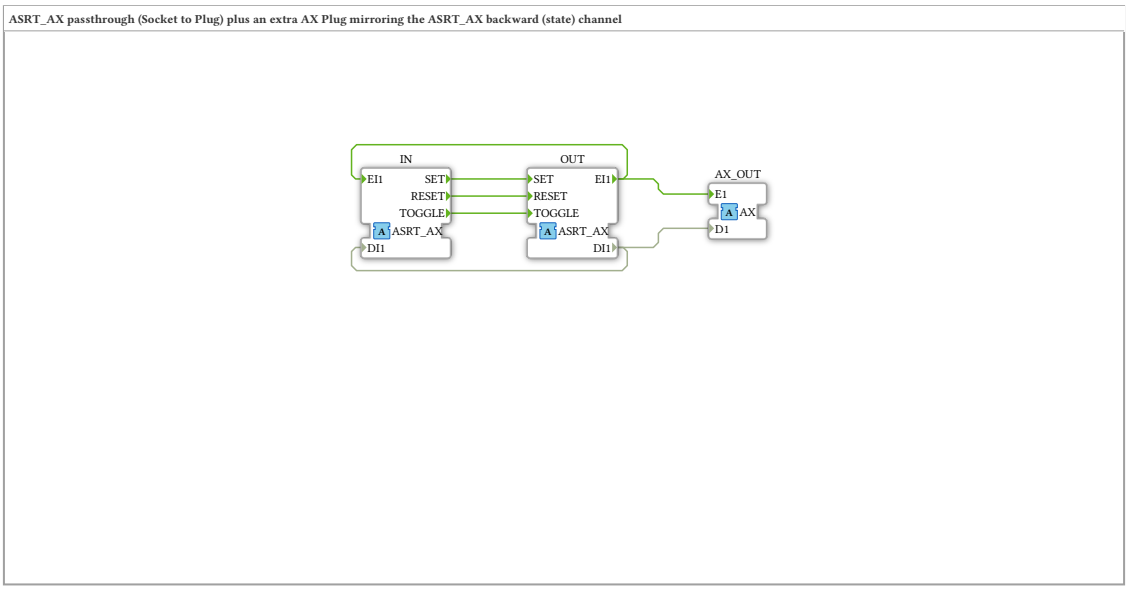
- Gestaffeltes Ein-/Ausschalten hintereinander geschalteter Förderbänder über eine `AX_T_FF_SR_SEQ`-Kette

Fazit

`ASR2_DELAY` verzögert Set und Reset eines `ASR2`-Kettenglieds in beiden Richtungen mit vier getrennt einstellbaren Zeiten und vier Stop-Eingängen.

ASRT_AX_AX_SPLIT





• – ○ • – ○ • – ○ •

Einleitung

Der ASRT_AX_AX_SPLIT ist ein Composite-Funktionsblock, der drei eingehende ASRT_AX-Ereignisse (Set/Reset/Toggle) unverändert von seinem Socket `IN` an den Plug `OUT` durchreicht und zusätzlich den auf dem Rückkanal von `OUT` gemeldeten Zustand (Event + Bool) über einen dritten, unidirektionalen `AX_OUT`-Plug nach außen spiegelt. Er ist die um Toggle erweiterte Variante von ASR_AX_AX_SPLIT.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden – Ereignisse kommen über die Adapter-Sockets/-Plugs

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

- **IN**: Bidirektionaler Adapter-Socket vom Typ `adapter::types::bidirectional::ASRT_AX` (Eingang)
- **OUT**: Bidirektionaler Adapter-Plug vom Typ `adapter::types::bidirectional::ASRT_AX` (Ausgang)
- **AX_OUT**: Unidirektionaler Adapter-Plug vom Typ `adapter::types::unidirectional::AX`, spiegelt den Rückkanal (Zustand) von ASRT_AX

Funktionsweise

1. `IN.SET`, `IN.RESET` und `IN.TOGGLE` werden unverändert an `OUT.SET`, `OUT.RESET` bzw. `OUT.TOGGLE` weitergeleitet – die Vorwärtsrichtung wird 1:1 durchgeschleift.
2. Das Rückkanal-Ereignis `OUT.EI1` (samt zugehörigem Datum `OUT.DI1`) wird sowohl an `IN.EI1`/`IN.DI1` zurückgemeldet als auch gleichzeitig an `AX_OUT.E1`/`AX_OUT.D1` weitergegeben.
3. Dadurch ist der über `OUT` gemeldete Zustand an zwei Stellen sichtbar: am ursprünglichen Socket `IN` und zusätzlich isoliert am `AX_OUT`-Plug.

Technische Besonderheiten

- Reine Ereignis-/Datenverbindungen (`FBNetwork`), keine eigene Logik oder Zustandsverwaltung
- Vorwärtsrichtung (Socket → Plug) ist ein einfacher 1:1-Passthrough für Set, Reset und Toggle
- Rückkanal wird verdoppelt statt aufgeteilt: beide Ziele (`IN` und `AX_OUT`) erhalten dieselbe Information

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand und arbeitet stateless. Jedes eingehende Ereignis wird sofort weitergeleitet bzw. gespiegelt.

Anwendungsszenarien

- Zusätzliches, isoliertes Abgreifen des ASRT_AX-Rückkanals (z. B. für eine separate Anzeige oder Protokollierung) ohne die eigentliche Verbindung zwischen `IN` und `OUT` zu verändern
- Aufbau von Diagnose-/Monitoring-Pfaden in ASRT_AX-basierten Steuerungsnetzwerken

⚖️ Vergleich mit ähnlichen Bausteinen

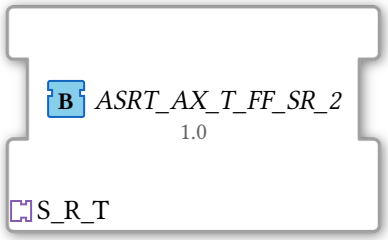
ASR_AX_AX_SPLIT folgt demselben Muster mit nur zwei Vorwärts-Ereignissen (ohne Toggle); AE_AX_AX_SPLIT reduziert es auf ein einzelnes Vorwärts-Ereignis.

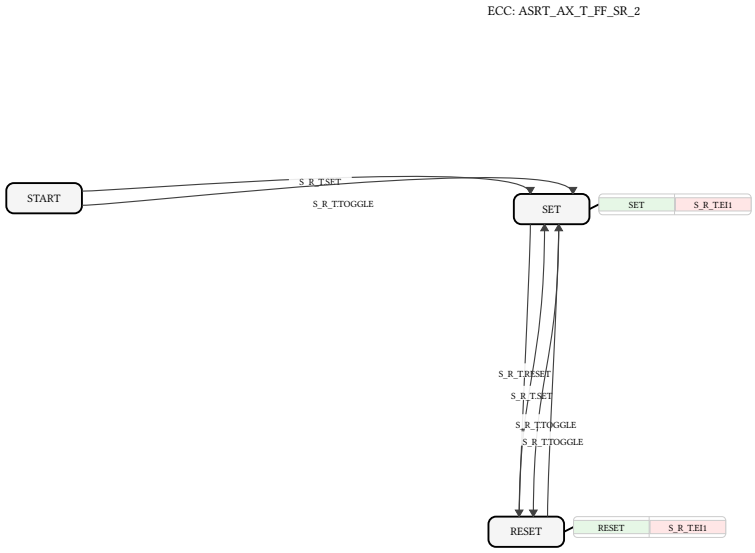
Fazit

Der ASRT_AX_AX_SPLIT ermöglicht es, den AX-Rückkanal eines ASRT_AX-Adapterpaares zusätzlich an einer zweiten Stelle im Netzwerk verfügbar zu machen, ohne die eigentliche Signalkette zu unterbrechen oder zu verändern.

ASRT_AX_T_FF_SR_2

Set/Reset/Toggle (in), Zustand (out) – *ASRT_AX*





• – ○ • – ○ • – ○ •

Einleitung

Der ASRT_AX_T_FF_SR_2 ist ein ereignisgesteuertes bistabiles Flipflop mit Toggle-Funktionalität und einem einzigen bidirektionalen `ASRT_AX`-Socket: Set/Reset/Toggle kommen am selben Adapter an, über den auch der neue Zustand zurückgemeldet wird. Er ist damit die bidirektionale Entsprechung von ASRT_AX_T_FF_SR (unidirektionaler ASRT-Socket + separater AX-Plug) und die um Toggle erweiterte Variante von ASR_AX_SR_2.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden – Set/Reset/Toggle kommen über den Adapter-Socket `S_R_T`

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Dateneingänge vorhanden

Daten-Ausgänge

Keine Datenausgänge vorhanden

Adapter

- **S_R_T**: Bidirektionaler Adapter-Socket vom Typ `adapter::types::bidirectional::ASRT_AX` – Set/Reset/Toggle (Eingang), Zustand (Ausgang) über denselben Adapter

Funktionsweise

Der ASRT_AX_T_FF_SR_2 verfügt über drei Betriebszustände:

- **START**: Initialzustand
- **SET**: Zustand TRUE
- **RESET**: Zustand FALSE

Die Zustandsübergänge werden durch die am Socket `S_R_T` eintreffenden Adapterereignisse gesteuert:

- `S_R_T.SET` führt von jedem Zustand in den SET-Zustand
- `S_R_T.RESET` führt von jedem Zustand in den RESET-Zustand
- `S_R_T.TOGGLE` toggelt den aktuellen Zustand (SET → RESET bzw. RESET → SET, aus START heraus nach SET)

Bei jedem Zustandswechsel wird der entsprechende Algorithmus ausgeführt, der `S_R_T.DI1` setzt (`TRUE` im SET-, `FALSE` im RESET-Zustand) und damit über `S_R_T.EI1` den Zustand zurück über denselben Adapter meldet.

Technische Besonderheiten

- Bidirektionale Kommunikation über einen einzigen Adapter-Socket statt getrenntem ASRT-Socket + AX-Plug
- Identische ECC-Struktur wie ASRT_AX_T_FF_SR einziger Unterschied ist die Adapterrichtung (bidirektional statt unidirektional + separater Plug)
- Direkter Ersatz für ASRT_AX_T_FF_SR in durchgängig adapterbasierten (ASRT_AX-)Anwendungen

Zustandsübersicht

```
START (Initialzustand)
    │
    ├── S_R_T.SET ─────→ SET (S_R_T.DI1 = TRUE)
    │
    └── S_R_T.TOGGLE ──→ SET (S_R_T.DI1 = TRUE)

SET (S_R_T.DI1 = TRUE)
    │
    ├── S_R_T.RESET ──→ RESET (S_R_T.DI1 = FALSE)
    │
    └── S_R_T.TOGGLE ──→ RESET (S_R_T.DI1 = FALSE)

RESET (S_R_T.DI1 = FALSE)
    │
    ├── S_R_T.SET ─────→ SET (S_R_T.DI1 = TRUE)
    │
    └── S_R_T.TOGGLE ──→ SET (S_R_T.DI1 = TRUE)
```

Anwendungsszenarien

- Zustandsspeicherung mit Set/Reset/Toggle über eine einzige bidirektionale Adapterverbindung, ohne separaten Zustandsausgang verdrahten zu müssen
- Ersatz für ASRT_AX_T_FF_SR in Netzwerken, die durchgängig ASRT_AX-Adapter verwenden

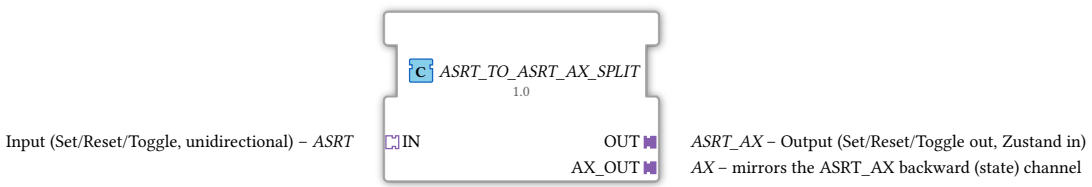
⚖️ Vergleich mit ähnlichen Bausteinen

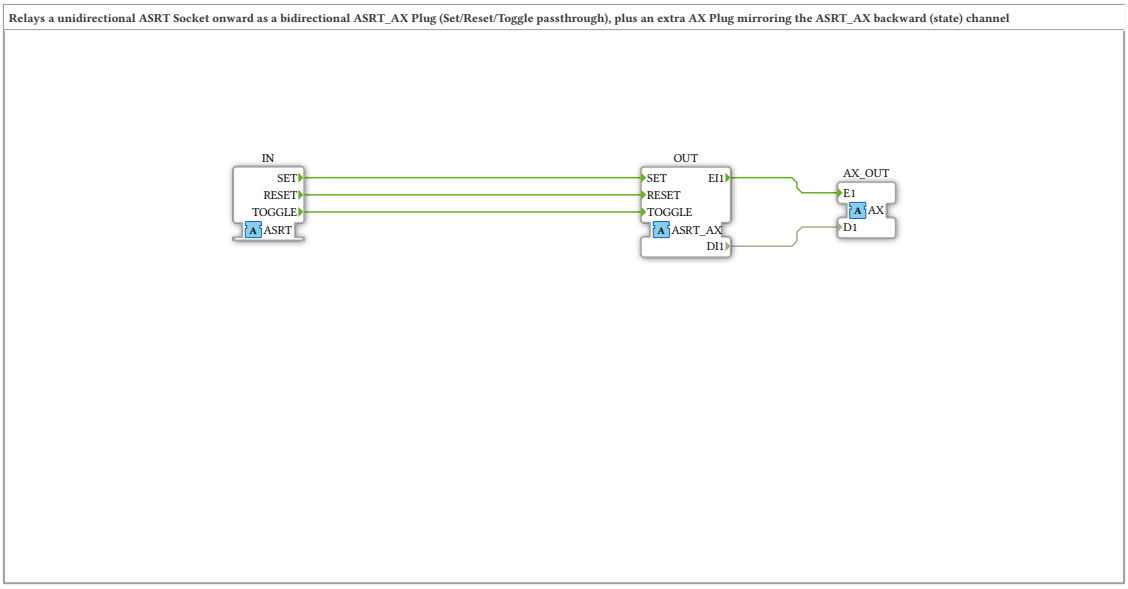
Im Vergleich zu ASRT_AX_T_FF_SR, das einen unidirektionalen ASRT-Socket und einen separaten AX-Plug für den Zustand nutzt, bündelt ASRT_AX_T_FF_SR_2 beide Richtungen in einem einzigen ASRT_AX-Socket. ASR_AX_SR_2 ist dieselbe Baureihe ohne Toggle-Eingang.

Fazit

Der ASRT_AX_T_FF_SR_2 überträgt das bewährte Set/Reset/Toggle-Flipflop-Muster in die bidirektionale Adapterwelt: Steuer-Eingang und Zustands-Rückmeldung teilen sich denselben ASRT_AX-Socket, was die Verdrahtung in adapterbasierten Netzwerken vereinfacht.

ASRT_TO_ASRT_AX_SPLIT





• – ○ • – ○ • – ○ •

Einleitung

Der ASRT_TO_ASRT_AX_SPLIT ist die Set/Reset/Toggle-Variante von `ASR_TO_ASR_AX_SPLIT`: er wandelt ein eingehendes, **unidirektionales** ASRT-Signal (Set/Reset/Toggle, ohne Rückkanal) an seinem Socket `IN` in ein **bidirektionales** ASRT_AX-Signal am Plug `OUT` um, und spiegelt den auf dem Rückkanal von `OUT` gemeldeten Zustand zusätzlich über einen dritten, unidirektionalen `AX_OUT`-Plug nach außen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden – Ereignisse kommen über die Adapter-Sockets/-Plugs

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

- **IN**: Unidirektionaler Adapter-Socket vom Typ `adapter::types::unidirectional::ASRT` (Set/Reset/Toggle-Eingang, kein Rückkanal)
- **OUT**: Bidirektionaler Adapter-Plug vom Typ `adapter::types::bidirectional::ASRT_AX` (Set/Reset/Toggle-Ausgang, mit Rückkanal)
- **AX_OUT**: Unidirektionaler Adapter-Plug vom Typ `adapter::types::unidirectional::AX`, spiegelt den Rückkanal (Zustand) von ASRT_AX nach außen

Funktionsweise

1. Jedes an `IN.SET` eintreffende Ereignis wird unverändert an `OUT.SET` weitergeleitet, ebenso `IN.RESET` an `OUT.RESET` und `IN.TOGGLE` an `OUT.TOGGLE` - alle drei Ereignispfade laufen unabhängig voneinander durch.
2. Das Rückkanal-Ereignis `OUT.EI1` (samt zugehörigem Datum `OUT.DI1`), das die nachgeschaltete Gegenstelle über `OUT` sendet, wird ausschließlich an `AX_OUT.E1`/`AX_OUT.D1` weitergegeben - eine Rückmeldung an `IN` ist technisch unmöglich, da `IN` als unidirektionaler ASRT-Adapter keinen Rückkanal besitzt.
3. `AX_OUT` ist damit der einzige Ort, an dem der von `OUT` gemeldete Zustand sichtbar wird.

Technische Besonderheiten

- Reine Ereignis-/Datenverbindungen (`FBNetwork`), keine eigene Logik oder Zustandsverwaltung
- Drei unabhängige Vorwärtspfade (SET, RESET, TOGGLE), jeweils ein einfacher 1:1-Passthrough
- Wandelt einen unidirektionalen Adapter (`IN`) in einen bidirektionalen (`OUT`) um, ohne selbst einen Rückkanal am Eingang zu benötigen
- Der Rückkanal wird nicht verdoppelt, sondern einzig über `AX_OUT` exponiert

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand und arbeitet stateless. Jedes eingehende Ereignis wird sofort weitergeleitet bzw. gespiegelt.

Anwendungsszenarien

- Anschluss eines bestehenden, rein unidirektionalen ASRT-Signalgebers (z. B. ein SoftKey mit Set/Reset/Toggle-Verhalten) an eine Kette, die einen bidirektionalen ASRT_AX-Adapter erwartet
- Sichtbarmachen des Zustands der nachgeschalteten ASRT_AX-Gegenstelle über `AX_OUT`, obwohl die ursprüngliche Quelle (`IN`) selbst keinen Rückkanal unterstützt

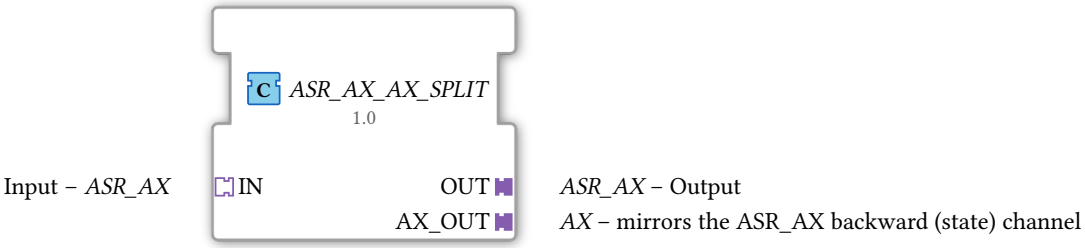
⚖️ Vergleich mit ähnlichen Bausteinen

Strukturell identisch zu `ASR_TO_ASR_AX_SPLIT`, nur mit einem zusätzlichen dritten Vorwärts-Ereignis (TOGGLE). Für die einfachere Set/Reset-Variante ohne Toggle siehe `ASR_TO_ASR_AX_SPLIT`, für die reine Ereignis-Variante ohne Set/Reset-Semantik `AE_TO_AE_AX_SPLIT`.

Fazit

Der ASRT_TO_ASRT_AX_SPLIT ermöglicht den Anschluss eines rein unidirektionalen ASRT-Signalgebers an eine bidirektionale ASRT_AX-Kette und macht deren Rückkanal isoliert über `AX_OUT` verfügbar, ohne dass am ursprünglichen Eingang selbst eine Rückmeldung möglich sein muss.

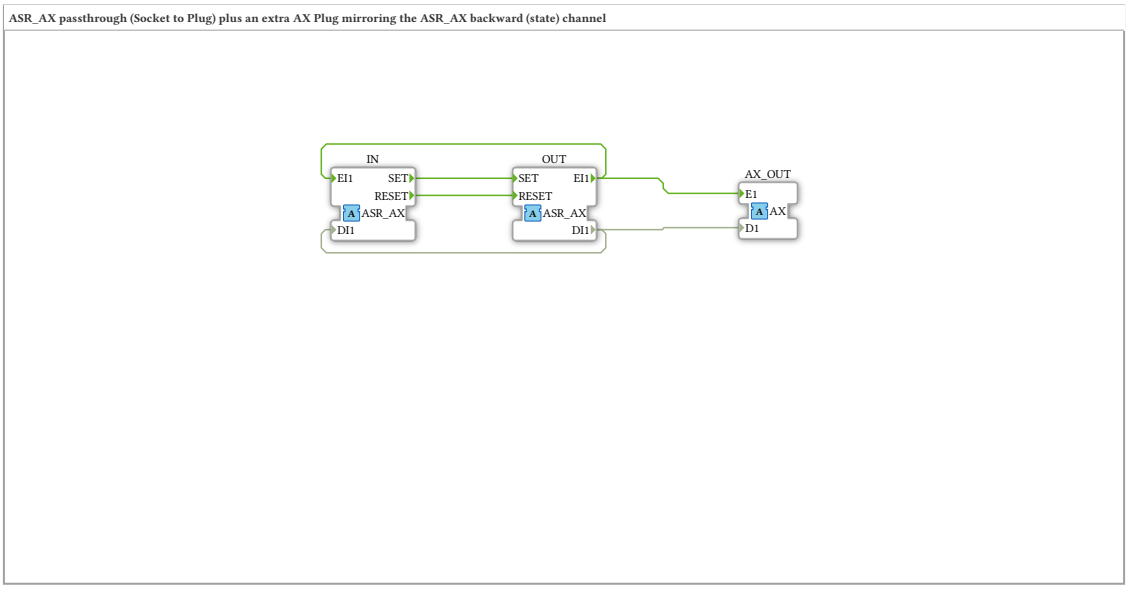
ASR_AX_AX_SPLIT



Input – *ASR_AX*

ASR_AX – Output

AX – mirrors the ASR_AX backward (state) channel



• – ○ • – ○ • – ○ •

Einleitung

Der ASR_AX_AX_SPLIT ist ein Composite-Funktionsblock, der zwei eingehende ASR_AX-Ereignisse (Set/Reset) unverändert von seinem Socket `IN` an den Plug `OUT` durchreicht und zusätzlich den auf dem Rückkanal von `OUT` gemeldeten Zustand (Event + Bool) über einen dritten, unidirektionalen `AX_OUT`-Plug nach außen spiegelt. Er ist die um Set/Reset erweiterte Variante von AE_AX_AX_SPLIT.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden – Ereignisse kommen über die Adapter-Sockets/-Plugs

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

- **IN**: Bidirektionaler Adapter-Socket vom Typ `adapter::types::bidirectional::ASR_AX` (Eingang)
- **OUT**: Bidirektionaler Adapter-Plug vom Typ `adapter::types::bidirectional::ASR_AX` (Ausgang)
- **AX_OUT**: Unidirektionaler Adapter-Plug vom Typ `adapter::types::unidirectional::AX`, spiegelt den Rückkanal (Zustand) von ASR_AX

Funktionsweise

1. `IN.SET` und `IN.RESET` werden unverändert an `OUT.SET` bzw. `OUT.RESET` weitergeleitet – die Vorwärtsrichtung wird 1:1 durchgeschleift.
2. Das Rückkanal-Ereignis `OUT.EI1` (samt zugehörigem Datum `OUT.DI1`) wird sowohl an `IN.EI1`/`IN.DI1` zurückgemeldet als auch gleichzeitig an `AX_OUT.E1`/`AX_OUT.D1` weitergegeben.
3. Dadurch ist der über `OUT` gemeldete Zustand an zwei Stellen sichtbar: am ursprünglichen Socket `IN` und zusätzlich isoliert am `AX_OUT`-Plug.

Technische Besonderheiten

- Reine Ereignis-/Datenverbindungen (`FBNetwork`), keine eigene Logik oder Zustandsverwaltung
- Vorwärtsrichtung (Socket ⟶ Plug) ist ein einfacher 1:1-Passthrough für Set und Reset
- Rückkanal wird verdoppelt statt aufgeteilt: beide Ziele (`IN` und `AX_OUT`) erhalten dieselbe Information

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand und arbeitet stateless. Jedes eingehende Ereignis wird sofort weitergeleitet bzw. gespiegelt.

Anwendungsszenarien

- Zusätzliches, isoliertes Abgreifen des ASR_AX-Rückkanals (z. B. für eine separate Anzeige oder Protokollierung) ohne die eigentliche Verbindung zwischen `IN` und `OUT` zu verändern
- Aufbau von Diagnose-/Monitoring-Pfaden in ASR_AX-basierten Steuerungsnetzwerken

⚖️ Vergleich mit ähnlichen Bausteinen

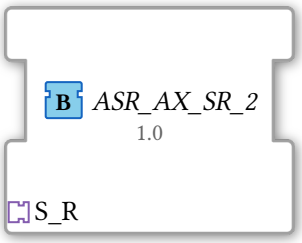
AE_AX_AX_SPLIT folgt demselben Muster mit nur einem Vorwärts-Ereignis; ASRT_AX_AX_SPLIT erweitert es um ein drittes Vorwärts-Ereignis (Toggle).

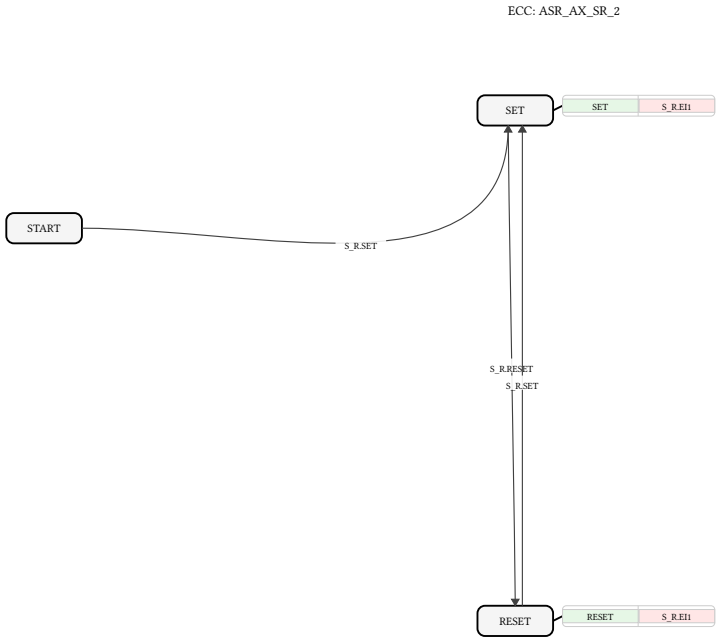
Fazit

Der ASR_AX_AX_SPLIT ermöglicht es, den AX-Rückkanal eines ASR_AX-Adapterpaares zusätzlich an einer zweiten Stelle im Netzwerk verfügbar zu machen, ohne die eigentliche Signalkette zu unterbrechen oder zu verändern.

ASR_AX_SR_2

Set/Reset (in), Zustand (out) – *ASR_AX*





• – ○ • – ○ • – ○ •

Einleitung

Der ASR_AX_SR_2 ist ein ereignisgesteuertes bistabiles Flipflop (Set/Reset) mit einem einzigen bidirektionalen `ASR_AX`-Socket: Set- und Reset-Ereignis kommen am selben Adapter an, über den auch der neue Zustand zurückgemeldet wird – im Gegensatz zu ASR_AX_SR, das einen unidirektionalen ASR-Socket und einen separaten AX-Plug für den Zustand verwendet.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden – Set/Reset kommen über den Adapter-Socket `S_R`

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Dateneingänge vorhanden

Daten-Ausgänge

Keine Datenausgänge vorhanden

Adapter

- **S_R**: Bidirektionaler Adapter-Socket vom Typ `adapter::types::bidirectional::ASR_AX` – Set/Reset (Eingang), Zustand (Ausgang) über denselben Adapter

Funktionsweise

Der ASR_AX_SR_2 verfügt über drei Betriebszustände:

- **START**: Initialzustand
- **SET**: Zustand TRUE
- **RESET**: Zustand FALSE

Die Zustandsübergänge werden durch die am Socket `S_R` eintreffenden Adapterereignisse gesteuert:

- `S_R.SET` führt von START oder RESET in den SET-Zustand
- `S_R.RESET` führt vom SET-Zustand in den RESET-Zustand

Bei jedem Zustandswechsel wird der entsprechende Algorithmus ausgeführt, der `S_R.DI1` setzt (`TRUE` im SET-, `FALSE` im RESET-Zustand) und damit über `S_R.EI1` den Zustand zurück über denselben Adapter meldet.

Technische Besonderheiten

- Bidirektionale Kommunikation über einen einzigen Adapter-Socket statt getrenntem ASR-Socket + AX-Plug
- Kein Toggle-Eingang – reines Set/Reset-Flipflop (im Unterschied zu ASRT_AX_T_FF_SR_2)
- Direkter Ersatz für ASR_AX_SR in durchgängig adapterbasierten (ASR_AX-)Anwendungen

Zustandsübersicht

```
START (Initialzustand)
    │
    └── S_R.SET ──→ SET (S_R.DI1 = TRUE)

SET (S_R.DI1 = TRUE)
    │
    └── S_R.RESET ─→ RESET (S_R.DI1 = FALSE)

RESET (S_R.DI1 = FALSE)
    │
    └── S_R.SET ──→ SET (S_R.DI1 = TRUE)
```

Anwendungsszenarien

- Zustandsspeicherung (Set/Reset) über eine einzige bidirektionale Adapterverbindung, ohne separaten Zustandsausgang verdrahten zu müssen
- Ersatz für ASR_AX_SR in Netzwerken, die durchgängig ASR_AX-Adapter verwenden

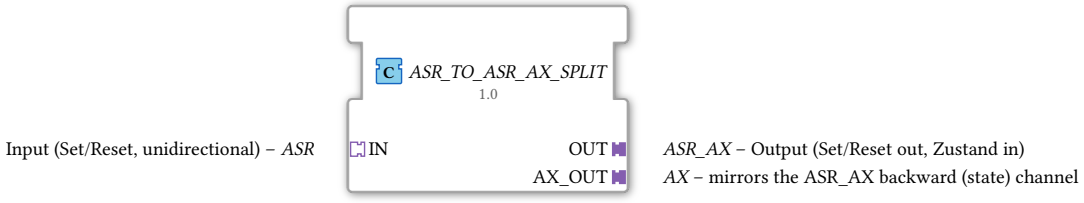
⚖️ Vergleich mit ähnlichen Bausteinen

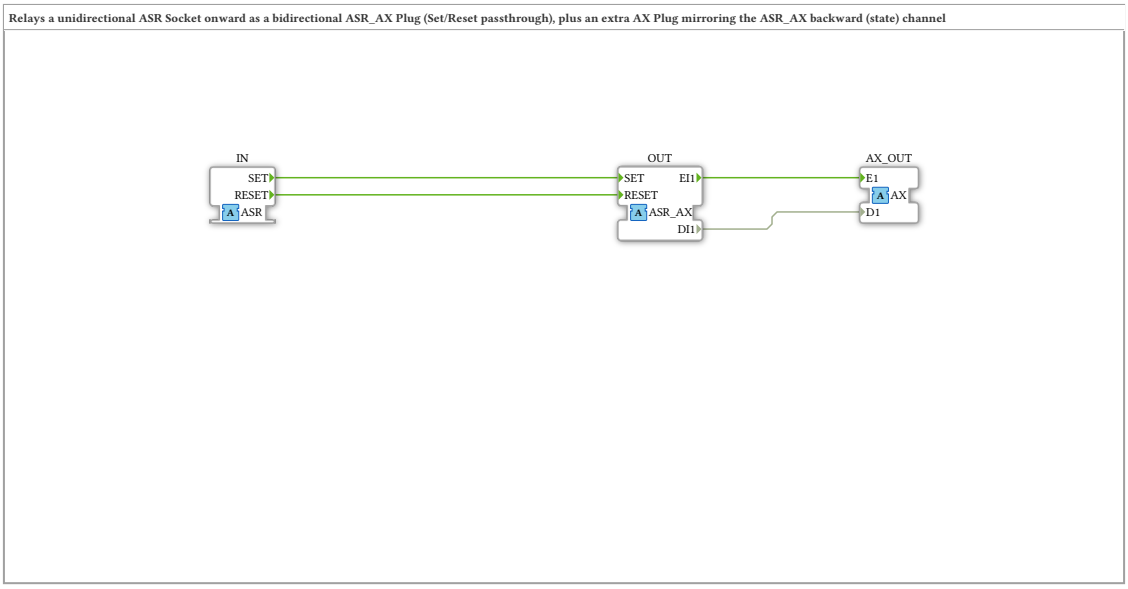
Im Vergleich zu ASR_AX_SR, das einen unidirektionalen ASR-Socket und einen separaten AX-Plug für den Zustand nutzt, bündelt ASR_AX_SR_2 beide Richtungen in einem einzigen ASR_AX-Socket. ASRT_AX_T_FF_SR_2 erweitert dasselbe Muster um einen zusätzlichen Toggle-Eingang.

Fazit

Der ASR_AX_SR_2 überträgt das bewährte Set/Reset-Flipflop-Muster in die bidirektionale Adapterwelt: Set/Reset-Eingang und Zustands-Rückmeldung teilen sich denselben ASR_AX-Socket, was die Verdrahtung in adapterbasierten Netzwerken vereinfacht.

ASR_TO_ASR_AX_SPLIT





• – ○ • – ○ • – ○ •

Einleitung

Der ASR_TO_ASR_AX_SPLIT ist die Set/Reset-Variante von AE_TO_AE_AX_SPLIT: er wandelt ein eingehendes, **unidirektionales** ASR-Signal (Set/Reset, ohne Rückkanal) an seinem Socket `IN` in ein **bidirektionales** ASR_AX-Signal am Plug `OUT` um, und spiegelt den auf dem Rückkanal von `OUT` gemeldeten Zustand zusätzlich über einen dritten, unidirektionalen `AX_OUT`-Plug nach außen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden – Ereignisse kommen über die Adapter-Sockets/-Plugs

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

- **IN**: Unidirektionaler Adapter-Socket vom Typ `adapter::types::unidirectional::ASR` (Set/Reset-Eingang, kein Rückkanal)
- **OUT**: Bidirektionaler Adapter-Plug vom Typ `adapter::types::bidirectional::ASR_AX` (Set/Reset-Ausgang, mit Rückkanal)
- **AX_OUT**: Unidirektionaler Adapter-Plug vom Typ `adapter::types::unidirectional::AX`, spiegelt den Rückkanal (Zustand) von ASR_AX nach außen

Funktionsweise

1. Jedes an `IN.SET` eintreffende Ereignis wird unverändert an `OUT.SET` weitergeleitet, ebenso jedes `IN.RESET` an `OUT.RESET` - beide Ereignispfade laufen unabhängig voneinander durch.
2. Das Rückkanal-Ereignis `OUT.EI1` (samt zugehörigem Datum `OUT.DI1`), das die nachgeschaltete Gegenstelle über `OUT` sendet, wird ausschließlich an `AX_OUT.E1`/`AX_OUT.D1` weitergegeben - eine Rückmeldung an `IN` ist technisch unmöglich, da `IN` als unidirektionaler ASR-Adapter keinen Rückkanal besitzt.
3. `AX_OUT` ist damit der einzige Ort, an dem der von `OUT` gemeldete Zustand sichtbar wird.

Technische Besonderheiten

- Reine Ereignis-/Datenverbindungen (`FBNetwork`), keine eigene Logik oder Zustandsverwaltung
- Zwei unabhängige Vorwärtspfade (SET, RESET), jeweils ein einfacher 1:1-Passthrough
- Wandelt einen unidirektionalen Adapter (`IN`) in einen bidirektionalen (`OUT`) um, ohne selbst einen Rückkanal am Eingang zu benötigen
- Der Rückkanal wird nicht verdoppelt, sondern einzig über `AX_OUT` exponiert

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand und arbeitet stateless. Jedes eingehende Ereignis wird sofort weitergeleitet bzw. gespiegelt.

Anwendungsszenarien

- Anschluss eines bestehenden, rein unidirektionalen ASR-Signalgebers (z. B. ein einfacher Set/Reset-Taster) an eine Kette, die einen bidirektionalen ASR_AX-Adapter erwartet
- Sichtbarmachen des Zustands der nachgeschalteten ASR_AX-Gegenstelle über `AX_OUT`, obwohl die ursprüngliche Quelle (`IN`) selbst keinen Rückkanal unterstützt

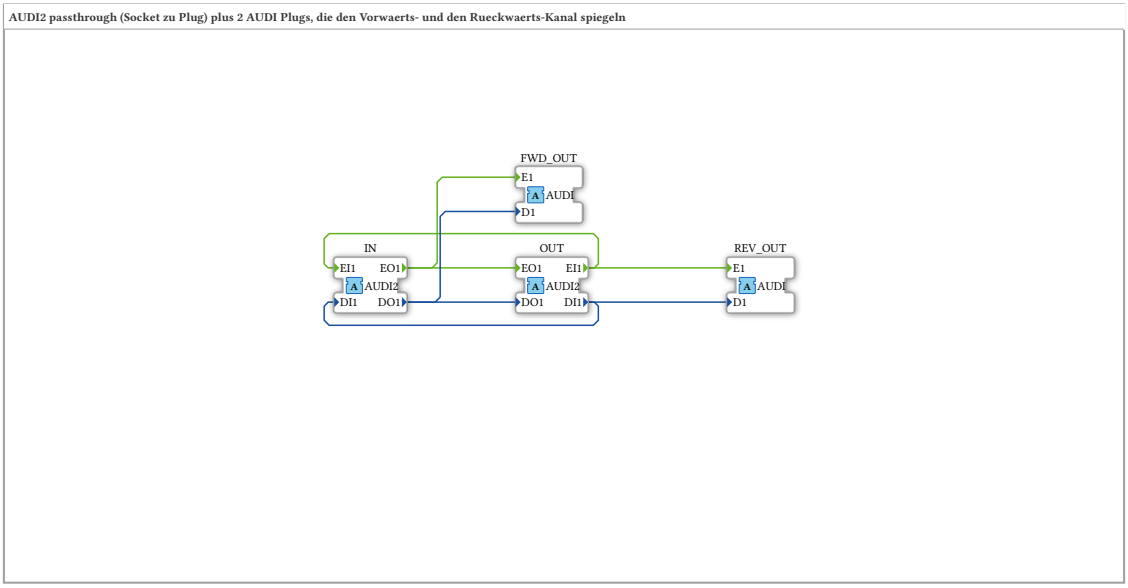
⚖️ Vergleich mit ähnlichen Bausteinen

Strukturell identisch zu `AE_TO_AE_AX_SPLIT`, nur mit zwei statt einem Vorwärts-Ereignis (SET, RESET). Für die zusätzliche TOGGLE-Variante siehe `ASRT_TO_ASRT_AX_SPLIT`.

Fazit

Der ASR_TO_ASR_AX_SPLIT ermöglicht den Anschluss eines rein unidirektionalen ASR-Signalgebers an eine bidirektionale ASR_AX-Kette und macht deren Rückkanal isoliert über `AX_OUT` verfügbar, ohne dass am ursprünglichen Eingang selbst eine Rückmeldung möglich sein muss.

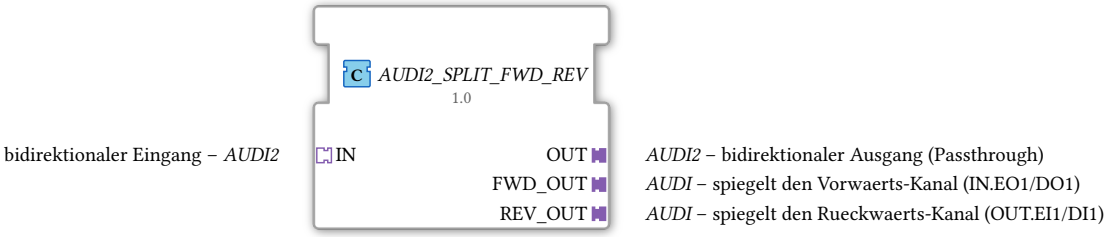
AUDI2_SPLIT_FWD_REV



• — ○ • — ○ • — ○ •

Einleitung

`AUDI2_SPLIT_FWD_REV` ist ein Composite-Funktionsblock, der einen bidirektionalen `AUDI2`-Adapter vom Socket `IN` unverändert zum Plug `OUT` durchreicht (Passthrough) und zusätzlich den Vorwärts- und den Rückwärts-Kanal auf zwei getrennte, unidirektionale `AUDI`-Plugs spiegelt. So können beide Richtungen einer `AUDI2`-Verbindung an weiteren Stellen mitgelesen werden, ohne die Verbindung selbst zu verändern.



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden – Ereignisse kommen über die Adapter-Sockets/-Plugs

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

- **IN** (Socket): Bidirektionaler Adapter vom Typ `adapter::types::bidirectional::AUDI2` (bidirektionaler Eingang)
- **OUT** (Plug): Bidirektionaler Adapter vom Typ `adapter::types::bidirectional::AUDI2` (bidirektionaler Ausgang, Passthrough)
- **FWD_OUT** (Plug): Unidirektionaler Adapter vom Typ `adapter::types::unidirectional::AUDI`, spiegelt den Vorwärts-Kanal (`IN.EO1/DO1`)
- **REV_OUT** (Plug): Unidirektionaler Adapter vom Typ `adapter::types::unidirectional::AUDI`, spiegelt den Rückwärts-Kanal (`OUT.EI1/DI1`)

Funktionsweise

1. **Vorwärts (Indikation)**: `IN.EO1/IN.DO1` wird unverändert an `OUT.EO1/OUT.DO1` weitergeleitet und zusätzlich an `FWD_OUT.E1/FWD_OUT.D1` gespiegelt.
2. **Rückwärts (Request)**: `OUT.EI1/OUT.DI1` wird unverändert an `IN.EI1/IN.DI1` weitergeleitet und zusätzlich an `REV_OUT.E1/REV_OUT.D1` gespiegelt.

Technische Besonderheiten

- Reine Ereignis-/Datenverbindungen (`FBNetwork`), keine eigene Logik und kein interner Zustand
- Jeder Ereignis-Ausgang des Passthrough speist zwei Ziele (Fan-out), die Daten laufen parallel zu den Ereignissen
- Die beiden Spiegel-Plugs sind unidirektional (`AUDI`) und liefern daher nur Lesezugriff auf den jeweiligen Kanal

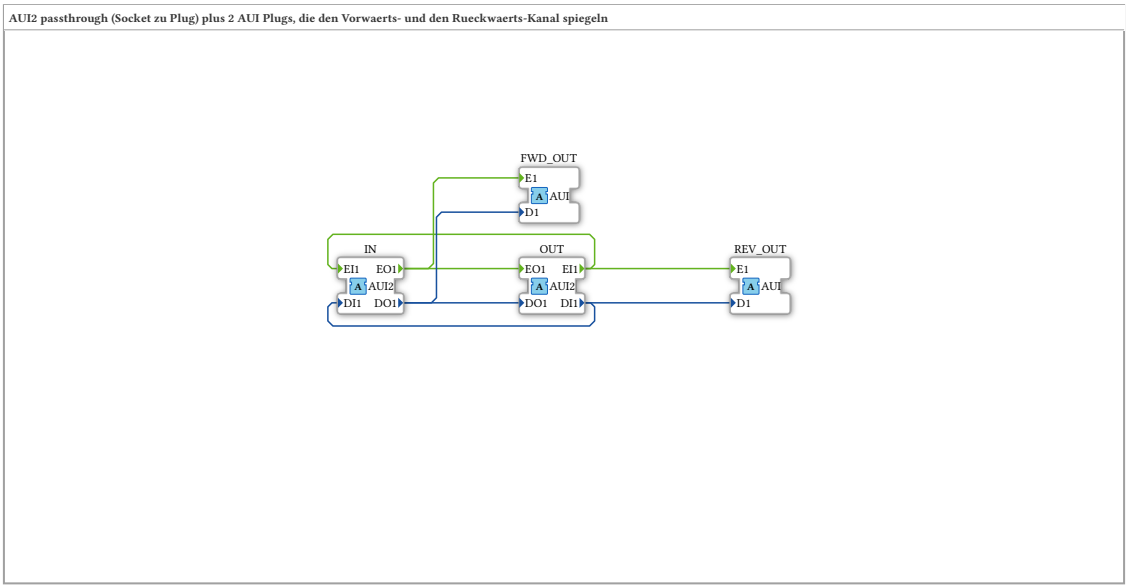
Anwendungsszenarien

- Mitlesen (Monitoring/Diagnose) beider Richtungen einer bidirektionalen `AUDI2`-Verbindung
- Aufteilen einer `AUDI2`-Verbindung in einen separat auswertbaren Vorwärts- und Rückwärtsstrom, ohne die Hauptverbindung zu unterbrechen

Fazit

`AUDI2_SPLIT_FWD_REV` reicht eine bidirektionale `AUDI2`-Verbindung transparent durch und stellt Vorwärts- und Rückwärts-Kanal zusätzlich als getrennte `AUDI`-Adapter zur Verfügung.

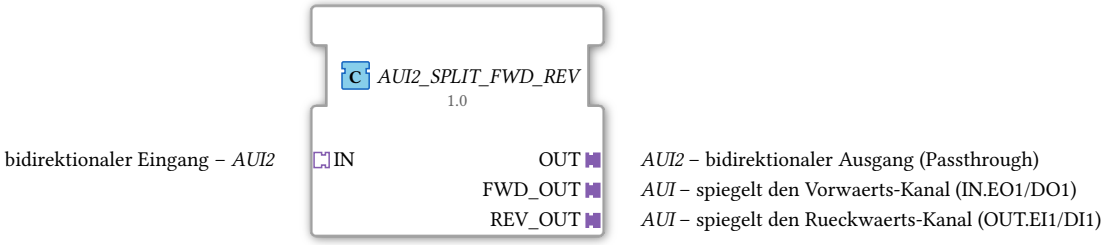
AUI2_SPLIT_FWD_REV



• – ○ • – ○ • – ○ •

Einleitung

`AUI2_SPLIT_FWD_REV` ist ein Composite-Funktionsblock, der einen bidirektionalen `AUI2`-Adapter vom Socket `IN` unverändert zum Plug `OUT` durchreicht (Passthrough) und zusätzlich den Vorwärts- und den Rückwärts-Kanal auf zwei getrennte, unidirektionale `AUI`-Plugs spiegelt. So können beide Richtungen einer `AUI2`-Verbindung an weiteren Stellen mitgelesen werden, ohne die Verbindung selbst zu verändern.



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden – Ereignisse kommen über die Adapter-Sockets/-Plugs

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

- **IN** (Socket): Bidirektionaler Adapter vom Typ `adapter::types::bidirectional::AUI2` (bidirektionaler Eingang)
- **OUT** (Plug): Bidirektionaler Adapter vom Typ `adapter::types::bidirectional::AUI2` (bidirektionaler Ausgang, Passthrough)
- **FWD_OUT** (Plug): Unidirektionaler Adapter vom Typ `adapter::types::unidirectional::AUI`, spiegelt den Vorwärts-Kanal (`IN.EO1/DO1`)

- **REV_OUT** (Plug): Unidirektionaler Adapter vom Typ `adapter::types::unidirectional::AUI`, spiegelt den Rückwärts-Kanal (`OUT.EI1/DI1`)

Funktionsweise

1. **Vorwärts (Indikation)**: `IN.EO1/IN.DO1` wird unverändert an `OUT.EO1/OUT.DO1` weitergeleitet und zusätzlich an `FWD_OUT.E1/FWD_OUT.D1` gespiegelt.
2. **Rückwärts (Request)**: `OUT.EI1/OUT.DI1` wird unverändert an `IN.EI1/IN.DI1` weitergeleitet und zusätzlich an `REV_OUT.E1/REV_OUT.D1` gespiegelt.

Technische Besonderheiten

- Reine Ereignis-/Datenverbindungen (`FBNetwork`), keine eigene Logik und kein interner Zustand
- Jeder Ereignis-Ausgang des Passthrough speist zwei Ziele (Fan-out), die Daten laufen parallel zu den Ereignissen
- Die beiden Spiegel-Plugs sind unidirektional (`AUI`) und liefern daher nur Lesezugriff auf den jeweiligen Kanal

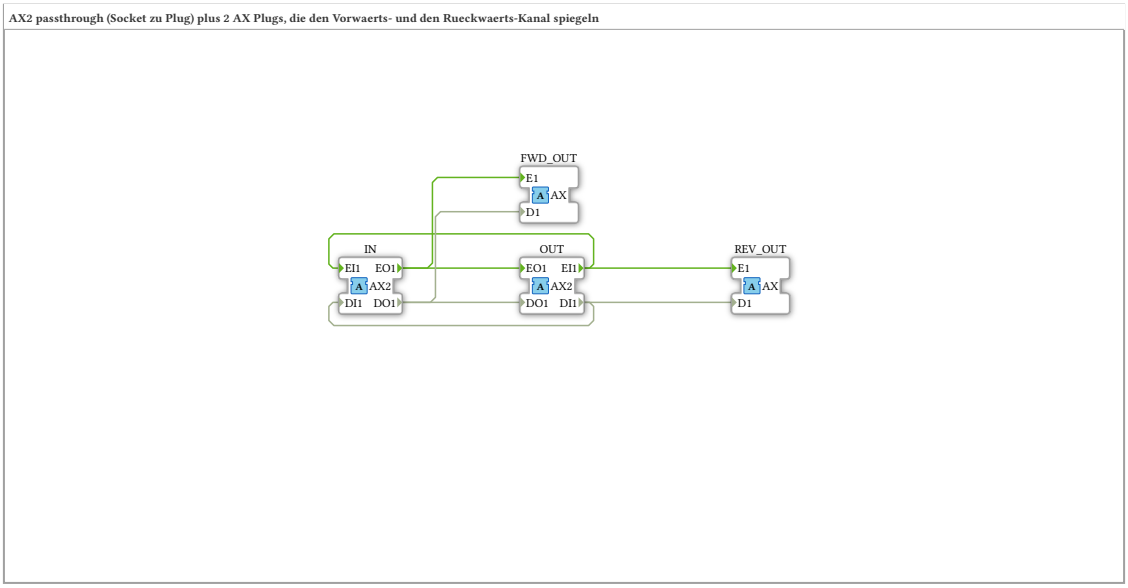
Anwendungsszenarien

- Mitlesen (Monitoring/Diagnose) beider Richtungen einer bidirektionalen `AUI2`-Verbindung
- Aufteilen einer `AUI2`-Verbindung in einen separat auswertbaren Vorwärts- und Rückwärtsstrom, ohne die Hauptverbindung zu unterbrechen

Fazit

`AUI2_SPLIT_FWD_REV` reicht eine bidirektionale `AUI2`-Verbindung transparent durch und stellt Vorwärts- und Rückwärts-Kanal zusätzlich als getrennte `AUI`-Adapter zur Verfügung.

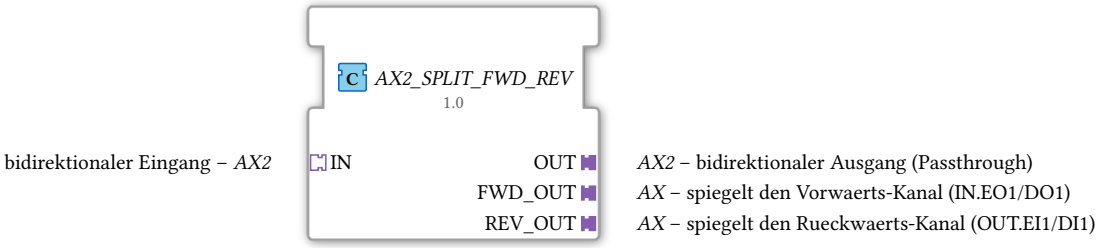
AX2_SPLIT_FWD_REV



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AX2_SPLIT_FWD_REV` führt einen bidirektionalen `AX2`-Adapter unverändert von `IN` (Socket) nach `OUT` (Plug) durch und spiegelt dabei zusätzlich beide Kanäle auf je einen unidirektionalen `AX`-Plug: den Vorwärts-Kanal (Indikation, `IN.EO1/DO1`) auf `FWD_OUT` und den Rückwärts-Kanal (Request, `OUT.EI1/DI1`) auf `REV_OUT`. Damit lassen sich beide Richtungen einer bidirektionalen AX2-Verbindung getrennt als einfache AX-Signale abgreifen.



Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock verfügt über keine direkten Ereignis-Eingänge.

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **IN** (Socket): bidirektionaler Eingang, Typ `adapter::types::bidirectional::AX2`
- **OUT** (Plug): bidirektionaler Ausgang (Passthrough), Typ `adapter::types::bidirectional::AX2`
- **FWD_OUT** (Plug): spiegelt den Vorwärts-Kanal (`IN.EO1/DO1`), Typ `adapter::types::unidirectional::AX`
- **REV_OUT** (Plug): spiegelt den Rückwärts-Kanal (`OUT.EI1/DI1`), Typ `adapter::types::unidirectional::AX`

Funktionsweise

Der Baustein besteht nur aus Verdrahtung:

- **Vorwärts** (Indikation): `IN.EO1` ⟶ `OUT.EO1` und zusätzlich `IN.EO1` ⟶ `FWD_OUT.E1`; `IN.DO1` ⟶ `OUT.DO1` und zusätzlich `IN.DO1` ⟶ `FWD_OUT.D1`.
- **Rückwärts** (Request): `OUT.EI1` ⟶ `IN.EI1` und zusätzlich `OUT.EI1` ⟶ `REV_OUT.E1`; `OUT.DI1` ⟶ `IN.DI1` und zusätzlich `OUT.DI1` ⟶ `REV_OUT.D1`.

Beide Richtungen laufen also unverändert durch, und jede wird zusätzlich als einfaches unidirektionales AX-Signal ausgegeben.

Technische Besonderheiten

- Reine Verdrahtung ohne Logik oder Zustand
- Der Passthrough `IN` ⟶ `OUT` bleibt vollständig erhalten; `FWD_OUT` und `REV_OUT` sind reine Beobachtungs-Abzweige
- Die Quellen werden jeweils auf zwei Ziele verzweigt (Ereignis und Daten); das Verhalten entspricht dem eines Splits auf beide Zweige

Anwendungsszenarien

- Mitlesen der Vorwärts- und der Rückwärts-Richtung einer AX2-Verbindung (z. B. für Anzeige, Logging oder Diagnose), ohne die Verbindung selbst zu verändern
- Getrennte Weiterverarbeitung von Indikation und Request als einfache AX-Signale

⚖️ Vergleich mit ähnlichen Bausteinen

Dieselbe Struktur existiert für weitere bidirektionale Adapter, z. B. `AR2_SPLIT_FWD_REV`, `AS2_SPLIT_FWD_REV`, `AX2_SPLIT_FWD_REV`, `AIS2_SPLIT_FWD_REV` und `ALR2_SPLIT_FWD_REV`; sie unterscheiden sich nur im Datentyp des Adapters.

Fazit

`AX2_SPLIT_FWD_REV` ist ein zustandsloser Passthrough für AX2, der Vorwärts- und Rückwärts-Kanal zusätzlich auf zwei unidirektionale AX-Plugs spiegelt.

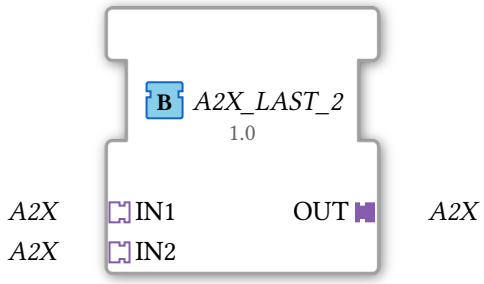
unidirectional

BOOL

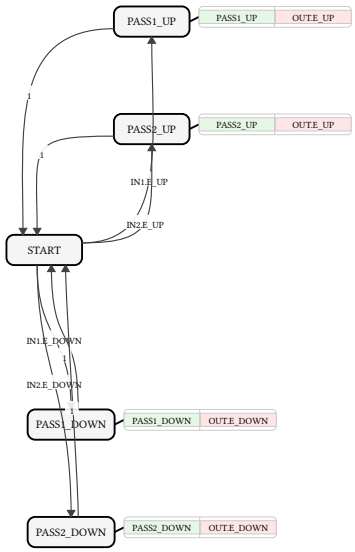
- A2X_SPLIT_2
- A2X_SPLIT_2_UNGATED
- A2X_SPLIT_3
- A2X_SPLIT_3_UNGATED
- A2X_SPLIT_4
- A2X_SPLIT_4_UNGATED
- A2X_SPLIT_5
- A2X_SPLIT_5_UNGATED
- A2X_SPLIT_6
- A2X_SPLIT_6_UNGATED
- A2X_SPLIT_7
- A2X_SPLIT_7_UNGATED
- A2X_SPLIT_8
- A2X_SPLIT_8_UNGATED
- A2X_SPLIT_9
- A2X_SPLIT_9_UNGATED
- ASR_AX_SR
- ASR_AX_SR_SYM
- ASRT_AX_T_FF_SR
- AX_AE_MERGE
- AX_ASR_RF_TRIG
- AX_ASR_RF_TRIG_X
- AX_ASR_SWITCH
- AX_ASR_SWITCH_X
- AX_D_FF
- AX_D_FF_TMIN
- AX_E_PERMIT
- AX_E_SWITCH
- AX_F_TRIG
- AX_PERMIT
- AX_R_TRIG
- AX_RF_TRIG
- AX_RS
- AX_RS_SYM
- AX_RS_SYM_INIT
- AX_SELECT
- AX_SPLIT_2
- AX_SPLIT_2_UNGATED
- AX_SPLIT_3
- AX_SPLIT_3_UNGATED
- AX_SPLIT_4
- AX_SPLIT_4_UNGATED
- AX_SPLIT_5
- AX_SPLIT_5_UNGATED
- AX_SPLIT_6
- AX_SPLIT_6_UNGATED
- AX_SPLIT_7
- AX_SPLIT_7_UNGATED
- AX_SPLIT_8
- AX_SPLIT_8_UNGATED
- AX_SPLIT_9
- AX_SPLIT_9_UNGATED

- AX_SR
- AX_SR_SYM
- AX_SR_SYM_INIT
- AX_SWITCH
- AX_T_FF
- AX_T_FF_INIT
- AX_T_FF_SR
- AX_T_FF_SR_SYM
- AX_T_FF_SR_SYM_INIT
- AX_T_FF_SR_SYM_STORE
- E_RS_SYM
- E_RS_SYM_INIT
- E_SR_SYM
- E_SR_SYM_INIT
- E_T_FF_INIT
- E_T_FF_SR_SYM
- E_T_FF_SR_SYM_INIT
- E_T_FF_SR_SYM_STORE

A2X_LAST_2



ECC: A2X_LAST_2



Einleitung

Der A2X_LAST_2 Funktionsblock führt zwei A2X-Adapter-Signale (**IN1**, **IN2**) auf einen gemeinsamen Ausgang (**OUT**) zusammen. Anders als beim einkanaligen `AX_LAST_2` trägt A2X zwei unabhängige Kanäle

(**UP** und **DOWN**), die deshalb auch unabhängig voneinander nach Last-Writer-Wins gemergt werden: welcher Socket zuletzt auf UP geschrieben hat, gewinnt für UP; welcher zuletzt auf DOWN geschrieben hat, gewinnt für DOWN - unabhängig davon, was auf dem jeweils anderen Kanal passiert.

Schnittstellenstruktur

Adapter

Eingangsadapter (Sockets):

- **IN1**: A2X-Adapter (unidirectional) - erste Signalquelle (UP + DOWN)
- **IN2**: A2X-Adapter (unidirectional) - zweite Signalquelle (UP + DOWN)

Ausgangsadapter (Plug):

- **OUT**: A2X-Adapter (unidirectional) - zusammengeführtes Signal (UP + DOWN)

Funktionsweise

Der Baustein ist als Basic-FB mit einer 5-Zustands-ECC umgesetzt (START, PASS1_UP, PASS2_UP, PASS1_DOWN, PASS2_DOWN), die beide Kanäle über dieselbe ECC bedient - da pro FB-Aufruf immer nur ein Ereignis verarbeitet wird, kollidieren UP- und DOWN-Verarbeitung nie miteinander. Trifft an IN1.UP oder IN2.UP ein Ereignis ein, wird der jeweilige UP-Wert nach OUT.UP kopiert und OUT.E_UP gesendet. Symmetrisch dazu wird bei IN1.DOWN/IN2.DOWN der DOWN-Wert nach OUT.DOWN kopiert und OUT.E_DOWN gesendet. Die beiden Kanäle haben also unabhängige „Gewinner“ - IN1 kann z. B. gerade den UP-Kanal dominieren, während IN2 den DOWN-Kanal dominiert.

Technische Besonderheiten

- Zwei unabhängige Last-Writer-Wins-Merges (UP und DOWN) in einem Baustein
- Erweiterung des generischen einkanaligen LAST_2-Musters auf den 2-Kanal-A2X-Adaptertyp
- Eine gemeinsame ECC für beide Kanäle, da immer nur ein Ereignis pro Aufruf verarbeitet wird - kein Zustandskonflikt möglich
- Keine Signalverzögerung: der Wert wird im selben Zyklus durchgereicht, in dem das auslösende Ereignis eintrifft

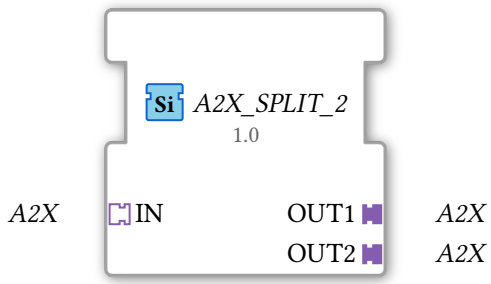
Anwendungsszenarien

- Zusammenführen zweier gleichwertiger A2X-Signalquellen (z. B. zwei Bedienstellen für eine Auf/Ab-Steuerung) zu einem einzigen nachgeschalteten Adapterpfad
- Arbitrierung zwischen zwei unabhängigen Auf/Ab-Gebern, bei denen UP und DOWN aus unterschiedlichen Quellen kommen dürfen

⚖️ Vergleich mit ähnlichen Bausteinen

A2X_LAST_2 ist die zweikanalige Variante des generischen `AX_LAST_2`-Musters, angewendet auf den A2X-Adaptertyp mit seinen zwei unabhängigen UP/DOWN-Kanälen. Für Mischbetrieb zwischen einem Daten- und einem reinen Ereignisadapter ist `AX_AE_MERGE` gedacht.

A2X_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `A2X_SPLIT_2` dient der Aufteilung eines eingehenden A2X-Adapter-Signals auf zwei identische Ausgangs-Adapter. Er ist als generischer Baustein (`GEN_A2X_SPLIT`) implementiert und ermög-

licht die Weiterleitung eines A2X-Signals an zwei unabhängige Zielbausteine ohne Signalverlust oder -veränderung.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge
Keine Daten-Eingänge vorhanden.

Daten-Ausgänge
Keine Daten-Ausgänge vorhanden.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::A2X`	`IN`	Socket	Eingangs-Adapter, der das zu verteilende A2X-Signal empfängt.
`adapter::types::unidirectional::A2X`	`OUT1`	Plug	Erster Ausgangs-Adapter, der eine identische Kopie des Eingangssignals bereitstellt.
`adapter::types::unidirectional::A2X`	`OUT2`	Plug	Zweiter Ausgangs-Adapter, der eine identische Kopie des Eingangssignals bereitstellt.

Funktionsweise
Der Baustein leitet das am Socket `IN` anliegende A2X-Adapter-Signal ohne jegliche Transformation oder Verzögerung direkt an die beiden Plugs `OUT1` und `OUT2` weiter. Jedes Ereignis, das über den A2X-Adapter übertragen wird, erscheint gleichzeitig und unverändert an beiden Ausgängen. Der Baustein besitzt keine interne Logik, keinen Zustand und keine Latenz.

Technische Besonderheiten
- **Generischer Baustein**: Der FB ist als generischer Typ `GEN_A2X_SPLIT` deklariert und nutzt das Eclipse 4diac Generic-Class-Konzept. Dies ermöglicht eine flexible Wiederverwendung in verschiedenen Projekten.
- **Reine Adapterweiterleitung**: Es werden keine Ereignis- oder Daten-Ein-/Ausgänge verwendet; die gesamte Kommunikation erfolgt ausschließlich über die A2X-Adapter-Schnittstellen.
- **Unidirektional**: Der A2X-Adapter ist unidirektional, daher fließen Daten und Ereignisse nur vom Socket zu den Plugs (Eingang → Ausgang).

Zustandsübersicht
Der Baustein besitzt keinen internen Zustandsautomaten und ist vollständig **stateless**. Jeder Eingangswert wird sofort und ohne Speicherung an beide Ausgänge weitergegeben.

Anwendungsszenarien
- **Signalvervielfachung**: Ein von einem Sensor oder einer Steuerung bereitgestelltes A2X-Signal soll an zwei verschiedene Verbraucher (z. B. Aktoren, Anzeigen, überlagerte Logiken) gleichzeitig verteilt werden.
- **Redundanz oder Überwachung**: Ein Signal soll parallel zu einem Hauptziel auch an einen Diagnose- oder Monitoring-Baustein geführt werden.
- **Test und Simulation**: Ein A2X-Signal soll während des Tests sowohl an die reale Hardware als auch an einen Simulator angeschlossen werden.

Vergleich mit ähnlichen Bausteinen

- **A2X_SPLIT_3**: Analoger Baustein mit drei Ausgängen statt zwei. Beide Bausteine arbeiten identisch, unterscheiden sich nur in der Anzahl der Ausgangsplugs.
- **A2X_MERGE**: Führt zwei A2X-Signale zu einem zusammen – gegensätzliche Funktionalität.
- **Generische Splitter (z. B. für andere Adaptertypen)**: Funktionsprinzip gleich, aber Adaptertyp angepasst.
- `A2X_SPLIT_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

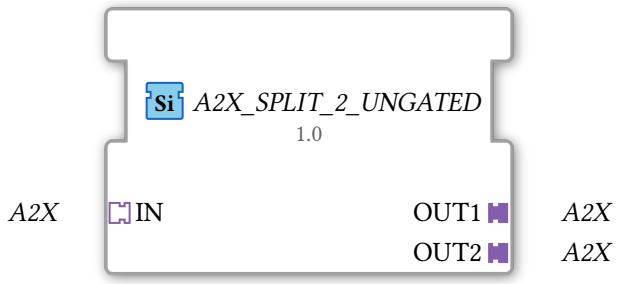
Der `A2X_SPLIT_2` ist ein einfacher, aber nützlicher Baustein zur Signalverteilung in 4diac-Anwendungen. Seine generische Natur ermöglicht eine flexible Nutzung ohne zusätzliche Konfiguration. Er eignet sich besonders für Szenarien, in denen ein A2X-Signal an mehrere Empfänger weitergegeben werden muss, ohne dass die Signalintegrität beeinträchtigt wird.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

A2X_SPLIT_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `A2X_SPLIT_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `A2X_SPLIT_2_UNGATED` dient der Aufteilung eines eingehenden A2X-Adapter-Signals auf zwei identische Ausgangs-Adapter. Er ist als generischer Baustein (`GEN_A2X_SPLIT`) implementiert und ermöglicht die Weiterleitung eines A2X-Signals an zwei unabhängige Zielbausteine ohne Signalverlust oder -veränderung.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::A2X`	`IN`	Socket	Eingangs-Adapter, der das zu verteilende A2X-Signal empfängt.
`adapter::types::unidirectional::A2X`	`OUT1`	Plug	Erster Ausgangs-Adapter, der eine identische Kopie des Eingangssignals bereitstellt.
`adapter::types::unidirectional::A2X`	`OUT2`	Plug	Zweiter Ausgangs-Adapter, der eine identische Kopie des Eingangssignals bereitstellt.

Funktionsweise

Der Baustein leitet das am Socket `IN` anliegende A2X-Adapter-Signal ohne jegliche Transformation oder Verzögerung direkt an die beiden Plugs `OUT1` und `OUT2` weiter. Jedes Ereignis, das über den A2X-Adapter übertragen wird, erscheint gleichzeitig und unverändert an beiden Ausgängen. Der Baustein besitzt keine interne Logik, keinen Zustand und keine Latenz.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ `GEN_A2X_SPLIT` deklariert und nutzt das Eclipse 4diac Generic-Class-Konzept. Dies ermöglicht eine flexible Wiederverwendung in verschiedenen Projekten.
- **Reine Adapterweiterleitung**: Es werden keine Ereignis- oder Daten-Ein-/Ausgänge verwendet; die gesamte Kommunikation erfolgt ausschließlich über die A2X-Adapter-Schnittstellen.
- **Unidirektional**: Der A2X-Adapter ist unidirektional, daher fließen Daten und Ereignisse nur vom Socket zu den Plugs (Eingang → Ausgang).

Zustandsübersicht

Der Baustein besitzt keinen internen Zustandsautomaten und ist vollständig **stateless**. Jeder Eingangswert wird sofort und ohne Speicherung an beide Ausgänge weitergegeben.

Anwendungsszenarien

- **Signalvervielfachung**: Ein von einem Sensor oder einer Steuerung bereitgestelltes A2X-Signal soll an zwei verschiedene Verbraucher (z. B. Aktoren, Anzeigen, überlagerte Logiken) gleichzeitig verteilt werden.
- **Redundanz oder Überwachung**: Ein Signal soll parallel zu einem Hauptziel auch an einen Diagnose- oder Monitoring-Baustein geführt werden.
- **Test und Simulation**: Ein A2X-Signal soll während des Tests sowohl an die reale Hardware als auch an einen Simulator angeschlossen werden.

Vergleich mit ähnlichen Bausteinen

- **A2X_SPLIT_3**: Analoger Baustein mit drei Ausgängen statt zwei. Beide Bausteine arbeiten identisch, unterscheiden sich nur in der Anzahl der Ausgangsplugs.
- **A2X_MERGE**: Führt zwei A2X-Signale zu einem zusammen – gegensätzliche Funktionalität.
- **Generische Splitter (z. B. für andere Adaptertypen)**: Funktionsprinzip gleich, aber Adaptertyp angepasst.
- `A2X_SPLIT_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

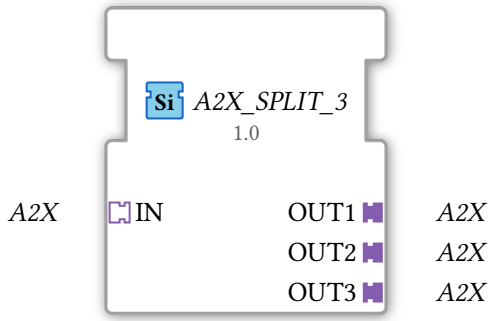
Fazit

Der `A2X_SPLIT_2_UNGATED` ist ein einfacher, aber nützlicher Baustein zur Signalverteilung in 4diac-Anwendungen. Seine generische Natur ermöglicht eine flexible Nutzung ohne zusätzliche Konfiguration. Er eignet sich besonders für Szenarien, in denen ein A2X-Signal an mehrere Empfänger weitergegeben werden muss, ohne dass die Signalintegrität beeinträchtigt wird.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

A2X_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **A2X_SPLIT_3** dient zur Aufteilung eines eingehenden A2X-Adapterdatenstroms auf drei identische Ausgänge. Er ist als generischer Baustein (generic FB) implementiert und eignet sich für Anwendungen, bei denen ein Signal mehrfach benötigt wird.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::A2X`	**IN**	Socket (Eingang)	Quell-Adapter, dessen Datenstrom aufgeteilt wird.
`adapter::types::unidirectional::A2X`	**OUT1**	Plug (Ausgang)	Erster identischer Ausgang.
`adapter::types::unidirectional::A2X`	**OUT2**	Plug (Ausgang)	Zweiter identischer Ausgang.
`adapter::types::unidirectional::A2X`	**OUT3**	Plug (Ausgang)	Dritter identischer Ausgang.

Die Adapter sind alle vom Typ **A2X** (unidirectional) und übertragen Daten in einer Richtung. Der Baustein ist passiv, d.h. er benötigt keine Ereignissteuerung für die Datenweitergabe.

Funktionsweise

Der Baustein nimmt den über den Socket **IN** ankommenden A2X-Datenstrom entgegen und leitet ihn unverändert an die drei Plugs **OUT1**, **OUT2** und **OUT3** weiter. Es findet keine Verarbeitung, Filterung oder Pufferung statt – der Datenstrom wird 1:1 auf alle Ausgänge verteilt.

Da es sich um einen generischen Baustein handelt, kann der konkrete Datentyp des A2X-Adapters zur Laufzeit durch eine Typisierung (z.B. über das Attribut `GenericClassName`) festgelegt werden.

Technische Besonderheiten

- **Generische Auslegung**: Der Baustein ist als generischer FB deklariert (`GenericClassName = 'GEN_A2X_SPLIT'`), sodass der A2X-Adapter je nach Anwendung mit unterschiedlichen konkreten Datenstrukturen parametriert werden kann.
- **Keine Zustandsautomatik**: Der Baustein besitzt keine ECC (Execution Control Chart) und keine Ereignis-Schnittstellen. Er arbeitet rein datenflussorientiert.
- **Laufzeit-Typanpassung**: Über das Attribut `TypeHash` kann eine eindeutige Typkennzeichnung für die generische Instanz hinterlegt werden (hier leer, muss projektspezifisch gesetzt werden).

Zustandsübersicht

Der Baustein besitzt keine internen Zustände, da er keine sequentielle Logik oder Zustandsmaschine enthält. Die Datenweitergabe erfolgt kontinuierlich ohne Verzögerung.

Anwendungsszenarien

- **Signalverteilung**: Ein von einem Sensor oder einer Steuerung kommendes A2X-Signal muss parallel an mehrere nachgelagerte Bausteine (z.B. Anzeigen, Logger, Aktoren) weitergeleitet werden.
- **Redundanz / Monitoring**: Ein Datenstrom soll sowohl an die eigentliche Verarbeitung als auch an ein Diagnose- oder Überwachungssystem gesendet werden.
- **Prototypische Aufbauten**: In der Entwicklungsphase kann ein einzelner generischer Splitter flexibel für verschiedene Adaptertypen verwendet werden.

Vergleich mit ähnlichen Bausteinen

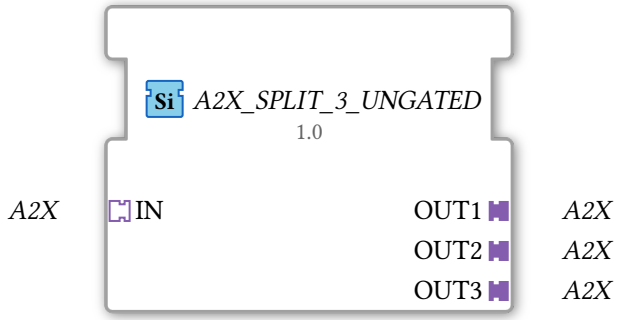
- **A2X_SPLIT_2**: Teilt einen A2X-Stream auf zwei Ausgänge auf. Der vorliegende Baustein erweitert dies auf drei Ausgänge.
- **Manuelles Duplizieren**: Ohne Splitter müsste der Architekt den Quell-Adapter mehrfach in der Konfiguration referenzieren, was die Lesbarkeit und Wartbarkeit verringert.
- **Generische Splitter**: Andere Splitting-Bausteine (z.B. für Daten- oder Ereignisadapter) folgen dem gleichen Prinzip, sind aber auf andere Adaptertypen spezialisiert.
- `A2X_SPLIT_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der **A2X_SPLIT_3** ist ein einfacher, aber nützlicher generischer Funktionsbaustein zur Vervielfältigung eines A2X-Datenstroms auf drei Pfade. Durch seinen generischen Charakter passt er sich flexibel an unterschiedliche Datenstrukturen an und erleichtert die modulare Verschaltung in IEC 61499-basierten Automatisierungssystemen.

A2X_SPLIT_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `A2X_SPLIT_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **A2X_SPLIT_3_UNGATED** dient zur Aufteilung eines eingehenden A2X-Adapterdatenstroms auf drei identische Ausgänge. Er ist als generischer Baustein (generic FB) implementiert und eignet sich für Anwendungen, bei denen ein Signal mehrfach benötigt wird.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::A2X`	**IN**	Socket (Eingang)	Quell-Adapter, dessen Datenstrom aufgeteilt wird.
`adapter::types::unidirectional::A2X`	**OUT1**	Plug (Ausgang)	Erster identischer Ausgang.
`adapter::types::unidirectional::A2X`	**OUT2**	Plug (Ausgang)	Zweiter identischer Ausgang.
`adapter::types::unidirectional::A2X`	**OUT3**	Plug (Ausgang)	Dritter identischer Ausgang.

Die Adapter sind alle vom Typ **A2X** (unidirectional) und übertragen Daten in einer Richtung. Der Baustein ist passiv, d.h. er benötigt keine Ereignissteuerung für die Datenweitergabe.

Funktionsweise

Der Baustein nimmt den über den Socket **IN** ankommenden A2X-Datenstrom entgegen und leitet ihn unverändert an die drei Plugs **OUT1**, **OUT2** und **OUT3** weiter. Es findet keine Verarbeitung, Filterung oder Pufferung statt – der Datenstrom wird 1:1 auf alle Ausgänge verteilt.

Da es sich um einen generischen Baustein handelt, kann der konkrete Datentyp des A2X-Adapters zur Laufzeit durch eine Typisierung (z.B. über das Attribut `GenericClassName`) festgelegt werden.

Technische Besonderheiten

- **Generische Auslegung**: Der Baustein ist als generischer FB deklariert (`GenericClassName = 'GEN_A2X_SPLIT'`), sodass der A2X-Adapter je nach Anwendung mit unterschiedlichen konkreten Datenstrukturen parametriert werden kann.
- **Keine Zustandsautomatik**: Der Baustein besitzt keine ECC (Execution Control Chart) und keine Ereignis-Schnittstellen. Er arbeitet rein datenflussorientiert.

- **Laufzeit-Typanpassung**: Über das Attribut `TypeHash` kann eine eindeutige Typkennzeichnung für die generische Instanz hinterlegt werden (hier leer, muss projektspezifisch gesetzt werden).

Zustandsübersicht

Der Baustein besitzt keine internen Zustände, da er keine sequentielle Logik oder Zustandsmaschine enthält. Die Datenweitergabe erfolgt kontinuierlich ohne Verzögerung.

Anwendungsszenarien

- **Signalverteilung**: Ein von einem Sensor oder einer Steuerung kommendes A2X-Signal muss parallel an mehrere nachgelagerte Bausteine (z.B. Anzeigen, Logger, Aktoren) weitergeleitet werden.
- **Redundanz / Monitoring**: Ein Datenstrom soll sowohl an die eigentliche Verarbeitung als auch an ein Diagnose- oder Überwachungssystem gesendet werden.
- **Prototypische Aufbauten**: In der Entwicklungsphase kann ein einzelner generischer Splitter flexibel für verschiedene Adaptertypen verwendet werden.

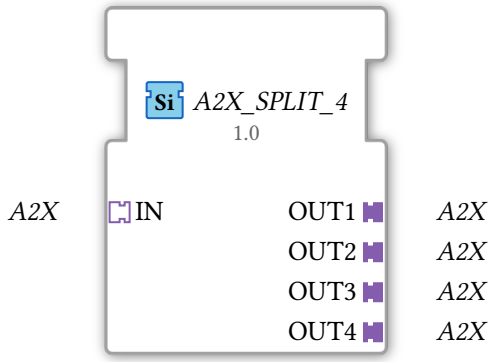
Vergleich mit ähnlichen Bausteinen

- **A2X_SPLIT_2**: Teilt einen A2X-Stream auf zwei Ausgänge auf. Der vorliegende Baustein erweitert dies auf drei Ausgänge.
- **Manuelles Duplizieren**: Ohne Splitter müsste der Architekt den Quell-Adapter mehrfach in der Konfiguration referenzieren, was die Lesbarkeit und Wartbarkeit verringert.
- **Generische Splitter**: Andere Splitting-Bausteine (z.B. für Daten- oder Ereignisadapter) folgen dem gleichen Prinzip, sind aber auf andere Adaptertypen spezialisiert.
- `A2X_SPLIT_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der **A2X_SPLIT_3_UNGATED** ist ein einfacher, aber nützlicher generischer Funktionsbaustein zur Vervielfältigung eines A2X-Datenstroms auf drei Pfade. Durch seinen generischen Charakter passt er sich flexibel an unterschiedliche Datenstrukturen an und erleichtert die modulare Verschaltung in IEC 61499-basierten Automatisierungssystemen.

A2X_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **A2X_SPLIT_4** dient der Aufteilung eines eingehenden A2X-Adaptersignals auf vier identische A2X-Ausgänge. Er ist als generischer Baustein ausgelegt und ermöglicht so eine flexible Verwendung in unterschiedlichsten Automatisierungsumgebungen, in denen ein A2X-Signal mehrfach benötigt wird.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Bezeichner	Typ	Beschreibung
Socket (Eingang)	IN	`adapter::types::unidirectional::A2X`	Ein eingehendes A2X-Signal (unidirektional)
Plug (Ausgang 1)	OUT1	`adapter::types::unidirectional::A2X`	Erster ausgehender A2X-Kanal
Plug (Ausgang 2)	OUT2	`adapter::types::unidirectional::A2X`	Zweiter ausgehender A2X-Kanal
Plug (Ausgang 3)	OUT3	`adapter::types::unidirectional::A2X`	Dritter ausgehender A2X-Kanal
Plug (Ausgang 4)	OUT4	`adapter::types::unidirectional::A2X`	Vierter ausgehender A2X-Kanal

Funktionsweise

Der Baustein arbeitet als reiner Signalverteiler. Das am Socket **IN** anliegende A2X-Adapter-Interface wird intern ohne weitere Verarbeitung oder Verzögerung auf die vier Ausgangs-Plugs **OUT1** bis **OUT4** durchgeschaltet. Jeder Ausgang liefert exakt dasselbe Signal wie der Eingang. Da es sich um unidirektionale Adapter handelt, erfolgt keine Rückkopplung von den Ausgängen zum Eingang.

Technische Besonderheiten

- **Generischer Baustein** – Der FB wird als `GEN_A2X_SPLIT` instanziiert und kann durch Typparameter an unterschiedliche A2X-Varianten angepasst werden.
- **Keine Ereignis- oder Datenverarbeitung** – Die gesamte Funktionalität beschränkt sich auf die Weiterleitung des Adaptersignals. Es existieren weder Ereignisse noch Daten-Ein-/Ausgänge.
- **Lizenzierung** – Der Baustein steht unter der Eclipse Public License 2.0 (EPL-2.0).

Zustandsübersicht

Der FB besitzt keine Zustandsautomaten (ECC) und kein internes Verhalten, das zeitlich oder zustandsabhängig wäre. Die Ausgangssignale folgen unmittelbar dem Eingangssignal.

Anwendungsszenarien

- **Verteilung eines A2X-Signals** an mehrere nachgelagerte Funktionsbausteine, die auf dieselben A2X-Informationen angewiesen sind.
- **Sternförmige Topologien** in der Kommunikation zwischen Komponenten eines verteilten Steuerungssystems.
- **Test- und Simulationsumgebungen**, bei denen ein A2X-Signal parallel aufgezeichnet oder analysiert werden soll.

Vergleich mit ähnlichen Bausteinen

- **A2X_SPLIT_2** / **A2X_SPLIT_3** / **A2X_SPLIT_N** – Diese Bausteine bieten dieselbe Funktionalität mit einer anderen Anzahl von Ausgängen.
- Im Gegensatz zu einem **Multiplexer** oder **Demultiplexer** findet hier keine Selektion statt – das Signal wird immer auf alle Ausgänge dupliziert.
- `A2X_SPLIT_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

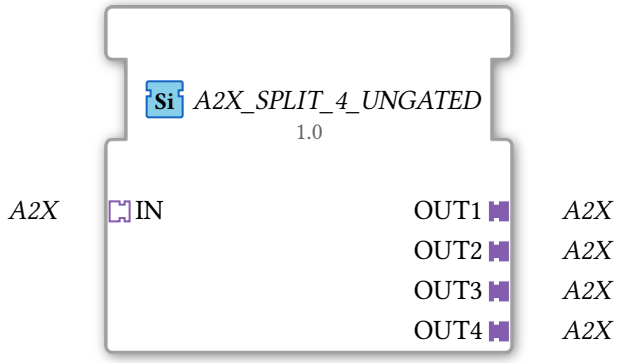
Der **A2X_SPLIT_4** ist ein einfacher, aber nützlicher Baustein zur Vervielfältigung eines unidirektionalen A2X-Adaptersignals. Durch seine generische Auslegung und die klare Schnittstelle lässt er sich leicht in bestehende 4diac-IDE-Projekte integrieren und trägt zur Modularisierung von Steuerungsapplikationen bei.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

A2X_SPLIT_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `A2X_SPLIT_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **A2X_SPLIT_4_UNGATED** dient der Aufteilung eines eingehenden A2X-Adaptersignals auf vier identische A2X-Ausgänge. Er ist als generischer Baustein ausgelegt und ermöglicht so eine flexible Verwendung in unterschiedlichsten Automatisierungsumgebungen, in denen ein A2X-Signal mehrfach benötigt wird.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Bezeichner	Typ	Beschreibung

Socket (Eingang)	IN	`adapter::types::unidirectional::A2X`	Ein eingehendes A2X-Signal (unidirektional)
Plug (Ausgang 1)	OUT1	`adapter::types::unidirectional::A2X`	Erster ausgehender A2X-Kanal
Plug (Ausgang 2)	OUT2	`adapter::types::unidirectional::A2X`	Zweiter ausgehender A2X-Kanal
Plug (Ausgang 3)	OUT3	`adapter::types::unidirectional::A2X`	Dritter ausgehender A2X-Kanal
Plug (Ausgang 4)	OUT4	`adapter::types::unidirectional::A2X`	Vierter ausgehender A2X-Kanal

Funktionsweise

Der Baustein arbeitet als reiner Signalverteiler. Das am Socket **IN** anliegende A2X-Adapter-Interface wird intern ohne weitere Verarbeitung oder Verzögerung auf die vier Ausgangs-Plugs **OUT1** bis **OUT4** durchgeschaltet. Jeder Ausgang liefert exakt dasselbe Signal wie der Eingang. Da es sich um unidirektionale Adapter handelt, erfolgt keine Rückkopplung von den Ausgängen zum Eingang.

Technische Besonderheiten

- **Generischer Baustein** – Der FB wird als `GEN_A2X_SPLIT` instanziiert und kann durch Typparameter an unterschiedliche A2X-Varianten angepasst werden.
- **Keine Ereignis- oder Datenverarbeitung** – Die gesamte Funktionalität beschränkt sich auf die Weiterleitung des Adaptersignals. Es existieren weder Ereignisse noch Daten-Ein-/Ausgänge.
- **Lizenzierung** – Der Baustein steht unter der Eclipse Public License 2.0 (EPL-2.0).

Zustandsübersicht

Der FB besitzt keine Zustandsautomaten (ECC) und kein internes Verhalten, das zeitlich oder zustandsabhängig wäre. Die Ausgangssignale folgen unmittelbar dem Eingangssignal.

Anwendungsszenarien

- **Verteilung eines A2X-Signals** an mehrere nachgelagerte Funktionsbausteine, die auf dieselben A2X-Informationen angewiesen sind.
- **Sternförmige Topologien** in der Kommunikation zwischen Komponenten eines verteilten Steuerungssystems.
- **Test- und Simulationsumgebungen**, bei denen ein A2X-Signal parallel aufgezeichnet oder analysiert werden soll.

Vergleich mit ähnlichen Bausteinen

- **A2X_SPLIT_2** / **A2X_SPLIT_3** / **A2X_SPLIT_N** – Diese Bausteine bieten dieselbe Funktionalität mit einer anderen Anzahl von Ausgängen.
- Im Gegensatz zu einem **Multiplexer** oder **Demultiplexer** findet hier keine Selektion statt – das Signal wird immer auf alle Ausgänge dupliziert.
- `A2X_SPLIT_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

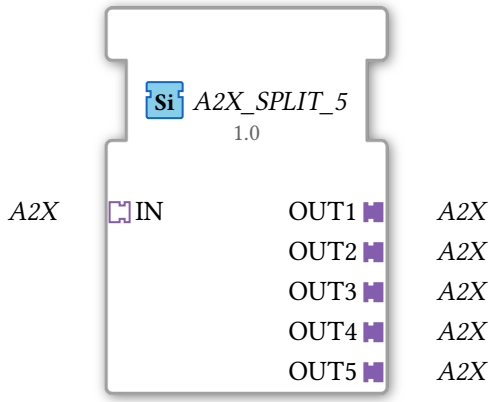
Fazit

Der **A2X_SPLIT_4_UNGATED** ist ein einfacher, aber nützlicher Baustein zur Vervielfältigung eines unidirektionalen A2X-Adaptersignals. Durch seine generische Auslegung und die klare Schnittstelle lässt er sich leicht in bestehende 4diac-IDE-Projekte integrieren und trägt zur Modularisierung von Steuerungsapplikationen bei.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

A2X_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **A2X_SPLIT_5** dient dazu, einen eingehenden A2X-Adapter in fünf identische A2X-Ausgänge aufzuteilen. Er wird als generischer Baustein bereitgestellt, der über Attribute konfiguriert werden kann.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	IN	A2X (unidirectional)	Eingangsadapter (wird aufgeteilt)
Plug	OUT1	A2X (unidirectional)	Erster Ausgangsadapter
Plug	OUT2	A2X (unidirectional)	Zweiter Ausgangsadapter
Plug	OUT3	A2X (unidirectional)	Dritter Ausgangsadapter
Plug	OUT4	A2X (unidirectional)	Vierter Ausgangsadapter
Plug	OUT5	A2X (unidirectional)	Fünfter Ausgangsadapter

Funktionsweise

Der Baustein leitet den am Socket `IN` anliegenden A2X-Adapter unverändert und parallel an alle fünf Plugs (`OUT1` bis `OUT5`) weiter. Eine Verzögerung oder Pufferung findet nicht statt – jede Änderung am Eingang wird sofort an alle Ausgänge propagiert. Somit verhält sich der Baustein wie ein einfacher Verteiler (Splitter) für A2X-Schnittstellen.

Technische Besonderheiten

- **Generischer Baustein:** Die tatsächliche Instanz kann über die Attribute `eclipse4diac::core::GenericClassName` und `eclipse4diac::core::TypeHash` konfiguriert werden, wodurch eine Anpassung an verschiedene A2X-Varianten möglich ist.
- **Keine Ereignis- oder Daten-Ports:** Die gesamte Kommunikation erfolgt ausschließlich über Adapter. Dies vereinfacht die Kopplung mit anderen A2X-basierten Komponenten.
- **Skalierbarkeit:** Der Baustein kann als Grundlage für Splitter mit anderer Anzahl von Ausgängen dienen (z. B. `A2X_SPLIT_2`, `A2X_SPLIT_10`).

Zustandsübersicht

Der Baustein besitzt keine internen Zustände oder Zustandsautomaten. Er arbeitet rein kombinatorisch als passive Weiterleitung.

Anwendungsszenarien

- **Verteilen eines Bus-Signals:** Wenn ein A2X-Adapter (z. B. für einen Feldbus) an mehrere nachgelagerte Module übergeben werden muss, ohne dass diese sich gegenseitig beeinflussen.
- **Test- und Simulationsaufbauten:** Ein eingehender A2X-Datenstrom kann parallel an verschiedene Analyse- oder Überwachungskomponenten geschaltet werden.
- **Erweiterung von Steuerungssystemen:** Ein Sensor- oder Aktor-Adapter kann auf mehrere Funktionsblöcke aufgeteilt werden, die unterschiedliche Logiken implementieren.

Vergleich mit ähnlichen Bausteinen

- **A2X_SPLIT_2 / A2X_SPLIT_N:** Diese Varianten unterscheiden sich nur in der Anzahl der Ausgänge. `A2X_SPLIT_5` bietet einen festen 5-fachen Split.
- **A2X_MERGE:** Während der Splitter einen Eingang auf viele Ausgänge verteilt, fasst der Merge-Baustein mehrere A2X-Eingänge zu einem Ausgang zusammen.
- **Datensplitter (z. B. SPLIT_INT):** Letztere arbeiten auf Daten-Ebene (z. B. ganze Zahlen) und nicht auf Adapterebene. Der hier beschriebene Baustein operiert auf der Adapterebene, was eine strukturelle Entkopplung ermöglicht.
- `A2X_SPLIT_5_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

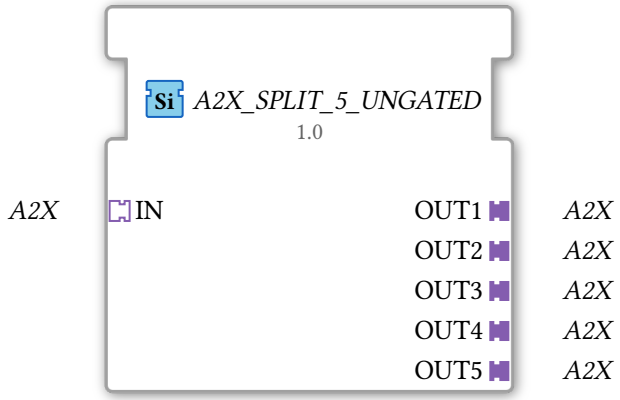
Der `A2X_SPLIT_5` ist ein einfacher, aber nützlicher Adapter-Splitter für A2X-Schnittstellen. Durch seine generische Natur und die reine Adapter-Kommunikation fügt er sich nahtlos in IEC 61499-basierte Steuerungssysteme ein. Er eignet sich besonders für Anwendungen, bei denen ein A2X-Signal auf mehrere Pfade verteilt werden muss, ohne zusätzliche Logik oder Zustände zu erfordern.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

A2X_SPLIT_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `A2X_SPLIT_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **A2X_SPLIT_5_UNGATED** dient dazu, einen eingehenden A2X-Adapter in fünf identische A2X-Ausgänge aufzuteilen. Er wird als generischer Baustein bereitgestellt, der über Attribute konfiguriert werden kann.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	IN	A2X (unidirectional)	Eingangsadapter (wird aufgeteilt)
Plug	OUT1	A2X (unidirectional)	Erster Ausgangsadapter
Plug	OUT2	A2X (unidirectional)	Zweiter Ausgangsadapter
Plug	OUT3	A2X (unidirectional)	Dritter Ausgangsadapter
Plug	OUT4	A2X (unidirectional)	Vierter Ausgangsadapter
Plug	OUT5	A2X (unidirectional)	Fünfter Ausgangsadapter

Funktionsweise

Der Baustein leitet den am Socket `IN` anliegenden A2X-Adapter unverändert und parallel an alle fünf Plugs (`OUT1` bis `OUT5`) weiter. Eine Verzögerung oder Pufferung findet nicht statt – jede Änderung am Eingang wird sofort an alle Ausgänge propagiert. Somit verhält sich der Baustein wie ein einfacher Verteiler (Splitter) für A2X-Schnittstellen.

Technische Besonderheiten

- **Generischer Baustein:** Die tatsächliche Instanz kann über die Attribute `eclipse4diac::core::GenericClassName` und `eclipse4diac::core::TypeHash` konfiguriert werden, wodurch eine Anpassung an verschiedene A2X-Varianten möglich ist.

- **Keine Ereignis- oder Daten-Ports:** Die gesamte Kommunikation erfolgt ausschließlich über Adapter. Dies vereinfacht die Kopplung mit anderen A2X-basierten Komponenten.
- **Skalierbarkeit:** Der Baustein kann als Grundlage für Splitter mit anderer Anzahl von Ausgängen dienen (z. B. `A2X_SPLIT_2`, `A2X_SPLIT_10`).

Zustandsübersicht

Der Baustein besitzt keine internen Zustände oder Zustandsautomaten. Er arbeitet rein kombinatorisch als passive Weiterleitung.

Anwendungsszenarien

- **Verteilen eines Bus-Signals:** Wenn ein A2X-Adapter (z. B. für einen Feldbus) an mehrere nachgelagerte Module übergeben werden muss, ohne dass diese sich gegenseitig beeinflussen.
- **Test- und Simulationsaufbauten:** Ein eingehender A2X-Datenstrom kann parallel an verschiedene Analyse- oder Überwachungskomponenten geschaltet werden.
- **Erweiterung von Steuerungssystemen:** Ein Sensor- oder Aktor-Adapter kann auf mehrere Funktionsblöcke aufgeteilt werden, die unterschiedliche Logiken implementieren.

Vergleich mit ähnlichen Bausteinen

- **A2X_SPLIT_2 / A2X_SPLIT_N:** Diese Varianten unterscheiden sich nur in der Anzahl der Ausgänge. `A2X_SPLIT_5_UNGATED` bietet einen festen 5-fachen Split.
- **A2X_MERGE:** Während der Splitter einen Eingang auf viele Ausgänge verteilt, fasst der Merge-Baustein mehrere A2X-Eingänge zu einem Ausgang zusammen.
- **Datensplitter (z. B. SPLIT_INT):** Letztere arbeiten auf Daten-Ebene (z. B. ganze Zahlen) und nicht auf Adapterebene. Der hier beschriebene Baustein operiert auf der Adapterebene, was eine strukturelle Entkopplung ermöglicht.
- `A2X_SPLIT_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `A2X_SPLIT_5_UNGATED` ist ein einfacher, aber nützlicher Adapter-Splitter für A2X-Schnittstellen. Durch seine generische Natur und die reine Adapter-Kommunikation fügt er sich nahtlos in IEC 61499-basierte Steuerungssysteme ein. Er eignet sich besonders für Anwendungen, bei denen ein A2X-Signal auf mehrere Pfade verteilt werden muss, ohne zusätzliche Logik oder Zustände zu erfordern.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

A2X_SPLIT_6

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `A2X_SPLIT_6` dient dazu, einen eingehenden A2X-Adapter auf sechs identische A2X-Ausgänge zu verteilen. Er ist als generischer Baustein (Generic FB) realisiert und ermöglicht die Weiterleitung von Adapterdaten ohne Verzögerung oder Zustandsänderung. Typische Einsatzbereiche sind die parallele Versorgung mehrerer Verbraucher oder die Signalkaskadierung in industriellen Steuerungen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Eingang (Socket)	`IN`	`adapter::types::unidirectional::A2X`	Empfängt das eingehende A2X-Adapter-Signal
Ausgang (Plug) 1	`OUT1`	`adapter::types::unidirectional::A2X`	Erster ausgehender A2X-Adapter
Ausgang (Plug) 2	`OUT2`	`adapter::types::unidirectional::A2X`	Zweiter ausgehender A2X-Adapter
Ausgang (Plug) 3	`OUT3`	`adapter::types::unidirectional::A2X`	Dritter ausgehender A2X-Adapter
Ausgang (Plug) 4	`OUT4`	`adapter::types::unidirectional::A2X`	Vierter ausgehender A2X-Adapter
Ausgang (Plug) 5	`OUT5`	`adapter::types::unidirectional::A2X`	Fünfter ausgehender A2X-Adapter
Ausgang (Plug) 6	`OUT6`	`adapter::types::unidirectional::A2X`	Sechster ausgehender A2X-Adapter

Funktionsweise

Der Baustein arbeitet als reiner Verteiler (Splitter). Sobald der Socket `IN` mit einem A2X-Adapter verbunden ist, wird das komplette Adaptersignal – bestehend aus allen in diesem Adapter enthaltenen Daten- und Ereigniskanälen – an alle sechs Plugs (`OUT1` … `OUT6`) weitergegeben. Die Weitergabe erfolgt ohne Pufferung oder eigene Logik; jede Änderung am Eingang spiegelt sich unverzögert auf allen Ausgängen wider.

Technische Besonderheiten

- **Unidirektionalität** – Der Adapter `A2X` ist unidirektional ausgelegt; eine Datenflussumkehr ist nicht vorgesehen.
- **Generische Auslegung** – Der FB ist als generischer Typ (`GenericClassName = 'GEN_A2X_SPLIT'`) hinterlegt, sodass er in verschiedenen Kontexten eingesetzt werden kann.
- **Keine Zustände oder Ereignisse** – Es existieren keine internen Zustandsautomaten oder ereignisgesteuerten Abläufe; der FB ist rein kombinatorisch.
- **Vollständige Entkopplung** – Durch die Verwendung von Adaptern werden die angeschlossenen Bausteine entkoppelt, was die Wiederverwendbarkeit erhöht.

Zustandsübersicht

Der Baustein besitzt keinen eigenen Zustandsautomaten. Er ist jederzeit passiv und aktiv nur durch die am Eingang anliegenden Adapterdaten. Daher existieren keine definierten Zustände wie INIT, RUN oder IDLE.

Anwendungsszenarien

- **Signalverteilung in der Automatisierung** – Ein Sensor liefert über einen A2X-Adapter Messwerte, die parallel an mehrere Steuerungen oder Auswerteinheiten weitergeleitet werden müssen.
- **Kaskadierung von Adaptersignalen** – In komplexen Hierarchien kann ein A2X-Signal auf mehrere untergeordnete Module aufgeteilt werden.
- **Test- und Simulationsumgebungen** – Zum gleichzeitigen Anschluss mehrerer Prüfstände oder Visualisierungskomponenten an eine Datenquelle.

Vergleich mit ähnlichen Bausteinen

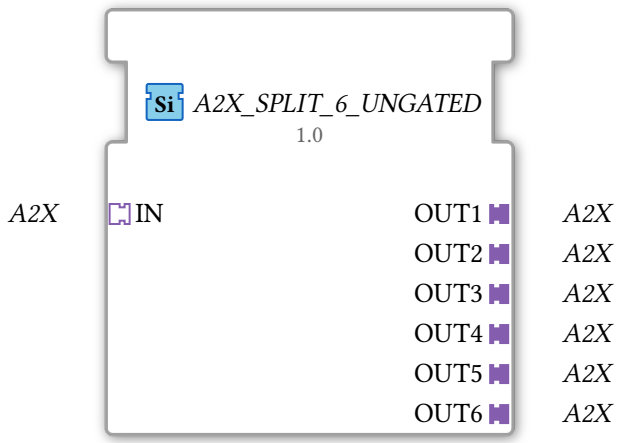
- **A2X_SPLIT_2 / A2X_SPLIT_4 / A2X_SPLIT_N** – Diese Bausteine bieten die gleiche Funktionalität, jedoch mit zwei, vier oder einer konfigurierbaren Anzahl von Ausgängen. `A2X_SPLIT_6` stellt eine feste 6-fach-Verteilung bereit.
- **A2X_MERGE** – Im Gegensatz zum Splitter fasst dieser Baustein mehrere A2X-Eingänge zu einem Ausgang zusammen.
- **Standard-Daten-Splitter** (z. B. für einfache Datentypen) – Adapterbasierte Splitter wie dieser arbeiten auf einer höheren Abstraktionsebene und kapseln komplexe Signalbündel.
- `A2X_SPLIT_6_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `A2X_SPLIT_6` ist ein einfacher, aber nützlicher Funktionsblock zur Vervielfachung von A2X-Adapter-Signalen. Durch seinen generischen Aufbau und die reine Verteilerfunktion eignet er sich besonders für Szenarien, in denen ein Signal ohne Verarbeitungslogik an mehrere Empfänger gesendet werden muss. Die Abwesenheit von Ereignissen und Zuständen macht ihn leicht verständlich und ressourcenschonend einsetzbar.

A2X_SPLIT_6_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `A2X_SPLIT_6`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `A2X_SPLIT_6_UNGATED` dient dazu, einen eingehenden A2X-Adapter auf sechs identische A2X-Ausgänge zu verteilen. Er ist als generischer Baustein (Generic FB) realisiert und ermöglicht die Weiterleitung von Adapterdaten ohne Verzögerung oder Zustandsänderung. Typische Einsatzbereiche

sind die parallele Versorgung mehrerer Verbraucher oder die Signalkaskadierung in industriellen Steuerungen.

Schnittstellenstruktur

Ereignis-Eingänge
Keine.

Ereignis-Ausgänge
Keine.

Daten-Eingänge
Keine.

Daten-Ausgänge
Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Eingang (Socket)	`IN`	`adapter::types::unidirectional::A2X`	Empfängt das eingehende A2X-Adapter-Signal
Ausgang (Plug) 1	`OUT1`	`adapter::types::unidirectional::A2X`	Erster ausgehender A2X-Adapter
Ausgang (Plug) 2	`OUT2`	`adapter::types::unidirectional::A2X`	Zweiter ausgehender A2X-Adapter
Ausgang (Plug) 3	`OUT3`	`adapter::types::unidirectional::A2X`	Dritter ausgehender A2X-Adapter
Ausgang (Plug) 4	`OUT4`	`adapter::types::unidirectional::A2X`	Vierter ausgehender A2X-Adapter
Ausgang (Plug) 5	`OUT5`	`adapter::types::unidirectional::A2X`	Fünfter ausgehender A2X-Adapter
Ausgang (Plug) 6	`OUT6`	`adapter::types::unidirectional::A2X`	Sechster ausgehender A2X-Adapter

Funktionsweise
Der Baustein arbeitet als reiner Verteiler (Splitter). Sobald der Socket `IN` mit einem A2X-Adapter verbunden ist, wird das komplette Adaptersignal – bestehend aus allen in diesem Adapter enthaltenen Daten- und Ereigniskanälen – an alle sechs Plugs (`OUT1` … `OUT6`) weitergegeben. Die Weitergabe erfolgt ohne Pufferung oder eigene Logik; jede Änderung am Eingang spiegelt sich unverzögert auf allen Ausgängen wider.

Technische Besonderheiten
- **Unidirektionalität** – Der Adapter `A2X` ist unidirektional ausgelegt; eine Datenflussumkehr ist nicht vorgesehen.
- **Generische Auslegung** – Der FB ist als generischer Typ (`GenericClassName = 'GEN_A2X_SPLIT'`) hinterlegt, sodass er in verschiedenen Kontexten eingesetzt werden kann.
- **Keine Zustände oder Ereignisse** – Es existieren keine internen Zustandsautomaten oder ereignisgesteuerten Abläufe; der FB ist rein kombinatorisch.
- **Vollständige Entkopplung** – Durch die Verwendung von Adaptern werden die angeschlossenen Bausteine entkoppelt, was die Wiederverwendbarkeit erhöht.

Zustandsübersicht
Der Baustein besitzt keinen eigenen Zustandsautomaten. Er ist jederzeit passiv und aktiv nur durch die am Eingang anliegenden Adapterdaten. Daher existieren keine definierten Zustände wie INIT, RUN oder IDLE.

Anwendungsszenarien
- **Signalverteilung in der Automatisierung** – Ein Sensor liefert über einen A2X-Adapter Messwerte, die parallel an mehrere Steuerungen oder Auswerteinheiten weitergeleitet werden müssen.

- **Kaskadierung von Adaptersignalen** – In komplexen Hierarchien kann ein A2X-Signal auf mehrere untergeordnete Module aufgeteilt werden.
- **Test- und Simulationsumgebungen** – Zum gleichzeitigen Anschluss mehrerer Prüfstände oder Visualisierungskomponenten an eine Datenquelle.

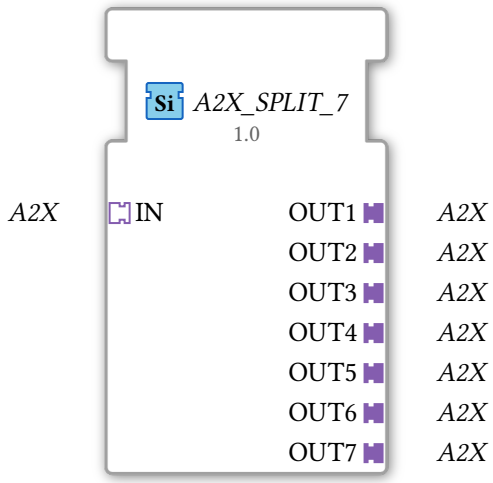
Vergleich mit ähnlichen Bausteinen

- **A2X_SPLIT_2 / A2X_SPLIT_4 / A2X_SPLIT_N** – Diese Bausteine bieten die gleiche Funktionalität, jedoch mit zwei, vier oder einer konfigurierbaren Anzahl von Ausgängen. `A2X_SPLIT_6_UNGATED` stellt eine feste 6-fach-Verteilung bereit.
- **A2X_MERGE** – Im Gegensatz zum Splitter fasst dieser Baustein mehrere A2X-Eingänge zu einem Ausgang zusammen.
- **Standard-Daten-Splitter** (z. B. für einfache Datentypen) – Adapterbasierte Splitter wie dieser arbeiten auf einer höheren Abstraktionsebene und kapseln komplexe Signalbündel.
- `A2X_SPLIT_6`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `A2X_SPLIT_6_UNGATED` ist ein einfacher, aber nützlicher Funktionsblock zur Vervielfachung von A2X-Adapter-Signalen. Durch seinen generischen Aufbau und die reine Verteilerfunktion eignet er sich besonders für Szenarien, in denen ein Signal ohne Verarbeitungslogik an mehrere Empfänger gesendet werden muss. Die Abwesenheit von Ereignissen und Zuständen macht ihn leicht verständlich und ressourcenschonend einsetzbar.

A2X_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **A2X_SPLIT_7** dient dazu, ein eingehendes A2X-Adaptersignal auf sieben identische Ausgänge zu verteilen. Er wird als generischer FB (generic FB) bereitgestellt und ermöglicht eine effiziente Signalvervielfachung innerhalb von 4diac-IDE-Projekten.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine

Ereignis-Ausgänge

- Keine

Daten-Eingänge

- Keine

Daten-Ausgänge

- Keine

Adapter

Typ	Name	Beschreibung
A2X (Socket)	**IN**	Eingangsadapter für das zu verteilende A2X-Signal
A2X (Plug)	**OUT1** ... **OUT7**	Sieben Ausgangsadapter, die jeweils eine identische Kopie des Eingangssignals bereitstellen

Funktionsweise

Der Baustein leitet das am Adapter **IN** anliegende A2X-Signal unverändert an alle sieben Ausgangsadapter (**OUT1** bis **OUT7**) weiter. Es findet keine Manipulation der Daten statt – die Signale werden lediglich passiv vervielfältigt. Der FB ist rein adapterbasiert und benötigt weder Ereignis- noch Dateneingänge.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_A2X_SPLIT`) deklariert und kann in verschiedenen Kontexten wiederverwendet werden.
- **Unidirektionale Adapter**: Alle verwendeten A2X-Adapter sind vom Typ `unidirectional`, d.h. die Daten fließen nur in eine Richtung (vom Socket zum Plug).
- **Keine Ereignissteuerung**: Die Signalweitergabe erfolgt rein datenflussgesteuert ohne zusätzliche Ereignisse oder Zustände.
- **Einfach erweiterbar**: Durch Anpassung der XML-Definition können beliebig viele Ausgänge (z.B. 2, 4, 10) realisiert werden.

Zustandsübersicht

Der FB besitzt keine eigene Zustandsmaschine (ECC). Die Ausgangssignale sind stets eine exakte Abbildung des Eingangssignals. Es gibt keine internen Zustände oder zeitlichen Verzögerungen.

Anwendungsszenarien

- **Verteilen eines Sensor- oder Steuerungssignals** an mehrere parallele Funktionsbausteine oder Subsysteme.
- **Bereitstellung identischer Adapterdaten** für redundante Verarbeitungspfade (z.B. Sicherheitskopien).
- **Simulation oder Testaufbauten**, bei denen ein Signal mehrfach benötigt wird, ohne die Quelle mehrfach zu instanziieren.

Vergleich mit ähnlichen Bausteinen

- **A2X_SPLIT_2, A2X_SPLIT_3, ...** : Diese Bausteine verteilen das Eingangssignal auf zwei, drei bzw. mehr Ausgänge. Der vorliegende FB ist speziell auf sieben Ausgänge ausgelegt und kann als Spezialisierung eines generischen Split-Bausteins betrachtet werden.
- **Manuelle Vervielfachung per Verbindung**: Theoretisch könnte ein A2X-Signal durch mehrfaches Verbinden eines Ausgangs ebenfalls verteilt werden – der Split-FB sorgt jedoch für eine saubere, strukturierte und wiederverwendbare Lösung.
- `A2X_SPLIT_7_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

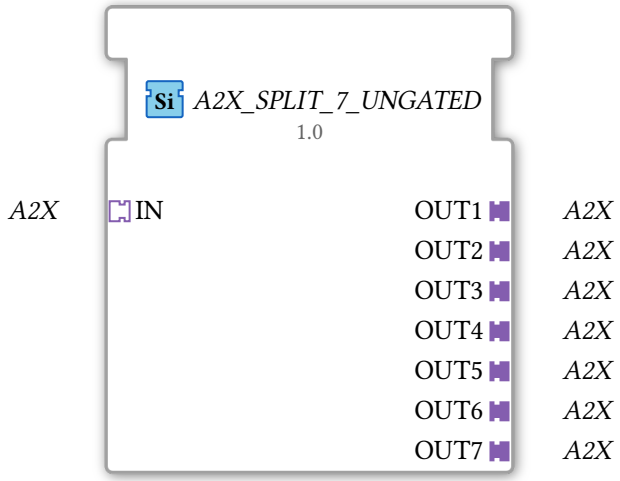
Der **A2X_SPLIT_7** ist ein einfacher, aber nützlicher Funktionsbaustein zur Signalvervielfachung in A2X-basierten Adaptersystemen. Durch seinen generischen Charakter und die klare Schnittstelle eignet er sich hervorragend für modulare Automatisierungslösungen mit der 4diac-IDE.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

A2X_SPLIT_7_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von A2X_SPLIT_7. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **A2X_SPLIT_7_UNGATED** dient dazu, ein eingehendes A2X-Adaptersignal auf sieben identische Ausgänge zu verteilen. Er wird als generischer FB (generic FB) bereitgestellt und ermöglicht eine effiziente Signalvervielfachung innerhalb von 4diac-IDE-Projekten.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine

Ereignis-Ausgänge

- Keine

Daten-Eingänge

- Keine

Daten-Ausgänge

- Keine

Adapter

Typ	Name	Beschreibung
A2X (Socket)	**IN**	Eingangsadapter für das zu verteilende A2X-Signal
A2X (Plug)	**OUT1** … **OUT7**	Sieben Ausgangsadapter, die jeweils eine identische Kopie des Eingangssignals bereitstellen

Funktionsweise

Der Baustein leitet das am Adapter **IN** anliegende A2X-Signal unverändert an alle sieben Ausgangsadapter (**OUT1** bis **OUT7**) weiter. Es findet keine Manipulation der Daten statt – die Signale werden lediglich passiv vervielfältigt. Der FB ist rein adapterbasiert und benötigt weder Ereignis- noch Dateneingänge.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_A2X_SPLIT`) deklariert und kann in verschiedenen Kontexten wiederverwendet werden.
- **Unidirektionale Adapter**: Alle verwendeten A2X-Adapter sind vom Typ `unidirectional`, d.h. die Daten fließen nur in eine Richtung (vom Socket zum Plug).
- **Keine Ereignissteuerung**: Die Signalweitergabe erfolgt rein datenflussgesteuert ohne zusätzliche Ereignisse oder Zustände.
- **Einfach erweiterbar**: Durch Anpassung der XML-Definition können beliebig viele Ausgänge (z.B. 2, 4, 10) realisiert werden.

Zustandsübersicht

Der FB besitzt keine eigene Zustandsmaschine (ECC). Die Ausgangssignale sind stets eine exakte Abbildung des Eingangssignals. Es gibt keine internen Zustände oder zeitlichen Verzögerungen.

Anwendungsszenarien

- **Verteilen eines Sensor- oder Steuerungssignals** an mehrere parallele Funktionsbausteine oder Subsysteme.
- **Bereitstellung identischer Adapterdaten** für redundante Verarbeitungspfade (z.B. Sicherheitskopien).
- **Simulation oder Testaufbauten**, bei denen ein Signal mehrfach benötigt wird, ohne die Quelle mehrfach zu instanziieren.

Vergleich mit ähnlichen Bausteinen

- **A2X_SPLIT_2, A2X_SPLIT_3, ...** : Diese Bausteine verteilen das Eingangssignal auf zwei, drei bzw. mehr Ausgänge. Der vorliegende FB ist speziell auf sieben Ausgänge ausgelegt und kann als Spezialisierung eines generischen Split-Bausteins betrachtet werden.
- **Manuelle Vervielfachung per Verbindung**: Theoretisch könnte ein A2X-Signal durch mehrfaches Verbinden eines Ausgangs ebenfalls verteilt werden – der Split-FB sorgt jedoch für eine saubere, strukturierte und wiederverwendbare Lösung.
- `A2X_SPLIT_7`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

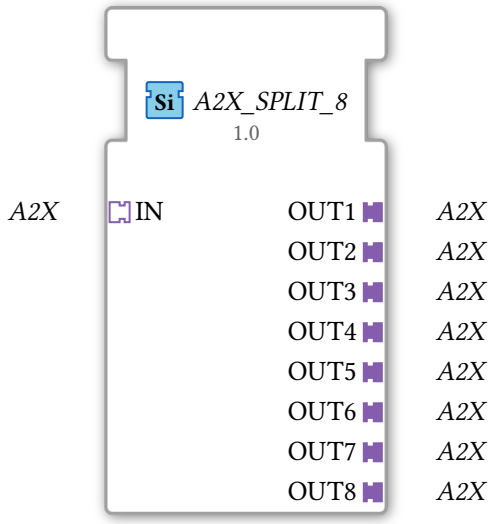
Fazit

Der **A2X_SPLIT_7_UNGATED** ist ein einfacher, aber nützlicher Funktionsbaustein zur Signalvervielfachung in A2X-basierten Adaptersystemen. Durch seinen generischen Charakter und die klare Schnittstelle eignet er sich hervorragend für modulare Automatisierungslösungen mit der 4diac-IDE.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

A2X_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock A2X_SPLIT_8 dient als generischer Baustein, um einen eingehenden A2X-Adapter (Socket) auf acht identische A2X-Adapter-Ausgänge (Plugs) zu verteilen. Er wird in der industriellen Automatisierung eingesetzt, wenn ein Signal mehrfach benötigt wird.

Schnittstellenstruktur

Ereignis-Eingänge

Nicht vorhanden.

Ereignis-Ausgänge

Nicht vorhanden.

Daten-Eingänge

Nicht vorhanden.

Daten-Ausgänge

Nicht vorhanden.

Adapter

Der FB besitzt folgende Adapter-Schnittstellen:

Socket (Eingang):

- `IN` (Typ: `adapter::types::unidirectional::A2X`)

Plugs (Ausgänge):

- `OUT1` bis `OUT8` (jeweils Typ: `adapter::types::unidirectional::A2X`)

Funktionsweise

Der FB leitet die über den Socket `IN` empfangenen Daten unverändert an alle acht Plugs `OUT1` bis `OUT8` weiter. Es findet keine Bearbeitung oder Umwandlung der Daten statt. Der FB fungiert als reiner Verteiler (Splitter) für A2X-Adapter.

Technische Besonderheiten

- Der Baustein ist als generischer FB (GenericClassName: `GEN_A2X_SPLIT`) implementiert, was eine flexible Wiederverwendung in verschiedenen Kontexten ermöglicht.
- Es sind keine Ereignisse oder Datenpunkte vorhanden; die gesamte Kommunikation erfolgt über die A2X-Adapter.

- Der FB ist kompatibel mit der Norm IEC 61499-2.
- Die Ausgänge sind voneinander unabhängig und können einzeln mit anderen FBs verbunden werden.

Zustandsübersicht

Da der FB weder Ereigniseingänge noch einen Ausführungszustand besitzt, existiert keine Zustandsmaschine. Der FB arbeitet kontinuierlich ohne internen Zustand.

Anwendungsszenarien

- Verteilung eines Sensorsignals (z. B. Temperatur, Druck) an mehrere Verbraucher oder Steuerungen.
- Parallelschaltung eines Steuersignals an mehrere Aktoren.
- Bereitstellung eines Referenzwertes für mehrere Regelkreise.

Vergleich mit ähnlichen Bausteinen

- **A2X_SPLIT_2**, **A2X_SPLIT_4**: Bausteine mit geringerer Ausgangsanzahl, die je nach Bedarf eingesetzt werden können.
- **A2X_MERGE**: Führt mehrere A2X-Eingänge zu einem Ausgang zusammen – gegensätzliche Funktion.
- **A2X_SELECT**: Selektiert einen von mehreren A2X-Eingängen, während der Splitter alle Ausgänge gleichzeitig bedient.
- `A2X_SPLIT_8_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

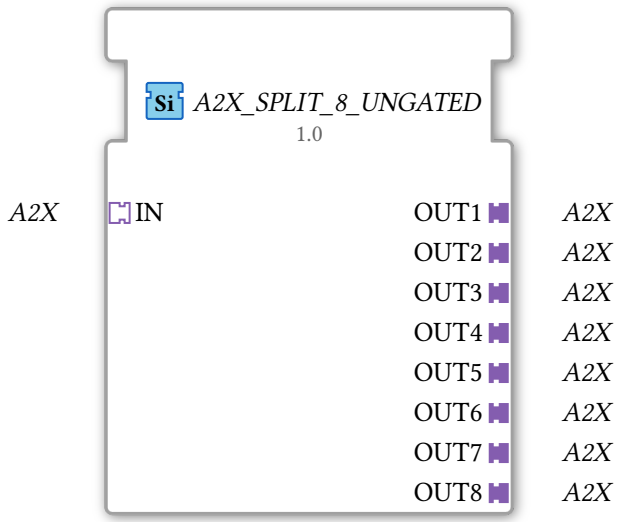
Der A2X_SPLIT_8 ist ein einfacher, aber nützlicher Baustein zur Vervielfältigung von A2X-Signalen in Automatisierungsprojekten. Durch seine generische Auslegung und die klare Trennung von Eingang und Ausgängen bietet er eine saubere und wiederverwendbare Lösung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Gesamtwiderstand in Reihen- & Parallelschaltung auf ms-muc-docs.de

A2X_SPLIT_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `A2X_SPLIT_8`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock A2X_SPLIT_8_UNGATED dient als generischer Baustein, um einen eingehenden A2X-Adapter (Socket) auf acht identische A2X-Adapter-Ausgänge (Plugs) zu verteilen. Er wird in der industriellen Automatisierung eingesetzt, wenn ein Signal mehrfach benötigt wird.

Schnittstellenstruktur

Ereignis-Eingänge

Nicht vorhanden.

Ereignis-Ausgänge

Nicht vorhanden.

Daten-Eingänge

Nicht vorhanden.

Daten-Ausgänge

Nicht vorhanden.

Adapter

Der FB besitzt folgende Adapter-Schnittstellen:

Socket (Eingang):

- `IN` (Typ: `adapter::types::unidirectional::A2X`)

Plugs (Ausgänge):

- `OUT1` bis `OUT8` (jeweils Typ: `adapter::types::unidirectional::A2X`)

Funktionsweise

Der FB leitet die über den Socket `IN` empfangenen Daten unverändert an alle acht Plugs `OUT1` bis `OUT8` weiter. Es findet keine Bearbeitung oder Umwandlung der Daten statt. Der FB fungiert als reiner Verteiler (Splitter) für A2X-Adapter.

Technische Besonderheiten

- Der Baustein ist als generischer FB (GenericClassName: `GEN_A2X_SPLIT`) implementiert, was eine flexible Wiederverwendung in verschiedenen Kontexten ermöglicht.
- Es sind keine Ereignisse oder Datenpunkte vorhanden; die gesamte Kommunikation erfolgt über die A2X-Adapter.
- Der FB ist kompatibel mit der Norm IEC 61499-2.
- Die Ausgänge sind voneinander unabhängig und können einzeln mit anderen FBs verbunden werden.

Zustandsübersicht

Da der FB weder Ereigniseingänge noch einen Ausführungszustand besitzt, existiert keine Zustandsmaschine. Der FB arbeitet kontinuierlich ohne internen Zustand.

Anwendungsszenarien

- Verteilung eines Sensorsignals (z. B. Temperatur, Druck) an mehrere Verbraucher oder Steuerungen.
- Parallelschaltung eines Steuersignals an mehrere Aktoren.
- Bereitstellung eines Referenzwertes für mehrere Regelkreise.

Vergleich mit ähnlichen Bausteinen

- **A2X_SPLIT_2**, **A2X_SPLIT_4**: Bausteine mit geringerer Ausgangsanzahl, die je nach Bedarf eingesetzt werden können.
- **A2X_MERGE**: Führt mehrere A2X-Eingänge zu einem Ausgang zusammen – gegensätzliche Funktion.
- **A2X_SELECT**: Selektiert einen von mehreren A2X-Eingängen, während der Splitter alle Ausgänge gleichzeitig bedient.
- **A2X_SPLIT_8**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

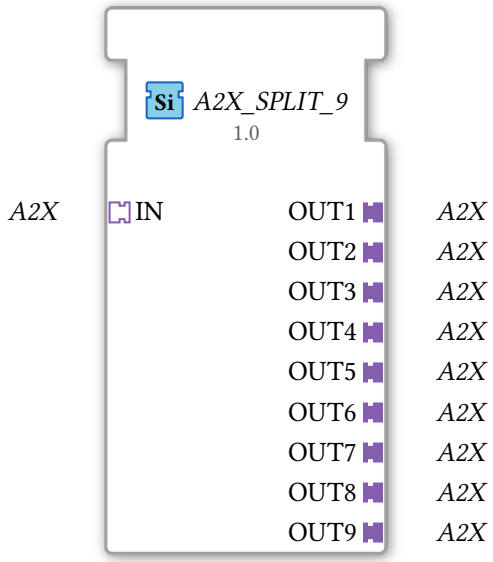
Fazit

Der A2X_SPLIT_8_UNGATED ist ein einfacher, aber nützlicher Baustein zur Vervielfältigung von A2X-Signalen in Automatisierungsprojekten. Durch seine generische Auslegung und die klare Trennung von Eingang und Ausgängen bietet er eine saubere und wiederverwendbare Lösung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Gesamtwiderstand in Reihen- & Parallelschaltung auf ms-muc-docs.de

A2X_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **A2X_SPLIT_9** ist ein generischer Funktionsblock, der einen eingehenden A2X-Adapter (Socket) auf neun separate A2X-Ausgangsadapter (Plugs) aufteilt. Er ermöglicht die sternförmige Verteilung eines A2X-Signals an bis zu neun unabhängige Empfänger, ohne dass eine eigene Logik oder Zustandsautomaten erforderlich sind.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Typ	Name	Richtung	Beschreibung
Socket	IN	Eingang	Eingehender A2X-Adapter, dessen Signale auf die neun Ausgänge verteilt werden.

Plug	OUT1 ... OUT9	Ausgang	Neun gleichartige A2X-Ausgangsadapter, die die Daten des Eingangs weiterleiten.

Funktionsweise

Der Funktionsblock besitzt keinerlei interne Logik oder Zustandsautomaten. Er fungiert als reine „Verdrahtungshilfe“ in der 4diac-IDE: Der am Socket **IN** anliegende A2X-Adapter wird unverändert und parallel an alle neun Plugs **OUT1** bis **OUT9** weitergegeben. Somit erhalten alle angeschlossenen Empfänger identische Signale wie der Eingang.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generisch deklariert (Attribut `GenericClassName = 'GEN_A2X_SPLIT'`). In der IDE kann er für verschiedene konkrete A2X-Adaptertypen parametrisiert werden.
- **Keine Laufzeitlogik**: Der FB enthält keine ECC (Execution Control Chart) und keine Algorithmen – die gesamte Funktionalität wird durch die interne Verbindung der Adapter realisiert.
- **Unidirektionale Adapter**: Die verwendeten Adapter `adapter::types::unidirectional::A2X` erlauben nur eine Datenflussrichtung (vom Socket zu den Plugs).

Zustandsübersicht

Eine Zustandsübersicht entfällt, da der Baustein keinen internen Zustand besitzt. Der Datenfluss erfolgt kontinuierlich und ohne Steuerung durch Ereignisse.

Anwendungsszenarien

- **Signalverteilung in der Automatisierung**: Verteilen eines analogen oder binären A2X-Signals an mehrere Aktoren oder übergeordnete Steuerungen.
- **Test- und Simulationsaufbauten**: Einspeisen eines Referenzsignals in mehrere parallele Prüfpfade.
- **Topologieumsetzung**: Sternförmige Verkabelung in der grafischen Konfiguration, um einen zentralen Datenpunkt mehrfach zu nutzen.

Vergleich mit ähnlichen Bausteinen

- **A2X_SPLIT_N**: Ein parametrierbarer Splitter mit variabler Anzahl von Ausgängen, z. B. 2, 4 oder 8. Der **A2X_SPLIT_9** ist fest auf neun Ausgänge ausgelegt.
- **A2X_MERGE**: Der umgekehrte Baustein, der mehrere Eingänge zu einem Ausgang zusammenführt (z. B. für Bus-Topologien).
- **Ereignisbasierte Splitter**: Bausteine wie `SPLIT` oder `E_SPLIT` arbeiten mit Ereignissignalen und erfordern Steuerlogik – der **A2X_SPLIT_9** kommt ohne Ereignisse aus.
- `A2X_SPLIT_9_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der **A2X_SPLIT_9** ist ein einfacher, aber effektiver Funktionsblock zur Verteilung eines A2X-Signals auf neun identische Ausgänge. Durch seine generische Natur und das Fehlen interner Logik ist er leichtgewichtig und ideal für reine Verdrahtungsszenarien in der 4diac-IDE. Er eignet sich besonders für Anwendungen, bei denen ein Signal mehrfach benötigt wird, ohne zusätzliche Programmierarbeit.

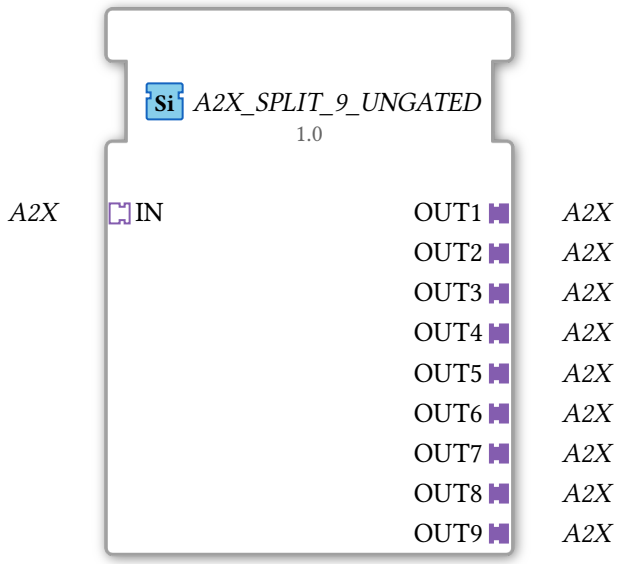
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

A2X_SPLIT_9_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `A2X_SPLIT_9`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht

gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **A2X_SPLIT_9_UNGATED** ist ein generischer Funktionsblock, der einen eingehenden A2X-Adapter (Socket) auf neun separate A2X-Ausgangsadapter (Plugs) aufteilt. Er ermöglicht die sternförmige Verteilung eines A2X-Signals an bis zu neun unabhängige Empfänger, ohne dass eine eigene Logik oder Zustandsautomaten erforderlich sind.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Typ	Name	Richtung	Beschreibung
Socket	IN	Eingang	Eingehender A2X-Adapter, dessen Signale auf die neun Ausgänge verteilt werden.
Plug	OUT1 ... OUT9	Ausgang	Neun gleichartige A2X-Ausgangsadapter, die die Daten des Eingangs weiterleiten.

Funktionsweise

Der Funktionsblock besitzt keinerlei interne Logik oder Zustandsautomaten. Er fungiert als reine „Verdrahtungshilfe“ in der 4diac-IDE: Der am Socket **IN** anliegende A2X-Adapter wird unverändert und parallel an alle neun Plugs **OUT1** bis **OUT9** weitergegeben. Somit erhalten alle angeschlossenen Empfänger identische Signale wie der Eingang.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generisch deklariert (Attribut `GenericClassName = 'GEN_A2X_SPLIT'`). In der IDE kann er für verschiedene konkrete A2X-Adaptertypen parametrisiert werden.
- **Keine Laufzeitlogik**: Der FB enthält keine ECC (Execution Control Chart) und keine Algorithmen – die gesamte Funktionalität wird durch die interne Verbindung der Adapter realisiert.
- **Unidirektionale Adapter**: Die verwendeten Adapter `adapter::types::unidirectional::A2X` erlauben nur eine Datenflussrichtung (vom Socket zu den Plugs).

Zustandsübersicht

Eine Zustandsübersicht entfällt, da der Baustein keinen internen Zustand besitzt. Der Datenfluss erfolgt kontinuierlich und ohne Steuerung durch Ereignisse.

Anwendungsszenarien

- **Signalverteilung in der Automatisierung**: Verteilen eines analogen oder binären A2X-Signals an mehrere Aktoren oder übergeordnete Steuerungen.
- **Test- und Simulationsaufbauten**: Einspeisen eines Referenzsignals in mehrere parallele Prüfpfade.
- **Topologieumsetzung**: Sternförmige Verkabelung in der grafischen Konfiguration, um einen zentralen Datenpunkt mehrfach zu nutzen.

Vergleich mit ähnlichen Bausteinen

- **A2X_SPLIT_N**: Ein parametrierbarer Splitter mit variabler Anzahl von Ausgängen, z. B. 2, 4 oder 8. Der **A2X_SPLIT_9_UNGATED** ist fest auf neun Ausgänge ausgelegt.
- **A2X_MERGE**: Der umgekehrte Baustein, der mehrere Eingänge zu einem Ausgang zusammenführt (z. B. für Bus-Topologien).
- **Ereignisbasierte Splitter**: Bausteine wie `SPLIT` oder `E_SPLIT` arbeiten mit Ereignissignalen und erfordern Steuerlogik – der **A2X_SPLIT_9_UNGATED** kommt ohne Ereignisse aus.
- `A2X_SPLIT_9`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

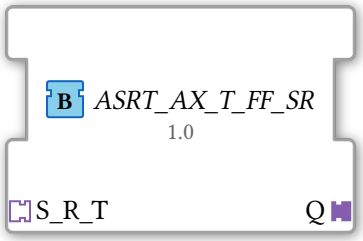
Der **A2X_SPLIT_9_UNGATED** ist ein einfacher, aber effektiver Funktionsblock zur Verteilung eines A2X-Signals auf neun identische Ausgänge. Durch seine generische Natur und das Fehlen interner Logik ist er leichtgewichtig und ideal für reine Verdrahtungsszenarien in der 4diac-IDE. Er eignet sich besonders für Anwendungen, bei denen ein Signal mehrfach benötigt wird, ohne zusätzliche Programmierarbeit.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

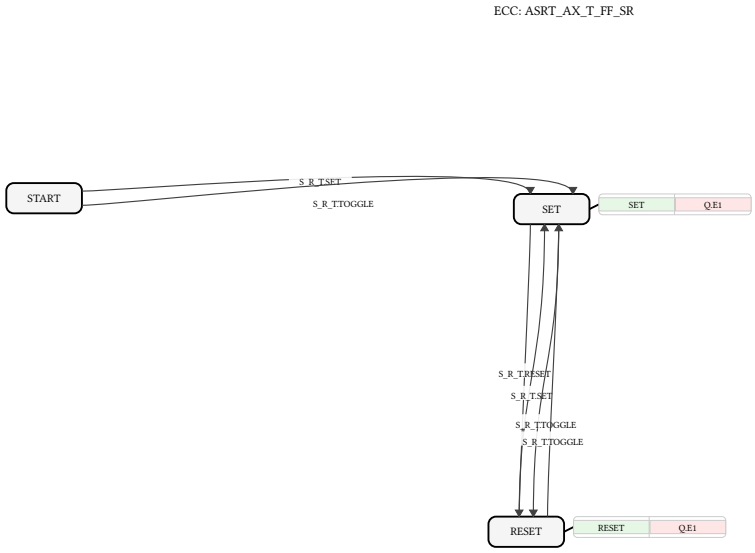
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ASRT_AX_T_FF_SR

Set/Reset/Toggle – *ASRT*



AX – value of flip flop



• – ○ • – ○ • – ○ •

Einleitung

Der ASRT_AX_T_FF_SR ist ein ereignisgesteuerter bistabiler Funktionsblock mit Toggle-Funktionalität – die direkte Entsprechung von AX_T_FF_SR, bei der die drei Ereigniseingänge S/R/CLK zu einem einzigen unidirektionalen **ASRT-Socket** (Set/Reset/Toggle) zusammengefasst sind, statt als drei separate EventInputs geführt zu werden.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkten Ereigniseingänge vorhanden – Set/Reset/Toggle kommen gebündelt über den Adapter-Socket `S_R_T`

Ereignis-Ausgänge

- Keine direkten Ereignisausgänge vorhanden

Daten-Eingänge

- Keine Dateneingänge vorhanden

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden

Adapter

- **S_R_T**: Unidirektionaler Adapter-Socket vom Typ `adapter::types::unidirectional::ASRT` (Set/Reset/Toggle)
- **Q**: Unidirektionaler Adapter-Plug vom Typ `adapter::types::unidirectional::AX`, der den Wert des Flipflops bereitstellt

Funktionsweise

Der ASRT_AX_T_FF_SR verfügt über drei Betriebszustände:

- **START**: Initialzustand
- **SET**: Ausgang Q ist TRUE

- **RESET**: Ausgang Q ist FALSE

Die Zustandsübergänge werden durch die am Socket `S_R_T` eintreffenden Adapterereignisse gesteuert:

- `S_R_T.SET` führt von jedem Zustand in den SET-Zustand
- `S_R_T.RESET` führt von jedem Zustand in den RESET-Zustand
- `S_R_T.TOGGLE` toggelt den aktuellen Zustand (SET → RESET bzw. RESET → SET, aus START heraus nach SET)

Bei jedem Zustandswechsel wird der entsprechende Algorithmus ausgeführt, der den Adapterwert `Q.D1` entsprechend setzt (`TRUE` im SET-, `FALSE` im RESET-Zustand).

Technische Besonderheiten

- Kombiniert SR-Flipflop- und T-Flipflop-Funktionalität wie AX_T_FF_SR, jedoch mit gebündelter Adapter-Schnittstelle statt dreier einzelner EventInputs
- Verwendet die unidirektionale ASRT-Adapter-Schnittstelle für Set/Reset/Toggle und die AX-Adapter-Schnittstelle für die Datenausgabe
- Initialzustand ist START, von wo aus direkt in SET oder durch TOGGLE in SET gewechselt werden kann

Zustandsübersicht

```
START (Initialzustand)
     │
     ├── S_R_T.SET ─────→ SET (Q.D1 = TRUE)
     │
     └── S_R_T.TOGGLE ──→ SET (Q.D1 = TRUE)

SET (Q.D1 = TRUE)
     │
     ├── S_R_T.RESET ──→ RESET (Q.D1 = FALSE)
     │
     └── S_R_T.TOGGLE ──→ RESET (Q.D1 = FALSE)

RESET (Q.D1 = FALSE)
     │
     ├── S_R_T.SET ─────→ SET (Q.D1 = TRUE)
     │
     └── S_R_T.TOGGLE ──→ SET (Q.D1 = TRUE)
```

Anwendungsszenarien

- Zustandsspeicherung in Steuerungsanwendungen, bei denen Set/Reset/Toggle bereits gebündelt über einen ASRT-Adapter vorliegen (z. B. von ASRT_CLIENT_0 oder ASRT_3EVENTS_TO_SRT)
- Taktteilung und Frequenzteilung
- Zustandsautomaten mit Speicherfunktion in durchgängig adapterbasierten Anwendungen

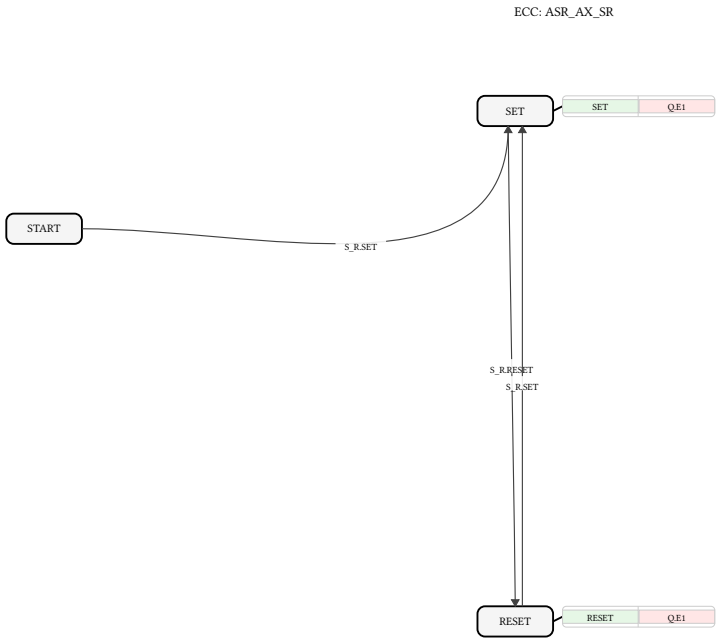
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu AX_T_FF_SR bietet der ASRT_AX_T_FF_SR dieselbe Funktionalität, jedoch mit einem einzigen ASRT-Adapter-Socket statt drei separaten EventInputs (S/R/CLK) – dadurch lässt er sich direkt an einen ASRT-liefernden Baustein anschließen, ohne die Ereignisse einzeln verdrahten zu müssen.

Fazit

Der ASRT_AX_T_FF_SR ist die adapterbasierte Variante des bewährten SR/T-Flipflop-Musters aus AX_T_FF_SR. Durch die Bündelung von Set/Reset/Toggle in einem ASRT-Socket eignet er sich besonders für durchgängig adapterbasierte Steuerungsarchitekturen.

ASR_AX_SR

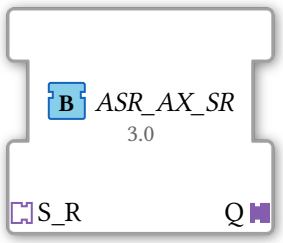


• – ○ • – ○ • – ○ •

Einleitung

Der ASR_AX_SR ist ein ereignisgesteuerter bistabiler Funktionsblock (Flip-Flop), der nach dem Set-Reset-Prinzip arbeitet. Er dient zur Speicherung eines binären Zustands und wird über Adapter-Schnittstellen angesteuert.

Set/Reset – *ASR*



AX – value of flip flop

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereigniseingänge vorhanden - die Steuerung erfolgt ausschließlich über Adapter.

Ereignis-Ausgänge

Keine direkten Ereignisausgänge vorhanden - die Ausgabe erfolgt ausschließlich über Adapter.

Daten-Eingänge

Keine direkten Dateneingänge vorhanden.

Daten-Ausgänge

Keine direkten Datenausgänge vorhanden.

Adapter

- **S_R (Socket)**: Set/Reset-Steueradapter vom Typ `adapter::types::unidirectional::ASR`
- **Q (Plug)**: Ausgabeadapter vom Typ `adapter::types::unidirectional::AX` für den Flip-Flop-Zustand

Funktionsweise

Der ASR_AX_SR arbeitet als Set-Reset-Flip-Flop mit drei Zuständen:

- **START**: Initialzustand
- **SET**: Gesetzter Zustand (Q = TRUE)
- **RESET**: Rückgesetzter Zustand (Q = FALSE)

Bei einem SET-Ereignis über den S_R-Adapter wechselt der Block in den SET-Zustand und setzt den Ausgang Q auf TRUE. Bei einem RESET-Ereignis wechselt er in den RESET-Zustand und setzt Q auf FALSE.

Technische Besonderheiten

- **Keine Priorisierung (Dominanz)**: Da das Setzen und Rücksetzen durch getrennte Ereignisse (`S_R.SET`, `S_R.RESET`) ausgelöst wird, gibt es keine gleichzeitige Dominanz wie in der IEC 61131-3. Der Zustand wird durch das zuletzt eingetroffene Ereignis bestimmt („Last Event Wins“).
- **Verwendung von Adaptern**: Verwendet unidirektionale Adapter für Ein- und Ausgabe (`ASR` für Set/Reset, `AX` für Ausgang).
- **Implementierung**: Realisiert als Basic Function Block mit explizitem Zustandsautomaten (ECC).
- **Schnittstelle**: Keine direkten Ein-/Ausgänge, ausschließlich Adapter-basierte Kommunikation.

Zustandsübergänge

```
START → SET:    bei S_R.SET Ereignis
SET → RESET:    bei S_R.RESET Ereignis
RESET → SET:    bei S_R.SET Ereignis
```

Algorithmen

- **SET**: Setzt den Ausgangswert Q.D1 auf TRUE
- **RESET**: Setzt den Ausgangswert Q.D1 auf FALSE

Anwendungsszenarien

- Speicherung von Schaltzuständen in Steuerungsanwendungen
- Zustandsspeicherung in sequentiellen Ablaufsteuerungen
- Flip-Flop-Funktionalität in verteilten Automatisierungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen

- **SR (IEC 61131-3)**: Der klassische SR-Baustein hat eine definierte Setz-Dominanz bei gleichzeitigen Signalen. Der `ASR_AX_SR` verhält sich dagegen zeitabhängig (letztes Ereignis zählt).
- **E_SR**: Funktional ähnlich (ereignisgesteuert), aber mit direkten Event-Pins statt Adaptern.
- **Konventionelle Flip-Flops**: Im Vergleich zu Flip-Flops mit direkten Ein-/Ausgängen erleichtert das Adapter-Design die Integration in modulare Systemarchitekturen.

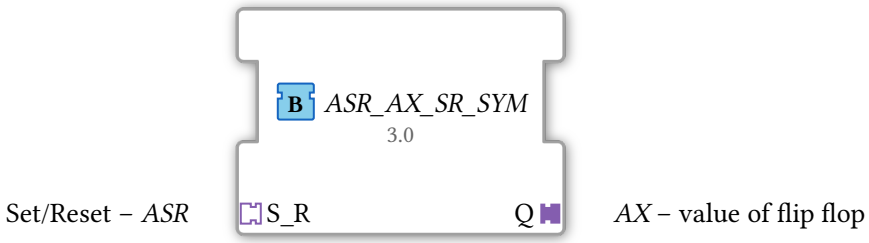
🛠️ Zugehörige Übungen

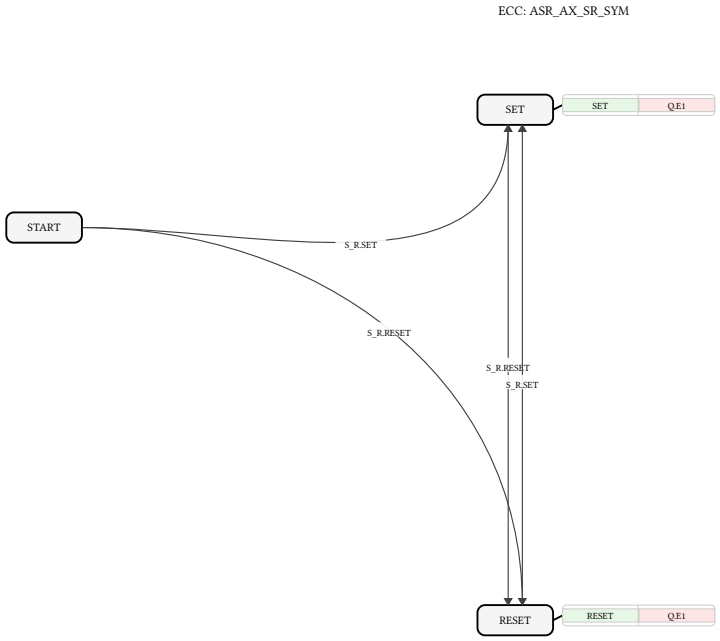
- Uebung_171_AX

Fazit

Der ASR_AX_SR bietet eine saubere, adapter-basierte Implementierung eines bistabilen Speicherelements, das ideal für den Einsatz in modularen IEC 61499-Systemen geeignet ist. Die ausschließliche Verwendung von Adaptern ermöglicht eine hohe Flexibilität bei der Systemintegration.

ASR_AX_SR_SYM





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ASR_AX_SR_SYM** realisiert ein ereignisgesteuertes, bistabiles Flipflop (SR-Flipflop) mit symmetrischem Startverhalten. Er ist als Basic-FB gemäß IEC 61499-1 (Annex A) implementiert und arbeitet ausschließlich über Adapter-Schnittstellen – ohne direkte Ereignis- oder Dateneingänge. Der Baustein eignet sich für Anwendungen, bei denen der initiale Zustand durch das erste eintreffende Ereignis bestimmt wird und keinem der beiden Eingänge eine Vorrangstellung eingeräumt ist.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine direkten Ereigniseingänge. Alle Ereignisse werden über den **Socket-Adapter `S_R`** (Typ `adapter::types::unidirectional::ASR`) empfangen. Dieser Adapter stellt die folgenden Ereignis-Eingänge bereit:

- `S_R.SET` – Setz-Ereignis
- `S_R.RESET` – Rücksetz-Ereignis

Ereignis-Ausgänge

Der FB besitzt keine direkten Ereignisausgänge. Ausgehende Ereignisse werden über den **Plug-Adapter `Q`** (Typ `adapter::types::unidirectional::AX`) gesendet. Dieser Adapter bietet den Ereignis-Ausgang:

- `Q.E1` – Ausgangsereignis, ausgelöst bei Zustandsänderung

Daten-Eingänge

Keine direkten Dateneingänge.

Daten-Ausgänge

Der Ausgangswert wird über den Plug-Adapter `Q` bereitgestellt:

- `Q.D1` (BOOL) – aktueller Zustand des Flipflops (`TRUE` = gesetzt, `FALSE` = rückgesetzt)

Adapter

Bezeichnung	Typ	Richtung	Beschreibung

S_R	adapter::types::unidirectional::ASR	Socket	Set-/Reset-Ereignisse (Eingang)
Q	adapter::types::unidirectional::AX	Plug	Zustandsausgabe (Ereignis + Daten)

Funktionsweise

Der Baustein startet nach der Initialisierung im Zustand **START**. In diesem Zustand wartet er auf das erste eintreffende Ereignis an `S_R.SET` oder `S_R.RESET`. Je nachdem, welches Ereignis zuerst eintrifft, wechselt er in den Zustand **SET** oder **RESET**:

- **SET-Zustand**:
  Der Datenausgang `Q.D1` wird auf `TRUE` gesetzt, das Ereignis `Q.E1` wird ausgelöst.
  Bei einem späteren `S_R.RESET` erfolgt der Übergang in den Zustand RESET.
- **RESET-Zustand**:
  Der Datenausgang `Q.D1` wird auf `FALSE` gesetzt, das Ereignis `Q.E1` wird ausgelöst.
  Bei einem späteren `S_R.SET` erfolgt der Übergang in den Zustand SET.

Der Baustein bleibt nach dem ersten Übergang in einem der beiden stabilen Zustände, bis das jeweils komplementäre Ereignis eintrifft. Es gibt keine Vorrangstellung von SET oder RESET – das erste Ereignis beim Start bestimmt den Anfangszustand.

Technische Besonderheiten

- **Symmetrisches Startverhalten**: Im Gegensatz zu üblichen SR-Flipflops hat keiner der Eingänge Priorität. Der Anfangszustand wird durch das zeitlich erste Ereignis bestimmt.
- **Reine Adapter-Schnittstelle**: Sowohl Ereignisse als auch Daten werden über IEC 61499-Adapter ausgetauscht, nicht über direkte Ein-/Ausgangsports. Dies vereinfacht die Wiederverwendung in komplexen, adapterbasierten Systemen.
- **Keine expliziten Ereignis-E/A**: Der FB verzichtet auf eigene EVENT-Eingänge/-Ausgänge und nutzt stattdessen die integrierten Ereignisports der Adapter.
- **Compact-ST-Implementierung**: Die beiden Algorithmen `SET` und `RESET` setzen lediglich den Datenausgang und lösen das Ausgangsereignis aus.

Zustandsübersicht

(Hinweis: Ein grafisches Zustandsdiagramm kann aus dem ECC-XML extrahiert werden, ist hier jedoch nicht als Bild eingebettet.)

Zustand	Beschreibung	Aktion	Ausgang
START	Initialer Wartezustand	–	keine Ausgabe
SET	Gespeicherter Zustand „gesetzt“	`Q.D1 := TRUE`	Auslösen von `Q.E1`
RESET	Gespeicherter Zustand „rückgesetzt“	`Q.D1 := FALSE`	Auslösen von `Q.E1`

Übergänge:

- START ⟶ SET : wenn `S_R.SET` eintrifft
- START ⟶ RESET : wenn `S_R.RESET` eintrifft
- SET ⟶ RESET : wenn `S_R.RESET` eintrifft (nach erstem Set)
- RESET ⟶ SET : wenn `S_R.SET` eintrifft (nach erstem Reset)

Anwendungsszenarien

- **Binäre Zustandspeicher**: Speichern eines Ein-/Aus-Zustands, der durch zwei separate Ereignisse gesteuert wird (z. B. „Start“ und „Stopp“).
- **Ansteuerung von Aktoren mit Setz- und Rücksetzeingang**: Verwendung in Steuerungen, bei denen ein Ausgang symmetrisch gesetzt und rückgesetzt werden soll, ohne Priorisierung.
- **Initialisierung durch das erste Ereignis**: Systeme, bei denen der Startzustand nicht vordefiniert ist, sondern durch den ersten Befehl (Setzen oder Rücksetzen) bestimmt wird.

Vergleich mit ähnlichen Bausteinen

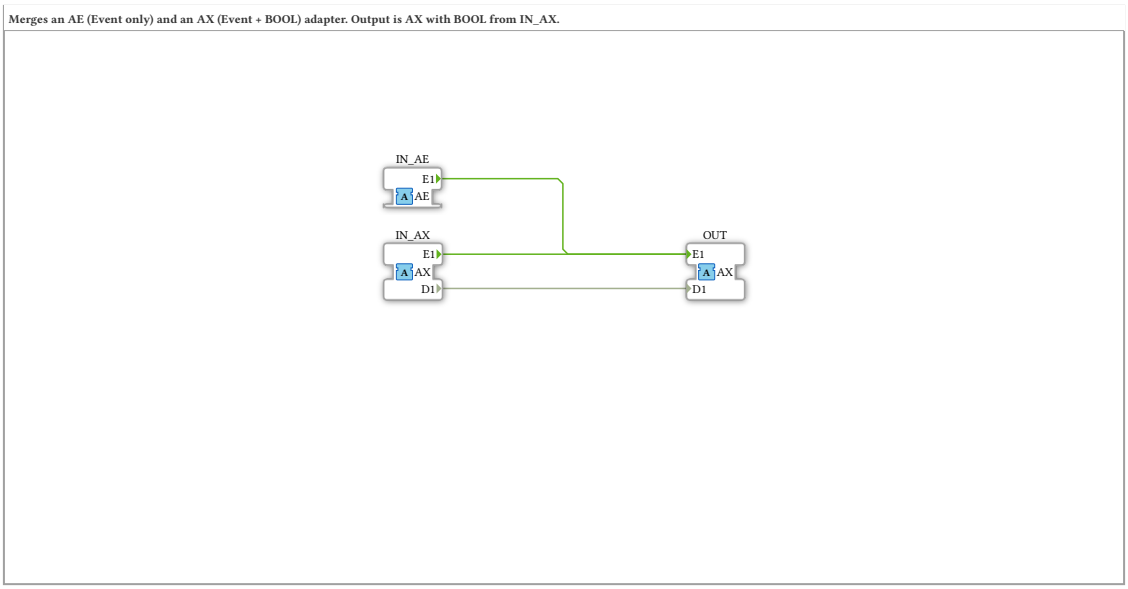
- **Standard-SR-Flipflop (z. B. `SR` klassisch)**: Häufig ist ein Eingang dominant (z. B. Reset hat Vorrang). Der `ASR_AX_SR_SYM` behandelt beide Eingänge gleichberechtigt – beim Start entscheidet das zeitlich erste Ereignis.
- **RS-Flipflop mit priorisiertem Set**: Andere Implementierungen setzen den Set-Eingang stärker.

Der vorliegende Baustein zeichnet sich durch seine symmetrische Behandlung beider Ereignisse und die ausschließliche Adapter-Schnittstelle aus, was ihn flexibel in adapterbasierten Bibliotheken macht.

Fazit

Der Funktionsblock `ASR_AX_SR_SYM` bietet ein zuverlässiges, ereignisgesteuertes SR-Flipflop mit symmetrischem Startverhalten. Durch seine Adapter-Schnittstellen fügt er sich nahtlos in IEC-61499-Systeme ein und eignet sich besonders für Anwendungen, die eine gleichberechtigte Setz-/Rücksetzsteuerung ohne Vorrang erfordern. Die einfache Zustandslogik und die geringe Anzahl an Algorithmen machen ihn effizient und leicht verständlich.

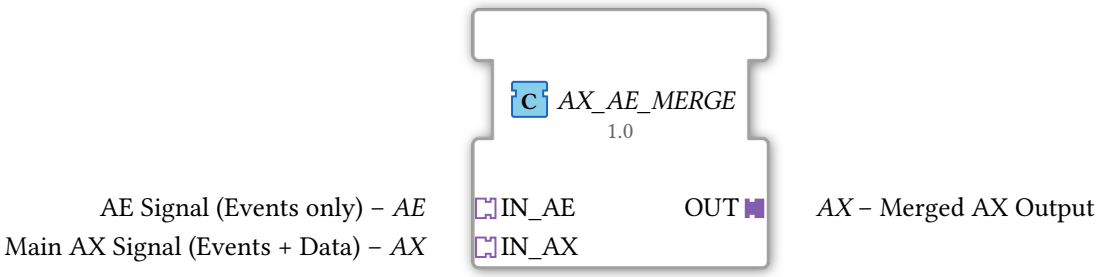
AX_AE_MERGE



Einleitung

Der AX_AE_MERGE Funktionsblock ermöglicht das Zusammenführen eines reinen Ereignis-Signals (**AE**) mit einem kombinierten Ereignis-Daten-Signal (**AX**). Das Ergebnis ist ein AX-Signalfluss, bei dem das Daten-Bit vom AX-Zweig erhalten bleibt, aber von beiden Quellen getriggert werden kann.

Dies ist besonders nützlich, wenn ein zyklisches Ereignis (z.B. von einem `AE_CYCLE`) einen Status-Check oder eine Aktion in einem AX-basierten Datenstrom erzwingen soll, ohne die Daten selbst zu verändern.



Schnittstellenstruktur

Adapter

- **IN_AX** (Socket): Primärer Eingang (Ereignis `E1` + Daten `D1`).
- **IN_AE** (Socket): Zusätzlicher Ereignis-Eingang (nur Ereignis `E1`).

- **OUT** (Plug): Merged Ausgang (Ereignis von beiden Eingängen, Daten nur von `IN_AX`).

Funktionsweise

Trifft an `IN_AX` ein Ereignis ein, wird dieses zusammen mit dem aktuellen Wert von `D1` an den Ausgang weitergereicht. Trifft an `IN_AE` ein Ereignis ein, wird am Ausgang ebenfalls ein Ereignis ausgelöst, wobei der *letzte bekannte* Wert von `IN_AX.D1` am Ausgang anliegt.

⚖️ Vergleich mit ähnlichen Bausteinen

In vielen Übungen (z. B. beim Blinker in `Uebung_007a3_AX`) wurde bisher fälschlicherweise ein `AX_OR_2` verwendet, um ein `AE`-Ereignis (vom `AE_CYCLE`) in einen `AX`-Datenstrom einzuspeisen.

- **AX_OR_2**: Erwartet zwei vollwertige AX-Adapter. Das Anschließen eines AE-Adapters ist technisch unsauber, da der Datenwert für einen der Eingänge undefiniert bleibt.
- **AX_AE_MERGE**: Ist exakt für diesen Mischbetrieb ausgelegt. Er stellt sicher, dass das Ereignis durchgereicht wird, während die Datenquelle (`IN_AX`) eindeutig definiert bleibt.

Technische Besonderheiten

✓ **Ereignis-Merge**: Nutzt intern einen `E_MERGE`. ✓ **Datenerhalt**: Keine Datenveränderung, nur zusätzliche Triggerung. ✓ **Saubere Typisierung**: Verhindert das direkte Verbinden von ungleichen Adaptertypen.

Anwendungsszenarien

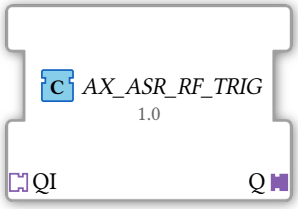
- **Blinker-Logik**: Ein `AE_CYCLE` triggert regelmäßig das Senden des aktuellen Zustands eines `AX_SR` Flip-Flops.
- **Status-Auffrischung**: Ein Timer triggert regelmäßig das Senden des aktuellen AX-Zustands.
- **Mehrfach-Trigger**: Ein AX-Signalfluss soll zusätzlich durch externe AE-Ereignisse (z.B. Taster-Clicks) angestoßen werden.

🛠️ Zugehörige Übungen

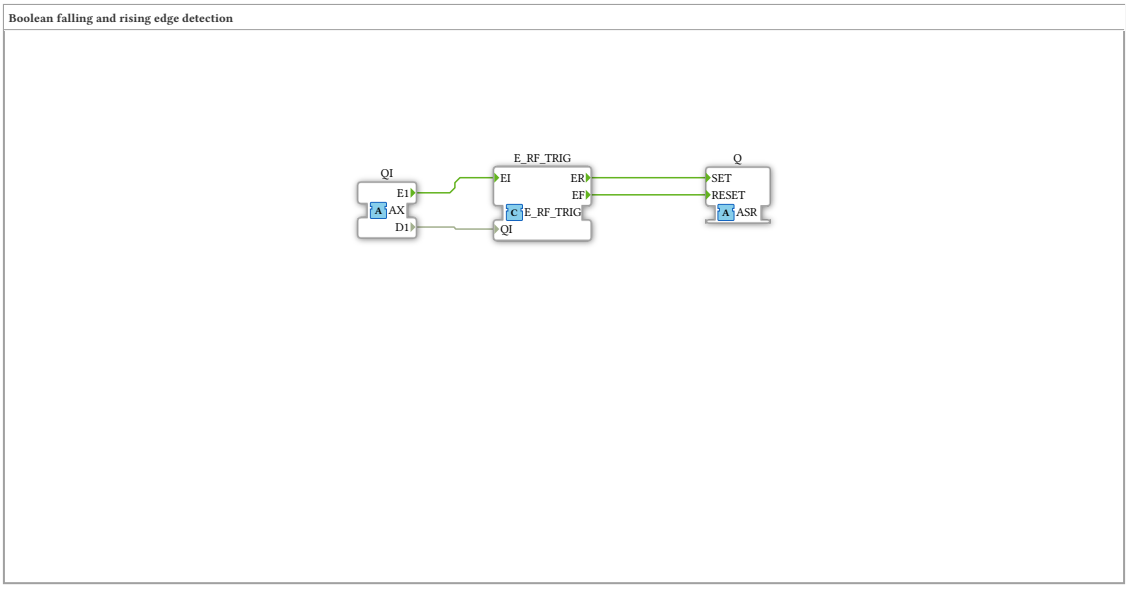
- Uebung_007a3_AX
- Uebung_008_AX
- Uebung_009_AX

AX_ASR_RF_TRIG

value to check for a rising or falling edge – *AX*



ASR – Set/Reset Output



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AX_ASR_RF_TRIG** dient der bidirektionalen Flankenerkennung (steigende und fallende) eines booleschen Eingangssignals. Er ist als Adapter-basierte Hüllkurve des IEC-61499-Standardbausteins `E_RF_TRIG` konzipiert. Über zwei unidirektionale Adapter wird das zu überwachende Signal eingelesen und das Ergebnis als Setz-/Rücksetz-Ausgang bereitgestellt.

Schnittstellenstruktur

Der FB besitzt **keine direkten** Ereignis- oder Datenein-/ausgänge. Die gesamte Kommunikation erfolgt über zwei Adapter.

Ereignis-Eingänge

- **keine**

Ereignis-Ausgänge

- **keine**

Daten-Eingänge

- **keine**

Daten-Ausgänge

- **keine**

Adapter

Adapter	Typ	Richtung	Beschreibung
QI	`adapter::types::unidirectional::AX`	Socket	Eingangsadapter – liefert den zu überwachenden Booleschen Wert (über Kanal `D1`) sowie ein Ereignis (`E1`) zur Verarbeitung.
Q	`adapter::types::unidirectional::ASR`	Plug	Ausgangsadapter – signalisiert über die Ereignisse `SET` (steigende Flanke) und `RESET` (fallende Flanke) den erkannten Flankenzustand.

Funktionsweise

1. Ein am Socket **QI** eingehendes Ereignis (über `QI.E1`) überträgt zeitgleich den booleschen Wert `QI.D1` an den internen `E_RF_TRIG`-Baustein.
2. Der integrierte `E_RF_TRIG` erkennt eine **steigende Flanke** (Übergang von FALSE auf TRUE) und eine **fallende Flanke** (Übergang von TRUE auf FALSE).
3. Bei einer steigenden Flanke erzeugt der interne FB das Ereignis `ER`, welches über den Plug **Q** als `Q.SET` ausgegeben wird.
4. Bei einer fallenden Flanke erzeugt der interne FB das Ereignis `EF`, welches als `Q.RESET` ausgegeben wird.

Somit wird das Eingangssignal in ein Setz-/Rücksetz-Signal umgesetzt, ohne dass der Anwender sich mit der internen Ereignisverarbeitung befassen muss.

Technische Besonderheiten

- **Reine Adapter-Schnittstelle:** Der FB besitzt keine direkten Eingänge/Ausgänge im FB-Typ-Interface, was eine lose Kopplung und Wiederverwendbarkeit in verschiedenen Adapternetzwerken ermöglicht.
- **Nutzung des Standardbausteins `E_RF_TRIG`:** Die Flankenerkennung basiert auf der bewährten IEC-61499-Implementierung, die sowohl steigende als auch fallende Flanken zuverlässig detektiert.
- **Keine Zustandsspeicherung außerhalb des internen FB:** Die gesamte Logik wird durch den eingebetteten `E_RF_TRIG` realisiert; der äußere Baustein dient ausschließlich als Kapselung und Adapteranpassung.

Zustandsübersicht

Der FB besitzt **keine eigene explizite Zustandsmaschine**. Der interne `E_RF_TRIG` arbeitet nach folgender impliziter Zustandslogik:

letzter Wert	aktueller Wert	ausgelöstes Ereignis
FALSE	TRUE	steigende Flanke (SET)
TRUE	FALSE	fallende Flanke (RESET)
sonst	–	kein Ereignis

Die Zustandsspeicherung erfolgt ausschließlich innerhalb des internen FB.

Anwendungsszenarien

- **Tastendruck-Erkennung:** Überwachung eines Tastersignals; `SET` beim Drücken, `RESET` beim Loslassen.
- **Signalflanken-Auswertung in Steuerungen:** Detektion von Grenzwertüberschreitungen oder Impulsanfängen/-enden.
- **Adapterbasierte Sensor-Integration:** Einbinden von Sensoren mit booleschem Ausgang in ein Adapter-Netzwerk, z. B. über die AX/ASR-Adapterfamilie.

Vergleich mit ähnlichen Bausteinen

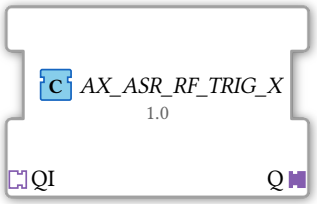
Baustein	Flankenerkennung	Ausgangssignal	Schnittstelle
`E_RF_TRIG`	steigend + fallend	separate Ereignisausgänge	direkte Events
`E_TRIG`	nur steigend	ein Ereignis	direkte Events
`E_F_TRIG`	nur fallend	ein Ereignis	direkte Events
AX_ASR_RF_TRIG	steigend + fallend	Setz-/Rücksetz-Ereignisse über Adapter	reine Adapter

Der vorliegende FB bietet im Gegensatz zu den reinen Event-Bausteinen eine Adapter-Schnittstelle, die eine einfachere Verbindung in hierarchischen oder typisierten Adapternetzen ermöglicht.

Fazit

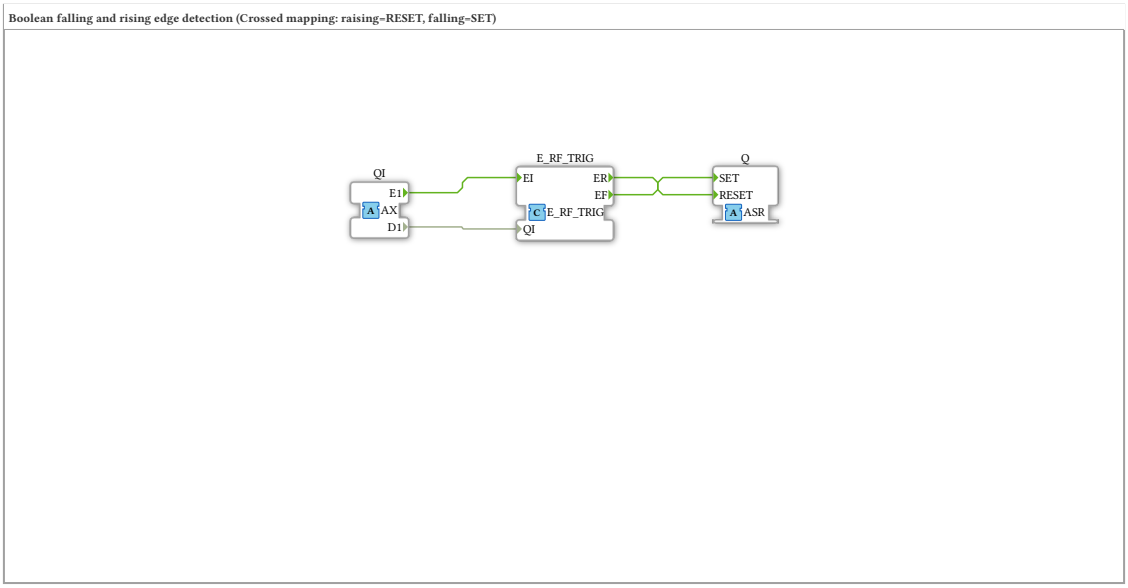
Der `AX_ASR_RF_TRIG` ist ein kompakter, auf Adapterbasis gekapselter Flankendetektor für boolesche Signale. Er vereinfacht die Verwendung des Standardbausteins `E_RF_TRIG` in adapterbasierten Architekturen und setzt die erkannten Flanken direkt in ein Setz-/Rücksetz-Signal um. Durch die lose Kopplung eignet er sich besonders für modulare, wiederverwendbare Steuerungsanwendungen.

AX_ASR_RF_TRIG_X



value to check for a rising or falling edge – *AX*

ASR – Set/Reset Output



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AX_ASR_RF_TRIG_X** dient der Erkennung von steigenden und fallenden Flanken eines booleschen Signals. Er bildet die erkannten Flanken auf ein Set/Reset-Ausgangssignal ab, wobei eine gekreuzte Zuordnung („crossed mapping“) verwendet wird: Die steigende Flanke löst einen Reset-Vorgang aus, die fallende Flanke einen Set-Vorgang. Der Baustein ist als reiner Adapterbaustein realisiert und nutzt intern den Standard-Funktionsblock `E_RF_TRIG`.

Schnittstellenstruktur

Der FB besitzt keine direkten Ereignis- oder Dateneingänge/-ausgänge auf der Schnittstellenebene. Die gesamte Kommunikation erfolgt über zwei spezifizierte Adapter.

Ereignis-Eingänge

Keine (Ereignissteuerung erfolgt über den Adapter **QI**)

Ereignis-Ausgänge

Keine (Ereignisausgabe erfolgt über den Adapter **Q**)

Daten-Eingänge

Keine (Dateneingabe erfolgt über den Adapter **QI**)

Daten-Ausgänge

Keine (Datenausgabe erfolgt über den Adapter **Q**)

Adapter

Adapter	Typ	Richtung	Beschreibung
QI	`adapter::types::unidirectional::AX`	Socket	Nimmt das auf Flanken zu prüfende boolesche Signal entgegen. Enthält

			das Ereignis `E1` (Auslöser) und das Datenwort `D1` (Wert).
Q	`adapter::types::unidirectional::ASR`	Plug	Gibt die Set/Reset-Ausgangssignale aus. Über die Ereignisse `SET` (fallende Flanke) und `RESET` (steigende Flanke) werden die entsprechenden Aktionen signalisiert.

Funktionsweise

Im FBNetzwerk des Bausteins wird eine Instanz des Standard-FBs `E_RF_TRIG` verwendet. Die Verbindungen sind wie folgt konfiguriert:

- Der Ereigniseingang `QI.E1` löst die Flankenerkennung aus und ist mit `E_RF_TRIG.EI` verbunden.
- Der Datenwert `QI.D1` wird an `E_RF_TRIG.QI` weitergeleitet.
- Das Ausgangsereignis `E_RF_TRIG.EF` (fallende Flanke) wird auf den Adapterausgang `Q.SET` gelegt.
- Das Ausgangsereignis `E_RF_TRIG.ER` (steigende Flanke) wird auf den Adapterausgang `Q.RESET` gelegt.

Durch diese Verschaltung ergibt sich das gewünschte „Crossed Mapping“:

- **Steigende Flanke** am Eingangswert → **RESET**-Impuls am Ausgang.
- **Fallende Flanke** am Eingangswert → **SET**-Impuls am Ausgang.

Technische Besonderheiten

- Der Baustein ist vollständig als Adapter-FB realisiert, d. h. er besitzt keine eigenen Ereignis-/Datenschnittstellen, sondern kommuniziert ausschließlich über die definierten Adapter **QI** und **Q**.
- Die inverse Zuordnung der Flanken (Rising → Reset, Falling → Set) ist eine Besonderheit gegenüber der Standardverwendung von `E_RF_TRIG`.
- Die Implementierung folgt der Spezifikation IEC 61499-1 und verwendet die Bibliothek `adapter::events::unidirectional`.

Zustandsübersicht

Der Baustein besitzt keinen eigenen Zustandsautomaten. Die interne Flankenerkennung erfolgt durch den integrierten FB `E_RF_TRIG`, dessen Zustandslogik (warten auf Flanke) für die Funktionalität verantwortlich ist.

Anwendungsszenarien

- **Umgekehrte Set/Reset-Logik**: In Steuerungen, bei denen ein aktiver HIGH-Zustand durch eine steigende Flanke zurückgesetzt und durch eine fallende Flanke gesetzt werden soll.
- **Schnittstellenanpassung**: Wenn ein System Flankensignale in genau dieser Zuordnung erwartet (z. B. bei speziellen Aktorsteuerungen).
- **Vereinfachung von Adapter-Verkettungen**: Durch die Kapselung der inversen Logik in einem eigenen FB wird die Netzwerkstruktur übersichtlicher.

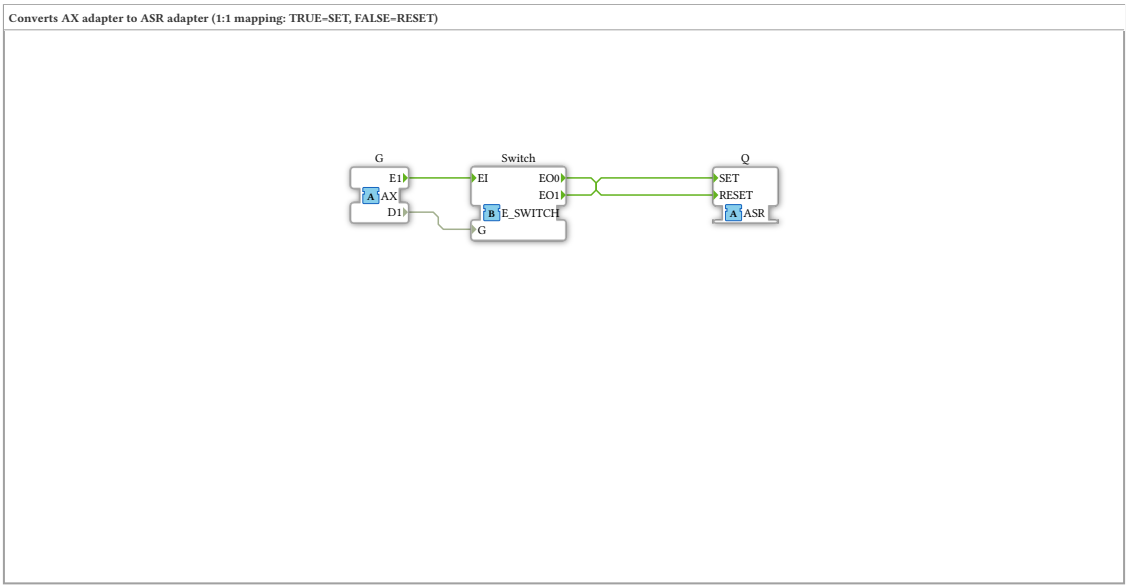
Vergleich mit ähnlichen Bausteinen

- **`E_RF_TRIG`** (IEC 61499 Standard): Liefert steigende Flanke auf `ER` und fallende Flanke auf `EF`. `AX_ASR_RF_TRIG_X` invertiert diese Zuordnung.
- **`E_RS_TRIG`**: Erkennt nur steigende Flanken. Der vorliegende Baustein erkennt beide Flanken und ordnet sie unterschiedlichen Ausgängen zu.
- **`AX_ASR_TRIG_X`**: Mögliche alternative Varianten mit anderer Mapping-Logik unterscheiden sich in der Flanken-Ausgangs-Zuordnung.

Fazit

Der Funktionsblock `AX_ASR_RF_TRIG_X` bietet eine spezialisierte Flankenerkennung mit invertierter Ausgangszuordnung. Durch die Verwendung von Adaptern fügt er sich nahtlos in eine ad-hoc basierte 61499-Kommunikation ein und ermöglicht eine klare, wiederverwendbare Kapselung dieser Logik. Er eignet sich besonders für Szenarien, in denen die Standardzuordnung von `E_RF_TRIG` nicht dem gewünschten Verhalten entspricht.

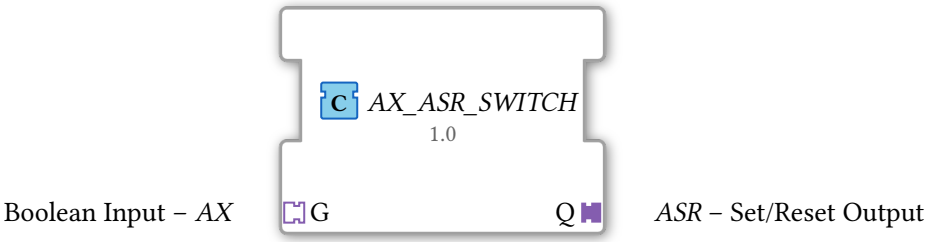
AX_ASR_SWITCH / AX_ASR_SWITCH_X



Einleitung

Diese Funktionsbausteine dienen als Brücke zwischen der datenorientierten AX-Welt und der ereignisorientierten ASR-Welt (Set/Reset). Sie wandeln ein boolesches Signal (verpackt in einen AX-Adapter) in dedizierte `SET` und `RESET` Ereignisse um.

Da die Richtung der Ereignisse (welcher Zustand setzt, welcher löscht) fest im Adapter verdrahtet ist, stehen zwei Varianten zur Verfügung.



Varianten

AX_ASR_SWITCH (1:1)

- Eingang `G = TRUE` (1) ➜ Ausgang `SET`
- Eingang `G = FALSE` (0) ➜ Ausgang `RESET`

AX_ASR_SWITCH_X (Gekreuzt)

- Eingang `G = TRUE` (1) ➜ Ausgang `RESET`
- Eingang `G = FALSE` (0) ➜ Ausgang `SET`

Schnittstellenstruktur

Adapter

- **G** (Socket): Eingangsadapter vom Typ `AX`.
- **Q** (Plug): Ausgangsadapter vom Typ `ASR`.

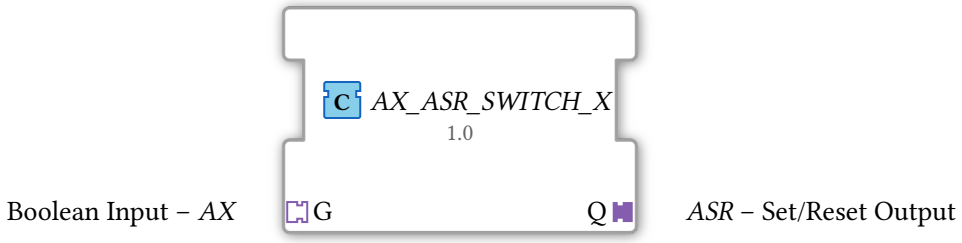
Funktionsweise

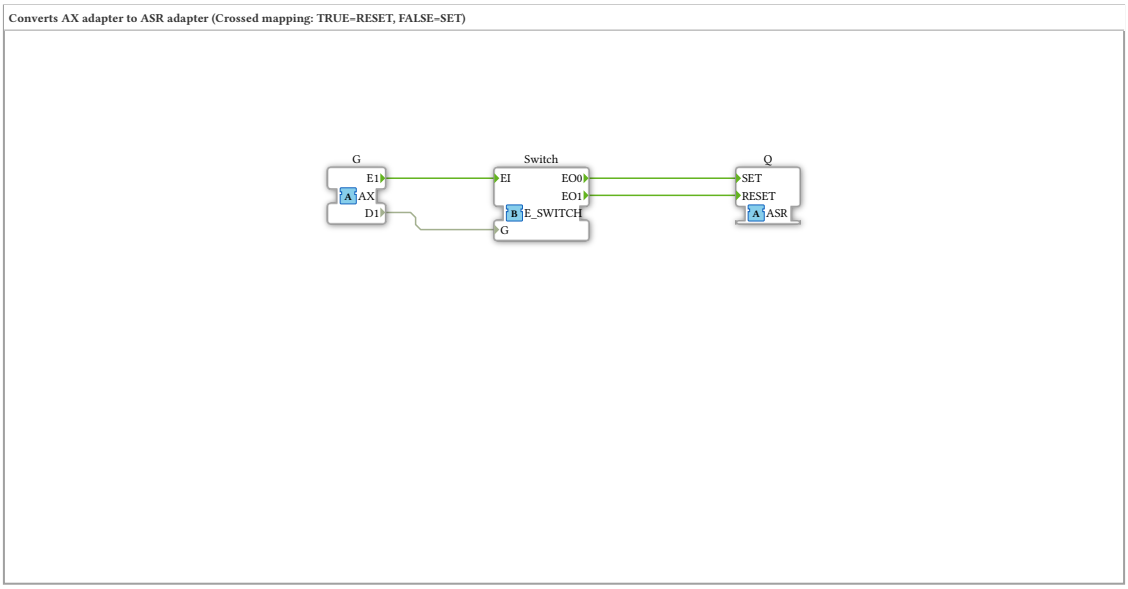
Die Bausteine nutzen intern einen `E_SWITCH`. Jedes Mal, wenn am Eingangsadapter `G` ein Ereignis (`E1`) eintrifft, wird der aktuelle Datenwert (`D1`) geprüft und das entsprechende Ereignis (`SET` oder `RESET`) am Ausgangsadapter `Q` ausgelöst.

Anwendungsszenarien

- Ansteuerung von bistabilen Bausteinen (Flip-Flops), die einen `ASR`-Eingang besitzen, direkt aus einem `AX`-Signalfluss heraus.
- Vereinfachung von Logikschaltungen, in denen Ereignisse basierend auf einem statischen Signalzustand generiert werden müssen.

AX_ASR_SWITCH_X





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AX_ASR_SWITCH_X** ist ein zusammengesetzter Baustein (Composite Function Block), der für die Konvertierung zwischen zwei verschiedenen Adaptertypen zuständig ist. Er wandelt Signale eines **AX**-Adapters (Boolean Event) in Signale eines **ASR**-Adapters (Asynchronous Set/Reset) um.

Das besondere Merkmal dieses Bausteins ist das im Suffix „X“ und im Kommentar angedeutete **„Crossed Mapping“ (gekreuzte Zuordnung)**. Im Gegensatz zu einer Standardzuordnung invertiert dieser Baustein die Logik: Ein `TRUE`-Signal am Eingang führt zu einem `RESET`-Ereignis am Ausgang, während ein `FALSE`-Signal ein `SET`-Ereignis auslöst.

Schnittstellenstruktur

Der Baustein interagiert hauptsächlich über Adapter-Schnittstellen.

Ereignis-Eingänge

Dieser Baustein verfügt über keine direkten Ereignis-Eingänge in der Schnittstellenliste, da diese innerhalb des Adapters `G` gekapselt sind.

Ereignis-Ausgänge

Dieser Baustein verfügt über keine direkten Ereignis-Ausgänge in der Schnittstellenliste, da diese innerhalb des Adapters `Q` gekapselt sind.

Daten-Eingänge

Dieser Baustein verfügt über keine direkten Daten-Eingänge in der Schnittstellenliste, da diese innerhalb des Adapters `G` gekapselt sind.

Daten-Ausgänge

Dieser Baustein verfügt über keine direkten Daten-Ausgänge.

Adapter

Name	Typ	Komponente	Kommentar
G	`adapter::types::unidirectional::AX`	Socket (Input)	Boolescher Eingangsadapter (enthält Ereignis `E1` und Daten `D1`).
Q	`adapter::types::unidirectional::ASR`	Plug (Output)	Set/Reset Ausgangsadapter (enthält Ereignisse `SET` und `RESET`).

Funktionsweise

Der **AX_ASR_SWITCH_X** nutzt intern einen Standard-`E_SWITCH`-Baustein (`iec61499::events::E_SWITCH`), um die Logik auszuführen. Die Verschaltung erfolgt wie folgt:

1. Ein Ereignis am Eingangsadapter **G** (Ereignis `E1`) löst die Verarbeitung aus.
2. Der Datenwert vom Eingangsadapter **G** (Datum `D1`) wird ausgewertet.
3. **Logik-Invertierung (Crossed Mapping):**
   - Ist **G.D1** = **TRUE**: Der interne Switch leitet das Ereignis auf seinen Ausgang `EO1`. Dieser ist mit **Q.RESET** verbunden.
   - Ist **G.D1** = **FALSE**: Der interne Switch leitet das Ereignis auf seinen Ausgang `EO0`. Dieser ist mit **Q.SET** verbunden.

Zusammenfassend:

- Input `TRUE` → Output `RESET`
- Input `FALSE` → Output `SET`

Technische Besonderheiten

- **Adapter-Konvertierung:** Brücke zwischen der einfachen booleschen Ereigniswelt (AX) und der Set/Reset-Steuerlogik (ASR).
- **Invertierte Logik:** Dieser Baustein ist spezifisch für Anwendungen konzipiert, bei denen ein aktives Signal (`TRUE`) eine Rücksetzung oder Deaktivierung (`RESET`) bewirken soll, und ein inaktives Signal (`FALSE`) eine Aktivierung (`SET`) – oder umgekehrt, je nach Interpretation der angeschlossenen Systeme.

Zustandsübersicht

Da es sich um einen Composite Function Block ohne eigene Zustandsmaschine (ECC) handelt, wird das Verhalten rein durch die Signaldurchleitung definiert.

Eingang G (D1) bei Event G (E1)	Ausgelöstes Ereignis bei Q	Logische Bedeutung
TRUE	**RESET**	Rücksetzen / Stoppen
FALSE	**SET**	Setzen / Starten

Anwendungsszenarien

- **Fail-Safe-Logik:** In Systemen, bei denen ein anliegendes Signal (High/True) den „Normalzustand“ oder „Ruhezustand“ (Reset) darstellt und das Wegfallen des Signals (Low/False) eine Aktion (Set) auslösen soll (Drahtbruchsicherheit).
- **Logik-Invertierung:** Anpassung von Sensoren, die „Active High“ liefern, an Aktoren oder Steuerbausteine, die „Active Low“ (bzw. Reset bei High) erwarten.
- **Spezifische Protokollanpassung:** Wenn ein Subsystem über einen AX-Adapter kommuniziert, das empfangende Subsystem jedoch einen ASR-Adapter mit invertierter Polarität benötigt.

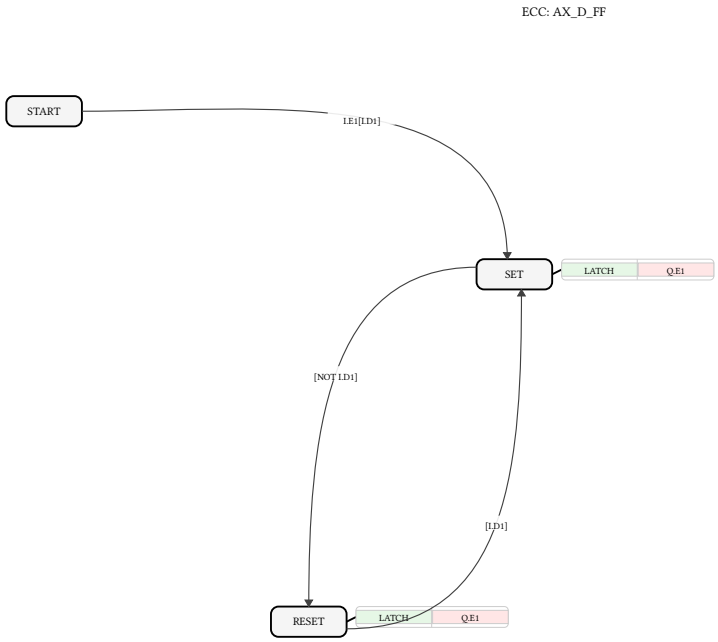
⚖️ Vergleich mit ähnlichen Bausteinen

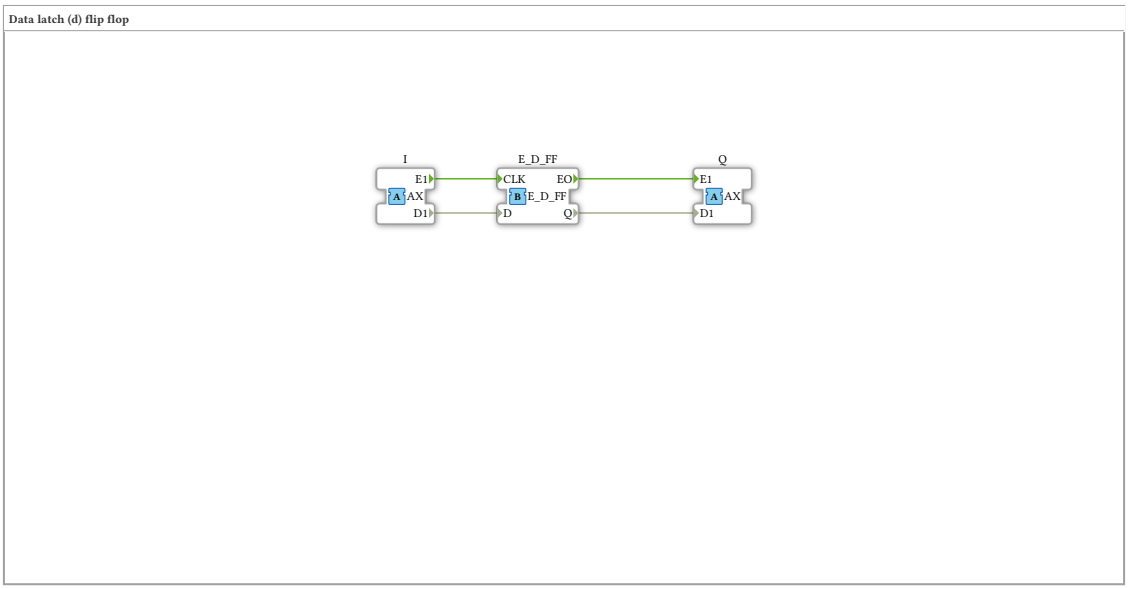
- **AX_ASR_SWITCH (hypothetisch/Standard):** Ein Standard-Wandler würde typischerweise `TRUE` auf `SET` und `FALSE` auf `RESET` mappen. Der hier beschriebene `_X` Baustein ist das genaue Gegenteil.
- **E_SWITCH:** Der zugrunde liegende IEC 61499 Standardbaustein, der die eigentliche Weichenstellung der Ereignisse basierend auf einem Boolean-Wert übernimmt, jedoch ohne die Kapselung in Adapter.

Fazit

Der **AX_ASR_SWITCH_X** ist ein spezialisierter Adapter-Konverter, der eine logische Invertierung („Crossed Mapping“) zwischen einem AX-Eingang und einem ASR-Ausgang vornimmt. Er ist essenziell für Systemintegrationen, bei denen die Signalpolarität zwischen Quelle und Senke umgekehrt werden muss, ohne zusätzliche diskrete Logikbausteine im Hauptnetzwerk zu platzieren.

AX_D_FF



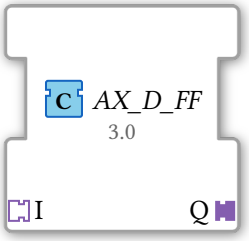


• — ○ • — ○ • — ○ •

Einleitung

Der AX_D_FF ist ein Daten-Latch-Flipflop, das als Basisbaustein in der digitalen Schaltungstechnik dient. Es handelt sich um ein D-Flipflop, das den anliegenden Datenwert speichert und bei bestimmten Ereignissen ausgibt.

Value to latch – *AX*



AX – Latched value

Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkten Ereigniseingänge vorhanden

Ereignis-Ausgänge

- Keine direkten Ereignisausgänge vorhanden

Daten-Eingänge

- Keine direkten Dateneingänge vorhanden

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden

Adapter

- **I** (Socket): Eingangsadapter vom Typ `adapter::types::unidirectional::AX` - Empfängt den zu speichernden Wert
- **Q** (Plug): Ausgangsadapter vom Typ `adapter::types::unidirectional::AX` - Gibt den gespeicherten Wert aus

Funktionsweise

Das AX_D_FF arbeitet als D-Latch-Flipflop mit drei Zuständen:

- **START**: Initialzustand
- **SET**: Speichert den Eingangswert und gibt ihn aus
- **RESET**: Setzt den Ausgang zurück

Der Algorithmus LATCH kopiert den Eingangswert `I.D1` auf den Ausgang `Q.D1`. Die Zustandsübergänge werden durch den Wert von `I.D1` gesteuert.

Technische Besonderheiten

- Verwendet unidirektionale Adapter für Ein- und Ausgänge
- Implementiert als Basic Function Block gemäß IEC 61499
- Besitzt eine einfache Latch-Funktionalität ohne Taktung

Zustandsübersicht

1. **START** → **SET**: Bei Eintreffen von `I.E1` mit `I.D1 = TRUE`
2. **SET** → **RESET**: Wenn `I.D1 = FALSE` wird
3. **RESET** → **SET**: Wenn `I.D1 = TRUE` wird

Anwendungsszenarien

- Datenspeicherung in Steuerungssystemen
- Zustandsspeicherung in sequenziellen Abläufen
- Signalverzögerung und -pufferung
- Als Grundbaustein für komplexere Flipflop-Schaltungen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu getakteten D-Flipflops arbeitet der AX_D_FF asynchron und speichert den Wert sofort bei Änderung der Eingangsbedingungen. Es handelt sich um ein level-getriggertes anstatt eines flankengetriggerten Elements.

Vergleich mit E_D_FF

🛠️ Zugehörige Übungen

- Uebung_170_AX

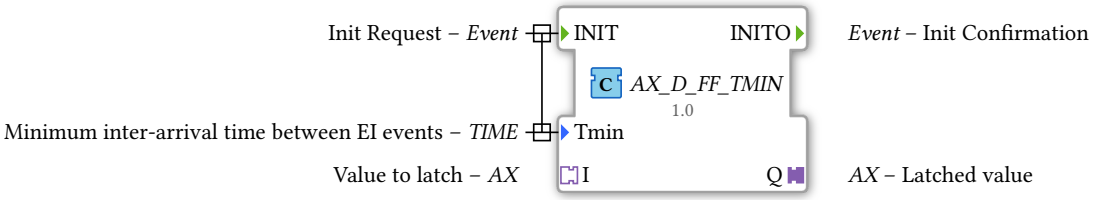
Fazit

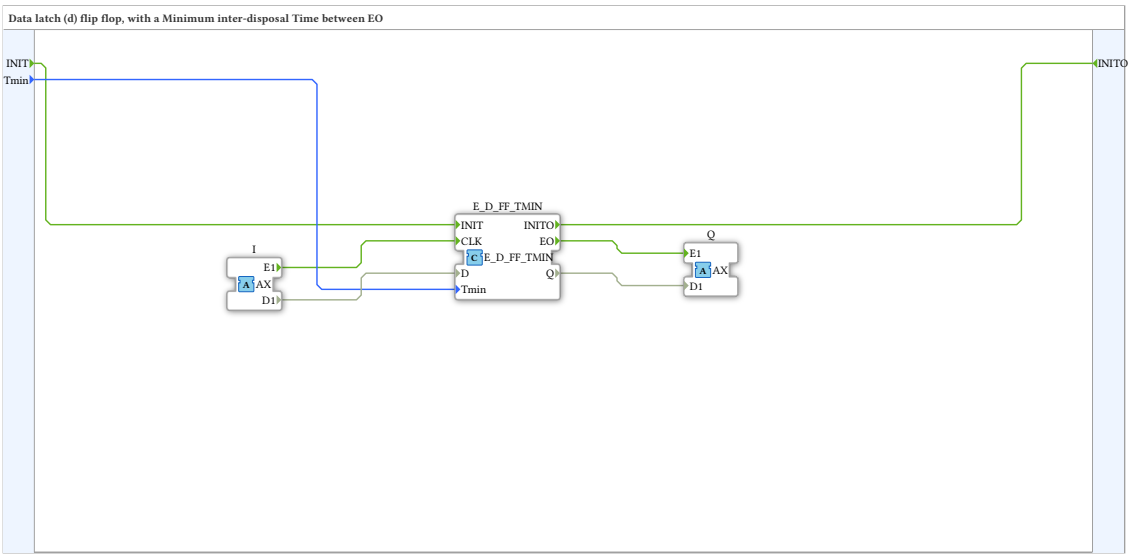
Der AX_D_FF bietet eine einfache und effektive Lösung für grundlegende Datenspeicheraufgaben in 4diac-Steuerungssystemen. Seine klare Zustandslogik und die Verwendung standardisierter Adapter machen ihn zu einem zuverlässigen Baustein für verschiedene Anwendungsfälle.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AX_D_FF_TMIN





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AX_D_FF_TMIN** realisiert ein data-latch D-Flip-Flop, das zusätzlich eine minimale Zeit zwischen aufeinanderfolgenden Ereignisausgängen (z.B. EO) erzwingt. Er dient als modulare Hülle um den IEC-61499-Standardbaustein `E_D_FF_TMIN` und verwendet einen unidirektionalen Adapter (`adapter::types::unidirectional::AX`) für die ereignisgesteuerte Datenübergabe. Der Baustein eignet sich besonders für Anwendungen, die eine zuverlässige Datenübernahme mit einstellbarem Mindestabstand zwischen Ausgangsimpulsen erfordern.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung
`INIT`	EInit	Initialisierungsanforderung; legt die minimale Verzögerungszeit fest (Parameter `Tmin`).

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung
`INITO`	EInit	Bestätigung der erfolgreichen Initialisierung.

Daten-Eingänge

Name	Typ	Beschreibung
`Tmin`	TIME	Minimale Sperrzeit (inter-disposal time) zwischen zwei aufeinanderfolgenden Ereignissen am Ausgang `EO`. Der Wert wird nur bei einem `INIT`-Ereignis übernommen.

Daten-Ausgänge

Der Baustein besitzt keine direkten Datenausgänge; der latchende Datenwert wird über den Adapter `Q` bereitgestellt.

Adapter

Rolle	Name	Typ	Beschreibung
Socket	`I`	adapter::types::unidirectional::AX	Eingangsadapter; liefert das zu latchende Datenwort (`D1`) und das Taktereignis (`E1`).
Plug	`Q`	adapter::types::unidirectional::AX	Ausgangsadapter; stellt das latchende Datenwort (`D1`) und das Ausgangsereignis (`E1`) zur Verfügung.

Die Adapter `AX` sind unidirektional ausgelegt: Sie übertragen genau ein Ereignis und einen zugehörigen Datenwert.

Funktionsweise

Der Baustein kapselt den internen Funktionsblock `E_D_FF_TMIN`, der wie folgt arbeitet:

1. **Initialisierung**: Wird ein `INIT`-Ereignis empfangen, wird die aktuelle Zeit `Tmin` als minimale Sperrzeit für den Ausgang `EO` gespeichert. Der Baustein quittiert dies mit `INITO`.
2. **Takten (D-Flip-Flop)**:
   - Ein Ereignis am Socket `I.E1` wird als Taktsignal (CLK) an das interne Flip-Flop weitergeleitet.
   - Gleichzeitig wird der Datenwert `I.D1` als neuer Eingangswert `D` übernommen.
   - Bei jeder steigenden Flanke des Takts (hier: jedes `E1`-Ereignis) wird der aktuelle Datenwert `D` übernommen und am Ausgang `Q` bereitgestellt.
   - Nach der Übernahme wird ein Ausgangsereignis `EO` am Plug `Q.E1` erzeugt – allerdings erst nach Ablauf der minimalen Sperrzeit seit dem letzten `EO`. Dadurch wird sichergestellt, dass Ausgangsimpulse nicht schneller als durch `Tmin` vorgegeben aufeinanderfolgen.
3. **Ausgang**: Der latchende Datenwert erscheint am Plug `Q.D1`, das zugehörige Ereignis am Plug `Q.E1`.

Technische Besonderheiten

- **Minimale Zwischenzeit**: Der Baustein erzwingt eine einstellbare Mindestzeit (`Tmin`) zwischen zwei Ausgangsereignissen. Überschreitet die Frequenz des Taktsignals diese Zeit, werden einzelne Taktereignisse unterdrückt, sodass der Ausgang eine maximal erlaubte Ereignisfolge nicht überschreitet.

- **Adapter-Schnittstelle**: Durch die Verwendung des unidirektionalen Adapters `AX` wird eine lose Kopplung zwischen Sender und Empfänger erreicht. Der Adapter transportiert sowohl das Ereignis als auch den Datenwert in einer logischen Einheit.
- **Initialisierungsabhängigkeit**: Die minimale Zeit `Tmin` wird nur einmal beim `INIT` gesetzt und bleibt bis zum nächsten `INIT` konstant. Eine Änderung während des Betriebs ist nicht vorgesehen.

Zustandsübersicht

Der Baustein besitzt keine explizite Zustandsmaschine. Die innere Logik des eingebetteten `E_D_FF_TMIN` kann jedoch folgende Zustände annehmen:

- **Idle**: Warten auf ein `INIT` oder ein Taktereignis.
- **Initialisiert**: `Tmin` wurde übernommen; der Baustein ist betriebsbereit.
- **Sperrzeit aktiv**: Nach einem Ausgangsereignis wird die durch `Tmin` festgelegte Verzögerung durchlaufen; eingehende Taktereignisse werden ignoriert.
- **Bereit**: Nach Ablauf der Sperrzeit kann das nächste Taktereignis ein neues Ausgangsereignis auslösen.

Anwendungsszenarien

- **Gepufferte Sensorwerte**: Ein Sensor liefert Daten mit unregelmäßiger Rate. Der `AX_D_FF_TMIN` übernimmt den letzten gültigen Wert und gibt ihn an eine nachfolgende Steuerung mit einer maximal zulässigen Ausgabefrequenz weiter.
- **Entprellung von Signalen**: Mechanische Taster oder Schalter erzeugen Prellimpulse. Die minimale Zwischenzeit unterdrückt zu schnelle Zustandswechsel.
- **Datenweiterleitung mit Bandbreitenbegrenzung**: In Kommunikationsnetzen kann der Baustein verwendet werden, um die Ausgaberate eines Datenkanals zu begrenzen, ohne dass der Absender adaptiert werden muss.

Vergleich mit ähnlichen Bausteinen

Baustein	Unterschied
`E_D_FF` (Standard-D-Flip-Flop)	Keine zeitliche Begrenzung der Ausgangsimpulse; Ereignisse werden sofort weitergeleitet.
`AX_D_FF_TMIN` (vorliegend)	Fügt die minimale Zwischenzeit `Tmin` zwischen Ausgangsereignissen ein, realisiert über den eingebauten `E_D_FF_TMIN`.
`E_CTD` / `E_CTU` (Zähler)	Zählen Ereignisse; bieten keine Datenübernahme oder Latch-Funktion.

Der `AX_D_FF_TMIN` vereint die latchende Datenübernahme eines Flip-Flops mit der zeitlichen Filterung eines Monoflops, ist aber als reiner Ereignis-Daten-Adapter ausgeführt.

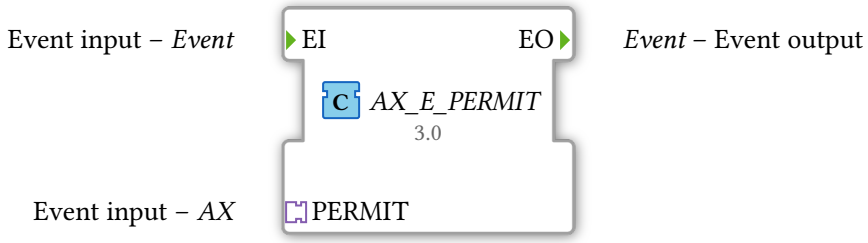
Fazit

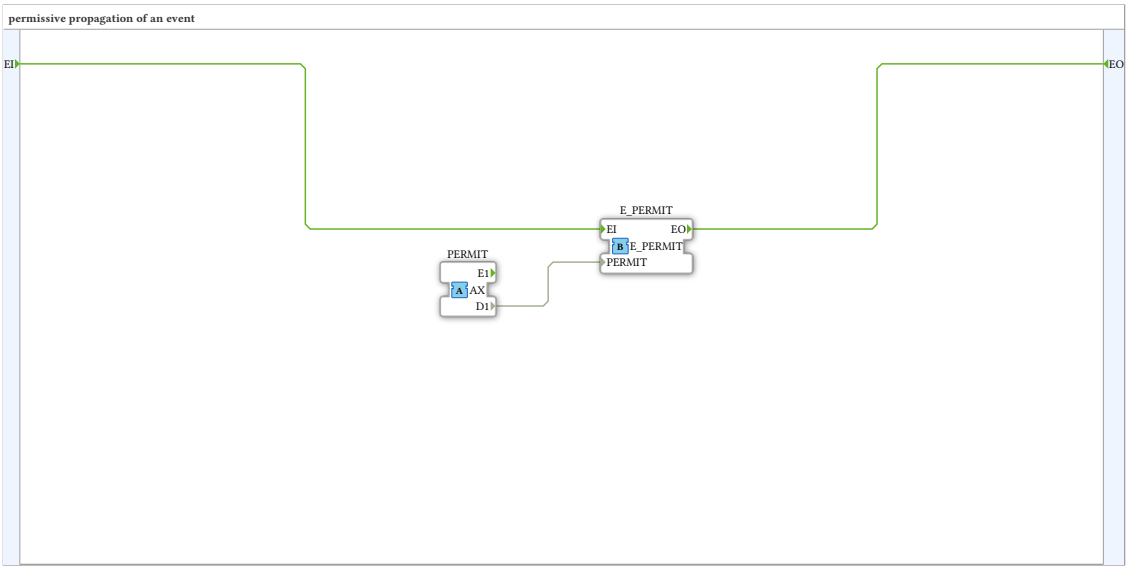
Der **AX_D_FF_TMIN** ist ein spezialisierter Funktionsblock für Anwendungen, die eine getaktete Datenübernahme mit garantierter Mindestzeit zwischen Ausgabeereignissen benötigen. Die Kapselung des Standardbausteins `E_D_FF_TMIN` und die Adapter-Schnittstelle erleichtern die Integration in modulare 4diac-Projekte. Der Baustein bietet eine saubere Trennung von Initialisierung und dynamischem Betrieb und eignet sich insbesondere für zeitkritische, ereignisgesteuerte Systeme.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AX_E_PERMIT





• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AX_E_PERMIT** realisiert eine bedingte Ereignisweitergabe (permissive propagation). Ein eingehendes Ereignis wird nur dann an den Ausgang weitergeleitet, wenn ein über einen Adapter bereitgestelltes Freigabesignal (Permit) den Wert `TRUE` besitzt. Der Baustein dient als Gate, das die Verarbeitungskette in Abhängigkeit von externen Bedingungen unterbrechen oder freigeben kann.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
EI	Event	Ereigniseingang

Ereignis-Ausgänge

Name	Typ	Kommentar
EO	Event	Ereignisausgang

Daten-Eingänge

Der Baustein besitzt keine eigenen Dateneingänge. Das Freigabesignal wird über einen Adapter bereitgestellt.

Daten-Ausgänge

Der Baustein besitzt keine Datenausgänge.

Adapter

Rolle	Name	Typ	Kommentar
Socket	PERMIT	`adapter::types::unidirectional::AX`	Liefert das Freigabesignal (D1)

Der Adapter ist unidirektional ausgelegt und stellt einen booleschen Wert (`D1`) zur Verfügung, der das Verhalten des Ereignisgates steuert.

Funktionsweise

Der Baustein enthält intern einen Standard-Funktionsblock `E_PERMIT` aus der IEC-61499-Bibliothek. Das eingehende Ereignis `EI` wird direkt an den internen `E_PERMIT`-Eingang weitergeleitet. Dessen Ausgang `EO` ist nur dann aktiv, wenn der zugehörige Freigabeeingang (`PERMIT` des internen Blocks) den Wert `TRUE` annimmt. Dieses Freigabesignal wird aus dem Adapter `PERMIT` über die Verbindung `PERMIT.D1` gespeist.

Somit gilt:

- **Permit = TRUE:** Das Ereignis `EI` wird unverzögert an `EO` weitergegeben.
- **Permit = FALSE:** Das Ereignis wird unterdrückt; `EO` bleibt inaktiv.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle:** Das Freigabesignal wird nicht als klassischer Dateneingang, sondern über einen Adapter (`AX`, unidirektional) bereitgestellt. Dies ermöglicht eine modulare Kopplung und Wiederverwendung der Freigabelogik.
- **Keine Datenweiterleitung:** Der Baustein gibt selbst keine Daten aus. Er dient ausschließlich der Steuerung des Ereignisflusses.
- **Unidirektionaler Adapter:** Der Adapter liefert nur einen Datenwert (D1) zum Baustein; eine Rückkopplung ist nicht vorgesehen.
- **Verwendung eines Standardbausteins:** Intern wird der bewährte `E_PERMIT`-Block eingesetzt, der die eigentliche Freigabelogik implementiert.

Zustandsübersicht

Der Baustein selbst besitzt keine sichtbaren Zustände. Das interne Verhalten wird durch den `E_PERMIT`-Block bestimmt, der im Wesentlichen eine passive Warteschleife darstellt:

- **Warten auf Ereignis:** Der Block wartet auf ein Ereignis am Eingang `EI`.
- **Prüfung des Permits:** Bei Eintreffen eines Ereignisses wird der Zustand des Freigabesignals ausgewertet.
- **Weiterleitung:** Nur wenn `PERMIT = TRUE` erfolgt die Ausgabe des Ereignisses an `EO`.

Eine explizite Zustandsmaschine ist nicht vorhanden; der Baustein arbeitet rein kombinatorisch auf Ereignisebene.

Anwendungsszenarien

- **Sicherheitsverriegelung:** Ein Ereignis (z. B. Start einer Maschine) wird nur dann weitergeleitet, wenn ein externer Sicherheitskreis ein Freigabesignal liefert.
- **Bedingte Verarbeitung:** In einer Produktionssteuerung kann der Datenfluss von Sensoren nur bei aktiviertem Betriebsmodus an die übergeordnete Steuerung gesendet werden.
- **Test- und Simulationsumgebungen:** Ein Ereignis kann über ein manuelles Freigabesignal gezielt ein- oder ausgeblendet werden, um bestimmte Szenarien zu testen.
- **Ressourcenschonung:** In einer komplexen Automatisierungsanlage können nicht benötigte Ereignisse durch Deaktivieren des Permits unterdrückt werden, um die Buslast zu reduzieren.

Vergleich mit ähnlichen Bausteinen

Baustein	Besonderheit
E_PERMIT	Standardblock mit direktem booleschem Eingang. Kein Adapter.
AX_E_PERMIT	Gleiche Logik, aber das Freigabesignal wird über einen unidirektionalen Adapter bereitgestellt, was die Kopplung über Subsysteme erleichtert.
E_SR (Set-Reset)	Ereignisbasierte Zustandsspeicherung; nicht direkt als Gate geeignet.

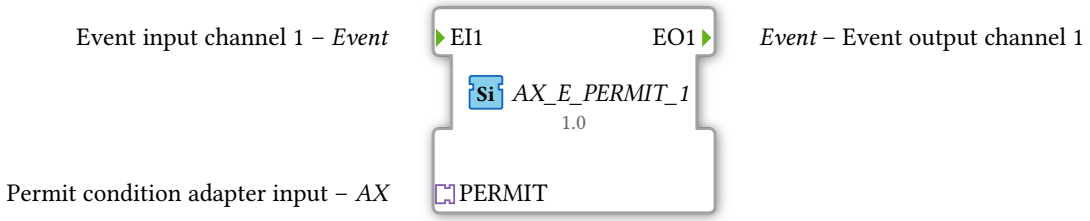
E_MUX	Ereignisweiche, die abhängig von einem Dateneingang zwischen zwei Ausgängen umschaltet – keine reine Sperrfunktion.

Der Vorteil von `AX_E_PERMIT` liegt in der sauberen Trennung der Freigabelogik durch einen Adapter, was die Wiederverwendung von Steuerungsmodulen fördert.

Fazit

Der Funktionsblock `AX_E_PERMIT` stellt eine flexible und modulare Lösung zur bedingten Ereignisweitergabe dar. Durch die Verwendung eines Adapters für das Freigabesignal wird die Schnittstelle klar definiert und kann über Hierarchiegrenzen hinweg eingesetzt werden. Der Baustein eignet sich besonders für sicherheitskritische oder konfigurierbare Steuerungsanwendungen, bei denen die Ereignisverarbeitung von externen Bedingungen abhängt.

AX_E_PERMIT_1



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AX_E_PERMIT_1` ist ein generischer Baustein zur kontrollierten Weiterleitung von Ereignissen. Er besitzt einen Ereignis-Eingang und einen Ereignis-Ausgang. Die Weitergabe eines Ereignisses von `EI1` zu `EO1` erfolgt nur dann, wenn die über einen Adapter definierte Bedingung (Freigabe) erfüllt ist. Der Baustein implementiert damit eine „permissive Propagation“: Ereignisse werden nur bei aktiver Erlaubnis durchgeschaltet, andernfalls verworfen. Die Bedingung wird über einen Adapter vom Typ `adapter::types::unidirectional::AX` extern bereitgestellt und kann je nach Anwendung flexibel gestaltet werden.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
`EI1`	Event	Ereignis-Eingangskanal 1

Ereignis-Ausgänge

Name	Typ	Kommentar
`EO1`	Event	Ereignis-Ausgangskanal 1

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

Name	Typ	Kommentar
`PERMIT`	`adapter::types::unidirectional::AX`	Adapter-Eingang für Freigabebedingung (Socket)

Funktionsweise

Der Baustein arbeitet rein ereignisbasiert. Er wartet auf ein Ereignis am Eingang `EI1`. Sobald ein Ereignis eintrifft, wird der Zustand des Adapters `PERMIT` ausgewertet. Liefert der Adapter eine „Erlaubnis“ (Freigabe), wird das Ereignis unmittelbar am Ausgang `EO1` weitergegeben. Ist die Bedingung nicht erfüllt, wird das Ereignis ignoriert und nicht weitergeleitet.

Die genaue Logik zur Bestimmung der Freigabe ist durch den angeschlossenen Adapter bestimmt. Der Adapter `AX` ist als unidirektionaler Typ definiert, d.h. er liefert ausschließlich eine boolesche oder vergleichbare Freigabesignal an den Funktionsblock. Der FB selbst enthält keine Datenverarbeitungslogik; er dient ausschließlich als steuerbares Ereignistor.

Technische Besonderheiten

- **Generischer Funktionsblock**: Der Baustein ist als generischer Typ (`GEN_AX_E_PERMIT`) implementiert und kann über den Adapter an verschiedene Freigabelogiken angepasst werden.
- **Keine Datenschnittstellen**: Der FB besitzt weder Daten-Eingänge noch Daten-Ausgänge. Die Kommunikation erfolgt ausschließlich über Ereignisse und den Adapter.
- **Unidirektionaler Adapter**: Der Adapter `PERMIT` ist als Socket deklariert, typisiert mit `adapter::types::unidirectional::AX`. Dadurch ist eine klare Trennung zwischen Freigabesignal und Ereignisfluss gegeben.
- **Einfache Ereignisweiterleitung**: Die Implementierung folgt dem Muster einer bedingten Ereignisübertragung, was den Baustein leicht in sicherheitsgerichtete oder ablaufsteuernde Systeme integrierbar macht.

Zustandsübersicht

Da der Baustein keine eigenen Zustände speichert, sondern jede Ereignisinstanz unabhängig prüft, lässt sich keine klassische Zustandsmaschine darstellen. Die Funktionsweise ist rein ereignisorientiert:

- **Warte auf Ereignis**: Kein Ereignis am Eingang, keine Aktivität.
- **Ereignis eingetroffen**: Prüfung der Adapterbedingung.
  - Bedingung erfüllt ⟶ Weiterleitung über `EO1`.
  - Bedingung nicht erfüllt ⟶ Verwerfen des Ereignisses.

Somit existiert lediglich ein impliziter, ereignisabhängiger Ablauf ohne dauerhafte Zustände.

Anwendungsszenarien

Der Baustein eignet sich für alle Systeme, in denen Ereignisse nur bei Vorliegen einer externen Freigabe wirksam werden dürfen. Beispiele:

- **Sicherheitskreise**: Not-Aus-Signale werden nur dann an die Steuerung weitergegeben, wenn eine Betriebsfreigabe aktiv ist.
- **Zugangssteuerungen**: Türen oder Schranken lassen nur bei erteilter Zugangsberechtigung (über Adapter) ein Ereignis (z.B. „Schließen“) zu.
- **Produktionsanlagen**: Ereignisse wie „Start“ oder „Stopp“ werden nur bei freigegebenem Prozessmodul durchgeschaltet.
- **Testumgebungen**: Gezieltes Filtern von Ereignissen während des Testbetriebs durch einen separaten Freigabeadapter.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem einfachen Ereignis-Durchschaltbaustein (z.B. `E_PERMIT`) bietet `AX_E_PERMIT_1` die Möglichkeit, die Freigabebedingung über einen Adapter flexibel extern zu definieren. Dadurch werden fest verdrahtete Bedingungslaogiken zu Gunsten einer modularen, wiederverwendbaren Architektur aufgelöst.

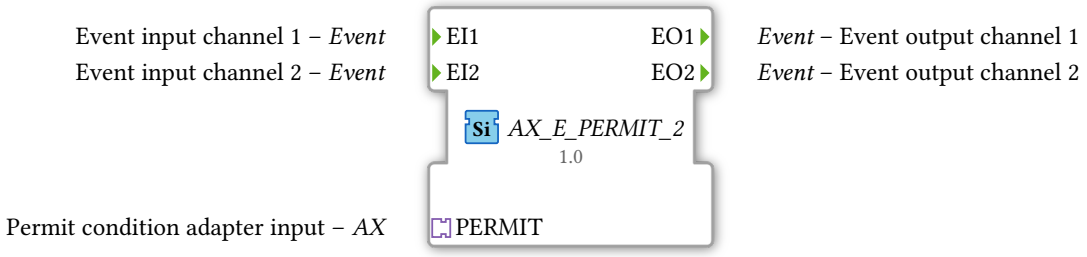
Gegenüber Bausteinen mit Datenfiltern (z.B. `E_CYCLE` oder `E_SWITCH`) fokussiert dieser FB ausschließlich auf die Ereignisfreigabe, ohne Daten zu manipulieren oder zeitliche Aspekte zu berücksichtigen. Der unidirektionale Adapter vereinfacht die Anbindung an vorhandene Freigabesignale, ohne zusätzliche Verdrahtungsaufwand für Rückkanäle.

Fazit

`AX_E_PERMIT_1` ist ein schlanker, generischer Funktionsblock zur bedingten Ereignisweiterleitung. Durch die Verwendung eines Adapters bleibt die Freigabelogik austauschbar und anpassbar. Die klare Beschränkung auf Ereignisse und den Adapter macht den Baustein zu einem vielseitigen Werkzeug für sicherheitsrelevante und steuerungstechnische Anwendungen, bei denen eine Erlaubnisprüfung vor der

Weitergabe von Ereignissen erforderlich ist. Seine einfache Struktur und die generische Konfiguration ermöglichen eine schnelle Integration in bestehende 4diac-Umgebungen.

AX_E_PERMIT_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AX_E_PERMIT_2** dient zur permissiven Weiterleitung von zwei Ereigniskanälen. Er ist als generischer Baustein konzipiert und verwendet einen Adapter, um die Ausführung der Ereignisse an eine externe Bedingung zu koppeln. Die Ereignisse EI1 und EI2 werden nur dann an die Ausgänge EO1 und EO2 weitergegeben, wenn der Adapter die Freigabe erteilt. Dadurch eignet sich der Baustein zur Steuerung von Abläufen in IEC 61499-Applikationen, bei denen Ereignisse gezielt unterbrochen oder blockiert werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

- **EI1** (Event): Erster Ereignis-Eingang, der bei Freigabe an den Ausgang EO1 weitergeleitet wird.
- **EI2** (Event): Zweiter Ereignis-Eingang, der bei Freigabe an den Ausgang EO2 weitergeleitet wird.

Ereignis-Ausgänge

- **EO1** (Event): Ereignis-Ausgang für den Kanal 1, aktiviert nach Freigabe des Adapters.
- **EO2** (Event): Ereignis-Ausgang für den Kanal 2, aktiviert nach Freigabe des Adapters.

Daten-Eingänge

Es sind keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Es sind keine Daten-Ausgänge vorhanden.

Adapter

- **PERMIT** (Socket, Typ `adapter::types::unidirectional::AX`): Dieser Adapter stellt die Freigabebedingung bereit. Nur wenn das Signal des Adapters aktiv ist, werden die Ereignisse durchgeschaltet. Die genaue Logik wird durch den konkreten Adapter-Typ `AX` bestimmt, der eine unidirektionale Kommunikation ermöglicht.

Funktionsweise

Der Funktionsblock arbeitet als ereignisgesteuerter Pfad mit Freigabe durch den Adapter. Im Normalzustand wartet er auf Ereignisse an EI1 oder EI2. Sobald ein Ereignis eintrifft, wird der Zustand des Adapters PERMIT überprüft. Ist die Bedingung erfüllt (Freigabe aktiv), wird das entsprechende Ereignis an EO1 bzw. EO2 weitergeleitet. Bei nicht aktivem Freigabesignal wird das Ereignis verworfen und es erfolgt keine Ausgabe.

Der Baustein ist generisch aufgebaut, d.h. die eigentliche Freigabelogik wird über den Adaptertyp `AX` und dessen Parameter realisiert. Dies ermöglicht eine flexible Anpassung an verschiedene Anforderungen, ohne die interne Struktur des Bausteins zu verändern.

Technische Besonderheiten

- **Generische Implementierung**: Durch die Nutzung des Adapters und der Eigenschaft `GenericClassName = 'GEN_AX_E_PERMIT'` ist der Baustein als generischer FB ausgelegt. Er kann mit unterschiedlichen Adapter-Instanzen des Typs `AX` konfiguriert werden.
- **Unidirektionale Adapterkommunikation**: Der Adapter ist vom Typ `adapter::types::unidirectional::AX`, was eine einfache und effiziente Signalübertragung von der aufrufenden zur aufgerufenen Seite ermöglicht.
- **Keine Datenvariablen**: Der Baustein besitzt weder Daten-Ein- noch -Ausgänge, wodurch er sich auf reine Ereignissteuerung beschränkt und keine Datenhaltung erfordert.
- **Skalierbarkeit**: Aufgrund des modularen Aufbaus kann der Baustein leicht erweitert oder angepasst werden, falls mehrere Ereigniskanäle benötigt werden (analog zu `AX_E_PERMIT`).

Zustandsübersicht

Der Funktionsblock besitzt keine expliziten internen Zustände. Sein Verhalten ist rein kombinatorisch: Die Ausgabe hängt ausschließlich vom aktuellen Eingangsereignis und dem Zustand des Adapters ab. Eine Zustandsänderung erfolgt nur durch den Adapter, der außerhalb des Bausteins definiert wird.

Anwendungsszenarien

- **Steuerung von Ereignisflüssen**: Einsatz in Produktionsanlagen, bei denen Maschinenbewegungen oder Prozessschritte nur bei erfüllter Sicherheitsbedingung ausgeführt werden dürfen.
- **Bedingte Verarbeitung**: In Automatisierungssystemen kann der Baustein verwendet werden, um Ereignisse nur dann zu verarbeiten, wenn eine übergeordnete Steuerung dies erlaubt (z.B. bei manuellen Eingriffen).
- **Resourcenverwaltung**: Bei der Parallelausführung mehrerer Tasks kann der Baustein als Gate dienen, um Ressourcenkonflikte zu vermeiden.

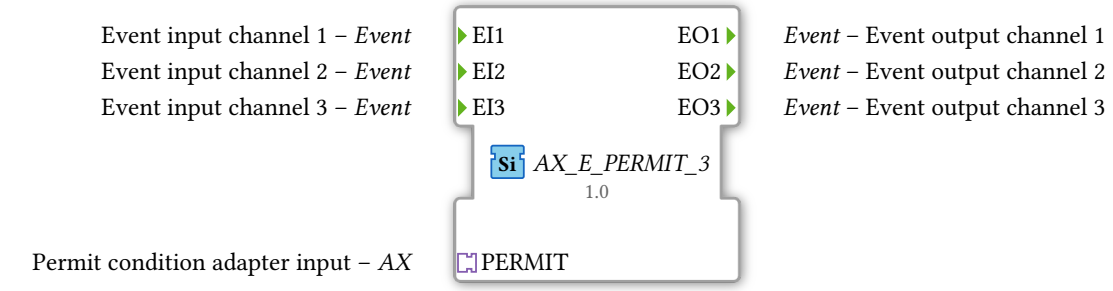
Vergleich mit ähnlichen Bausteinen

- **AX_E_PERMIT** (ohne Suffix): Der Baustein `AX_E_PERMIT` ist die Variante für einen einzelnen Ereigniskanal. `AX_E_PERMIT_2` erweitert diese Funktionalität auf zwei unabhängige Kanäle, wobei beide über denselben Adapter freigegeben werden.
- **Event-Propagations-Bausteine**: Im Gegensatz zu einfachen Weiterleitungs-Bausteinen (z.B. `E_DEMUX`) besitzt `AX_E_PERMIT_2` eine zusätzliche Freigabe-Logik. Dadurch ist er besser für sicherheitskritische Anwendungen geeignet.
- **Adapterbasierte Bausteine**: Andere Bausteine mit Adaptern bieten oft eine bidirektionale Kommunikation; dieser Baustein nutzt bewusst eine unidirektionale Schnittstelle, was die Implementierung vereinfacht.

Fazit

Der Funktionsblock `AX_E_PERMIT_2` ist ein flexibler und kompakter Baustein zur bedarfsgesteuerten Ereignisweiterleitung. Durch seine generische Adapter-Schnittstelle lässt er sich in vielfältigen industriellen Steuerungsumgebungen einsetzen. Die Beschränkung auf zwei Kanäle und den unidirektionalen Adaptertyp macht ihn besonders transparent und leicht wartbar. Er stellt eine effiziente Lösung dar, um Ereignisse gezielt zu blockieren oder zu ermöglichen, und trägt zur robusten Gestaltung von IEC 61499-Applikationen bei.

AX_E_PERMIT_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AX_E_PERMIT_3` dient der selektiven Weiterleitung von drei unabhängigen Ereigniskanälen unter einer gemeinsamen Freigabebedingung. Er ist als generischer Baustein ausgelegt und verwendet einen unidirektionalen Adapter, um das Freigabesignal zu empfangen. Nur wenn die Freigabe aktiv ist, werden eingehende Ereignisse an die entsprechenden Ausgänge durchgereicht; andernfalls werden sie ignoriert. Dadurch eignet sich der Baustein für Anwendungen, bei denen Ereignisströme zentral gesperrt oder ermöglicht werden sollen.

Schnittstellenstruktur

Ereignis-Eingänge

- **EI1**: Event input channel 1
- **EI2**: Event input channel 2
- **EI3**: Event input channel 3

Ereignis-Ausgänge

- **EO1**: Event output channel 1
- **EO2**: Event output channel 2
- **EO3**: Event output channel 3

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

- **PERMIT**: Typ `adapter::types::unidirectional::AX`
  Kommentar: „Permit condition adapter input“
  Dieser unidirektionale Adapter liefert die Freigabebedingung in den Baustein hinein. Typischerweise handelt es sich um ein boolesches Signal (TRUE = freigeben, FALSE = sperren).

Funktionsweise

Der Baustein überwacht kontinuierlich den Wert am Adapter-Eingang `PERMIT`. Sofern an diesem Eingang eine aktive Freigabe (z. B. TRUE) anliegt, werden ankommende Ereignisse an den jeweiligen Ausgangskanal weitergegeben:

- Ein Ereignis an `EI1` wird dann an `EO1` durchgereicht.
- Ein Ereignis an `EI2` wird dann an `EO2` durchgereicht.
- Ein Ereignis an `EI3` wird dann an `EO3` durchgereicht.

Liegt keine Freigabe vor, werden alle Ereignisse verworfen und es erfolgt keinerlei Ausgabe. Die drei Kanäle arbeiten unabhängig voneinander, die Freigabe gilt jedoch für alle gleichzeitig. Es gibt keine interne Pufferung oder zeitliche Verzögerung; die Weitergabe erfolgt unmittelbar bei Eintreffen eines Ereignisses, sofern die Bedingung erfüllt ist.

Technische Besonderheiten

- Der Baustein ist als **generischer Funktionsblock** definiert (Attribut `eclipse4diac::core::GenericClassName = 'GEN_AX_E_PERMIT'`). Dadurch kann er je nach Kontext mit verschiedenen konkreten Adaptertypen instanziiert werden, solange diese das in der Adapterdefinition festgelegte Interface erfüllen.
- Es werden keine Daten-Eingänge oder -Ausgänge verwendet – ausschließlich Ereignisse und ein Adapter. Dies reduziert die Komplexität und macht den Baustein für reine Ereignissteuerung geeignet.
- Der Adapter ist **unidirektional**, d.h. er überträgt nur Daten in den Baustein hinein; eine Rückkopplung oder ein bidirektionaler Datenaustausch ist nicht vorgesehen.
- Die Schnittstelle ist auf drei Ereigniskanäle ausgelegt, was typische Anforderungen an eine Mehrfachfreigabe in Automatisierungssystemen abdeckt.

Zustandsübersicht

Da der Baustein rein ereignisgesteuert ist, besitzt er keine klassischen internen Zustände im Sinne eines Moore- oder Mealy-Automaten. Das Verhalten wird direkt durch den Pegel am Adapter `PERMIT` bestimmt:

- **Freigabe aktiv** (z.B. TRUE): Ereignisse werden transparent durchgeschaltet.
- **Freigabe inaktiv** (z.B. FALSE): Ereignisse werden blockiert und verworfen.

Ein Umspringen der Freigabe hat keine verzögerte Wirkung – die aktuelle Bedingung gilt sofort für alle nachfolgenden Ereignisse.

Anwendungsszenarien

- **Sicherheitsgerichtete Steuerungen**: Ereignisse (z.B. Not-Aus-Befehle) dürfen nur bei gesetztem Freigabesignal weitergeleitet werden.
- **Modulare IEC 61499-Anwendungen**: Kopplung von Funktionsblöcken, wenn die Weitergabe von Ereignissen an eine übergeordnete Berechtigung gebunden ist.
- **Test- und Simulationssysteme**: Leichtes Ein- und Ausschalten von Ereignispfaden zu diagnostischen Zwecken.
- **Mehrkanalige Verarbeitung**: Wenn mehrere parallele Abläufe gleichzeitig freigegeben oder gesperrt werden sollen.

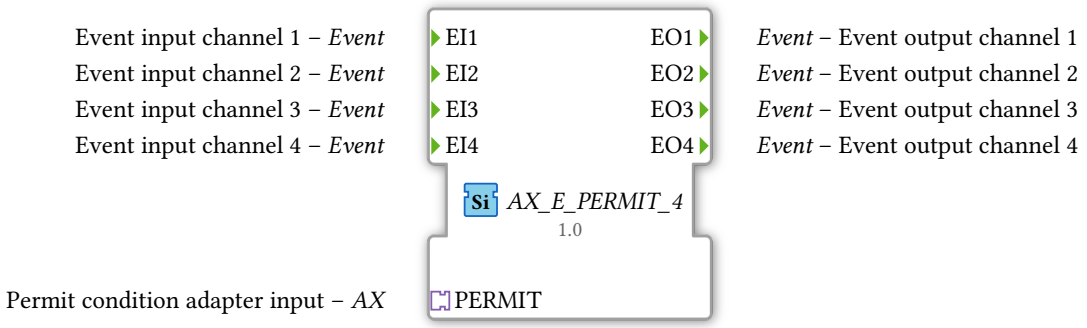
Vergleich mit ähnlichen Bausteinen

Gegenüber einem einfachen `E_PERMIT` (das nur einen Ereigniskanal bedient) erweitert `AX_E_PERMIT_3` die Funktionalität auf drei Kanäle, wobei die Freigabe zentral über einen Adapter erfolgt. Im Unterschied zu einem `E_SWITCH` oder Multiplexer wählt der Baustein nicht zwischen verschiedenen Quellen, sondern unterbricht die Signalweitergabe vollständig – es findet keine Umschaltung, sondern nur eine Freigabe/Sperre statt. Dadurch ist er ideal für Anwendungen, bei denen eine strikte Unterbrechung der Ereignisfolge gefordert ist. Die Adapter-basierte Anbindung der Freigabe unterscheidet ihn von Bausteinen mit direktem boolschen Eingang und erhöht die Flexibilität bei modularen Designs.

Fazit

`AX_E_PERMIT_3` ist ein kompakter und wiederverwendbarer Funktionsbaustein zur zentralen Freigabe dreier Ereigniskanäle. Die Verwendung eines unidirektionalen Adapters entkoppelt die Freigabelogik vom eigentlichen Ereignisfluss und ermöglicht eine saubere Modellierung in IEC 61499-Systemen. Durch den generischen Aufbau lässt sich der Baustein in unterschiedlichen Kontexten einsetzen, ohne dass die Kernlogik verändert werden muss. Damit stellt er ein praktisches Werkzeug für alle Szenarien dar, in denen Ereignisse nur unter bestimmten Bedingungen wirksam werden dürfen.

AX_E_PERMIT_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AX_E_PERMIT_4** dient zur bedingten Weitergabe von vier unabhängigen Ereigniskanälen. Er besitzt jeweils vier Ereignis-Eingänge (EI1–EI4) und vier Ereignis-Ausgänge (EO1–EO4). Die Durchschaltung der Ereignisse wird über einen einzigen Adapter-Eingang (PERMIT) gesteuert. Dieser Adapter liefert ein Freigabesignal (permit), das als Bedingung für die Weiterleitung aller Ereignisse wirkt. Der Baustein ist als generischer FB (Generic FB) ausgelegt und kann in der 4diac-IDE mittels des generischen Typs `GEN_AX_E_PERMIT` instanziiert werden.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Beschreibung
EI1	Ereignis-Eingangskanal 1
EI2	Ereignis-Eingangskanal 2
EI3	Ereignis-Eingangskanal 3
EI4	Ereignis-Eingangskanal 4

Ereignis-Ausgänge

Name	Beschreibung
EO1	Ereignis-Ausgangskanal 1
EO2	Ereignis-Ausgangskanal 2
EO3	Ereignis-Ausgangskanal 3
EO4	Ereignis-Ausgangskanal 4

Daten-Eingänge

Der Baustein besitzt keine Daten-Eingänge.

Daten-Ausgänge

Der Baustein besitzt keine Daten-Ausgänge.

Adapter

Name	Typ	Richtung	Beschreibung
PERMIT	adapter::types::unidirectional::AX	Socket (Eingang)	Freigabebedingung für die Ereignisweiterleitung.

Funktionsweise

Der Baustein überwacht den über den Adapter `PERMIT` ankommenden Zustand. Solange das Signal den Wert „freigegeben“ (permit) hat, werden eingehende Ereignisse nahezu verzögerungsfrei auf den jeweils korrespondierenden Ausgang durchgeschaltet (EI1 → EO1, EI2 → EO2, …). Ist das Signal nicht aktiv, werden eingehende Ereignisse ignoriert und nicht an die Ausgänge weitergegeben.

Es findet keine Verarbeitung oder Umkodierung der Ereignisse statt – die Kopplung zwischen Eingang und Ausgang ist strikt 1:1. Die Bedingung wird global für alle vier Kanäle gleichzeitig ausgewertet; eine individuelle Freigabe pro Kanal ist nicht möglich.

Technische Besonderheiten

- **Generischer FB**: Der Baustein ist als generischer Typ deklariert (`eclipse4diac::core::GenericClassName = 'GEN_AX_E_PERMIT'`). Dadurch kann er in der 4diac-IDE flexibel parametriert und an unterschiedliche Adaptertypen angepasst werden.
- **Adapter-Prinzip**: Die Freigabebedingung wird über einen unidirektionalen Adapter (Socket) realisiert. Dies erlaubt eine klare Trennung zwischen Ereignisfluss und Steuerfluss.
- **Keine Datenkanäle**: Es sind weder Daten-Eingänge noch Daten-Ausgänge vorhanden – die gesamte Logik reduziert sich auf reine Ereignisweiterleitung mit einer binären Bedingung.
- **Skalierbarkeit**: Durch die Verwendung von vier Kanälen eignet sich der Baustein für Anwendungen, bei denen mehrere unabhängige Ereignisströme gleichzeitig überwacht und gefiltert werden müssen.

Zustandsübersicht

Da der Baustein keine explizite Zustandsmaschine besitzt, kann er als ereignisgesteuert mit internem Schwellwert (permit) beschrieben werden. Es gibt keine persistenten Zustände; die Funktionsweise ist

rein kombinatorisch – das permit-Signal bestimmt unmittelbar, ob ein Ereignis durchgelassen wird. Eine Zustandsübersicht im herkömmlichen Sinne ist daher nicht erforderlich.

Anwendungsszenarien

Typische Einsatzbereiche sind:

- **Steuerungssysteme**: Durchschaltung von Alarm- oder Statusereignissen nur bei erfolgreicher Freigabe (z.B. durch übergeordnete Sicherheitslogik).
- **Sicherheitsgerichtete Kommunikation**: Puffern oder Blockieren von Ereignisketten in Abhängigkeit von Betriebszuständen (z.B. Stillstand, Not-Aus).
- **Test- und Simulationsumgebungen**: Gezieltes Ein-/Ausschalten von Ereignispfaden zu Testzwecken.
- **Modulare Automatisierung**: Einheitliche Ereignisweiterleitung in größeren Funktionsblöcken, wenn mehrere Signalquellen nur bei aktiver Bedingung durchgereicht werden dürfen.

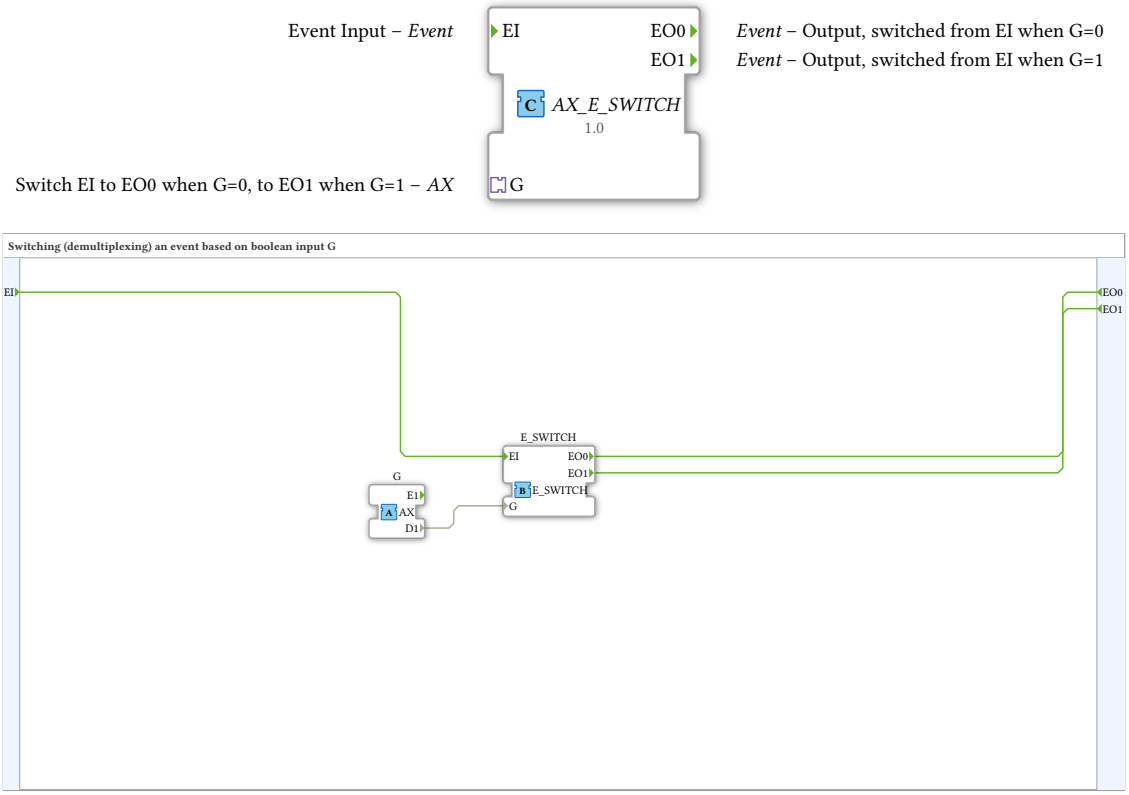
Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen Weiterleitungsbausteinen wie `E_PERMIT` (der nur einen Kanal besitzt) bietet `AX_E_PERMIT_4` eine Mehrkanalausführung mit gemeinsamer Freigabe. Im Gegensatz zu Bausteinen ohne Freigabebedingung (z.B. `E_SPLIT` oder `E_MERGE`) wird hier eine zusätzliche Steuerebene hinzugefügt, die die Ereignisübertragung konditioniert. Gegenüber einem Aufbau aus vier separaten `E_PERMIT`-Blöcken reduziert dieser FB die Anzahl benötigter Instanzen und erleichtert die Synchronisierung der Freigabe.

Fazit

Der Funktionsblock **AX_E_PERMIT_4** ist ein kompaktes und effizientes Werkzeug zur bedingten Weitergabe von vier unabhängigen Ereignisströmen. Durch die Integration eines Adapters für das Freigabesignal und die generische Auslegung eignet er sich besonders für modulare und sicherheitsbewusste Automatisierungslösungen. Seine einfache Struktur ohne Datenpfade erlaubt eine klare, zuverlässige Ereignissteuerung und kann leicht in bestehende 4diac-Projekte integriert werden.

AX_E_SWITCH



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AX_E_SWITCH** realisiert ein ereignisgesteuertes Demultiplexing (Weiche) auf Basis eines booleschen Signals. Er empfängt ein Ereignis am Eingang **EI** und leitet es abhängig vom Wert des Adapters **G** entweder an den Ausgang **EO0** oder **EO1** weiter. Dadurch lassen sich Ereignisflüsse in Abhängigkeit von Zuständen oder Entscheidungen aufteilen.

Schnittstellenstruktur

Ereignis-Eingänge

Bezeichnung	Typ	Beschreibung
EI	Event	Ereigniseingang, der umgeschaltet wird

Ereignis-Ausgänge

Bezeichnung	Typ	Beschreibung
EO0	Event	Wird aktiviert, wenn **G** = 0
EO1	Event	Wird aktiviert, wenn **G** = 1

Daten-Eingänge

Der Baustein besitzt keine separaten Dateneingänge. Die Schaltinformation wird über den Adapter **G** bereitgestellt.

Daten-Ausgänge

Keine Datenausgänge vorhanden.

Adapter

Bezeichnung	Typ	Beschreibung
G	`adapter::types::unidirectional::AX`	Boolesche Schnittstelle; liefert den Wert für die Umschaltung (0 ⟶ EO0, 1 ⟶ EO1)

Funktionsweise

Der **AX_E_SWITCH** ist intern als Kapselung des Standardbausteins **E_SWITCH** realisiert. Ein eingehendes Ereignis an **EI** wird an den internen Baustein weitergeleitet. Der Adapter **G** liefert den booleschen Wert (über die Datenverbindung `G.D1` ⟶ `E_SWITCH.G`).

- Bei **G = 0** wird das Ereignis an **EO0** ausgegeben.
- Bei **G = 1** wird das Ereignis an **EO1** ausgegeben.
  Der Baustein arbeitet deterministisch und ohne interne Zustandshaltung; die Umschaltung erfolgt in einem einzigen Ausführungsschritt.

Technische Besonderheiten

- **Adapter-Schnittstelle:** Der boolesche Schaltwert wird nicht als klassischer Dateneingang, sondern über einen Adapter bereitgestellt. Dies ermöglicht eine lose Kopplung und Wiederverwendung standardisierter Schnittstellen (Typ `unidirectional::AX`).
- **Wiederverwendung:** Intern wird der IEC-61499-Standardbaustein `E_SWITCH` verwendet, was Kompatibilität und einfache Nachvollziehbarkeit sicherstellt.
- **Keine Nebenwirkungen:** Der Baustein ist rein komponentenbasiert und verändert keine globalen Daten oder Zustände.

Zustandsübersicht

Der **AX_E_SWITCH** besitzt keinen eigenen expliziten Zustandsautomaten. Das Verhalten ist rein ereignisgesteuert und unterscheidet sich nicht von einer einfachen Weiche. Eine Zustandsübersicht entfällt daher.

Anwendungsszenarien

- **Ereignisweiche in Steuerungslogiken:** Aufteilen eines Ereignisstroms auf zwei Pfade, z. B. abhängig von einem Sensorwert oder einer Betriebsart.
- **Qualitätskontrolle:** Weiterleitung eines Ereignisses an unterschiedliche Verarbeitungsblöcke je nach Prüfergebnis (Gut/Schlecht).
- **Betriebsmodus-Umschaltung:** Steuerung von Abläufen, die im Normal- oder Störungsmodus unterschiedlich reagieren.

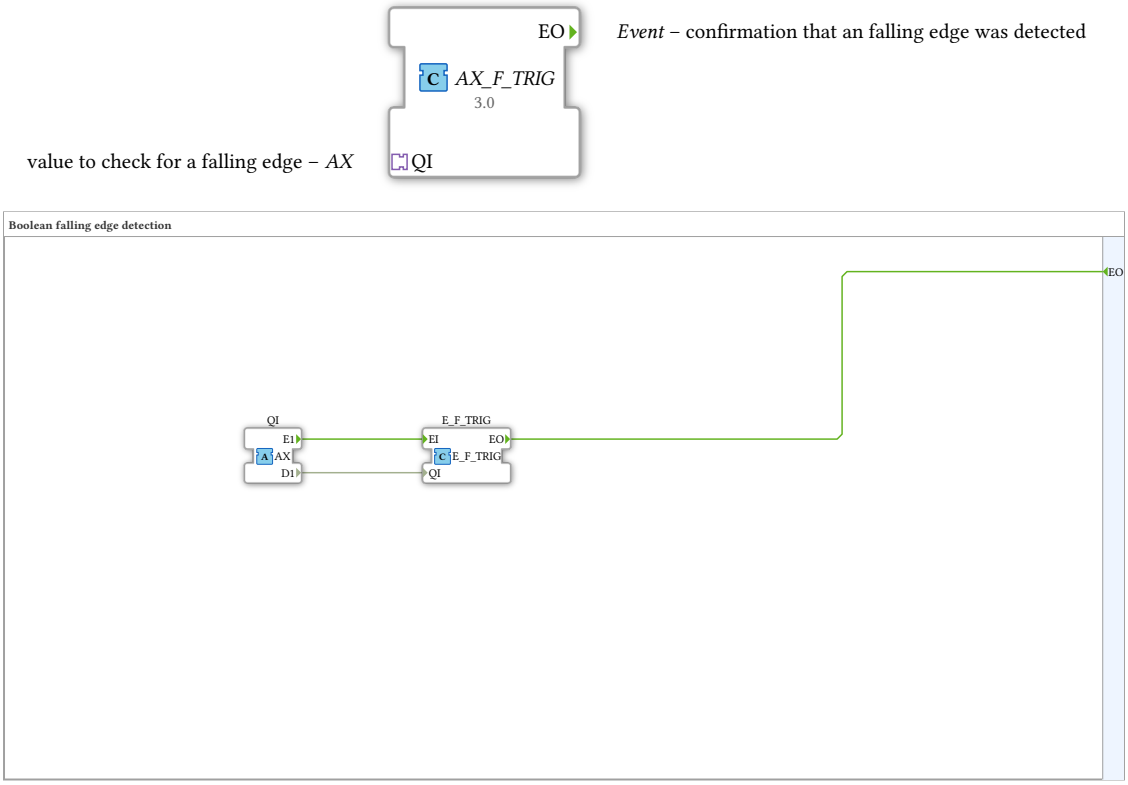
Vergleich mit ähnlichen Bausteinen

- **Standardbaustein `E_SWITCH`:** Besitzt einen direkten booleschen Dateneingang anstelle eines Adapters. Der **AX_E_SWITCH** kapselt diesen und bietet eine adapterbasierte Schnittstelle, die in modularen Architekturen bevorzugt wird.
- **`E_DEMUX` (vergleichbare Funktion):** Einige Bibliotheken bieten ebenfalls Demultiplexer, aber meist mit mehreren Ausgängen oder ohne Adapteranbindung. Der **AX_E_SWITCH** fokussiert auf den Spezialfall 2-fach mit Adapter.

Fazit

Der **AX_E_SWITCH** ist ein einfacher, aber nützlicher Funktionsblock zur ereignisbasierten Weichenstellung. Durch die Adapter-Schnittstelle fügt er sich nahtlos in komponentenorientierte Steuerungsprojekte ein und ermöglicht eine saubere Trennung von Ereignislogik und Entscheidungsdaten. Er eignet sich besonders für modulare und wiederverwendbare Automatisierungslösungen.

AX_F_TRIG



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AX_F_TRIG** (Boolean falling edge detection) dient der Erkennung einer fallenden Flanke (Wechsel von TRUE auf FALSE) basierend auf einem Adapter-Interface. Er ist speziell für die Verwendung mit dem unidirektionalen Adaptertyp `AX` ausgelegt.

Der Baustein überprüft den booleschen Wert, der über den Adapter empfangen wird, und löst ein Ausgabeereignis aus, wenn dieser Wert `FALSE` ist. Im Kontext von Ereignisketten, bei denen das Ereignis des Adapters eine Wertänderung signalisiert, entspricht dies der Detektion einer fallenden Flanke.

Schnittstellenstruktur

Der Baustein verfügt über eine Adapter-Schnittstelle (Socket) für den Eingang und einen Ereignis-Ausgang.

Ereignis-Eingänge

Dieser Baustein besitzt keine direkten Ereignis-Eingänge. Die Ereignissteuerung erfolgt über den Adapter.

Ereignis-Ausgänge

- **EO** (Event Output): Dieses Ereignis wird ausgelöst, wenn eine fallende Flanke (oder ein logischer FALSE-Zustand) am Adapter erkannt wurde.

Daten-Eingänge

Dieser Baustein besitzt keine direkten Daten-Eingänge. Die Daten werden über den Adapter bezogen.

Daten-Ausgänge

Dieser Baustein besitzt keine direkten Daten-Ausgänge.

Adapter

- **QI** (Typ: `adapter::types::unidirectional::AX`): Der Eingangs-Socket. Über diesen Adapter empfängt der Baustein sowohl das Trigger-Ereignis als auch den zu prüfenden booleschen Wert.
  - *Kommentar:* Wert, der auf eine fallende Flanke geprüft werden soll.

Funktionsweise

Der **AX_F_TRIG** ist als zusammengesetzter Funktionsbaustein (Composite FB) realisiert. Seine interne Logik basiert auf der Verarbeitung der Signale aus dem `QI`-Adapter:

1. Der Adapter `QI` liefert ein Ereignis (`E1`) und ein boolesches Datum (`D1`).
2. Diese Signale werden an ein D-Flip-Flop (`E_D_FF`) weitergeleitet. Das Ereignis fungiert als Takt (CLK), das Datum als Eingang (D).
3. Das Flip-Flop speichert den Wert und löst ein Bestätigungsereignis aus.
4. Ein nachgeschalteter `E_SWITCH`-Baustein wertet den Ausgangszustand (`Q`) des Flip-Flops aus.
5. Ist der Wert `FALSE` (logisch 0), schaltet der `E_SWITCH` das Ereignis auf den Ausgang **EO** durch.
6. Ist der Wert `TRUE` (logisch 1), wird kein Ereignis am Ausgang **EO** erzeugt.

Zusammenfassend: Wenn der Adapter ein Ereignis sendet und der dabei übertragene Datenwert `FALSE` ist, feuert der Baustein `EO`.

Technische Besonderheiten

- **Adapter-Integration:** Der Baustein kapselt die Logik zur Aufspaltung des `AX`-Adapters in Einzelereignisse und Daten, um diese logisch zu verarbeiten.
- **Composite Network:** Die Implementierung erfolgt rein durch Verschaltung von Standard-IEC-61499-Bausteinen (`E_D_FF`, `E_SWITCH`) und enthält keine eigenen Algorithmen in ST oder C++.

Zustandsübersicht

Da es sich um einen Composite Function Block handelt, besitzt der **AX_F_TRIG** kein eigenes ECC (Execution Control Chart). Der Zustand wird implizit durch den Zustand der internen Bausteine (insbesondere des D-Flip-Flops `E_D_FF`) bestimmt.

Anwendungsszenarien

- **Signalverarbeitung:** Einsatz in Systemen, die `AX`-Adapter nutzen, um binäre Zustandsänderungen zu kommunizieren.

- **Ausschalt-Logik:** Auslösen von Aktionen, wenn ein Sensor oder Signal, das über einen Adapter angebunden ist, in den Zustand `FALSE` (Aus/Low) wechselt.
- **Event-Filterung:** Blockieren von Ereignissen, solange das zugehörige Datensignal `TRUE` ist, und Passierenlassen, wenn es `FALSE` ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- **E_F_TRIG:** Der Standard-Baustein für fallende Flankenerkennung in IEC 61499. Er nutzt separate Ereignis- (`EI`) und Daten-Eingänge (`QI`) anstelle eines Adapters.
- **AX_R_TRIG:** Das direkte Gegenstück zu diesem Baustein. Es detektiert eine steigende Flanke (Wechsel auf `TRUE`) unter Verwendung desselben Adapter-Typs.
- **E_SWITCH:** Ein generischer Schalter, der Ereignisse basierend auf einem booleschen Wert routet. **AX_F_TRIG** ist im Wesentlichen eine Spezialisierung hiervon für Adapter.

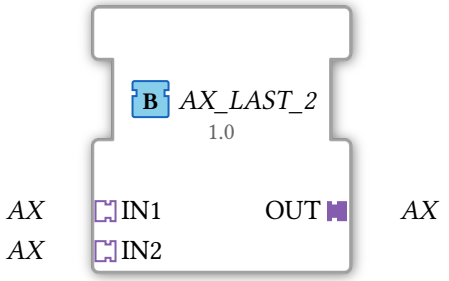
Fazit

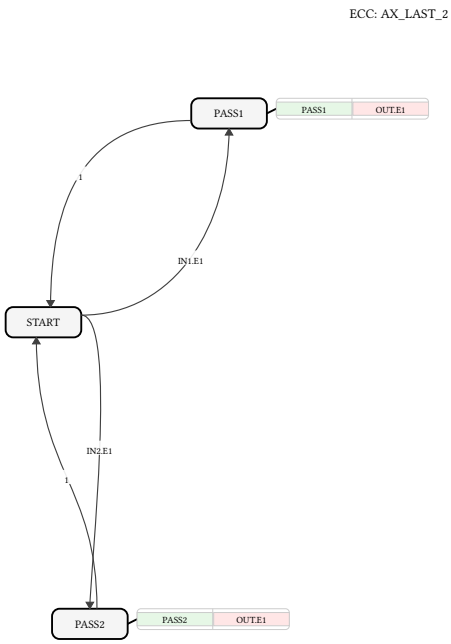
Der **AX_F_TRIG** ist ein spezialisierter Utility-Baustein für die strukturierte Ereignisverarbeitung in 4diac. Er vereinfacht den Umgang mit `AX`-Adaptern erheblich, indem er die Logik zur Erkennung des „Low“-Zustands (fallende Flanke) abstrahiert und direkt als Ereignis `EO` zur Verfügung stellt, ohne dass der Anwender das Adapter-Signal manuell aufspalten und prüfen muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AX_LAST_2





Einleitung

Der AX_LAST_2 Funktionsblock führt zwei gleichartige boolesches (BOOL)Adapter-Signale (**IN1**, **IN2**) auf einen gemeinsamen Ausgang (**OUT**) zusammen, nach dem Prinzip **Last Writer Wins**: welcher der beiden Sockets zuletzt ein Ereignis geschrieben hat, gewinnt, und sein Datenwert wird sofort und unverändert an OUT durchgereicht. Es handelt sich ausdrücklich nicht um ein naives Zusammenführen (z. B. Mittelwert oder ODER-Verknüpfung), sondern um eine reine Arbitrierung nach zeitlicher Reihenfolge.

Schnittstellenstruktur

Adapter

Eingangsadapter (Sockets):

- **IN1**: boolesches (BOOL)Adapter (unidirectional) - erste Signalquelle
- **IN2**: boolesches (BOOL)Adapter (unidirectional) - zweite Signalquelle

Ausgangsadapter (Plug):

- **OUT**: boolesches (BOOL)Adapter (unidirectional) - zusammengeführtes Signal

Funktionsweise

Der Baustein ist als einfacher Basic-FB mit einer 3-Zustands-ECC umgesetzt (START, PASS1, PASS2). Trifft an IN1 ein Ereignis ein, wechselt die ECC nach PASS1: der aktuelle Datenwert von IN1 wird nach OUT kopiert und das Adapter-Ereignis an OUT gesendet. Trifft an IN2 ein Ereignis ein, geschieht dasselbe symmetrisch über PASS2. Nach jeder Durchreichung kehrt die ECC sofort wieder nach START zurück und ist bereit für das nächste Ereignis von IN1 oder IN2.

Technische Besonderheiten

- Last-Writer-Wins-Semantik statt naiver Datenverknüpfung
- Einfacher Basic-FB (ECC mit 3 Zuständen), kein zusätzlicher Zwischenspeicher nötig
- Keine Signalverzögerung: der Wert wird im selben Zyklus durchgereicht, in dem das auslösende Ereignis eintrifft
- Beide Eingänge sind gleichberechtigt - es gibt keine feste Priorität außer der zeitlichen Reihenfolge

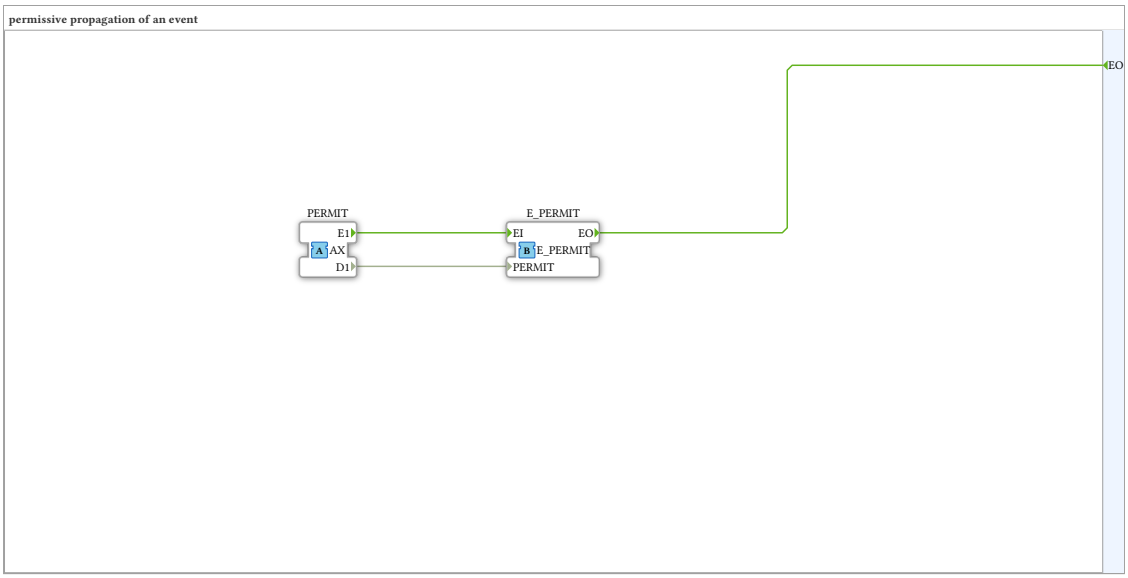
Anwendungsszenarien

- Zusammenführen zweier gleichwertiger boolesches (BOOL)Signalquellen zu einem einzigen nachgeschalteten Adapterpfad (z. B. zwei redundante Sensoren, oder ein manueller und ein automatischer Sollwertgeber)
- Arbitrierung zwischen zwei Bedienstellen, die denselben boolesches (BOOL)Wert schreiben dürfen
- Vereinfachung von Netzwerken, die sonst ein `F_SEL`/`E_RS`-Konstrukt inline verdrahten müssten

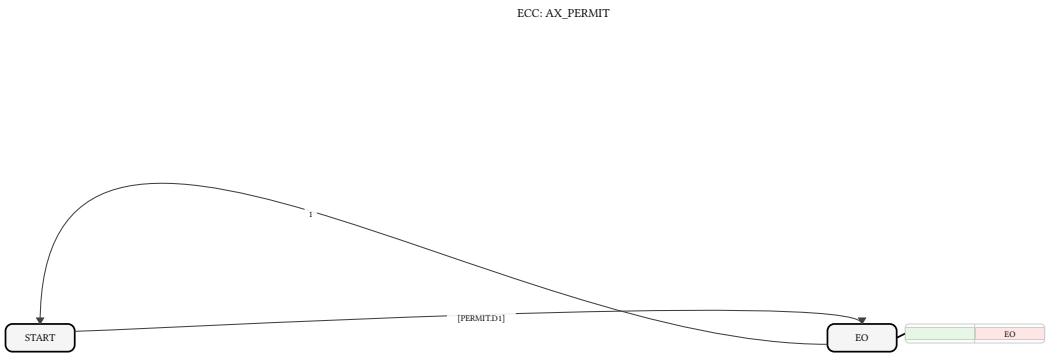
⚖️ Vergleich mit ähnlichen Bausteinen

AX_LAST_2 ist die Umkehrung von `AX_SPLIT_2`: AX_SPLIT_2 verteilt 1 Eingang auf 2 Ausgänge, AX_LAST_2 führt 2 Eingänge auf 1 Ausgang zusammen. Für Mischbetrieb zwischen einem Daten- und einem reinen Ereignisadapter (z. B. AX + AE) ist stattdessen `AX_AE_MERGE` gedacht, das nur das Ereignis mischt, ohne den Datenwert zu verändern.

AX_PERMIT (Unidirectional Adapter Permit)

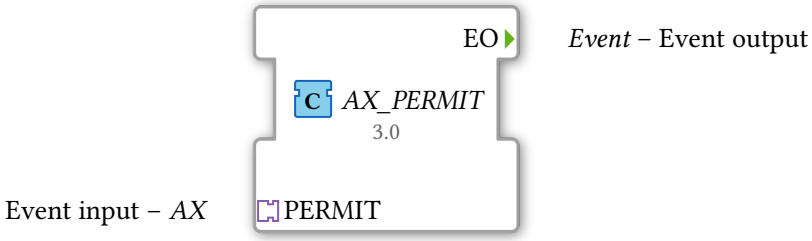


Einleitung



Der `AX_PERMIT` Funktionsblock ist eine adapterbasierte Variante des `E_PERMIT` Bausteins, implementiert als **Basic Function Block**. Er ermöglicht die bedingte Weitergabe eines Ereignisses, basierend auf dem booleschen Datenwert eines eingehenden `AX`-Adapters.

Der Baustein empfängt ein kombiniertes Ereignis- und Bool-Signal über einen `AX`-Adapter (genannt `PERMIT`). Das eingehende Ereignis wird nur dann an den Ausgang (`EO`) weitergeleitet, wenn der boolesche Datenwert des `PERMIT`-Adapters (`PERMIT.D1`) `TRUE` ist. Der Datenwert `PERMIT.D1` wird dabei **nicht** am Ausgang (`EO`) weitergereicht, sondern nur als Bedingung genutzt.



Schnittstellenstruktur

Adapter (Socket)

- **PERMIT**: Eingangsadapter vom Typ `AX` (Ereignis `E1` + Daten `D1`). Dieser Adapter steuert den Baustein.

Ereignis-Ausgänge

- **EO**: Reiner Ereignis-Ausgang (Typ `Event`). Dieser Event wird ausgelöst, wenn die Bedingung erfüllt ist.

Funktionsweise

1. **Ereignisempfang**: Wenn am `PERMIT`-Socket ein Ereignis (`PERMIT.E1`) eintrifft, wird der zugehörige boolesche Datenwert (`PERMIT.D1`) als Bedingung geprüft.
2. **Bedingte Weiterleitung**: Nur wenn `PERMIT.D1 = TRUE` ist, wird der Ereignis-Ausgang `EO` ausgelöst. Andernfalls wird das Ereignis blockiert.
3. **Datendurchleitung**: Der boolesche Datenwert (`PERMIT.D1`) wird vom Baustein verwendet, aber **nicht** über den Ereignis-Ausgang `EO` weitergereicht.

Technische Besonderheiten

✓ **Basic Function Block**: Direkte Implementierung der Logik. ✓ **Adapter-basiert**: Nutzt `AX`-Adapter als Eingang. ✓ **Permissive Logik**: Leitet Ereignisse bedingt weiter. ✓ **Keine Datenweitergabe am Ausgang**: Der Ausgang `EO` ist ein reiner Ereignis-Ausgang und enthält keine Daten (`With Var` ist nicht definiert).

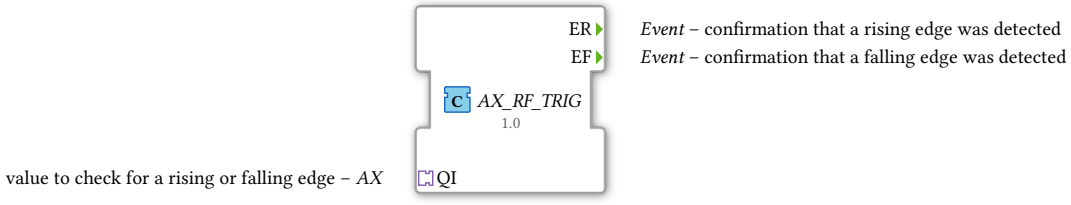
Anwendungsszenarien

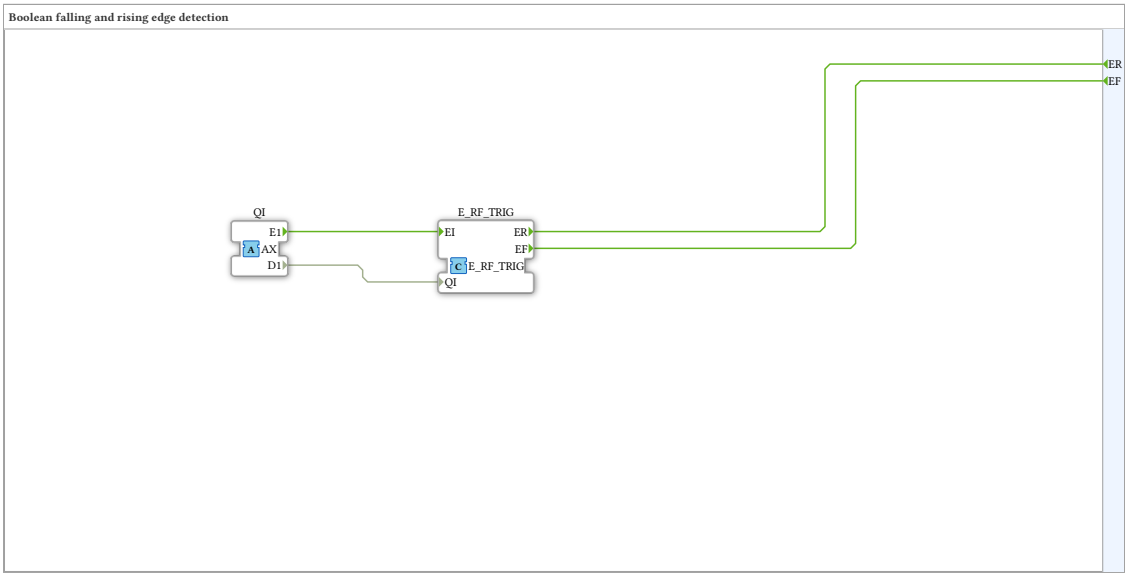
- **Freigabeschaltung**: Ein nachfolgender Baustein soll nur dann getriggert werden, wenn eine Freigabe (`PERMIT.D1 = TRUE`) aktiv ist.
- **Konditionale Ereignissteuerung**: Steuerung des Ereignisflusses basierend auf einem externen booleschen Status, der über einen AX-Adapter kommt.

🛠️ Zugehörige Übungen

- Uebung_009_AX

AX_RF_TRIG





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **AX_RF_TRIG** dient der Erkennung von steigenden und fallenden Flanken eines booleschen Signals. Er empfängt den Signalwert über einen Adapter und gibt bei einer Flankenänderung entsprechende Ereignisse aus. Der Baustein kapselt die Standardlogik der IEC-61499 Flankenerkennung (E_RF_TRIG) und stellt sie über eine einheitliche Adapterschnittstelle bereit.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine direkten Ereignis-Eingänge. Die Auslösung der Flankenerkennung erfolgt ereignisgetrieben über den angeschlossenen Adapter (siehe Adapter).

Ereignis-Ausgänge

Name	Kommentar
ER	Bestätigung, dass eine steigende Flanke (0⟶1) erkannt wurde.
EF	Bestätigung, dass eine fallende Flanke (1⟶0) erkannt wurde.

Daten-Eingänge

Keine direkten Dateneingänge. Der zu überwachende boolesche Wert wird über den Adapter bereitgestellt.

Daten-Ausgänge

Keine Datenausgänge.

Adapter

Name	Typ	Kommentar
QI	adapter::types::unidirectional::AX	Empfängt den booleschen Wert, dessen Flanken überwacht werden sollen. Der Adapter stellt sowohl ein Ereignis (E1) als auch den Datenwert (D1) bereit.

Funktionsweise

Der Baustein verwendet intern den IEC-61499 Standardbaustein **E_RF_TRIG**. Über den Adapter **QI** wird bei jedem eingehenden Ereignis der aktuelle boolesche Datenwert abgefragt. Tritt eine Änderung des Wertes auf, so erzeugt der interne Baustein ein Ereignis:

- **ER** bei einem Übergang von *false* nach *true* (steigende Flanke).
- **EF** bei einem Übergang von *true* nach *false* (fallende Flanke).

Ohne Flankenänderung werden keine Ausgangsereignisse gesendet.

Technische Besonderheiten

- Der Baustein ist als reiner Adapter-Wrapper realisiert. Er besitzt keine eigene Zustandsmaschine, sondern delegiert die gesamte Logik an den genormten **E_RF_TRIG**.
- Die Verwendung eines Adaptertyps (hier **AX**) ermöglicht eine lose Kopplung zwischen der Signalquelle und der Flankenerkennung. Der Adapter kann in verschiedenen Kontexten wiederverwendet werden.
- Es werden ausschließlich Ereignisausgänge bereitgestellt; der aktuelle Signalwert wird nicht als Datenausgang weitergegeben.

Zustandsübersicht

Der Baustein besitzt keine explizit dargestellte Zustandsmaschine. Das Verhalten entspricht dem des internen **E_RF_TRIG**, der einen impliziten Zustand (letzter Wert) speichert. Im Wesentlichen lassen sich zwei Phasen unterscheiden:

1. **Warten auf steigende Flanke:** der letzte Wert war *false* bei *true* wird ER ausgelöst.
2. **Warten auf fallende Flanke:** der letzte Wert war *true* bei *false* wird EF ausgelöst.

Anwendungsszenarien

- Überwachung von binären Sensorsignalen (z. B. Endschalter, Lichtschranken) in der Automatisierungstechnik.
- Auslösen von Aktionen bei Signalwechseln (z. B. Start/Stopp einer Maschine).
- Ereignisgesteuerte Verarbeitung von booleschen Zuständen in industriellen Steuerungen.

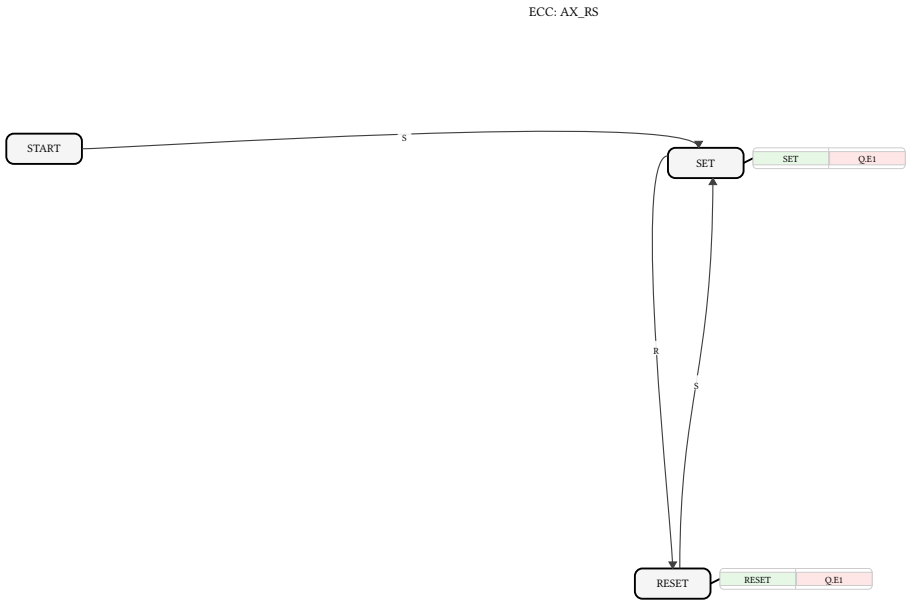
Vergleich mit ähnlichen Bausteinen

- **E_RF_TRIG:** Der Standardbaustein der IEC 61499. **AX_RF_TRIG** kapselt diesen und bietet eine Adapterschnittstelle an, sodass die Signalquelle nicht direkt an den Baustein gebunden werden muss.
- **R_TRIG** / **F_TRIG:** Erkennen nur eine Flankenrichtung. **AX_RF_TRIG** erkennt beide Richtungen und gibt entsprechende Ereignisse aus.
- **Adapter-basierte Alternativen:** Andere Adaptertypen mit ähnlicher Semantik können ebenfalls zur Flankenerkennung verwendet werden, sofern sie den booleschen Wert und ein Ereignis bereitstellen.

Fazit

Der Funktionsbaustein **AX_RF_TRIG** stellt eine praktische, adaptergestützte Lösung zur Erkennung beider Flanken eines booleschen Signals dar. Durch die Verwendung des Standardbausteins **E_RF_TRIG** ist die Funktion robust und normenkonform. Die Adapterschnittstelle erleichtert die Integration in modulare Systeme, bei denen Signalquellen und Verarbeitungslogik entkoppelt werden sollen.

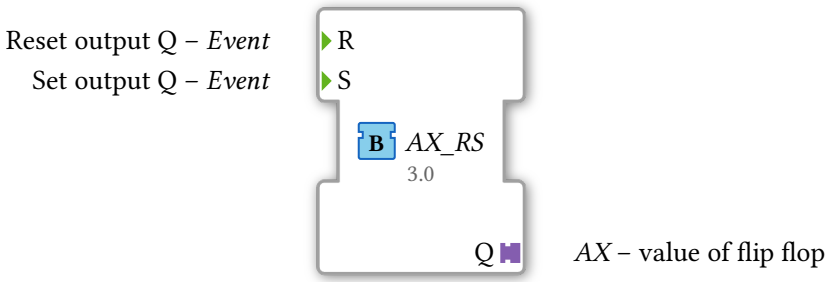
AX_RS



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_RS Funktionsblock ist ein ereignisgesteuertes bistabiles Element (Flip-Flop) nach dem RS-Prinzip. Es handelt sich um einen Basic Function Block, der einen Set-Reset-Mechanismus implementiert und über Adapter-Schnittstellen kommuniziert.



Schnittstellenstruktur

Ereignis-Eingänge

- **R**: Reset-Ereignis - setzt den Ausgang Q zurück
- **S**: Set-Ereignis - setzt den Ausgang Q

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

- **Q**: Adapter vom Typ `adapter::types::unidirectional::AX` - repräsentiert den Wert des Flip-Flops

Funktionsweise

Der AX_RS Funktionsblock arbeitet als RS-Flip-Flop mit folgenden Eigenschaften:

- Bei Eintreffen eines S-Ereignis (Set) wird der Ausgang Q auf TRUE gesetzt
- Bei Eintreffen eines R-Ereignis (Reset) wird der Ausgang Q auf FALSE gesetzt
- Der Zustand bleibt erhalten, bis ein gegenteiliges Ereignis eintrifft

Technische Besonderheiten und Normenvergleich

Wie bei allen ereignisgesteuerten bistabilen Elementen in der IEC 61499 (siehe auch Anmerkung 8 in Tabelle A.1 der DIN EN 61499-1) gibt es hier keine inhärente „Dominanz“ eines Eingangs, wie man sie von der IEC 61131-3 kennt.

- **Vergleich zur IEC 61131-3**: Siehe RS (Bistabil, vorrangig rücksetzen). Während in der klassischen SPS-Welt bei gleichzeitigem TRUE an S und R1 das Rücksetzen gewinnt, wird in der IEC 61499 jedes Ereignis nacheinander verarbeitet. Der Endzustand hängt davon ab, welches Ereignis zuletzt in der Ausführungskette (ECC) abgearbeitet wurde.
- **Funktionale Identität**: `AX_RS` ist funktional identisch zu AX_SR. Die unterschiedliche Benennung dient lediglich der Konsistenz zur klassischen Programmierung und der besseren Lesbarkeit im Schaltplan.
- **Adapter-Kommunikation**: Der Baustein gibt seinen Zustand ausschließlich über den Adapter `Q` (Typ `AX`) aus. Eine Änderung von `Q` löst das Ereignis `Q.E1` aus.

Zustandsübersicht

Der Funktionsblock verfügt über drei Zustände im ECC:

1. **START**: Initialzustand
2. **SET**: Zustand nach Set-Operation (Q.D1 = TRUE)
3. **RESET**: Zustand nach Reset-Operation (Q.D1 = FALSE)

Zustandsübergänge:

- START ⟶ SET: bei S-Ereignis
- SET ⟶ RESET: bei R-Ereignis
- RESET ⟶ SET: bei S-Ereignis

Anwendungsszenarien

- Speicherung von binären Zuständen mit Adapter-Ausgabe
- Implementierung von Verriegelungsschaltungen in verteilten Systemen
- Zustandsspeicherung in sequentiellen Abläufen
- Signalverarbeitung in ereignisgesteuerten Systemen

Verwandte Bausteine

- **AX_SR**: Funktional identisch, Eingänge im Symbol vertauscht.
- **E_RS**: Das Standard-Äquivalent mit direkten Daten-/Ereignisausgängen statt Adaptern.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Flip-Flop-Implementierungen:

- Verwendet Adapter-basierte Kommunikation statt direkter Datenausgänge
- Ereignisgesteuerte Zustandsänderungen
- Einfache RS-Logik ohne zusätzliche Takt- oder Enable-Signale

Vergleich mit E_RS

🛠️ Zugehörige Übungen

- Uebung_006b_AX
- Uebung_020a_AX
- Uebung_020b_AX
- Uebung_020d_AX

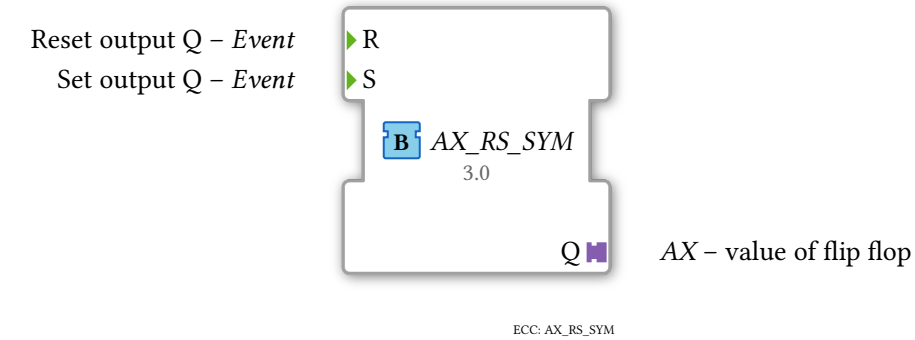
Fazit

Der AX_RS Funktionsblock bietet eine einfache und effiziente Implementierung eines RS-Flip-Flops für 4diac-basierte Steuerungssysteme. Durch die Verwendung von Adaptern ermöglicht er eine flexible Integration in verschiedene Systemarchitekturen und eignet sich besonders für Anwendungen, die zuverlässige Zustandsspeicherung mit ereignisgesteuerter Aktualisierung benötigen.

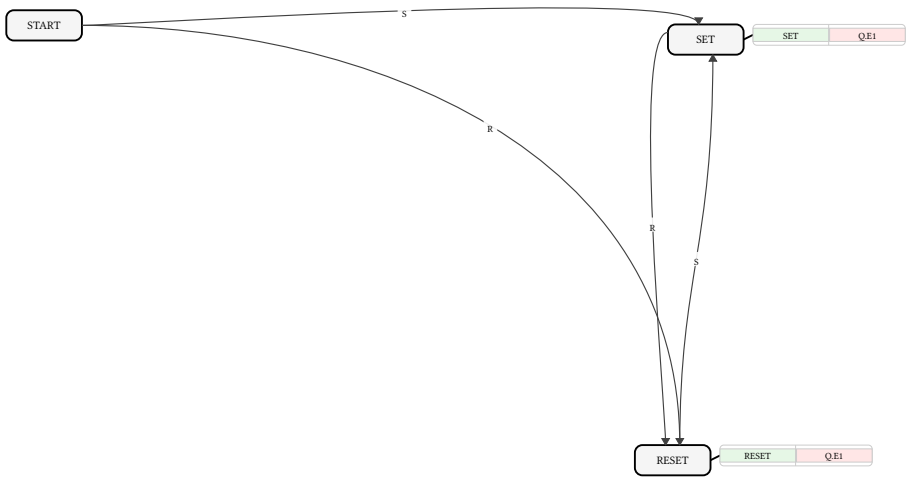
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AX_RS_SYM



ECC: AX_RS_SYM



• – ○ • – ○ • – ○ •

Einleitung

Der **AX_RS_SYM** ist ein ereignisgesteuerter, bistabiler Funktionsblock (RS-Flip-Flop) mit symmetrischem Start-Up-Verhalten. Er verfügt über zwei Ereigniseingänge `R` (Reset) und `S` (Set), die jeweils den Zustand des Ausgangs `Q` setzen. Der Ausgang wird als unidirektionaler Adapter bereitgestellt, über den sowohl ein Datenwert (D1) als auch ein Ereignis (E1) übertragen werden.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar

R	Event	Setzt den Ausgang Q.D1 auf FALSE und feuert Q.E1
S	Event	Setzt den Ausgang Q.D1 auf TRUE und feuert Q.E1

Ereignis-Ausgänge

Der FB besitzt keine eigenen Ereignis-Ausgänge. Das Ereignis wird über den Adapter Q mittels Q.E1 ausgegeben.

Daten-Eingänge

Keine.

Daten-Ausgänge

Der FB besitzt keine separaten Daten-Ausgänge. Der aktuelle Zustand wird über den Adapter Q als Q.D1 (vom Typ BOOL) bereitgestellt.

Adapter

Name	Typ	Kommentar
Q	adapter::types::unidirectional::AX	Bietet den Flip-Flop-Wert (D1) und ein Bestätigungsereignis (E1)

Funktionsweise

Der Funktionsblock realisiert ein klassisches RS-Flip-Flop:

- Bei Eintreffen des Ereignisses S wird der Ausgangswert Q.D1 auf TRUE gesetzt und anschließend das Ereignis Q.E1 ausgelöst.
- Bei Eintreffen des Ereignisses R wird der Ausgangswert Q.D1 auf FALSE gesetzt und ebenfalls Q.E1 ausgelöst.

Das Besondere ist das **symmetrische Start-Up-Verhalten**: Unmittelbar nach der Initialisierung befindet sich der FB im Zustand START. Wird als erstes Ereignis S empfangen, wechselt er in den Zustand SET; bei R wechselt er in den Zustand RESET. Damit ist von Anfang an ein definierter Zustand gegeben – es gibt keinen undefinierten oder zufälligen Anfangswert.

Technische Besonderheiten

- **Symmetrisches Startverhalten**: Beide Eingänge R und S werden gleichberechtigt behandelt. Der FB hat keinen bevorzugten Anfangszustand; der erste ankommende Impuls bestimmt den Zustand.
- **Adapter-basierte Ausgabe**: Der Ausgang ist als unidirektionaler Adapter realisiert. Dies erlaubt eine flexible Weiterverwendung des Wertes und des Ereignisses in komplexeren Strukturen.
- **Kein undefinierter Zustand**: Durch die Zustandsmaschine mit drei Zuständen (START, SET, RESET) wird vermieden, dass der FB nach dem Hochlauf in einem unbekannten oder metastabilen Zustand verharrt.

Zustandsübersicht

Die interne Zustandsmaschine besteht aus folgenden Zuständen und Transitionen:

Zustand	Bedeutung	Aktion bei Eintritt
START	Initialzustand nach dem Einschalten	Keine Aktion (wartet auf erstes Ereignis)
SET	Ausgang Q.D1 ist TRUE	Führt Algorithmus SET aus: Q.D1 := TRUE und feuert Q.E1
RESET	Ausgang Q.D1 ist FALSE	Führt Algorithmus RESET aus: Q.D1 := FALSE und feuert Q.E1

Transitionen:

- START ⟶ SET bei Bedingung S
- START ⟶ RESET bei Bedingung R
- SET ⟶ RESET bei Bedingung R
- RESET ⟶ SET bei Bedingung S

Anwendungsszenarien

- **Einfacher Schalter/Taster**: Ein Tastendruck setzt ein Merker-Bit, ein zweiter Tastendruck setzt es zurück.
- **Verriegelungen** (z. B. in Sicherheitssteuerungen): Ein Set-Impuls aktiviert eine Aktion, ein Reset-Impuls deaktiviert sie.
- **Zustandssteuerung in Ablaufketten**: Als Gedächtnisbaustein für Ereignisse, die später ausgewertet werden.
- **Start-Up-Verhalten**: Besonders geeignet, wenn die Anlage nach dem Einschalten in einem definierten, aber noch nicht durch Vorlogik vorgegebenen Zustand starten soll.

Vergleich mit ähnlichen Bausteinen

- **AX_RS**: Ein oft verwendetes RS-Flip-Flop mit asynchronem Set/Reset. Im Gegensatz zu `AX_RS_SYM` definiert die meisten Implementierungen einen bevorzugten Zustand (z. B. setzt das „R“-Signal beim Start den Ausgang zurück). `AX_RS_SYM` behandelt beide Ereignisse symmetrisch, was das Start-Up-Verhalten vorhersagbar macht.
- **AX_SR (Set-dominant)** oder **AX_RS (Reset-dominant)**: Diese Bausteine haben eine dominante Eingangsreihenfolge bei gleichzeitigem Eintreffen von S und R. Hier hingegen ist die Symmetrie nur auf das Startverhalten bezogen; während des Betriebs wird bei gleichzeitigen Ereignissen die Reihenfolge durch die EVENT-Verarbeitung der Laufzeitumgebung bestimmt – der FB selbst kann keine Gleichzeitigkeit abfangen.
- **SR-Flip-Flop im IEC-61131-3**: (z. B. `SR` oder `RS`) arbeiten meist mit statischen Signalen. `AX_RS_SYM` ist rein event-getrieben und daher für ereignisorientierte Steuerungen (z. B. 4diac-IDE mit FORTE) optimiert.

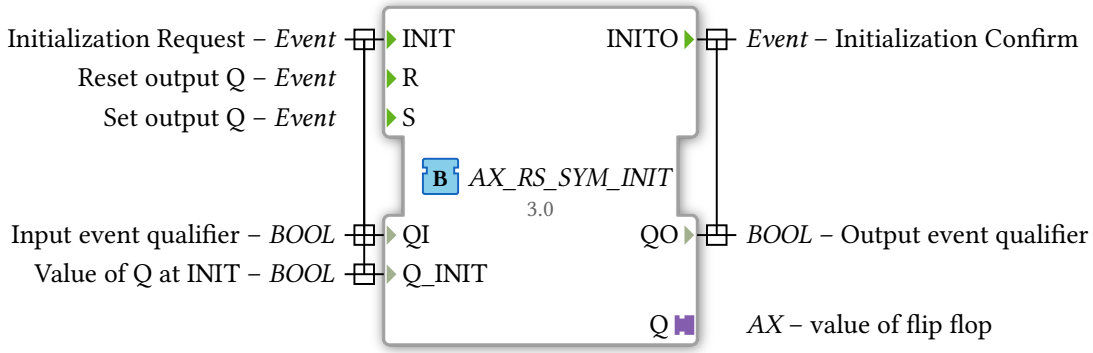
Fazit

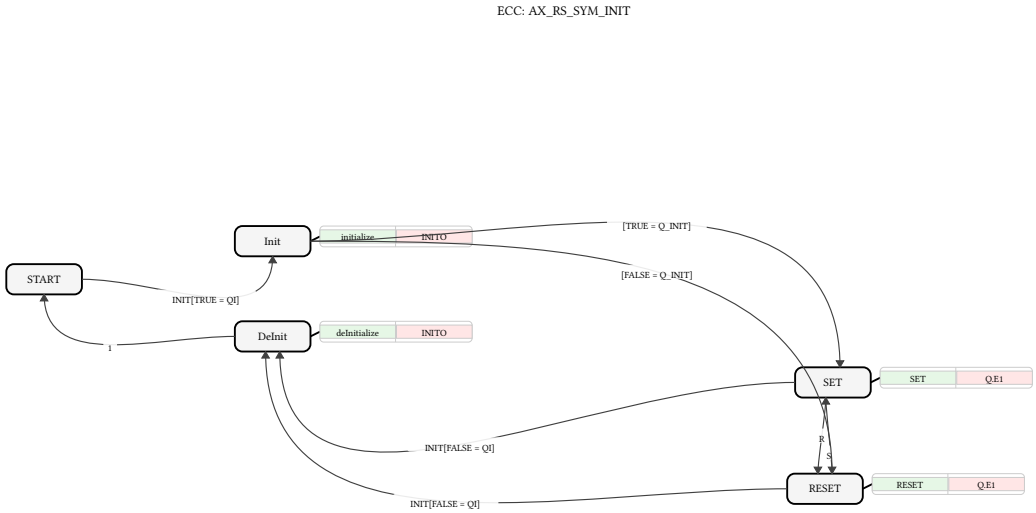
Der **AX_RS_SYM** ist ein einfacher, aber flexibler ereignisgesteuerter RS-Flip-Flop-Baustein. Seine Besonderheit liegt im symmetrischen Start-Up-Verhalten, das einen definierten Anfangszustand ohne feste Voreinstellung ermöglicht. Die Adapter-Schnittstelle erleichtert die Einbindung in größere Systeme. Für Anwendungen, bei denen nach dem Hochlauf weder ein Setzen noch ein Zurücksetzen bevorzugt werden soll, ist dieser Baustein die ideale Wahl.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AX_RS_SYM_INIT





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AX_RS_SYM_INIT** realisiert einen ereignisgesteuerten, bistabilen Flip-Flop (RS-Flip-Flop) mit symmetrischem Start-up-Verhalten und explizitem Initialisierungsmechanismus. Er erweitert einen einfachen RS-Flip-Flop um die Fähigkeit, beim Start (INIT) einen vorgegebenen Zustand einzunehmen und über den Ereignis-Eingang INIT sowohl die Initialisierung als auch die Deinitialisierung zu steuern. Der tatsächliche Zustand des Flip-Flops wird über einen Adapter bereitgestellt, während die Datenausgänge QO und der Adapter-Ausgang Q.D1 synchronisiert werden.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Initialisierungsanforderung (mit QI, Q_INIT)
R	Event	Reset-Signal (setzt Q zurück)
S	Event	Set-Signal (setzt Q)

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Initialisierungsbestätigung (mit QO)

Daten-Eingänge

Name	Typ	Kommentar
QI	BOOL	Ereignis-Qualifier – steuert die Ausführung
Q_INIT	BOOL	Gewünschter Wert von Q bei der Initialisierung

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Ereignis-Qualifier – gibt QI an INITO oder bei S/R weiter

Adapter

Name	Typ	Kommentar
Q	adapter::types::unidirectional::AX	Adapter für den Wert des Flip-Flops (D1)

Funktionsweise

Der FB durchläuft verschiedene Zustände, gesteuert durch die Ereignisse INIT, S und R.

- Im **START**-Zustand wartet er auf ein INIT-Ereignis.
  - Wird INIT mit `QI = TRUE` empfangen, wechselt er in den **Init**-Zustand. Die Aktion `initialize` setzt `QO := QI`.
  - Abhängig vom Wert von `Q_INIT`:
    - Ist `Q_INIT = TRUE` ⟶ Wechsel in **SET**-Zustand.
    - Ist `Q_INIT = FALSE` ⟶ Wechsel in **RESET**-Zustand.
  - Nach der Aktion wird `INITO` ausgelöst.
- Wird INIT mit `QI = FALSE` empfangen (egal ob aus **SET** oder **RESET**), wechselt er in den **DeInit**-Zustand. Die Aktion `deInitialize` setzt `QO := FALSE`. Anschließend wird `INITO` ausgelöst und der FB geht zurück zu **START**.
- In den Zuständen **SET** und **RESET** verhält sich der FB wie ein herkömmlicher RS-Flip-Flop:
  - Bei **S**-Ereignis wird die Aktion `SET` ausgeführt: `QO := QI`; wenn `QI = TRUE`, wird zusätzlich `Q.D1 := TRUE` gesetzt (Adapter-Ausgang).
  - Bei **R**-Ereignis wird die Aktion `RESET` ausgeführt: `QO := QI`; wenn `QI = TRUE`, wird `Q.D1 := FALSE` gesetzt.
  - Das Ereignis an `Q.E1` wird nach jeder Aktion gesendet.

Die `QI`-Variable fungiert als Qualifier: Nur wenn `QI = TRUE`, werden die Operationen auf den Adapter (Setzen/Rücksetzen von `Q.D1`) tatsächlich durchgeführt. Andernfalls wird nur `QO` weitergereicht.

Technische Besonderheiten

- **Symmetrisches Start-up-Verhalten**: Die Initialisierung berücksichtigt den Wert von `Q_INIT`, sodass der Flip-Flop gezielt in den SET- oder RESET-Zustand starten kann. Das Verhalten ist symmetrisch, da beide Möglichkeiten explizit abgebildet sind.
- **Adapter-Kopplung**: Der Zustand des Flip-Flops wird nicht direkt als Datenausgang geführt, sondern über einen **unidirektionalen Adapter** (`Q`) mit dem Ausgang `D1` bereitgestellt. Dies ermöglicht eine lose Kopplung mit weiteren Bausteinen.
- **Deinitialisierung**: Durch INIT mit `QI = FALSE` wird der FB zurückgesetzt und kehrt in den START-Zustand zurück. Dabei werden alle internen Variablen auf definierte Werte gesetzt (`QO = FALSE`).
- **Ereignis-Qualifier**: `QI` und `QO` dienen der Steuerung der Ereignisweitergabe und bieten eine zusätzliche Bedingungsebene.

Zustandsübersicht

Zustand	Beschreibung
START	Warten auf INIT-Ereignis
Init	Initialisierung: Setzt QO und wählt Zielzustand
DeInit	Deinitialisierung: Setzt QO = FALSE
SET	Aktiver Set-Zustand (Q.D1 = TRUE)
RESET	Aktiver Reset-Zustand (Q.D1 = FALSE)

Wichtige Transitionen:

- START ⟶ Init bei INIT mit QI = TRUE
- Init ⟶ SET bei Q_INIT = TRUE
- Init ⟶ RESET bei Q_INIT = FALSE
- SET ⟶ RESET bei R
- RESET ⟶ SET bei S
- SET/RESET ⟶ DeInit bei INIT mit QI = FALSE
- DeInit ⟶ START (automatisch nach Abschluss)

Anwendungsszenarien

- **Anlaufsteuerung**: Ein Flip-Flop, der nach einem Start oder Reset einen definierten Anfangszustand einnehmen soll (z. B. Maschine startet im Zustand „Bereit“).
- **Sichere Initialisierung**: In sicherheitskritischen Systemen, bei denen der Startzustand explizit vorgegeben werden muss.
- **Modularer Aufbau**: Einsatz als grundlegendes Speicherelement in einer Adapter-basierten Architektur, bei der der Zustand über den Adapter an andere Bausteine weitergegeben wird.

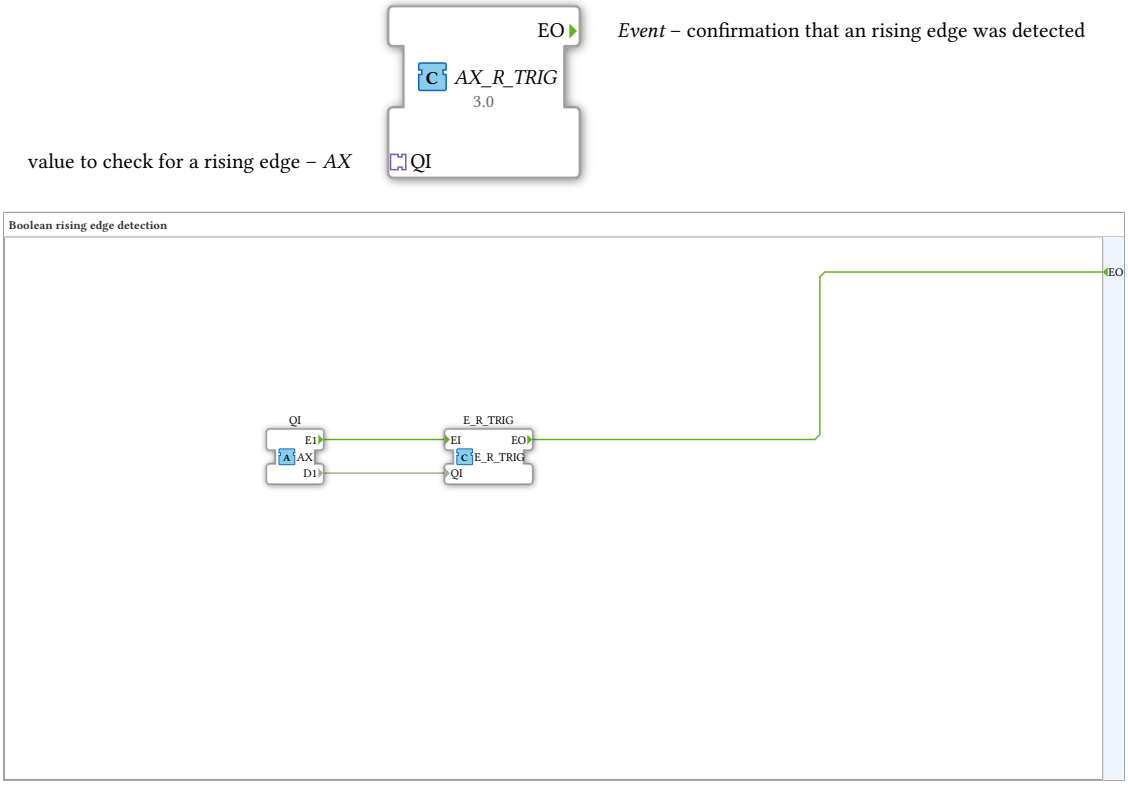
Vergleich mit ähnlichen Bausteinen

- **Standard RS (z. B. RS-Flipflop)**: Einfacher RS-Flip-Flop ohne INIT und ohne Adapter. `AX_RS_SYM_INIT` erweitert dies um Initialisierungslogik und Adapter-Schnittstelle.
- **Bistabile Kippglieder mit Reset**: Viele Bausteine bieten nur asynchrones Rücksetzen. Dieser FB integriert das Reset in das EVENT-gesteuerte Verhalten und erlaubt eine symmetrische Zielvorgabe.
- **SR-Latch mit INIT**: Ähnliche Bausteine existieren, jedoch meist ohne den Qualifier `QI` und die Deinitialisierungsmöglichkeit.

Fazit

Der FB `AX_RS_SYM_INIT` kombiniert ein klassisches RS-Flipflop-Verhalten mit einer flexiblen Initialisierungslogik, die sowohl Setzen als auch Rücksetzen beim Start erlaubt. Die Verwendung eines Adapters für den Zustandswert und die Einbeziehung von Qualifiern (`QI`, `QO`) machen ihn zu einem robusten und vielseitigen Baustein für ereignisgesteuerte Automatisierungsanwendungen, die einen definierten und rücksetzbaren Anfangszustand benötigen.

AX_R_TRIG



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AX_R_TRIG** (Boolean rising edge detection) dient der Erkennung einer steigenden Flanke basierend auf einem über einen Adapter bereitgestellten booleschen Signal. Er ist speziell für die Verwendung mit unidirektionalen Adaptern vom Typ `AX` konzipiert und leitet ein Ereignis nur dann weiter, wenn der übermittelte Datenwert `TRUE` ist.

Schnittstellenstruktur

Der Baustein verwendet primär eine Adapter-Schnittstelle zur Kommunikation der Eingangsdaten und Ereignisse.

Ereignis-Eingänge

Dieser Baustein verfügt über keine direkten Ereignis-Eingänge. Die Ereignisse werden über den Adapter `QI` empfangen.

Ereignis-Ausgänge

- **EO**: (Event Output) Bestätigung, dass eine steigende Flanke (oder ein High-Signal) erkannt wurde.

Daten-Eingänge

Dieser Baustein verfügt über keine direkten Daten-Eingänge. Die Daten werden über den Adapter `QI` empfangen.

Daten-Ausgänge

Dieser Baustein verfügt über keine direkten Daten-Ausgänge.

Adapter

- **QI** (Socket): Vom Typ `adapter::types::unidirectional::AX`. Dieser Adapter liefert den zu prüfenden Wert (boolesch) und das dazugehörige Ereignis.
  - Erwartet typischerweise ein Ereignis (z. B. `E1`) und ein boolesches Datum (z. B. `D1`).

Funktionsweise

Der Baustein ist als Composite Function Block (CFB) implementiert und basiert auf der internen Verschaltung eines D-Flip-Flops (`E_D_FF`) und eines Ereignis-Schalters (`E_SWITCH`).

1. Sobald über den Adapter **QI** ein Ereignis empfangen wird, wird dieses an den Takteingang des internen `E_D_FF` geleitet.
2. Gleichzeitig wird der boolesche Datenwert vom Adapter in das Flip-Flop geladen.
3. Das Flip-Flop gibt den gespeicherten Wert an einen `E_SWITCH` weiter.
4. Ist der Wert `TRUE` (High), schaltet der `E_SWITCH` durch und löst das Ausgangsereignis **EO** aus.
5. Ist der Wert `FALSE` (Low), wird das Ereignis blockiert und **EO** wird nicht ausgelöst.

Technisch gesehen fungiert der Baustein als Gate, das das Adapter-Ereignis nur passieren lässt, wenn das Adapter-Datum `TRUE` ist. In einem ereignisbasierten System, das nur bei Wertänderungen sendet, entspricht dies der Logik einer Flankenerkennung (`0 -> 1`).

Technische Besonderheiten

- **Interne Struktur:** Verwendet Standard-IEC-61499-Bausteine (`E_D_FF`, `E_SWITCH`).
- **Adapter-basiert:** Im Gegensatz zu klassischen Trigger-Bausteinen, die separate `CLK` und `QI` Eingänge haben, kapselt dieser Baustein die Signale in einem `AX`-Adapter.

Zustandsübersicht

Da es sich um einen Composite Function Block handelt, besitzt dieser Baustein keine eigene Zustandsmaschine (ECC). Das Verhalten wird rein durch den Datenfluss und die Verschaltung der internen Unterbausteine bestimmt.

Anwendungsszenarien

- **Signalfilterung:** Weiterleitung von Ereignissen nur dann, wenn ein bestimmter Zustand aktiv ist.
- **Adapter-Logik:** Einsatz in Systemen, die stark auf Adapter-Verbindungen setzen, um den Verdrahtungsaufwand zu minimieren.
- **Start-Trigger:** Auslösen von Prozessen, sobald ein boolesches Flag innerhalb einer Adapter-Struktur auf `TRUE` wechselt.

⚖️ Vergleich mit ähnlichen Bausteinen

- **E_R_TRIG:** Der Standard-Baustein für Ereignis-basierte steigende Flankenerkennung mit separaten Ereignis- und Dateneingängen. `AX_R_TRIG` ist das Äquivalent für Adapter-Verbindungen.
- **R_TRIG:** Der klassische SPS-Baustein (IEC 61131-3), der zyklisch abgefragt wird. `AX_R_TRIG` arbeitet hingegen rein ereignisgesteuert.

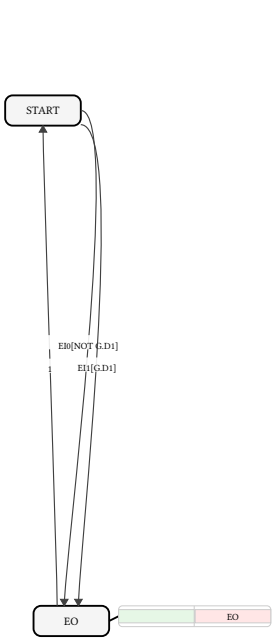
Fazit

Der **AX_R_TRIG** ist ein spezialisierter Baustein für die 4diac-Umgebung, der die Logik einer steigenden Flankenerkennung oder eines High-Pass-Filters nahtlos in Designs integriert, die `AX`-Adapter verwenden. Er vereinfacht das Netzwerk, indem er das explizite Aufsplitten von Adapter-Signalen zur Prüfung unnötig macht.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

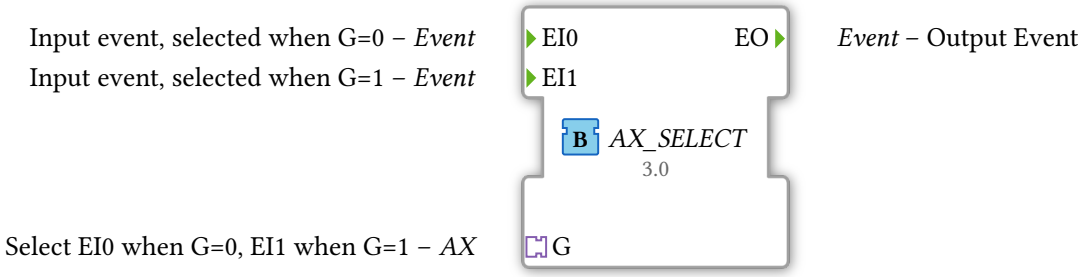
AX_SELECT



• – ○ • – ○ • – ○ •

Einleitung

Der AX_SELECT ist ein Funktionsblock, der basierend auf einem booleschen Eingang einen von zwei AX-Adapter-Eingängen auf den Ausgang schaltet.



Schnittstellenstruktur

Ereignis-Eingänge

- **EI0**: Eingangsereignis, wird bei G = 0 ausgewählt.
- **EI1**: Eingangsereignis, wird bei G = 1 ausgewählt.

Ereignis-Ausgänge

- **EO**: Ausgangsereignis.

Adapter

Sockets (Eingänge):

- **G** (adapter::types::unidirectional::AX): Auswahlsignal. EI0 wird bei G = 0 ausgewählt, EI1 bei G = 1.

Funktionsweise

Wenn G = 0, wird das Ereignis EI0 als EO ausgegeben. Wenn G = 1, wird das Ereignis EI1 als EO ausgegeben.

Technische Besonderheiten

- Verwendet unidirektionale Adapter.

Zustandsübersicht

Kombinatorisch.

Anwendungsszenarien

Auswahl von Signalen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **E_SELECT**: Standard Ereignis-Select ohne Adapter.

🛠️ Zugehörige Übungen

- Uebung_095_AX

Fazit

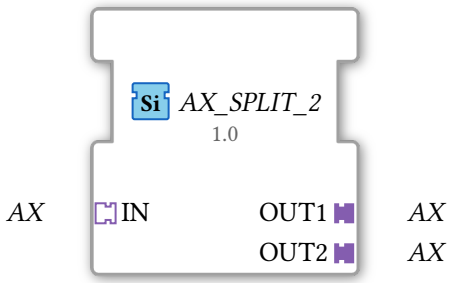
Adapter-basierter Select-Baustein.

AX_SPLIT_2

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_SPLIT_2 Funktionsblock dient als generischer Baustein zur Verteilung eines AX-Signals auf zwei separate Ausgänge. Der Block ermöglicht die Aufteilung eines eingehenden AX-Signals auf zwei unabhängige Ausgabekanäle.



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN**: AX-Adapter (unidirectional) - Empfängt das eingehende AX-Signal

Ausgangsadapter:

- **OUT1**: AX-Adapter (unidirectional) - Erster Ausgangskanal für das verteilte Signal
- **OUT2**: AX-Adapter (unidirectional) - Zweiter Ausgangskanal für das verteilte Signal

Funktionsweise

Der AX_SPLIT_2 Funktionsblock empfängt ein AX-Signal über den IN-Adapter und verteilt dieses Signal gleichzeitig auf beide Ausgangsadapter OUT1 und OUT2. Es handelt sich um eine 1:2-Verteilung, bei der das eingehende Signal ohne Änderung an beide Ausgänge weitergeleitet wird.

Technische Besonderheiten

- Generische Implementierung für AX-Signale
- Unidirektionale Signalübertragung
- Keine Signalverzögerung zwischen Ein- und Ausgang
- Gleichzeitige Aktivierung beider Ausgänge

Zustandsübersicht

Der Funktionsblock arbeitet zustandslos - bei jedem eingehenden Signal über den IN-Adapter werden sofort beide Ausgangsadapter aktiviert.

Anwendungsszenarien

- Signalverteilung in Steuerungssystemen
- Parallele Versorgung mehrerer Komponenten mit demselben Signal
- Verzweigung von AX-Kommunikationspfaden
- Redundante Signalweiterleitung

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Verteilungsbausteinen bietet AX_SPLIT_2 eine spezifische 1:2-Aufteilung für AX-Signale. Andere Splitter-Bausteine könnten unterschiedliche Anzahlen von Ausgängen oder andere Signaltypen unterstützen.

Vergleich mit E_SPLIT

- `AX_SPLIT_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

🛠️ Zugehörige Übungen

- Uebung_002_AX
- Uebung_004b_AX
- Uebung_004b_AX_ASR
- Uebung_004b_AX_ASR_X
- Uebung_006a3_sub_AX
- Uebung_007a3_AX
- Uebung_008_AX
- Uebung_010c2_AX
- AuxIXA_TO_logiBUS_QXA_BG
- Uebung_010c_AX
- Uebung_020c3_AX

- Uebung_020e2_AX
- Uebung_020f2_AX
- AX_SPLIT_TO_2x_AX_TP
- Uebung_020j_AX
- Uebung_035a2_AX
- Uebung_035a3_AX
- Uebung_094a_AX
- Uebung_160_AX
- Uebung_160b2_AX
- Uebung_160b_AX

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der AX_SPLIT_2 Funktionsblock stellt eine einfache und effiziente Lösung zur Verteilung von AX-Signalen auf zwei Ausgänge dar. Seine generische Natur und die unidirektionale Architektur machen ihn zu einem vielseitig einsetzbaren Baustein in verteilten Automatisierungssystemen.

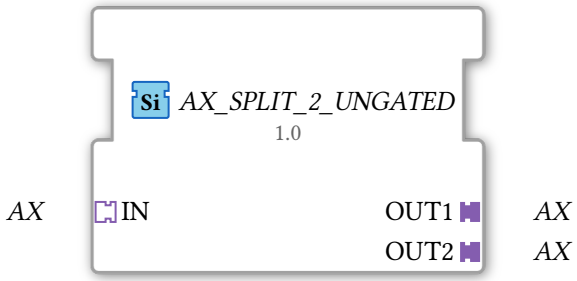
AX_SPLIT_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AX_SPLIT_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_SPLIT_2_UNGATED Funktionsblock dient als generischer Baustein zur Verteilung eines AX-Signals auf zwei separate Ausgänge. Der Block ermöglicht die Aufteilung eines eingehenden AX-Signals auf zwei unabhängige Ausgabekanäle.



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN**: AX-Adapter (unidirectional) - Empfängt das eingehende AX-Signal

Ausgangsadapter:

- **OUT1**: AX-Adapter (unidirectional) - Erster Ausgangskanal für das verteilte Signal
- **OUT2**: AX-Adapter (unidirectional) - Zweiter Ausgangskanal für das verteilte Signal

Funktionsweise

Der AX_SPLIT_2_UNGATED Funktionsblock empfängt ein AX-Signal über den IN-Adapter und verteilt dieses Signal gleichzeitig auf beide Ausgangsadapter OUT1 und OUT2. Es handelt sich um eine 1:2-Verteilung, bei der das eingehende Signal ohne Änderung an beide Ausgänge weitergeleitet wird.

Technische Besonderheiten

- Generische Implementierung für AX-Signale
- Unidirektionale Signalübertragung
- Keine Signalverzögerung zwischen Ein- und Ausgang
- Gleichzeitige Aktivierung beider Ausgänge

Zustandsübersicht

Der Funktionsblock arbeitet zustandslos - bei jedem eingehenden Signal über den IN-Adapter werden sofort beide Ausgangsadapter aktiviert.

Anwendungsszenarien

- Signalverteilung in Steuerungssystemen
- Parallele Versorgung mehrerer Komponenten mit demselben Signal
- Verzweigung von AX-Kommunikationspfaden
- Redundante Signalweiterleitung

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Verteilungsbausteinen bietet AX_SPLIT_2_UNGATED eine spezifische 1:2-Aufteilung für AX-Signale. Andere Splitter-Bausteine könnten unterschiedliche Anzahlen von Ausgängen oder andere Signaltypen unterstützen.

Vergleich mit E_SPLIT

- `AX_SPLIT_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

🛠️ Zugehörige Übungen

- Uebung_002_AX
- Uebung_004b_AX
- Uebung_004b_AX_ASR
- Uebung_004b_AX_ASR_X
- Uebung_006a3_sub_AX
- Uebung_007a3_AX
- Uebung_008_AX
- Uebung_010c2_AX
- AuxIXA_TO_logiBUS_QXA_BG
- Uebung_010c_AX
- Uebung_020c3_AX
- Uebung_020e2_AX
- Uebung_020f2_AX
- AX_SPLIT_TO_2x_AX_TP
- Uebung_020j_AX
- Uebung_035a2_AX
- Uebung_035a3_AX
- Uebung_094a_AX

- Uebung_160_AX
- Uebung_160b2_AX
- Uebung_160b_AX

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

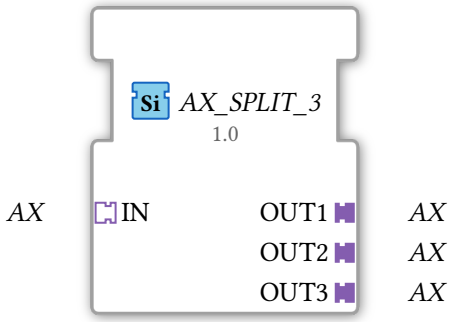
Der AX_SPLIT_2_UNGATED Funktionsblock stellt eine einfache und effiziente Lösung zur Verteilung von AX-Signalen auf zwei Ausgänge dar. Seine generische Natur und die unidirektionale Architektur machen ihn zu einem vielseitig einsetzbaren Baustein in verteilten Automatisierungssystemen.

AX_SPLIT_3

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_SPLIT_3 ist ein generischer Funktionsblock, der einen AX-Adapter-Eingang auf drei separate AX-Adapter-Ausgänge aufteilt. Der Baustein ermöglicht die Verteilung eines eingehenden AX-Signals an drei verschiedene Empfänger innerhalb eines 4diac-Systems.



Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN** - AX-Adapter (Socket) - Empfängt das eingehende AX-Signal

Ausgangsadapter:

- **OUT1** - AX-Adapter (Plug) - Erster Ausgangskanal
- **OUT2** - AX-Adapter (Plug) - Zweiter Ausgangskanal
- **OUT3** - AX-Adapter (Plug) - Dritter Ausgangskanal

Funktionsweise

Der Funktionsblock arbeitet als Signalverteiler für unidirektionale AX-Adapter. Jedes eingehende Signal am IN-Adapter wird parallel an alle drei Ausgangsadapter (OUT1, OUT2, OUT3) weitergeleitet. Die Verteilung erfolgt synchron, sodass alle Ausgänge gleichzeitig aktiviert werden.

Technische Besonderheiten

- Verwendet unidirektionale AX-Adapter für die Kommunikation
- Implementiert als generischer Funktionsblock (GEN_AX_SPLIT)
- Keine Ereignis- oder Dateneingänge - arbeitet ausschließlich über Adapter
- Plug-and-Socket Architektur gemäß IEC 61499 Standard

Zustandsübersicht

Der Funktionsblock besitzt einen einfachen Zustand: Im Betriebszustand leitet er eingehende Signale unverändert an alle drei Ausgänge weiter. Es gibt keine internen Zustandsübergänge oder Verzögerungen.

Anwendungsszenarien

- Verteilung von Steuersignalen an mehrere Aktoren
- Parallele Ansteuerung mehrerer Geräte mit demselben Signal
- Signalverzweigung in komplexen Steuerungsarchitekturen
- Redundante Signalverteilung für Sicherheitsanwendungen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Verteilungsbausteinen zeichnet sich AX_SPLIT_3 durch seine spezifische Ausrichtung auf AX-Adapter aus. Während allgemeine Verteilungsbausteine verschiedene Adaptertypen unterstützen können, ist dieser Baustein speziell für AX-Adapter optimiert.

Vergleich mit E_SPLIT

- **`AX_SPLIT_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

🛠️ Zugehörige Übungen

- Uebung_002a5b_AX
- Uebung_006a3_AX
- Uebung_009_AX

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der AX_SPLIT_3 bietet eine einfache und effiziente Lösung zur Signalverteilung in 4diac-Systemen. Seine generische Implementierung und standardkonforme Adapter-Schnittstelle machen ihn zu einem vielseitig einsetzbaren Baustein für verschiedene Automatisierungsanwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

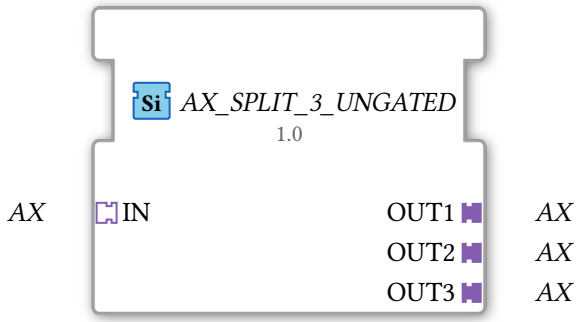
AX_SPLIT_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AX_SPLIT_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_SPLIT_3_UNGATED ist ein generischer Funktionsblock, der einen AX-Adapter-Eingang auf drei separate AX-Adapter-Ausgänge aufteilt. Der Baustein ermöglicht die Verteilung eines eingehenden AX-Signals an drei verschiedene Empfänger innerhalb eines 4diac-Systems.



Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN** - AX-Adapter (Socket) - Empfängt das eingehende AX-Signal

Ausgangsadapter:

- **OUT1** - AX-Adapter (Plug) - Erster Ausgangskanal
- **OUT2** - AX-Adapter (Plug) - Zweiter Ausgangskanal
- **OUT3** - AX-Adapter (Plug) - Dritter Ausgangskanal

Funktionsweise

Der Funktionsblock arbeitet als Signalverteiler für unidirektionale AX-Adapter. Jedes eingehende Signal am IN-Adapter wird parallel an alle drei Ausgangsadapter (OUT1, OUT2, OUT3) weitergeleitet. Die Verteilung erfolgt synchron, sodass alle Ausgänge gleichzeitig aktiviert werden.

Technische Besonderheiten

- Verwendet unidirektionale AX-Adapter für die Kommunikation
- Implementiert als generischer Funktionsblock (GEN_AX_SPLIT)
- Keine Ereignis- oder Dateneingänge - arbeitet ausschließlich über Adapter
- Plug-and-Socket Architektur gemäß IEC 61499 Standard

Zustandsübersicht

Der Funktionsblock besitzt einen einfachen Zustand: Im Betriebszustand leitet er eingehende Signale unverändert an alle drei Ausgänge weiter. Es gibt keine internen Zustandsübergänge oder Verzögerungen.

Anwendungsszenarien

- Verteilung von Steuersignalen an mehrere Aktoren

- Parallele Ansteuerung mehrerer Geräte mit demselben Signal
- Signalverzweigung in komplexen Steuerungsarchitekturen
- Redundante Signalverteilung für Sicherheitsanwendungen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Verteilungsbausteinen zeichnet sich AX_SPLIT_3_UNGATED durch seine spezifische Ausrichtung auf AX-Adapter aus. Während allgemeine Verteilungsbausteine verschiedene Adaptertypen unterstützen können, ist dieser Baustein speziell für AX-Adapter optimiert.

Vergleich mit E_SPLIT

- **AX_SPLIT_3**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

🛠️ Zugehörige Übungen

- Uebung_002a5b_AX
- Uebung_006a3_AX
- Uebung_009_AX

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der AX_SPLIT_3_UNGATED bietet eine einfache und effiziente Lösung zur Signalverteilung in 4diac-Systemen. Seine generische Implementierung und standardkonforme Adapter-Schnittstelle machen ihn zu einem vielseitig einsetzbaren Baustein für verschiedene Automatisierungsanwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

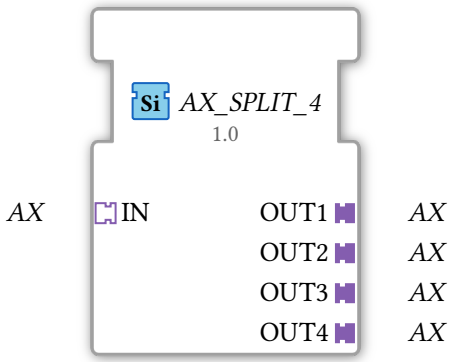
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AX_SPLIT_4

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_SPLIT_4 Funktionsblock ist ein generischer Baustein, der einen AX-Adapter-Eingang auf vier separate AX-Adapter-Ausgänge aufteilt. Der Baustein dient als Verteiler für unidirektionale AX-Adapter und ermöglicht die Weitergabe von Daten und Ereignissen an mehrere Empfänger.



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden (Ereignisverarbeitung erfolgt über Adapter)

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden (Ereignisverarbeitung erfolgt über Adapter)

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden (Datenverarbeitung erfolgt über Adapter)

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden (Datenverarbeitung erfolgt über Adapter)

Adapter

Eingangsadapter:

- `IN` - AX-Adapter (Socket) - Empfängt unidirektionale AX-Daten und Ereignisse

Ausgangsadapter:

- `OUT1` - AX-Adapter (Plug) - Erster Ausgangskanal
- `OUT2` - AX-Adapter (Plug) - Zweiter Ausgangskanal
- `OUT3` - AX-Adapter (Plug) - Dritter Ausgangskanal
- `OUT4` - AX-Adapter (Plug) - Vierter Ausgangskanal

Funktionsweise

Der AX_SPLIT_4 Block empfängt Daten und Ereignisse über den Eingangsadapter `IN` und verteilt diese parallel an alle vier Ausgangsadapter (`OUT1` bis `OUT4`). Alle eingehenden Informationen werden gleichzeitig an alle Ausgänge weitergeleitet, wodurch eine 1:4-Verteilung realisiert wird.

Technische Besonderheiten

- Generische Implementierung für maximale Wiederverwendbarkeit
- Verwendet unidirektionale AX-Adapter für die Datenübertragung
- Keine Pufferung oder Verzögerung der Datenweitergabe
- Parallele Verteilung ohne Priorisierung der Ausgänge

Zustandsübersicht

Der Baustein arbeitet zustandslos - eingehende Daten und Ereignisse werden sofort an alle Ausgänge weitergeleitet ohne internen Zustand zu speichern.

Anwendungsszenarien

- Verteilung von Steuerbefehlen an mehrere Aktoren
- Broadcasting von Sensordaten an verschiedene Verarbeitungseinheiten
- Aufteilung von Steuerinformationen in verteilten Systemen
- Redundante Datenverteilung für Sicherheitsanwendungen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen Split-Bausteinen bietet AX_SPLIT_4 eine spezifische 1:4-Aufteilung für AX-Adapter. Andere Split-Varianten könnten unterschiedliche Anzahlen von Ausgängen oder andere Adaptertypen unterstützen.

Vergleich mit E_SPLIT

- **`AX_SPLIT_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der AX_SPLIT_4 Funktionsblock bietet eine einfache und effiziente Lösung zur Verteilung von AX-Adapter-Daten auf vier parallele Ausgänge. Seine generische Natur und die Verwendung standardisierter Adapter machen ihn vielseitig einsetzbar in verschiedenen Automatisierungsanwendungen.

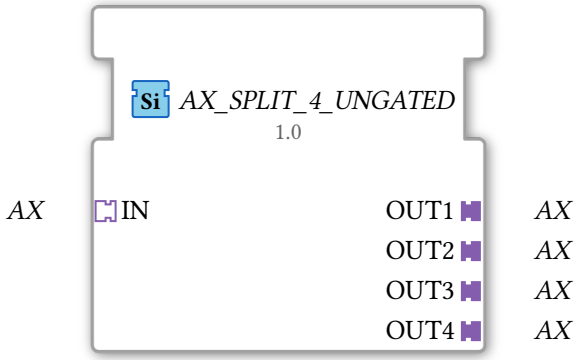
AX_SPLIT_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AX_SPLIT_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_SPLIT_4_UNGATED Funktionsblock ist ein generischer Baustein, der einen AX-Adapter-Eingang auf vier separate AX-Adapter-Ausgänge aufteilt. Der Baustein dient als Verteiler für unidirektionale AX-Adapter und ermöglicht die Weitergabe von Daten und Ereignissen an mehrere Empfänger.



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden (Ereignisverarbeitung erfolgt über Adapter)

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden (Ereignisverarbeitung erfolgt über Adapter)

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden (Datenverarbeitung erfolgt über Adapter)

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden (Datenverarbeitung erfolgt über Adapter)

Adapter

Eingangsadapter:

- `IN` - AX-Adapter (Socket) - Empfängt unidirektionale AX-Daten und Ereignisse

Ausgangsadapter:

- `OUT1` - AX-Adapter (Plug) - Erster Ausgangskanal
- `OUT2` - AX-Adapter (Plug) - Zweiter Ausgangskanal
- `OUT3` - AX-Adapter (Plug) - Dritter Ausgangskanal
- `OUT4` - AX-Adapter (Plug) - Vierter Ausgangskanal

Funktionsweise

Der AX_SPLIT_4_UNGATED Block empfängt Daten und Ereignisse über den Eingangsadapter `IN` und verteilt diese parallel an alle vier Ausgangsadapter (`OUT1` bis `OUT4`). Alle eingehenden Informationen werden gleichzeitig an alle Ausgänge weitergeleitet, wodurch eine 1:4-Verteilung realisiert wird.

Technische Besonderheiten

- Generische Implementierung für maximale Wiederverwendbarkeit
- Verwendet unidirektionale AX-Adapter für die Datenübertragung
- Keine Pufferung oder Verzögerung der Datenweitergabe
- Parallele Verteilung ohne Priorisierung der Ausgänge

Zustandsübersicht

Der Baustein arbeitet zustandslos - eingehende Daten und Ereignisse werden sofort an alle Ausgänge weitergeleitet ohne internen Zustand zu speichern.

Anwendungsszenarien

- Verteilung von Steuerbefehlen an mehrere Aktoren
- Broadcasting von Sensordaten an verschiedene Verarbeitungseinheiten
- Aufteilung von Steuerinformationen in verteilten Systemen
- Redundante Datenverteilung für Sicherheitsanwendungen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen Split-Bausteinen bietet AX_SPLIT_4_UNGATED eine spezifische 1:4-Aufteilung für AX-Adapter. Andere Split-Varianten könnten unterschiedliche Anzahlen von Ausgängen oder andere Adaptertypen unterstützen.

Vergleich mit E_SPLIT

- **AX_SPLIT_4**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

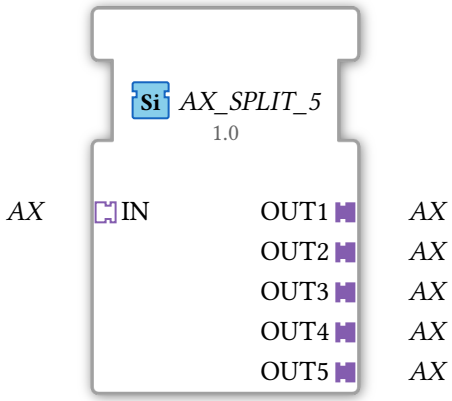
Der AX_SPLIT_4_UNGATED Funktionsblock bietet eine einfache und effiziente Lösung zur Verteilung von AX-Adapter-Daten auf vier parallele Ausgänge. Seine generische Natur und die Verwendung standardisierter Adapter machen ihn vielseitig einsetzbar in verschiedenen Automatisierungsanwendungen.

AX_SPLIT_5

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_SPLIT_5 Funktionsblock ist ein generischer Baustein, der dazu dient, einen AX-Adapter in fünf separate AX-Ausgänge aufzuteilen. Der Block ermöglicht die Verteilung eines Eingangssignals auf mehrere Ausgangskanäle und findet insbesondere in Steuerungsanwendungen Verwendung, wo Signale auf verschiedene Komponenten verteilt werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge
Keine direkten Ereignis-Eingänge vorhanden (Ereignisverarbeitung erfolgt über Adapter)

Ereignis-Ausgänge
Keine direkten Ereignis-Ausgänge vorhanden (Ereignisverarbeitung erfolgt über Adapter)

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN** - AX unidirektionaler Adapter (Socket)

Ausgangsadapter:

- **OUT1** - AX unidirektionaler Adapter (Plug)
- **OUT2** - AX unidirektionaler Adapter (Plug)
- **OUT3** - AX unidirektionaler Adapter (Plug)
- **OUT4** - AX unidirektionaler Adapter (Plug)
- **OUT5** - AX unidirektionaler Adapter (Plug)

Funktionsweise
Der AX_SPLIT_5 Funktionsblock empfängt Signale über den Eingangsadapter IN und verteilt diese gleichzeitig auf alle fünf Ausgangsadapter (OUT1 bis OUT5). Jedes am IN-Adapter eingehende Ereignis wird parallel an alle Ausgänge weitergeleitet, wodurch eine 1:5-Verteilung realisiert wird.

Technische Besonderheiten
- Generischer Funktionsblock mit der Klasse ‚GEN_AX_SPLIT‘
- Verwendet unidirektionale AX-Adapter für die Kommunikation
- Keine Datenverarbeitung oder -transformation
- Reine Ereignisverteilung ohne Verzögerung

Zustandsübersicht
Der Block besitzt einen einfachen Zustand: Im Ruhezustand wartet er auf eingehende Ereignisse am IN-Adapter. Bei Empfang eines Ereignisses werden sofort alle fünf Ausgangsadapter aktiviert.

Anwendungsszenarien
- Verteilung von Steuersignalen auf multiple Aktoren
- Aufteilung von Sensorwerten auf verschiedene Verarbeitungseinheiten
- Parallele Aktivierung mehrerer Subsysteme
- Signalverteilung in dezentralen Steuerungsarchitekturen

⚖️ Vergleich mit ähnlichen Bausteinen
Im Vergleich zu einfachen Split-Bausteinen bietet AX_SPLIT_5 eine feste Anzahl von fünf Ausgängen. Andere Split-Varianten können unterschiedliche Ausgangszahlen oder konfigurierbare Verteilungslogiken bieten. Der Fokus liegt hier auf der einfachen, festen 1:5-Verteilung.
Vergleich mit E_SPLIT

- `AX_SPLIT_5_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs

unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der AX_SPLIT_5 ist ein spezialisierter Verteilerbaustein, der durch seine einfache und deterministische Funktionsweise ideal für Anwendungen geeignet ist, bei denen ein Signal auf genau fünf Empfänger verteilt werden muss. Die Verwendung standardisierter AX-Adapter gewährleistet Kompatibilität innerhalb bestehender Steuerungssysteme.

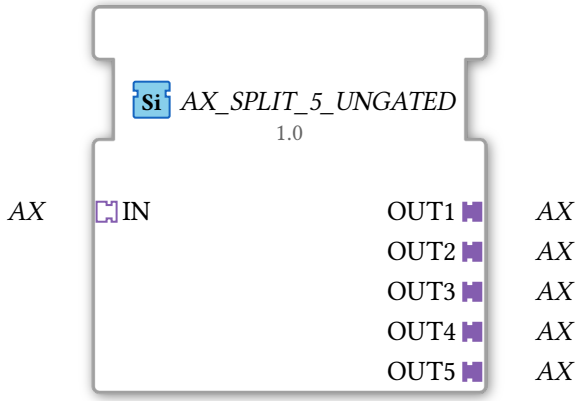
AX_SPLIT_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AX_SPLIT_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_SPLIT_5_UNGATED Funktionsblock ist ein generischer Baustein, der dazu dient, einen AX-Adapter in fünf separate AX-Ausgänge aufzuteilen. Der Block ermöglicht die Verteilung eines Eingangssignals auf mehrere Ausgangskanäle und findet insbesondere in Steuerungsanwendungen Verwendung, wo Signale auf verschiedene Komponenten verteilt werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden (Ereignisverarbeitung erfolgt über Adapter)

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden (Ereignisverarbeitung erfolgt über Adapter)

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN** - AX unidirektionaler Adapter (Socket)

Ausgangsadapter:

- **OUT1** - AX unidirektionaler Adapter (Plug)
- **OUT2** - AX unidirektionaler Adapter (Plug)
- **OUT3** - AX unidirektionaler Adapter (Plug)
- **OUT4** - AX unidirektionaler Adapter (Plug)
- **OUT5** - AX unidirektionaler Adapter (Plug)

Funktionsweise

Der AX_SPLIT_5_UNGATED Funktionsblock empfängt Signale über den Eingangsadapter IN und verteilt diese gleichzeitig auf alle fünf Ausgangsadapter (OUT1 bis OUT5). Jedes am IN-Adapter eingehende Ereignis wird parallel an alle Ausgänge weitergeleitet, wodurch eine 1:5-Verteilung realisiert wird.

Technische Besonderheiten

- Generischer Funktionsblock mit der Klasse ‚GEN_AX_SPLIT'
- Verwendet unidirektionale AX-Adapter für die Kommunikation
- Keine Datenverarbeitung oder -transformation
- Reine Ereignisverteilung ohne Verzögerung

Zustandsübersicht

Der Block besitzt einen einfachen Zustand: Im Ruhezustand wartet er auf eingehende Ereignisse am IN-Adapter. Bei Empfang eines Ereignisses werden sofort alle fünf Ausgangsadapter aktiviert.

Anwendungsszenarien

- Verteilung von Steuersignalen auf multiple Aktoren
- Aufteilung von Sensorwerten auf verschiedene Verarbeitungseinheiten
- Parallele Aktivierung mehrerer Subsysteme
- Signalverteilung in dezentralen Steuerungsarchitekturen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen Split-Bausteinen bietet AX_SPLIT_5_UNGATED eine feste Anzahl von fünf Ausgängen. Andere Split-Varianten können unterschiedliche Ausgangszahlen oder konfigurierbare Verteilungslogiken bieten. Der Fokus liegt hier auf der einfachen, festen 1:5-Verteilung.

Vergleich mit E_SPLIT

- `AX_SPLIT_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

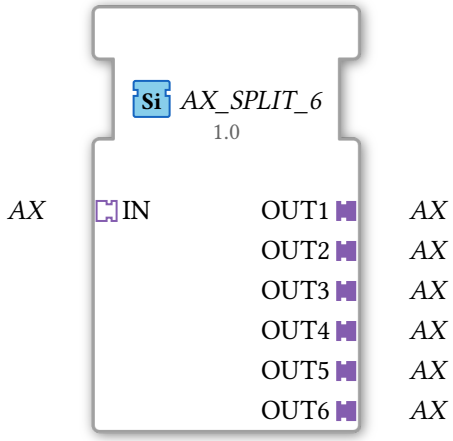
Der AX_SPLIT_5_UNGATED ist ein spezialisierter Verteilerbaustein, der durch seine einfache und deterministische Funktionsweise ideal für Anwendungen geeignet ist, bei denen ein Signal auf genau fünf Empfänger verteilt werden muss. Die Verwendung standardisierter AX-Adapter gewährleistet Kompatibilität innerhalb bestehender Steuerungssysteme.

AX_SPLIT_6

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_SPLIT_6 Funktionsblock ist ein generischer Baustein, der dazu dient, einen einzelnen AX-Adapter in sechs separate AX-Ausgänge aufzuteilen. Der Baustein ermöglicht die Verteilung von Signalen und Datenströmen von einer zentralen Quelle zu mehreren Verbrauchern.



Schnittstellenstruktur

Ereignis-Eingänge
Keine direkten Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN** (Socket): Empfängt die eingehenden AX-Signale vom Quellsystem

Ausgangsadapter:

- **OUT1** (Plug): Erster Ausgangskanal
- **OUT2** (Plug): Zweiter Ausgangskanal
- **OUT3** (Plug): Dritter Ausgangskanal
- **OUT4** (Plug): Vierter Ausgangskanal
- **OUT5** (Plug): Fünfter Ausgangskanal
- **OUT6** (Plug): Sechster Ausgangskanal

Alle Adapter verwenden den Typ: `adapter::types::unidirectional::AX`

Funktionsweise
Der AX_SPLIT_6 Baustein fungiert als Signalverteiler. Er empfängt Signale über den Eingangsadapter IN und leitet diese parallel an alle sechs Ausgangsadapter (OUT1 bis OUT6) weiter. Da es sich um unidirektionale AX-Adapter handelt, erfolgt die Signalübertragung nur in eine Richtung - vom Eingang zu den Ausgängen.

Technische Besonderheiten

- Generischer Funktionsblock mit der Klasse ‚GEN_AX_SPLIT‘
- Verwendet unidirektionale AX-Adapter für die Kommunikation
- Keine Datenverarbeitung oder -modifikation, nur Signalverteilung
- Parallele Ausgabe an alle sechs Ausgänge

Zustandsübersicht
Der Baustein besitzt einen einfachen Zustand:

- **Bereit**: Empfängt Signale von IN und verteilt sie an OUT1-OUT6

Anwendungsszenarien

- Verteilung von Steuersignalen an multiple Aktoren
- Aufteilung von Sensorwerten an verschiedene Verarbeitungseinheiten
- Signalverteilung in redundanten Systemen
- Multi-Casting von Steuerbefehlen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Splitter-Bausteinen bietet AX_SPLIT_6 eine feste Anzahl von sechs Ausgängen, was für spezifische Anwendungsfälle vorteilhaft ist. Andere Splitter können variabel mehr oder weniger Ausgänge bieten, sind aber möglicherweise komplexer in der Konfiguration.

Vergleich mit E_SPLIT

- `AX_SPLIT_6_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

AX_SPLIT_6 ist ein spezialisierter Verteilerbaustein, der sich ideal für Anwendungen eignet, bei denen ein Signal an genau sechs Empfänger verteilt werden muss. Die einfache Struktur und die Verwendung standardisierter AX-Adapter machen ihn zu einer zuverlässigen Lösung für Signalverteilungsaufgaben in automatisierungstechnischen Systemen.

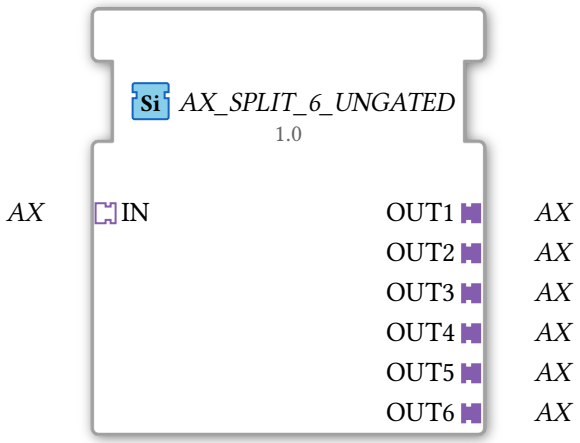
AX_SPLIT_6_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AX_SPLIT_6`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_SPLIT_6_UNGATED Funktionsblock ist ein generischer Baustein, der dazu dient, einen einzelnen AX-Adapter in sechs separate AX-Ausgänge aufzuteilen. Der Baustein ermöglicht die Verteilung von Signalen und Datenströmen von einer zentralen Quelle zu mehreren Verbrauchern.



Schnittstellenstruktur

Ereignis-Eingänge
Keine direkten Ereignis-Eingänge vorhanden

Ereignis-Ausgänge
Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
Eingangsadapter:

- **IN** (Socket): Empfängt die eingehenden AX-Signale vom Quellsystem

Ausgangsadapter:

- **OUT1** (Plug): Erster Ausgangskanal
- **OUT2** (Plug): Zweiter Ausgangskanal
- **OUT3** (Plug): Dritter Ausgangskanal
- **OUT4** (Plug): Vierter Ausgangskanal
- **OUT5** (Plug): Fünfter Ausgangskanal
- **OUT6** (Plug): Sechster Ausgangskanal

Alle Adapter verwenden den Typ: `adapter::types::unidirectional::AX`

Funktionsweise
Der AX_SPLIT_6_UNGATED Baustein fungiert als Signalverteiler. Er empfängt Signale über den Eingangsadapter IN und leitet diese parallel an alle sechs Ausgangsadapter (OUT1 bis OUT6) weiter. Da es sich um unidirektionale AX-Adapter handelt, erfolgt die Signalübertragung nur in eine Richtung - vom Eingang zu den Ausgängen.

Technische Besonderheiten
- Generischer Funktionsblock mit der Klasse ‚GEN_AX_SPLIT‘
- Verwendet unidirektionale AX-Adapter für die Kommunikation
- Keine Datenverarbeitung oder -modifikation, nur Signalverteilung
- Parallele Ausgabe an alle sechs Ausgänge

Zustandsübersicht
Der Baustein besitzt einen einfachen Zustand:

- **Bereit**: Empfängt Signale von IN und verteilt sie an OUT1-OUT6

Anwendungsszenarien
- Verteilung von Steuersignalen an multiple Aktoren
- Aufteilung von Sensorwerten an verschiedene Verarbeitungseinheiten
- Signalverteilung in redundanten Systemen
- Multi-Casting von Steuerbefehlen

⚖️ Vergleich mit ähnlichen Bausteinen
Im Vergleich zu anderen Splitter-Bausteinen bietet AX_SPLIT_6_UNGATED eine feste Anzahl von sechs Ausgängen, was für spezifische Anwendungsfälle vorteilhaft ist. Andere Splitter können variabel mehr oder weniger Ausgänge bieten, sind aber möglicherweise komplexer in der Konfiguration.
Vergleich mit E_SPLIT

- **`AX_SPLIT_6`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

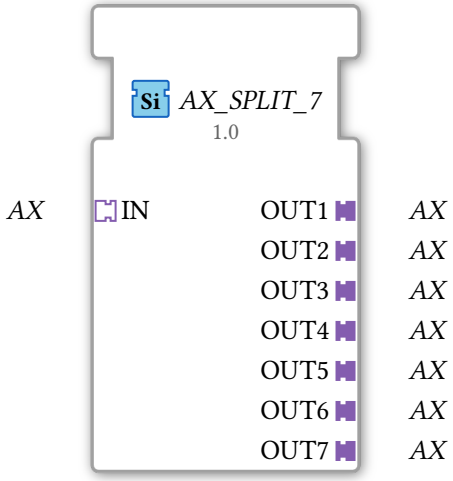
AX_SPLIT_6_UNGATED ist ein spezialisierter Verteilerbaustein, der sich ideal für Anwendungen eignet, bei denen ein Signal an genau sechs Empfänger verteilt werden muss. Die einfache Struktur und die Verwendung standardisierter AX-Adapter machen ihn zu einer zuverlässigen Lösung für Signalverteilungsaufgaben in automatisierungstechnischen Systemen.

AX_SPLIT_7

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_SPLIT_7 Funktionsblock ist ein generischer Baustein, der einen einzelnen AX-Adapter-Eingang auf sieben separate AX-Adapter-Ausgänge aufteilt. Der Baustein dient als Verteiler für unidirektionale AX-Adapter und ermöglicht die Verteilung eines Eingangssignals an mehrere Empfänger.



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden (Adapter-basierte Kommunikation)

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden (Adapter-basierte Kommunikation)

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden (Adapter-basierte Kommunikation)

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden (Adapter-basierte Kommunikation)

Adapter

Eingangsadapter:

- **IN** (Socket): Unidirektionaler AX-Adapter-Eingang

Ausgangsadapter:

- **OUT1** (Plug): Unidirektionaler AX-Adapter-Ausgang 1
- **OUT2** (Plug): Unidirektionaler AX-Adapter-Ausgang 2
- **OUT3** (Plug): Unidirektionaler AX-Adapter-Ausgang 3
- **OUT4** (Plug): Unidirektionaler AX-Adapter-Ausgang 4

- **OUT5** (Plug): Unidirektionaler AX-Adapter-Ausgang 5
- **OUT6** (Plug): Unidirektionaler AX-Adapter-Ausgang 6
- **OUT7** (Plug): Unidirektionaler AX-Adapter-Ausgang 7

Funktionsweise

Der AX_SPLIT_7 Funktionsblock empfängt Daten über den Eingangsadapter IN und verteilt diese gleichzeitig an alle sieben Ausgangsadapter (OUT1 bis OUT7). Jedes am IN-Adapter eingehende Signal wird parallel an alle sieben Ausgänge weitergeleitet, wodurch eine 1:7-Verteilung realisiert wird.

Technische Besonderheiten

- Generische Implementierung für maximale Wiederverwendbarkeit
- Verwendet unidirektionale AX-Adapter für die Kommunikation
- Keine internen Verzögerungen bei der Signalverteilung
- Parallele Ausgabe an alle sieben Ausgänge

Zustandsübersicht

Der Funktionsblock besitzt einen einfachen Zustand: Im aktiven Zustand leitet er eingehende Signale sofort an alle Ausgänge weiter. Es gibt keine komplexen Zustandsübergänge oder interne Verarbeitungslogik.

Anwendungsszenarien

- Verteilung von Steuersignalen an multiple Aktoren
- Aufteilung von Sensordaten an verschiedene Verarbeitungseinheiten
- Signalverteilung in verzweigten Steuerungsarchitekturen
- Redundante Signalweiterleitung an mehrere Empfänger

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren Splitter-Bausteinen bietet AX_SPLIT_7 eine feste Anzahl von sieben Ausgängen, was für spezifische Anwendungsfälle mit genau sieben Empfängern optimiert ist. Gegenüber variablen Splittern hat dieser Baustein den Vorteil der fest definierten Schnittstellen.

Vergleich mit E_SPLIT

- `AX_SPLIT_7_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der AX_SPLIT_7 Funktionsblock stellt eine effiziente Lösung für die Verteilung unidirektionaler AX-Signale an sieben Empfänger dar. Seine generische Natur und einfache Funktionsweise machen ihn zu einem robusten und zuverlässigen Baustein für verteilte Steuerungssysteme.

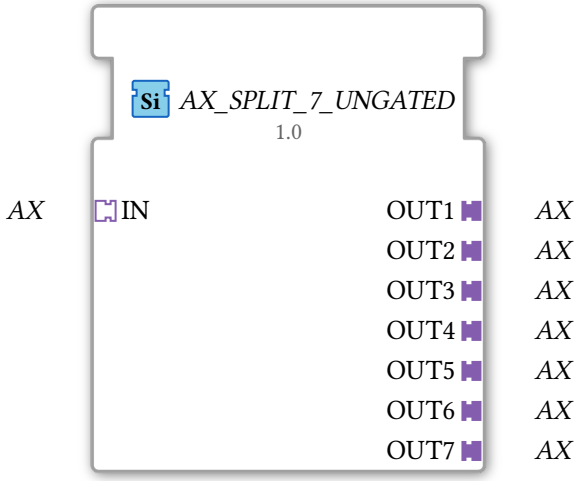
AX_SPLIT_7_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AX_SPLIT_7`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_SPLIT_7_UNGATED Funktionsblock ist ein generischer Baustein, der einen einzelnen AX-Adapter-Eingang auf sieben separate AX-Adapter-Ausgänge aufteilt. Der Baustein dient als Verteiler für unidirektionale AX-Adapter und ermöglicht die Verteilung eines Eingangssignals an mehrere Empfänger.



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden (Adapter-basierte Kommunikation)

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden (Adapter-basierte Kommunikation)

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden (Adapter-basierte Kommunikation)

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden (Adapter-basierte Kommunikation)

Adapter

Eingangsadapter:

- **IN** (Socket): Unidirektionaler AX-Adapter-Eingang

Ausgangsadapter:

- **OUT1** (Plug): Unidirektionaler AX-Adapter-Ausgang 1
- **OUT2** (Plug): Unidirektionaler AX-Adapter-Ausgang 2
- **OUT3** (Plug): Unidirektionaler AX-Adapter-Ausgang 3
- **OUT4** (Plug): Unidirektionaler AX-Adapter-Ausgang 4
- **OUT5** (Plug): Unidirektionaler AX-Adapter-Ausgang 5
- **OUT6** (Plug): Unidirektionaler AX-Adapter-Ausgang 6
- **OUT7** (Plug): Unidirektionaler AX-Adapter-Ausgang 7

Funktionsweise

Der AX_SPLIT_7_UNGATED Funktionsblock empfängt Daten über den Eingangsadapter IN und verteilt diese gleichzeitig an alle sieben Ausgangsadapter (OUT1 bis OUT7). Jedes am IN-Adapter eingehende Signal wird parallel an alle sieben Ausgänge weitergeleitet, wodurch eine 1:7-Verteilung realisiert wird.

Technische Besonderheiten

- Generische Implementierung für maximale Wiederverwendbarkeit
- Verwendet unidirektionale AX-Adapter für die Kommunikation
- Keine internen Verzögerungen bei der Signalverteilung
- Parallele Ausgabe an alle sieben Ausgänge

Zustandsübersicht

Der Funktionsblock besitzt einen einfachen Zustand: Im aktiven Zustand leitet er eingehende Signale sofort an alle Ausgänge weiter. Es gibt keine komplexen Zustandsübergänge oder interne Verarbeitungslogik.

Anwendungsszenarien

- Verteilung von Steuersignalen an multiple Aktoren
- Aufteilung von Sensordaten an verschiedene Verarbeitungseinheiten
- Signalverteilung in verzweigten Steuerungsarchitekturen
- Redundante Signalweiterleitung an mehrere Empfänger

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren Splitter-Bausteinen bietet AX_SPLIT_7_UNGATED eine feste Anzahl von sieben Ausgängen, was für spezifische Anwendungsfälle mit genau sieben Empfängern optimiert ist. Gegenüber variablen Splittern hat dieser Baustein den Vorteil der fest definierten Schnittstellen.

Vergleich mit E_SPLIT

- `AX_SPLIT_7`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

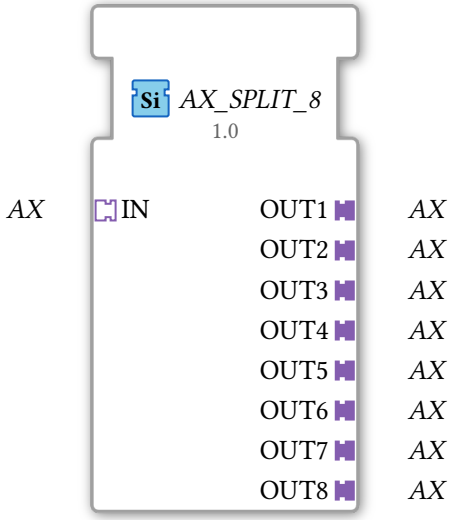
Der AX_SPLIT_7_UNGATED Funktionsblock stellt eine effiziente Lösung für die Verteilung unidirektionaler AX-Signale an sieben Empfänger dar. Seine generische Natur und einfache Funktionsweise machen ihn zu einem robusten und zuverlässigen Baustein für verteilte Steuerungssysteme.

AX_SPLIT_8

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_SPLIT_8 Funktionsblock ist ein generischer Baustein, der einen einzelnen AX-Adapter auf acht separate AX-Ausgänge aufteilt. Er dient zur Verteilung von Adapter-Signalen in komplexen Steuerungsarchitekturen.



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN** (Socket): AX-Adapter-Eingang vom Typ `adapter::types::unidirectional::AX`

Ausgangsadapter:

- **OUT1** (Plug): AX-Adapter-Ausgang 1 vom Typ `adapter::types::unidirectional::AX`
- **OUT2** (Plug): AX-Adapter-Ausgang 2 vom Typ `adapter::types::unidirectional::AX`
- **OUT3** (Plug): AX-Adapter-Ausgang 3 vom Typ `adapter::types::unidirectional::AX`
- **OUT4** (Plug): AX-Adapter-Ausgang 4 vom Typ `adapter::types::unidirectional::AX`
- **OUT5** (Plug): AX-Adapter-Ausgang 5 vom Typ `adapter::types::unidirectional::AX`
- **OUT6** (Plug): AX-Adapter-Ausgang 6 vom Typ `adapter::types::unidirectional::AX`
- **OUT7** (Plug): AX-Adapter-Ausgang 7 vom Typ `adapter::types::unidirectional::AX`
- **OUT8** (Plug): AX-Adapter-Ausgang 8 vom Typ `adapter::types::unidirectional::AX`

Funktionsweise

Der AX_SPLIT_8 Funktionsblock verteilt eingehende Adapter-Signale vom Eingangsadapter IN gleichzeitig an alle acht Ausgangsadapter (OUT1 bis OUT8). Als unidirektionaler Adapter-Baustein leitet er alle eingehenden Ereignisse und Daten parallel an alle angeschlossenen Ausgänge weiter.

Technische Besonderheiten

- Generischer Funktionsblock mit der generischen Klasse ‚GEN_AX_SPLIT‘
- Verwendet unidirektionale AX-Adapter
- Keine interne Signalverarbeitung oder -filterung
- Parallele Signalverteilung ohne Verzögerung

Zustandsübersicht

Der Baustein besitzt keinen internen Zustand und arbeitet stateless. Alle eingehenden Signale werden sofort an alle Ausgänge weitergeleitet.

Anwendungsszenarien

- Verteilung von Steuersignalen an multiple Aktoren
- Aufteilung von Sensorinformationen an verschiedene Verarbeitungseinheiten
- Signalverteilung in redundanten Systemarchitekturen
- Kaskadierung von Adapter-basierten Steuerungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Split-Bausteinen bietet AX_SPLIT_8 eine feste Anzahl von acht Ausgängen und arbeitet ausschließlich auf Adapter-Ebene. Andere Split-Komponenten können variabel viele Ausgänge besitzen oder auf Daten- bzw. Ereignisebene operieren.
Vergleich mit E_SPLIT

- `AX_SPLIT_8_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der AX_SPLIT_8 Funktionsblock stellt eine einfache und effiziente Lösung zur Verteilung von Adapter-Signalen in 4diac-Systemen dar. Seine feste Ausgangsanzahl von acht Ports macht ihn besonders geeignet für standardisierte Verteilungsaufgaben in industriellen Steuerungsanwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

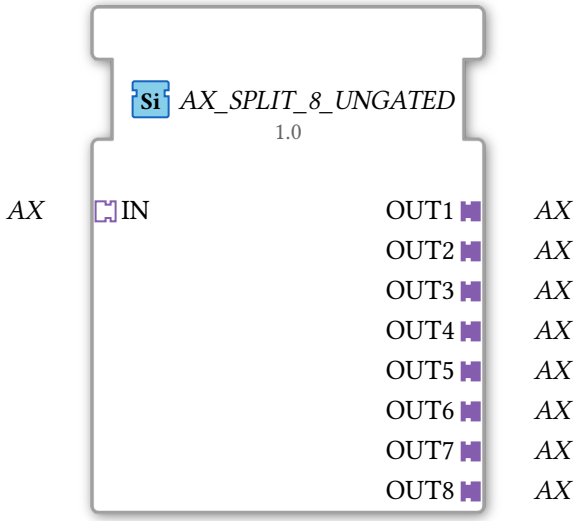
AX_SPLIT_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AX_SPLIT_8`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_SPLIT_8_UNGATED Funktionsblock ist ein generischer Baustein, der einen einzelnen AX-Adapter auf acht separate AX-Ausgänge aufteilt. Er dient zur Verteilung von Adapter-Signalen in komplexen Steuerungsarchitekturen.



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

Eingangsadapter:

- **IN** (Socket): AX-Adapter-Eingang vom Typ `adapter::types::unidirectional::AX`

Ausgangsadapter:

- **OUT1** (Plug): AX-Adapter-Ausgang 1 vom Typ `adapter::types::unidirectional::AX`
- **OUT2** (Plug): AX-Adapter-Ausgang 2 vom Typ `adapter::types::unidirectional::AX`
- **OUT3** (Plug): AX-Adapter-Ausgang 3 vom Typ `adapter::types::unidirectional::AX`
- **OUT4** (Plug): AX-Adapter-Ausgang 4 vom Typ `adapter::types::unidirectional::AX`
- **OUT5** (Plug): AX-Adapter-Ausgang 5 vom Typ `adapter::types::unidirectional::AX`
- **OUT6** (Plug): AX-Adapter-Ausgang 6 vom Typ `adapter::types::unidirectional::AX`
- **OUT7** (Plug): AX-Adapter-Ausgang 7 vom Typ `adapter::types::unidirectional::AX`
- **OUT8** (Plug): AX-Adapter-Ausgang 8 vom Typ `adapter::types::unidirectional::AX`

Funktionsweise

Der AX_SPLIT_8_UNGATED Funktionsblock verteilt eingehende Adapter-Signale vom Eingangsadapter IN gleichzeitig an alle acht Ausgangsadapter (OUT1 bis OUT8). Als unidirektionaler Adapter-Baustein leitet er alle eingehenden Ereignisse und Daten parallel an alle angeschlossenen Ausgänge weiter.

Technische Besonderheiten

- Generischer Funktionsblock mit der generischen Klasse ‚GEN_AX_SPLIT‘
- Verwendet unidirektionale AX-Adapter
- Keine interne Signalverarbeitung oder -filterung
- Parallele Signalverteilung ohne Verzögerung

Zustandsübersicht

Der Baustein besitzt keinen internen Zustand und arbeitet stateless. Alle eingehenden Signale werden sofort an alle Ausgänge weitergeleitet.

Anwendungsszenarien

- Verteilung von Steuersignalen an multiple Aktoren
- Aufteilung von Sensorinformationen an verschiedene Verarbeitungseinheiten
- Signalverteilung in redundanten Systemarchitekturen
- Kaskadierung von Adapter-basierten Steuerungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Split-Bausteinen bietet AX_SPLIT_8_UNGATED eine feste Anzahl von acht Ausgängen und arbeitet ausschließlich auf Adapter-Ebene. Andere Split-Komponenten können variabel viele Ausgänge besitzen oder auf Daten- bzw. Ereignisebene operieren.

Vergleich mit E_SPLIT

- `AX_SPLIT_8`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der AX_SPLIT_8_UNGATED Funktionsblock stellt eine einfache und effiziente Lösung zur Verteilung von Adapter-Signalen in 4diac-Systemen dar. Seine feste Ausgangsanzahl von acht Ports macht ihn besonders geeignet für standardisierte Verteilungsaufgaben in industriellen Steuerungsanwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

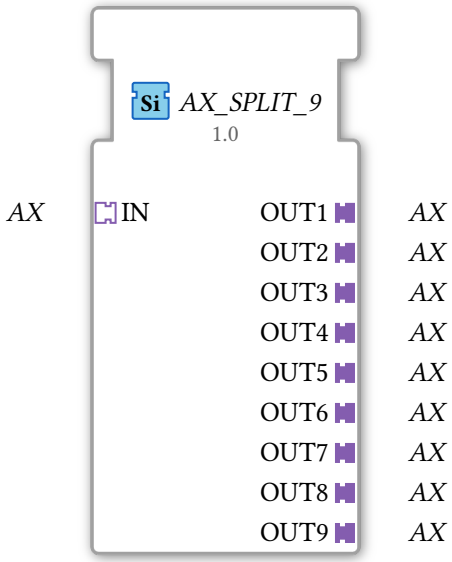
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AX_SPLIT_9

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_SPLIT_9 Funktionsblock ist ein generischer Baustein, der einen einzelnen AX-Adapter auf neun separate AX-Ausgänge aufteilt. Der Block dient als Verteiler für unidirektionale Adapterverbindungen und ermöglicht die Verteilung eines Eingangssignals auf multiple Ausgangskanäle.



Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereigniseingänge vorhanden

Ereignis-Ausgänge

Keine direkten Ereignisausgänge vorhanden

Daten-Eingänge

Keine direkten Dateneingänge vorhanden

Daten-Ausgänge

Keine direkten Datenausgänge vorhanden

Adapter

Eingangsadapter:

- `IN` - Unidirektionaler AX-Adapter (Socket)

Ausgangsadapter:

- `OUT1` bis `OUT9` - Neun unidirektionale AX-Adapter (Plugs)

Funktionsweise

Der AX_SPLIT_9 Block fungiert als Adapter-Splitter, der eingehende Signale und Daten vom Eingangsadapter `IN` parallel auf alle neun Ausgangsadapter (`OUT1` bis `OUT9`) verteilt. Jeder Ausgang erhält dabei eine identische Kopie des Eingangssignals.

Technische Besonderheiten

- Generische Implementierung für maximale Wiederverwendbarkeit
- Unidirektionale Adapterarchitektur
- Parallele Signalverteilung ohne Verzögerung
- Keine Datenverarbeitung oder -modifikation

Zustandsübersicht

Der Funktionsblock besitzt einen einfachen Zustand: Bei Aktivierung des Eingangsadapters werden alle neun Ausgangsadapter gleichzeitig aktiviert.

Anwendungsszenarien

- Verteilung von Steuersignalen auf multiple Aktoren
- Aufteilung von Sensorwerten auf verschiedene Verarbeitungseinheiten
- Signalverteilung in komplexen Steuerungssystemen
- Redundante Signalweiterleitung

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren Splitter-Blöcken bietet AX_SPLIT_9 eine höhere Anzahl an Ausgängen (9 statt typischerweise 2-4). Gegenüber seriellen Verteilern ermöglicht er eine gleichzeitige Aktivierung aller Ausgänge ohne sequentielle Verzögerung.

Vergleich mit E_SPLIT

- `AX_SPLIT_9_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der AX_SPLIT_9 Funktionsblock stellt eine effiziente Lösung für die parallele Verteilung unidirektionaler Adaptersignale dar. Seine generische Natur und die hohe Anzahl an Ausgängen machen ihn besonders geeignet für komplexe Steuerungsanwendungen, bei denen ein Signal an multiple Empfänger verteilt werden muss.

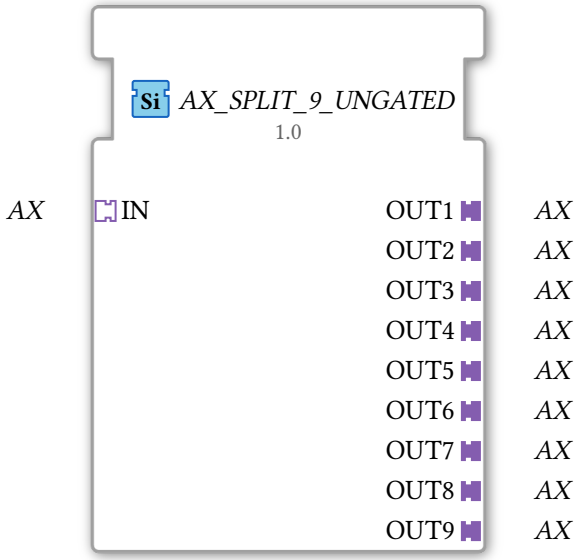
AX_SPLIT_9_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AX_SPLIT_9`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_SPLIT_9_UNGATED Funktionsblock ist ein generischer Baustein, der einen einzelnen AX-Adapter auf neun separate AX-Ausgänge aufteilt. Der Block dient als Verteiler für unidirektionale Adapterverbindungen und ermöglicht die Verteilung eines Eingangssignals auf multiple Ausgangskanäle.



Schnittstellenstruktur

Ereignis-Eingänge
Keine direkten Ereigniseingänge vorhanden

Ereignis-Ausgänge
Keine direkten Ereignisausgänge vorhanden

Daten-Eingänge
Keine direkten Dateneingänge vorhanden

Daten-Ausgänge
Keine direkten Datenausgänge vorhanden

Adapter
Eingangsadapter:

- `IN` - Unidirektionaler AX-Adapter (Socket)

Ausgangsadapter:

- `OUT1` bis `OUT9` - Neun unidirektionale AX-Adapter (Plugs)

Funktionsweise
Der AX_SPLIT_9_UNGATED Block fungiert als Adapter-Splitter, der eingehende Signale und Daten vom Eingangsadapter `IN` parallel auf alle neun Ausgangsadapter (`OUT1` bis `OUT9`) verteilt. Jeder Ausgang erhält dabei eine identische Kopie des Eingangssignals.

Technische Besonderheiten

- Generische Implementierung für maximale Wiederverwendbarkeit
- Unidirektionale Adapterarchitektur
- Parallele Signalverteilung ohne Verzögerung
- Keine Datenverarbeitung oder -modifikation

Zustandsübersicht
Der Funktionsblock besitzt einen einfachen Zustand: Bei Aktivierung des Eingangsadapters werden alle neun Ausgangsadapter gleichzeitig aktiviert.

Anwendungsszenarien

- Verteilung von Steuersignalen auf multiple Aktoren
- Aufteilung von Sensorwerten auf verschiedene Verarbeitungseinheiten

- Signalverteilung in komplexen Steuerungssystemen
- Redundante Signalweiterleitung

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren Splitter-Blöcken bietet AX_SPLIT_9_UNGATED eine höhere Anzahl an Ausgängen (9 statt typischerweise 2-4). Gegenüber seriellen Verteilern ermöglicht er eine gleichzeitige Aktivierung aller Ausgänge ohne sequentielle Verzögerung.

Vergleich mit E_SPLIT

- **AX_SPLIT_9**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

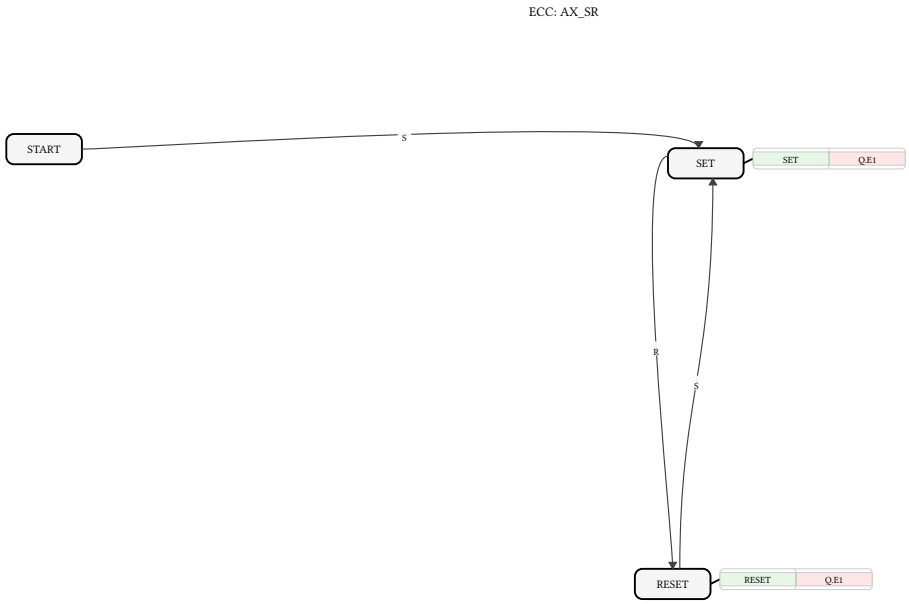
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der AX_SPLIT_9_UNGATED Funktionsblock stellt eine effiziente Lösung für die parallele Verteilung unidirektionaler Adaptersignale dar. Seine generische Natur und die hohe Anzahl an Ausgängen machen ihn besonders geeignet für komplexe Steuerungsanwendungen, bei denen ein Signal an multiple Empfänger verteilt werden muss.

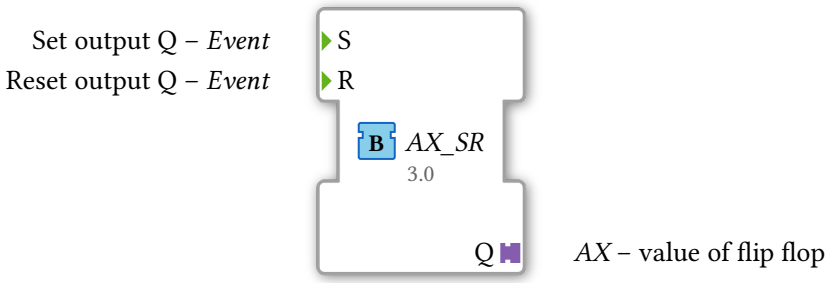
AX_SR



• – ○ • – ○ • – ○ •

Einleitung

Der AX_SR Funktionsblock ist ein ereignisgesteuertes bistabiles Element (Flip-Flop), das als Set-Reset-Speicher fungiert. Er ermöglicht das Setzen und Rücksetzen eines logischen Zustands und stellt diesen über eine Adapterschnittstelle zur Verfügung.



Schnittstellenstruktur

Ereignis-Eingänge

- **S**: Setzt den Ausgang Q auf TRUE
- **R**: Setzt den Ausgang Q auf FALSE

Ereignis-Ausgänge

- Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

- Keine Daten-Eingänge vorhanden

Daten-Ausgänge

- Keine direkten Daten-Ausgänge vorhanden

Adapter

- **Q**: Unidirektionaler Adapter vom Typ „adapter::types::unidirectional::AX“, der den aktuellen Zustand des Flip-Flops bereitstellt

Funktionsweise

Der AX_SR Funktionsblock arbeitet als Set-Reset-Flip-Flop mit folgenden Eigenschaften:

- Bei einem S-Ereignis wird der interne Zustand auf SET gesetzt und der Ausgang Q auf TRUE
- Bei einem R-Ereignis wird der interne Zustand auf RESET gesetzt und der Ausgang Q auf FALSE
- Der Zustand bleibt erhalten, bis ein gegenteiliges Ereignis eintrifft
- Die Zustandsänderungen werden über den Adapter Q kommuniziert

Technische Besonderheiten und Normenvergleich

Wie bei allen ereignisgesteuerten bistabilen Elementen in der IEC 61499 (siehe auch Anmerkung 8 in Tabelle A.1 der DIN EN 61499-1) gibt es hier keine inhärente „Dominanz“ eines Eingangs, wie man sie von der IEC 61131-3 kennt.

- **Vergleich zur IEC 61131-3**: Siehe SR (Bistabil, vorrangig setzen). Während in der klassischen SPS-Welt bei gleichzeitigem TRUE an S1 und R das Setzen gewinnt, wird in der IEC 61499 jedes Ereignis nacheinander verarbeitet. Der Endzustand hängt davon ab, welches Ereignis zuletzt in der Ausführungskette (ECC) abgearbeitet wurde.
- **Funktionale Identität**: `AX_SR` ist funktional identisch zu AX_RS. Die unterschiedliche Benennung und Anordnung der Pins dient lediglich der besseren Orientierung für Entwickler, die an die IEC 61131-3 gewöhnt sind.
- **Adapter-Kommunikation**: Der Baustein stellt seinen Zustand über den Adapter `Q` zur Verfügung.

Zustandsübersicht

1. **START**: Initialzustand
2. **SET**: Zustand nach S-Ereignis, Ausgang Q.D1 = TRUE
3. **RESET**: Zustand nach R-Ereignis, Ausgang Q.D1 = FALSE

Zustandsübergänge:

- START ⟶ SET: Bei S-Ereignis
- SET ⟶ RESET: Bei R-Ereignis

- RESET ⟶ SET: Bei S-Ereignis

Anwendungsszenarien

- Speicherung von Schaltzuständen in verteilten Steuerungsanwendungen
- Zustandsverwaltung über Adapterschnittstellen
- Signalverarbeitung mit Speicherfunktion
- Überwachung von Betriebszuständen

Verwandte Bausteine

- **AX_RS**: Funktional identisch, Eingänge im Symbol vertauscht.
- **E_SR**: Das Standard-Äquivalent mit direkten Daten-/Ereignisausgängen statt Adaptern.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Speicherelementen bietet AX_SR:

- Klare Trennung von Set- und Reset-Funktionalität
- Adapter-basierte Schnittstelle für flexible Integration
- Ereignisgesteuerte Zustandsänderungen
- Einfache und robuste Zustandsverwaltung

Vergleich mit E_SR

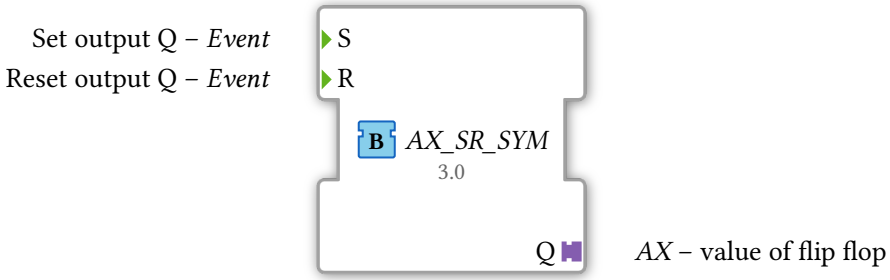
🛠️ Zugehörige Übungen

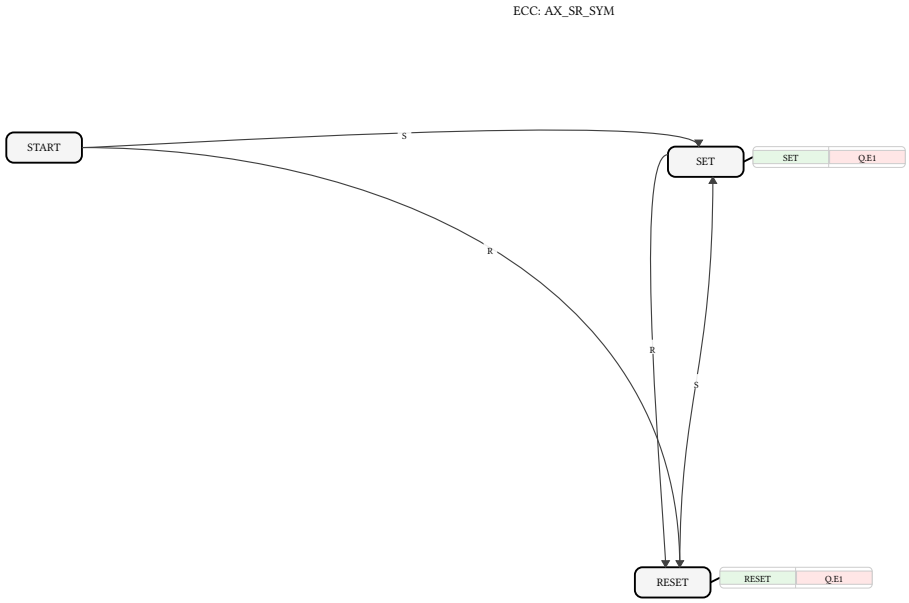
- Uebung_004b_AX
- Uebung_004b_AX_ASR
- Uebung_004b_AX_ASR_X
- Uebung_006_AX
- Uebung_006d_AX
- Uebung_007a3_AX
- Uebung_008_AX
- Uebung_009_AX
- Uebung_013_AX
- Uebung_160b_AX
- Uebung_171_AX

Fazit

Der AX_SR Funktionsblock stellt eine zuverlässige und einfach zu verwendende Lösung für bistabile Speicherfunktionen in verteilten Automatisierungssystemen dar. Durch die Verwendung von Adaptern ermöglicht er eine flexible Integration in verschiedene Systemarchitekturen und bietet eine klare, ereignisgesteuerte Schnittstelle für Set-Reset-Operationen.

AX_SR_SYM





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AX_SR_SYM** ist ein ereignisgesteuertes, bistabiles Flipflop mit symmetrischem Einschaltverhalten. Er implementiert einen Set-Reset-Speicher, der über die Ereignisse `S` (Set) und `R` (Reset) angesteuert wird. Im Gegensatz zu ähnlichen Bausteinen ist der Startzustand nicht vordefiniert; der Baustein reagiert auf das erste eintreffende Ereignis – unabhängig davon, ob es sich um `S` oder `R` handelt.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung
S	Event	Setzt den Ausgang Q auf TRUE.
R	Event	Setzt den Ausgang Q auf FALSE.

Ereignis-Ausgänge

Keine (der Ausgangswert wird über einen Adapter bereitgestellt).

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine (der Ausgangswert wird über einen Adapter bereitgestellt).

Adapter

Name	Typ	Beschreibung
Q	`adapter::types::unidirectional::AX`	Bietet den aktuellen binären Zustand. Der Adapter stellt ein Ereignis (`E1`) und einen Datenwert (`D1`) bereit. Nach jeder Zustandsänderung wird

		Q.E1 ausgelöst und der neue Wert an Q.D1 ausgegeben.

Funktionsweise

Der Baustein basiert auf einem einfachen Zustandsautomaten mit drei Zuständen:

- **START**: Initialzustand nach dem Start. Es wird kein Ausgangswert ausgegeben.
- **SET**: Der Ausgang wird auf TRUE gesetzt. Bei Betreten dieses Zustands wird Q.E1 ausgelöst und Q.D1 auf TRUE gesetzt.
- **RESET**: Der Ausgang wird auf FALSE gesetzt. Bei Betreten dieses Zustands wird Q.E1 ausgelöst und Q.D1 auf FALSE gesetzt.

Die Transitionen erfolgen ausschließlich durch die Ereignisse S und R. Aus dem START-Zustand führt sowohl S nach SET als auch R nach RESET (symmetrisches Verhalten). Aus SET gelangt man nur mit R nach RESET, und aus RESET nur mit S zurück nach SET.

Technische Besonderheiten

- **Symmetrisches Start-up-Verhalten**: Der Baustein startet im Zustand START, ohne einen definierten Ausgangswert. Erst das erste eintreffende Ereignis (S oder R) bestimmt den Anfangszustand. Dies unterscheidet sich von Bausteinen, die standardmäßig auf FALSE oder TRUE initialisieren.
- **Adapter-Schnittstelle**: Der Zustand wird nicht direkt als Datenausgang geführt, sondern über einen unidirektionalen Adapter (AX). Änderungen werden durch ein Ereignis (Q.E1) signalisiert, sodass der empfangende Baustein asynchron reagieren kann.

Zustandsübersicht

Zustand	Beschreibung
START	Warte auf erstes Ereignis; kein Ausgangswert definiert.
SET	Ausgangswert TRUE; gesetzt durch Ereignis S.
RESET	Ausgangswert FALSE; zurückgesetzt durch Ereignis R.

Transitionen:

- START ⟶ SET bei Ereignis S
- START ⟶ RESET bei Ereignis R
- SET ⟶ RESET bei Ereignis R
- RESET ⟶ SET bei Ereignis S

Anwendungsszenarien

- **Speicherung eines binären Zustands** in Steuerungslogiken, bei denen der Anfangswert erst durch ein Ereignis definiert werden soll.
- **Ereignisgesteuerte Schalter** in Verbindung mit Adapter-basierten Kommunikationsmustern.
- **Ersetzen eines SR-Flipflops**, wenn ein symmetrisches Einschaltverhalten gefordert ist.

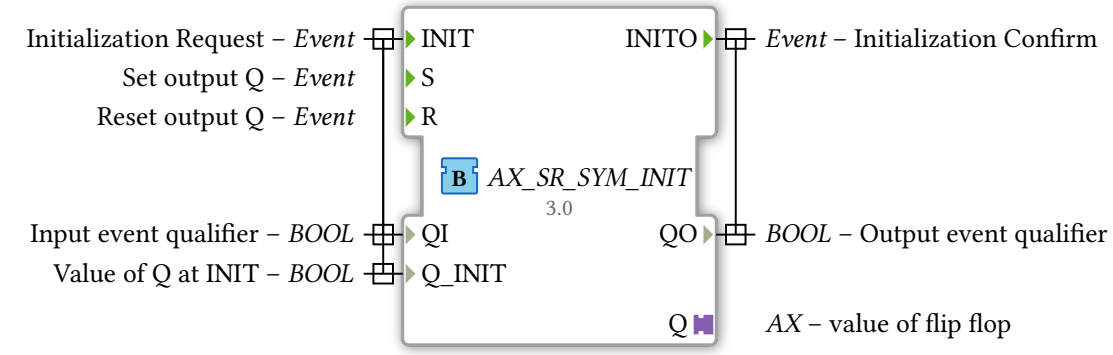
Vergleich mit ähnlichen Bausteinen

- **SR-Flipflop** (z. B. AX_SR): Besitzt oft einen definierten Startwert (z. B. FALSE) und verhält sich asymmetrisch, wenn beide Eingänge gleichzeitig aktiv sind. Der AX_SR_SYM legt dagegen den Anfangswert erst durch das erste Ereignis fest.
- **RS-Flipflop**: Grundsätzlich ähnliche Logik, aber mit umgekehrter Priorität (Reset dominiert). Der AX_SR_SYM hat keine Priorität – der zuletzt empfangene Eingang bestimmt den Zustand.
- **Adapterbasierte Bausteine**: Vorteil der ereignisgesteuerten Zustandsübergabe über Adapter; viele Standard-FB verwenden stattdessen direkte Datenausgänge.

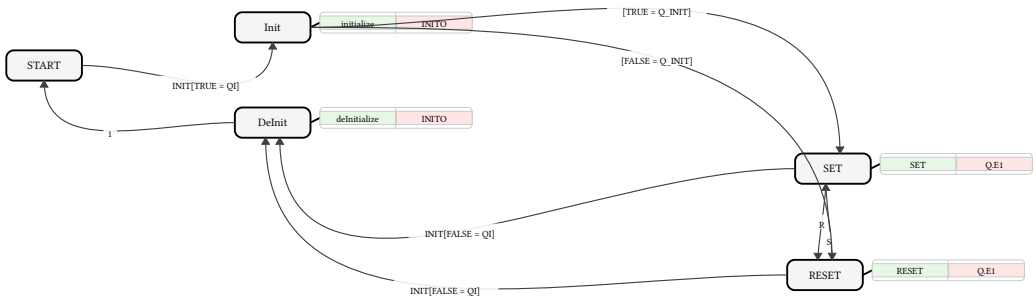
Fazit

Der AX_SR_SYM eignet sich ideal für Anwendungen, die ein bistabiles Speicherverhalten mit nicht vorbestimmtem Startwert erfordern. Die symmetrische Reaktion auf S und R im Startzustand sowie die Ausgabe über einen Adapter machen ihn zu einer flexiblen Komponente in ereignisgesteuerten Automatisierungssystemen.

AX_SR_SYM_INIT



ECC: AX_SR_SYM_INIT



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AX_SR_SYM_INIT** realisiert ein ereignisgesteuertes, bistabiles Flipflop (Set-Reset) mit symmetrischem Start-up-Verhalten und einer speziellen Initialisierungslogik. Er erweitert das klassische SR-Flipflop um einen qualifizierten Initialisierungs- und Deinitialisierungsablauf, bei dem der Ausgangszustand über den Eingang `Q_INIT` vorgegeben wird und die gesamte Logik nur dann aktiv ist, wenn der Qualifier `QI` den Wert `TRUE` besitzt.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung
INIT	EInit	Initialisierungsanforderung; mit `QI` und `Q_INIT` verknüpft
S	Event	Setzt den Ausgang `Q` (über Adapter)
R	Event	Setzt den Ausgang `Q` zurück

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung
INITO	EInit	Initialisierungsbestätigung; mit QO verknüpft

Daten-Eingänge

Variable	Typ	Beschreibung
QI	BOOL	Eingangsqualifier – steuert, ob Aktionen ausgeführt werden
Q_INIT	BOOL	Gewünschter Wert des Ausgangs Q nach einer Initialisierung

Daten-Ausgänge

Variable	Typ	Beschreibung
QO	BOOL	Ausgangsqualifier – spiegelt den Wert von QI zum Zeitpunkt des auslösenden Ereignisses

Adapter

Adapter	Typ	Beschreibung
Q	adapter::types::unidirectional::AX	Wert des Flipflops – wird über die Schnittstelle gesetzt/ zurückgesetzt

Funktionsweise

Der Baustein besitzt vier Hauptzustände: START, Init, DeInit, SET und RESET.

- **START** – Wartezustand nach dem Systemstart.
- **Init** – Wird aktiviert, wenn auf INIT eintrifft **und** QI = TRUE. In diesem Zustand wird der Algorithmus initialize ausgeführt, der QO := QI setzt. Anschließend verzweigt der Zustand je nach Q_INIT:
  - Ist Q_INIT = TRUE, wird in den SET-Zustand gewechselt (Flipflop wird gesetzt).
  - Ist Q_INIT = FALSE, wird in den RESET-Zustand gewechselt (Flipflop wird zurückgesetzt).
- **DeInit** – Wird aktiviert, wenn auf INIT eintrifft **und** QI = FALSE (aus den Zuständen SET oder RESET). Der Algorithmus deInitialize setzt QO := FALSE. Danach springt der Zustand zurück zu START.
- **SET** – Wird durch das Ereignis S erreicht (aus RESET oder nach Initialisierung mit Q_INIT=TRUE). Der Algorithmus SET setzt QO := QI und schreibt **nur wenn QI = TRUE** den Wert TRUE auf den Adapterausgang Q.D1. Ein erneutes S bleibt wirkungslos, solange der Zustand SET aktiv ist.
- **RESET** – Wird durch das Ereignis R erreicht (aus SET oder nach Initialisierung mit Q_INIT=FALSE). Der Algorithmus RESET setzt QO := QI und schreibt **nur wenn QI = TRUE** den Wert FALSE auf Q.D1. Ein erneutes R bleibt wirkungslos.

Die Transitionen zwischen SET und RESET werden nur durch die Ereignisse S bzw. R ausgelöst. Die INIT-Transitionen können aus beiden stabilen Zuständen in DeInit führen, wenn QI = FALSE ist.

Technische Besonderheiten

- **Qualifier-abhängige Aktionen**: Die eigentliche Änderung des Flipflop-Ausgangs (Q.D1) erfolgt nur, wenn QI = TRUE. Andernfalls werden nur die Output-Qualifier (QO) aktualisiert, der Adapter bleibt unverändert.
- **Symmetrisches Start-up**: Der Baustein kann sowohl mit einem „gesetzten“ als auch mit einem „zurückgesetzten“ Zustand initialisiert werden – abhängig vom Wert Q_INIT.
- **Deinitialisierung**: Wenn während der Initialisierung QI = FALSE ist, wird der Baustein in den DeInit-Zustand versetzt, der den Ausgangsqualifier löscht und dann in den Startzustand zurückkehrt.
- **Guard-Bedingungen**: Die Zustandsübergänge nutzen Bedingungen wie INIT[TRUE = QI] oder [FALSE = Q_INIT], um die Logik präzise zu steuern.

Zustandsübersicht

- **START** ⟶ Ruhezustand nach Start.
- **Init** ⟶ Initialisierungslauf (nur bei QI=TRUE).

- **DeInit** ⟶ Deinitialisierungslauf (nur bei `QI=FALSE`).
- **SET** ⟶ Stabiler Zustand, in dem `Q.D1 = TRUE` (sofern zuvor gesetzt).
- **RESET** ⟶ Stabiler Zustand, in dem `Q.D1 = FALSE` (sofern zuvor zurückgesetzt).

Übergänge:

- `START` ⟶ `Init` : wenn `INIT` & `QI=TRUE`
- `Init` ⟶ `SET` : wenn `Q_INIT=TRUE`
- `Init` ⟶ `RESET` : wenn `Q_INIT=FALSE`
- `SET` ⟶ `RESET` : wenn `R`
- `RESET` ⟶ `SET` : wenn `S`
- `SET` ⟶ `DeInit` : wenn `INIT` & `QI=FALSE`
- `RESET` ⟶ `DeInit` : wenn `INIT` & `QI=FALSE`
- `DeInit` ⟶ `START` : automatisch nach Abschluss des DeInit-Algorithmus

Anwendungsszenarien

- **Steuerung von Aktoren mit definiertem Startverhalten**: Ein Motor oder Ventil soll nach dem Einschalten einen bestimmten Zustand einnehmen (z. B. geschlossen = `Q=FALSE`), abhängig von einer Initialisierungsvorgabe.
- **Qualifizierte Zustandsänderungen**: In Systemen, in denen ein Freigabesignal (`QI`) erst die eigentliche Wertänderung erlaubt, während der Qualifier-Ausgang (`QO`) die Gültigkeit signalisiert.
- **Reset-fähige Initialisierungsroutine**: Eine Komponente kann durch einen INIT-Aufruf entweder gesetzt oder zurückgesetzt werden, kann aber später durch erneuten INIT-Aufruf mit `QI=FALSE` in einen definierten Ruhezustand versetzt werden.

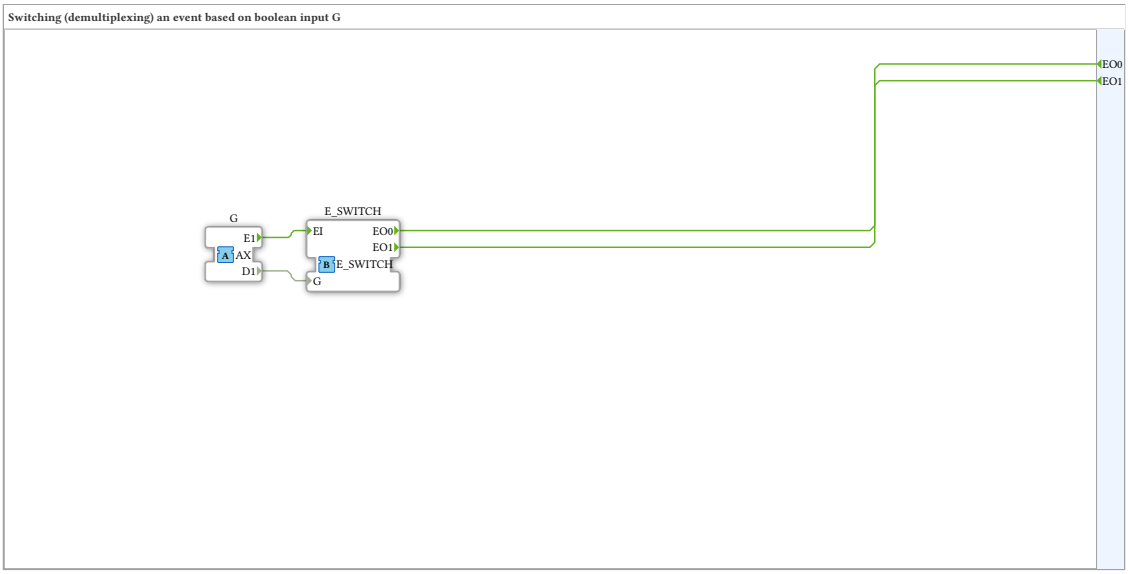
Vergleich mit ähnlichen Bausteinen

- **Standard-SR-Flipflop (z. B. `SR` oder `AX_SR`)**: Einfaches Set/Reset ohne Initialisierung und ohne Qualifier. `AX_SR_SYM_INIT` bietet zusätzlich einen qualifizierten Initialisierungs- und Deinitialisierungsablauf.
- **Bausteine mit INIT-Schnittstelle (z. B. `E_SR` mit `INIT`)**: Diese haben oft keine ausgeprägte Deinitialisierung oder keinen Qualifier. Der hier vorgestellte FB unterscheidet sich durch das symmetrische Verhalten – sowohl Setzen als auch Zurücksetzen kann im INIT-Schritt vorgegeben werden.
- **Bausteine mit Adapter (`unidirectional::AX`)**: Werden verwendet, um den Zustand über eine standardisierte Adapterschnittstelle nach außen zu tragen. Der `AX_SR_SYM_INIT` kapselt diese Logik in einem FB.

Fazit

Der `AX_SR_SYM_INIT` ist ein vielseitiger Funktionsblock für Anwendungen, die ein qualifiziertes Start- und Initialisierungsverhalten erfordern. Durch die Kombination von Ereignissteuerung, Qualifier-Logik und Adapter-Schnittstelle eignet er sich besonders für modulare Automatisierungslösungen nach IEC 61499, bei denen ein definierter Systemstart und eine saubere Deinitialisierung gefordert sind.

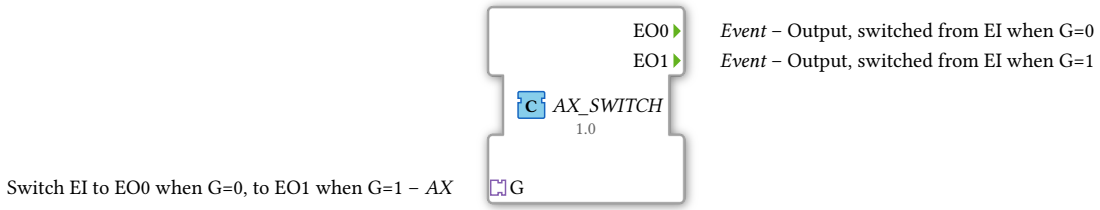
AX_SWITCH



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_SWITCH Funktionsblock dient als Schaltbaustein (Demultiplexer) für Ereignisse basierend auf einem booleschen Eingangswert. Er leitet eingehende Ereignisse je nach Zustand des Steuersignals auf einen von zwei möglichen Ausgängen weiter.



Schnittstellenstruktur

Ereignis-Eingänge

- Keine direkten Ereigniseingänge - Ereigniseingang erfolgt über Adapter

Ereignis-Ausgänge

- **EO0**: Ausgangsereignis, das aktiviert wird, wenn ein Ereignis bei G=0 eintrifft
- **EO1**: Ausgangsereignis, das aktiviert wird, wenn ein Ereignis bei G=1 eintrifft

Daten-Eingänge

- Keine direkten Dateneingänge - Daten werden über Adapter bereitgestellt

Daten-Ausgänge

- Keine Datenausgänge vorhanden

Adapter

- **G**: Adapter vom Typ `adapter::types::unidirectional::AX`
  - Schaltet eingehende Ereignisse von EI auf EO0, wenn G=0
  - Schaltet eingehende Ereignisse von EI auf EO1, wenn G=1

Funktionsweise

Der AX_SWITCH Block empfängt Ereignisse und Steuerdaten über den Adapter G. Basierend auf dem booleschen Wert des Steuersignals G wird jedes eingehende Ereignis entweder an Ausgang EO0 (bei G=0)

oder an Ausgang EO1 (bei G=1) weitergeleitet. Intern nutzt der Block einen E_SWITCH Baustein zur Realisierung der Schaltfunktion.

Technische Besonderheiten

- Implementiert als Wrapper um den Standard-E_SWITCH Baustein
- Verwendet unidirektionale Adapter für Ereignis- und Datenkommunikation
- Keine direkten Schnittstellen, alle Kommunikation erfolgt über Adapter

Zustandsübersicht

Der Block besitzt keine internen Zustände im klassischen Sinne. Er arbeitet rein ereignisgesteuert und leitet jedes eingehende Ereignis sofort basierend auf dem aktuellen Wert von G an den entsprechenden Ausgang weiter.

Anwendungsszenarien

- Steuerung von alternativen Prozesspfaden
- Umschaltung zwischen Betriebsmodi
- Verteilerfunktionen in ereignisbasierten Steuerungen
- Demultiplexing von Ereignisströmen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zum Standard-E_SWITCH Block bietet AX_SWITCH eine Adapter-basierte Schnittstelle, was eine bessere Integration in Adapter-basierte Architekturen ermöglicht. Während E_SWITCH direkte Ein- und Ausgänge besitzt, kommuniziert AX_SWITCH ausschließlich über Adapter.

Vergleich mit E_SWITCH

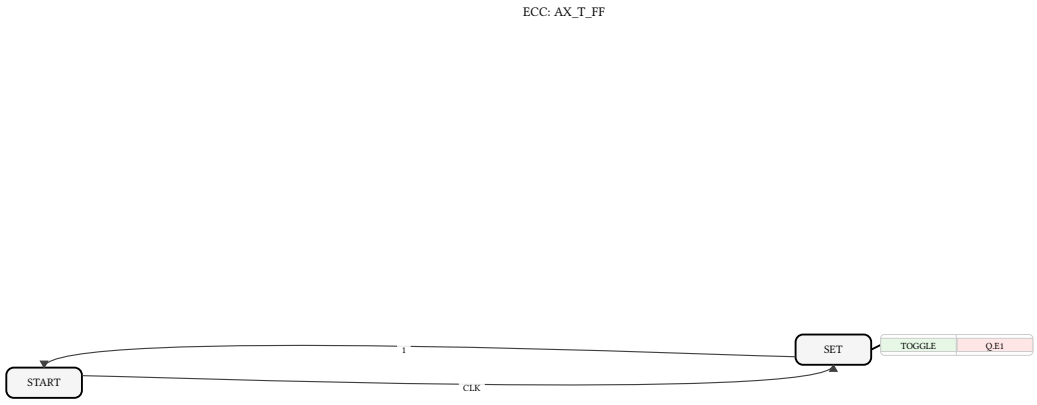
🛠️ Zugehörige Übungen

- Uebung_004b_AX
- Uebung_004b_AX_ASR
- Uebung_004b_AX_ASR_X
- Uebung_005_AX
- Uebung_006a3_sub_AX
- Uebung_007a3_AX
- Uebung_008_AX
- Uebung_009_AX
- Uebung_020a_AX
- Uebung_020b_AX
- Uebung_020c3_AX
- Uebung_020d_AX
- Uebung_020e2_AX
- Uebung_020f2_AX
- Uebung_020i_AX

Fazit

AX_SWITCH ist ein spezialisierter Schaltbaustein für ereignisbasierte Systeme, der durch seine Adapter-basierte Architektur eine flexible Integration in komplexere Steuerungssysteme ermöglicht. Die klare Trennung von Ereignis- und Datenfluss über Adapter sorgt für eine übersichtliche Systemstruktur.

AX_T_FF

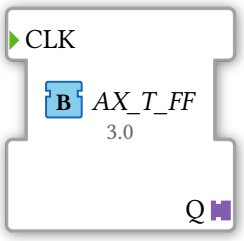


• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_T_FF (Toggle Flip-Flop) ist ein grundlegender Speicherbaustein in 4diac, der als Toggle-Flipflop fungiert. Bei jedem Taktereignis wechselt der Ausgabewert zwischen den beiden möglichen Zuständen. Der Baustein implementiert ein einfaches Schaltverhalten, bei dem der Ausgangswert bei jedem Taktimpuls umgeschaltet wird.

Clock for triggering a output toggle – *Event*



AX – value of flip flop

Schnittstellenstruktur

Ereignis-Eingänge

- **CLK**: Takteingang, der eine Ausgangsumschaltung auslöst

Ereignis-Ausgänge

- Keine direkten Ereignisausgänge vorhanden

Daten-Eingänge

- Keine Dateneingänge vorhanden

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden

Adapter

- **Q**: Unidirektionaler Adapter vom Typ AX, der den aktuellen Flipflop-Wert bereitstellt

Funktionsweise

Der AX_T_FF arbeitet als einfacher Toggle-Flipflop. Bei jedem eingehenden CLK-Ereignis wird der Algorithmus TOGGLE ausgeführt, der den aktuellen Wert des Adapters Q.D1 invertiert. Der Baustein beginnt im START-Zustand und wechselt bei jedem CLK-Ereignis in den SET-Zustand, wo die Umschaltung durchgeführt wird.

Technische Besonderheiten

- Verwendet einen unidirektionalen Adapter für die Ausgabe
- Implementiert als Basic Function Block (BFB)
- Besitzt einen einfachen Zustandsautomaten mit zwei Zuständen
- Der Algorithmus TOGGLE führt eine logische Negation des Ausgabewerts durch

Zustandsübersicht

Der Baustein verfügt über zwei Zustände:

1. **START**: Initialzustand, wartet auf CLK-Ereignis
2. **SET**: Aktiver Zustand, in dem der TOGGLE-Algorithmus ausgeführt wird

Zustandsübergänge:

- START ⟶ SET: Bei CLK-Ereignis
- SET ⟶ START: Immer (Condition „1“ = wahr)

Anwendungsszenarien

- Frequenzteilung von Taktsignalen
- Erzeugung von Rechtecksignalen mit halber Eingangsfrequenz
- Zähler- und Teiler-Schaltungen
- Zustandssteuerungen mit alternierendem Verhalten

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Flipflop-Typen wie RS- oder D-Flipflops bietet der Toggle-Flipflop eine spezialisierte Funktionalität für reine Umschaltoperationen. Er ist einfacher aufgebaut als universellere Flipflop-Typen und benötigt keine zusätzlichen Dateneingänge.

Vergleich mit E_T_FF

🛠️ Zugehörige Übungen

- Uebung_004a2_AX
- Uebung_004a3_AX
- Uebung_004a4_AX
- Uebung_004a5_AX
- Uebung_004a6_AX
- Uebung_004a8_AX
- Uebung_004a9_AX
- Uebung_004a_AX
- Uebung_004c1_AX
- Uebung_004c2_AX
- Uebung_004c3_AX
- Uebung_004c4_AX
- Uebung_004c5_AX
- Uebung_004c6_AX
- Uebung_004c7_AX
- Uebung_005_AX
- Uebung_006a3_sub_AX
- Uebung_007_AX
- Uebung_007a1_AX
- Uebung_007a2_AX
- Uebung_010b2_AX
- Uebung_010b3_AX
- Uebung_010b6_AX
- Uebung_010b7_AX

- Uebung_010b8_AX
- Uebung_010b9_AX
- Uebung_010bA2_AX
- Uebung_010bA3_AX
- Uebung_010bA4_AX
- Uebung_010bA_AX
- Uebung_035a2_AX
- Uebung_094a_AX
- Uebung_095_AX
- Uebung_150_AX
- Uebung_151_AX

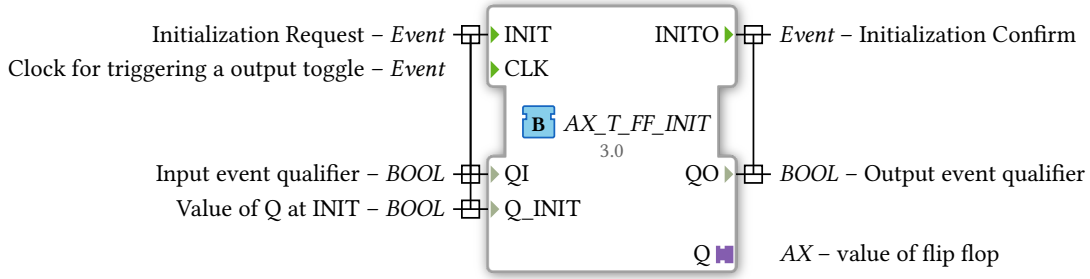
Fazit

Der AX_T_FF ist ein spezialisierter und effizienter Baustein für Anwendungen, die ein reines Toggle-Verhalten benötigen. Seine einfache Struktur und klare Funktionsweise machen ihn zu einer zuverlässigen Komponente für Frequenzteilung und Zustandsalternierung in Steuerungssystemen.

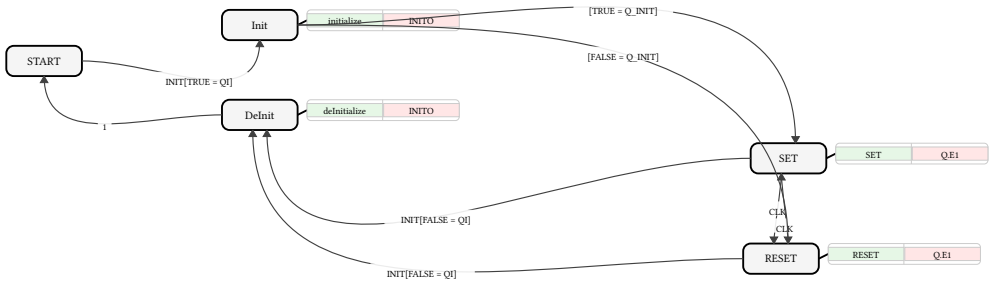
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AX_T_FF_INIT



ECC: AX_T_FF_INIT



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AX_T_FF_INIT** realisiert ein getriggertes Toggle-Flipflop mit Initialisierungsfunktion. Er kann über ein INIT-Ereignis in einen definierten Startzustand versetzt werden und wechselt anschließend bei jedem CLK-Ereignis zwischen den Zuständen SET und RESET. Der aktuelle Zustand wird über einen Adapterausgang bereitgestellt. Der Baustein ist besonders geeignet für Anwendungen, bei denen ein Ausgangssignal bei jedem Takt umgeschaltet werden soll, wobei der Anfangszustand über eine Initialisierung festgelegt wird.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Initialisierungsanforderung; triggert die Initialisierung des Flipflops. Die zugehörigen Daten-Eingänge QI und Q_INIT werden ausgewertet.
CLK	Event	Takteingang; bei jedem Ereignis wechselt der interne Zustand zwischen SET und RESET.

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Initialisierungsbestätigung; wird nach erfolgreicher Initialisierung oder Deinitialisierung ausgegeben.

Daten-Eingänge

Name	Typ	Kommentar
QI	BOOL	Eingangsqualifizierer; steuert, ob Aktionen tatsächlich ausgeführt werden (TRUE = aktiv).
Q_INIT	BOOL	Wert, der dem Flipflop bei der Initialisierung zugewiesen wird (TRUE = SET, FALSE = RESET).

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Ausgangsqualifizierer; spiegelt den Wert von QI wider, wenn eine Aktion ausgeführt wurde, sonst FALSE.

Adapter

Name	Typ	Kommentar
Q	Adapter `adapter::types::unidirectional::AX`	Adapterausgang, der den aktuellen Flipflop-Zustand (TRUE oder FALSE) an verbundene Bausteine weitergibt.

Funktionsweise

Der FB besitzt folgende interne Zustände: START, Init, DeInit, SET und RESET.

1. **Initialisierung (INIT-Ereignis)**
   - Bei einem INIT-Ereignis mit **QI** = **TRUE** wird vom Startzustand in den Zustand **Init** gewechselt.
   - Im Zustand Init wird der Algorithmus *initialize* ausgeführt: `QO := QI`.
   - Anschließend wird der Ausgangsqualifizierer QO auf TRUE gesetzt und das Ereignis **INITO** ausgegeben.
   - Abhängig vom Wert von **Q_INIT** erfolgt der Übergang:
     – Ist Q_INIT = TRUE ⟶ Wechsel in den Zustand **SET**

   - Ist Q_INIT = FALSE ⟶ Wechsel in den Zustand **RESET**

2. **Toggle-Betrieb (CLK-Ereignis)**
   - Ausgehend von **SET** oder **RESET** wechselt bei jedem **CLK**-Ereignis der Zustand:
     - Von SET ⟶ RESET (und umgekehrt)
   - Beim Verlassen der Zustände werden die entsprechenden Algorithmen ausgeführt:
     - **SET**: `QO := QI;` Wenn QI = TRUE, wird dem Adapterausgang **Q.D1** = **TRUE** zugewiesen.
     - **RESET**: `QO := QI;` Wenn QI = TRUE, wird dem Adapterausgang **Q.D1** = **FALSE** zugewiesen.
     - In beiden Fällen wird das Ereignis **Q.E1** ausgegeben (über den Adapter).

3. **Deinitialisierung (INIT-Ereignis mit QI = FALSE)**
   - Wenn während des Betriebs (in SET oder RESET) ein INIT-Ereignis mit **QI** = **FALSE** eintrifft, wird in den Zustand **DeInit** gewechselt.
   - Der Algorithmus *deInitialize* setzt `QO := FALSE`.
   - Es wird **INITO** ausgegeben, und der FB kehrt in den Startzustand zurück.

Der Baustein bleibt im Startzustand, bis ein INIT mit QI=TRUE eintrifft. Ohne Initialisierung ist kein Toggle-Betrieb möglich.

Technische Besonderheiten

- **Adapterbasiertes Ausgangssignal**: Der aktuelle Flipflop-Zustand wird nicht als einfacher Datenausgang, sondern über einen **Adapter** (`adapter::types::unidirectional::AX`) bereitgestellt. Dies ermöglicht eine flexible und typisierte Verbindung mit anderen Bausteinen, die denselben Adapter verwenden.
- **Zustandsabhängige Ausführung**: Die Aktionen (Setzen/Rücksetzen des Adapterausgangs) werden nur dann ausgeführt, wenn der Qualifizierer **QI** = TRUE ist. Ist QI = FALSE, wird der Adapterausgang nicht verändert, der Zustand wechselt jedoch dennoch.
- **Initialisierung mit Startwert**: Über den Eingang **Q_INIT** kann der Anfangszustand nach der Initialisierung gezielt festgelegt werden (TRUE = gesetzt, FALSE = zurückgesetzt).

Zustandsübersicht

Zustand	Bedeutung
START	Warte auf erstes INIT-Ereignis.
Init	Initialisierung läuft; QO wird auf QI gesetzt, dann Übergang zu SET oder RESET abhängig von Q_INIT.
DeInit	Deinitialisierung läuft; QO wird auf FALSE gesetzt, Rückkehr zu START.
SET	Flipflop ist im gesetzten Zustand; Adapterausgang Q.D1 = TRUE (bei QI=TRUE).
RESET	Flipflop ist im zurückgesetzten Zustand; Adapterausgang Q.D1 = FALSE (bei QI=TRUE).

Die Zustandsübergänge werden durch die Ereignisse INIT (mit entsprechender Bedingung an QI) und CLK gesteuert.

Anwendungsszenarien

- **Steuerung von Ausgängen mit Toggle-Funktion**: Ein digitaler Ausgang soll bei jedem Takt zwischen Ein und Aus wechseln, z. B. für Blinksignale oder Impulsfolgen.
- **Initialisierbare Zustandsmaschinen**: In Automatisierungssystemen, in denen ein Prozess nach einem Reset einen definierten Startzustand einnehmen muss (gesetzt oder zurückgesetzt).
- **Adapter-basierte Kommunikation**: Bausteine, die den gleichen Adapter verwenden, können den Flipflop-Zustand direkt auslesen, ohne auf separate Data-Outputs angewiesen zu sein.

Vergleich mit ähnlichen Bausteinen

Kriterium	AX_T_FF_INIT	Einfaches Toggle-FF (ohne INIT)	SR-Flipflop
Initialisierung	Ja, über INIT mit Q_INIT	Nein, Zustand nach Start undefiniert	Nein (set/reset über separate Eingänge)

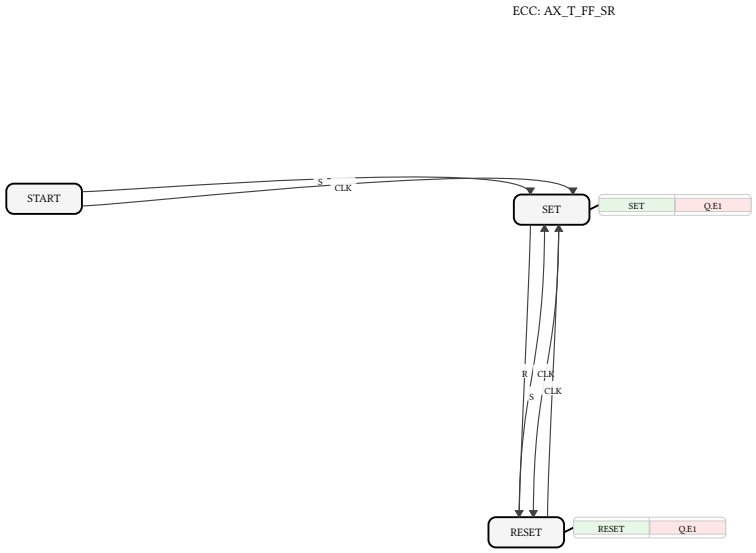
Zustandswechsel	Gettaktet durch CLK	Gettaktet durch CLK	Asynchron über S und R
Ausgang	Über Adapter (typisiert)	Oft als BOOL-Datenausgang	BOOL-Datenausgang
Qualifizierer QI	Steuert, ob Aktion ausgeführt wird	Nicht vorhanden	Nicht vorhanden

Der **AX_T_FF_INIT** bietet im Vergleich zu einem einfachen Toggle-Flipflop die Möglichkeit einer definierten Initialisierung und verwendet einen Adapter für den Ausgang, was in komplexen Systemen die Konfiguration vereinfacht.

Fazit

Der Funktionsblock **AX_T_FF_INIT** ist ein vielseitiger Baustein für getaktete Toggle-Operationen mit Initialisierungslogik. Seine Kombination aus INIT- und CLK-Ereignissen, dem Qualifizierer QI und dem Adapterausgang erlaubt eine präzise Steuerung des Zustandsverhaltens. Er eignet sich besonders für industrielle Automatisierungsanwendungen, in denen ein definierter Startwert und ein getaktetes Umschalten erforderlich sind.

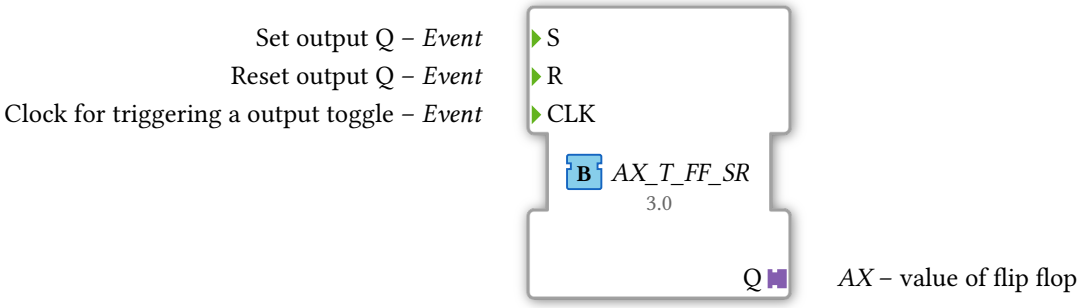
AX_T_FF_SR



• – ○ • – ○ • – ○ •

Einleitung

Der AX_T_FF_SR ist ein ereignisgesteuerter bistabiler Funktionsblock mit Toggle-Funktionalität. Es handelt sich um ein Flip-Flop-Element, das sowohl als Set-Reset-Flipflop als auch als Toggle-Flipflop betrieben werden kann. Der Baustein kombiniert die Eigenschaften eines SR-Flipflops mit zusätzlicher Toggle-Funktionalität über einen Takteingang.



Schnittstellenstruktur

Ereignis-Eingänge

- **S**: Setzt den Ausgang Q auf TRUE
- **R**: Setzt den Ausgang Q auf FALSE (Reset)
- **CLK**: Takt für das Umschalten (Toggle) des Ausgangs

Ereignis-Ausgänge

- Keine direkten Ereignisausgänge vorhanden

Daten-Eingänge

- Keine Dateneingänge vorhanden

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden

Adapter

- **Q**: Unidirektionaler Adapter vom Typ AX, der den Wert des Flipflops bereitstellt

Funktionsweise

Der AX_T_FF_SR verfügt über drei Betriebszustände:

- **START**: Initialzustand
- **SET**: Ausgang Q ist TRUE
- **RESET**: Ausgang Q ist FALSE

Die Zustandsübergänge werden durch die Ereigniseingänge gesteuert:

- S-Ereignis führt von jedem Zustand in den SET-Zustand
- R-Ereignis führt von jedem Zustand in den RESET-Zustand
- CLK-Ereignis toggelt den aktuellen Zustand (SET ⟶ RESET oder RESET ⟶ SET)

Bei jedem Zustandswechsel wird der entsprechende Algorithmus ausgeführt, der den Adapterwert Q.D1 entsprechend setzt.

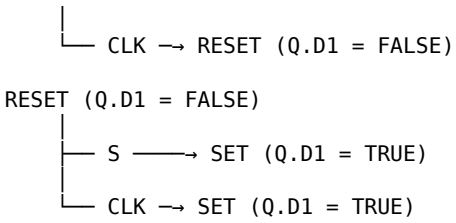
Technische Besonderheiten

- Kombiniert SR-Flipflop- und T-Flipflop-Funktionalität
- Verwendet Adapter-Schnittstelle für die Datenausgabe
- Unidirektionale Kommunikation über den Q-Adapter
- Initialzustand ist START, von wo aus direkt in SET oder durch CLK in SET gewechselt werden kann

Zustandsübersicht

```
START (Initialzustand)
    │
    ├── S ──→ SET (Q.D1 = TRUE)
    │
    └── CLK ─→ SET (Q.D1 = TRUE)

SET (Q.D1 = TRUE)
    │
    ├── R ───→ RESET (Q.D1 = FALSE)
```



Anwendungsszenarien

- Zustandsspeicherung in Steuerungsanwendungen
- Taktteilung und Frequenzteilung
- Ereigniszählung
- Zustandsautomaten mit Speicherfunktion
- Schaltnetze mit Rückkopplung

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem einfachen E_SR-Flipflop bietet der AX_T_FF_SR zusätzliche Toggle-Funktionalität durch den CLK-Eingang. Während ein reines SR-Flipflop nur über Set- und Reset-Eingänge verfügt, ermöglicht dieser Baustein zusätzlich das taktgesteuerte Umschalten des Ausgangszustands.

Vergleich mit E_T_FF_SR

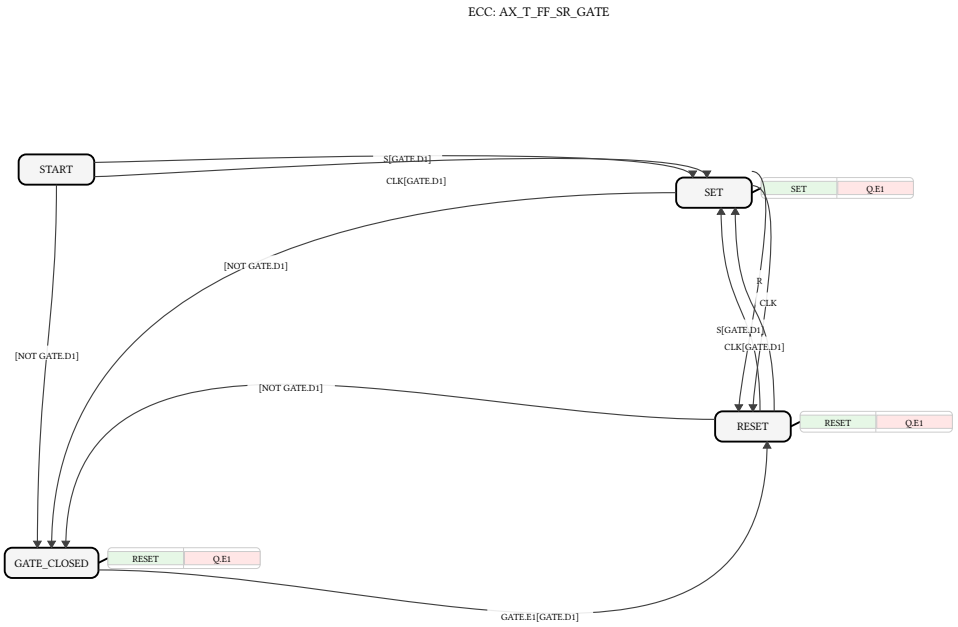
🛠️ Zugehörige Übungen

- Uebung_004a7_AX
- Uebung_006a2_AX
- Uebung_006a3_AX
- Uebung_006a4_AX
- Uebung_006a_AX

Fazit

Der AX_T_FF_SR ist ein vielseitiger bistabiler Speicherbaustein, der die Vorteile von SR- und T-Flipflops kombiniert. Durch die Adapter-basierte Schnittstelle ermöglicht er eine flexible Integration in größere Steuerungssysteme und eignet sich besonders für Anwendungen, die sowohl direkte Zustandssetzung als auch taktgesteuertes Umschalten erfordern.

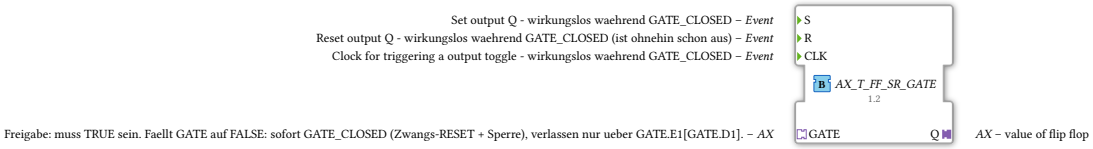
AX_T_FF_SR_GATE



• — ○ • — ○ • — ○ •

Einleitung

`AX_T_FF_SR_GATE` entspricht `AX_T_FF_SR`, hat aber zusätzlich einen Freigabe-Socket `GATE` (`AX`) und einen eigenen Zustand `GATE_CLOSED`, ohne Verkettung (kein `CHAIN_IN`/`CHAIN_OUT`). Er eignet sich für Fälle, in denen nur die Gate-Sperre gebraucht wird, aber keine Kettenverzögerung; mit Verkettung siehe `AX_T_FF_SR_SEQ_GATE`.



Schnittstellenstruktur

Ereignis-Eingänge

- **S**: Setzt den Ausgang `Q` – wirkungslos während `GATE_CLOSED`
- **R**: Setzt den Ausgang `Q` zurück – wirkungslos während `GATE_CLOSED` (ist dort ohnehin aus)
- **CLK**: Takt zum Umschalten (Toggle) des Ausgangs – wirkungslos während `GATE_CLOSED`

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge – das Ereignis wird über `Q.E1` ausgegeben.

Daten-Eingänge / Daten-Ausgänge

Keine.

Adapter

- **Q** (Plug, `adapter::types::unidirectional::AX`): Wert des Flipflops
- **GATE** (Socket, `adapter::types::unidirectional::AX`): Freigabe, muss `TRUE` sein. Fällt `GATE` auf `FALSE`, geht der Baustein sofort in `GATE_CLOSED` (Zwangs-Reset und Sperre); verlassen wird dieser Zustand nur über `GATE.E1[GATE.D1]`

Funktionsweise

Zustände und Aktionen:

- **START**: Initialzustand
- **SET**: Algorithmus `SET` (`Q.D1 := TRUE`), Ausgabe `Q.E1`
- **RESET**: Algorithmus `RESET` (`Q.D1 := FALSE`), Ausgabe `Q.E1`
- **GATE_CLOSED**: gesperrt (`GATE.D1 = FALSE`), `Q` zwangsweise `FALSE` (Algorithmus `RESET`, Ausgabe `Q.E1`); `S`/`R`/`CLK` sind hier wirkungslos

Transitionen (in Deklarationsreihenfolge):

Von	Nach	Bedingung
START	GATE_CLOSED	`[NOT GATE.D1]`
START	SET	`S[GATE.D1]`
START	SET	`CLK[GATE.D1]`
SET	GATE_CLOSED	`[NOT GATE.D1]`
SET	RESET	`R`
SET	RESET	`CLK`
RESET	GATE_CLOSED	`[NOT GATE.D1]`
RESET	SET	`S[GATE.D1]`
RESET	SET	`CLK[GATE.D1]`
GATE_CLOSED	RESET	`GATE.E1[GATE.D1]`

Technische Besonderheiten

- **Sperre zuerst**: `[NOT GATE.D1]` steht in jedem Zustand an erster Stelle der Deklarationsreihenfolge. Da `S`/`CLK` ohne Guard immer `TRUE` sind, würde sonst nach IEC 61499-1 Tabelle 1 Regel c) die Gate-Sperre nie zum Zug kommen (Bugfix v1.2). Ist das Gate offen, ist der Guard `FALSE` und wird übersprungen
- **Boot-Fenster**: Die ereignislose Transition `START` → `GATE_CLOSED` wird sofort beim Betreten von `START` ausgewertet und deckt den Zustand ab, bevor je ein `GATE.E1` gefeuert hat (v1.1)
- Aus `GATE_CLOSED` geht es nur nach `RESET` zurück, `Q` bleibt also `FALSE`, bis `S` oder `CLK` wirkt
- Kopie von `AX_T_FF_SR` mit `GATE`-Socket und Zustand `GATE_CLOSED`, identisches Muster wie `AX_T_FF_SR_SEQ_GATE`

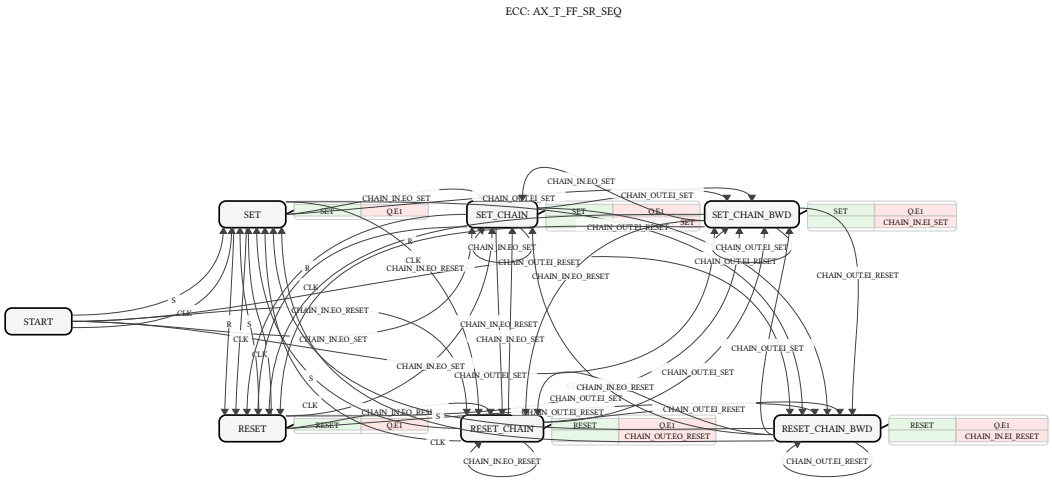
Anwendungsszenarien

- Toggle-/Set-Reset-Ausgang, der durch eine Freigabe zwangsweise gesperrt werden muss (z. B. Sicherheits- oder Betriebsfreigabe)

Fazit

`AX_T_FF_SR_GATE` ist ein Toggle-/SR-Flipflop mit Freigabe: Ist `GATE` `FALSE`, wird `Q` zurückgesetzt und alle Eingänge bleiben bis zur Freigabe wirkungslos.

AX_T_FF_SR_SEQ



• – ○ • – ○ • – ○ •

Einleitung

`AX_T_FF_SR_SEQ` ist ein ereignisgesteuertes bistabiles Flipflop mit Toggle-Funktion (Kopie von `AX_T_FF_SR`), das zusätzlich **echt bidirektional kettenfähig** über den Adapter `ASR2` (`CHAIN_IN`/`CHAIN_OUT`) ist. Damit lassen sich mehrere Bänder gestaffelt ein- und ausschalten (z. B. zusammen mit `ASR2_DELAY`). Die lokalen Ereignisse `S`/`R`/`CLK` verhalten sich exakt wie bei `AX_T_FF_SR` und lösen **nie** `CHAIN_IN`/`CHAIN_OUT` aus.

Schnittstellenstruktur

Ereignis-Eingänge

- **S**: Setzt den Ausgang `Q` (lokal, wirkt nie auf `CHAIN_IN`/`CHAIN_OUT`)
- **R**: Setzt den Ausgang `Q` zurück (lokal, wirkt nie auf `CHAIN_IN`/`CHAIN_OUT`)
- **CLK**: Takt zum Umschalten (Toggle) des Ausgangs (lokal, wirkt nie auf `CHAIN_IN`/`CHAIN_OUT`)

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden – Ereignisse laufen über die Adapter-Plugs

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden – der Zustand wird über den Adapter `Q` ausgegeben

Adapter

- **Q** (Plug): Unidirektionaler Adapter vom Typ `adapter::types::unidirectional::AX` – Wert des Flipflops
- **CHAIN_IN** (Socket): Bidirektionaler Adapter vom Typ `adapter::types::bidirectional::ASR2` – vorwärts: Set/Reset vom vorherigen Kettenglied empfangen (`EO_SET`/`EO_RESET`); rückwärts: Set/Reset an das vorherige Kettenglied weiterreichen (`EI_SET`/`EI_RESET`)

- **CHAIN_OUT** (Plug): Bidirektionaler Adapter vom Typ `adapter::types::bidirectional::ASR2` – vorwärts: Set/Reset an das nächste Kettenglied weiterreichen (`EO_SET`/`EO_RESET`); rückwärts: Set/Reset vom nächsten Kettenglied empfangen (`EI_SET`/`EI_RESET`)

Funktionsweise

Der Baustein besitzt sieben Zustände (`START`, `SET`, `RESET`, `SET_CHAIN`, `RESET_CHAIN`, `SET_CHAIN_BWD`, `RESET_CHAIN_BWD`). Der Ausgang `Q` wird bei jedem Eintritt in einen Setz-Zustand per Algorithmus `SET` auf `TRUE` und in einem Rücksetz-Zustand per `RESET` auf `FALSE` gesetzt und über `Q.E1` ausgegeben.

Es gibt **vier strikt getrennte, nicht verzweigende Kettenpfade**:

1. `CHAIN_IN.EO_SET` ⟶ lokal `SET` + `CHAIN_OUT.EO_SET` (Vorwärts-Set)
2. `CHAIN_IN.EO_RESET` ⟶ lokal `RESET` + `CHAIN_OUT.EO_RESET` (Vorwärts-Reset)
3. `CHAIN_OUT.EI_SET` ⟶ lokal `SET` + `CHAIN_IN.EI_SET` (Rückwärts-Set)
4. `CHAIN_OUT.EI_RESET` ⟶ lokal `RESET` + `CHAIN_IN.EI_RESET` (Rückwärts-Reset)

Ein über Pfad 1 ankommendes Set löst niemals zusätzlich Pfad 3 oder 4 aus und umgekehrt – jeder Zustand feuert genau eine Chain-Aktion. Lokal gilt:

- `S` bzw. `CLK` in `START` oder `RESET`/`RESET_CHAIN`/`RESET_CHAIN_BWD` ⟶ `SET`
- `R` bzw. `CLK` in `SET`/`SET_CHAIN`/`SET_CHAIN_BWD` ⟶ `RESET`
- Die vier Chain-Ereignisse sind aus jedem Zustand heraus möglich und führen in den jeweiligen Chain-Zustand.

Technische Besonderheiten

- Lokales `S`/`R`/`CLK` verändert nur `Q`, die Kette wird nicht angestoßen.
- Mit den Rückwärtspfaden (Version 1.1) lassen sich dieselben physischen Kettenglieder vorwärts (z. B. Set, Downstream zuerst) und rückwärts (z. B. Reset, Upstream zuerst) gestaffelt schalten, ohne dass ein Vorwärts-Ereignis auch rückwärts weiterläuft oder umgekehrt.
- Ein `R` im Zustand `START` bleibt wirkungslos, da von dort keine Transition mit `R` existiert.

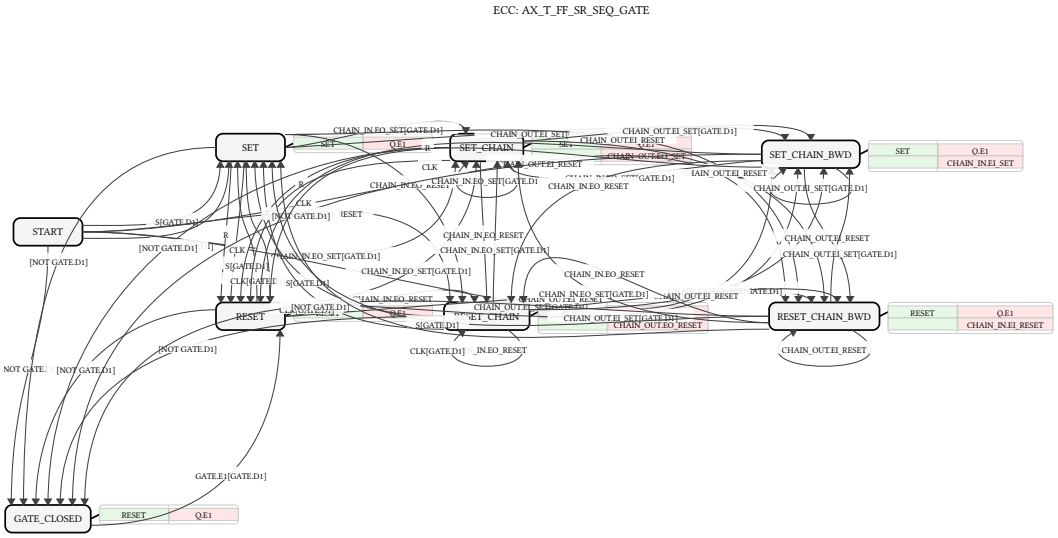
Anwendungsszenarien

- Gestaffeltes Ein-/Ausschalten mehrerer Förderbänder (z. B. Igelband, Elevator, Querband) mit Verzögerung über `ASR2_DELAY`

Fazit

`AX_T_FF_SR_SEQ` erweitert das Set/Reset/Toggle-Flipflop um vier getrennte Kettenpfade (vorwärts/rückwärts, Set/Reset) für kaskadierte, gestaffelte Schaltvorgänge.

AX_T_FF_SR_SEQ_GATE



• – ○ • – ○ • – ○ •

Einleitung

Der `AX_T_FF_SR_SEQ_GATE` ist ein Toggle-/Set-/Reset-Flipflop mit bidirektionaler Ketten-Weitergabe, wie `AX_T_FF_SR_SEQ`, und zusätzlich mit einem Freigabe-Socket `GATE` (Typ `AX`) und einem eigenen Zustand `GATE_CLOSED`. Solange `GATE` TRUE ist, arbeitet der Baustein normal. Fällt `GATE` auf FALSE, wird sofort zwangsweise ein RESET ausgeführt und der Baustein gesperrt; der einzige Ausweg aus `GATE_CLOSED` ist `GATE.E1[GATE.D1]`, also das Wiederöffnen des Gates.

Schnittstellenstruktur

Ereignis-Eingänge

- **S**: setzt den Ausgang `Q` (lokal, wirkt nie auf `CHAIN_IN`/`CHAIN_OUT`); wirkungslos während `GATE_CLOSED`
- **R**: setzt den Ausgang `Q` zurück (lokal); wirkungslos während `GATE_CLOSED`
- **CLK**: Takt zum Umschalten (Toggle) des Ausgangs (lokal); wirkungslos während `GATE_CLOSED`

Ereignis-Ausgänge

Der Funktionsblock verfügt über keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock verfügt über keine direkten Daten-Eingänge.

Daten-Ausgänge

Der Funktionsblock verfügt über keine direkten Daten-Ausgänge.

Adapter

- **Q** (Plug, `adapter::types::unidirectional::AX`): Wert des Flipflops
- **CHAIN_OUT** (Plug, `adapter::types::bidirectional::ASR2`): vorwärts Set/Reset an das nächste Kettenglied (`EO_SET`/`EO_RESET`), rückwärts Set/Reset vom nächsten Kettenglied empfangen (`EI_SET`/`EI_RESET`); nicht während `GATE_CLOSED`

- **CHAIN_IN** (Socket, `adapter::types::bidirectional::ASR2`): vorwärts Set/Reset vom vorherigen Kettenglied empfangen (`EO_SET`/`EO_RESET`), rückwärts Set/Reset an das vorherige Kettenglied weiterreichen (`EI_SET`/`EI_RESET`); nicht während `GATE_CLOSED`
- **GATE** (Socket, `adapter::types::unidirectional::AX`): Freigabe, muss TRUE sein. Fällt `GATE` auf FALSE, folgt sofort `GATE_CLOSED` (Zwangs-RESET und Sperre); verlassen wird der Zustand nur über `GATE.E1[GATE.D1]`

Funktionsweise

Der ECC besitzt die Zustände `START`, `SET`, `RESET`, `SET_CHAIN`, `RESET_CHAIN`, `SET_CHAIN_BWD`, `RESET_CHAIN_BWD` und `GATE_CLOSED`. Es gibt vier strikt getrennte Ketten-Pfade (Vorwärts-Set, Vorwärts-Reset, Rückwärts-Set, Rückwärts-Reset) ohne Event-Verdopplung:

- **Lokal**: `S[GATE.D1]` oder `CLK[GATE.D1]` führt nach `SET` (Algorithmus `SET`: `Q.D1 := TRUE`, Ausgabe `Q.E1`); `R` oder `CLK` aus einem gesetzten Zustand führt nach `RESET` (`Q.D1 := FALSE`). Diese lokalen Ereignisse wirken nie auf die Kette.
- **Vorwärts**: `CHAIN_IN.EO_SET[GATE.D1]` führt nach `SET_CHAIN` (setzt `Q` und gibt `CHAIN_OUT.EO_SET` aus), `CHAIN_IN.EO_RESET` führt nach `RESET_CHAIN` (setzt `Q` zurück und gibt `CHAIN_OUT.EO_RESET` aus).
- **Rückwärts**: `CHAIN_OUT.EI_SET[GATE.D1]` führt nach `SET_CHAIN_BWD` (setzt `Q` und gibt `CHAIN_IN.EI_SET` aus), `CHAIN_OUT.EI_RESET` führt nach `RESET_CHAIN_BWD` (setzt `Q` zurück und gibt `CHAIN_IN.EI_RESET` aus).
- **Gate**: Jeder Zustand außer `GATE_CLOSED` besitzt als erste Transition `[NOT GATE.D1]` nach `GATE_CLOSED`. Dort wird der Algorithmus `RESET` mit Ausgabe `Q.E1` ausgeführt; `S`, `R`, `CLK`, `CHAIN_IN` und `CHAIN_OUT` sind wirkungslos, weil keine Transition definiert ist. Zurück nach `RESET` geht es nur mit `GATE.E1[GATE.D1]`.

Technische Besonderheiten

- Die Gate-Prüfung `[NOT GATE.D1]` ist ereignislos und steht in jedem Zustand an erster Stelle der Transitionsreihenfolge, damit sie vor den ungeguardeten Ereignis-Transitionen (z. B. `R`, `CLK`, `CHAIN_IN.EO_RESET`) ausgewertet wird
- Die Sperre ist pegelbasiert und kein reiner Flanken-Puls auf `R`: Solange das Gate geschlossen ist, kann der Baustein nicht wieder eingeschaltet werden
- Set-Anforderungen (`S`, `CLK` zum Setzen, `CHAIN_IN.EO_SET`, `CHAIN_OUT.EI_SET`) werden nur bei `GATE.D1 = TRUE` akzeptiert; Reset-Pfade dürfen immer durchlaufen
- Während `GATE_CLOSED` wird nichts an `CHAIN_IN`/`CHAIN_OUT` weitergereicht

Zustandsübersicht

Zustand	Bedeutung
START	Anfangszustand; bei `GATE.D1 = FALSE` sofort nach `GATE_CLOSED`
SET	`Q` = TRUE durch lokales `S`/`CLK`
RESET	`Q` = FALSE durch lokales `R`/`CLK`
SET_CHAIN	`Q` = TRUE, Set kam vorwärts und wird an `CHAIN_OUT.EO_SET` weitergereicht
RESET_CHAIN	`Q` = FALSE, Reset kam vorwärts und wird an `CHAIN_OUT.EO_RESET` weitergereicht
SET_CHAIN_BWD	`Q` = TRUE, Set kam rückwärts und wird an `CHAIN_IN.EI_SET` weitergereicht
RESET_CHAIN_BWD	`Q` = FALSE, Reset kam rückwärts und wird an `CHAIN_IN.EI_RESET` weitergereicht
GATE_CLOSED	gesperrt, `Q` = FALSE; Ausgang nur über `GATE.E1[GATE.D1]` nach `RESET`

Anwendungsszenarien

- Schaltbare Ausgänge mit übergeordneter, pegelbasierter Sicherheitsabschaltung (z. B. Zapfwellen-Überwachung), die erst nach Wiedererteilung der Freigabe wieder eingeschaltet werden dürfen
- Sequenzierte Ketten aus mehreren Flipflops, in denen ein Glied freigabeabhängig gesperrt werden soll

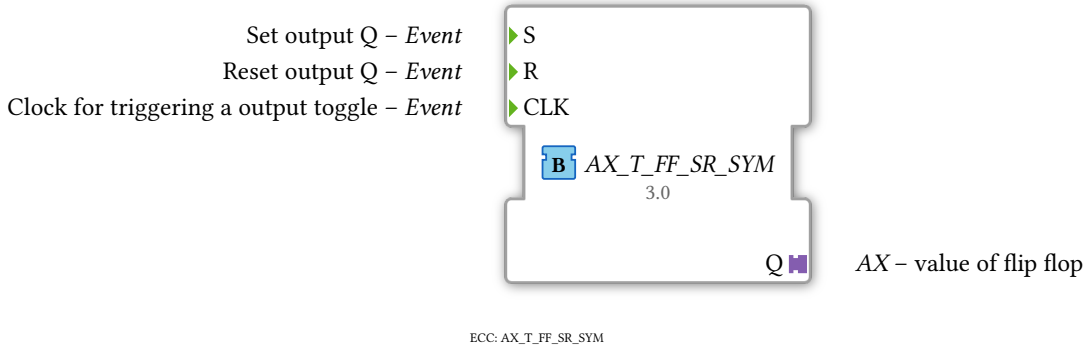
⚖️ Vergleich mit ähnlichen Bausteinen

Gegenüber `AX_T_FF_SR_SEQ` (gleiche vier Ketten-Pfade ohne Freigabe) bietet `AX_T_FF_SR_SEQ_GATE` zusätzlich den Socket `GATE` und den Zustand `GATE_CLOSED`. Die Sperre ist hier echt pegelbasiert und löst damit ein Problem, das ein reiner Flanken-Puls auf `R` nicht löst: das sofortige Wiedereinschalten trotz anliegender Abschaltung.

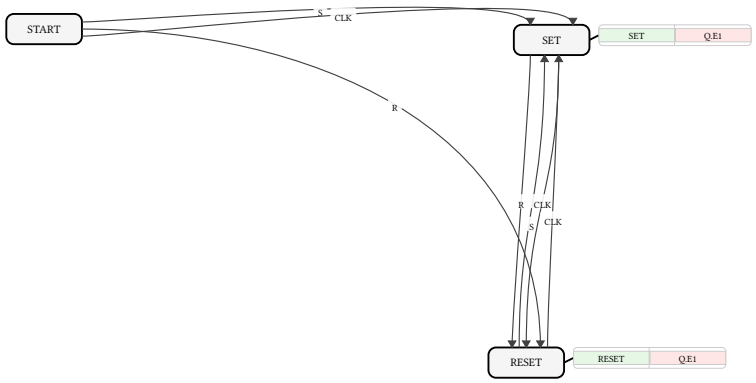
Fazit

Der `AX_T_FF_SR_SEQ_GATE` erweitert das sequenzierbare AX-Flipflop um eine pegelbasierte Freigabe: Fällt `GATE`, wird sofort zurückgesetzt und gesperrt, und erst ein erneutes `GATE.E1[GATE.D1]` gibt den Baustein wieder frei.

AX_T_FF_SR_SYM



ECC: AX_T_FF_SR_SYM



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock AX_T_FF_SR_SYM ist ein ereignisgesteuerter bistabiler Kippschalter mit zusätzlicher Toggle-Funktion und symmetrischem Startverhalten. Er kombiniert die Eigenschaften eines SR-Flipflops (Setzen und Rücksetzen) mit denen eines T-Flipflops (Umschalten bei jedem Taktsignal). Das Verhalten nach dem Einschalten ist symmetrisch ausgelegt, d.h. der Baustein befindet sich zunächst in einem neutralen Startzustand und geht erst durch ein Ereignis in einen definierten Zustand über.

Schnittstellenstruktur

Ereignis-Eingänge

- **S** – Set: Setzt den Ausgang Q auf TRUE.
- **R** – Reset: Setzt den Ausgang Q auf FALSE.
- **CLK** – Clock: Schaltet (toggelt) den Ausgang Q um (TRUE → FALSE oder FALSE → TRUE).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge. Die Ausgabe des Zustands erfolgt über den Adapter Q. Der Adapter löst bei Zustandsänderung das Ereignis Q.E1 aus.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine direkten Daten-Ausgänge. Der aktuelle Wert wird über den Adapter Q bereitgestellt (Q.D1).

Adapter

- **Q**: Adapter vom Typ `adapter::types::unidirectional::AX` (unidirektional). Stellt den aktuellen Zustand des Flipflops als booleschen Wert (TRUE/FALSE) zur Verfügung. Der Adapter löst nach jeder Zustandsänderung das Ereignis Q.E1 aus.

Funktionsweise

Der FB besitzt drei Ereignis-Eingänge und eine Zustandsmaschine mit drei Zuständen: `START`, `SET` und `RESET`.

- **START** ist der initiale Zustand nach dem Einschalten. In diesem Zustand ist kein Algorithmus aktiv, und der Ausgangswert ist undefiniert.
  - Bei Eintreffen von `S` oder `CLK` wechselt der FB in den Zustand `SET`.
  - Bei Eintreffen von `R` wechselt er in den Zustand `RESET`.
- Im Zustand **SET** wird der Algorithmus `SET` ausgeführt: `Q.D1 := TRUE`. Anschließend wird das Ereignis `Q.E1` ausgelöst.
  - Bei `R` oder `CLK` wechselt der FB in den Zustand `RESET`.
- Im Zustand **RESET** wird der Algorithmus `RESET` ausgeführt: `Q.D1 := FALSE`. Anschließend wird `Q.E1` ausgelöst.
  - Bei `S` oder `CLK` wechselt der FB zurück in den Zustand `SET`.

Die Toggle-Funktion wird durch die `CLK`-Transitionen realisiert: unabhängig vom aktuellen Zustand führt `CLK` zum jeweils anderen Zustand.

Technische Besonderheiten

- **Symmetrisches Startverhalten**: Nach dem Einschalten befindet sich der FB im neutralen `START`-Zustand. Erst das erste Ereignis legt den Anfangszustand fest. Dadurch kann das System definiert initialisiert werden.
- **Adapter-basierte Ausgabe**: Statt eines direkten Datenausgangs wird der Zustand über einen Adapter (`AX`) bereitgestellt. Dies ermöglicht eine flexible Kopplung mit anderen Bausteinen und eine einfache Ereignisweitergabe.
- **Kombination von SR- und T-Flipflop**: Der Baustein vereint Setzen, Rücksetzen und Toggeln in einer kompakten Einheit.

Zustandsübersicht

Zustand	Beschreibung	Ausgang Q.D1	Ereignis nach Algorithmus
START	Initialzustand, kein definierter Wert	undefiniert	-
SET	Ausgang wird auf TRUE gesetzt	TRUE	Q.E1
RESET	Ausgang wird auf FALSE gesetzt	FALSE	Q.E1

Übergänge (Ereignis ⟶ Zielzustand):

- START ⟶ SET: `S`, `CLK`
- START ⟶ RESET: `R`
- SET ⟶ RESET: `R`, `CLK`
- RESET ⟶ SET: `S`, `CLK`

Anwendungsszenarien

- **Steuerung von binären Ausgängen**: Ein Ausgang kann sowohl gesetzt, zurückgesetzt als auch umgeschaltet werden (z.B. manuelle Bedienung mit Set/Reset und automatischem Toggle durch einen Takt).
- **Hin- und Herschalten**: Ein Prozess, der bei jedem Taktsignal seinen Zustand wechselt, aber auch explizit gesetzt oder zurückgesetzt werden kann.
- **Definierte Startsequenzen**: Das symmetrische Startverhalten stellt sicher, dass der Baustein nach dem Einschalten noch keinen Ausgangswert ausgibt, bis ein Befehl eintrifft.

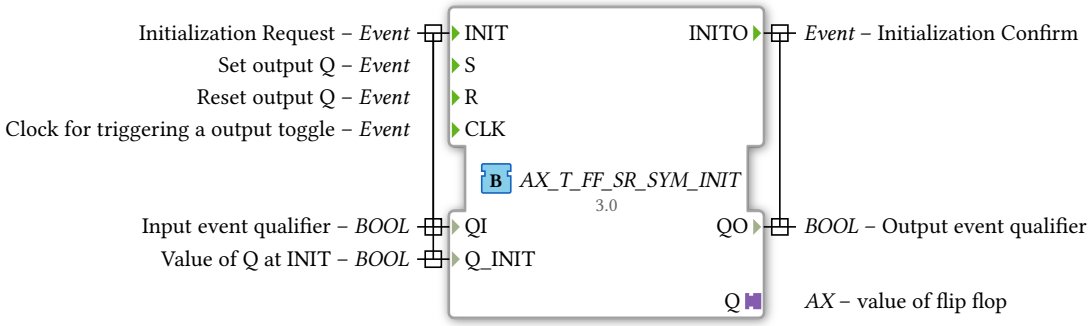
Vergleich mit ähnlichen Bausteinen

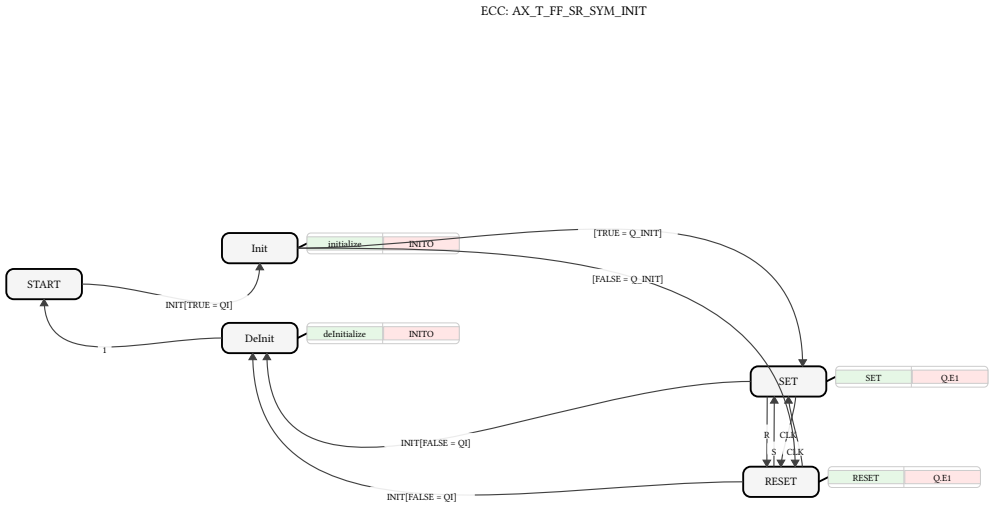
- **E_SR (reines SR-Flipflop)**: Hat nur Set- und Reset-Eingänge, keine Toggle-Funktion. Nach dem Start ist der Ausgang üblicherweise FALSE.
- **E_T (Toggle-Flipflop)**: Nur Toggle-Eingang, kein Set/Reset.
- **AX_T_FF_SR_SYM**: Vereint alle drei Funktionen und bietet zusätzlich einen neutralen Startzustand, der eine flexible Initialisierung erlaubt.

Fazit

Der AX_T_FF_SR_SYM ist ein vielseitiger Funktionsblock für binäre Zustandslogik. Die Kombination aus Set, Reset und Toggle sowie das symmetrische Startverhalten machen ihn besonders geeignet für Anwendungen, die sowohl manuelle Steuerung als auch automatisches Umschalten erfordern. Die Adapter-Schnittstelle erleichtert die Einbindung in modulare Systeme wie IEC 61499.

AX_T_FF_SR_SYM_INIT





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AX_T_FF_SR_SYM_INIT** realisiert ein ereignisgesteuertes bistabiles Kippglied (Flip-Flop) mit **Set**-, **Reset**- und **Toggle**-Funktionalität. Er zeichnet sich durch ein **symmetrisches Start-up-Verhalten** aus, bei dem der Ausgangszustand nach dem INIT-Ereignis über den Parameter `Q_INIT` definiert werden kann. Der Baustein ist gemäß IEC 61499-1 Annex A ausgelegt und eignet sich für Anwendungen, die einen rückgesetzten, gesetzten oder umschaltbaren binären Zustand mit einer initialisierbaren Voreinstellung benötigen.

Schnittstellenstruktur

Ereignis-Eingänge

Event	Typ	Kommentar
INIT	EInit	Initialisierungsanforderung
S	Event	Setzen des Ausgangs Q (abhängig von QI)
R	Event	Rücksetzen des Ausgangs Q (abhängig von QI)
CLK	Event	Takt zum Umschalten (Toggle) von Q

Ereignis-Ausgänge

Event	Typ	Kommentar
INITO	EInit	Bestätigung der Initialisierung

Daten-Eingänge

Variable	Typ	Kommentar
QI	BOOL	Ereignisqualifizierer (Freigabe von Aktionen)
Q_INIT	BOOL	Wert von Q beim INIT-Ereignis

Daten-Ausgänge

Variable	Typ	Kommentar
QO	BOOL	Ausgangs-Ereignisqualifizierer

Adapter

Adapter	Typ	Kommentar
Q	adapter::types::unidirectional::AX	Wert des Flip-Flops (D1)

Funktionsweise

Der Baustein besitzt eine endliche Zustandsmaschine mit fünf Zuständen: **START**, **Init**, **DeInit**, **SET** und **RESET**.

- **START** ist der initiale Zustand nach dem Einschalten.
- Ein **INIT**-Ereignis leitet die Initialisierung ein:
  - Wenn `QI = TRUE`, wird in den **Init**-Zustand gewechselt. Die Algorithmen `initialize` setzen `QO := QI`.
  - Anschließend wird anhand von `Q_INIT` entschieden, ob in den **SET**-Zustand (`Q_INIT = TRUE`) oder in den **RESET**-Zustand (`Q_INIT = FALSE`) übergegangen wird.
- Während des Betriebs können **S**, **R** oder **CLK** verarbeitet werden:
  - Bei **S** (Set) wird im Algorithmus `SET` der Adapter-Ausgang `Q.D1` auf `TRUE` gesetzt, sofern `QI = TRUE` ist. `QO` erhält den Wert von `QI`.
  - Bei **R** (Reset) wird `Q.D1` auf `FALSE` gesetzt, ebenfalls nur bei aktivem `QI`.
  - Bei **CLK** (Toggle) wird der Zustand umgeschaltet: Ist der FB im Zustand **SET**, führt ein CLK-Ereignis in den **RESET**-Zustand; ist er in **RESET**, führt es in **SET**. Die Toggle-Funktion arbeitet unabhängig von `QI`? *Anmerkung:* Die Transitionen `SET–CLK→RESET` und `RESET–CLK→SET` sind ohne weitere Bedingung definiert, jedoch wird in den Algorithmen `Q.D1` nur bei `QI = TRUE` geändert. Falls `QI = FALSE`, bleibt der Adapter-Wert unverändert, aber der Zustandsübergang findet trotzdem statt. Der interne Zustand (SET/RESET) wird also immer gewechselt, die tatsächliche Ausgabe auf `Q.D1` erfolgt aber nur, wenn `QI` wahr ist.
- Ein erneutes **INIT**-Ereignis mit `QI = FALSE` führt in den **DeInit**-Zustand, in dem `QO := FALSE` gesetzt wird. Danach kehrt der FB in den **START**-Zustand zurück.

Der Ausgang `QO` wird bei jedem ausgeführten Algorithmus auf den aktuellen Wert von `QI` gesetzt. Er dient als Ereignisqualifizierer für den INITO-Ausgang.

Technische Besonderheiten

- **Symmetrisches Start-up-Verhalten**: Der Anfangswert des Ausgangs wird über `Q_INIT` festgelegt, sodass sowohl gesetzte als auch rückgesetzte Startzustände möglich sind.
- **Ereignisqualifizierer QI**: Die eigentliche Änderung des Adapters `Q.D1` (also des Flip-Flop-Ausgangs) wird nur dann ausgeführt, wenn `QI = TRUE` ist. Bei `QI = FALSE` werden die Ereignisse S, R und CLK zwar verarbeitet (Zustandswechsel), aber der Adapter-Wert bleibt unverändert.
- **Toggle-Funktionalität**: Durch das CLK-Ereignis kann der Ausgang zwischen 0 und 1 umgeschaltet werden, was in vielen Anwendungen eine einfache Rechteckgenerierung oder Zustandswechsel ermöglicht.
- **INIT-Behandlung**: Das INIT-Ereignis kann sowohl für die Erstinitialisierung (`QI = TRUE`) als auch für die Deinitialisierung (`QI = FALSE`) genutzt werden.

Zustandsübersicht

Zustand	Aktion	Ausgang/Ereignis	Beschreibung
START	–	–	Wartet auf erstes INIT-Ereignis.
Init	`initialize`	INITO	Setzt `QO := QI`; danach Wechsel zu SET oder RESET je nach `Q_INIT`.
DeInit	`deInitialize`	INITO	Setzt `QO := FALSE`; danach Rücksprung zu START.
SET	`SET`	Q.E1	Setzt `Q.D1 := TRUE` (falls QI = TRUE) und `QO := QI`.

RESET	RESET	Q.E1	Setzt `Q.D1 := FALSE` (falls QI = TRUE) und `QO := QI`.

Transitionen (auszugsweise):

- START ⟶ Init: `INIT[QI = TRUE]`
- Init ⟶ SET: `Q_INIT = TRUE`
- Init ⟶ RESET: `Q_INIT = FALSE`
- SET ⟶ RESET: `R` oder `CLK`
- RESET ⟶ SET: `S` oder `CLK`
- SET ⟶ DeInit: `INIT[QI = FALSE]`
- RESET ⟶ DeInit: `INIT[QI = FALSE]`
- DeInit ⟶ START: (immer, Bedingung = 1)

Anwendungsszenarien

- **Maschinensteuerung**: Definierter Startzustand nach Einschalten (z.B. Ventil geschlossen = `Q_INIT = FALSE`).
- **Zustandswechsel mit Toggle**: Umschalten einer Beleuchtung oder eines Antriebs bei jedem CLK-Impuls.
- **Sichere Initialisierung**: Über `QI` kann die Aktion an Bedingungen geknüpft werden (z.B. Freigabe durch übergeordnete Steuerung).
- **Fehlerrückstellung**: Ein RESET-Ereignis setzt den Ausgang zurück, während ein Setzen durch S möglich ist.

Vergleich mit ähnlichen Bausteinen

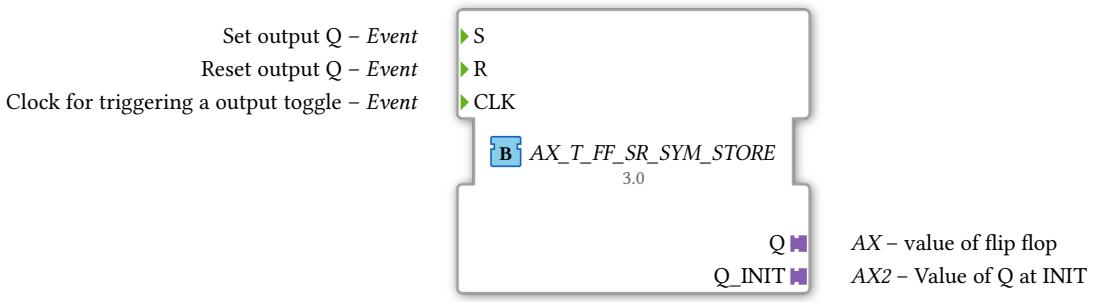
Baustein	Besonderheit
`E_SR` (Standard IEC 61499)	Reines Set/Reset-Flip-Flop ohne Toggle und ohne INIT-Symmetrie.
`E_RS`	Wie E_SR, aber Reset hat Vorrang.
`E_Toggle`	Nur Toggle-Funktion, kein Set/Reset, keine Initialisierung.
`AX_T_FF_SR_SYM_INIT`	Kombiniert Set, Reset, **Toggle**, **symmetrische INIT-Voreinstellung** und Ereignisqualifizierer `QI`.

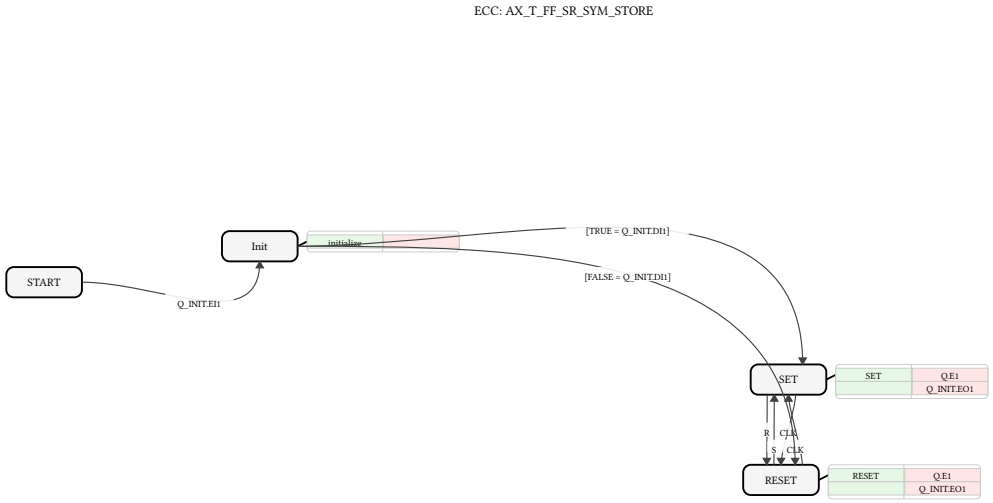
Der hier beschriebene FB bietet somit eine höhere Flexibilität für Anwendungen, die mehrere Betriebsmodi und einen definierten Anfangszustand erfordern.

Fazit

Der **AX_T_FF_SR_SYM_INIT** ist ein vielseitiger und robuster Funktionsblock für die binäre Zustandssteuerung in IEC 61499-Systemen. Seine Kombination aus Set, Reset und Toggle, gepaart mit einem frei konfigurierbaren Startzustand über `Q_INIT` und dem Qualifizierer `QI`, ermöglicht eine kompakte und sichere Abbildung vieler Automatisierungsanforderungen. Durch das durchdachte INIT-Handling eignet er sich besonders für Anwendungen, die ein definiertes und reproduzierbares Start-up-Verhalten verlangen.

AX_T_FF_SR_SYM_STORE





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `AX_T_FF_SR_SYM_STORE` realisiert ein ereignisgesteuertes, bistabiles Kippglied (Flip-flop) mit Setz- (S), Rücksetz- (R) und Toggle-Funktion (CLK). Seine Besonderheit ist das symmetrische Startverhalten: Über einen speziellen Adapter (`Q_INIT`) kann der Anfangszustand beim Systemstart definiert und gespeichert werden.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
S	Event	Setzt den Ausgang `Q` auf TRUE
R	Event	Setzt den Ausgang `Q` auf FALSE
CLK	Event	Toggelt den Ausgang `Q` (TRUE ↔ FALSE)

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Ausgangsdaten werden ausschließlich über die Adapter-Schnittstellen bereitgestellt.

Daten-Eingänge

Keine direkten Daten-Eingänge. Initialwert und Zustand werden über Adapter eingelesen bzw. ausgegeben.

Daten-Ausgänge

Keine direkten Daten-Ausgänge. Der aktuelle Zustand wird über die Adapter verfügbar gemacht.

Adapter

Name	Typ	Kommentar

Q	adapter::types::unidirectional::AX	Liefert den aktuellen Zustand des Flipflops (einadriger boolescher Wert)
Q_INIT	adapter::types::bidirectional::AX2	Bidirektionale Schnittstelle für den Startwert von `Q` bei INIT (beinhaltet drei Ein-/Ausgänge: z.B. DI1/DO1, EI1/EO1)

Funktionsweise

Der Baustein arbeitet als ein SR-Flipflop (Set-Reset) mit zusätzlicher Toggle-Funktion. Der Ablauf gliedert sich in Initialisierung und Betrieb:

1. **Initialisierung (START → Init)** Nach dem Systemstart befindet sich der FB im Zustand START. Sobald am Adapter `Q_INIT` das Ereignis `EI1` eintrifft, wechselt er in den Zustand Init. Dort wird der über `Q_INIT.DI1` bereitgestellte Initialwert ausgelesen.
   - Ist der Initialwert FALSE, wird anschließend der Zustand RESET eingenommen.
   - Ist der Initialwert TRUE, wird der Zustand SET eingenommen. In beiden Fällen wird der entsprechende Ausgang `Q.D1` und der Ausgang `Q_INIT.DO1` auf den Initialwert gesetzt.
2. **Betrieb (SET ↔ RESET)** Ausgehend von SET oder RESET wird bei jedem Ereignis an den Eingängen S, R oder CLK umgeschaltet:
   - **S** → wechselt in SET (setzt Q = TRUE)
   - **R** → wechselt in RESET (setzt Q = FALSE)
   - **CLK** → toggelt den Zustand (von SET nach RESET oder umgekehrt) Bei jeder Zustandsänderung werden die Ausgangsdaten über die Adapter aktualisiert und das Ereignis `Q.E1` sowie `Q_INIT.EO1` ausgegeben.

Technische Besonderheiten

- **Adapter-basierte Ein-/Ausgabe** Statt klassischer DataInputs/DataOutputs verwendet der Baustein Adapter-Schnittstellen. Dies ermöglicht eine flexible Kopplung mit anderen Bausteinen oder Systemkomponenten, die ebenfalls das Adapterprotokoll unterstützen.
- **Symmetrisches Startverhalten** Über den bidirektionalen Adapter `Q_INIT` kann der Anfangszustand sowohl extern vorgegeben als auch nach der Initialisierung ausgelesen werden. Dadurch wird ein deterministisches Wiederanlaufverhalten sichergestellt – besonders wichtig bei sicherheitskritischen oder zustandsbehafteten Anwendungen.
- **Toggle-Funktion (CLK)** Neben den klassischen Setz- und Rücksetzereignissen erlaubt der Takt-Eingang CLK das einfache Umschalten des Ausgangs, ohne dass hierfür ein separates Set/Reset-Signal benötigt wird.
- **Keine Daten-Ein-/Ausgänge** Diese funktionale Reduktion vereinfacht die Schnittstelle und vermeidet Verwechslungen mit den Adaptern.

Zustandsübersicht

Zustand	Beschreibung	Ausgabe bei Ereignis
START	Initialer Wartezustand	Wartet auf Initialisierungsereignis über Q_INIT.EI1
Init	Initialisierung	Liest Q_INIT.DI1, setzt Q und Q_INIT.DO1 entsprechend
SET	Ausgang Q = TRUE	Verarbeitet S, R, CLK (siehe Übergänge)
RESET	Ausgang Q = FALSE	Verarbeitet S, R, CLK (siehe Übergänge)

Zustandsübergänge (vereinfacht)

- START → Init (Ereignis Q_INIT.EI1)
- Init → SET (falls Q_INIT.DI1 = TRUE)
- Init → RESET (falls Q_INIT.DI1 = FALSE)
- SET → RESET (bei Ereignis R oder CLK)
- RESET → SET (bei Ereignis S oder CLK)

Anwendungsszenarien

- **Speicherung von Maschinenzuständen** In Fertigungsanlagen kann dieser Baustein den aktuellen Betriebsmodus (z.B. „Ein“ oder „Aus“) speichern und beim Neustart wiederherstellen.
- **Initialisierung von Steuerungssequenzen** Szenarien, bei denen ein definierter Startwert erforderlich ist (z.B. Ventilstellung zu Beginn einer Charge), profitieren vom symmetrischen INIT-Verhalten.
- **Ereignisgesteuerte Toggle-Funktionen** Zum Beispiel das Umschalten einer Signalleuchte bei jedem Tastendruck, ohne dass separate Ein- und Aus-Befehle nötig sind.

Vergleich mit ähnlichen Bausteinen

- **Standard E_SR (E_R-SET)** Einfaches Set-Reset-Flipflop ohne Toggle und ohne symmetrische Startinitialisierung. Der Anfangszustand ist meist undefiniert oder muss extern gesetzt werden.
- **E_RS** Bietet Priorisierung von Reset, aber ebenfalls keine Toggle- oder INIT-Funktion.
- **Toggle-Bausteine (z.B. E_Toggle)** Reine Toggle-FBs besitzen meist nur einen Takt-Eingang und keinen bidirektionalen Adapter für die Initialisierung. Der AX_T_FF_SR_SYM_STORE vereint alle drei Funktionen in einem Baustein.
- **Adapter-basierte FBs** Wenige FBs in der 4diac-Bibliothek verwenden Adapter für die Zustandsausgabe. Dieses Design ermöglicht eine saubere Trennung von Ereignis- und Datenpfaden und erleichtert die Wiederverwendung in komplexen Systemen.

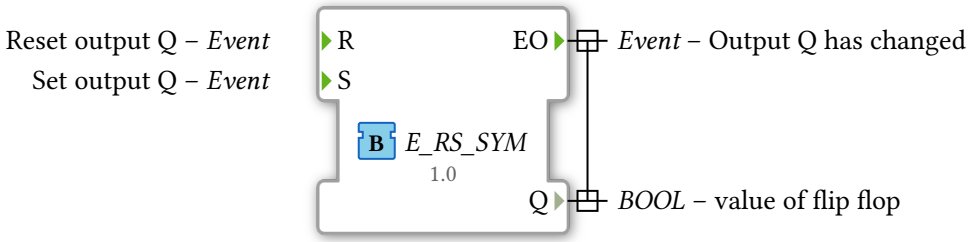
Fazit

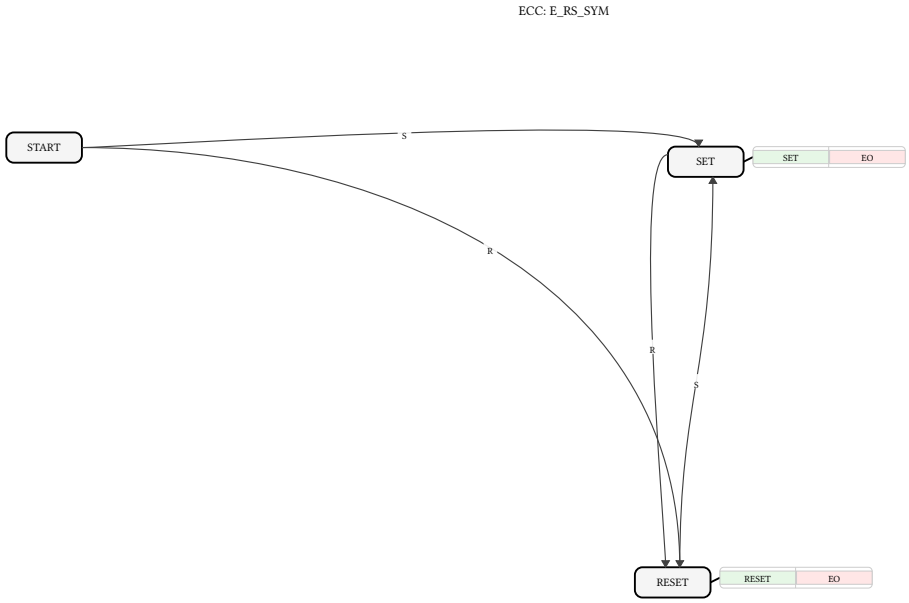
Der `AX_T_FF_SR_SYM_STORE` ist ein leistungsfähiger und flexibler Funktionsblock für zustandsbehaftete Steuerungsaufgaben. Seine Kombination aus Set, Reset und Toggle, gepaart mit dem symmetrischen Startverhalten über Adapter, macht ihn ideal für Anwendungen, die ein definiertes Wiederanlaufverhalten erfordern. Durch die Verwendung von Adaptern fügt er sich nahtlos in modulare und verteilte Automatisierungssysteme ein.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

E_RS_SYM





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **E_RS_SYM** ist ein ereignisgesteuertes, bistabiles Flipflop mit symmetrischem Startverhalten. Er realisiert eine Set-Reset-Funktionalität, bei der der Ausgang Q sowohl durch ein Set- als auch durch ein Reset-Ereignis definiert geschaltet wird. Im Gegensatz zu einem klassischen RS-Flipflop besitzt der Baustein einen expliziten Startzustand, der auf beide Ereignisse gleichermaßen reagiert.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung
R	Event	Reset des Ausgangs Q
S	Event	Setzen des Ausgangs Q

Ereignis-Ausgänge

Name	Typ	Beschreibung	Mit Daten
EO	Event	Ausgang Q hat sich geändert	Ja (mit Q)

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Name	Typ	Beschreibung
Q	BOOL	Wert des Flipflops (TRUE/FALSE)

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Baustein befindet sich nach dem Einschalten im **START**-Zustand. Von diesem Zustand aus wird bei einem Ereignis an `S` der Zustand **SET** und bei einem Ereignis an `R` der Zustand **RESET** erreicht. Dabei wird jeweils der Ausgang `Q` auf `TRUE` (SET) bzw. `FALSE` (RESET) gesetzt und das Ereignis `EO` ausgegeben.

- Im **SET**-Zustand führt ein Ereignis an `R` zurück in den **RESET**-Zustand.
- Im **RESET**-Zustand führt ein Ereignis an `S` zurück in den **SET**-Zustand.
- Der Ausgang `Q` wird nur bei einem Zustandswechsel (d.h. bei Übergang nach SET oder RESET) aktualisiert.

Das symmetrische Startverhalten bedeutet, dass unmittelbar nach der Initialisierung sowohl ein Set- als auch ein Reset-Ereignis akzeptiert werden – es gibt keine Voreinstellung von `Q`.

Technische Besonderheiten

- **Symmetrisches Startverhalten:** Im START-Zustand ist kein Ausgangswert festgelegt. Der Baustein reagiert auf das erste eintreffende Ereignis (S oder R) gleichberechtigt. Dies unterscheidet ihn von üblichen RS-Flipflops, die nach dem Start meist einen definierten (häufig FALSE) Anfangswert liefern.
- **Ereignisgesteuerter Ausgang:** Das Ereignis `EO` wird genau dann ausgelöst, wenn sich der Wert von `Q` ändert. Somit wird eine Signaländerung explizit weitergegeben.
- **Keine Daten-Eingänge:** Der Baustein arbeitet rein ereignisgesteuert ohne zusätzliche Datenparameter.

Zustandsübersicht

Der FB besitzt drei Zustände:

Zustand	Beschreibung
START	Initialzustand nach dem Einschalten; wartet auf erstes S- oder R-Ereignis.
SET	Q = TRUE; kann durch R in den RESET-Zustand wechseln.
RESET	Q = FALSE; kann durch S in den SET-Zustand wechseln.

Transitionen:

- START ⟶ SET bei Ereignis `S`
- START ⟶ RESET bei Ereignis `R`
- SET ⟶ RESET bei Ereignis `R`
- RESET ⟶ SET bei Ereignis `S`

Anwendungsszenarien

- **Initialisierung von Speicherbits** in Steuerungen, bei denen der Anfangswert bewusst nicht vorgegeben werden soll.
- **Symmetrische Umschaltlogiken**, die sowohl das Setzen als auch das Rücksetzen eines Merkers zu Beginn gleichermaßen erlauben.
- **Alternative zu klassischen RS-Haltern**, wenn ein definiertes Anlaufverhalten vermieden werden muss (z. B. bei sicherheitskritischen Anwendungen, wo ein unbestimmter Anfangszustand toleriert wird).

Vergleich mit ähnlichen Bausteinen

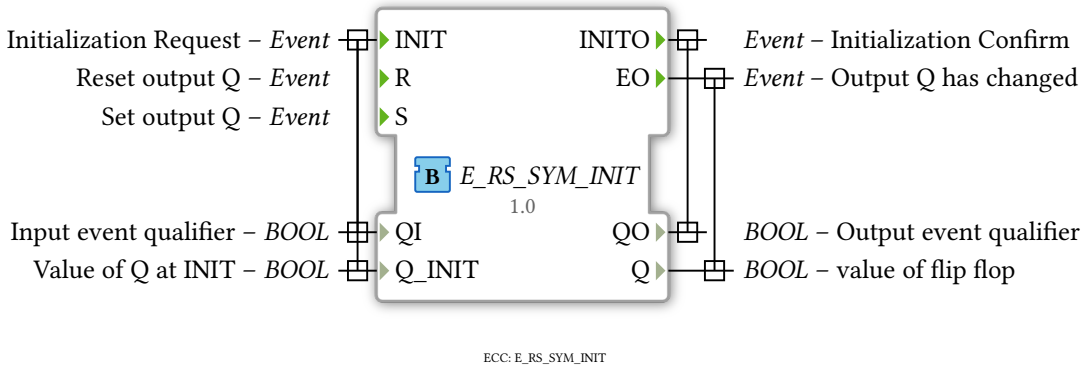
Baustein	Eigenschaft
E_RS	Klassisches RS-Flipflop mit meist asymmetrischem Startverhalten (Q initial FALSE).
E_RS_SYM	Wie E_RS, aber mit symmetrischem Startverhalten – kein voreingestellter Wert, Startzustand akzeptiert beide Ereignisse gleich.
SR-Flipflop	Ähnliche Funktion, jedoch häufig mit Priorität für Set oder Reset; E_RS_SYM ist prioritätsneutral.

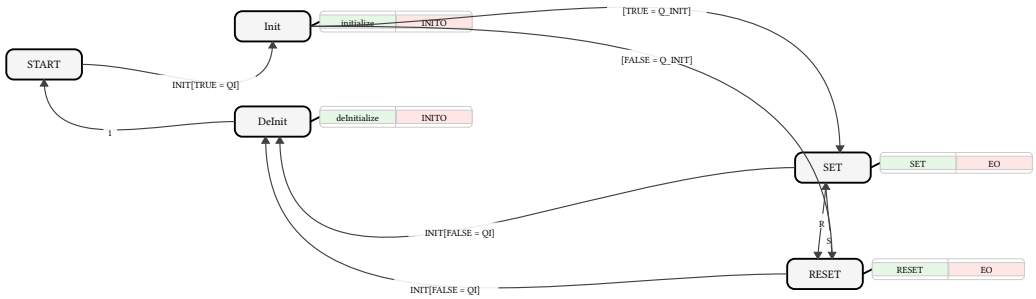
Fazit

Der **E_RS_SYM** eignet sich für alle Anwendungen, die ein einfaches, aber start-symmetrisches Speicherverhalten benötigen. Durch die saubere Trennung von Ereignisänderung (EO) und Datenausgabe (Q)

lässt er sich gut in ereignisgesteuerte Steuerungsarchitekturen nach IEC 61499 einbinden. Seine klare Zustandsmaschine macht ihn zuverlässig und einfach verständlich.

E_RS_SYM_INIT





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **E_RS_SYM_INIT** ist ein ereignisgesteuerter, bistabiler Kippschalter (Flipflop) mit symmetrischem Startverhalten und expliziter Initialisierung. Er realisiert die bekannte RS-Flipflop-Logik, wobei der Ausgang **Q** bei einer Initialisierung (Ereignis **INIT**) wahlweise auf TRUE oder FALSE gesetzt werden kann – je nach Wert des Parameters **Q_INIT**. Dies ermöglicht ein definiertes Verhalten nach einem Systemstart oder einer Reinitialisierung.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar
INIT	EInit	Initialisierungsanforderung; übernimmt die Werte von **QI** und **Q_INIT**
R	Event	Setzt den Ausgang **Q** zurück (Reset)
S	Event	Setzt den Ausgang **Q** (Set)

Ereignis-Ausgänge

Ereignis	Typ	Kommentar
INITO	EInit	Bestätigung der Initialisierung; übergibt den Wert von **QO**
EO	Event	Zeigt an, dass sich der Ausgang **Q** geändert hat

Daten-Eingänge

Variable	Typ	Kommentar
QI	BOOL	Ereignisqualifizierer: Nur wenn *QI=TRUE* werden Set-/Reset-Operationen ausgeführt. Bei Initialisierung legt *QI* den Wert von **QO** fest.
Q_INIT	BOOL	Legt den Anfangswert von **Q** fest, der während der Initialisierung übernommen wird.

Daten-Ausgänge

Variable	Typ	Kommentar
QO	BOOL	Ausgangsqualifizierer: Gibt an, ob eine Operation gültig war (Übernahme von **QI**).
Q	BOOL	Der eigentliche Flipflop-Ausgang.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der **E_RS_SYM_INIT** arbeitet als Zustandsautomat mit fünf Zuständen: **START**, **Init**, **DeInit**, **SET** und **RESET**.

- **Startverhalten**: Nach einem initialen Ereignis **INIT** mit **QI=TRUE** wird der Block in den Zustand **Init** überführt. Abhängig vom Wert **Q_INIT** wechselt er entweder nach **SET** (**Q** = TRUE) oder nach **RESET** (**Q** = FALSE). Der Ausgangsqualifizierer **QO** übernimmt dabei den Wert von **QI** (also TRUE) und wird über **INITO** ausgegeben.
- **Setzen und Rücksetzen**:
  - Im Zustand **SET** führt ein **R**-Ereignis in den Zustand **RESET** die Aktion setzt **Q** auf FALSE (wenn **QI=TRUE**).
  - Im Zustand **RESET** führt ein **S**-Ereignis in den Zustand **SET** die Aktion setzt **Q** auf TRUE (wenn **QI=TRUE**).
  - Bei jeder Zustandsänderung wird **QO** auf den aktuellen Wert von **QI** gesetzt und das Ereignis **EO** ausgegeben.
- **Deinitialisierung**: Sobald ein **INIT**-Ereignis eintritt, während **QI=FALSE** ist (z. B. Deinitialisierung), wechselt der Block aus **SET** oder **RESET** in den Zustand **DeInit**. Dort wird **QO** auf FALSE gesetzt, und der Block kehrt über **INITO** in den **START**-Zustand zurück. Das Flipflop wird dabei nicht zurückgesetzt – **Q** behält seinen bisherigen Wert.
- **Qualifizierer QI**: Die tatsächliche Änderung von **Q** erfolgt nur, wenn **QI=TRUE** ist. Ist **QI=FALSE**, werden die Setz- und Rücksetzsignale ignoriert, aber **QO** gibt dennoch den Wert von **QI** weiter. Dies ermöglicht ein bedingtes Verhalten, z. B. für gültige/ungültige Freigaben.

Technische Besonderheiten

- **Symmetrisches Startverhalten**: Der initiale Zustand von **Q** wird explizit über den Parameter **Q_INIT** festgelegt. Dies unterscheidet den Baustein von einem normalen RS-Flipflop, dessen Startzustand undefiniert ist.
- **INIT als Ereignis mit Parametern**: Der INIT-Ereigniseingang trägt gleichzeitig die Werte **QI** und **Q_INIT**, sodass Initialisierung und Deinitialisierung klar getrennt sind.
- **QC (Event-Qualifier) wird durchgeschliffen**: Bei jeder gültigen Operation (Set, Reset oder INIT) wird **QO** auf den Wert von **QI** gesetzt, sodass die aufrufende Applikation die Gültigkeit der Operation erkennen kann.

- **Zustandsmaschine mit fünf Zuständen**: Die Trennung von Start-, Initialisierungs- und Deinitialisierungszuständen ermöglicht ein robustes Verhalten, insbesondere in Automatisierungssystemen mit zyklischen Neustarts.

Zustandsübersicht

Zustand	Beschreibung	Aktion beim Eintritt	Ausgangsereignis
START	Initialer Ruhezustand nach Systemstart oder nach Deinitialisierung	keine	–
Init	Initialisierung wurde angefordert (**INIT** mit **QI**=**TRUE**)	setzt **QO** = **QI**	INITO
DeInit	Deinitialisierung wurde angefordert (**INIT** mit **QI**=**FALSE**)	setzt **QO** = FALSE	INITO
SET	Flipflop ist gesetzt (**Q** = **TRUE**)	Setzen von **Q** auf TRUE (nur wenn **QI**=**TRUE**)	EO
RESET	Flipflop ist zurückgesetzt (**Q** = **FALSE**)	Setzen von **Q** auf FALSE (nur wenn **QI**=**TRUE**)	EO

Übergänge:

- START ⟶ Init: INIT ∧ (QI = TRUE)
- Init ⟶ SET: Q_INIT = TRUE
- Init ⟶ RESET: Q_INIT = FALSE
- SET ⟶ RESET: R
- RESET ⟶ SET: S
- SET ⟶ DeInit: INIT ∧ (QI = FALSE)
- RESET ⟶ DeInit: INIT ∧ (QI = FALSE)
- DeInit ⟶ START: 1 (immer)

Anwendungsszenarien

- **Steuerungen mit definiertem Einschaltverhalten**: Wenn nach einem Neustart der Automatisierungsanlage ein bestimmter Startwert für einen Merker oder ein Ausgang benötigt wird (z. B. TRUE für „Anlage läuft“ oder FALSE für „stillgelegt“), kann **Q_INIT** entsprechend gesetzt werden.
- **Sichere Resets bei Störungen**: Durch die Deinitialisierung (INIT mit QI=FALSE) kann der Baustein in einen definierten „Ausgangszustand“ versetzt werden, ohne den aktuellen Wert von **Q** zu löschen. Dies ist nützlich, um z. B. einen „Wiederanlauf nach Störung“ zu realisieren.
- **Qualifizierte Setz- und Rücksetzoperationen**: Über **QI** kann eine übergeordnete Bedingung (z. B. „Freigabe aktiver Betrieb“) geschaltet werden. Nur wenn **QI** = TRUE, haben Setzen und Rücksetzen tatsächlich Auswirkung auf **Q**.
- **Initialisierung von Zustandsautomaten**: Der Baustein eignet sich hervorragend als Grundbaustein für eigene Zustandsautomaten, die bei INIT einen bestimmten Anfangszustand einnehmen sollen.

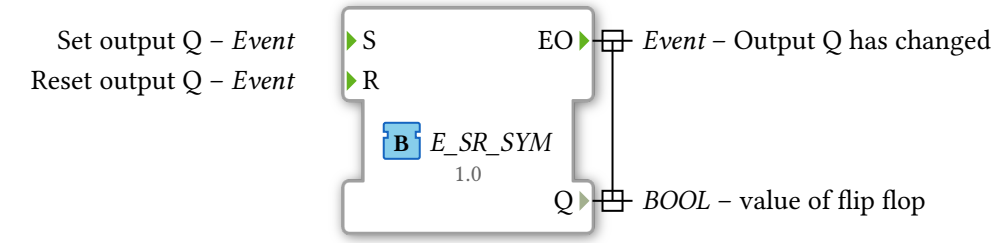
Vergleich mit ähnlichen Bausteinen

Baustein	Unterschied zu E_RS_SYM_INIT
E_RS (Standard RS-Flipflop)	Kein explizites Startverhalten; bei INIT wird **Q** nicht definiert. **Q_INIT** fehlt.
E_SR (Set-Dominant-Flipflop)	Dominanz von Set über Reset; kein initialisierbarer Startzustand.
E_RS_SYM	Symmetrische RS-Logik ohne INIT-Funktion; kein Startverhalten.
E_RS_SYM_INIT (dieser)	Kombiniert symmetrische RS-Logik mit wählbarem Initialwert und Deinitialisierung.

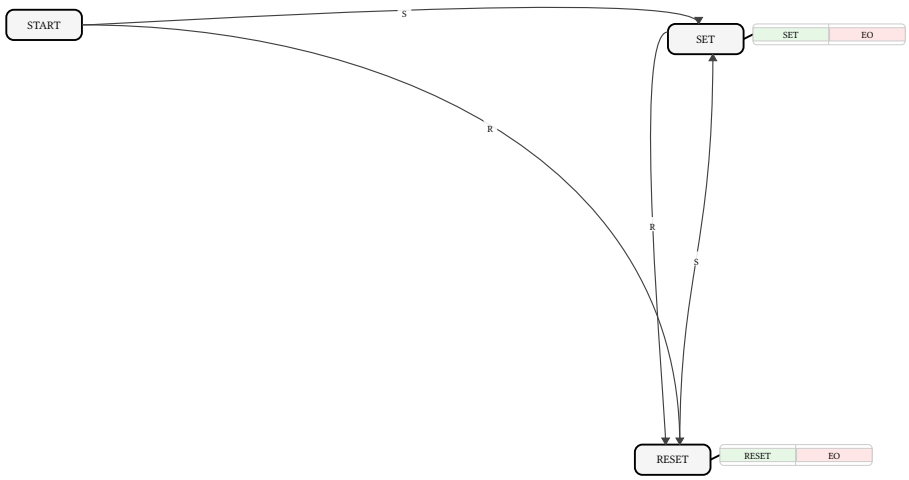
Fazit

E_RS_SYM_INIT ist ein leistungsstarker und flexibler Funktionsblock der IEC 61499-1 für ereignisgesteuerte bistabile Kippschalter. Durch die explizite Initialisierung mit **Q_INIT** und die Möglichkeit zur Deinitialisierung unter Beibehaltung des Ausgangswertes eignet er sich besonders für sicherheitsrelevante und automatisierungstechnische Anwendungen, bei denen ein definiertes Startverhalten und eine saubere Trennung von Initialisierung und Betrieb gefordert sind. Die Berücksichtigung eines Qualifizierers **QI** ermöglicht zudem eine effiziente Einbindung in komplexe Ablaufsteuerungen.

E_SR_SYM



ECC: E_SR_SYM



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock E_SR_SYM ist ein ereignisgesteuertes bistabiles Element mit symmetrischem Startverhalten. Er realisiert ein Flip-Flop, das auf Set- und Reset-Ereignisse reagiert und dessen Anfangszustand nicht vorab festgelegt ist, sondern durch das erste eintreffende Ereignis bestimmt wird.

Schnittstellenstruktur

Ereignis-Eingänge

- **S** (Event): Setzt den Ausgang (Q) auf `TRUE`.
- **R** (Event): Setzt den Ausgang (Q) auf `FALSE`.

Ereignis-Ausgänge

- **EO** (Event): Wird ausgelöst, sobald sich der Wert von (Q) ändert (nach einem S- oder R-Ereignis). Der Ausgangsdatenspeicher (Q) ist diesem Ereignis zugeordnet (`With Var="Q"`).

Daten-Eingänge

Keine.

Daten-Ausgänge

- **Q** (BOOL): Aktueller logischer Zustand des Flip-Flops (`TRUE` oder `FALSE`).

Adapter

Keine.

Funktionsweise

Der Baustein arbeitet zustandsbasiert und verfügt über drei Zustände:

- **START** – Initialzustand nach dem Einschalten oder Zurücksetzen der Steuerung.
- **SET** – Zustand nach einem Set-Ereignis.
- **RESET** – Zustand nach einem Reset-Ereignis.

Trifft im Zustand START ein Ereignis an **S** ein, wechselt der Baustein in den Zustand SET, setzt (Q := \text{TRUE}) und sendet das Ereignis **EO**. Trifft stattdessen ein Ereignis an **R** ein, wechselt er in RESET, setzt (Q := \text{FALSE}) und sendet ebenfalls **EO**.

Befindet sich der Baustein bereits in einem der aktiven Zustände (SET oder RESET), so kann er durch das jeweils andere Ereignis umgeschaltet werden. Dadurch ergibt sich ein symmetrisches Verhalten: Weder S noch R haben eine Priorität – der letzte eingegangene Impuls bestimmt den Ausgang.

Technische Besonderheiten

- **Symmetrisches Startverhalten:** Im Gegensatz zu klassischen SR-Flip-Flops, die nach dem Start einen festen Anfangswert (meist `FALSE`) liefern, wartet E_SR_SYM im Zustand START auf das erste Ereignis. Der Ausgang (Q) bleibt bis dahin in einem undefinierten bzw. nicht gesetzten Zustand. Dies ermöglicht eine flexible Initialisierung durch die Umgebung.
- **Zustandsautomaten mit drei Zuständen:** Die ECC verwendet die Zustände START, SET und RESET sowie zwei Algorithmen (SET, RESET) in Structured Text.
- **Kompakte Schnittstelle:** Nur zwei Ereigniseingänge, ein Ereignisausgang und ein Datenausgang – dadurch einfach in größere Ablaufsteuerungen integrierbar.

Zustandsübersicht

Zustand	Bedeutung	Aktion beim Eintritt	Auslöser für Transitionen
START	Initialzustand	Keine (Warten auf erstes Ereignis)	S ⟶ SET, R ⟶ RESET
SET	Ausgang (Q) ist `TRUE`	Führe Algorithmus SET aus ((Q := \text{TRUE})), sende EO	R ⟶ RESET
RESET	Ausgang (Q) ist `FALSE`	Führe Algorithmus RESET aus ((Q := \text{FALSE})), sende EO	S ⟶ SET

Transitionen:

- START ⟶ SET bei **S**
- START ⟶ RESET bei **R**
- SET ⟶ RESET bei **R**
- RESET ⟶ SET bei **S**

Anwendungsszenarien

- **Ereignisgesteuerte Freigabesignale:** Ein binäres Freigabesignal wird durch einen Set-Impuls gesetzt und durch einen Reset-Impuls zurückgesetzt.
- **Initial unbestimmte Systeme:** Anlaufverhalten, bei dem der erste gültige Befehl (Setzen oder Zurücksetzen) den weiteren Ablauf bestimmt.
- **Einfache Zweipunktregelung:** Schalten von Aktoren (z. B. Ventil auf/zu) mittels externer Befehlsgeber.
- **Toggel-Ersatz:** Kombination mit nachgeschalteten Logikbausteinen ermöglicht auch Toggle-Funktionen.

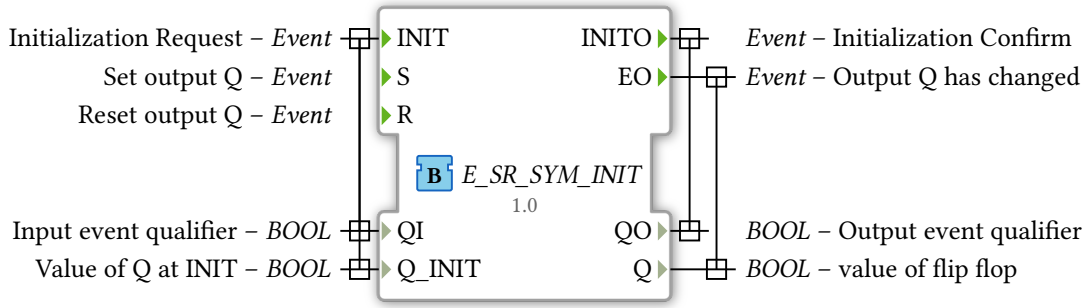
Vergleich mit ähnlichen Bausteinen

Der Standardbaustein **E_SR** (bzw. E_RS) besitzt in der Regel einen vordefinierten Anfangszustand (meist `FALSE`) und setzt sich nach dem Start sofort in den Zustand RESET. Bei E_SR_SYM hingegen ist der Anfangszustand neutral (START), was eine symmetrische Behandlung beider Ereignisse erlaubt. Im Gegensatz zu dominanten Varianten (E_SR mit Set-Dominanz oder E_RS mit Reset-Dominanz) gibt es hier keine Priorität – das zuletzt eingetroffene Ereignis bestimmt den Ausgang.

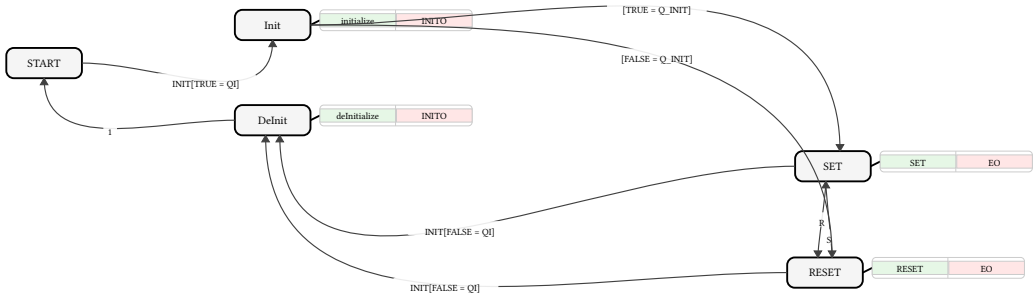
Fazit

Der Funktionsblock **E_SR_SYM** bietet eine flexible, ereignisgesteuerte bistabile Speicherzelle mit symmetrischem Startverhalten. Seine einfache Zustandsmaschine und die klare Schnittstelle machen ihn besonders für Steuerungsaufgaben geeignet, bei denen der Anfangszustand nicht vorgegeben, sondern durch den ersten Impuls der Umgebung bestimmt werden soll. Dadurch unterscheidet er sich von klassischen SR-Bausteinen und erweitert den Werkzeugkasten für ereignisbasierte Automatisierungslösungen.

E_SR_SYM_INIT



ECC: E_SR_SYM_INIT



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `E_SR_SYM_INIT` realisiert ein ereignisgesteuertes, bistabiles Flip-Flop mit symmetrischem Start-up-Verhalten und einer INIT-Schnittstelle. Er erweitert ein einfaches SR-Flip-Flop um die Möglichkeit, beim INIT-Event den Ausgang `Q` auf einen vorgegebenen Wert (`Q_INIT`) zu setzen und eine

Deinitialisierung durchzuführen. Die Eingangsqualifikation QI steuert, ob die Operationen (S, R, INIT) tatsächlich auf Q wirken.

Schnittstellenstruktur

Ereignis-Eingänge

Event	Typ	Kommentar
INIT	EInit	Initialisierungsanforderung; wird mit QI und Q_INIT ausgewertet
S	Event	Setze den Ausgang Q auf TRUE (nur wirksam wenn QI = TRUE)
R	Event	Setze den Ausgang Q auf FALSE (nur wirksam wenn QI = TRUE)

Ereignis-Ausgänge

Event	Typ	Kommentar
INITO	EInit	Bestätigung der Initialisierung/Deinitialisierung
EO	Event	Wird ausgelöst, wenn sich Q geändert hat (durch S oder R)

Daten-Eingänge

Variable	Typ	Kommentar
QI	BOOL	Eingangsqualifikator; steuert, ob die Operationen (S, R, INIT) auf Q wirken
Q_INIT	BOOL	Wert, den Q bei einer erfolgreichen Initialisierung (INIT mit QI = TRUE) annehmen soll

Daten-Ausgänge

Variable	Typ	Kommentar
QO	BOOL	Ausgangsqualifikator; gibt den Wert von QI während der letzten Aktion zurück (bei Deinitialisierung FALSE)
Q	BOOL	Wert des Flip-Flops (bistabiler Ausgang)

Adapter

Keine.

Funktionsweise

Der FB besitzt fünf Zustände: START, Init, DeInit, SET und RESET.

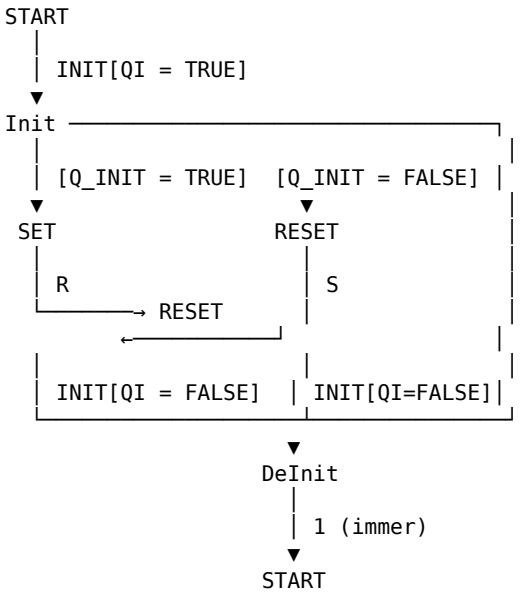
- **START**: Ruhezustand nach der Initialisierung.
- **Init**: Wird durch INIT mit QI = TRUE erreicht. Der Algorithmus initialize setzt QO := QI. Anschließend wird abhängig von Q_INIT in den Zustand SET (wenn Q_INIT = TRUE) oder RESET (wenn Q_INIT = FALSE) gewechselt. Dabei wird Q auf den Wert von Q_INIT gesetzt (über die Algorithmen SET bzw. RESET) und EO ausgelöst.
- **DeInit**: Wird durch INIT mit QI = FALSE aus den Zuständen SET oder RESET erreicht. Der Algorithmus deInitialize setzt QO := FALSE. Anschließend geht der FB automatisch zurück in den Zustand START.
- **SET** und **RESET**: Normale Arbeitszustände des Flip-Flops. Bei einem S-Ereignis wird der Algorithmus SET ausgeführt: QO := QI und falls QI = TRUE wird Q := TRUE gesetzt; EO wird ausgelöst. Bei einem R-Ereignis analog Q := FALSE. Ein INIT mit QI = FALSE führt zurück nach DeInit; INIT mit QI = TRUE wäre nur vom Zustand START möglich.

Die Qualifikation QI wirkt wie ein Freigabesignal: Nur wenn QI = TRUE beeinflussen S, R und die Initialisierung den Ausgang Q. Der Ausgang QO spiegelt den zuletzt gültigen Wert von QI wider (außer nach DeInit, dann ist QO = FALSE).

Technische Besonderheiten

- **Symmetrisches Start-up**: Der Wert von `Q` nach einer Initialisierung wird ausschließlich durch `Q_INIT` bestimmt – unabhängig vom vorherigen Zustand. Dies ermöglicht ein deterministisches Verhalten nach dem Start.
- **Qualifizierte Operationen**: Alle Aktionen (S, R, INIT) werden nur ausgeführt, wenn `QI = TRUE`. Damit kann der FB z. B. über eine übergeordnete Steuerung ein- und ausgeschaltet werden.
- **Deinitialisierung**: Durch ein `INIT` mit `QI = FALSE` gelangt der FB in einen definierten Deinitialisierungszustand (`DeInit`), der keine weiteren Aktionen mehr zulässt, bis ein erneutes `INIT` mit `QI = TRUE` erfolgt.
- **Keine Speicherung von QO**: `QO` wird bei jeder Transition neu gesetzt, es erfolgt keine eigene Speicherung.

Zustandsübersicht



Anwendungsszenarien

- **Steuerungen, die nach dem Einschalten einen definierten Ausgangszustand benötigen** (z. B. Maschinen: Ventil geschlossen oder geöffnet).
- **Systeme mit sicherheitskritischer Initialisierung**, bei denen der Ausgang erst nach Freigabe durch einen Qualifikator (`QI`) gesetzt werden darf.
- **Bausteinketten**, bei denen eine Deinitialisierung (z. B. bei einem Reset des Gesamtsystems) den FB in einen Grundzustand zurückversetzen soll, ohne dass die eigentlichen S/R-Signale noch wirken.

Vergleich mit ähnlichen Bausteinen

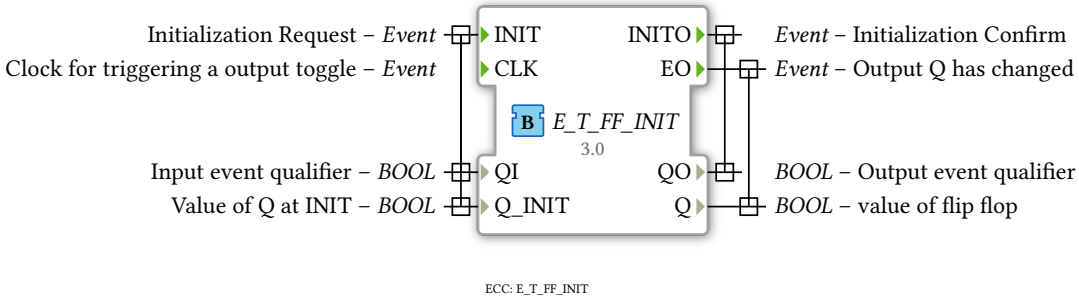
- **E_SR** (Standard SR-Flipflop): Hat kein INIT-Verhalten, startet undefiniert oder mit letztem Wert. `E_SR_SYM_INIT` erweitert dies um eine definierte Initialisierung und Deinitialisierung.
- **E_RS**: Vertauscht Set/Reset-Priorität, aber ohne INIT-Mechanismus.
- **E_SR_SYM**: Symmetrisches SR-Flipflop ohne INIT-Schnittstelle; `E_SR_SYM_INIT` fügt die INIT-Startlogik hinzu.

Der Vorteil des `E_SR_SYM_INIT` liegt in der Kombination von qualifizierter Initialisierung mit einem expliziten Deinitialisierungspfad.

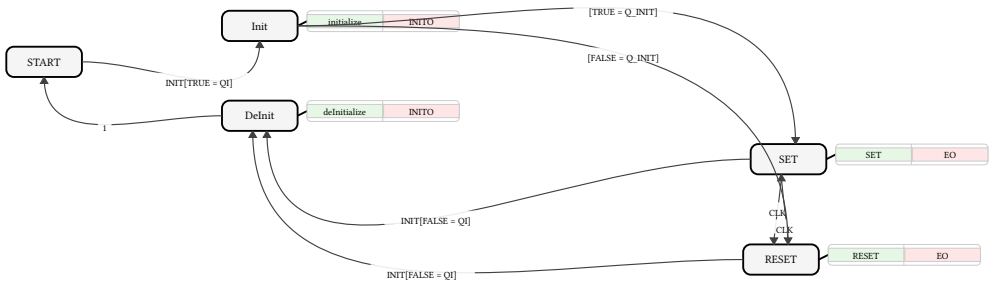
Fazit

Der Funktionsblock `E_SR_SYM_INIT` bietet ein robustes, qualifiziertes Flip-Flop mit definiertem Startzustand. Die symmetrische Startlogik und die Deinitialisierungsmöglichkeit machen ihn ideal für Anwendungen, die ein reproduzierbares Verhalten nach Systemstart oder -reset erfordern. Die Trennung von Ereignissen und Qualifikation erlaubt eine flexible Einbindung in übergeordnete Steuerungsstrukturen.

E_T_FF_INIT



ECC: E_T_FF_INIT



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **E_T_FF_INIT** realisiert ein **Toggle-Flip-Flop** mit integrierter Initialisierungslogik. Er kombiniert das klassische Verhalten eines T-Flip-Flops (Umschalten des Ausgangs *Q* bei jedem *CLK*-Ereignis) mit einer gezielten Setzung des Ausgangs während der Initialisierung. Der Baustein eignet sich besonders für Anwendungen, bei denen nach dem Start ein definierter Anfangszustand erforderlich ist, bevor der normale Toggle-Betrieb beginnt.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Mit Variablen	Beschreibung
INIT	EInit	QI, Q_INIT	Initialisierungsanforderung; setzt *Q* auf *Q_INIT*.
CLK	Event	–	Taktsignal; löst bei jeder steigenden Flanke das Umschalten von *Q* aus (sofern *QI* = TRUE).

Ereignis-Ausgänge

Ereignis	Typ	Mit Variablen	Beschreibung
INITO	EInit	QO	Initialisierungsbestätigung; signalisiert Ende der Initialisierung.

EO	Event	Q	Ausgangsereignis; wird nach jeder Zustandsänderung von *Q* ausgegeben.

Daten-Eingänge

Variable	Typ	Beschreibung
QI	BOOL	Freigabesignal für die normale Toggle-Funktion.
Q_INIT	BOOL	Gewünschter Wert von *Q* nach der Initialisierung (bei TRUE wird *Q* gesetzt).

Daten-Ausgänge

Variable	Typ	Beschreibung
QO	BOOL	Ausgangsqualifikator; wird bei INIT auf *QI* gesetzt, bei De-Initialisierung auf FALSE.
Q	BOOL	Toggle-Ausgang; wird bei jedem *CLK* umgeschaltet (nur wenn *QI* = TRUE).

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Baustein arbeitet als **zustandsgesteuerte Maschine (ECC)** mit fünf Zuständen: `START`, `Init`, `DeInit`, `SET` und `RESET`.

1. **Initialisierung (Zustand *Init*)**:
   - Wird durch das Ereignis **INIT** ausgelöst, sofern *QI* = TRUE.
   - Der Algorithmus `initialize` setzt *QO* auf den Wert von *QI* (hier TRUE).
   - Die Transition aus *Init* erfolgt abhängig vom Wert von *Q_INIT*:
     - *Q_INIT* = TRUE ⟶ Wechsel in den Zustand **SET** (*Q* wird TRUE).
     - *Q_INIT* = FALSE ⟶ Wechsel in den Zustand **RESET** (*Q* bleibt FALSE).
   - Nach der Transition wird das Ereignis **INITO** ausgegeben.
2. **De-Initialisierung (Zustand *DeInit*)**:
   - Wird durch ein **INIT**-Ereignis mit *QI* = FALSE ausgelöst (aus den Zuständen *SET* oder *RESET*).
   - Der Algorithmus `deInitialize` setzt *QO* auf FALSE.
   - Anschließend erfolgt der Übergang zurück in den Startzustand **START** und das Ereignis **INITO** wird ausgegeben.
3. **Normaler Toggle-Betrieb**:
   - In den Zuständen **SET** (*Q* = TRUE) und **RESET** (*Q* = FALSE).
   - Bei jedem **CLK**-Ereignis wird zwischen den beiden Zuständen umgeschaltet.
   - Die Algorithmen `SET` bzw. `RESET` setzen *QO* auf den aktuellen Wert von *QI* und, falls *QI* = TRUE, den Ausgang *Q* auf den entsprechenden Zustand (TRUE oder FALSE).
   - Nach jedem Wechsel wird das Ereignis **EO** ausgegeben.

Wichtig: Die Toggle-Funktion ist nur aktiv, wenn *QI* = TRUE. Ist *QI* = FALSE, bleibt *Q* unverändert (die IF-Bedingung in den Algorithmen verhindert die Änderung).

Technische Besonderheiten

- **Kombinierte Init-/Toggle-Logik**: Der Baustein erlaubt es, den Ausgang *Q* bereits während der Initialisierung auf einen definierten Wert zu setzen (*Q_INIT*). Dies unterscheidet ihn von einem einfachen T-Flip-Flop, das meist im RESET-Zustand startet.
- **Freigabe über QI**: Die Variable *QI* fungiert als globale Freigabe. Nur wenn *QI* = TRUE, erfolgt das Umschalten bei *CLK*. Bei *QI* = FALSE können zwar INIT-Ereignisse verarbeitet werden, aber der Toggle-Betrieb ist deaktiviert.
- **Ausgangsqualifikator QO**: *QO* spiegelt den Freigabezustand wider – er wird bei erfolgreicher Initialisierung auf TRUE, bei De-Initialisierung auf FALSE gesetzt. Dies ermöglicht eine einfache Überwachung des Baustein-Status.

Zustandsübersicht

Zustand	Beschreibung	Eingehende Transition(en)	Ausgehende Aktion (Algorithmus)	Ausgehendes Ereignis
START	Ruhezustand nach Initialisierung (oder nach De-Initialisierung).	*DeInit* ⟶ START	–	–
Init	Initialisierungsphase, setzt *QO* und entscheidet über *Q*.	START ⟶ Init (bei INIT & QI = TRUE)	`initialize`	INITO
DeInit	De-Initialisierungsphase, setzt *QO* auf FALSE.	SET/RESET ⟶ DeInit (bei INIT & QI = FALSE)	`deInitialize`	INITO
SET	Normaler Toggle-Zustand: *Q* = TRUE.	Init ⟶ SET (bei Q_INIT = TRUE)RESET ⟶ SET (bei CLK)	`SET`	EO
RESET	Normaler Toggle-Zustand: *Q* = FALSE.	Init ⟶ RESET (bei Q_INIT = FALSE)SET ⟶ RESET (bei CLK)	`RESET`	EO

Transitionsbedingungen (vereinfacht):

- `START → Init` : `INIT & (QI = TRUE)`
- `Init → SET` : `(Q_INIT = TRUE)`
- `Init → RESET` : `(Q_INIT = FALSE)`
- `SET ↔ RESET` : `CLK`
- `SET / RESET → DeInit` : `INIT & (QI = FALSE)`
- `DeInit → START` : immer (Bedingung = 1)

Anwendungsszenarien

1. **Initialisieren einer Maschine mit definiertem Startzustand**
   Ein Förderband soll nach dem Einschalten immer in der Position „bereit“ (*Q* = TRUE) sein. Mit *Q_INIT* = TRUE und *QI* = TRUE wird dies erreicht. Danach kann ein Taktgeber über *CLK* das Band zwischen den beiden Zuständen hin- und herschalten.
2. **Sicherheitsgesteuerter Toggle**
   Ein Sicherheitssignal *QI* schaltet den Baustein frei. Nur wenn dieses Signal aktiv ist, dürfen Toggle-Operationen durchgeführt werden. Ein Ausfall der Freigabe (*QI* = FALSE) führt zur De-Initialisierung – der Ausgang *Q* bleibt unverändert und *QO* wird FALSE.
3. **Zustandsablauf mit Initialisierungsblock**
   In einer Ablaufsteuerung wird *E_T_FF_INIT* als Flip-Flop für einen Schritt eingesetzt. Die Initialisierung erlaubt es, den Schritt direkt nach dem Start entweder aktiv oder inaktiv zu setzen, ohne einen separaten Setz- oder Rücksetzimpuls.

Vergleich mit ähnlichen Bausteinen

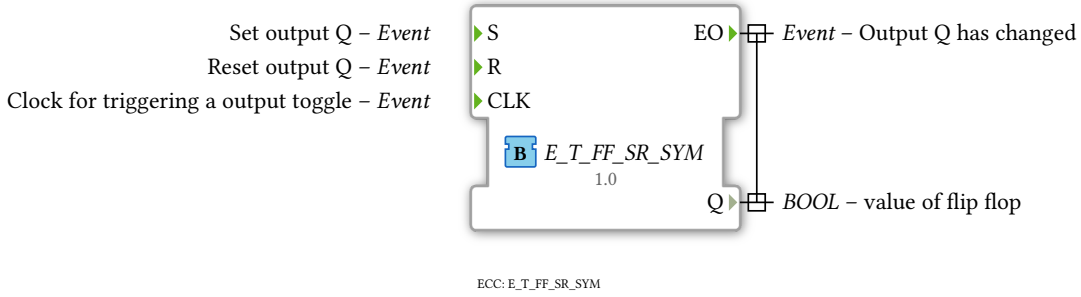
Baustein	Besonderheit
E_T_FF	Reines T-Flip-Flop ohne Initialisierungslogik. Startet stets im RESET-Zustand.
E_FF	S-R- oder D-Flip-Flop – erlaubt Setzen und Rücksetzen über separate Ereignisse.
E_SR	Speicherbaustein (Set-Reset) ohne Toggle-Eigenschaft.
E_T_FF_INIT	Kombiniert Toggle mit frei wählbarem Anfangszustand und Freigabe über *QI*.

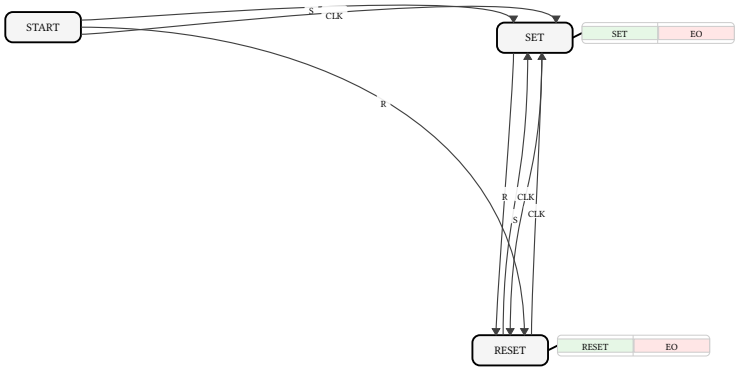
Im Gegensatz zu *E_T_FF* kann *E_T_FF_INIT* den Ausgang bei der Erstinitialisierung auf TRUE setzen und bietet über *QI* eine explizite Freigabe für den Toggle-Betrieb.

Fazit

Der Funktionsblock **E_T_FF_INIT** ist eine erweiterte Variante eines T-Flip-Flops, die speziell für Steuerungen entwickelt wurde, die nach dem Hochfahren einen definierten Ausgangszustand benötigen. Die Kombination aus Initialisierung (über *Q_INIT*), Freigabe (über *QI*) und Umschaltlogik (über *CLK*) macht ihn vielseitig einsetzbar. Er eignet sich für sicherheitskritische Abläufe, bei denen der Betrieb nur bei aktiver Freigabe erfolgen darf, sowie für Anwendungen, die einen reproduzierbaren Startzustand erfordern. Die klar strukturierte Zustandsmaschine gewährleistet ein deterministisches und vorhersagbares Verhalten.

E_T_FF_SR_SYM





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **E_T_FF_SR_SYM** ist ein ereignisgesteuerter, bistabiler Kippschalter mit integrierter Toggle-Funktion. Er vereint die Eigenschaften eines SR-Flipflops (Set-Reset) und eines Toggle-Flipflops (umschalten bei Takt). Die Besonderheit liegt im symmetrischen Startverhalten: Im Initialzustand führen sowohl Set- als auch Clock-Ereignisse zum Set-Zustand, während ein Reset-Ereignis direkt zum Reset-Zustand führt. Dies ermöglicht ein definiertes und vorhersagbares Verhalten direkt nach der Inbetriebnahme.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
S	Event	Setzen des Ausgangs Q (auf TRUE)
R	Event	Zurücksetzen des Ausgangs Q (auf FALSE)
CLK	Event	Takt zum Umschalten des Ausgangs Q (Toggle)

Ereignis-Ausgänge

Name	Typ	Mit Variable	Kommentar
EO	Event	Q	Ausgang Q hat sich geändert

Daten-Eingänge

Keine.

Daten-Ausgänge

Name	Typ	Kommentar
Q	BOOL	Wert des Flipflops (TRUE = gesetzt, FALSE = zurückgesetzt)

Adapter

Keine.

Funktionsweise

Der Baustein arbeitet als endlicher Automat mit drei Zuständen: **START**, **SET** und **RESET**.

- Im **START**-Zustand (Initialzustand) wird bei Eintreten eines **S**-Ereignisses oder eines **CLK**-Ereignisses in den **SET**-Zustand gewechselt. Bei einem **R**-Ereignis wechselt er in den **RESET**-Zustand.
- Im **SET**-Zustand führt ein **R**-Ereignis zum Wechsel in den **RESET**-Zustand, ein **CLK**-Ereignis ebenfalls. Ein **S**-Ereignis würde den Zustand halten (nicht explizit modelliert, aber durch die Ausgabeänderung nur bei Wechsel).
- Im **RESET**-Zustand führt ein **S**-Ereignis oder ein **CLK**-Ereignis in den **SET**-Zustand. Ein **R**-Ereignis würde den Zustand halten.

Bei jedem Zustandswechsel wird der entsprechende Algorithmus ausgeführt:

- **SET**: `Q := TRUE;`
- **RESET**: `Q := FALSE;`

Anschließend wird das Ereignis **EO** ausgegeben, das den neuen Wert von **Q** begleitet.

Das **Toggle**-Verhalten ist dadurch realisiert, dass ein CLK-Ereignis immer den aktuellen Zustand umschaltet – unabhängig davon, ob SET oder RESET aktiv ist.

Technische Besonderheiten

- Keine Dateneingänge: Die Steuerung erfolgt ausschließlich über Ereignisse. Dies reduziert die Anzahl der benötigten Verbindungen und vereinfacht die zeitliche Analyse.
- Symmetrisches Startverhalten: Der Initialzustand START ist nicht gleichbedeutend mit einem definierten Ausgangswert. Stattdessen wird bei jedem der drei möglichen Startereignisse (S, R, CLK) sofort ein definierter Folgezustand eingenommen. Dies vermeidet undefinierte Zustände nach dem Start.
- Alle Algorithmen sind in ST (Structured Text) implementiert und bestehen aus einfachen Zuweisungen.
- Der Ausgang Q wird nur bei einem Zustandswechsel aktualisiert – das EO-Ereignis zeigt genau diese Änderung an.

Zustandsübersicht

Zustand	Beschreibung

START	Initialzustand nach Aktivierung des Bausteins
SET	Ausgang Q = TRUE
RESET	Ausgang Q = FALSE

Übergänge (Ereignis ⟶ Zielzustand):

Ausgangszustand	Ereignis	Zielzustand
START	S	SET
START	R	RESET
START	CLK	SET
SET	R	RESET
SET	CLK	RESET
RESET	S	SET
RESET	CLK	SET

Hinweis: Verbleiben S-Ereignisse im SET-Zustand oder R-Ereignisse im RESET-Zustand werden nicht explizit abgebildet – der Zustand bleibt erhalten, und es erfolgt keine EO-Ausgabe.

Anwendungsszenarien

- **Umschaltvorgänge mit Takt**: Der Baustein eignet sich für Anwendungen, bei denen ein Binärsignal durch einen Takt umgeschaltet werden soll, aber auch ein separates Setzen oder Rücksetzen möglich sein muss (z. B. manuelle Vorrangsteuerung).
- **Symmetrische Startinitialisierung**: In sicherheitskritischen Systemen kann durch das gleichberechtigte Reagieren auf S oder CLK im Startzustand vermieden werden, dass ein undefinierter Wert vorliegt.
- **Pulsweitenmodulation oder Toggle-Sperren**: Als einfacher Toggle-Baustein mit Set/Reset-Vorrang.

Vergleich mit ähnlichen Bausteinen

Baustein	Unterschied
E_FF_SR	Reines SR-Flipflop ohne Toggle. Keine Taktfunktion.
E_FF_Toggle	Reines Toggle-Flipflop, kein Set/Reset.
E_T_FF_SR	Ähnlich, jedoch ohne symmetrisches Startverhalten. Hier führt CLK im START-Zustand ggf. zu undefiniertem Verhalten.
E_T_FF_SR_SYM	Kombiniert SR und Toggle mit definiertem Startverhalten für alle Eingangssignale.

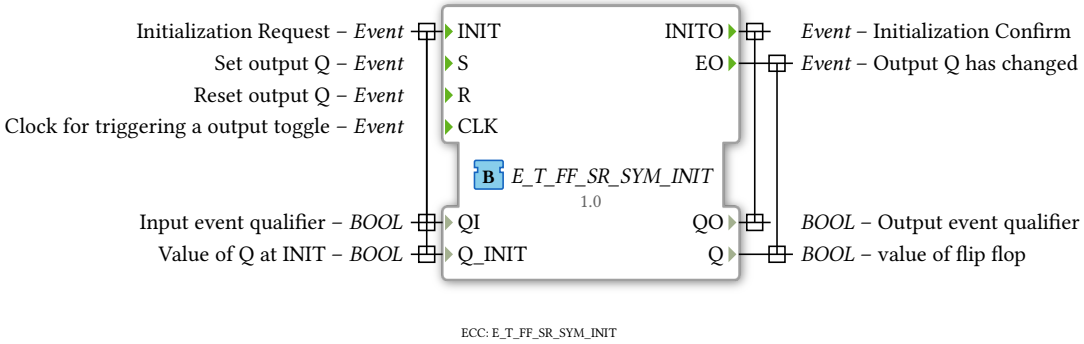
Fazit

Der **E_T_FF_SR_SYM** ist ein vielseitiger ereignisgesteuerter Flipflop-Baustein, der sowohl Set-/Reset- als auch Toggle-Funktionen vereint. Das symmetrische Startverhalten macht ihn besonders für Anwendungen geeignet, bei denen direkt nach der Aktivierung ein definierter Zustand erreicht werden muss – unabhängig davon, welches Ereignis zuerst eintrifft. Dank der reinen Ereignissteuerung lässt er sich einfach in zeitkritische Steuerungsabläufe einbinden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Das PWM-Signal & Infografik auf ms-muc-docs.de

E_T_FF_SR_SYM_INIT



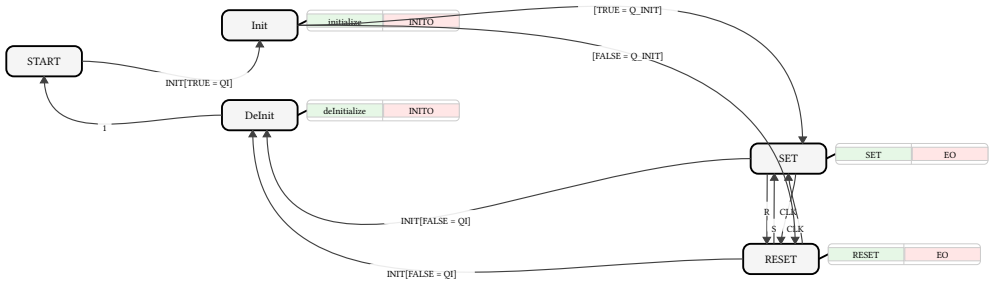


Bild des Funktionsblocks: (nicht verfügbar)

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `E_T_FF_SR_SYM_INIT` realisiert ein ereignisgesteuertes, bistabiles Flipflop mit Set-, Reset- und Toggle-Funktion. Es zeichnet sich durch ein symmetrisches Start-up-Verhalten aus: Beim Initialisierungsereignis (`INIT`) wird der Ausgang `Q` auf den am Eingang `Q_INIT` vorgegebenen Wert gesetzt, sofern der Qualifier `QI` aktiv ist. Der Baustein kombiniert die Eigenschaften eines SR-Flipflops und eines Toggle-Flipflops in einer Einheit.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Initialisierungsanforderung
S	Event	Setzt den Ausgang Q (gesetzt)
R	Event	Setzt den Ausgang Q (zurückgesetzt)
CLK	Event	Takt für das Umschalten (Toggle) von Q

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Bestätigung der Initialisierung
EO	Event	Signalisiert eine Änderung von Q

Daten-Eingänge

Name	Typ	Kommentar
QI	BOOL	Ereignisqualifier (steuert die Ausführung)
Q_INIT	BOOL	Wert, den Q bei INIT annehmen soll

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Ausgangsqualifier (Spiegel von QI)
Q	BOOL	Wert des Flipflops

Adapter

Keine.

Funktionsweise

Der Baustein durchläuft einen endlichen Automaten mit fünf Zuständen: `START`, `Init`, `DeInit`, `SET` und `RESET`.

- **Initialisierung (`INIT`)**
  Wenn am Ereigniseingang `INIT` ein Signal anliegt, wird abhängig vom Zustand des Qualifiers `QI` unterschieden:
  - Ist `QI = TRUE`, wechselt der Baustein in den Zustand `Init`. Dort wird der Ausgangsqualifier `QO` auf den Wert von `QI` gesetzt (also `TRUE`) und der Ausgang `INITO` wird ausgelöst. Anschließend geht der Automat – wiederum abhängig von `Q_INIT` – in den Zustand `SET` (falls `Q_INIT = TRUE`) oder in den Zustand `RESET` (falls `Q_INIT = FALSE`). Somit erhält `Q` den gewünschten Startwert.
  - Ist `QI = FALSE`, wechselt der Baustein in den Zustand `DeInit`. Dort wird `QO` auf `FALSE` gesetzt und `INITO` ausgelöst. Danach kehrt der Automat in den Startzustand zurück.
- **Setzen (`S`)**
  Bei einem Ereignis an `S` wird der Algorithmus `SET` ausgeführt:
  `QO` erhält den Wert von `QI`. Ist `QI = TRUE`, wird `Q` auf `TRUE` gesetzt. Der Ausgang `EO` wird ausgelöst, um eine Änderung von `Q` zu signalisieren.
- **Rücksetzen (`R`)**
  Bei einem Ereignis an `R` wird der Algorithmus `RESET` ausgeführt:
  `QO` erhält den Wert von `QI`. Ist `QI = TRUE`, wird `Q` auf `FALSE` gesetzt. Auch hier wird `EO` ausgelöst.
- **Toggle (`CLK`)**
  Ein Ereignis an `CLK` bewirkt stets einen Wechsel des Zustands:
  Befindet sich der Automat im Zustand `SET`, wechselt er zu `RESET` und umgekehrt. Dabei wird der jeweilige Algorithmus (`SET` bzw. `RESET`) mit der aktuellen Bewertung von `QI` ausgeführt. Der Ausgang `Q` wird also bei jedem Takt umgeschaltet, sofern `QI = TRUE` ist.

Technische Besonderheiten

- **Symmetrisches Start-up-Verhalten**: Der initiale Wert von `Q` wird durch den parametrierbaren Eingang `Q_INIT` festgelegt und nicht etwa auf einen festen Default gesetzt. Dadurch kann das System reproduzierbar in einen definierten Zustand starten.
- **Ereignisqualifier `QI`**: Alle Aktionen (`SET`, `RESET`) werden nur dann tatsächlich ausgeführt, wenn `QI = TRUE` ist. Bei `QI = FALSE` wird der Ausgangsqualifier `QO` lediglich auf `FALSE` gesetzt, der Wert von `Q` bleibt unverändert. Dies ermöglicht eine bedingte Steuerung durch vorgeschaltete Bausteine.

- **Deinitialisierung**: Wird während der Laufzeit ein `INIT`-Ereignis mit `QI = FALSE` empfangen, so wird der Baustein deinitialisiert und geht in den Ruhezustand (`START`). In diesem Zustand reagiert er auf keine weiteren Ereignisse, bis er erneut initialisiert wird.

Zustandsübersicht

Zustand	Beschreibung	Aktion bei Eintritt
START	Warten auf Initialisierung	–
Init	Initialisierung von QO	`QO := QI`; Auslösen von `INITO`
DeInit	Deinitialisierung (bei QI=false)	`QO := FALSE`; Auslösen von `INITO`
SET	Q ist gesetzt (`TRUE`)	`Q := TRUE` (nur wenn QI=true); Auslösen `EO`
RESET	Q ist zurückgesetzt (`FALSE`)	`Q := FALSE` (nur wenn QI=true); Auslösen `EO`

Übergänge:

- `START` ⟶ `Init` : bei `INIT` und `QI = TRUE`
- `START` ⟶ `DeInit` : bei `INIT` und `QI = FALSE`
- `Init` ⟶ `SET` : wenn `Q_INIT = TRUE`
- `Init` ⟶ `RESET` : wenn `Q_INIT = FALSE`
- `SET` ⇄ `RESET` : durch `S`, `R` oder `CLK` (jeweils unter Berücksichtigung von QI)
- `SET` ⟶ `DeInit` : bei erneuter `INIT` mit `QI = FALSE`
- `RESET` ⟶ `DeInit` : bei erneuter `INIT` mit `QI = FALSE`
- `DeInit` ⟶ `START` : immer (Transition mit Bedingung `1`)

Anwendungsszenarien

- **Steuerung mit definiertem Startzustand**: In Sicherheitssteuerungen oder Anlagen, die nach einem Neustart einen bestimmten Ausgangszustand benötigen, kann `Q_INIT` entsprechend gesetzt werden.
- **Toggle-Funktion für Befehlsschalter**: Ein einziger Taster (verbunden mit `CLK`) schaltet einen Ausgang bei jedem Drücken um – z. B. für Lichtsteuerungen.
- **Bedingte Set-/Reset-Logik**: Der Qualifier `QI` erlaubt es, die Aktionen von übergeordneten Bedingungen abhängig zu machen, ohne dass der Zustand des Flipflops verloren geht.

Vergleich mit ähnlichen Bausteinen

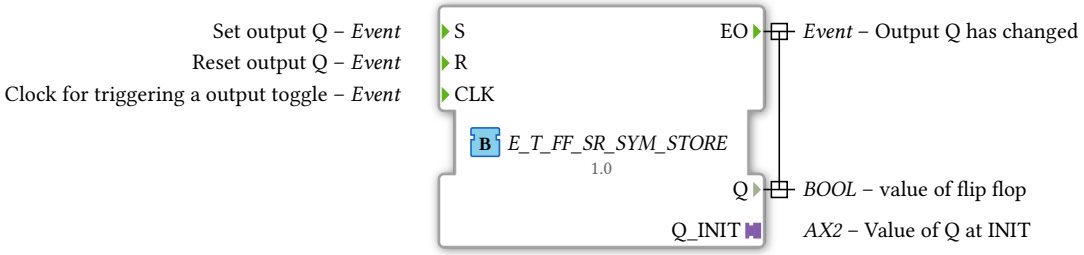
- **Standard SR-Flipflop (`E_SR`)**: Bietet nur Setzen und Rücksetzen, keine Toggle-Funktion und kein einstellbarer Startwert.
- **Toggle-Flipflop (`E_TOGGLE`)**: Nur Umschalten bei jedem Takt, kein separates Setzen/Rücksetzen, kein definierter Startwert.
- **SR-Flipflop mit Initialisierung (`E_SR_INIT`)**: Ähnlich, aber ohne Toggle-Funktion.

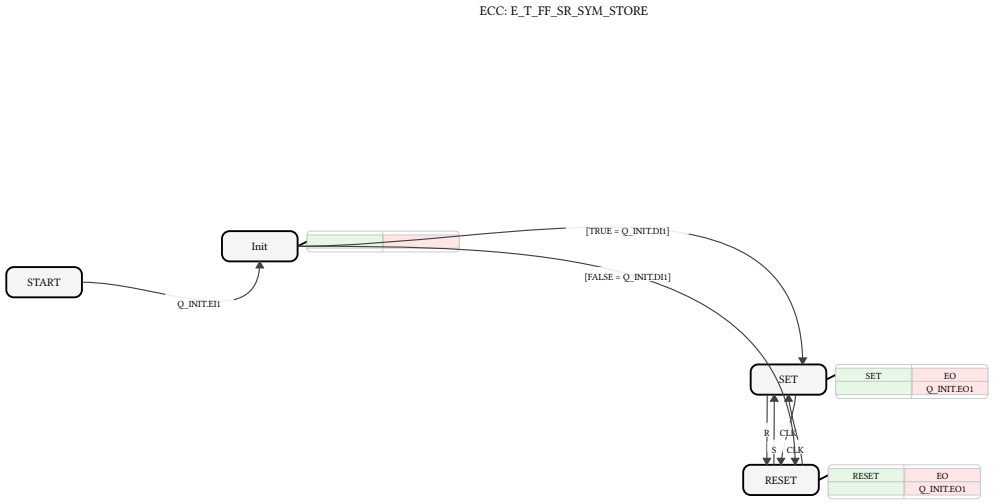
Der vorliegende Baustein vereint alle drei Funktionen (Set, Reset, Toggle) mit einem flexiblen Initialisierungsverhalten – eine umfassende Lösung für viele ereignisgesteuerte Steuerungsaufgaben.

Fazit

`E_T_FF_SR_SYM_INIT` ist ein vielseitiger Funktionsblock für die IEC-61499-basierte ereignisgesteuerte Automatisierung. Durch die Kombination von Setzen, Rücksetzen und Toggeln sowie der parametrierbaren Initialisierung deckt er ein breites Spektrum an Anforderungen ab. Die Verwendung des Qualifiers `QI` erhöht die Flexibilität und ermöglicht eine sichere Einbindung in komplexe Ablaufsteuerungen.

E_T_FF_SR_SYM_STORE





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `E_T_FF_SR_SYM_STORE` realisiert ein ereignisgesteuertes bistabiles Kippglied (Flipflop) mit den Funktionen **Setzen** (S), **Rücksetzen** (R) und **Toggeln** (CLK). Das besondere Merkmal ist das **symmetrische Start-Up-Verhalten**: Der Initialzustand des Ausgangs Q wird über einen Adapter (`Q_INIT`) beim Hochfahren oder nach einem `INIT`-Ereignis festgelegt. Dadurch verhält sich der Baustein zu jedem Startzeitpunkt deterministisch und wiederholbar – eine wichtige Eigenschaft für sicherheitskritische oder vorhersagbare Automatisierungssysteme.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
`S`	Event	Setzen des Ausgangs Q auf TRUE
`R`	Event	Rücksetzen des Ausgangs Q auf FALSE
`CLK`	Event	Toggle – wechselt Q (TRUE ⟶ FALSE oder FALSE ⟶ TRUE)

Ereignis-Ausgänge

Name	Typ	Mit Daten	Kommentar
`EO`	Event	Q	Zeigt an, dass sich der Wert von Q geändert hat

Daten-Eingänge

Keine (alle Eingaben erfolgen über Ereignisse).

Daten-Ausgänge

Name	Typ	Kommentar
`Q`	BOOL	Aktueller Wert des Flipflops (TRUE oder FALSE)

Adapter

Name	Typ	Kommentar
Q_INIT	adapter::types::bidirectional::AX2	Legt den Startwert von Q beim INIT-Ereignis fest. Der Adapter stellt einen ereignisgesteuerten bidirektionalen Kanal bereit: Q_INIT.EI1 löst die Initialisierung aus, Q_INIT.DI1 liefert den gewünschten Initialwert (TRUE oder FALSE). Zusätzlich gibt der Baustein über Q_INIT.DO1 den gesetzten Wert zurück.

Funktionsweise

Der interne Ablauf wird durch die Endliche Zustandsmaschine (ECC) gesteuert:

1. **START-Zustand:** Nach Spannungsversorgung oder System-Reset befindet sich der Baustein im Zustand START.
2. **Initialisierung (Zustand Init):** Sobald das Ereignis Q_INIT.EI1 eintrifft, wird der Zustand Init erreicht. Anschließend wird anhand des booleschen Wertes von Q_INIT.DI1 entschieden:
   - TRUE → Übergang in den Zustand SET
   - FALSE → Übergang in den Zustand RESET
3. **Betriebs-Zustände SET und RESET:**
   - **SET:** Setzt Q := TRUE und Q_INIT.DO1 := TRUE. Sendet Ausgangsereignis EO und Adapter-Ausgangsereignis Q_INIT.EO1.
   - **RESET:** Setzt Q := FALSE und Q_INIT.DO1 := FALSE. Sendet ebenfalls EO und Q_INIT.EO1.
4. **Umschalten zwischen SET und RESET:**
   - Ereignis S (wenn im Zustand RESET) → wechselt in SET.
   - Ereignis R (wenn im Zustand SET) → wechselt in RESET.
   - Ereignis CLK (immer) → toggelt zwischen den beiden Zuständen.

Der Ausgang EO wird bei jeder Zustandsänderung von Q aktiviert.

Technische Besonderheiten

- **Symmetrisches Start-Up:** Anders als einfache SR-Flipflops mit festem Startwert (z. B. FALSE) kann der Startwert über den Adapter konfiguriert werden. Dies erhöht die Flexibilität und Reproduzierbarkeit.
- **Toggle-Funktion:** Der CLK-Eingang erlaubt ein Wechseln des Zustands ohne Kenntnis des aktuellen Wertes – nützlich für Zähler, Blinkgeber oder Zustandsautomaten.
- **Bidirektionale Adapter-Schnittstelle:** Der Baustein meldet seinen gesetzten Wert zurück an den Adapter (Q_INIT.DO1), sodass der Initialisierungsgeber den endgültigen Zustand bestätigen kann.
- **Priorisierung:** Bei gleichzeitigem Eintreffen von S, R oder CLK wird nur das zuerst verarbeitete Ereignis berücksichtigt (basierend auf der Event-Reihenfolge in der 4diac-IDE). Im konkreten Entwurf wird CLK als Toggle auf den aktuellen Zustand angewendet, während S und R unabhängig vom Toggle den Zustand setzen. Die Zustandsübergänge zeigen, dass CLK aus jedem Zustand in den jeweils anderen führt.

Zustandsübersicht

Die Zustandsmaschine (ECC) umfasst vier Zustände:

Zustand	Beschreibung	Ausgehende Transitionen
START	Initialer Ruhezustand nach dem Einschalten	→ Init bei Q_INIT.EI1
Init	Initialisierungs-Zustand – entscheidet über Startwert	→ SET bei TRUE = Q_INIT.DI1→ RESET bei FALSE = Q_INIT.DI1
SET	Q = TRUE	→ RESET bei R oder CLK
RESET	Q = FALSE	→ SET bei S oder CLK

Nach dem Verlassen von Init wird der Zustand SET oder RESET erreicht; die Zustände START und Init werden im Normalbetrieb nicht mehr durchlaufen.

Anwendungsszenarien

- **Maschinensteuerung mit definiertem Startzustand:** Eine Anlage soll nach einem Reset immer in einem bestimmten Arbeitsmodus starten. Über `Q_INIT` kann z. B. über eine SPS-Konfiguration vorgegeben werden, ob eine Ventilposition beim Start geöffnet oder geschlossen ist.
- **Toggeln als Blinkfunktion:** Der `CLK`-Eingang kann periodisch getriggert werden (z. B. durch einen Timer-Baustein), um eine blinkende Signalisierung zu realisieren.
- **Redundante Rückmeldung:** Der Adapter-Ausgang `Q_INIT.DO1` liefert den gesetzten Wert zurück – ideal für Sicherheitsfunktionen, die eine Rückmeldung des Schaltzustandes benötigen.

Vergleich mit ähnlichen Bausteinen

Baustein	Eigenschaften	Unterschied zu `E_T_FF_SR_SYM_STORE`
`E_SR_FF` (Standard-SR-Flipflop)	Setzen, Rücksetzen; meist kein Toggle, kein konfigurierbarer Startwert	Fehlende Toggle-Funktion; Startwert oft fest auf FALSE
`E_RS_FF`	Setzen priorisiert	Andere Priorisierung; kein Toggle
`E_D_FF` (D-Flipflop)	Übernahme eines Dateneingangs mit Takt	Kein Setzen/Rücksetzen ohne Daten; kein symmetrischer Start
`E_CTU` (Zähler)	Zählen von Ereignissen	kein bistabiles Verhalten, sondern Zählerstand

Der `E_T_FF_SR_SYM_STORE` vereint SR-Funktion, Toggle und initialisierten Startwert in einem Baustein – eine seltene Kombination in der IEC 61499-Welt.

Fazit

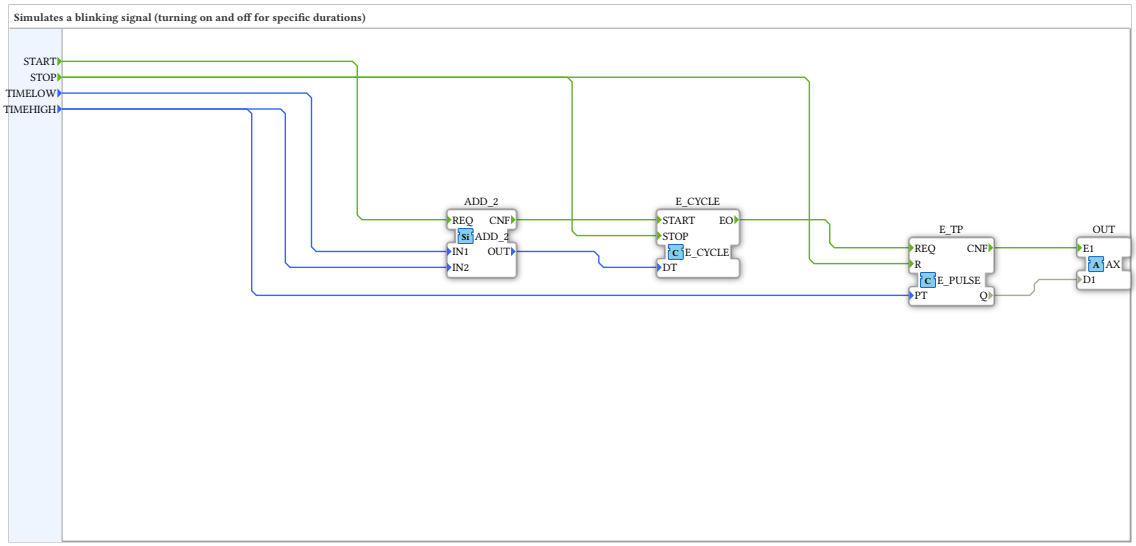
Der **E_T_FF_SR_SYM_STORE** ist ein leistungsfähiger ereignisgesteuerter Flipflop-Baustein für die 4diac-IDE. Er bietet eine vollständige SR-Funktion, eine Toggle-Option und ein symmetrisches, über einen Adapter konfigurierbares Start-Up-Verhalten. Dies macht ihn besonders für Anwendungen geeignet, die deterministische Initialisierung und flexible Umschaltmechanismen erfordern. Die bidirektionale Adapter-Schnittstelle erhöht die Integrationstiefe in komplexe Steuerungsarchitekturen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

signals

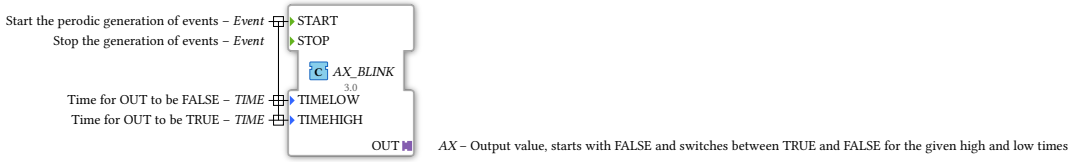
AX_BLINK



• – ○ • – ○ • – ○ •

Einleitung

Der AX_BLINK ist ein Funktionsblock, der ein blinkendes Signal über einen AX-Adapter ausgibt.



Schnittstellenstruktur

Ereignis-Eingänge

- **START**: Startet die periodische Erzeugung von Ereignissen; trägt `TIMELOW` und `TIMEHIGH`.
- **STOP**: Stoppt die Erzeugung von Ereignissen.

Daten-Eingänge

- **TIMELOW** (TIME): Zeit, in der OUT FALSE ist.
- **TIMEHIGH** (TIME): Zeit, in der OUT TRUE ist.

Adapter

Plugs (Ausgänge):

- **OUT** (adapter::types::unidirectional::AX): Ausgabewert, beginnt mit FALSE und wechselt zwischen TRUE und FALSE entsprechend den vorgegebenen High- und Low-Zeiten.

Funktionsweise

Der Baustein schaltet den Ausgang OUT nach `START` abwechselnd für die Zeit `TIMELOW` aus (FALSE) und für die Zeit `TIMEHIGH` ein (TRUE), bis `STOP` eintrifft.

Technische Besonderheiten

- Verwendet unidirektionale Adapter.

Zustandsübersicht

Zustandsbasiert (Blinken).

Anwendungsszenarien

Blinkende Anzeigen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **E_BLINK**: Standard Blink-Baustein.

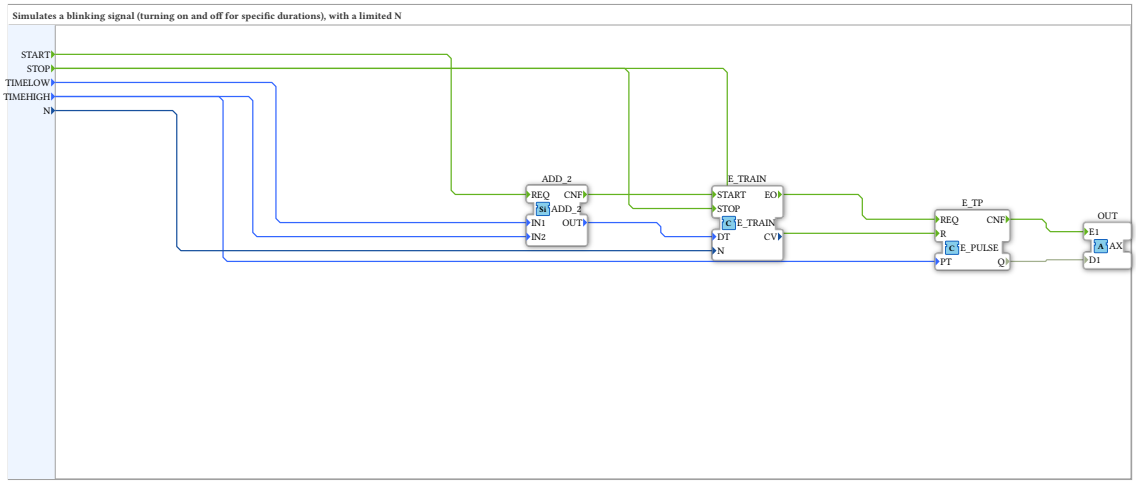
🛠️ Zugehörige Übungen

- Uebung_020f3_AX

Fazit

Adapter-basierter Blink-Baustein.

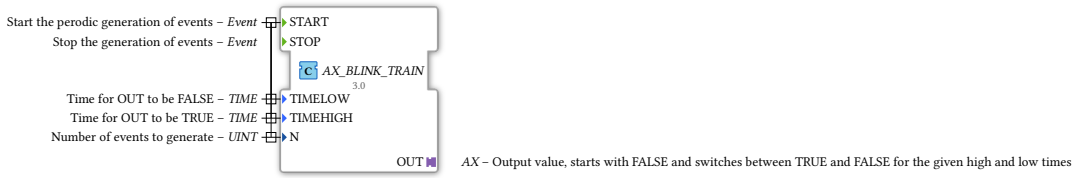
AX_BLINK_TRAIN



• — ○ • — ○ • — ○ •

Einleitung

Der AX_BLINK_TRAIN ist ein Funktionsblock, der ein Blinksignal mit begrenzter Anzahl N von Ereignissen (definierte Ein- und Ausschaltzeiten) über einen AX-Adapter ausgibt.



Schnittstellenstruktur

Ereignis-Eingänge

- **START** (Event): Startet die periodische Erzeugung der Ereignisse (mit TIMELOW, TIMEHIGH, N).
- **STOP** (Event): Stoppt die Erzeugung.

Daten-Eingänge

- **TIMELOW** (TIME): Zeit, in der OUT FALSE ist.
- **TIMEHIGH** (TIME): Zeit, in der OUT TRUE ist.
- **N** (UINT): Anzahl der zu erzeugenden Ereignisse.

Adapter

Plugs (Ausgänge):

- **OUT** (adapter::types::unidirectional::AX): Ausgangswert, beginnt mit FALSE und wechselt für die angegebenen High- und Low-Zeiten zwischen TRUE und FALSE.

Funktionsweise

Nach `START` gibt der Baustein am Adapter OUT ein Blinksignal aus: Der Ausgang beginnt mit FALSE und ist jeweils für TIMEHIGH TRUE und für TIMELOW FALSE. Es werden N Ereignisse erzeugt; `STOP` beendet die Erzeugung.

Technische Besonderheiten

- Verwendet unidirektionale Adapter.

Zustandsübersicht

Zustandsbasiert.

Anwendungsszenarien

Bahnübergänge, Warnsignale.

⚖️ Vergleich mit ähnlichen Bausteinen

- Keine direkte Standard-Entsprechung.

🛠️ Zugehörige Übungen

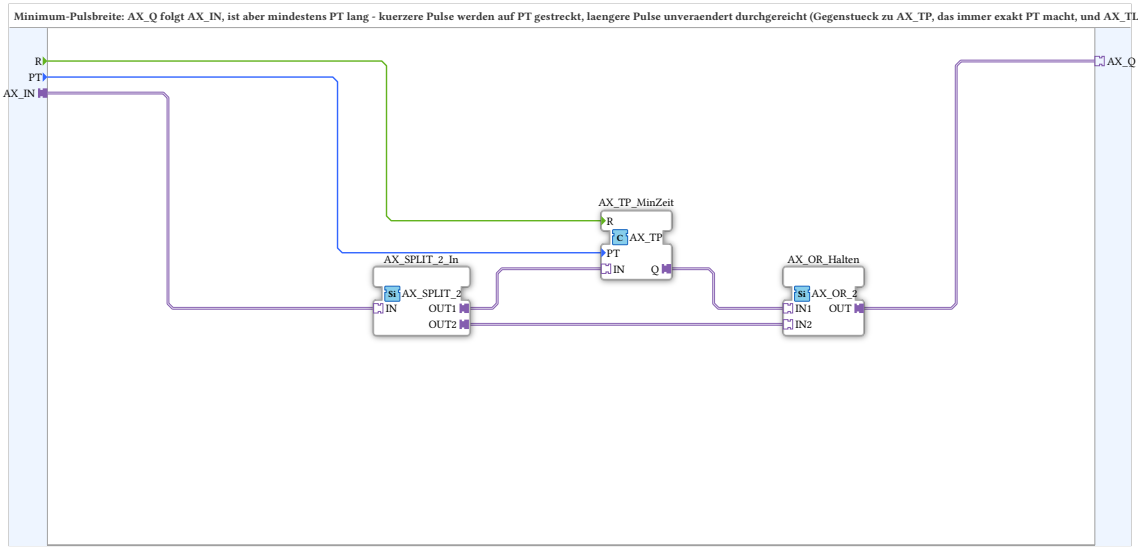
- Uebung_035a3_AX

Fazit

Adapter-basierter Zug-Blink-Baustein.

timers

AX_MIN_TP

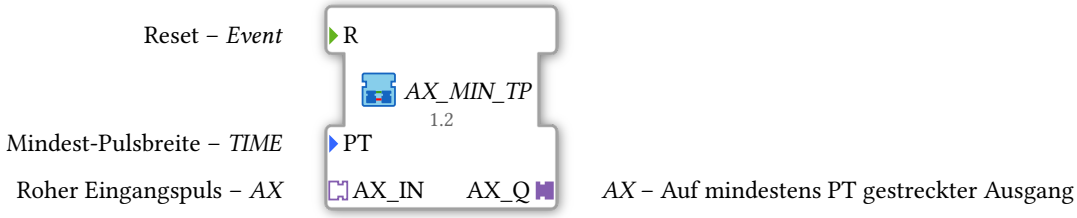


• — ○ • — ○ • — ○ •

Einleitung

`AX_MIN_TP` ist ein Mindest-Pulsbreiten-Baustein: `AX_Q` folgt `AX_IN`, ist aber **mindestens `PT` lang**. Kürzere Pulse werden auf `PT` gestreckt, längere Pulse unverändert durchgereicht. Er ist das Gegenstück zu `AX_TP`, das immer exakt `PT` ausgibt, und zu `AX_TLIM`, das auf maximal `PT` kappt. Anwendungsfall: Der Bunkerboden-Vorlauf im Automatikbetrieb braucht eine Mindestzeit.

Sicherheitshinweis (Martin Schuder): Der Baustein selbst hat **keinen Hard-Abbruch**. Ein Endschalter, der den Puls jederzeit sofort unterbrechen soll, muss extern per `AX_AND(AX_Q, NOT Endschalter)` **nach** diesem Baustein verknüpft werden.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: AX_MIN_TP

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **AX_SPLIT_2_In**: `adapter::events::unidirectional::AX_SPLIT_2` – `AX_IN` wird für `AX_TP_MinZeit.IN` **und** direkt für `AX_OR_Halten` gebraucht
  - **AX_TP_MinZeit**: `adapter::events::unidirectional::timers::AX_TP` – non-retriggerable: läuft nach der steigenden Flanke von `AX_IN` immer exakt `PT` durch, unabhängig davon, ob `AX_IN` vorher schon wieder abfällt
  - **AX_OR_Halten**: `adapter::booleanOperators::AX_OR_2` – `AX_IN OR AX_TP_MinZeit.Q`: hält den Ausgang, solange entweder der rohe Puls noch da ist (längere Pulse) oder die Mindestzeit noch läuft (kürzere Pulse)

Schnittstellenstruktur

Ereignis-Eingänge

- **R**: Reset (an `AX_TP_MinZeit.R` durchgereicht)

Ereignis-Ausgänge

Keine – Ereignisse laufen über die Adapter.

Daten-Eingänge

- **PT** (`TIME`): Mindest-Pulsbreite

Daten-Ausgänge

Keine.

Adapter

- **AX_IN** (Socket, `adapter::types::unidirectional::AX`): Roher Eingangspuls
- **AX_Q** (Plug, `adapter::types::unidirectional::AX`): Auf mindestens `PT` gestreckter Ausgang

Programmablauf und Verbindungen

- `AX_IN` ⟶ `AX_SPLIT_2_In.IN`
- `AX_SPLIT_2_In.OUT1` ⟶ `AX_TP_MinZeit.IN`
- `AX_SPLIT_2_In.OUT2` ⟶ `AX_OR_Halten.IN2`
- `AX_TP_MinZeit.Q` ⟶ `AX_OR_Halten.IN1`
- `AX_OR_Halten.OUT` ⟶ `AX_Q`
- `PT` ⟶ `AX_TP_MinZeit.PT`; `R` ⟶ `AX_TP_MinZeit.R`

Technische Besonderheiten

- Kein Latch und kein `AX_TON` nötig: `AX_TP` ist non-retriggerable, daher gilt `AX_Q = AX_IN OR AX_TP_MinZeit.Q`
- Kürzere Pulse werden gestreckt (`AX_TP.Q` hält bis `PT`), längere bleiben unverändert (`AX_IN` hält `Q` hoch, auch nachdem `AX_TP.Q` abgelaufen ist)
- Version 1.2 (2026-09-29): Vereinfachung gegenüber v1.1 (`ACTIVE`/`AX_TON`/`AX_R_TRIG`-Latch-Konstruktion, verworfen); neuer Ereigniseingang `R`

Anwendungsszenarien

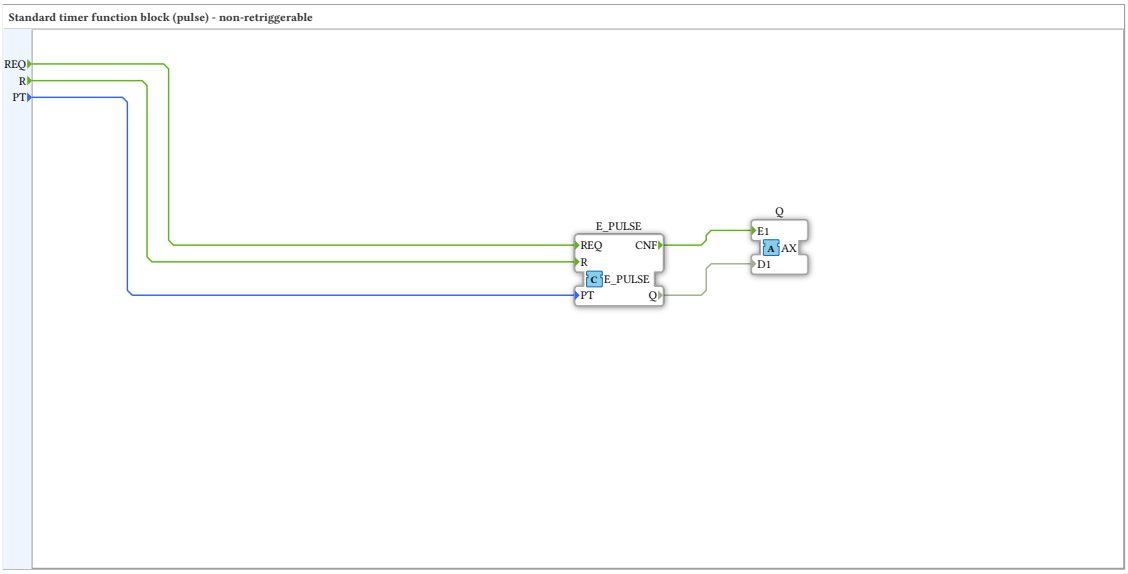
- Mindestlaufzeit für einen Antrieb (z. B. Bunkerboden-Vorlauf im Automatikbetrieb)

- Entprellen/Verlängern zu kurzer Impulse

Zusammenfassung

`AX_MIN_TP` gibt einen `AX`-Puls mit mindestens `PT` Länge aus, indem es den Eingang mit einem non-retriggerable `AX_TP` ODER-verknüpft.

AX_PULSE

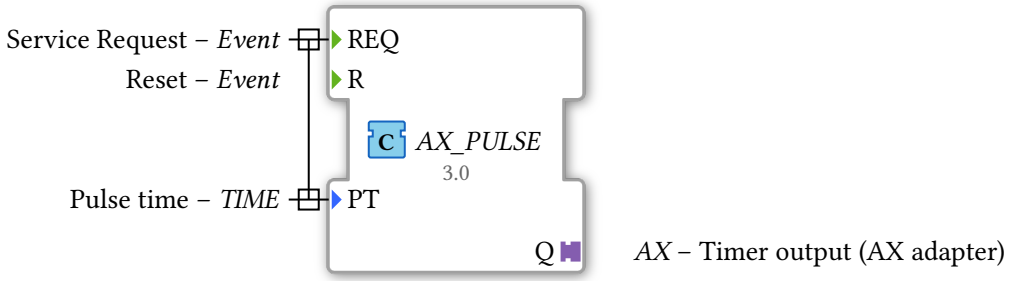


• – ◦ • – ◦ • – ◦ •

Einleitung

Wichtiger Hinweis: Dieser Baustein benötigt nur ein Ereignis (Event) und keine zyklischen Aufrufe. Er besitzt keinen Ausgang ET und zeigt die verstrichene Zeit nicht an.

Der AX_PULSE ist ein Funktionsblock, der einen Impuls über einen AX-Adapter ausgibt.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Event): Service-Anforderung (mit `PT`); löst den Impuls aus.
- **R** (Event): Reset.

Daten-Eingänge

- **PT** (TIME): Impulsdauer.

Adapter

Plugs (Ausgänge):

- **Q** (adapter::types::unidirectional::AX): Timer-Ausgang (AX-Adapter), liefert den Impuls.

Funktionsweise

Bei REQ wird Q für die Dauer PT aktiv (nicht nachtriggerbar); R setzt den Impuls zurück. Intern wird ein E_PULSE verwendet, dessen CNF/Q auf den Adapter Q geführt werden.

Technische Besonderheiten

- Verwendet unidirektionale Adapter.

Zustandsübersicht

Impuls.

Anwendungsszenarien

Signalisierung.

⚖️ Vergleich mit ähnlichen Bausteinen

- **E_PULSE**

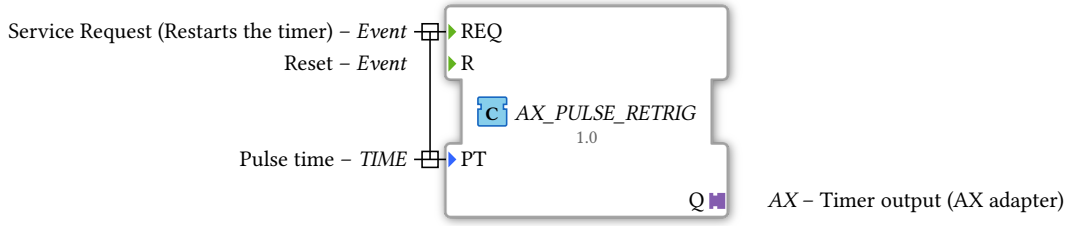
🛠️ Zugehörige Übungen

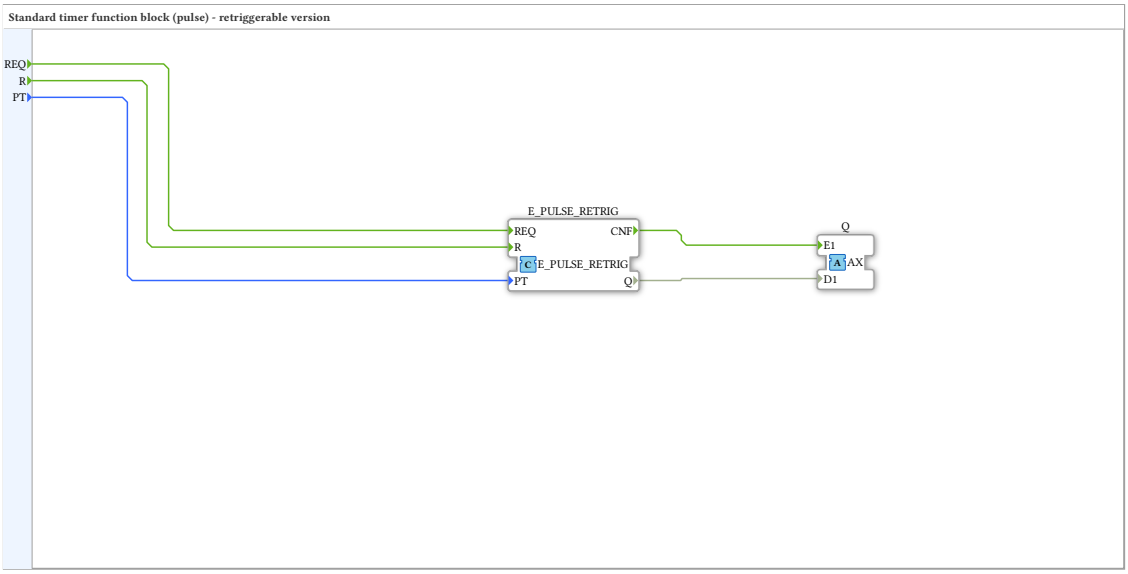
- Uebung_020h_AX
- Uebung_020i_AX

Fazit

Adapter-basierter Impuls-Baustein.

AX_PULSE_RETRIG





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AX_PULSE_RETRIG** ist ein nachtriggerbarer (retriggerbarer) Impulsformer mit AX-Adapter-Schnittstelle. Er erzeugt an seinem Ausgang einen Impuls mit einer vorgegebenen Dauer PT. Ein erneuter Startbefehl innerhalb der Impulsdauer setzt die Timer-Zeit zurück und verlängert den

Impuls. Der Block wird typischerweise eingesetzt, um zeitlich exakte, verlängerbare Ausgangssignale in Automatisierungssystemen zu generieren.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung
REQ	EVENT	Service-Anforderung zum Starten oder Neustarten des Timers (mit PT verbunden)
R	EVENT	Zurücksetzen des Timers (Impuls sofort beenden)

Ereignis-Ausgänge

Der Baustein besitzt keine direkten Ereignis-Ausgänge. Ausgangssignale werden über den Adapter Q bereitgestellt.

Daten-Eingänge

Name	Typ	Beschreibung
PT	TIME	Impulsdauer (Pulszeit)

Daten-Ausgänge

Der Baustein besitzt keine direkten Daten-Ausgänge. Ausgangsdaten werden über den Adapter Q bereitgestellt.

Adapter

Name	Typ	Beschreibung
Q	`adapter::types::unidirectional::AX`	Adapter-Schnittstelle, die den Impulsstatus (Ereignis und Daten) nach außen führt

Funktionsweise

- Der Funktionsblock kapselt intern den Standardbaustein `E_PULSE_RETRIG` (gemäß IEC 61499).
- Ein **REQ**-Ereignis startet den Timer: Der Ausgang Q (über Adapter) wird aktiv und der Impuls beginnt.
- Falls während der Impulsdauer erneut **REQ** eintrifft, wird der Timer zurückgesetzt und die Impulsdauer beginnt von Neuem (Retrigger-Funktion).
- Ein **R**-Ereignis unterbricht den Impuls sofort – der Ausgang Q wird inaktiv.
- Der Adapter Q gibt den aktuellen Impulszustand als Ereignis (über `Q.E1`) und als Datenwert (über `Q.D1`) weiter.

Technische Besonderheiten

- Nutzung des standardisierten `E_PULSE_RETRIG`-Bausteins als interne Implementierung.
- Die Ausgabe erfolgt ausschließlich über einen AX-Adapter, der sowohl ein Ereignissignal (bei Zustandsänderung) als auch den binären Impulszustand (Typ BOOL) bereitstellt.
- Ein erneuter Trigger führt zu einem Neustart der Zeitmessung – ideal für Taster mit Nachlauf oder pulsbreitenmodulierte Signale.
- Die Impulsdauer PT kann während der Laufzeit geändert werden, wirkt sich aber erst beim nächsten REQ aus.

Zustandsübersicht

Der Baustein bildet die Zustände des internen `E_PULSE_RETRIG` ab:

Zustand	Beschreibung
IDLE	Kein Impuls aktiv; Ausgang Q = FALSE
PULSE	Impuls läuft; Ausgang Q = TRUE; Timer läuft. Ein REQ-Ereignis führt zurück in PULSE mit neuem Timerstart.

- Übergang IDLE ⟶ PULSE durch REQ.
- Übergang PULSE ⟶ IDLE durch Ablauf der PT-Zeit oder durch Reset R.

Anwendungsszenarien

- **Verlängerbare Einschaltimpulse**: Wenn ein Sensor mehrfach innerhalb der Impulsdauer auslöst, soll der Ausgang dauerhaft aktiv bleiben (z. B. bei Füllstandsüberwachung).
- **Taktgeber mit nachtriggerbarer Pulsbreite**: Steuerung von Ventilen oder Lampen, die bei wiederholter Anforderung erneut für eine feste Zeit eingeschaltet werden.
- **Sicherheitsfunktionen**: Überwachung von Maschinen, bei denen ein erneuter Startbefehl die Stillstandszeit zurücksetzt.

Vergleich mit ähnlichen Bausteinen

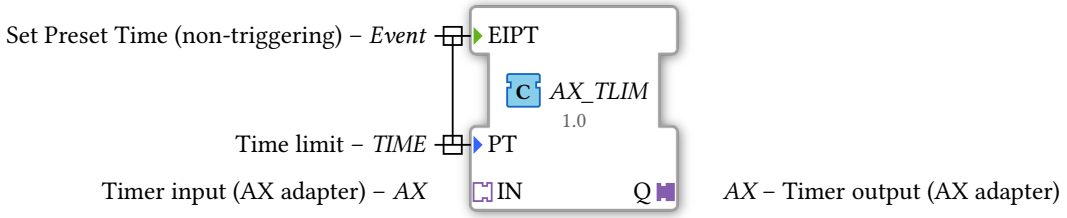
Baustein	Eigenschaft
E_PULSE	Einfacher Impuls ohne Retrigger – einmaliger Timerstart, kein Neustart bei erneutem REQ.
AX_PULSE_RETRIG	Retriggerbare Version mit AX-Adapter (Ereignis + Daten).
TON / TOF	Verzögerungsbausteine (Einschaltverzögerung / Ausschaltverzögerung) – kein Impulsformer.

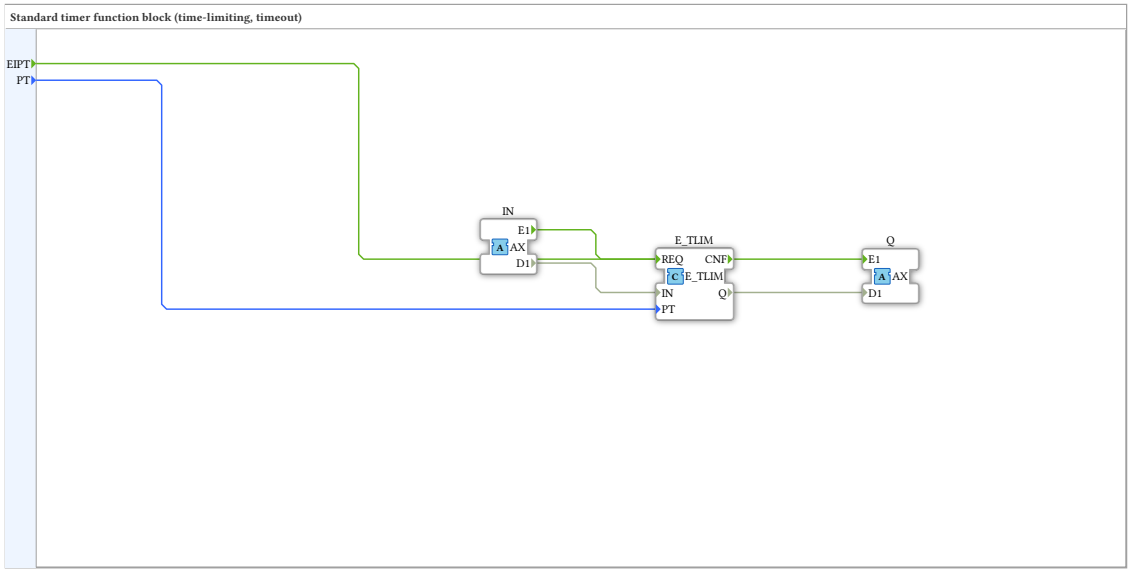
Der AX_PULSE_RETRIG bietet damit die Flexibilität eines nachtriggerbaren Impulses bei gleichzeitig standardisierter Adapter-Schnittstelle für die einfache Einbindung in größere Funktionsbaustein-Netzwerke.

Fazit

Der **AX_PULSE_RETRIG** ist ein kompakter, leistungsfähiger Funktionsblock zur Erzeugung von nachtriggerbaren Impulsen. Dank der AX-Adapter-Schnittstelle lässt er sich nahtlos mit anderen Bausteinen verbinden und eignet sich besonders für Anwendungen, bei denen ein Ausgangssignal bei wiederholten Anforderungen verlängert werden muss. Die Verwendung des etablierten `E_PULSE_RETRIG`-Bausteins gewährleistet standardkonformes und zuverlässiges Zeitverhalten.

AX_TLIM





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AX_TLIM** ist ein zeitbegrenzender Timer (Time-Limiting) gemäß IEC 61499-2. Er überwacht ein boolesches Eingangssignal, das über einen unidirektionalen **AX-Adapter** bereitgestellt wird. Solange das Signal aktiv (TRUE) ist, bleibt der Ausgang ebenfalls aktiv – jedoch maximal für eine vorgegebene Zeit. Überschreitet die Aktivierungsdauer den Grenzwert, wird der Ausgang zurückgesetzt (Timeout). Ein separater Ereigniseingang erlaubt das Voreinstellen der Zeitbegrenzung, ohne eine neue Ausführung auszulösen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung
`EIPT`	Event	Setzt die Zeitbegrenzung `PT` neu (löst keine Ausführung aus)

Ereignis-Ausgänge

Der Funktionsblock besitzt keine direkten Ereignis-Ausgänge. Stattdessen werden das Ausgangsereignis und der Ausgangswert über den **Adapter `Q`** bereitgestellt (siehe Abschnitt Adapter).

Daten-Eingänge

Name	Typ	Beschreibung
`PT`	`TIME`	Zeitbegrenzung (Preset Time) für den Timer

Daten-Ausgänge

Direkte Daten-Ausgänge sind nicht vorhanden. Der aktuelle Zustand des Timers wird über den **Adapter `Q`** ausgegeben.

Adapter

Name	Typ	Richtung	Beschreibung
`IN`	`AX`	Socket	Eingangsadapter: empfängt (über Ereignis `E1` und Datum `D1`) den zu überwachenden booleschen Wert
`Q`	`AX`	Plug	Ausgangsadapter: sendet (über Ereignis `E1` und Datum `D1`) den Timer-Zustand (`TRUE` = aktiv, `FALSE` = inaktiv oder Timeout)

Anmerkung: Der Adaptertyp AX ist ein unidirektionaler Standardadapter, bestehend aus einem Ereignis (E1) und einem booleschen Datum (D1).

Funktionsweise

1. **Initialzustand** – Die Ausgangsvariable Q.D1 ist FALSE, der Timer läuft nicht.
2. **Aktivierung** – Sobald der Eingangsadapter IN ein Ereignis (E1) mit dem Datum TRUE liefert, wird sofort Q.D1 auf TRUE gesetzt und ein interner Timer gestartet.
3. **Zeitüberwachung** – Der Timer zählt die Zeit, die IN.D1 auf TRUE bleibt. Der Timer verwendet den zuletzt via EIPT gesetzten Wert PT.
   - *Fall A:* IN.D1 wird vor Ablauf von PT wieder FALSE ⟶ dann wird Q.D1 sofort auf FALSE gesetzt.
   - *Fall B:* IN.D1 bleibt länger als PT auf TRUE ⟶ nach Ablauf von PT wird Q.D1 auf FALSE gesetzt (Timeout).
4. **Ereignisausgabe** – Jede Änderung von Q.D1 wird über den Ausgangsadapter Q.E1 signalisiert.
5. **Voreinstellung** – Das Ereignis EIPT setzt die Zeitbegrenzung PT neu, ohne die aktuelle Timer-Ausführung zu beeinflussen. Erst die nächste Aktivierung des Timers (durch IN.E1 mit TRUE) verwendet den neuen Wert.

Technische Besonderheiten

- Der Baustein ist als **Composite-FB** realisiert; intern nutzt er den Standard-FB E_TLIM (aus iec61499::events::timers). Die zugrunde liegende Logik ist identisch, jedoch wird die Schnittstelle über unidirektionale **AX-Adapter** bereitgestellt.
- Die Verwendung eines Adapter-Sockets (IN) und eines Adapter-Plugs (Q) ermöglicht eine saubere Trennung von Ereignis- und Datenkommunikation und vereinfacht die Anbindung an andere Bausteine mit kompatiblen Adaptern.
- Der Timer wird **flankengesteuert** ausgelöst: Nur eine steigende Flanke auf IN.D1 startet einen neuen Timer-Lauf. Ein erneutes Ereignis auf IN.E1 mit TRUE während eines laufenden Timers setzt den Timer nicht zurück (es sei denn, der Wert geht kurz auf FALSE und wieder auf TRUE).

Zustandsübersicht

Der Funktionsblock durchläuft folgende logische Zustände (nicht als explizite Zustandsmaschine im XML abgebildet):

Zustand	Bedingung IN.D1	Ausgang Q.D1	Timer läuft	Beschreibung
Inaktiv	FALSE	FALSE	nein	Warten auf Aktivierung
Aktiv	TRUE	TRUE	ja	Timer zählt, Ausgang aktiv
Timeout	TRUE	FALSE	nein	Zeitüberschreitung, Timer abgelaufen, Ausgang inaktiv
Rückkehr	FALSE	FALSE	nein	Eingang geht vor Timeout zurück, Ausgang sofort inaktiv

Ein Wechsel von **Aktiv** nach **Timeout** erfolgt, wenn die verbleibende Timer-Zeit null erreicht. Ein Wechsel von **Aktiv** nach **Rückkehr** erfolgt, wenn IN.D1 vor Ablauf des Timers auf FALSE wechselt.

Anwendungsszenarien

- **Überwachung von Aktoren** – z. B. Prüfung, ob ein Ventil nicht länger als erlaubt geöffnet bleibt.
- **Sicherheitsfunktionen** – Abschaltung eines Signals nach einer maximalen Einschaltdauer.
- **Zeitbasierte Plausibilitätsprüfung** – Erkennung eines hängenden Sensors, der dauerhaft TRUE meldet.
- **Steuerung von Blink- oder Impulsfolgen** – der Timer kann als einfacher Monoflop mit Timeout verwendet werden.

Vergleich mit ähnlichen Bausteinen

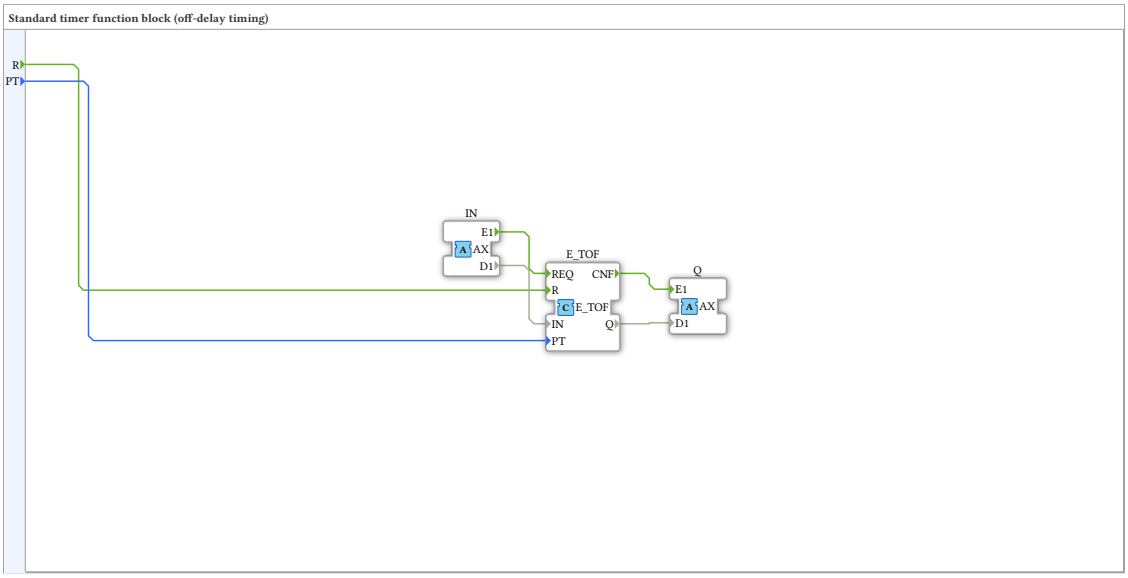
Baustein	Beschreibung	Unterschied zu AX_TLIM
E_TLIM	Standard-Timer (IEC 61499-2), direkte Ereignis- und Datenschnittstellen	AX_TLIM kapselt E_TLIM und bietet eine Adapter-basierte Schnittstelle (AX), die in

		bestimmten Architekturen leichter einzubinden ist.
E_DELAY	Verzögerungsbaustein (Ansprechverzögerung, Rückfallverzögerung)	E_DELAY verzögert Signaländerungen, während AX_TLIM eine maximale Aktivierungsdauer erzwingt.
E_SR	Set-Reset-Flipflop	AX_TLIM setzt den Ausgang nicht zurück, solange der Eingang aktiv ist (außer durch Timeout). E_SR reagiert auf separate Set/Reset-Ereignisse.

Fazit

Der **AX_TLIM** Funktionsblock bietet eine kompakte, adapterbasierte Lösung für zeitbegrenzte Überwachung boolescher Signale. Durch die Kapselung des Standard-E_TLIM und die Nutzung unidirektionaler AX-Adapter lässt er sich flexibel in IEC 61499-Systeme einbinden, insbesondere wenn Kommunikationspfade über Adapter gekapselt werden sollen. Die Funktionsweise ist intuitiv: Solange der Eingang aktiv ist, bleibt der Ausgang aktiv – maximal für die konfigurierbare Zeit PT. Dies macht den Baustein ideal für Sicherheits- und Überwachungsaufgaben in der Automatisierungstechnik.

AX_TOF

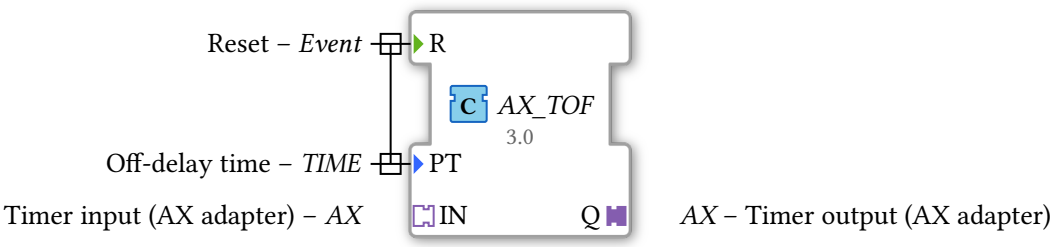


• – ○ • – ○ • – ○ •

Einleitung

Wichtiger Hinweis: Dieser Baustein benötigt nur ein Ereignis (Event) und keine zyklischen Aufrufe. Er besitzt keinen Ausgang ET und zeigt die verstrichene Zeit nicht an.

Der AX_TOF Funktionsblock ist ein standardmäßiger Timer-Baustein für Ausschaltverzögerungen (Off-Delay Timing). Er realisiert eine Zeitverzögerung beim Abschalten eines Signals und wird über Adapter-Schnittstellen angesteuert.



Schnittstellenstruktur

Ereignis-Eingänge
- **R**: Reset-Ereignis zum Zurücksetzen des Timers

Ereignis-Ausgänge
Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge
- **PT**: TIME - Prozesszeit für die Verzögerung

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden

Adapter
- **IN** (Socket): Eingangsadapter vom Typ `adapter::types::unidirectional::AX`
  - Ereignis: E1
  - Daten: D1
- **Q** (Plug): Ausgangsadapter vom Typ `adapter::types::unidirectional::AX`
  - Ereignis: E1
  - Daten: D1

Funktionsweise
Der AX_TOF Block implementiert eine Ausschaltverzögerung (Off-Delay). Bei einem eingehenden Signal über den IN-Adapter wird die Ausgabe sofort aktiviert. Beim Abschalten des Eingangssignals startet der Timer und hält die Ausgabe für die eingestellte Zeit PT aktiv, bevor diese deaktiviert wird.

Technische Besonderheiten
- Verwendet intern den E_TOF Standard-Timer-Baustein
- Unidirektionale Adapter-Schnittstellen für Ein- und Ausgang
- Reset-Funktionalität zum sofortigen Zurücksetzen des Timers
- Zeitbasis in TIME-Datentyp

Zustandsübersicht
1. **Ruhezustand**: Kein aktives Signal
2. **Aktiviert**: Eingangssignal vorhanden, Ausgabe aktiv
3. **Verzögerung**: Eingangssignal abgeschaltet, Timer läuft, Ausgabe bleibt aktiv
4. **Abgelaufen**: Timer abgelaufen, Ausgabe deaktiviert

Anwendungsszenarien
- Steuerung von Ausschaltverzögerungen in industriellen Prozessen
- Sicherheitsfunktionen mit Nachlaufzeiten
- Verzögerte Abschaltung von Motoren oder Aktoren
- Prozesssteuerung mit definierten Ausschaltsequenzen

⚖️ Vergleich mit ähnlichen Bausteinen
Im Vergleich zu anderen Timer-Bausteinen bietet AX_TOF:

- Spezialisiert auf Ausschaltverzögerungen
- Adapter-basierte Schnittstelle für modularen Aufbau
- Integrierte Reset-Funktionalität
- Standardisierte TIME-Datentypen

Vergleich mit E_TOF

🛠️ Zugehörige Übungen
- Uebung_020e_AX

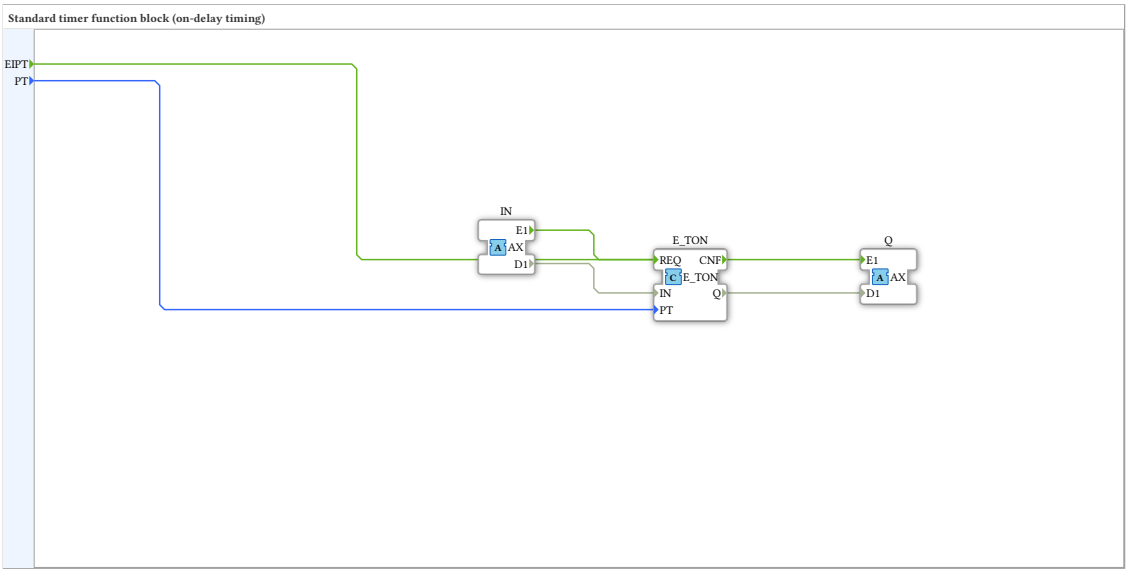
Fazit

Der AX_TOF Funktionsblock bietet eine zuverlässige und standardisierte Lösung für Ausschaltverzögerungen in 4diac-basierten Steuerungssystemen. Durch die Verwendung von Adapter-Schnittstellen ermöglicht er eine flexible Integration in komplexere Funktionsbaustein-Netzwerke.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AX_TON

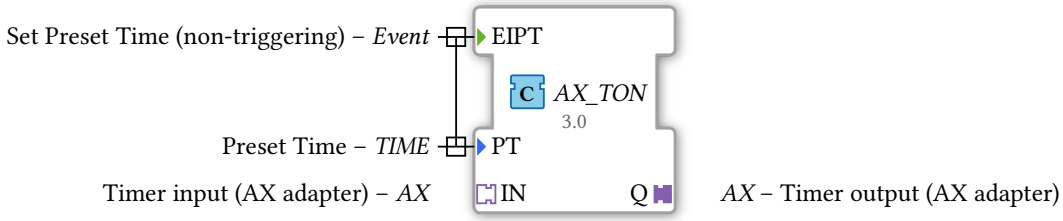


• – ○ • – ○ • – ○ •

Einleitung

Wichtiger Hinweis: Dieser Baustein benötigt nur ein Ereignis (Event) und keine zyklischen Aufrufe. Er besitzt keinen Ausgang ET und zeigt die verstrichene Zeit nicht an.

AX_TON ist ein standardisierter Timer-Funktionsblock, der eine Einschaltverzögerung (On-Delay Timing) realisiert. Der Baustein wartet nach Aktivierung eine definierte Zeit ab, bevor er das Ausgangssignal setzt.



Schnittstellenstruktur

Ereignis-Eingänge

- **EIPT** (Event): Setzt die Vorgabezeit (nicht auslösend); trägt `PT`

Ereignis-Ausgänge

- Keine direkten Ereignisausgänge vorhanden

Daten-Eingänge

- **PT** (TIME): Prozesszeit - definiert die Verzögerungsdauer

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden

Adapter

- **IN** (Input-Adapter): Empfängt Eingangsereignisse und -daten
- **Q** (Output-Adapter): Sendet Ausgangsereignisse und -daten

Funktionsweise

Der AX_TON-Block arbeitet als Einschaltverzögerungstimer. Bei Empfang eines Eingangsereignisses über den IN-Adapter startet der interne Timer. Nach Ablauf der eingestellten Prozesszeit (PT) wird über den Q-Adapter ein Ausgangsereignis mit dem Status TRUE gesendet. Wird der Eingang während der Zeitüberwachung deaktiviert, wird der Timer zurückgesetzt.

Technische Besonderheiten

- Implementiert auf Basis des E_TON-Standardtimerblocks
- Verwendet unidirektionale Adapter für Ein- und Ausgänge
- Unterstützt TIME-Datentyp für präzise Zeitsteuerung
- Kompatibel mit IEC 61499-2 Standard

Zustandsübersicht

1. **Ruhezustand**: Kein aktiver Timer, Ausgang FALSE
2. **Aktivierungszustand**: Timer läuft nach Eingangssignal
3. **Verzögerungszustand**: Timer zählt ab, Ausgang bleibt FALSE
4. **Auslösezustand**: Nach Zeitablauf, Ausgang wird TRUE

Anwendungsszenarien

- Steuerung von zeitverzögerten Prozessstarts
- Überwachung von Mindestlaufzeiten
- Verzögerte Aktivierung von Maschinenkomponenten
- Sicherheitsrelevante Zeitüberwachungen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Timer-Blöcken bietet AX_TON eine standardisierte Schnittstelle durch Adapter, was die Wiederverwendbarkeit und modulare Integration in größere Steuerungssysteme erleichtert.

Vergleich mit E_TON

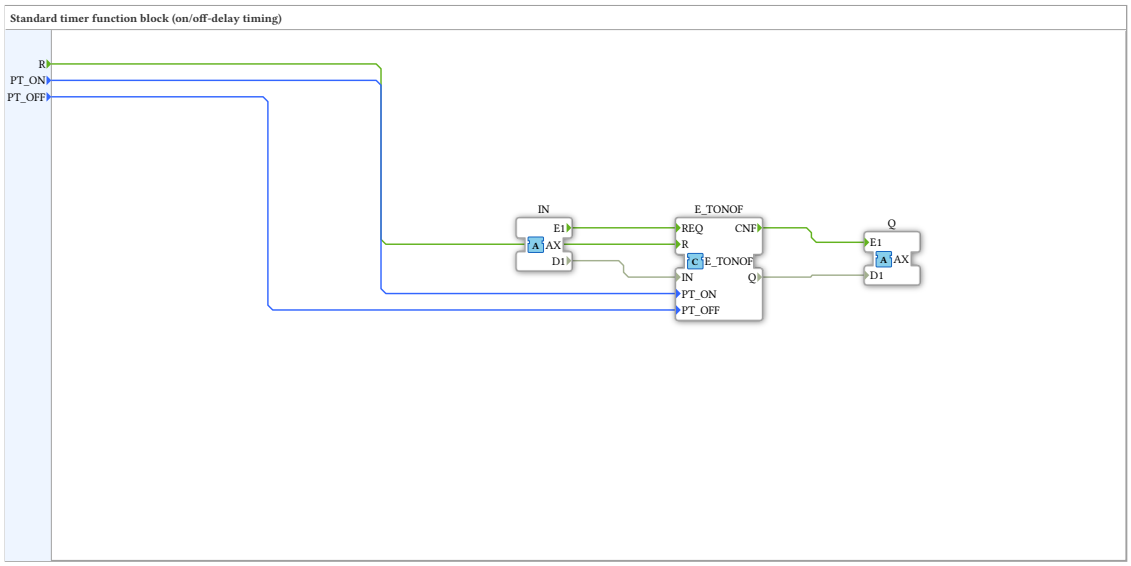
🛠️ Zugehörige Übungen

- Uebung_020c_AX
- Uebung_160b2_AX

Fazit

AX_TON ist ein robuster und zuverlässiger Timer-Baustein für Einschaltverzögerungen, der durch seine standardisierte Adapter-basierte Schnittstelle eine flexible Integration in verschiedene Automatisierungssysteme ermöglicht.

AX_TONOF

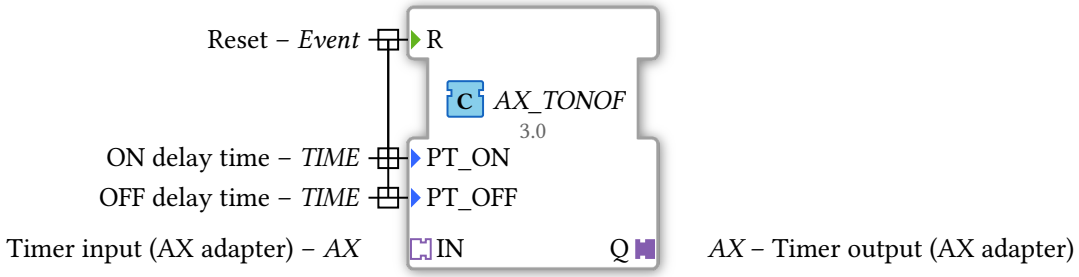


• – ○ • – ○ • – ○ •

Einleitung

Wichtiger Hinweis: Dieser Baustein benötigt nur ein Ereignis (Event) und keine zyklischen Aufrufe. Er besitzt keinen Ausgang ET und zeigt die verstrichene Zeit nicht an.

Der AX_TONOF ist ein standardisierter Timer-Funktionsblock, der sowohl Ein- als auch Ausschaltverzögerungen realisiert. Dieser Baustein kombiniert die Funktionalitäten eines Einschaltverzögerungs- und Ausschaltverzögerungs-Timers in einer kompakten Lösung.



Schnittstellenstruktur

Ereignis-Eingänge

- **R**: Reset-Ereignis zum Zurücksetzen des Timers

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

- **PT_ON**: TIME - Prozesszeit für die Einschaltverzögerung
- **PT_OFF**: TIME - Prozesszeit für die Ausschaltverzögerung

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden

Adapter

- **IN**: Eingangsadapter (Socket) für Steuersignale
- **Q**: Ausgangsadapter (Plug) für Statusausgaben

Funktionsweise

Der AX_TONOF-Funktionsblock verarbeitet Eingangssignale über den IN-Adapter und wendet je nach Signalzustand die konfigurierten Verzögerungszeiten an. Bei aktivem Eingangssignal wird die PT_ON-Verzögerung berücksichtigt, bei inaktivem Signal die PT_OFF-Verzögerung. Das Ergebnis wird über den Q-Adapter ausgegeben.

Technische Besonderheiten

- Unterstützt bidirektionale Zeitverzögerungen (Ein- und Ausschaltverzögerung)
- Integrierte Reset-Funktionalität
- Verwendet Adapter für Ein- und Ausgänge statt direkter Schnittstellen
- Basierend auf dem E_TONOF-Kernbaustein

Zustandsübersicht

Der Funktionsblock kann folgende Zustände annehmen:

- **Ruhezustand**: Keine aktive Zeitmessung
- **Einschaltverzögerung**: Zeitmessung für PT_ON aktiv
- **Ausschaltverzögerung**: Zeitmessung für PT_OFF aktiv
- **Reset-Zustand**: Alle aktiven Zeitmessungen werden abgebrochen

Anwendungsszenarien

- Motorsteuerungen mit Anlauf- und Stoppverzögerung
- Beleuchtungssteuerung mit Ein-/Ausschaltverzögerung
- Prozesssteuerungen mit zeitlich gestaffelten Schaltvorgängen
- Sicherheitsrelevante Anwendungen mit verzögerten Reaktionen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen Timer-Bausteinen wie TON oder TOF bietet AX_TONOF die kombinierte Funktionalität beider Timer-Typen in einem Baustein. Die Verwendung von Adaptern anstelle direkter Ein-/Ausgänge ermöglicht eine flexiblere Integration in komplexere Steuerungsstrukturen.

Vergleich mit E_TONOF

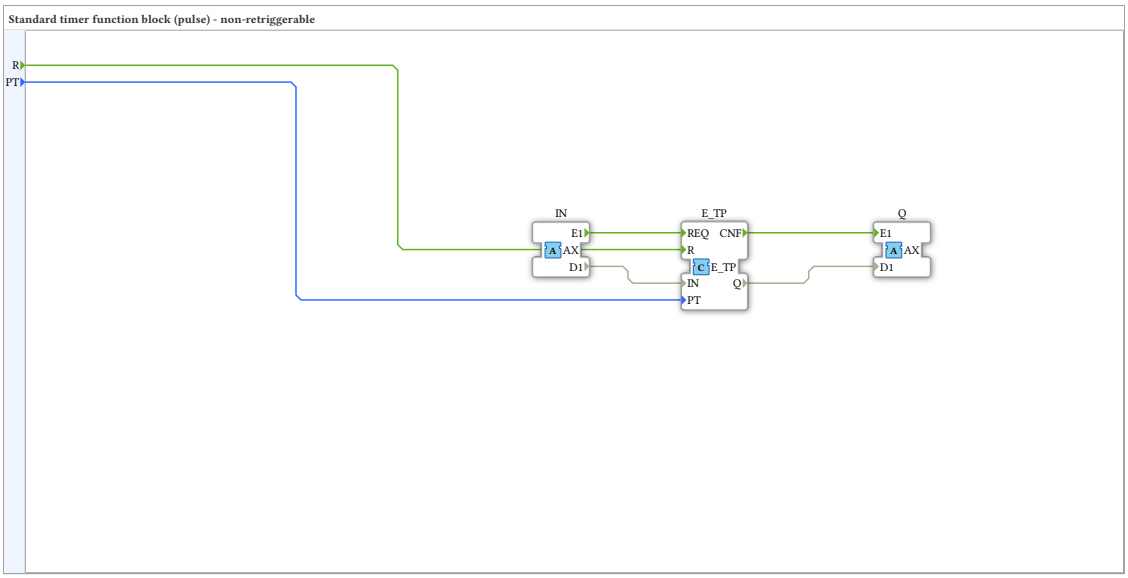
🛠️ Zugehörige Übungen

- Uebung_020g_AX

Fazit

Der AX_TONOF stellt einen vielseitigen Timer-Baustein dar, der durch seine kombinierte Ein-/Ausschaltverzögerungsfunktionalität und die Adapter-basierte Schnittstellengestaltung besonders für modulare Steuerungsarchitekturen geeignet ist. Die integrierte Reset-Funktion erhöht die Flexibilität in praktischen Anwendungen.

AX_TP

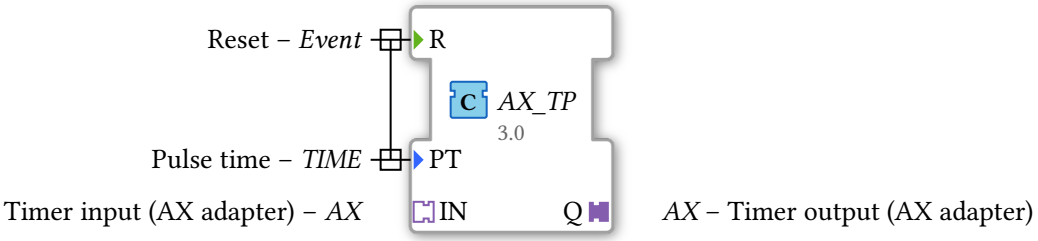


• – ○ • – ○ • – ○ •

Einleitung

Wichtiger Hinweis: Dieser Baustein benötigt nur ein Ereignis (Event) und keine zyklischen Aufrufe. Er besitzt keinen Ausgang ET und zeigt die verstrichene Zeit nicht an.

Der AX_TP ist ein standardisierter Timer-Funktionsblock vom Typ „Pulse“ (Impuls), der in 4diac-IDE Anwendungen verwendet wird. Dieser Baustein erzeugt einen zeitgesteuerten Impulsausgang basierend auf einem konfigurierbaren Zeitintervall.



Schnittstellenstruktur

Ereignis-Eingänge

- **R**: Reset-Ereignis - Setzt den Timer zurück

Ereignis-Ausgänge

- Keine direkten Ereignis-Ausgänge vorhanden

Daten-Eingänge

- **PT**: TIME - Pulszeit (Impulsdauer)

Daten-Ausgänge

- Keine direkten Daten-Ausgänge vorhanden

Adapter

- **IN**: Eingangsadapter (Socket) - Empfängt Eingangsereignisse und -daten
- **Q**: Ausgangsadapter (Plug) - Sendet Ausgangsereignisse und -daten

Funktionsweise

Der AX_TP-Funktionsblock arbeitet als Impulsgeber (Pulse Timer). Bei Empfang eines Ereignisses über den IN-Adapter wird ein zeitgesteuerter Impuls generiert, dessen Dauer durch den PT-Eingang definiert wird. Der Reset-Eingang (R) ermöglicht das vorzeitige Zurücksetzen des Timers. Der interne E_TP-Funktionsblock übernimmt die eigentliche Timer-Funktionalität.

Technische Besonderheiten

- Verwendet den standardisierten E_TP-Timerbaustein als Kernkomponente
- Implementiert über Adapter-Schnittstellen für modulare Anbindung
- Unterstützt TIME-Datentyp für Pulszeitkonfiguration
- Bietet separate Reset-Funktionalität

Zustandsübersicht

Der Funktionsblock verfügt über folgende Betriebszustände:

- **Inaktiv**: Kein aktiver Timer
- **Aktiv**: Timer läuft mit konfigurierter Pulszeit
- **Reset**: Timer wurde zurückgesetzt

Anwendungsszenarien

- Zeitgesteuerte Steuerungsabläufe
- Impulserzeugung für Aktoren
- Verzögerungsschaltungen
- Prozesssteuerung mit definierten Zeitintervallen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Timer-Bausteinen bietet AX_TP:

- Adapter-basierte Schnittstelle für bessere Modularität
- Explizite Reset-Funktionalität
- Standardisierte Puls-Timer-Implementierung

Vergleich mit E_TP

🛠️ Zugehörige Übungen

- Uebung_020f_AX
- AX_SPLIT_TO_2x_AX_TP
- Uebung_020j_AX
- Uebung_177_AX
- Uebung_178_AX

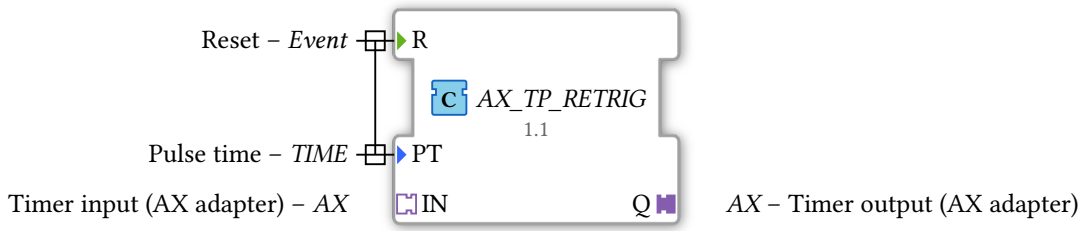
Fazit

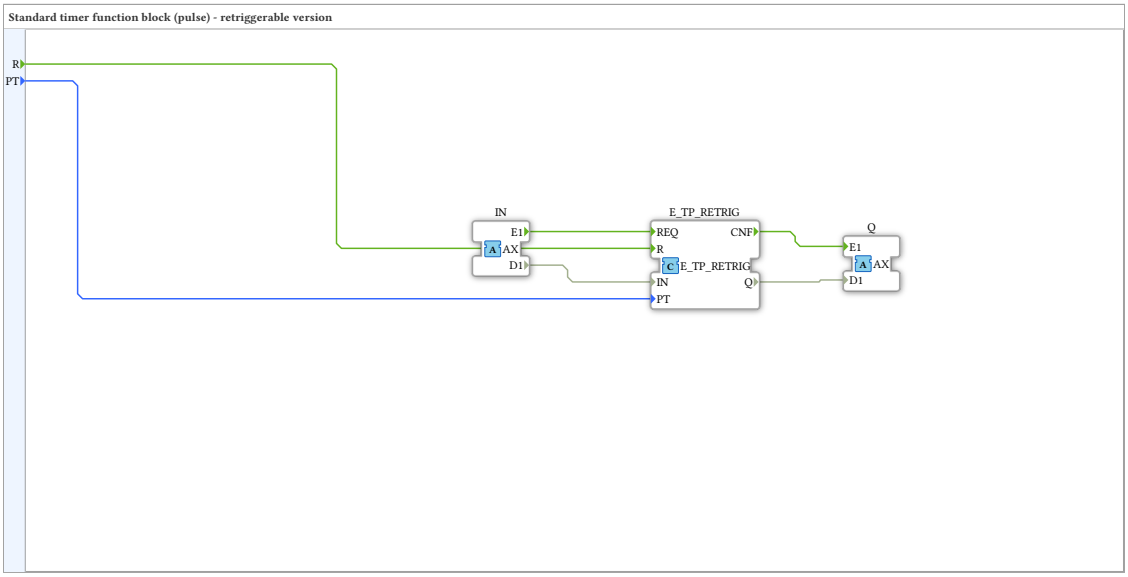
Der AX_TP ist ein robuster und standardkonformer Timer-Funktionsblock, der sich ideal für zeitgesteuerte Anwendungen in verteilten Automatisierungssystemen eignet. Seine Adapter-basierte Architektur ermöglicht flexible Integration in komplexere Steuerungsstrukturen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AX_TP_RETRIG





• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AX_TP_RETRIG** ist ein standardisierter Timer-Funktionsbaustein (Puls) in einer retriggerbaren Ausführung. Er erzeugt auf den Adapterausgang **Q** einen Impuls von der Dauer **PT**, sobald das Eingangssignal **IN** eine steigende Flanke (Ereignis REQ) empfängt. Der Impuls wird bei jedem neuen REQ-Ereignis neu gestartet (Retrigger). Ein Reset über den Ereigniseingang **R** setzt die Impulszeit **PT** vor und setzt den Timer zurück.

Der Baustein ist für den Einsatz in IEC 61499-basierten Automatisierungssystemen konzipiert und nutzt das AX-Adapter-Interface für eine effiziente, unidirektionale Signalübertragung.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung
R	Reset – Setzt die Impulszeit **PT** vor und beendet einen laufenden Impuls sofort.

Ereignis-Ausgänge

Keine eigenen Ereignisausgänge – die Ausgabe erfolgt über den Adapter **Q** (Ereignis **E1**).

Daten-Eingänge

Name	Typ	Beschreibung
PT	TIME	Impulsdauer (Pulszeit).

Daten-Ausgänge

Keine eigenen Datenausgänge – die Ausgabe des Impulszustands erfolgt über den Adapter **Q** (Datenwert **D1**).

Adapter

Name	Typ	Richtung	Beschreibung
IN	`adapter::types::unidirectional::AX`	Socket	Eingang – Empfängt das Startsignal (Ereignis **E1**) und den logischen Wert (**D1**). Eine steigende Flanke auf **E1** triggert den Impuls.

Q	`adapter::types::unidirectional::AX`	Plug	Ausgang – Gibt den Impuls aus: Solange der Impuls aktiv ist, ist **D1** = `TRUE`, sonst `FALSE**. Das Ereignis **E1** wird bei jedem Zustandswechsel gesendet.

Funktionsweise

1. **Initialzustand** – **Q.D1** ist `FALSE`, kein Impuls aktiv.
2. **Start** – Bei einem Ereignis **IN.E1** (steigende Flanke wird als REQ interpretiert) startet der Baustein einen Impuls der Dauer **PT**. **Q.D1** wird sofort auf `TRUE` gesetzt, das Ereignis **Q.E1** wird gesendet.
3. **Retrigger** – Während der Impuls noch aktiv ist, führt ein erneutes **IN.E1** dazu, dass der Timer zurückgesetzt wird und der Impuls für die volle Dauer **PT** neu beginnt. **Q.D1** bleibt während des gesamten Impulses `TRUE`.
4. **Impulsende** – Nach Ablauf von **PT** (ohne erneuten Trigger) wird **Q.D1** auf `FALSE` zurückgesetzt und **Q.E1** erneut gesendet.
5. **Reset** – Das Ereignis **R** beendet einen laufenden Impuls sofort und setzt **Q.D1** auf `FALSE`. Gleichzeitig wird **PT** mit dem aktuellen Wert am Eingang **PT** vorgesetzt (kann für eine neue Impulsdauer genutzt werden).

Technische Besonderheiten

- **Retriggerbarkeit** – Im Gegensatz zu einfachen Timer-Bausteinen wird ein laufender Impuls durch einen neuen Startimpuls nicht abgebrochen, sondern verlängert (Timer-Neustart).
- **Adapterschnittstelle** – Statt einzelner Ereignis-/Datenports wird das AX-Adapter-Interface verwendet, das eine standardisierte, unidirektionale Verbindung ermöglicht. Dadurch vereinfacht sich die Verschaltung in Adapter-basierten Architekturen.
- **Lizenz** – Der Baustein ist unter der **Eclipse Public License 2.0** veröffentlicht (siehe Copyright-Hinweise).
- **Implementierung** – Intern wird der IEC-61499-Standardbaustein **E_TP_RETRIG** genutzt, der über das Netzwerk eingebunden ist.

Zustandsübersicht

Der Baustein durchläuft folgende Zustände (basierend auf dem internen Timer-Baustein):

Zustand	Beschreibung
IDLE	Kein Impuls aktiv; **Q.D1** = `FALSE`.
RUNNING	Impuls aktiv; **Q.D1** = `TRUE`. Der Timer läuft für die Dauer **PT**. Bei erneutem Trigger wird der Timer zurückgesetzt (bleibt im Zustand RUNNING).
TIMEOUT	Impuls beendet; Wechsel zurück zu IDLE.

Hinweis: Der Reset (Ereignis **R**) führt immer sofort in den Zustand IDLE.

Anwendungsszenarien

- **Steuerung von Ausgangssignalen mit fester Impulsdauer** – z. B. Ansteuerung eines Relais, das für eine definierte Zeit geschaltet wird.
- **Taktgeber mit Nachtriggerung** – Wenn während eines Impulses ein erneutes Startsignal eintrifft, soll die Impulsdauer verlängert werden (z. B. bei einem manuellen Taster, der länger gedrückt wird).
- **Sicherheitsverriegelungen** – Ein Alarm oder eine Aktion soll für eine Mindestdauer aktiv bleiben, auch wenn der auslösende Zustand kurzzeitig verschwindet.
- **Zeitgesteuerte Freigaben** – z. B. eine Maschinenbewegung, die nach einem Startsignal für die Dauer **PT** ausgeführt wird, aber bei erneutem Startimpuls neu beginnt.

Vergleich mit ähnlichen Bausteinen

Baustein	Eigenschaft
AX_TP_RETRIG	Retriggerbarer Puls-Timer mit AX-Adapter.
E_TP (Standard IEC 61499)	Einfacher Puls-Timer, nicht retriggerbar – ein erneuter Start während des Impulses wird ignoriert.

E_TP_RETRIG	Retriggerbare Version ohne Adapter (reine Event/Daten-Schnittstelle).
E_DELAY (Verzögerung)	Verzögert ein Signal um eine feste Zeit, erzeugt aber keinen Impuls fester Dauer.

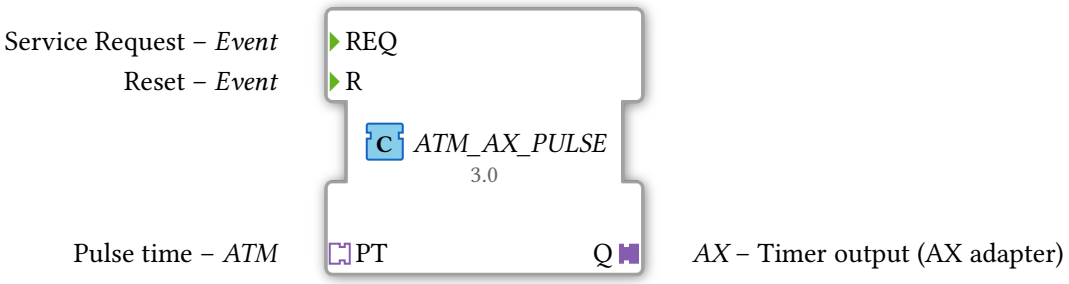
Der **AX_TP_RETRIG** kombiniert die Retrigger-Funktionalität mit der kompakten Adapter-Schnittstelle und eignet sich daher besonders für modulare, adapterbasierte Systeme.

Fazit

Der **AX_TP_RETRIG** ist ein flexibler und standardisierter Timer-Funktionsbaustein für IEC 61499-Anwendungen. Seine Retrigger-Eigenschaft macht ihn ideal für Situationen, in denen ein Impuls bei wiederholten Startbefehlen verlängert werden soll. Die Verwendung des AX-Adapter-Interfaces vereinfacht die Anbindung an andere Adapter-basierte Komponenten und fördert eine saubere, modulare Systemarchitektur. Durch die klare Trennung von Ereignis- und Datenflüssen sowie die Einhaltung des Standards eignet sich der Baustein für den zuverlässigen Einsatz in industriellen Steuerungssystemen.

ATM

ATM_AX_PULSE



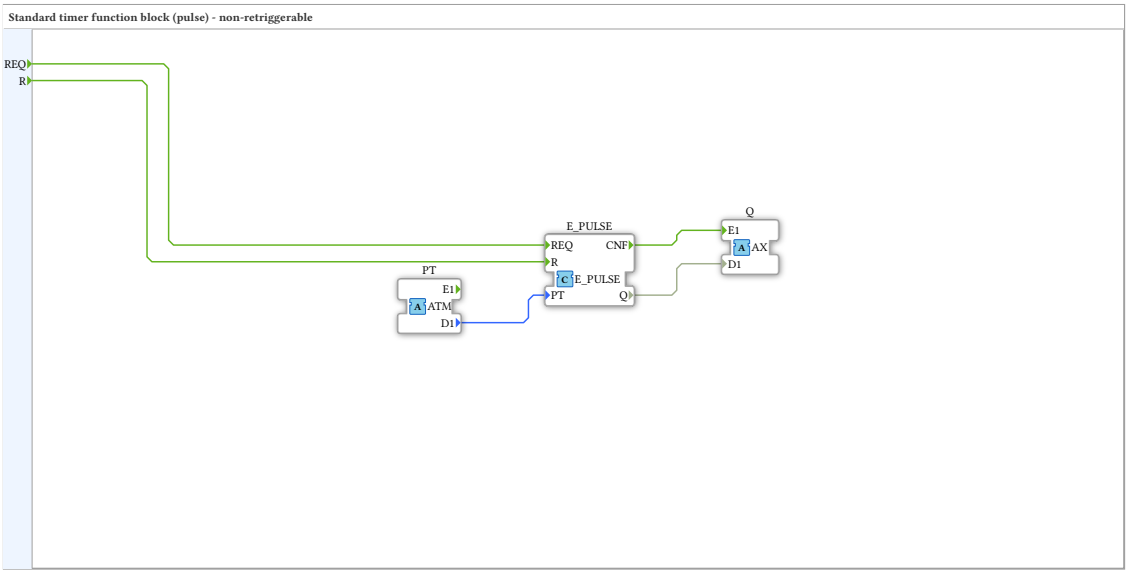


Bild nicht verfügbar

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ATM_AX_PULSE** ist ein standardisierter Timerbaustein zur Erzeugung eines nicht nachtriggerbaren Impulses (non-retriggerable). Er wird über die Adapter-Schnittstellen **PT** (ATM) und **Q** (AX) konfiguriert und ausgelesen. Der Baustein entspricht der IEC 61499-2 Norm und ist Teil der Bibliothek **adapter::events::unidirectional::timers**.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung
REQ	Event	Service Anforderung – startet den Impuls.
R	Event	Reset – beendet den Impuls vorzeitig.

Ereignis-Ausgänge

Keine direkten Ereignisausgänge. Die Ausgabe erfolgt über den Adapter **Q** (siehe unten).

Daten-Eingänge

Keine direkten Dateneingänge. Die Impulszeit wird über den Adapter **PT** bereitgestellt.

Daten-Ausgänge

Keine direkten Datenausgänge. Der Impulszustand wird über den Adapter **Q** abgefragt.

Adapter

Adapter	Richtung	Typ	Beschreibung
PT	Socket	`adapter::types::unidirectional::ATM`	Liefert die Impulsdauer als Zeitwert.
Q	Plug	`adapter::types::unidirectional::AX`	Gibt den Impulsstatus aus: **D1** = aktueller Wert (TRUE während des Impulses); **E1** = Ereignis bei Impulsende oder -wechsel.

Funktionsweise

Der Baustein realisiert das Verhalten eines nicht nachtriggerbaren Impulsgebers (E_PULSE). Beim Eintreffen des Ereignisses **REQ** wird ein Impuls gestartet, dessen Dauer durch den am Adapter **PT** anliegenden Wert bestimmt wird. Während des Impulses ist der Ausgang **Q.D1** = *TRUE*. Nach Ablauf der Impulszeit geht **Q.D1** auf *FALSE* und das Ereignis **Q.E1** wird ausgelöst. Ein erneutes **REQ** während eines laufenden Impulses wird ignoriert (non-retriggerable). Über den Reset-Eingang **R** kann der Impuls jederzeit vorzeitig beendet werden; auch dann wird **Q.E1** ausgelöst.

Technische Besonderheiten

- **Adapter-basierte Kommunikation:** Statt direkter Daten-/Event-Ports werden typisierte Adapter (ATM für Zeit, AX für Ausgabe) verwendet. Dies vereinfacht die Wiederverwendung in modularen Systemen.
- **Nicht nachtriggerbar:** Ein zweiter Startimpuls während des laufenden Impulses hat keine Wirkung.
- **Validierte Versionen:** Der Baustein wurde mehrfach überarbeitet (Versionen 1.0 bis 3.0, zuletzt 2025) und enthält eine Reset-Funktionalität.
- **Lizenziert unter EPL 2.0** (Eclipse Public License 2.0).

Zustandsübersicht

1. **Idle** – Kein Impuls aktiv. **Q.D1** = FALSE.
2. **PulseActive** – Impuls läuft. **Q.D1** = TRUE. Wird durch **REQ** betreten und entweder durch Zeitablauf oder **R** verlassen.
   - Bei Zeitablauf: Übergang zu Idle, Auslösen von **Q.E1**.
   - Bei **R**: Übergang zu Idle, Auslösen von **Q.E1**.
3. Während **PulseActive** wird **REQ** ignoriert.

Anwendungsszenarien

- Steuerung von Aktoren (z. B. Ventile, Lampen) mit definierter Einschaltdauer.
- Zeitgeber in Produktionsmaschinen, die einmalige Impulse benötigen.
- Signalaufbereitung in Sensor-Aktor-Netzwerken (AX-/ATM-kompatible Peripherie).

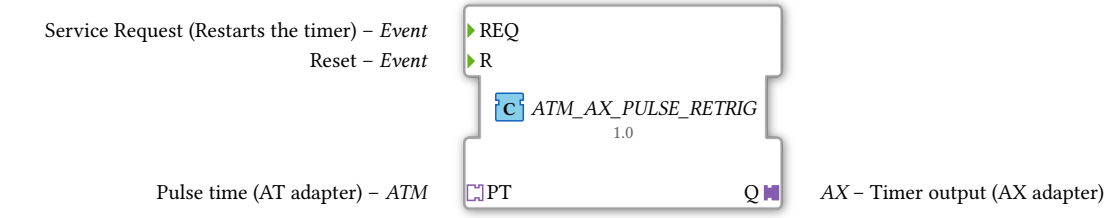
Vergleich mit ähnlichen Bausteinen

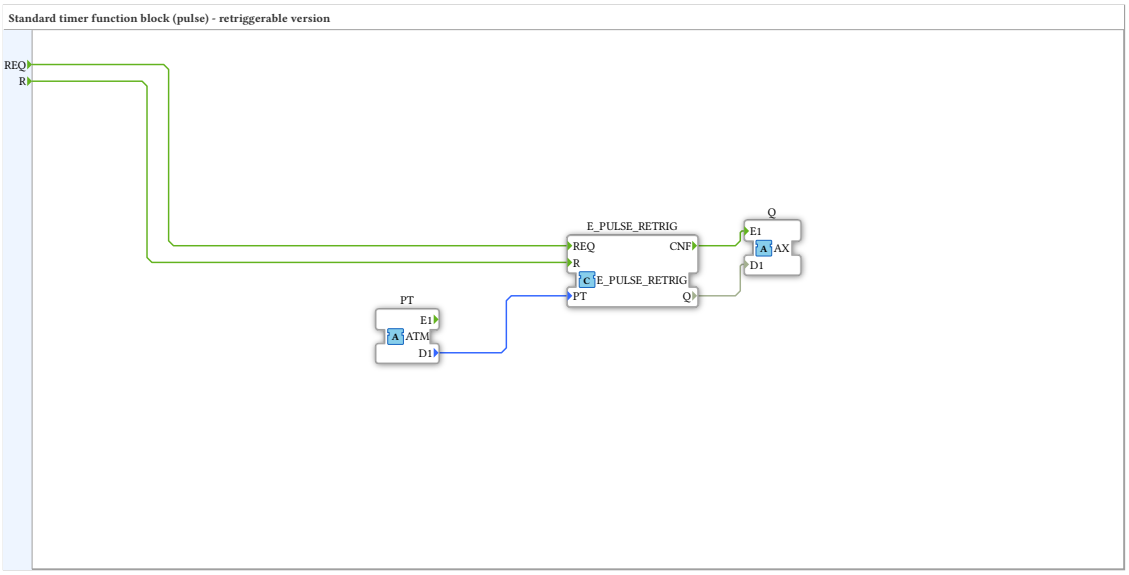
- **E_PULSE (iec61499::events::timers):** Standard-Implementierung ohne Adapter-Schnittstelle. ATM_AX_PULSE kapselt diesen Funktionsblock und fügt die Adapter-Anbindung hinzu.
- **ATM_AX_PULSE_R (retriggerable):** Nachtriggerbare Variante; ein erneuter **REQ** verlängert den Impuls. Der vorliegende Baustein ist nicht nachtriggerbar.
- **E_DELAY, E_CYCLE:** Andere Zeitverhalten (Verzögerung, periodisch) – nicht vergleichbar.

Fazit

Der **ATM_AX_PULSE** ist ein robuster, nicht nachtriggerbarer Impulsbaustein, der durch seine Adapter-Schnittstelle besonders gut in modulare Automatisierungslösungen mit AX/ATM-Protokoll integriert werden kann. Er bietet eine klare, zuverlässige Funktionalität zur Erzeugung zeitdefinierter Impulse und erfüllt die Anforderungen der IEC 61499-2.

ATM_AX_PULSE_RETRIG





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **ATM_AX_PULSE_RETRIG** erzeugt einen Impuls (Puls) definierter Dauer an seinem Ausgang. Er ist als retriggerbare Version ausgelegt, d.h. ein erneuter Startbefehl während eines laufenden Impulses setzt die Impulsdauer zurück. Die Steuerung und Parameterübergabe erfolgt ausschließlich über zwei standardisierte Adapter-Schnittstellen (ATM / AX), sodass der Baustein ohne direkte Daten-Ein-/Ausgänge auskommt.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
REQ	Event	Startbefehl – löst einen neuen Impuls aus und startet die Timer-Zeit neu
R	Event	Reset – beendet einen laufenden Impuls sofort

Ereignis-Ausgänge
Keine.

Daten-Eingänge
Keine. Die Impulsdauer wird über den Adapter **PT** (Socket) bereitgestellt.

Daten-Ausgänge
Keine. Der Impulsstatus (aktiv/inaktiv) wird über den Adapter **Q** (Plug) ausgegeben.

Adapter

Rolle	Name	Typ	Richtung	Beschreibung
Input	PT	ATM	Socket	Liefert die gewünschte Impulsdauer (D1 → PT)
Output	Q	AX	Plug	Zeigt den Impulszustand an (D1 = HIGH während des Impulses)

Funktionsweise
Wird der Baustein mit einem **REQ**-Ereignis angestoßen, startet er einen Timer mit der Dauer, die über den Adapter **PT** (Datenwert D1) vorgegeben ist. Der Ausgang **Q** (D1) wird sofort auf **HIGH** gesetzt und bleibt für genau diese Zeit aktiv. Trifft während eines laufenden Impulses erneut ein **REQ** ein, wird der Timer zurückgesetzt und die Impulsdauer beginnt von vorn (retriggerbar). Ein **Reset**-Ereignis (**R**) beendet den Impuls vorzeitig und setzt **Q** auf **LOW**.

Der Baustein implementiert die interne Logik mittels des standardisierten IEC-61499-Funktionsbausteins **E_PULSE_RETRIG**, der alle Zeitsteuerungen übernimmt. Die Kopplung der Adapter (PT → D1 an internen Timer, interner Ausgang Q → Q.D1) erfolgt im verborgenen Netzwerk.

Technische Besonderheiten
- **Retriggerbares Verhalten**: Jeder neue **REQ** während eines aktiven Impulses verlängert die Impulsdauer auf den vollen Wert von PT.
- **Adapter-basierte Schnittstelle**: Der Baustein besitzt keine direkten Ereignis-Ausgänge oder Daten-Ports; die gesamte Kommunikation erfolgt über die standardisierten **ATM** (Parameter) und **AX** (Ausgang) Adapter.
- **Wiederverwendbarkeit**: Durch die strikte Trennung von Steuerung und Parametrierung eignet sich der Baustein besonders für modulare und wiederverwendbare Anwendungen.

Zustandsübersicht
Der Baustein kennt zwei Zustände:

- **Inaktiv** (Q.D1 = FALSE): Kein Impuls aktiv, Timer läuft nicht.
- **Aktiv** (Q.D1 = TRUE): Impuls läuft; bei Eintritt durch REQ wird der Timer mit PT gestartet. Jeder weitere REQ setzt den Timer zurück. Ein R-Ereignis führt sofort zurück in den inaktiven Zustand.

Anwendungsszenarien
- **Taktgeber für Signalleuchten**: Ein kontinuierliches Startsignal (z. B. Taster oder Sensor) erzeugt einen definierten Lichtimpuls. Durch die Retrigger-Funktion bleibt die Leuchte bei Dauersignal dauerhaft an.
- **Impulsverlängerung in Steuerungen**: Kurze Eingangsimpulse werden auf eine minimale bzw. definierte Dauer aufgeweitet.
- **Ansteuerung von Aktoren mit Mindesteinschaltzeit**: Ventile, Motoren oder Relais, die eine bestimmte Einschaltdauer benötigen, können zuverlässig angesteuert werden.

Vergleich mit ähnlichen Bausteinen

Baustein	Retriggerbar	Schnittstelle	Einsatzgebiet
ATM_AX_PULSE_RETRIG	Ja	Adapter (ATM/AX)	Modulare Steuerungen mit Adapter-Konzept
ATM_AX_PULSE	Nein	Adapter (ATM/AX)	Einmalige Impulse, keine Neuauslösung

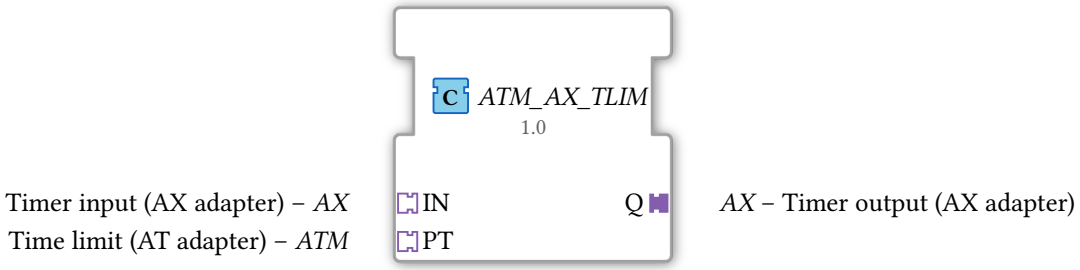
Standard E_PULSE_RETRIG	Ja	Direkte Ein-/Ausgänge	Flexibler, aber ohne Adapter-Kapselung

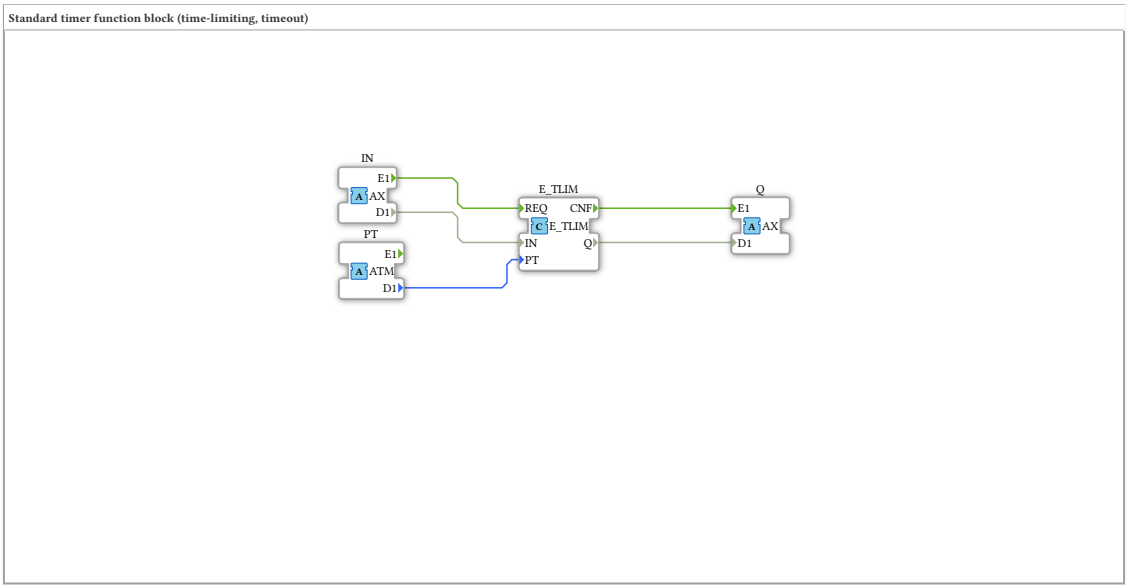
Der vorliegende Baustein hebt sich durch die ausschließliche Verwendung von Adapter-Schnittstellen ab, was die Einbindung in bestehende Adapter-basierte Bibliotheken erleichtert.

Fazit

ATM_AX_PULSE_RETRIG ist ein kompakter, retriggerbarer Impulsbaustein, der sich durch seine saubere Kapselung mittels ATM/AX-Adapter auszeichnet. Er eignet sich hervorragend für den Einsatz in verteilten Steuerungssystemen, die auf das IEC-61499-Adapterkonzept bauen. Durch die einfache Handhabung (zwei Ereignisse, zwei Adapter) und das bewährte Timer-Verhalten ist er eine verlässliche Komponente für zeitgesteuerte Ausgaben.

ATM_AX_TLIM





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ATM_AX_TLIM** ist ein standardisierter Zeitüberwachungsbaustein (Timer) mit einer speziellen Adapter-Schnittstelle. Er realisiert eine Zeitbegrenzung (Time-Limiting / Timeout) für boolesche Eingangssignale. Der Baustein kommuniziert über Adapter statt über einzelne Ereignis- und Datenports, was eine flexible und gekapselte Integration in industrielle Steuerungssysteme ermöglicht.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
EIPT	Event	Set Preset Time (nicht auslösend)

Ereignis-Ausgänge

Der Baustein besitzt keine separaten Ereignis-Ausgänge. Die Ereignisweiterleitung erfolgt ausschließlich über den Adapterausgang **Q**, der nach einer Änderung des Ausgangssignals ein Ereignis auf seinem `E1`-Port auslöst.

Daten-Eingänge

Es existieren keine direkten Dateneingänge. Die erforderlichen Daten (Eingangssignal IN und Zeitgrenze PT) werden über die entsprechenden Adapterports bereitgestellt.

Daten-Ausgänge

Analog dazu gibt es keine direkten Datenausgänge. Das Resultat (boolescher Wert Q) wird über den Adapterausgang **Q** (Port `D1`) ausgegeben.

Adapter

Bezeichnung	Typ	Richtung	Kommentar
IN	AX	Eingang	Boolescher Eingang (Adapter)
PT	ATM	Eingang	Zeitgrenze (Adapter)
Q	AX	Ausgang	Boolescher Ausgang (Adapter)

Funktionsweise

Der Baustein arbeitet intern mit dem Standard-Funktionsblock **E_TLIM** und verleiht diesem eine Adapter-Hülle. Die Logik ist wie folgt:

1. **Aktivierung**: Ein Ereignis am Adaptereingang **IN** (Port `E1`) löst die Verarbeitung aus. Gleichzeitig wird der boolesche Wert von **IN.D1** übernommen.
2. **Zeitmessung**: Wird das Eingangssignal `IN` **TRUE**, setzt der Baustein den Ausgang `Q` auf **TRUE** und startet einen Timer mit der Dauer **PT** (von Adapter **PT**, Port `D1`).
3. **Timeout**: Bleibt `IN` länger als `PT` aktiv, so wird `Q` auf **FALSE** zurückgesetzt (Timeout).
4. **Vorzeitiges Ende**: Wird `IN` vor Ablauf der Zeit **FALSE**, so setzt der Baustein `Q` sofort zurück (ebenfalls **FALSE**).
5. **Setzen der Zeitgrenze**: Der Ereigniseingang **EIPT** erlaubt es, den Wert von **PT** vorzugeben, ohne eine neue Auslösung zu starten. Dies dient zur Konfiguration des Zeitlimits vor dem eigentlichen Einsatz.

Der Ausgangsadapter **Q** gibt auf seinem `E1`-Port ein Ereignis aus, sobald sich `Q` ändert (steigende oder fallende Flanke).

Technische Besonderheiten

- **Adapterbasierte Kommunikation**: Statt einzelner Ereignis- und Datenports werden AX-/ATM-Adapter verwendet. Diese ermöglichen eine modulare und wiederverwendbare Kopplung mit anderen Bausteinen.
- **Interner Standardbaustein**: Die Implementierung nutzt den etablierten **E_TLIM** (IEC 61499), womit eine zuverlässige und getestete Zeitlogik gewährleistet ist.
- **Nicht-auslösendes Setzen der Zeit**: Der Eingang **EIPT** ändert nur das Zeitlimit, ohne den Timer zu starten oder zurückzusetzen. Dies ist nützlich, um Parameter dynamisch anzupassen.
- **Typische Zeitmessung**: Geeignet für Überwachungen, bei denen ein Signal nicht länger als eine bestimmte Dauer aktiv sein darf.

Zustandsübersicht

Der FB durchläuft implizit folgende Zustände:

Zustand	Beschreibung
Idle	IN = FALSE, Q = FALSE, Timer läuft nicht.
Timing	IN = TRUE, Q = TRUE, Timer läuft.
Timeout	IN = TRUE, Timer abgelaufen, Q = FALSE (bleibt bis IN = FALSE).

Ein Wechsel zurück nach **Idle** erfolgt, sobald IN auf FALSE geht. Der Zustand **Timeout** wird nur erreicht, wenn die Zeitüberschreitung eingetreten ist.

Anwendungsszenarien

- **Überwachung von Signalen** in der Automatisierungstechnik (z. B. maximale Einschaltdauer eines Aktors).
- **Sicherheitsfunktionen**, bei denen ein Ausgang deaktiviert werden muss, wenn ein Signal zu lange ansteht.
- **Zeitgesteuerte Resets** in Kommunikationsprotokollen (Watchdog-Funktionalität).
- **Einfache zeitliche Begrenzung** in komplexen Steuerungssystemen, die auf Adapter-Schnittstellen basieren.

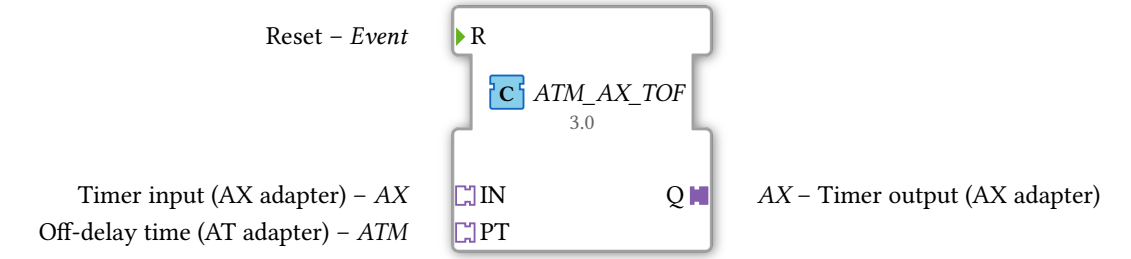
Vergleich mit ähnlichen Bausteinen

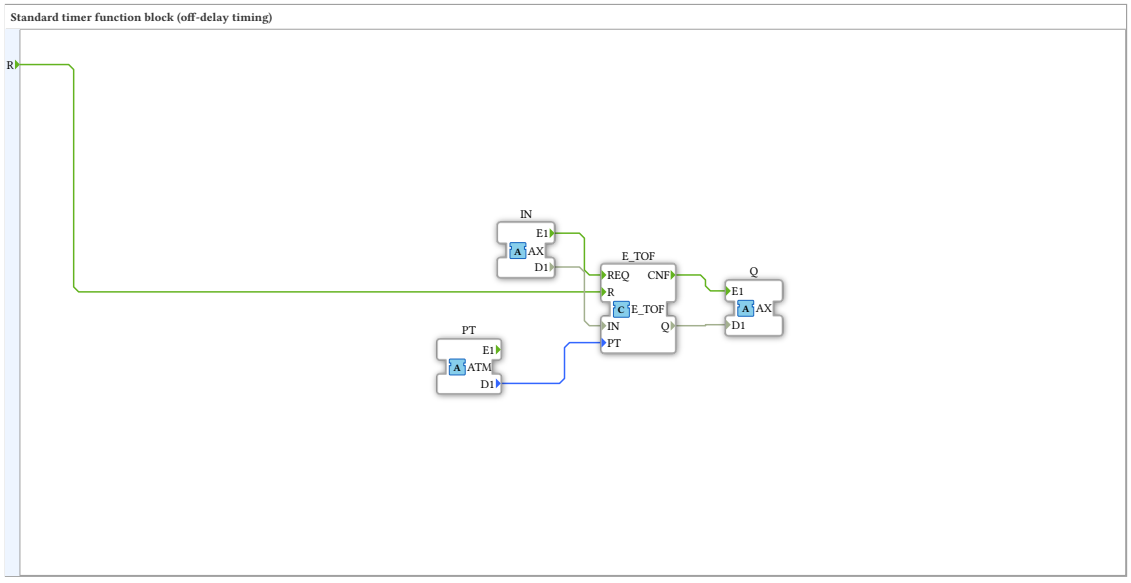
- **E_TLIM (Standard)**: Bietet die gleiche Kernlogik, jedoch mit klassischen Ein-/Ausgangsports. **ATM_AX_TLIM** kapselt diese Logik in eine Adapter-Schnittstelle und ist somit besser für modulare und wiederverwendbare Komponenten geeignet.
- **E_TON / E_TOF**: Diese Bausteine realisieren Einschalt- bzw. Ausschaltverzögerungen. Im Gegensatz dazu überwacht **ATM_AX_TLIM** die maximale Dauer eines aktiven Signals und reagiert mit einem Timeout.
- **E_CYCLE**: Ein zyklischer Timer, der periodisch Signale erzeugt, hat eine andere Zielsetzung als die reine Überwachung.

Fazit

Der Funktionsblock **ATM_AX_TLIM** bietet eine kompakte und adapterbasierte Lösung für Zeitüberwachungsaufgaben. Er kombiniert die bewährte Funktionalität des Standard-Timers **E_TLIM** mit der Flexibilität von Adapter-Schnittstellen und erlaubt eine dynamische Vorgabe der Zeitgrenze über **EIPT**. Dadurch eignet er sich besonders für moderne, modulare Automatisierungsprojekte, die auf IEC 61499 basieren und eine klare Trennung von Schnittstellen und Logik erfordern.

ATM_AX_TOF





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ATM_AX_TOF` realisiert eine Ausschaltverzögerung (off-delay timing) mit einer modularen Adapter-Schnittstelle auf Basis der Typen `AX` und `ATM`. Er wird typischerweise eingesetzt, um nach dem Wegfallen eines Eingangssignals einen Ausgang für eine definierte Zeit aktiv zu halten. Der Baustein ist als Standard-Timer-FB gemäß IEC 61499-2 klassifiziert und intern mit dem FB `E_TOF` realisiert.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
R	Event	Setzt den Timer zurück (Reset)

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Der Ausgang erfolgt über den Adapter `Q`.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden. Die Daten werden über die Adapter `IN` und `PT` bereitgestellt.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden. Der Ausgang erfolgt über den Adapter `Q`.

Adapter

Name	Richtung	Typ	Beschreibung
IN	Socket	AX	Eingangsadapter: Liefert das boolsche Triggersignal (D1: BOOL) und einen Event (E1) zur Auslösung der Timer-Funktion
PT	Socket	ATM	Zeitadapter: Liefert die Verzögerungszeit (D1: TIME)
Q	Plug	AX	Ausgangsadapter: Stellt das boolsche Ausgangssignal (D1: BOOL) und einen Bestätigungs-Event (E1) bereit

Funktionsweise

Der `ATM_AX_TOF` arbeitet nach dem Prinzip einer **Ausschaltverzögerung** (Off-Delay):

1. **Normalbetrieb:** Solange der Eingang `IN.D1` (BOOL) den Wert `TRUE` hat, ist der Ausgang `Q.D1` sofort `TRUE` (ohne Verzögerung).
2. **Ausschaltverzögerung:** Wechselt `IN.D1` von `TRUE` auf `FALSE`, bleibt `Q.D1` für die in `PT.D1` (TIME) angegebene Zeitspanne auf `TRUE`. Nach Ablauf dieser Zeit geht `Q.D1` auf `FALSE`.
3. **Reset:** Ein Ereignis am Eingang `R` setzt den internen Timer sofort zurück. `Q.D1` wird unverzüglich `FALSE`, unabhängig vom aktuellen Zustand von `IN.D1` oder von der Restzeit.

Die Auslösung erfolgt durch den Event `IN.E1` – ein neuer Wert an `IN.D1` wird erst nach einem Ereignis übernommen. Ein Ereignis auf `Q.E1` bestätigt jede Zustandsänderung von `Q.D1`.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle:** Die Verwendung der universellen Adapter `AX` und `ATM` ermöglicht eine flexible Anbindung an Adressierungs- oder andere Module ohne feste Verdrahtung.
- **Interne Realisierung:** Der Baustein nutzt den Standard-FB `E_TOF` aus der IEC 61499-Bibliothek, was eine robuste und geprüfte Timerlogik gewährleistet.
- **Reset-Funktion:** Der Reset-Eingang `R` erlaubt ein sofortiges Abbrechen der Verzögerung, z. B. zur Steuerung von Sicherheitsfunktionen.
- **Kompatibilität:** Entwickelt für die 4diac-IDE und basierend auf dem Eclipse Public License 2.0.

Zustandsübersicht

Der FB durchläuft folgende Betriebszustände:

Zustand	Bedingungen	Q.D1	Beschreibung
Idle	IN.D1 = FALSE, Timer läuft nicht	FALSE	Ruhezustand nach Ablauf oder Reset
Active	IN.D1 = TRUE	TRUE	Eingang aktiv, Ausgang sofort TRUE
Timing	IN.D1 von TRUE → FALSE, Timer aktiv	TRUE	Verzögerungsphase: Ausgang bleibt TRUE für Dauer PT
Resetting	Ereignis R während Timing oder Active	FALSE	Timer wird sofort gestoppt, Ausgang geht auf FALSE

Anwendungsszenarien

- **Nachlauf von Lüftern:** Nach Abschalten eines Motors läuft der Lüfter für eine einstellbare Zeit weiter, um Restwärme abzuführen.
- **Entprellung von Signalen:** Kurze Unterbrechungen eines Sensorsignals werden durch die Verzögerung überbrückt.
- **Beleuchtungssteuerung:** Licht bleibt nach Verlassen eines Raumes für eine definierte Nachlaufzeit an.
- **Sicherheitsabschaltungen:** Reset über R ermöglicht ein sofortiges Abschalten in Notfällen.

Vergleich mit ähnlichen Bausteinen

Eigenschaft	ATM_AX_TOF	E_TOF (Standard)
Schnittstelle	Adapter (AX/ATM)	Direkte Events/Daten
Reset-Funktion	Ja (Ereignis R)	Ja (Ereignis R)
Auslösung	Ereignisgesteuert über Adapter-Event	Ereignisgesteuert über REQ
Flexibilität	Höher durch Adapter-Kopplung	Einfacher, aber starrer
Einsatzbereich	Modulare Automatisierungssysteme	Grundfunktion in IEC 61499-Editoren

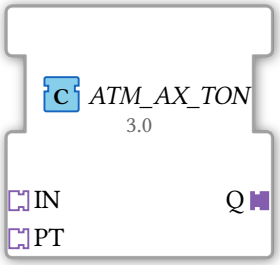
Fazit

Der `ATM_AX_TOF` bietet eine zuverlässige und flexible Ausschaltverzögerung für modulare Automatisierungsprojekte. Dank der Adapter-Schnittstelle lässt er sich leicht in verschiedene Systemlandschaften integrieren und erweitert die Funktionalität des Standard-Timers `E_TOF` um eine Reset-Möglichkeit sowie eine lose Kopplung über AX/AT-Adapter. Er ist ideal für Anwendungen, bei denen eine zeitverzögerte Abschaltung mit optionaler Sofortrücksetzung benötigt wird.

Passende Themen-Unterseiten auf ms-muc-docs.de

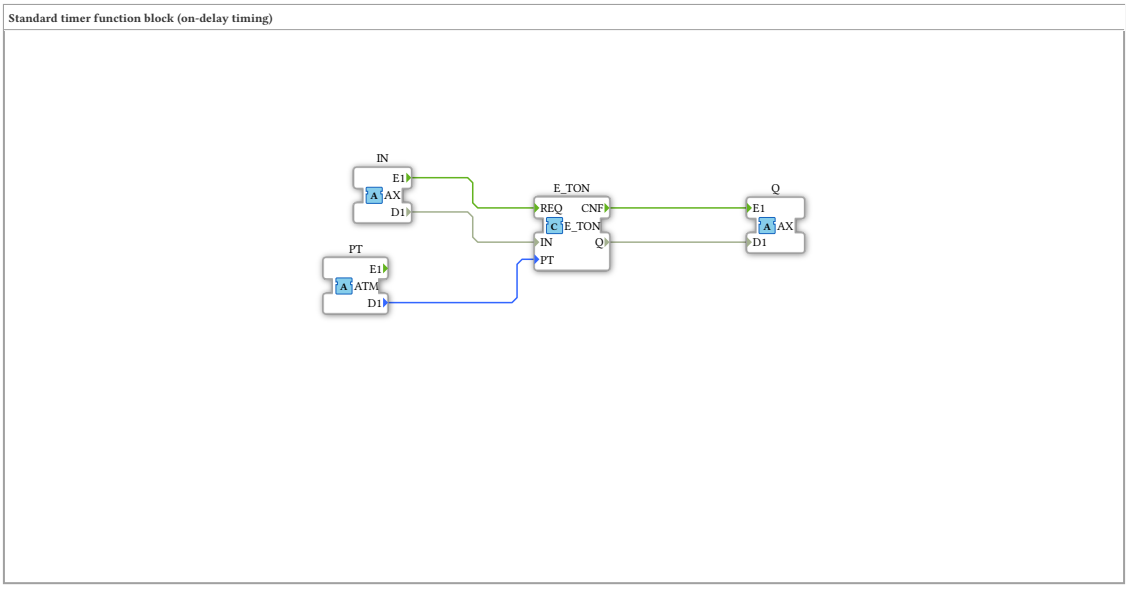
- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ATM_AX_TON



Timer input (AX adapter) – *AX*
Preset time (AT adapter) – *ATM*

AX – Timer output (AX adapter)



• – ○ • – ○ • – ○ •

Einleitung

Der **ATM_AX_TON** ist ein Funktionsblock zur Verzögerung beim Einschalten (On-Delay). Er realisiert eine Standard-Timer-Funktionalität nach IEC 61499-2, die über eine Adapter-Schnittstelle (AX/AT) angesteuert wird. Der Baustein ermöglicht es, einen Timer über einen Eingangsadapter zu starten und den Ausgangszustand sowie die Ablaufbestätigung über einen Ausgangsadapter bereitzustellen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
EIPT	Event	Setzt die Vorlaufzeit (nicht auslösend)

Ereignis-Ausgänge

Der FB besitzt keine eigenen Ereignis-Ausgänge. Bestätigungen und Zustandsänderungen werden über den Adapter **Q** (E1) kommuniziert.

Daten-Eingänge

Der FB hat keine expliziten Daten-Eingänge. Alle Eingabedaten werden über die Adapter-Schnittstellen bereitgestellt.

Daten-Ausgänge

Der FB hat keine expliziten Daten-Ausgänge. Ausgabedaten werden über den Adapter **Q** (D1) bereitgestellt.

Adapter

Richtung	Name	Typ	Kommentar
Socket	IN	adapter::types::unidirectional::AX	Timer-Eingang (AX-Adapter) – liefert das boolsche Eingangssignal und den Startimpuls
Socket	PT	adapter::types::unidirectional::ATM	Vorlaufzeit (AT-Adapter) – stellt die Verzögerungszeit bereit
Plug	Q	adapter::types::unidirectional::AX	Timer-Ausgang (AX-Adapter) – gibt den Timer-Zustand und die Bestätigung aus

Funktionsweise

Der **ATM_AX_TON** kapselt einen standardmäßigen On-Delay-Timer (E_TON) aus der IEC 61499-Bibliothek. Die Funktionsweise im Detail:

1. **Start der Zeitmessung:** Der Eingangsadapter **IN** liefert über sein Ereignis **E1** den Startimpuls (REQ) und über sein Datensignal **D1** den boolschen Wert **IN**. Wenn **IN.D1** auf WAHR wechselt, wird der Timer gestartet.
2. **Verzögerungszeit:** Die gewünschte Verzögerungszeit wird über den Adapter **PT** bereitgestellt (über **PT.D1**). Sobald der Timer läuft, wird nach Ablauf dieser Zeit der Ausgang **Q.D1** auf WAHR gesetzt.
3. **Ausgangssignal und Bestätigung:** Der Ausgangsadapter **Q** gibt über **Q.D1** den aktuellen Timer-Zustand aus (WAHR, wenn die Verzögerungszeit abgelaufen ist, FALSE, wenn der Eingang auf FALSE geht). Das Ereignis **Q.E1** wird nach jeder Zustandsänderung gesendet (CNF des internen E_TON).
4. **Setzen der Vorlaufzeit ohne Auslösung:** Der Ereignis-Eingang **EIPT** dient laut Spezifikation dazu, die Vorlaufzeit **PT** zu setzen, ohne den Timer zu starten. Im vorliegenden FBNetzwerk wird dieser Eingang jedoch nicht intern verbunden – er steht für externe Nutzung zur Verfügung, falls eine Anwendung eine Vorabsetzung der Zeit benötigt.
5. **Rückfall:** Wenn **IN** auf FALSE geht, wird der Timer sofort zurückgesetzt und **Q.D1** wird FALSE (unverzögert).

Technische Besonderheiten

- **Adapterbasierte Schnittstelle:** Der FB nutzt ausschließlich Adapter (AX und ATM) für die Kommunikation, keine traditionellen Ein-/Ausgangsgrooves. Dies ermöglicht eine flexible Kapselung und Wiederverwendung in komplexen Systemen.
- **Unidirektionale Adapter:** Die verwendeten Adapter sind unidirektional (AX für boolsche Werte und ATM für Zeitwerte). Der FB erwartet, dass die Adapter korrekt in die übergeordnete Hierarchie eingebunden werden.
- **Nicht verbundener Event-Eingang EIPT:** Der **EIPT**-Eingang ist definiert, aber nicht mit dem internen E_TON verbunden. Dies könnte auf eine geplante Erweiterung oder eine externe Verwendung durch den Anwender hindeuten. In der aktuellen Version hat er keine direkte Funktion im FBNetzwerk.
- **Autoren und Version:** Der Baustein wurde ursprünglich von Franz Höpfinger (HR Agrartechnik GmbH) entwickelt und später von Patrick Aigner überarbeitet (Version 3.0, April 2025). Er ist unter der Eclipse Public License 2.0 verfügbar.

Zustandsübersicht

Der FB selbst verwaltet keinen eigenen Zustandsautomaten – die Zustandslogik liegt im integrierten **E_TON**. Dennoch lässt sich das Verhalten wie folgt beschreiben:

Zustand des Eingangs IN	Verhalten	Ausgang Q.D1
FALSE → TRUE (steigende Flanke)	Timer startet; nach Ablauf von PT wird Q TRUE	FALSE (bis PT abgelaufen) → TRUE
TRUE (während Timer läuft)	Timer läuft weiter	FALSE
TRUE → FALSE (fallende Flanke)	Timer wird sofort zurückgesetzt	TRUE → FALSE
TRUE (nach Timerablauf)	Ausgang bleibt TRUE, solange IN TRUE	TRUE
FALSE (beliebig)	Ausgang bleibt FALSE	FALSE

Anwendungsszenarien

- **Maschinensteuerung:** Verzögerung beim Einschalten von Antrieben oder Ventilen, um ein gleichzeitiges Anlaufen mehrerer Komponenten zu verhindern.
- **Signalaufbereitung:** Entprellen von Schaltsignalen, indem ein kurzer Impuls erst nach einer definierten Zeit als gültig betrachtet wird.
- **Prozessautomatisierung:** Einleitung von Folgeschritten erst nach Ablauf einer Wartezeit (z. B. in Förderbändern oder Dosieranlagen).

- **Adapter-Integration:** Einsatz in Systemen, die bereits auf dem AX/AT-Adapterprotokoll basieren, ohne zusätzliche Datenleitungen.

Vergleich mit ähnlichen Bausteinen

Baustein	Typ	Besonderheit
E_TON (IEC 61499)	Standard On-Delay	Klassische Ein-/Ausgangsschnittstelle; direkte Ereignis- und Datenports
ATM_AX_TON	Adapter-basiert	Kapselt E_TON und bietet AX/AT-Adapter; zusätzlicher EIPT-Eingang (nicht verbunden)
E_TOF	Off-Delay	Verzögerung beim Ausschalten (entgegengesetztes Verhalten)
E_CTU	Aufwärtszähler	Zählt Ereignisse, kein Timer

Der **ATM_AX_TON** unterscheidet sich durch seine reine Adapterschnittstelle und die fehlende direkte Datenkopplung. Er eignet sich besonders für modulare, hierarchisch aufgebaute Steuerungen.

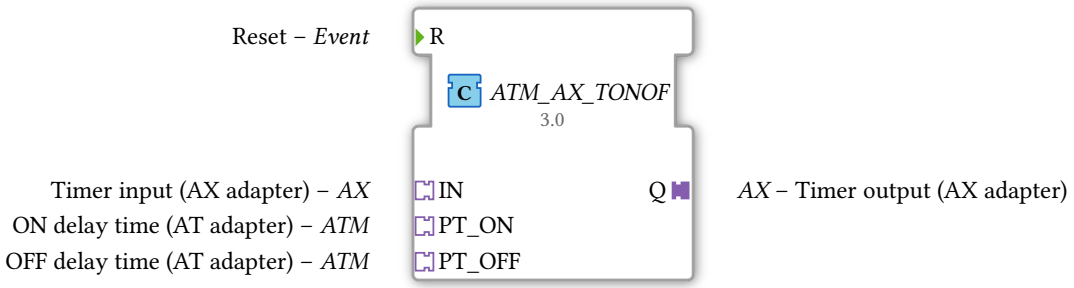
Fazit

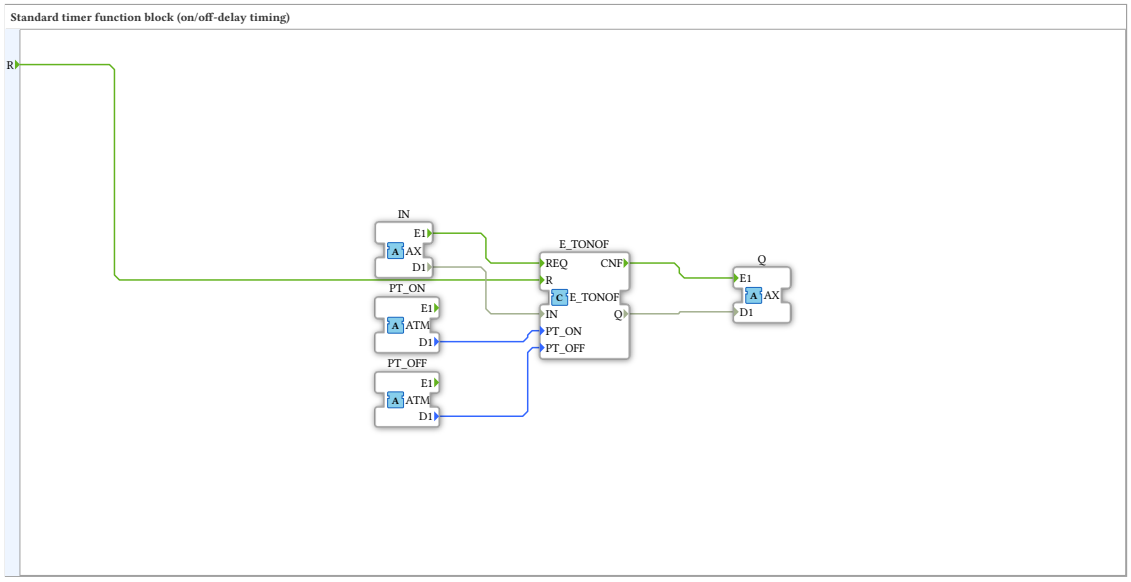
Der **ATM_AX_TON** ist ein fertig konfektionierter On-Delay-Timer mit einer modernen, adapterbasierten Anbindung. Er vereinfacht die Einbindung von Zeitverzögerungen in größere Systeme, die bereits das AX/AT-Konzept verwenden. Der nicht angeschlossene **EIPT**-Eingang bietet Potenzial für zukünftige Erweiterungen oder kundenspezifische Anpassungen. Insgesamt ist der Baustein robust, gut dokumentiert und unter freier Lizenz verfügbar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de

ATM_AX_TONOF





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ATM_AX_TONOF** realisiert einen kombinierten Ein-/Ausschaltverzögerungs-Timer. Er ist als Standard-Timerbaustein nach IEC 61499 ausgelegt und nutzt ausschließlich Adapter-Schnittstellen zur Kommunikation. Dadurch kann er flexibel in unterschiedliche Kommunikationsstrukturen eingebunden werden. Der Baustein verarbeitet ein boolesches Eingangssignal und gibt das verzögerte Signal aus – sowohl die Einschaltverzögerung (ON-Delay) als auch die Ausschaltverzögerung (OFF-Delay) werden über eigene Adapter konfiguriert.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung
R	Event	Setzt den Timer zurück. Beide laufenden Verzögerungen werden abgebrochen und der Ausgang Q wird sofort auf FALSE gesetzt.

Ereignis-Ausgänge

Der Funktionsblock besitzt keine eigenständigen Ereignis-Ausgänge. Das Ausgangsereignis wird über den Ausgangsadapter **Q** bereitgestellt (siehe Adapter).

Daten-Eingänge

Der FB verfügt über keine direkten Dateneingänge. Alle notwendigen Daten (boolescher Eingang und Verzögerungszeiten) werden über die Adapter-Schnittstellen eingelesen.

Daten-Ausgänge

Auch für Datenausgänge existieren keine eigenen Anschlüsse. Der aktuelle Ausgangszustand (BOOL) wird über den Ausgangsadapter **Q** ausgegeben.

Adapter

Name	Richtung	Typ	Beschreibung
IN	Socket	AX (unidirektional)	Eingangsadapter, der über sein Ereignis E1 die Ausführung anstößt und über sein Datum D1 den booleschen Eingangswert liefert.
PT_ON	Socket	ATM (unidirektional)	Adapter für die Einschaltverzögerungszeit (TIME). Der Wert wird über D1 bereitgestellt.

PT_OFF	Socket	ATM (unidirektional)	Adapter für die Ausschaltverzögerungszeit (TIME). Der Wert wird über D1 bereitgestellt.
Q	Plug	AX (unidirektional)	Ausgangsadapter. Das Ereignis E1 signalisiert die Aktualisierung des Ausgangswertes, das Datum D1 enthält den booleschen Zustand von Q (TRUE oder FALSE).

Funktionsweise

Der FB arbeitet nach folgendem Ablauf:

- Wenn das Eingangssignal **IN** von FALSE auf TRUE wechselt, startet der Timer die Einschaltverzögerung mit der Zeit **PT_ON**. Nach Ablauf dieser Zeit wird der Ausgang **Q** auf TRUE gesetzt.
- Wenn **IN** von TRUE auf FALSE wechselt, startet der Timer die Ausschaltverzögerung mit der Zeit **PT_OFF**. Nach Ablauf dieser Zeit wird **Q** auf FALSE gesetzt.
- Ein Ereignis am Reset-Eingang **R** stoppt beide laufenden Timer sofort und setzt **Q** auf FALSE, unabhängig vom aktuellen Zustand von **IN**.

Die interne Implementierung nutzt den standardisierten Baustein **E_TONOF** der IEC 61499-Bibliothek. Dies gewährleistet ein definiertes, robustes Zeitverhalten.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Alle Ein- und Ausgaben erfolgen über Adapter (AX, ATM) anstatt über diskrete Ereignis- und Datenports. Dies erleichtert die Integration in modulare Systeme und erlaubt eine saubere Trennung von Kommunikation und Logik.
- **Standardkonformität**: Der FB ist als Standard-Timer nach IEC 61499 klassifiziert und kann in beliebigen 4diac-IDE Projekten verwendet werden.
- **Reset-Funktion**: Der separate Reset-Eingang ermöglicht ein sofortiges Zurücksetzen ohne Abhängigkeit vom aktuellen Timer-Zustand.
- **Wiederverwendbare Adaptertypen**: Die Adapter AX und ATM werden in der Anwendung mehrfach verwendet, sodass keine zusätzlichen Typdefinitionen erforderlich sind.

Zustandsübersicht

Der interne Timer **E_TONOF** durchläuft folgende Zustände:

1. **IDLE**: Keine Verzögerung aktiv. Ausgang Q = Zustand von IN (sofern nach vorheriger Verzögerung aktualisiert).
2. **START_ON**: Nach positiver Flanke von IN – Timer für PT_ON läuft.
3. **START_OFF**: Nach negativer Flanke von IN – Timer für PT_OFF läuft.
4. **RESET**: Wird durch das Ereignis R ausgelöst, unterbricht jeden aktiven Timer und kehrt in den IDLE-Zustand mit Q = FALSE zurück.

Anwendungsszenarien

- **Verzögerte Ein- und Ausschaltung von Aktoren**: Beispielsweise zur Vermeidung von Prellern oder zur zeitlichen Staffelung von Schaltvorgängen.
- **Motorschutz**: Einschaltverzögerung verhindert häufiges Ein-/Ausschalten, Ausschaltverzögerung ermöglicht definierten Nachlauf.
- **Türsteuerung**: ON-Delay für zeitverzögertes Schließen nach Freigabe, OFF-Delay für automatisches Öffnen nach Ablauf einer Wartezeit.
- **HMI-Filterung**: Unterdrückung kurzer, unerwünschter Signalwechsel durch einstellbare Totzeiten.

Vergleich mit ähnlichen Bausteinen

- **E_TON (TON)**: Realisiert nur eine Einschaltverzögerung; der Ausgang wird nach Ablauf der Verzögerung auf TRUE gesetzt und bleibt TRUE, solange IN=TRUE ist.
- **E_TOF (TOF)**: Realisiert nur eine Ausschaltverzögerung; der Ausgang wird sofort bei IN=TRUE auf TRUE gesetzt und fällt nach Ablauf der Verzögerung nach IN=FALSE wieder auf FALSE.
- **E_TONOF**: Kombiniert beide Verzögerungen in einem Baustein. Der **ATM_AX_TONOF** erweitert dieses Konzept um eine Adapter-Schnittstelle, sodass die Anbindung an andere Adapter-basierte Komponenten erleichtert wird. Er bietet denselben Funktionsumfang, jedoch mit erhöhter Flexibilität in der Systemintegration.

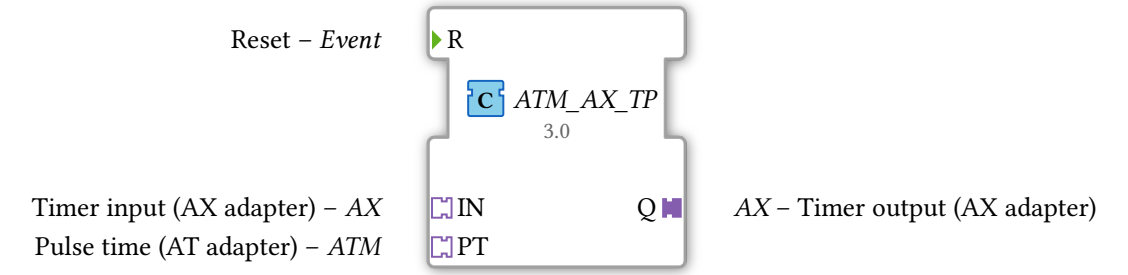
Fazit

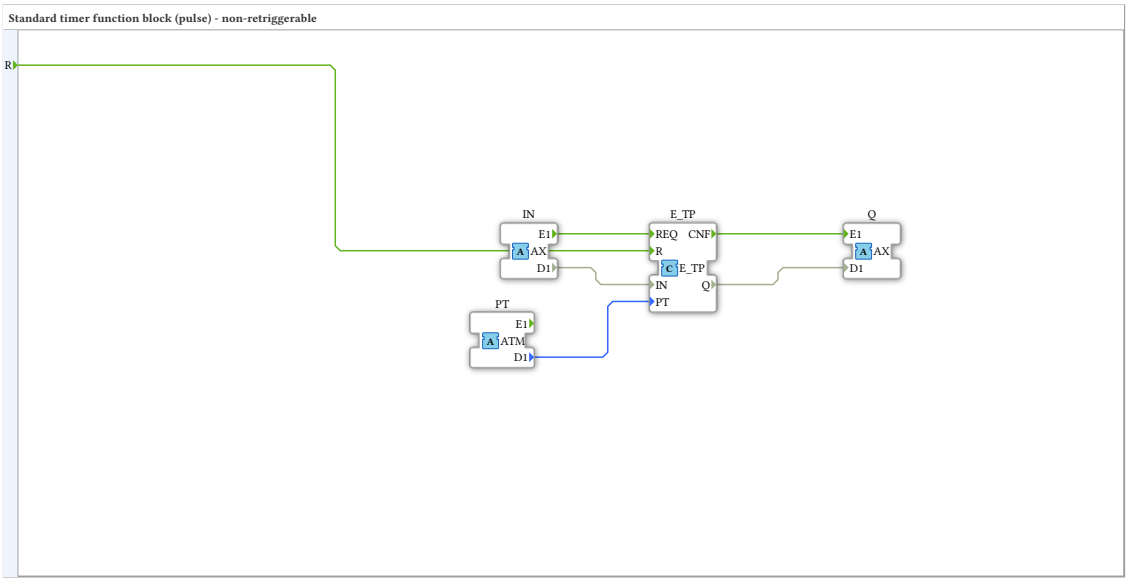
Der **ATM_AX_TONOF** ist ein leistungsfähiger Timer-Funktionsblock für verzögerte Ein-/Ausschaltvorgänge. Durch seine Adapter-basierte Schnittstelle passt er sich nahtlos in modulare Automatisierungslösungen ein und bleibt dennoch vollständig kompatibel zum IEC 61499-Standard. Die Kombination von ON-Delay und OFF-Delay in einem Baustein reduziert den Verdrahtungsaufwand und vereinfacht die Applikationserstellung. Dank des separaten Reset-Eingangs und der stabilen internen Implementierung eignet er sich für vielfältige industrielle Anwendungen.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ATM_AX_TP





• – ○ • – ○ • – ○ •

Einleitung

Der **ATM_AX_TP** ist ein spezieller Timer-Funktionsblock, der aus einem Eingangssignal einen zeitlich begrenzten Impuls erzeugt. Er implementiert die Funktionalität eines **nicht-nachtriggerbaren Impulsformers** (non-retriggerable pulse). Der Baustein wird über einen AX-Adapter angesteuert und gibt das Ergebnis ebenfalls über einen AX-Adapter aus; die Impulsdauer wird über einen separaten ATM-Adapter vorgegeben. Durch die Adapter-Schnittstellen lässt er sich besonders einfach in modular aufgebaute Automatisierungssysteme integrieren.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
R	Event	Setzt den Timer zurück

Ereignis-Ausgänge

Keine eigenen Ereignisausgänge – die Ausgabe des Impulses erfolgt über den Adapter **Q** (siehe Adapter-Sektion).

Daten-Eingänge

Keine direkten Dateneingänge. Die Eingangswerte (IN und PT) werden über die entsprechenden Adapter-Sockets bereitgestellt.

Daten-Ausgänge

Keine direkten Datenausgänge. Der Ausgangswert (Q) wird über den Adapter **Q** ausgegeben.

Adapter

Bezeichnung	Typ	Richtung	Beschreibung
Q	AX	Plug	Ausgang: liefert das Impulssignal
IN	AX	Socket	Eingang: Empfängt das auslösende Signal
PT	ATM	Socket	Eingang: Stellt die Impulsdauer (Pulszeit) bereit

Hinweis: Die Adapter sind vom Typ *adapter::types::unidirectional::AX* (bzw. *ATM*) und übertragen ein Ereignis zusammen mit einem Datenwert.

Funktionsweise

Der **ATM_AX_TP** arbeitet auf Basis des standardisierten IEC 61499-Funktionsbausteins **E_TP**, der im Inneren des Netzwerks instanziiert ist.

- **Auslösung:** Eine steigende Flanke am Eingang **IN** (Ereignis IN.E1 und Datenwert IN.D1 = TRUE) startet die Zeitmessung.
- **Impulserzeugung:** Der Ausgang **Q** wird unverzüglich auf TRUE gesetzt und bleibt für die Dauer **PT** aktiv. Nach Ablauf der Zeit kehrt Q auf FALSE zurück.
- **Nicht-nachtriggerbar:** Während eines laufenden Impulses wird eine erneute steigende Flanke von IN ignoriert – der Impuls läuft in der ursprünglichen Länge weiter.
- **Reset:** Ein Ereignis am Eingang **R** beendet den aktuellen Impuls sofort (Q = FALSE) und setzt den Timer zurück. Danach kann ein neuer Impuls gestartet werden.

Die Verdrahtung intern: `IN.E1 → E_TP.REQ` `IN.D1 → E_TP.IN` `PT.D1 → E_TP.PT` `E_TP.CNF → Q.E1` (Ereignisausgabe bei Impulsende) `E_TP.Q → Q.D1` (Ausgangsdatenwert) `R → E_TP.R`

Technische Besonderheiten

- **Adapter-basierte Anbindung:** Anders als ein direkter E_TP arbeitet ATM_AX_TP ausschließlich über Adapter (AX und ATM). Dies erlaubt eine lose Kopplung in verteilten Systemen und vereinfacht den Austausch von Signaltypen.
- **Kompatibilität:** Der Baustein ist für die Verwendung mit dem Eclipse 4diac-Framework optimiert (erkennbar am Package-Attribut `adapter::events::unidirectional::timers`).
- **Versionierung:** Mehrere Versionen dokumentieren die Evolution (Rücksetzfunktion, Paketänderung).
- **Lizenz:** Eclipse Public License 2.0 – der Quellcode darf verwendet, geändert und weitergegeben werden.

Zustandsübersicht

Der FB besitzt keine expliziten Zustandsautomaten auf der oberen Ebene – die Zustandslogik wird vollständig durch den internen **E_TP** realisiert. Dieser durchläuft folgende Zustände:

- **IDLE** – warten auf steigende Flanke von IN
- **ACTIVE** – Impuls läuft (Q = TRUE, Timer läuft)
- **RESET** – durch Ereignis R in IDLE zurück (Impuls vorzeitig beendet)

Nach Ablauf der Timer-Zeit wechselt E_TP selbstständig von ACTIVE nach IDLE.

Anwendungsszenarien

- **Steuerung kurzzeitiger Aktoren:** Ventile, Leuchten, Signalgeber, die nur für eine bestimmte Dauer eingeschaltet werden sollen.

- **Entprellung / Impulsformung:** Ein eingehendes Signal (z. B. Taster) wird in einen exakt definierten Impuls gewandelt.
- **Modulare Automatisierung:** Durch die Adapter-Schnittstellen eignet sich der Baustein für den Einsatz in verteilten Systemen, bei denen Signale über verschiedene Komponenten ausgetauscht werden.

Vergleich mit ähnlichen Bausteinen

Baustein	Eigenschaften	Unterschied zum ATM_AX_TP
E_TP (Standard)	Direkte Ereignis-/Datenanschlüsse, kein Adapter.	ATM_AX_TP kapselt E_TP und stellt Adapter bereit.
E_R_TRIG	Flankenerkennung ohne Impulsverlängerung.	Erzeugt nur einen sehr kurzen Impuls, keine Zeitsteuerung.
E_CYCLE	Zyklischer Timer, wiederholt Impulse.	Einmaliger Impuls (pulse).

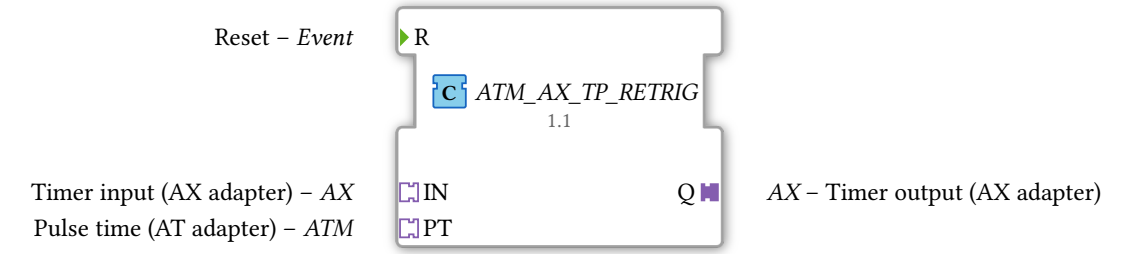
Fazit

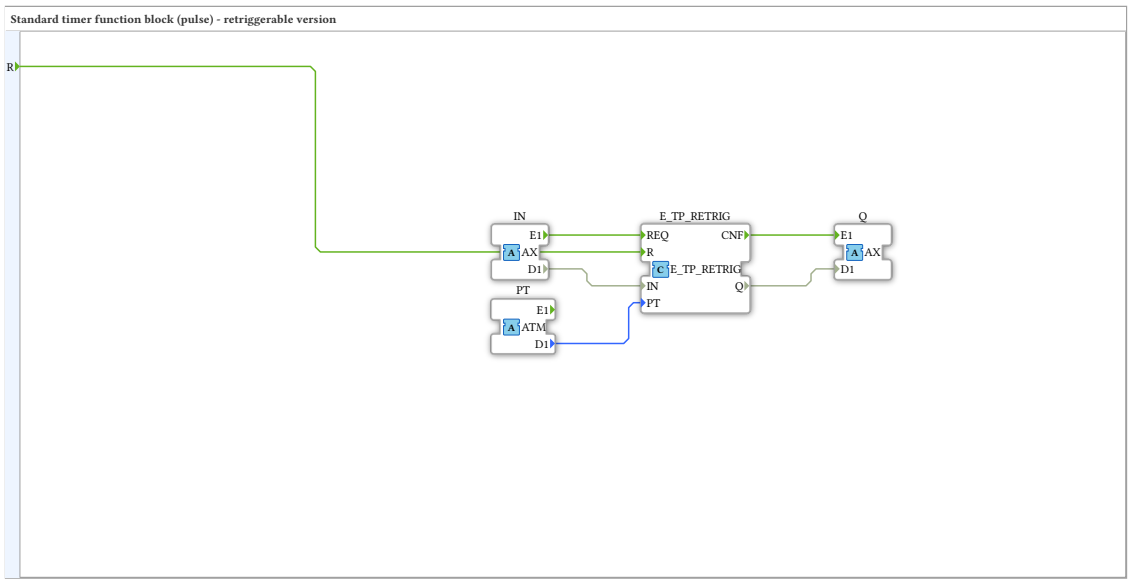
Der **ATM_AX_TP** ist ein nützlicher Funktionsbaustein zur Erzeugung eines nicht-nachtriggerbaren Impulses mit einstellbarer Dauer. Seine Besonderheit liegt in der Adapter-basierten Schnittstelle, die eine flexible und lose Kopplung in IEC 61499-Systemen ermöglicht. Durch den eingebetteten Standard-Timer E_TP bleibt die Funktionsweise robust und bewährt. Ein praktischer Baustein für alle Anwendungen, bei denen ein definiertes Zeitfenster auf ein externes Signal hin geschaltet werden muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ATM_AX_TP_RETRIG





• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ATM_AX_TP_RETRIG` ist ein retriggerbarer Impuls-Timer-Funktionsblock. Er erzeugt einen Impuls von definierter Dauer am Ausgang, wenn der Eingang auf TRUE wechselt. Während des laufenden Impulses können weitere Trigger den Timer neu starten (Retrigger-Funktion). Ein Reset-Eingang beendet den Impuls sofort. Der Baustein verwendet standardisierte Adapter für Ein- und Ausgang, was eine flexible Anbindung in modularen Steuerungssystemen ermöglicht.

Schnittstellenstruktur

Ereignis-Eingänge

- **R** (Event) – Reset: Setzt den Timer zurück und beendet den laufenden Impuls.

Ereignis-Ausgänge

- **Q.E1** (Event) – Ausgangs-Event: Wird bei Änderung des Impulszustands (Beginn oder Ende) über den Adapter Q ausgelöst.

Daten-Eingänge

- **IN.D1** (BOOL) – Triggereingang: Bei einer steigenden Flanke und TRUE-Wert startet der Timer.
- **PT.D1** (TIME) – Impulsdauer: Gibt die Zeit an, für die der Ausgang Q nach einem Trigger auf TRUE gesetzt wird.

Daten-Ausgänge

- **Q.D1** (BOOL) – Impulsausgang: Zeigt während der Impulsdauer TRUE an, sonst FALSE.

Adapter

- **IN** (Socket, Typ AX) – Eingangsadapter: Koppelt das Triggerevent (E1) und den boolschen Triggerwert (D1).
- **PT** (Socket, Typ ATM) – Zeitadapter: Liefert die Impulsdauer als TIME-Wert (D1).
- **Q** (Plug, Typ AX) – Ausgangsadapter: Stellt den Impulszustand (D1) und ein Ereignis bei Zustandsänderung (E1) bereit.

Funktionsweise

Der Baustein verwendet intern den Standard-Funktionsblock `E_TP_RETRIG`. Ein Ereignis am Eingang **IN** (steigende Flanke von IN.E1) startet den Timer, sofern **IN.D1** = TRUE ist. Der Ausgang **Q.D1** wird für die in **PT.D1** angegebene Zeit auf TRUE gesetzt. Während dieser Zeit löst ein erneuter Trigger (wiederholtes IN.E1 mit IN.D1 = TRUE) einen Retrigger aus: Der Timer wird zurückgesetzt und der Impuls beginnt von vorn. Ein Ereignis am Reset-Eingang **R** beendet den Impuls sofort, **Q.D1** wird FALSE. Nach Ablauf der Impulsdauer ohne erneuten Trigger wechselt **Q.D1** automatisch zurück auf FALSE.

Technische Besonderheiten

- **Retriggerbar**: Mehrfach-Trigger während eines laufenden Impulses starten den Timer neu, ohne den Impuls zu verlängern.
- **Adapterbasierte Schnittstelle**: Über die Adapter IN, PT und Q wird eine saubere Trennung von Ereignis- und Datenkommunikation erreicht.
- **Interner Standardbaustein**: Die eigentliche Timerlogik wird vom IEC 61499-konformen `E_TP_RETRIG` ausgeführt.
- **Unidirektionale Adapter**: Sowohl AX- als auch ATM-Adapter sind unidirektional ausgelegt, was die Datenflussrichtung klar definiert.

Zustandsübersicht

Der Baustein kennt zwei Hauptzustände:

Zustand	Q.D1	Beschreibung
IDLE	FALSE	Ruhezustand, kein Impuls aktiv.
TIMING	TRUE	Impuls läuft, Timer zählt die Dauer PT herab.

Übergänge:

- IDLE ⟶ TIMING: Trigger (IN.E1 mit IN.D1 = TRUE)

- TIMING ⟶ TIMING (Retrigger): Erneuter Trigger startet Timer neu.
- TIMING ⟶ IDLE: Timer abgelaufen oder Reset (R)
- IDLE ⟶ IDLE: Reset ohne Wirkung

Anwendungsszenarien

- **Impulssteuerung für Aktoren**: Ansteuerung von Ventilen, Lampen oder Motoren, die bei einem Startsignal für eine definierte Zeit aktiviert werden.
- **Sicherheitsanwendungen**: Retriggerbare Impulse ermöglichen eine erneute Anforderung während einer laufenden Aktion, z. B. bei Taster-Wiederholung.
- **Modulare Automatisierungssysteme**: Einsatz in Kombination mit anderen Adapter-basierten Komponenten zur einheitlichen Kommunikation über AX/ATM-Schnittstellen.

Vergleich mit ähnlichen Bausteinen

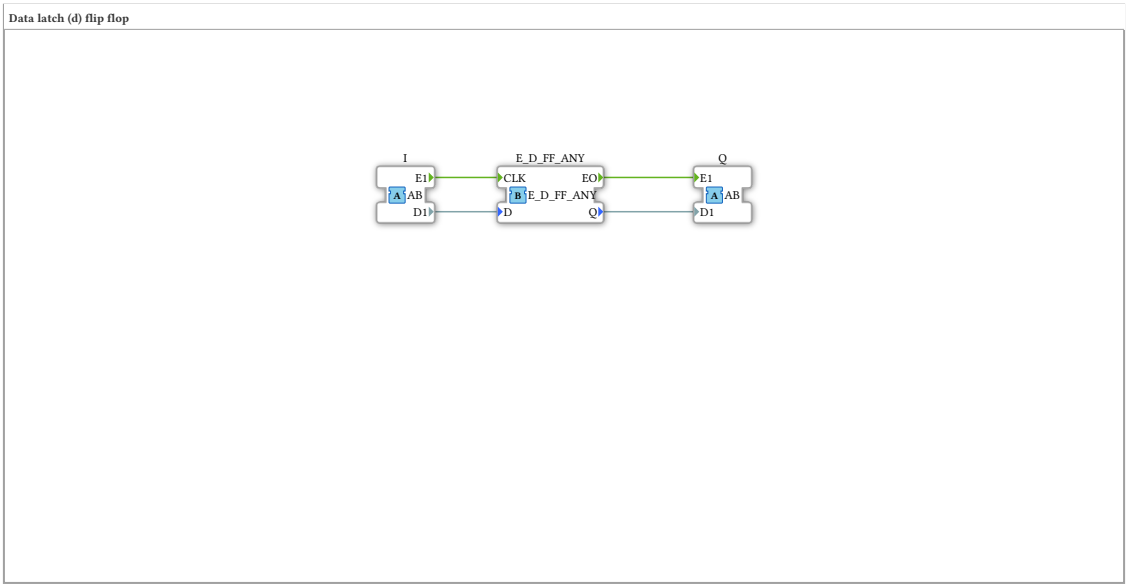
- *E_TP (nicht retriggerbar)*: Ignoriert einen erneuten Trigger während des laufenden Impulses; einmal gestartet, läuft der Impuls immer vollständig ab.
- *E_TP_RETRIG (Standard)*: Gleiche Timerlogik, jedoch ohne Adapter-Schnittstelle – direkt über Ereignis- und Dateneingänge angeschlossen.
- *TON / TOF*: Andere Timer-Typen; TON verzögert das Einschalten, TOF verzögert das Ausschalten – beide erzeugen keine Impulse.

Fazit

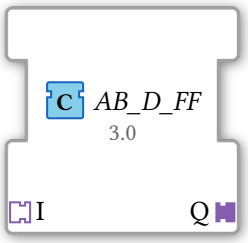
Der `ATM_AX_TP_RETRIG` kombiniert die bewährte retriggerbare Impulsfunktion mit einer adapterbasierten Schnittstelle. Durch die Verwendung standardisierter AX- und ATM-Adapter ist er besonders gut für modulare und wiederverwendbare Steuerungslösungen geeignet. Die Retrigger-Eigenschaft bietet Flexibilität in Anwendungen, bei denen ein Impuls bei erneuter Anforderung neu gestartet werden muss.

BYTE

AB_D_FF



Kein Bild vorhanden.



Value to latch – *AB*

AB – Latched value

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AB_D_FF** realisiert ein asynchrones Daten-Latch (D-Flipflop) auf Basis eines Adapter-Konzepts nach IEC 61499. Er übernimmt einen Datenwert über den Eingangsadapter **I**, sobald ein Ereignis anliegt, und gibt diesen Wert über den Ausgangsadapter **Q** aus. Der interne Zustand des Flipflops wird dabei durch einen standardisierten `E_D_FF_ANY`-Baustein abgebildet.

Schnittstellenstruktur

Ereignis-Eingänge

- **I.E1** (via Adapter `I`): Startet das Übernehmen des Datenwerts.

Ereignis-Ausgänge

- **Q.E1** (via Adapter `Q`): Signalisiert die Ausgabe des gelatchten Datenwerts.

Daten-Eingänge

- **I.D1** (via Adapter `I`): Der zu latchnde Datenwert (Typ beliebig, abhängig vom verwendeten Adaptertyp `adapter::types::unidirectional::AB`).

Daten-Ausgänge

- **Q.D1** (via Adapter `Q`): Der gelatchte und ausgegebene Datenwert.

Adapter

- **I** (Socket): Eingangsadapter vom Typ `adapter::types::unidirectional::AB` – nimmt den zu latchnden Wert und das Steuerereignis entgegen.
- **Q** (Plug): Ausgangsadapter vom selben Typ – gibt den gespeicherten Wert und das zugehörige Ereignis aus.

Funktionsweise

Der Funktionsblock arbeitet ereignisgesteuert:

1. Sobald am Eingangsadapter **I.E1** ein Ereignis eintrifft, wird der aktuelle Wert am Daten-Eingang **I.D1** in das interne D-Flipflop (`E_D_FF_ANY`) übernommen.
2. Anschließend wird der gespeicherte Wert an **Q.D1** ausgegeben und gleichzeitig ein Ereignis am Ausgang **Q.E1** erzeugt.

Somit verhält sich der Baustein wie ein klassischer flankengesteuerter D-Latch, bei dem die Übernahme und Ausgabe synchron zum Eingangsereignis erfolgen.

Technische Besonderheiten

- **Adapter-Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`adapter::types::unidirectional::AB`) für die Ein- und Ausgabe, was eine flexible Typisierung und lose Kopplung mit anderen Komponenten ermöglicht.
- **Interne Implementierung**: Der latchende Kern wird durch den standardisierten FB `E_D_FF_ANY` aus der IEC-61499-Bibliothek `iec61499::events` realisiert. Dieser speichert den Datenwert und erzeugt das Ausgangsereignis.
- **Ereignis- und Datenpfad** sind strikt getrennt: Das Ereignis löst die Datenübernahme aus, Daten fließen unabhängig.

Zustandsübersicht

Der Funktionsblock besitzt keinen expliziten internen Zustandsautomaten, sondern vertraut auf den impliziten Zustand des eingebetteten `E_D_FF_ANY`:

- **Idle**: Warten auf ein Ereignis an `I.E1`.
- **Latchen**: Nach Empfang des Ereignisses wird der Datenwert übernommen und ausgegeben.
- Der interne gemerkte Wert bleibt bis zum nächsten Ereignis erhalten.

Anwendungsszenarien

- **Datensynchronisation**: Sichere Übernahme von Messwerten oder Steuerdaten zu definierten Zeitpunkten (z. B. Taktimpuls).
- **Pufferung**: Zwischenspeichern von Werten in Ablaufsteuerungen zwischen verschiedenen Funktionsbausteinen.

- **Registerbildung**: Aufbau von Schieberegistern oder Speicherzellen in modularen Automatisierungssystemen.

Vergleich mit ähnlichen Bausteinen

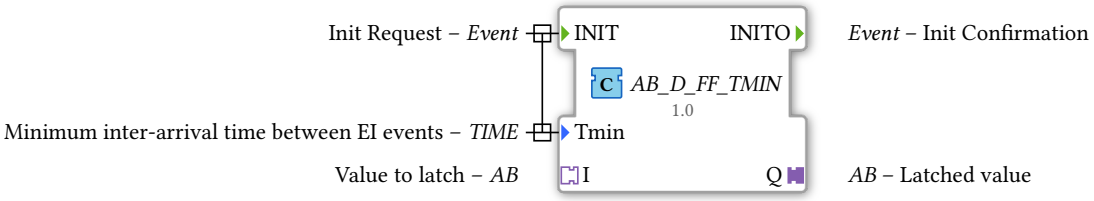
Baustein	Beschreibung
AB_SR_FF	Set-Reset-Flipflop, speichert zwei Zustände (Set/Reset) ohne Datenwert.
AB_T_FF	Toggle-Flipflop, wechselt den Ausgangszustand bei jedem Ereignis.
E_D_FF	Standard-D-Flipflop ohne Adapter-Schnittstelle, direkte Ereignis- und Daten-Pins.

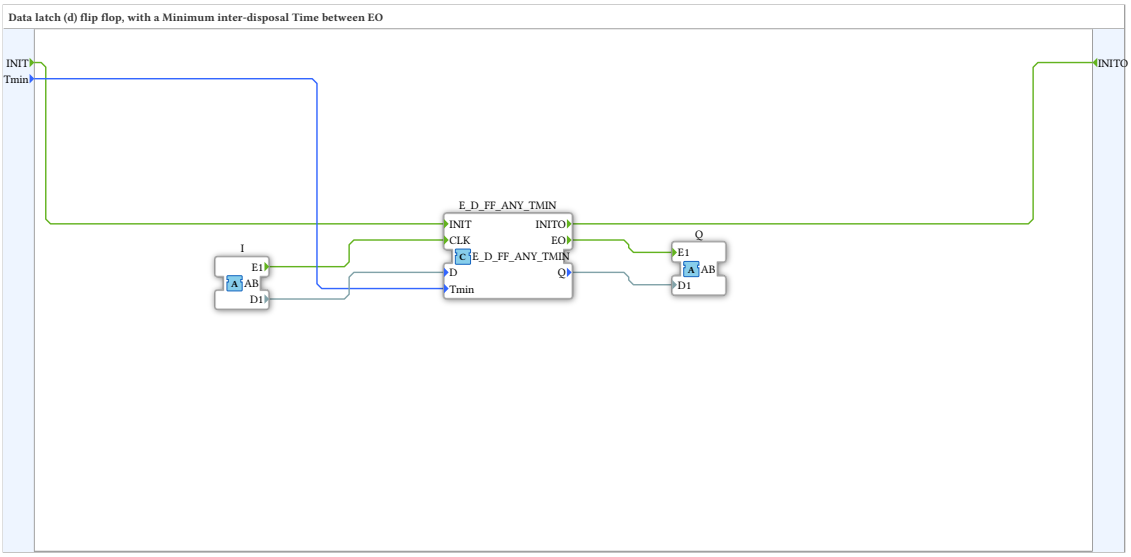
Der **AB_D_FF** bietet durch die Adapter-Schnittstelle eine typsichere und wiederverwendbare Kapselung, während vergleichbare Bausteine oft auf feste Datentypen angewiesen sind.

Fazit

Der **AB_D_FF** ist ein kompakter und flexibler D-Latch-Baustein, der die Vorteile der IEC-61499-Architektur nutzt. Durch die Adapter-Schnittstelle kann er in unterschiedlichsten Kontexten eingesetzt werden, ohne sich auf einen bestimmten Datentyp festlegen zu müssen. Zusammen mit dem internen Standard-Flipflop gewährleistet er eine zuverlässige und ereignisgesteuerte Datenhaltung.

AB_D_FF_TMIN





• – ○ • – ○ • – ○ •

Einleitung

Der **AB_D_FF_TMIN** ist ein Adapter-basierter Funktionsbaustein (FB) nach IEC 61499, der die Funktionalität eines Data-Latch (D-Flipflop) mit einer minimalen Zeitbedingung zwischen aufeinanderfolgenden Ausgangsereignissen realisiert. Er dient dazu, einen digitalen Datenwert von einem Adapter-Socket (Eingang) auf einen Adapter-Plug (Ausgang) zu übertragen und dabei sicherzustellen, dass die Ausgangsereignisse (EO) nicht schneller als durch einen konfigurierten Zeitparameter `Tmin` erlaubt auftreten. Der Baustein abstrahiert die reine D-Flipflop-Logik durch die Nutzung eines internen Flipflop-Bausteins und kapselt Daten und Ereignisse in Adapter-Schnittstellen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
`INIT`	EInit	Initialisierungsanforderung (setzt die minimale Zeit)

Ereignis-Ausgänge

Name	Typ	Kommentar
`INITO`	EInit	Initialisierungsbestätigung

Daten-Eingänge

Name	Typ	Kommentar
`Tmin`	TIME	Mindestzeit zwischen zwei Ausgangsereignissen EO

Daten-Ausgänge

Keine direkten Datenausgänge; die Datenausgabe erfolgt über den Adapter-Plug `Q`.

Adapter

Richtung	Name	Typ	Kommentar
Socket	`I`	adapter::types::unidirectional::AB	Eingangsadapter: liefert Ereignis `E1` und Datenwert `D1` (der zu latchende Wert)
Plug	`Q`	adapter::types::unidirectional::AB	Ausgangsadapter: gibt den gelatchten Datenwert `D1` sowie das Ereignis `E1` aus

Hinweis: Der Adapter `unidirectional::AB` definiert ein Ereignis `E1` und einen Datenwert `D1` (typischerweise ein boolescher oder beliebiger generischer Typ). Die genaue Typdefinition des Adapters liegt außerhalb dieses Bausteins.

Funktionsweise

Der Baustein arbeitet intern mit dem FB `E_D_FF_ANY_TMIN` (Typ `iec61499::events::E_D_FF_ANY_TMIN`), der die Kernlogik des D-Flipflops mit einer minimalen Haltezeit zwischen Ausgangsereignissen realisiert. Die Funktionsweise gliedert sich wie folgt:

1. **Initialisierung:**
   Durch ein Ereignis an `INIT` wird der Parameter `Tmin` (Mindestzeit) an den inneren FB übergeben. Der Baustein quittiert dies mit `INITO`.
2. **Datenerfassung und -weitergabe:**
   - Am Socket `I` tritt ein Ereignis `E1` auf, das den Datenwert `D1` (das zu latchende Signal) transportiert.
   - Dieses Ereignis wird als `CLK` an den inneren Flipflop-Baustein weitergeleitet, der gleichzeitig den Datenwert `D1` von `I.D1` aufnimmt.
   - Der innere Flipflop speichert den Wert `D1` und gibt ihn an seinem Ausgang `Q` aus.
   - Der gelatchte Wert wird über den Plug `Q.D1` nach außen gegeben, und zeitgleich wird das Ausgangsereignis `EO` des inneren FBs ausgelöst.
   - Dieses `EO` wird an `Q.E1` weitergeleitet, sodass der empfangende Adapter über die Aktualisierung informiert wird.
3. **Zeitsteuerung:**
   Der innere FB `E_D_FF_ANY_TMIN` stellt sicher, dass nach einem Ausgangsereignis `EO` für die Dauer von `Tmin` kein weiteres Ausgangsereignis erzeugt werden kann – unabhängig davon, wie schnell Eingangsereignisse eintreffen. Überschreitet die Zeit zwischen zwei `E1`-Eingängen die `Tmin`-Schranke, wird der Wert sofort übernommen, andernfalls wird er blockiert, bis die Mindestzeit abgelaufen ist.

Technische Besonderheiten

- **Adapter-basierte Schnittstelle:**
  Statt einzelner Ereignis-/Datenports verwendet der Baustein Adapter, die Ereignis und Daten als gekapselte Einheit transportieren. Dies erhöht die Wiederverwendbarkeit und reduziert die Anzahl äußerer Verbindungen.
- **Minimale Ausgabezeit (Tmin):**
  Die Einstellung `Tmin` verhindert ein zu schnelles Auslösen von Ausgangsereignissen, was in zeitkritischen Steuerungen (z. B. zur Entprellung oder zur Einhaltung von Verarbeitungszyklen) nützlich ist.
- **Interne Nutzung eines spezialisierten FB:**
  Der Baustein delegiert die eigentliche Flipflop-Logik an den systeminternen FB `E_D_FF_ANY_TMIN`, der für die korrekte zeitliche Steuerung zuständig ist. Dies hält den äußeren FB schlank und fehlerfrei.
- **Typisierter Adapter:**
  Der Adaptertyp `adapter::types::unidirectional::AB` ist als unidirektional definiert: Daten und Ereignisse fließen nur in eine Richtung (vom Socket zum Plug). Dadurch ist der FB strikt aufwärtskompatibel.

Zustandsübersicht

Der Baustein besitzt keinen expliziten sichtbaren Zustandsautomaten; der Zustand ergibt sich aus dem inneren Flipflop und der Zeitsteuerung:

- **Zustand 0 (initial / nicht gelatcht):** Nach INIT oder nach einem Reset (nicht explizit vorhanden) liegt undefinierter Zustand vor. Der Wert `Q.D1` wird erst nach dem ersten gültigen `E1` am Socket gesetzt.
- **Zustand 1 (gelatcht und aktiv):** Der innere Flipflop hat einen Datenwert gespeichert. Nach einem Ausgangsereignis wird für die Dauer `Tmin` ein Sperrzustand eingehalten, in dem eingehende `E1`-Ereignisse ignoriert oder verzögert werden. Nach Ablauf der Mindestzeit ist der FB wieder bereit für die nächste Übernahme.

(Im XML ist kein SFC oder ECC definiert; die Zustandslogik liegt vollständig im internen Baustein.)

Anwendungsszenarien

- **Entprellung von Signalen:**
  Wenn ein Sensor oder eine Taste mehrere schnelle Impulse erzeugt (Prellen), kann der FB durch Setzen von `Tmin` auf z. B. 20 ms verhindern, dass mehrere Werte in kurzer Folge weitergereicht werden.
- **Zyklische Datenweitergabe mit Mindestabstand:**
  In einer verteilten Steuerung soll ein Messwert nur in bestimmten Mindestintervallen an eine übergeordnete Komponente gesendet werden. Der FB dient als Puffer, der die Aktualisierungsrate begrenzt.
- **Sichere Übergabe in zeitkritischen Systemen:**
  Bei der Übergabe eines Wertes von einem schnellen in einen langsameren Taktbereich kann der FB mit geeignetem `Tmin` Datenverlust oder Rauschen verhindern.

Vergleich mit ähnlichen Bausteinen

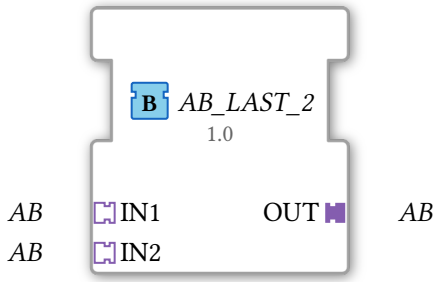
Baustein	Eigenschaften
AB_D_FF	Reines D-Flipflop ohne minimale Ausgabezeit – Ereignisse werden sofort weitergegeben.
E_D_FF_ANY	Standard-Ereignis-Daten-Flipflop ohne Zeitbeschränkung, aber mit diskreten Ports (keine Adapter).
AB_D_FF_TMIN	Kombiniert Adapter-Kapselung mit einer konfigurierbaren Mindestzeit zwischen Ausgangsereignissen.
R_TRIG / F_TRIG	Detektieren von steigenden/fallenden Flanken – keine Funktionalität zum Latchen eines Datenwerts.

Der **AB_D_FF_TMIN** hebt sich durch die Integration einer Zeitsteuerung in eine Adapter-basierte D-Flipflop-Logik ab und ist daher besonders für Anwendungen geeignet, die sowohl Datenlatching als auch zeitliche Entkopplung erfordern.

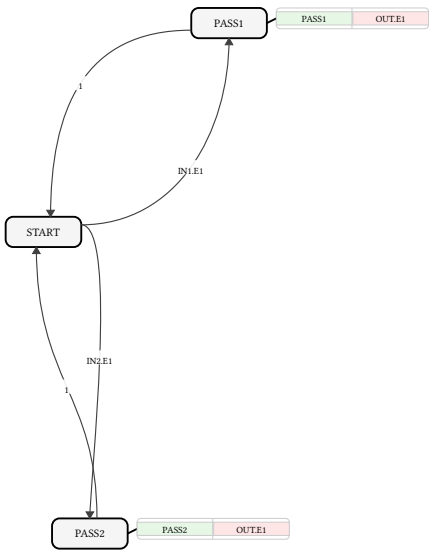
Fazit

Der **AB_D_FF_TMIN** ist ein kompakter, adapterbasierter Funktionsbaustein, der die Grundfunktion eines D-Flipflops um die Fähigkeit erweitert, eine minimale Wartezeit zwischen Ausgangsereignissen einzuhalten. Durch die gekapselte Adapter-Schnittstelle und die interne Nutzung eines spezialisierten Flipflop-Bausteins wird eine hohe Wiederverwendbarkeit und klare Trennung von Logik und Zeitsteuerung erreicht. Der Baustein eignet sich insbesondere für industrielle Steuerungsanwendungen, bei denen Signalverarbeitung mit zeitlichen Randbedingungen erforderlich ist.

AB_LAST_2



ECC: AB_LAST_2



Einleitung

Der AB_LAST_2 Funktionsblock führt zwei gleichartige BYTE-Adapter-Signale (**IN1**, **IN2**) auf einen gemeinsamen Ausgang (**OUT**) zusammen, nach dem Prinzip **Last Writer Wins**: welcher der beiden Sockets zuletzt ein Ereignis geschrieben hat, gewinnt, und sein Datenwert wird sofort und unverändert an OUT durchgereicht. Es handelt sich ausdrücklich nicht um ein naives Zusammenführen (z. B. Mittelwert oder ODER-Verknüpfung), sondern um eine reine Arbitrierung nach zeitlicher Reihenfolge.

Schnittstellenstruktur

Adapter

Eingangsadapter (Sockets):

- **IN1**: BYTE-Adapter (unidirectional) - erste Signalquelle
- **IN2**: BYTE-Adapter (unidirectional) - zweite Signalquelle

Ausgangsadapter (Plug):

- **OUT**: BYTE-Adapter (unidirectional) - zusammengeführtes Signal

Funktionsweise

Der Baustein ist als einfacher Basic-FB mit einer 3-Zustands-ECC umgesetzt (START, PASS1, PASS2). Trifft an IN1 ein Ereignis ein, wechselt die ECC nach PASS1: der aktuelle Datenwert von IN1 wird nach OUT kopiert und das Adapter-Ereignis an OUT gesendet. Trifft an IN2 ein Ereignis ein, geschieht dasselbe symmetrisch über PASS2. Nach jeder Durchreichung kehrt die ECC sofort wieder nach START zurück und ist bereit für das nächste Ereignis von IN1 oder IN2.

Technische Besonderheiten

- Last-Writer-Wins-Semantik statt naiver Datenverknüpfung
- Einfacher Basic-FB (ECC mit 3 Zuständen), kein zusätzlicher Zwischenspeicher nötig
- Keine Signalverzögerung: der Wert wird im selben Zyklus durchgereicht, in dem das auslösende Ereignis eintrifft
- Beide Eingänge sind gleichberechtigt - es gibt keine feste Priorität außer der zeitlichen Reihenfolge

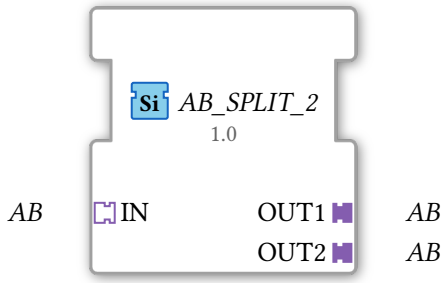
Anwendungsszenarien

- Zusammenführen zweier gleichwertiger BYTE-Signalquellen zu einem einzigen nachgeschalteten Adapterpfad (z. B. zwei redundante Sensoren, oder ein manueller und ein automatischer Sollwertgeber)
- Arbitrierung zwischen zwei Bedienstellen, die denselben BYTE-Wert schreiben dürfen
- Vereinfachung von Netzwerken, die sonst ein `F_SEL`/`E_RS`-Konstrukt inline verdrahten müssten

⚖ Vergleich mit ähnlichen Bausteinen

AB_LAST_2 ist die Umkehrung von `AB_SPLIT_2`: AB_SPLIT_2 verteilt 1 Eingang auf 2 Ausgänge, AB_LAST_2 führt 2 Eingänge auf 1 Ausgang zusammen. Für Mischbetrieb zwischen einem Daten- und einem reinen Ereignisadapter (z. B. AX + AE) ist stattdessen `AX_AE_MERGE` gedacht, das nur das Ereignis mischt, ohne den Datenwert zu verändern.

AB_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AB_SPLIT_2** dient der Aufteilung eines unidirektionalen Adapters vom Typ `AB` auf zwei separate Adapterausgänge. Er ist als generischer Baustein (Generic FB) implementiert und ermöglicht die gleichzeitige Weiterleitung des eingehenden Adapters an zwei unabhängige Zieladressaten. Dies vereinfacht die Topologie in Steuerungsapplikationen, wenn ein Signal mehrfach benötigt wird, ohne den ursprünglichen Datenfluss zu beeinträchtigen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden. Die Adapterkommunikation erfolgt ereignislos über den Datenaustauschmechanismus des Adaptertyps.

Daten-Eingänge

Keine Daten-Eingänge vorhanden. Der Datenfluss erfolgt ausschließlich über den Adapter-Eingang `IN`.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden. Die Ausgaben erfolgen über die Adapter-Ausgänge `OUT1` und `OUT2`.

Adapter

Rolle	Name	Typ	Richtung
Socket	IN	`adapter::types::unidirectional::AB`	Eingang
Plug	OUT1	`adapter::types::unidirectional::AB`	Ausgang
Plug	OUT2	`adapter::types::unidirectional::AB`	Ausgang

Funktionsweise

Der Baustein leitet den über den Socket `IN` verbundenen Adapter unverändert an beide Plugs `OUT1` und `OUT2` weiter. Jede Veränderung der Daten bzw. des Zustands des eingehenden Adapters wird simultan an beide Ausgänge propagiert. Die Aufteilung erfolgt ohne Verzögerung oder Pufferung – es handelt sich um eine reine Verteilungslogik.

Technische Besonderheiten

- **Generischer Baustein:** Der FB ist als generischer Typ deklariert (`GEN_AB_SPLIT`), sodass er mit verschiedenen Untertypen des Adapters `AB` arbeiten kann, sofern diese die gleiche unidirektionale Semantik besitzen.
- **Unidirektionalität:** Der Adapter ist ausschließlich in eine Richtung ausgelegt – Daten fließen vom Socket zu den Plugs. Rückwärtige Kommunikation wird nicht unterstützt.
- **Lizenz:** Der Baustein steht unter der Eclipse Public License 2.0 (EPL-2.0), was die freie Nutzung, Modifikation und Weitergabe in eigenen Projekten erlaubt.
- **Keine Zustandsautomaten:** Der FB besitzt keine inneren Zustände oder ereignisgesteuerten Abläufe, da die Adapterweiterleitung vollständig datengetrieben erfolgt.

Zustandsübersicht

Der FB besitzt keinen Zustandsautomaten. Die Ausgangsadapter spiegeln stets den aktuellen Zustand des Eingangsadapters wider.

Anwendungsszenarien

- **Signalverteilung in modularen Anlagen:** Trennung eines Sensorsignals von einer Feldbuskomponente zur parallelen Verarbeitung in zwei unabhängigen Steuerungslogiken.
- **Redundanzaufbau:** Ein eingehendes Signal kann über zwei separate Pfade an verschiedene Regler oder Überwachungseinheiten gesendet werden.
- **Prototypen und Testaufbauten:** Einfaches Duplizieren eines Adapterstroms für Simulations- und Debugzwecke, ohne die eigentliche Applikation zu verändern.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu dedizierten **SPLIT-FBs** mit Ereignis-/Datenports arbeitet `AB_SPLIT_2` rein adapterbasiert. Während klassische Split-Bausteine oft getriggerte Datenkopien erfordern, erfolgt die Verteilung hier kontinuierlich und ohne explizite Aktivierung. Ähnliche Funktionalität bieten Adapter-Multiplexer oder Buskoppler, jedoch mit komplexerer Konfiguration. Der vorliegende Baustein ist speziell auf den unidirektionalen `AB`-Adapter zugeschnitten.

- `AB_SPLIT_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

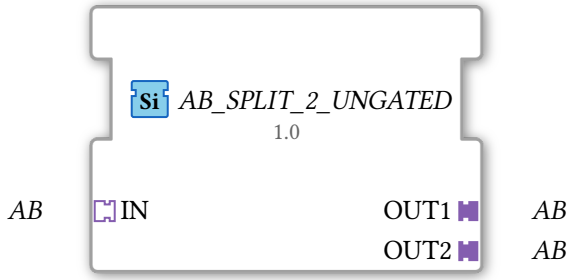
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

`AB_SPLIT_2` ist ein einfacher, aber effektiver generischer Funktionsblock zur Aufteilung eines unidirektionalen Adapters auf zwei Ausgänge. Er erweitert die Flexibilität in der Adapterverkabelung industrieller Steuerungssysteme gemäß IEC 61499 und ist dank der EPL-2.0-Lizenz frei einsetzbar. Für Anwendungen, die eine 1:2-Verteilung von Adapterdaten benötigen, bietet er eine saubere und wartbare Lösung.

AB_SPLIT_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AB_SPLIT_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AB_SPLIT_2_UNGATED** dient der Aufteilung eines unidirektionalen Adapters vom Typ `AB` auf zwei separate Adapterausgänge. Er ist als generischer Baustein (Generic FB) implementiert und ermöglicht die gleichzeitige Weiterleitung des eingehenden Adapters an zwei unabhängige Zieladressaten. Dies vereinfacht die Topologie in Steuerungsapplikationen, wenn ein Signal mehrfach benötigt wird, ohne den ursprünglichen Datenfluss zu beeinträchtigen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden. Die Adapterkommunikation erfolgt ereignislos über den Datenaustauschmechanismus des Adaptertyps.

Daten-Eingänge

Keine Daten-Eingänge vorhanden. Der Datenfluss erfolgt ausschließlich über den Adapter-Eingang `IN`.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden. Die Ausgaben erfolgen über die Adapter-Ausgänge `OUT1` und `OUT2`.

Adapter

Rolle	Name	Typ	Richtung
Socket	IN	`adapter::types::unidirectional::AB`	Eingang
Plug	OUT1	`adapter::types::unidirectional::AB`	Ausgang
Plug	OUT2	`adapter::types::unidirectional::AB`	Ausgang

Funktionsweise

Der Baustein leitet den über den Socket `IN` verbundenen Adapter unverändert an beide Plugs `OUT1` und `OUT2` weiter. Jede Veränderung der Daten bzw. des Zustands des eingehenden Adapters wird simultan an beide Ausgänge propagiert. Die Aufteilung erfolgt ohne Verzögerung oder Pufferung – es handelt sich um eine reine Verteilungslogik.

Technische Besonderheiten

- **Generischer Baustein:** Der FB ist als generischer Typ deklariert (`GEN_AB_SPLIT`), sodass er mit verschiedenen Untertypen des Adapters `AB` arbeiten kann, sofern diese die gleiche unidirektionale Semantik besitzen.
- **Unidirektionalität:** Der Adapter ist ausschließlich in eine Richtung ausgelegt – Daten fließen vom Socket zu den Plugs. Rückwärtige Kommunikation wird nicht unterstützt.
- **Lizenz:** Der Baustein steht unter der Eclipse Public License 2.0 (EPL-2.0), was die freie Nutzung, Modifikation und Weitergabe in eigenen Projekten erlaubt.
- **Keine Zustandsautomaten:** Der FB besitzt keine inneren Zustände oder ereignisgesteuerten Abläufe, da die Adapterweiterleitung vollständig datengetrieben erfolgt.

Zustandsübersicht

Der FB besitzt keinen Zustandsautomaten. Die Ausgangsadapter spiegeln stets den aktuellen Zustand des Eingangsadapters wider.

Anwendungsszenarien

- **Signalverteilung in modularen Anlagen:** Trennung eines Sensorsignals von einer Feldbuskomponente zur parallelen Verarbeitung in zwei unabhängigen Steuerungslogiken.
- **Redundanzaufbau:** Ein eingehendes Signal kann über zwei separate Pfade an verschiedene Regler oder Überwachungseinheiten gesendet werden.
- **Prototypen und Testaufbauten:** Einfaches Duplizieren eines Adapterstroms für Simulations- und Debugzwecke, ohne die eigentliche Applikation zu verändern.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu dedizierten **SPLIT-FBs** mit Ereignis-/Datenports arbeitet `AB_SPLIT_2_UNGATED` rein adapterbasiert. Während klassische Split-Bausteine oft getriggerte Datenkopien erfordern, erfolgt die Verteilung hier kontinuierlich und ohne explizite Aktivierung. Ähnliche Funktionalität bieten Adapter-Multiplexer oder Buskoppler, jedoch mit komplexerer Konfiguration. Der vorliegende Baustein ist speziell auf den unidirektionalen `AB`-Adapter zugeschnitten.

- `AB_SPLIT_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

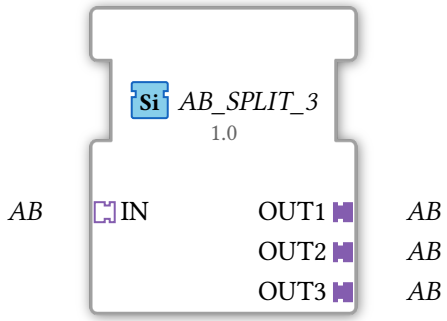
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

`AB_SPLIT_2_UNGATED` ist ein einfacher, aber effektiver generischer Funktionsblock zur Aufteilung eines unidirektionalen Adapters auf zwei Ausgänge. Er erweitert die Flexibilität in der Adapterverkabelung industrieller Steuerungssysteme gemäß IEC 61499 und ist dank der EPL-2.0-Lizenz frei einsetzbar. Für Anwendungen, die eine 1:2-Verteilung von Adapterdaten benötigen, bietet er eine saubere und wartbare Lösung.

AB_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AB_SPLIT_3** dient dazu, einen eingehenden unidirektionalen Daten-/Signal-Adapter (Typ `AB`) auf drei identische Ausgangs-Adapter aufzuteilen. Er ist als generischer Baustein implementiert und stellt eine einfache 1-zu-3-Verteilung ohne zusätzliche Logik oder Verzögerung dar.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AB`	`IN`	Socket (Eingang)	Empfängt einen unidirektionalen Adapter vom Typ AB.
`adapter::types::unidirectional::AB`	`OUT1`	Plug (Ausgang)	Erste Ausgabe des eingehenden Signals.
`adapter::types::unidirectional::AB`	`OUT2`	Plug (Ausgang)	Zweite Ausgabe des eingehenden Signals.
`adapter::types::unidirectional::AB`	`OUT3`	Plug (Ausgang)	Dritte Ausgabe des eingehenden Signals.

Funktionsweise

Der Baustein leitet den am Socket `IN` anliegenden unidirektionalen Adapter **ohne Manipulation** parallel an die drei Plugs `OUT1`, `OUT2` und `OUT3` weiter. Es findet keine Pufferung, Verarbeitung oder zeitliche Steuerung statt. Der Baustein fungiert als reiner **Splitter** (Verteiler) für adapterbasierte Signale.

Da der Adapter `AB` unidirektional ist, erfolgt die Datenübertragung ausschließlich in eine Richtung (vom Socket zu den Plugs). Die drei Ausgänge sind voneinander unabhängig und liefern stets den aktuellen Zustand des Eingangs.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist durch das Attribut `eclipse4diac::core::GenericClassName` als generisch markiert (`'GEN_AB_SPLIT'`). Das bedeutet, der tatsächliche Adapter-Typ kann zur Laufzeit

oder beim Instanziieren spezifiziert werden, solange er die Grundstruktur eines unidirektionalen AB-Adapters besitzt.
- **Unidirektionalität**: Sowohl Eingangs- als auch Ausgangs-Adapter sind vom Typ `unidirectional::AB`. Eine Rückwärtskommunikation ist nicht vorgesehen.
- **Keine Zustandshaltung**: Der Baustein besitzt keine interne Zustandsmaschine oder Speicher. Die Weiterleitung erfolgt passiv und kontinuierlich.
- **Copyright**: Der Baustein wurde von der HR Agrartechnik GmbH unter der Eclipse Public License 2.0 veröffentlicht.

Zustandsübersicht

Der Funktionsblock besitzt **keine explizite Zustandsmaschine**. Das Verhalten ist rein datenflussorientiert: Sobald der Eingangs-Adapter Daten liefert, werden diese unmittelbar an alle Ausgänge propagiert. Eine Zustandsüberwachung oder -steuerung ist nicht erforderlich.

Anwendungsszenarien

- **Signalverteilung in der Automatisierungstechnik**: Aufteilen eines Sensorsignals (z. B. Temperatur, Druck) an mehrere Verarbeitungseinheiten.
- **Parallelisierung von Steuerungsabläufen**: Bereitstellen desselben Adapter-Streams für verschiedene Funktionsblöcke, die unabhängig voneinander arbeiten.
- **Prototyping und Test**: Verwenden eines simulierten Adapters, um mehrere Systemkomponenten gleichzeitig zu versorgen.

Vergleich mit ähnlichen Bausteinen

Baustein	Verhalten	Besonderheit
`SPLIT_2`	1-zu-2-Verteilung	Analoger Baustein mit zwei Ausgängen.
`AB_SPLIT_3`	1-zu-3-Verteilung	Dieser Baustein.
`AB_JOIN`	Zusammenführen mehrerer Adapter	Gegenstück zum Splitter.

Während `SPLIT_2` nur zwei Ausgänge bietet, deckt `AB_SPLIT_3` den Bedarf für eine dreifache Verteilung ab. Es sind keine weiteren logischen Funktionen wie Filterung oder Verzögerung implementiert.

- `AB_SPLIT_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der **AB_SPLIT_3** ist ein einfacher, aber nützlicher generischer Splitter-Baustein für unidirektionale Adapter. Er ermöglicht eine saubere Strukturierung von Automatisierungsanwendungen, indem ein eingehender Signalpfad auf drei parallele Pfade aufgeteilt wird. Dank seiner generischen Natur kann er für beliebige AB-Adapter-Typen eingesetzt werden und fügt sich nahtlos in die IEC 61499-2-konforme Entwicklungsumgebung 4diac ein.

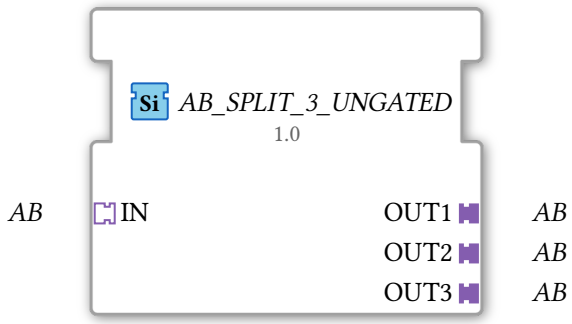
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AB_SPLIT_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AB_SPLIT_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht

gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AB_SPLIT_3_UNGATED** dient dazu, einen eingehenden unidirektionalen Daten-/Signal-Adapter (Typ `AB`) auf drei identische Ausgangs-Adapter aufzuteilen. Er ist als generischer Baustein implementiert und stellt eine einfache 1-zu-3-Verteilung ohne zusätzliche Logik oder Verzögerung dar.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AB`	`IN`	Socket (Eingang)	Empfängt einen unidirektionalen Adapter vom Typ AB.
`adapter::types::unidirectional::AB`	`OUT1`	Plug (Ausgang)	Erste Ausgabe des eingehenden Signals.
`adapter::types::unidirectional::AB`	`OUT2`	Plug (Ausgang)	Zweite Ausgabe des eingehenden Signals.
`adapter::types::unidirectional::AB`	`OUT3`	Plug (Ausgang)	Dritte Ausgabe des eingehenden Signals.

Funktionsweise

Der Baustein leitet den am Socket `IN` anliegenden unidirektionalen Adapter **ohne Manipulation** parallel an die drei Plugs `OUT1`, `OUT2` und `OUT3` weiter. Es findet keine Pufferung, Verarbeitung oder zeitliche Steuerung statt. Der Baustein fungiert als reiner **Splitter** (Verteiler) für adapterbasierte Signale.

Da der Adapter `AB` unidirektional ist, erfolgt die Datenübertragung ausschließlich in eine Richtung (vom Socket zu den Plugs). Die drei Ausgänge sind voneinander unabhängig und liefern stets den aktuellen Zustand des Eingangs.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist durch das Attribut `eclipse4diac::core::GenericClassName` als generisch markiert (`'GEN_AB_SPLIT'`). Das bedeutet, der tatsächliche Adapter-Typ kann zur Laufzeit oder beim Instanziieren spezifiziert werden, solange er die Grundstruktur eines unidirektionalen AB-Adapters besitzt.
- **Unidirektionalität**: Sowohl Eingangs- als auch Ausgangs-Adapter sind vom Typ `unidirectional::AB`. Eine Rückwärtskommunikation ist nicht vorgesehen.
- **Keine Zustandshaltung**: Der Baustein besitzt keine interne Zustandsmaschine oder Speicher. Die Weiterleitung erfolgt passiv und kontinuierlich.
- **Copyright**: Der Baustein wurde von der HR Agrartechnik GmbH unter der Eclipse Public License 2.0 veröffentlicht.

Zustandsübersicht

Der Funktionsblock besitzt **keine explizite Zustandsmaschine**. Das Verhalten ist rein datenflussorientiert: Sobald der Eingangs-Adapter Daten liefert, werden diese unmittelbar an alle Ausgänge propagiert. Eine Zustandsüberwachung oder -steuerung ist nicht erforderlich.

Anwendungsszenarien

- **Signalverteilung in der Automatisierungstechnik**: Aufteilen eines Sensorsignals (z. B. Temperatur, Druck) an mehrere Verarbeitungseinheiten.
- **Parallelisierung von Steuerungsabläufen**: Bereitstellen desselben Adapter-Streams für verschiedene Funktionsblöcke, die unabhängig voneinander arbeiten.
- **Prototyping und Test**: Verwenden eines simulierten Adapters, um mehrere Systemkomponenten gleichzeitig zu versorgen.

Vergleich mit ähnlichen Bausteinen

Baustein	Verhalten	Besonderheit
`SPLIT_2`	1-zu-2-Verteilung	Analoger Baustein mit zwei Ausgängen.
`AB_SPLIT_3_UNGATED`	1-zu-3-Verteilung	Dieser Baustein.
`AB_JOIN`	Zusammenführen mehrerer Adapter	Gegenstück zum Splitter.

Während `SPLIT_2` nur zwei Ausgänge bietet, deckt `AB_SPLIT_3_UNGATED` den Bedarf für eine dreifache Verteilung ab. Es sind keine weiteren logischen Funktionen wie Filterung oder Verzögerung implementiert.

- `AB_SPLIT_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

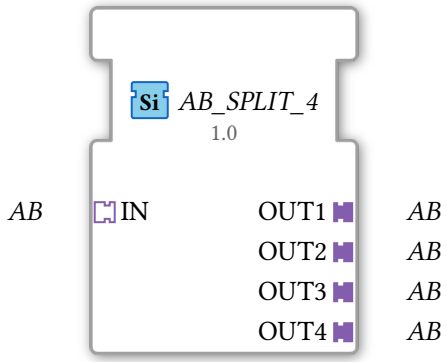
Fazit

Der **AB_SPLIT_3_UNGATED** ist ein einfacher, aber nützlicher generischer Splitter-Baustein für unidirektionale Adapter. Er ermöglicht eine saubere Strukturierung von Automatisierungsanwendungen, indem ein eingehender Signalpfad auf drei parallele Pfade aufgeteilt wird. Dank seiner generischen Natur kann er für beliebige AB-Adapter-Typen eingesetzt werden und fügt sich nahtlos in die IEC 61499-2-konforme Entwicklungsumgebung 4diac ein.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AB_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AB_SPLIT_4** dient dazu, eine einzelne eingehende unidirektionale Adapterverbindung vom Typ `AB` auf vier parallele ausgehende Adapterverbindungen gleichen Typs zu verteilen. Er ist als generischer Baustein konzipiert und ermöglicht die Aufteilung eines Daten- oder Signalflusses auf mehrere nachgeschaltete Komponenten.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	`IN`	`adapter::types::unidirectional::AB`	Eingehende unidirektionale Adapterverbindung
Plug	`OUT1`	`adapter::types::unidirectional::AB`	Erste ausgehende Adapterverbindung
Plug	`OUT2`	`adapter::types::unidirectional::AB`	Zweite ausgehende Adapterverbindung
Plug	`OUT3`	`adapter::types::unidirectional::AB`	Dritte ausgehende Adapterverbindung
Plug	`OUT4`	`adapter::types::unidirectional::AB`	Vierte ausgehende Adapterverbindung

Funktionsweise

Der Baustein fungiert als passiver Verteiler (Splitter) auf Adapterebene. Sobald am Socket `IN` eine Adapterverbindung des Typs `AB` hergestellt wird, leitet der Baustein die über diese Verbindung ankommenden Daten oder Signale unverändert an alle vier Plugs `OUT1` bis `OUT4` weiter. Die Aufteilung erfolgt ohne Datenmodifikation oder Pufferung. Der Baustein besitzt keine eigenen Ereignisse oder Datenpunkte; die gesamte Kommunikation erfolgt ausschließlich über die Adapterschnittstellen.

Technische Besonderheiten

- **Generische Auslegung:** Der Baustein ist als generischer FB (`GenericClassName = 'GEN_AB_SPLIT'`) implementiert und kann daher für verschiedene konkrete Ausprägungen des Adaptertyps `AB` verwendet werden.
- **Unidirektionalität:** Die Adapter sind als unidirektional deklariert. Dies bedeutet, dass der Datenfluss nur in eine Richtung (vom Socket zu den Plugs) erfolgt. Rückkanäle sind nicht vorgesehen.
- **Keine aktive Logik:** Der Baustein enthält keinen Algorithmus oder Zustandsautomaten. Er fungiert rein passiv als Verdrahtungshilfe auf der Architekturebene.
- **Einfachheit:** Der Baustein reduziert die Komplexität der Systemverkabelung, indem er eine physische oder logische Aufteilung eines Adaptersignals modelliert.

Zustandsübersicht

Der Baustein besitzt keinen internen Zustandsautomaten (ECC – Execution Control Chart). Er verhält sich statisch und leitet während der gesamten Laufzeit die an `IN` anliegenden Daten an alle Ausgänge weiter. Es gibt keine Unterbrechungs- oder Initialisierungszustände.

Anwendungsszenarien

- **Verteilen von Sensorwerten:** Ein Sensor liefert über einen `AB`-Adapter Daten, die parallel an mehrere Steuerungsmodule oder Anzeigeeinheiten weitergegeben werden müssen.
- **Test- und Diagnoseumgebungen:** Ein Datenstrom wird an eine Verarbeitungseinheit und gleichzeitig an eine Logging- oder Überwachungskomponente gesendet.
- **Modulare Systeme:** Innerhalb einer größeren Automatisierungsanwendung kann der Baustein verwendet werden, um ein gemeinsames Bussignal auf mehrere nachgeschaltete Funktionsbausteine aufzusplitten.

Vergleich mit ähnlichen Bausteinen

Ähnliche Splitter-Bausteine existieren für andere Adaptertypen oder für datenbasierte Weiterleitungen. Der `AB_SPLIT_4` zeichnet sich durch seine spezifische Passform für den Adaptertyp `AB` und die unidirektionale Auslegung aus. Im Gegensatz zu Bausteinen mit Ereignis- oder Datenschnittstellen benötigt er keine zusätzliche Taktung oder Synchronisation – die Aufteilung erfolgt implizit durch die Verbindungstopologie.

- `AB_SPLIT_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

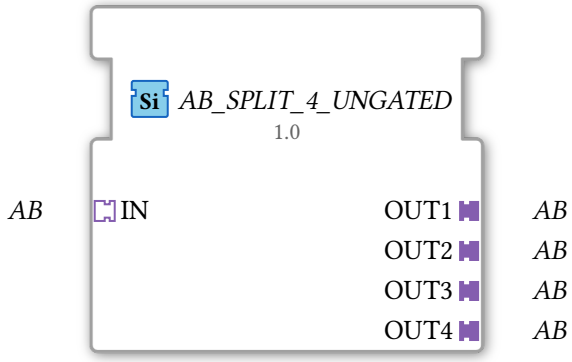
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der `AB_SPLIT_4` ist ein einfacher, aber nützlicher generischer Funktionsbaustein zur Vervielfachung einer unidirektionalen Adapterverbindung. Er erleichtert die Strukturierung von Steuerungsanwendungen, indem er redundante Verkabelung vermeidet und eine klare Aufteilung von Signalfüssen ermöglicht. Seine generische Implementierung macht ihn flexibel und in verschiedenen Kontexten einsetzbar.

AB_SPLIT_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AB_SPLIT_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AB_SPLIT_4_UNGATED** dient dazu, eine einzelne eingehende unidirektionale Adapterverbindung vom Typ `AB` auf vier parallele ausgehende Adapterverbindungen gleichen Typs zu verteilen. Er ist als generischer Baustein konzipiert und ermöglicht die Aufteilung eines Daten- oder Signalflusses auf mehrere nachgeschaltete Komponenten.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	`IN`	`adapter::types::unidirectional::AB`	Eingehende unidirektionale Adapterverbindung
Plug	`OUT1`	`adapter::types::unidirectional::AB`	Erste ausgehende Adapterverbindung
Plug	`OUT2`	`adapter::types::unidirectional::AB`	Zweite ausgehende Adapterverbindung
Plug	`OUT3`	`adapter::types::unidirectional::AB`	Dritte ausgehende Adapterverbindung
Plug	`OUT4`	`adapter::types::unidirectional::AB`	Vierte ausgehende Adapterverbindung

Funktionsweise

Der Baustein fungiert als passiver Verteiler (Splitter) auf Adapterebene. Sobald am Socket `IN` eine Adapterverbindung des Typs `AB` hergestellt wird, leitet der Baustein die über diese Verbindung ankommenden Daten oder Signale unverändert an alle vier Plugs `OUT1` bis `OUT4` weiter. Die Aufteilung erfolgt ohne Datenmodifikation oder Pufferung. Der Baustein besitzt keine eigenen Ereignisse oder Datenpunkte; die gesamte Kommunikation erfolgt ausschließlich über die Adapterschnittstellen.

Technische Besonderheiten

- **Generische Auslegung:** Der Baustein ist als generischer FB (`GenericClassName = 'GEN_AB_SPLIT'`) implementiert und kann daher für verschiedene konkrete Ausprägungen des Adaptertyps `AB` verwendet werden.
- **Unidirektionalität:** Die Adapter sind als unidirektional deklariert. Dies bedeutet, dass der Datenfluss nur in eine Richtung (vom Socket zu den Plugs) erfolgt. Rückkanäle sind nicht vorgesehen.
- **Keine aktive Logik:** Der Baustein enthält keinen Algorithmus oder Zustandsautomaten. Er fungiert rein passiv als Verdrahtungshilfe auf der Architekturebene.
- **Einfachheit:** Der Baustein reduziert die Komplexität der Systemverkabelung, indem er eine physische oder logische Aufteilung eines Adaptersignals modelliert.

Zustandsübersicht

Der Baustein besitzt keinen internen Zustandsautomaten (ECC – Execution Control Chart). Er verhält sich statisch und leitet während der gesamten Laufzeit die an `IN` anliegenden Daten an alle Ausgänge weiter. Es gibt keine Unterbrechungs- oder Initialisierungszustände.

Anwendungsszenarien

- **Verteilen von Sensorwerten:** Ein Sensor liefert über einen `AB`-Adapter Daten, die parallel an mehrere Steuerungsmodule oder Anzeigeeinheiten weitergegeben werden müssen.
- **Test- und Diagnoseumgebungen:** Ein Datenstrom wird an eine Verarbeitungseinheit und gleichzeitig an eine Logging- oder Überwachungskomponente gesendet.
- **Modulare Systeme:** Innerhalb einer größeren Automatisierungsanwendung kann der Baustein verwendet werden, um ein gemeinsames Bussignal auf mehrere nachgeschaltete Funktionsbausteine aufzusplitten.

Vergleich mit ähnlichen Bausteinen

Ähnliche Splitter-Bausteine existieren für andere Adaptertypen oder für datenbasierte Weiterleitungen. Der `AB_SPLIT_4_UNGATED` zeichnet sich durch seine spezifische Passform für den Adaptertyp `AB` und die unidirektionale Auslegung aus. Im Gegensatz zu Bausteinen mit Ereignis- oder Datenschnittstellen benötigt er keine zusätzliche Taktung oder Synchronisation – die Aufteilung erfolgt implizit durch die Verbindungstopologie.

- `AB_SPLIT_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

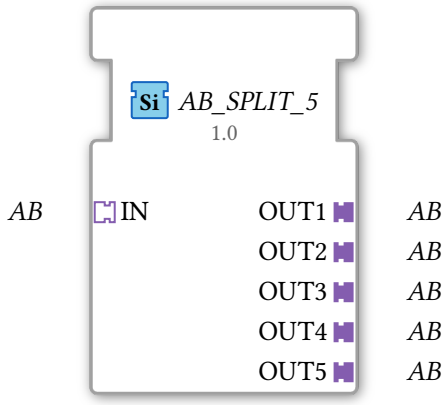
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AB_SPLIT_4_UNGATED` ist ein einfacher, aber nützlicher generischer Funktionsbaustein zur Vervielfachung einer unidirektionalen Adapterverbindung. Er erleichtert die Strukturierung von Steuerungsanwendungen, indem er redundante Verkabelung vermeidet und eine klare Aufteilung von Signalfüssen ermöglicht. Seine generische Implementierung macht ihn flexibel und in verschiedenen Kontexten einsetzbar.

AB_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AB_SPLIT_5` dient dazu, einen unidirektionalen AB-Adapter-Eingang (Socket) auf fünf identische AB-Adapter-Ausgänge (Plugs) zu verteilen. Es handelt sich um einen generischen Baustein, der für die Signalverteilung in Automatisierungssystemen konzipiert ist.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine. Die Datenübertragung erfolgt ausschließlich über die Adapter-Schnittstellen.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	`IN`	`adapter::types::unidirectional::AB`	Unidirektionaler AB-Adapter, der das Eingangssignal bereitstellt.
Plug (Ausgang)	`OUT1`	`adapter::types::unidirectional::AB`	Erster unidirektionaler AB-Adapter-Ausgang.
Plug (Ausgang)	`OUT2`	`adapter::types::unidirectional::AB`	Zweiter unidirektionaler AB-Adapter-Ausgang.
Plug (Ausgang)	`OUT3`	`adapter::types::unidirectional::AB`	Dritter unidirektionaler AB-Adapter-Ausgang.
Plug (Ausgang)	`OUT4`	`adapter::types::unidirectional::AB`	Vierter unidirektionaler AB-Adapter-Ausgang.
Plug (Ausgang)	`OUT5`	`adapter::types::unidirectional::AB`	Fünfter unidirektionaler AB-Adapter-Ausgang.

Funktionsweise

Der Baustein empfängt über den Socket `IN` ein unidirektionales AB-Signal. Dieses Signal wird ohne Änderung an alle fünf Plugs (`OUT1` bis `OUT5`) weitergeleitet. Dadurch entstehen fünf identische Kopien

des Eingangssignals, die parallel an verschiedene nachfolgende Bausteine oder Systeme angeschlossen werden können.

Da der Baustein keine Ereignis- oder Dateneingänge besitzt, erfolgt die gesamte Kommunikation rein über die Adapter-Schnittstellen. Die Verteilung geschieht statisch und ohne Verzögerung oder Pufferung.

Technische Besonderheiten

- **Generischer Baustein**: `AB_SPLIT_5` verwendet den generischen Klassennamen `GEN_AB_SPLIT` und ermöglicht eine typsichere Instanziierung in der 4diac-IDE.
- **Unidirektionalität**: Alle beteiligten Adapter sind vom Typ `unidirectional::AB`, d. h. die Daten fließen nur in eine Richtung (vom Socket zu den Plugs).
- **Keine Zustandslogik**: Der Baustein besitzt kein internes Zustandsdiagramm oder Ereignisverarbeitung – er ist rein strukturell und führt keine Transformation der Daten durch.

Zustandsübersicht

Es existiert kein Zustandsautomat. Der Baustein arbeitet kontinuierlich und gibt das empfangene Signal permanent an alle fünf Ausgänge weiter.

Anwendungsszenarien

- **Signalverteilung in Feldbussystemen**: Wenn ein Sensorwert oder Steuersignal (als AB-Adapter) an mehrere Aktoren oder Steuerungen gesendet werden muss.
- **Multiplexing in modularen Maschinen**: Verteilung eines zentralen AB-Signals an verschiedene Module einer Anlage.
- **Test und Simulation**: Gleichzeitiges Anschließen mehrerer Überwachungs- oder Logging-Einheiten an eine Datenquelle.

Vergleich mit ähnlichen Bausteinen

Baustein	Anzahl Ausgänge	Besonderheit
`AB_SPLIT_2`	2	Einfache 1:2-Verteilung
`AB_SPLIT_3`	3	1:3-Verteilung
`AB_SPLIT_5`	5	1:5-Verteilung (dieser Baustein)
`AB_SPLIT_N` (generisch)	parametrierbar	Individuelle Anzahl über GenericClassName

Gegenüber einem parametrierbaren Split-Baustein bietet `AB_SPLIT_5` eine feste, aber einfache und typsichere Lösung ohne zusätzliche Konfigurationsaufwände.

- **`AB_SPLIT_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

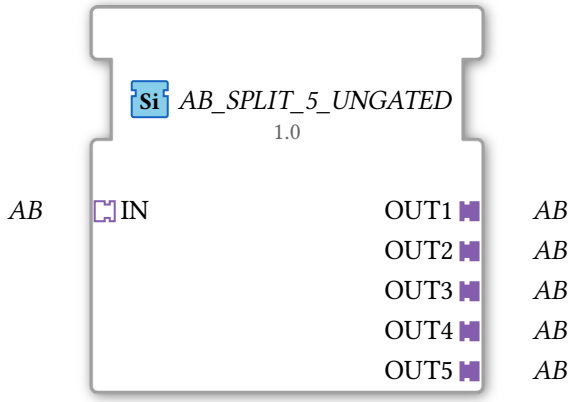
`AB_SPLIT_5` ist ein schlanker, spezialisierter Funktionsbaustein für die 1:5-Verteilung unidirektionaler AB-Adapter-Signale. Dank seiner generischen Implementierung und der klaren Schnittstellenstruktur eignet er sich ideal für den modularen Aufbau von Automatisierungssystemen, in denen ein Signal mehrfach benötigt wird.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AB_SPLIT_5_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `AB_SPLIT_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AB_SPLIT_5_UNGATED` dient dazu, einen unidirektionalen AB-Adapter-Eingang (Socket) auf fünf identische AB-Adapter-Ausgänge (Plugs) zu verteilen. Es handelt sich um einen generischen Baustein, der für die Signalverteilung in Automatisierungssystemen konzipiert ist.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine. Die Datenübertragung erfolgt ausschließlich über die Adapter-Schnittstellen.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	`IN`	`adapter::types::unidirectional::AB`	Unidirektionaler AB-Adapter, der das Eingangssignal bereitstellt.
Plug (Ausgang)	`OUT1`	`adapter::types::unidirectional::AB`	Erster unidirektionaler AB-Adapter-Ausgang.
Plug (Ausgang)	`OUT2`	`adapter::types::unidirectional::AB`	Zweiter unidirektionaler AB-Adapter-Ausgang.
Plug (Ausgang)	`OUT3`	`adapter::types::unidirectional::AB`	Dritter unidirektionaler AB-Adapter-Ausgang.

Plug (Ausgang)	`OUT4`	`adapter::types::unidirectional::AB`	Vierter unidirektionaler AB-Adapter-Ausgang.
Plug (Ausgang)	`OUT5`	`adapter::types::unidirectional::AB`	Fünfter unidirektionaler AB-Adapter-Ausgang.

Funktionsweise

Der Baustein empfängt über den Socket `IN` ein unidirektionales AB-Signal. Dieses Signal wird ohne Änderung an alle fünf Plugs (`OUT1` bis `OUT5`) weitergeleitet. Dadurch entstehen fünf identische Kopien des Eingangssignals, die parallel an verschiedene nachfolgende Bausteine oder Systeme angeschlossen werden können.

Da der Baustein keine Ereignis- oder Dateneingänge besitzt, erfolgt die gesamte Kommunikation rein über die Adapter-Schnittstellen. Die Verteilung geschieht statisch und ohne Verzögerung oder Pufferung.

Technische Besonderheiten

- **Generischer Baustein**: `AB_SPLIT_5_UNGATED` verwendet den generischen Klassennamen `GEN_AB_SPLIT` und ermöglicht eine typsichere Instanziierung in der 4diac-IDE.
- **Unidirektionalität**: Alle beteiligten Adapter sind vom Typ `unidirectional::AB`, d. h. die Daten fließen nur in eine Richtung (vom Socket zu den Plugs).
- **Keine Zustandslogik**: Der Baustein besitzt kein internes Zustandsdiagramm oder Ereignisverarbeitung – er ist rein strukturell und führt keine Transformation der Daten durch.

Zustandsübersicht

Es existiert kein Zustandsautomat. Der Baustein arbeitet kontinuierlich und gibt das empfangene Signal permanent an alle fünf Ausgänge weiter.

Anwendungsszenarien

- **Signalverteilung in Feldbussystemen**: Wenn ein Sensorwert oder Steuersignal (als AB-Adapter) an mehrere Aktoren oder Steuerungen gesendet werden muss.
- **Multiplexing in modularen Maschinen**: Verteilung eines zentralen AB-Signals an verschiedene Module einer Anlage.
- **Test und Simulation**: Gleichzeitiges Anschließen mehrerer Überwachungs- oder Logging-Einheiten an eine Datenquelle.

Vergleich mit ähnlichen Bausteinen

Baustein	Anzahl Ausgänge	Besonderheit
`AB_SPLIT_2`	2	Einfache 1:2-Verteilung
`AB_SPLIT_3`	3	1:3-Verteilung
`AB_SPLIT_5_UNGATED`	5	1:5-Verteilung (dieser Baustein)
`AB_SPLIT_N` (generisch)	parametrierbar	Individuelle Anzahl über GenericClassName

Gegenüber einem parametrierbaren Split-Baustein bietet `AB_SPLIT_5_UNGATED` eine feste, aber einfache und typsichere Lösung ohne zusätzliche Konfigurationsaufwände.

- `AB_SPLIT_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

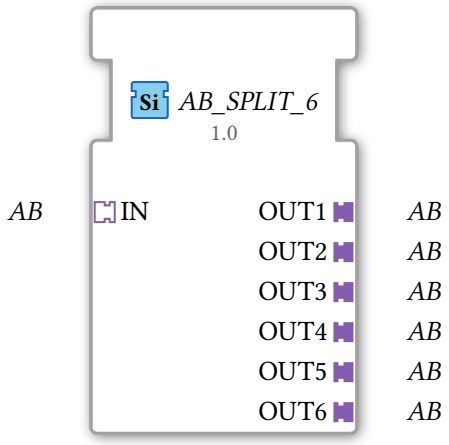
Fazit

`AB_SPLIT_5_UNGATED` ist ein schlanker, spezialisierter Funktionsbaustein für die 1:5-Verteilung unidirektionaler AB-Adapter-Signale. Dank seiner generischen Implementierung und der klaren Schnittstellenstruktur eignet er sich ideal für den modularen Aufbau von Automatisierungssystemen, in denen ein Signal mehrfach benötigt wird.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AB_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AB_SPLIT_6** ist ein generischer Adapter-Splitter. Er dient dazu, einen eingehenden unidirektionalen Adapterdatenstrom (Typ `AB`) gleichzeitig auf sechs separate Ausgänge zu verteilen. Dadurch können die Daten eines einzelnen Adapters mehreren nachgeschalteten Bausteinen ohne Datenverlust zur Verfügung gestellt werden.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine Ereignis-Eingänge. Die Datenweitergabe erfolgt ausschließlich über die Adapterschnittstelle.

Ereignis-Ausgänge

Es sind keine Ereignis-Ausgänge vorhanden. Die Ausgaben erfolgen rein über die Adapter.

Daten-Eingänge

Der Baustein verfügt über keine separaten Daten-Eingänge. Die eigentlichen Nutzdaten werden über den Adaptereingang `IN` transportiert.

Daten-Ausgänge

Es existieren keine expliziten Daten-Ausgänge. Die aufgeteilten Daten werden über die Adapterausgänge `OUT1` bis `OUT6` bereitgestellt.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AB`	`IN`	Socket	Eingangsadapter, der die zu verteilenden Daten liefert.
`adapter::types::unidirectional::AB`	`OUT1`	Plug	Erster Ausgangsadapter – erhält eine Kopie der Eingangsdaten.
`adapter::types::unidirectional::AB`	`OUT2`	Plug	Zweiter Ausgangsadapter.
`adapter::types::unidirectional::AB`	`OUT3`	Plug	Dritter Ausgangsadapter.
`adapter::types::unidirectional::AB`	`OUT4`	Plug	Vierter Ausgangsadapter.

`adapter::types::unidirectional::AB`	`OUT5`	Plug	Fünfter Ausgangsadapter.
`adapter::types::unidirectional::AB`	`OUT6`	Plug	Sechster Ausgangsadapter.

Funktionsweise

Der Baustein liest die vom Adapter `IN` ankommenden Daten ein und gibt diese identisch an alle sechs Ausgangsadapter `OUT1` bis `OUT6` weiter. Es handelt sich um eine reine 1:6-Verteilung ohne Pufferung oder Datenverarbeitung. Sobald am Eingang Daten anliegen, werden diese sofort und gleichzeitig auf alle Ausgänge übertragen. Die Anzahl der Ausgänge ist fest auf sechs festgelegt.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_SPLIT`) realisiert und kann im 4diac-IDE-Framework mehrfach verwendet werden.
- **Unidirektionale Schnittstelle**: Alle Adapter sind vom Typ `unidirectional::AB`, was bedeutet, dass der Datenfluss nur in eine Richtung (vom Eingang zu den Ausgängen) erfolgt.
- **Keine Zustandsabhängigkeit**: Da weder Ereignisse noch Triggermechanismen existieren, arbeitet der Baustein kontinuierlich und ohne inneren Zustand.
- **Lizenz**: Der Baustein steht unter der Eclipse Public License 2.0 (EPL-2.0).

Zustandsübersicht

Der Baustein besitzt keinen internen Automaten oder diskrete Zustände. Die Funktionsweise ist rein durchlaufend – es gibt keine Initialisierung, keine Fehlerzustände und keine zeitlichen Abhängigkeiten.

Anwendungsszenarien

- **Datenverteilung in Steuerungsanwendungen**: Wenn ein Sensor oder eine Datenquelle (z. B. ein IO-Link-Master über einen `AB`-Adapter) ihre Messwerte mehreren parallel arbeitenden Funktionsbausteinen zur Verfügung stellen muss.
- **Redundante Verarbeitung**: Ein Eingangssignal wird gleichzeitig an mehrere unabhängige Berechnungs- oder Überwachungslogiken weitergeleitet.
- **Prototyping**: Einfaches Duplizieren eines Adapter-Signals während der Entwicklungsphase, ohne dass eigene Splitter-Logik geschrieben werden muss.

Vergleich mit ähnlichen Bausteinen

- **AB_SPLIT_2 / AB_SPLIT_4**: Analog aufgebaut, aber mit zwei bzw. vier Ausgängen. Der vorliegende Baustein bietet die maximale Anzahl von sechs Ausgängen.
- **Generische Splits mit Ereignissen**: Manche Splitter arbeiten ereignisgesteuert (z. B. `E_SPLIT`), erfordern dann aber zusätzliche Trigger. Der `AB_SPLIT_6` arbeitet dagegen kontinuierlich und ereignislos über die Adapterschnittstelle.
- **Daten-Splitter (z. B. `F_SPLIT`)**: Diese teilen einzelne Datenwerte (z. B. ein Array) auf, während der `AB_SPLIT_6` den gesamten Adapter-Datenstrom unverändert kopiert.
- **`AB_SPLIT_6_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

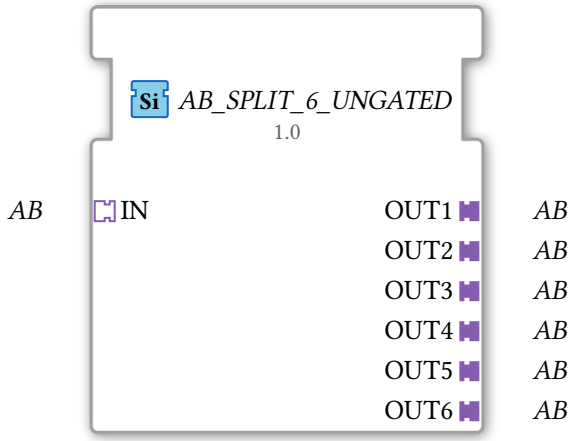
Der `AB_SPLIT_6` ist ein einfacher, aber nützlicher generischer Funktionsbaustein zur Vervielfachung eines unidirektionalen Adaptersignals auf bis zu sechs Ausgänge. Er ist leicht verständlich, benötigt keine Konfiguration und eignet sich hervorragend für die schnelle Verteilung von Adapterdaten in industriellen Steuerungsanwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AB_SPLIT_6_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AB_SPLIT_6`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AB_SPLIT_6_UNGATED** ist ein generischer Adapter-Splitter. Er dient dazu, einen eingehenden unidirektionalen Adapterdatenstrom (Typ `AB`) gleichzeitig auf sechs separate Ausgänge zu verteilen. Dadurch können die Daten eines einzelnen Adapters mehreren nachgeschalteten Bausteinen ohne Datenverlust zur Verfügung gestellt werden.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine Ereignis-Eingänge. Die Datenweitergabe erfolgt ausschließlich über die Adapterschnittstelle.

Ereignis-Ausgänge

Es sind keine Ereignis-Ausgänge vorhanden. Die Ausgaben erfolgen rein über die Adapter.

Daten-Eingänge

Der Baustein verfügt über keine separaten Daten-Eingänge. Die eigentlichen Nutzdaten werden über den Adaptereingang `IN` transportiert.

Daten-Ausgänge

Es existieren keine expliziten Daten-Ausgänge. Die aufgeteilten Daten werden über die Adapterausgänge `OUT1` bis `OUT6` bereitgestellt.

Adapter

Typ	Name	Richtung	Beschreibung

`adapter::types::unidirectional::AB`	`IN`	Socket	Eingangsadapter, der die zu verteilenden Daten liefert.
`adapter::types::unidirectional::AB`	`OUT1`	Plug	Erster Ausgangsadapter – erhält eine Kopie der Eingangsdaten.
`adapter::types::unidirectional::AB`	`OUT2`	Plug	Zweiter Ausgangsadapter.
`adapter::types::unidirectional::AB`	`OUT3`	Plug	Dritter Ausgangsadapter.
`adapter::types::unidirectional::AB`	`OUT4`	Plug	Vierter Ausgangsadapter.
`adapter::types::unidirectional::AB`	`OUT5`	Plug	Fünfter Ausgangsadapter.
`adapter::types::unidirectional::AB`	`OUT6`	Plug	Sechster Ausgangsadapter.

Funktionsweise

Der Baustein liest die vom Adapter `IN` ankommenden Daten ein und gibt diese identisch an alle sechs Ausgangsadapter `OUT1` bis `OUT6` weiter. Es handelt sich um eine reine 1:6-Verteilung ohne Pufferung oder Datenverarbeitung. Sobald am Eingang Daten anliegen, werden diese sofort und gleichzeitig auf alle Ausgänge übertragen. Die Anzahl der Ausgänge ist fest auf sechs festgelegt.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AB_SPLIT`) realisiert und kann im 4diac-IDE-Framework mehrfach verwendet werden.
- **Unidirektionale Schnittstelle**: Alle Adapter sind vom Typ `unidirectional::AB`, was bedeutet, dass der Datenfluss nur in eine Richtung (vom Eingang zu den Ausgängen) erfolgt.
- **Keine Zustandsabhängigkeit**: Da weder Ereignisse noch Triggermechanismen existieren, arbeitet der Baustein kontinuierlich und ohne inneren Zustand.
- **Lizenz**: Der Baustein steht unter der Eclipse Public License 2.0 (EPL-2.0).

Zustandsübersicht

Der Baustein besitzt keinen internen Automaten oder diskrete Zustände. Die Funktionsweise ist rein durchlaufend – es gibt keine Initialisierung, keine Fehlerzustände und keine zeitlichen Abhängigkeiten.

Anwendungsszenarien

- **Datenverteilung in Steuerungsanwendungen**: Wenn ein Sensor oder eine Datenquelle (z. B. ein IO-Link-Master über einen `AB`-Adapter) ihre Messwerte mehreren parallel arbeitenden Funktionsbausteinen zur Verfügung stellen muss.
- **Redundante Verarbeitung**: Ein Eingangssignal wird gleichzeitig an mehrere unabhängige Berechnungs- oder Überwachungslogiken weitergeleitet.
- **Prototyping**: Einfaches Duplizieren eines Adapter-Signals während der Entwicklungsphase, ohne dass eigene Splitter-Logik geschrieben werden muss.

Vergleich mit ähnlichen Bausteinen

- **AB_SPLIT_2 / AB_SPLIT_4**: Analog aufgebaut, aber mit zwei bzw. vier Ausgängen. Der vorliegende Baustein bietet die maximale Anzahl von sechs Ausgängen.
- **Generische Splits mit Ereignissen**: Manche Splitter arbeiten ereignisgesteuert (z. B. `E_SPLIT`), erfordern dann aber zusätzliche Trigger. Der `AB_SPLIT_6_UNGATED` arbeitet dagegen kontinuierlich und ereignislos über die Adapterschnittstelle.
- **Daten-Splitter (z. B. `F_SPLIT`)**: Diese teilen einzelne Datenwerte (z. B. ein Array) auf, während der `AB_SPLIT_6_UNGATED` den gesamten Adapter-Datenstrom unverändert kopiert.
- `AB_SPLIT_6`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

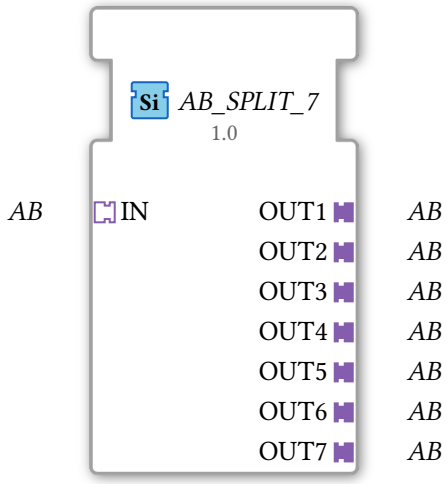
Fazit

Der `AB_SPLIT_6_UNGATED` ist ein einfacher, aber nützlicher generischer Funktionsbaustein zur Vervielfachung eines unidirektionalen Adaptersignals auf bis zu sechs Ausgänge. Er ist leicht verständlich, benötigt keine Konfiguration und eignet sich hervorragend für die schnelle Verteilung von Adapterdaten in industriellen Steuerungsanwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AB_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AB_SPLIT_7** dient dazu, einen einzelnen eingehenden Adapter vom Typ `adapter::types::unidirectional::AB` auf sieben identische Ausgangsadapter aufzuteilen. Er ist als generischer Baustein (`GEN_AB_SPLIT`) realisiert und kann daher für verschiedene konkrete Adaptertypen instanziiert werden, die dem unidirektionalen AB-Protokoll entsprechen. Entwickelt wurde er unter der Eclipse Public License 2.0.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

Name	Typ	Richtung	Beschreibung
IN	`adapter::types::unidirectional::AB`	Socket (Eingang)	Eingehender Adapter, der auf die Ausgänge verteilt wird.
OUT1	`adapter::types::unidirectional::AB`	Plug (Ausgang)	Erster ausgehender Adapter, identisch mit dem Eingang.

OUT2	adapter::types::unidirectional::AB	Plug (Ausgang)	Zweiter ausgehender Adapter.
OUT3	adapter::types::unidirectional::AB	Plug (Ausgang)	Dritter ausgehender Adapter.
OUT4	adapter::types::unidirectional::AB	Plug (Ausgang)	Vierter ausgehender Adapter.
OUT5	adapter::types::unidirectional::AB	Plug (Ausgang)	Fünfter ausgehender Adapter.
OUT6	adapter::types::unidirectional::AB	Plug (Ausgang)	Sechster ausgehender Adapter.
OUT7	adapter::types::unidirectional::AB	Plug (Ausgang)	Siebter ausgehender Adapter.

Funktionsweise

Der Baustein arbeitet als reiner Signalverteiler für Adapterverbindungen. Jeder Daten- oder Ereignisfluss, der über den Socket `IN` eintrifft, wird unverändert und parallel an alle sieben Ausgangs-Plugs (`OUT1` – `OUT7`) weitergeleitet. Es findet keine Verarbeitung, Pufferung oder Zustandsänderung statt. Der Baustein ist daher als passiver „Splitter“ zu verstehen.

Da es sich um unidirektionale Adapter handelt, erfolgt die Kommunikation nur in einer Richtung – vom Socket zu den Plugs. Die Ausgänge sind logisch identisch und können unabhängig voneinander mit nachfolgenden Bausteinen verbunden werden.

Technische Besonderheiten

- **Generischer Typ**: Der Baustein ist als generischer FB (GenericClassName `'GEN_AB_SPLIT'`) deklariert. Das ermöglicht die Wiederverwendung für verschiedene Ausprägungen des unidirektionalen AB-Adapters, ohne die Schnittstelle neu definieren zu müssen.
- **Keine interne Logik**: Es existieren keine Ereignisauslöser, Datenoperationen oder Zustandsautomaten. Der FB ist vollständig durch seine Adapterverdrahtung definiert.
- **Unidirektionalität**: Der Adaptertyp `unidirectional::AB` unterstützt nur eine Datenflussrichtung. Rückkanäle sind nicht vorhanden.

Zustandsübersicht

Der Funktionsbaustein besitzt keinen internen Zustandsautomaten (ECC). Er ist statisch und führt keinerlei sequenzielle Logik aus. Es gibt keine unterscheidbaren Betriebszustände.

Anwendungsszenarien

- **Verteilen einer Steuerungsinformation** an mehrere parallele Verbraucher (z. B. Ansteuerung mehrerer gleichartiger Aktoren).
- **Aufteilen eines Rückmeldesignals** für getrennte Auswertungen in verschiedenen Zweigen einer Automatisierungsanwendung.
- **Erzeugen mehrerer identischer Adapterstränge** für Redundanz oder separate Überwachung.

Durch die generische Auslegung kann der Baustein in verschiedenen Kontexten eingesetzt werden, sobald der konkrete Adaptertyp spezifiziert ist.

Vergleich mit ähnlichen Bausteinen

- **AB_MERGE**: Führt mehrere eingehende Adapter auf einen Ausgang zusammen – gegenteilige Funktion zu AB_SPLIT_7.
- **AB_SPLIT_N**: Verallgemeinerung mit parametrierbarer Anzahl an Ausgängen, falls verfügbar. AB_SPLIT_7 ist eine feste 1:7-Aufteilung.
- **Direkte Verdrahtung ohne Splitter**: Bei Bedarf an mehreren identischen Kopien müsste der Quell-FB mehrfach instanziiert werden, was den Entwurf aufbläht. Der Splitter reduziert Redundanz auf Ebene der Adapterverbindungen.
- `AB_SPLIT_7_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs

unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

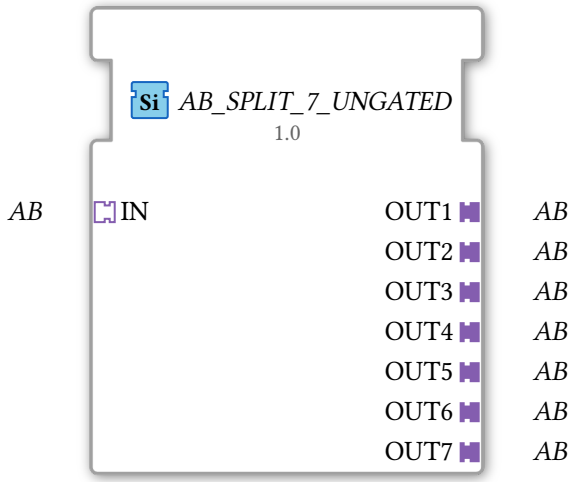
Der **AB_SPLIT_7** ist ein einfacher, aber nützlicher Baustein zur Vervielfachung einer unidirektionalen Adapterverbindung. Seine generische Natur und die klare Trennung von Ein- und Ausgängen machen ihn zu einem wiederverwendbaren Werkzeug in der 4diac-IDE, insbesondere in Szenarien, die eine parallele Verteilung von Signalen erfordern. Er trägt zur Modularität und Übersichtlichkeit industrieller Steuerungsanwendungen bei.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AB_SPLIT_7_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AB_SPLIT_7`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AB_SPLIT_7_UNGATED** dient dazu, einen einzelnen eingehenden Adapter vom Typ `adapter::types::unidirectional::AB` auf sieben identische Ausgangsadapter aufzuteilen. Er ist als generischer Baustein (`GEN_AB_SPLIT`) realisiert und kann daher für verschiedene konkrete Adaptertypen instanziiert werden, die dem unidirektionalen AB-Protokoll entsprechen. Entwickelt wurde er unter der Eclipse Public License 2.0.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

Name	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::AB	Socket (Eingang)	Eingehender Adapter, der auf die Ausgänge verteilt wird.
OUT1	adapter::types::unidirectional::AB	Plug (Ausgang)	Erster ausgehender Adapter, identisch mit dem Eingang.
OUT2	adapter::types::unidirectional::AB	Plug (Ausgang)	Zweiter ausgehender Adapter.
OUT3	adapter::types::unidirectional::AB	Plug (Ausgang)	Dritter ausgehender Adapter.
OUT4	adapter::types::unidirectional::AB	Plug (Ausgang)	Vierter ausgehender Adapter.
OUT5	adapter::types::unidirectional::AB	Plug (Ausgang)	Fünfter ausgehender Adapter.
OUT6	adapter::types::unidirectional::AB	Plug (Ausgang)	Sechster ausgehender Adapter.
OUT7	adapter::types::unidirectional::AB	Plug (Ausgang)	Siebter ausgehender Adapter.

Funktionsweise

Der Baustein arbeitet als reiner Signalverteiler für Adapterverbindungen. Jeder Daten- oder Ereignisfluss, der über den Socket `IN` eintrifft, wird unverändert und parallel an alle sieben Ausgangs-Plugs (`OUT1` – `OUT7`) weitergeleitet. Es findet keine Verarbeitung, Pufferung oder Zustandsänderung statt. Der Baustein ist daher als passiver „Splitter“ zu verstehen.

Da es sich um unidirektionale Adapter handelt, erfolgt die Kommunikation nur in einer Richtung – vom Socket zu den Plugs. Die Ausgänge sind logisch identisch und können unabhängig voneinander mit nachfolgenden Bausteinen verbunden werden.

Technische Besonderheiten

- **Generischer Typ**: Der Baustein ist als generischer FB (GenericClassName `'GEN_AB_SPLIT'`) deklariert. Das ermöglicht die Wiederverwendung für verschiedene Ausprägungen des unidirektionalen AB-Adapters, ohne die Schnittstelle neu definieren zu müssen.
- **Keine interne Logik**: Es existieren keine Ereignisauslöser, Datenoperationen oder Zustandsautomaten. Der FB ist vollständig durch seine Adapterverdrahtung definiert.
- **Unidirektionalität**: Der Adaptertyp `unidirectional::AB` unterstützt nur eine Datenflussrichtung. Rückkanäle sind nicht vorhanden.

Zustandsübersicht

Der Funktionsbaustein besitzt keinen internen Zustandsautomaten (ECC). Er ist statisch und führt keinerlei sequenzielle Logik aus. Es gibt keine unterscheidbaren Betriebszustände.

Anwendungsszenarien

- **Verteilen einer Steuerungsinformation** an mehrere parallele Verbraucher (z. B. Ansteuerung mehrerer gleichartiger Aktoren).
- **Aufteilen eines Rückmeldesignals** für getrennte Auswertungen in verschiedenen Zweigen einer Automatisierungsanwendung.
- **Erzeugen mehrerer identischer Adapterstränge** für Redundanz oder separate Überwachung.

Durch die generische Auslegung kann der Baustein in verschiedenen Kontexten eingesetzt werden, sobald der konkrete Adaptertyp spezifiziert ist.

Vergleich mit ähnlichen Bausteinen

- **AB_MERGE**: Führt mehrere eingehende Adapter auf einen Ausgang zusammen – gegenteilige Funktion zu AB_SPLIT_7_UNGATED.
- **AB_SPLIT_N**: Verallgemeinerung mit parametrierbarer Anzahl an Ausgängen, falls verfügbar. AB_SPLIT_7_UNGATED ist eine feste 1:7-Aufteilung.

- **Direkte Verdrahtung ohne Splitter**: Bei Bedarf an mehreren identischen Kopien müsste der Quell-FB mehrfach instanziiert werden, was den Entwurf aufbläht. Der Splitter reduziert Redundanz auf Ebene der Adapterverbindungen.
- `AB_SPLIT_7`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

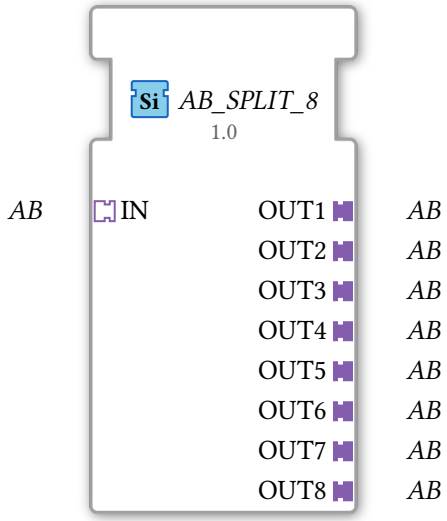
Fazit

Der **AB_SPLIT_7_UNGATED** ist ein einfacher, aber nützlicher Baustein zur Vervielfachung einer unidirektionalen Adapterverbindung. Seine generische Natur und die klare Trennung von Ein- und Ausgängen machen ihn zu einem wiederverwendbaren Werkzeug in der 4diac-IDE, insbesondere in Szenarien, die eine parallele Verteilung von Signalen erfordern. Er trägt zur Modularität und Übersichtlichkeit industrieller Steuerungsanwendungen bei.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AB_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AB_SPLIT_8** dient der Vervielfachung eines eingehenden Adaptersignals vom Typ `AB` auf acht gleichartige Ausgänge. Er fungiert als passiver Verteiler (Splitter) für den unidirektionalen Datenaustausch über die standardisierte Schnittstelle `adapter::types::unidirectional::AB`. Der Baustein ist als generischer Funktionblock ausgelegt und wird typischerweise in der Automatisierungstechnik eingesetzt, um ein AB-Signal parallel an mehrere nachfolgende Komponenten weiterzuleiten.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden. Der FB besitzt keine ereignisgesteuerten Eingänge; die Datenweitergabe erfolgt rein über die Adapterschnittstelle.

Ereignis-Ausgänge

Keine vorhanden. Es werden keine Ereignisse explizit ausgegeben.

Daten-Eingänge

Keine direkten Daten-Eingänge. Alle eingehenden Informationen werden über den Socket-Adapter transportiert.

Daten-Ausgänge

Keine direkten Daten-Ausgänge. Ausgehende Informationen werden über die Plug-Adapter bereitgestellt.

Adapter

Name	Typ	Richtung	Beschreibung
`IN`	`adapter::types::unidirectional::AB`	Socket	Eingangsadapter, der das zu verteilende Signal empfängt.
`OUT1`	`adapter::types::unidirectional::AB`	Plug	Erster Ausgangsadapter (identisch zum Eingangssignal).
`OUT2`	`adapter::types::unidirectional::AB`	Plug	Zweiter Ausgangsadapter.
`OUT3`	`adapter::types::unidirectional::AB`	Plug	Dritter Ausgangsadapter.
`OUT4`	`adapter::types::unidirectional::AB`	Plug	Vierter Ausgangsadapter.
`OUT5`	`adapter::types::unidirectional::AB`	Plug	Fünfter Ausgangsadapter.
`OUT6`	`adapter::types::unidirectional::AB`	Plug	Sechster Ausgangsadapter.
`OUT7`	`adapter::types::unidirectional::AB`	Plug	Siebter Ausgangsadapter.
`OUT8`	`adapter::types::unidirectional::AB`	Plug	Achter Ausgangsadapter.

Funktionsweise

Der **AB_SPLIT_8** arbeitet als reiner Signalverteiler ohne interne Logik oder Zustandsautomat. Das über den Socket `IN` ankommende AB-Signal wird identisch und zeitgleich an alle acht Plug-Ausgänge `OUT1` bis `OUT8` weitergeleitet. Eine Pufferung oder Verzögerung findet nicht statt. Der Baustein ist vollständig passiv – er reagiert ausschließlich auf Datenänderungen des angeschlossenen Adapters und gibt diese unverändert weiter.

Die Verteilung erfolgt unabhängig von der internen Struktur des AB-Adapters (z. B. enthaltene Daten- und Ereignisverbindungen). Der FB selbst besitzt keinen eigenen Ausführungszustand und benötigt keine sequenzielle Steuerung.

Technische Besonderheiten

- **Generischer Typ**: Der FB ist als generischer Baustein deklariert (`GenericClassName = 'GEN_AB_SPLIT'`), was eine flexible Verwendung in verschiedenen Projekten ermöglicht.
- **Keine Zustandsmaschine**: Es existiert kein ECC (Execution Control Chart); der FB ist vollständig datenflussgesteuert.
- **Adaptertyp**: Der verwendete Adapter `adapter::types::unidirectional::AB` ist standardisiert und ermöglicht eine einheitliche Schnittstelle.
- **Lizenz**: Der Baustein steht unter der **Eclipse Public License 2.0 (EPL-2.0)** und ist für die **4diac IDE** konzipiert.

Zustandsübersicht

Der **AB_SPLIT_8** besitzt keine internen Zustände oder Zustandsautomaten. Die Signalweitergabe erfolgt ohne Verzögerung und ohne Abhängigkeit von einer sequenziellen Logik.

Anwendungsszenarien

- **Verteilen eines einzigen AB-Sensorsignals** an mehrere Aktoren oder Steuerungen.
- **Parallelschaltung von Überwachungseinheiten** in der Landtechnik (z. B. Verteilen eines Acknowledge-Signals).
- **Erweiterung von Schaltschrankverkabelungen** durch softwarebasierte Signalvervielfachung.
- **Testumgebungen**, in denen ein AB-Signal gleichzeitig an mehrere Analyse-FBs gesendet werden soll.

Vergleich mit ähnlichen Bausteinen

- **`AB_SPLIT_2`, `AB_SPLIT_4`**: Diese Bausteine verteilen das Signal auf zwei bzw. vier Ausgänge. Der `AB_SPLIT_8` bietet die maximale Anzahl von acht Ausgängen in einem FB.
- **`DATA_SPLIT`**: Verteilt beliebige Datentypen (z. B. `INT`, `BOOL`). Der `AB_SPLIT_8` ist auf den Adaptertyp `AB` spezialisiert und behält die vollständige Adapterstruktur bei.
- **`EC_SPLIT`**: Ein Ereignis-Splitter, der nur Ereignisse vervielfacht, aber keine Daten. Im Gegensatz dazu arbeitet `AB_SPLIT_8` auf Adapterebene inklusive aller darin enthaltenen Daten und Ereignisse.
- **`AB_SPLIT_8_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

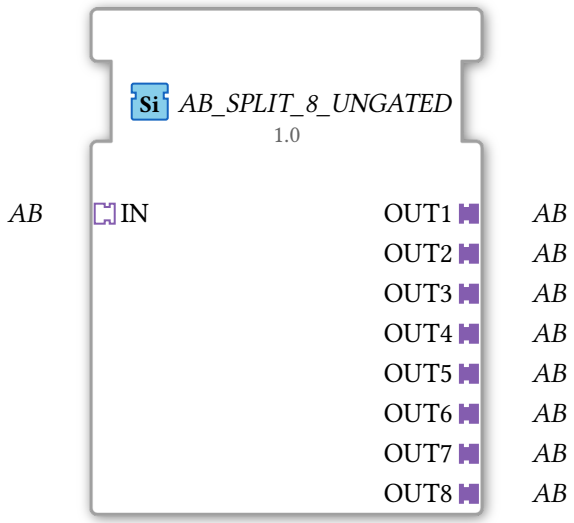
Der **AB_SPLIT_8** ist ein einfacher, aber effektiver Funktionsblock zur Vervielfachung eines unidirektionalen AB-Adaptersignals. Durch seine generische Ausführung und die passive Arbeitsweise eignet er sich ideal für die Parallelisierung von Steuerungs- und Überwachungssignalen in der Automatisierungstechnik. Die fehlende Zustandslogik sorgt für geringe Laufzeitkomplexität und hohe Zuverlässigkeit. Der Baustein ist eine sinnvolle Ergänzung für jede 4diac-Bibliothek, die auf Adapterkommunikation setzt.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 Gesamtwiderstand in Reihen- & Parallelschaltung auf ms-muc-docs.de

AB_SPLIT_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AB_SPLIT_8`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AB_SPLIT_8_UNGATED** dient der Vervielfachung eines eingehenden Adaptersignals vom Typ `AB` auf acht gleichartige Ausgänge. Er fungiert als passiver Verteiler (Splitter) für den unidirektionalen Datenaustausch über die standardisierte Schnittstelle `adapter::types::unidirectional::AB`. Der Baustein ist als generischer Funktionblock ausgelegt und wird typischerweise in der Automatisierungstechnik eingesetzt, um ein AB-Signal parallel an mehrere nachfolgende Komponenten weiterzuleiten.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden. Der FB besitzt keine ereignisgesteuerten Eingänge; die Datenweitergabe erfolgt rein über die Adapterschnittstelle.

Ereignis-Ausgänge

Keine vorhanden. Es werden keine Ereignisse explizit ausgegeben.

Daten-Eingänge

Keine direkten Daten-Eingänge. Alle eingehenden Informationen werden über den Socket-Adapter transportiert.

Daten-Ausgänge

Keine direkten Daten-Ausgänge. Ausgehende Informationen werden über die Plug-Adapter bereitgestellt.

Adapter

Name	Typ	Richtung	Beschreibung
`IN`	`adapter::types::unidirectional::AB`	Socket	Eingangsadapter, der das zu verteilende Signal empfängt.
`OUT1`	`adapter::types::unidirectional::AB`	Plug	Erster Ausgangsadapter (identisch zum Eingangssignal).
`OUT2`	`adapter::types::unidirectional::AB`	Plug	Zweiter Ausgangsadapter.
`OUT3`	`adapter::types::unidirectional::AB`	Plug	Dritter Ausgangsadapter.
`OUT4`	`adapter::types::unidirectional::AB`	Plug	Vierter Ausgangsadapter.
`OUT5`	`adapter::types::unidirectional::AB`	Plug	Fünfter Ausgangsadapter.
`OUT6`	`adapter::types::unidirectional::AB`	Plug	Sechster Ausgangsadapter.
`OUT7`	`adapter::types::unidirectional::AB`	Plug	Siebter Ausgangsadapter.
`OUT8`	`adapter::types::unidirectional::AB`	Plug	Achter Ausgangsadapter.

Funktionsweise

Der **AB_SPLIT_8_UNGATED** arbeitet als reiner Signalverteiler ohne interne Logik oder Zustandsautomat. Das über den Socket `IN` ankommende AB-Signal wird identisch und zeitgleich an alle acht Plug-Ausgänge `OUT1` bis `OUT8` weitergeleitet. Eine Pufferung oder Verzögerung findet nicht statt. Der Baustein ist vollständig passiv – er reagiert ausschließlich auf Datenänderungen des angeschlossenen Adapters und gibt diese unverändert weiter.

Die Verteilung erfolgt unabhängig von der internen Struktur des AB-Adapters (z. B. enthaltene Daten- und Ereignisverbindungen). Der FB selbst besitzt keinen eigenen Ausführungszustand und benötigt keine sequenzielle Steuerung.

Technische Besonderheiten

- **Generischer Typ**: Der FB ist als generischer Baustein deklariert (`GenericClassName = 'GEN_AB_SPLIT'`), was eine flexible Verwendung in verschiedenen Projekten ermöglicht.
- **Keine Zustandsmaschine**: Es existiert kein ECC (Execution Control Chart); der FB ist vollständig datenflussgesteuert.

- **Adaptertyp**: Der verwendete Adapter `adapter::types::unidirectional::AB` ist standardisiert und ermöglicht eine einheitliche Schnittstelle.
- **Lizenz**: Der Baustein steht unter der **Eclipse Public License 2.0 (EPL-2.0)** und ist für die **4diac IDE** konzipiert.

Zustandsübersicht

Der **AB_SPLIT_8_UNGATED** besitzt keine internen Zustände oder Zustandsautomaten. Die Signalweitergabe erfolgt ohne Verzögerung und ohne Abhängigkeit von einer sequenziellen Logik.

Anwendungsszenarien

- **Verteilen eines einzigen AB-Sensorsignals** an mehrere Aktoren oder Steuerungen.
- **Parallelschaltung von Überwachungseinheiten** in der Landtechnik (z. B. Verteilen eines Acknowledge-Signals).
- **Erweiterung von Schaltschrankverkabelungen** durch softwarebasierte Signalvervielfachung.
- **Testumgebungen**, in denen ein AB-Signal gleichzeitig an mehrere Analyse-FBs gesendet werden soll.

Vergleich mit ähnlichen Bausteinen

- **`AB_SPLIT_2`, `AB_SPLIT_4`**: Diese Bausteine verteilen das Signal auf zwei bzw. vier Ausgänge. Der `AB_SPLIT_8_UNGATED` bietet die maximale Anzahl von acht Ausgängen in einem FB.
- **`DATA_SPLIT`**: Verteilt beliebige Datentypen (z. B. `INT`, `BOOL`). Der `AB_SPLIT_8_UNGATED` ist auf den Adaptertyp `AB` spezialisiert und behält die vollständige Adapterstruktur bei.
- **`EC_SPLIT`**: Ein Ereignis-Splitter, der nur Ereignisse vervielfacht, aber keine Daten. Im Gegensatz dazu arbeitet `AB_SPLIT_8_UNGATED` auf Adapterebene inklusive aller darin enthaltenen Daten und Ereignisse.
- **`AB_SPLIT_8`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

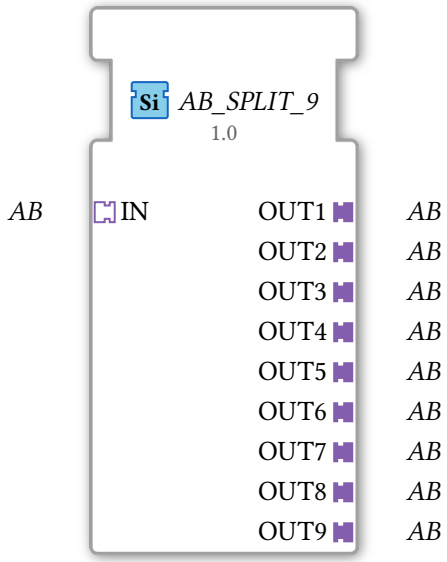
Fazit

Der **AB_SPLIT_8_UNGATED** ist ein einfacher, aber effektiver Funktionsblock zur Vervielfachung eines unidirektionalen AB-Adaptersignals. Durch seine generische Ausführung und die passive Arbeitsweise eignet er sich ideal für die Parallelisierung von Steuerungs- und Überwachungssignalen in der Automatisierungstechnik. Die fehlende Zustandslogik sorgt für geringe Laufzeitkomplexität und hohe Zuverlässigkeit. Der Baustein ist eine sinnvolle Ergänzung für jede 4diac-Bibliothek, die auf Adapterkommunikation setzt.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 Gesamtwiderstand in Reihen- & Parallelschaltung auf ms-muc-docs.de

AB_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AB_SPLIT_9** dient dazu, einen einzelnen eingehenden Adapter vom Typ **AB** auf neun identische ausgehende Adapter (**OUT1** bis **OUT9**) aufzuteilen. Er ist als generischer Baustein (GenericClassName = `'GEN_AB_SPLIT'`) implementiert und kann in beliebigen 4diac-Projekten eingesetzt werden, um eine Punkt-zu-Mehrpunkt-Verbindung über das Adapter-Interface zu realisieren.

Der Baustein besitzt keine ereignisgesteuerten oder datengetriebenen Ein-/Ausgänge – die gesamte Kommunikation erfolgt ausschließlich über die Adapter-Schnittstellen. Dies ermöglicht eine saubere, aufgabenorientierte Kopplung von Modulen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

Schnittstelle	Richtung	Typ	Beschreibung
IN	Socket	`adapter::types::unidirectional::AB`	Eingehender Adapter (Quelle)
OUT1 – OUT9	Plug	`adapter::types::unidirectional::AB`	Neun identische ausgehende Adapter (Senken)

Funktionsweise

Der Baustein kopiert den gesamten Daten- und Ereignis-Durchgriff des eingehenden Adapters **IN** auf die neun Ausgangs-Adapter **OUT1** bis **OUT9**. Jeder Ausgang erhält exakt dieselbe Information wie der Eingang – es findet keine Filterung, Umformung oder Verzögerung statt.

Der Baustein verhält sich somit wie ein **passiver Verteiler** (Splitting-Knoten) auf der Adapter-Ebene. Änderungen am Eingangs-Adapter (z. B. neue Werte oder Events) werden sofort an alle Ausgänge weitergereicht, sofern die angeschlossenen Module dies zulassen.

Technische Besonderheiten

- **Generischer Typ**: Der Baustein ist als generischer FB deklariert (`GenericClassName = 'GEN_AB_SPLIT'`). Dadurch kann er in verschiedenen Kontexten wiederverwendet werden, solange der Adapter-Typ `unidirectional::AB` kompatibel ist.
- **Keine Zustandslogik**: Es gibt kein ECC (Execution Control Chart) – der Baustein ist vollständig deklarativ und benötigt keine sequentielle Verarbeitung.
- **Einfache Erweiterbarkeit**: Das Prinzip lässt sich auf andere Anzahlen von Ausgängen übertragen (z. B. `AB_SPLIT_4`, `AB_SPLIT_16`), indem die XML-Struktur entsprechend angepasst wird.

Zustandsübersicht

Der Baustein besitzt **keine** eigenen Zustände. Das Verhalten ist rein kombinatorisch – die Verteilung erfolgt kontinuierlich und ohne Verzögerung. Eine Zustandsmaschine ist daher nicht erforderlich.

Anwendungsszenarien

- **Steuerung von mehreren parallelen Aktoren**: Ein Sensor (z. B. ein Drucksensor) liefert über einen AB-Adapter seinen Messwert. Dieser Wert soll gleichzeitig an mehrere Stellglieder (Ventile, Motoren) weitergegeben werden – der Split-Baustein verteilt die Information an alle.
- **Signalkopplung in modularen Anlagen**: In einer verteilten Automatisierungslösung können mehrere Subsysteme denselben Datenstrom empfangen, ohne dass jedes Subsystem eine separate Verbindung zur Quelle aufbauen muss.
- **Test- und Simulationsumgebungen**: Ein eingehender Datenstrom kann auf mehrere Überwachungs- oder Logging-Module aufgeteilt werden, ohne die ursprüngliche Verbindung zu beeinträchtigen.

Vergleich mit ähnlichen Bausteinen

Baustein	Ausgänge	Besonderheit
`AB_SPLIT_9`	9	Standard-Split für 1⟶9, generisch
`AB_SPLIT_4`	4	Gleiche Funktionsweise, geringere Anzahl
`AB_MERGE`	–	Fasst mehrere Eingänge zu einem Ausgang zusammen (entgegengesetzte Richtung)

Während `AB_SPLIT_9` eine Quelle auf viele Senken verteilt, realisiert `AB_MERGE` eine Zusammenführung mehrerer Quellen in einen Ausgang. `AB_SPLIT_9` ist daher die ideale Wahl, wenn ein Signal an viele Verbraucher weitergeleitet werden muss.

- **`AB_SPLIT_9_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

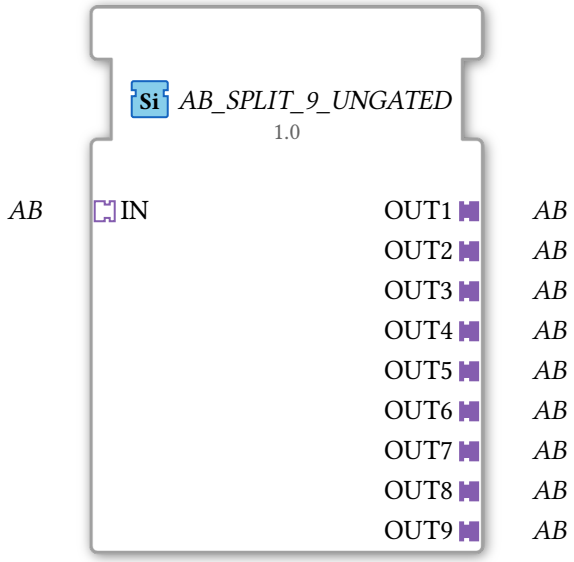
Der **AB_SPLIT_9** bietet eine einfache, aber leistungsfähige Möglichkeit, einen einzelnen Adapter-basierten Datenstrom auf neun parallele Pfade aufzuteilen. Durch seine generische Natur ist er in beliebigen 4diac-Projekten einsetzbar, erfordert keine Zustandsprogrammierung und ermöglicht eine saubere, modulare Architektur. Er ist ein grundlegender Baustein für die serielle oder parallele Verteilung von Adapter-Signalen in der Automatisierungstechnik.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AB_SPLIT_9_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `AB_SPLIT_9`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AB_SPLIT_9_UNGATED** dient dazu, einen einzelnen eingehenden Adapter vom Typ **AB** auf neun identische ausgehende Adapter (**OUT1** bis **OUT9**) aufzuteilen. Er ist als generischer Baustein (GenericClassName = `'GEN_AB_SPLIT'`) implementiert und kann in beliebigen 4diac-Projekten eingesetzt werden, um eine Punkt-zu-Mehrpunkt-Verbindung über das Adapter-Interface zu realisieren.

Der Baustein besitzt keine ereignisgesteuerten oder datengetriebenen Ein-/Ausgänge – die gesamte Kommunikation erfolgt ausschließlich über die Adapter-Schnittstellen. Dies ermöglicht eine saubere, aufgabenorientierte Kopplung von Modulen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

Schnittstelle	Richtung	Typ	Beschreibung
IN	Socket	`adapter::types::unidirectional::AB`	Eingehender Adapter (Quelle)

OUT1 – OUT9	Plug	adapter::types::unidirectional::AB	Neun identische ausgehende Adapter (Senken)

Funktionsweise

Der Baustein kopiert den gesamten Daten- und Ereignis-Durchgriff des eingehenden Adapters **IN** auf die neun Ausgangs-Adapter **OUT1** bis **OUT9**. Jeder Ausgang erhält exakt dieselbe Information wie der Eingang – es findet keine Filterung, Umformung oder Verzögerung statt.

Der Baustein verhält sich somit wie ein **passiver Verteiler** (Splitting-Knoten) auf der Adapter-Ebene. Änderungen am Eingangs-Adapter (z. B. neue Werte oder Events) werden sofort an alle Ausgänge weitergereicht, sofern die angeschlossenen Module dies zulassen.

Technische Besonderheiten

- **Generischer Typ**: Der Baustein ist als generischer FB deklariert (`GenericClassName = 'GEN_AB_SPLIT'`). Dadurch kann er in verschiedenen Kontexten wiederverwendet werden, solange der Adapter-Typ `unidirectional::AB` kompatibel ist.
- **Keine Zustandslogik**: Es gibt kein ECC (Execution Control Chart) – der Baustein ist vollständig deklarativ und benötigt keine sequentielle Verarbeitung.
- **Einfache Erweiterbarkeit**: Das Prinzip lässt sich auf andere Anzahlen von Ausgängen übertragen (z. B. `AB_SPLIT_4`, `AB_SPLIT_16`), indem die XML-Struktur entsprechend angepasst wird.

Zustandsübersicht

Der Baustein besitzt **keine** eigenen Zustände. Das Verhalten ist rein kombinatorisch – die Verteilung erfolgt kontinuierlich und ohne Verzögerung. Eine Zustandsmaschine ist daher nicht erforderlich.

Anwendungsszenarien

- **Steuerung von mehreren parallelen Aktoren**: Ein Sensor (z. B. ein Drucksensor) liefert über einen AB-Adapter seinen Messwert. Dieser Wert soll gleichzeitig an mehrere Stellglieder (Ventile, Motoren) weitergegeben werden – der Split-Baustein verteilt die Information an alle.
- **Signalkopplung in modularen Anlagen**: In einer verteilten Automatisierungslösung können mehrere Subsysteme denselben Datenstrom empfangen, ohne dass jedes Subsystem eine separate Verbindung zur Quelle aufbauen muss.
- **Test- und Simulationsumgebungen**: Ein eingehender Datenstrom kann auf mehrere Überwachungs- oder Logging-Module aufgeteilt werden, ohne die ursprüngliche Verbindung zu beeinträchtigen.

Vergleich mit ähnlichen Bausteinen

Baustein	Ausgänge	Besonderheit
`AB_SPLIT_9_UNGATED`	9	Standard-Split für 1⟶9, generisch
`AB_SPLIT_4`	4	Gleiche Funktionsweise, geringere Anzahl
`AB_MERGE`	–	Fasst mehrere Eingänge zu einem Ausgang zusammen (entgegengesetzte Richtung)

Während `AB_SPLIT_9_UNGATED` eine Quelle auf viele Senken verteilt, realisiert `AB_MERGE` eine Zusammenführung mehrerer Quellen in einen Ausgang. `AB_SPLIT_9_UNGATED` ist daher die ideale Wahl, wenn ein Signal an viele Verbraucher weitergeleitet werden muss.

- `AB_SPLIT_9`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AB_SPLIT_9_UNGATED** bietet eine einfache, aber leistungsfähige Möglichkeit, einen einzelnen Adapter-basierten Datenstrom auf neun parallele Pfade aufzuteilen. Durch seine generische Natur ist er in beliebigen 4diac-Projekten einsetzbar, erfordert keine Zustandsprogrammierung und ermöglicht eine

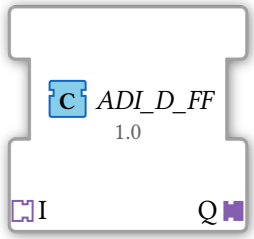
saubere, modulare Architektur. Er ist ein grundlegender Baustein für die serielle oder parallele Verteilung von Adapter-Signalen in der Automatisierungstechnik.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

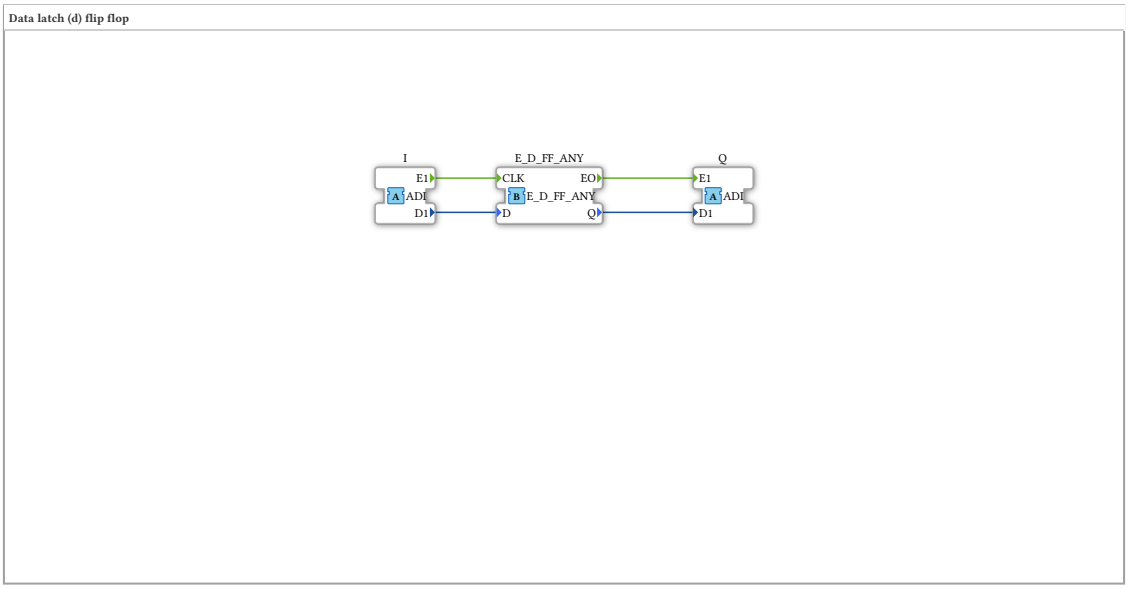
DINT

ADI_D_FF



Value to latch – *ADI*

ADI – Latched value



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ADI_D_FF` realisiert ein Daten-Latch (D-Flipflop) auf Basis von Adaptern. Er dient dazu, einen Booleschen Wert bei Eintreffen eines Ereignisses zu speichern und auszugeben. Der Baustein kapselt den industriellen Standard-FB `E_D_FF_ANY` und kommuniziert ausschließlich über die unidirektionalen Adapter `ADI`.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine separaten Ereignis–Eingänge. Die Ereignissteuerung erfolgt ausschließlich über den Adapter `I`.

Ereignis-Ausgänge

Der FB besitzt keine separaten Ereignis–Ausgänge. Die Ereignisausgabe erfolgt ausschließlich über den Adapter `Q`.

Daten-Eingänge

Der FB besitzt keine separaten Daten–Eingänge. Die Datenaufnahme erfolgt über den Adapter `I`.

Daten-Ausgänge

Der FB besitzt keine separaten Daten–Ausgänge. Die Datenausgabe erfolgt über den Adapter `Q`.

Adapter

Typ	Bezeichnung	Richtung	Beschreibung
`adapter::types::unidirectional::ADI`	`I`	Eingang (Socket)	Liefert das Taktsignal (`E1`) und den zu latchnden Datenwert (`D1`).
`adapter::types::unidirectional::ADI`	`Q`	Ausgang (Plug)	Gibt den gespeicherten Datenwert (`D1`) und ein Bestätigungsereignis (`E1`) aus.

Funktionsweise

Der `ADI_D_FF` arbeitet nach dem Prinzip eines flankengesteuerten D-Flipflops. Jedes am Adapter `I` eintreffende Ereignis `E1` wird als Takt (CLK) interpretiert. Bei jedem Takt wird der aktuelle Datenwert `D1` vom Eingang übernommen, intern gespeichert und am Adapter `Q` sowohl als neuer Datenwert (`D1`) als auch als Bestätigungsereignis (`E1`) ausgegeben.

Im Inneren wird der standardisierte IEC-61499-Baustein `E_D_FF_ANY` verwendet, der exakt diese Funktion bereitstellt. Die Adapter dienen als einheitliche, wiederverwendbare Schnittstellen für Ereignisse und Daten.

Technische Besonderheiten

- Der gesamte Daten- und Ereignisaustausch erfolgt über Adapter vom Typ `ADI` (unidirektionale Ereignis- und Datenübertragung). Dadurch ist der FB leicht in adapterbasierte Architekturen integrierbar.
- Es ist kein eigenständiger Zustandsautomat implementiert; die gesamte Logik wird durch den internen FB `E_D_FF_ANY` realisiert.
- Die Implementierung folgt dem Standard IEC 61499-1 (Annex A) und nutzt die Eclipse 4diac-IDE.

Zustandsübersicht

Der FB besitzt keine expliziten Zustände im Sinne einer Zustandsmaschine. Das Verhalten entspricht einem einfachen Flipflop:

- Bei jedem Takt (Ereignis an `I.E1`) wird der aktuelle Wert von `I.D1` übernommen und an `Q.D1` weitergegeben.
- Der gespeicherte Wert bleibt bis zum nächsten Takt konstant.
- Der Ausgang `Q.E1` signalisiert die Übernahme eines neuen Wertes.

Anwendungsszenarien

- **Synchronisation:** Festhalten eines asynchron eintreffenden Booleschen Signals zu einem definierten Taktzeitpunkt.
- **Zwischenspeicher (Latch):** Halten eines Datenwertes zur Weiterverarbeitung in nachfolgenden FB-Netzwerken.
- **Puffern:** Entkopplung von Ereignis- und Datenquellen von der weiteren Verarbeitung (z. B. bei zeitkritischen Abläufen).

Vergleich mit ähnlichen Bausteinen

Baustein	Funktion	Unterschiede
`ADI_SR_FF`	Set-Reset-Flipflop	Speichert aufgrund von Setzen/Rücksetzen, nicht taktgesteuert.
`ADI_T_FF`	T-Flipflop	Schaltet den Zustand bei jedem Takt um (Toggle).
`ADI_D_FF`	D-Flipflop	Speichert den anliegenden Datenwert bei Taktflanke (wie hier beschrieben).

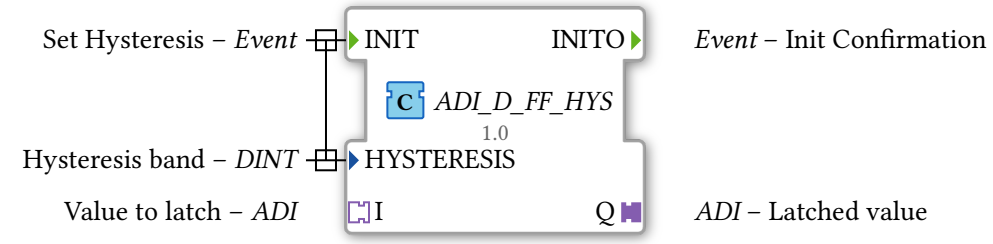
Fazit

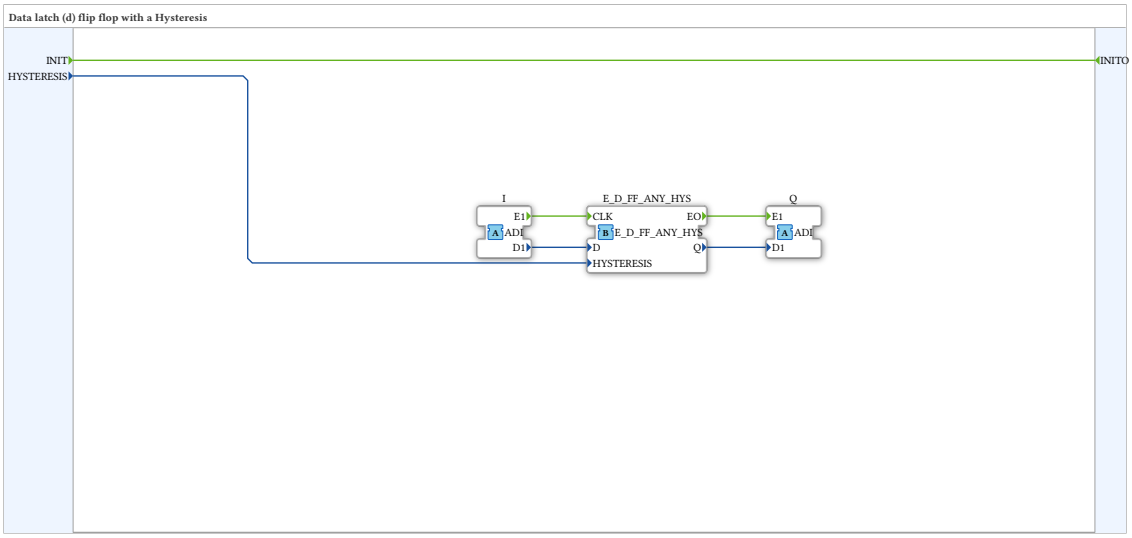
Der `ADI_D_FF` ist ein kompakter, adapterbasierter D-Flipflop-Baustein für die 4diac-IDE. Er bietet eine saubere Trennung von Takt und Datum und ist ideal für den Einsatz in ereignisgesteuerten Automatisierungslösungen geeignet. Die Kapselung durch Adapter erhöht die Wiederverwendbarkeit und vereinfacht die Integration in bestehende Projekte.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ADI_D_FF_HYS





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ADI_D_FF_HYS** realisiert ein datengetaktetes (D-) Flipflop mit einer einstellbaren Hysterese. Er empfängt Werte über einen unidirektionalen Adapter-Eingang `I` und gibt den mit Hysterese übernommenen Zustand über den Adapter-Ausgang `Q` aus. Die Hysterese-Bandbreite wird beim Initialisierungsereignis `INIT` gesetzt.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung
`INIT`	EInit	Setzt die Hysterese-Bandbreite (in Verbindung mit `HYSTERESIS`).

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung
`INITO`	EInit	Bestätigt die Ausführung von `INIT`.

Daten-Eingänge

Variable	Typ	Beschreibung
HYSTERESIS	DINT	Hysterese-Bandbreite (ganzzahliger Wert).

Daten-Ausgänge

Der Baustein besitzt keine direkten Datenausgänge; der latched Wert wird über den Adapter Q bereitgestellt.

Adapter

Schnittstelle	Typ	Richtung	Beschreibung
I	adapter::types::unidirectional::ADI	Socket	Eingangswert (bestehend aus Event E1 und Data D1).
Q	adapter::types::unidirectional::ADI	Plug	Latched Ausgabewert (bestehend aus Event E1 und Data D1).

Hinweis: Der Adapter ADI (unidirectional) stellt standardmäßig einen Event- (E1) und einen Datenkanal (D1) zur Verfügung.

Funktionsweise

1. **Initialisierung**
   Durch ein Ereignis an INIT wird der Wert an HYSTERESIS übernommen und die interne Hysterese-Bandbreite für das Flipflop gesetzt. Das Initialisierungsereignis wird direkt an INITO weitergegeben.
2. **Datenübernahme**
   Der Eingangsadapter I liefert mit jedem Event an I.E1 den aktuellen Datenwert I.D1.
   Der interne Funktionsblock E_D_FF_ANY_HYS übernimmt diesen Wert gemäß der konfigurierten Hysterese:
   - Liegt der Eingangswert **über** dem oberen Schwellwert (gespeicherter Wert + Hysterese), wird der Ausgang auf 1 gesetzt.
   - Liegt der Eingangswert **unter** dem unteren Schwellwert (gespeicherter Wert - Hysterese), wird der Ausgang auf 0 gesetzt.
   - Bleibt der Wert innerhalb des Hysteresebandes, ändert sich der Ausgang nicht.
3. **Ausgabe**
   Der intern latched Wert wird an den Datenausgang Q.D1 weitergegeben. Gleichzeitig wird ein Event an Q.E1 ausgelöst, um nachgeschalteten Logikbausteinen die Datenverarbeitung zu signalisieren.

Technische Besonderheiten

- **Parametrisierbare Hysterese:** Die Hysterese-Bandbreite kann zur Laufzeit über INIT und HYSTERESIS angepasst werden (ganzzahliger Wert, Datentyp DINT).
- **Adapterbasierte Kommunikation:** Der Baustein verwendet vollständig unidirektionale Adapter, die eine saubere Trennung von Signalfluss und Ereignissteuerung ermöglichen. Dadurch eignet er sich besonders für modulare und dezentrale Signalverarbeitung.
- **Interne Realisierung:** Die Hysterese-Logik wird durch den standardisierten FB E_D_FF_ANY_HYS aus der Bibliothek logiBUS::signalprocessing::hysteresis abgebildet.
- **Keine Verzögerung durch INIT:** Die Initialisierung (INIT → INITO) wird im gleichen Zyklus durchgeschleift; die Hysterese steht ab dem Folgezyklus zur Verfügung.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten, sondern basiert auf dem internen Zustand des D-Flipflops E_D_FF_ANY_HYS. Dieser speichert:

- Den aktuellen Ausgangswert (latched).
- Den zuletzt übernommenen Eingangswert (für die Hysteresebetrachtung).

Eine Änderung des Ausgangs erfolgt nur, wenn der neue Eingangswert das Hystereseband verlässt.

Anwendungsszenarien

- **Entprellung von Sensorschaltern:** Ein- und Ausschaltschwellen mit Hysterese verhindern Flankenrauschen.
- **Schwellwertüberwachung mit Rückfallverzögerung:** Temperatur-, Druck- oder Füllstandsregelung mit definierten Ein- und Ausschaltpunkten.
- **Digitalisierung analoger Signale:** Umwandlung eines kontinuierlichen Wertebereichs in einen stabilen Binärwert (Schmitt-Trigger-Verhalten).
- **Steuerung von Aktoren mit Hysterese:** Vermeidung von schnellem Ein-/Ausschalten, z. B. bei Thermostaten oder Pumpen.

Vergleich mit ähnlichen Bausteinen

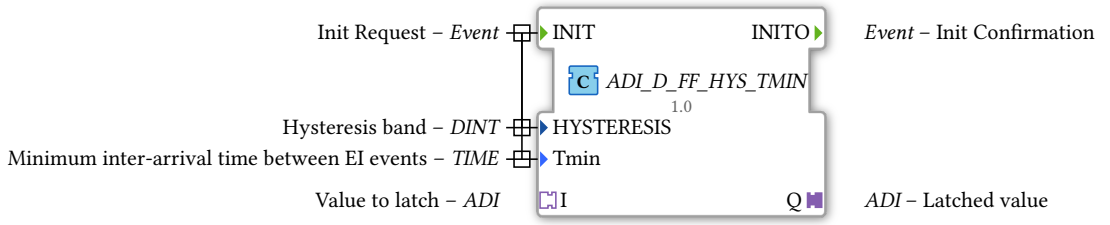
Baustein	Eigenschaften
E_D_FF	Standard-D-Flipflop ohne Hysterese (direkte Übernahme).
E_D_FF_ANY_HYS	Identische Funktion, jedoch mit direkten Datenports statt Adaptern.
Schmitt-Trigger	Ähnliches Verhalten, jedoch keine Parametrierbarkeit der Hysterese zur Laufzeit.

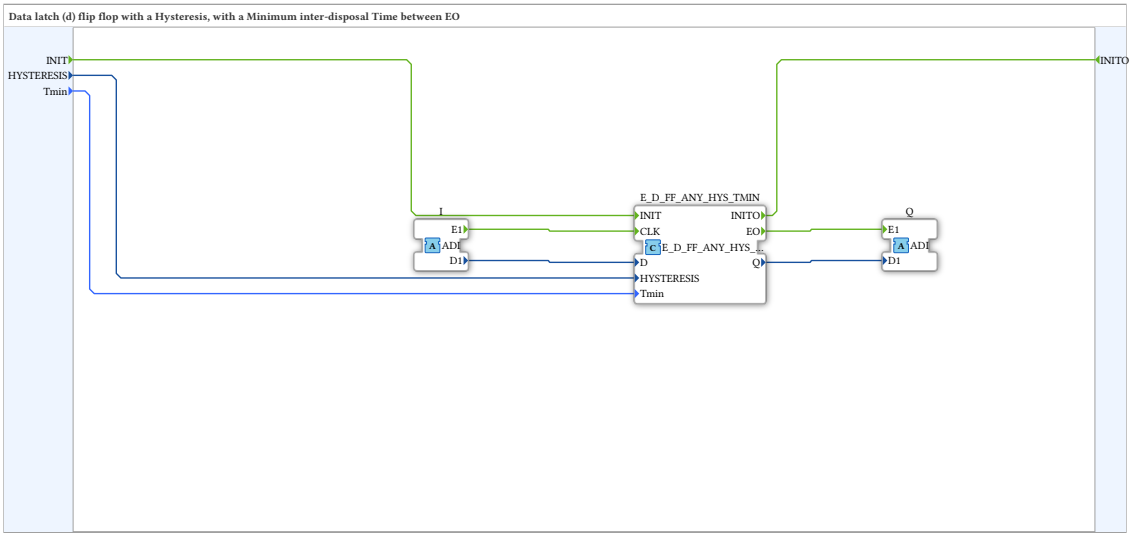
Der **ADI_D_FF_HYS** bietet durch die Adapterschnittstelle eine besonders flexible Einbindung in heterogene Systeme, während die Hysterese zur Laufzeit konfiguriert werden kann.

Fazit

Der **ADI_D_FF_HYS** ist ein robustes, parametrierbares D-Flipflop mit Hysterese, das sich ideal für die Signalverarbeitung mit stabilen Schaltschwellen eignet. Die Verwendung unidirektionaler Adapter erleichtert die Integration in prozessnahe Steuerungsanwendungen und ermöglicht eine klare Trennung von Datentransport und Ereignissteuerung.

ADI_D_FF_HYS_TMIN





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ADI_D_FF_HYS_TMIN** realisiert ein Daten-Latch (D-Flipflop) mit einer Hysterese und einer minimalen Zeitverzögerung zwischen aufeinanderfolgenden Ausgangsereignissen. Er dient der stabilen und entstörten Signalweitergabe in industriellen Steuerungsanwendungen, die auf unidirektionalen Adaptern basieren.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Init-Anforderung. Dieser Ereigniseingang initialisiert den Baustein mit den Parametern `HYSTERESIS` und `Tmin`.

Ereignis-Ausgänge

- **INITO** (EInit): Init-Bestätigung. Wird nach erfolgreicher Initialisierung ausgegeben.

Daten-Eingänge

- **HYSTERESIS** (DINT): Hystereseband. Definiert die Schaltschwelle, um ein Flattern am Ausgang zu verhindern.
- **Tmin** (TIME): Minimale Zeit zwischen zwei Ausgangsereignissen (Inter-Arrival Time). Verhindert zu schnelles Umschalten.

Daten-Ausgänge

Der Baustein besitzt keine eigenen Daten-Ausgänge. Die Ausgangsdaten werden über einen Adapter bereitgestellt.

Adapter

- **I** (Socket, Typ: `adapter::types::unidirectional::ADI`): Der zu latchnde Wert wird über diesen Adaptereingang empfangen (Ereignis und Daten).
- **Q** (Plug, Typ: `adapter::types::unidirectional::ADI`): Der gelatchte Wert wird über diesen Adapterausgang ausgegeben (Ereignis und Daten).

Funktionsweise

Der Baustein verwendet intern den FB `E_D_FF_ANY_HYS_TMIN`. Er verarbeitet die über den Adapter **I** eingehenden Ereignisse (`I.E1`) und Daten (`I.D1`). Die eigentliche Latch-Funktion erfolgt mit einer Hysterese, die durch den Parameter `HYSTERESIS` festgelegt wird. Zusätzlich wird eine minimale Wartezeit `Tmin` zwischen zwei Ausgangsereignissen eingehalten, um eine zu hohe Ereignisfrequenz zu unterdrücken. Die Ausgabe erfolgt über den Adapter **Q** (`Q.E1` und `Q.D1`). Die Initialisierung (`INIT`) setzt die Parameter und aktiviert den internen FB.

Technische Besonderheiten

- **Adapterbasierte Kommunikation:** Der Baustein nutzt unidirektionale Adapter (`ADI`) für den Datenaustausch. Dies ermöglicht eine lose Kopplung von Daten und Ereignissen.
- **Konfigurierbare Hysterese:** Die Hysteresebreite kann anwendungsspezifisch eingestellt werden, um Rauschen oder Prell-Effekte zu unterdrücken.
- **Minimale Ereignisabstandzeit:** Mit `Tmin` lässt sich die maximale Ausgangsfrequenz begrenzen, was zur Entlastung nachfolgender Verarbeitungseinheiten beiträgt.

Zustandsübersicht

Der FB besitzt zwei stabile interne Zustände (z. B. HIGH/LOW oder TRUE/FALSE), die den gelatchten Ausgangswert repräsentieren. Ein Zustandswechsel erfolgt nur, wenn der eingehende Wert (über **I**) die Hystereseschwelle überschreitet und die seit dem letzten Ausgangsereignis verstrichene Zeit mindestens `Tmin` beträgt. Andernfalls wird das Ereignis unterdrückt.

Zustand	Beschreibung
LOW	Ausgangswert ist logisch 0 bzw. niedrig
HIGH	Ausgangswert ist logisch 1 bzw. hoch

Wechsel zwischen den Zuständen erfolgen gemäß der Hysterese- und Zeitbedingungen.

Anwendungsszenarien

- **Entprellung von Schaltern oder Sensoren:** Verhindert mehrfaches Umschalten durch mechanisches Prellen.
- **Signalglättung in der Prozessautomatisierung:** Unterdrückt kleine Störungen oder Rauschen auf analogen oder digitalen Signalen.
- **Ereignisrate-Begrenzung:** Schutz nachfolgender Funktionen vor zu schnellen Ereignisfolgen, z. B. in Zählern oder Kommunikationsschnittstellen.

Vergleich mit ähnlichen Bausteinen

Ein herkömmliches D-Flipflop (z. B. `E_D_FF`) schaltet bei jedem Takt ohne Hysterese und ohne Zeitverzögerung. Der `ADI_D_FF_HYS_TMIN` hingegen bietet:

- **Hysterese:** Verhindert Oszillationen bei langsam ändernden oder verrauschten Signalen.
- **Minimale Inter-Arrival-Zeit:** Reduziert die Ereignisrate und ermöglicht eine gezielte Zeitsteuerung.
- **Adapter-Schnittstelle:** Speziell für den Einsatz in komplexen, adapterbasierten FB-Netzwerken ausgelegt.

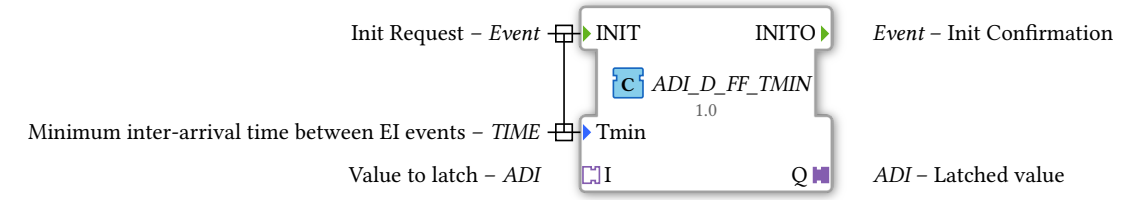
Fazit

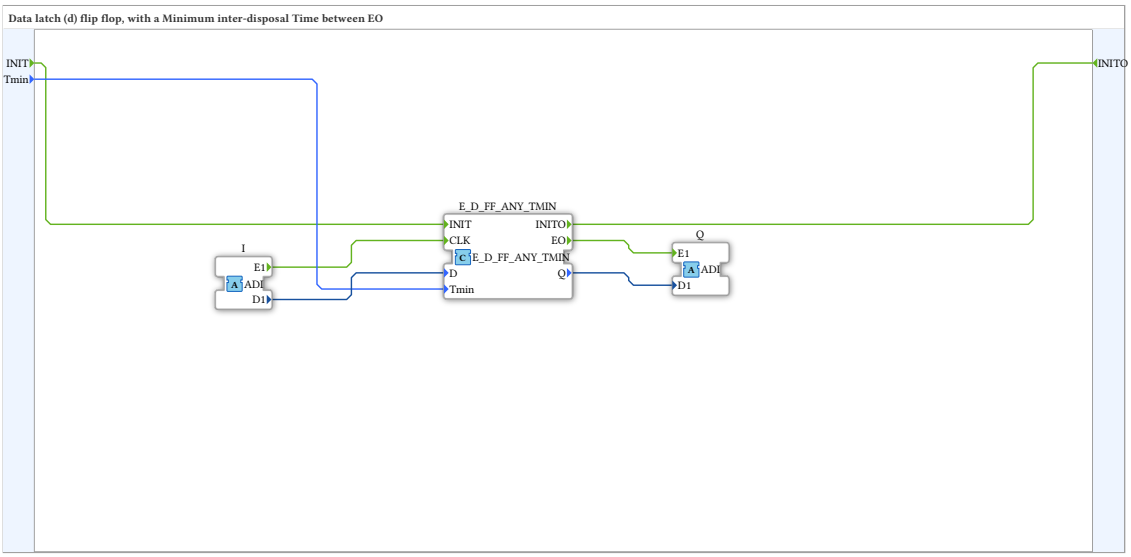
Der `ADI_D_FF_HYS_TMIN` ist ein robuster Funktionsblock für die zuverlässige Signalverarbeitung mit Hysterese und Zeitsteuerung. Er eignet sich besonders für Anwendungen, die eine entstörte, gefilterte und ratenbegrenzte Weitergabe von Ereignissen und Daten erfordern – typischerweise in industriellen Steuerungssystemen, die auf dem 4diac-Framework und unidirektionalen Adaptern basieren.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ADI_D_FF_TMIN





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ADI_D_FF_TMIN** realisiert ein Daten-Latch (D-Flip-Flop) mit einer konfigurierbaren Mindestverweilzeit zwischen zwei aufeinanderfolgenden Übernahmeereignissen. Er dient dazu, einen

Datenwert zu einem bestimmten Ereigniszeitpunkt zu sichern und erst dann wieder zu aktualisieren, wenn eine vorgegebene Zeitspanne (Tmin) seit der letzten Übergabe verstrichen ist.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Init-Anforderung; Übergibt den Parameter Tmin

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Init-Bestätigung

Daten-Eingänge

Name	Typ	Kommentar
Tmin	TIME	Mindestzeit zwischen zwei Ereignissen am Eingang I.E1

Daten-Ausgänge

Keine direkten Datenausgänge – der gelatchte Wert wird über den Adapter-Ausgang Q bereitgestellt.

Adapter

Richtung	Name	Typ	Kommentar
Socket (Eingang)	I	adapter::types::unidirectional::ADI	Wert und Ereignis zum Latchen
Plug (Ausgang)	Q	adapter::types::unidirectional::ADI	Gelatchter Wert und zugehöriges Ausgangsereignis

Funktionsweise

Der FB empfängt über den Adapter-Socket **I** ein Ereignis (I.E1) samt einem Datenwert (I.D1). Nach diesem Ereignis wird der Wert übernommen und am Adapter-Plug **Q** als Q.D1 ausgegeben; gleichzeitig wird ein Ereignis an Q.E1 gesendet.
Zwischen zwei aufeinanderfolgenden Übernahmen muss jedoch mindestens die in **Tmin** festgelegte Zeit vergangen sein. Solange diese Mindestfrist nicht abgelaufen ist, wird ein eingehendes Ereignis an I.E1 ignoriert (der Wert bleibt unverändert).
Der INIT-Eingang setzt den Parameter Tmin und bestätigt dies über INITO, ohne das Flip-Flop selbst zu beeinflussen.

Der FB nutzt intern den Baustein `iec61499::events::E_D_FF_ANY_TMIN`, der die eigentliche Logik (D-Flip-Flop mit Zeitsteuerung) kapselt.

Technische Besonderheiten

- **Adapter-basierte Schnittstelle:** Statt direkt verschalteter Ereignis- und Datenports werden die Ein- und Ausgänge über uni-direktionale Adapter (Typ `ADI`) realisiert. Dies vereinfacht die Wiederverwendung und Trennung von Steuerungs- und Datenflüssen.
- **Mindestzeit-Garantie:** Die Tmin-Zeit wird ab dem Moment des letzten gültigen Ausgangsereignisses gemessen. Bei einem zu frühen Eingangsereignis bleibt der Ausgang unverändert – der alte Wert bleibt erhalten.
- **Initialisierung notwendig:** Der Baustein muss zunächst über INIT mit einem Tmin-Wert initialisiert werden, bevor er auf eingehende Adapter-Ereignisse reagiert.

Zustandsübersicht

Der FB besitzt eine implizite Zustandslogik, die durch den internen `E_D_FF_ANY_TMIN` abgebildet wird:

- **Idle:** Warten auf das erste gültige Ereignis an I.E1 nach der Initialisierung.

- **Latched:** Ein Wert wurde übernommen; die Zeitmessung für Tmin läuft. Neue Ereignisse an I.E1 werden solange ignoriert, bis Tmin abgelaufen ist.
- **Ready:** Tmin ist abgelaufen; der Baustein ist bereit für die nächste Übernahme.

Die Zustandsübergänge werden durch die Ereignisse I.E1 und den internen Timer gesteuert. Eine Initialisierung setzt den Baustein zurück in den Idle-Zustand.

Anwendungsszenarien

- **Taktentkopplung in Industrieanwendungen:** Sicherstellen, dass Daten nur in definierten Mindestabständen übernommen werden (z. B. bei Sensorabfragen mit fester Rasterzeit).
- **Entprellung von Ereignissen:** Vermeidung von Doppelauswertungen bei kurz aufeinanderfolgenden Ereignissen (z. B. Taster- oder Schaltsignale mit Prellzeit).
- **Datenhaltung in sicherheitskritischen Systemen:** Einmal gelatchte Werte bleiben solange gültig, bis die Mindestzeit abgelaufen ist – schützt vor Rauschen oder transienten Störungen.

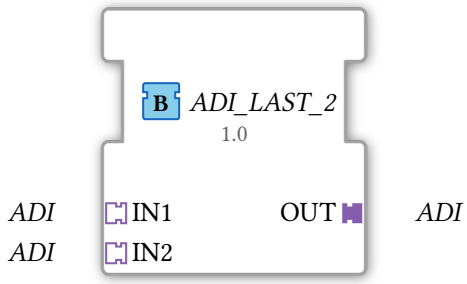
Vergleich mit ähnlichen Bausteinen

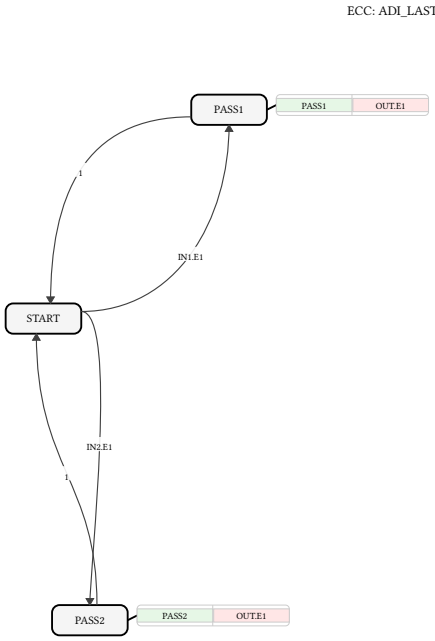
- **Standard-D-Flip-Flop (ohne Tmin):** Ein einfacher D-FF wie z. B. `E_D_FF` übernimmt jeden Wert sofort bei jedem Ereignis. Der vorliegende Baustein erweitert dies um eine zeitliche Sperre.
- **E_D_FF_ANY_TMIN:** Dies ist der direkte interne Basisbaustein. `ADI_D_FF_TMIN` kapselt dessen Funktionalität und bietet eine Adapter-Schnittstelle anstelle separater Ereignis-/Datenports – dies erhöht die Flexibilität bei der Verschaltung mit anderen adapterbasierten Komponenten.

Fazit

Der `ADI_D_FF_TMIN` ist ein nützlicher Funktionsblock für Anwendungen, die eine zuverlässige Datenübernahme mit einstellbarer zeitlicher Entkopplung erfordern. Die Verwendung von Adaptern macht ihn besonders geeignet für modulare, adapterbasierte Architekturen nach IEC 61499. Durch die Mindestzeit wird die Stabilität des Gesamtsystems in Umgebungen mit sporadischen oder prellbehafteten Ereignissen erhöht.

ADI_LAST_2





Einleitung

Der ADI_LAST_2 Funktionsblock führt zwei gleichartige DINT-Adapter-Signale (**IN1**, **IN2**) auf einen gemeinsamen Ausgang (**OUT**) zusammen, nach dem Prinzip **Last Writer Wins**: welcher der beiden Sockets zuletzt ein Ereignis geschrieben hat, gewinnt, und sein Datenwert wird sofort und unverändert an OUT durchgereicht. Es handelt sich ausdrücklich nicht um ein naives Zusammenführen (z. B. Mittelwert oder ODER-Verknüpfung), sondern um eine reine Arbitrierung nach zeitlicher Reihenfolge.

Schnittstellenstruktur

Adapter

Eingangsadapter (Sockets):

- **IN1**: DINT-Adapter (unidirectional) - erste Signalquelle
- **IN2**: DINT-Adapter (unidirectional) - zweite Signalquelle

Ausgangsadapter (Plug):

- **OUT**: DINT-Adapter (unidirectional) - zusammengeführtes Signal

Funktionsweise

Der Baustein ist als einfacher Basic-FB mit einer 3-Zustands-ECC umgesetzt (START, PASS1, PASS2). Trifft an IN1 ein Ereignis ein, wechselt die ECC nach PASS1: der aktuelle Datenwert von IN1 wird nach OUT kopiert und das Adapter-Ereignis an OUT gesendet. Trifft an IN2 ein Ereignis ein, geschieht dasselbe symmetrisch über PASS2. Nach jeder Durchreichung kehrt die ECC sofort wieder nach START zurück und ist bereit für das nächste Ereignis von IN1 oder IN2.

Technische Besonderheiten

- Last-Writer-Wins-Semantik statt naiver Datenverknüpfung
- Einfacher Basic-FB (ECC mit 3 Zuständen), kein zusätzlicher Zwischenspeicher nötig
- Keine Signalverzögerung: der Wert wird im selben Zyklus durchgereicht, in dem das auslösende Ereignis eintrifft
- Beide Eingänge sind gleichberechtigt - es gibt keine feste Priorität außer der zeitlichen Reihenfolge

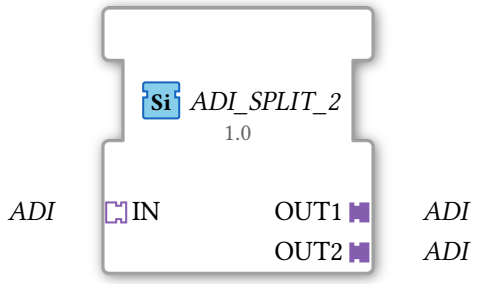
Anwendungsszenarien

- Zusammenführen zweier gleichwertiger DINT-Signalquellen zu einem einzigen nachgeschalteten Adapterpfad (z. B. zwei redundante Sensoren, oder ein manueller und ein automatischer Sollwertgeber)
- Arbitrierung zwischen zwei Bedienstellen, die denselben DINT-Wert schreiben dürfen
- Vereinfachung von Netzwerken, die sonst ein `F_SEL`/`E_RS`-Konstrukt inline verdrahten müssten

⚖️ Vergleich mit ähnlichen Bausteinen

ADI_LAST_2 ist die Umkehrung von `ADI_SPLIT_2`: ADI_SPLIT_2 verteilt 1 Eingang auf 2 Ausgänge, ADI_LAST_2 führt 2 Eingänge auf 1 Ausgang zusammen. Für Mischbetrieb zwischen einem Daten- und einem reinen Ereignisadapter (z. B. AX + AE) ist stattdessen `AX_AE_MERGE` gedacht, das nur das Ereignis mischt, ohne den Datenwert zu verändern.

ADI_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ADI_SPLIT_2` dient dazu, einen eingehenden ADI-Adapter (Socket) auf zwei identische ADI-Ausgangsadapter (Plugs) zu verteilen. Er ist als generischer Baustein konzipiert und ermöglicht eine einfache Signalbündelung in Adapter-basierten Architekturen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine – dieser Funktionsblock besitzt keine Ereignis-Eingänge.

Ereignis-Ausgänge

Keine – dieser Funktionsblock besitzt keine Ereignis-Ausgänge.

Daten-Eingänge

Keine – dieser Funktionsblock besitzt keine separaten Daten-Eingänge. Alle Daten werden über den ADI-Adapter transportiert.

Daten-Ausgänge

Keine – dieser Funktionsblock besitzt keine separaten Daten-Ausgänge. Alle Daten werden über die ADI-Adapter ausgegeben.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::ADI`	`IN`	Socket	Eingehender ADI-Adapter, der auf die beiden Ausgänge verteilt wird.
`adapter::types::unidirectional::ADI`	`OUT1`	Plug	Erster Ausgangs-Adapter (Duplikat des Eingangs).
`adapter::types::unidirectional::ADI`	`OUT2`	Plug	Zweiter Ausgangs-Adapter (Duplikat des Eingangs).

Funktionsweise

Der Baustein leitet den über den Socket `IN` empfangenen ADI-Adapter ohne Änderungen an beide Plugs `OUT1` und `OUT2` weiter. Dadurch stehen die vom Eingangsadapter bereitgestellten Daten und Ereignisse gleichzeitig an beiden Ausgängen zur Verfügung. Es findet keine Verarbeitung, Verzögerung oder Umformung der Daten statt.

Technische Besonderheiten

- **Generischer Baustein:** Der FB ist als generischer Typ (`GEN_ADI_SPLIT`) deklariert und kann in unterschiedlichen ADI-Umgebungen wiederverwendet werden.
- **Kein Zustandsautomat:** Aufgrund der reinen Signalweiterleitung besitzt der FB keinen eigenen Zustandsautomaten (ECC).
- **Adapter-basiert:** Der gesamte Datenaustausch erfolgt ausschließlich über Adapter, nicht über einzelne Daten- oder Ereignis-Ports.

Zustandsübersicht

Dieser Funktionsblock besitzt keinen internen Zustandsautomaten. Die Weiterleitung erfolgt statisch und ohne zeitliche Abhängigkeiten.

Anwendungsszenarien

- **Signalvervielfältigung:** Wenn ein von einem Sensor oder einer Steuerung gelieferter ADI-Adapter an mehrere nachfolgende Funktionseinheiten (z. B. zur Anzeige, Protokollierung oder parallelen Verarbeitung) verteilt werden soll.
- **Architekturentkopplung:** In Systemen, in denen ein Ausgangssignal mehrfach benötigt wird, ohne dass der Quellbaustein die Anzahl der Senken kennen muss.
- **Prototypen und Testaufbauten:** Schnelle Erweiterung einer bestehenden Adapter-Verschaltung um einen zweiten Abzweig.

Vergleich mit ähnlichen Bausteinen

- **Ereignis-/Daten-Splitter:** Konventionelle Splitter-FBs (z. B. `SPLIT` für `BOOL`) arbeiten auf einzelnen Daten- und Ereignis-Kanälen. `ADI_SPLIT_2` hingegen kopiert einen gesamten Adapter inklusive aller enthaltenen Daten und Ereignisse.
- **Mux/Demux-Bausteine:** Multiplexer und Demultiplexer führen eine Zusammenführung oder Verteilung mit Auswahl durch; `ADI_SPLIT_2` verteilt starr und ohne Selektion.
- `ADI_SPLIT_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

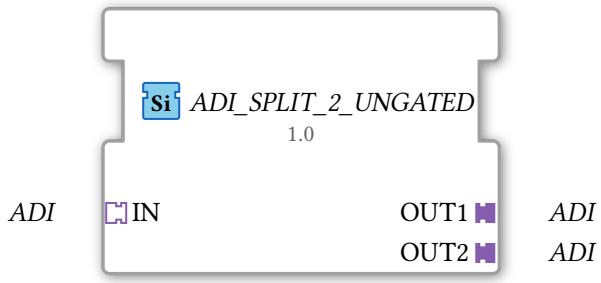
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

`ADI_SPLIT_2` ist ein einfacher, aber nützlicher Funktionsblock zur Verteilung eines ADI-Adapters auf zwei identische Ausgänge. Er erleichtert das Design von Adapter-basierten Steuerungen und vermeidet redundante Quellbausteine. Seine generische Natur und die fehlende interne Logik machen ihn zu einer flexiblen Komponente in IEC-61499-Applikationen.

ADI_SPLIT_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ADI_SPLIT_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ADI_SPLIT_2_UNGATED` dient dazu, einen eingehenden ADI-Adapter (Socket) auf zwei identische ADI-Ausgangsadapter (Plugs) zu verteilen. Er ist als generischer Baustein konzipiert und ermöglicht eine einfache Signalbündelung in Adapter-basierten Architekturen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine – dieser Funktionsblock besitzt keine Ereignis-Eingänge.

Ereignis-Ausgänge

Keine – dieser Funktionsblock besitzt keine Ereignis-Ausgänge.

Daten-Eingänge

Keine – dieser Funktionsblock besitzt keine separaten Daten-Eingänge. Alle Daten werden über den ADI-Adapter transportiert.

Daten-Ausgänge

Keine – dieser Funktionsblock besitzt keine separaten Daten-Ausgänge. Alle Daten werden über die ADI-Adapter ausgegeben.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::ADI`	`IN`	Socket	Eingehender ADI-Adapter, der auf die beiden Ausgänge verteilt wird.
`adapter::types::unidirectional::ADI`	`OUT1`	Plug	Erster Ausgangs-Adapter (Duplikat des Eingangs).
`adapter::types::unidirectional::ADI`	`OUT2`	Plug	Zweiter Ausgangs-Adapter (Duplikat des Eingangs).

Funktionsweise

Der Baustein leitet den über den Socket `IN` empfangenen ADI-Adapter ohne Änderungen an beide Plugs `OUT1` und `OUT2` weiter. Dadurch stehen die vom Eingangsadapter bereitgestellten Daten und Ereignisse gleichzeitig an beiden Ausgängen zur Verfügung. Es findet keine Verarbeitung, Verzögerung oder Umformung der Daten statt.

Technische Besonderheiten

- **Generischer Baustein:** Der FB ist als generischer Typ (`GEN_ADI_SPLIT`) deklariert und kann in unterschiedlichen ADI-Umgebungen wiederverwendet werden.
- **Kein Zustandsautomat:** Aufgrund der reinen Signalweiterleitung besitzt der FB keinen eigenen Zustandsautomaten (ECC).
- **Adapter-basiert:** Der gesamte Datenaustausch erfolgt ausschließlich über Adapter, nicht über einzelne Daten- oder Ereignis-Ports.

Zustandsübersicht

Dieser Funktionsblock besitzt keinen internen Zustandsautomaten. Die Weiterleitung erfolgt statisch und ohne zeitliche Abhängigkeiten.

Anwendungsszenarien

- **Signalvervielfältigung:** Wenn ein von einem Sensor oder einer Steuerung gelieferter ADI-Adapter an mehrere nachfolgende Funktionseinheiten (z. B. zur Anzeige, Protokollierung oder parallelen Verarbeitung) verteilt werden soll.
- **Architekturentkopplung:** In Systemen, in denen ein Ausgangssignal mehrfach benötigt wird, ohne dass der Quellbaustein die Anzahl der Senken kennen muss.
- **Prototypen und Testaufbauten:** Schnelle Erweiterung einer bestehenden Adapter-Verschaltung um einen zweiten Abzweig.

Vergleich mit ähnlichen Bausteinen

- **Ereignis-/Daten-Splitter:** Konventionelle Splitter-FBs (z. B. `SPLIT` für `BOOL`) arbeiten auf einzelnen Daten- und Ereignis-Kanälen. `ADI_SPLIT_2_UNGATED` hingegen kopiert einen gesamten Adapter inklusive aller enthaltenen Daten und Ereignisse.
- **Mux/Demux-Bausteine:** Multiplexer und Demultiplexer führen eine Zusammenführung oder Verteilung mit Auswahl durch; `ADI_SPLIT_2_UNGATED` verteilt starr und ohne Selektion.
- **`ADI_SPLIT_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

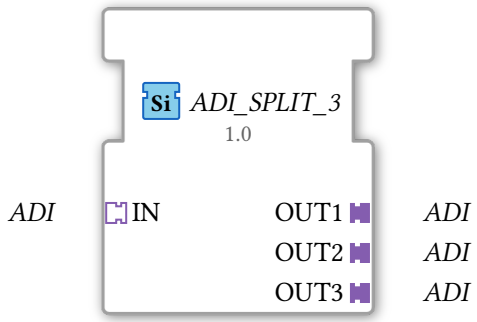
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

`ADI_SPLIT_2_UNGATED` ist ein einfacher, aber nützlicher Funktionsblock zur Verteilung eines ADI-Adapters auf zwei identische Ausgänge. Er erleichtert das Design von Adapter-basierten Steuerungen und vermeidet redundante Quellbausteine. Seine generische Natur und die fehlende interne Logik machen ihn zu einer flexiblen Komponente in IEC-61499-Applikationen.

ADI_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ADI_SPLIT_3** dient dazu, einen eingehenden ADI-Datenstrom (Adapter-Interface) auf drei identische Ausgänge zu verteilen. Er ist als generischer Baustein ausgelegt und ermöglicht eine einfache Signalvervielfachung in adapterbasierten 4diac-Anwendungen.

Schnittstellenstruktur

Der Baustein besitzt ausschließlich Adapter-Schnittstellen – weder Ereignis- noch Datenein-/-ausgänge sind vorhanden.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Bezeichnung	Typ	Richtung	Beschreibung
`IN`	`adapter::types::unidirectional::ADI`	Socket (Eingang)	Eingehender ADI-Datenstrom, der aufgeteilt wird.
`OUT1`	`adapter::types::unidirectional::ADI`	Plug (Ausgang)	Erster ausgehender ADI-Datenstrom (Kopie des Eingangs).
`OUT2`	`adapter::types::unidirectional::ADI`	Plug (Ausgang)	Zweiter ausgehender ADI-Datenstrom.
`OUT3`	`adapter::types::unidirectional::ADI`	Plug (Ausgang)	Dritter ausgehender ADI-Datenstrom.

Funktionsweise

Der Baustein leitet alle an seinem Socket `IN` ankommenden ADI-Datenpakete unverändert an die drei Ausgangs-Plugs `OUT1`, `OUT2` und `OUT3` weiter. Es findet keine Verarbeitung oder Pufferung statt – die Verteilung erfolgt direkt und ohne Verzögerung. Somit entspricht das Verhalten einer passiven Signalvervielfachung (1:N-Splitter).

Technische Besonderheiten

- **Generischer Typ:** Der Baustein ist mittels eines generischen Klassennamens (`GEN_ADI_SPLIT`) definiert, sodass er für beliebige ADI-konforme Datentypen wiederverwendet werden kann.
- **Typ-Hash:** Ein leerer Typ-Hash ist hinterlegt, was auf eine einfache, unparametrisierte Implementierung hinweist.
- **Keine Laufzeitlogik:** Da weder Ereignis- noch Datenschnittstellen existieren, ist der Baustein vollständig durch die Adapterverbindungen gesteuert und benötigt keinen eigenen ECC (Execution Control Chart).

Zustandsübersicht

Der Baustein besitzt keine internen Zustände. Seine Funktionalität beschränkt sich auf das passive Durchreichen von Adapterdaten – es gibt daher keine Zustandsmaschine.

Anwendungsszenarien

- **Signalverteilung in Steuerungssystemen:** Wenn ein ADI-Sensorwert gleichzeitig an mehrere Verbraucher (z. B. Visualisierung, Regelung, Protokollierung) gesendet werden muss.
- **Test- und Simulationsumgebungen:** Ein reales ADI-Signal kann mit `ADI_SPLIT_3` an verschiedene Testmodule oder Simulationskomponenten verteilt werden, ohne dass die Quelle mehrfach angesprochen werden muss.
- **Redundanz und Sicherheitskonzepte:** Verteilung eines sicherheitskritischen Signals an mehrere unabhängige Auswerteeinheiten.

Vergleich mit ähnlichen Bausteinen

- **`ADI_SPLIT_2`** – Aufteilung auf zwei Ausgänge, ansonsten identische Funktionalität.
- **`ADI_MERGE`** – Fasst mehrere ADI-Eingänge zu einem Ausgang zusammen (Gegenstück zum Splitter).
- **`ADI_SELECT`** – Wählt einen von mehreren ADI-Eingängen basierend auf einem Steuersignal aus (keine reine Verteilung).

Der `ADI_SPLIT_3` ist speziell für den Fall optimiert, dass genau drei identische Kopien eines ADI-Signals benötigt werden. Falls mehr oder weniger Ausgänge erforderlich sind, können andere Splitter-Varianten oder Kombinationen eingesetzt werden.

- **`ADI_SPLIT_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

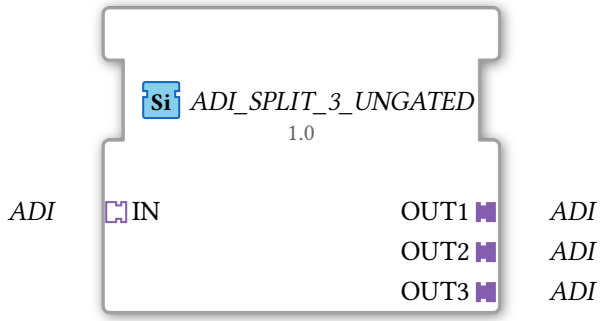
Der **ADI_SPLIT_3** ist ein einfacher, aber nützlicher generischer Baustein zur Vervielfachung von ADI-Datenströmen auf drei Wege. Durch seine reine Adapter-Schnittstelle lässt er sich nahtlos in bestehende 4diac-Projekte integrieren, ohne zusätzliche Ereignis- oder Datenverarbeitung zu erfordern. Er eignet sich besonders für passive Signalverteilungen in modularen, adapterbasierten Automatisierungslösungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ADI_SPLIT_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ADI_SPLIT_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ADI_SPLIT_3_UNGATED** dient dazu, einen eingehenden ADI-Datenstrom (Adapter-Interface) auf drei identische Ausgänge zu verteilen. Er ist als generischer Baustein ausgelegt und ermöglicht eine einfache Signalvervielfachung in adapterbasierten 4diac-Anwendungen.

Schnittstellenstruktur

Der Baustein besitzt ausschließlich Adapter-Schnittstellen – weder Ereignis- noch Datenein-/-ausgänge sind vorhanden.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Bezeichnung	Typ	Richtung	Beschreibung
`IN`	`adapter::types::unidirectional::ADI`	Socket (Eingang)	Eingehender ADI-Datenstrom, der aufgeteilt wird.
`OUT1`	`adapter::types::unidirectional::ADI`	Plug (Ausgang)	Erster ausgehender ADI-Datenstrom (Kopie des Eingangs).
`OUT2`	`adapter::types::unidirectional::ADI`	Plug (Ausgang)	Zweiter ausgehender ADI-Datenstrom.
`OUT3`	`adapter::types::unidirectional::ADI`	Plug (Ausgang)	Dritter ausgehender ADI-Datenstrom.

Funktionsweise

Der Baustein leitet alle an seinem Socket `IN` ankommenden ADI-Datenpakete unverändert an die drei Ausgangs-Plugs `OUT1`, `OUT2` und `OUT3` weiter. Es findet keine Verarbeitung oder Pufferung statt – die Verteilung erfolgt direkt und ohne Verzögerung. Somit entspricht das Verhalten einer passiven Signalvervielfachung (1:N-Splitter).

Technische Besonderheiten

- **Generischer Typ:** Der Baustein ist mittels eines generischen Klassennamens (`GEN_ADI_SPLIT`) definiert, sodass er für beliebige ADI-konforme Datentypen wiederverwendet werden kann.
- **Typ-Hash:** Ein leerer Typ-Hash ist hinterlegt, was auf eine einfache, unparametrisierte Implementierung hinweist.
- **Keine Laufzeitlogik:** Da weder Ereignis- noch Datenschnittstellen existieren, ist der Baustein vollständig durch die Adapterverbindungen gesteuert und benötigt keinen eigenen ECC (Execution Control Chart).

Zustandsübersicht

Der Baustein besitzt keine internen Zustände. Seine Funktionalität beschränkt sich auf das passive Durchreichen von Adapterdaten – es gibt daher keine Zustandsmaschine.

Anwendungsszenarien

- **Signalverteilung in Steuerungssystemen:** Wenn ein ADI-Sensorwert gleichzeitig an mehrere Verbraucher (z. B. Visualisierung, Regelung, Protokollierung) gesendet werden muss.
- **Test- und Simulationsumgebungen:** Ein reales ADI-Signal kann mit `ADI_SPLIT_3_UNGATED` an verschiedene Testmodule oder Simulationskomponenten verteilt werden, ohne dass die Quelle mehrfach angesprochen werden muss.
- **Redundanz und Sicherheitskonzepte:** Verteilung eines sicherheitskritischen Signals an mehrere unabhängige Auswerteeinheiten.

Vergleich mit ähnlichen Bausteinen

- **`ADI_SPLIT_2`** – Aufteilung auf zwei Ausgänge, ansonsten identische Funktionalität.
- **`ADI_MERGE`** – Fasst mehrere ADI-Eingänge zu einem Ausgang zusammen (Gegenstück zum Splitter).
- **`ADI_SELECT`** – Wählt einen von mehreren ADI-Eingängen basierend auf einem Steuersignal aus (keine reine Verteilung).

Der `ADI_SPLIT_3_UNGATED` ist speziell für den Fall optimiert, dass genau drei identische Kopien eines ADI-Signals benötigt werden. Falls mehr oder weniger Ausgänge erforderlich sind, können andere Splitter-Varianten oder Kombinationen eingesetzt werden.

- **`ADI_SPLIT_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

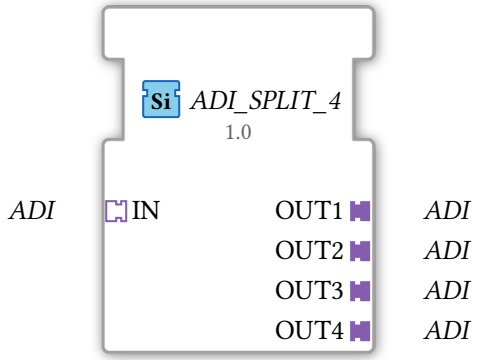
Fazit

Der **ADI_SPLIT_3_UNGATED** ist ein einfacher, aber nützlicher generischer Baustein zur Vervielfachung von ADI-Datenströmen auf drei Wege. Durch seine reine Adapter-Schnittstelle lässt er sich nahtlos in bestehende 4diac-Projekte integrieren, ohne zusätzliche Ereignis- oder Datenverarbeitung zu erfordern. Er eignet sich besonders für passive Signalverteilungen in modularen, adapterbasierten Automatisierungslösungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ADI_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ADI_SPLIT_4** dient als generischer Verteiler (Splitter) für einen einzelnen ADI-Datenstrom. Er empfängt einen ADI-Eingang über die Socket-Schnittstelle `IN` und stellt diesen auf vier identische ADI-Ausgänge über die Plug-Schnittstellen `OUT1`, `OUT2`, `OUT3` und `OUT4` zur Verfügung. Dieser Baustein ist ideal, wenn ein ADI-Signal parallel an mehrere nachfolgende Komponenten weitergeleitet werden soll.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Typ	Name	Richtung	Beschreibung
ADI (unidirektional)	IN	Socket	Empfang des zu verteilenden ADI-Datenstroms.
ADI (unidirektional)	OUT1	Plug	Erster Ausgang – identische Kopie des Eingangssignals.
ADI (unidirektional)	OUT2	Plug	Zweiter Ausgang – identische Kopie des Eingangssignals.
ADI (unidirektional)	OUT3	Plug	Dritter Ausgang – identische Kopie des Eingangssignals.
ADI (unidirektional)	OUT4	Plug	Vierter Ausgang – identische Kopie des Eingangssignals.

Funktionsweise

Der Baustein arbeitet als reine **Vervielfältigung** des ADI-Signals. Sobald ein gültiger ADI-Datenstrom am Socket `IN` anliegt, wird dieser unverändert und ohne Verzögerung an alle vier Plugs (`OUT1` … `OUT4`) weitergegeben. Es findet keinerlei Logik, Filterung oder Datenmanipulation statt – das Verhalten entspricht einem passiven Signalverteiler.

Da es sich um einen generischen Funktionsblock (generic FB) handelt, wird der zugrundeliegende ADI-Typ erst zur Laufzeit oder beim Einbinden in ein Projekt konkret bestimmt. Die Schnittstellendefinition selbst ist typunabhängig.

Technische Besonderheiten

- **Generischer Typ** – Der Baustein wird als `GEN_ADI_SPLIT` deklariert und kann mit jedem konkreten ADI-Adaptertyp instanziiert werden.
- **Keine Ereignisse** – Die Signalverteilung erfolgt rein über die Adapter-Schnittstellen; es sind keine Ereigniseingänge oder -ausgänge vorhanden.
- **Skalierbarkeit** – Der Baustein ist auf genau vier Ausgänge festgelegt. Für andere Anzahlen stehen abweichende Varianten (z. B. `ADI_SPLIT_2`) zur Verfügung.

Zustandsübersicht

Der Funktionsblock besitzt keine explizite Zustandsmaschine (ECC). Da er ausschließlich eine passive Signalweitergabe realisiert, ist kein interner Zustand erforderlich.

Anwendungsszenarien

- **Parallelbetrieb** – Ein von einem Sensor oder Vorgängerbaustein bereitgestellter ADI-Datenstrom soll gleichzeitig von mehreren Aktoren, Logikblöcken oder Visualisierungskomponenten genutzt werden.
- **Prototyping** – In der Entwicklungsphase können mit dem Splitter mehrere Pfade desselben Signals auf einfache Weise getestet werden.
- **Redundanz** – Das ADI-Signal kann an mehrere redundante Auswertepfade verteilt werden.

Vergleich mit ähnlichen Bausteinen

- **ADI_SPLIT_2** – Verteilt ein ADI-Signal auf zwei Ausgänge. Identische Funktionsweise, aber geringere Anzahl an Ausgängen.
- **ADI_MERGE** – Führt mehrere ADI-Eingänge zu einem Ausgang zusammen (Gegenstück zum Splitter).
- **Manuelle Verdrahtung** – Alternativ könnte die Verteilung auch durch mehrfaches Anschließen des gleichen Ausgangs erfolgen, jedoch ist der Splitter in der 4diac-IDE die sauberere und wiederverwendbare Lösung.
- `ADI_SPLIT_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

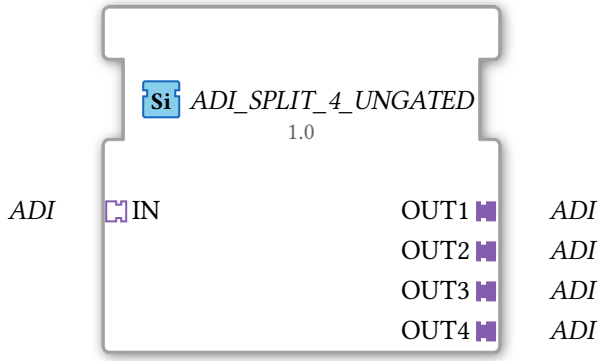
Der **ADI_SPLIT_4** ist ein einfacher, aber nützlicher generischer Funktionsblock zur Vervielfachung eines ADI-Datenstroms. Er erfordert keine Konfiguration und fügt sich nahtlos in modulare Automatisierungsprojekte ein. Durch die Adapter-basierte Schnittstelle bleibt er flexibel und lässt sich ohne Änderungen an der internen Logik in unterschiedliche Umgebungen einbinden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ADI_SPLIT_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ADI_SPLIT_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ADI_SPLIT_4_UNGATED** dient als generischer Verteiler (Splitter) für einen einzelnen ADI-Datenstrom. Er empfängt einen ADI-Eingang über die Socket-Schnittstelle `IN` und stellt diesen auf vier identische ADI-Ausgänge über die Plug-Schnittstellen `OUT1`, `OUT2`, `OUT3` und `OUT4` zur Verfügung. Dieser Baustein ist ideal, wenn ein ADI-Signal parallel an mehrere nachfolgende Komponenten weitergeleitet werden soll.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Typ	Name	Richtung	Beschreibung

ADI (unidirektional)	IN	Socket	Empfang des zu verteilenden ADI-Datenstroms.
ADI (unidirektional)	OUT1	Plug	Erster Ausgang – identische Kopie des Eingangssignals.
ADI (unidirektional)	OUT2	Plug	Zweiter Ausgang – identische Kopie des Eingangssignals.
ADI (unidirektional)	OUT3	Plug	Dritter Ausgang – identische Kopie des Eingangssignals.
ADI (unidirektional)	OUT4	Plug	Vierter Ausgang – identische Kopie des Eingangssignals.

Funktionsweise

Der Baustein arbeitet als reine **Vervielfältigung** des ADI-Signals. Sobald ein gültiger ADI-Datenstrom am Socket `IN` anliegt, wird dieser unverändert und ohne Verzögerung an alle vier Plugs (`OUT1` … `OUT4`) weitergegeben. Es findet keinerlei Logik, Filterung oder Datenmanipulation statt – das Verhalten entspricht einem passiven Signalverteiler.

Da es sich um einen generischen Funktionsblock (generic FB) handelt, wird der zugrundeliegende ADI-Typ erst zur Laufzeit oder beim Einbinden in ein Projekt konkret bestimmt. Die Schnittstellendefinition selbst ist typunabhängig.

Technische Besonderheiten

- **Generischer Typ** – Der Baustein wird als `GEN_ADI_SPLIT` deklariert und kann mit jedem konkreten ADI-Adaptertyp instanziiert werden.
- **Keine Ereignisse** – Die Signalverteilung erfolgt rein über die Adapter-Schnittstellen; es sind keine Ereigniseingänge oder -ausgänge vorhanden.
- **Skalierbarkeit** – Der Baustein ist auf genau vier Ausgänge festgelegt. Für andere Anzahlen stehen abweichende Varianten (z. B. `ADI_SPLIT_2`) zur Verfügung.

Zustandsübersicht

Der Funktionsblock besitzt keine explizite Zustandsmaschine (ECC). Da er ausschließlich eine passive Signalweitergabe realisiert, ist kein interner Zustand erforderlich.

Anwendungsszenarien

- **Parallelbetrieb** – Ein von einem Sensor oder Vorgängerbaustein bereitgestellter ADI-Datenstrom soll gleichzeitig von mehreren Aktoren, Logikblöcken oder Visualisierungskomponenten genutzt werden.
- **Prototyping** – In der Entwicklungsphase können mit dem Splitter mehrere Pfade desselben Signals auf einfache Weise getestet werden.
- **Redundanz** – Das ADI-Signal kann an mehrere redundante Auswertepfade verteilt werden.

Vergleich mit ähnlichen Bausteinen

- **ADI_SPLIT_2** – Verteilt ein ADI-Signal auf zwei Ausgänge. Identische Funktionsweise, aber geringere Anzahl an Ausgängen.
- **ADI_MERGE** – Führt mehrere ADI-Eingänge zu einem Ausgang zusammen (Gegenstück zum Splitter).
- **Manuelle Verdrahtung** – Alternativ könnte die Verteilung auch durch mehrfaches Anschließen des gleichen Ausgangs erfolgen, jedoch ist der Splitter in der 4diac-IDE die sauberere und wiederverwendbare Lösung.
- `ADI_SPLIT_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

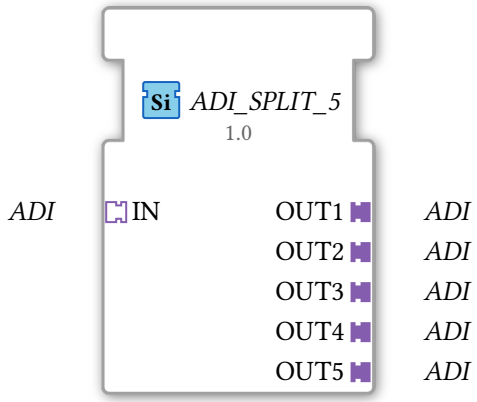
Fazit

Der **ADI_SPLIT_4_UNGATED** ist ein einfacher, aber nützlicher generischer Funktionsblock zur Vervielfachung eines ADI-Datenstroms. Er erfordert keine Konfiguration und fügt sich nahtlos in modulare Automatisierungsprojekte ein. Durch die Adapter-basierte Schnittstelle bleibt er flexibel und lässt sich ohne Änderungen an der internen Logik in unterschiedliche Umgebungen einbinden.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ADI_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ADI_SPLIT_5** dient dazu, einen eingehenden ADI-Adapter (unidirektional) auf fünf identische Ausgangs-Adapter aufzuteilen. Er ist als generischer Baustein (Generic FB) konzipiert und kann in IEC 61499-basierten Systemen verwendet werden, um ein Signal oder einen Datenstrom an mehrere nachfolgende Funktionsblöcke weiterzuleiten.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

- **IN** (Socket, Typ: `adapter::types::unidirectional::ADI`) Eingangs-Adapter, der das zu verteilende Signal liefert.

Daten-Ausgänge

- **OUT1** (Plug, Typ: `adapter::types::unidirectional::ADI`)
- **OUT2** (Plug, Typ: `adapter::types::unidirectional::ADI`)
- **OUT3** (Plug, Typ: `adapter::types::unidirectional::ADI`)
- **OUT4** (Plug, Typ: `adapter::types::unidirectional::ADI`)
- **OUT5** (Plug, Typ: `adapter::types::unidirectional::ADI`)

Alle Ausgänge sind unabhängige Instanzen des gleichen Adaptertyps und erhalten identische Daten.

Adapter

Dieser Baustein arbeitet ausschließlich über Adapter-Schnittstellen. Es werden keine Ereignis- oder Daten-Ein-/Ausgänge im klassischen Sinne verwendet. Die gesamte Kommunikation erfolgt über die ADI-Adapter, welche die Signal- und Datenweiterleitung implizit steuern.

Funktionsweise

Der FB **ADI_SPLIT_5** empfängt über den Socket **IN** einen ADI-Adapter und leitet die darin enthaltenen Daten und Ereignisse parallel an die fünf Plugs **OUT1** bis **OUT5** weiter. Es findet keine Verarbeitung oder Umformung der Daten statt – der Baustein fungiert als passiver Verteiler (Fan-Out).

Da es sich um unidirektionale ADI-Adapter handelt, wird das Signal nur von der Eingangsseite zur Ausgangsseite weitergegeben; Rückwirkungen von den Ausgängen sind nicht vorgesehen.

Technische Besonderheiten

- **Generischer Baustein**: Die Klasse `GEN_ADI_SPLIT` erlaubt eine flexible Wiederverwendung, z. B. durch Parametrisierung des Typs oder der Anzahl der Ausgänge (hier fest auf fünf).
- **Adapterbasiert**: Keine direkten Ereignis- oder Daten-Ein-/Ausgänge – die komplette Signalweitergabe erfolgt über die Adapter-Schnittstellen.
- **Unidirektionalität**: Die Adapter sind als `unidirectional` definiert, d. h. Daten fließen nur in eine Richtung (von IN zu OUT1-5).
- Der Baustein enthält keine interne Zustandslogik oder zeitliche Steuerung.

Zustandsübersicht

Der **ADI_SPLIT_5** besitzt keine eigenen Zustände. Er ist ein rein kombinatorischer Baustein, der sofort auf eingehende Adapter-Daten reagiert und diese unverändert an alle Ausgänge weitergibt.

Anwendungsszenarien

- **Signalkaskade**: Aufteilung eines Sensorsignals (z. B. Temperatur, Druck) an mehrere Auswerte- und Steuerungsblöcke.
- **Diagnose und Überwachung**: Gleichzeitige Verteilung eines Datenstroms an Verarbeitungs- und Logging-Module.
- **Lastverteilung**: Parallele Verarbeitung desselben Eingangs in unterschiedlichen Algorithmen (z. B. Filter, Grenzwertüberwachung, Visualisierung).

Vergleich mit ähnlichen Bausteinen

Im IEC 61499-Umfeld existieren häufig Varianten wie **SPLIT_2**, **SPLIT_3** oder **SPLIT_N**, die eine unterschiedliche Anzahl von Ausgängen bereitstellen. Der **ADI_SPLIT_5** ist eine spezielle Realisierung für exakt fünf Ausgänge unter Verwendung von unidirektionalen ADI-Adapter-Schnittstellen. Im Gegensatz zu ereignisbasierten Split-Bausteinen (z. B. `E_SPLIT`) erfolgt hier die Verteilung über Adapter, was eine typengebundene und typischerweise datenorientierte Kommunikation ermöglicht.

- `ADI_SPLIT_5_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

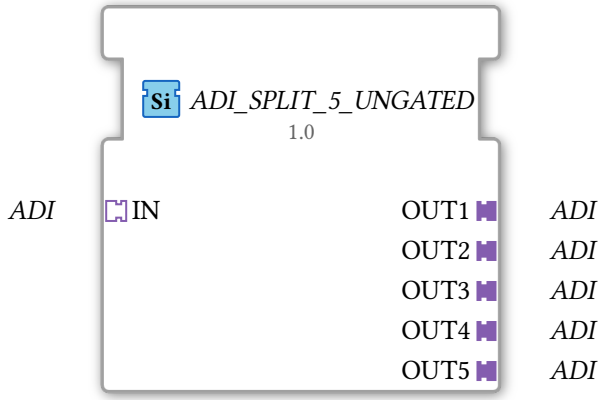
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der **ADI_SPLIT_5** ist ein einfacher, aber nützlicher Funktionsblock zur Verteilung eines unidirektionalen ADI-Signals auf bis zu fünf nachfolgende Bausteine. Er eignet sich besonders für modulare Automatisierungslösungen, bei denen ein Eingangssignal mehrfach genutzt werden muss, ohne dass eine zusätzliche Steuerungslogik erforderlich ist. Durch die generische Implementierung kann er in verschiedenen Kontexten wiederverwendet werden.

ADI_SPLIT_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ADI_SPLIT_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ADI_SPLIT_5_UNGATED** dient dazu, einen eingehenden ADI-Adapter (unidirektional) auf fünf identische Ausgangs-Adapter aufzuteilen. Er ist als generischer Baustein (Generic FB) konzipiert und kann in IEC 61499-basierten Systemen verwendet werden, um ein Signal oder einen Datenstrom an mehrere nachfolgende Funktionsblöcke weiterzuleiten.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

- **IN** (Socket, Typ: `adapter::types::unidirectional::ADI`) Eingangs-Adapter, der das zu verteilende Signal liefert.

Daten-Ausgänge

- **OUT1** (Plug, Typ: `adapter::types::unidirectional::ADI`)
- **OUT2** (Plug, Typ: `adapter::types::unidirectional::ADI`)
- **OUT3** (Plug, Typ: `adapter::types::unidirectional::ADI`)
- **OUT4** (Plug, Typ: `adapter::types::unidirectional::ADI`)
- **OUT5** (Plug, Typ: `adapter::types::unidirectional::ADI`)

Alle Ausgänge sind unabhängige Instanzen des gleichen Adaptertyps und erhalten identische Daten.

Adapter

Dieser Baustein arbeitet ausschließlich über Adapter-Schnittstellen. Es werden keine Ereignis- oder Daten-Ein-/Ausgänge im klassischen Sinne verwendet. Die gesamte Kommunikation erfolgt über die ADI-Adapter, welche die Signal- und Datenweiterleitung implizit steuern.

Funktionsweise

Der FB **ADI_SPLIT_5_UNGATED** empfängt über den Socket **IN** einen ADI-Adapter und leitet die darin enthaltenen Daten und Ereignisse parallel an die fünf Plugs **OUT1** bis **OUT5** weiter. Es findet keine Verarbeitung oder Umformung der Daten statt – der Baustein fungiert als passiver Verteiler (Fan-Out).

Da es sich um unidirektionale ADI-Adapter handelt, wird das Signal nur von der Eingangsseite zur Ausgangsseite weitergegeben; Rückwirkungen von den Ausgängen sind nicht vorgesehen.

Technische Besonderheiten

- **Generischer Baustein**: Die Klasse `GEN_ADI_SPLIT` erlaubt eine flexible Wiederverwendung, z. B. durch Parametrisierung des Typs oder der Anzahl der Ausgänge (hier fest auf fünf).

- **Adapterbasiert**: Keine direkten Ereignis- oder Daten-Ein-/Ausgänge – die komplette Signalweitergabe erfolgt über die Adapter-Schnittstellen.
- **Unidirektionalität**: Die Adapter sind als `unidirectional` definiert, d. h. Daten fließen nur in eine Richtung (von IN zu OUT1-5).
- Der Baustein enthält keine interne Zustandslogik oder zeitliche Steuerung.

Zustandsübersicht

Der **ADI_SPLIT_5_UNGATED** besitzt keine eigenen Zustände. Er ist ein rein kombinatorischer Baustein, der sofort auf eingehende Adapter-Daten reagiert und diese unverändert an alle Ausgänge weitergibt.

Anwendungsszenarien

- **Signalkaskade**: Aufteilung eines Sensorsignals (z. B. Temperatur, Druck) an mehrere Auswerte- und Steuerungsblöcke.
- **Diagnose und Überwachung**: Gleichzeitige Verteilung eines Datenstroms an Verarbeitungs- und Logging-Module.
- **Lastverteilung**: Parallele Verarbeitung desselben Eingangs in unterschiedlichen Algorithmen (z. B. Filter, Grenzwertüberwachung, Visualisierung).

Vergleich mit ähnlichen Bausteinen

Im IEC 61499-Umfeld existieren häufig Varianten wie **SPLIT_2**, **SPLIT_3** oder **SPLIT_N**, die eine unterschiedliche Anzahl von Ausgängen bereitstellen. Der **ADI_SPLIT_5_UNGATED** ist eine spezielle Realisierung für exakt fünf Ausgänge unter Verwendung von unidirektionalen ADI-Adapter-Schnittstellen. Im Gegensatz zu ereignisbasierten Split-Bausteinen (z. B. `E_SPLIT`) erfolgt hier die Verteilung über Adapter, was eine typengebundene und typischerweise datenorientierte Kommunikation ermöglicht.

- `ADI_SPLIT_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

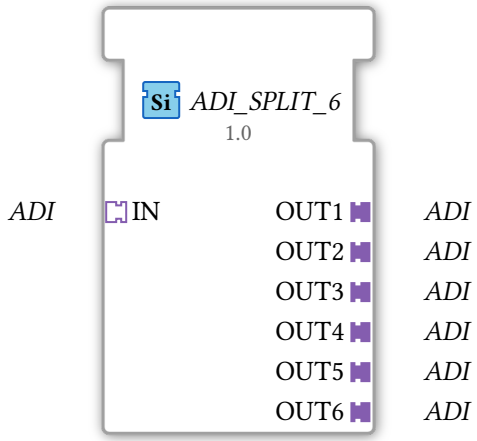
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **ADI_SPLIT_5_UNGATED** ist ein einfacher, aber nützlicher Funktionsblock zur Verteilung eines unidirektionalen ADI-Signals auf bis zu fünf nachfolgende Bausteine. Er eignet sich besonders für modulare Automatisierungslösungen, bei denen ein Eingangssignal mehrfach genutzt werden muss, ohne dass eine zusätzliche Steuerungslogik erforderlich ist. Durch die generische Implementierung kann er in verschiedenen Kontexten wiederverwendet werden.

ADI_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock ADI_SPLIT_6 dient dazu, einen einzelnen ADI-Adapter-Eingang (unidirektional) auf sechs identische ADI-Adapter-Ausgänge zu verteilen. Er ist als generischer Baustein für das Eclipse 4diac Framework konzipiert und wird typischerweise eingesetzt, wenn ein ADI-Signal an mehrere nachfolgende Komponenten weitergeleitet werden muss.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Der Baustein kommuniziert ausschließlich über Adapter-Schnittstellen:
Socket (Eingang):

- `IN`: Typ `adapter::types::unidirectional::ADI` – nimmt ein unidirektionales ADI-Signal entgegen.

Plugs (Ausgänge):

- `OUT1` bis `OUT6`: jeweils Typ `adapter::types::unidirectional::ADI` – geben das eingehende Signal identisch auf sechs parallele Ausgänge weiter.

Funktionsweise

Der FB führt eine reine 1:6-Verteilung des ADI-Adapter-Signals durch. Alle Daten und Ereignisse, die am Socket `IN` ankommen, werden unverändert und zeitgleich an alle sechs Plugs `OUT1`…`OUT6` weitergegeben. Es findet keine Pufferung, Filterung oder Verzögerung statt. Der Baustein arbeitet somit als passiver Splitter auf Adapterebene.

Technische Besonderheiten

- **Generische Implementierung:** Der FB ist als generischer Baustein (`eclipse4diac::core::GenericClassName = 'GEN_ADI_SPLIT'`) realisiert und kann in verschiedenen Projekten wiederverwendet werden.
- **Unidirektionale Adapter:** Die verwendeten ADI-Adapter sind unidirektional ausgelegt; eine Rückkopplung über die Ausgänge ist nicht vorgesehen.
- **Keine eigenen Zustände:** Da der FB keine Ereignis- oder Datenverarbeitung durchführt, besitzt er keinen internen Zustandsautomaten.
- **Performance:** Durch die direkte Weiterleitung ohne Logik oder Speicher entsteht praktisch keine Latenz.

Zustandsübersicht

Der Baustein verfügt über keinen Zustandsautomaten oder explizite Betriebszustände. Er ist stets aktiv und gibt das Eingangssignal sofort an alle Ausgänge weiter.

Anwendungsszenarien

- **Signalverteilung in Steuerungssystemen:** Ein von einem Sensor oder einer zentralen Logik bereitgestelltes ADI-Signal soll parallel an mehrere Aktoren, Anzeigen oder nachgelagerte Funktionen gesendet werden.
- **Testumgebungen:** Bei der Simulation oder beim Debugging kann ein einzelnes Signal auf mehrere Empfänger aufgeteilt werden, um verschiedene Komponenten gleichzeitig zu testen.
- **Redundanzabbildung:** Falls ein Signal mehrfach benötigt wird (z. B. für Überwachungs- und Steuerungspfade), kann dieser Splitter eine saubere Entkopplung der Ausgänge ermöglichen.

Vergleich mit ähnlichen Bausteinen

- **SPLIT für Standard-Datentypen:** Viele Frameworks bieten Splitter für einfache Datentypen (z. B. `SPLIT_INT`). Der `ADI_SPLIT_6` ist speziell für den Adapter-Typ `ADI` ausgelegt und arbeitet auf einer höheren Abstraktionsebene (Adapter statt einzelner Datenpunkte).
- **ADI_MERGE / ADI_COMBINE:** Während dieser Baustein ein Signal verteilt, fassen andere Bausteine mehrere Adapter-Eingänge zu einem Ausgang zusammen.
- **Weitere Splitter-Varianten:** Es existieren möglicherweise Splitter für andere Adapter-Typen (z. B. bidirektionale ADI). Der vorliegende FB beschränkt sich auf unidirektionale Signale.
- `ADI_SPLIT_6_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

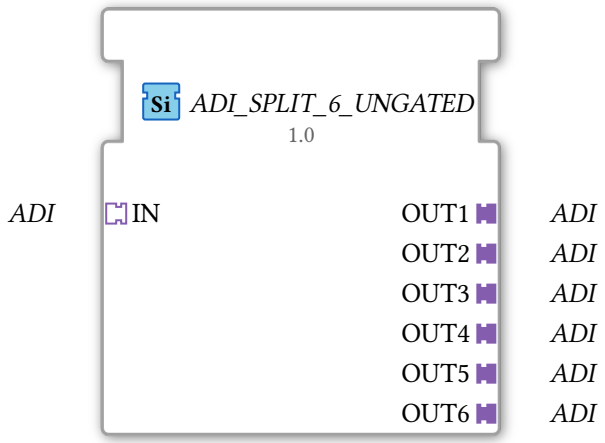
Der `ADI_SPLIT_6` ist ein einfacher aber effektiver Baustein zur Vervielfachung unidirektionaler ADI-Adapter-Signale. Durch die generische Auslegung und die geringe Komplexität eignet er sich hervorragend für den modularen Aufbau von Steuerungsanwendungen im Eclipse 4diac Umfeld. Seine Einsatzmöglichkeiten reichen von der reinen Signalverteilung bis hin zu Test- und Redundanzszenarien.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ADI_SPLIT_6_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ADI_SPLIT_6`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock ADI_SPLIT_6_UNGATED dient dazu, einen einzelnen ADI-Adapter-Eingang (unidirektional) auf sechs identische ADI-Adapter-Ausgänge zu verteilen. Er ist als generischer Baustein für das

Eclipse 4diac Framework konzipiert und wird typischerweise eingesetzt, wenn ein ADI-Signal an mehrere nachfolgende Komponenten weitergeleitet werden muss.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge
Keine Daten-Eingänge vorhanden.

Daten-Ausgänge
Keine Daten-Ausgänge vorhanden.

Adapter
Der Baustein kommuniziert ausschließlich über Adapter-Schnittstellen:
Socket (Eingang):

- `IN`: Typ `adapter::types::unidirectional::ADI` – nimmt ein unidirektionales ADI-Signal entgegen.

Plugs (Ausgänge):

- `OUT1` bis `OUT6`: jeweils Typ `adapter::types::unidirectional::ADI` – geben das eingehende Signal identisch auf sechs parallele Ausgänge weiter.

Funktionsweise
Der FB führt eine reine 1:6-Verteilung des ADI-Adapter-Signals durch. Alle Daten und Ereignisse, die am Socket `IN` ankommen, werden unverändert und zeitgleich an alle sechs Plugs `OUT1`…`OUT6` weitergegeben. Es findet keine Pufferung, Filterung oder Verzögerung statt. Der Baustein arbeitet somit als passiver Splitter auf Adapterebene.

Technische Besonderheiten

- **Generische Implementierung:** Der FB ist als generischer Baustein (`eclipse4diac::core::GenericClassName = 'GEN_ADI_SPLIT'`) realisiert und kann in verschiedenen Projekten wiederverwendet werden.
- **Unidirektionale Adapter:** Die verwendeten ADI-Adapter sind unidirektional ausgelegt; eine Rückkopplung über die Ausgänge ist nicht vorgesehen.
- **Keine eigenen Zustände:** Da der FB keine Ereignis- oder Datenverarbeitung durchführt, besitzt er keinen internen Zustandsautomaten.
- **Performance:** Durch die direkte Weiterleitung ohne Logik oder Speicher entsteht praktisch keine Latenz.

Zustandsübersicht
Der Baustein verfügt über keinen Zustandsautomaten oder explizite Betriebszustände. Er ist stets aktiv und gibt das Eingangssignal sofort an alle Ausgänge weiter.

Anwendungsszenarien

- **Signalverteilung in Steuerungssystemen:** Ein von einem Sensor oder einer zentralen Logik bereitgestelltes ADI-Signal soll parallel an mehrere Aktoren, Anzeigen oder nachgelagerte Funktionen gesendet werden.
- **Testumgebungen:** Bei der Simulation oder beim Debugging kann ein einzelnes Signal auf mehrere Empfänger aufgeteilt werden, um verschiedene Komponenten gleichzeitig zu testen.
- **Redundanzabbildung:** Falls ein Signal mehrfach benötigt wird (z. B. für Überwachungs- und Steuerungspfade), kann dieser Splitter eine saubere Entkopplung der Ausgänge ermöglichen.

Vergleich mit ähnlichen Bausteinen

- **SPLIT für Standard-Datentypen:** Viele Frameworks bieten Splitter für einfache Datentypen (z. B. `SPLIT_INT`). Der `ADI_SPLIT_6_UNGATED` ist speziell für den Adapter-Typ `ADI` ausgelegt und arbeitet auf einer höheren Abstraktionsebene (Adapter statt einzelner Datenpunkte).
- **ADI_MERGE / ADI_COMBINE:** Während dieser Baustein ein Signal verteilt, fassen andere Bausteine mehrere Adapter-Eingänge zu einem Ausgang zusammen.
- **Weitere Splitter-Varianten:** Es existieren möglicherweise Splitter für andere Adapter-Typen (z. B. bidirektionale ADI). Der vorliegende FB beschränkt sich auf unidirektionale Signale.
- **`ADI_SPLIT_6`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

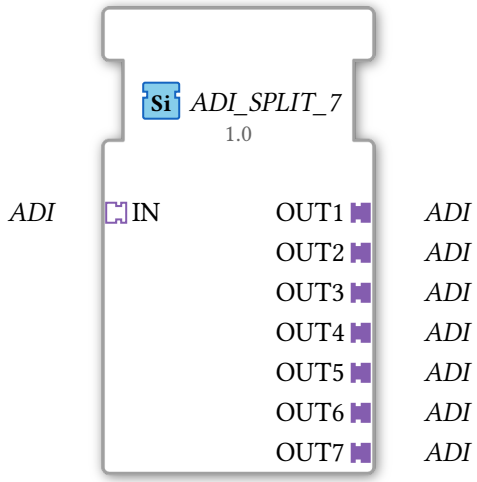
Fazit

Der `ADI_SPLIT_6_UNGATED` ist ein einfacher aber effektiver Baustein zur Vervielfachung unidirektionaler ADI-Adapter-Signale. Durch die generische Auslegung und die geringe Komplexität eignet er sich hervorragend für den modularen Aufbau von Steuerungsanwendungen im Eclipse 4diac Umfeld. Seine Einsatzmöglichkeiten reichen von der reinen Signalverteilung bis hin zu Test- und Redundanzszenarien.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ADI_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ADI_SPLIT_7** dient der Aufteilung eines eingehenden **ADI**-Adapter-Signals auf sieben identische Ausgänge. Er ist als generischer Baustein (Generic FB) implementiert und ermöglicht die unidirektionale Verteilung von Daten über einen Adapter-Socket an sieben Adapter-Plugs.

Schnittstellenstruktur

Ereignis-Eingänge

Keine – der Baustein verfügt über keine Ereignis-Eingänge. Die Datenweitergabe erfolgt ausschließlich über die Adapter-Schnittstelle.

Ereignis-Ausgänge

Keine – der Baustein verfügt über keine Ereignis-Ausgänge. Die Ausgangsadapter werden direkt mit den empfangenen Daten versorgt.

Daten-Eingänge

Keine – die Daten werden nicht über separate Daten-Eingänge bereitgestellt, sondern über den Adapter-Socket IN.

Daten-Ausgänge

Keine – die Daten werden nicht über separate Daten-Ausgänge ausgegeben, sondern über die Adapter-Plugs OUT1 ... OUT7.

Adapter

Name	Richtung	Typ	Beschreibung
IN	Socket	adi (unidirektional)	Eingangsadapter, der die zu verteilenden Daten empfängt.
OUT1	Plug	adi (unidirektional)	Erster Ausgangsadapter (identisch zu IN).
OUT2	Plug	adi (unidirektional)	Zweiter Ausgangsadapter (identisch zu IN).
OUT3	Plug	adi (unidirektional)	Dritter Ausgangsadapter (identisch zu IN).
OUT4	Plug	adi (unidirektional)	Vierter Ausgangsadapter (identisch zu IN).
OUT5	Plug	adi (unidirektional)	Fünfter Ausgangsadapter (identisch zu IN).
OUT6	Plug	adi (unidirektional)	Sechster Ausgangsadapter (identisch zu IN).
OUT7	Plug	adi (unidirektional)	Siebter Ausgangsadapter (identisch zu IN).

Funktionsweise

Der Baustein fungiert als **1-zu-7-Splitter** für ADI-Adapter-Signale. Wird das Signal am Eingangsadapter **IN** gesetzt, wird es ohne weitere Verarbeitung oder Verzögerung auf alle sieben Ausgangsadapter **OUT1** bis **OUT7** kopiert. Der Baustein besitzt keine interne Logik oder Zustandsmaschine – er leitet die Daten rein passiv weiter.

Technische Besonderheiten

- **Generischer Baustein** – der Typ wird als `GEN_ADI_SPLIT` instanziiert und kann in verschiedenen Anwendungen wiederverwendet werden.
- **Unidirektionale Adapter** – alle Adapter sind vom Typ `adi` (unidirektional). Ein Rückfluss von Daten von den Ausgängen zum Eingang ist nicht vorgesehen.
- **Keine Ereignissteuerung** – die Datenweitergabe erfolgt rein datengetrieben ohne Ereignis-Ein-/Ausgänge.
- **1:1-Kopie** – die Daten am Eingang werden unverändert auf alle Ausgänge übertragen.

Zustandsübersicht

Der Baustein besitzt **keine Zustandsmaschine** (ECC). Er ist stateless, d.h. er reagiert nicht auf Ereignisse und hat keine internen Zustände. Die Ausgangsdaten folgen direkt den Eingangsdaten.

Anwendungsszenarien

- **Signalverteilung** – ein einzelnes ADI-Signal (z.B. Sensorwert, Steuersignal) muss an mehrere Verbraucher (z.B. Steuerungsblöcke, Visualisierungen) verteilt werden.
- **Redundanz** – ein Signal wird für redundante Verarbeitungspfade benötigt.
- **Test- und Simulationsumgebungen** – ein eingehendes Signal wird für verschiedene Testinstanzen aufgeteilt.

Vergleich mit ähnlichen Bausteinen

Baustein	Anzahl Ausgänge	Eigenschaften

ADI_SPLIT_2	2	Verteilt 1 ADI auf 2 Ausgänge
ADI_SPLIT_4	4	Verteilt 1 ADI auf 4 Ausgänge
ADI_SPLIT_7	7	Verteilt 1 ADI auf 7 Ausgänge
ADI_MERGE	1 (Eingang: mehrere)	Fasst mehrere ADI-Eingänge zu einem Ausgang zusammen

Gegenüber einer manuellen Parallelschaltung mehrerer ADI_Split-Bausteine reduziert der `ADI_SPLIT_7` den Konfigurationsaufwand und verbessert die Übersicht im Netzwerk.

- **`ADI_SPLIT_7_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

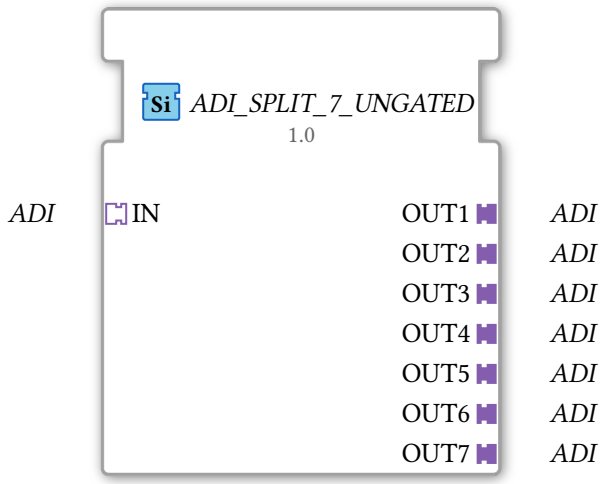
Der `ADI_SPLIT_7` ist ein einfacher, aber leistungsfähiger Baustein zur Verteilung eines unidirektionalen ADI-Signals auf sieben identische Ausgänge. Dank seiner generischen Implementierung und dem Verzicht auf Ereignissteuerung eignet er sich ideal für Reinverteilungen in Automatisierungs- und Steuerungsumgebungen, die auf dem ADI-Adapterprotokoll basieren.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Gesamtwiderstand in Reihen- & Parallelschaltung auf ms-muc-docs.de

ADI_SPLIT_7_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ADI_SPLIT_7`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ADI_SPLIT_7_UNGATED** dient der Aufteilung eines eingehenden **ADI**-Adapter-Signals auf sieben identische Ausgänge. Er ist als generischer Baustein (Generic FB) implementiert und ermöglicht die unidirektionale Verteilung von Daten über einen Adapter-Socket an sieben Adapter-Plugs.

Schnittstellenstruktur

Ereignis-Eingänge

Keine – der Baustein verfügt über keine Ereignis-Eingänge. Die Datenweitergabe erfolgt ausschließlich über die Adapter-Schnittstelle.

Ereignis-Ausgänge

Keine – der Baustein verfügt über keine Ereignis-Ausgänge. Die Ausgangsadapter werden direkt mit den empfangenen Daten versorgt.

Daten-Eingänge

Keine – die Daten werden nicht über separate Daten-Eingänge bereitgestellt, sondern über den Adapter-Socket IN.

Daten-Ausgänge

Keine – die Daten werden nicht über separate Daten-Ausgänge ausgegeben, sondern über die Adapter-Plugs OUT1 … OUT7.

Adapter

Name	Richtung	Typ	Beschreibung
IN	Socket	adi (unidirektional)	Eingangsadapter, der die zu verteilenden Daten empfängt.
OUT1	Plug	adi (unidirektional)	Erster Ausgangsadapter (identisch zu IN).
OUT2	Plug	adi (unidirektional)	Zweiter Ausgangsadapter (identisch zu IN).
OUT3	Plug	adi (unidirektional)	Dritter Ausgangsadapter (identisch zu IN).
OUT4	Plug	adi (unidirektional)	Vierter Ausgangsadapter (identisch zu IN).
OUT5	Plug	adi (unidirektional)	Fünfter Ausgangsadapter (identisch zu IN).
OUT6	Plug	adi (unidirektional)	Sechster Ausgangsadapter (identisch zu IN).
OUT7	Plug	adi (unidirektional)	Siebter Ausgangsadapter (identisch zu IN).

Funktionsweise

Der Baustein fungiert als **1-zu-7-Splitter** für ADI-Adapter-Signale. Wird das Signal am Eingangsadapter **IN** gesetzt, wird es ohne weitere Verarbeitung oder Verzögerung auf alle sieben Ausgangsadapter **OUT1** bis **OUT7** kopiert. Der Baustein besitzt keine interne Logik oder Zustandsmaschine – er leitet die Daten rein passiv weiter.

Technische Besonderheiten

- **Generischer Baustein** – der Typ wird als `GEN_ADI_SPLIT` instanziiert und kann in verschiedenen Anwendungen wiederverwendet werden.
- **Unidirektionale Adapter** – alle Adapter sind vom Typ `adi` (unidirektional). Ein Rückfluss von Daten von den Ausgängen zum Eingang ist nicht vorgesehen.
- **Keine Ereignissteuerung** – die Datenweitergabe erfolgt rein datengetrieben ohne Ereignis-Ein-/Ausgänge.
- **1:1-Kopie** – die Daten am Eingang werden unverändert auf alle Ausgänge übertragen.

Zustandsübersicht

Der Baustein besitzt **keine Zustandsmaschine** (ECC). Er ist stateless, d.h. er reagiert nicht auf Ereignisse und hat keine internen Zustände. Die Ausgangsdaten folgen direkt den Eingangsdaten.

Anwendungsszenarien

- **Signalverteilung** – ein einzelnes ADI-Signal (z.B. Sensorwert, Steuersignal) muss an mehrere Verbraucher (z.B. Steuerungsblöcke, Visualisierungen) verteilt werden.
- **Redundanz** – ein Signal wird für redundante Verarbeitungspfade benötigt.
- **Test- und Simulationsumgebungen** – ein eingehendes Signal wird für verschiedene Testinstanzen aufgeteilt.

Vergleich mit ähnlichen Bausteinen

Baustein	Anzahl Ausgänge	Eigenschaften
ADI_SPLIT_2	2	Verteilt 1 ADI auf 2 Ausgänge
ADI_SPLIT_4	4	Verteilt 1 ADI auf 4 Ausgänge
ADI_SPLIT_7_UNGATED	7	Verteilt 1 ADI auf 7 Ausgänge
ADI_MERGE	1 (Eingang: mehrere)	Fasst mehrere ADI-Eingänge zu einem Ausgang zusammen

Gegenüber einer manuellen Parallelschaltung mehrerer ADI_Split-Bausteine reduziert der `ADI_SPLIT_7_UNGATED` den Konfigurationsaufwand und verbessert die Übersicht im Netzwerk.

- **`ADI_SPLIT_7_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- **`ADI_SPLIT_7`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

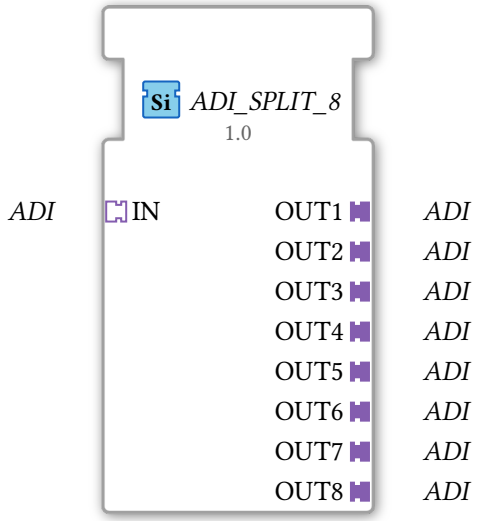
Fazit

Der `ADI_SPLIT_7_UNGATED` ist ein einfacher, aber leistungsfähiger Baustein zur Verteilung eines unidirektionalen ADI-Signals auf sieben identische Ausgänge. Dank seiner generischen Implementierung und dem Verzicht auf Ereignissteuerung eignet er sich ideal für Reinverteilungen in Automatisierungs- und Steuerungsumgebungen, die auf dem ADI-Adapterprotokoll basieren.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Gesamtwiderstand in Reihen- & Parallelschaltung auf ms-muc-docs.de

ADI_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ADI_SPLIT_8` dient dazu, einen einzelnen unidirektionalen ADI-Adapter (Application Data Interface) auf acht baugleiche Ausgangs-Adapter aufzuteilen. Er ist als generischer Baustein realisiert, der durch die Angabe eines `GenericClassName`-Attributs an verschiedene ADI-Kontexte angepasst werden kann. Der FB vereinfacht die Signaldistribution in Steuerungsanwendungen, indem er einen eingehenden Adapter-Pfad ohne Datenverlust auf mehrere Empfänger verteilt.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine Ereignis-Eingänge vorhanden. Die Datenweitergabe erfolgt kontinuierlich und ereignislos durch direkte Adapterverbindung.

Ereignis-Ausgänge

Es sind keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Es sind keine Daten-Eingänge im herkömmlichen Sinne definiert. Die Informationsübertragung erfolgt ausschließlich über die Adapterschnittstellen.

Daten-Ausgänge

Es sind keine Daten-Ausgänge vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	IN	ADI (unidirektional)	Eingangsschnittstelle, die auf acht Ausgänge verteilt wird.
Plug	OUT1	ADI (unidirektional)	Erster ausgehender Adapter (identisch zu IN).
Plug	OUT2	ADI (unidirektional)	Zweiter ausgehender Adapter.
Plug	OUT3	ADI (unidirektional)	Dritter ausgehender Adapter.
Plug	OUT4	ADI (unidirektional)	Vierter ausgehender Adapter.
Plug	OUT5	ADI (unidirektional)	Fünfter ausgehender Adapter.
Plug	OUT6	ADI (unidirektional)	Sechster ausgehender Adapter.
Plug	OUT7	ADI (unidirektional)	Siebter ausgehender Adapter.
Plug	OUT8	ADI (unidirektional)	Achter ausgehender Adapter.

Funktionsweise

Der FB leitet den über den Socket `IN` ankommenden ADI-Adapter unverändert auf alle acht `OUT`-Plugs weiter. Jeder Ausgang stellt eine eigenständige, aber inhaltlich identische Kopie des eingehenden Adapters dar. Die Verteilung erfolgt direkt – es findet keine Pufferung, Verzögerung oder Transformation statt. Änderungen am `IN`-Adapter werden sofort an alle `OUT`-Adapter propagiert. Der Baustein arbeitet rein kombinatorisch und benötigt keine Taktung oder Initialisierung.

Technische Besonderheiten

- **Generischer Typ**: Durch das Attribut `eclipse4diac::core::GenericClassName` kann der FB auf verschiedene konkrete ADI-Implementierungen parametrisiert werden. Standardmäßig ist der Klassenname auf `'GEN_ADI_SPLIT'` gesetzt.
- **Keine Zustandshaltung**: Der FB besitzt kein internes Gedächtnis und keine Zustandsautomaten; er agiert als einfacher Signalverteiler.
- **Unidirektionalität**: Alle Adapter sind unidirektional ausgelegt – die Daten fließen nur vom Socket zu den Plugs. Eine Rückkopplung ist nicht vorgesehen.
- **Keine Ereignissteuerung**: Die Verteilung geschieht ohne Ereignisse, was die Laufzeitdeterministik erhöht.

Zustandsübersicht

Der `ADI_SPLIT_8` FB besitzt keine internen Zustände oder einen ECC (Execution Control Chart). Er verhält sich statisch und führt zu jeder Zeit die gleiche Funktion aus. Eine Zustandsübersicht entfällt daher.

Anwendungsszenarien

- **Signalbroadcasting**: Verteilen eines Messwert- oder Steuerungssignals (z. B. Temperatur, Drehzahl) an mehrere parallele Regelkreise.
- **Adapter-Multiplexing**: Bereitstellung eines zentralen ADI-Pfades für mehrere Subsysteme, die denselben Datenstrom benötigen.
- **Test- und Simulationsumgebungen**: Einspeisen eines simulierten Signals auf mehrere Empfänger-Blöcke gleichzeitig.
- **Modulare Automatisierungsarchitekturen**: Aufbau redundanter oder paralleler Verarbeitungspfade durch einfache Duplikation von Adapterverbindungen.

Vergleich mit ähnlichen Bausteinen

- **SPLIT (Daten-Ereignis-Variante)**: Klassische Split-Bausteine (z. B. `E_SPLIT`) verteilen Ereignisse oder Datenwerte, nicht jedoch ganze Adapter-Pfade. `ADI_SPLIT_8` arbeitet auf einer höheren Abstraktionsebene (Adapterebene).
- **FORK-FBs**: Einige IEC 61499 Implementierungen bieten Fork-Bausteine für Adapter, die jedoch oft ereignisgesteuert sind. Dieser FB hier zeichnet sich durch den Verzicht auf Ereignisse und die feste Anzahl von acht Ausgängen aus.
- **Manuelle Verdrahtung**: Ohne diesen FB müsste jede Zielverbindung einzeln über Koppeladapter oder manuelle Verzweigungen realisiert werden – aufwändiger und fehleranfälliger.
- **`ADI_SPLIT_8_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

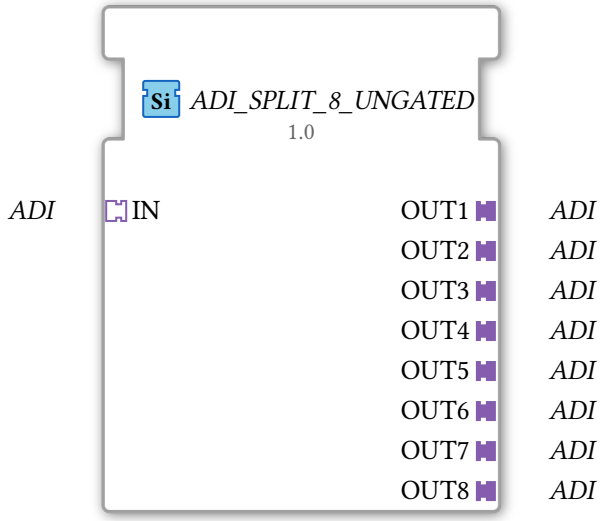
`ADI_SPLIT_8` ist ein leistungsfähiger, generischer Split-Baustein für unidirektionale ADI-Adapter in IEC 61499 Anwendungen. Er reduziert den Verdrahtungsaufwand erheblich, erhöht die Übersichtlichkeit und ermöglicht die einfache Vervielfachung eines Adapterpfades. Durch die generische Parametrierung und den ereignislosen Betrieb eignet er sich besonders für datengetriebene Automatisierungssysteme, die eine zuverlässige und latenzarme Signalverteilung erfordern.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ADI_SPLIT_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ADI_SPLIT_8`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ADI_SPLIT_8_UNGATED` dient dazu, einen einzelnen unidirektionalen ADI-Adapter (Application Data Interface) auf acht baugleiche Ausgangs-Adapter aufzuteilen. Er ist als generischer Baustein realisiert, der durch die Angabe eines `GenericClassName`-Attributs an verschiedene ADI-Kontexte angepasst werden kann. Der FB vereinfacht die Signaldistribution in Steuerungsanwendungen, indem er einen eingehenden Adapter-Pfad ohne Datenverlust auf mehrere Empfänger verteilt.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine Ereignis-Eingänge vorhanden. Die Datenweitergabe erfolgt kontinuierlich und ereignislos durch direkte Adapterverbindung.

Ereignis-Ausgänge

Es sind keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Es sind keine Daten-Eingänge im herkömmlichen Sinne definiert. Die Informationsübertragung erfolgt ausschließlich über die Adapterschnittstellen.

Daten-Ausgänge

Es sind keine Daten-Ausgänge vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	IN	ADI (unidirektional)	Eingangsschnittstelle, die auf acht Ausgänge verteilt wird.
Plug	OUT1	ADI (unidirektional)	Erster ausgehender Adapter (identisch zu IN).
Plug	OUT2	ADI (unidirektional)	Zweiter ausgehender Adapter.
Plug	OUT3	ADI (unidirektional)	Dritter ausgehender Adapter.
Plug	OUT4	ADI (unidirektional)	Vierter ausgehender Adapter.
Plug	OUT5	ADI (unidirektional)	Fünfter ausgehender Adapter.
Plug	OUT6	ADI (unidirektional)	Sechster ausgehender Adapter.

Plug	OUT7	ADI (unidirektional)	Siebter ausgehender Adapter.
Plug	OUT8	ADI (unidirektional)	Achter ausgehender Adapter.

Funktionsweise

Der FB leitet den über den Socket `IN` ankommenden ADI-Adapter unverändert auf alle acht `OUT`-Plugs weiter. Jeder Ausgang stellt eine eigenständige, aber inhaltlich identische Kopie des eingehenden Adapters dar. Die Verteilung erfolgt direkt – es findet keine Pufferung, Verzögerung oder Transformation statt. Änderungen am `IN`-Adapter werden sofort an alle `OUT`-Adapter propagiert. Der Baustein arbeitet rein kombinatorisch und benötigt keine Taktung oder Initialisierung.

Technische Besonderheiten

- **Generischer Typ**: Durch das Attribut `eclipse4diac::core::GenericClassName` kann der FB auf verschiedene konkrete ADI-Implementierungen parametrisiert werden. Standardmäßig ist der Klassenname auf `'GEN_ADI_SPLIT'` gesetzt.
- **Keine Zustandshaltung**: Der FB besitzt kein internes Gedächtnis und keine Zustandsautomaten; er agiert als einfacher Signalverteiler.
- **Unidirektionalität**: Alle Adapter sind unidirektional ausgelegt – die Daten fließen nur vom Socket zu den Plugs. Eine Rückkopplung ist nicht vorgesehen.
- **Keine Ereignissteuerung**: Die Verteilung geschieht ohne Ereignisse, was die Laufzeitdeterministik erhöht.

Zustandsübersicht

Der `ADI_SPLIT_8_UNGATED` FB besitzt keine internen Zustände oder einen ECC (Execution Control Chart). Er verhält sich statisch und führt zu jeder Zeit die gleiche Funktion aus. Eine Zustandsübersicht entfällt daher.

Anwendungsszenarien

- **Signalbroadcasting**: Verteilen eines Messwert- oder Steuerungssignals (z. B. Temperatur, Drehzahl) an mehrere parallele Regelkreise.
- **Adapter-Multiplexing**: Bereitstellung eines zentralen ADI-Pfades für mehrere Subsysteme, die denselben Datenstrom benötigen.
- **Test- und Simulationsumgebungen**: Einspeisen eines simulierten Signals auf mehrere Empfänger-Blöcke gleichzeitig.
- **Modulare Automatisierungsarchitekturen**: Aufbau redundanter oder paralleler Verarbeitungspfade durch einfache Duplikation von Adapterverbindungen.

Vergleich mit ähnlichen Bausteinen

- **SPLIT (Daten-Ereignis-Variante)**: Klassische Split-Bausteine (z. B. `E_SPLIT`) verteilen Ereignisse oder Datenwerte, nicht jedoch ganze Adapter-Pfade. `ADI_SPLIT_8_UNGATED` arbeitet auf einer höheren Abstraktionsebene (Adapterebene).
- **FORK-FBs**: Einige IEC 61499 Implementierungen bieten Fork-Bausteine für Adapter, die jedoch oft ereignisgesteuert sind. Dieser FB hier zeichnet sich durch den Verzicht auf Ereignisse und die feste Anzahl von acht Ausgängen aus.
- **Manuelle Verdrahtung**: Ohne diesen FB müsste jede Zielverbindung einzeln über Koppeladapter oder manuelle Verzweigungen realisiert werden – aufwändiger und fehleranfälliger.
- `ADI_SPLIT_8`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

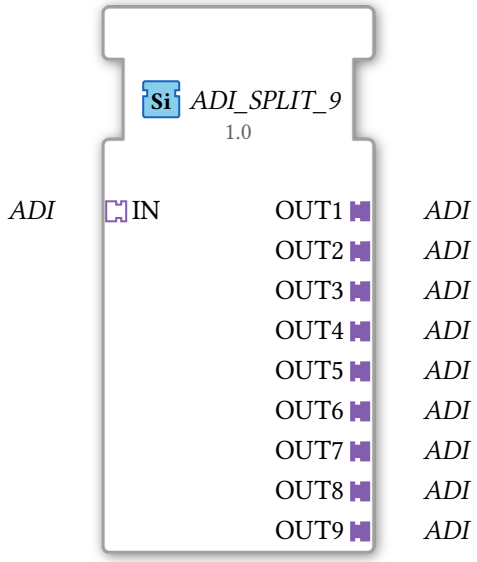
`ADI_SPLIT_8_UNGATED` ist ein leistungsfähiger, generischer Split-Baustein für unidirektionale ADI-Adapter in IEC 61499 Anwendungen. Er reduziert den Verdrahtungsaufwand erheblich, erhöht die Übersichtlichkeit und ermöglicht die einfache Vervielfachung eines Adapterpfades. Durch die generische

Parametrierung und den ereignislosen Betrieb eignet er sich besonders für datengetriebene Automatisierungssysteme, die eine zuverlässige und latenzarme Signalverteilung erfordern.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ADI_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ADI_SPLIT_9` ist ein generischer Baustein, der ein eingehendes ADI-Signal (Adapter) auf neun identische Ausgangssignale aufteilt. Er dient als reiner Splitter auf Adapterebene.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

- **IN** (Socket): Adapter vom Typ `adapter::types::unidirectional::ADI` – Eingangssignal.
- **OUT1** bis **OUT9** (Plugs): Neun Adapter desselben Typs – Ausgänge, die das Eingangssignal unverändert weitergeben.

Funktionsweise

Der FB leitet das am Socket `IN` anliegende ADI-Signal an alle neun Plugs `OUT1` bis `OUT9` weiter. Es findet keine Signalverarbeitung oder Umwandlung statt; die Verteilung erfolgt rein strukturell über die Adapterverbindungen. Aufgrund des unidirektionalen Adaptertyps ist die Datenflussrichtung vom Socket zu den Plugs festgelegt.

Technische Besonderheiten

- Der Baustein ist als generischer Funktionsblock definiert (Attribut `eclipse4diac::core::GenericClassName = 'GEN_ADI_SPLIT'`) und kann dadurch in unterschiedlichen Projekten wiederverwendet werden.
- Er besitzt weder Ereignisse noch Daten-Ein-/Ausgänge; die gesamte Funktionalität liegt in der Adaptertopologie.
- Das Typ-Hash-Attribut ist leer, was auf eine einfache, nicht versionierte Ausführung hinweist.

Zustandsübersicht

Der Funktionsblock besitzt keine Zustandsautomaten oder Algorithmen. Er ist rein passiv und reagiert nicht auf Ereignisse.

Anwendungsszenarien

- Verteilen eines analogen oder digitalen Sensorsignals (als ADI) an mehrere nachgeschaltete Bausteine.
- Aufteilen eines Steuersignals in einer Automatisierungsanlage, um identische Aktoren parallel anzusteuern.
- Erzeugen von Signalkopien für Überwachungs-, Test- oder Diagnosezwecke.

Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu daten- oder ereignisbasierten Splittern arbeitet `ADI_SPLIT_9` auf der höheren Adapterebene, was eine saubere architekturelle Trennung ermöglicht.
- Ähnliche Bausteine existieren mit variabler Ausgangsanzahl (z. B. `ADI_SPLIT_M`), während dieser FB eine feste Anzahl von neun Ausgängen bietet.
- **`ADI_SPLIT_9_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

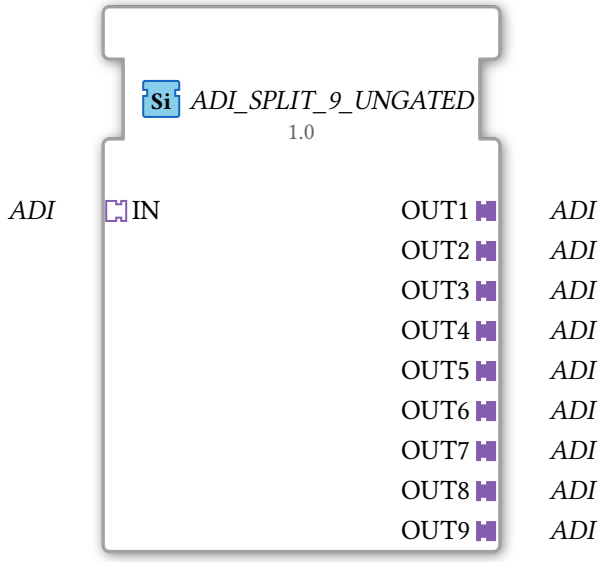
`ADI_SPLIT_9` ist ein einfacher, generischer Funktionsblock zur 1-zu-9-Verteilung eines unidirektionalen ADI-Signals. Er eignet sich für Anwendungen, die ein mehrfaches, identisches Weiterleiten von Adaptersignalen ohne zusätzliche Logik erfordern.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ADI_SPLIT_9_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ADI_SPLIT_9`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ADI_SPLIT_9_UNGATED` ist ein generischer Baustein, der ein eingehendes ADI-Signal (Adapter) auf neun identische Ausgangssignale aufteilt. Er dient als reiner Splitter auf Adapterebene.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

- **IN** (Socket): Adapter vom Typ `adapter::types::unidirectional::ADI` – Eingangssignal.
- **OUT1** bis **OUT9** (Plugs): Neun Adapter desselben Typs – Ausgänge, die das Eingangssignal unverändert weitergeben.

Funktionsweise

Der FB leitet das am Socket `IN` anliegende ADI-Signal an alle neun Plugs `OUT1` bis `OUT9` weiter. Es findet keine Signalverarbeitung oder Umwandlung statt; die Verteilung erfolgt rein strukturell über die Adapterverbindungen. Aufgrund des unidirektionalen Adaptertyps ist die Datenflussrichtung vom Socket zu den Plugs festgelegt.

Technische Besonderheiten

- Der Baustein ist als generischer Funktionsblock definiert (Attribut `eclipse4diac::core::GenericClassName = 'GEN_ADI_SPLIT'`) und kann dadurch in unterschiedlichen Projekten wiederverwendet werden.
- Er besitzt weder Ereignisse noch Daten-Ein-/Ausgänge; die gesamte Funktionalität liegt in der Adaptertopologie.
- Das Typ-Hash-Attribut ist leer, was auf eine einfache, nicht versionierte Ausführung hinweist.

Zustandsübersicht

Der Funktionsblock besitzt keine Zustandsautomaten oder Algorithmen. Er ist rein passiv und reagiert nicht auf Ereignisse.

Anwendungsszenarien

- Verteilen eines analogen oder digitalen Sensorsignals (als ADI) an mehrere nachgeschaltete Bausteine.
- Aufteilen eines Steuersignals in einer Automatisierungsanlage, um identische Aktoren parallel anzusteuern.
- Erzeugen von Signalkopien für Überwachungs-, Test- oder Diagnosezwecke.

Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu daten- oder ereignisbasierten Splittern arbeitet `ADI_SPLIT_9_UNGATED` auf der höheren Adapterebene, was eine saubere architekturelle Trennung ermöglicht.
- Ähnliche Bausteine existieren mit variabler Ausgangsanzahl (z. B. `ADI_SPLIT_M`), während dieser FB eine feste Anzahl von neun Ausgängen bietet.
- **`ADI_SPLIT_9`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

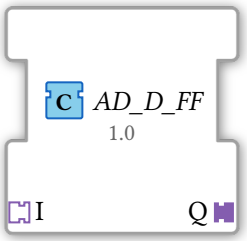
`ADI_SPLIT_9_UNGATED` ist ein einfacher, generischer Funktionsblock zur 1-zu-9-Verteilung eines unidirektionalen ADI-Signals. Er eignet sich für Anwendungen, die ein mehrfaches, identisches Weiterleiten von Adaptersignalen ohne zusätzliche Logik erfordern.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

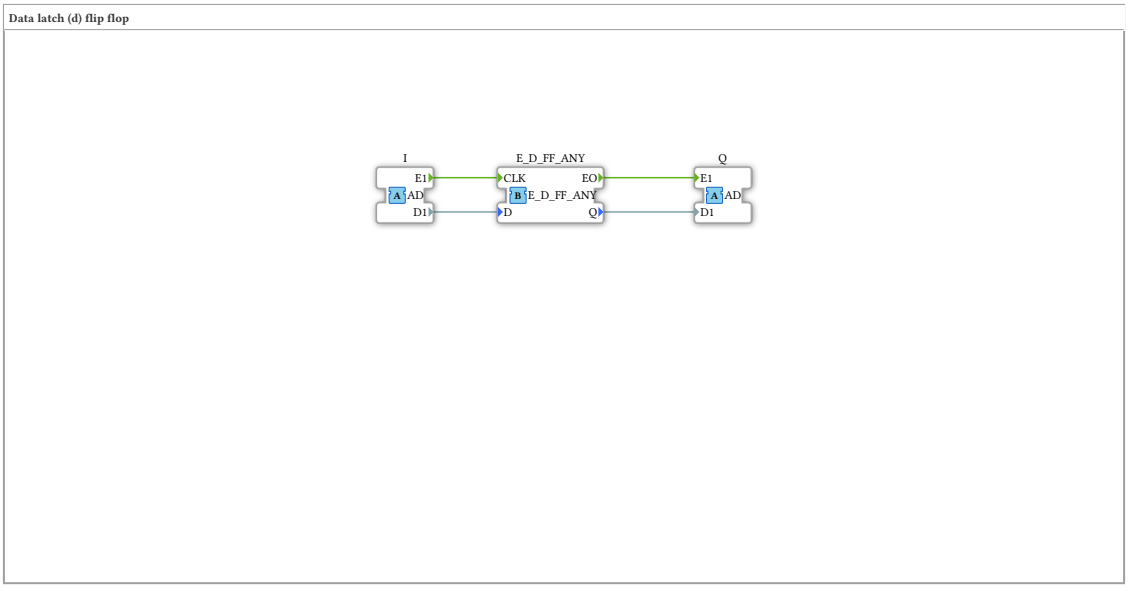
DWORD

AD_D_FF



Value to latch – *AD*

AD – Latched value



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AD_D_FF** realisiert ein **Data-Latch (D-Flipflop)**. Er übernimmt einen Datenwert bei einem eingehenden Ereignis und hält diesen Wert bis zur nächsten Übernahme bereit. Die Schnittstelle ist ausschließlich über standardisierte **Adapter** definiert, was eine flexible Verkabelung in komplexen Automatisierungssystemen ermöglicht.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge. Der FB verwendet ausschließlich den **Eingangsadapter `I`**, der über seinen internen Ereigniskanal das Taktsignal liefert.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge. Der FB verwendet ausschließlich den **Ausgangsadapter `Q`**, der über seinen internen Ereigniskanal signalisiert, dass ein neuer Datenwert bereitsteht.

Daten-Eingänge

Keine direkten Daten-Eingänge. Der zu latchende Wert wird über den **Datenkanal des Eingangsadapters `I`** bereitgestellt.

Daten-Ausgänge

Keine direkten Daten-Ausgänge. Der latched Wert wird über den **Datenkanal des Ausgangsadapters `Q`** zur Verfügung gestellt.

Adapter

Adapter	Richtung	Typ	Beschreibung
`I` (Socket)	Eingang	`adapter::types::unidirectional::AD`	Liefert das Taktsignal (E1) und den zu speichernden Datenwert (D1).
`Q` (Plug)	Ausgang	`adapter::types::unidirectional::AD`	Gibt nach dem Taktereignis den gespeicherten Datenwert aus (E1 als Bestätigung, D1 als Daten).

Die Adapter vom Typ `unidirectional::AD` definieren je einen Ereigniskanal (`E1`) und einen Datenkanal (`D1`). Die genauen Datentypen sind durch die Adapterdefinition festgelegt (üblicherweise `ANY`).

Funktionsweise

Der FB verwendet intern den elementaren Baustein `E_D_FF_ANY` (aus der IEC-61499-Standardbibliothek). Dessen Verhalten:

1. **Bei jedem Ereignis am Takteingang (CLK)** wird der aktuelle Datenwert am Eingang `D` an den Ausgang `Q` übernommen. Die Übernahme erfolgt flankengetriggert (steigende Flanke des Ereignisses).
2. Der Ausgang `Q` bleibt so lange unverändert, bis ein neues Taktereignis eintrifft.

Die Verschaltung im FBNetzwerk:

- Das Ereignis `I.E1` wird mit `CLK` verbunden.
- Das Datum `I.D1` wird mit `D` verbunden.
- Der Ereignisausgang `EO` des internen Bausteins wird an `Q.E1` weitergeleitet.
- Der Datenausgang `Q` des internen Bausteins wird an `Q.D1` weitergeleitet.

Somit wird bei jedem Ereignis am Eingangsadapter der anliegende Datenwert übernommen und sofort über den Ausgangsadapter ausgegeben. Der Wert bleibt bis zum nächsten Ereignis erhalten.

Technische Besonderheiten

- **Reine Adapter-Schnittstelle:** Der FB besitzt keine klassischen Event-/Data-Ports, sondern kommuniziert ausschließlich über Adapter. Dies ermöglicht die Verwendung in Systemen mit standardisierten Verbindungen (z. B. über bestimmte Feldbus-Adapterprofile).
- **Typtransparenz:** Da die Adapter generisch (`ANY`) sind, kann der FB mit verschiedenen Datentypen arbeiten – vorausgesetzt, sowohl der Eingangs- als auch der Ausgangsadapter verwenden denselben Datentyp.
- **Kein interner Zustandsautomat:** Der FB realisiert das Flipflop-Verhalten durch die reine Daten-/Ereignisverdrahtung des internen `E_D_FF_ANY`. Ein eigener Zustandsautomat ist nicht vorhanden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten. Der interne Baustein `E_D_FF_ANY` hält intern den gespeicherten Wert (Zustand). Der FB ist jedoch aus Sicht der Ereignisverarbeitung zustandslos: Jedes Ereignis am Eingang löst sofort eine Ausgabe aus.

Anwendungsszenarien

- **Puffern von Signalen:** Ein Datenwert soll bei einem bestimmten Ereignis „eingefroren“ und für nachfolgende Verarbeitungsschritte bereitgestellt werden.
- **Synchronisation asynchroner Daten:** Werte aus einem unsynchronisierten Kontext werden mittels eines Taktereignisses in den getakteten Bereich übernommen.
- **Standardisierte Datenkopplung:** In Architekturen, die auf Adapter-Schnittstellen setzen (z. B. in der Landmaschinentechnik), kann der FB als universelle Latch-Komponente eingesetzt werden.

Vergleich mit ähnlichen Bausteinen

- **RS-Flipflop (AD_RS_FF):** Speichert zwei Zustände (Set/Reset) und benötigt zwei Ereignis-/Datenkanäle. Der AD_D_FF arbeitet mit nur einem Taktsignal und einem Datum – einfacher und deterministischer.
- **E_D_FF_ANY (direkt):** Bietet die gleiche Funktionalität, jedoch mit klassischen Event-/Data-Ports anstelle von Adaptern. Der AD_D_FF kapselt diese Logik in eine Adapter-Schnittstelle, was die Wiederverwendung in adapterbasierten Umgebungen erleichtert.
- **Toggle-FF (AD_T_FF)** oder **JK-FF:** Komplexere Flipflops mit mehreren Eingängen; der AD_D_FF ist die einfachste Form eines datengesteuerten Speichers.

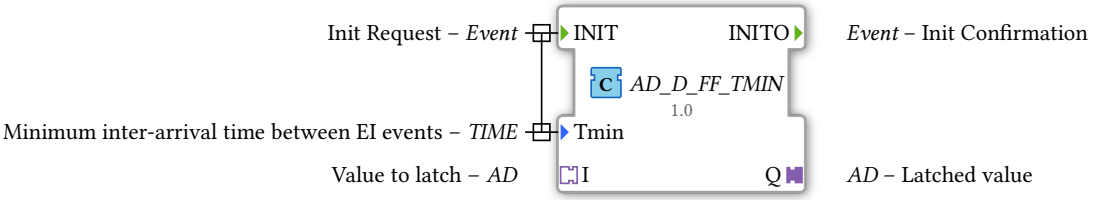
Fazit

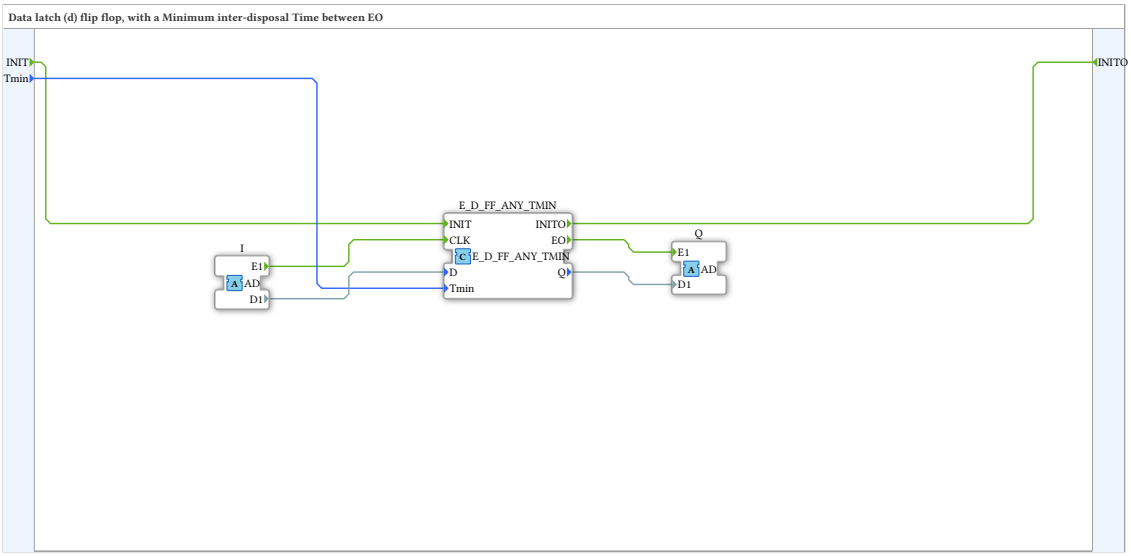
Der **AD_D_FF** ist ein schlanker, adapterbasierter D-Flipflop-Baustein für die 4diac-IDE. Er übernimmt bei jedem Taktereignis einen Datenwert und gibt ihn aus. Durch die reinen Adapter-Schnittstellen fügt er sich nahtlos in Architekturen ein, die auf standardisierte Verbindungen setzen. Seine einfache Funktionalität macht ihn zur idealen Wahl für Aufgaben der Datenpufferung und -synchronisation ohne zusätzliche Logik.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AD_D_FF_TMIN





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AD_D_FF_TMIN** realisiert ein datenrückhaltendes Flip-Flop (D-Latch) auf Basis von Adaptern mit unidirektionaler Schnittstelle. Er zeichnet sich durch eine konfigurierbare Mindestverweilzeit (*Minimum inter-disposal Time*) zwischen zwei aufeinanderfolgenden Ereignissen aus und eignet sich damit für zeitkritische Signalverarbeitungen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Initialisierungsanforderung (Übergabe der Mindestzeit)

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Initialisierungsbestätigung

Daten-Eingänge

Name	Datentyp	Kommentar
Tmin	TIME	Minimale Zeit zwischen zwei aufeinanderfolgenden Ereignissen am Socket *I*

Daten-Ausgänge

Keine direkten Datenausgänge vorhanden; die Ausgabe erfolgt ausschließlich über den Adapter *Q*.

Adapter

Typ	Richtung	Name	Kommentar
adapter::types::unidirectional::AD	**Plug** (Ausgang)	Q	Liefert den gelatchten Wert
adapter::types::unidirectional::AD	**Socket** (Eingang)	I	Empfängt den zu latchnden Wert sowie das Taktereignis

Funktionsweise

Der Baustein kapselt eine interne Instanz des Funktionsblocks `iec61499::events::E_D_FF_ANY_TMIN`. Der Socket *I* liefert sowohl das Taktereignis (über den Adapter-Ereignisausgang `E1`) als auch den Datenwert (über den Adapter-Datenausgang `D1`). Das interne Flip-Flop übernimmt bei jedem Takt den aktuellen Wert von `I.D1` an seinen Ausgang `Q`. Dieses Signal wird über den Plug *Q* (Ereignis `E1` und Daten `D1`) nach außen gegeben.

Die Mindestverweilzeit `Tmin` wird dem internen Baustein bei der Initialisierung übergeben und begrenzt die Frequenz der Taktereignisse: Ereignisse, die vor Ablauf von `Tmin` seit dem letzten Takt eintreffen, werden unterdrückt.

Technische Besonderheiten

- **Adapter-basierte Ein-/Ausgabe**: Der Baustein verwendet unidirektionale Adapter (Typ `adapter::types::unidirectional::AD`), die Ereignis und Daten in einer strukturierten Schnittstelle zusammenfassen.
- **Ereignis-gesteuerte Mindestzeit**: Die parametrierbare Zeit `Tmin` verhindert eine zu schnelle Taktfolge und schützt nachgeschaltete Logik vor Überlastung.
- **Wiederverwendung**: Die innere Logik ist als eigener FB (`E_D_FF_ANY_TMIN`) realisiert, was eine modulare Wartung und Tests erleichtert.

Zustandsübersicht

Der Baustein selbst besitzt keinen expliziten Zustandsautomaten (keine ECC-Definition). Sein Verhalten wird vollständig durch den eingebetteten FB `E_D_FF_ANY_TMIN` bestimmt. Dieser arbeitet als **D-Flip-Flop mit Zeitfilter**:

- **Ruhezustand**: Warten auf das nächste Taktereignis.
- **Taktverarbeitung**: Prüfen, ob seit dem letzten Takt die Zeitspanne `Tmin` vergangen ist. Falls ja → Datenübernahme und Erzeugung eines Ausgangsereignisses. Falls nein → Ereignis ignorieren.
- **Initialisierungszustand**: Nur durch `INIT` wird `Tmin` gesetzt und der interne Timer zurückgesetzt.

Anwendungsszenarien

- **Datenerfassung mit Mindestabstand**: Sensordaten sollen nur dann übernommen werden, wenn ein bestimmter zeitlicher Abstand zum vorherigen Wert eingehalten wird (z. B. Entprellung oder Abtastratenbegrenzung).
- **Pulsformung / Triggermanagement**: Der Baustein kann als „entprellter“ Flip-Flop in Steuerungen eingesetzt werden, um Signalflanken zu glätten und Störimpulse auszufiltern.
- **Adapter-orientierte Busarchitekturen**: Dank der Adapter-Schnittstelle lässt sich der FB nahtlos in bestehende unidirektionale Datenflüsse einbinden, z. B. in Sensor-Aktor-Netzwerken.

Vergleich mit ähnlichen Bausteinen

Baustein	Eigenschaft	Unterschied
E_D_FF	Standard-D-Flip-Flop	Keine zeitliche Einschränkung; jedes Taktereignis wird verarbeitet.
E_CTD / E_CTU	Zähler	Andere Grundfunktion; zählt Ereignisse.
AD_D_FF_TMIN	D-Flip-Flop mit *Minimum inter-disposal Time*	Verhindert zu schnelle Taktfolgen durch parametrierbare Sperrzeit.

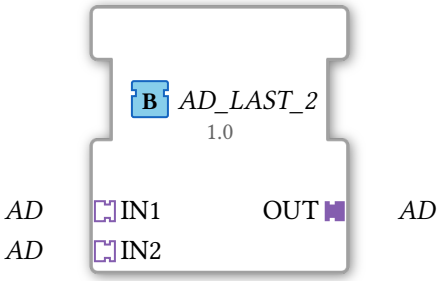
Fazit

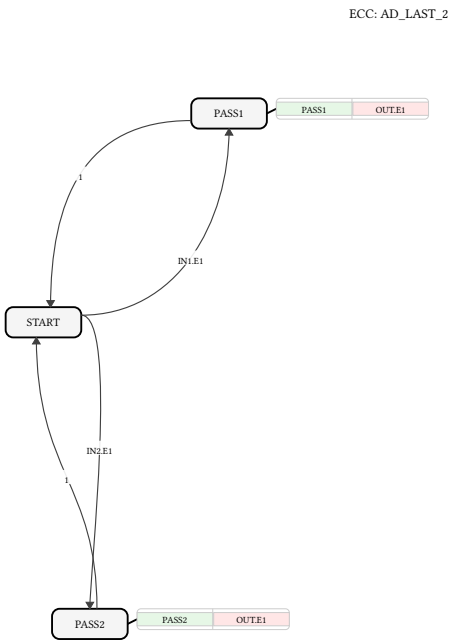
Der **AD_D_FF_TMIN** erweitert das klassische D-Flip-Flop um eine zeitliche Filterfunktion, die über den Parameter `Tmin` konfiguriert wird. Seine Adapter-Schnittstelle erlaubt eine saubere Kapselung von Ereignis und Daten in einem Stecker, was die Integration in komplexe Automatisierungssysteme vereinfacht. Der Baustein ist ideal für Anwendungen, bei denen eine minimale Ereignisdistanz sichergestellt werden muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de

AD_LAST_2





Einleitung

Der AD_LAST_2 Funktionsblock führt zwei gleichartige DWORD-Adapter-Signale (**IN1**, **IN2**) auf einen gemeinsamen Ausgang (**OUT**) zusammen, nach dem Prinzip **Last Writer Wins**: welcher der beiden Sockets zuletzt ein Ereignis geschrieben hat, gewinnt, und sein Datenwert wird sofort und unverändert an OUT durchgereicht. Es handelt sich ausdrücklich nicht um ein naives Zusammenführen (z. B. Mittelwert oder ODER-Verknüpfung), sondern um eine reine Arbitrierung nach zeitlicher Reihenfolge.

Schnittstellenstruktur

Adapter

Eingangsadapter (Sockets):

- **IN1**: DWORD-Adapter (unidirectional) - erste Signalquelle
- **IN2**: DWORD-Adapter (unidirectional) - zweite Signalquelle

Ausgangsadapter (Plug):

- **OUT**: DWORD-Adapter (unidirectional) - zusammengeführtes Signal

Funktionsweise

Der Baustein ist als einfacher Basic-FB mit einer 3-Zustands-ECC umgesetzt (START, PASS1, PASS2). Trifft an IN1 ein Ereignis ein, wechselt die ECC nach PASS1: der aktuelle Datenwert von IN1 wird nach OUT kopiert und das Adapter-Ereignis an OUT gesendet. Trifft an IN2 ein Ereignis ein, geschieht dasselbe symmetrisch über PASS2. Nach jeder Durchreichung kehrt die ECC sofort wieder nach START zurück und ist bereit für das nächste Ereignis von IN1 oder IN2.

Technische Besonderheiten

- Last-Writer-Wins-Semantik statt naiver Datenverknüpfung
- Einfacher Basic-FB (ECC mit 3 Zuständen), kein zusätzlicher Zwischenspeicher nötig
- Keine Signalverzögerung: der Wert wird im selben Zyklus durchgereicht, in dem das auslösende Ereignis eintrifft
- Beide Eingänge sind gleichberechtigt - es gibt keine feste Priorität außer der zeitlichen Reihenfolge

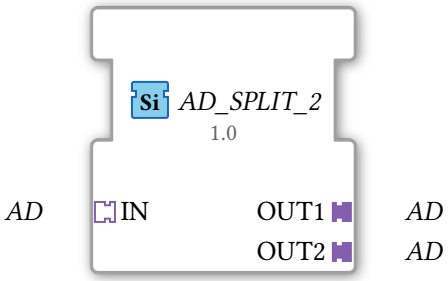
Anwendungsszenarien

- Zusammenführen zweier gleichwertiger DWORD-Signalquellen zu einem einzigen nachgeschalteten Adapterpfad (z. B. zwei redundante Sensoren, oder ein manueller und ein automatischer Sollwertgeber)
- Arbitrierung zwischen zwei Bedienstellen, die denselben DWORD-Wert schreiben dürfen
- Vereinfachung von Netzwerken, die sonst ein `F_SEL`/`E_RS`-Konstrukt inline verdrahten müssten

⚖️ Vergleich mit ähnlichen Bausteinen

AD_LAST_2 ist die Umkehrung von `AD_SPLIT_2`: AD_SPLIT_2 verteilt 1 Eingang auf 2 Ausgänge, AD_LAST_2 führt 2 Eingänge auf 1 Ausgang zusammen. Für Mischbetrieb zwischen einem Daten- und einem reinen Ereignisadapter (z. B. AX + AE) ist stattdessen `AX_AE_MERGE` gedacht, das nur das Ereignis mischt, ohne den Datenwert zu verändern.

AD_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AD_SPLIT_2 dient dazu, einen eingehenden unidirektionalen Adapter (AD) auf zwei identische Ausgangsadapter aufzuteilen. Er ist als generischer Baustein implementiert und ermöglicht die einfache Verzweigung eines Adaptersignals in zwei parallele Pfade.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Name	Typ	Richtung
IN	adapter::types::unidirectional::AD	Socket (Eingang)
OUT1	adapter::types::unidirectional::AD	Plug (Ausgang)
OUT2	adapter::types::unidirectional::AD	Plug (Ausgang)

Funktionsweise

Der Baustein empfängt einen unidirektionalen Adapter über den Socket `IN`. Alle über diesen Adapter übertragenen Daten und Ereignisse werden unverändert an beide Ausgangsadapter `OUT1` und `OUT2` weitergegeben. Es findet keine Vebarbeitung oder Filterung statt – die Signale werden lediglich dupliziert. Die Aufteilung erfolgt ohne Zeitverzögerung und ohne interne Pufferung.

Technische Besonderheiten

- **Generische Ausführung**: Der FB nutzt die Generics-Mechanismen von Eclipse 4diac (`GenericClassName`). Dadurch kann der konkrete Adaptertyp zur Entwurfszeit oder Laufzeit parametrisiert werden, solange er dem Adapter-Schema `unidirectional::AD` entspricht.
- **Typreinheit**: Die Ausgangsadapter sind vom selben Typ wie der Eingangsadapter; eine Typkonvertierung findet nicht statt.
- **Keine Zustandslogik**: Der Baustein besitzt keine interne Zustandsmaschine (ECC) und reagiert ereignisgesteuert auf die Adapterkommunikation.

Zustandsübersicht

Der Funktionsblock hat keine expliziten Zustände. Er arbeitet kontinuierlich und leitet jedes ankommende Signal sofort an beide Ausgänge weiter.

Anwendungsszenarien

- Aufteilung eines Sensordatenstroms auf zwei unabhängige Steuerungs- oder Überwachungseinheiten.
- Parallelisierung von Adaptersignalen in industriellen Automatisierungssystemen (z. B. Verteilung eines Feldbus-Adapterwerts auf zwei Regelkreise).
- Test- und Simulationsaufbauten, bei denen ein Signal simultan an verschiedene Verbraucher gesendet werden muss.

Vergleich mit ähnlichen Bausteinen

- **AD_SPLIT_3, AD_SPLIT_4**: Erweiterungen mit mehreren Ausgängen, basierend auf demselben Mechanismus.
- **AD_MERGE**: Führt mehrere Adaptereingänge zu einem Ausgang zusammen – die Umkehrfunktion zu AD_SPLIT_2.
- **AD_SELECT**: Wählt einen von mehreren Adaptereingängen aus und gibt ihn weiter, erfordert hingegen eine Steuerlogik.
- `AD_SPLIT_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

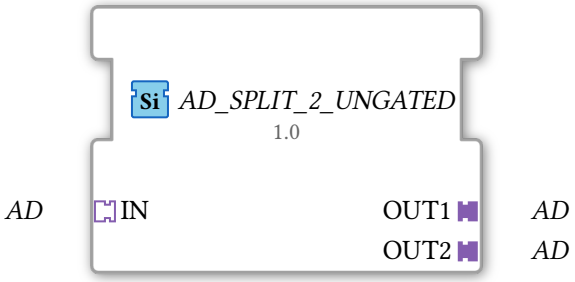
Der AD_SPLIT_2 ist ein einfacher, aber effektiver Baustein zur Signalverteilung in Adapter-basierten Architekturen. Seine generische Natur und die fehlende Zustandslogik machen ihn leicht konfigurierbar und ressourcenschonend. Er eignet sich besonders für modulare Automatisierungsprojekte, in denen ein Adaptersignal auf mehrere nachgelagerte Komponenten verteilt werden muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AD_SPLIT_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AD_SPLIT_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AD_SPLIT_2_UNGATED dient dazu, einen eingehenden unidirektionalen Adapter (AD) auf zwei identische Ausgangsadapter aufzuteilen. Er ist als generischer Baustein implementiert und ermöglicht die einfache Verzweigung eines Adaptersignals in zwei parallele Pfade.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Name	Typ	Richtung
IN	adapter::types::unidirectional::AD	Socket (Eingang)
OUT1	adapter::types::unidirectional::AD	Plug (Ausgang)
OUT2	adapter::types::unidirectional::AD	Plug (Ausgang)

Funktionsweise

Der Baustein empfängt einen unidirektionalen Adapter über den Socket `IN`. Alle über diesen Adapter übertragenen Daten und Ereignisse werden unverändert an beide Ausgangsadapter `OUT1` und `OUT2` weitergegeben. Es findet keine Vebarbeitung oder Filterung statt – die Signale werden lediglich dupliziert. Die Aufteilung erfolgt ohne Zeitverzögerung und ohne interne Pufferung.

Technische Besonderheiten

- **Generische Ausführung**: Der FB nutzt die Generics-Mechanismen von Eclipse 4diac (`GenericClassName`). Dadurch kann der konkrete Adaptertyp zur Entwurfszeit oder Laufzeit parametrisiert werden, solange er dem Adapter-Schema `unidirectional::AD` entspricht.
- **Typreinheit**: Die Ausgangsadapter sind vom selben Typ wie der Eingangsadapter; eine Typkonvertierung findet nicht statt.
- **Keine Zustandslogik**: Der Baustein besitzt keine interne Zustandsmaschine (ECC) und reagiert ereignisgesteuert auf die Adapterkommunikation.

Zustandsübersicht

Der Funktionsblock hat keine expliziten Zustände. Er arbeitet kontinuierlich und leitet jedes ankommende Signal sofort an beide Ausgänge weiter.

Anwendungsszenarien

- Aufteilung eines Sensordatenstroms auf zwei unabhängige Steuerungs- oder Überwachungseinheiten.
- Parallelisierung von Adaptersignalen in industriellen Automatisierungssystemen (z. B. Verteilung eines Feldbus-Adapterwerts auf zwei Regelkreise).
- Test- und Simulationsaufbauten, bei denen ein Signal simultan an verschiedene Verbraucher gesendet werden muss.

Vergleich mit ähnlichen Bausteinen

- **AD_SPLIT_3, AD_SPLIT_4**: Erweiterungen mit mehreren Ausgängen, basierend auf demselben Mechanismus.
- **AD_MERGE**: Führt mehrere Adaptereingänge zu einem Ausgang zusammen – die Umkehrfunktion zu AD_SPLIT_2_UNGATED.
- **AD_SELECT**: Wählt einen von mehreren Adaptereingängen aus und gibt ihn weiter, erfordert hingegen eine Steuerlogik.
- `AD_SPLIT_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

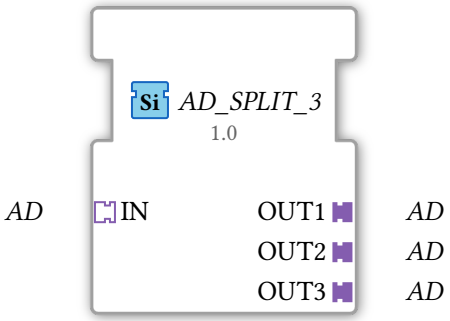
Fazit

Der AD_SPLIT_2_UNGATED ist ein einfacher, aber effektiver Baustein zur Signalverteilung in Adapterbasierten Architekturen. Seine generische Natur und die fehlende Zustandslogik machen ihn leicht konfigurierbar und ressourcenschonend. Er eignet sich besonders für modulare Automatisierungsprojekte, in denen ein Adaptersignal auf mehrere nachgelagerte Komponenten verteilt werden muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AD_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AD_SPLIT_3` dient als generischer Splitter für einen unidirektionalen Adapter (AD). Er nimmt einen eingehenden AD entgegen und leitet ihn an drei separate Ausgänge weiter. Dies ermöglicht eine Mehrfachnutzung desselben Daten- oder Signalpfades.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Typ	Richtung	Bezeichnung	Beschreibung
Socket (Eingang)	Input	IN	Unidirektionaler Adapter-Eingang, der auf die drei Ausgänge verteilt wird.
Plug (Ausgang)	Output	OUT1	Erster Ausgang mit dupliziertem Adapter-Signal.
Plug (Ausgang)	Output	OUT2	Zweiter Ausgang mit dupliziertem Adapter-Signal.
Plug (Ausgang)	Output	OUT3	Dritter Ausgang mit dupliziertem Adapter-Signal.

Alle Adapter sind vom Typ `adapter::types::unidirectional::AD`.

Funktionsweise

Der Baustein arbeitet als reiner Splitter: Der am Socket `IN` anliegende Adapter (Daten- oder Ereignisstrom) wird auf die drei Plugs `OUT1`, `OUT2` und `OUT3` identisch vervielfältigt. Änderungen oder Ereignisse am Eingang wirken sich unverzögert auf alle drei Ausgänge aus. Es erfolgt keine Umformung, Pufferung oder Logik – die Signalverteilung geschieht transparent.

Technische Besonderheiten

- **Generischer Typ**: Der FB ist als generischer Adapter-Splitter ausgelegt (`GenericClassName = 'GEN_AD_SPLIT'`) und kann mit beliebigen unidirektionalen Adaptern des Typs `adapter::types::unidirectional::AD` verwendet werden.
- **Reine Adapterschnittstelle**: Der Baustein besitzt weder Ereignis- noch Daten-Ein-/Ausgänge im klassischen Sinne, sondern kommuniziert ausschließlich über Adapter-Plugs und -Sockets.

- **Unidirektionalität**: Die Kommunikation erfolgt nur in eine Richtung (vom Socket zu den Plugs), daher ist keine Rückkopplung oder Handshake-Mechanismus erforderlich.
- **Keine Laufzeitabhängigkeit**: Die Verteilung erfolgt instantan und ohne Verzögerung.

Zustandsübersicht

Der `AD_SPLIT_3` verfügt über kein eigenes Zustandsdiagramm (ECC). Die Ausgänge folgen dem Eingang ohne innere Logik oder Speicherverhalten. Daher existiert nur ein einziger, passiver Zustand, in dem ständig die aktuelle Eingabe an die Ausgänge weitergegeben wird.

Anwendungsszenarien

1. **Signalfan-Out**: Ein Sensorsignal oder ein Befehl (z. B. Steuerwort oder Status) soll an mehrere parallele Verbraucher gesendet werden.
2. **Protokollierung**: Ein Datenstrom wird einmal verarbeitet und ein zweites Mal zur Aufzeichnung oder Überwachung weitergeleitet.
3. **Redundanz**: Ein kritischer Adapter-Pfad wird auf mehrere unabhängige Systeme verteilt.
4. **Testumgebungen**: In Simulations- oder Testaufbauten kann ein Signal an mehrere Prüfstellen gespiegelt werden.

Vergleich mit ähnlichen Bausteinen

- **AD_SPLIT_2**: Verteilt den Eingang auf nur zwei Ausgänge. `AD_SPLIT_3` erweitert dies auf drei Ausgänge.
- **AD_MERGE**: Führt mehrere Adapter-Eingänge zu einem Ausgang zusammen – funktionell das Gegenteil.
- **Spezifische Daten-Splitter**: Bausteine wie `SPLIT_INT` oder `SPLIT_BOOL` arbeiten auf Daten-Ebene, nicht auf Adapter-Ebene. `AD_SPLIT_3` ist für vollständige Adapterstrukturen (Daten plus Ereignisse) gedacht.
- `AD_SPLIT_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

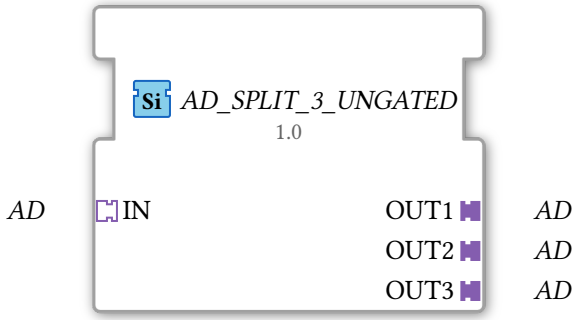
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der `AD_SPLIT_3` ist ein einfacher, aber nützlicher generischer Baustein zur Vervielfachung von unidirektionalen Adaptern. Durch seine reine Adapterschnittstelle und fehlende Logik eignet er sich besonders für modulare Architekturen, in denen ein Signal auf mehrere Empfänger verteilt werden muss, ohne zusätzliche Komplexität einzuführen.

AD_SPLIT_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AD_SPLIT_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AD_SPLIT_3_UNGATED` dient als generischer Splitter für einen unidirektionalen Adapter (AD). Er nimmt einen eingehenden AD entgegen und leitet ihn an drei separate Ausgänge weiter. Dies ermöglicht eine Mehrfachnutzung desselben Daten- oder Signalpfades.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Typ	Richtung	Bezeichnung	Beschreibung
Socket (Eingang)	Input	IN	Unidirektionaler Adapter-Eingang, der auf die drei Ausgänge verteilt wird.
Plug (Ausgang)	Output	OUT1	Erster Ausgang mit dupliziertem Adapter-Signal.
Plug (Ausgang)	Output	OUT2	Zweiter Ausgang mit dupliziertem Adapter-Signal.
Plug (Ausgang)	Output	OUT3	Dritter Ausgang mit dupliziertem Adapter-Signal.

Alle Adapter sind vom Typ `adapter::types::unidirectional::AD`.

Funktionsweise

Der Baustein arbeitet als reiner Splitter: Der am Socket `IN` anliegende Adapter (Daten- oder Ereignisstrom) wird auf die drei Plugs `OUT1`, `OUT2` und `OUT3` identisch vervielfältigt. Änderungen oder Ereignisse am Eingang wirken sich unverzögert auf alle drei Ausgänge aus. Es erfolgt keine Umformung, Pufferung oder Logik – die Signalverteilung geschieht transparent.

Technische Besonderheiten

- **Generischer Typ**: Der FB ist als generischer Adapter-Splitter ausgelegt (`GenericClassName = 'GEN_AD_SPLIT'`) und kann mit beliebigen unidirektionalen Adaptern des Typs `adapter::types::unidirectional::AD` verwendet werden.
- **Reine Adapterschnittstelle**: Der Baustein besitzt weder Ereignis- noch Daten-Ein-/Ausgänge im klassischen Sinne, sondern kommuniziert ausschließlich über Adapter-Plugs und -Sockets.
- **Unidirektionalität**: Die Kommunikation erfolgt nur in eine Richtung (vom Socket zu den Plugs), daher ist keine Rückkopplung oder Handshake-Mechanismus erforderlich.

- **Keine Laufzeitabhängigkeit**: Die Verteilung erfolgt instantan und ohne Verzögerung.

Zustandsübersicht

Der `AD_SPLIT_3_UNGATED` verfügt über kein eigenes Zustandsdiagramm (ECC). Die Ausgänge folgen dem Eingang ohne innere Logik oder Speicherverhalten. Daher existiert nur ein einziger, passiver Zustand, in dem ständig die aktuelle Eingabe an die Ausgänge weitergegeben wird.

Anwendungsszenarien

1. **Signalfan-Out**: Ein Sensorsignal oder ein Befehl (z. B. Steuerwort oder Status) soll an mehrere parallele Verbraucher gesendet werden.
2. **Protokollierung**: Ein Datenstrom wird einmal verarbeitet und ein zweites Mal zur Aufzeichnung oder Überwachung weitergeleitet.
3. **Redundanz**: Ein kritischer Adapter-Pfad wird auf mehrere unabhängige Systeme verteilt.
4. **Testumgebungen**: In Simulations- oder Testaufbauten kann ein Signal an mehrere Prüfstellen gespiegelt werden.

Vergleich mit ähnlichen Bausteinen

- **AD_SPLIT_2**: Verteilt den Eingang auf nur zwei Ausgänge. `AD_SPLIT_3_UNGATED` erweitert dies auf drei Ausgänge.
- **AD_MERGE**: Führt mehrere Adapter-Eingänge zu einem Ausgang zusammen – funktionell das Gegenteil.
- **Spezifische Daten-Splitter**: Bausteine wie `SPLIT_INT` oder `SPLIT_BOOL` arbeiten auf Daten-Ebene, nicht auf Adapter-Ebene. `AD_SPLIT_3_UNGATED` ist für vollständige Adapterstrukturen (Daten plus Ereignisse) gedacht.
- **`AD_SPLIT_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

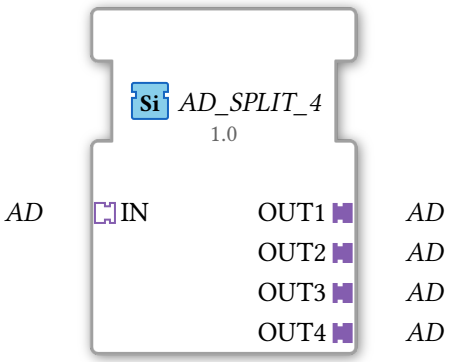
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AD_SPLIT_3_UNGATED` ist ein einfacher, aber nützlicher generischer Baustein zur Vervielfachung von unidirektionalen Adaptern. Durch seine reine Adapterschnittstelle und fehlende Logik eignet er sich besonders für modulare Architekturen, in denen ein Signal auf mehrere Empfänger verteilt werden muss, ohne zusätzliche Komplexität einzuführen.

AD_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AD_SPLIT_4** dient dazu, einen eingehenden unidirektionalen AD-Adapter (Adapter Definition) auf vier identische Ausgänge zu verteilen. Es handelt sich um einen generischen Baustein, der speziell für die Aufteilung eines einzelnen AD-Signals in bis zu vier separate Pfade konzipiert ist.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge
Keine Daten-Eingänge vorhanden. Die Datenübertragung erfolgt ausschließlich über den eingehenden AD-Adapter.

Daten-Ausgänge
Keine Daten-Ausgänge vorhanden. Die Ausgabe erfolgt ausschließlich über die vier ausgehenden AD-Adapter.

Adapter
Der Baustein verwendet ausschließlich Adapter vom Typ `adapter::types::unidirectional::AD`:

- **Socket (Eingang)** `IN` – Ein unidirektionaler AD-Eingang, der das aufzuteilende Signal empfängt.
- **Plugs (Ausgänge)** `OUT1`, `OUT2`, `OUT3`, `OUT4` – Vier unidirektionale AD-Ausgänge, die das vom Eingang empfangene Signal identisch weiterleiten.

Funktionsweise
Der **AD_SPLIT_4**-Baustein kopiert den an seinem Socket `IN` ankommenden AD-Datenstrom (ohne Ereignissteuerung) und stellt diesen an allen vier Plugs `OUT1` bis `OUT4` parallel zur Verfügung. Die Verteilung erfolgt ohne Verzögerung oder Pufferung – jede Änderung am Eingang wird sofort an alle Ausgänge propagiert. Der Baustein ist generisch ausgelegt, sodass der konkrete Datentyp des AD-Adapters erst zur Laufzeit festgelegt wird (GenericClassName = `'GEN_AD_SPLIT'`).

Technische Besonderheiten
- **Generischer Typ**: Der Baustein ist als generischer Funktionsblock implementiert. Der tatsächliche AD-Datentyp wird durch das Attribut `eclipse4diac::core::GenericClassName ('GEN_AD_SPLIT')` bestimmt.
- **Unidirektionale Kommunikation**: Der verwendete Adapter `adapter::types::unidirectional::AD` erlaubt nur eine Datenübertragung vom Socket zu den Plugs – eine Rückkopplung ist nicht vorgesehen.
- **Keine Ereignissteuerung**: Der Baustein benötigt keine Ereignisse, da die Datenweitergabe rein datengetrieben (oder durch die zugrundeliegende Adapter-Logik) erfolgt.

Zustandsübersicht
Da der Baustein keine Ereignis-Ein-/Ausgänge besitzt, gibt es keine Zustandsmaschine. Das Verhalten ist deterministisch: Die Ausgänge spiegeln stets den aktuellen Zustand des Eingangs wider.

Anwendungsszenarien
- **Signalverteilung**: Wenn ein einziges AD-Signal (z. B. ein Messwert oder ein Steuerungssignal) an mehrere nachfolgende Funktionsblöcke oder Hardware-Komponenten weitergeleitet werden muss.
- **Monitoring und Kontrolle**: Gleichzeitige Überwachung eines Signals in verschiedenen Logikeinheiten, ohne dass die Signalquelle mehrfach bereitgestellt werden muss.
- **Redundanzpfade**: Aufteilung eines Signals für parallele, unabhängige Verarbeitungspfade.

Vergleich mit ähnlichen Bausteinen
- **AD_SPLIT_2**, **AD_SPLIT_8**: Verwandte Bausteine mit identischer Funktionsweise, jedoch für zwei bzw. acht Ausgänge. Die Wahl hängt von der Anzahl der benötigten Verteilungen ab.
- **AD_MERGE_4**: Ein Gegenstück, das mehrere AD-Eingänge zu einem Ausgang zusammenführt, im Gegensatz zur Aufteilung.
- **Reine Daten-Split-Bausteine (z. B. D_SPLIT_4)**: Diese teilen einzelne Datenwerte (z. B. INT, REAL) auf, während AD_SPLIT_4 speziell für den komplexen AD-Adapter-Typ ausgelegt ist.

- `AD_SPLIT_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

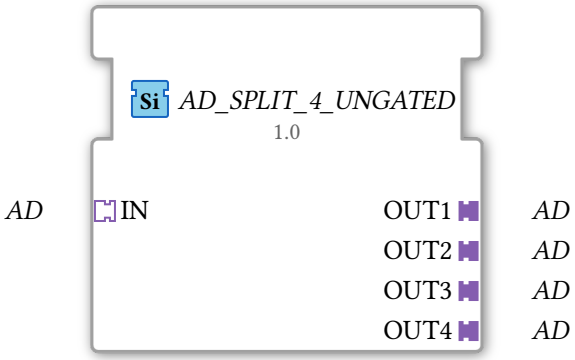
Der **AD_SPLIT_4**-Funktionsblock ist ein kompaktes, generisches Werkzeug zur einfachen Vervielfältigung eines unidirektionalen AD-Signals. Seine klare Schnittstelle ohne Ereignisse macht ihn besonders geeignet für datenflussorientierte Applikationen in der Automatisierungstechnik, bei denen ein Signal an mehrere Empfänger verteilt werden muss. Durch die generische Auslegung kann er flexibel an verschiedene Adapter-Datentypen angepasst werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AD_SPLIT_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AD_SPLIT_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AD_SPLIT_4_UNGATED** dient dazu, einen eingehenden unidirektionalen AD-Adapter (Adapter Definition) auf vier identische Ausgänge zu verteilen. Es handelt sich um einen generischen Baustein, der speziell für die Aufteilung eines einzelnen AD-Signals in bis zu vier separate Pfade konzipiert ist.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden. Die Datenübertragung erfolgt ausschließlich über den eingehenden AD-Adapter.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden. Die Ausgabe erfolgt ausschließlich über die vier ausgehenden AD-Adapter.

Adapter

Der Baustein verwendet ausschließlich Adapter vom Typ `adapter::types::unidirectional::AD`:

- **Socket (Eingang)** `IN` – Ein unidirektionaler AD-Eingang, der das aufzuteilende Signal empfängt.
- **Plugs (Ausgänge)** `OUT1`, `OUT2`, `OUT3`, `OUT4` – Vier unidirektionale AD-Ausgänge, die das vom Eingang empfangene Signal identisch weiterleiten.

Funktionsweise

Der **AD_SPLIT_4_UNGATED**-Baustein kopiert den an seinem Socket `IN` ankommenden AD-Datenstrom (ohne Ereignissteuerung) und stellt diesen an allen vier Plugs `OUT1` bis `OUT4` parallel zur Verfügung. Die Verteilung erfolgt ohne Verzögerung oder Pufferung – jede Änderung am Eingang wird sofort an alle Ausgänge propagiert. Der Baustein ist generisch ausgelegt, sodass der konkrete Datentyp des AD-Adapters erst zur Laufzeit festgelegt wird (GenericClassName = `'GEN_AD_SPLIT'`).

Technische Besonderheiten

- **Generischer Typ**: Der Baustein ist als generischer Funktionsblock implementiert. Der tatsächliche AD-Datentyp wird durch das Attribut `eclipse4diac::core::GenericClassName ('GEN_AD_SPLIT')` bestimmt.
- **Unidirektionale Kommunikation**: Der verwendete Adapter `adapter::types::unidirectional::AD` erlaubt nur eine Datenübertragung vom Socket zu den Plugs – eine Rückkopplung ist nicht vorgesehen.
- **Keine Ereignissteuerung**: Der Baustein benötigt keine Ereignisse, da die Datenweitergabe rein datengetrieben (oder durch die zugrundeliegende Adapter-Logik) erfolgt.

Zustandsübersicht

Da der Baustein keine Ereignis-Ein-/Ausgänge besitzt, gibt es keine Zustandsmaschine. Das Verhalten ist deterministisch: Die Ausgänge spiegeln stets den aktuellen Zustand des Eingangs wider.

Anwendungsszenarien

- **Signalverteilung**: Wenn ein einziges AD-Signal (z. B. ein Messwert oder ein Steuerungssignal) an mehrere nachfolgende Funktionsblöcke oder Hardware-Komponenten weitergeleitet werden muss.
- **Monitoring und Kontrolle**: Gleichzeitige Überwachung eines Signals in verschiedenen Logikeinheiten, ohne dass die Signalquelle mehrfach bereitgestellt werden muss.
- **Redundanzpfade**: Aufteilung eines Signals für parallele, unabhängige Verarbeitungspfade.

Vergleich mit ähnlichen Bausteinen

- **AD_SPLIT_2**, **AD_SPLIT_8**: Verwandte Bausteine mit identischer Funktionsweise, jedoch für zwei bzw. acht Ausgänge. Die Wahl hängt von der Anzahl der benötigten Verteilungen ab.
- **AD_MERGE_4**: Ein Gegenstück, das mehrere AD-Eingänge zu einem Ausgang zusammenführt, im Gegensatz zur Aufteilung.
- **Reine Daten-Split-Bausteine (z. B. D_SPLIT_4)**: Diese teilen einzelne Datenwerte (z. B. INT, REAL) auf, während AD_SPLIT_4_UNGATED speziell für den komplexen AD-Adapter-Typ ausgelegt ist.
- `AD_SPLIT_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

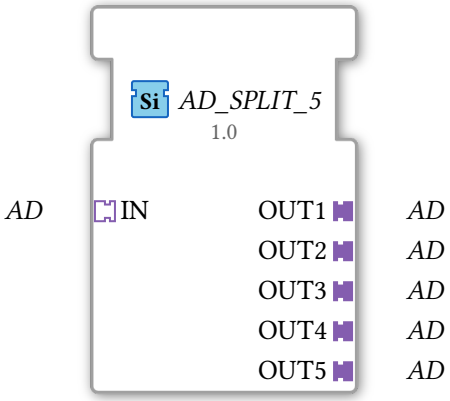
Fazit

Der **AD_SPLIT_4_UNGATED**-Funktionsblock ist ein kompaktes, generisches Werkzeug zur einfachen Vervielfältigung eines unidirektionalen AD-Signals. Seine klare Schnittstelle ohne Ereignisse macht ihn besonders geeignet für datenflussorientierte Applikationen in der Automatisierungstechnik, bei denen ein Signal an mehrere Empfänger verteilt werden muss. Durch die generische Auslegung kann er flexibel an verschiedene Adapter-Datentypen angepasst werden.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AD_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AD_SPLIT_5` ist ein generischer Baustein, der einen eingehenden unidirektionalen Adapter (Typ `adapter::types::unidirectional::AD`) auf fünf identische Ausgangsadapter aufteilt. Er dient dazu, ein Adaptersignal parallel an mehrere nachfolgende Bausteine weiterzuleiten, ohne dass der Eingang mehrfach verbunden werden muss.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden. Die gesamte Kommunikation erfolgt über den Adapter-Socket.

Daten-Ausgänge

Keine vorhanden. Die Ausgaben erfolgen ausschließlich über die Adapter-Plugs.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AD`	`IN`	Socket (Eingang)	Eingehendes Adaptersignal, welches auf alle Ausgänge verteilt wird.
`adapter::types::unidirectional::AD`	`OUT1`	Plug (Ausgang)	Erster ausgehender Adapter, identisch zum Eingangssignal.
`adapter::types::unidirectional::AD`	`OUT2`	Plug (Ausgang)	Zweiter ausgehender Adapter.

adapter::types::unidirectional::AD	OUT3	Plug (Ausgang)	Dritter ausgehender Adapter.
adapter::types::unidirectional::AD	OUT4	Plug (Ausgang)	Vierter ausgehender Adapter.
adapter::types::unidirectional::AD	OUT5	Plug (Ausgang)	Fünfter ausgehender Adapter.

Funktionsweise

Der Baustein besitzt keine Ereignis- oder Datenports im herkömmlichen Sinne. Stattdessen wird ein eingehender Adapter `IN` auf alle fünf ausgehenden Adapter `OUT1` bis `OUT5` durchgeschleift. Das bedeutet, dass Änderungen oder Ereignisse, die über den Eingangsadapter transportiert werden, an alle fünf Ausgänge gleichzeitig weitergegeben werden. Die Vervielfachung erfolgt ohne logische Verarbeitung oder Verzögerung.

Technische Besonderheiten

- **Generischer Typ**: Der Funktionsblock ist als generischer Baustein implementiert (Attribute `eclipse4diac::core::GenericClassName = 'GEN_AD_SPLIT'`). Dadurch kann er in verschiedenen Kontexten mit dem Adaptertyp `adapter::types::unidirectional::AD` verwendet werden.
- **Keine Zustandsmaschine**: Der Block besitzt kein ECC (Execution Control Chart) und arbeitet rein datenflussgetrieben. Er führt keine speichernden Operationen durch.
- **Unidirektionale Adapter**: Alle Adapter sind unidirektional, d.h. die Daten fließen nur in eine Richtung – vom Socket zu den Plugs.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände oder Schritte. Es handelt sich um einen reinen Verteiler (Splitter) ohne eigenes Verhalten. Eine Zustandsübersicht entfällt daher.

Anwendungsszenarien

- **Signalverteilung**: Wenn ein Adaptersignal mehrfach benötigt wird, z. B. um mehrere parallele Berechnungen oder Ausgaben zu speisen.
- **Modulare Verkettung**: In komplexen Automatisierungssystemen, in denen ein Sensorwert (als Adapter) an verschiedene Subsysteme weitergegeben werden muss.
- **Prototyping**: Schnelles Aufbauen von Testumgebungen, bei denen ein Signal an mehrere Stellen gleichzeitig gelangen soll.

Vergleich mit ähnlichen Bausteinen

- **AD_SPLIT_2, AD_SPLIT_3, AD_SPLIT_4**: Dieselbe Funktionalität mit zwei, drei bzw. vier Ausgängen. `AD_SPLIT_5` ist die Fünffach-Version.
- **Manuelle Verkabelung**: Ohne diesen Baustein müsste der Ausgang des Quellbausteins mehrfach verbunden werden, was in der 4diac-IDE nicht direkt möglich ist (ein Port kann nur eine Verbindung haben). Der Splitter löst dieses Problem.
- **Ereignisbasierte Splitter**: Es gibt auch Splitter für Ereignisse (z. B. `E_SPLIT`), die jedoch auf Ereignis- und Datenports basieren. `AD_SPLIT_5` arbeitet dagegen ausschließlich auf Adapterebene.
- **`AD_SPLIT_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

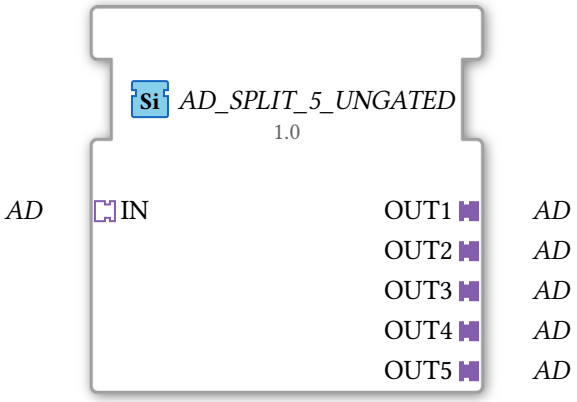
Der Funktionsblock `AD_SPLIT_5` ist ein einfaches, aber nützliches Werkzeug zur Vervielfachung von Adaptersignalen in der 4diac-Entwicklungsumgebung. Durch seine generische Auslegung und den Verzicht auf eine Zustandsmaschine bietet er eine effiziente Möglichkeit, ein Adaptersignal ohne Code-Änderungen auf fünf parallele Pfade zu verteilen. Er ist ideal für modulare und skalierbare Automatisierungsanwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AD_SPLIT_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AD_SPLIT_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AD_SPLIT_5_UNGATED` ist ein generischer Baustein, der einen eingehenden unidirektionalen Adapter (Typ `adapter::types::unidirectional::AD`) auf fünf identische Ausgangsadapter aufteilt. Er dient dazu, ein Adaptersignal parallel an mehrere nachfolgende Bausteine weiterzuleiten, ohne dass der Eingang mehrfach verbunden werden muss.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden. Die gesamte Kommunikation erfolgt über den Adapter-Socket.

Daten-Ausgänge

Keine vorhanden. Die Ausgaben erfolgen ausschließlich über die Adapter-Plugs.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AD`	`IN`	Socket (Eingang)	Eingehendes Adaptersignal, welches auf alle Ausgänge verteilt wird.
`adapter::types::unidirectional::AD`	`OUT1`	Plug (Ausgang)	Erster ausgehender Adapter, identisch zum Eingangssignal.

adapter::types::unidirectional::AD	OUT2	Plug (Ausgang)	Zweiter ausgehender Adapter.
adapter::types::unidirectional::AD	OUT3	Plug (Ausgang)	Dritter ausgehender Adapter.
adapter::types::unidirectional::AD	OUT4	Plug (Ausgang)	Vierter ausgehender Adapter.
adapter::types::unidirectional::AD	OUT5	Plug (Ausgang)	Fünfter ausgehender Adapter.

Funktionsweise

Der Baustein besitzt keine Ereignis- oder Datenports im herkömmlichen Sinne. Stattdessen wird ein eingehender Adapter `IN` auf alle fünf ausgehenden Adapter `OUT1` bis `OUT5` durchgeschleift. Das bedeutet, dass Änderungen oder Ereignisse, die über den Eingangsadapter transportiert werden, an alle fünf Ausgänge gleichzeitig weitergegeben werden. Die Vervielfachung erfolgt ohne logische Verarbeitung oder Verzögerung.

Technische Besonderheiten

- **Generischer Typ**: Der Funktionsblock ist als generischer Baustein implementiert (Attribute `eclipse4diac::core::GenericClassName = 'GEN_AD_SPLIT'`). Dadurch kann er in verschiedenen Kontexten mit dem Adaptertyp `adapter::types::unidirectional::AD` verwendet werden.
- **Keine Zustandsmaschine**: Der Block besitzt kein ECC (Execution Control Chart) und arbeitet rein datenflussgetrieben. Er führt keine speichernden Operationen durch.
- **Unidirektionale Adapter**: Alle Adapter sind unidirektional, d.h. die Daten fließen nur in eine Richtung – vom Socket zu den Plugs.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände oder Schritte. Es handelt sich um einen reinen Verteiler (Splitter) ohne eigenes Verhalten. Eine Zustandsübersicht entfällt daher.

Anwendungsszenarien

- **Signalverteilung**: Wenn ein Adaptersignal mehrfach benötigt wird, z. B. um mehrere parallele Berechnungen oder Ausgaben zu speisen.
- **Modulare Verkettung**: In komplexen Automatisierungssystemen, in denen ein Sensorwert (als Adapter) an verschiedene Subsysteme weitergegeben werden muss.
- **Prototyping**: Schnelles Aufbauen von Testumgebungen, bei denen ein Signal an mehrere Stellen gleichzeitig gelangen soll.

Vergleich mit ähnlichen Bausteinen

- **AD_SPLIT_2, AD_SPLIT_3, AD_SPLIT_4**: Dieselbe Funktionalität mit zwei, drei bzw. vier Ausgängen. `AD_SPLIT_5_UNGATED` ist die Fünffach-Version.
- **Manuelle Verkabelung**: Ohne diesen Baustein müsste der Ausgang des Quellbausteins mehrfach verbunden werden, was in der 4diac-IDE nicht direkt möglich ist (ein Port kann nur eine Verbindung haben). Der Splitter löst dieses Problem.
- **Ereignisbasierte Splitter**: Es gibt auch Splitter für Ereignisse (z. B. `E_SPLIT`), die jedoch auf Ereignis- und Datenports basieren. `AD_SPLIT_5_UNGATED` arbeitet dagegen ausschließlich auf Adapterebene.
- `AD_SPLIT_5_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- `AD_SPLIT_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

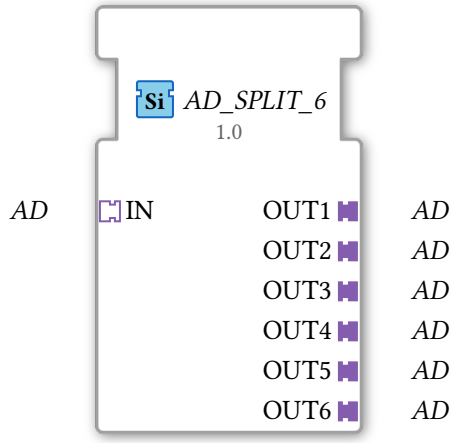
Der Funktionsblock `AD_SPLIT_5_UNGATED` ist ein einfaches, aber nützliches Werkzeug zur Vervielfachung von Adaptersignalen in der 4diac-Entwicklungsumgebung. Durch seine generische Auslegung und den Verzicht auf eine Zustandsmaschine bietet er eine effiziente Möglichkeit, ein Adaptersignal ohne

Code-Änderungen auf fünf parallele Pfade zu verteilen. Er ist ideal für modulare und skalierbare Automatisierungsanwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AD_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AD_SPLIT_6** dient der Verteilung eines eingehenden Adapters vom Typ `adapter::types::unidirectional::AD` auf sechs separate Ausgangsadapter desselben Typs. Er ist als generischer Baustein konzipiert, der eine einfache 1:6-Aufteilung ohne zusätzliche Logik oder Zustandsverwaltung realisiert.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AD`	IN	Socket	Eingangsadapter, der auf die sechs Ausgänge verteilt wird.
`adapter::types::unidirectional::AD`	OUT1	Plug	Erster Ausgangsadapter (identisch zu IN).
`adapter::types::unidirectional::AD`	OUT2	Plug	Zweiter Ausgangsadapter.
`adapter::types::unidirectional::AD`	OUT3	Plug	Dritter Ausgangsadapter.
`adapter::types::unidirectional::AD`	OUT4	Plug	Vierter Ausgangsadapter.

`adapter::types::unidirectional::AD`	OUT5	Plug	Fünfter Ausgangsadapter.
`adapter::types::unidirectional::AD`	OUT6	Plug	Sechster Ausgangsadapter.

Funktionsweise

Der Baustein leitet den über den Socket **IN** empfangenen Adapter unverändert an alle sechs Plugs (**OUT1** bis **OUT6**) weiter. Es findet keine Datenmanipulation, Filterung oder Verzögerung statt. Der Adapter wird als Referenz oder durch direkte Kopplung (abhängig von der Laufzeitumgebung) auf die Ausgänge multipliziert. Somit erhalten alle angeschlossenen nachfolgenden Funktionsbausteine denselben Adapter-Inhalt.

Technische Besonderheiten

- **Generischer Baustein**: Das Attribut `eclipse4diac::core::GenericClassName` ist auf `'GEN_AD_SPLIT'` gesetzt, was die Wiederverwendung in verschiedenen Typkonfigurationen ermöglicht.
- **Keine Ereignisse oder Daten**: Der Baustein arbeitet ausschließlich über Adapter-Schnittstellen; Ereignisse oder direkte Datenverbindungen werden nicht benötigt.
- **Einfache Topologie**: Die 1:6-Verteilung ist fest verdrahtet und kann nicht dynamisch angepasst werden.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände oder Zustandsautomaten. Er verhält sich passiv und gibt den Eingangswert permanent an alle Ausgänge weiter. Es gibt keine zeitlichen Bedingungen oder Wechsel zwischen Modi.

Anwendungsszenarien

- **Signalverteilung** in Steuerungsanwendungen, bei denen eine AD-Adapter-Schnittstelle an mehrere nachfolgende Bausteine verteilt werden muss.
- **Test- und Simulationsaufbauten**, um einen Datenstrom parallel zu überwachen und zu verarbeiten.
- **Bus-ähnliche Architekturen**, bei denen mehrere Verbraucher identische Adapterinformationen erhalten sollen.

Vergleich mit ähnlichen Bausteinen

Andere Splitt-Bausteine wie `AD_SPLIT_2`, `AD_SPLIT_4` oder `AD_SPLIT_8` unterscheiden sich lediglich in der Anzahl der Ausgänge. `AD_SPLIT_6` bietet mit sechs Ausgängen einen Mittelweg zwischen den varianten. Es existieren auch Bausteine mit zusätzlicher Logik (z.B. bedingte Verteilung), die jedoch hier nicht vorliegt.

- `AD_SPLIT_6_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

`AD_SPLIT_6` ist ein einfacher, aber nützlicher generischer Funktionsblock zur 1:6-Verteilung eines AD-Adapters. Seine klare Schnittstelle und das Fehlen von Zustandslogik machen ihn leicht verständlich und effizient für die Verteilungsaufgaben in IEC 61499-basierten Anwendungen.

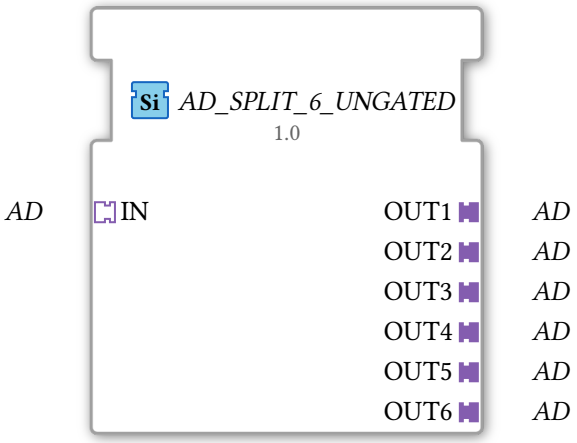
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AD_SPLIT_6_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AD_SPLIT_6`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weiter-

gegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AD_SPLIT_6_UNGATED** dient der Verteilung eines eingehenden Adapters vom Typ `adapter::types::unidirectional::AD` auf sechs separate Ausgangsadapter desselben Typs. Er ist als generischer Baustein konzipiert, der eine einfache 1:6-Aufteilung ohne zusätzliche Logik oder Zustandsverwaltung realisiert.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AD`	IN	Socket	Eingangsadapter, der auf die sechs Ausgänge verteilt wird.
`adapter::types::unidirectional::AD`	OUT1	Plug	Erster Ausgangsadapter (identisch zu IN).
`adapter::types::unidirectional::AD`	OUT2	Plug	Zweiter Ausgangsadapter.
`adapter::types::unidirectional::AD`	OUT3	Plug	Dritter Ausgangsadapter.
`adapter::types::unidirectional::AD`	OUT4	Plug	Vierter Ausgangsadapter.
`adapter::types::unidirectional::AD`	OUT5	Plug	Fünfter Ausgangsadapter.
`adapter::types::unidirectional::AD`	OUT6	Plug	Sechster Ausgangsadapter.

Funktionsweise

Der Baustein leitet den über den Socket **IN** empfangenen Adapter unverändert an alle sechs Plugs (**OUT1** bis **OUT6**) weiter. Es findet keine Datenmanipulation, Filterung oder Verzögerung statt. Der Adapter wird als Referenz oder durch direkte Kopplung (abhängig von der Laufzeitumgebung) auf die Ausgänge multipliziert. Somit erhalten alle angeschlossenen nachfolgenden Funktionsbausteine denselben Adapter-Inhalt.

Technische Besonderheiten

- **Generischer Baustein**: Das Attribut `eclipse4diac::core::GenericClassName` ist auf `'GEN_AD_SPLIT'` gesetzt, was die Wiederverwendung in verschiedenen Typkonfigurationen ermöglicht.
- **Keine Ereignisse oder Daten**: Der Baustein arbeitet ausschließlich über Adapter-Schnittstellen; Ereignisse oder direkte Datenverbindungen werden nicht benötigt.
- **Einfache Topologie**: Die 1:6-Verteilung ist fest verdrahtet und kann nicht dynamisch angepasst werden.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände oder Zustandsautomaten. Er verhält sich passiv und gibt den Eingangswert permanent an alle Ausgänge weiter. Es gibt keine zeitlichen Bedingungen oder Wechsel zwischen Modi.

Anwendungsszenarien

- **Signalverteilung** in Steuerungsanwendungen, bei denen eine AD-Adapter-Schnittstelle an mehrere nachfolgende Bausteine verteilt werden muss.
- **Test- und Simulationsaufbauten**, um einen Datenstrom parallel zu überwachen und zu verarbeiten.
- **Bus-ähnliche Architekturen**, bei denen mehrere Verbraucher identische Adapterinformationen erhalten sollen.

Vergleich mit ähnlichen Bausteinen

Andere Splitt-Bausteine wie `AD_SPLIT_2`, `AD_SPLIT_4` oder `AD_SPLIT_8` unterscheiden sich lediglich in der Anzahl der Ausgänge. `AD_SPLIT_6_UNGATED` bietet mit sechs Ausgängen einen Mittelweg zwischen den varianten. Es existieren auch Bausteine mit zusätzlicher Logik (z.B. bedingte Verteilung), die jedoch hier nicht vorliegt.

- **`AD_SPLIT_6`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

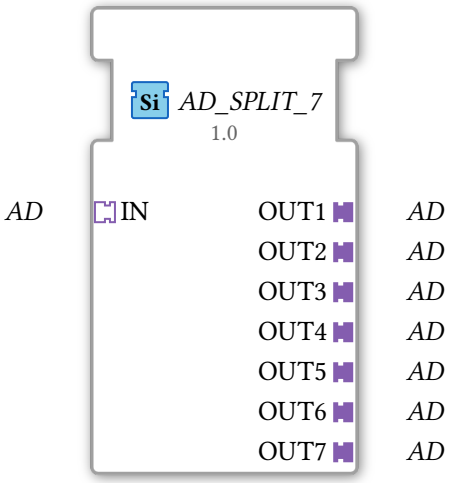
Fazit

`AD_SPLIT_6_UNGATED` ist ein einfacher, aber nützlicher generischer Funktionsblock zur 1:6-Verteilung eines AD-Adapters. Seine klare Schnittstelle und das Fehlen von Zustandslogik machen ihn leicht verständlich und effizient für die Verteilungsaufgaben in IEC 61499-basierten Anwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AD_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AD_SPLIT_7** ist ein generischer Baustein zur Aufteilung eines eingehenden Adapters vom Typ `AD` (unidirektional) auf sieben separate Ausgangsadapter. Er ermöglicht es, ein einzelnes Adaptersignal parallel an mehrere nachfolgende Komponenten weiterzuleiten, ohne die Daten zu verändern oder zu puffern. Der Baustein ist speziell für den Einsatz in der Steuerungstechnik (z. B. Landmaschinen) konzipiert und unterliegt der Eclipse Public License 2.0.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	`IN`	`adapter::types::unidirectional::AD`	Empfängt das zu verteilende Adaptersignal.
Plug (Ausgang) 1	`OUT1`	`adapter::types::unidirectional::AD`	Erster Ausgang – identische Kopie des Eingangssignals.
Plug (Ausgang) 2	`OUT2`	`adapter::types::unidirectional::AD`	Zweiter Ausgang.
Plug (Ausgang) 3	`OUT3`	`adapter::types::unidirectional::AD`	Dritter Ausgang.
Plug (Ausgang) 4	`OUT4`	`adapter::types::unidirectional::AD`	Vierter Ausgang.
Plug (Ausgang) 5	`OUT5`	`adapter::types::unidirectional::AD`	Fünfter Ausgang.
Plug (Ausgang) 6	`OUT6`	`adapter::types::unidirectional::AD`	Sechster Ausgang.
Plug (Ausgang) 7	`OUT7`	`adapter::types::unidirectional::AD`	Siebter Ausgang.

Funktionsweise

Der Baustein leitet das am Socket `IN` anliegende Adaptersignal (Typ `AD`) unverändert an alle sieben Plugs (`OUT1` … `OUT7`) weiter. Es findet keine Datenmanipulation, Synchronisation oder Pufferung statt. Die Ausgänge sind stets zeitgleich und identisch zum Eingang. Der Baustein arbeitet rein datenflussbasiert ohne Ereignisse oder Zustandslogik.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ deklariert (GenericClassName = `'GEN_AD_SPLIT'`), was eine Wiederverwendung mit verschiedenen Adapter-Varianten ermöglicht.
- **Unidirektionale Adapter**: Alle verwendeten Adapter sind unidirektional (Richtung vom Socket zu den Plugs).
- **Keine internen Zustände**: Es existiert keine Zustandsmaschine – die Ausgaben werden kontinuierlich aus der Eingabe abgeleitet.
- **Typ-Hash**: Ein leerer TypeHash (`''`) ist gesetzt, kann aber bei Bedarf für Versionskontrolle verwendet werden.

Zustandsübersicht

Der Baustein besitzt keine Zustände oder Zustandsautomaten. Das Verhalten ist rein kombinatorisch: Sobald ein gültiges Signal am Eingang anliegt, wird es sofort auf alle Ausgänge gespiegelt.

Anwendungsszenarien

- **Verteilen von Sensordaten**: Ein einziger Sensor (z. B. Drehzahl, Druck) soll mehrere unabhängige Auswerteeinheiten parallel versorgen.
- **Parallelbetrieb von Aktoren**: Ein Steuersignal wird gleichzeitig an mehrere identische Aktoren (z. B. Ventile, Motoren) weitergegeben.
- **Redundanzaufbau**: Derselbe Datenstrom wird an mehrere redundante Verarbeitungspfade gesendet.
- **Test- und Simulationsumgebungen**: Ein simuliertes Adaptersignal wird auf verschiedene Testmodule aufgeteilt.

Vergleich mit ähnlichen Bausteinen

Baustein	Anzahl Ausgänge	Besonderheit
`AD_SPLIT_2`	2	Einfache 1:2-Verteilung
`AD_SPLIT_4`	4	Typisch für 4-kanalige Systeme
`AD_SPLIT_7`	**7**	Erweiterte Verteilung für sieben parallele Pfade
`AD_MERGE`	–	Führt mehrere Eingänge zu einem Ausgang zusammen

Gegensätzlich zu einem Merge-Baustein dient `AD_SPLIT_7` ausschließlich der Vervielfältigung eines Signals ohne Datenkombination.

- **`AD_SPLIT_7_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

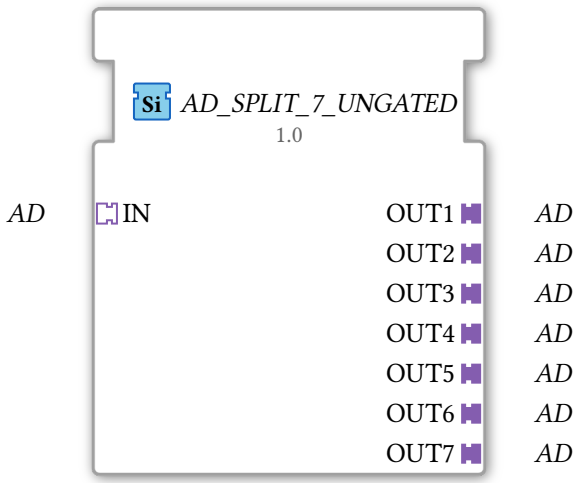
`AD_SPLIT_7` ist ein einfacher, aber nützlicher Baustein zur passiven Signalvervielfachung in adapterbasierten 4diac-Anwendungen. Seine generische Natur und die Unabhängigkeit von Ereignissen machen ihn ideal für reine Datenflussarchitekturen. Für Anwendungen, die eine höhere Anzahl paralleler Ausgänge benötigen, kann der Baustein durch Anpassung des GenericClassName erweitert werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AD_SPLIT_7_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AD_SPLIT_7`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AD_SPLIT_7_UNGATED** ist ein generischer Baustein zur Aufteilung eines eingehenden Adapters vom Typ `AD` (unidirektional) auf sieben separate Ausgangsadapter. Er ermöglicht es, ein einzelnes Adaptersignal parallel an mehrere nachfolgende Komponenten weiterzuleiten, ohne die Daten zu verändern oder zu puffern. Der Baustein ist speziell für den Einsatz in der Steuerungstechnik (z. B. Landmaschinen) konzipiert und unterliegt der Eclipse Public License 2.0.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung

Socket (Eingang)	IN	adapter::types::unidirectional::AD	Empfängt das zu verteilende Adaptersignal.
Plug (Ausgang) 1	OUT1	adapter::types::unidirectional::AD	Erster Ausgang – identische Kopie des Eingangssignals.
Plug (Ausgang) 2	OUT2	adapter::types::unidirectional::AD	Zweiter Ausgang.
Plug (Ausgang) 3	OUT3	adapter::types::unidirectional::AD	Dritter Ausgang.
Plug (Ausgang) 4	OUT4	adapter::types::unidirectional::AD	Vierter Ausgang.
Plug (Ausgang) 5	OUT5	adapter::types::unidirectional::AD	Fünfter Ausgang.
Plug (Ausgang) 6	OUT6	adapter::types::unidirectional::AD	Sechster Ausgang.
Plug (Ausgang) 7	OUT7	adapter::types::unidirectional::AD	Siebter Ausgang.

Funktionsweise

Der Baustein leitet das am Socket `IN` anliegende Adaptersignal (Typ `AD`) unverändert an alle sieben Plugs (`OUT1` … `OUT7`) weiter. Es findet keine Datenmanipulation, Synchronisation oder Pufferung statt. Die Ausgänge sind stets zeitgleich und identisch zum Eingang. Der Baustein arbeitet rein datenflussbasiert ohne Ereignisse oder Zustandslogik.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ deklariert (GenericClassName = `'GEN_AD_SPLIT'`), was eine Wiederverwendung mit verschiedenen Adapter-Varianten ermöglicht.
- **Unidirektionale Adapter**: Alle verwendeten Adapter sind unidirektional (Richtung vom Socket zu den Plugs).
- **Keine internen Zustände**: Es existiert keine Zustandsmaschine – die Ausgaben werden kontinuierlich aus der Eingabe abgeleitet.
- **Typ-Hash**: Ein leerer TypeHash (`''`) ist gesetzt, kann aber bei Bedarf für Versionskontrolle verwendet werden.

Zustandsübersicht

Der Baustein besitzt keine Zustände oder Zustandsautomaten. Das Verhalten ist rein kombinatorisch: Sobald ein gültiges Signal am Eingang anliegt, wird es sofort auf alle Ausgänge gespiegelt.

Anwendungsszenarien

- **Verteilen von Sensordaten**: Ein einziger Sensor (z. B. Drehzahl, Druck) soll mehrere unabhängige Auswerteeinheiten parallel versorgen.
- **Parallelbetrieb von Aktoren**: Ein Steuersignal wird gleichzeitig an mehrere identische Aktoren (z. B. Ventile, Motoren) weitergegeben.
- **Redundanzaufbau**: Derselbe Datenstrom wird an mehrere redundante Verarbeitungspfade gesendet.
- **Test- und Simulationsumgebungen**: Ein simuliertes Adaptersignal wird auf verschiedene Testmodule aufgeteilt.

Vergleich mit ähnlichen Bausteinen

Baustein	Anzahl Ausgänge	Besonderheit
`AD_SPLIT_2`	2	Einfache 1:2-Verteilung
`AD_SPLIT_4`	4	Typisch für 4-kanalige Systeme
`AD_SPLIT_7_UNGATED`	**7**	Erweiterte Verteilung für sieben parallele Pfade
`AD_MERGE`	–	Führt mehrere Eingänge zu einem Ausgang zusammen

Gegensätzlich zu einem Merge-Baustein dient `AD_SPLIT_7_UNGATED` ausschließlich der Vervielfältigung eines Signals ohne Datenkombination.

- **`AD_SPLIT_7_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- **`AD_SPLIT_7`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

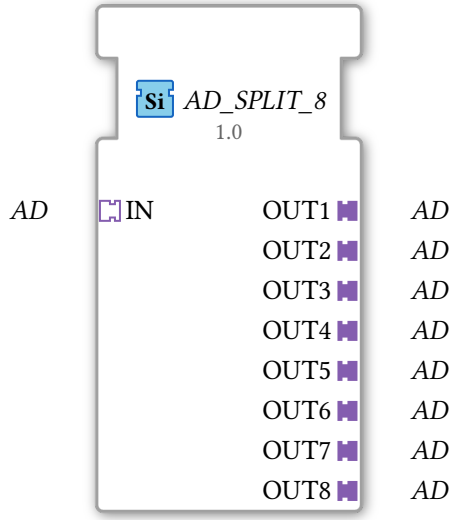
Fazit

`AD_SPLIT_7_UNGATED` ist ein einfacher, aber nützlicher Baustein zur passiven Signalvervielfachung in adapterbasierten 4diac-Anwendungen. Seine generische Natur und die Unabhängigkeit von Ereignissen machen ihn ideal für reine Datenflussarchitekturen. Für Anwendungen, die eine höhere Anzahl paralleler Ausgänge benötigen, kann der Baustein durch Anpassung des GenericClassName erweitert werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AD_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AD_SPLIT_8** ist ein generischer Baustein zur Aufteilung eines eingehenden unidirektionalen AD-Adapter-Signals auf acht separate Ausgänge. Er ermöglicht die Verteilung eines analogen oder digitalen Signals an mehrere nachfolgende Komponenten, ohne dass eine Signalveränderung stattfindet.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Richtung	Name	Typ	Beschreibung
Socket	**IN**	`adapter::types::unidirectional::AD`	Eingangsadapter – aufzuteilendes Signal
Plug	**OUT1**	`adapter::types::unidirectional::AD`	Erster Ausgang
Plug	**OUT2**	`adapter::types::unidirectional::AD`	Zweiter Ausgang
Plug	**OUT3**	`adapter::types::unidirectional::AD`	Dritter Ausgang
Plug	**OUT4**	`adapter::types::unidirectional::AD`	Vierter Ausgang
Plug	**OUT5**	`adapter::types::unidirectional::AD`	Fünfter Ausgang
Plug	**OUT6**	`adapter::types::unidirectional::AD`	Sechster Ausgang
Plug	**OUT7**	`adapter::types::unidirectional::AD`	Siebter Ausgang
Plug	**OUT8**	`adapter::types::unidirectional::AD`	Achter Ausgang

Funktionsweise

Der Baustein leitet das am Socket **IN** anliegende Signal unverändert an alle acht Plugs **OUT1** bis **OUT8** weiter. Es findet keine Signalverarbeitung, Verzögerung oder logische Änderung statt. Die Verteilung erfolgt kontinuierlich und ohne Triggerung durch Ereignisse.

Technische Besonderheiten

- **Generischer Typ**: Der Adaptertyp `adapter::types::unidirectional::AD` kann je nach Projektdefinition ein beliebiges unidirektionales Signal repräsentieren (z. B. analogen Wert, Byte, Struktur).
- **Keine Ereignissteuerung**: Der FB arbeitet rein datengetrieben (Adapterfluss) – keine INIT-, RSP- oder anderen Ereignisse erforderlich.
- **Kompatibilität**: Erforderliche Import-Deklarationen für `eclipse4diac::core::GenericClassName` und `eclipse4diac::core::TypeHash` sind im CompilerInfo enthalten.
- **Skalierbarkeit**: Durch Anpassung der generischen Parameter können ähnliche Bausteine mit anderer Ausgangsanzahl erstellt werden.

Zustandsübersicht

Der Funktionsblock besitzt keine Zustandsmaschine (ECC). Er arbeitet statisch und passt sein Ausgangsverhalten nicht an unterschiedliche Betriebsmodi an.

Anwendungsszenarien

- **Sensorverteilung**: Ein einzelner analoger Sensorwert (z. B. Temperatur) wird an mehrere Steuerungsmodule parallel weitergegeben.
- **Signalbereitstellung für Visualisierung und Regelung**: Das gleiche AD-Signal wird gleichzeitig an eine übergeordnete Steuerung und ein Bedienpanel gesendet.
- **Redundante Datenversorgung**: In sicherheitskritischen Anwendungen kann das Signal auf mehrere unabhängige Pfade aufgeteilt werden.

Vergleich mit ähnlichen Bausteinen

- **AD_SPLIT_2, AD_SPLIT_4** – Bausteine mit ähnlicher Funktion, aber geringerer Ausgangsanzahl.
- **AD_ROUTER** – Ein Baustein, der das eingehende Signal wahlweise auf einen von mehreren Ausgängen legt (Selektion).
- **AD_MULTIPLEXER** – Fasst mehrere Eingänge zu einem Ausgang zusammen (Gegenrichtung).

Im Gegensatz zu diesen Bausteinen erfolgt bei **AD_SPLIT_8** keine Auswahl – alle Ausgänge erhalten stets das gleiche Signal.

- `AD_SPLIT_8_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

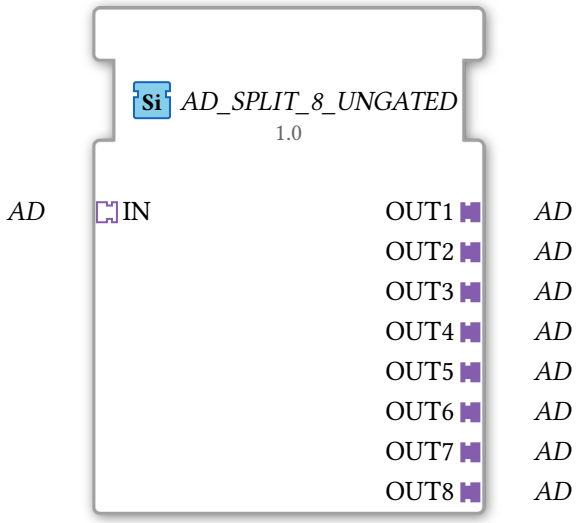
AD_SPLIT_8 ist ein einfacher, aber essenzieller Funktionsblock in IEC-61499-Systemen, der die Signalverteilung über Adapter-Schnittstellen ohne zusätzliche Logik oder Laufzeitkosten realisiert. Seine generische Natur macht ihn flexibel einsetzbar und erleichtert die modulare Strukturierung von Automatisierungsprojekten.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AD_SPLIT_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AD_SPLIT_8`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AD_SPLIT_8_UNGATED** ist ein generischer Baustein zur Aufteilung eines eingehenden unidirektionalen AD-Adapter-Signals auf acht separate Ausgänge. Er ermöglicht die Verteilung eines analogen oder digitalen Signals an mehrere nachfolgende Komponenten, ohne dass eine Signalveränderung stattfindet.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Richtung	Name	Typ	Beschreibung
Socket	**IN**	`adapter::types::unidirectional::AD`	Eingangsadapter – aufzuteilendes Signal
Plug	**OUT1**	`adapter::types::unidirectional::AD`	Erster Ausgang
Plug	**OUT2**	`adapter::types::unidirectional::AD`	Zweiter Ausgang
Plug	**OUT3**	`adapter::types::unidirectional::AD`	Dritter Ausgang
Plug	**OUT4**	`adapter::types::unidirectional::AD`	Vierter Ausgang
Plug	**OUT5**	`adapter::types::unidirectional::AD`	Fünfter Ausgang
Plug	**OUT6**	`adapter::types::unidirectional::AD`	Sechster Ausgang
Plug	**OUT7**	`adapter::types::unidirectional::AD`	Siebter Ausgang
Plug	**OUT8**	`adapter::types::unidirectional::AD`	Achter Ausgang

Funktionsweise

Der Baustein leitet das am Socket **IN** anliegende Signal unverändert an alle acht Plugs **OUT1** bis **OUT8** weiter. Es findet keine Signalverarbeitung, Verzögerung oder logische Änderung statt. Die Verteilung erfolgt kontinuierlich und ohne Triggerung durch Ereignisse.

Technische Besonderheiten

- **Generischer Typ**: Der Adaptertyp `adapter::types::unidirectional::AD` kann je nach Projektdefinition ein beliebiges unidirektionales Signal repräsentieren (z. B. analogen Wert, Byte, Struktur).
- **Keine Ereignissteuerung**: Der FB arbeitet rein datengetrieben (Adapterfluss) – keine INIT-, RSP- oder anderen Ereignisse erforderlich.
- **Kompatibilität**: Erforderliche Import-Deklarationen für `eclipse4diac::core::GenericClassName` und `eclipse4diac::core::TypeHash` sind im CompilerInfo enthalten.
- **Skalierbarkeit**: Durch Anpassung der generischen Parameter können ähnliche Bausteine mit anderer Ausgangsanzahl erstellt werden.

Zustandsübersicht

Der Funktionsblock besitzt keine Zustandsmaschine (ECC). Er arbeitet statisch und passt sein Ausgangsverhalten nicht an unterschiedliche Betriebsmodi an.

Anwendungsszenarien

- **Sensorverteilung**: Ein einzelner analoger Sensorwert (z. B. Temperatur) wird an mehrere Steuerungsmodule parallel weitergegeben.
- **Signalbereitstellung für Visualisierung und Regelung**: Das gleiche AD-Signal wird gleichzeitig an eine übergeordnete Steuerung und ein Bedienpanel gesendet.
- **Redundante Datenversorgung**: In sicherheitskritischen Anwendungen kann das Signal auf mehrere unabhängige Pfade aufgeteilt werden.

Vergleich mit ähnlichen Bausteinen

- **AD_SPLIT_2, AD_SPLIT_4** – Bausteine mit ähnlicher Funktion, aber geringerer Ausgangsanzahl.
- **AD_ROUTER** – Ein Baustein, der das eingehende Signal wahlweise auf einen von mehreren Ausgängen legt (Selektion).
- **AD_MULTIPLEXER** – Fasst mehrere Eingänge zu einem Ausgang zusammen (Gegenrichtung).

Im Gegensatz zu diesen Bausteinen erfolgt bei **AD_SPLIT_8_UNGATED** keine Auswahl – alle Ausgänge erhalten stets das gleiche Signal.

- `AD_SPLIT_8`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

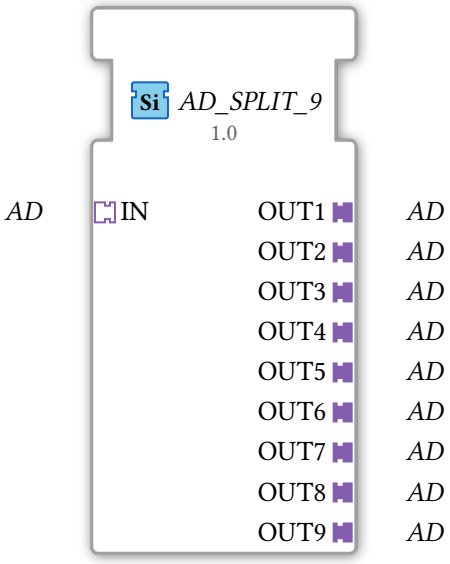
Fazit

AD_SPLIT_8_UNGATED ist ein einfacher, aber essenzieller Funktionsblock in IEC-61499-Systemen, der die Signalverteilung über Adapter-Schnittstellen ohne zusätzliche Logik oder Laufzeitkosten realisiert. Seine generische Natur macht ihn flexibel einsetzbar und erleichtert die modulare Strukturierung von Automatisierungsprojekten.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AD_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AD_SPLIT_9** verteilt einen eingehenden unidirektionalen Adapter (AD) auf neun identische Ausgänge. Er ist als generischer FB realisiert und eignet sich zur Mehrfachnutzung eines Adaptersignals in verschiedenen nachgelagerten Komponenten.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Name	Typ	Richtung
IN	adapter::types::unidirectional::AD	Socket (Eingang)
OUT1	adapter::types::unidirectional::AD	Plug (Ausgang)
OUT2	adapter::types::unidirectional::AD	Plug (Ausgang)
OUT3	adapter::types::unidirectional::AD	Plug (Ausgang)
OUT4	adapter::types::unidirectional::AD	Plug (Ausgang)
OUT5	adapter::types::unidirectional::AD	Plug (Ausgang)
OUT6	adapter::types::unidirectional::AD	Plug (Ausgang)
OUT7	adapter::types::unidirectional::AD	Plug (Ausgang)
OUT8	adapter::types::unidirectional::AD	Plug (Ausgang)
OUT9	adapter::types::unidirectional::AD	Plug (Ausgang)

Funktionsweise

Der FB führt eine **1:n-Verteilung** des eingehenden Adaptersignals durch. Jeder der neun Ausgangsadapter (OUT1 … OUT9) stellt exakt die gleichen Daten wie der Eingangsadapter (IN) bereit. Da der FB weder Ereignis- noch Dateneingänge besitzt, erfolgt die Signalweitergabe rein strukturell: die Verbindung zwischen IN und allen OUTs wird zur Konfigurationszeit festgelegt und zur Laufzeit transparent weitergeleitet.

Technische Besonderheiten

- **Generischer Baustein**: Der FB trägt das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `'GEN_AD_SPLIT'`, wodurch er als Platzhalter für eine typspezifische Instantiierung dient.
- **Typbindung**: Alle Adapter sind vom Typ `adapter::types::unidirectional::AD` – einer standardisierten unidirektionalen Adapterschnittstelle.
- **Keine Ereignissteuerung**: Die Verteilung erfolgt ohne zeitliche Synchronisation oder Ereignisse; Änderungen am Eingangssignal werden sofort an alle Ausgänge weitergegeben.

Zustandsübersicht

Der FB besitzt keinen internen Zustandsautomaten und führt keinerlei sequentielle Logik aus. Das Verhalten ist **rein kombinatorisch**.

Anwendungsszenarien

- **Signalmultiplex**: Ein von einem Sensor oder einer Steuerung bereitgestellter Adapter soll an mehrere unabhängige Auswerteeinheiten (z. B. Visualisierung, Protokollierung, Alarmgenerierung) angebunden werden.
- **Modulare Architektur**: In einer IEC 61499-basierten Anwendung kann ein zentraler Datenprovider seine Informationen über `AD_SPLIT_9` an bis zu neun verschiedene Submodule verteilen, ohne die Quelllogik zu vervielfachen.

Vergleich mit ähnlichen Bausteinen

- **AD_SPLIT_2 / AD_SPLIT_4 / AD_SPLIT_8**: Diese Bausteine bieten die gleiche Funktionalität, jedoch mit einer geringeren Anzahl von Ausgängen (2, 4, 8). `AD_SPLIT_9` ergänzt das Portfolio für Anwendungen, die genau neun parallele Ausgänge benötigen.
- **Andere Split-Bausteine**: Datentyp-orientierte Split-FBs (z. B. für INTEGER oder BOOL) teilen einzelne Datenwerte auf, während `AD_SPLIT_9` vollständige Adapterstrukturen vervielfältigt. Der Adapter kann dabei mehrere zusammengehörige Daten und Ereignisse bündeln.

- `AD_SPLIT_9_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

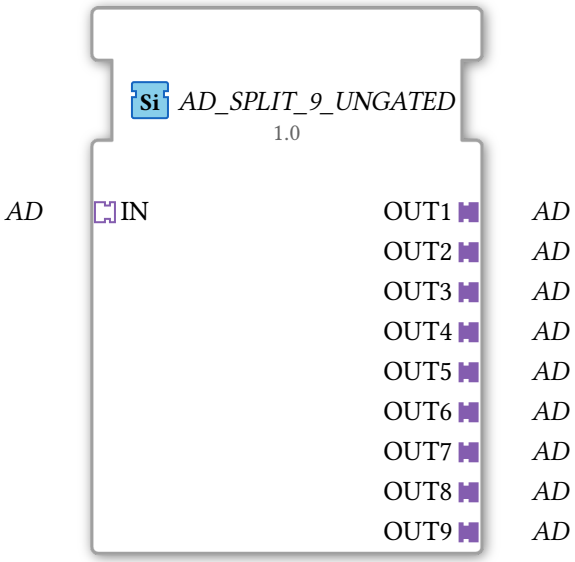
`AD_SPLIT_9` ist ein einfacher, aber nützlicher generischer Funktionsblock zur unidirektionalen Aufteilung eines Adaptersignals auf neun identische Ausgänge. Er hilft, Redundanz im Systemdesign zu vermeiden und erleichtert die modulare Strukturierung von IEC 61499-Anwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AD_SPLIT_9_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AD_SPLIT_9`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AD_SPLIT_9_UNGATED** verteilt einen eingehenden unidirektionalen Adapter (AD) auf neun identische Ausgänge. Er ist als generischer FB realisiert und eignet sich zur Mehrfachnutzung eines Adaptersignals in verschiedenen nachgelagerten Komponenten.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Name	Typ	Richtung
IN	adapter::types::unidirectional::AD	Socket (Eingang)
OUT1	adapter::types::unidirectional::AD	Plug (Ausgang)
OUT2	adapter::types::unidirectional::AD	Plug (Ausgang)
OUT3	adapter::types::unidirectional::AD	Plug (Ausgang)
OUT4	adapter::types::unidirectional::AD	Plug (Ausgang)
OUT5	adapter::types::unidirectional::AD	Plug (Ausgang)
OUT6	adapter::types::unidirectional::AD	Plug (Ausgang)
OUT7	adapter::types::unidirectional::AD	Plug (Ausgang)
OUT8	adapter::types::unidirectional::AD	Plug (Ausgang)
OUT9	adapter::types::unidirectional::AD	Plug (Ausgang)

Funktionsweise

Der FB führt eine **1:n-Verteilung** des eingehenden Adaptersignals durch. Jeder der neun Ausgangsadapter (OUT1 … OUT9) stellt exakt die gleichen Daten wie der Eingangsadapter (IN) bereit. Da der FB weder Ereignis- noch Dateneingänge besitzt, erfolgt die Signalweitergabe rein strukturell: die Verbindung zwischen IN und allen OUTs wird zur Konfigurationszeit festgelegt und zur Laufzeit transparent weitergeleitet.

Technische Besonderheiten

- **Generischer Baustein**: Der FB trägt das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `'GEN_AD_SPLIT'`, wodurch er als Platzhalter für eine typspezifische Instantiierung dient.
- **Typbindung**: Alle Adapter sind vom Typ `adapter::types::unidirectional::AD` – einer standardisierten unidirektionalen Adapterschnittstelle.
- **Keine Ereignissteuerung**: Die Verteilung erfolgt ohne zeitliche Synchronisation oder Ereignisse; Änderungen am Eingangssignal werden sofort an alle Ausgänge weitergegeben.

Zustandsübersicht

Der FB besitzt keinen internen Zustandsautomaten und führt keinerlei sequentielle Logik aus. Das Verhalten ist **rein kombinatorisch**.

Anwendungsszenarien

- **Signalmultiplex**: Ein von einem Sensor oder einer Steuerung bereitgestellter Adapter soll an mehrere unabhängige Auswerteeinheiten (z. B. Visualisierung, Protokollierung, Alarmgenerierung) angebunden werden.
- **Modulare Architektur**: In einer IEC 61499-basierten Anwendung kann ein zentraler Datenprovider seine Informationen über `AD_SPLIT_9_UNGATED` an bis zu neun verschiedene Submodule verteilen, ohne die Quelllogik zu vervielfachen.

Vergleich mit ähnlichen Bausteinen

- **AD_SPLIT_2 / AD_SPLIT_4 / AD_SPLIT_8**: Diese Bausteine bieten die gleiche Funktionalität, jedoch mit einer geringeren Anzahl von Ausgängen (2, 4, 8). `AD_SPLIT_9_UNGATED` ergänzt das Portfolio für Anwendungen, die genau neun parallele Ausgänge benötigen.
- **Andere Split-Bausteine**: Datentyp-orientierte Split-FBs (z. B. für INTEGER oder BOOL) teilen einzelne Datenwerte auf, während `AD_SPLIT_9_UNGATED` vollständige Adapterstrukturen vervielfältigt. Der Adapter kann dabei mehrere zusammengehörige Daten und Ereignisse bündeln.
- **`AD_SPLIT_9`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

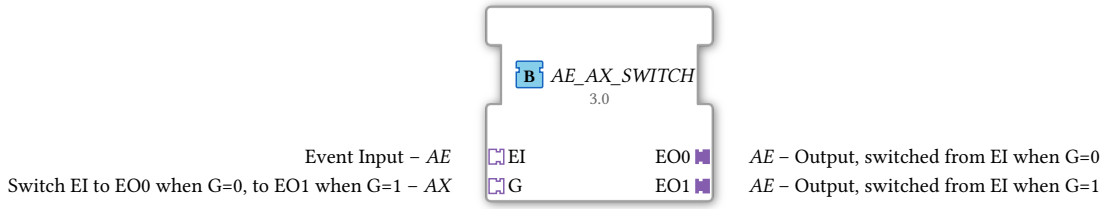
`AD_SPLIT_9_UNGATED` ist ein einfacher, aber nützlicher generischer Funktionsblock zur unidirektionalen Aufteilung eines Adaptersignals auf neun identische Ausgänge. Er hilft, Redundanz im Systemdesign zu vermeiden und erleichtert die modulare Strukturierung von IEC 61499-Anwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

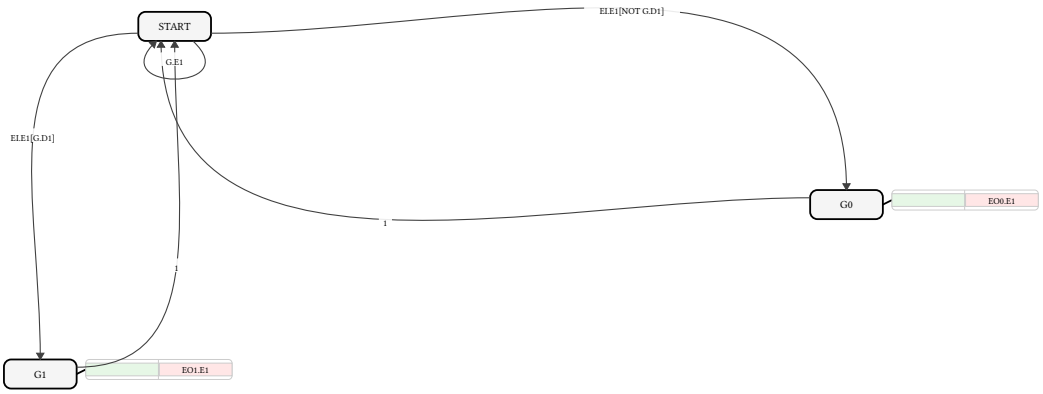
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

EVENT

AE_AX_SWITCH



ECC: AE_AX_SWITCH



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AE_AX_SWITCH** realisiert ein ereignisgesteuertes Umschalten (Demultiplexing). Ein eingehendes Ereignis wird an einen von zwei Ereignisausgängen weitergeleitet – abhängig vom booleschen Wert eines separaten Dateneingangs.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine direkten Ereignis-Eingänge. Ereignisse werden über den **Adapter `EI`** (Typ `adapter::types::unidirectional::AE`) empfangen.

Ereignis-Ausgänge

Der FB besitzt keine direkten Ereignis-Ausgänge. Ereignisse werden über die **Adapter `EO0` und `EO1`** (beide Typ `adapter::types::unidirectional::AE`) ausgegeben.

Daten-Eingänge

- **`G`** (Adapter Typ `adapter::types::unidirectional::AX`) – Boolescher Wert, der die Umschaltung steuert.
  - `G=0` ⟶ Ereignis wird an `EO0` geleitet
  - `G=1` ⟶ Ereignis wird an `EO1` geleitet

Daten-Ausgänge

Keine direkten Daten-Ausgänge. Die Ausgabe erfolgt ausschließlich über die Ereignis-Adapter.

Adapter

Name	Typ	Richtung	Beschreibung
`EI`	`AE` (Ereignis-Adapter)	Socket	Empfängt das zu schaltende Ereignis.
`G`	`AX` (Daten-Adapter)	Socket	Liefert den Schaltzustand (boolesch).
`EO0`	`AE`	Plug	Gibt das Ereignis aus, wenn `G=0` war.
`EO1`	`AE`	Plug	Gibt das Ereignis aus, wenn `G=1` war.

Funktionsweise

Der FB befindet sich stets im Zustand `START`. Sobald am Adapter `EI` ein Ereignis eintrifft (Signal `EI.E1`), wird der aktuelle Wert des Bool-Adapter `G.D1` ausgewertet:

- **`G.D1 = 0`** ⟶ Der FB wechselt in den Zustand `G0` und sendet ein Ereignis über den Adapter `EO0.E1`. Danach kehrt er sofort nach `START` zurück.
- **`G.D1 = 1`** ⟶ Der FB wechselt in den Zustand `G1` und sendet ein Ereignis über den Adapter `EO1.E1`. Danach kehrt er sofort nach `START` zurück.

Ein zusätzliches Ereignis am Adapter `G` (Signal `G.E1`) führt zu keiner Aktion – der FB verbleibt im Zustand `START`. Dies verhindert unerwünschte Schaltvorgänge bei Änderungen des Datenwerts.

Technische Besonderheiten

- **Adapter-basierte Schnittstelle** – Der FB nutzt lose gekoppelte Adapter der Typen `AE` (Ereignis) und `AX` (Daten). Dies ermöglicht eine flexible Verbindung mit anderen Bausteinen im 4diac-Framework.
- **Deterministische Umschaltung** – Die Entscheidung, welcher Ausgang aktiviert wird, fällt im selben Schritt wie der Empfang des Ereignisses, wodurch keine zusätzliche Verzögerung entsteht.
- **Kein Speicherverhalten** – Nach der Ausgabe kehrt der FB sofort in den Grundzustand zurück. Der Zustand `G` wird nicht zwischengespeichert – er wird bei jedem Ereignis neu gelesen.

Zustandsübersicht

Zustand	Beschreibung	Ausgabeaktion
`START`	Wartet auf ein Ereignis an `EI`.	–

G0	Ereignis von EI bei G.D1=0 verarbeitet.	EO0.E1
G1	Ereignis von EI bei G.D1=1 verarbeitet.	EO1.E1

Die Zustände G0 und G1 werden unmittelbar nach der Ausgabe wieder verlassen (automatische Transition nach START).

Anwendungsszenarien

- **Ereignis-Routing** – Weiterleitung eines Ereignisses an verschiedene nachgeschaltete Logikzweige, abhängig von einer Bedingung.
- **Zustandsabhängige Steuerung** – In Automatisierungssystemen kann ein Sensorereignis je nach Betriebsart an unterschiedliche Verarbeitungsblöcke gesendet werden.
- **Multiplexing-Umkehrung** – Als Demultiplexer für asynchrone Ereignisströme, z.B. in Kommunikationsprotokollen.

Vergleich mit ähnlichen Bausteinen

- **E_SWITCH** – Der Standard-FB E_SWITCH arbeitet mit direkten Ereignis-Ein-/Ausgängen. Der AE_AX_SWITCH erweitert dieses Konzept durch Adapter, wodurch eine losere Kopplung und Wiederverwendbarkeit in verschiedenen Kontexten ermöglicht wird.
- **E_DEMUX** – Andere Demultiplexer-Bausteine verwenden oft mehrere Eingänge oder Zustände. Der AE_AX_SWITCH fokussiert auf die einfache, boolesche Umschaltung eines einzelnen Ereignisses – ohne Pufferung oder Verzögerung.

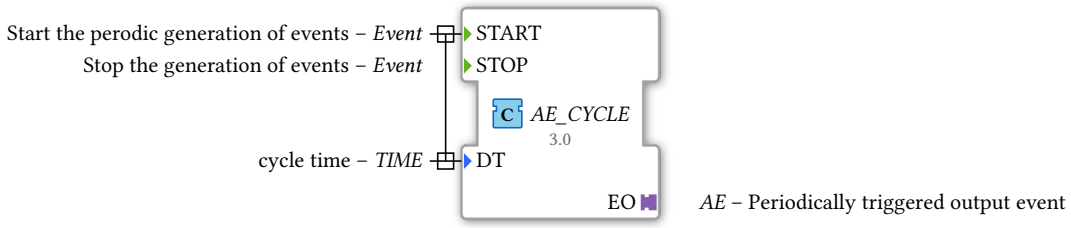
Fazit

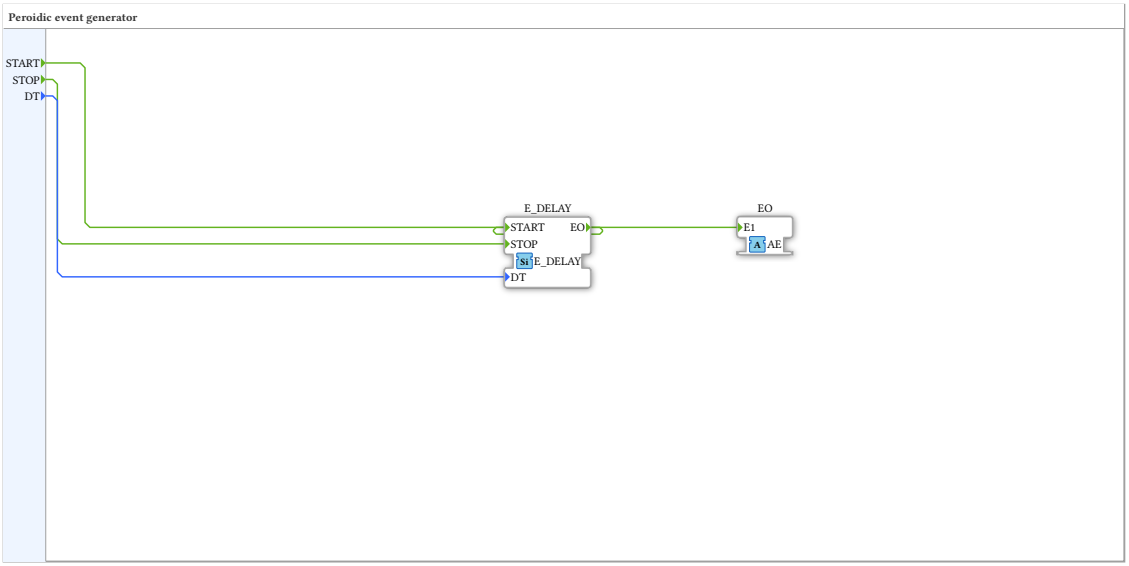
Der AE_AX_SWITCH ist ein kompakter, effizienter Demultiplexer für Ereignisse auf Basis eines booleschen Schaltsignals. Seine adapterbasierte Schnittstelle erlaubt eine flexible Integration in IEC 61499-Systeme, während das deterministische Verhalten eine zuverlässige Weiterleitung gewährleistet. Er eignet sich besonders für Anwendungen, in denen ein Ereignis je nach Zustand einer Bedingung an unterschiedliche Verarbeitungspfade gesendet werden muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AE_CYCLE





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AE_CYCLE** (Adapter Event Cycle) dient als periodischer Ereignisgenerator. Er erzeugt in regelmäßigen Abständen, definiert durch eine Zeitvorgabe, ein Ausgangsereignis. Im Gegensatz zum Standard-`E_CYCLE`-Baustein nutzt dieser FB einen Adapter für die Signalausgabe, was ihn besonders für architektonische Muster geeignet macht, die auf Event-Adaptern basieren.

Schnittstellenstruktur

Ereignis-Eingänge

- **START**: Startet die periodische Erzeugung von Ereignissen. Beim Auslösen dieses Ereignisses wird der Zeitgeber aktiviert.
- **STOP**: Stoppt die Erzeugung von Ereignissen und unterbricht den laufenden Zyklus.

Ereignis-Ausgänge

- *Keine direkten Ereignis-Ausgänge vorhanden (siehe Adapter).*

Daten-Eingänge

- **DT** (TIME): Die Zykluszeit (Cycle Time). Dieser Wert bestimmt das Intervall zwischen zwei aufeinanderfolgenden Ereignissen.

Daten-Ausgänge

- *Keine Daten-Ausgänge vorhanden.*

Adapter

- **EO** (Plug): Ein Adapter vom Typ `adapter::types::unidirectional::AE`. Über diesen Adapter wird das periodisch erzeugte Ereignis ausgegeben (konkret über das Adapter-Event `E1`).

Funktionsweise

Der **AE_CYCLE** ist als Composite Function Block (zusammengesetzter Funktionsbaustein) realisiert. Intern basiert seine Logik auf dem Standardbaustein `E_DELAY`.

1. **Starten**: Wenn das `START`-Ereignis eintrifft, wird der interne `E_DELAY`-Baustein mit der Zeitdauer `DT` gestartet.
2. **Verzögerung & Auslösen**: Nach Ablauf der Zeit `DT` erzeugt der interne Baustein ein Ausgangsereignis.
3. **Rückkopplung (Loop)**: Dieses interne Ereignis erfüllt zwei Aufgaben:
   - Es wird an den Adapter **EO** weitergeleitet, um das Signal nach außen zu geben.
   - Es wird direkt auf den `START`-Eingang des internen `E_DELAY` zurückgeführt.

4. **Zyklus**: Durch diese Rückkopplung startet der Timer sofort neu, wodurch eine kontinuierliche Schleife entsteht.
5. **Stoppen**: Das `STOP`-Ereignis unterbricht den internen `E_DELAY`-Baustein sofort und beendet die Schleife.

Technische Besonderheiten

- **Kapselung**: Der Baustein kapselt die Logik einer Oszillatorschaltung (Feedback-Loop) und stellt sie über eine vereinfachte Schnittstelle bereit.
- **Adapter-Nutzung**: Die Verwendung des `AE`-Adapters (Adapter Event) ermöglicht eine saubere Trennung und Gruppierung von Ereignisflüssen, passend zum `adapter::events::unidirectional` Paket.

Zustandsübersicht

Da es sich um einen Composite Function Block handelt, besitzt er keine eigene Zustandsmaschine (ECC). Der implizite Zustand wird durch den internen `E_DELAY` bestimmt:

- **Inaktiv**: Nach Initialisierung oder `STOP`.
- **Laufend**: Nach `START`, während der Timer läuft oder neu getriggert wird.

Anwendungsszenarien

- **Heartbeat-Signale**: Erzeugung eines Lebenszeichens für übergeordnete Systeme oder Watchdogs.
- **Polling**: Periodisches Abfragen von Sensordaten (Trigger für `READ`-Operationen).
- **Blinken**: Ansteuerung von Signalleuchten (in Kombination mit Toggle-Logik).
- **Zeitgesteuerte Abläufe**: Takterzeugung für Schrittketten oder periodische Berechnungen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **E_CYCLE**: Dies ist das direkte Standard-Pendant aus der IEC 61499 Bibliothek. `E_CYCLE` bietet die gleiche Funktionalität, nutzt jedoch einen direkten `EO`-Event-Ausgang statt eines Adapters.
- **E_DELAY**: `AE_CYCLE` basiert auf `E_DELAY`. Während `E_DELAY` ein Ereignis nur einmalig verzögert weiterleitet, sorgt `AE_CYCLE` durch die interne Verschaltung für eine endlose Wiederholung.
- **E_TRAIN**: Erzeugt eine begrenzte Anzahl von Impulsen (Pulse Train), während `AE_CYCLE` unendlich läuft, bis er gestoppt wird.

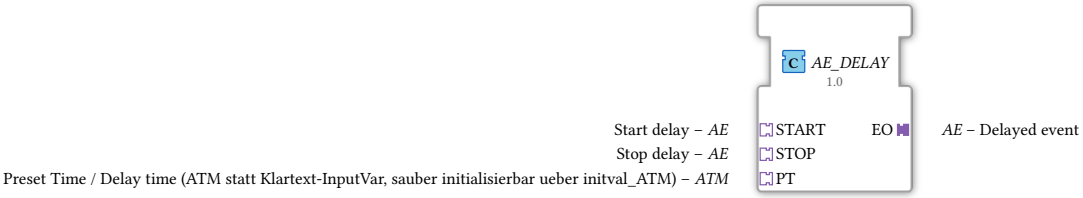
Fazit

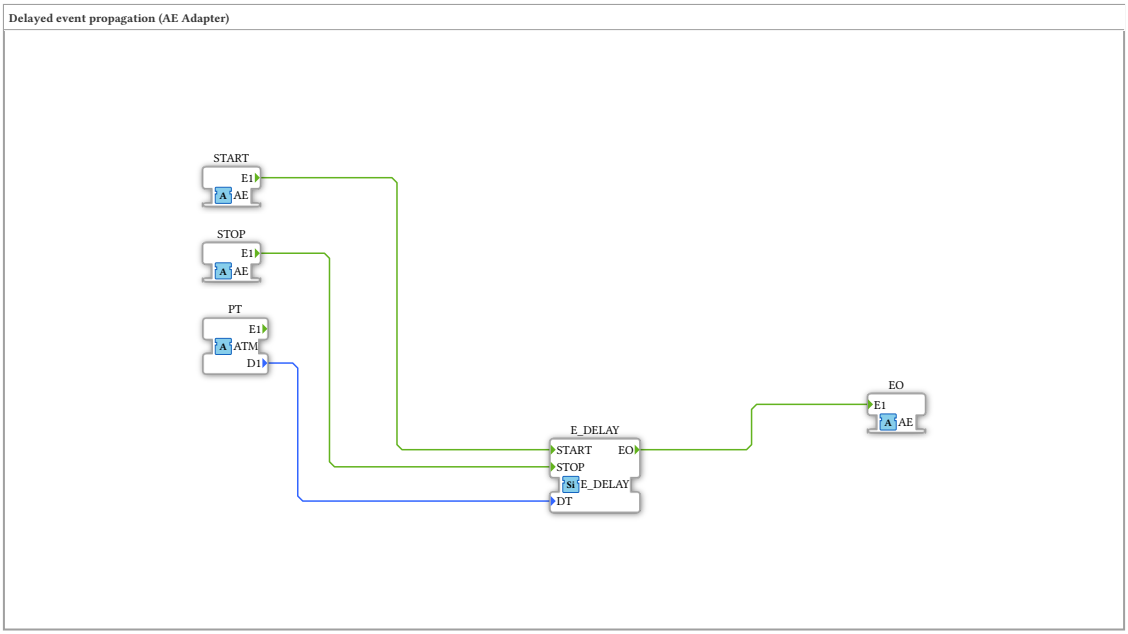
Der **AE_CYCLE** ist ein nützlicher Utility-Baustein für Entwickler, die innerhalb der 4diac-Umgebung mit Event-Adaptern arbeiten. Er abstrahiert die notwendige Rückkopplungslogik zur Erzeugung periodischer Signale und integriert sich nahtlos in Adapter-basierte Softwarearchitekturen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AE_DELAY





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **AE_DELAY** ist ein Wrapper für den Standard-IEC-61499-Baustein `E_DELAY`, der speziell für die Verwendung mit **Event-Adaptern (AE)** konzipiert wurde. Er ermöglicht die verzögerte Weiterleitung von Ereignissen innerhalb einer adapterbasierten Architektur. Anstatt herkömmlicher Ereignis-Pins verwendet dieser Baustein Adapter-Sockets und -Plugs, um Start- und Stopp-Signale zu empfangen und das verzögerte Ereignis auszugeben.

Schnittstellenstruktur

Der Baustein kapselt die Funktionalität eines Einschaltverzögerungs-Timers (On-Delay Timer) in Adapter-Schnittstellen.

Ereignis-Eingänge

Dieser Baustein verfügt über keine direkten Ereignis-Eingänge. Alle Eingangsereignisse werden über die Adapter `START` und `STOP` geleitet.

Ereignis-Ausgänge

Dieser Baustein verfügt über keine direkten Ereignis-Ausgänge auf der Hauptebene. Alle Ausgangsereignisse werden über den Adapter `EO` geleitet.

Daten-Eingänge

Dieser Baustein verfügt über keine direkten Daten-Eingänge. Die Verzögerungszeit wird über den Adapter `PT` bereitgestellt.

Daten-Ausgänge

Dieser Baustein verfügt über keine Daten-Ausgänge.

Adapter

Name	Typ	Art	Kommentar
START	adapter::types::unidirectional::AE	Socket (Eingang)	Startet die Zeitverzögerung (löst intern `START` aus).
STOP	adapter::types::unidirectional::AE	Socket (Eingang)	Stoppt/bricht die Zeitverzögerung ab (löst intern `STOP` aus).

PT	adapter::types::unidirectional::ATM	Socket (Eingang)	**Preset Time**: Verzögerungszeit (`PT.D1`, TIME).
EO	adapter::types::unidirectional::AE	Plug (Ausgang)	**Event Output**: Gibt das Ereignis nach Ablauf der Zeit `PT` aus.

Funktionsweise

Der **AE_DELAY** Baustein agiert als Brücke zwischen der Adapter-Welt und dem klassischen `E_DELAY` Timer:

1. **Initialisierung:** Über den Adapter-Socket `PT` (Datenwert `PT.D1`) wird die gewünschte Verzögerungszeit festgelegt.
2. **Starten des Timers:** Trifft ein Ereignis (üblicherweise `E1`) am Adapter-Socket **START** ein, wird dieses intern an den `START`-Eingang des eingebetteten `E_DELAY` Bausteins weitergeleitet. Der Timer beginnt zu laufen.
3. **Ablauf der Zeit:** Sobald die Zeit `PT` verstrichen ist, generiert der interne Timer ein Ereignis. Dieses wird an den Adapter-Plug **EO** (Ereignis `E1`) weitergeleitet.
4. **Abbrechen:** Trifft ein Ereignis am Adapter-Socket **STOP** ein, wird der laufende Timer sofort unterbrochen und zurückgesetzt. Es wird kein Ereignis am Ausgang `EO` generiert.

Technische Besonderheiten

- **Paket-Zugehörigkeit:** Der Baustein ist Teil des Pakets `adapter::events::unidirectional::timers`.
- **Kapselung:** Er enthält eine Instanz von `iec61499::events::E_DELAY`.
- **Adapter-Typ:** Verwendet den unidirektionalen Ereignis-Adapter `AE` (`adapter::types::unidirectional::AE`), der typischerweise ein einzelnes Ereignis (`E1`) transportiert.

Zustandsübersicht

Der Baustein selbst besitzt keine explizite Zustandsmaschine (ECC), da es sich um ein zusammengesetztes Netzwerk (Composite FB) handelt. Das Verhalten wird vollständig durch den internen `E_DELAY` bestimmt:

- **Idle:** Warten auf ein Ereignis am `START`-Adapter.
- **Timing:** Nach Empfang eines Signals an `START` läuft die Zeit `PT`.
- **Triggered:** Nach Ablauf von `PT` wird das Signal an `EO` gesendet und der Baustein kehrt in den Idle-Zustand zurück.
- **Reset:** Ein Signal an `STOP` während der Phase *Timing* setzt den Baustein sofort in den Idle-Zustand zurück.

Anwendungsszenarien

- **Adapter-basierte Steuerungen:** Ideal für Systeme, die stark auf Adapter-Verbindungen setzen, um den „Spaghetti-Code“ von Einzelverbindungen zu reduzieren.
- **Verzögerte Signalketten:** Einfügen von Wartezeiten in einer Ereigniskette, die logisch durch Adapter gruppiert ist.
- **Prozesssteuerung:** Verzögertes Starten von Aggregaten oder Prozessen, die über standardisierte AE-Schnittstellen angesteuert werden.

⚖️ Vergleich mit ähnlichen Bausteinen

Baustein	Unterschied
E_DELAY	Der Standard-Baustein mit direkten Event-Pins (START, STOP, EO). **AE_DELAY** ist der Wrapper hierfür.
E_CYCLE	Erzeugt periodische Ereignisse, während **AE_DELAY** ein Ereignis nur einmalig verzögert weiterleitet.
AE_SPLIT	Dient zum Aufteilen von Adapter-Ereignissen, bietet aber keine Zeitverzögerung.

Fazit

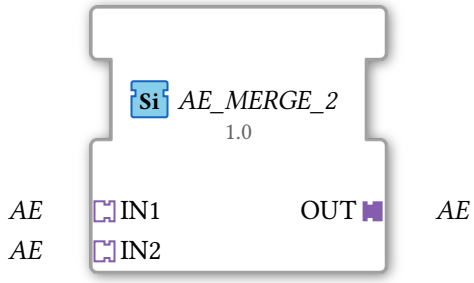
Der **AE_DELAY** ist ein unverzichtbarer Hilfsbaustein für Entwickler, die die Vorteile von 4diac-Adaptern (reduzierte Verbindungskomplexität, klare Schnittstellendefinitionen) nutzen möchten, ohne auf grund-

legende Zeitfunktionen wie Verzögerungen verzichten zu müssen. Er integriert den bewährten `E_DELAY` nahtlos in moderne, adapterbasierte Applikationsdesigns.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AE_MERGE_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AE_MERGE_2** führt zwei unidirektionale AE-Adapter (reines Ereignis, keine Nutzdaten) auf einen gemeinsamen Ausgangsadapter zusammen. Er ist die Umkehrung von AE_SPLIT_2: Statt ein Ereignis auf zwei Ausgänge zu verteilen, führt er zwei unabhängige Ereignisquellen auf einen gemeinsamen Ausgang. Der Baustein ist als generischer FB implementiert (`GenericClassName: 'GEN_AE_MERGE'`).

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignisübernahme erfolgt ausschließlich über die Adapter-Sockets.

Ereignis-Ausgänge

Keine. Die Ereignisweitergabe erfolgt ausschließlich über den Adapter-Plug.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket (Eingang 1)	`IN1`	`adapter::types::unidirectional::AE`	Erste Ereignisquelle.
Socket (Eingang 2)	`IN2`	`adapter::types::unidirectional::AE`	Zweite Ereignisquelle.
Plug (Ausgang)	`OUT`	`adapter::types::unidirectional::AE`	Zusammengeführtes Ereignis.

Funktionsweise

Trifft ein Ereignis an `IN1` **oder** `IN2` ein, wird es unverändert an `OUT` weitergereicht. Da `AE` keinen Datenwert transportiert, gibt es beim Zusammenführen nichts zu arbitrieren außer dem Ereignis selbst – jedes einzelne eingehende Ereignis erscheint unmittelbar am Ausgang, unabhängig davon, von welchem Socket es kam. Die Reihenfolge, in der beide Sockets auf ein gleichzeitiges Eintreffen geprüft werden, ist implementierungsseitig `IN1` vor `IN2`, was bei zwei innerhalb desselben Ausführungszyklus eintreffenden Ereignissen relevant werden kann.

Technische Besonderheiten

- **Generischer Typ**: Implementiert über die generische Basisklasse `CGenUnidirectMergeBase` (N Sockets, 1 Plug) – dieselbe C++-Basis, die auch `GEN_ASR_MERGE` und `GEN_ASRT_MERGE` zugrunde liegt. Die Anzahl der Eingangs-Sockets wird über den generischen Suffix (`_2`) zur Übersetzungszeit festgelegt.
- **Keine Zustände / Algorithmen**: Da `AE` keinen Datenwert trägt, gibt es keine Änderungserkennung und kein ECC – der Baustein ist reine Ereignisweiterleitung.
- **Kein Datenverlust bei gleichzeitigen Ereignissen**: Anders als eine reine Datenverbindung (die nur eine Quelle erlaubt) kann jedes Ereignis von `IN1` und `IN2` unabhängig durchgereicht werden; es geht kein Ereignis „verloren“, nur weil das jeweils andere Socket kurz zuvor gefeuert hat.

Zustandsübersicht

Der Baustein besitzt **keinen Zustandsautomaten**. Das Verhalten ist rein kombinatorisch: Jedes eingehende Ereignis an `IN1` oder `IN2` wird unmittelbar an `OUT` weitergereicht.

Anwendungsszenarien

- **Ereigniszusammenführung**: Zwei unabhängige Ereignisquellen (z. B. zwei Taster oder zwei Sensor-Trigger) sollen dieselbe Folgeaktion auslösen, ohne dass die nachgeschaltete Logik zweimal verdrahtet werden muss.
- **Redundante Auslösepfade**: Ein Ereignis kann über zwei unabhängige Quellen ausgelöst werden (z. B. ein manueller und ein automatischer Pfad), die beide auf denselben Ausgang münden.
- **Vereinfachung von Netzwerken**, die sonst zwei separate Ereignisverbindungen zum selben Ziel benötigen würden.

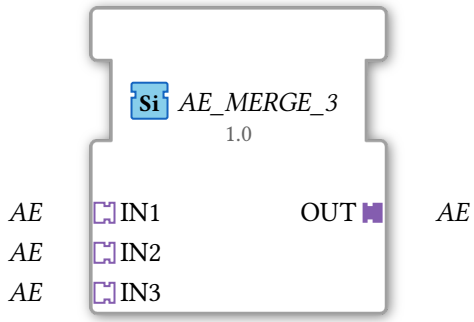
Vergleich mit ähnlichen Bausteinen

- AE_SPLIT_2: die Umkehrrichtung – verteilt ein eingehendes Ereignis auf zwei Ausgänge, statt zwei Eingänge zusammenzuführen.
- ASR_MERGE_2: dieselbe Zusammenführungslogik für den Set/Reset-Ereignisadapter `ASR` (2 Ereignisse statt 1).
- ASRT_MERGE_2: dieselbe Zusammenführungslogik für den Set/Reset/Toggle-Ereignisadapter `ASRT` (3 Ereignisse statt 1).

Fazit

Der **AE_MERGE_2** ist ein minimaler generischer Baustein zur Zusammenführung zweier reiner Ereignisquellen auf einen gemeinsamen AE-Adapterausgang. Seine fehlende Logik sorgt für minimale Latenz, während die generische Implementierung ihn konsistent mit den übrigen `GEN_*_MERGE`-Bausteinen der Familie macht.

AE_MERGE_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AE_MERGE_3** führt 3 unidirektionale AE-Adapter (reinen Ereignisadapter, keine Nutzdaten) auf einen gemeinsamen Ausgangsadapter zusammen. Er ist die 3-Eingangs-Variante von AE_MERGE_2 und wie dieser als generischer FB implementiert (`GenericClassName: 'GEN_AE_MERGE'`) — dieselbe C++-Basis (`CGenUnidirectMergeBase`), nur mit 3 statt 2 Sockets.

Schnittstellenstruktur

Ereignis-Eingänge
Keine. Die Ereignisübernahme erfolgt ausschließlich über die Adapter-Sockets.

Ereignis-Ausgänge
Keine. Die Ereignisweitergabe erfolgt ausschließlich über den Adapter-Plug.

Daten-Eingänge
Keine.

Daten-Ausgänge
Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket (Eingang 1)	`IN1`	`adapter::types::unidirectional::AE`	1. Quelle.
Socket (Eingang 2)	`IN2`	`adapter::types::unidirectional::AE`	2. Quelle.
Socket (Eingang 3)	`IN3`	`adapter::types::unidirectional::AE`	3. Quelle.
Plug (Ausgang)	`OUT`	`adapter::types::unidirectional::AE`	Zusammengeführtes Signal.

Funktionsweise
Trifft `E1` an einem der 3 Sockets (`IN1`, `IN2`, `IN3`) ein, wird es als gleichartiges Ereignis unverändert an `OUT` weitergereicht. Alle Ereignistypen werden unabhängig voneinander behandelt, und alle 3 Sockets werden gleichberechtigt behandelt — welcher Socket zuerst geprüft wird, folgt der Reihenfolge `IN1` … `IN3`, was nur bei mehreren innerhalb desselben Ausführungszyklus eintreffenden Ereignissen relevant wird. Da `AE` keinen Datenwert transportiert, gibt es nichts zu arbitrieren außer dem jeweiligen Ereignistyp selbst.

Technische Besonderheiten
- **Generischer Typ**: Implementiert über die generische Basisklasse `CGenUnidirectMergeBase` (N Sockets, 1 Plug); die Anzahl der Eingangs-Sockets wird über den generischen Suffix (`_3`) zur Übersetzungszeit festgelegt.
- **Keine Zustände / Algorithmen**: Da `AE` keinen Datenwert trägt, gibt es keine Änderungserkennung und kein ECC.
- **Beliebige Eingangsanzahl**: Dieselbe Implementierung deckt AE_MERGE_2 bis AE_MERGE_7 ab; für andere Eingangsanzahlen siehe `AE_MERGE_2`, `AE_MERGE_4`, `AE_MERGE_5`, `AE_MERGE_6`, `AE_MERGE_7`.

Zustandsübersicht
Der Baustein besitzt **keinen Zustandsautomaten**. Das Verhalten ist rein kombinatorisch: Jedes eingehende Ereignis an einem der 3 Sockets wird unmittelbar an `OUT` weitergereicht.

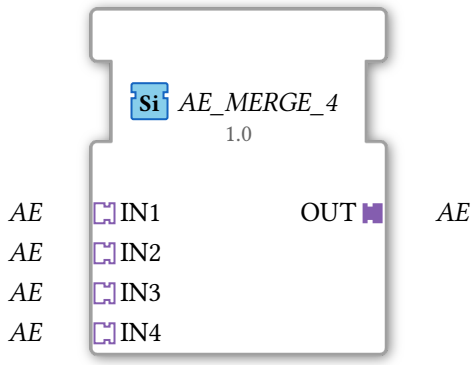
Anwendungsszenarien
- **Zusammenführen mehrerer gleichwertiger Signalquellen** (z. B. 3 redundante oder alternative Auslösepfade) auf einen gemeinsamen AE-Adapterausgang.
- **Vereinfachung von Netzwerken**, die sonst 3 separate Ereignisverbindungen zum selben Ziel benötigen würden.

Vergleich mit ähnlichen Bausteinen
- **AE_SPLIT_2**: die Umkehrrichtung für 2 Ausgänge (bei ASRT zusätzlich AE_SPLIT_3 bis AE_SPLIT_9 verfügbar).
- **`AE_MERGE_2`, `AE_MERGE_4`, `AE_MERGE_5`, `AE_MERGE_6`, `AE_MERGE_7`**: dieselbe generische Implementierung mit anderer Eingangsanzahl.
- ASR_MERGE_3, ASRT_MERGE_3: dieselbe Zusammenführungslogik für die jeweils anderen unidirektionalen Ereignisadapter.

Fazit

`AE_MERGE_3` liefert eine generisch implementierte Zusammenführung von 3 `AE`-Ereignisquellen auf einen gemeinsamen Adapterausgang und ergänzt die `GEN_AE_MERGE`-Familie um die 3-Eingangs-Variante.

AE_MERGE_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AE_MERGE_4** führt 4 unidirektionale AE-Adapter (reinen Ereignisadapter, keine Nutzdaten) auf einen gemeinsamen Ausgangsadapter zusammen. Er ist die 4-Eingangs-Variante von AE_MERGE_2 und wie dieser als generischer FB implementiert (`GenericClassName: 'GEN_AE_MERGE'`) — dieselbe C++-Basis (`CGenUnidirectMergeBase`), nur mit 4 statt 2 Sockets.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignisübernahme erfolgt ausschließlich über die Adapter-Sockets.

Ereignis-Ausgänge

Keine. Die Ereignisweitergabe erfolgt ausschließlich über den Adapter-Plug.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket (Eingang 1)	`IN1`	`adapter::types::unidirectional::AE`	1. Quelle.
Socket (Eingang 2)	`IN2`	`adapter::types::unidirectional::AE`	2. Quelle.
Socket (Eingang 3)	`IN3`	`adapter::types::unidirectional::AE`	3. Quelle.
Socket (Eingang 4)	`IN4`	`adapter::types::unidirectional::AE`	4. Quelle.
Plug (Ausgang)	`OUT`	`adapter::types::unidirectional::AE`	Zusammengeführtes Signal.

Funktionsweise

Trifft `E1` an einem der 4 Sockets (`IN1`, `IN2`, `IN3`, `IN4`) ein, wird es als gleichartiges Ereignis unverändert an `OUT` weitergereicht. Alle Ereignistypen werden unabhängig voneinander behandelt, und alle 4 Sockets werden gleichberechtigt behandelt — welcher Socket zuerst geprüft wird, folgt der Reihenfolge `IN1` … `IN4`, was nur bei mehreren innerhalb desselben Ausführungszyklus eintreffenden Ereignissen relevant wird. Da `AE` keinen Datenwert transportiert, gibt es nichts zu arbitrieren außer dem jeweiligen Ereignistyp selbst.

Technische Besonderheiten

- **Generischer Typ**: Implementiert über die generische Basisklasse `CGenUnidirectMergeBase` (N Sockets, 1 Plug); die Anzahl der Eingangs-Sockets wird über den generischen Suffix (`_4`) zur Übersetzungszeit festgelegt.
- **Keine Zustände / Algorithmen**: Da `AE` keinen Datenwert trägt, gibt es keine Änderungserkennung und kein ECC.
- **Beliebige Eingangsanzahl**: Dieselbe Implementierung deckt AE_MERGE_2 bis AE_MERGE_7 ab; für andere Eingangsanzahlen siehe `AE_MERGE_2`, `AE_MERGE_3`, `AE_MERGE_5`, `AE_MERGE_6`, `AE_MERGE_7`.

Zustandsübersicht

Der Baustein besitzt **keinen Zustandsautomaten**. Das Verhalten ist rein kombinatorisch: Jedes eingehende Ereignis an einem der 4 Sockets wird unmittelbar an `OUT` weitergereicht.

Anwendungsszenarien

- **Zusammenführen mehrerer gleichwertiger Signalquellen** (z. B. 4 redundante oder alternative Auslösepfade) auf einen gemeinsamen AE-Adapterausgang.
- **Vereinfachung von Netzwerken**, die sonst 4 separate Ereignisverbindungen zum selben Ziel benötigen würden.

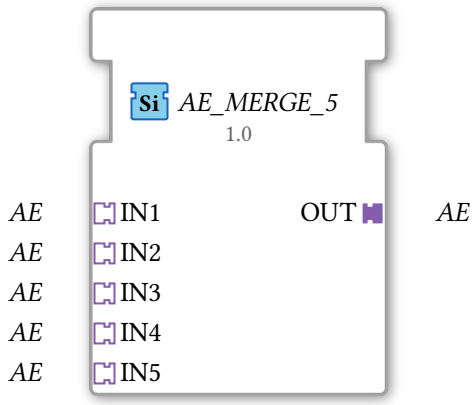
Vergleich mit ähnlichen Bausteinen

- AE_SPLIT_2: die Umkehrrichtung für 2 Ausgänge (bei ASRT zusätzlich AE_SPLIT_3 bis AE_SPLIT_9 verfügbar).
- **`AE_MERGE_2`, `AE_MERGE_3`, `AE_MERGE_5`, `AE_MERGE_6`, `AE_MERGE_7`**: dieselbe generische Implementierung mit anderer Eingangsanzahl.
- ASR_MERGE_4, ASRT_MERGE_4: dieselbe Zusammenführungslogik für die jeweils anderen unidirektionalen Ereignisadapter.

Fazit

`AE_MERGE_4` liefert eine generisch implementierte Zusammenführung von 4 `AE`-Ereignisquellen auf einen gemeinsamen Adapterausgang und ergänzt die `GEN_AE_MERGE`-Familie um die 4-Eingangs-Variante.

AE_MERGE_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AE_MERGE_5** führt 5 unidirektionale AE-Adapter (reinen Ereignisadapter, keine Nutzdaten) auf einen gemeinsamen Ausgangsadapter zusammen. Er ist die 5-Eingangs-Variante von AE_MERGE_2 und wie dieser als generischer FB implementiert (`GenericClassName: 'GEN_AE_MERGE'`) — dieselbe C++-Basis (`CGenUnidirectMergeBase`), nur mit 5 statt 2 Sockets.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignisübernahme erfolgt ausschließlich über die Adapter-Sockets.

Ereignis-Ausgänge

Keine. Die Ereignisweitergabe erfolgt ausschließlich über den Adapter-Plug.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket (Eingang 1)	`IN1`	`adapter::types::unidirectional::AE`	1. Quelle.
Socket (Eingang 2)	`IN2`	`adapter::types::unidirectional::AE`	2. Quelle.
Socket (Eingang 3)	`IN3`	`adapter::types::unidirectional::AE`	3. Quelle.
Socket (Eingang 4)	`IN4`	`adapter::types::unidirectional::AE`	4. Quelle.
Socket (Eingang 5)	`IN5`	`adapter::types::unidirectional::AE`	5. Quelle.
Plug (Ausgang)	`OUT`	`adapter::types::unidirectional::AE`	Zusammengeführtes Signal.

Funktionsweise

Trifft `E1` an einem der 5 Sockets (`IN1`, `IN2`, `IN3`, `IN4`, `IN5`) ein, wird es als gleichartiges Ereignis unverändert an `OUT` weitergereicht. Alle Ereignistypen werden unabhängig voneinander behandelt, und alle 5 Sockets werden gleichberechtigt behandelt — welcher Socket zuerst geprüft wird, folgt der Reihenfolge `IN1` … `IN5`, was nur bei mehreren innerhalb desselben Ausführungszyklus eintreffenden Ereignissen relevant wird. Da `AE` keinen Datenwert transportiert, gibt es nichts zu arbitrieren außer dem jeweiligen Ereignistyp selbst.

Technische Besonderheiten

- **Generischer Typ**: Implementiert über die generische Basisklasse `CGenUnidirectMergeBase` (N Sockets, 1 Plug); die Anzahl der Eingangs-Sockets wird über den generischen Suffix (`_5`) zur Übersetzungszeit festgelegt.
- **Keine Zustände / Algorithmen**: Da `AE` keinen Datenwert trägt, gibt es keine Änderungserkennung und kein ECC.
- **Beliebige Eingangsanzahl**: Dieselbe Implementierung deckt AE_MERGE_2 bis AE_MERGE_7 ab; für andere Eingangsanzahlen siehe `AE_MERGE_2`, `AE_MERGE_3`, `AE_MERGE_4`, `AE_MERGE_6`, `AE_MERGE_7`.

Zustandsübersicht

Der Baustein besitzt **keinen Zustandsautomaten**. Das Verhalten ist rein kombinatorisch: Jedes eingehende Ereignis an einem der 5 Sockets wird unmittelbar an `OUT` weitergereicht.

Anwendungsszenarien

- **Zusammenführen mehrerer gleichwertiger Signalquellen** (z. B. 5 redundante oder alternative Auslösepfade) auf einen gemeinsamen AE-Adapterausgang.
- **Vereinfachung von Netzwerken**, die sonst 5 separate Ereignisverbindungen zum selben Ziel benötigen würden.

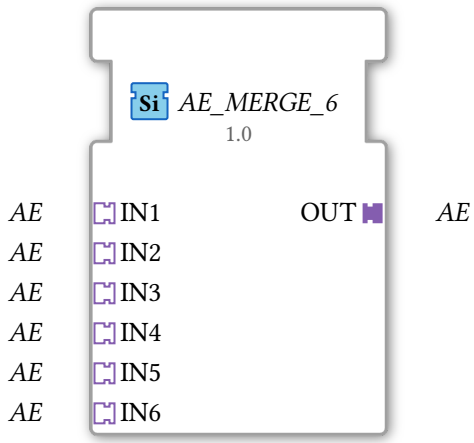
Vergleich mit ähnlichen Bausteinen

- **AE_SPLIT_2**: die Umkehrrichtung für 2 Ausgänge (bei ASRT zusätzlich AE_SPLIT_3 bis AE_SPLIT_9 verfügbar).
- **`AE_MERGE_2`, `AE_MERGE_3`, `AE_MERGE_4`, `AE_MERGE_6`, `AE_MERGE_7`**: dieselbe generische Implementierung mit anderer Eingangsanzahl.
- ASR_MERGE_5, ASRT_MERGE_5: dieselbe Zusammenführungslogik für die jeweils anderen unidirektionalen Ereignisadapter.

Fazit

`AE_MERGE_5` liefert eine generisch implementierte Zusammenführung von 5 `AE`-Ereignisquellen auf einen gemeinsamen Adapterausgang und ergänzt die `GEN_AE_MERGE`-Familie um die 5-Eingangs-Variante.

AE_MERGE_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AE_MERGE_6** führt 6 unidirektionale AE-Adapter (reinen Ereignisadapter, keine Nutzdaten) auf einen gemeinsamen Ausgangsadapter zusammen. Er ist die 6-Eingangs-Variante von AE_MERGE_2 und wie dieser als generischer FB implementiert (`GenericClassName: 'GEN_AE_MERGE'`) — dieselbe C++-Basis (`CGenUnidirectMergeBase`), nur mit 6 statt 2 Sockets.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignisübernahme erfolgt ausschließlich über die Adapter-Sockets.

Ereignis-Ausgänge

Keine. Die Ereignisweitergabe erfolgt ausschließlich über den Adapter-Plug.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket (Eingang 1)	`IN1`	`adapter::types::unidirectional::AE`	1. Quelle.
Socket (Eingang 2)	`IN2`	`adapter::types::unidirectional::AE`	2. Quelle.
Socket (Eingang 3)	`IN3`	`adapter::types::unidirectional::AE`	3. Quelle.
Socket (Eingang 4)	`IN4`	`adapter::types::unidirectional::AE`	4. Quelle.
Socket (Eingang 5)	`IN5`	`adapter::types::unidirectional::AE`	5. Quelle.
Socket (Eingang 6)	`IN6`	`adapter::types::unidirectional::AE`	6. Quelle.
Plug (Ausgang)	`OUT`	`adapter::types::unidirectional::AE`	Zusammengeführtes Signal.

Funktionsweise

Trifft `E1` an einem der 6 Sockets (`IN1`, `IN2`, `IN3`, `IN4`, `IN5`, `IN6`) ein, wird es als gleichartiges Ereignis unverändert an `OUT` weitergereicht. Alle Ereignistypen werden unabhängig voneinander behandelt, und alle 6 Sockets werden gleichberechtigt behandelt — welcher Socket zuerst geprüft wird, folgt der Reihenfolge `IN1` … `IN6`, was nur bei mehreren innerhalb desselben Ausführungszyklus eintreffenden Ereignissen relevant wird. Da `AE` keinen Datenwert transportiert, gibt es nichts zu arbitrieren außer dem jeweiligen Ereignistyp selbst.

Technische Besonderheiten

- **Generischer Typ**: Implementiert über die generische Basisklasse `CGenUnidirectMergeBase` (N Sockets, 1 Plug); die Anzahl der Eingangs-Sockets wird über den generischen Suffix (`_6`) zur Übersetzungszeit festgelegt.
- **Keine Zustände / Algorithmen**: Da `AE` keinen Datenwert trägt, gibt es keine Änderungserkennung und kein ECC.
- **Beliebige Eingangsanzahl**: Dieselbe Implementierung deckt AE_MERGE_2 bis AE_MERGE_7 ab; für andere Eingangsanzahlen siehe `AE_MERGE_2`, `AE_MERGE_3`, `AE_MERGE_4`, `AE_MERGE_5`, `AE_MERGE_7`.

Zustandsübersicht

Der Baustein besitzt **keinen Zustandsautomaten**. Das Verhalten ist rein kombinatorisch: Jedes eingehende Ereignis an einem der 6 Sockets wird unmittelbar an `OUT` weitergereicht.

Anwendungsszenarien

- **Zusammenführen mehrerer gleichwertiger Signalquellen** (z. B. 6 redundante oder alternative Auslösepfade) auf einen gemeinsamen AE-Adapterausgang.
- **Vereinfachung von Netzwerken**, die sonst 6 separate Ereignisverbindungen zum selben Ziel benötigen würden.

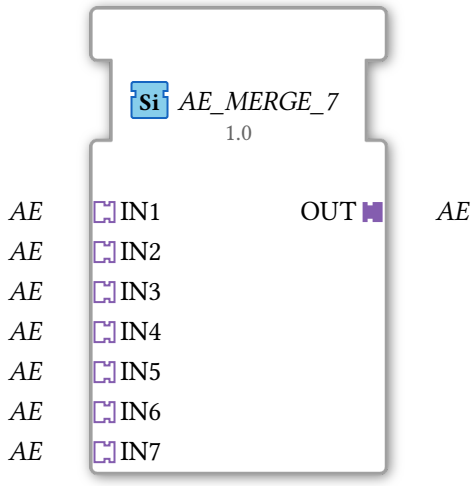
Vergleich mit ähnlichen Bausteinen

- **AE_SPLIT_2**: die Umkehrrichtung für 2 Ausgänge (bei ASRT zusätzlich AE_SPLIT_3 bis AE_SPLIT_9 verfügbar).
- **`AE_MERGE_2`, `AE_MERGE_3`, `AE_MERGE_4`, `AE_MERGE_5`, `AE_MERGE_7`**: dieselbe generische Implementierung mit anderer Eingangsanzahl.
- ASR_MERGE_6, ASRT_MERGE_6: dieselbe Zusammenführungslogik für die jeweils anderen unidirektionalen Ereignisadapter.

Fazit

`AE_MERGE_6` liefert eine generisch implementierte Zusammenführung von 6 `AE`-Ereignisquellen auf einen gemeinsamen Adapterausgang und ergänzt die `GEN_AE_MERGE`-Familie um die 6-Eingangs-Variante.

AE_MERGE_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AE_MERGE_7** führt 7 unidirektionale AE-Adapter (reinen Ereignisadapter, keine Nutzdaten) auf einen gemeinsamen Ausgangsadapter zusammen. Er ist die 7-Eingangs-Variante von AE_MERGE_2 und wie dieser als generischer FB implementiert (`GenericClassName: 'GEN_AE_MERGE'`) — dieselbe C++-Basis (`CGenUnidirectMergeBase`), nur mit 7 statt 2 Sockets.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignisübernahme erfolgt ausschließlich über die Adapter-Sockets.

Ereignis-Ausgänge

Keine. Die Ereignisweitergabe erfolgt ausschließlich über den Adapter-Plug.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket (Eingang 1)	`IN1`	`adapter::types::unidirectional::AE`	1. Quelle.
Socket (Eingang 2)	`IN2`	`adapter::types::unidirectional::AE`	2. Quelle.
Socket (Eingang 3)	`IN3`	`adapter::types::unidirectional::AE`	3. Quelle.
Socket (Eingang 4)	`IN4`	`adapter::types::unidirectional::AE`	4. Quelle.
Socket (Eingang 5)	`IN5`	`adapter::types::unidirectional::AE`	5. Quelle.
Socket (Eingang 6)	`IN6`	`adapter::types::unidirectional::AE`	6. Quelle.
Socket (Eingang 7)	`IN7`	`adapter::types::unidirectional::AE`	7. Quelle.
Plug (Ausgang)	`OUT`	`adapter::types::unidirectional::AE`	Zusammengeführtes Signal.

Funktionsweise

Trifft `E1` an einem der 7 Sockets (`IN1`, `IN2`, `IN3`, `IN4`, `IN5`, `IN6`, `IN7`) ein, wird es als gleichartiges Ereignis unverändert an `OUT` weitergereicht. Alle Ereignistypen werden unabhängig voneinander behandelt, und alle 7 Sockets werden gleichberechtigt behandelt — welcher Socket zuerst geprüft wird, folgt der Reihenfolge `IN1` … `IN7`, was nur bei mehreren innerhalb desselben Ausführungszyklus eintreffenden Ereignissen relevant wird. Da `AE` keinen Datenwert transportiert, gibt es nichts zu arbitrieren außer dem jeweiligen Ereignistyp selbst.

Technische Besonderheiten

- **Generischer Typ**: Implementiert über die generische Basisklasse `CGenUnidirectMergeBase` (N Sockets, 1 Plug); die Anzahl der Eingangs-Sockets wird über den generischen Suffix (`_7`) zur Übersetzungszeit festgelegt.
- **Keine Zustände / Algorithmen**: Da `AE` keinen Datenwert trägt, gibt es keine Änderungserkennung und kein ECC.
- **Beliebige Eingangsanzahl**: Dieselbe Implementierung deckt AE_MERGE_2 bis AE_MERGE_7 ab; für andere Eingangsanzahlen siehe `AE_MERGE_2`, `AE_MERGE_3`, `AE_MERGE_4`, `AE_MERGE_5`, `AE_MERGE_6`.

Zustandsübersicht

Der Baustein besitzt **keinen Zustandsautomaten**. Das Verhalten ist rein kombinatorisch: Jedes eingehende Ereignis an einem der 7 Sockets wird unmittelbar an `OUT` weitergereicht.

Anwendungsszenarien

- **Zusammenführen mehrerer gleichwertiger Signalquellen** (z. B. 7 redundante oder alternative Auslösepfade) auf einen gemeinsamen AE-Adapterausgang.
- **Vereinfachung von Netzwerken**, die sonst 7 separate Ereignisverbindungen zum selben Ziel benötigen würden.

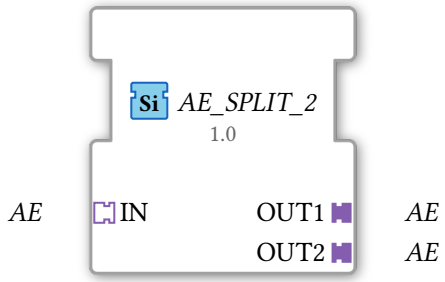
Vergleich mit ähnlichen Bausteinen

- AE_SPLIT_2: die Umkehrrichtung für 2 Ausgänge (bei ASRT zusätzlich AE_SPLIT_3 bis AE_SPLIT_9 verfügbar).
- **AE_MERGE_2, AE_MERGE_3, AE_MERGE_4, AE_MERGE_5, AE_MERGE_6**: dieselbe generische Implementierung mit anderer Eingangsanzahl.
- ASR_MERGE_7, ASRT_MERGE_7: dieselbe Zusammenführungslogik für die jeweils anderen unidirektionalen Ereignisadapter.

Fazit

AE_MERGE_7 liefert eine generisch implementierte Zusammenführung von 7 AE-Ereignisquellen auf einen gemeinsamen Adapterausgang und ergänzt die GEN_AE_MERGE-Familie um die 7-Eingangs-Variante.

AE_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AE_SPLIT_2** dient als generischer Verteiler für ein eingehendes Adapter-Ereignis (AE). Er empfängt ein unidirektionales Ereignis über den Socket IN und leitet es ohne Verzögerung oder Veränderung an die beiden Ausgangs-Plugs OUT1 und OUT2 weiter. Der Baustein wird als generischer FB (GenericClassName: 'GEN_AE_SPLIT') implementiert, sodass er theoretisch für verschiedene Adaptertypen verwendbar ist – vorausgesetzt, sie entsprechen dem unidirektionalen AE-Protokoll.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Der FB besitzt keine klassischen Ereigniseingänge. Die Ereignisweiterleitung erfolgt ausschließlich über den Adapter-Socket.

Ereignis-Ausgänge

Keine. Die Ereignisweitergabe geschieht über die Adapter-Plugs.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Rolle	Name	Typ	Beschreibung

Socket (Eingang)	`IN`	`adapter::types::unidirectional::AE`	Empfängt ein unidirektionales Ereignis, das auf beide Ausgänge verteilt wird.
Plug (Ausgang 1)	`OUT1`	`adapter::types::unidirectional::AE`	Erster Ausgang für das duplizierte Ereignis.
Plug (Ausgang 2)	`OUT2`	`adapter::types::unidirectional::AE`	Zweiter Ausgang für das duplizierte Ereignis.

Funktionsweise

Sobald am Adapter-Socket `IN` ein Ereignis eintrifft, wird es **sofort und parallel** an beide Ausgangs-Plugs `OUT1` und `OUT2` weitergeleitet. Der Baustein führt keinerlei Logik, Filterung oder Verzögerung durch – er fungiert als reiner Splitter auf Adapterebene. Die Verteilung erfolgt ereignisgetreu: Jedes einzelne eingehende Ereignis erscheint an beiden Ausgängen.

Technische Besonderheiten

- **Generischer Typ**: Der FB ist als generischer Baustein deklariert (`GenericClassName = 'GEN_AE_SPLIT'`). Dies erlaubt eine Anpassung an andere, vom Benutzer definierte Adaptertypen, solange sie die unidirektionale Ereignisschnittstelle nutzen. In der vorliegenden Konfiguration ist der konkrete Typ `adapter::types::unidirectional::AE` hinterlegt.
- **Keine Zustände / Algorithmen**: Da keinerlei Logik oder Speicherverhalten implementiert ist, besitzt der FB keinen ECC (Execution Control Chart) und keine internen Zustände.
- **Latenz**: Die Weiterleitung erfolgt verzögerungsfrei innerhalb eines Ausführungszyklus.

Zustandsübersicht

Der FB besitzt **keinen Zustandsautomaten**. Das Verhalten ist rein kombinatorisch: Jedes eingehende Adapter-Ereignis wird sofort dupliziert und an beiden Ausgängen ausgegeben.

Anwendungsszenarien

- **Ereignisverteilung**: Ein Sensor oder eine Steuerung liefert ein Ereignis (z. B. „Start“ oder „Alarm“), das von zwei unabhängigen Subsystemen verarbeitet werden muss.
- **Parallelschaltung**: Aufteilung eines Signals zur gleichzeitigen Ansteuerung zweier Aktoren oder zur parallelen Datenaufnahme.
- **Prototyping**: Temporäre Verzweigung eines Ereignisstroms während der Entwicklungs- und Testphase.

Vergleich mit ähnlichen Bausteinen

- **AE_SPLIT_1**: Ein Splitter mit nur einem Ausgang (d. h. reine Durchleitung) – meist überflüssig.
- **E_SPLIT_2**: Klassischer Ereignis-Splitter für standardmäßige Ereigniseingänge/-ausgänge. Im Gegensatz zu `AE_SPLIT_2` arbeitet dieser auf der Ereignisebene ohne Adapter-Kopplung.
- **AE_ROUTER**: Ein adapterbasierter Verteiler, der Ereignisse gezielt an bestimmte Ausgänge weiterleiten kann (z. B. basierend auf Datenwerten). `AE_SPLIT_2` ist simpler und verteilt immer an alle Ausgänge.

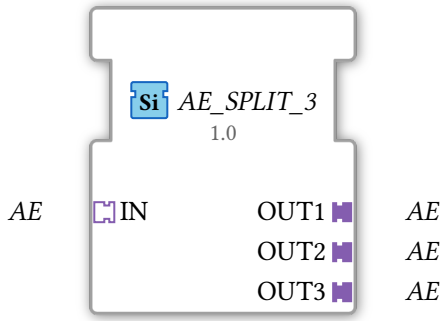
Fazit

Der **AE_SPLIT_2** ist ein minimaler, aber nützlicher Funktionsblock zur einfachen Verteilung von Adapter-Ereignissen. Seine generische Natur macht ihn flexibel einsetzbar, während die fehlende Logik für geringe Latenz und hohe Zuverlässigkeit sorgt. Er eignet sich ideal für alle Szenarien, in denen ein Ereignis an mehrere Empfänger weitergegeben werden muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Gesamtwiderstand in Reihen- & Parallelschaltung auf ms-muc-docs.de

AE_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AE_SPLIT_3** dient dazu, einen eingehenden Adapter-Datenstrom (vom Typ `AE`) auf drei identische Ausgänge zu verteilen. Er ist als generischer Baustein ausgeführt und ermöglicht die flexible Verwendung unterschiedlicher Adaptertypen durch Konfiguration zur Designzeit.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Sockel (Eingang)	IN	`adapter::types::unidirectional::AE`	Eingehender Adapter-Datenstrom, der auf die drei Ausgänge verteilt wird.
Stecker (Ausgang)	OUT1	`adapter::types::unidirectional::AE`	Erster Ausgang – erhält die gleichen Daten wie der Eingang.
Stecker (Ausgang)	OUT2	`adapter::types::unidirectional::AE`	Zweiter Ausgang – erhält die gleichen Daten wie der Eingang.
Stecker (Ausgang)	OUT3	`adapter::types::unidirectional::AE`	Dritter Ausgang – erhält die gleichen Daten wie der Eingang.

Funktionsweise

Der Baustein akzeptiert über den Socket `IN` einen unidirektionalen Adapter-Datenstrom vom Typ `AE`. Jedes ankommende Datenpaket oder Ereignis wird ohne Veränderung an alle drei Adapter-Plugs (`OUT1`, `OUT2`, `OUT3`) weitergeleitet. Es findet keine Verarbeitung oder Aufbereitung der Daten statt – der FB fungiert als reiner **Signalverteiler** für Adapterverbindungen.

Dank der generischen Auslegung (`eclipse4diac::core::GenericClassName = 'GEN_AE_SPLIT'`) kann der konkrete Adaptertyp (z. B. ein benutzerdefinierter AE-Subtyp) zur Entwurfszeit für `IN`, `OUT1`–`OUT3` festgelegt werden. Zur Laufzeit sind alle Instanzen identisch typisiert.

Technische Besonderheiten

- **Generischer Baustein** – der Adaptertyp wird erst bei der Instanziierung im Projekt festgelegt, was Wiederverwendbarkeit und Typ-Sicherheit erhöht.
- **Unidirektionale Kommunikation** – Daten fließen nur vom Eingang zu den Ausgängen; Rückwärtskommunikation ist nicht vorgesehen.
- **Lizenz** – Der Baustein steht unter der **Eclipse Public License 2.0 (EPL-2.0)**, die eine freie Nutzung, Modifikation und Weitergabe erlaubt.
- Keine ereignis- oder datenbasierten Ein-/Ausgänge – die gesamte Kommunikation erfolgt ausschließlich über die Adapterschnittstellen.

Zustandsübersicht

Der Funktionsblock besitzt kein internes Zustandsdiagramm (ECC). Er arbeitet **kombinatorisch**: Jede Änderung am Eingang wird sofort und ohne Verzögerung an alle Ausgänge weitergegeben.

Anwendungsszenarien

- **Signalvervielfältigung** – Ein von einem Sensor bereitgestellter Adapter-Datenstrom (z. B. `AE_Temperature`) soll parallel an drei verschiedene Auswerteeinheiten gesendet werden.
- **Aufteilung von Ereignisadaptern** – In Automatisierungssystemen, die auf Adapter-basierte Kommunikation setzen, können so mehrere Empfänger bedient werden.
- **Test- und Simulationsumgebungen** – Ein Datenstrom wird auf mehrere Überwachungs- oder Protokollierungsmodule verteilt.

Vergleich mit ähnlichen Bausteinen

Baustein	Anzahl Ausgänge	Besonderheit
`AE_SPLIT_2`	2	Verteilt auf zwei Ausgänge.
`AE_SPLIT_3`	3	Verteilt auf drei Ausgänge (vorliegender FB).
`AE_SPLIT_N` (hypothetisch)	variabel	Flexible Anzahl über Parameter – erfordert höheren Konfigurationsaufwand.

Gemeinsam ist allen Varianten, dass sie lediglich auf Adapterebene arbeiten und keine Datenmanipulation vornehmen.

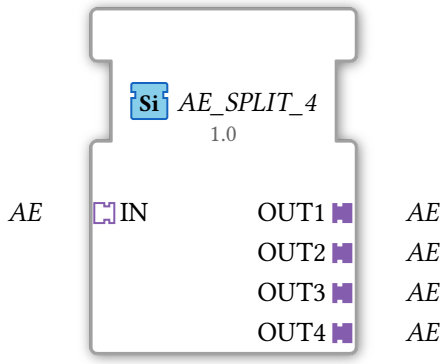
Fazit

Der **AE_SPLIT_3** ist ein einfacher und dennoch unverzichtbarer Funktionsblock zur Vervielfältigung von Adapterverbindungen in der 4diac-IDE. Seine generische Ausführung und die klare, unidirektionale Struktur machen ihn zu einem wartungsfreundlichen Werkzeug für die Signalverteilung ohne zusätzliche Logik. Durch die EPL-2.0-Lizenz kann er in eigenen Projekten ohne Einschränkungen eingesetzt werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AE_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AE_SPLIT_4** verteilt ein eingehendes Ereignis eines unidirektionalen **AE-Adapters** auf vier identische Ausgangsadapter. Er wird als generischer Baustein (Generic FB) ausgeführt und kann daher für beliebige Datentypen des Adapters verwendet werden. Die Verteilung erfolgt ohne Verzögerung oder Zustandsänderung – das ankommende Ereignis wird gleichzeitig an alle vier Ausgänge weitergeleitet.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock besitzt keine expliziten Ereignis-Eingänge. Die Ereignissteuerung erfolgt ausschließlich über den Socket **IN** (siehe Abschnitt Adapter).

Ereignis-Ausgänge

Der Funktionsblock besitzt keine expliziten Ereignis-Ausgänge. Die Ereignisweitergabe erfolgt über die Plugs **OUT1** bis **OUT4** (siehe Abschnitt Adapter).

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	**IN**	`adapter::types::unidirectional::AE`	Empfängt ein unidirektionales Adapter-Ereignis (Ereignis + zugehörige Daten).
Plug (Ausgang)	**OUT1**	`adapter::types::unidirectional::AE`	Erster Ausgang – erhält das gleiche Ereignis und dieselben Daten wie am Eingang.
Plug (Ausgang)	**OUT2**	`adapter::types::unidirectional::AE`	Zweiter Ausgang.
Plug (Ausgang)	**OUT3**	`adapter::types::unidirectional::AE`	Dritter Ausgang.
Plug (Ausgang)	**OUT4**	`adapter::types::unidirectional::AE`	Vierter Ausgang.

Funktionsweise

Sobald am Socket **IN** ein Ereignis des AE-Adapters anliegt (mit allen zu diesem Ereignis gehörenden Daten), wird dieses Ereignis sofort und parallel an alle vier Plugs **OUT1**, **OUT2**, **OUT3** und **OUT4** weitergegeben. Die Ausgabe erfolgt zeitgleich – es gibt keine Prioritäten oder Sequenzen. Die Datentypen der Adapter sind aufgrund der generischen Auslegung des Bausteins frei wählbar (z. B. `INT`, `REAL`, `BOOL` etc.), solange alle beteiligten Adapter denselben Datentyp verwenden.

Technische Besonderheiten

- **Generischer Baustein** – Der FB ist als generischer Typ (`GenericClassName = 'GEN_AE_SPLIT'`) realisiert. Dadurch kann der konkrete Datentyp des Adapters zur Projektierungszeit festgelegt werden.
- **Unidirektionale Adapter** – Der Baustein arbeitet ausschließlich mit Adaptern des Typs `unidirectional::AE`. Rückkanäle oder bidirektionale Kommunikation werden nicht unterstützt.
- **Lizenz** – Der Quellcode unterliegt der **Eclipse Public License 2.0** (EPL-2.0).

Zustandsübersicht

Der **AE_SPLIT_4** ist ein rein kombinatorischer Baustein und besitzt keine internen Zustände. Das Ausgangssignal hängt ausschließlich vom aktuellen Eingangssignal ab – es gibt keine Speicherung oder Verzögerung.

Anwendungsszenarien

- **Sensorverteilung** – Ein einzelner Sensor (z. B. Temperatur, Drehzahl) liefert Daten über einen AE-Adapter; diese sollen an mehrere parallel arbeitende Steuerungsfunktionen verteilt werden.
- **Signalweiterleitung in einer Steuerungskette** – Ein Ereignis muss mehrere nachgelagerte FBs parallel erreichen, z. B. zur Auslösung von Alarmen und Protokollierung.
- **Multiplexing von Ereignisquellen** – (in Verbindung mit einem vorgeschalteten AE-MUX) können verschiedene Quellen auf die gleichen vier Ausgänge geschaltet werden.

Vergleich mit ähnlichen Bausteinen

- **E_SPLIT** – Standard-4diac-Baustein zur Verteilung eines Ereignisses auf mehrere Ereignisausgänge ohne Datenweitergabe. **AE_SPLIT_4** erweitert dies um die Fähigkeit, auch die zum Ereignis gehörenden Adapter-Daten mitzugeben.
- **F_SPLIT** – Daten-Splitter (z. B. für IEC-61131-3 „MUX“) arbeiten meist nicht ereignisgesteuert. **AE_SPLIT_4** ist speziell für die ereignisbasierte Adapterkommunikation optimiert.
- **AE_MERGE** – Der umgekehrte Baustein: fasst mehrere AE-Eingänge auf einen Ausgang zusammen.

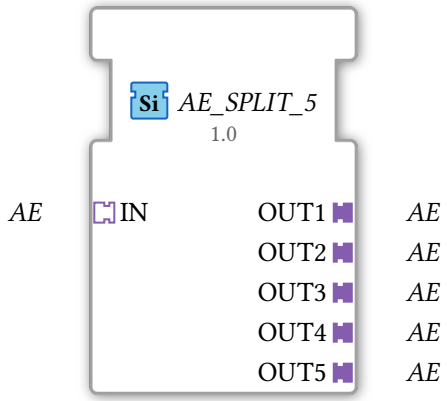
Fazit

Der **AE_SPLIT_4** ist ein einfacher, aber leistungsfähiger Funktionsblock, der die ereignisgesteuerte Verteilung von Adapterdaten auf bis zu vier gleichberechtigte Ausgänge ermöglicht. Dank seiner generischen Auslegung kann er ohne Anpassung für beliebige Datentypen verwendet werden und eignet sich hervorragend für modulare, parallele Steuerungsarchitekturen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AE_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AE_SPLIT_5** dient zur Vervielfältigung eines eingehenden Adapter-Ereignisses. Er empfängt ein Adapter-Ereignis (AE) an seinem Socket `IN` und leitet es identisch an fünf separate Ausgangs-Plugs (`OUT1` … `OUT5`) weiter. Der Baustein ist als generischer FB (GenericClassName `'GEN_AE_SPLIT'`) ausgeführt und ermöglicht so eine einfache Wiederverwendung in verschiedenen Kontexten.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	`IN`	`adapter::types::unidirectional::AE`	Ein unidirektionales Adapter-Ereignis, dessen Inhalt an alle Ausgänge weitergeleitet wird.
Plug (Ausgang 1)	`OUT1`	`adapter::types::unidirectional::AE`	Erster Ausgang, der das eingehende Ereignis bereitstellt.
Plug (Ausgang 2)	`OUT2`	`adapter::types::unidirectional::AE`	Zweiter Ausgang, der das eingehende Ereignis bereitstellt.
Plug (Ausgang 3)	`OUT3`	`adapter::types::unidirectional::AE`	Dritter Ausgang, der das eingehende Ereignis bereitstellt.
Plug (Ausgang 4)	`OUT4`	`adapter::types::unidirectional::AE`	Vierter Ausgang, der das eingehende Ereignis bereitstellt.
Plug (Ausgang 5)	`OUT5`	`adapter::types::unidirectional::AE`	Fünfter Ausgang, der das eingehende Ereignis bereitstellt.

Funktionsweise

Der FB **AE_SPLIT_5** ist ein reiner Verteilerbaustein. Sobald am Socket `IN` ein Adapter-Ereignis eintrifft, wird dieses Ereignis – ohne Verzögerung oder Datenmanipulation – parallel an alle fünf Ausgangs-Plugs `OUT1` bis `OUT5` gesendet. Es erfolgt keine interne Speicherung oder Zustandsveränderung; der FB ist vollständig kombinatorisch.

Technische Besonderheiten

- **Generische Implementierung**: Der Baustein ist als generischer FB deklariert (`GenericClassName = 'GEN_AE_SPLIT'`), sodass er bei Bedarf mit verschiedenen Adapter-Typen verwendet werden kann, sofern diese vom Typ `adapter::types::unidirectional::AE` sind.
- **Unidirektionalität**: Alle beteiligten Adapter sind unidirektional ausgelegt. Es findet keine Kommunikation zurück zum Sender statt.
- **Keine zusätzlichen Logiken**: Der FB besitzt weder Ereignis- noch Dateneingänge/-ausgänge. Die Weiterleitung erfolgt rein über Adapter.

Zustandsübersicht

Der Baustein besitzt keine definierte Zustandsmaschine oder Sequenzsteuerung. Sein Verhalten ist statisch: Bei jedem eingehenden Ereignis an `IN` werden alle Ausgänge gleichzeitig aktiviert. Es gibt keine Latenz oder Zwischenzustände.

Anwendungsszenarien

- **Ereignisverteilung in Steuerungssystemen**: Ein Sensor-Ereignis (z. B. „Position erreicht“) soll an mehrere unabhängige Steuerungskomponenten gesendet werden.
- **Multicast in modularen Architekturen**: In einer verteilten Automatisierungsumgebung kann ein zentraler Funktionsblock sein Ereignis an bis zu fünf nachgelagerte Module aufteilen.
- **Prototyping und Testumgebungen**: Zum gleichzeitigen Testen mehrerer Empfänger mit demselben Ereignis.

Vergleich mit ähnlichen Bausteinen

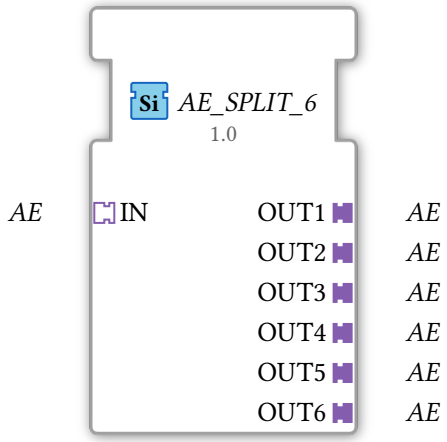
- **Ereignis-Splitter (E_SPLIT)**: Diese Bausteine verteilen **Ereignisse** (Event-Eingänge) auf mehrere Event-Ausgänge. **AE_SPLIT_5** hingegen verteilt **Adapter-Ereignisse** (komplexe Datenobjekte, die über Adapter kommunizieren) und bietet damit eine höhere Abstraktionsebene.

- **Daten-Splitter (F_SPLIT)**: Trennt Datenströme auf, besitzt aber keine Adapter-Schnittstelle. **AE_SPLIT_5** fokussiert auf die Verteilung von adapterbasierten Ereignisnachrichten.

Fazit

Der **AE_SPLIT_5** ist ein einfacher, aber effektiver Baustein zur Vervielfältigung eines unidirektionalen Adapter-Ereignisses auf fünf Ausgänge. Aufgrund seiner generischen Natur und der reinen Adapter-Schnittstelle eignet er sich besonders für modulare und skalierbare Steuerungsanwendungen, in denen Informationen parallel an mehrere Empfänger gesendet werden müssen.

AE_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AE_SPLIT_6** verteilt einen eingehenden AE-Adapter (Typ `adapter::types::unidirectional::AE`) auf sechs identische Ausgänge. Es handelt sich um einen generischen Baustein, der für verschiedene AE-Typen wiederverwendet werden kann. Der Name leitet sich von der 1:6-Aufteilung ab: Ein Eingang wird auf sechs Ausgänge gesplittet.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	IN	`adapter::types::unidirectional::AE`	Eingehender AE-Adapter, dessen Signal auf alle Ausgänge vervielfältigt wird.
Plug (Ausgang)	OUT1	`adapter::types::unidirectional::AE`	Erster Ausgang, identische Kopie des Eingangssignals.

Plug (Ausgang)	OUT2	adapter::types::unidirectional::AE	Zweiter Ausgang, identische Kopie des Eingangssignals.
Plug (Ausgang)	OUT3	adapter::types::unidirectional::AE	Dritter Ausgang, identische Kopie des Eingangssignals.
Plug (Ausgang)	OUT4	adapter::types::unidirectional::AE	Vierter Ausgang, identische Kopie des Eingangssignals.
Plug (Ausgang)	OUT5	adapter::types::unidirectional::AE	Fünfter Ausgang, identische Kopie des Eingangssignals.
Plug (Ausgang)	OUT6	adapter::types::unidirectional::AE	Sechster Ausgang, identische Kopie des Eingangssignals.

Funktionsweise

Der Baustein leitet das am Socket **IN** anliegende Adaptersignal unverändert an alle sechs Plugs (**OUT1** … **OUT6**) weiter. Es findet keinerlei logische Verarbeitung, Filterung oder Verzögerung statt – die Verteilung erfolgt passiv und verzögerungsfrei. Somit verhält sich der FB wie ein reiner 1:6-Splitter für den unidirektionalen Adaptertyp `AE`.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist mit dem Attribut `eclipse4diac::core::GenericClassName` als `'GEN_AE_SPLIT'` gekennzeichnet. Dadurch kann er in verschiedenen Projekten mit unterschiedlichen, aber strukturgleichen AE-Adaptertypen verwendet werden – erst zur Projektlaufzeit wird der konkrete Typ festgelegt.
- **Unidirektionalität**: Alle Adapter sind als unidirektional deklariert, d.h. die Daten-/Ereignisübertragung erfolgt nur vom Eingang (Socket) zu den Ausgängen (Plugs). Rückwirkungen sind nicht vorgesehen.
- **Keine Zustandslogik**: Der Baustein besitzt keine internen Zustände, keine Ereignisse und keine Daten-Ein-/Ausgänge. Er ist vollständig durch seine Adapterschnittstellen definiert.

Zustandsübersicht

Der FB **AE_SPLIT_6** ist ein rein kombinatorischer Baustein ohne Zustandsautomat. Es gibt keine internen Schritte, Transitionen oder Aktionen – die Verteilung erfolgt direkt und permanent.

Anwendungsszenarien

- **Signalverteilung**: Mehrere nachfolgende Funktionsblöcke sollen das gleiche AE-Signal empfangen, z.B. um parallele Berechnungen oder parallele Ausgabe zu starten.
- **Bus-Struktur**: In einer Steuerungsarchitektur, in der ein Ereignis an mehrere Komponenten (Aktoren, Anzeigen, Logger) gesendet werden muss, ohne die Quelle zu belasten.
- **Adapternetzwerke**: Wenn ein AE-Adapter von einem Sender bereitgestellt wird und mehrere Empfänger unabhängig voneinander darauf zugreifen sollen.

Vergleich mit ähnlichen Bausteinen

- **E_SPLIT-Bausteine**: Diese teilen ein Ereignissignal (z.B. in **E_SPLIT** oder **E_F_SPLIT**) auf mehrere Ereignisausgänge auf. Der **AE_SPLIT_6** hingegen arbeitet auf Adapterebene und verteilt den gesamten Adapterverbund, der sowohl Ereignisse als auch Daten enthalten kann – je nach konkreter Adapterinstanziierung.
- **AE_SPLIT_2 / AE_SPLIT_4**: Varianten mit zwei oder vier Ausgängen. Der **AE_SPLIT_6** bietet die maximale Anzahl von sechs Ausgängen in dieser Familie.
- **MUX/DEMUX-Bausteine**: Diese selektieren oder kombinieren Signale. Der **AE_SPLIT_6** hingegen vervielfältigt das Signal ohne Auswahl.

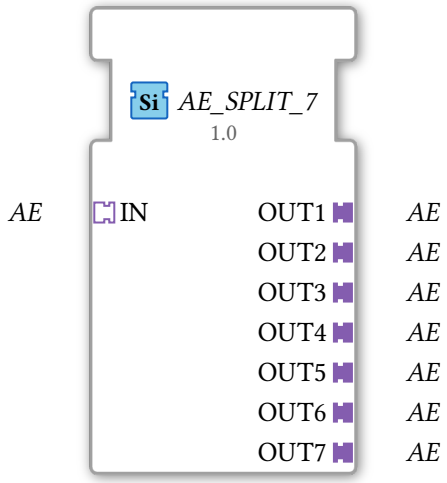
Fazit

Der **AE_SPLIT_6** ist ein einfacher, aber nützlicher generischer Funktionsblock zur 1:6-Verteilung eines unidirektionalen AE-Adapters. Durch seine generische Auslegung kann er in unterschiedlichsten Automatisierungs- und Steuerungsumgebungen eingesetzt werden, in denen ein Signal mehrfach benötigt wird. Er arbeitet ohne Verzögerung und ohne eigene Logik, was ihn besonders effizient und zuverlässig macht.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AE_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AE_SPLIT_7** dient dazu, ein einzelnes eingehendes Adapter-Ereignis (AE) auf sieben gleichartige Ausgangsadapter zu verteilen. Es handelt sich um einen generischen Baustein, der für unidirektionale Adapter des Typs `adapter::types::unidirectional::AE` ausgelegt ist. Der FB ermöglicht eine sternförmige Weiterleitung eines Ereignissignals an mehrere nachgelagerte Bausteine, ohne dass die Ereignisse verändert oder verzögert werden.

Schnittstellenstruktur

Ereignis-Eingänge

Keine – die Kommunikation erfolgt ausschließlich über Adapter.

Ereignis-Ausgänge

Keine – die Ausgabe erfolgt über Adapter.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AE`	**IN**	Socket	Eingangsadapter, der das zu verteilende Ereignis empfängt.
`adapter::types::unidirectional::AE`	**OUT1**	Plug	Erster Ausgangsadapter, das empfangene Ereignis wird hierhin weitergeleitet.
`adapter::types::unidirectional::AE`	**OUT2**	Plug	Zweiter Ausgangsadapter.
`adapter::types::unidirectional::AE`	**OUT3**	Plug	Dritter Ausgangsadapter.

adapter::types::unidirectional::AE	**OUT4**	Plug	Vierter Ausgangsadapter.
adapter::types::unidirectional::AE	**OUT5**	Plug	Fünfter Ausgangsadapter.
adapter::types::unidirectional::AE	**OUT6**	Plug	Sechster Ausgangsadapter.
adapter::types::unidirectional::AE	**OUT7**	Plug	Siebter Ausgangsadapter.

Funktionsweise

Der Baustein ist passiv und leitet ein ankommendes Ereignis am Eingangsadapter **IN** unverändert an alle sieben Ausgangsadapter **OUT1** bis **OUT7** weiter. Die Weiterleitung erfolgt gleichzeitig (logisch parallel), d.h. nach dem Empfang eines Ereignisses werden alle Ausgänge mit demselben Ereignis versorgt. Es findet keine Pufferung, Priorisierung oder Datenmanipulation statt. Da es sich um unidirektionale Adapter handelt, ist nur eine Richtung (vom Socket zum Plug) gegeben.

Technische Besonderheiten

- **Generischer Baustein:** Der FB ist als generischer Typ (`GEN_AE_SPLIT`) deklariert und kann in verschiedenen Kontexten mit dem entsprechenden Adaptertyp verwendet werden. Die Typüberprüfung erfolgt zur Entwurfszeit.
- **Keine Latenz:** Das Ereignis wird ohne Verzögerung durchgereicht; die Laufzeit hängt ausschließlich von der Infrastruktur des IEC 61499 Laufzeitsystems ab.
- **Typgleichheit:** Alle beteiligten Adapter müssen denselben Typ (`AE` – unidirectional) aufweisen. Eine Mischung unterschiedlicher Adaptertypen ist nicht zulässig.
- **Keine Zustandsspeicherung:** Der FB besitzt kein internes Gedächtnis – er führt lediglich eine 1:n-Weiterleitung durch.

Zustandsübersicht

Da der FB keine eigene Zustandsmaschine besitzt und ausschließlich reaktiv auf eingehende Ereignisse arbeitet, besteht nur ein einziger impliziter Zustand: **Bereit**. In diesem Zustand werden Ereignisse am Eingang sofort an alle Ausgänge weitergegeben. Es gibt keine Fehlerzustände oder Start/Stopp-Logik.

Anwendungsszenarien

- **Ereignisverteilung in Steuerungsanwendungen:** Ein einzelner Sensor oder Bus-Teilnehmer löst ein Ereignis aus, das gleichzeitig mehrere Aktoren, Alarme oder Überwachungsblöcke erreichen soll.
- **Parallele Verarbeitung:** Ein Ereignis soll in mehreren unabhängigen Funktionsblöcken gleichzeitig verarbeitet werden, z.B. für Redundanz oder unterschiedliche Auswertungen.
- **Strukturierung der Anwendung:** Ersetzt mehrere manuelle Verbindungsknoten und erhöht die Übersichtlichkeit im Netzwerkdiagramm.

Vergleich mit ähnlichen Bausteinen

- **AE_SPLIT_2, AE_SPLIT_3, … :** Diese Bausteine bieten die gleiche Funktionalität, jedoch mit einer geringeren Anzahl an Ausgängen. Die Wahl hängt von der benötigten Anzahl ab.
- **E_SPLIT (Ereignis-Splitter, datenlos):** Ein reiner Ereignis-Splitter (ohne Adapterkapselung) verteilt Ereignisse, kann aber nicht mit Adapter-Schnittstellen umgehen. Der AE_SPLIT_7 ist speziell für die Verwendung mit Adapter-Typen wie `AE` vorgesehen.
- **Adapter-Merger (z.B. AE_MERGE):** Führt mehrere Adapter-Ereignisse zu einem zusammen – das Gegenteil des vorliegenden Bausteins.

Fazit

Der **AE_SPLIT_7** ist ein einfacher, aber nützlicher Funktionsblock zur Vervielfältigung von Adapter-Ereignissen in IEC 61499 Anwendungen. Seine generische Natur, die vollständige Ereignistreue und die einfache Handhabung machen ihn zu einem grundlegenden Werkzeug für die strukturierte Ereignisverteilung. Er ist ideal geeignet, wenn ein Ereignis an mehrere gleichartige Zieladapter weitergeleitet werden muss und kein Dateninhalt transportiert wird.

AE_SPLIT_8

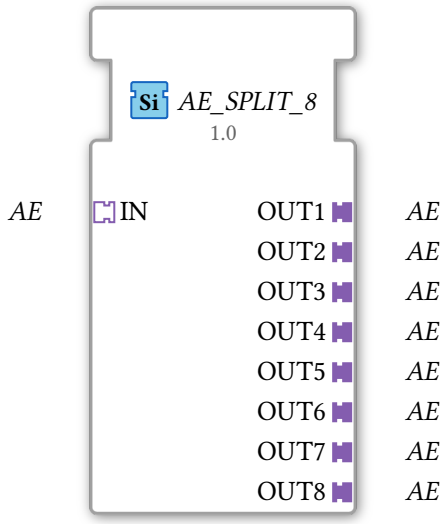


Bild des Funktionsblocks: [Hier einfügen, sofern vorhanden]

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AE_SPLIT_8** dient als generischer Verteiler für einen unidirektionalen Adapterereingang (AE) auf acht identische Adapterausgänge. Er ermöglicht die Mehrfachverwendung eines eingehenden Signals, ohne dieses zu modifizieren.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	`IN`	`adapter::types::unidirectional::AE`	Eingehender unidirektionaler AE-Adapter
Plug	`OUT1`	`adapter::types::unidirectional::AE`	Erster ausgehender AE-Adapter
Plug	`OUT2`	`adapter::types::unidirectional::AE`	Zweiter ausgehender AE-Adapter
Plug	`OUT3`	`adapter::types::unidirectional::AE`	Dritter ausgehender AE-Adapter
Plug	`OUT4`	`adapter::types::unidirectional::AE`	Vierter ausgehender AE-Adapter
Plug	`OUT5`	`adapter::types::unidirectional::AE`	Fünfter ausgehender AE-Adapter
Plug	`OUT6`	`adapter::types::unidirectional::AE`	Sechster ausgehender AE-Adapter
Plug	`OUT7`	`adapter::types::unidirectional::AE`	Siebter ausgehender AE-Adapter

Plug	OUT8	adapter::types::unidirectional::AE	Achter ausgehender AE-Adapter

Funktionsweise

Der Baustein leitet das am Socket IN anliegende Adaptersignal ohne Verzögerung oder Veränderung an alle acht Plugs (OUT1 bis OUT8) weiter. Es findet keine logische oder zeitliche Aufbereitung statt – der eingehende Adapter wird eins-zu-eins vervielfältigt.

Technische Besonderheiten

- **Generischer Typ**: Der FB ist als generischer Baustein (GenericClassName: GEN_AE_SPLIT) definiert und kann in verschiedenen Kontexten wiederverwendet werden.
- **Unidirektionalität**: Sowohl der Eingangs- als auch die Ausgangsadapter sind unidirektional ausgelegt. Eine Rückkopplung von den Ausgängen zum Eingang ist nicht möglich.
- **Keine Ereignissteuerung**: Der Baustein besitzt keine Ereigniseingänge und arbeitet rein passiv. Das Weiterleiten erfolgt sofort bei Verbindungsaufbau.
- **Skalierbarkeit**: Durch die feste Anzahl von acht Ausgängen ist der FB für Anwendungen optimiert, die genau eine 1:8-Verteilung benötigen.

Zustandsübersicht

Der FB besitzt keine internen Zustände oder Ablaufsteuerung. Es gibt keine ECC (Execution Control Chart) – die Funktionalität ist rein kombinatorisch.

Anwendungsszenarien

- **Signalverteilung in Steuerungsnetzwerken**: Ein von einem Sensor oder einer übergeordneten Steuerung kommender AE-Adapter soll gleichzeitig an mehrere nachgelagerte Funktionsbausteine weitergegeben werden.
- **Test- und Simulationsumgebungen**: Mehrere Empfänger sollen dasselbe Adaptersignal erhalten, um parallele Reaktionen zu beobachten.
- **Redundante Anbindung**: Das gleiche Signal kann auf mehreren Pfaden parallel verarbeitet werden, beispielsweise für Sicherheits- oder Überwachungsfunktionen.

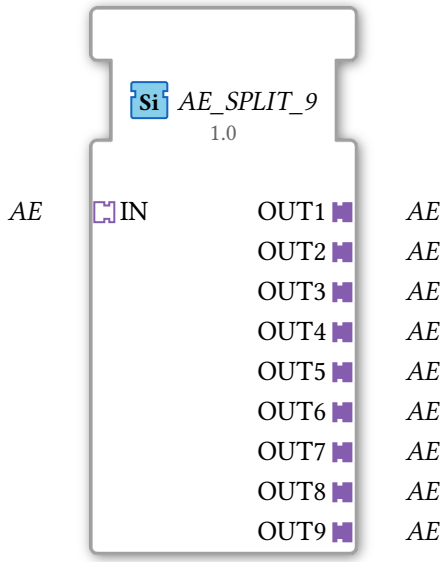
Vergleich mit ähnlichen Bausteinen

- **AE_SPLIT_2/AE_SPLIT_4**: Bausteine mit geringerer Anzahl an Ausgängen (2 bzw. 4) für kleinere Verteilungen. **AE_SPLIT_8** erweitert dies auf acht Ausgänge.
- **Daten-Splitter (z. B. SPLIT_INT, SPLIT_BOOL)**: Diese verteilen Datenwerte, während **AE_SPLIT_8** Adaptersignale verteilt. Adapter können komplexe zusammengesetzte Schnittstellen (Ereignisse und Daten) bündeln.
- **Ereignis-Splitter (z. B. E_SPLIT)**: Verteilen ausschließlich Ereignisse. **AE_SPLIT_8** überträgt den gesamten Adapterkontext, einschließlich aller darin enthaltenen Daten- und Ereignisverbindungen.

Fazit

Der **AE_SPLIT_8** ist ein einfacher und effizienter Baustein zur 1:8-Verteilung unidirektionaler Adaptersignale. Durch seine generische Natur und den Verzicht auf Ereignissteuerung eignet er sich besonders für Anwendungen, in denen ein Signal ohne Verzögerung an mehrere Empfänger parallel weitergeleitet werden muss.

AE_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AE_SPLIT_9 dient dazu, ein eingehendes unidirektionales Adapter-Event (AE) auf neun identische Ausgänge aufzuteilen. Er ist als generischer Baustein implementiert und ermöglicht die Verteilung von Ereignissen an mehrere nachgeschaltete Funktionsblöcke.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

- **IN** (Socket): Unidirektionaler AE-Adapter (Eingang)
- **OUT1 ... OUT9** (Plugs): Unidirektionale AE-Adapter (Ausgänge)

Funktionsweise

Der Baustein empfängt über den Socket „IN“ ein Adapter-Event. Jedes eintreffende Ereignis wird unverzögert und ohne weitere Logik auf alle neun Plug-Ausgänge (OUT1 bis OUT9) dupliziert. Somit verhält sich der FB wie ein 1:9-Splitter für AE-Signale.

Technische Besonderheiten

- Der Baustein ist als generischer FB (GenericClassName = ‚GEN_AE_SPLIT‘) implementiert, was eine flexible Wiederverwendung in unterschiedlichen Kontexten ermöglicht.
- Es werden keine Daten verarbeitet; ausschließlich das Adapter-Interface wird durchgeschliffen.
- Die Ausgänge sind voneinander unabhängig; ein Fehler an einem Ausgang beeinflusst die anderen nicht.

Zustandsübersicht

Der FB besitzt keinen internen Zustand. Er arbeitet rein kombinatorisch: Ein Ereignis am Eingang führt sofort zu Ereignissen an allen Ausgängen.

Anwendungsszenarien

- Verteilung eines Sensor-Ereignisses an mehrere Steuerungslogiken.
- Broadcast eines Alarms an verschiedene Überwachungseinheiten.
- Aufteilung eines Triggersignals für parallele Verarbeitungspfade.

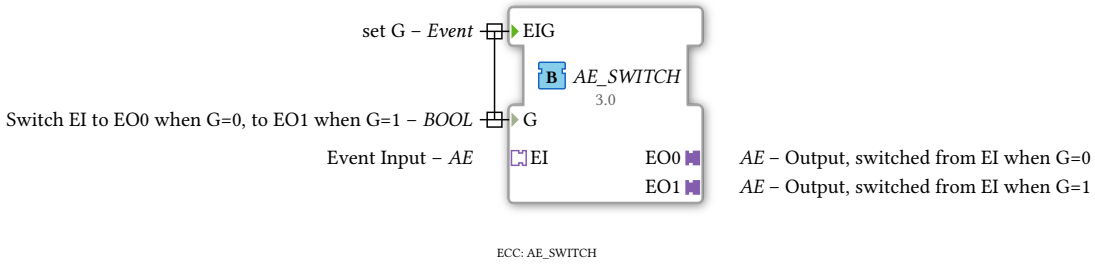
Vergleich mit ähnlichen Bausteinen

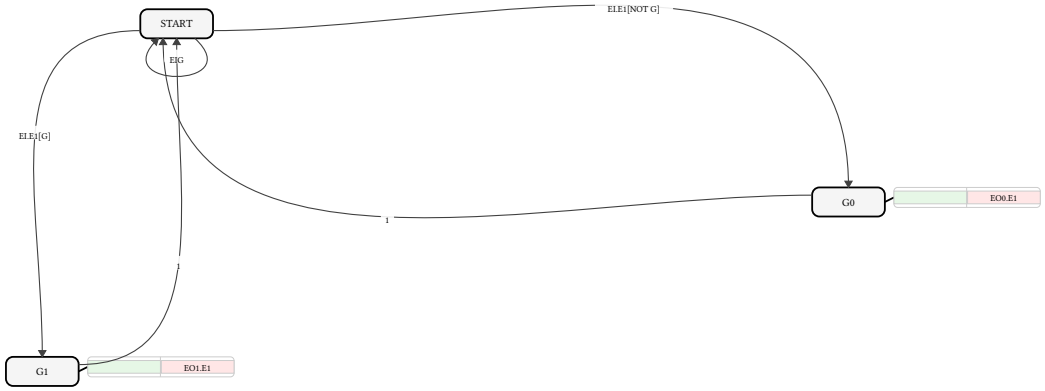
- AE_SPLIT_2, AE_SPLIT_4: gleiche Funktionalität mit weniger Ausgängen.
- Im Unterschied zu Daten-Splittern (z.B. SPLIT_INT) werden hier ausschließlich Ereignisse (ohne Daten) vervielfältigt.

Fazit

AE_SPLIT_9 ist ein einfacher, aber nützlicher Baustein zur Ereignisverteilung in IEC 61499-Systemen. Seine generische Natur und die klare 1:9-Aufteilung machen ihn ideal für Anwendungen, die eine breite Streuung von Adapter-Events erfordern.

AE_SWITCH





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AE_SWITCH** dient als Demultiplexer für Ereignisse. Er leitet ein eingehendes Ereignis je nach dem Wert eines booleschen Eingangs `G` an einen von zwei unidirektionalen Event-Adapter-Ausgängen weiter.

Schnittstellenstruktur

Ereignis-Eingänge

- **EIG** (Event): Setzt den Wert von `G`. Wird dieses Ereignis ausgelöst, wird der zugehörige Daten-Eingang `G` übernommen – es erfolgt keine Weiterleitung an die Ausgänge.

Ereignis-Ausgänge

- Keine eigenständigen Ereignis-Ausgänge vorhanden. Die Ausgabe erfolgt ausschließlich über die Adapter `EO0` und `EO1`.

Daten-Eingänge

Name	Typ	Kommentar
`G`	BOOL	Schalter: 0 ⟶ Ausgabe über `EO0`, 1 ⟶ Ausgabe über `EO1`

Daten-Ausgänge

- Keine Datenausgänge.

Adapter

Richtung	Name	Typ (unidirektional)	Kommentar
Socket (Eingang)	`EI`	`adapter::types::unidirectional::AE`	Ereignis-Eingang, der umgeschaltet wird.
Plug (Ausgang)	`EO0`	`adapter::types::unidirectional::AE`	Ereignis-Ausgang, aktiv bei `G = 0`
Plug (Ausgang)	`EO1`	`adapter::types::unidirectional::AE`	Ereignis-Ausgang, aktiv bei `G = 1`

Funktionsweise

Der Baustein befindet sich initial im Zustand **START**.

1. **Setzen des Schalters**: Ein Ereignis an `EIG` übernimmt den aktuellen Wert von `G`. Der Baustein bleibt danach im Zustand **START**.
2. **Weiterleiten des Eingangsereignisses**: Trifft ein Ereignis über den Adapter-Eingang `EI` ein, wird der Wert von `G` ausgewertet:
   - Ist `G = 0` ⟶ Zustandswechsel nach **G0**: Der Adapter-Ausgang `EO0` wird mit seinem Ereignis `E1` ausgelöst.
   - Ist `G = 1` ⟶ Zustandswechsel nach **G1**: Der Adapter-Ausgang `EO1` wird mit seinem Ereignis `E1` ausgelöst.
3. Nach Auslösen des jeweiligen Ausgangs kehrt der Baustein automatisch in den Zustand **START** zurück.

Technische Besonderheiten

- Die Ereignisweiterleitung erfolgt ausschließlich über **unidirektionale Adapter** vom Typ `AE`. Dadurch wird eine saubere Trennung von Ereignisquellen und -senken erreicht.
- Der Baustein realisiert ein **Demultiplexing** (1-zu-2) auf Ereignisebene: Ein eingehendes Ereignis wird je nach boolescher Bedingung an einen von zwei Ausgängen geroutet.
- Der Setz-Eingang `EIG` erlaubt es, den Schaltwert unabhängig von einem anstehenden Ereignis am Eingang `EI` zu ändern.

Zustandsübersicht

Zustand	Beschreibung	Aktion bei Eintritt
START	Warten auf ein Ereignis an `EI` oder `EIG`	–
G0	Ereignis an `EI` mit `G = 0` – leite an `EO0` weiter	`EO0.E1`
G1	Ereignis an `EI` mit `G = 1` – leite an `EO1` weiter	`EO1.E1`

Transitionen:

- START ⟶ G0 : `EI.E1[NOT G]`
- START ⟶ G1 : `EI.E1[G]`
- G0 ⟶ START : `1` (sofort)
- G1 ⟶ START : `1` (sofort)
- START ⟶ START : `EIG` (nur Setzen von G, keine Weiterleitung)

Anwendungsszenarien

- **Ereignis-Routing** in Automatisierungssystemen, bei dem ein zentraler Auslöser (z. B. ein Sensorereignis) je nach Betriebsmodus unterschiedliche Aktoren ansteuert.
- **Zustandsabhängige Verarbeitung**: Ein eingehendes Ereignis soll nur in einer bestimmten Betriebsart verarbeitet werden; die andere Betriebsart erhält ein separates Ereignis.
- **Test- und Diagnoseaufgaben**: Umschalten zwischen Normalbetrieb und Testmodus durch Setzen von `G`.

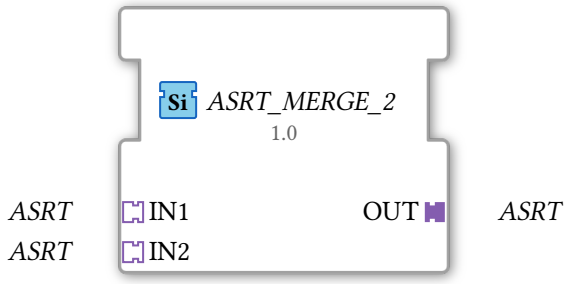
Vergleich mit ähnlichen Bausteinen

- **E_SWITCH** (Standard 61499): Ähnliche Funktionalität, jedoch über reguläre Ereignisausgänge. `AE_SWITCH` verwendet Adapter, was eine bessere Kapselung und Wiederverwendbarkeit in Komponenten mit standardisierten Schnittstellen ermöglicht.
- **E_MUX** (Multiplexer): Führt mehrere Ereignisse zu einem Ausgang zusammen – gegensätzliche Richtung. `AE_SWITCH` demultiplext ein Eingangsereignis auf zwei Ausgänge.

Fazit

Der `AE_SWITCH` bietet eine schlanke, adapterbasierte Lösung zum bedingten Weiterleiten von Ereignissen. Durch die Verwendung unidirektionaler Adapter ist er ideal für modulare und typsichere IEC-61499-Anwendungen geeignet. Seine einfache Zustandsmaschine macht ihn effizient und leicht verständlich.

ASRT_MERGE_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASRT_MERGE_2** führt zwei unidirektionale ASRT-Adapter (Set/Reset/Toggle-Ereignistripel, keine Nutzdaten) auf einen gemeinsamen Ausgangsadapter zusammen. `ASRT` erweitert `ASR` um ein drittes Ereignis `TOGGLE` (siehe `AX_T_FF_SR`). Der Baustein ist die Umkehrung von ASRT_SPLIT_2 und als generischer FB implementiert (`GenericClassName: 'GEN_ASRT_MERGE'`).

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignisübernahme erfolgt ausschließlich über die Adapter-Sockets.

Ereignis-Ausgänge

Keine. Die Ereignisweitergabe erfolgt ausschließlich über den Adapter-Plug.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket (Eingang 1)	IN1	adapter::types::unidirectional::ASRT	Erste SET/RESET/TOGGLE-Quelle.
Socket (Eingang 2)	IN2	adapter::types::unidirectional::ASRT	Zweite SET/RESET/TOGGLE-Quelle.
Plug (Ausgang)	OUT	adapter::types::unidirectional::ASRT	Zusammengeführtes SET/RESET/TOGGLE-Signal.

Funktionsweise

Trifft ein SET-, RESET- oder TOGGLE-Ereignis an IN1 **oder** IN2 ein, wird es als gleichartiges Ereignis an OUT weitergereicht. Alle drei Ereignistypen werden unabhängig voneinander behandelt – welches Ereignis an welchem Socket eintrifft, hat keinen Einfluss auf die anderen beiden Ereignistypen oder den jeweils anderen Socket. Da ASRT keinen Datenwert transportiert, gibt es nichts zu arbitrieren außer dem jeweiligen Ereignistyp selbst.

Technische Besonderheiten

- **Generischer Typ**: Implementiert über die generische Basisklasse CGenUnidirectMergeBase (N Sockets, 1 Plug) – dieselbe C++-Basis, die auch GEN_AE_MERGE und GEN_ASR_MERGE zugrunde liegt.
- **Drei unabhängige Ereigniskanäle**: SET, RESET und TOGGLE werden getrennt geprüft und weitergeleitet.
- **Keine Zustände / Algorithmen**: Da ASRT keinen Datenwert trägt, gibt es keine Änderungserkennung und kein ECC.

Zustandsübersicht

Der Baustein besitzt **keinen Zustandsautomaten**. Das Verhalten ist rein kombinatorisch: Jedes eingehende SET-, RESET- oder TOGGLE-Ereignis an IN1 oder IN2 wird unmittelbar als gleichartiges Ereignis an OUT weitergereicht.

Anwendungsszenarien

- **Zusammenführen redundanter Set/Reset/Toggle-Quellen**: z. B. zwei Bedienstellen, die dasselbe AX_T_FF_SR-gesteuerte Element über einen gemeinsamen ASRT-Adapter ansteuern sollen.
- **Vereinfachung von Netzwerken**, die sonst drei separate Ereignisverbindungspaare (SET, RESET, TOGGLE) je Quelle zum selben Ziel benötigen würden.

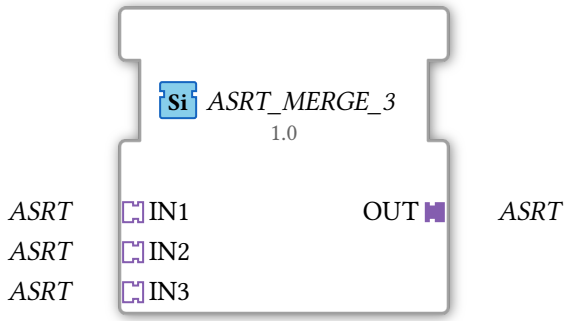
Vergleich mit ähnlichen Bausteinen

- ASRT_SPLIT_2: die Umkehrrichtung – verteilt ein eingehendes Set/Reset/Toggle-Signal auf zwei Ausgänge, statt zwei Eingänge zusammenzuführen.
- ASR_MERGE_2: dieselbe Zusammenführungslogik für den Set/Reset-Ereignisadapter ASR (ohne TOGGLE).
- AE_MERGE_2: dieselbe Zusammenführungslogik für den reinen Ereignisadapter AE (1 Ereignis statt SET/RESET/TOGGLE).

Fazit

Der **ASRT_MERGE_2** ist ein minimaler generischer Baustein zur Zusammenführung zweier Set/Reset/Toggle-Ereignisquellen auf einen gemeinsamen ASRT-Adapterausgang und komplettiert die GEN_*_MERGE-Familie für alle drei unidirektionalen Ereignisadapter (AE, ASR, ASRT).

ASRT_MERGE_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASRT_MERGE_3** führt 3 unidirektionale ASRT-Adapter (Set/Reset/Toggle-Ereignisadapter, keine Nutzdaten) auf einen gemeinsamen Ausgangsadapter zusammen. Er ist die 3-Eingangs-Variante von ASRT_MERGE_2 und wie dieser als generischer FB implementiert (`GenericClassName: 'GEN_ASRT_MERGE'`) — dieselbe C++-Basis (`CGenUnidirectMergeBase`), nur mit 3 statt 2 Sockets.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignisübernahme erfolgt ausschließlich über die Adapter-Sockets.

Ereignis-Ausgänge

Keine. Die Ereignisweitergabe erfolgt ausschließlich über den Adapter-Plug.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket (Eingang 1)	`IN1`	`adapter::types::unidirectional::ASRT`	1. Quelle.
Socket (Eingang 2)	`IN2`	`adapter::types::unidirectional::ASRT`	2. Quelle.
Socket (Eingang 3)	`IN3`	`adapter::types::unidirectional::ASRT`	3. Quelle.
Plug (Ausgang)	`OUT`	`adapter::types::unidirectional::ASRT`	Zusammengeführtes Signal.

Funktionsweise

Trifft `SET` bzw. `RESET` bzw. `TOGGLE` an einem der 3 Sockets (`IN1`, `IN2`, `IN3`) ein, wird es als gleichartiges Ereignis unverändert an `OUT` weitergereicht. Alle Ereignistypen werden unabhängig voneinander behandelt, und alle 3 Sockets werden gleichberechtigt behandelt — welcher Socket zuerst geprüft wird, folgt der Reihenfolge `IN1` … `IN3`, was nur bei mehreren innerhalb desselben Ausführungszyklus eintreffenden Ereignissen relevant wird. Da `ASRT` keinen Datenwert transportiert, gibt es nichts zu arbitrieren außer dem jeweiligen Ereignistyp selbst.

Technische Besonderheiten

- **Generischer Typ**: Implementiert über die generische Basisklasse `CGenUnidirectMergeBase` (N Sockets, 1 Plug); die Anzahl der Eingangs-Sockets wird über den generischen Suffix (`_3`) zur Übersetzungszeit festgelegt.
- **Keine Zustände / Algorithmen**: Da `ASRT` keinen Datenwert trägt, gibt es keine Änderungserkennung und kein ECC.

- **Beliebige Eingangsanzahl**: Dieselbe Implementierung deckt ASRT_MERGE_2 bis ASRT_MERGE_7 ab; für andere Eingangsanzahlen siehe `ASRT_MERGE_2`, `ASRT_MERGE_4`, `ASRT_MERGE_5`, `ASRT_MERGE_6`, `ASRT_MERGE_7`.

Zustandsübersicht

Der Baustein besitzt **keinen Zustandsautomaten**. Das Verhalten ist rein kombinatorisch: Jedes eingehende Ereignis an einem der 3 Sockets wird unmittelbar an `OUT` weitergereicht.

Anwendungsszenarien

- **Zusammenführen mehrerer gleichwertiger Signalquellen** (z. B. 3 redundante oder alternative Auslösepfade) auf einen gemeinsamen ASRT-Adapterausgang.
- **Vereinfachung von Netzwerken**, die sonst 3 separate Ereignisverbindungen zum selben Ziel benötigen würden.

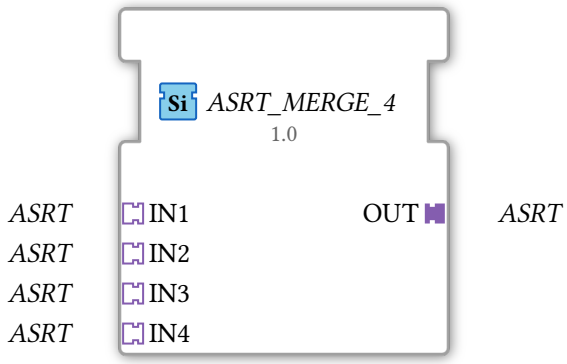
Vergleich mit ähnlichen Bausteinen

- ASRT_SPLIT_2: die Umkehrrichtung für 2 Ausgänge (bei ASRT zusätzlich ASRT_SPLIT_3 bis ASRT_SPLIT_9 verfügbar).
- **`ASRT_MERGE_2`, `ASRT_MERGE_4`, `ASRT_MERGE_5`, `ASRT_MERGE_6`, `ASRT_MERGE_7`**: dieselbe generische Implementierung mit anderer Eingangsanzahl.
- AE_MERGE_3, ASR_MERGE_3: dieselbe Zusammenführungslogik für die jeweils anderen unidirektionalen Ereignisadapter.

Fazit

`ASRT_MERGE_3` liefert eine generisch implementierte Zusammenführung von 3 `ASRT`-Ereignisquellen auf einen gemeinsamen Adapterausgang und ergänzt die `GEN_ASRT_MERGE`-Familie um die 3-Eingangs-Variante.

ASRT_MERGE_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASRT_MERGE_4** führt 4 unidirektionale ASRT-Adapter (Set/Reset/Toggle-Ereignisadapter, keine Nutzdaten) auf einen gemeinsamen Ausgangsadapter zusammen. Er ist die 4-Eingangs-Variante von ASRT_MERGE_2 und wie dieser als generischer FB implementiert (`GenericClassName: 'GEN_ASRT_MERGE'`) — dieselbe C++-Basis (`CGenUnidirectMergeBase`), nur mit 4 statt 2 Sockets.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignisübernahme erfolgt ausschließlich über die Adapter-Sockets.

Ereignis-Ausgänge

Keine. Die Ereignisweitergabe erfolgt ausschließlich über den Adapter-Plug.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket (Eingang 1)	`IN1`	`adapter::types::unidirectional::ASRT`	1. Quelle.
Socket (Eingang 2)	`IN2`	`adapter::types::unidirectional::ASRT`	2. Quelle.
Socket (Eingang 3)	`IN3`	`adapter::types::unidirectional::ASRT`	3. Quelle.
Socket (Eingang 4)	`IN4`	`adapter::types::unidirectional::ASRT`	4. Quelle.
Plug (Ausgang)	`OUT`	`adapter::types::unidirectional::ASRT`	Zusammengeführtes Signal.

Funktionsweise

Trifft `SET` bzw. `RESET` bzw. `TOGGLE` an einem der 4 Sockets (`IN1`, `IN2`, `IN3`, `IN4`) ein, wird es als gleichartiges Ereignis unverändert an `OUT` weitergereicht. Alle Ereignistypen werden unabhängig voneinander behandelt, und alle 4 Sockets werden gleichberechtigt behandelt — welcher Socket zuerst geprüft wird, folgt der Reihenfolge `IN1` … `IN4`, was nur bei mehreren innerhalb desselben Ausführungszyklus eintreffenden Ereignissen relevant wird. Da `ASRT` keinen Datenwert transportiert, gibt es nichts zu arbitrieren außer dem jeweiligen Ereignistyp selbst.

Technische Besonderheiten

- **Generischer Typ**: Implementiert über die generische Basisklasse `CGenUnidirectMergeBase` (N Sockets, 1 Plug); die Anzahl der Eingangs-Sockets wird über den generischen Suffix (`_4`) zur Übersetzungszeit festgelegt.
- **Keine Zustände / Algorithmen**: Da `ASRT` keinen Datenwert trägt, gibt es keine Änderungserkennung und kein ECC.
- **Beliebige Eingangsanzahl**: Dieselbe Implementierung deckt ASRT_MERGE_2 bis ASRT_MERGE_7 ab; für andere Eingangsanzahlen siehe `ASRT_MERGE_2`, `ASRT_MERGE_3`, `ASRT_MERGE_5`, `ASRT_MERGE_6`, `ASRT_MERGE_7`.

Zustandsübersicht

Der Baustein besitzt **keinen Zustandsautomaten**. Das Verhalten ist rein kombinatorisch: Jedes eingehende Ereignis an einem der 4 Sockets wird unmittelbar an `OUT` weitergereicht.

Anwendungsszenarien

- **Zusammenführen mehrerer gleichwertiger Signalquellen** (z. B. 4 redundante oder alternative Auslösepfade) auf einen gemeinsamen ASRT-Adapterausgang.
- **Vereinfachung von Netzwerken**, die sonst 4 separate Ereignisverbindungen zum selben Ziel benötigen würden.

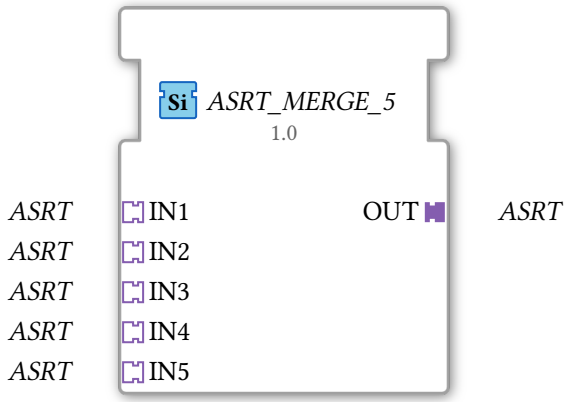
Vergleich mit ähnlichen Bausteinen

- **ASRT_SPLIT_2**: die Umkehrrichtung für 2 Ausgänge (bei ASRT zusätzlich ASRT_SPLIT_3 bis ASRT_SPLIT_9 verfügbar).
- **`ASRT_MERGE_2`, `ASRT_MERGE_3`, `ASRT_MERGE_5`, `ASRT_MERGE_6`, `ASRT_MERGE_7`**: dieselbe generische Implementierung mit anderer Eingangsanzahl.
- AE_MERGE_4, ASR_MERGE_4: dieselbe Zusammenführungslogik für die jeweils anderen unidirektionalen Ereignisadapter.

Fazit

`ASRT_MERGE_4` liefert eine generisch implementierte Zusammenführung von 4 `ASRT`-Ereignisquellen auf einen gemeinsamen Adapterausgang und ergänzt die `GEN_ASRT_MERGE`-Familie um die 4-Eingangs-Variante.

ASRT_MERGE_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASRT_MERGE_5** führt 5 unidirektionale ASRT-Adapter (Set/Reset/Toggle-Ereignisadapter, keine Nutzdaten) auf einen gemeinsamen Ausgangsadapter zusammen. Er ist die 5-Eingangs-Variante von ASRT_MERGE_2 und wie dieser als generischer FB implementiert (`GenericClassName: 'GEN_ASRT_MERGE'`) — dieselbe C++-Basis (`CGenUnidirectMergeBase`), nur mit 5 statt 2 Sockets.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignisübernahme erfolgt ausschließlich über die Adapter-Sockets.

Ereignis-Ausgänge

Keine. Die Ereignisweitergabe erfolgt ausschließlich über den Adapter-Plug.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket (Eingang 1)	`IN1`	`adapter::types::unidirectional::ASRT`	1. Quelle.
Socket (Eingang 2)	`IN2`	`adapter::types::unidirectional::ASRT`	2. Quelle.
Socket (Eingang 3)	`IN3`	`adapter::types::unidirectional::ASRT`	3. Quelle.
Socket (Eingang 4)	`IN4`	`adapter::types::unidirectional::ASRT`	4. Quelle.
Socket (Eingang 5)	`IN5`	`adapter::types::unidirectional::ASRT`	5. Quelle.
Plug (Ausgang)	`OUT`	`adapter::types::unidirectional::ASRT`	Zusammengeführtes Signal.

Funktionsweise

Trifft `SET` bzw. `RESET` bzw. `TOGGLE` an einem der 5 Sockets (`IN1`, `IN2`, `IN3`, `IN4`, `IN5`) ein, wird es als gleichartiges Ereignis unverändert an `OUT` weitergereicht. Alle Ereignistypen werden unabhängig voneinander behandelt, und alle 5 Sockets werden gleichberechtigt behandelt — welcher Socket zuerst geprüft wird, folgt der Reihenfolge `IN1` … `IN5`, was nur bei mehreren innerhalb desselben Ausführungszyklus eintreffenden Ereignissen relevant wird. Da `ASRT` keinen Datenwert transportiert, gibt es nichts zu arbitrieren außer dem jeweiligen Ereignistyp selbst.

Technische Besonderheiten

- **Generischer Typ**: Implementiert über die generische Basisklasse `CGenUnidirectMergeBase` (N Sockets, 1 Plug); die Anzahl der Eingangs-Sockets wird über den generischen Suffix (`_5`) zur Übersetzungszeit festgelegt.
- **Keine Zustände / Algorithmen**: Da `ASRT` keinen Datenwert trägt, gibt es keine Änderungserkennung und kein ECC.
- **Beliebige Eingangsanzahl**: Dieselbe Implementierung deckt ASRT_MERGE_2 bis ASRT_MERGE_7 ab; für andere Eingangsanzahlen siehe `ASRT_MERGE_2`, `ASRT_MERGE_3`, `ASRT_MERGE_4`, `ASRT_MERGE_6`, `ASRT_MERGE_7`.

Zustandsübersicht

Der Baustein besitzt **keinen Zustandsautomaten**. Das Verhalten ist rein kombinatorisch: Jedes eingehende Ereignis an einem der 5 Sockets wird unmittelbar an `OUT` weitergereicht.

Anwendungsszenarien

- **Zusammenführen mehrerer gleichwertiger Signalquellen** (z. B. 5 redundante oder alternative Auslösepfade) auf einen gemeinsamen ASRT-Adapterausgang.
- **Vereinfachung von Netzwerken**, die sonst 5 separate Ereignisverbindungen zum selben Ziel benötigen würden.

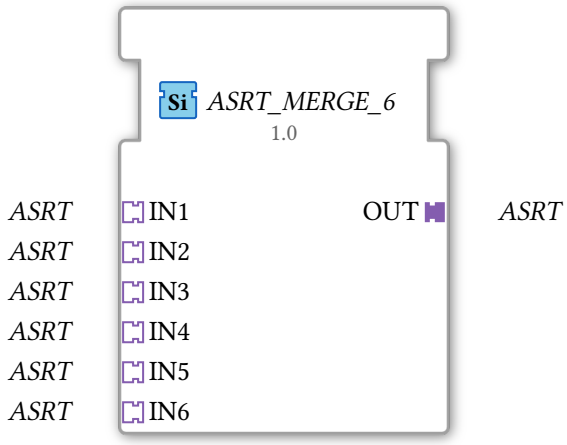
Vergleich mit ähnlichen Bausteinen

- ASRT_SPLIT_2: die Umkehrrichtung für 2 Ausgänge (bei ASRT zusätzlich ASRT_SPLIT_3 bis ASRT_SPLIT_9 verfügbar).
- **`ASRT_MERGE_2`, `ASRT_MERGE_3`, `ASRT_MERGE_4`, `ASRT_MERGE_6`, `ASRT_MERGE_7`**: dieselbe generische Implementierung mit anderer Eingangsanzahl.
- AE_MERGE_5, ASR_MERGE_5: dieselbe Zusammenführungslogik für die jeweils anderen unidirektionalen Ereignisadapter.

Fazit

`ASRT_MERGE_5` liefert eine generisch implementierte Zusammenführung von 5 `ASRT`-Ereignisquellen auf einen gemeinsamen Adapterausgang und ergänzt die `GEN_ASRT_MERGE`-Familie um die 5-Eingangs-Variante.

ASRT_MERGE_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASRT_MERGE_6** führt 6 unidirektionale ASRT-Adapter (Set/Reset/Toggle-Ereignisadapter, keine Nutzdaten) auf einen gemeinsamen Ausgangsadapter zusammen. Er ist die 6-Eingangs-Variante von ASRT_MERGE_2 und wie dieser als generischer FB implementiert (`GenericClassName: 'GEN_ASRT_MERGE'`) — dieselbe C++-Basis (`CGenUnidirectMergeBase`), nur mit 6 statt 2 Sockets.

Schnittstellenstruktur

Ereignis-Eingänge
Keine. Die Ereignisübernahme erfolgt ausschließlich über die Adapter-Sockets.

Ereignis-Ausgänge
Keine. Die Ereignisweitergabe erfolgt ausschließlich über den Adapter-Plug.

Daten-Eingänge
Keine.

Daten-Ausgänge
Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket (Eingang 1)	`IN1`	`adapter::types::unidirectional::ASRT`	1. Quelle.
Socket (Eingang 2)	`IN2`	`adapter::types::unidirectional::ASRT`	2. Quelle.
Socket (Eingang 3)	`IN3`	`adapter::types::unidirectional::ASRT`	3. Quelle.
Socket (Eingang 4)	`IN4`	`adapter::types::unidirectional::ASRT`	4. Quelle.
Socket (Eingang 5)	`IN5`	`adapter::types::unidirectional::ASRT`	5. Quelle.
Socket (Eingang 6)	`IN6`	`adapter::types::unidirectional::ASRT`	6. Quelle.
Plug (Ausgang)	`OUT`	`adapter::types::unidirectional::ASRT`	Zusammengeführtes Signal.

Funktionsweise
Trifft `SET` bzw. `RESET` bzw. `TOGGLE` an einem der 6 Sockets (`IN1`, `IN2`, `IN3`, `IN4`, `IN5`, `IN6`) ein, wird es als gleichartiges Ereignis unverändert an `OUT` weitergereicht. Alle Ereignistypen werden unabhängig voneinander behandelt, und alle 6 Sockets werden gleichberechtigt behandelt — welcher Socket zuerst geprüft wird, folgt der Reihenfolge `IN1` … `IN6`, was nur bei mehreren innerhalb desselben Ausführungszyklus eintreffenden Ereignissen relevant wird. Da `ASRT` keinen Datenwert transportiert, gibt es nichts zu arbitrieren außer dem jeweiligen Ereignistyp selbst.

Technische Besonderheiten
- **Generischer Typ**: Implementiert über die generische Basisklasse `CGenUnidirectMergeBase` (N Sockets, 1 Plug); die Anzahl der Eingangs-Sockets wird über den generischen Suffix (`_6`) zur Übersetzungszeit festgelegt.
- **Keine Zustände / Algorithmen**: Da `ASRT` keinen Datenwert trägt, gibt es keine Änderungserkennung und kein ECC.
- **Beliebige Eingangsanzahl**: Dieselbe Implementierung deckt ASRT_MERGE_2 bis ASRT_MERGE_7 ab; für andere Eingangsanzahlen siehe `ASRT_MERGE_2`, `ASRT_MERGE_3`, `ASRT_MERGE_4`, `ASRT_MERGE_5`, `ASRT_MERGE_7`.

Zustandsübersicht
Der Baustein besitzt **keinen Zustandsautomaten**. Das Verhalten ist rein kombinatorisch: Jedes eingehende Ereignis an einem der 6 Sockets wird unmittelbar an `OUT` weitergereicht.

Anwendungsszenarien
- **Zusammenführen mehrerer gleichwertiger Signalquellen** (z. B. 6 redundante oder alternative Auslösepfade) auf einen gemeinsamen ASRT-Adapterausgang.
- **Vereinfachung von Netzwerken**, die sonst 6 separate Ereignisverbindungen zum selben Ziel benötigen würden.

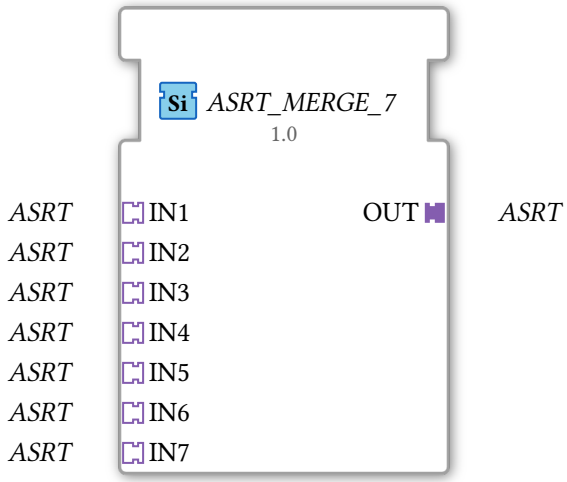
Vergleich mit ähnlichen Bausteinen

- **ASRT_SPLIT_2**: die Umkehrrichtung für 2 Ausgänge (bei ASRT zusätzlich ASRT_SPLIT_3 bis ASRT_SPLIT_9 verfügbar).
- **`ASRT_MERGE_2`, `ASRT_MERGE_3`, `ASRT_MERGE_4`, `ASRT_MERGE_5`, `ASRT_MERGE_7`**: dieselbe generische Implementierung mit anderer Eingangsanzahl.
- AE_MERGE_6, ASR_MERGE_6: dieselbe Zusammenführungslogik für die jeweils anderen unidirektionalen Ereignisadapter.

Fazit

`ASRT_MERGE_6` liefert eine generisch implementierte Zusammenführung von 6 `ASRT`-Ereignisquellen auf einen gemeinsamen Adapterausgang und ergänzt die `GEN_ASRT_MERGE`-Familie um die 6-Eingangs-Variante.

ASRT_MERGE_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASRT_MERGE_7** führt 7 unidirektionale ASRT-Adapter (Set/Reset/Toggle-Ereignisadapter, keine Nutzdaten) auf einen gemeinsamen Ausgangsadapter zusammen. Er ist die 7-Eingangs-Variante von ASRT_MERGE_2 und wie dieser als generischer FB implementiert (`GenericClassName: 'GEN_ASRT_MERGE'`) — dieselbe C++-Basis (`CGenUnidirectMergeBase`), nur mit 7 statt 2 Sockets.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignisübernahme erfolgt ausschließlich über die Adapter-Sockets.

Ereignis-Ausgänge

Keine. Die Ereignisweitergabe erfolgt ausschließlich über den Adapter-Plug.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket (Eingang 1)	`IN1`	`adapter::types::unidirectional::ASRT`	1. Quelle.
Socket (Eingang 2)	`IN2`	`adapter::types::unidirectional::ASRT`	2. Quelle.

Socket (Eingang 3)	IN3	adapter::types::unidirectional::ASRT	3.	Quelle.
Socket (Eingang 4)	IN4	adapter::types::unidirectional::ASRT	4.	Quelle.
Socket (Eingang 5)	IN5	adapter::types::unidirectional::ASRT	5.	Quelle.
Socket (Eingang 6)	IN6	adapter::types::unidirectional::ASRT	6.	Quelle.
Socket (Eingang 7)	IN7	adapter::types::unidirectional::ASRT	7.	Quelle.
Plug (Ausgang)	OUT	adapter::types::unidirectional::ASRT	Zusammengeführtes Signal.	

Funktionsweise

Trifft SET bzw. RESET bzw. TOGGLE an einem der 7 Sockets (IN1, IN2, IN3, IN4, IN5, IN6, IN7) ein, wird es als gleichartiges Ereignis unverändert an OUT weitergereicht. Alle Ereignistypen werden unabhängig voneinander behandelt, und alle 7 Sockets werden gleichberechtigt behandelt — welcher Socket zuerst geprüft wird, folgt der Reihenfolge IN1 … IN7, was nur bei mehreren innerhalb desselben Ausführungszyklus eintreffenden Ereignissen relevant wird. Da ASRT keinen Datenwert transportiert, gibt es nichts zu arbitrieren außer dem jeweiligen Ereignistyp selbst.

Technische Besonderheiten

- **Generischer Typ**: Implementiert über die generische Basisklasse CGenUnidirectMergeBase (N Sockets, 1 Plug); die Anzahl der Eingangs-Sockets wird über den generischen Suffix (_7) zur Übersetzungszeit festgelegt.
- **Keine Zustände / Algorithmen**: Da ASRT keinen Datenwert trägt, gibt es keine Änderungserkennung und kein ECC.
- **Beliebige Eingangsanzahl**: Dieselbe Implementierung deckt ASRT_MERGE_2 bis ASRT_MERGE_7 ab; für andere Eingangsanzahlen siehe ASRT_MERGE_2, ASRT_MERGE_3, ASRT_MERGE_4, ASRT_MERGE_5, ASRT_MERGE_6.

Zustandsübersicht

Der Baustein besitzt **keinen Zustandsautomaten**. Das Verhalten ist rein kombinatorisch: Jedes eingehende Ereignis an einem der 7 Sockets wird unmittelbar an OUT weitergereicht.

Anwendungsszenarien

- **Zusammenführen mehrerer gleichwertiger Signalquellen** (z. B. 7 redundante oder alternative Auslösepfade) auf einen gemeinsamen ASRT-Adapterausgang.
- **Vereinfachung von Netzwerken**, die sonst 7 separate Ereignisverbindungen zum selben Ziel benötigen würden.

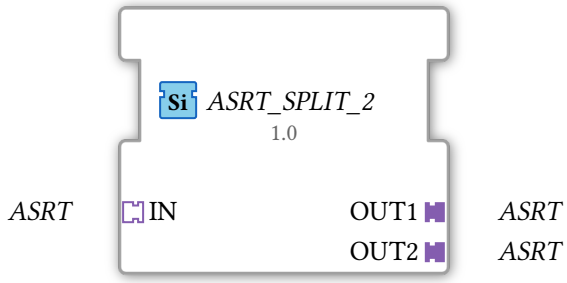
Vergleich mit ähnlichen Bausteinen

- ASRT_SPLIT_2: die Umkehrrichtung für 2 Ausgänge (bei ASRT zusätzlich ASRT_SPLIT_3 bis ASRT_SPLIT_9 verfügbar).
- **ASRT_MERGE_2, ASRT_MERGE_3, ASRT_MERGE_4, ASRT_MERGE_5, ASRT_MERGE_6**: dieselbe generische Implementierung mit anderer Eingangsanzahl.
- AE_MERGE_7, ASR_MERGE_7: dieselbe Zusammenführungslogik für die jeweils anderen unidirektionalen Ereignisadapter.

Fazit

ASRT_MERGE_7 liefert eine generisch implementierte Zusammenführung von 7 ASRT-Ereignisquellen auf einen gemeinsamen Adapterausgang und ergänzt die GEN_ASRT_MERGE-Familie um die 7-Eingangs-Variante.

ASRT_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASRT_SPLIT_2** verteilt einen eingehenden unidirektionalen ASRT-Adapter (Set/Reset/Toggle-Ereignistripel, keine Nutzdaten) auf zwei identische Ausgangsadapter. `ASRT` erweitert `ASR` um ein drittes Ereignis `TOGGLE` (siehe `AX_T_FF_SR`). Er ist als generischer FB implementiert (`GenericClassName: 'GEN_ASRT_SPLIT'`) und komplettiert die `SPLIT`-Familie neben AE_SPLIT_2 und ASR_SPLIT_2.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignisübernahme erfolgt ausschließlich über den Adapter-Socket.

Ereignis-Ausgänge

Keine. Die Ereignisweitergabe erfolgt ausschließlich über die Adapter-Plugs.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket (Eingang)	`IN`	`adapter::types::unidirectional::ASRT`	Empfängt ein SET-, RESET- oder TOGGLE-Ereignis, das auf beide Ausgänge verteilt wird.
Plug (Ausgang 1)	`OUT1`	`adapter::types::unidirectional::ASRT`	Erster Ausgang für das duplizierte Ereignis.
Plug (Ausgang 2)	`OUT2`	`adapter::types::unidirectional::ASRT`	Zweiter Ausgang für das duplizierte Ereignis.

Funktionsweise

Sobald am Adapter-Socket `IN` ein `SET`-, `RESET`- oder `TOGGLE`-Ereignis eintrifft, wird es **sofort und parallel** als gleichartiges Ereignis an beide Ausgangs-Plugs `OUT1` und `OUT2` weitergeleitet. Der Baustein führt keinerlei Logik, Filterung oder Verzögerung durch – er fungiert als reiner Splitter auf Adapterebene, unabhängig davon, welcher der drei Ereignistypen eintrifft.

Technische Besonderheiten

- **Generischer Typ**: Implementiert über die generische Basisklasse `CGenUnidirectSplitBase` (1 Socket, N Plugs) – dieselbe C++-Basis, die auch `GEN_AE_SPLIT` und `GEN_ASR_SPLIT` zugrunde liegt.
- **Drei unabhängige Ereigniskanäle**: `SET`, `RESET` und `TOGGLE` werden getrennt geprüft und jeweils auf beide Ausgänge verteilt.

- **Keine Zustände / Algorithmen**: Da ASRT keinen Datenwert trägt, gibt es keine Änderungserkennung und kein ECC.

Zustandsübersicht

Der Baustein besitzt **keinen Zustandsautomaten**. Das Verhalten ist rein kombinatorisch: Jedes eingehende SET-, RESET- oder TOGGLE-Ereignis am Socket IN wird unmittelbar an beiden Ausgängen dupliziert.

Anwendungsszenarien

- **Ereignisverteilung**: Ein Set/Reset/Toggle-Signal (z. B. von AX_T_FF_SR gesteuert) soll von zwei unabhängigen Subsystemen verarbeitet werden.
- **Parallelschaltung**: Aufteilung eines ASRT-Signals zur gleichzeitigen Ansteuerung zweier Aktoren.

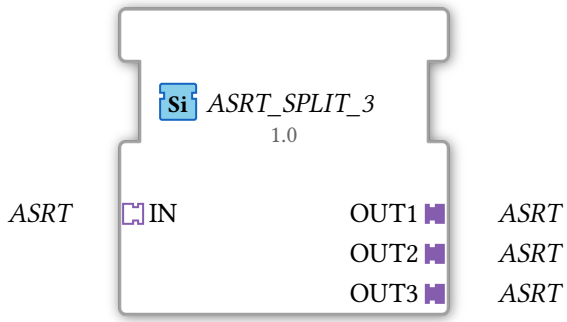
Vergleich mit ähnlichen Bausteinen

- ASRT_MERGE_2: die Umkehrrichtung – führt zwei eingehende Set/Reset/Toggle-Signale auf einen gemeinsamen Ausgang zusammen, statt ein Signal zu verteilen.
- ASR_SPLIT_2: dieselbe Verteilungslogik für den Set/Reset-Ereignisadapter ASR (ohne TOGGLE).
- AE_SPLIT_2: dieselbe Verteilungslogik für den reinen Ereignisadapter AE (1 Ereignis statt SET/RESET/TOGGLE).

Fazit

Der **ASRT_SPLIT_2** ist ein minimaler generischer Baustein zur Verteilung eines Set/Reset/Toggle-Ereignisses auf zwei ASRT-Adapterausgänge und schließt die zuletzt verbliebene Lücke in der GEN_*_SPLIT-Familie für alle drei unidirektionalen Ereignisadapter (AE, ASR, ASRT).

ASRT_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASRT_SPLIT_3** verteilt einen eingehenden unidirektionalen ASRT-Adapter (Set/Reset/Toggle-Ereignisadapter, keine Nutzdaten) auf 3 identische Ausgangsadapter. Er ist die 3-Ausgangs-Variante von ASRT_SPLIT_2 und wie dieser als generischer FB implementiert (GenericClassName: 'GEN_ASRT_SPLIT') — dieselbe C++-Basis (CGenUnidirectSplitBase), nur mit 3 statt 2 Plugs.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignisübernahme erfolgt ausschließlich über den Adapter-Socket.

Ereignis-Ausgänge

Keine. Die Ereignisweitergabe erfolgt ausschließlich über die Adapter-Plugs.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket (Eingang)	`IN`	`adapter::types::unidirectional::ASRT`	Empfängt `SET` bzw. `RESET` bzw. `TOGGLE`, das auf alle 3 Ausgänge verteilt wird.
Plug (Ausgang 1)	`OUT1`	`adapter::types::unidirectional::ASRT`	1. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 2)	`OUT2`	`adapter::types::unidirectional::ASRT`	2. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 3)	`OUT3`	`adapter::types::unidirectional::ASRT`	3. Ausgang für das duplizierte Ereignis.

Funktionsweise

Sobald am Adapter-Socket `IN` `SET` bzw. `RESET` bzw. `TOGGLE` eintrifft, wird es **sofort und parallel** als gleichartiges Ereignis an alle 3 Ausgangs-Plugs (`OUT1`, `OUT2`, `OUT3`) weitergeleitet. Der Baustein führt keinerlei Logik, Filterung oder Verzögerung durch – er fungiert als reiner Splitter auf Adapterebene.

Technische Besonderheiten

- **Generischer Typ**: Implementiert über die generische Basisklasse `CGenUnidirectSplitBase` (1 Socket, N Plugs); die Anzahl der Ausgangs-Plugs wird über den generischen Suffix (`_3`) zur Übersetzungszeit festgelegt.
- **Keine Zustände / Algorithmen**: Da `ASRT` keinen Datenwert trägt, gibt es keine Änderungserkennung und kein ECC.
- **Beliebige Ausgangsanzahl**: Dieselbe Implementierung deckt ASRT_SPLIT_2 bis ASRT_SPLIT_9 ab; für andere Ausgangsanzahlen siehe `ASRT_SPLIT_2`, `ASRT_SPLIT_4`, `ASRT_SPLIT_5`, `ASRT_SPLIT_6`, `ASRT_SPLIT_7`, `ASRT_SPLIT_8`, `ASRT_SPLIT_9`.

Zustandsübersicht

Der Baustein besitzt **keinen Zustandsautomaten**. Das Verhalten ist rein kombinatorisch: Jedes eingehende Ereignis am Socket `IN` wird unmittelbar an allen 3 Ausgängen dupliziert.

Anwendungsszenarien

- **Ereignisverteilung**: Ein Signal soll von 3 unabhängigen Subsystemen verarbeitet werden.
- **Parallelschaltung**: Aufteilung eines ASRT-Signals zur gleichzeitigen Ansteuerung mehrerer Aktoren.

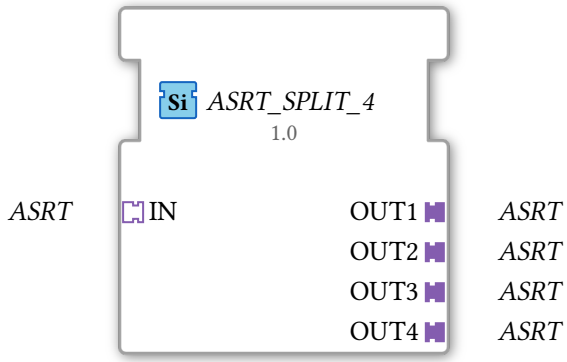
Vergleich mit ähnlichen Bausteinen

- **ASRT_MERGE_2**: die Umkehrrichtung für 2 Eingänge (bei ASRT zusätzlich ASRT_MERGE_3 bis ASRT_MERGE_7 verfügbar).
- **`ASRT_SPLIT_2`, `ASRT_SPLIT_4`, `ASRT_SPLIT_5`, `ASRT_SPLIT_6`, `ASRT_SPLIT_7`, `ASRT_SPLIT_8`, `ASRT_SPLIT_9`**: dieselbe generische Implementierung mit anderer Ausgangsanzahl.

Fazit

`ASRT_SPLIT_3` liefert eine generisch implementierte Verteilung eines `ASRT`-Ereignisses auf 3 Adapterausgänge und ergänzt die `GEN_ASRT_SPLIT`-Familie um die 3-Ausgangs-Variante.

ASRT_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASRT_SPLIT_4** verteilt einen eingehenden unidirektionalen ASRT-Adapter (Set/Reset/Toggle-Ereignisadapter, keine Nutzdaten) auf 4 identische Ausgangsadapter. Er ist die 4-Ausgangs-Variante von ASRT_SPLIT_2 und wie dieser als generischer FB implementiert (`GenericClassName: 'GEN_ASRT_SPLIT'`) — dieselbe C++-Basis (`CGenUnidirectSplitBase`), nur mit 4 statt 2 Plugs.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignisübernahme erfolgt ausschließlich über den Adapter-Socket.

Ereignis-Ausgänge

Keine. Die Ereignisweitergabe erfolgt ausschließlich über die Adapter-Plugs.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket (Eingang)	`IN`	`adapter::types::unidirectional::ASRT`	Empfängt `SET` bzw. `RESET` bzw. `TOGGLE`, das auf alle 4 Ausgänge verteilt wird.
Plug (Ausgang 1)	`OUT1`	`adapter::types::unidirectional::ASRT`	1. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 2)	`OUT2`	`adapter::types::unidirectional::ASRT`	2. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 3)	`OUT3`	`adapter::types::unidirectional::ASRT`	3. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 4)	`OUT4`	`adapter::types::unidirectional::ASRT`	4. Ausgang für das duplizierte Ereignis.

Funktionsweise

Sobald am Adapter-Socket `IN` `SET` bzw. `RESET` bzw. `TOGGLE` eintrifft, wird es **sofort und parallel** als gleichartiges Ereignis an alle 4 Ausgangs-Plugs (`OUT1`, `OUT2`, `OUT3`, `OUT4`) weitergeleitet. Der Baustein führt keinerlei Logik, Filterung oder Verzögerung durch – er fungiert als reiner Splitter auf Adapterebene.

Technische Besonderheiten

- **Generischer Typ**: Implementiert über die generische Basisklasse `CGenUnidirectSplitBase` (1 Socket, N Plugs); die Anzahl der Ausgangs-Plugs wird über den generischen Suffix (`_4`) zur Übersetzungszeit festgelegt.
- **Keine Zustände / Algorithmen**: Da `ASRT` keinen Datenwert trägt, gibt es keine Änderungserkennung und kein ECC.
- **Beliebige Ausgangsanzahl**: Dieselbe Implementierung deckt ASRT_SPLIT_2 bis ASRT_SPLIT_9 ab; für andere Ausgangsanzahlen siehe `ASRT_SPLIT_2`, `ASRT_SPLIT_3`, `ASRT_SPLIT_5`, `ASRT_SPLIT_6`, `ASRT_SPLIT_7`, `ASRT_SPLIT_8`, `ASRT_SPLIT_9`.

Zustandsübersicht

Der Baustein besitzt **keinen Zustandsautomaten**. Das Verhalten ist rein kombinatorisch: Jedes eingehende Ereignis am Socket `IN` wird unmittelbar an allen 4 Ausgängen dupliziert.

Anwendungsszenarien

- **Ereignisverteilung**: Ein Signal soll von 4 unabhängigen Subsystemen verarbeitet werden.
- **Parallelschaltung**: Aufteilung eines ASRT-Signals zur gleichzeitigen Ansteuerung mehrerer Aktoren.

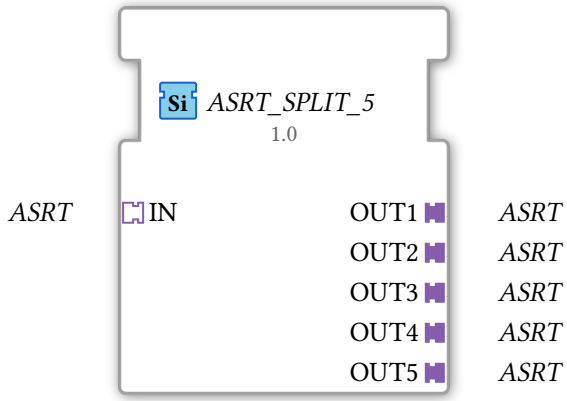
Vergleich mit ähnlichen Bausteinen

- ASRT_MERGE_2: die Umkehrrichtung für 2 Eingänge (bei ASRT zusätzlich ASRT_MERGE_3 bis ASRT_MERGE_7 verfügbar).
- **`ASRT_SPLIT_2`, `ASRT_SPLIT_3`, `ASRT_SPLIT_5`, `ASRT_SPLIT_6`, `ASRT_SPLIT_7`, `ASRT_SPLIT_8`, `ASRT_SPLIT_9`**: dieselbe generische Implementierung mit anderer Ausgangsanzahl.

Fazit

`ASRT_SPLIT_4` liefert eine generisch implementierte Verteilung eines `ASRT`-Ereignisses auf 4 Adapterausgänge und ergänzt die `GEN_ASRT_SPLIT`-Familie um die 4-Ausgangs-Variante.

ASRT_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASRT_SPLIT_5** verteilt einen eingehenden unidirektionalen ASRT-Adapter (Set/Reset/Toggle-Ereignisadapter, keine Nutzdaten) auf 5 identische Ausgangsadapter. Er ist die 5-Ausgangs-Variante von ASRT_SPLIT_2 und wie dieser als generischer FB implementiert (`GenericClassName: 'GEN_ASRT_SPLIT'`) — dieselbe C++-Basis (`CGenUnidirectSplitBase`), nur mit 5 statt 2 Plugs.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignisübernahme erfolgt ausschließlich über den Adapter-Socket.

Ereignis-Ausgänge

Keine. Die Ereignisweitergabe erfolgt ausschließlich über die Adapter-Plugs.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket (Eingang)	`IN`	`adapter::types::unidirectional::ASRT`	Empfängt `SET` bzw. `RESET` bzw. `TOGGLE`, das auf alle 5 Ausgänge verteilt wird.
Plug (Ausgang 1)	`OUT1`	`adapter::types::unidirectional::ASRT`	1. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 2)	`OUT2`	`adapter::types::unidirectional::ASRT`	2. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 3)	`OUT3`	`adapter::types::unidirectional::ASRT`	3. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 4)	`OUT4`	`adapter::types::unidirectional::ASRT`	4. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 5)	`OUT5`	`adapter::types::unidirectional::ASRT`	5. Ausgang für das duplizierte Ereignis.

Funktionsweise

Sobald am Adapter-Socket `IN` `SET` bzw. `RESET` bzw. `TOGGLE` eintrifft, wird es **sofort und parallel** als gleichartiges Ereignis an alle 5 Ausgangs-Plugs (`OUT1`, `OUT2`, `OUT3`, `OUT4`, `OUT5`) weitergeleitet. Der Baustein führt keinerlei Logik, Filterung oder Verzögerung durch – er fungiert als reiner Splitter auf Adapterebene.

Technische Besonderheiten

- **Generischer Typ**: Implementiert über die generische Basisklasse `CGenUnidirectSplitBase` (1 Socket, N Plugs); die Anzahl der Ausgangs-Plugs wird über den generischen Suffix (`_5`) zur Übersetzungszeit festgelegt.
- **Keine Zustände / Algorithmen**: Da `ASRT` keinen Datenwert trägt, gibt es keine Änderungserkennung und kein ECC.
- **Beliebige Ausgangsanzahl**: Dieselbe Implementierung deckt ASRT_SPLIT_2 bis ASRT_SPLIT_9 ab; für andere Ausgangsanzahlen siehe `ASRT_SPLIT_2`, `ASRT_SPLIT_3`, `ASRT_SPLIT_4`, `ASRT_SPLIT_6`, `ASRT_SPLIT_7`, `ASRT_SPLIT_8`, `ASRT_SPLIT_9`.

Zustandsübersicht

Der Baustein besitzt **keinen Zustandsautomaten**. Das Verhalten ist rein kombinatorisch: Jedes eingehende Ereignis am Socket `IN` wird unmittelbar an allen 5 Ausgängen dupliziert.

Anwendungsszenarien

- **Ereignisverteilung**: Ein Signal soll von 5 unabhängigen Subsystemen verarbeitet werden.
- **Parallelschaltung**: Aufteilung eines ASRT-Signals zur gleichzeitigen Ansteuerung mehrerer Aktoren.

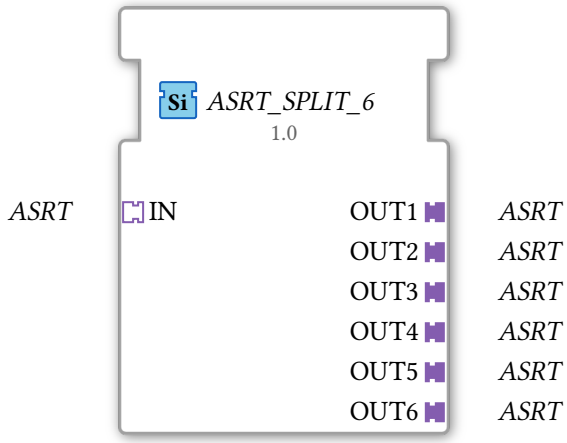
Vergleich mit ähnlichen Bausteinen

- **ASRT_MERGE_2**: die Umkehrrichtung für 2 Eingänge (bei ASRT zusätzlich ASRT_MERGE_3 bis ASRT_MERGE_7 verfügbar).
- **`ASRT_SPLIT_2`, `ASRT_SPLIT_3`, `ASRT_SPLIT_4`, `ASRT_SPLIT_6`, `ASRT_SPLIT_7`, `ASRT_SPLIT_8`, `ASRT_SPLIT_9`**: dieselbe generische Implementierung mit anderer Ausgangsanzahl.

Fazit

`ASRT_SPLIT_5` liefert eine generisch implementierte Verteilung eines `ASRT`-Ereignisses auf 5 Adapterausgänge und ergänzt die `GEN_ASRT_SPLIT`-Familie um die 5-Ausgangs-Variante.

ASRT_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASRT_SPLIT_6** verteilt einen eingehenden unidirektionalen ASRT-Adapter (Set/Reset/Toggle-Ereignisadapter, keine Nutzdaten) auf 6 identische Ausgangsadapter. Er ist die 6-Ausgangs-Variante von ASRT_SPLIT_2 und wie dieser als generischer FB implementiert (`GenericClassName: 'GEN_ASRT_SPLIT'`) — dieselbe C++-Basis (`CGenUnidirectSplitBase`), nur mit 6 statt 2 Plugs.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignisübernahme erfolgt ausschließlich über den Adapter-Socket.

Ereignis-Ausgänge

Keine. Die Ereignisweitergabe erfolgt ausschließlich über die Adapter-Plugs.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket (Eingang)	`IN`	`adapter::types::unidirectional::ASRT`	Empfängt `SET` bzw. `RESET` bzw. `TOGGLE`, das auf alle 6 Ausgänge verteilt wird.
Plug (Ausgang 1)	`OUT1`	`adapter::types::unidirectional::ASRT`	1. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 2)	`OUT2`	`adapter::types::unidirectional::ASRT`	2. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 3)	`OUT3`	`adapter::types::unidirectional::ASRT`	3. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 4)	`OUT4`	`adapter::types::unidirectional::ASRT`	4. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 5)	`OUT5`	`adapter::types::unidirectional::ASRT`	5. Ausgang für das duplizierte Ereignis.

Plug (Ausgang 6)	OUT6	adapter::types::unidirectional::ASRT	6. Ausgang für das duplizierte Ereignis.

Funktionsweise

Sobald am Adapter-Socket `IN` `SET` bzw. `RESET` bzw. `TOGGLE` eintrifft, wird es **sofort und parallel** als gleichartiges Ereignis an alle 6 Ausgangs-Plugs (`OUT1`, `OUT2`, `OUT3`, `OUT4`, `OUT5`, `OUT6`) weitergeleitet. Der Baustein führt keinerlei Logik, Filterung oder Verzögerung durch – er fungiert als reiner Splitter auf Adapterebene.

Technische Besonderheiten

- **Generischer Typ**: Implementiert über die generische Basisklasse `CGenUnidirectSplitBase` (1 Socket, N Plugs); die Anzahl der Ausgangs-Plugs wird über den generischen Suffix (`_6`) zur Übersetzungszeit festgelegt.
- **Keine Zustände / Algorithmen**: Da `ASRT` keinen Datenwert trägt, gibt es keine Änderungserkennung und kein ECC.
- **Beliebige Ausgangsanzahl**: Dieselbe Implementierung deckt ASRT_SPLIT_2 bis ASRT_SPLIT_9 ab; für andere Ausgangsanzahlen siehe `ASRT_SPLIT_2`, `ASRT_SPLIT_3`, `ASRT_SPLIT_4`, `ASRT_SPLIT_5`, `ASRT_SPLIT_7`, `ASRT_SPLIT_8`, `ASRT_SPLIT_9`.

Zustandsübersicht

Der Baustein besitzt **keinen Zustandsautomaten**. Das Verhalten ist rein kombinatorisch: Jedes eingehende Ereignis am Socket `IN` wird unmittelbar an allen 6 Ausgängen dupliziert.

Anwendungsszenarien

- **Ereignisverteilung**: Ein Signal soll von 6 unabhängigen Subsystemen verarbeitet werden.
- **Parallelschaltung**: Aufteilung eines ASRT-Signals zur gleichzeitigen Ansteuerung mehrerer Aktoren.

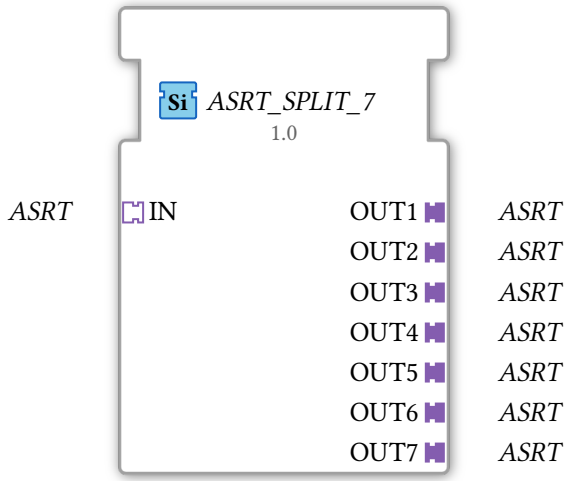
Vergleich mit ähnlichen Bausteinen

- ASRT_MERGE_2: die Umkehrrichtung für 2 Eingänge (bei ASRT zusätzlich ASRT_MERGE_3 bis ASRT_MERGE_7 verfügbar).
- **`ASRT_SPLIT_2`, `ASRT_SPLIT_3`, `ASRT_SPLIT_4`, `ASRT_SPLIT_5`, `ASRT_SPLIT_7`, `ASRT_SPLIT_8`, `ASRT_SPLIT_9`**: dieselbe generische Implementierung mit anderer Ausgangsanzahl.

Fazit

`ASRT_SPLIT_6` liefert eine generisch implementierte Verteilung eines `ASRT`-Ereignisses auf 6 Adapterausgänge und ergänzt die `GEN_ASRT_SPLIT`-Familie um die 6-Ausgangs-Variante.

ASRT_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASRT_SPLIT_7** verteilt einen eingehenden unidirektionalen ASRT-Adapter (Set/Reset/Toggle-Ereignisadapter, keine Nutzdaten) auf 7 identische Ausgangsadapter. Er ist die 7-Ausgangs-Variante von ASRT_SPLIT_2 und wie dieser als generischer FB implementiert (`GenericClassName: 'GEN_ASRT_SPLIT'`) — dieselbe C++-Basis (`CGenUnidirectSplitBase`), nur mit 7 statt 2 Plugs.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignisübernahme erfolgt ausschließlich über den Adapter-Socket.

Ereignis-Ausgänge

Keine. Die Ereignisweitergabe erfolgt ausschließlich über die Adapter-Plugs.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket (Eingang)	`IN`	`adapter::types::unidirectional::ASRT`	Empfängt `SET` bzw. `RESET` bzw. `TOGGLE`, das auf alle 7 Ausgänge verteilt wird.
Plug (Ausgang 1)	`OUT1`	`adapter::types::unidirectional::ASRT`	1. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 2)	`OUT2`	`adapter::types::unidirectional::ASRT`	2. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 3)	`OUT3`	`adapter::types::unidirectional::ASRT`	3. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 4)	`OUT4`	`adapter::types::unidirectional::ASRT`	4. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 5)	`OUT5`	`adapter::types::unidirectional::ASRT`	5. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 6)	`OUT6`	`adapter::types::unidirectional::ASRT`	6. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 7)	`OUT7`	`adapter::types::unidirectional::ASRT`	7. Ausgang für das duplizierte Ereignis.

Funktionsweise

Sobald am Adapter-Socket `IN` `SET` bzw. `RESET` bzw. `TOGGLE` eintrifft, wird es **sofort und parallel** als gleichartiges Ereignis an alle 7 Ausgangs-Plugs (`OUT1`, `OUT2`, `OUT3`, `OUT4`, `OUT5`, `OUT6`, `OUT7`) weitergeleitet. Der Baustein führt keinerlei Logik, Filterung oder Verzögerung durch – er fungiert als reiner Splitter auf Adapterebene.

Technische Besonderheiten

- **Generischer Typ**: Implementiert über die generische Basisklasse `CGenUnidirectSplitBase` (1 Socket, N Plugs); die Anzahl der Ausgangs-Plugs wird über den generischen Suffix (`_7`) zur Übersetzungszeit festgelegt.
- **Keine Zustände / Algorithmen**: Da `ASRT` keinen Datenwert trägt, gibt es keine Änderungserkennung und kein ECC.

- **Beliebige Ausgangsanzahl**: Dieselbe Implementierung deckt ASRT_SPLIT_2 bis ASRT_SPLIT_9 ab; für andere Ausgangsanzahlen siehe `ASRT_SPLIT_2`, `ASRT_SPLIT_3`, `ASRT_SPLIT_4`, `ASRT_SPLIT_5`, `ASRT_SPLIT_6`, `ASRT_SPLIT_8`, `ASRT_SPLIT_9`.

Zustandsübersicht

Der Baustein besitzt **keinen Zustandsautomaten**. Das Verhalten ist rein kombinatorisch: Jedes eingehende Ereignis am Socket `IN` wird unmittelbar an allen 7 Ausgängen dupliziert.

Anwendungsszenarien

- **Ereignisverteilung**: Ein Signal soll von 7 unabhängigen Subsystemen verarbeitet werden.
- **Parallelschaltung**: Aufteilung eines ASRT-Signals zur gleichzeitigen Ansteuerung mehrerer Aktoren.

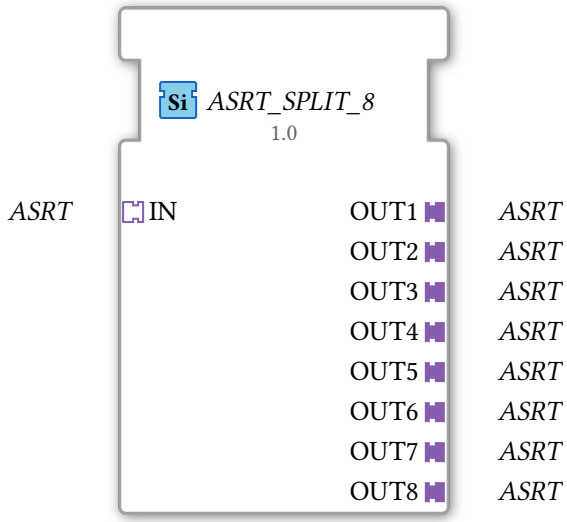
Vergleich mit ähnlichen Bausteinen

- ASRT_MERGE_2: die Umkehrrichtung für 2 Eingänge (bei ASRT zusätzlich ASRT_MERGE_3 bis ASRT_MERGE_7 verfügbar).
- **`ASRT_SPLIT_2`, `ASRT_SPLIT_3`, `ASRT_SPLIT_4`, `ASRT_SPLIT_5`, `ASRT_SPLIT_6`, `ASRT_SPLIT_8`, `ASRT_SPLIT_9`**: dieselbe generische Implementierung mit anderer Ausgangsanzahl.

Fazit

`ASRT_SPLIT_7` liefert eine generisch implementierte Verteilung eines `ASRT`-Ereignisses auf 7 Adapterausgänge und ergänzt die `GEN_ASRT_SPLIT`-Familie um die 7-Ausgangs-Variante.

ASRT_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASRT_SPLIT_8** verteilt einen eingehenden unidirektionalen ASRT-Adapter (Set/Reset/Toggle-Ereignisadapter, keine Nutzdaten) auf 8 identische Ausgangsadapter. Er ist die 8-Ausgangs-Variante von ASRT_SPLIT_2 und wie dieser als generischer FB implementiert (`GenericClassName: 'GEN_ASRT_SPLIT'`) — dieselbe C++-Basis (`CGenUnidirectSplitBase`), nur mit 8 statt 2 Plugs.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignisübernahme erfolgt ausschließlich über den Adapter-Socket.

Ereignis-Ausgänge

Keine. Die Ereignisweitergabe erfolgt ausschließlich über die Adapter-Plugs.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket (Eingang)	IN	adapter::types::unidirectional::ASRT	Empfängt SET bzw. RESET bzw. TOGGLE, das auf alle 8 Ausgänge verteilt wird.
Plug (Ausgang 1)	OUT1	adapter::types::unidirectional::ASRT	1. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 2)	OUT2	adapter::types::unidirectional::ASRT	2. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 3)	OUT3	adapter::types::unidirectional::ASRT	3. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 4)	OUT4	adapter::types::unidirectional::ASRT	4. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 5)	OUT5	adapter::types::unidirectional::ASRT	5. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 6)	OUT6	adapter::types::unidirectional::ASRT	6. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 7)	OUT7	adapter::types::unidirectional::ASRT	7. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 8)	OUT8	adapter::types::unidirectional::ASRT	8. Ausgang für das duplizierte Ereignis.

Funktionsweise

Sobald am Adapter-Socket IN SET bzw. RESET bzw. TOGGLE eintrifft, wird es **sofort und parallel** als gleichartiges Ereignis an alle 8 Ausgangs-Plugs (OUT1, OUT2, OUT3, OUT4, OUT5, OUT6, OUT7, OUT8) weitergeleitet. Der Baustein führt keinerlei Logik, Filterung oder Verzögerung durch – er fungiert als reiner Splitter auf Adapterebene.

Technische Besonderheiten

- **Generischer Typ**: Implementiert über die generische Basisklasse CGenUnidirectSplitBase (1 Socket, N Plugs); die Anzahl der Ausgangs-Plugs wird über den generischen Suffix (_8) zur Übersetzungszeit festgelegt.
- **Keine Zustände / Algorithmen**: Da ASRT keinen Datenwert trägt, gibt es keine Änderungserkennung und kein ECC.
- **Beliebige Ausgangsanzahl**: Dieselbe Implementierung deckt ASRT_SPLIT_2 bis ASRT_SPLIT_9 ab; für andere Ausgangsanzahlen siehe ASRT_SPLIT_2, ASRT_SPLIT_3, ASRT_SPLIT_4, ASRT_SPLIT_5, ASRT_SPLIT_6, ASRT_SPLIT_7, ASRT_SPLIT_9.

Zustandsübersicht

Der Baustein besitzt **keinen Zustandsautomaten**. Das Verhalten ist rein kombinatorisch: Jedes eingehende Ereignis am Socket IN wird unmittelbar an allen 8 Ausgängen dupliziert.

Anwendungsszenarien

- **Ereignisverteilung**: Ein Signal soll von 8 unabhängigen Subsystemen verarbeitet werden.
- **Parallelschaltung**: Aufteilung eines ASRT-Signals zur gleichzeitigen Ansteuerung mehrerer Aktoren.

Vergleich mit ähnlichen Bausteinen

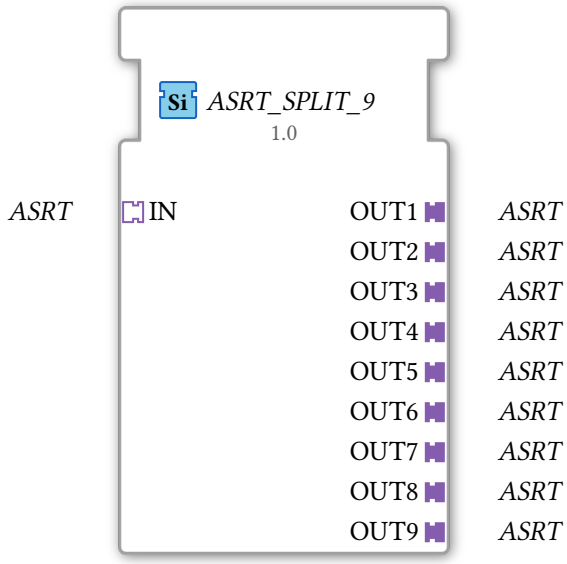
- ASRT_MERGE_2: die Umkehrrichtung für 2 Eingänge (bei ASRT zusätzlich ASRT_MERGE_3 bis ASRT_MERGE_7 verfügbar).

- **ASRT_SPLIT_2**, **ASRT_SPLIT_3**, **ASRT_SPLIT_4**, **ASRT_SPLIT_5**, **ASRT_SPLIT_6**, **ASRT_SPLIT_7**, **ASRT_SPLIT_9**: dieselbe generische Implementierung mit anderer Ausgangsanzahl.

Fazit

`ASRT_SPLIT_8` liefert eine generisch implementierte Verteilung eines `ASRT`-Ereignisses auf 8 Adapterausgänge und ergänzt die `GEN_ASRT_SPLIT`-Familie um die 8-Ausgangs-Variante.

ASRT_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASRT_SPLIT_9** verteilt einen eingehenden unidirektionalen ASRT-Adapter (Set/Reset/Toggle-Ereignisadapter, keine Nutzdaten) auf 9 identische Ausgangsadapter. Er ist die 9-Ausgangs-Variante von ASRT_SPLIT_2 und wie dieser als generischer FB implementiert (`GenericClassName: 'GEN_ASRT_SPLIT'`) — dieselbe C++-Basis (`CGenUnidirectSplitBase`), nur mit 9 statt 2 Plugs.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignisübernahme erfolgt ausschließlich über den Adapter-Socket.

Ereignis-Ausgänge

Keine. Die Ereignisweitergabe erfolgt ausschließlich über die Adapter-Plugs.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket (Eingang)	`IN`	`adapter::types::unidirectional::ASRT`	Empfängt `SET` bzw. `RESET` bzw. `TOGGLE`, das auf alle 9 Ausgänge verteilt wird.
Plug (Ausgang 1)	`OUT1`	`adapter::types::unidirectional::ASRT`	1. Ausgang für das duplizierte Ereignis.

Plug (Ausgang 2)	OUT2	adapter::types::unidirectional::ASRT	2. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 3)	OUT3	adapter::types::unidirectional::ASRT	3. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 4)	OUT4	adapter::types::unidirectional::ASRT	4. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 5)	OUT5	adapter::types::unidirectional::ASRT	5. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 6)	OUT6	adapter::types::unidirectional::ASRT	6. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 7)	OUT7	adapter::types::unidirectional::ASRT	7. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 8)	OUT8	adapter::types::unidirectional::ASRT	8. Ausgang für das duplizierte Ereignis.
Plug (Ausgang 9)	OUT9	adapter::types::unidirectional::ASRT	9. Ausgang für das duplizierte Ereignis.

Funktionsweise

Sobald am Adapter-Socket `IN` `SET` bzw. `RESET` bzw. `TOGGLE` eintrifft, wird es **sofort und parallel** als gleichartiges Ereignis an alle 9 Ausgangs-Plugs (`OUT1`, `OUT2`, `OUT3`, `OUT4`, `OUT5`, `OUT6`, `OUT7`, `OUT8`, `OUT9`) weitergeleitet. Der Baustein führt keinerlei Logik, Filterung oder Verzögerung durch – er fungiert als reiner Splitter auf Adapterebene.

Technische Besonderheiten

- **Generischer Typ**: Implementiert über die generische Basisklasse `CGenUnidirectSplitBase` (1 Socket, N Plugs); die Anzahl der Ausgangs-Plugs wird über den generischen Suffix (`_9`) zur Übersetzungszeit festgelegt.
- **Keine Zustände / Algorithmen**: Da `ASRT` keinen Datenwert trägt, gibt es keine Änderungserkennung und kein ECC.
- **Beliebige Ausgangsanzahl**: Dieselbe Implementierung deckt ASRT_SPLIT_2 bis ASRT_SPLIT_9 ab; für andere Ausgangsanzahlen siehe `ASRT_SPLIT_2`, `ASRT_SPLIT_3`, `ASRT_SPLIT_4`, `ASRT_SPLIT_5`, `ASRT_SPLIT_6`, `ASRT_SPLIT_7`, `ASRT_SPLIT_8`.

Zustandsübersicht

Der Baustein besitzt **keinen Zustandsautomaten**. Das Verhalten ist rein kombinatorisch: Jedes eingehende Ereignis am Socket `IN` wird unmittelbar an allen 9 Ausgängen dupliziert.

Anwendungsszenarien

- **Ereignisverteilung**: Ein Signal soll von 9 unabhängigen Subsystemen verarbeitet werden.
- **Parallelschaltung**: Aufteilung eines ASRT-Signals zur gleichzeitigen Ansteuerung mehrerer Aktoren.

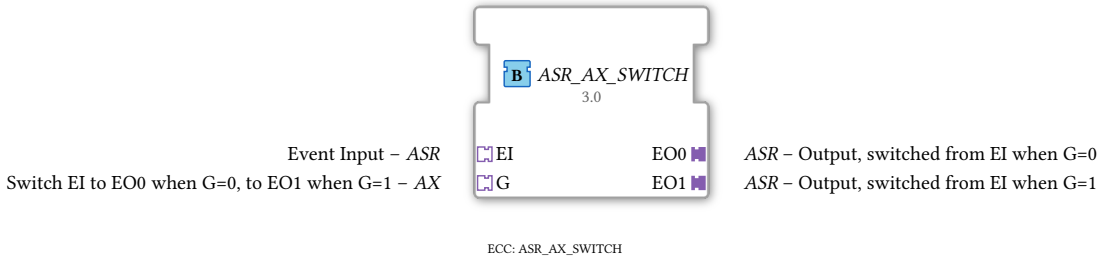
Vergleich mit ähnlichen Bausteinen

- **ASRT_MERGE_2**: die Umkehrrichtung für 2 Eingänge (bei ASRT zusätzlich ASRT_MERGE_3 bis ASRT_MERGE_7 verfügbar).
- **`ASRT_SPLIT_2`, `ASRT_SPLIT_3`, `ASRT_SPLIT_4`, `ASRT_SPLIT_5`, `ASRT_SPLIT_6`, `ASRT_SPLIT_7`, `ASRT_SPLIT_8`**: dieselbe generische Implementierung mit anderer Ausgangsanzahl.

Fazit

`ASRT_SPLIT_9` liefert eine generisch implementierte Verteilung eines `ASRT`-Ereignisses auf 9 Adapterausgänge und ergänzt die `GEN_ASRT_SPLIT`-Familie um die 9-Ausgangs-Variante.

ASR_AX_SWITCH



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ASR_AX_SWITCH** dient als Demultiplexer für asynchrone Set/Reset-Ereignisse (ASR). Er leitet ein eingehendes SET- oder RESET-Ereignis basierend auf dem Zustand eines binären Steuereingangs **G** wahlweise an einen von zwei Ausgängen (**EO0** oder **EO1**) weiter. Der Baustein ermöglicht eine bedarfsgerechte Verteilung von Steuersignalen in Automatisierungsanwendungen.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine klassischen Ereignis-Eingänge über *EventInputs* – die Ereignisse werden über den **Socket-Adapter `EI`** vom Typ `ASR` empfangen. Dieser Adapter stellt die beiden Ereignisse `SET` und `RESET` bereit.

Ereignis-Ausgänge

Die Ereignis-Ausgänge werden über die **Plug-Adapter `EO0`** und **`EO1`** (beide Typ `ASR`) bereitgestellt. Jeder dieser Adapter verfügt über die Ausgänge `SET` und `RESET`:

Adapter	Ereignis	Beschreibung
`EO0.SET`	SET	Wird ausgelöst, wenn ein `EI.SET`-Ereignis eintrifft und `G.D1 == 0`
`EO0.RESET`	RESET	Wird ausgelöst, wenn ein `EI.RESET`-Ereignis eintrifft und `G.D1 == 0`
`EO1.SET`	SET	Wird ausgelöst, wenn ein `EI.SET`-Ereignis eintrifft und `G.D1 == 1`

`EO1.RESET`	RESET	Wird ausgelöst, wenn ein `EI.RESET`-Ereignis eintrifft und `G.D1 == 1`

Daten-Eingänge

Der FB verfügt über keine separaten Daten-Eingänge. Die Steuergröße **G** wird über den **Socket-Adapter** `G` vom Typ `AX` bezogen – das logische Signal liegt am Datenausgang `D1` des Adapters an.

Daten-Ausgänge

Es sind keine expliziten Daten-Ausgänge vorhanden. Steuerinformationen werden ausschließlich über die Ereignisausgänge der ASR-Adapter übertragen.

Adapter

Name	Richtung	Typ	Beschreibung
`EI`	Socket (Eingang)	`adapter::types::unidirectional::ASR`	Empfängt die zu verteilenden SET-/RESET-Ereignisse
`G`	Socket (Eingang)	`adapter::types::unidirectional::AX`	Liefert das binäre Umschaltsignal über `D1` (0 ⟶ Kanal 0, 1 ⟶ Kanal 1)
`EO0`	Plug (Ausgang)	`adapter::types::unidirectional::ASR`	Ausgang für Kanal 0 (bei `G.D1 = 0`)
`EO1`	Plug (Ausgang)	`adapter::types::unidirectional::ASR`	Ausgang für Kanal 1 (bei `G.D1 = 1`)

Funktionsweise

Der FB arbeitet ereignisgesteuert. Nach dem Start befindet er sich im Zustand **START** und wartet auf eingehende Ereignisse über den `EI`-Adapter:

- **`EI.SET`** und **`EI.RESET`** werden nur dann verarbeitet, wenn gleichzeitig über den `G`-Adapter ein gültiger binärer Wert an `G.D1` anliegt.
- Die Bedingung für die Transition lautet:
  - `EI.SET[NOT G.D1]` ⟶ **G0_SET** ⟶ Ausgabe `EO0.SET`
  - `EI.SET[G.D1]` ⟶ **G1_SET** ⟶ Ausgabe `EO1.SET`
  - `EI.RESET[NOT G.D1]` ⟶ **G0_RESET** ⟶ Ausgabe `EO0.RESET`
  - `EI.RESET[G.D1]` ⟶ **G1_RESET** ⟶ Ausgabe `EO1.RESET`
- Nach dem Auslösen des Ausgangsereignisses kehrt der FB sofort in den **START**-Zustand zurück (unbedingte Transition mit `1`).
- Zusätzlich existiert eine Transition von **START** nach **START** bei **`G.E1`** – dies erlaubt eine asynchrone Aktualisierung des Steuersignals ohne sofortige Ereignisweitergabe (dient z.B. der Synchronisation oder Initialisierung).

Die Verarbeitung erfolgt deterministisch ohne Zwischenspeicherung der Ereignisse.

Technische Besonderheiten

- **Adapterbasierte Kommunikation**: Der FB verwendet ausschließlich Adapter (Sockets/Plugs) für Ereignis- und Datenaustausch – dies ermöglicht eine flexible Kopplung in der 4diac-IDE.
- **Keine Zustandsspeicherung**: Die Auswahl des Zielkanals erfolgt rein kombinatorisch zum Zeitpunkt des eingehenden Ereignisses. Es wird kein interner Zustand gehalten.
- **Volle Synchronität**: Nach jedem Ausgabeereignis kehrt der FB wieder in den Grundzustand zurück – es können keine Ereignisse gestapelt werden.
- **Eingebettete Lizenz**: Der Code ist unter der Eclipse Public License 2.0 veröffentlicht.

Zustandsübersicht

Zustand	Beschreibung	Aktion
START	Warte auf eingehendes SET/RESET-Ereignis	–
G0_SET	`EI.SET` bei `G.D1=0` empfangen	`EO0.SET` ausgeben
G1_SET	`EI.SET` bei `G.D1=1` empfangen	`EO1.SET` ausgeben

G0_RESET	`EI.RESET` bei `G.D1=0` empfangen	`EO0.RESET` ausgeben
G1_RESET	`EI.RESET` bei `G.D1=1` empfangen	`EO1.RESET` ausgeben

Die Zustandsübergänge erfolgen nach dem ECC-Schema: Bedingung führt von START in den Aktionszustand, von dort ohne weitere Bedingung sofort zurück zu START.

Anwendungsszenarien

- **Kanalumschaltung**: In einer modularen Steuerung soll ein zentraler Set-/Reset-Befehl je nach Betriebsmodus an verschiedene Aktoren (z.B. Ventile, Motoren) weitergeleitet werden.
- **Sicherheitslogik**: Abhängig von einem Freigabesignal (`G`) werden Resets nur an den aktiven Kanal geleitet.
- **Redundanzsteuerung**: Das Signal `G` wählt zwischen Haupt- und Ersatzkanal aus.
- **Sequenzielle Prozesse**: Mehrere Bearbeitungsstationen werden durch Umlegen von `G` nacheinander angesteuert.

Vergleich mit ähnlichen Bausteinen

Eigenschaft	ASR_AX_SWITCH	Einfacher Event-Multiplexer (z.B. SELECT)
Ereignistyp	SET/RESET (ASR-Adapter)	Beliebige Einzelereignisse
Steuerung	Binär über AX-Adapter (mit `D1`)	Boolescher Dateneingang
Ausgangstyp	Zwei ASR-Adapter	Je nach Typ Einzelereignis oder mehrere
Zustandsbehaftet	Nein (keine Speicherung)	Meist zustandslos
Adapter-Schnittstelle	Ja	Oft direkt über Event-/Data-Ports

Der ASR_AX_SWITCH ist speziell für asynchrone Set/Reset-Signale optimiert und integriert sich nahtlos in die ASR-Adapter-Welt der 4diac-Bibliothek. Gegenüber generischen Multiplexern entfällt die Notwendigkeit, die Adapter-Ereignisse manuell zu extrahieren.

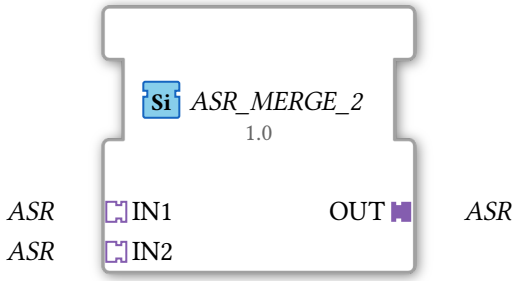
Fazit

Der **ASR_AX_SWITCH** ist ein kompakter, effizienter Demultiplexer für die gerichtete Weiterleitung von ASR-Steuerereignissen. Durch die adapterbasierte Schnittstelle und die klare Entscheidungslogik eignet er sich hervorragend für modulare Automatisierungsprojekte, bei denen ein einzelnes Steuersignal wahlweise auf verschiedene Pfade geschaltet werden muss. Die einfache Zustandsmaschine garantiert schnelle, deterministische Reaktionszeiten ohne Seiteneffekte.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ASR_MERGE_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASR_MERGE_2** führt zwei unidirektionale ASR-Adapter (Set/Reset-Ereignispaar, keine Nutzdaten) auf einen gemeinsamen Ausgangsadapter zusammen. Er ist die Umkehrung von ASR_SPLIT_2: Statt ein Set/Reset-Signal auf zwei Ausgänge zu verteilen, führt er zwei unabhängige

Set/Reset-Quellen auf einen gemeinsamen Ausgang. Der Baustein ist als generischer FB implementiert (`GenericClassName: 'GEN_ASR_MERGE'`).

Schnittstellenstruktur

Ereignis-Eingänge
Keine. Die Ereignisübernahme erfolgt ausschließlich über die Adapter-Sockets.

Ereignis-Ausgänge
Keine. Die Ereignisweitergabe erfolgt ausschließlich über den Adapter-Plug.

Daten-Eingänge
Keine.

Daten-Ausgänge
Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket (Eingang 1)	`IN1`	`adapter::types::unidirectional::ASR`	Erste SET/RESET-Quelle.
Socket (Eingang 2)	`IN2`	`adapter::types::unidirectional::ASR`	Zweite SET/RESET-Quelle.
Plug (Ausgang)	`OUT`	`adapter::types::unidirectional::ASR`	Zusammengeführtes SET/RESET-Signal.

Funktionsweise
Trifft ein `SET`-Ereignis an `IN1` **oder** `IN2` ein, wird ein `SET` an `OUT` weitergereicht; analog wird ein `RESET` an `IN1`/`IN2` an `OUT` als `RESET` durchgereicht. Beide Ereignistypen werden unabhängig voneinander behandelt – ein `SET` an `IN1` beeinflusst nicht, ob ein `RESET` an `IN2` durchgereicht wird. Da `ASR` keinen Datenwert transportiert, gibt es nichts zu arbitrieren außer dem jeweiligen Ereignistyp selbst.

Technische Besonderheiten
- **Generischer Typ**: Implementiert über die generische Basisklasse `CGenUnidirectMergeBase` (N Sockets, 1 Plug) – dieselbe C++-Basis, die auch `GEN_AE_MERGE` und `GEN_ASRT_MERGE` zugrunde liegt.
- **Zwei unabhängige Ereigniskanäle**: `SET` und `RESET` werden getrennt geprüft und weitergeleitet; ein Socket kann z. B. nur `SET`-Ereignisse liefern, während der andere nur `RESET` liefert, ohne dass sich beide gegenseitig stören.
- **Keine Zustände / Algorithmen**: Da `ASR` keinen Datenwert trägt, gibt es keine Änderungserkennung und kein ECC.

Zustandsübersicht
Der Baustein besitzt **keinen Zustandsautomaten**. Das Verhalten ist rein kombinatorisch: Jedes eingehende `SET`- oder `RESET`-Ereignis an `IN1` oder `IN2` wird unmittelbar als gleichartiges Ereignis an `OUT` weitergereicht.

Anwendungsszenarien
- **Zusammenführen redundanter Set/Reset-Quellen**: Zwei unabhängige Bedienstellen oder Sicherheitspfade sollen denselben ASR-Ausgang setzen bzw. zurücksetzen können.
- **Vereinfachung von Netzwerken**, die sonst zwei separate SET/RESET-Ereignisverbindungspaare zum selben Ziel benötigen würden.

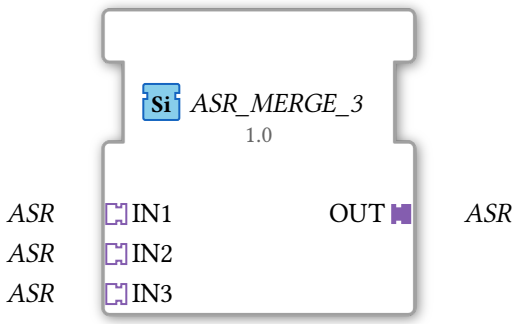
Vergleich mit ähnlichen Bausteinen
- ASR_SPLIT_2: die Umkehrrichtung – verteilt ein eingehendes Set/Reset-Signal auf zwei Ausgänge, statt zwei Eingänge zusammenzuführen.
- AE_MERGE_2: dieselbe Zusammenführungslogik für den reinen Ereignisadapter `AE` (1 Ereignis statt SET/RESET).

- ASRT_MERGE_2: dieselbe Zusammenführungslogik für den Set/Reset/Toggle-Ereignisadapter `ASRT` (zusätzlich `TOGGLE`).

Fazit

Der **ASR_MERGE_2** ist ein minimaler generischer Baustein zur Zusammenführung zweier Set/Reset-Ereignisquellen auf einen gemeinsamen ASR-Adapterausgang und schließt die zuvor als „hypothetisch“ vermerkte Lücke zu ASR_SPLIT_2.

ASR_MERGE_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASR_MERGE_3** führt 3 unidirektionale ASR-Adapter (Set/Reset-Ereignisadapter, keine Nutzdaten) auf einen gemeinsamen Ausgangsadapter zusammen. Er ist die 3-Eingangs-Variante von ASR_MERGE_2 und wie dieser als generischer FB implementiert (`GenericClassName: 'GEN_ASR_MERGE'`) — dieselbe C++-Basis (`CGenUnidirectMergeBase`), nur mit 3 statt 2 Sockets.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignisübernahme erfolgt ausschließlich über die Adapter-Sockets.

Ereignis-Ausgänge

Keine. Die Ereignisweitergabe erfolgt ausschließlich über den Adapter-Plug.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket (Eingang 1)	`IN1`	`adapter::types::unidirectional::ASR`	1. Quelle.
Socket (Eingang 2)	`IN2`	`adapter::types::unidirectional::ASR`	2. Quelle.
Socket (Eingang 3)	`IN3`	`adapter::types::unidirectional::ASR`	3. Quelle.
Plug (Ausgang)	`OUT`	`adapter::types::unidirectional::ASR`	Zusammengeführtes Signal.

Funktionsweise

Trifft `SET` bzw. `RESET` an einem der 3 Sockets (`IN1`, `IN2`, `IN3`) ein, wird es als gleichartiges Ereignis unverändert an `OUT` weitergereicht. Alle Ereignistypen werden unabhängig voneinander behandelt, und alle 3 Sockets werden gleichberechtigt behandelt — welcher Socket zuerst geprüft wird, folgt der Reihenfolge `IN1` … `IN3`, was nur bei mehreren innerhalb desselben Ausführungszyklus eintreffenden Ereignissen

relevant wird. Da ASR keinen Datenwert transportiert, gibt es nichts zu arbitrieren außer dem jeweiligen Ereignistyp selbst.

Technische Besonderheiten

- **Generischer Typ**: Implementiert über die generische Basisklasse `CGenUnidirectMergeBase` (N Sockets, 1 Plug); die Anzahl der Eingangs-Sockets wird über den generischen Suffix (_3) zur Übersetzungszeit festgelegt.
- **Keine Zustände / Algorithmen**: Da ASR keinen Datenwert trägt, gibt es keine Änderungserkennung und kein ECC.
- **Beliebige Eingangsanzahl**: Dieselbe Implementierung deckt ASR_MERGE_2 bis ASR_MERGE_7 ab; für andere Eingangsanzahlen siehe ASR_MERGE_2, ASR_MERGE_4, ASR_MERGE_5, ASR_MERGE_6, ASR_MERGE_7.

Zustandsübersicht

Der Baustein besitzt **keinen Zustandsautomaten**. Das Verhalten ist rein kombinatorisch: Jedes eingehende Ereignis an einem der 3 Sockets wird unmittelbar an OUT weitergereicht.

Anwendungsszenarien

- **Zusammenführen mehrerer gleichwertiger Signalquellen** (z. B. 3 redundante oder alternative Auslösepfade) auf einen gemeinsamen ASR-Adapterausgang.
- **Vereinfachung von Netzwerken**, die sonst 3 separate Ereignisverbindungen zum selben Ziel benötigen würden.

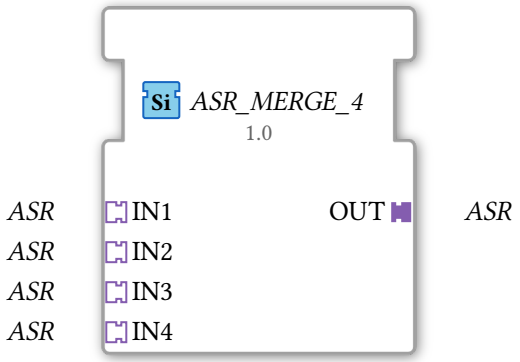
Vergleich mit ähnlichen Bausteinen

- **ASR_SPLIT_2**: die Umkehrrichtung für 2 Ausgänge (bei ASRT zusätzlich ASR_SPLIT_3 bis ASR_SPLIT_9 verfügbar).
- **ASR_MERGE_2, ASR_MERGE_4, ASR_MERGE_5, ASR_MERGE_6, ASR_MERGE_7**: dieselbe generische Implementierung mit anderer Eingangsanzahl.
- AE_MERGE_3, ASRT_MERGE_3: dieselbe Zusammenführungslogik für die jeweils anderen unidirektionalen Ereignisadapter.

Fazit

ASR_MERGE_3 liefert eine generisch implementierte Zusammenführung von 3 ASR-Ereignisquellen auf einen gemeinsamen Adapterausgang und ergänzt die GEN_ASR_MERGE-Familie um die 3-Eingangs-Variante.

ASR_MERGE_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASR_MERGE_4** führt 4 unidirektionale ASR-Adapter (Set/Reset-Ereignisadapter, keine Nutzdaten) auf einen gemeinsamen Ausgangsadapter zusammen. Er ist die 4-Eingangs-Variante von ASR_MERGE_2 und wie dieser als generischer FB implementiert (`GenericClassName: 'GEN_ASR_MERGE'`) — dieselbe C++-Basis (`CGenUnidirectMergeBase`), nur mit 4 statt 2 Sockets.

Schnittstellenstruktur

Ereignis-Eingänge
Keine. Die Ereignisübernahme erfolgt ausschließlich über die Adapter-Sockets.

Ereignis-Ausgänge
Keine. Die Ereignisweitergabe erfolgt ausschließlich über den Adapter-Plug.

Daten-Eingänge
Keine.

Daten-Ausgänge
Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket (Eingang 1)	`IN1`	`adapter::types::unidirectional::ASR`	1. Quelle.
Socket (Eingang 2)	`IN2`	`adapter::types::unidirectional::ASR`	2. Quelle.
Socket (Eingang 3)	`IN3`	`adapter::types::unidirectional::ASR`	3. Quelle.
Socket (Eingang 4)	`IN4`	`adapter::types::unidirectional::ASR`	4. Quelle.
Plug (Ausgang)	`OUT`	`adapter::types::unidirectional::ASR`	Zusammengeführtes Signal.

Funktionsweise
Trifft `SET` bzw. `RESET` an einem der 4 Sockets (`IN1`, `IN2`, `IN3`, `IN4`) ein, wird es als gleichartiges Ereignis unverändert an `OUT` weitergereicht. Alle Ereignistypen werden unabhängig voneinander behandelt, und alle 4 Sockets werden gleichberechtigt behandelt — welcher Socket zuerst geprüft wird, folgt der Reihenfolge `IN1` … `IN4`, was nur bei mehreren innerhalb desselben Ausführungszyklus eintreffenden Ereignissen relevant wird. Da `ASR` keinen Datenwert transportiert, gibt es nichts zu arbitrieren außer dem jeweiligen Ereignistyp selbst.

Technische Besonderheiten
- **Generischer Typ**: Implementiert über die generische Basisklasse `CGenUnidirectMergeBase` (N Sockets, 1 Plug); die Anzahl der Eingangs-Sockets wird über den generischen Suffix (`_4`) zur Übersetzungszeit festgelegt.
- **Keine Zustände / Algorithmen**: Da `ASR` keinen Datenwert trägt, gibt es keine Änderungserkennung und kein ECC.
- **Beliebige Eingangsanzahl**: Dieselbe Implementierung deckt ASR_MERGE_2 bis ASR_MERGE_7 ab; für andere Eingangsanzahlen siehe `ASR_MERGE_2`, `ASR_MERGE_3`, `ASR_MERGE_5`, `ASR_MERGE_6`, `ASR_MERGE_7`.

Zustandsübersicht
Der Baustein besitzt **keinen Zustandsautomaten**. Das Verhalten ist rein kombinatorisch: Jedes eingehende Ereignis an einem der 4 Sockets wird unmittelbar an `OUT` weitergereicht.

Anwendungsszenarien
- **Zusammenführen mehrerer gleichwertiger Signalquellen** (z. B. 4 redundante oder alternative Auslösepfade) auf einen gemeinsamen ASR-Adapterausgang.
- **Vereinfachung von Netzwerken**, die sonst 4 separate Ereignisverbindungen zum selben Ziel benötigen würden.

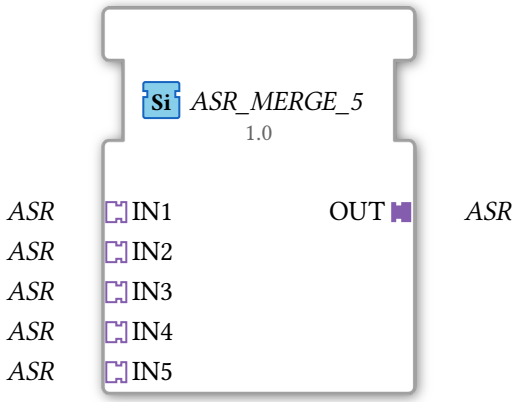
Vergleich mit ähnlichen Bausteinen
- ASR_SPLIT_2: die Umkehrrichtung für 2 Ausgänge (bei ASRT zusätzlich ASR_SPLIT_3 bis ASR_SPLIT_9 verfügbar).
- **`ASR_MERGE_2`, `ASR_MERGE_3`, `ASR_MERGE_5`, `ASR_MERGE_6`, `ASR_MERGE_7`**: dieselbe generische Implementierung mit anderer Eingangsanzahl.

- AE_MERGE_4, ASRT_MERGE_4: dieselbe Zusammenführungslogik für die jeweils anderen unidirektionalen Ereignisadapter.

Fazit

`ASR_MERGE_4` liefert eine generisch implementierte Zusammenführung von 4 `ASR`-Ereignisquellen auf einen gemeinsamen Adapterausgang und ergänzt die `GEN_ASR_MERGE`-Familie um die 4-Eingangs-Variante.

ASR_MERGE_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASR_MERGE_5** führt 5 unidirektionale ASR-Adapter (Set/Reset-Ereignisadapter, keine Nutzdaten) auf einen gemeinsamen Ausgangsadapter zusammen. Er ist die 5-Eingangs-Variante von ASR_MERGE_2 und wie dieser als generischer FB implementiert (`GenericClassName: 'GEN_ASR_MERGE'`) — dieselbe C++-Basis (`CGenUnidirectMergeBase`), nur mit 5 statt 2 Sockets.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignisübernahme erfolgt ausschließlich über die Adapter-Sockets.

Ereignis-Ausgänge

Keine. Die Ereignisweitergabe erfolgt ausschließlich über den Adapter-Plug.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket (Eingang 1)	`IN1`	`adapter::types::unidirectional::ASR`	1. Quelle.
Socket (Eingang 2)	`IN2`	`adapter::types::unidirectional::ASR`	2. Quelle.
Socket (Eingang 3)	`IN3`	`adapter::types::unidirectional::ASR`	3. Quelle.
Socket (Eingang 4)	`IN4`	`adapter::types::unidirectional::ASR`	4. Quelle.
Socket (Eingang 5)	`IN5`	`adapter::types::unidirectional::ASR`	5. Quelle.
Plug (Ausgang)	`OUT`	`adapter::types::unidirectional::ASR`	Zusammengeführtes Signal.

Funktionsweise

Trifft `SET` bzw. `RESET` an einem der 5 Sockets (`IN1`, `IN2`, `IN3`, `IN4`, `IN5`) ein, wird es als gleichartiges Ereignis unverändert an `OUT` weitergereicht. Alle Ereignistypen werden unabhängig voneinander behandelt, und alle 5 Sockets werden gleichberechtigt behandelt — welcher Socket zuerst geprüft wird, folgt der Reihenfolge `IN1` … `IN5`, was nur bei mehreren innerhalb desselben Ausführungszyklus eintreffenden Ereignissen relevant wird. Da `ASR` keinen Datenwert transportiert, gibt es nichts zu arbitrieren außer dem jeweiligen Ereignistyp selbst.

Technische Besonderheiten

- **Generischer Typ**: Implementiert über die generische Basisklasse `CGenUnidirectMergeBase` (N Sockets, 1 Plug); die Anzahl der Eingangs-Sockets wird über den generischen Suffix (`_5`) zur Übersetzungszeit festgelegt.
- **Keine Zustände / Algorithmen**: Da `ASR` keinen Datenwert trägt, gibt es keine Änderungserkennung und kein ECC.
- **Beliebige Eingangsanzahl**: Dieselbe Implementierung deckt ASR_MERGE_2 bis ASR_MERGE_7 ab; für andere Eingangsanzahlen siehe `ASR_MERGE_2`, `ASR_MERGE_3`, `ASR_MERGE_4`, `ASR_MERGE_6`, `ASR_MERGE_7`.

Zustandsübersicht

Der Baustein besitzt **keinen Zustandsautomaten**. Das Verhalten ist rein kombinatorisch: Jedes eingehende Ereignis an einem der 5 Sockets wird unmittelbar an `OUT` weitergereicht.

Anwendungsszenarien

- **Zusammenführen mehrerer gleichwertiger Signalquellen** (z. B. 5 redundante oder alternative Auslösepfade) auf einen gemeinsamen ASR-Adapterausgang.
- **Vereinfachung von Netzwerken**, die sonst 5 separate Ereignisverbindungen zum selben Ziel benötigen würden.

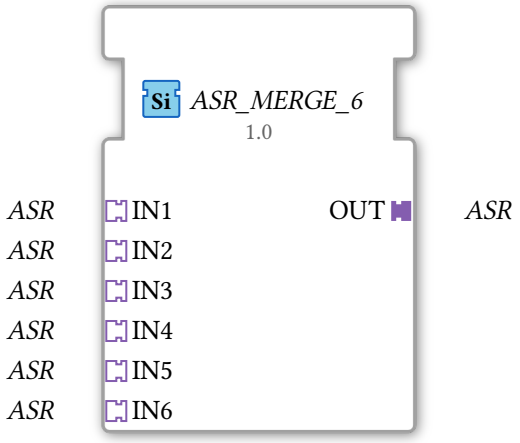
Vergleich mit ähnlichen Bausteinen

- ASR_SPLIT_2: die Umkehrrichtung für 2 Ausgänge (bei ASRT zusätzlich ASR_SPLIT_3 bis ASR_SPLIT_9 verfügbar).
- **`ASR_MERGE_2`, `ASR_MERGE_3`, `ASR_MERGE_4`, `ASR_MERGE_6`, `ASR_MERGE_7`**: dieselbe generische Implementierung mit anderer Eingangsanzahl.
- AE_MERGE_5, ASRT_MERGE_5: dieselbe Zusammenführungslogik für die jeweils anderen unidirektionalen Ereignisadapter.

Fazit

`ASR_MERGE_5` liefert eine generisch implementierte Zusammenführung von 5 `ASR`-Ereignisquellen auf einen gemeinsamen Adapterausgang und ergänzt die `GEN_ASR_MERGE`-Familie um die 5-Eingangs-Variante.

ASR_MERGE_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASR_MERGE_6** führt 6 unidirektionale ASR-Adapter (Set/Reset-Ereignisadapter, keine Nutzdaten) auf einen gemeinsamen Ausgangsadapter zusammen. Er ist die 6-Eingangs-Variante von ASR_MERGE_2 und wie dieser als generischer FB implementiert (`GenericClassName: 'GEN_ASR_MERGE'`) — dieselbe C++-Basis (`CGenUnidirectMergeBase`), nur mit 6 statt 2 Sockets.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignisübernahme erfolgt ausschließlich über die Adapter-Sockets.

Ereignis-Ausgänge

Keine. Die Ereignisweitergabe erfolgt ausschließlich über den Adapter-Plug.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket (Eingang 1)	`IN1`	`adapter::types::unidirectional::ASR`	1. Quelle.
Socket (Eingang 2)	`IN2`	`adapter::types::unidirectional::ASR`	2. Quelle.
Socket (Eingang 3)	`IN3`	`adapter::types::unidirectional::ASR`	3. Quelle.
Socket (Eingang 4)	`IN4`	`adapter::types::unidirectional::ASR`	4. Quelle.
Socket (Eingang 5)	`IN5`	`adapter::types::unidirectional::ASR`	5. Quelle.
Socket (Eingang 6)	`IN6`	`adapter::types::unidirectional::ASR`	6. Quelle.
Plug (Ausgang)	`OUT`	`adapter::types::unidirectional::ASR`	Zusammengeführtes Signal.

Funktionsweise

Trifft `SET` bzw. `RESET` an einem der 6 Sockets (`IN1`, `IN2`, `IN3`, `IN4`, `IN5`, `IN6`) ein, wird es als gleichartiges Ereignis unverändert an `OUT` weitergereicht. Alle Ereignistypen werden unabhängig voneinander behandelt, und alle 6 Sockets werden gleichberechtigt behandelt — welcher Socket zuerst geprüft wird, folgt der Reihenfolge `IN1` … `IN6`, was nur bei mehreren innerhalb desselben Ausführungszyklus eintreffenden Ereignissen relevant wird. Da `ASR` keinen Datenwert transportiert, gibt es nichts zu arbitrieren außer dem jeweiligen Ereignistyp selbst.

Technische Besonderheiten

- **Generischer Typ**: Implementiert über die generische Basisklasse `CGenUnidirectMergeBase` (N Sockets, 1 Plug); die Anzahl der Eingangs-Sockets wird über den generischen Suffix (`_6`) zur Übersetzungszeit festgelegt.
- **Keine Zustände / Algorithmen**: Da `ASR` keinen Datenwert trägt, gibt es keine Änderungserkennung und kein ECC.
- **Beliebige Eingangsanzahl**: Dieselbe Implementierung deckt ASR_MERGE_2 bis ASR_MERGE_7 ab; für andere Eingangsanzahlen siehe `ASR_MERGE_2`, `ASR_MERGE_3`, `ASR_MERGE_4`, `ASR_MERGE_5`, `ASR_MERGE_7`.

Zustandsübersicht

Der Baustein besitzt **keinen Zustandsautomaten**. Das Verhalten ist rein kombinatorisch: Jedes eingehende Ereignis an einem der 6 Sockets wird unmittelbar an `OUT` weitergereicht.

Anwendungsszenarien

- **Zusammenführen mehrerer gleichwertiger Signalquellen** (z. B. 6 redundante oder alternative Auslösepfade) auf einen gemeinsamen ASR-Adapterausgang.
- **Vereinfachung von Netzwerken**, die sonst 6 separate Ereignisverbindungen zum selben Ziel benötigen würden.

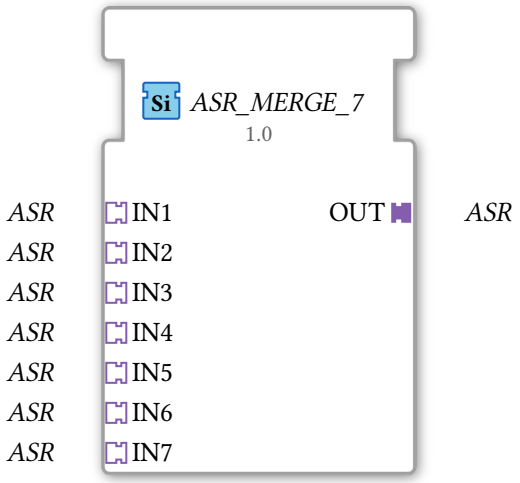
Vergleich mit ähnlichen Bausteinen

- ASR_SPLIT_2: die Umkehrrichtung für 2 Ausgänge (bei ASRT zusätzlich ASR_SPLIT_3 bis ASR_SPLIT_9 verfügbar).
- **ASR_MERGE_2, ASR_MERGE_3, ASR_MERGE_4, ASR_MERGE_5, ASR_MERGE_7**: dieselbe generische Implementierung mit anderer Eingangsanzahl.
- AE_MERGE_6, ASRT_MERGE_6: dieselbe Zusammenführungslogik für die jeweils anderen unidirektionalen Ereignisadapter.

Fazit

`ASR_MERGE_6` liefert eine generisch implementierte Zusammenführung von 6 `ASR`-Ereignisquellen auf einen gemeinsamen Adapterausgang und ergänzt die `GEN_ASR_MERGE`-Familie um die 6-Eingangs-Variante.

ASR_MERGE_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASR_MERGE_7** führt 7 unidirektionale ASR-Adapter (Set/Reset-Ereignisadapter, keine Nutzdaten) auf einen gemeinsamen Ausgangsadapter zusammen. Er ist die 7-Eingangs-Variante von ASR_MERGE_2 und wie dieser als generischer FB implementiert (`GenericClassName: 'GEN_ASR_MERGE'`) — dieselbe C++-Basis (`CGenUnidirectMergeBase`), nur mit 7 statt 2 Sockets.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignisübernahme erfolgt ausschließlich über die Adapter-Sockets.

Ereignis-Ausgänge

Keine. Die Ereignisweitergabe erfolgt ausschließlich über den Adapter-Plug.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Rolle	Name	Typ	Beschreibung
Socket (Eingang 1)	IN1	adapter::types::unidirectional::ASR	1. Quelle.
Socket (Eingang 2)	IN2	adapter::types::unidirectional::ASR	2. Quelle.
Socket (Eingang 3)	IN3	adapter::types::unidirectional::ASR	3. Quelle.
Socket (Eingang 4)	IN4	adapter::types::unidirectional::ASR	4. Quelle.
Socket (Eingang 5)	IN5	adapter::types::unidirectional::ASR	5. Quelle.
Socket (Eingang 6)	IN6	adapter::types::unidirectional::ASR	6. Quelle.
Socket (Eingang 7)	IN7	adapter::types::unidirectional::ASR	7. Quelle.
Plug (Ausgang)	OUT	adapter::types::unidirectional::ASR	Zusammengeführtes Signal.

Funktionsweise

Trifft SET bzw. RESET an einem der 7 Sockets (IN1, IN2, IN3, IN4, IN5, IN6, IN7) ein, wird es als gleichartiges Ereignis unverändert an OUT weitergereicht. Alle Ereignistypen werden unabhängig voneinander behandelt, und alle 7 Sockets werden gleichberechtigt behandelt — welcher Socket zuerst geprüft wird, folgt der Reihenfolge IN1 … IN7, was nur bei mehreren innerhalb desselben Ausführungszyklus eintreffenden Ereignissen relevant wird. Da ASR keinen Datenwert transportiert, gibt es nichts zu arbitrieren außer dem jeweiligen Ereignistyp selbst.

Technische Besonderheiten

- **Generischer Typ**: Implementiert über die generische Basisklasse CGenUnidirectMergeBase (N Sockets, 1 Plug); die Anzahl der Eingangs-Sockets wird über den generischen Suffix (_7) zur Übersetzungszeit festgelegt.
- **Keine Zustände / Algorithmen**: Da ASR keinen Datenwert trägt, gibt es keine Änderungserkennung und kein ECC.
- **Beliebige Eingangsanzahl**: Dieselbe Implementierung deckt ASR_MERGE_2 bis ASR_MERGE_7 ab; für andere Eingangsanzahlen siehe ASR_MERGE_2, ASR_MERGE_3, ASR_MERGE_4, ASR_MERGE_5, ASR_MERGE_6.

Zustandsübersicht

Der Baustein besitzt **keinen Zustandsautomaten**. Das Verhalten ist rein kombinatorisch: Jedes eingehende Ereignis an einem der 7 Sockets wird unmittelbar an OUT weitergereicht.

Anwendungsszenarien

- **Zusammenführen mehrerer gleichwertiger Signalquellen** (z. B. 7 redundante oder alternative Auslösepfade) auf einen gemeinsamen ASR-Adapterausgang.
- **Vereinfachung von Netzwerken**, die sonst 7 separate Ereignisverbindungen zum selben Ziel benötigen würden.

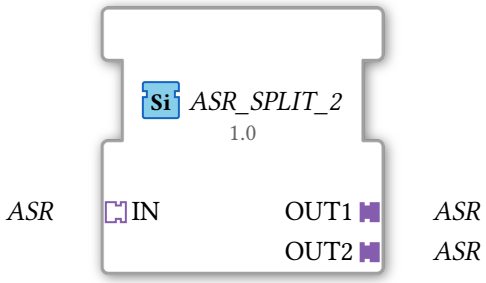
Vergleich mit ähnlichen Bausteinen

- **ASR_SPLIT_2**: die Umkehrrichtung für 2 Ausgänge (bei ASRT zusätzlich ASR_SPLIT_3 bis ASR_SPLIT_9 verfügbar).
- **ASR_MERGE_2, ASR_MERGE_3, ASR_MERGE_4, ASR_MERGE_5, ASR_MERGE_6**: dieselbe generische Implementierung mit anderer Eingangsanzahl.
- AE_MERGE_7, ASRT_MERGE_7: dieselbe Zusammenführungslogik für die jeweils anderen unidirektionalen Ereignisadapter.

Fazit

ASR_MERGE_7 liefert eine generisch implementierte Zusammenführung von 7 ASR-Ereignisquellen auf einen gemeinsamen Adapterausgang und ergänzt die GEN_ASR_MERGE-Familie um die 7-Eingangs-Variante.

ASR_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASR_SPLIT_2** dient der Aufteilung eines eingehenden unidirektionalen ASR-Adapters (Application Service Request) auf zwei identische Ausgangsadapter. Er ist als generischer Baustein konzipiert und ermöglicht es, ein ASR-Signal an zwei separate Verbraucher oder nachfolgende Bausteine weiterzuleiten, ohne die Logik zu verändern.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Typ	Richtung	Name	Beschreibung
`adapter::types::unidirectional::ASR`	Eingang (Socket)	**IN**	Eingehendes ASR-Signal
`adapter::types::unidirectional::ASR`	Ausgang (Plug)	**OUT1**	Erster ausgehender ASR-Pfad
`adapter::types::unidirectional::ASR`	Ausgang (Plug)	**OUT2**	Zweiter ausgehender ASR-Pfad

Funktionsweise

Der Baustein leitet das über den Socket **IN** eingehende ASR-Signal unverändert an beide Plugs **OUT1** und **OUT2** weiter. Es erfolgt keine Verarbeitung, Filterung oder Verzögerung – das Signal wird rein passiv auf zwei Pfade verzweigt. Jeder Ausgang stellt die exakt gleiche Information wie der Eingang bereit.

Technische Besonderheiten

- Der Baustein ist als **Generischer Funktionsblock (Generic FB)** implementiert. Die tatsächliche Typinstanz wird zur Laufzeit durch die Konfiguration des generischen Klassenattributs (`eclipse4diac::core::GenericClassName = 'GEN_ASR_SPLIT'`) festgelegt.
- Es werden keinerlei Ereignis- oder Daten-Eingänge/Ausgänge genutzt – die gesamte Kommunikation erfolgt ausschließlich über die Adapter-Schnittstellen.
- Die Ausgänge sind voneinander unabhängig; ein Fehler oder eine Blockade an einem Ausgang beeinflusst den anderen nicht.

Zustandsübersicht

Der Baustein besitzt kein eigenes Zustandsdiagramm, da er rein passiv arbeitet. Sein Verhalten entspricht einer statischen Verzweigung.

Anwendungsszenarien

- **Parallelisierung von Dienstanfragen:** Ein ASR (z. B. eine Anforderung eines Services) soll an zwei unabhängige Komponenten gesendet werden, z. B. an eine Logging-Komponente und gleichzeitig an einen Ausführungsdienst.
- **Redundanz:** Verteilung eines Signals auf zwei redundante Pfade zur Erhöhung der Ausfallsicherheit.
- **Adapter-Brücke:** Einsatz in Systemen, bei denen eine ASR-Verbindung auf zwei Adapter-Punkte aufgeteilt werden muss, ohne die Logik zu kopieren.

Vergleich mit ähnlichen Bausteinen

- ASR_MERGE_2: Führt zwei ASR-Eingänge auf einen Ausgang zusammen (jedes SET/RESET-Ereignis wird unverändert durchgereicht, keine Priorisierung). Der **ASR_SPLIT_2** macht das Gegenteil.
- **SPLIT_1_to_N** (generische Split-Komponenten): Ein allgemeiner Splitter, der mit einer konfigurierbaren Anzahl von Ausgängen arbeitet. Der **ASR_SPLIT_2** ist eine spezialisierte Version mit fest zwei Ausgängen.

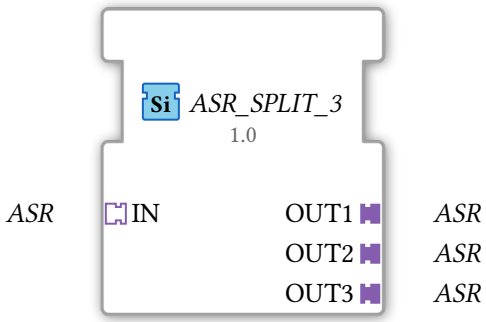
Fazit

Der **ASR_SPLIT_2** ist ein einfacher, aber nützlicher generischer Baustein zur Vervielfachung von ASR-Signalen. Durch die Verwendung von plattformunabhängigen Adaptern und die rein passive Funktionsweise eignet er sich für alle Szenarien, in denen ein Dienstsignal auf mehrere Pfade verteilt werden muss. Seine Integration erfolgt ohne Konfigurationsaufwand und ohne Seiteneffekte auf die übertragenen Daten.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ASR_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ASR_SPLIT_3` dient dazu, einen eingehenden ASR (Application-Specific Resource) Adapter auf drei identische ausgehende ASR-Adapter zu verteilen. Er ist als generischer Baustein konzipiert und ermöglicht eine einfache Signalfächerung innerhalb einer Adapter-basierten Kommunikationsstruktur.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Typ	Richtung	Name	Beschreibung
Socket	Eingang	IN	Eingehender ASR-Signaladapter.
Plug	Ausgang	OUT1	Erster ausgehender ASR-Signaladapter.
Plug	Ausgang	OUT2	Zweiter ausgehender ASR-Signaladapter.
Plug	Ausgang	OUT3	Dritter ausgehender ASR-Signaladapter.

Alle Adapter sind vom Typ `adapter::types::unidirectional::ASR`.

Funktionsweise

Der Baustein leitet alle über den Socket `IN` eintreffenden Signale und Ereignisse des ASR-Adapters unverändert an die drei Plug-Kanäle `OUT1`, `OUT2` und `OUT3` weiter. Es findet keine logische Verarbeitung oder Pufferung statt – die Verteilung erfolgt passiv und ohne Verzögerung. Der Baustein fungiert als einfacher Splitter für ASR-Verbindungen.

Technische Besonderheiten

- Der Baustein ist generisch (Attribut `eclipse4diac::core::GenericClassName = 'GEN_ASR_SPLIT'`), wodurch er für verschiedene Instanzen des ASR-Typs wiederverwendet werden kann.
- Es wird kein ECC (Execution Control Chart) benötigt, da die Weiterleitung direkt durch die Adapter-Kopplung erfolgt.
- Alle drei Ausgänge sind separat adressierbar, sodass unterschiedliche Zielbausteine unabhängig angeschlossen werden können.

Zustandsübersicht

Da der Baustein keine Ereignisse verarbeitet und zustandslos arbeitet, existiert kein Zustandsdiagramm. Die Funktionalität beschränkt sich auf die passive Durchleitung.

Anwendungsszenarien

- **Verteilen von Steuersignalen** an mehrere parallele Komponenten in einer Automatisierungsanlage, z. B. an drei Antriebe oder Sensoren.
- **Aufteilung einer ASR-Kommunikation** auf unterschiedliche Subsysteme, die die gleiche Adapterschnittstelle erwarten.
- **Test- und Debugging-Setups**, bei denen ein Signal gleichzeitig an mehrere Überwachungsstellen weitergeleitet werden soll.

Vergleich mit ähnlichen Bausteinen

- `ASR_SPLIT_2` teilt ein ASR-Signal auf zwei Ausgänge auf; `ASR_SPLIT_3` erweitert diese Funktionalität auf drei Ausgänge.
- `ASR_MERGE` (sofern vorhanden) fasst mehrere ASR-Signale zusammen, während `ASR_SPLIT_3` sie auseinanderfädelt.
- Im Gegensatz zu generischen Daten-Splittern arbeitet der Baustein direkt auf Adapterebene und erfordert keine zusätzlichen Datentypumsetzungen.

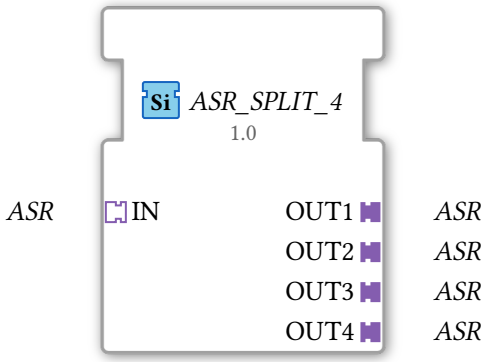
Fazit

Der `ASR_SPLIT_3` ist ein einfacher, aber effektiver Baustein zur Vervielfältigung eines ASR-Signals auf drei identische Ausgänge. Er eignet sich besonders für modulare Strukturen, in denen ein zentrales Signal mehrfach gebraucht wird. Die generische Auslegung erhöht die Wiederverwendbarkeit und reduziert den Konfigurationsaufwand.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ASR_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASR_SPLIT_4** ist ein generischer Baustein, der einen eingehenden unidirektionalen ASR-Adapter (Antriebs-Steuerungs-Adapter) auf vier separate, identische ASR-Ausgänge verteilt. Er ermöglicht die gleichzeitige Ansteuerung von bis zu vier unabhängigen Aktoren oder Subsystemen mit demselben Signal, ohne dass die Signalintegrität beeinträchtigt wird. Der Baustein ist für den Einsatz in verteilten Automatisierungssystemen nach IEC 61499 konzipiert.

Schnittstellenstruktur

Adapter-Eingänge (Sockets)

Name	Typ	Beschreibung
IN	`adapter::types::unidirectional::ASR`	Eingangs-Adapter, der die zu verteilende ASR-Verbindung bereitstellt.

Adapter-Ausgänge (Plugs)

Name	Typ	Beschreibung
OUT1	`adapter::types::unidirectional::ASR`	Erster Ausgangs-Adapter (identische Kopie des Eingangs).
OUT2	`adapter::types::unidirectional::ASR`	Zweiter Ausgangs-Adapter.
OUT3	`adapter::types::unidirectional::ASR`	Dritter Ausgangs-Adapter.
OUT4	`adapter::types::unidirectional::ASR`	Vierter Ausgangs-Adapter.

> **Hinweis:** Der konkrete Aufbau des ASR-Adapters (Ereignis-/Daten-Elemente) ist nicht im Baustein selbst definiert, sondern wird durch den zugehörigen Adaptertyp `adapter::types::unidirectional::ASR` festgelegt. Typischerweise umfasst dieser jedoch Steuer- und Rückmeldesignale für Antriebe.

Funktionsweise

Der ASR_SPLIT_4-Baustein arbeitet als reiner Signalverteiler: Alle über den `IN`-Socket eingehenden ASR-Verbindungselemente (Ereignisse und Daten) werden **1:1 auf alle vier Ausgangs-Plugs** (OUT1–OUT4) repliziert. Es erfolgt keine Signalverstärkung, Filterung oder logische Verarbeitung. Sobald der Eingangs-Adapter eine aktive Verbindung herstellt, wird diese sofort auf alle Ausgänge durchgeschaltet. Der Baustein benötigt keine eigene Zustandslogik – er agiert vollständig kombinatorisch.

Technische Besonderheiten

- **Generischer Funktionsblock:** Der Baustein ist als generischer Typ (`GEN_ASR_SPLIT`) deklariert, was bedeutet, dass die Schnittstellendefinition des ASR-Adapters zur Kompilierzeit aufgelöst wird. Er kann mit verschiedenen Ausprägungen des ASR-Adapters verwendet werden, solange diese dem unidirektionalen Protokoll entsprechen.
- **Unidirektionale Signalrichtung:** Die Adapter sind als unidirectional ausgelegt, d.h. Daten fließen nur vom Eingang zu den Ausgängen (kein Rückkanal vom Ausgang zum Eingang).
- **Keine Latenzpufferung:** Alle Ausgänge werden gleichzeitig mit demselben Signal versorgt; es gibt keine zeitliche Staffelung oder Zwischenspeicherung.

Zustandsübersicht

Der FB besitzt **keine expliziten internen Zustände**. Die Ausgänge folgen stets direkt den Eingangssignalen. Eine Zustandsmaschine ist nicht vorhanden. Das Verhalten kann als eine einzige „Durchschalt"-Operation beschrieben werden.

Anwendungsszenarien

- **Steuerung mehrerer paralleler Antriebe:** Ein zentraler Fahrtbefehlsgeber (z.B. ein AGV-Steuerblock) verteilt das Geschwindigkeits- oder Richtungssignal auf vier unabhängige Motorcontroller.
- **Redundante Signalverteilung:** In sicherheitskritischen Systemen kann derselbe Befehl an mehrere redundante Aktuatoren gesendet werden, um Ausfallsicherheit zu gewährleisten.
- **Test- und Simulationsumgebungen:** Ein simuliertes ASR-Signal wird gleichzeitig an mehrere Testinstanzen angelegt.

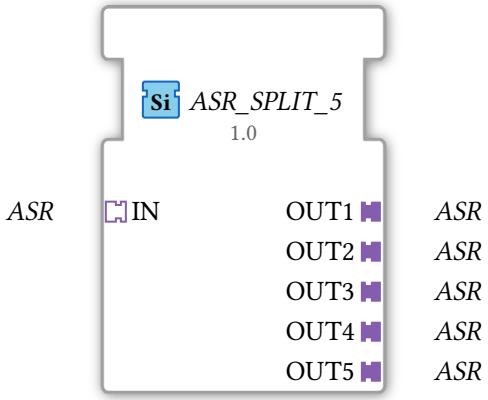
Vergleich mit ähnlichen Bausteinen

- **ASR_SPLIT_2 / ASR_SPLIT_N:** Diese Bausteine teilen den ASR-Eingang auf zwei bzw. eine beliebige Anzahl von Ausgängen auf. Der `ASR_SPLIT_4` ist eine spezielle, vierkanalige Variante.
- **Signal-Multiplexer:** Im Gegensatz zu einem echten Multiplexer, der aus mehreren Quellen auswählt, verteilt der Split-Baustein die *gleiche* Quelle auf mehrere Senken.
- **Adapter-Coupler:** Manche Bibliotheken bieten Koppel-Bausteine für die Adapter-Weiterleitung; der Split-Baustein erweitert dies um die Vervielfachung.

Fazit

Der **ASR_SPLIT_4** ist ein einfacher, aber nützlicher Funktionsblock, der die modulare und wiederverwendbare Kopplung von ASR-Adapter-Verbindungen in IEC-61499-Systemen ermöglicht. Seine generische Implementierung erlaubt den Einsatz in verschiedenen Kontexten, in denen ein Eingangssignal auf mehrere identische Ausgangsschnittstellen verteilt werden muss. Dank des adaptiven, zustandslosen Designs fügt er sich nahtlos in ereignisgesteuerte Automatisierungsarchitekturen ein.

ASR_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASR_SPLIT_5** dient der Aufteilung eines einzelnen unidirektionalen ASR-Adapters (Agricultural Standardized Interface) auf fünf gleichartige ASR-Ausgänge. Er ist als generischer Baustein

konzipiert und ermöglicht so die einfache Verteilung eines ASR-Signals an mehrere nachgelagerte Komponenten oder Steuerungen.

Schnittstellenstruktur

Der Baustein besitzt keine klassischen Ereignis- oder Datenein-/ausgänge. Die gesamte Kommunikation erfolgt über Adapter-Schnittstellen vom Typ `adapter::types::unidirectional::ASR`.

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Bezeichnung	Typ	Richtung
`IN`	`adapter::types::unidirectional::ASR`	Socket (Eingang)
`OUT1`	`adapter::types::unidirectional::ASR`	Plug (Ausgang)
`OUT2`	`adapter::types::unidirectional::ASR`	Plug (Ausgang)
`OUT3`	`adapter::types::unidirectional::ASR`	Plug (Ausgang)
`OUT4`	`adapter::types::unidirectional::ASR`	Plug (Ausgang)
`OUT5`	`adapter::types::unidirectional::ASR`	Plug (Ausgang)

Funktionsweise

Der Baustein ist ein reiner Multiplexer auf Adapterebene. Das am Socket `IN` anliegende ASR-Signal wird unverändert und ohne Verzögerung auf alle fünf Ausgangs-Plugs (`OUT1` … `OUT5`) durchgeschaltet. Somit empfängt jeder angeschlossene Empfänger das identische Signal des Eingangsadapters. Eine aktive Signalbearbeitung oder -aufbereitung findet nicht statt.

Technische Besonderheiten

- **Generischer Typ**: Der Baustein ist als generischer Funktionsblock (`GEN_ASR_SPLIT`) realisiert und kann automatisch an die spezifischen Datentypen des verwendeten ASR-Adapters angepasst werden.
- **Unidirektionale Kommunikation**: Alle Adapter sind unidirektional (`adapter::types::unidirectional::ASR`), d.h. Daten fließen nur vom Eingang zu den Ausgängen, Rückmeldungen sind nicht vorgesehen.
- **Keine Zustandslogik**: Da keine Ereignisse oder Algorithmen vorhanden sind, besitzt der Baustein kein ECC-Zustandsdiagramm und arbeitet ausschließlich datenflussorientiert.

Zustandsübersicht

Der Funktionsblock hat keinen internen Zustandsautomaten. Er verhält sich wie eine passive Signalverteilung und benötigt keine Zustandssteuerung.

Anwendungsszenarien

- **Signalverteilung**: Ein ASR-Signal (z. B. Sensordaten, Steuerbefehle) muss gleichzeitig an mehrere Verbraucher (z. B. mehrere Aktoren oder übergeordnete Steuerungen) weitergegeben werden.
- **Test- und Simulationsumgebungen**: Paralleles Einspeisen eines Testsignals in mehrere unabhängige Systempfade.
- **Redundanz**: Versorgung mehrerer gleichartiger Steuerungseinheiten mit der gleichen Eingangsinformation.

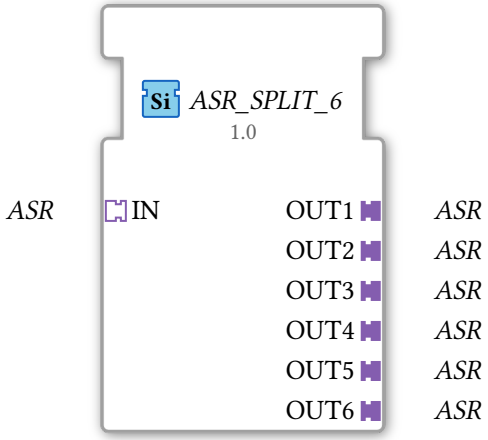
Vergleich mit ähnlichen Bausteinen

- **ASR_SPLIT_2 / ASR_SPLIT_3 / ASR_SPLIT_4**: Diese Bausteine teilen ein ASR-Signal auf zwei, drei bzw. vier Ausgänge auf – je nach Bedarf kann die passende Anzahl gewählt werden.
- **ASR_MERGE**: Im Gegensatz zum Splitter fasst ein MERGE mehrere ASR-Eingänge zu einem Ausgang zusammen.

Fazit

Der **ASR_SPLIT_5** ist ein einfacher, aber nützlicher Routing-Baustein zur Vervielfachung eines unidirektionalen ASR-Adapters. Dank seiner generischen Implementierung und der reinen Adapter-Schnittstelle fügt er sich nahtlos in IEC-61499-Anwendungen ein und erleichtert die strukturierte Signalverteilung in komplexen Automatisierungssystemen.

ASR_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ASR_SPLIT_6` verteilt ein eingehendes ASR-Signal (Generic Adapter Typ) auf sechs identische ASR-Ausgänge. Er dient als reiner Signalverteiler (Splitter) und wird als generischer Baustein in der 4diac-IDE bereitgestellt.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Adapter	Richtung	Typ	Beschreibung
`IN`	Socket	`adapter::types::unidirectional::ASR`	Eingangsadapter – das zu verteilende ASR-Signal
`OUT1` – `OUT6`	Plug	`adapter::types::unidirectional::ASR`	Sechs Ausgangsadapter – identische Kopien des Eingangssignals

Funktionsweise

Der Baustein führt keine Datenverarbeitung oder Zustandslogik durch. Er leitet den am Socket `IN` anliegenden Adapter (unidirektionales ASR) direkt und ohne Verzögerung an alle sechs Plugs (`OUT1` bis `OUT6`) weiter. Somit liegt an jedem Ausgang dasselbe Signal wie am Eingang an. Die Verteilung geschieht rein auf Adapterebene, ohne dass Ereignisse oder Daten-Ein-/Ausgänge beteiligt sind.

Technische Besonderheiten

- **Generischer Baustein**: Der FB verwendet die generische Typdefinition `GEN_ASR_SPLIT` (Attribut `eclipse4diac::core::GenericClassName`). Dies erlaubt eine flexible Wiederverwendung mit unterschiedlichen ASR-Adapter-Typen.
- **Unidirektionaler Adapter**: Sowohl Eingang als auch Ausgänge sind vom Typ `adapter::types::unidirectional::ASR`, d. h. die Daten fließen nur in einer Richtung (vom Socket zu den Plugs).
- **Keine Zustandslogik**: Der Baustein besitzt kein ECC (Execution Control Chart) und keine internen Zustände – es handelt sich um eine reine Verbindung ohne zeitliches Verhalten.
- **Skalierbarkeit**: Durch das generische Konzept können ähnliche Splitter mit beliebig vielen Ausgängen (z. B. 2, 4, 8) erstellt werden.

Zustandsübersicht

Es existiert kein Zustandsdiagramm (ECC), da der Baustein keine sequentielle Logik ausführt. Die Signalverteilung erfolgt statisch und kontinuierlich.

Anwendungsszenarien

- **Verteilung eines Sensorsignals** an mehrere Steuerungseinheiten oder Subsysteme.
- **Bereitstellung desselben ASR-Kommunikationskanals** für parallele Verarbeitungszweige.
- **Test- und Simulationsumgebungen**, in denen ein Eingangssignal mehrfach benötigt wird.

Vergleich mit ähnlichen Bausteinen

Baustein	Ausgänge	Besonderheit
`ASR_SPLIT_2`	2	Zweifach-Split
`ASR_SPLIT_4`	4	Vierfach-Split
`ASR_SPLIT_6`	**6**	**Sechsfach-Split (dieser Baustein)**
Generischer Split (z. B. über Template)	variabel	Erfordert individuelle Parametrisierung

Der `ASR_SPLIT_6` bietet eine feste Anzahl von sechs Ausgängen und ist damit speziell für Anwendungen ausgelegt, die genau diese Anzahl benötigen.

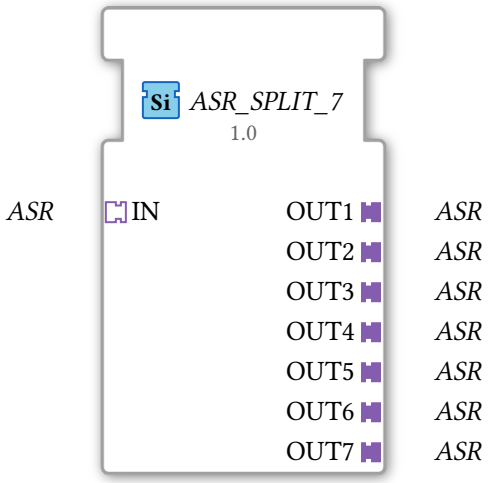
Fazit

Der `ASR_SPLIT_6` ist ein einfacher, aber nützlicher Funktionsblock zur Signalvervielfältigung auf Adapterebene. Durch seine generische Natur und die klare, ereignislose Funktionsweise eignet er sich ideal für modulare Steuerungsarchitekturen, die eine Mehrfachnutzung eines ASR-Signals erfordern.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ASR_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ASR_SPLIT_7** ist ein generischer Baustein, der einen einzelnen ASR-Adaptereingang auf sieben ASR-Adapterausgänge verteilt. Er dient als 1:7-Splitter für ASR-Daten und ermöglicht die parallele Weitergabe eines Signals an mehrere nachfolgende Komponenten.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Eingang (Socket)	**IN**	`adapter::types::unidirectional::ASR`	Quelladapter, dessen Daten auf alle Ausgänge verteilt werden.
Ausgang (Plug)	**OUT1**	`adapter::types::unidirectional::ASR`	Erster Ausgang, erhält die duplizierten Daten von `IN`.
Ausgang (Plug)	**OUT2**	`adapter::types::unidirectional::ASR`	Zweiter Ausgang.
Ausgang (Plug)	**OUT3**	`adapter::types::unidirectional::ASR`	Dritter Ausgang.
Ausgang (Plug)	**OUT4**	`adapter::types::unidirectional::ASR`	Vierter Ausgang.
Ausgang (Plug)	**OUT5**	`adapter::types::unidirectional::ASR`	Fünfter Ausgang.
Ausgang (Plug)	**OUT6**	`adapter::types::unidirectional::ASR`	Sechster Ausgang.
Ausgang (Plug)	**OUT7**	`adapter::types::unidirectional::ASR`	Siebter Ausgang.

Funktionsweise

Der Baustein empfängt über den Socket `IN` ein ASR-Signal. Dieses Signal wird intern auf alle sieben Plugs (`OUT1` … `OUT7`) dupliziert und ohne Veränderung weitergeleitet. Es findet keine Datenmanipulation oder Filterung statt – das eingehende Signal wird eins zu eins an jeden Ausgang kopiert.

Technische Besonderheiten

- **Generischer Typ**: Der Funktionsblock ist als generischer Baustein mit dem Klassennamen `GEN_ASR_SPLIT` deklariert. Dadurch kann er in verschiedenen Kontexten eingesetzt werden, in denen ein ASR-Splitter benötigt wird.
- **Reine Adapter-Schnittstelle**: Es werden ausschließlich Adapter (keine Ereignisse oder Daten-Ein-/Ausgänge) verwendet. Die Kommunikation erfolgt über das unidirektionale ASR-Adapterprotokoll.
- **Kein interner Zustand**: Der Baustein besitzt keine Zustandsautomatik; die Signalverteilung erfolgt kontinuierlich und verzögerungsfrei.

Zustandsübersicht

Der Baustein besitzt keine expliziten Zustände. Die Funktionalität beschränkt sich auf die statische Duplizierung des Eingangssignals.

Anwendungsszenarien

- **Signalverteilung**: Ein ASR-Signal (z. B. ein Steuerbefehl oder eine Konfigurationsnachricht) soll an mehrere unabhängige Empfänger gesendet werden.
- **Parallelverarbeitung**: In einer automatisierten Steuerung muss eine Information gleichzeitig mehrere Subsysteme erreichen, die jeweils auf das ASR-Protokoll ausgelegt sind.
- **Testumgebungen**: Ein Testadapter wird an `IN` angeschlossen und die Ausgänge mit verschiedenen Überwachungs- oder Logging-Bausteinen verbunden.

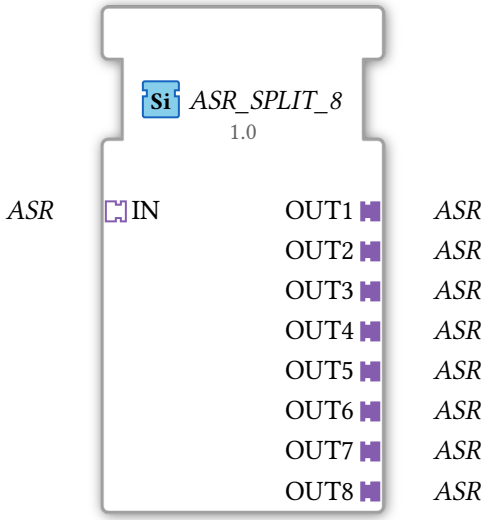
Vergleich mit ähnlichen Bausteinen

Es existieren weitere Splitter-Bausteine wie `ASR_SPLIT_2`, `ASR_SPLIT_4` oder `ASR_SPLIT_n`, die das Eingangssignal auf 2, 4 bzw. `n` Ausgänge verteilen. `ASR_SPLIT_7` stellt eine spezifische Variante mit sieben Ausgängen dar. Allen gemeinsam ist die unveränderte Duplizierung des eingehenden ASR-Signals.

Fazit

Der `ASR_SPLIT_7` ist ein einfacher, aber nützlicher Baustein zur Verteilung eines ASR-Signals auf sieben parallele Pfade. Seine generische Definition und der Verzicht auf Ereignisse oder Zustände machen ihn flexibel einsetzbar und robust im Betrieb.

ASR_SPLIT_8



(Bild des Funktionsblocks hier einfügen)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ASR_SPLIT_8` dient dazu, ein einzelnes eingehendes **ASR**-Adapter-Signal unverändert auf acht identische Ausgänge zu verteilen. Er wird als generischer Baustein definiert und eignet sich besonders für Anwendungen, in denen ein Sensorsignal oder Steuersignal mehrfach benötigt wird – beispielsweise in der landwirtschaftlichen Automatisierungstechnik.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden. Der Funktionsblock arbeitet rein adapterbasiert ohne ereignisgesteuerte Ausführung.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden. Alle Signalübertragungen erfolgen über die **Adapter**-Schnittstellen.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	IN	`adapter::types::unidirectional::ASR`	Eingehendes ASR-Signal, das auf alle Ausgänge verteilt wird.
Plug	OUT1	`adapter::types::unidirectional::ASR`	Erster Ausgang mit dem Signal von IN.
Plug	OUT2	`adapter::types::unidirectional::ASR`	Zweiter Ausgang mit dem Signal von IN.
Plug	OUT3	`adapter::types::unidirectional::ASR`	Dritter Ausgang mit dem Signal von IN.
Plug	OUT4	`adapter::types::unidirectional::ASR`	Vierter Ausgang mit dem Signal von IN.
Plug	OUT5	`adapter::types::unidirectional::ASR`	Fünfter Ausgang mit dem Signal von IN.
Plug	OUT6	`adapter::types::unidirectional::ASR`	Sechster Ausgang mit dem Signal von IN.
Plug	OUT7	`adapter::types::unidirectional::ASR`	Siebter Ausgang mit dem Signal von IN.
Plug	OUT8	`adapter::types::unidirectional::ASR`	Achter Ausgang mit dem Signal von IN.

Funktionsweise

Der `ASR_SPLIT_8`-Block stellt eine **1-zu-8-Signalverteilung** für den ASR-Adaptertyp dar. Das am Socket `IN` anliegende ASR-Signal wird ohne Verzögerung oder Zustandsänderung an alle acht Plug-Ausgänge (`OUT1`…`OUT8`) weitergeleitet. Es findet keine Signalaufbereitung, Filterung oder logische Verknüpfung statt – der Block arbeitet rein passiv und ohne eigenes Verhalten.

Da der Adaptertyp `ASR` als unidirectional definiert ist, fließen die Daten nur vom Socket zu den Plugs. Änderungen am Eingangssignal wirken sich unmittelbar auf alle Ausgänge aus.

Technische Besonderheiten

- **Generische Typparametrisierung**: Der FB besitzt die Attribute `eclipse4diac::core::GenericClassName` (Wert `'GEN_ASR_SPLIT'`) und `eclipse4diac::core::TypeHash`, welche die Verwendung eines generischen ASR-Datentyps ermöglichen. Dadurch kann der Baustein für verschiedene ASR-Untertypen wiederverwendet werden.

- **Keine Ereignissteuerung**: Der Baustein enthält keine Ereignisein- oder -ausgänge; die Signalweitergabe erfolgt ausschließlich über die Adapter-Schnittstellen. Dies vereinfacht die Einbindung in statische Datenflüsse.
- **Kein internes Verhalten**: Es existiert keine Zustandsmaschine oder Algorithmus – die Funktionalität beschränkt sich auf die reine Verdrahtungslogik der 4diac-IDE.

Zustandsübersicht

Der `ASR_SPLIT_8` besitzt **keine internen Zustände**. Die Ausgabe ist zu jedem Zeitpunkt eine direkte Funktion des Eingangssignals. Der Block verhält sich wie eine passive Verdrahtung (steriler Splitter) ohne Speicher- oder Verzögerungseffekte.

Anwendungsszenarien

- **Verteilung eines Sensorsignals** (z. B. Drehzahl, Druck oder Position) an mehrere parallele Regelkreise oder Anzeigeeinheiten.
- **Aufteilung eines Steuerbefehls** auf mehrere Aktoren in einer landwirtschaftlichen Maschine, wie z. B. die gleichzeitige Ansteuerung mehrerer Ventile oder Antriebe.
- **Monitoring und Logging**: Ein Signal kann über einen Ausgang an eine Steuerung und über einen zweiten Ausgang an ein Diagnosesystem weitergeleitet werden.

Vergleich mit ähnlichen Bausteinen

Baustein	Ausgänge	Besonderheit
`ASR_SPLIT_8`	8	Bietet die maximal mögliche Anzahl an ASR-Kopien in einem Block.
`ASR_SPLIT_4`	4	Kompaktere Alternative für kleinere Verteilungen.
`ASR_SPLIT_2`	2	Minimaler Splitter für Punkt-zu-Punkt-Weitergabe.

Die **8er-Version** ist die umfangreichste und erlaubt den Anschluss vieler Verbraucher ohne zusätzliche Kaskadierung. Gegenüber einer Kaskade mehrerer 2er- oder 4er-Splitter reduziert `ASR_SPLIT_8` die Anzahl der benötigten Blöcke und vereinfacht das Engineering.

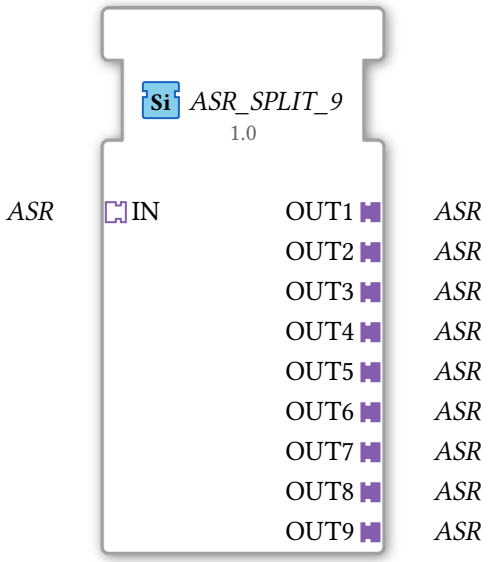
Fazit

Der `ASR_SPLIT_8` ist ein einfacher, aber effizienter Baustein, um ein ASR-Signal auf acht unabhängige Pfade zu verteilen. Durch seine generische Auslegung und das Fehlen von Zuständen und Ereignissen ist er robust, ressourcenschonend und ideal für unidirektionale Datenweiterleitungen in landwirtschaftlichen und industriellen Automatisierungssystemen geeignet. Er ergänzt die Familie der ASR-Splitter und deckt den Bedarf an einer hohen Anzahl paralleler Ausgänge ab.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ASR_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **ASR_SPLIT_9** dient dazu, einen einzelnen eingehenden ASR-Adapter (Application Specific Resource) auf neun separate ASR-Ausgänge aufzuteilen. Er ist als generischer Baustein (Generic FB) implementiert und ermöglicht die Verteilung eines ASR-Signals an bis zu neun unabhängige Empfänger.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine separaten Dateneingänge. Der gesamte Datenaustausch erfolgt über den ASR-Adapter **IN**.

Daten-Ausgänge

Keine separaten Datenausgänge. Die Daten werden über die ASR-Adapter **OUT1** bis **OUT9** ausgegeben.

Adapter

Socket (Eingang):

Name	Typ	Richtung
IN	`adapter::types::unidirectional::ASR`	Socket

Plugs (Ausgänge):

Name	Typ	Richtung
OUT1	`adapter::types::unidirectional::ASR`	Plug
OUT2	`adapter::types::unidirectional::ASR`	Plug
OUT3	`adapter::types::unidirectional::ASR`	Plug
OUT4	`adapter::types::unidirectional::ASR`	Plug
OUT5	`adapter::types::unidirectional::ASR`	Plug

OUT6	`adapter::types::unidirectional::ASR`	Plug
OUT7	`adapter::types::unidirectional::ASR`	Plug
OUT8	`adapter::types::unidirectional::ASR`	Plug
OUT9	`adapter::types::unidirectional::ASR`	Plug

Funktionsweise

Der Baustein empfängt über den Socket-Adapater **IN** ein ASR-Signal. Dieses Signal wird intern an alle neun Ausgangs-Adapter (**OUT1** bis **OUT9**) weitergeleitet. Somit liegt an jedem Ausgang das identische ASR-Signal an. Eine Verzögerung oder Pufferung findet nicht statt. Der Baustein ist rein passiv und führt keine Transformation der Daten durch.

Technische Besonderheiten

- **Generischer Baustein:** Der Typ ist als generischer FB (GenericClassName `'GEN_ASR_SPLIT'`) ausgelegt, sodass er je nach Projektkontext mit unterschiedlichen ASR-Typen parametrisiert werden kann.
- **Unidirektionale Adapter:** Alle verwendeten Adapter sind vom Typ `unidirectional::ASR`, was eine klare Datenflussrichtung vom Eingang zu den Ausgängen vorgibt.
- **Keine Ereignisse:** Der Baustein besitzt keine Ereignis-Eingänge oder -Ausgänge. Die Datenweitergabe erfolgt rein über die Adapter-Schnittstellen.
- **Plattformneutral:** Durch die Verwendung des IEC 61499-2 Standards ist der Baustein in verschiedenen 4diac-IDE Projekten einsetzbar.

Zustandsübersicht

Der Baustein besitzt keinen internen Zustandsautomaten (ECC). Er arbeitet statisch und gibt das eingehende Signal ohne zeitliche oder logische Abhängigkeiten an alle Ausgänge weiter.

Anwendungsszenarien

- **Verteilung eines ASR-Signals** an mehrere parallel arbeitende Ressourcen oder Funktionseinheiten.
- **Monitoring oder Parallelsteuerung**, wenn mehrere Komponenten dieselben ASR-Daten benötigen (z. B. ein Sensorwert an mehrere Auswertebausteine).
- **Einfache Bus-ähnliche Strukturen** in automatisierten Steuerungen, ohne aufwendige Multicast-Mechanismen.

Vergleich mit ähnlichen Bausteinen

- **ASR_SPLIT_4 / ASR_SPLIT_2:** Bausteine mit geringerer Ausgangsanzahl (4 bzw. 2). `ASR_SPLIT_9` bietet die maximale Aufteilung auf neun Kanäle.
- **ASR_MERGE:** Führt mehrere ASR-Signale zu einem zusammen, während `ASR_SPLIT_9` genau die umgekehrte Funktion erfüllt.
- **Data Distributor:** Bei Datentyp-orientierten Bausteinen (z. B. `DATA_SPLIT`) wird eher mit einfachen Datentypen gearbeitet. `ASR_SPLIT_9` ist speziell für komplexe Adapterstrukturen optimiert.

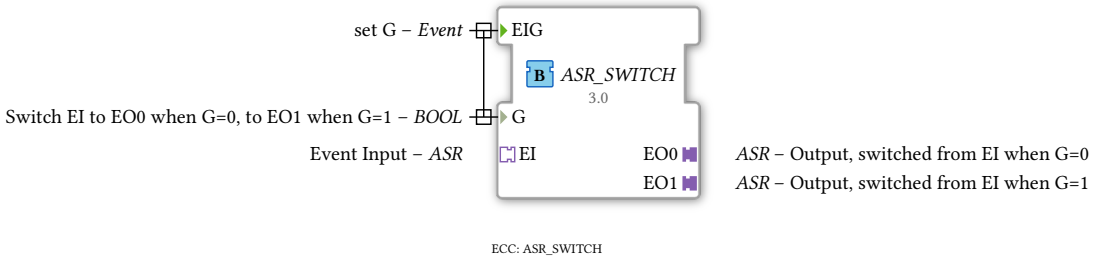
Fazit

Der Funktionsbaustein **ASR_SPLIT_9** ist eine kompakte und effiziente Lösung zur Verteilung eines ASR-Signals auf neun identische Ausgänge. Durch den generischen Charakter und die klare, ereignislose Adapterkopplung eignet er sich besonders für modulare, wiederverwendbare Automatisierungssysteme auf Basis von IEC 61499.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ASR_SWITCH



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ASR_SWITCH** dient als Demultiplexer für **ASR-Ereignisse** (SET/RESET). Er leitet die über den Adapter-Socket `EI` empfangenen Ereignisse wahlweise an den Adapter-Plug `EO0` oder `EO1` weiter – abhängig vom booleschen Wert des Eingangs `G`. Dadurch lassen sich zwei verschiedene Signalpfade in einer Automatisierungsanwendung umschalten.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Datenverbund	Beschreibung
`EIG`	`G`	Setzt den Schaltwert `G` auf den am Eingang `G` anliegenden Booleschen Wert.

Ereignis-Ausgänge

Der Baustein besitzt keine eigenständigen Ereignisausgänge. Die Ausgabe erfolgt ausschließlich über die **Adapterplugs**.

Daten-Eingänge

Variable	Typ	Beschreibung
`G`	`BOOL`	Schalterposition: `FALSE` ⟶ Ereignisse werden an `EO0` weitergeleitet, `TRUE` ⟶ an `EO1`.

Daten-Ausgänge

Keine.

Adapter

Adapter	Richtung	Typ	Beschreibung
EI	Socket	adapter::types::unidirectional::ASR	Empfang der hereinlaufenden ASR-Ereignisse (SET/RESET).
EO0	Plug	adapter::types::unidirectional::ASR	Ausgang für den Fall G = FALSE.
EO1	Plug	adapter::types::unidirectional::ASR	Ausgang für den Fall G = TRUE.

Funktionsweise

Der **ASR_SWITCH** wird durch ein eingehendes Ereignis vom Adapter-Socket EI oder durch das Ereignis EIG aktiviert. Die Verarbeitung erfolgt in der **Ereignisgesteuerten Ablaufsteuerung (ECC)**:

1. **Schalten von G:** Trifft das Ereignis EIG ein, wird der Wert von G übernommen. Der Baustein bleibt im Startzustand START und wartet auf die nächsten Ereignisse.
2. **Weiterleitung von EI.SET:**
   - EI.SET und G = FALSE: Übergang in den Zustand G0_SET, dort wird EO0.SET ausgegeben.
   - EI.SET und G = TRUE : Übergang in den Zustand G1_SET, dort wird EO1.SET ausgegeben.
3. **Weiterleitung von EI.RESET:**
   - EI.RESET und G = FALSE: Übergang in den Zustand G0_RESET, dort wird EO0.RESET ausgegeben.
   - EI.RESET und G = TRUE : Übergang in den Zustand G1_RESET, dort wird EO1.RESET ausgegeben.
4. **Rückkehr:** Nach Ausgabe der Aktion springt die ECC sofort (Bedingung = 1) zurück in den Startzustand START, um das nächste Ereignis zu verarbeiten.

Technische Besonderheiten

- Der Baustein arbeitet ausschließlich mit dem **unidirektionalen Adapter-Typ ASR**, das heißt, er kann sowohl SET- als auch RESET-Ereignisse über denselben Adapter transportieren.
- Die Umschaltung erfolgt **ereignisgetriggert** über EIG (nicht zyklisch).
- Es werden keine Datenausgänge benötigt – die gesamte Signalisierung erfolgt über die Adapterplugs.
- Durch das sofortige Zurücksetzen in den Startzustand ist der Baustein **deterministisch und schnell** wiedereintrittsbereit.

Zustandsübersicht

Die folgende Tabelle fasst die ECC-Zustände und die ausgeführten Aktionen zusammen:

Zustand	Beschreibung	Aktion(en)
START	Warten auf Ereignisse	–
G0_SET	EI.SET bei G = FALSE	Ausgabe EO0.SET
G1_SET	EI.SET bei G = TRUE	Ausgabe EO1.SET
G0_RESET	EI.RESET bei G = FALSE	Ausgabe EO0.RESET
G1_RESET	EI.RESET bei G = TRUE	Ausgabe EO1.RESET

Alle Zustände (außer START) kehren mit einer unbedingten Transition (1) zurück zu START.

Anwendungsszenarien

- **Steuerung zweier paralleler Prozesspfade:** Je nach Stellung eines Schalters werden SET/RESET-Signale auf unterschiedliche Anlagenteile verteilt.
- **Betriebsartenumschaltung:** Beispielsweise zwischen Automatik- und Handbetrieb – eine logische Größe G entscheidet, welche Ereignisse auf welche Logik wirken.
- **Testumgebungen:** Ein Testrahmen kann durch Umschalten von G die gleichen Ereignisse an unterschiedliche Testobjekte senden.

Vergleich mit ähnlichen Bausteinen

- **`SELECT`-Baustein:** Ein klassischer Multiplexer/Demultiplexer arbeitet meist mit Daten, nicht mit Ereignissen. Der **ASR_SWITCH** spezialisiert sich auf ASR-Ereignisweiterleitung.
- **`F_TRIG` / `R_TRIG` / `SR`:** Diese Bausteine setzen oder rücksetzen eine Speicherfunktion, leiten aber keine Ereignisse weiter. Der **ASR_SWITCH** erzeugt selbst keine Zustände, sondern kanalisiert nur vorhandene Ereignisse.
- **Adapterbasierte Lösungen:** In der 4diac-IDE ist die Verwendung von Adaptern typisch für modulare Kommunikation – der **ASR_SWITCH** ist ein Paradebeispiel für die flexible Kopplung von Funktionsbausteinen.

Fazit

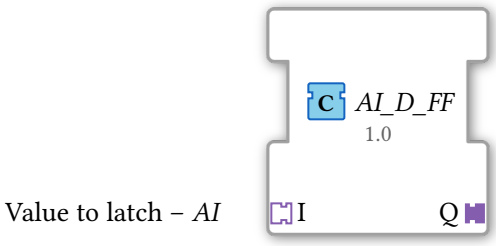
Der **ASR_SWITCH** ist ein einfacher, aber effektiver Funktionsbaustein zur demultiplexierenden Verteilung von SET/RESET-Ereignissen. Durch die Adapter-Schnittstelle lässt er sich nahtlos in ereignisbasierte Automatisierungslösungen integrieren. Seine klare Zustandslogik und minimale Schnittstelle machen ihn zu einem zuverlässigen Baustein für die Umschaltung zwischen zwei Signalpfaden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

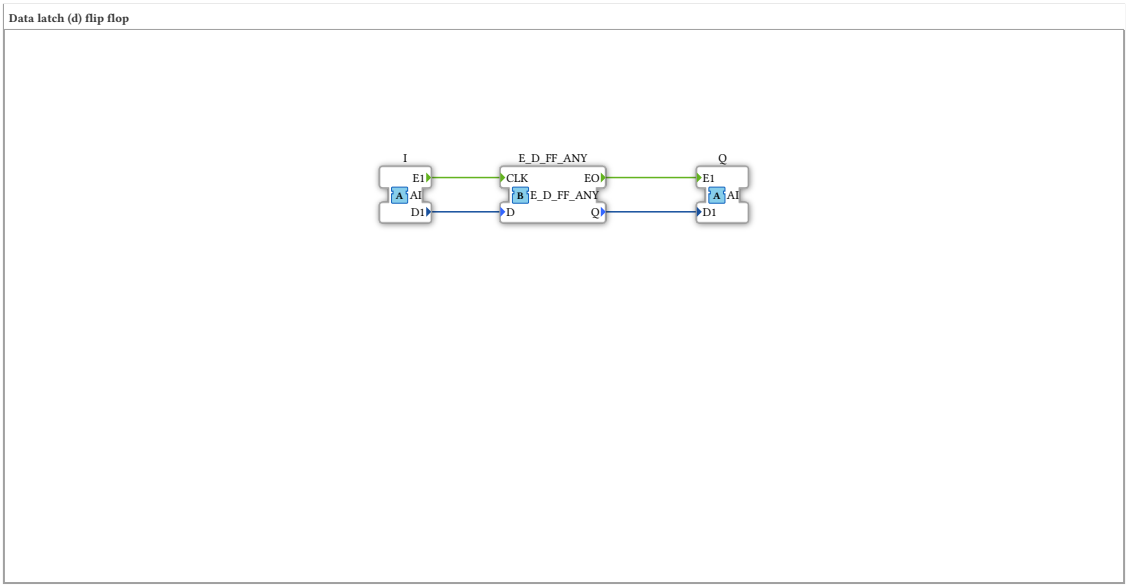
INT

AI_D_FF



Value to latch – *AI*

AI – Latched value



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AI_D_FF** realisiert ein data latch (D-Flipflop) zur temporären Speicherung eines Datenwerts. Er dient als Adapter-gekapselte Implementierung eines flankengesteuerten D-Flipflops nach IEC 61499, das bei einem Ereignis am Eingang den anliegenden Datenwert übernimmt und am Ausgang bereitstellt.

Schnittstellenstruktur

Ereignis-Eingänge
Keine direkten Ereignis-Eingänge.

Ereignis-Ausgänge
Keine direkten Ereignis-Ausgänge.

Daten-Eingänge
Keine direkten Daten-Eingänge.

Daten-Ausgänge
Keine direkten Daten-Ausgänge.

Adapter

Bezeichnung	Richtung	Typ	Beschreibung
I	Socket	`adapter::types::unidirectional::AI`	Liefert das Eingangsereignis (E1) und den zu latchenden Datenwert (D1).
Q	Plug	`adapter::types::unidirectional::AI`	Gibt nach der Übernahme das Ergebnisereignis (E1) und den gespeicherten Datenwert (D1) aus.

Der Adapter `AI` ist unidirektional, d.h. die Daten- und Ereignisflüsse verlaufen nur in eine Richtung. Socket **I** dient als Eingang, Plug **Q** als Ausgang.

Funktionsweise
Interne Komponente ist ein `E_D_FF_ANY`-Funktionsblock (IEC 61499 D-Flipflop). Die Verdrahtung verbindet:

- `I.E1` ⟶ `E_D_FF_ANY.CLK` (Taktflanke)
- `I.D1` ⟶ `E_D_FF_ANY.D` (Dateneingang)
- `E_D_FF_ANY.EO` ⟶ `Q.E1` (Ausgangsereignis)
- `E_D_FF_ANY.Q` ⟶ `Q.D1` (latchender Datenwert)

Ablauf:
Ein Ereignis an `I.E1` löst eine steigende Flanke am Takteingang des internen Flipflops aus. Der aktuelle Wert von `I.D1` wird dabei übernommen und steht nach Ende der Verarbeitung an `Q.D1` zur Verfügung. Gleichzeitig wird ein Ereignis an `Q.E1` ausgegeben, das die Gültigkeit des neuen Datenwerts signalisiert.

Technische Besonderheiten

- **Adapter-Kapselung:** Der FB besitzt keine direkten Ein-/Ausgänge, sondern ausschließlich Adapter-Schnittstellen. Dies erleichtert die Wiederverwendung in modularen Architekturen, bei denen mehrere Signale gebündelt transportiert werden.
- **Unidirektionaler Datenfluss:** Die Adapter sind als reine Quellen (Plug) oder Senken (Socket) ausgelegt, was die Signalrichtung eindeutig festlegt.
- **Flankengesteuert:** Die Übernahme erfolgt nur bei einem Ereignis (steigende Flanke), nicht bei Pegeländerungen. Somit eignet sich der Baustein für getaktete Systeme.

Zustandsübersicht
Der interne Zustand des Flipflops wird durch den gespeicherten Wert `Q` bestimmt. Es existieren zwei stabile Zustände:

Takt (CLK)	D (Eingang)	Q (Ausgang) vorher	Q (Ausgang) nachher
Kein Ereignis	-	Q_alt	Q_alt
Ereignis (steigende Flanke)	0	X	0
Ereignis (steigende Flanke)	1	X	1

Bei jedem Ereignis an `I.E1` wird der aktuelle D-Wert unabhängig vom vorherigen Q-Wert übernommen.

Anwendungsszenarien

- **Abtastung analoger Werte:** Ein Sensor (z.B. Temperatur, Druck) liefert über den Adapter `I` periodisch einen Messwert. Der FB speichert diesen bei einem Trigger-Ereignis und gibt ihn an die nachfolgende Logik weiter.
- **Zwischenspeicher in Datenpfaden:** In Fließband- oder Pipeline-Strukturen kann `AI_D_FF` als Datenhalte-Glied eingesetzt werden, um Werte für einen Taktzyklus zu puffern.
- **Entprellung / Flankenverarbeitung:** Kombiniert mit einem Taktgeber kann der Baustein verwendet werden, um eingehende Signale nur zu bestimmten Zeitpunkten zu übernehmen.

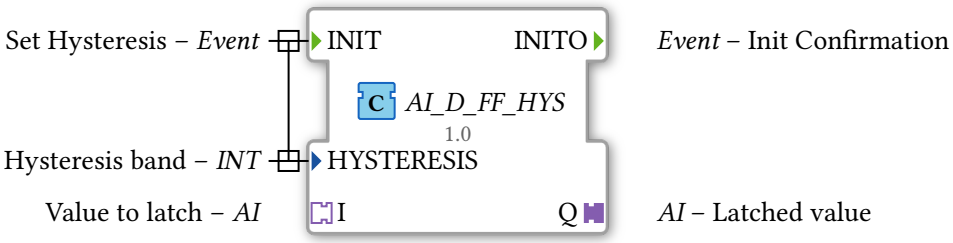
Vergleich mit ähnlichen Bausteinen

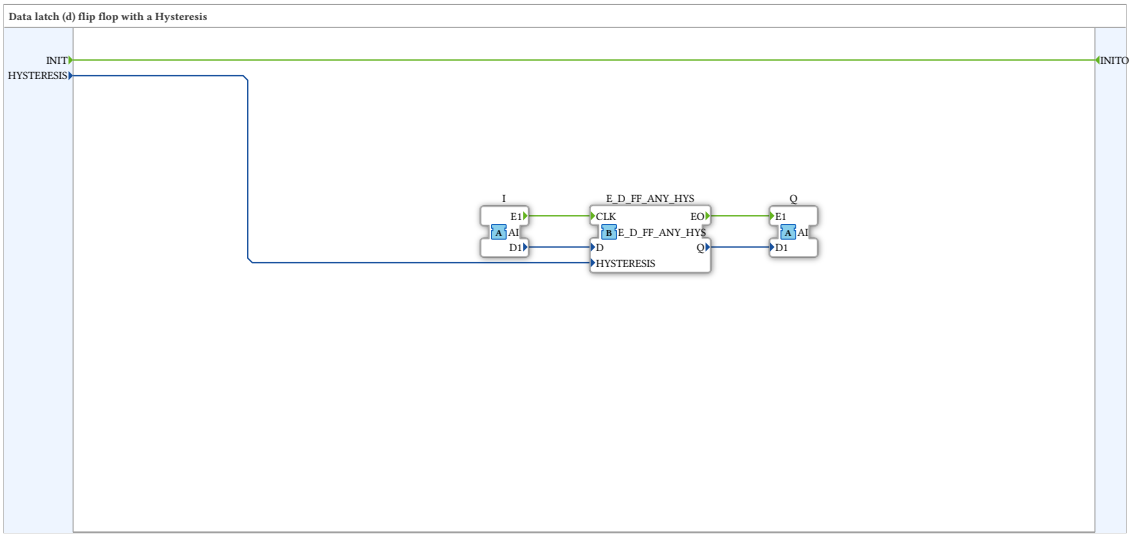
- **E_D_FF** (ohne Adapter): Bietet direkte Ereignis- und Dateneingänge/-ausgänge, ist aber weniger abstrahiert. `AI_D_FF` hingegen kapselt die Signale in Adaptern, was die Komposition in höheren Hierarchien vereinfacht.
- **SR-Flipflop (S/R dominant):** Speichert Zustand über Set/Reset, während `AI_D_FF` nur bei Taktflanken übernimmt und keine Reset-Funktion besitzt.
- **Latch (pegelgesteuert):** Anders als ein Latch reagiert `AI_D_FF` nur auf Flanken, wodurch es besser für synchronisierte Systeme geeignet ist.

Fazit

Der **AI_D_FF** ist ein kompakter, adapterbasierter D-Flipflop-Baustein, der einen Datenwert flankengesteuert speichert und ausgibt. Durch die Verwendung von unidirektionalen Adaptern fügt er sich nahtlos in moderne, modulare Automatisierungskonzepte ein und ermöglicht eine saubere Trennung von Daten- und Ereignissignalen. Seine Anwendung eignet sich vor allem für getaktete Abtast- und Zwischenspeicheraufgaben in IEC-61499-Systemen.

AI_D_FF_HYS





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AI_D_FF_HYS** realisiert ein taktgesteuertes D-Flipflop (Data Latch) mit einer konfigurierbaren Hysterese. Er dient dazu, ein binäres oder analoges Eingangssignal stabil zu verriegeln und Störungen im Signalverlauf auszufiltern. Die Hysterese verhindert ein ständiges Umschalten bei geringen Signaländerungen (Prellen, Rauschen) und sorgt so für einen sauberen, verzögerungsfreien Ausgangszustand.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Mit Variablen	Beschreibung
INIT	EInit	HYSTERESIS	Initialisierung und Vorgabe des Hysteresebandes

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung
INITO	EInit	Bestätigung der Initialisierung

Daten-Eingänge

Bezeichnung	Typ	Beschreibung
HYSTERESIS	INT	Hysterese-Bandbreite (z. B. in digitalen Schritten)

Daten-Ausgänge

Der Baustein besitzt keine direkten Daten-Ausgänge. Der verriegelte Wert wird über den **Adapter-Plug Q** zur Verfügung gestellt.

Adapter

Bezeichnung	Typ	Richtung	Beschreibung
I	AI	Socket	Eingangsadapter: Taktsignal (**E1**) und Daten (**D1**)
Q	AI	Plug	Ausgangsadapter: latched value (**D1**) und zugehöriges Ereignis (**E1**)

Der verwendete Adaptertyp `adapter::types::unidirectional::AI` ist ein unidirektionaler Daten- und Ereigniskanal. Über den Socket **I** werden die Werte des Takts (E1) und der Daten (D1) empfangen; über den Plug **Q** wird der latched Wert (D1) zusammen mit einem Bestätigungsereignis (E1) ausgegeben.

Funktionsweise

1. **Initialisierung**
   Der Baustein wird durch das Ereignis **INIT** aktiviert. Dabei wird der übergebene Wert **HYSTERESIS** als Hystereseband gespeichert. Nach erfolgreicher Initialisierung wird **INITO** ausgelöst.
2. **Datenverriegelung**
   Nach der Initialisierung wartet der Baustein auf ein Taktsignal am Adapter **I** (Ereignis **I.E1**). Bei jedem Taktereignis wird der aktuelle Datenwert **I.D1** eingelesen.
   Der interne Funktionsblock `E_D_FF_ANY_HYS` vergleicht den eingehenden Wert mit dem zuletzt verriegelten Wert unter Berücksichtigung der Hysterese:
   - Überschreitet der neue Wert den alten Wert um mehr als die Hälfte der Hysterese, wird der neue Wert übernommen (gesetzt).
   - Unterschreitet er den alten Wert um mehr als die Hälfte der Hysterese, wird der neue Wert übernommen (rückgesetzt).
   - Bei geringeren Differenzen bleibt der Ausgang unverändert.

   Der so ermittelte latched Wert wird am Adapter **Q** bereitgestellt: **Q.D1** enthält den digitalen Wert, und das Ereignis **Q.E1** signalisiert die Aktualisierung.

Technische Besonderheiten

- **Adapterbasierte Kommunikation**
  Der gesamte Daten- und Ereignisaustausch erfolgt über Adapter. Dies ermöglicht eine lose Kopplung zwischen dem Flipflop und den angeschlossenen Komponenten und erleichtert die Wiederverwendung in verschiedenen Kontexten.
- **Hysterese zur Störunterdrückung**
  Die Hysterese verhindert ein Flip-Flop-Verhalten bei Rauschen oder prellenden Signalen. Die Bandbreite wird als Ganzzahl (INT) übergeben und kann zur Laufzeit über **INIT** neu gesetzt werden.
- **Keine direkten Datenausgänge**
  Der latched Wert ist nur über den Adapter **Q** zugänglich. Dies erzwingt eine strukturierte Signalweiterleitung im Gesamtsystem.
- **Eingebetteter Funktionsblock**
  Das eigentliche Flipflop mit Hysterese wird durch den internen FB `E_D_FF_ANY_HYS` realisiert, was die Komplexität kapselt und die Wartung erleichtert.

Zustandsübersicht

Der FB kennt keine expliziten Zustände im Sinne einer Endlichen Automaten. Er befindet sich nach der **INIT**-Behandlung in einem Betriebszustand, in dem er auf Taktereignisse wartet. Ein erneuter **INIT**-Aufruf setzt die Hysterese neu und initialisiert die interne Logik.

Zustand	Beschreibung
Initialisiert	Nach erfolgreichem **INIT** bereit zur Verarbeitung
Warten auf Takt	Erwartet ein **I.E1**-Ereignis am Eingangsadapter
Datenverarbeitung	Bei **I.E1**: Hysterese-Vergleich und ggf. Übernahme

Anwendungsszenarien

- **Entprellung von Schaltern und Sensoren**
  Ein mechanischer Kontakt liefert prellende Signale. Die Hysterese unterdrückt kurzzeitige Änderungen und gibt nur stabile Zustände weiter.
- **Rauschunterdrückung bei analogen Signalen**
  Ein Analog-Digital-Wandler liefert verrauschte Werte. Die Hysterese verhindert ständiges Umschalten zwischen benachbarten Quantisierungsstufen.
- **Zustandsspeicher in Steuerungen**
  Der Baustein kann als einfaches D-Flipflop verwendet werden, um einen digitalen Wert taktgesteuert zu übernehmen und bis zum nächsten Takt zu halten.
- **Schnittstelle zwischen asynchronen und synchronen Systemen**
  Der Adapter erlaubt die Anbindung von Bausteinen mit unterschiedlichen Ereignistakten, indem das Flipflop als Puffer dient.

Vergleich mit ähnlichen Bausteinen

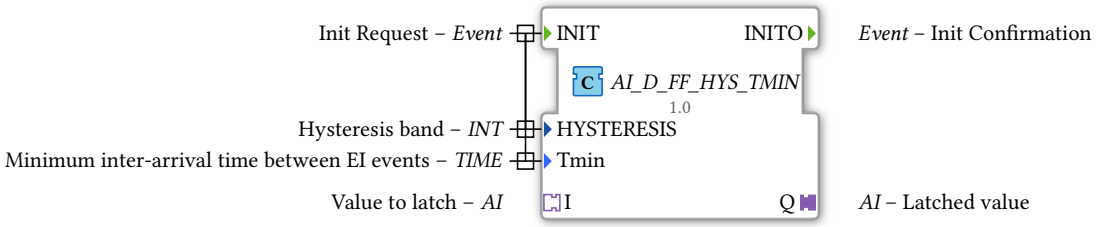
Baustein	Hysterese	Takteingang	Ausgabeart	Besonderheit
AI_D_FF_HYS	Ja	Ja	Adapter (AI)	Flexible Adapter-Kopplung
Standard D-FF (61499)	Nein	Ja	Direkte Datenausgänge	Keine Störunterdrückung
SR-Flipflop	Nein	Nein	Direkt	Set/Reset, kein Takt
Schmitt-Trigger	Ja	Nein	Analoge Schwelle	Nur Schwellwert, kein Speicher

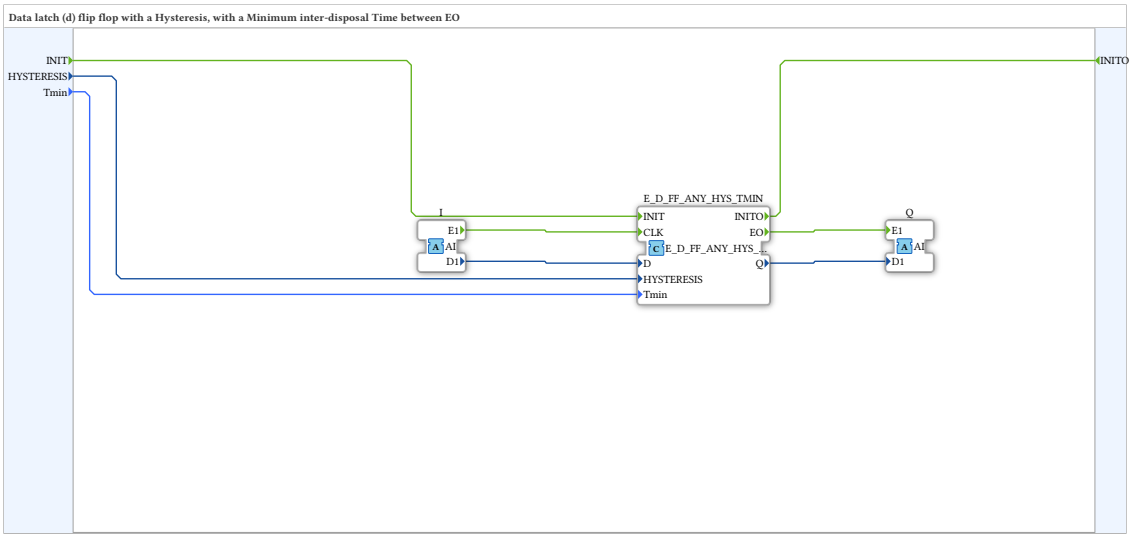
Der AI_D_FF_HYS kombiniert die Eigenschaften eines getakteten D-Flipflops mit einer einstellbaren Hysterese und bietet durch die Adapter-Schnittstelle eine plattformunabhängige Integration in IEC-61499-Systeme.

Fazit

Der **AI_D_FF_HYS** ist ein robuster und flexibler Funktionsblock zur taktgesteuerten Signalverriegelung mit Hysterese. Er eignet sich besonders für Anwendungen, bei denen Störungen aus dem Eingangssignal herausgefiltert werden müssen, ohne auf einen einfachen Takt-Flipflop verzichten zu können. Die adpaterbasierte Schnittstelle fördert modulare, wiederverwendbare Entwürfe und macht den Baustein zu einem wertvollen Element in der Automatisierungstechnik und Signalverarbeitung.

AI_D_FF_HYS_TMIN





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AI_D_FF_HYS_TMIN** realisiert ein Data-Latch (D-Flipflop) mit einem Hysteresband und einer Mindestzeit zwischen aufeinanderfolgenden Ereignisausgängen. Er dient als robuste Signalverarbeitungskomponente, die Störungen und schnelle Flankenwechsel unterdrückt und nur stabile Zustandsänderungen weitergibt.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar
INIT	EInit	Initialisierungsanforderung (setzt Hystereseband und Tmin)

Ereignis-Ausgänge

Ereignis	Typ	Kommentar
INITO	EInit	Initialisierungsbestätigung

Daten-Eingänge

Variable	Typ	Kommentar

HYSTERESIS	INT	Hystereseband (Wert in derselben Einheit wie das Eingangssignal)
Tmin	TIME	Mindestzeit zwischen zwei Ereignisausgängen (E1 am Q-Adapter)

Daten-Ausgänge

Der Baustein besitzt keine eigenen Datenausgänge. Der latched Wert wird über den Ausgangsadapter **Q** bereitgestellt.

Adapter

Adapter	Typ	Kommentar
I	adapter::types::unidirectional::AI	Eingangsadapter: liefert das zu verarbeitende Signal (über Ereignis E1 und Daten D1)
Q	adapter::types::unidirectional::AI	Ausgangsadapter: gibt den latched Wert aus (über Ereignis E1 und Daten D1)

Funktionsweise

Der Baustein verwendet intern das Unterprogramm **E_D_FF_ANY_HYS_TMIN** aus der Bibliothek *logiBUS::signalprocessing::hysteresis*. Die Funktionsweise im Überblick:

1. **Initialisierung**
   Über den Ereigniseingang **INIT** werden die Parameter `HYSTERESIS` und `Tmin` übernommen. Nach erfolgreicher Initialisierung wird **INITO** gesendet.
2. **Signalverarbeitung**
   - Ein neuer Wert wird über den Adapter **I** (Ereignis **E1** mit zugehörigem Datenwert **D1**) empfangen.
   - Der Baustein vergleicht den neuen Wert mit dem aktuell gespeicherten (latched) Wert unter Berücksichtigung des Hysteresebandes:
     - Eine Änderung wird nur dann übernommen, wenn die Differenz zwischen neuem Wert und altem Wert größer als `HYSTERESIS` ist.
   - Zusätzlich muss die seit dem letzten Ereignisausgang verstrichene Zeit mindestens `Tmin` betragen. Erst wenn beide Bedingungen erfüllt sind, wird der neue Wert übernommen und über den Adapter **Q** (Ereignis **E1** und Daten **D1**) ausgegeben.
3. **Ausgang**
   Der aktuell gültige latched Wert wird über den **Q**-Adapter bereitgestellt. Bei jeder gültigen Zustandsänderung wird ein Ereignis an **Q.E1** gesendet.

Technische Besonderheiten

- **Hysterese** – Unterdrückt Rauschen und schnelle Schwankungen, indem eine Änderung nur dann akzeptiert wird, wenn sie das Hystereseband überschreitet.
- **Mindestzeit zwischen Ereignissen (Tmin)** – Verhindert zu häufige Ausgaben (z. B. durch hochfrequente Signale oder Prellen). Ereignisse werden erst nach Ablauf von `Tmin` seit dem letzten Ereignis freigegeben.
- **Adapterbasierte Schnittstelle** – Die Ein- und Ausgabe erfolgt über Unidirectional-Adapters, was eine flexible Kopplung mit anderen Komponenten erlaubt.

Zustandsübersicht

Der Baustein besitzt keine expliziten benannten Zustände. Das interne Flipflop arbeitet als bistabiles Element mit zwei stabilen Zuständen (hoch/niedrig oder dem letzten gültigen Wert). Die Zustandsübergänge werden durch die Hysterese- und Zeitbedingungen kontrolliert.

Anwendungsszenarien

- **Entprellung von Sensorwerten** – Glättung analoger oder digitaler Eingangssignale (z. B. Temperatur, Druck, Füllstand) bei Schwellwertüberschreitungen.
- **Filterung von Rauschspitzen** – Vermeidung von Fehlauslösungen durch kurzzeitige Störungen.
- **Zeitgesteuerte Signalüberwachung** – Wenn Änderungen nur in bestimmten Zeitabständen gemeldet werden sollen (z. B. für Logging oder Aktualisierungsraten).

- **Schnittstellenentkopplung** – Verwendung als stabilisierte Zwischenstufe zwischen einem schnellen Signalgeber und einer trägen Aktorik.

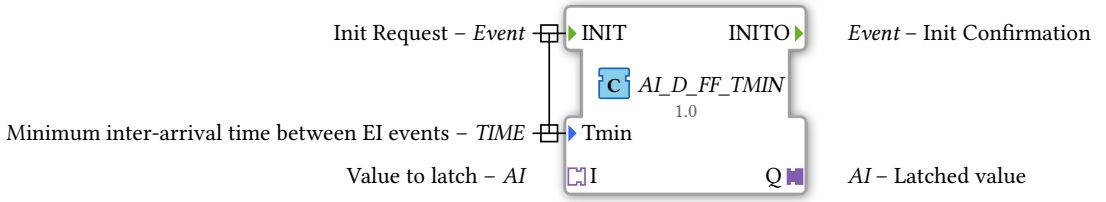
Vergleich mit ähnlichen Bausteinen

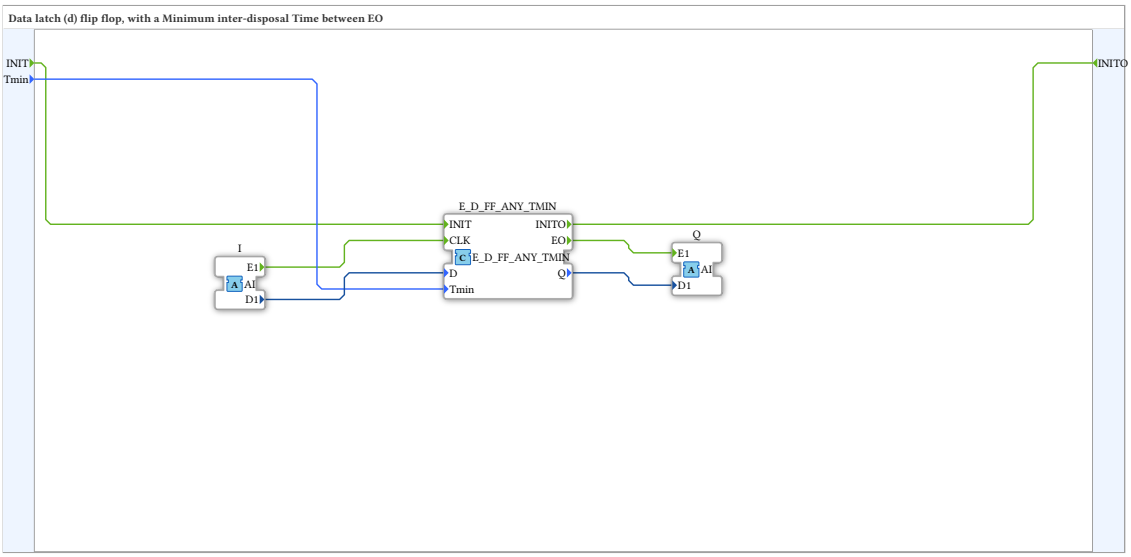
Baustein	Wesentlicher Unterschied
E_D_FF (einfaches D-Flipflop)	Kein Hystereseband, keine Mindestzeit – übernimmt jede Änderung sofort.
AI_D_FF_HYS (ohne Tmin)	Enthält Hysterese, jedoch keine zeitliche Begrenzung der Ausgabe.
AI_D_FF_TMIN (ohne Hysterese)	Enthält nur die Mindestzeit, aber keine Hysterese.
AI_D_FF_HYS_TMIN (dieser)	Kombiniert beide Filter: Hysterese und zeitliche Begrenzung.

Fazit

Der **AI_D_FF_HYS_TMIN** ist ein vielseitiger Funktionsblock zur robusten Signalaufbereitung. Seine Kombination aus Hysterese und Mindestintervall macht ihn ideal für Anwendungen, die sowohl störungssichere als auch zeitlich kontrollierte Zustandsänderungen erfordern. Die adapterbasierte Anbindung erleichtert die Integration in modulare Automatisierungssysteme.

AI_D_FF_TMIN





• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AI_D_FF_TMIN** realisiert ein D-Flip-Flop (Data Latch) mit einer minimalen zeitlichen Sperre zwischen aufeinanderfolgenden Ausgabeereignissen (inter-disposal time). Er dient als Adapter-gekapselte Komponente, die einen asynchron eingehenden Datenwert über einen Socket entgegennimmt, bei einem aktiven Ereignis zwischenspeichert und über einen Plug weitergibt. Die Ausgabe wird dabei erst dann wieder freigegeben, wenn die konfigurierte Mindestzeit (T_{\text{min}}) seit der letzten Ausgabe verstrichen ist.

Schnittstellenstruktur

Ereignis-Eingänge

Bezeichnung	Typ	Mit Variable	Kommentar
`INIT`	EInit	`Tmin`	Initialisierungsanforderung

Ereignis-Ausgänge

Bezeichnung	Typ	Mit Variable	Kommentar
`INITO`	EInit	–	Bestätigung der Initialisierung

Daten-Eingänge

Bezeichnung	Typ	Kommentar
`Tmin`	TIME	Minimale Zeitspanne zwischen zwei EO-Ereignissen

Daten-Ausgänge

Der Baustein besitzt keine eigenständigen Daten-Ausgänge; die Ausgabe erfolgt ausschließlich über den Adapter `Q`.

Adapter

Bezeichnung	Typ (Adapter)	Richtung	Kommentar
`I`	`adapter::types::unidirectional::AI`	Socket	Eingangsdaten zum Latchen
`Q`	`adapter::types::unidirectional::AI`	Plug	Gelatchter Ausgabewert

Funktionsweise

Der Baustein kapselt den internen Funktionsblock `E_D_FF_ANY_TMIN`, der das eigentliche D-Flip-Flop mit Zeitsteuerung implementiert. Die Verschaltung erfolgt wie folgt:

- Das Initialisierungsereignis `INIT` wird zusammen mit dem Parameter `Tmin` an den internen FB übergeben.
- Ein Daten-Latch wird durch ein Ereignis auf dem Socket `I` ausgelöst (Adapterereignis `I.E1`). Dieses Ereignis dient als Takt (`CLK`) für das Flip-Flop.
- Der aktuelle Datenwert von `I.D1` wird bei jedem Takt übernommen und am Ausgang des internen FBs bereitgestellt.
- Der interne FB erzeugt am Ausgang `EO` ein Ereignis, das über den Adapter `Q` als `Q.E1` weitergegeben wird.
- Zwischen zwei aufeinanderfolgenden `EO`-Ereignissen muss mindestens die Zeit `Tmin` liegen. Geht während dieser Sperrzeit ein weiterer Takt ein, wird der Datenwert zwar gelatcht, die Ausgabe jedoch erst nach Ablauf der Sperrzeit freigegeben.
- Das Ereignis `INITO` bestätigt den abgeschlossenen Initialisierungsvorgang.

Technische Besonderheiten

- **Adapter-Schnittstelle:** Die Ein- und Ausgabe erfolgen über unidirektionale Adapter vom Typ `AI`. Diese ermöglichen eine lose Kopplung und einfache Wiederverwendung des Bausteins in unterschiedlichen Umgebungen.
- **Minimale Ausgabezeit:** Der Parameter `Tmin` ermöglicht eine gezielte Begrenzung der Ereignisrate am Ausgang. Damit wird vermieden, dass nachgeschaltete Komponenten durch zu schnelle Ereignisfolgen überlastet werden.
- **Initialisierungspflicht:** Vor der ersten Nutzung muss der Baustein über `INIT` initialisiert werden, da während dieser Phase der interne Zustand gesetzt und `Tmin` übernommen wird.

Zustandsübersicht

Der Baustein besitzt keinen explizit sichtbaren Zustandsautomaten; das Zeitverhalten wird vollständig durch den internen FB `E_D_FF_ANY_TMIN` gesteuert. Im Wesentlichen durchläuft der Baustein folgende Phasen:

1. **Initialisierung** – Nach Erhalt von `INIT` wird der interne FB konfiguriert.
2. **Warten auf Takt** – Der Baustein wartet auf ein Ereignis am Socket `I`.
3. **Latchen** – Bei Takt wird der Datenwert übernommen und ein Ausgabeereignis erzeugt.
4. **Sperrphase** – Nach der Ausgabe wird für die Dauer von `Tmin` jedes weitere Ausgabeereignis verzögert.
5. **Bereit** – Nach Ablauf der Sperrzeit kann erneut ein Takt verarbeitet werden.

Anwendungsszenarien

- **Sensor-Datenübernahme:** Ein asynchron arbeitender Sensor liefert Messwerte, die nur in einem definierten Mindestabstand an ein Steuerungssystem weitergegeben werden sollen (z. B. Temperaturerfassung mit Entprellung).
- **Pufferung zeitkritischer Signale:** In Automatisierungsanwendungen, bei denen nachgeschaltete Verarbeitungsbausteine eine begrenzte Ereignisrate tolerieren, wird der Baustein als „Rate Limiter“ eingesetzt.
- **Synchronisation:** Der FB kann verwendet werden, um zyklische Abläufe zu synchronisieren, indem er die maximale Aktualisierungsrate eines Ausgangssignals begrenzt.

Vergleich mit ähnlichen Bausteinen

Baustein	Beschreibung	Unterschied zu AI_D_FF_TMIN
`E_D_FF`	Einfaches D-Flip-Flop ohne Zeitbegrenzung	Keine Mindestzeit zwischen Ausgabeereignissen
`E_D_FF_ANY`	D-Flip-Flop für beliebige Datentypen	Ebenfalls ohne zeitliche Steuerung
`AI_D_FF`	D-Flip-Flop mit Adapter-Schnittstelle	Fehlende `Tmin`-Option
`E_CYCLE`	Takterzeuger mit einstellbarer Periode	Erzeugt periodische Ereignisse, kein Latch

Der `AI_D_FF_TMIN` vereint die Funktionalität eines Adapter-gekapselten Flip-Flops mit einer frei konfigurierbaren Ausgabesperre – eine Kombination, die in Standard-Flip-Flops nicht enthalten ist.

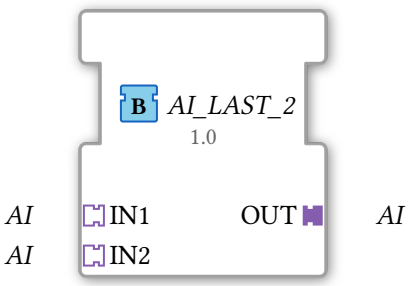
Fazit

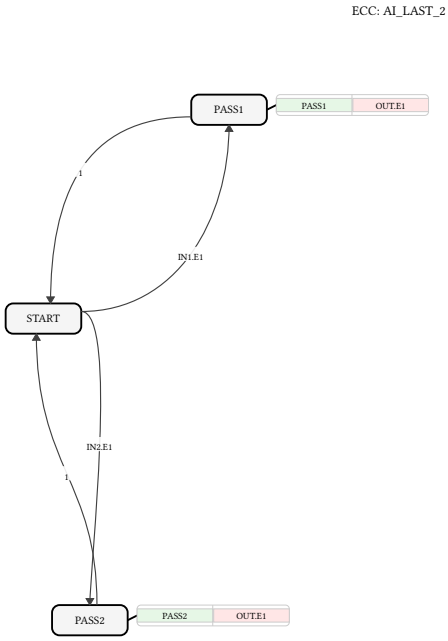
`AI_D_FF_TMIN` bietet eine kompakte Lösung für das Zwischenspeichern und zeitlich kontrollierte Weitergeben von Daten über Adapterschnittstellen. Die integrierte Mindest-Ausgabezeit schützt nachgelagerte Bausteine vor Überlastung und erlaubt eine robuste Signalverarbeitung in industriellen Steuerungssystemen. Durch die modulare Adapter-Schnittstelle ist der Baustein flexibel in verschiedenste 4diac-Netzwerke integrierbar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AI_LAST_2





Einleitung

Der AI_LAST_2 Funktionsblock führt zwei gleichartige INT-Adapter-Signale (**IN1**, **IN2**) auf einen gemeinsamen Ausgang (**OUT**) zusammen, nach dem Prinzip **Last Writer Wins**: welcher der beiden Sockets zuletzt ein Ereignis geschrieben hat, gewinnt, und sein Datenwert wird sofort und unverändert an OUT durchgereicht. Es handelt sich ausdrücklich nicht um ein naives Zusammenführen (z. B. Mittelwert oder ODER-Verknüpfung), sondern um eine reine Arbitrierung nach zeitlicher Reihenfolge.

Schnittstellenstruktur

Adapter

Eingangsadapter (Sockets):

- **IN1**: INT-Adapter (unidirectional) - erste Signalquelle
- **IN2**: INT-Adapter (unidirectional) - zweite Signalquelle

Ausgangsadapter (Plug):

- **OUT**: INT-Adapter (unidirectional) - zusammengeführtes Signal

Funktionsweise

Der Baustein ist als einfacher Basic-FB mit einer 3-Zustands-ECC umgesetzt (START, PASS1, PASS2). Trifft an IN1 ein Ereignis ein, wechselt die ECC nach PASS1: der aktuelle Datenwert von IN1 wird nach OUT kopiert und das Adapter-Ereignis an OUT gesendet. Trifft an IN2 ein Ereignis ein, geschieht dasselbe symmetrisch über PASS2. Nach jeder Durchreichung kehrt die ECC sofort wieder nach START zurück und ist bereit für das nächste Ereignis von IN1 oder IN2.

Technische Besonderheiten

- Last-Writer-Wins-Semantik statt naiver Datenverknüpfung
- Einfacher Basic-FB (ECC mit 3 Zuständen), kein zusätzlicher Zwischenspeicher nötig
- Keine Signalverzögerung: der Wert wird im selben Zyklus durchgereicht, in dem das auslösende Ereignis eintrifft
- Beide Eingänge sind gleichberechtigt - es gibt keine feste Priorität außer der zeitlichen Reihenfolge

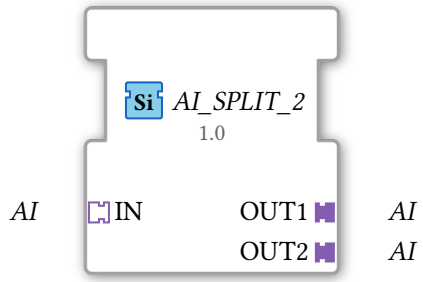
Anwendungsszenarien

- Zusammenführen zweier gleichwertiger INT-Signalquellen zu einem einzigen nachgeschalteten Adapterpfad (z. B. zwei redundante Sensoren, oder ein manueller und ein automatischer Sollwertgeber)
- Arbitrierung zwischen zwei Bedienstellen, die denselben INT-Wert schreiben dürfen
- Vereinfachung von Netzwerken, die sonst ein `F_SEL`/`E_RS`-Konstrukt inline verdrahten müssten

⚖️ Vergleich mit ähnlichen Bausteinen

AI_LAST_2 ist die Umkehrung von `AI_SPLIT_2`: AI_SPLIT_2 verteilt 1 Eingang auf 2 Ausgänge, AI_LAST_2 führt 2 Eingänge auf 1 Ausgang zusammen. Für Mischbetrieb zwischen einem Daten- und einem reinen Ereignisadapter (z. B. AX + AE) ist stattdessen `AX_AE_MERGE` gedacht, das nur das Ereignis mischt, ohne den Datenwert zu verändern.

AI_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AI_SPLIT_2 dient dazu, ein analoges Eingangssignal (AI) auf zwei identische analoge Ausgänge zu verteilen. Er ist als generischer Baustein konzipiert und ermöglicht eine flexible Weiterverarbeitung des Signals in zwei unabhängigen Pfaden.

Schnittstellenstruktur

Der Baustein besitzt keine Ereignis- oder Dateneingänge/-ausgänge im klassischen Sinne, sondern kommuniziert ausschließlich über Adapter-Schnittstellen vom Typ `adapter::types::unidirectional::AI`.

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::AI	Eingang (Socket)	Nimmt ein analoges Eingangssignal auf.
OUT1	adapter::types::unidirectional::AI	Ausgang (Plug)	Gibt das an IN anliegende Signal unverändert weiter.
OUT2	adapter::types::unidirectional::AI	Ausgang (Plug)	Gibt das an IN anliegende Signal unverändert weiter.

Funktionsweise

Der Baustein leitet das über den Socket `IN` empfangene analoge Signal (AI) gleichzeitig an beide Ausgänge (`OUT1` und `OUT2`) weiter. Es erfolgt keine Signalverarbeitung oder -veränderung; die Aufteilung ist rein passiv und sofort wirksam. Die Adapter stellen die notwendige Schnittstellenkompatibilität sicher.

Technische Besonderheiten

- **Generischer Aufbau**: Der Baustein ist als generischer FB (`GEN_AI_SPLIT`) implementiert und kann mit verschiedenen AI-Adaptern verwendet werden, die dem unidirektionalen AI-Protokoll entsprechen.
- **Keine Zustandslogik**: Es existiert kein ECC (Execution Control Chart), daher ist das Verhalten rein kombinatorisch.
- **Plattformunabhängigkeit**: Der Baustein ist gemäß IEC 61499-2 spezifiziert und in Umgebungen einsetzbar, die das Adapterkonzept unterstützen.

Zustandsübersicht

Der Baustein besitzt kein internes Zustandsverhalten. Die Ausgänge folgen direkt und ohne Verzögerung dem Eingangssignal.

Anwendungsszenarien

- **Signalverteilung**: Ein einzelner analoger Sensor (z. B. Drucksensor, Temperaturfühler) soll von zwei unabhängigen Steuerungskomponenten ausgewertet werden.
- **Redundanz**: Aufteilung eines Signals für parallele Überwachungs- und Regelungspfade.
- **Simulation**: Erzeugung eines zweiten identischen Signalkanals für Test- oder Analysezwecke.

Vergleich mit ähnlichen Bausteinen

- **AI_SPLIT_3**: Erweiterung auf drei Ausgänge.
- **AI_SELECT**: Wählt einen von mehreren Eingängen aus, anstatt zu verteilen.
- **AI_MERGE**: Fasst mehrere AI-Signale zu einem zusammen (z. B. Mittelwert). AI_SPLIT_2 ist spezialisiert auf die einfache 1-zu-2-Verteilung ohne Logik oder Konfiguration.
- `AI_SPLIT_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

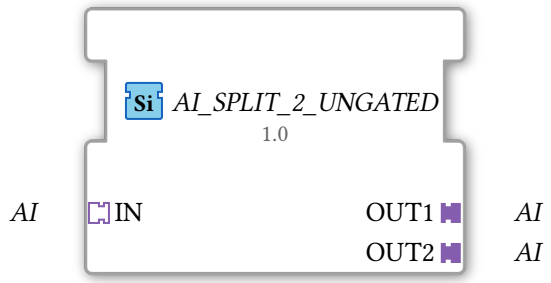
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

AI_SPLIT_2 ist ein einfacher, aber nützlicher Baustein zur passiven Aufteilung eines analogen Eingangssignals. Durch den generischen Adapteransatz ist er flexibel einsetzbar und erleichtert die modulare Strukturierung von Applikationen nach IEC 61499.

AI_SPLIT_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AI_SPLIT_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AI_SPLIT_2_UNGATED dient dazu, ein analoges Eingangssignal (AI) auf zwei identische analoge Ausgänge zu verteilen. Er ist als generischer Baustein konzipiert und ermöglicht eine flexible Weiterverarbeitung des Signals in zwei unabhängigen Pfaden.

Schnittstellenstruktur

Der Baustein besitzt keine Ereignis- oder Dateneingänge/-ausgänge im klassischen Sinne, sondern kommuniziert ausschließlich über Adapter-Schnittstellen vom Typ `adapter::types::unidirectional::AI`.

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::AI	Eingang (Socket)	Nimmt ein analoges Eingangssignal auf.
OUT1	adapter::types::unidirectional::AI	Ausgang (Plug)	Gibt das an IN anliegende Signal unverändert weiter.
OUT2	adapter::types::unidirectional::AI	Ausgang (Plug)	Gibt das an IN anliegende Signal unverändert weiter.

Funktionsweise

Der Baustein leitet das über den Socket `IN` empfangene analoge Signal (AI) gleichzeitig an beide Ausgänge (`OUT1` und `OUT2`) weiter. Es erfolgt keine Signalverarbeitung oder -veränderung; die Aufteilung ist rein passiv und sofort wirksam. Die Adapter stellen die notwendige Schnittstellenkompatibilität sicher.

Technische Besonderheiten

- **Generischer Aufbau**: Der Baustein ist als generischer FB (`GEN_AI_SPLIT`) implementiert und kann mit verschiedenen AI-Adaptern verwendet werden, die dem unidirektionalen AI-Protokoll entsprechen.
- **Keine Zustandslogik**: Es existiert kein ECC (Execution Control Chart), daher ist das Verhalten rein kombinatorisch.
- **Plattformunabhängigkeit**: Der Baustein ist gemäß IEC 61499-2 spezifiziert und in Umgebungen einsetzbar, die das Adapterkonzept unterstützen.

Zustandsübersicht

Der Baustein besitzt kein internes Zustandsverhalten. Die Ausgänge folgen direkt und ohne Verzögerung dem Eingangssignal.

Anwendungsszenarien

- **Signalverteilung**: Ein einzelner analoger Sensor (z. B. Drucksensor, Temperaturfühler) soll von zwei unabhängigen Steuerungskomponenten ausgewertet werden.
- **Redundanz**: Aufteilung eines Signals für parallele Überwachungs- und Regelungspfade.
- **Simulation**: Erzeugung eines zweiten identischen Signalkanals für Test- oder Analysezwecke.

Vergleich mit ähnlichen Bausteinen

- **AI_SPLIT_3**: Erweiterung auf drei Ausgänge.
- **AI_SELECT**: Wählt einen von mehreren Eingängen aus, anstatt zu verteilen.
- **AI_MERGE**: Fasst mehrere AI-Signale zu einem zusammen (z. B. Mittelwert). AI_SPLIT_2_UNGATED ist spezialisiert auf die einfache 1-zu-2-Verteilung ohne Logik oder Konfiguration.
- `AI_SPLIT_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

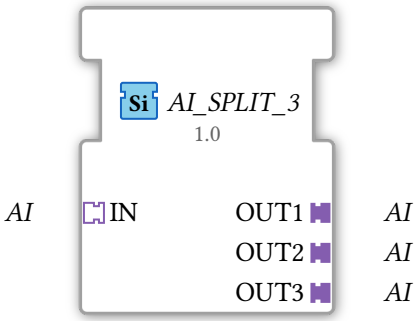
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

AI_SPLIT_2_UNGATED ist ein einfacher, aber nützlicher Baustein zur passiven Aufteilung eines analogen Eingangssignals. Durch den generischen Adapteransatz ist er flexibel einsetzbar und erleichtert die modulare Strukturierung von Applikationen nach IEC 61499.

AI_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AI_SPLIT_3** ist ein generischer Baustein zur Vervielfältigung eines analogen Eingangssignals (AI). Er teilt ein eingehendes AI-Signal auf drei separate, identische Ausgangssignale auf. Der Baustein ist vollständig über Adapter vom Typ `AI` definiert und arbeitet rein datenflussorientiert ohne eigene Ereignis- oder Datenpunkt-Schnittstellen.

Schnittstellenstruktur

Der Baustein bietet keine direkten Ereignis- oder Daten-Ein-/Ausgänge. Die gesamte Kommunikation erfolgt über **Adapter**.

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine direkten Daten-Eingänge – das AI-Signal wird über den Socket-Adapter `IN` aufgenommen.

Daten-Ausgänge

Keine direkten Daten-Ausgänge – die drei identischen AI-Signale werden über die Plug-Adapter `OUT1`, `OUT2` und `OUT3` bereitgestellt.

Adapter

Richtung	Bezeichnung	Typ	Beschreibung
Socket	`IN`	`ADAPTER_AI`	Eingehendes analoges Signal (Quelle)
Plug	`OUT1`	`ADAPTER_AI`	Erster Ausgang – identisch zum Eingangssignal
Plug	`OUT2`	`ADAPTER_AI`	Zweiter Ausgang – identisch zum Eingangssignal
Plug	`OUT3`	`ADAPTER_AI`	Dritter Ausgang – identisch zum Eingangssignal

Die Adapter sind vom Typ `adapter::types::unidirectional::AI`. Der Baustein ist generisch, d. h. die konkrete Datenstruktur des AI-Signals (z. B. Wert, Einheit, Status) wird erst bei der Instanziierung durch die angebundenen Adapter festgelegt.

Funktionsweise

Der Funktionsblock leitet das über den Socket `IN` empfangene analoge Signal unverändert und gleichzeitig an alle drei Plug-Adapter `OUT1`, `OUT2` und `OUT3` weiter. Es findet keine Signalumformung, Verstärkung oder Verzögerung statt. Der Baustein fungiert als reiner **1-zu-3-Splitter**.

Da keine Ereignisse oder Zustandsautomaten vorhanden sind, erfolgt die Weitergabe rein über den Datenfluss. Sobald ein neues AI-Signal am `IN` anliegt, steht es an allen drei Ausgängen zur Verfügung.

Technische Besonderheiten

- **Generischer Baustein:** Der FB ist mithilfe von `eclipse4diac::core::GenericClassName` als `'GEN_AI_SPLIT'` gekennzeichnet. Dadurch kann er in verschiedenen Anwendungen für unterschiedliche AI-Datentypen verwendet werden, ohne dass der Funktionsblock selbst angepasst werden muss.
- **Reine Adapter-Schnittstelle:** Es werden keine direkten Ereignis- oder Daten-Ein-/Ausgänge benötigt. Die gesamte Datenübertragung erfolgt über die Adapter, was eine flexible Kopplung mit anderen Bausteinen ermöglicht.
- **Keine Zustandslogik:** Der Baustein besitzt keinen eigenen Zustandsautomaten (ECC) – die Funktionalität ist rein datenflussbasiert und daher sehr leichtgewichtig.

Zustandsübersicht

Der Baustein verfügt über **keine** Zustandsmaschine oder internen Zustände. Die Signale werden direkt und ohne Verzögerung weitergeleitet. Eine Zustandsübersicht entfällt daher.

Anwendungsszenarien

- **Signalverteilung in Automatisierungssystemen:** Ein analoger Messwert (z. B. Drucksensor, Temperatur) soll an mehrere Steuerungs- oder Überwachungseinheiten übergeben werden.
- **Redundanz und Parallelverarbeitung:** Das gleiche AI-Signal wird für unterschiedliche Algorithmen parallel benötigt (z. B. Regelung, Alarmierung, Protokollierung).
- **Testumgebungen:** Ein simulierte Analogwert kann gleichzeitig an mehrere zu testende Bausteine verteilt werden.

Vergleich mit ähnlichen Bausteinen

- **AI_SPLIT_2:** Verteilt das Eingangssignal auf zwei statt drei Ausgänge.
- **AI_SPLIT_4 / AI_SPLIT_N:** Erweiterte Versionen mit vier oder mehr Ausgängen (falls vorhanden).
- **AI_SELECT:** Wählt aus mehreren Eingängen einen aus, anstatt ein Signal zu vervielfältigen.
- **AI_MERGE:** Fasst mehrere AI-Eingänge zu einem Ausgang zusammen (Gegenstück zum Splitter).

Gemeinsam ist allen Split-Bausteinen, dass sie das Eingangssignal ohne Veränderung an alle Ausgänge weiterleiten. Der AI_SPLIT_3 stellt eine typische 3-fach-Ausführung dar.

- `AI_SPLIT_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

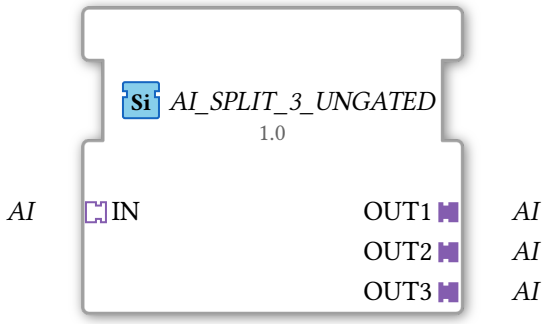
Der **AI_SPLIT_3** ist ein einfacher, aber effektiver Baustein zur Vervielfältigung eines analogen Signals auf drei Ausgänge. Durch seine generische Natur und die Verwendung von Adaptern fügt er sich nahtlos in Adapter-basierte 4diac-Applikationen ein. Er eignet sich ideal für Anwendungen, in denen ein AI-Wert an mehrere Verbraucher verteilt werden muss, ohne zusätzliche Logik oder Konfiguration.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AI_SPLIT_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AI_SPLIT_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AI_SPLIT_3_UNGATED** ist ein generischer Baustein zur Vervielfältigung eines analogen Eingangssignals (AI). Er teilt ein eingehendes AI-Signal auf drei separate, identische Ausgangssignale auf. Der Baustein ist vollständig über Adapter vom Typ `AI` definiert und arbeitet rein datenflussorientiert ohne eigene Ereignis- oder Datenpunkt-Schnittstellen.

Schnittstellenstruktur

Der Baustein bietet keine direkten Ereignis- oder Daten-Ein-/Ausgänge. Die gesamte Kommunikation erfolgt über **Adapter**.

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine direkten Daten-Eingänge – das AI-Signal wird über den Socket-Adapter `IN` aufgenommen.

Daten-Ausgänge

Keine direkten Daten-Ausgänge – die drei identischen AI-Signale werden über die Plug-Adapter `OUT1`, `OUT2` und `OUT3` bereitgestellt.

Adapter

Richtung	Bezeichnung	Typ	Beschreibung
Socket	`IN`	`ADAPTER_AI`	Eingehendes analoges Signal (Quelle)
Plug	`OUT1`	`ADAPTER_AI`	Erster Ausgang – identisch zum Eingangssignal
Plug	`OUT2`	`ADAPTER_AI`	Zweiter Ausgang – identisch zum Eingangssignal
Plug	`OUT3`	`ADAPTER_AI`	Dritter Ausgang – identisch zum Eingangssignal

Die Adapter sind vom Typ `adapter::types::unidirectional::AI`. Der Baustein ist generisch, d. h. die konkrete Datenstruktur des AI-Signals (z. B. Wert, Einheit, Status) wird erst bei der Instanziierung durch die angebundenen Adapter festgelegt.

Funktionsweise

Der Funktionsblock leitet das über den Socket `IN` empfangene analoge Signal unverändert und gleichzeitig an alle drei Plug-Adapter `OUT1`, `OUT2` und `OUT3` weiter. Es findet keine Signalumformung, Verstärkung oder Verzögerung statt. Der Baustein fungiert als reiner **1-zu-3-Splitter**.

Da keine Ereignisse oder Zustandsautomaten vorhanden sind, erfolgt die Weitergabe rein über den Datenfluss. Sobald ein neues AI-Signal am `IN` anliegt, steht es an allen drei Ausgängen zur Verfügung.

Technische Besonderheiten

- **Generischer Baustein:** Der FB ist mithilfe von `eclipse4diac::core::GenericClassName` als `'GEN_AI_SPLIT'` gekennzeichnet. Dadurch kann er in verschiedenen Anwendungen für unterschiedliche AI-Datentypen verwendet werden, ohne dass der Funktionsblock selbst angepasst werden muss.
- **Reine Adapter-Schnittstelle:** Es werden keine direkten Ereignis- oder Daten-Ein-/Ausgänge benötigt. Die gesamte Datenübertragung erfolgt über die Adapter, was eine flexible Kopplung mit anderen Bausteinen ermöglicht.
- **Keine Zustandslogik:** Der Baustein besitzt keinen eigenen Zustandsautomaten (ECC) – die Funktionalität ist rein datenflussbasiert und daher sehr leichtgewichtig.

Zustandsübersicht

Der Baustein verfügt über **keine** Zustandsmaschine oder internen Zustände. Die Signale werden direkt und ohne Verzögerung weitergeleitet. Eine Zustandsübersicht entfällt daher.

Anwendungsszenarien

- **Signalverteilung in Automatisierungssystemen:** Ein analoger Messwert (z. B. Drucksensor, Temperatur) soll an mehrere Steuerungs- oder Überwachungseinheiten übergeben werden.
- **Redundanz und Parallelverarbeitung:** Das gleiche AI-Signal wird für unterschiedliche Algorithmen parallel benötigt (z. B. Regelung, Alarmierung, Protokollierung).
- **Testumgebungen:** Ein simulierte Analogwert kann gleichzeitig an mehrere zu testende Bausteine verteilt werden.

Vergleich mit ähnlichen Bausteinen

- **AI_SPLIT_2:** Verteilt das Eingangssignal auf zwei statt drei Ausgänge.
- **AI_SPLIT_4 / AI_SPLIT_N:** Erweiterte Versionen mit vier oder mehr Ausgängen (falls vorhanden).
- **AI_SELECT:** Wählt aus mehreren Eingängen einen aus, anstatt ein Signal zu vervielfältigen.
- **AI_MERGE:** Fasst mehrere AI-Eingänge zu einem Ausgang zusammen (Gegenstück zum Splitter).

Gemeinsam ist allen Split-Bausteinen, dass sie das Eingangssignal ohne Veränderung an alle Ausgänge weiterleiten. Der AI_SPLIT_3_UNGATED stellt eine typische 3-fach-Ausführung dar.

- `AI_SPLIT_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

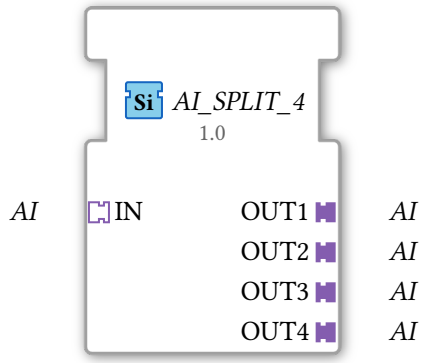
Fazit

Der **AI_SPLIT_3_UNGATED** ist ein einfacher, aber effektiver Baustein zur Vervielfältigung eines analogen Signals auf drei Ausgänge. Durch seine generische Natur und die Verwendung von Adaptern fügt er sich nahtlos in Adapter-basierte 4diac-Applikationen ein. Er eignet sich ideal für Anwendungen, in denen ein AI-Wert an mehrere Verbraucher verteilt werden muss, ohne zusätzliche Logik oder Konfiguration.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AI_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AI_SPLIT_4** ist ein generischer Baustein, der einen unidirektionalen Analog-Eingang (AI) auf vier separate Analog-Ausgänge aufteilt. Er dient als Signalverteiler für analoge Signale und erlaubt, ein einzelnes AI-Signal an verschiedene nachfolgende Bausteine weiterzugeben. Der Baustein ist vollständig adapterbasiert und besitzt keine eigenen Ereignis- oder Datenschnittstellen. Er ist unter der Eclipse Public License 2.0 verfügbar.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine

Ereignis-Ausgänge

- Keine

Daten-Eingänge

- Keine

Daten-Ausgänge

- Keine

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	`IN`	`adapter::types::unidirectional::AI`	Eingang des zu verteilenden analogen Signals

Plug (Ausgang)	OUT1	adapter::types::unidirectional::AI	Erster Ausgang (Kopie des Eingangssignals)
Plug (Ausgang)	OUT2	adapter::types::unidirectional::AI	Zweiter Ausgang (Kopie des Eingangssignals)
Plug (Ausgang)	OUT3	adapter::types::unidirectional::AI	Dritter Ausgang (Kopie des Eingangssignals)
Plug (Ausgang)	OUT4	adapter::types::unidirectional::AI	Vierter Ausgang (Kopie des Eingangssignals)

Funktionsweise

Der Baustein stellt eine einfache 1:4-Verteilung eines analogen Signals dar. Das am Adapter `IN` anliegende Signal wird unverändert auf alle vier Ausgangs-Adapter (`OUT1` … `OUT4`) durchgereicht. Es findet keinerlei Skalierung, Filterung oder Umwandlung statt – das Signal wird auf feldbussebene 1:1 kopiert. Die Anzahl der Kanäle ist auf vier festgelegt.

Technische Besonderheiten

- **Reiner Adapter-Baustein**: Der FB enthält keine Ereignis- oder Datenschnittstellen; die gesamte Kommunikation erfolgt über Adapter.
- **Generischer Typ**: Über das Attribut `GenericClassName` (`'GEN_AI_SPLIT'`) wird der Baustein als generisch ausgewiesen. In der 4diac-IDE kann dadurch ein konkreter Typ abgeleitet werden.
- **Unidirektional**: Der verwendete Adaptertyp `unidirectional::AI` überträgt Daten nur in eine Richtung (vom Socket zu den Plugs).
- **Kein Zustandsautomat**: Da keine Ereignisse vorhanden sind, existiert kein ECC (Execution Control Chart). Der FB ist rein datenflussorientiert.

Zustandsübersicht

Der Baustein verfügt über keinen Zustandsautomaten – er ist ereignisfrei und gibt das Eingangssignal kontinuierlich an alle Ausgänge weiter. Eine Zustandsübersicht entfällt daher.

Anwendungsszenarien

- **Analog-Signalverteilung**: Ein einzelner Sensorwert (z. B. Temperatur, Druck, Füllstand) soll an mehrere Steuerungslogiken oder Visualisierungskomponenten gleichzeitig gesendet werden.
- **Redundanz oder Monitoring**: Das Signal wird parallel an einen Hauptregler und einen unabhängigen Überwachungsbaustein weitergeleitet.
- **Entkopplung von Subsystemen**: In einer modularen Steuerung können verschiedene Subsysteme denselben Analogwert empfangen, ohne dass ein Mehrfachzugriff auf den Sensor erforderlich ist.

Vergleich mit ähnlichen Bausteinen

- **AI_SPLIT_2 / AI_SPLIT_8**: Ähnliche Bausteine mit 2 bzw. 8 Ausgängen. Die Wahl hängt von der benötigten Anzahl paralleler Signale ab.
- **AI_SELECT** oder **AI_MUX**: Diese Bausteine wählen aus mehreren Eingängen einen aus, während der AI_SPLIT das Eingangssignal vervielfacht.
- **Standard-Adaptersplit**: Manche Bibliotheken bieten generische Split-Bausteine, die jedoch oft zusätzliche Ereignis- oder Datenschnittstellen benötigen. Der AI_SPLIT_4 ist durch seinen reinen Adapteransatz besonders einfach in Adapterketten integrierbar.
- `AI_SPLIT_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

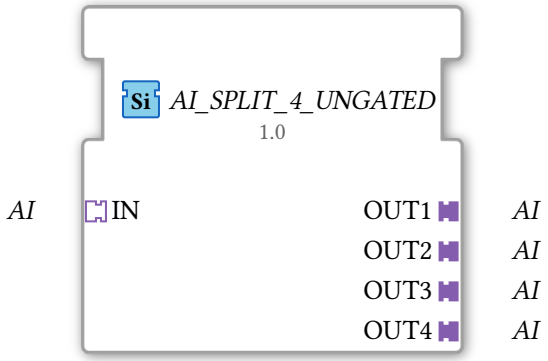
Der Funktionsblock **AI_SPLIT_4** ist ein spezialisierter, aber einfach einsetzbarer Baustein zur Vervielfachung analoger Signale. Dank seiner reinen Adapterschnittstelle lässt er sich nahtlos in 4diac-Projekte einbinden, ohne zusätzliche Ereignislogik. Er eignet sich besonders für modulare und skalierbare Automatisierungslösungen, bei denen ein einzelner Messwert an mehrere Verbraucher verteilt werden muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AI_SPLIT_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AI_SPLIT_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AI_SPLIT_4_UNGATED** ist ein generischer Baustein, der einen unidirektionalen Analog-Eingang (AI) auf vier separate Analog-Ausgänge aufteilt. Er dient als Signalverteiler für analoge Signale und erlaubt, ein einzelnes AI-Signal an verschiedene nachfolgende Bausteine weiterzugeben. Der Baustein ist vollständig adapterbasiert und besitzt keine eigenen Ereignis- oder Datenschnittstellen. Er ist unter der Eclipse Public License 2.0 verfügbar.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine

Ereignis-Ausgänge

- Keine

Daten-Eingänge

- Keine

Daten-Ausgänge

- Keine

Adapter

Richtung	Name	Typ	Beschreibung

Socket (Eingang)	`IN`	`adapter::types::unidirectional::AI`	Eingang des zu verteilenden analogen Signals
Plug (Ausgang)	`OUT1`	`adapter::types::unidirectional::AI`	Erster Ausgang (Kopie des Eingangssignals)
Plug (Ausgang)	`OUT2`	`adapter::types::unidirectional::AI`	Zweiter Ausgang (Kopie des Eingangssignals)
Plug (Ausgang)	`OUT3`	`adapter::types::unidirectional::AI`	Dritter Ausgang (Kopie des Eingangssignals)
Plug (Ausgang)	`OUT4`	`adapter::types::unidirectional::AI`	Vierter Ausgang (Kopie des Eingangssignals)

Funktionsweise

Der Baustein stellt eine einfache 1:4-Verteilung eines analogen Signals dar. Das am Adapter `IN` anliegende Signal wird unverändert auf alle vier Ausgangs-Adapter (`OUT1` … `OUT4`) durchgereicht. Es findet keinerlei Skalierung, Filterung oder Umwandlung statt – das Signal wird auf feldbussebene 1:1 kopiert. Die Anzahl der Kanäle ist auf vier festgelegt.

Technische Besonderheiten

- **Reiner Adapter-Baustein**: Der FB enthält keine Ereignis- oder Datenschnittstellen; die gesamte Kommunikation erfolgt über Adapter.
- **Generischer Typ**: Über das Attribut `GenericClassName` (`'GEN_AI_SPLIT'`) wird der Baustein als generisch ausgewiesen. In der 4diac-IDE kann dadurch ein konkreter Typ abgeleitet werden.
- **Unidirektional**: Der verwendete Adaptertyp `unidirectional::AI` überträgt Daten nur in eine Richtung (vom Socket zu den Plugs).
- **Kein Zustandsautomat**: Da keine Ereignisse vorhanden sind, existiert kein ECC (Execution Control Chart). Der FB ist rein datenflussorientiert.

Zustandsübersicht

Der Baustein verfügt über keinen Zustandsautomaten – er ist ereignisfrei und gibt das Eingangssignal kontinuierlich an alle Ausgänge weiter. Eine Zustandsübersicht entfällt daher.

Anwendungsszenarien

- **Analog-Signalverteilung**: Ein einzelner Sensorwert (z. B. Temperatur, Druck, Füllstand) soll an mehrere Steuerungslogiken oder Visualisierungskomponenten gleichzeitig gesendet werden.
- **Redundanz oder Monitoring**: Das Signal wird parallel an einen Hauptregler und einen unabhängigen Überwachungsbaustein weitergeleitet.
- **Entkopplung von Subsystemen**: In einer modularen Steuerung können verschiedene Subsysteme denselben Analogwert empfangen, ohne dass ein Mehrfachzugriff auf den Sensor erforderlich ist.

Vergleich mit ähnlichen Bausteinen

- **AI_SPLIT_2 / AI_SPLIT_8**: Ähnliche Bausteine mit 2 bzw. 8 Ausgängen. Die Wahl hängt von der benötigten Anzahl paralleler Signale ab.
- **AI_SELECT** oder **AI_MUX**: Diese Bausteine wählen aus mehreren Eingängen einen aus, während der AI_SPLIT das Eingangssignal vervielfacht.
- **Standard-Adaptersplit**: Manche Bibliotheken bieten generische Split-Bausteine, die jedoch oft zusätzliche Ereignis- oder Datenschnittstellen benötigen. Der AI_SPLIT_4_UNGATED ist durch seinen reinen Adapteransatz besonders einfach in Adapterketten integrierbar.
- `AI_SPLIT_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

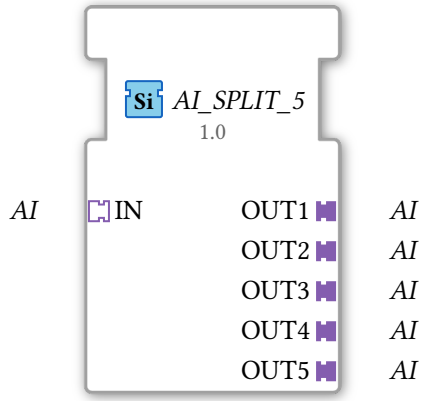
Fazit

Der Funktionsblock **AI_SPLIT_4_UNGATED** ist ein spezialisierter, aber einfach einsetzbarer Baustein zur Vervielfachung analoger Signale. Dank seiner reinen Adapterschnittstelle lässt er sich nahtlos in 4diac-Projekte einbinden, ohne zusätzliche Ereignislogik. Er eignet sich besonders für modulare und skalierbare Automatisierungslösungen, bei denen ein einzelner Messwert an mehrere Verbraucher verteilt werden muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AI_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AI_SPLIT_5** dient dazu, ein eingehendes analogen Signal (AI) auf fünf identische Ausgänge zu verteilen. Es handelt sich um einen generischen Baustein, der speziell für die Aufteilung eines Analogwerts in mehrere parallele Pfade ausgelegt ist. Die Anwendung erfolgt über unidirektionale Adapter vom Typ `AI`.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden. Die Signalübertragung erfolgt ausschließlich über den Adapter-Eingang.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden. Die Signalübertragung erfolgt ausschließlich über die Adapter-Ausgänge.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	IN	`adapter::types::unidirectional::AI`	Empfängt das eingehende analoge Signal.
Plug	OUT1	`adapter::types::unidirectional::AI`	Erster Ausgang, stellt das gleiche Signal wie IN bereit.

Plug	OUT2	`adapter::types::unidirectional::AI`	Zweiter Ausgang, stellt das gleiche Signal wie IN bereit.
Plug	OUT3	`adapter::types::unidirectional::AI`	Dritter Ausgang, stellt das gleiche Signal wie IN bereit.
Plug	OUT4	`adapter::types::unidirectional::AI`	Vierter Ausgang, stellt das gleiche Signal wie IN bereit.
Plug	OUT5	`adapter::types::unidirectional::AI`	Fünfter Ausgang, stellt das gleiche Signal wie IN bereit.

Funktionsweise

Der Baustein leitet das am Adapter-Eingang *IN* anliegende analoge Signal unverändert an alle fünf Adapter-Ausgänge *OUT1* bis *OUT5* weiter. Es findet keinerlei Signalverarbeitung oder Pufferung statt; die Verteilung erfolgt rein topologisch auf der Verbindungsebene. Der FB ist dadurch ideal geeignet, um ein einzelnes Analog-Signal an mehrere nachfolgende Funktionen oder Geräte zu verteilen, ohne dass eine Vervielfältigung auf Applikationsebene programmiert werden muss.

Technische Besonderheiten

- **Generischer Typ**: Der FB ist als generischer Baustein (`GEN_AI_SPLIT`) implementiert und kann durch entsprechende Typzuweisungen an unterschiedliche analoge Signaltypen angepasst werden.
- **Unidirektionale Adapter**: Sämtliche Adapter sind unidirektional ausgelegt, d.h. die Daten fließen nur vom Eingang zu den Ausgängen.
- **Kein internes Zustandsverhalten**: Der FB besitzt kein eigenes Zustandsmodell (ECC), da er rein kombinatorisch arbeitet.
- **Skalierbarkeit**: Der Baustein ist auf fünf Ausgänge fest ausgelegt. Für andere Anzahlen können ähnliche Bausteine (z.B. AI_SPLIT_2, AI_SPLIT_3) verwendet oder durch Anpassung des generischen Musters erstellt werden.

Zustandsübersicht

Der FB besitzt kein explizites Zustandsdiagramm. Da keine Ereignisse verarbeitet werden, ist das Verhalten statisch: das Signal am Eingang wird permanent auf die Ausgänge durchgeschaltet. Eine Initialisierung oder ein spezielles Verhalten bei Inbetriebnahme ist nicht erforderlich.

Anwendungsszenarien

- **Verteilung eines Messwerts**: Ein analoger Sensor (z.B. Temperatur, Druck) liefert einen Wert, der parallel an mehrere Regelungs-, Überwachungs- oder Visualisierungskomponenten weitergegeben werden muss.
- **Parallele Verarbeitung**: In einer Steuerungsapplikation soll derselbe Analogwert in verschiedenen Zweigen unterschiedlich weiterverarbeitet werden (z.B. sowohl in einer PID-Regelung als auch in einer Grenzwertüberwachung).
- **Redundanz**: Das Signal wird an mehrere unabhängige Auswerteeinheiten gesendet, um Ausfallsicherheit zu gewährleisten.

Vergleich mit ähnlichen Bausteinen

- **AI_SPLIT_2, AI_SPLIT_3, AI_SPLIT_N**: Diese Bausteine teilen ein analoges Signal auf eine andere Anzahl von Ausgängen auf. AI_SPLIT_5 ist speziell für fünf Ausgänge optimiert.
- **Analoge Splitter in anderen Bibliotheken**: Oft werden solche Funktionen durch Adapter-Kopplung oder Datenkopierer realisiert. Der AI_SPLIT_5 bietet eine saubere, wiederverwendbare Struktur innerhalb des 4diac-Frameworks.
- **Unterschied zu Daten-Ereignis-basierten Splittern**: Da der FB keine Ereignisse nutzt, ist er rein signalflussorientiert und benötigt keine Synchronisation oder Ablaufsteuerung.
- `AI_SPLIT_5_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

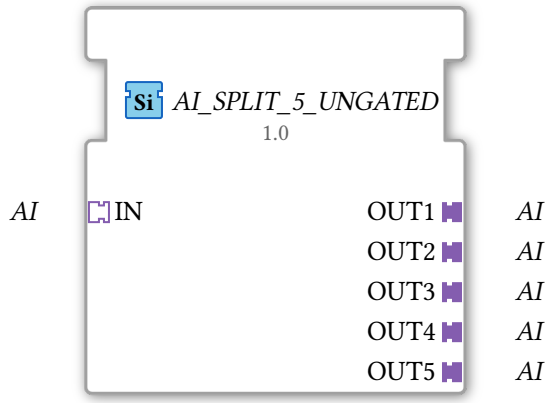
Der AI_SPLIT_5 ist ein einfacher, aber nützlicher Funktionsblock zur Vervielfältigung eines analogen Signals. Seine generische Natur und die klare Adapter-Schnittstelle machen ihn zu einem idealen Baustein für modulare Automatisierungslösungen, bei denen ein Analogwert mehrfach benötigt wird. Die fehlende Ereignissteuerung reduziert die Komplexität und ermöglicht eine direkte Signalweitergabe ohne zeitliche Verzögerungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AI_SPLIT_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AI_SPLIT_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AI_SPLIT_5_UNGATED** dient dazu, ein eingehendes analogen Signal (AI) auf fünf identische Ausgänge zu verteilen. Es handelt sich um einen generischen Baustein, der speziell für die Aufteilung eines Analogwerts in mehrere parallele Pfade ausgelegt ist. Die Anwendung erfolgt über unidirektionale Adapter vom Typ `AI`.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden. Die Signalübertragung erfolgt ausschließlich über den Adapter-Eingang.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden. Die Signalübertragung erfolgt ausschließlich über die Adapter-Ausgänge.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	IN	`adapter::types::unidirectional::AI`	Empfängt das eingehende analoge Signal.
Plug	OUT1	`adapter::types::unidirectional::AI`	Erster Ausgang, stellt das gleiche Signal wie IN bereit.
Plug	OUT2	`adapter::types::unidirectional::AI`	Zweiter Ausgang, stellt das gleiche Signal wie IN bereit.
Plug	OUT3	`adapter::types::unidirectional::AI`	Dritter Ausgang, stellt das gleiche Signal wie IN bereit.
Plug	OUT4	`adapter::types::unidirectional::AI`	Vierter Ausgang, stellt das gleiche Signal wie IN bereit.
Plug	OUT5	`adapter::types::unidirectional::AI`	Fünfter Ausgang, stellt das gleiche Signal wie IN bereit.

Funktionsweise

Der Baustein leitet das am Adapter-Eingang *IN* anliegende analoge Signal unverändert an alle fünf Adapter-Ausgänge *OUT1* bis *OUT5* weiter. Es findet keinerlei Signalverarbeitung oder Pufferung statt; die Verteilung erfolgt rein topologisch auf der Verbindungsebene. Der FB ist dadurch ideal geeignet, um ein einzelnes Analog-Signal an mehrere nachfolgende Funktionen oder Geräte zu verteilen, ohne dass eine Vervielfältigung auf Applikationsebene programmiert werden muss.

Technische Besonderheiten

- **Generischer Typ**: Der FB ist als generischer Baustein (`GEN_AI_SPLIT`) implementiert und kann durch entsprechende Typzuweisungen an unterschiedliche analoge Signaltypen angepasst werden.
- **Unidirektionale Adapter**: Sämtliche Adapter sind unidirektional ausgelegt, d.h. die Daten fließen nur vom Eingang zu den Ausgängen.
- **Kein internes Zustandsverhalten**: Der FB besitzt kein eigenes Zustandsmodell (ECC), da er rein kombinatorisch arbeitet.
- **Skalierbarkeit**: Der Baustein ist auf fünf Ausgänge fest ausgelegt. Für andere Anzahlen können ähnliche Bausteine (z.B. AI_SPLIT_2, AI_SPLIT_3) verwendet oder durch Anpassung des generischen Musters erstellt werden.

Zustandsübersicht

Der FB besitzt kein explizites Zustandsdiagramm. Da keine Ereignisse verarbeitet werden, ist das Verhalten statisch: das Signal am Eingang wird permanent auf die Ausgänge durchgeschaltet. Eine Initialisierung oder ein spezielles Verhalten bei Inbetriebnahme ist nicht erforderlich.

Anwendungsszenarien

- **Verteilung eines Messwerts**: Ein analoger Sensor (z.B. Temperatur, Druck) liefert einen Wert, der parallel an mehrere Regelungs-, Überwachungs- oder Visualisierungskomponenten weitergegeben werden muss.
- **Parallele Verarbeitung**: In einer Steuerungsapplikation soll derselbe Analogwert in verschiedenen Zweigen unterschiedlich weiterverarbeitet werden (z.B. sowohl in einer PID-Regelung als auch in einer Grenzwertüberwachung).

- **Redundanz**: Das Signal wird an mehrere unabhängige Auswerteeinheiten gesendet, um Ausfallsicherheit zu gewährleisten.

Vergleich mit ähnlichen Bausteinen

- **AI_SPLIT_2, AI_SPLIT_3, AI_SPLIT_N**: Diese Bausteine teilen ein analoges Signal auf eine andere Anzahl von Ausgängen auf. AI_SPLIT_5_UNGATED ist speziell für fünf Ausgänge optimiert.
- **Analoge Splitter in anderen Bibliotheken**: Oft werden solche Funktionen durch Adapter-Kopplung oder Datenkopierer realisiert. Der AI_SPLIT_5_UNGATED bietet eine saubere, wiederverwendbare Struktur innerhalb des 4diac-Frameworks.
- **Unterschied zu Daten-Ereignis-basierten Splittern**: Da der FB keine Ereignisse nutzt, ist er rein signalflussorientiert und benötigt keine Synchronisation oder Ablaufsteuerung.
- `AI_SPLIT_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

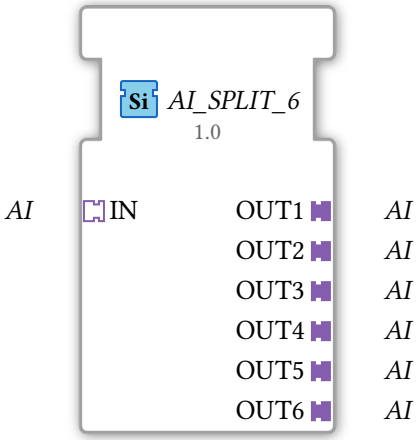
Der AI_SPLIT_5_UNGATED ist ein einfacher, aber nützlicher Funktionsblock zur Vervielfältigung eines analogen Signals. Seine generische Natur und die klare Adapter-Schnittstelle machen ihn zu einem idealen Baustein für modulare Automatisierungslösungen, bei denen ein Analogwert mehrfach benötigt wird. Die fehlende Ereignissteuerung reduziert die Komplexität und ermöglicht eine direkte Signalweitergabe ohne zeitliche Verzögerungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AI_SPLIT_6

Einleitung



Der Funktionsblock **AI_SPLIT_6** dient als generischer Splitter, der einen analogen Eingangswert (AI) auf sechs identische Ausgänge verteilt. Er realisiert eine 1:6-Verteilung eines AI-Adapters, ohne dass Daten oder Ereignisse modifiziert werden. Der Baustein ist als generischer FB (GenericClassName: `GEN_AI_SPLIT`) definiert und arbeitet rein adapterbasiert.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Eingang (Socket)	IN	`adapter::types::unidirectional::AI`	Einzige analoge Eingangsadapter (Quelle)
Ausgang (Plug)	OUT1	`adapter::types::unidirectional::AI`	Erster Ausgang (1:1-Weiterleitung von IN)
Ausgang (Plug)	OUT2	`adapter::types::unidirectional::AI`	Zweiter Ausgang
Ausgang (Plug)	OUT3	`adapter::types::unidirectional::AI`	Dritter Ausgang
Ausgang (Plug)	OUT4	`adapter::types::unidirectional::AI`	Vierter Ausgang
Ausgang (Plug)	OUT5	`adapter::types::unidirectional::AI`	Fünfter Ausgang
Ausgang (Plug)	OUT6	`adapter::types::unidirectional::AI`	Sechster Ausgang

Funktionsweise

Der Baustein besitzt keinerlei interne Logik oder Zustandsautomaten. Er fungiert als passiver Splitter: Der am Socket **IN** anliegende AI-Adapter wird unverändert an alle sechs Plug-Ausgänge **OUT1** bis **OUT6** dupliziert. Änderungen am AI-Wert (z. B. Messwert oder Status) an der Quelle werden sofort und gleichzeitig an allen Ausgängen sichtbar.

Technische Besonderheiten

- **Generischer FB**: Der Baustein ist mittels `eclipse4diac::core::GenericClassName` als `GEN_AI_SPLIT` markiert. Dadurch kann er je nach Laufzeitumgebung parametrisiert oder instanziiert werden, ohne seine grundlegende Splitterfunktion zu ändern.
- **Reine Adapter-Kopplung**: Es werden weder Ereignisse noch Datenport-Variablen verwendet, sondern ausschließlich unidirektionale Adapter des Typs `adapter::types::unidirectional::AI`. Dies reduziert den Overhead und ermöglicht eine direkte Verschaltung mit anderen AI-Adapter-Bausteinen.
- **Keine Zustände**: Da kein ECC (Execution Control Chart) vorhanden ist, benötigt der FB keine Algorithmen oder Zustandsübergänge. Die Weiterleitung erfolgt rein strukturell.

Zustandsübersicht

Der Baustein besitzt keine Zustandsmaschine (ECC) und führt keine aktiven Operationen aus. Es gibt keine unterscheidbaren Betriebszustände.

Anwendungsszenarien

- **Verteilung eines analogen Sensorsignals**: Ein einzelner analoger Eingang (z. B. 4-20 mA-Messumformer) soll parallel an mehrere Steuerungsfunktionen oder Anzeigeeinheiten weitergegeben werden.
- **Redundanz / Plausibilisierung**: Derselbe Messwert wird auf mehrere parallel arbeitende Auswerte-Bausteine geführt, die z. B. eine Mittelwertbildung oder Fehlererkennung durchführen.
- **Simulation und Test**: Ein simulierter AI-Adapter wird auf mehrere Testbausteine verteilt, um das Systemverhalten unter Last zu prüfen.

Vergleich mit ähnlichen Bausteinen

- **AI_SPLIT_2 / AI_SPLIT_4**: Diese Bausteine teilen einen AI-Eingang auf zwei bzw. vier Ausgänge. **AI_SPLIT_6** stellt die Variante mit sechs Ausgängen dar. Alle Splitter haben identische Funktionsweise und unterscheiden sich nur in der Anzahl der Ausgangsadapter.

- **AI_DUPLICATE (hypothetisch)**: Ein Baustein, der den Wert zu Kopierzwecken dupliziert, jedoch meist mit Datenports arbeitet. AI_SPLIT_6 nutzt dagegen ausschließlich Adapter, was die direkte Kopplung ohne zusätzliche Ereignissteuerung erlaubt.
- `AI_SPLIT_6_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der **AI_SPLIT_6** ist ein minimaler, aber nützlicher Baustein zur Verteilung analoger Adaptersignale in der 4diac-IDE. Seine generische Definition und die reine Adapter-Schnittstelle machen ihn besonders geeignet für Architekturen, die eine lose Kopplung und einfache Erweiterbarkeit erfordern. Durch die Aufteilung auf sechs Ausgänge wird eine flexible Mehrfachnutzung eines analogen Eingangswerts ermöglicht.

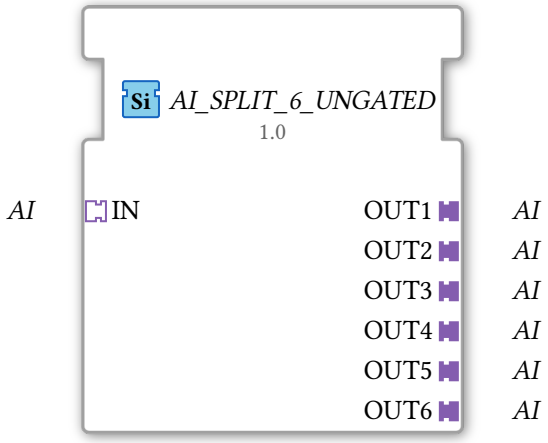
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AI_SPLIT_6_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AI_SPLIT_6`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

Einleitung



Der Funktionsblock **AI_SPLIT_6_UNGATED** dient als generischer Splitter, der einen analogen Eingangswert (AI) auf sechs identische Ausgänge verteilt. Er realisiert eine 1:6-Verteilung eines AI-Adapters, ohne dass Daten oder Ereignisse modifiziert werden. Der Baustein ist als generischer FB (GenericClass-Name: `GEN_AI_SPLIT`) definiert und arbeitet rein adapterbasiert.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Eingang (Socket)	IN	`adapter::types::unidirectional::AI`	Einzige analoge Eingangsadapter (Quelle)
Ausgang (Plug)	OUT1	`adapter::types::unidirectional::AI`	Erster Ausgang (1:1-Weiterleitung von IN)
Ausgang (Plug)	OUT2	`adapter::types::unidirectional::AI`	Zweiter Ausgang
Ausgang (Plug)	OUT3	`adapter::types::unidirectional::AI`	Dritter Ausgang
Ausgang (Plug)	OUT4	`adapter::types::unidirectional::AI`	Vierter Ausgang
Ausgang (Plug)	OUT5	`adapter::types::unidirectional::AI`	Fünfter Ausgang
Ausgang (Plug)	OUT6	`adapter::types::unidirectional::AI`	Sechster Ausgang

Funktionsweise

Der Baustein besitzt keinerlei interne Logik oder Zustandsautomaten. Er fungiert als passiver Splitter: Der am Socket **IN** anliegende AI-Adapter wird unverändert an alle sechs Plug-Ausgänge **OUT1** bis **OUT6** dupliziert. Änderungen am AI-Wert (z. B. Messwert oder Status) an der Quelle werden sofort und gleichzeitig an allen Ausgängen sichtbar.

Technische Besonderheiten

- **Generischer FB**: Der Baustein ist mittels `eclipse4diac::core::GenericClassName` als `GEN_AI_SPLIT` markiert. Dadurch kann er je nach Laufzeitumgebung parametrisiert oder instanziiert werden, ohne seine grundlegende Splitterfunktion zu ändern.
- **Reine Adapter-Kopplung**: Es werden weder Ereignisse noch Datenport-Variablen verwendet, sondern ausschließlich unidirektionale Adapter des Typs `adapter::types::unidirectional::AI`. Dies reduziert den Overhead und ermöglicht eine direkte Verschaltung mit anderen AI-Adapter-Bausteinen.
- **Keine Zustände**: Da kein ECC (Execution Control Chart) vorhanden ist, benötigt der FB keine Algorithmen oder Zustandsübergänge. Die Weiterleitung erfolgt rein strukturell.

Zustandsübersicht

Der Baustein besitzt keine Zustandsmaschine (ECC) und führt keine aktiven Operationen aus. Es gibt keine unterscheidbaren Betriebszustände.

Anwendungsszenarien

- **Verteilung eines analogen Sensorsignals**: Ein einzelner analoger Eingang (z. B. 4-20 mA-Messumformer) soll parallel an mehrere Steuerungsfunktionen oder Anzeigeeinheiten weitergegeben werden.
- **Redundanz / Plausibilisierung**: Derselbe Messwert wird auf mehrere parallel arbeitende Auswerte-Bausteine geführt, die z. B. eine Mittelwertbildung oder Fehlererkennung durchführen.
- **Simulation und Test**: Ein simulierter AI-Adapter wird auf mehrere Testbausteine verteilt, um das Systemverhalten unter Last zu prüfen.

Vergleich mit ähnlichen Bausteinen

- **AI_SPLIT_2 / AI_SPLIT_4**: Diese Bausteine teilen einen AI-Eingang auf zwei bzw. vier Ausgänge. **AI_SPLIT_6_UNGATED** stellt die Variante mit sechs Ausgängen dar. Alle Splitter haben identische Funktionsweise und unterscheiden sich nur in der Anzahl der Ausgangsadapter.

- **AI_DUPLICATE (hypothetisch)**: Ein Baustein, der den Wert zu Kopierzwecken dupliziert, jedoch meist mit Datenports arbeitet. AI_SPLIT_6_UNGATED nutzt dagegen ausschließlich Adapter, was die direkte Kopplung ohne zusätzliche Ereignissteuerung erlaubt.
- `AI_SPLIT_6`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

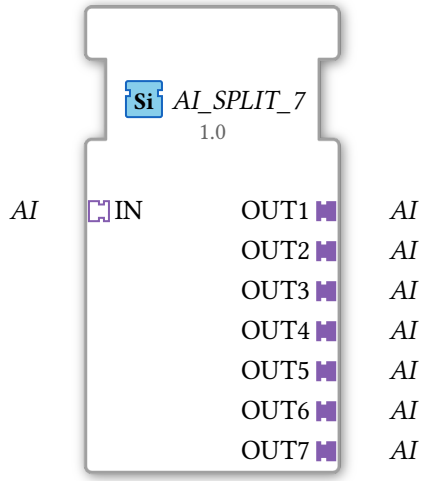
Fazit

Der **AI_SPLIT_6_UNGATED** ist ein minimaler, aber nützlicher Baustein zur Verteilung analoger Adaptersignale in der 4diac-IDE. Seine generische Definition und die reine Adapter-Schnittstelle machen ihn besonders geeignet für Architekturen, die eine lose Kopplung und einfache Erweiterbarkeit erfordern. Durch die Aufteilung auf sechs Ausgänge wird eine flexible Mehrfachnutzung eines analogen Eingangswerts ermöglicht.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AI_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AI_SPLIT_7** dient dazu, einen einzelnen analogen Eingang (AI) auf sieben identische analoge Ausgänge zu verteilen. Es handelt sich um einen generischen Baustein, der den eingehenden analogen Wert unverändert an alle sieben Ausgangsadapter weitergibt. Dadurch lassen sich mehrere Verbraucher aus einer Signalquelle versorgen, ohne das Signal mehrfach erfassen oder duplizieren zu müssen.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine Ereignis-Eingänge. Die Datenweitergabe erfolgt rein über Adapter.

Ereignis-Ausgänge

Es sind keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Direkte Dateneingänge sind nicht definiert. Der analoge Wert wird ausschließlich über den Adapter **IN** (Socket) eingelesen.

Daten-Ausgänge

Direkte Datenausgänge sind nicht vorhanden. Die analogen Werte werden über die sieben Adapter **OUT1** bis **OUT7** (Plugs) ausgegeben.

Adapter

- **IN** (Socket): Typ `adapter::types::unidirectional::AI` – Eingang für das analoge Signal.
- **OUT1** bis **OUT7** (Plugs): Typ `adapter::types::unidirectional::AI` – sieben Ausgänge, die den eingehenden Wert identisch weiterleiten.

Funktionsweise

AI_SPLIT_7 arbeitet als reiner Datenverteiler. Ein am Adapter **IN** anliegender analoger Wert wird ohne Verzögerung oder Transformation auf alle sieben Ausgangsadapter (**OUT1** bis **OUT7**) kopiert. Dadurch erhalten alle angeschlossenen Komponenten denselben analogen Wert. Die interne Logik ist auf eine unidirektionale Datenübertragung ausgelegt.

Technische Besonderheiten

- **Generischer Baustein:** Der FB ist als generischer Typ (`GEN_AI_SPLIT`) implementiert und kann flexibel in verschiedene Anwendungen integriert werden.
- **Adapter-basiert:** Die Schnittstellen sind als IEC 61499-Adapter realisiert, was eine lose Kopplung zwischen Bausteinen ermöglicht.
- **Keine Ereignissteuerung:** Da keine Ereignisse verwendet werden, erfolgt die Datenweitergabe ausschließlich über die Adapter verbindungen und wird durch die Laufzeitumgebung getrieben.

Zustandsübersicht

Der Baustein besitzt keine expliziten Zustände, da er keine sequentielle oder ereignisgesteuerte Logik enthält. Er arbeitet kontinuierlich als reiner Signalverteiler.

Anwendungsszenarien

- **Verteilen eines Sensorsignals:** Ein analoger Sensor (z. B. Temperatur, Druck) soll an mehrere Steuerungslogiken oder Anzeigen weitergegeben werden.
- **Redundante Verarbeitung:** Derselbe Messwert wird parallel von verschiedenen Algorithmen oder Überwachungseinheiten benötigt.
- **Simulation/Test:** Ein einzelner generierter Analogwert soll an mehrere Eingänge eines Gesamtsystems gespeist werden.

Vergleich mit ähnlichen Bausteinen

- **AI_SPLIT_N:** Bausteine wie `AI_SPLIT_2` oder `AI_SPLIT_3` unterscheiden sich nur in der Anzahl der Ausgangsadapter. `AI_SPLIT_7` bietet die maximale Verteilung auf sieben Kanäle.
- **Andere Splitter:** Im Gegensatz zu ereignisgesteuerten Splittern (z. B. `E_SPLIT`) arbeitet dieser FB rein analog ohne Triggerung durch Ereignisse. Er ist für kontinuierliche analoge Signale optimiert.
- `AI_SPLIT_7_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

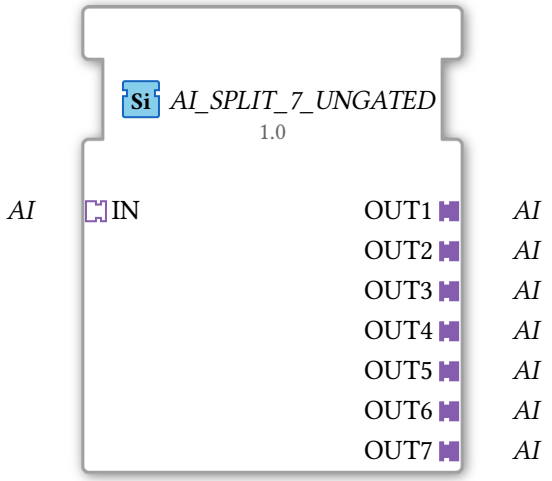
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

AI_SPLIT_7 ist ein einfacher, aber effektiver Funktionsblock zur Vervielfältigung analoger Signale in IEC 61499-Systemen. Durch den Einsatz von Adaptern und die generische Implementierung lässt er sich leicht in unterschiedliche Automatisierungs- und Steuerungsanwendungen einbinden. Seine klare, ereignislose Struktur macht ihn besonders geeignet für unidirektionale Datenweitergabe ohne zusätzliche Steuerlogik.

AI_SPLIT_7_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `AI_SPLIT_7`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AI_SPLIT_7_UNGATED** dient dazu, einen einzelnen analogen Eingang (AI) auf sieben identische analoge Ausgänge zu verteilen. Es handelt sich um einen generischen Baustein, der den eingehenden analogen Wert unverändert an alle sieben Ausgangsadapter weitergibt. Dadurch lassen sich mehrere Verbraucher aus einer Signalquelle versorgen, ohne das Signal mehrfach erfassen oder duplizieren zu müssen.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine Ereignis-Eingänge. Die Datenweitergabe erfolgt rein über Adapter.

Ereignis-Ausgänge

Es sind keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Direkte Dateneingänge sind nicht definiert. Der analoge Wert wird ausschließlich über den Adapter **IN** (Socket) eingelesen.

Daten-Ausgänge

Direkte Datenausgänge sind nicht vorhanden. Die analogen Werte werden über die sieben Adapter **OUT1** bis **OUT7** (Plugs) ausgegeben.

Adapter

- **IN** (Socket): Typ `adapter::types::unidirectional::AI` – Eingang für das analoge Signal.
- **OUT1** bis **OUT7** (Plugs): Typ `adapter::types::unidirectional::AI` – sieben Ausgänge, die den eingehenden Wert identisch weiterleiten.

Funktionsweise

AI_SPLIT_7_UNGATED arbeitet als reiner Datenverteiler. Ein am Adapter **IN** anliegender analoger Wert wird ohne Verzögerung oder Transformation auf alle sieben Ausgangsadapter (**OUT1** bis **OUT7**) kopiert. Dadurch erhalten alle angeschlossenen Komponenten denselben analogen Wert. Die interne Logik ist auf eine unidirektionale Datenübertragung ausgelegt.

Technische Besonderheiten

- **Generischer Baustein:** Der FB ist als generischer Typ (`GEN_AI_SPLIT`) implementiert und kann flexibel in verschiedene Anwendungen integriert werden.
- **Adapter-basiert:** Die Schnittstellen sind als IEC 61499-Adapter realisiert, was eine lose Kopplung zwischen Bausteinen ermöglicht.
- **Keine Ereignissteuerung:** Da keine Ereignisse verwendet werden, erfolgt die Datenweitergabe ausschließlich über die Adapter verbindungen und wird durch die Laufzeitumgebung getrieben.

Zustandsübersicht

Der Baustein besitzt keine expliziten Zustände, da er keine sequentielle oder ereignisgesteuerte Logik enthält. Er arbeitet kontinuierlich als reiner Signalverteiler.

Anwendungsszenarien

- **Verteilen eines Sensorsignals:** Ein analoger Sensor (z. B. Temperatur, Druck) soll an mehrere Steuerungslogiken oder Anzeigen weitergegeben werden.
- **Redundante Verarbeitung:** Derselbe Messwert wird parallel von verschiedenen Algorithmen oder Überwachungseinheiten benötigt.
- **Simulation/Test:** Ein einzelner generierter Analogwert soll an mehrere Eingänge eines Gesamtsystems gespeist werden.

Vergleich mit ähnlichen Bausteinen

- **AI_SPLIT_N:** Bausteine wie `AI_SPLIT_2` oder `AI_SPLIT_3` unterscheiden sich nur in der Anzahl der Ausgangsadapter. `AI_SPLIT_7_UNGATED` bietet die maximale Verteilung auf sieben Kanäle.
- **Andere Splitter:** Im Gegensatz zu ereignisgesteuerten Splittern (z. B. `E_SPLIT`) arbeitet dieser FB rein analog ohne Triggerung durch Ereignisse. Er ist für kontinuierliche analoge Signale optimiert.
- `AI_SPLIT_7_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- `AI_SPLIT_7`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

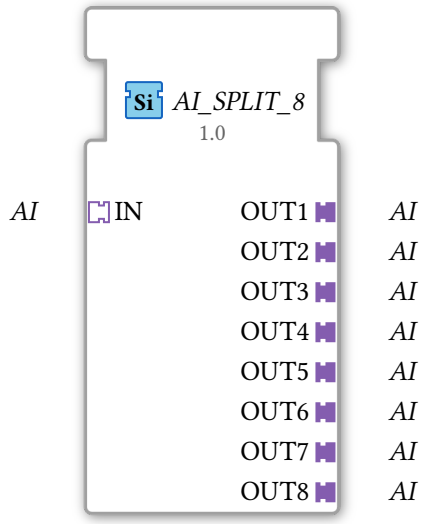
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

AI_SPLIT_7_UNGATED ist ein einfacher, aber effektiver Funktionsblock zur Vervielfältigung analoger Signale in IEC 61499-Systemen. Durch den Einsatz von Adaptern und die generische Implementierung lässt er sich leicht in unterschiedliche Automatisierungs- und Steuerungsanwendungen einbinden. Seine klare, ereignislose Struktur macht ihn besonders geeignet für unidirektionale Datenweitergabe ohne zusätzliche Steuerlogik.

AI_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AI_SPLIT_8** dient dazu, ein analoges Eingangssignal (Typ *AI*) auf acht identische analoge Ausgangssignale zu verteilen. Der Baustein ist als generischer FB (GenericClassName: `GEN_AI_SPLIT`) implementiert und wird in der 4diac-IDE verwendet, um ein einzelnes analoges Signal mehrfach nutzbar zu machen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Name	Typ	Richtung	Beschreibung
IN	`adapter::types::unidirectional::AI`	Socket (Eingang)	Empfängt das analoge Signal, das auf alle Ausgänge verteilt wird.
OUT1 – **OUT8**	`adapter::types::unidirectional::AI`	Plug (Ausgang)	Acht identische Ausgänge, die das am Eingang anliegende Signal bereitstellen.

Funktionsweise

Der FB besitzt keine eigene Verarbeitungslogik (keine Zustandsmaschine, kein ECC). Das am Adapter-Socket **IN** anliegende analoge Signal wird direkt und ohne Verzögerung an alle acht Adapter-Plugs (**OUT1** bis **OUT8**) weitergegeben. Es findet keine Signalumformung oder -verstärkung statt – das Signal wird eins zu eins dupliziert.

Technische Besonderheiten

- **Generischer Baustein:** Der FB ist als generischer Typ (`GEN_AI_SPLIT`) deklariert und kann daher zur Laufzeit mit einem konkreten analogen Datentyp parametrisiert werden.
- **Unidirektionale Adapter:** Alle verwendeten Adapter sind unidirektional, d.h. der Datenfluss verläuft ausschließlich vom Socket zu den Plugs.
- **Keine Ereignissteuerung:** Die Verteilung erfolgt ereignislos; ein anliegendes Signal wird sofort an alle Ausgänge weitergeleitet.

Zustandsübersicht

Der FB besitzt keine interne Zustandsmaschine (ECC). Es gibt keine definierten Zustände – der Baustein arbeitet kontinuierlich und ohne eigenes Verhalten.

Anwendungsszenarien

- **Signalverteilung:** Ein analoges Sensorsignal (z.B. Temperatur, Druck) soll an mehrere Steuerungsmodule oder Überwachungseinheiten gleichzeitig gesendet werden.
- **Redundanz:** Mehrere parallele Auswertungen desselben Signals in unterschiedlichen Teilen einer Anlage.
- **Prototyping:** Schnelles Bereitstellen eines Signals für mehrere Testpunkte in einer Entwicklungs- oder Simulationsumgebung.

Vergleich mit ähnlichen Bausteinen

- **AI_SPLIT_2, AI_SPLIT_4:** Diese Bausteine verteilen ein analoges Signal auf zwei bzw. vier Ausgänge. Der vorliegende FB bietet mit acht Ausgängen eine höhere Anzahl paralleler Verbindungen.
- **DIO_SPLIT (digital):** Ein ähnlicher Splitter für digitale Signale, jedoch auf Basis von *DI/DO*-Adaptertypen. Der AI_SPLIT_8 ist spezifisch für analoge Signale ausgelegt.
- `AI_SPLIT_8_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

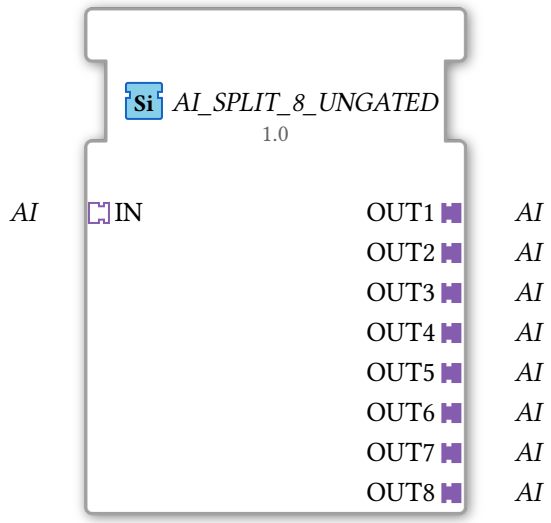
Der **AI_SPLIT_8** ist ein einfacher, generischer Funktionsblock zur Vervielfachung eines analogen Signals. Seine klare Struktur – ein Eingang, acht Ausgänge, keine Ereignisse – macht ihn ideal für die unkomplizierte Signalverteilung in Automatisierungsanwendungen. Der generische Charakter erlaubt eine flexible Anpassung an verschiedene analoge Datentypen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AI_SPLIT_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AI_SPLIT_8`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AI_SPLIT_8_UNGATED** dient dazu, ein analoges Eingangssignal (Typ *AI*) auf acht identische analoge Ausgangssignale zu verteilen. Der Baustein ist als generischer FB (GenericClassName: `GEN_AI_SPLIT`) implementiert und wird in der 4diac-IDE verwendet, um ein einzelnes analoges Signal mehrfach nutzbar zu machen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Name	Typ	Richtung	Beschreibung
IN	`adapter::types::unidirectional::AI`	Socket (Eingang)	Empfängt das analoge Signal, das auf alle Ausgänge verteilt wird.
OUT1 – OUT8	`adapter::types::unidirectional::AI`	Plug (Ausgang)	Acht identische Ausgänge, die das am Eingang anliegende Signal bereitstellen.

Funktionsweise

Der FB besitzt keine eigene Verarbeitungslogik (keine Zustandsmaschine, kein ECC). Das am Adapter-Socket **IN** anliegende analoge Signal wird direkt und ohne Verzögerung an alle acht Adapter-Plugs (**OUT1** bis **OUT8**) weitergegeben. Es findet keine Signalumformung oder -verstärkung statt – das Signal wird eins zu eins dupliziert.

Technische Besonderheiten

- **Generischer Baustein:** Der FB ist als generischer Typ (`GEN_AI_SPLIT`) deklariert und kann daher zur Laufzeit mit einem konkreten analogen Datentyp parametrisiert werden.
- **Unidirektionale Adapter:** Alle verwendeten Adapter sind unidirektional, d.h. der Datenfluss verläuft ausschließlich vom Socket zu den Plugs.
- **Keine Ereignissteuerung:** Die Verteilung erfolgt ereignislos; ein anliegendes Signal wird sofort an alle Ausgänge weitergeleitet.

Zustandsübersicht

Der FB besitzt keine interne Zustandsmaschine (ECC). Es gibt keine definierten Zustände – der Baustein arbeitet kontinuierlich und ohne eigenes Verhalten.

Anwendungsszenarien

- **Signalverteilung:** Ein analoges Sensorsignal (z.B. Temperatur, Druck) soll an mehrere Steuerungsmodule oder Überwachungseinheiten gleichzeitig gesendet werden.
- **Redundanz:** Mehrere parallele Auswertungen desselben Signals in unterschiedlichen Teilen einer Anlage.
- **Prototyping:** Schnelles Bereitstellen eines Signals für mehrere Testpunkte in einer Entwicklungs- oder Simulationsumgebung.

Vergleich mit ähnlichen Bausteinen

- **AI_SPLIT_2, AI_SPLIT_4:** Diese Bausteine verteilen ein analoges Signal auf zwei bzw. vier Ausgänge. Der vorliegende FB bietet mit acht Ausgängen eine höhere Anzahl paralleler Verbindungen.
- **DIO_SPLIT (digital):** Ein ähnlicher Splitter für digitale Signale, jedoch auf Basis von *DI/DO*-Adaptertypen. Der AI_SPLIT_8_UNGATED ist spezifisch für analoge Signale ausgelegt.
- `AI_SPLIT_8`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

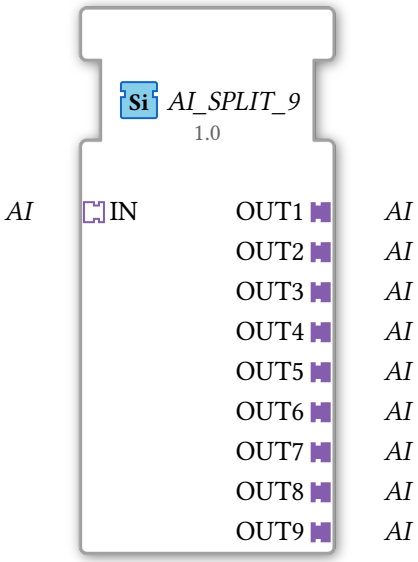
Fazit

Der **AI_SPLIT_8_UNGATED** ist ein einfacher, generischer Funktionsblock zur Vervielfachung eines analogen Signals. Seine klare Struktur – ein Eingang, acht Ausgänge, keine Ereignisse – macht ihn ideal für die unkomplizierte Signalverteilung in Automatisierungsanwendungen. Der generische Charakter erlaubt eine flexible Anpassung an verschiedene analoge Datentypen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AI_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AI_SPLIT_9** ist ein generischer Baustein, der ein einzelnes analoges Eingangssignal (Adaptertyp `AI`) auf neun identische Ausgänge (OUT1 bis OUT9) verteilt. Er dient als Signalverteiler, der es ermöglicht, ein AI-Signal mehrfach in einer Steuerungsanwendung zu nutzen, ohne die Quelle mehrfach zu verknüpfen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Eingang (Socket)	`IN`	`adapter::types::unidirectional::AI`	Einzelner analoger Eingang, der auf alle Ausgänge verteilt wird.
Ausgang (Plug)	`OUT1`	`adapter::types::unidirectional::AI`	Erster ausgehender AI-Kanal.
Ausgang (Plug)	`OUT2`	`adapter::types::unidirectional::AI`	Zweiter ausgehender AI-Kanal.
Ausgang (Plug)	`OUT3`	`adapter::types::unidirectional::AI`	Dritter ausgehender AI-Kanal.
Ausgang (Plug)	`OUT4`	`adapter::types::unidirectional::AI`	Vierter ausgehender AI-Kanal.
Ausgang (Plug)	`OUT5`	`adapter::types::unidirectional::AI`	Fünfter ausgehender AI-Kanal.
Ausgang (Plug)	`OUT6`	`adapter::types::unidirectional::AI`	Sechster ausgehender AI-Kanal.

Ausgang (Plug)	OUT7	adapter::types::unidirectional::AI	Siebter ausgehender AI-Kanal.
Ausgang (Plug)	OUT8	adapter::types::unidirectional::AI	Achter ausgehender AI-Kanal.
Ausgang (Plug)	OUT9	adapter::types::unidirectional::AI	Neunter ausgehender AI-Kanal.

Funktionsweise

Der Baustein arbeitet als reine Signalverteilung: Der am Adapter-Socket `IN` anliegende AI-Wert wird unverändert und gleichzeitig an alle neun Adapter-Plugs `OUT1` … `OUT9` weitergegeben. Es findet keine Signalverarbeitung, Skalierung oder sonstige Manipulation statt. Die Verteilung erfolgt kontinuierlich und ohne internes Zustandsverhalten – der Ausgang folgt dem Eingang ohne Verzögerung.

Technische Besonderheiten

- **Generischer Baustein:** Der FB ist als `GEN_AI_SPLIT` typisiert und nutzt die Generics-Mechanismen von Eclipse 4diac (`GenericClassName`, `TypeHash`). Dadurch kann er in Typ-Hierarchien und Bibliotheken flexibel eingesetzt werden.
- **Adapterbasierte Schnittstelle:** Statt einzelner Daten-Ports werden Adapter verwendet, die eine gekapselte, wiederverwendbare Schnittstelle für analoge Eingänge (`AI`) bereitstellen. Dies vereinfacht das Verschaltungskonzept in 4diac-Applikationen.
- **Keine Latenz:** Da keine interne Logik oder Abläufe vorhanden sind, ist die Weiterleitung des Signals verzögerungsfrei.

Zustandsübersicht

Der Baustein besitzt keinen internen Zustandsautomaten (kein ECC). Er verhält sich rein datenflussgesteuert: Der Wert des Eingangs wird permanent auf die Ausgänge durchgeschaltet.

Anwendungsszenarien

- **Signalvervielfachung:** Ein einzelner analoger Sensor (z. B. Temperatur, Druck) soll in mehreren Funktionsblöcken parallel ausgewertet werden.
- **Redundante Verarbeitung:** Ein AI-Signal wird an mehrere unabhängige Pfade weitergeleitet, um Ausfallsicherheit oder Plausibilitätsprüfungen zu ermöglichen.
- **Modulare Aufteilung:** In einer großen Steuerungsapplikation wird ein analoges Eingangssignal in verschiedene Module (z. B. Regelung, Überwachung, Visualisierung) geführt.

Vergleich mit ähnlichen Bausteinen

- **AI_SPLIT_2 / AI_SPLIT_4 / …:** Analoge Bausteine mit 2 bzw. 4 Ausgängen. `AI_SPLIT_9` erweitert das Konzept auf neun Ausgänge und ist daher für Anwendungen mit vielen Empfängern geeignet.
- **AI_MERGE**: Führt mehrere AI-Eingänge zu einem Ausgang zusammen – die gegenteilige Operation.
- **Direkte Adapter-Verknüpfung**: In 4diac kann ein einzelner Adapter-Port konzeptionell auch mehrfach verbunden werden, jedoch ist die explizite Splittung mit diesem FB dokumentierter und klarer strukturiert.
- `AI_SPLIT_9_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

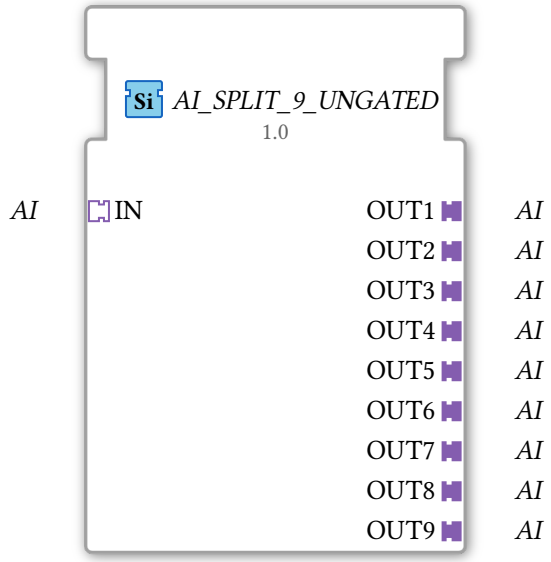
Der **AI_SPLIT_9** ist ein einfacher, aber nützlicher Funktionsblock zur Signalvervielfachung von analogen Eingängen in IEC 61499-Applikationen. Durch seinen generischen Aufbau und die Adapter-Schnittstelle fügt er sich nahtlos in das 4diac-Ökosystem ein und erleichtert die strukturierte Verschaltung von Signalen. Seine Verwendung empfiehlt sich überall dort, wo ein AI-Signal gleichzeitig an mehrere Verarbeitungseinheiten übergeben werden muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AI_SPLIT_9_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AI_SPLIT_9`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AI_SPLIT_9_UNGATED** ist ein generischer Baustein, der ein einzelnes analoges Eingangssignal (Adaptertyp `AI`) auf neun identische Ausgänge (OUT1 bis OUT9) verteilt. Er dient als Signalverteiler, der es ermöglicht, ein AI-Signal mehrfach in einer Steuerungsanwendung zu nutzen, ohne die Quelle mehrfach zu verknüpfen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung

Eingang (Socket)	IN	adapter::types::unidirectional::AI	Einzelner analoger Eingang, der auf alle Ausgänge verteilt wird.
Ausgang (Plug)	OUT1	adapter::types::unidirectional::AI	Erster ausgehender AI-Kanal.
Ausgang (Plug)	OUT2	adapter::types::unidirectional::AI	Zweiter ausgehender AI-Kanal.
Ausgang (Plug)	OUT3	adapter::types::unidirectional::AI	Dritter ausgehender AI-Kanal.
Ausgang (Plug)	OUT4	adapter::types::unidirectional::AI	Vierter ausgehender AI-Kanal.
Ausgang (Plug)	OUT5	adapter::types::unidirectional::AI	Fünfter ausgehender AI-Kanal.
Ausgang (Plug)	OUT6	adapter::types::unidirectional::AI	Sechster ausgehender AI-Kanal.
Ausgang (Plug)	OUT7	adapter::types::unidirectional::AI	Siebter ausgehender AI-Kanal.
Ausgang (Plug)	OUT8	adapter::types::unidirectional::AI	Achter ausgehender AI-Kanal.
Ausgang (Plug)	OUT9	adapter::types::unidirectional::AI	Neunter ausgehender AI-Kanal.

Funktionsweise

Der Baustein arbeitet als reine Signalverteilung: Der am Adapter-Socket `IN` anliegende AI-Wert wird unverändert und gleichzeitig an alle neun Adapter-Plugs `OUT1` … `OUT9` weitergegeben. Es findet keine Signalverarbeitung, Skalierung oder sonstige Manipulation statt. Die Verteilung erfolgt kontinuierlich und ohne internes Zustandsverhalten – der Ausgang folgt dem Eingang ohne Verzögerung.

Technische Besonderheiten

- **Generischer Baustein:** Der FB ist als `GEN_AI_SPLIT` typisiert und nutzt die Generics-Mechanismen von Eclipse 4diac (`GenericClassName`, `TypeHash`). Dadurch kann er in Typ-Hierarchien und Bibliotheken flexibel eingesetzt werden.
- **Adapterbasierte Schnittstelle:** Statt einzelner Daten-Ports werden Adapter verwendet, die eine gekapselte, wiederverwendbare Schnittstelle für analoge Eingänge (`AI`) bereitstellen. Dies vereinfacht das Verschaltungskonzept in 4diac-Applikationen.
- **Keine Latenz:** Da keine interne Logik oder Abläufe vorhanden sind, ist die Weiterleitung des Signals verzögerungsfrei.

Zustandsübersicht

Der Baustein besitzt keinen internen Zustandsautomaten (kein ECC). Er verhält sich rein datenflussgesteuert: Der Wert des Eingangs wird permanent auf die Ausgänge durchgeschaltet.

Anwendungsszenarien

- **Signalvervielfachung:** Ein einzelner analoger Sensor (z. B. Temperatur, Druck) soll in mehreren Funktionsblöcken parallel ausgewertet werden.
- **Redundante Verarbeitung:** Ein AI-Signal wird an mehrere unabhängige Pfade weitergeleitet, um Ausfallsicherheit oder Plausibilitätsprüfungen zu ermöglichen.
- **Modulare Aufteilung:** In einer großen Steuerungsapplikation wird ein analoges Eingangssignal in verschiedene Module (z. B. Regelung, Überwachung, Visualisierung) geführt.

Vergleich mit ähnlichen Bausteinen

- **AI_SPLIT_2 / AI_SPLIT_4 / …**: Analoge Bausteine mit 2 bzw. 4 Ausgängen. `AI_SPLIT_9_UNGATED` erweitert das Konzept auf neun Ausgänge und ist daher für Anwendungen mit vielen Empfängern geeignet.
- **AI_MERGE**: Führt mehrere AI-Eingänge zu einem Ausgang zusammen – die gegenteilige Operation.
- **Direkte Adapter-Verknüpfung**: In 4diac kann ein einzelner Adapter-Port konzeptionell auch mehrfach verbunden werden, jedoch ist die explizite Splittung mit diesem FB dokumentierter und klarer strukturiert.
- **`AI_SPLIT_9_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- **`AI_SPLIT_9`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

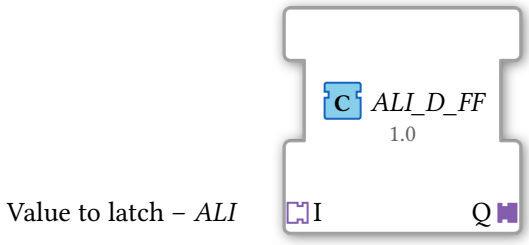
Der **AI_SPLIT_9_UNGATED** ist ein einfacher, aber nützlicher Funktionsblock zur Signalvervielfachung von analogen Eingängen in IEC 61499-Applikationen. Durch seinen generischen Aufbau und die Adapter-Schnittstelle fügt er sich nahtlos in das 4diac-Ökosystem ein und erleichtert die strukturierte Verschaltung von Signalen. Seine Verwendung empfiehlt sich überall dort, wo ein AI-Signal gleichzeitig an mehrere Verarbeitungseinheiten übergeben werden muss.

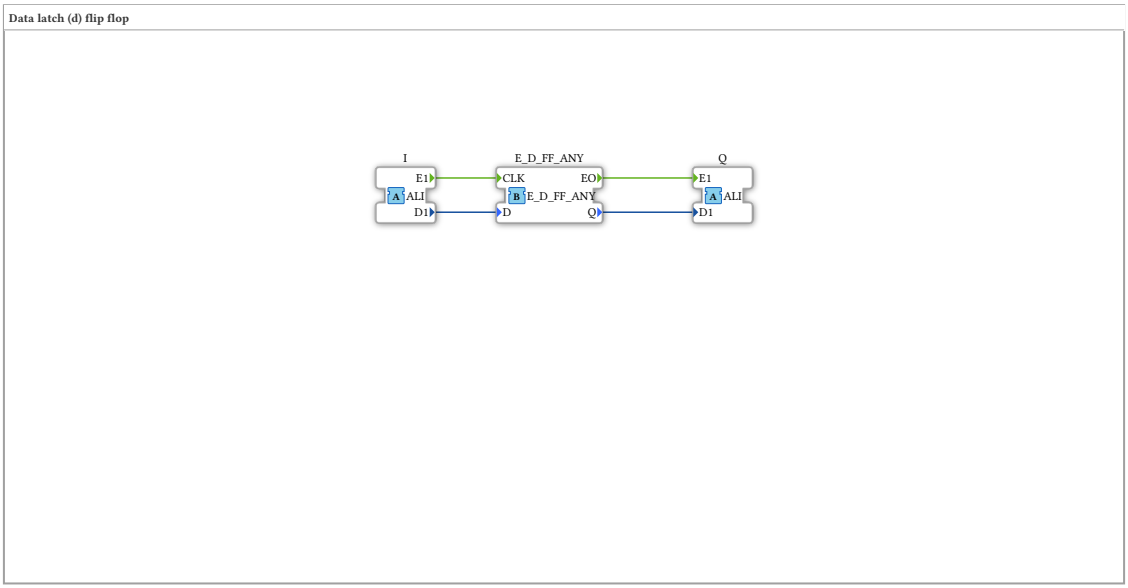
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

LINT

ALI_D_FF





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ALI_D_FF** realisiert ein taktgesteuertes D-Flip-Flop (Data Latch) über Adapter-Schnittstellen. Er speichert einen Datenwert bei einer steigenden Flanke eines Taktsignals und gibt diesen bis zur nächsten Taktflanke aus. Der Baustein eignet sich für synchrone Datenspeicherung in ereignisgesteuerten Automatisierungsumgebungen.

Schnittstellenstruktur

Ereignis-Eingänge

- **I.E1** (über Adapter I): Takteingang – bei einer steigenden Flanke wird das Datum übernommen.

Ereignis-Ausgänge

- **Q.E1** (über Adapter Q): Bestätigung der Datenübernahme – wird nach erfolgreicher Flankenerkennung und Wertzuweisung ausgegeben.

Daten-Eingänge

- **I.D1** (über Adapter I): Dateneingang – der zu speichernde Wert (beliebiger Datentyp, abhängig vom Adaptertyp `ALI`).

Daten-Ausgänge

- **Q.D1** (über Adapter Q): Datenausgang – der aktuell gespeicherte (gelatchte) Wert.

Adapter

Bezeichnung	Typ	Richtung	Beschreibung
I	`adapter::types::unidirectional::ALI`	Socket	Bietet Ereignis- und Dateneingang.
Q	`adapter::types::unidirectional::ALI`	Plug	Bietet Ereignis- und Datenausgang.

Funktionsweise

Der Baustein verwendet intern die Standard-Funktion `E_D_FF_ANY` aus der IEC 61499 Bibliothek.

- Bei einer steigenden Flanke am Ereigniseingang **I.E1** wird der aktuelle Wert von **I.D1** in den internen Zustand übernommen.
- Anschließend wird am Ausgang **Q.E1** ein Ereignis erzeugt, das signalisiert, dass der neue Wert an **Q.D1** anliegt.
- Der Datenausgang **Q.D1** hält den gespeicherten Wert so lange, bis eine neue steigende Flanke auftritt.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der FB besitzt keine direkten Ereignis-/Dateneingänge, sondern nutzt zwei Adapter (`I` und `Q`). Dies ermöglicht eine flexible Kapselung der Verbindungen und erleichtert die Wiederverwendung in komplexen Netzwerken.
- **Interne Umsetzung**: Der Kern des Bausteins ist der FB `E_D_FF_ANY`, der typunabhängig arbeitet. Der Datentyp wird durch die verwendeten Adapter bestimmt.
- **Ereignisgesteuert**: Die Zustandsänderung erfolgt ausschließlich bei Ereignissen, was die Integration in ereignisgesteuerte Abläufe erlaubt.

Zustandsübersicht

Der FB kennt zwei logische Zustände:

- **Zustand 0**: Ausgang `Q.D1` enthält den zuletzt gelatchten Wert, der z. B. logisch `0` sein kann.
- **Zustand 1**: Ausgang `Q.D1` enthält einen gültigen Wert ungleich `0` (sofern der Datentyp dies zulässt).

Ein Zustandswechsel erfolgt nur bei einer steigenden Flanke auf `I.E1`. Ohne Takt bleibt der Ausgang stabil.

Anwendungsszenarien

- **Datensynchronisation**: Puffern eines Sensorsignals bis zum nächsten Takt einer Steuerung.
- **Zustandsspeicher**: Halten eines Schaltzustands in Sequenzsteuerungen.
- **Rückkopplungsfreie Verzögerung**: Zwischenspeichern von Werten in Datenpfaden, um asynchrone Nebenläufigkeiten zu vermeiden.

Vergleich mit ähnlichen Bausteinen

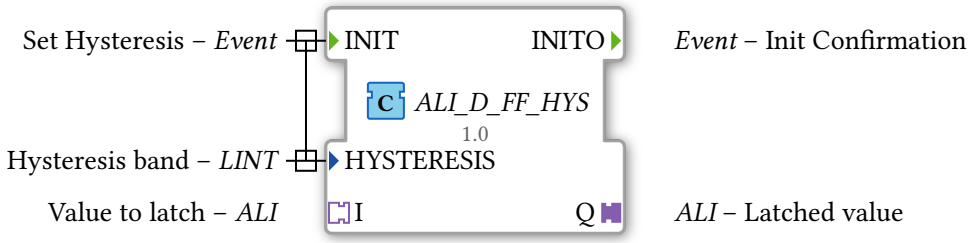
- **ALI_D_FF vs. einfacher D-FF (z. B. `E_D_FF_ANY`)**: Der ALI_D_FF kapselt Ereignis- und Datenverbindungen in Adaptern und ist dadurch modularer und leichter in vordefinierte Schnittstellen einbindbar.

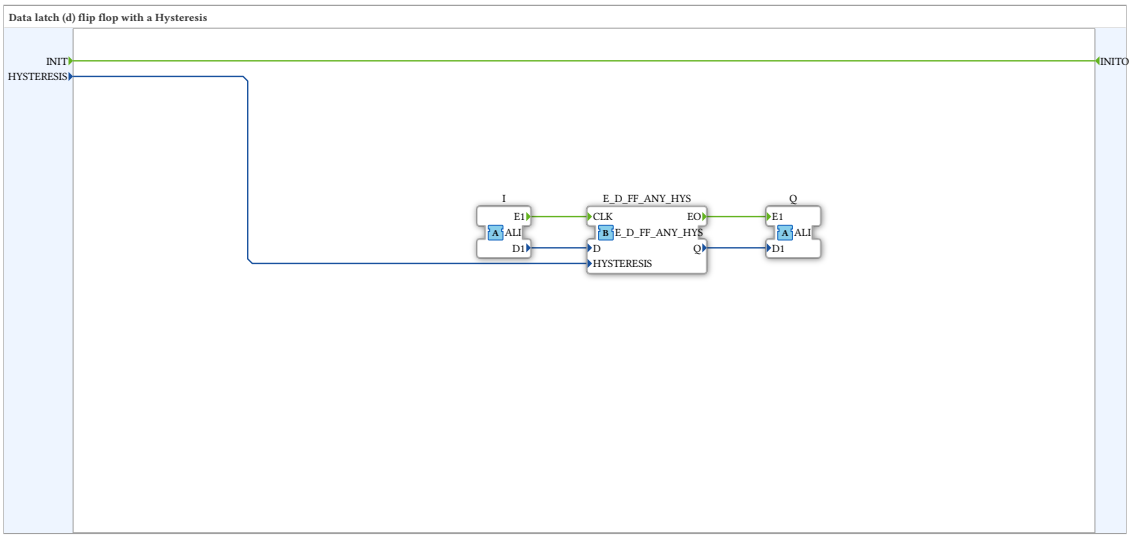
- **ALI_D_FF vs. RS-Flip-Flop**: Ein RS-FF erlaubt Setzen und Rücksetzen (asynchron), während der D-FF rein taktgesteuert arbeitet und keine direkten Set-/Reset-Eingänge bietet.
- **ALI_D_FF vs. T-Flip-Flop**: Der T-FF toggelt bei jedem Takt, der D-FF übernimmt einen externen Wert.

Fazit

Der **ALI_D_FF** bietet eine saubere, adapterbasierte Lösung für synchrone Datenspeicherung. Er kombiniert die bewährte Funktionalität eines D-Flip-Flops mit der Flexibilität von IEC 61499 Adaptern. Damit eignet er sich besonders für modulare, wiederverwendbare Automatisierungskomponenten in verteilten Systemen.

ALI_D_FF_HYS





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ALI_D_FF_HYS** realisiert ein Datenlatch (D-Flipflop) mit einer Hysteresefunktion. Er dient der stabilen und entprellten Übertragung eines binären Signals: Ein eingehendes Datensignal wird bei einer steigenden Taktflanke übernommen und ausgegeben, wobei eine konfigurierbare Hysterese verhindert, dass kleine Signaländerungen (Rauschen) zu unerwünschten Ausgangsänderungen führen. Der Baustein ist speziell für die Verwendung der unidirektionalen **ALI**-Adapterstruktur in IEC 61499-Umgebungen ausgelegt.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar
`INIT`	EInit	Setzt den Hysteresewert für die Schaltschwelle.

Ereignis-Ausgänge

Ereignis	Typ	Kommentar

INITO	EInit	Bestätigung der erfolgreichen Initialisierung.

Daten-Eingänge

Variable	Typ	Kommentar
HYSTERESIS	LINT	Hysterese-Bandbreite (z. B. als Ganzzahl)

Daten-Ausgänge

Keine direkten Datenausgänge – die Ausgabe erfolgt über den Adapter Q.

Adapter

Richtung	Name	Typ	Kommentar
Socket	I	adapter::types::unidirectional::ALI	Eingangswert (Daten + Takt)
Plug	Q	adapter::types::unidirectional::ALI	Ausgangswert (gelatchter Wert)

Der Adapter I liefert zwei Signale:

- **I.E1** – Taktsignal (Ereignis)
- **I.D1** – Dateneingang (z. B. BOOL oder numerisch)

Der Adapter Q stellt den gelatchten Wert als **Q.D1** zur Verfügung.

Funktionsweise

Der Baustein arbeitet intern mit einem weiteren FB **E_D_FF_ANY_HYS** (aus der Bibliothek logiBUS::signalprocessing::hysteresis), welcher die Kernlogik umsetzt. Die Funktionsweise im Überblick:

1. **Initialisierung**
   Beim Eintreffen des Ereignisses INIT wird der Wert von HYSTERESIS an den inneren Baustein übergeben und das Ereignis unverzögert als INITO bestätigt.
2. **Datenverarbeitung**
   Die Ereignis- und Datenverbindungen sind direkt zwischen den Adaptern und dem inneren Baustein geschaltet:
   - I.E1 ⟶ **CLK** (Taktsignal)
   - I.D1 ⟶ **D** (Eingangsdaten)
   - **Q** ⟶ Q.D1 (gelatchtes Ausgangssignal)
3. **Hystereseverhalten**
   Bei jedem Takt (CLK) wird der aktuelle Datenwert D ausgewertet. Liegt D innerhalb des Hysteresebandes um den aktuellen Ausgangswert Q, so ändert sich der Ausgang nicht. Überschreitet D die obere bzw. untere Schwelle, wird der neue Wert übernommen. Dies unterdrückt Störungen und sorgt für einen sauberen, entprellten Signalverlauf.

Technische Besonderheiten

- **Reine Adapterkommunikation**: Der FB verwendet ausschließlich unidirektionale Adapter (ALI) für die Ereignis- und Datenübergabe. Dadurch ist er in modularen, adapterbasierten Architekturen einfach einsetzbar.
- **Indirekte Hystereseimplementierung**: Die Hysterese wird nicht direkt im FB-Body, sondern durch den internen Baustein E_D_FF_ANY_HYS realisiert. Dies ermöglicht eine flexible Wiederverwendung der Logik.
- **Sofortige INIT-Bestätigung**: Das INIT-Ereignis wird durch eine direkte Verbindung (INIT → INITO) unmittelbar quittiert, ohne auf die Hystereselogik zu warten.
- **Typ des Hystereseparameters**: HYSTERESIS ist als LINT definiert, erlaubt also große ganzzahlige Werte.

Zustandsübersicht

Der Baustein besitzt selbst keine sichtbaren Zustände, da alle Zustandslogik im internen FB gekapselt ist. Verhalten des D-Flipflops mit Hysterese:

- **Ruhezustand**: Ausgangswerte bleiben stabil, solange kein Takt anliegt.
- **Taktflanke (steigend)**:
  - Wenn D > Q + HYSTERESIS ⟶ Q = D (Setzen)
  - Wenn D < Q - HYSTERESIS ⟶ Q = D (Rücksetzen)
  - Sonst ⟶ Q unverändert (Hysterese-Bereich)

Anwendungsszenarien

Typische Einsatzfelder dieses Bausteins sind:

- **Entprellung von Schaltersignalen** in der Automatisierungstechnik (z. B. Endschalter, Näherungssensoren).
- **Signalaufbereitung in rauer Umgebung** (Agrartechnik, Maschinenbau) – die Hysterese verhindert Oszillationen durch Vibrationen oder Rauschen.
- **Grundbaustein für komplexere Logik** mit Adapter-basierter Datenweitergabe, z. B. als Eingangsstufe eines Zustandsautomaten.

Vergleich mit ähnlichen Bausteinen

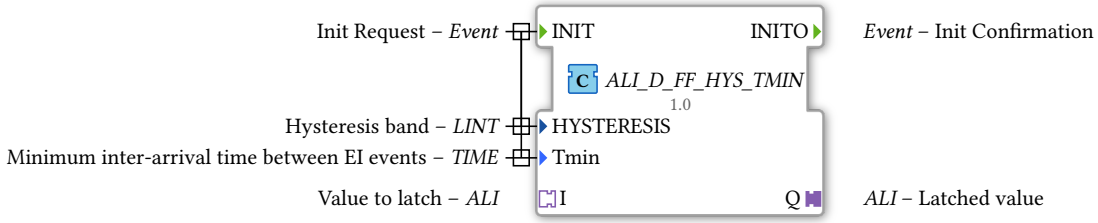
Baustein	Merkmal
ALI_D_FF (ohne Hyst.)	Einfaches D-FF ohne Rauschunterdrückung
ALI_D_FF_HYS (dieser)	D-FF mit parametrierbarer Hysterese zur Störungsfilterung
Schmitt-Trigger-FB	Ähnliche Funktion, jedoch oft mit festen Schwellen
E_D_FF_ANY_HYS	Interne Logik, allgemeiner Typ (kein Adapter-Format)

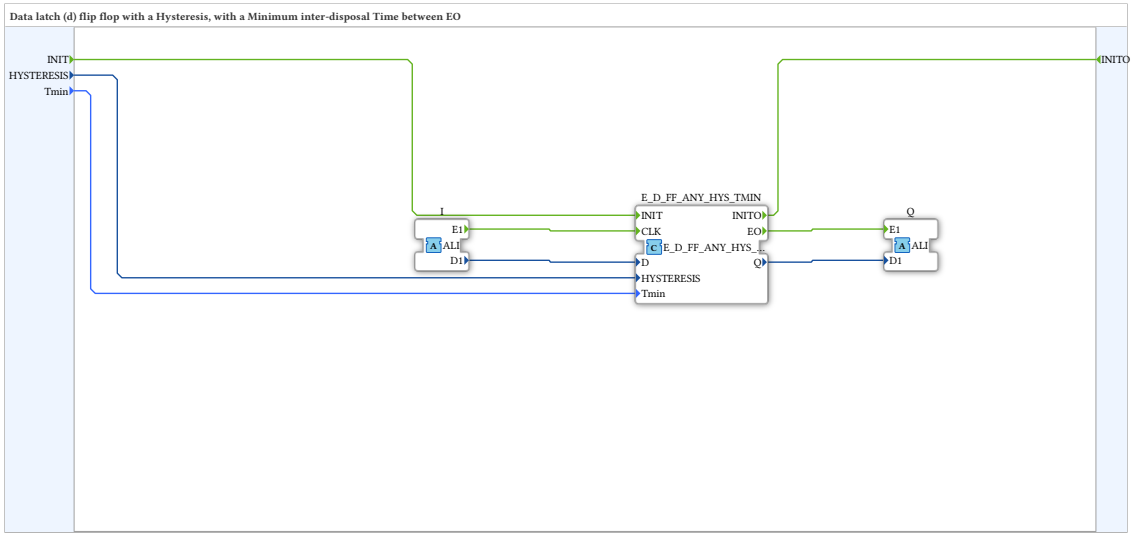
Der Vorteil von **ALI_D_FF_HYS** liegt in der direkten Adapteranbindung, die eine nahtlose Integration in bestehende IEC 61499-Adapter-Netzwerke ermöglicht.

Fazit

Der Funktionsblock **ALI_D_FF_HYS** bietet eine bewährte Kombination aus D-Flipflop und einstellbarer Hysterese, verpackt in einem standardkonformen IEC 61499-Adapter-Interface. Er eignet sich besonders für Anwendungen, bei denen Signale unter realen Bedingungen (Rauschen, Prellen) zuverlässig verarbeitet werden müssen. Durch die klare Trennung von Initialisierung und Datenverarbeitung sowie die Nutzung eines bewährten Unterbausteins ist er robust, wartbar und flexibel einsetzbar.

ALI_D_FF_HYS_TMIN





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ALI_D_FF_HYS_TMIN** realisiert ein datengesteuertes D-Flip-Flop mit einer einstellbaren Hysteresebreite (Schwellwertband) und einer minimalen Wartezeit zwischen zwei aufeinanderfolgenden Ausgangsereignissen. Er dient zur Entprellung und Stabilisierung von Signalsprüngen in der industriellen Automatisierungstechnik, insbesondere wenn ein Signal erst dann als gültig übernommen werden soll, wenn es einen definierten Toleranzbereich verlassen hat und eine Mindestzeit seit der letzten gültigen Flanke vergangen ist.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Initialisierungsanforderung – setzt die Hysteresebreite und die Mindestzeit fest.

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Bestätigung der erfolgreichen Initialisierung.

Daten-Eingänge

Name	Typ	Kommentar
HYSTERESIS	LINT	Hystereseband als ganzzahliger Wert (LINT).
Tmin	TIME	Minimale Zeit zwischen zwei Ausgangsereignissen (Inter-Disposal Time).

Daten-Ausgänge

Keine direkten Datenausgänge vorhanden.
Der Baustein gibt seinen Zustand über den Adapter **Q** aus (siehe Adapter).

Adapter

Typ	Name	Richtung	Kommentar
`adapter::types::unidirectional::ALI`	I	Socket	Eingangsadapter – liefert das zu übernehmende Signal (Ereignis und Datenwert).

`adapter::types::unidirectional::ALI`	Q	Plug	Ausgangsadapter – gibt den zuletzt gültigen Datenwert sowie ein Ereignis aus.

Der Adaptertyp `ALI` umfasst ein Ereignis (E1) und einen Datenwert (D1) und dient der unidirektionalen Kommunikation zwischen Funktionsblöcken.

Funktionsweise

Der Baustein **ALI_D_FF_HYS_TMIN** ist als Komposition realisiert und nutzt intern den FB `logiBUS::signalprocessing::hysteresis::E_D_FF_ANY_HYS_TMIN`. Seine Arbeitsweise lässt sich wie folgt beschreiben:

1. **Initialisierung** – Über das Ereignis **INIT** werden die Parameter **HYSTERESIS** und **Tmin** gesetzt. Nach erfolgreicher Initialisierung wird **INITO** ausgelöst.
2. **Signalübernahme** – Jedes Ereignis am Socket **I** (Ereignis **I.E1**) wird als Taktflanke interpretiert. Der zugehörige Datenwert **I.D1** wird nur dann in den internen Speicher übernommen, wenn er sich außerhalb des Hysteresebandes um den aktuell gespeicherten Wert befindet (d. h. die Änderung überschreitet die Hysterese).
3. **Mindestzeitsteuerung** – Nach der Übernahme eines gültigen Wertes wird die Ausgabe über den Plug **Q** (Ereignis **Q.E1** und Wert **Q.D1**) erst dann freigegeben, wenn seit dem letzten Ausgangsereignis mindestens **Tmin** verstrichen ist. Dadurch werden unerwünschte schnelle Signalwechsel unterdrückt.
4. **Ausgabe** – Sobald die Mindestzeit eingehalten wurde, erfolgt die Ausgabe des neuen Wertes über den Ausgangsadapter **Q**. Der Baustein wartet anschließend auf das nächste Eingangsereignis.

Technische Besonderheiten

- **Hysterese-Datenbreite** – Die Hysterese wird als LINT (64-Bit-Ganzzahl) definiert; dies ermöglicht die Verarbeitung sehr großer Wertebereiche.
- **Zeitparameter** – Die minimale Inter-Disposal-Zeit **Tmin** wird als IEC-61499-`TIME` angegeben (z. B. `T#10ms`).
- **Adapterbasierte Kopplung** – Die Verwendung der Adapter `ALI` erlaubt eine lose Kopplung zwischen signalerzeugenden und signalverarbeitenden Bausteinen. Der Socket **I** empfängt die Daten, der Plug **Q** stellt sie bereit.
- **Typ der Abhängigkeit** – Der Baustein basiert auf dem internen FB `E_D_FF_ANY_HYS_TMIN`, der die eigentliche Hysterese- und Zeitlogik implementiert. Der äußere Baustein kapselt lediglich die Adapter-Schnittstelle.

Zustandsübersicht

Der Baustein durchläuft folgende Zustände:

1. **INIT** – Warten auf Initialisierungsparameter.
2. **IDLE** – Bereit; wartet auf ein Ereignis am Socket **I**.
3. **SAMPLE** – Ein Ereignis ist eingetroffen; der Datenwert wird auf Überschreitung der Hysterese geprüft.
   - *Hysterese verletzt:* Der Wert wird ignoriert, zurück zu **IDLE**.
   - *Hysterese erfüllt:* Wechsel in Zustand **WAIT_TMIN**.
4. **WAIT_TMIN** – Der neue Wert ist gespeichert, die Freigabe wird um **Tmin** verzögert.
   - Nach Ablauf der Zeit erfolgt die Ausgabe über **Q** und Rücksprung nach **IDLE**.

Eine formale Zustandsmaschine ist nicht über die XML-Schnittstelle sichtbar, ergibt sich aber aus dem Verhalten des internen Flip-Flops.

Anwendungsszenarien

- **Sensorwertglättung** – Ein digitaler Sensor (z. B. Drehgeber) liefert Werte, die aufgrund von Vibrationen oder Rauschen kurzzeitig schwanken. Die Hysterese verhindert, dass kleine Schwankungen als gültige Änderung gewertet werden.
- **Entprellung von Tastern** – Bei mechanischen Kontakten wird durch **Tmin** sichergestellt, dass Prellvorgänge innerhalb einer bestimmten Zeit nicht zu mehrfachen Ausgangsereignissen führen.
- **Messwertübernahme mit Totzeit** – In Dosier- oder Positioniersystemen muss die Mindestzeit zwischen zwei gültigen Messwerten eingehalten werden, um die Systemdynamik nicht zu überlasten.

Vergleich mit ähnlichen Bausteinen

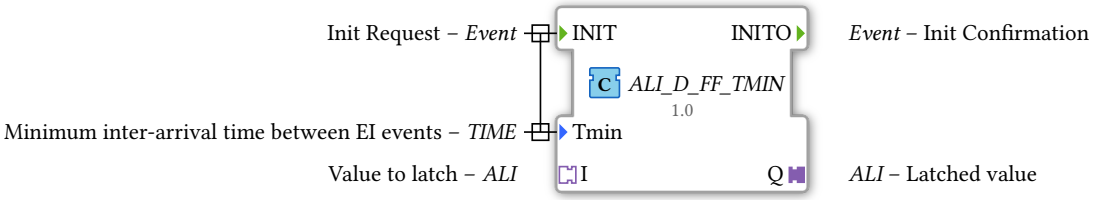
Baustein	Hysterese	Mindestzeit	Adapterschnittstelle
`E_D_FF` (einfach)	Nein	Nein	Nein (direkte I/O)
`E_D_FF_HYS`	Ja	Nein	Nein
`ALI_D_FF_HYS_TMIN`	Ja	Ja	Ja (Adapternutzung)

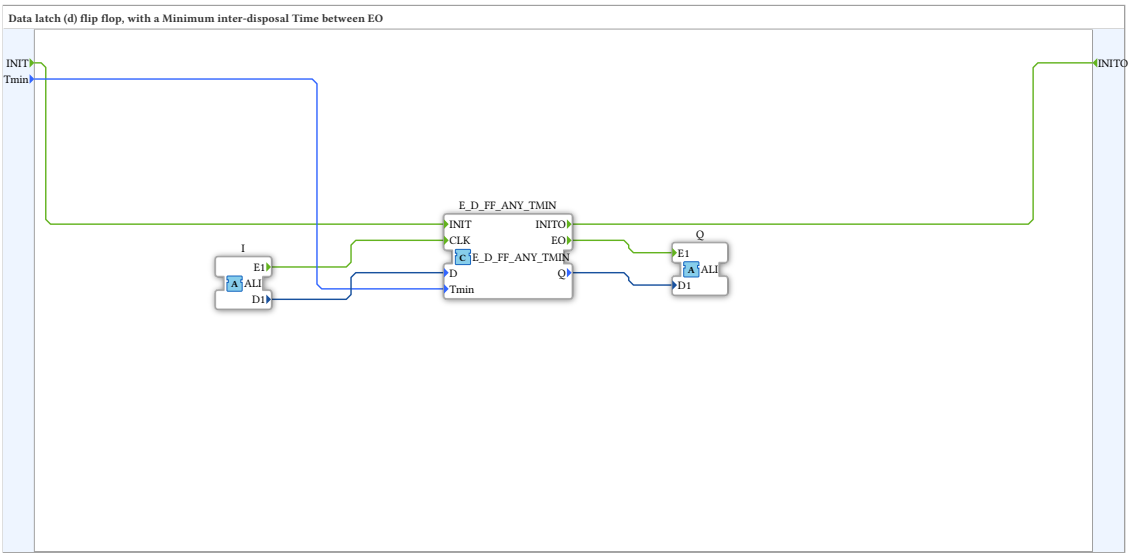
Der hier beschriebene Baustein vereint die Vorteile von Hysterese und zeitlichem Mindestabstand und ist durch seine Adapter-Schnittstelle besonders gut für eine modulare, ereignisgesteuerte Architektur nach IEC 61499 geeignet.

Fazit

Der Baustein **ALI_D_FF_HYS_TMIN** stellt eine robuste Lösung zur stabilen Übernahme von Signalen in rauer Umgebung dar. Durch die Kombination von Hysterese und minimaler Ausgabezeit werden sowohl Amplitudenschwankungen als auch zeitliche Preller effektiv unterdrückt. Die adapterbasierte Ein- und Ausgabe macht ihn flexibel einsetzbar in komplexen Automatisierungsnetzwerken.

ALI_D_FF_TMIN





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ALI_D_FF_TMIN** realisiert einen Datenspeicher (D-Flip-Flop) mit einem Adapter-gestützten Ein- und Ausgang und einer konfigurierbaren Mindestverweilzeit zwischen den Ausgangsereignissen. Er dient dazu, einen anliegenden Datenwert bei einer steigenden Flanke am Eingang zu übernehmen und nach Ablauf einer minimalen Zeit freizugeben, um zyklische Bedingungen oder Zeitauflösungsprobleme in der Kommunikation zu vermeiden.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Mit Variablen	Kommentar

INIT	EInit	Tmin	Initialisierungsanforderung, setzt die minimale Haltezeit

Ereignis-Ausgänge

Ereignis	Typ	Mit Variablen	Kommentar
INITO	EInit	–	Bestätigung der Initialisierung

Daten-Eingänge

Variable	Typ	Kommentar
Tmin	TIME	Minimale Zeit zwischen zwei Ausgangsereignissen (EO)

Daten-Ausgänge

Keine direkten Datenausgänge. Der latched Wert wird über den **Q-Adapter** bereitgestellt.

Adapter

Bezeichnung	Typ	Richtung	Kommentar
I	ALI (unidirectional)	Socket (Eingang)	Wert, der übernommen werden soll
Q	ALI (unidirectional)	Plug (Ausgang)	Gespeicherter Wert

Funktionsweise

Der **ALI_D_FF_TMIN** kapselt einen internen FB vom Typ `E_D_FF_ANY_TMIN`. Dessen Funktionsweise:

- Bei einem Ereignis am Eingangsadapter **I.E1** (steigende Flanke) wird der Datenwert **I.D1** übernommen.
- Das Ausgangsereignis **EO** wird jedoch erst ausgelöst, wenn seit dem letzten **EO** mindestens die Zeit **Tmin** vergangen ist.
- Die Initialisierung (INIT) setzt den internen Zeitgeber und bereitet den Baustein vor.

Der Baustein arbeitet asynchron, die Zeitsteuerung erfolgt über den internen FB.

Technische Besonderheiten

- Die Logik ist vollständig in den internen FB `E_D_FF_ANY_TMIN` ausgelagert, der einen eigenen Zustandsautomaten und Zeitgeber besitzt.
- Die **Tmin**-Zeit wird nur einmal bei INIT gesetzt und gilt für die gesamte Lebensdauer.
- Adapter **I** (Eingang) und **Q** (Ausgang) sind unidirektional vom Typ `ALI`, sie übertragen sowohl Ereignis als auch Daten.
- Der Baustein implementiert kein eigenes Verhalten auf der obersten Ebene, sondern delegiert alle Funktionen.

Zustandsübersicht

Da der FB intern `E_D_FF_ANY_TMIN` verwendet, besitzt er implizit folgende Zustände:

1. **Initialisiert**: Nach INIT, bereit für erste Flanke.
2. **Flanke empfangen**: Wert wurde gespeichert, EO wird nach **Tmin** gesendet.
3. **Warten auf Mindestzeit**: Kein neues EO möglich, bis Tmin abgelaufen.
4. **Bereit**: Nach Ablauf der Wartezeit kann das nächste EO ausgelöst werden.

Der genaue Zustandsautomat liegt jedoch im internen FB und ist hier nicht abgebildet.

Anwendungsszenarien

- **Datensicherung in zeitgesteuerten Bussystemen**: Pufferung von Werten bei einem Takt, die nur in definierten Abständen weitergegeben werden dürfen.
- **Entprellung von Signalen**: Verhindert, dass schnelles Ein- und Ausschalten zu vielen Ereignissen auf der Logikebene führt.
- **Sequenzielle Verarbeitung mit Mindestabstand**: Sicherstellung, dass nachfolgende Bausteine genügend Zeit zur Verarbeitung haben.

Vergleich mit ähnlichen Bausteinen

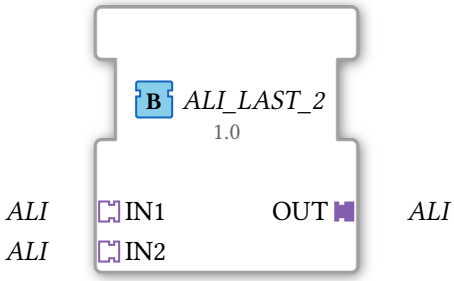
Baustein	Unterschied
`ALI_D_FF` (ohne TMIN)	Keine Mindestzeit zwischen Ausgangsereignissen, sofortige Weiterleitung bei jeder Flanke.
`E_D_FF` (Standard-61499)	Ereignis-Ein- und Ausgänge direkt, ohne Adapter und ohne Zeitsteuerung.
`E_D_FF_ANY_TMIN`	Gleiche Funktionalität, aber mit eigenen Daten- und Ereignis-Ports, nicht über Adapter.

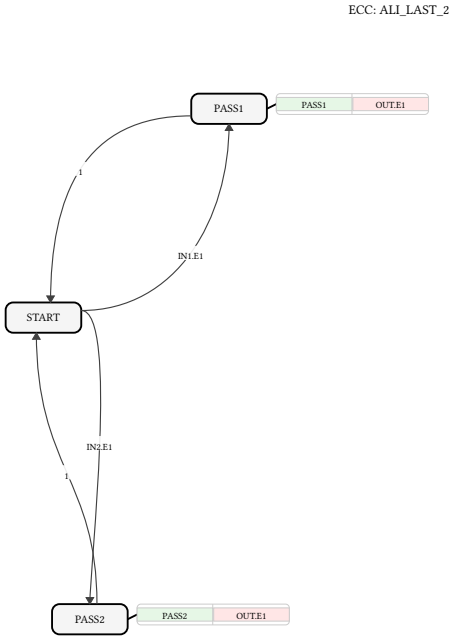
Der **ALI_D_FF_TMIN** erweitert die Adapter-Schnittstelle um eine komfortable Kapselung und Fügt die Zeitsteuerung hinzu, die bei einfachen Flip-Flops fehlt.

Fazit

Der **ALI_D_FF_TMIN** ist ein nützlicher Baustein für Anwendungen, die eine verzögerte oder getaktete Weitergabe von Daten über Adapter erfordern. Durch die konfigurierbare Mindestzeit zwischen den Ausgangsereignissen eignet er sich besonders für zeitkritische oder entprellte Kommunikationsstrecken. Die interne Implementierung über einen bewährten FB sorgt für Robustheit und Wiederverwendbarkeit.

ALI_LAST_2





Einleitung

Der ALI_LAST_2 Funktionsblock führt zwei gleichartige LINT-Adapter-Signale (**IN1**, **IN2**) auf einen gemeinsamen Ausgang (**OUT**) zusammen, nach dem Prinzip **Last Writer Wins**: welcher der beiden

Sockets zuletzt ein Ereignis geschrieben hat, gewinnt, und sein Datenwert wird sofort und unverändert an OUT durchgereicht. Es handelt sich ausdrücklich nicht um ein naives Zusammenführen (z. B. Mittelwert oder ODER-Verknüpfung), sondern um eine reine Arbitrierung nach zeitlicher Reihenfolge.

Schnittstellenstruktur

Adapter

Eingangsadapter (Sockets):

- **IN1**: LINT-Adapter (unidirectional) - erste Signalquelle
- **IN2**: LINT-Adapter (unidirectional) - zweite Signalquelle

Ausgangsadapter (Plug):

- **OUT**: LINT-Adapter (unidirectional) - zusammengeführtes Signal

Funktionsweise

Der Baustein ist als einfacher Basic-FB mit einer 3-Zustands-ECC umgesetzt (START, PASS1, PASS2). Trifft an IN1 ein Ereignis ein, wechselt die ECC nach PASS1: der aktuelle Datenwert von IN1 wird nach OUT kopiert und das Adapter-Ereignis an OUT gesendet. Trifft an IN2 ein Ereignis ein, geschieht dasselbe symmetrisch über PASS2. Nach jeder Durchreichung kehrt die ECC sofort wieder nach START zurück und ist bereit für das nächste Ereignis von IN1 oder IN2.

Technische Besonderheiten

- Last-Writer-Wins-Semantik statt naiver Datenverknüpfung
- Einfacher Basic-FB (ECC mit 3 Zuständen), kein zusätzlicher Zwischenspeicher nötig
- Keine Signalverzögerung: der Wert wird im selben Zyklus durchgereicht, in dem das auslösende Ereignis eintrifft
- Beide Eingänge sind gleichberechtigt - es gibt keine feste Priorität außer der zeitlichen Reihenfolge

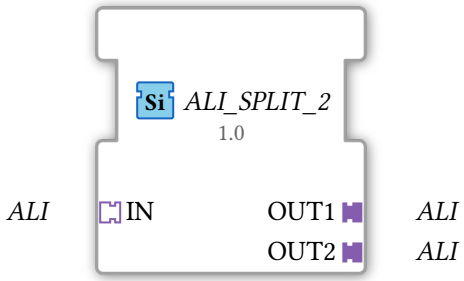
Anwendungsszenarien

- Zusammenführen zweier gleichwertiger LINT-Signalquellen zu einem einzigen nachgeschalteten Adapterpfad (z. B. zwei redundante Sensoren, oder ein manueller und ein automatischer Sollwertgeber)
- Arbitrierung zwischen zwei Bedienstellen, die denselben LINT-Wert schreiben dürfen
- Vereinfachung von Netzwerken, die sonst ein `F_SEL`/`E_RS`-Konstrukt inline verdrahten müssten

⚖️ Vergleich mit ähnlichen Bausteinen

ALI_LAST_2 ist die Umkehrung von `ALI_SPLIT_2`: ALI_SPLIT_2 verteilt 1 Eingang auf 2 Ausgänge, ALI_LAST_2 führt 2 Eingänge auf 1 Ausgang zusammen. Für Mischbetrieb zwischen einem Daten- und einem reinen Ereignisadapter (z. B. AX + AE) ist stattdessen `AX_AE_MERGE` gedacht, das nur das Ereignis mischt, ohne den Datenwert zu verändern.

ALI_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ALI_SPLIT_2** dient als generischer Verteiler für einen unidirektionalen ALI-Adapter (Application Layer Interface). Er kopiert alle eingehenden Daten und Ereignisse von einem einzelnen Eingangs-Socket auf zwei Ausgangs-Plugs. Dadurch kann ein Signal oder eine Datenquelle an zwei unabhängige Empfänger weitergegeben werden.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden. Die Datenweitergabe erfolgt rein über die Adapter-Schnittstelle.

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden. Die Ereignissteuerung wird über die Adapter-Verbindungen abgewickelt.

Daten-Eingänge
Keine dedizierten Daten-Eingänge. Alle Daten werden über den eingehenden Adapter **IN** bereitgestellt.

Daten-Ausgänge
Keine dedizierten Daten-Ausgänge. Die verteilten Daten werden über die beiden ausgehenden Adapter **OUT1** und **OUT2** bereitgestellt.

Adapter

Typ	Name	Richtung	Beschreibung
Socket	`IN`	Eingang	Unidirektionaler ALI-Adapter, der die eingehenden Daten/Ereignisse empfängt.
Plug	`OUT1`	Ausgang	Erster unidirektionaler ALI-Adapter, an den die Daten/Ereignisse weitergeleitet werden.
Plug	`OUT2`	Ausgang	Zweiter unidirektionaler ALI-Adapter, an den die Daten/Ereignisse weitergeleitet werden.

Alle Adapter sind vom Typ `adapter::types::unidirectional::ALI`.

Funktionsweise
Der Baustein realisiert eine 1-zu-2-Verteilung (Split) auf der Adapter-Ebene. Sobald Daten oder Ereignisse am Socket **IN** anliegen, werden diese unverändert und gleichzeitig an beide Plugs **OUT1** und **OUT2** weitergegeben. Es erfolgt keine Pufferung oder Verarbeitung der Daten – der Baustein arbeitet als reiner Signalverteiler.

Technische Besonderheiten
- **Generischer Baustein**: Über das Attribut `eclipse4diac::core::GenericClassName` wird der Baustein als generische Instanz `GEN_ALI_SPLIT` markiert. Er kann daher mit unterschiedlichen ALI-Typen oder -Parametern verwendet werden, solange die zugrundeliegende Adapter-Definition kompatibel ist.
- **Keine Zustandshaltung**: Der Baustein besitzt keine internen Zustände oder Variablen – die Funktionalität ist rein kombinatorisch.
- **Unidirektionalität**: Die Adapter sind als unidirektional definiert, d.h. die Daten fließen nur vom Eingang zu den Ausgängen; Rückwärtskommunikation ist nicht vorgesehen.

Zustandsübersicht
Da der Baustein keine Ereignis-Eingänge/Ausgänge oder internen Zustände besitzt, existiert keine Zustandsmaschine. Die Datenweitergabe erfolgt kontinuierlich und ohne Verzögerung.

Anwendungsszenarien
- **Verteilen eines Sensorsignals**: Ein analoger oder digitaler ALI-Sensor wird an zwei Steuerungseinheiten angeschlossen, die unabhängig voneinander auf die Messdaten zugreifen müssen.
- **Parallele Verarbeitung**: Ein Datenstrom soll gleichzeitig von zwei verschiedenen Funktionsblöcken verarbeitet werden, ohne dass eine manuelle Verzweigung in der Applikation nötig ist.
- **Redundanz**: Eine Anwendung erfordert die doppelte Ausführung einer ALI-Kommunikation, um Ausfallsicherheit zu erhöhen.

Vergleich mit ähnlichen Bausteinen

Baustein	Funktion	Unterschied

ALI_SPLIT_2	Verteilt einen ALI-Eingang auf zwei ALI-Ausgänge.	–
ALI_SPLIT_3 (hypothetisch)	Verteilt auf drei Ausgänge.	Höhere Ausgangszahl.
ALI_MERGE_2	Fasst zwei ALI-Eingänge zu einem ALI-Ausgang zusammen.	Gegenteilige Funktion (Join).
ALI_FILTER	Leitet nur bestimmte Daten/Ereignisse weiter.	Enthält Filterlogik.

- `ALI_SPLIT_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

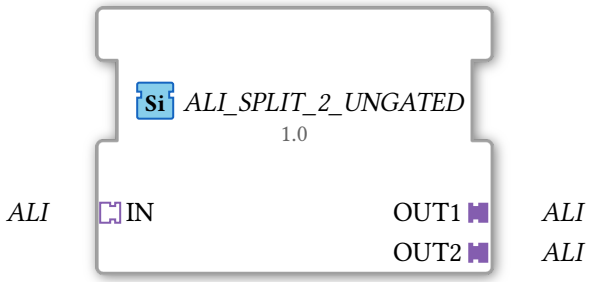
ALI_SPLIT_2 ist ein einfacher, aber essenzieller Funktionsblock für die Verteilung unidirektionaler ALI-Schnittstellen. Seine generische Natur und die schlanke Implementierung machen ihn zu einem universellen Werkzeug in der IEC 61499-Komponentenentwicklung, wenn ein Signal oder Datenstrom an mehrere nachfolgende Blöcke weitergegeben werden muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ALI_SPLIT_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALI_SPLIT_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ALI_SPLIT_2_UNGATED** dient als generischer Verteiler für einen unidirektionalen ALI-Adapter (Application Layer Interface). Er kopiert alle eingehenden Daten und Ereignisse von einem einzelnen Eingangs-Socket auf zwei Ausgangs-Plugs. Dadurch kann ein Signal oder eine Datenquelle an zwei unabhängige Empfänger weitergegeben werden.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden. Die Datenweitergabe erfolgt rein über die Adapter-Schnittstelle.

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden. Die Ereignissteuerung wird über die Adapter-Verbindungen abgewickelt.

Daten-Eingänge
Keine dedizierten Daten-Eingänge. Alle Daten werden über den eingehenden Adapter **IN** bereitgestellt.

Daten-Ausgänge
Keine dedizierten Daten-Ausgänge. Die verteilten Daten werden über die beiden ausgehenden Adapter **OUT1** und **OUT2** bereitgestellt.

Adapter

Typ	Name	Richtung	Beschreibung
Socket	`IN`	Eingang	Unidirektionaler ALI-Adapter, der die eingehenden Daten/Ereignisse empfängt.
Plug	`OUT1`	Ausgang	Erster unidirektionaler ALI-Adapter, an den die Daten/Ereignisse weitergeleitet werden.
Plug	`OUT2`	Ausgang	Zweiter unidirektionaler ALI-Adapter, an den die Daten/Ereignisse weitergeleitet werden.

Alle Adapter sind vom Typ `adapter::types::unidirectional::ALI`.

Funktionsweise
Der Baustein realisiert eine 1-zu-2-Verteilung (Split) auf der Adapter-Ebene. Sobald Daten oder Ereignisse am Socket **IN** anliegen, werden diese unverändert und gleichzeitig an beide Plugs **OUT1** und **OUT2** weitergegeben. Es erfolgt keine Pufferung oder Verarbeitung der Daten – der Baustein arbeitet als reiner Signalverteiler.

Technische Besonderheiten
- **Generischer Baustein**: Über das Attribut `eclipse4diac::core::GenericClassName` wird der Baustein als generische Instanz `GEN_ALI_SPLIT` markiert. Er kann daher mit unterschiedlichen ALI-Typen oder -Parametern verwendet werden, solange die zugrundeliegende Adapter-Definition kompatibel ist.
- **Keine Zustandshaltung**: Der Baustein besitzt keine internen Zustände oder Variablen – die Funktionalität ist rein kombinatorisch.
- **Unidirektionalität**: Die Adapter sind als unidirektional definiert, d.h. die Daten fließen nur vom Eingang zu den Ausgängen; Rückwärtskommunikation ist nicht vorgesehen.

Zustandsübersicht
Da der Baustein keine Ereignis-Eingänge/Ausgänge oder internen Zustände besitzt, existiert keine Zustandsmaschine. Die Datenweitergabe erfolgt kontinuierlich und ohne Verzögerung.

Anwendungsszenarien
- **Verteilen eines Sensorsignals**: Ein analoger oder digitaler ALI-Sensor wird an zwei Steuerungseinheiten angeschlossen, die unabhängig voneinander auf die Messdaten zugreifen müssen.
- **Parallele Verarbeitung**: Ein Datenstrom soll gleichzeitig von zwei verschiedenen Funktionsblöcken verarbeitet werden, ohne dass eine manuelle Verzweigung in der Applikation nötig ist.
- **Redundanz**: Eine Anwendung erfordert die doppelte Ausführung einer ALI-Kommunikation, um Ausfallsicherheit zu erhöhen.

Vergleich mit ähnlichen Bausteinen

Baustein	Funktion	Unterschied

ALI_SPLIT_2_UNGATED	Verteilt einen ALI-Eingang auf zwei ALI-Ausgänge.	–
ALI_SPLIT_3 (hypothetisch)	Verteilt auf drei Ausgänge.	Höhere Ausgangszahl.
ALI_MERGE_2	Fasst zwei ALI-Eingänge zu einem ALI-Ausgang zusammen.	Gegenteilige Funktion (Join).
ALI_FILTER	Leitet nur bestimmte Daten/Ereignisse weiter.	Enthält Filterlogik.

- `ALI_SPLIT_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

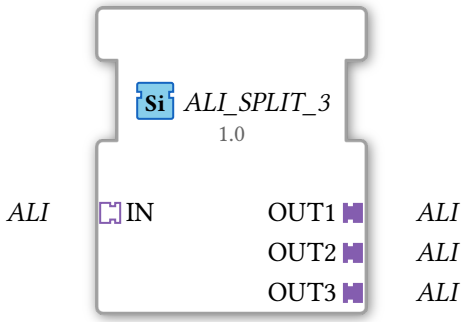
Fazit

ALI_SPLIT_2_UNGATED ist ein einfacher, aber essenzieller Funktionsblock für die Verteilung unidirektionaler ALI-Schnittstellen. Seine generische Natur und die schlanke Implementierung machen ihn zu einem universellen Werkzeug in der IEC 61499-Komponentenentwicklung, wenn ein Signal oder Datenstrom an mehrere nachfolgende Blöcke weitergegeben werden muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ALI_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ALI_SPLIT_3** dient dazu, einen eingehenden unidirektionalen Adapter vom Typ **ALI** (Application Layer Interface) auf drei identische Ausgangsadapter desselben Typs zu verteilen. Er ist als generischer Baustein konzipiert und ermöglicht eine Signalvervielfachung ohne Datenveränderung.

Schnittstellenstruktur

Ereignis-Eingänge

– Keine –

Ereignis-Ausgänge

– Keine –

Daten-Eingänge

– Keine (die gesamte Kommunikation erfolgt über die Adapter-Schnittstellen)

Daten-Ausgänge

– Keine (die gesamte Kommunikation erfolgt über die Adapter-Schnittstellen)

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	`IN`	`adapter::types::unidirectional::ALI`	Empfängt das eingehende ALI-Signal.
Plug (Ausgang)	`OUT1`	`adapter::types::unidirectional::ALI`	Erste Ausgangskopie des Signals.
Plug (Ausgang)	`OUT2`	`adapter::types::unidirectional::ALI`	Zweite Ausgangskopie des Signals.
Plug (Ausgang)	`OUT3`	`adapter::types::unidirectional::ALI`	Dritte Ausgangskopie des Signals.

Hinweis: Der Adaptertyp `ALI` ist ein unidirektionaler Interface-Adapter, der in der Regel Daten- und Ereignisfluss in einer Richtung kapselt. Die exakte interne Struktur ist anwendungsspezifisch definiert.

Funktionsweise

Der FB führt eine reine 1:3-Verteilung des am Socket `IN` anliegenden ALI-Adapters durch. Alle am Eingang empfangenen Informationen (z. B. Datenwerte, Ereignisse oder Zustände) werden ohne Verzögerung oder Transformation parallel an die drei Ausgänge `OUT1`, `OUT2` und `OUT3` weitergeleitet. Es findet keine Pufferung oder Logik statt – der Baustein arbeitet rein datenflusssynchron.

Technische Besonderheiten

- **Generischer Typ:** Der Baustein ist als generischer Funktionsblock deklariert (`eclipse4diac::core::GenericClassName = 'GEN_ALI_SPLIT'`). Dadurch kann der eigentliche Adaptertyp zur Konfigurationszeit festgelegt werden, sofern das zugrundeliegende System dies unterstützt.
- **Keine Eigendynamik:** Der FB besitzt keine eigenen Ereignis- oder Daten-Ein-/Ausgänge außerhalb der Adapter. Die gesamte Kommunikation läuft ausschließlich über die Adapter-Schnittstellen.
- **Geringe Latenz:** Durch den Verzicht auf interne Verarbeitung ist die Signalverteilung nahezu verzögerungsfrei.

Zustandsübersicht

Der Baustein besitzt keine Zustandsautomaten oder Zustandsspeicher. Sein Verhalten ist deterministisch und rein kombinatorisch: Der aktuelle Eingangszustand wird unmittelbar auf alle Ausgänge abgebildet.

Anwendungsszenarien

- **Signalvervielfachung in Steuerungssystemen:** Wenn ein ALI-Signal (z. B. ein Sensordatenstrom) mehrfach von verschiedenen nachgelagerten Bausteinen benötigt wird, kann ALI_SPLIT_3 die Verteilung übernehmen.
- **Test- und Simulationsumgebungen:** Mehrere Komponenten sollen mit demselben Eingangssignal versorgt werden, um parallele Reaktionen zu beobachten.
- **Redundanz:** Ein Signal wird auf drei Pfade aufgeteilt, die unabhängig voneinander weiterverarbeitet werden (z. B. für Vergleich oder Fehlererkennung).

Vergleich mit ähnlichen Bausteinen

- **ALI_SPLIT_2:** Verteilt ein ALI-Signal auf zwei statt drei Ausgänge. ALI_SPLIT_3 erweitert diese Anzahl auf drei.
- **ALI_MERGE (hypothetisch):** Führt mehrere ALI-Eingänge zu einem Ausgang zusammen – funktional gegensätzlich.
- **Ereignis-basierte Splitter (z. B. E_SPLIT):** Arbeiten mit reinen Ereignissen, während ALI_SPLIT_3 Daten- und Ereignisanteile gemeinsam über einen Adapter verteilt.
- **`ALI_SPLIT_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs

unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

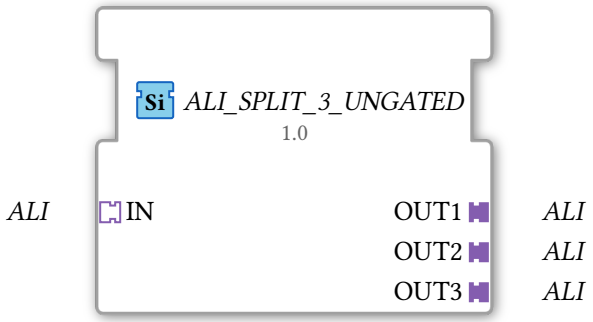
ALI_SPLIT_3 ist ein einfacher, aber nützlicher Funktionsblock zur Vervielfachung von ALI-Adapterverbindungen. Seine generische Natur und die reine Signalweitergabe machen ihn zu einem flexiblen Werkzeug in der Automatisierungs- und Steuerungstechnik, insbesondere wenn ein Signal mehrfach benötigt wird.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ALI_SPLIT_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALI_SPLIT_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ALI_SPLIT_3_UNGATED** dient dazu, einen eingehenden unidirektionalen Adapter vom Typ **ALI** (Application Layer Interface) auf drei identische Ausgangsadapter desselben Typs zu verteilen. Er ist als generischer Baustein konzipiert und ermöglicht eine Signalvervielfachung ohne Datenveränderung.

Schnittstellenstruktur

Ereignis-Eingänge

– Keine –

Ereignis-Ausgänge

– Keine –

Daten-Eingänge

– Keine (die gesamte Kommunikation erfolgt über die Adapter-Schnittstellen)

Daten-Ausgänge

– Keine (die gesamte Kommunikation erfolgt über die Adapter-Schnittstellen)

Adapter

Richtung	Name	Typ	Beschreibung

Socket (Eingang)	IN	adapter::types::unidirectional::ALI	Empfängt das eingehende ALI-Signal.
Plug (Ausgang)	OUT1	adapter::types::unidirectional::ALI	Erste Ausgangskopie des Signals.
Plug (Ausgang)	OUT2	adapter::types::unidirectional::ALI	Zweite Ausgangskopie des Signals.
Plug (Ausgang)	OUT3	adapter::types::unidirectional::ALI	Dritte Ausgangskopie des Signals.

Hinweis: Der Adaptertyp `ALI` ist ein unidirektionaler Interface-Adapter, der in der Regel Daten- und Ereignisfluss in einer Richtung kapselt. Die exakte interne Struktur ist anwendungsspezifisch definiert.

Funktionsweise

Der FB führt eine reine 1:3-Verteilung des am Socket `IN` anliegenden ALI-Adapters durch. Alle am Eingang empfangenen Informationen (z. B. Datenwerte, Ereignisse oder Zustände) werden ohne Verzögerung oder Transformation parallel an die drei Ausgänge `OUT1`, `OUT2` und `OUT3` weitergeleitet. Es findet keine Pufferung oder Logik statt – der Baustein arbeitet rein datenflusssynchron.

Technische Besonderheiten

- **Generischer Typ:** Der Baustein ist als generischer Funktionsblock deklariert (`eclipse4diac::core::GenericClassName = 'GEN_ALI_SPLIT'`). Dadurch kann der eigentliche Adaptertyp zur Konfigurationszeit festgelegt werden, sofern das zugrundeliegende System dies unterstützt.
- **Keine Eigendynamik:** Der FB besitzt keine eigenen Ereignis- oder Daten-Ein-/Ausgänge außerhalb der Adapter. Die gesamte Kommunikation läuft ausschließlich über die Adapter-Schnittstellen.
- **Geringe Latenz:** Durch den Verzicht auf interne Verarbeitung ist die Signalverteilung nahezu verzögerungsfrei.

Zustandsübersicht

Der Baustein besitzt keine Zustandsautomaten oder Zustandsspeicher. Sein Verhalten ist deterministisch und rein kombinatorisch: Der aktuelle Eingangszustand wird unmittelbar auf alle Ausgänge abgebildet.

Anwendungsszenarien

- **Signalvervielfachung in Steuerungssystemen:** Wenn ein ALI-Signal (z. B. ein Sensordatenstrom) mehrfach von verschiedenen nachgelagerten Bausteinen benötigt wird, kann ALI_SPLIT_3_UNGATED die Verteilung übernehmen.
- **Test- und Simulationsumgebungen:** Mehrere Komponenten sollen mit demselben Eingangssignal versorgt werden, um parallele Reaktionen zu beobachten.
- **Redundanz:** Ein Signal wird auf drei Pfade aufgeteilt, die unabhängig voneinander weiterverarbeitet werden (z. B. für Vergleich oder Fehlererkennung).

Vergleich mit ähnlichen Bausteinen

- **ALI_SPLIT_2:** Verteilt ein ALI-Signal auf zwei statt drei Ausgänge. ALI_SPLIT_3_UNGATED erweitert diese Anzahl auf drei.
- **ALI_MERGE (hypothetisch):** Führt mehrere ALI-Eingänge zu einem Ausgang zusammen – funktional gegensätzlich.
- **Ereignis-basierte Splitter (z. B. E_SPLIT):** Arbeiten mit reinen Ereignissen, während ALI_SPLIT_3_UNGATED Daten- und Ereignisanteile gemeinsam über einen Adapter verteilt.
- **`ALI_SPLIT_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

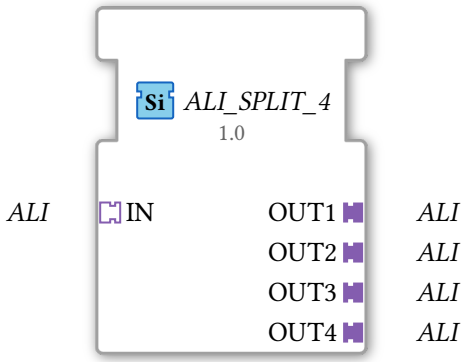
Fazit

ALI_SPLIT_3_UNGATED ist ein einfacher, aber nützlicher Funktionsblock zur Vervielfachung von ALI-Adapterverbindungen. Seine generische Natur und die reine Signalweitergabe machen ihn zu einem flexiblen Werkzeug in der Automatisierungs- und Steuerungstechnik, insbesondere wenn ein Signal mehrfach benötigt wird.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ALI_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **ALI_SPLIT_4** dient dazu, einen einzelnen ALI-Adapter-Eingang (Typ: `adapter::types::unidirectional::ALI`) auf vier separate ALI-Adapter-Ausgänge zu verteilen. Er ist als generischer Baustein (generic FB) ausgelegt und ermöglicht eine unidirektionale Signalweitergabe von einer Quelle an bis zu vier Senken.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	IN	adapter::types::unidirectional::ALI	Einheitlicher ALI-Eingang (Quelle)
Plug	OUT1	adapter::types::unidirectional::ALI	Erster ALI-Ausgang (Ziel)
Plug	OUT2	adapter::types::unidirectional::ALI	Zweiter ALI-Ausgang (Ziel)
Plug	OUT3	adapter::types::unidirectional::ALI	Dritter ALI-Ausgang (Ziel)
Plug	OUT4	adapter::types::unidirectional::ALI	Vierter ALI-Ausgang (Ziel)

Funktionsweise

Der Baustein empfängt über den Socket **IN** eine ALI-Adapterverbindung und leitet die eingehenden Signale unverändert an alle vier Plugs **OUT1** bis **OUT4** weiter. Es findet keine Datenverarbeitung, Filterung oder logische Verknüpfung statt – die Verteilung erfolgt rein passiv. Durch die generische Implementierung kann der Baustein in verschiedenen Kontexten eingesetzt werden, in denen ein ALI-Signal mehrfach benötigt wird.

Technische Besonderheiten

- Der FB ist als generischer Baustein (`GEN_ALI_SPLIT`) deklariert, was eine flexible Typanpassung in der Entwicklungsumgebung 4diac IDE ermöglicht.
- Es werden keinerlei Ereignisse oder Datenpunkte benötigt – die gesamte Kommunikation erfolgt ausschließlich über die Adapter-Schnittstellen.
- Der Baustein ist unidirektional ausgelegt: eine Rückkopplung von den Ausgängen zum Eingang ist nicht vorgesehen.
- Die Attribute `GenericClassName` und `TypeHash` dienen der eindeutigen Identifikation und Versionierung des generischen Typs.

Zustandsübersicht

Der Baustein besitzt keine eigenen Zustände oder Steuerlogik. Das Verhalten ist deterministisch und besteht ausschließlich in der Weiterleitung des Adaptersignals. Daher entfällt eine separate Zustandsmaschine.

Anwendungsszenarien

- **Signalverteilung**: Ein zentraler ALI-Datenstrom (z. B. von einem Sensor oder Steuergerät) soll parallel an mehrere nachgelagerte Komponenten gesendet werden.
- **Prototyping**: In frühen Entwicklungsphasen kann der Baustein verwendet werden, um ein Signal schnell auf mehrere Testeinheiten zu verteilen.
- **Redundanz**: Mehrere identische Ausgänge ermöglichen den Anschluss von Backup-Systemen oder separaten Überwachungseinheiten.

Vergleich mit ähnlichen Bausteinen

- **ALI_SPLIT_2**: Verteilt das Eingangssignal auf zwei statt vier Ausgänge. Der ALI_SPLIT_4 bietet die doppelte Anzahl an Ausgangskanälen.
- **ALI_MERGE**: Führt mehrere ALI-Eingänge zu einem Ausgang zusammen – die inverse Funktion zum Split.
- **ALI_SELECT**: Wählt basierend auf einem Steuersignal einen von mehreren Eingängen aus und leitet ihn an einen Ausgang weiter (keine Parallelverteilung).
- `ALI_SPLIT_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

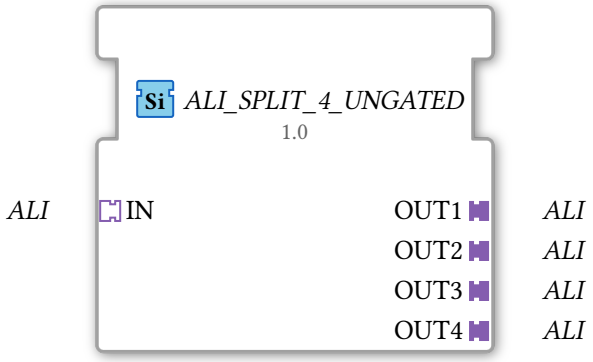
Der **ALI_SPLIT_4** ist ein einfacher, aber nützlicher Funktionsbaustein zur passiven Verteilung eines unidirektionalen ALI-Signals auf vier identische Ausgänge. Durch die generische Bauweise und das Fehlen von Ereignis- oder Datenlogik eignet er sich besonders für schnelle Signalvervielfachungen in der Automatisierungstechnik, ohne zusätzliche Rechenlast zu verursachen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ALI_SPLIT_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALI_SPLIT_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **ALI_SPLIT_4_UNGATED** dient dazu, einen einzelnen ALI-Adapter-Eingang (Typ: `adapter::types::unidirectional::ALI`) auf vier separate ALI-Adapter-Ausgänge zu verteilen. Er ist als generischer Baustein (generic FB) ausgelegt und ermöglicht eine unidirektionale Signalweitergabe von einer Quelle an bis zu vier Senken.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	IN	adapter::types::unidirectional::ALI	Einheitlicher ALI-Eingang (Quelle)
Plug	OUT1	adapter::types::unidirectional::ALI	Erster ALI-Ausgang (Ziel)
Plug	OUT2	adapter::types::unidirectional::ALI	Zweiter ALI-Ausgang (Ziel)
Plug	OUT3	adapter::types::unidirectional::ALI	Dritter ALI-Ausgang (Ziel)
Plug	OUT4	adapter::types::unidirectional::ALI	Vierter ALI-Ausgang (Ziel)

Funktionsweise

Der Baustein empfängt über den Socket **IN** eine ALI-Adapterverbindung und leitet die eingehenden Signale unverändert an alle vier Plugs **OUT1** bis **OUT4** weiter. Es findet keine Datenverarbeitung, Filterung oder logische Verknüpfung statt – die Verteilung erfolgt rein passiv. Durch die generische Implementierung kann der Baustein in verschiedenen Kontexten eingesetzt werden, in denen ein ALI-Signal mehrfach benötigt wird.

Technische Besonderheiten

- Der FB ist als generischer Baustein (`GEN_ALI_SPLIT`) deklariert, was eine flexible Typanpassung in der Entwicklungsumgebung 4diac IDE ermöglicht.
- Es werden keinerlei Ereignisse oder Datenpunkte benötigt – die gesamte Kommunikation erfolgt ausschließlich über die Adapter-Schnittstellen.

- Der Baustein ist unidirektional ausgelegt: eine Rückkopplung von den Ausgängen zum Eingang ist nicht vorgesehen.
- Die Attribute `GenericClassName` und `TypeHash` dienen der eindeutigen Identifikation und Versionierung des generischen Typs.

Zustandsübersicht

Der Baustein besitzt keine eigenen Zustände oder Steuerlogik. Das Verhalten ist deterministisch und besteht ausschließlich in der Weiterleitung des Adaptersignals. Daher entfällt eine separate Zustandsmaschine.

Anwendungsszenarien

- **Signalverteilung**: Ein zentraler ALI-Datenstrom (z. B. von einem Sensor oder Steuergerät) soll parallel an mehrere nachgelagerte Komponenten gesendet werden.
- **Prototyping**: In frühen Entwicklungsphasen kann der Baustein verwendet werden, um ein Signal schnell auf mehrere Testeinheiten zu verteilen.
- **Redundanz**: Mehrere identische Ausgänge ermöglichen den Anschluss von Backup-Systemen oder separaten Überwachungseinheiten.

Vergleich mit ähnlichen Bausteinen

- **ALI_SPLIT_2**: Verteilt das Eingangssignal auf zwei statt vier Ausgänge. Der ALI_SPLIT_4_UNGATED bietet die doppelte Anzahl an Ausgangskanälen.
- **ALI_MERGE**: Führt mehrere ALI-Eingänge zu einem Ausgang zusammen – die inverse Funktion zum Split.
- **ALI_SELECT**: Wählt basierend auf einem Steuersignal einen von mehreren Eingängen aus und leitet ihn an einen Ausgang weiter (keine Parallelverteilung).
- `ALI_SPLIT_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

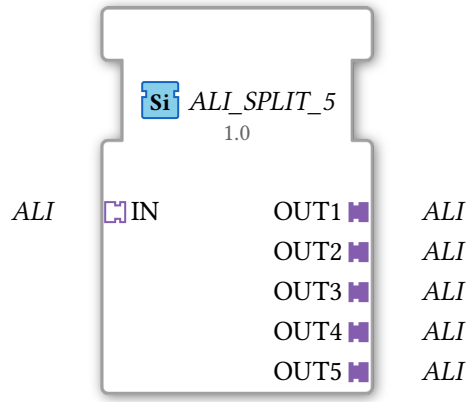
Fazit

Der **ALI_SPLIT_4_UNGATED** ist ein einfacher, aber nützlicher Funktionsbaustein zur passiven Verteilung eines unidirektionalen ALI-Signals auf vier identische Ausgänge. Durch die generische Bauweise und das Fehlen von Ereignis- oder Datenlogik eignet er sich besonders für schnelle Signalvervielfachungen in der Automatisierungstechnik, ohne zusätzliche Rechenlast zu verursachen.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ALI_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ALI_SPLIT_5` ist ein generischer Adapter-Split-Baustein. Er nimmt einen einzigen eingehenden ALI-Adapter (`adapter::types::unidirectional::ALI`) entgegen und stellt die gleichen ALI-Signale an fünf ausgehenden ALI-Adapter-Schnittstellen bereit. Dadurch kann eine ALI-Verbindung auf mehrere nachfolgende Bausteine oder Geräte verteilt werden, ohne die logische Signalkette zu unterbrechen.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine

Ereignis-Ausgänge

- Keine

Daten-Eingänge

- Keine

Daten-Ausgänge

- Keine

Adapter

Richtung	Name	Typ	Beschreibung
Eingang (Socket)	`IN`	`adapter::types::unidirectional::ALI`	Einziger Eingang: enthält die ALI-Daten, die auf die fünf Ausgänge verteilt werden sollen.
Ausgang (Plug)	`OUT1`	`adapter::types::unidirectional::ALI`	Erster Ausgang (identische Signale wie am Eingang).
Ausgang (Plug)	`OUT2`	`adapter::types::unidirectional::ALI`	Zweiter Ausgang.
Ausgang (Plug)	`OUT3`	`adapter::types::unidirectional::ALI`	Dritter Ausgang.
Ausgang (Plug)	`OUT4`	`adapter::types::unidirectional::ALI`	Vierter Ausgang.
Ausgang (Plug)	`OUT5`	`adapter::types::unidirectional::ALI`	Fünfter Ausgang.

Funktionsweise

Sobald der Baustein mit einer gültigen ALI-Verbindung am Socket `IN` verbunden ist, wird der gesamte Daten- und Ereignisverkehr des eingehenden Adapters **unverändert** und **parallel** auf die fünf Plug-Adapter `OUT1` bis `OUT5` weitergeleitet. Es findet keine logische Verarbeitung, Filterung oder Verzögerung statt – der Baustein wirkt als passive Verteilereinheit („Splitter“) für den ALI-Kanal.

Die Weiterleitung erfolgt bidirektional entsprechend der ALI-Adapterdefinition: Sowohl Ereignisse als auch Daten (wie sie im ALI-Typ vorgesehen sind) werden an allen Ausgängen synchron zur Verfügung gestellt. Der Baustein besitzt keine eigene Laufzeitlogik und keinen inneren Zustand.

Technische Besonderheiten

- **Generischer Baustein** – Die konkrete Implementierung wird durch die Meta-Attribute `GenericClassName` und `TypeHash` gesteuert. Dadurch kann dieselbe FB-Definition für verschiedene ALI-Varianten (z. B. mit unterschiedlichen Datenstrukturen) wiederverwendet werden.
- **Keine Verzögerung** – Die Signale werden ohne nennenswerte Laufzeitverzögerung auf die Ausgänge repliziert.
- **Einfach erweiterbar** – Das Prinzip dieses Splitters kann auf andere Adapter-Typen übertragen werden; durch die generische Architektur ist keine Änderung der Basislogik erforderlich.
- **Direkte Kopplung** – Der Ausgangszustand entspricht stets dem aktuellen Eingangszustand; es gibt keine Pufferung oder Zwischenspeicherung.

Zustandsübersicht

Der Baustein besitzt **keinen internen Zustandsautomaten**. Er arbeitet direkt und ohne Verzögerung als reine Verbindungsweiche („Wired-Split“). Zustandsänderungen treten nur durch die angeschlossenen Adapter-Partner auf, diese werden transparent weitergeleitet.

Anwendungsszenarien

- **Verteilen eines Sensorsignals** an mehrere Auswerteeinheiten oder Steuerungen, die den gleichen ALI-Datenstrom benötigen.
- **Parallelisierung von ALI-Kommunikationspfaden** in einer Maschinensteuerung, um redundante oder unabhängige Verarbeitungsketten zu versorgen.
- **Test- und Debugging-Umgebungen**, bei denen ein ALI-Signal gleichzeitig beobachtet und aufgezeichnet werden soll.

Vergleich mit ähnlichen Bausteinen

Andere Split- oder Fanout-Bausteine bieten meist eine feste Anzahl von Ausgängen oder sind auf bestimmte Datentypen festgelegt. Der `ALI_SPLIT_5` ist speziell auf den unidirektionalen ALI-Adapter zugeschnitten, jedoch durch seine generische Vorlage flexibel einsetzbar. Im Gegensatz zu einem **Multiplexer** oder **Router** leitet dieser Baustein das Signal **ohne Umschaltung** stets an alle Ausgänge weiter.

- `ALI_SPLIT_5_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

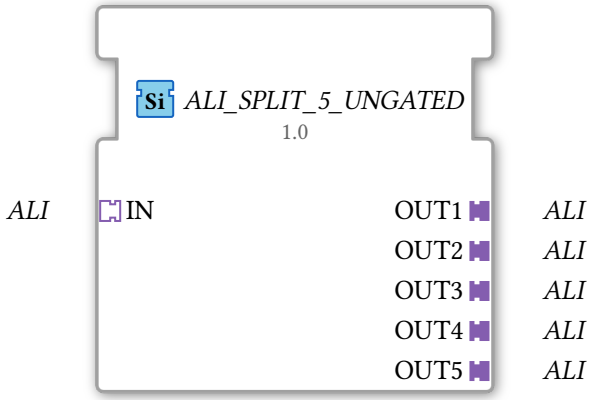
Der `ALI_SPLIT_5` ist ein einfacher, aber essenzieller Baustein zur Vervielfachung eines ALI-Signals. Aufgrund seiner generischen Natur lässt er sich problemlos in verschiedene 4diac-Projekte integrieren und vereinfacht die Signalverteilung in komplexen Automatisierungsarchitekturen. Er ist wartungsarm, da er keine eigene Logik enthält, und trägt zur Modularisierung und Wiederverwendbarkeit von ALI-Verbindungen bei.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ALI_SPLIT_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALI_SPLIT_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ALI_SPLIT_5_UNGATED` ist ein generischer Adapter-Split-Baustein. Er nimmt einen einzigen eingehenden ALI-Adapter (`adapter::types::unidirectional::ALI`) entgegen und stellt die gleichen ALI-Signale an fünf ausgehenden ALI-Adapter-Schnittstellen bereit. Dadurch kann eine ALI-Verbindung auf mehrere nachfolgende Bausteine oder Geräte verteilt werden, ohne die logische Signalkette zu unterbrechen.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine

Ereignis-Ausgänge

- Keine

Daten-Eingänge

- Keine

Daten-Ausgänge

- Keine

Adapter

Richtung	Name	Typ	Beschreibung
Eingang (Socket)	`IN`	`adapter::types::unidirectional::ALI`	Einziger Eingang: enthält die ALI-Daten, die auf die fünf Ausgänge verteilt werden sollen.
Ausgang (Plug)	`OUT1`	`adapter::types::unidirectional::ALI`	Erster Ausgang (identische Signale wie am Eingang).
Ausgang (Plug)	`OUT2`	`adapter::types::unidirectional::ALI`	Zweiter Ausgang.
Ausgang (Plug)	`OUT3`	`adapter::types::unidirectional::ALI`	Dritter Ausgang.
Ausgang (Plug)	`OUT4`	`adapter::types::unidirectional::ALI`	Vierter Ausgang.
Ausgang (Plug)	`OUT5`	`adapter::types::unidirectional::ALI`	Fünfter Ausgang.

Funktionsweise

Sobald der Baustein mit einer gültigen ALI-Verbindung am Socket `IN` verbunden ist, wird der gesamte Daten- und Ereignisverkehr des eingehenden Adapters **unverändert** und **parallel** auf die fünf Plug-Adapter `OUT1` bis `OUT5` weitergeleitet. Es findet keine logische Verarbeitung, Filterung oder Verzögerung statt – der Baustein wirkt als passive Verteilereinheit („Splitter“) für den ALI-Kanal.

Die Weiterleitung erfolgt bidirektional entsprechend der ALI-Adapterdefinition: Sowohl Ereignisse als auch Daten (wie sie im ALI-Typ vorgesehen sind) werden an allen Ausgängen synchron zur Verfügung gestellt. Der Baustein besitzt keine eigene Laufzeitlogik und keinen inneren Zustand.

Technische Besonderheiten

- **Generischer Baustein** – Die konkrete Implementierung wird durch die Meta-Attribute `GenericClassName` und `TypeHash` gesteuert. Dadurch kann dieselbe FB-Definition für verschiedene ALI-Varianten (z. B. mit unterschiedlichen Datenstrukturen) wiederverwendet werden.
- **Keine Verzögerung** – Die Signale werden ohne nennenswerte Laufzeitverzögerung auf die Ausgänge repliziert.
- **Einfach erweiterbar** – Das Prinzip dieses Splitters kann auf andere Adapter-Typen übertragen werden; durch die generische Architektur ist keine Änderung der Basislogik erforderlich.
- **Direkte Kopplung** – Der Ausgangszustand entspricht stets dem aktuellen Eingangszustand; es gibt keine Pufferung oder Zwischenspeicherung.

Zustandsübersicht

Der Baustein besitzt **keinen internen Zustandsautomaten**. Er arbeitet direkt und ohne Verzögerung als reine Verbindungsweiche („Wired-Split“). Zustandsänderungen treten nur durch die angeschlossenen Adapter-Partner auf, diese werden transparent weitergeleitet.

Anwendungsszenarien

- **Verteilen eines Sensorsignals** an mehrere Auswerteeinheiten oder Steuerungen, die den gleichen ALI-Datenstrom benötigen.
- **Parallelisierung von ALI-Kommunikationspfaden** in einer Maschinensteuerung, um redundante oder unabhängige Verarbeitungsketten zu versorgen.
- **Test- und Debugging-Umgebungen**, bei denen ein ALI-Signal gleichzeitig beobachtet und aufgezeichnet werden soll.

Vergleich mit ähnlichen Bausteinen

Andere Split- oder Fanout-Bausteine bieten meist eine feste Anzahl von Ausgängen oder sind auf bestimmte Datentypen festgelegt. Der `ALI_SPLIT_5_UNGATED` ist speziell auf den unidirektionalen ALI-Adapter zugeschnitten, jedoch durch seine generische Vorlage flexibel einsetzbar. Im Gegensatz zu einem **Multiplexer** oder **Router** leitet dieser Baustein das Signal **ohne Umschaltung** stets an alle Ausgänge weiter.

- `ALI_SPLIT_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

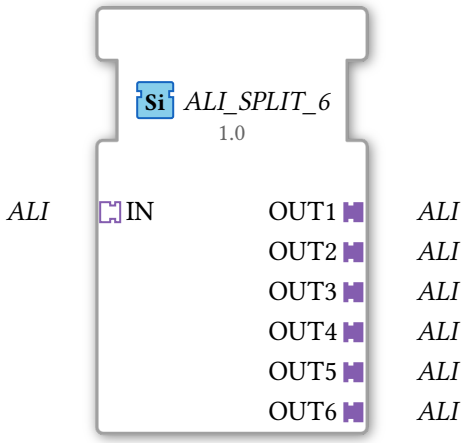
Fazit

Der `ALI_SPLIT_5_UNGATED` ist ein einfacher, aber essenzieller Baustein zur Vervielfachung eines ALI-Signals. Aufgrund seiner generischen Natur lässt er sich problemlos in verschiedene 4diac-Projekte integrieren und vereinfacht die Signalverteilung in komplexen Automatisierungsarchitekturen. Er ist wartungsarm, da er keine eigene Logik enthält, und trägt zur Modularisierung und Wiederverwendbarkeit von ALI-Verbindungen bei.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ALI_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock ALI_SPLIT_6 dient dazu, ein eingehendes unidirektionales ALI-Adapter-Signal (Application Layer Interface) auf sechs identische Ausgänge zu verteilen. Er ist als generischer Baustein konzipiert und ermöglicht eine einfache Signalvervielfachung ohne zusätzliche Logik.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::ALI`	`IN`	Socket (Eingang)	Ein ALI-Adapter, dessen Signal auf alle Ausgänge verteilt wird.
`adapter::types::unidirectional::ALI`	`OUT1`	Plug (Ausgang)	Erster Ausgang mit dem Signal des Eingangs.
`adapter::types::unidirectional::ALI`	`OUT2`	Plug (Ausgang)	Zweiter Ausgang mit dem Signal des Eingangs.
`adapter::types::unidirectional::ALI`	`OUT3`	Plug (Ausgang)	Dritter Ausgang mit dem Signal des Eingangs.
`adapter::types::unidirectional::ALI`	`OUT4`	Plug (Ausgang)	Vierter Ausgang mit dem Signal des Eingangs.
`adapter::types::unidirectional::ALI`	`OUT5`	Plug (Ausgang)	Fünfter Ausgang mit dem Signal des Eingangs.
`adapter::types::unidirectional::ALI`	`OUT6`	Plug (Ausgang)	Sechster Ausgang mit dem Signal des Eingangs.

Funktionsweise

Der ALI_SPLIT_6 realisiert eine einfache 1:6-Verteilung. Das über den Socket `IN` empfangene ALI-Adapter-Signal wird unverändert und gleichzeitig an alle sechs Plugs (`OUT1` bis `OUT6`) weitergeleitet. Es findet keine Signalverarbeitung, Verzögerung oder Zustandsänderung statt. Die Verteilung erfolgt rein strukturell über die Adapterverbindungen.

Technische Besonderheiten

- **Generischer Baustein:** Der FB ist als generischer Funktionsblock definiert. Über das Attribut `eclipse4diac::core::GenericClassName` kann der konkrete Klassenname (z. B. `'GEN_ALI_SPLIT'`) festgelegt werden. Das Attribut `eclipse4diac::core::TypeHash` kann zur Identifikation der spezifischen Konfiguration verwendet werden.
- **Keine Zustandsautomaten:** Der FB besitzt kein ECC (Execution Control Chart) und keinerlei interne Logik. Er arbeitet rein passiv.
- **Unidirektionale ALI-Adapterschnittstelle:** Sowohl Ein- als auch Ausgänge verwenden den Typ `adapter::types::unidirectional::ALI`, was eine einfache und standardisierte Kommunikation ermöglicht.

Zustandsübersicht

Der FB besitzt keine eigenen Zustände oder Verhaltensmodi, da er nur eine strukturelle Weiterleitung vornimmt. Eine Zustandsbetrachtung entfällt daher.

Anwendungsszenarien

- **Steuerung mehrerer Aktoren:** Ein ALI-Signal, z. B. ein Freigabe- oder Stellsignal, soll gleichzeitig an sechs Empfänger (z. B. Servomotoren oder Ventilinseln) weitergegeben werden.
- **Signalverteilung in modularen Systemen:** In einer industriellen Steuerung kann ein zentrales ALI-Signal auf mehrere dezentrale Einheiten aufgeteilt werden.
- **Ersatz von manuellen Verdrahtungen:** Der FB ersetzt die physische Aufteilung eines Signals durch eine softwaredefinierte, wartbare Lösung.

Vergleich mit ähnlichen Bausteinen

Baustein	Ausgänge	Besonderheit
ALI_SPLIT_2	2	Zweifachverteilung
ALI_SPLIT_4	4	Vierfachverteilung
ALI_SPLIT_6	**6**	**Sechsfachverteilung, generisch**
ALI_SPLIT_8	8	Achtfachverteilung

Der ALI_SPLIT_6 unterscheidet sich von anderen Split-Varianten lediglich durch die Anzahl der Ausgänge. Alle Bausteine dieser Familie arbeiten nach dem gleichen passiven Verteilerprinzip. Der hier beschriebene FB ist als generischer Baustein ausgeführt, während andere Varianten möglicherweise als einfache FB-Typen ohne Generizität implementiert sind.

- `ALI_SPLIT_6_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

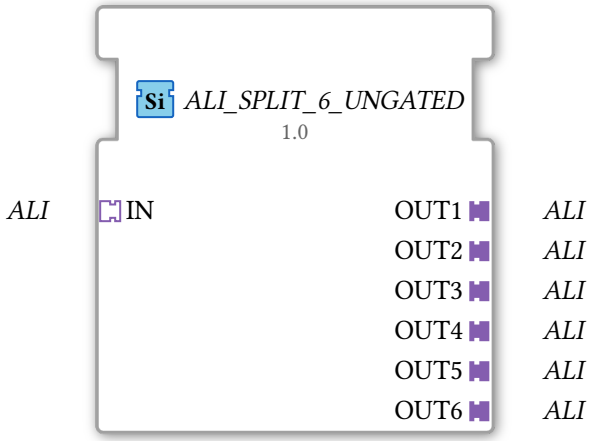
Der ALI_SPLIT_6 ist ein einfacher, aber nützlicher Funktionsblock zur Vervielfachung eines unidirektionalen ALI-Adapter-Signals auf sechs identische Ausgänge. Dank seiner generischen Eigenschaften lässt er sich flexibel in verschiedene Steuerungssysteme einbinden. Aufgrund der fehlenden Verarbeitungslogik ist er besonders ressourcenschonend und leicht verständlich. Er eignet sich ideal für Anwendungen, bei denen ein Signal parallel mehreren Empfängern zur Verfügung gestellt werden muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ALI_SPLIT_6_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALI_SPLIT_6`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock ALI_SPLIT_6_UNGATED dient dazu, ein eingehendes unidirektionales ALI-Adapter-Signal (Application Layer Interface) auf sechs identische Ausgänge zu verteilen. Er ist als generischer Baustein konzipiert und ermöglicht eine einfache Signalvervielfachung ohne zusätzliche Logik.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::ALI`	`IN`	Socket (Eingang)	Ein ALI-Adapter, dessen Signal auf alle Ausgänge verteilt wird.
`adapter::types::unidirectional::ALI`	`OUT1`	Plug (Ausgang)	Erster Ausgang mit dem Signal des Eingangs.

adapter::types::unidirectional::ALI	OUT2	Plug (Ausgang)	Zweiter Ausgang mit dem Signal des Eingangs.
adapter::types::unidirectional::ALI	OUT3	Plug (Ausgang)	Dritter Ausgang mit dem Signal des Eingangs.
adapter::types::unidirectional::ALI	OUT4	Plug (Ausgang)	Vierter Ausgang mit dem Signal des Eingangs.
adapter::types::unidirectional::ALI	OUT5	Plug (Ausgang)	Fünfter Ausgang mit dem Signal des Eingangs.
adapter::types::unidirectional::ALI	OUT6	Plug (Ausgang)	Sechster Ausgang mit dem Signal des Eingangs.

Funktionsweise

Der ALI_SPLIT_6_UNGATED realisiert eine einfache 1:6-Verteilung. Das über den Socket `IN` empfangene ALI-Adapter-Signal wird unverändert und gleichzeitig an alle sechs Plugs (`OUT1` bis `OUT6`) weitergeleitet. Es findet keine Signalverarbeitung, Verzögerung oder Zustandsänderung statt. Die Verteilung erfolgt rein strukturell über die Adapterverbindungen.

Technische Besonderheiten

- **Generischer Baustein:** Der FB ist als generischer Funktionsblock definiert. Über das Attribut `eclipse4diac::core::GenericClassName` kann der konkrete Klassenname (z. B. `'GEN_ALI_SPLIT'`) festgelegt werden. Das Attribut `eclipse4diac::core::TypeHash` kann zur Identifikation der spezifischen Konfiguration verwendet werden.
- **Keine Zustandsautomaten:** Der FB besitzt kein ECC (Execution Control Chart) und keinerlei interne Logik. Er arbeitet rein passiv.
- **Unidirektionale ALI-Adapterschnittstelle:** Sowohl Ein- als auch Ausgänge verwenden den Typ `adapter::types::unidirectional::ALI`, was eine einfache und standardisierte Kommunikation ermöglicht.

Zustandsübersicht

Der FB besitzt keine eigenen Zustände oder Verhaltensmodi, da er nur eine strukturelle Weiterleitung vornimmt. Eine Zustandsbetrachtung entfällt daher.

Anwendungsszenarien

- **Steuerung mehrerer Aktoren:** Ein ALI-Signal, z. B. ein Freigabe- oder Stellsignal, soll gleichzeitig an sechs Empfänger (z. B. Servomotoren oder Ventilinseln) weitergegeben werden.
- **Signalverteilung in modularen Systemen:** In einer industriellen Steuerung kann ein zentrales ALI-Signal auf mehrere dezentrale Einheiten aufgeteilt werden.
- **Ersatz von manuellen Verdrahtungen:** Der FB ersetzt die physische Aufteilung eines Signals durch eine softwaredefinierte, wartbare Lösung.

Vergleich mit ähnlichen Bausteinen

Baustein	Ausgänge	Besonderheit
ALI_SPLIT_2	2	Zweifachverteilung
ALI_SPLIT_4	4	Vierfachverteilung
ALI_SPLIT_6_UNGATED	**6**	**Sechsfachverteilung, generisch**
ALI_SPLIT_8	8	Achtfachverteilung

Der ALI_SPLIT_6_UNGATED unterscheidet sich von anderen Split-Varianten lediglich durch die Anzahl der Ausgänge. Alle Bausteine dieser Familie arbeiten nach dem gleichen passiven Verteilerprinzip. Der hier beschriebene FB ist als generischer Baustein ausgeführt, während andere Varianten möglicherweise als einfache FB-Typen ohne Generizität implementiert sind.

- **`ALI_SPLIT_6_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

- `ALI_SPLIT_6`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

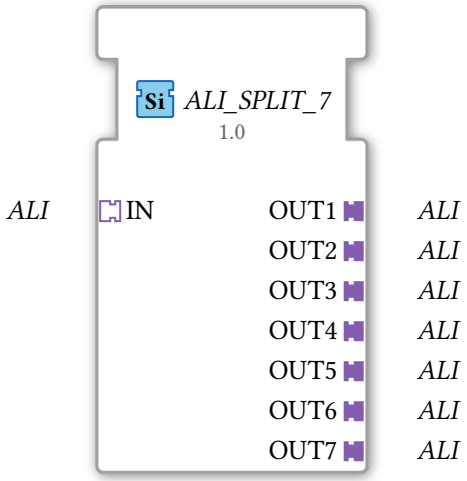
Fazit

Der ALI_SPLIT_6_UNGATED ist ein einfacher, aber nützlicher Funktionsblock zur Vervielfachung eines unidirektionalen ALI-Adapter-Signals auf sechs identische Ausgänge. Dank seiner generischen Eigenschaften lässt er sich flexibel in verschiedene Steuerungssysteme einbinden. Aufgrund der fehlenden Verarbeitungslogik ist er besonders ressourcenschonend und leicht verständlich. Er eignet sich ideal für Anwendungen, bei denen ein Signal parallel mehreren Empfängern zur Verfügung gestellt werden muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ALI_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **ALI_SPLIT_7** ist ein generischer Splitter, der einen eingehenden ALI-Adapter (Socket `IN`) auf sieben ausgehende ALI-Adapter (Plugs `OUT1`…`OUT7`) verteilt. Er dient zur 1:7-Verteilung von Daten und Ereignissen innerhalb eines unidirektionalen ALI-Kommunikationspfades.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	`IN`	`adapter::types::unidirectional::ALI`	Eingangsadapter, dessen Daten und Ereignisse verteilt werden.
Plug (Ausgang)	`OUT1`	`adapter::types::unidirectional::ALI`	Erster Ausgangsadapter.
Plug (Ausgang)	`OUT2`	`adapter::types::unidirectional::ALI`	Zweiter Ausgangsadapter.
Plug (Ausgang)	`OUT3`	`adapter::types::unidirectional::ALI`	Dritter Ausgangsadapter.
Plug (Ausgang)	`OUT4`	`adapter::types::unidirectional::ALI`	Vierter Ausgangsadapter.
Plug (Ausgang)	`OUT5`	`adapter::types::unidirectional::ALI`	Fünfter Ausgangsadapter.
Plug (Ausgang)	`OUT6`	`adapter::types::unidirectional::ALI`	Sechster Ausgangsadapter.
Plug (Ausgang)	`OUT7`	`adapter::types::unidirectional::ALI`	Siebter Ausgangsadapter.

Funktionsweise

Der Baustein leitet sämtliche über den Socket `IN` eingehenden ALI-Signale (sowohl Daten als auch Ereignisse) unverändert an alle sieben Ausgangsadapter `OUT1`…`OUT7` weiter. Somit entsteht eine strenge 1:7-Parallelverteilung ohne Verzögerung oder Pufferung. Der FB verhält sich rein kombinatorisch; es findet keine Verarbeitung oder Filterung der Inhalte statt.

Technische Besonderheiten

- **Generischer Typ**: Der FB ist als generischer Baustein deklariert (`GenericClassName = 'GEN_ALI_SPLIT'`). Dadurch kann er für unterschiedliche ALI-Adapter-Varianten verwendet werden, sofern diese das gleiche Schnittstellenprotokoll (unidirektionale Adapter) unterstützen.
- **Keine Zustände**: Der Baustein besitzt keine Zustandsmaschine (ECC) und speichert keinerlei Daten. Jeder Durchlauf ist deterministisch und nebenwirkungsfrei.
- **Flexible Verteilung**: Durch die feste Anzahl von sieben Ausgängen eignet sich der FB speziell für Anwendungen, die eine Aufteilung auf genau sieben Pfade erfordern.

Zustandsübersicht

Der FB verfügt über **keine expliziten Zustände**. Seine Funktionalität ist rein passiv und datenflussgetrieben.

Anwendungsszenarien

- **Parallele Signalverteilung**: Ein von einem übergeordneten Steuermodul gesendeter ALI-Datenstrom soll gleichzeitig an mehrere nachgelagerte Module (z. B. Aktoren, Visualisierungen oder Protokollierungen) weitergegeben werden.
- **Redundanz oder Lastverteilung**: Mehrere identische Verarbeitungseinheiten erhalten dieselben ALI-Eingangsdaten und arbeiten unabhängig voneinander.
- **Adapter-Konvertierung**: In Kombination mit spezialisierten ALI-Adaptern kann der Splitter dazu dienen, einen Datenpfad auf unterschiedliche Logik- oder Hardware-Schnittstellen zu vervielfachen.

Vergleich mit ähnlichen Bausteinen

- **ALI_SPLIT_2 / ALI_SPLIT_4**: Diese Bausteine bieten Aufteilungen auf zwei bzw. vier Ausgänge. `ALI_SPLIT_7` ist die Erweiterung auf sieben Ausgänge für spezifische Anforderungen.
- **ALI_MERGE**: Das Gegenstück, das mehrere ALI-Eingänge zu einem Ausgang zusammenführt. Ein Splitter wie `ALI_SPLIT_7` arbeitet in umgekehrter Richtung (Fan-Out).
- **`ALI_SPLIT_7_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs

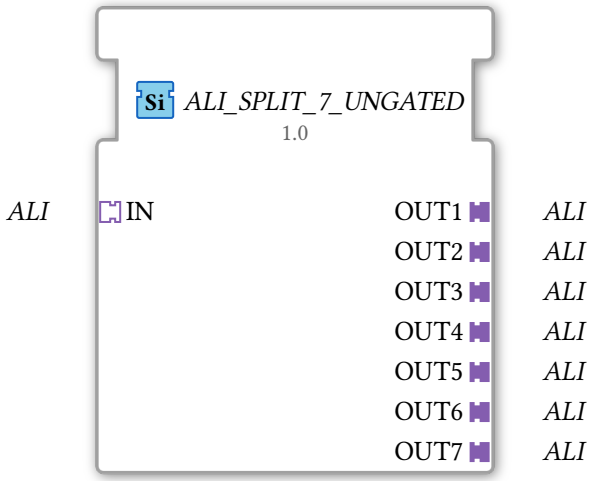
unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der **ALI_SPLIT_7** ist ein spezialisierter, generischer Funktionsbaustein zur sauberen und verlustfreien 1:7-Verteilung von unidirektionalen ALI-Signalen. Durch seine einfache Struktur, die fehlende Zustandslogik und die generische Auslegung eignet er sich ideal für modulare Steuerungsarchitekturen, bei denen ein Signal parallel an mehrere Empfänger weitergeleitet werden muss.

ALI_SPLIT_7_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALI_SPLIT_7`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **ALI_SPLIT_7_UNGATED** ist ein generischer Splitter, der einen eingehenden ALI-Adapter (Socket `IN`) auf sieben ausgehende ALI-Adapter (Plugs `OUT1`…`OUT7`) verteilt. Er dient zur 1:7-Verteilung von Daten und Ereignissen innerhalb eines unidirektionalen ALI-Kommunikationspfades.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	`IN`	`adapter::types::unidirectional::ALI`	Eingangsadapter, dessen Daten und Ereignisse verteilt werden.
Plug (Ausgang)	`OUT1`	`adapter::types::unidirectional::ALI`	Erster Ausgangsadapter.
Plug (Ausgang)	`OUT2`	`adapter::types::unidirectional::ALI`	Zweiter Ausgangsadapter.
Plug (Ausgang)	`OUT3`	`adapter::types::unidirectional::ALI`	Dritter Ausgangsadapter.
Plug (Ausgang)	`OUT4`	`adapter::types::unidirectional::ALI`	Vierter Ausgangsadapter.
Plug (Ausgang)	`OUT5`	`adapter::types::unidirectional::ALI`	Fünfter Ausgangsadapter.
Plug (Ausgang)	`OUT6`	`adapter::types::unidirectional::ALI`	Sechster Ausgangsadapter.
Plug (Ausgang)	`OUT7`	`adapter::types::unidirectional::ALI`	Siebter Ausgangsadapter.

Funktionsweise

Der Baustein leitet sämtliche über den Socket `IN` eingehenden ALI-Signale (sowohl Daten als auch Ereignisse) unverändert an alle sieben Ausgangsadapter `OUT1`…`OUT7` weiter. Somit entsteht eine strenge 1:7-Parallelverteilung ohne Verzögerung oder Pufferung. Der FB verhält sich rein kombinatorisch; es findet keine Verarbeitung oder Filterung der Inhalte statt.

Technische Besonderheiten

- **Generischer Typ**: Der FB ist als generischer Baustein deklariert (`GenericClassName = 'GEN_ALI_SPLIT'`). Dadurch kann er für unterschiedliche ALI-Adapter-Varianten verwendet werden, sofern diese das gleiche Schnittstellenprotokoll (unidirektionale Adapter) unterstützen.
- **Keine Zustände**: Der Baustein besitzt keine Zustandsmaschine (ECC) und speichert keinerlei Daten. Jeder Durchlauf ist deterministisch und nebenwirkungsfrei.
- **Flexible Verteilung**: Durch die feste Anzahl von sieben Ausgängen eignet sich der FB speziell für Anwendungen, die eine Aufteilung auf genau sieben Pfade erfordern.

Zustandsübersicht

Der FB verfügt über **keine expliziten Zustände**. Seine Funktionalität ist rein passiv und datenflussgetrieben.

Anwendungsszenarien

- **Parallele Signalverteilung**: Ein von einem übergeordneten Steuermodul gesendeter ALI-Datenstrom soll gleichzeitig an mehrere nachgelagerte Module (z. B. Aktoren, Visualisierungen oder Protokollierungen) weitergegeben werden.
- **Redundanz oder Lastverteilung**: Mehrere identische Verarbeitungseinheiten erhalten dieselben ALI-Eingangsdaten und arbeiten unabhängig voneinander.
- **Adapter-Konvertierung**: In Kombination mit spezialisierten ALI-Adaptern kann der Splitter dazu dienen, einen Datenpfad auf unterschiedliche Logik- oder Hardware-Schnittstellen zu vervielfachen.

Vergleich mit ähnlichen Bausteinen

- **ALI_SPLIT_2 / ALI_SPLIT_4**: Diese Bausteine bieten Aufteilungen auf zwei bzw. vier Ausgänge. `ALI_SPLIT_7_UNGATED` ist die Erweiterung auf sieben Ausgänge für spezifische Anforderungen.
- **ALI_MERGE**: Das Gegenstück, das mehrere ALI-Eingänge zu einem Ausgang zusammenführt. Ein Splitter wie `ALI_SPLIT_7_UNGATED` arbeitet in umgekehrter Richtung (Fan-Out).
- `ALI_SPLIT_7_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- `ALI_SPLIT_7`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

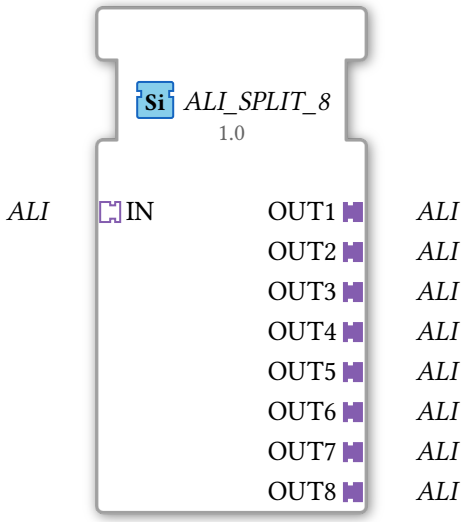
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **ALI_SPLIT_7_UNGATED** ist ein spezialisierter, generischer Funktionsbaustein zur sauberen und verlustfreien 1:7-Verteilung von unidirektionalen ALI-Signalen. Durch seine einfache Struktur, die fehlende Zustandslogik und die generische Auslegung eignet er sich ideal für modulare Steuerungsarchitekturen, bei denen ein Signal parallel an mehrere Empfänger weitergeleitet werden muss.

ALI_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ALI_SPLIT_8` dient als generischer Splitter für das **Agricultural Light Interface (ALI)**. Er verteilt ein eingehendes ALI-Signal unverändert auf acht parallele Ausgänge. Dadurch können mehrere nachfolgende Verbraucher oder Steuerungen mit dem gleichen Signal versorgt werden, ohne die Signalintegrität zu beeinträchtigen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

- **Socket (Eingang):** `IN` – vom Typ `adapter::types::unidirectional::ALI` Nimmt das eingehende ALI-Signal auf.
- **Plugs (Ausgänge):** `OUT1` … `OUT8` – jeweils vom Typ `adapter::types::unidirectional::ALI` Stellen die acht identischen Kopien des Eingangssignals bereit.

Funktionsweise

Der Baustein leitet das am Socket `IN` anliegende ALI-Signal unverändert und ohne Verzögerung an alle acht Plugs (`OUT1` … `OUT8`) weiter. Da es sich um eine reine Signalverteilung handelt, findet keine Bearbeitung, Pufferung oder Statusänderung statt. Sämtliche Ereignisse und Daten, die über den ALI-Adapter transportiert werden, werden auf jeden Ausgang repliziert.

Technische Besonderheiten

- **Generischer Baustein:** Der FB ist als generischer Typ (`GEN_ALI_SPLIT`) definiert und kann in verschiedenen Kontexten des ALI-Ökosystems eingesetzt werden.
- **Kein ECC:** Es existiert kein Ausführungszustandsautomat (ECC); die Verteilung erfolgt rein kombinatorisch.
- **Unidirektional:** Der Adaptertyp ist unidirektional – das Signal fließt nur vom Socket zu den Plugs, eine Rückkopplung ist nicht vorgesehen.
- **Skalierbarkeit:** Durch den modularen Aufbau können ähnliche Splitter für andere Ausgangszahlen (z.B. `ALI_SPLIT_2`) abgeleitet werden.

Zustandsübersicht

Der Baustein besitzt keine internen Zustände oder sequentielle Logik. Die Signalverteilung arbeitet kontinuierlich und ohne Verzögerung.

Anwendungsszenarien

- **Verteilschaltungen:** Ein ALI-Signal (z.B. von einem Steuergerät) muss gleichzeitig an mehrere Aktoren oder Sensoren weitergegeben werden.
- **Redundanz:** Parallele Ansteuerung baugleicher Komponenten, ohne separate Signalquellen vorzusehen.
- **Busstrukturen:** Aufbau von sternförmigen ALI-Netzwerken mit einem zentralen Splitter.

Vergleich mit ähnlichen Bausteinen

Andere Splitter-Bausteine wie `ALI_SPLIT_2`, `ALI_SPLIT_4` oder `ALI_SPLIT_N` unterscheiden sich lediglich in der Anzahl der Ausgänge. Der `ALI_SPLIT_8` bietet die maximale Verteilung in der Standardfamilie. Im Gegensatz zu einem Multiplexer (`ALI_MUX`) oder einer Weiche (`ALI_SWITCH`) wird hier **jeder Ausgang mit dem gleichen Signal** versorgt – es findet keine Auswahl oder Umschaltung statt.

- `ALI_SPLIT_8_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

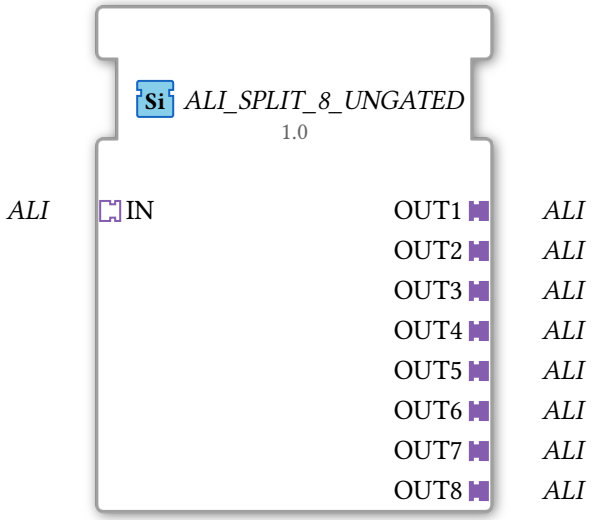
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der `ALI_SPLIT_8` ist ein einfacher, aber essenzieller Baustein zur Vervielfältigung von ALI-Signalen. Durch seine generische Definition und den Verzicht auf Zustandslogik eignet er sich besonders für robuste, latenzarme Verteilungen in der landwirtschaftlichen Automatisierungstechnik.

ALI_SPLIT_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALI_SPLIT_8`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ALI_SPLIT_8_UNGATED` dient als generischer Splitter für das **Agricultural Light Interface (ALI)**. Er verteilt ein eingehendes ALI-Signal unverändert auf acht parallele Ausgänge. Dadurch können mehrere nachfolgende Verbraucher oder Steuerungen mit dem gleichen Signal versorgt werden, ohne die Signalintegrität zu beeinträchtigen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

- **Socket (Eingang):** `IN` – vom Typ `adapter::types::unidirectional::ALI` Nimmt das eingehende ALI-Signal auf.
- **Plugs (Ausgänge):** `OUT1` … `OUT8` – jeweils vom Typ `adapter::types::unidirectional::ALI` Stellen die acht identischen Kopien des Eingangssignals bereit.

Funktionsweise

Der Baustein leitet das am Socket `IN` anliegende ALI-Signal unverändert und ohne Verzögerung an alle acht Plugs (`OUT1` … `OUT8`) weiter. Da es sich um eine reine Signalverteilung handelt, findet keine Bearbeitung, Pufferung oder Statusänderung statt. Sämtliche Ereignisse und Daten, die über den ALI-Adapter transportiert werden, werden auf jeden Ausgang repliziert.

Technische Besonderheiten

- **Generischer Baustein:** Der FB ist als generischer Typ (`GEN_ALI_SPLIT`) definiert und kann in verschiedenen Kontexten des ALI-Ökosystems eingesetzt werden.
- **Kein ECC:** Es existiert kein Ausführungszustandsautomat (ECC); die Verteilung erfolgt rein kombinatorisch.

- **Unidirektional:** Der Adaptertyp ist unidirektional – das Signal fließt nur vom Socket zu den Plugs, eine Rückkopplung ist nicht vorgesehen.
- **Skalierbarkeit:** Durch den modularen Aufbau können ähnliche Splitter für andere Ausgangszahlen (z.B. `ALI_SPLIT_2`) abgeleitet werden.

Zustandsübersicht

Der Baustein besitzt keine internen Zustände oder sequentielle Logik. Die Signalverteilung arbeitet kontinuierlich und ohne Verzögerung.

Anwendungsszenarien

- **Verteilschaltungen:** Ein ALI-Signal (z.B. von einem Steuergerät) muss gleichzeitig an mehrere Aktoren oder Sensoren weitergegeben werden.
- **Redundanz:** Parallele Ansteuerung baugleicher Komponenten, ohne separate Signalquellen vorzusehen.
- **Busstrukturen:** Aufbau von sternförmigen ALI-Netzwerken mit einem zentralen Splitter.

Vergleich mit ähnlichen Bausteinen

Andere Splitter-Bausteine wie `ALI_SPLIT_2`, `ALI_SPLIT_4` oder `ALI_SPLIT_N` unterscheiden sich lediglich in der Anzahl der Ausgänge. Der `ALI_SPLIT_8_UNGATED` bietet die maximale Verteilung in der Standardfamilie. Im Gegensatz zu einem Multiplexer (`ALI_MUX`) oder einer Weiche (`ALI_SWITCH`) wird hier **jeder Ausgang mit dem gleichen Signal** versorgt – es findet keine Auswahl oder Umschaltung statt.

- `ALI_SPLIT_8`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

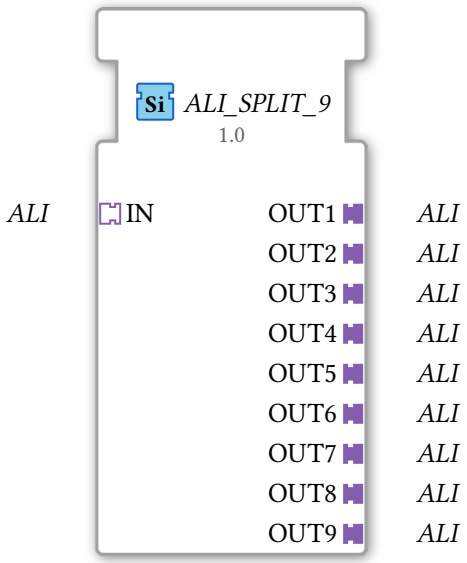
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `ALI_SPLIT_8_UNGATED` ist ein einfacher, aber essenzieller Baustein zur Vervielfältigung von ALI-Signalen. Durch seine generische Definition und den Verzicht auf Zustandslogik eignet er sich besonders für robuste, latenzarme Verteilungen in der landwirtschaftlichen Automatisierungstechnik.

ALI_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **ALI_SPLIT_9** ist ein generischer Funktionsbaustein (FB) innerhalb der 4diac-IDE, der dazu dient, ein eingehendes **ALI**-Adapter-Signal (unidirektional) auf bis zu neun separate Ausgangsadapter aufzuteilen.

Der Baustein ist als generischer Typ („Generic FB“) realisiert und wird üblicherweise über die Typ-Parameter `eclipse4diac::core::GenericClassName` und `eclipse4diac::core::TypeHash` individualisiert. Ziel ist die flexible Signalverteilung ohne zusätzliche Logik, rein auf Adapterebene.

Schnittstellenstruktur

Ereignis-Eingänge
Keine

Ereignis-Ausgänge
Keine

Daten-Eingänge
Keine

Daten-Ausgänge
Keine

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	IN	`ALI` (unidirektional)	Einspeisung des zu verteilenden ALI-Signals
Plug (Ausgang 1)	OUT1	`ALI` (unidirektional)	Erster Ausgang – identische Kopie des Eingangssignals
Plug (Ausgang 2)	OUT2	`ALI` (unidirektional)	Zweiter Ausgang
Plug (Ausgang 3)	OUT3	`ALI` (unidirektional)	Dritter Ausgang
Plug (Ausgang 4)	OUT4	`ALI` (unidirektional)	Vierter Ausgang
Plug (Ausgang 5)	OUT5	`ALI` (unidirektional)	Fünfter Ausgang
Plug (Ausgang 6)	OUT6	`ALI` (unidirektional)	Sechster Ausgang
Plug (Ausgang 7)	OUT7	`ALI` (unidirektional)	Siebter Ausgang
Plug (Ausgang 8)	OUT8	`ALI` (unidirektional)	Achter Ausgang
Plug (Ausgang 9)	OUT9	`ALI` (unidirektional)	Neunter Ausgang

Funktionsweise
Der Baustein arbeitet passiv als reiner **Signalverteiler**. Das am Socket **IN** anliegende ALI-Adapterprotokoll wird unverändert auf alle neun Plug-Ausgänge **OUT1…OUT9** dupliziert. Es findet keinerlei Datenverarbeitung, Ereignisgenerierung oder Filterung statt. Jeder Ausgang bietet eine identische logische und zeitliche Kopie des eingehenden Signals.

Die interne Implementierung erfolgt über die generischen Typmechanismen der 4diac-IDE (`GenericClassName` und `TypeHash`), sodass der Baustein zur Laufzeit spezifisch an die Anforderungen des Projekts angepasst werden kann.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als `GEN_ALI_SPLIT` deklariert (laut Attribut `eclipse4diac::core::GenericClassName`). Konkrete Instanzen erhalten eine eindeutige Typkennung über `TypeHash`.
- **Keine Ereignisse oder Daten**: Die Schnittstelle besteht ausschließlich aus Adaptern; es gibt keine Ereignis-Ein-/Ausgänge oder Data-Ein-/Ausgänge.
- **Unidirektionale ALI-Adapter**: Alle verwendeten Adapter sind vom Typ `adapter::types::unidirectional::ALI`, d. h. die Kommunikation erfolgt in einer Richtung (Eingang → Ausgänge). Rückwirkung von den Ausgängen auf den Eingang ist nicht vorgesehen.
- **Einfache Topologie**: Der Baustein realisiert eine 1:9-Sternverteilung ohne zusätzliche Puffer oder Synchronisierung.

Zustandsübersicht

Der Baustein besitzt keinen eigenen Zustandsautomaten. Sein Verhalten ist rein durch die Adapter-Schnittstellen bestimmt: Solange der Socket **IN** ein Signal liefert, sind alle neun Plugs aktiv. Sobald **IN** inaktiv wird (kein Signal), liefern auch alle Ausgänge kein Signal.

Anwendungsszenarien

- **Verteilen eines ALI-Bus-Signals** an mehrere nachgeschaltete Subsysteme (z. B. Sensordaten an mehrere Steuergeräte).
- **Aufteilung eines Master-Signals** für Redundanz oder Parallelverarbeitung in der Anlagensteuerung.
- **Test- und Debugging-Umgebungen**: Ein ALI-Signal kann gleichzeitig an ein Analysegerät und an mehrere Zielsysteme weitergeleitet werden.
- **Generische Schnittstellenerweiterung**: Wenn ein System nur einen ALI-Ausgang besitzt, aber mehrere Verbraucher bedient werden müssen.

Vergleich mit ähnlichen Bausteinen

- **ALI_SPLIT_2 / ALI_SPLIT_4 / ALI_SPLIT_8**: Varianten mit 2, 4 oder 8 Ausgängen. Der hier beschriebene Baustein bietet die maximale Aufteilung auf 9 Kanäle.
- **ALI_MERGE** (hypothetisch): Fasst mehrere ALI-Eingänge zu einem Ausgang zusammen – gegenteilige Funktion.
- **Ereignisbasierte Splitter**: Andere Bausteine nutzen Ereignisse zur Steuerung des Signalflusses; dieser Adapter-Splitter arbeitet signalgetrieben ohne Ereignislogik.
- `ALI_SPLIT_9_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

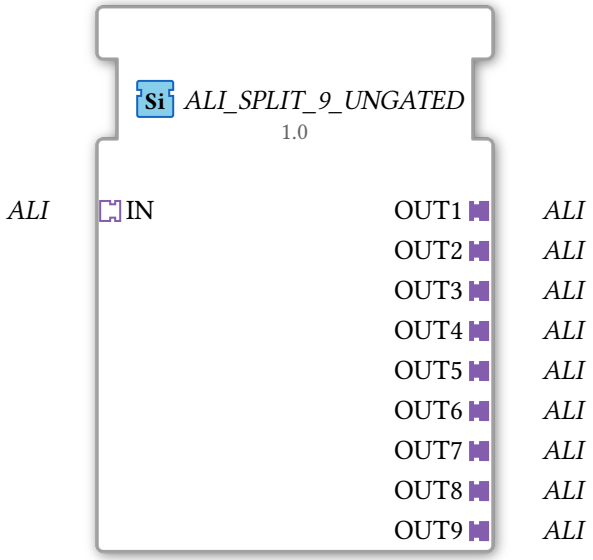
Der **ALI_SPLIT_9** ist ein spezialisierter, generischer Adapter-FB, der in der industriellen Automatisierung mit 4diac-IDE die einfache und zuverlässige Verteilung eines unidirektionalen ALI-Signals auf bis zu neun separate Empfänger ermöglicht. Durch seine reine Adapterschnittstelle ist er besonders ressourcenschonend und eignet sich ideal für sternförmige Vernetzungstopologien in IEC 61499-basierten Systemen. Der generische Charakter erlaubt eine flexible Anpassung an spezifische Typanforderungen ohne Änderung der Bausteinstruktur.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ALI_SPLIT_9_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALI_SPLIT_9`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **ALI_SPLIT_9_UNGATED** ist ein generischer Funktionsbaustein (FB) innerhalb der 4diac-IDE, der dazu dient, ein eingehendes **ALI**-Adapter-Signal (unidirektional) auf bis zu neun separate Ausgangsadapter aufzuteilen. Der Baustein ist als generischer Typ („Generic FB“) realisiert und wird üblicherweise über die Typ-Parameter `eclipse4diac::core::GenericClassName` und `eclipse4diac::core::TypeHash` individualisiert. Ziel ist die flexible Signalverteilung ohne zusätzliche Logik, rein auf Adapterebene.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	IN	`ALI` (unidirektional)	Einspeisung des zu verteilenden ALI-Signals
Plug (Ausgang 1)	OUT1	`ALI` (unidirektional)	Erster Ausgang – identische Kopie des Eingangssignals
Plug (Ausgang 2)	OUT2	`ALI` (unidirektional)	Zweiter Ausgang
Plug (Ausgang 3)	OUT3	`ALI` (unidirektional)	Dritter Ausgang
Plug (Ausgang 4)	OUT4	`ALI` (unidirektional)	Vierter Ausgang
Plug (Ausgang 5)	OUT5	`ALI` (unidirektional)	Fünfter Ausgang
Plug (Ausgang 6)	OUT6	`ALI` (unidirektional)	Sechster Ausgang
Plug (Ausgang 7)	OUT7	`ALI` (unidirektional)	Siebter Ausgang

Plug (Ausgang 8)	OUT8	`ALI` (unidirektional)	Achter Ausgang
Plug (Ausgang 9)	OUT9	`ALI` (unidirektional)	Neunter Ausgang

Funktionsweise

Der Baustein arbeitet passiv als reiner **Signalverteiler**. Das am Socket **IN** anliegende ALI-Adapterprotokoll wird unverändert auf alle neun Plug-Ausgänge **OUT1…OUT9** dupliziert. Es findet keinerlei Datenverarbeitung, Ereignisgenerierung oder Filterung statt. Jeder Ausgang bietet eine identische logische und zeitliche Kopie des eingehenden Signals.

Die interne Implementierung erfolgt über die generischen Typmechanismen der 4diac-IDE (`GenericClassName` und `TypeHash`), sodass der Baustein zur Laufzeit spezifisch an die Anforderungen des Projekts angepasst werden kann.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als `GEN_ALI_SPLIT` deklariert (laut Attribut `eclipse4diac::core::GenericClassName`). Konkrete Instanzen erhalten eine eindeutige Typkennung über `TypeHash`.
- **Keine Ereignisse oder Daten**: Die Schnittstelle besteht ausschließlich aus Adaptern; es gibt keine Ereignis-Ein-/Ausgänge oder Data-Ein-/Ausgänge.
- **Unidirektionale ALI-Adapter**: Alle verwendeten Adapter sind vom Typ `adapter::types::unidirectional::ALI`, d. h. die Kommunikation erfolgt in einer Richtung (Eingang → Ausgänge). Rückwirkung von den Ausgängen auf den Eingang ist nicht vorgesehen.
- **Einfache Topologie**: Der Baustein realisiert eine 1:9-Sternverteilung ohne zusätzliche Puffer oder Synchronisierung.

Zustandsübersicht

Der Baustein besitzt keinen eigenen Zustandsautomaten. Sein Verhalten ist rein durch die Adapter-Schnittstellen bestimmt: Solange der Socket **IN** ein Signal liefert, sind alle neun Plugs aktiv. Sobald **IN** inaktiv wird (kein Signal), liefern auch alle Ausgänge kein Signal.

Anwendungsszenarien

- **Verteilen eines ALI-Bus-Signals** an mehrere nachgeschaltete Subsysteme (z. B. Sensordaten an mehrere Steuergeräte).
- **Aufteilung eines Master-Signals** für Redundanz oder Parallelverarbeitung in der Anlagensteuerung.
- **Test- und Debugging-Umgebungen**: Ein ALI-Signal kann gleichzeitig an ein Analysegerät und an mehrere Zielsysteme weitergeleitet werden.
- **Generische Schnittstellenerweiterung**: Wenn ein System nur einen ALI-Ausgang besitzt, aber mehrere Verbraucher bedient werden müssen.

Vergleich mit ähnlichen Bausteinen

- **ALI_SPLIT_2 / ALI_SPLIT_4 / ALI_SPLIT_8**: Varianten mit 2, 4 oder 8 Ausgängen. Der hier beschriebene Baustein bietet die maximale Aufteilung auf 9 Kanäle.
- **ALI_MERGE** (hypothetisch): Fasst mehrere ALI-Eingänge zu einem Ausgang zusammen – gegenteilige Funktion.
- **Ereignisbasierte Splitter**: Andere Bausteine nutzen Ereignisse zur Steuerung des Signalflusses; dieser Adapter-Splitter arbeitet signalgetrieben ohne Ereignislogik.
- `ALI_SPLIT_9`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **ALI_SPLIT_9_UNGATED** ist ein spezialisierter, generischer Adapter-FB, der in der industriellen Automatisierung mit 4diac-IDE die einfache und zuverlässige Verteilung eines unidirektionalen ALI-Signals auf bis zu neun separate Empfänger ermöglicht. Durch seine reine Adapterschnittstelle ist er

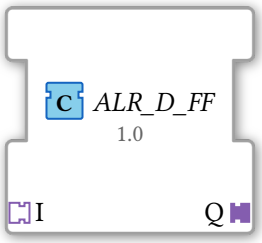
besonders ressourcenschonend und eignet sich ideal für sternförmige Vernetzungstopologien in IEC 61499-basierten Systemen. Der generische Charakter erlaubt eine flexible Anpassung an spezifische Typanforderungen ohne Änderung der Bausteinstruktur.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

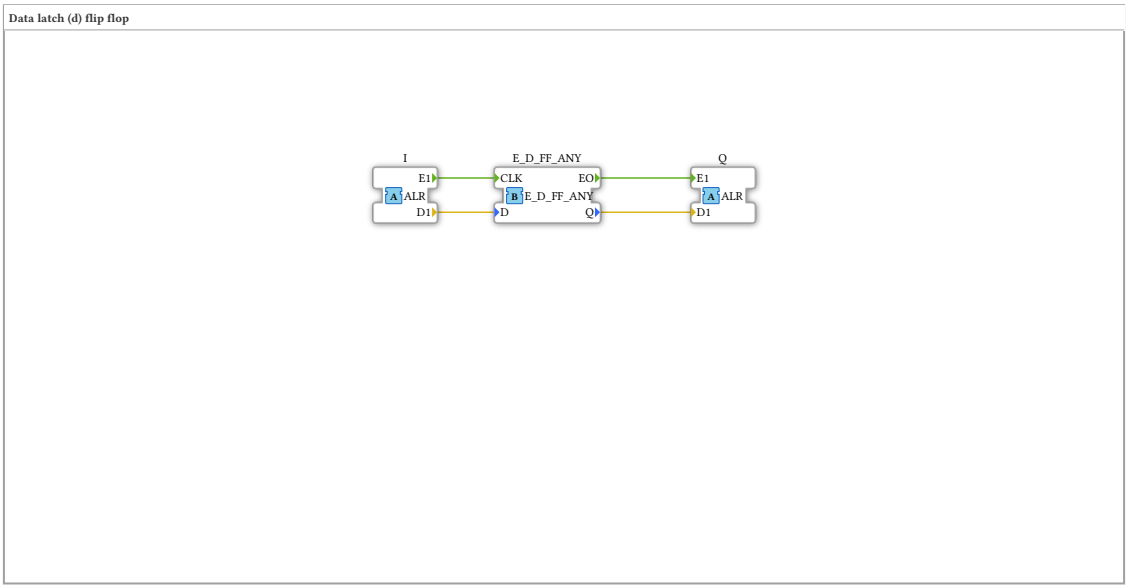
LREAL

ALR_D_FF



Value to latch – *ALR*

ALR – Latched value



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ALR_D_FF** implementiert ein datengesteuertes Flip-Flop (D-Latch) mit Adapter-basierter Schnittstelle. Er übernimmt einen Datenwert bei einem Taktereignis und stellt diesen am Ausgang zur Verfügung. Der Baustein kapselt die IEC-61499-Standardfunktion `E_D_FF_ANY` und bietet eine unidirektionale Adapter-Anbindung für eine modulare Systemintegration.

Schnittstellenstruktur

Der FB besitzt keine direkten Ereignis- oder Dateneingänge/-ausgänge. Die gesamte Kommunikation erfolgt über zwei Adapter:

Ereignis-Eingänge

- **I.E1** (über Socket `I`): Taktereignis – bei einer steigenden Flanke wird der aktuelle Datenwert übernommen.

Ereignis-Ausgänge

- **Q.E1** (über Plug `Q`): Bestätigungsereignis – wird nach erfolgreicher Übernahme des Datenwertes gesendet.

Daten-Eingänge

- **I.D1** (über Socket `I`): Dateneingang – der zu latchende Wert (Datentyp wird durch den Adapter `ALR` definiert).

Daten-Ausgänge

- **Q.D1** (über Plug `Q`): Datenausgang – der gespeicherte, latched Wert.

Adapter

Adapter	Typ	Richtung	Beschreibung
`I`	adapter::types::unidirectional::ALR	Socket (Eingang)	Liefert das Taktereignis und den Datenwert.
`Q`	adapter::types::unidirectional::ALR	Plug (Ausgang)	Gibt das Ausgangsereignis und den latched Wert aus.

Funktionsweise

1. Der FB erwartet ein Taktereignis am Adapter-Eingang `I.E1`.
2. Gleichzeitig wird der Datenwert von `I.D1` gelesen.
3. Der interne Baustein `E_D_FF_ANY` übernimmt den Datenwert bei der steigenden Flanke des Taktes.
4. Der übernommene Wert wird unverändert auf `Q.D1` ausgegeben.
5. Ein Ausgangsereignis auf `Q.E1` signalisiert, dass der neue Wert bereitsteht.

Der Vorgang wiederholt sich bei jedem weiteren Taktereignis. Zwischen den Takten bleibt der Ausgangswert stabil (Latch-Verhalten).

Technische Besonderheiten

- **Adapter-basierte Schnittstelle**: Der FB nutzt den unidirektionalen Adapter `ALR`, der genau ein Ereignis und einen Datenwert transportiert. Dies ermöglicht eine lose Kopplung und einfaches Zusammenschalten mit anderen Adapter-Bausteinen.
- **Delegation an IEC-61499-Standard**: Die eigentliche Flip-Flop-Logik wird durch den Standard-FB `E_D_FF_ANY` realisiert, der für beliebige Datentypen ausgelegt ist („ANY“). Der `ALR_D_FF` erbt dadurch dessen Robustheit und Typneutralität.
- **Keine eigene Zustandsmaschine**: Der FB besitzt keinen internen Zustandsautomaten – die Zustandsverwaltung erfolgt vollständig innerhalb des eingebetteten `E_D_FF_ANY`.

Zustandsübersicht

Der Baustein hat keine expliziten Zustände. Die interne Logik (Flip-Flop) kennt lediglich zwei implizite Zustände:

- **Warten auf Taktereignis**: Der Ausgang behält den zuletzt gelatchten Wert.
- **Latch-Moment**: Bei einem Taktereignis wird der Eingangswert übernommen und der Ausgang aktualisiert.

Anwendungsszenarien

- **Datenspeicherung**: Festhalten eines digitalen Signals (z. B. Schalterstellung) bis zur nächsten Abfrage.
- **Synchronisation**: Übernahme von asynchronen Daten in einen getakteten Kontext.
- **Puffern**: Zwischenspeichern von Werten in Steuerungsabläufen, z. B. in Zustandsautomaten oder Sequenzsteuerungen.

Vergleich mit ähnlichen Bausteinen

- **`ALR_SR_FF`** (Set-Reset-Flip-Flop): Bietet zwei Steuereingänge (Set/Reset) statt einem Takteingang. Geeignet für prioritätsgesteuerte Anwendungen.
- **`ALR_JK_FF`** (JK-Flip-Flop): Komplexeres Schaltverhalten mit Toggeln und Halten. Flexibler, aber aufwändiger.
- **`E_D_FF_ANY`** (direkt): Bietet die gleiche Funktionalität, jedoch mit separaten Ereignis- und Dateneingängen/-ausgängen anstelle eines Adapters. Der `ALR_D_FF` vereinfacht die Kopplung in Adapter-basierten Frameworks.

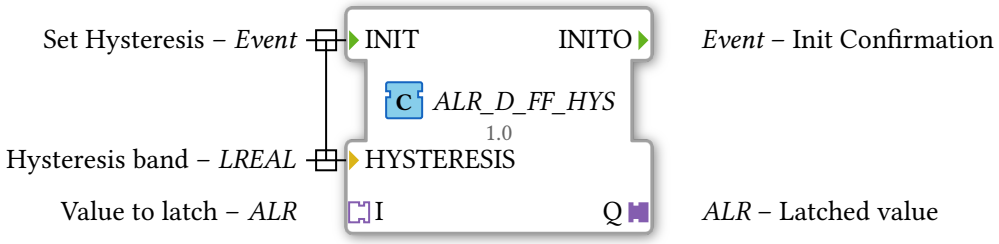
Fazit

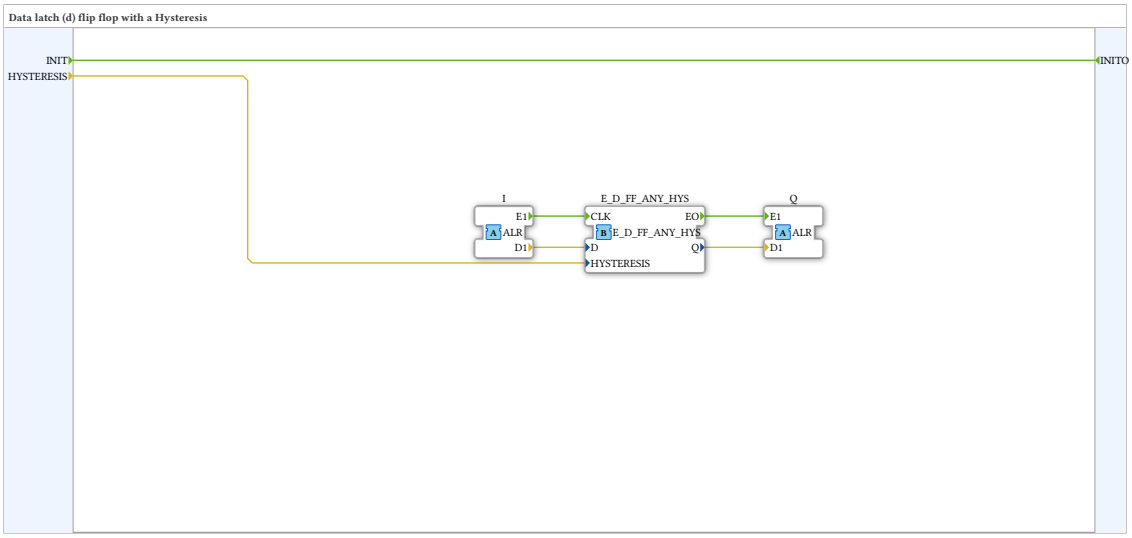
Der `ALR_D_FF` ist ein schlanker, aber zuverlässiger D-Flip-Flop-Baustein für den Einsatz in adapterorientierten 4diac-Anwendungen. Er kombiniert die bewährte IEC-61499-Standardlogik mit einer klar strukturierten Adapter-Schnittstelle und eignet sich besonders für modulare, wiederverwendbare Steuerungssysteme.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ALR_D_FF_HYS





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ALR_D_FF_HYS** realisiert ein Data-Latch (D-Flip-Flop) mit einer einstellbaren Hysterese. Er empfängt einen Datenwert über einen Adapter-Socket und gibt den gelatchten Wert über einen Adapter-Plug aus. Die Hysterese wird beim Initialisierungsereignis gesetzt und dient der Unterdrückung von Rauschen oder kleinen Schwankungen im Eingangssignal.

Schnittstellenstruktur

Ereignis-Eingänge

Event	Typ	Beschreibung
INIT	EInit	Setzt die Hysterese (mit dem Datenwert `HYSTERESIS`) und startet die Initialisierung.

Ereignis-Ausgänge

Event	Typ	Beschreibung
INITO	EInit	Bestätigt die erfolgreiche Initialisierung.

Daten-Eingänge

Variable	Typ	Beschreibung
HYSTERESIS	LREAL	Größe des Hysterese-Bandes (wird beim `INIT`-Event übernommen).

Daten-Ausgänge

Der FB besitzt keine direkten Daten-Ausgänge. Der gelatchte Wert wird ausschließlich über den Adapter-Plug `Q` ausgegeben.

Adapter

Bezeichnung	Typ	Richtung	Beschreibung
I (Socket)	`adapter::types::unidirectional::ALR`	Eingang	Empfängt den zu latchnden Wert über das Ereignis `I.E1` und den zugehörigen Datenwert `I.D1`.
Q (Plug)	`adapter::types::unidirectional::ALR`	Ausgang	Gibt den gelatchten Wert über das Ereignis `Q.E1` und den Datenwert `Q.D1` aus.

Der verwendete Adapter **ALR** ist ein unidirektionaler Daten-Adapter mit einem Ereignis (`E1`) und einem Datenwert (`D1`).

Funktionsweise

Der FB arbeitet intern mit dem Baustein `E_D_FF_ANY_HYS`, der die eigentliche Latch- und Hysterese-Logik implementiert. Die Abläufe sind:

1. **Initialisierung**: Beim Eintreffen von `INIT` wird der Wert von `HYSTERESIS` an den internen Baustein weitergeleitet und das Ausgangsereignis `INITO` sofort bestätigt.
2. **Latch-Vorgang**: Bei jeder positiven Flanke des Ereignisses `I.E1` (vom Socket) wird der aktuelle Datenwert `I.D1` übernommen. Der gelatchte Wert wird jedoch erst ausgegeben, wenn er das durch die Hysterese bestimmte Band verlässt. Dadurch werden kleine Änderungen um den aktuellen Latch-Wert unterdrückt.
3. **Ausgabe**: Sobald der gelatchte Wert das Hysterese-Band verlässt, erzeugt der interne Baustein ein Ereignis auf `EO`, welches als `Q.E1` nach außen gegeben wird, und der neue Wert steht auf `Q.D1` zur Verfügung.

Technische Besonderheiten

- Die Hysterese wird ausschließlich beim `INIT`-Ereignis gesetzt und bleibt während des Betriebs konstant.
- Der FB verwendet keinen eigenen Zustandsautomat, sondern delegiert die gesamte Logik an den internen Baustein `E_D_FF_ANY_HYS`.
- Der initiale Zustand des Latch ist nicht definiert; eine Initialisierung mit `INIT` ist zwingend erforderlich, bevor gültige Latch-Vorgänge stattfinden können.

Zustandsübersicht

Da der FB das Verhalten über einen internen Baustein realisiert, ergibt sich kein eigenständiges Zustandsdiagramm. Der interne Baustein `E_D_FF_ANY_HYS` arbeitet typischerweise mit folgenden Zuständen:

- **Initial** – Warten auf die Hysterese-Einstellung.
- **Normal** – Latch-Betrieb mit Prüfung des Hysterese-Bandes (verzögerte Aktualisierung des Ausgangs).
- **Ausgabe** – Senden des neuen gelatchten Wertes über den Ausgangs-Adapter.

Anwendungsszenarien

- **Signalaufbereitung**: Latch von Messwerten mit Rauschunterdrückung in der Automatisierungstechnik.
- **Entprellung**: Verarbeitung von Schaltsignalen, bei denen kurze Impulse oder Prellen unterdrückt werden sollen.
- **Schwellwertüberwachung**: Erfassung analoger Werte, die nur ober- oder unterhalb eines Hysterese-Bandes einen neuen gültigen Zustand annehmen.

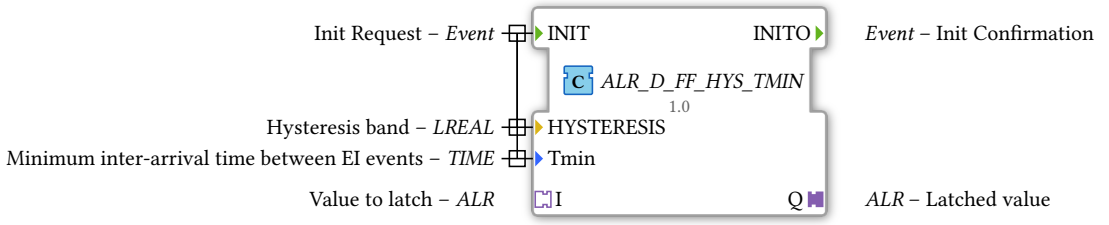
Vergleich mit ähnlichen Bausteinen

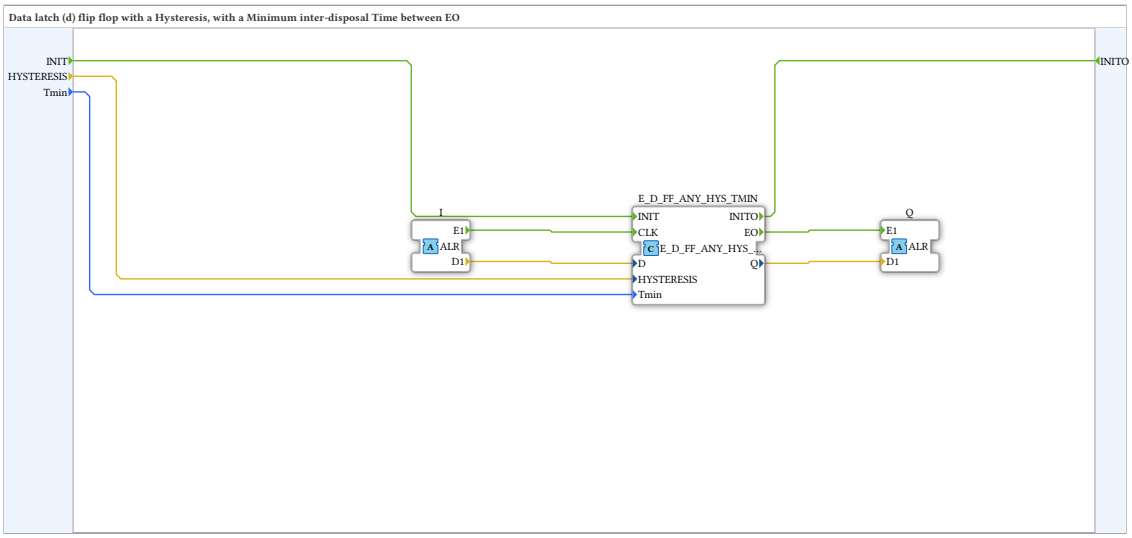
Baustein	Eigenschaft
Einfaches D-Flip-Flop (z. B. `E_D_FF`)	Latch ohne Hysterese – jede Änderung am Eingang wird sofort übernommen.
D-Flip-Flop mit Totband (`E_D_FF_TOTBAND`)	Hat ein Totband, das Änderungen blockiert, solange der Wert innerhalb des Bandes bleibt.
ALR_D_FF_HYS	Hysterese wird nur in einer Richtung wirksam (Schwellwertverhalten), typisch zur Vermeidung von Flattern.

Fazit

Der ALR_D_FF_HYS ist ein spezialisierter Funktionsblock für Anwendungen, bei denen ein Datenwert gelatcht und gleichzeitig eine Hysterese zur Störunterdrückung angewendet werden soll. Die Kapselung der Logik in einen internen Baustein hält die Schnittstelle einfach und ermöglicht eine flexible Nutzung über die Adapter-Schnittstelle.

ALR_D_FF_HYS_TMIN





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ALR_D_FF_HYS_TMIN** realisiert ein taktgesteuertes D-Latch (Flipflop) mit einer einstellbaren Hysterese für die Datenübernahme und einer minimalen Zeitverzögerung zwischen aufeinanderfolgenden Ausgangsereignissen. Er verarbeitet einen analogen oder booleschen Wert über einen Adapter-Eingang und gibt den zwischengespeicherten Wert über einen Adapter-Ausgang aus. Die Hysterese verhindert ein ungewolltes Umschalten bei schwankenden Eingangswerten, während die Mindestzeit (Tmin) die Ereignisfrequenz am Ausgang begrenzt.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar
INIT	EInit	Initialisierungsanforderung (setzt Hysterese und Tmin)

Ereignis-Ausgänge

Ereignis	Typ	Kommentar
INITO	EInit	Initialisierungsbestätigung

Daten-Eingänge

Name	Typ	Kommentar
HYSTERESIS	LREAL	Hysterese-Band für die Datenübernahme
Tmin	TIME	Mindestzeit zwischen zwei Ausgangsereignissen

Daten-Ausgänge

Der FB besitzt keine eigenen Daten-Ausgänge. Der latched Wert wird über den Adapter-Ausgang `Q` bereitgestellt.

Adapter

Richtung	Name	Typ (unidirektional)	Kommentar
Socket	I	ALR	Eingangswert, der gelatcht werden soll
Plug	Q	ALR	Ausgangswert (latched)

Funktionsweise

Der FB ist als zusammengesetzter Baustein realisiert, der intern einen FB vom Typ `E_D_FF_ANY_HYS_TMIN` (aus der Bibliothek `logiBUS::signalprocessing::hysteresis`) verwendet. Die Funktionsweise im Einzelnen:

1. **Initialisierung (INIT)**: Beim Ereignis `INIT` werden die Parameter `HYSTERESIS` und `Tmin` an den internen FB übergeben. Der interne FB wird initialisiert und bestätigt mit `INITO`.
2. **Datenübernahme**: Ein Ereignis am Adapter-Eingang `I.E1` triggert den Takt (`CLK`) des internen Flipflops. Gleichzeitig wird der Datenwert `I.D1` als neuer Eingang `D` übernommen.
3. **Hysterese**: Die Übernahme des Wertes erfolgt nur, wenn die Abweichung zum aktuell gelatchten Wert größer oder gleich der eingestellten Hysterese ist. Dies verhindert ständiges Umschalten bei kleinen Schwankungen.
4. **Mindestzeit (Tmin)**: Nach jedem ausgegebenen Ereignis (`E0`) wird eine Verzögerung von `Tmin` eingehalten, bevor ein weiteres Ausgangsereignis erzeugt werden kann. Ereignisse, die innerhalb dieser Sperrfrist eintreffen, werden ignoriert.
5. **Ausgabe**: Der gelatchte Wert wird über den Adapter-Ausgang `Q.D1` bereitgestellt, und ein Ereignis (`Q.E1`) signalisiert die Aktualisierung.

Technische Besonderheiten

- **Hysterese**: Der Baustein arbeitet mit einem einstellbaren Hysterese-Band. Die tatsächliche Umsetzung (ob symmetrisch oder mit Schwellwerten) hängt vom internen FB ab. Üblich ist eine Schmitt-Trigger-Charakteristik.
- **Zeitgesteuerte Ausgabe**: `Tmin` begrenzt die maximale Ausgabefrequenz. Dies ist nützlich, um nachgeschaltete Komponenten zu entlasten oder eine minimale Signalverweildauer zu garantieren.
- **Adapter-Schnittstelle**: Die Verwendung von Adaptern (`ALR`) ermöglicht eine lose Kopplung zwischen Datenwert und Ereignis – ein Ereignis transportiert implizit den zugehörigen Datenwert.
- **Wiederverwendbarkeit**: Der FB ist aus einem standardisierten Hysterese-Flipflop aufgebaut und kann leicht an andere Datenformate angepasst werden.

Zustandsübersicht

Da der FB aus mehreren internen Zuständen besteht (z. B. Warten auf Takt, Hysterese-Prüfung, Warten auf Tmin), wird hier eine vereinfachte Zustandsbeschreibung gegeben:

1. **Initialisiert** (nach INIT und INITO)
2. **Bereit** – wartet auf ein Ereignis am Eingang `I.E1`
3. **Hysterese-Prüfung** – vergleicht neuen Wert mit gelatchtem Wert
   - Bei Überschreitung: Übernahme des neuen Wertes, Übergang zu **Warten auf Tmin**
   - Sonst: zurück zu **Bereit** ohne Ausgabe
4. **Warten auf Tmin** – nach Ablauf der Mindestzeit wird das Ausgangsereignis `Q.E1` erzeugt und der gelatchte Wert auf `Q.D1` ausgegeben; danach zurück zu **Bereit**

Anwendungsszenarien

- **Analoge Schaltschwellen**: Überwachung von Sensoren (z. B. Temperatur, Druck) mit einstellbarer Hysterese und minimaler Schaltpause, um Prellen oder Rauschen zu unterdrücken.
- **Digitale Signalaufbereitung**: Verarbeitung von Rechtecksignalen mit Rauschen, wobei die Hysterese Flanken stabilisiert und Tmin eine definierte Impulsdauer erzwingt.
- **Schrittmotorensteuerung**: Latch von Positionswerten mit Hysterese gegen mechanische Schwingungen, begleitet von zeitlicher Entprellung der Ausgabebefehle.
- **Sicherheitskritische Systeme**: Mindestzeit zwischen Ausgabeänderungen verhindert zu schnelle Zustandswechsel und schützt nachfolgende Aktoren.

Vergleich mit ähnlichen Bausteinen

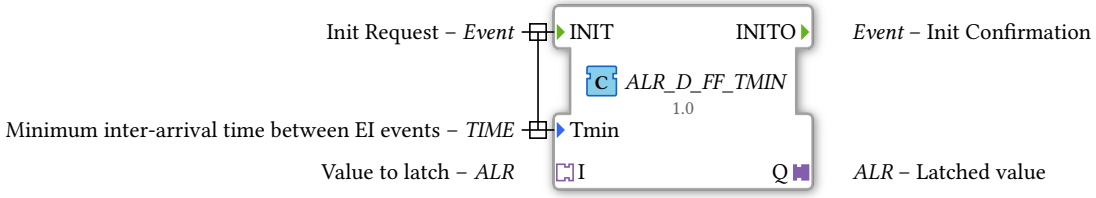
Baustein	Eigenschaften	Unterschied zu ALR_D_FF_HYS_TMIN
Einfaches D-Flipflop (D_FF)	Keine Hysterese, keine Zeitsteuerung	Fehlende Signalstabilisierung und Frequenzbegrenzung
Schmitt-Trigger (z. B. S_TRIG)	Nur Hysterese, keine Zeitsteuerung, oft nur binär	Unterscheidet sich bei analogen Werten und erzwungenen Pausen
Flankengesteuertes Latch mit Totzeit	Zeitsteuerung (Totzeit) vorhanden, aber meist ohne Hysterese	Kombination beider Effekte (Hysterese + Tmin)
Filterbaustein (z. B. FILTER)	Rauschunterdrückung z. B. als Tiefpass, keine diskrete Zustandsänderung	Explizites Latch mit Ereignisausgabe und einstellbarer Hysterese

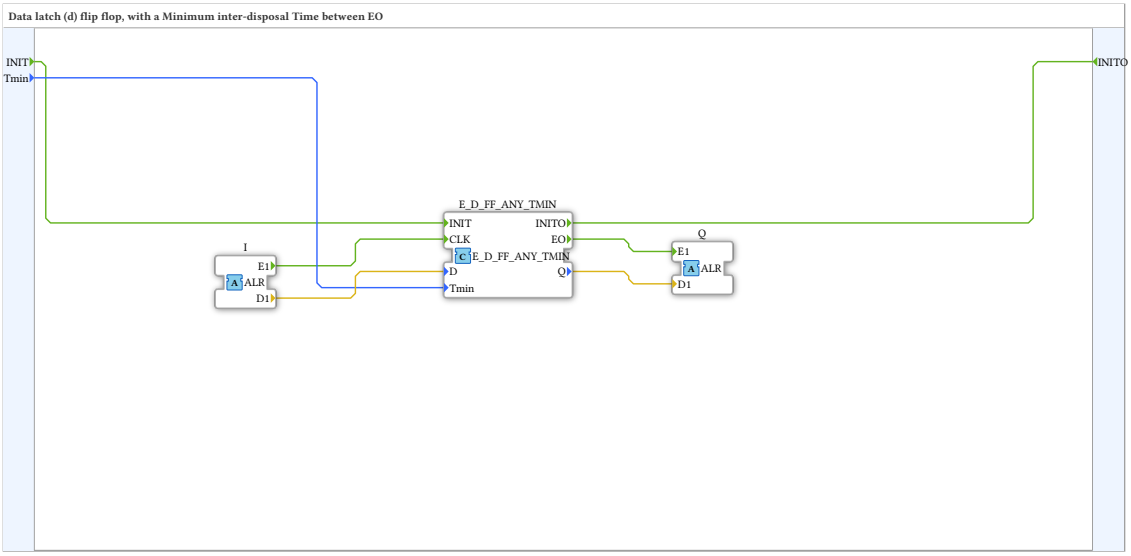
Der vorliegende Baustein vereint die Vorteile einer Hysterese mit einer minimalen Ereignisabstandsfunktion und ist damit besonders für raue industrielle Umgebungen geeignet.

Fazit

Der **ALR_D_FF_HYS_TMIN** ist ein robuster, konfigurierbarer Funktionsblock für die zuverlässige Signalverarbeitung mit Hysterese und zeitlicher Entprellung. Er eignet sich für alle Anwendungen, bei denen ein weicher Schwellwert und eine Begrenzung der Ausgabe-Ereignisfrequenz erforderlich sind. Die Adapter-basierte Schnittstelle erleichtert die Integration in komplexe Steuerungsarchitekturen.

ALR_D_FF_TMIN





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ALR_D_FF_TMIN** realisiert einen taktflankengesteuerten Data-Latch-Flipflop (D-Flipflop) mit einer minimalen Verzögerungszeit zwischen zwei aufeinanderfolgenden Ereignisausgängen (Minimum inter-disposal Time). Er ermöglicht das latchen eines Datenwerts bei Empfang eines Ereignisses und stellt sicher, dass der Ausgangsereignis-Stoß nicht schneller als ein definierbares Zeitintervall erfolgt. Dies ist besonders nützlich, um nachgeschaltete Systeme vor Überlastung zu schützen oder zeitliche Mindestabstände in der Ereigniskommunikation einzuhalten.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar
INIT	EInit	Initialisierungsanforderung; setzt die Mindestzeit `Tmin`

Ereignis-Ausgänge

Ereignis	Typ	Kommentar
INITO	EInit	Bestätigung der erfolgreichen Initialisierung

Daten-Eingänge

Name	Typ	Kommentar
Tmin	TIME	Minimale Wartezeit zwischen zwei EO-Ereignissen

Daten-Ausgänge

Keine eigenen Datenausgänge vorhanden – der gelatchte Wert wird über den Adapterausgang bereitgestellt.

Adapter

Adapter	Richtung	Typ	Kommentar
I	Socket	ALR (unidirektional)	Eingangswert (`D1`) und Takt (`E1`)
Q	Plug	ALR (unidirektional)	Ausgangswert (`D1`) und Quittung (`E1`)

Der Adapter `ALR` (unidirectional) besitzt einen Ereigniseingang `E1` und einen Dateneingang `D1` (Socket-Seite) bzw. einen Ereignisausgang `E1` und Datenausgang `D1` (Plug-Seite). – Die genaue Semantik von `E1` und `D1` ist anwendungsspezifisch; hier dient `E1` als Takt und `D1` als der zu latchende Datenwert.

Funktionsweise

Nach der Initialisierung (INIT) arbeitet der Baustein wie folgt:

1. **Ereignis am Eingangsadapter `I.E1`** löst den Latch-Vorgang aus. Der aktuelle Wert `I.D1` wird intern übernommen.
2. Der gelatchte Wert wird sofort am Ausgangsadapter `Q.D1` ausgegeben.
3. Gleichzeitig wird ein Ereignis `Q.E1` ausgelöst – jedoch erst, wenn seit dem letzten Ausgangsereignis mindestens die Zeit `Tmin` vergangen ist.
   Falls ein weiteres `I.E1`-Ereignis innerhalb der Wartezeit eintrifft, wird es ignoriert (bzw. erst nach Ablauf von `Tmin` verarbeitet). Die minimale Verzögerung wird durch den internen Funktionsblock `E_D_FF_ANY_TMIN` realisiert.

Der Datenwert wird also bei jedem gültigen Takt aktualisiert, die Ausgabe des Ereignisses jedoch zeitlich gestreckt.

Technische Besonderheiten

- **Zusammengesetzter FB**: Der Baustein ist als Composite-FB realisiert und verwendet intern den FB `iec61499::events::E_D_FF_ANY_TMIN`. Die Logik des eigentlichen Flipflops und die Zeitsteuerung wird dort gekapselt.
- **Universelles Zeitintervall**: `Tmin` kann zur Laufzeit über INIT gesetzt werden und ist nicht fest vorgegeben.
- **Gepufferte Ereignisse**: Da die Ereignisausgabe verzögert wird, können ankommende Taktereignisse während der Wartezeit verloren gehen, es sei denn, der interne FB puffert sie (hier nicht spezifiziert). Typisch ist ein einfaches Ignorieren.
- **Adapterbasierte Schnittstelle**: Die Verwendung von Adaptern ermöglicht eine flexible Verschaltung in modularen Steuerungsanwendungen, ohne dass die Anschlüsse fest verdrahtet werden müssen.

Zustandsübersicht

Der Baustein besitzt keine expliziten Zustände, sondern arbeitet ereignisgesteuert:

- **Idle**: Warten auf `I.E1` (nach INIT).
- **Latchen**: Bei `I.E1` wird der Datenwert übernommen und `Q.E1` nach Ablauf von `Tmin` ausgelöst.
- **Blockiert**: Während `Tmin` noch nicht abgelaufen ist, werden weitere `I.E1` ignoriert.
- Nach Ablauf der Wartezeit kehrt der Baustein in den idle-Zustand zurück.

Anwendungsszenarien

- **Entprellung von Sensorsignalen**: Ein Sensor liefert Ereignisse schneller als die nachfolgende Steuerung verarbeiten kann. `Tmin` legt den Mindestabstand fest.
- **Pulsweitenbegrenzung**: Sicherstellung, dass Aktorsteuerungen nicht zu schnell hintereinander geschaltet werden (z. B. Ventile, Motoren).
- **Taktsynchronisation**: Wenn ein Datenwert mit einem asynchronen Takt gelatcht und erst nach einer definierten Totzeit weitergegeben werden soll.
- **Sicherheitsfunktionen**: Verhindern von „Rapid-Fire“-Befehlen in kritischen Anwendungen.

Vergleich mit ähnlichen Bausteinen

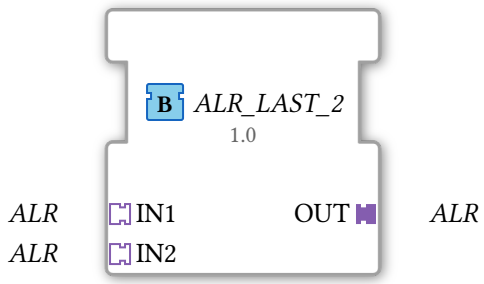
- **E_D_FF_ANY_TMIN**: Ist der interne Baustein – er bietet die gleiche Funktionalität, jedoch ohne die Adapter-Schnittstelle. `ALR_D_FF_TMIN` kapselt diesen in eine anwendungsfreundlichere, adapterbasierte Form.
- **Standard IEC 61499 D-Flipflops**: Diese haben oft keine zeitliche Steuerung der Ausgangsereignisse. Der hier beschriebene FB erweitert sie um die `Tmin`-Verzögerung.
- **FB mit TON/Timer**: Ein reiner Timer würde nur zeitverzögert auslösen, aber keinen Datenwert halten. Die Kombination mit einem Latch ist spezifisch.

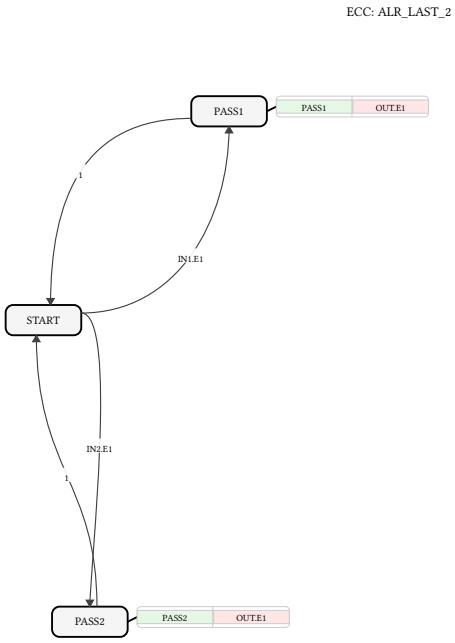
Fazit

Der Funktionsblock `ALR_D_FF_TMIN` vereint die Eigenschaften eines D-Flipflops mit einer einstellbaren Mindestzeit zwischen Ereignisausgaben. Durch die Adaptertechnik und die einfache Initialisierung lässt er sich flexibel in IoT-, Automatisierungs- oder Steuerungsumgebungen einsetzen. Besonders geeignet ist

er für Anwendungen, die eine zeitliche Entkopplung zwischen Datenerfassung und Ereignisweitergabe erfordern.

ALR_LAST_2





Einleitung

Der ALR_LAST_2 Funktionsblock führt zwei gleichartige LREAL-Adapter-Signale (**IN1**, **IN2**) auf einen gemeinsamen Ausgang (**OUT**) zusammen, nach dem Prinzip **Last Writer Wins**: welcher der beiden Sockets zuletzt ein Ereignis geschrieben hat, gewinnt, und sein Datenwert wird sofort und unverändert an OUT durchgereicht. Es handelt sich ausdrücklich nicht um ein naives Zusammenführen (z. B. Mittelwert oder ODER-Verknüpfung), sondern um eine reine Arbitrierung nach zeitlicher Reihenfolge.

Schnittstellenstruktur

Adapter

Eingangsadapter (Sockets):

- **IN1**: LREAL-Adapter (unidirectional) - erste Signalquelle
- **IN2**: LREAL-Adapter (unidirectional) - zweite Signalquelle

Ausgangsadapter (Plug):

- **OUT**: LREAL-Adapter (unidirectional) - zusammengeführtes Signal

Funktionsweise

Der Baustein ist als einfacher Basic-FB mit einer 3-Zustands-ECC umgesetzt (START, PASS1, PASS2). Trifft an IN1 ein Ereignis ein, wechselt die ECC nach PASS1: der aktuelle Datenwert von IN1 wird nach OUT kopiert und das Adapter-Ereignis an OUT gesendet. Trifft an IN2 ein Ereignis ein, geschieht dasselbe symmetrisch über PASS2. Nach jeder Durchreichung kehrt die ECC sofort wieder nach START zurück und ist bereit für das nächste Ereignis von IN1 oder IN2.

Technische Besonderheiten

- Last-Writer-Wins-Semantik statt naiver Datenverknüpfung
- Einfacher Basic-FB (ECC mit 3 Zuständen), kein zusätzlicher Zwischenspeicher nötig
- Keine Signalverzögerung: der Wert wird im selben Zyklus durchgereicht, in dem das auslösende Ereignis eintrifft
- Beide Eingänge sind gleichberechtigt - es gibt keine feste Priorität außer der zeitlichen Reihenfolge

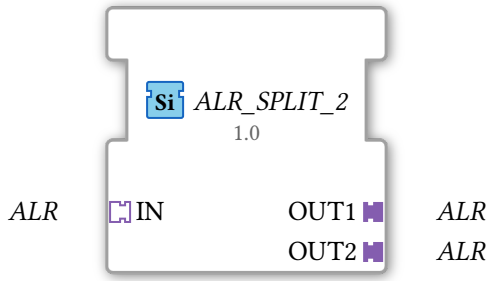
Anwendungsszenarien

- Zusammenführen zweier gleichwertiger LREAL-Signalquellen zu einem einzigen nachgeschalteten Adapterpfad (z. B. zwei redundante Sensoren, oder ein manueller und ein automatischer Sollwertgeber)
- Arbitrierung zwischen zwei Bedienstellen, die denselben LREAL-Wert schreiben dürfen
- Vereinfachung von Netzwerken, die sonst ein `F_SEL`/`E_RS`-Konstrukt inline verdrahten müssten

⚖️ Vergleich mit ähnlichen Bausteinen

ALR_LAST_2 ist die Umkehrung von `ALR_SPLIT_2`: ALR_SPLIT_2 verteilt 1 Eingang auf 2 Ausgänge, ALR_LAST_2 führt 2 Eingänge auf 1 Ausgang zusammen. Für Mischbetrieb zwischen einem Daten- und einem reinen Ereignisadapter (z. B. AX + AE) ist stattdessen `AX_AE_MERGE` gedacht, das nur das Ereignis mischt, ohne den Datenwert zu verändern.

ALR_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock ALR_SPLIT_2 dient zum Aufteilen eines eingehenden ALR-Adapter-Signals in zwei identische Ausgangssignale. Er ist generisch ausgelegt (`GEN_ALR_SPLIT`) und eignet sich für die Verteilung von Alarm- oder Lebenszyklus-Signalen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

- **IN** (Socket): Typ `adapter::types::unidirectional::ALR` – Eingangsadapter für das ALR-Signal.
- **OUT1** (Plug): Typ `adapter::types::unidirectional::ALR` – erster Ausgangsadapter.
- **OUT2** (Plug): Typ `adapter::types::unidirectional::ALR` – zweiter Ausgangsadapter.

Funktionsweise

Der Baustein leitet das über den Socket `IN` empfangene ALR-Signal unverändert an beide Plugs `OUT1` und `OUT2` weiter. Es findet keine Transformation oder Verzögerung statt. Das Signal wird passiv aufgeteilt.

Technische Besonderheiten

- Der Baustein ist als generischer Funktionsblock implementiert (Attribut `GenericClassName = 'GEN_ALR_SPLIT'`).
- Verwendet werden unidirektionale Adapter des Typs `ALR`, die nur eine Datenflussrichtung unterstützen.
- Keine internen Timer, Zustände oder Ereignisse – reine Signalverteilung.

Zustandsübersicht

Der Baustein besitzt keine Zustandsautomaten. Das Verhalten ist ausschließlich durch die Adapterdefinition bestimmt.

Anwendungsszenarien

- Verteilung eines ALR-Signals (z.B. Alarm, Temperaturgrenzwert) an zwei unterschiedliche Empfänger.
- Als Bestandteil einer Logik, die ein Alarmsignal mehrfach auswerten muss.
- Einsatz in Steuerungssystemen, die ein Signal parallel an zwei Funktionsbausteine weiterleiten.

Vergleich mit ähnlichen Bausteinen

- **ALR_SPLIT_2** teilt ein ALR-Signal auf zwei Ausgänge; vergleichbar mit Ereignis-Splittern wie `E_SWITCH` oder `E_SPLIT`, jedoch für ALR-Adapter.
- Im Gegensatz zu datenbasierten Splittern (`MUX`, `F_SPLIT`) findet hier keine Datenmanipulation statt.
- Es existieren möglicherweise Varianten mit mehr Ausgängen (z.B. `ALR_SPLIT_4`), die eine höhere Anzahl an Verteilungen erlauben.
- `ALR_SPLIT_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

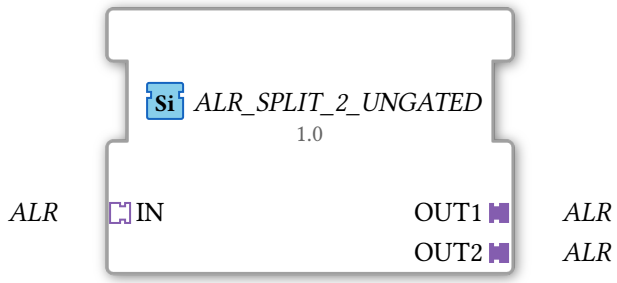
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

`ALR_SPLIT_2` ist ein einfacher und effektiver Baustein zur Aufteilung eines eingehenden ALR-Adapter-Signals auf zwei Ausgänge. Aufgrund seiner generischen Natur und der fehlenden Logik eignet er sich ideal für die saubere Signalverteilung in IEC 61499-Systemen.

ALR_SPLIT_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALR_SPLIT_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock ALR_SPLIT_2_UNGATED dient zum Aufteilen eines eingehenden ALR-Adapter-Signals in zwei identische Ausgangssignale. Er ist generisch ausgelegt (`GEN_ALR_SPLIT`) und eignet sich für die Verteilung von Alarm- oder Lebenszyklus-Signalen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

- **IN** (Socket): Typ `adapter::types::unidirectional::ALR` – Eingangsadapter für das ALR-Signal.
- **OUT1** (Plug): Typ `adapter::types::unidirectional::ALR` – erster Ausgangsadapter.
- **OUT2** (Plug): Typ `adapter::types::unidirectional::ALR` – zweiter Ausgangsadapter.

Funktionsweise

Der Baustein leitet das über den Socket `IN` empfangene ALR-Signal unverändert an beide Plugs `OUT1` und `OUT2` weiter. Es findet keine Transformation oder Verzögerung statt. Das Signal wird passiv aufgeteilt.

Technische Besonderheiten

- Der Baustein ist als generischer Funktionsblock implementiert (Attribut `GenericClassName = 'GEN_ALR_SPLIT'`).
- Verwendet werden unidirektionale Adapter des Typs `ALR`, die nur eine Datenflussrichtung unterstützen.
- Keine internen Timer, Zustände oder Ereignisse – reine Signalverteilung.

Zustandsübersicht

Der Baustein besitzt keine Zustandsautomaten. Das Verhalten ist ausschließlich durch die Adapterdefinition bestimmt.

Anwendungsszenarien

- Verteilung eines ALR-Signals (z.B. Alarm, Temperaturgrenzwert) an zwei unterschiedliche Empfänger.
- Als Bestandteil einer Logik, die ein Alarmsignal mehrfach auswerten muss.
- Einsatz in Steuerungssystemen, die ein Signal parallel an zwei Funktionsbausteine weiterleiten.

Vergleich mit ähnlichen Bausteinen

- **ALR_SPLIT_2_UNGATED** teilt ein ALR-Signal auf zwei Ausgänge; vergleichbar mit Ereignis-Splittern wie `E_SWITCH` oder `E_SPLIT`, jedoch für ALR-Adapter.

- Im Gegensatz zu datenbasierten Splittern (`MUX`, `F_SPLIT`) findet hier keine Datenmanipulation statt.
- Es existieren möglicherweise Varianten mit mehr Ausgängen (z.B. `ALR_SPLIT_4`), die eine höhere Anzahl an Verteilungen erlauben.
- **`ALR_SPLIT_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

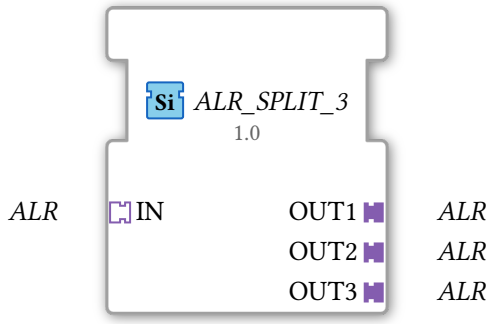
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

`ALR_SPLIT_2_UNGATED` ist ein einfacher und effektiver Baustein zur Aufteilung eines eingehenden ALR-Adapter-Signals auf zwei Ausgänge. Aufgrund seiner generischen Natur und der fehlenden Logik eignet er sich ideal für die saubere Signalverteilung in IEC 61499-Systemen.

ALR_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ALR_SPLIT_3** dient als generischer SPLIT-Baustein, der ein eingehendes ALR-Signal (Alarm- oder Ereignissignal) auf drei identische Ausgänge aufteilt. Er wird vor allem in Automatisierungssystemen eingesetzt, wo ein einzelnes Signal parallel an mehrere nachfolgende Komponenten verteilt werden muss, ohne die Signalqualität oder -logik zu verändern.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

- **IN** – *Socket* (Eingang) vom Typ `adapter::types::unidirectional::ALR` Nimmt das zu verteilende ALR-Signal entgegen.
- **OUT1** – *Plug* (Ausgang 1) vom Typ `adapter::types::unidirectional::ALR` Gibt das eingehende Signal unverändert weiter.
- **OUT2** – *Plug* (Ausgang 2) vom Typ `adapter::types::unidirectional::ALR` Gibt das eingehende Signal unverändert weiter.

- **OUT3** – *Plug* (Ausgang 3) vom Typ `adapter::types::unidirectional::ALR` Gibt das eingehende Signal unverändert weiter.

Funktionsweise

Der Baustein arbeitet als reiner Signalverteiler. Sobald am Adapter-Socket **IN** ein ALR-Signal anliegt (z. B. ein Alarmzustand, ein Ereignis oder eine Aktivierung), wird dieses Signal ohne Verzögerung oder Manipulation an alle drei Adapter-Plugs **OUT1**, **OUT2** und **OUT3** weitergeleitet. Es findet keine logische Verarbeitung oder Filterung statt. Die Verteilung erfolgt passiv über die Verbindungslogik der 4diac-IDE.

Technische Besonderheiten

- **Generischer Funktionsblock** – Über die Attribute `eclipse4diac::core::GenericClassName` und `eclipse4diac::core::TypeHash` kann der Baustein als generische Instanz angelegt werden. Dies erlaubt eine einfache Wiederverwendung und Parametrisierung in verschiedenen Projekten.
- **Keine eigene Zustandsmaschine** – Der Baustein besitzt kein ECC (Execution Control Chart), da das Verhalten rein durch die Adapter-Konnektivität definiert wird.
- **Unidirektionale Adapter** – Alle verwendeten ALR-Adapter sind vom Typ `unidirectional`; die Signalrichtung ist festgelegt (Eingang ⟶ Ausgänge).

Zustandsübersicht

Der Baustein definiert keine internen Zustände. Das Ausgangssignal entspricht stets momentan dem Eingangssignal. Eine separate Zustandsbeschreibung ist daher nicht erforderlich.

Anwendungsszenarien

- **Alarmverteilung** – Ein zentraler Alarmmelder (z. B. übergeordnetes Leitsystem) wird auf drei unabhängige Steuerungseinheiten aufgeschaltet, die parallel auf den Alarm reagieren sollen.
- **Signal-Multiplex** – Ein boolesches oder wertbasiertes Signal aus einem Sensor wird an mehrere Aktoren oder Visualisierungen verteilt.
- **Redundanzarchitekturen** – Das gleiche Signal wird an zwei oder drei unabhängige Pfade geschickt, um eine Ausfallsicherheit zu erreichen.

Vergleich mit ähnlichen Bausteinen

- **ALR_SPLIT_2** – Verteilt ein Signal auf zwei Ausgänge (identische Funktion, aber weniger Ausgänge).
- **ALR_SPLIT_4** – Verteilt ein Signal auf vier Ausgänge.
- **Manuelle Kopplung** – Ohne SPLIT-Baustein müsste das Signal mehrfach in der Applikation abgegriffen oder über eine Sammelleitung verteilt werden, was die Übersichtlichkeit und Wartbarkeit verschlechtert.
- `ALR_SPLIT_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

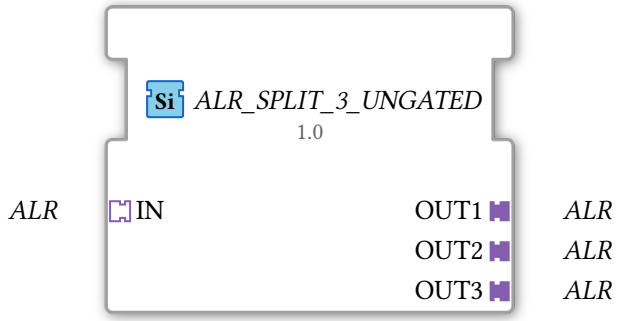
Der **ALR_SPLIT_3** ist ein einfacher, aber nützlicher generischer Baustein zur Vervielfachung eines ALR-Signals. Er erleichtert die strukturierte und wiederverwendbare Verschaltung in Automatisierungslösungen, bei denen ein Signal an mehrere Empfänger weitergegeben werden muss, ohne zusätzliche Logik oder Latenz.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ALR_SPLIT_3_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `ALR_SPLIT_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ALR_SPLIT_3_UNGATED** dient als generischer SPLIT-Baustein, der ein eingehendes ALR-Signal (Alarm- oder Ereignissignal) auf drei identische Ausgänge aufteilt. Er wird vor allem in Automatisierungssystemen eingesetzt, wo ein einzelnes Signal parallel an mehrere nachfolgende Komponenten verteilt werden muss, ohne die Signalqualität oder -logik zu verändern.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

- **IN** – *Socket* (Eingang) vom Typ `adapter::types::unidirectional::ALR` Nimmt das zu verteilende ALR-Signal entgegen.
- **OUT1** – *Plug* (Ausgang 1) vom Typ `adapter::types::unidirectional::ALR` Gibt das eingehende Signal unverändert weiter.
- **OUT2** – *Plug* (Ausgang 2) vom Typ `adapter::types::unidirectional::ALR` Gibt das eingehende Signal unverändert weiter.
- **OUT3** – *Plug* (Ausgang 3) vom Typ `adapter::types::unidirectional::ALR` Gibt das eingehende Signal unverändert weiter.

Funktionsweise

Der Baustein arbeitet als reiner Signalverteiler. Sobald am Adapter-Socket **IN** ein ALR-Signal anliegt (z. B. ein Alarmzustand, ein Ereignis oder eine Aktivierung), wird dieses Signal ohne Verzögerung oder

Manipulation an alle drei Adapter-Plugs **OUT1**, **OUT2** und **OUT3** weitergeleitet. Es findet keine logische Verarbeitung oder Filterung statt. Die Verteilung erfolgt passiv über die Verbindungslogik der 4diac-IDE.

Technische Besonderheiten

- **Generischer Funktionsblock** – Über die Attribute `eclipse4diac::core::GenericClassName` und `eclipse4diac::core::TypeHash` kann der Baustein als generische Instanz angelegt werden. Dies erlaubt eine einfache Wiederverwendung und Parametrisierung in verschiedenen Projekten.
- **Keine eigene Zustandsmaschine** – Der Baustein besitzt kein ECC (Execution Control Chart), da das Verhalten rein durch die Adapter-Konnektivität definiert wird.
- **Unidirektionale Adapter** – Alle verwendeten ALR-Adapter sind vom Typ `unidirectional`; die Signalrichtung ist festgelegt (Eingang → Ausgänge).

Zustandsübersicht

Der Baustein definiert keine internen Zustände. Das Ausgangssignal entspricht stets momentan dem Eingangssignal. Eine separate Zustandsbeschreibung ist daher nicht erforderlich.

Anwendungsszenarien

- **Alarmverteilung** – Ein zentraler Alarmmelder (z. B. übergeordnetes Leitsystem) wird auf drei unabhängige Steuerungseinheiten aufgeschaltet, die parallel auf den Alarm reagieren sollen.
- **Signal-Multiplex** – Ein boolesches oder wertbasiertes Signal aus einem Sensor wird an mehrere Aktoren oder Visualisierungen verteilt.
- **Redundanzarchitekturen** – Das gleiche Signal wird an zwei oder drei unabhängige Pfade geschickt, um eine Ausfallsicherheit zu erreichen.

Vergleich mit ähnlichen Bausteinen

- **ALR_SPLIT_2** – Verteilt ein Signal auf zwei Ausgänge (identische Funktion, aber weniger Ausgänge).
- **ALR_SPLIT_4** – Verteilt ein Signal auf vier Ausgänge.
- **Manuelle Kopplung** – Ohne SPLIT-Baustein müsste das Signal mehrfach in der Applikation abgegriffen oder über eine Sammelleitung verteilt werden, was die Übersichtlichkeit und Wartbarkeit verschlechtert.
- `ALR_SPLIT_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

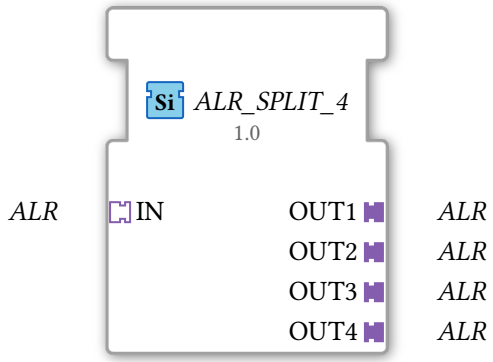
Fazit

Der **ALR_SPLIT_3_UNGATED** ist ein einfacher, aber nützlicher generischer Baustein zur Vervielfachung eines ALR-Signals. Er erleichtert die strukturierte und wiederverwendbare Verschaltung in Automatisierungslösungen, bei denen ein Signal an mehrere Empfänger weitergegeben werden muss, ohne zusätzliche Logik oder Latenz.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ALR_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **ALR_SPLIT_4** dient der Aufteilung eines eingehenden ALR-Adapter-Signals auf vier identische Ausgänge. Es handelt sich um einen generischen Baustein (generic FB), der mit verschiedenen ALR-Adaptertypen verwendet werden kann.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden. Die Ereignissteuerung erfolgt über den Eingangsadapter.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden. Ereignisse werden über die Ausgangsadapter weitergeleitet.

Daten-Eingänge

Keine Daten-Eingänge vorhanden. Daten werden über den Eingangsadapter bereitgestellt.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden. Daten werden über die Ausgangsadapter ausgegeben.

Adapter

- **Socket (Eingang)**:
  - **IN**: Unidirektionaler ALR-Adapter (Typ `adapter::types::unidirectional::ALR`). Nimmt das eingehende Signal auf.
- **Plugs (Ausgänge)**:
  - **OUT1**, **OUT2**, **OUT3**, **OUT4**: Jeweils ein unidirektionaler ALR-Adapter (gleicher Typ). Geben das verteilte Signal aus.

Funktionsweise

Der Baustein besitzt eine direkte Durchschaltung: Alle am Eingangsadapter **IN** ankommenden Ereignisse und Daten werden unverändert und gleichzeitig an die vier Ausgangsadapter **OUT1** bis **OUT4** weitergegeben. Es findet keine Verarbeitung, Pufferung oder Filterung statt. Der Baustein arbeitet rein kombinatorisch.

Technische Besonderheiten

- **Generischer Typ**: Der konkrete ALR-Adatper-Typ kann zur Entwurfszeit durch das Attribut `eclipse4diac::core::GenericClassName` festgelegt werden (z. B. `'GEN_ALR_SPLIT'`).
- **Unidirektional**: Die Datenflussrichtung ist fest vom Eingang zu den Ausgängen definiert. Rückwärtige Kommunikation wird nicht unterstützt.
- **Keine Zustandslogik**: Der Baustein enthält keine Ereignisgesteuerte Zustandsmaschine (ECC) und keine zeitlichen Verzögerungen.

Zustandsübersicht

Der Baustein besitzt keine eigenen Zustände. Die Signalweitergabe erfolgt kontinuierlich und ohne Verzögerung.

Anwendungsszenarien

- Verteilung eines ALR-Signals (z. B. Steuerbefehle, Statusmeldungen) an mehrere parallele Module oder Aktoren.
- Erzeugung von Abzweigungen in einer Adapter-basierten Kommunikation, wenn mehrere nachgelagerte Komponenten dieselben Informationen benötigen.
- Einsatz in modularen Automatisierungssystemen, bei denen ein gemeinsames Signal auf mehrere gleichartige Einheiten aufgeteilt werden muss.

Vergleich mit ähnlichen Bausteinen

- **ALR_SPLIT_2 / ALR_SPLIT_N**: Split-Bausteine mit anderer Anzahl an Ausgängen (z. B. 2 oder 8).
- **Event-Splitter**: Teilen Ereignisse auf, arbeiten aber auf reiner Event-Ebene. ALR_SPLIT_4 hingegen verteilt komplette Adapter-Signale (Ereignisse und Daten gekapselt).
- **Daten-Splitter**: Verteilen einzelne Datenwerte, jedoch ohne Adapter-Kapselung. ALR_SPLIT_4 ist speziell für die Verwendung mit unidirektionalen ALR-Adaptertypen optimiert.
- `ALR_SPLIT_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

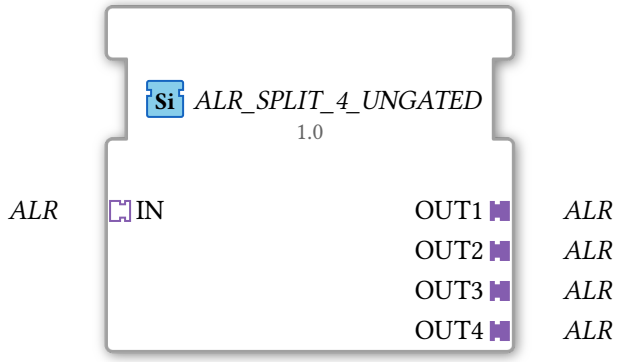
ALR_SPLIT_4 ist ein einfacher, aber essentieller Baustein zur Vervielfachung von Adapterverbindungen. Seine generische Auslegung ermöglicht den Einsatz mit verschiedenen ALR-Adaptertypen, ohne dass eine Anpassung der Bausteinlogik erforderlich ist. Er eignet sich besonders für modulare Architekturen, in denen ein Signal parallel an mehrere Empfänger verteilt werden muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ALR_SPLIT_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALR_SPLIT_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **ALR_SPLIT_4_UNGATED** dient der Aufteilung eines eingehenden ALR-Adapter-Signals auf vier identische Ausgänge. Es handelt sich um einen generischen Baustein (generic FB), der mit verschiedenen ALR-Adaptertypen verwendet werden kann.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden. Die Ereignissteuerung erfolgt über den Eingangsadapter.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden. Ereignisse werden über die Ausgangsadapter weitergeleitet.

Daten-Eingänge

Keine Daten-Eingänge vorhanden. Daten werden über den Eingangsadapter bereitgestellt.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden. Daten werden über die Ausgangsadapter ausgegeben.

Adapter

- **Socket (Eingang)**:
  - **IN**: Unidirektionaler ALR-Adapter (Typ `adapter::types::unidirectional::ALR`). Nimmt das eingehende Signal auf.
- **Plugs (Ausgänge)**:
  - **OUT1**, **OUT2**, **OUT3**, **OUT4**: Jeweils ein unidirektionaler ALR-Adapter (gleicher Typ). Geben das verteilte Signal aus.

Funktionsweise

Der Baustein besitzt eine direkte Durchschaltung: Alle am Eingangsadapter **IN** ankommenden Ereignisse und Daten werden unverändert und gleichzeitig an die vier Ausgangsadapter **OUT1** bis **OUT4** weitergegeben. Es findet keine Verarbeitung, Pufferung oder Filterung statt. Der Baustein arbeitet rein kombinatorisch.

Technische Besonderheiten

- **Generischer Typ**: Der konkrete ALR-Adatper-Typ kann zur Entwurfszeit durch das Attribut `eclipse4diac::core::GenericClassName` festgelegt werden (z. B. `'GEN_ALR_SPLIT'`).
- **Unidirektional**: Die Datenflussrichtung ist fest vom Eingang zu den Ausgängen definiert. Rückwärtige Kommunikation wird nicht unterstützt.
- **Keine Zustandslogik**: Der Baustein enthält keine Ereignisgesteuerte Zustandsmaschine (ECC) und keine zeitlichen Verzögerungen.

Zustandsübersicht

Der Baustein besitzt keine eigenen Zustände. Die Signalweitergabe erfolgt kontinuierlich und ohne Verzögerung.

Anwendungsszenarien

- Verteilung eines ALR-Signals (z. B. Steuerbefehle, Statusmeldungen) an mehrere parallele Module oder Aktoren.
- Erzeugung von Abzweigungen in einer Adapter-basierten Kommunikation, wenn mehrere nachgelagerte Komponenten dieselben Informationen benötigen.
- Einsatz in modularen Automatisierungssystemen, bei denen ein gemeinsames Signal auf mehrere gleichartige Einheiten aufgeteilt werden muss.

Vergleich mit ähnlichen Bausteinen

- **ALR_SPLIT_2 / ALR_SPLIT_N**: Split-Bausteine mit anderer Anzahl an Ausgängen (z. B. 2 oder 8).
- **Event-Splitter**: Teilen Ereignisse auf, arbeiten aber auf reiner Event-Ebene. ALR_SPLIT_4_UNGATED hingegen verteilt komplette Adapter-Signale (Ereignisse und Daten gekapselt).
- **Daten-Splitter**: Verteilen einzelne Datenwerte, jedoch ohne Adapter-Kapselung. ALR_SPLIT_4_UNGATED ist speziell für die Verwendung mit unidirektionalen ALR-Adaptertypen optimiert.
- `ALR_SPLIT_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

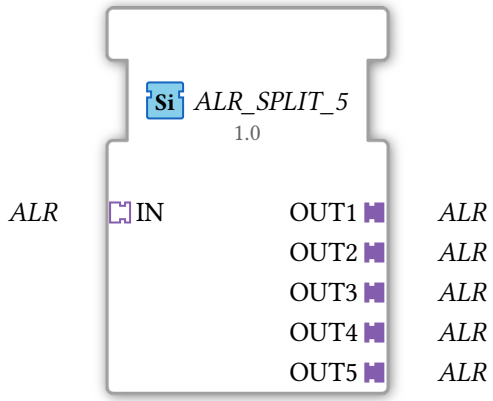
Fazit

ALR_SPLIT_4_UNGATED ist ein einfacher, aber essentieller Baustein zur Vervielfachung von Adapterverbindungen. Seine generische Auslegung ermöglicht den Einsatz mit verschiedenen ALR-Adaptertypen, ohne dass eine Anpassung der Bausteinlogik erforderlich ist. Er eignet sich besonders für modulare Architekturen, in denen ein Signal parallel an mehrere Empfänger verteilt werden muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ALR_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ALR_SPLIT_5** dient dazu, ein eingehendes ALR-Signal auf fünf identische Ausgänge zu verteilen. Er ist als generischer Baustein (Generic FB) implementiert und ermöglicht eine einfache Signalvervielfachung in Steuerungssystemen, die auf dem Adapterkonzept von IEC 61499 basieren.

Schnittstellenstruktur

Ereignis-Eingänge
Es sind keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge
Es sind keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge
Es sind keine Daten-Eingänge vorhanden. Die gesamte Datenübertragung erfolgt ausschließlich über Adapter.

Daten-Ausgänge
Es sind keine Daten-Ausgänge vorhanden.

Adapter
Der Baustein besitzt einen Socket (Eingang) und fünf Plugs (Ausgänge):

Richtung	Name	Typ	Beschreibung
Eingang	IN	`adapter::types::unidirectional::ALR`	Empfängt das zu verteilende ALR-Signal
Ausgang	OUT1	`adapter::types::unidirectional::ALR`	Erster Ausgang mit dem identischen ALR-Signal
Ausgang	OUT2	`adapter::types::unidirectional::ALR`	Zweiter Ausgang mit dem identischen ALR-Signal
Ausgang	OUT3	`adapter::types::unidirectional::ALR`	Dritter Ausgang mit dem identischen ALR-Signal
Ausgang	OUT4	`adapter::types::unidirectional::ALR`	Vierter Ausgang mit dem identischen ALR-Signal
Ausgang	OUT5	`adapter::types::unidirectional::ALR`	Fünfter Ausgang mit dem identischen ALR-Signal

Funktionsweise
Der Funktionsblock leitet das vom Socket `IN` eingehende ALR-Signal unverändert an alle fünf Plugs `OUT1` bis `OUT5` weiter. Es handelt sich um eine rein kombinatorische Weiterleitung – es findet keine Logik, Verzögerung oder Datenmanipulation statt. Der Baustein arbeitet ereignis- und zustandslos; sobald ein Signal am Eingang anliegt, steht es an allen Ausgängen bereit.

Technische Besonderheiten
- **Generischer Baustein:** Der FB ist als Generic FB definiert und verwendet die Eclipse 4diac Generics-Mechanismen (`eclipse4diac::core::GenericClassName`). Dadurch kann er in verschiedenen Kontexten ohne Anpassung der Kernlogik eingesetzt werden.
- **Reine Adapterkommunikation:** Es werden keine klassischen Ereignis- oder Datenein-/ausgänge genutzt. Die gesamte Kommunikation erfolgt über Adapter vom Typ `ALR`, die einen unidirektionalen Datenaustausch ermöglichen.
- **Einfache Struktur:** Keine Zustandsautomaten oder zeitlichen Abhängigkeiten – der FB ist deterministisch und ressourcenschonend.

Zustandsübersicht
Der Funktionsblock besitzt keine explizite Zustandslogik oder Zustandsmaschine. Er arbeitet vollständig kombinatorisch, d.h. das Ausgangssignal ergibt sich direkt und ohne Verzögerung aus dem Eingangssignal.

Anwendungsszenarien

- **Signalverteilung in der Automatisierung:** Ein Sensor liefert ein ALR-Signal, das parallel an mehrere Aktoren (z.B. Ventile, Antriebe) weitergeleitet werden muss.
- **Redundanz:** Das Signal kann an verschiedene Steuerungseinheiten oder Überwachungsmodule verteilt werden.
- **Test- und Simulationsumgebungen:** Ein zentrales ALR-Signal wird auf mehrere Testpunkte aufgeteilt, um parallele Auswertungen zu ermöglichen.

Vergleich mit ähnlichen Bausteinen

- **ALR_SPLIT_2, ALR_SPLIT_3, ALR_SPLIT_4:** Diese Bausteine teilen ein ALR-Signal auf zwei, drei bzw. vier Ausgänge auf. Der vorliegende FB erweitert dies auf fünf Ausgänge.
- **Allgemeine Split-FBs (z.B. DATA_SPLIT):** Ähnliche Bausteine existieren für andere Datentypen. Die Funktionsweise ist identisch, lediglich der verwendete Adaptertyp unterscheidet sich. Der Vorteil des ALR_SPLIT_5 liegt in der direkten Nutzung des ALR-Adapterprotokolls ohne zusätzliche Typkonvertierung.
- `ALR_SPLIT_5_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

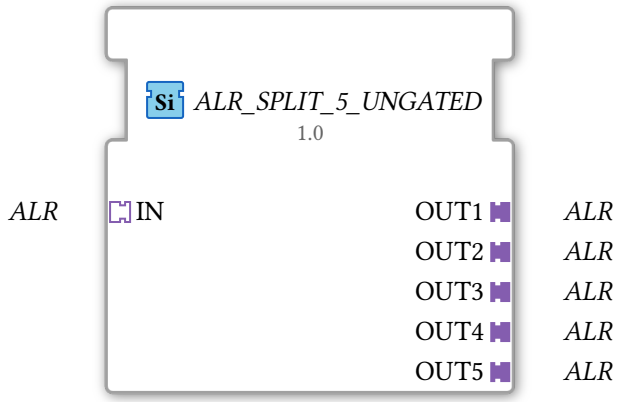
Der **ALR_SPLIT_5** ist ein einfacher, aber wertvoller Baustein zur Vervielfachung von ALR-Signalen in IEC-61499-Systemen. Seine generische und kombinatorische Implementierung macht ihn leicht integrierbar und ressourceneffizient. Er eignet sich besonders für Anwendungen, in denen ein Signal an mehrere Empfänger verteilt werden muss, ohne zusätzliche Logik oder zeitliche Steuerung zu benötigen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ALR_SPLIT_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALR_SPLIT_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ALR_SPLIT_5_UNGATED** dient dazu, ein eingehendes ALR-Signal auf fünf identische Ausgänge zu verteilen. Er ist als generischer Baustein (Generic FB) implementiert und ermöglicht eine einfache Signalvervielfachung in Steuerungssystemen, die auf dem Adapterkonzept von IEC 61499 basieren.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Es sind keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Es sind keine Daten-Eingänge vorhanden. Die gesamte Datenübertragung erfolgt ausschließlich über Adapter.

Daten-Ausgänge

Es sind keine Daten-Ausgänge vorhanden.

Adapter

Der Baustein besitzt einen Socket (Eingang) und fünf Plugs (Ausgänge):

Richtung	Name	Typ	Beschreibung
Eingang	IN	`adapter::types::unidirectional::ALR`	Empfängt das zu verteilende ALR-Signal
Ausgang	OUT1	`adapter::types::unidirectional::ALR`	Erster Ausgang mit dem identischen ALR-Signal
Ausgang	OUT2	`adapter::types::unidirectional::ALR`	Zweiter Ausgang mit dem identischen ALR-Signal
Ausgang	OUT3	`adapter::types::unidirectional::ALR`	Dritter Ausgang mit dem identischen ALR-Signal
Ausgang	OUT4	`adapter::types::unidirectional::ALR`	Vierter Ausgang mit dem identischen ALR-Signal
Ausgang	OUT5	`adapter::types::unidirectional::ALR`	Fünfter Ausgang mit dem identischen ALR-Signal

Funktionsweise

Der Funktionsblock leitet das vom Socket `IN` eingehende ALR-Signal unverändert an alle fünf Plugs `OUT1` bis `OUT5` weiter. Es handelt sich um eine rein kombinatorische Weiterleitung – es findet keine Logik, Verzögerung oder Datenmanipulation statt. Der Baustein arbeitet ereignis- und zustandslos; sobald ein Signal am Eingang anliegt, steht es an allen Ausgängen bereit.

Technische Besonderheiten

- **Generischer Baustein:** Der FB ist als Generic FB definiert und verwendet die Eclipse 4diac Generics-Mechanismen (`eclipse4diac::core::GenericClassName`). Dadurch kann er in verschiedenen Kontexten ohne Anpassung der Kernlogik eingesetzt werden.
- **Reine Adapterkommunikation:** Es werden keine klassischen Ereignis- oder Datenein-/ausgänge genutzt. Die gesamte Kommunikation erfolgt über Adapter vom Typ `ALR`, die einen unidirektionalen Datenaustausch ermöglichen.
- **Einfache Struktur:** Keine Zustandsautomaten oder zeitlichen Abhängigkeiten – der FB ist deterministisch und ressourcenschonend.

Zustandsübersicht

Der Funktionsblock besitzt keine explizite Zustandslogik oder Zustandsmaschine. Er arbeitet vollständig kombinatorisch, d.h. das Ausgangssignal ergibt sich direkt und ohne Verzögerung aus dem Eingangssignal.

Anwendungsszenarien

- **Signalverteilung in der Automatisierung:** Ein Sensor liefert ein ALR-Signal, das parallel an mehrere Aktoren (z.B. Ventile, Antriebe) weitergeleitet werden muss.
- **Redundanz:** Das Signal kann an verschiedene Steuerungseinheiten oder Überwachungsmodule verteilt werden.
- **Test- und Simulationsumgebungen:** Ein zentrales ALR-Signal wird auf mehrere Testpunkte aufgeteilt, um parallele Auswertungen zu ermöglichen.

Vergleich mit ähnlichen Bausteinen

- **ALR_SPLIT_2, ALR_SPLIT_3, ALR_SPLIT_4:** Diese Bausteine teilen ein ALR-Signal auf zwei, drei bzw. vier Ausgänge auf. Der vorliegende FB erweitert dies auf fünf Ausgänge.
- **Allgemeine Split-FBs (z.B. DATA_SPLIT):** Ähnliche Bausteine existieren für andere Datentypen. Die Funktionsweise ist identisch, lediglich der verwendete Adaptertyp unterscheidet sich. Der Vorteil des ALR_SPLIT_5_UNGATED liegt in der direkten Nutzung des ALR-Adapterprotokolls ohne zusätzliche Typkonvertierung.
- `ALR_SPLIT_5_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- `ALR_SPLIT_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

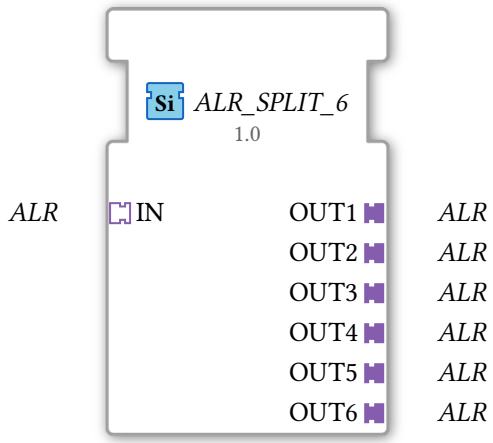
Fazit

Der **ALR_SPLIT_5_UNGATED** ist ein einfacher, aber wertvoller Baustein zur Vervielfachung von ALR-Signalen in IEC-61499-Systemen. Seine generische und kombinatorische Implementierung macht ihn leicht integrierbar und ressourceneffizient. Er eignet sich besonders für Anwendungen, in denen ein Signal an mehrere Empfänger verteilt werden muss, ohne zusätzliche Logik oder zeitliche Steuerung zu benötigen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ALR_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock ALR_SPLIT_6 ist ein generischer 1:6-Splitter für den Adaptertyp `adapter::types::unidirectional::ALR`. Er verteilt ein eingehendes ALR-Signal unverändert auf sechs Ausgänge und dient als passiver Signalverteiler ohne Datenmanipulation.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Name	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::ALR	Socket (Eingang)	ALR-Eingangssignal, das auf alle Ausgänge kopiert wird
OUT1	adapter::types::unidirectional::ALR	Plug (Ausgang)	Erster ALR-Ausgang
OUT2	adapter::types::unidirectional::ALR	Plug (Ausgang)	Zweiter ALR-Ausgang
OUT3	adapter::types::unidirectional::ALR	Plug (Ausgang)	Dritter ALR-Ausgang
OUT4	adapter::types::unidirectional::ALR	Plug (Ausgang)	Vierter ALR-Ausgang
OUT5	adapter::types::unidirectional::ALR	Plug (Ausgang)	Fünfter ALR-Ausgang
OUT6	adapter::types::unidirectional::ALR	Plug (Ausgang)	Sechster ALR-Ausgang

Funktionsweise

Der Baustein leitet das am Socket IN anliegende Adaptersignal unverändert auf alle sechs Plugs OUT1–OUT6 weiter. Es findet keine Transformation, Verzögerung oder Auswahl statt – alle Ausgänge erhalten stets die identische Signalkopie. Der FB ist als generischer Baustein (`GEN_ALR_SPLIT`) deklariert und kann durch die Angabe des konkreten Adaptertyps typisiert werden.

Technische Besonderheiten

- Rein passiver Splitter ohne Ereignisse, Datenports oder Zustandsautomat.
- Generische Implementierung über die Metadaten `eclipse4diac::core::GenericClassName = 'GEN_ALR_SPLIT'`.
- Alle Adapter-Schnittstellen sind unidirektional; es existieren keine Rückkanäle.
- Keine Laufzeitabhängigkeiten oder Initialisierung erforderlich.

Zustandsübersicht

Der Baustein besitzt keinen ECC (Execution Control Chart) und keine internen Zustände. Er arbeitet kontinuierlich und gibt jedes eingehende Adaptersignal sofort an alle Ausgänge weiter.

Anwendungsszenarien

- Vervielfältigung eines ALR-Signals für parallele Verbraucher in Steuerungsnetzwerken.
- Signalverteilung in landwirtschaftlichen Automatisierungssystemen (basierend auf dem ALR-Adapterkontext).
- Einfache Bereitstellung einer identischen Referenz für mehrere nachfolgende Logikbausteine.

Vergleich mit ähnlichen Bausteinen

- **ALR_SPLIT_2 / ALR_SPLIT_3 / …** – gleiche Funktionsweise, aber mit weniger Ausgängen.
- **Ereignisbasierte Splitter (z. B. E_SPLIT)** – arbeiten mit Ereignis-/Datenseparierung, während ALR_SPLIT_6 ausschließlich Adapter nutzt.
- **Daten-Multiplexer** – wählen einen Ausgang aus; ALR_SPLIT_6 aktiviert immer alle Ausgänge.
- `ALR_SPLIT_6_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

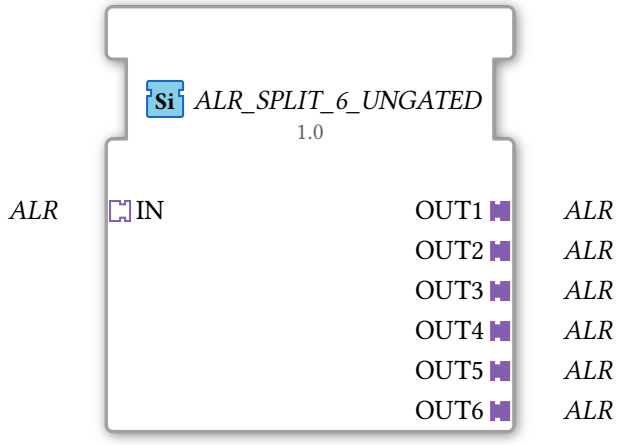
ALR_SPLIT_6 ist ein minimalistischer und zuverlässiger Baustein zur Signalvervielfachung auf Adapterebene. Die generische Auslegung erleichtert die Wiederverwendung in verschiedenen Projekten, und der Verzicht auf komplexe Logik hält das Datenflussmodell klar und einfach.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ALR_SPLIT_6_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALR_SPLIT_6`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock ALR_SPLIT_6_UNGATED ist ein generischer 1:6-Splitter für den Adaptertyp `adapter::types::unidirectional::ALR`. Er verteilt ein eingehendes ALR-Signal unverändert auf sechs Ausgänge und dient als passiver Signalverteiler ohne Datenmanipulation.

Schnittstellenstruktur

Ereignis-Eingänge
Keine

Ereignis-Ausgänge
Keine

Daten-Eingänge
Keine

Daten-Ausgänge
Keine

Adapter

Name	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::ALR	Socket (Eingang)	ALR-Eingangssignal, das auf alle Ausgänge kopiert wird
OUT1	adapter::types::unidirectional::ALR	Plug (Ausgang)	Erster ALR-Ausgang
OUT2	adapter::types::unidirectional::ALR	Plug (Ausgang)	Zweiter ALR-Ausgang
OUT3	adapter::types::unidirectional::ALR	Plug (Ausgang)	Dritter ALR-Ausgang
OUT4	adapter::types::unidirectional::ALR	Plug (Ausgang)	Vierter ALR-Ausgang
OUT5	adapter::types::unidirectional::ALR	Plug (Ausgang)	Fünfter ALR-Ausgang
OUT6	adapter::types::unidirectional::ALR	Plug (Ausgang)	Sechster ALR-Ausgang

Funktionsweise
Der Baustein leitet das am Socket IN anliegende Adaptersignal unverändert auf alle sechs Plugs OUT1–OUT6 weiter. Es findet keine Transformation, Verzögerung oder Auswahl statt – alle Ausgänge erhalten stets die identische Signalkopie. Der FB ist als generischer Baustein (`GEN_ALR_SPLIT`) deklariert und kann durch die Angabe des konkreten Adaptertyps typisiert werden.

Technische Besonderheiten
- Rein passiver Splitter ohne Ereignisse, Datenports oder Zustandsautomat.
- Generische Implementierung über die Metadaten `eclipse4diac::core::GenericClassName = 'GEN_ALR_SPLIT'`.
- Alle Adapter-Schnittstellen sind unidirektional; es existieren keine Rückkanäle.
- Keine Laufzeitabhängigkeiten oder Initialisierung erforderlich.

Zustandsübersicht
Der Baustein besitzt keinen ECC (Execution Control Chart) und keine internen Zustände. Er arbeitet kontinuierlich und gibt jedes eingehende Adaptersignal sofort an alle Ausgänge weiter.

Anwendungsszenarien
- Vervielfältigung eines ALR-Signals für parallele Verbraucher in Steuerungsnetzwerken.
- Signalverteilung in landwirtschaftlichen Automatisierungssystemen (basierend auf dem ALR-Adapterkontext).
- Einfache Bereitstellung einer identischen Referenz für mehrere nachfolgende Logikbausteine.

Vergleich mit ähnlichen Bausteinen
- **ALR_SPLIT_2 / ALR_SPLIT_3 / …** – gleiche Funktionsweise, aber mit weniger Ausgängen.
- **Ereignisbasierte Splitter (z. B. E_SPLIT)** – arbeiten mit Ereignis-/Datenseparierung, während ALR_SPLIT_6_UNGATED ausschließlich Adapter nutzt.

- **Daten-Multiplexer** – wählen einen Ausgang aus; ALR_SPLIT_6_UNGATED aktiviert immer alle Ausgänge.
- `ALR_SPLIT_6`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

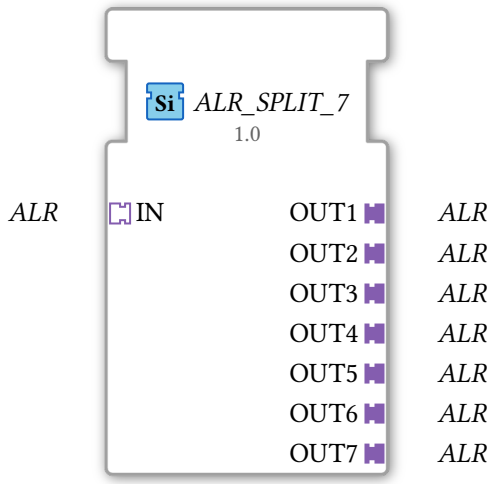
Fazit

ALR_SPLIT_6_UNGATED ist ein minimalistischer und zuverlässiger Baustein zur Signalvervielfachung auf Adapterebene. Die generische Auslegung erleichtert die Wiederverwendung in verschiedenen Projekten, und der Verzicht auf komplexe Logik hält das Datenflussmodell klar und einfach.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ALR_SPLIT_7



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ALR_SPLIT_7` dient dazu, ein eingehendes ALR-Signal (Agricultural Logic Resource) auf sieben identische Ausgangssignale aufzuteilen. Er ist als generischer Baustein konzipiert und ermöglicht die parallele Verteilung eines Alarm- oder Steuersignals an bis zu sieben nachfolgende Komponenten oder Subsysteme.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Name	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::ALR	Socket (Eingang)	Eingehendes ALR-Signal, das auf die Ausgänge verteilt wird.
OUT1	adapter::types::unidirectional::ALR	Plug (Ausgang)	Erster Ausgang des geteilten Signals.
OUT2	adapter::types::unidirectional::ALR	Plug (Ausgang)	Zweiter Ausgang des geteilten Signals.
OUT3	adapter::types::unidirectional::ALR	Plug (Ausgang)	Dritter Ausgang des geteilten Signals.
OUT4	adapter::types::unidirectional::ALR	Plug (Ausgang)	Vierter Ausgang des geteilten Signals.
OUT5	adapter::types::unidirectional::ALR	Plug (Ausgang)	Fünfter Ausgang des geteilten Signals.
OUT6	adapter::types::unidirectional::ALR	Plug (Ausgang)	Sechster Ausgang des geteilten Signals.
OUT7	adapter::types::unidirectional::ALR	Plug (Ausgang)	Siebter Ausgang des geteilten Signals.

Funktionsweise

Der `ALR_SPLIT_7` – Baustein leitet das an seinem Socket `IN` anliegende ALR-Signal unverändert an alle sieben Plugs (`OUT1` bis `OUT7`) weiter. Es findet keine Transformation, Verzögerung oder Filterung des Signals statt. Der Baustein fungiert als reiner Verteiler (Splitter) für einen unidirektionalen ALR-Datenstrom.

Technische Besonderheiten

- **Generic FB:** Der Baustein ist als generischer Funktionsblock (`GenericClassName = 'GEN_ALR_SPLIT'`) realisiert und kann je nach Anwendung parametriert oder instanziiert werden.
- **Unidirektional:** Alle Adapter sind vom Typ `adapter::types::unidirectional::ALR`, d. h. die Daten fließen nur in eine Richtung – vom Eingang zu den Ausgängen.
- **Keine Ereignissteuerung:** Der Baustein besitzt keine Ereignis-Ein- oder -Ausgänge; die Signalweitergabe erfolgt kontinuierlich (datengesteuert).

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustandsautomaten. Es gibt keine Zustände im Sinne von ECC (Execution Control Chart). Er arbeitet rein kombinatorisch und gibt das Eingangssignal stets auf allen Ausgängen aus.

Anwendungsszenarien

- **Verteilen eines Alarms auf mehrere Einheiten:** Ein zentrales Alarmsignal (z. B. eine Störung) soll an sieben verschiedene Steuerungen oder Anzeigeelemente weitergegeben werden.
- **Parallele Verarbeitung:** Ein ALR-Signal wird benötigt, um mehrere unabhängige Subsysteme gleichzeitig zu triggern oder zu überwachen.
- **Redundanz und Test:** Das Signal wird an mehrere Prüf- oder Logging-Komponenten gesendet, ohne dass diese sich gegenseitig beeinflussen.

Vergleich mit ähnlichen Bausteinen

Baustein	Anzahl Ausgänge	Besonderheit
`ALR_SPLIT_2`	2	Einfache Zweifachverteilung
`ALR_SPLIT_4`	4	Vierfachverteilung

ALR_SPLIT_7	7	Siebenfachverteilung (dieser Baustein)
ALR_SPLIT_N (generisch)	Parametrierbar	Flexibler Splitter mit einstellbarer Anzahl

Alle Variationen basieren auf dem gleichen Prinzip: unidirektionale Verteilung ohne Signalveränderung.

- **ALR_SPLIT_7_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

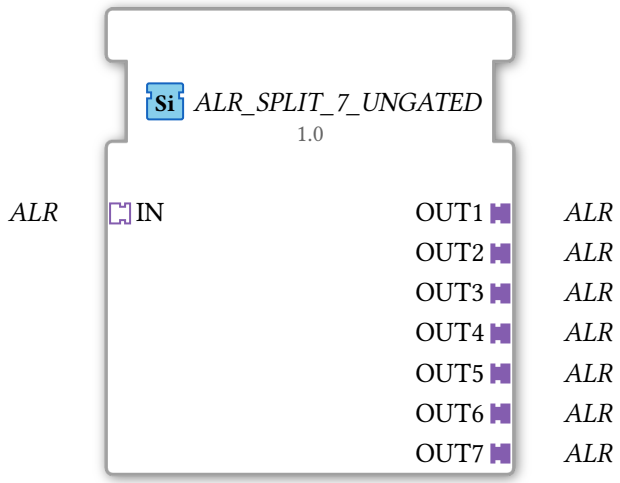
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der ALR_SPLIT_7 ist ein einfacher, aber nützlicher Funktionsblock zur Aufteilung eines einzelnen ALR-Signals auf sieben parallele Pfade. Durch seine generische Implementierung kann er in verschiedenen industriellen oder landwirtschaftlichen Automatisierungsszenarien eingesetzt werden, wo ein Signal mehrfach benötigt wird. Er bietet keine komplexe Logik, erfüllt aber zuverlässig seine Aufgabe als passiver Verteiler.

ALR_SPLIT_7_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von ALR_SPLIT_7. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock ALR_SPLIT_7_UNGATED dient dazu, ein eingehendes ALR-Signal (Agricultural Logic Resource) auf sieben identische Ausgangssignale aufzuteilen. Er ist als generischer Baustein konzipiert und ermöglicht die parallele Verteilung eines Alarm- oder Steuersignals an bis zu sieben nachfolgende Komponenten oder Subsysteme.

Schnittstellenstruktur

Ereignis-Eingänge
Keine

Ereignis-Ausgänge
Keine

Daten-Eingänge
Keine

Daten-Ausgänge
Keine

Adapter

Name	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::ALR	Socket (Eingang)	Eingehendes ALR-Signal, das auf die Ausgänge verteilt wird.
OUT1	adapter::types::unidirectional::ALR	Plug (Ausgang)	Erster Ausgang des geteilten Signals.
OUT2	adapter::types::unidirectional::ALR	Plug (Ausgang)	Zweiter Ausgang des geteilten Signals.
OUT3	adapter::types::unidirectional::ALR	Plug (Ausgang)	Dritter Ausgang des geteilten Signals.
OUT4	adapter::types::unidirectional::ALR	Plug (Ausgang)	Vierter Ausgang des geteilten Signals.
OUT5	adapter::types::unidirectional::ALR	Plug (Ausgang)	Fünfter Ausgang des geteilten Signals.
OUT6	adapter::types::unidirectional::ALR	Plug (Ausgang)	Sechster Ausgang des geteilten Signals.
OUT7	adapter::types::unidirectional::ALR	Plug (Ausgang)	Siebter Ausgang des geteilten Signals.

Funktionsweise

Der `ALR_SPLIT_7_UNGATED` – Baustein leitet das an seinem Socket `IN` anliegende ALR-Signal unverändert an alle sieben Plugs (`OUT1` bis `OUT7`) weiter. Es findet keine Transformation, Verzögerung oder Filterung des Signals statt. Der Baustein fungiert als reiner Verteiler (Splitter) für einen unidirektionalen ALR-Datenstrom.

Technische Besonderheiten

- **Generic FB:** Der Baustein ist als generischer Funktionsblock (`GenericClassName = 'GEN_ALR_SPLIT'`) realisiert und kann je nach Anwendung parametriert oder instanziiert werden.
- **Unidirektional:** Alle Adapter sind vom Typ `adapter::types::unidirectional::ALR`, d. h. die Daten fließen nur in eine Richtung – vom Eingang zu den Ausgängen.
- **Keine Ereignissteuerung:** Der Baustein besitzt keine Ereignis-Ein- oder -Ausgänge; die Signalweitergabe erfolgt kontinuierlich (datengesteuert).

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustandsautomaten. Es gibt keine Zustände im Sinne von ECC (Execution Control Chart). Er arbeitet rein kombinatorisch und gibt das Eingangssignal stets auf allen Ausgängen aus.

Anwendungsszenarien

- **Verteilen eines Alarms auf mehrere Einheiten:** Ein zentrales Alarmsignal (z. B. eine Störung) soll an sieben verschiedene Steuerungen oder Anzeigeelemente weitergegeben werden.
- **Parallele Verarbeitung:** Ein ALR-Signal wird benötigt, um mehrere unabhängige Subsysteme gleichzeitig zu triggern oder zu überwachen.
- **Redundanz und Test:** Das Signal wird an mehrere Prüf- oder Logging-Komponenten gesendet, ohne dass diese sich gegenseitig beeinflussen.

Vergleich mit ähnlichen Bausteinen

Baustein	Anzahl Ausgänge	Besonderheit
`ALR_SPLIT_2`	2	Einfache Zweifachverteilung
`ALR_SPLIT_4`	4	Vierfachverteilung
`ALR_SPLIT_7_UNGATED`	7	Siebenfachverteilung (dieser Baustein)
`ALR_SPLIT_N` (generisch)	Parametrierbar	Flexibler Splitter mit einstellbarer Anzahl

Alle Variationen basieren auf dem gleichen Prinzip: unidirektionale Verteilung ohne Signalveränderung.

- `ALR_SPLIT_7`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

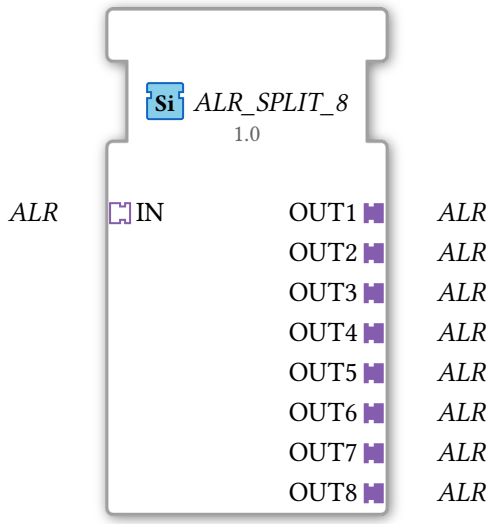
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `ALR_SPLIT_7_UNGATED` ist ein einfacher, aber nützlicher Funktionsblock zur Aufteilung eines einzelnen ALR-Signals auf sieben parallele Pfade. Durch seine generische Implementierung kann er in verschiedenen industriellen oder landwirtschaftlichen Automatisierungsszenarien eingesetzt werden, wo ein Signal mehrfach benötigt wird. Er bietet keine komplexe Logik, erfüllt aber zuverlässig seine Aufgabe als passiver Verteiler.

ALR_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ALR_SPLIT_8** dient der Aufteilung eines einzelnen ALR-Signals (über einen Socket) auf acht identische ALR-Ausgänge (Plugs). Dieser Baustein ist als generischer FB konzipiert und ermöglicht die Signalverteilung innerhalb einer IEC 61499-basierten Steuerung.

Schnittstellenstruktur

Ereignis-Eingänge
Keine

Ereignis-Ausgänge
Keine

Daten-Eingänge
Keine

Daten-Ausgänge
Keine

Adapter

Bezeichnung	Typ	Richtung	Beschreibung
IN	`adapter::types::unidirectional::ALR`	Socket	Eingangssignal, das auf alle Ausgänge weitergeleitet wird
OUT1	`adapter::types::unidirectional::ALR`	Plug	Ausgang 1 – dupliziertes Signal
OUT2	`adapter::types::unidirectional::ALR`	Plug	Ausgang 2 – dupliziertes Signal
OUT3	`adapter::types::unidirectional::ALR`	Plug	Ausgang 3 – dupliziertes Signal
OUT4	`adapter::types::unidirectional::ALR`	Plug	Ausgang 4 – dupliziertes Signal
OUT5	`adapter::types::unidirectional::ALR`	Plug	Ausgang 5 – dupliziertes Signal
OUT6	`adapter::types::unidirectional::ALR`	Plug	Ausgang 6 – dupliziertes Signal
OUT7	`adapter::types::unidirectional::ALR`	Plug	Ausgang 7 – dupliziertes Signal
OUT8	`adapter::types::unidirectional::ALR`	Plug	Ausgang 8 – dupliziertes Signal

Funktionsweise
Der Baustein leitet das über den Socket **IN** anliegende ALR-Signal unverändert an alle acht Ausgangsadapter **OUT1** bis **OUT8** weiter. Es findet keinerlei Datenverarbeitung oder zeitliche Verzögerung statt – die Signale werden direkt durchgeschliffen. Dadurch können mehrere nachfolgende Bausteine parallel mit dem gleichen Signal versorgt werden.

Technische Besonderheiten
- **Generischer Typ:** Der FB ist als generischer Baustein (`GenericClassName = 'GEN_ALR_SPLIT'`) implementiert, sodass der zugrundeliegende ALR-Typ bei der Instanziierung spezifiziert werden kann.
- **Reine Signalverteilung:** Keine Zustände, keine Datenkonvertierung und kein eigenes Zeitverhalten – der Baustein arbeitet rein passiv.
- **Adapterbasiert:** Die gesamte Kommunikation erfolgt über Adapter, wodurch eine lose Kopplung und einfache Wiederverwendung gewährleistet wird.

Zustandsübersicht
Der FB besitzt keine Ereignisse und keinen internen Zustandsautomaten. Die Ausführung erfolgt strikt datenflussgetrieben: Ändert sich das Signal am Eingang, wird es sofort an alle Ausgänge propagiert.

Anwendungsszenarien
- **Signalsplittung** in Steuerungsarchitekturen, wenn ein ALR-Signal mehrfach benötigt wird (z. B. für parallele Auswertungen oder Redundanz).
- **Verschaltung mehrerer Subsysteme** mit identischen Steuerdaten, ohne mehrfache Signalerzeugung.
- **Testaufbauten** zur gleichzeitigen Stimulation mehrerer Empfänger mit demselben Wert.

Vergleich mit ähnlichen Bausteinen

Andere Splitter-FBs (z. B. SPLIT_4 oder SPLIT_16) unterscheiden sich lediglich in der Anzahl der Ausgänge. Der ALR_SPLIT_8 ist speziell für den ALR-Adaptertyp ausgelegt, während generische Splitter (z. B. ANY_SPLIT) für beliebige Datentypen verwendet werden können. Der vorliegende Baustein bietet eine optimierte, typsichere Lösung für ALR-Verteilanwendungen.

- `ALR_SPLIT_8_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

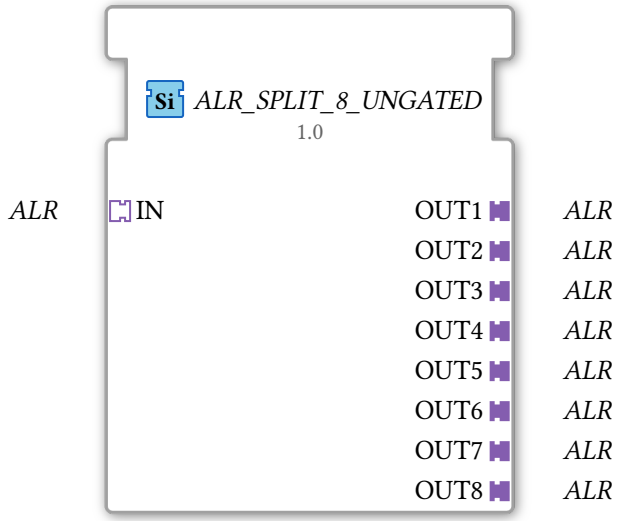
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der **ALR_SPLIT_8** ist ein einfacher, aber effektiver Funktionsblock zur Vervielfachung eines ALR-Signals auf acht Ausgänge. Seine generische Natur, die adapterbasierte Schnittstelle und die passive Funktionsweise machen ihn zu einer robusten und flexiblen Komponente für IEC 61499-Applikationen.

ALR_SPLIT_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALR_SPLIT_8`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ALR_SPLIT_8_UNGATED** dient der Aufteilung eines einzelnen ALR-Signals (über einen Socket) auf acht identische ALR-Ausgänge (Plugs). Dieser Baustein ist als generischer FB konzipiert und ermöglicht die Signalverteilung innerhalb einer IEC 61499-basierten Steuerung.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge
Keine

Daten-Eingänge
Keine

Daten-Ausgänge
Keine

Adapter

Bezeichnung	Typ	Richtung	Beschreibung
IN	`adapter::types::unidirectional::ALR`	Socket	Eingangssignal, das auf alle Ausgänge weitergeleitet wird
OUT1	`adapter::types::unidirectional::ALR`	Plug	Ausgang 1 – dupliziertes Signal
OUT2	`adapter::types::unidirectional::ALR`	Plug	Ausgang 2 – dupliziertes Signal
OUT3	`adapter::types::unidirectional::ALR`	Plug	Ausgang 3 – dupliziertes Signal
OUT4	`adapter::types::unidirectional::ALR`	Plug	Ausgang 4 – dupliziertes Signal
OUT5	`adapter::types::unidirectional::ALR`	Plug	Ausgang 5 – dupliziertes Signal
OUT6	`adapter::types::unidirectional::ALR`	Plug	Ausgang 6 – dupliziertes Signal
OUT7	`adapter::types::unidirectional::ALR`	Plug	Ausgang 7 – dupliziertes Signal
OUT8	`adapter::types::unidirectional::ALR`	Plug	Ausgang 8 – dupliziertes Signal

Funktionsweise
Der Baustein leitet das über den Socket **IN** anliegende ALR-Signal unverändert an alle acht Ausgangsadapter **OUT1** bis **OUT8** weiter. Es findet keinerlei Datenverarbeitung oder zeitliche Verzögerung statt – die Signale werden direkt durchgeschliffen. Dadurch können mehrere nachfolgende Bausteine parallel mit dem gleichen Signal versorgt werden.

Technische Besonderheiten
- **Generischer Typ:** Der FB ist als generischer Baustein (`GenericClassName = 'GEN_ALR_SPLIT'`) implementiert, sodass der zugrundeliegende ALR-Typ bei der Instanziierung spezifiziert werden kann.
- **Reine Signalverteilung:** Keine Zustände, keine Datenkonvertierung und kein eigenes Zeitverhalten – der Baustein arbeitet rein passiv.
- **Adapterbasiert:** Die gesamte Kommunikation erfolgt über Adapter, wodurch eine lose Kopplung und einfache Wiederverwendung gewährleistet wird.

Zustandsübersicht
Der FB besitzt keine Ereignisse und keinen internen Zustandsautomaten. Die Ausführung erfolgt strikt datenflussgetrieben: Ändert sich das Signal am Eingang, wird es sofort an alle Ausgänge propagiert.

Anwendungsszenarien
- **Signalsplittung** in Steuerungsarchitekturen, wenn ein ALR-Signal mehrfach benötigt wird (z. B. für parallele Auswertungen oder Redundanz).
- **Verschaltung mehrerer Subsysteme** mit identischen Steuerdaten, ohne mehrfache Signalerzeugung.
- **Testaufbauten** zur gleichzeitigen Stimulation mehrerer Empfänger mit demselben Wert.

Vergleich mit ähnlichen Bausteinen
Andere Splitter-FBs (z. B. SPLIT_4 oder SPLIT_16) unterscheiden sich lediglich in der Anzahl der Ausgänge. Der ALR_SPLIT_8_UNGATED ist speziell für den ALR-Adaptertyp ausgelegt, während generische Splitter (z. B. ANY_SPLIT) für beliebige Datentypen verwendet werden können. Der vorliegende Baustein bietet eine optimierte, typsichere Lösung für ALR-Verteilanwendungen.

- `ALR_SPLIT_8_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- `ALR_SPLIT_8`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

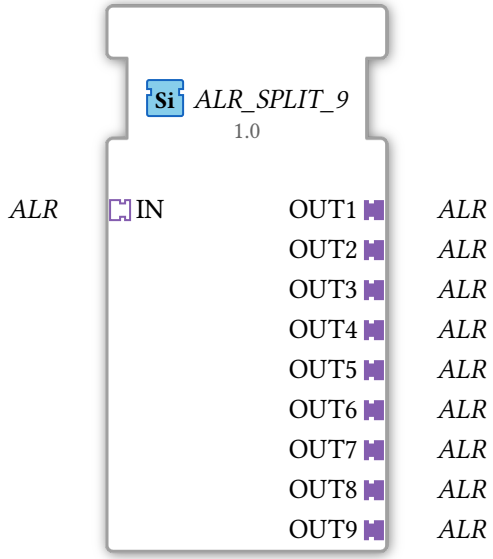
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **ALR_SPLIT_8_UNGATED** ist ein einfacher, aber effektiver Funktionsblock zur Vervielfachung eines ALR-Signals auf acht Ausgänge. Seine generische Natur, die adapterbasierte Schnittstelle und die passive Funktionsweise machen ihn zu einer robusten und flexiblen Komponente für IEC 61499-Applikationen.

ALR_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **ALR_SPLIT_9** ist ein generischer Funktionsblock, der ein eingehendes ALR-Signal (unidirektionaler Adapter) auf genau neun identische Ausgänge verteilt. Er fungiert als 1:9-Splitter für den Adaptertyp `ALR` aus dem Paket `adapter::types::unidirectional`.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ
Socket	`IN`	`ALR` (unidirectional) – Der Eingang, der auf die neun Ausgänge verteilt wird.
Plug	`OUT1` ... `OUT9`	`ALR` (unidirectional) – Die neun Ausgänge, an denen das eingehende Signal bereitgestellt wird.

Funktionsweise

Der Baustein leitet das am Socket `IN` anliegende ALR-Signal unverändert und ohne Verzögerung an alle neun Plugs (`OUT1` bis `OUT9`) weiter. Es findet keine Verarbeitung, Umwandlung oder Zustandslogik statt – der FB realisiert eine reine Signalverteilung auf der Adapterebene.

Technische Besonderheiten

- Der FB ist generisch und besitzt die Attribute `GenericClassName` (Wert: `'GEN_ALR_SPLIT'`) sowie `TypeHash`.
- Er verwendet ausschließlich Adapter-Schnittstellen; es sind keine Ereignisse oder Ein-/Ausgangsdaten definiert.
- Da keinerlei Algorithmus oder Zustandsautomat implementiert ist, arbeitet der FB passiv und ressourcenschonend.

Zustandsübersicht

Der FB besitzt keine Zustandsmaschine, da er keine interne Logik oder Speicherverhalten aufweist.

Anwendungsszenarien

- **Signalverteilung:** Ein von einer Steuerung erzeugtes ALR-Ereignis soll gleichzeitig mehrere Aktoren oder Komponenten erreichen.
- **Redundante Ansteuerung:** Mehrere identische Verbraucher müssen das gleiche Ausgangssignal erhalten (z. B. parallele Hydraulikventile in der Landtechnik).
- **Monitoring:** Ein Signal soll ohne Beeinflussung an eine überwachende Einheit und gleichzeitig an den eigentlichen Verbraucher gehen.

Vergleich mit ähnlichen Bausteinen

- **ALR_SPLIT_4 / ALR_SPLIT_8:** Analoge Bausteine mit anderer Anzahl von Ausgängen.
- **Daten-Typ-Splittern (z. B. `SPLIT_INT`):** Diese arbeiten auf Datenebene, während `ALR_SPLIT_9` auf der Adapterebene (Signal-/Ereignisweitergabe) operiert.
- **Adapter-Multiplexer/Demultiplexer:** Im Gegensatz zu diesen wählt oder kombiniert `ALR_SPLIT_9` keine Signale, sondern leitet das eingehende Signal ausschließlich 1:9 weiter.
- `ALR_SPLIT_9_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

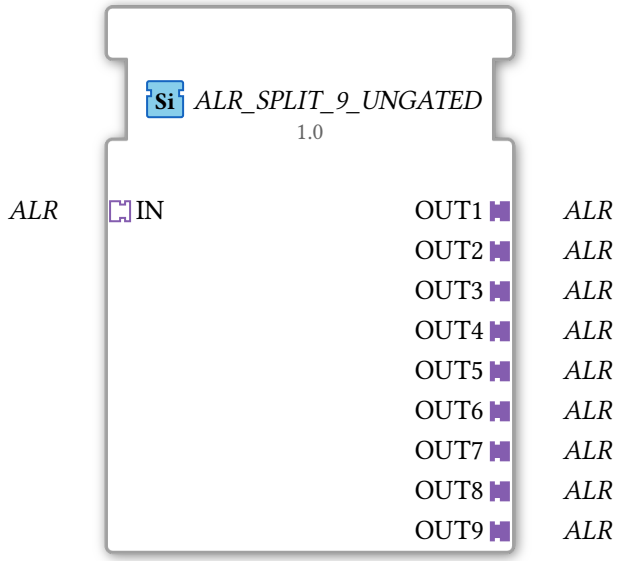
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der **ALR_SPLIT_9** ist ein einfacher, aber praktischer Funktionsblock zur Verteilung eines unidirektionalen ALR-Signals auf neun parallele Pfade. Er ist leicht verständlich, benötigt keine Ressourcen für Logik oder Zustände und lässt sich direkt in Steuerungsprogramme integrieren, die eine Signalvervielfältigung erfordern.

ALR_SPLIT_9_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `ALR_SPLIT_9`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **ALR_SPLIT_9_UNGATED** ist ein generischer Funktionsblock, der ein eingehendes ALR-Signal (unidirektionaler Adapter) auf genau neun identische Ausgänge verteilt. Er fungiert als 1:9-Splitter für den Adaptertyp `ALR` aus dem Paket `adapter::types::unidirectional`.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ
Socket	`IN`	`ALR` (unidirectional) – Der Eingang, der auf die neun Ausgänge verteilt wird.
Plug	`OUT1` … `OUT9`	`ALR` (unidirectional) – Die neun Ausgänge, an denen das eingehende Signal bereitgestellt wird.

Funktionsweise

Der Baustein leitet das am Socket `IN` anliegende ALR-Signal unverändert und ohne Verzögerung an alle neun Plugs (`OUT1` bis `OUT9`) weiter. Es findet keine Verarbeitung, Umwandlung oder Zustandslogik statt – der FB realisiert eine reine Signalverteilung auf der Adapterebene.

Technische Besonderheiten

- Der FB ist generisch und besitzt die Attribute `GenericClassName` (Wert: `'GEN_ALR_SPLIT'`) sowie `TypeHash`.
- Er verwendet ausschließlich Adapter-Schnittstellen; es sind keine Ereignisse oder Ein-/Ausgangsdaten definiert.
- Da keinerlei Algorithmus oder Zustandsautomat implementiert ist, arbeitet der FB passiv und ressourcenschonend.

Zustandsübersicht

Der FB besitzt keine Zustandsmaschine, da er keine interne Logik oder Speicherverhalten aufweist.

Anwendungsszenarien

- **Signalverteilung:** Ein von einer Steuerung erzeugtes ALR-Ereignis soll gleichzeitig mehrere Aktoren oder Komponenten erreichen.
- **Redundante Ansteuerung:** Mehrere identische Verbraucher müssen das gleiche Ausgangssignal erhalten (z. B. parallele Hydraulikventile in der Landtechnik).
- **Monitoring:** Ein Signal soll ohne Beeinflussung an eine überwachende Einheit und gleichzeitig an den eigentlichen Verbraucher gehen.

Vergleich mit ähnlichen Bausteinen

- **ALR_SPLIT_4 / ALR_SPLIT_8:** Analoge Bausteine mit anderer Anzahl von Ausgängen.
- **Daten-Typ-Splittern (z. B. `SPLIT_INT`):** Diese arbeiten auf Datenebene, während `ALR_SPLIT_9_UNGATED` auf der Adapterebene (Signal-/Ereignisweitergabe) operiert.
- **Adapter-Multiplexer/Demultiplexer:** Im Gegensatz zu diesen wählt oder kombiniert `ALR_SPLIT_9_UNGATED` keine Signale, sondern leitet das eingehende Signal ausschließlich 1:9 weiter.
- `ALR_SPLIT_9`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

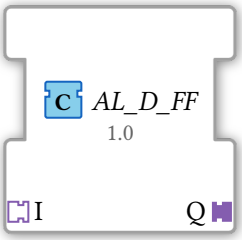
Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **ALR_SPLIT_9_UNGATED** ist ein einfacher, aber praktischer Funktionsblock zur Verteilung eines unidirektionalen ALR-Signals auf neun parallele Pfade. Er ist leicht verständlich, benötigt keine Ressourcen für Logik oder Zustände und lässt sich direkt in Steuerungsprogramme integrieren, die eine Signalvervielfältigung erfordern.

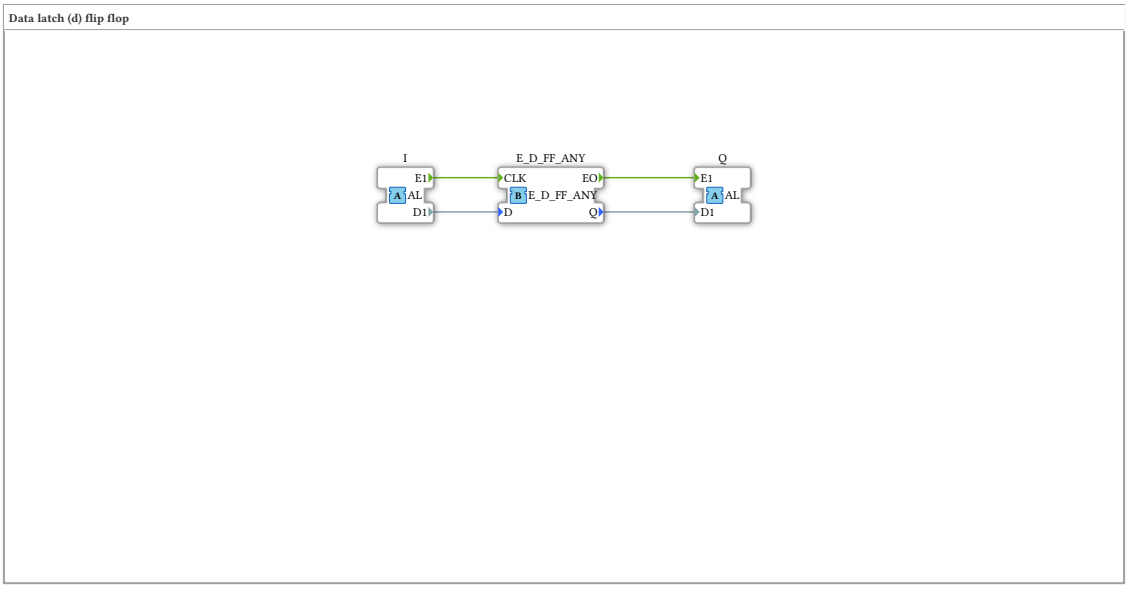
LWORD

AL_D_FF



Value to latch – *AL*

AL – Latched value



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AL_D_FF** realisiert ein Data-Latch-Flipflop (D-Flipflop). Er dient dazu, einen digitalen Datenwert bei jedem eingehenden Ereignis zu speichern und den gespeicherten Wert über einen Adapterausgang zur Verfügung zu stellen. Der Baustein kapselt die wiederkehrende Aufgabe eines D-Flipflops und bietet über Adapter eine saubere, typisierte Schnittstelle für Ereignis- und Datenaustausch.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge. Das Ereignis zum Übernehmen des Datenwertes wird über den Adapter-Socket **I** (E1) bereitgestellt.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge. Das Ausgangsereignis wird über den Adapter-Plug **Q** (E1) zur Verfügung gestellt.

Daten-Eingänge

Keine direkten Daten-Eingänge. Der zu speichernde Datenwert wird über den Adapter-Socket **I** (D1) geliefert.

Daten-Ausgänge

Keine direkten Daten-Ausgänge. Der gespeicherte Datenwert wird über den Adapter-Plug **Q** (D1) ausgegeben.

Adapter

Name	Typ	Richtung	Kommentar
I	`adapter::types::unidirectional::AL`	Socket	Wert, der gelatcht werden soll
Q	`adapter::types::unidirectional::AL`	Plug	Gelatchter Wert

Die Adapter sind unidirektional und vom Typ `AL`. Jeder Adapter enthält einen Ereigniseingang/-ausgang (E1) und einen Dateneingang/-ausgang (D1). Der Socket **I** dient als Eingangsschnittstelle, der Plug **Q** als Ausgangsschnittstelle.

Funktionsweise

Der Baustein verwendet intern den Standard-Funktionsblock `iec61499::events::E_D_FF_ANY`.
Bei jedem Ereignis am Adaptereingang **I** (E1) wird der Datenwert von **I.D1** übernommen und am

Adapterausgang **Q.D1** gespeichert. Gleichzeitig wird ein Ereignis am Ausgang **Q.E1** erzeugt, um der nachgeschalteten Logik die Verfügbarkeit des neuen Werts zu signalisieren.

Die Übernahme erfolgt flankengesteuert (steigende Flanke) – das Verhalten entspricht einem getakteten D-Latch oder D-Flipflop, je nach Definition des intern verwendeten FB.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Statt einzelner Ein-/Ausgänge wird die gesamte Signalgruppe (Ereignis + Daten) über einen Adapter zusammengefasst. Dies vereinfacht die Verbindung in der Applikation und sorgt für eine klare semantische Kapselung.
- **Wiederverwendung vorhandener Logik**: Der Baustein delegiert die Kernfunktion an den IEC-61499-Standard-FB `E_D_FF_ANY`, sodass eine erprobte und portable Implementierung zum Einsatz kommt.
- **Typsicherheit**: Durch die Verwendung eines festgelegten Adaptertyps (`AL`) wird sichergestellt, dass nur kompatible Signale angeschlossen werden können.

Zustandsübersicht

Der Baustein besitzt keinen eigenen ECC-Zustandsautomaten. Sein internes Verhalten wird vollständig durch den eingebetteten FB `E_D_FF_ANY` bestimmt. Dieser realisiert einen D-Flipflop mit zwei logischen Zuständen:

- **Zustand 0**: Gespeicherter Wert ist `false` (bzw. der Datenwert des Adapters ist 0).
- **Zustand 1**: Gespeicherter Wert ist `true` (bzw. der Datenwert des Adapters ist 1).

Ein Zustandswechsel erfolgt nur bei einem Ereignis an **I.E1** dabei wird der an **I.D1** anliegende Wert übernommen.

Anwendungsszenarien

- **Signalverzögerung und Synchronisation**: Ein Datenwert soll erst bei einem definierten Ereignis (z. B. Taktflanke) weitergegeben werden.
- **Speicherung eines binären Zustands**: z. B. als Teil eines Schieberegisters, eines Zustandsregisters oder einer Merkerlogik.
- **Kopplung mit Adapter-basierten Komponenten**: Einfache Integration in Applikationen, die bereits den Adaptertyp `adapter::types::unidirectional::AL` verwenden.

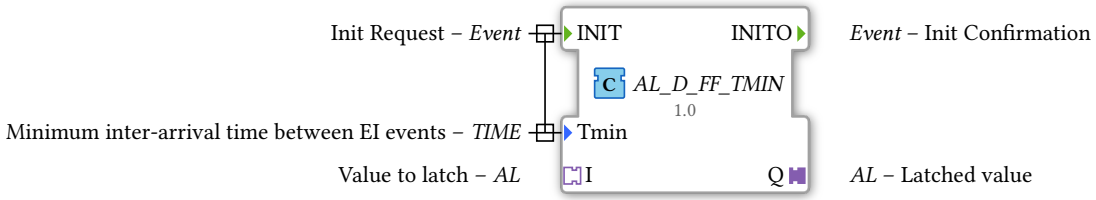
Vergleich mit ähnlichen Bausteinen

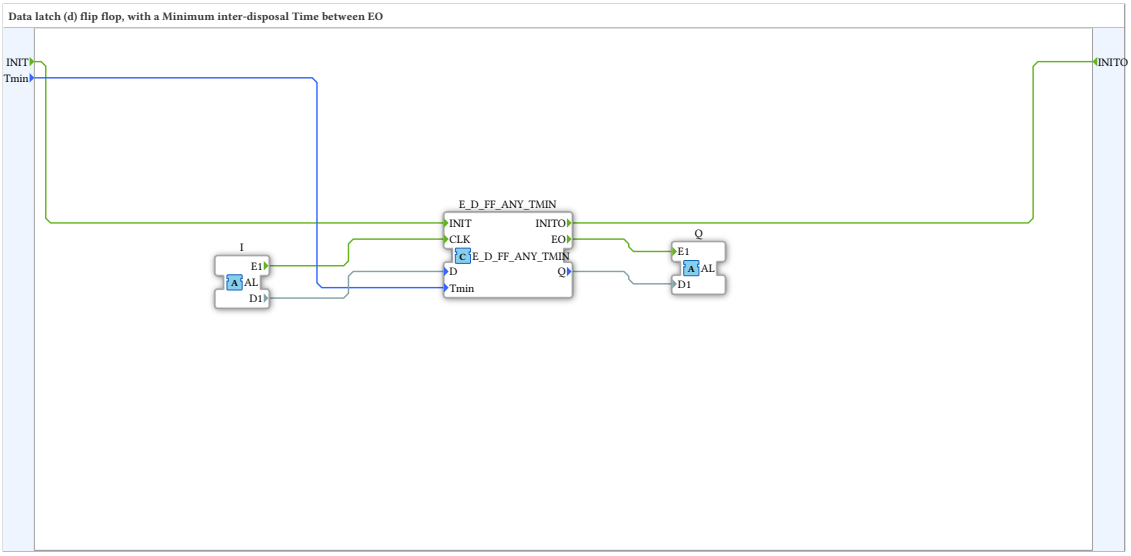
- **E_D_FF_ANY**: Standard-D-Flipflop mit einzelnen Ereignis- und Dateneingängen. `AL_D_FF` kapselt diesen Baustein und bietet eine Adapterschnittstelle.
- **SR_FF oder JK_FF**: Andere Flipflop-Typen mit mehreren Steuereingängen. Der `AL_D_FF` beschränkt sich auf die einfache D-Funktionalität.
- **AL_D_LATCH**: Ein reiner Latch ohne Ereignissteuerung? (Nicht vorhanden). Der `AL_D_FF` arbeitet flankengesteuert und eignet sich daher für taktsynchrone Systeme.

Fazit

Der Funktionsblock `AL_D_FF` bietet eine gekapselte, adapterbasierte Lösung für die grundlegende D-Flipflop-Funktionalität. Er vereinfacht die Handhabung von Signal- und Ereignisgruppen in IEC-61499-Applikationen, erhöht die Wiederverwendbarkeit und bleibt durch die Rückführung auf einen standardisierten Kernbaustein zuverlässig und nachvollziehbar.

AL_D_FF_TMIN





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock AL_D_FF_TMIN implementiert ein flankengesteuertes D-Flip-Flop (Data Latch) mit einer einstellbaren minimalen Verweilzeit zwischen den Ausgangsereignissen (Minimum inter-disposal time). Er übernimmt einen Datenwert bei einem eingehenden Ereignis, speichert diesen und gibt ihn – zusammen mit einem eigenen Ereignis – erst nach Ablauf der vorgegebenen Mindestzeit wieder aus. So wird eine Begrenzung der Ausgabeereignisrate erreicht.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung; legt den Parameter Tmin für die Mindestzeit zwischen Ausgangsereignissen fest.

Ereignis-Ausgänge

- **INITO**: Bestätigung der durchgeführten Initialisierung.

Daten-Eingänge

- **Tmin** (TIME): Minimaler zeitlicher Abstand, der zwischen zwei aufeinanderfolgenden Ausgangsereignissen (EO) eingehalten werden muss.

Daten-Ausgänge

- Keine direkten Datenausgänge – die gelatchten Werte werden ausschließlich über den Ausgangs-Adapter Q übertragen.

Adapter

- **I** (Socket, Typ: `adapter::types::unidirectional::AL`): Eingangs-Adapter für den zu latchierenden Wert. Über die Ereignisschnittstelle (E1) wird der Latch-Impuls empfangen, über die Datenschnittstelle (D1) der zu übernehmende Datenwert bereitgestellt.
- **Q** (Plug, Typ: `adapter::types::unidirectional::AL`): Ausgangs-Adapter für den gespeicherten Wert. Bei jedem erfolgreichen Latch-Vorgang wird ein Ereignis (E1) gesendet und der aktuell gespeicherte Datenwert (D1) ausgegeben.

Funktionsweise

Der Baustein nutzt intern den vordefinierten Flip-Flop-Baustein `E_D_FF_ANY_TMIN`.
Sobald ein Ereignis am Eingangs-Adapter (I.E1) eintrifft, wird der gleichzeitig anliegende Datenwert (I.D1) übernommen und intern gespeichert. Das Ausgangsereignis (Q.E1) wird jedoch erst dann erzeugt, wenn seit dem letzten Ausgangsereignis mindestens die mit Tmin festgelegte Zeitspanne vergangen ist. Erst

dann wird der gespeicherte Wert an den Ausgangs-Adapter (Q.D1) weitergegeben. Die Initialisierung mit INIT setzt den Parameter Tmin und muss vor dem ersten Gebrauch erfolgen.

Technische Besonderheiten

- **Mindestzeit zwischen Ausgangsereignissen**: Die Verzögerung Tmin begrenzt die Frequenz der Ausgangsereignisse – nützlich zur Vermeidung von Busüberlastungen oder zur Begrenzung der Aktor-Ansteuerungsrate.
- **Adapter-Schnittstellen**: Die Kopplung erfolgt über standardisierte, unidirektionale Adapter (`adapter::types::unidirectional::AL`). Dadurch ist der Baustein flexibel in verschiedene Applikationen integrierbar.
- **Flankengetriggert**: Der interne Flip-Flop reagiert auf die steigende Flanke des Ereignisses am Eingangs-Adapter (I.E1).

Zustandsübersicht

Der Baustein besitzt keinen explizit dargestellten Zustandsautomaten. Sein Verhalten ergibt sich aus dem internen D-Flip-Flop mit Zeitsteuerung:

- Nach erfolgreicher Initialisierung (INIT → INITO) wartet der Baustein auf ein Ereignis am Eingangs-Adapter I.
- Bei einem Ereignis wird der anliegende Datenwert gelatcht. Sofort danach wird geprüft, ob die seit dem letzten Ausgangsereignis vergangene Zeit bereits Tmin überschreitet.
  - Falls ja: Das Ausgangsereignis wird sofort erzeugt.
  - Falls nein: Der Ausgang wird solange blockiert, bis die Wartezeit abgelaufen ist; danach erfolgt die Ausgabe.
- Während der Wartezeit eingehende Ereignisse werden (je nach interner Implementierung des zugrunde liegenden Bausteins) möglicherweise ignoriert oder zwischengespeichert – in der Regel wird nur der aktuellste Datenwert für die nächste Ausgabe berücksichtigt.

Anwendungsszenarien

- **Synchronisation zeitkritischer Steuerungen**, bei denen Daten nicht häufiger als in einem vorgegebenen Rhythmus aktualisiert werden dürfen.
- **Entprellung von Signalen** oder Drosselung von Ereignisketten in Kommunikationsnetzwerken.
- **Einfache Zustandsmaschinen**, die auf äußere Ereignisse reagieren und eine minimale Reaktionszeit einhalten müssen (z. B. in der Automatisierungstechnik oder Steuerung physikalischer Aktoren).

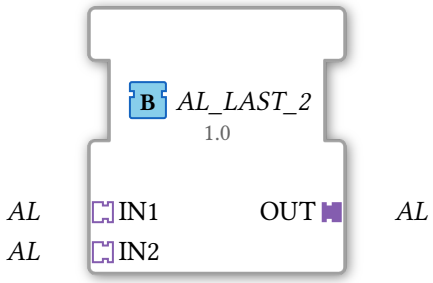
Vergleich mit ähnlichen Bausteinen

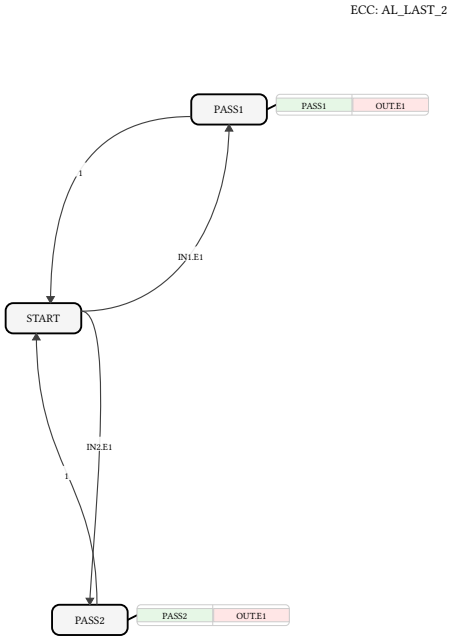
- **AL_D_FF**: Ein einfaches D-Flip-Flop ohne Mindestzeit zwischen Ausgangsereignissen – AL_D_FF_TMIN erweitert dieses um die Tmin-Steuerung und die Adapter-Schnittstellen.
- **E_D_FF_ANY_TMIN**: Der zugrunde liegende, generische Flip-Flop, der direkt (ohne Adapter-Kapselung) verwendet werden kann. AL_D_FF_TMIN kapselt diesen in ein plug-and-play-fähiges Adapter-Format.
- **R_TRIG / F_TRIG**: Erkennen steigende bzw. fallende Flanken, speichern aber keine Datenwerte und bieten keine einstellbare Ausgabe-Verzögerung.

Fazit

Der Funktionsblock AL_D_FF_TMIN bietet eine zuverlässige, flankengesteuerte D-Latch-Funktion mit einer einstellbaren Mindestzeit zwischen Ausgangsereignissen. Durch die Verwendung standardisierter Adapter-Schnittstellen ist er einfach in IEC-61499-Applikationen integrierbar und eignet sich besonders für Anwendungen, die eine Begrenzung der Ereignisrate oder eine zeitlich gesteuerte Datenfreigabe erfordern.

AL_LAST_2





Einleitung

Der AL_LAST_2 Funktionsblock führt zwei gleichartige LWORD-Adapter-Signale (**IN1**, **IN2**) auf einen gemeinsamen Ausgang (**OUT**) zusammen, nach dem Prinzip **Last Writer Wins**: welcher der beiden Sockets zuletzt ein Ereignis geschrieben hat, gewinnt, und sein Datenwert wird sofort und unverändert an OUT durchgereicht. Es handelt sich ausdrücklich nicht um ein naives Zusammenführen (z. B. Mittelwert oder ODER-Verknüpfung), sondern um eine reine Arbitrierung nach zeitlicher Reihenfolge.

Schnittstellenstruktur

Adapter

Eingangsadapter (Sockets):

- **IN1**: LWORD-Adapter (unidirectional) - erste Signalquelle
- **IN2**: LWORD-Adapter (unidirectional) - zweite Signalquelle

Ausgangsadapter (Plug):

- **OUT**: LWORD-Adapter (unidirectional) - zusammengeführtes Signal

Funktionsweise

Der Baustein ist als einfacher Basic-FB mit einer 3-Zustands-ECC umgesetzt (START, PASS1, PASS2). Trifft an IN1 ein Ereignis ein, wechselt die ECC nach PASS1: der aktuelle Datenwert von IN1 wird nach OUT kopiert und das Adapter-Ereignis an OUT gesendet. Trifft an IN2 ein Ereignis ein, geschieht dasselbe

symmetrisch über PASS2. Nach jeder Durchreichung kehrt die ECC sofort wieder nach START zurück und ist bereit für das nächste Ereignis von IN1 oder IN2.

Technische Besonderheiten

- Last-Writer-Wins-Semantik statt naiver Datenverknüpfung
- Einfacher Basic-FB (ECC mit 3 Zuständen), kein zusätzlicher Zwischenspeicher nötig
- Keine Signalverzögerung: der Wert wird im selben Zyklus durchgereicht, in dem das auslösende Ereignis eintrifft
- Beide Eingänge sind gleichberechtigt - es gibt keine feste Priorität außer der zeitlichen Reihenfolge

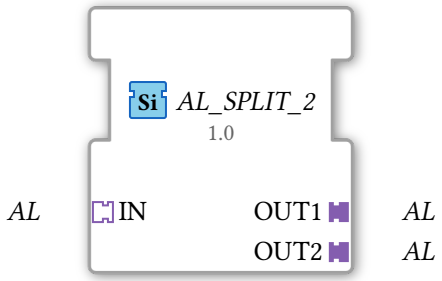
Anwendungsszenarien

- Zusammenführen zweier gleichwertiger LWORD-Signalquellen zu einem einzigen nachgeschalteten Adapterpfad (z. B. zwei redundante Sensoren, oder ein manueller und ein automatischer Sollwertgeber)
- Arbitrierung zwischen zwei Bedienstellen, die denselben LWORD-Wert schreiben dürfen
- Vereinfachung von Netzwerken, die sonst ein `F_SEL`/`E_RS`-Konstrukt inline verdrahten müssten

⚖️ Vergleich mit ähnlichen Bausteinen

AL_LAST_2 ist die Umkehrung von `AL_SPLIT_2`: AL_SPLIT_2 verteilt 1 Eingang auf 2 Ausgänge, AL_LAST_2 führt 2 Eingänge auf 1 Ausgang zusammen. Für Mischbetrieb zwischen einem Daten- und einem reinen Ereignisadapter (z. B. AX + AE) ist stattdessen `AX_AE_MERGE` gedacht, das nur das Ereignis mischt, ohne den Datenwert zu verändern.

AL_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AL_SPLIT_2** dient dazu, ein eingehendes unidirektionales Adapter-Signal auf zwei identische Ausgänge aufzuteilen. Er ist als generischer Baustein konzipiert, sodass der konkrete Adaptertyp erst bei der Verwendung festgelegt wird.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Typ	Richtung	Name	Beschreibung

adapter::types::unidirectional::AL	**Socket** (Eingang)	IN	Eingangssignal, das auf beide Ausgänge verteilt wird
adapter::types::unidirectional::AL	**Plug** (Ausgang)	OUT1	Erster Ausgang mit demselben Adapter-Signal wie IN
adapter::types::unidirectional::AL	**Plug** (Ausgang)	OUT2	Zweiter Ausgang mit demselben Adapter-Signal wie IN

Funktionsweise

Der Baustein leitet das am Socket IN anliegende unidirektionale Adapter-Signal unverändert an beide Plugs OUT1 und OUT2 weiter. Es findet keine Datenbearbeitung oder -transformation statt – der FB arbeitet rein passiv und durchschleifend. Sobald ein Signal am Eingang anliegt, steht es zeitgleich an beiden Ausgängen zur Verfügung.

Technische Besonderheiten

- **Generischer Typ**: Der FB ist als generischer Baustein definiert (Attribute GenericClassName und TypeHash). Dadurch kann er mit unterschiedlichen unidirektionalen Adapter-Instanzen verwendet werden, ohne dass der Quellcode angepasst werden muss.
- **Adapter statt Datenpunkte**: Die Signalübertragung erfolgt ausschließlich über Adapter, nicht über diskrete Ereignis- oder Dateneingänge/-ausgänge. Dies ermöglicht eine strukturierte und wiederverwendbare Kapselung von Kommunikationsschnittstellen.
- **Keine Zustandsabhängigkeit**: Der FB besitzt kein internes Gedächtnis und keine Zustandsmaschine. Die Ausgangssignale folgen direkt dem Eingangssignal.

Zustandsübersicht

Der Baustein führt keine eigene Zustandsverwaltung. Er kann als **kombinatorischer** Block betrachtet werden, bei dem die Ausgänge stets mit dem aktuellen Eingangswert belegt sind. Ein expliziter Zustandsautomat ist nicht vorhanden.

Anwendungsszenarien

- **Signalverteilung**: Wenn ein Adapter-Signal von einer Quelle an mehrere unabhängige Empfänger weitergegeben werden muss (z. B. ein Steuerbefehl an zwei Aktoren).
- **Protokollierung / Überwachung**: Ein Adapter-Signal kann auf einen Pfad zur eigentlichen Verarbeitung und gleichzeitig auf einen Überwachungs- oder Logging-Pfad gesplittet werden.
- **Modulare Architekturen**: In komplexen Systemen, die mit Adaptern arbeiten, vereinfacht der AL_SPLIT_2 die topologische Aufteilung von Signalen, ohne dass manuelle Verbindungen oder Duplikate erstellt werden müssen.

Vergleich mit ähnlichen Bausteinen

Es existieren in der 4diac-IDE verschiedene Splitter-Bausteine für unterschiedliche Signalarten:

Baustein	Signalart	Ausgänge	Besonderheit
AL_SPLIT_2	Adapter (unidirektional)	2	Generisch, keine Daten-/Ereignisse
SPLIT	Ereignisse	2	Verteilt Ereignisse ohne Daten
E_SPLIT	Ereignisse (mit Daten)	2	Verteilt Ereignisse samt begleitender Daten
F_SPLIT	Daten (beliebiger Typ)	n	Verteilt einen Datenwert auf mehrere Ausgänge

Der AL_SPLIT_2 ist speziell für Adapter-Schnittstellen optimiert und bietet damit eine saubere Möglichkeit, komplexe, adapterbasierte Verbindungen aufzutrennen, ohne in die Daten- oder Ereignisebene einzugreifen.

- AL_SPLIT_2_UNGATED: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

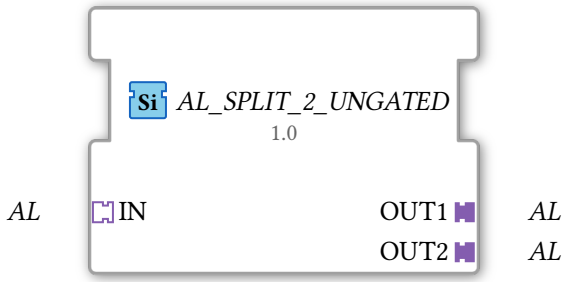
Der `AL_SPLIT_2` ist ein einfacher, aber nützlicher generischer Funktionsblock zur Aufteilung unidirektionaler Adapter-Signale. Er ermöglicht eine flexible und strukturierte Signalverteilung in modularen Automatisierungslösungen, ohne zusätzliche Logik oder Zustandsabhängigkeiten. Sein generischer Charakter macht ihn universell einsetzbar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AL_SPLIT_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AL_SPLIT_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AL_SPLIT_2_UNGATED** dient dazu, ein eingehendes unidirektionales Adapter-Signal auf zwei identische Ausgänge aufzuteilen. Er ist als generischer Baustein konzipiert, sodass der konkrete Adaptertyp erst bei der Verwendung festgelegt wird.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Typ	Richtung	Name	Beschreibung
adapter::types::unidirectional::AL	**Socket** (Eingang)	IN	Eingangssignal, das auf beide Ausgänge verteilt wird
adapter::types::unidirectional::AL	**Plug** (Ausgang)	OUT1	Erster Ausgang mit demselben Adapter-Signal wie IN
adapter::types::unidirectional::AL	**Plug** (Ausgang)	OUT2	Zweiter Ausgang mit demselben Adapter-Signal wie IN

Funktionsweise

Der Baustein leitet das am Socket IN anliegende unidirektionale Adapter-Signal unverändert an beide Plugs OUT1 und OUT2 weiter. Es findet keine Datenbearbeitung oder -transformation statt – der FB arbeitet rein passiv und durchschleifend. Sobald ein Signal am Eingang anliegt, steht es zeitgleich an beiden Ausgängen zur Verfügung.

Technische Besonderheiten

- **Generischer Typ**: Der FB ist als generischer Baustein definiert (Attribute GenericClassName und TypeHash). Dadurch kann er mit unterschiedlichen unidirektionalen Adapter-Instanzen verwendet werden, ohne dass der Quellcode angepasst werden muss.
- **Adapter statt Datenpunkte**: Die Signalübertragung erfolgt ausschließlich über Adapter, nicht über diskrete Ereignis- oder Dateneingänge/-ausgänge. Dies ermöglicht eine strukturierte und wiederverwendbare Kapselung von Kommunikationsschnittstellen.
- **Keine Zustandsabhängigkeit**: Der FB besitzt kein internes Gedächtnis und keine Zustandsmaschine. Die Ausgangssignale folgen direkt dem Eingangssignal.

Zustandsübersicht

Der Baustein führt keine eigene Zustandsverwaltung. Er kann als **kombinatorischer** Block betrachtet werden, bei dem die Ausgänge stets mit dem aktuellen Eingangswert belegt sind. Ein expliziter Zustandsautomat ist nicht vorhanden.

Anwendungsszenarien

- **Signalverteilung**: Wenn ein Adapter-Signal von einer Quelle an mehrere unabhängige Empfänger weitergegeben werden muss (z. B. ein Steuerbefehl an zwei Aktoren).
- **Protokollierung / Überwachung**: Ein Adapter-Signal kann auf einen Pfad zur eigentlichen Verarbeitung und gleichzeitig auf einen Überwachungs- oder Logging-Pfad gesplittet werden.
- **Modulare Architekturen**: In komplexen Systemen, die mit Adaptern arbeiten, vereinfacht der AL_SPLIT_2_UNGATED die topologische Aufteilung von Signalen, ohne dass manuelle Verbindungen oder Duplikate erstellt werden müssen.

Vergleich mit ähnlichen Bausteinen

Es existieren in der 4diac-IDE verschiedene Splitter-Bausteine für unterschiedliche Signalarten:

Baustein	Signalart	Ausgänge	Besonderheit
AL_SPLIT_2_UNGATED	Adapter (unidirektional)	2	Generisch, keine Daten-/Ereignisse
SPLIT	Ereignisse	2	Verteilt Ereignisse ohne Daten
E_SPLIT	Ereignisse (mit Daten)	2	Verteilt Ereignisse samt begleitender Daten
F_SPLIT	Daten (beliebiger Typ)	n	Verteilt einen Datenwert auf mehrere Ausgänge

Der AL_SPLIT_2_UNGATED ist speziell für Adapter-Schnittstellen optimiert und bietet damit eine saubere Möglichkeit, komplexe, adapterbasierte Verbindungen aufzutrennen, ohne in die Daten- oder Ereignisebene einzugreifen.

- AL_SPLIT_2: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

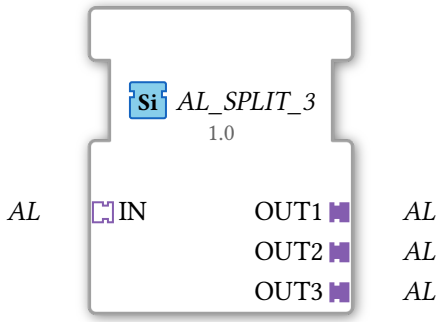
Fazit

Der `AL_SPLIT_2_UNGATED` ist ein einfacher, aber nützlicher generischer Funktionsblock zur Aufteilung unidirektionaler Adapter-Signale. Er ermöglicht eine flexible und strukturierte Signalverteilung in modularen Automatisierungslösungen, ohne zusätzliche Logik oder Zustandsabhängigkeiten. Sein generischer Charakter macht ihn universell einsetzbar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AL_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AL_SPLIT_3** dient der Aufteilung eines eingehenden unidirektionalen Adapter-Signals (AL – Adapter Label) auf drei gleichartige Ausgangs-Adapter. Er ist als generischer Baustein konzipiert und ermöglicht die mehrfache Weiterleitung eines Adapter-basierten Daten- oder Ereignisflusses innerhalb einer IEC 61499 Anwendung.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AL`	IN	Socket	Eingangs-Adapter, der das zu verteilende Signal bereitstellt.
`adapter::types::unidirectional::AL`	OUT1	Plug	Erster Ausgangs-Adapter – identische Kopie des Eingangssignals.

`adapter::types::unidirectional::AL`	OUT2	Plug	Zweiter Ausgangs-Adapter – identische Kopie des Eingangssignals.
`adapter::types::unidirectional::AL`	OUT3	Plug	Dritter Ausgangs-Adapter – identische Kopie des Eingangssignals.

Funktionsweise

Der FB leitet das am Socket **IN** anliegende unidirektionale Adapter-Signal **unverändert und verzögerungsfrei** an die drei Plugs **OUT1**, **OUT2** und **OUT3** weiter. Es findet keine logische oder zeitliche Manipulation statt – die Verteilung ist rein struktureller Natur. Das Verhalten entspricht einer passiven Verdrahtung („Fan-Out“) auf Adapterebene.

Technische Besonderheiten

- **Generische Implementierung:** Der Baustein verwendet den generischen Klassennamen `'GEN_AL_SPLIT'`, sodass er für verschiedene Ausprägungen des unidirektionalen AL-Typs wiederverwendet werden kann, solange der konkrete Typ zur Entwurfszeit festgelegt wird.
- **Keine Zustandslogik:** Es existieren keine internen Zustände, Ereignisse oder Datenvariablen. Der FB ist vollständig passiv und benötigt keine aktive Verarbeitung.
- **Keine Laufzeitabhängigkeiten:** Die Funktionalität wird bereits zur Entwurfszeit durch das Verschaltungssystem der 4diac-IDE realisiert.

Zustandsübersicht

Der FB besitzt **keinen internen Zustand** und keine Zustandsmaschine. Er führt keine sequenziellen Abläufe aus. Das Ausgangssignal ist jederzeit eine direkte Kopie des Eingangssignals.

Anwendungsszenarien

- **Mehrfachnutzung eines Adapters:** Ein von einer Komponente bereitgestelltes unidirektionales AL-Signal soll an mehrere nachfolgende Funktionsblöcke verteilt werden (z. B. Sensorsignal an verschiedene Auswerteeinheiten).
- **Architektur-Strukturierung:** Aufteilung eines Datenstroms zur parallelen Verarbeitung oder Überwachung, ohne dass die Quellkomponente die Anzahl der Senken kennen muss.
- **Test- und Diagnosezwecke:** Anbinden zusätzlicher Überwachungs- oder Logging-Blöcke an eine bestehende Adapter-Verbindung.

Vergleich mit ähnlichen Bausteinen

- **Ereignis-basierte Splitter (z. B. E_SPLIT):** Diese teilen Ereignisse auf, während AL_SPLIT_3 Adapter-Objekte (komplexe Datenstrukturen) verteilt. Ereignis-Splitter benötigen oft eine Behandlung von Ereignis-Reihenfolgen.
- **Adapter-Merger (z. B. AL_MERGE):** Führen mehrere Adapter-Signale zu einem zusammen – das Gegenteil der hier beschriebenen Funktion.
- **Individuelle AL-Splitter:** Für andere Ausgangsanzahlen (z. B. AL_SPLIT_2, AL_SPLIT_4) existieren ebenfalls generische Varianten, die sich nur in der Anzahl der Plugs unterscheiden.
- `AL_SPLIT_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

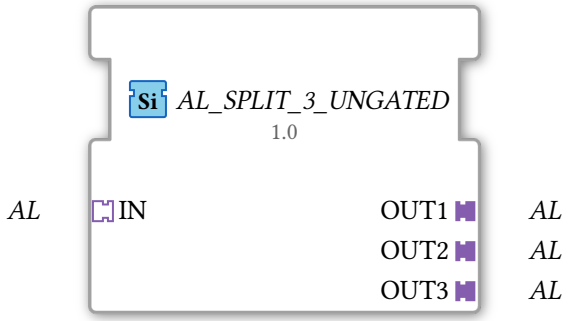
AL_SPLIT_3 ist ein einfacher, aber nützlicher generischer Funktionsblock zur Vervielfältigung eines unidirektionalen Adapter-Signals auf drei unabhängige Ausgänge. Aufgrund seiner passiven Natur ist er ressourcenschonend und eignet sich ideal zur flexiblen Gestaltung von IEC 61499 Anwendungen, in denen ein Signal mehrfach benötigt wird, ohne die Quelllogik zu verändern.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AL_SPLIT_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AL_SPLIT_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AL_SPLIT_3_UNGATED** dient der Aufteilung eines eingehenden unidirektionalen Adapter-Signals (AL – Adapter Label) auf drei gleichartige Ausgangs-Adapter. Er ist als generischer Baustein konzipiert und ermöglicht die mehrfache Weiterleitung eines Adapter-basierten Daten- oder Ereignisflusses innerhalb einer IEC 61499 Anwendung.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AL`	IN	Socket	Eingangs-Adapter, der das zu verteilende Signal bereitstellt.
`adapter::types::unidirectional::AL`	OUT1	Plug	Erster Ausgangs-Adapter – identische Kopie des Eingangssignals.
`adapter::types::unidirectional::AL`	OUT2	Plug	Zweiter Ausgangs-Adapter – identische Kopie des Eingangssignals.

`adapter::types::unidirectional::AL`	OUT3	Plug	Dritter Ausgangs-Adapter – identische Kopie des Eingangssignals.

Funktionsweise

Der FB leitet das am Socket **IN** anliegende unidirektionale Adapter-Signal **unverändert und verzögerungsfrei** an die drei Plugs **OUT1**, **OUT2** und **OUT3** weiter. Es findet keine logische oder zeitliche Manipulation statt – die Verteilung ist rein struktureller Natur. Das Verhalten entspricht einer passiven Verdrahtung („Fan-Out“) auf Adapterebene.

Technische Besonderheiten

- **Generische Implementierung:** Der Baustein verwendet den generischen Klassennamen `'GEN_AL_SPLIT'`, sodass er für verschiedene Ausprägungen des unidirektionalen AL-Typs wiederverwendet werden kann, solange der konkrete Typ zur Entwurfszeit festgelegt wird.
- **Keine Zustandslogik:** Es existieren keine internen Zustände, Ereignisse oder Datenvariablen. Der FB ist vollständig passiv und benötigt keine aktive Verarbeitung.
- **Keine Laufzeitabhängigkeiten:** Die Funktionalität wird bereits zur Entwurfszeit durch das Verschaltungssystem der 4diac-IDE realisiert.

Zustandsübersicht

Der FB besitzt **keinen internen Zustand** und keine Zustandsmaschine. Er führt keine sequenziellen Abläufe aus. Das Ausgangssignal ist jederzeit eine direkte Kopie des Eingangssignals.

Anwendungsszenarien

- **Mehrfachnutzung eines Adapters:** Ein von einer Komponente bereitgestelltes unidirektionales AL-Signal soll an mehrere nachfolgende Funktionsblöcke verteilt werden (z. B. Sensorsignal an verschiedene Auswerteeinheiten).
- **Architektur-Strukturierung:** Aufteilung eines Datenstroms zur parallelen Verarbeitung oder Überwachung, ohne dass die Quellkomponente die Anzahl der Senken kennen muss.
- **Test- und Diagnosezwecke:** Anbinden zusätzlicher Überwachungs- oder Logging-Blöcke an eine bestehende Adapter-Verbindung.

Vergleich mit ähnlichen Bausteinen

- **Ereignis-basierte Splitter (z. B. E_SPLIT):** Diese teilen Ereignisse auf, während AL_SPLIT_3_UNGATED Adapter-Objekte (komplexe Datenstrukturen) verteilt. Ereignis-Splitter benötigen oft eine Behandlung von Ereignis-Reihenfolgen.
- **Adapter-Merger (z. B. AL_MERGE):** Führen mehrere Adapter-Signale zu einem zusammen – das Gegenteil der hier beschriebenen Funktion.
- **Individuelle AL-Splitter:** Für andere Ausgangsanzahlen (z. B. AL_SPLIT_2, AL_SPLIT_4) existieren ebenfalls generische Varianten, die sich nur in der Anzahl der Plugs unterscheiden.
- `AL_SPLIT_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

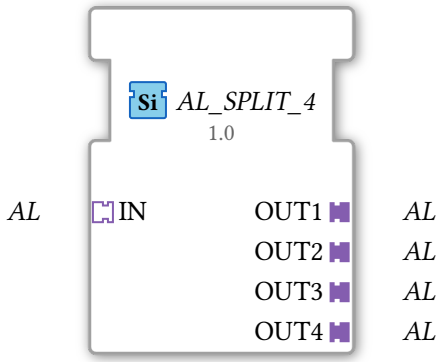
Fazit

AL_SPLIT_3_UNGATED ist ein einfacher, aber nützlicher generischer Funktionsblock zur Vervielfältigung eines unidirektionalen Adapter-Signals auf drei unabhängige Ausgänge. Aufgrund seiner passiven Natur ist er ressourcenschonend und eignet sich ideal zur flexiblen Gestaltung von IEC 61499 Anwendungen, in denen ein Signal mehrfach benötigt wird, ohne die Quelllogik zu verändern.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AL_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AL_SPLIT_4** ist ein generischer Baustein, der einen einzelnen Adapter-Eingang (Socket) vom Typ `adapter::types::unidirectional::AL` auf vier separate Ausgänge (Plugs) desselben Typs aufteilt. Er dient dazu, einen eingehenden Adapter-Datenstrom auf mehrere parallele Verbraucher zu verteilen, ohne dass die Daten selbst verändert werden.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AL`	**IN**	Socket (Eingang)	Eingehender Adapter-Datenstrom, der auf alle Ausgänge verteilt wird.
`adapter::types::unidirectional::AL`	**OUT1**	Plug (Ausgang)	Erster Ausgang – erhält eine identische Kopie des Eingangsdatenstroms.
`adapter::types::unidirectional::AL`	**OUT2**	Plug (Ausgang)	Zweiter Ausgang – erhält eine identische Kopie des Eingangsdatenstroms.
`adapter::types::unidirectional::AL`	**OUT3**	Plug (Ausgang)	Dritter Ausgang – erhält eine identische Kopie des Eingangsdatenstroms.
`adapter::types::unidirectional::AL`	**OUT4**	Plug (Ausgang)	Vierter Ausgang – erhält eine identische Kopie des Eingangsdatenstroms.

Funktionsweise

Der Baustein ist ein reiner Signalverteiler. Er besitzt eine interne Logik, die – sobald der Adapter-Eingang **IN** mit einer Datenquelle verbunden ist – die eingehenden Datenpakete (Events und Daten) unmittelbar und unverändert an alle vier Ausgangsadapter **OUT1** bis **OUT4** weiterleitet. Es findet keine Pufferung, Verzögerung oder logische Verarbeitung statt. Jeder Ausgang erhält exakt dieselben Informationen wie der Eingang.

Technische Besonderheiten

- **Generischer Baustein:** Die Klasse `GEN_AL_SPLIT` ist im Attribut `eclipse4diac::core::GenericClassName` definiert, was bedeutet, dass der Baustein als Vorlage für verschiedene Ausprägungen des Adaptertyps `AL` wiederverwendet werden kann.
- **Keine Zustände oder Zeitsteuerung:** Da der FB ausschließlich leitungsbasiert arbeitet und keinerlei Verarbeitungslogik besitzt, existieren keine internen Zustände oder nebenläufige Ausführungssequenzen.
- **Adapter-Typ:** Der verwendete Adapter `adapter::types::unidirectional::AL` ist ein standardisierter, unidirektionaler Adapter aus der Bibliothek `adapter::types::unidirectional`. Seine genaue Schnittstelle (enthaltene Events und Daten) wird durch die Adapterdefinition vorgegeben und nicht durch den Split-Baustein selbst beeinflusst.

Zustandsübersicht

Der Baustein besitzt keinen Zustandsautomaten. Er arbeitet statisch und verteilt das Eingangssignal permanent auf alle Ausgänge.

Anwendungsszenarien

- **Verteilen eines Steuerungssignals** an mehrere parallele Aktoreinheiten, z. B. in der Agrartechnik (wie im Copyright-Hinweis der HR Agrartechnik GmbH angedeutet).
- **Aufteilen einer Messdatenquelle** auf mehrere Analyse- oder Logging-Module.
- **Testen und Debugging:** Ein Original-Adaptersignal kann auf mehrere Ausgänge geführt werden, um es gleichzeitig an verschiedene Test-Instanzen zu senden.

Vergleich mit ähnlichen Bausteinen

- **AL_SPLIT_2** – teilt ein AL-Signal auf zwei Ausgänge auf (weniger Ausgänge, geringere Fan-out-Kapazität).
- **AL_MERGE** – führt mehrere AL-Eingänge zu einem Ausgang zusammen (umgekehrte Funktion).
- **Ereignisbasierte Split-Bausteine (z. B. SPLIT_EVENT)** – teilen nur Ereignisse auf, ohne die assoziierten Datenadapter zu duplizieren.

Im Gegensatz zu diesen Bausteinen arbeitet `AL_SPLIT_4` ausschließlich auf Adapterebene und kopiert vollständige Adapter-Kommunikationsströme.

- `AL_SPLIT_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

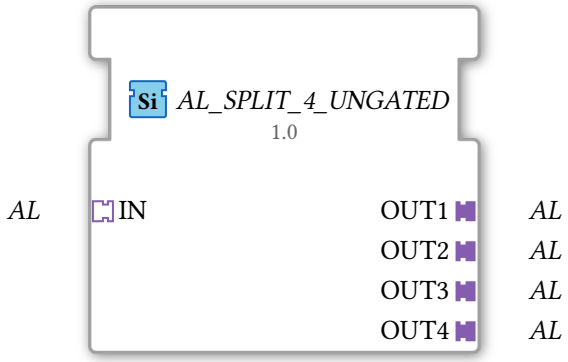
Der Funktionsblock **AL_SPLIT_4** ist ein einfacher, effizienter Signalfächer für den Adaptertyp `AL`. Er eignet sich ideal für alle Anwendungen, in denen ein einzelner Adapterdatenstrom an mehrere Verbraucher weitergegeben werden muss, ohne zusätzliche Logik oder Verzögerungen. Seine generische Natur erlaubt den Einsatz in unterschiedlichen Kontexten der Automatisierungstechnik, insbesondere in der Landmaschinensteuerung, wie vom Hersteller vorgesehen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AL_SPLIT_4_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `AL_SPLIT_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AL_SPLIT_4_UNGATED** ist ein generischer Baustein, der einen einzelnen Adapter-Eingang (Socket) vom Typ `adapter::types::unidirectional::AL` auf vier separate Ausgänge (Plugs) desselben Typs aufteilt. Er dient dazu, einen eingehenden Adapter-Datenstrom auf mehrere parallele Verbraucher zu verteilen, ohne dass die Daten selbst verändert werden.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AL`	**IN**	Socket (Eingang)	Eingehender Adapter-Datenstrom, der auf alle Ausgänge verteilt wird.
`adapter::types::unidirectional::AL`	**OUT1**	Plug (Ausgang)	Erster Ausgang – erhält eine identische Kopie des Eingangsdatenstroms.
`adapter::types::unidirectional::AL`	**OUT2**	Plug (Ausgang)	Zweiter Ausgang – erhält eine identische Kopie des Eingangsdatenstroms.

adapter::types::unidirectional::AL	**OUT3**	Plug (Ausgang)	Dritter Ausgang – erhält eine identische Kopie des Eingangsdatenstroms.
adapter::types::unidirectional::AL	**OUT4**	Plug (Ausgang)	Vierter Ausgang – erhält eine identische Kopie des Eingangsdatenstroms.

Funktionsweise

Der Baustein ist ein reiner Signalverteiler. Er besitzt eine interne Logik, die – sobald der Adapter-Eingang **IN** mit einer Datenquelle verbunden ist – die eingehenden Datenpakete (Events und Daten) unmittelbar und unverändert an alle vier Ausgangsadapter **OUT1** bis **OUT4** weiterleitet. Es findet keine Pufferung, Verzögerung oder logische Verarbeitung statt. Jeder Ausgang erhält exakt dieselben Informationen wie der Eingang.

Technische Besonderheiten

- **Generischer Baustein:** Die Klasse `GEN_AL_SPLIT` ist im Attribut `eclipse4diac::core::GenericClassName` definiert, was bedeutet, dass der Baustein als Vorlage für verschiedene Ausprägungen des Adaptertyps `AL` wiederverwendet werden kann.
- **Keine Zustände oder Zeitsteuerung:** Da der FB ausschließlich leitungsbasiert arbeitet und keinerlei Verarbeitungslogik besitzt, existieren keine internen Zustände oder nebenläufige Ausführungssequenzen.
- **Adapter-Typ:** Der verwendete Adapter `adapter::types::unidirectional::AL` ist ein standardisierter, unidirektionaler Adapter aus der Bibliothek `adapter::types::unidirectional`. Seine genaue Schnittstelle (enthaltene Events und Daten) wird durch die Adapterdefinition vorgegeben und nicht durch den Split-Baustein selbst beeinflusst.

Zustandsübersicht

Der Baustein besitzt keinen Zustandsautomaten. Er arbeitet statisch und verteilt das Eingangssignal permanent auf alle Ausgänge.

Anwendungsszenarien

- **Verteilen eines Steuerungssignals** an mehrere parallele Aktoreinheiten, z. B. in der Agrartechnik (wie im Copyright-Hinweis der HR Agrartechnik GmbH angedeutet).
- **Aufteilen einer Messdatenquelle** auf mehrere Analyse- oder Logging-Module.
- **Testen und Debugging:** Ein Original-Adaptersignal kann auf mehrere Ausgänge geführt werden, um es gleichzeitig an verschiedene Test-Instanzen zu senden.

Vergleich mit ähnlichen Bausteinen

- **AL_SPLIT_2** – teilt ein AL-Signal auf zwei Ausgänge auf (weniger Ausgänge, geringere Fan-out-Kapazität).
- **AL_MERGE** – führt mehrere AL-Eingänge zu einem Ausgang zusammen (umgekehrte Funktion).
- **Ereignisbasierte Split-Bausteine (z. B. SPLIT_EVENT)** – teilen nur Ereignisse auf, ohne die assoziierten Datenadapter zu duplizieren.

Im Gegensatz zu diesen Bausteinen arbeitet `AL_SPLIT_4_UNGATED` ausschließlich auf Adapterebene und kopiert vollständige Adapter-Kommunikationsströme.

- `AL_SPLIT_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

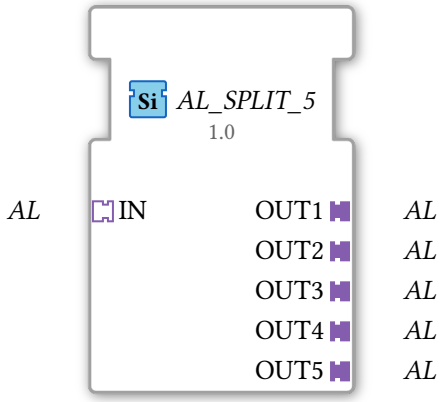
Der Funktionsblock **AL_SPLIT_4_UNGATED** ist ein einfacher, effizienter Signalfächer für den Adaptertyp `AL`. Er eignet sich ideal für alle Anwendungen, in denen ein einzelner Adapterdatenstrom an mehrere Verbraucher weitergegeben werden muss, ohne zusätzliche Logik oder Verzögerungen. Seine generische

Natur erlaubt den Einsatz in unterschiedlichen Kontexten der Automatisierungstechnik, insbesondere in der Landmaschinensteuerung, wie vom Hersteller vorgesehen.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AL_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AL_SPLIT_5** dient als generischer Splitter für einen unidirektionalen Adapter vom Typ `AL`. Er teilt ein eingehendes Signal (über den `IN`-Socket) auf fünf separate Ausgänge (`OUT1` bis `OUT5`) auf. Der Baustein ist als generisch gekennzeichnet, wodurch der konkrete Adaptertyp beim Einsatz festgelegt werden kann.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Typ	Richtung	Name	Beschreibung
Socket	Eingang	IN	Unidirektionaler `AL`-Adapter-Eingang
Plug	Ausgang	OUT1	Erster Ausgang (unidirektionaler `AL`)
Plug	Ausgang	OUT2	Zweiter Ausgang (unidirektionaler `AL`)
Plug	Ausgang	OUT3	Dritter Ausgang (unidirektionaler `AL`)
Plug	Ausgang	OUT4	Vierter Ausgang (unidirektionaler `AL`)
Plug	Ausgang	OUT5	Fünfter Ausgang (unidirektionaler `AL`)

Funktionsweise

Der Baustein empfängt über den Socket `IN` ein Signal des Typs `AL` und leitet dieses unverändert an alle fünf Plugs `OUT1` bis `OUT5` weiter. Es findet keine Datenverarbeitung oder Ereignisbehandlung statt; die Aufteilung erfolgt rein auf Adapterebene. Dadurch können mehrere nachgeschaltete Komponenten parallel mit dem gleichen Adaptersignal versorgt werden.

Der Funktionsblock ist als generischer Baustein (`GEN_AL_SPLIT`) ausgelegt. Der konkrete Adaptertyp `AL` kann bei der Instanziierung über ein Attribut festgelegt werden.

Technische Besonderheiten

- **Keine Ereignisse oder Daten**: Der Baustein besitzt ausschließlich Adapterschnittstellen, keine Ereignis- oder Datenports.
- **Generische Auslegung**: Der Typ des `AL`-Adapters ist parametrierbar (generic class), sodass verschiedene Ausprägungen desselben Adaptertyps verwendet werden können.
- **reine Signalverteilung**: Es handelt sich um eine reine 1:5-Verteilung ohne jegliche Logik, Verzögerung oder Transformation.

Zustandsübersicht

Der Baustein besitzt keinen internen Zustandsautomaten, da er keine Ereignisse verarbeitet. Er verhält sich statisch und leitet das Eingangssignal permanent an alle Ausgänge weiter.

Anwendungsszenarien

- **Signalverteilung**: Wenn mehrere Komponenten dasselbe Adaptersignal eines `AL`-Typs benötigen, z. B. um einen Sensorwert an verschiedene Steuerungslogiken zu senden.
- **Test- und Debugging-Umgebungen**: Ein eingehendes Adaptersignal kann an mehrere Monitor- oder Logging-Bausteine gleichzeitig weitergegeben werden.
- **Redundanz**: Das Signal kann an parallele Verarbeitungspfade verteilt werden.

Vergleich mit ähnlichen Bausteinen

Im 4diac-Ökosystem existieren weitere Splitter-Bausteine, z. B. `SPLIT` für Ereignis- oder Datenflüsse. Der Vorteil von `AL_SPLIT_5` liegt in der ausschließlichen Adapter-Schnittstelle. Während ein `SPLIT`-Baustein Ereignisse oder Daten trennt, arbeitet dieser Baustein direkt auf der Adapterebene und erfordert keine Umsetzung zwischen Ereignissen und Daten. Ähnliche Bausteine mit anderer Anzahl von Ausgängen (z. B. `AL_SPLIT_2`, `AL_SPLIT_3`) sind denkbar, jedoch nicht standardmäßig in dieser generischen Form enthalten.

- `AL_SPLIT_5_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der **AL_SPLIT_5** ist ein einfacher, aber nützlicher generischer Funktionsblock zur Vervielfältigung eines unidirektionalen Adaptersignals auf fünf Ausgänge. Aufgrund seiner reinen Adapterschnittstelle und fehlenden Ereignis-/Datenlogik eignet er sich besonders für Anwendungen, die ein Signal ohne Verarbeitung an mehrere Empfänger verteilen müssen. Die generische Auslegung erhöht die Flexibilität bei der Wiederverwendung in verschiedenen Projekten.

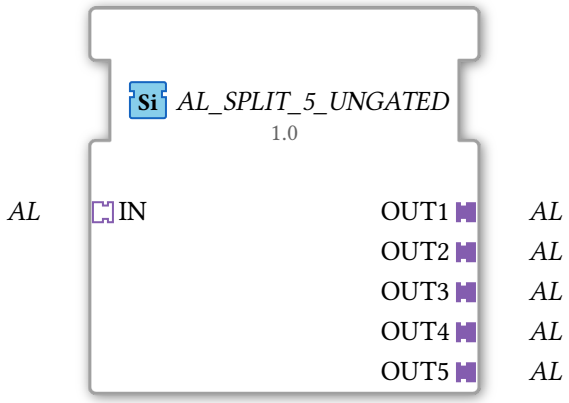
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AL_SPLIT_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AL_SPLIT_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weiter-

gegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AL_SPLIT_5_UNGATED** dient als generischer Splitter für einen unidirektionalen Adapter vom Typ `AL`. Er teilt ein eingehendes Signal (über den `IN`-Socket) auf fünf separate Ausgänge (`OUT1` bis `OUT5`) auf. Der Baustein ist als generisch gekennzeichnet, wodurch der konkrete Adaptertyp beim Einsatz festgelegt werden kann.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Typ	Richtung	Name	Beschreibung
Socket	Eingang	IN	Unidirektionaler `AL`-Adapter-Eingang
Plug	Ausgang	OUT1	Erster Ausgang (unidirektionaler `AL`)
Plug	Ausgang	OUT2	Zweiter Ausgang (unidirektionaler `AL`)
Plug	Ausgang	OUT3	Dritter Ausgang (unidirektionaler `AL`)
Plug	Ausgang	OUT4	Vierter Ausgang (unidirektionaler `AL`)
Plug	Ausgang	OUT5	Fünfter Ausgang (unidirektionaler `AL`)

Funktionsweise

Der Baustein empfängt über den Socket `IN` ein Signal des Typs `AL` und leitet dieses unverändert an alle fünf Plugs `OUT1` bis `OUT5` weiter. Es findet keine Datenverarbeitung oder Ereignisbehandlung statt;

die Aufteilung erfolgt rein auf Adapterebene. Dadurch können mehrere nachgeschaltete Komponenten parallel mit dem gleichen Adaptersignal versorgt werden.

Der Funktionsblock ist als generischer Baustein (`GEN_AL_SPLIT`) ausgelegt. Der konkrete Adaptertyp `AL` kann bei der Instanziierung über ein Attribut festgelegt werden.

Technische Besonderheiten

- **Keine Ereignisse oder Daten**: Der Baustein besitzt ausschließlich Adapterschnittstellen, keine Ereignis- oder Datenports.
- **Generische Auslegung**: Der Typ des `AL`-Adapters ist parametrierbar (generic class), sodass verschiedene Ausprägungen desselben Adaptertyps verwendet werden können.
- **reine Signalverteilung**: Es handelt sich um eine reine 1:5-Verteilung ohne jegliche Logik, Verzögerung oder Transformation.

Zustandsübersicht

Der Baustein besitzt keinen internen Zustandsautomaten, da er keine Ereignisse verarbeitet. Er verhält sich statisch und leitet das Eingangssignal permanent an alle Ausgänge weiter.

Anwendungsszenarien

- **Signalverteilung**: Wenn mehrere Komponenten dasselbe Adaptersignal eines `AL`-Typs benötigen, z. B. um einen Sensorwert an verschiedene Steuerungslogiken zu senden.
- **Test- und Debugging-Umgebungen**: Ein eingehendes Adaptersignal kann an mehrere Monitor- oder Logging-Bausteine gleichzeitig weitergegeben werden.
- **Redundanz**: Das Signal kann an parallele Verarbeitungspfade verteilt werden.

Vergleich mit ähnlichen Bausteinen

Im 4diac-Ökosystem existieren weitere Splitter-Bausteine, z. B. `SPLIT` für Ereignis- oder Datenflüsse. Der Vorteil von `AL_SPLIT_5_UNGATED` liegt in der ausschließlichen Adapter-Schnittstelle. Während ein `SPLIT`-Baustein Ereignisse oder Daten trennt, arbeitet dieser Baustein direkt auf der Adapterebene und erfordert keine Umsetzung zwischen Ereignissen und Daten. Ähnliche Bausteine mit anderer Anzahl von Ausgängen (z. B. `AL_SPLIT_2`, `AL_SPLIT_3`) sind denkbar, jedoch nicht standardmäßig in dieser generischen Form enthalten.

- `AL_SPLIT_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

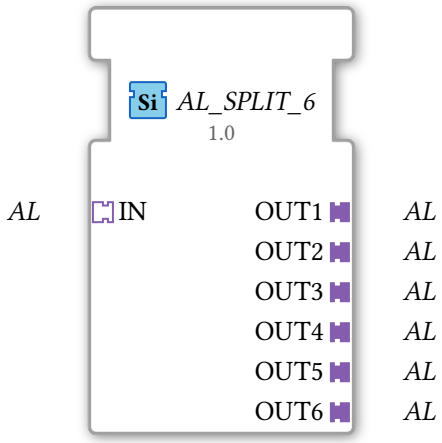
Fazit

Der **AL_SPLIT_5_UNGATED** ist ein einfacher, aber nützlicher generischer Funktionsblock zur Vervielfältigung eines unidirektionalen Adaptersignals auf fünf Ausgänge. Aufgrund seiner reinen Adapterschnittstelle und fehlenden Ereignis-/Datenlogik eignet er sich besonders für Anwendungen, die ein Signal ohne Verarbeitung an mehrere Empfänger verteilen müssen. Die generische Auslegung erhöht die Flexibilität bei der Wiederverwendung in verschiedenen Projekten.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AL_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AL_SPLIT_6** dient dazu, einen einzelnen unidirektionalen AL-Adapter-Eingang (Socket) auf sechs identische AL-Adapter-Ausgänge (Plugs) aufzuteilen. Er ermöglicht eine einfache Signalverteilung innerhalb einer 4diac-Applikation, ohne dass zusätzliche Logik oder eine Zustandsmaschine erforderlich ist.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Richtung	Bezeichner	Typ	Beschreibung
In	IN	adapter::types::unidirectional::AL	Unidirektionaler AL-Adapter als Eingangssignal.
Out	OUT1	adapter::types::unidirectional::AL	Erster Ausgangskanal (identisch zu IN).
Out	OUT2	adapter::types::unidirectional::AL	Zweiter Ausgangskanal.
Out	OUT3	adapter::types::unidirectional::AL	Dritter Ausgangskanal.
Out	OUT4	adapter::types::unidirectional::AL	Vierter Ausgangskanal.
Out	OUT5	adapter::types::unidirectional::AL	Fünfter Ausgangskanal.
Out	OUT6	adapter::types::unidirectional::AL	Sechster Ausgangskanal.

Funktionsweise

Der Baustein besitzt keinerlei interne Logik oder Zustandsmaschine. Er leitet das am Socket **IN** ankommende AL-Signal unverändert an alle sechs Plugs **OUT1** … **OUT6** weiter. Jede Änderung oder jedes

Ereignis am Eingang wird somit simultan auf alle Ausgänge übertragen. Der FB arbeitet rein strukturell als Verteiler (Splitter) und benötigt keine Aktivierung durch Ereignisse.

Technische Besonderheiten

- **Generischer FB:** Der Baustein nutzt das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `GEN_AL_SPLIT`. Dadurch kann er in 4diac IDE als generischer Typ behandelt werden und erlaubt eine saubere Typsystem-Integration.
- **Keine Daten-/Ereignisschnittstelle:** Die Aufteilung erfolgt ausschließlich über die Adapter-Schnittstelle. Es werden keine zusätzlichen Daten- oder Ereignisports benötigt.
- **Keine Laufzeitabhängigkeiten:** Da keine Algorithmen oder Zustände existieren, ist der Baustein ressourcenschonend und benötigt keine spezifische Initialisierung.

Zustandsübersicht

Der FB besitzt keine Zustandsmaschine (kein ECC) und daher keinen definierten Zustandsautomaten. Das Verhalten ist vollständig durch die Schnittstellenstruktur und die reine Weiterleitung des Adaptersignals bestimmt.

Anwendungsszenarien

- **Signalverteilung in der Landtechnik oder Automatisierung:** Ein zentraler AL-Adapter (z. B. ein Steuerbefehl) soll an mehrere nachgelagerte Komponenten gleichzeitig gesendet werden.
- **Tests und Simulationen:** Ein Testsignal kann parallel an verschiedene Module verteilt werden, um das Systemverhalten zu überprüfen.
- **Redundante Pfade:** Der gleiche AL-Adapter kann über mehrere Kanäle zu unterschiedlichen Verarbeitungseinheiten geführt werden.

Vergleich mit ähnlichen Bausteinen

- **AL_SPLIT_2, AL_SPLIT_4 … AL_SPLIT_N**: Diese Bausteine unterscheiden sich lediglich in der Anzahl der Ausgangskanäle. Die Funktionsweise ist identisch – eine einfache 1:N-Aufteilung.
- **Andere Split-Bausteine mit Ereignissen oder Daten:** Im Unterschied zu diesen besitzt **AL_SPLIT_6** keine Daten- oder Ereignisschnittstellen, sondern arbeitet rein über Adapter. Das vereinfacht die Konfiguration, schränkt aber die Typvielfalt ein.
- `AL_SPLIT_6_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

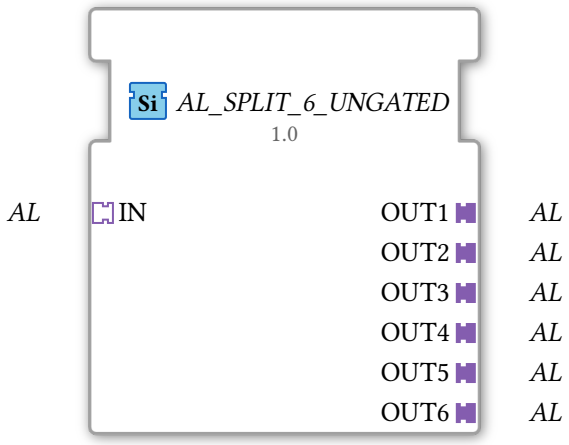
Der **AL_SPLIT_6** ist ein kompakter und effizienter Funktionsblock zur Vervielfachung eines unidirektionalen AL-Adapter-Signals. Durch seinen generischen Charakter und die einfache Struktur eignet er sich hervorragend für alle Anwendungen, bei denen ein Eingangssignal auf mehrere Ausgänge verteilt werden muss, ohne zusätzliche Logik oder Ereignissteuerung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AL_SPLIT_6_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AL_SPLIT_6`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AL_SPLIT_6_UNGATED** dient dazu, einen einzelnen unidirektionalen AL-Adapter-Eingang (Socket) auf sechs identische AL-Adapter-Ausgänge (Plugs) aufzuteilen. Er ermöglicht eine einfache Signalverteilung innerhalb einer 4diac-Applikation, ohne dass zusätzliche Logik oder eine Zustandsmaschine erforderlich ist.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Richtung	Bezeichner	Typ	Beschreibung
In	IN	adapter::types::unidirectional::AL	Unidirektionaler AL-Adapter als Eingangssignal.
Out	OUT1	adapter::types::unidirectional::AL	Erster Ausgangskanal (identisch zu IN).
Out	OUT2	adapter::types::unidirectional::AL	Zweiter Ausgangskanal.
Out	OUT3	adapter::types::unidirectional::AL	Dritter Ausgangskanal.
Out	OUT4	adapter::types::unidirectional::AL	Vierter Ausgangskanal.
Out	OUT5	adapter::types::unidirectional::AL	Fünfter Ausgangskanal.
Out	OUT6	adapter::types::unidirectional::AL	Sechster Ausgangskanal.

Funktionsweise

Der Baustein besitzt keinerlei interne Logik oder Zustandsmaschine. Er leitet das am Socket **IN** ankommende AL-Signal unverändert an alle sechs Plugs **OUT1** … **OUT6** weiter. Jede Änderung oder jedes Ereignis am Eingang wird somit simultan auf alle Ausgänge übertragen. Der FB arbeitet rein strukturell als Verteiler (Splitter) und benötigt keine Aktivierung durch Ereignisse.

Technische Besonderheiten

- **Generischer FB:** Der Baustein nutzt das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `GEN_AL_SPLIT`. Dadurch kann er in 4diac IDE als generischer Typ behandelt werden und erlaubt eine saubere Typsystem-Integration.
- **Keine Daten-/Ereignisschnittstelle:** Die Aufteilung erfolgt ausschließlich über die Adapter-Schnittstelle. Es werden keine zusätzlichen Daten- oder Ereignisports benötigt.
- **Keine Laufzeitabhängigkeiten:** Da keine Algorithmen oder Zustände existieren, ist der Baustein ressourcenschonend und benötigt keine spezifische Initialisierung.

Zustandsübersicht

Der FB besitzt keine Zustandsmaschine (kein ECC) und daher keinen definierten Zustandsautomaten. Das Verhalten ist vollständig durch die Schnittstellenstruktur und die reine Weiterleitung des Adaptersignals bestimmt.

Anwendungsszenarien

- **Signalverteilung in der Landtechnik oder Automatisierung:** Ein zentraler AL-Adapter (z. B. ein Steuerbefehl) soll an mehrere nachgelagerte Komponenten gleichzeitig gesendet werden.
- **Tests und Simulationen:** Ein Testsignal kann parallel an verschiedene Module verteilt werden, um das Systemverhalten zu überprüfen.
- **Redundante Pfade:** Der gleiche AL-Adapter kann über mehrere Kanäle zu unterschiedlichen Verarbeitungseinheiten geführt werden.

Vergleich mit ähnlichen Bausteinen

- **AL_SPLIT_2, AL_SPLIT_4 … AL_SPLIT_N**: Diese Bausteine unterscheiden sich lediglich in der Anzahl der Ausgangskanäle. Die Funktionsweise ist identisch – eine einfache 1:N-Aufteilung.
- **Andere Split-Bausteine mit Ereignissen oder Daten:** Im Unterschied zu diesen besitzt **AL_SPLIT_6_UNGATED** keine Daten- oder Ereignisschnittstellen, sondern arbeitet rein über Adapter. Das vereinfacht die Konfiguration, schränkt aber die Typvielfalt ein.
- `AL_SPLIT_6`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

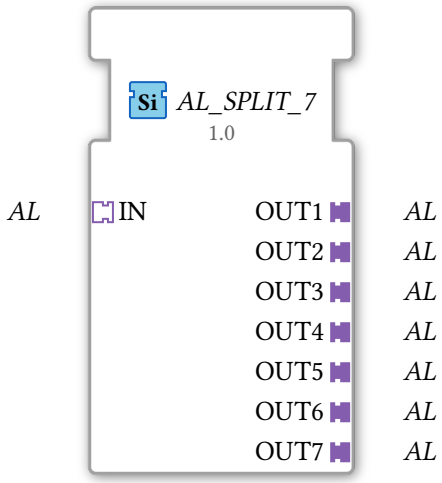
Fazit

Der **AL_SPLIT_6_UNGATED** ist ein kompakter und effizienter Funktionsblock zur Vervielfachung eines unidirektionalen AL-Adapter-Signals. Durch seinen generischen Charakter und die einfache Struktur eignet er sich hervorragend für alle Anwendungen, bei denen ein Eingangssignal auf mehrere Ausgänge verteilt werden muss, ohne zusätzliche Logik oder Ereignissteuerung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AL_SPLIT_7



(Bild des Funktionsblocks nicht verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AL_SPLIT_7** dient als generischer Verteiler (Splitter) für einen unidirektionalen Adapter vom Typ `AL`. Er nimmt an seinem Socket `IN` einen einzigen Adapter entgegen und leitet die ankommenden Daten und Ereignisse unverändert an sieben identische Adapter-Ausgänge (`OUT1` … `OUT7`) weiter. Der Baustein ist als generischer Typ (`GEN_AL_SPLIT`) implementiert und ermöglicht so eine flexible Verwendung in Steuerungs- und Automatisierungsanwendungen.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine expliziten Ereignis-Eingänge. Die gesamte Ereignissteuerung erfolgt über den angeschlossenen Adapter `IN`.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge vorhanden. Ereignisse werden über die sieben Adapter-Ausgänge (`OUT1` … `OUT7`) weitergegeben.

Daten-Eingänge

Der Baustein hat keine eigenständigen Daten-Eingänge. Die Datenübergabe geschieht ausschließlich über den Adapter `IN`.

Daten-Ausgänge

Entsprechend gibt es keine direkten Daten-Ausgänge. Die Daten werden über die sieben Adapter-Ausgänge repliziert.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	`IN`	`adapter::types::unidirectional::AL`	Adapter-Eingang, der die zu verteilende AL-Verbindung bereitstellt.
Plug	`OUT1` … `OUT7`	`adapter::types::unidirectional::AL`	Sieben Adapter-Ausgänge, die die identische Kopie des Eingangsadapter bereitstellen.

Funktionsweise

Der **AL_SPLIT_7** ist ein reiner Verteilerbaustein. Wenn ein Adapter vom Typ `AL` an den Socket `IN` angeschlossen wird, werden alle über diesen Adapter eingehenden Daten und Ereignisse unverändert an

jeden der sieben Plugs (`OUT1` … `OUT7`) weitergeleitet. Es findet keine logische oder zeitliche Bearbeitung statt – die Verteilung erfolgt eins-zu-sieben ohne Verzögerung oder Pufferung. Aufgrund der generischen Implementierung kann der Baustein mit verschiedenen AL-Adapterinstanzen verwendet werden, die denselben Datentyp und dasselbe Ereignisschema aufweisen.

Technische Besonderheiten

- **Generische Ausführung:** Der Baustein ist als generischer Typ (`GEN_AL_SPLIT`) deklariert. Dadurch kann er in unterschiedlichen Projekten ohne Anpassung des Quellcodes verwendet werden, solange der zugrundeliegende Adaptertyp `AL` dem erwarteten Interface entspricht.
- **Unidirektionalität:** Der Baustein arbeitet ausschließlich unidirektional – Daten und Ereignisse fließen vom Socket zu den Plugs. Eine Rückkopplung oder bidirektionale Kommunikation ist nicht vorgesehen.
- **Keine Zustandslogik:** Der Baustein besitzt keine eigene Zustandsmaschine; er ist transparenter Natur.

Zustandsübersicht

Der Baustein hat keine internen Zustände. Er ist jederzeit bereit, eingehende Adapter-Verbindungen zu verteilen. Ein aktiver Zustand ergibt sich allein aus der Anwesenheit eines gültigen Adapters am `IN`-Socket.

Anwendungsszenarien

- **Steuerungsverteilung:** Ein übergeordneter Steuerungsadapter (z. B. zur Ansteuerung einer Achse oder eines Sensors) soll identische Signale an mehrere Parallelkomponenten senden.
- **Testumgebungen:** In Simulations- oder Testsetups können mit einem Adapter mehrere gleichartige Teilsysteme versorgt werden.
- **Signalbündelung:** Der Baustein kann als Verteiler in einer Hierarchie von Adaptern verwendet werden, wenn ein Signal auf mehrere nachfolgende Funktionsbausteine aufgeteilt werden muss.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem **AL_SPLIT_2** oder **AL_SPLIT_4** bietet **AL_SPLIT_7** eine höhere Anzahl von Ausgängen (sieben statt zwei oder vier). Während kleinere Splitter bei geringen Anforderungen ausreichen, ermöglicht dieser Baustein eine direkte Verteilung an sieben Empfänger, ohne mehrere Splitter kaskadieren zu müssen. Gegenüber einem **AL_MERGE** (der mehrere Eingänge zusammenführt) ist die Funktion genau entgegengesetzt.

- `AL_SPLIT_7_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

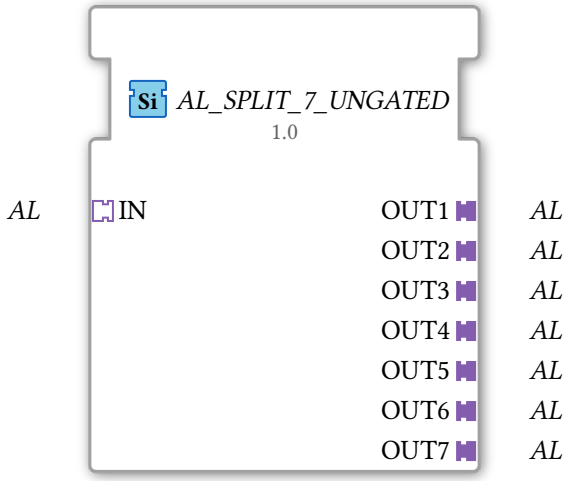
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der **AL_SPLIT_7** ist ein einfacher, aber nützlicher generischer Baustein zur Vervielfältigung von unidirektionalen AL-Adapterverbindungen. Dank seiner generischen Natur und der klaren Schnittstelle lässt er sich flexibel in Automatisierungs- und Steuerungssysteme integrieren, in denen ein Signal auf mehrere Pfade verteilt werden muss. Seine implementationsbedingte Transparenz gewährleistet eine verlustfreie Weitergabe ohne zusätzliche Latenz.

AL_SPLIT_7_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AL_SPLIT_7`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Bild des Funktionsblocks nicht verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AL_SPLIT_7_UNGATED** dient als generischer Verteiler (Splitter) für einen unidirektionalen Adapter vom Typ `AL`. Er nimmt an seinem Socket `IN` einen einzigen Adapter entgegen und leitet die ankommenden Daten und Ereignisse unverändert an sieben identische Adapter-Ausgänge (`OUT1` ... `OUT7`) weiter. Der Baustein ist als generischer Typ (`GEN_AL_SPLIT`) implementiert und ermöglicht so eine flexible Verwendung in Steuerungs- und Automatisierungsanwendungen.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine expliziten Ereignis-Eingänge. Die gesamte Ereignissteuerung erfolgt über den angeschlossenen Adapter `IN`.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge vorhanden. Ereignisse werden über die sieben Adapter-Ausgänge (`OUT1` ... `OUT7`) weitergegeben.

Daten-Eingänge

Der Baustein hat keine eigenständigen Daten-Eingänge. Die Datenübergabe geschieht ausschließlich über den Adapter `IN`.

Daten-Ausgänge

Entsprechend gibt es keine direkten Daten-Ausgänge. Die Daten werden über die sieben Adapter-Ausgänge repliziert.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	`IN`	`adapter::types::unidirectional::AL`	Adapter-Eingang, der die zu verteilende AL-Verbindung bereitstellt.
Plug	`OUT1` ... `OUT7`	`adapter::types::unidirectional::AL`	Sieben Adapter-Ausgänge, die die identische Kopie des Eingangsadapter bereitstellen.

Funktionsweise

Der **AL_SPLIT_7_UNGATED** ist ein reiner Verteilerbaustein. Wenn ein Adapter vom Typ `AL` an den Socket `IN` angeschlossen wird, werden alle über diesen Adapter eingehenden Daten und Ereignisse unverändert an jeden der sieben Plugs (`OUT1` ... `OUT7`) weitergeleitet. Es findet keine logische oder zeitliche

Bearbeitung statt – die Verteilung erfolgt eins-zu-sieben ohne Verzögerung oder Pufferung. Aufgrund der generischen Implementierung kann der Baustein mit verschiedenen AL-Adapterinstanzen verwendet werden, die denselben Datentyp und dasselbe Ereignisschema aufweisen.

Technische Besonderheiten

- **Generische Ausführung:** Der Baustein ist als generischer Typ (`GEN_AL_SPLIT`) deklariert. Dadurch kann er in unterschiedlichen Projekten ohne Anpassung des Quellcodes verwendet werden, solange der zugrundeliegende Adaptertyp `AL` dem erwarteten Interface entspricht.
- **Unidirektionalität:** Der Baustein arbeitet ausschließlich unidirektional – Daten und Ereignisse fließen vom Socket zu den Plugs. Eine Rückkopplung oder bidirektionale Kommunikation ist nicht vorgesehen.
- **Keine Zustandslogik:** Der Baustein besitzt keine eigene Zustandsmaschine; er ist transparenter Natur.

Zustandsübersicht

Der Baustein hat keine internen Zustände. Er ist jederzeit bereit, eingehende Adapter-Verbindungen zu verteilen. Ein aktiver Zustand ergibt sich allein aus der Anwesenheit eines gültigen Adapters am `IN`-Socket.

Anwendungsszenarien

- **Steuerungsverteilung:** Ein übergeordneter Steuerungsadapter (z. B. zur Ansteuerung einer Achse oder eines Sensors) soll identische Signale an mehrere Parallelkomponenten senden.
- **Testumgebungen:** In Simulations- oder Testsetups können mit einem Adapter mehrere gleichartige Teilsysteme versorgt werden.
- **Signalbündelung:** Der Baustein kann als Verteiler in einer Hierarchie von Adaptern verwendet werden, wenn ein Signal auf mehrere nachfolgende Funktionsbausteine aufgeteilt werden muss.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem **AL_SPLIT_2** oder **AL_SPLIT_4** bietet **AL_SPLIT_7_UNGATED** eine höhere Anzahl von Ausgängen (sieben statt zwei oder vier). Während kleinere Splitter bei geringen Anforderungen ausreichen, ermöglicht dieser Baustein eine direkte Verteilung an sieben Empfänger, ohne mehrere Splitter kaskadieren zu müssen. Gegenüber einem **AL_MERGE** (der mehrere Eingänge zusammenführt) ist die Funktion genau entgegengesetzt.

- `AL_SPLIT_7`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

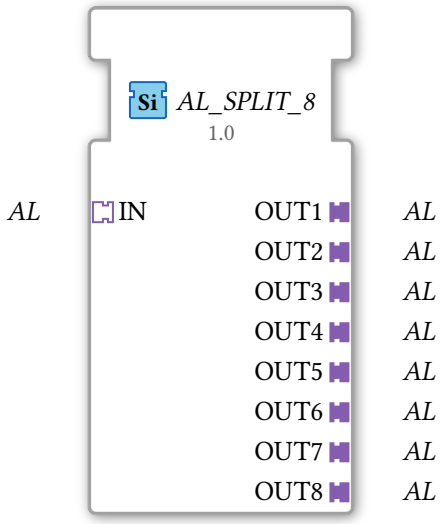
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AL_SPLIT_7_UNGATED** ist ein einfacher, aber nützlicher generischer Baustein zur Vervielfältigung von unidirektionalen AL-Adapterverbindungen. Dank seiner generischen Natur und der klaren Schnittstelle lässt er sich flexibel in Automatisierungs- und Steuerungssysteme integrieren, in denen ein Signal auf mehrere Pfade verteilt werden muss. Seine implementationsbedingte Transparenz gewährleistet eine verlustfreie Weitergabe ohne zusätzliche Latenz.

AL_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AL_SPLIT_8` verteilt ein eingehendes unidirektionales AL-Signal (Socket `IN`) auf acht identische Ausgänge (Plugs `OUT1` bis `OUT8`). Er wird als generischer Baustein deklariert und dient der einfachen Signalvervielfachung in 4diac-Anwendungen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AL`	`IN`	Socket (Eingang)	Eingehendes AL-Signal
`adapter::types::unidirectional::AL`	`OUT1` … `OUT8`	Plugs (Ausgang)	Acht ausgehende AL-Signale

Funktionsweise

Der Baustein besitzt keine eigene Logik oder Zustandsmaschine. Ein an `IN` anliegendes AL-Adapter-Signal wird unverändert und gleichzeitig an alle acht Ausgangs-Adapter weitergeleitet. Die Vervielfachung erfolgt rein datenflussgesteuert, ohne Verzögerung oder Pufferung.

Technische Besonderheiten

- **Generischer Typ**: Der FB ist als generischer Baustein (`GenericClassName = 'GEN_AL_SPLIT'`) deklariert, kann aber aufgrund der festen Adapter-Definition nur mit dem Typ `AL` verwendet werden.
- **Keine Ereignissteuerung**: Es existiert weder ein ECC noch eine Ereignis-Schnittstelle; die Weiterleitung erfolgt passiv über den Datenfluss.
- **Unidirektionalität**: Der Adapter ist nur in eine Richtung ausgelegt (Eingang → Ausgänge).

Zustandsübersicht

Der Baustein besitzt keine internen Zustände. Er ist vollständig zustandslos und führt keine Verarbeitung durch.

Anwendungsszenarien

- **Alarmverteilung**: Ein zentrales Alarmsignal wird an mehrere Subsysteme (z. B. Anzeige, Protokollierung, Steuerung) weitergegeben.
- **Redundante Signalisierung**: Das gleiche Signal kann parallel an mehrere Empfänger gesendet werden, um Ausfallsicherheit zu erhöhen.
- **Ansteuerung mehrerer Aktoren**: Ein Sensor- oder Steuersignal wird auf mehrere unabhängige Aktoren aufgeteilt.

Vergleich mit ähnlichen Bausteinen

- **Standard-SPLIT-Bausteine** (z. B. `SPLIT` für einfache Datentypen) verteilen einzelne Werte, während `AL_SPLIT_8` speziell für den unidirektionalen Adapter `AL` ausgelegt ist.
- **Adapter-Merger** (wie ein hypothetischer `AL_MERGE`) führen mehrere Signale zu einem zusammen; der `AL_SPLIT_8` realisiert die umgekehrte 1:n-Funktionalität.
- `AL_SPLIT_8_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

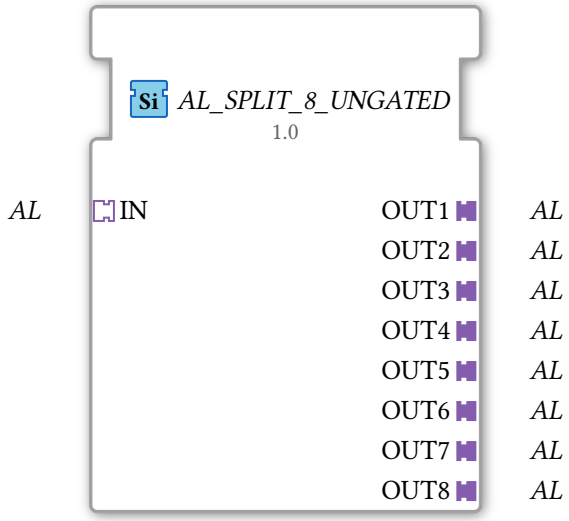
Der `AL_SPLIT_8` ist ein schlichter, aber wichtiger Baustein zur Vervielfachung von AL-Adapter-Signalen. Durch seine passive, zustandslose Arbeitsweise fügt er sich nahtlos in datenflussorientierte 4diac-Anwendungen ein und erleichtert die strukturierte Verteilung von Alarm- oder Steuersignalen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AL_SPLIT_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AL_SPLIT_8`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AL_SPLIT_8_UNGATED` verteilt ein eingehendes unidirektionales AL-Signal (Socket `IN`) auf acht identische Ausgänge (Plugs `OUT1` bis `OUT8`). Er wird als generischer Baustein deklariert und dient der einfachen Signalvervielfachung in 4diac-Anwendungen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AL`	`IN`	Socket (Eingang)	Eingehendes AL-Signal
`adapter::types::unidirectional::AL`	`OUT1` … `OUT8`	Plugs (Ausgang)	Acht ausgehende AL-Signale

Funktionsweise

Der Baustein besitzt keine eigene Logik oder Zustandsmaschine. Ein an `IN` anliegendes AL-Adapter-Signal wird unverändert und gleichzeitig an alle acht Ausgangs-Adapter weitergeleitet. Die Vervielfachung erfolgt rein datenflussgesteuert, ohne Verzögerung oder Pufferung.

Technische Besonderheiten

- **Generischer Typ**: Der FB ist als generischer Baustein (`GenericClassName = 'GEN_AL_SPLIT'`) deklariert, kann aber aufgrund der festen Adapter-Definition nur mit dem Typ `AL` verwendet werden.
- **Keine Ereignissteuerung**: Es existiert weder ein ECC noch eine Ereignis-Schnittstelle; die Weiterleitung erfolgt passiv über den Datenfluss.
- **Unidirektionalität**: Der Adapter ist nur in eine Richtung ausgelegt (Eingang → Ausgänge).

Zustandsübersicht

Der Baustein besitzt keine internen Zustände. Er ist vollständig zustandslos und führt keine Verarbeitung durch.

Anwendungsszenarien

- **Alarmverteilung**: Ein zentrales Alarmsignal wird an mehrere Subsysteme (z. B. Anzeige, Protokollierung, Steuerung) weitergegeben.
- **Redundante Signalisierung**: Das gleiche Signal kann parallel an mehrere Empfänger gesendet werden, um Ausfallsicherheit zu erhöhen.
- **Ansteuerung mehrerer Aktoren**: Ein Sensor- oder Steuersignal wird auf mehrere unabhängige Aktoren aufgeteilt.

Vergleich mit ähnlichen Bausteinen

- **Standard-SPLIT-Bausteine** (z. B. `SPLIT` für einfache Datentypen) verteilen einzelne Werte, während `AL_SPLIT_8_UNGATED` speziell für den unidirektionalen Adapter `AL` ausgelegt ist.
- **Adapter-Merger** (wie ein hypothetischer `AL_MERGE`) führen mehrere Signale zu einem zusammen; der `AL_SPLIT_8_UNGATED` realisiert die umgekehrte 1:n-Funktionalität.
- **`AL_SPLIT_8`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

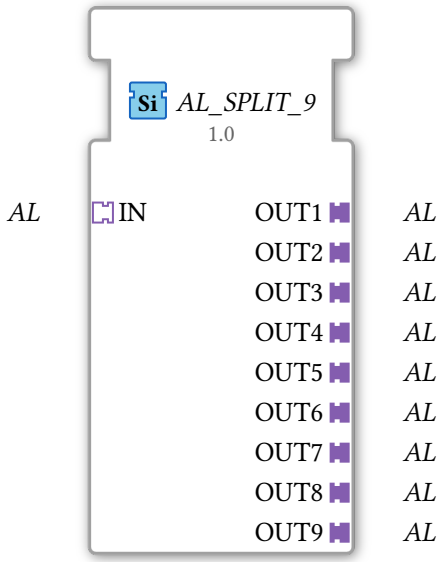
Fazit

Der `AL_SPLIT_8_UNGATED` ist ein schlichter, aber wichtiger Baustein zur Vervielfachung von AL-Adapter-Signalen. Durch seine passive, zustandslose Arbeitsweise fügt er sich nahtlos in datenflussorientierte 4diac-Anwendungen ein und erleichtert die strukturierte Verteilung von Alarm- oder Steuersignalen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AL_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AL_SPLIT_9** ist ein generischer Baustein, der einen eingehenden Adapter vom Typ `AL` auf neun gleichartige ausgehende Adapter aufteilt. Er wird verwendet, um ein Signal oder einen Datenfluss, der über einen einzigen Adapter ankommt, an mehrere nachfolgende Komponenten zu verteilen. Der Baustein ist als generischer Typ (`GEN_AL_SPLIT`) definiert und muss vor der Verwendung an den konkreten Adaptertyp gebunden werden.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine Ereignis-Eingänge. Die Datenweitergabe erfolgt ausschließlich über Adapterverbindungen.

Ereignis-Ausgänge

Es sind keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine separaten Dateneingänge – die eigentlichen Daten werden über den eingehenden Adapter übertragen.

Daten-Ausgänge

Keine separaten Datenausgänge – die Ausgangsdaten werden über die neun ausgehenden Adapter bereitgestellt.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	`IN`	`adapter::types::unidirectional::AL`	Ein eingehender Adapter, der das zu verteilende Signal oder die Daten liefert.
Plug (Ausgang)	`OUT1` … `OUT9`	`adapter::types::unidirectional::AL`	Neun ausgehende Adapter, die jeweils eine Kopie des Eingangssignals ausgeben.

Funktionsweise

Der Baustein agiert als passiver Verteiler. Sobald der eingehende Adapter `IN` mit einem Signal oder Datenpaket belegt wird, leitet der Baustein dieses an alle neun Ausgangsadapter (`OUT1` … `OUT9`) weiter. Die Weiterleitung erfolgt ohne Pufferung oder Verzögerung – es handelt sich um eine reine 1:9-Aufteilung.

Da der Baustein generisch ist, muss ihm vor der Nutzung ein konkreter `AL`-Adaptertyp zugewiesen werden. Dies geschieht in der Entwicklungsumgebung durch die Angabe des spezifischen Adaptertyps, der die eigentlichen Datenfelder und Ereignisse definiert.

Technische Besonderheiten

- **Generischer Baustein:** Der FB ist als `GEN_AL_SPLIT` deklariert und erfordert eine Bindung an einen konkreten `AL`-Adaptertyp (z. B. `SimpleData_AL`). Die Bindung wird durch das Attribut `eclipse4diac::core::GenericClassName` gesteuert.
- **Typensicherheit:** Alle neun Ausgänge verwenden denselben Adaptertyp wie der Eingang. Dadurch wird sichergestellt, dass die Datenstruktur identisch bleibt.
- **Zustandslos:** Der Baustein besitzt keine internen Zustände oder Speicher – die Verteilung erfolgt instantan bei jedem neuen Datendurchlauf.
- **Keine Ereignissteuerung:** Da keine Ereigniseingänge definiert sind, wird die Verteilung allein durch die Datenänderung am Eingangsadapter ausgelöst.

Zustandsübersicht

Der Baustein verfügt über keinen internen Zustandsautomaten. Er arbeitet zustandslos und rein datenflussgetrieben.

Anwendungsszenarien

- **Signalverteilung:** Ein Sensorwert (z. B. Temperatur, Druck) soll gleichzeitig an mehrere Auswerteeinheiten oder Steuerungen gesendet werden.
- **Datenreplikation:** Ein Steuerbefehl soll parallel an mehrere Aktoren verteilt werden.
- **Modulare Architekturen:** Aufteilung eines Datenstroms in einer Pipeline, um verschiedene Verarbeitungszweige zu speisen.

Vergleich mit ähnlichen Bausteinen

- **AL_SPLIT_2, AL_SPLIT_4:** Bausteine mit analoger Funktionalität, die das Eingangssignal auf 2 bzw. 4 Ausgänge verteilen. `AL_SPLIT_9` deckt eine größere Anzahl von Ausgängen ab, was insbesondere in komplexen Verteilstrukturen nützlich ist.
- **Einfache Kopplung (z. B. direkte Verbindung):** Ohne Split-Baustein müsste der Absender mehrere Adapterverbindungen selbst bereitstellen. Der Split-Baustein kapselt diese Logik und vereinfacht die Gesamtarchitektur.
- `AL_SPLIT_9_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

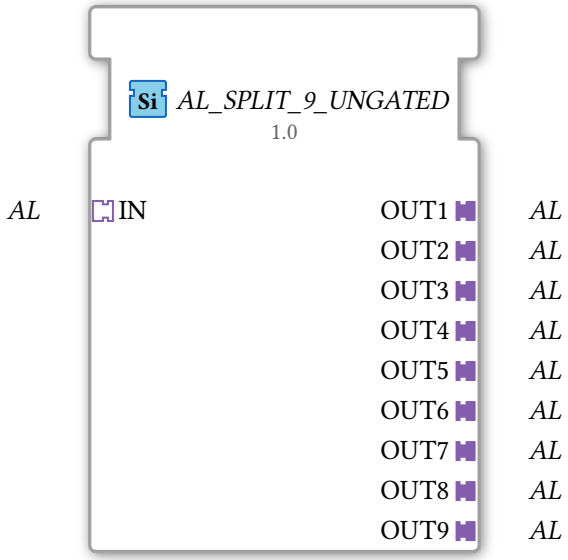
Der `AL_SPLIT_9` ist ein einfacher, aber leistungsfähiger generischer Verteilerbaustein für Adapterschnittstellen vom Typ `AL`. Er ermöglicht eine saubere, wartbare Aufteilung eines Datenflusses auf neun unabhängige Pfade. Dank seiner generischen Natur und der Zustandslosigkeit eignet er sich hervorragend für modulare und skalierbare Automatisierungslösungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AL_SPLIT_9_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AL_SPLIT_9`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AL_SPLIT_9_UNGATED** ist ein generischer Baustein, der einen eingehenden Adapter vom Typ `AL` auf neun gleichartige ausgehende Adapter aufteilt. Er wird verwendet, um ein Signal oder einen Datenfluss, der über einen einzigen Adapter ankommt, an mehrere nachfolgende Komponenten zu verteilen. Der Baustein ist als generischer Typ (`GEN_AL_SPLIT`) definiert und muss vor der Verwendung an den konkreten Adaptertyp gebunden werden.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine Ereignis-Eingänge. Die Datenweitergabe erfolgt ausschließlich über Adapterverbindungen.

Ereignis-Ausgänge

Es sind keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine separaten Dateneingänge – die eigentlichen Daten werden über den eingehenden Adapter übertragen.

Daten-Ausgänge

Keine separaten Datenausgänge – die Ausgangsdaten werden über die neun ausgehenden Adapter bereitgestellt.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	`IN`	`adapter::types::unidirectional::AL`	Ein eingehender Adapter, der das zu verteilende Signal oder die Daten liefert.
Plug (Ausgang)	`OUT1` … `OUT9`	`adapter::types::unidirectional::AL`	Neun ausgehende Adapter, die jeweils eine Kopie des Eingangssignals ausgeben.

Funktionsweise

Der Baustein agiert als passiver Verteiler. Sobald der eingehende Adapter `IN` mit einem Signal oder Datenpaket belegt wird, leitet der Baustein dieses an alle neun Ausgangsadapter (`OUT1` … `OUT9`) weiter. Die Weiterleitung erfolgt ohne Pufferung oder Verzögerung – es handelt sich um eine reine 1:9-Aufteilung.

Da der Baustein generisch ist, muss ihm vor der Nutzung ein konkreter `AL`-Adaptertyp zugewiesen werden. Dies geschieht in der Entwicklungsumgebung durch die Angabe des spezifischen Adaptertyps, der die eigentlichen Datenfelder und Ereignisse definiert.

Technische Besonderheiten

- **Generischer Baustein:** Der FB ist als `GEN_AL_SPLIT` deklariert und erfordert eine Bindung an einen konkreten `AL`-Adaptertyp (z. B. `SimpleData_AL`). Die Bindung wird durch das Attribut `eclipse4diac::core::GenericClassName` gesteuert.
- **Typensicherheit:** Alle neun Ausgänge verwenden denselben Adaptertyp wie der Eingang. Dadurch wird sichergestellt, dass die Datenstruktur identisch bleibt.
- **Zustandslos:** Der Baustein besitzt keine internen Zustände oder Speicher – die Verteilung erfolgt instantan bei jedem neuen Datendurchlauf.
- **Keine Ereignissteuerung:** Da keine Ereigniseingänge definiert sind, wird die Verteilung allein durch die Datenänderung am Eingangsadapter ausgelöst.

Zustandsübersicht

Der Baustein verfügt über keinen internen Zustandsautomaten. Er arbeitet zustandslos und rein datenflussgetrieben.

Anwendungsszenarien

- **Signalverteilung:** Ein Sensorwert (z. B. Temperatur, Druck) soll gleichzeitig an mehrere Auswerteeinheiten oder Steuerungen gesendet werden.
- **Datenreplikation:** Ein Steuerbefehl soll parallel an mehrere Aktoren verteilt werden.
- **Modulare Architekturen:** Aufteilung eines Datenstroms in einer Pipeline, um verschiedene Verarbeitungszweige zu speisen.

Vergleich mit ähnlichen Bausteinen

- **AL_SPLIT_2, AL_SPLIT_4:** Bausteine mit analoger Funktionalität, die das Eingangssignal auf 2 bzw. 4 Ausgänge verteilen. `AL_SPLIT_9_UNGATED` deckt eine größere Anzahl von Ausgängen ab, was insbesondere in komplexen Verteilstrukturen nützlich ist.
- **Einfache Kopplung (z. B. direkte Verbindung):** Ohne Split-Baustein müsste der Absender mehrere Adapterverbindungen selbst bereitstellen. Der Split-Baustein kapselt diese Logik und vereinfacht die Gesamtarchitektur.
- `AL_SPLIT_9`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

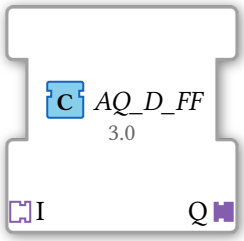
Der `AL_SPLIT_9_UNGATED` ist ein einfacher, aber leistungsfähiger generischer Verteilerbaustein für Adapterschnittstellen vom Typ `AL`. Er ermöglicht eine saubere, wartbare Aufteilung eines Datenflusses auf neun unabhängige Pfade. Dank seiner generischen Natur und der Zustandslosigkeit eignet er sich hervorragend für modulare und skalierbare Automatisierungslösungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

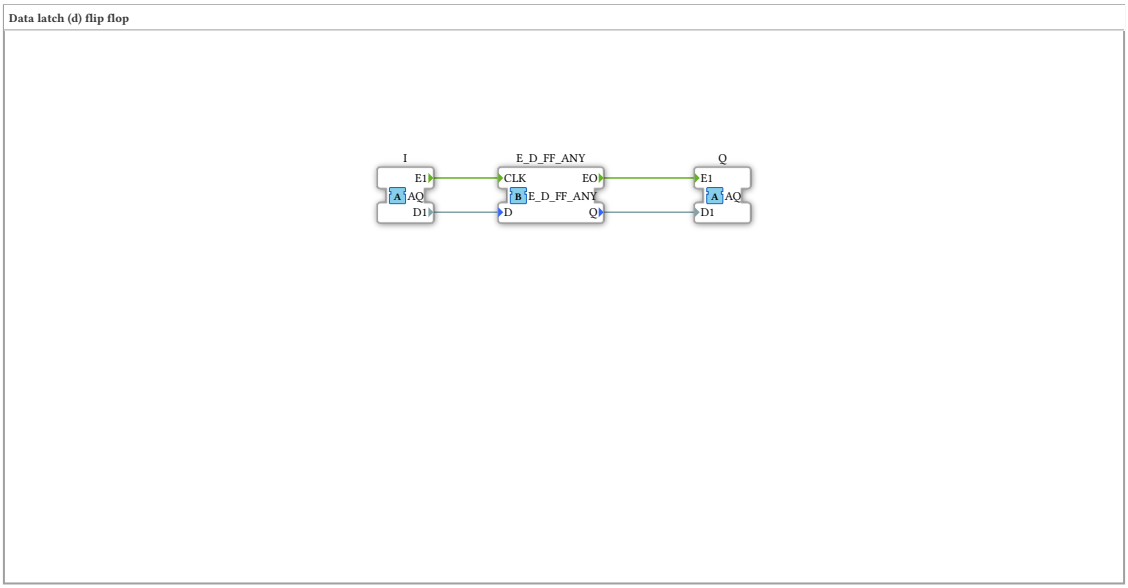
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

QUARTER

AQ_D_FF



Value to latch – *AQ* *AQ* – Latched value



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AQ_D_FF` realisiert ein Datenspeicherglied (D-Latch) auf Basis des Adapterkonzepts der IEC 61499. Er übernimmt einen eingehenden Datenwert bei einem eingehenden Ereignis und stellt diesen bis zum nächsten Ereignis am Ausgang bereit. Der Baustein kapselt die Latch-Funktionalität vollständig in einer Adapter-Schnittstelle und ermöglicht so eine modulare Integration in systemweit genutzte Daten- und Ereignispfade.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung erfolgt ausschließlich über den Adapter `I`.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Ereignisausgabe erfolgt ausschließlich über den Adapter `Q`.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden. Der zu latchende Datenwert wird über den Adapter `I` bereitgestellt.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden. Der gelatchte Datenwert wird über den Adapter `Q` ausgegeben.

Adapter

Adapter	Richtung	Typ	Beschreibung
`I`	Socket (Eingang)	`adapter::types::unidirectional::AQ`	Liefert den zu latchenden Datenwert sowie das auslösende Ereignis.
`Q`	Plug (Ausgang)	`adapter::types::unidirectional::AQ`	Gibt den gelatchten Datenwert sowie ein Bestätigungsereignis aus.

Der verwendete Adaptertyp `adapter::types::unidirectional::AQ` stellt eine unidirektionale Verbindung bereit: Vom Socket zum Plug werden jeweils ein Ereignis- und ein Datenkanal übertragen.

Funktionsweise

Der `AQ_D_FF` arbeitet nach dem Prinzip eines flankengesteuerten D-Latches. Intern wird der Funktionsblock `iec61499::events::E_D_FF_ANY` verwendet:

- Sobald am Adapter `I` ein Ereignis ankommt, wird der aktuell anliegende Datenwert `D1` (aus dem Adapter `I`) übernommen.
- Der übernommene Wert wird am Adapter `Q` als `D1` ausgegeben, gleichzeitig wird das Ereignis `E1` am Ausgangsadapter ausgelöst.
- Zwischen zwei Ereignissen bleibt der Ausgangswert konstant. Bei jedem neuen Ereignis am Eingang wird der Ausgang auf den dann gültigen Eingangswert aktualisiert.

Technische Besonderheiten

- **Reine Adapter-Schnittstelle**: Der Baustein besitzt keine direkten Ereignis- oder Daten-Ein-/Ausgänge. Die gesamte Kommunikation erfolgt über die Adapter `I` und `Q`.
- **Unidirektionale Datenübertragung**: Die Adapter sind vom Typ `adapter::types::unidirectional::AQ`, was bedeutet, dass Daten und Ereignisse nur vom Eingang zum Ausgang fließen. Eine Rückmeldung oder bidirektionale Kommunikation ist nicht vorgesehen.
- **Interne Realisierung**: Die Latch-Funktion wird durch den universellen FB `E_D_FF_ANY` aus der IEC 61499-Ereignisbibliothek implementiert, was eine robuste und standardkonforme Verarbeitung gewährleistet.
- **Keine Initialisierung**: Der Anfangszustand des gelatchten Werts ist undefiniert, bis das erste Ereignis am Eingang eintrifft.

Zustandsübersicht

Der `AQ_D_FF` besitzt keinen expliziten Zustandsautomaten. Sein Verhalten lässt sich jedoch durch zwei logische Zustände beschreiben:

Zustand	Beschreibung
Warten auf Ereignis	Der Ausgang `Q` hält den zuletzt gelatchten Wert. Es liegt kein Ereignis an `I` an.
Übernahme/Ausgabe	Bei Eintreffen eines Ereignisses an `I` wird der neue Wert von `I` übernommen, an `Q` ausgegeben und ein Ereignis auf `E1` von `Q` generiert. Danach kehrt der Baustein in den Wartezustand zurück.

Anwendungsszenarien

- **Zwischenspeicherung von Sensorwerten**: Ein Messwert wird bei einem Trigger-Ereignis (z. B. Takt oder Abfrage) übernommen und steht dann für die weitere Verarbeitung bereit.
- **Synchronisation asynchroner Datenströme**: Wenn Daten asynchron ankommen, kann der `AQ_D_FF` den letzten gültigen Wert festhalten, bis ein nachgelagerter Prozess ihn abholt.
- **Ereignisgesteuerte Datenweitergabe**: Der Baustein kann als einfacher „Data-Holder“ in einem Adapter-basierten Kommunikationspfad verwendet werden, um Daten nur bei bestimmten Ereignissen weiterzuleiten.

Vergleich mit ähnlichen Bausteinen

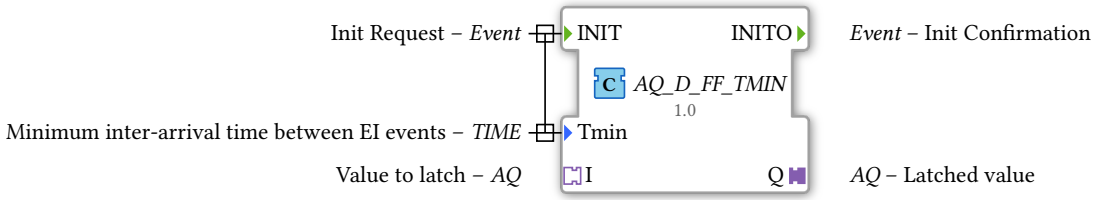
Baustein	Eigenschaften

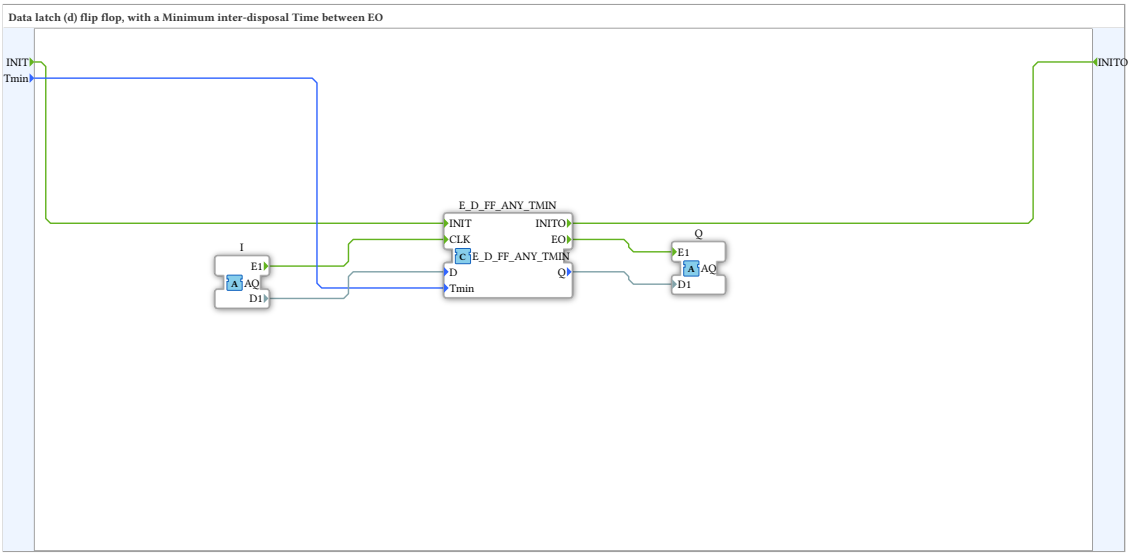
`E_D_FF` (direkte IOs)	Gleiche Latch-Funktionalität, aber mit separaten Ereignis- und Dateneingängen/-ausgängen. Keine Adapter.
`E_SR_FF`	Set/Reset-Flipflop mit zwei getrennten Eingangsereignissen. Hier kein Datenwert, sondern binäre Zustände.
`AQ_D_FF`	Ermöglicht die Einbettung der Latch-Funktion in bestehende Adapter-Schnittstellen, ohne zusätzliche Verdrahtung auf FB-Ebene.

Fazit

Der Funktionsblock `AQ_D_FF` bietet eine kompakte, adapterbasierte Realisierung eines D-Latches. Er integriert sich nahtlos in systemweit genutzte Adapterpfade und reduziert so den Verdrahtungsaufwand auf der FB-Ebene. Seine Funktionsweise ist einfach und zuverlässig, insbesondere für Anwendungen, bei denen ein Datenwert bei einem Ereignis übernommen und bis zum nächsten Ereignis gehalten werden muss.

AQ_D_FF_TMIN





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AQ_D_FF_TMIN** realisiert ein datenverriegelndes D-Flipflop mit einer minimalen Zwischenzeit (Minimum inter-disposal Time) zwischen zwei aufeinanderfolgenden Ausgangsereignissen (EO). Er dient als Erweiterung eines einfachen D-Flipflops um eine zeitliche Entprellung oder Mindestabstandssteuerung und wird über Adapter mit dem Signalgeber und dem Signalempfänger verbunden.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Mit Variable	Kommentar
INIT	EInit	Tmin	Initialisierungsanfrage, setzt die minimale Zwischenzeit.

Ereignis-Ausgänge

Ereignis	Typ	Kommentar
INITO	EInit	Initialisierungsbestätigung.

Daten-Eingänge

Variable	Typ	Kommentar
Tmin	TIME	Minimale Wartezeit zwischen zwei Ausgangsereignissen (EO).

Daten-Ausgänge

Keine eigenen Datenausgänge, das latched Signal wird über den Adapter `Q` ausgegeben.

Adapter

Typ	Bezeichnung	Richtung	Kommentar
`adapter::types::unidirectional:`	`AQ` I	Socket (Eingang)	Wert zum latchen, enthält je ein Daten- (D1) und Ereignissignal (E1).
`adapter::types::unidirectional:`	`AQ` Q	Plug (Ausgang)	Gelatchter Wert, enthält je ein Daten- (D1) und Ereignissignal (E1).

Funktionsweise

Der Baustein kapselt einen internen FB vom Typ `iec61499::events::E_D_FF_ANY_TMIN`. Dieser arbeitet wie ein flankengesteuertes D-Flipflop, bei dem das Ereignis `I.E1` als Takt und `I.D1` als Dateneingang dient. Das Ausgangsereignis `EO` wird nur dann am Adapter `Q.E1` ausgegeben, wenn seit dem letzten ausgegebenen Ereignis mindestens die in `Tmin` festgelegte Zeit vergangen ist. Der Datenwert am Ausgang `Q.D1` entspricht dem zum Zeitpunkt der steigenden Flanke an `I.E1` anliegenden Wert `I.D1`.

Die Initialisierung (`INIT`) übergibt den Parameter `Tmin` an den internen FB, sodass die zeitliche Bedingung festgelegt wird.

Technische Besonderheiten

- **Minimale Zwischenzeit (Tmin):** Verhindert zu schnelle aufeinanderfolgende Ausgangsereignisse, z. B. zum Entprellen oder zur Begrenzung der Aktualisierungsrate.
- **Adapterbasierte Kommunikation:** Der FB verwendet unidirektionale Adapter (`AQ`), die sowohl Daten als auch Ereignisse bündeln und eine lose Kopplung zwischen Signalquelle und -senke ermöglichen.
- **IEC 61499-konform:** Entspricht dem Standard 61499-1 Annex A und nutzt die Ereignis-/Datenflüsse der Norm.

Zustandsübersicht

Der FB besitzt keinen expliziten öffentlichen Zustandsautomaten. Sein Verhalten wird durch den internen FB `E_D_FF_ANY_TMIN` bestimmt, der als Zustand den letzten gelatchten Wert und eine Wartezeitlogik aufweist. Vereinfacht dargestellt:

Zustand	Bedeutung
Initialisiert	Nach INIT bereit, Tmin ist gesetzt
Bereit	Wartet auf Takt (I.E1)
Gelatcht	Datenwert wurde übernommen, Ausgangsereignis wird nach Ablauf von Tmin ausgegeben
Wartend	Tmin läuft – Ausgangsereignis wird blockiert

Eine detaillierte Zustandsdarstellung ist im Quellcode des internen FBs ersichtlich.

Anwendungsszenarien

- **Signalglättung/Entprellung:** Vermeiden von Mehrfachauslösungen bei mechanischen Tastern oder schnellen Signalwechseln.
- **Datenratenbegrenzung:** Begrenzen der Ausgabehäufigkeit von Messwerten in Überwachungssystemen.
- **Synchronisation asynchroner Signale:** Weiterleitung eines Datenwerts mit definiertem zeitlichem Mindestabstand.
- **Steuerung langsamer Aktoren:** Sicherstellen, dass Aktoren nicht schneller angesteuert werden als ihre mechanische Reaktionszeit zulässt.

Vergleich mit ähnlichen Bausteinen

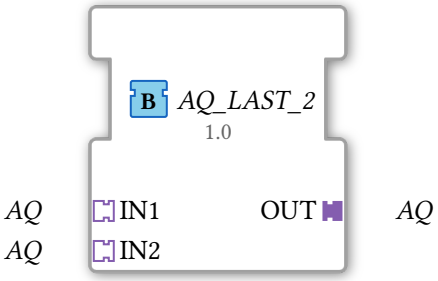
Baustein	Besonderheit
`AQ_D_FF` / `E_D_FF`	Einfaches D-Flipflop ohne zeitliche Einschränkung des Ausgangs.
`AQ_D_FF_TMIN` (dieser FB)	Erweitert um minimale Zwischenzeit Tmin zwischen Ausgangsereignissen.
`AQ_D_FF_TMAX` (hypothetisch)	Begrenzung der maximalen Zeit zwischen zwei Ausgangsereignissen.

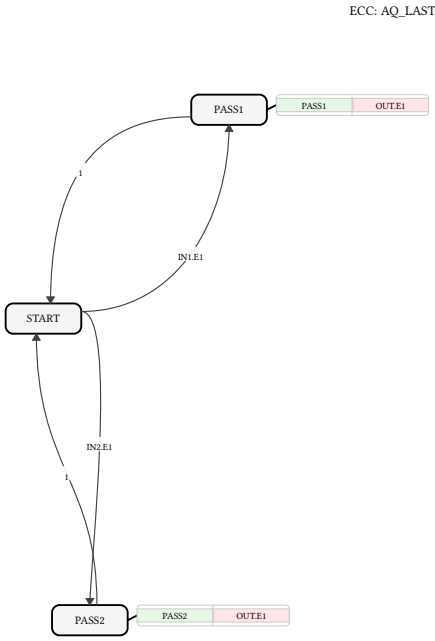
Gegenüber einem Standard-D-Flipflop bietet `AQ_D_FF_TMIN` zusätzlichen Schutz vor zu rascher Folge von Ausgangsereignissen, was in der Praxis häufig erforderlich ist.

Fazit

Der Funktionsblock `AQ_D_FF_TMIN` kombiniert die Funktion eines D-Flipflops mit einer einstellbaren minimalen Verzögerung am Ausgang. Durch die Adapter-Schnittstellen und die IEC-61499-Konformität lässt er sich einfach in industrielle Steuerungsanwendungen integrieren, bei denen sowohl Datenübernahme als auch zeitliche Entkopplung der Ereignisse erforderlich sind. Er bietet eine robuste Lösung für Signalverarbeitung und Aktorsteuerung mit definierten zeitlichen Randbedingungen.

AQ_LAST_2





Einleitung

Der AQ_LAST_2 Funktionsblock führt zwei gleichartige QUARTER-Adapter-Signale (**IN1**, **IN2**) auf einen gemeinsamen Ausgang (**OUT**) zusammen, nach dem Prinzip **Last Writer Wins**: welcher der beiden Sockets zuletzt ein Ereignis geschrieben hat, gewinnt, und sein Datenwert wird sofort und unverändert an OUT durchgereicht. Es handelt sich ausdrücklich nicht um ein naives Zusammenführen (z. B. Mittelwert oder ODER-Verknüpfung), sondern um eine reine Arbitrierung nach zeitlicher Reihenfolge.

Schnittstellenstruktur

Adapter

Eingangsadapter (Sockets):

- **IN1**: QUARTER-Adapter (unidirectional) - erste Signalquelle
- **IN2**: QUARTER-Adapter (unidirectional) - zweite Signalquelle

Ausgangsadapter (Plug):

- **OUT**: QUARTER-Adapter (unidirectional) - zusammengeführtes Signal

Funktionsweise

Der Baustein ist als einfacher Basic-FB mit einer 3-Zustands-ECC umgesetzt (START, PASS1, PASS2). Trifft an IN1 ein Ereignis ein, wechselt die ECC nach PASS1: der aktuelle Datenwert von IN1 wird nach OUT kopiert und das Adapter-Ereignis an OUT gesendet. Trifft an IN2 ein Ereignis ein, geschieht dasselbe symmetrisch über PASS2. Nach jeder Durchreichung kehrt die ECC sofort wieder nach START zurück und ist bereit für das nächste Ereignis von IN1 oder IN2.

Technische Besonderheiten

- Last-Writer-Wins-Semantik statt naiver Datenverknüpfung
- Einfacher Basic-FB (ECC mit 3 Zuständen), kein zusätzlicher Zwischenspeicher nötig
- Keine Signalverzögerung: der Wert wird im selben Zyklus durchgereicht, in dem das auslösende Ereignis eintrifft
- Beide Eingänge sind gleichberechtigt - es gibt keine feste Priorität außer der zeitlichen Reihenfolge

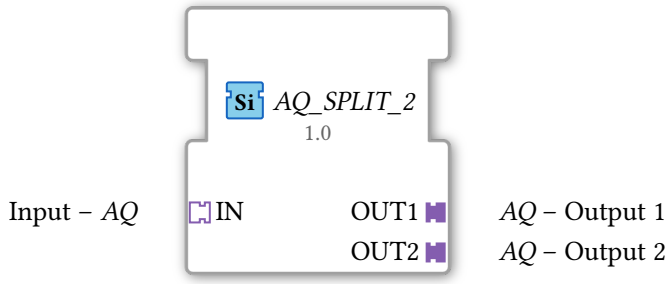
Anwendungsszenarien

- Zusammenführen zweier gleichwertiger QUARTER-Signalquellen zu einem einzigen nachgeschalteten Adapterpfad (z. B. zwei redundante Sensoren, oder ein manueller und ein automatischer Sollwertgeber)
- Arbitrierung zwischen zwei Bedienstellen, die denselben QUARTER-Wert schreiben dürfen
- Vereinfachung von Netzwerken, die sonst ein `F_SEL`/`E_RS`-Konstrukt inline verdrahten müssten

⚖️ **Vergleich mit ähnlichen Bausteinen**

AQ_LAST_2 ist die Umkehrung von `AQ_SPLIT_2`: AQ_SPLIT_2 verteilt 1 Eingang auf 2 Ausgänge, AQ_LAST_2 führt 2 Eingänge auf 1 Ausgang zusammen. Für Mischbetrieb zwischen einem Daten- und einem reinen Ereignisadapter (z. B. AX + AE) ist stattdessen `AX_AE_MERGE` gedacht, das nur das Ereignis mischt, ohne den Datenwert zu verändern.

AQ_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AQ_SPLIT_2` dient als Fan-Out-Baustein. Er verteilt ein eingehendes AQ (Analog Quantity) Signal unidirektional auf zwei identische Ausgänge. Er realisiert damit eine einfache 1:2-Signalvervielfachung für analoge Größen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AQ`	`IN`	Socket (Eingang)	Eingangsadapter für das zu teilende AQ-Signal.
`adapter::types::unidirectional::AQ`	`OUT1`	Plug (Ausgang)	Erster Ausgang – liefert unverändertes Eingangssignal.
`adapter::types::unidirectional::AQ`	`OUT2`	Plug (Ausgang)	Zweiter Ausgang – liefert ebenfalls unverändertes Eingangssignal.

Funktionsweise

Der Baustein empfängt ein analoges Signal über den Adapter-Socket `IN`. Dieses Signal wird intern ohne Verzögerung oder Umwandlung auf beide Adapter-Plugs `OUT1` und `OUT2` durchgeschaltet. Es findet keinerlei Signalverarbeitung statt – der FB agiert als reiner Splitter.

Technische Besonderheiten

- **Generische Implementierung**: Der FB ist als generischer Typ `GEN_AQ_SPLIT` deklariert und kann für beliebige AQ-Adapter verwendet werden, solange diese dem unidirektionalen AQ-Interface entsprechen.
- **Keine Ereignissteuerung**: Der Baustein arbeitet rein datengetrieben ohne Ereignisse – eine Zustandsänderung am Eingang wirkt sich sofort auf beide Ausgänge aus.
- **Unidirektional**: Die Adapter sind für eine Richtung ausgelegt (Eingang ⟶ Ausgänge).

Zustandsübersicht

Der FB besitzt keine internen Zustände. Er ist zustandslos (kombinatorisch) und gibt zu jedem Zeitpunkt das aktuell anliegende Eingangssignal an beiden Ausgängen aus.

Anwendungsszenarien

- **Signalverteilung** in Steuerungssystemen: Ein von einer Sensorik bereitgestelltes analoges Signal soll parallel an zwei unterschiedliche Verbraucher (z.B. zwei Regler oder Anzeigeeinheiten) weitergegeben werden.
- **Test- und Simulationsaufbauten**: Ein Referenzsignal wird ohne Rückwirkung auf mehrere Testpunkte aufgeteilt.
- **Redundante Weiterleitung**: Das Signal wird an zwei unabhängige Pfade gesendet, um Ausfallsicherheit zu erhöhen.

Vergleich mit ähnlichen Bausteinen

- **`AQ_SPLIT_N`** (hypothetisch): Ein vergleichbarer Baustein mit `N` Ausgängen – `AQ_SPLIT_2` ist eine spezialisierte 1:2-Version.
- **Ereignisgesteuerte Split-Bausteine**: Diese benötigen oft Trigger-Ereignisse und führen dann eine Signalweitergabe durch. `AQ_SPLIT_2` arbeitet dagegen rein signalgetrieben ohne Ereignisse.
- **Bus-basierte Splitter**: Diese verteilen mehrere Signale über einen gemeinsamen Bus (z.B. über Multicast). `AQ_SPLIT_2` arbeitet punktuell und einzeln über Adapter.
- **`AQ_SPLIT_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der `AQ_SPLIT_2` ist ein einfacher, aber nützlicher Baustein zur Verteilung eines analogen Signals auf zwei identische Ausgänge. Dank seiner generischen Natur lässt er sich in verschiedenen 4diac-Projekten ohne Anpassungen einsetzen. Die zustandslose Funktionsweise minimiert die Komplexität und macht ihn ideal für Echtzeitanwendungen, bei denen eine sofortige Signalweitergabe erforderlich ist.

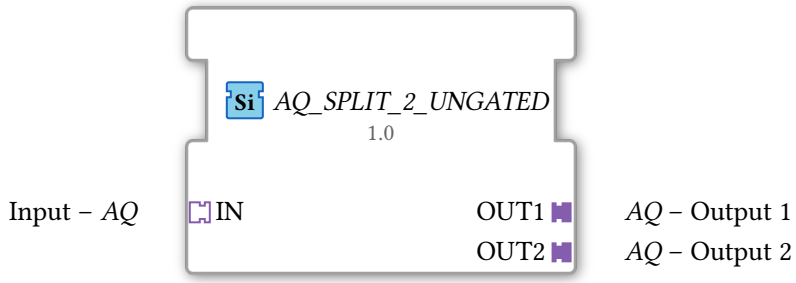
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AQ_SPLIT_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AQ_SPLIT_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz

unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AQ_SPLIT_2_UNGATED` dient als Fan-Out-Baustein. Er verteilt ein eingehendes AQ (Analog Quantity) Signal unidirektional auf zwei identische Ausgänge. Er realisiert damit eine einfache 1:2-Signalvervielfachung für analoge Größen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AQ`	`IN`	Socket (Eingang)	Eingangsadapter für das zu teilende AQ-Signal.
`adapter::types::unidirectional::AQ`	`OUT1`	Plug (Ausgang)	Erster Ausgang – liefert unverändertes Eingangssignal.
`adapter::types::unidirectional::AQ`	`OUT2`	Plug (Ausgang)	Zweiter Ausgang – liefert ebenfalls unverändertes Eingangssignal.

Funktionsweise

Der Baustein empfängt ein analoges Signal über den Adapter-Socket `IN`. Dieses Signal wird intern ohne Verzögerung oder Umwandlung auf beide Adapter-Plugs `OUT1` und `OUT2` durchgeschaltet. Es findet keinerlei Signalverarbeitung statt – der FB agiert als reiner Splitter.

Technische Besonderheiten

- **Generische Implementierung**: Der FB ist als generischer Typ `GEN_AQ_SPLIT` deklariert und kann für beliebige AQ-Adapter verwendet werden, solange diese dem unidirektionalen AQ-Interface entsprechen.
- **Keine Ereignissteuerung**: Der Baustein arbeitet rein datengetrieben ohne Ereignisse – eine Zustandsänderung am Eingang wirkt sich sofort auf beide Ausgänge aus.
- **Unidirektional**: Die Adapter sind für eine Richtung ausgelegt (Eingang ⟶ Ausgänge).

Zustandsübersicht

Der FB besitzt keine internen Zustände. Er ist zustandslos (kombinatorisch) und gibt zu jedem Zeitpunkt das aktuell anliegende Eingangssignal an beiden Ausgängen aus.

Anwendungsszenarien

- **Signalverteilung** in Steuerungssystemen: Ein von einer Sensorik bereitgestelltes analoges Signal soll parallel an zwei unterschiedliche Verbraucher (z.B. zwei Regler oder Anzeigeeinheiten) weitergegeben werden.
- **Test- und Simulationsaufbauten**: Ein Referenzsignal wird ohne Rückwirkung auf mehrere Testpunkte aufgeteilt.
- **Redundante Weiterleitung**: Das Signal wird an zwei unabhängige Pfade gesendet, um Ausfallsicherheit zu erhöhen.

Vergleich mit ähnlichen Bausteinen

- `AQ_SPLIT_N` (hypothetisch): Ein vergleichbarer Baustein mit `N` Ausgängen – `AQ_SPLIT_2_UNGATED` ist eine spezialisierte 1:2-Version.
- **Ereignisgesteuerte Split-Bausteine**: Diese benötigen oft Trigger-Ereignisse und führen dann eine Signalweitergabe durch. `AQ_SPLIT_2_UNGATED` arbeitet dagegen rein signalgetrieben ohne Ereignisse.
- **Bus-basierte Splitter**: Diese verteilen mehrere Signale über einen gemeinsamen Bus (z.B. über Multicast). `AQ_SPLIT_2_UNGATED` arbeitet punktuell und einzeln über Adapter.
- `AQ_SPLIT_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

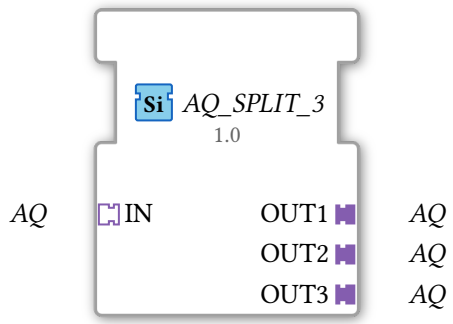
Fazit

Der `AQ_SPLIT_2_UNGATED` ist ein einfacher, aber nützlicher Baustein zur Verteilung eines analogen Signals auf zwei identische Ausgänge. Dank seiner generischen Natur lässt er sich in verschiedenen 4diac-Projekten ohne Anpassungen einsetzen. Die zustandslose Funktionsweise minimiert die Komplexität und macht ihn ideal für Echtzeitanwendungen, bei denen eine sofortige Signalweitergabe erforderlich ist.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AQ_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AQ_SPLIT_3` realisiert eine 1-zu-3-Aufteilung (Fan-Out) eines unidirektionalen AQ-Adapter-Signals. Ein eingehender AQ-Adapter wird auf drei identische Ausgangs-Adapter kopiert und parallel bereitgestellt.

Schnittstellenstruktur

Ereignis-Eingänge
Keine.

Ereignis-Ausgänge
Keine.

Daten-Eingänge
Keine.

Daten-Ausgänge
Keine.

Adapter
Socket (Eingang)

Bezeichnung	Typ	Beschreibung
`IN`	`adapter::types::unidirectional::AQ`	Eingangs-Adapter, dessen Signal auf drei Ausgänge verteilt wird

Plug (Ausgänge)

Bezeichnung	Typ	Beschreibung
`OUT1`	`adapter::types::unidirectional::AQ`	Erster Ausgang (Kopie des Eingangssignals)
`OUT2`	`adapter::types::unidirectional::AQ`	Zweiter Ausgang (Kopie des Eingangssignals)
`OUT3`	`adapter::types::unidirectional::AQ`	Dritter Ausgang (Kopie des Eingangssignals)

Funktionsweise

Der Baustein leitet das am Socket `IN` anliegende AQ-Signal ohne Verzögerung oder Datenmanipulation an alle drei Plugs (`OUT1`, `OUT2`, `OUT3`) weiter. Es handelt sich um eine reine Signalverteilung (Fan-Out) – das eingehende Adapter-Interface wird auf drei identische Ausgangs-Interfaces gespiegelt. Da keine Ereignisse oder Daten außerhalb des Adapters verarbeitet werden, erfolgt die Weiterleitung passiv und transparent.

Technische Besonderheiten

- **Generischer Typ**: Der Baustein verwendet das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `'GEN_AQ_SPLIT'`. Dadurch kann er in verschiedenen Kontexten als generischer AQ-Splitter eingesetzt werden.
- **Unidirektional**: Die Adapter sind als unidirektionale AQ-Schnittstellen definiert – die Daten fließen nur vom Eingang zu den Ausgängen.
- **Keine Zustandslogik**: Der Baustein besitzt keine Ereignis-E/A, keinen internen Zustand und keine Daten-E/A. Er fungiert als reine Verbindungslogik.

Zustandsübersicht

Der Baustein ist zustandslos. Er führt keine zeitabhängigen oder sequenziellen Abläufe aus. Die Ausgangssignale entsprechen stets unmittelbar dem aktuellen Eingangssignal.

Anwendungsszenarien

- **Signalverteilung in der Automatisierung**: Wenn ein AQ-Signal (z. B. ein Qualitätswert, ein Messwert- oder Status-Adapter) gleichzeitig an mehrere Verbraucher weitergegeben werden muss.
- **Multicasting von Adaptern**: In Architekturen, in denen ein Dienst sein Ergebnis über einen AQ-Adapter an mehrere nachfolgende Bausteine sendet.
- **Generische Kopplung**: Einsatz als generischer Splitter in Bibliotheken oder Frameworks, die auf dem `GEN_AQ_SPLIT`-Typ basieren.

Vergleich mit ähnlichen Bausteinen

- **AQ_SPLIT_2**: Verteilt nur auf zwei Ausgänge – `AQ_SPLIT_3` bietet einen zusätzlichen dritten Ausgang.
- **Manuelle Dreifachverdrahtung**: Statt eines Bausteins könnte man das Eingangssignal manuell auf drei Sockets verbinden. Der Splitter erhöht jedoch die Übersichtlichkeit und Wartbarkeit des Netzwerks.
- **Generische Splitter anderer Adaptertypen**: Ähnliche Bausteine existieren z. B. für Daten- oder Ereignis-Signale (z. B. `DATA_SPLIT_3`). `AQ_SPLIT_3` ist speziell auf den AQ-Adapter-Typ zugeschnitten.
- `AQ_SPLIT_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

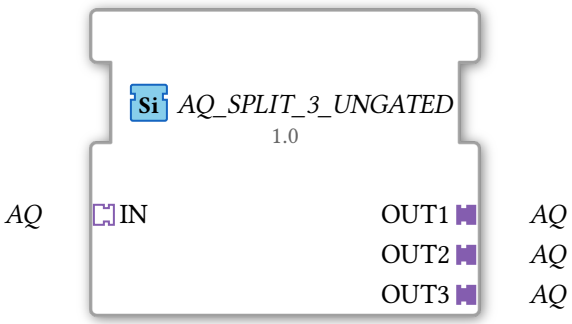
`AQ_SPLIT_3` ist ein einfacher, funktionaler Fan-Out-Baustein für unidirektionale AQ-Adapter. Dank seiner generischen Auslegung und der klaren 1-zu-3-Verteilung eignet er sich ideal für alle Anwendungen, die ein AQ-Signal mehrfach bereitstellen müssen. Die Abwesenheit von Ereignis- und Daten-E/A macht ihn leichtgewichtig und in beliebigen Ablaufkontexten einsetzbar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AQ_SPLIT_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AQ_SPLIT_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AQ_SPLIT_3_UNGATED` realisiert eine 1-zu-3-Aufteilung (Fan-Out) eines unidirektionalen AQ-Adapter-Signals. Ein eingehender AQ-Adapter wird auf drei identische Ausgangs-Adapter kopiert und parallel bereitgestellt.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge
Keine.

Daten-Eingänge
Keine.

Daten-Ausgänge
Keine.

Adapter
Socket (Eingang)

Bezeichnung	Typ	Beschreibung
IN	adapter::types::unidirectional::AQ	Eingangs-Adapter, dessen Signal auf drei Ausgänge verteilt wird

Plug (Ausgänge)

Bezeichnung	Typ	Beschreibung
OUT1	adapter::types::unidirectional::AQ	Erster Ausgang (Kopie des Eingangssignals)
OUT2	adapter::types::unidirectional::AQ	Zweiter Ausgang (Kopie des Eingangssignals)
OUT3	adapter::types::unidirectional::AQ	Dritter Ausgang (Kopie des Eingangssignals)

Funktionsweise
Der Baustein leitet das am Socket `IN` anliegende AQ-Signal ohne Verzögerung oder Datenmanipulation an alle drei Plugs (`OUT1`, `OUT2`, `OUT3`) weiter. Es handelt sich um eine reine Signalverteilung (Fan-Out) – das eingehende Adapter-Interface wird auf drei identische Ausgangs-Interfaces gespiegelt. Da keine Ereignisse oder Daten außerhalb des Adapters verarbeitet werden, erfolgt die Weiterleitung passiv und transparent.

Technische Besonderheiten
- **Generischer Typ**: Der Baustein verwendet das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `'GEN_AQ_SPLIT'`. Dadurch kann er in verschiedenen Kontexten als generischer AQ-Splitter eingesetzt werden.
- **Unidirektional**: Die Adapter sind als unidirektionale AQ-Schnittstellen definiert – die Daten fließen nur vom Eingang zu den Ausgängen.
- **Keine Zustandslogik**: Der Baustein besitzt keine Ereignis-E/A, keinen internen Zustand und keine Daten-E/A. Er fungiert als reine Verbindungslogik.

Zustandsübersicht
Der Baustein ist zustandslos. Er führt keine zeitabhängigen oder sequenziellen Abläufe aus. Die Ausgangssignale entsprechen stets unmittelbar dem aktuellen Eingangssignal.

Anwendungsszenarien
- **Signalverteilung in der Automatisierung**: Wenn ein AQ-Signal (z. B. ein Qualitätswert, ein Messwert- oder Status-Adapter) gleichzeitig an mehrere Verbraucher weitergegeben werden muss.
- **Multicasting von Adaptern**: In Architekturen, in denen ein Dienst sein Ergebnis über einen AQ-Adapter an mehrere nachfolgende Bausteine sendet.
- **Generische Kopplung**: Einsatz als generischer Splitter in Bibliotheken oder Frameworks, die auf dem `GEN_AQ_SPLIT`-Typ basieren.

Vergleich mit ähnlichen Bausteinen
- **AQ_SPLIT_2**: Verteilt nur auf zwei Ausgänge – `AQ_SPLIT_3_UNGATED` bietet einen zusätzlichen dritten Ausgang.

- **Manuelle Dreifachverdrahtung**: Statt eines Bausteins könnte man das Eingangssignal manuell auf drei Sockets verbinden. Der Splitter erhöht jedoch die Übersichtlichkeit und Wartbarkeit des Netzwerks.
- **Generische Splitter anderer Adaptertypen**: Ähnliche Bausteine existieren z. B. für Daten- oder Ereignis-Signale (z. B. `DATA_SPLIT_3`). `AQ_SPLIT_3_UNGATED` ist speziell auf den AQ-Adapter-Typ zugeschnitten.
- `AQ_SPLIT_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

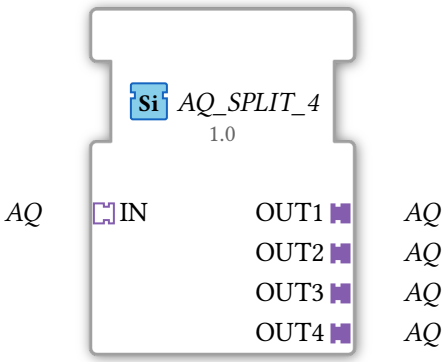
Fazit

`AQ_SPLIT_3_UNGATED` ist ein einfacher, funktionaler Fan-Out-Baustein für unidirektionale AQ-Adapter. Dank seiner generischen Auslegung und der klaren 1-zu-3-Verteilung eignet er sich ideal für alle Anwendungen, die ein AQ-Signal mehrfach bereitstellen müssen. Die Abwesenheit von Ereignis- und Daten-E/A macht ihn leichtgewichtig und in beliebigen Ablaufkontexten einsetzbar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AQ_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AQ_SPLIT_4` dient als Fan-Out-Komponente zur Verteilung eines eingehenden AQ-Adapters auf vier identische Ausgangs-Adapter. Er wird eingesetzt, um ein analoges Signal oder eine Steuergröße (AQ – Analog Quantity) von einem Sender an mehrere unabhängige Empfänger weiterzuleiten, ohne die Signalintegrität zu beeinträchtigen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Typ	Richtung	Name	Beschreibung
`adapter::types::unidirectional::AQ`	Socket (Eingang)	`IN`	Eingehender AQ-Adapter, der auf die vier Ausgänge verteilt wird.
`adapter::types::unidirectional::AQ`	Plug (Ausgang)	`OUT1`	Erster ausgehender AQ-Adapter.
`adapter::types::unidirectional::AQ`	Plug (Ausgang)	`OUT2`	Zweiter ausgehender AQ-Adapter.
`adapter::types::unidirectional::AQ`	Plug (Ausgang)	`OUT3`	Dritter ausgehender AQ-Adapter.
`adapter::types::unidirectional::AQ`	Plug (Ausgang)	`OUT4`	Vierter ausgehender AQ-Adapter.

Funktionsweise

Der `AQ_SPLIT_4` leitet alle über den Socket `IN` ankommenden Daten und Ereignisse des AQ-Adapters unverändert an die vier Plug-Adapter `OUT1` bis `OUT4` weiter. Es findet keine Verarbeitung, Filterung oder Verzögerung der Signale statt. Die Verteilung erfolgt rein auf Verbindungsebene, sodass alle Ausgänge stets denselben Zustand wie der Eingang haben.

Technische Besonderheiten

- **Generische Typisierung**: Der FB verwendet das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `'GEN_AQ_SPLIT'`. Dadurch kann er in 4diac-IDE als generischer Baustein angesprochen werden, was die Wiederverwendbarkeit in verschiedenen Anwendungskontexten erhöht.
- **Keine Zustandslogik**: Der Baustein besitzt keine interne Zustandsmaschine (ECC) und keine Ein-/Ausgangs-Ereignisse. Er agiert als reine „Verdrahtungshilfe“ für Adapterverbindungen.
- **Unidirektionaler Adapter**: Alle verwendeten Adapter sind vom Typ `unidirectional::AQ`, d.h. die Daten fließen nur in eine Richtung (vom Socket zu den Plugs).

Zustandsübersicht

Der Funktionsblock verfügt über keine Zustandsmaschine. Er ist passiv und führt keinerlei zeit- oder ereignisgesteuerte Operationen aus. Die Signalweitergabe erfolgt kontinuierlich und ohne Zwischenspeicherung.

Anwendungsszenarien

- **Verteilen eines Messwerts**: Ein analoger Sensor (z.B. Temperatur, Druck, Durchfluss) wird über einen AQ-Adapter angeschlossen und muss an mehrere Regel- oder Überwachungseinheiten gesendet werden.
- **Mehrfachnutzung eines Stellsignals**: Ein von einer Steuerung ausgegebenes analoges Stellsignal wird parallel an mehrere Aktoren (z.B. Ventile, Frequenzumrichter) weitergegeben.
- **Signalüberwachung**: Das ursprüngliche Signal wird unverändert an die Endgeräte gesendet, während ein zusätzlicher Ausgang zur Diagnose oder Protokollierung verwendet wird.

Vergleich mit ähnlichen Bausteinen

- **AQ_SPLIT_2**: Verteilt ein AQ-Signal auf zwei statt vier Ausgänge – platzsparender bei geringerem Bedarf.
- **IQ_SPLIT_4**: Analoger Baustein für digitale Adapter (z.B. `adapter::types::unidirectional::IQ`), ansonsten identische Funktionsweise.
- **Manuelle Parallelschaltung**: Theoretisch könnte man einen AQ-Socket manuell mit mehreren Plugs verbinden, aber der `AQ_SPLIT_4` bietet eine übersichtliche und wiederverwendbare Kapselung.
- **`AQ_SPLIT_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

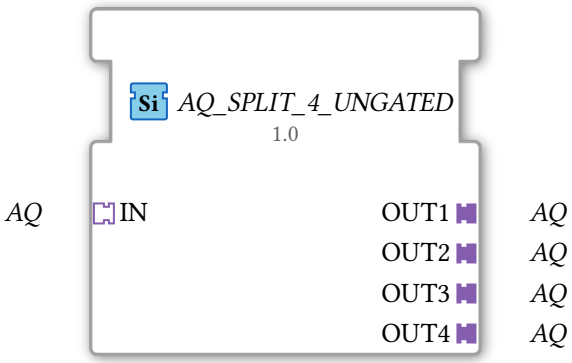
Der `AQ_SPLIT_4` ist ein einfacher, aber nützlicher Funktionsblock zur Signalverteilung in der industriellen Automatisierung mit 4diac. Durch seine generische Auslegung und das Fehlen interner Logik ist er leicht verständlich, robust und flexibel einsetzbar. Er trägt zur Strukturierung und Wiederverwendbarkeit von Funktionsblocknetzwerken bei.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 Gesamtwiderstand in Reihen- & Parallelschaltung auf ms-muc-docs.de

AQ_SPLIT_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AQ_SPLIT_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AQ_SPLIT_4_UNGATED` dient als Fan-Out-Komponente zur Verteilung eines eingehenden AQ-Adapters auf vier identische Ausgangs-Adapter. Er wird eingesetzt, um ein analoges Signal oder eine Steuergröße (AQ – Analog Quantity) von einem Sender an mehrere unabhängige Empfänger weiterzuleiten, ohne die Signalintegrität zu beeinträchtigen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Typ	Richtung	Name	Beschreibung

adapter::types::unidirectional::AQ	Socket (Eingang)	IN	Eingehender AQ-Adapter, der auf die vier Ausgänge verteilt wird.
adapter::types::unidirectional::AQ	Plug (Ausgang)	OUT1	Erster ausgehender AQ-Adapter.
adapter::types::unidirectional::AQ	Plug (Ausgang)	OUT2	Zweiter ausgehender AQ-Adapter.
adapter::types::unidirectional::AQ	Plug (Ausgang)	OUT3	Dritter ausgehender AQ-Adapter.
adapter::types::unidirectional::AQ	Plug (Ausgang)	OUT4	Vierter ausgehender AQ-Adapter.

Funktionsweise

Der AQ_SPLIT_4_UNGATED leitet alle über den Socket IN ankommenden Daten und Ereignisse des AQ-Adapters unverändert an die vier Plug-Adapter OUT1 bis OUT4 weiter. Es findet keine Verarbeitung, Filterung oder Verzögerung der Signale statt. Die Verteilung erfolgt rein auf Verbindungsebene, sodass alle Ausgänge stets denselben Zustand wie der Eingang haben.

Technische Besonderheiten

- **Generische Typisierung**: Der FB verwendet das Attribut eclipse4diac::core::GenericClassName mit dem Wert 'GEN_AQ_SPLIT'. Dadurch kann er in 4diac-IDE als generischer Baustein angesprochen werden, was die Wiederverwendbarkeit in verschiedenen Anwendungskontexten erhöht.
- **Keine Zustandslogik**: Der Baustein besitzt keine interne Zustandsmaschine (ECC) und keine Ein-/Ausgangs-Ereignisse. Er agiert als reine „Verdrahtungshilfe“ für Adapterverbindungen.
- **Unidirektionaler Adapter**: Alle verwendeten Adapter sind vom Typ unidirectional::AQ, d.h. die Daten fließen nur in eine Richtung (vom Socket zu den Plugs).

Zustandsübersicht

Der Funktionsblock verfügt über keine Zustandsmaschine. Er ist passiv und führt keinerlei zeit- oder ereignisgesteuerte Operationen aus. Die Signalweitergabe erfolgt kontinuierlich und ohne Zwischenspeicherung.

Anwendungsszenarien

- **Verteilen eines Messwerts**: Ein analoger Sensor (z.B. Temperatur, Druck, Durchfluss) wird über einen AQ-Adapter angeschlossen und muss an mehrere Regel- oder Überwachungseinheiten gesendet werden.
- **Mehrfachnutzung eines Stellsignals**: Ein von einer Steuerung ausgegebenes analoges Stellsignal wird parallel an mehrere Aktoren (z.B. Ventile, Frequenzumrichter) weitergegeben.
- **Signalüberwachung**: Das ursprüngliche Signal wird unverändert an die Endgeräte gesendet, während ein zusätzlicher Ausgang zur Diagnose oder Protokollierung verwendet wird.

Vergleich mit ähnlichen Bausteinen

- **AQ_SPLIT_2**: Verteilt ein AQ-Signal auf zwei statt vier Ausgänge – platzsparender bei geringerem Bedarf.
- **IQ_SPLIT_4**: Analoger Baustein für digitale Adapter (z.B. adapter::types::unidirectional::IQ), ansonsten identische Funktionsweise.
- **Manuelle Parallelschaltung**: Theoretisch könnte man einen AQ-Socket manuell mit mehreren Plugs verbinden, aber der AQ_SPLIT_4_UNGATED bietet eine übersichtliche und wiederverwendbare Kapselung.
- **AQ_SPLIT_4_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- **AQ_SPLIT_4**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

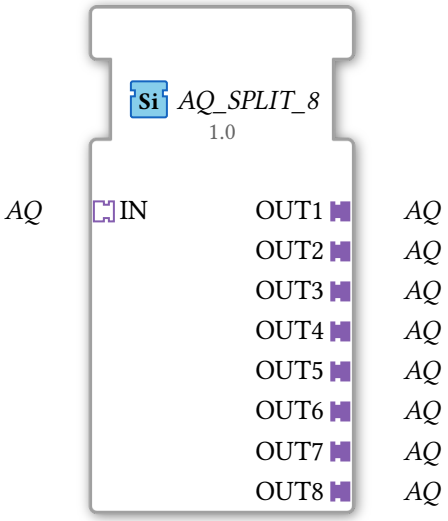
Fazit

Der `AQ_SPLIT_4_UNGATED` ist ein einfacher, aber nützlicher Funktionsblock zur Signalverteilung in der industriellen Automatisierung mit 4diac. Durch seine generische Auslegung und das Fehlen interner Logik ist er leicht verständlich, robust und flexibel einsetzbar. Er trägt zur Strukturierung und Wiederverwendbarkeit von Funktionsblocknetzwerken bei.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 Gesamtwiderstand in Reihen- & Parallelschaltung auf ms-muc-docs.de

AQ_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AQ_SPLIT_8** ist ein Fan-Out-Baustein, der einen einzigen AQ-Adapter-Eingang auf acht identische AQ-Adapter-Ausgänge verteilt. Er dient der Signalvervielfachung in Analogausgangs-Pfaden und ermöglicht so die parallele Ansteuerung mehrerer gleichartiger Aktoren oder Subsysteme. Die Implementierung basiert auf dem generischen Mechanismus `GEN_AQ_SPLIT` und ist für unidirektionale AQ-Schnittstellen ausgelegt.

Schnittstellenstruktur

Ereignis-Eingänge

Keine – der Baustein arbeitet rein adapterbasiert ohne Ereignissteuerung.

Ereignis-Ausgänge

Keine – der Baustein besitzt keine Ereignisausgänge.

Daten-Eingänge

Keine – die gesamte Signalübertragung erfolgt über AQ-Adapter.

Daten-Ausgänge

Keine – siehe Daten-Eingänge.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	`adapter::types::unidirectional::AQ`	Socket (Eingang)	Empfängt das zu verteilende AQ-Signal.

OUT1 – OUT8	`Adapter::types::unidirectional::AP`	Plug (Ausgang)	Identische Ausgänge, die das Signal von `IN` weiterleiten.

Funktionsweise

Der Baustein leitet das am Socket `IN` anliegende AQ-Signal durch interne Verdrahtung an alle acht Plug-Ausgänge `OUT1` bis `OUT8` weiter. Dabei erfolgt keine Signalverarbeitung oder -änderung – es handelt sich um eine reine 1:8-Verteilung (Fan-Out). Die Adapter sind unidirektional, d. h. Daten und Ereignisse fließen nur vom Eingang zu den Ausgängen.

Technische Besonderheiten

- **Keine getaktete Logik**: Der Baustein benötigt keine Ereignis-E/As, da die Weiterleitung direkt über die Adapter-Interface-Ebene erfolgt.
- **Generischer Typ**: Der Baustein ist als generische Instanz (`GEN_AQ_SPLIT`) deklariert – der konkrete Adaptertyp kann je nach Systemumgebung angepasst werden.
- **Erweiterbarkeit**: Die 8 Ausgänge sind fest verdrahtet; eine dynamische Anzahl oder Konfiguration ist nicht vorgesehen.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die Funktionalität ist rein statisch: Sobald der `IN`-Adapter aktiv ist, geben alle Ausgänge das identische Signal aus. Es gibt keine Betriebsarten, Fehlerzustände oder Lifecycle-Events.

Anwendungsszenarien

- **Parallelverkabelung**: Ein analoges Ausgangssignal gleichzeitig an mehrere Stellglieder oder Messstellen verteilen.
- **Test- und Simulationsumgebungen**: Ein Signalgenerator speist über `IN` und über `OUT1–OUT8` werden mehrere simulierte Komponenten versorgt.
- **Redundante Ansteuerung**: Falls mehrere Subsysteme denselben Sollwert erhalten sollen, ohne dass ein Bus oder eine Softwareverzweigung nötig ist.

Vergleich mit ähnlichen Bausteinen

- **AQ_SPLIT_4**: Bietet nur 4 Ausgänge – geeignet bei geringerer Verteilungsanforderung.
- **AQ_SELECT**: Ein Multiplexer, der aus mehreren Eingängen einen auswählt – eher Gegenteil der Split-Funktion.
- **DQ_SPLIT_8**: Analoger Baustein für digitale Signale (DQ-Typ) – strukturell identisch, aber für andere Signalart.
- `AQ_SPLIT_8_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

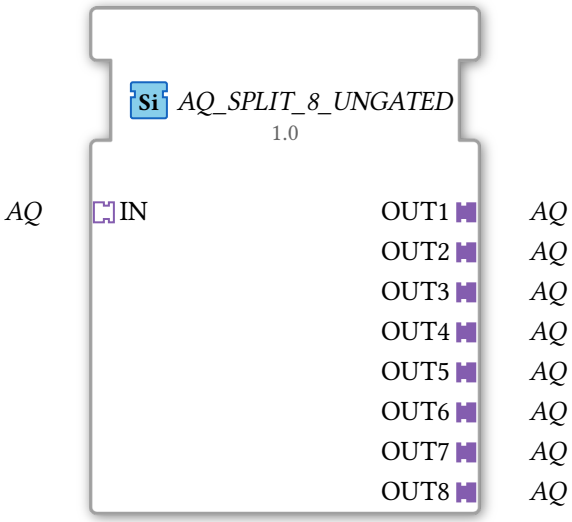
Fazit

Der **AQ_SPLIT_8** ist ein einfacher, zuverlässiger Fan-Out-Baustein für die 1:8-Verteilung von Analogausgangssignalen über Adapter-Schnittstellen. Er verzichtet auf Ereignislogik und Zustandsautomaten und eignet sich daher besonders für klar definierte, statische Signalverteilungen in Automatisierungssystemen. Die generische Basis erleichtert die Wiederverwendung in unterschiedlichen Projekten.

AQ_SPLIT_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AQ_SPLIT_8`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz

unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AQ_SPLIT_8_UNGATED** ist ein Fan-Out-Baustein, der einen einzigen AQ-Adapter-Eingang auf acht identische AQ-Adapter-Ausgänge verteilt. Er dient der Signalvervielfachung in Analogausgangs-Pfaden und ermöglicht so die parallele Ansteuerung mehrerer gleichartiger Aktoren oder Subsysteme. Die Implementierung basiert auf dem generischen Mechanismus `GEN_AQ_SPLIT` und ist für unidirektionale AQ-Schnittstellen ausgelegt.

Schnittstellenstruktur

Ereignis-Eingänge

Keine – der Baustein arbeitet rein adapterbasiert ohne Ereignissteuerung.

Ereignis-Ausgänge

Keine – der Baustein besitzt keine Ereignisausgänge.

Daten-Eingänge

Keine – die gesamte Signalübertragung erfolgt über AQ-Adapter.

Daten-Ausgänge

Keine – siehe Daten-Eingänge.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	`adapter::types::unidirectional::AQ`	Socket (Eingang)	Empfängt das zu verteilende AQ-Signal.
OUT1 – OUT8	`adapter::types::unidirectional::AQ`	Plug (Ausgang)	Identische Ausgänge, die das Signal von `IN` weiterleiten.

Funktionsweise

Der Baustein leitet das am Socket `IN` anliegende AQ-Signal durch interne Verdrahtung an alle acht Plug-Ausgänge `OUT1` bis `OUT8` weiter. Dabei erfolgt keine Signalverarbeitung oder -änderung – es handelt sich

um eine reine 1:8-Verteilung (Fan-Out). Die Adapter sind unidirektional, d. h. Daten und Ereignisse fließen nur vom Eingang zu den Ausgängen.

Technische Besonderheiten

- **Keine getaktete Logik**: Der Baustein benötigt keine Ereignis-E/As, da die Weiterleitung direkt über die Adapter-Interface-Ebene erfolgt.
- **Generischer Typ**: Der Baustein ist als generische Instanz (`GEN_AQ_SPLIT`) deklariert – der konkrete Adaptertyp kann je nach Systemumgebung angepasst werden.
- **Erweiterbarkeit**: Die 8 Ausgänge sind fest verdrahtet; eine dynamische Anzahl oder Konfiguration ist nicht vorgesehen.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die Funktionalität ist rein statisch: Sobald der `IN`-Adapter aktiv ist, geben alle Ausgänge das identische Signal aus. Es gibt keine Betriebsarten, Fehlerzustände oder Lifecycle-Events.

Anwendungsszenarien

- **Parallelverkabelung**: Ein analoges Ausgangssignal gleichzeitig an mehrere Stellglieder oder Messstellen verteilen.
- **Test- und Simulationsumgebungen**: Ein Signalgenerator speist über `IN` und über `OUT1`–`OUT8` werden mehrere simulierte Komponenten versorgt.
- **Redundante Ansteuerung**: Falls mehrere Subsysteme denselben Sollwert erhalten sollen, ohne dass ein Bus oder eine Softwareverzweigung nötig ist.

Vergleich mit ähnlichen Bausteinen

- **AQ_SPLIT_4**: Bietet nur 4 Ausgänge – geeignet bei geringerer Verteilungsanforderung.
- **AQ_SELECT**: Ein Multiplexer, der aus mehreren Eingängen einen auswählt – eher Gegenteil der Split-Funktion.
- **DQ_SPLIT_8**: Analoger Baustein für digitale Signale (DQ-Typ) – strukturell identisch, aber für andere Signalart.
- `AQ_SPLIT_8`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

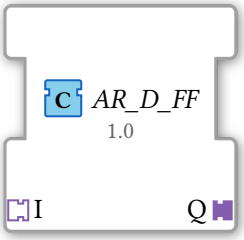
Fazit

Der **AQ_SPLIT_8_UNGATED** ist ein einfacher, zuverlässiger Fan-Out-Baustein für die 1:8-Verteilung von Analogausgangssignalen über Adapter-Schnittstellen. Er verzichtet auf Ereignislogik und Zustandsautomaten und eignet sich daher besonders für klar definierte, statische Signalverteilungen in Automatisierungssystemen. Die generische Basis erleichtert die Wiederverwendung in unterschiedlichen Projekten.

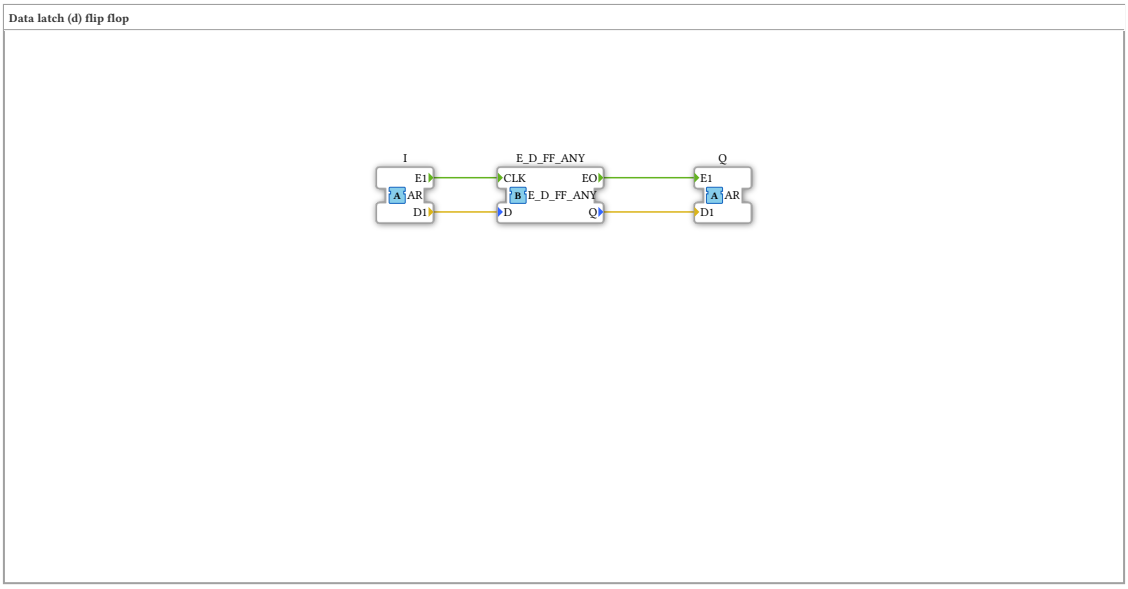
REAL

AR_D_FF

Value to latch – *AR*



AR – Latched value



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **AR_D_FF** realisiert ein daten-gesteuertes Flipflop (D-Latch) gemäß der Spezifikation IEC 61499-1 Annex A. Er dient als Kapselung des Standardbausteins `iec61499::events::E_D_FF_ANY` und stellt dessen Funktionalität über zwei unidirektionale Adapter zur Verfügung. Der Baustein übernimmt bei einer positiven Flanke des Taktsignals einen Datenwert und hält diesen bis zur nächsten Flanke.

Schnittstellenstruktur

Der AR_D_FF besitzt keine direkten Ereignis- oder Datenein-/ausgänge; die gesamte Kommunikation erfolgt über seine zwei Adapter.

Ereignis-Eingänge

- **Keine direkten.**
  Das Taktereignis wird über den Eingangsadapter **I** bereitgestellt (siehe Adapter-Abschnitt).

Ereignis-Ausgänge

- **Keine direkten.**
  Das Bestätigungsereignis wird über den Ausgangsadapter **Q** bereitgestellt (siehe Adapter-Abschnitt).

Daten-Eingänge

- **Keine direkten.**
  Der zu latchnde Datenwert wird über den Eingangsadapter **I** bereitgestellt (siehe Adapter-Abschnitt).

Daten-Ausgänge

- **Keine direkten.**
  Der gelatchte Datenwert wird über den Ausgangsadapter **Q** bereitgestellt (siehe Adapter-Abschnitt).

Adapter

Adapter	Richtung / Typ	Beschreibung	Enthaltene Elemente
Socket	**I** (`adapter::types::unidirectional::AR`)	Wert, der gelatcht werden soll.	**E1** (Ereigniseingang) – Taktsignal (CLK)**D1** (Dateneingang) – Datenwert
Plug	**Q** (`adapter::types::unidirectional::AR`)	Gelatchter Wert.	**E1** (Ereignisausgang) – Bestätigung nach erfolgreichem Latch**D1** (Datenaus-

			gang) – gelatchter Datenwert

Funktionsweise

Der Baustein arbeitet nach dem Prinzip eines flankengesteuerten D-Flipflops:

1. Ein Ereignis am Takteingang **I.E1** aktiviert die Übernahme.
2. Zum Zeitpunkt der positiven Flanke wird der aktuelle Datenwert **I.D1** intern gespeichert.
3. Der gespeicherte Wert wird sofort an **Q.D1** ausgegeben.
4. Gleichzeitig wird ein Ereignis auf **Q.E1** ausgelöst, um die Verarbeitung zu bestätigen.

Der interne Zustand bleibt zwischen zwei Taktflanken unverändert (Speicherverhalten).

Technische Besonderheiten

- **Adapter-basierte Kapselung:** Der AR_D_FF nutzt ausschließlich unidirektionale Adapter (Typ `AR`) für Ein- und Ausgabe. Dies ermöglicht eine modulare, standardisierte Einbindung in größere Adapter-Netzwerke.
- **Verwendung des Standardbausteins `E_D_FF_ANY`:** Die eigentliche Flipflop-Logik wird durch den IEC-61499-Standardbaustein realisiert. Änderungen an der internen Implementierung wirken sich nicht auf die nach außen sichtbare Schnittstelle aus.
- **Kein Reset-Eingang:** Im Gegensatz zu anderen Flipflop-Varianten verfügt dieser Baustein über keine Rücksetzfunktion. Der Zustand kann nur durch einen neuen Datenwert und einen Taktimpuls geändert werden.

Zustandsübersicht

Der Baustein besitzt einen einzigen internen Speicher (den gelatchten Datenwert). Zustandsübergänge erfolgen ausschließlich bei einer positiven Flanke am Takteingang **I.E1**:

Alter Zustand	Eingang D1 (aktuell)	Neuer Zustand	Ausgang D1	Ereignis Q.E1
beliebig	*beliebig*	Wert von D1	Wert von D1	wird ausgelöst

Es gibt keine weiteren internen Zustände oder Verzweigungen.

Anwendungsszenarien

- **Datensynchronisation:** Übernahme eines Datenwortes zu einem definierten Zeitpunkt, z. B. in einem zyklischen Steuerungssystem.
- **Pufferung:** Zwischenspeicherung eines Wertes zwischen zwei Berechnungsschritten.
- **Ereignisgesteuerte Signalweitergabe:** Anwendungen, in denen ein Wert nur dann aktualisiert werden darf, wenn ein bestimmtes Ereignis eintritt.

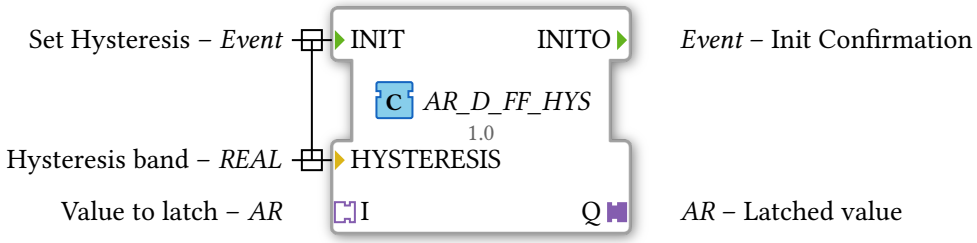
Vergleich mit ähnlichen Bausteinen

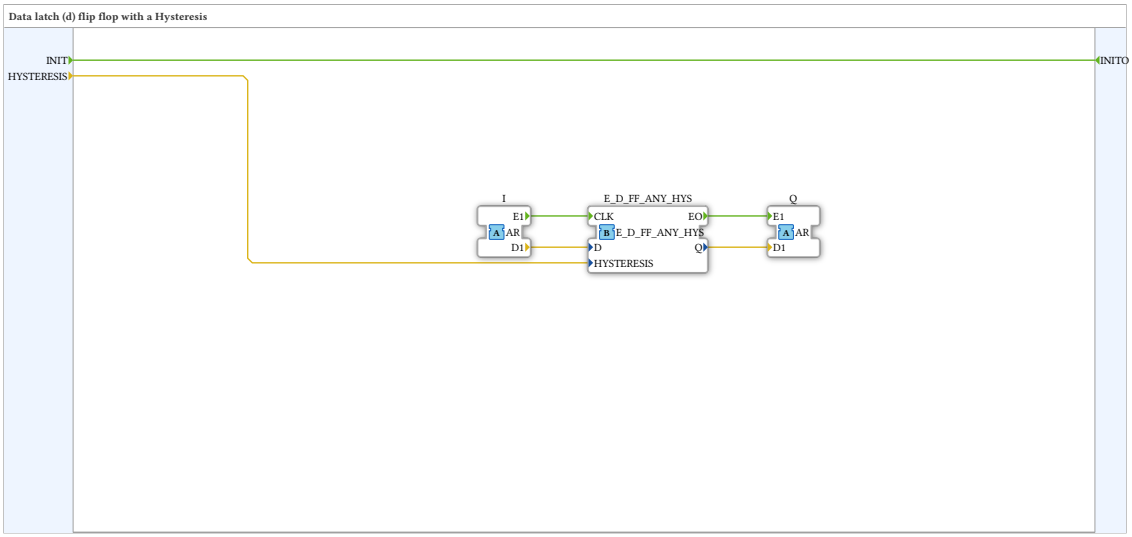
Baustein	Eigenschaft	Unterschied zu AR_D_FF
AR_SR_FF	Set-Reset-Flipflop	Latcht keinen Datenwert, sondern wird über Set/Reset gesetzt/rückgesetzt. Besitzt zwei Ereigniseingänge.
E_D_FF_ANY	Standard-D-Flipflop ohne Adapter	Gleiche Kernlogik, aber direkte Ereignis-/Datenanschlüsse; keine Adapter-Kapselung.
AR_T_FF	Toggle-Flipflop	Schaltet bei jedem Taktimpuls den Ausgang um; kein Dateneingang.

Fazit

Der **AR_D_FF** ist ein einfaches, zuverlässiges D-Flipflop, das durch die Adapter-Schnittstelle besonders für modulare und wiederverwendbare IEC-61499-Anwendungen geeignet ist. Die Kapselung des Standardbausteins erhöht die Robustheit und erleichtert die Integration in größere Systeme. Für Anwendungen, die einen Reset oder eine andere Logik erfordern, stehen spezialisierte Flipflop-Varianten zur Verfügung.

AR_D_FF_HYS





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **AR_D_FF_HYS** realisiert ein Data-Latch (D-)Flipflop mit Hysterese. Er dient zur stabilen Erfassung und Weitergabe eines analogen (realen) Wertes, wobei ein Hystereseband unerwünschtes Flattern am Ausgang unterdrückt. Der Baustein ist für den Einsatz in der industriellen Signalverarbeitung konzipiert.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung
INIT	EInit	Initialisiert den Baustein und setzt das Hystereseband.

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung
INITO	EInit	Bestätigt die erfolgreiche Initialisierung.

Daten-Eingänge

Name	Typ	Beschreibung
HYSTERESIS	REAL	Größe des Hystereseb ands.

Daten-Ausgänge

(keine direkten Datenausgänge; der latente Wert wird über den Adapter Q ausgegeben)

Adapter

Bezeichnung	Typ	Richtung	Beschreibung
I	adapter::types::unidirectional::AR	Socket (Eingang)	Empfängt den zu latchten Wert (Triggerevent auf E1, Daten auf D1).
Q	adapter::types::unidirectional::AR	Plug (Ausgang)	Gibt den gelatchten Wert aus (Ereignis auf E1, Daten auf D1).

Funktionsweise

Der Baustein enthält intern einen Sub-FB `E_D_FF_ANY_HYS` aus der Bibliothek `logiBUS::signalprocessing::hysteresis`.

- Bei jedem steigenden Flanke auf dem Ereigniseingang `I.E1` wird der aktuelle Datenwert von `I.D1` übernommen.
- Der übernommene Wert wird jedoch erst dann auf `Q.D1` weitergegeben, wenn er sich um mehr als die eingestellte Hysterese (`HYSTERESIS`) vom zuletzt ausgegebenen Wert unterscheidet.
- Das Ereignis `I.E1` wird synchron als `Q.E1` weitergereicht, um den Empfänger über einen neuen Wert zu informieren.
- Die Initialisierung (`INIT`) setzt das Hystereseband und bestätigt sofort mit `INITO`. Eine Änderung von `HYSTERESIS` während des Betriebs ist nicht vorgesehen; die Hysterese wird nur beim INIT-Ereignis übernommen.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Sowohl der Eingang als auch der Ausgang verwenden den generischen Adapter `adapter::types::unidirectional::AR`, der es erlaubt, den Baustein flexibel in vorhandene Adapterketten einzubinden.
- **Hysterese-Funktion**: Unterdrückt Rauschen und Oszillationen an Schwellwerten und sorgt für einen sauberen, stabilen Ausgang.
- **EPL 2.0**: Der Baustein ist unter der Eclipse Public License 2.0 lizenziert und kann frei verwendet und modifiziert werden.
- **Interne Implementierung**: Die eigentliche Hystereselogik wird von einem spezialisierten Subbaustein (`E_D_FF_ANY_HYS`) ausgeführt, was Wartbarkeit und Wiederverwendbarkeit verbessert.

Zustandsübersicht

Der Baustein besitzt einen impliziten Speicherzustand für den zuletzt ausgegebenen Wert. Da es sich um ein D-Flipflop mit Hysterese handelt, kann man drei logische Bereiche unterscheiden:

- **Eingang ≥ (Ausgang + Hysterese/2)** → Ausgang wird auf den neuen Eingang gesetzt.
- **Eingang ≤ (Ausgang – Hysterese/2)** → Ausgang wird auf den neuen Eingang gesetzt.
- **Wert innerhalb des Hystereseb ands** → Ausgang bleibt unverändert (Historie).

Explizite Zustandsautomaten sind nicht vorhanden; der Baustein arbeit rein ereignisgesteuert.

Anwendungsszenarien

- **Signalglättung**: Entfernen von hochfrequentem Rauschen aus analogen Messwerten.
- **Schwellwertschalter**: Definierte Ein- und Ausschaltpunkte für binäre Entscheidungen (z. B. Temperaturregelung).
- **Flankenerkennung mit Hysterese**: Zuverlässige Erkennung von steigenden oder fallenden Flanken eines analogen Signals.
- **Pegelanpassung**: Stabiles Nachführen eines Ausgangswerts an einen schwankenden Eingang.

Vergleich mit ähnlichen Bausteinen

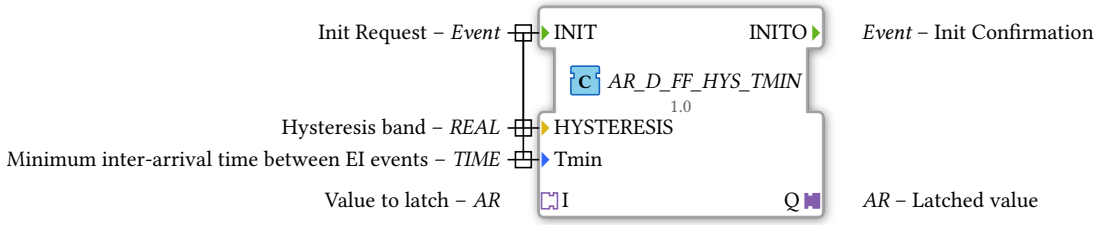
Baustein	Eigenschaft
AR_D_FF (ohne Hysterese)	Übernimmt jeden neuen Wert sofort; empfindlich gegen Rauschen.
AR_D_FF_HYS (dieser FB)	Verwendet ein Hystereseband zur Unterdrückung von Flimmern.

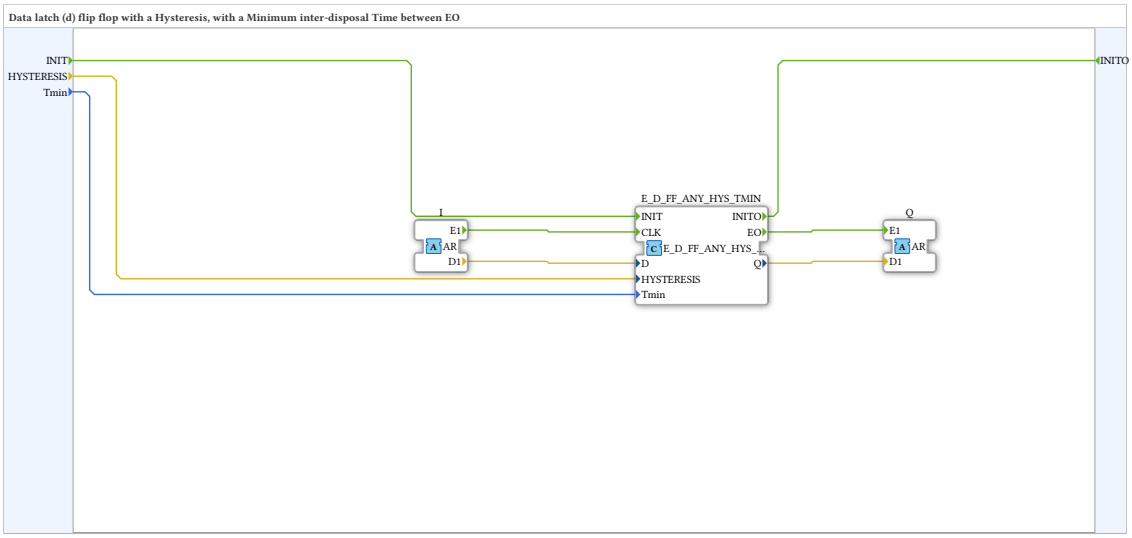
Schmitt-Trigger	Realisiert eine ähnliche Hysterese, aber meist für binäre (boolesche) Signale. Dieser Baustein arbeitet mit analogen (REAL) Werten.

Fazit

Der **AR_D_FF_HYS** ist ein robuster, adapterbasierter Funktionsbaustein für die zuverlässige Weitergabe analoger Werte mit einstellbarer Hysterese. Er eignet sich besonders für Anwendungen, in denen Störungen auf der Signalleitung auftreten oder ein definiertes Schaltverhalten erforderlich ist. Die einfache Initialisierung und die klare Schnittstellenstruktur erleichtern die Integration in automatisierungstechnische Systeme.

AR_D_FF_HYS_TMIN





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AR_D_FF_HYS_TMIN** realisiert ein Data-Latch (D-Flip-Flop) mit einer Hysterese-Funktion sowie einer minimalen Wartezeit zwischen aufeinanderfolgenden Ausgabeereignissen am Ausgangsadapter. Er dient zur stabilen Erfassung und Weiterleitung von analogen Werten über unidirektionale AR-Adapter, wobei er Flankenrauschen unterdrückt und die Ausgabefrequenz begrenzt.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung
INIT	EInit	Initialisierungsanforderung; übergibt die Parameter HYSTERESIS und Tmin an die interne Logik.

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung

INITO	EInit	Bestätigung der erfolgten Initialisierung.

Daten-Eingänge

Variable	Typ	Beschreibung
HYSTERESIS	REAL	Hysterese-Band: Eine Änderung des Eingangswerts muss diesen Betrag überschreiten, damit ein neuer Wert gelatcht wird.
Tmin	TIME	Mindestzeit zwischen zwei Ausgabeereignissen (inter-disposal time). Verhindert zu schnelles Weiterschalten.

Daten-Ausgänge

Der FB besitzt keine eigenständigen Datenausgänge; die Ausgabedaten werden über den Adapter **Q** bereitgestellt.

Adapter

Bezeichnung	Typ	Richtung	Beschreibung
I	adapter::types::unidirectional::AR (Socket)	Eingang	Liefert den zu latchnden Wert über **I.D1** und das Taktsignal über **I.E1**.
Q	adapter::types::unidirectional::AR (Plug)	Ausgang	Stellt den gelatchten Wert über **Q.D1** bereit und signalisiert eine Aktualisierung über **Q.E1**.

Funktionsweise

Der Baustein arbeitet intern mit einem Sub-FB vom Typ `logiBUS::signalprocessing::hysteresis::E_D_FF_ANY_HYS_TMIN` zusammen.

1. **Initialisierung**: Beim Ereignis **INIT** werden die Parameter **HYSTERESIS** und **Tmin** an die interne Logik übergeben. Nach erfolgreicher Initialisierung wird **INITO** ausgegeben.
2. **Datenerfassung**: Bei jedem Taktereignis am Adapter **I.E1** wird der aktuell anliegende Wert **I.D1** mit dem zuletzt gespeicherten Wert verglichen.
   - Eine Übernahme erfolgt nur, wenn die absolute Differenz zwischen **I.D1** und dem gespeicherten Wert größer als **HYSTERESIS** ist.
   - Ist die Differenz kleiner, bleibt der alte Wert unverändert (Hysterese-Funktion).
3. **Ausgabe mit Zeitbegrenzung**: Sobald der neue Wert übernommen wurde, wird das Ausgangssignal **Q.D1** aktualisiert. Das zugehörige Ereignis **Q.E1** wird jedoch nur dann gesendet, wenn seit dem letzten **Q.E1** mindestens die Zeitspanne **Tmin** vergangen ist. Dadurch wird die maximale Ausgabefrequenz begrenzt.

Das gesamte Verhalten kann als taktgesteuerter, hysteresebehafneter Signalhaltebaustein mit Ausgabesperre betrachtet werden.

Technische Besonderheiten

- **Hysterese**: Unterdrückt Rauschen und kleine Schwankungen des Eingangssignals. Erst wenn die Änderung den Wert **HYSTERESIS** überschreitet, wird ein neuer Zustand angenommen.
- **Minimale Intervallzeit (Tmin)**: Verhindert eine zu hohe Ereignisrate am Ausgang. Dies ist insbesondere bei nachgeschalteten, rechenintensiven oder kommunikationsbegrenzten Systemen vorteilhaft.
- **Adapterbasierte Schnittstellen**: Sowohl Eingang als auch Ausgang verwenden den unidirektionalen AR-Adapter, der Daten und Ereignis kombiniert. Dies vereinfacht die modulare Verkettung von Signalverarbeitungsbausteinen.

Zustandsübersicht

Der FB besitzt keinen expliziten FSM (endlichen Automaten) in der oberen Ebene, da die Zustandslogik im internen Sub-FB gekapselt ist. Das logische Verhalten entspricht jedoch einem D-Flip-Flop mit erweiterten Bedingungen:

- **Ruhezustand**: Warten auf den nächsten Takt (I.E1).
- **Prüfzustand**: Vergleich des neuen Werts mit dem gespeicherten Wert unter Berücksichtigung der Hysterese.
- **Ausgabezustand**: Aktualisierung von **Q.D1** und, sofern **Tmin** abgelaufen ist, Senden von **Q.E1**.

Eine grafische Zustandsmaschine kann bei Bedarf aus dem internen Sub-FB abgeleitet werden.

Anwendungsszenarien

- **Messwerterfassung mit Rauschunterdrückung**: Glätten von Sensorsignalen (z. B. Temperatur, Druck) und Weitergabe nur bei signifikanten Änderungen.
- **Zeitgesteuerte Prozessausgaben**: Steuerung von Aktoren, die nicht schneller als eine vorgegebene Taktfrequenz angesteuert werden dürfen.
- **Signalaufbereitung in Feldbussystemen**: Reduzierung der Buslast durch Begrenzung der Ereignisrate, ohne Informationsverlust durch Hysterese.
- **Kaskadierung von Flip-Flops**: Einsatz als stabilisierte Zwischenstufe in Datenketten mit mehreren Abtaststufen.

Vergleich mit ähnlichen Bausteinen

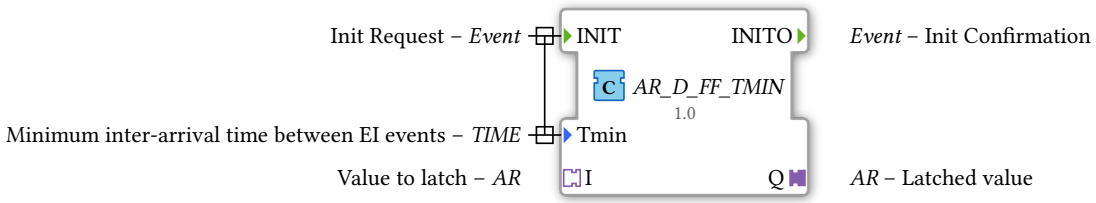
- **Einfaches D-Flip-Flop (z. B. E_D_FF)**: Besitzt weder Hysterese noch Ausgabezeitbegrenzung. Ändert seinen Ausgang bei jedem Takt, unabhängig von der Änderungsgröße oder Frequenz.
- **Hysterese-Flip-Flop (z. B. E_D_FF_HYS)**: Enthält die Hysterese, aber keine minimale Intervallzeit. Ereignisse können sehr schnell aufeinander folgen, wenn die Hysterese ständig überschritten wird.
- **Begrenztes Flip-Flop (ohne Hysterese)**: Nur die minimale Ausgabezeit wird eingehalten; kleine Wertänderungen werden dennoch sofort verarbeitet, was zu Oszillationen führen kann.

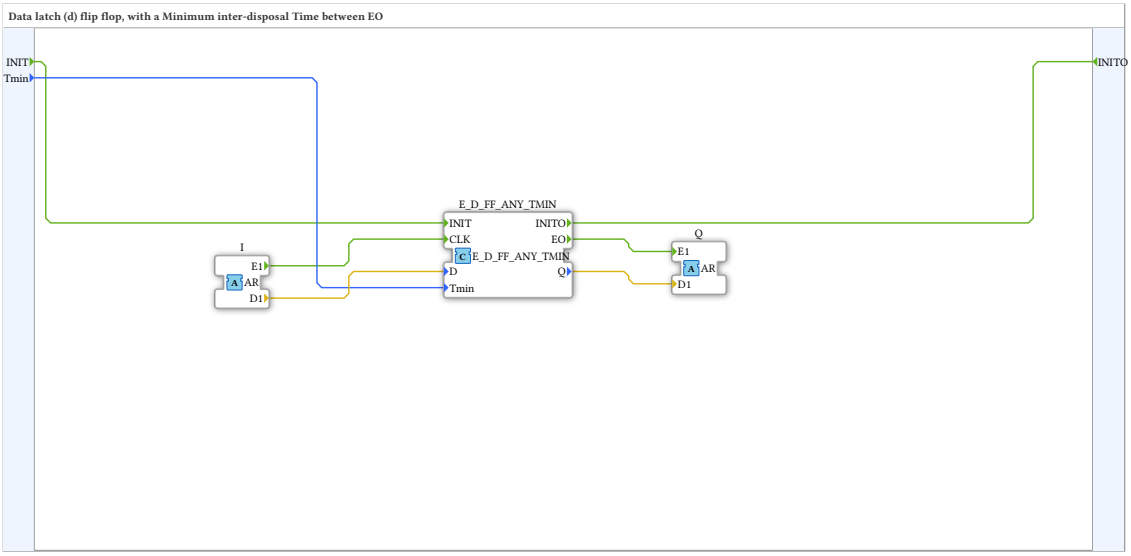
Der **AR_D_FF_HYS_TMIN** vereint beide Mechanismen und bietet damit eine robuste Lösung für praktische Mess- und Steuerungsaufgaben.

Fazit

Der Funktionsblock **AR_D_FF_HYS_TMIN** ist ein vielseitiges Werkzeug zur verlässlichen Signalweiterleitung in Automatisierungssystemen. Durch die Kombination von Hysterese und minimaler interdisposal time werden typische Probleme wie Rauschen, Flimmern und Überlastung nachgeschalteter Verarbeitungsstufen effektiv vermieden. Die adapterbasierte Schnittstelle erleichtert die Einbindung in bestehende modulare Architekturen.

AR_D_FF_TMIN





• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AR_D_FF_TMIN** realisiert ein Data-Latch (D-Flipflop) mit einem Eingangs- und einem Ausgangsadapter. Er zeichnet sich durch die optionale Vorgabe einer Mindestzeit (`Tmin`) zwischen aufeinanderfolgenden Ereignisausgaben aus. Der Kern des Bausteins basiert auf dem Standard-FB `E_D_FF_ANY_TMIN`, der die eigentliche Flipflop-Logik sowie die Zeitsteuerung übernimmt.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Initialisierungsanforderung (mit Parameter `Tmin`)

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Initialisierungsbestätigung

Daten-Eingänge

Name	Typ	Kommentar
Tmin	TIME	Mindestverzögerung zwischen zwei EO-Ereignissen

Daten-Ausgänge

Keine eigenständigen Daten-Ausgänge; der Wert wird über den Adapter `Q` ausgegeben.

Adapter

Richtung	Name	Typ	Kommentar
Socket	I	`adapter::types::unidirectional::AR`	zu übernehmender Wert
Plug	Q	`adapter::types::unidirectional::AR`	gelatchter Wert

Der Adapter `adapter::types::unidirectional::AR` vereint ein Ereignis (`E1`) und ein Datenwort (`D1`) pro Richtung.

Funktionsweise

Der Baustein arbeitet intern mit dem FB `E_D_FF_ANY_TMIN` zusammen:

1. **Initialisierung**: Durch ein Ereignis an `INIT` wird der Parameter `Tmin` an den internen Baustein übergeben. Danach wird `INITO` ausgegeben.
2. **Flipflop-Betrieb**: Ein Ereignis am Adaptereingang `I.E1` (Socket) wird an den **CLK**-Eingang des internen Flipflops weitergeleitet. Gleichzeitig wird der Datenwert `I.D1` an den **D**-Eingang des Flipflops übergeben.
3. **Ausgabe**: Der intern erzeugte Zustand (`Q`) und das zugehörige Ereignis (`EO`) werden – unter Einhaltung der eingestellten Mindestzeit `Tmin` – an den Adapterausgang `Q` (Plug) weitergegeben. Das Ereignis `Q.E1` und der Datenwert `Q.D1` repräsentieren den gelatchten Wert.
4. **Timing**: Der Parameter `Tmin` legt fest, wie viel Zeit zwischen zwei aufeinanderfolgenden `EO`-Ereignissen mindestens vergehen muss. Überschreitet die Frequenz der Eingangsereignisse diesen Grenzwert, wird die Ausgabe entsprechend verzögert oder blockiert.

Technische Besonderheiten

- **Adapterbasierte Kommunikation**: Der FB verwendet ausschließlich unidirektionale AR-Adapter, was ihn für lose gekoppelte oder verteilte Systeme geeignet macht.
- **Wiederverwendung eines Standardbausteins**: Die Kernlogik wird vom IEC-61499-Standardbaustein `E_D_FF_ANY_TMIN` bereitgestellt – somit ist die Funktionalität normkonform und erweiterbar.
- **Konfigurierbare Sperrzeit**: Die Mindestzeit `Tmin` wird zur Initialisierung übergeben und kann während der Laufzeit nicht geändert werden (nur über einen erneuten INIT-Zyklus).

Zustandsübersicht

Der FB selbst besitzt kein eigenes Zustandsdiagramm. Das Verhalten wird vollständig durch den eingebetteten Standardbaustein `E_D_FF_ANY_TMIN` bestimmt. Dieser befindet sich nach der INIT-Phase im Wartezustand und wechselt bei jedem CLK-Ereignis in einen Ausgabezustand, sofern die `Tmin`-Zeit abgelaufen ist.

Anwendungsszenarien

- **Entprellung von Signalen**: Wenn ein Sensorsignal (z. B. über `I`) nur mit einer Mindestpause weitergegeben werden soll, um Preller oder Rauschen zu unterdrücken.
- **Begrenzung der Ereignisrate**: In Regelungs- oder Überwachungssystemen, bei denen nachgeschaltete Komponenten nicht schneller als mit einer definierten Rate bedient werden dürfen.
- **Sichere Übergabe von Daten**: Der gelatchte Wert steht stabil zur Verfügung, bis das nächste CLK-Ereignis (am Adaptereingang) eintrifft.

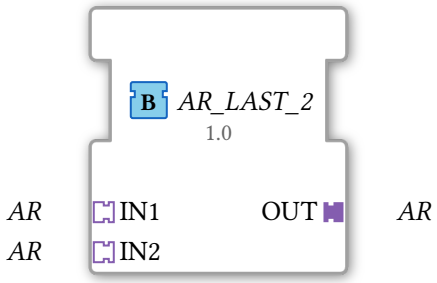
Vergleich mit ähnlichen Bausteinen

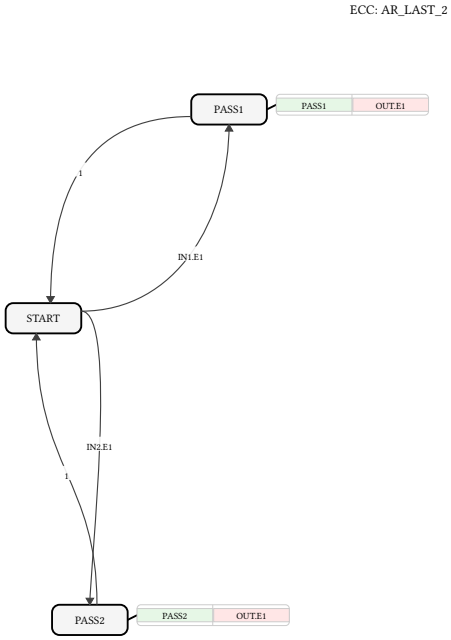
- **AR_D_FF** (ohne TMIN): Ein reiner D-Flipflop-Baustein ohne zeitliche Beschränkung der Ausgabeereignisse. Er eignet sich für Echtzeitanwendungen ohne Verzögerungsanforderung.
- **AR_D_FF_TMIN**: Erweitert den Basisbaustein um die Möglichkeit, die Ereignisrate zu begrenzen, was in industriellen Steuerungen mit Zeitauflagen häufig erforderlich ist.
- **AR_SR_FF**: Setzt Set-Reset-Logik um – im Gegensatz zum hier realisierten D-Flipflop ist der Ausgang nicht taktgesteuert.

Fazit

Der `AR_D_FF_TMIN` ist ein funktional erweiterter D-Flipflop-Baustein, der die Latch-Funktionalität mit einer konfigurierbaren Mindestzeit zwischen Ausgabeereignissen kombiniert. Seine adapterbasierte Schnittstelle und die Nutzung eines standardisierten Kerns machen ihn zu einer flexiblen und normkonformen Komponente für zeitkritische Steuerungsanwendungen im IEC-61499-Umfeld.

AR_LAST_2





Einleitung

Der AR_LAST_2 Funktionsblock führt zwei gleichartige REAL-Adapter-Signale (**IN1**, **IN2**) auf einen gemeinsamen Ausgang (**OUT**) zusammen, nach dem Prinzip **Last Writer Wins**: welcher der beiden Sockets zuletzt ein Ereignis geschrieben hat, gewinnt, und sein Datenwert wird sofort und unverändert an OUT durchgereicht. Es handelt sich ausdrücklich nicht um ein naives Zusammenführen (z. B. Mittelwert oder ODER-Verknüpfung), sondern um eine reine Arbitrierung nach zeitlicher Reihenfolge.

Schnittstellenstruktur

Adapter

Eingangsadapter (Sockets):

- **IN1**: REAL-Adapter (unidirectional) - erste Signalquelle
- **IN2**: REAL-Adapter (unidirectional) - zweite Signalquelle

Ausgangsadapter (Plug):

- **OUT**: REAL-Adapter (unidirectional) - zusammengeführtes Signal

Funktionsweise

Der Baustein ist als einfacher Basic-FB mit einer 3-Zustands-ECC umgesetzt (START, PASS1, PASS2). Trifft an IN1 ein Ereignis ein, wechselt die ECC nach PASS1: der aktuelle Datenwert von IN1 wird nach OUT kopiert und das Adapter-Ereignis an OUT gesendet. Trifft an IN2 ein Ereignis ein, geschieht dasselbe

symmetrisch über PASS2. Nach jeder Durchreichung kehrt die ECC sofort wieder nach START zurück und ist bereit für das nächste Ereignis von IN1 oder IN2.

Technische Besonderheiten

- Last-Writer-Wins-Semantik statt naiver Datenverknüpfung
- Einfacher Basic-FB (ECC mit 3 Zuständen), kein zusätzlicher Zwischenspeicher nötig
- Keine Signalverzögerung: der Wert wird im selben Zyklus durchgereicht, in dem das auslösende Ereignis eintrifft
- Beide Eingänge sind gleichberechtigt - es gibt keine feste Priorität außer der zeitlichen Reihenfolge

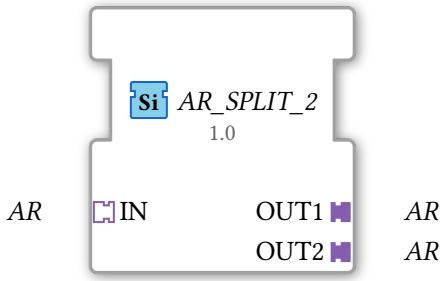
Anwendungsszenarien

- Zusammenführen zweier gleichwertiger REAL-Signalquellen zu einem einzigen nachgeschalteten Adapterpfad (z. B. zwei redundante Sensoren, oder ein manueller und ein automatischer Sollwertgeber)
- Arbitrierung zwischen zwei Bedienstellen, die denselben REAL-Wert schreiben dürfen
- Vereinfachung von Netzwerken, die sonst ein `F_SEL`/`E_RS`-Konstrukt inline verdrahten müssten

⚖️ Vergleich mit ähnlichen Bausteinen

AR_LAST_2 ist die Umkehrung von `AR_SPLIT_2`: AR_SPLIT_2 verteilt 1 Eingang auf 2 Ausgänge, AR_LAST_2 führt 2 Eingänge auf 1 Ausgang zusammen. Für Mischbetrieb zwischen einem Daten- und einem reinen Ereignisadapter (z. B. AX + AE) ist stattdessen `AX_AE_MERGE` gedacht, das nur das Ereignis mischt, ohne den Datenwert zu verändern.

AR_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AR_SPLIT_2** ist ein generischer Funktionsbaustein, der einen eingehenden Adapter vom Typ `adapter::types::unidirectional::AR` auf zwei separate Ausgänge (OUT1, OUT2) aufteilt. Er dient zur Weiterleitung eines Signals an zwei nachfolgende Bausteine, ohne die Daten zu verändern. Der Baustein ist als generischer Typ (`GEN_AR_SPLIT`) implementiert und wird zur Laufzeit parametrisiert.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Typ	Name	Richtung	Beschreibung

`adapter::types::unidirectional::AR`	IN	Socket	Eingangssignal (AR)
`adapter::types::unidirectional::AR`	OUT1	Plug	Erster Ausgang (identisch zu IN)
`adapter::types::unidirectional::AR`	OUT2	Plug	Zweiter Ausgang (identisch zu IN)

Funktionsweise

Der Baustein leitet das am Socket **IN** anliegende AR-Signal unverändert auf beide Plugs **OUT1** und **OUT2** weiter. Es findet keine Datenmanipulation, Filterung oder Verzögerung statt. Die Aufteilung erfolgt rein strukturell: Jeder Ausgang erhält eine eigene Kopie der Referenz auf das zugrundeliegende AR-Objekt.

Technische Besonderheiten

- **Generische Implementierung**: Der FB verwendet das Generic-Class-Name-Attribut (`eclipse4diac::core::GenericClassName`) mit dem Wert `'GEN_AR_SPLIT'`, sodass er anwendungsspezifisch konfiguriert werden kann.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `adapter::types::unidirectional::AR` (nur Vorwärtsrichtung).
- **Keine Zustandsautomatik**: Der Baustein besitzt keinen expliziten ECC (Execution Control Chart); die Weiterleitung erfolgt direkt und ereignisunabhängig.

Zustandsübersicht

Der Baustein implementiert keine Zustandsautomaten. Die Funktionalität beschränkt sich auf die passive Weiterleitung des Eingangssignals an beide Ausgänge. Eine Zustandsvisualisierung ist daher nicht erforderlich.

Anwendungsszenarien

- **Signalverteilung**: Aufteilung eines AR-basierten Sensorsignals an zwei parallel arbeitende Steuerungskomponenten.
- **Redundanz**: Bereitstellung des gleichen AR-Datenstroms für ein primäres und ein redundantes System.
- **Debugging**: Anschluss eines Analyse- oder Logging-Bausteins parallel zum bestehenden Pfad, ohne die Originalsignalkette zu unterbrechen.

Vergleich mit ähnlichen Bausteinen

Ähnliche Funktionalität bieten Bausteine wie `AR_SPLIT_3` oder `AR_SPLIT_N`, die ein Signal auf drei bzw. N Ausgänge verteilen. Die Auswahl hängt von der benötigten Anzahl an Ausgängen ab. Ein Verbinden mehrerer `AR_SPLIT_2`-Instanzen in Kaskade ist ebenfalls möglich, erhöht jedoch die Komplexität gegenüber einem dedizierten Splitter mit mehreren Ausgängen.

- **`AR_SPLIT_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

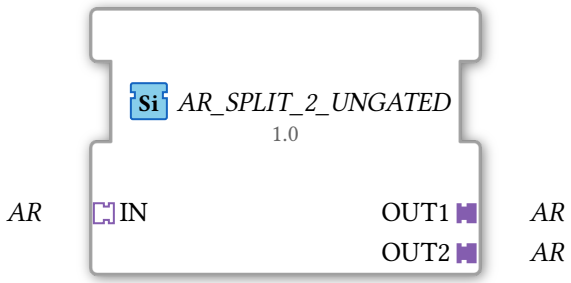
Der **AR_SPLIT_2** ist ein einfacher, aber essenzieller Baustein zur Vervielfachung eines AR-Signals in IEC 61499-basierten Steuerungssystemen. Seine generische Auslegung und die klare Schnittstelle machen ihn zur ersten Wahl, wenn ein Signal an zwei unabhängige Zielbausteine weitergegeben werden muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AR_SPLIT_2_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `AR_SPLIT_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AR_SPLIT_2_UNGATED** ist ein generischer Funktionsbaustein, der einen eingehenden Adapter vom Typ `adapter::types::unidirectional::AR` auf zwei separate Ausgänge (OUT1, OUT2) aufteilt. Er dient zur Weiterleitung eines Signals an zwei nachfolgende Bausteine, ohne die Daten zu verändern. Der Baustein ist als generischer Typ (`GEN_AR_SPLIT`) implementiert und wird zur Laufzeit parametrisiert.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AR`	IN	Socket	Eingangssignal (AR)
`adapter::types::unidirectional::AR`	OUT1	Plug	Erster Ausgang (identisch zu IN)
`adapter::types::unidirectional::AR`	OUT2	Plug	Zweiter Ausgang (identisch zu IN)

Funktionsweise

Der Baustein leitet das am Socket **IN** anliegende AR-Signal unverändert auf beide Plugs **OUT1** und **OUT2** weiter. Es findet keine Datenmanipulation, Filterung oder Verzögerung statt. Die Aufteilung erfolgt rein strukturell: Jeder Ausgang erhält eine eigene Kopie der Referenz auf das zugrundeliegende AR-Objekt.

Technische Besonderheiten

- **Generische Implementierung**: Der FB verwendet das Generic-Class-Name-Attribut (`eclipse4diac::core::GenericClassName`) mit dem Wert `'GEN_AR_SPLIT'`, sodass er anwendungsspezifisch konfiguriert werden kann.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `adapter::types::unidirectional::AR` (nur Vorwärtsrichtung).
- **Keine Zustandsautomatik**: Der Baustein besitzt keinen expliziten ECC (Execution Control Chart); die Weiterleitung erfolgt direkt und ereignisunabhängig.

Zustandsübersicht

Der Baustein implementiert keine Zustandsautomaten. Die Funktionalität beschränkt sich auf die passive Weiterleitung des Eingangssignals an beide Ausgänge. Eine Zustandsvisualisierung ist daher nicht erforderlich.

Anwendungsszenarien

- **Signalverteilung**: Aufteilung eines AR-basierten Sensorsignals an zwei parallel arbeitende Steuerungskomponenten.
- **Redundanz**: Bereitstellung des gleichen AR-Datenstroms für ein primäres und ein redundantes System.
- **Debugging**: Anschluss eines Analyse- oder Logging-Bausteins parallel zum bestehenden Pfad, ohne die Originalsignalkette zu unterbrechen.

Vergleich mit ähnlichen Bausteinen

Ähnliche Funktionalität bieten Bausteine wie `AR_SPLIT_3` oder `AR_SPLIT_N`, die ein Signal auf drei bzw. N Ausgänge verteilen. Die Auswahl hängt von der benötigten Anzahl an Ausgängen ab. Ein Verbinden mehrerer `AR_SPLIT_2_UNGATED`-Instanzen in Kaskade ist ebenfalls möglich, erhöht jedoch die Komplexität gegenüber einem dedizierten Splitter mit mehreren Ausgängen.

- `AR_SPLIT_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

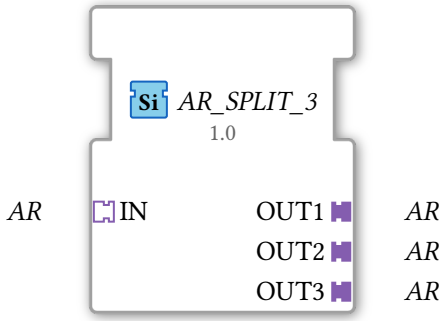
Fazit

Der **AR_SPLIT_2_UNGATED** ist ein einfacher, aber essenzieller Baustein zur Vervielfachung eines AR-Signals in IEC 61499-basierten Steuerungssystemen. Seine generische Auslegung und die klare Schnittstelle machen ihn zur ersten Wahl, wenn ein Signal an zwei unabhängige Zielbausteine weitergegeben werden muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AR_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AR_SPLIT_3** dient als generischer Splitter für Adapterverbindungen. Er verteilt einen eingehenden Daten- und Ereignisstrom (über einen AR-Adapter) auf drei separate, identisch strukturierte Ausgangsadapter. Dies ermöglicht eine effiziente Mehrfachnutzung von Informationen in Automatisierungsabläufen, ohne die Logik zu duplizieren.

Schnittstellenstruktur

Ereignis-Eingänge

Keine – der Baustein besitzt ausschließlich Adapter-Schnittstellen. Ereignisse werden implizit über den Socket IN (vom Typ AR) übertragen.

Ereignis-Ausgänge

Keine – Ereignisse werden über die Plugs OUT1, OUT2 und OUT3 (Typ AR) weitergegeben.

Daten-Eingänge

Keine – alle Daten werden über den Adapter IN bereitgestellt.

Daten-Ausgänge

Keine – alle Daten werden über die drei Ausgangsadapter (OUT1, OUT2, OUT3) ausgegeben.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	IN	adapter::types::unidirectional::AR	Eingehender Adapter als Quelle der Daten/Ereignisse.
Plug	OUT1	adapter::types::unidirectional::AR	Erster ausgehender Adapter (Ziel 1).
Plug	OUT2	adapter::types::unidirectional::AR	Zweiter ausgehender Adapter (Ziel 2).
Plug	OUT3	adapter::types::unidirectional::AR	Dritter ausgehender Adapter (Ziel 3).

Funktionsweise

Der Baustein realisiert eine **1:3-Verteilung** auf Adapterebene. Ein an IN anliegender AR-Adapter (unidirektional) wird intern auf die drei Ausgangsadapter OUT1, OUT2 und OUT3 aufgeschaltet. Jedes an IN eingehende Ereignis oder Datenpaket wird gleichzeitig an alle drei Ausgänge weitergeleitet. Die Verteilung erfolgt ohne Verzögerung und ohne Datenveränderung. Der Baustein ist generisch und kann mit beliebigen AR-Adapterprofilen verwendet werden – seine interne Logik passt sich automatisch der Struktur des eingesetzten Adaptertyps an.

Technische Besonderheiten

- **Generischer Typ**: Der FB ist als generischer Baustein (GEN_AR_SPLIT) ausgelegt und kann mit unterschiedlichen AR-Adapterdefinitionen (unidirektional) verwendet werden, sofern die Schnittstellen kompatibel sind.
- **Unidirektionale Adapter**: Sowohl Eingang als auch Ausgänge sind unidirektional, d.h. die Daten-/Ereignisrichtung ist fest vorgegeben (vom Sender zum Empfänger). Eine Rückkopplung ist über diesen Baustein nicht möglich.
- **Einfache Skalierung**: Die Aufteilung erfolgt ohne zusätzliche Konfiguration. Für andere Anzahlen von Ausgängen können Varianten wie AR_SPLIT_2 oder ein parametrierbarer Splitter (AR_SPLIT_N) eingesetzt werden.
- **Keine Zustandsänderung**: Der Baustein besitzt keinen internen Speicher und keine Zustandsmaschine – jedes Ereignis wird sofort und vollständig repliziert.

Zustandsübersicht

Der Baustein ist **zustandslos** (kombinatorisch). Es gibt keine internen Zustände oder sequenziellen Abläufe. Die Verteilung erfolgt rein durch Verbindung der Adapter-Signale.

Anwendungsszenarien

- **Verteilen von Sensordaten** an mehrere parallele Verarbeitungslogiken (z.B. Auswertung, Visualisierung, Protokollierung).
- **Ansteuern mehrerer Aktorik-Gruppen** mit einem identischen Steuersignal (z.B. parallele Ventil- oder Motorsteuerung).
- **Aufteilen eines Steuerungsereignisses** in verschiedene Funktionsbausteine, ohne die Quelllogik zu kopieren.
- **Debugging und Monitoring**: Ein Signal kann an einen echten Empfänger und parallel an eine Überwachungskomponente geleitet werden.

Vergleich mit ähnlichen Bausteinen

Baustein	Beschreibung
AR_SPLIT_2	Verteilung auf zwei Ausgänge – kompakter für kleinere Anwendungen.
AR_SPLIT_N	Verteilung auf eine konfigurierbare Anzahl von Ausgängen (generischer).
AR_CONNECT	Einfache 1:1-Verbindung (keine Verteilung).

Im Gegensatz zu dedizierten Splittern mit Daten- oder Ereignis-Ein-/Ausgängen arbeitet der `AR_SPLIT_3` ausschließlich über Adapter, was ihn besonders flexibel und typenunabhängig macht.

- `AR_SPLIT_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

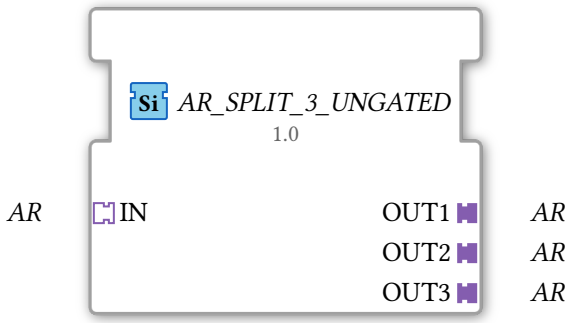
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

AR_SPLIT_3 ist ein einfacher, aber nützlicher generischer Funktionsbaustein zur Mehrfachverteilung von Adapter-Signalen. Er reduziert den Aufwand für die Signalvervielfältigung in IEC 61499-Applikationen und trägt durch seine generische Natur zur Wiederverwendbarkeit und Übersichtlichkeit von Steuerungslogiken bei. Dank des reinen Adapter-Ansatzes kann er ohne Modifikation in unterschiedlichste Umgebungen integriert werden.

AR_SPLIT_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AR_SPLIT_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AR_SPLIT_3_UNGATED** dient als generischer Splitter für Adapterverbindungen. Er verteilt einen eingehenden Daten- und Ereignisstrom (über einen AR-Adapter) auf drei separate, identisch strukturierte Ausgangsadapter. Dies ermöglicht eine effiziente Mehrfachnutzung von Informationen in Automatisierungsabläufen, ohne die Logik zu duplizieren.

Schnittstellenstruktur

Ereignis-Eingänge

Keine – der Baustein besitzt ausschließlich Adapter-Schnittstellen. Ereignisse werden implizit über den Socket IN (vom Typ AR) übertragen.

Ereignis-Ausgänge

Keine – Ereignisse werden über die Plugs OUT1, OUT2 und OUT3 (Typ AR) weitergegeben.

Daten-Eingänge

Keine – alle Daten werden über den Adapter IN bereitgestellt.

Daten-Ausgänge

Keine – alle Daten werden über die drei Ausgangsadapter (OUT1, OUT2, OUT3) ausgegeben.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	IN	adapter::types::unidirectional::AR	Eingehender Adapter als Quelle der Daten/Ereignisse.
Plug	OUT1	adapter::types::unidirectional::AR	Erster ausgehender Adapter (Ziel 1).
Plug	OUT2	adapter::types::unidirectional::AR	Zweiter ausgehender Adapter (Ziel 2).
Plug	OUT3	adapter::types::unidirectional::AR	Dritter ausgehender Adapter (Ziel 3).

Funktionsweise

Der Baustein realisiert eine **1:3-Verteilung** auf Adapterebene. Ein an IN anliegender AR-Adapter (unidirektional) wird intern auf die drei Ausgangsadapter OUT1, OUT2 und OUT3 aufgeschaltet. Jedes an IN eingehende Ereignis oder Datenpaket wird gleichzeitig an alle drei Ausgänge weitergeleitet. Die Verteilung erfolgt ohne Verzögerung und ohne Datenveränderung. Der Baustein ist generisch und kann mit beliebigen AR-Adapterprofilen verwendet werden – seine interne Logik passt sich automatisch der Struktur des eingesetzten Adaptertyps an.

Technische Besonderheiten

- **Generischer Typ**: Der FB ist als generischer Baustein (GEN_AR_SPLIT) ausgelegt und kann mit unterschiedlichen AR-Adapterdefinitionen (unidirektional) verwendet werden, sofern die Schnittstellen kompatibel sind.
- **Unidirektionale Adapter**: Sowohl Eingang als auch Ausgänge sind unidirektional, d.h. die Daten-/Ereignisrichtung ist fest vorgegeben (vom Sender zum Empfänger). Eine Rückkopplung ist über diesen Baustein nicht möglich.
- **Einfache Skalierung**: Die Aufteilung erfolgt ohne zusätzliche Konfiguration. Für andere Anzahlen von Ausgängen können Varianten wie AR_SPLIT_2 oder ein parametrierbarer Splitter (AR_SPLIT_N) eingesetzt werden.
- **Keine Zustandsänderung**: Der Baustein besitzt keinen internen Speicher und keine Zustandsmaschine – jedes Ereignis wird sofort und vollständig repliziert.

Zustandsübersicht

Der Baustein ist **zustandslos** (kombinatorisch). Es gibt keine internen Zustände oder sequenziellen Abläufe. Die Verteilung erfolgt rein durch Verbindung der Adapter-Signale.

Anwendungsszenarien

- **Verteilen von Sensordaten** an mehrere parallele Verarbeitungslogiken (z.B. Auswertung, Visualisierung, Protokollierung).
- **Ansteuern mehrerer Aktorik-Gruppen** mit einem identischen Steuersignal (z.B. parallele Ventil- oder Motorsteuerung).
- **Aufteilen eines Steuerungsereignisses** in verschiedene Funktionsbausteine, ohne die Quelllogik zu kopieren.
- **Debugging und Monitoring**: Ein Signal kann an einen echten Empfänger und parallel an eine Überwachungskomponente geleitet werden.

Vergleich mit ähnlichen Bausteinen

Baustein	Beschreibung
AR_SPLIT_2	Verteilung auf zwei Ausgänge – kompakter für kleinere Anwendungen.
AR_SPLIT_N	Verteilung auf eine konfigurierbare Anzahl von Ausgängen (generischer).
AR_CONNECT	Einfache 1:1-Verbindung (keine Verteilung).

Im Gegensatz zu dedizierten Splittern mit Daten- oder Ereignis-Ein-/Ausgängen arbeitet der `AR_SPLIT_3_UNGATED` ausschließlich über Adapter, was ihn besonders flexibel und typenunabhängig macht.

- `AR_SPLIT_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

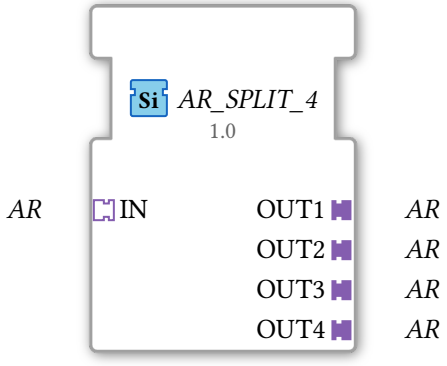
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

AR_SPLIT_3_UNGATED ist ein einfacher, aber nützlicher generischer Funktionsbaustein zur Mehrfachverteilung von Adapter-Signalen. Er reduziert den Aufwand für die Signalvervielfältigung in IEC 61499-Applikationen und trägt durch seine generische Natur zur Wiederverwendbarkeit und Übersichtlichkeit von Steuerungslogiken bei. Dank des reinen Adapter-Ansatzes kann er ohne Modifikation in unterschiedlichste Umgebungen integriert werden.

AR_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AR_SPLIT_4** dient dazu, einen eingehenden AR-Adapter (unidirektional) auf vier ausgehende AR-Adapter aufzuteilen. Er ist als generischer Baustein (`GenericClassName = 'GEN_AR_SPLIT'`) implementiert und ermöglicht eine einfache Signalvervielfachung für adapterbasierte Kommunikation in Steuerungssystemen.

Schnittstellenstruktur

Ereignis-Eingänge
Keine

Ereignis-Ausgänge
Keine

Daten-Eingänge

Name	Typ	Kommentar
(keine)	–	–

Daten-Ausgänge

Name	Typ	Kommentar
(keine)	–	–

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AR`	**IN**	Socket (Eingang)	Einzugehender AR-Adaptersignal, das auf die vier Ausgänge verteilt wird.
`adapter::types::unidirectional::AR`	**OUT1**	Plug (Ausgang)	Erster ausgehender AR-Kanal.
`adapter::types::unidirectional::AR`	**OUT2**	Plug (Ausgang)	Zweiter ausgehender AR-Kanal.
`adapter::types::unidirectional::AR`	**OUT3**	Plug (Ausgang)	Dritter ausgehender AR-Kanal.
`adapter::types::unidirectional::AR`	**OUT4**	Plug (Ausgang)	Vierter ausgehender AR-Kanal.

Funktionsweise

Der Baustein nimmt über den Socket **IN** ein AR-Adaptersignal entgegen und leitet es unverändert an alle vier Plugs **OUT1** bis **OUT4** weiter. Die Aufteilung erfolgt parallel – jedes über **IN** eingehende Ereignis oder jeder Datenwert wird gleichzeitig auf allen Ausgängen bereitgestellt. Der FB selbst führt keine logische Verarbeitung oder Umformung durch; er fungiert als reiner Verteiler (Splitter) für unidirektionale AR-Adapter.

Technische Besonderheiten

- **Generischer Typ**: Der Baustein ist als generischer FB (`GenericClassName`) definiert, was bedeutet, dass der konkrete AR-Adaptertyp zur Übersetzungszeit festgelegt wird. Dies erhöht die Wiederverwendbarkeit in verschiedenen Anwendungen.
- **Keine Zustandsmaschine**: Der FB besitzt kein ECC (Execution Control Chart) und keine internen Zustände. Alle Operationen sind rein kombinatorisch bzw. ereignisgetrieben.
- **Unidirektional**: Der Adaptertyp ist als unidirektional spezifiziert; eine Rückmeldung vom Zielbaustein ist nicht vorgesehen.

Zustandsübersicht

Da der FB keine Zustandsmaschine besitzt, existieren keine expliziten Zustände. Das Verhalten ist deterministisch: Ein eingehender Trigger am Socket **IN** wird sofort an alle Ausgänge weitergeleitet.

Anwendungsszenarien

- **Signalvervielfachung**: Ein Steuerungssignal (z. B. „Start“ oder „Reset“) soll mehrere parallele Funktionsblöcke gleichzeitig erreichen.
- **Überwachung und Parallelverarbeitung**: Ein Messwert oder ein Alarm wird an mehrere unabhängige Auswerteblöcke verteilt.

- **Test und Simulation**: Während der Entwicklung kann ein Signal aufgezeichnet oder simuliert werden, während es gleichzeitig an den eigentlichen Zielblock geht.

Vergleich mit ähnlichen Bausteinen

- **AR_SPLIT_2**: Teilt ein AR-Signal auf zwei Ausgänge auf – geeignet für kleinere Verteilungen.
- **AR_SPLIT_N**: Ein nicht standardisierter, konfigurierbarer Splitter (z. B. über Parameter), der für beliebig viele Ausgänge ausgelegt ist.
- **AR_MERGE**: Führt mehrere AR-Signale zu einem zusammen – die Umkehrfunktion des AR_SPLIT_4.

Gegenüber diesen Alternativen bietet AR_SPLIT_4 eine feste Anzahl von vier Ausgängen, was eine kompakte und übersichtliche Implementierung ohne Konfigurationsaufwand ermöglicht.

- `AR_SPLIT_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

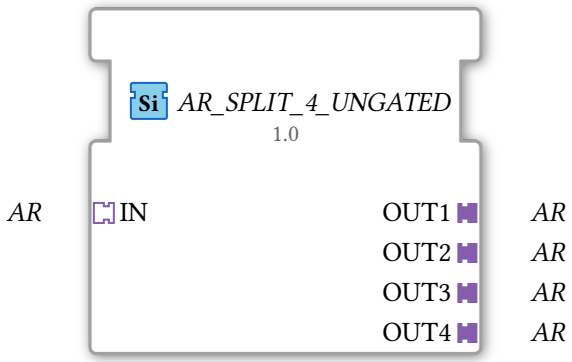
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der **AR_SPLIT_4** ist ein einfacher, aber nützlicher generischer Funktionsblock zur Verteilung eines unidirektionalen AR-Adaptersignals auf vier parallele Pfade. Aufgrund seines schlanken Designs (keine Ereignis-/Datenschnittstellen, keine Zustandsmaschine) eignet er sich ideal für alle Anwendungen, bei denen ein Signal mehrfach benötigt wird, ohne zusätzliche Logik einzubringen.

AR_SPLIT_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AR_SPLIT_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AR_SPLIT_4_UNGATED** dient dazu, einen eingehenden AR-Adapter (unidirektional) auf vier ausgehende AR-Adapter aufzuteilen. Er ist als generischer Baustein (`GenericClassName = 'GEN_AR_SPLIT'`) implementiert und ermöglicht eine einfache Signalvervielfachung für adapterbasierte Kommunikation in Steuerungssystemen.

Schnittstellenstruktur

Ereignis-Eingänge
Keine

Ereignis-Ausgänge
Keine

Daten-Eingänge

Name	Typ	Kommentar
(keine)	–	–

Daten-Ausgänge

Name	Typ	Kommentar
(keine)	–	–

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AR`	**IN**	Socket (Eingang)	Einzugehender AR-Adaptersignal, das auf die vier Ausgänge verteilt wird.
`adapter::types::unidirectional::AR`	**OUT1**	Plug (Ausgang)	Erster ausgehender AR-Kanal.
`adapter::types::unidirectional::AR`	**OUT2**	Plug (Ausgang)	Zweiter ausgehender AR-Kanal.
`adapter::types::unidirectional::AR`	**OUT3**	Plug (Ausgang)	Dritter ausgehender AR-Kanal.
`adapter::types::unidirectional::AR`	**OUT4**	Plug (Ausgang)	Vierter ausgehender AR-Kanal.

Funktionsweise
Der Baustein nimmt über den Socket **IN** ein AR-Adaptersignal entgegen und leitet es unverändert an alle vier Plugs **OUT1** bis **OUT4** weiter. Die Aufteilung erfolgt parallel – jedes über **IN** eingehende Ereignis oder jeder Datenwert wird gleichzeitig auf allen Ausgängen bereitgestellt. Der FB selbst führt keine logische Verarbeitung oder Umformung durch; er fungiert als reiner Verteiler (Splitter) für unidirektionale AR-Adapter.

Technische Besonderheiten
- **Generischer Typ**: Der Baustein ist als generischer FB (`GenericClassName`) definiert, was bedeutet, dass der konkrete AR-Adaptertyp zur Übersetzungszeit festgelegt wird. Dies erhöht die Wiederverwendbarkeit in verschiedenen Anwendungen.
- **Keine Zustandsmaschine**: Der FB besitzt kein ECC (Execution Control Chart) und keine internen Zustände. Alle Operationen sind rein kombinatorisch bzw. ereignisgetrieben.
- **Unidirektional**: Der Adaptertyp ist als unidirektional spezifiziert; eine Rückmeldung vom Zielbaustein ist nicht vorgesehen.

Zustandsübersicht
Da der FB keine Zustandsmaschine besitzt, existieren keine expliziten Zustände. Das Verhalten ist deterministisch: Ein eingehender Trigger am Socket **IN** wird sofort an alle Ausgänge weitergeleitet.

Anwendungsszenarien
- **Signalvervielfachung**: Ein Steuerungssignal (z. B. „Start“ oder „Reset“) soll mehrere parallele Funktionsblöcke gleichzeitig erreichen.
- **Überwachung und Parallelverarbeitung**: Ein Messwert oder ein Alarm wird an mehrere unabhängige Auswerteblöcke verteilt.

- **Test und Simulation**: Während der Entwicklung kann ein Signal aufgezeichnet oder simuliert werden, während es gleichzeitig an den eigentlichen Zielblock geht.

Vergleich mit ähnlichen Bausteinen

- **AR_SPLIT_2**: Teilt ein AR-Signal auf zwei Ausgänge auf – geeignet für kleinere Verteilungen.
- **AR_SPLIT_N**: Ein nicht standardisierter, konfigurierbarer Splitter (z. B. über Parameter), der für beliebig viele Ausgänge ausgelegt ist.
- **AR_MERGE**: Führt mehrere AR-Signale zu einem zusammen – die Umkehrfunktion des AR_SPLIT_4_UNGATED.

Gegenüber diesen Alternativen bietet AR_SPLIT_4_UNGATED eine feste Anzahl von vier Ausgängen, was eine kompakte und übersichtliche Implementierung ohne Konfigurationsaufwand ermöglicht.

- `AR_SPLIT_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

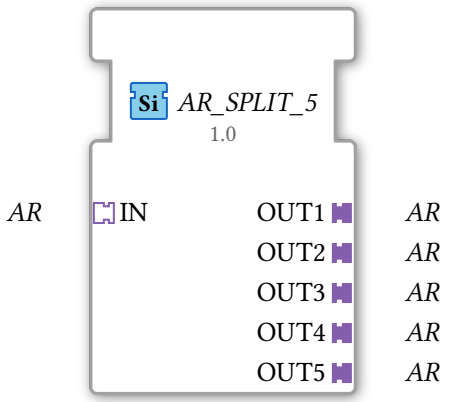
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AR_SPLIT_4_UNGATED** ist ein einfacher, aber nützlicher generischer Funktionsblock zur Verteilung eines unidirektionalen AR-Adaptersignals auf vier parallele Pfade. Aufgrund seines schlanken Designs (keine Ereignis-/Datenschnittstellen, keine Zustandsmaschine) eignet er sich ideal für alle Anwendungen, bei denen ein Signal mehrfach benötigt wird, ohne zusätzliche Logik einzubringen.

AR_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AR_SPLIT_5 dient dazu, einen eingehenden AR-Adapter in fünf identische AR-Adapter-Ausgänge aufzuteilen. Es handelt sich um einen generischen Baustein, der speziell für die Verteilung von unidirektionalen Adaptern (Typ: `adapter::types::unidirectional::AR`) konzipiert ist.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

- **Socket (Eingang)**:
  - `IN`: Typ `adapter::types::unidirectional::AR` – Empfängt einen einzigen AR-Adapter.
- **Plugs (Ausgänge)**:
  - `OUT1`: Typ `adapter::types::unidirectional::AR`
  - `OUT2`: Typ `adapter::types::unidirectional::AR`
  - `OUT3`: Typ `adapter::types::unidirectional::AR`
  - `OUT4`: Typ `adapter::types::unidirectional::AR`
  - `OUT5`: Typ `adapter::types::unidirectional::AR` Alle Ausgänge sind identisch aufgebaut und geben den eingehenden AR-Adapter unverändert weiter.

Funktionsweise

Der Baustein arbeitet als passiver Verteiler: Sobald ein AR-Adapter an den Eingang `IN` angeschlossen wird, wird dieser auf die fünf Ausgänge `OUT1` bis `OUT5` vervielfältigt. Jede Verbindung erhält eine unabhängige, aber identische Kopie des Adapters. Es findet keine Datenverarbeitung oder -änderung statt.

Technische Besonderheiten

- **Unidirektionalität**: Die Adapter des Typs `AR` sind ausschließlich unidirektional ausgelegt, d.h. Signale fließen nur vom Eingang zu den Ausgängen.
- **Generische Implementierung**: Der FB ist als generischer Typ (`GEN_AR_SPLIT`) realisiert, was eine einfache Wiederverwendung und Anpassung an andere Adapter-Typen ermöglicht.
- **Keine Zustandslogik**: Der Baustein besitzt keine internen Zustände oder Ereignisse; die Aufteilung erfolgt durch reine Verbindungstechnik.

Zustandsübersicht

Der AR_SPLIT_5 verfügt über keinen eigenen Zustandsautomaten. Sein Verhalten ist statisch und rein strukturell: Der Eingang wird direkt auf alle Ausgänge durchgeschleift.

Anwendungsszenarien

- **Signalverteilung**: Wenn ein AR-Adapter ein Steuer- oder Datensignal liefert, das an mehrere nachfolgende Komponenten (z.B. Aktoren, Sensoren) weitergegeben werden muss.
- **Module Kaskadierung**: In komplexen Automatisierungssystemen, in denen ein AR-Signal an verschiedene Subsysteme aufgeteilt werden soll.

Vergleich mit ähnlichen Bausteinen

- **AR_SPLIT_2, AR_SPLIT_3, AR_SPLIT_4**: Diese Bausteine teilen einen AR-Adapter auf zwei, drei bzw. vier Ausgänge auf. Der AR_SPLIT_5 ist die Erweiterung auf fünf Ausgänge.
- **AR_MERGE_***: Im Gegensatz zu Split-Bausteinen führen Merge-Bausteine mehrere AR-Adapter zu einem zusammen.
- **`AR_SPLIT_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

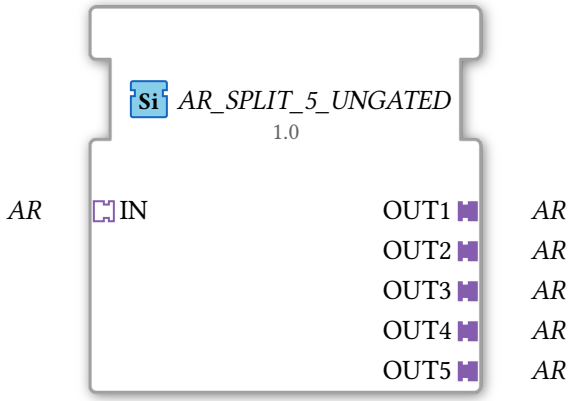
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der AR_SPLIT_5 ist ein einfacher, aber nützlicher Funktionsblock zur Verteilung eines unidirektionalen AR-Adapters auf fünf unabhängige Ausgänge. Seine generische und ereignislose Struktur macht ihn zu einem grundlegenden Werkzeug für die Adapter-basierte Kommunikation in IEC 61499-Systemen.

AR_SPLIT_5_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `AR_SPLIT_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AR_SPLIT_5_UNGATED dient dazu, einen eingehenden AR-Adapter in fünf identische AR-Adapter-Ausgänge aufzuteilen. Es handelt sich um einen generischen Baustein, der speziell für die Verteilung von unidirektionalen Adaptern (Typ: `adapter::types::unidirectional::AR`) konzipiert ist.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

- **Socket (Eingang)**:
  - `IN`: Typ `adapter::types::unidirectional::AR` – Empfängt einen einzigen AR-Adapter.
- **Plugs (Ausgänge)**:
  - `OUT1`: Typ `adapter::types::unidirectional::AR`
  - `OUT2`: Typ `adapter::types::unidirectional::AR`
  - `OUT3`: Typ `adapter::types::unidirectional::AR`
  - `OUT4`: Typ `adapter::types::unidirectional::AR`
  - `OUT5`: Typ `adapter::types::unidirectional::AR` Alle Ausgänge sind identisch aufgebaut und geben den eingehenden AR-Adapter unverändert weiter.

Funktionsweise

Der Baustein arbeitet als passiver Verteiler: Sobald ein AR-Adapter an den Eingang `IN` angeschlossen wird, wird dieser auf die fünf Ausgänge `OUT1` bis `OUT5` vervielfältigt. Jede Verbindung erhält eine unabhängige, aber identische Kopie des Adapters. Es findet keine Datenverarbeitung oder -änderung statt.

Technische Besonderheiten

- **Unidirektionalität**: Die Adapter des Typs `AR` sind ausschließlich unidirektional ausgelegt, d.h. Signale fließen nur vom Eingang zu den Ausgängen.
- **Generische Implementierung**: Der FB ist als generischer Typ (`GEN_AR_SPLIT`) realisiert, was eine einfache Wiederverwendung und Anpassung an andere Adapter-Typen ermöglicht.
- **Keine Zustandslogik**: Der Baustein besitzt keine internen Zustände oder Ereignisse; die Aufteilung erfolgt durch reine Verbindungstechnik.

Zustandsübersicht

Der AR_SPLIT_5_UNGATED verfügt über keinen eigenen Zustandsautomaten. Sein Verhalten ist statisch und rein strukturell: Der Eingang wird direkt auf alle Ausgänge durchgeschleift.

Anwendungsszenarien

- **Signalverteilung**: Wenn ein AR-Adapter ein Steuer- oder Datensignal liefert, das an mehrere nachfolgende Komponenten (z.B. Aktoren, Sensoren) weitergegeben werden muss.
- **Module Kaskadierung**: In komplexen Automatisierungssystemen, in denen ein AR-Signal an verschiedene Subsysteme aufgeteilt werden soll.

Vergleich mit ähnlichen Bausteinen

- **AR_SPLIT_2, AR_SPLIT_3, AR_SPLIT_4**: Diese Bausteine teilen einen AR-Adapter auf zwei, drei bzw. vier Ausgänge auf. Der AR_SPLIT_5_UNGATED ist die Erweiterung auf fünf Ausgänge.
- **AR_MERGE_***: Im Gegensatz zu Split-Bausteinen führen Merge-Bausteine mehrere AR-Adapter zu einem zusammen.
- `AR_SPLIT_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

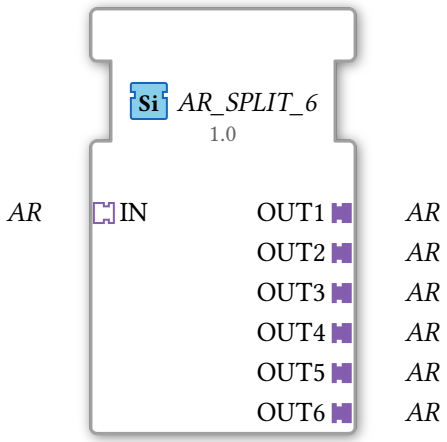
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der AR_SPLIT_5_UNGATED ist ein einfacher, aber nützlicher Funktionsblock zur Verteilung eines unidirektionalen AR-Adapters auf fünf unabhängige Ausgänge. Seine generische und ereignislose Struktur macht ihn zu einem grundlegenden Werkzeug für die Adapter-basierte Kommunikation in IEC 61499-Systemen.

AR_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AR_SPLIT_6** dient dazu, einen einzelnen AR-Adapter-Eingang auf sechs separate AR-Adapter-Ausgänge zu verteilen. Er ist als generischer Baustein (Generic FB) ausgelegt und ermöglicht eine einfache Vervielfachung einer AR-Schnittstelle ohne Datenverlust oder protokollspezifische Umwandlung.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden. Die Funktionalität wird rein über die Adapter-Schnittstelle gesteuert.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AR`	**IN**	Socket (Eingang)	Ein eingehender AR-Adapter, der auf alle Ausgänge verteilt wird.
`adapter::types::unidirectional::AR`	**OUT1**	Plug (Ausgang)	Erster ausgehender AR-Adapter (identisch mit IN).
`adapter::types::unidirectional::AR`	**OUT2**	Plug (Ausgang)	Zweiter ausgehender AR-Adapter (identisch mit IN).
`adapter::types::unidirectional::AR`	**OUT3**	Plug (Ausgang)	Dritter ausgehender AR-Adapter (identisch mit IN).
`adapter::types::unidirectional::AR`	**OUT4**	Plug (Ausgang)	Vierter ausgehender AR-Adapter (identisch mit IN).
`adapter::types::unidirectional::AR`	**OUT5**	Plug (Ausgang)	Fünfter ausgehender AR-Adapter (identisch mit IN).
`adapter::types::unidirectional::AR`	**OUT6**	Plug (Ausgang)	Sechster ausgehender AR-Adapter (identisch mit IN).

Funktionsweise

Der Baustein leitet die Signale des eingehenden AR-Adapters (Socket **IN**) unverändert an alle sechs Ausgangs-Adapter (**OUT1** … **OUT6**) weiter. Es findet keine logische oder zeitliche Verarbeitung statt – jede Änderung am Eingang wird sofort auf allen Ausgängen sichtbar. Der FB arbeitet rein passiv und benötigt keine Ereignissteuerung.

Technische Besonderheiten

- **Generischer Baustein (Generic FB):** Der FB ist als generischer Typ deklariert (`eclipse4diac::core::GenericClassName = 'GEN_AR_SPLIT'`), sodass er in verschiedenen Kontexten wiederverwendet werden kann, ohne den zugrundeliegenden AR-Typ neu definieren zu müssen.
- **Unidirektionale Adapter:** Alle verwendeten Adapter sind vom Typ `adapter::types::unidirectional::AR`, d.h. die Daten fließen nur vom Socket zu den Plugs.
- **Keine Zustandsmaschine:** Da der FB keine Ereignisse oder interne Logik besitzt, existiert kein ECC (Execution Control Chart).

Zustandsübersicht

Dieser FB enthält keine Zustandsmaschine. Er arbeitet datengetrieben und stellt die Ein-/Ausgangsbeziehung kontinuierlich und ohne Verzögerung her.

Anwendungsszenarien

- **Verteilen eines AR-Signals an mehrere Verbraucher:** Wenn ein Sensor oder Aktor seine Daten über einen AR-Adapter bereitstellt, können mit diesem FB mehrere nachgeschaltete Module parallel versorgt werden (z.B. mehrere Regler oder Anzeigen).
- **Test- und Simulationsumgebungen:** Der FB eignet sich, um ein einzelnes Testsignal auf mehrere Auswerteeinheiten gleichzeitig zu schalten.
- **Redundanz oder Parallelisierung:** In Steuerungen, die mehrere identische Verarbeitungsketten benötigen, wird der Eingang einfach dupliziert.

Vergleich mit ähnlichen Bausteinen

- **AR_SPLIT_2 / AR_SPLIT_4:** Diese Bausteine teilen den Eingang auf zwei bzw. vier Ausgänge auf und unterscheiden sich nur in der Anzahl der Ausgänge. Der AR_SPLIT_6 erweitert dies auf sechs Ausgänge.
- **Allgemeine Split-FBs:** Es existieren weitere Split-Bausteine für Datentypen (z.B. INT_SPLIT), die jedoch auf spezifische Datenformate ausgelegt sind. Dieser FB arbeitet ausschließlich auf Adapterebene und ist dadurch flexibler, wenn der Datenaustausch über Adapter erfolgt.
- **Eigene Implementierung:** Alternativ könnte die Aufteilung auch durch manuelles Verkabeln mehrerer Adapterknoten erfolgen, was jedoch die Übersichtlichkeit und Wartbarkeit verschlechtert.
- `AR_SPLIT_6_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

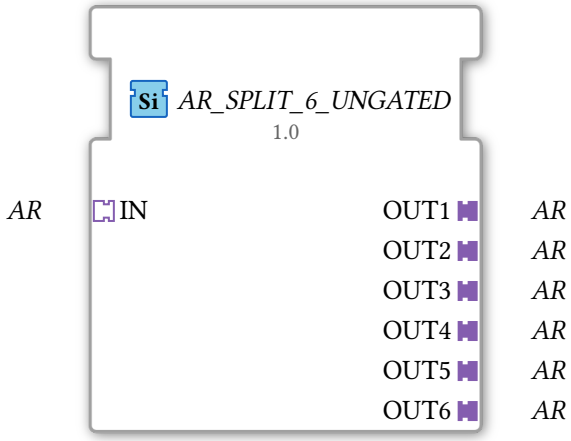
Der **AR_SPLIT_6** ist ein einfacher, aber nützlicher generischer Baustein zur Vervielfachung eines AR-Adapter-Signals auf sechs parallele Ausgänge. Er zeichnet sich durch minimale Komplexität, klare Struktur und hohe Wiederverwendbarkeit in Automatisierungsprojekten aus, die auf der 4diac-IDE und dem IEC 61499-Standard basieren.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AR_SPLIT_6_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AR_SPLIT_6`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AR_SPLIT_6_UNGATED** dient dazu, einen einzelnen AR-Adapter-Eingang auf sechs separate AR-Adapter-Ausgänge zu verteilen. Er ist als generischer Baustein (Generic FB) ausgelegt und ermöglicht eine einfache Vervielfachung einer AR-Schnittstelle ohne Datenverlust oder protokollspezifische Umwandlung.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden. Die Funktionalität wird rein über die Adapter-Schnittstelle gesteuert.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AR`	**IN**	Socket (Eingang)	Ein eingehender AR-Adapter, der auf alle Ausgänge verteilt wird.
`adapter::types::unidirectional::AR`	**OUT1**	Plug (Ausgang)	Erster ausgehender AR-Adapter (identisch mit IN).
`adapter::types::unidirectional::AR`	**OUT2**	Plug (Ausgang)	Zweiter ausgehender AR-Adapter (identisch mit IN).
`adapter::types::unidirectional::AR`	**OUT3**	Plug (Ausgang)	Dritter ausgehender AR-Adapter (identisch mit IN).
`adapter::types::unidirectional::AR`	**OUT4**	Plug (Ausgang)	Vierter ausgehender AR-Adapter (identisch mit IN).
`adapter::types::unidirectional::AR`	**OUT5**	Plug (Ausgang)	Fünfter ausgehender AR-Adapter (identisch mit IN).

adapter::types::unidirectional::AR	**OUT6**	Plug (Ausgang)	Sechster ausgehender AR-Adapter (identisch mit IN).

Funktionsweise

Der Baustein leitet die Signale des eingehenden AR-Adapters (Socket **IN**) unverändert an alle sechs Ausgangs-Adapter (**OUT1** … **OUT6**) weiter. Es findet keine logische oder zeitliche Verarbeitung statt – jede Änderung am Eingang wird sofort auf allen Ausgängen sichtbar. Der FB arbeitet rein passiv und benötigt keine Ereignissteuerung.

Technische Besonderheiten

- **Generischer Baustein (Generic FB):** Der FB ist als generischer Typ deklariert (`eclipse4diac::core::GenericClassName = 'GEN_AR_SPLIT'`), sodass er in verschiedenen Kontexten wiederverwendet werden kann, ohne den zugrundeliegenden AR-Typ neu definieren zu müssen.
- **Unidirektionale Adapter:** Alle verwendeten Adapter sind vom Typ `adapter::types::unidirectional::AR`, d.h. die Daten fließen nur vom Socket zu den Plugs.
- **Keine Zustandsmaschine:** Da der FB keine Ereignisse oder interne Logik besitzt, existiert kein ECC (Execution Control Chart).

Zustandsübersicht

Dieser FB enthält keine Zustandsmaschine. Er arbeitet datengetrieben und stellt die Ein-/Ausgangsbeziehung kontinuierlich und ohne Verzögerung her.

Anwendungsszenarien

- **Verteilen eines AR-Signals an mehrere Verbraucher:** Wenn ein Sensor oder Aktor seine Daten über einen AR-Adapter bereitstellt, können mit diesem FB mehrere nachgeschaltete Module parallel versorgt werden (z.B. mehrere Regler oder Anzeigen).
- **Test- und Simulationsumgebungen:** Der FB eignet sich, um ein einzelnes Testsignal auf mehrere Auswerteeinheiten gleichzeitig zu schalten.
- **Redundanz oder Parallelisierung:** In Steuerungen, die mehrere identische Verarbeitungsketten benötigen, wird der Eingang einfach dupliziert.

Vergleich mit ähnlichen Bausteinen

- **AR_SPLIT_2 / AR_SPLIT_4:** Diese Bausteine teilen den Eingang auf zwei bzw. vier Ausgänge auf und unterscheiden sich nur in der Anzahl der Ausgänge. Der AR_SPLIT_6_UNGATED erweitert dies auf sechs Ausgänge.
- **Allgemeine Split-FBs:** Es existieren weitere Split-Bausteine für Datentypen (z.B. INT_SPLIT), die jedoch auf spezifische Datenformate ausgelegt sind. Dieser FB arbeitet ausschließlich auf Adapterebene und ist dadurch flexibler, wenn der Datenaustausch über Adapter erfolgt.
- **Eigene Implementierung:** Alternativ könnte die Aufteilung auch durch manuelles Verkabeln mehrerer Adapterknoten erfolgen, was jedoch die Übersichtlichkeit und Wartbarkeit verschlechtert.
- `AR_SPLIT_6_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- `AR_SPLIT_6`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

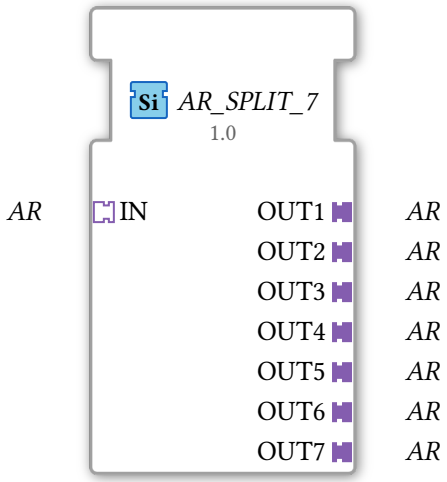
Fazit

Der **AR_SPLIT_6_UNGATED** ist ein einfacher, aber nützlicher generischer Baustein zur Vervielfachung eines AR-Adapter-Signals auf sechs parallele Ausgänge. Er zeichnet sich durch minimale Komplexität, klare Struktur und hohe Wiederverwendbarkeit in Automatisierungsprojekten aus, die auf der 4diac-IDE und dem IEC 61499-Standard basieren.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AR_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AR_SPLIT_7** ist ein generischer Funktionsblock, der einen eingehenden AR-Adapter-Socket (vom Typ `adapter::types::unidirectional::AR`) auf sieben separate AR-Adapter-Plugs verteilt. Er dient dazu, ein AR-Signal an bis zu sieben verschiedene Empfänger weiterzuleiten, ohne dass die Daten mehrfach bereitgestellt werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AR`	IN	Socket	Eingehender AR-Adapter, der auf die sieben Ausgänge verteilt wird.
`adapter::types::unidirectional::AR`	OUT1 … OUT7	Plug	Sieben Ausgabeadapter, an die der eingehende AR-Adapter unverändert weitergegeben wird.

Funktionsweise

Der FB empfängt über den Socket `IN` einen AR-Adapter. Sobald eine Verbindung zu diesem Socket hergestellt ist, leitet der Baustein den kompletten Adapter – einschließlich aller darin enthaltenen Daten und Ereignisse – an alle sieben Plug-Adapter (`OUT1` bis `OUT7`) weiter. Es findet keine Transformation, Filterung oder Pufferung der Daten statt; der AR-Adapter wird **1:1 auf alle Ausgänge gespiegelt**.

Der FB ist als **generischer Baustein** realisiert (siehe Attribut `eclipse4diac::core::GenericClassName = 'GEN_AR_SPLIT'`), sodass er unabhängig von der konkreten Parametrierung des AR-Adapters arbeitet.

Technische Besonderheiten

- **Generische Ausführung** – Der FB benötigt keine spezifische Typinformationen des AR-Adapters; er arbeitet rein über die Adapterschnittstelle.
- **Kein Zustandsautomat** – Der FB besitzt keinen ECC (Execution Control Chart) und ist rein datenflusstechnisch; er reagiert direkt auf Adapterverbindungen.
- **Copyright & Lizenz** – Der Baustein unterliegt der Eclipse Public License 2.0 (EPL-2.0) und wurde von HR Agrartechnik GmbH bereitgestellt (Version 1.0, 2025-01-24).

Zustandsübersicht

Der FB verfügt über keine Zustandsmaschine (ECC). Die Verteilung erfolgt statisch und ohne Laufzeitlogik.

Anwendungsszenarien

- **Signalverteilung in Automatisierungssystemen** – Ein Sensor- oder Steueradapter (AR) muss mehrere nachgelagerte Bausteine parallel versorgen.
- **Monitor- oder Diagnoseanbindung** – Ein AR-Signal wird gleichzeitig an einen Steuerungs- und einen Überwachungszweig weitergegeben.
- **Test- und Simulationsumgebungen** – Ein Adapter wird an mehrere Test- oder Mock-Bausteine verteilt, ohne die Quelle mehrfach zu instanziieren.

Vergleich mit ähnlichen Bausteinen

- **AR_SPLIT_2, AR_SPLIT_3, ...** – Diese Bausteine bieten die gleiche Funktionalität mit einer unterschiedlichen Anzahl von Ausgängen (2, 3, ...). Der `AR_SPLIT_7` ist die Variante mit genau sieben Ausgängen.
- **AR_MERGE** – Ein Zusammenführen mehrerer AR-Adapter auf einen, also die inverse Operation.
- **AR_COPY** – Oft als dedizierter FB für eine einzelne 1:1-Verteilung, während `AR_SPLIT_7` mehrere Ausgänge auf einmal bedient.
- `AR_SPLIT_7_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der **AR_SPLIT_7** ist ein schlanker, generischer Funktionsblock zur einfachen Verteilung eines AR-Adapters auf bis zu sieben Zieladapter. Dank seiner generischen Natur ist er ohne Anpassung der Typinformationen sofort einsetzbar und eignet sich besonders für lose gekoppelte, datenflussorientierte Architekturen in der Automatisierungstechnik.

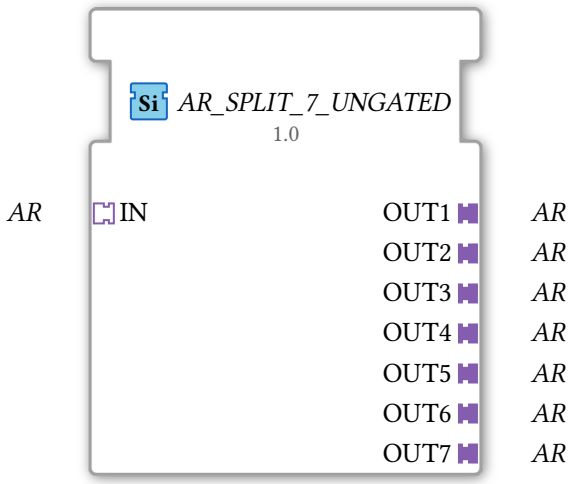
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AR_SPLIT_7_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AR_SPLIT_7`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht

gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AR_SPLIT_7_UNGATED** ist ein generischer Funktionsblock, der einen eingehenden AR-Adapter-Socket (vom Typ `adapter::types::unidirectional::AR`) auf sieben separate AR-Adapter-Plugs verteilt. Er dient dazu, ein AR-Signal an bis zu sieben verschiedene Empfänger weiterzuleiten, ohne dass die Daten mehrfach bereitgestellt werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AR`	IN	Socket	Eingehender AR-Adapter, der auf die sieben Ausgänge verteilt wird.
`adapter::types::unidirectional::AR`	OUT1 … OUT7	Plug	Sieben Ausgabeadapter, an die der eingehende AR-Adapter unverändert weitergegeben wird.

Funktionsweise

Der FB empfängt über den Socket `IN` einen AR-Adapter. Sobald eine Verbindung zu diesem Socket hergestellt ist, leitet der Baustein den kompletten Adapter – einschließlich aller darin enthaltenen Daten und Ereignisse – an alle sieben Plug-Adapter (`OUT1` bis `OUT7`) weiter. Es findet keine Transformation, Filterung oder Pufferung der Daten statt; der AR-Adapter wird **1:1 auf alle Ausgänge gespiegelt**.

Der FB ist als **generischer Baustein** realisiert (siehe Attribut `eclipse4diac::core::GenericClassName = 'GEN_AR_SPLIT'`), sodass er unabhängig von der konkreten Parametrierung des AR-Adapters arbeitet.

Technische Besonderheiten

- **Generische Ausführung** – Der FB benötigt keine spezifische Typinformationen des AR-Adapters; er arbeitet rein über die Adapterschnittstelle.
- **Kein Zustandsautomat** – Der FB besitzt keinen ECC (Execution Control Chart) und ist rein datenflusstechnisch; er reagiert direkt auf Adapterverbindungen.
- **Copyright & Lizenz** – Der Baustein unterliegt der Eclipse Public License 2.0 (EPL-2.0) und wurde von HR Agrartechnik GmbH bereitgestellt (Version 1.0, 2025-01-24).

Zustandsübersicht

Der FB verfügt über keine Zustandsmaschine (ECC). Die Verteilung erfolgt statisch und ohne Laufzeitlogik.

Anwendungsszenarien

- **Signalverteilung in Automatisierungssystemen** – Ein Sensor- oder Steueradapter (AR) muss mehrere nachgelagerte Bausteine parallel versorgen.
- **Monitor- oder Diagnoseanbindung** – Ein AR-Signal wird gleichzeitig an einen Steuerungs- und einen Überwachungszweig weitergegeben.
- **Test- und Simulationsumgebungen** – Ein Adapter wird an mehrere Test- oder Mock-Bausteine verteilt, ohne die Quelle mehrfach zu instanziieren.

Vergleich mit ähnlichen Bausteinen

- **AR_SPLIT_2, AR_SPLIT_3, ...** – Diese Bausteine bieten die gleiche Funktionalität mit einer unterschiedlichen Anzahl von Ausgängen (2, 3, ...). Der `AR_SPLIT_7_UNGATED` ist die Variante mit genau sieben Ausgängen.
- **AR_MERGE** – Ein Zusammenführen mehrerer AR-Adapter auf einen, also die inverse Operation.
- **AR_COPY** – Oft als dedizierter FB für eine einzelne 1:1-Verteilung, während `AR_SPLIT_7_UNGATED` mehrere Ausgänge auf einmal bedient.
- `AR_SPLIT_7`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

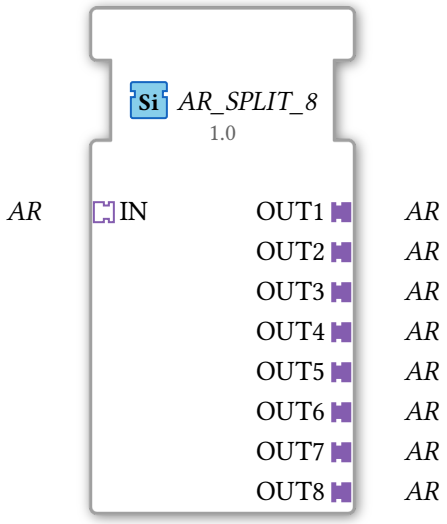
Fazit

Der **AR_SPLIT_7_UNGATED** ist ein schlanker, generischer Funktionsblock zur einfachen Verteilung eines AR-Adapters auf bis zu sieben Zieladapter. Dank seiner generischen Natur ist er ohne Anpassung der Typinformationen sofort einsetzbar und eignet sich besonders für lose gekoppelte, datenflussorientierte Architekturen in der Automatisierungstechnik.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AR_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AR_SPLIT_8 dient zur Aufteilung eines einzelnen unidirektionalen AR-Adapter-Signals auf acht identische AR-Ausgänge. Er ist als generischer Baustein implementiert und ermöglicht eine einfache Signalverteilung ohne zusätzliche Logik.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Bezeichnung	Typ	Richtung
IN	`adapter::types::unidirectional::AR`	Socket (Eingang)
OUT1 – OUT8	`adapter::types::unidirectional::AR`	Plug (Ausgang)

Funktionsweise

Der Baustein empfängt über den Socket **IN** die vollständige Schnittstelle eines unidirektionalen AR-Adapters. Die an dieser Schnittstelle anliegenden Ereignisse und Daten werden ohne Verzögerung oder Modifikation auf alle acht Ausgangsadapter (**OUT1** bis **OUT8**) repliziert. Somit kann ein einzelner Daten-/Ereignisstrom parallel an bis zu acht nachgeschaltete Komponenten verteilt werden.

Technische Besonderheiten

- Der Baustein ist generisch (generic FB) und wird durch das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `'GEN_AR_SPLIT'` als solcher gekennzeichnet.
- Es ist kein internes Zustandsdiagramm (ECC) vorhanden – die Weiterleitung erfolgt direkt und zu jedem Zeitpunkt zustandslos.

- Die Implementierung ist auf unidirektionale AR-Adapter ausgelegt; eine Verwendung mit bidirektionalen Adaptern ist nicht vorgesehen.

Zustandsübersicht

Der Funktionsblock besitzt kein explizites Zustandsdiagramm und arbeitet zustandslos. Die Adapterverteilung erfolgt kontinuierlich, ohne zeitliche Abhängigkeiten oder innere Logik.

Anwendungsszenarien

- **Verteilung von Steuerungssignalen**: Ein zentraler Steuerungsalgorithmus (z.B. ein PID-Regler) gibt seinen Ausgang über AR an mehrere parallele Aktoren weiter.
- **Broadcast in sternförmigen Anlagenteilen**: Signale von einer übergeordneten Steuerung werden an acht gleichartige Unterstationen verteilt.
- **Test- und Simulationsumgebungen**: Ein Signalgenerator speist mehrere Testobjekte gleichzeitig mit dem gleichen Adaptersignal.

Vergleich mit ähnlichen Bausteinen

- **AR_SPLIT_2, AR_SPLIT_4**: Analoge Bausteine mit 2 bzw. 4 Ausgängen; AR_SPLIT_8 erweitert die Anzahl auf acht Ausgänge.
- **AR_MERGE_X**: Führt mehrere AR-Signale zu einem zusammen – gegensätzliche Funktion.
- SPLIT_ *für andere Adaptertypen**: Es existieren Split-Bausteine für andere unidirektionale bzw. bidirektionale Adapterdefinitionen, die eine ähnliche Aufteilungslogik umsetzen.
- `AR_SPLIT_8_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

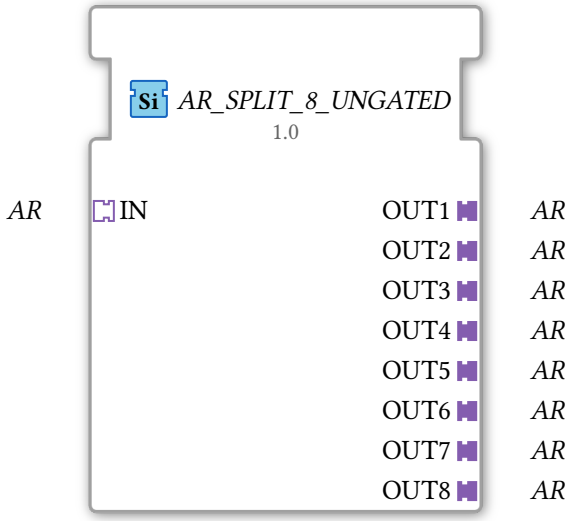
Der AR_SPLIT_8 ist ein einfacher, aber äußerst nützlicher Baustein zum Verteilen von unidirektionalen AR-Adaptersignalen auf bis zu acht parallele Pfade. Seine generische Implementierung ermöglicht den flexiblen Einsatz in verschiedenen Automatisierungsprojekten, ohne dass zusätzliche Logik zur Signalvervielfachung implementiert werden muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AR_SPLIT_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AR_SPLIT_8`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AR_SPLIT_8_UNGATED dient zur Aufteilung eines einzelnen unidirektionalen AR-Adapter-Signals auf acht identische AR-Ausgänge. Er ist als generischer Baustein implementiert und ermöglicht eine einfache Signalverteilung ohne zusätzliche Logik.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Bezeichnung	Typ	Richtung
IN	`adapter::types::unidirectional::AR`	Socket (Eingang)
OUT1 – OUT8	`adapter::types::unidirectional::AR`	Plug (Ausgang)

Funktionsweise

Der Baustein empfängt über den Socket **IN** die vollständige Schnittstelle eines unidirektionalen AR-Adapters. Die an dieser Schnittstelle anliegenden Ereignisse und Daten werden ohne Verzögerung oder Modifikation auf alle acht Ausgangsadapter (**OUT1** bis **OUT8**) repliziert. Somit kann ein einzelner Daten-/Ereignisstrom parallel an bis zu acht nachgeschaltete Komponenten verteilt werden.

Technische Besonderheiten

- Der Baustein ist generisch (generic FB) und wird durch das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `'GEN_AR_SPLIT'` als solcher gekennzeichnet.
- Es ist kein internes Zustandsdiagramm (ECC) vorhanden – die Weiterleitung erfolgt direkt und zu jedem Zeitpunkt zustandslos.
- Die Implementierung ist auf unidirektionale AR-Adapter ausgelegt; eine Verwendung mit bidirektionalen Adaptern ist nicht vorgesehen.

Zustandsübersicht

Der Funktionsblock besitzt kein explizites Zustandsdiagramm und arbeitet zustandslos. Die Adapterverteilung erfolgt kontinuierlich, ohne zeitliche Abhängigkeiten oder innere Logik.

Anwendungsszenarien

- **Verteilung von Steuerungssignalen**: Ein zentraler Steuerungsalgorithmus (z.B. ein PID-Regler) gibt seinen Ausgang über AR an mehrere parallele Aktoren weiter.
- **Broadcast in sternförmigen Anlagenteilen**: Signale von einer übergeordneten Steuerung werden an acht gleichartige Unterstationen verteilt.
- **Test- und Simulationsumgebungen**: Ein Signalgenerator speist mehrere Testobjekte gleichzeitig mit dem gleichen Adaptersignal.

Vergleich mit ähnlichen Bausteinen

- **AR_SPLIT_2, AR_SPLIT_4**: Analoge Bausteine mit 2 bzw. 4 Ausgängen; AR_SPLIT_8_UNGATED erweitert die Anzahl auf acht Ausgänge.
- **AR_MERGE_X**: Führt mehrere AR-Signale zu einem zusammen – gegensätzliche Funktion.
- SPLIT_ *für andere Adaptertypen**: Es existieren Split-Bausteine für andere unidirektionale bzw. bidirektionale Adapterdefinitionen, die eine ähnliche Aufteilungslogik umsetzen.
- `AR_SPLIT_8`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

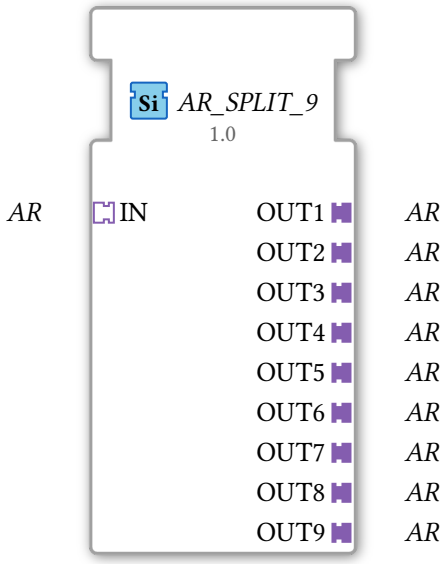
Fazit

Der AR_SPLIT_8_UNGATED ist ein einfacher, aber äußerst nützlicher Baustein zum Verteilen von unidirektionalen AR-Adaptersignalen auf bis zu acht parallele Pfade. Seine generische Implementierung ermöglicht den flexiblen Einsatz in verschiedenen Automatisierungsprojekten, ohne dass zusätzliche Logik zur Signalvervielfachung implementiert werden muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AR_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AR_SPLIT_9** teilt einen eingehenden AR-Adapter (Typ `unidirectional`) in neun separate AR-Ausgänge auf. Er dient als Verteiler für ein AR-Signal an mehrere nachgeschaltete Komponenten. Der Baustein ist vollständig generisch ausgelegt.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Richtung	Adapter	Typ	Beschreibung
Socket (Eingang)	`IN`	`adapter::types::unidirectional::AR`	Eingehendes AR-Signal, das verteilt wird.
Plug (Ausgang)	`OUT1` … `OUT9`	`adapter::types::unidirectional::AR`	Neun identische Ausgänge, die das eingehende AR-Signal weiterleiten.

Funktionsweise

Der Baustein leitet das am **IN**-Socket anliegende AR-Signal unverändert an alle neun Ausgänge **OUT1** bis **OUT9** weiter. Es findet keine Verzögerung, Filterung oder Umwandlung statt. Die Signalausbreitung erfolgt rein kombinatorisch.

Technische Besonderheiten

- **Generischer Baustein**: Der FB verwendet ein generisches Class-Name-Attribut (`eclipse4diac::core::GenericClassName` mit dem Wert `'GEN_AR_SPLIT'`) und ein Typ-Hash-Attribut, um eine flexible Typanpassung im IEC 61499-Modell zu ermöglichen.
- **Unidirektionale Adapter**: Sowohl der Eingang als auch alle Ausgänge sind als unidirektionale AR-Adapter definiert. Eine Rückkopplung oder bidirektionale Kommunikation ist nicht vorgesehen.
- **Keine Ereignisse oder Daten**: Die gesamte Logik erfolgt ausschließlich über Adapter-Schnittstellen – Ereignis- oder Daten-Ein-/Ausgänge sind nicht vorhanden.

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand (keine Zustandsmaschine). Das Ausgangssignal entspricht zu jedem Zeitpunkt dem aktuellen Eingangssignal. Es gibt keine Initialisierungs- oder Fehlerzustände.

Anwendungsszenarien

- **Verteilung eines Start-/Stopp-Signals**: Ein zentraler AR-Befehl (z. B. „Start“) wird auf mehrere Aktoren oder Maschinenteile verteilt.
- **Multicast-Kopplung**: Ein AR-Signal, das von einer übergeordneten Steuerung kommt, wird gleichzeitig an mehrere untergeordnete Module gesendet.
- **Parallele Ansteuerung** von neun identischen oder unterschiedlichen Komponenten, die dasselbe AR-Protokoll verwenden.

Vergleich mit ähnlichen Bausteinen

Baustein	Anzahl Ausgänge	Bemerkung

AR_SPLIT_4	4	Gleiche Funktionsweise, aber nur vier Ausgänge.
AR_SPLIT_9	9	Vorliegender Baustein (neun Ausgänge).
AR_SPLIT_2	2	Minimaler Splitter für zwei Pfade.

Die Bausteine unterscheiden sich lediglich in der Anzahl der Ausgänge; die Funktionsweise und die generische Eigenschaft sind identisch.

- **AR_SPLIT_9_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

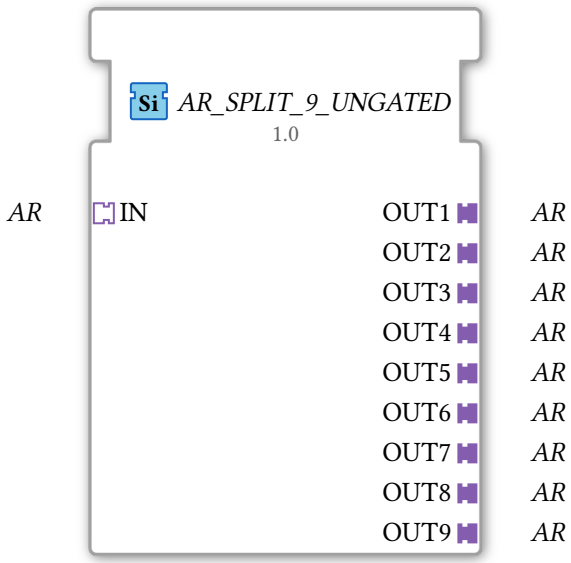
Der **AR_SPLIT_9** ist ein einfacher, aber hilfreicher Verteilerbaustein für unidirektionale AR-Signale. Aufgrund seiner generischen Auslegung kann er flexibel in unterschiedlichen Steuerungsarchitekturen eingesetzt werden, ohne dass eine Typanpassung von Hand erforderlich ist. Er eignet sich besonders für Szenarien, in denen ein Signal auf viele Pfade gleichzeitig dupliziert werden muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AR_SPLIT_9_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AR_SPLIT_9. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AR_SPLIT_9_UNGATED** teilt einen eingehenden AR-Adapter (Typ `unidirectional`) in neun separate AR-Ausgänge auf. Er dient als Verteiler für ein AR-Signal an mehrere nachgeschaltete Komponenten. Der Baustein ist vollständig generisch ausgelegt.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Richtung	Adapter	Typ	Beschreibung
Socket (Eingang)	`IN`	`adapter::types::unidirectional::AR`	Eingehendes AR-Signal, das verteilt wird.
Plug (Ausgang)	`OUT1` … `OUT9`	`adapter::types::unidirectional::AR`	Neun identische Ausgänge, die das eingehende AR-Signal weiterleiten.

Funktionsweise

Der Baustein leitet das am **IN**-Socket anliegende AR-Signal unverändert an alle neun Ausgänge **OUT1** bis **OUT9** weiter. Es findet keine Verzögerung, Filterung oder Umwandlung statt. Die Signalausbreitung erfolgt rein kombinatorisch.

Technische Besonderheiten

- **Generischer Baustein**: Der FB verwendet ein generisches Class-Name-Attribut (`eclipse4diac::core::GenericClassName` mit dem Wert `'GEN_AR_SPLIT'`) und ein Typ-Hash-Attribut, um eine flexible Typanpassung im IEC 61499-Modell zu ermöglichen.
- **Unidirektionale Adapter**: Sowohl der Eingang als auch alle Ausgänge sind als unidirektionale AR-Adapter definiert. Eine Rückkopplung oder bidirektionale Kommunikation ist nicht vorgesehen.
- **Keine Ereignisse oder Daten**: Die gesamte Logik erfolgt ausschließlich über Adapter-Schnittstellen – Ereignis- oder Daten-Ein-/Ausgänge sind nicht vorhanden.

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand (keine Zustandsmaschine). Das Ausgangssignal entspricht zu jedem Zeitpunkt dem aktuellen Eingangssignal. Es gibt keine Initialisierungs- oder Fehlerzustände.

Anwendungsszenarien

- **Verteilung eines Start-/Stopp-Signals**: Ein zentraler AR-Befehl (z. B. „Start“) wird auf mehrere Aktoren oder Maschinenteile verteilt.
- **Multicast-Kopplung**: Ein AR-Signal, das von einer übergeordneten Steuerung kommt, wird gleichzeitig an mehrere untergeordnete Module gesendet.
- **Parallele Ansteuerung** von neun identischen oder unterschiedlichen Komponenten, die dasselbe AR-Protokoll verwenden.

Vergleich mit ähnlichen Bausteinen

Baustein	Anzahl Ausgänge	Bemerkung

AR_SPLIT_4	4	Gleiche Funktionsweise, aber nur vier Ausgänge.
AR_SPLIT_9_UNGATED	9	Vorliegender Baustein (neun Ausgänge).
AR_SPLIT_2	2	Minimaler Splitter für zwei Pfade.

Die Bausteine unterscheiden sich lediglich in der Anzahl der Ausgänge; die Funktionsweise und die generische Eigenschaft sind identisch.

- **AR_SPLIT_9**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

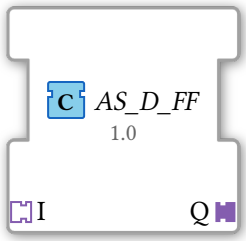
Der **AR_SPLIT_9_UNGATED** ist ein einfacher, aber hilfreicher Verteilerbaustein für unidirektionale AR-Signale. Aufgrund seiner generischen Auslegung kann er flexibel in unterschiedlichen Steuerungsarchitekturen eingesetzt werden, ohne dass eine Typanpassung von Hand erforderlich ist. Er eignet sich besonders für Szenarien, in denen ein Signal auf viele Pfade gleichzeitig dupliziert werden muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

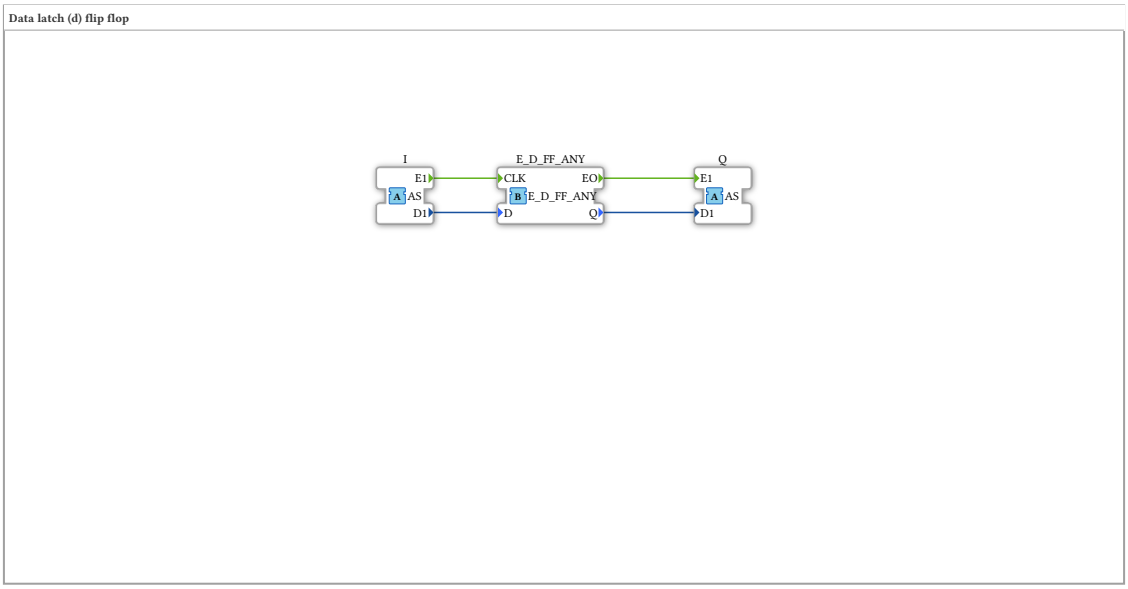
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SINT

AS_D_FF



Value to latch – *AS* *AS* – Latched value



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **AS_D_FF** realisiert ein einfaches Datenlatch (D-Flipflop) auf Basis von Adaptern. Er speichert einen eingehenden Datenwert bei jedem eingehenden Ereignis und gibt den gespeicherten Wert als Ausgangssignal weiter. Der Baustein kapselt die Logik des 4diac-Standardbausteins `E_D_FF_ANY` und stellt seine Schnittstellen ausschließlich über unidirektionale Adapter vom Typ `adapter::types::unidirectional::AS` bereit. Dies ermöglicht eine modulare und standardisierte Einbindung in komplexere Systeme.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereigniseingänge. Der Baustein empfängt Ereignisse ausschließlich über den Adapter **I** (Socket). Das Ereignis wird am Adapter-Eingang `I.E1` bereitgestellt.

Ereignis-Ausgänge

Keine direkten Ereignisausgänge. Der Baustein sendet Ereignisse ausschließlich über den Adapter **Q** (Plug). Das ausgegebene Ereignis erscheint am Adapter-Ausgang `Q.E1`.

Daten-Eingänge

Keine direkten Dateneingänge. Der zu speichernde Datenwert wird über den Adapter **I** (Socket) als `I.D1` bereitgestellt.

Daten-Ausgänge

Keine direkten Datenausgänge. Der gespeicherte Datenwert wird über den Adapter **Q** (Plug) als `Q.D1` ausgegeben.

Adapter

Adapter	Richtung	Typ	Beschreibung
I	Socket (Eingang)	`adapter::types::unidirectional::AS`	Stellt das Eingangsereignis (`E1`) und den zu latchenden Datenwert (`D1`) bereit.
Q	Plug (Ausgang)	`adapter::types::unidirectional::AS`	Gibt ein Ausgangsereignis (`E1`) und den gespeicherten Datenwert (`D1`) aus.

Funktionsweise

Der Baustein **AS_D_FF** arbeitet als asynchroner Datenlatch. Bei jedem positiven Ereignis auf `I.E1` wird der aktuelle Wert von `I.D1` im internen Speicher des Bausteins übernommen. Unmittelbar danach wird ein Ereignis auf `Q.E1` ausgelöst, und der gespeicherte Wert steht auf `Q.D1` zur Verfügung. Diese Logik wird durch den intern verwendeten Baustein `E_D_FF_ANY` realisiert:

- `I.E1` → `E_D_FF_ANY.CLK`
- `I.D1` → `E_D_FF_ANY.D`
- `E_D_FF_ANY.EO` → `Q.E1`
- `E_D_FF_ANY.Q` → `Q.D1`

Der Baustein verhält sich somit wie ein klassisches D-Flipflop, jedoch mit einer standardisierten Adapter-Schnittstelle anstelle von direkten Ein-/Ausgängen.

Technische Besonderheiten

- **Adapter-basierte Kapselung**: Der gesamte Datenaustausch erfolgt ausschließlich über unidirektionale Adapter vom Typ `adapter::types::unidirectional::AS`. Dies erleichtert die Wiederverwendung in unterschiedlichen Kontexten und gewährleistet eine klare Trennung zwischen Ereignis- und Datenflüssen.
- **Interne Nutzung von `E_D_FF_ANY`**: Der Baustein greift auf den Standard-Funktionsbaustein `iec61499::events::E_D_FF_ANY` zurück, der eine bewährte und effiziente Implementierung eines D-Flipflops bietet.

- **Keine direkten E/A-Pins**: Der Baustein definiert selbst keine Ereignis- oder Dateneingänge/-ausgänge; die gesamte Kommunikation wird über die Adapter abgewickelt.
- **Unidirektionale Adapter**: Beide Adapter (I und Q) sind als unidirektional definiert, d.h. Ereignis- und Datenfluss erfolgen nur in eine Richtung (vom Socket zum Plug).

Zustandsübersicht

Der Baustein besitzt keine explizite Zustandsmaschine. Sein Verhalten wird durch das interne D-Flipflop bestimmt, das zwei logische Zustände kennt:

- **Zustand 0**: Der gespeicherte Datenwert ist 0 (bzw. `FALSE`).
- **Zustand 1**: Der gespeicherte Datenwert ist 1 (bzw. `TRUE`).

Der Zustand wird durch ein Ereignis auf `I.E1` umgeschaltet (sofern sich der Datenwert geändert hat) und bleibt bis zum nächsten Ereignis erhalten.

Anwendungsszenarien

- **Signalpufferung**: Zwischenspeichern eines digitalen Signals, das zu bestimmten Zeitpunkten übernommen werden soll (z.B. Sample-and-Hold).
- **Synchronisation**: Anpassen von asynchronen Datenströmen an einen taktgesteuerten Ablauf, indem ein Wert bei einem Ereignis festgehalten wird.
- **Modulare Verschaltung**: Einsatz in Systemen, die auf Adapter-Kommunikation basieren, um eine einheitliche Schnittstelle beizubehalten (z.B. in einer Kette von Verarbeitungsblöcken).
- **Einfache Zustandsspeicherung**: Realisierung von einfachen Speicherfunktionen in Automatisierungsanwendungen, ohne auf komplexe Speicherbausteine zurückgreifen zu müssen.

Vergleich mit ähnlichen Bausteinen

Baustein	Eigenschaft	Unterschied zu AS_D_FF
SR_FF	Set-Reset-Flipflop	Hat separate Set- und Reset-Eingänge, kein Taktsignal.
E_D_FF_ANY	Reines D-Flipflop mit direkten E/A	Bietet keine Adapter-Schnittstelle; erfordert direkte Verbindungen.
AS_FF_RS (hypothetisch)	RS-Flipflop mit Adaptern	Verwendet zwei Adapter für Set und Reset, kein Taktsignal.

Der **AS_D_FF** kombiniert die klare Logik eines D-Flipflops mit der Flexibilität einer Adapter-basierten Schnittstelle. Er eignet sich besonders für Systeme, die eine standardisierte, unidirektionale Kommunikation bevorzugen.

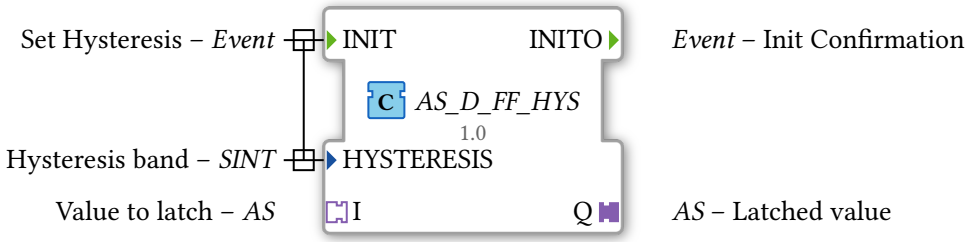
Fazit

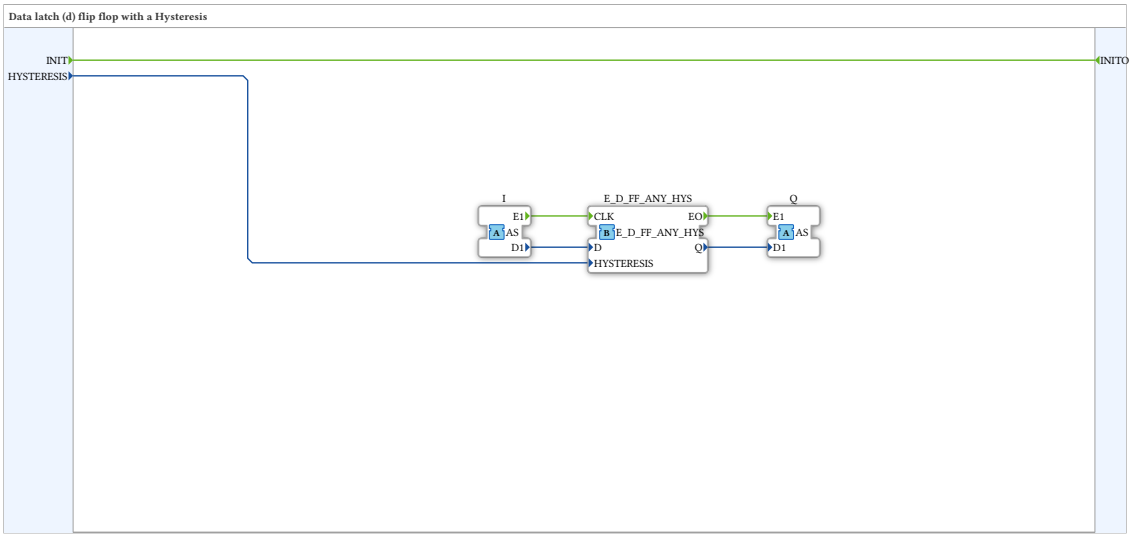
Der Funktionsbaustein **AS_D_FF** stellt eine praktische Kapselung eines D-Flipflops dar, die ausschließlich über Adapter kommuniziert. Sein internes Verhalten entspricht einem zuverlässigen Datenlatch, das auf jedes eingehende Ereignis reagiert. Die Verwendung von Adaptern erhöht die Modularität und Wiederverwendbarkeit, während die Implementation auf dem bewährten `E_D_FF_ANY` basiert. Der Baustein eignet sich ideal für alle Anwendungen, die eine einfache, ereignisgesteuerte Datenspeicherung mit standardisierter Schnittstelle erfordern.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AS_D_FF_HYS





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AS_D_FF_HYS** realisiert ein datengetaktetes Flip-Flop (D-Latch) mit einer **Hysterese** (Schalthysterese). Der Baustein wird typischerweise in der Signalverarbeitung eingesetzt, um verrauschte oder schwankende Eingangssignale zu glätten und definierte Schaltpunkte zu erzeugen. Er verfügt über einen Ereigniseingang `INIT`, um den Hysterese-Wert zu setzen, sowie über zwei Adapter-Schnittstellen: einen Eingangs-Adapter `I` für das zu latchnde Signal und einen Ausgangs-Adapter `Q` für das gelatchte und mit Hysterese gefilterte Ausgangssignal.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar	Mit Daten
`INIT`	EInit	Setzt den Hysterese-Wert	`HYSTERESIS`

Ereignis-Ausgänge

Name	Typ	Kommentar
`INITO`	EInit	Bestätigung der INIT-Verarbeitung

Daten-Eingänge

Name	Typ	Kommentar
`HYSTERESIS`	SINT	Hysterese-Bandbreite (ganzzahliger Wert)

Daten-Ausgänge

Keine direkten Datenausgänge – der Ausgangswert wird über den Adapter `Q` bereitgestellt.

Adapter

Richtung	Name	Typ	Kommentar
Socket	`I`	`adapter::types::unidirectional::AS`	Eingangssignal (Wert und Takt)
Plug	`Q`	`adapter::types::unidirectional::AS`	Ausgangssignal (gelatcht mit Hysterese)

Die Adapter vom Typ `AS` (unidirectional) besitzen typischerweise einen Ereignis-Port `E1` und einen Daten-Port `D1`. Der Socket `I` liefert über `I.E1` das Taktsignal und über `I.D1` den Datenwert, der gelatcht werden soll. Der Plug `Q` gibt über `Q.E1` ein Ereignis und über `Q.D1` den gefilterten Ausgangswert aus.

Funktionsweise

Der FB empfängt über den Adapter `I` ein Taktsignal (Ereignis `I.E1`) und einen Datenwert (`I.D1`). Bei jedem Takt wird der aktuelle Datenwert mit dem zuletzt ausgegebenen Wert verglichen, unter Berücksichtigung des Hysterese-Bandes. Der Ausgang `Q.D1` ändert sich nur dann, wenn der neue Eingangswert außerhalb des Intervalls `[letzter Ausgangswert - HYSTERESIS , letzter Ausgangswert + HYSTERESIS]` liegt. Dadurch werden kleine Schwankungen (Rauschen) um den eingestellten Arbeitspunkt unterdrückt.

Das Ereignis `INIT` dient zur einmaligen Konfiguration des Hysterese-Werts. Der Hysterese-Wert vom Dateneingang `HYSTERESIS` wird dabei übernommen. Nach erfolgreicher Initialisierung wird das Ereignis `INITO` ausgelöst.

Der FB nutzt intern den Baustein `E_D_FF_ANY_HYS` aus der Bibliothek `logiBUS::signalprocessing::hysteresis`, der die eigentliche Flip-Flop-Logik mit Hysterese implementiert.

Technische Besonderheiten

- Der Hysterese-Wert ist als `SINT` (signed integer) festgelegt.
- Die eigentliche Signalverarbeitung wird durch einen untergeordneten FB (`E_D_FF_ANY_HYS`) realisiert, der vom Anwender nicht direkt eingesehen wird.
- Die Adapter-Schnittstelle erlaubt eine lose Kopplung des Bausteins an andere Komponenten.
- Das Ereignis `INIT` verarbeitet keinen Wert – die Hystereseparameter können nur vor dem ersten Betrieb gesetzt werden; ein nachträgliches Ändern ist nur durch erneutes Senden von `INIT` möglich.

Zustandsübersicht

Der FB besitzt **keine explizite Zustandsmaschine**, die nach außen sichtbar ist. Intern wird der aktuelle Ausgangswert gespeichert. Die Hysterese bewirkt ein Zustandsgedächtnis: Der Ausgang springt nicht bei jeder Änderung des Eingangs, sondern nur bei Über- oder Unterschreiten der Hysterese-Schwellen.

Anwendungsszenarien

- **Entprellung von Schaltkontakten** – Unterdrückung von Prellern und kurzen Störpulsen.
- **Schwellwertschalter mit Hysterese** – z. B. Temperaturregler, die Ein- und Ausschaltpunkte trennen.
- **Signalglättung in der Sensorik** – Verarbeitung verrauschter Analogwerte in diskrete Zustände.
- **Daten-Latch mit Rauschunterdrückung** – für langsame, aber störungsbehaftete Datensignale.

Vergleich mit ähnlichen Bausteinen

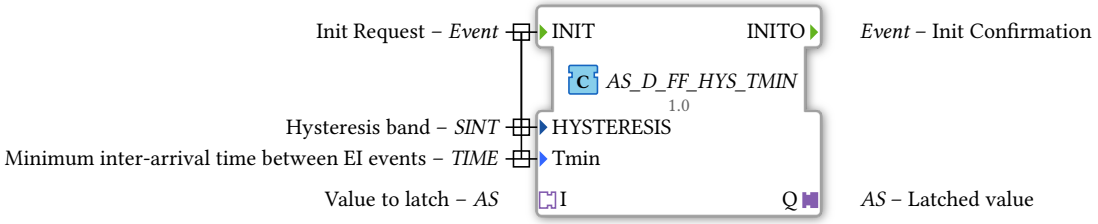
Baustein	Eigenschaft	Unterschied zu AS_D_FF_HYS
`E_D_FF`	Klassisches D-Flip-Flop ohne Hysterese	Keine Filterung; Ausgang folgt jedem Takt.
`E_D_FF_HYS`	D-Flip-Flop mit Hysterese (gleiche Logik)	Kein eigener `INIT`-Eingang; Hysterese wird ggf. über Daten gesetzt.
`AS_FF_HYS`	Flip-Flop mit Hysterese und eigener Adapter-Schnittstelle	Möglicherweise andere Kopplung, ähnliche Funktion.

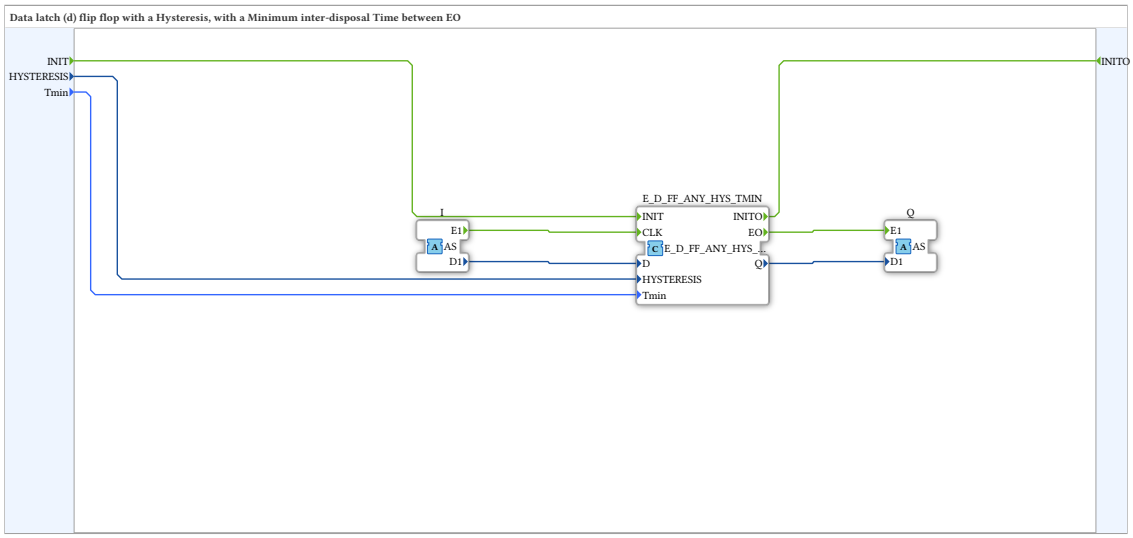
Fazit

Der `AS_D_FF_HYS` kombiniert ein D-Flip-Flop mit einer einstellbaren Hysterese und kapselt die komplexe Logik in einem einfach zu verwendenden Adapter-basierten Baustein. Er eignet sich besonders für Anwendungen, in denen verrauschte diskrete Signale sauber und mit definierten Schaltpunkten verarbeitet

werden müssen. Die initiale Konfiguration über `INIT` ermöglicht eine flexible Anpassung an verschiedene Störumgebungen.

AS_D_FF_HYS_TMIN





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **AS_D_FF_HYS_TMIN** realisiert ein datengetriggertes Flipflop (D-Flipflop) mit Hysterese und einer minimalen Zeitverzögerung zwischen aufeinanderfolgenden Ereignissen. Er dient zur stabilen, entprellten Signalweitergabe in zeitkritischen Steuerungsabläufen. Der Baustein nutzt Adapter zur unidirektionalen Kommunikation, wobei ein Eingangssignal über den Socket **I** empfangen und das latchende Ausgangssignal über den Plug **Q** bereitgestellt wird.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Initialisierungsanfrage; setzt Hysterese und Mindestzeit

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Bestätigung der abgeschlossenen Initialisierung

Daten-Eingänge

Name	Typ	Kommentar
HYSTERESIS	SINT	Hysterese-Band (ganzzahliger Wert, z. B. 8 Bit)
Tmin	TIME	Minimale Zeit zwischen zwei Ereignissen (z. B. für Entprellung)

Daten-Ausgänge

Keine direkten Datenausgänge – das Ausgangssignal wird über den Adapter **Q** bereitgestellt.

Adapter

Typ	Bezeichnung	Richtung	Kommentar
AS	I (Socket)	Eingang	Unidirektionaler Adapter für das zu latchnde Signal (z. B. Daten- + Ereignisleitung)
AS	Q (Plug)	Ausgang	Unidirektionaler Adapter für das latchende Ausgangssignal

Funktionsweise

Der Baustein arbeitet intern mit einem vordefinierten Funktionsbaustein **E_D_FF_ANY_HYS_TMIN**. Beim Ereignis **INIT** werden die Parameter **HYSTERESIS** und **Tmin** übernommen. Das eingehende Ereignis (über den Adapter **I**, Leitung **E1**) triggert den Flipflop-Clock, während der zugehörige Datenwert (über **I.D1**) übernommen wird. Der Ausgang **Q.D1** wird nur dann aktualisiert, wenn

- der neue Datenwert um mindestens die **Hysterese** vom aktuell gespeicherten Wert abweicht, und
- seit dem letzten Ereignis mindestens die Zeit **Tmin** vergangen ist.

Das Ausgangsereignis (über **Q.E1**) signalisiert eine gültige Aktualisierung. Der Baustein unterdrückt somit Rauschen und verhindert zu schnelle Flankenwechsel.

Technische Besonderheiten

- **Adapterbasierte Ein-/Ausgabe** – Die Signalübertragung über die Adapter **I** und **Q** ermöglicht eine einfache Integration in komplexe Adapter-Netzwerke und erhöht die Wiederverwendbarkeit.
- **Hysterese** – Vermeidet Oszillationen bei schwankenden oder verrauschten Eingangssignalen, indem nur signifikante Änderungen übernommen werden.
- **Minimale Ereigniszeit (Tmin)** – Erzwingt eine Mindestpause zwischen zwei Triggerungen; dient als zusätzlicher Entprellmechanismus.
- **Interne Nutzung eines bewährten Flipflop-Bausteins** – Die Implementierung delegiert die Kernlogik an **E_D_FF_ANY_HYS_TMIN**, der für generische Datentypen ausgelegt ist.

Zustandsübersicht

Der Baustein kennt keine expliziten, extern sichtbaren Zustände. Intern führt das Flipflop einen gespeicherten Wert (letzter gültiger Ausgang), der nur bei Erfüllung der Hysterese- und Zeitbedingungen aktualisiert wird. Nach der Initialisierung (INIT/INITO) ist der gespeicherte Wert undefiniert, bis das erste gültige Signal eintrifft.

Anwendungsszenarien

- **Entprellung von Sensorsignalen** – z. B. Endschalter oder Näherungssensoren mit mechanischem Prellen.
- **Frequenzbegrenzung** – Verhindern zu schneller Schaltvorgänge in nachgeschalteten Aktoren oder Kommunikationsschnittstellen.
- **Hysteresebasierte Schwellwerterkennung** – z. B. Temperatur- oder Druckregelung mit Rücksetzband.
- **Signalweitergabe in zeitkritischen, störungsbehafteten Umgebungen** (z. B. Landmaschinen-Steuerungen).

Vergleich mit ähnlichen Bausteinen

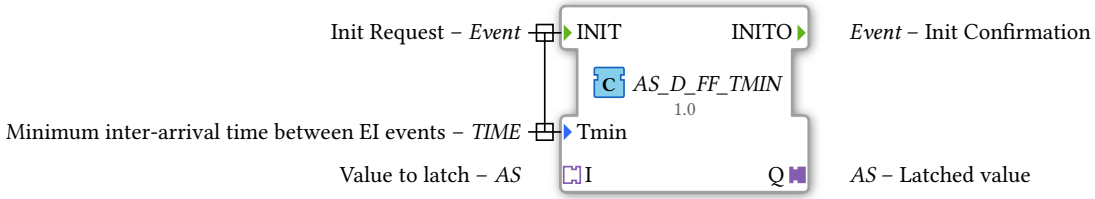
- **Einfaches D-Flipflop (z. B. E_D_FF)** – ohne Hysterese oder Zeitverzögerung; übernimmt jeden Takt ohne Filterung.
- **Schmitt-Trigger-Baustein** – ebenfalls mit Hysterese, jedoch meist ohne asynchrone Ereignissteuerung und ohne einstellbare Mindestzeit.
- **Verzögerungsbaustein (z. B. E_DELAY)** – verzögert nur das Signal, hat aber keine Hysterese oder Latch-Funktion.

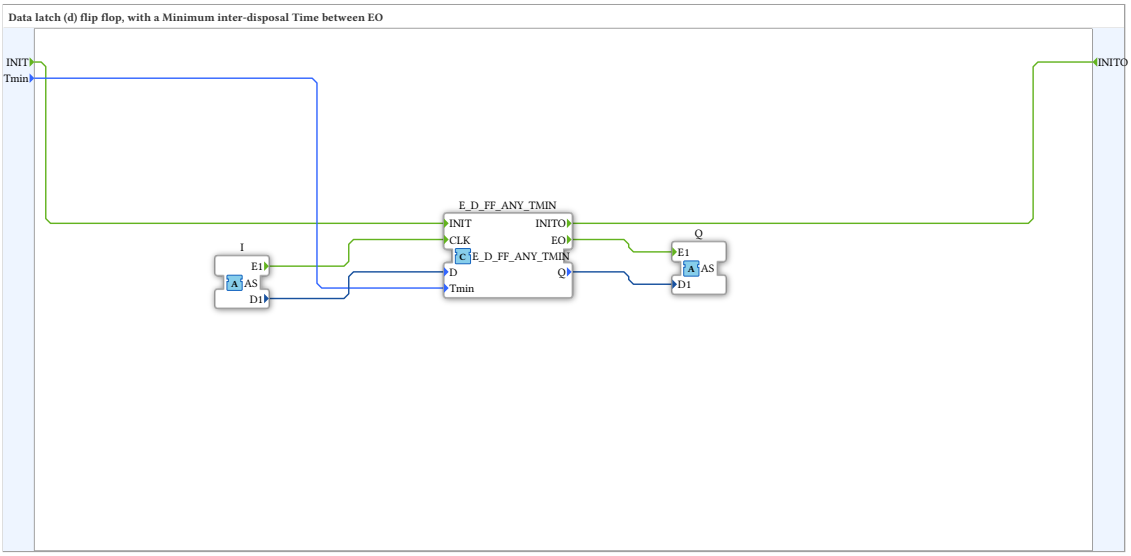
AS_D_FF_HYS_TMIN vereint Latch-Funktion, Hysterese und Zeitsteuerung in einem Baustein und ist daher speziell für robuste, zeitgesteuerte Signalverarbeitung optimiert.

Fazit

Der **AS_D_FF_HYS_TMIN** ist ein spezialisierter Funktionsbaustein für industrielle Steuerungen, die eine zuverlässige, rauschunempfindliche Signalweitergabe mit einstellbarer Totzeit erfordern. Die Kombination aus D-Flipflop, Hysterese und Minimalzeit macht ihn zu einem wertvollen Werkzeug für Automatisierungslösungen, insbesondere in Umgebungen mit hohen Störeinflüssen oder engen zeitlichen Anforderungen.

AS_D_FF_TMIN





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AS_D_FF_TMIN** realisiert ein datenverriegelndes D-Flipflop, das einen über einen Adapter zugeführten Datenwert bei Eintreffen eines Ereignisses übernimmt und über einen weiteren Adapter ausgibt. Die Besonderheit liegt in einer konfigurierbaren Mindestzeit (`Tmin`) zwischen zwei aufeinanderfolgenden Übernahmeereignissen. Dadurch werden zu schnelle Ereignisfolgen unterdrückt, was z. B. zur Entprellung oder zur Einhaltung von Prozessbeschränkungen genutzt werden kann.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar
INIT	EInit	Initialisierungsanforderung (liest `Tmin`)

Ereignis-Ausgänge

Ereignis	Typ	Kommentar
INITO	EInit	Bestätigung der Initialisierung

Daten-Eingänge

Name	Typ	Kommentar
Tmin	TIME	Minimale Zeit zwischen zwei aufeinanderfolgenden CLK-Ereignissen

Daten-Ausgänge

Der Block besitzt keine expliziten Datenausgänge; die Ausgabe erfolgt ausschließlich über den Adapter **Q**.

Adapter

Richtung	Adapter	Typ	Kommentar
Socket	I	adapter::types::unidirectional::AS	Eingangsdaten (D1) und Takt (E1)
Plug	Q	adapter::types::unidirectional::AS	Ausgangsdaten (D1) und Ereignis (E1)

Die Adapter vom Typ `AS` bieten je einen Ereignis-Ein-/Ausgang (E1) und einen Daten-Ein-/Ausgang (D1).

Funktionsweise

Nach einer erfolgreichen Initialisierung (INIT mit gültigem `Tmin`) arbeitet der Block wie ein getaktetes D-Flipflop mit Zeitfilter:

- Ein Ereignis am Adaptereingang **I.E1** wird als Takt (CLK) interpretiert.
- Der Datenwert **I.D1** wird beim Eintreffen von I.E1 übernommen, **sofern der zeitliche Abstand zum letzten CLK-Ereignis mindestens `Tmin` beträgt**. Andernfalls wird das Ereignis ignoriert.
- Bei erfolgreicher Übernahme wird der gespeicherte Wert am Datenausgang **Q.D1** ausgegeben und ein Ereignis **Q.E1** ausgelöst.

Die Zeitfilterung wird durch den intern verwendeten Baustein `E_D_FF_ANY_TMIN` realisiert, der das Taktereignis verzögert oder verwirft, falls die Mindestzeit nicht eingehalten wird.

Technische Besonderheiten

- **Adapter-basierte Ein-/Ausgabe**: Daten und Ereignisse werden nicht als separate Variablen, sondern über unidirektionale Adapter (`AS`) ausgetauscht. Das erlaubt eine modulare Verbindung mit anderen Blöcken.
- **Parameterisierbare Mindestzeit**: Die Zeit `Tmin` wird zur Initialisierung gesetzt und kann im laufenden Betrieb nicht geändert werden (ein erneuter INIT-Aufruf ist erforderlich).
- **Ereignis-Unterdrückung**: Zu dicht aufeinanderfolgende Taktereignisse werden stillschweigend ignoriert – es gibt keine Fehlermeldung.

Zustandsübersicht

Der Block besitzt keine expliziten Zustandsmaschinen im Sinne eines Statecharts. Das Verhalten lässt sich jedoch in zwei Phasen beschreiben:

1. **Initialisierungsphase**: Nach dem INIT-Ereignis wird `Tmin` intern gesetzt. Erst danach ist der Block bereit, CLK-Ereignisse zu verarbeiten.
2. **Betriebsphase**: Der Block wartet auf CLK-Ereignisse. Trifft eines ein, wird geprüft, ob die Zeitsperre (seit dem letzten CLK) abgelaufen ist. Wenn ja, wird der Datenwert übernommen und ausgegeben. Wenn nein, wird das Ereignis verworfen.

Anwendungsszenarien

- **Entprellung von digitalen Signalen**: Ein Sensor liefert schnelle, prellende Signale; mit `Tmin` wird die minimale Pulsbreite festgelegt.
- **Taktsynchronisation in zeitkritischen Systemen**: Vermeidung von Datenübernahmen schneller als eine vorgegebene Systemtaktgrenze.
- **Ratenbegrenzung für Kommunikationsschnittstellen**: Nur alle `Tmin` Sekunden wird ein neuer Wert akzeptiert (z. B. bei Bus-Zugriffen).

Vergleich mit ähnlichen Bausteinen

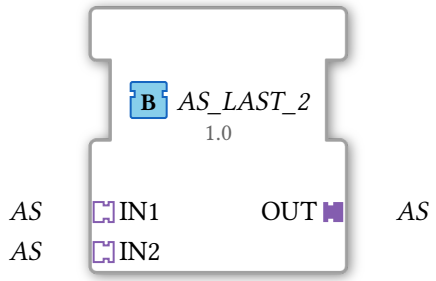
Baustein	Zeitfilterung	Zusätzliche Merkmale

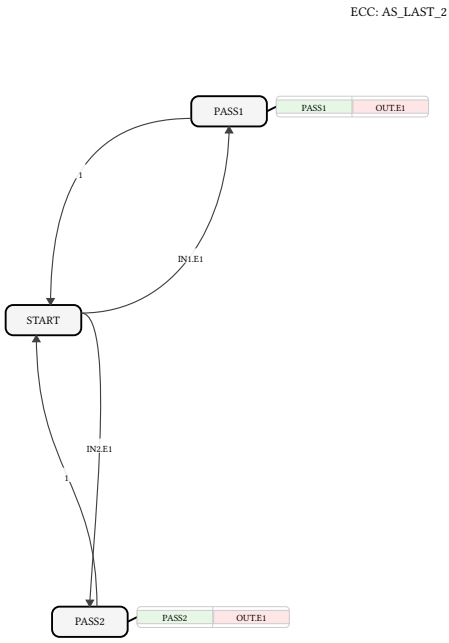
AS_D_FF	Nein (sofortige Übernahme)	Einfaches D-Flipflop ohne Zeitbeschränkung
AS_D_FF_TMIN (dieser)	Ja, konfigurierbar über `Tmin`	Unterdrückt zu schnelle Taktfolgen
AS_D_FF_TMAX	Ja, maximale Zeit zwischen Takten	Erzwingt regelmäßige Übernahmen

Fazit

Der Baustein **AS_D_FF_TMIN** erweitert ein einfaches D-Flipflop um eine einstellbare Mindestzeit zwischen zwei Übernahmeereignissen. Durch die Adapter-Schnittstelle und die klare Trennung von Initialisierung und Betrieb eignet er sich besonders für industrielle Steuerungen und zeitdiskrete Verarbeitungen, bei denen ein zu schnelles Schalten vermieden werden muss.

AS_LAST_2





Einleitung

Der AS_LAST_2 Funktionsblock führt zwei gleichartige SINT-Adapter-Signale (**IN1**, **IN2**) auf einen gemeinsamen Ausgang (**OUT**) zusammen, nach dem Prinzip **Last Writer Wins**: welcher der beiden Sockets zuletzt ein Ereignis geschrieben hat, gewinnt, und sein Datenwert wird sofort und unverändert an OUT durchgereicht. Es handelt sich ausdrücklich nicht um ein naives Zusammenführen (z. B. Mittelwert oder ODER-Verknüpfung), sondern um eine reine Arbitrierung nach zeitlicher Reihenfolge.

Schnittstellenstruktur

Adapter

Eingangsadapter (Sockets):

- **IN1**: SINT-Adapter (unidirectional) - erste Signalquelle
- **IN2**: SINT-Adapter (unidirectional) - zweite Signalquelle

Ausgangsadapter (Plug):

- **OUT**: SINT-Adapter (unidirectional) - zusammengeführtes Signal

Funktionsweise

Der Baustein ist als einfacher Basic-FB mit einer 3-Zustands-ECC umgesetzt (START, PASS1, PASS2). Trifft an IN1 ein Ereignis ein, wechselt die ECC nach PASS1: der aktuelle Datenwert von IN1 wird nach OUT kopiert und das Adapter-Ereignis an OUT gesendet. Trifft an IN2 ein Ereignis ein, geschieht dasselbe symmetrisch über PASS2. Nach jeder Durchreichung kehrt die ECC sofort wieder nach START zurück und ist bereit für das nächste Ereignis von IN1 oder IN2.

Technische Besonderheiten

- Last-Writer-Wins-Semantik statt naiver Datenverknüpfung
- Einfacher Basic-FB (ECC mit 3 Zuständen), kein zusätzlicher Zwischenspeicher nötig
- Keine Signalverzögerung: der Wert wird im selben Zyklus durchgereicht, in dem das auslösende Ereignis eintrifft
- Beide Eingänge sind gleichberechtigt - es gibt keine feste Priorität außer der zeitlichen Reihenfolge

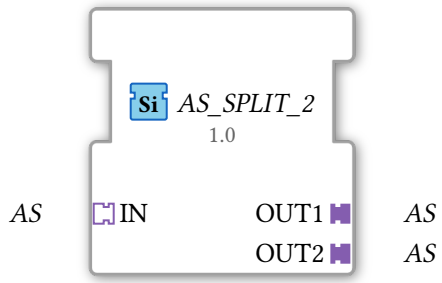
Anwendungsszenarien

- Zusammenführen zweier gleichwertiger SINT-Signalquellen zu einem einzigen nachgeschalteten Adapterpfad (z. B. zwei redundante Sensoren, oder ein manueller und ein automatischer Sollwertgeber)
- Arbitrierung zwischen zwei Bedienstellen, die denselben SINT-Wert schreiben dürfen
- Vereinfachung von Netzwerken, die sonst ein `F_SEL`/`E_RS`-Konstrukt inline verdrahten müssten

⚖️ Vergleich mit ähnlichen Bausteinen

AS_LAST_2 ist die Umkehrung von `AS_SPLIT_2`: AS_SPLIT_2 verteilt 1 Eingang auf 2 Ausgänge, AS_LAST_2 führt 2 Eingänge auf 1 Ausgang zusammen. Für Mischbetrieb zwischen einem Daten- und einem reinen Ereignisadapter (z. B. AX + AE) ist stattdessen `AX_AE_MERGE` gedacht, das nur das Ereignis mischt, ohne den Datenwert zu verändern.

AS_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AS_SPLIT_2** dient dem Aufteilen eines eingehenden Adapter-Signals vom Typ `adapter::types::unidirectional::AS` auf zwei identische Ausgangsadapter. Er ist als generischer Baustein konzipiert und eignet sich für Anwendungen, in denen ein Adapter-Datenstrom an mehrere nachfolgende Komponenten verteilt werden muss, ohne das Originalsignal zu verändern.

Schnittstellenstruktur

Ereignis-Eingänge
Keine vorhanden. Der Baustein arbeitet rein datengetrieben über die Adapter.

Ereignis-Ausgänge
Keine vorhanden.

Daten-Eingänge
Keine vorhanden. Die gesamte Datenübertragung erfolgt über den Adapter-Eingang `IN`.

Daten-Ausgänge
Keine vorhanden. Die Ausgangsdaten werden über die Adapter `OUT1` und `OUT2` bereitgestellt.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	`IN`	`adapter::types::unidirectional::AS`	Eingangsadapter, dessen Daten auf beide Ausgänge verteilt werden.
Plug	`OUT1`	`adapter::types::unidirectional::AS`	Erster Ausgangsadapter – erhält eine Kopie der eingehenden Daten.
Plug	`OUT2`	`adapter::types::unidirectional::AS`	Zweiter Ausgangsadapter – erhält ebenfalls eine Kopie der eingehenden Daten.

Funktionsweise
Der Baustein fungiert als passiver Verteiler: Ein am Socket `IN` anliegender Adapter-Datenstrom wird unverändert an beide Plugs `OUT1` und `OUT2` weitergeleitet. Es findet keinerlei Datenmanipulation, Pufferung oder Filterung statt. Die Aufteilung erfolgt signalgetreu, sodass die an `OUT1` und `OUT2` anliegenden Daten zeitlich und inhaltlich exakt den Daten von `IN` entsprechen.

Technische Besonderheiten
- **Generischer Typ:** Der Baustein ist als generischer Funktionsblock deklariert (`GenericClassName = GEN_AS_SPLIT`). Dies ermöglicht die Verwendung in verschiedenen Kontexten, sofern die angeschlossenen Adapter dem Typ `AS` entsprechen.
- **Keine Ereignissteuerung:** Der Datenfluss erfolgt rein über die Adapter-Schnittstellen. Es sind keine Ereignis-Ein- oder Ausgänge vorhanden, wodurch der Baustein keine eigenständige Ablaufsteuerung besitzt.
- **Keine Datenpuffer:** Da weder Speicher noch Verzögerung implementiert sind, eignet sich der Baustein besonders für Echtzeitanwendungen mit niedriger Latenz.

Zustandsübersicht
Der Baustein besitzt keine inneren Zustände oder sequentielle Logik. Er verhält sich wie eine reine Verdrahtungskomponente und arbeitet kontinuierlich ohne explizite Zustandsmaschine.

Anwendungsszenarien
- **Signalfan-out:** Verteilung eines Sensor-Adapters an mehrere parallel geschaltete Auswerteeinheiten.
- **Redundanz:** Einspeisung eines Steuersignals in zwei unabhängige Aktorstränge.
- **Monitoring:** Anschluss eines Analyse- oder Logging-Adapters parallel zum bestehenden Datenpfad.

Vergleich mit ähnlichen Bausteinen
Im Gegensatz zu einem einfachen Verbindungsknoten (der nur eine 1:1-Leitung realisiert) ermöglicht `AS_SPLIT_2` eine saubere, konfigurierbare Aufteilung auf zwei Ausgänge. Gegenüber einem **AS_MUX** oder **AS_DEMUX** fehlen diesem Baustein Auswahl- oder Priorisierungslogik – er verteilt immer alle eingehenden Daten an alle Ausgänge. Ähnliche Blöcke wie `AS_SPLIT_3` oder `AS_SPLIT_N` erweitern die Anzahl der Ausgänge entsprechend.

- `AS_SPLIT_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

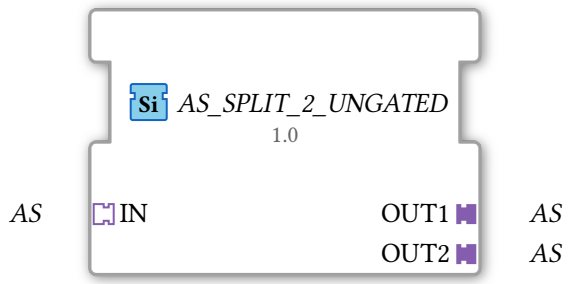
Der **AS_SPLIT_2** ist ein einfacher, aber nützlicher Baustein zur Vervielfältigung von Adapter-Datenströmen. Durch seine generische Definition und den Verzicht auf komplexe Logik eignet er sich hervorragend als universelles Verteilerelement in 4diac-basierten Automatisierungssystemen, bei denen Signalaufteilung ohne Datenveränderung gefordert ist.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AS_SPLIT_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AS_SPLIT_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AS_SPLIT_2_UNGATED** dient dem Aufteilen eines eingehenden Adapter-Signals vom Typ `adapter::types::unidirectional::AS` auf zwei identische Ausgangsadapter. Er ist als generischer Baustein konzipiert und eignet sich für Anwendungen, in denen ein Adapter-Datenstrom an mehrere nachfolgende Komponenten verteilt werden muss, ohne das Originalsignal zu verändern.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden. Der Baustein arbeitet rein datengetrieben über die Adapter.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden. Die gesamte Datenübertragung erfolgt über den Adapter-Eingang `IN`.

Daten-Ausgänge

Keine vorhanden. Die Ausgangsdaten werden über die Adapter `OUT1` und `OUT2` bereitgestellt.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	`IN`	`adapter::types::unidirectional::AS`	Eingangsadapter, dessen Daten auf beide Ausgänge verteilt werden.
Plug	`OUT1`	`adapter::types::unidirectional::AS`	Erster Ausgangsadapter – erhält eine Kopie der eingehenden Daten.
Plug	`OUT2`	`adapter::types::unidirectional::AS`	Zweiter Ausgangsadapter – erhält ebenfalls eine Kopie der eingehenden Daten.

Funktionsweise

Der Baustein fungiert als passiver Verteiler: Ein am Socket `IN` anliegender Adapter-Datenstrom wird unverändert an beide Plugs `OUT1` und `OUT2` weitergeleitet. Es findet keinerlei Datenmanipulation, Pufferung oder Filterung statt. Die Aufteilung erfolgt signalgetreu, sodass die an `OUT1` und `OUT2` anliegenden Daten zeitlich und inhaltlich exakt den Daten von `IN` entsprechen.

Technische Besonderheiten

- **Generischer Typ:** Der Baustein ist als generischer Funktionsblock deklariert (`GenericClassName = GEN_AS_SPLIT`). Dies ermöglicht die Verwendung in verschiedenen Kontexten, sofern die angeschlossenen Adapter dem Typ `AS` entsprechen.
- **Keine Ereignissteuerung:** Der Datenfluss erfolgt rein über die Adapter-Schnittstellen. Es sind keine Ereignis-Ein- oder Ausgänge vorhanden, wodurch der Baustein keine eigenständige Ablaufsteuerung besitzt.
- **Keine Datenpuffer:** Da weder Speicher noch Verzögerung implementiert sind, eignet sich der Baustein besonders für Echtzeitanwendungen mit niedriger Latenz.

Zustandsübersicht

Der Baustein besitzt keine inneren Zustände oder sequentielle Logik. Er verhält sich wie eine reine Verdrahtungskomponente und arbeitet kontinuierlich ohne explizite Zustandsmaschine.

Anwendungsszenarien

- **Signalfan-out:** Verteilung eines Sensor-Adapters an mehrere parallel geschaltete Auswerteeinheiten.
- **Redundanz:** Einspeisung eines Steuersignals in zwei unabhängige Aktorstränge.
- **Monitoring:** Anschluss eines Analyse- oder Logging-Adapters parallel zum bestehenden Datenpfad.

Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu einem einfachen Verbindungsknoten (der nur eine 1:1-Leitung realisiert) ermöglicht `AS_SPLIT_2_UNGATED` eine saubere, konfigurierbare Aufteilung auf zwei Ausgänge. Gegenüber einem **AS_MUX** oder **AS_DEMUX** fehlen diesem Baustein Auswahl- oder Priorisierungslogik – er verteilt immer alle eingehenden Daten an alle Ausgänge. Ähnliche Blöcke wie `AS_SPLIT_3` oder `AS_SPLIT_N` erweitern die Anzahl der Ausgänge entsprechend.

- `AS_SPLIT_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

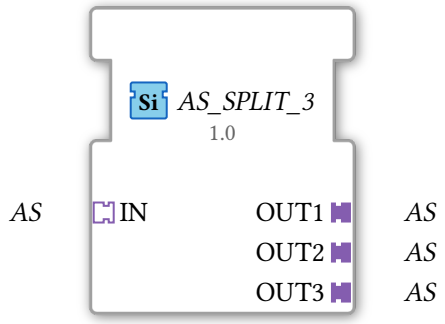
Fazit

Der **AS_SPLIT_2_UNGATED** ist ein einfacher, aber nützlicher Baustein zur Vervielfältigung von Adapter-Datenströmen. Durch seine generische Definition und den Verzicht auf komplexe Logik eignet er sich hervorragend als universelles Verteilerelement in 4diac-basierten Automatisierungssystemen, bei denen Signalaufteilung ohne Datenveränderung gefordert ist.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AS_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AS_SPLIT_3 dient der Verteilung eines eingehenden Adapter-Datenstroms auf drei identische Ausgänge. Er ist als generischer Baustein realisiert und auf die Verwendung mit dem `adapter::types::unidirectional::AS`-Adaptertyp ausgelegt.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Bezeichnung	Typ	Richtung	Beschreibung
IN	`adapter::types::unidirectional::AS`	Socket	Eingangsadapter – Datenquelle, die auf drei Ausgänge verteilt wird.
OUT1	`adapter::types::unidirectional::AS`	Plug	Erster Ausgang – erhält die unveränderten Daten von IN.
OUT2	`adapter::types::unidirectional::AS`	Plug	Zweiter Ausgang – erhält die unveränderten Daten von IN.
OUT3	`adapter::types::unidirectional::AS`	Plug	Dritter Ausgang – erhält die unveränderten Daten von IN.

Funktionsweise

Der Baustein leitet die über den Socket `IN` empfangenen Adapterdaten identisch und ohne Verzögerung an alle drei Adapter-Plugs `OUT1`, `OUT2` und `OUT3` weiter. Es findet keine Datenmanipulation, Filterung oder Pufferung statt. Die Verteilung erfolgt in einem reinen Durchleitungsmodus.

Technische Besonderheiten

- **Generischer Typ**: Der Baustein verwendet das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `'GEN_AS_SPLIT'`, um eine generische Instanziierung zu ermöglichen. Der konkrete Adaptertyp (`adapter::types::unidirectional::AS`) wird zur Projektierungszeit festgelegt.
- **Ereignislos**: Da keine Ereignisse (Event-Ein-/Ausgänge) definiert sind, erfolgt die Datenweitergabe rein datengetrieben – Änderungen am Eingangsadapter werden sofort an alle Ausgänge propagiert.
- **Eingetragenes Copyright**: Der FB ist urheberrechtlich geschützt unter der Eclipse Public License 2.0.

Zustandsübersicht

Der Baustein besitzt keinen internen Zustandsautomaten und kein ECC (Execution Control Chart). Das Verhalten ist rein kombinatorisch: Sobald sich die Daten des Eingangsadapters ändern, werden die Ausgangsdaten entsprechend aktualisiert.

Anwendungsszenarien

- **Datenverteilung in Steuerungssystemen**: Aufteilung eines Sensordatenstroms (z.B. AS-i-Bus-Daten) auf mehrere parallel arbeitende Verarbeitungsblöcke.
- **Signalkopie für Diagnosezwecke**: Anschließen eines separaten Monitor- oder Logging-Pfads ohne Beeinflussung der Hauptdaten.
- **Multiplikation von Steuersignalen**: Verteilung eines einzigen Befehlssatzes an mehrere Aktoren oder Subsysteme.

Vergleich mit ähnlichen Bausteinen

- **AS_SPLIT_2**: Ein analoger Baustein mit nur zwei Ausgängen. AS_SPLIT_3 erweitert die Anzahl auf drei Ausgänge.
- **AS_MERGE (theoretisch)**: Im Gegensatz zu einem Merge-Baustein, der mehrere Eingänge zu einem Ausgang zusammenführt, realisiert AS_SPLIT_3 die umgekehrte Funktion (1 → N).
- **Generische Split-FBs**: Ähnliche Konzepte existieren für Daten-Eingänge (z.B. SPLIT_INT), jedoch arbeiten diese auf Elementardaten, während AS_SPLIT_3 Adapter (komplexe Datentypen) verarbeitet.
- `AS_SPLIT_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

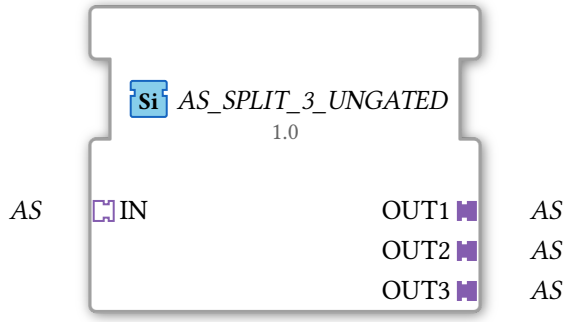
AS_SPLIT_3 ist ein einfacher, aber nützlicher Funktionsblock zur Vervielfältigung von Adapterdaten in der 4diac-IDE. Seine generische Auslegung und der fehlende Ereignisverkehr machen ihn besonders leichtgewichtig und in vielen Szenarien einsetzbar. Die dokumentierte Lizenz (EPL 2.0) stellt sicher, dass der Baustein rechtlich klar in eigene Projekte integriert werden kann.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AS_SPLIT_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AS_SPLIT_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AS_SPLIT_3_UNGATED dient der Verteilung eines eingehenden Adapter-Datenstroms auf drei identische Ausgänge. Er ist als generischer Baustein realisiert und auf die Verwendung mit dem `adapter::types::unidirectional::AS`-Adaptertyp ausgelegt.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Bezeichnung	Typ	Richtung	Beschreibung
IN	`adapter::types::unidirectional::AS`	Socket	Eingangsadapter – Datenquelle, die auf drei Ausgänge verteilt wird.
OUT1	`adapter::types::unidirectional::AS`	Plug	Erster Ausgang – erhält die unveränderten Daten von IN.
OUT2	`adapter::types::unidirectional::AS`	Plug	Zweiter Ausgang – erhält die unveränderten Daten von IN.
OUT3	`adapter::types::unidirectional::AS`	Plug	Dritter Ausgang – erhält die unveränderten Daten von IN.

Funktionsweise

Der Baustein leitet die über den Socket `IN` empfangenen Adapterdaten identisch und ohne Verzögerung an alle drei Adapter-Plugs `OUT1`, `OUT2` und `OUT3` weiter. Es findet keine Datenmanipulation, Filterung oder Pufferung statt. Die Verteilung erfolgt in einem reinen Durchleitungsmodus.

Technische Besonderheiten

- **Generischer Typ**: Der Baustein verwendet das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `'GEN_AS_SPLIT'`, um eine generische Instanziierung zu ermöglichen. Der konkrete Adaptertyp (`adapter::types::unidirectional::AS`) wird zur Projektierungszeit festgelegt.
- **Ereignislos**: Da keine Ereignisse (Event-Ein-/Ausgänge) definiert sind, erfolgt die Datenweitergabe rein datengetrieben – Änderungen am Eingangsadapter werden sofort an alle Ausgänge propagiert.
- **Eingetragenes Copyright**: Der FB ist urheberrechtlich geschützt unter der Eclipse Public License 2.0.

Zustandsübersicht

Der Baustein besitzt keinen internen Zustandsautomaten und kein ECC (Execution Control Chart). Das Verhalten ist rein kombinatorisch: Sobald sich die Daten des Eingangsadapters ändern, werden die Ausgangsdaten entsprechend aktualisiert.

Anwendungsszenarien

- **Datenverteilung in Steuerungssystemen**: Aufteilung eines Sensordatenstroms (z.B. AS-i-Bus-Daten) auf mehrere parallel arbeitende Verarbeitungsblöcke.
- **Signalkopie für Diagnosezwecke**: Anschließen eines separaten Monitor- oder Logging-Pfads ohne Beeinflussung der Hauptdaten.
- **Multiplikation von Steuersignalen**: Verteilung eines einzigen Befehlssatzes an mehrere Aktoren oder Subsysteme.

Vergleich mit ähnlichen Bausteinen

- **AS_SPLIT_2**: Ein analoger Baustein mit nur zwei Ausgängen. AS_SPLIT_3_UNGATED erweitert die Anzahl auf drei Ausgänge.
- **AS_MERGE (theoretisch)**: Im Gegensatz zu einem Merge-Baustein, der mehrere Eingänge zu einem Ausgang zusammenführt, realisiert AS_SPLIT_3_UNGATED die umgekehrte Funktion (1 ⟶ N).
- **Generische Split-FBs**: Ähnliche Konzepte existieren für Daten-Eingänge (z.B. SPLIT_INT), jedoch arbeiten diese auf Elementardaten, während AS_SPLIT_3_UNGATED Adapter (komplexe Datentypen) verarbeitet.
- `AS_SPLIT_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

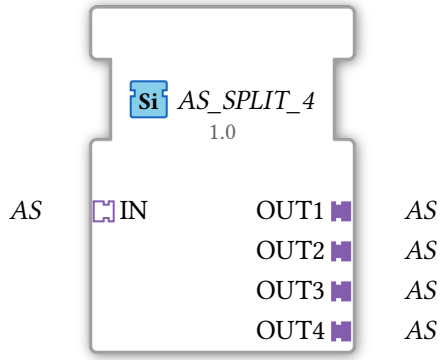
Fazit

AS_SPLIT_3_UNGATED ist ein einfacher, aber nützlicher Funktionsblock zur Vervielfältigung von Adapterdaten in der 4diac-IDE. Seine generische Auslegung und der fehlende Ereignisverkehr machen ihn besonders leichtgewichtig und in vielen Szenarien einsetzbar. Die dokumentierte Lizenz (EPL 2.0) stellt sicher, dass der Baustein rechtlich klar in eigene Projekte integriert werden kann.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AS_SPLIT_4



Kein Bild verfügbar.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AS_SPLIT_4** teilt einen eingehenden unidirektionalen AS-Adapter in vier identische Ausgänge auf. Er dient als generischer Splitter, der das über den Socket `IN` empfangene Signal unverändert an die vier Plugs `OUT1` bis `OUT4` weiterleitet.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AS`	IN	Socket	Eingehender AS-Signalpfad (Eingang)
`adapter::types::unidirectional::AS`	OUT1	Plug	Erster Ausgang (identisch zu IN)
`adapter::types::unidirectional::AS`	OUT2	Plug	Zweiter Ausgang (identisch zu IN)
`adapter::types::unidirectional::AS`	OUT3	Plug	Dritter Ausgang (identisch zu IN)
`adapter::types::unidirectional::AS`	OUT4	Plug	Vierter Ausgang (identisch zu IN)

Funktionsweise

Der FB arbeitet rein passiv: Jedes über den Adapter `IN` eingehende Signal wird ohne Verzögerung oder Transformation an alle vier Plugs (`OUT1`–`OUT4`) weitergegeben. Es findet keine Datenpufferung, keine logische Verarbeitung und keine Zustandsänderung statt. Die Verteilung ist deterministisch und erfolgt zur Kompilier- oder Laufzeit entsprechend der Adapterverträge.

Technische Besonderheiten

- **Keine Ereignis-/Datenschnittstellen:** Die gesamte Kommunikation läuft ausschließlich über Adapter.
- **Einheitlicher Typ:** Alle fünf Adapter sind vom gleichen unidirektionalen Typ `AS`.
- **Generische Parametrisierung:** Der FB trägt die Attribute `GenericClassName` (Wert: `'GEN_AS_SPLIT'`) und `TypeHash`, wodurch er anwendungsspezifisch angepasst werden kann (z. B. durch Refactoring oder Modelltransformation).
- **Zustandslos:** Kein ECC, keine internen Variablen – das Verhalten ist vollständig durch die Adapterverträge definiert.

Zustandsübersicht

Der FB besitzt keine Zustandsautomaten. Das Verhalten ist zeitlos und gleichbleibend: Das Eingangssignal wird stets unmittelbar auf alle Ausgänge dupliziert.

Anwendungsszenarien

- **Signalverteilung in Steuerungen:** Aufteilung einer Geschwindigkeits- oder Positionsvorgabe auf mehrere parallele Achsen.
- **Sensorwert-Multiplex:** Verteilen eines gemessenen Werts an mehrere Auswerte- oder Überwachungsinstanzen.
- **Modulare Anbindung:** Einsatz in hierarchischen Systemen, bei denen ein Adaptersignal mehrfach benötigt wird, ohne den Sender mehrfach zu instanziieren.

Vergleich mit ähnlichen Bausteinen

- **AS_MERGE:** Führt mehrere AS-Eingänge zu einem Ausgang zusammen – das funktionale Gegenstück zum Split.
- **AS_SPLIT_2:** Gleiche Funktionalität, aber nur zwei Ausgänge. `AS_SPLIT_4` erweitert dies auf vier.
- **Ereignis-basierte Splitter (z. B. E_SPLIT):** Arbeiten mit Ereignis- und Dateneingängen/-ausgängen und verbreiten Ereignisse. `AS_SPLIT_4` operiert dagegen auf Adapterebene und leitet den gesamten Adaptervertrag (inkl. aller darin enthaltenen Ereignisse und Daten) weiter.
- `AS_SPLIT_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

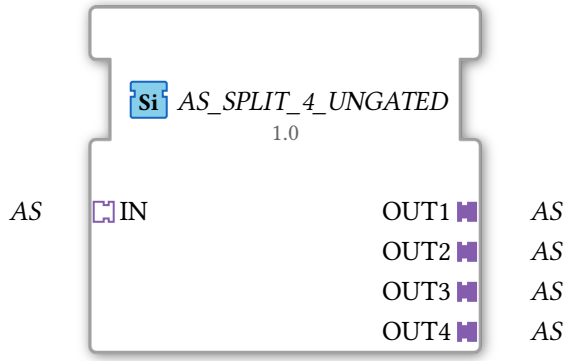
`AS_SPLIT_4` ist ein einfacher, aber essentieller Baustein zur strukturellen Vervielfältigung von unidirektionalen AS-Adapterverbindungen in 4diac-Anwendungen. Seine generische Auslegung und die reine Adapterschnittstelle machen ihn ideal für modulare und skalierbare Steuerungsarchitekturen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AS_SPLIT_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AS_SPLIT_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



Kein Bild verfügbar.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AS_SPLIT_4_UNGATED** teilt einen eingehenden unidirektionalen AS-Adapter in vier identische Ausgänge auf. Er dient als generischer Splitter, der das über den Socket `IN` empfangene Signal unverändert an die vier Plugs `OUT1` bis `OUT4` weiterleitet.

Schnittstellenstruktur

Ereignis-Eingänge
Keine.

Ereignis-Ausgänge
Keine.

Daten-Eingänge
Keine.

Daten-Ausgänge
Keine.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AS`	IN	Socket	Eingehender AS-Signalpfad (Eingang)
`adapter::types::unidirectional::AS`	OUT1	Plug	Erster Ausgang (identisch zu IN)
`adapter::types::unidirectional::AS`	OUT2	Plug	Zweiter Ausgang (identisch zu IN)
`adapter::types::unidirectional::AS`	OUT3	Plug	Dritter Ausgang (identisch zu IN)
`adapter::types::unidirectional::AS`	OUT4	Plug	Vierter Ausgang (identisch zu IN)

Funktionsweise
Der FB arbeitet rein passiv: Jedes über den Adapter `IN` eingehende Signal wird ohne Verzögerung oder Transformation an alle vier Plugs (`OUT1`–`OUT4`) weitergegeben. Es findet keine Datenpufferung, keine logische Verarbeitung und keine Zustandsänderung statt. Die Verteilung ist deterministisch und erfolgt zur Kompilier- oder Laufzeit entsprechend der Adapterverträge.

Technische Besonderheiten
- **Keine Ereignis-/Datenschnittstellen:** Die gesamte Kommunikation läuft ausschließlich über Adapter.
- **Einheitlicher Typ:** Alle fünf Adapter sind vom gleichen unidirektionalen Typ `AS`.
- **Generische Parametrisierung:** Der FB trägt die Attribute `GenericClassName` (Wert: `'GEN_AS_SPLIT'`) und `TypeHash`, wodurch er anwendungsspezifisch angepasst werden kann (z. B. durch Refactoring oder Modelltransformation).
- **Zustandslos:** Kein ECC, keine internen Variablen – das Verhalten ist vollständig durch die Adapterverträge definiert.

Zustandsübersicht
Der FB besitzt keine Zustandsautomaten. Das Verhalten ist zeitlos und gleichbleibend: Das Eingangssignal wird stets unmittelbar auf alle Ausgänge dupliziert.

Anwendungsszenarien
- **Signalverteilung in Steuerungen:** Aufteilung einer Geschwindigkeits- oder Positionsvorgabe auf mehrere parallele Achsen.
- **Sensorwert-Multiplex:** Verteilen eines gemessenen Werts an mehrere Auswerte- oder Überwachungsinstanzen.
- **Modulare Anbindung:** Einsatz in hierarchischen Systemen, bei denen ein Adaptersignal mehrfach benötigt wird, ohne den Sender mehrfach zu instanziieren.

Vergleich mit ähnlichen Bausteinen
- **AS_MERGE:** Führt mehrere AS-Eingänge zu einem Ausgang zusammen – das funktionale Gegenstück zum Split.
- **AS_SPLIT_2:** Gleiche Funktionalität, aber nur zwei Ausgänge. `AS_SPLIT_4_UNGATED` erweitert dies auf vier.

- **Ereignis-basierte Splitter (z. B. E_SPLIT):** Arbeiten mit Ereignis- und Dateneingängen/-ausgängen und verbreiten Ereignisse. `AS_SPLIT_4_UNGATED` operiert dagegen auf Adapterebene und leitet den gesamten Adaptervertrag (inkl. aller darin enthaltenen Ereignisse und Daten) weiter.
- **`AS_SPLIT_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

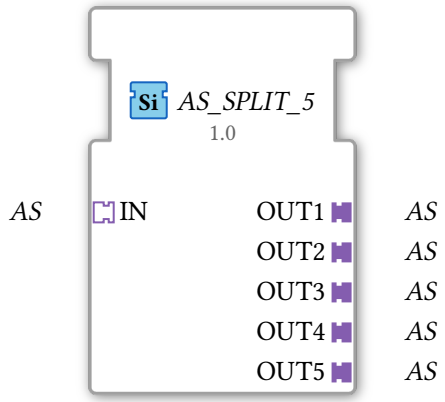
Fazit

`AS_SPLIT_4_UNGATED` ist ein einfacher, aber essentieller Baustein zur strukturellen Vervielfältigung von unidirektionalen AS-Adapterverbindungen in 4diac-Anwendungen. Seine generische Auslegung und die reine Adapterschnittstelle machen ihn ideal für modulare und skalierbare Steuerungsarchitekturen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AS_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AS_SPLIT_5 ist ein generischer Baustein, der einen eingehenden Adapter vom Typ „AS“ (unidirectional) auf fünf separate Ausgangsadapter desselben Typs verteilt. Er dient zur Signalvervielfachung in Steuerungs- und Automatisierungssystemen, basierend auf der IEC 61499-2 Norm.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge definiert.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge definiert.

Daten-Eingänge

Keine Daten-Eingänge definiert.

Daten-Ausgänge

Keine Daten-Ausgänge definiert.

Adapter

Rolle	Name	Typ	Beschreibung

Socket	IN	adapter::types::unidirectional::AS	Eingangs-Adapter, der das zu verteilende AS-Signal empfängt.
Plug	OUT1	adapter::types::unidirectional::AS	Erster Ausgangs-Adapter für das verteilte Signal.
Plug	OUT2	adapter::types::unidirectional::AS	Zweiter Ausgangs-Adapter.
Plug	OUT3	adapter::types::unidirectional::AS	Dritter Ausgangs-Adapter.
Plug	OUT4	adapter::types::unidirectional::AS	Vierter Ausgangs-Adapter.
Plug	OUT5	adapter::types::unidirectional::AS	Fünfter Ausgangs-Adapter.

Funktionsweise

Der AS_SPLIT_5 Baustein fungiert als passiver Verteiler. Das am Socket „IN“ anliegende AS-Signal wird ohne Verzögerung oder Veränderung identisch an alle fünf Ausgangs-Plugs (OUT1 bis OUT5) weitergeleitet. Die Verteilung erfolgt strikt nach dem Prinzip des Broadcasts – jedes ausgehende Signal entspricht dem Originalsignal. Die Implementierung ist rein adapterbasiert und benötigt keine zusätzlichen Ereignis- oder Datenportverbindungen.

Technische Besonderheiten

- Der Baustein ist als generischer Typ (GenericClassName = ‚GEN_AS_SPLIT‘) definiert und kann daher durch die 4diac-IDE parametrisiert werden.
- Es wird ein TypeHash verwendet, um die Eindeutigkeit des Bausteins im Projekt sicherzustellen.
- Die Anzahl der Ausgänge ist fest auf fünf festgelegt, eine dynamische Konfiguration ist nicht vorgesehen.
- Die Laufzeitumgebung muss das unidirektionale Adapter-Interface „adapter::types::unidirectional::AS“ unterstützen.

Zustandsübersicht

Der Baustein besitzt keinen internen Zustandsautomaten. Die Signalverteilung erfolgt kontinuierlich und ohne sequentielle Logik. Daher entfällt eine Zustandsübersicht.

Anwendungsszenarien

- **Steuerungsverdrahtung**: Aufteilung eines von einer übergeordneten Steuerung kommenden AS-Signals auf mehrere Unterstationen oder Aktoren.
- **Simulation und Test**: Vervielfachung eines Testsignals zur parallelen Ansteuerung mehrerer simulierter Komponenten.
- **Redundanz**: Bereitstellung desselben Signals auf verschiedenen Pfaden, um Redundanz oder Lastverteilung zu ermöglichen.

Vergleich mit ähnlichen Bausteinen

Da AS_SPLIT_5 für eine spezifische Adapterarchitektur (unidirectional AS) konzipiert ist, existieren typischerweise ähnliche Bausteine wie z. B. „AS_SPLIT_2“, „AS_SPLIT_3“ oder verallgemeinerte Splitter mit variabler Anzahl von Ausgängen. Der vorliegende Baustein unterscheidet sich durch die feste Anzahl von fünf Ausgängen und die reine Adapter-Schnittstelle ohne Daten- oder Ereignisports.

- `AS_SPLIT_5_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

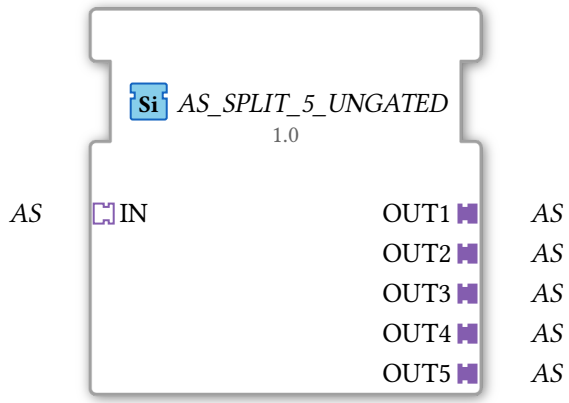
Der AS_SPLIT_5 ist ein einfacher, aber effektiver Funktionsblock zur Vervielfachung eines AS-Adaptersignals auf fünf parallele Ausgänge. Durch seine generische Implementierung und klare Schnittstelle eignet er sich gut für modulare Automatisierungslösungen nach IEC 61499.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AS_SPLIT_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AS_SPLIT_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AS_SPLIT_5_UNGATED ist ein generischer Baustein, der einen eingehenden Adapter vom Typ „AS“ (unidirectional) auf fünf separate Ausgangsadapter desselben Typs verteilt. Er dient zur Signalvervielfachung in Steuerungs- und Automatisierungssystemen, basierend auf der IEC 61499-2 Norm.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge definiert.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge definiert.

Daten-Eingänge

Keine Daten-Eingänge definiert.

Daten-Ausgänge

Keine Daten-Ausgänge definiert.

Adapter

Rolle	Name	Typ	Beschreibung
Socket	IN	adapter::types::unidirectional::AS	Eingangs-Adapter, der das zu verteilende AS-Signal empfängt.
Plug	OUT1	adapter::types::unidirectional::AS	Erster Ausgangs-Adapter für das verteilte Signal.

Plug	OUT2	adapter::types::unidirectional::AS	Zweiter Ausgangs-Adapter.
Plug	OUT3	adapter::types::unidirectional::AS	Dritter Ausgangs-Adapter.
Plug	OUT4	adapter::types::unidirectional::AS	Vierter Ausgangs-Adapter.
Plug	OUT5	adapter::types::unidirectional::AS	Fünfter Ausgangs-Adapter.

Funktionsweise

Der AS_SPLIT_5_UNGATED Baustein fungiert als passiver Verteiler. Das am Socket „IN“ anliegende AS-Signal wird ohne Verzögerung oder Veränderung identisch an alle fünf Ausgangs-Plugs (OUT1 bis OUT5) weitergeleitet. Die Verteilung erfolgt strikt nach dem Prinzip des Broadcasts – jedes ausgehende Signal entspricht dem Originalsignal. Die Implementierung ist rein adapterbasiert und benötigt keine zusätzlichen Ereignis- oder Datenportverbindungen.

Technische Besonderheiten

- Der Baustein ist als generischer Typ (GenericClassName = ‚GEN_AS_SPLIT‘) definiert und kann daher durch die 4diac-IDE parametrisiert werden.
- Es wird ein TypeHash verwendet, um die Eindeutigkeit des Bausteins im Projekt sicherzustellen.
- Die Anzahl der Ausgänge ist fest auf fünf festgelegt, eine dynamische Konfiguration ist nicht vorgesehen.
- Die Laufzeitumgebung muss das unidirektionale Adapter-Interface „adapter::types::unidirectional::AS“ unterstützen.

Zustandsübersicht

Der Baustein besitzt keinen internen Zustandsautomaten. Die Signalverteilung erfolgt kontinuierlich und ohne sequentielle Logik. Daher entfällt eine Zustandsübersicht.

Anwendungsszenarien

- **Steuerungsverdrahtung**: Aufteilung eines von einer übergeordneten Steuerung kommenden AS-Signals auf mehrere Unterstationen oder Aktoren.
- **Simulation und Test**: Vervielfachung eines Testsignals zur parallelen Ansteuerung mehrerer simulierter Komponenten.
- **Redundanz**: Bereitstellung desselben Signals auf verschiedenen Pfaden, um Redundanz oder Lastverteilung zu ermöglichen.

Vergleich mit ähnlichen Bausteinen

Da AS_SPLIT_5_UNGATED für eine spezifische Adapterarchitektur (unidirectional AS) konzipiert ist, existieren typischerweise ähnliche Bausteine wie z. B. „AS_SPLIT_2“, „AS_SPLIT_3“ oder verallgemeinerte Splitter mit variabler Anzahl von Ausgängen. Der vorliegende Baustein unterscheidet sich durch die feste Anzahl von fünf Ausgängen und die reine Adapter-Schnittstelle ohne Daten- oder Ereignisports.

- `AS_SPLIT_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

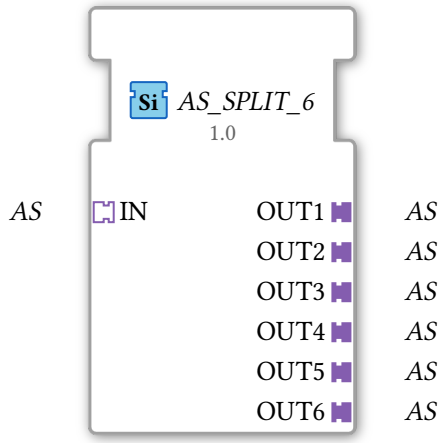
Fazit

Der AS_SPLIT_5_UNGATED ist ein einfacher, aber effektiver Funktionsblock zur Vervielfachung eines AS-Adaptersignals auf fünf parallele Ausgänge. Durch seine generische Implementierung und klare Schnittstelle eignet er sich gut für modulare Automatisierungslösungen nach IEC 61499.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AS_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AS_SPLIT_6** dient der Aufteilung eines einzelnen eingehenden AS-Adapter-Signals auf sechs separate AS-Adapter-Ausgänge. Er ist als generischer Baustein realisiert und kann mit unterschiedlichen unidirektionalen AS-Adapter-Typen (z. B. Sensor- oder Aktor-Adaptern) verwendet werden.

Schnittstellenstruktur

Ereignis-Eingänge

– Keine –

Ereignis-Ausgänge

– Keine –

Daten-Eingänge

– Keine –

Daten-Ausgänge

– Keine –

Adapter

Richtung	Name	Typ
SOCKET	IN	unidirectional::AS (generisch)
PLUG	OUT1	unidirectional::AS (generisch)
PLUG	OUT2	unidirectional::AS (generisch)
PLUG	OUT3	unidirectional::AS (generisch)
PLUG	OUT4	unidirectional::AS (generisch)
PLUG	OUT5	unidirectional::AS (generisch)
PLUG	OUT6	unidirectional::AS (generisch)

Funktionsweise

Der Baustein leitet alle am SOCKET **IN** anliegenden Adapter-Signale unverändert an alle sechs PLUGs (**OUT1** bis **OUT6**) weiter. Es findet keine Datenverarbeitung oder -umwandlung statt – die Verteilung erfolgt rein strukturell. Somit verhalten sich alle Ausgänge identisch zum Eingang.

Technische Besonderheiten

- **Generischer Baustein:** Der konkrete AS-Adapter-Typ wird erst bei der Instanziierung festgelegt (z. B. durch die Angabe eines GenericClassNamens wie `'GEN_AS_SPLIT'`). Dadurch ist der Baustein für verschiedene Protokolle oder Datenstrukturen einsetzbar.
- **Keine Zustandslogik:** Der Baustein besitzt keine Zustandsmaschine, keine Algorithmen und keine Ereignisse – er arbeitet rein passiv und sofortig.
- **Unidirektional:** Die Adapter sind vom Typ `unidirectional::AS`, d. h. die Daten fließen nur vom Eingang zu den Ausgängen.

Zustandsübersicht

Der Baustein verfügt über keine Eckwert-Zustandsautomatik (ECC), da keine sequentiellen Abläufe oder Verarbeitungsschritte definiert sind. Die Signalverteilung erfolgt statisch und kontinuierlich.

Anwendungsszenarien

- **Signalvervielfältigung:** Ein einzelner Sensor-Adapter (z. B. Temperaturmessung) wird mehreren Verarbeitungseinheiten zur Verfügung gestellt.
- **Datenbus-Kopplung:** Ein gemeinsames Signal wird an sechs parallele Steuerungsmodule oder Anzeigen weitergegeben.
- **Prototyping und Test:** Vereinfachte Erzeugung mehrerer identischer Schnittstellen aus einer Quelle.

Vergleich mit ähnlichen Bausteinen

- **AS_SPLIT_2 / AS_SPLIT_4** – Identische Funktionalität, jedoch mit nur zwei bzw. vier Ausgängen. Der vorliegende Baustein eignet sich für Anwendungen mit genau sechs Zielen.
- **AS_MERGE_...** – Führt mehrere Adapter-Eingänge zu einem Ausgang zusammen – gegensätzliche Aufgabe.
- **Manuelle Verdrahtung** – Könnte über mehrere Adapter-Kopplungen realisiert werden, erhöht jedoch den Konfigurationsaufwand.
- `AS_SPLIT_6_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

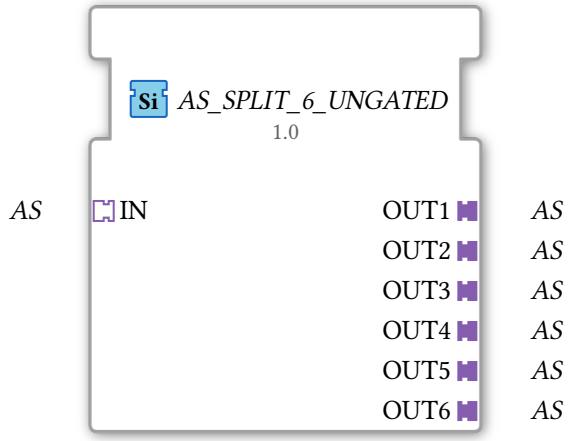
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der **AS_SPLIT_6** ist ein einfacher, aber effektiver generischer Funktionsbaustein zur Aufteilung eines unidirektionalen AS-Signals auf sechs separate Ausgänge. Durch seine generische Natur kann er flexibel in verschiedenen Automatisierungsumgebungen eingesetzt werden, wo eine schnelle und übersichtliche Signalvervielfältigung erforderlich ist.

AS_SPLIT_6_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AS_SPLIT_6`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AS_SPLIT_6_UNGATED** dient der Aufteilung eines einzelnen eingehenden AS-Adapter-Signals auf sechs separate AS-Adapter-Ausgänge. Er ist als generischer Baustein realisiert und kann mit unterschiedlichen unidirektionalen AS-Adapter-Typen (z. B. Sensor- oder Aktor-Adaptern) verwendet werden.

Schnittstellenstruktur

Ereignis-Eingänge

– Keine –

Ereignis-Ausgänge

– Keine –

Daten-Eingänge

– Keine –

Daten-Ausgänge

– Keine –

Adapter

Richtung	Name	Typ
SOCKET	IN	unidirectional::AS (generisch)
PLUG	OUT1	unidirectional::AS (generisch)
PLUG	OUT2	unidirectional::AS (generisch)
PLUG	OUT3	unidirectional::AS (generisch)
PLUG	OUT4	unidirectional::AS (generisch)
PLUG	OUT5	unidirectional::AS (generisch)
PLUG	OUT6	unidirectional::AS (generisch)

Funktionsweise

Der Baustein leitet alle am SOCKET **IN** anliegenden Adapter-Signale unverändert an alle sechs PLUGs (**OUT1** bis **OUT6**) weiter. Es findet keine Datenverarbeitung oder -umwandlung statt – die Verteilung erfolgt rein strukturell. Somit verhalten sich alle Ausgänge identisch zum Eingang.

Technische Besonderheiten

- **Generischer Baustein:** Der konkrete AS-Adapter-Typ wird erst bei der Instanziierung festgelegt (z. B. durch die Angabe eines GenericClassNamens wie `'GEN_AS_SPLIT'`). Dadurch ist der Baustein für verschiedene Protokolle oder Datenstrukturen einsetzbar.
- **Keine Zustandslogik:** Der Baustein besitzt keine Zustandsmaschine, keine Algorithmen und keine Ereignisse – er arbeitet rein passiv und sofortig.
- **Unidirektional:** Die Adapter sind vom Typ `unidirectional::AS`, d. h. die Daten fließen nur vom Eingang zu den Ausgängen.

Zustandsübersicht

Der Baustein verfügt über keine Eckwert-Zustandsautomatik (ECC), da keine sequentiellen Abläufe oder Verarbeitungsschritte definiert sind. Die Signalverteilung erfolgt statisch und kontinuierlich.

Anwendungsszenarien

- **Signalvervielfältigung:** Ein einzelner Sensor-Adapter (z. B. Temperaturmessung) wird mehreren Verarbeitungseinheiten zur Verfügung gestellt.
- **Datenbus-Kopplung:** Ein gemeinsames Signal wird an sechs parallele Steuerungsmodule oder Anzeigen weitergegeben.
- **Prototyping und Test:** Vereinfachte Erzeugung mehrerer identischer Schnittstellen aus einer Quelle.

Vergleich mit ähnlichen Bausteinen

- **AS_SPLIT_2 / AS_SPLIT_4** – Identische Funktionalität, jedoch mit nur zwei bzw. vier Ausgängen. Der vorliegende Baustein eignet sich für Anwendungen mit genau sechs Zielen.
- **AS_MERGE_...** – Führt mehrere Adapter-Eingänge zu einem Ausgang zusammen – gegensätzliche Aufgabe.
- **Manuelle Verdrahtung** – Könnte über mehrere Adapter-Kopplungen realisiert werden, erhöht jedoch den Konfigurationsaufwand.
- `AS_SPLIT_6`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

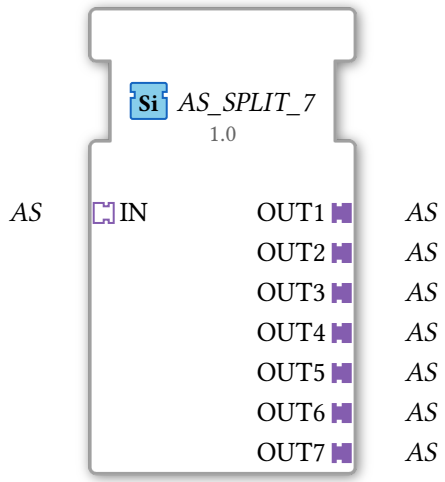
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AS_SPLIT_6_UNGATED** ist ein einfacher, aber effektiver generischer Funktionsbaustein zur Aufteilung eines unidirektionalen AS-Signals auf sechs separate Ausgänge. Durch seine generische Natur kann er flexibel in verschiedenen Automatisierungsumgebungen eingesetzt werden, wo eine schnelle und übersichtliche Signalvervielfältigung erforderlich ist.

AS_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AS_SPLIT_7** dient dazu, ein ankommendes unidirektionales AS-Signal (z. B. ein analoges oder binäres Automatisierungssignal) auf sieben identische Ausgänge zu verteilen. Der Baustein ist als generischer FB realisiert und wird häufig in der Automatisierungstechnik eingesetzt, um ein Signal mehreren nachfolgenden Verbrauchern oder Prozessschritten bereitzustellen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine (Die Signalweitergabe erfolgt ausschließlich über Adapter.)

Daten-Ausgänge

Keine

Adapter

Name	Typ	Richtung	Beschreibung
IN	`adapter::types::unidirectional::AS`	Socket	Empfängt das zu verteilende AS-Signal.
OUT1	`adapter::types::unidirectional::AS`	Plug	Erster Ausgang – gleiches Signal wie am Eingang.
OUT2	`adapter::types::unidirectional::AS`	Plug	Zweiter Ausgang – gleiches Signal wie am Eingang.
OUT3	`adapter::types::unidirectional::AS`	Plug	Dritter Ausgang – gleiches Signal wie am Eingang.
OUT4	`adapter::types::unidirectional::AS`	Plug	Vierter Ausgang – gleiches Signal wie am Eingang.
OUT5	`adapter::types::unidirectional::AS`	Plug	Fünfter Ausgang – gleiches Signal wie am Eingang.
OUT6	`adapter::types::unidirectional::AS`	Plug	Sechster Ausgang – gleiches Signal wie am Eingang.
OUT7	`adapter::types::unidirectional::AS`	Plug	Siebter Ausgang – gleiches Signal wie am Eingang.

Funktionsweise

Der Baustein leitet das am Socket `IN` anliegende AS-Signal unverändert an alle sieben Plug-Ausgänge weiter. Es findet keine Signalaufbereitung, Verstärkung oder Verarbeitung statt. Der FB arbeitet rein passiv und ohne zeitliche Verzögerung – sobald am Eingang ein Signal anliegt, steht es an allen Ausgängen gleichzeitig zur Verfügung.

Technische Besonderheiten

- **Generischer Baustein:** Der FB ist als generischer Typ (`GEN_AS_SPLIT`) implementiert und kann durch Parametrierung in verschiedenen Kontexten verwendet werden.
- **Adapter-basierte Schnittstelle:** Die Kommunikation erfolgt ausschließlich über unidirektionale Adapter des Typs `adapter::types::unidirectional::AS`. Dadurch ist der FB besonders geeignet für modulare Architekturen, bei denen Signale über klar definierte Schnittstellen ausgetauscht werden.

- **Keine Zustandslogik:** Der FB besitzt keine internen Zustände, keine Ereignisse und keine Verarbeitungslogik – die Verteilung erfolgt rein kombinatorisch.
- **Einfache Erweiterbarkeit:** Durch die generische Natur des Bausteins lassen sich analoge Splitter-Varianten (z. B. `AS_SPLIT_2`, `AS_SPLIT_4`) leicht ableiten oder durch Anpassung der Parameter erzeugen.

Zustandsübersicht

Der **AS_SPLIT_7** besitzt keine Zustandsautomaten, da er rein passiv arbeitet. Es gibt keine Start-, Stopp- oder Fehlerzustände. Das Verhalten ist zu jeder Zeit ausschließlich durch den aktuellen Wert des Eingangssignals bestimmt.

Anwendungsszenarien

- **Signalverteilung in Leitsystemen:** Ein analoges Messsignal (z. B. Druck, Temperatur) wird auf mehrere Anzeigen, Regler oder Datensammler verteilt.
- **Parallelschaltung von Aktoren:** Ein Steuersignal (z. B. 0-10 V) soll gleichzeitig mehrere Stellglieder ansteuern.
- **Redundante Signalpfade:** Das gleiche Signal wird mehreren unabhängigen Verarbeitungseinheiten zur Verfügung gestellt, um Redundanz zu schaffen.
- **Test- und Simulationsumgebungen:** Ein generiertes Testsignal wird auf mehrere Simulationskomponenten aufgeteilt.

Vergleich mit ähnlichen Bausteinen

Der **AS_SPLIT_7** gehört zu einer Familie von Splitter-Bausteinen, die sich nur in der Anzahl der Ausgänge unterscheiden (z. B. `AS_SPLIT_2`, `AS_SPLIT_3`, ..., `AS_SPLIT_N`). Gegenüber einer manuellen Verkabelung bietet der FB den Vorteil der Wiederverwendbarkeit, klaren Schnittstellendefinition und einfachen Parametrierbarkeit. Im Gegensatz zu aktiven Signalverteilern (z. B. mit Verstärkung) nimmt der Baustein keinerlei Einfluss auf das Signal und verhält sich daher ideal für passive Anwendungen.

- `AS_SPLIT_7_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

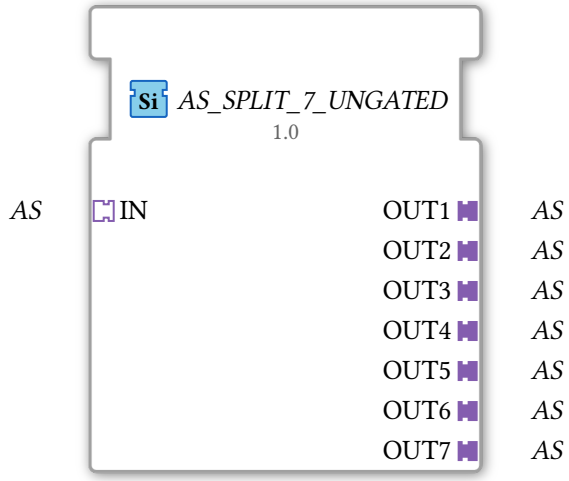
Der **AS_SPLIT_7** ist ein einfacher, aber nützlicher Funktionsblock zur passiven Verteilung eines unidirektionalen AS-Signals auf sieben Ausgänge. Dank seiner generischen Implementierung und der Adapter-Schnittstelle fügt er sich nahtlos in moderne, komponentenbasierte Automatisierungslösungen ein. Für Anwendungen, die eine reine Signalvervielfachung ohne Verarbeitung erfordern, stellt er eine zuverlässige und leicht verständliche Lösung dar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Gesamtwiderstand in Reihen- & Parallelschaltung auf ms-muc-docs.de

AS_SPLIT_7_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AS_SPLIT_7`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AS_SPLIT_7_UNGATED** dient dazu, ein ankommendes unidirektionales AS-Signal (z. B. ein analoges oder binäres Automatisierungssignal) auf sieben identische Ausgänge zu verteilen. Der Baustein ist als generischer FB realisiert und wird häufig in der Automatisierungstechnik eingesetzt, um ein Signal mehreren nachfolgenden Verbrauchern oder Prozessschritten bereitzustellen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine (Die Signalweitergabe erfolgt ausschließlich über Adapter.)

Daten-Ausgänge

Keine

Adapter

Name	Typ	Richtung	Beschreibung
IN	`adapter::types::unidirectional::AS`	Socket	Empfängt das zu verteilende AS-Signal.
OUT1	`adapter::types::unidirectional::AS`	Plug	Erster Ausgang – gleiches Signal wie am Eingang.
OUT2	`adapter::types::unidirectional::AS`	Plug	Zweiter Ausgang – gleiches Signal wie am Eingang.
OUT3	`adapter::types::unidirectional::AS`	Plug	Dritter Ausgang – gleiches Signal wie am Eingang.
OUT4	`adapter::types::unidirectional::AS`	Plug	Vierter Ausgang – gleiches Signal wie am Eingang.
OUT5	`adapter::types::unidirectional::AS`	Plug	Fünfter Ausgang – gleiches Signal wie am Eingang.

OUT6	`adapter::types::unidirectional::AS`	Plug	Sechster Ausgang – gleiches Signal wie am Eingang.
OUT7	`adapter::types::unidirectional::AS`	Plug	Siebter Ausgang – gleiches Signal wie am Eingang.

Funktionsweise

Der Baustein leitet das am Socket `IN` anliegende AS-Signal unverändert an alle sieben Plug-Ausgänge weiter. Es findet keine Signalaufbereitung, Verstärkung oder Verarbeitung statt. Der FB arbeitet rein passiv und ohne zeitliche Verzögerung – sobald am Eingang ein Signal anliegt, steht es an allen Ausgängen gleichzeitig zur Verfügung.

Technische Besonderheiten

- **Generischer Baustein:** Der FB ist als generischer Typ (`GEN_AS_SPLIT`) implementiert und kann durch Parametrierung in verschiedenen Kontexten verwendet werden.
- **Adapter-basierte Schnittstelle:** Die Kommunikation erfolgt ausschließlich über unidirektionale Adapter des Typs `adapter::types::unidirectional::AS`. Dadurch ist der FB besonders geeignet für modulare Architekturen, bei denen Signale über klar definierte Schnittstellen ausgetauscht werden.
- **Keine Zustandslogik:** Der FB besitzt keine internen Zustände, keine Ereignisse und keine Verarbeitungslogik – die Verteilung erfolgt rein kombinatorisch.
- **Einfache Erweiterbarkeit:** Durch die generische Natur des Bausteins lassen sich analoge Splitter-Varianten (z. B. `AS_SPLIT_2`, `AS_SPLIT_4`) leicht ableiten oder durch Anpassung der Parameter erzeugen.

Zustandsübersicht

Der **AS_SPLIT_7_UNGATED** besitzt keine Zustandsautomaten, da er rein passiv arbeitet. Es gibt keine Start-, Stopp- oder Fehlerzustände. Das Verhalten ist zu jeder Zeit ausschließlich durch den aktuellen Wert des Eingangssignals bestimmt.

Anwendungsszenarien

- **Signalverteilung in Leitsystemen:** Ein analoges Messsignal (z. B. Druck, Temperatur) wird auf mehrere Anzeigen, Regler oder Datensammler verteilt.
- **Parallelschaltung von Aktoren:** Ein Steuersignal (z. B. 0-10 V) soll gleichzeitig mehrere Stellglieder ansteuern.
- **Redundante Signalpfade:** Das gleiche Signal wird mehreren unabhängigen Verarbeitungseinheiten zur Verfügung gestellt, um Redundanz zu schaffen.
- **Test- und Simulationsumgebungen:** Ein generiertes Testsignal wird auf mehrere Simulationskomponenten aufgeteilt.

Vergleich mit ähnlichen Bausteinen

Der **AS_SPLIT_7_UNGATED** gehört zu einer Familie von Splitter-Bausteinen, die sich nur in der Anzahl der Ausgänge unterscheiden (z. B. `AS_SPLIT_2`, `AS_SPLIT_3`, ..., `AS_SPLIT_N`). Gegenüber einer manuellen Verkabelung bietet der FB den Vorteil der Wiederverwendbarkeit, klaren Schnittstellendefinition und einfachen Parametrierbarkeit. Im Gegensatz zu aktiven Signalverteilern (z. B. mit Verstärkung) nimmt der Baustein keinerlei Einfluss auf das Signal und verhält sich daher ideal für passive Anwendungen.

- `AS_SPLIT_7`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

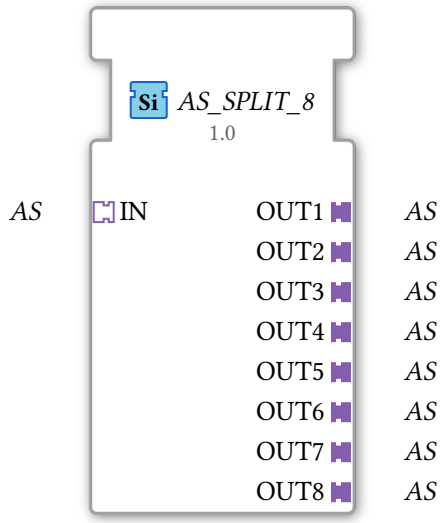
Fazit

Der **AS_SPLIT_7_UNGATED** ist ein einfacher, aber nützlicher Funktionsblock zur passiven Verteilung eines unidirektionalen AS-Signals auf sieben Ausgänge. Dank seiner generischen Implementierung und der Adapter-Schnittstelle fügt er sich nahtlos in moderne, komponentenbasierte Automatisierungslösungen ein. Für Anwendungen, die eine reine Signalvervielfachung ohne Verarbeitung erfordern, stellt er eine zuverlässige und leicht verständliche Lösung dar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Gesamtwiderstand in Reihen- & Parallelschaltung auf ms-muc-docs.de

AS_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AS_SPLIT_8** dient dazu, einen eingehenden *Application Specific* (AS)-Adapter-Datenstrom in acht identische Ausgänge zu verzweigen. Er wird als generischer Baustein (generic FB) bereitgestellt und ist speziell für die Aufteilung von Adapterdaten innerhalb einer IEC 61499-basierten Steuerungsapplikation konzipiert.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AS`	IN	Socket (Eingang)	Empfängt den zu verteilenden AS-Datenstrom.
`adapter::types::unidirectional::AS`	OUT1	Plug (Ausgang)	Erste Kopie des eingehenden AS-Datenstroms.
`adapter::types::unidirectional::AS`	OUT2	Plug (Ausgang)	Zweite Kopie des eingehenden AS-Datenstroms.
`adapter::types::unidirectional::AS`	OUT3	Plug (Ausgang)	Dritte Kopie.
`adapter::types::unidirectional::AS`	OUT4	Plug (Ausgang)	Vierte Kopie.

adapter::types::unidirectional::AS	OUT5	Plug (Ausgang)	Fünfte Kopie.
adapter::types::unidirectional::AS	OUT6	Plug (Ausgang)	Sechste Kopie.
adapter::types::unidirectional::AS	OUT7	Plug (Ausgang)	Siebte Kopie.
adapter::types::unidirectional::AS	OUT8	Plug (Ausgang)	Achte Kopie.

Funktionsweise

Der Baustein arbeitet rein adapterbasiert, ohne Nutzung von Ereignissen oder Daten-Ein-/Ausgängen. Der am Socket `IN` anliegende Adapter-Datenstrom wird intern auf alle acht Ausgangs-Plugs (`OUT1` … `OUT8`) gespiegelt. Jeder Ausgang gibt exakt dieselben Daten weiter, die am Eingang anliegen. Es findet keine Verzögerung, Filterung oder Modifikation statt. Die Verzweigung erfolgt passiv und ohne aktive Laufzeitlogik (kein Zustandsautomat).

Technische Besonderheiten

- **Reiner Adapterbaustein**: Der FB besitzt weder Ereignis- noch Datenschnittstellen im klassischen Sinne; die gesamte Datenübertragung erfolgt über die Adapterverbindungen.
- **Generische Implementierung**: Durch die Deklaration als generischer FB (`GenericClassName = 'GEN_AS_SPLIT'`) kann der Baustein für beliebige AS-Adapter-Typen eingesetzt werden, sofern der zugrunde liegende Adapter-Typ `unidirectional::AS` ist.
- **Keine Zustandsmaschine**: Aufgrund der fehlenden Ereignisse entfällt ein ECC (Execution Control Chart). Die Verteilung erfolgt kontinuierlich.
- **Einfaches 1:8-Splitting**: Optimiert für Situationen, in denen ein Signal auf mehrere nachfolgende Module verteilt werden muss.

Zustandsübersicht

Nicht zutreffend – der Baustein besitzt keine internen Zustände oder Ablaufsteuerung.

Anwendungsszenarien

- **Parallelverteilung von Sensordaten**: Ein einzelner Adapter, der z. B. Füllstandsdaten eines Tanks bereitstellt, wird auf acht überwachnde oder regelnde Funktionseinheiten aufgeteilt.
- **Signal-Multicasting in der Industrie 4.0**: Verteilung eines Steuerbefehls an mehrere Aktoren oder Subsysteme.
- **Testumgebungen** zur Simulation mehrfacher Empfänger eines Datenstroms.

Vergleich mit ähnlichen Bausteinen

- **AS_SPLIT_2 / AS_SPLIT_4**: Analoge Bausteine mit geringerer Ausgangszahl. AS_SPLIT_8 bietet die maximale Verteilung in einem Schritt bei acht Ausgängen.
- **Ereignisbasierte Splitter (z. B. E_SPLIT)**: Diese arbeiten mit Ereignissen und verteilen diese zeitlich gesteuert. Der vorliegende Baustein verteilt dagegen dauerhaft die gesamten Adapterdaten ohne Ereigniskontrolle.
- **Daten-basierte Splitter (z. B. ANY_DISTRIBUTE)**: Trennen Datenwerte auf, benötigen aber zusätzliche Ereignisse. AS_SPLIT_8 ist für reine Adapter-Weiterleitungen optimiert.
- `AS_SPLIT_8_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

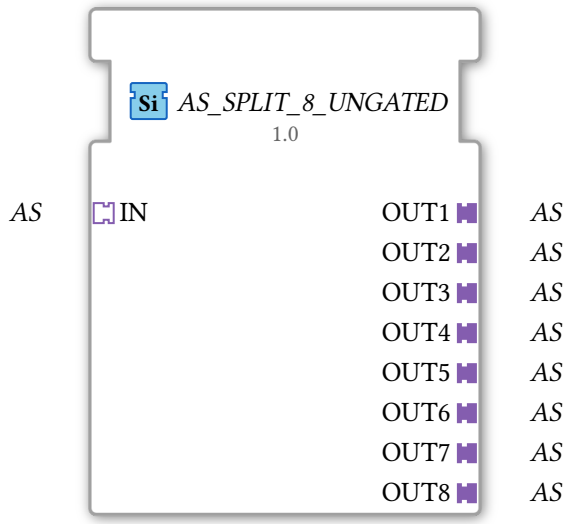
Fazit

Der **AS_SPLIT_8** ist ein einfacher, aber nützlicher Adapterverteiler, der ohne zusätzliche Logik einen eingehenden AS-Datenstrom auf acht Ausgänge dupliziert. Sein generischer Charakter und die klare Struktur machen ihn zu einer soliden Lösung für Szenarien, in denen ein Adaptersignal mehrfach verwendet werden muss. Die Dokumentation des Bausteins erhebt keinen Anspruch auf Vollständigkeit

hinsichtlich der Implementierungsdetails; die genaue Funktionalität kann je nach verwendeter Laufzeitumgebung variieren.

AS_SPLIT_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AS_SPLIT_8`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AS_SPLIT_8_UNGATED** dient dazu, einen eingehenden *Application Specific* (AS)-Adapter-Datenstrom in acht identische Ausgänge zu verzweigen. Er wird als generischer Baustein (generic FB) bereitgestellt und ist speziell für die Aufteilung von Adapterdaten innerhalb einer IEC 61499-basierten Steuerungsapplikation konzipiert.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AS`	IN	Socket (Eingang)	Empfängt den zu verteilenden AS-Datenstrom.

adapter::types::unidirectional::AS	OUT1	Plug (Ausgang)	Erste Kopie des eingehenden AS-Datenstroms.
adapter::types::unidirectional::AS	OUT2	Plug (Ausgang)	Zweite Kopie des eingehenden AS-Datenstroms.
adapter::types::unidirectional::AS	OUT3	Plug (Ausgang)	Dritte Kopie.
adapter::types::unidirectional::AS	OUT4	Plug (Ausgang)	Vierte Kopie.
adapter::types::unidirectional::AS	OUT5	Plug (Ausgang)	Fünfte Kopie.
adapter::types::unidirectional::AS	OUT6	Plug (Ausgang)	Sechste Kopie.
adapter::types::unidirectional::AS	OUT7	Plug (Ausgang)	Siebte Kopie.
adapter::types::unidirectional::AS	OUT8	Plug (Ausgang)	Achte Kopie.

Funktionsweise

Der Baustein arbeitet rein adapterbasiert, ohne Nutzung von Ereignissen oder Daten-Ein-/Ausgängen. Der am Socket `IN` anliegende Adapter-Datenstrom wird intern auf alle acht Ausgangs-Plugs (`OUT1` … `OUT8`) gespiegelt. Jeder Ausgang gibt exakt dieselben Daten weiter, die am Eingang anliegen. Es findet keine Verzögerung, Filterung oder Modifikation statt. Die Verzweigung erfolgt passiv und ohne aktive Laufzeitlogik (kein Zustandsautomat).

Technische Besonderheiten

- **Reiner Adapterbaustein**: Der FB besitzt weder Ereignis- noch Datenschnittstellen im klassischen Sinne; die gesamte Datenübertragung erfolgt über die Adapterverbindungen.
- **Generische Implementierung**: Durch die Deklaration als generischer FB (`GenericClassName = 'GEN_AS_SPLIT'`) kann der Baustein für beliebige AS-Adapter-Typen eingesetzt werden, sofern der zugrunde liegende Adapter-Typ `unidirectional::AS` ist.
- **Keine Zustandsmaschine**: Aufgrund der fehlenden Ereignisse entfällt ein ECC (Execution Control Chart). Die Verteilung erfolgt kontinuierlich.
- **Einfaches 1:8-Splitting**: Optimiert für Situationen, in denen ein Signal auf mehrere nachfolgende Module verteilt werden muss.

Zustandsübersicht

Nicht zutreffend – der Baustein besitzt keine internen Zustände oder Ablaufsteuerung.

Anwendungsszenarien

- **Parallelverteilung von Sensordaten**: Ein einzelner Adapter, der z. B. Füllstandsdaten eines Tanks bereitstellt, wird auf acht überwachnde oder regelnde Funktionseinheiten aufgeteilt.
- **Signal-Multicasting in der Industrie 4.0**: Verteilung eines Steuerbefehls an mehrere Aktoren oder Subsysteme.
- **Testumgebungen** zur Simulation mehrfacher Empfänger eines Datenstroms.

Vergleich mit ähnlichen Bausteinen

- **AS_SPLIT_2 / AS_SPLIT_4**: Analoge Bausteine mit geringerer Ausgangszahl. AS_SPLIT_8_UNGATED bietet die maximale Verteilung in einem Schritt bei acht Ausgängen.
- **Ereignisbasierte Splitter (z. B. E_SPLIT)**: Diese arbeiten mit Ereignissen und verteilen diese zeitlich gesteuert. Der vorliegende Baustein verteilt dagegen dauerhaft die gesamten Adapterdaten ohne Ereigniskontrolle.
- **Daten-basierte Splitter (z. B. ANY_DISTRIBUTE)**: Trennen Datenwerte auf, benötigen aber zusätzliche Ereignisse. AS_SPLIT_8_UNGATED ist für reine Adapter-Weiterleitungen optimiert.
- `AS_SPLIT_8`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

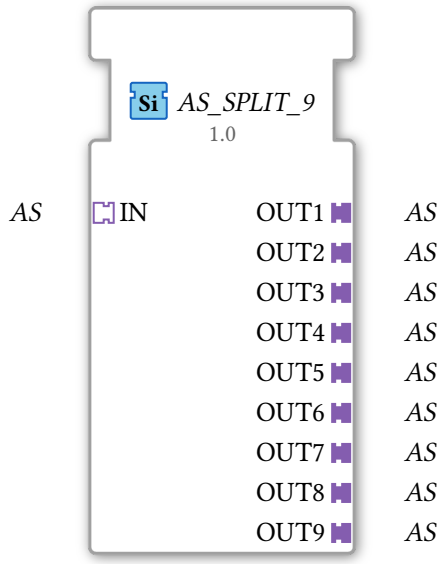
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AS_SPLIT_8_UNGATED** ist ein einfacher, aber nützlicher Adapterverteiler, der ohne zusätzliche Logik einen eingehenden AS-Datenstrom auf acht Ausgänge dupliziert. Sein generischer Charakter und die klare Struktur machen ihn zu einer soliden Lösung für Szenarien, in denen ein Adaptersignal mehrfach verwendet werden muss. Die Dokumentation des Bausteins erhebt keinen Anspruch auf Vollständigkeit hinsichtlich der Implementierungsdetails; die genaue Funktionalität kann je nach verwendeter Laufzeitumgebung variieren.

AS_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AS_SPLIT_9** dient dazu, einen eingehenden **AS-Adapter** (unidirektional) auf neun separate Ausgangsadapter aufzuteilen. Er wird als generischer Baustein ausgeführt und verteilt das am Socket `IN` anliegende AS-Signal identisch an alle neun Plug-Ausgänge (`OUT1` … `OUT9`). Damit eignet er sich ideal, um ein einziges AS-Signal mehrfach an verschiedene nachgeschaltete Komponenten weiterzuleiten.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine

Ereignis-Ausgänge

- Keine

Daten-Eingänge

- Keine

Daten-Ausgänge

- Keine

Adapter

Richtung	Name	Typ	Beschreibung
Socket	`IN`	`adapter::types::unidirectional::AS`	Eingangsadapter (AS-Schnittstelle)
Plug	`OUT1`	`adapter::types::unidirectional::AS`	Erster Ausgangsadapter
Plug	`OUT2`	`adapter::types::unidirectional::AS`	Zweiter Ausgangsadapter

Plug	`OUT3`	`adapter::types::unidirectional::AS`	Dritter Ausgangsadapter
Plug	`OUT4`	`adapter::types::unidirectional::AS`	Vierter Ausgangsadapter
Plug	`OUT5`	`adapter::types::unidirectional::AS`	Fünfter Ausgangsadapter
Plug	`OUT6`	`adapter::types::unidirectional::AS`	Sechster Ausgangsadapter
Plug	`OUT7`	`adapter::types::unidirectional::AS`	Siebter Ausgangsadapter
Plug	`OUT8`	`adapter::types::unidirectional::AS`	Achter Ausgangsadapter
Plug	`OUT9`	`adapter::types::unidirectional::AS`	Neunter Ausgangsadapter

Funktionsweise

Der Baustein empfängt über den Socket `IN` eine AS-Verbindung. Interne Logik leitet dieses eingehende Signal ohne Veränderung an alle neun Plug-Ausgänge (`OUT1` … `OUT9`) weiter. Dadurch stehen an allen Ausgängen stets die gleichen AS-Daten und/oder Ereignisse zur Verfügung. Eine separate Zustandsmaschine oder interne Verarbeitung existiert nicht – die Verteilung erfolgt direkt und ohne zeitliche Verzögerung.

Technische Besonderheiten

- **Generischer Baustein** – Der FB wird als **GEN_AS_SPLIT** klassifiziert und kann bei Bedarf parametrisiert werden (z. B. für eine variable Anzahl von Ausgängen).
- **Unidirektionale Adapter** – Sowohl Ein- als auch Ausgänge nutzen den Typ `adapter::types::unidirectional::AS`. Dies stellt sicher, dass die Daten-/Ereignisrichtung von IN zu OUT strikt eingehalten wird.
- Keine eigenen Ereignis- oder Dateneingänge erforderlich – die gesamte Schnittstelle besteht ausschließlich aus Adaptern.

Zustandsübersicht

Der FB besitzt keine explizite Zustandsmaschine (ECC). Die Verteilung erfolgt rein kombinatorisch, sobald der Eingangsadapter gültige Daten oder Ereignisse liefert.

Anwendungsszenarien

- **Signalverteilung** – Ein AS-Sensor oder eine Steuerung muss mehrere Aktoren oder überwachte Komponenten gleichzeitig versorgen.
- **Parallelbetrieb** – An ein AS-Netzwerk sollen mehrere Teilnehmer angeschlossen werden, ohne dass eine aktive Replikation erforderlich ist.
- **Test- und Simulationsumgebungen** – Ein ausgehendes AS-Signal wird auf mehrere Prüfmodule oder Analysewerkzeuge aufgeteilt.

Vergleich mit ähnlichen Bausteinen

- **AS_SPLIT_2**, **AS_SPLIT_4** u. Ä. – Diese Bausteine bieten eine geringere Anzahl von Ausgängen (2, 4 usw.) und sind für kleinere Verteilungen optimiert.
- **AS_MUX** – Ein Multiplexer, der mehrere Eingänge auf einen Ausgang legt; das Gegenteil des Splitters.
- **AS_COPY** – Kopiert ein Signal auf einen zweiten Ausgang; entspricht einem 1:2-Split.
- **AS_SPLIT_9** stellt mit neun Ausgängen die größte Standardvariante dar und deckt umfangreiche Verteilungsanforderungen ab.
- `AS_SPLIT_9_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

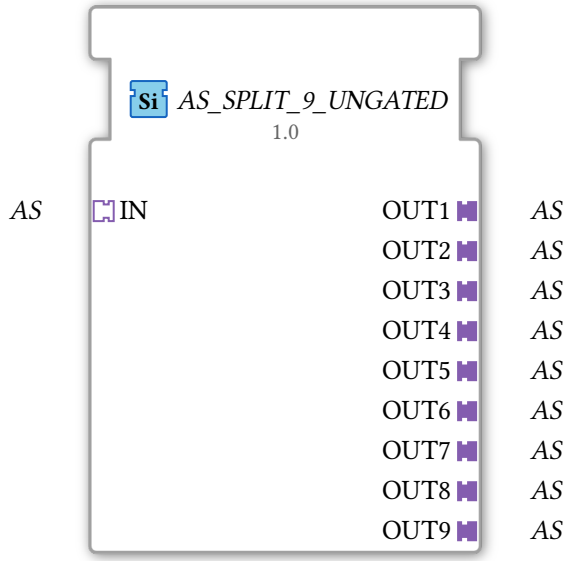
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der **AS_SPLIT_9** ist ein unkomplizierter, zuverlässiger Funktionsblock zur Vervielfältigung eines AS-Signals. Durch die reine Adapter-Schnittstelle und das Fehlen interner Logik eignet er sich besonders für Anwendungen, in denen ein Ausgangssignal auf viele Empfänger verteilt werden muss. Seine generische Natur erlaubt zudem eine einfache Anpassung an spezifische Projektanforderungen.

AS_SPLIT_9_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AS_SPLIT_9`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AS_SPLIT_9_UNGATED** dient dazu, einen eingehenden **AS-Adapter** (unidirektional) auf neun separate Ausgangsadapter aufzuteilen. Er wird als generischer Baustein ausgeführt und verteilt das am Socket `IN` anliegende AS-Signal identisch an alle neun Plug-Ausgänge (`OUT1` … `OUT9`). Damit eignet er sich ideal, um ein einziges AS-Signal mehrfach an verschiedene nachgeschaltete Komponenten weiterzuleiten.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine

Ereignis-Ausgänge

- Keine

Daten-Eingänge

- Keine

Daten-Ausgänge

- Keine

Adapter

Richtung	Name	Typ	Beschreibung
Socket	`IN`	`adapter::types::unidirectional::AS`	Eingangsadapter (AS-Schnittstelle)
Plug	`OUT1`	`adapter::types::unidirectional::AS`	Erster Ausgangsadapter
Plug	`OUT2`	`adapter::types::unidirectional::AS`	Zweiter Ausgangsadapter
Plug	`OUT3`	`adapter::types::unidirectional::AS`	Dritter Ausgangsadapter
Plug	`OUT4`	`adapter::types::unidirectional::AS`	Vierter Ausgangsadapter
Plug	`OUT5`	`adapter::types::unidirectional::AS`	Fünfter Ausgangsadapter
Plug	`OUT6`	`adapter::types::unidirectional::AS`	Sechster Ausgangsadapter
Plug	`OUT7`	`adapter::types::unidirectional::AS`	Siebter Ausgangsadapter
Plug	`OUT8`	`adapter::types::unidirectional::AS`	Achter Ausgangsadapter
Plug	`OUT9`	`adapter::types::unidirectional::AS`	Neunter Ausgangsadapter

Funktionsweise

Der Baustein empfängt über den Socket `IN` eine AS-Verbindung. Interne Logik leitet dieses eingehende Signal ohne Veränderung an alle neun Plug-Ausgänge (`OUT1` … `OUT9`) weiter. Dadurch stehen an allen Ausgängen stets die gleichen AS-Daten und/oder Ereignisse zur Verfügung. Eine separate Zustandsmaschine oder interne Verarbeitung existiert nicht – die Verteilung erfolgt direkt und ohne zeitliche Verzögerung.

Technische Besonderheiten

- **Generischer Baustein** – Der FB wird als **GEN_AS_SPLIT** klassifiziert und kann bei Bedarf parametrisiert werden (z. B. für eine variable Anzahl von Ausgängen).
- **Unidirektionale Adapter** – Sowohl Ein- als auch Ausgänge nutzen den Typ `adapter::types::unidirectional::AS`. Dies stellt sicher, dass die Daten-/Ereignisrichtung von IN zu OUT strikt eingehalten wird.
- Keine eigenen Ereignis- oder Dateneingänge erforderlich – die gesamte Schnittstelle besteht ausschließlich aus Adaptern.

Zustandsübersicht

Der FB besitzt keine explizite Zustandsmaschine (ECC). Die Verteilung erfolgt rein kombinatorisch, sobald der Eingangsadapter gültige Daten oder Ereignisse liefert.

Anwendungsszenarien

- **Signalverteilung** – Ein AS-Sensor oder eine Steuerung muss mehrere Aktoren oder überwachte Komponenten gleichzeitig versorgen.
- **Parallelbetrieb** – An ein AS-Netzwerk sollen mehrere Teilnehmer angeschlossen werden, ohne dass eine aktive Replikation erforderlich ist.
- **Test- und Simulationsumgebungen** – Ein ausgehendes AS-Signal wird auf mehrere Prüfmodule oder Analysewerkzeuge aufgeteilt.

Vergleich mit ähnlichen Bausteinen

- **AS_SPLIT_2**, **AS_SPLIT_4** u. Ä. – Diese Bausteine bieten eine geringere Anzahl von Ausgängen (2, 4 usw.) und sind für kleinere Verteilungen optimiert.
- **AS_MUX** – Ein Multiplexer, der mehrere Eingänge auf einen Ausgang legt; das Gegenteil des Splitters.
- **AS_COPY** – Kopiert ein Signal auf einen zweiten Ausgang; entspricht einem 1:2-Split.
- **AS_SPLIT_9_UNGATED** stellt mit neun Ausgängen die größte Standardvariante dar und deckt umfangreiche Verteilungsanforderungen ab.
- `AS_SPLIT_9_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- `AS_SPLIT_9`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

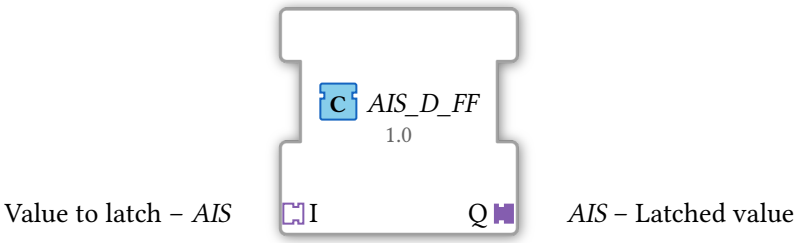
Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

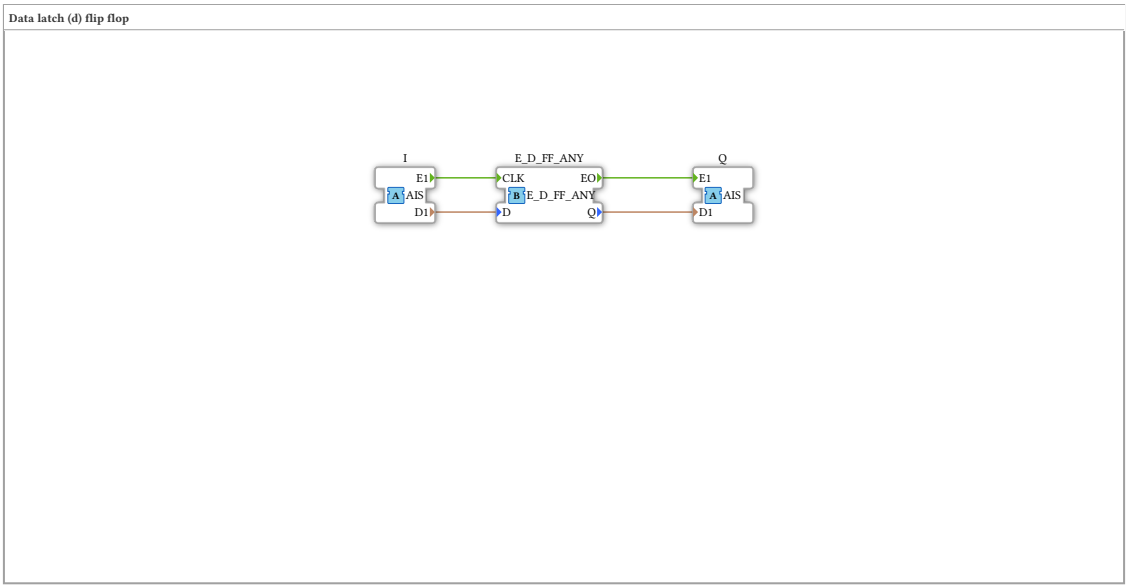
Fazit

Der **AS_SPLIT_9_UNGATED** ist ein unkomplizierter, zuverlässiger Funktionsblock zur Vervielfältigung eines AS-Signals. Durch die reine Adapter-Schnittstelle und das Fehlen interner Logik eignet er sich besonders für Anwendungen, in denen ein Ausgangssignal auf viele Empfänger verteilt werden muss. Seine generische Natur erlaubt zudem eine einfache Anpassung an spezifische Projektanforderungen.

STRING

AIS_D_FF





• – ○ • – ○ • – ○ •

Einleitung

Der Baustein `AIS_D_FF` realisiert ein taktflankengesteuertes D-Flipflop (Data Latch). Er dient dazu, einen Datenwert bei einem eingehenden Taktereignis zu speichern und auszugeben. Die Kommunikation erfolgt ausschließlich über AIS-Adapter (unidirectional), sodass der Baustein in adapterbasierte Komponentenarchitekturen eingebunden werden kann.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge. Das Taktsignal wird über den Socket-Adapter **I** (Ereignisport `E1`) bereitgestellt.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge. Das Ausgangsereignis wird über den Plug-Adapter **Q** (Ereignisport `E1`) gesendet.

Daten-Eingänge

Keine direkten Daten-Eingänge. Der zu übernehmende Datenwert wird über den Socket-Adapter **I** (Datenport `D1`) empfangen.

Daten-Ausgänge

Keine direkten Daten-Ausgänge. Der gespeicherte Wert wird über den Plug-Adapter **Q** (Datenport `D1`) ausgegeben.

Adapter

Adapter	Typ	Richtung	Beschreibung
I	`adapter::types::unidirectional::AIS`	Socket	Eingangsadapter: Taktsignal (Ereignis `E1`) und Dateneingang (Daten `D1`)
Q	`adapter::types::unidirectional::AIS`	Plug	Ausgangsadapter: Bestätigungsereignis (Ereignis `E1`) und gespeicherter Datenwert (Daten `D1`)

Funktionsweise

Der Baustein kapselt den IEC-61499-Standardbaustein `E_D_FF_ANY`. Bei einer positiven Flanke des Taktsignals (Ereignis `I.E1`) wird der aktuelle Wert von `I.D1` übernommen und intern im Flipflop gespeichert. Anschließend wird ein Ereignis am Ausgang `Q.E1` ausgelöst und der gespeicherte Wert am Port `Q.D1` bereitgestellt. Der Ausgangswert bleibt solange erhalten, bis das nächste Taktereignis eintrifft.

Technische Besonderheiten

- **Reine Adapter-Schnittstelle**: Der Baustein besitzt keine direkten Event- oder Datenports, sondern verwendet ausschließlich AIS-Adapter. Dies ermöglicht eine flexible Kopplung mit anderen adapterbasierten Komponenten.
- **Generischer Datentyp**: Der interne `E_D_FF_ANY`-Baustein unterstützt beliebige Datentypen. Die Datenbreite und der Typ werden durch die verwendete Adapter-Implementierung bestimmt.
- **Ereignisgesteuerte Datenweitergabe**: Eine Aktualisierung des Ausgangs erfolgt nur bei einem eingehenden Taktereignis. Ohne Taktereignis bleibt der Ausgang unverändert.

Zustandsübersicht

Da der Baustein das Verhalten eines D-Flipflops implementiert, können zwei Zustände unterschieden werden:

Zustand	Beschreibung
Ruhezustand	Der gespeicherte Wert bleibt konstant. Es wird kein Ausgangsereignis erzeugt.
Übernahme	Bei Eintreffen eines Taktereignisses wird der aktuelle Eingangswert übernommen, der Ausgang wird aktualisiert und ein Ausgangsereignis gesendet.

Das Flipflop ist flankengesteuert (steigende Flanke) und nicht pegelgesteuert.

Anwendungsszenarien

- **Datenpuffer in Adapter-Netzwerken**: Zwischenspeichern von Werten, die nur bei bestimmten Ereignissen gültig sind (z. B. Taktsignale von Sensoren).
- **Synchronisation**: Abgleich von Daten zwischen Komponenten, die über asynchrone Ereignisse kommunizieren.
- **Register in Steuerungsanwendungen**: Aufbau von Schieberegistern oder einfachen Speicherzellen in IEC-61499-Systemen, die AIS-Adapter verwenden.

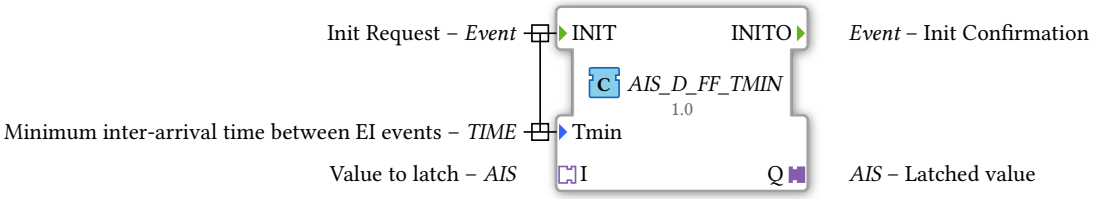
Vergleich mit ähnlichen Bausteinen

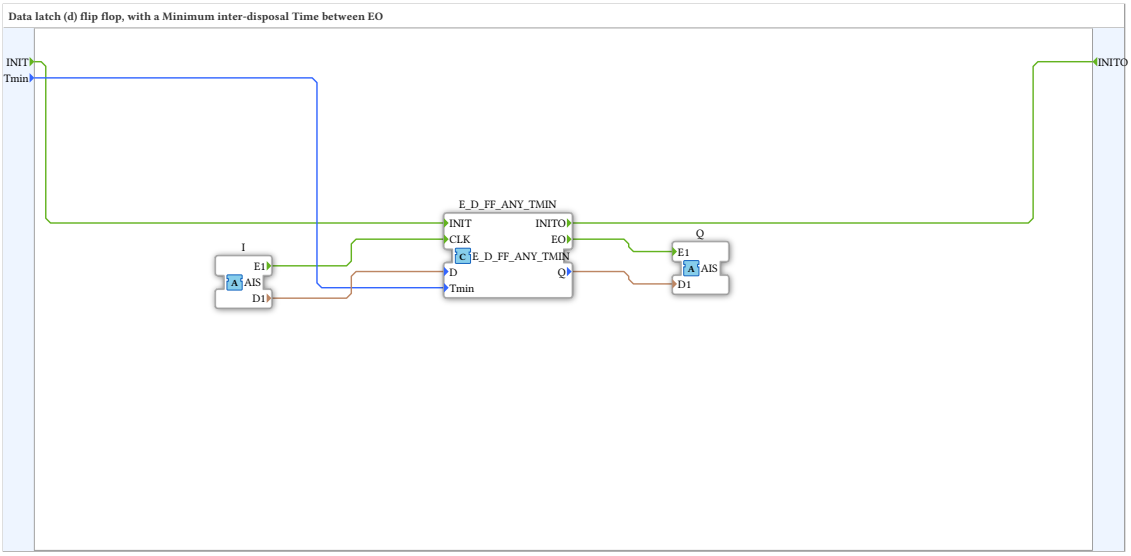
- **`E_D_FF_ANY`** – Das direkte Funktionsblock-Äquivalent mit standardmäßigen Event- und Datenports. `AIS_D_FF` kapselt diesen Baustein in eine adapterbasierte Schnittstelle.
- **SR-Flipflop** – Speichert einen binären Zustand über Set- und Reset-Eingänge, während `AIS_D_FF` einen beliebigen Datenwert über ein Taktsignal speichert.
- **Latch (pegelgesteuert)** – Ein Latch übernimmt den Eingangswert, solange das Taktsignal aktiv ist; `AIS_D_FF` reagiert dagegen nur auf eine Flanke und ist damit unempfindlicher gegen Rauschen und Glitches.

Fazit

`AIS_D_FF` ist ein nützlicher Baustein, um ein flankengesteuertes D-Flipflop in adapterbasierten Architekturen einzusetzen. Durch die Kapselung des bewährten `E_D_FF_ANY`-Funktionsblocks bietet er zuverlässige Speicherfunktionalität und lässt sich nahtlos in bestehende AIS-Kommunikationsstrukturen integrieren.

AIS_D_FF_TMIN





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **AIS_D_FF_TMIN** realisiert ein datenabhängiges Flip-Flop (D-Flip-Flop) mit einer minimalen Zwischenzeit (*minimum inter-disposal time*) zwischen zwei aufeinanderfolgenden Ausgabeereignissen (EO). Er dient dazu, einen eingehenden Datenwert über einen AIS-Adapter zu latchen und nur dann auszugeben, wenn die durch den Parameter `Tmin` festgelegte Mindestzeit seit dem letzten EO verstrichen ist. Dadurch werden zu schnelle Signalwechsel unterdrückt und eine definierte Signalverzögerung sichergestellt.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar
`INIT`	EInit	Initialisierungsanforderung (mit `Tmin`)

Ereignis-Ausgänge

Ereignis	Typ	Kommentar
`INITO`	EInit	Initialisierungsbestätigung

Daten-Eingänge

Variable	Typ	Kommentar
`Tmin`	TIME	Mindestzeitabstand zwischen zwei EO-Ereignissen

Daten-Ausgänge

Der FB besitzt keine eigenen Datenausgänge. Die Ausgabe erfolgt ausschließlich über den Adapter-Plug (siehe unten).

Adapter

Richtung	Bezeichnung	Adaptertyp	Kommentar
Plug (Ausgang)	`Q`	`adapter::types::unidirectional::AIS`	Gelatchter Ausgabewert
Socket (Eingang)	`I`	`adapter::types::unidirectional::AIS`	Zu latchnder Eingabewert

Der Adapter `AIS` ist unidirektional und liefert über die Ereignis-/Datenschnittstellen `E1`/`D1` die eigentlichen Signale.

Funktionsweise

Der Baustein besteht intern aus einem geschachtelten FB `E_D_FF_ANY_TMIN` (aus der IEC-61499-Bibliothek).
Die Verarbeitung läuft wie folgt ab:

1. **Initialisierung**: Beim Ereignis `INIT` wird der Parameter `Tmin` an den internen FB weitergeleitet und dieser initialisiert. Anschließend wird `INITO` ausgelöst.
2. **Latch-Vorgang**: Ein am **Socket `I`** eintreffendes Ereignis `E1` (verbunden mit dem internen `CLK`) übernimmt den gleichzeitig anliegenden Datenwert `D1` (verbunden mit `D`) in das Flip-Flop.
3. **Minimale Verzögerung**: Der interne FB prüft, ob seit dem letzten Ausgangsereignis (`EO`) mindestens die Zeit `Tmin` vergangen ist.
   - Ist die Bedingung erfüllt, wird das gelatchte Datum über den **Plug `Q`** ausgegeben (Ereignis `E1` und Datenwert `D1`).
   - Andernfalls wird der Wert zurückgehalten, bis die geforderte Wartezeit abgelaufen ist.

Somit wird sichergestellt, dass Ausgabeereignisse nicht schneller als durch `Tmin` vorgegeben erfolgen.

Technische Besonderheiten

- **Adaptor-basierte Ein-/Ausgabe**: Die Signale werden nicht als separate Ein-/Ausgangsvariablen, sondern über standardisierte unidirektionale AIS-Adapter geführt. Dies ermöglicht eine flexible Verschaltung mit anderen Bausteinen oder Subsystemen.
- **Einhaltung einer minimalen Zwischenzeit**: Die Verzögerung verhindert „Rauschen“ oder zu schnelle Signalwechsel, die in zeitkritischen Systemen zu Fehlfunktionen führen können.
- **Interner IEC-61499-Standard-FB**: Die Implementierung nutzt den bibliothekseigenen `E_D_FF_ANY_TMIN`, was eine hohe Zuverlässigkeit und Wiederverwendbarkeit gewährleistet.

Zustandsübersicht

Der FB selbst besitzt keine explizit modellierten Zustände in der XML-Definition. Das Verhalten wird vollständig durch den internen FB `E_D_FF_ANY_TMIN` bestimmt. Dieser durchläuft typischerweise folgende Phasen (vereinfacht):

- **Idle**: Warten auf ein Taktereignis am Eingang `I.E1`.
- **Sampling**: Beim Taktereignis wird der aktuelle Datenwert `I.D1` übernommen.
- **Verzögerung**: Nach der Übernahme wird die seit dem letzten `EO` verstrichene Zeit mit `Tmin` verglichen. Bei Unterschreitung wird in einen Wartezustand gewechselt.
- **Ausgabe**: Nach Ablauf der Mindestzeit wird das gelatchte Datum über `Q.E1` und `Q.D1` ausgegeben.

Anwendungsszenarien

- **Signalglättung**: Unterdrückung von Prellern oder kurzen Impulsen an digitalen Eingängen.
- **Zyklussteuerung**: Sicherstellung, dass ein Ausgangssignal nur in einem definierten Mindestabakt (z. B. alle 100 ms) aktualisiert wird.
- **Schnittstellen-Pufferung**: Puffern von Werten zwischen unterschiedlich schnellen Systemkomponenten, wobei eine minimale Aktualisierungsrate eingehalten werden muss.
- **Automatisierungstechnik**: Steuerung von Aktoren, die mechanische Totzeiten oder Mindestschaltintervalle erfordern.

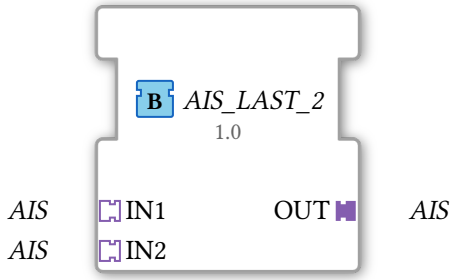
Vergleich mit ähnlichen Bausteinen

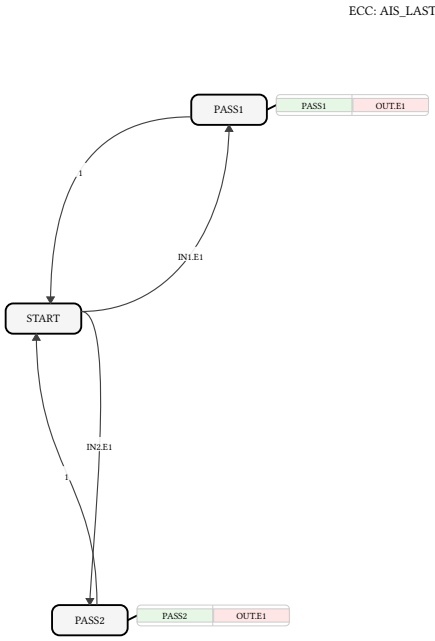
- **Einfaches D-Flip-Flop (ohne Zeitsteuerung)**: Ein Standard-D-Flip-Flop gibt den gelatchten Wert sofort mit dem nächsten Taktereignis aus – ohne Verzögerungsbegrenzung. `AIS_D_FF_TMIN` fügt die zusätzliche Zeitbedingung hinzu.
- **Monostabile Kippstufe**: Ein monostabiler Baustein erzeugt einen Impuls fester Dauer; der hier beschriebene FB hält dagegen keine Impulsdauer, sondern einen Mindestabstand zwischen Ausgaben ein.
- **IEC-61499 `E_D_FF_ANY`**: Ist die Basis ohne Zeitbeschränkung. `E_D_FF_ANY_TMIN` erweitert diese Variante um den `Tmin`-Parameter.

Fazit

Der `AIS_D_FF_TMIN` bietet eine robuste Möglichkeit, digitale Signale mit einer einstellbaren minimalen Ausgabezeitverzögerung zu latchen. Durch die Verwendung von AIS-Adaptern und internen IEC-61499-Standard-Bausteinen ist er sowohl flexibel einsetzbar als auch zuverlässig. Er eignet sich besonders für Automatisierungsaufgaben, bei denen schnelle Signalwechsel unterdrückt oder definierte Mindestpausen zwischen Ausgabeereignissen eingehalten werden müssen.

AIS_LAST_2





Einleitung

Der AIS_LAST_2 Funktionsblock führt zwei gleichartige STRING-Adapter-Signale (**IN1**, **IN2**) auf einen gemeinsamen Ausgang (**OUT**) zusammen, nach dem Prinzip **Last Writer Wins**: welcher der beiden Sockets zuletzt ein Ereignis geschrieben hat, gewinnt, und sein Datenwert wird sofort und unverändert an OUT durchgereicht. Es handelt sich ausdrücklich nicht um ein naives Zusammenführen (z. B. Mittelwert oder ODER-Verknüpfung), sondern um eine reine Arbitrierung nach zeitlicher Reihenfolge.

Schnittstellenstruktur

Adapter

Eingangsadapter (Sockets):

- **IN1**: STRING-Adapter (unidirectional) - erste Signalquelle
- **IN2**: STRING-Adapter (unidirectional) - zweite Signalquelle

Ausgangsadapter (Plug):

- **OUT**: STRING-Adapter (unidirectional) - zusammengeführtes Signal

Funktionsweise

Der Baustein ist als einfacher Basic-FB mit einer 3-Zustands-ECC umgesetzt (START, PASS1, PASS2). Trifft an IN1 ein Ereignis ein, wechselt die ECC nach PASS1: der aktuelle Datenwert von IN1 wird nach OUT kopiert und das Adapter-Ereignis an OUT gesendet. Trifft an IN2 ein Ereignis ein, geschieht dasselbe symmetrisch über PASS2. Nach jeder Durchreichung kehrt die ECC sofort wieder nach START zurück und ist bereit für das nächste Ereignis von IN1 oder IN2.

Technische Besonderheiten

- Last-Writer-Wins-Semantik statt naiver Datenverknüpfung
- Einfacher Basic-FB (ECC mit 3 Zuständen), kein zusätzlicher Zwischenspeicher nötig
- Keine Signalverzögerung: der Wert wird im selben Zyklus durchgereicht, in dem das auslösende Ereignis eintrifft
- Beide Eingänge sind gleichberechtigt - es gibt keine feste Priorität außer der zeitlichen Reihenfolge

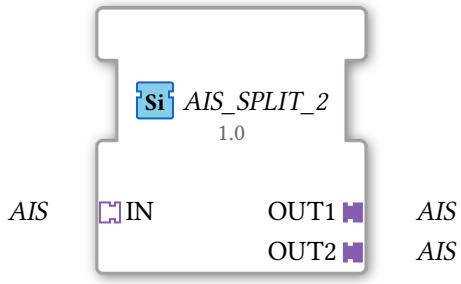
Anwendungsszenarien

- Zusammenführen zweier gleichwertiger STRING-Signalquellen zu einem einzigen nachgeschalteten Adapterpfad (z. B. zwei redundante Sensoren, oder ein manueller und ein automatischer Sollwertgeber)
- Arbitrierung zwischen zwei Bedienstellen, die denselben STRING-Wert schreiben dürfen
- Vereinfachung von Netzwerken, die sonst ein `F_SEL`/`E_RS`-Konstrukt inline verdrahten müssten

⚖ Vergleich mit ähnlichen Bausteinen

AIS_LAST_2 ist die Umkehrung von `AIS_SPLIT_2`: AIS_SPLIT_2 verteilt 1 Eingang auf 2 Ausgänge, AIS_LAST_2 führt 2 Eingänge auf 1 Ausgang zusammen. Für Mischbetrieb zwischen einem Daten- und einem reinen Ereignisadapter (z. B. AX + AE) ist stattdessen `AX_AE_MERGE` gedacht, das nur das Ereignis mischt, ohne den Datenwert zu verändern.

AIS_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AIS_SPLIT_2** ist ein generischer Adapter-Baustein, der einen eingehenden unidirektionalen AIS-Adapter (Application Interface Socket) auf zwei identische, ausgehende AIS-Adapter aufteilt. Er dient als passiver Splitter ohne eigene Logik oder Ereignisverarbeitung.

Schnittstellenstruktur

Ereignis-Eingänge

Nicht vorhanden.

Ereignis-Ausgänge

Nicht vorhanden.

Daten-Eingänge

Nicht vorhanden.

Daten-Ausgänge

Nicht vorhanden.

Adapter

- **IN** (Socket): Typ `adapter::types::unidirectional::AIS` – Eingangsschnittstelle für den zu verteilenden Adapter.
- **OUT1** (Plug): Typ `adapter::types::unidirectional::AIS` – erster Ausgang, erhält alle Daten/Ereignisse von IN.
- **OUT2** (Plug): Typ `adapter::types::unidirectional::AIS` – zweiter Ausgang, erhält identische Daten/Ereignisse wie OUT1.

Funktionsweise

Der Baustein leitet den am Socket **IN** anliegenden Adapter unverändert und ohne Verzögerung an beide Plugs **OUT1** und **OUT2** weiter. Jegliche über den AIS-Adapter definierten Ereignisse und Daten werden parallel auf beide Ausgänge kopiert. Der FB selbst besitzt keine eigene Verarbeitungslogik, Zustände oder Zeitverhalten.

Technische Besonderheiten

- **Generische Parametrierung**: Der konkrete AIS-Adapter-Typ wird erst zur Projektierungszeit über das Attribut `eclipse4diac::core::GenericClassName` festgelegt (Standardwert: `'GEN_AIS_SPLIT'`). Dadurch kann der Baustein für verschiedene AIS-Varianten wiederverwendet werden.
- **Keine Eigenzustände**: Da keine ECC-Zustandsmaschine oder Data-/Event-Ein-/Ausgänge vorhanden sind, ist der FB rein passiv und verhält sich wie eine „Verdrahtungshilfe“.
- **Unidirektional**: Die Adapter sind als unidirektional deklariert – eine Datenflussrichtung von IN zu OUT1/OUT2 ist implizit.

Zustandsübersicht

Der AIS_SPLIT_2 besitzt keine eigene Zustandsmaschine. Der Betriebszustand wird ausschließlich durch die jeweils angeschlossenen AIS-Adapter bestimmt.

Anwendungsszenarien

- **Parallelverarbeitung**: Ein AIS-Signal soll gleichzeitig von zwei nachfolgenden Funktionsblöcken verarbeitet werden (z. B. Visualisierung und Steuerung).
- **Monitoring**: Abzweigen eines AIS-Signals zu einer Überwachungsinstanz, ohne das Originalsignal zu beeinflussen.
- **Generische Aufteilung**: Einsatz in Bibliotheken, wo die genaue AIS-Struktur erst zur Laufzeit oder Konfigurationszeit festgelegt wird.

Vergleich mit ähnlichen Bausteinen

- **AIS_SPLIT_4**: Bietet vier Ausgänge statt zwei; ansonsten gleiche Funktionsweise.
- **Ereignis-/Datensplitter**: Andere Splitter arbeiten mit einfachen Events oder Daten (z. B. F_SPLIT), während AIS_SPLIT_2 komplette Adapter-Schnittstellen aufteilt.
- **AIS_MERGE**: Das Gegenstück, das zwei AIS-Eingänge zu einem Ausgang zusammenführt.
- `AIS_SPLIT_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

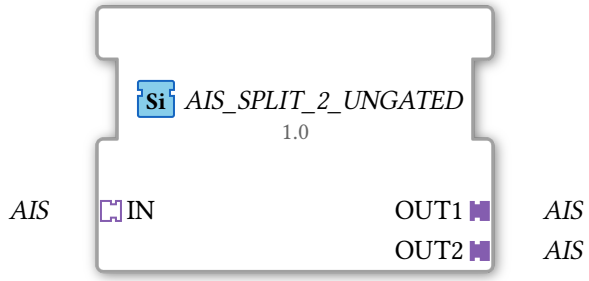
Der AIS_SPLIT_2 ist ein minimalistischer, generisch einsetzbarer Adapter-Splitter, der die flexible Verzweigung von AIS-Schnittstellen in 4diac-Projekten ermöglicht. Durch die Parametrierung über das GenericClassName-Attribut kann er ohne Anpassung des Quellcodes für verschiedene AIS-Typen verwendet werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AIS_SPLIT_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AIS_SPLIT_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AIS_SPLIT_2_UNGATED** ist ein generischer Adapter-Baustein, der einen eingehenden unidirektionalen AIS-Adapter (Application Interface Socket) auf zwei identische, ausgehende AIS-Adapter aufteilt. Er dient als passiver Splitter ohne eigene Logik oder Ereignisverarbeitung.

Schnittstellenstruktur

Ereignis-Eingänge

Nicht vorhanden.

Ereignis-Ausgänge

Nicht vorhanden.

Daten-Eingänge

Nicht vorhanden.

Daten-Ausgänge

Nicht vorhanden.

Adapter

- **IN** (Socket): Typ `adapter::types::unidirectional::AIS` – Eingangsschnittstelle für den zu verteilenden Adapter.
- **OUT1** (Plug): Typ `adapter::types::unidirectional::AIS` – erster Ausgang, erhält alle Daten/Ereignisse von IN.
- **OUT2** (Plug): Typ `adapter::types::unidirectional::AIS` – zweiter Ausgang, erhält identische Daten/Ereignisse wie OUT1.

Funktionsweise

Der Baustein leitet den am Socket **IN** anliegenden Adapter unverändert und ohne Verzögerung an beide Plugs **OUT1** und **OUT2** weiter. Jegliche über den AIS-Adapter definierten Ereignisse und Daten werden parallel auf beide Ausgänge kopiert. Der FB selbst besitzt keine eigene Verarbeitungslogik, Zustände oder Zeitverhalten.

Technische Besonderheiten

- **Generische Parametrierung**: Der konkrete AIS-Adapter-Typ wird erst zur Projektierungszeit über das Attribut `eclipse4diac::core::GenericClassName` festgelegt (Standardwert: `'GEN_AIS_SPLIT'`). Dadurch kann der Baustein für verschiedene AIS-Varianten wiederverwendet werden.
- **Keine Eigenzustände**: Da keine ECC-Zustandsmaschine oder Data-/Event-Ein-/Ausgänge vorhanden sind, ist der FB rein passiv und verhält sich wie eine „Verdrahtungshilfe“.
- **Unidirektional**: Die Adapter sind als unidirektional deklariert – eine Datenflussrichtung von IN zu OUT1/OUT2 ist implizit.

Zustandsübersicht

Der AIS_SPLIT_2_UNGATED besitzt keine eigene Zustandsmaschine. Der Betriebszustand wird ausschließlich durch die jeweils angeschlossenen AIS-Adapter bestimmt.

Anwendungsszenarien

- **Parallelverarbeitung**: Ein AIS-Signal soll gleichzeitig von zwei nachfolgenden Funktionsblöcken verarbeitet werden (z. B. Visualisierung und Steuerung).
- **Monitoring**: Abzweigen eines AIS-Signals zu einer Überwachungsinstanz, ohne das Originalsignal zu beeinflussen.
- **Generische Aufteilung**: Einsatz in Bibliotheken, wo die genaue AIS-Struktur erst zur Laufzeit oder Konfigurationszeit festgelegt wird.

Vergleich mit ähnlichen Bausteinen

- **AIS_SPLIT_4**: Bietet vier Ausgänge statt zwei; ansonsten gleiche Funktionsweise.
- **Ereignis-/Datensplitter**: Andere Splitter arbeiten mit einfachen Events oder Daten (z. B. F_SPLIT), während AIS_SPLIT_2_UNGATED komplette Adapter-Schnittstellen aufteilt.
- **AIS_MERGE**: Das Gegenstück, das zwei AIS-Eingänge zu einem Ausgang zusammenführt.
- `AIS_SPLIT_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

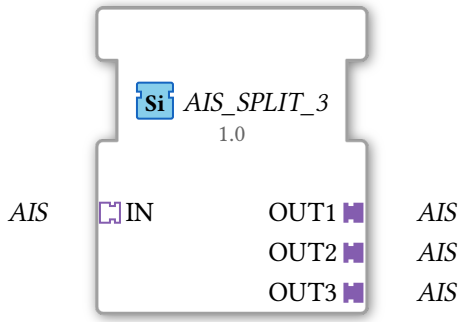
Fazit

Der AIS_SPLIT_2_UNGATED ist ein minimalistischer, generisch einsetzbarer Adapter-Splitter, der die flexible Verzweigung von AIS-Schnittstellen in 4diac-Projekten ermöglicht. Durch die Parametrierung über das GenericClassName-Attribut kann er ohne Anpassung des Quellcodes für verschiedene AIS-Typen verwendet werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AIS_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AIS_SPLIT_3** dient dazu, einen eingehenden AIS-Datenstrom auf drei identische Ausgänge zu verteilen. Er ist als generischer Baustein realisiert und ermöglicht eine einfache Vervielfachung von AIS-Informationen in Steuerungssystemen nach IEC 61499.

Schnittstellenstruktur

Ereignis-Eingänge

Nicht vorhanden – der Baustein arbeitet rein daten- und adaptergesteuert.

Ereignis-Ausgänge

Nicht vorhanden.

Daten-Eingänge

Nicht vorhanden – die Eingangsdaten werden ausschließlich über den Adapter-Socket bereitgestellt.

Daten-Ausgänge

Nicht vorhanden – die Ausgangsdaten werden ausschließlich über die Adapter-Plugs ausgegeben.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AIS`	IN	Socket (Eingang)	Empfängt einen AIS-Datenstrom.
`adapter::types::unidirectional::AIS`	OUT1	Plug (Ausgang)	Erster Ausgangskanal – identische Kopie des Eingangs.
`adapter::types::unidirectional::AIS`	OUT2	Plug (Ausgang)	Zweiter Ausgangskanal – identische Kopie des Eingangs.
`adapter::types::unidirectional::AIS`	OUT3	Plug (Ausgang)	Dritter Ausgangskanal – identische Kopie des Eingangs.

Funktionsweise

Der Baustein nimmt über den **IN**-Socket einen AIS-Datenstrom entgegen und leitet diesen ohne Verzögerung oder Manipulation an die drei Ausgangsadapter **OUT1**, **OUT2** und **OUT3** weiter. Jeder Ausgang erhält identische Daten. Es findet keine Pufferung, Filterung oder Protokollumsetzung statt – der Baustein fungiert als reiner Verteiler (Splitter) auf Adapterebene.

Technische Besonderheiten

- **Generischer Typ**: Der Baustein ist mit dem generischen Klassennamen `GEN_AIS_SPLIT` versehen, sodass er in verschiedenen Kontexten und mit unterschiedlichen AIS-Datentypen wiederverwendet werden kann.
- **Unidirektionale Adapter**: Sowohl Ein- als auch Ausgänge verwenden den unidirektionalen Adapter `adapter::types::unidirectional::AIS`, was eine klare Datenflussrichtung gewährleistet.
- **Keine ereignisgesteuerte Synchronisation**: Da keine Ereignis-Ein-/Ausgänge vorhanden sind, erfolgt die Datenweitergabe sofort und ohne Triggerung durch externe Events.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten. Sein Verhalten entspricht einer ständig aktiven Durchschleifverbindung: Solange der Eingangsadapter Daten liefert, werden diese auf allen drei Ausgängen ausgegeben. Es gibt keine internen Zustände oder Moden.

Anwendungsszenarien

- **Verteilen von AIS-Daten an mehrere Verbraucher** – z. B. gleichzeitige Visualisierung, Logging und Weiterverarbeitung.
- **Redundante Datenbereitstellung** – Einspeisung in verschiedene Teilsysteme ohne zusätzliche Belastung der Datenquelle.
- **Test- und Diagnoseaufbauten** – paralleles Überwachen eines AIS-Streams an mehreren Stellen.

Vergleich mit ähnlichen Bausteinen

Baustein	Anzahl Ausgänge	Besonderheit
AIS_SPLIT_2	2	Verteilt auf zwei Kanäle.
AIS_SPLIT_3	**3**	**Standard-Splitter mit drei Ausgängen.**
AIS_SPLIT_N	variabel	Generische Version mit konfigurierbarer Ausgangsanzahl (sofern verfügbar).

Der AIS_SPLIT_3 füllt die Lücke zwischen dem einfachen 2-Wege-Splitter und einem vollständig parametrierbaren Splitter.

- `AIS_SPLIT_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

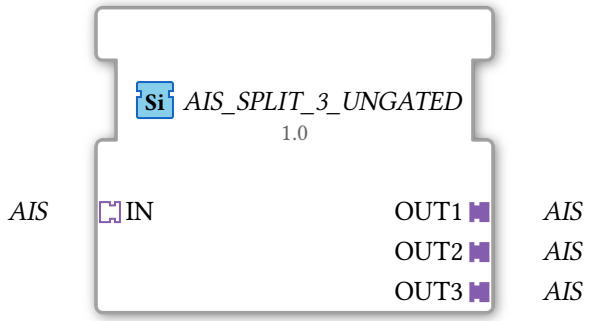
Der **AIS_SPLIT_3** ist ein einfacher, aber effektiver Baustein zur Vervielfachung eines AIS-Datenstroms. Dank seiner generischen Natur und der klaren adapterbasierten Schnittstelle lässt er sich leicht in größere 4diac-Projekte integrieren. Für Anwendungen, die eine mehrfache Verteilung ohne Datenänderung erfordern, bietet er eine robuste und wartbare Lösung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AIS_SPLIT_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AIS_SPLIT_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AIS_SPLIT_3_UNGATED** dient dazu, einen eingehenden AIS-Datenstrom auf drei identische Ausgänge zu verteilen. Er ist als generischer Baustein realisiert und ermöglicht eine einfache Vervielfachung von AIS-Informationen in Steuerungssystemen nach IEC 61499.

Schnittstellenstruktur

Ereignis-Eingänge

Nicht vorhanden – der Baustein arbeitet rein daten- und adaptergesteuert.

Ereignis-Ausgänge

Nicht vorhanden.

Daten-Eingänge

Nicht vorhanden – die Eingangsdaten werden ausschließlich über den Adapter-Socket bereitgestellt.

Daten-Ausgänge

Nicht vorhanden – die Ausgangsdaten werden ausschließlich über die Adapter-Plugs ausgegeben.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AIS`	IN	Socket (Eingang)	Empfängt einen AIS-Datenstrom.
`adapter::types::unidirectional::AIS`	OUT1	Plug (Ausgang)	Erster Ausgangskanal – identische Kopie des Eingangs.
`adapter::types::unidirectional::AIS`	OUT2	Plug (Ausgang)	Zweiter Ausgangskanal – identische Kopie des Eingangs.
`adapter::types::unidirectional::AIS`	OUT3	Plug (Ausgang)	Dritter Ausgangskanal – identische Kopie des Eingangs.

Funktionsweise

Der Baustein nimmt über den **IN**-Socket einen AIS-Datenstrom entgegen und leitet diesen ohne Verzögerung oder Manipulation an die drei Ausgangsadapter **OUT1**, **OUT2** und **OUT3** weiter. Jeder Ausgang erhält identische Daten. Es findet keine Pufferung, Filterung oder Protokollumsetzung statt – der Baustein fungiert als reiner Verteiler (Splitter) auf Adapterebene.

Technische Besonderheiten

- **Generischer Typ**: Der Baustein ist mit dem generischen Klassennamen `GEN_AIS_SPLIT` versehen, sodass er in verschiedenen Kontexten und mit unterschiedlichen AIS-Datentypen wiederverwendet werden kann.
- **Unidirektionale Adapter**: Sowohl Ein- als auch Ausgänge verwenden den unidirektionalen Adapter `adapter::types::unidirectional::AIS`, was eine klare Datenflussrichtung gewährleistet.
- **Keine ereignisgesteuerte Synchronisation**: Da keine Ereignis-Ein-/Ausgänge vorhanden sind, erfolgt die Datenweitergabe sofort und ohne Triggerung durch externe Events.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten. Sein Verhalten entspricht einer ständig aktiven Durchschleifverbindung: Solange der Eingangsadapter Daten liefert, werden diese auf allen drei Ausgängen ausgegeben. Es gibt keine internen Zustände oder Moden.

Anwendungsszenarien

- **Verteilen von AIS-Daten an mehrere Verbraucher** – z. B. gleichzeitige Visualisierung, Logging und Weiterverarbeitung.
- **Redundante Datenbereitstellung** – Einspeisung in verschiedene Teilsysteme ohne zusätzliche Belastung der Datenquelle.
- **Test- und Diagnoseaufbauten** – paralleles Überwachen eines AIS-Streams an mehreren Stellen.

Vergleich mit ähnlichen Bausteinen

Baustein	Anzahl Ausgänge	Besonderheit
AIS_SPLIT_2	2	Verteilt auf zwei Kanäle.
AIS_SPLIT_3_UNGATED	**3**	**Standard-Splitter mit drei Ausgängen.**
AIS_SPLIT_N	variabel	Generische Version mit konfigurierbarer Ausgangsanzahl (sofern verfügbar).

Der AIS_SPLIT_3_UNGATED füllt die Lücke zwischen dem einfachen 2-Wege-Splitter und einem vollständig parametrierbaren Splitter.

- `AIS_SPLIT_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

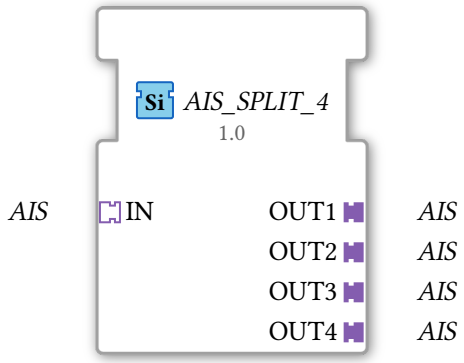
Fazit

Der **AIS_SPLIT_3_UNGATED** ist ein einfacher, aber effektiver Baustein zur Vervielfachung eines AIS-Datenstroms. Dank seiner generischen Natur und der klaren adapterbasierten Schnittstelle lässt er sich leicht in größere 4diac-Projekte integrieren. Für Anwendungen, die eine mehrfache Verteilung ohne Datenänderung erfordern, bietet er eine robuste und wartbare Lösung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AIS_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AIS_SPLIT_4** dient dazu, ein eingehendes AIS-Adaptersignal (Typ `adapter::types::unidirectional::AIS`) auf vier separate Ausgänge zu verteilen. Er realisiert eine 1-zu-4-Aufteilung ohne Veränderung der Daten. Der Baustein ist generisch ausgeführt und kann über den generischen Klassennamen an unterschiedliche Signaltypen angepasst werden.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

- **Socket (Eingang):** `IN` – Typ `adapter::types::unidirectional::AIS`
- **Plugs (Ausgänge):**
  - `OUT1` – Typ `adapter::types::unidirectional::AIS`
  - `OUT2` – Typ `adapter::types::unidirectional::AIS`
  - `OUT3` – Typ `adapter::types::unidirectional::AIS`
  - `OUT4` – Typ `adapter::types::unidirectional::AIS`

Funktionsweise

Der Baustein leitet das am Socket `IN` anliegende Adaptersignal unverändert an alle vier Ausgangsadapter (`OUT1` … `OUT4`) weiter. Es findet keine Datenmanipulation, Filterung oder Pufferung statt. Jeder Ausgang entspricht exakt dem Eingangssignal. Die Verteilung erfolgt asynchron und ohne Ereignissteuerung – die Adapterschnittstelle übernimmt die Datenweitergabe entsprechend ihrer Kommunikationslogik.

Technische Besonderheiten

- **Generischer Typ:** Der Baustein ist als generischer Funktionsblock ausgeführt. Über das Attribut `eclipse4diac::core::GenericClassName` (Wert `'GEN_AIS_SPLIT'`) kann der eigentliche Signaltyp zur Erstellungszeit festgelegt werden.
- **Typ-Hash:** Ein Leerstring (`''`) für das Attribut `eclipse4diac::core::TypeHash` weist darauf hin, dass der Baustein bei Bedarf dynamisch konfiguriert wird.
- **Kompaktheit:** Da weder Ereignis- noch Dateneingänge/-ausgänge existieren, ist der Baustein rein adapterbasiert und frei von zeitlichen Abhängigkeiten.

Zustandsübersicht

Der Baustein besitzt keine Zustandsmaschine – er arbeitet kontinuierlich und zustandslos. Es gibt keine internalen Zustände oder Ereignisse, die geschaltet werden müssten. Die Funktionalität erschöpft sich in der passiven Durchleitung des Adaptersignals.

Anwendungsszenarien

- **Signalverteilung in Steuerungsarchitekturen:** Ein von einer Quelle (z. B. Sensor) bereitgestelltes AIS-Signal soll mehreren nachgeschalteten Verarbeitungseinheiten oder Anzeigen zur Verfügung gestellt werden.
- **Redundante Datenweitergabe:** In sicherheitskritischen Systemen kann das Eingangssignal an mehrere unabhängige Pfade geleitet werden, ohne es zu duplizieren.
- **Test- und Debugumgebungen:** Ein Signal wird parallel an die eigentliche Logik und an ein Analyse-/Protokolltool geführt.

Vergleich mit ähnlichen Bausteinen

- **AIS_SPLIT_2 / AIS_SPLIT_8:** Diese Bausteine stellen Varianten mit zwei bzw. acht Ausgängen dar. Die Funktionsweise ist identisch; lediglich die Anzahl der ausgegebenen Adapter unterscheidet sich.
- **Datenbasierte Splitter (z. B. SPLIT_INT):** Im Unterschied zu adapterbasierten Splittern arbeiten datenbasierte Splitter mit konkreten Datentypen (z. B. Ganzzahlen) und benötigen meist Ereignisse zur Auslösung. Der AIS_SPLIT_4 ist reiner Adapter-split und benötigt keine explizite Triggerung.
- **Adapter-Multiplexer (z. B. AIS_MUX):** Ein Multiplexer wählt aus mehreren Eingängen einen aus, während der Splitter einen Eingang auf mehrere Ausgänge verteilt.
- `AIS_SPLIT_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

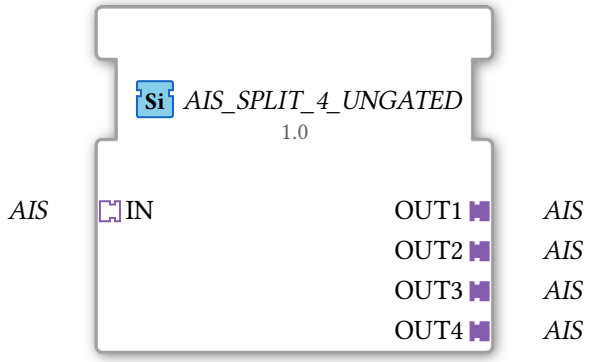
Der **AIS_SPLIT_4** ist ein einfacher, aber praktischer Funktionsblock zur zuverlässigen Verteilung eines AIS-Signals auf vier identische Ausgänge. Seine generische Auslegung und die reine Adapterlogik machen ihn flexibel einsetzbar und erfordern keine zusätzliche Ereignissteuerung. Er eignet sich ideal für Anwendungen, in denen ein Signal mehrfach benötigt wird, ohne Datenverarbeitung oder Synchronisation.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AIS_SPLIT_4_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `AIS_SPLIT_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AIS_SPLIT_4_UNGATED** dient dazu, ein eingehendes AIS-Adaptersignal (Typ `adapter::types::unidirectional::AIS`) auf vier separate Ausgänge zu verteilen. Er realisiert eine 1-zu-4-Aufteilung ohne Veränderung der Daten. Der Baustein ist generisch ausgeführt und kann über den generischen Klassennamen an unterschiedliche Signaltypen angepasst werden.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

- **Socket (Eingang):** `IN` – Typ `adapter::types::unidirectional::AIS`
- **Plugs (Ausgänge):**
  - `OUT1` – Typ `adapter::types::unidirectional::AIS`
  - `OUT2` – Typ `adapter::types::unidirectional::AIS`
  - `OUT3` – Typ `adapter::types::unidirectional::AIS`
  - `OUT4` – Typ `adapter::types::unidirectional::AIS`

Funktionsweise

Der Baustein leitet das am Socket `IN` anliegende Adaptersignal unverändert an alle vier Ausgangsadapter (`OUT1` … `OUT4`) weiter. Es findet keine Datenmanipulation, Filterung oder Pufferung statt. Jeder Ausgang entspricht exakt dem Eingangssignal. Die Verteilung erfolgt asynchron und ohne Ereignissteuerung – die Adapterschnittstelle übernimmt die Datenweitergabe entsprechend ihrer Kommunikationslogik.

Technische Besonderheiten

- **Generischer Typ:** Der Baustein ist als generischer Funktionsblock ausgeführt. Über das Attribut `eclipse4diac::core::GenericClassName` (Wert `'GEN_AIS_SPLIT'`) kann der eigentliche Signaltyp zur Erstellungszeit festgelegt werden.
- **Typ-Hash:** Ein Leerstring (`''`) für das Attribut `eclipse4diac::core::TypeHash` weist darauf hin, dass der Baustein bei Bedarf dynamisch konfiguriert wird.
- **Kompaktheit:** Da weder Ereignis- noch Dateneingänge/-ausgänge existieren, ist der Baustein rein adapterbasiert und frei von zeitlichen Abhängigkeiten.

Zustandsübersicht

Der Baustein besitzt keine Zustandsmaschine – er arbeitet kontinuierlich und zustandslos. Es gibt keine internalen Zustände oder Ereignisse, die geschaltet werden müssten. Die Funktionalität erschöpft sich in der passiven Durchleitung des Adaptersignals.

Anwendungsszenarien

- **Signalverteilung in Steuerungsarchitekturen:** Ein von einer Quelle (z. B. Sensor) bereitgestelltes AIS-Signal soll mehreren nachgeschalteten Verarbeitungseinheiten oder Anzeigen zur Verfügung gestellt werden.
- **Redundante Datenweitergabe:** In sicherheitskritischen Systemen kann das Eingangssignal an mehrere unabhängige Pfade geleitet werden, ohne es zu duplizieren.
- **Test- und Debugumgebungen:** Ein Signal wird parallel an die eigentliche Logik und an ein Analyse-/Protokolltool geführt.

Vergleich mit ähnlichen Bausteinen

- **AIS_SPLIT_2 / AIS_SPLIT_8:** Diese Bausteine stellen Varianten mit zwei bzw. acht Ausgängen dar. Die Funktionsweise ist identisch; lediglich die Anzahl der ausgegebenen Adapter unterscheidet sich.
- **Datenbasierte Splitter (z. B. SPLIT_INT):** Im Unterschied zu adapterbasierten Splittern arbeiten datenbasierte Splitter mit konkreten Datentypen (z. B. Ganzzahlen) und benötigen meist Ereignisse zur Auslösung. Der AIS_SPLIT_4_UNGATED ist reiner Adapter-split und benötigt keine explizite Triggerung.
- **Adapter-Multiplexer (z. B. AIS_MUX):** Ein Multiplexer wählt aus mehreren Eingängen einen aus, während der Splitter einen Eingang auf mehrere Ausgänge verteilt.
- `AIS_SPLIT_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

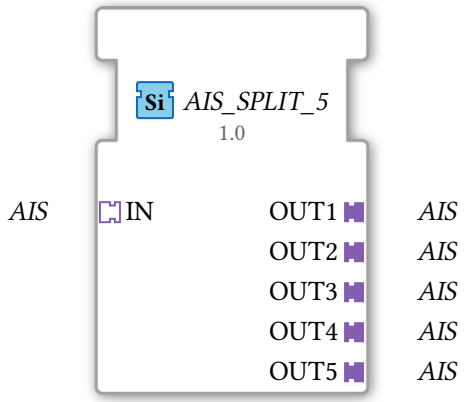
Fazit

Der **AIS_SPLIT_4_UNGATED** ist ein einfacher, aber praktischer Funktionsblock zur zuverlässigen Verteilung eines AIS-Signals auf vier identische Ausgänge. Seine generische Auslegung und die reine Adapterlogik machen ihn flexibel einsetzbar und erfordern keine zusätzliche Ereignissteuerung. Er eignet sich ideal für Anwendungen, in denen ein Signal mehrfach benötigt wird, ohne Datenverarbeitung oder Synchronisation.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AIS_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AIS_SPLIT_5** dient als generischer Verteiler (Splitter) für Adapter vom Typ `AIS` (unidirektional). Er nimmt ein eingehendes AIS-Signal über einen Socket entgegen und leitet es an fünf separate AIS-Plugs weiter. Dies ermöglicht die Vervielfachung eines Adaptersignals auf mehrere Zielbausteine.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	IN	`adapter::types::unidirectional::AIS`	Eingehendes AIS-Adapter-Signal
Plug (Ausgang 1)	OUT1	`adapter::types::unidirectional::AIS`	Erster ausgehender AIS-Adapter-Anschluss
Plug (Ausgang 2)	OUT2	`adapter::types::unidirectional::AIS`	Zweiter ausgehender AIS-Adapter-Anschluss
Plug (Ausgang 3)	OUT3	`adapter::types::unidirectional::AIS`	Dritter ausgehender AIS-Adapter-Anschluss
Plug (Ausgang 4)	OUT4	`adapter::types::unidirectional::AIS`	Vierter ausgehender AIS-Adapter-Anschluss
Plug (Ausgang 5)	OUT5	`adapter::types::unidirectional::AIS`	Fünfter ausgehender AIS-Adapter-Anschluss

Funktionsweise

Der Baustein arbeitet als reine Durchleitung (Broadcast) ohne eigene Logik oder Zustände. Sobald ein Ereignis oder eine Datenänderung über den Socket `IN` ankommt, wird dieses Signal unverändert an alle

fünf Plugs (`OUT1`–`OUT5`) weitergegeben. Jeder Plug kann mit einem anderen Empfängerbaustein verbunden werden, sodass das ursprüngliche Signal gleichzeitig von mehreren Komponenten verarbeitet werden kann.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Funktionsbaustein markiert (`eclipse4diac::core::GenericClassName = 'GEN_AIS_SPLIT'`). Dadurch kann er in verschiedenen Kontexten mit jeweils passenden AIS-Adapter-Typen instanziiert werden.
- **Keine Ereignis- oder Daten-Ein-/Ausgänge**: Die gesamte Kommunikation erfolgt ausschließlich über die Adapter-Schnittstellen. Dies macht ihn besonders einfach und effizient.
- **Unidirektional**: Der Adaptertyp `AIS` ist unidirektional, d. h. die Signalrichtung ist festgelegt (vom Socket zu den Plugs). Eine Rückkopplung ist nicht vorgesehen.

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandslogik (kein ECC). Es gibt keinen internen Automaten oder Verzögerung. Die Funktionsweise entspricht einer starren, kombinatorischen Weiterleitung.

Anwendungsszenarien

- **Signalverteilung in Steuerungssystemen**: Ein von einer Quelle (z. B. Sensor oder Kommunikationsbaustein) geliefertes AIS-Signal soll an mehrere unabhängige Verarbeitungsblöcke weitergegeben werden.
- **Test- und Simulationsumgebungen**: Ein Testsignal kann gleichzeitig an verschiedene Überwachungs- oder Analysebausteine gesendet werden.
- **Redundante Verarbeitung**: Das gleiche Signal wird mehrfach ausgewertet, um Fehler zu erkennen oder unterschiedliche Algorithmen zu vergleichen.

Vergleich mit ähnlichen Bausteinen

Es gibt weitere Splitter-Bausteine wie `AIS_SPLIT_2`, `AIS_SPLIT_3` oder `AIS_SPLIT_4`, die eine geringere Anzahl an Ausgängen bereitstellen. Gemeinsam ist allen, dass sie ein einzelnes AIS-Eingangssignal auf mehrere Ausgänge verteilen. Der `AIS_SPLIT_5` bietet die maximale Anzahl von fünf Ausgängen dieser Familie. Je nach Bedarf kann der passende Splitter gewählt werden, um unnötige Verkabelung zu vermeiden.

- `AIS_SPLIT_5_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der `AIS_SPLIT_5` ist ein einfacher, aber wirkungsvoller generischer Funktionsbaustein zur Vervielfachung eines unidirektionalen AIS-Adapter-Signals. Er reduziert den Verkabelungsaufwand in der 4diac-IDE und ermöglicht eine saubere, modulare Strukturierung von Steuerungsanwendungen. Für Anwendungen, die eine Aufteilung eines Signals auf bis zu fünf Empfänger erfordern, stellt er eine optimale Lösung dar.

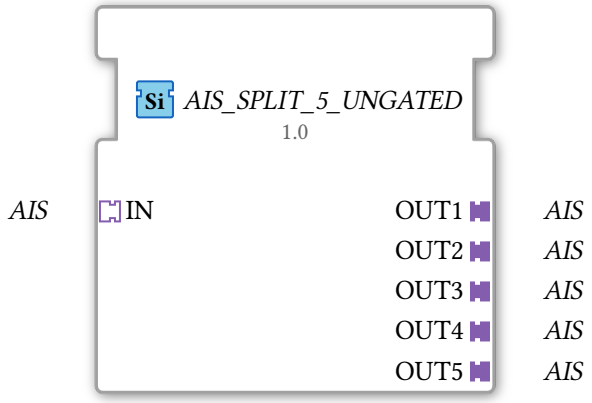
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AIS_SPLIT_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AIS_SPLIT_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht

gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AIS_SPLIT_5_UNGATED** dient als generischer Verteiler (Splitter) für Adapter vom Typ `AIS` (unidirektional). Er nimmt ein eingehendes AIS-Signal über einen Socket entgegen und leitet es an fünf separate AIS-Plugs weiter. Dies ermöglicht die Vervielfachung eines Adaptersignals auf mehrere Zielbausteine.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	IN	`adapter::types::unidirectional::AIS`	Eingehendes AIS-Adapter-Signal
Plug (Ausgang 1)	OUT1	`adapter::types::unidirectional::AIS`	Erster ausgehender AIS-Adapter-Anschluss
Plug (Ausgang 2)	OUT2	`adapter::types::unidirectional::AIS`	Zweiter ausgehender AIS-Adapter-Anschluss
Plug (Ausgang 3)	OUT3	`adapter::types::unidirectional::AIS`	Dritter ausgehender AIS-Adapter-Anschluss
Plug (Ausgang 4)	OUT4	`adapter::types::unidirectional::AIS`	Vierter ausgehender AIS-Adapter-Anschluss
Plug (Ausgang 5)	OUT5	`adapter::types::unidirectional::AIS`	Fünfter ausgehender AIS-Adapter-Anschluss

Funktionsweise

Der Baustein arbeitet als reine Durchleitung (Broadcast) ohne eigene Logik oder Zustände. Sobald ein Ereignis oder eine Datenänderung über den Socket `IN` ankommt, wird dieses Signal unverändert an alle fünf Plugs (`OUT1`–`OUT5`) weitergegeben. Jeder Plug kann mit einem anderen Empfängerbaustein verbunden werden, sodass das ursprüngliche Signal gleichzeitig von mehreren Komponenten verarbeitet werden kann.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Funktionsbaustein markiert (`eclipse4diac::core::GenericClassName = 'GEN_AIS_SPLIT'`). Dadurch kann er in verschiedenen Kontexten mit jeweils passenden AIS-Adapter-Typen instanziiert werden.
- **Keine Ereignis- oder Daten-Ein-/Ausgänge**: Die gesamte Kommunikation erfolgt ausschließlich über die Adapter-Schnittstellen. Dies macht ihn besonders einfach und effizient.
- **Unidirektional**: Der Adaptertyp `AIS` ist unidirektional, d. h. die Signalrichtung ist festgelegt (vom Socket zu den Plugs). Eine Rückkopplung ist nicht vorgesehen.

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandslogik (kein ECC). Es gibt keinen internen Automaten oder Verzögerung. Die Funktionsweise entspricht einer starren, kombinatorischen Weiterleitung.

Anwendungsszenarien

- **Signalverteilung in Steuerungssystemen**: Ein von einer Quelle (z. B. Sensor oder Kommunikationsbaustein) geliefertes AIS-Signal soll an mehrere unabhängige Verarbeitungsblöcke weitergegeben werden.
- **Test- und Simulationsumgebungen**: Ein Testsignal kann gleichzeitig an verschiedene Überwachungs- oder Analysebausteine gesendet werden.
- **Redundante Verarbeitung**: Das gleiche Signal wird mehrfach ausgewertet, um Fehler zu erkennen oder unterschiedliche Algorithmen zu vergleichen.

Vergleich mit ähnlichen Bausteinen

Es gibt weitere Splitter-Bausteine wie `AIS_SPLIT_2`, `AIS_SPLIT_3` oder `AIS_SPLIT_4`, die eine geringere Anzahl an Ausgängen bereitstellen. Gemeinsam ist allen, dass sie ein einzelnes AIS-Eingangssignal auf mehrere Ausgänge verteilen. Der `AIS_SPLIT_5_UNGATED` bietet die maximale Anzahl von fünf Ausgängen dieser Familie. Je nach Bedarf kann der passende Splitter gewählt werden, um unnötige Verkabelung zu vermeiden.

- **`AIS_SPLIT_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

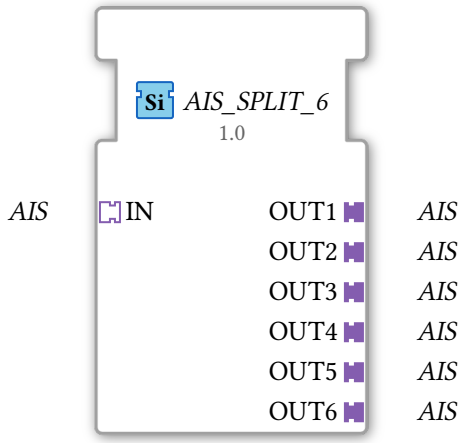
Fazit

Der `AIS_SPLIT_5_UNGATED` ist ein einfacher, aber wirkungsvoller generischer Funktionsbaustein zur Vervielfachung eines unidirektionalen AIS-Adapter-Signals. Er reduziert den Verkabelungsaufwand in der 4diac-IDE und ermöglicht eine saubere, modulare Strukturierung von Steuerungsanwendungen. Für Anwendungen, die eine Aufteilung eines Signals auf bis zu fünf Empfänger erfordern, stellt er eine optimale Lösung dar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AIS_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AIS_SPLIT_6** dient dazu, ein eingehendes AIS-Signal (Automation Interface Signal) auf sechs separate AIS-Ausgänge zu verteilen. Er fungiert als generischer Splitter, der das ankommende Signal ohne Verzögerung oder Manipulation an alle angeschlossenen Ausgänge weiterleitet.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereigniseingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignisausgänge vorhanden.

Daten-Eingänge

Keine Dateneingänge vorhanden (die Datenübertragung erfolgt ausschließlich über den Adapter).

Daten-Ausgänge

Keine Datenausgänge vorhanden (die Datenübertragung erfolgt ausschließlich über die Adapter).

Adapter

Typ	Richtung	Name	Beschreibung
`adapter::types::unidirectional::AIS`	Eingang (Socket)	IN	Eingehendes AIS-Signal, das auf alle Ausgänge verteilt wird.
`adapter::types::unidirectional::AIS`	Ausgang (Plug)	OUT1	Erster Ausgang mit dem gesplitteten AIS-Signal.
`adapter::types::unidirectional::AIS`	Ausgang (Plug)	OUT2	Zweiter Ausgang mit dem gesplitteten AIS-Signal.
`adapter::types::unidirectional::AIS`	Ausgang (Plug)	OUT3	Dritter Ausgang mit dem gesplitteten AIS-Signal.
`adapter::types::unidirectional::AIS`	Ausgang (Plug)	OUT4	Vierter Ausgang mit dem gesplitteten AIS-Signal.
`adapter::types::unidirectional::AIS`	Ausgang (Plug)	OUT5	Fünfter Ausgang mit dem gesplitteten AIS-Signal.
`adapter::types::unidirectional::AIS`	Ausgang (Plug)	OUT6	Sechster Ausgang mit dem gesplitteten AIS-Signal.

Funktionsweise

Der Baustein empfängt ein AIS-Signal über den Socket `IN` und leitet es gleichzeitig und identisch an alle sechs Plugs `OUT1` bis `OUT6` weiter. Es findet keine Datenverarbeitung, Filterung oder Zwischenspeicherung statt – die Verteilung erfolgt rein signaltechnisch. Der Baustein arbeitet unidirektional (nur vom Eingang zu den Ausgängen) und stellt somit eine einfache 1:6-Vervielfachung des AIS-Signals dar.

Technische Besonderheiten

- **Generischer Typ**: Der Baustein ist als generischer Funktionsblock (`GEN_AIS_SPLIT`) ausgeführt, kann also mit verschiedenen AIS-Typen verwendet werden, solange die Adapterdefinition `adapter::types::unidirectional::AIS` übereinstimmt.
- **Unidirektional**: Der Datenaustausch erfolgt nur in eine Richtung – vom Eingang zu den Ausgängen. Rückwirkungen von den Ausgängen auf den Eingang sind nicht vorgesehen.
- **Keine Eigenlogik**: Da weder Ereignisse noch Dateneingänge/-ausgänge existieren, läuft der Baustein rein passiv und benötigt keine Zustandsmaschine oder Algorithmen.
- **Einfache Skalierbarkeit**: Durch die feste Anzahl von sechs Ausgängen ist der Baustein für typische Anwendungen optimiert, bei denen genau sechs parallele AIS-Verbindungen benötigt werden.

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine (ECC). Sein Verhalten ist deterministisch und zeitlos: Ein ankommendes AIS-Signal wird sofort an alle Ausgänge weitergegeben. Es gibt keine internen Zustände oder Schritte.

Anwendungsszenarien

- **Verteilen von Steuersignalen**: Ein AIS-Signal von einer zentralen Steuerung (z. B. einem übergeordneten Funktionsbaustein) muss an mehrere untergeordnete Komponenten weitergeleitet werden.
- **Monitoring und Parallelbetrieb**: Ein Signal soll gleichzeitig für Regelung, Überwachung und Protokollierung genutzt werden.
- **Testaufbauten**: Einheitliches Testsignal wird auf mehrere simulierte oder reale Module verteilt.
- **Redundante Verkabelung**: Signalverteilung über mehrere Pfade zur Erhöhung der Ausfallsicherheit.

Vergleich mit ähnlichen Bausteinen

Im 4diac-Framework existieren möglicherweise andere Split-Bausteine, die sich in der Anzahl der Ausgänge oder im Datentyp unterscheiden. `AIS_SPLIT_6` ist speziell auf die AIS-Adapter-Schnittstelle und sechs Ausgänge zugeschnitten. Andere Splitter könnten eventuell über zusätzliche Ereignisse oder Konfigurationsparameter verfügen (z. B. selektives Weiterleiten), während dieser Baustein bewusst einfach und ohne Konfiguration gehalten ist.

- **`AIS_SPLIT_6_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

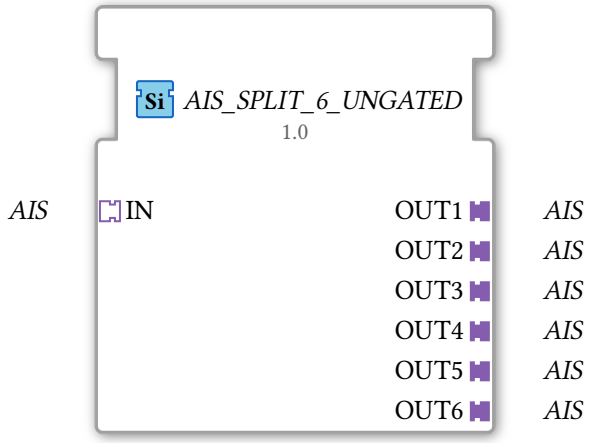
Der `AIS_SPLIT_6` ist ein nützlicher und minimalistischer Baustein zur Signalfächerung in AIS-basierten Steuerungssystemen. Er erfüllt zuverlässig die Aufgabe, ein eingehendes Signal auf sechs Ausgänge zu verteilen, und zeichnet sich durch seine Einfachheit, generische Auslegung und klare Trennung der Schnittstellen aus. Dadurch eignet er sich besonders für modulare, transparente Steuerungsarchitekturen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AIS_SPLIT_6_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `AIS_SPLIT_6`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AIS_SPLIT_6_UNGATED** dient dazu, ein eingehendes AIS-Signal (Automation Interface Signal) auf sechs separate AIS-Ausgänge zu verteilen. Er fungiert als generischer Splitter, der das ankommende Signal ohne Verzögerung oder Manipulation an alle angeschlossenen Ausgänge weiterleitet.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereigniseingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignisausgänge vorhanden.

Daten-Eingänge

Keine Dateneingänge vorhanden (die Datenübertragung erfolgt ausschließlich über den Adapter).

Daten-Ausgänge

Keine Datenausgänge vorhanden (die Datenübertragung erfolgt ausschließlich über die Adapter).

Adapter

Typ	Richtung	Name	Beschreibung
`adapter::types::unidirectional::AIS`	Eingang (Socket)	IN	Eingehendes AIS-Signal, das auf alle Ausgänge verteilt wird.
`adapter::types::unidirectional::AIS`	Ausgang (Plug)	OUT1	Erster Ausgang mit dem gesplitteten AIS-Signal.
`adapter::types::unidirectional::AIS`	Ausgang (Plug)	OUT2	Zweiter Ausgang mit dem gesplitteten AIS-Signal.

adapter::types::unidirectional::AIS	Ausgang (Plug)	OUT3	Dritter Ausgang mit dem gesplitteten AIS-Signal.
adapter::types::unidirectional::AIS	Ausgang (Plug)	OUT4	Vierter Ausgang mit dem gesplitteten AIS-Signal.
adapter::types::unidirectional::AIS	Ausgang (Plug)	OUT5	Fünfter Ausgang mit dem gesplitteten AIS-Signal.
adapter::types::unidirectional::AIS	Ausgang (Plug)	OUT6	Sechster Ausgang mit dem gesplitteten AIS-Signal.

Funktionsweise

Der Baustein empfängt ein AIS-Signal über den Socket `IN` und leitet es gleichzeitig und identisch an alle sechs Plugs `OUT1` bis `OUT6` weiter. Es findet keine Datenverarbeitung, Filterung oder Zwischenspeicherung statt – die Verteilung erfolgt rein signaltechnisch. Der Baustein arbeitet unidirektional (nur vom Eingang zu den Ausgängen) und stellt somit eine einfache 1:6-Vervielfachung des AIS-Signals dar.

Technische Besonderheiten

- **Generischer Typ**: Der Baustein ist als generischer Funktionsblock (`GEN_AIS_SPLIT`) ausgeführt, kann also mit verschiedenen AIS-Typen verwendet werden, solange die Adapterdefinition `adapter::types::unidirectional::AIS` übereinstimmt.
- **Unidirektional**: Der Datenaustausch erfolgt nur in eine Richtung – vom Eingang zu den Ausgängen. Rückwirkungen von den Ausgängen auf den Eingang sind nicht vorgesehen.
- **Keine Eigenlogik**: Da weder Ereignisse noch Dateneingänge/-ausgänge existieren, läuft der Baustein rein passiv und benötigt keine Zustandsmaschine oder Algorithmen.
- **Einfache Skalierbarkeit**: Durch die feste Anzahl von sechs Ausgängen ist der Baustein für typische Anwendungen optimiert, bei denen genau sechs parallele AIS-Verbindungen benötigt werden.

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine (ECC). Sein Verhalten ist deterministisch und zeitlos: Ein ankommendes AIS-Signal wird sofort an alle Ausgänge weitergegeben. Es gibt keine internen Zustände oder Schritte.

Anwendungsszenarien

- **Verteilen von Steuersignalen**: Ein AIS-Signal von einer zentralen Steuerung (z. B. einem übergeordneten Funktionsbaustein) muss an mehrere untergeordnete Komponenten weitergeleitet werden.
- **Monitoring und Parallelbetrieb**: Ein Signal soll gleichzeitig für Regelung, Überwachung und Protokollierung genutzt werden.
- **Testaufbauten**: Einheitliches Testsignal wird auf mehrere simulierte oder reale Module verteilt.
- **Redundante Verkabelung**: Signalverteilung über mehrere Pfade zur Erhöhung der Ausfallsicherheit.

Vergleich mit ähnlichen Bausteinen

Im 4diac-Framework existieren möglicherweise andere Split-Bausteine, die sich in der Anzahl der Ausgänge oder im Datentyp unterscheiden. `AIS_SPLIT_6_UNGATED` ist speziell auf die AIS-Adapter-Schnittstelle und sechs Ausgänge zugeschnitten. Andere Splitter könnten eventuell über zusätzliche Ereignisse oder Konfigurationsparameter verfügen (z. B. selektives Weiterleiten), während dieser Baustein bewusst einfach und ohne Konfiguration gehalten ist.

- `AIS_SPLIT_6`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

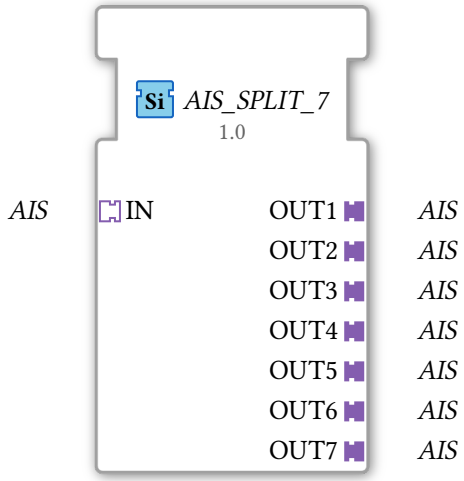
Der `AIS_SPLIT_6_UNGATED` ist ein nützlicher und minimalistischer Baustein zur Signalfächerung in AIS-basierten Steuerungssystemen. Er erfüllt zuverlässig die Aufgabe, ein eingehendes Signal auf sechs Ausgänge zu verteilen, und zeichnet sich durch seine Einfachheit, generische Auslegung und klare

Trennung der Schnittstellen aus. Dadurch eignet er sich besonders für modulare, transparente Steuerungsarchitekturen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AIS_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AIS_SPLIT_7** dient zur Verteilung eines eingehenden AIS-Datenstroms auf sieben identische Ausgänge. Er ist als generischer Baustein implementiert und ermöglicht die Aufteilung eines unidirektionalen AIS-Adapters (Socket IN) auf sieben gleichartige Adapter-Plugs (OUT1 … OUT7). Die Bezeichnung „AIS“ steht hier für einen anwendungsspezifischen, abstrakten Datentyp im Adapter-Kontext der IEC 61499.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	IN	`adapter::types::unidirectional::AIS`	Eingehender AIS-Datenstrom, der auf alle Ausgänge verteilt wird.
Plug (Ausgang)	OUT1 … OUT7	`adapter::types::unidirectional::AIS`	Sieben identische Ausgänge, die den eingehenden AIS-Datenstrom jeweils unverändert weiterleiten.

Funktionsweise

Der Baustein fungiert als reiner Verteiler (Splitter) für AIS-Adapterverbindungen. Jedes ankommende Datum oder Ereignis am Socket `IN` wird **ohne Verzögerung oder Verarbeitung** auf alle sieben Adapter-Plugs `OUT1` bis `OUT7` repliziert. Die Weiterleitung erfolgt parallel, d. h. alle Ausgänge erhalten gleichzeitig die gleichen Informationen. Eine interne Logik oder Zustandsmaschine ist nicht erforderlich, da es sich um eine passive, verbindungsorientierte Datenweitergabe handelt.

Technische Besonderheiten

- **Generischer Baustein** – Der FB ist als generische Instanz (`GEN_AIS_SPLIT`) deklariert und kann für beliebige AIS-Typen parametrisiert werden.
- **Unidirektionale Adapter** – Sowohl Eingang als auch Ausgänge verwenden ausschließlich den unidirektionalen Adaptertyp, sodass eine reine Flussrichtung vom Eingang zu den Ausgängen sichergestellt ist.
- **Keine Ereignis- oder Daten-Ein-/Ausgänge** – Die gesamte Kommunikation erfolgt ausschließlich über Adapter, was eine saubere Kapselung und Wiederverwendung ermöglicht.
- **Skalierbarkeit** – Das Prinzip lässt sich leicht auf andere Anzahlen von Ausgängen (z. B. AIS_SPLIT_2, AIS_SPLIT_8) übertragen.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC), da er keinerlei sequentielle Logik oder verzögerte Operationen ausführt. Die Funktion reduziert sich auf eine statische, durchgehende Weiterleitung. Ein Betriebszustand ist daher nicht definiert.

Anwendungsszenarien

- **Verteilung von AIS-Signalen** an mehrere parallele Verarbeitungseinheiten, z. B. in Automatisierungsanlagen, wenn ein Sensorwert gleichzeitig an mehrere Regelkreise, Anzeigen oder Analysewerkzeuge übergeben werden muss.
- **Redundante Signalweiterleitung** – Einsatz zur Einspeisung eines AIS-Datenstroms in mehrere unabhängige Pfade zu Test- oder Überwachungszwecken.
- **Modulare Systemarchitektur** – Als Komponente in einer Bibliothek generischer Adapter-Bausteine, um die Wiederverwendung und Austauschbarkeit von Verbindungen zu erhöhen.

Vergleich mit ähnlichen Bausteinen

- **Daten-Splitter (z. B. SPLIT_1_to_N)** – Diese teilen meist einfache Datentypen (BOOL, INT, REAL) auf. Der AIS_SPLIT_7 ist speziell für Adapter-Schnittstellen ausgelegt und nutzt das Adapterkonzept, das in IEC 61499 für komplexe, strukturierte Datenströme vorgesehen ist.
- **Adapter-Multiplexer (z. B. AIS_MERGE)** – Während ein Splitter einen Eingang auf mehrere Ausgänge verteilt, fasst ein Merger mehrere Eingänge zu einem Ausgang zusammen. Der AIS_SPLIT_7 ist das Gegenstück dazu.
- **Event-Splitter** – Unterscheiden sich durch die Verarbeitung von Ereignissen; der AIS_SPLIT_7 arbeitet hingegen rein adapterbasiert ohne separate Ereigniskanäle.
- `AIS_SPLIT_7_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

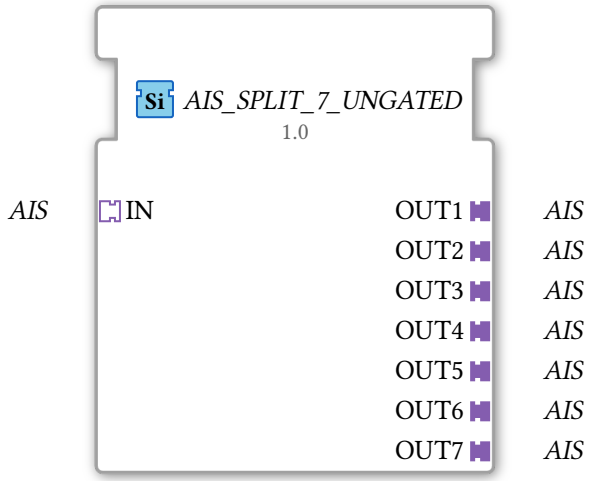
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der **AIS_SPLIT_7** ist ein einfacher, aber effektiver Funktionsblock zur Verteilung eines unidirektionalen AIS-Datenstroms auf sieben parallele Ausgänge. Seine generische Ausführung und die ausschließliche Nutzung von Adaptern machen ihn zu einem flexiblen Werkzeug in der automatisierungstechnischen Anwendungsentwicklung nach IEC 61499. Durch das Fehlen von Zustandslogik und Ereignissteuerung ist er leichtgewichtig und für reine Verteilungsaufgaben optimal geeignet.

AIS_SPLIT_7_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `AIS_SPLIT_7`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AIS_SPLIT_7_UNGATED** dient zur Verteilung eines eingehenden AIS-Datenstroms auf sieben identische Ausgänge. Er ist als generischer Baustein implementiert und ermöglicht die Aufteilung eines unidirektionalen AIS-Adapters (Socket IN) auf sieben gleichartige Adapter-Plugs (OUT1 … OUT7). Die Bezeichnung „AIS“ steht hier für einen anwendungsspezifischen, abstrakten Datentyp im Adapter-Kontext der IEC 61499.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	IN	`adapter::types::unidirectional::AIS`	Eingehender AIS-Datenstrom, der auf alle Ausgänge verteilt wird.
Plug (Ausgang)	OUT1 … OUT7	`adapter::types::unidirectional::AIS`	Sieben identische Ausgänge, die den eingehenden AIS-Da-

			tenstrom jeweils unverändert weiterleiten.

Funktionsweise

Der Baustein fungiert als reiner Verteiler (Splitter) für AIS-Adapterverbindungen. Jedes ankommende Datum oder Ereignis am Socket `IN` wird **ohne Verzögerung oder Verarbeitung** auf alle sieben Adapter-Plugs `OUT1` bis `OUT7` repliziert. Die Weiterleitung erfolgt parallel, d. h. alle Ausgänge erhalten gleichzeitig die gleichen Informationen. Eine interne Logik oder Zustandsmaschine ist nicht erforderlich, da es sich um eine passive, verbindungsorientierte Datenweitergabe handelt.

Technische Besonderheiten

- **Generischer Baustein** – Der FB ist als generische Instanz (`GEN_AIS_SPLIT`) deklariert und kann für beliebige AIS-Typen parametrisiert werden.
- **Unidirektionale Adapter** – Sowohl Eingang als auch Ausgänge verwenden ausschließlich den unidirektionalen Adaptertyp, sodass eine reine Flussrichtung vom Eingang zu den Ausgängen sichergestellt ist.
- **Keine Ereignis- oder Daten-Ein-/Ausgänge** – Die gesamte Kommunikation erfolgt ausschließlich über Adapter, was eine saubere Kapselung und Wiederverwendung ermöglicht.
- **Skalierbarkeit** – Das Prinzip lässt sich leicht auf andere Anzahlen von Ausgängen (z. B. AIS_SPLIT_2, AIS_SPLIT_8) übertragen.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC), da er keinerlei sequentielle Logik oder verzögerte Operationen ausführt. Die Funktion reduziert sich auf eine statische, durchgehende Weiterleitung. Ein Betriebszustand ist daher nicht definiert.

Anwendungsszenarien

- **Verteilung von AIS-Signalen** an mehrere parallele Verarbeitungseinheiten, z. B. in Automatisierungsanlagen, wenn ein Sensorwert gleichzeitig an mehrere Regelkreise, Anzeigen oder Analysewerkzeuge übergeben werden muss.
- **Redundante Signalweiterleitung** – Einsatz zur Einspeisung eines AIS-Datenstroms in mehrere unabhängige Pfade zu Test- oder Überwachungszwecken.
- **Modulare Systemarchitektur** – Als Komponente in einer Bibliothek generischer Adapter-Bausteine, um die Wiederverwendung und Austauschbarkeit von Verbindungen zu erhöhen.

Vergleich mit ähnlichen Bausteinen

- **Daten-Splitter (z. B. SPLIT_1_to_N)** – Diese teilen meist einfache Datentypen (BOOL, INT, REAL) auf. Der AIS_SPLIT_7_UNGATED ist speziell für Adapter-Schnittstellen ausgelegt und nutzt das Adapterkonzept, das in IEC 61499 für komplexe, strukturierte Datenströme vorgesehen ist.
- **Adapter-Multiplexer (z. B. AIS_MERGE)** – Während ein Splitter einen Eingang auf mehrere Ausgänge verteilt, fasst ein Merger mehrere Eingänge zu einem Ausgang zusammen. Der AIS_SPLIT_7_UNGATED ist das Gegenstück dazu.
- **Event-Splitter** – Unterscheiden sich durch die Verarbeitung von Ereignissen; der AIS_SPLIT_7_UNGATED arbeitet hingegen rein adapterbasiert ohne separate Ereigniskanäle.
- `AIS_SPLIT_7`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

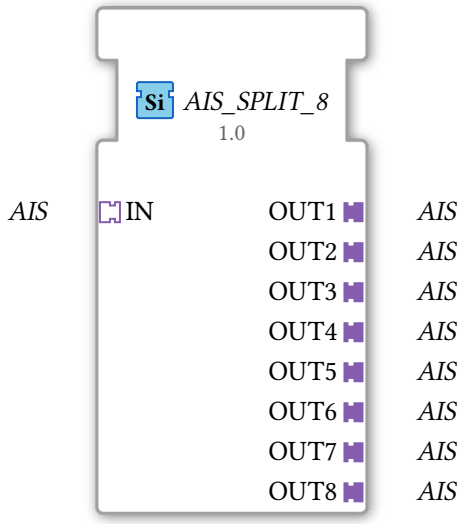
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AIS_SPLIT_7_UNGATED** ist ein einfacher, aber effektiver Funktionsblock zur Verteilung eines unidirektionalen AIS-Datenstroms auf sieben parallele Ausgänge. Seine generische Ausführung und die ausschließliche Nutzung von Adaptern machen ihn zu einem flexiblen Werkzeug in der automatisierungstechnischen Anwendungsentwicklung nach IEC 61499. Durch das Fehlen von Zustandslogik und Ereignissteuerung ist er leichtgewichtig und für reine Verteilungsaufgaben optimal geeignet.

AIS_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AIS_SPLIT_8** dient der Verteilung eines eingehenden AIS (Application Interface Socket) auf acht identische Ausgangsadapter. Er ist als generischer Baustein konzipiert und erlaubt die gleichzeitige Weitergabe eines AIS-Signals an mehrere nachfolgende Funktionsblöcke ohne Datenverlust oder Verzögerung.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Eingang	IN	`adapter::types::unidirectional::AIS`	Eingehendes AIS-Signal, das verteilt werden soll.
Ausgang	OUT1	`adapter::types::unidirectional::AIS`	Erster Ausgangskanal (identisch zu IN).
Ausgang	OUT2	`adapter::types::unidirectional::AIS`	Zweiter Ausgangskanal (identisch zu IN).
Ausgang	OUT3	`adapter::types::unidirectional::AIS`	Dritter Ausgangskanal (identisch zu IN).
Ausgang	OUT4	`adapter::types::unidirectional::AIS`	Vierter Ausgangskanal (identisch zu IN).

Ausgang	OUT5	`adapter::types::unidirectional::AIS`	Fünfter Ausgangskanal (identisch zu IN).
Ausgang	OUT6	`adapter::types::unidirectional::AIS`	Sechster Ausgangskanal (identisch zu IN).
Ausgang	OUT7	`adapter::types::unidirectional::AIS`	Siebter Ausgangskanal (identisch zu IN).
Ausgang	OUT8	`adapter::types::unidirectional::AIS`	Achter Ausgangskanal (identisch zu IN).

Funktionsweise

Der Baustein leitet das am Adapter-Eingang `IN` anliegende AIS-Signal unverändert und ohne Zeitverzögerung an alle acht Ausgangsadapter `OUT1` bis `OUT8` weiter. Es findet keine Transformation, Filterung oder Pufferung statt. Das Verhalten entspricht einer passiven Verteilung (Broadcast) eines AIS-Interfaces.

Technische Besonderheiten

- **Generischer Typ** – Der Baustein ist als `GEN_AIS_SPLIT` deklariert und kann über Typparametrisierung an spezifische AIS-Ausprägungen angepasst werden (z. B. für verschiedene Datentypen oder Ereignisstrukturen).
- **Keine Laufzeitlogik** – Die gesamte Funktionalität wird allein durch die Verschaltung der Adapter realisiert; es existieren keine internen Algorithmen oder Zustandsautomaten.
- **Minimaler Ressourcenbedarf** – Da keine aktive Verarbeitung stattfindet, verursacht der Baustein weder CPU-Last noch Speicherverbrauch bei der Ausführung.

Zustandsübersicht

Der Baustein besitzt keine internen Zustände. Das Ausgangssignal ist stets eine direkte Abbildung des Eingangssignals.

Anwendungsszenarien

- **Parallelverarbeitung** – Aufteilung eines Sensorsignals auf mehrere unabhängige Auswertungsblöcke.
- **Redundanz** – Verteilung eines Steuersignals an mehrere Aktoren in sicherheitskritischen Systemen.
- **Test- und Simulationsumgebungen** – Bereitstellung eines identischen Signals für verschiedene Beobachter oder Logger.

Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu Bausteinen wie `AIS_MERGE_2` oder `AIS_SELECT` besitzt `AIS_SPLIT_8` keine Selektions- oder Priorisierungslogik. Verwandte Varianten unterscheiden sich lediglich in der Anzahl der Ausgänge (z. B. `AIS_SPLIT_2` oder `AIS_SPLIT_4`). `AIS_SPLIT_8` ist die maximale Standardausführung für acht Kanäle.

- `AIS_SPLIT_8_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

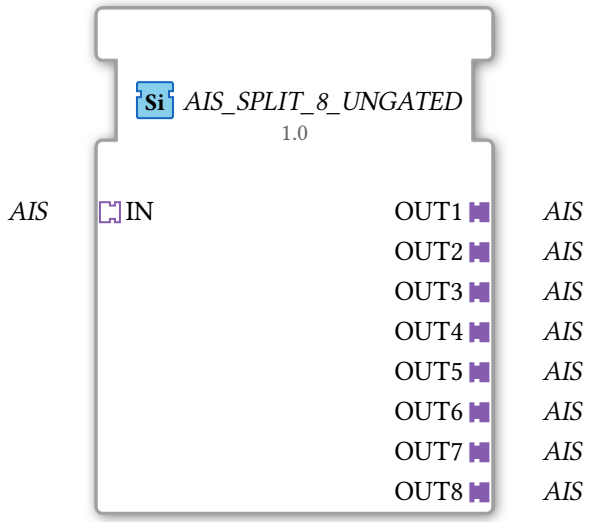
Fazit

`AIS_SPLIT_8` ist ein einfacher und effizienter Baustein zur Vervielfältigung eines AIS-Interfaces. Durch seine generische Natur und die reine Adapter-basierte Umsetzung eignet er sich ideal für den Aufbau modularer Automatisierungsarchitekturen, die eine verteilte Signalweiterleitung benötigen.

AIS_SPLIT_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AIS_SPLIT_8`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weiter-

gegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AIS_SPLIT_8_UNGATED** dient der Verteilung eines eingehenden AIS (Application Interface Socket) auf acht identische Ausgangsadapter. Er ist als generischer Baustein konzipiert und erlaubt die gleichzeitige Weitergabe eines AIS-Signals an mehrere nachfolgende Funktionsblöcke ohne Datenverlust oder Verzögerung.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Eingang	IN	`adapter::types::unidirectional::AIS`	Eingehendes AIS-Signal, das verteilt werden soll.
Ausgang	OUT1	`adapter::types::unidirectional::AIS`	Erster Ausgangskanal (identisch zu IN).
Ausgang	OUT2	`adapter::types::unidirectional::AIS`	Zweiter Ausgangskanal (identisch zu IN).
Ausgang	OUT3	`adapter::types::unidirectional::AIS`	Dritter Ausgangskanal (identisch zu IN).

Ausgang	OUT4	`adapter::types::unidirectional::AIS`	Vierter Ausgangskanal (identisch zu IN).
Ausgang	OUT5	`adapter::types::unidirectional::AIS`	Fünfter Ausgangskanal (identisch zu IN).
Ausgang	OUT6	`adapter::types::unidirectional::AIS`	Sechster Ausgangskanal (identisch zu IN).
Ausgang	OUT7	`adapter::types::unidirectional::AIS`	Siebter Ausgangskanal (identisch zu IN).
Ausgang	OUT8	`adapter::types::unidirectional::AIS`	Achter Ausgangskanal (identisch zu IN).

Funktionsweise

Der Baustein leitet das am Adapter-Eingang `IN` anliegende AIS-Signal unverändert und ohne Zeitverzögerung an alle acht Ausgangsadapter `OUT1` bis `OUT8` weiter. Es findet keine Transformation, Filterung oder Pufferung statt. Das Verhalten entspricht einer passiven Verteilung (Broadcast) eines AIS-Interfaces.

Technische Besonderheiten

- **Generischer Typ** – Der Baustein ist als `GEN_AIS_SPLIT` deklariert und kann über Typparametrisierung an spezifische AIS-Ausprägungen angepasst werden (z. B. für verschiedene Datentypen oder Ereignisstrukturen).
- **Keine Laufzeitlogik** – Die gesamte Funktionalität wird allein durch die Verschaltung der Adapter realisiert; es existieren keine internen Algorithmen oder Zustandsautomaten.
- **Minimaler Ressourcenbedarf** – Da keine aktive Verarbeitung stattfindet, verursacht der Baustein weder CPU-Last noch Speicherverbrauch bei der Ausführung.

Zustandsübersicht

Der Baustein besitzt keine internen Zustände. Das Ausgangssignal ist stets eine direkte Abbildung des Eingangssignals.

Anwendungsszenarien

- **Parallelverarbeitung** – Aufteilung eines Sensorsignals auf mehrere unabhängige Auswertungsblöcke.
- **Redundanz** – Verteilung eines Steuersignals an mehrere Aktoren in sicherheitskritischen Systemen.
- **Test- und Simulationsumgebungen** – Bereitstellung eines identischen Signals für verschiedene Beobachter oder Logger.

Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu Bausteinen wie `AIS_MERGE_2` oder `AIS_SELECT` besitzt `AIS_SPLIT_8_UNGATED` keine Selektions- oder Priorisierungslogik. Verwandte Varianten unterscheiden sich lediglich in der Anzahl der Ausgänge (z. B. `AIS_SPLIT_2` oder `AIS_SPLIT_4`). `AIS_SPLIT_8_UNGATED` ist die maximale Standardausführung für acht Kanäle.

- `AIS_SPLIT_8`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

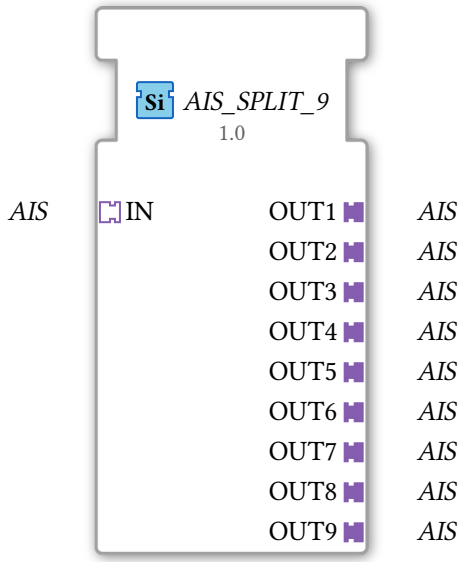
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

`AIS_SPLIT_8_UNGATED` ist ein einfacher und effizienter Baustein zur Vervielfältigung eines AIS-Interfaces. Durch seine generische Natur und die reine Adapter-basierte Umsetzung eignet er sich ideal für den Aufbau modularer Automatisierungsarchitekturen, die eine verteilte Signalweiterleitung benötigen.

AIS_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AIS_SPLIT_9** dient dazu, einen eingehenden unidirektionalen **AIS**-Adapter (Application Interface Socket) auf neun identische ausgehende **AIS**-Adapter (Plugs) zu verteilen. Er ist als generischer Baustein konzipiert und ermöglicht eine einfache Vervielfachung eines AIS-Signals für bis zu neun nachgeschaltete Funktionsblöcke.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Bezeichnung	Typ	Richtung	Beschreibung
IN	AIS (unidirectional)	Socket	Eingangsadapter – das zu verteilende AIS-Signal.
OUT1	AIS (unidirectional)	Plug	Erster Ausgang – identische Kopie des Eingangssignals.
OUT2	AIS (unidirectional)	Plug	Zweiter Ausgang – identische Kopie des Eingangssignals.
OUT3	AIS (unidirectional)	Plug	Dritter Ausgang – identische Kopie des Eingangssignals.
OUT4	AIS (unidirectional)	Plug	Vierter Ausgang – identische Kopie des Eingangssignals.

OUT5	AIS (unidirectional)	Plug	Fünfter Ausgang – identische Kopie des Eingangssignals.
OUT6	AIS (unidirectional)	Plug	Sechster Ausgang – identische Kopie des Eingangssignals.
OUT7	AIS (unidirectional)	Plug	Siebter Ausgang – identische Kopie des Eingangssignals.
OUT8	AIS (unidirectional)	Plug	Achter Ausgang – identische Kopie des Eingangssignals.
OUT9	AIS (unidirectional)	Plug	Neunter Ausgang – identische Kopie des Eingangssignals.

Funktionsweise

Der **AIS_SPLIT_9** ist ein rein kombinatorischer Baustein ohne eigene Zustandslogik oder Ereignissteuerung. Er empfängt am **Socket IN** ein AIS-Signal und leitet dieses unverändert an alle neun **Plugs OUT1** bis **OUT9** weiter. Jede Änderung am Eingang wird sofort auf alle Ausgänge übertragen. Der Baustein fungiert somit als passiver Signalverteiler („Broadcast“) auf der AIS-Schnittstelle.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ definiert (`GenericClassName = GEN_AIS_SPLIT`). Dies ermöglicht eine flexible Wiederverwendung in unterschiedlichen AIS-Kontexten, ohne dass der Baustein für jede Anwendung erneut erstellt werden muss.
- **Unidirektionale Übertragung**: Sowohl Eingang als auch Ausgänge verwenden den unidirektionalen **AIS**-Adaptertyp, d. h. die Daten fließen nur in eine Richtung (vom Eingang zu den Ausgängen).
- **Keine Datenpufferung oder Verzögerung**: Aufgrund des fehlenden Ereignis- und Daten-Handlings arbeitet der Baustein ohne Zwischenspeicher oder Taktung – er ist strikt signaldurchschaltend.
- **Einfaches Interface**: Der Baustein besitzt weder Ereignisse noch Daten-Ein-/Ausgänge, sondern ausschließlich Adapter. Dies macht ihn besonders leichtgewichtig und für reine Signalkopplungen geeignet.

Zustandsübersicht

Der **AIS_SPLIT_9** besitzt keine internen Zustände oder Ablaufsteuerungen. Sein Verhalten ist deterministisch und jederzeit durch das Eingangssignal bestimmt. Es existieren keine initialisierenden oder fehlerhaften Zustände.

Anwendungsszenarien

- **Signalvervielfachung**: Ein AIS-basiertes Steuerungssignal soll parallel an mehrere unabhängige Aktoren, Regler oder Anzeigen weitergegeben werden.
- **Broadcast in Modularen Systemen**: In einer verteilten Automatisierungsumgebung wird ein zentraler Messwert (z. B. Temperatur, Druck) über AIS bereitgestellt und muss gleichzeitig von mehreren Funktionsblöcken konsumiert werden.
- **Prototyping und Test**: Ein Testsignal soll auf mehrere Instanzen eines gleichen FB-Typs verteilt werden, ohne dass der Quell-FB mehrfach instanziiert werden muss.

Vergleich mit ähnlichen Bausteinen

- **AIS_SPLIT_2 / AIS_SPLIT_4 / AIS_SPLIT_8** etc.: Diese Bausteine verteilen das Eingangssignal auf 2, 4 oder 8 Ausgänge. Der **AIS_SPLIT_9** ist speziell für Anwendungen ausgelegt, die genau neun parallele Verbindungen benötigen. Die Funktionsweise ist identisch – der einzige Unterschied liegt in der Anzahl der Ausgänge.
- **AIS_MERGE**: Im Gegensatz zum Split-Baustein fasst ein Merge-Baustein mehrere Eingänge zu einem Ausgang zusammen. **AIS_SPLIT_9** realisiert den umgekehrten Datenfluss.
- **Direkte Kopplung**: Anstelle eines Split-Bausteins könnte man auch eine manuelle Verdrahtung (mehrere OUT-Adapter) in der 4diac-IDE vornehmen. Der **AIS_SPLIT_9** vereinfacht jedoch die grafische Darstellung und reduziert die Komplexität des Netzwerks.

- `AIS_SPLIT_9_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

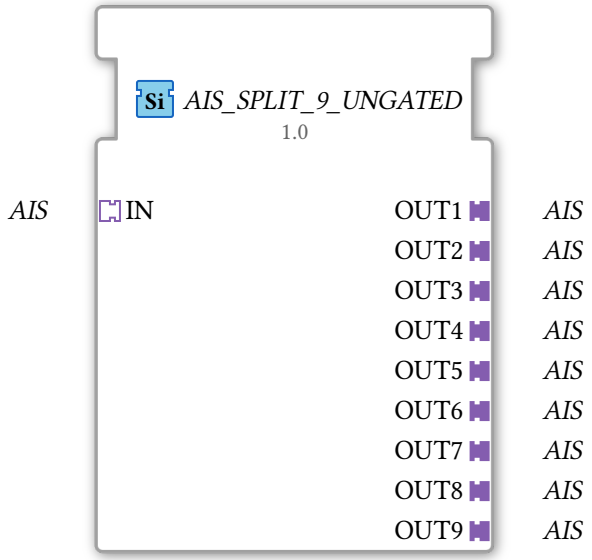
Der **AIS_SPLIT_9** ist ein einfacher, aber äußerst nützlicher Funktionsblock zur Vervielfältigung eines unidirektionalen AIS-Signals auf neun Ausgänge. Seine generische Auslegung, der Verzicht auf Ereignis- oder Datenlogik und die klare Schnittstelle machen ihn zu einem verlässlichen Werkzeug für modulare Automatisierungslösungen. Er eignet sich besonders für Anwendungen, bei denen ein Signal ohne Verzögerung oder Änderung an mehrere Empfänger verteilt werden muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AIS_SPLIT_9_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AIS_SPLIT_9`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AIS_SPLIT_9_UNGATED** dient dazu, einen eingehenden unidirektionalen **AIS**-Adapter (Application Interface Socket) auf neun identische ausgehende **AIS**-Adapter (Plugs) zu verteilen. Er ist als generischer Baustein konzipiert und ermöglicht eine einfache Vervielfachung eines AIS-Signals für bis zu neun nachgeschaltete Funktionsblöcke.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge
Keine Daten-Eingänge vorhanden.

Daten-Ausgänge
Keine Daten-Ausgänge vorhanden.

Adapter

Bezeichnung	Typ	Richtung	Beschreibung
IN	AIS (unidirectional)	Socket	Eingangsadapter – das zu verteilende AIS-Signal.
OUT1	AIS (unidirectional)	Plug	Erster Ausgang – identische Kopie des Eingangssignals.
OUT2	AIS (unidirectional)	Plug	Zweiter Ausgang – identische Kopie des Eingangssignals.
OUT3	AIS (unidirectional)	Plug	Dritter Ausgang – identische Kopie des Eingangssignals.
OUT4	AIS (unidirectional)	Plug	Vierter Ausgang – identische Kopie des Eingangssignals.
OUT5	AIS (unidirectional)	Plug	Fünfter Ausgang – identische Kopie des Eingangssignals.
OUT6	AIS (unidirectional)	Plug	Sechster Ausgang – identische Kopie des Eingangssignals.
OUT7	AIS (unidirectional)	Plug	Siebter Ausgang – identische Kopie des Eingangssignals.
OUT8	AIS (unidirectional)	Plug	Achter Ausgang – identische Kopie des Eingangssignals.
OUT9	AIS (unidirectional)	Plug	Neunter Ausgang – identische Kopie des Eingangssignals.

Funktionsweise

Der **AIS_SPLIT_9_UNGATED** ist ein rein kombinatorischer Baustein ohne eigene Zustandslogik oder Ereignissteuerung. Er empfängt am **Socket IN** ein AIS-Signal und leitet dieses unverändert an alle neun **Plugs OUT1** bis **OUT9** weiter. Jede Änderung am Eingang wird sofort auf alle Ausgänge übertragen. Der Baustein fungiert somit als passiver Signalverteiler („Broadcast“) auf der AIS-Schnittstelle.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ definiert (`GenericClassName = GEN_AIS_SPLIT`). Dies ermöglicht eine flexible Wiederverwendung in unterschiedlichen AIS-Kontexten, ohne dass der Baustein für jede Anwendung erneut erstellt werden muss.
- **Unidirektionale Übertragung**: Sowohl Eingang als auch Ausgänge verwenden den unidirektionalen **AIS**-Adaptertyp, d. h. die Daten fließen nur in eine Richtung (vom Eingang zu den Ausgängen).
- **Keine Datenpufferung oder Verzögerung**: Aufgrund des fehlenden Ereignis- und Daten-Handlings arbeitet der Baustein ohne Zwischenspeicher oder Taktung – er ist strikt signaldurchschaltend.

- **Einfaches Interface**: Der Baustein besitzt weder Ereignisse noch Daten-Ein-/Ausgänge, sondern ausschließlich Adapter. Dies macht ihn besonders leichtgewichtig und für reine Signalkopplungen geeignet.

Zustandsübersicht

Der **AIS_SPLIT_9_UNGATED** besitzt keine internen Zustände oder Ablaufsteuerungen. Sein Verhalten ist deterministisch und jederzeit durch das Eingangssignal bestimmt. Es existieren keine initialisierenden oder fehlerhaften Zustände.

Anwendungsszenarien

- **Signalvervielfachung**: Ein AIS-basiertes Steuerungssignal soll parallel an mehrere unabhängige Aktoren, Regler oder Anzeigen weitergegeben werden.
- **Broadcast in Modularen Systemen**: In einer verteilten Automatisierungsumgebung wird ein zentraler Messwert (z. B. Temperatur, Druck) über AIS bereitgestellt und muss gleichzeitig von mehreren Funktionsblöcken konsumiert werden.
- **Prototyping und Test**: Ein Testsignal soll auf mehrere Instanzen eines gleichen FB-Typs verteilt werden, ohne dass der Quell-FB mehrfach instanziiert werden muss.

Vergleich mit ähnlichen Bausteinen

- **AIS_SPLIT_2 / AIS_SPLIT_4 / AIS_SPLIT_8** etc.: Diese Bausteine verteilen das Eingangssignal auf 2, 4 oder 8 Ausgänge. Der **AIS_SPLIT_9_UNGATED** ist speziell für Anwendungen ausgelegt, die genau neun parallele Verbindungen benötigen. Die Funktionsweise ist identisch – der einzige Unterschied liegt in der Anzahl der Ausgänge.
- **AIS_MERGE**: Im Gegensatz zum Split-Baustein fasst ein Merge-Baustein mehrere Eingänge zu einem Ausgang zusammen. **AIS_SPLIT_9_UNGATED** realisiert den umgekehrten Datenfluss.
- **Direkte Kopplung**: Anstelle eines Split-Bausteins könnte man auch eine manuelle Verdrahtung (mehrere OUT-Adapter) in der 4diac-IDE vornehmen. Der **AIS_SPLIT_9_UNGATED** vereinfacht jedoch die grafische Darstellung und reduziert die Komplexität des Netzwerks.
- `AIS_SPLIT_9`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

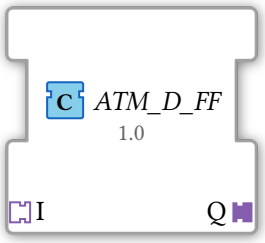
Der **AIS_SPLIT_9_UNGATED** ist ein einfacher, aber äußerst nützlicher Funktionsblock zur Vervielfältigung eines unidirektionalen AIS-Signals auf neun Ausgänge. Seine generische Auslegung, der Verzicht auf Ereignis- oder Datenlogik und die klare Schnittstelle machen ihn zu einem verlässlichen Werkzeug für modulare Automatisierungslösungen. Er eignet sich besonders für Anwendungen, bei denen ein Signal ohne Verzögerung oder Änderung an mehrere Empfänger verteilt werden muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

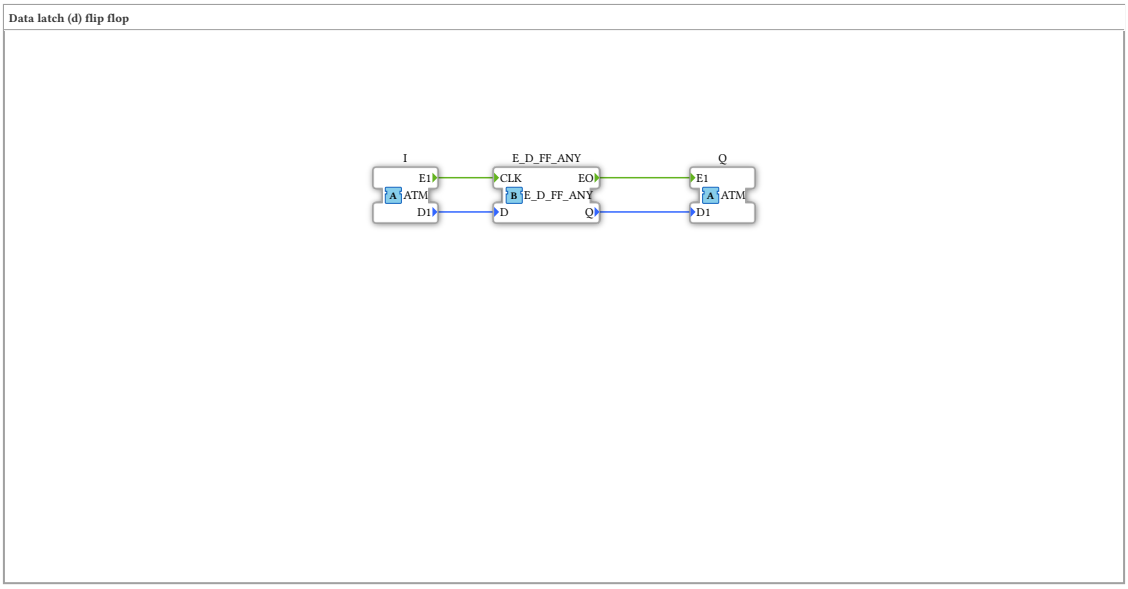
TIME

ATM_D_FF



Value to latch – *ATM*

ATM – Latched value



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `ATM_D_FF` realisiert ein Data Latch (D-Flipflop) zur Speicherung eines einzelnen Datenwertes. Die Kommunikation erfolgt ausschließlich über Adapter vom Typ `adapter::types::unidirectional::ATM`, sodass Ereignisse und Daten als gebündelte Schnittstelle übertragen werden. Der FB ist gemäß IEC 61499-1 Annex A implementiert und steht unter der Eclipse Public License 2.0.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine direkten Ereignis-Eingänge. Über den Socket-Adapter **I** (Typ: `adapter::types::unidirectional::ATM`) wird ein Ereignis an der Adapter-Schnittstelle *E1* entgegengenommen.

Ereignis-Ausgänge

Der FB besitzt keine direkten Ereignis-Ausgänge. Über den Plug-Adapter **Q** (Typ: `adapter::types::unidirectional::ATM`) wird ein Ereignis an der Adapter-Schnittstelle *E1* ausgegeben.

Daten-Eingänge

Der FB besitzt keine direkten Daten-Eingänge. Über den Socket-Adapter **I** wird ein Datenwert an der Adapter-Schnittstelle *D1* bereitgestellt.

Daten-Ausgänge

Der FB besitzt keine direkten Daten-Ausgänge. Über den Plug-Adapter **Q** wird ein Datenwert an der Adapter-Schnittstelle *D1* ausgegeben.

Adapter

Adapter	Typ	Richtung	Kommentar
I	`adapter::types::unidirectional::ATM`	Socket (eingehend)	Wert zum Übernehmen (latch)
Q	`adapter::types::unidirectional::ATM`	Plug (ausgehend)	Gespeicherter Wert

Funktionsweise

Der FB verwendet intern den Standard-Baustein `iec61499::events::E_D_FF_ANY`. Bei einem Ereignis am Eingangsadapter **I.E1** (Takt) wird der aktuelle Datenwert von **I.D1** übernommen und zwischengespei-

chert. Unmittelbar danach wird ein Ereignis am Ausgangsadapter **Q.E1** ausgelöst und der gespeicherte Wert an **Q.D1** ausgegeben. Das Verhalten entspricht einem flankengesteuerten D-Flipflop.

Technische Besonderheiten

- Keine direkten Ereignis- oder Datenports – die gesamte Kommunikation erfolgt über Adapter, was eine flexible Kopplung in IEC 61499-Systemen ermöglicht.
- Der Baustein ist vollständig durch den internen `E_D_FF_ANY` realisiert; die Zustandslogik wird von diesem übernommen.
- Lizenziert unter der Eclipse Public License 2.0; Quelltext und Metadaten sind im FB-Typ hinterlegt.

Zustandsübersicht

Der `ATM_D_FF` besitzt keine eigene Zustandsmaschine. Sein Verhalten ist rein datenflussgesteuert:

- **Warten auf Ereignis** – der zuletzt gespeicherte Wert bleibt erhalten.
- **Bei Ereignis an I.E1** – Wert von I.D1 wird übernommen und sofort an Q ausgegeben.

Intern wird durch `E_D_FF_ANY` eine flankengesteuerte Übernahme realisiert (steigende Flanke des Ereignisses).

Anwendungsszenarien

- **Zwischenspeicher** für Daten, die synchron zu einem Ereignis aktualisiert werden sollen.
- **Zustandsmerkmal** in Automatisierungsabläufen (z. B. Merker für „Wert gültig“).
- **Datenweitergabe** zwischen asynchronen Prozessschritten, wenn ein einfacher Latch ausreicht.
- **Adapter-basierte Kommunikation** in verteilten Systemen, wo flache Schnittstellen bevorzugt werden.

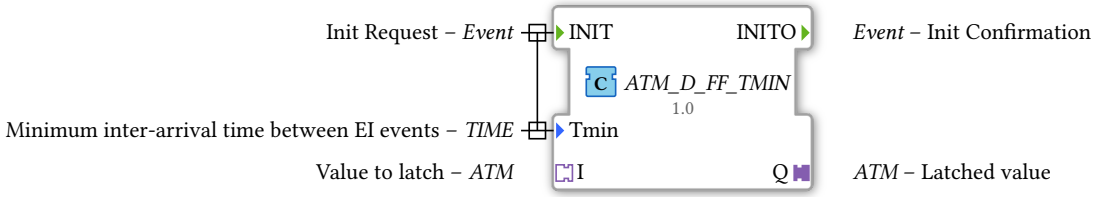
Vergleich mit ähnlichen Bausteinen

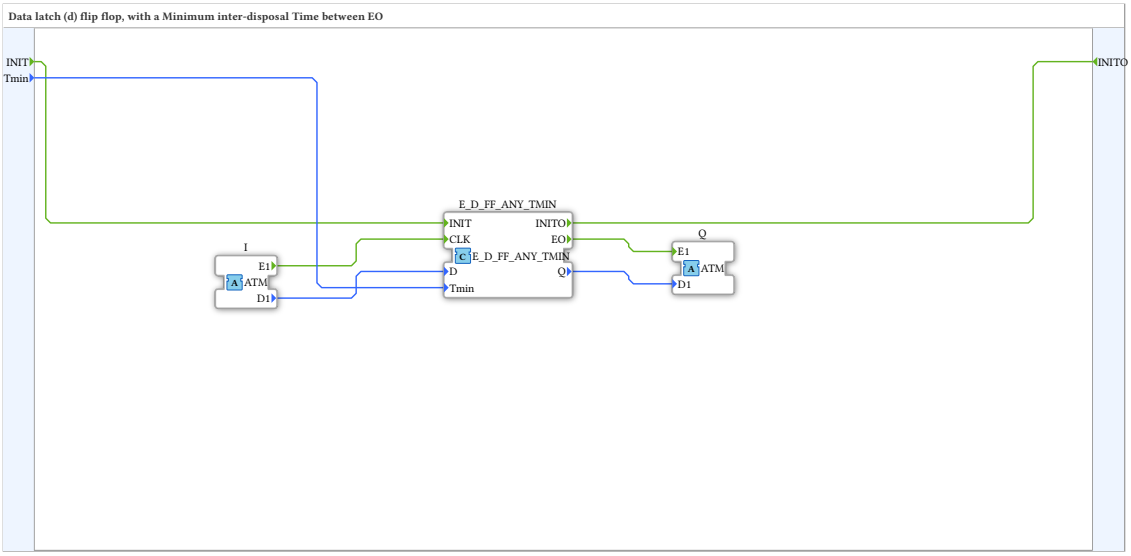
- **SR-Flipflop** (`ATM_SR_FF`): Setzt/Rücksetzt Zustand über zwei Ereignisse, kein taktgesteuerter Dateneingang.
- **T-Flipflop** (`ATM_T_FF`): Toggelt den Zustand bei jedem Ereignis.
- **Direkter `E_D_FF_ANY`**: Bietet die gleiche Logik, jedoch mit expliziten Ein-/Ausgangsports statt Adaptern. Der `ATM_D_FF` kapselt diese Logik und standardisiert die Schnittstelle auf den unidirektionalen ATM-Adapter, was die Wiederverwendung in Adapter-basierten Frameworks erleichtert.

Fazit

`ATM_D_FF` ist ein kompakter, zuverlässiger Funktionsblock zur Realisierung eines D-Flipflops in IEC 61499-Umgebungen. Durch den Einsatz von Adaptern wird eine saubere Trennung von Ereignis- und Datenpfaden erreicht, die insbesondere in modularen, wiederverwendbaren Automatisierungslösungen von Vorteil ist. Der Baustein ist einfach zu integrieren und erfüllt die grundlegende Anforderung an einen flankengesteuerten Datenspeicher.

ATM_D_FF_TMIN





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `ATM_D_FF_TMIN` realisiert ein taktgesteuertes D-Flipflop (Data Latch), das den anliegenden Datenwert bei einem Taktereignis übernimmt und als gespeicherten Wert ausgibt. Eine Besonderheit ist die einstellbare Mindestzeit (`Tmin`) zwischen zwei aufeinanderfolgenden Ausgangsereignissen, die eine definierte Totzeit garantiert.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Initialisierungsanforderung (mit `Tmin`)

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Initialisierungsbestätigung

Daten-Eingänge

Name	Typ	Kommentar
Tmin	TIME	Minimaler zeitlicher Abstand zwischen zwei EO-Ereignissen

Daten-Ausgänge

Der Baustein besitzt keine direkten Datenausgänge. Der gespeicherte Wert wird über den Adapter-Ausgang `Q.D1` bereitgestellt.

Adapter

Richtung	Name	Typ	Kommentar
Eingang (Socket)	I	`adapter::types::unidirectional::ATM`	Zu übernehmender Wert
Ausgang (Plug)	Q	`adapter::types::unidirectional::ATM`	Gespeicherter Wert

Der Adapter `ATM` bietet je einen Ereignis- und einen Datenkanal (`E1`, `D1` für Eingang; `E1`, `D1` für Ausgang).

Funktionsweise

1. **Initialisierung**: Durch ein Ereignis an `INIT` wird der Baustein initialisiert und der Parameter `Tmin` übernommen.
2. **Takten**: Ein Ereignis am Adapter-Eingang `I.E1` (Takt) veranlasst den Baustein, den aktuellen Datenwert von `I.D1` zu übernehmen. Bei gültiger Übernahme wird der Wert am Adapter-Ausgang `Q.D1` ausgegeben und gleichzeitig ein Ereignis an `Q.E1` gesendet.
3. **Minimale Ausgangsverzögerung**: Nach jedem ausgegebenen Ereignis (`Q.E1`) wird eine interne Sperre aktiviert. Erst nach Ablauf der Zeit `Tmin` kann ein weiteres Taktereignis zu einer erneuten Ausgabe führen. Geht innerhalb der Sperrzeit ein neues Taktsignal ein, wird es ignoriert oder die Ausgabe bis zum Ablauf von `Tmin` verzögert (abhängig von der Implementierung des internen FB `E_D_FF_ANY_TMIN`).

Technische Besonderheiten

- **Totzeit-Regelung**: Der Parameter `Tmin` erzwingt eine minimale Pause zwischen zwei Ausgangsereignissen – nützlich, um nachgeschaltete Komponenten zu schützen oder Signalflanken zu entprellen.
- **Interne Realisierung**: Der Baustein verwendet intern den FB `iec61499::events::E_D_FF_ANY_TMIN`, der die eigentliche Flipflop-Logik mit Zeitüberwachung bereitstellt. Der `ATM_D_FF_TMIN` fungiert als Schnittstellenanpassung für den Adapter `ATM`.
- **Adapter-Schnittstelle**: Sowohl Eingangs- als auch Ausgangsdaten werden über unidirektionale Adapter ausgetauscht, was eine lose Kopplung in der Applikation ermöglicht.

Zustandsübersicht

Obwohl kein explizites Zustandsdiagramm vorliegt, lassen sich folgende logische Zustände ableiten:

1. **Bereit**: Wartet auf ein Taktereignis, keine Sperre aktiv.
2. **Gesetzt**: Wert wurde übernommen und ausgegeben, die Sperre `Tmin` läuft.
3. **Gesperrt**: Während `Tmin` werden eingehende Taktereignisse ignoriert oder zurückgehalten.

Nach Ablauf der Sperre kehrt der Baustein in den Zustand **Bereit** zurück.

Anwendungsszenarien

- **Signalaufbereitung** in der Automatisierungstechnik, bei der ein Schaltsignal nach einer definierten Mindestpause erneut ausgewertet werden soll (z. B. Totzeit bei Ventilumschaltungen).
- **Entprellung** von binären Sensoren, bei denen ein nachgeschalteter Ausgang nur nach einer stabilen Zeit von `Tmin` erzeugt werden darf.
- **Synchronisation** von Datenströmen, bei denen eine minimale Paketlücke eingehalten werden muss.

Vergleich mit ähnlichen Bausteinen

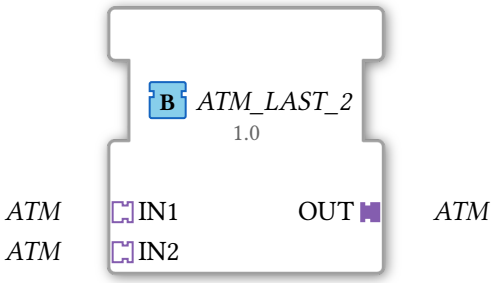
- **Standard-D-Flipflop** (z. B. `E_D_FF`): Liefert sofort nach jedem Takt einen Ausgang – ohne zeitliche Einschränkung. Der `ATM_D_FF_TMIN` erweitert dies um die Totzeit `Tmin`.
- **Monostabile Kippglieder** (Timer): Erzeugen einen Impuls definierter Länge. Der `ATM_D_FF_TMIN` hingegen sperrt nur die Ausgabe, ohne den Ausgangspegel zurückzusetzen.

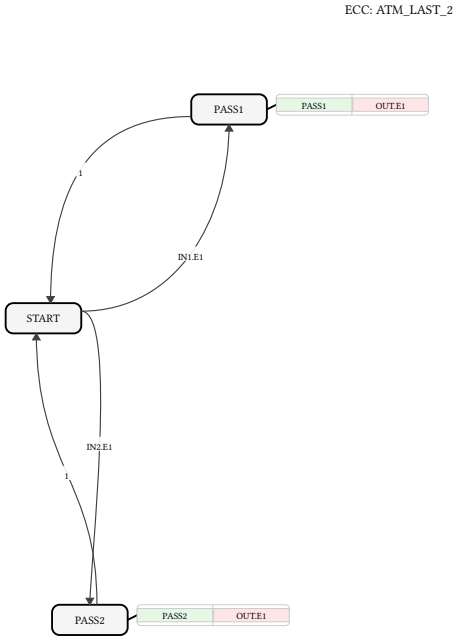
Der vorliegende Baustein vereint die Eigenschaften eines Flipflops mit einer einstellbaren Mindestausblendzeit und ist daher für zeitkritische Steuerungsaufgaben besonders geeignet.

Fazit

`ATM_D_FF_TMIN` ist ein spezialisierter D-Flipflop-Baustein, der die minimale Verweilzeit zwischen zwei Ausgangsereignissen garantiert. Die Nutzung von Adaptern vereinfacht die Integration in modulare Steuerungsanwendungen, während die einstellbare Totzeit hohe Flexibilität für unterschiedliche physikalische Randbedingungen bietet. Er ist eine sinnvolle Erweiterung der IEC-61499-Bibliothek für zeitdiskrete Signalverarbeitung.

ATM_LAST_2





Einleitung

Der ATM_LAST_2 Funktionsblock führt zwei gleichartige TIME-Adapter-Signale (**IN1**, **IN2**) auf einen gemeinsamen Ausgang (**OUT**) zusammen, nach dem Prinzip **Last Writer Wins**: welcher der beiden Sockets zuletzt ein Ereignis geschrieben hat, gewinnt, und sein Datenwert wird sofort und unverändert an OUT durchgereicht. Es handelt sich ausdrücklich nicht um ein naives Zusammenführen (z. B. Mittelwert oder ODER-Verknüpfung), sondern um eine reine Arbitrierung nach zeitlicher Reihenfolge.

Schnittstellenstruktur

Adapter

Eingangsadapter (Sockets):

- **IN1**: TIME-Adapter (unidirectional) - erste Signalquelle
- **IN2**: TIME-Adapter (unidirectional) - zweite Signalquelle

Ausgangsadapter (Plug):

- **OUT**: TIME-Adapter (unidirectional) - zusammengeführtes Signal

Funktionsweise

Der Baustein ist als einfacher Basic-FB mit einer 3-Zustands-ECC umgesetzt (START, PASS1, PASS2). Trifft an IN1 ein Ereignis ein, wechselt die ECC nach PASS1: der aktuelle Datenwert von IN1 wird nach OUT kopiert und das Adapter-Ereignis an OUT gesendet. Trifft an IN2 ein Ereignis ein, geschieht dasselbe

symmetrisch über PASS2. Nach jeder Durchreichung kehrt die ECC sofort wieder nach START zurück und ist bereit für das nächste Ereignis von IN1 oder IN2.

Technische Besonderheiten

- Last-Writer-Wins-Semantik statt naiver Datenverknüpfung
- Einfacher Basic-FB (ECC mit 3 Zuständen), kein zusätzlicher Zwischenspeicher nötig
- Keine Signalverzögerung: der Wert wird im selben Zyklus durchgereicht, in dem das auslösende Ereignis eintrifft
- Beide Eingänge sind gleichberechtigt - es gibt keine feste Priorität außer der zeitlichen Reihenfolge

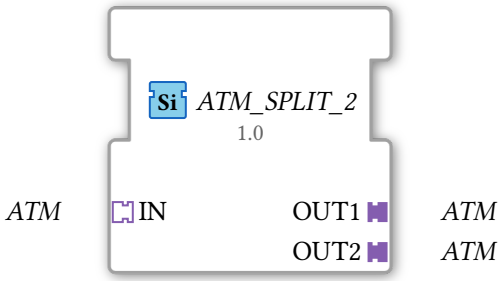
Anwendungsszenarien

- Zusammenführen zweier gleichwertiger TIME-Signalquellen zu einem einzigen nachgeschalteten Adapterpfad (z. B. zwei redundante Sensoren, oder ein manueller und ein automatischer Sollwertgeber)
- Arbitrierung zwischen zwei Bedienstellen, die denselben TIME-Wert schreiben dürfen
- Vereinfachung von Netzwerken, die sonst ein `F_SEL`/`E_RS`-Konstrukt inline verdrahten müssten

⚖️ Vergleich mit ähnlichen Bausteinen

ATM_LAST_2 ist die Umkehrung von `ATM_SPLIT_2`: ATM_SPLIT_2 verteilt 1 Eingang auf 2 Ausgänge, ATM_LAST_2 führt 2 Eingänge auf 1 Ausgang zusammen. Für Mischbetrieb zwischen einem Daten- und einem reinen Ereignisadapter (z. B. AX + AE) ist stattdessen `AX_AE_MERGE` gedacht, das nur das Ereignis mischt, ohne den Datenwert zu verändern.

ATM_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **ATM_SPLIT_2** ist ein generischer Funktionsbaustein, der einen eingehenden Adapter vom Typ `adapter::types::unidirectional::ATM` auf 2 separate Ausgänge (OUT1, OUT2) aufteilt. Er dient zur Weiterleitung eines Zeitsignals an 2 nachfolgende Bausteine, ohne die Daten zu verändern. Der Baustein ist als generischer Typ (`GEN_ATM_SPLIT`) implementiert und wird zur Laufzeit parametrisiert.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Typ	Name	Richtung	Beschreibung

`adapter::types::unidirectional::ATM`	IN	Socket	Eingangssignal (ATM)
`adapter::types::unidirectional::ATM`	OUT1	Plug	Ausgang 1 (identisch zu IN)
`adapter::types::unidirectional::ATM`	OUT2	Plug	Ausgang 2 (identisch zu IN)

Funktionsweise

Der Baustein leitet das am Socket **IN** anliegende ATM-Signal unverändert auf alle 2 Plugs (OUT1, OUT2) weiter. Es findet keine Datenmanipulation, Filterung oder Verzögerung statt. Die Aufteilung erfolgt rein strukturell: Jeder Ausgang erhält eine eigene Kopie der Referenz auf den zugrundeliegenden Zeitwert.

Technische Besonderheiten

- **Generische Implementierung**: Der FB verwendet das Generic-Class-Name-Attribut (`eclipse4diac::core::GenericClassName`) mit dem Wert `'GEN_ATM_SPLIT'`, sodass dieselbe Klasse über den GenericClassName-Mechanismus die Aritäten `ATM_SPLIT_2`, `ATM_SPLIT_3` und `ATM_SPLIT_4` abdeckt.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `adapter::types::unidirectional::ATM` (nur Vorwärtsrichtung).
- **Keine Zustandsautomatik**: Der Baustein besitzt keinen expliziten ECC (Execution Control Chart); die Weiterleitung erfolgt direkt und ereignisunabhängig.

Zustandsübersicht

Der Baustein implementiert keine Zustandsautomaten. Die Funktionalität beschränkt sich auf die passive Weiterleitung des Eingangssignals an alle 2 Ausgänge. Eine Zustandsvisualisierung ist daher nicht erforderlich.

Anwendungsszenarien

- **Signalverteilung**: Aufteilung eines ATM-basierten Zeitsignals (z. B. einer Verzögerungsdauer) an mehrere parallel arbeitende Steuerungskomponenten.
- **Redundanz**: Bereitstellung desselben Zeitwerts für ein primäres und ein redundantes System.
- **Debugging**: Anschluss eines Analyse- oder Logging-Bausteins parallel zum bestehenden Pfad, ohne die Originalsignalkette zu unterbrechen.

Vergleich mit ähnlichen Bausteinen

Ähnliche Funktionalität bieten die weiteren Aritäten `ATM_SPLIT_2`/`ATM_SPLIT_3`/`ATM_SPLIT_4` sowie der strukturell identische `AR_SPLIT_2` für den Datentyp `REAL`. Die Auswahl der Arität hängt von der benötigten Anzahl an Ausgängen ab.

- `ATM_SPLIT_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

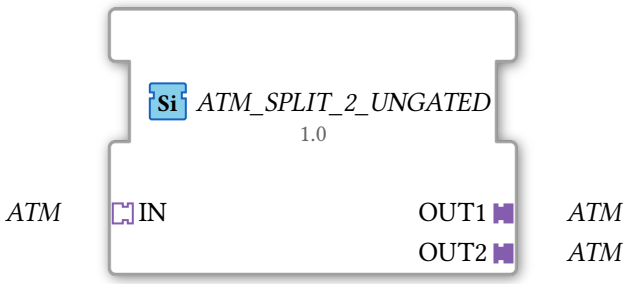
Fazit

Der **ATM_SPLIT_2** ist ein einfacher, aber essenzieller Baustein zur Vervielfachung eines Zeitsignals in IEC 61499-basierten Steuerungssystemen. Seine generische Auslegung und die klare Schnittstelle machen ihn zur ersten Wahl, wenn ein Zeitwert an mehrere unabhängige Zielbausteine weitergegeben werden muss.

ATM_SPLIT_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ATM_SPLIT_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht

gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **ATM_SPLIT_2_UNGATED** ist ein generischer Funktionsbaustein, der einen eingehenden Adapter vom Typ `adapter::types::unidirectional::ATM` auf 2 separate Ausgänge (OUT1, OUT2) aufteilt. Er dient zur Weiterleitung eines Zeitsignals an 2 nachfolgende Bausteine, ohne die Daten zu verändern. Der Baustein ist als generischer Typ (`GEN_ATM_SPLIT`) implementiert und wird zur Laufzeit parametrisiert.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::ATM`	IN	Socket	Eingangssignal (ATM)
`adapter::types::unidirectional::ATM`	OUT1	Plug	Ausgang 1 (identisch zu IN)
`adapter::types::unidirectional::ATM`	OUT2	Plug	Ausgang 2 (identisch zu IN)

Funktionsweise

Der Baustein leitet das am Socket **IN** anliegende ATM-Signal unverändert auf alle 2 Plugs (OUT1, OUT2) weiter. Es findet keine Datenmanipulation, Filterung oder Verzögerung statt. Die Aufteilung erfolgt rein strukturell: Jeder Ausgang erhält eine eigene Kopie der Referenz auf den zugrundeliegenden Zeitwert.

Technische Besonderheiten

- **Generische Implementierung**: Der FB verwendet das Generic-Class-Name-Attribut (`eclipse4diac::core::GenericClassName`) mit dem Wert `'GEN_ATM_SPLIT'`, sodass dieselbe Klasse über den GenericClassName-Mechanismus die Aritäten `ATM_SPLIT_2_UNGATED`, `ATM_SPLIT_3` und `ATM_SPLIT_4` abdeckt.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `adapter::types::unidirectional::ATM` (nur Vorwärtsrichtung).
- **Keine Zustandsautomatik**: Der Baustein besitzt keinen expliziten ECC (Execution Control Chart); die Weiterleitung erfolgt direkt und ereignisunabhängig.

Zustandsübersicht

Der Baustein implementiert keine Zustandsautomaten. Die Funktionalität beschränkt sich auf die passive Weiterleitung des Eingangssignals an alle 2 Ausgänge. Eine Zustandsvisualisierung ist daher nicht erforderlich.

Anwendungsszenarien

- **Signalverteilung**: Aufteilung eines ATM-basierten Zeitsignals (z. B. einer Verzögerungsdauer) an mehrere parallel arbeitende Steuerungskomponenten.
- **Redundanz**: Bereitstellung desselben Zeitwerts für ein primäres und ein redundantes System.
- **Debugging**: Anschluss eines Analyse- oder Logging-Bausteins parallel zum bestehenden Pfad, ohne die Originalsignalkette zu unterbrechen.

Vergleich mit ähnlichen Bausteinen

Ähnliche Funktionalität bieten die weiteren Aritäten `ATM_SPLIT_2_UNGATED`/`ATM_SPLIT_3`/`ATM_SPLIT_4` sowie der strukturell identische `AR_SPLIT_2` für den Datentyp `REAL`. Die Auswahl der Arität hängt von der benötigten Anzahl an Ausgängen ab.

- `ATM_SPLIT_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

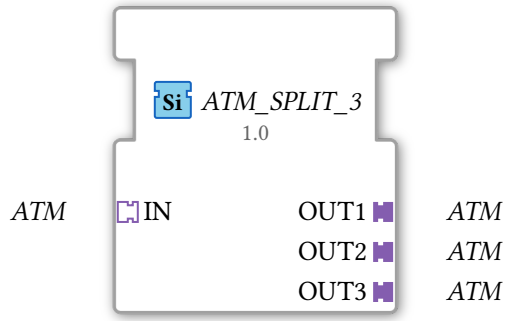
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **ATM_SPLIT_2_UNGATED** ist ein einfacher, aber essenzieller Baustein zur Vervielfachung eines Zeitsignals in IEC 61499-basierten Steuerungssystemen. Seine generische Auslegung und die klare Schnittstelle machen ihn zur ersten Wahl, wenn ein Zeitwert an mehrere unabhängige Zielbausteine weitergegeben werden muss.

ATM_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **ATM_SPLIT_3** ist ein generischer Funktionsbaustein, der einen eingehenden Adapter vom Typ `adapter::types::unidirectional::ATM` auf 3 separate Ausgänge (OUT1, OUT2, OUT3) aufteilt. Er dient zur Weiterleitung eines Zeitsignals an 3 nachfolgende Bausteine, ohne die Daten zu verändern. Der Baustein ist als generischer Typ (`GEN_ATM_SPLIT`) implementiert und wird zur Laufzeit parametrisiert.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::ATM`	IN	Socket	Eingangssignal (ATM)
`adapter::types::unidirectional::ATM`	OUT1	Plug	Ausgang 1 (identisch zu IN)
`adapter::types::unidirectional::ATM`	OUT2	Plug	Ausgang 2 (identisch zu IN)
`adapter::types::unidirectional::ATM`	OUT3	Plug	Ausgang 3 (identisch zu IN)

Funktionsweise

Der Baustein leitet das am Socket **IN** anliegende ATM-Signal unverändert auf alle 3 Plugs (OUT1, OUT2, OUT3) weiter. Es findet keine Datenmanipulation, Filterung oder Verzögerung statt. Die Aufteilung erfolgt rein strukturell: Jeder Ausgang erhält eine eigene Kopie der Referenz auf den zugrundeliegenden Zeitwert.

Technische Besonderheiten

- **Generische Implementierung**: Der FB verwendet das Generic-Class-Name-Attribut (`eclipse4diac::core::GenericClassName`) mit dem Wert `'GEN_ATM_SPLIT'`, sodass dieselbe Klasse über den GenericClassName-Mechanismus die Aritäten `ATM_SPLIT_2`, `ATM_SPLIT_3` und `ATM_SPLIT_4` abdeckt.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `adapter::types::unidirectional::ATM` (nur Vorwärtsrichtung).
- **Keine Zustandsautomatik**: Der Baustein besitzt keinen expliziten ECC (Execution Control Chart); die Weiterleitung erfolgt direkt und ereignisunabhängig.

Zustandsübersicht

Der Baustein implementiert keine Zustandsautomaten. Die Funktionalität beschränkt sich auf die passive Weiterleitung des Eingangssignals an alle 3 Ausgänge. Eine Zustandsvisualisierung ist daher nicht erforderlich.

Anwendungsszenarien

- **Signalverteilung**: Aufteilung eines ATM-basierten Zeitsignals (z. B. einer Verzögerungsdauer) an mehrere parallel arbeitende Steuerungskomponenten.
- **Redundanz**: Bereitstellung desselben Zeitwerts für ein primäres und ein redundantes System.
- **Debugging**: Anschluss eines Analyse- oder Logging-Bausteins parallel zum bestehenden Pfad, ohne die Originalsignalkette zu unterbrechen.

Vergleich mit ähnlichen Bausteinen

Ähnliche Funktionalität bieten die weiteren Aritäten `ATM_SPLIT_2`/`ATM_SPLIT_3`/`ATM_SPLIT_4` sowie der strukturell identische `AR_SPLIT_3` für den Datentyp `REAL`. Die Auswahl der Arität hängt von der benötigten Anzahl an Ausgängen ab.

- `ATM_SPLIT_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

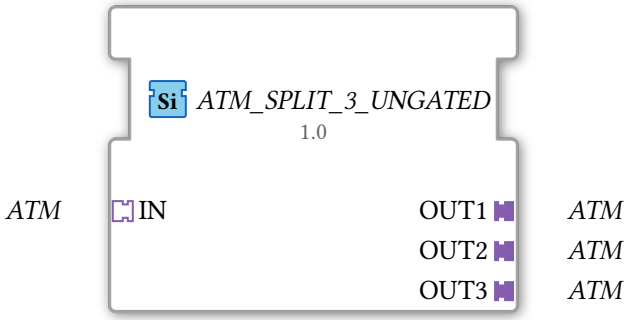
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der **ATM_SPLIT_3** ist ein einfacher, aber essenzieller Baustein zur Vervielfachung eines Zeitsignals in IEC 61499-basierten Steuerungssystemen. Seine generische Auslegung und die klare Schnittstelle machen ihn zur ersten Wahl, wenn ein Zeitwert an mehrere unabhängige Zielbausteine weitergegeben werden muss.

ATM_SPLIT_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ATM_SPLIT_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **ATM_SPLIT_3_UNGATED** ist ein generischer Funktionsbaustein, der einen eingehenden Adapter vom Typ `adapter::types::unidirectional::ATM` auf 3 separate Ausgänge (OUT1, OUT2, OUT3) aufteilt. Er dient zur Weiterleitung eines Zeitsignals an 3 nachfolgende Bausteine, ohne die Daten zu verändern. Der Baustein ist als generischer Typ (`GEN_ATM_SPLIT`) implementiert und wird zur Laufzeit parametrisiert.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::ATM`	IN	Socket	Eingangssignal (ATM)
`adapter::types::unidirectional::ATM`	OUT1	Plug	Ausgang 1 (identisch zu IN)
`adapter::types::unidirectional::ATM`	OUT2	Plug	Ausgang 2 (identisch zu IN)

`adapter::types::unidirectional::ATM`	OUT3	Plug	Ausgang 3 (identisch zu IN)

Funktionsweise

Der Baustein leitet das am Socket **IN** anliegende ATM-Signal unverändert auf alle 3 Plugs (OUT1, OUT2, OUT3) weiter. Es findet keine Datenmanipulation, Filterung oder Verzögerung statt. Die Aufteilung erfolgt rein strukturell: Jeder Ausgang erhält eine eigene Kopie der Referenz auf den zugrundeliegenden Zeitwert.

Technische Besonderheiten

- **Generische Implementierung**: Der FB verwendet das Generic-Class-Name-Attribut (`eclipse4diac::core::GenericClassName`) mit dem Wert `'GEN_ATM_SPLIT'`, sodass dieselbe Klasse über den GenericClassName-Mechanismus die Aritäten `ATM_SPLIT_2`, `ATM_SPLIT_3_UNGATED` und `ATM_SPLIT_4` abdeckt.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `adapter::types::unidirectional::ATM` (nur Vorwärtsrichtung).
- **Keine Zustandsautomatik**: Der Baustein besitzt keinen expliziten ECC (Execution Control Chart); die Weiterleitung erfolgt direkt und ereignisunabhängig.

Zustandsübersicht

Der Baustein implementiert keine Zustandsautomaten. Die Funktionalität beschränkt sich auf die passive Weiterleitung des Eingangssignals an alle 3 Ausgänge. Eine Zustandsvisualisierung ist daher nicht erforderlich.

Anwendungsszenarien

- **Signalverteilung**: Aufteilung eines ATM-basierten Zeitsignals (z. B. einer Verzögerungsdauer) an mehrere parallel arbeitende Steuerungskomponenten.
- **Redundanz**: Bereitstellung desselben Zeitwerts für ein primäres und ein redundantes System.
- **Debugging**: Anschluss eines Analyse- oder Logging-Bausteins parallel zum bestehenden Pfad, ohne die Originalsignalkette zu unterbrechen.

Vergleich mit ähnlichen Bausteinen

Ähnliche Funktionalität bieten die weiteren Aritäten `ATM_SPLIT_2`/`ATM_SPLIT_3_UNGATED`/`ATM_SPLIT_4` sowie der strukturell identische `AR_SPLIT_3` für den Datentyp `REAL`. Die Auswahl der Arität hängt von der benötigten Anzahl an Ausgängen ab.

- `ATM_SPLIT_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

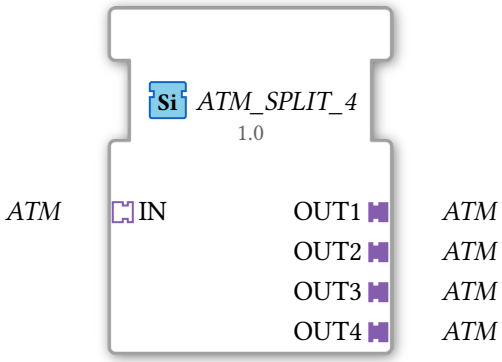
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **ATM_SPLIT_3_UNGATED** ist ein einfacher, aber essenzieller Baustein zur Vervielfachung eines Zeitsignals in IEC 61499-basierten Steuerungssystemen. Seine generische Auslegung und die klare Schnittstelle machen ihn zur ersten Wahl, wenn ein Zeitwert an mehrere unabhängige Zielbausteine weitergegeben werden muss.

ATM_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **ATM_SPLIT_4** ist ein generischer Funktionsbaustein, der einen eingehenden Adapter vom Typ `adapter::types::unidirectional::ATM` auf 4 separate Ausgänge (OUT1, OUT2, OUT3, OUT4) aufteilt. Er dient zur Weiterleitung eines Zeitsignals an 4 nachfolgende Bausteine, ohne die Daten zu verändern. Der Baustein ist als generischer Typ (`GEN_ATM_SPLIT`) implementiert und wird zur Laufzeit parametrisiert.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::ATM`	IN	Socket	Eingangssignal (ATM)
`adapter::types::unidirectional::ATM`	OUT1	Plug	Ausgang 1 (identisch zu IN)
`adapter::types::unidirectional::ATM`	OUT2	Plug	Ausgang 2 (identisch zu IN)
`adapter::types::unidirectional::ATM`	OUT3	Plug	Ausgang 3 (identisch zu IN)
`adapter::types::unidirectional::ATM`	OUT4	Plug	Ausgang 4 (identisch zu IN)

Funktionsweise

Der Baustein leitet das am Socket **IN** anliegende ATM-Signal unverändert auf alle 4 Plugs (OUT1, OUT2, OUT3, OUT4) weiter. Es findet keine Datenmanipulation, Filterung oder Verzögerung statt. Die Aufteilung erfolgt rein strukturell: Jeder Ausgang erhält eine eigene Kopie der Referenz auf den zugrundeliegenden Zeitwert.

Technische Besonderheiten

- **Generische Implementierung**: Der FB verwendet das Generic-Class-Name-Attribut (`eclipse4diac::core::GenericClassName`) mit dem Wert `'GEN_ATM_SPLIT'`, sodass dieselbe Klasse über den GenericClassName-Mechanismus die Aritäten `ATM_SPLIT_2`, `ATM_SPLIT_3` und `ATM_SPLIT_4` abdeckt.

- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `adapter::types::unidirectional::ATM` (nur Vorwärtsrichtung).
- **Keine Zustandsautomatik**: Der Baustein besitzt keinen expliziten ECC (Execution Control Chart); die Weiterleitung erfolgt direkt und ereignisunabhängig.

Zustandsübersicht

Der Baustein implementiert keine Zustandsautomaten. Die Funktionalität beschränkt sich auf die passive Weiterleitung des Eingangssignals an alle 4 Ausgänge. Eine Zustandsvisualisierung ist daher nicht erforderlich.

Anwendungsszenarien

- **Signalverteilung**: Aufteilung eines ATM-basierten Zeitsignals (z. B. einer Verzögerungsdauer) an mehrere parallel arbeitende Steuerungskomponenten.
- **Redundanz**: Bereitstellung desselben Zeitwerts für ein primäres und ein redundantes System.
- **Debugging**: Anschluss eines Analyse- oder Logging-Bausteins parallel zum bestehenden Pfad, ohne die Originalsignalkette zu unterbrechen.

Vergleich mit ähnlichen Bausteinen

Ähnliche Funktionalität bieten die weiteren Aritäten `ATM_SPLIT_2`/`ATM_SPLIT_3`/`ATM_SPLIT_4` sowie der strukturell identische `AR_SPLIT_4` für den Datentyp `REAL`. Die Auswahl der Arität hängt von der benötigten Anzahl an Ausgängen ab.

- `ATM_SPLIT_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

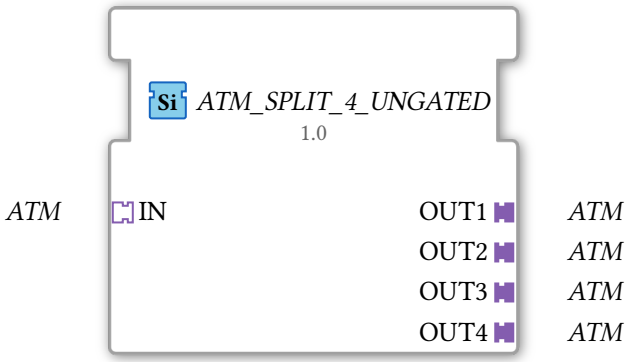
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der **ATM_SPLIT_4** ist ein einfacher, aber essenzieller Baustein zur Vervielfachung eines Zeitsignals in IEC 61499-basierten Steuerungssystemen. Seine generische Auslegung und die klare Schnittstelle machen ihn zur ersten Wahl, wenn ein Zeitwert an mehrere unabhängige Zielbausteine weitergegeben werden muss.

ATM_SPLIT_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ATM_SPLIT_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **ATM_SPLIT_4_UNGATED** ist ein generischer Funktionsbaustein, der einen eingehenden Adapter vom Typ `adapter::types::unidirectional::ATM` auf 4 separate Ausgänge (OUT1, OUT2, OUT3, OUT4) aufteilt. Er dient zur Weiterleitung eines Zeitsignals an 4 nachfolgende Bausteine, ohne die Daten zu verändern. Der Baustein ist als generischer Typ (`GEN_ATM_SPLIT`) implementiert und wird zur Laufzeit parametrisiert.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::ATM`	IN	Socket	Eingangssignal (ATM)
`adapter::types::unidirectional::ATM`	OUT1	Plug	Ausgang 1 (identisch zu IN)
`adapter::types::unidirectional::ATM`	OUT2	Plug	Ausgang 2 (identisch zu IN)
`adapter::types::unidirectional::ATM`	OUT3	Plug	Ausgang 3 (identisch zu IN)
`adapter::types::unidirectional::ATM`	OUT4	Plug	Ausgang 4 (identisch zu IN)

Funktionsweise

Der Baustein leitet das am Socket **IN** anliegende ATM-Signal unverändert auf alle 4 Plugs (OUT1, OUT2, OUT3, OUT4) weiter. Es findet keine Datenmanipulation, Filterung oder Verzögerung statt. Die Aufteilung erfolgt rein strukturell: Jeder Ausgang erhält eine eigene Kopie der Referenz auf den zugrundeliegenden Zeitwert.

Technische Besonderheiten

- **Generische Implementierung**: Der FB verwendet das Generic-Class-Name-Attribut (`eclipse4diac::core::GenericClassName`) mit dem Wert `'GEN_ATM_SPLIT'`, sodass dieselbe Klasse über den GenericClassName-Mechanismus die Aritäten `ATM_SPLIT_2`, `ATM_SPLIT_3` und `ATM_SPLIT_4_UNGATED` abdeckt.
- **Unidirektionale Adapter**: Alle Adapter sind vom Typ `adapter::types::unidirectional::ATM` (nur Vorwärtsrichtung).
- **Keine Zustandsautomatik**: Der Baustein besitzt keinen expliziten ECC (Execution Control Chart); die Weiterleitung erfolgt direkt und ereignisunabhängig.

Zustandsübersicht

Der Baustein implementiert keine Zustandsautomaten. Die Funktionalität beschränkt sich auf die passive Weiterleitung des Eingangssignals an alle 4 Ausgänge. Eine Zustandsvisualisierung ist daher nicht erforderlich.

Anwendungsszenarien

- **Signalverteilung**: Aufteilung eines ATM-basierten Zeitsignals (z. B. einer Verzögerungsdauer) an mehrere parallel arbeitende Steuerungskomponenten.

- **Redundanz**: Bereitstellung desselben Zeitwerts für ein primäres und ein redundantes System.
- **Debugging**: Anschluss eines Analyse- oder Logging-Bausteins parallel zum bestehenden Pfad, ohne die Originalsignalkette zu unterbrechen.

Vergleich mit ähnlichen Bausteinen

Ähnliche Funktionalität bieten die weiteren Aritäten `ATM_SPLIT_2`/`ATM_SPLIT_3`/`ATM_SPLIT_4_UNGATED` sowie der strukturell identische `AR_SPLIT_4` für den Datentyp `REAL`. Die Auswahl der Arität hängt von der benötigten Anzahl an Ausgängen ab.

- `ATM_SPLIT_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

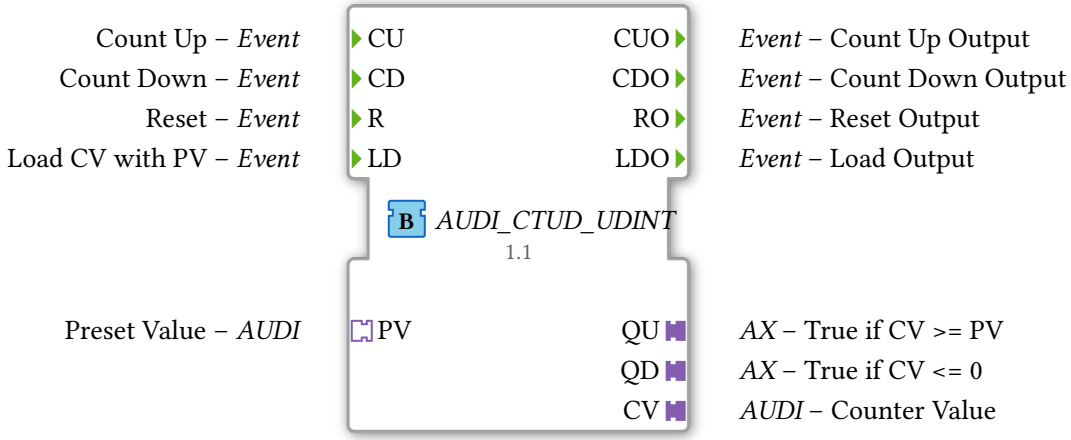
Der **ATM_SPLIT_4_UNGATED** ist ein einfacher, aber essenzieller Baustein zur Vervielfachung eines Zeitsignals in IEC 61499-basierten Steuerungssystemen. Seine generische Auslegung und die klare Schnittstelle machen ihn zur ersten Wahl, wenn ein Zeitwert an mehrere unabhängige Zielbausteine weitergegeben werden muss.

UDINT

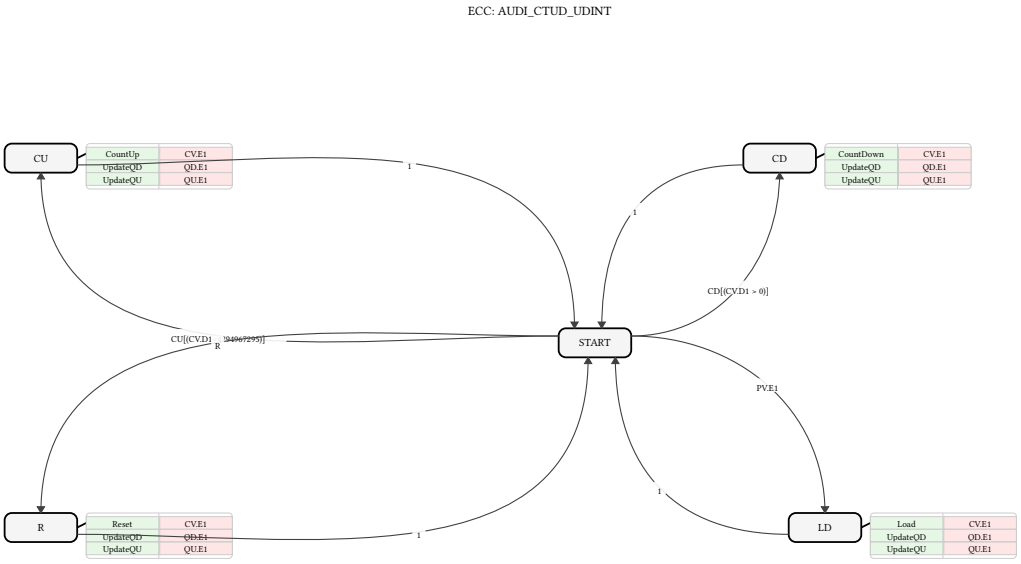
AUDI_CTUD_UDINT (Adapter-basierter Auf-/Abwärtszähler)

Einleitung

Der `AUDI_CTUD_UDINT` ist ein ereignisgesteuerter Auf-/Abwärtszähler (Counter Up/Down) für vorzeichenlose 32-Bit-Ganzzahlen (UDINT), der speziell für die Integration in adapterbasierte Systeme entwickelt wurde. Er nutzt `AUDI`-Adapter für die Übergabe des Zählwerts und des Vorgabewerts, was eine saubere Trennung von Ereignis- und Datenfluss ermöglicht.



Schnittstellenstruktur



Ereignis-Eingänge

- **CU**: `Event` - Zählt den Wert um eins hoch (`Count Up`; nur wenn `CV.D1` < 4294967295).
- **CD**: `Event` - Zählt den Wert um eins herunter (`Count Down`; nur wenn `CV.D1` > 0).
- **R**: `Event` - Setzt den Zähler auf Null zurück (`Reset`).
- **LD**: `Event` - Lädt den Zählwert `CV.D1` mit dem Vorgabewert `PV.D1` (`Load CV with PV`).

Ereignis-Ausgänge

- **CUO**: `Event` - Bestätigung von `CU` (`Count Up Output`).
- **CDO**: `Event` - Bestätigung von `CD` (`Count Down Output`).
- **RO**: `Event` - Bestätigung von `R` (`Reset Output`).
- **LDO**: `Event` - Bestätigung von `LD` (`Load Output`).

Output-Variablen

Keine klassischen Daten-Ausgänge; die Statussignale `QU` und `QD` werden als AX-Adapter (Plugs) ausgegeben, siehe Abschnitt Adapter.

Adapter

- **QU** (Plug): `AX` - `TRUE`, wenn der Zählwert (`CV.D1`) größer oder gleich dem Vorgabewert (`PV.D1`) ist. Das Ereignis `QU.E1` wird nur bei einer Änderung von `QU` ausgelöst.
- **QD** (Plug): `AX` - `TRUE`, wenn der Zählwert (`CV.D1`) kleiner oder gleich Null ist. Das Ereignis `QD.E1` wird nur bei einer Änderung von `QD` ausgelöst.
- **CV** (Plug): `AUDI` - Der aktuelle Zählwert (`Counter Value`).
- **PV** (Socket): `AUDI` - Der Vorgabewert (`Preset Value`), gegen den der Zähler geprüft wird (für `QU`).

Funktionsweise

Der Zähler reagiert auf die Ereignis-Eingänge `CU`, `CD`, `R` und `LD`. Ein `CU`-Ereignis erhöht `CV.D1` um 1, ein `CD`-Ereignis verringert `CV.D1` um 1. Ein `R`-Ereignis setzt `CV.D1` auf 0, ein `LD`-Ereignis setzt `CV.D1` auf `PV.D1`.

Trifft am `PV`-Adapter ein Ereignis (`PV.E1`) ein, werden nur die Statussignale `QU`/`QD` neu bewertet; der Zählwert `CV.D1` bleibt unverändert.

Der Zählwert wird über den `CV`-Adapter als `AUDI`-Signal ausgegeben. Nach jeder Zählaktion bestätigen `CUO`, `CDO`, `RO` bzw. `LDO` den Vorgang, und `CV.E1` signalisiert die Wertänderung; `QU.E1`/`QD.E1` werden nur bei einer Änderung des jeweiligen Status ausgelöst (On-Change-Triggering).

Technische Besonderheiten

✓ **Adapter-basiert**: Nahtlose Integration in AX-Systeme. ✓ **Ereignisgesteuert**: Keine zyklischen Aufrufe notwendig. ✓ **UDINT-Basis**: Unterstützt den vollen Wertebereich von vorzeichenlosen 32-Bit-Ganzzahlen. ✓ **On-Change-Triggering**: `QU.E1` und `QD.E1` feuern nur, wenn sich `QU` bzw. `QD` ändern; `PV.E1` bewertet beide Status neu.

Anwendungsszenarien

- **Stückzählung**: Zählen von Objekten in Fertigungslinien.
- **Betriebsstundenzähler**: Erfassen von Betriebszeiten (in Kombination mit Timern).
- **Positionsüberwachung**: Einfache Positionszähler in Handling-Systemen.

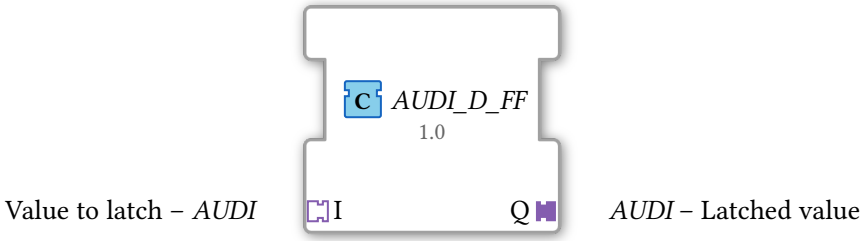
🛠️ Zugehörige Übungen

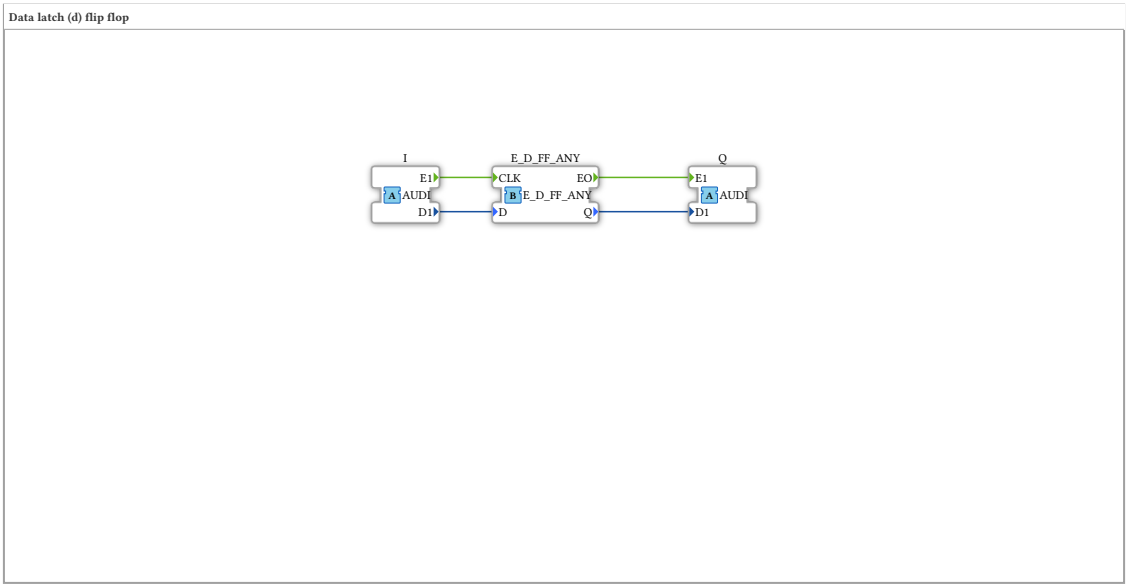
- Uebung_009_AX
- Uebung_083_AX

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de

AUDI_D_FF





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUDI_D_FF** realisiert ein taktgesteuertes D-Flip-Flop (Data Latch) auf Basis von IEC 61499-Adaptern. Er übernimmt bei einem Ereignis am Eingangsadapter den anliegenden Datenwert, speichert ihn und gibt ihn über den Ausgangsadapter aus. Der Baustein kapselt die gesamte Steuerlogik in einem kompakten, adapterbasierten Interface und erlaubt so eine saubere Trennung von Ereignis- und Datenkommunikation.

Schnittstellenstruktur

Ereignis-Eingänge
Keine – die Ereignissteuerung erfolgt ausschließlich über den Socket-Adapter.

Ereignis-Ausgänge
Keine – die Ereignisausgabe erfolgt ausschließlich über den Plug-Adapter.

Daten-Eingänge
Keine – die Datenaufnahme erfolgt ausschließlich über den Socket-Adapter.

Daten-Ausgänge
Keine – die Datenausgabe erfolgt ausschließlich über den Plug-Adapter.

Adapter

Bezeichnung	Typ	Richtung	Beschreibung
I	`adapter::types::unidirectional::AUDI`	Socket (Eingang)	Liefert das Taktereignis (E1) und den zu übernehmenden Datenwert (D1).
Q	`adapter::types::unidirectional::AUDI`	Plug (Ausgang)	Gibt das Ausgangsereignis (E1) und den gespeicherten Datenwert (D1) aus.

Beide Adapter sind vom gleichen unidirektionalen Typ **AUDI**, der ein Ereignis und einen Datenwert gemeinschaftlich transportiert. Der Socket **I** dient als Quelle, der Plug **Q** als Senke.

Funktionsweise
Der FB arbeitet nach der klassischen D-Flip-Flop-Funktion:

1. **Warten auf Takt**: Solange kein Ereignis am Eingang **I.E1** anliegt, behält der interne Speicher seinen letzten Wert bei.
2. **Übernahme bei Taktflanke**: Ein Ereignis an **I.E1** (als positive Flanke interpretiert) veranlasst den Baustein, den aktuellen Datenwert von **I.D1** zu übernehmen und im internen Speicher abzulegen.
3. **Ausgabe**: Nach der Übernahme wird ein Ereignis am Ausgang **Q.E1** ausgegeben und der gespeicherte Datenwert erscheint an **Q.D1**. Dieser Wert bleibt stabil, bis das nächste Taktereignis eintrifft.

Die interne Implementierung verwendet den IEC-61499-Standardbaustein `E_D_FF_ANY`, der eine typunabhängige D-Flip-Flop-Funktion bereitstellt. Die Adapter **I** und **Q** sind direkt mit dessen Ereignis- und Daten-Ports verschaltet.

Technische Besonderheiten

- **Adapter-basierte Schnittstelle**: Der FB nutzt keine einzelnen Ereignis- oder Daten-Ein-/Ausgänge, sondern kommuniziert ausschließlich über unidirektionale Adapter. Dies vereinfacht die Verbindung in komplexen Netzwerken und unterstützt eine modulare Typprüfung.
- **Typunabhängigkeit**: Der interne Baustein `E_D_FF_ANY` arbeitet mit beliebigen Datentypen (ANY). Der eigentliche Datentyp wird durch die Adapterdefinition des Typs `AUDI` festgelegt.
- **Unidirektionale Kommunikation**: Der Adaptertyp `adapter::types::unidirectional::AUDI` überträgt Ereignis und Daten nur in eine Richtung – vom Socket zum Plug. Eine Rückkopplung ist nicht vorgesehen.
- **Kein zusätzlicher Initialisierungszustand**: Der FB startet ohne definierten Datenwert; der interne Zustand entspricht dem Initialzustand des verwendeten `E_D_FF_ANY`-Bausteins.

Zustandsübersicht
Da es sich um einen rein kombinatorischen, ereignisgesteuerten Speicher handelt, kann das Verhalten durch zwei stabile Zustände beschrieben werden:

- **Zustand A (Warten)**: Kein Taktereignis empfangen – der Ausgang **Q.D1** hält den zuletzt übernommenen Wert.
- **Zustand B (Übernahme)**: Ein Taktereignis an **I.E1** löst die Übernahme von **I.D1** aus. Nach der Verarbeitung kehrt der FB sofort wieder in den Wartezustand zurück (kein mehrstufiger Automat).

Eine grafische Darstellung als Zustandsautomat entfällt, da die Logik auf den internen `E_D_FF_ANY` delegiert wird, dessen Zustandsdiagramm in der IEC 61499-Spezifikation definiert ist.

Anwendungsszenarien

- **Datenpuffer**: Übernehmen eines Sensorsignals zu einem definierten Messzeitpunkt und Weiterleiten an eine Steuerung.
- **Synchronisation**: Gleichzeitiges Auslesen mehrerer Datenquellen durch ein gemeinsames Taktereignis.
- **Realisation von Registern**: Mehrere `AUDI_D_FF`-Bausteine können parallel geschaltet werden, um ein n-Bit-Register aufzubauen.
- **Schnittstelle zu ereignisgesteuerten Kommunikationsprotokollen**: Der Adapter eignet sich besonders für Systeme, die standardisierte Daten-Ereignis-Paare verwenden (z. B. OPC UA oder MQTT-Anbindungen).

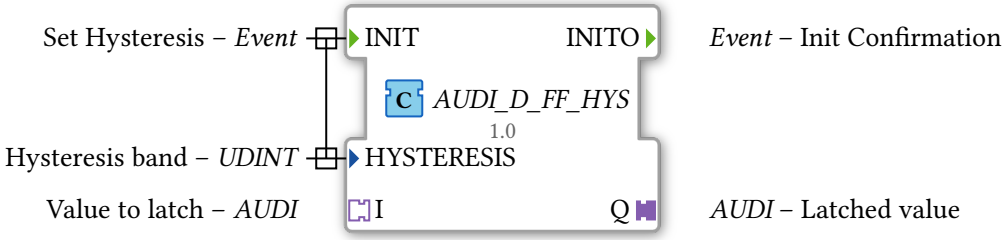
Vergleich mit ähnlichen Bausteinen

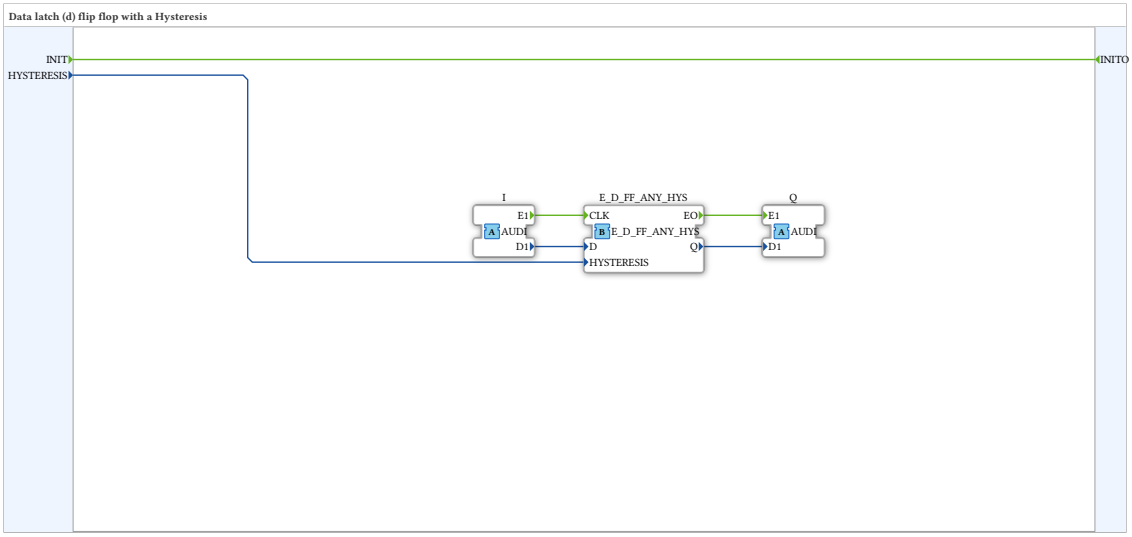
- **`E_D_FF_ANY`** (IEC 61499-Standardbausstein): Besitzt separate Ereignis- und Daten-Ein-/Ausgänge, kein Adapterinterface. Der `AUDI_D_FF` kapselt diese Schnittstellen in einem Adapter und vereinfacht so die Verwendung in adapterbasierten Netzwerken.
- **SR-Flip-Flop (Set-Reset)**: Hat zwei Dateneingänge (Set/Reset) und erlaubt keinen reinen Taktbetrieb. Der `AUDI_D_FF` ist auf den einfachen D-Flip-Flop-Anwendungsfall zugeschnitten.
- **AUDI-Adapter-basierte Bausteine (z. B. `AUDI_D_FF_2`)** : Andere Implementierungen könnten zusätzliche Logik wie asynchronen Reset enthalten. Der vorliegende FB realiert das minimale D-Flip-Flop ohne zusätzliche Steuerfunktionen.

Fazit

Der **AUDI_D_FF** stellt eine sauber gekapselte, adapterbasierte Realisierung eines D-Flip-Flops dar. Durch den Einsatz des unidirektionalen `AUDI`-Adaptertyps wird die Schnittstelle auf das Wesentliche reduziert – ein Ereignis und ein Datenwert pro Seite. Die interne Nutzung des bewährten `E_D_FF_ANY`-Bausteins gewährleistet eine robuste, standardkonforme Funktion. Der FB eignet sich besonders für modulare Architekturen, die auf Adapter-Konnektivität setzen, und erleichtert die Wiederverwendung in verschiedenen Steuerungsprojekten.

AUDI_D_FF_HYS





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AUDI_D_FF_HYS` realisiert ein datengetriggertes Flip-Flop (D-Latch) mit einer Hysteresefunktion. Er dient dazu, einen eingehenden Datenwert bei einem Ereignis zu übernehmen, wobei durch die Hysterese ein Schaltverhalten mit definiertem Totband ermöglicht wird. Der Baustein kapselt die Logik des internen Bausteins `E_D_FF_ANY_HYS` und stellt die Schnittstellen über zwei unidirektionale Adapter vom Typ `AUDI` bereit.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar	Mit Variable
`INIT`	EInit	Set Hysteresis	`HYSTERESIS`

Ereignis-Ausgänge

Ereignis	Typ	Kommentar
`INITO`	EInit	Init Confirmation

Daten-Eingänge

Variable	Typ	Kommentar
`HYSTERESIS`	UDINT	Hysteresis band

Daten-Ausgänge

Keine direkten Datenausgänge – die Ausgabedaten werden über den Adapter `Q` bereitgestellt.

Adapter

Rolle	Name	Typ (Adapter)	Kommentar
Plug	`Q`	`adapter::types::unidirectional::AUDI`	Latched value
Socket	`I`	`adapter::types::unidirectional::AUDI`	Value to latch

Der Adapter `AUDI` enthält jeweils einen Ereignis- und einen Datenkanal:

- `I.E1` – Ereignis, das den Latch-Vorgang auslöst
- `I.D1` – Datenwert, der gelatcht werden soll

- `Q.E1` – Bestätigungsereignis nach erfolgreichem Latch
- `Q.D1` – der gelatchte (und hysteresebeeinflusste) Datenwert

Funktionsweise

Der Baustein arbeitet als Wrapper um den internen Baustein `E_D_FF_ANY_HYS`. Beim Initialisierungsereignis `INIT` wird der Hysteresewert (an `HYSTERESIS`) übernommen; das `INITO`-Ereignis bestätigt die Initialisierung.
Während des Betriebs wird bei jedem Ereignis auf dem Adaptereingang `I.E1` der aktuelle Datenwert von `I.D1` übernommen und mit der eingestellten Hysterese verarbeitet. Das Ergebnis wird über den Adapterausgang `Q.D1` ausgegeben, begleitet von einem Ereignis `Q.E1`.
Die Hysterese verhindert ein ständiges Umschalten bei geringen Signaländerungen: Der Ausgang wechselt seinen Zustand erst, wenn der Eingangswert das Hystereseband verlässt.

Technische Besonderheiten

- Der Baustein nutzt die generische Implementierung `E_D_FF_ANY_HYS` (aus der Bibliothek `logiBUS::signalprocessing::hysteresis`).
- Die Hysterese wird als `UDINT` (unsigned double integer) angegeben – der Wertebereich ist daher 0 … 4 294 967 295.
- Die gesamte Ein-/Ausgabe erfolgt ausschließlich über unidirektionale Adapter (`AUDI`), was eine klare Trennung und Wiederverwendbarkeit in komplexen Netzwerken fördert.
- Das INIT-Ereignis wird direkt an `INITO` durchgeschleift, ohne Zwischenverarbeitung – die Hysterese-Einstellung erfolgt aber vor dem `INITO` (laut Datenverbindung).

Zustandsübersicht

Da der FB eine reine Funktionsnetzwerk-Verschaltung ist (kein eigenes ECC), ergibt sich der Zustand aus dem internen Baustein `E_D_FF_ANY_HYS`. Vereinfacht:

- **Rückgesetzt**: Ausgang `Q.D1` ist 0 oder einem definierten Initialwert.
- **Setzen**: Bei Überschreitung der oberen Hystereseschwelle wird der Ausgang auf 1 gesetzt.
- **Halten**: Bei Werten innerhalb des Hysteresebands bleibt der Ausgang unverändert.
- **Rücksetzen**: Bei Unterschreitung der unteren Hystereseschwelle wird der Ausgang zurückgesetzt.

Anwendungsszenarien

- **Entprellung von Sensorsignalen**: Verhindert Flattern bei mechanischen Schaltern oder Näherungssensoren.
- **Schwellwertschalter mit Rückkopplungsunterdrückung**: Z. B. Temperaturregelung mit einstellbarem Totband.
- **Puffern digitaler Eingänge**: Übernahme eines stabilen Werts bei definiertem Ereignis (z. B. Taktflanke).

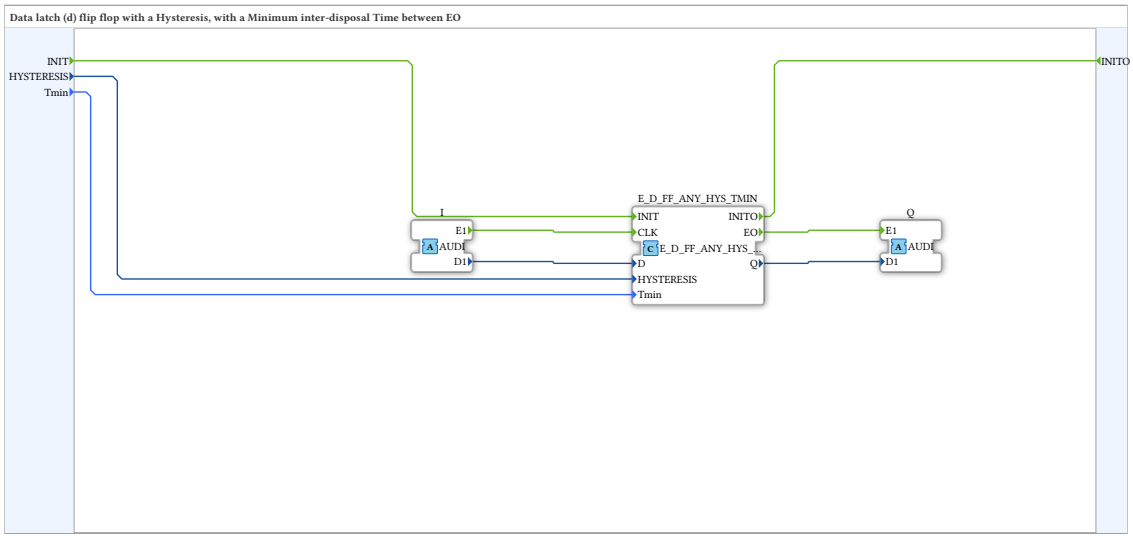
Vergleich mit ähnlichen Bausteinen

- **Standard-D-Flip-Flop (ohne Hysterese)**: Reagiert sofort auf jede Änderung – keine Unterdrückung von Rauschen oder Prellen.
- **RS-Flip-Flop mit Hysterese (z. B. `RS_FF_HYS`)**: Steuert über separate Set-/Reset-Ereignisse, während `AUDI_D_FF_HYS` einen einzigen Daten-/Ereigniseingang nutzt.
- **Schmitt-Trigger-Baustein**: Funktional ähnlich, aber meist ohne Latch-/Speicherverhalten – hier wird der Wert bei einem Ereignis gelatcht und bis zum nächsten Ereignis gehalten.

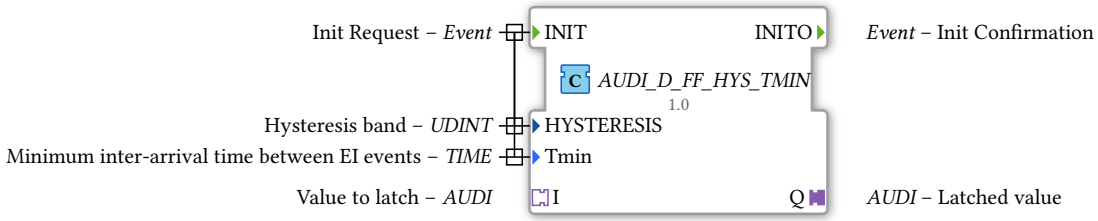
Fazit

`AUDI_D_FF_HYS` kombiniert die Eigenschaften eines datengetriggerten Flip-Flops mit einer einstellbaren Hysterese. Durch die Kapselung als Adapter-Baustein lässt er sich leicht in modulare Automatisierungsnetzwerke einbinden und bietet eine robuste Lösung für Signalverarbeitung mit Entprellung oder Schwellwertlogik.

AUDI_D_FF_HYS_TMIN



Kein Bild vorhanden



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUDI_D_FF_HYS_TMIN** realisiert ein taktflankengesteuertes Daten-Latch (D-Flipflop) mit einer Hysterese-Funktion und einer Mindestverweilzeit zwischen den Ausgangsereignissen (EO). Er dient der stabilen und entstörten Übernahme eines Datenwertes bei gleichzeitiger Unterdrückung von Rauschen und unerwünscht schnellen Schaltvorgängen. Der FB kapselt die Logik eines internen Bausteins `E_D_FF_ANY_HYS_TMIN` und stellt die Ein-/Ausgänge über standardisierte Adapter bereit.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar
INIT	EInit	Initialisierungsanfrage, setzt Hysterese und Mindestzeit

Ereignis-Ausgänge

Ereignis	Typ	Kommentar
INITO	EInit	Initialisierungsbestätigung

Daten-Eingänge

Variable	Typ	Kommentar
HYSTERESIS	UDINT	Hysterese-Band (Breite der Totzone zur Störunterdrückung)
Tmin	TIME	Mindestzeit zwischen zwei aufeinanderfolgenden Ausgangsereignissen

Daten-Ausgänge

Keine direkten Datenausgänge – die Ausgangsdaten werden über den Adapter **Q** bereitgestellt.

Adapter

Bezeichnung	Typ	Richtung	Kommentar
I	`adapter::types::unidirectional::AUDI`	Eingang	Zu übernehmender Datenwert
Q	`adapter::types::unidirectional::AUDI`	Ausgang	Latched Wert

Funktionsweise

1. **Initialisierung** – Über den Ereigniseingang `INIT` werden die Parameter `HYSTERESIS` und `Tmin` gesetzt. Nach erfolgreicher Initialisierung wird `INITO` ausgegeben.
2. **Datenlatch** – Jedes Ereignis am Adaptereingang `I.E1` wird als Takt (CLK) interpretiert. Der aktuelle Datenwert von `I.D1` wird übernommen.
3. **Hysterese** – Eine Änderung des Ausgangswertes erfolgt nur, wenn die Differenz zwischen neuem und altem Wert das Hysterese-Band (`HYSTERESIS`) überschreitet. Dadurch werden kleine Schwankungen (z. B. Rauschen) unterdrückt.
4. **Mindestzeit** – Nach einem Ausgangsereignis (`Q.E1`) wird für die Dauer `Tmin` kein weiteres Ausgangsereignis erzeugt, auch wenn am Eingang neue Takte anliegen. Dies verhindert eine zu hohe Rate der Ausgangsereignisse.

Der FB verwendet intern den Baustein `logiBUS::signalprocessing::hysteresis::E_D_FF_ANY_HYS_TMIN`, der die beschriebene Logik implementiert.

Technische Besonderheiten

- **Adapterbasierte Ein-/Ausgabe** – Der FB kommuniziert über standardisierte unidirektionale Adapter vom Typ `AUDI`. Dies ermöglicht eine typsichere und modulare Anbindung an andere Bausteine.
- **Parametrierbare Hysterese** – Die Hysterese-Breite kann zur Laufzeit über `HYSTERESIS` angepasst werden.
- **Mindestzeit zwischen Ereignissen** – Die minimale Zeit `Tmin` zwischen zwei Ausgangsereignissen schützt nachgeschaltete Komponenten vor Überlastung und definiert eine maximale Aktualisierungsrate.
- **Initialisierungszwang** – Vor dem ersten Betrieb muss der FB über `INIT` konfiguriert werden, andernfalls sind die internen Parameter undefiniert.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten im XML. Das Verhalten wird vollständig durch den inneren Baustein `E_D_FF_ANY_HYS_TMIN` bestimmt. Folgende Zustände lassen sich logisch ableiten:

- **Initial** – Nach Reset, wartet auf `INIT`.
- **Bereit** – Nach `INIT` und `INITO`, bereit zur Datenübernahme.
- **Warten auf Takt** – Erwartet ein Ereignis an `I.E1`.
- **Verarbeitung** – Daten werden ausgewertet, Hysterese geprüft.
- **Ausgabe gesperrt** – Nach einem Ausgangsereignis läuft die `Tmin`-Zeit, während der keine weiteren Ausgaben erfolgen.

Anwendungsszenarien

- **Sensorwert-Verdrahtung** – Stabilisierung analoger oder digitaler Sensorsignale mit Hysterese (z. B. Füllstand, Temperatur, Druck) und Begrenzung der Aktualisierungsrate.
- **Taktflankengesteuerte Datenübernahme** – Zuverlässiges Einlesen von Werten aus rauschender Umgebung, wenn eine Mindestzeit zwischen zwei Wertänderungen erforderlich ist.
- **Steuerung von Aktoren** – Schutz von Aktoren vor zu schnellem Schalten durch Begrenzung der Ausgangsereignisfrequenz.

Vergleich mit ähnlichen Bausteinen

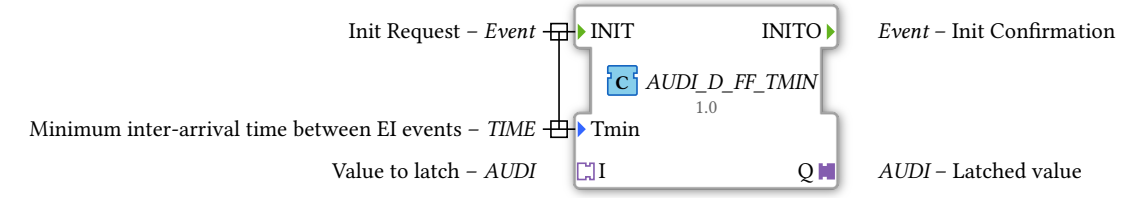
- **Standard D-Flipflop (z. B. `E_D_FF`)** – Keine Hysterese und keine Zeitbegrenzung; reagiert auf jede Taktflanke ohne Filterung.

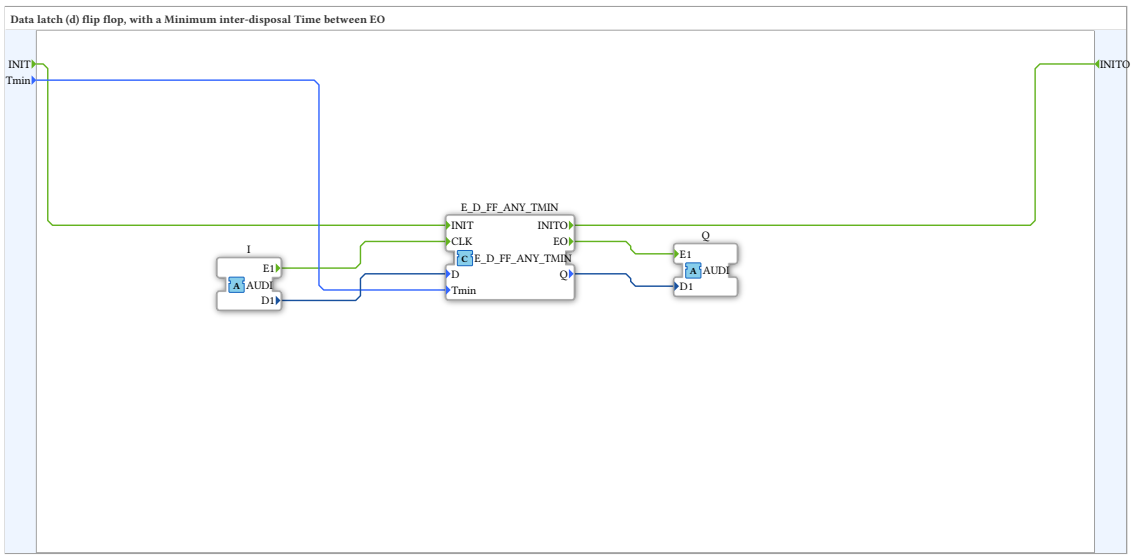
- **D-Flipflop mit Hysterese (z. B. `AUDI_D_FF_HYS`)** – Enthält Hysterese, aber keine Mindestzeit zwischen Ausgangsereignissen.
- **AUDI_D_FF_HYS_TMIN** – Kombiniert beide Eigenschaften: Hysterese zur Rauschunterdrückung und `Tmin` zur Begrenzung der Ereignisrate.

Fazit

Der FB `AUDI_D_FF_HYS_TMIN` bietet eine robuste, parametrierbare Lösung für die taktgesteuerte Datenübernahme in industriellen Automatisierungsumgebungen. Durch die Kombination von Hysterese und minimaler Inter-Ereignis-Zeit eignet er sich besonders für Anwendungen mit gestörten Signalen oder begrenzter Ausgangsbandbreite. Die Adapter-basierte Schnittstelle erleichtert die Integration in vorhandene Systeme und fördert die Wiederverwendbarkeit.

AUDI_D_FF_TMIN





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUDI_D_FF_TMIN** realisiert ein D-Flipflop (Data Latch) mit einer minimalen Sperrzeit zwischen aufeinanderfolgenden Ausgangsereignissen. Er dient dazu, einen über einen Adapter eingehenden Wert zu übernehmen und über einen zweiten Adapter auszugeben, wobei die Weitergabe des Ausgangsereignisses zeitlich gesteuert wird.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung
INIT	EInit	Initialisierung des Bausteins; setzt die minimale Zeit zwischen zwei EO-Ereignissen.

Ereignis-Ausgänge

Name	Typ	Beschreibung
INITO	EInit	Bestätigung der abgeschlossenen Initialisierung.

Daten-Eingänge

Name	Typ	Beschreibung
Tmin	TIME	Minimale Zeitdauer, die zwischen zwei aufeinanderfolgenden EO-Ereignissen mindestens vergehen muss.

Daten-Ausgänge

Keine direkten Datenausgänge; der gelatchte Wert wird über den Adapter **Q** bereitgestellt.

Adapter

Name	Typ (unidirektional)	Rolle
I	AUDI	Eingangsadapter: liefert den zu latchnden Wert (über D1) und das Latch-Ereignis (über E1).
Q	AUDI	Ausgangsadapter: gibt den gelatchten Wert (über D1) sowie ein Ausgangsereignis (über E1) aus.

Funktionsweise

Der Baustein enthält intern eine Instanz des FBs `E_D_FF_ANY_TMIN`, der die Kernlogik eines D-Flipflops mit zeitlicher Begrenzung realisiert.

1. **Initialisierung**
   Ein INIT-Ereignis setzt die minimale Verzögerungszeit `Tmin` (über die With-Assoziation) und wird an die interne Instanz weitergeleitet. Nach erfolgreicher Initialisierung wird INITO ausgegeben.
2. **Latch-Vorgang**
   Ein Ereignis am Eingang `I.E1` (des Adapters) wird als Takt (CLK) an das interne Flipflop übergeben. Gleichzeitig wird der Wert von `I.D1` (D-Eingang) übernommen. Der gelatchte Wert erscheint am Ausgang `Q.D1`, das entsprechende Ausgangsereignis am Ereignisausgang des Adapters (`Q.E1`).
3. **Minimale Wartezeit**
   Der interne FB erzwingt eine zeitliche Mindestdistanz zwischen aufeinanderfolgenden `Q.E1`-Ereignissen. Überschreitet die an `Tmin` definierte Zeit, wird das nächste Ausgangsereignis erst nach Ablauf dieser Zeitspanne ausgelöst. Auf diese Weise lassen sich Überlastungen auf der Ausgangsseite vermeiden.

Technische Besonderheiten

- **Adapterbasierte Kommunikation**
  Der Baustein verwendet zwei unidirektionale AUDI-Adapter für Ein- und Ausgang. Dadurch wird eine lose Kopplung zwischen Datenfluss und Ereignissteuerung erreicht.
- **Intervallsteuerung**
  Die minimale Zeit zwischen zwei EO-Ereignissen wird über den Parameter `Tmin` konfiguriert. Der interne FB `E_D_FF_ANY_TMIN` stellt sicher, dass diese Bedingung eingehalten wird, indem er eingehende Taktereignisse bei Bedarf verzögert.
- **Keine expliziten Datenausgänge**
  Der gelatchte Wert wird ausschließlich über den Adapter `Q` zur Verfügung gestellt. Dies vereinfacht die Verbindung zu nachfolgenden Bausteinen mit gleichem Adaptertyp.

Zustandsübersicht

Der FB besitzt keine nach außen sichtbaren Zustände, da die Zustandslogik im inneren FB gekapselt ist. Die interne Logik durchläuft mindestens die folgenden internen Zustände:

- **IDLE**: Warten auf INIT oder auf ein Taktereignis (I.E1).
- **LATCH**: Wert wird gespeichert; Wartezeit prüfen.
- **OUTPUT**: Ausgabe von Q.E1 und Q.D1, wenn die Mindestzeit seit dem letzten Ausgangsereignis abgelaufen ist.

Ein INIT-Ereignis setzt alle Zustände zurück.

Anwendungsszenarien

- **Datenpuffer mit Zeitsteuerung**
  Übernahme von Messwerten oder Steuersignalen in festen, zeitlichen Abständen, z. B. zur Entkopplung eines schnellen Sensors von einem langsamen Aktor.
- **Ereignisfilter**
  Unterdrückung von Ereignissen, die zu schnell aufeinander folgen (Debouncing auf Anwendungsebene).
- **Synchronisation**
  Verwendung als Basisbaustein für die Realisierung von Verzögerungsgliedern oder Taktsperren in IEC-61499-Anwendungen.

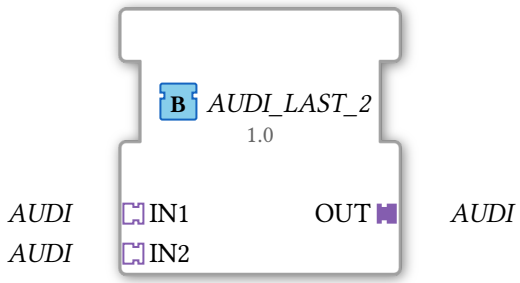
Vergleich mit ähnlichen Bausteinen

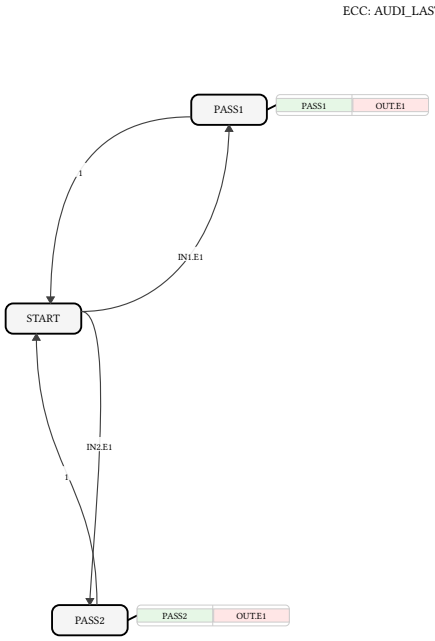
Baustein	Besonderheit
AUDI_D_FF	Reines D-Flipflop ohne zeitliche Begrenzung zwischen EO-Ereignissen.
E_D_FF_ANY	Standard-D-Flipflop mit universellen Datentypen, jedoch ohne minimale Wartezeit.
AUDI_D_FF_TMIN	Wie AUDI_D_FF, aber mit zusätzlicher `Tmin`-Parameter zur Steuerung der Mindestpause zwischen Ausgangsereignissen.

Fazit

Der **AUDI_D_FF_TMIN** ist ein spezialisierter D-Flipflop-Baustein, der die Ausgabe von Ereignissen und Werten zeitlich begrenzt. Durch die Adapter-Schnittstelle fügt er sich nahtlos in unidirektionale Kommunikationsstrukturen ein. Die Kombination aus Latch-Funktion und einstellbarer Sperrzeit macht ihn zu einem wertvollen Werkzeug für die ereignisgesteuerte Echtzeitverarbeitung in Automatisierungssystemen.

AUDI_LAST_2





Einleitung

Der AUDI_LAST_2 Funktionsblock führt zwei gleichartige UDINT-Adapter-Signale (**IN1**, **IN2**) auf einen gemeinsamen Ausgang (**OUT**) zusammen, nach dem Prinzip **Last Writer Wins**: welcher der beiden Sockets zuletzt ein Ereignis geschrieben hat, gewinnt, und sein Datenwert wird sofort und unverändert an OUT durchgereicht. Es handelt sich ausdrücklich nicht um ein naives Zusammenführen (z. B. Mittelwert oder ODER-Verknüpfung), sondern um eine reine Arbitrierung nach zeitlicher Reihenfolge.

Schnittstellenstruktur

Adapter

Eingangsadapter (Sockets):

- **IN1**: UDINT-Adapter (unidirectional) - erste Signalquelle
- **IN2**: UDINT-Adapter (unidirectional) - zweite Signalquelle

Ausgangsadapter (Plug):

- **OUT**: UDINT-Adapter (unidirectional) - zusammengeführtes Signal

Funktionsweise

Der Baustein ist als einfacher Basic-FB mit einer 3-Zustands-ECC umgesetzt (START, PASS1, PASS2). Trifft an IN1 ein Ereignis ein, wechselt die ECC nach PASS1: der aktuelle Datenwert von IN1 wird nach OUT kopiert und das Adapter-Ereignis an OUT gesendet. Trifft an IN2 ein Ereignis ein, geschieht dasselbe symmetrisch über PASS2. Nach jeder Durchreichung kehrt die ECC sofort wieder nach START zurück und ist bereit für das nächste Ereignis von IN1 oder IN2.

Technische Besonderheiten

- Last-Writer-Wins-Semantik statt naiver Datenverknüpfung
- Einfacher Basic-FB (ECC mit 3 Zuständen), kein zusätzlicher Zwischenspeicher nötig
- Keine Signalverzögerung: der Wert wird im selben Zyklus durchgereicht, in dem das auslösende Ereignis eintrifft
- Beide Eingänge sind gleichberechtigt - es gibt keine feste Priorität außer der zeitlichen Reihenfolge

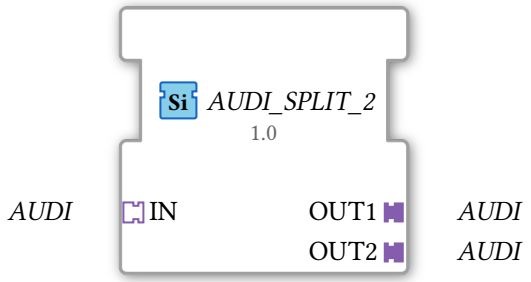
Anwendungsszenarien

- Zusammenführen zweier gleichwertiger UDINT-Signalquellen zu einem einzigen nachgeschalteten Adapterpfad (z. B. zwei redundante Sensoren, oder ein manueller und ein automatischer Sollwertgeber)
- Arbitrierung zwischen zwei Bedienstellen, die denselben UDINT-Wert schreiben dürfen
- Vereinfachung von Netzwerken, die sonst ein `F_SEL`/`E_RS`-Konstrukt inline verdrahten müssten

⚖️ **Vergleich mit ähnlichen Bausteinen**

AUDI_LAST_2 ist die Umkehrung von `AUDI_SPLIT_2`: AUDI_SPLIT_2 verteilt 1 Eingang auf 2 Ausgänge, AUDI_LAST_2 führt 2 Eingänge auf 1 Ausgang zusammen. Für Mischbetrieb zwischen einem Daten- und einem reinen Ereignisadapter (z. B. AX + AE) ist stattdessen `AX_AE_MERGE` gedacht, das nur das Ereignis mischt, ohne den Datenwert zu verändern.

AUDI_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUDI_SPLIT_2** ist ein generischer Baustein zur Signalverteilung. Er empfängt über einen einzelnen **AUDI**-Adapter (unidirektional) ein Signal und leitet es an zwei identische **AUDI**-Ausgangsadapter weiter. Der FB wird als **generic FB** deklariert, d.h. der konkrete Signaltyp kann zur Projektierungszeit über das Attribut `GenericClassName` festgelegt werden. Entwickelt wurde er von der HR Agrartechnik GmbH (Version 1.0, 2025-01-24).

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	`IN`	`adapter::types::unidirectional::AUDI`	Eingehender unidirektionaler AUDI-Adapter (Signalquelle).
Plug	`OUT1`	`adapter::types::unidirectional::AUDI`	Erster Ausgang – leitet das Eingangssignal weiter.
Plug	`OUT2`	`adapter::types::unidirectional::AUDI`	Zweiter Ausgang – leitet das Eingangssignal weiter.

Funktionsweise

Der Baustein arbeitet als **passiver Splitter** ohne eigene Logik oder Zustände. Das über den Socket `IN` ankommende Signal wird auf beide Plugs `OUT1` und `OUT2` durchgeschliffen. Die Weiterleitung erfolgt unverändert und ohne Pufferung – jede Änderung am Eingang wird sofort an beiden Ausgängen sichtbar.

Da es sich um einen **unidirektionalen** Adapter handelt, fließen Daten nur vom Socket zu den Plugs. Eine Rückwirkung von den Ausgängen auf den Eingang ist nicht vorgesehen.

Technische Besonderheiten

- **Generischer Baustein**: Das Attribut `eclipse4diac::core::GenericClassName` erlaubt die Anpassung an unterschiedliche AUDI-Typen (z.B. `AUDI_Int`, `AUDI_Bool`). Der tatsächliche Typ wird erst bei der Instanziierung festgelegt.
- **Kein internes ECC**: Der FB besitzt keinen Ausführungszustandsautomaten, da er keine Ereignisse verarbeitet. Die Datenweitergabe erfolgt rein strukturell.
- **Unidirektionale Schnittstelle**: Sowohl Socket als auch Plugs sind vom Typ `adapter::types::unidirectional::AUDI`, was bedeutet, dass die Datenflussrichtung fest vorgegeben ist.
- **Urheberrecht**: Der Baustein steht unter der Eclipse Public License 2.0.

Zustandsübersicht

Der FB hat keine eigenen Zustandsdiagramme, da er keine ereignisgesteuerten Abläufe beinhaltet. Die Funktion beschränkt sich auf eine reine Leitungsverzweigung.

Anwendungsszenarien

- **Signalverteilung**: Ein zentraler Sensor oder eine Steuerung (z.B. ein PID-Regler) sendet einen Wert, der parallel an zwei nachgelagerte Aktoren oder Überwachungseinheiten weitergegeben werden soll.
- **Redundante Überwachung**: Ein Messwert wird gleichzeitig an zwei unabhängige Auswerteblöcke gesendet, um Vergleichs- oder Sicherheitsfunktionen zu realisieren.
- **Generische Adapter-Architektur**: Besonders nützlich in Systemen, bei denen der konkrete AUDI-Typ erst zur Laufzeit festgelegt wird (z.B. in einer konfigurierbaren Geräteplattform).

Vergleich mit ähnlichen Bausteinen

- **Nicht-generische Splitter** (z.B. `AUDI_SPLIT_2` mit festem Typ): Lassen keine Typanpassung zu; der hier vorgestellte FB ist flexibler.
- **Andere Splitter mit mehr Ausgängen** (z.B. `AUDI_SPLIT_3`): Erweitern die Verzweigungszahl, folgen aber dem gleichen Prinzip.
- **Ereignisbasierte Splitter** (z.B. `E_SPLIT`): Benötigen Ereignis- und Daten-Eingänge/-Ausgänge und führen eine synchronisierte Verteilung durch – im Gegensatz zum vorliegenden asynchronen Daten-Adapter-Split.
- **`AUDI_SPLIT_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

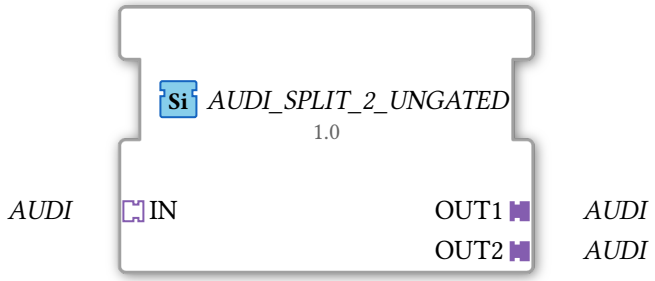
Der **AUDI_SPLIT_2** ist ein kompakter, generischer Adapter-Splitter für die 4diac-IDE. Er erfüllt die einfache Aufgabe der Signalverzweigung auf zwei Ausgänge, ohne zusätzliche Latenz oder Logik zu verursachen. Durch die generische Auslegung eignet er sich für unterschiedlichste AUDI-Datentypen und ermöglicht eine flexible Wiederverwendung in modularen Automatisierungsprojekten. Seine Einfachheit und Typsicherheit machen ihn zu einem soliden Grundbaustein für verteilte Steuerungssysteme.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUDI_SPLIT_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUDI_SPLIT_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUDI_SPLIT_2_UNGATED** ist ein generischer Baustein zur Signalverteilung. Er empfängt über einen einzelnen **AUDI**-Adapter (unidirektional) ein Signal und leitet es an zwei identische **AUDI**-Ausgangsadapter weiter. Der FB wird als **generic FB** deklariert, d.h. der konkrete Signaltyp kann zur Projektierungszeit über das Attribut `GenericClassName` festgelegt werden. Entwickelt wurde er von der HR Agrartechnik GmbH (Version 1.0, 2025-01-24).

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	`IN`	`adapter::types::unidirectional::AUDI`	Eingehender unidirektionaler AUDI-Adapter (Signalquelle).
Plug	`OUT1`	`adapter::types::unidirectional::AUDI`	Erster Ausgang – leitet das Eingangssignal weiter.
Plug	`OUT2`	`adapter::types::unidirectional::AUDI`	Zweiter Ausgang – leitet das Eingangssignal weiter.

Funktionsweise

Der Baustein arbeitet als **passiver Splitter** ohne eigene Logik oder Zustände. Das über den Socket `IN` ankommende Signal wird auf beide Plugs `OUT1` und `OUT2` durchgeschliffen. Die Weiterleitung erfolgt unverändert und ohne Pufferung – jede Änderung am Eingang wird sofort an beiden Ausgängen sichtbar.

Da es sich um einen **unidirektionalen** Adapter handelt, fließen Daten nur vom Socket zu den Plugs. Eine Rückwirkung von den Ausgängen auf den Eingang ist nicht vorgesehen.

Technische Besonderheiten

- **Generischer Baustein**: Das Attribut `eclipse4diac::core::GenericClassName` erlaubt die Anpassung an unterschiedliche AUDI-Typen (z.B. `AUDI_Int`, `AUDI_Bool`). Der tatsächliche Typ wird erst bei der Instanziierung festgelegt.
- **Kein internes ECC**: Der FB besitzt keinen Ausführungszustandsautomaten, da er keine Ereignisse verarbeitet. Die Datenweitergabe erfolgt rein strukturell.
- **Unidirektionale Schnittstelle**: Sowohl Socket als auch Plugs sind vom Typ `adapter::types::unidirectional::AUDI`, was bedeutet, dass die Datenflussrichtung fest vorgegeben ist.
- **Urheberrecht**: Der Baustein steht unter der Eclipse Public License 2.0.

Zustandsübersicht

Der FB hat keine eigenen Zustandsdiagramme, da er keine ereignisgesteuerten Abläufe beinhaltet. Die Funktion beschränkt sich auf eine reine Leitungsverzweigung.

Anwendungsszenarien

- **Signalverteilung**: Ein zentraler Sensor oder eine Steuerung (z.B. ein PID-Regler) sendet einen Wert, der parallel an zwei nachgelagerte Aktoren oder Überwachungseinheiten weitergegeben werden soll.
- **Redundante Überwachung**: Ein Messwert wird gleichzeitig an zwei unabhängige Auswerteblöcke gesendet, um Vergleichs- oder Sicherheitsfunktionen zu realisieren.
- **Generische Adapter-Architektur**: Besonders nützlich in Systemen, bei denen der konkrete AUDI-Typ erst zur Laufzeit festgelegt wird (z.B. in einer konfigurierbaren Geräteplattform).

Vergleich mit ähnlichen Bausteinen

- **Nicht-generische Splitter** (z.B. `AUDI_SPLIT_2_UNGATED` mit festem Typ): Lassen keine Typanpassung zu; der hier vorgestellte FB ist flexibler.
- **Andere Splitter mit mehr Ausgängen** (z.B. `AUDI_SPLIT_3`): Erweitern die Verzweigungszahl, folgen aber dem gleichen Prinzip.
- **Ereignisbasierte Splitter** (z.B. `E_SPLIT`): Benötigen Ereignis- und Daten-Eingänge/-Ausgänge und führen eine synchronisierte Verteilung durch – im Gegensatz zum vorliegenden asynchronen Daten-Adapter-Split.
- **`AUDI_SPLIT_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

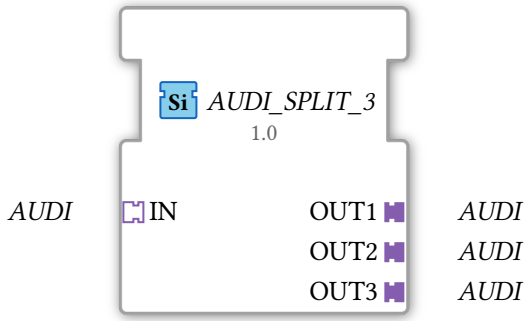
Fazit

Der **AUDI_SPLIT_2_UNGATED** ist ein kompakter, generischer Adapter-Splitter für die 4diac-IDE. Er erfüllt die einfache Aufgabe der Signalverzweigung auf zwei Ausgänge, ohne zusätzliche Latenz oder Logik zu verursachen. Durch die generische Auslegung eignet er sich für unterschiedlichste AUDI-Datentypen und ermöglicht eine flexible Wiederverwendung in modularen Automatisierungsprojekten. Seine Einfachheit und Typsicherheit machen ihn zu einem soliden Grundbaustein für verteilte Steuerungssysteme.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUDI_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUDI_SPLIT_3** dient der Vervielfachung eines eingehenden **AUDI**-Adapter-Signals auf drei separate Ausgänge. Er ist als generischer Baustein (Generic FB) konzipiert und eignet sich besonders für Architekturen, bei denen ein Signal (z. B. ein Steuer- oder Datenstrom) an mehrere nachgelagerte Komponenten verteilt werden muss.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine

Ereignis-Ausgänge

- Keine

Daten-Eingänge

- Keine

Daten-Ausgänge

- Keine

Adapter

Typ	Name	Beschreibung
Socket (adapter::types::unidirectional::AUDI)	**IN**	Eingangsadapter zum Empfang des AUDI-Signals
Plug (adapter::types::unidirectional::AUDI)	**OUT1**	Erster Ausgangsadapter
Plug (adapter::types::unidirectional::AUDI)	**OUT2**	Zweiter Ausgangsadapter
Plug (adapter::types::unidirectional::AUDI)	**OUT3**	Dritter Ausgangsadapter

Funktionsweise

Der Baustein leitet das am **IN**-Socket anliegende AUDI-Signal unverändert an alle drei **OUT**-Plugs weiter. Es findet keine Umwandlung, Filterung oder logische Verarbeitung statt – die Verteilung erfolgt transparent. Durch die Nutzung von Adaptern des Typs `unidirectional::AUDI` wird sichergestellt, dass die Signalrichtung (Eingang → Ausgänge) eindeutig definiert ist.

Technische Besonderheiten

- **Generische Realisierung**: Der Baustein ist als generischer FB (`eclipse4diac::core::GenericClassName = 'GEN_AUDI_SPLIT'`) implementiert. Das ermöglicht eine einfache Wiederverwendung in unterschiedlichen Kontexten, ohne dass der konkrete AUDI-Typ bereits zur Entwicklungszeit vollständig festgelegt sein muss.
- **Typ-Hash**: Der Typ-Hash kann bei Bedarf zur Laufzeit gesetzt werden, um eine statische Typprüfung zu ermöglichen.

- **Reine Adapter-Schnittstelle**: Der FB besitzt weder Ereignis- noch Datenein-/ausgänge im klassischen Sinne, sondern kommuniziert ausschließlich über Adapter.

Zustandsübersicht

Da der FB über keine Ereignis-Eingänge verfügt, besitzt er keinen expliziten Zustandsautomaten. Die Signalweitergabe erfolgt kontinuierlich und asynchron, sobald der IN-Adapter verbunden ist.

Anwendungsszenarien

- **Signalvervielfachung**: Verteilung eines gemeinsamen Steuer- oder Messsignals an verschiedene Funktionsblöcke (z. B. Parallelschaltung mehrerer Aktoren oder Sensoren).
- **Test- und Simulationsumgebungen**: Einschleusen eines simulierten AUDI-Signals in mehrere unabhängige Analyse- oder Überwachungspfade.
- **Architektur-Kopplung**: Einsatz in modularen Systemen, bei denen ein Datenstrom an verschiedene Subsysteme weitergegeben werden muss, ohne das Originalsignal zu schwächen.

Vergleich mit ähnlichen Bausteinen

- **AUDI_MERGE** (Gegenstück): Ein kombinierendes Element, das mehrere AUDI-Eingänge auf einen Ausgang vereint – im Gegensatz zum split-Baustein.
- **AUDI_SPLIT_2**: Ein auf zwei Ausgänge reduzierter Splitter; der vorliegende Block erweitert dies auf drei Abzweigungen.
- **Generische Splitter**: In vielen Bibliotheken existieren Splitt-Varianten für andere Adaptertypen (z. B. `DATA_SPLIT`), die jedoch datentypspezifisch sind.
- **`AUDI_SPLIT_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

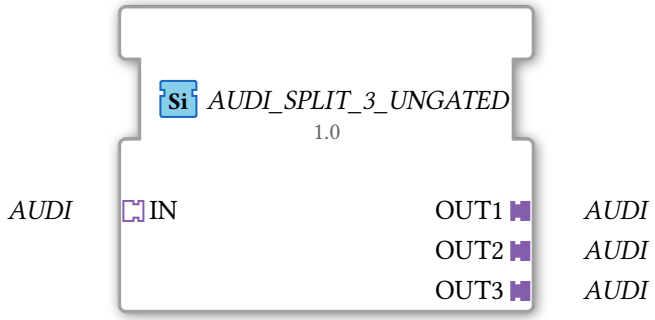
Der **AUDI_SPLIT_3** ist ein einfacher, aber nützlicher Funktionsblock zur Verteilung eines unidirektionalen AUDI-Adaptersignals auf drei Ausgänge. Durch seine generische Natur und die reine Adapterschnittstelle fügt er sich nahtlos in ereignisgesteuerte und datenstromorientierte IEC 61499-Anwendungen ein. Er bietet eine saubere, wartbare Lösung für architektonische Signalfächerungen und reduziert die Notwendigkeit manueller Signalweiterschaltung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 Gesamtwiderstand in Reihen- & Parallelschaltung auf ms-muc-docs.de

AUDI_SPLIT_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUDI_SPLIT_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUDI_SPLIT_3_UNGATED** dient der Vervielfachung eines eingehenden **AUDI**-Adapter-Signals auf drei separate Ausgänge. Er ist als generischer Baustein (Generic FB) konzipiert und eignet sich besonders für Architekturen, bei denen ein Signal (z. B. ein Steuer- oder Datenstrom) an mehrere nachgelagerte Komponenten verteilt werden muss.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine

Ereignis-Ausgänge

- Keine

Daten-Eingänge

- Keine

Daten-Ausgänge

- Keine

Adapter

Typ	Name	Beschreibung
Socket (adapter::types::unidirectional::AUDI)	**IN**	Eingangsadapter zum Empfang des AUDI-Signals
Plug (adapter::types::unidirectional::AUDI)	**OUT1**	Erster Ausgangsadapter
Plug (adapter::types::unidirectional::AUDI)	**OUT2**	Zweiter Ausgangsadapter
Plug (adapter::types::unidirectional::AUDI)	**OUT3**	Dritter Ausgangsadapter

Funktionsweise

Der Baustein leitet das am **IN**-Socket anliegende AUDI-Signal unverändert an alle drei **OUT**-Plugs weiter. Es findet keine Umwandlung, Filterung oder logische Verarbeitung statt – die Verteilung erfolgt transparent. Durch die Nutzung von Adaptern des Typs `unidirectional::AUDI` wird sichergestellt, dass die Signalrichtung (Eingang → Ausgänge) eindeutig definiert ist.

Technische Besonderheiten

- **Generische Realisierung**: Der Baustein ist als generischer FB (`eclipse4diac::core::GenericClassName = 'GEN_AUDI_SPLIT'`) implementiert. Das ermöglicht eine einfache Wiederverwendung in unterschiedlichen Kontexten, ohne dass der konkrete AUDI-Typ bereits zur Entwicklungszeit vollständig festgelegt sein muss.
- **Typ-Hash**: Der Typ-Hash kann bei Bedarf zur Laufzeit gesetzt werden, um eine statische Typprüfung zu ermöglichen.

- **Reine Adapter-Schnittstelle**: Der FB besitzt weder Ereignis- noch Datenein-/ausgänge im klassischen Sinne, sondern kommuniziert ausschließlich über Adapter.

Zustandsübersicht

Da der FB über keine Ereignis-Eingänge verfügt, besitzt er keinen expliziten Zustandsautomaten. Die Signalweitergabe erfolgt kontinuierlich und asynchron, sobald der IN-Adapter verbunden ist.

Anwendungsszenarien

- **Signalvervielfachung**: Verteilung eines gemeinsamen Steuer- oder Messsignals an verschiedene Funktionsblöcke (z. B. Parallelschaltung mehrerer Aktoren oder Sensoren).
- **Test- und Simulationsumgebungen**: Einschleusen eines simulierten AUDI-Signals in mehrere unabhängige Analyse- oder Überwachungspfade.
- **Architektur-Kopplung**: Einsatz in modularen Systemen, bei denen ein Datenstrom an verschiedene Subsysteme weitergegeben werden muss, ohne das Originalsignal zu schwächen.

Vergleich mit ähnlichen Bausteinen

- **AUDI_MERGE** (Gegenstück): Ein kombinierendes Element, das mehrere AUDI-Eingänge auf einen Ausgang vereint – im Gegensatz zum split-Baustein.
- **AUDI_SPLIT_2**: Ein auf zwei Ausgänge reduzierter Splitter; der vorliegende Block erweitert dies auf drei Abzweigungen.
- **Generische Splitter**: In vielen Bibliotheken existieren Splitt-Varianten für andere Adaptertypen (z. B. `DATA_SPLIT`), die jedoch datentypspezifisch sind.
- **`AUDI_SPLIT_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

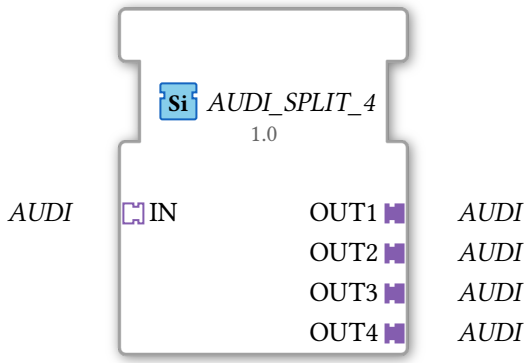
Fazit

Der **AUDI_SPLIT_3_UNGATED** ist ein einfacher, aber nützlicher Funktionsblock zur Verteilung eines unidirektionalen AUDI-Adaptersignals auf drei Ausgänge. Durch seine generische Natur und die reine Adapterschnittstelle fügt er sich nahtlos in ereignisgesteuerte und datenstromorientierte IEC 61499-Anwendungen ein. Er bietet eine saubere, wartbare Lösung für architektonische Signalfächerungen und reduziert die Notwendigkeit manueller Signalweiterschaltung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 Gesamtwiderstand in Reihen- & Parallelschaltung auf ms-muc-docs.de

AUDI_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUDI_SPLIT_4** dient dazu, ein eingehendes unidirektionales **AUDI**-Adapter-Signal auf vier identische Ausgänge zu verteilen. Er fungiert als einfacher Signalverteiler (Fan-out) für den generischen Adaptertyp **AUDI**. Der Baustein ist als generischer Funktionsblock (Generic FB) realisiert und kann über die Klasse `GEN_AUDI_SPLIT` an unterschiedliche Adapter-Typen angepasst werden.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	`IN`	`adapter::types::unidirectional::AUDI`	Eingangsadapter, der das zu verteilende Signal bereitstellt.
Plug	`OUT1`	`adapter::types::unidirectional::AUDI`	Erster Ausgang, identisch zum Eingangssignal.
Plug	`OUT2`	`adapter::types::unidirectional::AUDI`	Zweiter Ausgang, identisch zum Eingangssignal.
Plug	`OUT3`	`adapter::types::unidirectional::AUDI`	Dritter Ausgang, identisch zum Eingangssignal.
Plug	`OUT4`	`adapter::types::unidirectional::AUDI`	Vierter Ausgang, identisch zum Eingangssignal.

Funktionsweise

Der Baustein arbeitet **ereignislos** und **kontinuierlich**: Sobald der Socket `IN` mit einem AUDI-Adapter verbunden ist, werden die über diesen Adapter übertragenen Daten oder Signale unverändert an alle vier Ausgangsadapter (`OUT1` bis `OUT4`) weitergeleitet. Es findet keine Verarbeitung oder Umwandlung der Daten statt – der Baustein fungiert ausschließlich als passiver Verteiler (Splitter). Da keine Ereignisse oder Zustandsänderungen beteiligt sind, erfolgt die Verteilung implizit durch die Verbindungsstruktur.

Technische Besonderheiten

- **Generischer Funktionsblock**: Der Baustein ist als generischer Typ mit dem Klassennamen `'GEN_AUDI_SPLIT'` deklariert. Dadurch kann der konkrete Adaptertyp (z. B. ein spezialisierter AUDI-Subtyp) zur Entwurfszeit oder durch Code-Generierung angepasst werden.
- **Unidirektionale Adapter**: Alle verwendeten Adapter (`AUDI`) sind unidirektional, d. h. der Datenfluss erfolgt nur vom Socket zu den Plugs.
- **Keine Ereignissteuerung**: Der Baustein besitzt keine Ereigniseingänge oder -ausgänge. Die Signalverteilung geschieht datenflussgetrieben und benötigt keine explizite Auslösung.
- **Skalierbarkeit**: Der Baustein ist speziell für 4 Ausgänge ausgelegt. Für andere Anzahlen existieren entsprechende Varianten (z. B. `AUDI_SPLIT_2`, `AUDI_SPLIT_3`).

Zustandsübersicht

Der Baustein verfügt über **keine Zustandsmaschine (ECC)**, da er keine Ereignisse verarbeitet und keinen internen Zustand speichert. Die Funktion ist rein kombinatorisch: Die Ausgänge spiegeln kontinuierlich den Eingang wider.

Anwendungsszenarien

- **Signalverteilung in Steuerungssystemen**: Ein vom Sensor oder übergeordnetem System bereitgestelltes AUDI-Signal soll parallel an mehrere Verbraucher (Aktoren, Visualisierungen, Logikeinheiten) weitergegeben werden.
- **Test- und Simulationsumgebungen**: Ein einzelnes Testsignal wird auf mehrere parallele Prüfpfade aufgeteilt.
- **Prototypische Adapter-Kopplung**: Wenn mehrere nachgeschaltete Blöcke das gleiche Eingangssignal benötigen, ohne dass eine logische Vervielfachung nötig ist.

Vergleich mit ähnlichen Bausteinen

Baustein	Funktion	Besonderheit
`AUDI_SPLIT_2`	Verteilt ein AUDI-Signal auf zwei Ausgänge.	Entspricht dem Prinzip, jedoch mit 2 Ausgängen.
`AUDI_SPLIT_4`	Verteilt ein AUDI-Signal auf vier Ausgänge.	Vorliegender Baustein, einfacher 1:4-Splitter.
`AUDI_MERGE`	Fasst mehrere AUDI-Eingänge zu einem Ausgang zusammen.	Umkehrung des Splitters (1:1-Mapping, aber invers).
Generische Splitter	Können über Typ-Parameter auf andere Adapter angewendet werden.	Bieten mehr Flexibilität als fest verdrahtete Varianten.

- `AUDI_SPLIT_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

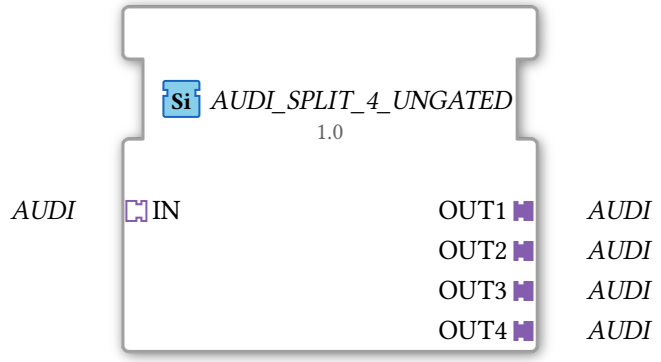
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der `AUDI_SPLIT_4` ist ein einfacher, aber nützlicher Funktionsblock zur Vervielfachung eines unidirektionalen AUDI-Adapter-Signals auf vier parallele Ausgänge. Seine generische Natur erlaubt die Anpassung an verschiedene Adaptertypen, und die ereignislose Arbeitsweise macht ihn besonders geeignet für reine Datenverteilungsaufgaben ohne zusätzliche Steuerlogik. Der Baustein trägt zur Strukturierung und Wiederverwendbarkeit von Signalen in komplexen Automatisierungslösungen bei.

AUDI_SPLIT_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUDI_SPLIT_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUDI_SPLIT_4_UNGATED** dient dazu, ein eingehendes unidirektionales **AUDI**-Adapter-Signal auf vier identische Ausgänge zu verteilen. Er fungiert als einfacher Signalverteiler (Fan-out) für den generischen Adaptertyp **AUDI**. Der Baustein ist als generischer Funktionsblock (Generic FB) realisiert und kann über die Klasse `GEN_AUDI_SPLIT` an unterschiedliche Adapter-Typen angepasst werden.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	`IN`	`adapter::types::unidirectional::AUDI`	Eingangsadapter, der das zu verteilende Signal bereitstellt.
Plug	`OUT1`	`adapter::types::unidirectional::AUDI`	Erster Ausgang, identisch zum Eingangssignal.
Plug	`OUT2`	`adapter::types::unidirectional::AUDI`	Zweiter Ausgang, identisch zum Eingangssignal.
Plug	`OUT3`	`adapter::types::unidirectional::AUDI`	Dritter Ausgang, identisch zum Eingangssignal.
Plug	`OUT4`	`adapter::types::unidirectional::AUDI`	Vierter Ausgang, identisch zum Eingangssignal.

Funktionsweise

Der Baustein arbeitet **ereignislos** und **kontinuierlich**: Sobald der Socket `IN` mit einem AUDI-Adapter verbunden ist, werden die über diesen Adapter übertragenen Daten oder Signale unverändert an alle vier Ausgangsadapter (`OUT1` bis `OUT4`) weitergeleitet. Es findet keine Verarbeitung oder Umwandlung der Daten statt – der Baustein fungiert ausschließlich als passiver Verteiler (Splitter). Da keine Ereignisse oder Zustandsänderungen beteiligt sind, erfolgt die Verteilung implizit durch die Verbindungsstruktur.

Technische Besonderheiten

- **Generischer Funktionsblock**: Der Baustein ist als generischer Typ mit dem Klassennamen `'GEN_AUDI_SPLIT'` deklariert. Dadurch kann der konkrete Adaptertyp (z. B. ein spezialisierter AUDI-Subtyp) zur Entwurfszeit oder durch Code-Generierung angepasst werden.
- **Unidirektionale Adapter**: Alle verwendeten Adapter (`AUDI`) sind unidirektional, d. h. der Datenfluss erfolgt nur vom Socket zu den Plugs.
- **Keine Ereignissteuerung**: Der Baustein besitzt keine Ereigniseingänge oder -ausgänge. Die Signalverteilung geschieht datenflussgetrieben und benötigt keine explizite Auslösung.
- **Skalierbarkeit**: Der Baustein ist speziell für 4 Ausgänge ausgelegt. Für andere Anzahlen existieren entsprechende Varianten (z. B. `AUDI_SPLIT_2`, `AUDI_SPLIT_3`).

Zustandsübersicht

Der Baustein verfügt über **keine Zustandsmaschine (ECC)**, da er keine Ereignisse verarbeitet und keinen internen Zustand speichert. Die Funktion ist rein kombinatorisch: Die Ausgänge spiegeln kontinuierlich den Eingang wider.

Anwendungsszenarien

- **Signalverteilung in Steuerungssystemen**: Ein vom Sensor oder übergeordnetem System bereitgestelltes AUDI-Signal soll parallel an mehrere Verbraucher (Aktoren, Visualisierungen, Logikeinheiten) weitergegeben werden.
- **Test- und Simulationsumgebungen**: Ein einzelnes Testsignal wird auf mehrere parallele Prüfpfade aufgeteilt.
- **Prototypische Adapter-Kopplung**: Wenn mehrere nachgeschaltete Blöcke das gleiche Eingangssignal benötigen, ohne dass eine logische Vervielfachung nötig ist.

Vergleich mit ähnlichen Bausteinen

Baustein	Funktion	Besonderheit
`AUDI_SPLIT_2`	Verteilt ein AUDI-Signal auf zwei Ausgänge.	Entspricht dem Prinzip, jedoch mit 2 Ausgängen.
`AUDI_SPLIT_4_UNGATED`	Verteilt ein AUDI-Signal auf vier Ausgänge.	Vorliegender Baustein, einfacher 1:4-Splitter.
`AUDI_MERGE`	Fasst mehrere AUDI-Eingänge zu einem Ausgang zusammen.	Umkehrung des Splitters (1:1-Mapping, aber invers).
Generische Splitter	Können über Typ-Parameter auf andere Adapter angewendet werden.	Bieten mehr Flexibilität als fest verdrahtete Varianten.

- `AUDI_SPLIT_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

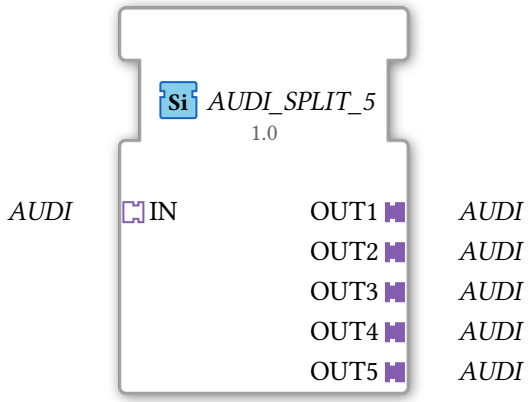
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AUDI_SPLIT_4_UNGATED` ist ein einfacher, aber nützlicher Funktionsblock zur Vervielfachung eines unidirektionalen AUDI-Adapter-Signals auf vier parallele Ausgänge. Seine generische Natur erlaubt die Anpassung an verschiedene Adaptertypen, und die ereignislose Arbeitsweise macht ihn besonders geeignet für reine Datenverteilungsaufgaben ohne zusätzliche Steuerlogik. Der Baustein trägt zur Strukturierung und Wiederverwendbarkeit von Signalen in komplexen Automatisierungslösungen bei.

AUDI_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUDI_SPLIT_5** dient dazu, ein einzelnes eingehendes AUDI-Signal auf fünf identische AUDI-Ausgänge zu verteilen. Er ist als generischer Adapterbaustein konzipiert und erlaubt eine flexible Weiterverarbeitung des Signals in mehreren Pfaden.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	`IN`	`adapter::types::unidirectional::AUDI`	Eingehendes AUDI-Signal
Plug	`OUT1`	`adapter::types::unidirectional::AUDI`	Erster Ausgang
Plug	`OUT2`	`adapter::types::unidirectional::AUDI`	Zweiter Ausgang
Plug	`OUT3`	`adapter::types::unidirectional::AUDI`	Dritter Ausgang
Plug	`OUT4`	`adapter::types::unidirectional::AUDI`	Vierter Ausgang
Plug	`OUT5`	`adapter::types::unidirectional::AUDI`	Fünfter Ausgang

Funktionsweise

Der Baustein leitet das am Socket `IN` anliegende AUDI-Signal unverändert und gleichzeitig an alle fünf Plugs `OUT1` bis `OUT5` weiter. Es erfolgt keine Signalverarbeitung, Wandlung oder Pufferung – die Verteilung erfolgt auf reiner Adapterebene, sobald eine Verbindung zu den Ausgängen besteht.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AUDI_SPLIT`) realisiert. Er kann über entsprechende Mechanismen der Laufzeitumgebung für verschiedene Datentypen instantiiert werden, solange diese dem unidirektionalen AUDI-Adapterprotokoll entsprechen.

- **Keine Zustandslogik**: Der Baustein enthält keine Ereignis- oder Datenverarbeitung und keinen internen Zustandsautomaten.
- **Einfache und performante Signalverteilung**: Aufgrund seiner reinen Adapterstruktur eignet er sich besonders für Szenarien, in denen ein Signal ohne Latenz vervielfältigt werden muss.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten. Das Verhalten ist deterministisch: Solange der Baustein in Betrieb ist, wird jedes an `IN` ankommende Signal sofort an alle Ausgänge weitergegeben.

Anwendungsszenarien

- **Verteilen von Steuersignalen** in einer industriellen Automatisierung, z. B. ein Start-/Stop-Befehl, der mehrere gleichartige Module erreichen soll.
- **Vervielfältigung eines Messwerts** für parallele Auswertungen oder Visualisierungen.
- **Aufbau von Redundanzpfaden**, indem dasselbe Signal mehrfach genutzt wird.

Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu Bausteinen wie `AUDI_MERGE_5` (zusammenführen mehrerer Signale) oder `AUDI_SPLIT_2` (nur zwei Ausgänge) bietet `AUDI_SPLIT_5` eine spezifische Aufteilung auf genau fünf Ausgänge. Generische Split-Bausteine für andere Anzahlen (z. B. `AUDI_SPLIT_N`) existieren oft als Template, während dieser FB eine feste, aber häufig benötigte Konfiguration abdeckt.

- `AUDI_SPLIT_5_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

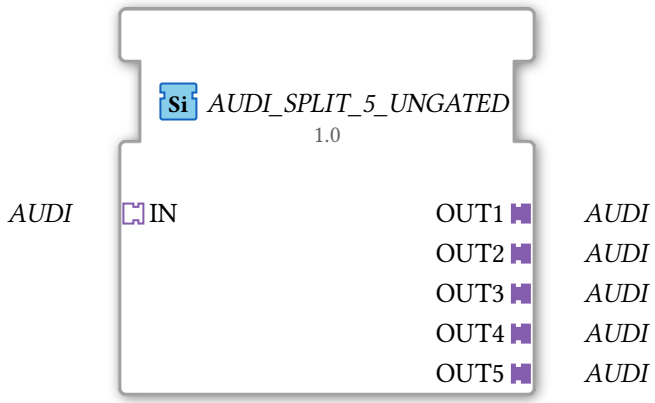
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

`AUDI_SPLIT_5` ist ein einfacher und effizienter Adapterbaustein zur Signalverteilung. Seine generische Auslegung und das Fehlen von Ereignis-/Datenlogik machen ihn leichtgewichtig und vielseitig einsetzbar. Er eignet sich ideal für alle Anwendungen, in denen ein AUDI-Signal auf fünf parallele Pfade aufgeteilt werden muss.

AUDI_SPLIT_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUDI_SPLIT_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUDI_SPLIT_5_UNGATED** dient dazu, ein einzelnes eingehendes AUDI-Signal auf fünf identische AUDI-Ausgänge zu verteilen. Er ist als generischer Adapterbaustein konzipiert und erlaubt eine flexible Weiterverarbeitung des Signals in mehreren Pfaden.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	`IN`	`adapter::types::unidirectional::AUDI`	Eingehendes AUDI-Signal
Plug	`OUT1`	`adapter::types::unidirectional::AUDI`	Erster Ausgang
Plug	`OUT2`	`adapter::types::unidirectional::AUDI`	Zweiter Ausgang
Plug	`OUT3`	`adapter::types::unidirectional::AUDI`	Dritter Ausgang
Plug	`OUT4`	`adapter::types::unidirectional::AUDI`	Vierter Ausgang
Plug	`OUT5`	`adapter::types::unidirectional::AUDI`	Fünfter Ausgang

Funktionsweise

Der Baustein leitet das am Socket `IN` anliegende AUDI-Signal unverändert und gleichzeitig an alle fünf Plugs `OUT1` bis `OUT5` weiter. Es erfolgt keine Signalverarbeitung, Wandlung oder Pufferung – die Verteilung erfolgt auf reiner Adapterebene, sobald eine Verbindung zu den Ausgängen besteht.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AUDI_SPLIT`) realisiert. Er kann über entsprechende Mechanismen der Laufzeitumgebung für verschiedene Datentypen instantiiert werden, solange diese dem unidirektionalen AUDI-Adapterprotokoll entsprechen.
- **Keine Zustandslogik**: Der Baustein enthält keine Ereignis- oder Datenverarbeitung und keinen internen Zustandsautomaten.
- **Einfache und performante Signalverteilung**: Aufgrund seiner reinen Adapterstruktur eignet er sich besonders für Szenarien, in denen ein Signal ohne Latenz vervielfältigt werden muss.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten. Das Verhalten ist deterministisch: Solange der Baustein in Betrieb ist, wird jedes an `IN` ankommende Signal sofort an alle Ausgänge weitergegeben.

Anwendungsszenarien

- **Verteilen von Steuersignalen** in einer industriellen Automatisierung, z. B. ein Start-/Stop-Befehl, der mehrere gleichartige Module erreichen soll.
- **Vervielfältigung eines Messwerts** für parallele Auswertungen oder Visualisierungen.
- **Aufbau von Redundanzpfaden**, indem dasselbe Signal mehrfach genutzt wird.

Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu Bausteinen wie `AUDI_MERGE_5` (zusammenführen mehrerer Signale) oder `AUDI_SPLIT_2` (nur zwei Ausgänge) bietet `AUDI_SPLIT_5_UNGATED` eine spezifische Aufteilung auf genau fünf Ausgänge. Generische Split-Bausteine für andere Anzahlen (z. B. `AUDI_SPLIT_N`) existieren oft als Template, während dieser FB eine feste, aber häufig benötigte Konfiguration abdeckt.

- **`AUDI_SPLIT_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

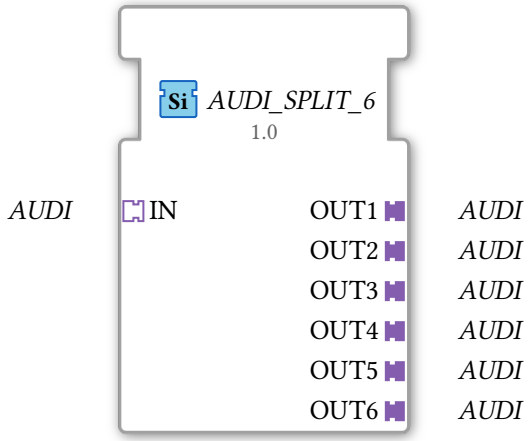
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

`AUDI_SPLIT_5_UNGATED` ist ein einfacher und effizienter Adapterbaustein zur Signalverteilung. Seine generische Auslegung und das Fehlen von Ereignis-/Datenlogik machen ihn leichtgewichtig und vielseitig einsetzbar. Er eignet sich ideal für alle Anwendungen, in denen ein AUDI-Signal auf fünf parallele Pfade aufgeteilt werden muss.

AUDI_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUDI_SPLIT_6** dient als generischer Verteiler (Splitter) für einen unidirektionalen AUDI-Adapter. Er empfängt einen eingehenden Adapter-Datenstrom über einen Socket und leitet diesen unverändert an sechs separate Ausgangsadapter (Plugs) weiter. Der Baustein ist generisch ausgelegt, sodass der konkrete Adaptertyp über das Attribut `GenericClassName` konfiguriert werden kann.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	**IN**	`adapter::types::unidirectional::AUDI`	Eingehender Adapter für den zu verteilenden Datenstrom.
Plug	**OUT1**	`adapter::types::unidirectional::AUDI`	Erster Ausgangsadapter (Kopie des Eingangs).
Plug	**OUT2**	`adapter::types::unidirectional::AUDI`	Zweiter Ausgangsadapter.
Plug	**OUT3**	`adapter::types::unidirectional::AUDI`	Dritter Ausgangsadapter.
Plug	**OUT4**	`adapter::types::unidirectional::AUDI`	Vierter Ausgangsadapter.
Plug	**OUT5**	`adapter::types::unidirectional::AUDI`	Fünfter Ausgangsadapter.
Plug	**OUT6**	`adapter::types::unidirectional::AUDI`	Sechster Ausgangsadapter.

Funktionsweise

Der Baustein agiert als reiner Verteiler ohne Datenverarbeitung. Der über den Socket **IN** eintreffende Datenstrom wird unverändert und parallel an alle sechs Ausgangsadapter (OUT1–OUT6) weitergeleitet. Die Verteilung erfolgt kontinuierlich, sobald der Eingangsadapter Daten bereitstellt. Da der Baustein generisch ist, kann er durch Setzen des Attributs `eclipse4diac::core::GenericClassName` auf einen spezifischen Adaptertyp (z. B. einen eigenen abgeleiteten Typ) angepasst werden.

Technische Besonderheiten

- **Generische Auslegung:** Der FB nutzt die 4diac-Generik-Mechanismen (`GenericClassName`). Dadurch kann der Adaptertyp zur Laufzeit oder bei der Instanziierung festgelegt werden, ohne den Baustein neu zu kompilieren.
- **Unidirektional:** Sowohl der Eingangs- als auch die Ausgangsadapter sind unidirektional. Eine Rückmeldung von den Ausgängen zum Eingang ist nicht vorgesehen.
- **Keine Ereignissteuerung:** Der Baustein besitzt kein Ereignis-Interface und kein ECC. Die Datenweitergabe erfolgt rein datengetrieben (sobald am IN-Socket Daten anliegen).
- **Kein interner Zustand:** Es findet keine Zustandsänderung statt; der FB ist stateless.

Zustandsübersicht

Es existiert kein Zustandsautomat (ECC). Der Baustein führt keine sequenziellen Abläufe aus.

Anwendungsszenarien

- **Datenverteilung:** Ein von einer Sensorik oder einem anderen FB bereitgestellter AUDI-Datenstrom soll an mehrere nachfolgende Komponenten gleichzeitig gesendet werden, z. B. an eine Visualisierung, eine Logging-Komponente und eine Steuerungslogik.
- **Parallele Verarbeitung:** Wenn derselbe Datenstrom in unterschiedlichen Pfadabschnitten parallel weiterverarbeitet werden muss, z. B. Filterung, Aufbereitung, Speicherung.
- **Generischer Einsatz:** Durch die konfigurierbare Typzuweisung kann der FB in verschiedenen Projekten mit unterschiedlichen Adapterdefinitionen eingesetzt werden, ohne die Quellbasis anpassen zu müssen.

Vergleich mit ähnlichen Bausteinen

Es existieren andere Split-Bausteine (z. B. `AUDI_SPLIT_2`, `AUDI_SPLIT_4`), die sich lediglich in der Anzahl der Ausgänge unterscheiden. Der `AUDI_SPLIT_6` bietet mit sechs Ausgängen eine höhere Verteilungsdichte. Im Gegensatz zu reinen Daten-Splittern für Basisdatentypen arbeitet dieser FB ausschließlich mit AUDI-Adaptern und berücksichtigt die unidirektionale Kopplung.

- `AUDI_SPLIT_6_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs

unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

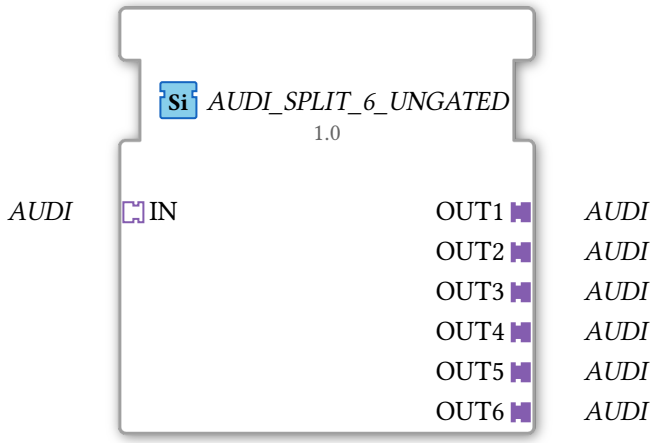
Der `AUDI_SPLIT_6` ist ein einfacher, aber flexibler Baustein zur Vervielfachung von Adapterverbindungen in einer 4diac-Umgebung. Seine generische Natur und die schlanke Implementierung ohne Ereignis-Overhead machen ihn ideal für Szenarien, in denen ein eingehender Datenstrom an mehrere unabhängige Empfänger verteilt werden muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUDI_SPLIT_6_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUDI_SPLIT_6`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUDI_SPLIT_6_UNGATED** dient als generischer Verteiler (Splitter) für einen unidirektionalen AUDI-Adapter. Er empfängt einen eingehenden Adapter-Datenstrom über einen Socket und leitet diesen unverändert an sechs separate Ausgangsadapter (Plugs) weiter. Der Baustein ist generisch ausgelegt, sodass der konkrete Adaptertyp über das Attribut `GenericClassName` konfiguriert werden kann.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	**IN**	`adapter::types::unidirectional::AUDI`	Eingehender Adapter für den zu verteilenden Datenstrom.
Plug	**OUT1**	`adapter::types::unidirectional::AUDI`	Erster Ausgangsadapter (Kopie des Eingangs).
Plug	**OUT2**	`adapter::types::unidirectional::AUDI`	Zweiter Ausgangsadapter.
Plug	**OUT3**	`adapter::types::unidirectional::AUDI`	Dritter Ausgangsadapter.
Plug	**OUT4**	`adapter::types::unidirectional::AUDI`	Vierter Ausgangsadapter.
Plug	**OUT5**	`adapter::types::unidirectional::AUDI`	Fünfter Ausgangsadapter.
Plug	**OUT6**	`adapter::types::unidirectional::AUDI`	Sechster Ausgangsadapter.

Funktionsweise

Der Baustein agiert als reiner Verteiler ohne Datenverarbeitung. Der über den Socket **IN** eintreffende Datenstrom wird unverändert und parallel an alle sechs Ausgangsadapter (OUT1–OUT6) weitergeleitet. Die Verteilung erfolgt kontinuierlich, sobald der Eingangsadapter Daten bereitstellt. Da der Baustein generisch ist, kann er durch Setzen des Attributs `eclipse4diac::core::GenericClassName` auf einen spezifischen Adaptertyp (z. B. einen eigenen abgeleiteten Typ) angepasst werden.

Technische Besonderheiten

- **Generische Auslegung:** Der FB nutzt die 4diac-Generik-Mechanismen (`GenericClassName`). Dadurch kann der Adaptertyp zur Laufzeit oder bei der Instanziierung festgelegt werden, ohne den Baustein neu zu kompilieren.
- **Unidirektional:** Sowohl der Eingangs- als auch die Ausgangsadapter sind unidirektional. Eine Rückmeldung von den Ausgängen zum Eingang ist nicht vorgesehen.
- **Keine Ereignissteuerung:** Der Baustein besitzt kein Ereignis-Interface und kein ECC. Die Datenweitergabe erfolgt rein datengetrieben (sobald am IN-Socket Daten anliegen).
- **Kein interner Zustand:** Es findet keine Zustandsänderung statt; der FB ist stateless.

Zustandsübersicht

Es existiert kein Zustandsautomat (ECC). Der Baustein führt keine sequenziellen Abläufe aus.

Anwendungsszenarien

- **Datenverteilung:** Ein von einer Sensorik oder einem anderen FB bereitgestellter AUDI-Datenstrom soll an mehrere nachfolgende Komponenten gleichzeitig gesendet werden, z. B. an eine Visualisierung, eine Logging-Komponente und eine Steuerungslogik.
- **Parallele Verarbeitung:** Wenn derselbe Datenstrom in unterschiedlichen Pfadabschnitten parallel weiterverarbeitet werden muss, z. B. Filterung, Aufbereitung, Speicherung.
- **Generischer Einsatz:** Durch die konfigurierbare Typzuweisung kann der FB in verschiedenen Projekten mit unterschiedlichen Adapterdefinitionen eingesetzt werden, ohne die Quellbasis anpassen zu müssen.

Vergleich mit ähnlichen Bausteinen

Es existieren andere Split-Bausteine (z. B. `AUDI_SPLIT_2`, `AUDI_SPLIT_4`), die sich lediglich in der Anzahl der Ausgänge unterscheiden. Der `AUDI_SPLIT_6_UNGATED` bietet mit sechs Ausgängen eine höhere Verteilungsdichte. Im Gegensatz zu reinen Daten-Splittern für Basisdatentypen arbeitet dieser FB ausschließlich mit AUDI-Adaptern und berücksichtigt die unidirektionale Kopplung.

- **`AUDI_SPLIT_6`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

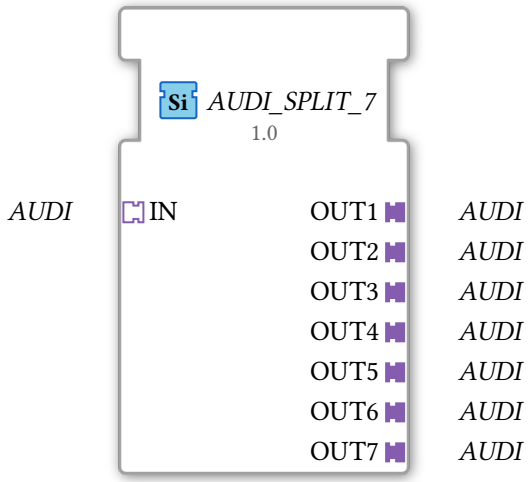
Fazit

Der `AUDI_SPLIT_6_UNGATED` ist ein einfacher, aber flexibler Baustein zur Vervielfachung von Adapterverbindungen in einer 4diac-Umgebung. Seine generische Natur und die schlanke Implementierung ohne Ereignis-Overhead machen ihn ideal für Szenarien, in denen ein eingehender Datenstrom an mehrere unabhängige Empfänger verteilt werden muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUDI_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AUDI_SPLIT_7` dient dazu, ein einzelnes AUDI-Eingangssignal auf sieben identische AUDI-Ausgangssignale zu verteilen. Er ist als generischer Baustein konzipiert und eignet sich für alle unidirektionalen AUDI-Adaptertypen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Name	Typ	Richtung
`IN`	`adapter::types::unidirectional::AUDI`	Socket (Eingang)

OUT1	adapter::types::unidirectional::AUDI	Plug (Ausgang)
OUT2	adapter::types::unidirectional::AUDI	Plug (Ausgang)
OUT3	adapter::types::unidirectional::AUDI	Plug (Ausgang)
OUT4	adapter::types::unidirectional::AUDI	Plug (Ausgang)
OUT5	adapter::types::unidirectional::AUDI	Plug (Ausgang)
OUT6	adapter::types::unidirectional::AUDI	Plug (Ausgang)
OUT7	adapter::types::unidirectional::AUDI	Plug (Ausgang)

Alle Adapter sind vom unidirektionalen Typ AUDI und übertragen Datenfluss ausschließlich vom Socket zu den Plugs.

Funktionsweise

Der Baustein kopiert die über den Socket IN eingehenden AUDI-Daten unverändert auf alle sieben Plug-Ausgänge OUT1 bis OUT7. Es erfolgt keine Verarbeitung oder Umwandlung der Daten – die Funktion beschränkt sich auf eine reine 1-zu-7-Verteilung (Fan-Out). Da keine Ereignisse oder explizite Datenports vorhanden sind, wird das Signalverhalten vollständig durch die angeschlossenen Adapter definiert.

Technische Besonderheiten

- **Generischer Typ**: Über das Attribut eclipse4diac::core::GenericClassName ('GEN_AUDI_SPLIT') wird der Baustein als generisch gekennzeichnet. Damit kann er für verschiedene konkrete Ausprägungen des AUDI-Adapters verwendet werden, ohne dass eine separate Implementierung erforderlich ist.
- **Keine Laufzeitabhängigkeit**: Der Baustein besitzt keinen Algorithmus und kein Zustandsdiagramm; die Signalweitergabe erfolgt rein strukturell durch die Adapterverkabelung.
- **Typ-Hash**: Das Attribut eclipse4diac::core::TypeHash ist leer, was bedeutet, dass die Typidentität zur Laufzeit nicht zusätzlich abgesichert wird.

Zustandsübersicht

Der Baustein enthält keine Zustandsmaschine (ECC). Das Verhalten ist vollständig deterministisch und ereignislos – eine Zustandsdarstellung ist daher nicht erforderlich.

Anwendungsszenarien

- **Signalverteilung**: Ein von einer Sensor- oder Steuereinheit kommendes AUDI-Signal muss parallel an mehrere Verbraucher (Aktoren, Anzeigen, Überwachungssysteme) weitergeleitet werden.
- **Systemerweiterung**: Vorhandene Systeme, die ein einzelnes AUDI-Signal bereitstellen, sollen um weitere Komponenten ergänzt werden, ohne die Quelllogik zu ändern.
- **Testaufbauten**: Ein generiertes Testsignal soll zeitgleich an mehrere Prüflinge gesendet werden.

Vergleich mit ähnlichen Bausteinen

Baustein	Verteilung	Adaptertyp
AUDI_SPLIT_7	1 Eingang ⟶ 7 Ausgänge	Unidirektional AUDI
SPLIT_1_TO_2 (analog)	1 ⟶ 2	Beliebig (generisch)
AUDI_MERGE	Mehrere Eingänge ⟶ 1	Unidirektional AUDI

Während AUDI_SPLIT_7 auf den spezifischen AUDI-Adaptertyp ausgelegt ist, existieren generische Split-Bausteine für andere Datenformate. Die Beschränkung auf eine feste Anzahl von sieben Ausgängen unterscheidet ihn von flexiblen Splittern mit parametrierbarer Ausgangsanzahl.

- **AUDI_SPLIT_7_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs

unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

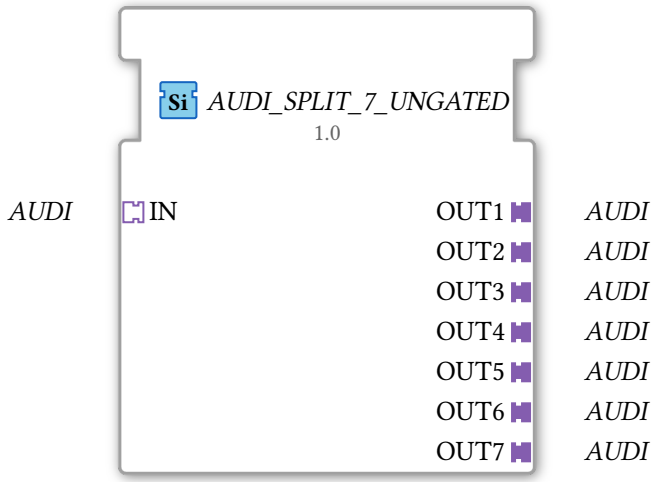
`AUDI_SPLIT_7` ist ein einfacher, aber effektiver Baustein zum Verteilen eines AUDI-Signals auf sieben identische Pfade. Durch die generische Auslegung kann er in verschiedenen Kontexten wiederverwendet werden. Der Verzicht auf Ereignisse und interne Logik macht ihn leichtgewichtig und zuverlässig für reine Signalverteilungsaufgaben.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUDI_SPLIT_7_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUDI_SPLIT_7`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AUDI_SPLIT_7_UNGATED` dient dazu, ein einzelnes AUDI-Eingangssignal auf sieben identische AUDI-Ausgangssignale zu verteilen. Er ist als generischer Baustein konzipiert und eignet sich für alle unidirektionalen AUDI-Adaptertypen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Name	Typ	Richtung
IN	adapter::types::unidirectional::AUDI	Socket (Eingang)
OUT1	adapter::types::unidirectional::AUDI	Plug (Ausgang)
OUT2	adapter::types::unidirectional::AUDI	Plug (Ausgang)
OUT3	adapter::types::unidirectional::AUDI	Plug (Ausgang)
OUT4	adapter::types::unidirectional::AUDI	Plug (Ausgang)
OUT5	adapter::types::unidirectional::AUDI	Plug (Ausgang)
OUT6	adapter::types::unidirectional::AUDI	Plug (Ausgang)
OUT7	adapter::types::unidirectional::AUDI	Plug (Ausgang)

Alle Adapter sind vom unidirektionalen Typ `AUDI` und übertragen Datenfluss ausschließlich vom Socket zu den Plugs.

Funktionsweise

Der Baustein kopiert die über den Socket `IN` eingehenden AUDI-Daten unverändert auf alle sieben Plug-Ausgänge `OUT1` bis `OUT7`. Es erfolgt keine Verarbeitung oder Umwandlung der Daten – die Funktion beschränkt sich auf eine reine 1-zu-7-Verteilung (Fan-Out). Da keine Ereignisse oder explizite Datenports vorhanden sind, wird das Signalverhalten vollständig durch die angeschlossenen Adapter definiert.

Technische Besonderheiten

- **Generischer Typ**: Über das Attribut `eclipse4diac::core::GenericClassName ('GEN_AUDI_SPLIT')` wird der Baustein als generisch gekennzeichnet. Damit kann er für verschiedene konkrete Ausprägungen des AUDI-Adapters verwendet werden, ohne dass eine separate Implementierung erforderlich ist.
- **Keine Laufzeitabhängigkeit**: Der Baustein besitzt keinen Algorithmus und kein Zustandsdiagramm; die Signalweitergabe erfolgt rein strukturell durch die Adapterverkabelung.
- **Typ-Hash**: Das Attribut `eclipse4diac::core::TypeHash` ist leer, was bedeutet, dass die Typidentität zur Laufzeit nicht zusätzlich abgesichert wird.

Zustandsübersicht

Der Baustein enthält keine Zustandsmaschine (ECC). Das Verhalten ist vollständig deterministisch und ereignislos – eine Zustandsdarstellung ist daher nicht erforderlich.

Anwendungsszenarien

- **Signalverteilung**: Ein von einer Sensor- oder Steuereinheit kommendes AUDI-Signal muss parallel an mehrere Verbraucher (Aktoren, Anzeigen, Überwachungssysteme) weitergeleitet werden.
- **Systemerweiterung**: Vorhandene Systeme, die ein einzelnes AUDI-Signal bereitstellen, sollen um weitere Komponenten ergänzt werden, ohne die Quelllogik zu ändern.
- **Testaufbauten**: Ein generiertes Testsignal soll zeitgleich an mehrere Prüflinge gesendet werden.

Vergleich mit ähnlichen Bausteinen

Baustein	Verteilung	Adaptertyp
AUDI_SPLIT_7_UNGATED	1 Eingang ⟶ 7 Ausgänge	Unidirektional AUDI
SPLIT_1_TO_2 (analog)	1 ⟶ 2	Beliebig (generisch)
AUDI_MERGE	Mehrere Eingänge ⟶ 1	Unidirektional AUDI

Während `AUDI_SPLIT_7_UNGATED` auf den spezifischen AUDI-Adaptertyp ausgelegt ist, existieren generische Split-Bausteine für andere Datenformate. Die Beschränkung auf eine feste Anzahl von sieben Ausgängen unterscheidet ihn von flexiblen Splittern mit parametrierbarer Ausgangsanzahl.

- **AUDI_SPLIT_7_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

- **AUDI_SPLIT_7**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

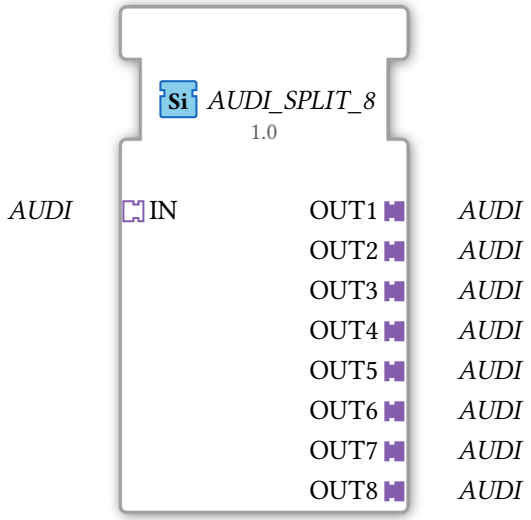
Fazit

`AUDI_SPLIT_7_UNGATED` ist ein einfacher, aber effektiver Baustein zum Verteilen eines AUDI-Signals auf sieben identische Pfade. Durch die generische Auslegung kann er in verschiedenen Kontexten wiederverwendet werden. Der Verzicht auf Ereignisse und interne Logik macht ihn leichtgewichtig und zuverlässig für reine Signalverteilungsaufgaben.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUDI_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUDI_SPLIT_8** dient dazu, ein eingehendes unidirektionales **AUDI**-Adapter-Signal auf acht parallele Ausgänge zu verteilen. Er ist als generischer Baustein (generic FB) implementiert und ermöglicht eine einfache Weiterleitung eines Adaptersignals an bis zu acht Empfänger.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Typ	Richtung	Name	Beschreibung
`adapter::types::unidirectional::AUDI`	Socket (Eingang)	`IN`	Eingangsadapter, der das zu verteilende Signal empfängt.
`adapter::types::unidirectional::AUDI`	Plug (Ausgang)	`OUT1` .. `OUT8`	Acht Ausgangsadapter, die jeweils das identische Signal des Eingangs bereitstellen.

Funktionsweise

Der Baustein leitet das am Socket `IN` anliegende **AUDI**-Adapter-Signal unverändert an alle acht Plugs (`OUT1` bis `OUT8`) weiter. Es findet keine Verarbeitung, Filterung oder Pufferung statt – die Verteilung erfolgt vollständig passiv auf Adapterebene. Durch die Verwendung eines generischen Typs (siehe Technische Besonderheiten) kann der Baustein flexibel für verschiedene konkrete AUDI-Adapterausprägungen eingesetzt werden.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ deklariert (`eclipse4diac::core::GenericClassName = 'GEN_AUDI_SPLIT'`). Dies erlaubt die Wiederverwendung für beliebige AUDI-Adapter-Spezialisierungen, ohne den FB selbst ändern zu müssen.
- **Keine Zustandslogik**: Da ausschließlich Adapter verwendet werden, besitzt der Baustein weder Ereignisse noch eine eigene Zustandsmaschine.
- **Platzersparnis**: Durch die Zusammenfassung von acht Ausgängen in einem einzigen FB wird das Netzwerkdiagramm übersichtlicher als bei Verwendung mehrerer einfacher Split-Blöcke.

Zustandsübersicht

Der Baustein besitzt keine interne Zustandsmaschine (kein ECC). Die Funktionalität ist rein kombinatorisch: Das Signal am Eingang wird permanent auf alle Ausgänge durchgeschaltet.

Anwendungsszenarien

- **Signalbroadcast** in Steuerungsanwendungen, bei denen ein einziges AUDI-Adapter-Signal an mehrere unabhängige Empfängerbausteine verteilt werden muss (z. B. intelligente Sensoren, Aktuatoren oder Buskoppler).
- **Erweiterung von Teilnehmern** in einer bestehenden AUDI-Verdrahtung, ohne das Original-Signal zu beeinflussen.
- **Strukturierte Automatisierungslösungen** in landwirtschaftlichen Maschinen, in denen Adapter nach IEC 61499 für die Kommunikation zwischen Komponenten eingesetzt werden (z. B. in HR Agrartechnik GmbH Systemen).

Vergleich mit ähnlichen Bausteinen

- **Einfacher Split-FB (z. B. SPLIT_2)** : Verteilt ein Signal auf zwei Ausgänge – der AUDI_SPLIT_8 bietet acht Ausgänge in einem Baustein.
- **Data Distribution FB**: Einige Bibliotheken stellen generische Verteiler für Datenports bereit; dieser FB ist speziell für den AUDI-Adapter-Typ optimiert und benötigt keine zusätzliche Konfiguration der Datenstruktur.
- **Manuelle Parallelschaltung**: Mehrere AUDI-Socket-zu-Plug-Verbindungen könnten das Gleiche bewirken, sind aber unübersichtlicher und fehleranfälliger.
- **`AUDI_SPLIT_8_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

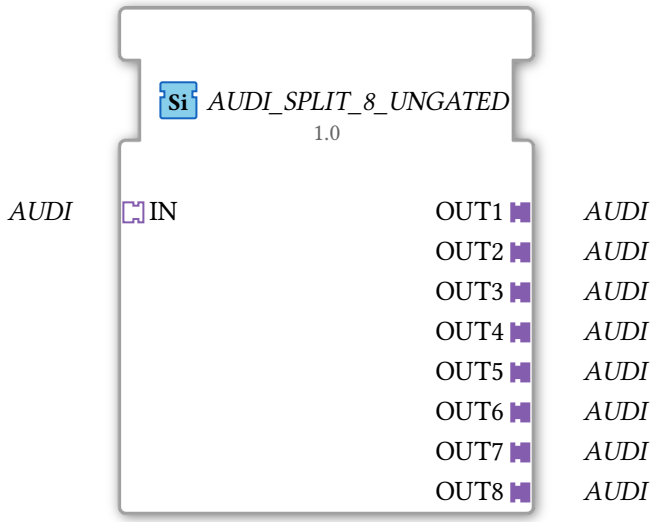
AUDI_SPLIT_8 ist ein kompakter, generischer Spezialbaustein zur Signalverteilung auf Adapterebene. Er reduziert den Modellierungsaufwand, erhöht die Lesbarkeit von Steuerungsprogrammen und lässt sich durch seine generische Natur flexibel in verschiedenen AUDI-basierten Umgebungen einsetzen. Für Anwendungen, die eine 1:8-Aufteilung eines unidirektionalen AUDI-Adapters benötigen, stellt er eine ideale und saubere Lösung dar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 Gesamtwiderstand in Reihen- & Parallelschaltung auf ms-muc-docs.de

AUDI_SPLIT_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUDI_SPLIT_8`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUDI_SPLIT_8_UNGATED** dient dazu, ein eingehendes unidirektionales **AUDI**-Adapter-Signal auf acht parallele Ausgänge zu verteilen. Er ist als generischer Baustein (generic FB) implementiert und ermöglicht eine einfache Weiterleitung eines Adaptersignals an bis zu acht Empfänger.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Typ	Richtung	Name	Beschreibung
`adapter::types::unidirectional::AUDI`	Socket (Eingang)	`IN`	Eingangsadapter, der das zu verteilende Signal empfängt.
`adapter::types::unidirectional::AUDI`	Plug (Ausgang)	`OUT1` .. `OUT8`	Acht Ausgangsadapter, die jeweils das identische Signal des Eingangs bereitstellen.

Funktionsweise

Der Baustein leitet das am Socket `IN` anliegende **AUDI**-Adapter-Signal unverändert an alle acht Plugs (`OUT1` bis `OUT8`) weiter. Es findet keine Verarbeitung, Filterung oder Pufferung statt – die Verteilung erfolgt vollständig passiv auf Adapterebene. Durch die Verwendung eines generischen Typs (siehe Technische Besonderheiten) kann der Baustein flexibel für verschiedene konkrete AUDI-Adapterausprägungen eingesetzt werden.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ deklariert (`eclipse4diac::core::GenericClassName = 'GEN_AUDI_SPLIT'`). Dies erlaubt die Wiederverwendung für beliebige AUDI-Adapter-Spezialisierungen, ohne den FB selbst ändern zu müssen.
- **Keine Zustandslogik**: Da ausschließlich Adapter verwendet werden, besitzt der Baustein weder Ereignisse noch eine eigene Zustandsmaschine.
- **Platzersparnis**: Durch die Zusammenfassung von acht Ausgängen in einem einzigen FB wird das Netzwerkdiagramm übersichtlicher als bei Verwendung mehrerer einfacher Split-Blöcke.

Zustandsübersicht

Der Baustein besitzt keine interne Zustandsmaschine (kein ECC). Die Funktionalität ist rein kombinatorisch: Das Signal am Eingang wird permanent auf alle Ausgänge durchgeschaltet.

Anwendungsszenarien

- **Signalbroadcast** in Steuerungsanwendungen, bei denen ein einziges AUDI-Adapter-Signal an mehrere unabhängige Empfängerbausteine verteilt werden muss (z. B. intelligente Sensoren, Aktuatoren oder Buskoppler).
- **Erweiterung von Teilnehmern** in einer bestehenden AUDI-Verdrahtung, ohne das Original-Signal zu beeinflussen.
- **Strukturierte Automatisierungslösungen** in landwirtschaftlichen Maschinen, in denen Adapter nach IEC 61499 für die Kommunikation zwischen Komponenten eingesetzt werden (z. B. in HR Agrartechnik GmbH Systemen).

Vergleich mit ähnlichen Bausteinen

- **Einfacher Split-FB (z. B. SPLIT_2)** : Verteilt ein Signal auf zwei Ausgänge – der AUDI_SPLIT_8_UNGATED bietet acht Ausgänge in einem Baustein.
- **Data Distribution FB**: Einige Bibliotheken stellen generische Verteiler für Datenports bereit; dieser FB ist speziell für den AUDI-Adapter-Typ optimiert und benötigt keine zusätzliche Konfiguration der Datenstruktur.
- **Manuelle Parallelschaltung**: Mehrere AUDI-Socket-zu-Plug-Verbindungen könnten das Gleiche bewirken, sind aber unübersichtlicher und fehleranfälliger.
- **`AUDI_SPLIT_8`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

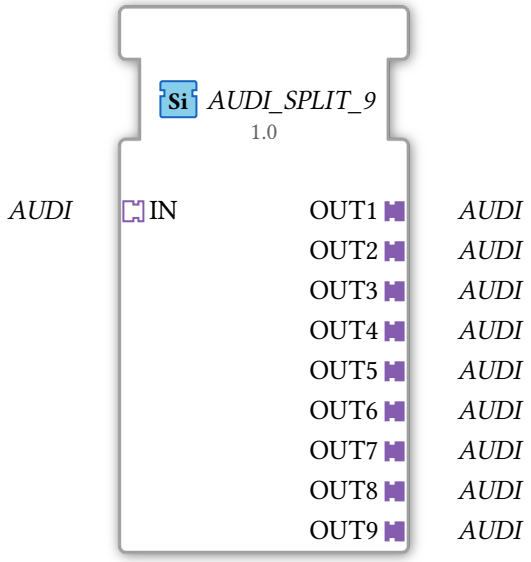
Fazit

AUDI_SPLIT_8_UNGATED ist ein kompakter, generischer Spezialbaustein zur Signalverteilung auf Adapterebene. Er reduziert den Modellierungsaufwand, erhöht die Lesbarkeit von Steuerungsprogrammen und lässt sich durch seine generische Natur flexibel in verschiedenen AUDI-basierten Umgebungen einsetzen. Für Anwendungen, die eine 1:8-Aufteilung eines unidirektionalen AUDI-Adapters benötigen, stellt er eine ideale und saubere Lösung dar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 Gesamtwiderstand in Reihen- & Parallelschaltung auf ms-muc-docs.de

AUDI_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AUDI_SPLIT_9` dient dazu, ein ankommendes AUDI-Signal über einen Socket auf neun identische AUDI-Plugs zu verteilen. Er realisiert eine 1-zu-9-Split-Funktionalität, bei der alle Ausgänge stets den gleichen Wert wie der Eingang führen. Der Baustein ist als generischer FB (GenericClassName `GEN_AUDI_SPLIT`) definiert und ermöglicht die einfache Vervielfältigung von AUDI-Datenströmen innerhalb einer 4diac-IDE-Applikation.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden. Die Datenübertragung erfolgt ausschließlich über die Adapter-Schnittstellen.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden. Die Ausgangssignale werden direkt über die Adapter-Plugs bereitgestellt.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden. Die Eingangsdaten werden über den Socket-Adapter `IN` eingelesen.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden. Die Ausgangsdaten werden über die neun Plug-Adapter `OUT1` bis `OUT9` ausgegeben.

Adapter

Adapter	Typ	Richtung	Beschreibung
`IN`	`adapter::types::unidirectional::AUDI`	Socket (Eingang)	Empfängt das zu verteilende AUDI-Signal.
`OUT1` … `OUT9`	`adapter::types::unidirectional::AUDI`	Plug (Ausgang)	Neun Ausgänge, die den Wert von `IN` replizieren. Jeder Ausgang ist identisch und unabhängig.

Funktionsweise

Der Baustein stellt eine passive Kopplung zwischen dem Socket `IN` und den neun Plugs `OUT1` bis `OUT9` her. Immer wenn der Socket ein gültiges AUDI-Datenpaket empfängt, wird dieses unverändert an alle neun Plugs weitergereicht. Es findet keinerlei Datenmodifikation, Pufferung oder zeitliche Verzögerung statt. Der Baustein besitzt kein eigenes Verhalten – er fungiert als reine Verdrahtungshilfe zur Signalfächerung.

Technische Besonderheiten

- **Generische Implementierung:** Der FB trägt das Attribut `eclipse4diac::core::GenericClassName = 'GEN_AUDI_SPLIT'`, wodurch er als Instanz eines generischen Split-Bausteins erkannt wird und in Werkzeugen mit generischer Typunterstützung korrekt dargestellt werden kann.
- **Keine Ereignissteuerung:** Da keinerlei Ereignisse vorhanden sind, erfolgt die Datenweitergabe rein über die Adapter-Mechanismen des Zielsystems (z. B. kontinuierlich oder zyklisch, abhängig von der Laufzeitumgebung).
- **Skalierbarkeit:** Die Anzahl der Ausgänge ist fest auf neun ausgelegt. Für andere Anzahlen müssen separate FB-Typen (z. B. `AUDI_SPLIT_2`, `AUDI_SPLIT_4` etc.) definiert oder ein parametrierbarer Baustein verwendet werden.

Zustandsübersicht

Der Baustein besitzt keinen internen Zustand. Sein Verhalten ist vollständig statisch und durch die reine Weiterleitung des Signals definiert. Eine Zustandsmaschine existiert nicht.

Anwendungsszenarien

- **Verteilen eines Mess-Signals** an mehrere parallele Auswerte- oder Steuereinheiten.
- **Aufteilung eines Steuerbefehls** an mehrere Aktoren, sofern diese das gleiche AUDI-Protokoll verwenden.
- **Test- und Simulationsumgebungen**, bei denen ein generiertes Signal an mehrere Beobachter (z. B. Logging, Visualisierung, Regelung) gleichzeitig gesendet werden soll.
- **Ersatz für mehrere Kopien** eines Quellbausteins, wenn die Quelle nur einmal vorhanden ist.

Vergleich mit ähnlichen Bausteinen

Gegenüber einem allgemeinen `AUDI_MERGE` (der mehrere Eingänge zu einem Ausgang zusammenführt) oder einem `AUDI_SELECT` (der zwischen mehreren Eingängen umschaltet) bietet der Split-Baustein die exakte Umkehrfunktion: Vervielfältigung statt Zusammenführung oder Selektion. Im Vergleich zu einem eigenen, in Software realisierten Split (z. B. durch mehrfache Verbindung eines Ausgangs in der Applikation) gewährleistet der FB eine saubere, typisierte und wiederverwendbare Schnittstelle und vermeidet potenzielle Mehrfachverbindungsfehler im Editor.

- **`AUDI_SPLIT_9_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

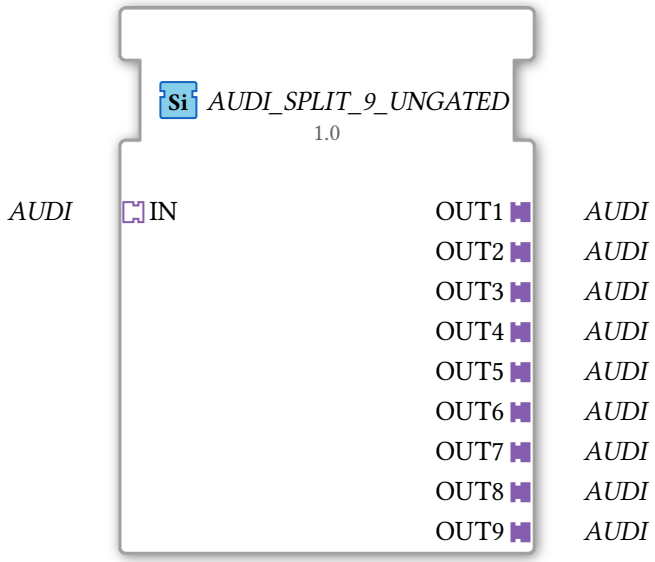
Der `AUDI_SPLIT_9` ist ein einfacher, aber nützlicher Funktionsbaustein zur Vervielfältigung von AUDI-Signalen. Seine Implementation als generischer Typ und die klare Trennung von Eingang und neun Ausgängen erleichtert den Aufbau modularer Automatisierungslösungen. Aufgrund seiner statischen Natur ist er besonders geeignet für datenflussorientierte Anwendungen, bei denen keine Ereignissteuerung erforderlich ist.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUDI_SPLIT_9_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUDI_SPLIT_9`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AUDI_SPLIT_9_UNGATED` dient dazu, ein ankommendes AUDI-Signal über einen Socket auf neun identische AUDI-Plugs zu verteilen. Er realisiert eine 1-zu-9-Split-Funktionalität, bei der alle Ausgänge stets den gleichen Wert wie der Eingang führen. Der Baustein ist als generischer FB (GenericClassName `GEN_AUDI_SPLIT`) definiert und ermöglicht die einfache Vervielfältigung von AUDI-Datenströmen innerhalb einer 4diac-IDE-Applikation.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden. Die Datenübertragung erfolgt ausschließlich über die Adapter-Schnittstellen.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden. Die Ausgangssignale werden direkt über die Adapter-Plugs bereitgestellt.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden. Die Eingangsdaten werden über den Socket-Adapter `IN` eingelesen.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden. Die Ausgangsdaten werden über die neun Plug-Adapter `OUT1` bis `OUT9` ausgegeben.

Adapter

Adapter	Typ	Richtung	Beschreibung
`IN`	`adapter::types::unidirectional::AUDI`	Socket (Eingang)	Empfängt das zu verteilende AUDI-Signal.
`OUT1` … `OUT9`	`adapter::types::unidirectional::AUDI`	Plug (Ausgang)	Neun Ausgänge, die den Wert von `IN` replizieren. Jeder Ausgang ist identisch und unabhängig.

Funktionsweise

Der Baustein stellt eine passive Kopplung zwischen dem Socket `IN` und den neun Plugs `OUT1` bis `OUT9` her. Immer wenn der Socket ein gültiges AUDI-Datenpaket empfängt, wird dieses unverändert an alle neun Plugs weitergereicht. Es findet keinerlei Datenmodifikation, Pufferung oder zeitliche Verzögerung statt. Der Baustein besitzt kein eigenes Verhalten – er fungiert als reine Verdrahtungshilfe zur Signalfächerung.

Technische Besonderheiten

- **Generische Implementierung:** Der FB trägt das Attribut `eclipse4diac::core::GenericClassName = 'GEN_AUDI_SPLIT'`, wodurch er als Instanz eines generischen Split-Bausteins erkannt wird und in Werkzeugen mit generischer Typunterstützung korrekt dargestellt werden kann.
- **Keine Ereignissteuerung:** Da keinerlei Ereignisse vorhanden sind, erfolgt die Datenweitergabe rein über die Adapter-Mechanismen des Zielsystems (z. B. kontinuierlich oder zyklisch, abhängig von der Laufzeitumgebung).
- **Skalierbarkeit:** Die Anzahl der Ausgänge ist fest auf neun ausgelegt. Für andere Anzahlen müssen separate FB-Typen (z. B. `AUDI_SPLIT_2`, `AUDI_SPLIT_4` etc.) definiert oder ein parametrierbarer Baustein verwendet werden.

Zustandsübersicht

Der Baustein besitzt keinen internen Zustand. Sein Verhalten ist vollständig statisch und durch die reine Weiterleitung des Signals definiert. Eine Zustandsmaschine existiert nicht.

Anwendungsszenarien

- **Verteilen eines Mess-Signals** an mehrere parallele Auswerte- oder Steuereinheiten.
- **Aufteilung eines Steuerbefehls** an mehrere Aktoren, sofern diese das gleiche AUDI-Protokoll verwenden.
- **Test- und Simulationsumgebungen**, bei denen ein generiertes Signal an mehrere Beobachter (z. B. Logging, Visualisierung, Regelung) gleichzeitig gesendet werden soll.
- **Ersatz für mehrere Kopien** eines Quellbausteins, wenn die Quelle nur einmal vorhanden ist.

Vergleich mit ähnlichen Bausteinen

Gegenüber einem allgemeinen `AUDI_MERGE` (der mehrere Eingänge zu einem Ausgang zusammenführt) oder einem `AUDI_SELECT` (der zwischen mehreren Eingängen umschaltet) bietet der Split-Baustein die exakte Umkehrfunktion: Vervielfältigung statt Zusammenführung oder Selektion. Im Vergleich zu einem eigenen, in Software realisierten Split (z. B. durch mehrfache Verbindung eines Ausgangs in der Applikation) gewährleistet der FB eine saubere, typisierte und wiederverwendbare Schnittstelle und vermeidet potenzielle Mehrfachverbindungsfehler im Editor.

- **`AUDI_SPLIT_9`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

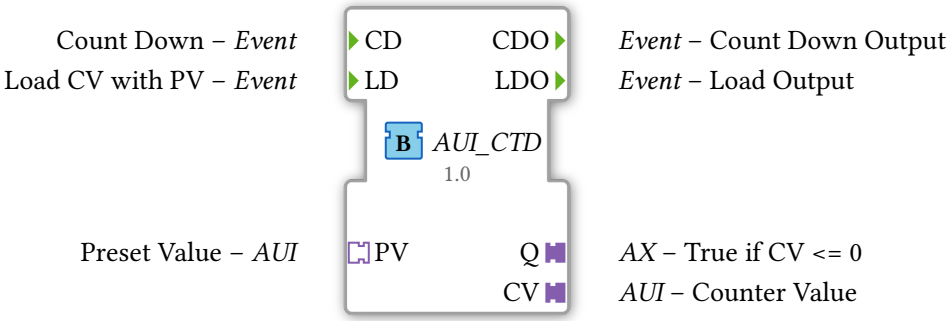
Der `AUDI_SPLIT_9_UNGATED` ist ein einfacher, aber nützlicher Funktionsbaustein zur Vervielfältigung von AUDI-Signalen. Seine Implementation als generischer Typ und die klare Trennung von Eingang und neun Ausgängen erleichtert den Aufbau modularer Automatisierungslösungen. Aufgrund seiner statischen Natur ist er besonders geeignet für datenflussorientierte Anwendungen, bei denen keine Ereignissteuerung erforderlich ist.

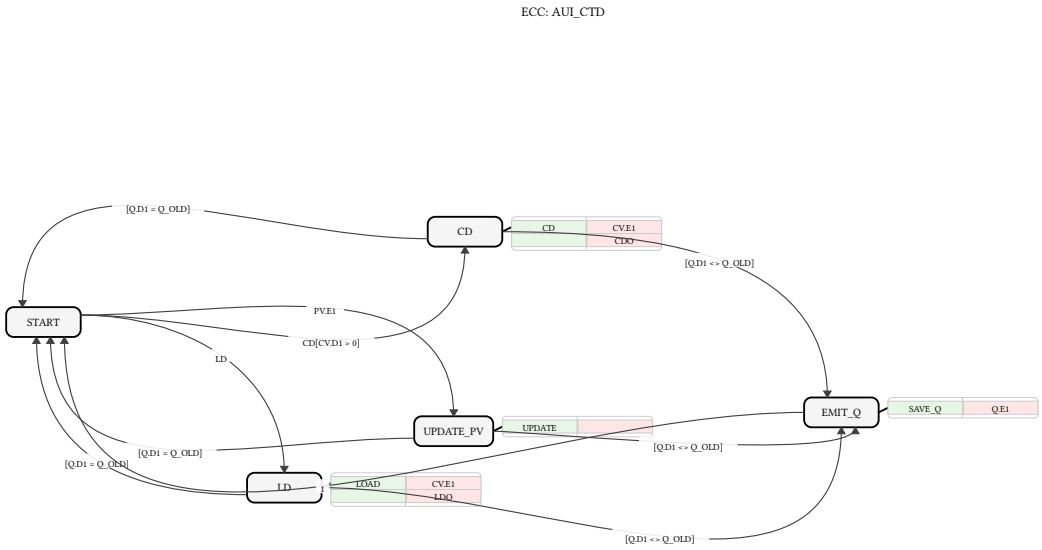
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

UINT

AUI_CTD





• – ○ • – ○ • – ○ •

Einleitung

Der **AUI_CTD** ist ein ereignisgesteuerter Abwärtszähler (Down Counter) mit Adapter-Schnittstellen. Er verfügt über eine **On-Change-Auslösung**: Das Ausgangssignal **Q** wird nur dann aktualisiert, wenn sich sein logischer Zustand tatsächlich ändert. Dadurch werden unnötige Ereignisse vermieden und die Effizienz in ereignisbasierten Systemen gesteigert.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung
CD	Count Down – reduziert den Zählerwert um 1.
LD	Load – lädt den Zählerwert **CV** mit dem Preset-Wert **PV**.

Ereignis-Ausgänge

Ereignis	Beschreibung
CDO	Count Down Output – wird nach jedem CD-Schritt ausgegeben.
LDO	Load Output – wird nach einem erfolgreichen Laden ausgegeben.

Daten-Eingänge

Der FB besitzt keine direkten Daten-Eingänge. Stattdessen werden Werte über die Adapter-Schnittstelle **PV** bereitgestellt.

Daten-Ausgänge

Der FB besitzt keine direkten Daten-Ausgänge. Die aktuellen Werte von **CV** und **Q** werden über die jeweiligen Adapter-Ports ausgegeben.

Adapter

Port	Typ	Richtung	Beschreibung
Q	`adapter::types::unidirectional::AX` (BOOL)	Plug	Statusausgang: `TRUE`, wenn **CV** ≤ **0**.
CV	`adapter::types::unidirectional::AUI` (INT)	Plug	Aktueller Zählerwert.
PV	`adapter::types::unidirectional::AUI` (INT)	Socket	Preset-Wert, der beim LD-Ereignis in **CV** geladen wird.

Funktionsweise

1. **Count Down (CD)** Jedes **CD**-Ereignis dekrementiert **CV** um 1 und setzt **Q** auf `TRUE`, falls **CV** ≤ **0** ist. Anschließend wird **CDO** ausgegeben.
2. **Load (LD)** Ein **LD**-Ereignis kopiert den Wert von **PV** nach **CV**. **Q** wird entsprechend aktualisiert (TRUE bei **CV** ≤ **0**). Danach wird **LDO** ausgegeben.
3. **On-Change-Triggering** Der FB vergleicht den aktuellen **Q**-Wert mit dem gespeicherten Vorgängerwert **Q_OLD**. Nur wenn sich **Q** geändert hat, wird das Adapter-Ereignis **Q.E1** ausgelöst. Andernfalls kehrt der FB direkt zum Startzustand zurück. Dies verhindert wiederholte identische Meldungen.
4. **PV-Änderungen** Wenn sich der **PV**-Adapter meldet (Ereignis **PV.E1**), wird **Q** neu berechnet (**UPDATE**-Algorithmus). Auch hier erfolgt die Ausgabe von **Q** nur bei einer Zustandsänderung.

Technische Besonderheiten

- **On-Change-Triggering** – Reduziert Ereignislast, indem der **Q**-Ausgang nur bei tatsächlichen Änderungen aktiviert wird.
- **Adapter-Schnittstelle** – Ermöglicht lose Kopplung und Wiederverwendung in komplexen Steuerungsstrukturen.
- **Initialwert** – **Q_OLD** wird mit `FALSE` initialisiert, sodass beim ersten Durchlauf eine Änderung erkannt wird, falls **CV** bereits ≤ 0 ist.

Zustandsübersicht

Der Ablauf wird durch folgende ECC-Zustände gesteuert:

Zustand	Beschreibung	Aktionen
START	Wartet auf eingehende Ereignisse (CD, LD, PV.E1).	–
CD	Führt den Count-Down-Algorithmus aus.	`CD (CV--, Q aktualisieren), CV.E1, CDO`
LD	Lädt **PV** in **CV**.	`LOAD (CV:=PV; Q aktualisieren), CV.E1, LDO`
UPDATE_PV	Neuberechnung von **Q** bei Änderung von **PV**.	`UPDATE (Q aus CV berechnen)`
EMIT_Q	Speichert den aktuellen **Q**-Wert und löst **Q.E1** aus.	`SAVE_Q (Q_OLD:=Q), Q.E1`

Transitionen:

- Nach **CD**, **LD** oder **UPDATE_PV** wird nur dann zu **EMIT_Q** gewechselt, wenn `Q.D1 ≠ Q_OLD`.
- Bei Gleichheit (`Q.D1 = Q_OLD`) erfolgt sofort die Rückkehr zu **START**, ohne **Q.E1** auszulösen.
- Von **EMIT_Q** wird stets zurück zu **START** gewechselt.

Anwendungsszenarien

- **Materialzähler** in Förderanlagen: Bei jedem Teil, das die Station verlässt, wird **CD** getriggert. Sobald **CV** ≤ **0**, wird eine Nachschub-Anforderung gesendet.

- **Zyklussteuerung** für wiederholte Abläufe: Der Zähler läuft von einem Startwert (geladen über **LD**) bis Null und löst dann eine Aktion aus.
- **Ereignisoptimierte Systeme** mit vielen parallelen Zählern: Dank On-Change-Triggering werden nur relevante Statusänderungen weitergemeldet.

Vergleich mit ähnlichen Bausteinen

Baustein	Besonderheit	Unterschied zu AUI_CTD
E_CTD	Standard-Abwärtszähler ohne Adapter	Verwendet direkte Daten-Ein-/Ausgänge und keine Adapter. Kein On-Change-Triggering.
E_CTUD	Aufwärts-/Abwärtszähler	Bietet beide Zählrichtungen, jedoch ohne Adapter und ohne On-Change.
AUI_CTU	Aufwärtszähler mit Adapter	Zählt aufwärts, analoge Adapter-Struktur, ebenfalls On-Change.

Der **AUI_CTD** füllt die Lücke eines spezialisierten Abwärtszählers mit moderneren Adapter-Konzepten und reduziertem Ereignisaufkommen.

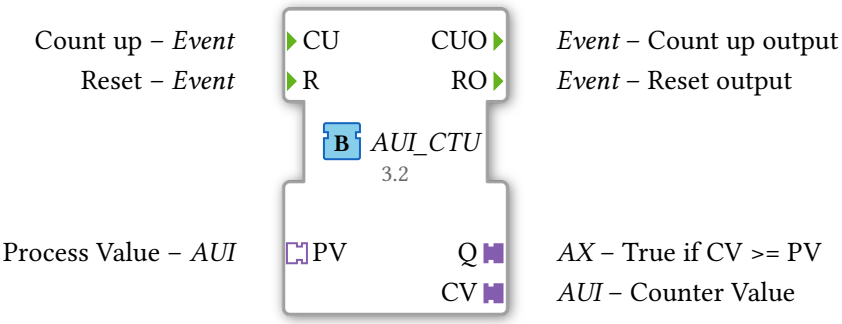
Fazit

Der **AUI_CTD** ist ein leistungsfähiger Abwärtszähler für IEC 61499-konforme Steuerungen. Durch die Kombination von Adapter-Schnittstellen und On-Change-Triggerung eignet er sich besonders für modulare, ressourcenschonende Automatisierungslösungen. Sein Verhalten ist deterministisch und kann über die ECC einfach nachvollzogen werden. Damit ist er eine optimale Wahl für Applikationen, die eine zuverlässige und effiziente Zählerfunktion mit minimierter Ereigniskommunikation erfordern.

Passende Themen-Unterseiten auf ms-muc-docs.de

- E_CTU Event Counter Baustein auf ms-muc-docs.de

AUI_CTU



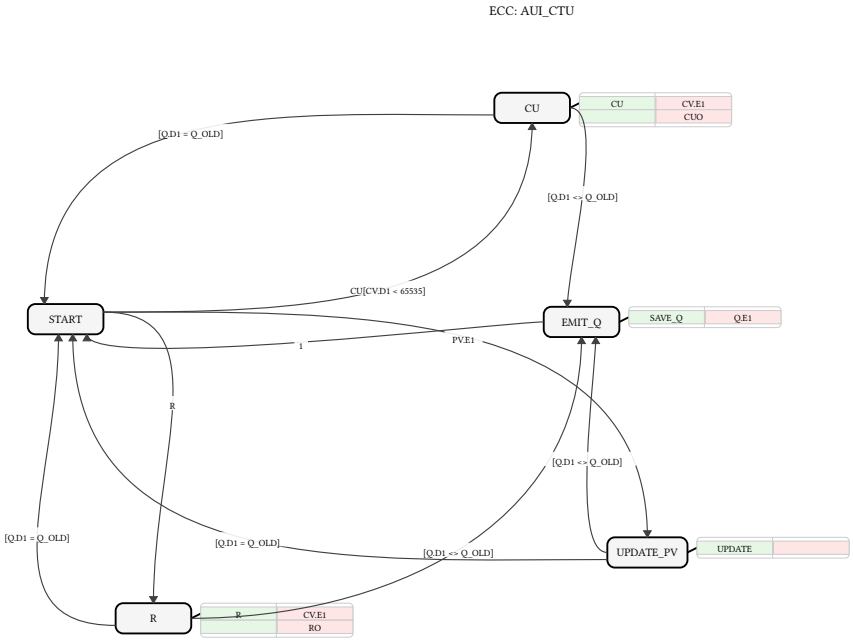


Bild des Funktionsblocks nicht verfügbar

• — ○ • — ○ • — ○ •

Einleitung

Der AUI_CTU ist ein ereignisgesteuerter Aufwärtszähler mit Adapterschnittstelle. Er zählt bei jedem positiven Flankenereignis am Eingang `CU` hoch und gibt das Zählergebnis über den Adapter `CV` aus. Die Ausgabe `Q` signalisiert, ob der Zählerstand (`CV`) den eingestellten Grenzwert (`PV`) erreicht oder überschritten hat. Das Besondere an dieser Implementierung ist die „On-Change“-Triggerung: Das Ereignis am Adapter `Q.E1` wird nur dann ausgelöst, wenn sich der logische Zustand von `Q` tatsächlich ändert. Dies reduziert unnötige Ereignisse in der nachfolgenden Verarbeitung.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
`CU`	Event	Zähle hoch (Count Up)
`R`	Event	Setze den Zähler zurück (Reset)

Ereignis-Ausgänge

Name	Typ	Kommentar
`CUO`	Event	Ausgabe nach erfolgreichem Hochzählen
`RO`	Event	Ausgabe nach erfolgreichem Rücksetzen

Daten-Eingänge

Direkte Daten-Eingänge sind nicht vorhanden. Der Grenzwert (`PV`) wird über einen Adapter-Socket bereitgestellt.

Daten-Ausgänge

Direkte Daten-Ausgänge sind nicht vorhanden. Der aktuelle Zählerwert (`CV`) und der Vergleichsstatus (`Q`) werden über Adapter-Plugs ausgegeben.

Adapter

Typ	Richtung	Name	Beschreibung
`adapter::types::unidirectional::AX`	Plug (Ausgang)	`Q`	Gibt `TRUE` aus, wenn `CV >= PV`, sonst `FALSE`. Das Ereignis `Q.E1` wird nur bei Zustandsänderung gesendet.
`adapter::types::unidirectional::AUI`	Plug (Ausgang)	`CV`	Liefert den aktuellen Zählerwert (vorzeichenloser Ganzzahl). Das Ereignis `CV.E1` wird nach jedem Hochzählen oder Rücksetzen ausgelöst.
`adapter::types::unidirectional::AUI`	Socket (Eingang)	`PV`	Empfängt den Schwellwert (Grenzwert) vom Typ `UINT`. Eine Änderung dieses Wertes führt automatisch zur Neuberechnung von `Q`.

Funktionsweise

Der Baustein implementiert eine endliche Zustandsmaschine (ECC) mit folgenden Algorithmen:

- **`CU`**: Erhöht den internen Zähler `CV` um `1` und berechnet anschließend: `Q := (CV >= PV)`.
- **`R`**: Setzt `CV` auf `0` und berechnet `Q` erneut.
- **`UPDATE`**: Berechnet `Q` aus dem aktuellen `CV` und dem (geänderten) `PV`, ohne den Zähler zu verändern.
- **`SAVE_Q`**: Speichert den aktuellen Wert von `Q` in der internen Variable `Q_OLD` für die nächste Zustandsänderungserkennung.

Ablauf:

1. **Ereignis `CU`** (und `CV < 65535`): Übergang in Zustand `CU`. Es wird der Zähler inkrementiert, `Q` neu berechnet und `CV.E1` sowie das Ereignis `CUO` ausgegeben.
   - Wenn sich `Q` gegenüber dem letztgespeicherten Wert (`Q_OLD`) geändert hat, wechselt der Zustand nach `EMIT_Q`.
   - Andernfalls kehrt der Baustein in den Zustand `START` zurück.
2. **Ereignis `R`**: Übergang in Zustand `R`. Der Zähler wird zurückgesetzt, `Q` neu berechnet und `CV.E1` sowie `RO` ausgegeben. Anschließend wird analog zu `CU` entschieden, ob `EMIT_Q` erreicht wird oder zurück nach `START`.
3. **Ereignis des Adapters `PV.E1`** (Grenzwertänderung): Übergang in Zustand `UPDATE_PV`. Der Algorithmus `UPDATE` berechnet `Q` neu. Auch hier wird nur bei Änderung von `Q` der Zustand `EMIT_Q` durchlaufen.
4. **Zustand `EMIT_Q`**: Führt den Algorithmus `SAVE_Q` aus (speichert neuen `Q`-Wert in `Q_OLD`) und sendet das Ereignis `Q.E1`. Danach kehrt der Baustein immer in `START` zurück.

Technische Besonderheiten

- **On-Change-Triggerung für Adapter `Q`**: Das Ereignis `Q.E1` wird nur dann ausgelöst, wenn sich der logische Wert von `Q` (wahr/falsch) ändert. Dies wird durch die interne Statusvariable `Q_OLD` realisiert. Dadurch entfallen unnötige Ereignisüberflutungen in der Applikation.
- **Automatische Reaktion auf Grenzwertänderung**: Ändert sich der über den Socket `PV` zugeführte Schwellwert, wird `Q` sofort neu berechnet und bei Änderung ausgegeben. Der Zähler selbst bleibt dabei unberührt.
- **Überlaufschutz**: Der Zähler wird nur inkrementiert, wenn `CV` kleiner als 65535 ist (maximaler Wert eines `UINT`). Ein weiteres Hochzählen über diesen Wert hinaus ist nicht möglich.
- **Adapterbasiert**: Statt klassischer direkter Ein-/Ausgänge werden alle Werte über standardisierte Adapter ausgetauscht. Dies ermöglicht eine einfache Integration in bestehende 4diac-Adapterkonzepte.

Zustandsübersicht

Zustand	Beschreibung	Aktionen	Ausgehende Transitionen
START	Ruhezustand, wartet auf Ereignisse	–	CU ⟶ CU, R ⟶ R, PV.E1 ⟶ UPDATE_PV
CU	Hochzählen	CU-Algorithmus, sende CV.E1 und CUO	[Q != Q_OLD] ⟶ EMIT_Q, [Q == Q_OLD] ⟶ START
R	Rücksetzen	R-Algorithmus, sende CV.E1 und RO	[Q != Q_OLD] ⟶ EMIT_Q, [Q == Q_OLD] ⟶ START
UPDATE_PV	Neuberechnung nach PV-Änderung	UPDATE-Algorithmus	[Q != Q_OLD] ⟶ EMIT_Q, [Q == Q_OLD] ⟶ START
EMIT_Q	Emission des Q-Ereignisses	SAVE_Q-Algorithmus, sende Q.E1	1 ⟶ START

Die Transitionen sind durch Bedingungen ausgelöst:

- `CU[CV.D1 < 65535]`: Ereignis CU, wenn Zähler nicht am Maximum.
- `R`: Ereignis R, immer möglich.
- `PV.E1`: Ereignis vom Adapter PV.
- `[Q.D1 <> Q_OLD]` / `[Q.D1 = Q_OLD]`: Vergleich des aktuellen Q mit dem gespeicherten vorherigen Wert.

Anwendungsszenarien

- **Ereigniszähler mit Schwellwertüberwachung**: Zählen von Impulsen (z. B. Werkstücke auf einem Förderband) und Auslösen einer Aktion, sobald ein bestimmter Grenzwert erreicht ist.
- **Pegelüberwachung**: Verwendung als Grenzwertschalter, der bei Überschreiten eines Sollwerts einen Status weitergibt.
- **Zustandsabhängige Steuerungen**: Z. B. in einer Batch-Verarbeitung, bei der nach einer bestimmten Anzahl von Schritten ein Prozessschritt ausgelöst werden soll.
- **Systeme mit dynamischem Grenzwert**: Der Grenzwert kann während des Betriebs über den Adapter `PV` geändert werden, der Baustein reagiert automatisch.

Vergleich mit ähnlichen Bausteinen

Merkmal	AUI_CTU	Standard CTU (IEC 61131-3)	CTUD (Auf-/Abwärtszähler)
Schnittstelle	Adapter-basiert	Direkte Ein-/Ausgänge	Direkte Ein-/Ausgänge
Ereignis bei Q-Änderung	Ja (On-Change)	Nein (immer bei Zählereignis)	Nein
Reaktion auf PV-Änderung	Automatisch	Nicht vorgesehen	Nicht vorgesehen
Überlaufschutz	Ja (max. 65535)	Ja, über Konfiguration	Ja
Abwärtszählen	Nein	Nein	Ja

Der `AUI_CTU` ist speziell für ereignisgesteuerte Systeme mit Adapterkonzept optimiert. Die On-Change-Triggerung und die automatische PV-Neuberechnung sind markante Unterschiede zu klassischen Zählern.

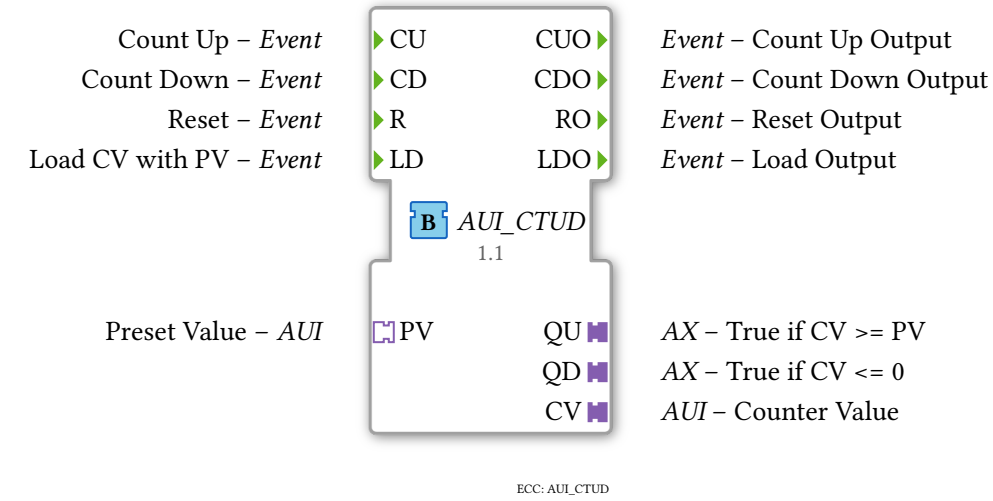
Fazit

Der `AUI_CTU` ist ein moderner, adapterbasierter Aufwärtszähler, der sich durch seine effiziente Ereignisbehandlung auszeichnet. Dank der On-Change-Triggerung für den Status `Q` werden unnötige Nachrichten im System vermieden. Die Unterstützung dynamischer Grenzwerte über den Adapter `PV` erhöht die Flexibilität in Steuerungsanwendungen. Der Baustein ist robust gegen Überläufe und bietet eine klare, zustandsgesteuerte Ablaufstruktur. Er eignet sich besonders für verteilte Automatisierungssysteme, bei denen lose Kopplung und ereignisgesteuerte Kommunikation im Vordergrund stehen.

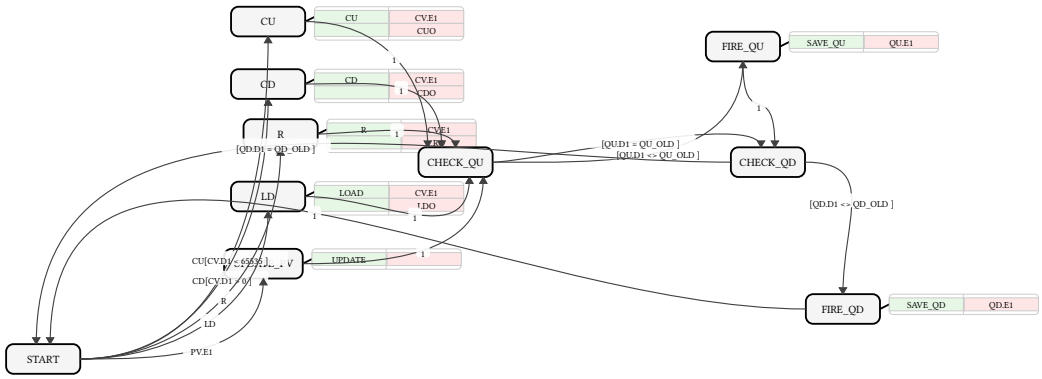
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUI_CTUD



ECC: AUI_CTUD



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUI_CTUD** ist ein ereignisgesteuerter Aufwärts-/Abwärtszähler in der Adapter-Version (Adapter Version). Er basiert auf dem Standardbaustein E_CTUD und erweitert diesen um eine „On-Change“-Triggerung für die Ausgangszustände. Über die Adapter-Schnittstellen werden der Zählerwert (CV), der Preset-Wert (PV) sowie die Vergleichsergebnisse (QU, QD) mit anderen Bausteinen ausgetauscht. Der Baustein ist für den Einsatz in IEC 61499-Anwendungen vorgesehen, bei denen eine lose Kopplung über Adapter gewünscht ist.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung
CU	Zähler um 1 erhöhen (Count Up)
CD	Zähler um 1 verringern (Count Down)
R	Zähler auf 0 zurücksetzen (Reset)
LD	Zähler mit dem Preset-Wert (PV) laden (Load)

Ereignis-Ausgänge

Ereignis	Beschreibung
CUO	Bestätigung für einen erfolgten Count-Up-Vorgang
CDO	Bestätigung für einen erfolgten Count-Down-Vorgang
RO	Bestätigung für einen erfolgten Reset
LDO	Bestätigung für einen erfolgten Load-Vorgang

Daten-Eingänge

Der Baustein besitzt keine direkten Dateneingänge. Der Preset-Wert (PV) wird über einen **Socket-Adapter** bereitgestellt:

Adapter (Socket)	Typ	Beschreibung
PV	`adapter::types::unidirectional::AUI`	Preset-Wert (Vorwahl) für den Zähler

Daten-Ausgänge

Der Baustein besitzt keine direkten Datenausgänge. Die Ergebnisse werden über **Plug-Adapter** ausgegeben:

Adapter (Plug)	Typ	Beschreibung
QU	`adapter::types::unidirectional::AX`	Wahr, wenn der aktuelle Zählerwert ≥ PV ist
QD	`adapter::types::unidirectional::AX`	Wahr, wenn der aktuelle Zählerwert ≤ 0 ist
CV	`adapter::types::unidirectional::AUI`	Aktueller Zählerwert (Unsigned Integer)

Adapter

Der Baustein definiert drei Plug-Adapter (QU, QD, CV) und einen Socket-Adapter (PV). Die Adapter sind vom Typ `adapter::types::unidirectional` und ermöglichen eine lose Kopplung zwischen den Bausteinen – die Werte werden über Ereignisse (z. B. `CV.E1`, `QU.E1`) ausgetauscht.

Funktionsweise

Der Baustein arbeitet als ereignisgesteuerter Zähler mit einer Auflösung von 0 bis 65535 (maximaler Wert für eine unsigned 16-Bit-Ganzzahl). Die folgende Schrittfolge wird durchlaufen:

1. **Eingangsereignis** (CU, CD, R oder LD) löst eine Transition vom Startzustand in den entsprechenden Verarbeitungszustand aus.
2. **Berechnung**: Im jeweiligen Algorithmus wird der interne Zähler (`CV.D1`) aktualisiert und die Ausgangsadapter QU und QD neu berechnet:
   - **CU**: `CV.D1 := CV.D1 + 1` (nur, wenn der Wert vorher < 65535)
   - **CD**: `CV.D1 := CV.D1 - 1` (nur, wenn der Wert vorher > 0)

- **R**: `CV.D1 := 0`
- **LD**: `CV.D1 := PV.D1`
- Anschließend werden `QU.D1 := (CV.D1 >= PV.D1)` und `QD.D1 := (CV.D1 <= 0)` gesetzt.

3. **Ausgabe des Zählerwerts**: Nach jeder Wertänderung wird das Ereignis `CV.E1` gesendet, um den neuen Zählerwert über den Adapter zu propagieren.
4. **On-Change-Triggering**:
   - Nach jedem Zählvorgang oder nach einer Änderung des PV (Ereignis von PV-Adapter) wird der Zustand von QU und QD überprüft.
   - Die internen Variablen `QU_OLD` und `QD_OLD` speichern den vorherigen Zustand. Nur wenn sich der Wert geändert hat, wird das entsprechende Adapter-Ereignis (`QU.E1` bzw. `QD.E1`) ausgelöst.
   - Dieses Verhalten verhindert unnötige Ereignisfluten bei gleichbleibenden Schwellwerten.
5. **Verarbeitung von PV-Änderungen**: Ein eingehendes Ereignis auf dem PV-Adapter (`PV.E1`) führt in den Zustand `UPDATE_PV`, in dem nur die Vergleichsergebnisse neu berechnet werden (ohne den Zähler zu verändern). Danach erfolgt ebenfalls die On-Change-Überprüfung.

Technische Besonderheiten

- **On-Change Triggering** (Change Detection): Der Baustein sendet Ereignisse auf den Ausgangsadaptern nur dann, wenn sich der logische Zustand im Vergleich zum letzten Durchlauf tatsächlich geändert hat. Dies reduziert die Buslast und vermeidet Endlosschleifen in zyklischen Systemen.
- **Adapterbasierte Kommunikation**: Alle Ein- und Ausgangswerte werden über Adapter (Plug/Socket) ausgetauscht, was eine flexible Verschaltung ohne feste Datenverbindungen ermöglicht. Der Zählerwert (CV) und die Vergleichsergebnisse (QU, QD) werden als Plug-Adapter bereitgestellt, der Preset-Wert (PV) wird als Socket von externen Bausteinen geliefert.
- **Zustandsüberlaufschutz**: Bei Count-Up wird geprüft, ob der aktuelle Wert kleiner als 65535 ist, bei Count-Down, ob er größer 0 ist. Dadurch wird ein Über- oder Unterlauf verhindert.
- **Zwei interne Zustandsvariablen**: `QU_OLD` und `QD_OLD` dienen als Schattenregister für die Change Detection.
- **Erweiterte ECC**: Der Baustein durchläuft nach jedem Zählvorgang eine sequentielle Kette von Zuständen (z. B. CU → CHECK_QU → CHECK_QD → START), um die Change Detection zu gewährleisten.

Zustandsübersicht

Zustand	Beschreibung
START	Warten auf ein Eingangsereignis (CU, CD, R, LD) oder eine PV-Änderung
CU	Zähler erhöhen und neuen Wert über CV-Adapter ausgeben
CD	Zähler verringern und neuen Wert über CV-Adapter ausgeben
R	Zähler auf 0 setzen und neuen Wert über CV-Adapter ausgeben
LD	Zähler mit PV laden und neuen Wert über CV-Adapter ausgeben
UPDATE_PV	Neuberechnung von QU und QD nach PV-Änderung (ohne Zähleränderung)
CHECK_QU	Prüfung, ob sich QU geändert hat (Vergleich mit QU_OLD)
FIRE_QU	Wenn QU geändert: QU_OLD aktualisieren und Ereignis auf QU-Adapter senden
CHECK_QD	Prüfung, ob sich QD geändert hat (Vergleich mit QD_OLD)
FIRE_QD	Wenn QD geändert: QD_OLD aktualisieren und Ereignis auf QD-Adapter senden

Die Zustandsübergänge werden durch die Ereignisse und die Bedingungen `[QU.D1 <> QU_OLD]`, `[QD.D1 <> QD_OLD]` gesteuert. Nach Abschluss der Change Detection kehrt der Baustein in den START-Zustand zurück.

Anwendungsszenarien

- **Zähler mit externem Preset**: In einer Anlage kann der Preset-Wert (PV) von einem übergeordneten System oder einem HMI über den PV-Adapter dynamisch gesetzt werden. Der Baustein zählt daraufhin Ereignisse und meldet das Erreichen oder Unterschreiten der Schwellen über die QU- und QD-Adapter.

- **Richtungsabhängige Zähler**: Durch Verwendung von CU und CD kann ein Vorwärts-/Rückwärtszähler realisiert werden, z. B. für Positionserfassung oder Bestandszählung.
- **Ereignisgesteuerte Grenzwertüberwachung**: Die On-Change-Triggerung von QU und QD eignet sich ideal, um ein diskretes Signal (z. B. „Füllstand erreicht“) nur bei Zustandswechsel zu senden – ähnlich einem Flanken-Detektor.
- **Modulare Steuerungsarchitektur**: Dank der Adapter-Schnittstelle kann der Baustein ohne feste Datenport-Verkabelung in eine vorhandene IEC 61499-Applikation integriert werden. Die logische Verbindung erfolgt allein über die Adapter-Laufzeitumgebung.

Vergleich mit ähnlichen Bausteinen

Der Standardbaustein **E_CTUD** (aus der IEC 61499-Bibliothek) bietet ebenfalls einen Aufwärts-/Abwärtszähler mit Ereignissteuerung, jedoch:

Merkmal	**E_CTUD** (Standard)	**AUI_CTUD** (Adapter-Version)
Schnittstelle	Feste Event- und Daten-Ports (z. B. CV als BOOL)	Lose Kopplung über Adapter (Plug/Socket)
Ausgabe Zählerwert	Datenport CV (INT/UDINT)	Adapter CV (Typ AUI)
Grenzwertausgabe	Bool-Ports QU, QD	Adapter QU, QD (Typ AX)
Ereignis bei Änderung	Keine Change Detection; QU/QD werden bei jedem Zählvorgang berechnet	On-Change-Triggerung: Ereignis nur bei Zustandswechsel
Rückmeldung über Aktionen	Keine eigenen Rücksignal-Ereignisse	CUO, CDO, RO, LDO als Bestätigungsereignisse

Der AUI_CTUD ist daher flexibler in der Verschaltung und reduziert die Ereignislast in verteilten Systemen. Für einfache Anwendungen ohne Adapter kann jedoch der Standard-E_CTUD ausreichend sein.

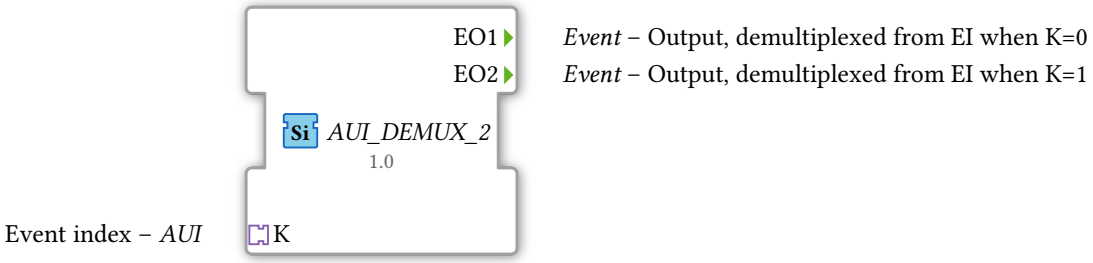
Fazit

Der Funktionsblock **AUI_CTUD** stellt eine moderne, adaptergestützte Variante eines Aufwärts-/Abwärtszählers dar. Besonders hervorzuheben sind die On-Change-Triggerung der Grenzwerte und die Rückmeldung über eigene Ereignisausgänge. Durch die Verwendung von Adapter-Schnittstellen wird eine lose Kopplung und Wiederverwendbarkeit in unterschiedlichsten Steuerungsprojekten ermöglicht. Die integrierten Schutzmaßnahmen gegen Über-/Unterlauf sowie die saubere Trennung von Zähl- und Auswerte-Logik machen ihn zu einem robusten Baustein für industrielle Automatisierungsanwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de

AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AUI_DEMUX_2** ist ein Ereignis-Demultiplexer mit zwei Ausgängen. Er verteilt eingehende Ereignisse anhand eines über einen Adapter gelieferten Index auf einen der beiden Ausgänge. Im Gegensatz zu dem klassischen Baustein `E_DEMUX_2` wird die Steuerinformation nicht über einen

separaten Dateneingang, sondern über einen **AUI-Adapter** (Socket) bereitgestellt. Dadurch ist eine entkoppelte und flexiblere Anbindung an andere Bausteine möglich.

Schnittstellenstruktur

Ereignis-Eingänge
Keine vorhanden.

Ereignis-Ausgänge

Name	Datentyp	Kommentar
`EO1`	`Event`	Ausgang für Index `0`
`EO2`	`Event`	Ausgang für Index `1`

Daten-Eingänge
Keine vorhanden.

Daten-Ausgänge
Keine vorhanden.

Adapter

Name	Richtung	Typ	Kommentar
`K`	Socket	`adapter::types::unidirectional::AUI`	Liefert den Index zur Demultiplexierung

Funktionsweise
Der Baustein wartet auf ein Ereignis, das über den Adapter `K` eintrifft. Der Adapter übermittelt dabei zusätzlich einen Wert (typischerweise 0 oder 1), der als Demultiplex-Index interpretiert wird.

- Bei Index `0` wird der Ereignisausgang `EO1` ausgelöst.
- Bei Index `1` wird der Ereignisausgang `EO2` ausgelöst.

Die Verarbeitung erfolgt rein ereignisgesteuert; es werden keine Daten zwischengespeichert oder weitergeleitet. Der Baustein verhält sich wie ein 1-zu-2-Demultiplexer, wobei die Steuerung über eine standardisierte Adapterschnittstelle erfolgt.

Technische Besonderheiten
- **Adapterbasierte Steuerung**: Statt eines separaten Dateneingangs wird der Index über einen AUI-Adapter bereitgestellt. Dies ermöglicht eine lose Kopplung und vereinfacht die Wiederverwendung in unterschiedlichen Kontexten.
- **Generische Struktur**: Der Baustein ist als generischer Funktionsblock (`GEN_E_DEMUX`) gekennzeichnet und kann je nach Konfiguration auch mit anderen Adaptertypen arbeiten – im vorliegenden Fall ist der Typ `AUI` festgelegt.
- **Keine Datenübertragung**: Der Baustein überträgt keine Nutzdaten, nur Ereignisse werden weitergegeben.

Zustandsübersicht
Der Baustein besitzt **keine internen Zustände**. Er ist ereignisgetrieben und reagiert ausschließlich auf das Eintreffen eines Ereignisses am Adapter. Die Auswahl des Ausgangs erfolgt deterministisch anhand des empfangenen Index.

Anwendungsszenarien
- **Ereignisverteilung in Automatisierungssystemen**: Wenn eine übergeordnete Steuerung ein Ereignis basierend auf einem Auswahlwert an verschiedene Verarbeitungszweige weiterleiten muss.
- **Modulare Architekturen**: Wenn die Steuerinformation über einen Adapter aus einer externen Quelle (z. B. einer Datenbank oder einem Kommunikationsmodul) stammt und eine flexible Anbindung gewünscht ist.

- **Sichere Ereignisweiche**: Einsatz als einfache Weiche, die ein Ereignis nur an einen von zwei möglichen Empfängern sendet.

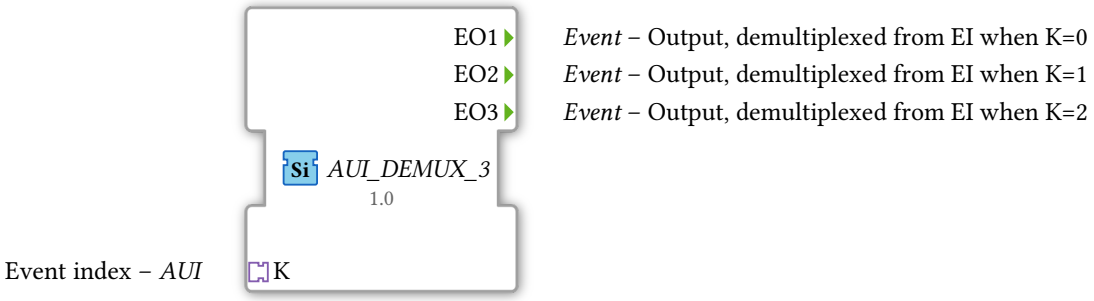
Vergleich mit ähnlichen Bausteinen

Der klassische `E_DEMUX_2` besitzt einen separaten Ereigniseingang (`EI`) und einen Dateneingang (`K`). Der vorliegende Baustein `AUI_DEMUX_2` ersetzt diese beiden Eingänge durch einen einzelnen Adapter. Vorteile sind eine reduzierte Anzahl an Verbindungen und eine klarere Schnittstellenabstraktion. Nachteilig könnte die Abhängigkeit von einem spezifischen Adaptertyp sein, falls keine generische Konfiguration vorgenommen wird.

Fazit

Der `AUI_DEMUX_2` ist ein praktischer Ereignis-Demultiplexer, der sich durch seine adapterbasierte Steuerung besonders für modulare und serviceorientierte Architekturen eignet. Er reduziert die Verdrahtung und erhöht die Flexibilität gegenüber klassischen Demultiplexern, ohne dabei an Funktionalität einzubüßen. Seine einfache, ereignisgesteuerte Struktur macht ihn zu einem vielseitigen Werkzeug in der industriellen Automatisierungstechnik.

AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUI_DEMUX_3** ist ein ereignisbasierter Demultiplexer, der über einen AUI-Adapter (Application-to-Unification Interface) einen Index empfängt und das ankommende Ereignis auf einen von drei Ausgängen weiterleitet. Er stellt eine kompakte Variante des klassischen `E_DEMUX_3` dar, bei dem der Ereignis-Eingang und der Index-Eingang in einem einzigen AUI-Adapter zusammengefasst sind. Dadurch vereinfacht sich die Verdrahtung in Systemen, die bereits AUI-Kommunikation verwenden.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine vorhanden. Das auslösende Ereignis wird über den Adapter `K` übertragen.

Ereignis-Ausgänge

- `EO1` (Typ: Event) – Ausgang, der aktiviert wird, wenn der Index den Wert 0 hat.
- `EO2` (Typ: Event) – Ausgang, der aktiviert wird, wenn der Index den Wert 1 hat.
- `EO3` (Typ: Event) – Ausgang, der aktiviert wird, wenn der Index den Wert 2 hat.

Daten-Eingänge

- Keine vorhanden.

Daten-Ausgänge

- Keine vorhanden.

Adapter

- `K` (Socket, Typ: `adapter::types::unidirectional::AUI`) – Ein unidirektionaler AUI-Adapter, der den Index (z.B. als Ganzzahl) liefert und gleichzeitig das Ereignis transportiert, das den Demultiplexer triggert.

Funktionsweise

Der Funktionsblock wartet auf ein Ereignis, das über den AUI-Adapter `K` ankommt. Zeitgleich wird über denselben Adapter ein Indexwert (0, 1 oder 2) bereitgestellt. Sobald das Ereignis eintrifft, wird es je nach Indexwert an den entsprechenden Ausgang (`EO1`, `EO2` oder `EO3`) weitergeleitet. Ist der Index ungleich 0, 1 oder 2, wird das Ereignis verworfen (der FB reagiert nicht). Die Verarbeitung erfolgt ereignisgesteuert ohne interne Zustandslogik; jeder ankommende Event wird sofort verarbeitet.

Technische Besonderheiten

- **Adapter-basierte Schnittstelle**: Der FB nutzt einen AUI-Adapter anstelle separater Event- und Daten-Eingänge, was die Anzahl der Verbindungen reduziert und die Integration in adapterorientierte Architekturen erleichtert.
- **Generische Erweiterbarkeit**: Durch das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `'GEN_E_DEMUX'` kann der FB generisch für andere Ausgangsanzahlen (z.B. `AUI_DEMUX_N`) angepasst werden.
- **Kompakte Parametrierung**: Der Adapter übernimmt sowohl die Ereignis- als auch die Datenübertragung, wodurch eine klare Trennung von Steuerfluss und Datenfluss entfällt.
- **Keine internen Zustände**: Der FB ist vollständig ereignisgetrieben und benötigt keine Zustandsverwaltung.

Zustandsübersicht

Es existiert kein expliziter interner Zustand. Der FB verhält sich wie ein reiner Multiplexer: Bei jedem eingehenden Ereignis wird der aktuelle Index ausgewertet und das Ereignis direkt an einen Ausgang weitergegeben. Damit ist der FB deterministisch und benötigt keine Initialisierung oder Reset.

Anwendungsszenarien

- **Routing von Ereignissen**: In Automatisierungssystemen, wenn ein einzelnes Ereignis (z.B. eine Messung oder ein Alarm) je nach Kontext verschiedene nachgelagerte Verarbeitungspfade auslösen soll.
- **Adapterorientierte Systeme**: Überall dort, wo bereits eine AUI-basierte Kommunikation verwendet wird und Ereignisse mit einem Index (z.B. einer Priorität oder einer Betriebsart) verbunden sind.
- **Vereinfachung von Verschaltungen**: Reduziert die Anzahl der Verbindungen gegenüber klassischen Demultiplexern und verbessert die Lesbarkeit großer Steuerungslogiken.

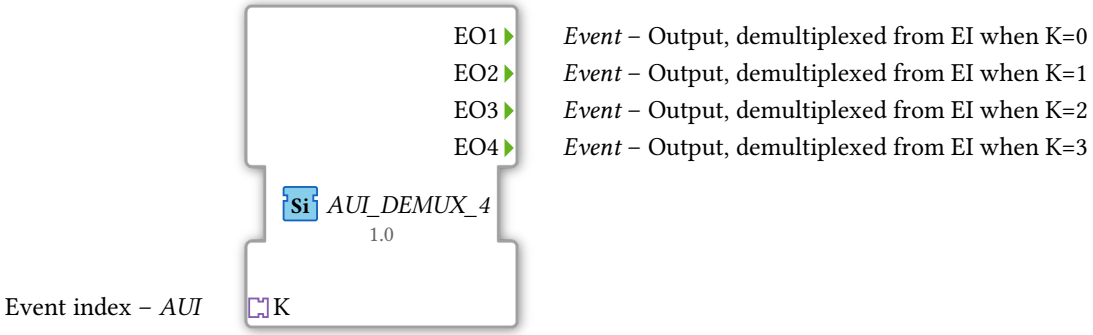
Vergleich mit ähnlichen Bausteinen

- **E_DEMUX_3** (klassisch): Verfügt über getrennte Eingänge für das Ereignis (`EI`) und den Index (`K`). Erfordert zwei separate Verdrahtungen, ist aber in Standardumgebungen weiter verbreitet.
- **AUI_DEMUX_3** (dieser FB): Bündelt beide Funktionen in einem AUI-Adapter. Dies reduziert die Anzahl der Ports und macht die Verbindung kompakter, erfordert jedoch eine passende AUI-Infrastruktur.
- **Weitere Demultiplexer**: Z.B. `E_DEMUX` mit variabler Ausgangsanzahl. Die AUI-Variante bietet den Vorteil einer einheitlichen Schnittstellenbeschreibung und erleichtert die Wiederverwendung von Adaptertypen.

Fazit

Der Funktionsblock `AUI_DEMUX_3` ist eine praktische und moderne Alternative zum klassischen `E_DEMUX_3`, die durch die Verwendung eines AUI-Adapters eine kompakte und klar strukturierte Schnittstelle bietet. Er eignet sich besonders für Systeme, die bereits mit AUI-Adaptern arbeiten und eine flexible Ereignisweiterleitung benötigen. Die generische Auslegung ermöglicht eine einfache Erweiterung auf andere Ausgangsanzahlen, wodurch der Baustein in einer Vielzahl von Steuerungs- und Automatisierungsanwendungen eingesetzt werden kann.

AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AUI_DEMUX_4** ist ein Ereignis-Demultiplexer mit vier Ausgängen. Er empfängt ein Ereignis über einen AUI-Adapter (Application Interconnection Unit) und leitet es – abhängig von einem über denselben Adapter übermittelten Index – an einen der vier Ereignis-Ausgänge weiter. Im Gegensatz zum klassischen `E_DEMUX`, bei dem Ereignis (`EI`) und Index (`K`) als getrennte Schnittstellen vorliegen, werden hier beide Informationen über einen unidirektionalen Adapter gebündelt. Dadurch vereinfacht sich die Verdrahtung in komplexen Systemen und die Schnittstelle wird kompakter.

Der Baustein ist als generischer Funktionsbaustein (`GEN_E_DEMUX`) implementiert und kann bei Bedarf auf andere Kanalzahlen erweitert werden.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt **keine** konventionellen Ereignis-Eingänge. Ereignisse werden ausschließlich über den Adapter `K` empfangen, der das Ereignis und den Demultiplex-Index in einem einzigen logischen Kanal bündelt.

Ereignis-Ausgänge

Name	Typ	Kommentar
EO1	Event	Ausgang, demultiplext von K, wenn Index = 0
EO2	Event	Ausgang, demultiplext von K, wenn Index = 1
EO3	Event	Ausgang, demultiplext von K, wenn Index = 2
EO4	Event	Ausgang, demultiplext von K, wenn Index = 3

Daten-Eingänge

Es sind keine separaten Daten-Eingänge vorhanden. Der Index wird über den Adapter transportiert.

Daten-Ausgänge

Es sind keine Daten-Ausgänge vorhanden.

Adapter

Name	Typ	Kommentar
K	`adapter::types::unidirectional::AUI`	Ereignis-Index (Demultiplex)

Der Adapter ist ein **Socket** und erwartet einen unidirektionalen Datenfluss von einem Plug, der sowohl das Ereignis als auch den Indexwert bereitstellt.

Funktionsweise

Der Baustein operiert rein ereignisgesteuert. Sobald über den Adapter `K` ein Ereignis eintrifft, wird der im Adapter enthaltene Indexwert ausgewertet. Abhängig von diesem Index wird das Ereignis an genau einen

der vier Ausgänge (`E01`, `E02`, `E03` oder `E04`) weitergeleitet. Der Index kann Werte von 0 bis 3 annehmen, wobei gilt:

- Index 0 ⟶ Ereignis an `E01`
- Index 1 ⟶ Ereignis an `E02`
- Index 2 ⟶ Ereignis an `E03`
- Index 3 ⟶ Ereignis an `E04`

Die Weiterleitung erfolgt synchron zum eintreffenden Ereignis; es findet keine Zwischenspeicherung oder Warteschlangenbildung statt. Der Baustein ist damit ein klassischer 1-zu-4-Demultiplexer auf Ereignisebene.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Die Verwendung des AUI-Adapters kapselt Ereignis und Index in einem einzigen Kanal. Dies reduziert die Anzahl der Verdrahtungsverbindungen und erhöht die Modularität.
- **Generische Implementierung**: Der Baustein ist als Instanz des generischen Bausteins `GEN_E_DEMUX` definiert. Dadurch kann er – je nach Konfiguration – auch für andere Kanalzahlen (z. B. 2, 8, 16) eingesetzt werden, ohne die interne Logik neu implementieren zu müssen.
- **Unidirektionaler Adapter**: Der Adapter `K` ist als Socket vom Typ `adapter::types::unidirectional::AUI` ausgeführt. Er erwartet ausschließlich eingehende Daten und stellt keine Rückkanal zur Verfügung.
- **Keine Datenpfade**: Der Baustein verarbeitet ausschließlich Ereignisse und Indizes; es werden keine Nutzdaten (z. B. Werte) durchgereicht.

Zustandsübersicht

Der Baustein besitzt keinen persistenten internen Zustand. Er verhält sich rein kombinatorisch (ereignisgesteuert) und reagiert unmittelbar auf jedes eintreffende Ereignis am Adapter. Es gibt keine Initialisierungs- oder Wartezustände. Die Funktionsweise lässt sich als ereignisbasierte Verzweigung beschreiben, die ohne Speicherung von Zuständen arbeitet.

Anwendungsszenarien

- **Verteilen von Ereignissen in Steuerungen**: In einer Automatisierungsumgebung können Ereignisse (z. B. Startsignale, Messimpulse) je nach einem Indexwert an verschiedene Aktoren oder Teilnehmer weitergeleitet werden.
- **Modulare Aufrufsteuerung**: In SPS-Programmen kann der Baustein verwendet werden, um verschiedene Funktionsbausteine selektiv zu aktivieren, ohne mehrere parallele Verbindungen aufzubauen.
- **Datenfluss-Management**: In Kommunikationsprotokollen kann ein Ereignis mit einem Adress- oder Kanalindex versehen werden und so an die entsprechende Verarbeitungseinheit geroutet werden.
- **Basis für erweiterte Demultiplexer**: Durch Anpassung der Kanalzahl (generische Variante) lässt sich der Baustein für Systeme mit mehr oder weniger Ausgängen skalieren.

Vergleich mit ähnlichen Bausteinen

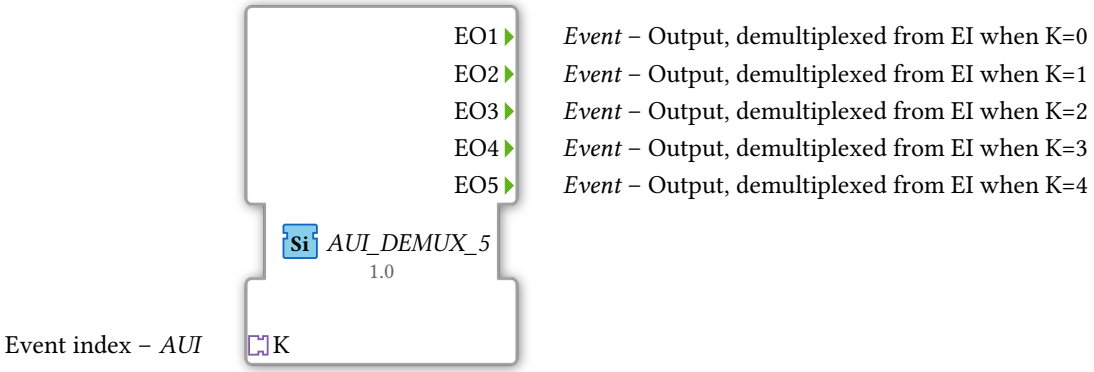
Baustein	Beschreibung
`E_DEMUX`	Klassischer Demultiplexer mit separatem Ereignis-Eingang `EI` und Index-Eingang `K` (Daten). Kein Adapter.
`AUI_DEMUX_4`	Adapterbasierte Variante, wie hier beschrieben.
`GEN_E_DEMUX`	Generischer Basisbaustein, der je nach Parameter unterschiedliche Kanalzahlen (2, 4, 8, …) unterstützt. `AUI_DEMUX_4` ist eine instanziierte Ausprägung.
`E_MUX` (Ereignis-Multiplexer)	Führt mehrere Ereignisse zu einem Ausgang zusammen – Gegenrichtung zum Demultiplexer.

Der wesentliche Unterschied zu `E_DEMUX` ist die Kapselung von Ereignis und Index über den AUI-Adapter. Dadurch entfällt die separate Datenleitung für den Index und die Schnittstelle wird abgeschlossen und typsicher. Im Vergleich zum generischen Baustein bietet `AUI_DEMUX_4` eine feste Kanalzahl von 4, wodurch die Schnittstelle im Projekt klar dokumentiert ist.

Fazit

Der `AUI_DEMUX_4` ist ein kompakter, ereignisbasierter Demultiplexer, der durch die Verwendung eines AUI-Adapters die Schnittstelle moderner und modularer gestaltet als klassische Demultiplexer-Bausteine. Er eignet sich hervorragend für verteilte Automatisierungssysteme, in denen eine saubere Trennung von Ereignis- und Indexübertragung gefordert ist. Durch seine generische Basis lässt er sich außerdem leicht auf andere Kanalzahlen erweitern, was die Wiederverwendbarkeit erhöht. Die Implementierung ist bewusst schlank gehalten und verzichtet auf unnötige Datenpfade, wodurch sie sich ideal für ereignisgesteuerte Steuerungsanwendungen eignet.

AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUI_DEMUX_5** ist ein ereignisbasierter Demultiplexer mit fünf Ausgängen. Er empfängt ein Ereignis sowie einen Auswahlindex über einen AUI-Adapter (unidirektional) und leitet das Ereignis an genau einen der fünf Ereignis-Ausgänge weiter, abhängig vom Wert des Index. Der Baustein stellt eine Adapter-basierte Variante des klassischen **E_DEMUX_5** dar und ist generisch implementiert (Generic-Class **GEN_E_DEMUX**). Dadurch kann er flexibel in verschiedenen Kontexten eingesetzt werden, in denen eine Demultiplexierung von Ereignissen über eine standardisierte Adapterschnittstelle erforderlich ist.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine expliziten Ereignis-Eingänge. Das eingehende Ereignis wird ausschließlich über den AUI-Adapter (Socket **K**) transportiert, der sowohl das Ereignis als auch den Auswahlindex (als Dateninhalt des Adapters) übermittelt.

Ereignis-Ausgänge

Ausgang	Datentyp	Kommentar
EO1	Event	Demultiplextes Ereignis, wenn `K = 0`
EO2	Event	Demultiplextes Ereignis, wenn `K = 1`
EO3	Event	Demultiplextes Ereignis, wenn `K = 2`
EO4	Event	Demultiplextes Ereignis, wenn `K = 3`
EO5	Event	Demultiplextes Ereignis, wenn `K = 4`

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Typ	Richtung	Name	Kommentar
`adapter::types::unidirectional::AUI`	Socket	**K**	Ereignisindex (0–4) zur Auswahl des Ausgangs

Der Adapter **K** ist unidirektional und liefert sowohl das auslösende Ereignis als auch den numerischen Index (`0` bis `4`) für die Demultiplexierung.

Funktionsweise

Der **AUI_DEMUX_5** arbeitet nach dem Prinzip eines Ereignis-Demultiplexers:

1. Über den AUI-Adapter **K** wird ein Ereignis empfangen. Der Adapter übermittelt zusätzlich einen Wert (Index) im Bereich 0–4.
2. Der Baustein wertet den empfangenen Index aus.
3. Abhängig vom Index wird das eingehende Ereignis an genau einen der fünf Ausgänge weitergeleitet:
   - `K = 0` ⟶ Ereignis an `EO1`
   - `K = 1` ⟶ Ereignis an `EO2`
   - `K = 2` ⟶ Ereignis an `EO3`
   - `K = 3` ⟶ Ereignis an `EO4`
   - `K = 4` ⟶ Ereignis an `EO5`

Alle anderen Ausgänge bleiben währenddessen inaktiv. Das Ereignis wird nicht verzögert oder verändert weitergegeben; es handelt sich um eine reine Weiterleitung an den selektierten Ausgang.

Technische Besonderheiten

- **Adapter-basiert:** Statt herkömmlicher Einzel-Eingänge (Ereignis + Dateneingang für Index) verwendet der Baustein einen AUI-Adapter (unidirektional), was die Integration in moderne, adapterorientierte Systeme erleichtert.
- **Generische Implementierung:** Der Baustein ist als generische Klasse (GenericClassName `GEN_E_DEMUX`) im System hinterlegt. Dadurch kann er in verschiedenen Kontexten mit unterschiedlichen Adaptertypen wiederverwendet werden, ohne dass der Quellcode angepasst werden muss.
- **Keine Datenpfade:** Der Baustein verarbeitet ausschließlich Ereignisse; es werden keine Daten weitergeleitet oder transformiert.
- **Unidirektionaler Adapter:** Der Socket ist als unidirektionaler AUI definiert, d.h. er sendet nur Informationen in den Baustein hinein (Ereignis + Index), nicht nach außen.

Zustandsübersicht

Der Baustein besitzt keinen persistenten internen Zustand. Er ist rein funktional: Bei jedem eingehenden Ereignis am Adapter wird unmittelbar der entsprechende Ausgang aktiviert. Sobald das Ereignis verarbeitet ist, kehrt der Baustein in einen passiven Ruhezustand zurück, bis das nächste Ereignis eintrifft. Eine explizite Zustandsmaschine ist daher nicht erforderlich.

Anwendungsszenarien

- **Ereignisweiche in Steuerungssystemen:** Verteilung eines zentralen Ereignisses an verschiedene Verarbeitungseinheiten abhängig von einem Prioritäts- oder Modusindikator.
- **Multiplex-Kommunikation:** Einsatz in Systemen, in denen über einen gemeinsamen Adapterkanal Ereignisse mit einer Kennung gesendet werden, die eine gezielte Weiterleitung an verschiedene Funktionsblöcke ermöglicht.
- **Modulare Architekturen:** Nutzung als Schnittstellenbaustein in prozessnahen oder automatisierten Anlagen, wo eine standardisierte Adapterverbindung die Verdrahtung zwischen Komponenten vereinfacht.

Vergleich mit ähnlichen Bausteinen

Baustein	Unterschied
E_DEMUX_5	Klassischer Demultiplexer mit separatem Ereignis-Eingang `EI` und Dateneingang `K` (INT). Verwendet keine Adapter.

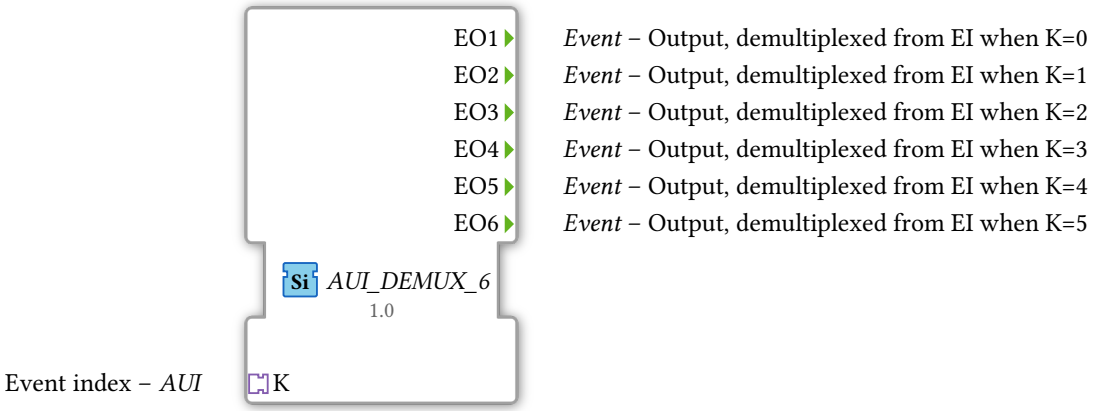
GEN_E_DEMUX	Generische Basis, die je nach Konfiguration unterschiedliche Schnittstellen (auch Adapter) aufweisen kann. `AUI_DEMUX_5` ist eine konkrete Instanziierung mit AUI.
AUI_DEMUX_8 (analog)	Variante mit acht Ausgängen; Skalierung des Indexbereichs.

Gegenüber dem klassischen **E_DEMUX_5** bietet die Adaptervariante eine einheitliche, gekapselte Schnittstelle, die in Adapter-basierten Architekturen einfacher zu integrieren ist. Dafür entfällt die Möglichkeit, das Ereignis und den Index getrennt über verschiedene Eingänge zu speisen.

Fazit

Der **AUI_DEMUX_5** ist ein kompakter, adapterbasierter Ereignis-Demultiplexer für fünf Ausgänge. Er eignet sich besonders für moderne Systeme, die eine standardisierte unidirektionale Adapterkommunikation verwenden. Durch seine generische Implementierung und die klare Funktionsweise ist er flexibel einsetzbar und bildet eine zuverlässige Grundlage für Ereignisverteilung in verteilten Automatisierungsumgebungen.

AUI_DEMUX_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUI_DEMUX_6** ist ein Ereignis-Demultiplexer mit sechs Ausgängen. Er stellt eine spezielle Variante des generischen Bausteins `GEN_E_DEMUX` dar, bei dem der übliche separate Ereignis-Eingang (EI) zusammen mit dem Daten-Eingang (K) durch einen einzigen AUI-Adapter (Socket) ersetzt wurde. Über diesen Adapter wird ein Ereignis sowie eine Indexinformation übertragen, anhand derer das Ereignis an genau einen der sechs Ausgänge weitergeleitet wird.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock besitzt **keine direkten Ereignis-Eingänge**. Das Eingangsereignis wird stattdessen über den Adapter `K` empfangen. Der Adapter trägt hierbei sowohl das eigentliche Ereignissignal als auch den zugehörigen Indexwert.

Ereignis-Ausgänge

- **EO1** – Ereignisausgang, aktiv wenn der empfangene Index = 0 ist
- **EO2** – Ereignisausgang, aktiv wenn der empfangene Index = 1 ist
- **EO3** – Ereignisausgang, aktiv wenn der empfangene Index = 2 ist
- **EO4** – Ereignisausgang, aktiv wenn der empfangene Index = 3 ist
- **EO5** – Ereignisausgang, aktiv wenn der empfangene Index = 4 ist
- **EO6** – Ereignisausgang, aktiv wenn der empfangene Index = 5 ist

Daten-Eingänge

Es sind keine separate Daten-Eingänge vorhanden. Alle benötigten Daten (insbesondere der Indexwert) werden über den Adapter bereitgestellt.

Daten-Ausgänge

Keine Daten-Ausgänge spezifiziert.

Adapter

- **K** (Socket) – Typ: `adapter::types::unidirectional::AUI`
  Über diesen Adapter empfängt der Baustein das Ereignis und den zugehörigen Indexwert. Der Adapter ist als unidirektionale Schnittstelle ausgelegt und stellt die Verbindung zu einem entsprechenden Plug eines anderen Bausteins her.

Funktionsweise

Der Baustein wartet auf ein Ereignis, das über den Adapter `K` eintrifft. Zusammen mit diesem Ereignis wird ein ganzzahliger Indexwert (im Bereich 0 bis 5) über denselben Adapter übertragen. Sobald ein Ereignis empfangen wird, bestimmt der FB anhand des Indexwerts, welcher der sechs Ausgänge `EO1` bis `EO6` aktiviert wird. Das Ereignis wird also dem entsprechenden Ausgang zugeordnet und dort als Ausgangsereignis weitergegeben.

Der Ablauf ist rein ereignisgesteuert und besitzt keine internen Zustände. Die Zuordnung ist deterministisch: Index 0 ⟶ `EO1`, Index 1 ⟶ `EO2`, … Index 5 ⟶ `EO6`.

Technische Besonderheiten

- **Adapterbasierte Eingangsschnittstelle:** Durch die Verwendung des AUI-Adapters wird der klassische Aufbau (separater Ereignis-Eingang und Daten-Eingang) zu einer einzigen, sauber gekapselten Schnittstelle zusammengefasst.
- **Generischer Baustein:** Über das Attribut `eclipse4diac::core::GenericClassName` wird der FB als spezielle Implementierung des generischen Typs `GEN_E_DEMUX` gekennzeichnet. Dadurch kann er in generischen Anwendungen verwendet werden.
- **Unidirektionale Kommunikation:** Der Adapter ist unidirektional ausgelegt, d.h. die Datenübertragung erfolgt ausschließlich vom Sender zum Empfänger.
- **Keine Datenhaltung:** Der Baustein ist stateless und benötigt keine internen Variablen, was eine einfache und robuste Wiederverwendung ermöglicht.

Zustandsübersicht

Da der Funktionsblock keine internen Zustände besitzt, existiert keine Zustandsmaschine. Die Funktionalität beschränkt sich auf eine reine Ereignis-zu-Ereignis-Umschaltung, die ausschließlich vom aktuellen Indexwert abhängt.

Anwendungsszenarien

- **Ereignisverteilung in Automatisierungssystemen:** Wenn in einer Steuerung ein Ereignis (z.B. ein Alarmsignal) in Abhängigkeit eines Prioritäts- oder Klassencodes an verschiedene Verarbeitungszweige weitergeleitet werden soll.
- **Modulare Signalweiche:** Einsatz in Systemen, die über Adapter kommunizieren, um lose gekoppelte Komponenten zu verbinden und gleichzeitig die Anzahl der Leitungen zu reduzieren.
- **Basis für eigene Demultiplexer-Erweiterungen:** Der Baustein kann als Referenz für ähnliche Adapter-basierte Varianten dienen, z.B. für eine andere Anzahl von Ausgängen oder andere Adaptertypen.

Vergleich mit ähnlichen Bausteinen

Der Standard-Funktionsblock `E_DEMUX` verwendet einen separaten Ereignis-Eingang (`EI`) und einen Daten-Eingang (`K`) zur Auswahl des Ausgangs. Der `AUI_DEMUX_6` ersetzt diese beiden Eingänge durch einen einzigen AUI-Adapter, der sowohl das Ereignis als auch den Index überträgt.

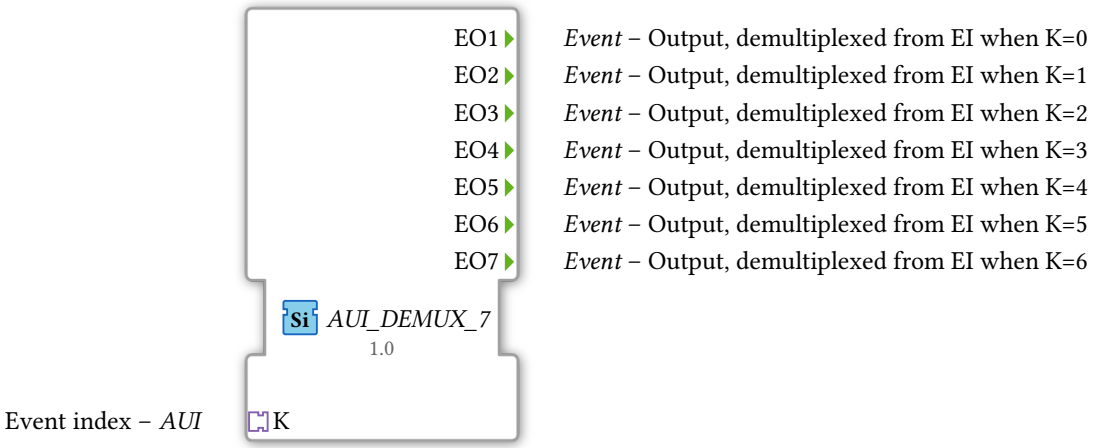
Eigenschaft	E_DEMUX	AUI_DEMUX_6
Ereignis-Eingang	Ja (EI)	Nein, über Adapter
Daten-Eingang	Ja (K)	Nein, über Adapter
Kommunikationsart	Direkt verdrahtet	Über Adapter (AUI)
Modularität	geringer	höher (via Plug/Socket)
Verdrahtungsaufwand	höher	geringer

Der Vorteil der Adapter-Variante liegt in der verbesserten Modularität und Wiederverwendbarkeit. Der Nachteil besteht darin, dass der empfangende Baustein an eine spezifische Adapter-Schnittstelle gebunden ist, was die Kompatibilität mit anderen Bausteinen einschränken kann.

Fazit

Der Funktionsblock `AUI_DEMUX_6` ist eine moderne, adapterbasierte Realisierung eines Ereignis-Demultiplexers für sechs Ausgänge. Er vereinfacht die Verkabelung, erhöht die Modularität und eignet sich besonders für Systeme, die auf der 4diac-Adaptertechnik basieren. Durch seine generische Natur lässt er sich leicht an unterschiedliche Anforderungen anpassen und stellt eine nützliche Erweiterung der Standard-Demultiplexer dar.

AUI_DEMUX_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AUI_DEMUX_7 ist ein generischer Event-Demultiplexer mit sieben Ausgängen. Er leitet ein eingehendes Ereignis (EI) abhängig von einem über einen AUI-Adapter bereitgestellten Index (K) an einen der sieben Ereignisausgänge weiter. Dieser Baustein stellt eine Alternative zum klassischen E_DEMUX_7 dar, bei dem der Index über einen einfachen Dateneingang übergeben wird, während hier eine adapterbasierte Verbindung verwendet wird.

Schnittstellenstruktur

Ereignis-Eingänge

Es ist kein Ereignis-Eingang in der XML-Schnittstelle explizit deklariert. Als generischer Funktionsbaustein (GenericClassName: `GEN_E_DEMUX`) besitzt der Baustein jedoch einen impliziten Ereigniseingang `EI`, der bei der Instanziierung ergänzt wird. Dieser Eingang nimmt das zu multiplexende Ereignis entgegen.

Ereignis-Ausgänge

- **EO1** – Ausgang für Index 0
- **EO2** – Ausgang für Index 1
- **EO3** – Ausgang für Index 2
- **EO4** – Ausgang für Index 3
- **EO5** – Ausgang für Index 4
- **EO6** – Ausgang für Index 5
- **EO7** – Ausgang für Index 6

Die Kommentare in der XML geben an, dass jeder Ausgang das demultiplexte Signal von EI abhängig vom Wert von K liefert.

Daten-Eingänge

Keine Daten-Eingänge vorhanden. Die Auswahl des Zielausgangs erfolgt ausschließlich über den Adapter.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

- **K** (Socket) – Typ: `adapter::types::unidirectional::AUI`
  Dient zur Übergabe des Indexwertes (0–6), der bestimmt, welcher Ereignisausgang aktiviert wird.

Funktionsweise

Ein am Eingang `EI` ankommendes Ereignis wird über den Demultiplexer an den Ausgang weitergeleitet, dessen Index dem aktuellen Wert des Adapters `K` entspricht. Wenn der Wert von `K` im Bereich 0 bis 6 liegt, wird genau einer der Ausgänge `EO1` bis `EO7` getriggert. Bei Werten außerhalb dieses Bereichs wird kein Ausgang aktiviert, das Ereignis wird verworfen.

Da der Baustein generisch ist, kann der Typ des Eingangsereignisses und der anderen Parameter über die generische Instanziierung angepasst werden. Der konkrete Ereigniseingang `EI` wird durch die zugrunde liegende generische Klasse `GEN_E_DEMUX` ergänzt.

Technische Besonderheiten

- **Generischer Baustein:** Durch das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `'GEN_E_DEMUX'` wird der Baustein als generischer Funktionsbaustein deklariert. Das bedeutet, dass der Funktionsumfang und die Schnittstelle teilweise zur Laufzeit definiert werden.
- **Adapterbasierte Indexübergabe:** Der Index wird nicht über einen einfachen Dateneingang, sondern über einen Socket vom Typ `AUI` (unidirectional) übergeben. Dies ermöglicht eine modulare und lose gekoppelte Verbindung zur übergeordneten Logik.
- **Anzahl der Ausgänge:** Sieben Ausgänge sind fest im Typ definiert; eine dynamische Änderung der Ausgangsanzahl ist nicht vorgesehen.

Zustandsübersicht

Es existiert kein expliziter Zustandsautomat. Die Funktionsweise ist rein ereignisgetrieben: Beim Eintreffen eines Ereignisses am Eingang `EI` wird sofort der passende Ausgang gemäß dem aktuellen Adapterwert `K` aktiviert.

Anwendungsszenarien

- Verteilung von Ereignissen in einer Steuerungsanwendung auf verschiedene Verarbeitungseinheiten (z. B. je nach ausgewähltem Modus).
- Umschaltung zwischen verschiedenen Betriebsmodi über einen Adapter-basierten Auswahlmechanismus.
- Einsatz in modularen Systemen, in denen die Auswahl logik über eine Adapter-Schnittstelle erfolgen soll.

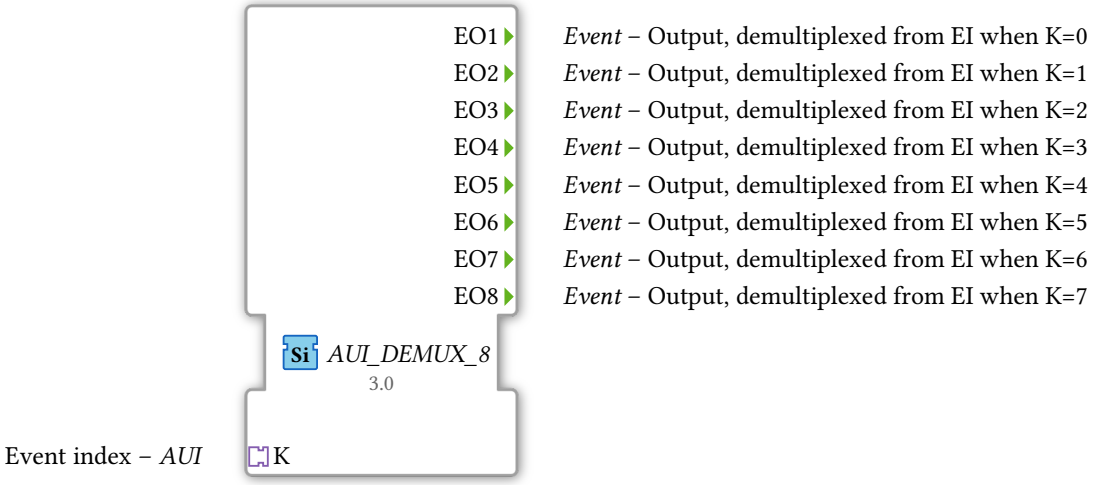
Vergleich mit ähnlichen Bausteinen

Der klassische `E_DEMUX_7` besitzt einen direkten Ereigniseingang `EI` und einen Dateneingang `K` (typischerweise als INTEGER). Der `AUI_DEMUX_7` ersetzt den Dateneingang durch einen Adapter-Socket. Dies erhöht die Flexibilität bei der Anbindung, da der Index nicht als separate Datenleitung, sondern über eine komplexere Adapterstruktur übergeben wird. Dadurch ist eine klarere Trennung von Steuerlogik und Ereignisweiterleitung möglich. Als generischer Baustein kann er zudem in unterschiedlichen Kontexten mit variierenden Ereignistypen verwendet werden.

Fazit

Der `AUI_DEMUX_7` ist ein nützlicher generischer Event-Demultiplexer, der sich durch die Verwendung eines AUI-Adapters für die Indexübergabe von herkömmlichen Demultiplexern abhebt. Er eignet sich besonders für modulare Steuerungsarchitekturen, in denen eine lose Kopplung und flexible Konfiguration gewünscht ist. Die feste Anzahl von sieben Ausgängen deckt typische Anwendungen ab, bei denen eine Auswahl aus mehreren Optionen getroffen werden muss.

AUI_DEMUX_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUI_DEMUX_8** ist ein Ereignis-Demultiplexer mit 8 Ausgängen. Er empfängt über einen AUI-Adapter (Latching-Adapter) sowohl das auslösende Ereignis als auch den Auswahlindex. Abhängig vom übergebenen Index wird das eingehende Ereignis an genau einen der acht Ereignis-Ausgänge weitergeleitet. Der Baustein ist als generischer FB (GenericClassName `GEN_E_DEMUX`) implementiert und stellt eine moderne, adapterbasierte Alternative zum klassischen `E_DEMUX` dar.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine direkten Ereignis-Eingänge. Das Eingangsereignis wird über den AUI-Adapter `K` empfangen, der entsprechend der Adapterdefinition ein Ereignis sowie einen Datenwert (Index) bündelt.

Ereignis-Ausgänge

Name	Datentyp	Kommentar
`EO1`	Event	Ausgang, demultiplext vom Eingangsereignis bei K = 0
`EO2`	Event	Ausgang, demultiplext vom Eingangsereignis bei K = 1
`EO3`	Event	Ausgang, demultiplext vom Eingangsereignis bei K = 2
`EO4`	Event	Ausgang, demultiplext vom Eingangsereignis bei K = 3
`EO5`	Event	Ausgang, demultiplext vom Eingangsereignis bei K = 4
`EO6`	Event	Ausgang, demultiplext vom Eingangsereignis bei K = 5
`EO7`	Event	Ausgang, demultiplext vom Eingangsereignis bei K = 6
`EO8`	Event	Ausgang, demultiplext vom Eingangsereignis bei K = 7

Daten-Eingänge

Es existieren keine direkten Daten-Eingänge. Der Index `K` wird als Datenwert über den AUI-Adapter bereitgestellt.

Daten-Ausgänge

Der Baustein besitzt keine Daten-Ausgänge.

Adapter

Name	Typ	Richtung	Kommentar
K	adapter::types::unidirectional::AUI	Socket	Ereignisindex (Kanalauswahl)

Funktionsweise

Der Demultiplexer wartet auf ein Ereignis, das über den AUI-Adapter K eintrifft. Mit diesem Ereignis wird gleichzeitig ein Datenwert (Index) übertragen. Dieser Index bestimmt, welcher der acht Ausgänge E01 bis E08 aktiviert wird:

- Bei K = 0 wird das Ereignis an E01 weitergegeben.
- Bei K = 1 wird das Ereignis an E02 weitergegeben.
- ...
- Bei K = 7 wird das Ereignis an E08 weitergegeben.

Sobald das Ereignis eingetroffen und der Index ausgewertet ist, wird genau **ein** Ausgangsereignis erzeugt. Die Zuordnung ist deterministisch und ohne Verzögerung (kein interner Zustand).

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**
  Der Baustein verwendet einen AUI-Adapter anstelle getrennter Eingänge für Ereignis (EI) und Index (K). Dies vereinfacht die Verbindung in Systemen, die bereits auf Adapterkommunikation aufgebaut sind, und erhöht die Wiederverwendbarkeit.
- **Generischer Baustein**
  Über das Attribut GenericClassName wird auf den generischen FB GEN_E_DEMUX verwiesen. Dies ermöglicht eine flexible Parametrisierung und spätere Erweiterung (z. B. andere Demultiplexer-Breiten).
- **Keine Datenhaltung**
  Der Baustein ist zustandslos. Es werden keine internen Variablen oder Zustände gespeichert, was ihn für hochdynamische Prozesse geeignet macht.
- **Vollständige Abdeckung des Indexbereichs**
  Für die Werte 0 bis 7 sind alle Ausgänge definiert. Ein ungültiger Index (außerhalb dieses Bereichs) führt zu keinem Ausgangsereignis.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten. Das Verhalten ist rein kombinatorisch:

1. Warten auf Ereignis über den Adapter.
2. Empfang des Ereignisses inkl. Index.
3. Auswahl und Aktivierung des zugeordneten Ausgangs.
4. Rückkehr in den Wartezustand.

Es gibt keine internen Zustandsübergänge oder zeitlichen Bedingungen.

Anwendungsszenarien

- **Verteilte Ereignisverarbeitung**
  Ein zentrales Steuermodul sendet Ereignisse mit einem Prioritäts- oder Kanalindex. Der AUI_DEMUX_8 leitet jedes Ereignis an die entsprechende Verarbeitungseinheit (z. B. verschiedene Regelungsblöcke) weiter.
- **Kanalumschaltung in Kommunikationssystemen**
  In Nachrichten- oder Datenflussarchitekturen können über den Index unterschiedliche Empfänger adressiert werden, ohne dass die Verbindungslogik neu konfiguriert werden muss.
- **Steuerung von Mehrwegeventilen oder multiplexierten Aktoren**
  Durch die Auswahl des Ausgangs können z. B. verschiedene Aktuatoren angesteuert werden, wobei nur ein Aktuator pro Ereignis reagieren soll.

Vergleich mit ähnlichen Bausteinen

Eigenschaft	AUI_DEMUX_8	E_DEMUX (klassisch)

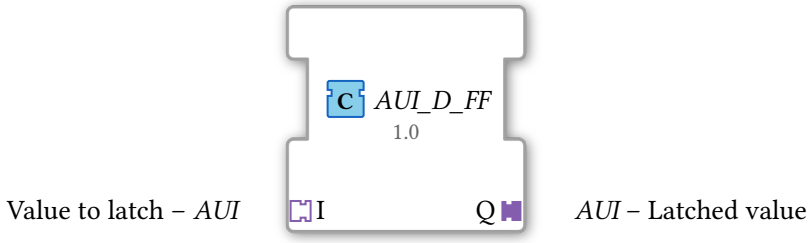
Ereigniseingang	Über Adapter `K`	Separater Eingang `EI`
Indexeingang	Über Adapter `K` (Datenkanal)	Separater Dateneingang `K`
Anzahl Ausgänge	8 (EO1–EO8)	Konfigurierbar, z. B. 2, 4, 8
Schnittstelle	Einheitlicher AUI-Adapter	Einzelne Ein-/Ausgänge
Generisch	Ja (über `GEN_E_DEMUX`)	Teilspezifisch

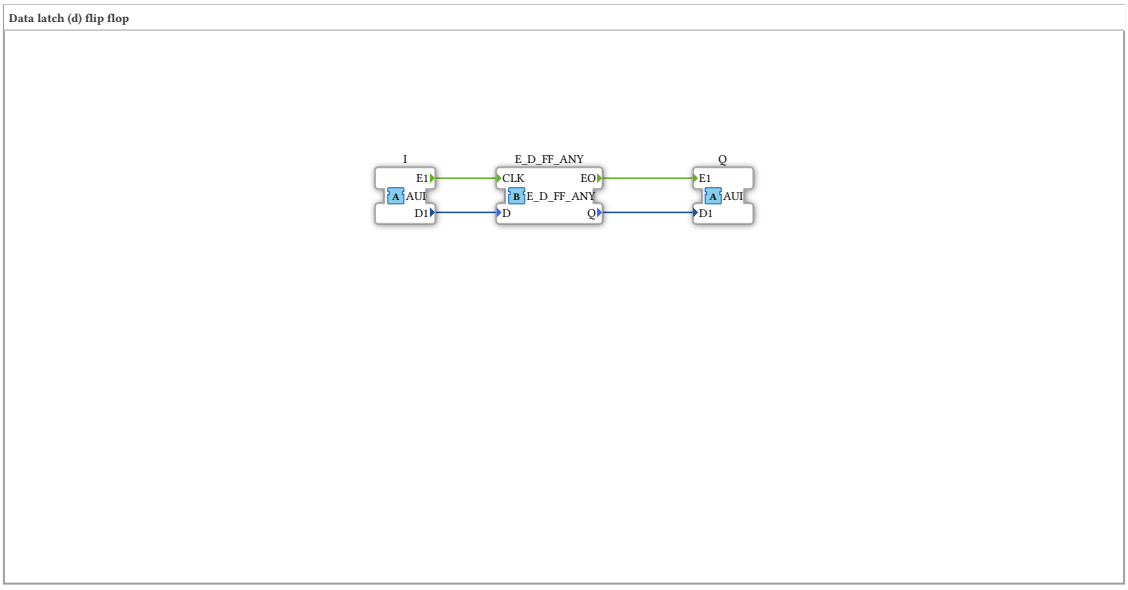
Der wesentliche Unterschied liegt in der Zusammenführung von Ereignis und Index in einem Adapter. Dadurch wird die Verkabelung vereinfacht und die Schnittstelle für modulare Systeme standardisiert.

Fazit

Der `AUI_DEMUX_8` ist ein leistungsfähiger, ereignisbasierter Demultiplexer, der die Vorteile moderner Adaptertechnik mit der Flexibilität generischer Bausteine verbindet. Durch die klare Zuordnung von Index zu Ausgang und die zustandslose Arbeitsweise eignet er sich ideal für viele industrielle Automatisierungs- und Kommunikationsszenarien. Seine adapterbasierte Schnittstelle reduziert Verdrahtungsaufwand und erhöht die Wartbarkeit der Gesamtapplikation.

AUI_D_FF





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **AUI_D_FF** realisiert ein taktgesteuertes D-Flipflop (Data Latch). Er übernimmt den Wert eines Dateneingangs bei einem anliegenden Ereignis und gibt den gespeicherten Wert am Ausgang aus. Die Ein- und Ausgangsschnittstellen sind als unidirektionale **Adapter** ausgeführt, was eine flexible und standardisierte Verbindung in der 4diac-IDE ermöglicht.

Schnittstellenstruktur

Ereignis-Eingänge
Keine – die Ereignissteuerung erfolgt ausschließlich über den Adapter **I**.

Ereignis-Ausgänge
Keine – die Ereignisausgabe erfolgt ausschließlich über den Adapter **Q**.

Daten-Eingänge
Keine – die Datenübergabe erfolgt ausschließlich über den Adapter **I**.

Daten-Ausgänge
Keine – die Datenausgabe erfolgt ausschließlich über den Adapter **Q**.

Adapter

Bezeichnung	Typ	Richtung	Beschreibung
I	`adapter::types::unidirectional::AUI`	Socket (Eingang)	Eingangswert (D) und Taktereignis (CLK)
Q	`adapter::types::unidirectional::AUI`	Plug (Ausgang)	Gespeicherter Wert (Q) und Ausgabeereignis (EO)

Der Adaptertyp **AUI** bietet einen unidirektionalen Kanal, der genau ein Ereignis und einen Datenwert überträgt. Somit werden die internen Verbindungen zwischen dem Flipflop und der Außenwelt über Adapter realisiert.

Funktionsweise
Der FB nutzt intern den Baustein `E_D_FF_ANY` aus der IEC-61499-Standardbibliothek.

- Bei jedem **Taktereignis** auf `I.E1` wird der aktuelle Datenwert von `I.D1` übernommen.
- Der übernommene Wert wird am Ausgang `Q.D1` bereitgestellt und parallel ein **Ausgabeereignis** auf `Q.E1` ausgelöst.
- Der gespeicherte Wert bleibt solange erhalten, bis ein neues Taktereignis einen neuen Datenwert übernimmt.

Technische Besonderheiten
- Die Schnittstelle besteht ausschließlich aus **Adaptern**, nicht aus getrennten Ereignis- und Datenports. Dies erlaubt eine lose Kopplung und Wiederverwendbarkeit des Adaptertyps in verschiedenen Kontexten.
- Der Adapter `AUI` ist **unidirektional** – Daten und Ereignisse fließen nur in eine Richtung.
- Der Baustein ist typsicher: Der Datentyp wird durch den Adapter vorgegeben (im Standard `ANY`).

Zustandsübersicht
Der FB besitzt zwei logische Zustände, die dem internen Flipflop entsprechen:

Zustand	Beschreibung
Q = 0	Ausgangswert ist LOW (logisch 0).
Q = 1	Ausgangswert ist HIGH (logisch 1).

Der Zustand wechselt ausschließlich bei einer steigenden Flanke des Taktereignisses (sofern das Ereignis im Adapter ausgelöst wird).

Anwendungsszenarien
- **Synchronisation**: Übernahme eines Datenwertes genau zu einem definierten Zeitpunkt (z.B. bei einem Sensor-Event).
- **Zwischenspeicher**: Puffern eines Wertes für die Weiterverarbeitung in nachfolgenden Bausteinen.
- **Zustandsmaschinen**: Speicherung von Zuständen in diskreten Steuerungen, wenn eine einfache D-Flipflop-Funktion ausreicht.

Vergleich mit ähnlichen Bausteinen

- **E_D_FF** (Standard-IEC-61499): Bietet dieselbe Flipflop-Funktionalität, jedoch mit separaten Ereignis- und Datenports. Der `AUI_D_FF` kapselt diese Ports in einem Adapter und ermöglicht so eine saubere, adapterbasierte Kommunikation.
- **E_RS** oder **E_SR** (Reset-Set-Flipflops): Im Gegensatz zum D-Flipflop benötigen diese Bausteine separate Set-/Reset-Ereignisse und keinen Takt. Der `AUI_D_FF` ist für taktgesteuerte Anwendungen optimiert.

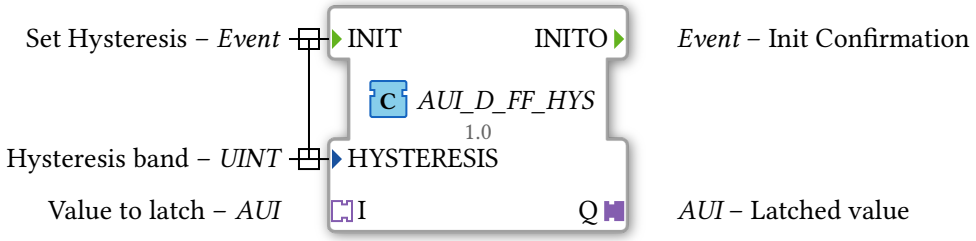
Fazit

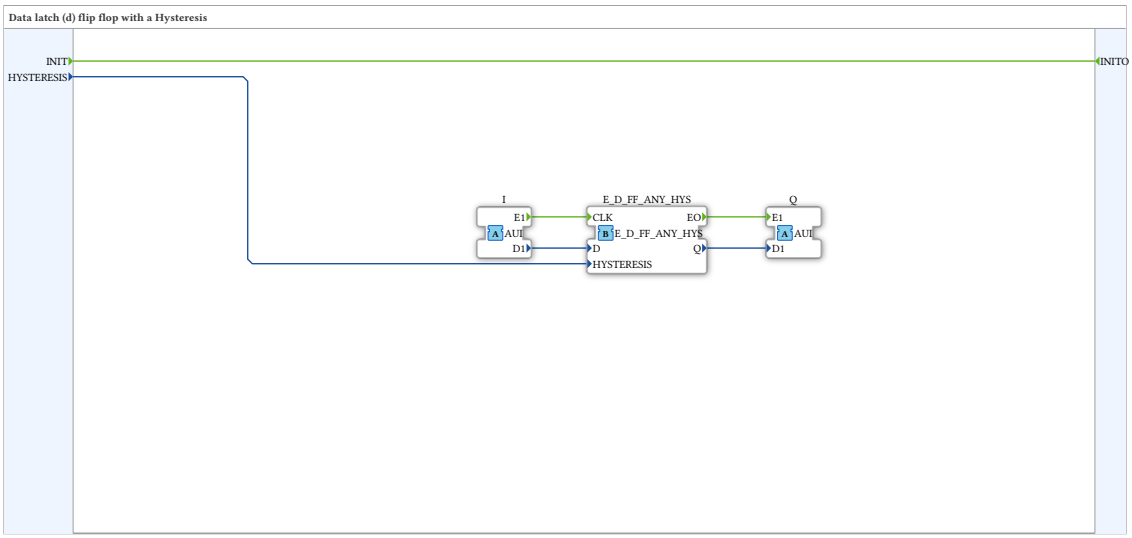
Der `AUI_D_FF` stellt ein standardkonformes D-Flipflop in der 4diac-IDE dar, das durch die Verwendung von **Adaptern** eine modulare und wiederverwendbare Schnittstelle bietet. Er eignet sich besonders für Anwendungen, bei denen eine einheitliche, unidirektionale Verbindungsstruktur gewünscht ist, ohne auf die bewährte Flipflop-Logik verzichten zu müssen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUI_D_FF_HYS





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein `AUI_D_FF_HYS` realisiert ein Daten-Latch (D-Flipflop) mit einer Hysteresefunktion. Er dient zum zuverlässigen Übernehmen eines analogen oder digitalen Werts, wobei kleine Schwankungen (Rauschen) durch ein einstellbares Hystereseband unterdrückt werden. Der Baustein ist für den Einsatz in der Automatisierungstechnik konzipiert und folgt der Norm IEC 61499-1 Annex A.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar

INIT	EInit	Setzt die Hysteresebreite

Ereignis-Ausgänge

Ereignis	Typ	Kommentar
INITO	EInit	Bestätigung der Initialisierung

Daten-Eingänge

Daten	Typ	Kommentar
HYSTERESIS	UINT	Breite des Hysteresebandes

Daten-Ausgänge

Keine dedizierten Datenausgänge; der latched Wert wird über den Adapter `Q` bereitgestellt.

Adapter

Name	Typ	Richtung	Kommentar
I	adapter::types::unidirectional::AUI	Socket	Wert zum Latchen
Q	adapter::types::unidirectional::AUI	Plug	Gelatchter Wert

Die Adapter `I` und `Q` vom Typ `AUI` (unidirektional) transportieren sowohl Daten (z.B. `D1`) als auch zugehörige Ereignisse (z.B. `E1`).

Funktionsweise

Der Baustein enthält intern die Instanz `E_D_FF_ANY_HYS` (Typ `logiBUS::signalprocessing::hysteresis::E_D_FF_ANY_HYS`), die das eigentliche Latch mit Hysterese ausführt. Die Signalflüsse sind:

- Das Ereignis `I.E1` (vom Adapter `I`) wird direkt als Takt (`CLK`) an den inneren Baustein weitergegeben.
- Der Datenwert `I.D1` wird als `D`-Eingang an das interne Flipflop übergeben.
- Das Hystereseband wird über den Parameter `HYSTERESIS` gesetzt.
- Der Ausgang des internen Bausteins (`Q`) wird auf den Adapter `Q.D1` gelegt, das Ausgangsereignis (`EO`) auf `Q.E1`.
- Das Ereignis `INIT` wird sofort auf `INITO` durchgeschliffen (keine Verzögerung).

Die Hysterese bewirkt, dass der latched Wert nur dann auf den neuen Eingangswert springt, wenn sich dieser um mehr als die festgelegte Schwelle (`HYSTERESIS`) vom aktuell gespeicherten Wert unterscheidet. Dadurch werden Flankenrauschen und unstete Signalwechsel unterdrückt.

Technische Besonderheiten

- Der Baustein ist ein reiner **Composite-Funktionsbaustein**, der die gesamte Logik an eine interne Instanz delegiert.
- Die Hysterese wird als vorzeichenloser Integer (`UINT`) übergeben.
- Die Adapter sind unidirektional ausgelegt – Daten und Ereignisse fließen nur in einer Richtung (Socket → Plug).
- Der Baustein enthält keine eigene Zustandsmaschine; der Zustand wird im inneren Flipflop gehalten.

Zustandsübersicht

Da `AUI_D_FF_HYS` kein eigenes ECC (Execution Control Chart) besitzt, ergibt sich das Verhalten allein aus dem internen Baustein `E_D_FF_ANY_HYS`. Typischerweise besitzt ein D-Flipflop mit Hysterese zwei stabile Ausgangszustände, die durch den angelegten Takt und die Hysteresebedingung umgeschaltet werden. Eine detaillierte Zustandsbeschreibung ist der Dokumentation des internen Bausteins zu entnehmen.

Anwendungsszenarien

- **Entprellung von Sensordaten**: Werte, die aus analogen oder digitalen Sensoren kommen, werden mit einer einstellbaren Hysterese stabilisiert.

- **Schwellwertschalter**: Der Baustein kann als Schmitt-Trigger verwendet werden, wenn ein Eingangswert nur oberhalb einer Ober- und unterhalb einer Unterschwelle (bestimmt durch Hysterese) seinen Ausgang wechselt.
- **Signalaufbereitung in der Prozessautomatisierung**: Rauschbehaftete Signale werden geglättet, bevor sie an nachfolgende Logikbausteine weitergegeben werden.

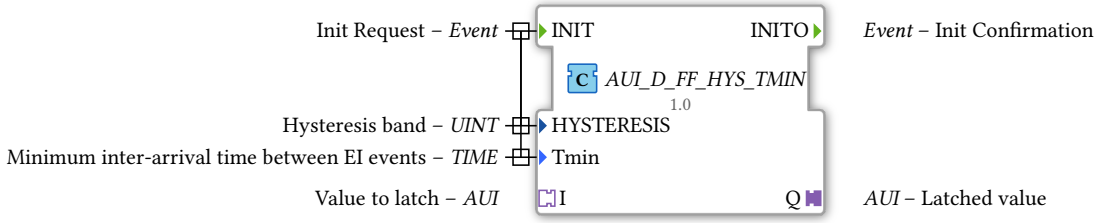
Vergleich mit ähnlichen Bausteinen

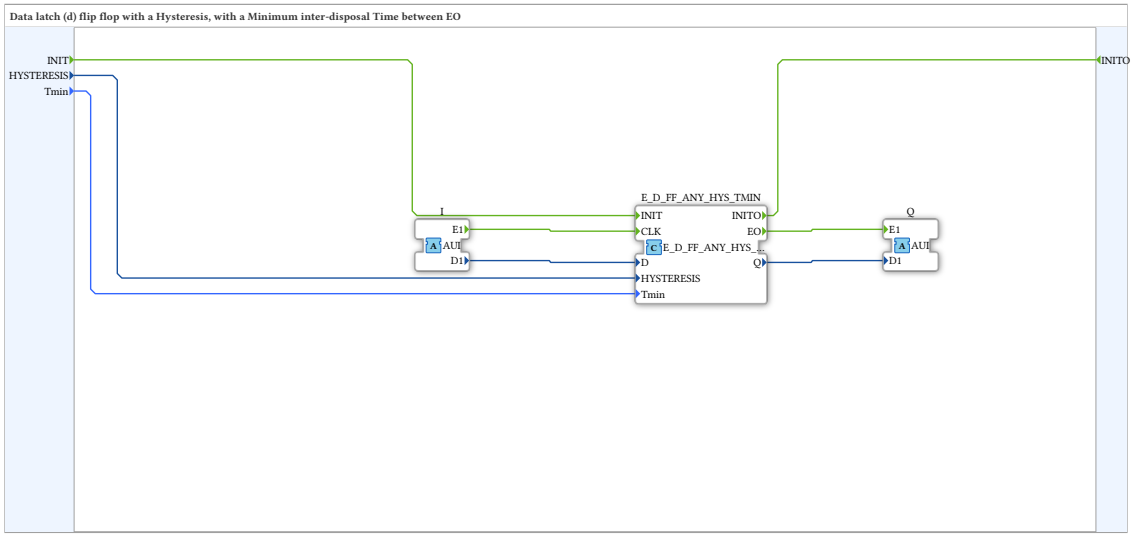
- **E_D_FF**: Ein reines D-Flipflop ohne Hysterese. Es übernimmt den Eingangswert bei jeder Taktflanke, unabhängig von minimalen Schwankungen. `AUI_D_FF_HYS` bietet eine verbesserte Störsicherheit.
- **E_RS_BISTABLE**: Ein Set-Reset-Flipflop, das keine Hysterese besitzt und auf binäre Zustände beschränkt ist. `AUI_D_FF_HYS` arbeitet mit analogen/ganzzahligen Werten und ermöglicht eine dynamische Schwelle.
- **E_SPLIT_HYS**: Ein Baustein, der ein Signal mit einer festen Hysterese aufteilt. Im Gegensatz dazu kann die Hysterese in `AUI_D_FF_HYS` über den Eingang `HYSTERESIS` zur Laufzeit geändert werden.

Fazit

Der `AUI_D_FF_HYS` ist ein nützlicher Baustein zur stabilen Latchung von Werten in Umgebungen mit Rauschen oder Signalinstabilitäten. Er vereint die Funktionalität eines D-Flipflops mit einer einstellbaren Hystereseschwelle und ist durch die Verwendung von Adaptern leicht in modulare IEC-61499-Steuerungen integrierbar.

AUI_D_FF_HYS_TMIN





• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AUI_D_FF_HYS_TMIN** ist ein Datenspeicher (D-Flip-Flop) mit einstellbarer Hysterese und einer minimalen Zwischenereigniszeit (`Tmin`). Er wird über Adapter-Schnittstellen (Socket `I` und Plug `Q`) mit anderen Bausteinen verbunden. Der interne Kern entspricht dem FB `E_D_FF_ANY_HYS_TMIN` aus der Bibliothek `logiBUS::signalprocessing::hysteresis`. Die Initialisierung legt die Parameter fest, danach wird bei jedem eingehenden Ereignis am Eingang `I.E1` der aktuelle Wert von `I.D1` übernommen, gefiltert und über `Q.D1` ausgegeben.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Initialisierungsanforderung

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Initialisierungsbestätigung

Daten-Eingänge

Name	Typ	Kommentar
HYSTERESIS	UINT	Hysterese-Band
Tmin	TIME	Minimale Zwischenankunftszeit der EI-Ereignisse

Daten-Ausgänge

Keine eigenständigen Datenausgänge – der ausgegebene Wert wird über den Adapter `Q` übertragen.

Adapter

Typ	Bezeichnung	Kommentar
`adapter::types::unidirectional::AUI` (Socket)	I	Zu speichernder Wert
`adapter::types::unidirectional::AUI` (Plug)	Q	Gespeicherter Wert

Der Adapter `AUI` ist ein unidirektionaler Typ, der ein Ereignis und einen Datenwert bündelt (`E1` und `D1`).

Funktionsweise

1. **Initialisierung** Beim Ereignis `INIT` werden die Parameter `HYSTERESIS` (als UINT) und `Tmin` (als TIME) an den internen Baustein übergeben. Nach erfolgreicher Initialisierung wird `INITO` ausgelöst.
2. **Betrieb** Der interne FB `E_D_FF_ANY_HYS_TMIN` arbeitet als getaktetes D-Flip-Flop mit Hysterese und Zeitbegrenzung:
   - Ein Ereignis an `I.E1` (Clock) übernimmt den momentanen Datenwert von `I.D1`.
   - Der übernommene Wert wird nur dann als gültiger Ausgang `Q.D1` weitergegeben, wenn der Abstand zum letzten gültigen Ereignis größer oder gleich `Tmin` ist.
   - Zusätzlich wird der Wert mit einer Hysterese (band `HYSTERESIS`) geglättet: Kleine Änderungen unterhalb der Hystereseschwelle werden unterdrückt.
   - Der Ausgang wird über das Ereignis `Q.E1` signalisiert.
3. **Ergebnis** Der Ausgangsadapter `Q` liefert den zuletzt angenommenen, gefilterten Wert.

Technische Besonderheiten

- **Hysterese** in Verbindung mit einer **Mindestzeit zwischen zwei Ereignissen** verhindert sowohl Rauschen als auch zu schnelle Zustandswechsel.
- Der Baustein ist als **Adapter-Wrapper** aufgebaut – er nutzt standardisierte unidirektionale AUI-Adapter für Ein- und Ausgabe.
- Die Implementierung delegiert die gesamte Logik an den FB `E_D_FF_ANY_HYS_TMIN`, der universell für beliebige Datentypen ausgelegt ist. Hier wird er im Kontext des AUI-Adapters verwendet.

Zustandsübersicht

Der Baustein besitzt keinen expliziten internen Zustandsautomaten. Das Verhalten wird durch den internen FB abgebildet:

- **Idle:** Wartet auf nächstes Ereignis an `I.E1`.
- **Sampling:** Bei Eintreffen von `I.E1` wird der Datenwert erfasst.

- **Filterung:** Prüfung gegen Hysterese und Tmin.
- **Ausgabe:** Bei Erfolg wird `Q.E1` ausgelöst und der neue Wert auf `Q.D1` gelegt.

Anwendungsszenarien

- **Sensor-Wert-Erfassung** in der Landtechnik (siehe Copyright HR Agrartechnik GmbH): Verarbeitung von Analog- oder Digitalsignalen mit Entprellung und Mindestabtastrate.
- **Steuerungskomponenten** in Automatisierungssystemen, bei denen Rauschen unterdrückt und eine Mindestpulsdauer gefordert wird.
- **Datenlogik** in verteilten Systemen, die auf asynchrone Ereignisse mit Stabilitätsanforderungen reagieren müssen.

Vergleich mit ähnlichen Bausteinen

Baustein	Eigenschaft
`E_D_FF`	Einfaches D-Flip-Flop ohne Hysterese oder Zeitbegrenzung
`E_D_FF_HYS`	D-Flip-Flop mit Hysterese, aber ohne `Tmin`
`E_D_FF_HYS_TMIN` (dieser FB)	Kombiniert Hysterese *und* minimale Zwischenzeit → robuster gegen Rauschen und Burst-Ereignisse

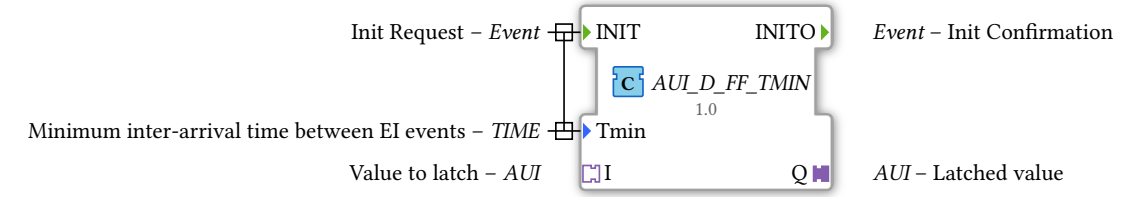
Fazit

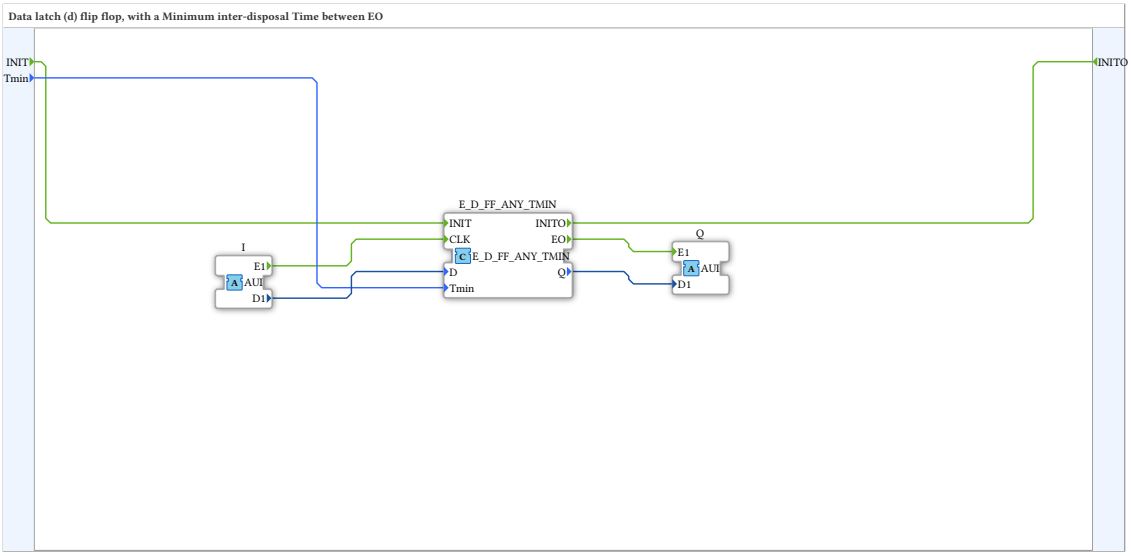
Der `AUI_D_FF_HYS_TMIN` bietet eine kompakte und zuverlässige Lösung zur signalverarbeitenden Datenerfassung, die durch Hysterese und Zeitbegrenzung unerwünschte Störimpulse unterdrückt. Die Adapter-Schnittstelle erleichtert die Integration in bestehende 4diac-Projekte mittels standardisierter unidirektionaler AUI-Verbindungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUI_D_FF_TMIN





• – ○ • – ○ • – ○ • –

Einleitung

Der Funktionsblock **AUI_D_FF_TMIN** realisiert ein datenverriegelndes D-Flipflop (Data Latch) mit einer minimalen zeitlichen Sperre zwischen zwei aufeinanderfolgenden Ausgabeereignissen (*Minimum inter-disposal Time*). Er dient dazu, Werte über den unidirektionalen **AUI**-Adapter zu übernehmen und mit einer definierten Mindestverzögerung weiterzugeben, um eine übermäßig schnelle Ereignisfolge am Ausgang zu vermeiden.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Kommentar
INIT	Initialisierungsanforderung; konfiguriert den Block mit dem Parameter `Tmin`.

Ereignis-Ausgänge

Ereignis	Kommentar
INITO	Bestätigung der erfolgreichen Initialisierung.

Daten-Eingänge

Variable	Typ	Kommentar
Tmin	TIME	Minimale Zeitdauer, die zwischen zwei Ausgabeereignissen (`EO`) mindestens verstreichen muss.

Daten-Ausgänge

Keine direkten Datenausgänge – die Ausgabewerte werden über den Adapter `Q` bereitgestellt.

Adapter

Richtung	Name	Typ	Kommentar
Plug (Ausgang)	**Q**	`adapter::types::unidirectional::AUI`	Liefert den verriegelten Datenwert (Ereignis- und Datenleitung).
Socket (Eingang)	**I**	`adapter::types::unidirectional::AUI`	Empfängt den zu verriegelnden Datenwert (Ereignis- und Datenleitung).

Funktionsweise

1. **Initialisierung**
   Über den Ereigniseingang `INIT` wird der Block initialisiert und die Mindestzeit `Tmin` festgelegt.
2. **Datenübernahme**
   Jedes Ereignis, das am Adaptereingang `I` über die Ereignisleitung `I.E1` eintrifft, übernimmt den zugehörigen Datenwert `I.D1` in das Flipflop. Der übernommene Wert wird sofort intern gespeichert.
3. **Ausgabe mit Mindestzeit**
   Die Ausgabe des gespeicherten Werts erfolgt erst dann als Ereignis auf `Q.E1` und Datenwert auf `Q.D1`, wenn seit dem letzten Ausgabeereignis mindestens die Zeit `Tmin` vergangen ist. Tritt vor Ablauf dieser Frist ein neues Übernahmeereignis ein, so wird der neue Wert gespeichert und die Ausgabe weiter verzögert, bis die Mindestzeit eingehalten ist.
4. **Zustandssicherung**
   Der Block verhält sich wie ein flankengetriggertes D-Flipflop, bei dem jeder gültige Takt (Ereignis auf `I.E1`) den aktuellen Datenwert übernimmt, die tatsächliche Weitergabe jedoch zeitlich gedrosselt ist.

Technische Besonderheiten

- **Adapterbasierte Ein-/Ausgabe** – Der Baustein verwendet ausschließlich den standardisierten unidirektionalen **AUI**-Adapter. Dadurch ist er leicht in bestehende Adapter-orientierte Architekturen integrierbar.
- **Wiederverwendung von Basis-Bausteinen** – Intern wird der IEC-61499-Baustein `E_D_FF_ANY_TMIN` eingesetzt, der die zeitliche Drosselung implementiert. Der vorliegende Block kapselt diesen und vereinfacht die Anbindung über Adapter.
- **Keine eigenen Zustandsautomaten** – Das Verhalten wird vollständig durch den inneren Baustein bestimmt; der äußere FB ist rein konfigurativ und verbindend.

Zustandsübersicht

Der Funktionsblock besitzt keinen extern sichtbaren Zustandsautomaten (ECC). Intern kann der verwendete `E_D_FF_ANY_TMIN` folgende implizite Zustände aufweisen:

Zustand	Beschreibung
Idle	Wartet auf ein Ereignis am Eingang `I`.
Getaktet	Daten wurden übernommen; Ausgabe wird bis zum Ablauf von `Tmin` zurückgehalten.
Ausgabe	Mindestzeit eingehalten – Ausgabeereignis wird gesendet.

Diese Zustände sind für den Anwender transparent.

Anwendungsszenarien

- **Sensorwert-Glättung** – Verhindert, dass ein Sensor, der in kurzen Abständen Messwerte liefert, den nachfolgenden Verarbeitungsblock mit Ereignissen überflutet.
- **Aktor-Ansteuerung mit Mindestschaltzeit** – Stellt sicher, dass ein Aktor (z. B. ein Ventil) nur nach einer definierten Mindestpause geschaltet werden kann, um Verschleiß zu reduzieren.
- **Datenflussbegrenzung in Kommunikationsleitungen** – Drosselt die Ausgaberate eines Datenstroms, um eine Überlastung des Empfängers zu vermeiden.

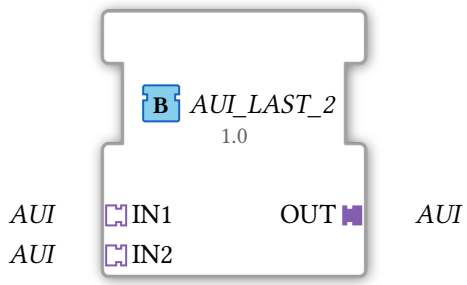
Vergleich mit ähnlichen Bausteinen

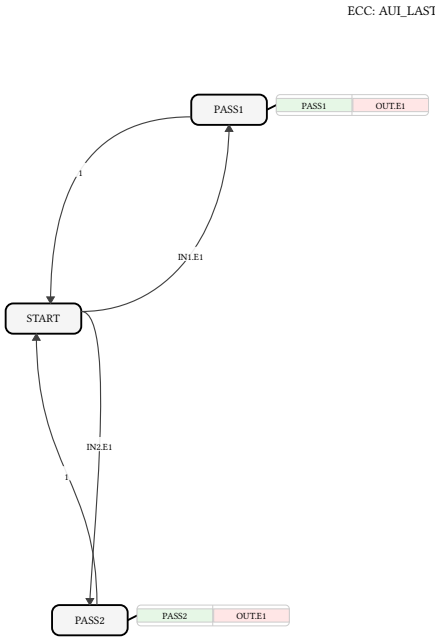
Baustein	Merkmal
E_D_FF (Standard-D-Flipflop)	Keine Zeitbeschränkung; jedes Ereignis wird sofort ausgegeben.
E_D_FF_ANY_TMIN	Gleiches Verhalten, jedoch ohne Adapter-Kapselung; direkte ereignis-/datenbasierte Anbindung.
AUI_D_FF_TMIN (dieser Block)	Kombiniert die D-Flipflop-Funktionalität mit einer zeitlichen Drosselung und bietet eine adapterbasierte Schnittstelle.

Fazit

Der **AUI_D_FF_TMIN** ist ein nützlicher Funktionsblock, wenn Daten in zeitlich kontrollierten Abständen weitergegeben werden sollen. Er vereinfacht die Implementierung von Drosselungslogiken in adapterbasierten IEC-61499-Systemen und eignet sich besonders für Anwendungen in der Landtechnik und anderen zeitkritischen Steuerungsumgebungen.

AUI_LAST_2





Einleitung

Der AUI_LAST_2 Funktionsblock führt zwei gleichartige UINT-Adapter-Signale (**IN1**, **IN2**) auf einen gemeinsamen Ausgang (**OUT**) zusammen, nach dem Prinzip **Last Writer Wins**: welcher der beiden Sockets zuletzt ein Ereignis geschrieben hat, gewinnt, und sein Datenwert wird sofort und unverändert an OUT durchgereicht. Es handelt sich ausdrücklich nicht um ein naives Zusammenführen (z. B. Mittelwert oder ODER-Verknüpfung), sondern um eine reine Arbitrierung nach zeitlicher Reihenfolge.

Schnittstellenstruktur

Adapter

Eingangsadapter (Sockets):

- **IN1**: UINT-Adapter (unidirectional) - erste Signalquelle
- **IN2**: UINT-Adapter (unidirectional) - zweite Signalquelle

Ausgangsadapter (Plug):

- **OUT**: UINT-Adapter (unidirectional) - zusammengeführtes Signal

Funktionsweise

Der Baustein ist als einfacher Basic-FB mit einer 3-Zustands-ECC umgesetzt (START, PASS1, PASS2). Trifft an IN1 ein Ereignis ein, wechselt die ECC nach PASS1: der aktuelle Datenwert von IN1 wird nach OUT kopiert und das Adapter-Ereignis an OUT gesendet. Trifft an IN2 ein Ereignis ein, geschieht dasselbe symmetrisch über PASS2. Nach jeder Durchreichung kehrt die ECC sofort wieder nach START zurück und ist bereit für das nächste Ereignis von IN1 oder IN2.

Technische Besonderheiten

- Last-Writer-Wins-Semantik statt naiver Datenverknüpfung
- Einfacher Basic-FB (ECC mit 3 Zuständen), kein zusätzlicher Zwischenspeicher nötig
- Keine Signalverzögerung: der Wert wird im selben Zyklus durchgereicht, in dem das auslösende Ereignis eintrifft
- Beide Eingänge sind gleichberechtigt - es gibt keine feste Priorität außer der zeitlichen Reihenfolge

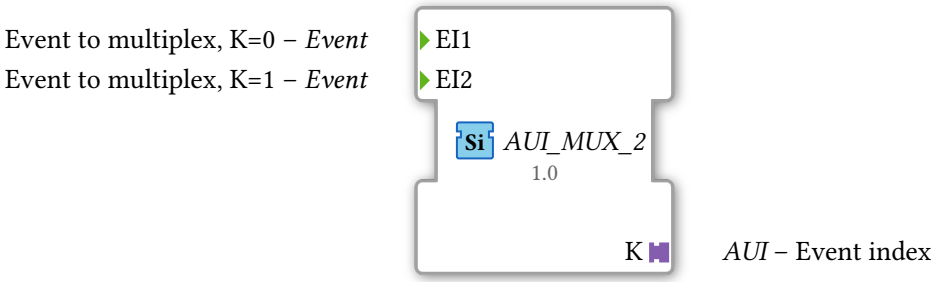
Anwendungsszenarien

- Zusammenführen zweier gleichwertiger UINT-Signalquellen zu einem einzigen nachgeschalteten Adapterpfad (z. B. zwei redundante Sensoren, oder ein manueller und ein automatischer Sollwertgeber)
- Arbitrierung zwischen zwei Bedienstellen, die denselben UINT-Wert schreiben dürfen
- Vereinfachung von Netzwerken, die sonst ein `F_SEL`/`E_RS`-Konstrukt inline verdrahten müssten

⚖️ Vergleich mit ähnlichen Bausteinen

AUI_LAST_2 ist die Umkehrung von `AUI_SPLIT_2`: AUI_SPLIT_2 verteilt 1 Eingang auf 2 Ausgänge, AUI_LAST_2 führt 2 Eingänge auf 1 Ausgang zusammen. Für Mischbetrieb zwischen einem Daten- und einem reinen Ereignisadapter (z. B. AX + AE) ist stattdessen `AX_AE_MERGE` gedacht, das nur das Ereignis mischt, ohne den Datenwert zu verändern.

AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AUI_MUX_2** ist ein ereignisgesteuerter Multiplexer-Baustein, der zwei Ereignis-Eingänge (EI1, EI2) über einen **AUI-Adapter** (Typ `adapter::types::unidirectional::AUI`) als Ausgangsschnittstelle bereitstellt. Er stellt eine spezielle Variante des klassischen `E_MUX_2` dar, bei dem der übliche Ereignis-Ausgang (EO) und der Daten-Eingang (K) durch einen einzigen unidirektionalen Adapter ersetzt wurden. Dadurch wird die Schnittstellenanzahl reduziert und eine nahtlose Integration in adapterbasierte Kommunikationsstrukturen ermöglicht.

Der Baustein ist als generischer Funktionsblock gekennzeichnet (Attribut `GenericClassName = 'GEN_E_MUX'`) und eignet sich für Anwendungen, bei denen zwei verschiedene Ereignisquellen auf eine gemeinsame, über einen Adapter angebundene Verarbeitungseinheit geführt werden sollen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Datentyp	Kommentar
EI1	Event	Ereignis 1, das über den Adapter weitergegeben wird
EI2	Event	Ereignis 2, das über den Adapter weitergegeben wird

Ereignis-Ausgänge

Der Baustein besitzt **keine** klassischen Ereignis-Ausgänge. Die Ausgabe erfolgt ausschließlich über den AUI-Adapter (siehe Abschnitt **Adapter**).

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

Name	Typ	Richtung	Kommentar

K	adapter::types::unidirectional::AUI	Ausgang	Überträgt das gewählte/weitergeleitete Ereignis an den angeschlossenen Partner

Der Adapter K stellt die einzige Ausgabeschnittstelle dar. Er ist als unidirektionaler Adapter ausgelegt und sendet bei jedem eintreffenden Ereignis an EI1 oder EI2 ein entsprechendes Ereignis an die angeschlossene Komponente.

Funktionsweise

Der **AUI_MUX_2** arbeitet rein ereignisgesteuert:

- Wird an **EI1** ein Ereignis empfangen, wird dieses unmittelbar über den Adapter K nach außen gesendet.
- Wird an **EI2** ein Ereignis empfangen, wird ebenso ein Ereignis über den Adapter K ausgegeben.

Der Baustein führt also eine einfache **Zusammenführung** (Multiplexing) der beiden Eingangsereignisse auf einen einzigen Ausgang durch. Es findet keine Auswahl oder Priorisierung statt – beide Ereignisse werden parallel und in der Reihenfolge ihres Auftretens weitergeleitet. Dies entspricht funktional einer **ODER-Verknüpfung** (OR) auf Ereignisebene.

Da keine Daten-Eingänge oder Auswahlparameter existieren, ist das Verhalten deterministisch und benötigt keine interne Zustandslogik.

Technische Besonderheiten

- **Adapterbasierte Ausgabe:** Der klassische Daten-Eingang K (zur Auswahl) und der separate Ereignis-Ausgang EO von E_MUX_2 wurden durch einen einzigen unidirektionalen AUI-Adapter ersetzt. Dadurch wird die physikalische Schnittstelle vereinfacht und die Kopplung an andere Bausteine über standardisierte Adapter ermöglicht.
- **Generischer Baustein:** Der FB ist als generisch deklariert (GenericClassName = 'GEN_E_MUX'), was bedeutet, dass er in verschiedenen Kontexten mit unterschiedlichen Adaptertypen wiederverwendet werden kann.
- **Keine Zustandsabhängigkeit:** Es gibt keine internen Zustände oder Variablen, die das Verhalten beeinflussen. Das macht den Baustein besonders robust und einfach zu analysieren.
- **Lizenz und Version:** Der Baustein unterliegt der Eclipse Public License 2.0 (EPL-2.0) und wurde für die Verwendung in der 4diac-IDE (61499-1 Annex A) entwickelt.

Zustandsübersicht

Der Baustein besitzt **keine** expliziten Zustände, da er kein prozedurales Verhalten aufweist. Die Ereignisweiterleitung erfolgt unmittelbar bei jedem Auftreten eines Eingangsereignisses. Es gibt weder Initialisierungs- noch Wartezustände; der FB ist immer bereit, Ereignisse zu verarbeiten.

Anwendungsszenarien

- **Zusammenführung von Sensorereignissen:** Zwei Sensoren (z.B. Endlage und Not-Aus) senden Ereignisse an den FB; der AUI-Adapter leitet diese an eine zentrale Steuerung weiter.
- **Adapterbasierte Erweiterung:** In Systemen, die bereits mit AUI-Adaptern arbeiten, kann dieser FB verwendet werden, um zwei unabhängige Ereignisquellen ohne zusätzliche Verschaltung auf einen bestehenden Adapterkanal zu legen.
- **Prototypenbau:** Durch die generische Struktur eignet sich der Baustein für schnelle Tests und Simulationen, bei denen Ereignisse ohne aufwendige Multiplexer-Logik gebündelt werden sollen.

Vergleich mit ähnlichen Bausteinen

Baustein	Ereignis-Ausgang	Daten-Eingang	Adapter-Ausgang	Auswahlsteuerung
E_MUX_2	Ja (EO)	Ja (K)	Nein	Ja, über K (0/1)
AUI_MUX_2	Nein	Nein	Ja (K)	Nein (reine OR-Verbindung)
E_OR (falls vorhanden)	Ja	Nein	Optional	Nein (ODER-Verknüpfung)

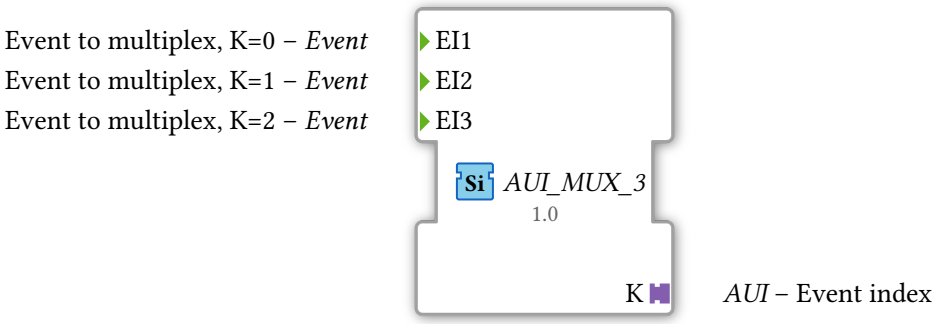
Der wesentliche Unterschied zum klassischen E_MUX_2 besteht darin, dass der AUI_MUX_2 keine Möglichkeit bietet, zwischen den Eingängen umzuschalten. Er leitet jedes ankommende Ereignis ungeachtet

einer Auswahl weiter. Wenn eine gezielte Auswahl (z.B. über einen Dateneingang oder ein Steuersignal) erforderlich ist, muss der `E_MUX_2` oder eine erweiterte Variante mit Adapter verwendet werden.

Fazit

Der **AUI_MUX_2** ist ein schlanker, ereignisgesteuerter Baustein, der zwei Ereignisquellen über eine einzige AUI-Adapter-Schnittstelle zusammenführt. Er besticht durch seine einfache Struktur, die vollständige Kompatibilität mit adapterbasierten Architekturen und das Fehlen interner Zustände. Seine Funktionalität beschränkt sich auf die reine Ereignisweitergabe (OR-Verknüpfung) – eine echte Multiplexer-Funktion mit Auswahlsteuerung wird durch diese Variante nicht angeboten. Daher ist er ideal für Anwendungen, bei denen mehrere Ereignisse ohne Priorisierung gebündelt und über einen gemeinsamen Adapter übertragen werden sollen. Für komplexe Multiplexer-Anforderungen mit Auswahl logik sollte auf den Standard-`E_MUX_2` zurückgegriffen werden.

AUI_MUX_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUI_MUX_3** implementiert einen Ereignis-Multiplexer mit drei Eingängen. Er stellt eine Erweiterung des Standardbausteins `E_MUX_3` dar, bei dem der selektierte Ereignisindex nicht über einen separaten Datenausgang `K`, sondern über einen unidirektionalen Adapter (Typ `adapter::types::unidirectional::AUI`) bereitgestellt wird. Dadurch kann das Ergebnis direkt an andere Bausteine mit kompatibler Adapterschnittstelle weitergegeben werden, ohne zusätzliche Verbindungen über Datenleitungen.

Schnittstellenstruktur

Ereignis-Eingänge

- **EI1** (Event): Ereignis, das den Index 0 auswählt.
- **EI2** (Event): Ereignis, das den Index 1 auswählt.
- **EI3** (Event): Ereignis, das den Index 2 auswählt.

Ereignis-Ausgänge

- Keine expliziten Ereignis-Ausgänge vorhanden. Das Ergebnis wird über den Adapter `K` kommuniziert.

Daten-Eingänge

- Keine.

Daten-Ausgänge

- Keine direkten Datenausgänge. Der ausgewählte Index wird über den Adapter `K` übertragen.

Adapter

- **K** (Plug, Typ `adapter::types::unidirectional::AUI`): Unidirektionaler Adapterausgang, der den Index (0, 1 oder 2) des zuletzt empfangenen Ereignisses als Datenwert bereitstellt. Die genaue Kodierung ist abhängig von der Definition des Adaptertyps.

Funktionsweise

Der Baustein wartet auf Ereignisse an den Eingängen `EI1`, `EI2` oder `EI3`. Sobald eines dieser Ereignisse eintritt, wird der zugehörige Index (0, 1 bzw. 2) intern gespeichert und über den Adapterausgang `K` ausgegeben. Der Adapter übernimmt dabei die Rolle des sonst üblichen Datenausgangs `K` sowie des Ereignisausgangs `EO` in einem einzigen Interface. Dadurch wird die Weitergabe des selektierten Ereignisses an nachgeschaltete Bausteine vereinfacht, die eine einheitliche Adapterschnittstelle verwenden.

Der Baustein ist als generischer Funktionsblock (GenericClassName `'GEN_E_MUX'`) konzipiert und kann daher in unterschiedlichen Umgebungen eingesetzt werden.

Technische Besonderheiten

- **Adapterbasierte Ausgabe**: Statt getrennter Event- und Datenausgänge wird ein unidirektionaler Adapter verwendet. Dies ermöglicht eine kompakte und standardisierte Anbindung an andere Bausteine mit demselben Adaptertyp.
- **Generischer Aufbau**: Durch die Zuweisung der generischen Klasse `GEN_E_MUX` ist der Baustein für verschiedene Konfigurationen wiederverwendbar.
- **Keine Datenabhängigkeit**: Die Auswahl des Index erfolgt rein über Ereignis-Eingänge; es werden keine zusätzlichen Datenwerte benötigt.
- **Kompatibilität**: Der Adaptertyp `adapter::types::unidirectional::AUI` ist Teil des Pakets `adapter::events::unidirectional` und muss in der Zielumgebung verfügbar sein.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten. Intern wird lediglich der zuletzt empfangene Index gespeichert und über den Adapter ausgegeben. Es gibt keine kontinuierliche Verarbeitung; die Ausgabe wird ausschließlich durch Ereignisse am Eingang aktualisiert.

Anwendungsszenarien

- **Ereignis-Routing**: Auswahl einer von drei Quellen oder Pfaden basierend auf einem eingehenden Ereignis.
- **Prioritätssteuerung**: Zuordnung einer Prioritätsstufe (0, 1, 2) zu einem Ereignis.
- **Protokollumsetzung**: Einsatz in Systemen, bei denen eine Adapterschnittstelle für die Kommunikation zwischen Bausteinen verwendet wird, um eine saubere Trennung von Daten- und Steuerebene zu erreichen.
- **Modulare Automatisierungslösungen**: Anbindung an einen übergeordneten Auswahlbaustein über einen standardisierten Adapter.

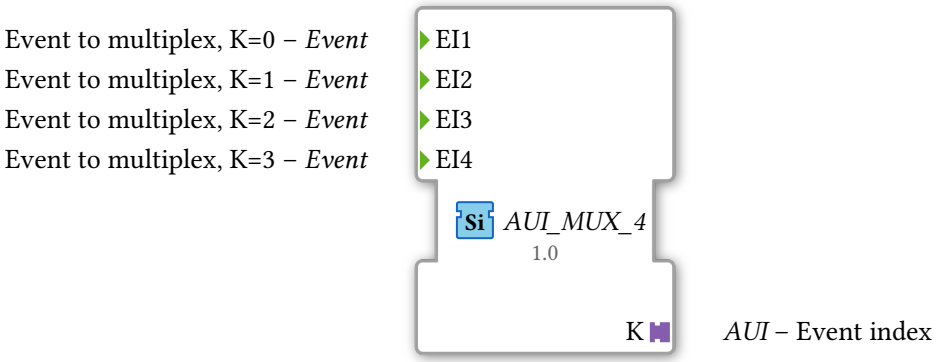
Vergleich mit ähnlichen Bausteinen

Der Standardbaustein **E_MUX_3** besitzt neben den drei Ereignis-Eingängen einen Ereignis-Ausgang `EO` und einen Datenausgang `K` (Integer), die separat verdrahtet werden müssen. **AUI_MUX_3** ersetzt diese beiden Ausgänge durch einen einzelnen Adapter `K`. Dadurch werden Verbindungen vereinfacht und die Schnittstelle wird einheitlicher, insbesondere wenn mehrere Bausteine über denselben Adaptertyp kommunizieren sollen. Die interne Funktionalität – Auswahl und Bereitstellung des Index – ist identisch, lediglich die Ausgabeschnittstelle unterscheidet sich.

Fazit

Der Funktionsblock **AUI_MUX_3** ist eine moderne, adapterbasierte Variante des klassischen Ereignis-Multiplexers. Er bietet eine kompakte und flexible Möglichkeit, Ereignisse zu selektieren und das Ergebnis über eine standardisierte Schnittstelle weiterzugeben. Insbesondere für Systeme, die auf Adapterkommunikation setzen, stellt er eine sinnvolle Erweiterung dar und erleichtert die Verschaltung von Automatisierungskomponenten.

AUI_MUX_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `AUI_MUX_4` ist ein Ereignis-Multiplexer, der aus vier Ereignis-Eingängen ein einzelnes Ereignis auswählt und über einen AUI-Adapter als Ausgang weiterleitet. Im Gegensatz zum klassischen `E_MUX_4`, der separate Ereignis-Ausgänge und einen Dateneingang für den Index verwendet, fasst `AUI_MUX_4` das ausgewählte Ereignis und den zugehörigen Index in einem einzigen AUI-Adapterkanal zusammen. Dadurch wird die Schnittstellenanzahl reduziert und eine kompakte, generische Ereignisverteilung ermöglicht.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
EI1	Event	Ereignis zum Multiplexen, K = 0
EI2	Event	Ereignis zum Multiplexen, K = 1
EI3	Event	Ereignis zum Multiplexen, K = 2
EI4	Event	Ereignis zum Multiplexen, K = 3

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

Name	Typ	Kommentar
K	`adapter::types::unidirectional::AUI`	Ereignisindex

Der Adapter `K` ist als Plug (Server) ausgelegt und dient als Ausgang für das multiplexierte Ereignis. Über den Adapter wird sowohl das Ereignis als auch der zugehörige Index (0–3) übertragen.

Funktionsweise

Der Baustein `AUI_MUX_4` verhält sich wie ein Ereignis-Multiplexer: Wird an einem der vier Ereignis-Eingänge `EI1` bis `EI4` ein Ereignis empfangen, so wird dieses Ereignis über den Adapter `K` ausgegeben. Der eindeutige Index des Eingangs (0 für `EI1`, 1 für `EI2`, 2 für `EI3`, 3 für `EI4`) wird dabei als Teil der AUI-Kommunikation mitgesendet. Dies ermöglicht dem Empfänger, die Herkunft des Ereignisses zu identifizieren, ohne separate Leitungen für jeden Kanal zu benötigen.

Der Baustein ist als generischer Funktionsbaustein definiert (GenericClassName `'GEN_E_MUX'`), was eine flexible Wiederverwendung in verschiedenen Kontexten erlaubt. Die eigentliche Auswahl- und Übertragungslogik wird durch die generische Implementierung bereitgestellt, während die konkrete Ereignisanzahl und der AUI-Typ über die Schnittstelle festgelegt sind.

Technische Besonderheiten

- **Verwendung eines AUI-Adapters**: Statt getrennter Ereignis-Ausgänge und eines Dateneingangs wird der Index über den AUI-Adapter `K` transportiert. Dadurch wird die Schnittstellenanzahl auf ein Minimum reduziert und die Datenintegrität durch die Adapterkapselung erhöht.
- **Unidirektionaler Adapter**: Der Typ `unidirectional` bedeutet, dass der Adapter nur in eine Richtung Daten sendet (hier als Ausgang). Dies vereinfacht die Verwendung und verhindert unbeabsichtigte Rückkopplungen.
- **Generische Instanziierung**: Durch das Attribut `GenericClassName` kann der Baustein als generische Vorlage für ähnliche Multiplexer-Konfigurationen dienen, z. B. für andere Kanalanzahlen.
- **Keine Ereignis-Ausgänge**: Die Weiterleitung erfolgt ausschließlich über den Adapter, was die korrekte Handhabung des Ereignisses und des Index in einem Paket sicherstellt.

Zustandsübersicht

Da `AUI_MUX_4` als ereignisgesteuerter Baustein ohne explizite Zustandsmaschine (ECC) implementiert ist, erfolgt die Verarbeitung direkt bei Auftreten eines Ereignisses an einem der Eingänge. Es gibt keine internen Zustände; jeder Eingabeereignis führt zu einer unmittelbaren Ausgabe über den Adapter. Die Logik ist rein reaktiv und ereignisbasiert.

Anwendungsszenarien

- **Ereignisbündelung**: Mehrere Ereignisquellen (z. B. Sensoren oder Steuersignale) können über separate Eingänge angebunden werden. Das ausgewählte Ereignis wird mitsamt seines Index über einen einzigen AUI-Kanal an eine übergeordnete Steuerung oder ein Kommunikationsmodul übertragen.
- **Schnittstellenreduzierung**: In Systemen mit begrenzten I/O-Ressourcen ersetzt `AUI_MUX_4` mehrere separate Ereignisausgänge durch einen einzigen Adapter, wodurch Verdrahtung und Konfiguration vereinfacht werden.
- **Indexbasierte Ereignisweitergabe**: Wenn nachfolgende Bausteine den Ursprung eines Ereignisses benötigen, liefert `AUI_MUX_4` diesen Index direkt über den Adapter, ohne zusätzliche Datenleitungen.

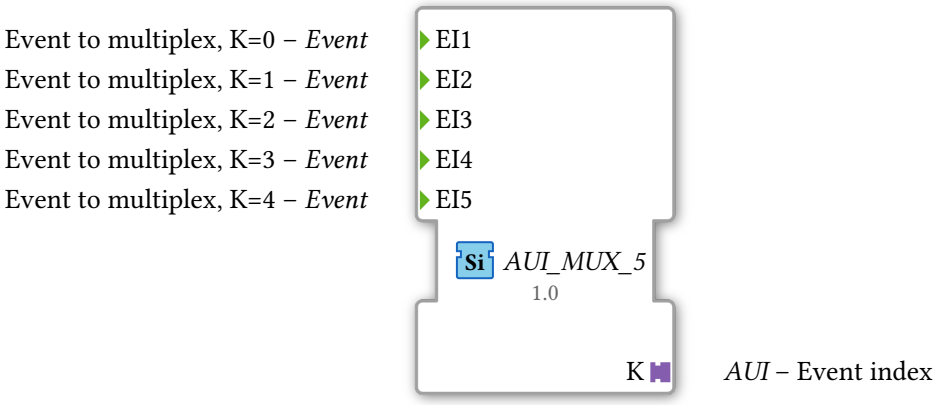
Vergleich mit ähnlichen Bausteinen

Der Standard-Funktionsbaustein `E_MUX_4` besitzt vier Ereignis-Eingänge, vier Ereignis-Ausgänge und einen Dateneingang `K` zur Auswahl des aktiven Kanals. Im Gegensatz dazu verwendet `AUI_MUX_4` einen einzelnen AUI-Adapter als Ausgang, der das ausgewählte Ereignis und den Index zusammen transportiert. Dies reduziert die Anzahl der Ausgänge erheblich und bietet eine kompaktere, integrierte Lösung. Der Preis dafür ist eine spezifische Adapterabhängigkeit und eine weniger intuitive Schnittstellenstruktur, da kein separater Index-Dateneingang vorhanden ist – der Index wird implizit über die AUI-Kommunikation übermittelt. Bausteine wie `GEN_E_MUX` (generische Version) können ähnliche Funktionalitäten ohne AUI-Adapter realisieren, jedoch mit höherer Ausgangsanzahl.

Fazit

`AUI_MUX_4` ist ein spezialisierter Ereignis-Multiplexer, der die Vorteile der Adaptertechnik nutzt, um eine kompakte und effiziente Ereignisweitergabe mit Indexinformation zu realisieren. Durch den Verzicht auf separate Ausgänge wird die Schnittstelle vereinfacht und die Integration in komplexe Automatisierungssysteme erleichtert. Die generische Definition ermöglicht zudem eine flexible Anpassung an unterschiedliche Anforderungen. Für Anwendungen, bei denen eine direkte Auswahl über einen Dateneingang bevorzugt wird, ist der klassische `E_MUX_4` weiterhin eine geeignete Alternative. Insgesamt stellt `AUI_MUX_4` jedoch eine moderne und platzsparende Lösung für ereignisbasierte Multiplexaufgaben dar.

AUI_MUX_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUI_MUX_5** ist ein Ereignis-Multiplexer, der fünf Ereignis-Eingänge (EI1–EI5) besitzt und über einen Adapter-Ausgang (Typ `adapter::types::unidirectional::AUI`) ein ausgewähltes Ereignis an eine nachgeschaltete Einheit weitergibt. Er stellt eine Adapter-basierte Variante des klassischen E_MUX_5 dar und ermöglicht eine flexible, ereignisgesteuerte Auswahl ohne separate Ereignis-Ausgänge.

Schnittstellenstruktur

Der FB besitzt ausschließlich Ereignis-Eingänge und einen Adapter-Plug. Es sind keine Daten-Eingänge, Daten-Ausgänge oder Ereignis-Ausgänge vorhanden.

Ereignis-Eingänge

- **EI1** – Ereignis, das bei Auswahlindex 0 (K=0) weitergeleitet wird.
- **EI2** – Ereignis, das bei Auswahlindex 1 (K=1) weitergeleitet wird.
- **EI3** – Ereignis, das bei Auswahlindex 2 (K=2) weitergeleitet wird.
- **EI4** – Ereignis, das bei Auswahlindex 3 (K=3) weitergeleitet wird.
- **EI5** – Ereignis, das bei Auswahlindex 4 (K=4) weitergeleitet wird.

Ereignis-Ausgänge

Keine vorhanden. Die Ereignisweiterleitung erfolgt ausschließlich über den Adapter.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

- **K** (Plug) – Adapter vom Typ `adapter::types::unidirectional::AUI`. Über diesen Adapter wird das ausgewählte Ereignis ausgegeben und gegebenenfalls der aktuelle Auswahlindex (0–4) bereitgestellt.

Funktionsweise

Der Funktionsblock arbeitet als Ereignis-Multiplexer. Der über den Adapter **K** bereitgestellte Auswahlindex bestimmt, welcher der fünf Ereignis-Eingänge bei einem anliegenden Ereignis aktiviert und über den Adapter nach außen weitergegeben wird.

- Wenn K den Wert 0 hat, wird ein an EI1 ankommendes Ereignis über den Adapter ausgegeben.
- Bei K=1 wird EI2 durchgeschaltet, bei K=2 EI3, bei K=3 EI4 und bei K=4 EI5.

Ein Ereignis an einem nicht ausgewählten Eingang wird ignoriert. Der Adapter stellt dabei sowohl das Ereignis als auch den verwendeten Index bereit.

Technische Besonderheiten

- **Generische Implementierung**: Der Baustein ist als generischer FB mit dem Klassennamen `GEN_E_MUX` definiert, wodurch eine parametrisierbare Nutzung möglich ist.
- **Adapter-basierte Ausgabe**: Statt diskreter Ereignis-Ausgänge (EO1–EO5) wird ein einzelner unidirektionaler Adapter verwendet. Dies vereinfacht die Verbindung zu Schnittstellen, die einen einheitlichen Event-Kanal erwarten.
- **Keine Datenhaltung**: Der FB ist rein ereignisgesteuert und besitzt keine internen Datenzustände.

Zustandsübersicht

Da der Baustein rein ereignisbasiert arbeitet, existiert kein expliziter Zustandsautomat. Der interne Zustand wird ausschließlich durch den aktuellen Auswahlindex K bestimmt, der über den Adapter gelesen wird. Die Verarbeitung erfolgt ereignisgetrieben: Jedes ankommende Eingangsereignis wird abhängig von K entweder weitergeleitet oder verworfen.

Anwendungsszenarien

- **Ereignis-Routing**: Auswahl einer von mehreren Ereignisquellen (z. B. Sensoren, Alarme) und Weiterleitung über einen gemeinsamen Adapterausgang an eine Verarbeitungseinheit.
- **Modulare Systemkopplung**: Einsatz in Systemen, bei denen eine einheitliche Adapter-Schnittstelle (z. B. für Bus-Kommunikation) verwendet wird und mehrere Ereignisquellen dynamisch zugeschaltet werden sollen.
- **Test- und Simulationsumgebungen**: Gezielte Steuerung, welches Ereignis aktiviert wird, ohne die Verdrahtung zu ändern.

Vergleich mit ähnlichen Bausteinen

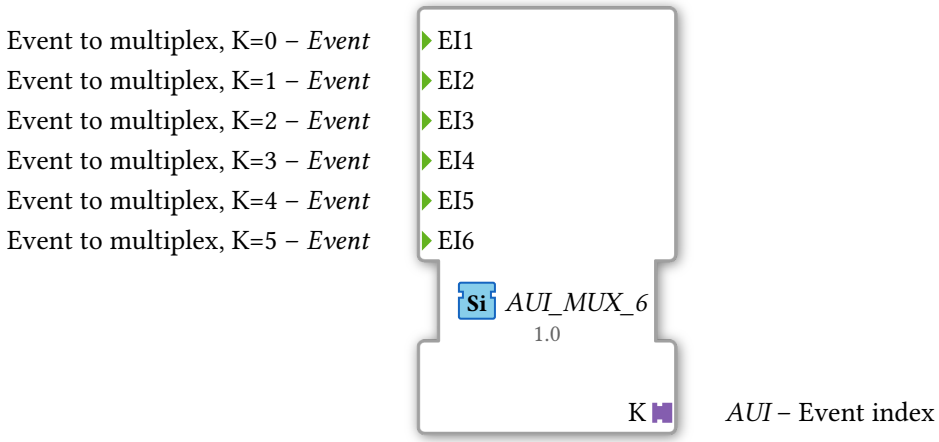
- **E_MUX_5** (Standard): Besitzt fünf Ereignis-Eingänge und fünf separate Ereignis-Ausgänge (EO1–EO5). Die Auswahl erfolgt über einen Daten-Eingang K.
  Unterschied: AUI_MUX_5 verwendet einen Adapter-Ausgang statt einzelner Ereignis-Ausgänge. Dadurch wird die Schnittstelle kompakter und besser für Verbindungen mit Adapter-basierten Komponenten geeignet.
- **AUI_MUX_N** (verallgemeinert): Kann mehrere Eingänge besitzen; AUI_MUX_5 ist eine spezielle Instanz mit fünf Eingängen.

Der Hauptvorteil der Adapter-Variante liegt in der einheitlichen Schnittstellengestaltung, die insbesondere dann nützlich ist, wenn adäquate Adapter-Typen im Projekt bereits verwendet werden.

Fazit

Der **AUI_MUX_5** ist ein spezialisierter Ereignis-Multiplexer, der die Vorteile einer Adapter-basierten Ausgabe mit einer klaren, fünf Eingängen umfassenden Ereignisauswahl verbindet. Durch die Reduktion auf einen Adapterausgang wird die Verdrahtung vereinfacht und die Wiederverwendbarkeit in Adapter-orientierten Architekturen erhöht. Die generische Ausführung erlaubt zudem eine flexible Anpassung an unterschiedliche Anforderungen.

AUI_MUX_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AUI_MUX_6** ist ein Ereignis-Multiplexer, der sechs unabhängige Ereignis-Eingänge besitzt und über einen **AUI-Adapter** (unidirektional) das empfangene Ereignis zusammen mit dem zugehörigen Index (0–5) als Ausgabe bereitstellt. Er stellt eine generische Variante des klassischen E_MUX_6 dar, bei dem der konventionelle Ereignis-Ausgang und der Datenausgang für den Index durch einen einzigen Adapter ersetzt werden. Dadurch wird die Schnittstelle vereinheitlicht und die Weiterverarbeitung in IEC-61499-Systemen erleichtert.

Schnittstellenstruktur

Ereignis-Eingänge

- **EI1** (Event): Ereignis, das auf Kanal 0 gemultiplext wird.
- **EI2** (Event): Ereignis, das auf Kanal 1 gemultiplext wird.
- **EI3** (Event): Ereignis, das auf Kanal 2 gemultiplext wird.
- **EI4** (Event): Ereignis, das auf Kanal 3 gemultiplext wird.
- **EI5** (Event): Ereignis, das auf Kanal 4 gemultiplext wird.
- **EI6** (Event): Ereignis, das auf Kanal 5 gemultiplext wird.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

- **K** (Plug, Typ `adapter::types::unidirectional::AUI`): Unidirektionaler Adapter, der das empfangene Ereignis sowie den dazugehörigen Index (0–5) als gebündelte Schnittstelle nach außen führt.

Funktionsweise

Der FB arbeitet als ereignisgesteuerter Multiplexer. Trifft ein Ereignis an einem der sechs Eingänge **EI1** bis **EI6** ein, so wird dieses Ereignis über den Adapter **K** ausgegeben. Gleichzeitig wird der entsprechende Index (0 für EI1, 1 für EI2, …, 5 für EI6) über denselben Adapter übermittelt. Dadurch kann der angeschlossene Empfänger sowohl das Ereignis als auch die Information, welcher Eingang das Ereignis ausgelöst hat, in einem Schritt verarbeiten. Es findet keine Priorisierung oder Verzögerung statt; jede Aktivierung eines Eingangs erzeugt sofort eine Ausgabe am Adapter.

Technische Besonderheiten

- **Generische Implementierung**: Der Baustein ist als generische Klasse (`GenericClassName = 'GEN_E_MUX'`) definiert und kann für eine beliebige Anzahl von Eingängen instanziiert werden. Die vorliegende Instanz ist auf 6 Eingänge festgelegt.
- **Adapter-basierte Schnittstelle**: Der Einsatz eines Adapters anstelle separater Ereignis- und Datenausgänge vereinfacht die Verdrahtung in Systemen, die bereits Adapter für die Kommunikation verwenden.
- **Unidirektionaler Adapter**: Der Typ AUI ist als unidirektional definiert, d.h. er überträgt nur in eine Richtung (vom FB zum Empfänger).
- **Keine Zustandshaltung**: Der FB ist zustandslos – er speichert keine internen Werte und reagiert rein ereignisgetrieben.

Zustandsübersicht

Da es sich um einen reinen Ereignis-Multiplexer ohne Zustandsspeicher handelt, existiert keine interne Zustandsmaschine. Der FB wechselt zwischen keinen Zuständen und führt bei jedem Ereignis am Eingang sofort die Multiplex-Funktion aus.

Anwendungsszenarien

- **Zusammenführung von Ereignissen**: Mehrere Quellen (z. B. Sensoren, Alarme) erzeugen Ereignisse, die auf einem gemeinsamen Kommunikationspfad übertragen werden sollen – der Adapter bündelt Ereignis und Quellenidentifikation.
- **Steuerung von Ablaufsteuerungen**: In Fertigungsanlagen können verschiedene Maschinenzustände als Ereignisse gesammelt und an eine übergeordnete Steuerung weitergeleitet werden.
- **Datenvorverarbeitung**: Der Baustein kann als Eingangspuffer für zentrale Verarbeitungseinheiten dienen, die mehrere Ereigniskanäle mit einer einheitlichen Schnittstelle ansprechen möchten.

Vergleich mit ähnlichen Bausteinen

Klassische Vertreter wie der **E_MUX_6** besitzen einen Ereignis-Ausgang (EO) und einen Datenausgang (K) für den Index. Der **AUI_MUX_6** ersetzt diese beiden Ausgänge durch einen einzigen AUI-Adapter, der beide Informationen zusammenführt. Vorteile:

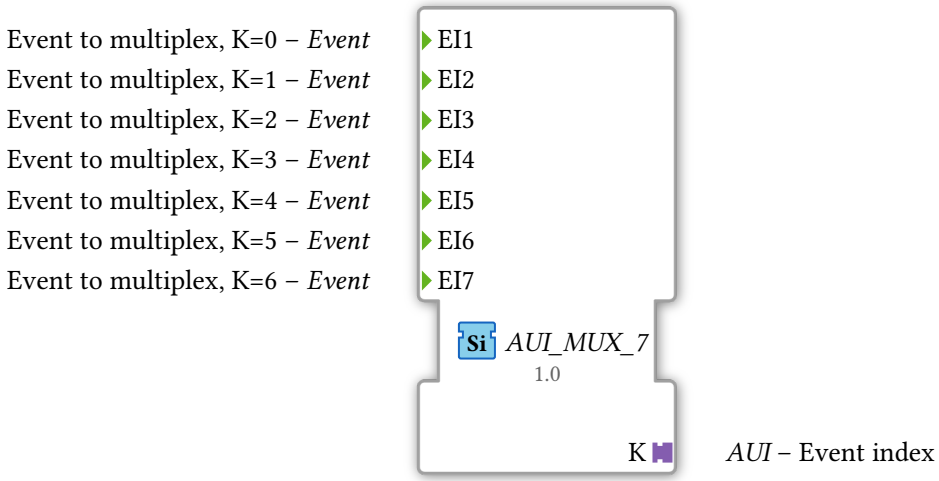
- Reduzierung der Verbindungsleitungen,
- einheitliche Schnittstellenbeschreibung,
- bessere Kompatibilität mit adapterbasierten Kommunikationsmustern in IEC 61499.

Nachteile gegenüber dem klassischen Baustein könnten darin liegen, dass der Empfänger zwingend einen passenden Adapter aufweisen muss, was bei einfachen Systemen mit getrennten Signalen unnötig wäre.

Fazit

AUI_MUX_6 stellt eine moderne, adapterbasierte Lösung für die Multiplexierung von Ereignissen dar. Durch die Bündelung von Ereignis und Index in einer einzigen Schnittstelle wird die Systemarchitektur vereinfacht und die Wiederverwendbarkeit erhöht. Die generische Natur des Bausteins ermöglicht eine flexible Anpassung an unterschiedliche Anforderungen, während die zustandslose Arbeitsweise eine präzise und zeitnahe Reaktion auf externe Ereignisse gewährleistet. Damit ist er eine wertvolle Komponente in ereignisgesteuerten Automatisierungssystemen.

AUI_MUX_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AUI_MUX_7 ist ein Ereignis-Multiplexer, der sieben Ereignis-Eingänge besitzt und das jeweils empfangene Ereignis über einen AUI-Adapter weitergibt. Anstatt des klassischen Ausgangs bestehend aus einem Ereignis-Ausgang (EO) und einem Datenausgang für den Index (K), bündelt der Adapter beide Informationen in einer standardisierten Schnittstelle. Dadurch wird die Verkabelung vereinfacht und eine einheitliche Kommunikationsstruktur ermöglicht.

Schnittstellenstruktur

Ereignis-Eingänge

- `EI1` – Ereignis, dem der Index 0 zugeordnet ist

- `EI2` – Ereignis, dem der Index 1 zugeordnet ist
- `EI3` – Ereignis, dem der Index 2 zugeordnet ist
- `EI4` – Ereignis, dem der Index 3 zugeordnet ist
- `EI5` – Ereignis, dem der Index 4 zugeordnet ist
- `EI6` – Ereignis, dem der Index 5 zugeordnet ist
- `EI7` – Ereignis, dem der Index 6 zugeordnet ist

Ereignis-Ausgänge

- Keine vorhanden. Die Ereignisausgabe erfolgt ausschließlich über den Adapter.

Daten-Eingänge

- Keine vorhanden.

Daten-Ausgänge

- Keine vorhanden.

Adapter

- **Plug `K`** vom Typ `adapter::types::unidirectional::AUI` – Dieser Adapter dient als Ausgangsschnittstelle. Er überträgt das empfangene Ereignis sowie den zugehörigen Index (1 bis 7) gebündelt an einen passenden Gegenspieler.

Funktionsweise

Der AUI_MUX_7 arbeitet als selektiver Weiterleiter. Sobald an einem der Ereignis-Eingänge (`EI1` … `EI7`) ein Ereignis auftritt, wird dieses über den AUI-Adapter `K` ausgegeben. Parallel dazu liefert der Adapter den Index des auslösenden Eingangs, sodass der Empfänger eindeutig erkennen kann, welche Quelle das Ereignis erzeugt hat. Es wird angenommen, dass nur ein Ereignis zu einem Zeitpunkt aktiv ist; bei gleichzeitigem Auftreten erfolgt die Auswahl gemäß der internen Logik des generischen Bausteins (im Normalfall priorisiert nach der niedrigsten Eingangsnummer).

Technische Besonderheiten

- Der Baustein ist als generischer Funktionsbaustein angelegt, erkennbar an den Attributen `eclipse4diac::core::GenericClassName` mit Wert `'GEN_E_MUX'` und `eclipse4diac::core::TypeHash` mit leerem Wert.
- Die Verwendung eines AUI-Adapters anstatt separater EO-/K-Ausgänge reduziert die Anzahl benötigter Verbindungen und vereinheitlicht die Schnittstelle.
- Die Anzahl der Eingänge ist fest auf sieben konfiguriert, kann aber durch Anpassung der Generik auf andere Werte erweitert oder reduziert werden.

Zustandsübersicht

Der AUI_MUX_7 besitzt keine expliziten internen Zustände. Er ist rein reaktiv und leitet jedes ankommende Ereignis unmittelbar an den Adapter weiter. Es findet keine Speicherung oder Verzögerung statt.

Anwendungsszenarien

- In Automatisierungssystemen, in denen mehrere Ereignisquellen (z. B. Sensoren) auf einen gemeinsamen Verarbeitungsbaustein geführt werden sollen, wobei der Index die Quellenidentifikation liefert.
- Bei modularen Steuerungen, die über genormte AUI-Schnittstellen kommunizieren und dadurch eine einfache Austauschbarkeit der Komponenten ermöglichen.
- Überall dort, wo eine saubere Trennung zwischen Ereignisübertragung und zusätzlicher Information (Index) gewünscht ist, ohne separate Leitungen zu verdrahten.

Vergleich mit ähnlichen Bausteinen

- **Vergleich mit dem klassischen `E_MUX`**: Der AUI_MUX_7 ersetzt die separate Ereignis- und Datenausgabe durch einen einzelnen AUI-Adapter. Das reduziert die Verdrahtung und bietet eine kompaktere Schnittstelle.
- **Gegenüber einem Event-Fan-Out** (z. B. E_SPLIT): Der Multiplexer wählt nur einen Eingang aus und gibt nicht alle Ereignisse unverändert weiter, sondern ergänzt die Indexinformation.

Fazit

Der AUI_MUX_7 ist ein praktischer Ereignis-Multiplexer, der durch die Verwendung eines AUI-Adapters eine moderne und standardisierte Verbindungstechnik unterstützt. Er vereinfacht die Verdrahtung, verbessert die Modularität von Steuerungssystemen und bietet eine klare Zuordnung eingehender Ereignisse über den Index. Dank seiner generischen Struktur lässt er sich leicht an unterschiedliche Anforderungen anpassen.

AUI_MUX_8

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUI_MUX_8** ist ein Ereignis-Multiplexer, der acht Eingangsereignisse (EI1 bis EI8) zu einem einzigen Ausgangsereignis bündelt. Das ausgewählte Ereignis wird nicht über einen klassischen Event-Ausgang (EO) ausgegeben, sondern über einen **Adapter** vom Typ `adapter::types::unidirectional::AUI` bereitgestellt. Dies ermöglicht eine flexible und lose Kopplung zwischen dem Multiplexer und nachgeschalteten Funktionsbausteinen oder Subapplikationen.

Der Baustein ist als generischer Funktionsblock implementiert und basiert auf der Kernklasse `GEN_E_MUX`, die in der 4diac-IDE für verschiedene Multiplexer-Varianten verwendet wird.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar
`EI1`	Event	Ereignis für K = 0
`EI2`	Event	Ereignis für K = 1
`EI3`	Event	Ereignis für K = 2
`EI4`	Event	Ereignis für K = 3
`EI5`	Event	Ereignis für K = 4
`EI6`	Event	Ereignis für K = 5
`EI7`	Event	Ereignis für K = 6
`EI8`	Event	Ereignis für K = 7

Ereignis-Ausgänge

Keine direkten Event-Ausgänge vorhanden. Die Ausgabe erfolgt über den Adapter `K`.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Adapter	Typ	Kommentar
K	adapter::types::unidirectional::AUI	Ereignis-Index (Auswahl des Eingangsereignisses)

Der Adapter K übernimmt eine Doppelfunktion:

- Er empfängt den Auswahlindex (0–7) vom verbundenen Kommunikationspartner.
- Über ihn wird das ausgewählte Eingangsereignis als Adapter-Ereignis an den Partner weitergeleitet.

Funktionsweise

Der **AUI_MUX_8** wartet auf eines der acht Ereignis-Eingänge EI1–EI8. Gleichzeitig muss über den Adapter K ein gültiger Index (0–7) bereitgestellt werden. Sobald ein Ereignis an einem der Eingänge auftritt, prüft der Funktionsblock den aktuellen Indexwert und leitet das entsprechende Ereignis über den Adapter K an den angeschlossenen Baustein weiter.

Die Zuordnung ist:

- EI1 ⟶ Index 0
- EI2 ⟶ Index 1
- ...
- EI8 ⟶ Index 7

Der Baustein arbeitet **ereignisgesteuert**: Nur wenn ein Ereignis an einem Eingang anliegt und der Index gültig ist, wird das Ereignis zum Ausgang transportiert. Ist der Index ungültig (z.B. außerhalb 0–7) oder fehlt ein Ereignis, wird nichts ausgelöst.

Technische Besonderheiten

- **Adapter statt Ausgang**: Statt eines klassischen Event-Ausgangs wird ein unidirektionaler Adapter (AUI) verwendet. Dadurch kann der Multiplexer direkt mit anderen Bausteinen über Adapter-Schnittstellen verbunden werden, ohne zusätzliche Leitungen zu verlegen.
- **Generische Basis**: Der FB nutzt die generische Klasse GEN_E_MUX und ist über das Attribut eclipse4diac::core::GenericClassName spezifiziert.
- **Paketzuordnung**: Die Compiler-Info verweist auf das Paket adapter::events::unidirectional, das für die Adapter-Datentypen und Schnittstellen verantwortlich ist.
- **Versionierung**: Der Baustein wurde mehrfach überarbeitet; aktuelle Version 3.0 (2025-04-14) sowie frühere Versionen von fortiss GmbH und HR Agrartechnik GmbH sind dokumentiert.

Zustandsübersicht

Der Funktionsblock besitzt keine expliziten Zustände. Sein Verhalten ist rein ereignisbasiert:

- **Bereitschaft**: Kein Eingangsereignis aktiv, keine Ausgabe.
- **Selektion und Weiterleitung**: Eines der Eingangsereignisse trifft ein und wird gemäß Index über den Adapter ausgegeben.

Eine Zustandsautomaten-Darstellung entfällt daher.

Anwendungsszenarien

- **Ereignis-Routing in verteilten Systemen**: Auswahl verschiedener Ereignisquellen (z.B. Sensoren, Alarme) und Weiterleitung an eine zentrale Verarbeitungslogik.
- **Adapterbasierte Kommunikation**: Einbindung in Steuerungssysteme, bei denen Adapter als standardisierte Schnittstellen zwischen Hardware und Software verwendet werden.
- **Modularisierung**: Einsatz in Subapplikationen, um mehrere asynchrone Ereignisse zu einem kontrollierten Datenstrom zu bündeln.

Vergleich mit ähnlichen Bausteinen

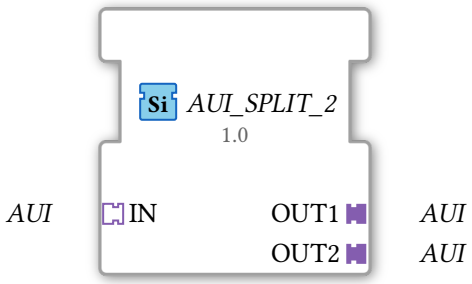
Im Gegensatz zum klassischen **E_MUX8**, der einen dedizierten Event-Ausgang `EO` und einen Daten-Eingang `K` besitzt, verwendet **AUI_MUX_8** einen Adapter für die Auswahl und Ausgabe. Das bietet:

- **Flexible Topologien** durch Adapterverbindungen.
- **Reduktion von Verdrahtungsaufwand** (kein separater Datenkanal für den Index notwendig, da der Index direkt über den Adapter übertragen wird).
- **Einschränkung**: Die Auswahl muss über den Adapter erfolgen, was bei sehr einfachen Anwendungen ohne Adapter-Infrastruktur weniger geeignet ist.

Fazit

Der Funktionsblock **AUI_MUX_8** ist eine moderne, adapterbasierte Variante eines Ereignis-Multiplexers. Er eignet sich besonders für systeme, in denen Kommunikationsstrukturen über Adapter standardisiert werden und eine flexible Anbindung an unterschiedliche Teilnehmer gefordert ist. Durch die Verwendung der generischen Basisklasse ist eine einfache Wiederverwendung und Anpassung in der 4diac-IDE gewährleistet.

AUI_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUI_SPLIT_2** verteilt ein von einem AUI-Adapter (Socket) eingehendes Signal unverändert auf zwei identische AUI-Adapter (Plugs). Er wird als generischer Baustein (GenericClassName = `'GEN_AUI_SPLIT'`) ausgeliefert und ermöglicht die einfache Aufteilung eines unidirektionalen AUI-Datenstroms auf zwei unabhängige Senken.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine direkten Dateneingänge vorhanden. Der Datentransport erfolgt ausschließlich über die Adapter.

Daten-Ausgänge

Keine direkten Datenausgänge vorhanden. Der Datentransport erfolgt ausschließlich über die Adapter.

Adapter

Name	Typ	Richtung / Rolle
`IN`	`adapter::types::unidirectional::AUI`	Socket (Eingang)
`OUT1`	`adapter::types::unidirectional::AUI`	Plug (Ausgang 1)
`OUT2`	`adapter::types::unidirectional::AUI`	Plug (Ausgang 2)

Alle drei Adapter sind vom Typ **AUI** und arbeiten unidirektional. Der Socket `IN` empfängt Daten, die Plugs `OUT1` und `OUT2` senden die gleichen Daten weiter.

Funktionsweise

Der Baustein leitet jedes ankommende AUI-Signal am Socket `IN` gleichzeitig an beide Plugs `OUT1` und `OUT2` weiter. Es findet keine Verarbeitung, Filterung oder Pufferung der Daten statt – die Daten werden **1:1 kopiert und gesendet**. Die Aufteilung erfolgt streng deterministisch und ohne zusätzliche Verzögerung (abgesehen von der Laufzeit der internen Verbindung).

Technische Besonderheiten

- **Generischer Typ**: Der FB wird durch das Attribut `GenericClassName` als `'GEN_AUI_SPLIT'` identifiziert und kann durch Type-Hash-Mechanismen (`TypeHash` aktuell leer) zur Laufzeit instanziiert werden.
- **Keine ECC (Execution Control Chart)**: Da keine ereignisgesteuerte Logik vorhanden ist, besitzt der Baustein keine Zustandsmaschine. Die Weiterleitung erfolgt rein datengetrieben.
- **EPL-2.0 Lizenziert**: Der Baustein wird unter der Eclipse Public License 2.0 bereitgestellt.

Zustandsübersicht

Der AUI_SPLIT_2 verfügt über **keine Zustandsmaschine**. Es gibt keine internen Zustände oder Übergänge – das Signal wird kontinuierlich und ohne Taktung weitergeleitet.

Anwendungsszenarien

- **Signalverteilung**: Ein AUI-Datenstrom muss parallel an zwei unabhängige Empfänger oder Subsysteme übergeben werden (z. B. zwei Controller, Anzeigen oder Aktoren).
- **Redundanz**: Identische Daten sollen auf zwei Pfaden bereitgestellt werden, um Failover-Mechanismen zu unterstützen.
- **Protokoll-Multicasting**: Ein unidirektionaler AUI-Stream wird an mehrere Verbraucher verteilt, ohne dass der Sender mehrfach angesprochen werden muss.

Vergleich mit ähnlichen Bausteinen

- **AUI_SPLIT_N**: Erweiterte Variante, die einen Eingang auf N Ausgänge aufteilt (z. B. 1:3, 1:4). Der AUI_SPLIT_2 ist der einfachste Vertreter dieser Familie.
- **AUI_MERGE_2**: Führt zwei AUI-Eingänge zu einem Ausgang zusammen (ggf. mit Arbitration). Der Splitter arbeitet in die entgegengesetzte Richtung.
- **AUI_PASS**: Ein reiner 1:1-Durchgangsadapter ohne Verzweigung.
- `AUI_SPLIT_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

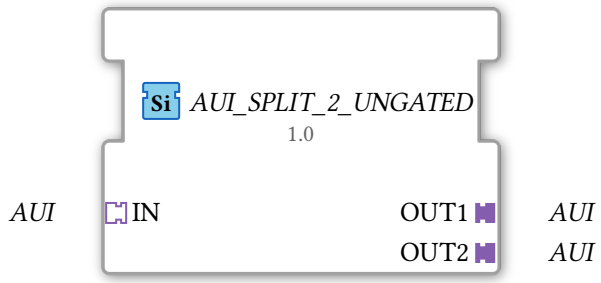
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der **AUI_SPLIT_2** ist ein minimalistischer, aber nützlicher Funktionsblock zur Aufteilung eines unidirektionalen AUI-Signals auf zwei Ausgänge. Dank seiner generischen Implementierung und dem Verzicht auf komplexe Logik eignet er sich ideal für Szenarien, in denen eine saubere, verlustfreie Vervielfältigung von Datenströmen erforderlich ist.

AUI_SPLIT_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUI_SPLIT_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUI_SPLIT_2_UNGATED** verteilt ein von einem AUI-Adapter (Socket) eingehendes Signal unverändert auf zwei identische AUI-Adapter (Plugs). Er wird als generischer Baustein (GenericClassName = `'GEN_AUI_SPLIT'`) ausgeliefert und ermöglicht die einfache Aufteilung eines unidirektionalen AUI-Datenstroms auf zwei unabhängige Senken.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine direkten Dateneingänge vorhanden. Der Datentransport erfolgt ausschließlich über die Adapter.

Daten-Ausgänge

Keine direkten Datenausgänge vorhanden. Der Datentransport erfolgt ausschließlich über die Adapter.

Adapter

Name	Typ	Richtung / Rolle
`IN`	`adapter::types::unidirectional::AUI`	Socket (Eingang)
`OUT1`	`adapter::types::unidirectional::AUI`	Plug (Ausgang 1)
`OUT2`	`adapter::types::unidirectional::AUI`	Plug (Ausgang 2)

Alle drei Adapter sind vom Typ **AUI** und arbeiten unidirektional. Der Socket `IN` empfängt Daten, die Plugs `OUT1` und `OUT2` senden die gleichen Daten weiter.

Funktionsweise

Der Baustein leitet jedes ankommende AUI-Signal am Socket `IN` gleichzeitig an beide Plugs `OUT1` und `OUT2` weiter. Es findet keine Verarbeitung, Filterung oder Pufferung der Daten statt – die Daten werden **1:1 kopiert und gesendet**. Die Aufteilung erfolgt streng deterministisch und ohne zusätzliche Verzögerung (abgesehen von der Laufzeit der internen Verbindung).

Technische Besonderheiten

- **Generischer Typ**: Der FB wird durch das Attribut `GenericClassName` als `'GEN_AUI_SPLIT'` identifiziert und kann durch Type-Hash-Mechanismen (`TypeHash` aktuell leer) zur Laufzeit instanziiert werden.
- **Keine ECC (Execution Control Chart)**: Da keine ereignisgesteuerte Logik vorhanden ist, besitzt der Baustein keine Zustandsmaschine. Die Weiterleitung erfolgt rein datengetrieben.
- **EPL-2.0 Lizenziert**: Der Baustein wird unter der Eclipse Public License 2.0 bereitgestellt.

Zustandsübersicht

Der AUI_SPLIT_2_UNGATED verfügt über **keine Zustandsmaschine**. Es gibt keine internen Zustände oder Übergänge – das Signal wird kontinuierlich und ohne Taktung weitergeleitet.

Anwendungsszenarien

- **Signalverteilung**: Ein AUI-Datenstrom muss parallel an zwei unabhängige Empfänger oder Subsysteme übergeben werden (z. B. zwei Controller, Anzeigen oder Aktoren).
- **Redundanz**: Identische Daten sollen auf zwei Pfaden bereitgestellt werden, um Failover-Mechanismen zu unterstützen.
- **Protokoll-Multicasting**: Ein unidirektionaler AUI-Stream wird an mehrere Verbraucher verteilt, ohne dass der Sender mehrfach angesprochen werden muss.

Vergleich mit ähnlichen Bausteinen

- **AUI_SPLIT_N**: Erweiterte Variante, die einen Eingang auf N Ausgänge aufteilt (z. B. 1:3, 1:4). Der AUI_SPLIT_2_UNGATED ist der einfachste Vertreter dieser Familie.
- **AUI_MERGE_2**: Führt zwei AUI-Eingänge zu einem Ausgang zusammen (ggf. mit Arbitration). Der Splitter arbeitet in die entgegengesetzte Richtung.
- **AUI_PASS**: Ein reiner 1:1-Durchgangsadapter ohne Verzweigung.
- `AUI_SPLIT_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

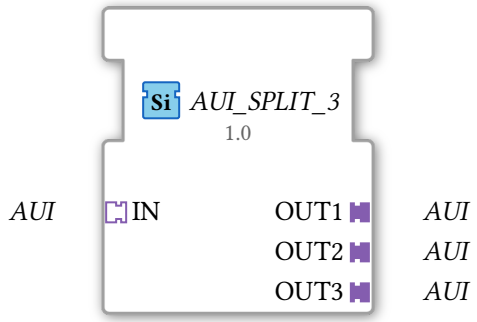
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AUI_SPLIT_2_UNGATED** ist ein minimalistischer, aber nützlicher Funktionsblock zur Aufteilung eines unidirektionalen AUI-Signals auf zwei Ausgänge. Dank seiner generischen Implementierung und dem Verzicht auf komplexe Logik eignet er sich ideal für Szenarien, in denen eine saubere, verlustfreie Vervielfältigung von Datenströmen erforderlich ist.

AUI_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AUI_SPLIT_3** dient als generischer Splitter, der eine einzelne eingehende AUI-Schnittstelle auf drei identische ausgehende AUI-Schnittstellen verteilt. Er ermöglicht die einfache Vervielfachung eines unidirektionalen Adapters (vom Typ `adapter::types::unidirectional::AUI`), ohne dass die darin transportierten Ereignisse oder Daten verändert werden.

Schnittstellenstruktur

Ereignis-Eingänge

Keine – der FB besitzt keine eigenständigen Ereignis-Eingänge. Die Ereignisweiterleitung erfolgt ausschließlich über die AUI-Adapter.

Ereignis-Ausgänge

Keine – der FB besitzt keine eigenständigen Ereignis-Ausgänge. Die Ereignisweiterleitung erfolgt ausschließlich über die AUI-Adapter.

Daten-Eingänge

Keine – der FB besitzt keine eigenständigen Daten-Eingänge. Daten werden über die AUI-Adapter übertragen.

Daten-Ausgänge

Keine – der FB besitzt keine eigenständigen Daten-Ausgänge. Daten werden über die AUI-Adapter übertragen.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	`IN`	`adapter::types::unidirectional::AUI`	Eingehende AUI-Schnittstelle, die auf drei Ausgänge verteilt wird.
Plug (Ausgang)	`OUT1`	`adapter::types::unidirectional::AUI`	Erster ausgehender AUI-Ausgang (identisch zum Eingangssignal).
Plug (Ausgang)	`OUT2`	`adapter::types::unidirectional::AUI`	Zweiter ausgehender AUI-Ausgang (identisch zum Eingangssignal).
Plug (Ausgang)	`OUT3`	`adapter::types::unidirectional::AUI`	Dritter ausgehender AUI-Ausgang (identisch zum Eingangssignal).

Funktionsweise

Der Baustein arbeitet als reiner Signalverteiler. Er nimmt über den Socket `IN` eine AUI-Schnittstelle entgegen und leitet alle über diese Schnittstelle eingehenden Ereignisse und Daten unverändert an die drei Plugs `OUT1`, `OUT2` und `OUT3` weiter. Es findet keinerlei Verarbeitung, Verzögerung oder Zustandsänderung statt. Der FB ist somit **stateless** und verhält sich wie eine einfache Verdrahtungsvervielfachung.

Technische Besonderheiten

- **Generischer Aufbau**: Der Baustein ist als generischer FB (`GEN_AUI_SPLIT`) gekennzeichnet, sodass er für verschiedene AUI-Adater-Varianten (mit unterschiedlichen Ereignis-/Datensignaturen) eingesetzt werden kann.
- **Unidirektional**: Der Adaptertyp `unidirectional` bedeutet, dass die Daten- und Ereignisflüsse nur in eine Richtung (vom Socket zu den Plugs) erfolgen. Rückwärtige Kommunikation ist nicht vorgesehen.
- **Keine Latenz**: Durch das Fehlen interner Logik tritt keine messbare Verzögerung auf.
- **Keine Konfiguration**: Der FB benötigt keine Parameter – die Anzahl der Ausgänge ist fest auf drei festgelegt.

Zustandsübersicht

Der Baustein besitzt keinen internen Zustandsautomaten. Es gibt nur einen einzigen, durchgängigen Betriebszustand, in dem die Eingangsschnittstelle permanent auf die drei Ausgangsschnittstellen gespiegelt wird.

Anwendungsszenarien

- **Signalvervielfachung**: Ein AUI-Signal eines Sensors oder einer Steuerung soll an mehrere unabhängige Empfänger (z. B. Aktoren, Anzeigen, andere Steuerungen) weitergeleitet werden.
- **Monitoring**: Ein bestehender AUI-Datenstrom wird an eine Überwachungseinheit kopiert, ohne die Originalsignalpfade zu beeinträchtigen.
- **Test- und Simulationsumgebungen**: Ein Testreiz soll parallel an mehrere zu testende Komponenten verteilt werden.

Vergleich mit ähnlichen Bausteinen

Baustein	Ausgänge	Eigenschaften
`AUI_SPLIT_2`	2	Gleiche Funktionsweise, aber nur zwei Ausgänge.
`AUI_SPLIT_4`	4	Erweiterte Version mit vier Ausgängen.

AUI_MERGE	–	Führt mehrere AUI-Eingänge zu einem Ausgang zusammen (entgegengesetzte Funktion).

Der AUI_SPLIT_3 stellt eine spezifische Konfiguration dar, die genau drei identische Ausgänge bereitstellt. Er kann bei Bedarf durch andere Splitter-Varianten ersetzt werden.

- AUI_SPLIT_3_UNGATED: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

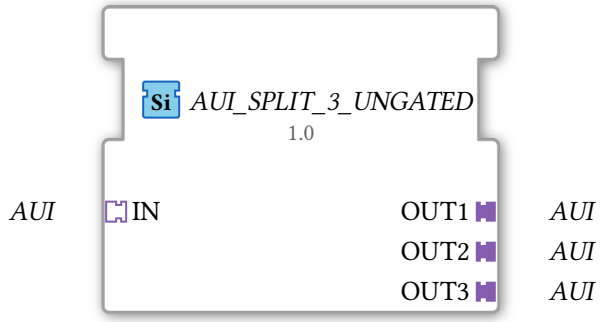
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der AUI_SPLIT_3 ist ein einfacher, aber nützlicher generischer Funktionsbaustein zur Vervielfachung einer unidirektionalen AUI-Schnittstelle. Dank seiner reinen Durchleitungsfunktion und fehlenden Zustandslogik ist er zuverlässig, performant und leicht in bestehende Automatisierungslösungen integrierbar. Er eignet sich besonders für Anwendungen, in denen ein Signal auf mehrere Pfade aufgeteilt werden muss.

AUI_SPLIT_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AUI_SPLIT_3. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AUI_SPLIT_3_UNGATED** dient als generischer Splitter, der eine einzelne eingehende AUI-Schnittstelle auf drei identische ausgehende AUI-Schnittstellen verteilt. Er ermöglicht die einfache Vervielfachung eines unidirektionalen Adapters (vom Typ `adapter::types::unidirectional::AUI`), ohne dass die darin transportierten Ereignisse oder Daten verändert werden.

Schnittstellenstruktur

Ereignis-Eingänge

Keine – der FB besitzt keine eigenständigen Ereignis-Eingänge. Die Ereignisweiterleitung erfolgt ausschließlich über die AUI-Adapter.

Ereignis-Ausgänge

Keine – der FB besitzt keine eigenständigen Ereignis-Ausgänge. Die Ereignisweiterleitung erfolgt ausschließlich über die AUI-Adapter.

Daten-Eingänge

Keine – der FB besitzt keine eigenständigen Daten-Eingänge. Daten werden über die AUI-Adapter übertragen.

Daten-Ausgänge

Keine – der FB besitzt keine eigenständigen Daten-Ausgänge. Daten werden über die AUI-Adapter übertragen.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	`IN`	`adapter::types::unidirectional::AUI`	Eingehende AUI-Schnittstelle, die auf drei Ausgänge verteilt wird.
Plug (Ausgang)	`OUT1`	`adapter::types::unidirectional::AUI`	Erster ausgehender AUI-Ausgang (identisch zum Eingangssignal).
Plug (Ausgang)	`OUT2`	`adapter::types::unidirectional::AUI`	Zweiter ausgehender AUI-Ausgang (identisch zum Eingangssignal).
Plug (Ausgang)	`OUT3`	`adapter::types::unidirectional::AUI`	Dritter ausgehender AUI-Ausgang (identisch zum Eingangssignal).

Funktionsweise

Der Baustein arbeitet als reiner Signalverteiler. Er nimmt über den Socket `IN` eine AUI-Schnittstelle entgegen und leitet alle über diese Schnittstelle eingehenden Ereignisse und Daten unverändert an die drei Plugs `OUT1`, `OUT2` und `OUT3` weiter. Es findet keinerlei Verarbeitung, Verzögerung oder Zustandsänderung statt. Der FB ist somit **stateless** und verhält sich wie eine einfache Verdrahtungsvervielfachung.

Technische Besonderheiten

- **Generischer Aufbau**: Der Baustein ist als generischer FB (`GEN_AUI_SPLIT`) gekennzeichnet, sodass er für verschiedene AUI-Adater-Varianten (mit unterschiedlichen Ereignis-/Datensignaturen) eingesetzt werden kann.
- **Unidirektional**: Der Adaptertyp `unidirectional` bedeutet, dass die Daten- und Ereignisflüsse nur in eine Richtung (vom Socket zu den Plugs) erfolgen. Rückwärtige Kommunikation ist nicht vorgesehen.
- **Keine Latenz**: Durch das Fehlen interner Logik tritt keine messbare Verzögerung auf.
- **Keine Konfiguration**: Der FB benötigt keine Parameter – die Anzahl der Ausgänge ist fest auf drei festgelegt.

Zustandsübersicht

Der Baustein besitzt keinen internen Zustandsautomaten. Es gibt nur einen einzigen, durchgängigen Betriebszustand, in dem die Eingangsschnittstelle permanent auf die drei Ausgangsschnittstellen gespiegelt wird.

Anwendungsszenarien

- **Signalvervielfachung**: Ein AUI-Signal eines Sensors oder einer Steuerung soll an mehrere unabhängige Empfänger (z. B. Aktoren, Anzeigen, andere Steuerungen) weitergeleitet werden.
- **Monitoring**: Ein bestehender AUI-Datenstrom wird an eine Überwachungseinheit kopiert, ohne die Originalsignalpfade zu beeinträchtigen.
- **Test- und Simulationsumgebungen**: Ein Testreiz soll parallel an mehrere zu testende Komponenten verteilt werden.

Vergleich mit ähnlichen Bausteinen

Baustein	Ausgänge	Eigenschaften
`AUI_SPLIT_2`	2	Gleiche Funktionsweise, aber nur zwei Ausgänge.
`AUI_SPLIT_4`	4	Erweiterte Version mit vier Ausgängen.
`AUI_MERGE`	–	Führt mehrere AUI-Eingänge zu einem Ausgang zusammen (entgegengesetzte Funktion).

Der `AUI_SPLIT_3_UNGATED` stellt eine spezifische Konfiguration dar, die genau drei identische Ausgänge bereitstellt. Er kann bei Bedarf durch andere Splitter-Varianten ersetzt werden.

- **`AUI_SPLIT_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

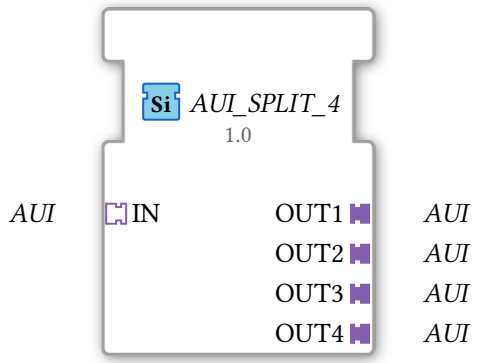
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AUI_SPLIT_3_UNGATED` ist ein einfacher, aber nützlicher generischer Funktionsbaustein zur Vervielfachung einer unidirektionalen AUI-Schnittstelle. Dank seiner reinen Durchleitungsfunktion und fehlenden Zustandslogik ist er zuverlässig, performant und leicht in bestehende Automatisierungslösungen integrierbar. Er eignet sich besonders für Anwendungen, in denen ein Signal auf mehrere Pfade aufgeteilt werden muss.

AUI_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUI_SPLIT_4** dient als generischer Verteiler für AUI-Signale. Er nimmt einen einzelnen AUI-Eingang (Socket) entgegen und leitet ihn auf vier separate AUI-Ausgänge (Plugs) weiter. Dieser Baustein wird typischerweise in der Automatisierungstechnik eingesetzt, wenn ein unidirektionales Signal mehrfach parallel benötigt wird.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	`IN`	`adapter::types::unidirectional::AUI`	Eingangssignal (AUI)
Plug	`OUT1`	`adapter::types::unidirectional::AUI`	Erster Ausgang (identisch zu IN)
Plug	`OUT2`	`adapter::types::unidirectional::AUI`	Zweiter Ausgang (identisch zu IN)
Plug	`OUT3`	`adapter::types::unidirectional::AUI`	Dritter Ausgang (identisch zu IN)
Plug	`OUT4`	`adapter::types::unidirectional::AUI`	Vierter Ausgang (identisch zu IN)

Funktionsweise

Der Baustein führt keine Datenmanipulation durch. Er stellt eine passive Verteilereinheit dar: Sobald über den Adapter `IN` ein Signal anliegt, wird dieses ohne Verzögerung oder Änderung an alle vier Ausgangsadapter `OUT1`–`OUT4` weitergegeben. Die Signalrichtung ist unidirektional vom Socket zu den Plugs.

Technische Besonderheiten

- **Generischer Typ:** Der FB ist als generischer Baustein deklariert (`GenericClassName = 'GEN_AUI_SPLIT'`). Er kann somit mit verschiedenen Ausprägungen des AUI-Adaptertyps verwendet werden.
- **Keine Zustände:** Da keine Ereignisse oder Algorithmen vorhanden sind, besitzt der Baustein keinen internen Zustandsautomaten.
- **Lizenz:** Der Quellcode ist unter der **Eclipse Public License 2.0** verfügbar.

Zustandsübersicht

Der Baustein hat keine Ereignis-Eingänge oder -Ausgänge und führt keinen Algorithmus aus. Es existiert kein expliziter Zustandsautomat. Der Baustein verhält sich wie eine reine Signalverteilung (Drahtverbindung).

Anwendungsszenarien

- **Signalkaskadierung:** Ein Sensorsignal (z. B. AUI-basiertes Taktsignal) muss an mehrere Aktoren verteilt werden.
- **Testumgebungen:** Ein Testsignal soll parallel zu mehreren Prüflingen geführt werden.
- **Redundanz:** Ein Signal wird auf mehrere parallele Pfade aufgeteilt, um Ausfallsicherheit zu gewährleisten.
- **Bus-Erweiterung:** Als passiver Splitter in einem AUI-Kommunikationssystem.

Vergleich mit ähnlichen Bausteinen

- **AUI_SPLIT_2** – Teilt einen Eingang auf zwei Ausgänge (analoges Prinzip, geringere Anzahl).
- **AUI_MERGE** – Fasst mehrere AUI-Eingänge zu einem Ausgang zusammen (Gegenstück).
- **AUI_SELECT** – Wählt mit einem Steuersignal einen von mehreren Eingängen aus (mit Auswahlfunktion).
- Im Gegensatz zu diesen Bausteinen bietet `AUI_SPLIT_4` eine reine 1:4-Verteilung ohne Selektionslogik oder Zusammenführung.
- `AUI_SPLIT_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

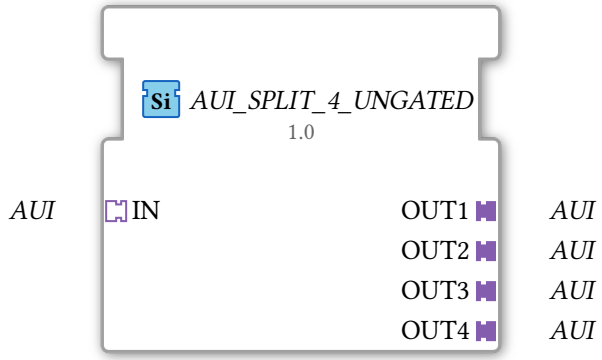
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der Funktionsblock **AUI_SPLIT_4** ist ein einfacher, aber nützlicher Baustein zur passiven Vervielfältigung von AUI-Signalen. Seine generische Natur und die klare Schnittstelle machen ihn zu einem flexiblen Werkzeug in der Automatisierungsentwicklung, insbesondere wenn mehrere gleichartige Empfänger mit demselben Signal versorgt werden müssen.

AUI_SPLIT_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUI_SPLIT_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUI_SPLIT_4_UNGATED** dient als generischer Verteiler für AUI-Signale. Er nimmt einen einzelnen AUI-Eingang (Socket) entgegen und leitet ihn auf vier separate AUI-Ausgänge (Plugs) weiter. Dieser Baustein wird typischerweise in der Automatisierungstechnik eingesetzt, wenn ein unidirektionales Signal mehrfach parallel benötigt wird.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	`IN`	`adapter::types::unidirectional::AUI`	Eingangssignal (AUI)
Plug	`OUT1`	`adapter::types::unidirectional::AUI`	Erster Ausgang (identisch zu IN)
Plug	`OUT2`	`adapter::types::unidirectional::AUI`	Zweiter Ausgang (identisch zu IN)

Plug	OUT3	adapter::types::unidirectional::AUI	Dritter Ausgang (identisch zu IN)
Plug	OUT4	adapter::types::unidirectional::AUI	Vierter Ausgang (identisch zu IN)

Funktionsweise

Der Baustein führt keine Datenmanipulation durch. Er stellt eine passive Verteilereinheit dar: Sobald über den Adapter IN ein Signal anliegt, wird dieses ohne Verzögerung oder Änderung an alle vier Ausgangsadapter OUT1–OUT4 weitergegeben. Die Signalrichtung ist unidirektional vom Socket zu den Plugs.

Technische Besonderheiten

- **Generischer Typ:** Der FB ist als generischer Baustein deklariert (GenericClassName = 'GEN_AUI_SPLIT'). Er kann somit mit verschiedenen Ausprägungen des AUI-Adaptertyps verwendet werden.
- **Keine Zustände:** Da keine Ereignisse oder Algorithmen vorhanden sind, besitzt der Baustein keinen internen Zustandsautomaten.
- **Lizenz:** Der Quellcode ist unter der **Eclipse Public License 2.0** verfügbar.

Zustandsübersicht

Der Baustein hat keine Ereignis-Eingänge oder -Ausgänge und führt keinen Algorithmus aus. Es existiert kein expliziter Zustandsautomat. Der Baustein verhält sich wie eine reine Signalverteilung (Drahtverbindung).

Anwendungsszenarien

- **Signalkaskadierung:** Ein Sensorsignal (z. B. AUI-basiertes Taktsignal) muss an mehrere Aktoren verteilt werden.
- **Testumgebungen:** Ein Testsignal soll parallel zu mehreren Prüflingen geführt werden.
- **Redundanz:** Ein Signal wird auf mehrere parallele Pfade aufgeteilt, um Ausfallsicherheit zu gewährleisten.
- **Bus-Erweiterung:** Als passiver Splitter in einem AUI-Kommunikationssystem.

Vergleich mit ähnlichen Bausteinen

- **AUI_SPLIT_2** – Teilt einen Eingang auf zwei Ausgänge (analoges Prinzip, geringere Anzahl).
- **AUI_MERGE** – Fasst mehrere AUI-Eingänge zu einem Ausgang zusammen (Gegenstück).
- **AUI_SELECT** – Wählt mit einem Steuersignal einen von mehreren Eingängen aus (mit Auswahlfunktion).
- Im Gegensatz zu diesen Bausteinen bietet AUI_SPLIT_4_UNGATED eine reine 1:4-Verteilung ohne Selektionslogik oder Zusammenführung.
- AUI_SPLIT_4: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

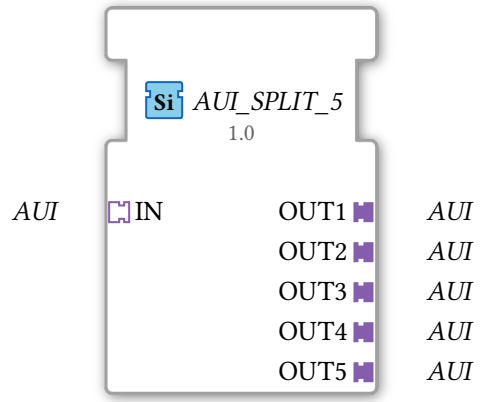
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der Funktionsblock **AUI_SPLIT_4_UNGATED** ist ein einfacher, aber nützlicher Baustein zur passiven Vervielfältigung von AUI-Signalen. Seine generische Natur und die klare Schnittstelle machen ihn zu einem flexiblen Werkzeug in der Automatisierungsentwicklung, insbesondere wenn mehrere gleichartige Empfänger mit demselben Signal versorgt werden müssen.

AUI_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUI_SPLIT_5** ist ein generischer Baustein zur Aufteilung eines eingehenden AUI-Adapters („Automation Unit Interface“) auf fünf separate, gleichartige AUI-Ausgänge. Er realisiert eine 1:5-Verteilung in unidirektionaler Datenrichtung. Entwickelt für den Einsatz in der Automatisierungstechnik, ermöglicht er die redundante oder verteilte Weiterleitung von AUI-Signalen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden. Die Signale werden ausschließlich über die Adapter-Schnittstellen transportiert.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Der Baustein verfügt über sechs Adapter-Schnittstellen vom Typ `adapter::types::unidirectional::AUI`:

- **IN** (Socket) – Eingangsadapter für das zu verteilende AUI-Signal.
- **OUT1** bis **OUT5** (Plugs) – Fünf Ausgangsadapter, die das eingehende Signal spiegeln.

Sämtliche Adapter sind unidirektional ausgelegt und übertragen Daten sowie optional zugehörige Ereignisse gemäß der AUI-Spezifikation.

Funktionsweise

Der FB fungiert als reiner Splitter ohne Logik oder Pufferung. Ein am Socket **IN** anliegendes AUI-Signal wird parallel und unverändert auf alle fünf Plugs **OUT1** bis **OUT5** kopiert. Die Weiterleitung erfolgt instantan, d.h. ohne Verzögerung durch interne Verarbeitung. Die einzelnen Ausgänge sind galvanisch oder logisch nicht entkoppelt – eine Änderung am Eingang wirkt sich sofort auf alle Ausgänge aus.

Technische Besonderheiten

- **Generischer Typ**: Der FB ist als generischer Baustein deklariert (Attribut `eclipse4diac::core::GenericClassName = 'GEN_AUI_SPLIT'`). Dies ermöglicht eine spätere Typanpassung oder Instanziierung mit einem konkreten AUI-Typ.
- **Keine Zustandshaltung**: Da keine Ereignisse oder Daten existieren, die auf Änderungen reagieren, ist der FB zustandslos. Eine Zustandsmaschine ist nicht erforderlich.

- **Volle Bandbreite**: Die Ausgänge liefern das Eingangssignal ohne Einschränkung – alle Daten- und Ereigniskanäle des AUI-Adapters werden 1:1 kopiert.

Zustandsübersicht

Der Baustein besitzt keine eigenen Zustände oder Zustandsautomaten. Er verhält sich wie eine passive Verdrahtung. Eine Zustandsübersicht ist daher nicht anwendbar.

Anwendungsszenarien

- **Signalverteilung**: Aufteilung eines zentralen AUI-Datenstroms auf mehrere parallele Verbraucher (z.B. Anzeigen, Überwachungseinheiten, Steuerungen).
- **Test- und Diagnoseaufbauten**: Verwendung in Prüfständen, um ein Signal simultan an mehrere Messgeräte oder Logging-Systeme zu legen.
- **Redundanz**: Bereitstellung identischer Signale für Failover-Systeme, wenn die AUI-Kommunikation unidirektional ist.
- **Bus-Kopplung**: Einsatz als passiver Splitter in AUI-basierten Linien- oder Sternstrukturen.

Vergleich mit ähnlichen Bausteinen

- **AUI_SPLIT_2 / AUI_SPLIT_3 / AUI_SPLIT_4**: Analoge Bausteine mit weniger Ausgängen; Skalierung nur durch Auswahl des passenden Split-Typs.
- **AUI_MERGE (z.B. AUI_MERGE_5)**: Führt mehrere AUI-Eingänge zu einem Ausgang zusammen – gegensätzliche Funktion.
- **Reine Event-Splitter (E_SPLIT)**: Teilen nur Ereignisse, aber keine Datenadapter. Der AUI_SPLIT_5 hingegen verteilt komplette Adapter-Schnittstellen.
- `AUI_SPLIT_5_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

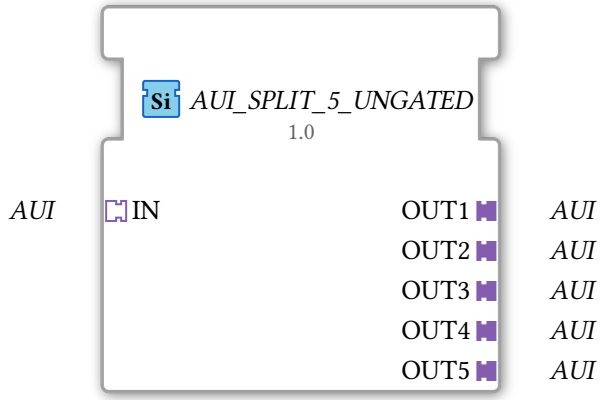
Der **AUI_SPLIT_5** ist ein einfacher, aber essenzieller Baustein zur Vervielfältigung von AUI-Signalen. Seine generische Auslegung und die vollständige Durchleitung des Adapters machen ihn flexibel einsetzbar. Für Anwendungen, die eine unidirektionale 1:5-Verteilung bei gleichzeitiger Minimierung von Latenz erfordern, stellt er eine optimale Lösung dar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUI_SPLIT_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUI_SPLIT_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUI_SPLIT_5_UNGATED** ist ein generischer Baustein zur Aufteilung eines eingehenden AUI-Adapters („Automation Unit Interface“) auf fünf separate, gleichartige AUI-Ausgänge. Er realisiert eine 1:5-Verteilung in unidirektionaler Datenrichtung. Entwickelt für den Einsatz in der Automatisierungstechnik, ermöglicht er die redundante oder verteilte Weiterleitung von AUI-Signalen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden. Die Signale werden ausschließlich über die Adapter-Schnittstellen transportiert.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Der Baustein verfügt über sechs Adapter-Schnittstellen vom Typ `adapter::types::unidirectional::AUI`:

- **IN** (Socket) – Eingangsadapter für das zu verteilende AUI-Signal.
- **OUT1** bis **OUT5** (Plugs) – Fünf Ausgangsadapter, die das eingehende Signal spiegeln.

Sämtliche Adapter sind unidirektional ausgelegt und übertragen Daten sowie optional zugehörige Ereignisse gemäß der AUI-Spezifikation.

Funktionsweise

Der FB fungiert als reiner Splitter ohne Logik oder Pufferung. Ein am Socket **IN** anliegendes AUI-Signal wird parallel und unverändert auf alle fünf Plugs **OUT1** bis **OUT5** kopiert. Die Weiterleitung erfolgt instantan, d.h. ohne Verzögerung durch interne Verarbeitung. Die einzelnen Ausgänge sind galvanisch oder logisch nicht entkoppelt – eine Änderung am Eingang wirkt sich sofort auf alle Ausgänge aus.

Technische Besonderheiten

- **Generischer Typ**: Der FB ist als generischer Baustein deklariert (Attribut `eclipse4diac::core::GenericClassName = 'GEN_AUI_SPLIT'`). Dies ermöglicht eine spätere Typanpassung oder Instanziierung mit einem konkreten AUI-Typ.
- **Keine Zustandshaltung**: Da keine Ereignisse oder Daten existieren, die auf Änderungen reagieren, ist der FB zustandslos. Eine Zustandsmaschine ist nicht erforderlich.

- **Volle Bandbreite**: Die Ausgänge liefern das Eingangssignal ohne Einschränkung – alle Daten- und Ereigniskanäle des AUI-Adapters werden 1:1 kopiert.

Zustandsübersicht

Der Baustein besitzt keine eigenen Zustände oder Zustandsautomaten. Er verhält sich wie eine passive Verdrahtung. Eine Zustandsübersicht ist daher nicht anwendbar.

Anwendungsszenarien

- **Signalverteilung**: Aufteilung eines zentralen AUI-Datenstroms auf mehrere parallele Verbraucher (z.B. Anzeigen, Überwachungseinheiten, Steuerungen).
- **Test- und Diagnoseaufbauten**: Verwendung in Prüfständen, um ein Signal simultan an mehrere Messgeräte oder Logging-Systeme zu legen.
- **Redundanz**: Bereitstellung identischer Signale für Failover-Systeme, wenn die AUI-Kommunikation unidirektional ist.
- **Bus-Kopplung**: Einsatz als passiver Splitter in AUI-basierten Linien- oder Sternstrukturen.

Vergleich mit ähnlichen Bausteinen

- **AUI_SPLIT_2 / AUI_SPLIT_3 / AUI_SPLIT_4**: Analoge Bausteine mit weniger Ausgängen; Skalierung nur durch Auswahl des passenden Split-Typs.
- **AUI_MERGE (z.B. AUI_MERGE_5)**: Führt mehrere AUI-Eingänge zu einem Ausgang zusammen – gegensätzliche Funktion.
- **Reine Event-Splitter (E_SPLIT)**: Teilen nur Ereignisse, aber keine Datenadapter. Der AUI_SPLIT_5_UNGATED hingegen verteilt komplette Adapter-Schnittstellen.
- `AUI_SPLIT_5_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- `AUI_SPLIT_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

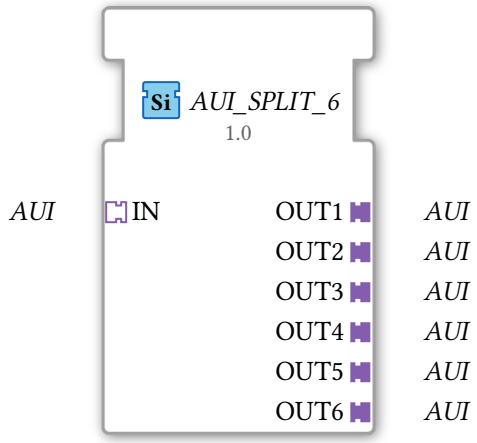
Fazit

Der **AUI_SPLIT_5_UNGATED** ist ein einfacher, aber essenzieller Baustein zur Vervielfältigung von AUI-Signalen. Seine generische Auslegung und die vollständige Durchleitung des Adapters machen ihn flexibel einsetzbar. Für Anwendungen, die eine unidirektionale 1:5-Verteilung bei gleichzeitiger Minimierung von Latenz erfordern, stellt er eine optimale Lösung dar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUI_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUI_SPLIT_6** dient dazu, einen eingehenden **AUI-Adapter** (Application Universal Interface) auf sechs ausgehende AUI-Adapter aufzuteilen. Er ist als generischer Baustein konzipiert, der in der Lage ist, das eingehende AUI-Signal ohne Datenverarbeitung oder Logik auf alle sechs Ausgänge zu verteilen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Der Baustein verwendet ausschließlich Adapter für die Kommunikation:

Richtung	Name	Typ	Beschreibung
Eingang	**IN**	`adapter::types::unidirectional::AUI`	Eingehender AUI-Adapter, der aufgeteilt wird.
Ausgang	**OUT1**	`adapter::types::unidirectional::AUI`	Erster ausgehender AUI-Adapter.
Ausgang	**OUT2**	`adapter::types::unidirectional::AUI`	Zweiter ausgehender AUI-Adapter.
Ausgang	**OUT3**	`adapter::types::unidirectional::AUI`	Dritter ausgehender AUI-Adapter.
Ausgang	**OUT4**	`adapter::types::unidirectional::AUI`	Vierter ausgehender AUI-Adapter.
Ausgang	**OUT5**	`adapter::types::unidirectional::AUI`	Fünfter ausgehender AUI-Adapter.
Ausgang	**OUT6**	`adapter::types::unidirectional::AUI`	Sechster ausgehender AUI-Adapter.

Funktionsweise

Der Baustein leitet das am Adapter-Eingang **IN** ankommende AUI-Signal ohne jede Verzögerung oder Verarbeitung an alle sechs Adapter-Ausgänge (**OUT1** bis **OUT6**) weiter. Es findet keine Datenmanipulation statt – der Baustein wirkt als reiner Verteiler (Splitter) für AUI-Signale.

Technische Besonderheiten

- Der Baustein ist als **generischer Funktionsblock** deklariert: Das Attribut `eclipse4diac::core::GenericClassName` ist auf `'GEN_AUI_SPLIT'` gesetzt.
- Es erfolgt keine Zustandsverwaltung oder Ereignislogik – der Baustein ist rein kombinatorisch.
- Die Ausgänge sind voneinander unabhängig; ein Fehler an einem Ausgang beeinflusst die anderen nicht.
- Durch die Verwendung von AUI-Adaptern ist der Baustein für eine unidirektionale Signalweitergabe optimiert.

Zustandsübersicht

Der Baustein besitzt **keine Zustände**. Er arbeitet kontinuierlich und ohne interne Zustandsmaschine. Das Eingangssignal wird sofort und ständig auf alle Ausgänge gespiegelt.

Anwendungsszenarien

- **Signalverteilung in der Steuerungstechnik**: Wenn ein AUI-Signal (z.B. ein Messwert oder Steuerbefehl) an mehrere Verbraucher weitergegeben werden muss.
- **Redundanz oder Parallelverarbeitung**: Verteilung eines Signals an mehrere identische nachgelagerte Funktionseinheiten.
- **Test- und Simulationsaufbauten**: Aufteilung eines AUI-Signals für gleichzeitige Überwachung und Verarbeitung.

Vergleich mit ähnlichen Bausteinen

- **AUI_SPLIT_2 / AUI_SPLIT_4**: Diese Bausteine teilen ein Signal auf zwei bzw. vier Ausgänge auf. **AUI_SPLIT_6** bietet die größte Anzahl (6) an Ausgängen und deckt damit Anwendungen mit vielen parallelen Zielen ab.
- **AUI_MERGE_***: Im Gegensatz zu Splittern fassen Merge-Bausteine mehrere AUI-Signale zu einem zusammen.
- `AUI_SPLIT_6_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

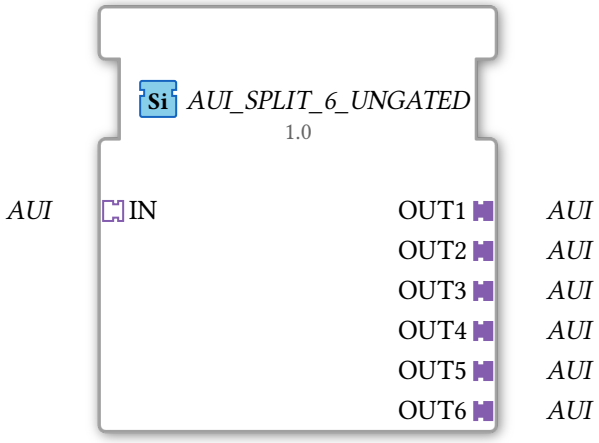
Der **AUI_SPLIT_6** ist ein einfacher, aber effektiver Verteilerbaustein für AUI-Signale. Er ermöglicht die sternförmige Weiterleitung eines Eingangssignals an bis zu sechs unabhängige Ausgänge. Aufgrund seiner generischen Natur und fehlenden Logik eignet er sich hervorragend für flexible und erweiterbare Automatisierungslösungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUI_SPLIT_6_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUI_SPLIT_6`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUI_SPLIT_6_UNGATED** dient dazu, einen eingehenden **AUI-Adapter** (Application Universal Interface) auf sechs ausgehende AUI-Adapter aufzuteilen. Er ist als generischer Baustein konzipiert, der in der Lage ist, das eingehende AUI-Signal ohne Datenverarbeitung oder Logik auf alle sechs Ausgänge zu verteilen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Der Baustein verwendet ausschließlich Adapter für die Kommunikation:

Richtung	Name	Typ	Beschreibung
Eingang	**IN**	`adapter::types::unidirectional::AUI`	Eingehender AUI-Adapter, der aufgeteilt wird.
Ausgang	**OUT1**	`adapter::types::unidirectional::AUI`	Erster ausgehender AUI-Adapter.
Ausgang	**OUT2**	`adapter::types::unidirectional::AUI`	Zweiter ausgehender AUI-Adapter.
Ausgang	**OUT3**	`adapter::types::unidirectional::AUI`	Dritter ausgehender AUI-Adapter.
Ausgang	**OUT4**	`adapter::types::unidirectional::AUI`	Vierter ausgehender AUI-Adapter.
Ausgang	**OUT5**	`adapter::types::unidirectional::AUI`	Fünfter ausgehender AUI-Adapter.
Ausgang	**OUT6**	`adapter::types::unidirectional::AUI`	Sechster ausgehender AUI-Adapter.

Funktionsweise

Der Baustein leitet das am Adapter-Eingang **IN** ankommende AUI-Signal ohne jede Verzögerung oder Verarbeitung an alle sechs Adapter-Ausgänge (**OUT1** bis **OUT6**) weiter. Es findet keine Datenmanipulation statt – der Baustein wirkt als reiner Verteiler (Splitter) für AUI-Signale.

Technische Besonderheiten

- Der Baustein ist als **generischer Funktionsblock** deklariert: Das Attribut `eclipse4diac::core::GenericClassName` ist auf `'GEN_AUI_SPLIT'` gesetzt.
- Es erfolgt keine Zustandsverwaltung oder Ereignislogik – der Baustein ist rein kombinatorisch.
- Die Ausgänge sind voneinander unabhängig; ein Fehler an einem Ausgang beeinflusst die anderen nicht.
- Durch die Verwendung von AUI-Adaptern ist der Baustein für eine unidirektionale Signalweitergabe optimiert.

Zustandsübersicht

Der Baustein besitzt **keine Zustände**. Er arbeitet kontinuierlich und ohne interne Zustandsmaschine. Das Eingangssignal wird sofort und ständig auf alle Ausgänge gespiegelt.

Anwendungsszenarien

- **Signalverteilung in der Steuerungstechnik**: Wenn ein AUI-Signal (z.B. ein Messwert oder Steuerbefehl) an mehrere Verbraucher weitergegeben werden muss.
- **Redundanz oder Parallelverarbeitung**: Verteilung eines Signals an mehrere identische nachgelagerte Funktionseinheiten.
- **Test- und Simulationsaufbauten**: Aufteilung eines AUI-Signals für gleichzeitige Überwachung und Verarbeitung.

Vergleich mit ähnlichen Bausteinen

- **AUI_SPLIT_2 / AUI_SPLIT_4**: Diese Bausteine teilen ein Signal auf zwei bzw. vier Ausgänge auf. **AUI_SPLIT_6_UNGATED** bietet die größte Anzahl (6) an Ausgängen und deckt damit Anwendungen mit vielen parallelen Zielen ab.
- **AUI_MERGE_***: Im Gegensatz zu Splittern fassen Merge-Bausteine mehrere AUI-Signale zu einem zusammen.
- `AUI_SPLIT_6`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

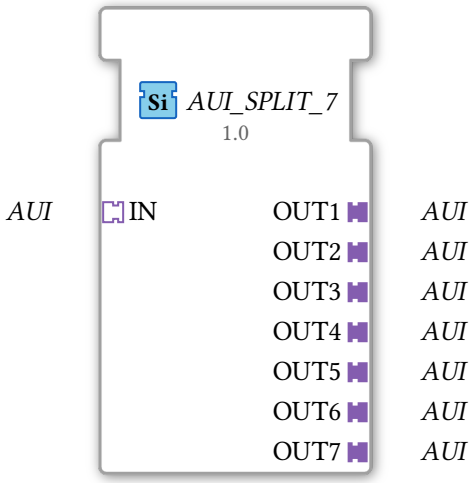
Fazit

Der **AUI_SPLIT_6_UNGATED** ist ein einfacher, aber effektiver Verteilerbaustein für AUI-Signale. Er ermöglicht die sternförmige Weiterleitung eines Eingangssignals an bis zu sechs unabhängige Ausgänge. Aufgrund seiner generischen Natur und fehlenden Logik eignet er sich hervorragend für flexible und erweiterbare Automatisierungslösungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUI_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AUI_SPLIT_7` dient dazu, ein eingehendes AUI-Signal (Application User Interface) auf bis zu sieben identische Ausgänge zu verteilen. Er ist als generischer Baustein konzipiert und arbeitet ausschließlich über Adapter-Schnittstellen – ohne Ereignis- oder Daten-Ein-/Ausgänge im klassischen Sinne. Dadurch eignet er sich besonders für reine Signalverteilungen in adapterbasierten Architekturen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

Typ	Richtung	Name
Socket (adapter::types::unidirectional::AUI)	Eingang	IN
Plug (adapter::types::unidirectional::AUI)	Ausgang	OUT1
Plug (adapter::types::unidirectional::AUI)	Ausgang	OUT2
Plug (adapter::types::unidirectional::AUI)	Ausgang	OUT3
Plug (adapter::types::unidirectional::AUI)	Ausgang	OUT4
Plug (adapter::types::unidirectional::AUI)	Ausgang	OUT5
Plug (adapter::types::unidirectional::AUI)	Ausgang	OUT6
Plug (adapter::types::unidirectional::AUI)	Ausgang	OUT7

Funktionsweise

Der Baustein stellt eine 1-zu-7-Verteilerlogik für AUI-Adapter dar. Das über den Socket `IN` eingehende Signal wird ohne weitere Verarbeitung oder logische Bedingungen auf alle sieben Plugs (`OUT1` ... `OUT7`) repliziert. Die Weiterleitung geschieht rein passiv: jedes empfangene Signal wird an jedem Ausgang gleichzeitig bereitgestellt.

Technische Besonderheiten

- **Reiner Adapter-Baustein**: Der FB enthält weder Ereignis- noch Daten-Ein-/Ausgänge – die gesamte Kommunikation erfolgt über die AUI-Adapter.
- **Unidirektional**: Die Adapter sind vom Typ `unidirectional`, d. h. die Daten fließen nur vom Eingang (Socket) zu den Ausgängen (Plugs).
- **Generischer Typ**: Der FB ist als generischer Baustein (`GenericClassName = 'GEN_AUI_SPLIT'`) implementiert und kann für verschiedene AUI-Untertypen wiederverwendet werden.
- **Keine Zustandslogik**: Aufgrund der reinen Verteilungsfunktion besitzt der Baustein keinen internen Zustandsautomaten – er arbeitet kontinuierlich und ohne Verzögerung.

Zustandsübersicht

Es existiert keine Zustandsmaschine. Der Baustein ist rein datenflussgesteuert und besitzt nur einen statischen Betriebszustand.

Anwendungsszenarien

- **Signalverteilung in der Feldautomatisierung**: Ein von einer zentralen Steuerung kommendes AUI-Signal (z. B. Sensordaten oder Steuerbefehle) wird an mehrere parallele Aktoren oder Anzeigeeinheiten weitergeleitet.
- **Test- und Simulationsumgebungen**: Einsatz als Splitter, um ein Eingangssignal gleichzeitig auf mehrere Simulations- oder Diagnoseeinheiten zu geben.
- **Modulare Systeme**: In adapterbasierten Architekturen (z. B. nach IEC 61499) wird der Baustein genutzt, um eine einzige Schnittstelle auf mehrere nachgelagerte Funktionsblöcke aufzuteilen.

Vergleich mit ähnlichen Bausteinen

- **AUI_SPLIT_2, AUI_SPLIT_4** – Bausteine mit gleicher Funktionsweise, jedoch auf 2 bzw. 4 Ausgänge reduziert. Der `AUI_SPLIT_7` bietet die größte Anzahl an Ausgängen und deckt damit umfangreichere Verteilungsanforderungen ab.
- **AUI_MERGE** – das Gegenstück, das mehrere AUI-Eingänge zu einem Ausgang zusammenführt. Im Gegensatz dazu ist der `SPLIT` rein passiv und aggregiert keine Daten.
- `AUI_SPLIT_7_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

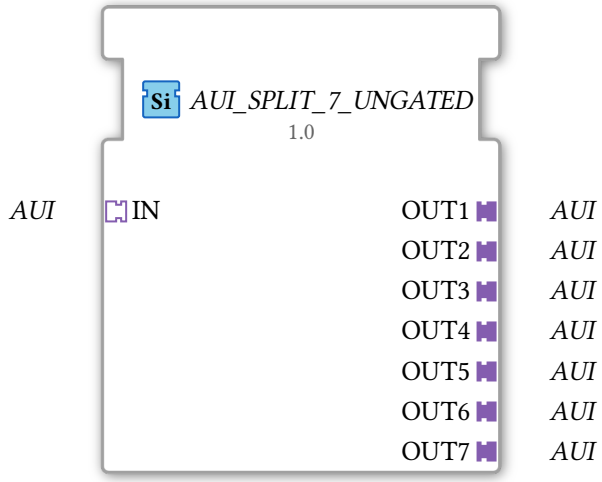
Fazit

Der `AUI_SPLIT_7` ist ein einfacher, aber essenzieller Funktionsblock zur unidirektionalen Signalverteilung in adapterbasierten Steuerungssystemen. Dank seines generischen Designs und der klaren 1-zu-N-Struktur lässt er sich flexibel in beliebige AUI-basierte Architekturen integrieren. Für Anwendungen, die eine breite Parallelverteilung ohne Nebenwirkungen erfordern, stellt er eine robuste und wartungsfreundliche Lösung dar.

AUI_SPLIT_7_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUI_SPLIT_7`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht

gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AUI_SPLIT_7_UNGATED` dient dazu, ein eingehendes AUI-Signal (Application User Interface) auf bis zu sieben identische Ausgänge zu verteilen. Er ist als generischer Baustein konzipiert und arbeitet ausschließlich über Adapter-Schnittstellen – ohne Ereignis- oder Daten-Ein-/Ausgänge im klassischen Sinne. Dadurch eignet er sich besonders für reine Signalverteilungen in adapterbasierten Architekturen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

Typ	Richtung	Name
Socket (adapter::types::unidirectional::AUI)	Eingang	IN
Plug (adapter::types::unidirectional::AUI)	Ausgang	OUT1
Plug (adapter::types::unidirectional::AUI)	Ausgang	OUT2
Plug (adapter::types::unidirectional::AUI)	Ausgang	OUT3
Plug (adapter::types::unidirectional::AUI)	Ausgang	OUT4
Plug (adapter::types::unidirectional::AUI)	Ausgang	OUT5
Plug (adapter::types::unidirectional::AUI)	Ausgang	OUT6
Plug (adapter::types::unidirectional::AUI)	Ausgang	OUT7

Funktionsweise

Der Baustein stellt eine 1-zu-7-Verteilerlogik für AUI-Adapter dar. Das über den Socket `IN` eingehende Signal wird ohne weitere Verarbeitung oder logische Bedingungen auf alle sieben Plugs (`OUT1` … `OUT7`) repliziert. Die Weiterleitung geschieht rein passiv: jedes empfangene Signal wird an jedem Ausgang gleichzeitig bereitgestellt.

Technische Besonderheiten

- **Reiner Adapter-Baustein**: Der FB enthält weder Ereignis- noch Daten-Ein-/Ausgänge – die gesamte Kommunikation erfolgt über die AUI-Adapter.
- **Unidirektional**: Die Adapter sind vom Typ `unidirectional`, d. h. die Daten fließen nur vom Eingang (Socket) zu den Ausgängen (Plugs).
- **Generischer Typ**: Der FB ist als generischer Baustein (`GenericClassName = 'GEN_AUI_SPLIT'`) implementiert und kann für verschiedene AUI-Untertypen wiederverwendet werden.
- **Keine Zustandslogik**: Aufgrund der reinen Verteilungsfunktion besitzt der Baustein keinen internen Zustandsautomaten – er arbeitet kontinuierlich und ohne Verzögerung.

Zustandsübersicht

Es existiert keine Zustandsmaschine. Der Baustein ist rein datenflussgesteuert und besitzt nur einen statischen Betriebszustand.

Anwendungsszenarien

- **Signalverteilung in der Feldautomatisierung**: Ein von einer zentralen Steuerung kommendes AUI-Signal (z. B. Sensordaten oder Steuerbefehle) wird an mehrere parallele Aktoren oder Anzeigeeinheiten weitergeleitet.
- **Test- und Simulationsumgebungen**: Einsatz als Splitter, um ein Eingangssignal gleichzeitig auf mehrere Simulations- oder Diagnoseeinheiten zu geben.
- **Modulare Systeme**: In adapterbasierten Architekturen (z. B. nach IEC 61499) wird der Baustein genutzt, um eine einzige Schnittstelle auf mehrere nachgelagerte Funktionsblöcke aufzuteilen.

Vergleich mit ähnlichen Bausteinen

- **AUI_SPLIT_2, AUI_SPLIT_4** – Bausteine mit gleicher Funktionsweise, jedoch auf 2 bzw. 4 Ausgänge reduziert. Der `AUI_SPLIT_7_UNGATED` bietet die größte Anzahl an Ausgängen und deckt damit umfangreichere Verteilungsanforderungen ab.
- **AUI_MERGE** – das Gegenstück, das mehrere AUI-Eingänge zu einem Ausgang zusammenführt. Im Gegensatz dazu ist der `SPLIT` rein passiv und aggregiert keine Daten.
- **`AUI_SPLIT_7`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

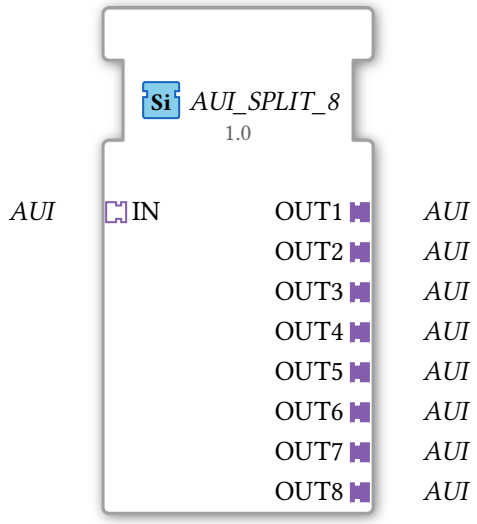
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AUI_SPLIT_7_UNGATED` ist ein einfacher, aber essenzieller Funktionsblock zur unidirektionalen Signalverteilung in adapterbasierten Steuerungssystemen. Dank seines generischen Designs und der klaren 1-zu-N-Struktur lässt er sich flexibel in beliebige AUI-basierte Architekturen integrieren. Für Anwendungen, die eine breite Parallelverteilung ohne Nebenwirkungen erfordern, stellt er eine robuste und wartungsfreundliche Lösung dar.

AUI_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `AUI_SPLIT_8` dient dazu, ein eingehendes AUI-Adapter-Signal (unidirektional) auf acht identische Ausgangsadapter zu verteilen. Er fungiert als passiver Splitter und ermöglicht die einfache Verteilung eines Signalpfades an mehrere nachfolgende Komponenten. Der Funktionsblock ist generisch ausgelegt und nutzt das Adapter-Pattern der IEC 61499.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Adapter	Typ	Richtung
IN	`adapter::types::unidirectional::AUI`	Socket (Eingang)
OUT1	`adapter::types::unidirectional::AUI`	Plug (Ausgang)
OUT2	`adapter::types::unidirectional::AUI`	Plug (Ausgang)
OUT3	`adapter::types::unidirectional::AUI`	Plug (Ausgang)
OUT4	`adapter::types::unidirectional::AUI`	Plug (Ausgang)
OUT5	`adapter::types::unidirectional::AUI`	Plug (Ausgang)
OUT6	`adapter::types::unidirectional::AUI`	Plug (Ausgang)
OUT7	`adapter::types::unidirectional::AUI`	Plug (Ausgang)
OUT8	`adapter::types::unidirectional::AUI`	Plug (Ausgang)

Funktionsweise

Der Baustein leitet alle über die `IN`-Socket empfangenen Daten und Ereignisse des unidirektionalen AUI-Adapters an alle acht `OUT`-Adapter weiter. Es findet keine logische oder zeitliche Verarbeitung statt; die Weiterleitung erfolgt eins-zu-eins und ohne Latenz. Die Adapter sind vom gleichen Typ (`AUI`), sodass die vollständige Signatur (Daten, Ereignisse, mitgelieferte Schnittstellen) unverändert durchgereicht wird.

Technische Besonderheiten

- **Generischer Typ**: Der Baustein ist mittels `GenericClassName` (`GEN_AUI_SPLIT`) als generisch deklariert. Er kann daher für verschiedene AUI-Adapter-Varianten (z. B. unterschiedliche Datenbreiten) eingesetzt werden, sofern der Typ `adapter::types::unidirectional::AUI` zugrunde liegt.
- **Unidirektionalität**: Die Adapter sind unidirektional ausgelegt – eine Datenübertragung ist nur in eine Richtung (von IN zu OUT) vorgesehen. Rückkanäle sind nicht enthalten.
- **Keine Zustandslogik**: Der Baustein besitzt keine inneren Zustände, Algorithmen oder Ereignisverarbeitung. Er ist rein strukturell (passive Verdrahtung).

Zustandsübersicht

Der Baustein besitzt keine Zustände. Er verhält sich statisch und leitet immer alle ankommenden Signale durch.

Anwendungsszenarien

- **Signalverteilung in Automatisierungsnetzwerken**: Ein Sensor oder Steuergerät liefert ein AUI-konformes Signal, das an mehrere Aktoren oder Subsysteme weitergegeben werden muss.
- **Test- und Simulationsumgebungen**: Verteilen eines Testsignals auf mehrere parallele Prüfpfade.
- **Redundante Signalpfade**: Gleichzeitige Versorgung mehrerer unabhängiger Steuerungen mit derselben Information.

Vergleich mit ähnlichen Bausteinen

- **`AUI_MERGE_8`** (Gegenstück): Fasst acht AUI-Eingänge zu einem Ausgang zusammen (inverser Operator).
- **`AUI_SPLIT_2`, `AUI_SPLIT_4`**: Gleiche Funktionalität mit geringerer Ausgangsanzahl.
- **Ereignis-Splitter (z. B. `E_SPLIT`)**: Verteilen nur Ereignisse ohne die mitgeführten Daten eines Adapters. `AUI_SPLIT_8` hingegen repliziert den gesamten Adapter-Inhalt.
- **`AUI_SPLIT_8_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

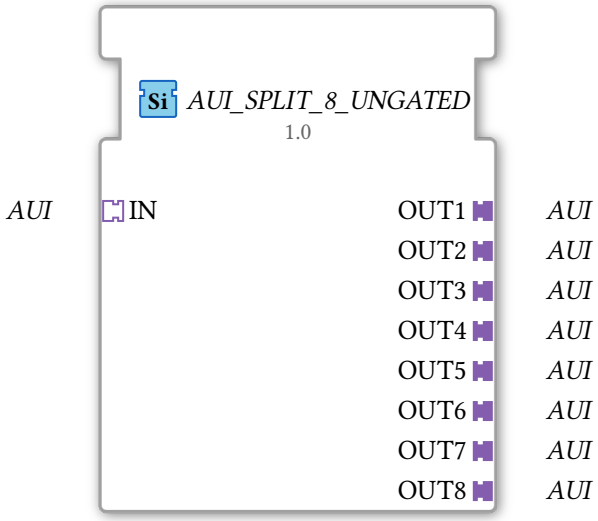
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der `AUI_SPLIT_8` ist ein einfacher, aber essenzieller Baustein zur Vervielfältigung eines unidirektionalen AUI-Signalpfades. Aufgrund seiner generischen Auslegung und rein passiven Struktur eignet er sich für beliebige AUI-basierte Anwendungen, in denen ein Signal an bis zu acht Empfänger verteilt werden muss. Die Implementierung ist schlank, fehlerarm und benötigt keinerlei Konfiguration.

AUI_SPLIT_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUI_SPLIT_8`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `AUI_SPLIT_8_UNGATED` dient dazu, ein eingehendes AUI-Adapter-Signal (unidirektional) auf acht identische Ausgangsadapter zu verteilen. Er fungiert als passiver Splitter und ermöglicht die einfache Verteilung eines Signalpfades an mehrere nachfolgende Komponenten. Der Funktionsblock ist generisch ausgelegt und nutzt das Adapter-Pattern der IEC 61499.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Adapter	Typ	Richtung
IN	`adapter::types::unidirectional::AUI`	Socket (Eingang)
OUT1	`adapter::types::unidirectional::AUI`	Plug (Ausgang)
OUT2	`adapter::types::unidirectional::AUI`	Plug (Ausgang)
OUT3	`adapter::types::unidirectional::AUI`	Plug (Ausgang)
OUT4	`adapter::types::unidirectional::AUI`	Plug (Ausgang)
OUT5	`adapter::types::unidirectional::AUI`	Plug (Ausgang)
OUT6	`adapter::types::unidirectional::AUI`	Plug (Ausgang)
OUT7	`adapter::types::unidirectional::AUI`	Plug (Ausgang)
OUT8	`adapter::types::unidirectional::AUI`	Plug (Ausgang)

Funktionsweise

Der Baustein leitet alle über die `IN`-Socket empfangenen Daten und Ereignisse des unidirektionalen AUI-Adapters an alle acht `OUT`-Adapter weiter. Es findet keine logische oder zeitliche Verarbeitung statt; die Weiterleitung erfolgt eins-zu-eins und ohne Latenz. Die Adapter sind vom gleichen Typ (`AUI`), sodass die vollständige Signatur (Daten, Ereignisse, mitgelieferte Schnittstellen) unverändert durchgereicht wird.

Technische Besonderheiten

- **Generischer Typ**: Der Baustein ist mittels `GenericClassName` (`GEN_AUI_SPLIT`) als generisch deklariert. Er kann daher für verschiedene AUI-Adapter-Varianten (z. B. unterschiedliche Datenbreiten) eingesetzt werden, sofern der Typ `adapter::types::unidirectional::AUI` zugrunde liegt.
- **Unidirektionalität**: Die Adapter sind unidirektional ausgelegt – eine Datenübertragung ist nur in eine Richtung (von IN zu OUT) vorgesehen. Rückkanäle sind nicht enthalten.
- **Keine Zustandslogik**: Der Baustein besitzt keine inneren Zustände, Algorithmen oder Ereignisverarbeitung. Er ist rein strukturell (passive Verdrahtung).

Zustandsübersicht

Der Baustein besitzt keine Zustände. Er verhält sich statisch und leitet immer alle ankommenden Signale durch.

Anwendungsszenarien

- **Signalverteilung in Automatisierungsnetzwerken**: Ein Sensor oder Steuergerät liefert ein AUI-konformes Signal, das an mehrere Aktoren oder Subsysteme weitergegeben werden muss.
- **Test- und Simulationsumgebungen**: Verteilen eines Testsignals auf mehrere parallele Prüfpfade.
- **Redundante Signalpfade**: Gleichzeitige Versorgung mehrerer unabhängiger Steuerungen mit derselben Information.

Vergleich mit ähnlichen Bausteinen

- **`AUI_MERGE_8`** (Gegenstück): Fasst acht AUI-Eingänge zu einem Ausgang zusammen (inverser Operator).
- **`AUI_SPLIT_2`, `AUI_SPLIT_4`**: Gleiche Funktionalität mit geringerer Ausgangsanzahl.
- **Ereignis-Splitter (z. B. `E_SPLIT`)**: Verteilen nur Ereignisse ohne die mitgeführten Daten eines Adapters. `AUI_SPLIT_8_UNGATED` hingegen repliziert den gesamten Adapter-Inhalt.
- **`AUI_SPLIT_8`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

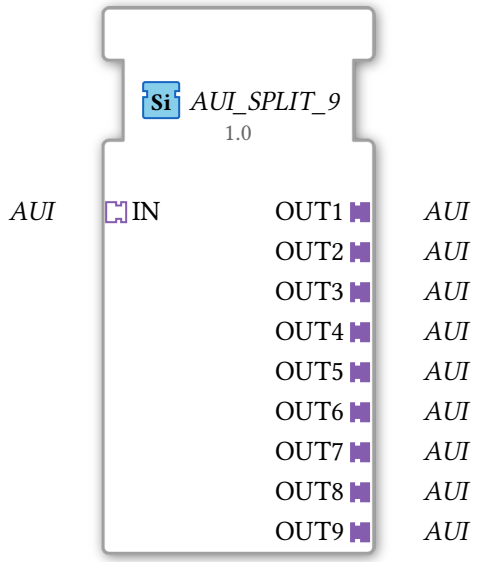
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AUI_SPLIT_8_UNGATED` ist ein einfacher, aber essenzieller Baustein zur Vervielfältigung eines unidirektionalen AUI-Signalpfades. Aufgrund seiner generischen Auslegung und rein passiven Struktur eignet er sich für beliebige AUI-basierte Anwendungen, in denen ein Signal an bis zu acht Empfänger verteilt werden muss. Die Implementierung ist schlank, fehlerarm und benötigt keinerlei Konfiguration.

AUI_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AUI_SPLIT_9** dient als generischer Splitter für unidirektionale AUI-Adapterverbindungen. Er empfängt ein eingehendes AUI-Signal über den Socket `IN` und verteilt dieses identisch auf neun separate Ausgangs-Plugs (`OUT1` bis `OUT9`). Der Baustein ist typisiert als generischer FB (`GEN_AUI_SPLIT`) und kann in verschiedenen Automatisierungsumgebungen eingesetzt werden, in denen ein AUI-Signal an mehrere nachfolgende Komponenten weitergegeben werden muss.

Schnittstellenstruktur

Ereignis-Eingänge

Nicht vorhanden. Der FB arbeitet rein adapterbasiert ohne Ereignissteuerung.

Ereignis-Ausgänge

Nicht vorhanden.

Daten-Eingänge

Nicht vorhanden.

Daten-Ausgänge

Nicht vorhanden.

Adapter

Socket (Eingang):

- `IN` – Typ: `adapter::types::unidirectional::AUI` Empfängt das zu verteilende AUI-Signal.

Plugs (Ausgänge):

- `OUT1` bis `OUT9` – jeweils Typ: `adapter::types::unidirectional::AUI` Neun identische Ausgänge, die das eingehende AUI-Signal unverändert weiterleiten.

Der FB ist ein reiner Adapter-Splitter ohne zusätzliche Logik und arbeitet unidirektional. Alle Plugs sind vom gleichen Typ wie der Socket.

Funktionsweise

Der Baustein kopiert das am Socket `IN` anliegende AUI-Signal auf alle neun Ausgangs-Plugs. Es findet keine Signalaufbereitung, Verzögerung oder Filterung statt – die Ausgänge sind exakte Repliken des Eingangs. Die Verteilung erfolgt parallel und ohne Datenpufferung. Da es sich um einen generischen

Baustein handelt, ist der konkrete Signaltyp (z. B. Feldbusdaten, Messwerte, Steuerbefehle) nicht festgelegt; er ergibt sich aus der AUI-Adapterdefinition.

Technische Besonderheiten

- **Generischer Typ**: Der FB ist als `GEN_AUI_SPLIT` mit einem Type-Hash attributiert, was die dynamische Erzeugung von Instanzen zur Laufzeit in 4diac-IDE unterstützt.
- **Keine Zustandsabhängigkeit**: Der Baustein besitzt keine internen Zustände oder Zeitverhalten – die Ausgänge sind immer eine direkte Wegschaltung des Eingangs.
- **Erweiterbarkeit**: Durch die unidirektionale Adapter-Schnittstelle kann der FB in beliebige AUI-basierte Kommunikationsketten eingefügt werden.
- **Kompatibilität**: Erfordert die Importe `eclipse4diac::core::GenericClassName` und `eclipse4diac::core::TypeHash`, die in der Ziel-4diac-Umgebung verfügbar sein müssen.

Zustandsübersicht

Der FB besitzt keinen Zustandsautomaten. Es findet keine sequentielle Abarbeitung statt; die Signalweitergabe erfolgt statisch und kontinuierlich. Ein expliziter Zustand entfällt.

Anwendungsszenarien

- **Signalverteilung in Feldbussystemen**: Aufteilung eines AUI-basierten Bussignals an mehrere Teilanlagen oder Aktoren.
- **Test- und Debugging-Umgebungen**: Einem Signalgenerator nachgeschaltet, um mehrere Testpunkte parallel zu versorgen.
- **Redundanz oder Parallelbetrieb**: Wenn ein Signal gleichzeitig an mehrere Steuerungen oder Überwachungseinheiten gesendet werden muss.
- **Prototypenentwicklung**: Als Platzhalter für spätere Verzweigungen in der Adapterlogik.

Vergleich mit ähnlichen Bausteinen

- **AUI_SPLIT_4 / AUI_SPLIT_16**: Diese Bausteine bieten die gleiche Funktionalität mit 4 bzw. 16 Ausgängen. Die Wahl hängt von der benötigten Anzahl der Abzweigungen ab.
- **AUI_MERGE_***: Ein Merger fasst mehrere AUI-Eingänge zu einem Ausgang zusammen – die entgegengesetzte Funktion. Der Splitter verteilt hingegen.
- **Generische Splitter anderer Adaptertypen (z. B. Daten-basiert)**: Im Gegensatz zu Daten-Ereignis-Splittern arbeitet AUI_SPLIT_9 rein auf Adapterebene und benötigt keine Ereignissteuerung, was den Implementierungsaufwand reduziert.
- **`AUI_SPLIT_9_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

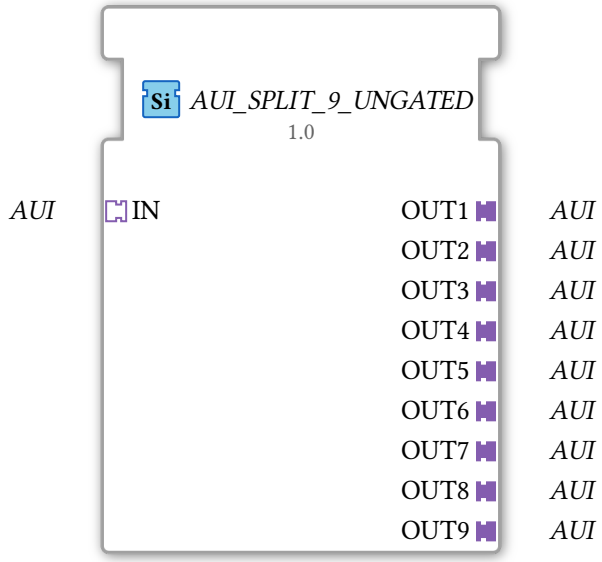
Der **AUI_SPLIT_9** ist ein einfacher, aber effektiver Funktionsbaustein zur Mehrfachverteilung unidirektionaler AUI-Signale. Seine generische Natur ermöglicht eine flexible Wiederverwendung in verschiedenen Projekten. Aufgrund der fehlenden Eigenlogik ist der Baustein ressourcenschonend und leicht verständlich. Er eignet sich besonders für die schnelle Realisierung von 1-zu-n-Verbindungen in adapterbasierten Automatisierungsarchitekturen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUI_SPLIT_9_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `AUI_SPLIT_9`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AUI_SPLIT_9_UNGATED** dient als generischer Splitter für unidirektionale AUI-Adapterverbindungen. Er empfängt ein eingehendes AUI-Signal über den Socket `IN` und verteilt dieses identisch auf neun separate Ausgangs-Plugs (`OUT1` bis `OUT9`). Der Baustein ist typisiert als generischer FB (`GEN_AUI_SPLIT`) und kann in verschiedenen Automatisierungsumgebungen eingesetzt werden, in denen ein AUI-Signal an mehrere nachfolgende Komponenten weitergegeben werden muss.

Schnittstellenstruktur

Ereignis-Eingänge

Nicht vorhanden. Der FB arbeitet rein adapterbasiert ohne Ereignissteuerung.

Ereignis-Ausgänge

Nicht vorhanden.

Daten-Eingänge

Nicht vorhanden.

Daten-Ausgänge

Nicht vorhanden.

Adapter

Socket (Eingang):

- `IN` – Typ: `adapter::types::unidirectional::AUI` Empfängt das zu verteilende AUI-Signal.

Plugs (Ausgänge):

- `OUT1` bis `OUT9` – jeweils Typ: `adapter::types::unidirectional::AUI` Neun identische Ausgänge, die das eingehende AUI-Signal unverändert weiterleiten.

Der FB ist ein reiner Adapter-Splitter ohne zusätzliche Logik und arbeitet unidirektional. Alle Plugs sind vom gleichen Typ wie der Socket.

Funktionsweise

Der Baustein kopiert das am Socket `IN` anliegende AUI-Signal auf alle neun Ausgangs-Plugs. Es findet keine Signalaufbereitung, Verzögerung oder Filterung statt – die Ausgänge sind exakte Repliken des Eingangs. Die Verteilung erfolgt parallel und ohne Datenpufferung. Da es sich um einen generischen Baustein handelt, ist der konkrete Signaltyp (z. B. Feldbusdaten, Messwerte, Steuerbefehle) nicht festgelegt; er ergibt sich aus der AUI-Adapterdefinition.

Technische Besonderheiten

- **Generischer Typ**: Der FB ist als `GEN_AUI_SPLIT` mit einem Type-Hash attributiert, was die dynamische Erzeugung von Instanzen zur Laufzeit in 4diac-IDE unterstützt.
- **Keine Zustandsabhängigkeit**: Der Baustein besitzt keine internen Zustände oder Zeitverhalten – die Ausgänge sind immer eine direkte Wegschaltung des Eingangs.
- **Erweiterbarkeit**: Durch die unidirektionale Adapter-Schnittstelle kann der FB in beliebige AUI-basierte Kommunikationsketten eingefügt werden.
- **Kompatibilität**: Erfordert die Importe `eclipse4diac::core::GenericClassName` und `eclipse4diac::core::TypeHash`, die in der Ziel-4diac-Umgebung verfügbar sein müssen.

Zustandsübersicht

Der FB besitzt keinen Zustandsautomaten. Es findet keine sequentielle Abarbeitung statt; die Signalweitergabe erfolgt statisch und kontinuierlich. Ein expliziter Zustand entfällt.

Anwendungsszenarien

- **Signalverteilung in Feldbussystemen**: Aufteilung eines AUI-basierten Bussignals an mehrere Teilanlagen oder Aktoren.
- **Test- und Debugging-Umgebungen**: Einem Signalgenerator nachgeschaltet, um mehrere Testpunkte parallel zu versorgen.
- **Redundanz oder Parallelbetrieb**: Wenn ein Signal gleichzeitig an mehrere Steuerungen oder Überwachungseinheiten gesendet werden muss.
- **Prototypenentwicklung**: Als Platzhalter für spätere Verzweigungen in der Adapterlogik.

Vergleich mit ähnlichen Bausteinen

- **AUI_SPLIT_4 / AUI_SPLIT_16**: Diese Bausteine bieten die gleiche Funktionalität mit 4 bzw. 16 Ausgängen. Die Wahl hängt von der benötigten Anzahl der Abzweigungen ab.
- **AUI_MERGE_***: Ein Merger fasst mehrere AUI-Eingänge zu einem Ausgang zusammen – die entgegengesetzte Funktion. Der Splitter verteilt hingegen.
- **Generische Splitter anderer Adaptertypen (z. B. Daten-basiert)**: Im Gegensatz zu Daten-Ereignis-Splittern arbeitet AUI_SPLIT_9_UNGATED rein auf Adapterebene und benötigt keine Ereignissteuerung, was den Implementierungsaufwand reduziert.
- `AUI_SPLIT_9`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

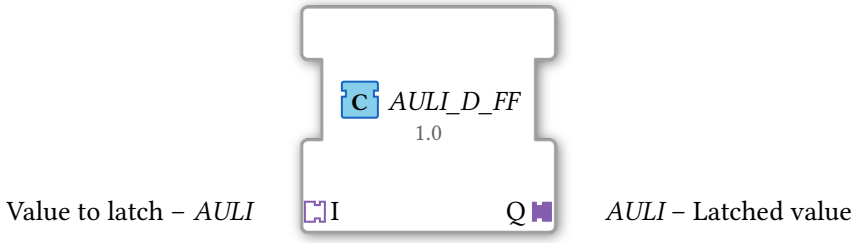
Der **AUI_SPLIT_9_UNGATED** ist ein einfacher, aber effektiver Funktionsbaustein zur Mehrfachverteilung unidirektionaler AUI-Signale. Seine generische Natur ermöglicht eine flexible Wiederverwendung in verschiedenen Projekten. Aufgrund der fehlenden Eigenlogik ist der Baustein ressourcenschonend und leicht verständlich. Er eignet sich besonders für die schnelle Realisierung von 1-zu-n-Verbindungen in adapterbasierten Automatisierungsarchitekturen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

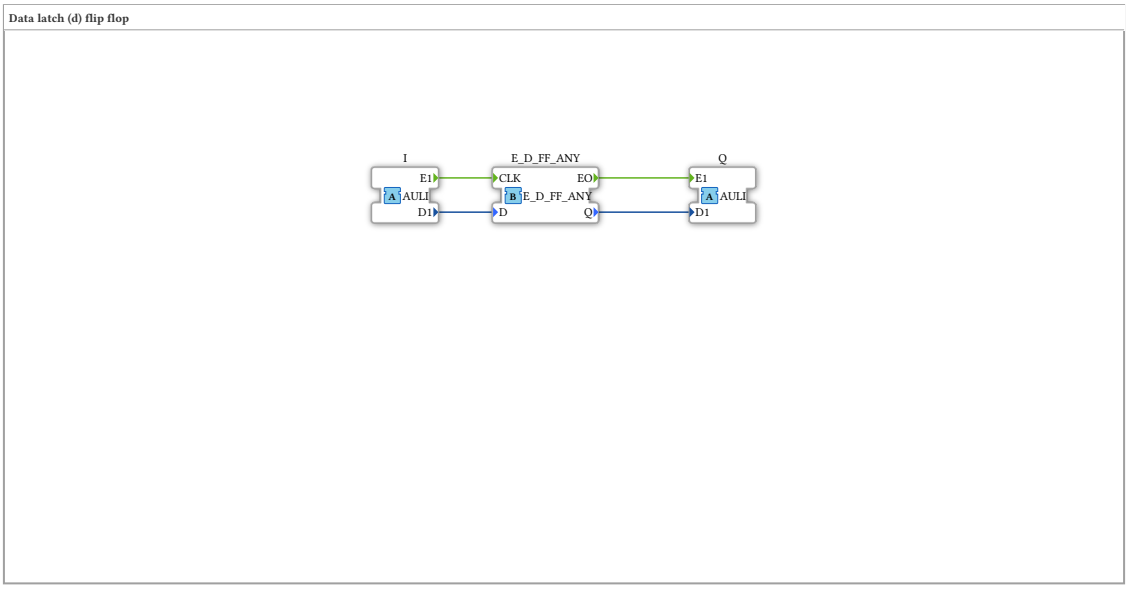
ULINT

AULI_D_FF



Value to latch – *AULI*

AULI – Latched value



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AULI_D_FF realisiert ein taktflankengesteuertes D-Flipflop (Data Latch). Er übernimmt bei einem eingehenden Ereignis den anliegenden Datenwert und hält diesen bis zum nächsten Takt fest. Die Ein- und Ausgabe erfolgt über genormte unidirektionale Adapter, sodass der Baustein in modularen Steuerungsarchitekturen einfach eingebunden werden kann.

Schnittstellenstruktur

Ereignis-Eingänge

- **I.E1** – Taktereignis (positive Flanke): löst das Übernehmen des Datenwertes aus

Ereignis-Ausgänge

- **Q.E1** – Ausgangsereignis: wird nach erfolgreicher Übernahme des Wertes gesendet

Daten-Eingänge

- **I.D1** – zu latchender Datenwert (beliebiger Typ, da der interne Baustein `E_D_FF_ANY` verwendet wird)

Daten-Ausgänge

- **Q.D1** – gelatchter Datenwert (entspricht dem zuletzt übernommenen Wert von I.D1)

Adapter

Der FB besitzt zwei unidirektionale Adapter vom Typ `adapter::types::unidirectional::AULI`:

- **Socket I** (Eingang): nimmt das Taktereignis und den Datenwert auf.
- **Plug Q** (Ausgang): gibt das Quittungsereignis und den gespeicherten Datenwert weiter.

Funktionsweise

Sobald am Eingangsadapter I ein Ereignis an **I.E1** eintrifft, wird der aktuelle Wert von **I.D1** in das interne Flipflop übernommen. Unmittelbar danach wird am Ausgangsadapter Q das Ereignis **Q.E1** gesendet, und der neue Wert wird an **Q.D1** ausgegeben. Der gespeicherte Wert bleibt solange erhalten, bis ein weiteres Taktereignis einen neuen Wert übernimmt.

Der FB arbeitet als **getakteter Speicher** und eignet sich besonders für die Synchronisation von Datenflüssen in ereignisgesteuerten Automatisierungssystemen.

Technische Besonderheiten

- Der Baustein ist als **Composition** realisiert und nutzt intern den Standard-FB `iec61499::events::E_D_FF_ANY`, der die eigentliche Flipflop-Funktionalität bereitstellt.
- Die Verwendung von Adaptern ermöglicht eine **einheitliche Anbindung** an verschiedenste Komponenten, die denselben Adaptertyp unterstützen.
- Es sind keine weiteren Konfigurationsparameter oder Initialisierungen erforderlich – der FB arbeitet nach der ersten Takflanke deterministisch.

Zustandsübersicht

Der interne Zustand des D-Flipflops kann logisch 0 oder 1 sein. Da der Datenwert beliebig sein kann, entspricht der Zustand dem zuletzt gelatchten Wert. Ein explizites Zustandsdiagramm ist nicht erforderlich, da das Verhalten durch die einfache Flankensteuerung vollständig beschrieben wird.

Takt (I.E1)	Vorheriger Q-Wert	Neuer Q-Wert
Ereignis	beliebig	Wert von I.D1
kein Ereignis	unverändert	unverändert

Anwendungsszenarien

- **Datenpufferung** in Produktionslinien: Ein Sensorwert wird bei jedem Takt erfasst und zwischengespeichert.
- **Zustandsmaschinen**: Realisierung von Speicherzellen für Schrittketten, in denen der nächste Zustand vom aktuellen Ausgang abhängt.
- **Synchronisation**: Anpassung asynchroner Datenströme an einen einheitlichen Takt in verteilten Steuerungssystemen.
- **Signalverarbeitung**: Halten eines digitalen Signals für die Dauer eines Taktzyklus.

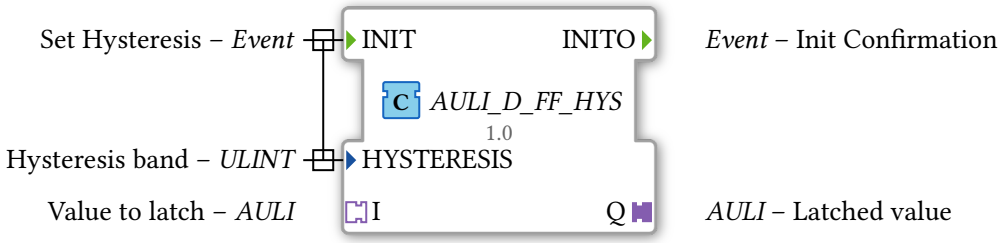
Vergleich mit ähnlichen Bausteinen

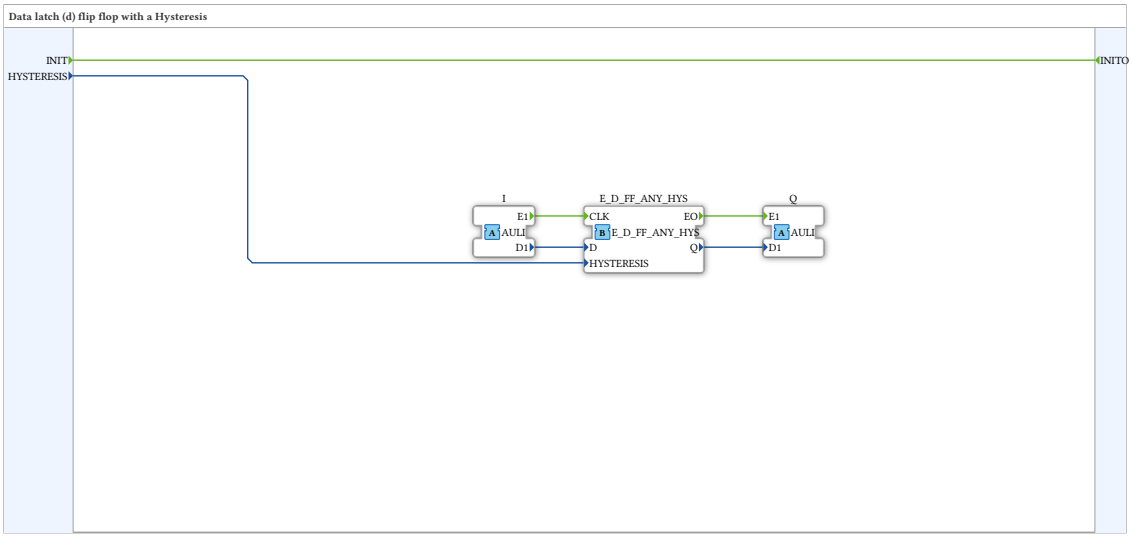
Baustein	Beschreibung	Unterschied zu AULI_D_FF
SR-FF	Set-Reset-Flipflop	Erfordert zwei getrennte Eingänge; kann illegale Zustände haben.
JK-FF	Universelles Flipflop	Komplexer, mit Toggle-Funktion; hier nicht benötigt.
T-FF	Toggle-Flipflop	Ändert den Zustand bei jedem Takt; kein Dateneingang.
AULI_D_FF	D-Flipflop	Einfach, kein Toggle, keine illegalen Zustände, ideal zum Speichern eines Werts.

Fazit

Der AULI_D_FF ist ein kompakter, ereignisgesteuerter D-Flipflop-Baustein, der über standardisierte Adapter kommuniziert. Er eignet sich hervorragend zum Zwischenspehern von Datenwerten in ereignisdiskreten Steuerungsumgebungen. Durch die interne Verwendung eines bewährten Standard-FBs ist das Verhalten robust und deterministisch. Die modulare Adapter-Schnittstelle erleichtert die Integration in komplexe Automatisierungsprojekte.

AULI_D_FF_HYS





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AULI_D_FF_HYS` realisiert ein Data-Latch (D-Flipflop) mit einstellbarer Hysterese. Er kapselt die Logik eines solchen Flipflops in einer adapterbasierten Schnittstelle und erlaubt die Weitergabe von Daten und Ereignissen über unidirektionale Adapter vom Typ `AULI`.

Schnittstellenstruktur

Ereignis-Eingänge

- `INIT` (EInit): Setzt den Hysteresewert und initialisiert den Baustein.

Ereignis-Ausgänge

- `INITO` (EInit): Bestätigt den erfolgreichen Initialisierungsvorgang (durchgeschleiftes INIT).

Daten-Eingänge

- `HYSTERESIS` (ULINT): Legt die Hysteresebreite fest. Der Wert wird beim INIT-Ereignis übernommen.

Daten-Ausgänge

Keine direkten Datenausgänge vorhanden; die Ausgabedaten werden über den Adapter `Q` bereitgestellt.

Adapter

- **Socket `I`** (Typ `adapter::types::unidirectional::AULI`): Empfängt den zu latchnden Wert und das zugehörige Taktsignal.
- **Plug `Q`** (Typ `adapter::types::unidirectional::AULI`): Gibt den gelatchten Wert und das Ausgangsereignis weiter.

Funktionsweise

Der Baustein verwendet intern den Sub-Baustein `E_D_FF_ANY_HYS`. Über den Socket `I` wird das Taktsignal (`I.E1`) an den CLK-Eingang und der Datenwert (`I.D1`) an den D-Eingang des Sub-Bausteins geführt. Der Daten-Eingang `HYSTERESIS` wird direkt mit dem gleichnamigen Eingang des Sub-Bausteins verbunden.

Der Ausgang `Q` des Sub-Bausteins wird auf den Datenausgang des Plugs `Q` (`Q.D1`) übertragen, und das Ereignis `EO` des Sub-Bausteins wird auf das Ausgangsereignis des Plugs `Q` (`Q.E1`) gelegt.

Das INIT-Ereignis wird unverändert als INITO durchgeschleift. Die eigentliche Initialisierung der Hysterese erfolgt durch den Sub-Baustein beim Empfang von INIT, wobei der aktuelle Wert von `HYSTERESIS` übernommen wird.

Die Hysterese verhindert ein sofortiges Umschalten des Ausgangs bei kleinen Schwankungen des Eingangssignals. Erst wenn der Eingangswert den gelatchten Wert um mehr als das Hystereseband über- oder unterschreitet, wird ein neuer Wert übernommen.

Technische Besonderheiten

- Der Baustein ist als reines Zusammenspiel von Adaptern und einem Sub-Baustein definiert (keine eigene ECC- oder SIFB-Logik).
- Die Verwendung des Adapter-Typs `AULI` (unidirektional) erlaubt eine klare Trennung von Daten und Ereignissen in einer standardisierten Schnittstelle.
- Die Hysterese wird nur beim INIT-Ereignis gesetzt, danach bleibt der Wert für die Dauer des Betriebs konstant.
- Der Baustein ist unter der Eclipse Public License 2.0 verfügbar und in der IEC 61499-Annex-A-Standardstruktur definiert.

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine. Die gesamte Zustandslogik ist im Sub-Baustein `E_D_FF_ANY_HYS` implementiert, der vermutlich zwei Zustände (latch setzen, Hysterese prüfen) realisiert.

Anwendungsszenarien

- **Entprellung von Sensoren oder Schaltern**: Ein rauschbehaftetes digitales Signal wird nur dann als stabiler Wert übernommen, wenn es das Hystereseband überschreitet.
- **Signalglättung**: Kleine Fluktuationen eines gemessenen Werts werden unterdrückt, während signifikante Änderungen sofort durchgereicht werden.
- **Datensicherung in asynchronen Systemen**: Der Baustein dient als zuverlässiger Puffer zwischen einer Signalquelle und einer nachfolgenden Verarbeitungseinheit.

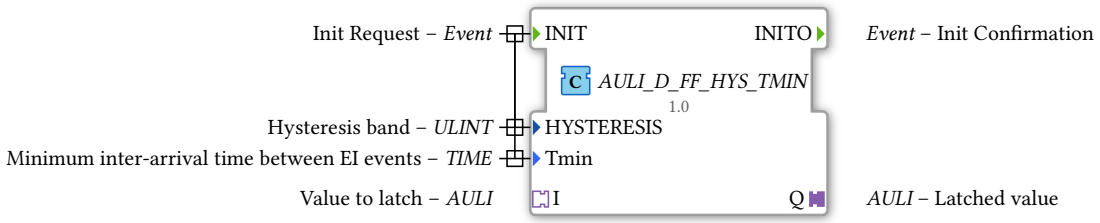
Vergleich mit ähnlichen Bausteinen

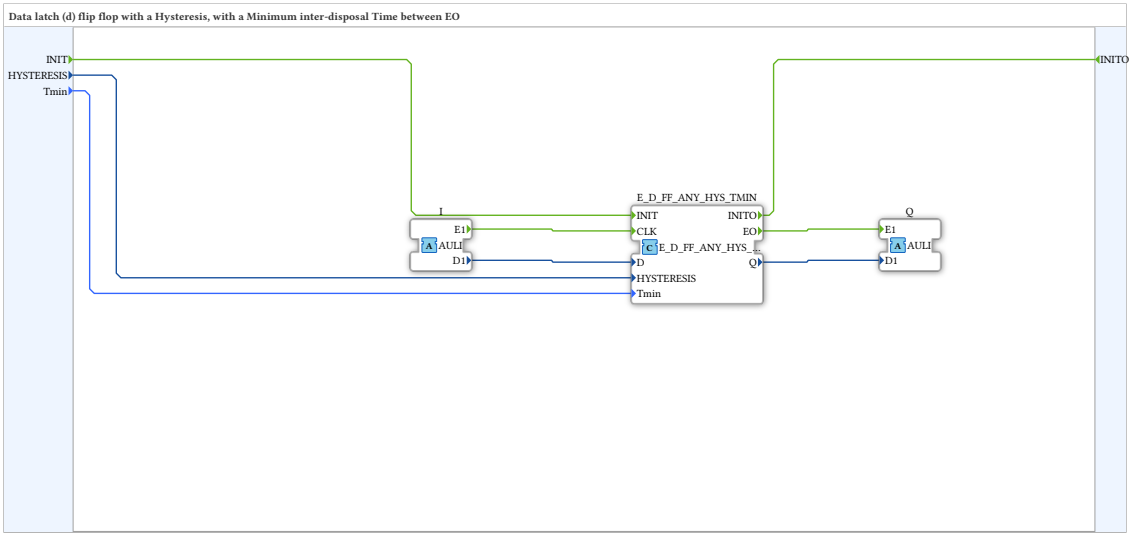
- Ein einfaches D-Flipflop (z.B. `E_D_FF`) übernimmt den Eingangswert bei jeder steigenden Flanke ohne Hysterese. `AULI_D_FF_HYS` erweitert dies um eine konfigurierbare Schaltschwelle.
- Im Vergleich zu einem Schmitt-Trigger-Baustein liegt der Fokus hier auf dem Latch-Verhalten mit zusätzlicher Hysterese, nicht auf der Pegelumsetzung.

Fazit

Der `AULI_D_FF_HYS` ist ein kompakter, adapterbasierter Funktionsbaustein für die zuverlässige Signalaufbereitung mit Hysterese. Durch die Delegation der Logik an einen spezialisierten Sub-Baustein bleibt das Design wartbar und erweiterbar. Er eignet sich besonders für Anwendungen, bei denen Rauschen oder unstete Eingangssignale gefiltert werden müssen, ohne auf ein einfaches Flipflop verzichten zu können.

AULI_D_FF_HYS_TMIN





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AULI_D_FF_HYS_TMIN** realisiert ein taktflankengesteuertes D-Flipflop (Data Latch) mit Hysterese und einer Mindestverweilzeit zwischen aufeinanderfolgenden Ereignissen. Er wird typischerweise eingesetzt, um verrauschte oder schwankende Eingangssignale zu glätten und ein ungewolltes schnelles Umschalten zu unterdrücken. Der Baustein kommuniziert über standardisierte unidirektionale Adapter.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar
`INIT`	`EInit`	Initialisierungsanforderung. Setzt die Parameter Hysterese und Mindestzeit.

Ereignis-Ausgänge

Ereignis	Typ	Kommentar
`INITO`	`EInit`	Bestätigung der erfolgreichen Initialisierung.

Daten-Eingänge

Variable	Typ	Kommentar
`HYSTERESIS`	`ULINT`	Hystereseband als Ganzzahl ohne Vorzeichen. Der Ausgang ändert seinen Wert erst, wenn der Eingang das Band überschreitet.
`Tmin`	`TIME`	Mindestzeit zwischen zwei ausgelösten Ereignissen am Ausgang. Verhindert zu schnelles Schalten.

Daten-Ausgänge

Keine direkten Datenausgänge. Die latched Information wird über den Adapter `Q` bereitgestellt.

Adapter

Adapter	Richtung	Typ	Kommentar
`I` (Socket)	**Eingang**	`adapter::types::unidirectional::AULI`	Aufnehmender Adapter, der das zu latchdende Signal liefert. Enthält die

			Ereignis- und Datensignale `E1` und `D1`.
`Q` (Plug)	**Ausgang**	`adapter::types::unidirectional::AULI`	Ausgebender Adapter, der den aktuell gehaltenen Wert bereitstellt. Signal `E1` zeigt eine Wertänderung an, `D1` enthält den Wert.

Funktionsweise

Der Baustein verwendet intern den FB `E_D_FF_ANY_HYS_TMIN`, der die eigentliche Logik implementiert. Die Arbeitsweise ist:

1. **Initialisierung:** Ein Ereignis `INIT` übernimmt die Parameter `HYSTERESIS` und `Tmin` in den internen Baustein.
2. **Taktflankensteuerung:** Das Ereignis `I.E1` (Takteingang) bewirkt, dass der aktuelle Datenwert `I.D1` übernommen wird – sofern die Hysterese- und Timing-Bedingungen erfüllt sind.
3. **Hysterese:** Der neue Ausgangswert wird nur dann übernommen, wenn die Differenz zwischen dem aktuellen Eingang `D1` und dem letzten gültigen Ausgang den Betrag der Hysterese überschreitet.
4. **Mindestzeit:** Nach jeder Übernahme (Ausgabe eines Ereignisses auf `Q.E1`) wird für die Dauer `Tmin` keine weitere Übernahme zugelassen.
5. **Ausgabe:** Bei einer gültigen Flanke wird der neue Wert über `Q.D1` ausgegeben und ein Ereignis auf `Q.E1` erzeugt.

Technische Besonderheiten

- **Hysterese:** Unterdrückt Rauschen oder kleine Schwankungen am Eingang, indem ein Totband definiert wird. Der Ausgang ändert sich nur bei signifikanten Abweichungen.
- **Mindestverweilzeit (`Tmin`):** Erzwingt eine Pause zwischen aufeinanderfolgenden Ausgangsereignissen. Dadurch werden mechanische oder systembedingte Mindestumschaltzeiten eingehalten.
- **Adapter-Schnittstelle:** Die Verwendung eines unidirektionalen Adaptertyps ermöglicht eine lose Kopplung und Wiederverwendbarkeit in verschiedenen Umgebungen.
- **Interner FB:** Die eigentliche Logik ist in einen getypten FB ausgelagert, was die Testbarkeit und Wartbarkeit verbessert.

Zustandsübersicht

Der Baustein besitzt keine expliziten eigenen Zustände, sondern delegiert an den internen FB. Die wesentlichen Verhaltensweisen lassen sich jedoch wie folgt beschreiben:

Phase	Beschreibung
Initial	Nach dem Einschalten werden `HYSTERESIS` und `Tmin` noch nicht angewendet. Ein `INIT`-Ereignis muss die Parameter setzen.
Bereit	Nach erfolgreicher Initialisierung wartet der FB auf Taktereignisse am Eingang `I.E1`.
Sperre (Tmin aktiv)	Nach einem Ausgangsereignis wird für die Dauer `Tmin` jede weitere Übernahme blockiert.
Übernahme (bei gültiger Flanke)	Wenn die Hysterese überschritten ist und keine Sperre vorliegt, wird der aktuelle Eingangswert übernommen und ausgegeben.

Anwendungsszenarien

- **Entprellung von Sensorsignalen:** Ein Taster oder Näherungsschalter liefert ein schwankendes Signal. Mit Hysterese und Mindestzeit wird eine saubere Schaltflanke erzeugt.
- **Datenlogik mit Signalaufbereitung:** In der Automatisierungstechnik, wo ein digitaler Eingangswert übernommen, aber durch Rauschen oder Prellen verfälscht werden kann.
- **Zeitgesteuerte Stellglieder:** Wenn ein Aktuator eine minimale Umschaltzeit benötigt (z.B. Ventile), verhindert `Tmin` eine zu schnelle Ansteuerung.

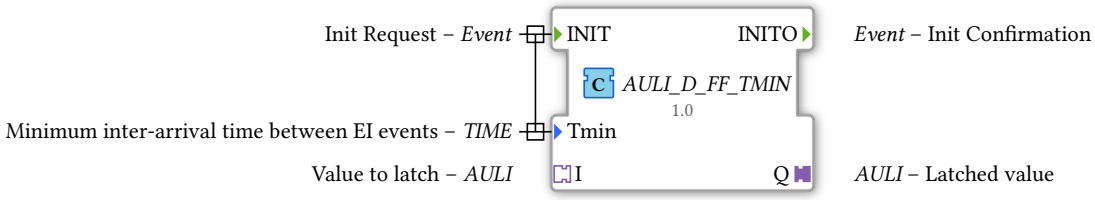
Vergleich mit ähnlichen Bausteinen

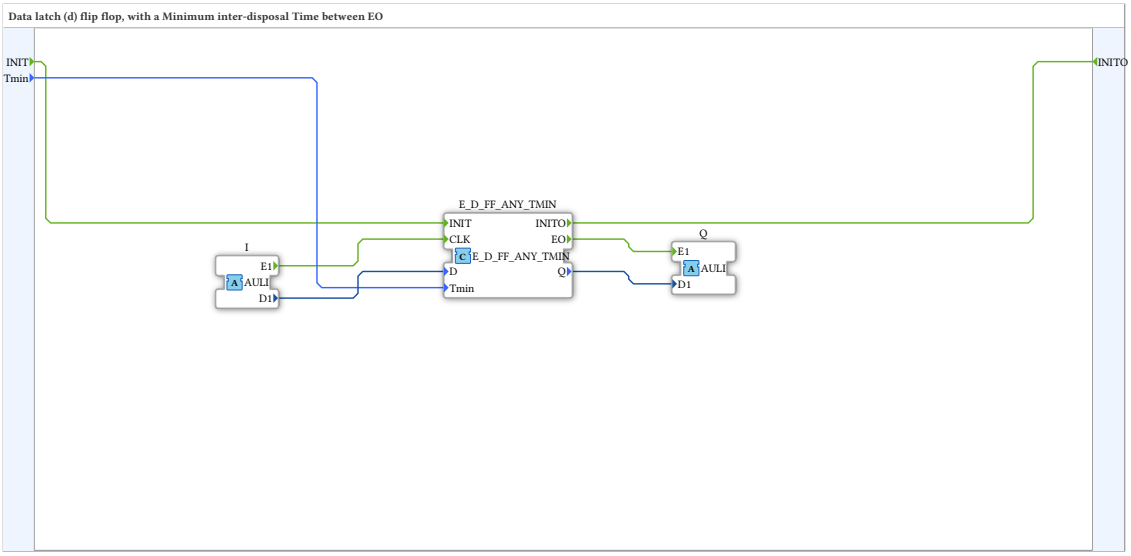
Baustein	Eigenschaften
Einfaches D-FF	Keine Hysterese, keine Verzögerung – übernimmt jede Taktflanke sofort.
D-FF mit Hysterese	Enthält nur das Hystereseband, keine zeitliche Begrenzung zwischen Ausgaben.
D-FF mit Tmin	Nur Mindestzeit ohne Hysterese – kann bei Rauschen trotzdem schnell schalten.
AULI_D_FF_HYS_TMIN	Kombiniert Hysterese und Mindestverweilzeit – robuster gegenüber Rauschen und schützt nachgeschaltete Komponenten vor zu schnellem Schalten.

Fazit

Der **AULI_D_FF_HYS_TMIN** bietet eine zuverlässige Kombination aus Hysterese und Zeitbegrenzung für taktgesteuerte Signalübernahmen. Er eignet sich besonders für industrielle Steuerungen, bei denen Störungen unterdrückt und Aktuatoren geschont werden müssen. Die modulare Adapter-Schnittstelle erleichtert die Integration in bestehende IEC 61499-Anwendungen.

AULI_D_FF_TMIN





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AULI_D_FF_TMIN** realisiert ein Daten-Latch (D-Flipflop) mit einer einstellbaren Mindestverzögerungszeit zwischen aufeinanderfolgenden Ereignisausgaben. Er dient als Speicherglied für asynchrone Ereignis-/Datenschnittstellen und stellt sicher, dass nach einer Datenübernahme ein definiertes Zeitintervall vergeht, bevor ein weiteres Ereignis ausgelöst wird. Der Baustein arbeitet ausschließlich über Adapter-Schnittstellen vom Typ `AULI` (unidirektional) und ist dadurch besonders für modulare und verteilte Steuerungsarchitekturen geeignet.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Initialisierungsanforderung; wird mit `Tmin` synchronisiert

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Bestätigung der Initialisierung

Daten-Eingänge

Name	Typ	Kommentar
Tmin	TIME	Mindestzeit zwischen zwei Ereignisausgaben (EO) am Ausgangsadapter

Daten-Ausgänge

Keine

Adapter

Name	Rolle	Typ	Kommentar
I	Socket	AULI	Eingang: zu latchierender Wert (Ereignis E1, Daten D1)
Q	Plug	AULI	Ausgang: latchender Wert (Ereignis E1, Daten D1)

Funktionsweise

Der Baustein kapselt einen internen `E_D_FF_ANY_TMIN`-Funktionsblock aus der IEC-61499-Bibliothek. Er arbeitet als flankengesteuertes D-Flipflop mit folgenden Abläufen:

1. **Initialisierung** – Ein Ereignis am Eingang `INIT` setzt den internen Zustand zurück und übernimmt den Parameter `Tmin`. Der Ausgang `INITO` quittiert die Initialisierung.
2. **Ereignis-gesteuerte Datenübernahme** –
   - Ein Ereignis am Adapter-Eingang `I.E1` (als Taktsignal) übernimmt den aktuellen Datenwert `I.D1` in das interne Flipflop.
   - Gleichzeitig wird die Mindestzeit `Tmin` gestartet.
   - Nach Ablauf von `Tmin` wird ein Ereignis am Adapter-Ausgang `Q.E1` ausgelöst und der übernommene Datenwert `Q.D1` ausgegeben.
3. **Sperrzeit** – Während `Tmin` noch aktiv ist, wird ein erneutes Eingangsereignis ignoriert oder gepuffert (abhängig von der internen Implementierung). Erst nach Ablauf der Zeit kann der nächste Wert übernommen werden.

Technische Besonderheiten

- **Unidirektionale Adapter** – Der Baustein verwendet ausschließlich den standardisierten Adaptertyp `AULI` für Ereignisse und Daten, was eine einfache Kopplung mit anderen AULI-kompatiblen Bausteinen ermöglicht.
- **Mindestzeit (`Tmin`)** – Erzwingt eine minimale Pause zwischen zwei aufeinanderfolgenden Ausgangsereignissen. Dies verhindert Datenverluste oder Überlastungen in nachgeschalteten Komponenten mit begrenzter Verarbeitungsgeschwindigkeit.
- **Keine eigenen Daten-Ausgänge** – Die Ausgabedaten werden ausschließlich über den Plugin-Adapter `Q` bereitgestellt, was die Modularität erhöht.

Zustandsübersicht

Der interne FB `E_D_FF_ANY_TMIN` kann vereinfacht durch drei Zustände beschrieben werden:

- **IDLE** – Warten auf ein Taktereignis am Eingang `I.E1`.
- **LATCHED** – Daten wurden übernommen und `Tmin`-Timer läuft. Keine neue Übernahme möglich.

- **OUTPUT** – `Tmin` abgelaufen; Ausgangsereignis wird ausgelöst und zurück in den IDLE-Zustand gewechselt.

Die Initialisierung (`INIT`) setzt den Baustein in den IDLE-Zustand zurück.

Anwendungsszenarien

- **Datensynchronisation** in zeitkritischen Regelkreisen, bei denen ein Sensorwert nur in einem Mindestabstand abgetastet werden darf.
- **Pufferung** von asynchronen Ereignissen, um eine Lastspitze im Datenstrom zu glätten.
- **Kaskadierung** von Flipflops mit unterschiedlichen Mindestzeiten, um eine zeitlich gestaffelte Signalverarbeitung zu realisieren.
- **Schnittstelle zwischen schneller und langsamer Komponente** (z. B. zwischen einer schnellen Logik und einem langsamen Aktor).

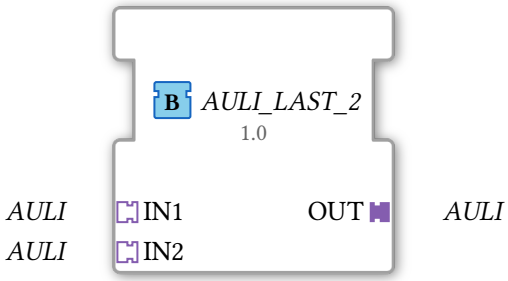
Vergleich mit ähnlichen Bausteinen

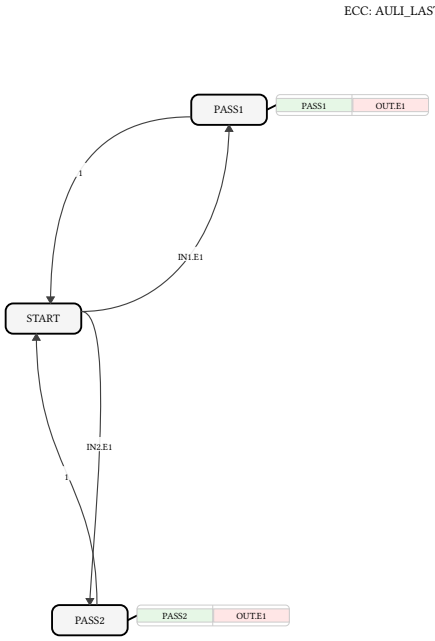
- **E_D_FF** – Einfaches D-Flipflop ohne Zeitsteuerung; übernimmt Daten bei jeder Taktflanke sofort. Der `AULI_D_FF_TMIN` erweitert dies um eine einstellbare Mindestzeit.
- **E_D_FF_TMIN** – Gleicher interner Kern, jedoch mit direkten Ereignis-/Datenschnittstellen anstelle von Adaptern. Die Adapter-Version `AULI_D_FF_TMIN` ist für Systeme entworfen, die auf standardisierte, wiederverwendbare Schnittstellen setzen.
- **E_D_FF_ANY_TMIN** (direkt genutzt) – Bietet dieselbe Funktionalität, ist aber für beliebige Datentypen ausgelegt. `AULI_D_FF_TMIN` ist speziell auf den AULI-Adapter zugeschnitten.

Fazit

Der `AULI_D_FF_TMIN` ist ein spezialisierter D-Flipflop-Baustein für Ereignis-/Daten-Adapter-Schnittstellen, der durch die integrierte Mindestzeit `Tmin` eine zuverlässige und vorhersagbare Datenübernahme in zeitkritischen Umgebungen gewährleistet. Seine modulare Bauweise und die Einhaltung des IEC-61499-Anhangs A machen ihn zu einer robusten Komponente für industrielle Steuerungsanwendungen.

AULI_LAST_2





Einleitung

Der AULI_LAST_2 Funktionsblock führt zwei gleichartige ULINT-Adapter-Signale (**IN1**, **IN2**) auf einen gemeinsamen Ausgang (**OUT**) zusammen, nach dem Prinzip **Last Writer Wins**: welcher der beiden Sockets zuletzt ein Ereignis geschrieben hat, gewinnt, und sein Datenwert wird sofort und unverändert an OUT durchgereicht. Es handelt sich ausdrücklich nicht um ein naives Zusammenführen (z. B. Mittelwert oder ODER-Verknüpfung), sondern um eine reine Arbitrierung nach zeitlicher Reihenfolge.

Schnittstellenstruktur

Adapter

Eingangsadapter (Sockets):

- **IN1**: ULINT-Adapter (unidirectional) - erste Signalquelle
- **IN2**: ULINT-Adapter (unidirectional) - zweite Signalquelle

Ausgangsadapter (Plug):

- **OUT**: ULINT-Adapter (unidirectional) - zusammengeführtes Signal

Funktionsweise

Der Baustein ist als einfacher Basic-FB mit einer 3-Zustands-ECC umgesetzt (START, PASS1, PASS2). Trifft an IN1 ein Ereignis ein, wechselt die ECC nach PASS1: der aktuelle Datenwert von IN1 wird nach OUT kopiert und das Adapter-Ereignis an OUT gesendet. Trifft an IN2 ein Ereignis ein, geschieht dasselbe symmetrisch über PASS2. Nach jeder Durchreichung kehrt die ECC sofort wieder nach START zurück und ist bereit für das nächste Ereignis von IN1 oder IN2.

Technische Besonderheiten

- Last-Writer-Wins-Semantik statt naiver Datenverknüpfung
- Einfacher Basic-FB (ECC mit 3 Zuständen), kein zusätzlicher Zwischenspeicher nötig
- Keine Signalverzögerung: der Wert wird im selben Zyklus durchgereicht, in dem das auslösende Ereignis eintrifft
- Beide Eingänge sind gleichberechtigt - es gibt keine feste Priorität außer der zeitlichen Reihenfolge

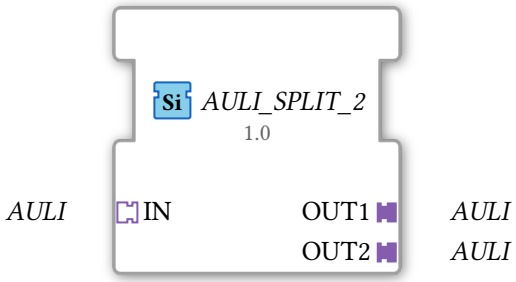
Anwendungsszenarien

- Zusammenführen zweier gleichwertiger ULINT-Signalquellen zu einem einzigen nachgeschalteten Adapterpfad (z. B. zwei redundante Sensoren, oder ein manueller und ein automatischer Sollwertgeber)
- Arbitrierung zwischen zwei Bedienstellen, die denselben ULINT-Wert schreiben dürfen
- Vereinfachung von Netzwerken, die sonst ein `F_SEL`/`E_RS`-Konstrukt inline verdrahten müssten

⚖️ Vergleich mit ähnlichen Bausteinen

AULI_LAST_2 ist die Umkehrung von `AULI_SPLIT_2`: AULI_SPLIT_2 verteilt 1 Eingang auf 2 Ausgänge, AULI_LAST_2 führt 2 Eingänge auf 1 Ausgang zusammen. Für Mischbetrieb zwischen einem Daten- und einem reinen Ereignisadapter (z. B. AX + AE) ist stattdessen `AX_AE_MERGE` gedacht, das nur das Ereignis mischt, ohne den Datenwert zu verändern.

AULI_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AULI_SPLIT_2** dient dazu, einen eingehenden AULI-Adapter (unidirektional) auf zwei identische Ausgänge zu verteilen. Er ist als generischer Baustein realisiert und ermöglicht die einfache Signalvervielfachung in IEC 61499-basierten Systemen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	IN	AULI (unidirektional)	Eingehender AULI-Datenstrom
Plug (Ausgang)	OUT1	AULI (unidirektional)	Erster Ausgang, identische Kopie von IN
Plug (Ausgang)	OUT2	AULI (unidirektional)	Zweiter Ausgang, identische Kopie von IN

Funktionsweise

Der Baustein leitet die über den **IN**-Socket empfangenen AULI-Daten unverändert an beide Ausgänge **OUT1** und **OUT2** weiter. Es findet keine Datenmanipulation, Pufferung oder Protokollumsetzung statt. Die Verteilung erfolgt passiv und ohne Verzögerung.

Technische Besonderheiten

- **Generischer Typ**: Der FB ist über das Attribut `eclipse4diac::core::GenericClassName` als `'GEN_AULI_SPLIT'` deklariert, was eine flexible Wiederverwendung in verschiedenen Engineering-Umgebungen ermöglicht.
- **Keine Zustandshaltung**: Der FB besitzt keinen internen Zustandsautomaten (ECC) und kein Speicherverhalten. Er ist vollständig passiv.
- **Reine Adapter-Schnittstelle**: Es werden ausschließlich Adapter (`Plugs` und `Sockets`) verwendet; Daten- oder Ereigniseingänge sind nicht vorhanden.

Zustandsübersicht

Da der Funktionsblock keine interne Logik oder Zustandsmaschine enthält, existiert keine Zustandsbeschreibung. Das Verhalten beschränkt sich auf die konstante Weiterleitung des eingehenden Signals.

Anwendungsszenarien

- **Signalverteilung**: Aufteilung eines AULI-Datenstroms an zwei nachfolgende Funktionsblöcke, z. B. zur parallelen Verarbeitung oder Steuerung.
- **Redundante Übertragung**: Bereitstellung eines zweiten, identischen Datenpfads für Überwachungs- oder Sicherheitszwecke.
- **Punkt-zu-Mehrpunkt-Kommunikation**: Ermöglichung einer einfachen 1-zu-2-Verdrahtung in Adapter-basierten Architekturen.

Vergleich mit ähnlichen Bausteinen

Baustein	Funktion
AULI_SPLIT_2	Split auf zwei Ausgänge (identisch zu IN)
AULI_SPLIT_N	Generalisierte Variante mit konfigurierbarer Anzahl an Ausgängen
AULI_MERGE	Zusammenführung mehrerer Eingänge auf einen Ausgang

Während **AULI_SPLIT_2** eine feste 1-zu-2-Aufteilung vornimmt, erlauben generische Splitter eine flexible Anzahl von Ausgängen. Zusammenführer wie **AULI_MERGE** lösen die umgekehrte Aufgabe.

- `AULI_SPLIT_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der **AULI_SPLIT_2** ist ein minimaler und effizienter Baustein zur Signalvervielfachung unidirektionaler AULI-Adapter. Aufgrund seiner passiven, zustandslosen Natur eignet er sich ideal für Echtzeitanwendungen, in denen Kopien eines Datenstroms ohne zusätzliche Latenz oder Logik benötigt werden. Seine generische Implementierung erleichtert den Einsatz in unterschiedlichen Entwicklungswerkzeugen und Bibliotheken.

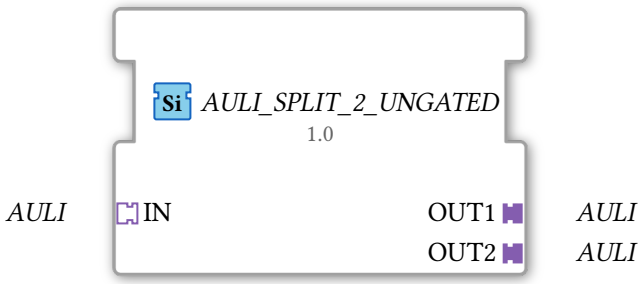
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AULI_SPLIT_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AULI_SPLIT_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die

sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AULI_SPLIT_2_UNGATED** dient dazu, einen eingehenden AULI-Adapter (unidirektional) auf zwei identische Ausgänge zu verteilen. Er ist als generischer Baustein realisiert und ermöglicht die einfache Signalvervielfachung in IEC 61499-basierten Systemen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	IN	AULI (unidirektional)	Eingehender AULI-Datenstrom
Plug (Ausgang)	OUT1	AULI (unidirektional)	Erster Ausgang, identische Kopie von IN
Plug (Ausgang)	OUT2	AULI (unidirektional)	Zweiter Ausgang, identische Kopie von IN

Funktionsweise

Der Baustein leitet die über den **IN**-Socket empfangenen AULI-Daten unverändert an beide Ausgänge **OUT1** und **OUT2** weiter. Es findet keine Datenmanipulation, Pufferung oder Protokollumsetzung statt. Die Verteilung erfolgt passiv und ohne Verzögerung.

Technische Besonderheiten

- **Generischer Typ**: Der FB ist über das Attribut `eclipse4diac::core::GenericClassName` als `'GEN_AULI_SPLIT'` deklariert, was eine flexible Wiederverwendung in verschiedenen Engineering-Umgebungen ermöglicht.
- **Keine Zustandshaltung**: Der FB besitzt keinen internen Zustandsautomaten (ECC) und kein Speicherverhalten. Er ist vollständig passiv.
- **Reine Adapter-Schnittstelle**: Es werden ausschließlich Adapter (`Plugs` und `Sockets`) verwendet; Daten- oder Ereigniseingänge sind nicht vorhanden.

Zustandsübersicht

Da der Funktionsblock keine interne Logik oder Zustandsmaschine enthält, existiert keine Zustandsbeschreibung. Das Verhalten beschränkt sich auf die konstante Weiterleitung des eingehenden Signals.

Anwendungsszenarien

- **Signalverteilung**: Aufteilung eines AULI-Datenstroms an zwei nachfolgende Funktionsblöcke, z. B. zur parallelen Verarbeitung oder Steuerung.
- **Redundante Übertragung**: Bereitstellung eines zweiten, identischen Datenpfads für Überwachungs- oder Sicherheitszwecke.
- **Punkt-zu-Mehrpunkt-Kommunikation**: Ermöglichung einer einfachen 1-zu-2-Verdrahtung in Adapter-basierten Architekturen.

Vergleich mit ähnlichen Bausteinen

Baustein	Funktion
AULI_SPLIT_2_UNGATED	Split auf zwei Ausgänge (identisch zu IN)
AULI_SPLIT_N	Generalisierte Variante mit konfigurierbarer Anzahl an Ausgängen
AULI_MERGE	Zusammenführung mehrerer Eingänge auf einen Ausgang

Während **AULI_SPLIT_2_UNGATED** eine feste 1-zu-2-Aufteilung vornimmt, erlauben generische Splitter eine flexible Anzahl von Ausgängen. Zusammenführer wie **AULI_MERGE** lösen die umgekehrte Aufgabe.

- `AULI_SPLIT_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

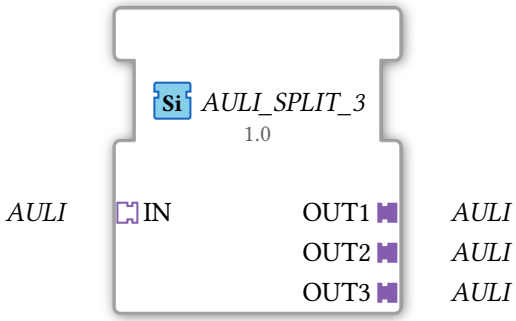
Fazit

Der **AULI_SPLIT_2_UNGATED** ist ein minimaler und effizienter Baustein zur Signalvervielfachung unidirektionaler AULI-Adapter. Aufgrund seiner passiven, zustandslosen Natur eignet er sich ideal für Echtzeitanwendungen, in denen Kopien eines Datenstroms ohne zusätzliche Latenz oder Logik benötigt werden. Seine generische Implementierung erleichtert den Einsatz in unterschiedlichen Entwicklungswerkzeugen und Bibliotheken.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AULI_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AULI_SPLIT_3** dient der Verteilung eines einzelnen AULI-Adaptersignals auf drei identische Ausgänge. Er fungiert als einfacher Splitter für unidirektionale AULI-Verbindungen und er-

möglicht den Broadcast eines Signals an mehrere nachfolgende Komponenten. Der Baustein ist generisch ausgelegt und kann für verschiedene AULI-Typen instantiiert werden.

Schnittstellenstruktur

Ereignis-Eingänge
Keine.

Ereignis-Ausgänge
Keine.

Daten-Eingänge
Keine.

Daten-Ausgänge
Keine.

Adapter

Name	Typ	Beschreibung
IN	AULI (unidirectional)	Eingangsadapter – empfängt das zu verteilende Signal.
OUT1	AULI (unidirectional)	Erster Ausgangsadapter.
OUT2	AULI (unidirectional)	Zweiter Ausgangsadapter.
OUT3	AULI (unidirectional)	Dritter Ausgangsadapter.

Funktionsweise
Der Baustein leitet jede am Eingangsadapter **IN** ankommende AULI-Information (sowohl Ereignisse als auch zugehörige Daten) direkt und ohne Verzögerung an alle drei Ausgangsadapter **OUT1**, **OUT2** und **OUT3** weiter. Es findet keine Filterung, Transformation oder Pufferung statt. Die Ausgänge sind identisch und arbeiten parallel, sodass das Eingangssignal gleichzeitig an drei unabhängige Empfänger gesendet wird.

Technische Besonderheiten
- **Generischer Typ:** Der FB ist als generischer Baustein deklariert (GenericClassName `'GEN_AULI_SPLIT'`). Er kann mit verschiedenen konkreten AULI-Adaptertypen verwendet werden, sofern diese dem unidirektionalen AULI-Interface entsprechen.
- **Latenz:** Durch die reine Weiterleitung ohne interne Logik ist die Latenz minimal und beschränkt sich auf die Durchlaufzeit des Laufzeitsystems.
- **Copyright:** Der Baustein unterliegt dem Eclipse Public License 2.0 (SPDX-License-Identifier: EPL-2.0).

Zustandsübersicht
Der Baustein besitzt keinen internen Zustandsautomaten. Die Funktionalität ist rein kombinatorisch und kontinuierlich aktiv – sobald ein Signal am Eingang anliegt, wird es an alle Ausgänge weitergegeben.

Anwendungsszenarien
- **Signalverteilung:** Ein AULI-Signal (z. B. Steuerbefehle, Messwerte) soll an mehrere unabhängige Aktoren oder Subsysteme gesendet werden.
- **Test- und Simulationsaufbauten:** Ein Testsignal wird aufgezeichnet oder von einer Quelle gesendet und parallel an mehrere Überwachungs- oder Analyseblöcke verteilt.
- **Redundante Verarbeitung:** Derselbe Eingangswert wird von mehreren parallel arbeitenden Logiken gleichzeitig benötigt.

Vergleich mit ähnlichen Bausteinen
Im Gegensatz zu einem **AULI_MERGE** (der mehrere Eingänge auf einen Ausgang zusammenführt) oder einem **AULI_SELECT** (der wahlweise einen von mehreren Eingängen durchschaltet), bietet **AULI_SPLIT_3** eine reine Fan-out-Funktion. Es gibt keinen Entscheidungsmechanismus, keine Priori-

sierung und keine Datenmodifikation. Ähnliche Splitter mit anderer Ausgangsanzahl (z. B. SPLIT_2) unterscheiden sich lediglich in der Anzahl der Ausgänge.

- `AULI_SPLIT_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

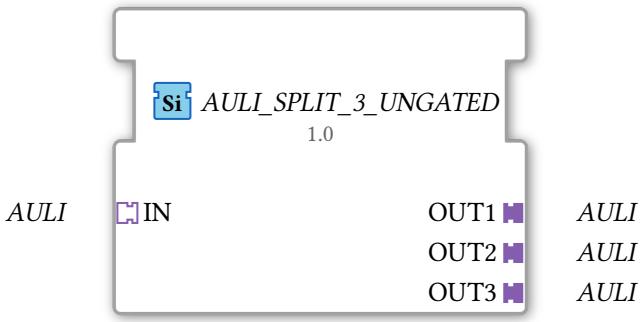
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

AULI_SPLIT_3 ist ein einfacher, aber essenzieller Baustein zur Vervielfachung unidirektionaler AULI-Signale. Durch die generische Auslegung und die verzögerungsfreie Weiterleitung eignet er sich besonders für Broadcast-Szenarien in der Automatisierungstechnik, wo ein Signal gleichzeitig an mehrere Verbraucher übergeben werden muss.

AULI_SPLIT_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AULI_SPLIT_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AULI_SPLIT_3_UNGATED** dient der Verteilung eines einzelnen AULI-Adaptersignals auf drei identische Ausgänge. Er fungiert als einfacher Splitter für unidirektionale AULI-Verbindungen und ermöglicht den Broadcast eines Signals an mehrere nachfolgende Komponenten. Der Baustein ist generisch ausgelegt und kann für verschiedene AULI-Typen instantiiert werden.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Name	Typ	Beschreibung
IN	AULI (unidirectional)	Eingangsadapter – empfängt das zu verteilende Signal.
OUT1	AULI (unidirectional)	Erster Ausgangsadapter.
OUT2	AULI (unidirectional)	Zweiter Ausgangsadapter.
OUT3	AULI (unidirectional)	Dritter Ausgangsadapter.

Funktionsweise

Der Baustein leitet jede am Eingangsadapter **IN** ankommende AULI-Information (sowohl Ereignisse als auch zugehörige Daten) direkt und ohne Verzögerung an alle drei Ausgangsadapter **OUT1**, **OUT2** und **OUT3** weiter. Es findet keine Filterung, Transformation oder Pufferung statt. Die Ausgänge sind identisch und arbeiten parallel, sodass das Eingangssignal gleichzeitig an drei unabhängige Empfänger gesendet wird.

Technische Besonderheiten

- **Generischer Typ:** Der FB ist als generischer Baustein deklariert (GenericClassName `'GEN_AULI_SPLIT'`). Er kann mit verschiedenen konkreten AULI-Adaptertypen verwendet werden, sofern diese dem unidirektionalen AULI-Interface entsprechen.
- **Latenz:** Durch die reine Weiterleitung ohne interne Logik ist die Latenz minimal und beschränkt sich auf die Durchlaufzeit des Laufzeitsystems.
- **Copyright:** Der Baustein unterliegt dem Eclipse Public License 2.0 (SPDX-License-Identifier: EPL-2.0).

Zustandsübersicht

Der Baustein besitzt keinen internen Zustandsautomaten. Die Funktionalität ist rein kombinatorisch und kontinuierlich aktiv – sobald ein Signal am Eingang anliegt, wird es an alle Ausgänge weitergegeben.

Anwendungsszenarien

- **Signalverteilung:** Ein AULI-Signal (z. B. Steuerbefehle, Messwerte) soll an mehrere unabhängige Aktoren oder Subsysteme gesendet werden.
- **Test- und Simulationsaufbauten:** Ein Testsignal wird aufgezeichnet oder von einer Quelle gesendet und parallel an mehrere Überwachungs- oder Analyseblöcke verteilt.
- **Redundante Verarbeitung:** Derselbe Eingangswert wird von mehreren parallel arbeitenden Logiken gleichzeitig benötigt.

Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu einem **AULI_MERGE** (der mehrere Eingänge auf einen Ausgang zusammenführt) oder einem **AULI_SELECT** (der wahlweise einen von mehreren Eingängen durchschaltet), bietet **AULI_SPLIT_3_UNGATED** eine reine Fan-out-Funktion. Es gibt keinen Entscheidungsmechanismus, keine Priorisierung und keine Datenmodifikation. Ähnliche Splitter mit anderer Ausgangsanzahl (z. B. SPLIT_2) unterscheiden sich lediglich in der Anzahl der Ausgänge.

- `AULI_SPLIT_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

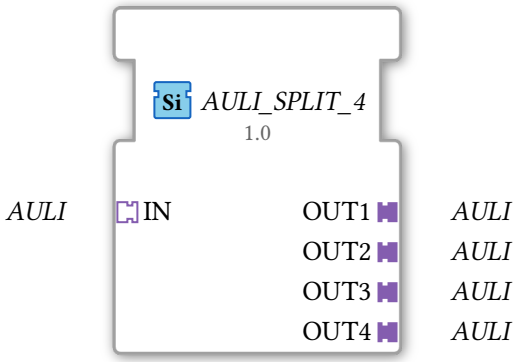
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

AULI_SPLIT_3_UNGATED ist ein einfacher, aber essenzieller Baustein zur Vervielfachung unidirektionaler AULI-Signale. Durch die generische Auslegung und die verzögerungsfreie Weiterleitung eignet er sich besonders für Broadcast-Szenarien in der Automatisierungstechnik, wo ein Signal gleichzeitig an mehrere Verbraucher übergeben werden muss.

AULI_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AULI_SPLIT_4** dient dazu, einen eingehenden **AULI**-Adapter in vier separate, identische Ausgänge aufzuteilen. Er ist als generischer Baustein (Generic FB) ausgelegt und wird unter der Eclipse Public License 2.0 vertrieben. Entwickelt wurde er für die Verwendung in der Automatisierungstechnik, insbesondere im Umfeld der HR Agrartechnik GmbH.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AULI`	`IN`	Socket	Eingangsadapter für das zu verteilende AULI-Signal
`adapter::types::unidirectional::AULI`	`OUT1`	Plug	Erster Ausgangsadapter (identisch zum Eingang)
`adapter::types::unidirectional::AULI`	`OUT2`	Plug	Zweiter Ausgangsadapter
`adapter::types::unidirectional::AULI`	`OUT3`	Plug	Dritter Ausgangsadapter
`adapter::types::unidirectional::AULI`	`OUT4`	Plug	Vierter Ausgangsadapter

Funktionsweise

Der Baustein arbeitet als passive Verteiler-Struktur. Der an `IN` anliegende AULI-Adapter wird intern auf die vier Ausgänge `OUT1` bis `OUT4` durchgeschaltet. Da keine aktive Logik, Verarbeitung oder Zwischenspeicherung stattfindet, werden die am Eingang anliegenden Daten (z. B. Signale, Werte, Zustände) unverändert an alle vier Ausgänge weitergegeben. Die Verbindung ist unidirektional – Änderungen an den Ausgängen wirken nicht auf den Eingang zurück.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als Generic FB mit dem generischen Klassennamen `GEN_AULI_SPLIT` deklariert. Das ermöglicht eine flexible Anpassung und Wiederverwendung in verschiedenen Kontexten (z. B. durch Parametrisierung bei der Instanziierung).
- **Unidirektionale Adapter**: Alle verwendeten Adapter sind vom Typ `adapter::types::unidirectional::AULI`, d. h. sie unterstützen nur einen Datenfluss in eine Richtung.
- **Lizenz**: Der Baustein ist unter der Eclipse Public License 2.0 lizenziert und darf entsprechend den Bedingungen dieser Lizenz genutzt und modifiziert werden.
- **Keine eigenen Ereignisse oder Daten**: Die gesamte Funktionalität wird ausschließlich über Adapter realisiert. Dies reduziert den Overhead und vereinfacht die Integration in bestehende 4diac-Architekturen.

Zustandsübersicht

Der Baustein besitzt keinen eigenen Zustandsautomaten (keine ECC-Zustände). Er verhält sich stateless: Die Ausgänge spiegeln stets den aktuellen Zustand des Eingangs wider. Eine Initialisierung oder besondere Zustandsübergänge sind nicht erforderlich.

Anwendungsszenarien

- **Signalverteilung**: Ein AULI-basiertes Signal (z. B. Steuerbefehl, Messwert) soll an mehrere Verbraucher oder nachgeschaltete Funktionsblöcke weitergeleitet werden.
- **Redundanz oder Parallelverarbeitung**: Mehrere unabhängige Instanzen einer Komponente benötigen denselben Eingangswert – der Split ermöglicht eine einfache Aufteilung ohne zusätzliche Logik.
- **Test- und Simulationsumgebungen**: In Testaufbauten kann ein AULI-Signal an mehrere Beobachter oder Protokollierer verteilt werden.

Vergleich mit ähnlichen Bausteinen

Der **AULI_SPLIT_4** ist ein spezialisierter Split-Baustein ausschließlich für den unidirektionalen AULI-Adaptertyp. Im Gegensatz zu allgemeinen Split-Bausteinen (z. B. `SPLIT` für verschiedene Datentypen) ist er auf genau eine Adapter-Schnittstelle festgelegt, was die Typklarheit erhöht und Fehlkonfigurationen vermeidet. Er unterscheidet sich von aktiven Verteilern (z. B. `MUX` oder `DEMUX`) durch seine passive, verlustfreie Verteilung ohne Umschaltlogik oder Adressierung.

- `AULI_SPLIT_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der **AULI_SPLIT_4** ist ein einfacher, aber nützlicher Funktionsblock zur Vervielfachung eines AULI-Adapters auf vier Ausgänge. Durch die generische Auslegung und den Verzicht auf überflüssige Logik fügt er sich nahtlos in modulare 4diac-Projekte ein. Er eignet sich besonders für Anwendungen, in denen ein Signal an mehrere Empfänger weitergegeben werden muss, ohne dass eine Weiterverarbeitung oder Selektion gewünscht ist.

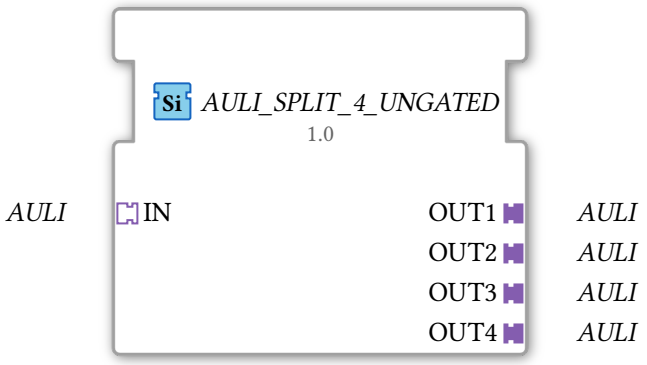
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AULI_SPLIT_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AULI_SPLIT_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die

sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AULI_SPLIT_4_UNGATED** dient dazu, einen eingehenden **AULI**-Adapter in vier separate, identische Ausgänge aufzuteilen. Er ist als generischer Baustein (Generic FB) ausgelegt und wird unter der Eclipse Public License 2.0 vertrieben. Entwickelt wurde er für die Verwendung in der Automatisierungstechnik, insbesondere im Umfeld der HR Agrartechnik GmbH.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AULI`	`IN`	Socket	Eingangsadapter für das zu verteilende AULI-Signal
`adapter::types::unidirectional::AULI`	`OUT1`	Plug	Erster Ausgangsadapter (identisch zum Eingang)
`adapter::types::unidirectional::AULI`	`OUT2`	Plug	Zweiter Ausgangsadapter
`adapter::types::unidirectional::AULI`	`OUT3`	Plug	Dritter Ausgangsadapter
`adapter::types::unidirectional::AULI`	`OUT4`	Plug	Vierter Ausgangsadapter

Funktionsweise

Der Baustein arbeitet als passive Verteiler-Struktur. Der an `IN` anliegende AULI-Adapter wird intern auf die vier Ausgänge `OUT1` bis `OUT4` durchgeschaltet. Da keine aktive Logik, Verarbeitung oder Zwischenspeicherung stattfindet, werden die am Eingang anliegenden Daten (z. B. Signale, Werte, Zustände) unverändert an alle vier Ausgänge weitergegeben. Die Verbindung ist unidirektional – Änderungen an den Ausgängen wirken nicht auf den Eingang zurück.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als Generic FB mit dem generischen Klassennamen `GEN_AULI_SPLIT` deklariert. Das ermöglicht eine flexible Anpassung und Wiederverwendung in verschiedenen Kontexten (z. B. durch Parametrisierung bei der Instanziierung).
- **Unidirektionale Adapter**: Alle verwendeten Adapter sind vom Typ `adapter::types::unidirectional::AULI`, d. h. sie unterstützen nur einen Datenfluss in eine Richtung.
- **Lizenz**: Der Baustein ist unter der Eclipse Public License 2.0 lizenziert und darf entsprechend den Bedingungen dieser Lizenz genutzt und modifiziert werden.
- **Keine eigenen Ereignisse oder Daten**: Die gesamte Funktionalität wird ausschließlich über Adapter realisiert. Dies reduziert den Overhead und vereinfacht die Integration in bestehende 4diac-Architekturen.

Zustandsübersicht

Der Baustein besitzt keinen eigenen Zustandsautomaten (keine ECC-Zustände). Er verhält sich stateless: Die Ausgänge spiegeln stets den aktuellen Zustand des Eingangs wider. Eine Initialisierung oder besondere Zustandsübergänge sind nicht erforderlich.

Anwendungsszenarien

- **Signalverteilung**: Ein AULI-basiertes Signal (z. B. Steuerbefehl, Messwert) soll an mehrere Verbraucher oder nachgeschaltete Funktionsblöcke weitergeleitet werden.
- **Redundanz oder Parallelverarbeitung**: Mehrere unabhängige Instanzen einer Komponente benötigen denselben Eingangswert – der Split ermöglicht eine einfache Aufteilung ohne zusätzliche Logik.
- **Test- und Simulationsumgebungen**: In Testaufbauten kann ein AULI-Signal an mehrere Beobachter oder Protokollierer verteilt werden.

Vergleich mit ähnlichen Bausteinen

Der **AULI_SPLIT_4_UNGATED** ist ein spezialisierter Split-Baustein ausschließlich für den unidirektionalen AULI-Adaptertyp. Im Gegensatz zu allgemeinen Split-Bausteinen (z. B. `SPLIT` für verschiedene Datentypen) ist er auf genau eine Adapter-Schnittstelle festgelegt, was die Typklarheit erhöht und Fehlkonfigurationen vermeidet. Er unterscheidet sich von aktiven Verteilern (z. B. `MUX` oder `DEMUX`) durch seine passive, verlustfreie Verteilung ohne Umschaltlogik oder Adressierung.

- `AULI_SPLIT_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

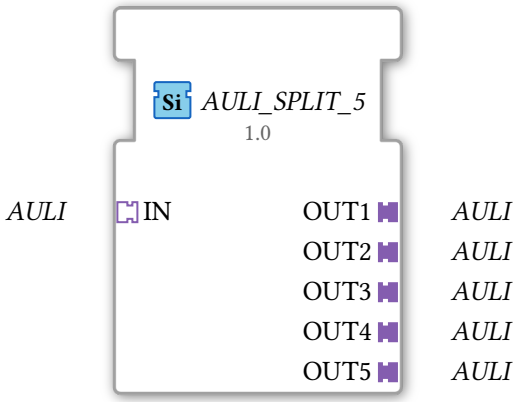
Fazit

Der **AULI_SPLIT_4_UNGATED** ist ein einfacher, aber nützlicher Funktionsblock zur Vervielfachung eines AULI-Adapters auf vier Ausgänge. Durch die generische Auslegung und den Verzicht auf überflüssige Logik fügt er sich nahtlos in modulare 4diac-Projekte ein. Er eignet sich besonders für Anwendungen, in denen ein Signal an mehrere Empfänger weitergegeben werden muss, ohne dass eine Weiterverarbeitung oder Selektion gewünscht ist.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AULI_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AULI_SPLIT_5** dient als generischer Splitter für den unidirektionalen AULI-Adaptertyp. Er erzeugt aus einem eingehenden AULI-Signal fünf identische Ausgangssignale, die an separate Verbraucher weitergegeben werden können. Der Baustein ist als generischer FB realisiert und wird durch die Attribute `GenericClassName` und `TypeHash` für das Eclipse 4diac Core Framework parametrisiert.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	`IN`	`adapter::types::unidirectional::AULI`	Eingehender AULI-Adapter (Quelle)
Plug	`OUT1`	`adapter::types::unidirectional::AULI`	Erster ausgehender AULI-Adapter
Plug	`OUT2`	`adapter::types::unidirectional::AULI`	Zweiter ausgehender AULI-Adapter
Plug	`OUT3`	`adapter::types::unidirectional::AULI`	Dritter ausgehender AULI-Adapter
Plug	`OUT4`	`adapter::types::unidirectional::AULI`	Vierter ausgehender AULI-Adapter
Plug	`OUT5`	`adapter::types::unidirectional::AULI`	Fünfter ausgehender AULI-Adapter

Das Interface besteht ausschließlich aus einem Socket und fünf Plugs desselben Typs. Alle Adapter sind unidirektional ausgelegt.

Funktionsweise

Der FB leitet das am Socket `IN` ankommende AULI-Signal ohne Verzögerung oder Bearbeitung an alle fünf Plugs (`OUT1` … `OUT5`) weiter. Es findet eine reine **1-zu-5-Vervielfachung** statt. Der Baustein besitzt keine Logik, keine Zustände und keinen internen Puffer. Die Weitergabe erfolgt passiv: Sobald der Quell-Adapter Daten sendet, werden diese an alle Ausgänge repliziert.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ deklariert (`GenericClassName = 'GEN_AULI_SPLIT'`). Der konkret verwendete Typ wird erst zur Laufzeit durch das Eclipse 4diac Core Framework aufgelöst.
- **Unidirektionale Adapter**: Sämtliche Adapter verwenden den Typ `adapter::types::unidirectional::AULI`, der nur eine Datenflussrichtung von der Quelle zu den Senken erlaubt.
- **Keine Bedingungen oder Filter**: Das Eingangssignal wird ohne Qualitätsprüfung, Umwandlung oder Synchronisation auf alle Ausgänge verteilt.
- **Kompatibilität**: Der FB ist für den Einsatz in IEC 61499-2 konformen Systemen ausgelegt und erfordert das Importieren der Pakete `eclipse4diac::core::GenericClassName` und `eclipse4diac::core::TypeHash`.

Zustandsübersicht

Der FB besitzt **keinen internen Zustandsautomaten**. Es existieren keine Zustände, keine Transitionen und keine Aktionen. Die Funktionalität beschränkt sich auf die passive Signalverteilung.

Anwendungsszenarien

- **Signalverteilung**: Ein Sensorwert oder ein Steuersignal (z. B. ein AULI-basierter Datensfrom) muss an mehrere parallele Verarbeitungseinheiten oder Aktoren gesendet werden.
- **Testumgebungen**: In Simulations- oder Testsystemen kann das gleiche Eingangssignal gleichzeitig an mehrere Analyseblöcke oder Logfunktionen angelegt werden.
- **Redundante Ausgaben**: Wenn mehrere Subsysteme denselben Wert benötigen, jedoch keine kaskadierte Verkettung gewünscht ist.

Vergleich mit ähnlichen Bausteinen

Baustein	Typ	Ausgänge	Besonderheit
`AULI_SPLIT_5`	generischer Splitter	5	spezifisch für AULI-Adapter
`SPLIT_2` (nicht standardisiert)	generischer Splitter	2	meist für Datentypen wie INT oder BOOL
`F_SPLIT` (aus IEC 61499-Bibliotheken)	funktionaler Splitter	variabel	basiert auf ECC und Ereignissen

Der `AULI_SPLIT_5` ist im Gegensatz zu ereignisbasierten Splittern **ereignislos** und rein adapterbasiert. Er lässt sich daher besonders in Datenflusssystemen ohne Taktsignal einsetzen.

- **`AULI_SPLIT_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

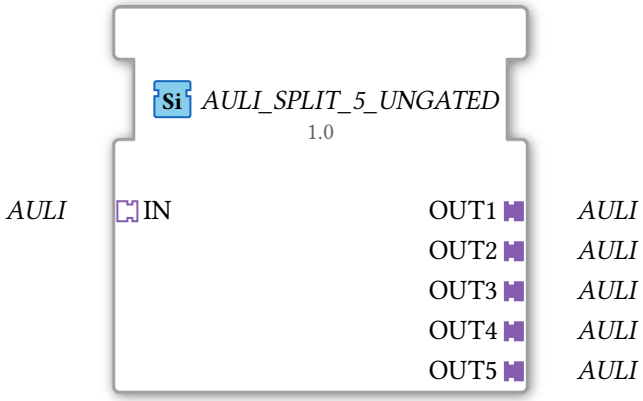
Der `AULI_SPLIT_5` ist ein minimalistischer, aber nützlicher Funktionsblock zur Vervielfachung eines unidirektionalen AULI-Signals auf fünf Ausgänge. Seine generische Natur erlaubt eine flexible Parametrisierung im Eclipse-4diac-Framework. Aufgrund des fehlenden Zustandsautomaten und der reinen Durchleitung ist er extrem ressourcenschonend und für Hochgeschwindigkeits-Datenverteilungen geeignet.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AULI_SPLIT_5_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `AULI_SPLIT_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AULI_SPLIT_5_UNGATED** dient als generischer Splitter für den unidirektionalen AULI-Adaptertyp. Er erzeugt aus einem eingehenden AULI-Signal fünf identische Ausgangssignale, die an separate Verbraucher weitergegeben werden können. Der Baustein ist als generischer FB realisiert und wird durch die Attribute `GenericClassName` und `TypeHash` für das Eclipse 4diac Core Framework parametrisiert.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	`IN`	`adapter::types::unidirectional::AULI`	Eingehender AULI-Adapter (Quelle)
Plug	`OUT1`	`adapter::types::unidirectional::AULI`	Erster ausgehender AULI-Adapter
Plug	`OUT2`	`adapter::types::unidirectional::AULI`	Zweiter ausgehender AULI-Adapter
Plug	`OUT3`	`adapter::types::unidirectional::AULI`	Dritter ausgehender AULI-Adapter
Plug	`OUT4`	`adapter::types::unidirectional::AULI`	Vierter ausgehender AULI-Adapter

Plug	`OUT5`	`adapter::types::unidirectional::AULI`	Fünfter ausgehender AULI-Adapter

Das Interface besteht ausschließlich aus einem Socket und fünf Plugs desselben Typs. Alle Adapter sind unidirektional ausgelegt.

Funktionsweise

Der FB leitet das am Socket `IN` ankommende AULI-Signal ohne Verzögerung oder Bearbeitung an alle fünf Plugs (`OUT1` … `OUT5`) weiter. Es findet eine reine **1-zu-5-Vervielfachung** statt. Der Baustein besitzt keine Logik, keine Zustände und keinen internen Puffer. Die Weitergabe erfolgt passiv: Sobald der Quell-Adapter Daten sendet, werden diese an alle Ausgänge repliziert.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ deklariert (`GenericClassName = 'GEN_AULI_SPLIT'`). Der konkret verwendete Typ wird erst zur Laufzeit durch das Eclipse 4diac Core Framework aufgelöst.
- **Unidirektionale Adapter**: Sämtliche Adapter verwenden den Typ `adapter::types::unidirectional::AULI`, der nur eine Datenflussrichtung von der Quelle zu den Senken erlaubt.
- **Keine Bedingungen oder Filter**: Das Eingangssignal wird ohne Qualitätsprüfung, Umwandlung oder Synchronisation auf alle Ausgänge verteilt.
- **Kompatibilität**: Der FB ist für den Einsatz in IEC 61499-2 konformen Systemen ausgelegt und erfordert das Importieren der Pakete `eclipse4diac::core::GenericClassName` und `eclipse4diac::core::TypeHash`.

Zustandsübersicht

Der FB besitzt **keinen internen Zustandsautomaten**. Es existieren keine Zustände, keine Transitionen und keine Aktionen. Die Funktionalität beschränkt sich auf die passive Signalverteilung.

Anwendungsszenarien

- **Signalverteilung**: Ein Sensorwert oder ein Steuersignal (z. B. ein AULI-basierter Datensfrom) muss an mehrere parallele Verarbeitungseinheiten oder Aktoren gesendet werden.
- **Testumgebungen**: In Simulations- oder Testsystemen kann das gleiche Eingangssignal gleichzeitig an mehrere Analyseblöcke oder Logfunktionen angelegt werden.
- **Redundante Ausgaben**: Wenn mehrere Subsysteme denselben Wert benötigen, jedoch keine kaskadierte Verkettung gewünscht ist.

Vergleich mit ähnlichen Bausteinen

Baustein	Typ	Ausgänge	Besonderheit
`AULI_SPLIT_5_UNGATED`	generischer Splitter	5	spezifisch für AULI-Adapter
`SPLIT_2` (nicht standardisiert)	generischer Splitter	2	meist für Datentypen wie INT oder BOOL
`F_SPLIT` (aus IEC 61499-Bibliotheken)	funktionaler Splitter	variabel	basiert auf ECC und Ereignissen

Der `AULI_SPLIT_5_UNGATED` ist im Gegensatz zu ereignisbasierten Splittern **ereignislos** und rein adapterbasiert. Er lässt sich daher besonders in Datenflusssystemen ohne Taktsignal einsetzen.

- **`AULI_SPLIT_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

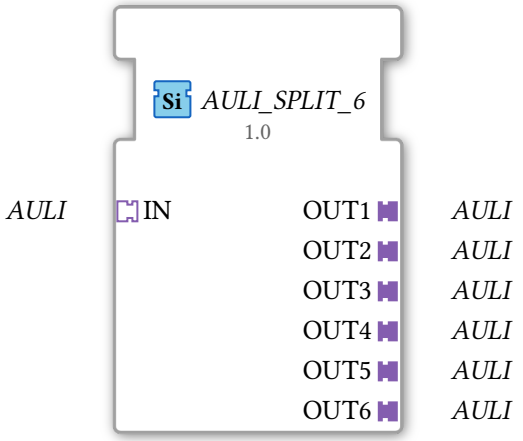
Der `AULI_SPLIT_5_UNGATED` ist ein minimalistischer, aber nützlicher Funktionsblock zur Vervielfachung eines unidirektionalen AULI-Signals auf fünf Ausgänge. Seine generische Natur erlaubt eine flexible Parametrisierung im Eclipse-4diac-Framework. Aufgrund des fehlenden Zustandsautomaten und der

reinen Durchleitung ist er extrem ressourcenschonend und für Hochgeschwindigkeits-Datenverteilungen geeignet.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AULI_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AULI_SPLIT_6** dient dazu, einen einzelnen AULI-Adapter-Eingang (unidirektional) auf sechs identische AULI-Ausgänge zu verteilen. Er stellt eine generische Lösung dar, um ein AULI-Signal an mehrere nachfolgende Bausteine weiterzuleiten, ohne die Signalintegrität zu beeinträchtigen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Adapter	Richtung	Typ	Beschreibung
IN	Socket (Eingang)	adapter::types::unidirectional::AULI	Empfängt das zu verteilende AULI-Signal.
OUT1	Plug (Ausgang)	adapter::types::unidirectional::AULI	Erster identischer Ausgang des verteilten Signals.
OUT2	Plug (Ausgang)	adapter::types::unidirectional::AULI	Zweiter identischer Ausgang.
OUT3	Plug (Ausgang)	adapter::types::unidirectional::AULI	Dritter identischer Ausgang.

OUT4	Plug (Ausgang)	adapter::types::unidirectional::AULI	Vierter identischer Ausgang.
OUT5	Plug (Ausgang)	adapter::types::unidirectional::AULI	Fünfter identischer Ausgang.
OUT6	Plug (Ausgang)	adapter::types::unidirectional::AULI	Sechster identischer Ausgang.

Funktionsweise

Der Baustein arbeitet als reine Signalverteilung (Splitter). Ein an **IN** anliegendes AULI-Signal wird ohne Verzögerung oder Datenverlust auf alle sechs Ausgangs-Adapter (**OUT1** bis **OUT6**) dupliziert. Es findet keine aktive Verarbeitung oder Filterung statt – die Datenübertragung erfolgt ausschließlich über die unidirektionalen AULI-Adapter.

Da keine Ereignisse oder Daten-Ein-/Ausgänge vorhanden sind, basiert die Kommunikation vollständig auf der Adapter-Schnittstelle. Der Baustein ist ereignisfrei und wird allein durch die Verbindung der Adapter aktiv.

Technische Besonderheiten

- **Generischer Typ:** Der Baustein ist als generischer Funktionsbaustein (`GEN_AULI_SPLIT`) implementiert, sodass er für verschiedene AULI-Varianten wiederverwendet werden kann.
- **Unidirektionale Adapter:** Die verwendeten Adapter sind als unidirektionale Schnittstellen deklariert – sie unterstützen nur eine Datenflussrichtung (vom Socket zu den Plugs).
- **Keine Daten- oder Ereignisverarbeitung:** Der Baustein enthält keine internen Algorithmen oder Zustände; er fungiert ausschließlich als passiver Splitter.
- **Einfache Skalierung:** Durch die feste Anzahl von sechs Ausgängen ist der Baustein für typische Anwendungen in der Landtechnik (z. B. HR Agrartechnik) optimiert.

Zustandsübersicht

Da der Baustein keine Ereignissteuerung besitzt und keine internen Zustände verwaltet, existiert kein Zustandsautomat. Der Baustein ist rein kombinatorisch und gibt die Eingangssignale unverändert an alle Ausgänge weiter.

Anwendungsszenarien

- **Verteilen eines Sensorsignals:** In landwirtschaftlichen Steuerungssystemen kann ein einzelner AULI-Sensor (z. B. für Erntegutfluss) an mehrere nachgelagerte Auswertemodule angeschlossen werden.
- **Parallelisierung von Aktorsteuerungen:** Ein Steuerbefehl (z. B. für Hydraulikventile) soll gleichzeitig an sechs identische Aktoren weitergeleitet werden.
- **Redundanzaufbau:** Bei sicherheitskritischen Anwendungen kann das gleiche Signal mehrfach an redundante Steuereinheiten geführt werden.

Vergleich mit ähnlichen Bausteinen

Baustein	Anzahl Ausgänge	Besonderheit
AULI_SPLIT_6	6	Fester Splitter für AULI-Adapter
AULI_SPLIT_2	2	Kleinere Variante für zwei Ausgänge
AULI_SPLIT_4	4	Mittlere Variante für vier Ausgänge

Der **AULI_SPLIT_6** bietet im Vergleich zu kleineren Splittern eine höhere Anzahl an Ausgängen, was den Verdrahtungsaufwand in komplexen Systemen reduziert und gleichzeitig eine saubere Signalstruktur erhält.

- `AULI_SPLIT_6_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs

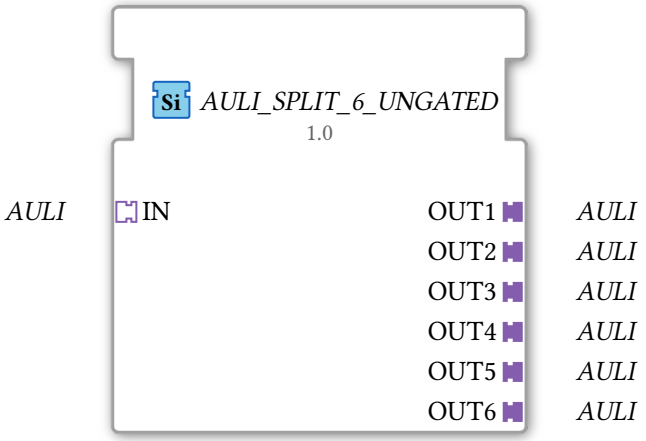
unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der **AULI_SPLIT_6** ist ein einfacher, aber effektiver Funktionsbaustein zur Verteilung eines AULI-Adaptersignals auf sechs parallele Ausgänge. Dank seiner generischen Implementierung und der reinen Adapter-Schnittstelle lässt er sich leicht in IEC-61499-Anwendungen der Landtechnik und Automatisierungstechnik integrieren. Er eignet sich besonders für Szenarien, in denen ein Signal mehrfach benötigt wird, ohne zusätzliche Logik oder Zustandsverwaltung.

AULI_SPLIT_6_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AULI_SPLIT_6`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AULI_SPLIT_6_UNGATED** dient dazu, einen einzelnen AULI-Adapter-Eingang (unidirektional) auf sechs identische AULI-Ausgänge zu verteilen. Er stellt eine generische Lösung dar, um ein AULI-Signal an mehrere nachfolgende Bausteine weiterzuleiten, ohne die Signalintegrität zu beeinträchtigen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Adapter	Richtung	Typ	Beschreibung
IN	Socket (Eingang)	adapter::types::unidirectional::AULI	Empfängt das zu verteilende AULI-Signal.
OUT1	Plug (Ausgang)	adapter::types::unidirectional::AULI	Erster identischer Ausgang des verteilten Signals.
OUT2	Plug (Ausgang)	adapter::types::unidirectional::AULI	Zweiter identischer Ausgang.
OUT3	Plug (Ausgang)	adapter::types::unidirectional::AULI	Dritter identischer Ausgang.
OUT4	Plug (Ausgang)	adapter::types::unidirectional::AULI	Vierter identischer Ausgang.
OUT5	Plug (Ausgang)	adapter::types::unidirectional::AULI	Fünfter identischer Ausgang.
OUT6	Plug (Ausgang)	adapter::types::unidirectional::AULI	Sechster identischer Ausgang.

Funktionsweise

Der Baustein arbeitet als reine Signalverteilung (Splitter). Ein an **IN** anliegendes AULI-Signal wird ohne Verzögerung oder Datenverlust auf alle sechs Ausgangs-Adapter (**OUT1** bis **OUT6**) dupliziert. Es findet keine aktive Verarbeitung oder Filterung statt – die Datenübertragung erfolgt ausschließlich über die unidirektionalen AULI-Adapter.

Da keine Ereignisse oder Daten-Ein-/Ausgänge vorhanden sind, basiert die Kommunikation vollständig auf der Adapter-Schnittstelle. Der Baustein ist ereignisfrei und wird allein durch die Verbindung der Adapter aktiv.

Technische Besonderheiten

- **Generischer Typ:** Der Baustein ist als generischer Funktionsbaustein (`GEN_AULI_SPLIT`) implementiert, sodass er für verschiedene AULI-Varianten wiederverwendet werden kann.
- **Unidirektionale Adapter:** Die verwendeten Adapter sind als unidirektionale Schnittstellen deklariert – sie unterstützen nur eine Datenflussrichtung (vom Socket zu den Plugs).
- **Keine Daten- oder Ereignisverarbeitung:** Der Baustein enthält keine internen Algorithmen oder Zustände; er fungiert ausschließlich als passiver Splitter.
- **Einfache Skalierung:** Durch die feste Anzahl von sechs Ausgängen ist der Baustein für typische Anwendungen in der Landtechnik (z. B. HR Agrartechnik) optimiert.

Zustandsübersicht

Da der Baustein keine Ereignissteuerung besitzt und keine internen Zustände verwaltet, existiert kein Zustandsautomat. Der Baustein ist rein kombinatorisch und gibt die Eingangssignale unverändert an alle Ausgänge weiter.

Anwendungsszenarien

- **Verteilen eines Sensorsignals:** In landwirtschaftlichen Steuerungssystemen kann ein einzelner AULI-Sensor (z. B. für Erntegutfluss) an mehrere nachgelagerte Auswertemodule angeschlossen werden.
- **Parallelisierung von Aktorsteuerungen:** Ein Steuerbefehl (z. B. für Hydraulikventile) soll gleichzeitig an sechs identische Aktoren weitergeleitet werden.
- **Redundanzaufbau:** Bei sicherheitskritischen Anwendungen kann das gleiche Signal mehrfach an redundante Steuereinheiten geführt werden.

Vergleich mit ähnlichen Bausteinen

Baustein	Anzahl Ausgänge	Besonderheit
AULI_SPLIT_6_UNGATED	6	Fester Splitter für AULI-Adapter

AULI_SPLIT_2	2	Kleinere Variante für zwei Ausgänge
AULI_SPLIT_4	4	Mittlere Variante für vier Ausgänge

Der **AULI_SPLIT_6_UNGATED** bietet im Vergleich zu kleineren Splittern eine höhere Anzahl an Ausgängen, was den Verdrahtungsaufwand in komplexen Systemen reduziert und gleichzeitig eine saubere Signalstruktur erhält.

- `AULI_SPLIT_6`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

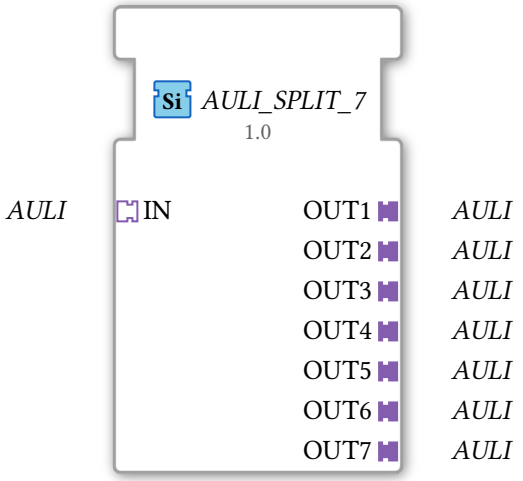
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AULI_SPLIT_6_UNGATED** ist ein einfacher, aber effektiver Funktionsbaustein zur Verteilung eines AULI-Adaptersignals auf sechs parallele Ausgänge. Dank seiner generischen Implementierung und der reinen Adapter-Schnittstelle lässt er sich leicht in IEC-61499-Anwendungen der Landtechnik und Automatisierungstechnik integrieren. Er eignet sich besonders für Szenarien, in denen ein Signal mehrfach benötigt wird, ohne zusätzliche Logik oder Zustandsverwaltung.

AULI_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AULI_SPLIT_7** dient als generischer Splitter, der ein eingehendes AULI-Adapter-Signal (unidirektional) auf sieben separate AULI-Ausgänge verteilt. Er ist speziell für die Aufteilung eines Befehls- oder Datenstroms (AULI-Protokoll) ausgelegt und ermöglicht die gleichzeitige Versorgung mehrerer nachgeschalteter Bausteine mit dem gleichen Signal.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	IN	adapter::types::unidirectional::AULI	Empfängt das aufzuteilende AULI-Signal
Plug (Ausgang)	OUT1	adapter::types::unidirectional::AULI	Erster Ausgang für das AULI-Signal
Plug (Ausgang)	OUT2	adapter::types::unidirectional::AULI	Zweiter Ausgang
Plug (Ausgang)	OUT3	adapter::types::unidirectional::AULI	Dritter Ausgang
Plug (Ausgang)	OUT4	adapter::types::unidirectional::AULI	Vierter Ausgang
Plug (Ausgang)	OUT5	adapter::types::unidirectional::AULI	Fünfter Ausgang
Plug (Ausgang)	OUT6	adapter::types::unidirectional::AULI	Sechster Ausgang
Plug (Ausgang)	OUT7	adapter::types::unidirectional::AULI	Siebter Ausgang

Funktionsweise

Der Baustein leitet das am Socket **IN** anliegende AULI-Signal unverändert an alle sieben Plugs **OUT1** bis **OUT7** weiter. Es findet keine Signalverarbeitung, Filterung oder Verzögerung statt. Jeder Ausgang empfängt simultan dieselbe Signalfolge wie der Eingang. Die Aufteilung erfolgt rein leitungs- bzw. signaltechnisch innerhalb der 4diac-IDE; es sind keine Berechnungen oder Zustandsänderungen erforderlich.

Technische Besonderheiten

- **Generischer Baustein:** Der FB ist als generischer Typ (`GEN_AULI_SPLIT`) deklariert und verwendet das Attribut `eclipse4diac::core::GenericClassName` für die typsichere Adapterzuordnung.
- **Unidirektionaler Adapter:** Sowohl Eingang als auch Ausgänge basieren auf dem Adapter-Typ `adapter::types::unidirectional::AULI`, der nur eine Datenflussrichtung unterstützt.
- **Keine Ereignissteuerung:** Der FB benötigt keine Trigger-Ereignisse; die Signalweitergabe erfolgt automatisch bei Änderung des Eingangssignals.
- **Keine internen Zustände:** Es gibt keinen Zustandsautomaten oder Speicherverhalten – der Baustein ist rein kombinatorisch.

Zustandsübersicht

Da keine Zustandsmaschine (ECC) vorhanden ist, besitzt der Baustein keine internen Zustände. Das Ausgangssignal folgt unmittelbar dem Eingangssignal ohne Verzögerung oder Logik.

Anwendungsszenarien

- **Parallelversorgung mehrerer Aktuatoren**: Ein gemeinsames Steuersignal (z.B. über AULI-Adapter) soll gleichzeitig an mehrere Stellglieder oder Subsysteme verteilt werden.
- **Signal-Multicasting**: Verteilung eines Sensorsignals oder einer Konfigurationsnachricht an mehrere Empfänger in der Automatisierungsanlage.
- **Test- und Simulationsaufbauten**: Aufteilen eines Eingangssignals zur gleichzeitigen Überwachung an verschiedenen Messpunkten.

Vergleich mit ähnlichen Bausteinen

Die 4diac-Bibliothek enthält Splitter-Bausteine für verschiedene Ausgangszahlen, z.B. `AULI_SPLIT_3` oder `AULI_SPLIT_5`. Der `AULI_SPLIT_7` unterscheidet sich lediglich durch die Anzahl der Ausgänge (7). Bausteine zur Aufteilung anderer Adaptertypen (z.B. `BOOL_SPLIT`, `INT_SPLIT`) haben eine ähnliche Logik, arbeiten jedoch mit unterschiedlichen Daten- bzw. Adapterformaten.

- `AULI_SPLIT_7_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

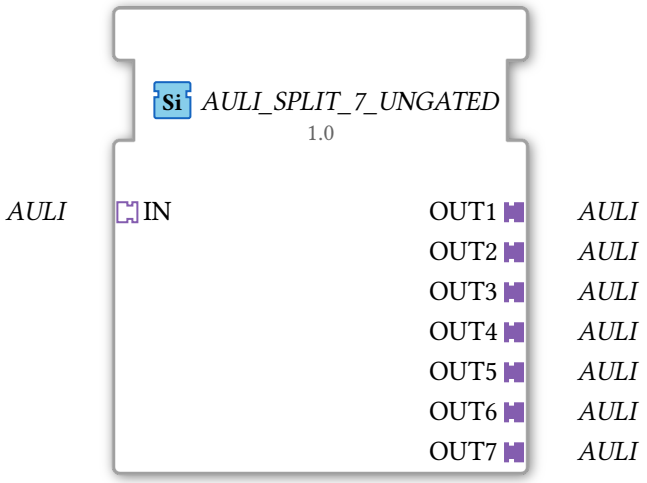
Der `AULI_SPLIT_7` ist ein einfacher, aber essenzieller Baustein zur Signalverteilung innerhalb der AULI-Adapterlandschaft. Er erlaubt die saubere und typsichere Aufteilung eines unidirektionalen Signals auf bis zu sieben Pfade, ohne zusätzliche Logik oder Verzögerungen. Dank seiner generischen Auslegung lässt er sich flexibel in IEC 61499-basierten Automatisierungsprojekten einsetzen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AULI_SPLIT_7_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AULI_SPLIT_7`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AULI_SPLIT_7_UNGATED** dient als generischer Splitter, der ein eingehendes AULI-Adapter-Signal (unidirektional) auf sieben separate AULI-Ausgänge verteilt. Er ist speziell für die Aufteilung eines Befehls- oder Datenstroms (AULI-Protokoll) ausgelegt und ermöglicht die gleichzeitige Versorgung mehrerer nachgeschalteter Bausteine mit dem gleichen Signal.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	IN	adapter::types::unidirectional::AULI	Empfängt das aufzuteilende AULI-Signal
Plug (Ausgang)	OUT1	adapter::types::unidirectional::AULI	Erster Ausgang für das AULI-Signal
Plug (Ausgang)	OUT2	adapter::types::unidirectional::AULI	Zweiter Ausgang
Plug (Ausgang)	OUT3	adapter::types::unidirectional::AULI	Dritter Ausgang
Plug (Ausgang)	OUT4	adapter::types::unidirectional::AULI	Vierter Ausgang
Plug (Ausgang)	OUT5	adapter::types::unidirectional::AULI	Fünfter Ausgang
Plug (Ausgang)	OUT6	adapter::types::unidirectional::AULI	Sechster Ausgang
Plug (Ausgang)	OUT7	adapter::types::unidirectional::AULI	Siebter Ausgang

Funktionsweise

Der Baustein leitet das am Socket **IN** anliegende AULI-Signal unverändert an alle sieben Plugs **OUT1** bis **OUT7** weiter. Es findet keine Signalverarbeitung, Filterung oder Verzögerung statt. Jeder Ausgang empfängt simultan dieselbe Signalfolge wie der Eingang. Die Aufteilung erfolgt rein leitungs- bzw. signaltechnisch innerhalb der 4diac-IDE; es sind keine Berechnungen oder Zustandsänderungen erforderlich.

Technische Besonderheiten

- **Generischer Baustein:** Der FB ist als generischer Typ (`GEN_AULI_SPLIT`) deklariert und verwendet das Attribut `eclipse4diac::core::GenericClassName` für die typsichere Adapterzuordnung.
- **Unidirektionaler Adapter:** Sowohl Eingang als auch Ausgänge basieren auf dem Adapter-Typ `adapter::types::unidirectional::AULI`, der nur eine Datenflussrichtung unterstützt.
- **Keine Ereignissteuerung:** Der FB benötigt keine Trigger-Ereignisse; die Signalweitergabe erfolgt automatisch bei Änderung des Eingangssignals.
- **Keine internen Zustände:** Es gibt keinen Zustandsautomaten oder Speicherverhalten – der Baustein ist rein kombinatorisch.

Zustandsübersicht

Da keine Zustandsmaschine (ECC) vorhanden ist, besitzt der Baustein keine internen Zustände. Das Ausgangssignal folgt unmittelbar dem Eingangssignal ohne Verzögerung oder Logik.

Anwendungsszenarien

- **Parallelversorgung mehrerer Aktuatoren**: Ein gemeinsames Steuersignal (z.B. über AULI-Adapter) soll gleichzeitig an mehrere Stellglieder oder Subsysteme verteilt werden.
- **Signal-Multicasting**: Verteilung eines Sensorsignals oder einer Konfigurationsnachricht an mehrere Empfänger in der Automatisierungsanlage.
- **Test- und Simulationsaufbauten**: Aufteilen eines Eingangssignals zur gleichzeitigen Überwachung an verschiedenen Messpunkten.

Vergleich mit ähnlichen Bausteinen

Die 4diac-Bibliothek enthält Splitter-Bausteine für verschiedene Ausgangszahlen, z.B. `AULI_SPLIT_3` oder `AULI_SPLIT_5`. Der `AULI_SPLIT_7_UNGATED` unterscheidet sich lediglich durch die Anzahl der Ausgänge (7). Bausteine zur Aufteilung anderer Adaptertypen (z.B. `BOOL_SPLIT`, `INT_SPLIT`) haben eine ähnliche Logik, arbeiten jedoch mit unterschiedlichen Daten- bzw. Adapterformaten.

- **`AULI_SPLIT_7`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

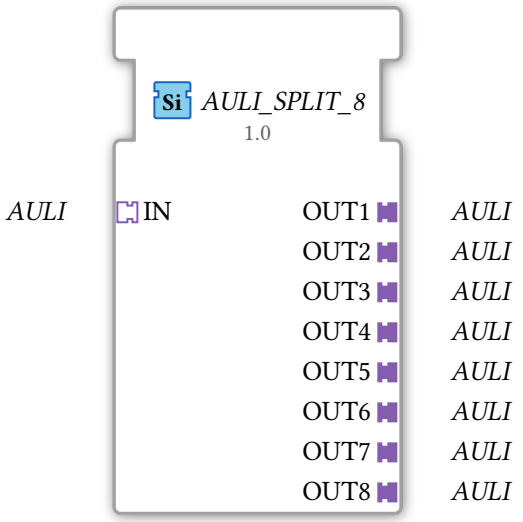
Fazit

Der `AULI_SPLIT_7_UNGATED` ist ein einfacher, aber essenzieller Baustein zur Signalverteilung innerhalb der AULI-Adapterlandschaft. Er erlaubt die saubere und typsichere Aufteilung eines unidirektionalen Signals auf bis zu sieben Pfade, ohne zusätzliche Logik oder Verzögerungen. Dank seiner generischen Auslegung lässt er sich flexibel in IEC 61499-basierten Automatisierungsprojekten einsetzen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AULI_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AULI_SPLIT_8` dient dazu, einen eingehenden unidirektionalen AULI-Adapter (Socket `IN`) auf acht baugleiche AULI-Ausgangsadapter (`OUT1` bis `OUT8`) aufzuteilen. Er arbeitet als reiner Verteiler – das eingehende Signal wird ohne Verzögerung oder Logikänderung an alle Ausgänge weitergegeben. Der Baustein ist generisch ausgelegt und kann mit verschiedenen AULI-Typen verwendet werden (siehe Attribut `GenericClassName`).

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	`IN`	`adapter::types::unidirectional::AULI`	Eingehender AULI-Adapter, der auf die acht Ausgänge verteilt wird.
Plug (Ausgang)	`OUT1` – `OUT8`	`adapter::types::unidirectional::AULI`	Acht parallele Ausgänge, die das identische Signal des Eingangs weiterleiten.

Funktionsweise

Der Baustein verfügt über keine eigene Ablaufsteuerung (kein ECC) und keine datenverarbeitenden Algorithmen. Er verbindet den eingehenden Adapter `IN` direkt mit allen acht Ausgangsadaptern `OUT1` bis `OUT8`. Jeder Datenaustausch, der über den Adapter `IN` stattfindet (z. B. Ereignisse, Werteänderungen), wird ohne Änderung an sämtliche Ausgangsadapter weitergegeben. Die Verteilung erfolgt kombinatorisch – es gibt keine Zwischenspeicherung oder zeitliche Staffelung.

Technische Besonderheiten

- **Generischer Aufbau:** Der Baustein ist als generischer FB mit `eclipse4diac::core::GenericClassName = 'GEN_AULI_SPLIT'` deklariert. Dadurch kann er für verschiedene AULI-Adaptertypen (z. B. mit unterschiedlichen Datentypen) parametrisiert werden.
- **Reine Adapter-Verbindung:** Es werden weder Ereignisein-/ausgänge noch Datenelemente auf der FB-Ebene benötigt – die gesamte Kommunikation erfolgt über die AULI-Adapter.
- **Keine Zustandslogik:** Der FB besitzt kein ECC; das Verhalten ist vollständig durch die Adapterverdrahtung bestimmt.

Zustandsübersicht

Der Funktionsbaustein enthält keinen internen Zustandsautomaten. Es gibt keine unterscheidbaren Betriebsmodi oder Zustandsänderungen – das Verhalten ist rein kombinatorisch.

Anwendungsszenarien

- **Signalverteilung:** Ein AULI-Sensor (z. B. Temperatur, Druck) soll sein Signal an mehrere unabhängige Verbraucher (Steuerungen, Anzeigen, Datenlogger) gleichzeitig weiterleiten.
- **Redundanz / Broadcasting:** Ein Befehl oder eine Konfiguration wird von einer zentralen Instanz an bis zu acht Aktoren oder Subsysteme verteilt.
- **Testumgebungen:** Ein simulierter AULI-Wert wird parallel an mehrere FB-Instanzen zur Validierung gesendet.

Vergleich mit ähnlichen Bausteinen

In der 4diac-IDE existieren häufig Bausteine wie `SPLIT_2`, `SPLIT_4` oder generische Split-Bausteine für Ereignis-/Datenleitungen. `AULI_SPLIT_8` spezialisiert dies auf den AULI-Adapter und bietet eine kompakte 1:8-Verteilung ohne zusätzliche Datentyp-Konvertierung. Gegenüber einer manuellen Verkettung von einfacheren Split-Bausteinen reduziert er die Verdrahtungskomplexität und erhöht die Übersichtlichkeit.

- `AULI_SPLIT_8_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der `AULI_SPLIT_8` ist ein einfacher, aber nützlicher Verteilerbaustein für unidirektionale AULI-Adapter. Er ermöglicht eine saubere, generische Aufteilung eines Signals auf bis zu acht Ausgänge und eignet sich besonders für Broadcasting-Szenarien, in denen mehrere Empfänger denselben Adapter-Wert benötigen.

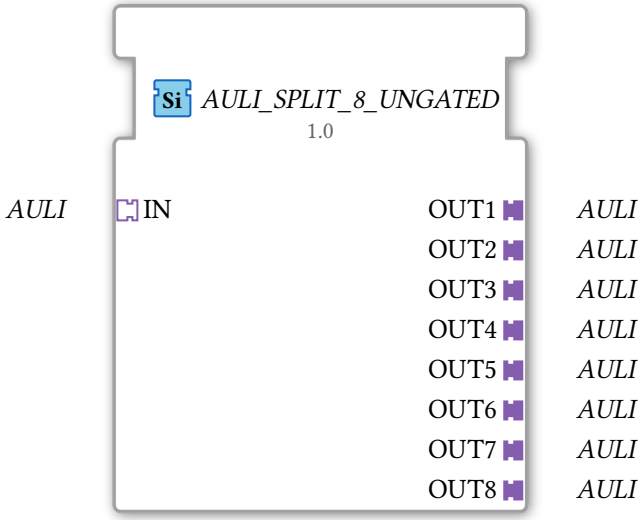
Sein generischer Charakter macht ihn flexibel einsetzbar und erleichtert die Wiederverwendung in verschiedenen Projekten.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AULI_SPLIT_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AULI_SPLIT_8`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AULI_SPLIT_8_UNGATED` dient dazu, einen eingehenden unidirektionalen AULI-Adapter (Socket `IN`) auf acht baugleiche AULI-Ausgangsadapter (`OUT1` bis `OUT8`) aufzuteilen. Er arbeitet als reiner Verteiler – das eingehende Signal wird ohne Verzögerung oder Logikänderung an alle Ausgänge weitergegeben. Der Baustein ist generisch ausgelegt und kann mit verschiedenen AULI-Typen verwendet werden (siehe Attribut `GenericClassName`).

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	`IN`	`adapter::types::unidirectional::AULI`	Eingehender AULI-Adapter, der auf die acht Ausgänge verteilt wird.
Plug (Ausgang)	`OUT1` – `OUT8`	`adapter::types::unidirectional::AULI`	Acht parallele Ausgänge, die das identische Signal des Eingangs weiterleiten.

Funktionsweise

Der Baustein verfügt über keine eigene Ablaufsteuerung (kein ECC) und keine datenverarbeitenden Algorithmen. Er verbindet den eingehenden Adapter `IN` direkt mit allen acht Ausgangsadaptern `OUT1` bis `OUT8`. Jeder Datenaustausch, der über den Adapter `IN` stattfindet (z. B. Ereignisse, Werteänderungen), wird ohne Änderung an sämtliche Ausgangsadapter weitergegeben. Die Verteilung erfolgt kombinatorisch – es gibt keine Zwischenspeicherung oder zeitliche Staffelung.

Technische Besonderheiten

- **Generischer Aufbau:** Der Baustein ist als generischer FB mit `eclipse4diac::core::GenericClassName = 'GEN_AULI_SPLIT'` deklariert. Dadurch kann er für verschiedene AULI-Adaptertypen (z. B. mit unterschiedlichen Datentypen) parametrisiert werden.
- **Reine Adapter-Verbindung:** Es werden weder Ereignisein-/ausgänge noch Datenelemente auf der FB-Ebene benötigt – die gesamte Kommunikation erfolgt über die AULI-Adapter.
- **Keine Zustandslogik:** Der FB besitzt kein ECC; das Verhalten ist vollständig durch die Adapterverdrahtung bestimmt.

Zustandsübersicht

Der Funktionsbaustein enthält keinen internen Zustandsautomaten. Es gibt keine unterscheidbaren Betriebsmodi oder Zustandsänderungen – das Verhalten ist rein kombinatorisch.

Anwendungsszenarien

- **Signalverteilung:** Ein AULI-Sensor (z. B. Temperatur, Druck) soll sein Signal an mehrere unabhängige Verbraucher (Steuerungen, Anzeigen, Datenlogger) gleichzeitig weiterleiten.
- **Redundanz / Broadcasting:** Ein Befehl oder eine Konfiguration wird von einer zentralen Instanz an bis zu acht Aktoren oder Subsysteme verteilt.
- **Testumgebungen:** Ein simulierter AULI-Wert wird parallel an mehrere FB-Instanzen zur Validierung gesendet.

Vergleich mit ähnlichen Bausteinen

In der 4diac-IDE existieren häufig Bausteine wie `SPLIT_2`, `SPLIT_4` oder generische Split-Bausteine für Ereignis-/Datenleitungen. `AULI_SPLIT_8_UNGATED` spezialisiert dies auf den AULI-Adapter und bietet eine kompakte 1:8-Verteilung ohne zusätzliche Datentyp-Konvertierung. Gegenüber einer manuellen Verkettung von einfacheren Split-Bausteinen reduziert er die Verdrahtungskomplexität und erhöht die Übersichtlichkeit.

- `AULI_SPLIT_8`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

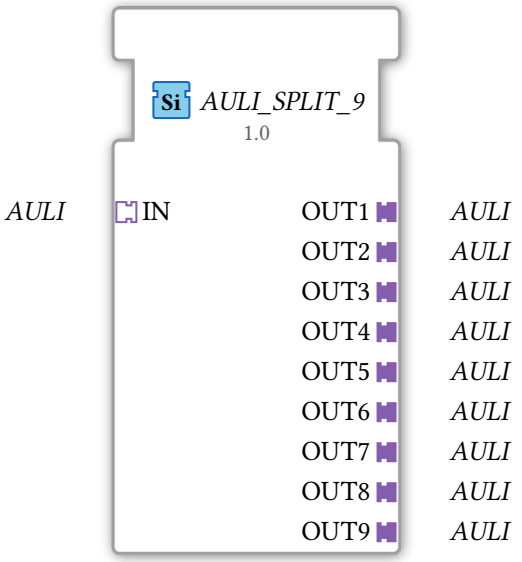
Fazit

Der `AULI_SPLIT_8_UNGATED` ist ein einfacher, aber nützlicher Verteilerbaustein für unidirektionale AULI-Adapter. Er ermöglicht eine saubere, generische Aufteilung eines Signals auf bis zu acht Ausgänge und eignet sich besonders für Broadcasting-Szenarien, in denen mehrere Empfänger denselben Adapter-Wert benötigen. Sein generischer Charakter macht ihn flexibel einsetzbar und erleichtert die Wiederverwendung in verschiedenen Projekten.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AULI_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AULI_SPLIT_9** dient als Verteiler für den unidirektionalen AULI-Adapter. Er nimmt einen eingehenden AULI-Datensatz über den Socket **IN** entgegen und leitet diesen unverändert an alle neun Ausgangs-Adapter (**OUT1** bis **OUT9**) weiter. Der Baustein ist als generischer Splitter konzipiert und eignet sich für Anwendungen, in denen ein AULI-Signal auf mehrere Verbraucher aufgeteilt werden muss.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden. Die Weiterleitung erfolgt rein datengetrieben.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden. Der Baustein erzeugt keine Steuerereignisse.

Daten-Eingänge

Keine Daten-Eingänge vorhanden. Der gesamte Datenaustausch erfolgt über den Adapter **IN**.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden. Die Ausgabe der Daten erfolgt ausschließlich über die Adapter **OUT1** bis **OUT9**.

Adapter

Typ	Richtung	Name	Beschreibung
Socket	Eingang	**IN**	Unidirektionaler AULI-Adapter – empfängt den zu verteilenden Datensatz.
Plug	Ausgang	**OUT1** … **OUT9**	Neun unidirektionale AULI-Adapter – geben den identischen Eingangsdatensatz aus.

Funktionsweise

Der Baustein arbeitet als reine 1:9-Verteilerstufe. Sobald ein Datensatz am Socket **IN** anliegt, wird dieser Datensatz unverändert auf alle neun Plugs **OUT1** bis **OUT9** kopiert. Da keine ereignisgesteuerten

Aktivierungen oder Verarbeitungsschritte existieren, erfolgt die Weitergabe implizit durch die Laufzeitumgebung, sobald sich der Eingangsdatensatz ändert. Der Baustein besitzt keinerlei interne Logik oder Zustandsspeicher.

Technische Besonderheiten

- **Generischer Aufbau**: Der Baustein ist unter dem generischen Klassennamen `GEN_AULI_SPLIT` registriert und kann je nach Konfiguration der Laufzeitumgebung angepasst werden.
- **Unidirektionaler Datenfluss**: Alle beteiligten Adapter sind vom Typ `adapter::types::unidirectional::AULI`, was eine klare Datenflussrichtung vorgibt – vom Socket IN zu den Plugs. Rückwirkungen sind nicht vorgesehen.
- **Keine Zustandshaltung**: Der Baustein ist zustandslos und benötigt keine Initialisierung oder spezielle Steuerung.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten. Er befindet sich stets im aktiven Durchleitungsmodus. Einziger „Zustand“ ist die Identität des Eingangsdatensatzes, der durchgereicht wird.

Anwendungsszenarien

- **Vervielfachung eines Steuer-/Messsignals** in der Automatisierungstechnik, z. B. in der Agrar- oder Landtechnik (Herkunft des Bausteins deutet auf entsprechende Umgebungen hin).
- **Aufteilung eines AULI-basierten Protokoll-Kanals** auf mehrere parallele Empfänger, ohne dass eine aktive Kopie oder Signalverstärkung nötig ist.
- **Einfache Sternverteilung** innerhalb einer 4diac-Applikation, wenn mehrere nachfolgende FBs denselben AULI-Datensatz benötigen.

Vergleich mit ähnlichen Bausteinen

- **AULI_SPLIT_4 / AULI_SPLIT_8**: Diese Varianten unterscheiden sich lediglich in der Anzahl der Ausgangskanäle. Der hier beschriebene Baustein bietet mit 9 Ausgängen eine besonders hohe Verteilungsdichte.
- **Generische Splitter-FBs für andere Adaptertypen**: Prinzipiell existieren analoge Splitter für z. B. `AULI`-Adapter mit geringerer Ausgangszahl. Allen gemeinsam ist die zustandslose 1:n-Vervielfältigung.
- `AULI_SPLIT_9_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der **AULI_SPLIT_9** ist ein einfacher, aber effektiver Verteilerbaustein für den unidirektionalen AULI-Adapter. Dank seiner generischen Natur und der vollständigen Signaldurchleitung ohne Latenz oder Verarbeitungsverzögerung eignet er sich ideal für Szenarien, in denen ein einzelner Datensatz an viele Empfänger gestreut werden muss. Die fehlende Ereignissteuerung und der reine Adapter-Ansatz machen ihn besonders leichtgewichtig und performant.

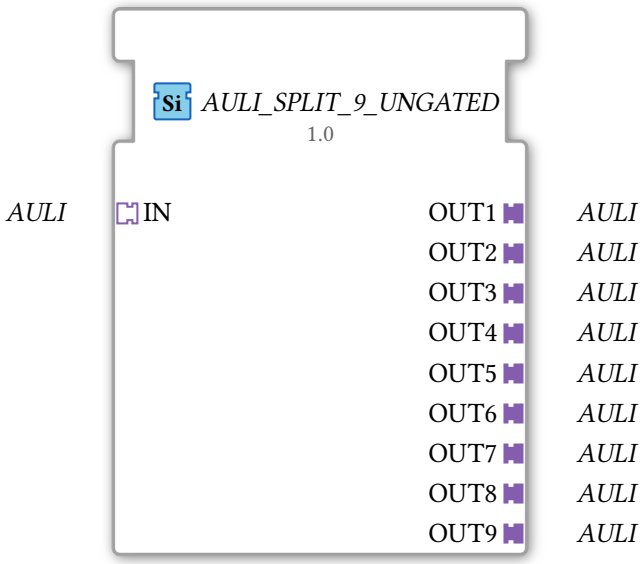
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AULI_SPLIT_9_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AULI_SPLIT_9`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die

sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AULI_SPLIT_9_UNGATED** dient als Verteiler für den unidirektionalen AULI-Adapter. Er nimmt einen eingehenden AULI-Datensatz über den Socket **IN** entgegen und leitet diesen unverändert an alle neun Ausgangs-Adapter (**OUT1** bis **OUT9**) weiter. Der Baustein ist als generischer Splitter konzipiert und eignet sich für Anwendungen, in denen ein AULI-Signal auf mehrere Verbraucher aufgeteilt werden muss.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden. Die Weiterleitung erfolgt rein datengetrieben.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden. Der Baustein erzeugt keine Steuerereignisse.

Daten-Eingänge

Keine Daten-Eingänge vorhanden. Der gesamte Datenaustausch erfolgt über den Adapter **IN**.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden. Die Ausgabe der Daten erfolgt ausschließlich über die Adapter **OUT1** bis **OUT9**.

Adapter

Typ	Richtung	Name	Beschreibung
Socket	Eingang	**IN**	Unidirektionaler AULI-Adapter – empfängt den zu verteilenden Datensatz.
Plug	Ausgang	**OUT1** … **OUT9**	Neun unidirektionale AULI-Adapter – geben den identischen Eingangsdatensatz aus.

Funktionsweise

Der Baustein arbeitet als reine 1:9-Verteilerstufe. Sobald ein Datensatz am Socket **IN** anliegt, wird dieser Datensatz unverändert auf alle neun Plugs **OUT1** bis **OUT9** kopiert. Da keine ereignisgesteuerten

Aktivierungen oder Verarbeitungsschritte existieren, erfolgt die Weitergabe implizit durch die Laufzeitumgebung, sobald sich der Eingangsdatensatz ändert. Der Baustein besitzt keinerlei interne Logik oder Zustandsspeicher.

Technische Besonderheiten

- **Generischer Aufbau**: Der Baustein ist unter dem generischen Klassennamen `GEN_AULI_SPLIT` registriert und kann je nach Konfiguration der Laufzeitumgebung angepasst werden.
- **Unidirektionaler Datenfluss**: Alle beteiligten Adapter sind vom Typ `adapter::types::unidirectional::AULI`, was eine klare Datenflussrichtung vorgibt – vom Socket IN zu den Plugs. Rückwirkungen sind nicht vorgesehen.
- **Keine Zustandshaltung**: Der Baustein ist zustandslos und benötigt keine Initialisierung oder spezielle Steuerung.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten. Er befindet sich stets im aktiven Durchleitungsmodus. Einziger „Zustand“ ist die Identität des Eingangsdatensatzes, der durchgereicht wird.

Anwendungsszenarien

- **Vervielfachung eines Steuer-/Messsignals** in der Automatisierungstechnik, z. B. in der Agrar- oder Landtechnik (Herkunft des Bausteins deutet auf entsprechende Umgebungen hin).
- **Aufteilung eines AULI-basierten Protokoll-Kanals** auf mehrere parallele Empfänger, ohne dass eine aktive Kopie oder Signalverstärkung nötig ist.
- **Einfache Sternverteilung** innerhalb einer 4diac-Applikation, wenn mehrere nachfolgende FBs denselben AULI-Datensatz benötigen.

Vergleich mit ähnlichen Bausteinen

- **AULI_SPLIT_4 / AULI_SPLIT_8**: Diese Varianten unterscheiden sich lediglich in der Anzahl der Ausgangskanäle. Der hier beschriebene Baustein bietet mit 9 Ausgängen eine besonders hohe Verteilungsdichte.
- **Generische Splitter-FBs für andere Adaptertypen**: Prinzipiell existieren analoge Splitter für z. B. `AULI`-Adapter mit geringerer Ausgangszahl. Allen gemeinsam ist die zustandslose 1:n-Vervielfältigung.
- `AULI_SPLIT_9`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

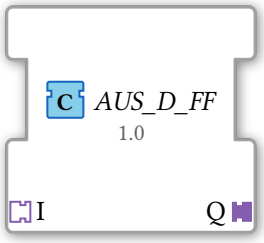
Der **AULI_SPLIT_9_UNGATED** ist ein einfacher, aber effektiver Verteilerbaustein für den unidirektionalen AULI-Adapter. Dank seiner generischen Natur und der vollständigen Signaldurchleitung ohne Latenz oder Verarbeitungsverzögerung eignet er sich ideal für Szenarien, in denen ein einzelner Datensatz an viele Empfänger gestreut werden muss. Die fehlende Ereignissteuerung und der reine Adapter-Ansatz machen ihn besonders leichtgewichtig und performant.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

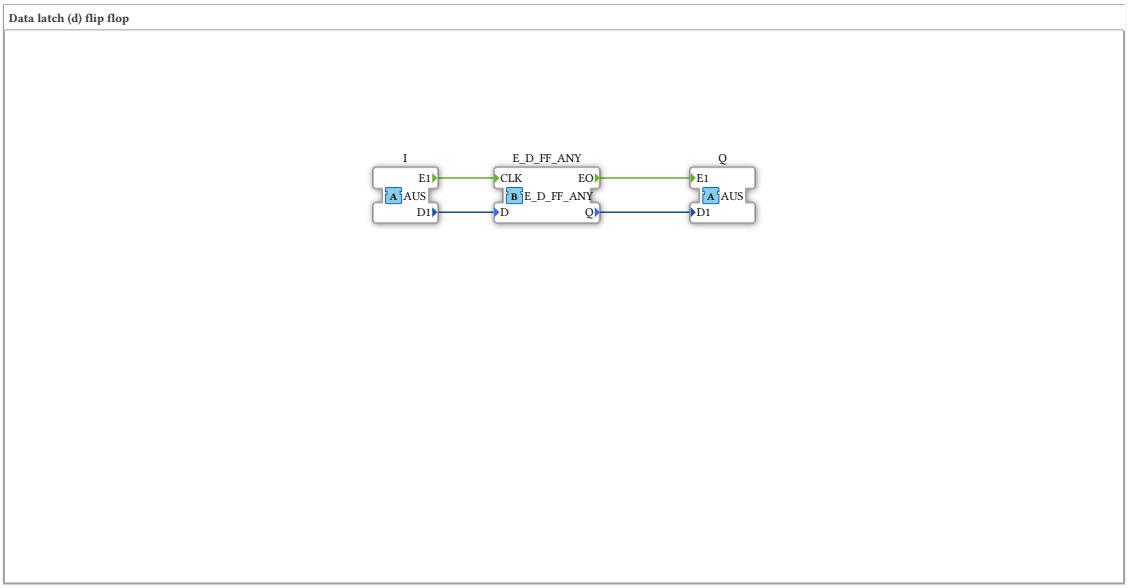
USINT

AUS_D_FF



Value to latch – *AUS*

AUS – Latched value



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AUS_D_FF** realisiert ein Data Latch (D-Flip-Flop) für den Datentyp `AUS`. Er speichert bei einem eingehenden Ereignis den anliegenden Datenwert und gibt diesen gespeicherten Wert über einen Adapter-Ausgang aus. Die Kommunikation erfolgt ausschließlich über unidirektionale Adapter.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge. Die ereignisgesteuerte Datenübernahme erfolgt über den Adapter-Eingang **I** (siehe Abschnitt Adapter).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge. Die Ausgabe des gespeicherten Wertes wird über den Adapter-Ausgang **Q** signalisiert (siehe Abschnitt Adapter).

Daten-Eingänge

Keine direkten Daten-Eingänge. Die zu speichernden Daten werden über den Adapter-Eingang **I** bereitgestellt.

Daten-Ausgänge

Keine direkten Daten-Ausgänge. Der gespeicherte Wert wird über den Adapter-Ausgang **Q** ausgegeben.

Adapter

Adapter	Richtung	Typ	Beschreibung
I	Socket	`adapter::types::unidirectional::AUS`	Liefert das zu latchende Signal (Ereignis E1 und Daten D1).
Q	Plug	`adapter::types::unidirectional::AUS`	Gibt den gelatchten Wert aus (Ereignis E1 und Daten D1).

Der intern verwendete Baustein `E_D_FF_ANY` erwartet an seinem **CLK**-Eingang ein Ereignis (übernommen von **I.E1**) und an **D** den Datenwert (von **I.D1**). Das ausgegebene Ereignis **EO** wird an **Q.E1** weitergeleitet, der gespeicherte Datenwert **Q** an **Q.D1**.

Funktionsweise

Sobald am Adapter-Eingang **I** ein Ereignis eintrifft (Pin **E1**), wird der gleichzeitig anliegende Datenwert (Pin **D1**) in den internen Speicher übernommen. Nach erfolgreicher Übernahme wird ein Ereignis am Adapter-Ausgang **Q** (Pin **E1**) ausgelöst und der gespeicherte Wert steht an **Q.D1** zur Verfügung. Der gespeicherte Wert bleibt solange erhalten, bis ein neues Ereignis an **I.E1** einen neuen Wert übernimmt.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle:** Der Baustein verwendet ausschließlich unidirektionale Adapter, was eine lose Kopplung und flexible Verschaltung ermöglicht.
- **Keine eigenen Ereignis-/Daten-Pins:** Die gesamte Ein- und Ausgabe erfolgt über die Adapter **I** und **Q**.
- **Interner Standard-Flip-Flop:** Die eigentliche Latch-Funktionalität wird vom IEC-61499-Baustein `E_D_FF_ANY` bereitgestellt, der spezifiziert ist, auf ein beliebiges Datentyp-Template zu arbeiten.

Zustandsübersicht

Der Baustein besitzt einen impliziten internen Zustand – den gespeicherten Datenwert (`Q`-Ausgang). Dieser ändert sich nur bei einem Ereignis an **I.E1**. Es gibt keine explizite Zustandsmaschine im FBNetz; der Zustand wird durch den internen `E_D_FF_ANY` verwaltet.

Anwendungsszenarien

- **Zwischenspeicherung:** Ein Signal (Wert + Ereignis) soll für eine spätere Verarbeitung zwischengespeichert werden, z. B. in einem Steuerungsablauf, bei dem ein Sensorwert erst bei einem bestimmten Takt übernommen werden soll.
- **Datensynchronisation:** Ein asynchron ankommender Datenwert wird synchron zu einem Ereignis gelatcht und dann im weiteren Programmfluss verwendet.
- **Zustandshaltung:** Der Baustein merkt sich den letzten gültigen Wert, bis ein neuer Wert explizit übernommen wird.

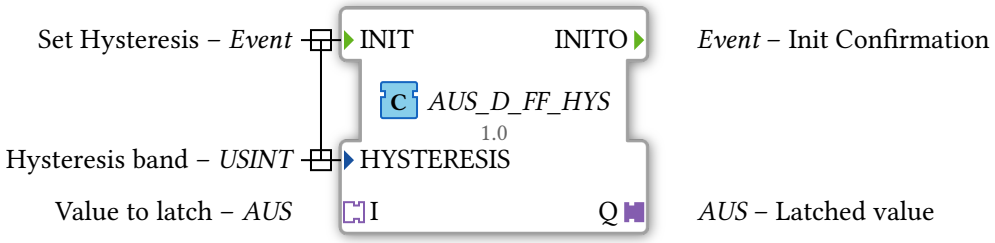
Vergleich mit ähnlichen Bausteinen

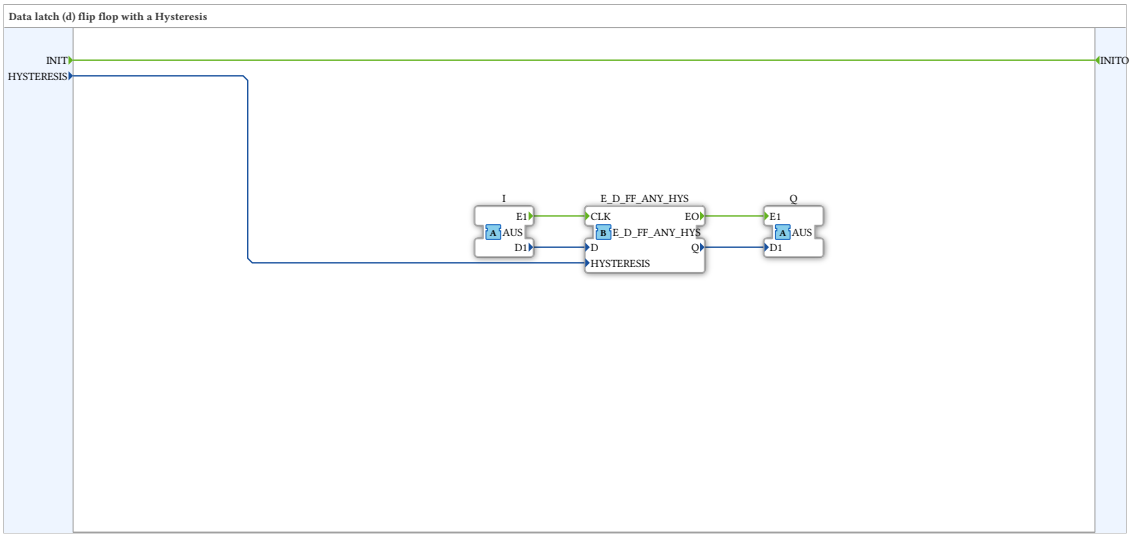
- **SR-Flip-Flop:** Setzt oder rückgesetzt den Ausgang über separate Set-/Reset-Eingänge; speichert nur boolesche oder binäre Zustände. `AUS_D_FF` hingegen speichert allgemeine Datenwerte vom Typ `AUS`.
- **Toggle-Flip-Flop (T-FF):** Wechselt den Ausgang bei jedem Takt. `AUS_D_FF` übernimmt den aktuellen Datenwert und gibt ihn unverändert aus.
- **MUX-basiertes Latch:** Ein Multiplexer könnte ebenfalls einen Wert speichern, benötigt aber zusätzliche Rücksetzlogik. `AUS_D_FF` kapselt diese Logik bereits.

Fazit

Der **AUS_D_FF** ist ein einfach zu verwendender Latch-Baustein, der über Adapter ein Datenwert-Ereignis-Paar übernimmt und gespeichert weitergibt. Aufgrund der Adapter-Schnittstelle und der Nutzung eines standardisierten internen Flip-Flops eignet er sich für modulare und wiederverwendbare Steuerungsentwürfe in der IEC-61499-Umgebung.

AUS_D_FF_HYS





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `AUS_D_FF_HYS` realisiert ein Data-Latch (D-Flipflop) mit einer Hysterese (Schwellwertband). Er dient dazu, einen eingehenden Datenwert flankengesteuert zu übernehmen und am Ausgang bereitzustellen, wobei durch die Hysterese ein stabiles Schaltverhalten bei rauschenden oder schwankenden Eingangssignalen erreicht wird. Der Baustein verwendet generische Adapter (`AUS`), die eine unidirektionale Datenübertragung mit Ereignissteuerung ermöglichen, und ist damit für beliebige Datentypen einsetzbar.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Comment	Mit Parameter
`INIT`	EInit	Set Hysteresis	`HYSTERESIS`

Ereignis-Ausgänge

Ereignis	Typ	Comment
`INITO`	EInit	Init Confirmation

Daten-Eingänge

Name	Typ	Comment
`HYSTERESIS`	USINT	Hysteresis band

Daten-Ausgänge

Der FB hat keine direkten Datenausgänge. Der latched Wert wird über den Adapterausgang `Q` bereitgestellt.

Adapter

Name	Typ	Richtung	Comment
`I`	`adapter::types::unidirectional::AUS`	Socket	Value to latch
`Q`	`adapter::types::unidirectional::AUS`	Plug	Latched value

Die Adapter sind unidirektional und enthalten jeweils ein Ereignis (`E1`) und einen Datenwert (`D1`). Über `I` wird der Eingangswert und dessen Übernahmeereignis empfangen, über `Q` wird der ausgegebene Wert und ein Bestätigungsereignis gesendet.

Funktionsweise

Der FB kapselt einen internen Baustein `E_D_FF_ANY_HYS`, der das eigentliche D-Flipflop mit Hysterese implementiert.

- **Initialisierung:** Mit dem Ereignis `INIT` wird der Hysterese-Wert (`HYSTERESIS`) an den internen Baustein übergeben. Das Ereignis `INITO` quittiert die Initialisierung.
- **Datenübernahme:** Ein Ereignis auf `I.E1` veranlasst den internen Baustein, den aktuellen Datenwert von `I.D1` zu übernehmen und den gelatchten Wert an `Q.D1` weiterzugeben. Gleichzeitig wird `Q.E1` ausgelöst.
- **Hysteresefunktion:** Der Ausgang des Flipflops ändert seinen Zustand nur dann, wenn der Eingangswert eine obere oder untere Schwelle überschreitet, die durch den Hysterese-Wert definiert ist. Dadurch wird ein Prellen oder Rauschen des Eingangssignals unterdrückt (Schmitt-Trigger-Verhalten).

Die genaue Schaltschwelle hängt vom verwendeten Datentyp des Adapters ab (der Adapter `AUS` unterstützt beliebige Typen). Der Hysterese-Wert (`USINT`) wird intern auf den entsprechenden Zahlenbereich skaliert oder interpretiert.

Technische Besonderheiten

- **Generischer Adapter:** Die Verwendung des unidirektionalen Adapters `AUS` erlaubt den Einsatz mit unterschiedlichsten Datentypen (z. B. `INT`, `REAL`, `BOOL`), ohne dass der FB selbst typisiert werden muss. Die Typanpassung erfolgt durch den internen Baustein.
- **Hysterese als USINT:** Die Hysterese wird als vorzeichenloser 8-Bit-Wert angegeben. Die konkrete Interpretation (z. B. als absolute Differenz oder prozentualer Anteil) ist implementierungsabhängig und liegt im internen Baustein.
- **Ereignis-Passthrough:** Das INIT-Ereignis wird direkt an das INITO-Ereignis durchgeschleift. Dies ermöglicht eine einfache Initialisierungskette im Netzwerk.
- **Keine Zustandsmaschine sichtbar:** Der FB besitzt kein eigenes ECC (Execution Control Chart); die Logik wird vollständig durch den internen FB realisiert.

Zustandsübersicht

Da der FB keine explizite Zustandsmaschine offenlegt, ergibt sich das Verhalten aus dem internen D-Flipflop mit Hysterese. Dieses besitzt zwei stabile Ausgangszustände:

- **Zustand 0:** Ausgangswert entspricht dem logischen „Low“-Niveau (bzw. dem unteren Grenzwert des Datenbereichs).
- **Zustand 1:** Ausgangswert entspricht dem logischen „High“-Niveau (bzw. dem oberen Grenzwert).

Zustandsübergänge erfolgen nur bei Überschreiten der jeweiligen Hystereseschwellen:

Übergang	Bedingung (Eingangswert)
$0 \longrightarrow 1$	Eingang überschreitet obere Schwelle
$1 \longrightarrow 0$	Eingang unterschreitet untere Schwelle

Die Schwellen sind um den Hysterese-Wert gegenüber einem Mittelwert oder Referenzwert versetzt (typischerweise obere Schwelle = Referenz + Hysterese/2, untere Schwelle = Referenz - Hysterese/2).

Anwendungsszenarien

- **Entprellung von Sensoren:** Ein digitaler oder analoger Sensor liefert schwankende Werte (z. B. durch mechanisches Prellen). Der FB glättet das Signal und gibt einen stabilen Ausgang.
- **Schwellwertschalter mit Rücksetzverzögerung:** Überwachung eines Prozesswertes, der bei Überschreiten eines oberen Grenzwertes ein Signal auslöst und erst beim Unterschreiten eines tieferen Grenzwertes wieder zurückgesetzt wird.
- **Signalaufbereitung in der Gebäudeautomatisierung:** Unterdrückung kurzer Störimpulse bei Temperatur-, Helligkeits- oder Füllstandssensoren.
- **Hysterese in Lagereglern:** Verhindern schnelles Ein- und Ausschalten von Aktoren nahe einer Sollwertgrenze.

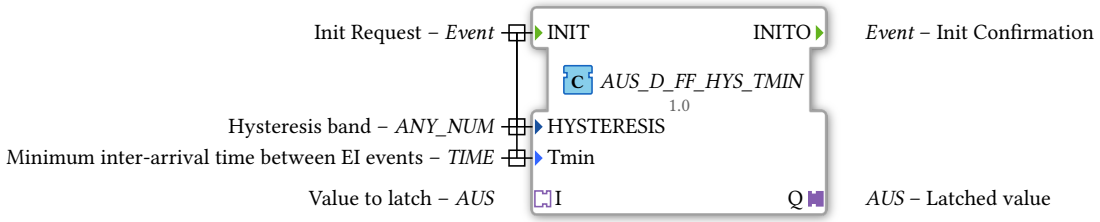
Vergleich mit ähnlichen Bausteinen

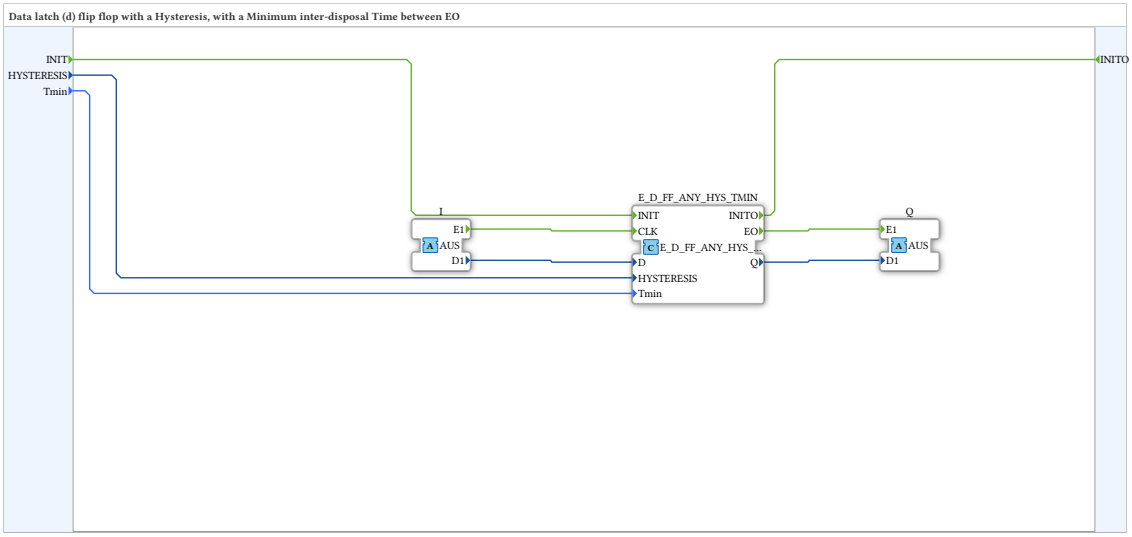
Baustein	Eigenschaften	Unterschied zu `AUS_D_FF_HYS`
Standard-D-Flipflop (`E_D_FF`)	Übernimmt Daten bei Taktflanke, keine Hysterese	Keine Rauschunterdrückung; Zustandswechsel bei jeder Flanke.
Schmitt-Trigger (`E_D_FF_ANY_HYS`)	Flipflop mit Hysterese, aber meist datentypspezifisch	`AUS_D_FF_HYS` kapselt den generischen Adapter und bietet eine einheitliche Schnittstelle für beliebige Typen.
SR-Flipflop (`E_SR_FF`)	Set-/Reset-Funktion ohne Takt	Keine flankengesteuerte Datenübernahme; Hysterese nicht direkt integriert.

Fazit

`AUS_D_FF_HYS` ist ein vielseitiger Funktionsblock für Anwendungen, die eine robuste, flankengesteuerte Datenübernahme mit Hysterese erfordern. Der Einsatz generischer Adapter macht ihn unabhängig vom konkreten Datentyp und erleichtert die Wiederverwendung in unterschiedlichen Kontexten. Besonders in der Automatisierungstechnik, Signalverarbeitung und Sensorauswertung bietet er durch die Unterdrückung von Rauschen und Prellen eine zuverlässige Signalaufbereitung. Die Initialisierung über `INIT` und die direkte Durchleitung des Ereignisses vereinfachen die Einbindung in bestehende Steuerungsnetzwerke.

AUS_D_FF_HYS_TMIN





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUS_D_FF_HYS_TMIN** realisiert ein datengetriggertes Flip-Flop (D-Flip-Flop) mit einer Hysterese und einer minimalen Zwischenereigniszeit. Er ist als Sub-Applikation (FB-Netzwerk) aufgebaut, die die eigentliche Logik an den inneren Baustein `logiBUS::signalprocessing::hysteresis::E_D_FF_ANY_HYS_TMIN` delegiert. Der Baustein dient dazu, einen analogen oder diskreten Wert mit vorgegebenem Hystereseband zu latchen und gleichzeitig die Rate der Ereignisse (über den Adapter-Eingang) auf einen konfigurierbaren Mindestabstand zu begrenzen.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Mit Variablen	Beschreibung
`INIT`	EInit	`HYSTERESIS`, `Tmin`	Initialisierungsanforderung; setzt Hysterese und Mindestzeit.

Ereignis-Ausgänge

Ereignis	Typ	Mit Variablen	Beschreibung
`INITO`	EInit	–	Bestätigung der abgeschlossenen Initialisierung.

Daten-Eingänge

Name	Typ	Beschreibung
`HYSTERESIS`	ANY_NUM	Hystereseband – Schwellwertdifferenz, die über-/unterschritten werden muss, damit ein neuer Wert gelatcht wird.
`Tmin`	TIME	Minimale Zeitspanne zwischen zwei eingehenden Ereignissen (Inter-arrival-Time).

Daten-Ausgänge

– Keine direkten Datenausgänge vorhanden. Der gelatchte Wert wird über den Adapter-Plug `Q` ausgegeben.

Adapter

Adapter	Richtung	Typ	Beschreibung

I	Socket (In)	adapter::types::unidirectional::AUS	Empfängt das zu latchende Ereignis und den dazugehörigen Datenwert.
Q	Plug (Out)	adapter::types::unidirectional::AUS	Gibt das verarbeitete Ereignis und den gelatchten Datenwert nach außen.

Funktionsweise

Der Baustein arbeitet als reine Kapselung eines internen Flip-Flops. Nach dem Start muss zuerst INIT mit den Parametern HYSTERESIS und Tmin aufgerufen werden. Danach können über den Adapter-Socket I Ereignisse (via I.E1) und zugehörige Datenwerte (via I.D1) anliegen.

- Jedes Ereignis am Eingang I.E1 wird an den internen FB E_D_FF_ANY_HYS_TMIN weitergeleitet.
- Die Daten I.D1 werden mit dem Hystereseband verglichen: Nur wenn der neue Wert außerhalb des durch die Hysterese definierten Totbands um den zuletzt gelatchten Wert liegt, wird das Flip-Flop aktualisiert.
- Zusätzlich wird die Zeit seit dem letzten gültigen Ereignis überwacht (Tmin). Liegt ein Ereignis zu früh (kürzer als Tmin) nach dem vorherigen, wird es ignoriert.
- Ein gültiges, gelatchtes Ereignis wird über den Adapter-Plug Q als Q.E1 und der zugehörige Datenwert als Q.D1 ausgegeben.

Der interne FB übernimmt die gesamte Logik; der äußere Baustein dient als Schnittstellen-Wrapper, der die Ereignisse und Daten entsprechend verbindet.

Technische Besonderheiten

- **Adapter-Schnittstelle:** Die Kommunikation erfolgt über die unidirektionalen Adapter adapter::types::unidirectional::AUS. Dies ermöglicht eine klare Trennung von Ereignis- und Datenfluss und eine einfache Wiederverwendung in verschiedenen Kontexten.
- **Hysterese:** Verhindert ständiges Umkippen bei verrauschten Signalen. Der gelatchte Wert ändert sich erst, wenn der Eingang das Hystereseband verlässt.
- **Minimale Zwischenereigniszeit (Tmin):** Entprellt die Ereignisse und verhindert Bursts. Nur Ereignisse, die mindestens Tmin auseinanderliegen, werden akzeptiert.
- **Kompakte Kapselung:** Der FB enthält kein eigenes Verhalten, sondern bildet eine logische Einheit aus Initialisierung, Ereignisverarbeitung und Ausgabe – ideal für modulare Steuerungsarchitekturen.

Zustandsübersicht

Da der Baustein kein eigenes Zustandsdiagramm besitzt (die Logik liegt im internen FB), ergibt sich das Zustandsverhalten aus dem Zusammenspiel von Ereignissteuerung und den Parametern:

1. **Initialisierung (INIT → INITO)**: Nach dem Start wird der interne FB konfiguriert. Erst danach können Ereignisse verarbeitet werden.
2. **Warten auf Eingangsereignis**: Der FB ist passiv, bis ein Ereignis über I.E1 eintrifft.
3. **Ereignisprüfung** (intern):
   - Zeitprüfung: Ist der Abstand zum letzten Ereignis ≥ Tmin?
   - Hystereseprüfung: Überschreitet der neue Wert den alten Wert um mehr als HYSTERESIS?
4. **Ausgabe**: Bei bestandener Prüfung wird Q.E1 ausgelöst und der neue Datenwert an Q.D1 übergeben.
5. **Rückkehr in Wartezustand**.

Anwendungsszenarien

- **Sensorauswertung mit Rauschen:** Ein Analogwert (z. B. Temperatur, Druck) soll nur dann einen Schaltvorgang auslösen, wenn er eine signifikante Änderung (Hysterese) zeigt und nicht zu schnell hintereinander kommt (Entprellung).
- **Frequenzbegrenzung:** In Kommunikationsprotokollen, bei denen ein Ereignis nicht häufiger als eine bestimmte Rate verarbeitet werden darf.
- **Sicherheitskritische Anwendungen:** Verhindern von Fehlauslösungen durch kurzzeitige Störimpulse (Mindestzeit zwischen zwei Flanken).

Vergleich mit ähnlichen Bausteinen

Baustein	Eigenschaften

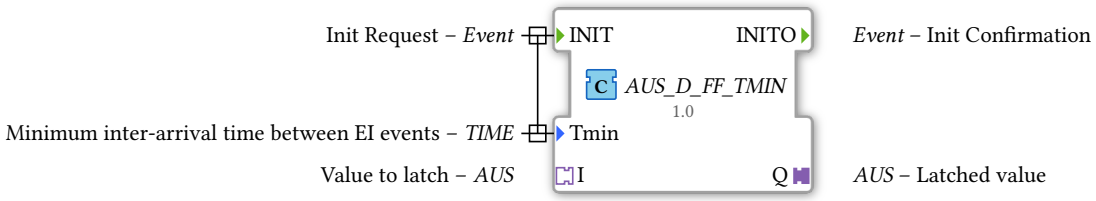
Standard-D-Flip-Flop	Latcht bei jeder steigenden Flanke ohne Hysterese oder Zeitbegrenzung.
`E_D_FF_HYS` (ohne Tmin)	Nur Hysterese, keine zeitliche Entprellung.
`E_D_FF_TMIN` (ohne Hys)	Nur Mindestzeit, keine Hysterese.
`AUS_D_FF_HYS_TMIN`	Kombiniert Hysterese **und** Mindestzeit – robuster gegen Rauschen und Bursts.

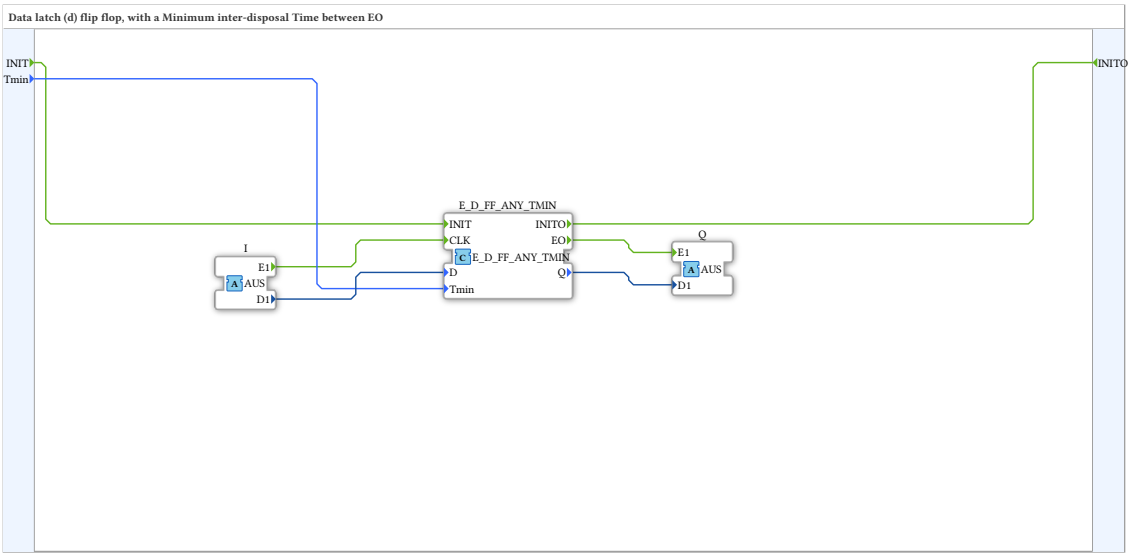
Der vorliegende Baustein eignet sich daher besonders dann, wenn beide Effekte (Signaltoleranz und zeitliche Begrenzung) gleichzeitig erforderlich sind.

Fazit

`AUS_D_FF_HYS_TMIN` ist ein spezialisierter D-Flip-Flop-Wrapper, der durch die Kombination von Hysterese und minimaler Ereignisabstandszeit eine zuverlässige, entprellte Datenlatch-Funktion bietet. Die Kapselung als Sub-Applikation mit Adapter-Schnittstellen macht ihn einfach in größere Steuerungsnetzwerke integrierbar. Die Initialisierung über `INIT` sorgt für eine definierte Startkonfiguration. Für Anwendungen, die eine robuste Schwellwerterkennung mit Rauschunterdrückung und Frequenzbegrenzung benötigen, ist dieser Baustein eine ideale Lösung.

AUS_D_FF_TMIN





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUS_D_FF_TMIN** realisiert ein taktgesteuertes D-Latch (Data Flip-Flop) mit einer minimalen Wartezeit zwischen zwei aufeinanderfolgenden Ausgabe-Ereignissen (Minimum inter-disposal Time between EO). Er eignet sich besonders für Anwendungen, bei denen ein eingehender Wert (über einen Adapter-Socket) übernommen und erst nach Ablauf einer einstellbaren Mindestzeit wieder ausgegeben werden darf. Die Initialisierung erfolgt über den Standard-Ereigniseingang **INIT**.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar	Mit Variable
INIT	EInit	Init Request	Tmin

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Init Confirmation

Daten-Eingänge

Name	Typ	Kommentar
HYSTERESIS	ANY_NUM	Hysterese-Band (derzeit nicht im internen Netzwerk verbunden)
Tmin	TIME	Minimale Zeitspanne zwischen zwei EO-Ereignissen

Daten-Ausgänge

Keine direkten Daten-Ausgänge – die Ausgabedaten werden ausschließlich über den **Adapter-Plug Q** bereitgestellt.

Adapter

Richtung	Name	Typ	Kommentar
Socket	I	adapter::types::unidirectional::AUS	Wert, der übernommen werden soll (enthält Ereignis E1 und Daten D1)
Plug	Q	adapter::types::unidirectional::AUS	Gelatchter Ausgabewert (enthält Ereignis E1 und Daten D1)

Funktionsweise

Der Baustein enthält intern einen vorgefertigten FB **E_D_FF_ANY_TMIN** (aus der Bibliothek `iec61499::events`). Dieser übernimmt bei jedem eingehenden Ereignis auf dem Clock-Eingang den momentanen Datenwert und gibt ihn am Ausgang weiter – jedoch nur, wenn die seit dem letzten Ausgabe-Ereignis vergangene Zeit mindestens **Tmin** beträgt.

- Beim **INIT**-Ereignis wird der Parameter *Tmin* an den internen FB übergeben.
- Ein Ereignis auf **I.E1** (Adapter-Socket) wirkt als Takt für den internen Flip-Flop.
- Der gelatchte Datenwert erscheint auf **Q.D1** (Adapter-Plug) und wird gleichzeitig mit einem Ereignis auf **Q.E1** quittiert – sofern die Zeitbedingung erfüllt ist.
- Der Parameter **HYSTERESIS** ist zwar deklariert, wird im aktuellen Netzwerk aber nicht verwendet. Er kann für zukünftige Erweiterungen oder für manuelle Verdrahtung genutzt werden.

Technische Besonderheiten

- **Mindestzeit zwischen Ausgaben**: Die Ausgabe des gelatchten Werts wird erst dann freigegeben, wenn die seit dem letzten **EO** vergangene Zeit ≥ *Tmin* ist. Dies verhindert eine zu hohe Ereignisrate am Ausgang.
- **Adapter-Schnittstelle**: Ein- und Ausgang sind als unidirektionale Adapter vom Typ **AUS** realisiert, sodass der Baustein leicht in bestehende Adapter-basierte Architekturen eingefügt werden kann.
- **Hysterese-Parameter**: Obwohl nicht direkt verdrahtet, steht der Eingang *HYSTERESIS* zur Verfügung – beispielsweise zur Realisierung eines Schmitt-Trigger-ähnlichen Verhaltens bei späteren Versionen.

Zustandsübersicht

Der interne FB **E_D_FF_ANY_TMIN** besitzt einen Speicher für den zuletzt übernommenen Datenwert. Dieser Zustand wird bei jedem Clock-Ereignis (I.E1) aktualisiert. Eine explizite Zustandsmaschine ist auf der oberen Ebene nicht vorhanden; das Verhalten ergibt sich aus der Logik des unterlagerten Flip-Flops.

Anwendungsszenarien

- **Sensorwert-Entprellung**: Ein- und Ausschaltschwellen mit minimaler Wiederholrate.
- **Datenpuffer mit Ausgabebegrenzung**: Verhindert das Überschwemmen nachgeschalteter Verbraucher mit zu vielen Aktualisierungen.
- **Zeitgesteuerte Latch-Funktion**: Übernahme eines Wertes nur dann, wenn die letzte Ausgabe länger als *Tmin* zurückliegt.

Vergleich mit ähnlichen Bausteinen

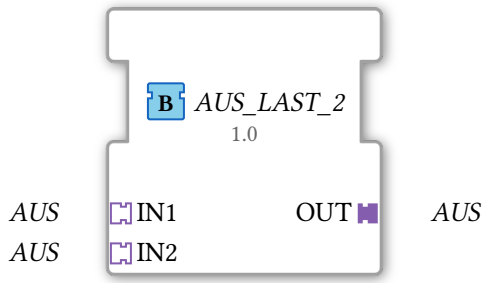
Baustein	Eigenschaften
E_D_FF (einfaches D-Flip-Flop)	Übernimmt Wert sofort bei jedem Takt, keine Zeitsteuerung.
AUS_D_FF_TMIN	Wie **E_D_FF**, aber mit zusätzlicher Mindestzeit zwischen den Ausgabeereignissen.
R_TRIG / F_TRIG	Erkennen Flanken, speichern aber keinen Datenwert.

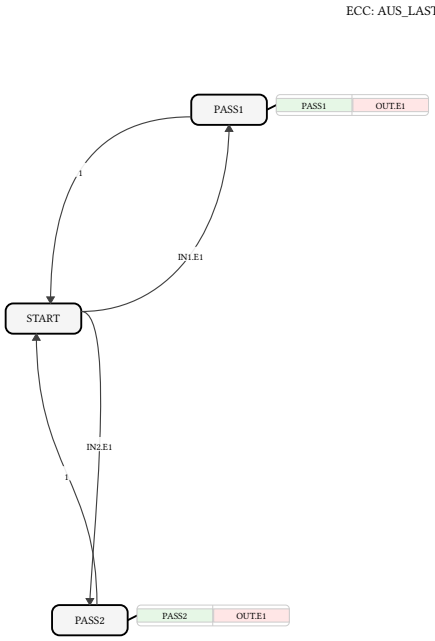
Der **AUS_D_FF_TMIN** eignet sich daher speziell für zeitkritische Anwendungen, bei denen eine minimale Abtast- oder Aktualisierungsrate eingehalten werden muss.

Fazit

Der **AUS_D_FF_TMIN** ist ein nützlicher Erweiterungsbaustein für das D-Flip-Flop um eine zeitliche Entkopplung der Ausgabe. Dank der Adapter-Schnittstelle und der einfachen Konfiguration über **Tmin** lässt er sich flexibel in IEC-61499-Systeme integrieren. Der vorhandene, aber ungenutzte **HYSTERESIS**-Eingang bietet Potenzial für künftige Funktionserweiterungen.

AUS_LAST_2





Einleitung

Der AUS_LAST_2 Funktionsblock führt zwei gleichartige USINT-Adapter-Signale (**IN1**, **IN2**) auf einen gemeinsamen Ausgang (**OUT**) zusammen, nach dem Prinzip **Last Writer Wins**: welcher der beiden Sockets zuletzt ein Ereignis geschrieben hat, gewinnt, und sein Datenwert wird sofort und unverändert an OUT durchgereicht. Es handelt sich ausdrücklich nicht um ein naives Zusammenführen (z. B. Mittelwert oder ODER-Verknüpfung), sondern um eine reine Arbitrierung nach zeitlicher Reihenfolge.

Schnittstellenstruktur

Adapter

Eingangsadapter (Sockets):

- **IN1**: USINT-Adapter (unidirectional) - erste Signalquelle
- **IN2**: USINT-Adapter (unidirectional) - zweite Signalquelle

Ausgangsadapter (Plug):

- **OUT**: USINT-Adapter (unidirectional) - zusammengeführtes Signal

Funktionsweise

Der Baustein ist als einfacher Basic-FB mit einer 3-Zustands-ECC umgesetzt (START, PASS1, PASS2). Trifft an IN1 ein Ereignis ein, wechselt die ECC nach PASS1: der aktuelle Datenwert von IN1 wird nach OUT kopiert und das Adapter-Ereignis an OUT gesendet. Trifft an IN2 ein Ereignis ein, geschieht dasselbe symmetrisch über PASS2. Nach jeder Durchreichung kehrt die ECC sofort wieder nach START zurück und ist bereit für das nächste Ereignis von IN1 oder IN2.

Technische Besonderheiten

- Last-Writer-Wins-Semantik statt naiver Datenverknüpfung
- Einfacher Basic-FB (ECC mit 3 Zuständen), kein zusätzlicher Zwischenspeicher nötig
- Keine Signalverzögerung: der Wert wird im selben Zyklus durchgereicht, in dem das auslösende Ereignis eintrifft
- Beide Eingänge sind gleichberechtigt - es gibt keine feste Priorität außer der zeitlichen Reihenfolge

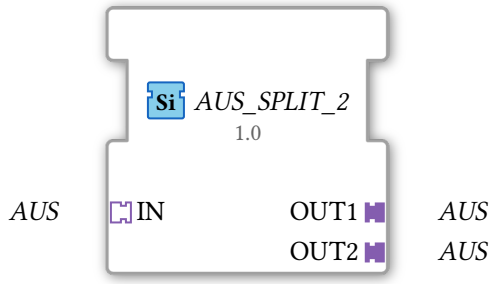
Anwendungsszenarien

- Zusammenführen zweier gleichwertiger USINT-Signalquellen zu einem einzigen nachgeschalteten Adapterpfad (z. B. zwei redundante Sensoren, oder ein manueller und ein automatischer Sollwertgeber)
- Arbitrierung zwischen zwei Bedienstellen, die denselben USINT-Wert schreiben dürfen
- Vereinfachung von Netzwerken, die sonst ein `F_SEL`/`E_RS`-Konstrukt inline verdrahten müssten

⚖️ Vergleich mit ähnlichen Bausteinen

AUS_LAST_2 ist die Umkehrung von `AUS_SPLIT_2`: AUS_SPLIT_2 verteilt 1 Eingang auf 2 Ausgänge, AUS_LAST_2 führt 2 Eingänge auf 1 Ausgang zusammen. Für Mischbetrieb zwischen einem Daten- und einem reinen Ereignisadapter (z. B. AX + AE) ist stattdessen `AX_AE_MERGE` gedacht, das nur das Ereignis mischt, ohne den Datenwert zu verändern.

AUS_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AUS_SPLIT_2** dient dazu, ein eingehendes AUS-Signal auf zwei identische Ausgänge zu verteilen. Er ist als generischer Baustein (generic FB) realisiert und eignet sich für Anwendungen, bei denen ein universelles Steuersignal mehrfach benötigt wird. Der Baustein arbeitet rein adapterbasiert und besitzt keine eigenen Ereignis- oder Dateneingänge.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Sockets	IN	AUS (unidirectional)	Eingangsadapter, der das zu verteilende Signal empfängt
Plugs	OUT1	AUS (unidirectional)	Erster Ausgangsadapter, identisch zum Eingangssignal
Plugs	OUT2	AUS (unidirectional)	Zweiter Ausgangsadapter, identisch zum Eingangssignal

Funktionsweise

Der Baustein leitet das am Socket **IN** anliegende AUS-Signal unverändert an beide Plugs **OUT1** und **OUT2** weiter. Es findet keine Verarbeitung oder Pufferung statt – die Aufteilung erfolgt rein topologisch. Das

Eingangssignal wird auf beiden Ausgängen gleichzeitig und ohne Verzögerung zur Verfügung gestellt. Die Verbindung wird erst dann aktiv, wenn der Socket mit einem entsprechenden Adapter verbunden ist.

Technische Besonderheiten

- **Generischer Typ**: Der Baustein ist als `GEN_AUS_SPLIT` mit dem Attribut `eclipse4diac::core::GenericClassName` deklariert. Dadurch kann er in verschiedenen Projekten ohne Typanpassung wiederverwendet werden.
- **Keine Zustandsabhängigkeit**: Der Baustein arbeitet zustandslos – es gibt kein internes Verhalten, das von einer Zustandsmaschine gesteuert wird.
- **Adapterbasiert**: Alle Schnittstellen sind als Adapter vom Typ `adapter::types::unidirectional::AUS` ausgeführt. Dies ermöglicht flexible Verkabelung in einer gerichteten Kommunikation.
- **Copyright**: Der Baustein stammt von der HR Agrartechnik GmbH und unterliegt der Eclipse Public License 2.0.

Zustandsübersicht

Da der Baustein keinerlei Zustandslogik besitzt, existiert keine Zustandsmaschine. Die Funktionalität beschränkt sich auf die einfache Signaldurchleitung.

Anwendungsszenarien

- **Signalverteilung in der Steuerungstechnik**: Wenn ein Bus-Signal oder ein universelles Steuersignal an mehrere parallele Module weitergegeben werden muss.
- **Test- und Simulationsaufbauten**: Um ein einzelnes Testsignal auf zwei parallele Pfade zu splitten.
- **Redundante Anbindung**: Falls ein Signal aus Gründen der Verfügbarkeit an zwei unabhängige Empfänger gesendet werden soll.

Vergleich mit ähnlichen Bausteinen

- **AUS_SPLIT_3 / AUS_SPLIT_N**: Analoge Bausteine mit drei oder mehr Ausgängen. `AUS_SPLIT_2` ist die einfachste Variante zur Aufteilung auf zwei Kanäle.
- **Ereignis-basierte Splitter**: Im Gegensatz zu Bausteinen mit Ereignis-Eingängen (z.B. `E_SPLIT`) arbeitet `AUS_SPLIT_2` ausschließlich über Adapter und eignet sich daher für reine Signalverteilung ohne Steuerlogik.
- **Merge-Bausteine**: Während Splitter Signale vervielfältigen, fassen Merge-Bausteine mehrere Signale zu einem zusammen (z.B. `AUS_MERGE_2`).
- **`AUS_SPLIT_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der `AUS_SPLIT_2` ist ein minimalistischer, aber nützlicher Funktionsbaustein zur dezentralen Signalverteilung in 4diac-Anwendungen. Seine generische Natur und die reine Adapter-Schnittstelle machen ihn universell einsetzbar, insbesondere wenn nur eine unidirektionale Signalkopie benötigt wird. Für komplexere Aufgaben mit Steuer- oder Verarbeitungslogik sind erweiterte Varianten erforderlich.

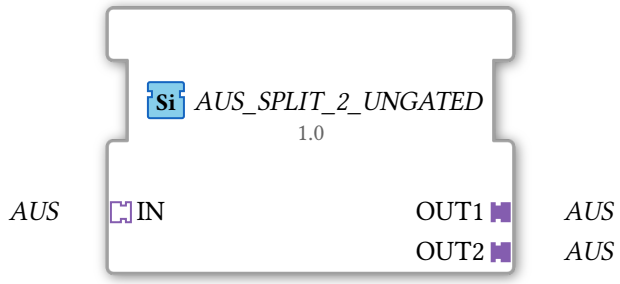
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUS_SPLIT_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUS_SPLIT_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weiter-

gegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AUS_SPLIT_2_UNGATED** dient dazu, ein eingehendes AUS-Signal auf zwei identische Ausgänge zu verteilen. Er ist als generischer Baustein (generic FB) realisiert und eignet sich für Anwendungen, bei denen ein universelles Steuersignal mehrfach benötigt wird. Der Baustein arbeitet rein adapterbasiert und besitzt keine eigenen Ereignis- oder Dateneingänge.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Sockets	IN	AUS (unidirectional)	Eingangsadapter, der das zu verteilende Signal empfängt
Plugs	OUT1	AUS (unidirectional)	Erster Ausgangsadapter, identisch zum Eingangssignal
Plugs	OUT2	AUS (unidirectional)	Zweiter Ausgangsadapter, identisch zum Eingangssignal

Funktionsweise

Der Baustein leitet das am Socket **IN** anliegende AUS-Signal unverändert an beide Plugs **OUT1** und **OUT2** weiter. Es findet keine Verarbeitung oder Pufferung statt – die Aufteilung erfolgt rein topologisch. Das Eingangssignal wird auf beiden Ausgängen gleichzeitig und ohne Verzögerung zur Verfügung gestellt. Die Verbindung wird erst dann aktiv, wenn der Socket mit einem entsprechenden Adapter verbunden ist.

Technische Besonderheiten

- **Generischer Typ**: Der Baustein ist als `GEN_AUS_SPLIT` mit dem Attribut `eclipse4diac::core::GenericClassName` deklariert. Dadurch kann er in verschiedenen Projekten ohne Typanpassung wiederverwendet werden.
- **Keine Zustandsabhängigkeit**: Der Baustein arbeitet zustandslos – es gibt kein internes Verhalten, das von einer Zustandsmaschine gesteuert wird.

- **Adapterbasiert**: Alle Schnittstellen sind als Adapter vom Typ `adapter::types::unidirectional::AUS` ausgeführt. Dies ermöglicht flexible Verkabelung in einer gerichteten Kommunikation.
- **Copyright**: Der Baustein stammt von der HR Agrartechnik GmbH und unterliegt der Eclipse Public License 2.0.

Zustandsübersicht

Da der Baustein keinerlei Zustandslogik besitzt, existiert keine Zustandsmaschine. Die Funktionalität beschränkt sich auf die einfache Signaldurchleitung.

Anwendungsszenarien

- **Signalverteilung in der Steuerungstechnik**: Wenn ein Bus-Signal oder ein universelles Steuersignal an mehrere parallele Module weitergegeben werden muss.
- **Test- und Simulationsaufbauten**: Um ein einzelnes Testsignal auf zwei parallele Pfade zu splitten.
- **Redundante Anbindung**: Falls ein Signal aus Gründen der Verfügbarkeit an zwei unabhängige Empfänger gesendet werden soll.

Vergleich mit ähnlichen Bausteinen

- **AUS_SPLIT_3 / AUS_SPLIT_N**: Analoge Bausteine mit drei oder mehr Ausgängen. `AUS_SPLIT_2_UNGATED` ist die einfachste Variante zur Aufteilung auf zwei Kanäle.
- **Ereignis-basierte Splitter**: Im Gegensatz zu Bausteinen mit Ereignis-Eingängen (z.B. `E_SPLIT`) arbeitet `AUS_SPLIT_2_UNGATED` ausschließlich über Adapter und eignet sich daher für reine Signalverteilung ohne Steuerlogik.
- **Merge-Bausteine**: Während Splitter Signale vervielfältigen, fassen Merge-Bausteine mehrere Signale zu einem zusammen (z.B. `AUS_MERGE_2`).
- `AUS_SPLIT_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

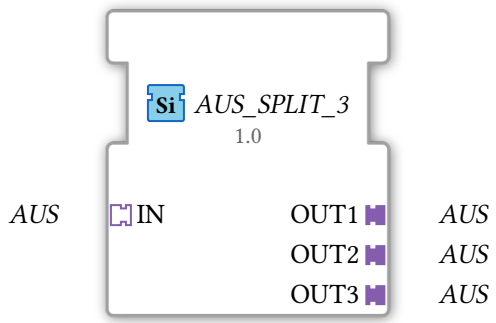
Fazit

Der `AUS_SPLIT_2_UNGATED` ist ein minimalistischer, aber nützlicher Funktionsbaustein zur dezentralen Signalverteilung in 4diac-Anwendungen. Seine generische Natur und die reine Adapter-Schnittstelle machen ihn universell einsetzbar, insbesondere wenn nur eine unidirektionale Signalkopie benötigt wird. Für komplexere Aufgaben mit Steuer- oder Verarbeitungslogik sind erweiterte Varianten erforderlich.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUS_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUS_SPLIT_3** dient als universeller Verteiler für eingehende Adapter-Signale vom Typ `adapter::types::unidirectional::AUS`. Er nimmt ein solches Signal entgegen und leitet es unverändert an drei baugleiche Ausgangsadapter weiter. Dadurch lassen sich Steuerungs-, Alarm- oder Statusinformationen einfach auf mehrere nachfolgende Bausteine aufteilen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine – die Signalweitergabe erfolgt über den Adapter-IN.

Ereignis-Ausgänge

Keine – die Ausgabe erfolgt über die Adapter-OUTs.

Daten-Eingänge

Keine – der Baustein besitzt keine dedizierten Dateneingänge; alle Daten werden über den Adapter `IN` transportiert.

Daten-Ausgänge

Keine – die Daten werden über die Adapter `OUT1`, `OUT2`, `OUT3` ausgegeben.

Adapter

Name	Typ	Richtung	Beschreibung
`IN`	`adapter::types::unidirectional::AUS`	Socket (eingehend)	Empfängt ein AUS-Signal.
`OUT1`	`adapter::types::unidirectional::AUS`	Plug (ausgehend)	Erste Ausgabe des eingehenden Signals.
`OUT2`	`adapter::types::unidirectional::AUS`	Plug (ausgehend)	Zweite Ausgabe des eingehenden Signals.
`OUT3`	`adapter::types::unidirectional::AUS`	Plug (ausgehend)	Dritte Ausgabe des eingehenden Signals.

Funktionsweise

Der FB arbeitet als reiner Signalverteiler. Sobald ein am Socket `IN` anliegendes AUS-Signal (bestehend aus Ereignis und zugehörigen Daten) aktiv wird, wird es ohne Verzögerung oder Umformung identisch an alle drei Plugs `OUT1`, `OUT2` und `OUT3` weitergegeben. Jeder Ausgang empfängt dabei dasselbe Ereignis und dieselben Daten. Es findet keine Pufferung, Filterung oder logische Verarbeitung statt.

Technische Besonderheiten

- **Generischer Typ**: Der Baustein ist als generischer FB deklariert (`eclipse4diac::core::GenericClassName = 'GEN_AUS_SPLIT'`). Dadurch kann er an unterschiedliche AUS-Adapter-Typen angepasst werden, sofern diese die gleiche Schnittstellenstruktur aufweisen.
- **Zustandslos**: Der FB besitzt kein internes Zustandsdiagramm (ECC). Seine Reaktion ist rein datenflussgetrieben und benötigt keine Initialisierung oder Resetlogik.
- **Lizenzhinweis**: Der Baustein wird unter der Eclipse Public License 2.0 bereitgestellt.

Zustandsübersicht

Da **AUS_SPLIT_3** über keine Ablaufsteuerung oder explizite Zustände verfügt, entfällt eine Zustandsmaschine. Das Verhalten ist deterministisch und sofort: Ein Eingangssignal erzeugt synchron drei identische Ausgangssignale.

Anwendungsszenarien

- **Verteilung von Alarmsignalen** an mehrere Anzeige- oder Logikeinheiten.
- **Mehrfachnutzung eines Sensorsignals** in verschiedenen Automatisierungszweigen.
- **Aufteilung eines Steuerbefehls** (z. B. „Stillsetzen“) auf mehrere Aktoren oder Subsysteme.
- **Signalkaskadierung** in modularen 4diac-Applikationen.

Vergleich mit ähnlichen Bausteinen

- **EVENT_SPLIT**: Verteilt nur Ereignisse (ohne Daten). **AUS_SPLIT_3** verteilt dagegen komplette Adapter-Signale inklusive aller assoziierten Daten.
- **DATA_SPLIT**: Verteilt reine Datenkanäle, erfordert separate Ereignissteuerung. Der AUS-Splitter kapselt Ereignis und Daten gemeinsam im Adapter.
- **Manuelle Implementierung**: Ohne diesen Baustein müsste jeder Ausgang mit einer eigenen Kopie des Quelladapters verbunden werden. **AUS_SPLIT_3** vereinfacht die Verdrahtung und erhöht die Übersichtlichkeit.
- `AUS_SPLIT_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

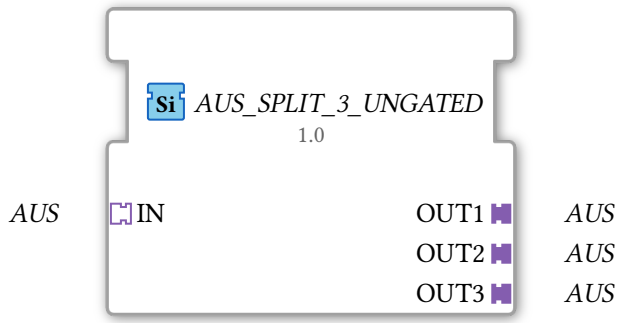
Der **AUS_SPLIT_3** ist ein einfacher aber nützlicher Funktionsblock zur Vervielfachung von AUS-Adapter-Signalen. Aufgrund seiner generischen Auslegung und Zustandslosigkeit eignet er sich hervorragend für die modulare und wartungsfreundliche Strukturierung von 4diac-Anwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUS_SPLIT_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUS_SPLIT_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUS_SPLIT_3_UNGATED** dient als universeller Verteiler für eingehende Adapter-Signale vom Typ `adapter::types::unidirectional::AUS`. Er nimmt ein solches Signal entgegen und leitet es unverändert an drei baugleiche Ausgangsadapter weiter. Dadurch lassen sich Steuerungs-, Alarm- oder Statusinformationen einfach auf mehrere nachfolgende Bausteine aufteilen.

Schnittstellenstruktur

Ereignis-Eingänge
Keine – die Signalweitergabe erfolgt über den Adapter-IN.

Ereignis-Ausgänge
Keine – die Ausgabe erfolgt über die Adapter-OUTs.

Daten-Eingänge
Keine – der Baustein besitzt keine dedizierten Dateneingänge; alle Daten werden über den Adapter `IN` transportiert.

Daten-Ausgänge
Keine – die Daten werden über die Adapter `OUT1`, `OUT2`, `OUT3` ausgegeben.

Adapter

Name	Typ	Richtung	Beschreibung
`IN`	`adapter::types::unidirectional::AUS`	Socket (eingehend)	Empfängt ein AUS-Signal.
`OUT1`	`adapter::types::unidirectional::AUS`	Plug (ausgehend)	Erste Ausgabe des eingehenden Signals.
`OUT2`	`adapter::types::unidirectional::AUS`	Plug (ausgehend)	Zweite Ausgabe des eingehenden Signals.
`OUT3`	`adapter::types::unidirectional::AUS`	Plug (ausgehend)	Dritte Ausgabe des eingehenden Signals.

Funktionsweise
Der FB arbeitet als reiner Signalverteiler. Sobald ein am Socket `IN` anliegendes AUS-Signal (bestehend aus Ereignis und zugehörigen Daten) aktiv wird, wird es ohne Verzögerung oder Umformung identisch an alle drei Plugs `OUT1`, `OUT2` und `OUT3` weitergegeben. Jeder Ausgang empfängt dabei dasselbe Ereignis und dieselben Daten. Es findet keine Pufferung, Filterung oder logische Verarbeitung statt.

Technische Besonderheiten
- **Generischer Typ**: Der Baustein ist als generischer FB deklariert (`eclipse4diac::core::GenericClassName = 'GEN_AUS_SPLIT'`). Dadurch kann er an unterschiedliche AUS-Adapter-Typen angepasst werden, sofern diese die gleiche Schnittstellenstruktur aufweisen.
- **Zustandslos**: Der FB besitzt kein internes Zustandsdiagramm (ECC). Seine Reaktion ist rein datenflussgetrieben und benötigt keine Initialisierung oder Resetlogik.
- **Lizenzhinweis**: Der Baustein wird unter der Eclipse Public License 2.0 bereitgestellt.

Zustandsübersicht
Da **AUS_SPLIT_3_UNGATED** über keine Ablaufsteuerung oder explizite Zustände verfügt, entfällt eine Zustandsmaschine. Das Verhalten ist deterministisch und sofort: Ein Eingangssignal erzeugt synchron drei identische Ausgangssignale.

Anwendungsszenarien
- **Verteilung von Alarmsignalen** an mehrere Anzeige- oder Logikeinheiten.
- **Mehrfachnutzung eines Sensorsignals** in verschiedenen Automatisierungszweigen.
- **Aufteilung eines Steuerbefehls** (z. B. „Stillsetzen“) auf mehrere Aktoren oder Subsysteme.
- **Signalkaskadierung** in modularen 4diac-Applikationen.

Vergleich mit ähnlichen Bausteinen
- **EVENT_SPLIT**: Verteilt nur Ereignisse (ohne Daten). **AUS_SPLIT_3_UNGATED** verteilt dagegen komplette Adapter-Signale inklusive aller assoziierten Daten.
- **DATA_SPLIT**: Verteilt reine Datenkanäle, erfordert separate Ereignissteuerung. Der AUS-Splitter kapselt Ereignis und Daten gemeinsam im Adapter.

- **Manuelle Implementierung**: Ohne diesen Baustein müsste jeder Ausgang mit einer eigenen Kopie des Quelladapters verbunden werden. **AUS_SPLIT_3_UNGATED** vereinfacht die Verdrahtung und erhöht die Übersichtlichkeit.
- `AUS_SPLIT_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

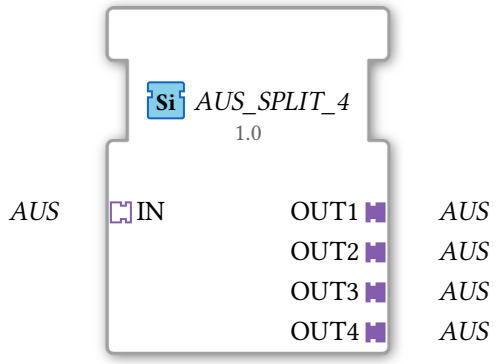
Fazit

Der **AUS_SPLIT_3_UNGATED** ist ein einfacher aber nützlicher Funktionsblock zur Vervielfachung von AUS-Adapter-Signalen. Aufgrund seiner generischen Auslegung und Zustandslosigkeit eignet er sich hervorragend für die modulare und wartungsfreundliche Strukturierung von 4diac-Anwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUS_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AUS_SPLIT_4` verteilt ein eingehendes **AUS**-Signal (typischerweise ein binäres Steuersignal für „Aus“) auf vier identische Ausgangssignale. Er dient als generischer Splitter und ermöglicht es, einen einzelnen Befehl oder Zustand an mehrere nachgeschaltete Komponenten weiterzuleiten, ohne dass eine separate Verzweigungslogik erforderlich ist.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Der Baustein kommuniziert ausschließlich über Adapter vom Typ `adapter::types::unidirectional::AUS`. Es handelt sich um einen unidirektionalen Adapter, der ein boolesches Signal (AUS-Kommando) überträgt.

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	`IN`	`adapter::types::unidirectional::AUS`	Empfängt das ursprüngliche AUS-Signal.
Plug (Ausgang)	`OUT1`	`adapter::types::unidirectional::AUS`	Erster ausgehender AUS-Pfad.
Plug (Ausgang)	`OUT2`	`adapter::types::unidirectional::AUS`	Zweiter ausgehender AUS-Pfad.
Plug (Ausgang)	`OUT3`	`adapter::types::unidirectional::AUS`	Dritter ausgehender AUS-Pfad.
Plug (Ausgang)	`OUT4`	`adapter::types::unidirectional::AUS`	Vierter ausgehender AUS-Pfad.

Funktionsweise

Der Baustein arbeitet als reine Signalverteilung: Sobald am Socket `IN` ein AUS-Signal anliegt, wird dieses unverändert an alle vier Plugs (`OUT1` bis `OUT4`) weitergegeben. Es findet keine logische Verarbeitung, Verzögerung oder Zustandsänderung statt. Die Verteilung erfolgt parallel und sofort.

Der generische Charakter des Bausteins (erkennbar am Attribut `GenericClassName = 'GEN_AUS_SPLIT'`) erlaubt die Wiederverwendung in verschiedenen Applikationen, bei denen der gleiche AUS-Signaltyp verwendet wird.

Technische Besonderheiten

- **Lizenz**: Der Quellcode unterliegt der **Eclipse Public License 2.0** (EPL-2.0).
- **Urheber**: Entwickelt von **HR Agrartechnik GmbH**, Version 1.0 vom 24.01.2025.
- **Generische Implementierung**: Der Baustein ist als generischer FB deklariert (`GenericClassName = 'GEN_AUS_SPLIT'`), was eine einfache Anpassung an verschiedene Signaltypen oder Konfigurationen ermöglicht.
- **Adapterbasiert**: Die Kommunikation erfolgt ausschließlich über Adapter, nicht über klassische Ereignis- oder Datenports. Dadurch kann der Baustein in einer adapterbasierten Architektur nahtlos integriert werden.
- **Keine Zustandsmaschine**: Es existiert keine interne Zustandslogik – die Verteilung ist rein kombinatorisch.

Zustandsübersicht

Der Baustein besitzt keine Zustandsmaschine. Die Ausgangssignale folgen direkt dem Eingangssignal. Eine grafische Zustandsübersicht entfällt daher.

Anwendungsszenarien

- **Rückkopplung von AUS-Signalen**: Ein zentrales „Aus“-Kommando einer Steuerung soll gleichzeitig mehrere Aktoren oder Subsysteme abschalten.
- **Redundante Überwachung**: Verteilen eines AUS-Signals an mehrere Überwachungseinheiten, die unabhängig voneinander auf den Befehl reagieren müssen.
- **Modulare Maschinenstruktur**: In einer modularen Anlage wird ein einmal erfasstes AUS-Signal über Busse oder Koppelelemente zu mehreren Modulen geführt. `AUS_SPLIT_4` ersetzt aufwändige Verdrahtung oder logische ODER-Verknüpfungen.
- **Test und Simulation**: Zum gleichzeitigen Ansteuern mehrerer simulierter Komponenten mit demselben Signal.

Vergleich mit ähnlichen Bausteinen

Baustein	Funktion	Besonderheit
`AUS_SPLIT_2`	Verteilt ein AUS-Signal auf zwei Ausgänge	Weniger Ports, kompakter.
`AUS_SPLIT_4`	Verteilt auf vier Ausgänge	Dieses FB.
`AUS_SPLIT_N`	Parametrierbarer Splitter (z. B. durch generische Adapterlisten)	Flexiblere Anzahl, aber aufwändiger in der Konfiguration.
`AUS_MERGE`	Fasst mehrere AUS-Eingänge zu einem Ausgang zusammen	Gegenstück zum Splitter.

Der AUS_SPLIT_4 liegt in der Mitte zwischen einem einfachen 2-fach-Splitter und einem vollparametrierbaren Splitter. Er ist ideal, wenn exakt vier Ausgänge benötigt werden – ohne zusätzliche Konfiguration.

- **AUS_SPLIT_4_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

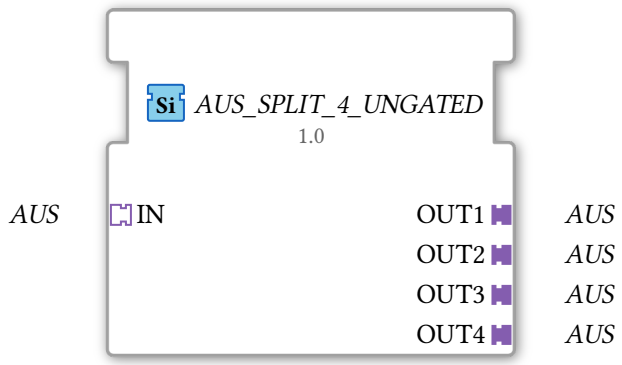
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der AUS_SPLIT_4 ist ein einfacher, aber nützlicher generischer Funktionsblock zur Verteilung eines AUS-Signals auf vier parallele Pfade. Seine adapterbasierte Schnittstelle und die klare Trennung von Logik und Signalweitergabe machen ihn zu einem robusten Baustein in der IEC 61499-basierten Automatisierung. Dank der generischen Deklaration kann er problemlos in verschiedene Projekte eingebunden und ggf. an eigene Signaltypen angepasst werden.

AUS_SPLIT_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AUS_SPLIT_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AUS_SPLIT_4_UNGATED verteilt ein eingehendes **AUS**-Signal (typischerweise ein binäres Steuersignal für „Aus“) auf vier identische Ausgangssignale. Er dient als generischer Splitter und ermöglicht es, einen einzelnen Befehl oder Zustand an mehrere nachgeschaltete Komponenten weiterzuleiten, ohne dass eine separate Verzweigungslogik erforderlich ist.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Der Baustein kommuniziert ausschließlich über Adapter vom Typ `adapter::types::unidirectional::AUS`. Es handelt sich um einen unidirektionalen Adapter, der ein boolesches Signal (AUS-Kommando) überträgt.

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	`IN`	`adapter::types::unidirectional::AUS`	Empfängt das ursprüngliche AUS-Signal.
Plug (Ausgang)	`OUT1`	`adapter::types::unidirectional::AUS`	Erster ausgehender AUS-Pfad.
Plug (Ausgang)	`OUT2`	`adapter::types::unidirectional::AUS`	Zweiter ausgehender AUS-Pfad.
Plug (Ausgang)	`OUT3`	`adapter::types::unidirectional::AUS`	Dritter ausgehender AUS-Pfad.
Plug (Ausgang)	`OUT4`	`adapter::types::unidirectional::AUS`	Vierter ausgehender AUS-Pfad.

Funktionsweise

Der Baustein arbeitet als reine Signalverteilung: Sobald am Socket `IN` ein AUS-Signal anliegt, wird dieses unverändert an alle vier Plugs (`OUT1` bis `OUT4`) weitergegeben. Es findet keine logische Verarbeitung, Verzögerung oder Zustandsänderung statt. Die Verteilung erfolgt parallel und sofort.

Der generische Charakter des Bausteins (erkennbar am Attribut `GenericClassName = 'GEN_AUS_SPLIT'`) erlaubt die Wiederverwendung in verschiedenen Applikationen, bei denen der gleiche AUS-Signaltyp verwendet wird.

Technische Besonderheiten

- **Lizenz**: Der Quellcode unterliegt der **Eclipse Public License 2.0** (EPL-2.0).
- **Urheber**: Entwickelt von **HR Agrartechnik GmbH**, Version 1.0 vom 24.01.2025.
- **Generische Implementierung**: Der Baustein ist als generischer FB deklariert (`GenericClassName = 'GEN_AUS_SPLIT'`), was eine einfache Anpassung an verschiedene Signaltypen oder Konfigurationen ermöglicht.
- **Adapterbasiert**: Die Kommunikation erfolgt ausschließlich über Adapter, nicht über klassische Ereignis- oder Datenports. Dadurch kann der Baustein in einer adapterbasierten Architektur nahtlos integriert werden.
- **Keine Zustandsmaschine**: Es existiert keine interne Zustandslogik – die Verteilung ist rein kombinatorisch.

Zustandsübersicht

Der Baustein besitzt keine Zustandsmaschine. Die Ausgangssignale folgen direkt dem Eingangssignal. Eine grafische Zustandsübersicht entfällt daher.

Anwendungsszenarien

- **Rückkopplung von AUS-Signalen**: Ein zentrales „Aus“-Kommando einer Steuerung soll gleichzeitig mehrere Aktoren oder Subsysteme abschalten.
- **Redundante Überwachung**: Verteilen eines AUS-Signals an mehrere Überwachungseinheiten, die unabhängig voneinander auf den Befehl reagieren müssen.
- **Modulare Maschinenstruktur**: In einer modularen Anlage wird ein einmal erfasstes AUS-Signal über Busse oder Koppelelemente zu mehreren Modulen geführt. `AUS_SPLIT_4_UNGATED` ersetzt aufwändige Verdrahtung oder logische ODER-Verknüpfungen.
- **Test und Simulation**: Zum gleichzeitigen Ansteuern mehrerer simulierter Komponenten mit demselben Signal.

Vergleich mit ähnlichen Bausteinen

Baustein	Funktion	Besonderheit

AUS_SPLIT_2	Verteilt ein AUS-Signal auf zwei Ausgänge	Weniger Ports, kompakter.
AUS_SPLIT_4_UNGATED	Verteilt auf vier Ausgänge	Dieses FB.
AUS_SPLIT_N	Parametrierbarer Splitter (z. B. durch generische Adapterlisten)	Flexiblere Anzahl, aber aufwändiger in der Konfiguration.
AUS_MERGE	Fasst mehrere AUS-Eingänge zu einem Ausgang zusammen	Gegenstück zum Splitter.

Der AUS_SPLIT_4_UNGATED liegt in der Mitte zwischen einem einfachen 2-fach-Splitter und einem vollparametrierbaren Splitter. Er ist ideal, wenn exakt vier Ausgänge benötigt werden – ohne zusätzliche Konfiguration.

- **AUS_SPLIT_4_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- **AUS_SPLIT_4**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

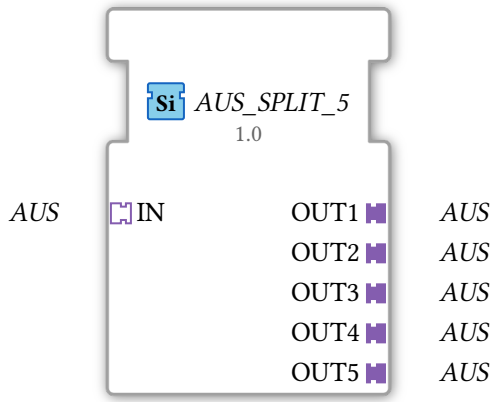
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der AUS_SPLIT_4_UNGATED ist ein einfacher, aber nützlicher generischer Funktionsblock zur Verteilung eines AUS-Signals auf vier parallele Pfade. Seine adapterbasierte Schnittstelle und die klare Trennung von Logik und Signalweitergabe machen ihn zu einem robusten Baustein in der IEC 61499-basierten Automatisierung. Dank der generischen Deklaration kann er problemlos in verschiedene Projekte eingebunden und ggf. an eigene Signaltypen angepasst werden.

AUS_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AUS_SPLIT_5 verteilt ein eingehendes AUS-Ereignis (z. B. ein Signal oder eine Nachricht) auf fünf identische Ausgänge. Es handelt sich um einen generischen Baustein, der in der 4diac-IDE als Platzhalter für einen anwendungsspezifischen Typ (GEN_AUS_SPLIT) dient.

Schnittstellenstruktur

Ereignis-Eingänge

Keine (Signalaustausch erfolgt ausschließlich über Adapter).

Ereignis-Ausgänge

Keine (Signalaustausch erfolgt ausschließlich über Adapter).

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ
Socket (Eingang)	`IN`	`adapter::types::unidirectional::AUS`
Plug (Ausgang)	`OUT1`	`adapter::types::unidirectional::AUS`
Plug (Ausgang)	`OUT2`	`adapter::types::unidirectional::AUS`
Plug (Ausgang)	`OUT3`	`adapter::types::unidirectional::AUS`
Plug (Ausgang)	`OUT4`	`adapter::types::unidirectional::AUS`
Plug (Ausgang)	`OUT5`	`adapter::types::unidirectional::AUS`

Funktionsweise

Der Baustein empfängt über den Socket `IN` ein unidirektionales AUS-Ereignis (z. B. einen Impuls oder eine logische 1) und leitet dieses ohne Verzögerung oder Verarbeitung an alle fünf angeschlossenen Plugs (`OUT1` bis `OUT5`) weiter. Jedes ausgehende Ereignis ist eine exakte Kopie des eingehenden Ereignisses.

Technische Besonderheiten

- **Generischer Typ**: Der FB wird durch das Attribut `GenericClassName = 'GEN_AUS_SPLIT'` als generischer Platzhalter definiert. Bei der Instanziierung kann der tatsächliche Adaptertyp (z. B. spezifische Ereignissignaturen) ersetzt werden.
- **Keine Zustände**: Der Baustein besitzt keine interne Logik oder Zustandsmaschine – die Weiterleitung erfolgt rein strukturell.
- **Unidirektionale Kommunikation**: Alle Adapter sind als `unidirectional::AUS` deklariert, d. h. Daten fließen nur vom Eingang zu den Ausgängen.

Zustandsübersicht

Der Baustein hat keinen internen Zustand. Er arbeitet deterministisch und durchgängig in einem passiven Modus.

Anwendungsszenarien

- **Signalverteilung**: Ein zentrales Ereignis (z. B. „Start“ oder „Alarm“) soll parallel mehrere Komponenten ansteuern.
- **Bus-Topologien**: Ersatz für mehrfaches manuelles Verkabeln eines Signals in der IEC-61499-Umgebung.
- **Prototyping**: Schnelle Aufteilung einer Ereignisquelle auf fünf Zielbausteine während der Entwicklungsphase.

Vergleich mit ähnlichen Bausteinen

- **AUS_SPLIT_2 / AUS_SPLIT_3** – Bausteine mit gleicher Funktionalität, aber unterschiedlicher Anzahl an Ausgängen (2 bzw. 3).
- **E_SPLIT** – Ein Standard-Ereignis-Split-Baustein, der Ereignis-Eingänge und -Ausgänge statt Adapter nutzt. `AUS_SPLIT_5` ist adapterbasiert und daher flexibler in der Wiederverwendung mit verschiedenen Protokollen.
- `AUS_SPLIT_5_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs

unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

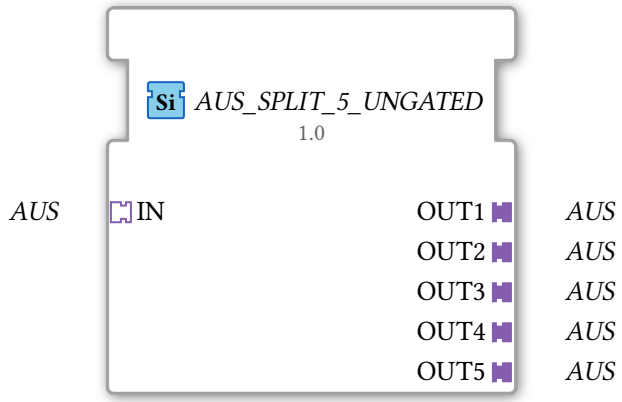
`AUS_SPLIT_5` ist ein einfacher, generischer Verteilerbaustein für unidirektionale AUS-Signale. Er vereinfacht die modulare Steuerungslogik, indem er eine 1:5-Aufteilung ohne zusätzliche Logik realisiert. Dank der Adapter-Schnittstelle kann er in unterschiedlichen Kontexten (z. B. Ereignis-, Daten- oder gemischten Strömen) eingesetzt werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUS_SPLIT_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUS_SPLIT_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AUS_SPLIT_5_UNGATED` verteilt ein eingehendes AUS-Ereignis (z. B. ein Signal oder eine Nachricht) auf fünf identische Ausgänge. Es handelt sich um einen generischen Baustein, der in der 4diac-IDE als Platzhalter für einen anwendungsspezifischen Typ (`GEN_AUS_SPLIT`) dient.

Schnittstellenstruktur

Ereignis-Eingänge

Keine (Signalaustausch erfolgt ausschließlich über Adapter).

Ereignis-Ausgänge

Keine (Signalaustausch erfolgt ausschließlich über Adapter).

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ
Socket (Eingang)	`IN`	`adapter::types::unidirectional::AUS`
Plug (Ausgang)	`OUT1`	`adapter::types::unidirectional::AUS`
Plug (Ausgang)	`OUT2`	`adapter::types::unidirectional::AUS`
Plug (Ausgang)	`OUT3`	`adapter::types::unidirectional::AUS`
Plug (Ausgang)	`OUT4`	`adapter::types::unidirectional::AUS`
Plug (Ausgang)	`OUT5`	`adapter::types::unidirectional::AUS`

Funktionsweise

Der Baustein empfängt über den Socket `IN` ein unidirektionales AUS-Ereignis (z. B. einen Impuls oder eine logische 1) und leitet dieses ohne Verzögerung oder Verarbeitung an alle fünf angeschlossenen Plugs (`OUT1` bis `OUT5`) weiter. Jedes ausgehende Ereignis ist eine exakte Kopie des eingehenden Ereignisses.

Technische Besonderheiten

- **Generischer Typ**: Der FB wird durch das Attribut `GenericClassName = 'GEN_AUS_SPLIT'` als generischer Platzhalter definiert. Bei der Instanziierung kann der tatsächliche Adaptertyp (z. B. spezifische Ereignissignaturen) ersetzt werden.
- **Keine Zustände**: Der Baustein besitzt keine interne Logik oder Zustandsmaschine – die Weiterleitung erfolgt rein strukturell.
- **Unidirektionale Kommunikation**: Alle Adapter sind als `unidirectional::AUS` deklariert, d. h. Daten fließen nur vom Eingang zu den Ausgängen.

Zustandsübersicht

Der Baustein hat keinen internen Zustand. Er arbeitet deterministisch und durchgängig in einem passiven Modus.

Anwendungsszenarien

- **Signalverteilung**: Ein zentrales Ereignis (z. B. „Start“ oder „Alarm“) soll parallel mehrere Komponenten ansteuern.
- **Bus-Topologien**: Ersatz für mehrfaches manuelles Verkabeln eines Signals in der IEC-61499-Umgebung.
- **Prototyping**: Schnelle Aufteilung einer Ereignisquelle auf fünf Zielbausteine während der Entwicklungsphase.

Vergleich mit ähnlichen Bausteinen

- **AUS_SPLIT_2 / AUS_SPLIT_3** – Bausteine mit gleicher Funktionalität, aber unterschiedlicher Anzahl an Ausgängen (2 bzw. 3).
- **E_SPLIT** – Ein Standard-Ereignis-Split-Baustein, der Ereignis-Eingänge und -Ausgänge statt Adapter nutzt. `AUS_SPLIT_5_UNGATED` ist adapterbasiert und daher flexibler in der Wiederverwendung mit verschiedenen Protokollen.
- `AUS_SPLIT_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

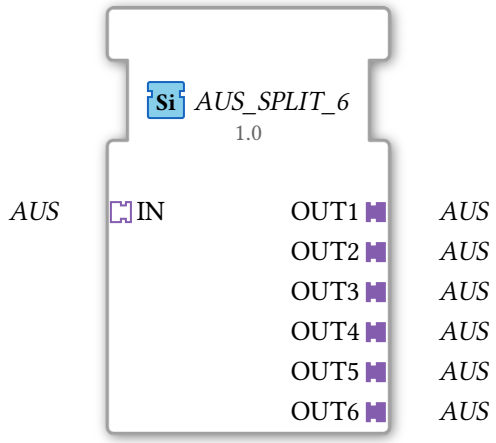
Fazit

`AUS_SPLIT_5_UNGATED` ist ein einfacher, generischer Verteilerbaustein für unidirektionale AUS-Signale. Er vereinfacht die modulare Steuerungslogik, indem er eine 1:5-Aufteilung ohne zusätzliche Logik realisiert. Dank der Adapter-Schnittstelle kann er in unterschiedlichen Kontexten (z. B. Ereignis-, Daten- oder gemischten Strömen) eingesetzt werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUS_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AUS_SPLIT_6** ist ein generischer Baustein, der ein eingehendes AUS-Signal (z. B. ein einheitliches Steuersignal oder Ereignis) auf sechs identische Ausgänge verteilt. Er dient als Splitter in der IEC 61499–Kommunikation über Adapter und ermöglicht eine einfache Mehrfachnutzung eines Signals ohne zusätzliche Logik.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine eigenständigen Ereigniseingänge vorhanden. Die gesamte Kommunikation erfolgt über den integrierten Adapter-Socket.

Ereignis-Ausgänge

- Keine eigenständigen Ereignisausgänge vorhanden. Die Ausgaben werden ausschließlich über die Adapter-Plugs bereitgestellt.

Daten-Eingänge

- Keine direkten Dateneingänge. Die Datenübergabe wird indirekt über den Adapter-Socket realisiert.

Daten-Ausgänge

- Keine direkten Datenausgänge. Die Ausgabedaten werden über die Adapter-Plugs zur Verfügung gestellt.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AUS`	**IN**	Socket	Eingehendes AUS-Signal, das auf alle Ausgänge verteilt wird.
`adapter::types::unidirectional::AUS`	**OUT1**	Plug	Erster Ausgang – identisch zu **IN**.
`adapter::types::unidirectional::AUS`	**OUT2**	Plug	Zweiter Ausgang – identisch zu **IN**.
`adapter::types::unidirectional::AUS`	**OUT3**	Plug	Dritter Ausgang – identisch zu **IN**.
`adapter::types::unidirectional::AUS`	**OUT4**	Plug	Vierter Ausgang – identisch zu **IN**.
`adapter::types::unidirectional::AUS**	**OUT5**	Plug	Fünfter Ausgang – identisch zu **IN**.

`adapter::types::unidirectional::AUS`	**OUT6**	Plug	Sechster Ausgang – identisch zu **IN**.

Hinweis: Der Adapter-Typ „AUS“ ist ein unidirektionaler Adapter, der ein Signal (Ereignis + ggf. Daten) in einer Richtung weitergibt.

Funktionsweise

Der Baustein fungiert als passiver Verteiler. Das Signal, das am Socket **IN** anliegt, wird unverändert und ohne zeitliche Verzögerung auf alle sechs Plugs **OUT1** bis **OUT6** kopiert. Es findet keine Filterung, Transformation oder Pufferung statt. Der Baustein verhält sich wie ein „Fan-Out“-Knoten für AUS-Adapterverbindungen.

Technische Besonderheiten

- **Generischer Typ**: Der Baustein ist als generischer Funktionsbaustein (GenericClassName = `'GEN_AUS_SPLIT'`) realisiert. In der 4diac-IDE kann er je nach konkretem AUS-Adaptertyp instanziiert werden.
- **Abhängigkeiten**: Erfordert den Adapter `adapter::types::unidirectional::AUS` aus dem Paket `adaper::events::unidirectional`. Die Implementierung importiert die Eclipse 4diac-Basistypen `GenericClassName` und `TypeHash`.
- **Kein Zustand**: Der Baustein besitzt keinen internen Zustandsautomaten (keine ECC). Er arbeitet rein kombinatorisch – jede Änderung am Eingang wird sofort auf alle Ausgänge durchgereicht.
- **Unidirektional**: Die Adapter sind ausschließlich für eine Signalrichtung ausgelegt. Rückwirkungen von den Ausgängen auf den Eingang sind nicht möglich.

Zustandsübersicht

Der Baustein besitzt **keine expliziten Zustände**. Sein Verhalten ergibt sich ausschließlich aus der aktuellen Belegung des Eingangs-Adapters. Eine Zustandsänderung im Sinne eines Automaten findet nicht statt.

Anwendungsszenarien

- **Verteilen eines „AUS“-Steuersignals**: In einer Anlage soll ein einzelnes Stopp- oder Ausschaltsignal gleichzeitig an mehrere Aktoren (Motoren, Ventile, etc.) weitergeleitet werden.
- **Redundante Signalpfade**: Ein Signal wird auf mehrere parallele Pfade aufgeteilt, um in späteren Verarbeitungsschritten verschiedene Aktionen auszulösen.
- **Test- und Simulationsumgebungen**: Ein generiertes Testsignal wird unverändert an verschiedene Überwachungs- oder Logikbausteine gesendet.

Vergleich mit ähnlichen Bausteinen

- **Ereignis-Splitter (z. B. `E_SPLIT`)**: Verarbeitet diskrete Ereignisse an eigenen Eingangs-/Ausgangs-Events. Der `AUS_SPLIT_6` arbeitet dagegen auf Adapterebene und verteilt sowohl Ereignis als auch zugehörige Daten in einem Zug.
- **Datensplitter**: Reine Datenverteiler (z. B. `F_MUX`, `F_DIST`) erfordern separate Datentypen. Der vorliegende Baustein ist spezifisch für den Adaptertyp `AUS` und kapselt die Signalstruktur.
- **Generic-Fähigkeit**: Dank der generischen Deklaration kann der Baustein in verschiedenen Kontexten mit unterschiedlichen AUS-Adapterimplementierungen wiederverwendet werden.
- **`AUS_SPLIT_6_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

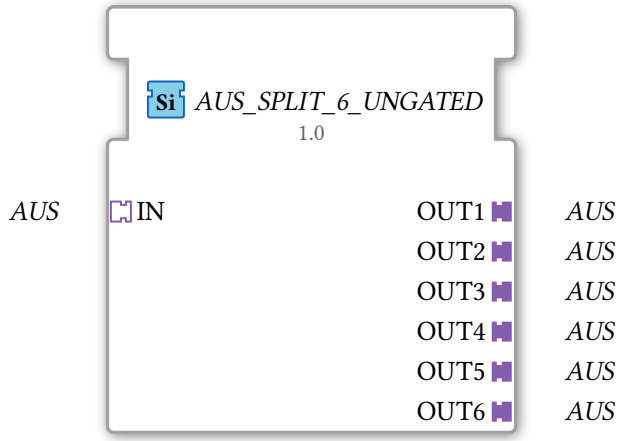
Der **AUS_SPLIT_6** ist ein einfacher, aber nützlicher generischer Funktionsbaustein zur Signalverteilung in IEC 61499-Anwendungen. Er reduziert den Verdrahtungsaufwand, indem er ein einzelnes AUS-Signal auf sechs parallele Ausgänge bringt. Seine generische Natur macht ihn flexibel einsetzbar, solange der verwendete AUS-Adapter den unidirektionalen Vertrag einhält.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUS_SPLIT_6_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUS_SPLIT_6`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AUS_SPLIT_6_UNGATED** ist ein generischer Baustein, der ein eingehendes AUS-Signal (z. B. ein einheitliches Steuersignal oder Ereignis) auf sechs identische Ausgänge verteilt. Er dient als Splitter in der IEC 61499–Kommunikation über Adapter und ermöglicht eine einfache Mehrfachnutzung eines Signals ohne zusätzliche Logik.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine eigenständigen Ereigniseingänge vorhanden. Die gesamte Kommunikation erfolgt über den integrierten Adapter-Socket.

Ereignis-Ausgänge

- Keine eigenständigen Ereignisausgänge vorhanden. Die Ausgaben werden ausschließlich über die Adapter-Plugs bereitgestellt.

Daten-Eingänge

- Keine direkten Dateneingänge. Die Datenübergabe wird indirekt über den Adapter-Socket realisiert.

Daten-Ausgänge

- Keine direkten Datenausgänge. Die Ausgabedaten werden über die Adapter-Plugs zur Verfügung gestellt.

Adapter

Typ	Name	Richtung	Beschreibung

`adapter::types::unidirectional::AUS`	**IN**	Socket	Eingehendes AUS-Signal, das auf alle Ausgänge verteilt wird.
`adapter::types::unidirectional::AUS`	**OUT1**	Plug	Erster Ausgang – identisch zu **IN**.
`adapter::types::unidirectional::AUS`	**OUT2**	Plug	Zweiter Ausgang – identisch zu **IN**.
`adapter::types::unidirectional::AUS`	**OUT3**	Plug	Dritter Ausgang – identisch zu **IN**.
`adapter::types::unidirectional::AUS`	**OUT4**	Plug	Vierter Ausgang – identisch zu **IN**.
`adapter::types::unidirectional::AUS**	**OUT5**	Plug	Fünfter Ausgang – identisch zu **IN**.
`adapter::types::unidirectional::AUS`	**OUT6**	Plug	Sechster Ausgang – identisch zu **IN**.

Hinweis: Der Adapter-Typ „AUS“ ist ein unidirektionaler Adapter, der ein Signal (Ereignis + ggf. Daten) in einer Richtung weitergibt.

Funktionsweise

Der Baustein fungiert als passiver Verteiler. Das Signal, das am Socket **IN** anliegt, wird unverändert und ohne zeitliche Verzögerung auf alle sechs Plugs **OUT1** bis **OUT6** kopiert. Es findet keine Filterung, Transformation oder Pufferung statt. Der Baustein verhält sich wie ein „Fan-Out“-Knoten für AUS-Adapterverbindungen.

Technische Besonderheiten

- **Generischer Typ**: Der Baustein ist als generischer Funktionsbaustein (GenericClassName = `'GEN_AUS_SPLIT'`) realisiert. In der 4diac-IDE kann er je nach konkretem AUS-Adaptertyp instanziiert werden.
- **Abhängigkeiten**: Erfordert den Adapter `adapter::types::unidirectional::AUS` aus dem Paket `adaper::events::unidirectional`. Die Implementierung importiert die Eclipse 4diac-Basistypen `GenericClassName` und `TypeHash`.
- **Kein Zustand**: Der Baustein besitzt keinen internen Zustandsautomaten (keine ECC). Er arbeitet rein kombinatorisch – jede Änderung am Eingang wird sofort auf alle Ausgänge durchgereicht.
- **Unidirektional**: Die Adapter sind ausschließlich für eine Signalrichtung ausgelegt. Rückwirkungen von den Ausgängen auf den Eingang sind nicht möglich.

Zustandsübersicht

Der Baustein besitzt **keine expliziten Zustände**. Sein Verhalten ergibt sich ausschließlich aus der aktuellen Belegung des Eingangs-Adapters. Eine Zustandsänderung im Sinne eines Automaten findet nicht statt.

Anwendungsszenarien

- **Verteilen eines „AUS“-Steuersignals**: In einer Anlage soll ein einzelnes Stopp- oder Ausschaltsignal gleichzeitig an mehrere Aktoren (Motoren, Ventile, etc.) weitergeleitet werden.
- **Redundante Signalpfade**: Ein Signal wird auf mehrere parallele Pfade aufgeteilt, um in späteren Verarbeitungsschritten verschiedene Aktionen auszulösen.
- **Test- und Simulationsumgebungen**: Ein generiertes Testsignal wird unverändert an verschiedene Überwachungs- oder Logikbausteine gesendet.

Vergleich mit ähnlichen Bausteinen

- **Ereignis-Splitter (z. B. `E_SPLIT`)**: Verarbeitet diskrete Ereignisse an eigenen Eingangs-/Ausgangs-Events. Der `AUS_SPLIT_6_UNGATED` arbeitet dagegen auf Adapterebene und verteilt sowohl Ereignis als auch zugehörige Daten in einem Zug.
- **Datensplitter**: Reine Datenverteiler (z. B. `F_MUX`, `F_DIST`) erfordern separate Datentypen. Der vorliegende Baustein ist spezifisch für den Adaptertyp `AUS` und kapselt die Signalstruktur.
- **Generic-Fähigkeit**: Dank der generischen Deklaration kann der Baustein in verschiedenen Kontexten mit unterschiedlichen AUS-Adapterimplementierungen wiederverwendet werden.
- **`AUS_SPLIT_6`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

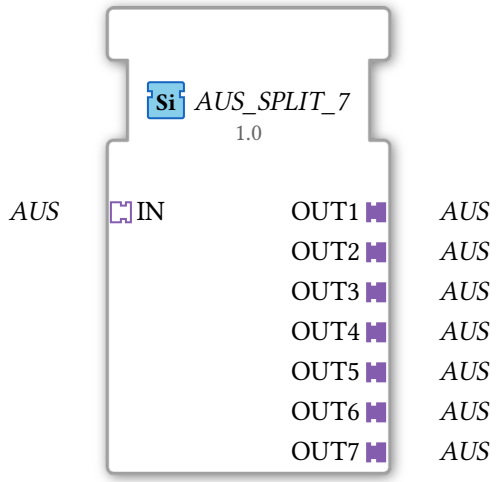
Fazit

Der **AUS_SPLIT_6_UNGATED** ist ein einfacher, aber nützlicher generischer Funktionsbaustein zur Signalverteilung in IEC 61499-Anwendungen. Er reduziert den Verdrahtungsaufwand, indem er ein einzelnes AUS-Signal auf sechs parallele Ausgänge bringt. Seine generische Natur macht ihn flexibel einsetzbar, solange der verwendete AUS-Adapter den unidirektionalen Vertrag einhält.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUS_SPLIT_7



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUS_SPLIT_7** ist ein generischer Baustein zur Verteilung eines einzelnen AUS-Adapter-Signals auf sieben identische AUS-Ausgänge. Er dient als Signalvervielfacher in Steuerungsanwendungen, die auf der IEC 61499-Norm basieren. Der Baustein besitzt keine ereignis- oder datenbasierten Schnittstellen, sondern kommuniziert ausschließlich über Adapter des Typs `adapter::types::unidirectional::AUS`. Die eigentliche Typisierung des Signals wird erst zur Laufzeit über das generische Attribut `GenericClassName` festgelegt.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	`IN`	`adapter::types::unidirectional::AUS`	Einheitliches AUS-Eingangssignal
Plug (Ausgang)	`OUT1` – `OUT7`	`adapter::types::unidirectional::AUS`	Sieben parallel ausgegebene AUS-Signale

Funktionsweise

Der Baustein empfängt über den Adapter-Socket `IN` ein AUS-Signal. Dieses Signal wird unverändert und ohne Verzögerung an alle sieben Adapter-Plugs `OUT1` bis `OUT7` weitergeleitet. Da der Baustein als generischer Funktionsblock deklariert ist (`GenericClassName = 'GEN_AUS_SPLIT'`), kann der konkrete Typ des AUS-Adapters zur Entwurfszeit oder während der Systemkonfiguration überschrieben werden. Das Verhalten bleibt dabei stets dasselbe: 1-zu-n-Verteilung.

Technische Besonderheiten

- **Generischer Baustein**: Der tatsächliche Adaptertyp wird über das Attribut `eclipse4diac::core::GenericClassName` definiert. Standardmäßig ist `'GEN_AUS_SPLIT'` gesetzt.
- **Keine ECC / Zustandsmaschine**: Aufgrund des Fehlens von Ereigniseingängen besitzt der Baustein keinen internen Zustandsautomaten. Die Ausgänge folgen direkt dem Eingangssignal (kombinatorisches Verhalten).
- **Reine Adapterkommunikation**: Es werden keine IEC 61499-Datentypen (BOOL, INT, etc.) verwendet; die gesamte Datenübertragung erfolgt über unidirektionale AUS-Adapter.
- **Kompatibilität**: Der Baustein ist für den Einsatz in 4diac-IDE und Laufzeitumgebungen vorgesehen, die Adapter des Typs `adapter::types::unidirectional::AUS` unterstützen.

Zustandsübersicht

Der Funktionsblock verfügt über keinen Zustandsautomaten. Das Ausgangssignal jeder der sieben OUT-Adapter entspricht stets dem aktuellen Wert des IN-Adapters. Eine zeitliche Verzögerung oder Zustandsspeicherung findet nicht statt.

Anwendungsszenarien

- **Signalverteilung in redundanten Systemen**: Ein zentrales AUS-Steuersignal (z. B. „Not-Aus“) soll an mehrere dezentrale Aktoren oder Subsysteme weitergeleitet werden.
- **Modulare Automatisierungsstrukturen**: Ein von einer übergeordneten Steuerung empfangenes AUS-Signal wird auf sieben parallele Pfade aufgeteilt, die unabhängige Maschinenmodule steuern.
- **Test- und Simulationsumgebungen**: Zur einfachen Vervielfachung eines Testsignals ohne Änderung der Logik.

Vergleich mit ähnlichen Bausteinen

- **AUS_SPLIT_N** (z. B. für 2, 4, 8 Ausgänge) existieren als Varianten mit unterschiedlicher Ausgangsanzahl. AUS_SPLIT_7 ist eine spezifische Ausprägung für genau sieben Ausgänge.
- Im Gegensatz zu Daten-Splittern (z. B. `SPLIT_INT`, `SPLIT_BOOL`) arbeitet dieser Baustein ausschließlich mit Adaptern und nicht mit elementaren Datentypen. Dadurch eignet er sich besonders für protokollbasierte oder typsichere Verbindungen.
- Bausteine mit Ereignis-Schnittstellen (z. B. `E_SPLIT`) verteilen Ereignisse, nicht Signale. AUS_SPLIT_7 verteilt dagegen einen kontinuierlichen Signalwert.
- `AUS_SPLIT_7_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

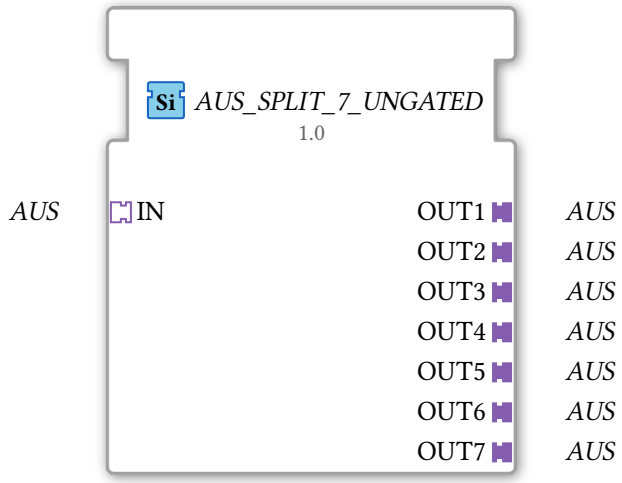
Der **AUS_SPLIT_7** ist ein einfacher, aber effektiver generischer Funktionsblock zur unidirektionalen Verteilung eines AUS-Adapter-Signals auf sieben parallele Ausgänge. Durch seine reine Adapter-basierte Schnittstelle und die fehlende Ereignislogik eignet er sich für alle Anwendungen, bei denen ein Signal ohne Verzögerung oder Zustandsspeicherung vervielfacht werden muss. Er bietet eine saubere, modulare Lösung für die Signalverteilung in IEC 61499-basierten Automatisierungssystemen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUS_SPLIT_7_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUS_SPLIT_7`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUS_SPLIT_7_UNGATED** ist ein generischer Baustein zur Verteilung eines einzelnen AUS-Adapter-Signals auf sieben identische AUS-Ausgänge. Er dient als Signalvervielfacher in Steuerungsanwendungen, die auf der IEC 61499-Norm basieren. Der Baustein besitzt keine ereignis- oder datenbasierten Schnittstellen, sondern kommuniziert ausschließlich über Adapter des Typs `adapter::types::unidirectional::AUS`. Die eigentliche Typisierung des Signals wird erst zur Laufzeit über das generische Attribut `GenericClassName` festgelegt.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	`IN`	`adapter::types::unidirectional::AUS`	Einheitliches AUS-Eingangssignal
Plug (Ausgang)	`OUT1` – `OUT7`	`adapter::types::unidirectional::AUS`	Sieben parallel ausgegebene AUS-Signale

Funktionsweise

Der Baustein empfängt über den Adapter-Socket `IN` ein AUS-Signal. Dieses Signal wird unverändert und ohne Verzögerung an alle sieben Adapter-Plugs `OUT1` bis `OUT7` weitergeleitet. Da der Baustein als generischer Funktionsblock deklariert ist (`GenericClassName = 'GEN_AUS_SPLIT'`), kann der konkrete Typ des AUS-Adapters zur Entwurfszeit oder während der Systemkonfiguration überschrieben werden. Das Verhalten bleibt dabei stets dasselbe: 1-zu-n-Verteilung.

Technische Besonderheiten

- **Generischer Baustein**: Der tatsächliche Adaptertyp wird über das Attribut `eclipse4diac::core::GenericClassName` definiert. Standardmäßig ist `'GEN_AUS_SPLIT'` gesetzt.
- **Keine ECC / Zustandsmaschine**: Aufgrund des Fehlens von Ereigniseingängen besitzt der Baustein keinen internen Zustandsautomaten. Die Ausgänge folgen direkt dem Eingangssignal (kombinatorisches Verhalten).
- **Reine Adapterkommunikation**: Es werden keine IEC 61499-Datentypen (BOOL, INT, etc.) verwendet; die gesamte Datenübertragung erfolgt über unidirektionale AUS-Adapter.
- **Kompatibilität**: Der Baustein ist für den Einsatz in 4diac-IDE und Laufzeitumgebungen vorgesehen, die Adapter des Typs `adapter::types::unidirectional::AUS` unterstützen.

Zustandsübersicht

Der Funktionsblock verfügt über keinen Zustandsautomaten. Das Ausgangssignal jeder der sieben OUT-Adapter entspricht stets dem aktuellen Wert des IN-Adapters. Eine zeitliche Verzögerung oder Zustandsspeicherung findet nicht statt.

Anwendungsszenarien

- **Signalverteilung in redundanten Systemen**: Ein zentrales AUS-Steuersignal (z. B. „Not-Aus“) soll an mehrere dezentrale Aktoren oder Subsysteme weitergeleitet werden.
- **Modulare Automatisierungsstrukturen**: Ein von einer übergeordneten Steuerung empfangenes AUS-Signal wird auf sieben parallele Pfade aufgeteilt, die unabhängige Maschinenmodule steuern.
- **Test- und Simulationsumgebungen**: Zur einfachen Vervielfachung eines Testsignals ohne Änderung der Logik.

Vergleich mit ähnlichen Bausteinen

- **AUS_SPLIT_N** (z. B. für 2, 4, 8 Ausgänge) existieren als Varianten mit unterschiedlicher Ausgangsanzahl. AUS_SPLIT_7_UNGATED ist eine spezifische Ausprägung für genau sieben Ausgänge.
- Im Gegensatz zu Daten-Splittern (z. B. `SPLIT_INT`, `SPLIT_BOOL`) arbeitet dieser Baustein ausschließlich mit Adaptern und nicht mit elementaren Datentypen. Dadurch eignet er sich besonders für protokollbasierte oder typsichere Verbindungen.
- Bausteine mit Ereignis-Schnittstellen (z. B. `E_SPLIT`) verteilen Ereignisse, nicht Signale. AUS_SPLIT_7_UNGATED verteilt dagegen einen kontinuierlichen Signalwert.
- `AUS_SPLIT_7`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

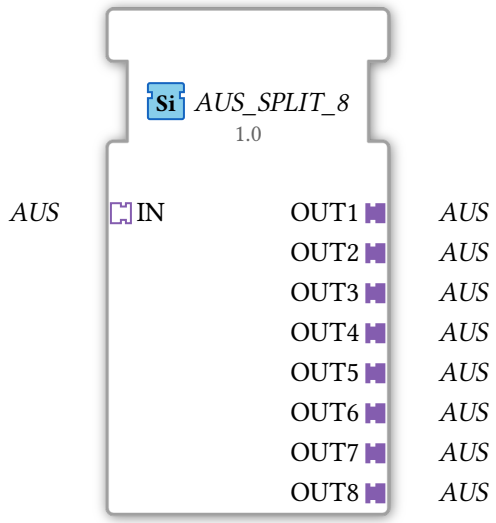
Fazit

Der **AUS_SPLIT_7_UNGATED** ist ein einfacher, aber effektiver generischer Funktionsblock zur unidirektionalen Verteilung eines AUS-Adapter-Signals auf sieben parallele Ausgänge. Durch seine reine Adapter-basierte Schnittstelle und die fehlende Ereignislogik eignet er sich für alle Anwendungen, bei denen ein Signal ohne Verzögerung oder Zustandsspeicherung vervielfacht werden muss. Er bietet eine saubere, modulare Lösung für die Signalverteilung in IEC 61499-basierten Automatisierungssystemen.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUS_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AUS_SPLIT_8** ist ein generischer Funktionsbaustein zur Verteilung eines eingehenden AUS-Signals auf bis zu acht separate Ausgänge. Er dient als einfacher „Splitter“ für Signale vom Typ **adapter::types::unidirectional::AUS** und ermöglicht die parallele Ansteuerung mehrerer Verbraucher oder nachfolgender Bausteine.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Schnittstelle	Typ	Richtung	Beschreibung
IN	`adapter::types::unidirectional::AUS`	Socket	Eingang – empfängt das zu verteilende AUS-Signal

OUT1	adapter::types::unidirectional::AUS	Plug	Ausgang 1 – gibt das eingehende Signal weiter
OUT2	adapter::types::unidirectional::AUS	Plug	Ausgang 2 – gibt das eingehende Signal weiter
OUT3	adapter::types::unidirectional::AUS	Plug	Ausgang 3 – gibt das eingehende Signal weiter
OUT4	adapter::types::unidirectional::AUS	Plug	Ausgang 4 – gibt das eingehende Signal weiter
OUT5	adapter::types::unidirectional::AUS	Plug	Ausgang 5 – gibt das eingehende Signal weiter
OUT6	adapter::types::unidirectional::AUS	Plug	Ausgang 6 – gibt das eingehende Signal weiter
OUT7	adapter::types::unidirectional::AUS	Plug	Ausgang 7 – gibt das eingehende Signal weiter
OUT8	adapter::types::unidirectional::AUS	Plug	Ausgang 8 – gibt das eingehende Signal weiter

Funktionsweise

Der Baustein leitet das am Socket **IN** anliegende AUS-Signal unverändert an alle acht Plugs **OUT1** bis **OUT8** weiter. Es findet keine Verzögerung, Filterung oder logische Verarbeitung statt. Das Signal wird auf allen Ausgängen gleichzeitig zur Verfügung gestellt. Die Funktion entspricht einer 1-zu-8-Verteilung.

Technische Besonderheiten

- **Generischer Baustein**: Der FB wird als generischer Typ (`GEN_AUS_SPLIT`) ausgeführt. Die konkrete Typisierung erfolgt zur Laufzeit über das Attribut `eclipse4diac::core::GenericClassName`.
- **Keine Zustandsmaschine**: Da der Baustein nur eine direkte Weiterleitung ohne Logik vornimmt, besitzt er keine eigene Ausführungssteuerung (ECC). Die Signalweitergabe erfolgt rein über die Adapter-Schnittstellen.
- **Adapter-Typ**: Alle Schnittstellen verwenden den unidirektionalen `AUS`-Adaptertyp, der eine standardisierte Signalübertragung ermöglicht.

Zustandsübersicht

Der Baustein besitzt keine Zustandsmaschine. Es existieren keine internen Zustände oder Übergänge.

Anwendungsszenarien

- **Verteilen eines zentralen Steuersignals**: Ein einzelnes „Aus“-Signal (z. B. Ausschaltbefehl) wird an mehrere Aktoren oder Unterbausteine weitergegeben.
- **Parallelschaltung von Verbrauchern**: In landwirtschaftlichen oder industriellen Steuerungen, wenn ein Signal gleichzeitig mehrere Ventile, Motoren oder Leuchten ansteuern soll.
- **Bussignale aufteilen**: Wenn ein Adapter-basierter Signalpfad in mehrere parallele Pfade verzweigt werden muss.

Vergleich mit ähnlichen Bausteinen

- **AUS_SPLIT_4**: Verteilt ein Signal auf vier Ausgänge. Der AUS_SPLIT_8 bietet mit acht Ausgängen eine höhere Fan-Out-Kapazität.
- **AUS_MERGE / AUS_JOIN**: Führen mehrere Signale zusammen – funktional das Gegenteil.
- **AUS_ROUTER**: Kann ein Signal wahlweise auf einen von mehreren Ausgängen leiten, während der Splitter immer alle Ausgänge aktiviert.
- `AUS_SPLIT_8_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

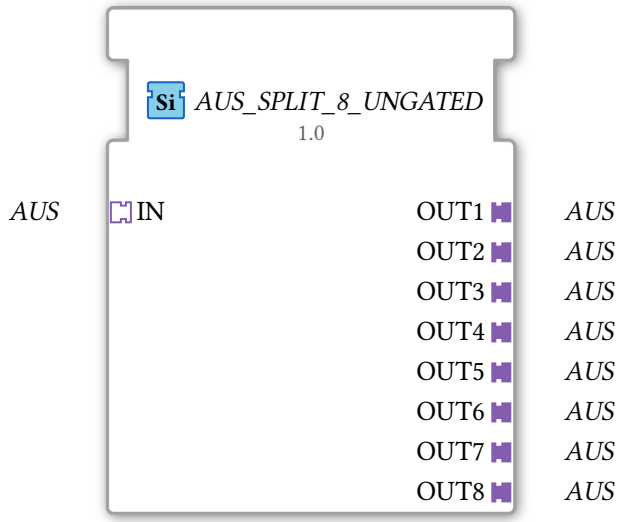
Der **AUS_SPLIT_8** ist ein einfacher, aber nützlicher Funktionsbaustein zur Vervielfachung eines AUS-Signals. Durch den Verzicht auf Logik und Zustände ist er effizient und leicht verständlich. Er eignet sich besonders für Anwendungen, in denen ein Steuersignal auf mehrere Empfänger verteilt werden muss, ohne dass eine Auswahl oder Verarbeitung erforderlich ist.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 Gesamtwiderstand in Reihen- & Parallelschaltung auf ms-muc-docs.de

AUS_SPLIT_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUS_SPLIT_8`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AUS_SPLIT_8_UNGATED** ist ein generischer Funktionsbaustein zur Verteilung eines eingehenden AUS-Signals auf bis zu acht separate Ausgänge. Er dient als einfacher „Splitter“ für Signale vom Typ **adapter::types::unidirectional::AUS** und ermöglicht die parallele Ansteuerung mehrerer Verbraucher oder nachfolgender Bausteine.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Schnittstelle	Typ	Richtung	Beschreibung
IN	`adapter::types::unidirectional::AUS`	Socket	Eingang – empfängt das zu verteilende AUS-Signal
OUT1	`adapter::types::unidirectional::AUS`	Plug	Ausgang 1 – gibt das eingehende Signal weiter
OUT2	`adapter::types::unidirectional::AUS`	Plug	Ausgang 2 – gibt das eingehende Signal weiter
OUT3	`adapter::types::unidirectional::AUS`	Plug	Ausgang 3 – gibt das eingehende Signal weiter
OUT4	`adapter::types::unidirectional::AUS`	Plug	Ausgang 4 – gibt das eingehende Signal weiter
OUT5	`adapter::types::unidirectional::AUS`	Plug	Ausgang 5 – gibt das eingehende Signal weiter
OUT6	`adapter::types::unidirectional::AUS`	Plug	Ausgang 6 – gibt das eingehende Signal weiter
OUT7	`adapter::types::unidirectional::AUS`	Plug	Ausgang 7 – gibt das eingehende Signal weiter
OUT8	`adapter::types::unidirectional::AUS`	Plug	Ausgang 8 – gibt das eingehende Signal weiter

Funktionsweise

Der Baustein leitet das am Socket **IN** anliegende AUS-Signal unverändert an alle acht Plugs **OUT1** bis **OUT8** weiter. Es findet keine Verzögerung, Filterung oder logische Verarbeitung statt. Das Signal wird auf allen Ausgängen gleichzeitig zur Verfügung gestellt. Die Funktion entspricht einer 1-zu-8-Verteilung.

Technische Besonderheiten

- **Generischer Baustein**: Der FB wird als generischer Typ (`GEN_AUS_SPLIT`) ausgeführt. Die konkrete Typisierung erfolgt zur Laufzeit über das Attribut `eclipse4diac::core::GenericClassName`.
- **Keine Zustandsmaschine**: Da der Baustein nur eine direkte Weiterleitung ohne Logik vornimmt, besitzt er keine eigene Ausführungssteuerung (ECC). Die Signalweitergabe erfolgt rein über die Adapter-Schnittstellen.
- **Adapter-Typ**: Alle Schnittstellen verwenden den unidirektionalen `AUS`-Adaptertyp, der eine standardisierte Signalübertragung ermöglicht.

Zustandsübersicht

Der Baustein besitzt keine Zustandsmaschine. Es existieren keine internen Zustände oder Übergänge.

Anwendungsszenarien

- **Verteilen eines zentralen Steuersignals**: Ein einzelnes „Aus“-Signal (z. B. Ausschaltbefehl) wird an mehrere Aktoren oder Unterbausteine weitergegeben.
- **Parallelschaltung von Verbrauchern**: In landwirtschaftlichen oder industriellen Steuerungen, wenn ein Signal gleichzeitig mehrere Ventile, Motoren oder Leuchten ansteuern soll.

- **Bussignale aufteilen**: Wenn ein Adapter-basierter Signalpfad in mehrere parallele Pfade verzweigt werden muss.

Vergleich mit ähnlichen Bausteinen

- **AUS_SPLIT_4**: Verteilt ein Signal auf vier Ausgänge. Der AUS_SPLIT_8_UNGATED bietet mit acht Ausgängen eine höhere Fan-Out-Kapazität.
- **AUS_MERGE / AUS_JOIN**: Führen mehrere Signale zusammen – funktional das Gegenteil.
- **AUS_ROUTER**: Kann ein Signal wahlweise auf einen von mehreren Ausgängen leiten, während der Splitter immer alle Ausgänge aktiviert.
- `AUS_SPLIT_8`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

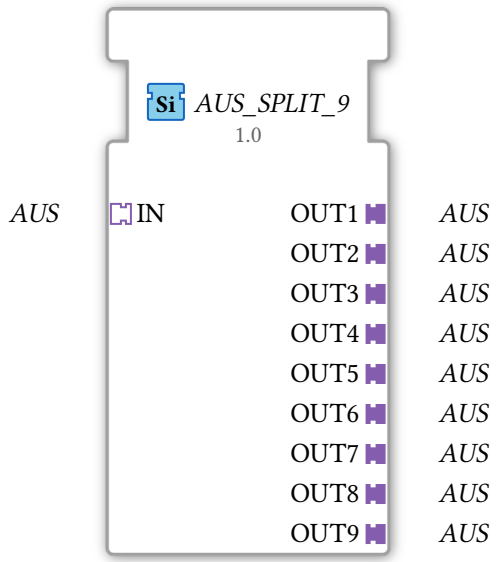
Fazit

Der **AUS_SPLIT_8_UNGATED** ist ein einfacher, aber nützlicher Funktionsbaustein zur Vervielfachung eines AUS-Signals. Durch den Verzicht auf Logik und Zustände ist er effizient und leicht verständlich. Er eignet sich besonders für Anwendungen, in denen ein Steuersignal auf mehrere Empfänger verteilt werden muss, ohne dass eine Auswahl oder Verarbeitung erforderlich ist.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 Gesamtwiderstand in Reihen- & Parallelschaltung auf ms-muc-docs.de

AUS_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUS_SPLIT_9** ist ein generischer Baustein, der ein eingehendes AUS-Signal auf neun separate Ausgänge verteilt. Er dient als Fan-out für unidirektionale Adapterverbindungen und eignet sich besonders für Anwendungen, in denen ein Signal parallel an mehrere Empfänger weitergeleitet werden muss.

Schnittstellenstruktur

Ereignis-Eingänge
Keine.

Ereignis-Ausgänge
Keine.

Daten-Eingänge
Keine.

Daten-Ausgänge
Keine.

Adapter

Richtung	Name	Typ	Beschreibung
SOCKET	IN	`adapter::types::unidirectional::AUS`	Eingehendes AUS-Signal
PLUG	OUT1 ... OUT9	`adapter::types::unidirectional::AUS`	Neun ausgehende AUS-Signale

Funktionsweise
Der Baustein leitet das am SOCKET *IN* ankommende AUS-Signal unverändert an alle neun PLUG-Ausgänge *OUT1* bis *OUT9* weiter. Die Weiterleitung erfolgt kontinuierlich ohne eigene Logik oder Verzögerung. Da es sich um einen generischen FB handelt, ist das konkrete Datenformat des AUS-Adaptertyps nicht festgelegt – es können boolesche Werte, Skalare oder komplexe Strukturen sein, je nach verwendeter Adapterdefinition.

Technische Besonderheiten
- **Generischer Baustein**: Der FB wird durch das Attribut `GenericClassName` (`GEN_AUS_SPLIT`) als generisch deklariert und kann für verschiedene AUS-Adapterinstanzen parametrisiert werden.
- **Adapterbasiert**: Die Kommunikation erfolgt ausschließlich über Adapter (Sockets/Plugs) – es sind keine Ereignis- oder Dateneingänge erforderlich.
- **Unidirektional**: Der verwendete Adaptertyp `adapter::types::unidirectional::AUS` unterstützt nur eine Datenflussrichtung (vom SOCKET zu den PLUGs).

Zustandsübersicht
Der Baustein besitzt keine eigenen Zustände oder eine Ereignisablaufsteuerung (ECC). Die Signalweitergabe erfolgt rein datengetrieben über die Adapterverbindungen.

Anwendungsszenarien
- **Signalverteilung**: Ein zentrales Steuersignal (z.B. „AUS“ für eine Maschine) soll an mehrere Aktoren oder Subsysteme gleichzeitig gesendet werden.
- **Redundanz**: Parallele Ausgänge für redundante Steuerpfade.
- **Test- und Simulationsaufbauten**: Ein Simulationssignal wird auf mehrere Beobachter oder Protokollierungsblöcke verteilt.

Vergleich mit ähnlichen Bausteinen

Baustein	Typ	Ausgänge	Besonderheit
AUS_SPLIT_9	Adapter	9	Generisch, für unidirektionale AUS-Adapter
AUS_SPLIT_4	Adapter	4	Geringere Ausgangsanzahl
SPLIT_1_2 (z.B. für Daten)	Daten	beliebig	Arbeitet mit Daten-Ereignis-Kombinationen

Im Gegensatz zu datenbasierten Split-Bausteinen benötigt AUS_SPLIT_9 keine Ereignissteuerung, da die Adapterverbindung die Datenweitergabe implizit abwickelt.

- `AUS_SPLIT_9_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

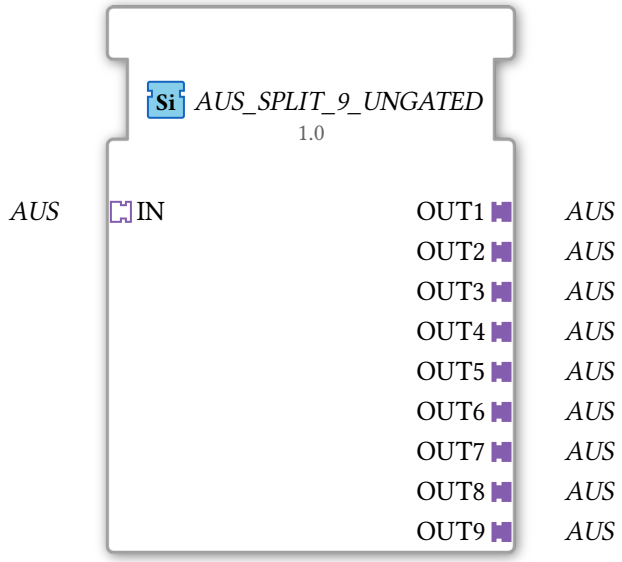
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der **AUS_SPLIT_9** ist ein einfacher, aber effektiver Funktionsblock zur Aufteilung eines einzelnen AUS-Signals auf neun parallele Pfade. Seine generische Natur und die reine Adapter-Schnittstelle machen ihn flexibel einsetzbar in IEC 61499-basierten Steuerungsanwendungen, die eine unidirektionale Signalvervielfältigung erfordern.

AUS_SPLIT_9_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUS_SPLIT_9`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUS_SPLIT_9_UNGATED** ist ein generischer Baustein, der ein eingehendes AUS-Signal auf neun separate Ausgänge verteilt. Er dient als Fan-out für unidirektionale Adapterverbindungen und eignet sich besonders für Anwendungen, in denen ein Signal parallel an mehrere Empfänger weitergeleitet werden muss.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
SOCKET	IN	`adapter::types::unidirectional::AUS`	Eingehendes AUS-Signal
PLUG	OUT1 ... OUT9	`adapter::types::unidirectional::AUS`	Neun ausgehende AUS-Signale

Funktionsweise

Der Baustein leitet das am SOCKET *IN* ankommende AUS-Signal unverändert an alle neun PLUG-Ausgänge *OUT1* bis *OUT9* weiter. Die Weiterleitung erfolgt kontinuierlich ohne eigene Logik oder Verzögerung. Da es sich um einen generischen FB handelt, ist das konkrete Datenformat des AUS-Adaptertyps nicht festgelegt – es können boolesche Werte, Skalare oder komplexe Strukturen sein, je nach verwendeter Adapterdefinition.

Technische Besonderheiten

- **Generischer Baustein**: Der FB wird durch das Attribut `GenericClassName` (`GEN_AUS_SPLIT`) als generisch deklariert und kann für verschiedene AUS-Adapterinstanzen parametrisiert werden.
- **Adapterbasiert**: Die Kommunikation erfolgt ausschließlich über Adapter (Sockets/Plugs) – es sind keine Ereignis- oder Dateneingänge erforderlich.
- **Unidirektional**: Der verwendete Adaptertyp `adapter::types::unidirectional::AUS` unterstützt nur eine Datenflussrichtung (vom SOCKET zu den PLUGs).

Zustandsübersicht

Der Baustein besitzt keine eigenen Zustände oder eine Ereignisablaufsteuerung (ECC). Die Signalweitergabe erfolgt rein datengetrieben über die Adapterverbindungen.

Anwendungsszenarien

- **Signalverteilung**: Ein zentrales Steuersignal (z.B. „AUS“ für eine Maschine) soll an mehrere Aktoren oder Subsysteme gleichzeitig gesendet werden.
- **Redundanz**: Parallele Ausgänge für redundante Steuerpfade.
- **Test- und Simulationsaufbauten**: Ein Simulationssignal wird auf mehrere Beobachter oder Protokollierungsblöcke verteilt.

Vergleich mit ähnlichen Bausteinen

Baustein	Typ	Ausgänge	Besonderheit
AUS_SPLIT_9_UNGATED	Adapter	9	Generisch, für unidirektionale AUS-Adapter
AUS_SPLIT_4	Adapter	4	Geringere Ausgangsanzahl
SPLIT_1_2 (z.B. für Daten)	Daten	beliebig	Arbeitet mit Daten-Ereignis-Kombinationen

Im Gegensatz zu datenbasierten Split-Bausteinen benötigt AUS_SPLIT_9_UNGATED keine Ereignissteuerung, da die Adapterverbindung die Datenweitergabe implizit abwickelt.

- `AUS_SPLIT_9`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

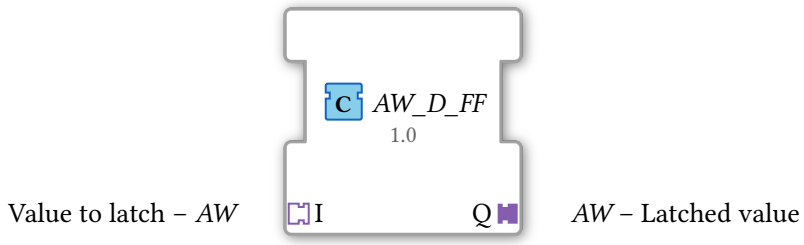
Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

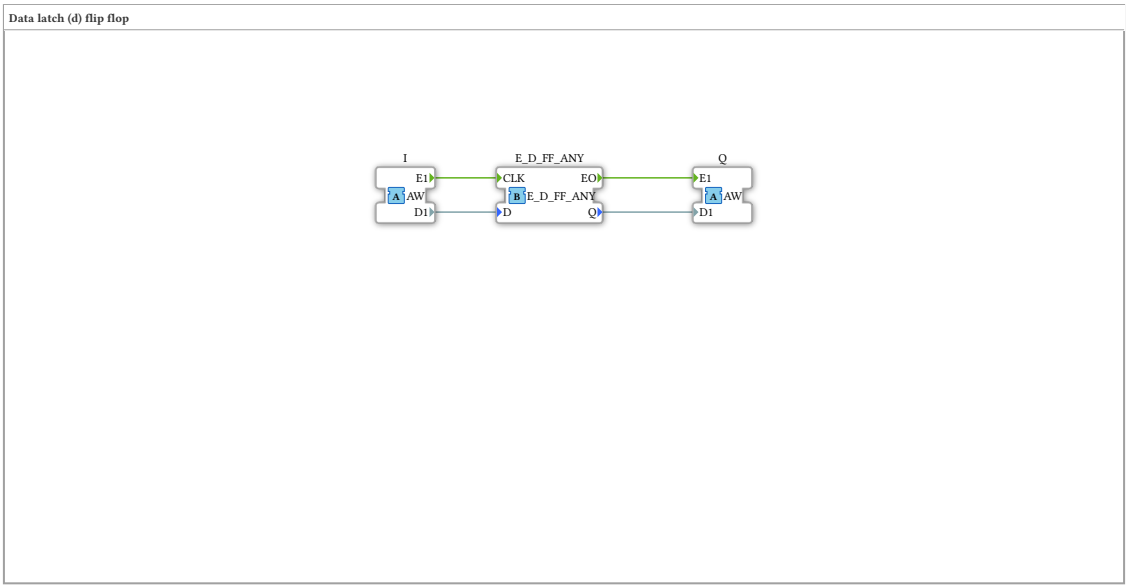
Fazit

Der **AUS_SPLIT_9_UNGATED** ist ein einfacher, aber effektiver Funktionsblock zur Aufteilung eines einzelnen AUS-Signals auf neun parallele Pfade. Seine generische Natur und die reine Adapter-Schnittstelle machen ihn flexibel einsetzbar in IEC 61499-basierten Steuerungsanwendungen, die eine unidirektionale Signalvervielfältigung erfordern.

WORD

AW_D_FF





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AW_D_FF** realisiert ein datengesteuertes Flipflop (D-Latch) nach IEC 61499. Er übernimmt einen binären Datenwert an seinem Eingang und gibt diesen taktgesteuert an seinem Ausgang weiter. Die Schnittstelle erfolgt ausschließlich über Adapter, was eine modulare und flexible Einbindung in bestehende Komponenten ermöglicht.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge auf FB-Ebene. Die Ereignissteuerung erfolgt über den Eingangsadapter **I**.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge auf FB-Ebene. Die Ereignisausgabe erfolgt über den Ausgangsadapter **Q**.

Daten-Eingänge

Keine direkten Daten-Eingänge auf FB-Ebene. Die Datenübergabe erfolgt über den Eingangsadapter **I**.

Daten-Ausgänge

Keine direkten Daten-Ausgänge auf FB-Ebene. Die Datenausgabe erfolgt über den Ausgangsadapter **Q**.

Adapter

Adapter	Typ	Beschreibung
I (Socket)	`adapter::types::unidirectional::AW`	Eingangsadapter – erhält das zu latchende Datum (D1) und das Taktsignal (E1).
Q (Plug)	`adapter::types::unidirectional::AW`	Ausgangsadapter – gibt das latched Datum (D1) sowie ein Bestätigungsereignis (E1) aus.

Beide Adapter sind vom gleichen unidirektionalen Typ und kapseln jeweils ein Ereignis und einen Datenwert.

Funktionsweise

Der Baustein arbeitet intern mit dem IEC-61499-Standardbaustein `E_D_FF_ANY`.
Bei einer positiven Flanke des Taktsignals (Ereignis `E1` am Adapter **I**) wird der aktuelle Datenwert (`D1`) übernommen und am Ausgang bereitgestellt. Der Ausgangswert bleibt so lange erhalten, bis eine neue positive Taktflanke einen neuen Wert übernimmt.

Die logische Tabelle (D-Flipflop) lautet:

Takt (E1)	Eingang D	Ausgang Q (nach Takt)
↑	0	0
↑	1	1
sonst	beliebig	unverändert

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der FB verwendet ausschließlich Adapter für die Kommunikation, was eine lose Kopplung und einfache Wiederverwendung in verschiedenen Kontexten ermöglicht.
- **Standardkonformität**: Entwickelt gemäß IEC 61499-1 Annex A und unter der Eclipse Public License 2.0 (EPL-2.0) lizenziert.
- **Interne Implementierung**: Nutzt den vordefinierten Baustein `E_D_FF_ANY`, der ein allgemeines D-Flipflop bereitstellt.
- **Unidirektionaler Datenfluss**: Die Adapter sind unidirektional ausgelegt, d. h. der Daten- und Ereignisfluss ist nur in eine Richtung definiert.

Zustandsübersicht

Ein D-Flipflop besitzt zwei stabile Zustände für den Ausgang Q (logisch 0 oder 1). Der Zustandswechsel erfolgt ausschließlich bei einer steigenden Flanke des Taktsignals:

- **Zustand 0**: Ausgang Q = 0 (solange kein Takt mit D=1 eintrifft)
- **Zustand 1**: Ausgang Q = 1 (nach einem Takt mit D=1, bis ein Takt mit D=0 eintrifft)

Es gibt keinen internen weiteren Zustandsautomaten; die Funktionalität ist rein speichernd.

Anwendungsszenarien

- **Synchronisation asynchroner Signale**: Ein stabiler Datenwert wird zu einem definierten Zeitpunkt (Takt) übernommen.
- **Zustandsregister**: Als 1-Bit-Speicher in einfachen Steuerungslogiken.
- **Datenpuffer**: Kurzzeitiges Zwischenspeichern eines Binärwerts für nachfolgende Verarbeitungsschritte.
- **Taktgesteuerte Schwellwertentscheidungen**: Nur bei Taktflanke wird ein Eingangswert als gültig betrachtet.

Vergleich mit ähnlichen Bausteinen

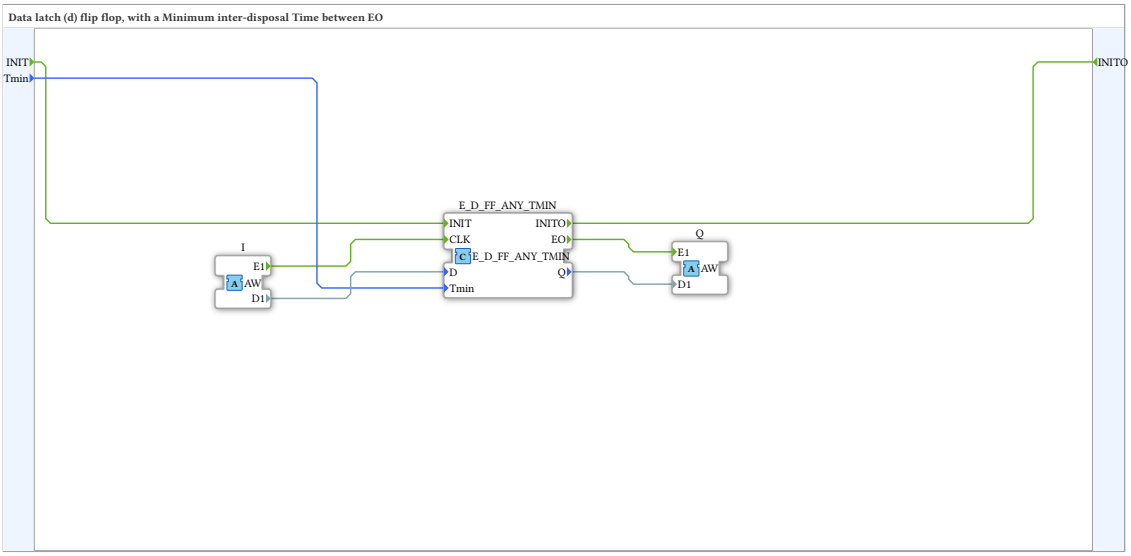
Baustein	Eigenschaft	Unterschied zu AW_D_FF

RS-Flipflop	Setzt/rücksetzt asynchron	AW_D_FF benötigt Takt, keine getrennten Set-/Reset-Eingänge.
T-Flipflop	Toggelt bei Takt	AW_D_FF übernimmt den Wert von D, keine Toggle-Funktion.
JK-Flipflop	Universell (Set, Reset, Toggle)	AW_D_FF ist einfacher (nur D-Eingang), kein Toggle-Modus.
E_D_FF_ANY	Direkt nutzbar mit Events/Daten	AW_D_FF kapselt diesen Baustein und bietet eine Adapter-Schnittstelle.

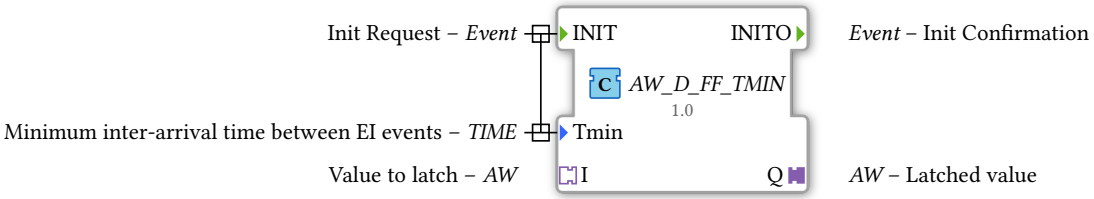
Fazit

Der **AW_D_FF** ist ein kompakter und standardkonformer D-Flipflop-Baustein, der durch seine Adapter-Schnittstelle besonders flexibel in modularen IEC-61499-Anwendungen einsetzbar ist. Er bietet eine zuverlässige taktgesteuerte Datenübernahme und eignet sich ideal für synchronisierte Speicheroperationen in industriellen Steuerungssystemen.

AW_D_FF_TMIN



(Bild des Funktionsblocks – in der XML nicht enthalten, daher kein Bild)



• – ○ • – ○ • – ○ •

Einleitung

Der **AW_D_FF_TMIN** ist ein komponierter IEC 61499-Funktionsblock (Typ eines Daten-Latch-Flipflops), der einen eingehenden Datenwert auf einem Adapter empfängt, bei einem Ereignis übernimmt und auf einem Ausgangsadapter mit einer minimalen Verzögerungszeit zwischen zwei aufeinanderfolgenden Ausgabeereignissen weitergibt. Er dient als zuverlässiger Zwischenspeicher, der die Ausgabe-Ereignisse auf eine durch den Parameter `Tmin` definierte Mindestzeit streckt.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar
`INIT`	EInit	Initialisierungsanforderung; legt die minimale Inter-Event-Zeit `Tmin` fest.

Ereignis-Ausgänge

Ereignis	Typ	Kommentar
`INITO`	EInit	Bestätigung der erfolgreichen Initialisierung.

Daten-Eingänge

Daten	Typ	Kommentar
`Tmin`	TIME	Minimale Zeit (z. B. `T#100ms`), die zwischen zwei aufeinanderfolgenden Ausgabeereignissen (`Q.E1`) mindestens vergehen muss.

Daten-Ausgänge

Der FB besitzt keine direkten Datenausgänge; der latchende Wert wird über den Adapter `Q` bereitgestellt.

Adapter

Richtung	Name	Typ	Kommentar
`Socket`	`I`	`adapter::types::unidirectional::AW`	Wert, der gelatcht werden soll (Eingangsadapter). Enthält die Ereignis- und Datenschnittstellen `I.E1` und `I.D1`.
`Plug`	`Q`	`adapter::types::unidirectional::AW`	Latched Wert (Ausgangsadapter). Enthält die Ereignis- und Datenschnittstellen `Q.E1` und `Q.D1`.

Funktionsweise

Der **AW_D_FF_TMIN** kapselt den IEC-61499-Standardfunktionsblock `E_D_FF_ANY_TMIN`. Die Arbeitsweise ist wie folgt:

1. **Initialisierung**
   Über den Ereigniseingang `INIT` wird der Parameter `Tmin` an den inneren FB übergeben. Nach erfolgreicher Initialisierung wird `INITO` ausgegeben.
2. **Daten-Latching**
   Trifft am Adapter-Eingang `I.E1` ein Ereignis ein, wird der gleichzeitig an `I.D1` anliegende Datenwert übernommen (flankengesteuertes D-Flipflop).
3. **Ausgabe mit Mindestzeit**
   Der gelatchte Wert wird am Adapter-Ausgang `Q.D1` ausgegeben. Gleichzeitig wird ein Ereignis auf `Q.E1` erzeugt – jedoch nur, wenn seit dem letzten Ereignis auf `Q.E1` mindestens die Zeit `Tmin` vergangen ist. Dadurch wird sichergestellt, dass die Ausgangsereignisse nicht schneller als durch `Tmin` erlaubt aufeinander folgen.
4. **Weitere Zyklen**
   Jedes neue Ereignis auf `I.E1` wiederholt das Latching und die zeitgesteuerte Ausgabe.

Technische Besonderheiten

- Der Funktionsblock ist nach IEC 61499-1 Annex A spezifiziert und verwendet die Adapter-Typdefinition `adapter::types::unidirectional::AW`.
- Die Implementierung basiert auf dem reproduzierbaren Baustein `E_D_FF_ANY_TMIN`, der die eigentliche Latch- und Timing-Logik bereitstellt.

- Die minimale Inter-Event-Zeit `Tmin` wird zur Laufzeit über `INIT` gesetzt und kann nicht während des Betriebs geändert werden.
- Der Baustein ist unter der Eclipse Public License 2.0 (EPL-2.0) veröffentlicht, Version 1.0 (2026-05-15, HR Agrartechnik GmbH).

Zustandsübersicht

Der interne FB `E_D_FF_ANY_TMIN` besitzt einen impliziten Zustandsautomaten:

- **Initialisierungszustand**: Nach `INIT` wird die Konfiguration übernommen; bei Erfolg wird `INITO` gesendet.
- **Betriebszustand**: Nach der Initialisierung wartet der FB auf Ereignisse an `I.E1`. Bei jedem Ereignis wird der Datenwert gelatcht, die Timer-Überwachung gestartet und die Ausgabe nur dann freigegeben, wenn die Mindestzeit `Tmin` seit dem letzten Ausgangsereignis abgelaufen ist.
- **Warte-Zustand**: Falls ein Ausgangsereignis noch nicht freigegeben werden darf (weil die Zeit seit dem letzten Ereignis kleiner als `Tmin` ist), wird der gelatchte Wert zurückgehalten, bis die geforderte Verzögerung abgelaufen ist. Erst dann wird `Q.E1` ausgelöst.

Anwendungsszenarien

- **Signalentprellung**: Verhindert, dass mechanische oder kommunikationsbedingte Preller in schneller Folge auf einen nachgeschalteten Prozess wirken.
- **Taktsteuerung mit Mindestpulsdauer**: Sicherstellung, dass nachfolgende Bausteine eine Mindestzeit zwischen zwei Verarbeitungsschritten erhalten.
- **Synchronisation von Datenströmen**: Puffert einen Datenwert und gibt ihn mit einer definierten Mindestverzögerung weiter, um Jitter zu glätten.

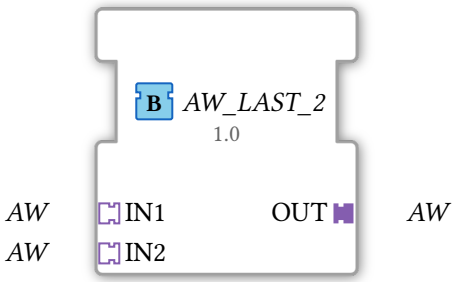
Vergleich mit ähnlichen Bausteinen

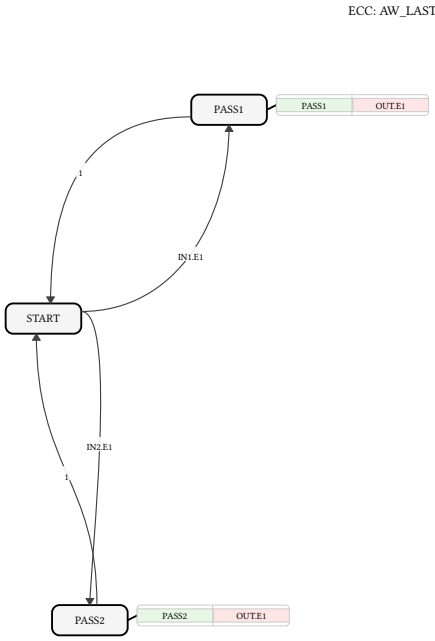
Baustein	Eigenschaft	Unterschied
`E_D_FF`	Einfaches D-Flipflop ohne Zeitbegrenzung	Fehlende Kontrolle über die maximale Ausgabefrequenz.
`SR_FF`	Set-Reset-Flipflop	Keine flankengesteuerte Datenübernahme.
`E_D_FF_ANY_TMIN` (direkt)	Wie AW_D_FF_TMIN, aber reiner Ereignis-/Daten-FB	Besitzt kein Adapter-Interface; die Variante mit Adapter erhöht die Austauschbarkeit und Kapselung in steckbaren Komponenten.

Fazit

Der **AW_D_FF_TMIN** ist ein wartungsfreundlicher, adapterbasierter D-Latch-Baustein, der die Übernahme eines Datenwerts mit einer konfigurierbaren Mindestzeit zwischen Ausgabeereignissen kombiniert. Durch die Verwendung eines standardisierten Adapter-Typs kann er leicht in hierarchische Module eingebunden werden und bietet eine robuste Lösung für zeitkritische Signalketten in IEC-61499-Systemen.

AW_LAST_2





Einleitung

Der AW_LAST_2 Funktionsblock führt zwei gleichartige WORD-Adapter-Signale (**IN1**, **IN2**) auf einen gemeinsamen Ausgang (**OUT**) zusammen, nach dem Prinzip **Last Writer Wins**: welcher der beiden Sockets zuletzt ein Ereignis geschrieben hat, gewinnt, und sein Datenwert wird sofort und unverändert an OUT durchgereicht. Es handelt sich ausdrücklich nicht um ein naives Zusammenführen (z. B. Mittelwert oder ODER-Verknüpfung), sondern um eine reine Arbitrierung nach zeitlicher Reihenfolge.

Schnittstellenstruktur

Adapter

Eingangsadapter (Sockets):

- **IN1**: WORD-Adapter (unidirectional) - erste Signalquelle
- **IN2**: WORD-Adapter (unidirectional) - zweite Signalquelle

Ausgangsadapter (Plug):

- **OUT**: WORD-Adapter (unidirectional) - zusammengeführtes Signal

Funktionsweise

Der Baustein ist als einfacher Basic-FB mit einer 3-Zustands-ECC umgesetzt (START, PASS1, PASS2). Trifft an IN1 ein Ereignis ein, wechselt die ECC nach PASS1: der aktuelle Datenwert von IN1 wird nach OUT kopiert und das Adapter-Ereignis an OUT gesendet. Trifft an IN2 ein Ereignis ein, geschieht dasselbe symmetrisch über PASS2. Nach jeder Durchreichung kehrt die ECC sofort wieder nach START zurück und ist bereit für das nächste Ereignis von IN1 oder IN2.

Technische Besonderheiten

- Last-Writer-Wins-Semantik statt naiver Datenverknüpfung
- Einfacher Basic-FB (ECC mit 3 Zuständen), kein zusätzlicher Zwischenspeicher nötig
- Keine Signalverzögerung: der Wert wird im selben Zyklus durchgereicht, in dem das auslösende Ereignis eintrifft
- Beide Eingänge sind gleichberechtigt - es gibt keine feste Priorität außer der zeitlichen Reihenfolge

Anwendungsszenarien

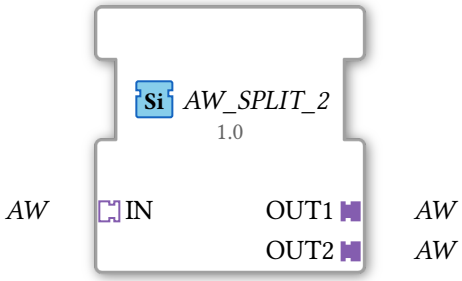
- Zusammenführen zweier gleichwertiger WORD-Signalquellen zu einem einzigen nachgeschalteten Adapterpfad (z. B. zwei redundante Sensoren, oder ein manueller und ein automatischer Sollwertgeber)
- Arbitrierung zwischen zwei Bedienstellen, die denselben WORD-Wert schreiben dürfen
- Vereinfachung von Netzwerken, die sonst ein `F_SEL`/`E_RS`-Konstrukt inline verdrahten müssten

⚖️ Vergleich mit ähnlichen Bausteinen

AW_LAST_2 ist die Umkehrung von `AW_SPLIT_2`: AW_SPLIT_2 verteilt 1 Eingang auf 2 Ausgänge, AW_LAST_2 führt 2 Eingänge auf 1 Ausgang zusammen. Für Mischbetrieb zwischen einem Daten- und einem reinen Ereignisadapter (z. B. AX + AE) ist stattdessen `AX_AE_MERGE` gedacht, das nur das Ereignis mischt, ohne den Datenwert zu verändern.

AW_SPLIT_2

!Bild des FB (nicht verfügbar)



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AW_SPLIT_2** teilt einen eingehenden unidirektionalen **Adapter (AW)** in zwei identische ausgehende Adapter auf. Er ist als generischer Baustein implementiert und eignet sich für Anwendungen, bei denen ein Signalpfad oder eine Adapterverbindung auf zwei nachfolgende Komponenten aufgespalten werden muss.

Schnittstellenstruktur

Ereignis-Eingänge

Kein Ereignis-Eingang vorhanden.

Ereignis-Ausgänge

Kein Ereignis-Ausgang vorhanden.

Daten-Eingänge

Kein Daten-Eingang vorhanden.

Daten-Ausgänge

Kein Daten-Ausgang vorhanden.

Adapter

Bezeichnung	Richtung	Typ	Beschreibung
`IN`	Socket	`adapter::types::unidirectional::AW`	Eingangs-Adapter – der zu teilende unidirektionale AW.
`OUT1`	Plug	`adapter::types::unidirectional::AW`	Erster Ausgang – erhält eine identische Kopie des eingehenden Adapters.

OUT2	Plug	adapter::types::unidirectional::AW	Zweiter Ausgang – erhält ebenfalls eine identische Kopie des eingehenden Adapters.

Funktionsweise

Der Baustein leitet den an **IN** anliegenden unidirektionalen Adapter (AW) unverändert an beide Ausgänge **OUT1** und **OUT2** weiter. Es findet keine Datenmanipulation, keine Ereignissteuerung und keine Zustandslogik statt – der Baustein fungiert ausschließlich als passive Verteilereinheit auf Adapterebene.

Da der Adaptertyp als generisch deklariert ist (`GEN_AW_SPLIT`), kann der FB mit verschiedenen konkreten Ausprägungen des AW-Adapters verwendet werden, solange sie dem gleichen `unidirectional::AW` Protokoll folgen.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generische Komponente (GenericClassName `GEN_AW_SPLIT`) definiert. Er kann in Projekten mit unterschiedlichen AW-Adapter-Typen eingesetzt werden, ohne dass der Baustein selbst angepasst werden muss.
- **Keine Laufzeitlogik**: Die Aufteilung erfolgt rein strukturell – der FB besitzt weder einen Zustandsautomaten (ECC) noch verarbeitet er Ereignisse oder Daten. Dadurch entsteht kein zusätzlicher Rechenaufwand zur Laufzeit.
- **Lizenz**: Der Baustein steht unter der **Eclipse Public License 2.0 (EPL-2.0)**.

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustandsautomaten. Die Weiterleitung ist statisch und erfolgt ohne zeitliche oder ereignisabhängige Steuerung.

Anwendungsszenarien

- **Verteilen eines Mess- oder Status-Adapterpfades** an zwei unabhängige Verarbeitungseinheiten (z. B. an eine Hauptsteuerung und eine Überwachungskomponente).
- **Aufbau redundanter Verarbeitungsketten**, bei denen derselbe Adapterstrom parallel in zwei separate Logikzweige eingespeist wird.
- **Test- und Diagnosearchitekturen**, die einen Datenstrom sowohl in die produktive Logik als auch in einen Analyse-Pfad aufteilen.

Vergleich mit ähnlichen Bausteinen

- **AW_MERGE_2** – führt zwei AW-Eingänge zu einem Ausgang zusammen (Gegenstück zum Split).
- **Daten-Split-Bausteine** (z. B. `SPLIT_INT`, `SPLIT_BOOL`) arbeiten auf Datensignalen, nicht auf Adaptern. `AW_SPLIT_2` ist speziell für die Aufteilung von Adapter-Schnittstellen ausgelegt.
- `AW_SPLIT_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

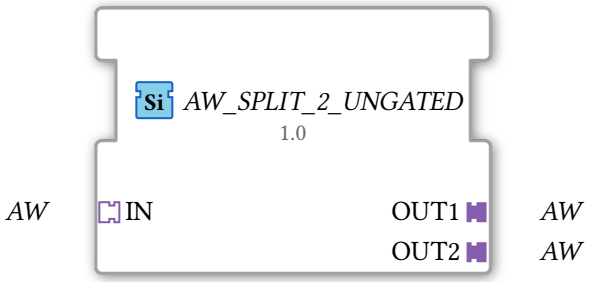
`AW_SPLIT_2` ist ein einfacher, aber wesentlicher generischer Baustein zur Adapter-Verteilung. Er ermöglicht eine saubere, wiederverwendbare Aufspaltung eines unidirektionalen AW-Pfades, ohne zusätzliche Logik oder Laufzeitkosten. Besonders in modularen, adapterbasierten Steuerungssystemen erleichtert er die Signalverteilung und fördert die Wiederverwendbarkeit von Komponenten.

AW_SPLIT_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AW_SPLIT_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz

unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

!Bild des FB (nicht verfügbar)



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AW_SPLIT_2_UNGATED** teilt einen eingehenden unidirektionalen **Adapter (AW)** in zwei identische ausgehende Adapter auf. Er ist als generischer Baustein implementiert und eignet sich für Anwendungen, bei denen ein Signalpfad oder eine Adapterverbindung auf zwei nachfolgende Komponenten aufgespalten werden muss.

Schnittstellenstruktur

Ereignis-Eingänge

Kein Ereignis-Eingang vorhanden.

Ereignis-Ausgänge

Kein Ereignis-Ausgang vorhanden.

Daten-Eingänge

Kein Daten-Eingang vorhanden.

Daten-Ausgänge

Kein Daten-Ausgang vorhanden.

Adapter

Bezeichnung	Richtung	Typ	Beschreibung
`IN`	Socket	`adapter::types::unidirectional::AW`	Eingangs-Adapter – der zu teilende unidirektionale AW.
`OUT1`	Plug	`adapter::types::unidirectional::AW`	Erster Ausgang – erhält eine identische Kopie des eingehenden Adapters.
`OUT2`	Plug	`adapter::types::unidirectional::AW`	Zweiter Ausgang – erhält ebenfalls eine identische Kopie des eingehenden Adapters.

Funktionsweise

Der Baustein leitet den an **IN** anliegenden unidirektionalen Adapter (AW) unverändert an beide Ausgänge **OUT1** und **OUT2** weiter. Es findet keine Datenmanipulation, keine Ereignissteuerung und keine Zustandslogik statt – der Baustein fungiert ausschließlich als passive Verteilereinheit auf Adapterebene.

Da der Adaptertyp als generisch deklariert ist (`GEN_AW_SPLIT`), kann der FB mit verschiedenen konkreten Ausprägungen des AW-Adapters verwendet werden, solange sie dem gleichen `unidirectional::AW` Protokoll folgen.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generische Komponente (GenericClassName `GEN_AW_SPLIT`) definiert. Er kann in Projekten mit unterschiedlichen AW-Adapter-Typen eingesetzt werden, ohne dass der Baustein selbst angepasst werden muss.
- **Keine Laufzeitlogik**: Die Aufteilung erfolgt rein strukturell – der FB besitzt weder einen Zustandsautomaten (ECC) noch verarbeitet er Ereignisse oder Daten. Dadurch entsteht kein zusätzlicher Rechenaufwand zur Laufzeit.
- **Lizenz**: Der Baustein steht unter der **Eclipse Public License 2.0 (EPL-2.0)**.

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustandsautomaten. Die Weiterleitung ist statisch und erfolgt ohne zeitliche oder ereignisabhängige Steuerung.

Anwendungsszenarien

- **Verteilen eines Mess- oder Status-Adapterpfades** an zwei unabhängige Verarbeitungseinheiten (z. B. an eine Hauptsteuerung und eine Überwachungskomponente).
- **Aufbau redundanter Verarbeitungsketten**, bei denen derselbe Adapterstrom parallel in zwei separate Logikzweige eingespeist wird.
- **Test- und Diagnosearchitekturen**, die einen Datenstrom sowohl in die produktive Logik als auch in einen Analyse-Pfad aufteilen.

Vergleich mit ähnlichen Bausteinen

- **AW_MERGE_2** – führt zwei AW-Eingänge zu einem Ausgang zusammen (Gegenstück zum Split).
- **Daten-Split-Bausteine** (z. B. `SPLIT_INT`, `SPLIT_BOOL`) arbeiten auf Datensignalen, nicht auf Adaptern. `AW_SPLIT_2_UNGATED` ist speziell für die Aufteilung von Adapter-Schnittstellen ausgelegt.
- `AW_SPLIT_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

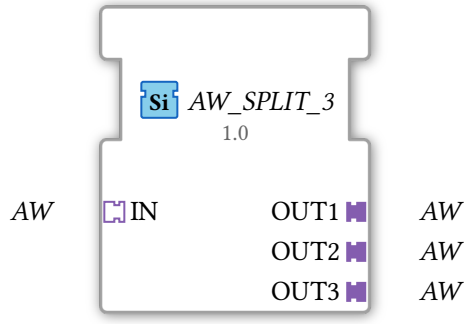
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

`AW_SPLIT_2_UNGATED` ist ein einfacher, aber wesentlicher generischer Baustein zur Adapter-Verteilung. Er ermöglicht eine saubere, wiederverwendbare Aufspaltung eines unidirektionalen AW-Pfades, ohne zusätzliche Logik oder Laufzeitkosten. Besonders in modularen, adapterbasierten Steuerungssystemen erleichtert er die Signalverteilung und fördert die Wiederverwendbarkeit von Komponenten.

AW_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AW_SPLIT_3** dient als generischer Verteiler für einen unidirektionalen Adapter des Typs `AW`. Er nimmt eine einzige Adapterverbindung (Socket `IN`) entgegen und stellt diese identisch auf drei Ausgänge (Plugs `OUT1`, `OUT2`, `OUT3`) bereit. Der Baustein ist als generischer Funktionsblock (Generic

FB) ausgelegt, sodass der konkrete Datentyp des Adapters `AW` zur Laufzeit festgelegt werden kann. Die Anwendung liegt in der Signalkopplung oder Befehlsweitergabe an mehrere nachfolgende Komponenten.

Schnittstellenstruktur

Ereignis-Eingänge
Keine.

Ereignis-Ausgänge
Keine.

Daten-Eingänge
Keine.

Daten-Ausgänge
Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	`IN`	`adapter::types::unidirectional::AW`	Eingangsadapter – Quelle des zu verteilenden Signals.
Plug	`OUT1`	`adapter::types::unidirectional::AW`	Erster Ausgang – identisches Signal wie `IN`.
Plug	`OUT2`	`adapter::types::unidirectional::AW`	Zweiter Ausgang – identisches Signal wie `IN`.
Plug	`OUT3`	`adapter::types::unidirectional::AW`	Dritter Ausgang – identisches Signal wie `IN`.

Funktionsweise
Der Baustein arbeitet ohne eigenen internen Zustandsautomaten oder Datenverarbeitung. Er stellt eine passive Verteilerstruktur dar: Die am Socket `IN` ankommende Adapterverbindung wird auf die drei Plugs `OUT1`, `OUT2` und `OUT3` durchgeschaltet. Alle ausgehenden Adapter sind logisch und zeitlich identisch zum eingehenden Signal. Eine Verzögerung, Pufferung oder Transformation findet nicht statt.

Da es sich um einen generischen Funktionsblock handelt, muss der Adaptertyp `AW` vor der Inbetriebnahme mit einem konkreten, kompatiblen Typ (z. B. `Analogwert` oder `Steuerbefehl`) parametrisiert werden. Dies geschieht über die Angabe des Attributs `eclipse4diac::core::GenericClassName`. Die Typauflösung erfolgt entweder zur Entwurfszeit (IDE) oder zur Laufzeit des Systems.

Technische Besonderheiten
- **Generischer Ansatz**: Der Baustein ist als *Generic FB* realisiert (`GenericClassName = 'GEN_AW_SPLIT'`). Dadurch kann derselbe Funktionsblock für verschiedene Adaptertypen verwendet werden, ohne dass für jeden Typ ein eigener FB erstellt werden muss.
- **Unidirektionalität**: Die Adapter sind als `unidirectional` deklariert. Das bedeutet, dass die Datenflussrichtung nur vom Socket zu den Plugs verläuft. Eine Rückmeldung von den Ausgängen zum Eingang ist nicht vorgesehen.
- **Keine Laufzeitlogik**: Der FB ist rein strukturell (passiv) – es werden keine Ereignisse, Zustände oder Berechnungen ausgeführt. Dies reduziert den Ressourcenverbrauch in Echtzeitsystemen.
- **Verwendung der Eclipse 4diac-Umgebung**: Der Baustein setzt auf die von 4diac bereitgestellten Mechanismen für generische Typen und Adapter.

Zustandsübersicht
Der Funktionsblock besitzt keinen eigenen Zustandsautomaten (ECC). Er ist ein reiner Verbindungsbaustein ohne zeitliches Verhalten oder Zustandsänderungen. Die Funktionalität beschränkt sich auf die statische Weiterleitung des Adapters.

Anwendungsszenarien

- **Signalfan-out in Steuerungen**: Ein Messwert (z. B. Temperatur, Druck) soll parallel an mehrere Regler oder Überwachungseinheiten gesendet werden.
- **Verteilen von Steuerbefehlen**: Ein zentrales Kommando wird an mehrere Aktoren oder Subsysteme gleichzeitig weitergegeben.
- **Redundante Signalpfade**: In sicherheitskritischen Anwendungen kann das gleiche Signal auf mehrere Pfade aufgeteilt werden, um unabhängige Auswertungen zu ermöglichen.
- **Prototypenentwicklung**: Der generische Charakter erlaubt es, den Baustein frühzeitig einzusetzen und den konkreten Adaptertyp später festzulegen.

Vergleich mit ähnlichen Bausteinen

- **AW_MERGE_3** (hypothetisch): Ein Zusammenführer von drei AW-Eingängen auf einen Ausgang. Der AW_SPLIT_3 ist die logische Umkehrung.
- **SPLIT_ALL_2**: Ein nicht-generischer Splitter für zwei Ausgänge, der spezifische Datentypen verwendet. Der AW_SPLIT_3 bietet durch die Generizität mehr Flexibilität.
- **REPEATER**: Ein einfacher Verstärker oder Repeater für Adapter, der nur einen Ausgang besitzt. Der AW_SPLIT_3 erweitert dies auf drei parallele Ausgänge.
- `AW_SPLIT_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

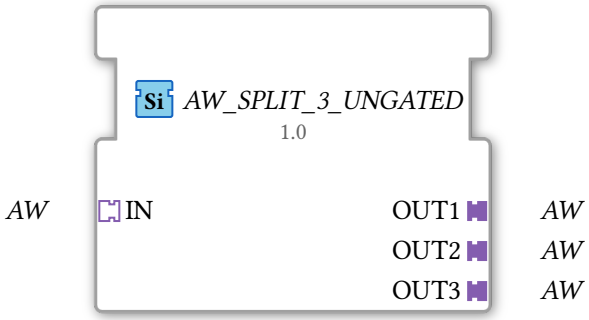
Der **AW_SPLIT_3** ist ein grundlegender, aber flexibler Verteilerbaustein in Eclipse 4diac. Seine generische Auslegung macht ihn universell einsetzbar für alle unidirektionalen Adaptertypen. Die klare, passive Struktur ohne zusätzliche Logik erlaubt eine effiziente und zuverlässige Signalverteilung in Automatisierungslösungen. Besonders in Kombination mit der Typvariabilität und 4diac-Adaptermechanik stellt er ein wertvolles Werkzeug für modulare und wiederverwendbare Steuerungsarchitekturen dar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AW_SPLIT_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AW_SPLIT_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AW_SPLIT_3_UNGATED** dient als generischer Verteiler für einen unidirektionalen Adapter des Typs `AW`. Er nimmt eine einzige Adapterverbindung (Socket `IN`) entgegen und stellt diese identisch auf drei Ausgänge (Plugs `OUT1`, `OUT2`, `OUT3`) bereit. Der Baustein ist als generischer Funktionsblock (Generic FB) ausgelegt, sodass der konkrete Datentyp des Adapters `AW` zur Laufzeit festgelegt werden kann. Die Anwendung liegt in der Signalkopplung oder Befehlsweitergabe an mehrere nachfolgende Komponenten.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	`IN`	`adapter::types::unidirectional::AW`	Eingangsadapter – Quelle des zu verteilenden Signals.
Plug	`OUT1`	`adapter::types::unidirectional::AW`	Erster Ausgang – identisches Signal wie `IN`.
Plug	`OUT2`	`adapter::types::unidirectional::AW`	Zweiter Ausgang – identisches Signal wie `IN`.
Plug	`OUT3`	`adapter::types::unidirectional::AW`	Dritter Ausgang – identisches Signal wie `IN`.

Funktionsweise

Der Baustein arbeitet ohne eigenen internen Zustandsautomaten oder Datenverarbeitung. Er stellt eine passive Verteilerstruktur dar: Die am Socket `IN` ankommende Adapterverbindung wird auf die drei Plugs `OUT1`, `OUT2` und `OUT3` durchgeschaltet. Alle ausgehenden Adapter sind logisch und zeitlich identisch zum eingehenden Signal. Eine Verzögerung, Pufferung oder Transformation findet nicht statt.

Da es sich um einen generischen Funktionsblock handelt, muss der Adaptertyp `AW` vor der Inbetriebnahme mit einem konkreten, kompatiblen Typ (z. B. `Analogwert` oder `Steuerbefehl`) parametrisiert werden. Dies geschieht über die Angabe des Attributs `eclipse4diac::core::GenericClassName`. Die Typauflösung erfolgt entweder zur Entwurfszeit (IDE) oder zur Laufzeit des Systems.

Technische Besonderheiten

- **Generischer Ansatz**: Der Baustein ist als *Generic FB* realisiert (`GenericClassName = 'GEN_AW_SPLIT'`). Dadurch kann derselbe Funktionsblock für verschiedene Adaptertypen verwendet werden, ohne dass für jeden Typ ein eigener FB erstellt werden muss.
- **Unidirektionalität**: Die Adapter sind als `unidirectional` deklariert. Das bedeutet, dass die Datenflussrichtung nur vom Socket zu den Plugs verläuft. Eine Rückmeldung von den Ausgängen zum Eingang ist nicht vorgesehen.
- **Keine Laufzeitlogik**: Der FB ist rein strukturell (passiv) – es werden keine Ereignisse, Zustände oder Berechnungen ausgeführt. Dies reduziert den Ressourcenverbrauch in Echtzeitsystemen.
- **Verwendung der Eclipse 4diac-Umgebung**: Der Baustein setzt auf die von 4diac bereitgestellten Mechanismen für generische Typen und Adapter.

Zustandsübersicht

Der Funktionsblock besitzt keinen eigenen Zustandsautomaten (ECC). Er ist ein reiner Verbindungsbaustein ohne zeitliches Verhalten oder Zustandsänderungen. Die Funktionalität beschränkt sich auf die statische Weiterleitung des Adapters.

Anwendungsszenarien

- **Signalfan-out in Steuerungen**: Ein Messwert (z. B. Temperatur, Druck) soll parallel an mehrere Regler oder Überwachungseinheiten gesendet werden.
- **Verteilen von Steuerbefehlen**: Ein zentrales Kommando wird an mehrere Aktoren oder Subsysteme gleichzeitig weitergegeben.
- **Redundante Signalpfade**: In sicherheitskritischen Anwendungen kann das gleiche Signal auf mehrere Pfade aufgeteilt werden, um unabhängige Auswertungen zu ermöglichen.
- **Prototypenentwicklung**: Der generische Charakter erlaubt es, den Baustein frühzeitig einzusetzen und den konkreten Adaptertyp später festzulegen.

Vergleich mit ähnlichen Bausteinen

- **AW_MERGE_3** (hypothetisch): Ein Zusammenführer von drei AW-Eingängen auf einen Ausgang. Der AW_SPLIT_3_UNGATED ist die logische Umkehrung.
- **SPLIT_ALL_2**: Ein nicht-generischer Splitter für zwei Ausgänge, der spezifische Datentypen verwendet. Der AW_SPLIT_3_UNGATED bietet durch die Generizität mehr Flexibilität.
- **REPEATER**: Ein einfacher Verstärker oder Repeater für Adapter, der nur einen Ausgang besitzt. Der AW_SPLIT_3_UNGATED erweitert dies auf drei parallele Ausgänge.
- `AW_SPLIT_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

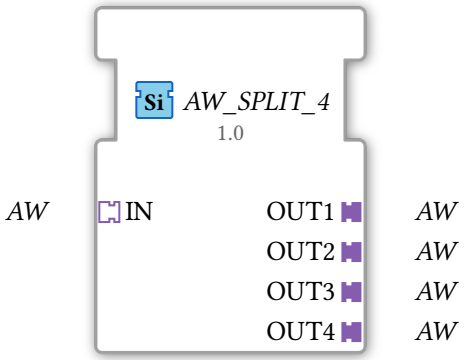
Fazit

Der **AW_SPLIT_3_UNGATED** ist ein grundlegender, aber flexibler Verteilerbaustein in Eclipse 4diac. Seine generische Auslegung macht ihn universell einsetzbar für alle unidirektionalen Adaptertypen. Die klare, passive Struktur ohne zusätzliche Logik erlaubt eine effiziente und zuverlässige Signalverteilung in Automatisierungslösungen. Besonders in Kombination mit der Typvariabilität und 4diac-Adaptermechanik stellt er ein wertvolles Werkzeug für modulare und wiederverwendbare Steuerungsarchitekturen dar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AW_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AW_SPLIT_4` verteilt einen eingehenden AW-Adapter (unidirektional) auf vier identische AW-Ausgangsadapter. Der Baustein ist generisch ausgelegt und kann mit unterschiedlichen AW-Typen verwendet werden.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine (die Datenübertragung erfolgt ausschließlich über den Adapter-Socket).

Daten-Ausgänge

Keine (die Ausgabe erfolgt ausschließlich über die Adapter-Plugs).

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AW`	`IN`	Socket	Eingang für den zu verteilenden AW-Datenstrom
`adapter::types::unidirectional::AW`	`OUT1`	Plug	Erster Ausgang (Kopie des Eingangs)
`adapter::types::unidirectional::AW`	`OUT2`	Plug	Zweiter Ausgang
`adapter::types::unidirectional::AW`	`OUT3`	Plug	Dritter Ausgang
`adapter::types::unidirectional::AW`	`OUT4`	Plug	Vierter Ausgang

Funktionsweise

Der Baustein agiert als passiver Verteiler: Jeder über den Socket `IN` ankommende Wert (bzw. jedes Datenpaket) wird unverändert an alle vier Plugs `OUT1` bis `OUT4` weitergeleitet. Es findet keine Verarbeitung, Pufferung oder zeitliche Synchronisation statt. Der gesamte Datenfluss ist ereignislos und rein adapterbasiert.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist mit dem generischen Namen `GEN_AW_SPLIT` deklariert. Das ermöglicht eine späte Bindung an konkrete AW-Adapter-Typen (z. B. `AT_AW`, `CT_AW` etc.) zur Compile- oder Konfigurationszeit.
- **Unidirektionale Kommunikation**: Die verwendeten Adapter sind vom Typ `unidirectional::AW`, d. h. die Daten fließen nur in eine Richtung. Es gibt keine Rückkanäle oder Bestätigungen.
- **Keine Eigendynamik**: Der FB besitzt keine Ereignisse, Zustände oder zeitabhängige Logik. Er ist rein kombinatorisch.

Zustandsübersicht

Nicht anwendbar – der FB enthält keine Zustandsmaschine (ECC) und arbeitet durchgehend datengetrieben.

Anwendungsszenarien

- Verteilung eines einzelnen Sensor- oder Aktor-Signalpfads auf mehrere parallele Auswertungen.
- Aufbereitung einer Datenquelle für unterschiedliche Verbraucher in einer Steuerungsanwendung.
- Erzeugung von Test- oder Simulationspunkten durch Vervielfachung eines AW-Datenstroms.

Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu daten- oder ereignisbasierten Split-Bausteinen (z. B. `SPLIT`, `F_SPLIT`) arbeitet `AW_SPLIT_4` ausschließlich auf Adapterebene. Der Vorteil liegt in der losen Kopplung und der Wiederverwendbarkeit

über verschiedene AW-Typen hinweg. Nachteil: Es können nur AW-Daten verteilt werden, nicht etwa Ereignisse oder allgemeine Werte.

- `AW_SPLIT_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

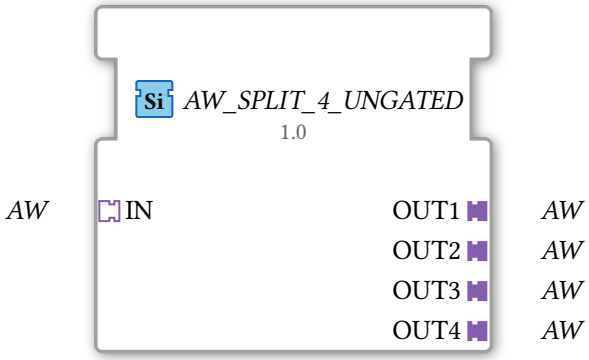
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der Baustein `AW_SPLIT_4` ist ein einfacher, generischer Verteiler für unidirektionale AW-Adapter. Er eignet sich ideal, um ein einzelnes AW-Signal auf bis zu vier Ausgänge aufzuteilen, und fügt sich dank seiner generischen Natur flexibel in unterschiedliche IEC 61499-Anwendungen ein.

AW_SPLIT_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AW_SPLIT_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AW_SPLIT_4_UNGATED` verteilt einen eingehenden AW-Adapter (unidirektional) auf vier identische AW-Ausgangsadapter. Der Baustein ist generisch ausgelegt und kann mit unterschiedlichen AW-Typen verwendet werden.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine (die Datenübertragung erfolgt ausschließlich über den Adapter-Socket).

Daten-Ausgänge

Keine (die Ausgabe erfolgt ausschließlich über die Adapter-Plugs).

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AW`	`IN`	Socket	Eingang für den zu verteilenden AW-Datenstrom
`adapter::types::unidirectional::AW`	`OUT1`	Plug	Erster Ausgang (Kopie des Eingangs)
`adapter::types::unidirectional::AW`	`OUT2`	Plug	Zweiter Ausgang
`adapter::types::unidirectional::AW`	`OUT3`	Plug	Dritter Ausgang
`adapter::types::unidirectional::AW`	`OUT4`	Plug	Vierter Ausgang

Funktionsweise

Der Baustein agiert als passiver Verteiler: Jeder über den Socket `IN` ankommende Wert (bzw. jedes Datenpaket) wird unverändert an alle vier Plugs `OUT1` bis `OUT4` weitergeleitet. Es findet keine Verarbeitung, Pufferung oder zeitliche Synchronisation statt. Der gesamte Datenfluss ist ereignislos und rein adapterbasiert.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist mit dem generischen Namen `GEN_AW_SPLIT` deklariert. Das ermöglicht eine späte Bindung an konkrete AW-Adapter-Typen (z. B. `AT_AW`, `CT_AW` etc.) zur Compile- oder Konfigurationszeit.
- **Unidirektionale Kommunikation**: Die verwendeten Adapter sind vom Typ `unidirectional::AW`, d. h. die Daten fließen nur in eine Richtung. Es gibt keine Rückkanäle oder Bestätigungen.
- **Keine Eigendynamik**: Der FB besitzt keine Ereignisse, Zustände oder zeitabhängige Logik. Er ist rein kombinatorisch.

Zustandsübersicht

Nicht anwendbar – der FB enthält keine Zustandsmaschine (ECC) und arbeitet durchgehend datengetrieben.

Anwendungsszenarien

- Verteilung eines einzelnen Sensor- oder Aktor-Signalpfads auf mehrere parallele Auswertungen.
- Aufbereitung einer Datenquelle für unterschiedliche Verbraucher in einer Steuerungsanwendung.
- Erzeugung von Test- oder Simulationspunkten durch Vervielfachung eines AW-Datenstroms.

Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu daten- oder ereignisbasierten Split-Bausteinen (z. B. `SPLIT`, `F_SPLIT`) arbeitet `AW_SPLIT_4_UNGATED` ausschließlich auf Adapterebene. Der Vorteil liegt in der losen Kopplung und der Wiederverwendbarkeit über verschiedene AW-Typen hinweg. Nachteil: Es können nur AW-Daten verteilt werden, nicht etwa Ereignisse oder allgemeine Werte.

- **`AW_SPLIT_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

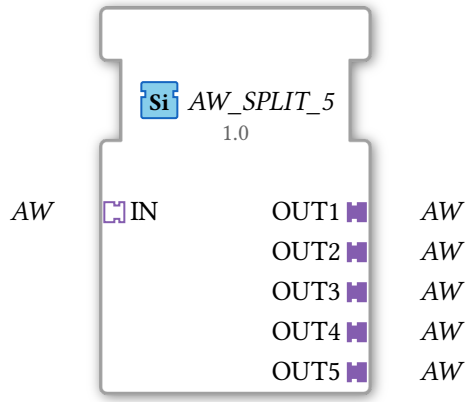
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der Baustein `AW_SPLIT_4_UNGATED` ist ein einfacher, generischer Verteiler für unidirektionale AW-Adapter. Er eignet sich ideal, um ein einzelnes AW-Signal auf bis zu vier Ausgänge aufzuteilen, und fügt sich dank seiner generischen Natur flexibel in unterschiedliche IEC 61499-Anwendungen ein.

AW_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AW_SPLIT_5** dient dazu, einen eingehenden AW-Adapter („Aktor/Wege“) auf fünf separate, identische AW-Ausgänge zu verteilen. Er ist als generischer Baustein ausgelegt und kann in Steuerungsarchitekturen eingesetzt werden, in denen ein einzelnes Signal oder eine Datenverbindung auf mehrere parallele Pfade aufgeteilt werden muss.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

- **IN** (Socket) – Typ: `adapter::types::unidirectional::AW` Eingangsschnittstelle für den ursprünglichen AW-Datenstrom.
- **OUT1** bis **OUT5** (Plugs) – Typ: `adapter::types::unidirectional::AW` Fünf parallele Ausgangsschnittstellen, die jeweils das gleiche Signal wie am Eingang bereitstellen.

Funktionsweise

Der FB leitet den eingehenden AW-Adapter passiv auf alle fünf Ausgänge weiter. Eine aktive Logik oder Zustandsmaschine ist nicht vorhanden – die Verteilung erfolgt strukturell durch die Verdrahtung im FB-Inneren. Änderungen am Eingang werden unverzögert an allen Ausgängen sichtbar.

Technische Besonderheiten

- **Generischer Baustein** – Der FB ist als generischer Typ (`GEN_AW_SPLIT`) deklariert und kann in unterschiedlichen Kontexten wiederverwendet werden.
- **Keine Ereignissteuerung** – Die Signalweitergabe erfolgt kontinuierlich ohne ereignisgesteuerte Aktivierung.
- **Unidirektionale Adapter** – Der verwendete Adaptertyp `AW` ist unidirektional, d. h. die Daten fließen nur vom Eingang zu den Ausgängen. Eine Rückmeldung von den Ausgängen ist nicht vorgesehen.

Zustandsübersicht

Der FB besitzt keine interne Zustandsmaschine (ECC). Es gibt keine definierten Zustände oder Zustandsübergänge – die Funktionalität ist rein datenflussorientiert.

Anwendungsszenarien

- **Verteilen eines Sensorsignals** an mehrere nachgelagerte Steuerungseinheiten.
- **Parallele Ansteuerung** mehrerer Aktoren, die denselben Sollwert oder dasselbe Kommando erhalten sollen.
- **Testaufbauten** – Ein Signal aufzeichnen oder gleichzeitig an verschiedene Simulations- und Analysekomponenten weiterleiten.

Vergleich mit ähnlichen Bausteinen

- **AW_SPLIT_N** – Ein weiterer generischer Split-FB, der auf eine variable Anzahl von Ausgängen erweiterbar ist. `AW_SPLIT_5` ist eine spezialisierte Variante mit fest fünf Ausgängen.
- **Reine Daten-Splitter** – Diese arbeiten auf Datentypen wie `INT` oder `BOOL` und nutzen andere Schnittstellentypen. `AW_SPLIT_5` ist speziell für den AW-Adapter konzipiert.
- `AW_SPLIT_5_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

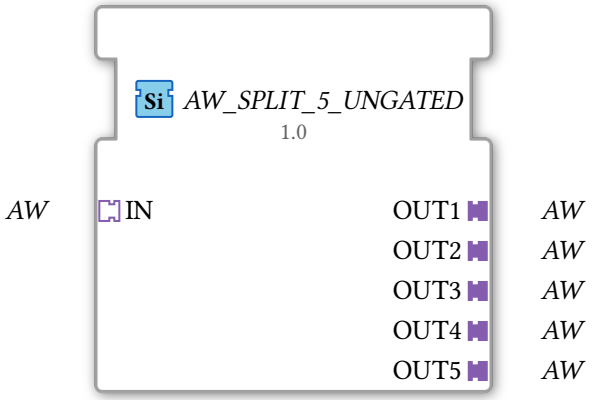
Der `AW_SPLIT_5` ist ein einfacher, aber nützlicher Baustein zur Aufteilung eines AW-Adapter-Signals auf fünf parallele Pfade. Durch seinen generischen Aufbau kann er flexibel in unterschiedlichen Steuerungsprojekten der 4diac-IDE eingesetzt werden. Die fehlende Ereignis- und Daten-Schnittstelle reduziert die Komplexität auf das Wesentliche – die reine Signalvervielfältigung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AW_SPLIT_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AW_SPLIT_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AW_SPLIT_5_UNGATED** dient dazu, einen eingehenden AW-Adapter („Aktor/Wege“) auf fünf separate, identische AW-Ausgänge zu verteilen. Er ist als generischer Baustein ausgelegt und kann in Steuerungsarchitekturen eingesetzt werden, in denen ein einzelnes Signal oder eine Datenverbindung auf mehrere parallele Pfade aufgeteilt werden muss.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

- **IN** (Socket) – Typ: `adapter::types::unidirectional::AW` Eingangsschnittstelle für den ursprünglichen AW-Datenstrom.
- **OUT1** bis **OUT5** (Plugs) – Typ: `adapter::types::unidirectional::AW` Fünf parallele Ausgangsschnittstellen, die jeweils das gleiche Signal wie am Eingang bereitstellen.

Funktionsweise

Der FB leitet den eingehenden AW-Adapter passiv auf alle fünf Ausgänge weiter. Eine aktive Logik oder Zustandsmaschine ist nicht vorhanden – die Verteilung erfolgt strukturell durch die Verdrahtung im FB-Inneren. Änderungen am Eingang werden unverzögert an allen Ausgängen sichtbar.

Technische Besonderheiten

- **Generischer Baustein** – Der FB ist als generischer Typ (`GEN_AW_SPLIT`) deklariert und kann in unterschiedlichen Kontexten wiederverwendet werden.
- **Keine Ereignissteuerung** – Die Signalweitergabe erfolgt kontinuierlich ohne ereignisgesteuerte Aktivierung.
- **Unidirektionale Adapter** – Der verwendete Adaptertyp `AW` ist unidirektional, d. h. die Daten fließen nur vom Eingang zu den Ausgängen. Eine Rückmeldung von den Ausgängen ist nicht vorgesehen.

Zustandsübersicht

Der FB besitzt keine interne Zustandsmaschine (ECC). Es gibt keine definierten Zustände oder Zustandsübergänge – die Funktionalität ist rein datenflussorientiert.

Anwendungsszenarien

- **Verteilen eines Sensorsignals** an mehrere nachgelagerte Steuerungseinheiten.
- **Parallele Ansteuerung** mehrerer Aktoren, die denselben Sollwert oder dasselbe Kommando erhalten sollen.
- **Testaufbauten** – Ein Signal aufzeichnen oder gleichzeitig an verschiedene Simulations- und Analysekomponenten weiterleiten.

Vergleich mit ähnlichen Bausteinen

- **AW_SPLIT_N** – Ein weiterer generischer Split-FB, der auf eine variable Anzahl von Ausgängen erweiterbar ist. `AW_SPLIT_5_UNGATED` ist eine spezialisierte Variante mit fest fünf Ausgängen.
- **Reine Daten-Splitter** – Diese arbeiten auf Datentypen wie `INT` oder `BOOL` und nutzen andere Schnittstellentypen. `AW_SPLIT_5_UNGATED` ist speziell für den AW-Adapter konzipiert.
- `AW_SPLIT_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

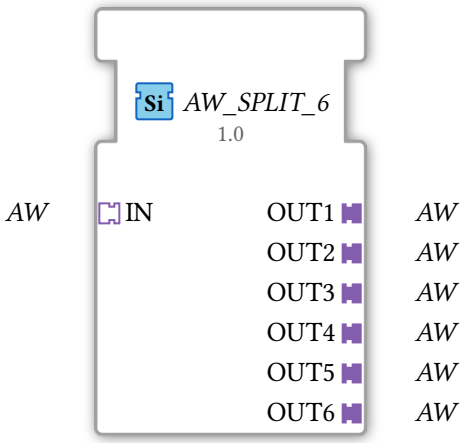
Der `AW_SPLIT_5_UNGATED` ist ein einfacher, aber nützlicher Baustein zur Aufteilung eines AW-Adapter-Signals auf fünf parallele Pfade. Durch seinen generischen Aufbau kann er flexibel in unterschiedlichen Steuerungsprojekten der 4diac-IDE eingesetzt werden. Die fehlende Ereignis- und Daten-Schnittstelle reduziert die Komplexität auf das Wesentliche – die reine Signalvervielfältigung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AW_SPLIT_6

Bild nicht verfügbar



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AW_SPLIT_6** dient dazu, einen eingehenden unidirektionalen **AW**-Adapter auf sechs identische Ausgangs-Adapter aufzuteilen. Er ist als generischer Baustein konzipiert, der in der 4diac-IDE zur Verteilung von AW-Signalen verwendet wird. Der Baustein ist für Anwendungen gedacht, bei denen ein Signalwert – z. B. ein analoger oder digitaler Wert – mehrfach parallel benötigt wird, ohne dass eine manuelle Vervielfältigung der Verbindungen erfolgen muss.

Schnittstellenstruktur

Ereignis-Eingänge
Keine

Ereignis-Ausgänge
Keine

Daten-Eingänge
Keine

Daten-Ausgänge
Keine

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AW`	IN	Socket	Eingangs-Adapter für das zu verteilende AW-Signal
`adapter::types::unidirectional::AW`	OUT1 – OUT6	Plugs	Sechs Ausgangs-Adapter, die jeweils eine identische Kopie des Eingangssignals bereitstellen

Funktionsweise
Der Baustein empfängt über den Socket **IN** ein unidirektionales **AW**-Signal. Dieses Signal wird intern auf alle sechs Ausgangs-Plugs (**OUT1** bis **OUT6**) dupliziert. Die Weiterleitung erfolgt ohne Verzögerung oder Datenmanipulation – der Wert des eingehenden Adapters wird auf jedem der sechs Ausgänge direkt verfügbar gemacht. Der Baustein besitzt keine eigene Zustandslogik oder Ereignissteuerung; er arbeitet rein passiv und leitet das Signal durch.

Technische Besonderheiten
- **Generischer Typ**: Der Baustein ist als generischer Funktionsblock (`GEN_AW_SPLIT`) implementiert, was eine flexible Wiederverwendung in verschiedenen Projekten erlaubt.
- **Typ-Hash**: Die Type-Hash-Eigenschaft ist leer, was bedeutet, dass bei der Instanziierung keine automatische Überprüfung auf Typidentität erfolgt.
- **Keine Event- oder Daten-Ports**: Die gesamte Signalweitergabe erfolgt ausschließlich über Adapter. Ereignisse werden nicht verarbeitet; daher eignet sich der Baustein nur für reine Datenweitergabe ohne zeitliche Steuerung.

Zustandsübersicht
Der Funktionsblock besitzt keine internen Zustände. Er ist ein reiner Split-Baustein ohne sequenzielle Logik. Das Verhalten ist deterministisch und jederzeit identisch: Das Eingangssignal wird auf alle Ausgänge kopiert.

Anwendungsszenarien
- **Verteilung eines analogen Messwerts** an mehrere Verbraucher (z. B. Anzeigen, Regler, Aufzeichnungssysteme)
- **Aufteilung eines digitalen Steuersignals** auf parallele Aktor– oder Sensor-Schnittstellen
- **Sternförmige Signalverteilung** in modularen Steuerungsarchitekturen, bei denen eine Quelle mehrere Senken versorgt

Vergleich mit ähnlichen Bausteinen
Im Vergleich zu einem **AW_SPLIT_2** oder **AW_SPLIT_4** bietet der vorliegende Baustein eine höhere Anzahl von Ausgängen (6). Damit eignet er sich besonders für Systeme mit vielen parallelen Empfängern. Alternativ könnte eine Kaskadierung mehrerer kleiner Split-Bausteine verwendet werden, was jedoch die Übersichtlichkeit und Performance beeinträchtigen kann. Der **AW_SPLIT_6** ist als Komplettlösung für diese Anforderung optimiert.

- `AW_SPLIT_6_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der **AW_SPLIT_6** ist ein einfacher, aber nützlicher Funktionsblock zur Vervielfältigung eines unidirektionalen AW-Signals auf sechs Ausgänge. Durch seine generische Natur und die Adapter-basierte Schnittstelle fügt er sich nahtlos in eine 4diac-Umgebung ein. Er empfiehlt sich für alle Anwendungen, bei denen ein einzelnes Signal ohne Umwege mehrfach benötigt wird.

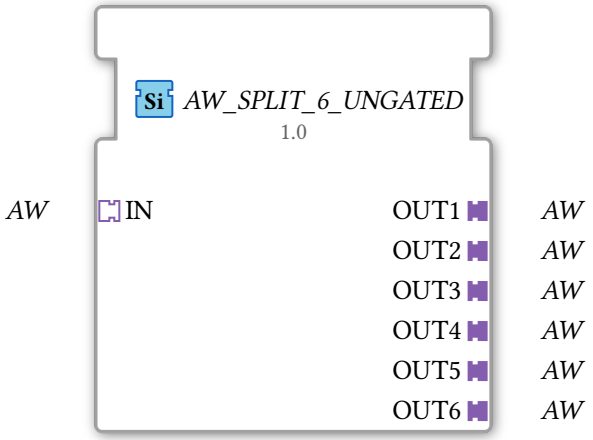
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AW_SPLIT_6_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AW_SPLIT_6`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

Bild nicht verfügbar



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AW_SPLIT_6_UNGATED** dient dazu, einen eingehenden unidirektionalen **AW**-Adapter auf sechs identische Ausgangs-Adapter aufzuteilen. Er ist als generischer Baustein konzipiert, der in der 4diac-IDE zur Verteilung von AW-Signalen verwendet wird. Der Baustein ist für Anwendungen gedacht, bei denen ein Signalwert – z. B. ein analoger oder digitaler Wert – mehrfach parallel benötigt wird, ohne dass eine manuelle Vervielfältigung der Verbindungen erfolgen muss.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AW`	IN	Socket	Eingangs-Adapter für das zu verteilende AW-Signal
`adapter::types::unidirectional::AW`	OUT1 – OUT6	Plugs	Sechs Ausgangs-Adapter, die jeweils eine identische Kopie des Eingangssignals bereitstellen

Funktionsweise

Der Baustein empfängt über den Socket **IN** ein unidirektionales **AW**-Signal. Dieses Signal wird intern auf alle sechs Ausgangs-Plugs (**OUT1** bis **OUT6**) dupliziert. Die Weiterleitung erfolgt ohne Verzögerung oder Datenmanipulation – der Wert des eingehenden Adapters wird auf jedem der sechs Ausgänge direkt verfügbar gemacht. Der Baustein besitzt keine eigene Zustandslogik oder Ereignissteuerung; er arbeitet rein passiv und leitet das Signal durch.

Technische Besonderheiten

- **Generischer Typ**: Der Baustein ist als generischer Funktionsblock (`GEN_AW_SPLIT`) implementiert, was eine flexible Wiederverwendung in verschiedenen Projekten erlaubt.
- **Typ-Hash**: Die Type-Hash-Eigenschaft ist leer, was bedeutet, dass bei der Instanziierung keine automatische Überprüfung auf Typidentität erfolgt.
- **Keine Event- oder Daten-Ports**: Die gesamte Signalweitergabe erfolgt ausschließlich über Adapter. Ereignisse werden nicht verarbeitet; daher eignet sich der Baustein nur für reine Datenweitergabe ohne zeitliche Steuerung.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Er ist ein reiner Split-Baustein ohne sequenzielle Logik. Das Verhalten ist deterministisch und jederzeit identisch: Das Eingangssignal wird auf alle Ausgänge kopiert.

Anwendungsszenarien

- **Verteilung eines analogen Messwerts** an mehrere Verbraucher (z. B. Anzeigen, Regler, Aufzeichnungssysteme)
- **Aufteilung eines digitalen Steuersignals** auf parallele Aktor– oder Sensor-Schnittstellen
- **Sternförmige Signalverteilung** in modularen Steuerungsarchitekturen, bei denen eine Quelle mehrere Senken versorgt

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem **AW_SPLIT_2** oder **AW_SPLIT_4** bietet der vorliegende Baustein eine höhere Anzahl von Ausgängen (6). Damit eignet er sich besonders für Systeme mit vielen parallelen Empfängern. Alternativ könnte eine Kaskadierung mehrerer kleiner Split-Bausteine verwendet werden, was jedoch die Übersichtlichkeit und Performance beeinträchtigen kann. Der **AW_SPLIT_6_UNGATED** ist als Komplettlösung für diese Anforderung optimiert.

- `AW_SPLIT_6_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- `AW_SPLIT_6`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

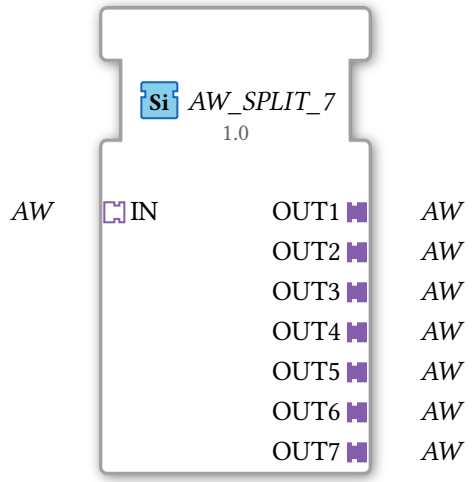
Fazit

Der **AW_SPLIT_6_UNGATED** ist ein einfacher, aber nützlicher Funktionsblock zur Vervielfältigung eines unidirektionalen AW-Signals auf sechs Ausgänge. Durch seine generische Natur und die Adapter-basierte Schnittstelle fügt er sich nahtlos in eine 4diac-Umgebung ein. Er empfiehlt sich für alle Anwendungen, bei denen ein einzelnes Signal ohne Umwege mehrfach benötigt wird.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AW_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AW_SPLIT_7` dient dazu, einen einzelnen eingehenden AW-Adapter (unidirektional) auf sieben identische AW-Adapter-Ausgänge zu verteilen. Er ist als generischer Baustein ausgelegt und ermöglicht eine einfache Verteilung von Adapterverbindungen ohne Datenverarbeitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine

Ereignis-Ausgänge

- Keine

Daten-Eingänge

- Keine

Daten-Ausgänge

- Keine

Adapter

Typ	Name	Richtung	Beschreibung

adapter::types::unidirectional::AW	IN	Socket (Eingang)	Eingehender AW-Adapter, der auf die Ausgänge verteilt wird.
adapter::types::unidirectional::AW	OUT1 ... OUT7	Plug (Ausgang)	Sieben ausgehende Adapter, die jeweils die gleichen Daten wie der Eingang bereitstellen.

Funktionsweise

Der Baustein besitzt keine eigenen Ereignis- oder Datenports. Er wirkt als reine Adaptervervielfältigung: Der eingehende AW-Adapter (`IN`) wird intern ohne Verarbeitung auf alle sieben Ausgangs-Adapter (`OUT1` bis `OUT7`) durchgeschleift. Jeder angeschlossene Baustein an einem der Ausgänge erhält somit exakt die gleiche Instanz des AW-Adapters.

Die Implementierung ist generisch (Generic-FB), sodass der tatsächliche Adaptertyp durch Parameter dynamisch festgelegt werden kann. Dadurch eignet sich der Baustein für beliebige unidirektionale AW-Schnittstellen.

Technische Besonderheiten

- **Generischer Typ**: Der FB nutzt den Mechanismus generischer Funktionsblöcke, um den konkreten Adaptertyp zur Laufzeit oder beim Kompilieren zu bestimmen. Der Platzhalter `GEN_AW_SPLIT` wird durch den tatsächlichen Adapter ersetzt.
- **Reine Adapter-Verteilung**: Es findet keine Datenkopie oder Serialisierung statt – die Referenzen auf den AW-Adapter werden weitergegeben.
- **Keine Ereignissteuerung** erforderlich: Der Baustein arbeitet asynchron und ohne zeitliche Abhängigkeiten, da Adapter ohnehin datenspezifisch verbunden sind.

Zustandsübersicht

Der `AW_SPLIT_7` verfügt über keinen internen Zustandsautomaten. Das Verhalten ist deterministisch und besteht ausschließlich aus der Durchleitung des Adapters. Eine Zustandsmaschine ist daher nicht erforderlich.

Anwendungsszenarien

- **Verteilen von Sensoradaptern** an mehrere Auswerte- oder Steuereinheiten in einer Automatisierungsachse.
- **Aufteilung eines Bus-Signals** in mehrere parallele Verbindungen (z. B. für redundante Überwachung).
- **Kaskadierung von Adaptern** in einem IEC 61499-System, wenn mehrere Funktionsblöcke auf den gleichen Datenstrom zugreifen müssen.

Vergleich mit ähnlichen Bausteinen

- **SPLIT-Bausteine mit begrenzter Ausgangsanzahl**: Andere Split-FBs (z. B. `SPLIT_2`, `SPLIT_4`) teilen Adapter auf weniger Kanäle auf – hier werden genau sieben Ausgänge bereitgestellt.
- **Adapter-Multiplexer**: Im Gegensatz zu Multiplexern, die zwischen mehreren Eingängen wählen, verteilt der `AW_SPLIT_7` einen Eingang ohne Selektion.
- **Daten-Split-FBs**: Falls der Baustein Daten- und Ereignisports hätte, müsste er Daten kopieren. Adapter-basierte Splits arbeiten effizienter, da nur Referenzen weitergegeben werden.
- `AW_SPLIT_7_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

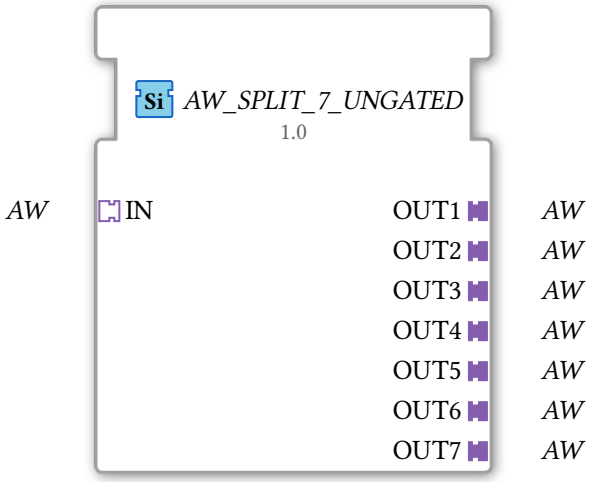
Fazit

Der `AW_SPLIT_7` ist ein einfacher, aber nützlicher generischer Funktionsblock zur Vervielfältigung eines unidirektionalen AW-Adapters auf sieben Ausgänge. Seine Stärken liegen in der flexiblen Typanpassung

und der effizienten, ereignislosen Weiterleitung. Er eignet sich besonders für modulare Automatisierungssysteme, bei denen ein Adapter mehrfach genutzt werden muss.

AW_SPLIT_7_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AW_SPLIT_7`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AW_SPLIT_7_UNGATED` dient dazu, einen einzelnen eingehenden AW-Adapter (unidirektional) auf sieben identische AW-Adapter-Ausgänge zu verteilen. Er ist als generischer Baustein ausgelegt und ermöglicht eine einfache Verteilung von Adapterverbindungen ohne Datenverarbeitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine

Ereignis-Ausgänge

- Keine

Daten-Eingänge

- Keine

Daten-Ausgänge

- Keine

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AW`	IN	Socket (Eingang)	Eingehender AW-Adapter, der auf die Ausgänge verteilt wird.

adapter::types::unidirectional::AW	OUT1 ... OUT7	Plug (Ausgang)	Sieben ausgehende Adapter, die jeweils die gleichen Daten wie der Eingang bereitstellen.

Funktionsweise

Der Baustein besitzt keine eigenen Ereignis- oder Datenports. Er wirkt als reine Adaptervervielfältigung: Der eingehende AW-Adapter (`IN`) wird intern ohne Verarbeitung auf alle sieben Ausgangs-Adapter (`OUT1` bis `OUT7`) durchgeschleift. Jeder angeschlossene Baustein an einem der Ausgänge erhält somit exakt die gleiche Instanz des AW-Adapters.

Die Implementierung ist generisch (Generic-FB), sodass der tatsächliche Adaptertyp durch Parameter dynamisch festgelegt werden kann. Dadurch eignet sich der Baustein für beliebige unidirektionale AW-Schnittstellen.

Technische Besonderheiten

- **Generischer Typ**: Der FB nutzt den Mechanismus generischer Funktionsblöcke, um den konkreten Adaptertyp zur Laufzeit oder beim Kompilieren zu bestimmen. Der Platzhalter `GEN_AW_SPLIT` wird durch den tatsächlichen Adapter ersetzt.
- **Reine Adapter-Verteilung**: Es findet keine Datenkopie oder Serialisierung statt – die Referenzen auf den AW-Adapter werden weitergegeben.
- **Keine Ereignissteuerung** erforderlich: Der Baustein arbeitet asynchron und ohne zeitliche Abhängigkeiten, da Adapter ohnehin datenspezifisch verbunden sind.

Zustandsübersicht

Der `AW_SPLIT_7_UNGATED` verfügt über keinen internen Zustandsautomaten. Das Verhalten ist deterministisch und besteht ausschließlich aus der Durchleitung des Adapters. Eine Zustandsmaschine ist daher nicht erforderlich.

Anwendungsszenarien

- **Verteilen von Sensoradaptern** an mehrere Auswerte- oder Steuereinheiten in einer Automatisierungsachse.
- **Aufteilung eines Bus-Signals** in mehrere parallele Verbindungen (z. B. für redundante Überwachung).
- **Kaskadierung von Adaptern** in einem IEC 61499-System, wenn mehrere Funktionsblöcke auf den gleichen Datenstrom zugreifen müssen.

Vergleich mit ähnlichen Bausteinen

- **SPLIT-Bausteine mit begrenzter Ausgangsanzahl**: Andere Split-FBs (z. B. `SPLIT_2`, `SPLIT_4`) teilen Adapter auf weniger Kanäle auf – hier werden genau sieben Ausgänge bereitgestellt.
- **Adapter-Multiplexer**: Im Gegensatz zu Multiplexern, die zwischen mehreren Eingängen wählen, verteilt der `AW_SPLIT_7_UNGATED` einen Eingang ohne Selektion.
- **Daten-Split-FBs**: Falls der Baustein Daten- und Ereignisports hätte, müsste er Daten kopieren. Adapter-basierte Splits arbeiten effizienter, da nur Referenzen weitergegeben werden.
- **`AW_SPLIT_7`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

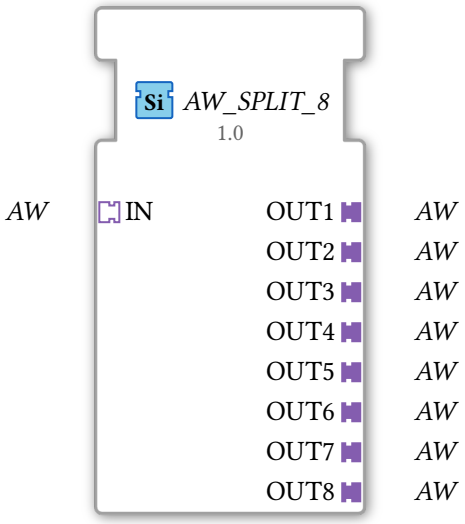
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AW_SPLIT_7_UNGATED` ist ein einfacher, aber nützlicher generischer Funktionsblock zur Vervielfältigung eines unidirektionalen AW-Adapters auf sieben Ausgänge. Seine Stärken liegen in der flexiblen Typanpassung und der effizienten, ereignislosen Weiterleitung. Er eignet sich besonders für modulare Automatisierungssysteme, bei denen ein Adapter mehrfach genutzt werden muss.

AW_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AW_SPLIT_8** ist ein generischer Baustein, der ein eingehendes unidirektionales AW-Adapter-Signal auf acht identische Ausgangssignale aufteilt. Der Baustein wurde als generischer FB (generic FB) implementiert und kann zur Laufzeit mit einem konkreten Typnamen und Typprüfcode versehen werden. Er dient als 1:8-Verteiler für AW-Datenströme und wird typischerweise in der Automatisierungstechnik eingesetzt, wenn ein Signal gleichzeitig an mehrere Senken weitergeleitet werden muss.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden. Die Kommunikation erfolgt ausschließlich über die Adapter-Schnittstellen.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden. Die Datenweitergabe erfolgt über die Adapter-Plugs.

Daten-Eingänge

Keine Daten-Eingänge vorhanden. Alle Eingangsdaten werden über den Socket-Adapter `IN` bereitgestellt.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden. Die Ausgangsdaten werden über die Plug-Adapter `OUT1` bis `OUT8` ausgegeben.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AW`	`IN`	Socket (Eingang)	Quell-Adapter, der das aufzuteilende Signal liefert
`adapter::types::unidirectional::AW`	`OUT1`	Plug (Ausgang)	Erster Ausgang, identisch zum IN-Signal
`adapter::types::unidirectional::AW`	`OUT2`	Plug (Ausgang)	Zweiter Ausgang
`adapter::types::unidirectional::AW`	`OUT3`	Plug (Ausgang)	Dritter Ausgang
`adapter::types::unidirectional::AW`	`OUT4`	Plug (Ausgang)	Vierter Ausgang
`adapter::types::unidirectional::AW`	`OUT5`	Plug (Ausgang)	Fünfter Ausgang

adapter::types::unidirectional::AW	OUT6	Plug (Ausgang)	Sechster Ausgang
adapter::types::unidirectional::AW	OUT7	Plug (Ausgang)	Siebter Ausgang
adapter::types::unidirectional::AW	OUT8	Plug (Ausgang)	Achter Ausgang

Funktionsweise

Der Baustein verhält sich wie ein passiver Splitter. Sobald am Socket `IN` ein gültiges AW-Adapter-Signal anliegt (z. B. durch ein Ereignis aus dem Adapterprotokoll), wird dieses Signal simultan an alle acht Plug-Adapter (`OUT1`–`OUT8`) weitergegeben. Es erfolgt keine Datenänderung, Pufferung oder Verzögerung. Der Baustein ist ereignisgesteuert im Sinne des zugrundeliegenden Adapterprotokolls: Die Weitergabe wird durch den Adapter-Mechanismus der IEC 61499-2 ausgelöst.

Die Ausgänge liefern stets den identischen Dateninhalt wie der Eingang. Falls der Eingang keine gültigen Daten enthält, geben auch die Ausgänge keine Daten aus.

Technische Besonderheiten

- **Generischer Funktionsblock (generic FB):** Der Baustein wird als generisch deklariert und trägt die Attribute `eclipse4diac::core::GenericClassName` (Standardwert: `'GEN_AW_SPLIT'`) und `eclipse4diac::core::TypeHash` (Standardwert: `''`). Dies ermöglicht die späte Bindung an einen konkreten Funktionsblock-Typen, z. B. um typsicherere Instanzen zu erzeugen.
- **Keine eigenen Ereignisse oder Variablen:** Der Baustein besitzt keine expliziten Ereignis-Ein-/Ausgänge und keine Daten-Ein-/Ausgänge im klassischen Sinne. Die gesamte Kommunikation wird über die Adapter-Schnittstellen abgewickelt, was die Komplexität reduziert und die Wiederverwendbarkeit erhöht.
- **Unidirektionale Adapter:** Verwendet werden ausschließlich unidirektionale AW-Adapter. Eine Rückkopplung von den Ausgängen zum Eingang ist nicht vorgesehen.

Zustandsübersicht

Der Funktionsblock besitzt keine eigene Zustandsmaschine (z. B. ECC – Execution Control Chart). Sein Verhalten wird vollständig durch das Verhalten des verwendeten Adaptertyps `adapter::types::unidirectional::AW` bestimmt. Im einfachsten Fall gibt es zwei implizite Zustände:

- **Idle:** Kein gültiges Signal am Eingang. Alle Ausgänge sind inaktiv.
- **Active:** Ein gültiges Signal liegt am Eingang an und wird auf alle Ausgänge repliziert.

Eine genauere Zustandsbeschreibung ist aus den Adapterdefinitionen abzuleiten.

Anwendungsszenarien

- **Parallelversorgung mehrerer Verbraucher:** Ein Sensor- oder Steuerungssignal (z. B. Geschwindigkeitssollwert) soll gleichzeitig an mehrere Aktoren oder Subsysteme verteilt werden, ohne dass ein eigener Verteiler-Baustein programmiert werden muss.
- **Test- und Simulationsumgebungen:** Ein einzelnes Testsignal kann auf mehrere Analyse- oder Logging-Kanäle aufgeschaltet werden.
- **Datenvervielfachung in hierarchischen Steuerungen:** In modularen Anlagen kann ein zentraler Steuerungsdatenstrom an mehrere dezentrale Einheiten (z. B. Feldgeräte) weitergegeben werden.

Vergleich mit ähnlichen Bausteinen

- **AW_SPLIT_2, AW_SPLIT_4:** Analog aufgebaute Splitter mit weniger Ausgängen (2 bzw. 4). Die Anzahl der Ausgänge ist der einzige Unterschied. Für Anwendungen mit genau 8 benötigten Ausgängen ist `AW_SPLIT_8` die passende Wahl.
- **Allgemeine Splitter (z. B. für andere Adaptertypen):** Es existieren Splitter für verschiedene Adapter-Datentypen (z. B. `BOOL_SPLIT`, `INT_SPLIT`), die analoge Struktur aufweisen, aber unterschiedliche Adapter-Schnittstellen besitzen.
- **Multicast-Bausteine:** Komplexere Bausteine können zusätzlich Filter-, Priorisierungs- oder Pufferfunktionen bieten; `AW_SPLIT_8` ist absichtlich minimalistisch gehalten, um keine unerwünschten Seiteneffekte zu erzeugen.
- `AW_SPLIT_8_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

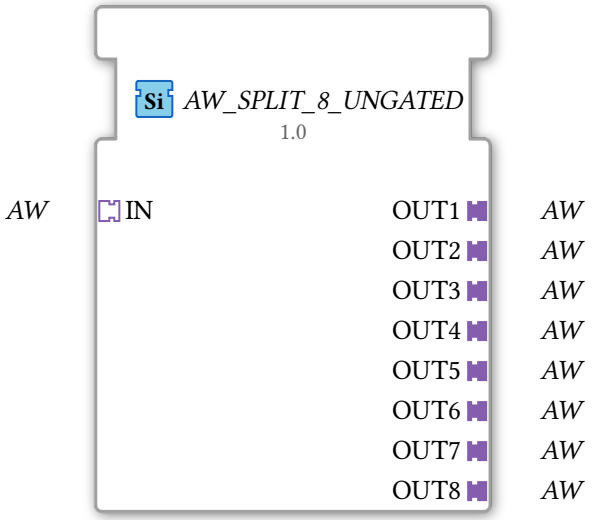
Der Baustein `AW_SPLIT_8` ist ein einfacher, aber nützlicher generischer 1:8-Splitter für unidirektionale AW-Adapter. Er erfüllt die grundlegende Anforderung, ein eingehendes Signal ohne Verzögerung oder Veränderung auf acht Ausgänge zu verteilen. Durch seine generische Auslegung kann er in verschiedenen Kontexten eingesetzt werden, in denen eine typsichere Vervielfältigung von Adapterdatenströmen benötigt wird. Die strikte Einhaltung des IEC 61499-2-Standards und die Verwendung von Adaptern erleichtern die Integration in bestehende 4diac-Projekte.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AW_SPLIT_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AW_SPLIT_8`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AW_SPLIT_8_UNGATED** ist ein generischer Baustein, der ein eingehendes unidirektionales AW-Adapter-Signal auf acht identische Ausgangssignale aufteilt. Der Baustein wurde als generischer FB (generic FB) implementiert und kann zur Laufzeit mit einem konkreten Typnamen und Typprüfcode versehen werden. Er dient als 1:8-Verteiler für AW-Datenströme und wird typischerweise in der Automatisierungstechnik eingesetzt, wenn ein Signal gleichzeitig an mehrere Senken weitergeleitet werden muss.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden. Die Kommunikation erfolgt ausschließlich über die Adapter-Schnittstellen.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden. Die Datenweitergabe erfolgt über die Adapter-Plugs.

Daten-Eingänge

Keine Daten-Eingänge vorhanden. Alle Eingangsdaten werden über den Socket-Adapter `IN` bereitgestellt.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden. Die Ausgangsdaten werden über die Plug-Adapter `OUT1` bis `OUT8` ausgegeben.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AW`	`IN`	Socket (Eingang)	Quell-Adapter, der das aufzuteilende Signal liefert
`adapter::types::unidirectional::AW`	`OUT1`	Plug (Ausgang)	Erster Ausgang, identisch zum IN-Signal
`adapter::types::unidirectional::AW`	`OUT2`	Plug (Ausgang)	Zweiter Ausgang
`adapter::types::unidirectional::AW`	`OUT3`	Plug (Ausgang)	Dritter Ausgang
`adapter::types::unidirectional::AW`	`OUT4`	Plug (Ausgang)	Vierter Ausgang
`adapter::types::unidirectional::AW`	`OUT5`	Plug (Ausgang)	Fünfter Ausgang
`adapter::types::unidirectional::AW`	`OUT6`	Plug (Ausgang)	Sechster Ausgang
`adapter::types::unidirectional::AW`	`OUT7`	Plug (Ausgang)	Siebter Ausgang
`adapter::types::unidirectional::AW`	`OUT8`	Plug (Ausgang)	Achter Ausgang

Funktionsweise

Der Baustein verhält sich wie ein passiver Splitter. Sobald am Socket `IN` ein gültiges AW-Adapter-Signal anliegt (z. B. durch ein Ereignis aus dem Adapterprotokoll), wird dieses Signal simultan an alle acht Plug-Adapter (`OUT1`–`OUT8`) weitergegeben. Es erfolgt keine Datenänderung, Pufferung oder Verzögerung. Der Baustein ist ereignisgesteuert im Sinne des zugrundeliegenden Adapterprotokolls: Die Weitergabe wird durch den Adapter-Mechanismus der IEC 61499-2 ausgelöst.

Die Ausgänge liefern stets den identischen Dateninhalt wie der Eingang. Falls der Eingang keine gültigen Daten enthält, geben auch die Ausgänge keine Daten aus.

Technische Besonderheiten

- **Generischer Funktionsblock (generic FB):** Der Baustein wird als generisch deklariert und trägt die Attribute `eclipse4diac::core::GenericClassName` (Standardwert: `'GEN_AW_SPLIT'`) und `eclipse4diac::core::TypeHash` (Standardwert: `''`). Dies ermöglicht die späte Bindung an einen konkreten Funktionsblock-Typen, z. B. um typsicherere Instanzen zu erzeugen.
- **Keine eigenen Ereignisse oder Variablen:** Der Baustein besitzt keine expliziten Ereignis-Ein-/Ausgänge und keine Daten-Ein-/Ausgänge im klassischen Sinne. Die gesamte Kommunikation wird über die Adapter-Schnittstellen abgewickelt, was die Komplexität reduziert und die Wiederverwendbarkeit erhöht.
- **Unidirektionale Adapter:** Verwendet werden ausschließlich unidirektionale AW-Adapter. Eine Rückkopplung von den Ausgängen zum Eingang ist nicht vorgesehen.

Zustandsübersicht

Der Funktionsblock besitzt keine eigene Zustandsmaschine (z. B. ECC – Execution Control Chart). Sein Verhalten wird vollständig durch das Verhalten des verwendeten Adaptertyps `adapter::types::unidirectional::AW` bestimmt. Im einfachsten Fall gibt es zwei implizite Zustände:

- **Idle:** Kein gültiges Signal am Eingang. Alle Ausgänge sind inaktiv.
- **Active:** Ein gültiges Signal liegt am Eingang an und wird auf alle Ausgänge repliziert.

Eine genauere Zustandsbeschreibung ist aus den Adapterdefinitionen abzuleiten.

Anwendungsszenarien

- **Parallelversorgung mehrerer Verbraucher:** Ein Sensor- oder Steuerungssignal (z. B. Geschwindigkeitssollwert) soll gleichzeitig an mehrere Aktoren oder Subsysteme verteilt werden, ohne dass ein eigener Verteiler-Baustein programmiert werden muss.
- **Test- und Simulationsumgebungen:** Ein einzelnes Testsignal kann auf mehrere Analyse- oder Logging-Kanäle aufgeschaltet werden.
- **Datenvervielfachung in hierarchischen Steuerungen:** In modularen Anlagen kann ein zentraler Steuerungsdatenstrom an mehrere dezentrale Einheiten (z. B. Feldgeräte) weitergegeben werden.

Vergleich mit ähnlichen Bausteinen

- **AW_SPLIT_2, AW_SPLIT_4:** Analog aufgebaute Splitter mit weniger Ausgängen (2 bzw. 4). Die Anzahl der Ausgänge ist der einzige Unterschied. Für Anwendungen mit genau 8 benötigten Ausgängen ist `AW_SPLIT_8_UNGATED` die passende Wahl.
- **Allgemeine Splitter (z. B. für andere Adaptertypen):** Es existieren Splitter für verschiedene Adapter-Datentypen (z. B. `BOOL_SPLIT`, `INT_SPLIT`), die analoge Struktur aufweisen, aber unterschiedliche Adapter-Schnittstellen besitzen.
- **Multicast-Bausteine:** Komplexere Bausteine können zusätzlich Filter-, Priorisierungs- oder Pufferfunktionen bieten; `AW_SPLIT_8_UNGATED` ist absichtlich minimalistisch gehalten, um keine unerwünschten Seiteneffekte zu erzeugen.
- `AW_SPLIT_8`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

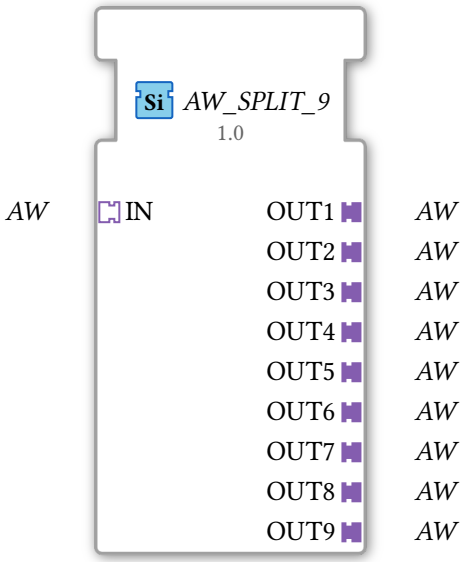
Fazit

Der Baustein `AW_SPLIT_8_UNGATED` ist ein einfacher, aber nützlicher generischer 1:8-Splitter für unidirektionale AW-Adapter. Er erfüllt die grundlegende Anforderung, ein eingehendes Signal ohne Verzögerung oder Veränderung auf acht Ausgänge zu verteilen. Durch seine generische Auslegung kann er in verschiedenen Kontexten eingesetzt werden, in denen eine typsichere Vervielfältigung von Adapterdatenströmen benötigt wird. Die strikte Einhaltung des IEC 61499-2-Standards und die Verwendung von Adaptern erleichtern die Integration in bestehende 4diac-Projekte.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AW_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AW_SPLIT_9` dient als generischer Verteiler für einen eingehenden Adapter vom Typ `AW` auf neun identisch ausgelegte Ausgangsadapter. Er ermöglicht die Mehrfachnutzung eines einzelnen Adaptersignals, ohne dass die ursprüngliche Datenquelle dupliziert werden muss. Die Bezeichnung „SPLIT 1 AW into 9 AW“ beschreibt die Kernfunktion: Ein AW-Eingang wird auf neun AW-Ausgänge verteilt.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AW`	`IN`	Socket (Eingang)	Eingehender AW-Adapter, der auf alle Ausgänge verteilt wird.
`adapter::types::unidirectional::AW`	`OUT1`	Plug (Ausgang)	Erster ausgehender AW-Adapter.
`adapter::types::unidirectional::AW`	`OUT2`	Plug (Ausgang)	Zweiter ausgehender AW-Adapter.
`adapter::types::unidirectional::AW`	`OUT3`	Plug (Ausgang)	Dritter ausgehender AW-Adapter.
`adapter::types::unidirectional::AW`	`OUT4`	Plug (Ausgang)	Vierter ausgehender AW-Adapter.
`adapter::types::unidirectional::AW`	`OUT5`	Plug (Ausgang)	Fünfter ausgehender AW-Adapter.
`adapter::types::unidirectional::AW`	`OUT6`	Plug (Ausgang)	Sechster ausgehender AW-Adapter.

adapter::types::unidirectional::AW	OUT7	Plug (Ausgang)	Siebter ausgehender AW-Adapter.
adapter::types::unidirectional::AW	OUT8	Plug (Ausgang)	Achter ausgehender AW-Adapter.
adapter::types::unidirectional::AW	OUT9	Plug (Ausgang)	Neunter ausgehender AW-Adapter.

Funktionsweise

Der Baustein verfügt über keinerlei Verarbeitungslogik (keine ECC, keine Algorithmen). Er fungiert als reiner Verdrahtungshelfer: Das am Socket `IN` anliegende AW-Signal wird strukturell an alle neun Plug-Adapter `OUT1` bis `OUT9` weitergeleitet. Im IEC 61499-Modell bedeutet dies, dass alle Verbindungen, die an die Ausgänge angeschlossen sind, dasselbe Signal erhalten, das am Eingang anliegt. Eine zeitliche Steuerung oder Datenmanipulation findet nicht statt.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generische Klasse (`GEN_AW_SPLIT`) deklariert. Das ermöglicht die Wiederverwendung mit verschiedenen AW-Adapter-Typen, sofern die Grundstruktur (unidirektionaler AW-Adapter) erhalten bleibt.
- **Keine Ereignisse oder Daten**: Die gesamte Kommunikation erfolgt ausschließlich über Adapter-Schnittstellen. Dies vereinfacht die Handhabung in rein adapterbasierten Architekturen.
- **Keine Zustandslogik**: Der Baustein besitzt kein eigenes Verhalten; er ist ein reines „passives“ Element zur Signalkopierung.

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine, da keine Ereignisse oder Algorithmen definiert sind. Sein Verhalten ist rein strukturell: Solange der Eingangsadapter verbunden ist, sind alle Ausgänge stets aktiv – eine Zustandsänderung tritt nicht auf.

Anwendungsszenarien

- **Signalfächer**: Ein zentrales AW-Signal (z. B. ein Steuerbefehl oder Sensorwert) muss an mehrere nachgeschaltete Komponenten verteilt werden.
- **Redundante Verbindungen**: In sicherheitskritischen Systemen kann das gleiche AW-Signal an mehrere parallele Auswerteeinheiten gesendet werden.
- **Verdrahtungsvereinfachung**: Der FB ersetzt manuelle Stern-Verdrahtungen in der Applikation und verbessert die Übersichtlichkeit des Funktionsplans.

Vergleich mit ähnlichen Bausteinen

- **AW_SPLIT_N**: Ein generischer Split-Baustein mit variabler Anzahl von Ausgängen (z. B. `AW_SPLIT_4`, `AW_SPLIT_16`). `AW_SPLIT_9` ist eine spezielle Ausprägung für genau neun Ausgänge.
- **Daten-Split-Bausteine**: Bausteine, die Daten-Eingänge (z. B. `INT`, `REAL`) aufteilen. `AW_SPLIT_9` arbeitet ausschließlich auf Adapterebene und nicht auf einzelnen Datenpunkten.
- **Multiplexer (MUX)**: Ein Multiplexer wählt einen von mehreren Eingängen aus, während dieser Baustein einen Eingang auf viele Ausgänge verteilt (Fan-Out).
- **`AW_SPLIT_9_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

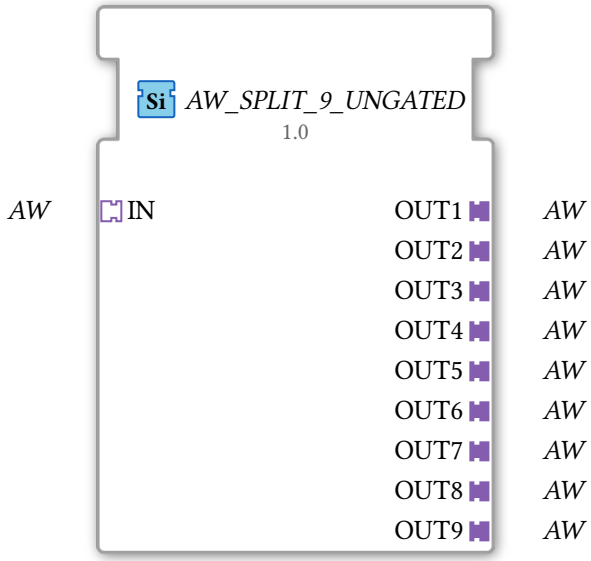
Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der `AW_SPLIT_9` ist ein einfacher, aber nützlicher generischer Funktionsblock für die Aufteilung eines unidirektionalen AW-Adapter-Signals auf neun parallele Ausgänge. Seine Struktur ist minimalistisch und rein passiv, was ihn zu einem zuverlässigen Werkzeug in adapterorientierten Steuerungsarchitekturen macht. Er benötigt weder Ereignisse noch Daten und kann ohne Nebenwirkungen in bestehende Applikationen integriert werden.

AW_SPLIT_9_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `AW_SPLIT_9`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AW_SPLIT_9_UNGATED` dient als generischer Verteiler für einen eingehenden Adapter vom Typ `AW` auf neun identisch ausgelegte Ausgangsadapter. Er ermöglicht die Mehrfachnutzung eines einzelnen Adaptersignals, ohne dass die ursprüngliche Datenquelle dupliziert werden muss. Die Bezeichnung „SPLIT 1 AW into 9 AW“ beschreibt die Kernfunktion: Ein AW-Eingang wird auf neun AW-Ausgänge verteilt.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AW`	`IN`	Socket (Eingang)	Eingehender AW-Adapter, der auf alle Ausgänge verteilt wird.

adapter::types::unidirectional::AW	OUT1	Plug (Ausgang)	Erster ausgehender AW-Adapter.
adapter::types::unidirectional::AW	OUT2	Plug (Ausgang)	Zweiter ausgehender AW-Adapter.
adapter::types::unidirectional::AW	OUT3	Plug (Ausgang)	Dritter ausgehender AW-Adapter.
adapter::types::unidirectional::AW	OUT4	Plug (Ausgang)	Vierter ausgehender AW-Adapter.
adapter::types::unidirectional::AW	OUT5	Plug (Ausgang)	Fünfter ausgehender AW-Adapter.
adapter::types::unidirectional::AW	OUT6	Plug (Ausgang)	Sechster ausgehender AW-Adapter.
adapter::types::unidirectional::AW	OUT7	Plug (Ausgang)	Siebter ausgehender AW-Adapter.
adapter::types::unidirectional::AW	OUT8	Plug (Ausgang)	Achter ausgehender AW-Adapter.
adapter::types::unidirectional::AW	OUT9	Plug (Ausgang)	Neunter ausgehender AW-Adapter.

Funktionsweise

Der Baustein verfügt über keinerlei Verarbeitungslogik (keine ECC, keine Algorithmen). Er fungiert als reiner Verdrahtungshelfer: Das am Socket `IN` anliegende AW-Signal wird strukturell an alle neun Plug-Adapter `OUT1` bis `OUT9` weitergeleitet. Im IEC 61499-Modell bedeutet dies, dass alle Verbindungen, die an die Ausgänge angeschlossen sind, dasselbe Signal erhalten, das am Eingang anliegt. Eine zeitliche Steuerung oder Datenmanipulation findet nicht statt.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generische Klasse (`GEN_AW_SPLIT`) deklariert. Das ermöglicht die Wiederverwendung mit verschiedenen AW-Adapter-Typen, sofern die Grundstruktur (unidirektionaler AW-Adapter) erhalten bleibt.
- **Keine Ereignisse oder Daten**: Die gesamte Kommunikation erfolgt ausschließlich über Adapter-Schnittstellen. Dies vereinfacht die Handhabung in rein adapterbasierten Architekturen.
- **Keine Zustandslogik**: Der Baustein besitzt kein eigenes Verhalten; er ist ein reines „passives“ Element zur Signalkopierung.

Zustandsübersicht

Der Baustein besitzt keine eigene Zustandsmaschine, da keine Ereignisse oder Algorithmen definiert sind. Sein Verhalten ist rein strukturell: Solange der Eingangsadapter verbunden ist, sind alle Ausgänge stets aktiv – eine Zustandsänderung tritt nicht auf.

Anwendungsszenarien

- **Signalfächer**: Ein zentrales AW-Signal (z. B. ein Steuerbefehl oder Sensorwert) muss an mehrere nachgeschaltete Komponenten verteilt werden.
- **Redundante Verbindungen**: In sicherheitskritischen Systemen kann das gleiche AW-Signal an mehrere parallele Auswerteeinheiten gesendet werden.
- **Verdrahtungsvereinfachung**: Der FB ersetzt manuelle Stern-Verdrahtungen in der Applikation und verbessert die Übersichtlichkeit des Funktionsplans.

Vergleich mit ähnlichen Bausteinen

- **AW_SPLIT_N**: Ein generischer Split-Baustein mit variabler Anzahl von Ausgängen (z. B. `AW_SPLIT_4`, `AW_SPLIT_16`). `AW_SPLIT_9_UNGATED` ist eine spezielle Ausprägung für genau neun Ausgänge.
- **Daten-Split-Bausteine**: Bausteine, die Daten-Eingänge (z. B. `INT`, `REAL`) aufteilen. `AW_SPLIT_9_UNGATED` arbeitet ausschließlich auf Adapterebene und nicht auf einzelnen Datenpunkten.
- **Multiplexer (MUX)**: Ein Multiplexer wählt einen von mehreren Eingängen aus, während dieser Baustein einen Eingang auf viele Ausgänge verteilt (Fan-Out).
- `AW_SPLIT_9`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

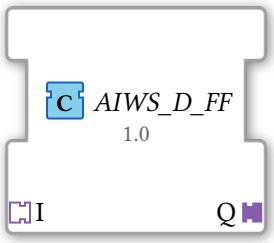
Fazit

Der `AW_SPLIT_9_UNGATED` ist ein einfacher, aber nützlicher generischer Funktionsblock für die Aufteilung eines unidirektionalen AW-Adapter-Signals auf neun parallele Ausgänge. Seine Struktur ist minimalistisch und rein passiv, was ihn zu einem zuverlässigen Werkzeug in adapterorientierten Steuerungsarchitekturen macht. Er benötigt weder Ereignisse noch Daten und kann ohne Nebenwirkungen in bestehende Applikationen integriert werden.

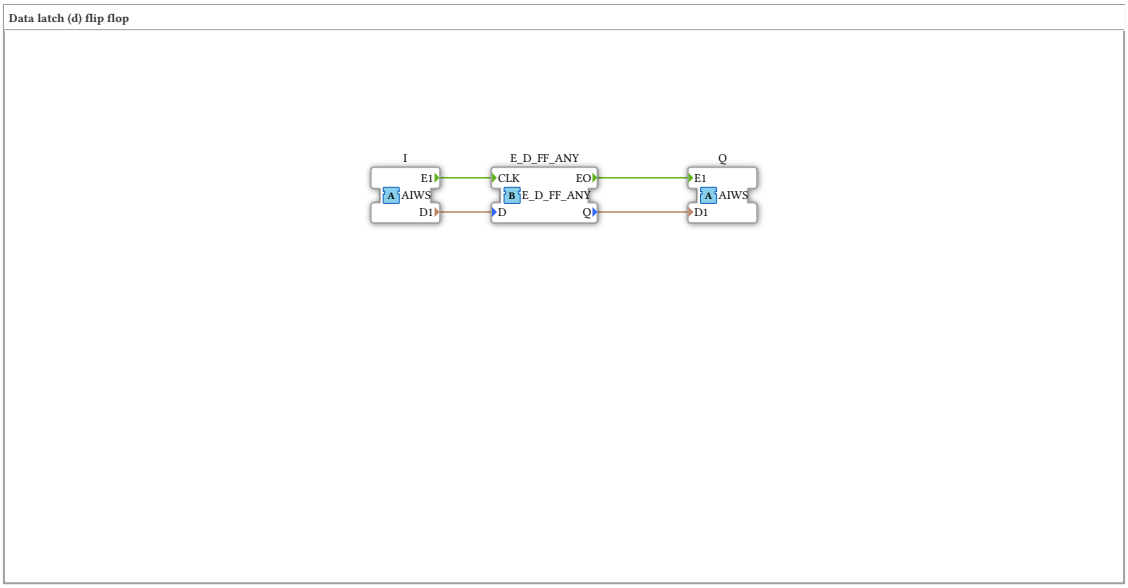
WSTRING

AIWS_D_FF

Value to latch – *AIWS*



AIWS – Latched value



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock AIWS_D_FF realisiert ein data latch (D)-Flipflop, das einen eingehenden Datenwert speichert und bei einem Ereignis an den Ausgang weitergibt. Die Kommunikation erfolgt ausschließlich über Adapter, sodass der Baustein nahtlos in modulare Systeme integriert werden kann.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Datentyp	Beschreibung
I.E1	Ereignis (über Adapter)	Ereignis, das das Übernehmen des Datenwertes auslöst

Ereignis-Ausgänge

Name	Datentyp	Beschreibung
Q.E1	Ereignis (über Adapter)	Ereignis, das den gespeicherten Datenwert anzeigt

Daten-Eingänge

Name	Datentyp	Beschreibung
I.D1	ANY (über Adapter)	Datenwert, der beim Eintreffen von I.E1 übernommen wird

Daten-Ausgänge

Name	Datentyp	Beschreibung
Q.D1	ANY (über Adapter)	Gespeicherter Datenwert, der mit Q.E1 ausgegeben wird

Adapter

Name	Typ	Richtung	Beschreibung
I	`adapter::types::unidirectional::AIWS`	Socket (Eingang)	Liefert den zu latchenden Wert sowie das auslösende Ereignis
Q	`adapter::types::unidirectional::AIWS`	Plug (Ausgang)	Gibt den gespeicherten Wert mit einem Ereignis aus

Funktionsweise

Der AIWS_D_FF kapselt einen internen `E_D_FF_ANY`-Baustein. Der eingehende Adapter `I` liefert über `I.E1` ein Ereignis, das den am Datenport `I.D1` anliegenden Wert übernimmt. Der gespeicherte Wert wird anschließend über `Q.D1` ausgegeben, begleitet von einem Ereignis auf `Q.E1`. Somit verhält sich der Baustein wie ein klassisches getaktetes D-Flipflop, jedoch mit einer vollständig adapterbasierten Schnittstelle.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle:** Alle Ein- und Ausgänge sind über unidirektionale Adapter realisiert, was eine einfache Kapselung und Wiederverwendung in unterschiedlichen Kontexten ermöglicht.
- **Interner Standard-FB:** Die Implementierung nutzt den IEC-61499-Standardbaustein `E_D_FF_ANY`, der eine generische und bewährte Latch-Funktionalität bereitstellt.
- **Unidirektionaler Datenfluss:** Der Adaptertyp `AIWS` erlaubt nur eine Datenrichtung, was die Übersichtlichkeit und Fehlervermeidung in der Vernetzung verbessert.

Zustandsübersicht

Der AIWS_D_FF besitzt keine expliziten inneren Zustände auf der obersten Ebene. Der integrierte `E_D_FF_ANY` durchläuft zwei Phasen:

1. **Warten:** Er wartet auf ein Ereignis am Clock-Eingang.
2. **Übernahme & Ausgabe:** Bei einem Clock-Ereignis wird der Datenwert übernommen und ein Ausgangsereignis erzeugt.

Diese Abfolge wiederholt sich bei jedem eingehenden Ereignis.

Anwendungsszenarien

- **Datensynchronisation:** Puffern eines Sensorsignals, das erst bei einem bestimmten Triggerereignis weitergegeben werden soll.
- **Steuerungslogik:** Speichern eines Stellsignals in Abhängigkeit eines Takts, z. B. für zeitlich abgestimmte Aktorsteuerungen.
- **Modulare Systeme:** Einbinden in Adapter-basierte Architekturen, bei denen Funktionsblöcke über standardisierte Schnittstellen verbunden werden.

Vergleich mit ähnlichen Bausteinen

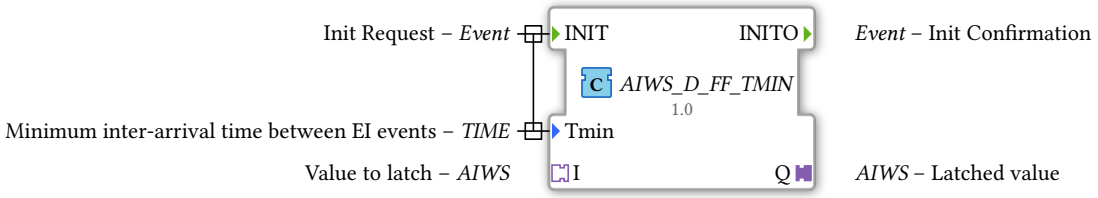
- **Einfaches D-Flipflop (z. B. `E_D_FF`):** Besitzt separate Ereignis- und Datenports, während der AIWS_D_FF die gleiche Logik über Adapter kapselt.
- **SR-Flipflop:** Speichert zwei Zustände (Set/Reset), wohingegen der AIWS_D_FF nur einen einzigen Datenwert übernimmt und keine Rücksetzfunktion bietet.

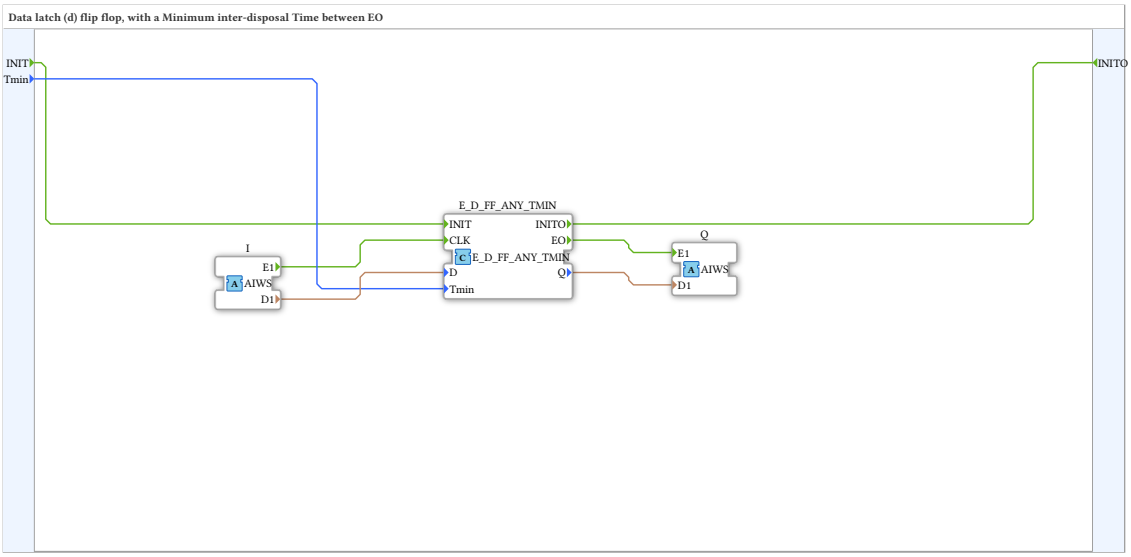
- **Registerbausteine:** Können mehrere Bits parallel speichern, während AIWS_D_FF auf einen einzelnen Datenwert (ANY) ausgelegt ist.

Fazit

Der AIWS_D_FF ist ein kompakter und modularer Funktionsblock zur Realisierung eines D-Flipflops in IEC-61499-Systemen. Durch die ausschließliche Nutzung von Adaptern wird eine hohe Flexibilität und Wiederverwendbarkeit erreicht. Er eignet sich besonders für Anwendungen, die eine entkoppelte, ereignisgesteuerte Datenübernahme erfordern.

AIWS_D_FF_TMIN





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AIWS_D_FF_TMIN** realisiert einen Daten-Latch (D-Flip-Flop) mit einer minimalen Wartezeit zwischen zwei aufeinanderfolgenden Ausgabeereignissen (Minimum inter-disposal Time). Er wird über Adapter-Schnittstellen (Typ: AIWS, unidirektional) angesprochen, die einen Wert aufnehmen und als gelatchten Wert wieder ausgeben. Die Steuerung erfolgt über das Ereignis des Eingabeadapters (CLK-Flanke), wobei die Ausgabe erst nach Ablauf einer einstellbaren Totzeit (Tmin) erfolgt.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar
INIT	EInit	Initialisierungsanforderung (mit Tmin-Parameter)

Ereignis-Ausgänge

Ereignis	Typ	Kommentar
INITO	EInit	Initialisierungsbestätigung

Daten-Eingänge

Name	Typ	Kommentar
Tmin	TIME	Mindestzeit zwischen zwei Ausgabeereignissen (inter-disposal time)

Daten-Ausgänge

Keine eigenen Datenausgänge – der gelatchte Wert wird ausschließlich über den Adapter **Q** bereitgestellt.

Adapter

Richtung	Name	Typ	Kommentar
Socket (Eingang)	I	adapter::types::unidirectional::AIWS	Wert, der gelatcht werden soll
Plug (Ausgang)	Q	adapter::types::unidirectional::AIWS	Gelatchter Wert

Funktionsweise

Der FB verwendet intern den Baustein `E_D_FF_ANY_TMIN` aus der IEC 61499-Bibliothek. Der Ablauf ist wie folgt:

1. Bei **INIT** wird die Mindestzeit `Tmin` übernommen und der Baustein initialisiert (`INITO` bestätigt den erfolgreichen Start).
2. Jedes eingehende Ereignis vom Adapter `I.E1` wird als Taktflanke (CLK) interpretiert.
3. Der Datenwert vom Adapter `I.D1` wird bei jeder CLK-Flanke in das interne Flip-Flop übernommen.
4. Der gelatchte Wert wird auf dem Ausgangsadapter `Q.D1` bereitgestellt.
5. Das zugehörige Ereignis `Q.E1` wird jedoch erst dann gesendet, wenn seit dem letzten EO-Ereignis die Zeit `Tmin` verstrichen ist (inter-disposal time). Damit wird die Ausgaberate begrenzt.

Die Verbindung im FBNetzwerk stellt sicher, dass Ereignisse und Daten korrekt zwischen dem Adapter `I`, dem internen FB und dem Adapter `Q` weitergeleitet werden.

Technische Besonderheiten

- **Adapterbasierte Ein-/Ausgabe:** Der Baustein verwendet ausschließlich Adapter des Typs `AIWS` (unidirektional) für die Signalverarbeitung. Dies erlaubt eine lose Kopplung und Wiederverwendung der Schnittstellen.
- **Interne Nutzung von `E_D_FF_ANY_TMIN`:** Die eigentliche Latch- und Timerlogik wird durch einen spezialisierten Baustein realisiert, der bereits eine minimale Zeit zwischen zwei EO-Ereignissen sicherstellt.
- **Initialisierung erforderlich:** Der Baustein muss vor der ersten Nutzung mit `INIT` initialisiert werden, dabei wird `Tmin` gesetzt.

Zustandsübersicht

Der FB besitzt im Wesentlichen zwei Betriebsmodi:

- **Initialisierungsphase:** Nach Empfang von `INIT` wird der interne Timer und das Flip-Flop zurückgesetzt. Nach Abschluss wird `INITO` gesendet.
- **Normalbetrieb:** Nach der Initialisierung wartet der Baustein auf CLK-Ereignisse (`I.E1`). Bei jedem CLK wird der Datenwert gelatcht. Ein EO-Ereignis (`Q.E1`) wird erst dann ausgelöst, wenn die seit dem letzten EO verstrichene Zeit ≥ **Tmin** ist. Falls die Bedingung nicht erfüllt ist, wird das EO verzögert.

Eine vollständige Zustandsmaschine ist im internen FB implementiert; der hier beschriebene Baustein bildet diese transparent ab.

Anwendungsszenarien

- **Sensorwert-Erfassung mit minimaler Wiederholrate:** In der Automatisierung soll ein analoger oder digitaler Messwert in definierten Mindestabständen an eine nachgelagerte Steuerung weitergegeben werden, um Überlastung oder unerwünschte Schaltspiele zu vermeiden.
- **Entprellung von Signalflanken:** Das Latch kann verwendet werden, um einen Eingangswert stabil zu halten, während die Ausgabe durch `Tmin` zeitlich entprellt wird.
- **Datenpuffer mit Zeitsteuerung:** In Kommunikationsprotokollen, bei denen Datenpakete nur mit einer minimalen Pause verschickt werden dürfen.

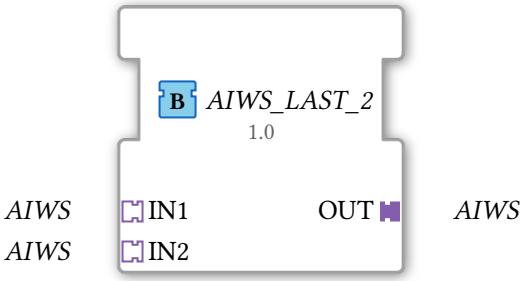
Vergleich mit ähnlichen Bausteinen

- **Normales D-Flip-Flop (z. B. `E_D_FF`):** Ein Standard-D-FF übernimmt den Eingang bei jeder CLK-Flanke und gibt sofort ein Ereignis aus. Es fehlt die zeitliche Begrenzung der Ausgaberate.
- **`AIWS_D_FF_TMIN`:** Erweitert das einfache D-FF um eine konfigurierbare Mindestzeit zwischen Ausgabeereignissen. Dadurch eignet er sich besonders für zeitkritische und ratebegrenzte Anwendungen.
- **Timer-basierte Latch-Bausteine (z. B. `E_D_FF_ANY_TMIN`):** Der hier verwendete interne FB ist die eigentliche Realisierung; `AIWS_D_FF_TMIN` kapselt diesen in eine adapterbasierte Schnittstelle und erleichtert so die Einbindung in eine IEC 61499-Architektur.

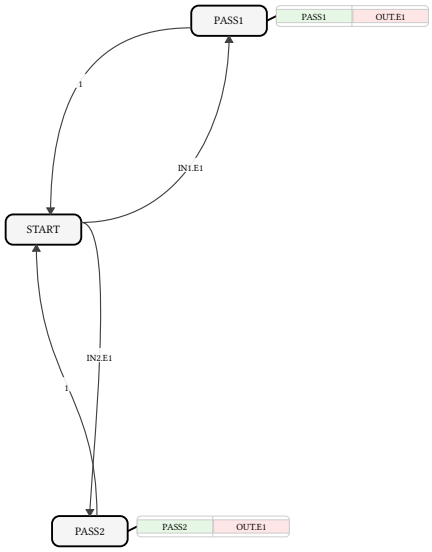
Fazit

Der Funktionsblock `AIWS_D_FF_TMIN` kombiniert die Funktion eines Daten-Latches mit einer programmierbaren minimalen Ausgabeverzögerung. Durch die adapterbasierte Schnittstellendefinition ist er flexibel in verteilten Automatisierungssystemen einsetzbar. Insbesondere für Anwendungen, bei denen Signalwechsel nur in bestimmten Zeitabständen erfolgen dürfen, bietet er eine einfache und robuste Lösung.

AIWS_LAST_2



ECC: AIWS_LAST_2



Einleitung

Der AIWS_LAST_2 Funktionsblock führt zwei gleichartige WSTRING-Adapter-Signale (**IN1**, **IN2**) auf einen gemeinsamen Ausgang (**OUT**) zusammen, nach dem Prinzip **Last Writer Wins**: welcher der beiden Sockets zuletzt ein Ereignis geschrieben hat, gewinnt, und sein Datenwert wird sofort und unverändert an OUT durchgereicht. Es handelt sich ausdrücklich nicht um ein naives Zusammenführen (z. B. Mittelwert oder ODER-Verknüpfung), sondern um eine reine Arbitrierung nach zeitlicher Reihenfolge.

Schnittstellenstruktur

Adapter

Eingangsadapter (Sockets):

- **IN1**: WSTRING-Adapter (unidirectional) - erste Signalquelle
- **IN2**: WSTRING-Adapter (unidirectional) - zweite Signalquelle

Ausgangsadapter (Plug):

- **OUT**: WSTRING-Adapter (unidirectional) - zusammengeführtes Signal

Funktionsweise

Der Baustein ist als einfacher Basic-FB mit einer 3-Zustands-ECC umgesetzt (START, PASS1, PASS2). Trifft an IN1 ein Ereignis ein, wechselt die ECC nach PASS1: der aktuelle Datenwert von IN1 wird nach OUT kopiert und das Adapter-Ereignis an OUT gesendet. Trifft an IN2 ein Ereignis ein, geschieht dasselbe symmetrisch über PASS2. Nach jeder Durchreichung kehrt die ECC sofort wieder nach START zurück und ist bereit für das nächste Ereignis von IN1 oder IN2.

Technische Besonderheiten

- Last-Writer-Wins-Semantik statt naiver Datenverknüpfung
- Einfacher Basic-FB (ECC mit 3 Zuständen), kein zusätzlicher Zwischenspeicher nötig
- Keine Signalverzögerung: der Wert wird im selben Zyklus durchgereicht, in dem das auslösende Ereignis eintrifft
- Beide Eingänge sind gleichberechtigt - es gibt keine feste Priorität außer der zeitlichen Reihenfolge

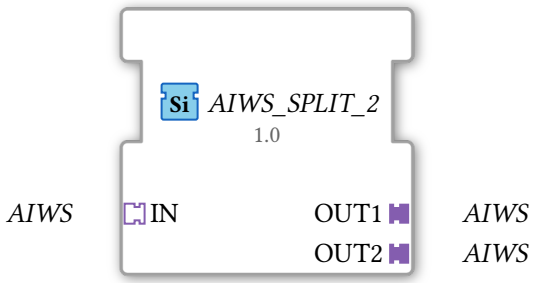
Anwendungsszenarien

- Zusammenführen zweier gleichwertiger WSTRING-Signalquellen zu einem einzigen nachgeschalteten Adapterpfad (z. B. zwei redundante Sensoren, oder ein manueller und ein automatischer Sollwertgeber)
- Arbitrierung zwischen zwei Bedienstellen, die denselben WSTRING-Wert schreiben dürfen
- Vereinfachung von Netzwerken, die sonst ein `F_SEL`/`E_RS`-Konstrukt inline verdrahten müssten

⚖️ Vergleich mit ähnlichen Bausteinen

AIWS_LAST_2 ist die Umkehrung von `AIWS_SPLIT_2`: AIWS_SPLIT_2 verteilt 1 Eingang auf 2 Ausgänge, AIWS_LAST_2 führt 2 Eingänge auf 1 Ausgang zusammen. Für Mischbetrieb zwischen einem Daten- und einem reinen Ereignisadapter (z. B. AX + AE) ist stattdessen `AX_AE_MERGE` gedacht, das nur das Ereignis mischt, ohne den Datenwert zu verändern.

AIWS_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AIWS_SPLIT_2** dient dazu, ein eingehendes AIWS-Adapter-Signal auf zwei identische Ausgänge aufzuteilen. Er ist als generischer Baustein ausgelegt und ermöglicht die Mehrfachverwendung eines AIWS-Signals innerhalb der gleichen Applikation, ohne dass eine separate Signalverteilung programmiert werden muss.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge
Keine Daten-Eingänge vorhanden.

Daten-Ausgänge
Keine Daten-Ausgänge vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	**IN**	`adapter::types::unidirectional::AIWS`	Eingehendes AIWS-Signal, das auf beide Ausgänge verteilt wird.
Plug (Ausgang)	**OUT1**	`adapter::types::unidirectional::AIWS`	Erster Ausgang – identisches Signal wie am Eingang.
Plug (Ausgang)	**OUT2**	`adapter::types::unidirectional::AIWS`	Zweiter Ausgang – identisches Signal wie am Eingang.

Funktionsweise
Der Baustein leitet das am Socket **IN** anliegende AIWS-Signal unverändert an beide Plugs **OUT1** und **OUT2** weiter. Es findet keine logische oder zeitliche Verarbeitung statt – die Aufteilung erfolgt rein passiv durch die Verdrahtung innerhalb des Funktionsblocks. Dadurch können zwei unabhängige Adapter-Instanzen gleichzeitig mit denselben Daten versorgt werden.

Technische Besonderheiten
- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AIWS_SPLIT`) deklariert, sodass er für verschiedene AIWS-Ausprägungen wiederverwendet werden kann.
- **Reine Adapterlogik**: Es werden weder Ereignisse noch Daten-E/A verwendet; die gesamte Kommunikation erfolgt ausschließlich über die Adapter-Schnittstellen.
- **Keine Zustandsabhängigkeit**: Der FB besitzt keine interne Zustandsmaschine – das Ausgangssignal folgt dem Eingangssignal ohne Verzögerung oder Pufferung.

Zustandsübersicht
Der Funktionsblock ist zustandslos. Es existieren keine internen Zustände oder Übergänge. Die Ausgänge sind zu jedem Zeitpunkt eine direkte Abbildung des Eingangs.

Anwendungsszenarien
- **Verteilen eines Sensorsignals**: Ein einzelner AIWS-Sensor (z. B. analoger Wert) soll von zwei unabhängigen Regelungs- oder Überwachungsbausteinen gleichzeitig gelesen werden.
- **Redundante Verarbeitung**: Bei Sicherheitsanwendungen kann dasselbe Signal an mehrere parallele Auswertefunktionen übergeben werden.
- **Logische Verzweigung**: Aufteilung eines Datenstroms für unterschiedliche Auswertungen oder Visualisierungen in einer 4diac-Applikation.

Vergleich mit ähnlichen Bausteinen
- **AIWS_SPLIT_3, AIWS_SPLIT_4**: Erweiterte Varianten mit drei bzw. vier Ausgängen – prinzipiell gleiche Funktionsweise.
- **SPLIT-Bausteine für andere Adaptertypen**: In 4diac existieren analoge Split-Bausteine für andere Adapter (z. B. `DINT_SPLIT`), die ein Datensignal auf mehrere Ausgänge verteilen – jedoch auf Datenebene und nicht über Adapter.

- **Ereignis-Split-Bausteine**: Bausteine wie `E_SPLIT` verteilen Ereignisse, nicht Datenströme – AIWS_SPLIT_2 adressiert spezifisch die AIWS-Adapterkommunikation.
- **`AIWS_SPLIT_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

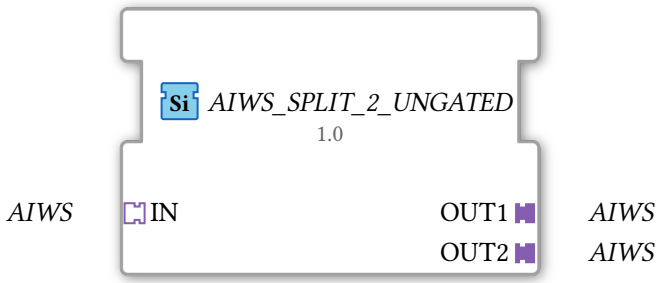
Der **AIWS_SPLIT_2** ist ein einfacher, aber nützlicher generischer Funktionsblock zur Vervielfältigung eines AIWS-Adaptersignals. Er reduziert den Verdrahtungsaufwand in Applikationen, die dasselbe analoge oder gemischte Signal mehrfach benötigen, und fügt sich durch seine passive, zustandslose Architektur transparent in bestehende Abläufe ein.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AIWS_SPLIT_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AIWS_SPLIT_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AIWS_SPLIT_2_UNGATED** dient dazu, ein eingehendes AIWS-Adapter-Signal auf zwei identische Ausgänge aufzuteilen. Er ist als generischer Baustein ausgelegt und ermöglicht die Mehrfachverwendung eines AIWS-Signals innerhalb der gleichen Applikation, ohne dass eine separate Signalverteilung programmiert werden muss.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	**IN**	`adapter::types::unidirectional::AIWS`	Eingehendes AIWS-Signal, das auf beide Ausgänge verteilt wird.
Plug (Ausgang)	**OUT1**	`adapter::types::unidirectional::AIWS`	Erster Ausgang – identisches Signal wie am Eingang.
Plug (Ausgang)	**OUT2**	`adapter::types::unidirectional::AIWS`	Zweiter Ausgang – identisches Signal wie am Eingang.

Funktionsweise

Der Baustein leitet das am Socket **IN** anliegende AIWS-Signal unverändert an beide Plugs **OUT1** und **OUT2** weiter. Es findet keine logische oder zeitliche Verarbeitung statt – die Aufteilung erfolgt rein passiv durch die Verdrahtung innerhalb des Funktionsblocks. Dadurch können zwei unabhängige Adapter-Instanzen gleichzeitig mit denselben Daten versorgt werden.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generischer Typ (`GEN_AIWS_SPLIT`) deklariert, sodass er für verschiedene AIWS-Ausprägungen wiederverwendet werden kann.
- **Reine Adapterlogik**: Es werden weder Ereignisse noch Daten-E/A verwendet; die gesamte Kommunikation erfolgt ausschließlich über die Adapter-Schnittstellen.
- **Keine Zustandsabhängigkeit**: Der FB besitzt keine interne Zustandsmaschine – das Ausgangssignal folgt dem Eingangssignal ohne Verzögerung oder Pufferung.

Zustandsübersicht

Der Funktionsblock ist zustandslos. Es existieren keine internen Zustände oder Übergänge. Die Ausgänge sind zu jedem Zeitpunkt eine direkte Abbildung des Eingangs.

Anwendungsszenarien

- **Verteilen eines Sensorsignals**: Ein einzelner AIWS-Sensor (z. B. analoger Wert) soll von zwei unabhängigen Regelungs- oder Überwachungsbausteinen gleichzeitig gelesen werden.
- **Redundante Verarbeitung**: Bei Sicherheitsanwendungen kann dasselbe Signal an mehrere parallele Auswertefunktionen übergeben werden.
- **Logische Verzweigung**: Aufteilung eines Datenstroms für unterschiedliche Auswertungen oder Visualisierungen in einer 4diac-Applikation.

Vergleich mit ähnlichen Bausteinen

- **AIWS_SPLIT_3, AIWS_SPLIT_4**: Erweiterte Varianten mit drei bzw. vier Ausgängen – prinzipiell gleiche Funktionsweise.
- **SPLIT-Bausteine für andere Adaptertypen**: In 4diac existieren analoge Split-Bausteine für andere Adapter (z. B. `DINT_SPLIT`), die ein Datensignal auf mehrere Ausgänge verteilen – jedoch auf Datenebene und nicht über Adapter.
- **Ereignis-Split-Bausteine**: Bausteine wie `E_SPLIT` verteilen Ereignisse, nicht Datenströme – AIWS_SPLIT_2_UNGATED adressiert spezifisch die AIWS-Adapterkommunikation.
- `AIWS_SPLIT_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

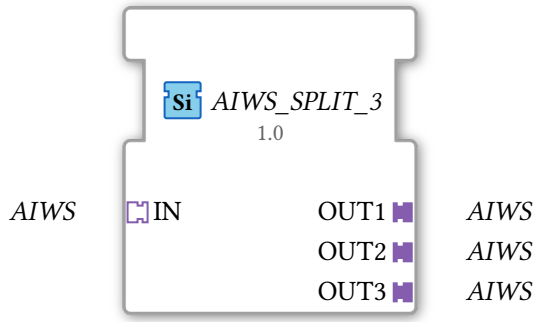
Fazit

Der **AIWS_SPLIT_2_UNGATED** ist ein einfacher, aber nützlicher generischer Funktionsblock zur Vervielfältigung eines AIWS-Adaptersignals. Er reduziert den Verdrahtungsaufwand in Applikationen, die dasselbe analoge oder gemischte Signal mehrfach benötigen, und fügt sich durch seine passive, zustandslose Architektur transparent in bestehende Abläufe ein.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AIWS_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AIWS_SPLIT_3** dient dazu, einen einzelnen eingehenden AIWS-Adapter (Typ `adapter::types::unidirectional::AIWS`) auf drei identische Ausgangsadapter aufzuteilen. Alle Daten, die über den Eingangsadapter empfangen werden, werden unverändert an alle drei Ausgänge weitergeleitet. Der Baustein ist generisch ausgelegt und kann mit verschiedenen AIWS-Typen verwendet werden.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	IN	`adapter::types::unidirectional::AIWS`	Eingangsadapter, der das zu verteilende Signal empfängt.
Plug (Ausgang)	OUT1	`adapter::types::unidirectional::AIWS`	Erster Ausgangsadapter.
Plug (Ausgang)	OUT2	`adapter::types::unidirectional::AIWS`	Zweiter Ausgangsadapter.
Plug (Ausgang)	OUT3	`adapter::types::unidirectional::AIWS`	Dritter Ausgangsadapter.

Funktionsweise

Die interne Logik des Bausteins ist denkbar einfach: Sobald der Socket **IN** Daten vom angeschlossenen Adapter empfängt, werden diese Daten simultan an die drei Plugs **OUT1**, **OUT2** und **OUT3** weiterge-

geben. Es findet keine Transformation, Filterung oder Pufferung statt. Die Datenweitergabe erfolgt unidirektional und ohne zeitliche Verzögerung.

Technische Besonderheiten

- **Generischer Funktionsbaustein:** Der FB ist als generischer Baustein (`eclipse4diac::core::GenericClassName = 'GEN_AIWS_SPLIT'`) implementiert. Dadurch kann er mit verschiedenen Spezialisierungen des AIWS-Adapters verwendet werden, solange diese dem unidirektionalen Kommunikationsprotokoll entsprechen. Die konkrete Typisierung erfolgt zur Entwurfszeit in der 4diac-IDE.
- **Keine Zustandsautomatik:** Der Baustein besitzt keine Ereignisse und keine Zustandsmaschine (ECC). Die Datenweitergabe erfolgt rein datenflussgetrieben.
- **Skalierbarkeit:** Der Baustein ist speziell für eine 1-zu-3-Aufteilung ausgelegt. Für andere Aufteilungsverhältnisse stehen separate Bausteine (z. B. AIWS_SPLIT_2, AIWS_SPLIT_4) zur Verfügung.

Zustandsübersicht

Der Baustein besitzt keine Zustandsmaschine (ECC), daher gibt es keine Zustände oder Zustandsübergänge. Die Funktion ist rein kombinatorisch auf Adapterebene.

Anwendungsszenarien

- **Signalverteilung:** Ein von einer Quelle kommender AIWS-Datenstrom soll parallel an mehrere nachfolgende Verarbeitungsblöcke weitergeleitet werden (z. B. Überwachung, parallele Filterung oder Visualisierung).
- **Redundanz:** Das gleiche Signal wird auf verschiedenen Pfaden genutzt, um Redundanz oder Vergleichbarkeit zu erreichen.
- **Test und Simulation:** In Testumgebungen kann das Eingangssignal ohne Nebenwirkungen mehreren Testmodulen gleichzeitig zur Verfügung gestellt werden.

Vergleich mit ähnlichen Bausteinen

- **AIWS_SPLIT_2 / AIWS_SPLIT_N:** Bausteine mit gleicher Funktionalität, aber unterschiedlicher Anzahl von Ausgängen (2 bzw. n Ausgänge).
- **ALL_SPLIT (für Daten):** Ein generischer Baustein zum Aufteilen von Daten-Ein-/Ausgängen, nicht spezifisch für Adapter. AIWS_SPLIT_3 ist dagegen speziell für den AIWS-Adapter-Typ optimiert und arbeitet auf Adapterebene.
- **AIWS_JOIN_3:** Der inverse Baustein, der drei AIWS-Eingänge zu einem Ausgang zusammenführt.
- `AIWS_SPLIT_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

Der **AIWS_SPLIT_3** ist ein einfacher aber effektiver generischer Funktionsbaustein zur Vervielfältigung eines AIWS-Adapter-Signals auf drei parallele Ausgänge. Dank seiner generischen Natur lässt er sich flexibel in verschiedenen Automatisierungs- und Steuerungsanwendungen einsetzen, in denen ein Datenstrom mehrfach benötigt wird.

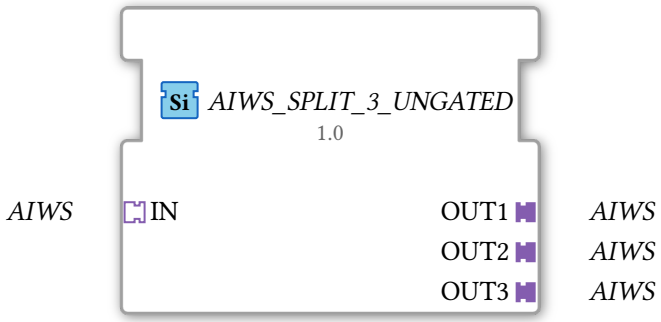
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AIWS_SPLIT_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AIWS_SPLIT_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos

weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AIWS_SPLIT_3_UNGATED** dient dazu, einen einzelnen eingehenden AIWS-Adapter (Typ `adapter::types::unidirectional::AIWS`) auf drei identische Ausgangsadapter aufzuteilen. Alle Daten, die über den Eingangsadapter empfangen werden, werden unverändert an alle drei Ausgänge weitergeleitet. Der Baustein ist generisch ausgelegt und kann mit verschiedenen AIWS-Typen verwendet werden.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	IN	`adapter::types::unidirectional::AIWS`	Eingangsadapter, der das zu verteilende Signal empfängt.
Plug (Ausgang)	OUT1	`adapter::types::unidirectional::AIWS`	Erster Ausgangsadapter.
Plug (Ausgang)	OUT2	`adapter::types::unidirectional::AIWS`	Zweiter Ausgangsadapter.
Plug (Ausgang)	OUT3	`adapter::types::unidirectional::AIWS`	Dritter Ausgangsadapter.

Funktionsweise

Die interne Logik des Bausteins ist denkbar einfach: Sobald der Socket **IN** Daten vom angeschlossenen Adapter empfängt, werden diese Daten simultan an die drei Plugs **OUT1**, **OUT2** und **OUT3** weitergegeben. Es findet keine Transformation, Filterung oder Pufferung statt. Die Datenweitergabe erfolgt unidirektional und ohne zeitliche Verzögerung.

Technische Besonderheiten

- **Generischer Funktionsbaustein:** Der FB ist als generischer Baustein (`eclipse4diac::core::GenericClassName = 'GEN_AIWS_SPLIT'`) implementiert. Dadurch kann er mit verschiedenen Spezialisierungen des AIWS-Adapters verwendet werden, solange diese dem unidirektionalen Kommunikationsprotokoll entsprechen. Die konkrete Typisierung erfolgt zur Entwurfszeit in der 4diac-IDE.
- **Keine Zustandsautomatik:** Der Baustein besitzt keine Ereignisse und keine Zustandsmaschine (ECC). Die Datenweitergabe erfolgt rein datenflussgetrieben.
- **Skalierbarkeit:** Der Baustein ist speziell für eine 1-zu-3-Aufteilung ausgelegt. Für andere Aufteilungsverhältnisse stehen separate Bausteine (z. B. AIWS_SPLIT_2, AIWS_SPLIT_4) zur Verfügung.

Zustandsübersicht

Der Baustein besitzt keine Zustandsmaschine (ECC), daher gibt es keine Zustände oder Zustandsübergänge. Die Funktion ist rein kombinatorisch auf Adapterebene.

Anwendungsszenarien

- **Signalverteilung:** Ein von einer Quelle kommender AIWS-Datenstrom soll parallel an mehrere nachfolgende Verarbeitungsblöcke weitergeleitet werden (z. B. Überwachung, parallele Filterung oder Visualisierung).
- **Redundanz:** Das gleiche Signal wird auf verschiedenen Pfaden genutzt, um Redundanz oder Vergleichbarkeit zu erreichen.
- **Test und Simulation:** In Testumgebungen kann das Eingangssignal ohne Nebenwirkungen mehreren Testmodulen gleichzeitig zur Verfügung gestellt werden.

Vergleich mit ähnlichen Bausteinen

- **AIWS_SPLIT_2 / AIWS_SPLIT_N:** Bausteine mit gleicher Funktionalität, aber unterschiedlicher Anzahl von Ausgängen (2 bzw. n Ausgänge).
- **ALL_SPLIT (für Daten):** Ein generischer Baustein zum Aufteilen von Daten-Ein-/Ausgängen, nicht spezifisch für Adapter. AIWS_SPLIT_3_UNGATED ist dagegen speziell für den AIWS-Adapter-Typ optimiert und arbeitet auf Adapterebene.
- **AIWS_JOIN_3:** Der inverse Baustein, der drei AIWS-Eingänge zu einem Ausgang zusammenführt.
- `AIWS_SPLIT_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

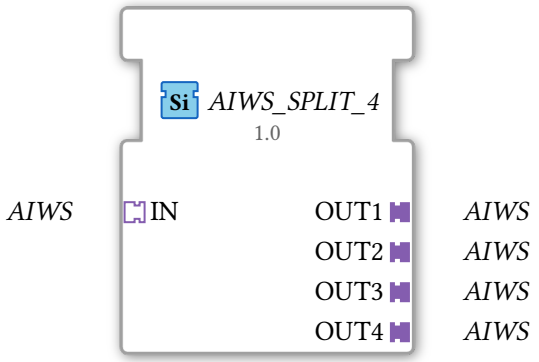
Fazit

Der **AIWS_SPLIT_3_UNGATED** ist ein einfacher aber effektiver generischer Funktionsbaustein zur Vervielfältigung eines AIWS-Adapter-Signals auf drei parallele Ausgänge. Dank seiner generischen Natur lässt er sich flexibel in verschiedenen Automatisierungs- und Steuerungsanwendungen einsetzen, in denen ein Datenstrom mehrfach benötigt wird.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AIWS_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AIWS_SPLIT_4` dient dazu, eine eingehende unidirektionale **AIWS**-Adapterverbindung auf vier parallele Ausgänge aufzuteilen. Er fungiert als passiver Splitter, der die über den Socket empfangenen Werte an alle vier Plugs weiterleitet, ohne selbst Ereignisse oder Daten zu verarbeiten.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Typ	Richtung	Name	Beschreibung
`adapter::types::unidirectional::AIWS`	Socket	**IN**	Eingangsadapter für die zu verteilende AIWS-Verbindung
`adapter::types::unidirectional::AIWS`	Plug	**OUT1**	Erster Ausgangsadapter
`adapter::types::unidirectional::AIWS`	Plug	**OUT2**	Zweiter Ausgangsadapter
`adapter::types::unidirectional::AIWS**	Plug	**OUT3**	Dritter Ausgangsadapter
`adapter::types::unidirectional::AIWS`	Plug	**OUT4**	Vierter Ausgangsadapter

Funktionsweise

Der Baustein ist ein reiner „Verteiler“ (Splitter) für den unidirektionalen Datentyp **AIWS**. Er besitzt keine interne Logik, keine Zustände und keine eigenen Ereignisse. Alle Werte, die über den Eingangsadapter `IN` ankommen, werden identisch auf die vier Ausgangsadapter `OUT1` bis `OUT4` kopiert. Dadurch können mehrere nachfolgende FB parallel mit denselben Daten versorgt werden.

Da der FB generisch ausgelegt ist (`GenericClassName = 'GEN_AIWS_SPLIT'`), muss der konkrete Datentyp **AIWS** beim Einfügen in ein Projekt definiert werden.

Technische Besonderheiten

- **Generische Implementierung**: Der FB wird über das Attribut `eclipse4diac::core::GenericClassName` als generischer Splitter markiert. Der tatsächliche Typ wird erst bei der Instanziierung festgelegt.
- **Keine aktive Steuerung**: Der FB benötigt keine Ereignisse zur Auslösung – die Verteilung erfolgt passiv über die Adapterverbindungen.
- **Vollständige Transparenz**: Änderungen an den Adapterdaten werden ohne Verzögerung an alle Ausgänge weitergegeben.

Zustandsübersicht

Der FB besitzt keine eigenen Zustände oder Abläufe. Es liegt eine rein kombinatorische Weiterleitung vor.

Anwendungsszenarien

- **Verteilung von Sensordaten** – Ein Sensorwert (z.B. Temperatur, Druck) soll mehreren Verarbeitungsblöcken gleichzeitig zur Verfügung gestellt werden.
- **Redundante Datensicherung** – Ein Datenstrom soll parallel an zwei oder mehr voneinander unabhängige Logikeinheiten gehen.
- **Test- und Simulationsaufbauten** – Ein Signal wird auf mehrere Pfade aufgeteilt, um verschiedene Reaktionsmuster zu prüfen.

Vergleich mit ähnlichen Bausteinen

- **Manuelles Verkabeln**: Ohne Splitter müsste jeder nachfolgende FB eine eigene Verbindung zum Quell-FB haben, was die Übersichtlichkeit verringert.
- **Event-basierte Splitter**: Manche Splittter verwenden Ereignisausgänge (z.B. `SPLIT` für beliebige Events). `AIWS_SPLIT_4` ist speziell auf den unidirektionalen **AIWS**-Adapter zugeschnitten und benötigt keine Events.
- **Mehrere Ausgänge**: Es gibt Varianten auf zwei (z.B. `AIWS_SPLIT_2`) oder mehr Ausgänge; `AIWS_SPLIT_4` stellt standardmäßig vier bereit.
- `AIWS_SPLIT_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

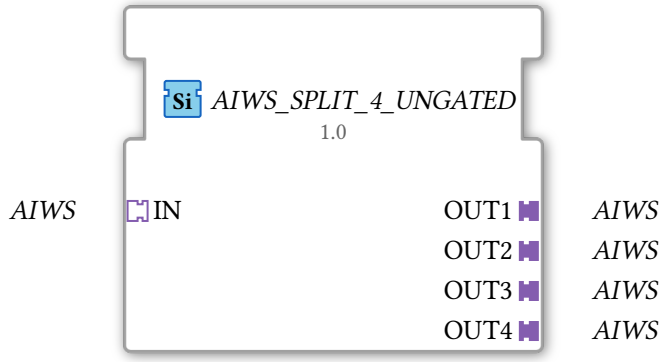
Der `AIWS_SPLIT_4` ist ein einfacher, aber essenzieller Baustein zur Verteilung von unidirektionalen **AIWS**-Adapterverbindungen. Er reduziert den Verdrahtungsaufwand, sorgt für klare Strukturen und ermöglicht die parallele Nutzung eines Datenstroms durch mehrere Funktionsblöcke ohne zusätzliche Logik.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AIWS_SPLIT_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AIWS_SPLIT_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AIWS_SPLIT_4_UNGATED` dient dazu, eine eingehende unidirektionale **AIWS**-Adapterverbindung auf vier parallele Ausgänge aufzuteilen. Er fungiert als passiver Splitter, der die über den Socket empfangenen Werte an alle vier Plugs weiterleitet, ohne selbst Ereignisse oder Daten zu verarbeiten.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Typ	Richtung	Name	Beschreibung
`adapter::types::unidirectional::AIWS`	Socket	**IN**	Eingangsadapter für die zu verteilende AIWS-Verbindung
`adapter::types::unidirectional::AIWS`	Plug	**OUT1**	Erster Ausgangsadapter
`adapter::types::unidirectional::AIWS`	Plug	**OUT2**	Zweiter Ausgangsadapter
`adapter::types::unidirectional::AIWS**	Plug	**OUT3**	Dritter Ausgangsadapter
`adapter::types::unidirectional::AIWS`	Plug	**OUT4**	Vierter Ausgangsadapter

Funktionsweise

Der Baustein ist ein reiner „Verteiler“ (Splitter) für den unidirektionalen Datentyp **AIWS**. Er besitzt keine interne Logik, keine Zustände und keine eigenen Ereignisse. Alle Werte, die über den Eingangsadapter `IN` ankommen, werden identisch auf die vier Ausgangsadapter `OUT1` bis `OUT4` kopiert. Dadurch können mehrere nachfolgende FB parallel mit denselben Daten versorgt werden.

Da der FB generisch ausgelegt ist (`GenericClassName = 'GEN_AIWS_SPLIT'`), muss der konkrete Datentyp **AIWS** beim Einfügen in ein Projekt definiert werden.

Technische Besonderheiten

- **Generische Implementierung**: Der FB wird über das Attribut `eclipse4diac::core::GenericClassName` als generischer Splitter markiert. Der tatsächliche Typ wird erst bei der Instanziierung festgelegt.
- **Keine aktive Steuerung**: Der FB benötigt keine Ereignisse zur Auslösung – die Verteilung erfolgt passiv über die Adapterverbindungen.
- **Vollständige Transparenz**: Änderungen an den Adapterdaten werden ohne Verzögerung an alle Ausgänge weitergegeben.

Zustandsübersicht

Der FB besitzt keine eigenen Zustände oder Abläufe. Es liegt eine rein kombinatorische Weiterleitung vor.

Anwendungsszenarien

- **Verteilung von Sensordaten** – Ein Sensorwert (z.B. Temperatur, Druck) soll mehreren Verarbeitungsblöcken gleichzeitig zur Verfügung gestellt werden.
- **Redundante Datensicherung** – Ein Datenstrom soll parallel an zwei oder mehr voneinander unabhängige Logikeinheiten gehen.
- **Test- und Simulationsaufbauten** – Ein Signal wird auf mehrere Pfade aufgeteilt, um verschiedene Reaktionsmuster zu prüfen.

Vergleich mit ähnlichen Bausteinen

- **Manuelles Verkabeln**: Ohne Splitter müsste jeder nachfolgende FB eine eigene Verbindung zum Quell-FB haben, was die Übersichtlichkeit verringert.
- **Event-basierte Splitter**: Manche Splittter verwenden Ereignisausgänge (z.B. `SPLIT` für beliebige Events). `AIWS_SPLIT_4_UNGATED` ist speziell auf den unidirektionalen **AIWS**-Adapter zugeschnitten und benötigt keine Events.
- **Mehrere Ausgänge**: Es gibt Varianten auf zwei (z.B. `AIWS_SPLIT_2`) oder mehr Ausgänge; `AIWS_SPLIT_4_UNGATED` stellt standardmäßig vier bereit.
- `AIWS_SPLIT_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

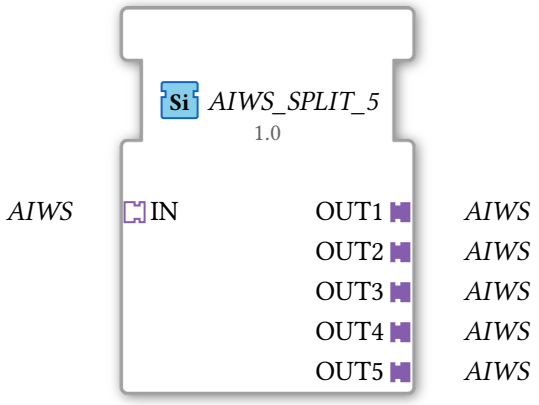
Fazit

Der `AIWS_SPLIT_4_UNGATED` ist ein einfacher, aber essenzieller Baustein zur Verteilung von unidirektionalen **AIWS**-Adapterverbindungen. Er reduziert den Verdrahtungsaufwand, sorgt für klare Strukturen und ermöglicht die parallele Nutzung eines Datenstroms durch mehrere Funktionsblöcke ohne zusätzliche Logik.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AIWS_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AIWS_SPLIT_5** ist ein generischer Baustein zur Aufteilung eines einzelnen **AIWS**-Adapters auf fünf identische **AIWS**-Adapter. Er ermöglicht es, ein von einem Socket bereitgestelltes AIWS-Signal an mehrere unabhängige Adapter-Plugs weiterzuleiten, ohne dass eine event- oder datenverarbeitende Logik erforderlich ist. Dieser FB eignet sich besonders für die Signalverteilung in industriellen Automatisierungsarchitekturen, die auf dem **IEC 61499**-Standard basieren.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden. Der Baustein arbeitet rein daten- bzw. adapter-basiert.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden. Die Weiterleitung erfolgt implizit über die Adapter-Schnittstellen.

Daten-Eingänge

Keine Daten-Eingänge vorhanden. Die gesamte Informationsübertragung erfolgt über den Socket-Adapter.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden. Die Ausgabe geschieht über die Plug-Adapter.

Adapter

Typ	Bezeichnung	Richtung	Adapter-Typ
Socket (Eingang)	IN	Eingang	`adapter::types::unidirectional::AIWS`
Plug (Ausgang)	OUT1	Ausgang	`adapter::types::unidirectional::AIWS`
Plug (Ausgang)	OUT2	Ausgang	`adapter::types::unidirectional::AIWS`
Plug (Ausgang)	OUT3	Ausgang	`adapter::types::unidirectional::AIWS`
Plug (Ausgang)	OUT4	Ausgang	`adapter::types::unidirectional::AIWS`
Plug (Ausgang)	OUT5	Ausgang	`adapter::types::unidirectional::AIWS`

Der FB enthält einen einzelnen **Socket** (`IN`) und fünf **Plugs** (`OUT1` … `OUT5`), die alle vom gleichen, unidirektionalen **AIWS-Adaptertyp** sind.

Funktionsweise

Der **AIWS_SPLIT_5** leitet das an seinem Socket `IN` anliegende Adapter-Signal unverändert an alle fünf Plug-Adapter `OUT1` bis `OUT5` weiter. Es erfolgt keine Transformation oder Filterung der Daten – der Baustein fungiert als reiner **Signalverteiler** (Fan-Out). Sobald eine Verbindung über den Socket hergestellt

ist, stehen die Ausgangsadapter sofort für die Kommunikation zur Verfügung. Ereignissteuerung oder zustandsabhängige Logik sind nicht erforderlich.

Technische Besonderheiten

- **Generische Ausprägung**: Der FB besitzt die Attribute `eclipse4diac::core::GenericClassName` und `eclipse4diac::core::TypeHash`, die eine nachträgliche Konfiguration bzw. Typanpassung erlauben. Standardmäßig ist der Klassenname auf `'GEN_AIWS_SPLIT'` gesetzt; der TypeHash ist leer.
- **Einheitlicher Adaptertyp**: Alle Schnittstellen verwenden denselben unidirektionalen AIWS-Adapter. Dadurch ist die Kompatibilität zwischen Ein- und Ausgängen sichergestellt.
- **Keine Daten- oder Event-Ports**: Die gesamte Kommunikation erfolgt ausschließlich über die Adapter-Schnittstellen, was die Konfiguration vereinfacht und die Anzahl der Verbindungen reduziert.

Zustandsübersicht

Der **AIWS_SPLIT_5** besitzt keine explizite Zustandsmaschine. Seine Funktionalität ist rein kombinatorisch – sobald der Socket verbunden ist, liegen an allen Ausgängen die gleichen Adapter-Signale an. Daher existieren keine internen Zustände, die das Verhalten beeinflussen.

Anwendungsszenarien

- **Signalverteilung in modularen Anlagen**: Ein von einem Sensor oder einer Steuerung bereitgestelltes AIWS-Signal muss parallel an mehrere Verbraucher (z. B. Aktoren, HMI-Systeme, übergeordnete Controller) weitergeleitet werden.
- **Test- und Simulationsumgebungen**: Ein einzelnes AIWS-Signal wird zur Überwachung oder Aufzeichnung an verschiedene Test- und Analysekomponenten verteilt.
- **Redundante Verarbeitung**: In sicherheitskritischen Systemen kann das identische Signal an mehrere unabhängige Logikeinheiten geführt werden, um eine parallele Auswertung zu ermöglichen.

Vergleich mit ähnlichen Bausteinen

Baustein	Anzahl Ausgänge	Typ	Besonderheit
AIWS_SPLIT_5	5	AIWS-Adapter	Generisch, keine Events/Daten
AIWS_SPLIT_2	2	AIWS-Adapter	Gleiche Funktionsweise, minimale Verteilung
AIWS_SPLIT_3	3	AIWS-Adapter	Mittlere Anzahl Ausgänge
AIWS_SPLIT_4	4	AIWS-Adapter	Alternative zu **5**

Alle Split-Varianten basieren auf demselben Prinzip und unterscheiden sich lediglich in der Anzahl der Ausgangsadapter. Der **AIWS_SPLIT_5** deckt einen mittleren Verteilungsbedarf ab und ist besonders dann vorteilhaft, wenn genau fünf parallele Ausgänge benötigt werden.

- `AIWS_SPLIT_5_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

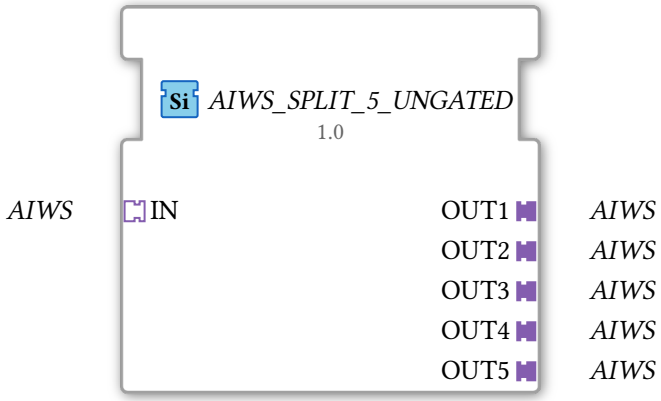
Der **AIWS_SPLIT_5** ist ein einfacher, aber essenzieller Funktionsblock zur Vervielfältigung von AIWS-Adapter-Signalen. Durch seine generische Auslegung und den Verzicht auf Events oder Datenportierung lässt er sich ohne zusätzliche Konfiguration in bestehende IEC 61499-Applikationen integrieren. Ideal für Anwendungen, die eine punktgenaue Verteilung eines Adapters auf bis zu fünf unabhängige Pfade erfordern.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AIWS_SPLIT_5_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `AIWS_SPLIT_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AIWS_SPLIT_5_UNGATED** ist ein generischer Baustein zur Aufteilung eines einzelnen **AIWS**-Adapters auf fünf identische **AIWS**-Adapter. Er ermöglicht es, ein von einem Socket bereitgestelltes AIWS-Signal an mehrere unabhängige Adapter-Plugs weiterzuleiten, ohne dass eine event- oder datenverarbeitende Logik erforderlich ist. Dieser FB eignet sich besonders für die Signalverteilung in industriellen Automatisierungsarchitekturen, die auf dem **IEC 61499**-Standard basieren.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden. Der Baustein arbeitet rein daten- bzw. adapter-basiert.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden. Die Weiterleitung erfolgt implizit über die Adapter-Schnittstellen.

Daten-Eingänge

Keine Daten-Eingänge vorhanden. Die gesamte Informationsübertragung erfolgt über den Socket-Adapter.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden. Die Ausgabe geschieht über die Plug-Adapter.

Adapter

Typ	Bezeichnung	Richtung	Adapter-Typ
Socket (Eingang)	IN	Eingang	`adapter::types::unidirectional::AIWS`
Plug (Ausgang)	OUT1	Ausgang	`adapter::types::unidirectional::AIWS`
Plug (Ausgang)	OUT2	Ausgang	`adapter::types::unidirectional::AIWS`
Plug (Ausgang)	OUT3	Ausgang	`adapter::types::unidirectional::AIWS`
Plug (Ausgang)	OUT4	Ausgang	`adapter::types::unidirectional::AIWS`

Plug (Ausgang)	OUT5	Ausgang	`adapter::types::unidirectional::AIWS`

Der FB enthält einen einzelnen **Socket** (`IN`) und fünf **Plugs** (`OUT1` … `OUT5`), die alle vom gleichen, unidirektionalen **AIWS-Adaptertyp** sind.

Funktionsweise

Der **AIWS_SPLIT_5_UNGATED** leitet das an seinem Socket `IN` anliegende Adapter-Signal unverändert an alle fünf Plug-Adapter `OUT1` bis `OUT5` weiter. Es erfolgt keine Transformation oder Filterung der Daten – der Baustein fungiert als reiner **Signalverteiler** (Fan-Out). Sobald eine Verbindung über den Socket hergestellt ist, stehen die Ausgangsadapter sofort für die Kommunikation zur Verfügung. Ereignissteuerung oder zustandsabhängige Logik sind nicht erforderlich.

Technische Besonderheiten

- **Generische Ausprägung**: Der FB besitzt die Attribute `eclipse4diac::core::GenericClassName` und `eclipse4diac::core::TypeHash`, die eine nachträgliche Konfiguration bzw. Typanpassung erlauben. Standardmäßig ist der Klassenname auf `'GEN_AIWS_SPLIT'` gesetzt; der TypeHash ist leer.
- **Einheitlicher Adaptertyp**: Alle Schnittstellen verwenden denselben unidirektionalen AIWS-Adapter. Dadurch ist die Kompatibilität zwischen Ein- und Ausgängen sichergestellt.
- **Keine Daten- oder Event-Ports**: Die gesamte Kommunikation erfolgt ausschließlich über die Adapter-Schnittstellen, was die Konfiguration vereinfacht und die Anzahl der Verbindungen reduziert.

Zustandsübersicht

Der **AIWS_SPLIT_5_UNGATED** besitzt keine explizite Zustandsmaschine. Seine Funktionalität ist rein kombinatorisch – sobald der Socket verbunden ist, liegen an allen Ausgängen die gleichen Adapter-Signale an. Daher existieren keine internen Zustände, die das Verhalten beeinflussen.

Anwendungsszenarien

- **Signalverteilung in modularen Anlagen**: Ein von einem Sensor oder einer Steuerung bereitgestelltes AIWS-Signal muss parallel an mehrere Verbraucher (z. B. Aktoren, HMI-Systeme, übergeordnete Controller) weitergeleitet werden.
- **Test- und Simulationsumgebungen**: Ein einzelnes AIWS-Signal wird zur Überwachung oder Aufzeichnung an verschiedene Test- und Analysekomponenten verteilt.
- **Redundante Verarbeitung**: In sicherheitskritischen Systemen kann das identische Signal an mehrere unabhängige Logikeinheiten geführt werden, um eine parallele Auswertung zu ermöglichen.

Vergleich mit ähnlichen Bausteinen

Baustein	Anzahl Ausgänge	Typ	Besonderheit
AIWS_SPLIT_5_UNGATED	5	AIWS-Adapter	Generisch, keine Events/Daten
AIWS_SPLIT_2	2	AIWS-Adapter	Gleiche Funktionsweise, minimale Verteilung
AIWS_SPLIT_3	3	AIWS-Adapter	Mittlere Anzahl Ausgänge
AIWS_SPLIT_4	4	AIWS-Adapter	Alternative zu **5**

Alle Split-Varianten basieren auf demselben Prinzip und unterscheiden sich lediglich in der Anzahl der Ausgangsadapter. Der **AIWS_SPLIT_5_UNGATED** deckt einen mittleren Verteilungsbedarf ab und ist besonders dann vorteilhaft, wenn genau fünf parallele Ausgänge benötigt werden.

- `AIWS_SPLIT_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

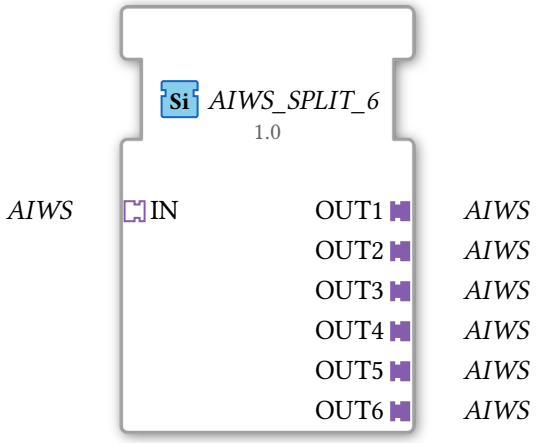
Fazit

Der **AIWS_SPLIT_5_UNGATED** ist ein einfacher, aber essenzieller Funktionsblock zur Vervielfältigung von AIWS-Adapter-Signalen. Durch seine generische Auslegung und den Verzicht auf Events oder Datenportierung lässt er sich ohne zusätzliche Konfiguration in bestehende IEC 61499-Applikationen integrieren. Ideal für Anwendungen, die eine punktgenaue Verteilung eines Adapters auf bis zu fünf unabhängige Pfade erfordern.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AIWS_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AIWS_SPLIT_6 dient dazu, ein eingehendes AIWS-Signal (Adaptertyp) auf sechs identische Ausgänge zu verteilen. Er ist als generischer Baustein konzipiert und ermöglicht eine einfache Signalvervielfachung in Steuerungsanwendungen, die auf dem 4diac-IDE-Framework basieren.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Socket (Eingang):

- `IN` (Typ: `adapter::types::unidirectional::AIWS`): Eingangsadapter für das zu verteilende AIWS-Signal.

Plugs (Ausgänge):

- `OUT1` (Typ: `adapter::types::unidirectional::AIWS`): Erster Ausgang mit dem gesplitteten Signal.
- `OUT2` (Typ: `adapter::types::unidirectional::AIWS`): Zweiter Ausgang.
- `OUT3` (Typ: `adapter::types::unidirectional::AIWS`): Dritter Ausgang.
- `OUT4` (Typ: `adapter::types::unidirectional::AIWS`): Vierter Ausgang.

- `OUT5` (Typ: `adapter::types::unidirectional::AIWS`): Fünfter Ausgang.
- `OUT6` (Typ: `adapter::types::unidirectional::AIWS`): Sechster Ausgang.

Funktionsweise

Der AIWS_SPLIT_6-Baustein nimmt das an seinem Socket `IN` anliegende AIWS-Signal entgegen und leitet es unverändert an alle sechs Plug-Ausgänge `OUT1` bis `OUT6` weiter. Es findet keine Signalverarbeitung oder -änderung statt; der Baustein fungiert als passiver Splitter (1:6). Die Kommunikation erfolgt über Adapter gemäß der unidirektionalen AIWS-Schnittstellendefinition.

Technische Besonderheiten

- Der Baustein ist als generischer Funktionsblock implementiert, erkennbar an den Attributen `GenericClassName` und `TypeHash`. Dadurch kann er in verschiedenen Ausprägungen (z. B. AIWS_SPLIT_2, AIWS_SPLIT_4) als Instanz eines generischen Typs erstellt werden.
- Es sind keine Ereignis- oder Daten-Ein-/Ausgänge vorhanden; die gesamte Datenübertragung erfolgt ausschließlich über die Adapter-Schnittstellen.
- Der Baustein enthält keine eigene Zustandslogik oder Verarbeitungsalgorithmen.

Zustandsübersicht

Der Baustein besitzt keine definierten Zustände, da er keine ereignisgesteuerte Ausführung erfordert. Die Signalweitergabe erfolgt kontinuierlich und passiv.

Anwendungsszenarien

- Verteilung eines analogen Eingangssignals (z. B. von einem Sensor) an mehrere nachgeschaltete Funktionsblöcke.
- Bereitstellung eines Referenzsignals für mehrere parallele Regelkreise.
- Einsatz in modularen Automatisierungslösungen, die eine Signalvervielfachung ohne zusätzliche Hardware erfordern.

Vergleich mit ähnlichen Bausteinen

- **AIWS_SPLIT_2, AIWS_SPLIT_4:** Diese Bausteine verteilen das Signal auf zwei bzw. vier Ausgänge. AIWS_SPLIT_6 erweitert die Anzahl auf sechs.
- **AIWS_MERGE:** Ein zusammensetzender Baustein, der mehrere AIWS-Eingänge zu einem Ausgang zusammenführt – funktionale Umkehrung des Splitters.
- **Direkte Verbindung:** Anstatt eines Splitter-Bausteins könnten mehrere parallele Verbindungen im Applikationsmodell gezogen werden, was jedoch die Übersichtlichkeit reduziert und bei Änderungen aufwändiger ist.
- `AIWS_SPLIT_6_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

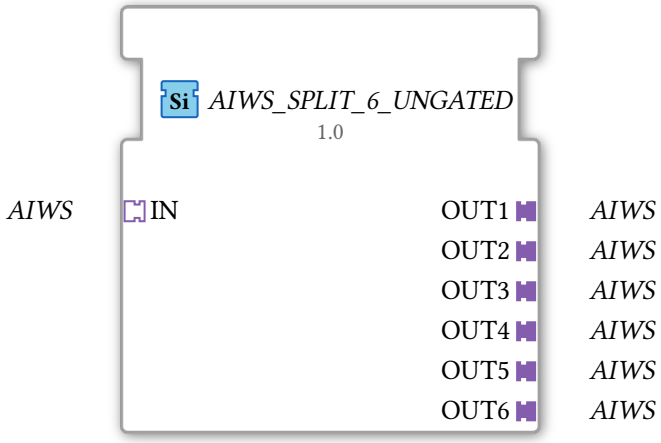
Der AIWS_SPLIT_6 ist ein einfacher, aber nützlicher Baustein zur Vervielfachung von AIWS-Signalen in IEC 61499-basierten Steuerungen. Seine generische Natur und die klare Adapter-Schnittstelle erleichtern die Wiederverwendung und Integration in größere Projekte.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AIWS_SPLIT_6_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `AIWS_SPLIT_6`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AIWS_SPLIT_6_UNGATED dient dazu, ein eingehendes AIWS-Signal (Adaptertyp) auf sechs identische Ausgänge zu verteilen. Er ist als generischer Baustein konzipiert und ermöglicht eine einfache Signalvervielfachung in Steuerungsanwendungen, die auf dem 4diac-IDE-Framework basieren.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Socket (Eingang):

- `IN` (Typ: `adapter::types::unidirectional::AIWS`): Eingangsadapter für das zu verteilende AIWS-Signal.

Plugs (Ausgänge):

- `OUT1` (Typ: `adapter::types::unidirectional::AIWS`): Erster Ausgang mit dem gesplitteten Signal.
- `OUT2` (Typ: `adapter::types::unidirectional::AIWS`): Zweiter Ausgang.
- `OUT3` (Typ: `adapter::types::unidirectional::AIWS`): Dritter Ausgang.
- `OUT4` (Typ: `adapter::types::unidirectional::AIWS`): Vierter Ausgang.
- `OUT5` (Typ: `adapter::types::unidirectional::AIWS`): Fünfter Ausgang.
- `OUT6` (Typ: `adapter::types::unidirectional::AIWS`): Sechster Ausgang.

Funktionsweise

Der AIWS_SPLIT_6_UNGATED-Baustein nimmt das an seinem Socket `IN` anliegende AIWS-Signal entgegen und leitet es unverändert an alle sechs Plug-Ausgänge `OUT1` bis `OUT6` weiter. Es findet keine Signalverarbeitung oder -änderung statt; der Baustein fungiert als passiver Splitter (1:6). Die Kommunikation erfolgt über Adapter gemäß der unidirektionalen AIWS-Schnittstellendefinition.

Technische Besonderheiten

- Der Baustein ist als generischer Funktionsblock implementiert, erkennbar an den Attributen `GenericClassName` und `TypeHash`. Dadurch kann er in verschiedenen Ausprägungen (z. B. AIWS_SPLIT_2, AIWS_SPLIT_4) als Instanz eines generischen Typs erstellt werden.
- Es sind keine Ereignis- oder Daten-Ein-/Ausgänge vorhanden; die gesamte Datenübertragung erfolgt ausschließlich über die Adapter-Schnittstellen.
- Der Baustein enthält keine eigene Zustandslogik oder Verarbeitungsalgorithmen.

Zustandsübersicht

Der Baustein besitzt keine definierten Zustände, da er keine ereignisgesteuerte Ausführung erfordert. Die Signalweitergabe erfolgt kontinuierlich und passiv.

Anwendungsszenarien

- Verteilung eines analogen Eingangssignals (z. B. von einem Sensor) an mehrere nachgeschaltete Funktionsblöcke.
- Bereitstellung eines Referenzsignals für mehrere parallele Regelkreise.
- Einsatz in modularen Automatisierungslösungen, die eine Signalvervielfachung ohne zusätzliche Hardware erfordern.

Vergleich mit ähnlichen Bausteinen

- **AIWS_SPLIT_2, AIWS_SPLIT_4:** Diese Bausteine verteilen das Signal auf zwei bzw. vier Ausgänge. AIWS_SPLIT_6_UNGATED erweitert die Anzahl auf sechs.
- **AIWS_MERGE:** Ein zusammensetzender Baustein, der mehrere AIWS-Eingänge zu einem Ausgang zusammenführt – funktionale Umkehrung des Splitters.
- **Direkte Verbindung:** Anstatt eines Splitter-Bausteins könnten mehrere parallele Verbindungen im Applikationsmodell gezogen werden, was jedoch die Übersichtlichkeit reduziert und bei Änderungen aufwändiger ist.
- `AIWS_SPLIT_6`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

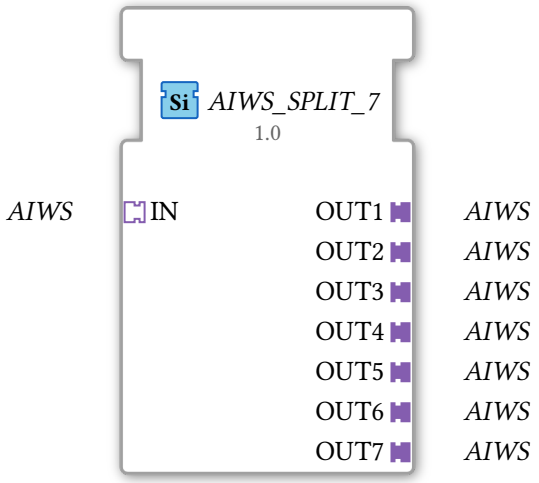
Fazit

Der AIWS_SPLIT_6_UNGATED ist ein einfacher, aber nützlicher Baustein zur Vervielfachung von AIWS-Signalen in IEC 61499-basierten Steuerungen. Seine generische Natur und die klare Adapter-Schnittstelle erleichtern die Wiederverwendung und Integration in größere Projekte.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AIWS_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AIWS_SPLIT_7** ist ein generischer Baustein, der einen einzelnen eingehenden AIWS-Adapter (unidirektional) auf sieben identische ausgehende AIWS-Adapter aufteilt. Er dient als Signalverteiler für automatisierungstechnische Anwendungen, bei denen ein Datenstrom oder eine Verbindung auf mehrere parallele Pfade kopiert werden muss.

Schnittstellenstruktur

Ereignis-Eingänge

Keine (der Funktionsblock besitzt keine Ereignis-Eingänge).

Ereignis-Ausgänge

Keine (der Funktionsblock besitzt keine Ereignis-Ausgänge).

Daten-Eingänge

Keine (der Funktionsblock besitzt keine direkten Daten-Eingänge; die Datenübertragung erfolgt ausschließlich über die Adapter-Schnittstellen).

Daten-Ausgänge

Keine (der Funktionsblock besitzt keine direkten Daten-Ausgänge; die Datenübertragung erfolgt ausschließlich über die Adapter-Schnittstellen).

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AIWS`	**IN**	Socket (Eingang)	Empfängt den zu verteilenden AIWS-Datenstrom.
`adapter::types::unidirectional::AIWS`	**OUT1**	Plug (Ausgang)	Erster identischer Ausgangskanal.
`adapter::types::unidirectional::AIWS`	**OUT2**	Plug (Ausgang)	Zweiter identischer Ausgangskanal.
`adapter::types::unidirectional::AIWS`	**OUT3**	Plug (Ausgang)	Dritter identischer Ausgangskanal.
`adapter::types::unidirectional::AIWS`	**OUT4**	Plug (Ausgang)	Vierter identischer Ausgangskanal.

adapter::types::unidirectional::AIWS	**OUT5**	Plug (Ausgang)	Fünfter identischer Ausgangskanal.
adapter::types::unidirectional::AIWS	**OUT6**	Plug (Ausgang)	Sechster identischer Ausgangskanal.
adapter::types::unidirectional::AIWS	**OUT7**	Plug (Ausgang)	Siebter identischer Ausgangskanal.

Funktionsweise

Der Funktionsblock empfängt über den Socket **IN** einen unidirektionalen Datenstrom vom Typ `AIWS`. Dieser Datenstrom wird intern ohne Änderung dupliziert und gleichzeitig an alle sieben Plugs **OUT1** bis **OUT7** weitergegeben. Jeder Ausgangskanal erhält somit exakt dieselben Daten wie der Eingang. Es findet weder eine Verzögerung noch eine Datenmanipulation statt.

Technische Besonderheiten

- **Generischer Baustein** – Der FB ist als generische Komponente definiert (Attribut `GenericClassName = 'GEN_AIWS_SPLIT'`), was eine typsichere Wiederverwendung in verschiedenen Kontexten ermöglicht.
- **Unidirektionale Adapter** – Alle verwendeten Adapter sind unidirektional, d.h. die Daten fließen nur vom Socket zu den Plugs.
- **Keine Ereignissteuerung** – Der FB besitzt keinen internen Zustandsautomaten und keine Ereignis-Ein-/Ausgänge; die Verteilung erfolgt rein datengetrieben.
- **Skalierbare Architektur** – Durch die modulare Adapter-Schnittstelle kann der Baustein in komplexen Hierarchien von Funktionsblöcken eingesetzt werden.

Zustandsübersicht

Da der Funktionsblock über keinen Zustandsautomaten (ECC – Execution Control Chart) verfügt, existieren keine expliziten Zustände. Die Ausgangssignale sind stets direkt von den Eingangssignalen abhängig. Der FB arbeitet als reine Datenverteilung ohne Verzögerung oder Schalthandlungen.

Anwendungsszenarien

- **Parallelverarbeitung** – Verteilung eines Messwertes oder Steuersignals auf mehrere unabhängige Regelungs- oder Überwachungspfade.
- **Signalüberwachung** – Gleiches Signal wird an verschiedene Diagnose- oder Logikblöcke geführt.
- **Redundanz** – Bereitstellung identischer Datenströme für unterschiedliche Empfänger (z. B. Bedienung, Visualisierung und Archivierung).
- **Testumgebungen** – Ein Testsignal kann gleichzeitig an mehrere Simulationsbausteine gesendet werden.

Vergleich mit ähnlichen Bausteinen

- **AIWS_SPLIT_2, AIWS_SPLIT_4** – Bausteine mit gleicher Funktionalität, jedoch unterschiedlicher Anzahl an Ausgängen (2 bzw. 4). AIWS_SPLIT_7 bietet die maximale Anzahl an Ausgängen im Rahmen dieser Splitter-Familie.
- **Generische 1:N-Splitter** – In anderen IEC 61499-Bibliotheken existieren oft generische Splitter für Datentypen wie `ANY`. AIWS_SPLIT_7 ist speziell für den Adaptertyp `AIWS` ausgelegt und benötigt daher keine Typkonvertierung.
- `AIWS_SPLIT_7_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

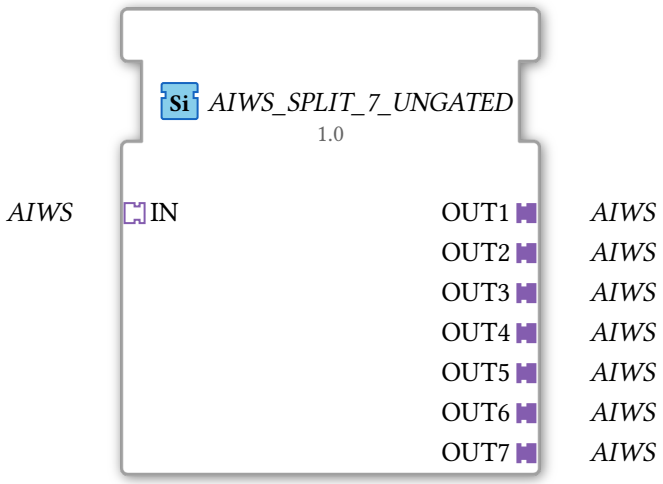
Fazit

Der **AIWS_SPLIT_7** ist ein einfacher, aber leistungsfähiger Funktionsblock zur Vervielfachung eines unidirektionalen AIWS-Datenstroms auf sieben parallele Ausgänge. Seine generische Definition und

die rein adapterbasierte Schnittstelle ermöglichen eine flexible Integration in IEC 61499-Anwendungen, insbesondere für Aufgaben der Signalverteilung und Parallelverarbeitung.

AIWS_SPLIT_7_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AIWS_SPLIT_7`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AIWS_SPLIT_7_UNGATED** ist ein generischer Baustein, der einen einzelnen eingehenden AIWS-Adapter (unidirektional) auf sieben identische ausgehende AIWS-Adapter aufteilt. Er dient als Signalverteiler für automatisierungstechnische Anwendungen, bei denen ein Datenstrom oder eine Verbindung auf mehrere parallele Pfade kopiert werden muss.

Schnittstellenstruktur

Ereignis-Eingänge

Keine (der Funktionsblock besitzt keine Ereignis-Eingänge).

Ereignis-Ausgänge

Keine (der Funktionsblock besitzt keine Ereignis-Ausgänge).

Daten-Eingänge

Keine (der Funktionsblock besitzt keine direkten Daten-Eingänge; die Datenübertragung erfolgt ausschließlich über die Adapter-Schnittstellen).

Daten-Ausgänge

Keine (der Funktionsblock besitzt keine direkten Daten-Ausgänge; die Datenübertragung erfolgt ausschließlich über die Adapter-Schnittstellen).

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AIWS`	**IN**	Socket (Eingang)	Empfängt den zu verteilenden AIWS-Datenstrom.

adapter::types::unidirectional::AIWS	**OUT1**	Plug (Ausgang)	Erster identischer Ausgangskanal.
adapter::types::unidirectional::AIWS	**OUT2**	Plug (Ausgang)	Zweiter identischer Ausgangskanal.
adapter::types::unidirectional::AIWS	**OUT3**	Plug (Ausgang)	Dritter identischer Ausgangskanal.
adapter::types::unidirectional::AIWS	**OUT4**	Plug (Ausgang)	Vierter identischer Ausgangskanal.
adapter::types::unidirectional::AIWS	**OUT5**	Plug (Ausgang)	Fünfter identischer Ausgangskanal.
adapter::types::unidirectional::AIWS	**OUT6**	Plug (Ausgang)	Sechster identischer Ausgangskanal.
adapter::types::unidirectional::AIWS	**OUT7**	Plug (Ausgang)	Siebter identischer Ausgangskanal.

Funktionsweise

Der Funktionsblock empfängt über den Socket **IN** einen unidirektionalen Datenstrom vom Typ `AIWS`. Dieser Datenstrom wird intern ohne Änderung dupliziert und gleichzeitig an alle sieben Plugs **OUT1** bis **OUT7** weitergegeben. Jeder Ausgangskanal erhält somit exakt dieselben Daten wie der Eingang. Es findet weder eine Verzögerung noch eine Datenmanipulation statt.

Technische Besonderheiten

- **Generischer Baustein** – Der FB ist als generische Komponente definiert (Attribut `GenericClassName = 'GEN_AIWS_SPLIT'`), was eine typsichere Wiederverwendung in verschiedenen Kontexten ermöglicht.
- **Unidirektionale Adapter** – Alle verwendeten Adapter sind unidirektional, d.h. die Daten fließen nur vom Socket zu den Plugs.
- **Keine Ereignissteuerung** – Der FB besitzt keinen internen Zustandsautomaten und keine Ereignis-Ein-/Ausgänge; die Verteilung erfolgt rein datengetrieben.
- **Skalierbare Architektur** – Durch die modulare Adapter-Schnittstelle kann der Baustein in komplexen Hierarchien von Funktionsblöcken eingesetzt werden.

Zustandsübersicht

Da der Funktionsblock über keinen Zustandsautomaten (ECC – Execution Control Chart) verfügt, existieren keine expliziten Zustände. Die Ausgangssignale sind stets direkt von den Eingangssignalen abhängig. Der FB arbeitet als reine Datenverteilung ohne Verzögerung oder Schalthandlungen.

Anwendungsszenarien

- **Parallelverarbeitung** – Verteilung eines Messwertes oder Steuersignals auf mehrere unabhängige Regelungs- oder Überwachungspfade.
- **Signalüberwachung** – Gleiches Signal wird an verschiedene Diagnose- oder Logikblöcke geführt.
- **Redundanz** – Bereitstellung identischer Datenströme für unterschiedliche Empfänger (z. B. Bedienung, Visualisierung und Archivierung).
- **Testumgebungen** – Ein Testsignal kann gleichzeitig an mehrere Simulationsbausteine gesendet werden.

Vergleich mit ähnlichen Bausteinen

- **AIWS_SPLIT_2, AIWS_SPLIT_4** – Bausteine mit gleicher Funktionalität, jedoch unterschiedlicher Anzahl an Ausgängen (2 bzw. 4). AIWS_SPLIT_7_UNGATED bietet die maximale Anzahl an Ausgängen im Rahmen dieser Splitter-Familie.
- **Generische 1:N-Splitter** – In anderen IEC 61499-Bibliotheken existieren oft generische Splitter für Datentypen wie `ANY`. AIWS_SPLIT_7_UNGATED ist speziell für den Adaptertyp `AIWS` ausgelegt und benötigt daher keine Typkonvertierung.
- `AIWS_SPLIT_7`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

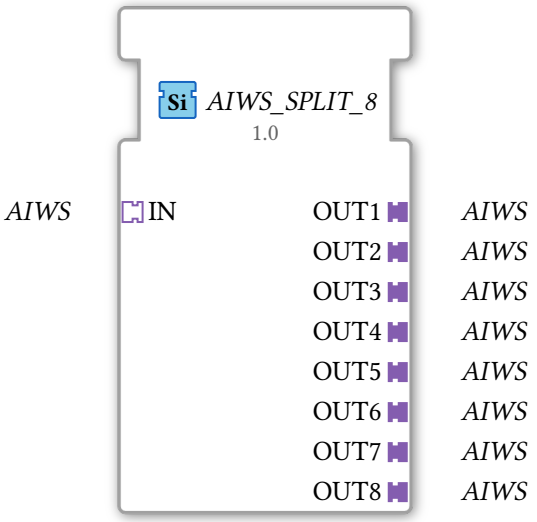
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AIWS_SPLIT_7_UNGATED** ist ein einfacher, aber leistungsfähiger Funktionsblock zur Vervielfachung eines unidirektionalen AIWS-Datenstroms auf sieben parallele Ausgänge. Seine generische Definition und die rein adapterbasierte Schnittstelle ermöglichen eine flexible Integration in IEC 61499-Anwendungen, insbesondere für Aufgaben der Signalverteilung und Parallelverarbeitung.

AIWS_SPLIT_8



• − ◦ • − ◦ • − ◦ •

Einleitung

Der Baustein **AIWS_SPLIT_8** teilt ein eingehendes AIWS-Adapter-Signal auf acht identische Ausgänge auf. Er fungiert als generischer Splitter für unidirektionale Adapterverbindungen und ermöglicht die parallele Weiterleitung eines Signals an mehrere nachfolgende Funktionsbausteine.

Schnittstellenstruktur

Der Baustein besitzt weder Ereignis- noch Dateneingänge oder -ausgänge. Die gesamte Kommunikation erfolgt ausschließlich über Adapter.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Eingang (Socket)	IN	adapater::types::unidirectional::AIWS	Eingehendes AIWS-Signal, das aufgeteilt werden soll.

Ausgang (Plug)	OUT1	adapater::types::unidirectional::AIWS	Erster paralleler Ausgang des Signals.
Ausgang (Plug)	OUT2	adapater::types::unidirectional::AIWS	Zweiter paralleler Ausgang des Signals.
Ausgang (Plug)	OUT3	adapater::types::unidirectional::AIWS	Dritter paralleler Ausgang des Signals.
Ausgang (Plug)	OUT4	adapater::types::unidirectional::AIWS	Vierter paralleler Ausgang des Signals.
Ausgang (Plug)	OUT5	adapater::types::unidirectional::AIWS	Fünfter paralleler Ausgang des Signals.
Ausgang (Plug)	OUT6	adapater::types::unidirectional::AIWS	Sechster paralleler Ausgang des Signals.
Ausgang (Plug)	OUT7	adapater::types::unidirectional::AIWS	Siebter paralleler Ausgang des Signals.
Ausgang (Plug)	OUT8	adapater::types::unidirectional::AIWS	Achter paralleler Ausgang des Signals.

Funktionsweise

Der Baustein empfängt über den **IN**-Socket ein Signal des Typs AIWS (Adapter). Dieses Signal wird ohne Verarbeitung oder Verzögerung auf alle acht **OUT**-Plugs dupliziert. Änderungen am Eingangssignal werden sofort an allen Ausgängen widergespiegelt.

Technische Besonderheiten

- Der Baustein ist generisch und für eine feste Anzahl von acht Ausgängen ausgelegt.
- Es existieren keine ereignisgesteuerten Abläufe – die Signalweitergabe erfolgt rein adapterbasiert und kontinuierlich.
- Die Implementierung ist als reiner Splitter ohne interne Logik oder Zustandsverwaltung realisiert.

Zustandsübersicht

Der Baustein besitzt keine Zustandsmaschine. Es gibt keine internen Zustände oder Ereignissequenzen – die Ausgabe folgt direkt der Eingabe.

Anwendungsszenarien

- **Signalverteilung** in Automatisierungsanwendungen: Ein AIWS-Sensorwert soll mehreren unabhängigen Regelungs- oder Überwachungsblöcken gleichzeitig zur Verfügung gestellt werden.
- **Parallelisierung** von Algorithmen: Nachfolgende Bausteine benötigen den gleichen Adapterdatenstrom, arbeiten aber parallel auf verschiedenen Ressourcen.
- **Test- und Simulationsumgebungen**: Ein eingehendes Signal wird auf mehrere Auswertungsinstanzen verteilt, um verschiedene Verarbeitungspfade zu vergleichen.

Vergleich mit ähnlichen Bausteinen

- **AIWS_SPLIT_4** – Splitter mit vier Ausgängen.
- **AIWS_SPLIT_16** – Splitter mit 16 Ausgängen.
- **AIWS_MERGE** – Zusammenführung mehrerer AIWS-Adapter zu einem Ausgang (Gegenstück).

Gemeinsam ist allen Split-Bausteinen die unveränderte Weitergabe des Signals; sie unterscheiden sich nur in der Anzahl der Ausgänge.

- `AIWS_SPLIT_8_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs

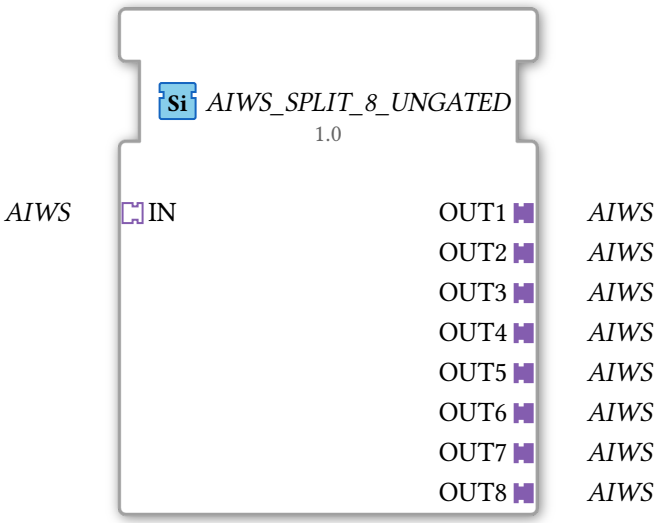
unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

AIWS_SPLIT_8 ist ein einfacher, aber notwendiger Baustein zur Vervielfachung eines AIWS-Adapter-Signals auf acht parallele Pfade. Seine Implementation ohne Ereignissteuerung oder Zustandslogik macht ihn leichtgewichtig und zuverlässig für den Einsatz in verteilten Automatisierungssystemen.

AIWS_SPLIT_8_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AIWS_SPLIT_8`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AIWS_SPLIT_8_UNGATED** teilt ein eingehendes AIWS-Adapter-Signal auf acht identische Ausgänge auf. Er fungiert als generischer Splitter für unidirektionale Adapterverbindungen und ermöglicht die parallele Weiterleitung eines Signals an mehrere nachfolgende Funktionsbausteine.

Schnittstellenstruktur

Der Baustein besitzt weder Ereignis- noch Dateneingänge oder -ausgänge. Die gesamte Kommunikation erfolgt ausschließlich über Adapter.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Eingang (Socket)	IN	adapater::types::unidirectional::AIWS	Eingehendes AIWS-Signal, das aufgeteilt werden soll.
Ausgang (Plug)	OUT1	adapater::types::unidirectional::AIWS	Erster paralleler Ausgang des Signals.
Ausgang (Plug)	OUT2	adapater::types::unidirectional::AIWS	Zweiter paralleler Ausgang des Signals.
Ausgang (Plug)	OUT3	adapater::types::unidirectional::AIWS	Dritter paralleler Ausgang des Signals.
Ausgang (Plug)	OUT4	adapater::types::unidirectional::AIWS	Vierter paralleler Ausgang des Signals.
Ausgang (Plug)	OUT5	adapater::types::unidirectional::AIWS	Fünfter paralleler Ausgang des Signals.
Ausgang (Plug)	OUT6	adapater::types::unidirectional::AIWS	Sechster paralleler Ausgang des Signals.
Ausgang (Plug)	OUT7	adapater::types::unidirectional::AIWS	Siebter paralleler Ausgang des Signals.
Ausgang (Plug)	OUT8	adapater::types::unidirectional::AIWS	Achter paralleler Ausgang des Signals.

Funktionsweise

Der Baustein empfängt über den **IN**-Socket ein Signal des Typs AIWS (Adapter). Dieses Signal wird ohne Verarbeitung oder Verzögerung auf alle acht **OUT**-Plugs dupliziert. Änderungen am Eingangssignal werden sofort an allen Ausgängen widergespiegelt.

Technische Besonderheiten

- Der Baustein ist generisch und für eine feste Anzahl von acht Ausgängen ausgelegt.
- Es existieren keine ereignisgesteuerten Abläufe – die Signalweitergabe erfolgt rein adapterbasiert und kontinuierlich.
- Die Implementierung ist als reiner Splitter ohne interne Logik oder Zustandsverwaltung realisiert.

Zustandsübersicht

Der Baustein besitzt keine Zustandsmaschine. Es gibt keine internen Zustände oder Ereignissequenzen – die Ausgabe folgt direkt der Eingabe.

Anwendungsszenarien

- **Signalverteilung** in Automatisierungsanwendungen: Ein AIWS-Sensorwert soll mehreren unabhängigen Regelungs- oder Überwachungsblöcken gleichzeitig zur Verfügung gestellt werden.
- **Parallelisierung** von Algorithmen: Nachfolgende Bausteine benötigen den gleichen Adapterdatenstrom, arbeiten aber parallel auf verschiedenen Ressourcen.
- **Test- und Simulationsumgebungen**: Ein eingehendes Signal wird auf mehrere Auswertungsinstanzen verteilt, um verschiedene Verarbeitungspfade zu vergleichen.

Vergleich mit ähnlichen Bausteinen

- **AIWS_SPLIT_4** – Splitter mit vier Ausgängen.
- **AIWS_SPLIT_16** – Splitter mit 16 Ausgängen.
- **AIWS_MERGE** – Zusammenführung mehrerer AIWS-Adapter zu einem Ausgang (Gegenstück).

Gemeinsam ist allen Split-Bausteinen die unveränderte Weitergabe des Signals; sie unterscheiden sich nur in der Anzahl der Ausgänge.

- `AIWS_SPLIT_8`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

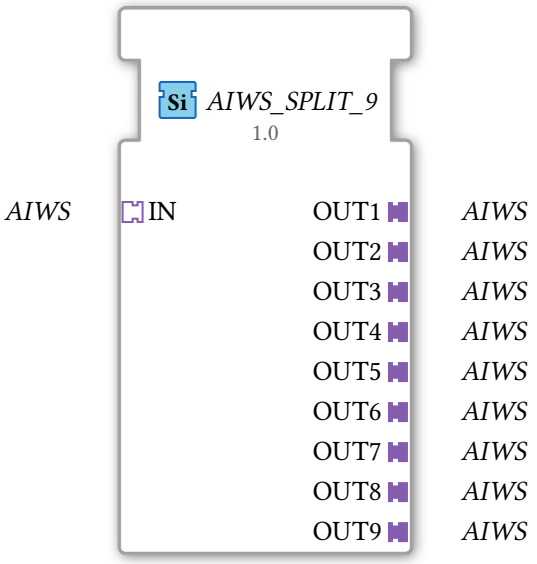
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

AIWS_SPLIT_8_UNGATED ist ein einfacher, aber notwendiger Baustein zur Vervielfachung eines AIWS-Adapter-Signals auf acht parallele Pfade. Seine Implementation ohne Ereignissteuerung oder Zustandslogik macht ihn leichtgewichtig und zuverlässig für den Einsatz in verteilten Automatisierungssystemen.

AIWS_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AIWS_SPLIT_9` dient der Aufteilung eines eingehenden `AIWS`-Adapter-Signals in neun separate, identisch typisierte Ausgänge. Er wird als generischer Baustein (generic FB) bereitgestellt und ermöglicht die Verteilung eines einzelnen Adapter-Datenstroms an mehrere nachfolgende Funktionseinheiten. Der Block ist unidirektional ausgelegt und benötigt keine Ereignis- oder Dateneingänge, da die Datenübertragung ausschließlich über die Adapter-Schnittstellen erfolgt.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung

Socket	IN	adapter::types::unidirectional::AIWS	Eingehendes AIWS-Signal, das auf die neun Ausgänge verteilt wird.
Plug	OUT1	adapter::types::unidirectional::AIWS	Erster ausgehender AIWS-Ausgang.
Plug	OUT2	adapter::types::unidirectional::AIWS	Zweiter ausgehender AIWS-Ausgang.
Plug	OUT3	adapter::types::unidirectional::AIWS	Dritter ausgehender AIWS-Ausgang.
Plug	OUT4	adapter::types::unidirectional::AIWS	Vierter ausgehender AIWS-Ausgang.
Plug	OUT5	adapter::types::unidirectional::AIWS	Fünfter ausgehender AIWS-Ausgang.
Plug	OUT6	adapter::types::unidirectional::AIWS	Sechster ausgehender AIWS-Ausgang.
Plug	OUT7	adapter::types::unidirectional::AIWS	Siebter ausgehender AIWS-Ausgang.
Plug	OUT8	adapter::types::unidirectional::AIWS	Achter ausgehender AIWS-Ausgang.
Plug	OUT9	adapter::types::unidirectional::AIWS	Neunter ausgehender AIWS-Ausgang.

Funktionsweise

Der Baustein nimmt am Socket `IN` ein AIWS-Adapter-Signal entgegen und leitet dieses unverändert an alle neun Plugs `OUT1` bis `OUT9` weiter. Es findet keine Datenverarbeitung, Umwandlung oder Verzögerung statt – es handelt sich um eine reine Signalvervielfachung (Fan-Out). Die interne Logik ist als generischer Funktionsblock realisiert, der zur Laufzeit durch die Angabe eines generischen Klassennamens (`eclipse4diac::core::GenericClassName`) parametrisiert werden kann.

Technische Besonderheiten

- **Generischer Funktionsblock**: Der Baustein ist als generischer `GEN_AIWS_SPLIT` gekennzeichnet. Dadurch kann er in verschiedenen Kontexten wiederverwendet werden, indem der konkrete Adapter-Typ zur Entwurfszeit festgelegt wird.
- **Unidirektionalität**: Sowohl der Eingang als auch alle Ausgänge sind unidirektional (`adapter::types::unidirectional::AIWS`), d.h. Daten fließen nur vom Socket zu den Plugs.
- **Keine Ereignissteuerung**: Der Split-Block arbeitet ohne Ereignisse – die Datenweitergabe erfolgt implizit über die Adapter-Verbindungen.
- **Keine Zustandsmaschine**: Es existiert kein ECC (Execution Control Chart), daher ist der Block zustandslos und benötigt keine Initialisierung.
- **Typ-Hash**: Ein optionales Attribut `eclipse4diac::core::TypeHash` kann für Identifikations- und Versionierungszwecke gesetzt werden, ist aber im vorliegenden Fall leer.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Er ist rein kombinatorisch und führt keine sequenziellen Abläufe aus.

Anwendungsszenarien

- **Signalverteilung in Automatisierungssystemen**: Wenn ein Sensor- oder Steuerungssignal (z.B. ein AIWS-konformer Temperatur- oder Druckwert) an mehrere unabhängige Module weitergegeben werden muss.
- **Redundante Verarbeitung**: Ein Signal wird an identische, parallel arbeitende Algorithmen oder Sicherheitslogiken verteilt.
- **Test- und Simulationsumgebungen**: Ein einzelnes AIWS-Signal soll auf mehrere Prüf- oder Überwachungsblöcke aufgeteilt werden.
- **Architekturvereinfachung**: Erspart die manuelle Verkabelung mehrerer Split-Blöcke und reduziert die Komplexität des Applikationsdiagramms.

Vergleich mit ähnlichen Bausteinen

- **AIWS_SPLIT_n (z.B. AIWS_SPLIT_4)**: Andere Varianten mit einer unterschiedlichen Anzahl von Ausgängen (z.B. 2, 4, 8). Der vorliegende Block spezialisiert sich auf eine Aufteilung in genau neun Stränge.

- **Manueller Split mit mehreren FB-Instanzen**: Ohne diesen Baustein müsste man das AIWS-Signal durch Hintereinanderschaltung mehrerer 2- oder 3-fach-Split-Blöcke realisieren, was die Übersichtlichkeit verringert.
- **Datenverteiler auf Basis von Ereignissen**: Bausteine, die über Ereignisse gesteuert werden, benötigen zusätzliche Event-Verdrahtung und sind für reine Datenweiterleitung weniger effizient.
- `AIWS_SPLIT_9_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Jeder Ausgangs-Plug wird unabhängig aktualisiert: Der eingehende Wert wird nur dann auf einen Ausgang geschrieben und dessen Adapter-Event gesendet, wenn er sich vom aktuellen Wert dieses Ausgangs unterscheidet. Bereits synchrone Ausgänge bleiben still, während ein gerade erst verbundener (oder nicht mehr synchroner) Ausgang weiterhin die nötige Aktualisierung erhält.

Fazit

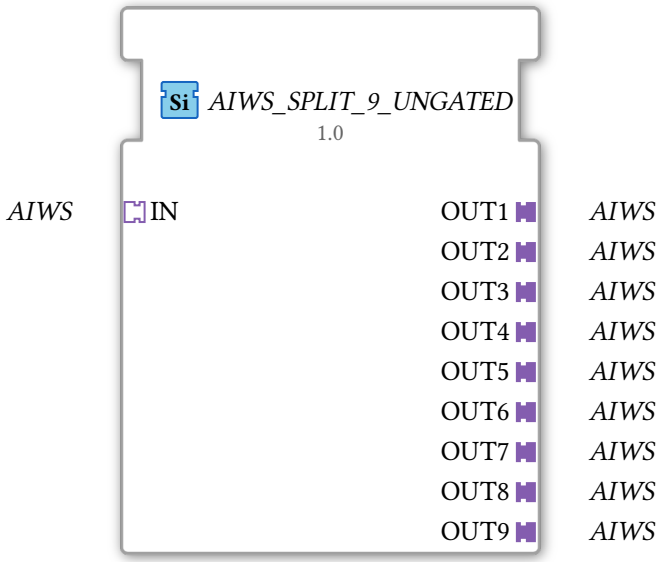
`AIWS_SPLIT_9` ist ein einfacher, aber nützlicher generischer Funktionsblock zur Vervielfachung eines unidirektionalen AIWS-Adapter-Signals auf neun Ausgänge. Er verzichtet auf überflüssige Komplexität, benötigt keine Ereignissteuerung und kann ohne zusätzliche Konfiguration direkt in IEC 61499-Applikationen eingesetzt werden. Dank seiner generischen Natur ist er flexibel an verschiedene Adapter-Typen anpassbar und eignet sich besonders für Signalverteilungen in modularen Automatisierungsarchitekturen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AIWS_SPLIT_9_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AIWS_SPLIT_9`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AIWS_SPLIT_9_UNGATED` dient der Aufteilung eines eingehenden `AIWS`-Adapter-Signals in neun separate, identisch typisierte Ausgänge. Er wird als generischer Baustein (generic FB) bereitgestellt und ermöglicht die Verteilung eines einzelnen Adapter-Datenstroms an mehrere nachfolgende Funktionseinheiten. Der Block ist unidirektional ausgelegt und benötigt keine Ereignis- oder Dateneingänge, da die Datenübertragung ausschließlich über die Adapter-Schnittstellen erfolgt.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	`IN`	`adapter::types::unidirectional::AIWS`	Eingehendes AIWS-Signal, das auf die neun Ausgänge verteilt wird.
Plug	`OUT1`	`adapter::types::unidirectional::AIWS`	Erster ausgehender AIWS-Ausgang.
Plug	`OUT2`	`adapter::types::unidirectional::AIWS`	Zweiter ausgehender AIWS-Ausgang.
Plug	`OUT3`	`adapter::types::unidirectional::AIWS`	Dritter ausgehender AIWS-Ausgang.
Plug	`OUT4`	`adapter::types::unidirectional::AIWS`	Vierter ausgehender AIWS-Ausgang.
Plug	`OUT5`	`adapter::types::unidirectional::AIWS`	Fünfter ausgehender AIWS-Ausgang.
Plug	`OUT6`	`adapter::types::unidirectional::AIWS`	Sechster ausgehender AIWS-Ausgang.
Plug	`OUT7`	`adapter::types::unidirectional::AIWS`	Siebter ausgehender AIWS-Ausgang.
Plug	`OUT8`	`adapter::types::unidirectional::AIWS`	Achter ausgehender AIWS-Ausgang.
Plug	`OUT9`	`adapter::types::unidirectional::AIWS`	Neunter ausgehender AIWS-Ausgang.

Funktionsweise

Der Baustein nimmt am Socket `IN` ein AIWS-Adapter-Signal entgegen und leitet dieses unverändert an alle neun Plugs `OUT1` bis `OUT9` weiter. Es findet keine Datenverarbeitung, Umwandlung oder Verzögerung statt – es handelt sich um eine reine Signalvervielfachung (Fan-Out). Die interne Logik ist als generischer Funktionsblock realisiert, der zur Laufzeit durch die Angabe eines generischen Klassennamens (`eclipse4diac::core::GenericClassName`) parametrisiert werden kann.

Technische Besonderheiten

- **Generischer Funktionsblock**: Der Baustein ist als generischer `GEN_AIWS_SPLIT` gekennzeichnet. Dadurch kann er in verschiedenen Kontexten wiederverwendet werden, indem der konkrete Adapter-Typ zur Entwurfszeit festgelegt wird.
- **Unidirektionalität**: Sowohl der Eingang als auch alle Ausgänge sind unidirektional (`adapter::types::unidirectional::AIWS`), d.h. Daten fließen nur vom Socket zu den Plugs.
- **Keine Ereignissteuerung**: Der Split-Block arbeitet ohne Ereignisse – die Datenweitergabe erfolgt implizit über die Adapter-Verbindungen.
- **Keine Zustandsmaschine**: Es existiert kein ECC (Execution Control Chart), daher ist der Block zustandslos und benötigt keine Initialisierung.

- **Typ-Hash**: Ein optionales Attribut `eclipse4diac::core::TypeHash` kann für Identifikations- und Versionierungszwecke gesetzt werden, ist aber im vorliegenden Fall leer.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Er ist rein kombinatorisch und führt keine sequenziellen Abläufe aus.

Anwendungsszenarien

- **Signalverteilung in Automatisierungssystemen**: Wenn ein Sensor- oder Steuerungssignal (z.B. ein AIWS-konformer Temperatur- oder Druckwert) an mehrere unabhängige Module weitergegeben werden muss.
- **Redundante Verarbeitung**: Ein Signal wird an identische, parallel arbeitende Algorithmen oder Sicherheitslogiken verteilt.
- **Test- und Simulationsumgebungen**: Ein einzelnes AIWS-Signal soll auf mehrere Prüf- oder Überwachungsblöcke aufgeteilt werden.
- **Architekturvereinfachung**: Erspart die manuelle Verkabelung mehrerer Split-Blöcke und reduziert die Komplexität des Applikationsdiagramms.

Vergleich mit ähnlichen Bausteinen

- **AIWS_SPLIT_n (z.B. AIWS_SPLIT_4)**: Andere Varianten mit einer unterschiedlichen Anzahl von Ausgängen (z.B. 2, 4, 8). Der vorliegende Block spezialisiert sich auf eine Aufteilung in genau neun Stränge.
- **Manueller Split mit mehreren FB-Instanzen**: Ohne diesen Baustein müsste man das AIWS-Signal durch Hintereinanderschaltung mehrerer 2- oder 3-fach-Split-Blöcke realisieren, was die Übersichtlichkeit verringert.
- **Datenverteiler auf Basis von Ereignissen**: Bausteine, die über Ereignisse gesteuert werden, benötigen zusätzliche Event-Verdrahtung und sind für reine Datenweiterleitung weniger effizient.
- `AIWS_SPLIT_9`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

`AIWS_SPLIT_9_UNGATED` ist ein einfacher, aber nützlicher generischer Funktionsblock zur Vervielfachung eines unidirektionalen AIWS-Adapter-Signals auf neun Ausgänge. Er verzichtet auf überflüssige Komplexität, benötigt keine Ereignissteuerung und kann ohne zusätzliche Konfiguration direkt in IEC 61499-Applikationen eingesetzt werden. Dank seiner generischen Natur ist er flexibel an verschiedene Adapter-Typen anpassbar und eignet sich besonders für Signalverteilungen in modularen Automatisierungsarchitekturen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

timers

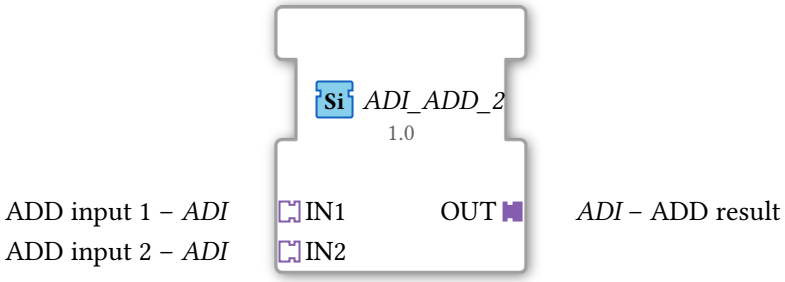
2.1.11.10 iec61131

timers

2.1.11.11 Iec61131 3

arithmetic

ADI_ADD_2



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ADI_ADD_2` ist ein generischer Funktionsbaustein zur Durchführung von arithmetischen Additionsoperationen. Im Gegensatz zu herkömmlichen mathematischen Funktionsbausteinen nutzt dieser Baustein ein adapterbasiertes Schnittstellenkonzept (unidirektionale `ADI`-Adapter), um Daten und zugehörige Steuerungssignale gebündelt zu übertragen. Er ermöglicht die Addition von zwei Eingangswerten zu einem Ausgangswert.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten, dedizierten Ereignis-Eingänge. Die Ereignissteuerung wird vollständig über die Adapter abgewickelt.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten, dedizierten Ereignis-Ausgänge. Die Ereignisweiterleitung wird vollständig über den Ausgangs-Adapter abgewickelt.

Daten-Eingänge

Dieser Funktionsbaustein besitzt keine standardmäßigen, direkten Daten-Eingänge.

Daten-Ausgänge

Dieser Funktionsbaustein besitzt keine standardmäßigen, direkten Daten-Ausgänge.

Adapter

Sockets (Buchsen - Eingänge)

- **IN1**: Typ `adapter::types::unidirectional::ADI`
  - *Beschreibung:* Erster Eingangswert für die mathematische Addition.
- **IN2**: Typ `adapter::types::unidirectional::ADI`
  - *Beschreibung:* Zweiter Eingangswert für die mathematische Addition.

Plugs (Stecker - Ausgänge)

- **OUT**: Typ `adapter::types::unidirectional::ADI`
  - *Beschreibung:* Ausgang, der das Ergebnis der Addition (`IN1 + IN2`) bereitstellt.

Funktionsweise

Der Baustein ADI_ADD_2 realisiert die arithmetische Operation:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} \tag{2.4}$$

Sobald sich an den Eingangs-Adaptern IN1 oder IN2 Daten ändern oder ein entsprechendes Übertragungsereignis über die Adapterstruktur signalisiert wird, verarbeitet der Baustein die Werte. Die Summe wird berechnet und über das entsprechende Ereignis-/Datenbündel des Ausgangs-Adapters OUT an nachfolgende Bausteine weitergeleitet.

Da es sich um einen generischen Baustein (GEN_ADI_ADD) handelt, richtet sich der konkret verwendete Datentyp nach den Spezifikationen und der Instanziierung der genutzten ADI-Adapter.

Technische Besonderheiten

- **Generischer Typ:** Der Baustein ist als GEN_ADI_ADD deklariert. Dies erlaubt eine flexible Handhabung unterschiedlicher numerischer Datentypen, sofern diese durch die verwendeten Adapter unterstützt werden.
- **Adapterbasiertes Design:** Durch die Verwendung von unidirektionalen Adaptern vom Typ ADI wird die Anzahl der expliziten Verbindungslinien (Verdrahtungsaufwand im Application Editor der 4diac-IDE) drastisch reduziert, da Daten und Synchronisationsereignisse in einer Verbindung gekapselt sind.

Zustandsübersicht

Der Baustein verhält sich wie ein zustandsloser (bzw. rein ereignisgesteuerter kombinatorischer) Baustein:

1. **Warten auf Datenaktualisierung:** Der Baustein verharrt im Ruhezustand, bis über IN1 oder IN2 neue Werte signalisiert werden.
2. **Berechnung:** Bei Eintreffen eines Ereignisses an den Eingängen wird die mathematische Summe gebildet.
3. **Ausgabe:** Das Ergebnis wird direkt an den Ausgangs-Adapter OUT übergeben und das zugehörige Trigger-Ereignis ausgelöst.

Anwendungsszenarien

- **Modulare Signalverarbeitung:** Addition von Messwerten (z. B. Sensor 1 + Sensor 2 zur Ermittlung eines Gesamtwerts) in Systemen, die konsequent auf einer Adapter-Architektur aufbauen.
- **Kaskadierte Berechnungen:** Einfache Integration in komplexe arithmetische Berechnungsnetzwerke durch saubere, strukturierte Adapter-Verbindungen.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem klassischen ADD-Baustein nach IEC 61499 (welcher typischerweise dedizierte REQ- und CNF-Ereignisports sowie direkte Dateneingänge wie IN1 und IN2 als ANY_NUM besitzt) kapselt der ADI_ADD_2 diese Schnittstellen in Adaptern.

- **Standard ADD:** Erfordert manuelle Verdrahtung von min. 2 Ereignissen und 3 Datenleitungen (insgesamt 5 Verbindungen).
- **ADI_ADD_2:** Benötigt lediglich die Verbindung der 3 Adapterleitungen (IN1, IN2, OUT), was die Lesbarkeit von komplexen Steuerungsdiagrammen signifikant erhöht.

- **ADI_ADD_2_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

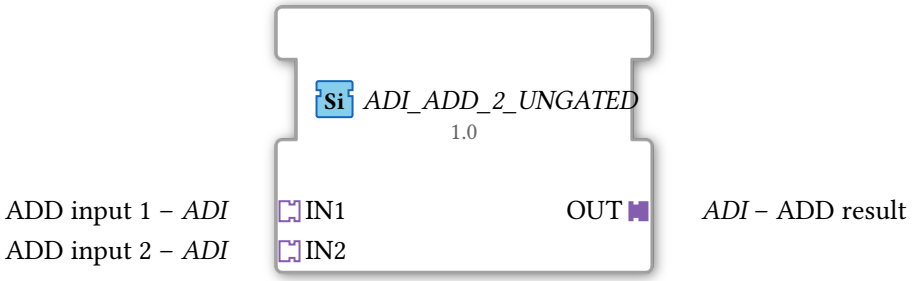
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `ADI_ADD_2` ist ein hocheffizienter, übersichtlicher und moderner Baustein für arithmetische Additionen in der 4diac-IDE. Er eignet sich hervorragend für anspruchsvolle Architekturen, bei denen Übersichtlichkeit durch den Einsatz von standardisierten Adaptern im Vordergrund steht.

ADI_ADD_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ADI_ADD_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ADI_ADD_2_UNGATED` ist ein generischer Funktionsbaustein zur Durchführung von arithmetischen Additionsoperationen. Im Gegensatz zu herkömmlichen mathematischen Funktionsbausteinen nutzt dieser Baustein ein adapterbasiertes Schnittstellenkonzept (unidirektionale `ADI`-Adapter), um Daten und zugehörige Steuerungssignale gebündelt zu übertragen. Er ermöglicht die Addition von zwei Eingangswerten zu einem Ausgangswert.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten, dedizierten Ereignis-Eingänge. Die Ereignissteuerung wird vollständig über die Adapter abgewickelt.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten, dedizierten Ereignis-Ausgänge. Die Ereignisweiterleitung wird vollständig über den Ausgangs-Adapter abgewickelt.

Daten-Eingänge

Dieser Funktionsbaustein besitzt keine standardmäßigen, direkten Daten-Eingänge.

Daten-Ausgänge

Dieser Funktionsbaustein besitzt keine standardmäßigen, direkten Daten-Ausgänge.

Adapter

Sockets (Buchsen - Eingänge)

- **IN1**: Typ `adapter::types::unidirectional::ADI`
  - *Beschreibung:* Erster Eingangswert für die mathematische Addition.
- **IN2**: Typ `adapter::types::unidirectional::ADI`
  - *Beschreibung:* Zweiter Eingangswert für die mathematische Addition.

Plugs (Stecker - Ausgänge)

- **OUT**: Typ `adapter::types::unidirectional::ADI`
  - *Beschreibung:* Ausgang, der das Ergebnis der Addition (`IN1 + IN2`) bereitstellt.

Funktionsweise

Der Baustein `ADI_ADD_2_UNGATED` realisiert die arithmetische Operation:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} \tag{2.5}$$

Sobald sich an den Eingangs-Adaptern `IN1` oder `IN2` Daten ändern oder ein entsprechendes Übertragungsereignis über die Adapterstruktur signalisiert wird, verarbeitet der Baustein die Werte. Die Summe wird berechnet und über das entsprechende Ereignis-/Datenbündel des Ausgangs-Adapters `OUT` an nachfolgende Bausteine weitergeleitet.

Da es sich um einen generischen Baustein (`GEN_ADI_ADD`) handelt, richtet sich der konkret verwendete Datentyp nach den Spezifikationen und der Instanziierung der genutzten `ADI`-Adapter.

Technische Besonderheiten

- **Generischer Typ:** Der Baustein ist als `GEN_ADI_ADD` deklariert. Dies erlaubt eine flexible Handhabung unterschiedlicher numerischer Datentypen, sofern diese durch die verwendeten Adapter unterstützt werden.
- **Adapterbasiertes Design:** Durch die Verwendung von unidirektionalen Adaptern vom Typ `ADI` wird die Anzahl der expliziten Verbindungslinien (Verdrahtungsaufwand im Application Editor der 4diac-IDE) drastisch reduziert, da Daten und Synchronisationsereignisse in einer Verbindung gekapselt sind.

Zustandsübersicht

Der Baustein verhält sich wie ein zustandsloser (bzw. rein ereignisgesteuerter kombinatorischer) Baustein:

1. **Warten auf Datenaktualisierung:** Der Baustein verharrt im Ruhezustand, bis über `IN1` oder `IN2` neue Werte signalisiert werden.
2. **Berechnung:** Bei Eintreffen eines Ereignisses an den Eingängen wird die mathematische Summe gebildet.
3. **Ausgabe:** Das Ergebnis wird direkt an den Ausgangs-Adapter `OUT` übergeben und das zugehörige Trigger-Ereignis ausgelöst.

Anwendungsszenarien

- **Modulare Signalverarbeitung:** Addition von Messwerten (z. B. Sensor 1 + Sensor 2 zur Ermittlung eines Gesamtwerts) in Systemen, die konsequent auf einer Adapter-Architektur aufbauen.
- **Kaskadierte Berechnungen:** Einfache Integration in komplexe arithmetische Berechnungsnetzwerke durch saubere, strukturierte Adapter-Verbindungen.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem klassischen `ADD`-Baustein nach IEC 61499 (welcher typischerweise dedizierte `REQ`- und `CNF`-Ereignisports sowie direkte Dateneingänge wie `IN1` und `IN2` als `ANY_NUM` besitzt) kapselt der `ADI_ADD_2_UNGATED` diese Schnittstellen in Adaptern.

- **Standard ADD:** Erfordert manuelle Verdrahtung von min. 2 Ereignissen und 3 Datenleitungen (insgesamt 5 Verbindungen).
- **ADI_ADD_2_UNGATED:** Benötigt lediglich die Verbindung der 3 Adapterleitungen (`IN1`, `IN2`, `OUT`), was die Lesbarkeit von komplexen Steuerungsdiagrammen signifikant erhöht.

- **`ADI_ADD_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

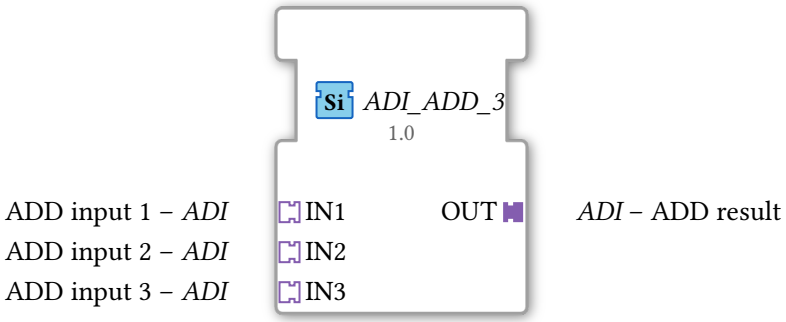
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `ADI_ADD_2_UNGATED` ist ein hocheffizienter, übersichtlicher und moderner Baustein für arithmetische Additionen in der 4diac-IDE. Er eignet sich hervorragend für anspruchsvolle Architekturen, bei denen Übersichtlichkeit durch den Einsatz von standardisierten Adaptern im Vordergrund steht.

ADI_ADD_3



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ADI_ADD_3` ist ein generischer arithmetischer Baustein, der zur Addition von drei Eingangswerten dient. Er basiert auf dem Standard IEC 61499-2 und nutzt ein adapterbasiertes Kommunikationsmodell. Durch die Verwendung von unidirektionalen Adaptern vom Typ `ADI` (Adapter für Datenintegration/Interaktion) ermöglicht der Baustein eine strukturierte und saubere Signal- und Datenführung innerhalb von 4diac-ide.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine direkten, eigenständigen Ereignis-Eingänge vorhanden. Die Ereignissteuerung wird über die zugeordneten Adapter-Eingänge (Sockets) realisiert.

Ereignis-Ausgänge

Es sind keine direkten, eigenständigen Ereignis-Ausgänge vorhanden. Die Ereignisweiterleitung wird über den Adapter-Ausgang (Plug) realisiert.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübertragung erfolgt gekoppelt über die Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Die Datenübertragung erfolgt gekoppelt über den Ausgangs-Adapter.

Adapter

Sockets (Eingangs-Schnittstellen)

- **IN1** (Typ: `adapter::types::unidirectional::ADI`): Erster Eingangssummand für die Addition.
- **IN2** (Typ: `adapter::types::unidirectional::ADI`): Zweiter Eingangssummand für die Addition.
- **IN3** (Typ: `adapter::types::unidirectional::ADI`): Dritter Eingangssummand für die Addition.

Plugs (Ausgangs-Schnittstellen)

- **OUT** (Typ: `adapter::types::unidirectional::ADI`): Ausgang für das berechnete Additionsergebnis ($OUT = IN1 + IN2 + IN3$).

Funktionsweise

Sobald an einem der Eingangs-Sockets (`IN1`, `IN2` oder `IN3`) ein neues Ereignis oder neue Daten eintreffen, liest der Funktionsbaustein die Werte aus den drei Adaptern aus. Er führt eine arithmetische Addition der drei Werte durch:

$$\text{Ergebnis} = \text{IN1} + \text{IN2} + \text{IN3} \tag{2.6}$$

Das berechnete Ergebnis sowie das zugehörige Aktualisierungsereignis werden anschließend über den Ausgangs-Plug `OUT` an die nachfolgenden Bausteine im System übermittelt.

Technische Besonderheiten

- **Generischer Baustein (Generic FB):** Durch das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `GEN_ADI_ADD` verhält sich dieser Baustein polymorph. Er ist nicht auf einen festen Datentyp (wie z. B. `INT` oder `REAL`) fixiert, sondern passt sich flexibel an den im Adapter definierten Datentyp an.
- **Adapterbasierte Architektur:** Die Kapselung von Daten und Events in den `ADI`-Adaptern reduziert das „Kabelgewirr“ in der 4diac-ide erheblich, da keine separaten Event- und Datenleitungen für jeden Summanden gezogen werden müssen.

Zustandsübersicht

Der Funktionsbaustein verhält sich rein zustandslos (reaktiv):

1. **Wartezustand (Idle):** Der Baustein wartet auf Trigger-Signale der Eingangs-Sockets.
2. **Berechnung & Weiterleitung:** Bei Triggerung an `IN1`, `IN2` oder `IN3` wird die Summe instantan gebildet und über den Plug `OUT` ausgegeben.

Anwendungsszenarien

- **Sensor-Konsolidierung:** Aufsummierung von drei verschiedenen physikalischen Größen (z. B. drei Teilströme in einer Anlage zur Ermittlung des Gesamtstroms).
- **Sollwert-Offset-Berechnung:** Kombination aus einem Hauptsollwert, einem Korrekturwert und einem zusätzlichen Toleranz- oder Offsetwert.
- **Prozesswert-Kalkulation:** Bilanzierung von Zuflüssen an drei verschiedenen Messstellen in der Verfahrenstechnik.

Vergleich mit ähnlichen Bausteinen

- **Standard `ADD` (IEC 61131-3):** Der klassische ADD-Baustein verwendet direkte Pins für Daten und Events. `ADI_ADD_3` ist durch die Verwendung von unidirektionalen Adaptern besser für modular aufgebaute und hierarchische Architekturen in IEC 61499 geeignet.
- **Zwei-Wege-Addierer (z.B. `ADI_ADD_2`):** Während für die Addition von drei Werten mit Standardbausteinen zwei Bausteine kaskadiert werden müssten, erledigt `ADI_ADD_3` diese Aufgabe in einem einzigen Schritt, was Ressourcen schont und die Übersichtlichkeit erhöht.

- `ADI_ADD_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

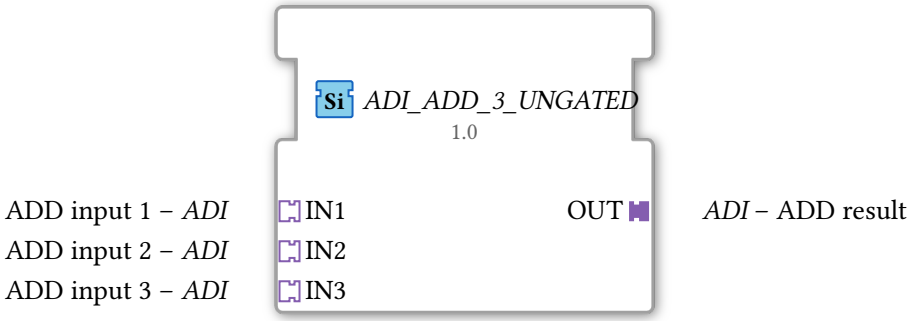
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `ADI_ADD_3` ist ein praktischer, generischer Hilfsbaustein für arithmetische Operationen in modernen IEC 61499 Steuerungsanwendungen. Durch die konsequente Nutzung von Adaptern fügt er sich nahtlos in saubere Software-Architekturen ein und minimiert den Verdrahtungsaufwand in der Entwicklungsumgebung.

ADI_ADD_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ADI_ADD_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ADI_ADD_3_UNGATED` ist ein generischer arithmetischer Baustein, der zur Addition von drei Eingangswerten dient. Er basiert auf dem Standard IEC 61499-2 und nutzt ein adapterbasiertes Kommunikationsmodell. Durch die Verwendung von unidirektionalen Adaptern vom Typ `ADI` (Adapter für Datenintegration/Interaktion) ermöglicht der Baustein eine strukturierte und saubere Signal- und Datenführung innerhalb von 4diac-ide.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine direkten, eigenständigen Ereignis-Eingänge vorhanden. Die Ereignissteuerung wird über die zugeordneten Adapter-Eingänge (Sockets) realisiert.

Ereignis-Ausgänge

Es sind keine direkten, eigenständigen Ereignis-Ausgänge vorhanden. Die Ereignisweiterleitung wird über den Adapter-Ausgang (Plug) realisiert.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübertragung erfolgt gekoppelt über die Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Die Datenübertragung erfolgt gekoppelt über den Ausgangs-Adapter.

Adapter

Sockets (Eingangs-Schnittstellen)

- **IN1** (Typ: `adapter::types::unidirectional::ADI`): Erster Eingangssummand für die Addition.
- **IN2** (Typ: `adapter::types::unidirectional::ADI`): Zweiter Eingangssummand für die Addition.
- **IN3** (Typ: `adapter::types::unidirectional::ADI`): Dritter Eingangssummand für die Addition.

Plugs (Ausgangs-Schnittstellen)

- **OUT** (Typ: `adapter::types::unidirectional::ADI`): Ausgang für das berechnete Additionsergebnis ($OUT = IN1 + IN2 + IN3$).

Funktionsweise

Sobald an einem der Eingangs-Sockets (`IN1`, `IN2` oder `IN3`) ein neues Ereignis oder neue Daten eintreffen, liest der Funktionsbaustein die Werte aus den drei Adaptern aus. Er führt eine arithmetische Addition der drei Werte durch:

$$\text{Ergebnis} = \text{IN1} + \text{IN2} + \text{IN3} \tag{2.7}$$

Das berechnete Ergebnis sowie das zugehörige Aktualisierungsereignis werden anschließend über den Ausgangs-Plug `OUT` an die nachfolgenden Bausteine im System übermittelt.

Technische Besonderheiten

- **Generischer Baustein (Generic FB):** Durch das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `GEN_ADI_ADD` verhält sich dieser Baustein polymorph. Er ist nicht auf einen festen Datentyp (wie z. B. `INT` oder `REAL`) fixiert, sondern passt sich flexibel an den im Adapter definierten Datentyp an.
- **Adapterbasierte Architektur:** Die Kapselung von Daten und Events in den `ADI`-Adaptern reduziert das „Kabelgewirr“ in der 4diac-ide erheblich, da keine separaten Event- und Datenleitungen für jeden Summanden gezogen werden müssen.

Zustandsübersicht

Der Funktionsbaustein verhält sich rein zustandslos (reaktiv):

1. **Wartezustand (Idle):** Der Baustein wartet auf Trigger-Signale der Eingangs-Sockets.
2. **Berechnung & Weiterleitung:** Bei Triggerung an `IN1`, `IN2` oder `IN3` wird die Summe instantan gebildet und über den Plug `OUT` ausgegeben.

Anwendungsszenarien

- **Sensor-Konsolidierung:** Aufsummierung von drei verschiedenen physikalischen Größen (z. B. drei Teilströme in einer Anlage zur Ermittlung des Gesamtstroms).
- **Sollwert-Offset-Berechnung:** Kombination aus einem Hauptsollwert, einem Korrekturwert und einem zusätzlichen Toleranz- oder Offsetwert.
- **Prozesswert-Kalkulation:** Bilanzierung von Zuflüssen an drei verschiedenen Messstellen in der Verfahrenstechnik.

Vergleich mit ähnlichen Bausteinen

- **Standard `ADD` (IEC 61131-3):** Der klassische ADD-Baustein verwendet direkte Pins für Daten und Events. `ADI_ADD_3_UNGATED` ist durch die Verwendung von unidirektionalen Adaptern besser für modular aufgebaute und hierarchische Architekturen in IEC 61499 geeignet.
- **Zwei-Wege-Addierer (z.B. `ADI_ADD_2`):** Während für die Addition von drei Werten mit Standardbausteinen zwei Bausteine kaskadiert werden müssten, erledigt `ADI_ADD_3_UNGATED` diese Aufgabe in einem einzigen Schritt, was Ressourcen schont und die Übersichtlichkeit erhöht.

- **`ADI_ADD_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

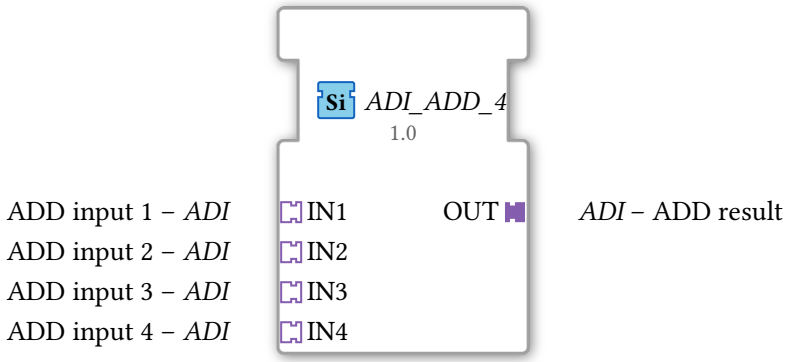
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `ADI_ADD_3_UNGATED` ist ein praktischer, generischer Hilfsbaustein für arithmetische Operationen in modernen IEC 61499 Steuerungsanwendungen. Durch die konsequente Nutzung von Adaptern fügt er sich nahtlos in saubere Software-Architekturen ein und minimiert den Verdrahtungsaufwand in der Entwicklungsumgebung.

ADI_ADD_4



(Bild des Funktionsblocks nicht verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ADI_ADD_4` dient zur Durchführung einer arithmetischen Addition von bis zu vier Werten. Es handelt sich hierbei um einen generischen Funktionsbaustein (Generic FB), der auf dem unidirektionalen Adaptertyp `ADI` basiert. Er ermöglicht eine übersichtliche und modularisierte Addition innerhalb von IEC 61499 Anwendungen in der 4diac-IDE.

Schnittstellenstruktur

Da der Baustein vollständig auf Adaptern basiert, verfügt er über keine klassischen, separaten Ereignis- oder Datenkanäle auf der Hauptebene des Typs.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung wird über die angebundenen Adapter realisiert.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Ereignisweiterleitung wird über den Ausgangs-Adapter realisiert.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

- **Sockets (Eingangs-Adapter / Buchsen):**
  - `IN1` (Typ: `adapter::types::unidirectional::ADI`): Eingang für den ersten Summanden.
  - `IN2` (Typ: `adapter::types::unidirectional::ADI`): Eingang für den zweiten Summanden.
  - `IN3` (Typ: `adapter::types::unidirectional::ADI`): Eingang für den dritten Summanden.
  - `IN4` (Typ: `adapter::types::unidirectional::ADI`): Eingang für den vierten Summanden.
- **Plugs (Ausgangs-Adapter / Stecker):**
  - `OUT` (Typ: `adapter::types::unidirectional::ADI`): Ausgang für das berechnete Gesamtergebnis der Addition.

Funktionsweise

Sobald an den Eingangs-Adaptern (`IN1` bis `IN4`) neue Datenwerte oder entsprechende Trigger-Ereignisse anliegen, führt der Funktionsbaustein die Addition der Werte durch. Das Ergebnis wird berechnet nach der Formel:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} + \mathrm{IN4} \tag{2.8}$$

Das berechnete Resultat sowie das zugehörige Trigger-Ereignis werden direkt über den Ausgangs-Adapter `OUT` bereitgestellt.

Technische Besonderheiten

- **Generischer Baustein:** Durch die Zuordnung des Attributs `GenericClassName` mit dem Wert `'GEN_ADI_ADD'` ist der Baustein flexibel aufgebaut. Er kann je nach Implementierung des zugrundeliegenden `ADI`-Adapters für verschiedene numerische Datentypen verwendet werden.
- **Adapterbasiertes Design:** Die Kapselung von Daten und Events in Adaptern (hier vom Typ `ADI`) minimiert den Verdrahtungsaufwand im Funktionsplan der 4diac-IDE drastisch und sorgt für ein sauberes, strukturiertes Anwendungsdesign.
- **Unidirektionale Kopplung:** Die Verwendung von unidirektionalen Adaptern stellt sicher, dass der Datenfluss strikt sequenziell von den Eingängen zum Ausgang verläuft.

Zustandsübersicht

Da es sich um einen adapterbasierten Baustein zur reinen arithmetischen Datenverarbeitung handelt, besitzt der `ADI_ADD_4` keine komplexe interne Zustandsmaschine (ECC). Seine Ausführung verhält sich rein reaktiv:

- **Trigger an einem der Eingänge (`IN1` - `IN4`):** Berechnung der Summe aller vier Eingänge und unmittelbare Weiterleitung des Ergebnisses an `OUT`.

Anwendungsszenarien

- **Sensordaten-Aggregation:** Zusammenführen von bis zu vier analogen Messwerten (z. B. vier einzelnen Durchflussmessern oder Temperatursensoren) zu einem Gesamtwert.
- **Sollwertberechnung mit Offsets:** Addition eines Hauptsollwerts mit bis zu drei verschiedenen Korrekturfaktoren oder Offsets in Regelungsprozessen.
- **Strukturierte Signalverarbeitung:** Einsatz in Systemen, bei denen die Signalverarbeitungskette durchgehend auf dem `ADI`-Adapterstandard basiert, um unnötiges Entpacken und Verpacken von Signalen zu vermeiden.

Vergleich mit ähnlichen Bausteinen

- **Standard-Addierer (z. B. `F_ADD`):** Ein klassischer IEC 61131-3 Addier-Baustein benötigt separate Datenleitungen und Event-Verbindungen. `ADI_ADD_4` bündelt diese in Adaptern, was die visuelle Komplexität im Editor reduziert.
- **Zweifach-Addierer (`ADI_ADD_2`):** Zur Addition von vier Werten müssten drei zweifache Addierer kaskadiert werden. Der `ADI_ADD_4` spart hierbei signifikant Platz und Ausführungszeit, indem er die Berechnung in einem einzigen Schritt bündelt.
- `ADI_ADD_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

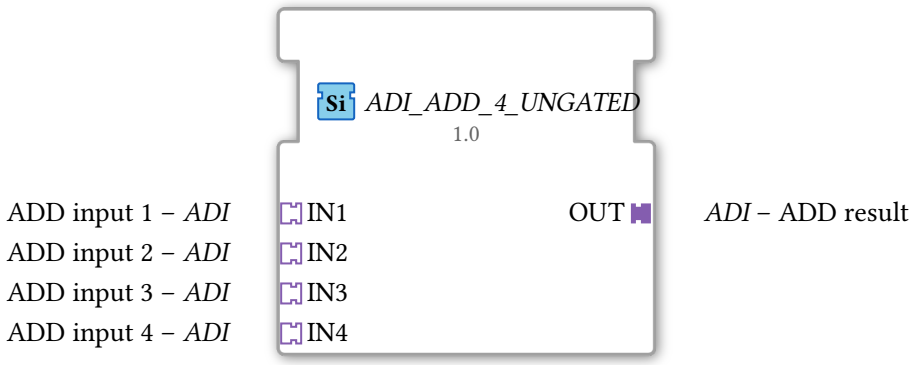
Fazit

Der `ADI_ADD_4` ist ein performanter und effizienter Funktionsbaustein zur Addition von vier numerischen Signalen. Durch die konsequente Nutzung von Adaptern reduziert er die Anzahl der physischen Verbindungen im Steuerungsprogramm und trägt maßgeblich zu einer übersichtlichen und wartungsfreundlichen Softwarearchitektur bei.

ADI_ADD_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ADI_ADD_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht

gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Bild des Funktionsblocks nicht verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ADI_ADD_4_UNGATED` dient zur Durchführung einer arithmetischen Addition von bis zu vier Werten. Es handelt sich hierbei um einen generischen Funktionsbaustein (Generic FB), der auf dem unidirektionalen Adaptertyp `ADI` basiert. Er ermöglicht eine übersichtliche und modularisierte Addition innerhalb von IEC 61499 Anwendungen in der 4diac-IDE.

Schnittstellenstruktur

Da der Baustein vollständig auf Adaptern basiert, verfügt er über keine klassischen, separaten Ereignis- oder Datenkanäle auf der Hauptebene des Typs.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung wird über die angebundenen Adapter realisiert.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Ereignisweiterleitung wird über den Ausgangs-Adapter realisiert.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

- **Sockets (Eingangs-Adapter / Buchsen):**
  - `IN1` (Typ: `adapter::types::unidirectional::ADI`): Eingang für den ersten Summanden.
  - `IN2` (Typ: `adapter::types::unidirectional::ADI`): Eingang für den zweiten Summanden.
  - `IN3` (Typ: `adapter::types::unidirectional::ADI`): Eingang für den dritten Summanden.
  - `IN4` (Typ: `adapter::types::unidirectional::ADI`): Eingang für den vierten Summanden.
- **Plugs (Ausgangs-Adapter / Stecker):**
  - `OUT` (Typ: `adapter::types::unidirectional::ADI`): Ausgang für das berechnete Gesamtergebnis der Addition.

Funktionsweise

Sobald an den Eingangs-Adaptern (`IN1` bis `IN4`) neue Datenwerte oder entsprechende Trigger-Ereignisse anliegen, führt der Funktionsbaustein die Addition der Werte durch. Das Ergebnis wird berechnet nach der Formel:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} + \mathrm{IN4} \tag{2.9}$$

Das berechnete Resultat sowie das zugehörige Trigger-Ereignis werden direkt über den Ausgangs-Adapter `OUT` bereitgestellt.

Technische Besonderheiten

- **Generischer Baustein:** Durch die Zuordnung des Attributs `GenericClassName` mit dem Wert `'GEN_ADI_ADD'` ist der Baustein flexibel aufgebaut. Er kann je nach Implementierung des zugrundeliegenden `ADI`-Adapters für verschiedene numerische Datentypen verwendet werden.
- **Adapterbasiertes Design:** Die Kapselung von Daten und Events in Adaptern (hier vom Typ `ADI`) minimiert den Verdrahtungsaufwand im Funktionsplan der 4diac-IDE drastisch und sorgt für ein sauberes, strukturiertes Anwendungsdesign.
- **Unidirektionale Kopplung:** Die Verwendung von unidirektionalen Adaptern stellt sicher, dass der Datenfluss strikt sequenziell von den Eingängen zum Ausgang verläuft.

Zustandsübersicht

Da es sich um einen adapterbasierten Baustein zur reinen arithmetischen Datenverarbeitung handelt, besitzt der `ADI_ADD_4_UNGATED` keine komplexe interne Zustandsmaschine (ECC). Seine Ausführung verhält sich rein reaktiv:

- **Trigger an einem der Eingänge (`IN1` - `IN4`):** Berechnung der Summe aller vier Eingänge und unmittelbare Weiterleitung des Ergebnisses an `OUT`.

Anwendungsszenarien

- **Sensordaten-Aggregation:** Zusammenführen von bis zu vier analogen Messwerten (z. B. vier einzelnen Durchflussmessern oder Temperatursensoren) zu einem Gesamtwert.
- **Sollwertberechnung mit Offsets:** Addition eines Hauptsollwerts mit bis zu drei verschiedenen Korrekturfaktoren oder Offsets in Regelungsprozessen.
- **Strukturierte Signalverarbeitung:** Einsatz in Systemen, bei denen die Signalverarbeitungskette durchgehend auf dem `ADI`-Adapterstandard basiert, um unnötiges Entpacken und Verpacken von Signalen zu vermeiden.

Vergleich mit ähnlichen Bausteinen

- **Standard-Addierer (z. B. `F_ADD`):** Ein klassischer IEC 61131-3 Addier-Baustein benötigt separate Datenleitungen und Event-Verbindungen. `ADI_ADD_4_UNGATED` bündelt diese in Adaptern, was die visuelle Komplexität im Editor reduziert.
- **Zweifach-Addierer (`ADI_ADD_2`):** Zur Addition von vier Werten müssten drei zweifache Addierer kaskadiert werden. Der `ADI_ADD_4_UNGATED` spart hierbei signifikant Platz und Ausführungszeit, indem er die Berechnung in einem einzigen Schritt bündelt.
- `ADI_ADD_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

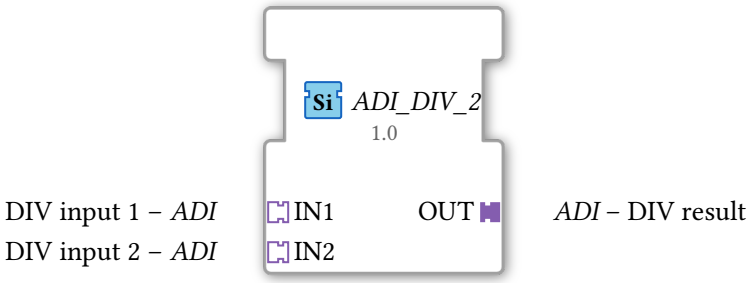
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `ADI_ADD_4_UNGATED` ist ein performanter und effizienter Funktionsbaustein zur Addition von vier numerischen Signalen. Durch die konsequente Nutzung von Adaptern reduziert er die Anzahl der physischen Verbindungen im Steuerungsprogramm und trägt maßgeblich zu einer übersichtlichen und wartungsfreundlichen Softwarearchitektur bei.

ADI_DIV_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ADI_DIV_2` dient zur Durchführung einer arithmetischen Division (DIV) von zwei Eingangswerten über Adapterverbindungen. Es handelt sich hierbei um einen generischen Funktionsbaustein (Generic FB), der auf dem Typ `GEN_ADI_DIV` basiert. Durch die konsequente Nutzung von unidirektionalen Adaptern des Typs `ADI` wird eine saubere Kapselung von Daten und Ereignissen erreicht. Dies vereinfacht das Applikationsdesign in der 4diac-IDE, da die Anzahl der sichtbaren Verbindungslinien signifikant reduziert wird.

Schnittstellenstruktur

Da dieser Funktionsbaustein vollständig adapterbasiert arbeitet, verfügt er auf der obersten Ebene nicht über klassische, diskrete Ereignis- oder Datenanschlüsse. Die gesamte Interaktion findet über die Adapter-Schnittstellen statt.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung wird über die Eingangs-Adapter (`IN1`, `IN2`) abgewickelt.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Ereignisausgabe wird über den Ausgangs-Adapter (`OUT`) abgewickelt.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

- **Sockets (Eingangs-Adapter):**
  - **`IN1`** (Typ: `adapter::types::unidirectional::ADI`): Erster Eingangswert (Dividend / Zähler) für die Division.
  - **`IN2`** (Typ: `adapter::types::unidirectional::ADI`): Zweiter Eingangswert (Divisor / Nenner) für die Division.
- **Plugs (Ausgangs-Adapter):**
  - **`OUT`** (Typ: `adapter::types::unidirectional::ADI`): Das berechnete Ergebnis (Quotient) der Division.

Funktionsweise

Der Baustein dividiert den über den Adapter `IN1` empfangenen Wert durch den über den Adapter `IN2` empfangenen Wert:

$$\text{Ergebnis (OUT)} = \frac{\text{Wert von IN1}}{\text{Wert von IN2}} \tag{2.10}$$

Die Berechnung wird getriggert, sobald an den Eingangs-Adaptern entsprechende Aktualisierungsereignisse eingehen. Nach erfolgreicher Berechnung wird der neue Quotient zusammen mit einem

Ausgangsereignis über den Adapter `OUT` weitergegeben. Aufgrund der generischen Natur des Bausteins (`GEN_ADI_DIV`) passt sich das Verhalten an die zugrunde liegenden Datentypen der genutzten `ADI`-Adapter an.

Technische Besonderheiten

- **Generische Instanziierung:** Der Baustein nutzt das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `'GEN_ADI_DIV'`. Dies erlaubt eine flexible Typisierung je nach verwendetem Adaptersubtyp zur Laufzeit.
- **Kapselung durch Adapter:** Daten und die dazugehörigen Trigger-Ereignisse sind im Adaptertyp `ADI` gebündelt, was eine lose Kopplung der Softwarekomponenten begünstigt.
- **Division durch Null:** Bei der Implementierung und Anwendung muss darauf geachtet werden, dass der Wert am Adapter `IN2` (Divisor) ungleich Null ist. Eine Division durch Null muss anwendungsorientiert vor dem Baustein abgefangen werden, um Laufzeitfehler zu vermeiden.

Zustandsübersicht

Der Funktionsbaustein verhält sich zustandslos (bzw. rein reaktiv):

1. **Wartezustand:** Der Baustein wartet auf Ereignisse an den Sockets `IN1` oder `IN2`.
2. **Berechnung:** Nach Erhalt gültiger Daten/Ereignisse an den Eingängen wird die Division ausgeführt.
3. **Aktualisierung:** Das Ergebnis wird an den Ausgangs-Plug `OUT` übergeben und das dortige Sende-Ereignis ausgelöst.

Anwendungsszenarien

- **Messwertskalierung:** Normierung von analogen Sensorwerten (z. B. Division eines Rohwerts durch einen Skalierungsfaktor).
- **Berechnung von Verhältnissen:** Bestimmung von Mischungsverhältnissen oder Prozentwerten in verfahrenstechnischen Anlagen.
- **Mittelwertberechnungen:** Division einer Summe durch die Anzahl der erfassten Werte in einfachen mathematischen Berechnungsnetzwerken.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem klassischen, standardmäßigen `DIV`-Funktionsbaustein (nach IEC 61131-3), der mit diskreten Daten-Pins (`IN1`, `IN2`, `OUT`) und expliziten Event-Pins (`REQ`, `CNF`) arbeitet:

- **Vorteile von `ADI_DIV_2`:** Deutlich aufgeräumteres Erscheinungsbild im Funktionsplan (FBD) von 4diac. Datenkonsistenz wird durch die Kopplung im Adapter direkt erzwungen.
- **Nachteile:** Ein direkter Einsatz mit Standard-Datenströmen ohne Adapter-Kapselung ist nicht möglich. Es müssen dedizierte Konverterbausteine verwendet werden, falls die Signalquelle keine Adapter unterstützt.
- `ADI_DIV_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

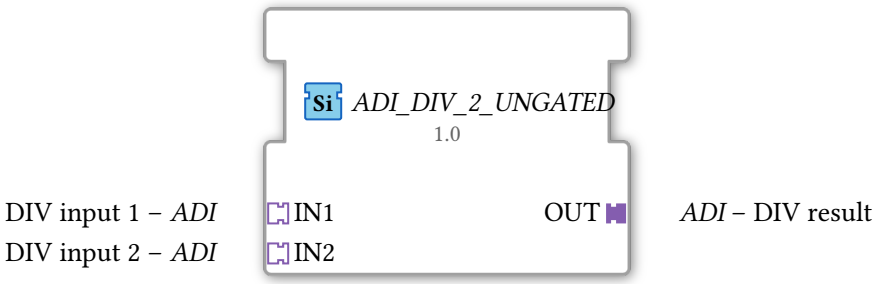
Fazit

`ADI_DIV_2` ist ein spezialisierter und effizienter Baustein für moderne IEC 61499 Architekturen innerhalb von 4diac. Er eignet sich hervorragend für Entwickler, die auf eine saubere, adapterbasierte und modulare Softwarearchitektur setzen und mathematische Divisionen übersichtlich implementieren möchten.

ADI_DIV_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ADI_DIV_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht

gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ADI_DIV_2_UNGATED` dient zur Durchführung einer arithmetischen Division (DIV) von zwei Eingangswerten über Adapterverbindungen. Es handelt sich hierbei um einen generischen Funktionsbaustein (Generic FB), der auf dem Typ `GEN_ADI_DIV` basiert. Durch die konsequente Nutzung von unidirektionalen Adaptern des Typs `ADI` wird eine saubere Kapselung von Daten und Ereignissen erreicht. Dies vereinfacht das Applikationsdesign in der 4diac-IDE, da die Anzahl der sichtbaren Verbindungslinien signifikant reduziert wird.

Schnittstellenstruktur

Da dieser Funktionsbaustein vollständig adapterbasiert arbeitet, verfügt er auf der obersten Ebene nicht über klassische, diskrete Ereignis- oder Datenanschlüsse. Die gesamte Interaktion findet über die Adapter-Schnittstellen statt.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung wird über die Eingangs-Adapter (`IN1`, `IN2`) abgewickelt.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Ereignisausgabe wird über den Ausgangs-Adapter (`OUT`) abgewickelt.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

- **Sockets (Eingangs-Adapter):**
  - **`IN1`** (Typ: `adapter::types::unidirectional::ADI`): Erster Eingangswert (Dividend / Zähler) für die Division.
  - **`IN2`** (Typ: `adapter::types::unidirectional::ADI`): Zweiter Eingangswert (Divisor / Nenner) für die Division.
- **Plugs (Ausgangs-Adapter):**
  - **`OUT`** (Typ: `adapter::types::unidirectional::ADI`): Das berechnete Ergebnis (Quotient) der Division.

Funktionsweise

Der Baustein dividiert den über den Adapter `IN1` empfangenen Wert durch den über den Adapter `IN2` empfangenen Wert:

$$\text{Ergebnis (OUT)} = \frac{\text{Wert von IN1}}{\text{Wert von IN2}} \tag{2.11}$$

Die Berechnung wird getriggert, sobald an den Eingangs-Adaptern entsprechende Aktualisierungsereignisse eingehen. Nach erfolgreicher Berechnung wird der neue Quotient zusammen mit einem Ausgangsereignis über den Adapter `OUT` weitergegeben. Aufgrund der generischen Natur des Bausteins (`GEN_ADI_DIV`) passt sich das Verhalten an die zugrunde liegenden Datentypen der genutzten `ADI`-Adapter an.

Technische Besonderheiten

- **Generische Instanziierung:** Der Baustein nutzt das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `'GEN_ADI_DIV'`. Dies erlaubt eine flexible Typisierung je nach verwendetem Adaptersubtyp zur Laufzeit.
- **Kapselung durch Adapter:** Daten und die dazugehörigen Trigger-Ereignisse sind im Adaptertyp `ADI` gebündelt, was eine lose Kopplung der Softwarekomponenten begünstigt.
- **Division durch Null:** Bei der Implementierung und Anwendung muss darauf geachtet werden, dass der Wert am Adapter `IN2` (Divisor) ungleich Null ist. Eine Division durch Null muss anwendungsorientiert vor dem Baustein abgefangen werden, um Laufzeitfehler zu vermeiden.

Zustandsübersicht

Der Funktionsbaustein verhält sich zustandslos (bzw. rein reaktiv):

1. **Wartezustand:** Der Baustein wartet auf Ereignisse an den Sockets `IN1` oder `IN2`.
2. **Berechnung:** Nach Erhalt gültiger Daten/Ereignisse an den Eingängen wird die Division ausgeführt.
3. **Aktualisierung:** Das Ergebnis wird an den Ausgangs-Plug `OUT` übergeben und das dortige Sende-Ereignis ausgelöst.

Anwendungsszenarien

- **Messwertskalierung:** Normierung von analogen Sensorwerten (z. B. Division eines Rohwerts durch einen Skalierungsfaktor).
- **Berechnung von Verhältnissen:** Bestimmung von Mischungsverhältnissen oder Prozentwerten in verfahrenstechnischen Anlagen.
- **Mittelwertberechnungen:** Division einer Summe durch die Anzahl der erfassten Werte in einfachen mathematischen Berechnungsnetzwerken.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem klassischen, standardmäßigen `DIV`-Funktionsbaustein (nach IEC 61131-3), der mit diskreten Daten-Pins (`IN1`, `IN2`, `OUT`) und expliziten Event-Pins (`REQ`, `CNF`) arbeitet:

- **Vorteile von `ADI_DIV_2_UNGATED`:** Deutlich aufgeräumteres Erscheinungsbild im Funktionsplan (FBD) von 4diac. Datenkonsistenz wird durch die Kopplung im Adapter direkt erzwungen.
- **Nachteile:** Ein direkter Einsatz mit Standard-Datenströmen ohne Adapter-Kapselung ist nicht möglich. Es müssen dedizierte Konverterbausteine verwendet werden, falls die Signalquelle keine Adapter unterstützt.
- `ADI_DIV_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- `ADI_DIV_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

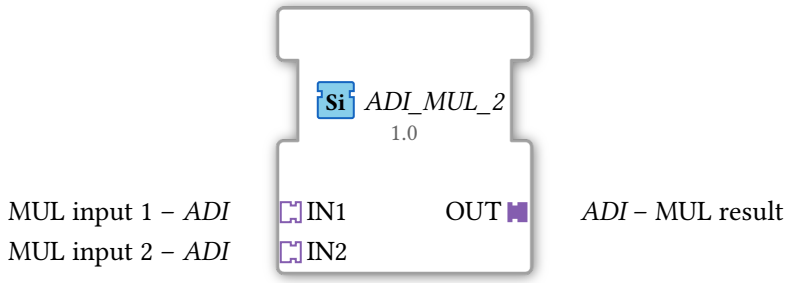
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

`ADI_DIV_2_UNGATED` ist ein spezialisierter und effizienter Baustein für moderne IEC 61499 Architekturen innerhalb von 4diac. Er eignet sich hervorragend für Entwickler, die auf eine saubere, adapterbasierte und modulare Softwarearchitektur setzen und mathematische Divisionen übersichtlich implementieren möchten.

ADI_MUL_2



(Kein Bild vorhanden)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ADI_MUL_2` ist ein generischer arithmetischer Multiplikationsbaustein für die 4diac-IDE, der auf dem IEC 61499-Standard basiert. Er dient dazu, zwei Eingangswerte miteinander zu multiplizieren. Die Besonderheit dieses Bausteins liegt in der Verwendung von unidirektionalen Adaptern (`ADI`) zur Datenübertragung anstelle von klassischen Daten- und Ereignis-Anschlüssen. Dies ermöglicht eine modulare und lose gekoppelte Signalverarbeitung.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine direkten Ereignis-Eingänge vorhanden. Die Steuerung und Synchronisation erfolgt implizit über die genutzten Adapter.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge vorhanden. Die Ereignisweiterleitung erfolgt gekoppelt über den Ausgangs-Adapter.

Daten-Eingänge

Der Baustein besitzt keine klassischen Daten-Eingänge. Die Datenaufnahme wird vollständig über die Adapter-Schnittstellen realisiert.

Daten-Ausgänge

Der Baustein besitzt keine klassischen Daten-Ausgänge. Die Datenausgabe wird über den Ausgangs-Adapter realisiert.

Adapter

Der Funktionsbaustein nutzt Adapter zur Kapselung von Daten und Ereignissen:

- **Sockets (Eingangs-Adapter):**
  - **IN1** (Typ: `adapter::types::unidirectional::ADI`): Erster Faktor für die Multiplikation (Multiplikand).
  - **IN2** (Typ: `adapter::types::unidirectional::ADI`): Zweiter Faktor für die Multiplikation (Multiplikator).
- **Plugs (Ausgangs-Adapter):**
  - **OUT** (Typ: `adapter::types::unidirectional::ADI`): Das Ergebnis der Multiplikation ($OUT = IN1 \times IN2$).

Funktionsweise

Sobald über die Adapter `IN1` und `IN2` gültige Werte und die dazugehörigen Trigger-Ereignisse anliegen, führt der Baustein die arithmetische Multiplikation aus:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \tag{2.12}$$

Das Ergebnis wird unmittelbar an den Ausgangs-Adapter `OUT` übergeben und steht für nachfolgende Bausteine zur Verfügung. Da der Baustein als generischer Typ deklariert ist (`GEN_ADI_MUL`), ist die genaue

Datentyp-Breite (z. B. INT, REAL, LREAL) flexibel und wird bei der Instanziierung in der 4diac-IDE basierend auf den verbundenen Adaptern bestimmt.

Technische Besonderheiten

- **Generische Implementierung:** Durch das Attribut `GenericClassName = 'GEN_ADI_MUL'` ist der Baustein datentypunabhängig konzipiert.
- **Kapselung durch Adapter:** Die Verwendung des unidirektionalen Adapters `ADI` reduziert die Anzahl der sichtbaren Linien im Funktionsplan, da Daten und Ereignisse in einer Verbindung gebündelt übertragen werden.
- **Compiler-Kontext:** Der Baustein ist im Paket `adapter::iec61131::arithmetic` organisiert und importiert die Klasse `eclipse4diac::core::GenericClassName`.

Zustandsübersicht

Der Funktionsbaustein verhält sich rein zustandslos (bzw. kombinatorisch). Es existiert keine interne State-Machine (ECC). Jede Aktivierung an den Eingangs-Adaptern führt direkt zur Berechnung des Ausgangswertes und zur Aktualisierung des Ausgangs-Adapters `OUT`.

Anwendungsszenarien

- **Modulare Signalverarbeitung:** Perfekt geeignet für Steuerungsarchitekturen, die konsequent auf Adapter-Verbindungen setzen, um die Übersichtlichkeit von komplexen Funktionsplänen zu wahren.
- **Skalierbare Berechnungen:** Einsatz in mathematischen Berechnungsnetzwerken innerhalb von IEC 61499 Anwendungen, bei denen unterschiedliche numerische Datentypen multipliziert werden müssen.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-Multiplikationsbaustein (wie z. B. dem klassischen `MUL`-Baustein aus der IEC 61131-3-Bibliothek), der mit expliziten Daten-Pins (`IN1`, `IN2`, `OUT`) und Ereignissen (`REQ`, `CNF`) arbeitet, entfällt beim `ADI_MUL_2` die manuelle Verdrahtung von Trigger-Ereignissen. Dies erhöht die Wiederverwendbarkeit und sorgt für ein saubereres Software-Design.

- `ADI_MUL_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

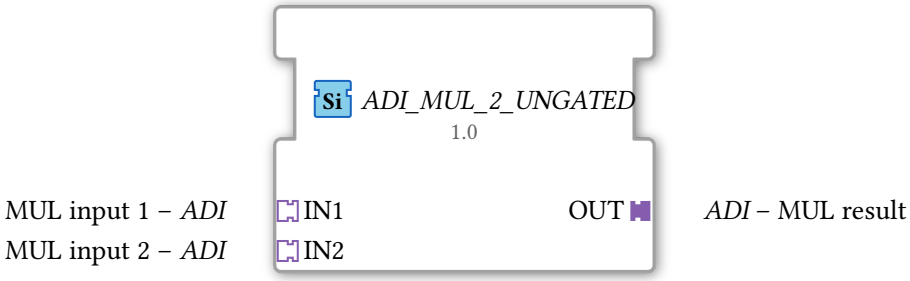
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

`ADI_MUL_2` ist ein moderner, flexibler und übersichtlich zu verdrahtender Multiplikationsbaustein. Durch die konsequente Nutzung von unidirektionalen Adaptern eignet er sich hervorragend für serviceorientierte Architekturen in der industriellen Automation und erleichtert die Erstellung sauber strukturierter IEC 61499-Applikationen.

ADI_MUL_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ADI_MUL_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild vorhanden)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ADI_MUL_2_UNGATED` ist ein generischer arithmetischer Multiplikationsbaustein für die 4diac-IDE, der auf dem IEC 61499-Standard basiert. Er dient dazu, zwei Eingangswerte miteinander zu multiplizieren. Die Besonderheit dieses Bausteins liegt in der Verwendung von unidirektionalen Adaptern (`ADI`) zur Datenübertragung anstelle von klassischen Daten- und Ereignis-Anschlüssen. Dies ermöglicht eine modulare und lose gekoppelte Signalverarbeitung.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine direkten Ereignis-Eingänge vorhanden. Die Steuerung und Synchronisation erfolgt implizit über die genutzten Adapter.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge vorhanden. Die Ereignisweiterleitung erfolgt gekoppelt über den Ausgangs-Adapter.

Daten-Eingänge

Der Baustein besitzt keine klassischen Daten-Eingänge. Die Datenaufnahme wird vollständig über die Adapter-Schnittstellen realisiert.

Daten-Ausgänge

Der Baustein besitzt keine klassischen Daten-Ausgänge. Die Datenausgabe wird über den Ausgangs-Adapter realisiert.

Adapter

Der Funktionsbaustein nutzt Adapter zur Kapselung von Daten und Ereignissen:

- **Sockets (Eingangs-Adapter):**
  - **IN1** (Typ: `adapter::types::unidirectional::ADI`): Erster Faktor für die Multiplikation (Multiplikand).
  - **IN2** (Typ: `adapter::types::unidirectional::ADI`): Zweiter Faktor für die Multiplikation (Multiplikator).
- **Plugs (Ausgangs-Adapter):**
  - **OUT** (Typ: `adapter::types::unidirectional::ADI`): Das Ergebnis der Multiplikation ($OUT = IN1 \times IN2$).

Funktionsweise

Sobald über die Adapter `IN1` und `IN2` gültige Werte und die dazugehörigen Trigger-Ereignisse anliegen, führt der Baustein die arithmetische Multiplikation aus:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \tag{2.13}$$

Das Ergebnis wird unmittelbar an den Ausgangs-Adapter `OUT` übergeben und steht für nachfolgende Bausteine zur Verfügung. Da der Baustein als generischer Typ deklariert ist (`GEN_ADI_MUL`), ist die genaue Datentyp-Breite (z. B. INT, REAL, LREAL) flexibel und wird bei der Instanziierung in der 4diac-IDE basierend auf den verbundenen Adaptern bestimmt.

Technische Besonderheiten

- **Generische Implementierung:** Durch das Attribut `GenericClassName = 'GEN_ADI_MUL'` ist der Baustein datentypunabhängig konzipiert.
- **Kapselung durch Adapter:** Die Verwendung des unidirektionalen Adapters `ADI` reduziert die Anzahl der sichtbaren Linien im Funktionsplan, da Daten und Ereignisse in einer Verbindung gebündelt übertragen werden.
- **Compiler-Kontext:** Der Baustein ist im Paket `adapter::iec61131::arithmetic` organisiert und importiert die Klasse `eclipse4diac::core::GenericClassName`.

Zustandsübersicht

Der Funktionsbaustein verhält sich rein zustandslos (bzw. kombinatorisch). Es existiert keine interne State-Machine (ECC). Jede Aktivierung an den Eingangs-Adaptern führt direkt zur Berechnung des Ausgangswertes und zur Aktualisierung des Ausgangs-Adapters `OUT`.

Anwendungsszenarien

- **Modulare Signalverarbeitung:** Perfekt geeignet für Steuerungsarchitekturen, die konsequent auf Adapter-Verbindungen setzen, um die Übersichtlichkeit von komplexen Funktionsplänen zu wahren.
- **Skalierbare Berechnungen:** Einsatz in mathematischen Berechnungsnetzwerken innerhalb von IEC 61499 Anwendungen, bei denen unterschiedliche numerische Datentypen multipliziert werden müssen.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-Multiplikationsbaustein (wie z. B. dem klassischen `MUL`-Baustein aus der IEC 61131-3-Bibliothek), der mit expliziten Daten-Pins (`IN1`, `IN2`, `OUT`) und Ereignissen (`REQ`, `CNF`) arbeitet, entfällt beim `ADI_MUL_2_UNGATED` die manuelle Verdrahtung von Trigger-Ereignissen. Dies erhöht die Wiederverwendbarkeit und sorgt für ein saubereres Software-Design.

- **`ADI_MUL_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- **`ADI_MUL_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

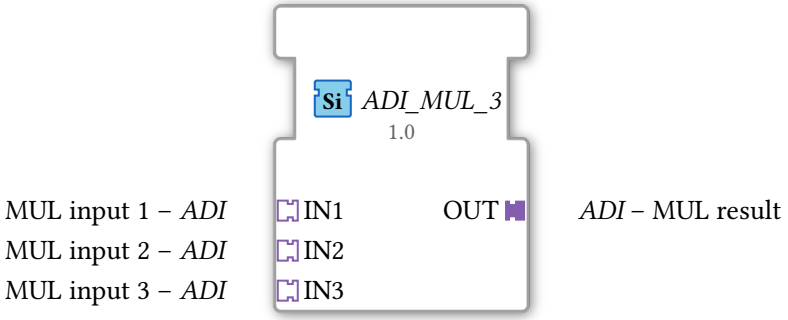
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

`ADI_MUL_2_UNGATED` ist ein moderner, flexibler und übersichtlich zu verdrahtender Multiplikationsbaustein. Durch die konsequente Nutzung von unidirektionalen Adaptern eignet er sich hervorragend für serviceorientierte Architekturen in der industriellen Automation und erleichtert die Erstellung sauber strukturierter IEC 61499-Applikationen.

ADI_MUL_3



Kein Bild verfügbar

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ADI_MUL_3` ist ein generischer Arithmetik-Baustein für die 4diac-IDE. Er dient zur Multiplikation von drei numerischen Eingangswerten, die über standardisierte, unidirektionale Adapter übertragen werden. Durch die Kapselung der mathematischen Operation in einer Adapter-Schnittstelle eignet sich dieser Baustein besonders für modulare und saubere Anwendungsarchitekturen innerhalb des IEC 61499-Standards.

Schnittstellenstruktur

Da es sich um einen adapterbasierten Funktionsbaustein handelt, besitzt der `ADI_MUL_3` keine direkten klassischen Ereignis- oder Daten-Pins auf oberster Ebene. Die gesamte Kommunikation wird über die Adapter-Schnittstellen abgewickelt.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden (Steuerung erfolgt über die Adapter).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden (Steuerung erfolgt über die Adapter).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::ADI`)
  - Erster Multiplikand für die Berechnungsoperation.
- **IN2** (Typ: `adapter::types::unidirectional::ADI`)
  - Zweiter Multiplikand für die Berechnungsoperation.
- **IN3** (Typ: `adapter::types::unidirectional::ADI`)
  - Dritter Multiplikand für die Berechnungsoperation.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::ADI`)
  - Ausgangsadapter, der das mathematische Produkt der drei Eingangswerte liefert.

Funktionsweise

Sobald an den Eingangs-Adaptern (`IN1`, `IN2`, `IN3`) neue Werte signalisiert werden, führt der Baustein die Multiplikation durch. Das Ergebnis wird nach folgender Formel berechnet:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \tag{2.14}$$

Das berechnete Ergebnis wird unmittelbar an den Ausgangs-Adapter `OUT` übergeben und ein entsprechendes Update-Ereignis über den Adapter weitergeleitet.

Technische Besonderheiten

- **Generischer Typ:** Der Baustein basiert auf der generischen Klasse `'GEN_ADI_MUL'`. Dies ermöglicht eine flexible Handhabung unterschiedlicher numerischer Datentypen (z. B. `INT`, `UINT`, `REAL`, `LREAL`), je nachdem, wie die zugrunde liegenden ADI-Adapter implementiert sind.
- **Unidirektionale Adapter:** Die Verwendung des Typs `unidirectional::ADI` sorgt für einen klaren Datenfluss in eine Richtung, was die Kopplung zwischen Programmteilen minimiert und die Systemstabilität erhöht.

Zustandsübersicht

Der Funktionsbaustein ist als rein mathematischer, zustandsloser Baustein (stateless) konzipiert. Er speichert keine internen Historienwerte. Jede Aktualisierung an den Eingängen führt zu einer direkten Neuberechnung des Ausgangs.

Anwendungsszenarien

- **Volumenberechnungen:** Berechnung des Volumens aus drei Dimensionen (Länge × Breite × Höhe).
- **Dreiphasen-Messungen:** Leistungs- oder Energieberechnungen, bei denen mehrere Faktoren (z. B. Strom, Spannung und ein Skalierungs-/Korrekturfaktor) miteinander multipliziert werden müssen.
- **Mehrstufige Skalierung:** Einlesen eines Sensorwerts, der mit einem Kalibrierungswert und einem zusätzlichen Verstärkungsfaktor multipliziert werden muss.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu klassischen IEC 61131-3 `MUL`-Bausteinen bietet der `ADI_MUL_3` durch die Verwendung von Adaptern den Vorteil, dass Ereignisse und Daten logisch in einer einzigen Verbindung gekapselt sind. Dies reduziert den Verdrahtungsaufwand in der 4diac-IDE erheblich. Im Vergleich zu einem zweifachen Multiplizierer (`ADI_MUL_2`) spart dieser Baustein zudem ein Kaskadieren von Blöcken ein, was die Ausführungszeit verkürzt und das Anwendungsdiagramm übersichtlicher gestaltet.

- `ADI_MUL_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

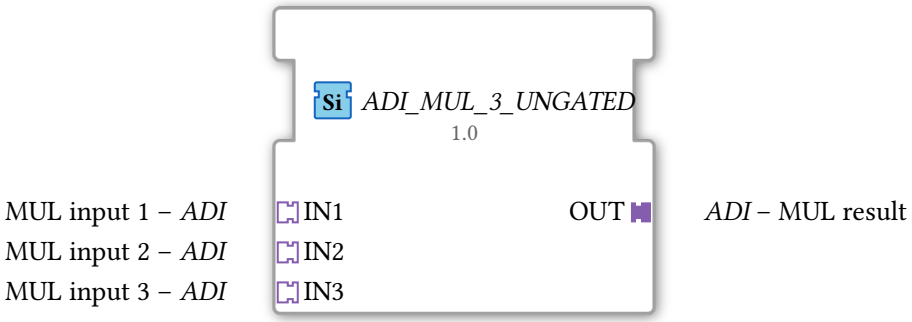
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `ADI_MUL_3` ist ein praktischer Hilfsbaustein für arithmetische Berechnungen in IEC 61499-Anwendungen. Durch die konsequente Nutzung von unidirektionalen Adaptern trägt er zu einem sauberen, lesbaren und wartungsfreundlichen Steuerungsentwurf bei.

ADI_MUL_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ADI_MUL_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



Kein Bild verfügbar

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ADI_MUL_3_UNGATED` ist ein generischer Arithmetik-Baustein für die 4diac-IDE. Er dient zur Multiplikation von drei numerischen Eingangswerten, die über standardisierte, unidirektionale Adapter übertragen werden. Durch die Kapselung der mathematischen Operation in einer Adapter-Schnittstelle eignet sich dieser Baustein besonders für modulare und saubere Anwendungsarchitekturen innerhalb des IEC 61499-Standards.

Schnittstellenstruktur

Da es sich um einen adapterbasierten Funktionsbaustein handelt, besitzt der `ADI_MUL_3_UNGATED` keine direkten klassischen Ereignis- oder Daten-Pins auf oberster Ebene. Die gesamte Kommunikation wird über die Adapter-Schnittstellen abgewickelt.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden (Steuerung erfolgt über die Adapter).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden (Steuerung erfolgt über die Adapter).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::ADI`)
  - Erster Multiplikand für die Berechnungsoperation.
- **IN2** (Typ: `adapter::types::unidirectional::ADI`)
  - Zweiter Multiplikand für die Berechnungsoperation.
- **IN3** (Typ: `adapter::types::unidirectional::ADI`)
  - Dritter Multiplikand für die Berechnungsoperation.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::ADI`)
  - Ausgangsadapter, der das mathematische Produkt der drei Eingangswerte liefert.

Funktionsweise

Sobald an den Eingangs-Adaptern (`IN1`, `IN2`, `IN3`) neue Werte signalisiert werden, führt der Baustein die Multiplikation durch. Das Ergebnis wird nach folgender Formel berechnet:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \tag{2.15}$$

Das berechnete Ergebnis wird unmittelbar an den Ausgangs-Adapter `OUT` übergeben und ein entsprechendes Update-Ereignis über den Adapter weitergeleitet.

Technische Besonderheiten

- **Generischer Typ:** Der Baustein basiert auf der generischen Klasse `'GEN_ADI_MUL'`. Dies ermöglicht eine flexible Handhabung unterschiedlicher numerischer Datentypen (z. B. `INT`, `UINT`, `REAL`, `LREAL`), je nachdem, wie die zugrunde liegenden ADI-Adapter implementiert sind.
- **Unidirektionale Adapter:** Die Verwendung des Typs `unidirectional::ADI` sorgt für einen klaren Datenfluss in eine Richtung, was die Kopplung zwischen Programmteilen minimiert und die Systemstabilität erhöht.

Zustandsübersicht

Der Funktionsbaustein ist als rein mathematischer, zustandsloser Baustein (stateless) konzipiert. Er speichert keine internen Historienwerte. Jede Aktualisierung an den Eingängen führt zu einer direkten Neuberechnung des Ausgangs.

Anwendungsszenarien

- **Volumenberechnungen:** Berechnung des Volumens aus drei Dimensionen (Länge $\times$ Breite $\times$ Höhe).
- **Dreiphasen-Messungen:** Leistungs- oder Energieberechnungen, bei denen mehrere Faktoren (z. B. Strom, Spannung und ein Skalierungs-/Korrekturfaktor) miteinander multipliziert werden müssen.
- **Mehrstufige Skalierung:** Einlesen eines Sensorwerts, der mit einem Kalibrierungswert und einem zusätzlichen Verstärkungsfaktor multipliziert werden muss.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu klassischen IEC 61131-3 `MUL`-Bausteinen bietet der `ADI_MUL_3_UNGATED` durch die Verwendung von Adaptern den Vorteil, dass Ereignisse und Daten logisch in einer einzigen Verbindung gekapselt sind. Dies reduziert den Verdrahtungsaufwand in der 4diac-IDE erheblich. Im Vergleich zu einem zweifachen Multiplizierer (`ADI_MUL_2`) spart dieser Baustein zudem ein Kaskadieren von Blöcken ein, was die Ausführungszeit verkürzt und das Anwendungsdiagramm übersichtlicher gestaltet.

- **`ADI_MUL_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

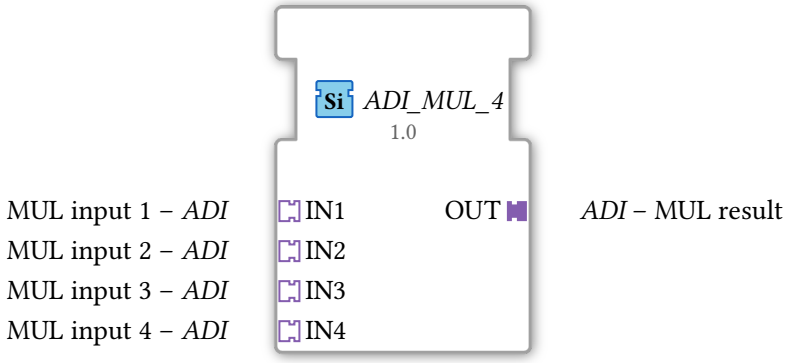
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `ADI_MUL_3_UNGATED` ist ein praktischer Hilfsbaustein für arithmetische Berechnungen in IEC 61499-Anwendungen. Durch die konsequente Nutzung von unidirektionalen Adaptern trägt er zu einem sauberen, lesbaren und wartungsfreundlichen Steuerungsentwurf bei.

ADI_MUL_4



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ADI_MUL_4` ist ein generischer, adapterbasierter Baustein für die IEC 61499, der zur Durchführung von arithmetischen Multiplikationen dient. Er berechnet das Produkt aus vier Eingangsgrößen, die über unidirektionale ADI-Adapter (`adapter::types::unidirectional::ADI`) bereitgestellt werden, und gibt das Ergebnis über einen entsprechenden Ausgangs-Adapter aus. Durch die Kapselung von Signalen in Adaptern sorgt dieser Baustein für ein übersichtlicheres Anwendungsdiagramm in der 4diac-IDE.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Baustein besitzt keine direkten Ereignis-Eingänge. Die Ereignissteuerung ist in den verwendeten Adaptern gekapselt.

Ereignis-Ausgänge

Dieser Baustein besitzt keine direkten Ereignis-Ausgänge. Die Ereignisweiterleitung ist in den verwendeten Adaptern gekapselt.

Daten-Eingänge

Dieser Baustein besitzt keine direkten Daten-Eingänge. Die Datenübergabe erfolgt über die Sockets.

Daten-Ausgänge

Dieser Baustein besitzt keine direkten Daten-Ausgänge. Die Datenausgabe erfolgt über den Plug.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::ADI`): Erster Faktor für die Multiplikation.
- **IN2** (Typ: `adapter::types::unidirectional::ADI`): Zweiter Faktor für die Multiplikation.
- **IN3** (Typ: `adapter::types::unidirectional::ADI`): Dritter Faktor für die Multiplikation.
- **IN4** (Typ: `adapter::types::unidirectional::ADI`): Vierter Faktor für die Multiplikation.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::ADI`): Ergebnis der Multiplikation ($OUT = IN1 \times IN2 \times IN3 \times IN4$).

Funktionsweise

Sobald an den Eingangs-Adaptern (`IN1` bis `IN4`) neue Werte und die dazugehörigen Ereignisse eintreffen, führt der Baustein die Multiplikation der vier Werte aus:

$$\mathrm{OUT} = \mathrm{IN1} \cdot \mathrm{IN2} \cdot \mathrm{IN3} \cdot \mathrm{IN4} \tag{2.16}$$

Das berechnete Ergebnis wird zusammen mit dem entsprechenden Trigger-Ereignis über den Ausgangs-Adapter `OUT` an die nachfolgenden Bausteine weitergeleitet. Da es sich um einen generischen Funktionsbaustein (`GEN_ADI_MUL`) handelt, passt sich das Berechnungsverhalten dem im Adapter definierten Datentyp an.

Technische Besonderheiten

- **Generischer Typ (`GEN_ADI_MUL`):** Der Baustein ist intern als generischer Baustein deklariert. Dies ermöglicht eine flexible Handhabung verschiedener Datentypen (z.B. `INT`, `REAL`, `LREAL`), je nach Spezifikation der verbundenen ADI-Adapter.
- **Kapselung durch Adapter:** Da sowohl Daten als auch Ereignisse über unidirektionale Adapter (`ADI`) geführt werden, verringert sich die Anzahl der sichtbaren Verbindungslinien in der 4diac-IDE drastisch. Dies erhöht die Übersichtlichkeit komplexer Steuerungsanwendungen.

Zustandsübersicht

Der Funktionsbaustein `ADI_MUL_4` ist ein **zustandsloser (stateless)** Rechenbaustein. Er besitzt keine interne State Machine (ECC). Jede Aktivierung durch ein Ereignis an den Eingangs-Adaptern führt direkt zur Neuberechnung des Ausgangs auf Basis der aktuell anliegenden Eingangswerte.

Anwendungsszenarien

- **Physikalische Berechnungen:** Berechnung von Volumina ($V = l \times b \times h$) mit einem zusätzlichen Skalierungs- oder Korrekturfaktor.
- **Messwertskalierung:** Mehrstufige Gewichtung oder Skalierung von analogen Sensorwerten über mehrere Faktoren hinweg.
- **Leistungsberechnungen:** Multiplikation von verschiedenen elektrischen oder mechanischen Kenngrößen zur Ermittlung von Gesamtleistungen oder Wirkungsgraden in der Prozessautomatisierung.

Vergleich mit ähnlichen Bausteinen

- **Standard `MUL`-Baustein (IEC 61131-3):** Klassische Multiplizierer nutzen direkte Daten- und Ereignispins. `ADI_MUL_4` hingegen nutzt Adapter, was das Routing vereinfacht, jedoch eine entsprechende Adapter-Infrastruktur im Projekt voraussetzt.
- **ADI_MUL_2 / ADI_MUL_3:** Ähnliche Bausteine mit weniger Eingängen. `ADI_MUL_4` eignet sich speziell dann, wenn exakt vier Faktoren miteinander multipliziert werden müssen, ohne mehrere 2-fach-Multiplizierer kaskadieren zu müssen.
- `ADI_MUL_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

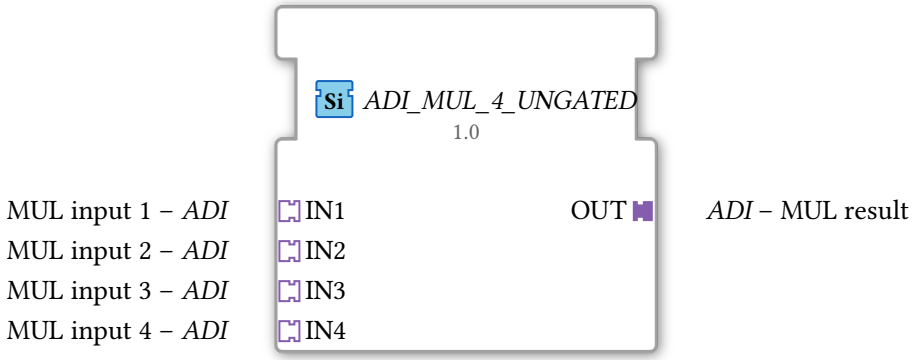
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `ADI_MUL_4` ist ein hocheffizienter Hilfsbaustein für die mathematische Signalverarbeitung in IEC 61499-Systemen. Durch die konsequente Nutzung von unidirektionalen Adaptern fördert er ein sauberes, modulares Software-Design und reduziert die Komplexität der grafischen Verdrahtung innerhalb der 4diac-IDE.

ADI_MUL_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ADI_MUL_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ADI_MUL_4_UNGATED` ist ein generischer, adapterbasierter Baustein für die IEC 61499, der zur Durchführung von arithmetischen Multiplikationen dient. Er berechnet das Produkt aus vier Eingangsgrößen, die über unidirektionale ADI-Adapter (`adapter::types::unidirectional::ADI`) bereitgestellt werden, und gibt das Ergebnis über einen entsprechenden Ausgangs-Adapter aus. Durch die Kapselung von Signalen in Adaptern sorgt dieser Baustein für ein übersichtlicheres Anwendungsdiagramm in der 4diac-IDE.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Baustein besitzt keine direkten Ereignis-Eingänge. Die Ereignissteuerung ist in den verwendeten Adaptern gekapselt.

Ereignis-Ausgänge

Dieser Baustein besitzt keine direkten Ereignis-Ausgänge. Die Ereignisweiterleitung ist in den verwendeten Adaptern gekapselt.

Daten-Eingänge

Dieser Baustein besitzt keine direkten Daten-Eingänge. Die Datenübergabe erfolgt über die Sockets.

Daten-Ausgänge

Dieser Baustein besitzt keine direkten Daten-Ausgänge. Die Datenausgabe erfolgt über den Plug.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::ADI`): Erster Faktor für die Multiplikation.
- **IN2** (Typ: `adapter::types::unidirectional::ADI`): Zweiter Faktor für die Multiplikation.
- **IN3** (Typ: `adapter::types::unidirectional::ADI`): Dritter Faktor für die Multiplikation.
- **IN4** (Typ: `adapter::types::unidirectional::ADI`): Vierter Faktor für die Multiplikation.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::ADI`): Ergebnis der Multiplikation ($OUT = IN1 \times IN2 \times IN3 \times IN4$).

Funktionsweise

Sobald an den Eingangs-Adaptern (`IN1` bis `IN4`) neue Werte und die dazugehörigen Ereignisse eintreffen, führt der Baustein die Multiplikation der vier Werte aus:

$$\mathrm{OUT} = \mathrm{IN1} \cdot \mathrm{IN2} \cdot \mathrm{IN3} \cdot \mathrm{IN4} \tag{2.17}$$

Das berechnete Ergebnis wird zusammen mit dem entsprechenden Trigger-Ereignis über den Ausgangs-Adapter `OUT` an die nachfolgenden Bausteine weitergeleitet. Da es sich um einen generischen Funktionsbaustein (`GEN_ADI_MUL`) handelt, passt sich das Berechnungsverhalten dem im Adapter definierten Datentyp an.

Technische Besonderheiten

- **Generischer Typ (`GEN_ADI_MUL`):** Der Baustein ist intern als generischer Baustein deklariert. Dies ermöglicht eine flexible Handhabung verschiedener Datentypen (z.B. `INT`, `REAL`, `LREAL`), je nach Spezifikation der verbundenen ADI-Adapter.
- **Kapselung durch Adapter:** Da sowohl Daten als auch Ereignisse über unidirektionale Adapter (`ADI`) geführt werden, verringert sich die Anzahl der sichtbaren Verbindungslinien in der 4diac-IDE drastisch. Dies erhöht die Übersichtlichkeit komplexer Steuerungsanwendungen.

Zustandsübersicht

Der Funktionsbaustein `ADI_MUL_4_UNGATED` ist ein **zustandsloser (stateless)** Rechenbaustein. Er besitzt keine interne State Machine (ECC). Jede Aktivierung durch ein Ereignis an den Eingangs-Adaptern führt direkt zur Neuberechnung des Ausgangs auf Basis der aktuell anliegenden Eingangswerte.

Anwendungsszenarien

- **Physikalische Berechnungen:** Berechnung von Volumina ($V = l \times b \times h$) mit einem zusätzlichen Skalierungs- oder Korrekturfaktor.
- **Messwertskalierung:** Mehrstufige Gewichtung oder Skalierung von analogen Sensorwerten über mehrere Faktoren hinweg.
- **Leistungsberechnungen:** Multiplikation von verschiedenen elektrischen oder mechanischen Kenngrößen zur Ermittlung von Gesamtleistungen oder Wirkungsgraden in der Prozessautomatisierung.

Vergleich mit ähnlichen Bausteinen

- **Standard `MUL`-Baustein (IEC 61131-3):** Klassische Multiplizierer nutzen direkte Daten- und Ereignispins. `ADI_MUL_4_UNGATED` hingegen nutzt Adapter, was das Routing vereinfacht, jedoch eine entsprechende Adapter-Infrastruktur im Projekt voraussetzt.
- **ADI_MUL_2 / ADI_MUL_3:** Ähnliche Bausteine mit weniger Eingängen. `ADI_MUL_4_UNGATED` eignet sich speziell dann, wenn exakt vier Faktoren miteinander multipliziert werden müssen, ohne mehrere 2-fach-Multiplizierer kaskadieren zu müssen.
- `ADI_MUL_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

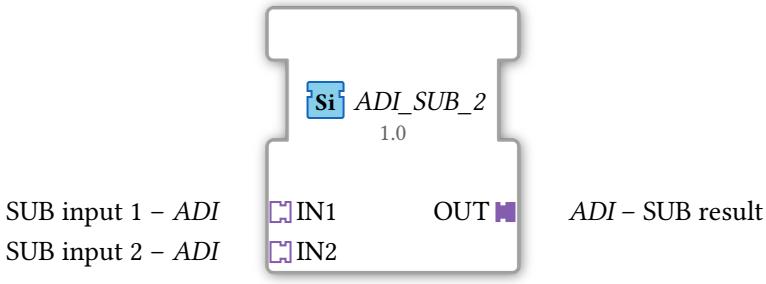
Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `ADI_MUL_4_UNGATED` ist ein hocheffizienter Hilfsbaustein für die mathematische Signalverarbeitung in IEC 61499-Systemen. Durch die konsequente Nutzung von unidirektionalen Adaptern fördert er ein sauberes, modulares Software-Design und reduziert die Komplexität der grafischen Verdrahtung innerhalb der 4diac-IDE.

ADI_SUB_2

```
       +-----------------------------------+
       |             ADI_SUB_2             |
       |                                   |
IN1 ==# [ADI]                         [ADI] #== OUT
IN2 ==# [ADI]                               |
       +-----------------------------------+
```



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ADI_SUB_2` ist ein generischer Funktionsbaustein (Generic FB) zur Durchführung von arithmetischen Subtraktionen in IEC 61499 Anwendungen. Der Baustein nutzt unidirektionale Adapter des Typs `ADI`, um Eingangsdaten zu empfangen und das berechnete Ergebnis auszugeben. Durch den Einsatz von Adaptern wird die Verdrahtung innerhalb der 4diac-IDE vereinfacht und die Modularität erhöht.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsbaustein besitzt keine direkten Ereignis-Eingänge. Die Ereignissteuerung wird vollständig über die angeschlossenen Adapter abgewickelt.

Ereignis-Ausgänge

Der Funktionsbaustein besitzt keine direkten Ereignis-Ausgänge. Die Ereignisweiterleitung erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge definiert. Daten werden über die Eingangs-Adapter übergeben.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge definiert. Das Ergebnis wird über den Ausgangs-Adapter bereitgestellt.

Adapter

Typ	Name	Adaptertyp	Beschreibung
Socket (Eingang)	`IN1`	`adapter::types::unidirectional::ADI`	Erster Eingangswert (Minuend) für die Subtraktion.
Socket (Eingang)	`IN2`	`adapter::types::unidirectional::ADI`	Zweiter Eingangswert (Subtrahend) für die Subtraktion.

Plug (Ausgang)	`OUT`	`adapter::types::unidirectional::ADI`	Ergebnis (Differenz) der Subtraktion ($OUT = IN1 - IN2$).

Funktionsweise

Sobald an den Eingangs-Adaptern `IN1` oder `IN2` ein entsprechendes Berechnungsereignis signalisiert wird, führt der Baustein die Subtraktion aus:

$$\mathrm{OUT} = \mathrm{IN1} - \mathrm{IN2} \tag{2.18}$$

Das Ergebnis wird an den Ausgangs-Adapter `OUT` übergeben, und das zugehörige Ausgangsereignis des Adapters wird ausgelöst, um nachfolgende Bausteine über den neuen Wert zu informieren.

Technische Besonderheiten

- **Generischer Typ:** Der Baustein ist als generischer Typ definiert (`eclipse4diac::core::GenericClassName` mit dem Wert `'GEN_ADI_SUB'`). Dies ermöglicht eine flexible Anpassung an unterschiedliche Datentypen innerhalb des `ADI`-Adapters.
- **Unidirektionale Adapter:** Durch die Verwendung der unidirektionalen Adapterstruktur (`adapter::types::unidirectional::ADI`) bleibt der Datenfluss gerichtet und zyklische Abhängigkeiten werden vermieden.

Zustandsübersicht

Da es sich um einen mathematischen Berechnungsbaustein handelt, ist das Verhalten im Wesentlichen zustandslos und ereignisgesteuert:

1. **Warten:** Der Baustein wartet auf ein Trigger-Ereignis an einem der Eingangs-Adapter (`IN1` oder `IN2`).
2. **Berechnung:** Bei Ereigniseingang werden die Datenwerte ausgelesen und subtrahiert.
3. **Ausgabe:** Der berechnete Wert wird auf den Adapter `OUT` geschrieben und das dortige Sende-Ereignis getriggert.

Anwendungsszenarien

- **Regelungstechnik:** Berechnung einer Regelabweichung ($e = w - x$), bei der Sollwert (w) und Istwert (x) über standardisierte ADI-Schnittstellen übertragen werden.
- **Messwertverarbeitung:** Differenzdruckmessung oder Offset-Korrektur von Sensorwerten, die über Adapter in die Applikation eingebunden sind.
- **Signalvorverarbeitung:** Reduzierung von Signalwerten um feste oder variable Grenzwerte vor der Weiterleitung an Aktoren.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-Subtraktionsbaustein (wie z.B. dem klassischen `SUB`-Baustein nach IEC 61131-3), der direkte Daten- und Ereignispins nutzt, bietet `ADI_SUB_2` folgende Vorteile:

- **Weniger Verbindungslinien:** Daten und Ereignisse sind im `ADI`-Adapter gebündelt, was das Anwendungsdiagramm in 4diac übersichtlicher macht.
- **Standardisierte Schnittstellen:** Erleichtert den Austausch von Berechnungs- und Verarbeitungsblöcken, da nur noch die Adapter verbunden werden müssen.
- **`ADI_SUB_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

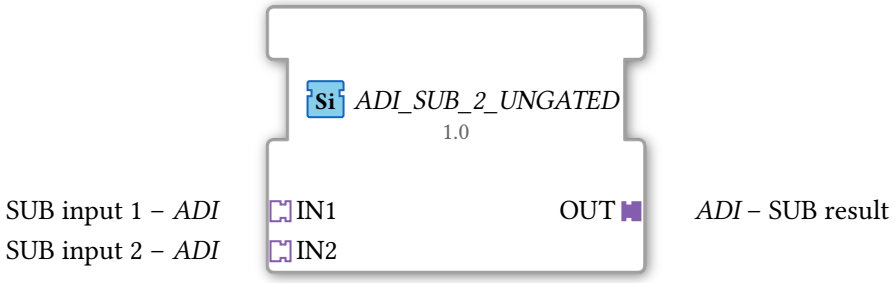
Fazit

Der `ADI_SUB_2` ist ein effizienter Hilfsbaustein für strukturierte IEC 61499 Steuerungsanwendungen. Er eignet sich hervorragend für anspruchsvolle Architekturen, bei denen Übersichtlichkeit und standardisierte Adapter-Schnittstellen im Vordergrund stehen.

ADI_SUB_2_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `ADI_SUB_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

```
      +-----------------------------------+
      |            ADI_SUB_2_UNGATED      |
      |                                   |
IN1 ==# [ADI]                       [ADI] #== OUT
IN2 ==# [ADI]                             |
      +-----------------------------------+
```



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ADI_SUB_2_UNGATED` ist ein generischer Funktionsbaustein (Generic FB) zur Durchführung von arithmetischen Subtraktionen in IEC 61499 Anwendungen. Der Baustein nutzt unidirektionale Adapter des Typs `ADI`, um Eingangsdaten zu empfangen und das berechnete Ergebnis auszugeben. Durch den Einsatz von Adaptern wird die Verdrahtung innerhalb der 4diac-IDE vereinfacht und die Modularität erhöht.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsbaustein besitzt keine direkten Ereignis-Eingänge. Die Ereignissteuerung wird vollständig über die angeschlossenen Adapter abgewickelt.

Ereignis-Ausgänge

Der Funktionsbaustein besitzt keine direkten Ereignis-Ausgänge. Die Ereignisweiterleitung erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge definiert. Daten werden über die Eingangs-Adapter übergeben.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge definiert. Das Ergebnis wird über den Ausgangs-Adapter bereitgestellt.

Adapter

Typ	Name	Adaptertyp	Beschreibung
Socket (Eingang)	`IN1`	`adapter::types::unidirectional::ADI`	Erster Eingangswert (Minuend) für die Subtraktion.

Socket (Eingang)	IN2	adapter::types::unidirectional::ADI	Zweiter Eingangswert (Subtrahend) für die Subtraktion.
Plug (Ausgang)	OUT	adapter::types::unidirectional::ADI	Ergebnis (Differenz) der Subtraktion ($OUT = IN1 - IN2$).

Funktionsweise

Sobald an den Eingangs-Adaptern IN1 oder IN2 ein entsprechendes Berechnungsereignis signalisiert wird, führt der Baustein die Subtraktion aus:

$$\mathrm{OUT} = \mathrm{IN1} - \mathrm{IN2} \tag{2.19}$$

Das Ergebnis wird an den Ausgangs-Adapter OUT übergeben, und das zugehörige Ausgangsereignis des Adapters wird ausgelöst, um nachfolgende Bausteine über den neuen Wert zu informieren.

Technische Besonderheiten

- **Generischer Typ:** Der Baustein ist als generischer Typ definiert (eclipse4diac::core::GenericClassName mit dem Wert 'GEN_ADI_SUB'). Dies ermöglicht eine flexible Anpassung an unterschiedliche Datentypen innerhalb des ADI-Adapters.
- **Unidirektionale Adapter:** Durch die Verwendung der unidirektionalen Adapterstruktur (adapter::types::unidirectional::ADI) bleibt der Datenfluss gerichtet und zyklische Abhängigkeiten werden vermieden.

Zustandsübersicht

Da es sich um einen mathematischen Berechnungsbaustein handelt, ist das Verhalten im Wesentlichen zustandslos und ereignisgesteuert:

1. **Warten:** Der Baustein wartet auf ein Trigger-Ereignis an einem der Eingangs-Adapter (IN1 oder IN2).
2. **Berechnung:** Bei Ereigniseingang werden die Datenwerte ausgelesen und subtrahiert.
3. **Ausgabe:** Der berechnete Wert wird auf den Adapter OUT geschrieben und das dortige Sende-Ereignis getriggert.

Anwendungsszenarien

- **Regelungstechnik:** Berechnung einer Regelabweichung ($e = w - x$), bei der Sollwert (w) und Istwert (x) über standardisierte ADI-Schnittstellen übertragen werden.
- **Messwertverarbeitung:** Differenzdruckmessung oder Offset-Korrektur von Sensorwerten, die über Adapter in die Applikation eingebunden sind.
- **Signalvorverarbeitung:** Reduzierung von Signalwerten um feste oder variable Grenzwerte vor der Weiterleitung an Aktoren.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-Subtraktionsbaustein (wie z.B. dem klassischen SUB-Baustein nach IEC 61131-3), der direkte Daten- und Ereignispins nutzt, bietet ADI_SUB_2_UNGATED folgende Vorteile:

- **Weniger Verbindungslinien:** Daten und Ereignisse sind im ADI-Adapter gebündelt, was das Anwendungsdiagramm in 4diac übersichtlicher macht.
- **Standardisierte Schnittstellen:** Erleichtert den Austausch von Berechnungs- und Verarbeitungsblöcken, da nur noch die Adapter verbunden werden müssen.
- **ADI_SUB_2_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- **ADI_SUB_2**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

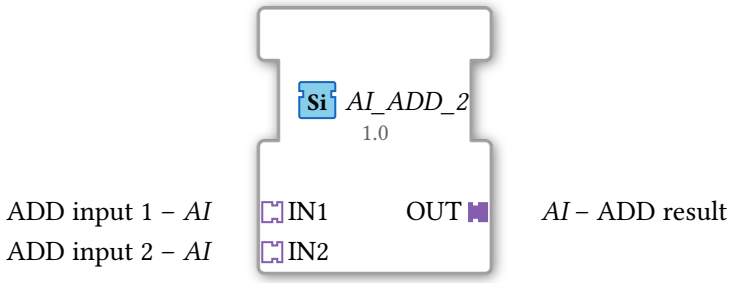
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `ADI_SUB_2_UNGATED` ist ein effizienter Hilfsbaustein für strukturierte IEC 61499 Steuerungsanwendungen. Er eignet sich hervorragend für anspruchsvolle Architekturen, bei denen Übersichtlichkeit und standardisierte Adapter-Schnittstellen im Vordergrund stehen.

AI_ADD_2



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein (FB) **AI_ADD_2** ist ein generischer Baustein zur Durchführung einer arithmetischen Addition zweier analoger Eingangswerte. Er basiert auf dem Konzept generischer Funktionsbausteine (`GEN_AI_ADD`) und nutzt Adapter-Schnittstellen anstelle von klassischen, lose gekoppelten Daten- und Ereignis-Anschlüssen. Dies ermöglicht eine strukturierte, übersichtliche und modulare Signalverarbeitung innerhalb von 4diac-ide-basierten Steuerungsanwendungen.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung erfolgt gekapselt über die verwendeten Adapter.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge vorhanden. Die Ereignissteuerung erfolgt gekapselt über die verwendeten Adapter.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Schnittstellen)

- **IN1** (Typ: `adapter::types::unidirectional::AI`): Der erste Summand der Additionsoperation. Über diesen Adapter werden sowohl der Wert als auch das dazugehörige Aktualisierungsereignis empfangen.
- **IN2** (Typ: `adapter::types::unidirectional::AI`): Der zweite Summand der Additionsoperation.

Plugs (Ausgangs-Schnittstellen)

- **OUT** (Typ: `adapter::types::unidirectional::AI`): Das Ergebnis der Addition ($OUT = IN1 + IN2$). Über diesen Adapter werden das Ergebnis und das entsprechende Aktualisierungsereignis an nachfolgende Bausteine weitergegeben.

Funktionsweise

Der Baustein **AI_ADD_2** fungiert als mathematischer Addierer. Sobald an einem der Eingangs-Adapter (`IN1` oder `IN2`) ein neues Ereignis signalisiert, dass sich die Eingangsdaten aktualisiert haben, führt der Baustein die Addition aus:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} \tag{2.20}$$

Das berechnete Ergebnis wird unmittelbar an den Ausgangs-Adapter `OUT` übergeben, und das zugehörige Aktualisierungsereignis wird ausgelöst, um nachfolgende Programmteile über den neuen Wert zu informieren.

Da es sich um einen generischen Funktionsbaustein (`GEN_AI_ADD`) handelt, ist der zugrundeliegende Datentyp (z. B. `REAL`, `INT`, `LREAL`) flexibel und wird durch die Belegung der verbundenen Adapter bestimmt.

Technische Besonderheiten

- **Generischer Typ (`GEN_AI_ADD`):** Der Baustein ist nicht auf einen festen Datentyp festgelegt, was seine Wiederverwendbarkeit für unterschiedliche analoge Signalbereiche erhöht.
- **Unidirektionale Adapter:** Durch die Verwendung des Typs `adapter::types::unidirectional::AI` wird der Verdrahtungsaufwand in der 4diac-ide drastisch reduziert, da Werte und Events in einer einzigen Verbindung gebündelt sind.
- **Saubere Kapselung:** Keine losen Event- und Datenleitungen auf der Oberfläche des Funktionsbausteins.

Zustandsübersicht

Der Funktionsbaustein verhält sich ereignisgesteuert und zustandslos. Es wird kein interner Zustand (wie bei einem Speicherglied) gehalten. Jedes eintreffende Ereignis an den Eingängen triggert sofort die Neuberechnung und die Ausgabe am Ausgang `OUT`.

Anwendungsszenarien

- **Sensordaten-Aggregation:** Addition zweier Durchflussmengen (z. B. Hauptstrom und Nebenstrom) zu einem Gesamtdurchfluss.
- **Sollwert-Offsetting:** Aufaddieren eines festen oder dynamischen Offsets (z. B. Kalibrierungswert) auf ein analoges Eingangssignal.
- **Signalmischung:** Einfache mathematische Zusammenführung zweier analoger Steuersignale in der Prozessautomatisierung.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zum Standard-IEC-61131-3-Additionsbaustein (`ADD`), welcher mit klassischen Daten-Pins (z. B. `IN1`, `IN2` vom Typ `ANY_NUM`) und expliziten Kontroll-Events (`REQ` / `CNF`) arbeitet, bietet **AI_ADD_2** eine deutlich aufgeräumtere visuelle Darstellung. Durch die Kapselung in Adaptern entfällt das manuelle Ziehen separater Event-Linien, was das Fehlerrisiko bei der Applikationserstellung minimiert.

- `AI_ADD_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

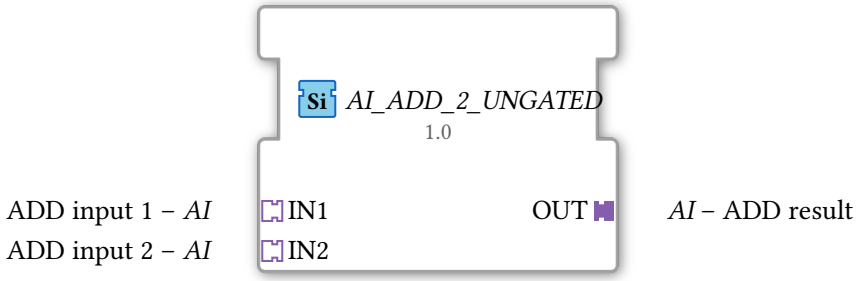
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AI_ADD_2** ist ein moderner, hocheffizienter Funktionsbaustein für die 4diac-ide. Er kombiniert die mathematische Grundfunktion der Addition mit den strukturellen Vorteilen von IEC 61499-Adaptern. Dies macht ihn zur idealen Wahl für modular aufgebaute Regelungs- und Steuerungsarchitekturen.

AI_ADD_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AI_ADD_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein (FB) **AI_ADD_2_UNGATED** ist ein generischer Baustein zur Durchführung einer arithmetischen Addition zweier analoger Eingangswerte. Er basiert auf dem Konzept generischer Funktionsbausteine (`GEN_AI_ADD`) und nutzt Adapter-Schnittstellen anstelle von klassischen, lose gekoppelten Daten- und Ereignis-Anschlüssen. Dies ermöglicht eine strukturierte, übersichtliche und modulare Signalverarbeitung innerhalb von 4diac-ide-basierten Steuerungsanwendungen.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung erfolgt gekapselt über die verwendeten Adapter.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge vorhanden. Die Ereignissteuerung erfolgt gekapselt über die verwendeten Adapter.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Schnittstellen)

- **IN1** (Typ: `adapter::types::unidirectional::AI`): Der erste Summand der Additionsoperation. Über diesen Adapter werden sowohl der Wert als auch das dazugehörige Aktualisierungsereignis empfangen.
- **IN2** (Typ: `adapter::types::unidirectional::AI`): Der zweite Summand der Additionsoperation.

Plugs (Ausgangs-Schnittstellen)

- **OUT** (Typ: `adapter::types::unidirectional::AI`): Das Ergebnis der Addition ($OUT = IN1 + IN2$). Über diesen Adapter werden das Ergebnis und das entsprechende Aktualisierungsereignis an nachfolgende Bausteine weitergegeben.

Funktionsweise

Der Baustein **AI_ADD_2_UNGATED** fungiert als mathematischer Addierer. Sobald an einem der Eingangs-Adapter (`IN1` oder `IN2`) ein neues Ereignis signalisiert, dass sich die Eingangsdaten aktualisiert haben, führt der Baustein die Addition aus:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} \tag{2.21}$$

Das berechnete Ergebnis wird unmittelbar an den Ausgangs-Adapter `OUT` übergeben, und das zugehörige Aktualisierungsereignis wird ausgelöst, um nachfolgende Programmteile über den neuen Wert zu informieren.

Da es sich um einen generischen Funktionsbaustein (`GEN_AI_ADD`) handelt, ist der zugrundeliegende Datentyp (z. B. `REAL`, `INT`, `LREAL`) flexibel und wird durch die Belegung der verbundenen Adapter bestimmt.

Technische Besonderheiten

- **Generischer Typ (`GEN_AI_ADD`):** Der Baustein ist nicht auf einen festen Datentyp festgelegt, was seine Wiederverwendbarkeit für unterschiedliche analoge Signalbereiche erhöht.
- **Unidirektionale Adapter:** Durch die Verwendung des Typs `adapter::types::unidirectional::AI` wird der Verdrahtungsaufwand in der 4diac-ide drastisch reduziert, da Werte und Events in einer einzigen Verbindung gebündelt sind.
- **Saubere Kapselung:** Keine losen Event- und Datenleitungen auf der Oberfläche des Funktionsbausteins.

Zustandsübersicht

Der Funktionsbaustein verhält sich ereignisgesteuert und zustandslos. Es wird kein interner Zustand (wie bei einem Speicherglied) gehalten. Jedes eintreffende Ereignis an den Eingängen triggert sofort die Neuberechnung und die Ausgabe am Ausgang `OUT`.

Anwendungsszenarien

- **Sensordaten-Aggregation:** Addition zweier Durchflussmengen (z. B. Hauptstrom und Nebenstrom) zu einem Gesamtdurchfluss.
- **Sollwert-Offsetting:** Aufaddieren eines festen oder dynamischen Offsets (z. B. Kalibrierungswert) auf ein analoges Eingangssignal.
- **Signalmischung:** Einfache mathematische Zusammenführung zweier analoger Steuersignale in der Prozessautomatisierung.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zum Standard-IEC-61131-3-Additionsbaustein (`ADD`), welcher mit klassischen Daten-Pins (z. B. `IN1`, `IN2` vom Typ `ANY_NUM`) und expliziten Kontroll-Events (`REQ` / `CNF`) arbeitet, bietet **AI_ADD_2_UNGATED** eine deutlich aufgeräumtere visuelle Darstellung. Durch die Kapselung in Adaptern entfällt das manuelle Ziehen separater Event-Linien, was das Fehlerrisiko bei der Applikationserstellung minimiert.

- `AI_ADD_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

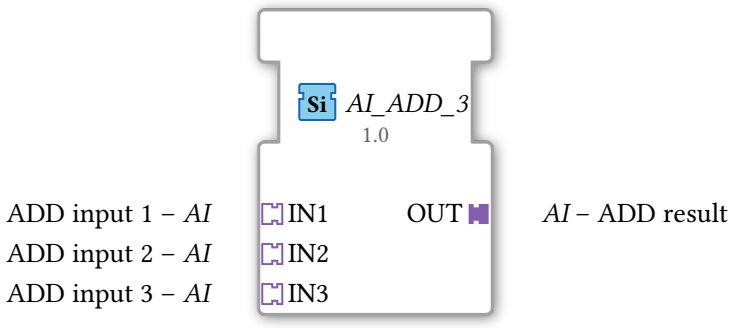
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der **AI_ADD_2_UNGATED** ist ein moderner, hocheffizienter Funktionsbaustein für die 4diac-ide. Er kombiniert die mathematische Grundfunktion der Addition mit den strukturellen Vorteilen von IEC 61499-Adaptern. Dies macht ihn zur idealen Wahl für modular aufgebaute Regelungs- und Steuerungsarchitekturen.

AI_ADD_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AI_ADD_3` ist ein generischer, arithmetischer Additionsbaustein für die Entwicklungsumgebung 4diac-ide. Er dient dazu, die Werte von drei analogen Eingängen (AI) zu addieren und das Ergebnis an einem analogen Ausgang bereitzustellen. Durch die Verwendung von unidirektionalen Adaptern wird eine saubere Kapselung von Daten und den dazugehörigen Ereignissen erreicht.

Schnittstellenstruktur

Der Baustein verwendet ein adapterbasiertes Schnittstellendesign. Er besitzt keine direkten, klassischen Ereignis- oder Datenkanäle auf der Hauptebene, sondern delegiert diese vollständig an die Adapter.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. (Die Ereignissteuerung erfolgt über die Adapter-Schnittstellen).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. (Die Ereignissteuerung erfolgt über die Adapter-Schnittstellen).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AI`): Erster analoger Eingangswert für die Additionsoperation.
- **IN2** (Typ: `adapter::types::unidirectional::AI`): Zweiter analoger Eingangswert für die Additionsoperation.
- **IN3** (Typ: `adapter::types::unidirectional::AI`): Dritter analoger Eingangswert für die Additionsoperation.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AI`): Ausgangsadapter, der das berechnete Ergebnis der Addition bereitstellt.

Funktionsweise

Sobald an einem der Eingangs-Adapter (`IN1`, `IN2` oder `IN3`) ein Aktualisierungsereignis eintrifft, liest der Funktionsbaustein die analogen Werte der drei Sockets aus. Es wird folgende mathematische Operation durchgeführt:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} \tag{2.22}$$

Nach erfolgreicher Berechnung wird der Ergebniswert an den Ausgangs-Plug `OUT` übergeben und synchron ein entsprechendes Aktualisierungsereignis am Ausgang ausgelöst.

Technische Besonderheiten

- **Generischer Baustein (Generic FB):** Der Baustein ist intern als `GEN_AI_ADD` typisiert. Dies ermöglicht eine flexible Anpassung an unterschiedliche numerische Datentypen (z. B. `INT`, `REAL`, `LREAL`), je nachdem, welcher konkrete Typ durch die verbundenen Adapter vorgegeben wird.
- **Adapter-Kopplung:** Durch die Nutzung von unidirektionalen Adaptern (`unidirectional::AI`) wird das Signal-Routing im Applikationsdiagramm stark vereinfacht. Daten und Trigger-Events werden in einer einzigen Verbindungslinie gebündelt.

Zustandsübersicht

Der Baustein arbeitet rein ereignisgesteuert:

1. **Bereitschaftszustand (Idle):** Der Baustein wartet auf eingehende Ereignisse an den Sockets `IN1`, `IN2` oder `IN3`.
2. **Berechnungsphase:** Ein Event triggert die Addition der anliegenden Werte.
3. **Aktualisierungsphase:** Das Ergebnis wird an `OUT` angelegt und das Ausgangs-Event getriggert. Der Baustein kehrt sofort in den Bereitschaftszustand zurück.

Anwendungsszenarien

- **Sensordaten-Fusion:** Zusammenführung und Aufsummierung von drei analogen Messwerten (z. B. Ermittlung eines Gesamtvolumenstroms aus drei einzelnen Teilströmen).
- **Sollwert-Kalkulation:** Berechnung eines Gesamtsollwerts, der sich aus einem Basissollwert und zwei Korrektur- oder Offsetwerten zusammensetzt.
- **Mischprozesse:** Gewichtung und Zusammenführung von analogen Steuerungssignalen in der Prozessautomatisierung.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu klassischen IEC 61131-3 Addierern (wie z. B. `F_ADD`) oder Standard-IEC-61499-Funktionsbausteinen entfällt bei `AI_ADD_3` die manuelle und fehleranfällige Verdrahtung von separaten `REQ`- und `CNF`-Ereignisleitungen.

Während Standard-Addierer meist nur zwei Eingänge unterstützen, spart der `AI_ADD_3` bei der Addition von drei Werten die Kaskadierung von zwei separaten Bausteinen ein, was die Ausführungszeit optimiert und die Übersichtlichkeit im System erhöht.

- `AI_ADD_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

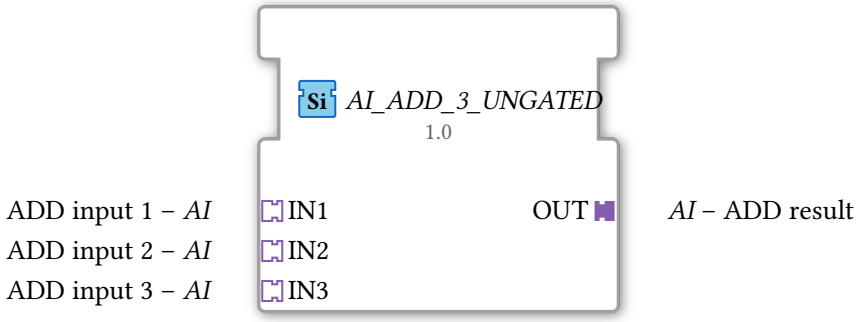
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `AI_ADD_3` ist ein modularer und übersichtlicher Hilfsbaustein zur effizienten analogen Signalverarbeitung. Durch die konsequente Nutzung von Adaptern reduziert er das visuelle „Spaghetti-Wiring“ in 4diac-Anwendungen erheblich und trägt zu einer performanten, ereignisgesteuerten Steuerungsarchitektur bei.

AI_ADD_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AI_ADD_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AI_ADD_3_UNGATED` ist ein generischer, arithmetischer Additionsbaustein für die Entwicklungsumgebung 4diac-ide. Er dient dazu, die Werte von drei analogen Eingängen (AI) zu addieren und das Ergebnis an einem analogen Ausgang bereitzustellen. Durch die Verwendung von unidirektionalen Adaptern wird eine saubere Kapselung von Daten und den dazugehörigen Ereignissen erreicht.

Schnittstellenstruktur

Der Baustein verwendet ein adapterbasiertes Schnittstellendesign. Er besitzt keine direkten, klassischen Ereignis- oder Datenkanäle auf der Hauptebene, sondern delegiert diese vollständig an die Adapter.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. (Die Ereignissteuerung erfolgt über die Adapter-Schnittstellen).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. (Die Ereignissteuerung erfolgt über die Adapter-Schnittstellen).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AI`): Erster analoger Eingangswert für die Additionsoperation.
- **IN2** (Typ: `adapter::types::unidirectional::AI`): Zweiter analoger Eingangswert für die Additionsoperation.
- **IN3** (Typ: `adapter::types::unidirectional::AI`): Dritter analoger Eingangswert für die Additionsoperation.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AI`): Ausgangsadapter, der das berechnete Ergebnis der Addition bereitstellt.

Funktionsweise

Sobald an einem der Eingangs-Adapter (`IN1`, `IN2` oder `IN3`) ein Aktualisierungsereignis eintrifft, liest der Funktionsbaustein die analogen Werte der drei Sockets aus. Es wird folgende mathematische Operation durchgeführt:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} \tag{2.23}$$

Nach erfolgreicher Berechnung wird der Ergebniswert an den Ausgangs-Plug `OUT` übergeben und synchron ein entsprechendes Aktualisierungsereignis am Ausgang ausgelöst.

Technische Besonderheiten

- **Generischer Baustein (Generic FB):** Der Baustein ist intern als `GEN_AI_ADD` typisiert. Dies ermöglicht eine flexible Anpassung an unterschiedliche numerische Datentypen (z. B. `INT`, `REAL`, `LREAL`), je nachdem, welcher konkrete Typ durch die verbundenen Adapter vorgegeben wird.
- **Adapter-Kopplung:** Durch die Nutzung von unidirektionalen Adaptern (`unidirectional::AI`) wird das Signal-Routing im Applikationsdiagramm stark vereinfacht. Daten und Trigger-Events werden in einer einzigen Verbindungslinie gebündelt.

Zustandsübersicht

Der Baustein arbeitet rein ereignisgesteuert:

1. **Bereitschaftszustand (Idle):** Der Baustein wartet auf eingehende Ereignisse an den Sockets `IN1`, `IN2` oder `IN3`.
2. **Berechnungsphase:** Ein Event triggert die Addition der anliegenden Werte.
3. **Aktualisierungsphase:** Das Ergebnis wird an `OUT` angelegt und das Ausgangs-Event getriggert. Der Baustein kehrt sofort in den Bereitschaftszustand zurück.

Anwendungsszenarien

- **Sensordaten-Fusion:** Zusammenführung und Aufsummierung von drei analogen Messwerten (z. B. Ermittlung eines Gesamtvolumenstroms aus drei einzelnen Teilströmen).
- **Sollwert-Kalkulation:** Berechnung eines Gesamtsollwerts, der sich aus einem Basissollwert und zwei Korrektur- oder Offsetwerten zusammensetzt.
- **Mischprozesse:** Gewichtung und Zusammenführung von analogen Steuerungssignalen in der Prozessautomatisierung.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu klassischen IEC 61131-3 Addierern (wie z. B. `F_ADD`) oder Standard-IEC-61499-Funktionsbausteinen entfällt bei `AI_ADD_3_UNGATED` die manuelle und fehleranfällige Verdrahtung von separaten `REQ`- und `CNF`-Ereignisleitungen.

Während Standard-Addierer meist nur zwei Eingänge unterstützen, spart der `AI_ADD_3_UNGATED` bei der Addition von drei Werten die Kaskadierung von zwei separaten Bausteinen ein, was die Ausführungszeit optimiert und die Übersichtlichkeit im System erhöht.

- `AI_ADD_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

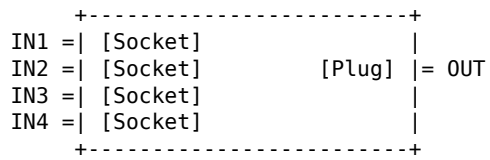
Änderungserkennung

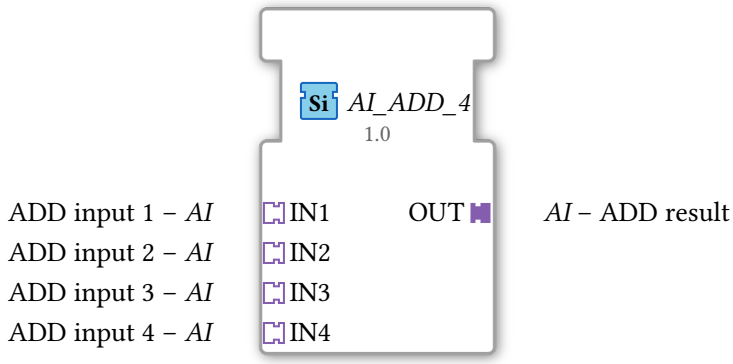
Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AI_ADD_3_UNGATED` ist ein modularer und übersichtlicher Hilfsbaustein zur effizienten analogen Signalverarbeitung. Durch die konsequente Nutzung von Adaptern reduziert er das visuelle „Spaghetti-Wiring“ in 4diac-Anwendungen erheblich und trägt zu einer performanten, ereignisgesteuerten Steuerungsarchitektur bei.

AI_ADD_4





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AI_ADD_4` ist ein generischer Arithmetik-Baustein für die 4diac-IDE, der zur Addition von vier analogen bzw. numerischen Werten dient. Die Besonderheit dieses Bausteins liegt in der Verwendung von unidirektionalen Adaptern des Typs `AI` (Analog Input/Arithmetic Input) anstelle von klassischen diskreten Daten- und Ereigniseingängen. Dies ermöglicht eine saubere, modulare und übersichtliche Strukturierung von Signalflüssen in IEC 61499 Anwendungen.

Schnittstellenstruktur

Der Baustein besitzt keine klassischen ereignis- oder datenbasierten Ein- und Ausgänge auf der obersten Ebene. Die gesamte Kommunikation wird über Adapter abgewickelt.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung ist in den Adaptern gekapselt.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Ereignissteuerung ist in den Adaptern gekapselt.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AI`): Erster Summand für die Additionsoperation.
- **IN2** (Typ: `adapter::types::unidirectional::AI`): Zweiter Summand für die Additionsoperation.
- **IN3** (Typ: `adapter::types::unidirectional::AI`): Dritter Summand für die Additionsoperation.
- **IN4** (Typ: `adapter::types::unidirectional::AI`): Vierter Summand für die Additionsoperation.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AI`): Ausgangsadapter, der das Ergebnis der Addition (`IN1 + IN2 + IN3 + IN4`) bereitstellt.

Funktionsweise

Sobald an einem der Eingangs-Adapter (`IN1` bis `IN4`) ein neues Ereignis signalisiert, dass sich ein Wert geändert hat, führt der Baustein intern die Addition durch:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} + \mathrm{IN4} \tag{2.24}$$

Das Ergebnis wird unmittelbar an den Ausgangs-Adapter `OUT` übergeben, und das entsprechende Aktualisierungsereignis des Ausgangs-Adapters wird ausgelöst. Da es sich um einen generischen Baustein (`GEN_AI_ADD`) handelt, richtet sich der tatsächliche Datentyp (z. B. `REAL`, `INT`, `LREAL`) nach der konkreten Implementierung und Instanziierung des verwendeten Adaptertyps.

Technische Besonderheiten

- **Generisches Design:** Der Baustein nutzt das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `'GEN_AI_ADD'`. Er ist dadurch hochflexibel und kann auf unterschiedliche numerische Datentypen angewendet werden, sofern die verwendeten Adapter dies unterstützen.
- **Unidirektionale Adapter:** Durch die Verwendung von `unidirectional::AI`-Adaptern wird der Verdrahtungsaufwand im Funktionsplan drastisch reduziert, da Daten und Trigger-Ereignisse in einer einzigen Verbindung gebündelt sind.

Zustandsübersicht

Der Baustein verhält sich rein reaktiv (ereignisgesteuert):

1. **Wartezustand (Idle):** Der Baustein wartet auf ein Aktualisierungsereignis an einem der Sockets (`IN1` bis `IN4`).
2. **Berechnung:** Bei Eintreffen eines Ereignisses werden die aktuellen Werte aller vier Eingänge gelesen und addiert.
3. **Ausgabe:** Der berechnete Wert wird an den Plug `OUT` geschrieben und das dortige Ausgangsereignis getriggert. Der Baustein kehrt sofort in den Wartezustand zurück.

Anwendungsszenarien

- **Messwert-Agggregation:** Summierung von Strömen, Leistungen oder Durchflussmengen aus vier verschiedenen Quellen oder Sensoren.
- **Mittelwertvorbereitung:** Zusammenführung von vier analogen Messwerten zur anschließenden Division (z. B. zur gleitenden Mittelwertbildung oder Redundanzprüfung).
- **Sollwert-Offset-Berechnung:** Addition von Basis-Sollwerten, Korrekturwerten und Benutzer-Offsets in verfahrenstechnischen Anlagen.

Vergleich mit ähnlichen Bausteinen

Feature / Baustein	`AI_ADD_4`	Standard `ADD` (IEC 61131-3)
Schnittstellentyp	Adapter-basiert (`AI`)	Standard-Daten- und Ereignispins
Anzahl Eingänge	Fest (4)	Variabel / Fest (meist 2)
Verdrahtungsaufwand	Sehr gering (nur 5 Verbindungen)	Hoch (separate Event- und Datenlinien)
Typisierung	Generisch über Adapter	Explizit über FB-Typisierung (z.B. `ADD_REAL`)

- `AI_ADD_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

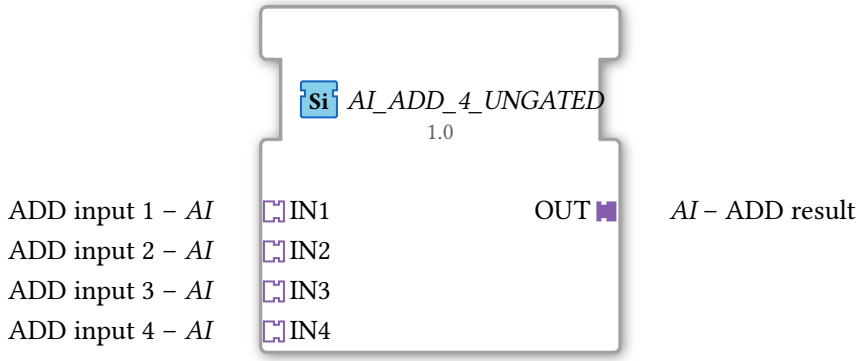
Der `AI_ADD_4` ist ein hocheffizienter und moderner Funktionsbaustein, der die Vorteile der adapterbasierten Kopplung in IEC 61499 voll ausschöpft. Er eignet sich hervorragend für komplexe Steuerungsarchitekturen, bei denen Übersichtlichkeit im Applikationsdiagramm und eine klare Signalstruktur im Vordergrund stehen.

AI_ADD_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AI_ADD_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht

gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

```
     +-------------------------+
IN1 =| [Socket]                |
IN2 =| [Socket]         [Plug] |= OUT
IN3 =| [Socket]                |
IN4 =| [Socket]                |
     +-------------------------+
```



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AI_ADD_4_UNGATED` ist ein generischer Arithmetik-Baustein für die 4diac-IDE, der zur Addition von vier analogen bzw. numerischen Werten dient. Die Besonderheit dieses Bausteins liegt in der Verwendung von unidirektionalen Adaptern des Typs `AI` (Analog Input/Arithmetic Input) anstelle von klassischen diskreten Daten- und Ereigniseingängen. Dies ermöglicht eine saubere, modulare und übersichtliche Strukturierung von Signalflüssen in IEC 61499 Anwendungen.

Schnittstellenstruktur

Der Baustein besitzt keine klassischen ereignis- oder datenbasierten Ein- und Ausgänge auf der obersten Ebene. Die gesamte Kommunikation wird über Adapter abgewickelt.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung ist in den Adaptern gekapselt.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Ereignissteuerung ist in den Adaptern gekapselt.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AI`): Erster Summand für die Additionsoperation.
- **IN2** (Typ: `adapter::types::unidirectional::AI`): Zweiter Summand für die Additionsoperation.
- **IN3** (Typ: `adapter::types::unidirectional::AI`): Dritter Summand für die Additionsoperation.
- **IN4** (Typ: `adapter::types::unidirectional::AI`): Vierter Summand für die Additionsoperation.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AI`): Ausgangsadapter, der das Ergebnis der Addition (`IN1 + IN2 + IN3 + IN4`) bereitstellt.

Funktionsweise

Sobald an einem der Eingangs-Adapter (`IN1` bis `IN4`) ein neues Ereignis signalisiert, dass sich ein Wert geändert hat, führt der Baustein intern die Addition durch:

$$\text{OUT} = \text{IN1} + \text{IN2} + \text{IN3} + \text{IN4} \tag{2.25}$$

Das Ergebnis wird unmittelbar an den Ausgangs-Adapter `OUT` übergeben, und das entsprechende Aktualisierungsereignis des Ausgangs-Adapters wird ausgelöst. Da es sich um einen generischen Baustein (`GEN_AI_ADD`) handelt, richtet sich der tatsächliche Datentyp (z. B. `REAL`, `INT`, `LREAL`) nach der konkreten Implementierung und Instanziierung des verwendeten Adaptertyps.

Technische Besonderheiten

- **Generisches Design:** Der Baustein nutzt das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `'GEN_AI_ADD'`. Er ist dadurch hochflexibel und kann auf unterschiedliche numerische Datentypen angewendet werden, sofern die verwendeten Adapter dies unterstützen.
- **Unidirektionale Adapter:** Durch die Verwendung von `unidirectional::AI`-Adaptern wird der Verdrahtungsaufwand im Funktionsplan drastisch reduziert, da Daten und Trigger-Ereignisse in einer einzigen Verbindung gebündelt sind.

Zustandsübersicht

Der Baustein verhält sich rein reaktiv (ereignisgesteuert):

1. **Wartezustand (Idle):** Der Baustein wartet auf ein Aktualisierungsereignis an einem der Sockets (`IN1` bis `IN4`).
2. **Berechnung:** Bei Eintreffen eines Ereignisses werden die aktuellen Werte aller vier Eingänge gelesen und addiert.
3. **Ausgabe:** Der berechnete Wert wird an den Plug `OUT` geschrieben und das dortige Ausgangsereignis getriggert. Der Baustein kehrt sofort in den Wartezustand zurück.

Anwendungsszenarien

- **Messwert-Agggregation:** Summierung von Strömen, Leistungen oder Durchflussmengen aus vier verschiedenen Quellen oder Sensoren.
- **Mittelwertvorbereitung:** Zusammenführung von vier analogen Messwerten zur anschließenden Division (z. B. zur gleitenden Mittelwertbildung oder Redundanzprüfung).
- **Sollwert-Offset-Berechnung:** Addition von Basis-Sollwerten, Korrekturwerten und Benutzer-Offsets in verfahrenstechnischen Anlagen.

Vergleich mit ähnlichen Bausteinen

Feature / Baustein	`AI_ADD_4_UNGATED`	Standard `ADD` (IEC 61131-3)
Schnittstellentyp	Adapter-basiert (`AI`)	Standard-Daten- und Ereignispins
Anzahl Eingänge	Fest (4)	Variabel / Fest (meist 2)
Verdrahtungsaufwand	Sehr gering (nur 5 Verbindungen)	Hoch (separate Event- und Datenlinien)
Typisierung	Generisch über Adapter	Explizit über FB-Typisierung (z.B. `ADD_REAL`)

- **`AI_ADD_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

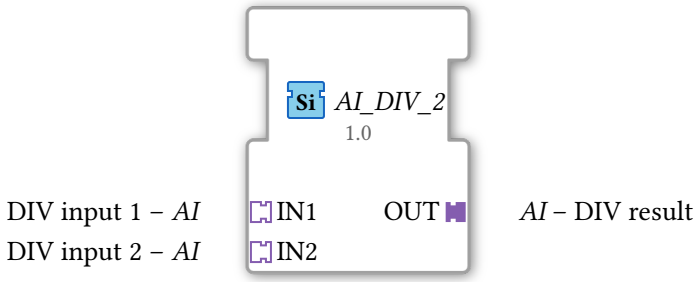
Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AI_ADD_4_UNGATED` ist ein hocheffizienter und moderner Funktionsbaustein, der die Vorteile der adapterbasierten Kopplung in IEC 61499 voll ausschöpft. Er eignet sich hervorragend für komplexe Steue-

rungsarchitekturen, bei denen Übersichtlichkeit im Applikationsdiagramm und eine klare Signalstruktur im Vordergrund stehen.

AI_DIV_2



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AI_DIV_2` ist ein generischer Arithmetik-Baustein für die 4diac-Plattform. Er dient zur Durchführung einer mathematischen Division (DIV) von zwei analogen Werten. Im Gegensatz zu klassischen Arithmetikbausteinen nutzt dieser Baustein standardisierte, unidirektionale Adapter vom Typ `AI` (Analog Input) für die Ein- und Ausgänge. Dies ermöglicht eine strukturierte, modulare und übersichtliche Signalverdrahtung innerhalb von IEC 61499 Anwendungen.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine direkten Ereignis-Eingänge. Die Synchronisation und Ereignissteuerung erfolgt implizit über die angebundenen Adapter.

Ereignis-Ausgänge

Der Baustein besitzt keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübertragung wird vollständig über die Adapter-Schnittstellen abgewickelt.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden.

Adapter

Name	Typ	Schnittstellen-Typ	Beschreibung
IN1	`adapter::types::unidirectional::AI`	Socket (Buchse)	Erster Eingangswert (Dividend)
IN2	`adapter::types::unidirectional::AI`	Socket (Buchse)	Zweiter Eingangswert (Divisor)
OUT	`adapter::types::unidirectional::AI`	Plug (Stecker)	Ergebniswert der Division (Quotient)

Funktionsweise

Der Funktionsbaustein berechnet den Quotienten aus den Werten der beiden Eingangs-Adapter:

$$OUT = \frac{IN1}{IN2} \tag{2.26}$$

Sobald sich die Werte an den Sockets `IN1` oder `IN2` ändern und ein entsprechendes Aktualisierungsereignis über den Adapter eingeht, wird die Division ausgeführt. Das Ergebnis sowie das dazugehörige Aktualisierungsereignis werden unmittelbar an den Ausgangs-Adapter `OUT` (Plug) weitergeleitet.

Aufgrund des generischen Typs (GEN_AI_DIV) ist der Baustein nicht auf einen festen Datentyp (wie z. B. REAL oder INT) beschränkt. Der konkrete Datentyp wird flexibel bei der Instanziierung in der 4diac-IDE bestimmt.

Technische Besonderheiten

- **Generische Typisierung:** Über das Attribut GenericClassName mit dem Wert GEN_AI_DIV ist der Baustein polymorph einsetzbar und kann mit verschiedenen kompatiblen analogen Datentypen arbeiten.
- **Adapter-Kapselung:** Durch die Verwendung von unidirektionalen Adaptern (unidirectional::AI) werden Daten- und Ereignisleitungen in einer einzigen Verbindung gebündelt. Dies reduziert das Risiko von Verdrahtungsfehlern und erhöht die Übersichtlichkeit im Funktionsplan (FBD).
- **Division durch Null:** Da es sich um eine mathematische Division handelt, muss anlagenseitig bzw. in der vorgeschalteten Logik sichergestellt werden, dass der Wert an IN2 ungleich Null ist, um Berechnungsfehler oder Division-by-Zero-Ausnahmen zur Laufzeit zu vermeiden.

Zustandsübersicht

Der Baustein AI_DIV_2 ist ein zustandsloser (stateless) Kombinationsbaustein. Er besitzt kein internes Execution Control Chart (ECC). Jedes Ereignis an den Eingangs-Adaptern triggert direkt die Berechnung und die Aktualisierung des Ausgangs-Adapters.

Anwendungsszenarien

- **Verhältnisberechnungen:** Bestimmung von Verhältnissen in verfahrenstechnischen Anlagen (z. B. Luft-Brennstoff-Verhältnis oder Mischungsverhältnisse von Flüssigkeiten).
- **Skalierung und Normierung:** Teilen eines analogen Rohwertes (z. B. Sensor-Messwert) durch einen Skalierungsfaktor zur Umrechnung in physikalische Einheiten.
- **Messwert-Mittelung:** Einsatz in Berechnungsnetzwerken, bei denen Summenwerte durch eine feste oder variable Anzahl dividiert werden müssen.

Vergleich mit ähnlichen Bausteinen

- **Standard-Arithmetikbaustein (DIV):** Ein klassischer DIV-Baustein der IEC 61131-3 / IEC 61499 arbeitet mit direkten PIN-Eingängen (z. B. IN1, IN2 als REAL) und separaten Event-Pins (REQ, CNF). AI_DIV_2 hingegen bündelt diese Signale in Adaptern, was eine sauberere Schnittstellentrennung bei komplexen Signalwegen ermöglicht.
- **AI_SUB_2 / AI_ADD_2:** Diese Bausteine teilen dieselbe Adapter-Philosophie, führen jedoch Subtraktionen bzw. Additionen anstelle einer Division aus.

- AI_DIV_2_UNGATED: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

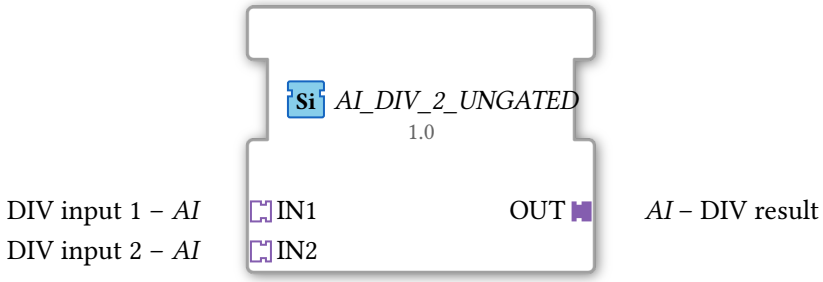
Fazit

Der AI_DIV_2 ist ein hocheffizienter und moderner Hilfsbaustein für die analoge Signalverarbeitung in 4diac. Durch die konsequente Nutzung von Adaptern fügt er sich nahtlos in serviceorientierte Steuerungsarchitekturen ein und minimiert den Implementierungs- und Testaufwand für mathematische Grundoperationen.

AI_DIV_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AI_DIV_2. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht

gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AI_DIV_2_UNGATED` ist ein generischer Arithmetik-Baustein für die 4diac-Plattform. Er dient zur Durchführung einer mathematischen Division (DIV) von zwei analogen Werten. Im Gegensatz zu klassischen Arithmetikbausteinen nutzt dieser Baustein standardisierte, unidirektionale Adapter vom Typ `AI` (Analog Input) für die Ein- und Ausgänge. Dies ermöglicht eine strukturierte, modulare und übersichtliche Signalverdrahtung innerhalb von IEC 61499 Anwendungen.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine direkten Ereignis-Eingänge. Die Synchronisation und Ereignissteuerung erfolgt implizit über die angebundenen Adapter.

Ereignis-Ausgänge

Der Baustein besitzt keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübertragung wird vollständig über die Adapter-Schnittstellen abgewickelt.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden.

Adapter

Name	Typ	Schnittstellen-Typ	Beschreibung
IN1	`adapter::types::unidirectional::AI`	Socket (Buchse)	Erster Eingangswert (Dividend)
IN2	`adapter::types::unidirectional::AI`	Socket (Buchse)	Zweiter Eingangswert (Divisor)
OUT	`adapter::types::unidirectional::AI`	Plug (Stecker)	Ergebniswert der Division (Quotient)

Funktionsweise

Der Funktionsbaustein berechnet den Quotienten aus den Werten der beiden Eingangs-Adapter:

$$OUT = \frac{IN1}{IN2} \tag{2.27}$$

Sobald sich die Werte an den Sockets `IN1` oder `IN2` ändern und ein entsprechendes Aktualisierungsereignis über den Adapter eingeht, wird die Division ausgeführt. Das Ergebnis sowie das dazugehörige Aktualisierungsereignis werden unmittelbar an den Ausgangs-Adapter `OUT` (Plug) weitergeleitet.

Aufgrund des generischen Typs (GEN_AI_DIV) ist der Baustein nicht auf einen festen Datentyp (wie z. B. REAL oder INT) beschränkt. Der konkrete Datentyp wird flexibel bei der Instanziierung in der 4diac-IDE bestimmt.

Technische Besonderheiten

- **Generische Typisierung:** Über das Attribut GenericClassName mit dem Wert GEN_AI_DIV ist der Baustein polymorph einsetzbar und kann mit verschiedenen kompatiblen analogen Datentypen arbeiten.
- **Adapter-Kapselung:** Durch die Verwendung von unidirektionalen Adaptern (unidirectional::AI) werden Daten- und Ereignisleitungen in einer einzigen Verbindung gebündelt. Dies reduziert das Risiko von Verdrahtungsfehlern und erhöht die Übersichtlichkeit im Funktionsplan (FBD).
- **Division durch Null:** Da es sich um eine mathematische Division handelt, muss anlagenseitig bzw. in der vorgeschalteten Logik sichergestellt werden, dass der Wert an IN2 ungleich Null ist, um Berechnungsfehler oder Division-by-Zero-Ausnahmen zur Laufzeit zu vermeiden.

Zustandsübersicht

Der Baustein AI_DIV_2_UNGATED ist ein zustandsloser (stateless) Kombinationsbaustein. Er besitzt kein internes Execution Control Chart (ECC). Jedes Ereignis an den Eingangs-Adaptern triggert direkt die Berechnung und die Aktualisierung des Ausgangs-Adapters.

Anwendungsszenarien

- **Verhältnisberechnungen:** Bestimmung von Verhältnissen in verfahrenstechnischen Anlagen (z. B. Luft-Brennstoff-Verhältnis oder Mischungsverhältnisse von Flüssigkeiten).
- **Skalierung und Normierung:** Teilen eines analogen Rohwertes (z. B. Sensor-Messwert) durch einen Skalierungsfaktor zur Umrechnung in physikalische Einheiten.
- **Messwert-Mittelung:** Einsatz in Berechnungsnetzwerken, bei denen Summenwerte durch eine feste oder variable Anzahl dividiert werden müssen.

Vergleich mit ähnlichen Bausteinen

- **Standard-Arithmetikbaustein (DIV):** Ein klassischer DIV-Baustein der IEC 61131-3 / IEC 61499 arbeitet mit direkten PIN-Eingängen (z. B. IN1, IN2 als REAL) und separaten Event-Pins (REQ, CNF). AI_DIV_2_UNGATED hingegen bündelt diese Signale in Adaptern, was eine sauberere Schnittstellentrennung bei komplexen Signalwegen ermöglicht.
- **AI_SUB_2 / AI_ADD_2:** Diese Bausteine teilen dieselbe Adapter-Philosophie, führen jedoch Subtraktionen bzw. Additionen anstelle einer Division aus.

- AI_DIV_2: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

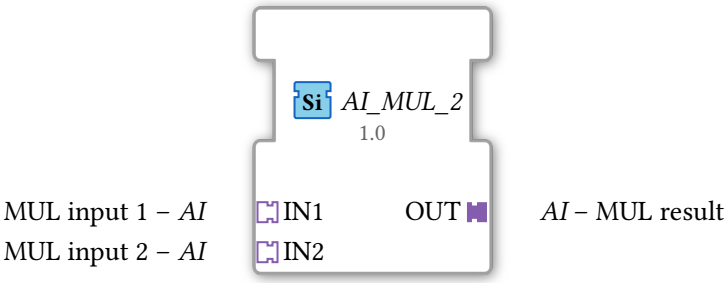
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der AI_DIV_2_UNGATED ist ein hocheffizienter und moderner Hilfsbaustein für die analoge Signalverarbeitung in 4diac. Durch die konsequente Nutzung von Adaptern fügt er sich nahtlos in serviceorientierte Steuerungsarchitekturen ein und minimiert den Implementierungs- und Testaufwand für mathematische Grundoperationen.

AI_MUL_2



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AI_MUL_2` ist ein generischer arithmetischer Funktionsbaustein für die 4diac-IDE, der der Norm IEC 61131-3 entspricht. Seine Hauptaufgabe ist die Multiplikation von zwei Eingangswerten, die über unidirektionale Adapter bereitgestellt werden. Das Ergebnis dieser Multiplikation wird ebenfalls über einen unidirektionalen Adapter ausgegeben.

Durch die Verwendung von Adaptern anstelle von klassischen diskreten Ein- und Ausgängen wird die Komplexität der Verkabelung im Steuerungsentwurf erheblich reduziert, da zusammengehörige Daten und Ereignisse in einer einzigen Verbindung gebündelt werden.

Schnittstellenstruktur

Die Schnittstelle dieses Funktionsbausteins basiert vollständig auf Adaptern. Es gibt keine direkten, klassischen Ereignis- oder Datenein- und -ausgänge auf der Bausteinoberfläche.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden (die Steuerung erfolgt über die Adapter).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden (die Steuerung erfolgt über die Adapter).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Name	Typ	Richtung (Modus)	Beschreibung
IN1	`adapter::types::unidirectional::AI`	Socket (Eingang)	Erster Eingangswert (Multiplikand) für die arithmetische Operation.
IN2	`adapter::types::unidirectional::AI`	Socket (Eingang)	Zweiter Eingangswert (Multiplikator) für die arithmetische Operation.
OUT	`adapter::types::unidirectional::AI`	Plug (Ausgang)	Ausgang für das berechnete Produkt der beiden Eingangswerte.

Funktionsweise

Der Funktionsbaustein realisiert die mathematische Multiplikation:

$$\text{OUT} = \text{IN1} \times \text{IN2} \tag{2.28}$$

Da der Baustein als generischer Funktionsbaustein (GEN_AI_MUL) definiert ist, kann er je nach Instanzierung und Typisierung der verwendeten Adapter unterschiedliche numerische Datentypen (z. B. REAL, LREAL, INT etc.) verarbeiten.

Sobald über die Eingangs-Adapter (IN1 und/oder IN2) ein Aktualisierungsereignis empfangen wird, führt der Baustein intern die Multiplikation aus und signalisiert die Aktualisierung des Ergebnisses über den Ausgangs-Adapter OUT.

Technische Besonderheiten

- **Generische Implementierung:** Der Baustein nutzt die Klasse GEN_AI_MUL. Dies ermöglicht eine hohe Flexibilität, da der konkrete Datentyp erst bei der Verwendung im System bestimmt wird.
- **Unidirektionale Adapter:** Die Schnittstellen nutzen den Typ adapter::types::unidirectional::AI. Das bedeutet, dass der Informationsfluss strikt in eine Richtung verläuft, was die Systemstabilität und die Performance erhöht.
- **Kapselung:** Durch das Fehlen einzelner Signal-Pins bleibt das Anwendungsdiagramm auch bei vielen mathematischen Operationen übersichtlich und aufgeräumt.

Zustandsübersicht

Der Baustein verhält sich im Wesentlichen zustandslos (analogen Charakter aufweisend):

- **Initialisierung / Ruhezustand:** Der Baustein wartet auf eingehende Werte über die Adapter IN1 und IN2.
- **Berechnung:** Bei Eintreffen eines neuen Wertes oder Triggersignals an den Sockets wird das Produkt neu berechnet.
- **Ausgabe:** Das Ergebnis wird unmittelbar an den Plug OUT übergeben und ein entsprechendes Ausgangsereignis im Adapter getriggert.

Anwendungsszenarien

- **Skalierung von Sensorwerten:** Multiplikation eines analogen Rohwertes (z. B. von einem Stromeingang 4–20 mA) mit einem Skalierungsfaktor zur Umrechnung in eine physikalische Größe.
- **Berechnung physikalischer Größen:** Berechnung von Leistung ($P = U \times I$) aus gemessener Spannung und Stromstärke, sofern diese über entsprechende Adapterstrukturen geliefert werden.
- **Verstärkungsglieder in Regelkreisen:** Einsatz als proportionaler Verstärkungsfaktor (P-Glied) in einer softwarebasierten Regelung.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zum Standard-IEC-61131-3-Baustein MUL entfällt beim AI_MUL_2 die manuelle Verdrahtung von Trigger-Ereignissen (wie REQ und CNF) sowie der einzelnen Daten-Pins. Während ein klassischer MUL-Baustein für jede Verbindung separate Linien für Daten und Events benötigt, bündelt der AI_MUL_2 diese über die AI-Adapter. Dies eignet sich besonders für fortgeschrittene, objektorientierte oder modularisierte Softwarearchitekturen in 4diac.

- AI_MUL_2_UNGATED: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

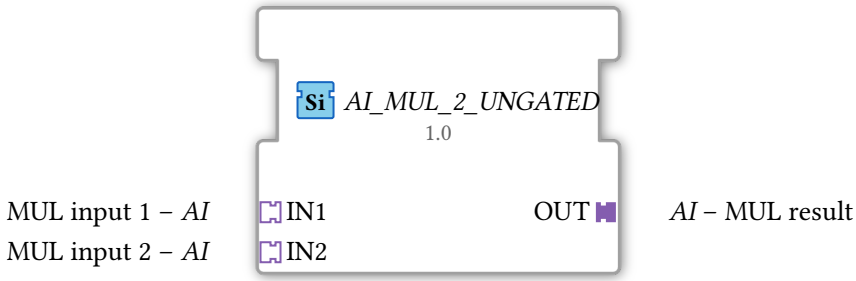
Fazit

Der AI_MUL_2 ist ein spezialisierter, aber durch seine generische Natur dennoch flexibler Baustein zur Multiplikation zweier Werte über Adapterverbindungen. Er eignet sich hervorragend für saubere, übersichtliche Steuerungsarchitekturen, bei denen analoge Signale standardisiert über unidirektionale Adapter übertragen und verarbeitet werden sollen.

AI_MUL_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AI_MUL_2. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weiter-

gegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AI_MUL_2_UNGATED` ist ein generischer arithmetischer Funktionsbaustein für die 4diac-IDE, der der Norm IEC 61131-3 entspricht. Seine Hauptaufgabe ist die Multiplikation von zwei Eingangswerten, die über unidirektionale Adapter bereitgestellt werden. Das Ergebnis dieser Multiplikation wird ebenfalls über einen unidirektionalen Adapter ausgegeben.

Durch die Verwendung von Adaptern anstelle von klassischen diskreten Ein- und Ausgängen wird die Komplexität der Verkabelung im Steuerungsentwurf erheblich reduziert, da zusammengehörige Daten und Ereignisse in einer einzigen Verbindung gebündelt werden.

Schnittstellenstruktur

Die Schnittstelle dieses Funktionsbausteins basiert vollständig auf Adaptern. Es gibt keine direkten, klassischen Ereignis- oder Datenein- und -ausgänge auf der Bausteinoberfläche.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden (die Steuerung erfolgt über die Adapter).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden (die Steuerung erfolgt über die Adapter).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Name	Typ	Richtung (Modus)	Beschreibung
IN1	`adapter::types::unidirectional::AI`	Socket (Eingang)	Erster Eingangswert (Multiplikand) für die arithmetische Operation.
IN2	`adapter::types::unidirectional::AI`	Socket (Eingang)	Zweiter Eingangswert (Multiplikator) für die arithmetische Operation.
OUT	`adapter::types::unidirectional::AI`	Plug (Ausgang)	Ausgang für das berechnete Produkt der beiden Eingangswerte.

Funktionsweise

Der Funktionsbaustein realisiert die mathematische Multiplikation:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \tag{2.29}$$

Da der Baustein als generischer Funktionsbaustein (`GEN_AI_MUL`) definiert ist, kann er je nach Instanzierung und Typisierung der verwendeten Adapter unterschiedliche numerische Datentypen (z. B. `REAL`, `LREAL`, `INT` etc.) verarbeiten.

Sobald über die Eingangs-Adapter (`IN1` und/oder `IN2`) ein Aktualisierungsereignis empfangen wird, führt der Baustein intern die Multiplikation aus und signalisiert die Aktualisierung des Ergebnisses über den Ausgangs-Adapter `OUT`.

Technische Besonderheiten

- **Generische Implementierung:** Der Baustein nutzt die Klasse `GEN_AI_MUL`. Dies ermöglicht eine hohe Flexibilität, da der konkrete Datentyp erst bei der Verwendung im System bestimmt wird.
- **Unidirektionale Adapter:** Die Schnittstellen nutzen den Typ `adapter::types::unidirectional::AI`. Das bedeutet, dass der Informationsfluss strikt in eine Richtung verläuft, was die Systemstabilität und die Performance erhöht.
- **Kapselung:** Durch das Fehlen einzelner Signal-Pins bleibt das Anwendungsdiagramm auch bei vielen mathematischen Operationen übersichtlich und aufgeräumt.

Zustandsübersicht

Der Baustein verhält sich im Wesentlichen zustandslos (analogen Charakter aufweisend):

- **Initialisierung / Ruhezustand:** Der Baustein wartet auf eingehende Werte über die Adapter `IN1` und `IN2`.
- **Berechnung:** Bei Eintreffen eines neuen Wertes oder Triggersignals an den Sockets wird das Produkt neu berechnet.
- **Ausgabe:** Das Ergebnis wird unmittelbar an den Plug `OUT` übergeben und ein entsprechendes Ausgangsereignis im Adapter getriggert.

Anwendungsszenarien

- **Skalierung von Sensorwerten:** Multiplikation eines analogen Rohwertes (z. B. von einem Stromeingang 4–20 mA) mit einem Skalierungsfaktor zur Umrechnung in eine physikalische Größe.
- **Berechnung physikalischer Größen:** Berechnung von Leistung ($P = U \times I$) aus gemessener Spannung und Stromstärke, sofern diese über entsprechende Adapterstrukturen geliefert werden.
- **Verstärkungsglieder in Regelkreisen:** Einsatz als proportionaler Verstärkungsfaktor (P-Glied) in einer softwarebasierten Regelung.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zum Standard-IEC-61131-3-Baustein `MUL` entfällt beim `AI_MUL_2_UNGATED` die manuelle Verdrahtung von Trigger-Ereignissen (wie `REQ` und `CNF`) sowie der einzelnen Daten-Pins. Während ein klassischer `MUL`-Baustein für jede Verbindung separate Linien für Daten und Events benötigt, bündelt der `AI_MUL_2_UNGATED` diese über die `AI`-Adapter. Dies eignet sich besonders für fortgeschrittene, objektorientierte oder modularisierte Softwarearchitekturen in 4diac.

- `AI_MUL_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- `AI_MUL_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

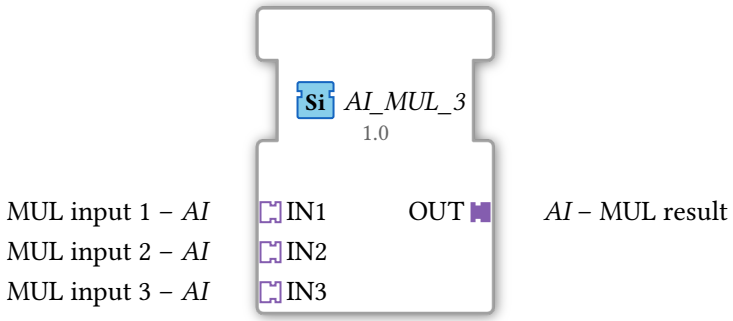
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AI_MUL_2_UNGATED` ist ein spezialisierter, aber durch seine generische Natur dennoch flexibler Baustein zur Multiplikation zweier Werte über Adapterverbindungen. Er eignet sich hervorragend für saubere, übersichtliche Steuerungsarchitekturen, bei denen analoge Signale standardisiert über unidirektionale Adapter übertragen und verarbeitet werden sollen.

AI_MUL_3



(Kein Bild vorhanden)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AI_MUL_3` ist ein generischer arithmetischer Baustein, der für die Multiplikation von drei Eingangswerten entwickelt wurde. Er entspricht der Klassifizierung für Standard-Arithmetikfunktionen nach IEC 61131-3 und nutzt das Adapter-Konzept von 4diac. Durch die Kapselung der Signale in Adaptern wird die visuelle Komplexität im Funktionsplan erheblich reduziert.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Eingänge. Die Ereignissteuerung wird intern über die Adapter abgewickelt.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Ausgänge. Die Ereignissteuerung wird intern über die Adapter abgewickelt.

Daten-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Eingänge. Die Datenübergabe erfolgt über die Eingangs-Adapter.

Daten-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Ausgänge. Die Datenausgabe erfolgt über den Ausgangs-Adapter.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AI`): Der erste Multiplikand (Eingangswert 1).
- **IN2** (Typ: `adapter::types::unidirectional::AI`): Der zweite Multiplikand (Eingangswert 2).
- **IN3** (Typ: `adapter::types::unidirectional::AI`): Der dritte Multiplikand (Eingangswert 3).

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AI`): Das Ergebnis der Multiplikation.

Funktionsweise

Der Funktionsbaustein führt eine kontinuierliche oder ereignisgesteuerte arithmetische Multiplikation der drei über die Sockets anliegenden Werte aus.

Die mathematische Formel lautet:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \tag{2.30}$$

Sobald sich die Werte an den Eingangs-Adaptern ändern oder ein entsprechendes Aktualisierungsereignis über die Adapter getriggert wird, berechnet der Baustein das Produkt und stellt dieses am Ausgangs-Adapter `OUT` zur Verfügung.

Technische Besonderheiten

- **Generischer Charakter:** Der Baustein ist als generischer Typ deklariert (`GenericClassName = 'GEN_AI_MUL'`). Das bedeutet, dass er flexibel mit verschiedenen numerischen Datentypen (wie z. B. REAL, LREAL, INT) arbeiten kann, sofern diese vom Adaptertyp `AI` unterstützt werden.
- **Adapter-Kopplung:** Durch die Verwendung von unidirektionalen Adaptern (`unidirectional::AI`) werden Daten und deren Gültigkeitsereignisse zusammengefasst. Dies vereinfacht die Verdrahtung in der 4diac-IDE drastisch, da keine separaten Event- und Datenleitungen gezogen werden müssen.

Zustandsübersicht

Der Baustein besitzt kein komplexes internes Zustandsdiagramm (zustandslos). Er reagiert rein reaktiv auf Wertänderungen oder Trigger-Ereignisse an den Eingängen `IN1`, `IN2` und `IN3` und gibt das Ergebnis unmittelbar an `OUT` weiter.

Anwendungsszenarien

- **Messwertskalierung:** Multiplikation eines analogen Rohwertes (`IN1`) mit einem Kalibrierungsfaktor (`IN2`) und einem weiteren Korrekturfaktor (`IN3`).
- **Physikalische Berechnungen:** Berechnung von Volumina ($V = l \times b \times h$) oder Leistungen, bei denen drei Faktoren direkt miteinander multipliziert werden müssen.
- **Kaskadierte Verstärkungen:** Signalverarbeitungsketten, bei denen ein Signal nacheinander zwei Verstärkungsstufen durchläuft.

Vergleich mit ähnlichen Bausteinen

- **Standard-MUL-Baustein (IEC 61131-3):** Klassische Multiplikationsbausteine besitzen direkte Datenpins (z. B. `ANY_NUM`) und benötigen explizite Event-Verbindungen (`REQ` / `CNF`). `AI_MUL_3` bündelt diese Logik in Adaptern.
- **AI_MUL_2 (Zweifach-Multiplizierer):** Während bei der Multiplikation von drei Werten mit einem Standard-Zweifach-Multiplizierer zwei Bausteine hintereinandergeschaltet werden müssen, erledigt `AI_MUL_3` dies in einem einzigen Schritt, was Ressourcen und Platz im Steuerungsdiagramm spart.

- `AI_MUL_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

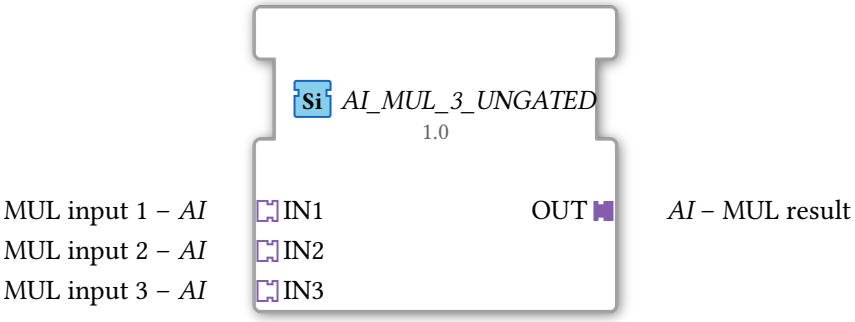
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `AI_MUL_3` Funktionsbaustein bietet eine effiziente, saubere und performante Möglichkeit, dreifache Multiplikationen innerhalb einer 4diac-Anwendung zu realisieren. Durch die konsequente Nutzung des Adapter-Konzepts trägt er maßgeblich zur Übersichtlichkeit und Wartbarkeit von Steuerungssoftware bei.

AI_MUL_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AI_MUL_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild vorhanden)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AI_MUL_3_UNGATED` ist ein generischer arithmetischer Baustein, der für die Multiplikation von drei Eingangswerten entwickelt wurde. Er entspricht der Klassifizierung für Standard-Arithmetikfunktionen nach IEC 61131-3 und nutzt das Adapter-Konzept von 4diac. Durch die Kapselung der Signale in Adaptern wird die visuelle Komplexität im Funktionsplan erheblich reduziert.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Eingänge. Die Ereignissteuerung wird intern über die Adapter abgewickelt.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Ausgänge. Die Ereignissteuerung wird intern über die Adapter abgewickelt.

Daten-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Eingänge. Die Datenübergabe erfolgt über die Eingangs-Adapter.

Daten-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Ausgänge. Die Datenausgabe erfolgt über den Ausgangs-Adapter.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AI`): Der erste Multiplikand (Eingangswert 1).
- **IN2** (Typ: `adapter::types::unidirectional::AI`): Der zweite Multiplikand (Eingangswert 2).
- **IN3** (Typ: `adapter::types::unidirectional::AI`): Der dritte Multiplikand (Eingangswert 3).

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AI`): Das Ergebnis der Multiplikation.

Funktionsweise

Der Funktionsbaustein führt eine kontinuierliche oder ereignisgesteuerte arithmetische Multiplikation der drei über die Sockets anliegenden Werte aus.

Die mathematische Formel lautet:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \tag{2.31}$$

Sobald sich die Werte an den Eingangs-Adaptern ändern oder ein entsprechendes Aktualisierungsereignis über die Adapter getriggert wird, berechnet der Baustein das Produkt und stellt dieses am Ausgangs-Adapter `OUT` zur Verfügung.

Technische Besonderheiten

- **Generischer Charakter:** Der Baustein ist als generischer Typ deklariert (`GenericClassName = 'GEN_AI_MUL'`). Das bedeutet, dass er flexibel mit verschiedenen numerischen Datentypen (wie z. B. REAL, LREAL, INT) arbeiten kann, sofern diese vom Adaptertyp `AI` unterstützt werden.
- **Adapter-Kopplung:** Durch die Verwendung von unidirektionalen Adaptern (`unidirectional::AI`) werden Daten und deren Gültigkeitsereignisse zusammengefasst. Dies vereinfacht die Verdrahtung in der 4diac-IDE drastisch, da keine separaten Event- und Datenleitungen gezogen werden müssen.

Zustandsübersicht

Der Baustein besitzt kein komplexes internes Zustandsdiagramm (zustandslos). Er reagiert rein reaktiv auf Wertänderungen oder Trigger-Ereignisse an den Eingängen `IN1`, `IN2` und `IN3` und gibt das Ergebnis unmittelbar an `OUT` weiter.

Anwendungsszenarien

- **Messwertskalierung:** Multiplikation eines analogen Rohwertes (`IN1`) mit einem Kalibrierungsfaktor (`IN2`) und einem weiteren Korrekturfaktor (`IN3`).
- **Physikalische Berechnungen:** Berechnung von Volumina ($V = l \times b \times h$) oder Leistungen, bei denen drei Faktoren direkt miteinander multipliziert werden müssen.
- **Kaskadierte Verstärkungen:** Signalverarbeitungsketten, bei denen ein Signal nacheinander zwei Verstärkungsstufen durchläuft.

Vergleich mit ähnlichen Bausteinen

- **Standard-MUL-Baustein (IEC 61131-3):** Klassische Multiplikationsbausteine besitzen direkte Datenpins (z. B. `ANY_NUM`) und benötigen explizite Event-Verbindungen (`REQ` / `CNF`). `AI_MUL_3_UNGATED` bündelt diese Logik in Adaptern.
- **AI_MUL_2 (Zweifach-Multiplizierer):** Während bei der Multiplikation von drei Werten mit einem Standard-Zweifach-Multiplizierer zwei Bausteine hintereinandergeschaltet werden müssen, erledigt `AI_MUL_3_UNGATED` dies in einem einzigen Schritt, was Ressourcen und Platz im Steuerungsdiagramm spart.

- `AI_MUL_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

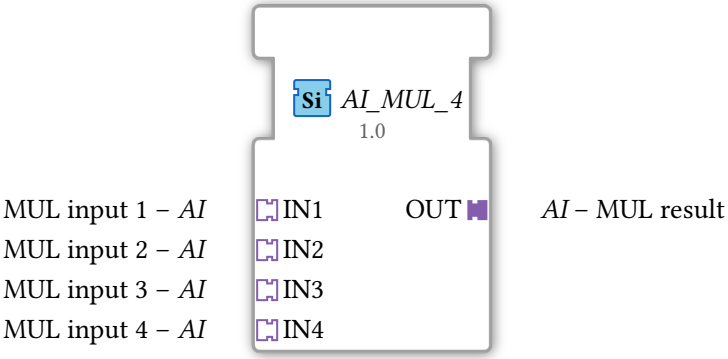
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AI_MUL_3_UNGATED` Funktionsbaustein bietet eine effiziente, saubere und performante Möglichkeit, dreifache Multiplikationen innerhalb einer 4diac-Anwendung zu realisieren. Durch die konsequente Nutzung des Adapter-Konzepts trägt er maßgeblich zur Übersichtlichkeit und Wartbarkeit von Steuerungssoftware bei.

AI_MUL_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AI_MUL_4` ist ein generischer arithmetischer Baustein für 4diac-IDE, der der Norm IEC 61131-3 entspricht. Seine Hauptaufgabe besteht darin, das mathematische Produkt aus vier analogen Eingangswerten zu berechnen. Der Baustein nutzt unidirektionale Adapter vom Typ `AI` (Analog Input), um sowohl Daten als auch die dazugehörigen Ereignisse gekapselt zu übertragen. Dies reduziert den Verdrahtungsaufwand im Funktionsplan erheblich und sorgt für eine sauberere Strukturierung der Steuerungslogik.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsbaustein besitzt keine direkten, eigenständigen Ereignis-Eingänge. Die Ereignissteuerung wird vollständig über die angeschlossenen Adapter abgewickelt.

Ereignis-Ausgänge

Der Funktionsbaustein besitzt keine direkten, eigenständigen Ereignis-Ausgänge. Die Triggerung nachfolgender Bausteine erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Wertebereitstellung erfolgt über die Adapter-Eingänge.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Die Ausgabe des berechneten Werts erfolgt über den Adapter-Ausgang.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AI`): Erster Multiplikand.
- **IN2** (Typ: `adapter::types::unidirectional::AI`): Zweiter Multiplikand.
- **IN3** (Typ: `adapter::types::unidirectional::AI`): Dritter Multiplikand.
- **IN4** (Typ: `adapter::types::unidirectional::AI`): Vierter Multiplikand.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AI`): Produkt der Multiplikation aller vier Eingänge ($OUT = IN1 \cdot IN2 \cdot IN3 \cdot IN4$).

Funktionsweise

Sobald an einem der Eingangs-Adapter (`IN1` bis `IN4`) ein neues Ereignis signalisiert, dass sich die Daten aktualisiert haben, liest der Funktionsbaustein die analogen Werte der vier Adapter aus.

Die Berechnung erfolgt nach der Formel:

$$\text{Ergebnis} = \text{Wert } (IN1) \cdot \text{Wert } (IN2) \cdot \text{Wert } (IN3) \cdot \text{Wert } (IN4) \tag{2.32}$$

Das Ergebnis dieser Berechnung wird an den Ausgangs-Adapter `OUT` übergeben, und das entsprechende Aktualisierungsereignis des Adapters wird ausgelöst, um nachfolgende Bausteine im Kontrollfluss über den neuen Wert zu informieren.

Technische Besonderheiten

- **Generische Klasse:** Der Baustein basiert intern auf der generischen Klasse `GEN_AI_MUL`. Dies ermöglicht eine flexible Verarbeitung der Datenströme unabhängig von spezifischen Hardware-Implementierungen.
- **Unidirektionale Adapter:** Durch die Verwendung des Typs `adapter::types::unidirectional::AI` fließen Daten und Trigger-Signale in nur eine Richtung (vom Sender zum Empfänger). Das vereinfacht das Signal-Design und vermeidet Rückkopplungsschleifen.
- **Kompakte Struktur:** Durch die Verarbeitung von vier Eingängen in einem einzigen Baustein wird das Kaskadieren mehrerer Multiplikationsbausteine überflüssig.

Zustandsübersicht

Da es sich bei AI_MUL_4 um einen rein mathematischen, daten- und ereignisgesteuerten Funktionsbaustein handelt, besitzt er keinen internen Zustandsautomaten (ECC). Die Ausführung verhält sich rein zustandslos:

1. **Warten:** Der Baustein wartet auf ein Aktualisierungsereignis an einem der Sockets (IN1 bis IN4).
2. **Berechnen:** Bei Triggerung werden die aktuellen Werte aller vier Eingänge multipliziert.
3. **Ausgeben:** Das Ergebnis wird an den Plug OUT angelegt und das Ausgabeereignis getriggert. Der Baustein kehrt sofort in den Wartezustand zurück.

Anwendungsszenarien

- **Skalierung und Kalibrierung:** Berechnung von zusammengesetzten Korrekturfaktoren für analoge Sensorwerte (z. B. Sensorwert · Kalibrierungsfaktor · Temperaturkompensation · Einheitenumrechnung).
- **Physikalische Berechnungen:** Berechnung von Werten, die von mehreren analogen Faktoren abhängen, wie beispielsweise Volumenströme oder elektrische Leistungen unter Einbeziehung verschiedener Wirkungsgrade.
- **Kaskadierte Verstärkungsglieder:** Einsatz in Regelungskreisen, bei denen mehrere Verstärkungsfaktoren (Gains) nacheinander auf ein analoges Signal angewendet werden müssen.

Vergleich mit ähnlichen Bausteinen

- **Standard-MUL (IEC 61131-3):** Klassische Multiplikationsbausteine arbeiten meist nur mit elementaren Datentypen (z. B. REAL, INT) und benötigen separate Event-Leitungen (REQ / CNF). Zudem unterstützen sie oft standardmäßig nur zwei Eingänge. AI_MUL_4 hingegen verarbeitet vier Eingänge direkt und nutzt Adapter zur Kapselung, was das Netzwerklayout übersichtlicher macht.
- **AI_MUL_2:** Ein ähnlicher adapterbasierter Baustein, jedoch nur für zwei Eingänge. AI_MUL_4 spart bei komplexeren Berechnungen mit bis zu vier Faktoren zusätzlichen Verdrahtungsaufwand und Baustein-Instanzen.
- AI_MUL_4_UNGATED: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

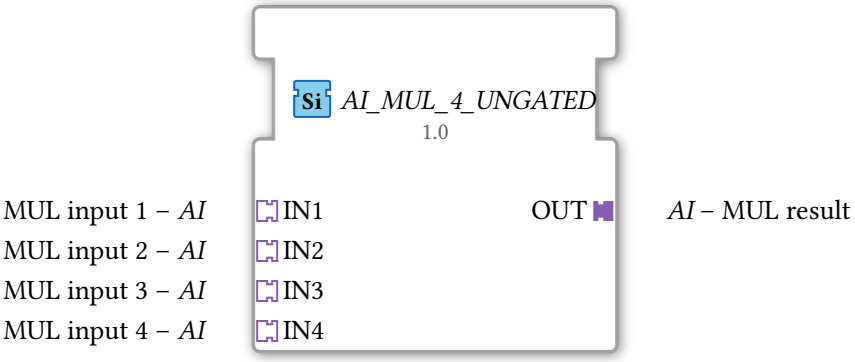
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der AI_MUL_4-Funktionsbaustein bietet eine effiziente und saubere Möglichkeit, komplexe Multiplikationsaufgaben mit bis zu vier analogen Signalen in 4diac-IDE zu realisieren. Durch die konsequente Nutzung von unidirektionalen Adaptern wird der Verkabelungsaufwand minimiert und die Lesbarkeit des Applikationsdesigns signifikant verbessert.

AI_MUL_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AI_MUL_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AI_MUL_4_UNGATED` ist ein generischer arithmetischer Baustein für 4diac-IDE, der der Norm IEC 61131-3 entspricht. Seine Hauptaufgabe besteht darin, das mathematische Produkt aus vier analogen Eingangswerten zu berechnen. Der Baustein nutzt unidirektionale Adapter vom Typ `AI` (Analog Input), um sowohl Daten als auch die dazugehörigen Ereignisse gekapselt zu übertragen. Dies reduziert den Verdrahtungsaufwand im Funktionsplan erheblich und sorgt für eine sauberere Strukturierung der Steuerungslogik.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsbaustein besitzt keine direkten, eigenständigen Ereignis-Eingänge. Die Ereignissteuerung wird vollständig über die angeschlossenen Adapter abgewickelt.

Ereignis-Ausgänge

Der Funktionsbaustein besitzt keine direkten, eigenständigen Ereignis-Ausgänge. Die Triggerung nachfolgender Bausteine erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Wertebereitstellung erfolgt über die Adapter-Eingänge.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Die Ausgabe des berechneten Werts erfolgt über den Adapter-Ausgang.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AI`): Erster Multiplikand.
- **IN2** (Typ: `adapter::types::unidirectional::AI`): Zweiter Multiplikand.
- **IN3** (Typ: `adapter::types::unidirectional::AI`): Dritter Multiplikand.
- **IN4** (Typ: `adapter::types::unidirectional::AI`): Vierter Multiplikand.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AI`): Produkt der Multiplikation aller vier Eingänge ($OUT = IN1 \cdot IN2 \cdot IN3 \cdot IN4$).

Funktionsweise

Sobald an einem der Eingangs-Adapter (`IN1` bis `IN4`) ein neues Ereignis signalisiert, dass sich die Daten aktualisiert haben, liest der Funktionsbaustein die analogen Werte der vier Adapter aus.

Die Berechnung erfolgt nach der Formel:

$$\text{Ergebnis} = \text{Wert}\ (IN1) \cdot \text{Wert}\ (IN2) \cdot \text{Wert}\ (IN3) \cdot \text{Wert}\ (IN4) \tag{2.33}$$

Das Ergebnis dieser Berechnung wird an den Ausgangs-Adapter `OUT` übergeben, und das entsprechende Aktualisierungsereignis des Adapters wird ausgelöst, um nachfolgende Bausteine im Kontrollfluss über den neuen Wert zu informieren.

Technische Besonderheiten

- **Generische Klasse:** Der Baustein basiert intern auf der generischen Klasse `GEN_AI_MUL`. Dies ermöglicht eine flexible Verarbeitung der Datenströme unabhängig von spezifischen Hardware-Implementierungen.
- **Unidirektionale Adapter:** Durch die Verwendung des Typs `adapter::types::unidirectional::AI` fließen Daten und Trigger-Signale in nur eine Richtung (vom Sender zum Empfänger). Das vereinfacht das Signal-Design und vermeidet Rückkopplungsschleifen.
- **Kompakte Struktur:** Durch die Verarbeitung von vier Eingängen in einem einzigen Baustein wird das Kaskadieren mehrerer Multiplikationsbausteine überflüssig.

Zustandsübersicht

Da es sich bei `AI_MUL_4_UNGATED` um einen rein mathematischen, daten- und ereignisgesteuerten Funktionsbaustein handelt, besitzt er keinen internen Zustandsautomaten (ECC). Die Ausführung verhält sich rein zustandslos:

1. **Warten:** Der Baustein wartet auf ein Aktualisierungsereignis an einem der Sockets (`IN1` bis `IN4`).
2. **Berechnen:** Bei Triggerung werden die aktuellen Werte aller vier Eingänge multipliziert.
3. **Ausgeben:** Das Ergebnis wird an den Plug `OUT` angelegt und das Ausgabeereignis getriggert. Der Baustein kehrt sofort in den Wartezustand zurück.

Anwendungsszenarien

- **Skalierung und Kalibrierung:** Berechnung von zusammengesetzten Korrekturfaktoren für analoge Sensorwerte (z. B. Sensorwert · Kalibrierungsfaktor · Temperaturkompensation · Einheitenumrechnung).
- **Physikalische Berechnungen:** Berechnung von Werten, die von mehreren analogen Faktoren abhängen, wie beispielsweise Volumenströme oder elektrische Leistungen unter Einbeziehung verschiedener Wirkungsgrade.
- **Kaskadierte Verstärkungsglieder:** Einsatz in Regelungskreisen, bei denen mehrere Verstärkungsfaktoren (Gains) nacheinander auf ein analoges Signal angewendet werden müssen.

Vergleich mit ähnlichen Bausteinen

- **Standard-MUL (IEC 61131-3):** Klassische Multiplikationsbausteine arbeiten meist nur mit elementaren Datentypen (z. B. `REAL`, `INT`) und benötigen separate Event-Leitungen (`REQ` / `CNF`). Zudem unterstützen sie oft standardmäßig nur zwei Eingänge. `AI_MUL_4_UNGATED` hingegen verarbeitet vier Eingänge direkt und nutzt Adapter zur Kapselung, was das Netzwerklayout übersichtlicher macht.
- **AI_MUL_2:** Ein ähnlicher adapterbasierter Baustein, jedoch nur für zwei Eingänge. `AI_MUL_4_UNGATED` spart bei komplexeren Berechnungen mit bis zu vier Faktoren zusätzlichen Verdrahtungsaufwand und Baustein-Instanzen.
- `AI_MUL_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- `AI_MUL_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

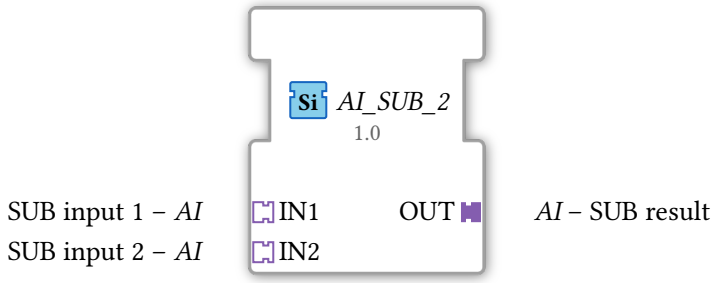
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AI_MUL_4_UNGATED`-Funktionsbaustein bietet eine effiziente und saubere Möglichkeit, komplexe Multiplikationsaufgaben mit bis zu vier analogen Signalen in 4diac-IDE zu realisieren. Durch die konsequente Nutzung von unidirektionalen Adaptern wird der Verkabelungsaufwand minimiert und die Lesbarkeit des Applikationsdesigns signifikant verbessert.

AI_SUB_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AI_SUB_2` ist ein generischer, arithmetischer Funktionsbaustein zur Durchführung einer Subtraktion. Er ist speziell für den Einsatz mit analogen Schnittstellen-Adaptern konzipiert. Der Baustein berechnet die Differenz zwischen zwei analogen Eingangswerten und stellt das Ergebnis an einem analogen Ausgangs-Adapter zur Verfügung. Durch die Verwendung von Adaptern wird das Schnittstellendesign in der 4diac-ide übersichtlich gehalten.

Schnittstellenstruktur

Da dieser Funktionsbaustein vollständig auf einer Adapter-Architektur basiert, besitzt er keine direkten, klassischen Ereignis- oder Datenkanäle auf der Bausteinoberfläche. Die gesamte Kommunikation wird über die Adapter abgewickelt.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden (die Ereignissteuerung ist in den Adaptern gekapselt).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden (die Ereignissteuerung ist in den Adaptern gekapselt).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AI`): Der erste analoge Eingangswert (Minuend) für die Subtraktionsoperation.
- **IN2** (Typ: `adapter::types::unidirectional::AI`): Der zweite analoge Eingangswert (Subtrahend), der von `IN1` abgezogen wird.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AI`): Das Ergebnis der Subtraktion (Differenz: `IN1 - IN2`).

Funktionsweise

Der Baustein führt die grundlegende arithmetische Subtraktion aus:

$$\mathrm{OUT} = \mathrm{IN1} - \mathrm{IN2} \tag{2.34}$$

Sobald sich die Eingangswerte an den Sockets `IN1` oder `IN2` ändern bzw. ein entsprechendes Aktualisierungsereignis über die Adapter empfangen wird, berechnet der Baustein intern die Differenz. Das Ergebnis sowie das zugehörige Aktualisierungsereignis werden unmittelbar an den Ausgangs-Plug `OUT` weitergeleitet.

Technische Besonderheiten

- **Generischer Bausteintyp (`GEN_AI_SUB`)**: Der Baustein ist intern als generischer Typ definiert. Dies ermöglicht eine flexible Anpassung an die im System verwendeten analogen Datentypen der Adapter.
- **Adapter-Kapselung**: Die Verwendung von unidirektionalen analogen Adaptern (`unidirectional::AI`) bündelt Daten- und Eventleitungen. Dies minimiert den Verdrahtungsaufwand im FBD (Function Block Diagram) und erhöht die Übersichtlichkeit signifikant.

Zustandsübersicht

Der Baustein verhält sich wie ein rein funktionaler Baustein ohne internen Speicher (analog zu einem klassisch kombinatorischen Gatter). Er besitzt keine komplexe Execution Control Chart (ECC). Jede Aktivität an den Eingangs-Adaptern triggert direkt die Berechnung und aktualisiert den Ausgangs-Adapter.

Anwendungsszenarien

- **Differenzdruck- und Differenztemperaturmessung**: Berechnung von physikalischen Differenzwerten in der Prozesstechnik (z. B. Ermittlung des Druckabfalls über einem Filter oder der Temperaturdifferenz zwischen Vor- und Rücklauf).
- **Regelungsabweichung**: Berechnung der Regelabweichung ($e = w - x$) in Regelungskreisen, bei denen Soll- und Istwert als Adapterstrukturen vorliegen.
- **Nullpunktkorrektur / Offset-Abzug**: Subtraktion eines Kalibrier- oder Offset-Wertes von einem kontinuierlichen Messsignal.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-Subtraktionsbausteinen der IEC 61131-3 (wie z. B. einem klassischen `SUB`-Baustein mit diskreten Pins) entfällt beim `AI_SUB_2` das manuelle Verdrahten von separaten `REQ`- und `CNF`-Eventleitungen sowie einzelnen analogen Datenleitungen. Alle relevanten Informationen werden sauber in den `AI`-Adaptern übertragen.

- `AI_SUB_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

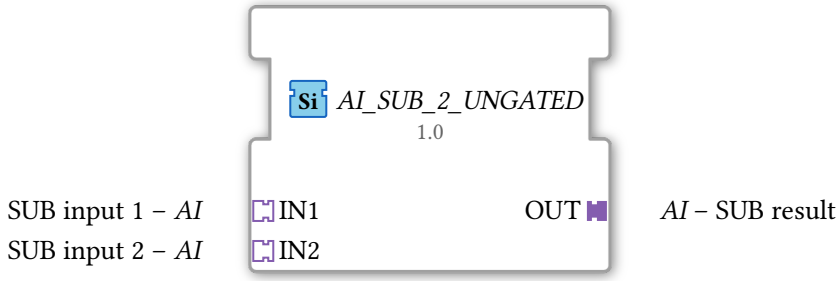
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `AI_SUB_2` ist ein spezialisierter und hocheffizienter Baustein für die moderne, adapterbasierte Programmierung in 4diac. Er eignet sich hervorragend für saubere Softwarearchitekturen im Bereich der analogen Signalvorverarbeitung, bei denen Übersichtlichkeit und Kapselung im Vordergrund stehen.

AI_SUB_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AI_SUB_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AI_SUB_2_UNGATED` ist ein generischer, arithmetischer Funktionsbaustein zur Durchführung einer Subtraktion. Er ist speziell für den Einsatz mit analogen Schnittstellen-Adaptern konzipiert. Der Baustein berechnet die Differenz zwischen zwei analogen Eingangswerten und stellt das Ergebnis an einem analogen Ausgangs-Adapter zur Verfügung. Durch die Verwendung von Adaptern wird das Schnittstellendesign in der 4diac-ide übersichtlich gehalten.

Schnittstellenstruktur

Da dieser Funktionsbaustein vollständig auf einer Adapter-Architektur basiert, besitzt er keine direkten, klassischen Ereignis- oder Datenkanäle auf der Bausteinoberfläche. Die gesamte Kommunikation wird über die Adapter abgewickelt.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden (die Ereignissteuerung ist in den Adaptern gekapselt).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden (die Ereignissteuerung ist in den Adaptern gekapselt).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AI`): Der erste analoge Eingangswert (Minuend) für die Subtraktionsoperation.
- **IN2** (Typ: `adapter::types::unidirectional::AI`): Der zweite analoge Eingangswert (Subtrahend), der von `IN1` abgezogen wird.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AI`): Das Ergebnis der Subtraktion (Differenz: `IN1 - IN2`).

Funktionsweise

Der Baustein führt die grundlegende arithmetische Subtraktion aus:

$$\mathrm{OUT} = \mathrm{IN1} - \mathrm{IN2} \tag{2.35}$$

Sobald sich die Eingangswerte an den Sockets `IN1` oder `IN2` ändern bzw. ein entsprechendes Aktualisierungsereignis über die Adapter empfangen wird, berechnet der Baustein intern die Differenz. Das Ergebnis sowie das zugehörige Aktualisierungsereignis werden unmittelbar an den Ausgangs-Plug `OUT` weitergeleitet.

Technische Besonderheiten

- **Generischer Bausteintyp (`GEN_AI_SUB`)**: Der Baustein ist intern als generischer Typ definiert. Dies ermöglicht eine flexible Anpassung an die im System verwendeten analogen Datentypen der Adapter.
- **Adapter-Kapselung**: Die Verwendung von unidirektionalen analogen Adaptern (`unidirectional::AI`) bündelt Daten- und Eventleitungen. Dies minimiert den Verdrahtungsaufwand im FBD (Function Block Diagram) und erhöht die Übersichtlichkeit signifikant.

Zustandsübersicht

Der Baustein verhält sich wie ein rein funktionaler Baustein ohne internen Speicher (analog zu einem klassisch kombinatorischen Gatter). Er besitzt keine komplexe Execution Control Chart (ECC). Jede Aktivität an den Eingangs-Adaptern triggert direkt die Berechnung und aktualisiert den Ausgangs-Adapter.

Anwendungsszenarien

- **Differenzdruck- und Differenztemperaturmessung**: Berechnung von physikalischen Differenzwerten in der Prozesstechnik (z. B. Ermittlung des Druckabfalls über einem Filter oder der Temperaturdifferenz zwischen Vor- und Rücklauf).
- **Regelungsabweichung**: Berechnung der Regelabweichung ($e = w - x$) in Regelungskreisen, bei denen Soll- und Istwert als Adapterstrukturen vorliegen.
- **Nullpunktkorrektur / Offset-Abzug**: Subtraktion eines Kalibrier- oder Offset-Wertes von einem kontinuierlichen Messsignal.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-Subtraktionsbausteinen der IEC 61131-3 (wie z. B. einem klassischen `SUB`-Baustein mit diskreten Pins) entfällt beim `AI_SUB_2_UNGATED` das manuelle Verdrahten von separaten `REQ`- und `CNF`-Eventleitungen sowie einzelnen analogen Datenleitungen. Alle relevanten Informationen werden sauber in den `AI`-Adaptern übertragen.

- `AI_SUB_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

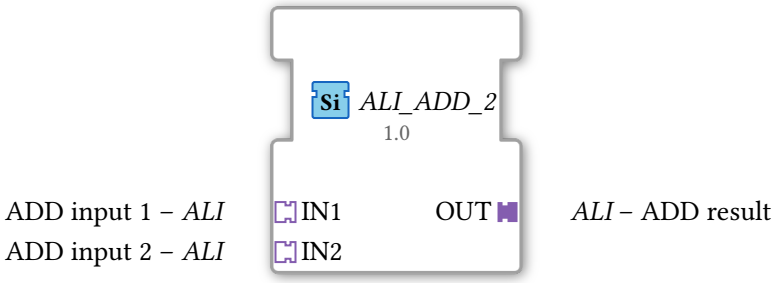
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AI_SUB_2_UNGATED` ist ein spezialisierter und hocheffizienter Baustein für die moderne, adapterbasierte Programmierung in 4diac. Er eignet sich hervorragend für saubere Softwarearchitekturen im Bereich der analogen Signalvorverarbeitung, bei denen Übersichtlichkeit und Kapselung im Vordergrund stehen.

ALI_ADD_2



(Kein Bild vorhanden)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ALI_ADD_2` ist ein generischer arithmetischer Baustein zur Durchführung einer Addition (ADD) auf Basis des Standards IEC 61131-3. Er ist als Standard-Arithmetikfunktion klassifiziert und dient dazu, zwei Eingangswerte über eine Adapter-Schnittstelle zu addieren und das Ergebnis auszugeben.

Durch die Verwendung von Adaptern ermöglicht dieser Baustein eine hochgradig modulare und übersichtliche Verdrahtung innerhalb der 4diac-IDE.

Schnittstellenstruktur

Ereignis-Eingänge
Keine direkten Ereignis-Eingänge vorhanden. Die Steuerung und Synchronisation erfolgt über die Adapter.

Ereignis-Ausgänge
Keine direkten Ereignis-Ausgänge vorhanden. Die Steuerung und Synchronisation erfolgt über die Adapter.

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden.

Adapter
- **IN1 (Socket / Buchse)**: Eingangsadapter vom Typ `adapter::types::unidirectional::ALI`. Repräsentiert den ersten Summanden der Addition.
- **IN2 (Socket / Buchse)**: Eingangsadapter vom Typ `adapter::types::unidirectional::ALI`. Repräsentiert den zweiten Summanden der Addition.
- **OUT (Plug / Stecker)**: Ausgangsadapter vom Typ `adapter::types::unidirectional::ALI`. Liefert das mathematische Ergebnis der Addition (`IN1 + IN2`).

Funktionsweise
Der Funktionsbaustein `ALI_ADD_2` führt eine klassische mathematische Addition aus. Sobald sich die Werte an den Eingangs-Adaptern `IN1` und/oder `IN2` ändern oder ein entsprechendes Übertragungsereignis über die Adapter getriggert wird, berechnet der Baustein die Summe:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} \tag{2.36}$$

Das Ergebnis wird unmittelbar über den Ausgangsadapter `OUT` zur Verfügung gestellt. Da der Baustein als generischer Typ definiert ist (implementiert über die generische Klasse `GEN_ALI_ADD`), ist er nicht auf einen festen Datentyp fixiert, sondern kann sich an verschiedene kompatible numerische Datentypen anpassen.

Technische Besonderheiten
- **Generische Implementierung**: Durch das Attribut `GenericClassName` mit dem Wert `'GEN_ALI_ADD'` kann der Baustein flexibel für unterschiedliche Datentypen (z. B. INT, REAL, LREAL) instanziiert werden.
- **Adapter-Kopplung**: Im Gegensatz zu traditionellen Funktionsbausteinen, die diskrete Pins für Daten und Events nutzen, bündelt dieser Baustein die Signale in unidirektionalen `ALI`-Adaptern. Dies reduziert das „Drahtgewitter“ im Funktionsplan erheblich.

Zustandsübersicht
Der Funktionsbaustein ist zustandslos (kombinatorisches Verhalten). Er besitzt keine interne State Machine (ECC). Jede Änderung an den Eingängen führt direkt zur Aktualisierung des Ausgangs.

Anwendungsszenarien
- **Signalverarbeitung**: Summierung von zwei Sensorwerten oder physikalischen Größen, die bereits über ein Adapter-System in der Anwendung verteilt werden.
- **Sollwert-Generierung**: Aufaddieren eines Basis-Sollwerts und eines Korrektur-Offsets.
- **Modularer Anlagenbau**: Einsatz in Systemen, bei denen Schnittstellen konsequent über Adapter abgebildet werden, um die Wiederverwendbarkeit und Lesbarkeit des Codes zu maximieren.

Vergleich mit ähnlichen Bausteinen
Klassische `ADD`-Bausteine nach IEC 61131-3 verfügen meist über direkte Dateneingänge (`IN1`, `IN2`...) und Event-Pins (`REQ`, `CNF`). Der `ALI_ADD_2` unterscheidet sich dadurch, dass er diese Kanäle in Adaptern (`ALI`) kapselt. Dies führt zu einer besseren Kapselung und einer saubereren Strukturierung in komplexen 4diac-

Applikationen, erfordert jedoch, dass die verbundenen Bausteine ebenfalls das `ALI`-Adapterprotokoll unterstützen.

- **`ALI_ADD_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

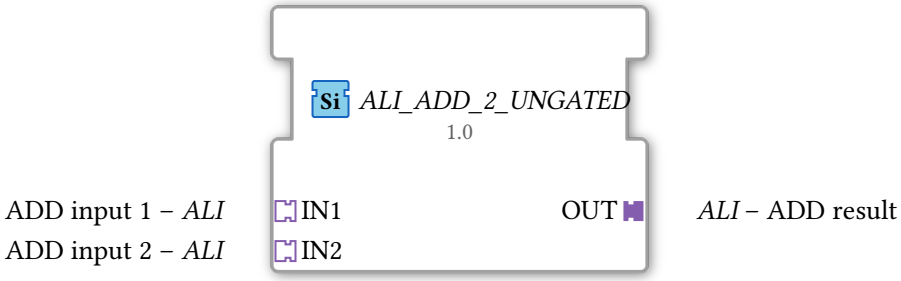
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `ALI_ADD_2` ist ein spezialisierter, hocheffizienter Additionsbaustein für adapterbasierte Architekturen in der 4diac-IDE. Er vereint die mathematische Standardfunktionalität der Addition mit den strukturellen Vorteilen moderner, adapterbasierter Kommunikationsparadigmen.

ALI_ADD_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALI_ADD_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild vorhanden)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ALI_ADD_2_UNGATED` ist ein generischer arithmetischer Baustein zur Durchführung einer Addition (ADD) auf Basis des Standards IEC 61131-3. Er ist als Standard-Arithmetikfunktion klassifiziert und dient dazu, zwei Eingangswerte über eine Adapter-Schnittstelle zu addieren und das Ergebnis auszugeben.

Durch die Verwendung von Adaptern ermöglicht dieser Baustein eine hochgradig modulare und übersichtliche Verdrahtung innerhalb der 4diac-IDE.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die Steuerung und Synchronisation erfolgt über die Adapter.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Steuerung und Synchronisation erfolgt über die Adapter.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

- **IN1 (Socket / Buchse)**: Eingangsadapter vom Typ `adapter::types::unidirectional::ALI`. Repräsentiert den ersten Summanden der Addition.
- **IN2 (Socket / Buchse)**: Eingangsadapter vom Typ `adapter::types::unidirectional::ALI`. Repräsentiert den zweiten Summanden der Addition.
- **OUT (Plug / Stecker)**: Ausgangsadapter vom Typ `adapter::types::unidirectional::ALI`. Liefert das mathematische Ergebnis der Addition (`IN1 + IN2`).

Funktionsweise

Der Funktionsbaustein `ALI_ADD_2_UNGATED` führt eine klassische mathematische Addition aus. Sobald sich die Werte an den Eingangs-Adaptern `IN1` und/oder `IN2` ändern oder ein entsprechendes Übertragungsereignis über die Adapter getriggert wird, berechnet der Baustein die Summe:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} \tag{2.37}$$

Das Ergebnis wird unmittelbar über den Ausgangsadapter `OUT` zur Verfügung gestellt. Da der Baustein als generischer Typ definiert ist (implementiert über die generische Klasse `GEN_ALI_ADD`), ist er nicht auf einen festen Datentyp fixiert, sondern kann sich an verschiedene kompatible numerische Datentypen anpassen.

Technische Besonderheiten

- **Generische Implementierung**: Durch das Attribut `GenericClassName` mit dem Wert `'GEN_ALI_ADD'` kann der Baustein flexibel für unterschiedliche Datentypen (z. B. INT, REAL, LREAL) instanziiert werden.
- **Adapter-Kopplung**: Im Gegensatz zu traditionellen Funktionsbausteinen, die diskrete Pins für Daten und Events nutzen, bündelt dieser Baustein die Signale in unidirektionalen `ALI`-Adaptern. Dies reduziert das „Drahtgewitter“ im Funktionsplan erheblich.

Zustandsübersicht

Der Funktionsbaustein ist zustandslos (kombinatorisches Verhalten). Er besitzt keine interne State Machine (ECC). Jede Änderung an den Eingängen führt direkt zur Aktualisierung des Ausgangs.

Anwendungsszenarien

- **Signalverarbeitung**: Summierung von zwei Sensorwerten oder physikalischen Größen, die bereits über ein Adapter-System in der Anwendung verteilt werden.
- **Sollwert-Generierung**: Aufaddieren eines Basis-Sollwerts und eines Korrektur-Offsets.
- **Modularer Anlagenbau**: Einsatz in Systemen, bei denen Schnittstellen konsequent über Adapter abgebildet werden, um die Wiederverwendbarkeit und Lesbarkeit des Codes zu maximieren.

Vergleich mit ähnlichen Bausteinen

Klassische `ADD`-Bausteine nach IEC 61131-3 verfügen meist über direkte Dateneingänge (`IN1`, `IN2`...) und Event-Pins (`REQ`, `CNF`). Der `ALI_ADD_2_UNGATED` unterscheidet sich dadurch, dass er diese Kanäle in Adaptern (`ALI`) kapselt. Dies führt zu einer besseren Kapselung und einer saubereren Strukturierung in komplexen 4diac-Applikationen, erfordert jedoch, dass die verbundenen Bausteine ebenfalls das `ALI`-Adapterprotokoll unterstützen.

- **`ALI_ADD_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

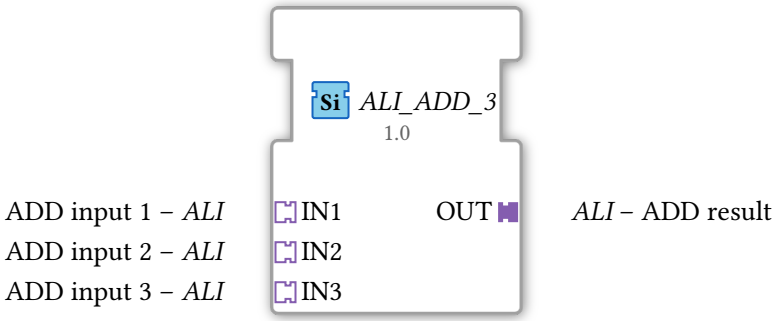
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `ALI_ADD_2_UNGATED` ist ein spezialisierter, hocheffizienter Additionsbaustein für adapterbasierte Architekturen in der 4diac-IDE. Er vereint die mathematische Standardfunktionalität der Addition mit den strukturellen Vorteilen moderner, adapterbasierter Kommunikationsparadigmen.

ALI_ADD_3



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ALI_ADD_3` ist ein generischer, arithmetischer Baustein zur Addition von drei Werten. Er basiert auf dem Standard IEC 61131-3 für arithmetische Funktionen. Anstelle von klassischen, separaten Daten- und Ereignisanschlüssen nutzt dieser Baustein unidirektionale Adapter des Typs `ALI`, um eine strukturierte und übersichtliche Signalübertragung in 4diac-ide zu ermöglichen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die Steuerung und Triggerung erfolgt implizit über die angeschlossenen Adapter.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Weiterleitung von Ereignissen erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Der Baustein kommuniziert ausschließlich über Adapter-Schnittstellen:

- **Sockets (Eingangs-Schnittstellen):**
  - `IN1` (Typ: `adapter::types::unidirectional::ALI`): Schnittstelle für den ersten Summanden (Eingangswert 1).
  - `IN2` (Typ: `adapter::types::unidirectional::ALI`): Schnittstelle für den zweiten Summanden (Eingangswert 2).
  - `IN3` (Typ: `adapter::types::unidirectional::ALI`): Schnittstelle für den dritten Summanden (Eingangswert 3).
- **Plugs (Ausgangs-Schnittstellen):**
  - `OUT` (Typ: `adapter::types::unidirectional::ALI`): Schnittstelle zur Ausgabe des berechneten Ergebnisses.

Funktionsweise

Der Baustein `ALI_ADD_3` führt eine standardmäßige arithmetische Addition durch. Sobald an den Eingangs-Adaptern (`IN1`, `IN2`, `IN3`) neue Werte signalisiert werden, berechnet der Baustein die Summe nach folgender Formel:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} \tag{2.38}$$

Das Ergebnis sowie das zugehörige Aktualisierungsereignis werden anschließend über den Ausgangs-Adapter `OUT` an die nachfolgenden Bausteine weitergegeben.

Technische Besonderheiten

- **Generische Implementierung:** Der Baustein ist als generischer Typ (`GEN_ALI_ADD`) deklariert. Dies ermöglicht eine flexible Anpassung an verschiedene Datentypen, die vom `ALI`-Adapter unterstützt werden.
- **Verwendung von Adaptern:** Durch die Kapselung von Daten und Events in unidirektionalen Adaptern (`ALI`) wird die Anzahl der notwendigen Verbindungslinien im Application-Editor von 4diac-ide drastisch reduziert, was zu einer besseren Übersichtlichkeit führt.

Zustandsübersicht

Der Funktionsbaustein arbeitet rein zustandslos (stateless). Es gibt keine interne Zustandsmaschine (ECC). Die Berechnung erfolgt rein daten- bzw. ereignisgesteuert: Ein eintreffendes Ereignis an einem der Eingangs-Adapter triggert direkt die Addition und die Aktualisierung des Ausgangs-Adapters.

Anwendungsszenarien

- **Messwert-Summierung:** Zusammenfassung von drei analogen Einzelmesswerten (z. B. Leistungsaufnahme von drei Phasen, Durchflussmengen aus drei Rohren) zu einem Gesamtwert.
- **Mittelwertvorbereitung:** Aufsummierung von drei Werten zur anschließenden Division durch 3 in einem nachfolgenden Baustein.
- **Strukturierte Signalverarbeitung:** Einsatz in komplexen Projekten, bei denen analoge Signale standardisiert und übersichtlich über Adapter-Kanäle geführt werden sollen.

Vergleich mit ähnlichen Bausteinen

- **Standard-ADD (IEC 61131-3):** Ein Standard-ADD-Baustein verwendet klassische Daten- und Ereignis-Verbindungen. `ALI_ADD_3` hingegen kapselt diese Schnittstellen in Adaptern, was die visuelle Komplexität im Systementwurf verringert.
- **Kaskadierte 2-fach-Addierer:** Um drei Werte mit herkömmlichen 2-fach-Addierern zu addieren, müssten zwei Bausteine in Reihe geschaltet werden. `ALI_ADD_3` spart hierbei einen Baustein sowie die dazwischenliegende Instanziierung und Verdrahtung ein.
- `ALI_ADD_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

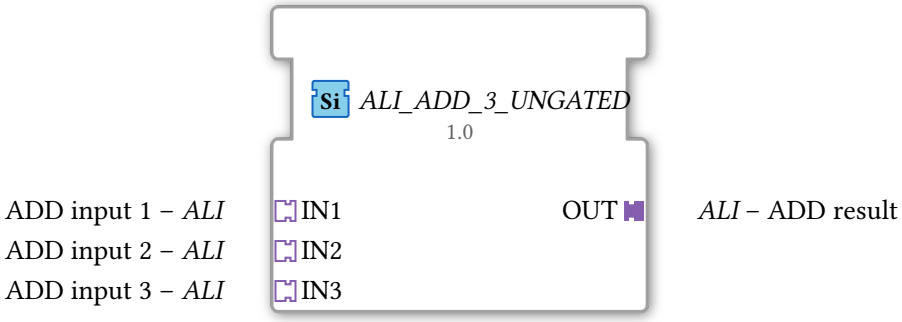
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

`ALI_ADD_3` ist ein kompakter und effizienter Hilfsbaustein für 4diac-ide. Durch die konsequente Nutzung von unidirektionalen Adaptern trägt er maßgeblich zur Erstellung von sauberem, modularisiertem und leicht lesbarem Steuerungscode bei der Verarbeitung von analogen Signalen bei.

ALI_ADD_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALI_ADD_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ALI_ADD_3_UNGATED` ist ein generischer, arithmetischer Baustein zur Addition von drei Werten. Er basiert auf dem Standard IEC 61131-3 für arithmetische Funktionen. Anstelle von klassischen, separaten Daten- und Ereignisanschlüssen nutzt dieser Baustein unidirektionale Adapter des Typs `ALI`, um eine strukturierte und übersichtliche Signalübertragung in 4diac-ide zu ermöglichen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die Steuerung und Triggerung erfolgt implizit über die angeschlossenen Adapter.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Weiterleitung von Ereignissen erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Der Baustein kommuniziert ausschließlich über Adapter-Schnittstellen:

- **Sockets (Eingangs-Schnittstellen):**
  - `IN1` (Typ: `adapter::types::unidirectional::ALI`): Schnittstelle für den ersten Summanden (Eingangswert 1).
  - `IN2` (Typ: `adapter::types::unidirectional::ALI`): Schnittstelle für den zweiten Summanden (Eingangswert 2).
  - `IN3` (Typ: `adapter::types::unidirectional::ALI`): Schnittstelle für den dritten Summanden (Eingangswert 3).
- **Plugs (Ausgangs-Schnittstellen):**
  - `OUT` (Typ: `adapter::types::unidirectional::ALI`): Schnittstelle zur Ausgabe des berechneten Ergebnisses.

Funktionsweise

Der Baustein `ALI_ADD_3_UNGATED` führt eine standardmäßige arithmetische Addition durch. Sobald an den Eingangs-Adaptern (`IN1`, `IN2`, `IN3`) neue Werte signalisiert werden, berechnet der Baustein die Summe nach folgender Formel:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} \tag{2.39}$$

Das Ergebnis sowie das zugehörige Aktualisierungsereignis werden anschließend über den Ausgangs-Adapter `OUT` an die nachfolgenden Bausteine weitergegeben.

Technische Besonderheiten

- **Generische Implementierung:** Der Baustein ist als generischer Typ (`GEN_ALI_ADD`) deklariert. Dies ermöglicht eine flexible Anpassung an verschiedene Datentypen, die vom `ALI`-Adapter unterstützt werden.
- **Verwendung von Adaptern:** Durch die Kapselung von Daten und Events in unidirektionalen Adaptern (`ALI`) wird die Anzahl der notwendigen Verbindungslinien im Application-Editor von 4diac-ide drastisch reduziert, was zu einer besseren Übersichtlichkeit führt.

Zustandsübersicht

Der Funktionsbaustein arbeitet rein zustandslos (stateless). Es gibt keine interne Zustandsmaschine (ECC). Die Berechnung erfolgt rein daten- bzw. ereignisgesteuert: Ein eintreffendes Ereignis an einem der Eingangs-Adapter triggert direkt die Addition und die Aktualisierung des Ausgangs-Adapters.

Anwendungsszenarien

- **Messwert-Summierung:** Zusammenfassung von drei analogen Einzelmesswerten (z. B. Leistungsaufnahme von drei Phasen, Durchflussmengen aus drei Rohren) zu einem Gesamtwert.
- **Mittelwertvorbereitung:** Aufsummierung von drei Werten zur anschließenden Division durch 3 in einem nachfolgenden Baustein.
- **Strukturierte Signalverarbeitung:** Einsatz in komplexen Projekten, bei denen analoge Signale standardisiert und übersichtlich über Adapter-Kanäle geführt werden sollen.

Vergleich mit ähnlichen Bausteinen

- **Standard-ADD (IEC 61131-3):** Ein Standard-ADD-Baustein verwendet klassische Daten- und Ereignis-Verbindungen. `ALI_ADD_3_UNGATED` hingegen kapselt diese Schnittstellen in Adaptern, was die visuelle Komplexität im Systementwurf verringert.
- **Kaskadierte 2-fach-Addierer:** Um drei Werte mit herkömmlichen 2-fach-Addierern zu addieren, müssten zwei Bausteine in Reihe geschaltet werden. `ALI_ADD_3_UNGATED` spart hierbei einen Baustein sowie die dazwischenliegende Instanziierung und Verdrahtung ein.
- **`ALI_ADD_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

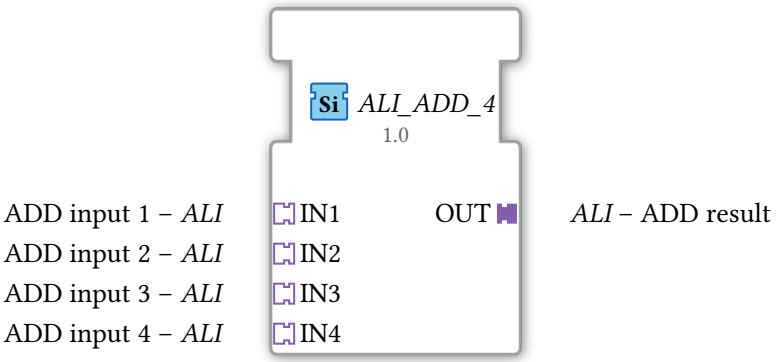
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

`ALI_ADD_3_UNGATED` ist ein kompakter und effizienter Hilfsbaustein für 4diac-ide. Durch die konsequente Nutzung von unidirektionalen Adaptern trägt er maßgeblich zur Erstellung von sauberem, modularisiertem und leicht lesbarem Steuerungscode bei der Verarbeitung von analogen Signalen bei.

ALI_ADD_4



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ALI_ADD_4` ist ein generischer arithmetischer Funktionsbaustein nach dem IEC 61131-3 Standard. Seine Hauptaufgabe besteht darin, die Summe von vier Eingangswerten zu berechnen. Im Gegensatz zu klassischen arithmetischen Bausteinen, die mit elementaren Datentypen und standardmäßigen Event-Eingängen/-Ausgängen arbeiten, nutzt dieser Baustein unidirektionale Adapter-Verbindungen des Typs `ALI` (Analog Line Interface). Dies ermöglicht eine gekapselte und übersichtliche Signalübertragung von Daten und den dazugehörigen Trigger-Ereignissen innerhalb der 4diac-ide.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine direkten Ereignis-Eingänge. Die Ereignissteuerung wird vollständig über die angeschlossenen Adapter abgewickelt.

Ereignis-Ausgänge

Der Baustein besitzt keine direkten Ereignis-Ausgänge. Die Weiterleitung von Ereignissen erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Der Baustein besitzt keine direkten Daten-Eingänge. Die Datenübergabe erfolgt über die Eingangs-Adapter (Sockets).

Daten-Ausgänge

Der Baustein besitzt keine direkten Daten-Ausgänge. Die Datenausgabe erfolgt über den Ausgangs-Adapter (Plug).

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::ALI`): Erster Summand für die Addition.
- **IN2** (Typ: `adapter::types::unidirectional::ALI`): Zweiter Summand für die Addition.
- **IN3** (Typ: `adapter::types::unidirectional::ALI`): Dritter Summand für die Addition.
- **IN4** (Typ: `adapter::types::unidirectional::ALI`): Vierter Summand für die Addition.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::ALI`): Ausgabe des berechneten Additionsergebnisses.

Funktionsweise

Sobald an einem der Eingangs-Adapter (`IN1` bis `IN4`) ein neues Ereignis mitsamt aktualisiertem Datenwert eingeht, führt der Baustein die mathematische Addition der vier Werte aus:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} + \mathrm{IN4} \tag{2.40}$$

Das Ergebnis der Berechnung wird unmittelbar an den Ausgangs-Adapter `OUT` übergeben und ein entsprechendes Ausgabe-Ereignis getriggert, um nachfolgende Bausteine im Kontrollfluss zu informieren.

Technische Besonderheiten

- **Generische Klasse:** Der Baustein basiert auf der generischen Basisklasse `GEN_ALI_ADD` (Attribut: `eclipse4diac::core::GenericClassName`).
- **Adapter-Kopplung:** Durch die Verwendung von unidirektionalen Adaptern (`ALI`) werden Daten und Ereignisse in einer einzigen Verbindung gebündelt. Dies reduziert die visuelle Komplexität (den „Kabelsalat“) im FBD-Editor (Function Block Diagram) von 4diac-ide drastisch.
- **IEC 61131-3 Konformität:** Entspricht den Richtlinien für standardmäßige arithmetische Funktionen.

Zustandsübersicht

Der Baustein `ALI_ADD_4` ist ein zustandsloser (kombinatorischer) Funktionsbaustein. Er speichert keine internen Zustände oder historischen Werte zwischen den Berechnungszyklen. Jede Berechnung basiert

ausschließlich auf den aktuell an den Eingangs-Adaptern anliegenden Werten zum Zeitpunkt des Trigger-Ereignisses.

Anwendungsszenarien

- **Messwert-Konsolidierung:** Aufsummierung von vier analogen Messgrößen (z. B. Teillastströme, Durchflussmengen oder Einzelleistungen) zu einem Gesamtwert.
- **Mischprozesse:** Berechnung von Gesamtgewichten oder Gesamtvolumina in Rezepturanlagen, bei denen vier Komponenten zusammengeführt werden.
- **Signalverarbeitung:** Strukturierte Vorverarbeitung von Sensorsignalen in modular aufgebauten Steuerungsarchitekturen, die konsequent auf Adapter-Verbindungen setzen.

Vergleich mit ähnlichen Bausteinen

- **Standard ADD-Baustein (IEC 61131-3):** Ein Standard-ADD-Baustein verwendet elementare Datentypen (z. B. `REAL`, `INT`) sowie separate Event-Ports (`REQ` / `CNF`). `ALI_ADD_4` hingegen kapselt diese Schnittstellen in Adaptern, was die Wiederverwendbarkeit und Übersichtlichkeit erhöht.
- **ALI_ADD_2 / ALI_ADD_3:** Diese Bausteine funktionieren nach demselben Prinzip, sind jedoch auf die Addition von nur zwei bzw. drei Eingangssignalen ausgelegt.

- `ALI_ADD_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

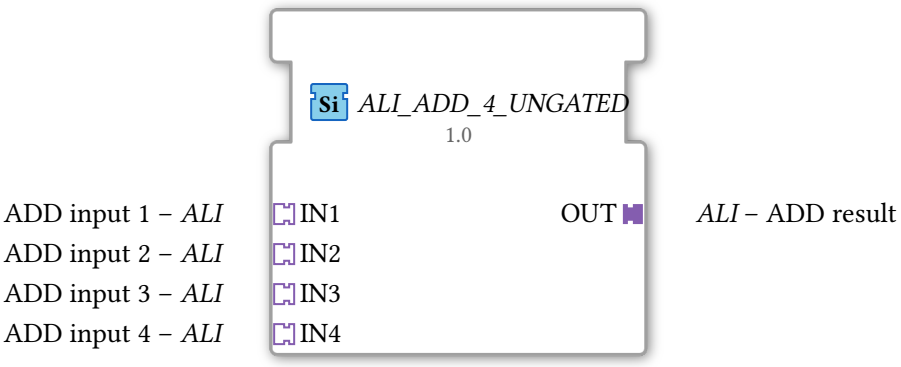
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der Funktionsbaustein `ALI_ADD_4` bietet eine elegante und saubere Möglichkeit, vier analoge Werte innerhalb einer adapterbasierten 4diac-Applikation zu addieren. Durch die Kapselung von Daten- und Eventkanälen in `ALI`-Adaptern unterstützt er die Erstellung modularer, übersichtlicher und leicht wartbarer Steuerungssoftware.

ALI_ADD_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALI_ADD_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ALI_ADD_4_UNGATED` ist ein generischer arithmetischer Funktionsbaustein nach dem IEC 61131-3 Standard. Seine Hauptaufgabe besteht darin, die Summe von vier Eingangswerten zu berechnen. Im Gegensatz zu klassischen arithmetischen Bausteinen, die mit elementaren Datentypen und standardmäßigen Event-Eingängen/-Ausgängen arbeiten, nutzt dieser Baustein unidirektionale Adapter-Verbindungen des Typs `ALI` (Analog Line Interface). Dies ermöglicht eine gekapselte und übersichtliche Signalübertragung von Daten und den dazugehörigen Trigger-Ereignissen innerhalb der 4diac-ide.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine direkten Ereignis-Eingänge. Die Ereignissteuerung wird vollständig über die angeschlossenen Adapter abgewickelt.

Ereignis-Ausgänge

Der Baustein besitzt keine direkten Ereignis-Ausgänge. Die Weiterleitung von Ereignissen erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Der Baustein besitzt keine direkten Daten-Eingänge. Die Datenübergabe erfolgt über die Eingangs-Adapter (Sockets).

Daten-Ausgänge

Der Baustein besitzt keine direkten Daten-Ausgänge. Die Datenausgabe erfolgt über den Ausgangs-Adapter (Plug).

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::ALI`): Erster Summand für die Addition.
- **IN2** (Typ: `adapter::types::unidirectional::ALI`): Zweiter Summand für die Addition.
- **IN3** (Typ: `adapter::types::unidirectional::ALI`): Dritter Summand für die Addition.
- **IN4** (Typ: `adapter::types::unidirectional::ALI`): Vierter Summand für die Addition.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::ALI`): Ausgabe des berechneten Additionsergebnisses.

Funktionsweise

Sobald an einem der Eingangs-Adapter (`IN1` bis `IN4`) ein neues Ereignis mitsamt aktualisiertem Datenwert eingeht, führt der Baustein die mathematische Addition der vier Werte aus:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} + \mathrm{IN4} \tag{2.41}$$

Das Ergebnis der Berechnung wird unmittelbar an den Ausgangs-Adapter `OUT` übergeben und ein entsprechendes Ausgabe-Ereignis getriggert, um nachfolgende Bausteine im Kontrollfluss zu informieren.

Technische Besonderheiten

- **Generische Klasse:** Der Baustein basiert auf der generischen Basisklasse `GEN_ALI_ADD` (Attribut: `eclipse4diac::core::GenericClassName`).
- **Adapter-Kopplung:** Durch die Verwendung von unidirektionalen Adaptern (`ALI`) werden Daten und Ereignisse in einer einzigen Verbindung gebündelt. Dies reduziert die visuelle Komplexität (den „Kabelsalat“) im FBD-Editor (Function Block Diagram) von 4diac-ide drastisch.
- **IEC 61131-3 Konformität:** Entspricht den Richtlinien für standardmäßige arithmetische Funktionen.

Zustandsübersicht

Der Baustein `ALI_ADD_4_UNGATED` ist ein zustandsloser (kombinatorischer) Funktionsbaustein. Er speichert keine internen Zustände oder historischen Werte zwischen den Berechnungszyklen. Jede Berech-

nung basiert ausschließlich auf den aktuell an den Eingangs-Adaptern anliegenden Werten zum Zeitpunkt des Trigger-Ereignisses.

Anwendungsszenarien

- **Messwert-Konsolidierung:** Aufsummierung von vier analogen Messgrößen (z. B. Teillastströme, Durchflussmengen oder Einzelleistungen) zu einem Gesamtwert.
- **Mischprozesse:** Berechnung von Gesamtgewichten oder Gesamtvolumina in Rezepturanlagen, bei denen vier Komponenten zusammengeführt werden.
- **Signalverarbeitung:** Strukturierte Vorverarbeitung von Sensorsignalen in modular aufgebauten Steuerungsarchitekturen, die konsequent auf Adapter-Verbindungen setzen.

Vergleich mit ähnlichen Bausteinen

- **Standard ADD-Baustein (IEC 61131-3):** Ein Standard-ADD-Baustein verwendet elementare Datentypen (z. B. `REAL`, `INT`) sowie separate Event-Ports (`REQ` / `CNF`). `ALI_ADD_4_UNGATED` hingegen kapselt diese Schnittstellen in Adaptern, was die Wiederverwendbarkeit und Übersichtlichkeit erhöht.
- **ALI_ADD_2 / ALI_ADD_3:** Diese Bausteine funktionieren nach demselben Prinzip, sind jedoch auf die Addition von nur zwei bzw. drei Eingangssignalen ausgelegt.

- `ALI_ADD_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

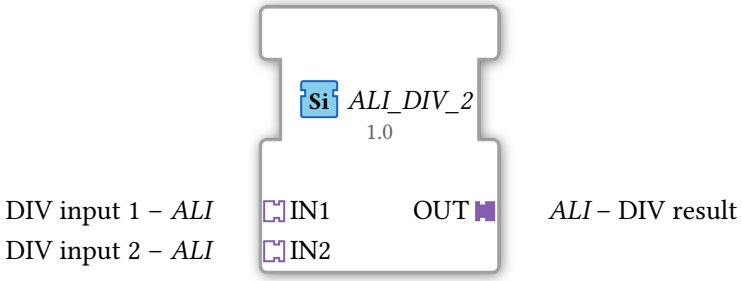
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der Funktionsbaustein `ALI_ADD_4_UNGATED` bietet eine elegante und saubere Möglichkeit, vier analoge Werte innerhalb einer adapterbasierten 4diac-Applikation zu addieren. Durch die Kapselung von Daten- und Eventkanälen in `ALI`-Adaptern unterstützt er die Erstellung modularer, übersichtlicher und leicht wartbarer Steuerungssoftware.

ALI_DIV_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ALI_DIV_2` ist ein generischer arithmetischer Funktionsbaustein für die IEC 61499, der für die Durchführung von Divisionsoperationen entwickelt wurde. Die Besonderheit dieses Bausteins liegt in der Verwendung von unidirektionalen Adaptern (`ALI`) anstelle von klassischen Daten- und Ereignis-Schnittstellen. Dies ermöglicht eine modulare und übersichtliche Strukturierung von Datenflüssen innerhalb der 4diac-IDE.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Eingänge. Die Steuerung der Ausführung erfolgt über die angebundenen Adapter.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Ausgänge. Die Weiterleitung von Ereignissen erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Eingänge.

Daten-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Ausgänge.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::ALI`): Der Dividend (Zähler) der Division.
- **IN2** (Typ: `adapter::types::unidirectional::ALI`): Der Divisor (Nenner) der Division.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::ALI`): Das Ergebnis der mathematischen Division ($OUT = IN1/IN2$).

Funktionsweise

Der Baustein `ALI_DIV_2` berechnet den Quotienten aus den Werten, die über die Eingangs-Adapter `IN1` und `IN2` bereitgestellt werden. Das Ergebnis wird an den Ausgangs-Adapter `OUT` übergeben.

Da es sich um einen generischen Funktionsbaustein (`GEN_ALI_DIV`) handelt, kann der Baustein flexibel mit verschiedenen Datentypen arbeiten, sofern die verwendeten `ALI`-Adapter diese unterstützen (z. B. Ganzzahlen oder Gleitkommazahlen). Die eigentliche Berechnung wird durch die im Adapter definierten Ereignisse getriggert. Sobald neue Daten an den Eingängen anliegen und ein entsprechendes Trigger-Signal über die Adapter-Schnittstelle empfangen wird, wird die Division ausgeführt und das Ergebnis am Ausgang bereitgestellt.

Technische Besonderheiten

- **Generische Implementierung:** Durch die Zuordnung zur generischen Klasse `GEN_ALI_DIV` ist der Baustein nicht auf einen festen Datentyp (wie z.B. `REAL` oder `INT`) beschränkt. Er passt sich dynamisch den Typdefinitionen der verbundenen Adapter an.
- **Kapselung durch Adapter:** Die Verwendung unidirektionaler `ALI`-Adapter reduziert den Verkabelungsaufwand in der 4diac-IDE drastisch, da Daten und zugehörige Trigger-Ereignisse in einer einzigen Verbindung gebündelt übertragen werden.
- **Division durch Null:** Bei der Implementierung und Laufzeitumgebung muss darauf geachtet werden, wie mit einem Wert von `0` am Eingang `IN2` umgegangen wird, um Systemabstürze zu vermeiden (z. B. durch standardmäßige Rückgabe von `NaN`, Unendlich oder das Abfangen über die Adapter-Fehlerbehandlung).

Zustandsübersicht

Da es sich bei `ALI_DIV_2` um einen daten- und ereignisgesteuerten Rechenbaustein handelt, besitzt er keine komplexe interne Zustandsmaschine (keinen klassischen ECC). Er verhält sich wie ein funktionaler Block:

1. **Warten auf Trigger:** Der Baustein wartet auf ein Aktualisierungsereignis an den Eingangs-Adaptern `IN1` oder `IN2`.
2. **Berechnung:** Bei Signalempfang werden die Daten ausgelesen und dividiert.
3. **Ausgabe:** Das Ergebnis wird an `OUT` angelegt und das zugehörige Ereignis über den Ausgangs-Adapter signalisiert.

Anwendungsszenarien

- **Signalnormierung und Skalierung:** Division von Sensorwerten durch konstante Faktoren zur Einheitenumrechnung innerhalb einer Adapter-basierten Steuerungsarchitektur.
- **Verhältnisberechnung:** Bestimmung von Verhältnissen (z. B. Kraftstoff-Luft-Gemisch, Durchflussverhältnisse) in verfahrenstechnischen Anlagen.
- **Verteilte Steuerungssysteme:** Einsatz in komplexen IEC 61499 Applikationen, bei denen ein sauberer, modularer Signalfluss mittels Adaptern erzwungen wird.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-DIV-Baustein der IEC 61131-3 bzw. IEC 61499, welcher mit klassischen Daten-Pins (IN1, IN2, OUT) und separaten Event-Pins (REQ, CNF) arbeitet, bietet der ALI_DIV_2 durch die Nutzung von Adaptern eine deutlich aufgeräumtere Oberfläche im Funktionsplan. Er eignet sich besonders für fortgeschrittene Software-Architekturen, die konsequent auf die Vorteile von Adaptern zur Strukturierung setzen.

- **ALI_DIV_2_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

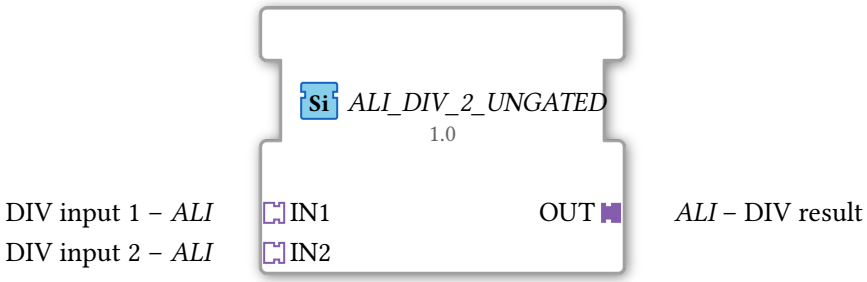
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der ALI_DIV_2 ist ein hocheffizienter und flexibel einsetzbarer Divisionsbaustein für moderne IEC 61499 Anwendungen. Durch die konsequente Nutzung generischer Typen und des ALI-Adapter-Konzepts trägt er maßgeblich zur Übersichtlichkeit, Wiederverwendbarkeit und Wartbarkeit von Steuerungssoftware in der 4diac-IDE bei.

ALI_DIV_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von ALI_DIV_2. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein ALI_DIV_2_UNGATED ist ein generischer arithmetischer Funktionsbaustein für die IEC 61499, der für die Durchführung von Divisionsoperationen entwickelt wurde. Die Besonderheit dieses Bausteins liegt in der Verwendung von unidirektionalen Adaptern (ALI) anstelle von klassischen Daten- und Ereignis-Schnittstellen. Dies ermöglicht eine modulare und übersichtliche Strukturierung von Datenflüssen innerhalb der 4diac-IDE.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Eingänge. Die Steuerung der Ausführung erfolgt über die angebundenen Adapter.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Ausgänge. Die Weiterleitung von Ereignissen erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Eingänge.

Daten-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Ausgänge.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::ALI`): Der Dividend (Zähler) der Division.
- **IN2** (Typ: `adapter::types::unidirectional::ALI`): Der Divisor (Nenner) der Division.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::ALI`): Das Ergebnis der mathematischen Division ($OUT = IN1/IN2$).

Funktionsweise

Der Baustein `ALI_DIV_2_UNGATED` berechnet den Quotienten aus den Werten, die über die Eingangs-Adapter `IN1` und `IN2` bereitgestellt werden. Das Ergebnis wird an den Ausgangs-Adapter `OUT` übergeben.

Da es sich um einen generischen Funktionsbaustein (`GEN_ALI_DIV`) handelt, kann der Baustein flexibel mit verschiedenen Datentypen arbeiten, sofern die verwendeten `ALI`-Adapter diese unterstützen (z. B. Ganzzahlen oder Gleitkommazahlen). Die eigentliche Berechnung wird durch die im Adapter definierten Ereignisse getriggert. Sobald neue Daten an den Eingängen anliegen und ein entsprechendes Trigger-Signal über die Adapter-Schnittstelle empfangen wird, wird die Division ausgeführt und das Ergebnis am Ausgang bereitgestellt.

Technische Besonderheiten

- **Generische Implementierung:** Durch die Zuordnung zur generischen Klasse `GEN_ALI_DIV` ist der Baustein nicht auf einen festen Datentyp (wie z.B. `REAL` oder `INT`) beschränkt. Er passt sich dynamisch den Typdefinitionen der verbundenen Adapter an.
- **Kapselung durch Adapter:** Die Verwendung unidirektionaler `ALI`-Adapter reduziert den Verkabelungsaufwand in der 4diac-IDE drastisch, da Daten und zugehörige Trigger-Ereignisse in einer einzigen Verbindung gebündelt übertragen werden.
- **Division durch Null:** Bei der Implementierung und Laufzeitumgebung muss darauf geachtet werden, wie mit einem Wert von `0` am Eingang `IN2` umgegangen wird, um Systemabstürze zu vermeiden (z. B. durch standardmäßige Rückgabe von `NaN`, Unendlich oder das Abfangen über die Adapter-Fehlerbehandlung).

Zustandsübersicht

Da es sich bei `ALI_DIV_2_UNGATED` um einen daten- und ereignisgesteuerten Rechenbaustein handelt, besitzt er keine komplexe interne Zustandsmaschine (keinen klassischen ECC). Er verhält sich wie ein funktionaler Block:

1. **Warten auf Trigger:** Der Baustein wartet auf ein Aktualisierungsereignis an den Eingangs-Adaptern `IN1` oder `IN2`.
2. **Berechnung:** Bei Signalempfang werden die Daten ausgelesen und dividiert.
3. **Ausgabe:** Das Ergebnis wird an `OUT` angelegt und das zugehörige Ereignis über den Ausgangs-Adapter signalisiert.

Anwendungsszenarien

- **Signalnormierung und Skalierung:** Division von Sensorwerten durch konstante Faktoren zur Einheitenumrechnung innerhalb einer Adapter-basierten Steuerungsarchitektur.
- **Verhältnisberechnung:** Bestimmung von Verhältnissen (z. B. Kraftstoff-Luft-Gemisch, Durchflussverhältnisse) in verfahrenstechnischen Anlagen.
- **Verteilte Steuerungssysteme:** Einsatz in komplexen IEC 61499 Applikationen, bei denen ein sauberer, modularer Signalfluss mittels Adaptern erzwungen wird.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-DIV-Baustein der IEC 61131-3 bzw. IEC 61499, welcher mit klassischen Daten-Pins (IN1, IN2, OUT) und separaten Event-Pins (REQ, CNF) arbeitet, bietet der ALI_DIV_2_UNGATED durch die Nutzung von Adaptern eine deutlich aufgeräumtere Oberfläche im Funktionsplan. Er eignet sich besonders für fortgeschrittene Software-Architekturen, die konsequent auf die Vorteile von Adaptern zur Strukturierung setzen.

- **ALI_DIV_2_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- **ALI_DIV_2**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

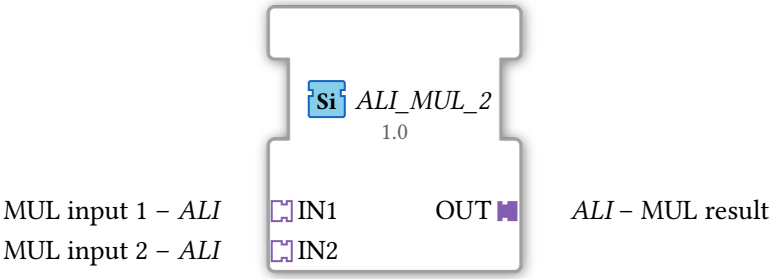
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der ALI_DIV_2_UNGATED ist ein hocheffizienter und flexibel einsetzbarer Divisionsbaustein für moderne IEC 61499 Anwendungen. Durch die konsequente Nutzung generischer Typen und des ALI-Adapter-Konzepts trägt er maßgeblich zur Übersichtlichkeit, Wiederverwendbarkeit und Wartbarkeit von Steuerungssoftware in der 4diac-IDE bei.

ALI_MUL_2



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein ALI_MUL_2 dient zur Durchführung einer arithmetischen Multiplikation zweier Werte. Es handelt sich um einen generischen Funktionsbaustein, der für eine saubere und modulare Architektur in IEC 61499-Systemen konzipiert ist. Anstelle von klassisch diskreten Daten- und Ereignisports nutzt dieser Baustein unidirektionale Adapter, um Steuerungsereignisse und Datenwerte gebündelt zu übertragen.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung ist in den Eingangs-Adaptern integriert.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge vorhanden. Die Ereignissteuerung ist im Ausgangs-Adapter integriert.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübertragung erfolgt gekoppelt über die Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Die Datenübertragung erfolgt gekoppelt über den Ausgangs-Adapter.

Adapter

- **Sockets (Eingangs-Adapter):**
  - `IN1` (Typ: `adapter::types::unidirectional::ALI`): Erster Eingang für die Multiplikation (Multiplikand).
  - `IN2` (Typ: `adapter::types::unidirectional::ALI`): Zweiter Eingang für die Multiplikation (Multiplikator).
- **Plugs (Ausgangs-Adapter):**
  - `OUT` (Typ: `adapter::types::unidirectional::ALI`): Ausgang für das Ergebnis der Multiplikation (Produkt).

Funktionsweise

Der Funktionsbaustein `ALI_MUL_2` multipliziert die über die Adapter-Sockets `IN1` und `IN2` bereitgestellten Werte. Sobald ein entsprechendes Trigger-Ereignis über die Eingangs-Adapter signalisiert wird, führt der Baustein die Berechnung aus:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \tag{2.42}$$

Das berechnete Ergebnis sowie das zugehörige Ausgangs-Ereignis werden anschließend über den Ausgangs-Plug `OUT` an die nachfolgenden Bausteine weitergeleitet.

Technische Besonderheiten

- **Generischer Charakter:** Der Baustein ist über das Attribut `GenericClassName` als `GEN_ALI_MUL` deklariert. Dies ermöglicht eine flexible Handhabung verschiedener Datentypen, die durch das übergeordnete System oder den verwendeten Adaptertyp spezifiziert werden.
- **Adapter-Kopplung:** Durch die Verwendung des unidirektionalen Adaptertyps `ALI` werden Daten und Trigger-Events in einer einzigen Verbindung gebündelt, was den Verdrahtungsaufwand in der Entwicklungsumgebung erheblich minimiert.

Zustandsübersicht

Da es sich um einen rein datenfluss- und adapterbasierten Funktionsbaustein handelt, besitzt der Baustein keine komplexe interne Zustandsmaschine (Execution Control Chart - ECC). Die Abarbeitung erfolgt ereignisgesteuert und direkt gekoppelt an die über die Adapter eingehenden Trigger.

Anwendungsszenarien

- **Messwertskalierung:** Multiplikation von Sensorwerten mit Kalibrierungs- oder Skalierungsfaktoren.
- **Leistungsberechnung:** Ermittlung von physikalischen Größen wie der elektrischen Leistung ($P = U \times I$) in adapterbasierten Steuerungsnetzwerken.
- **Kompakte Berechnungsnetzwerke:** Einsatz in komplexen mathematischen Berechnungen, bei denen die visuelle Übersichtlichkeit des Steuerungsdiagramms durch den Einsatz von Adaptern gewahrt bleiben soll.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu klassischen mathematischen Bausteinen der IEC 61131-3 (wie z. B. einem Standard-`MUL`-Baustein mit diskreten `REQ` / `CNF` Events und separaten Datenleitungen) entfällt beim `ALI_MUL_2` das manuelle Verknüpfen von Event- und Datenpfaden. Dies verhindert Fehler bei der Event-Daten-Zuordnung und führt zu einem wesentlich aufgeräumteren und lesbareren Applikationsdiagramm in 4diac-ide.

- **`ALI_MUL_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

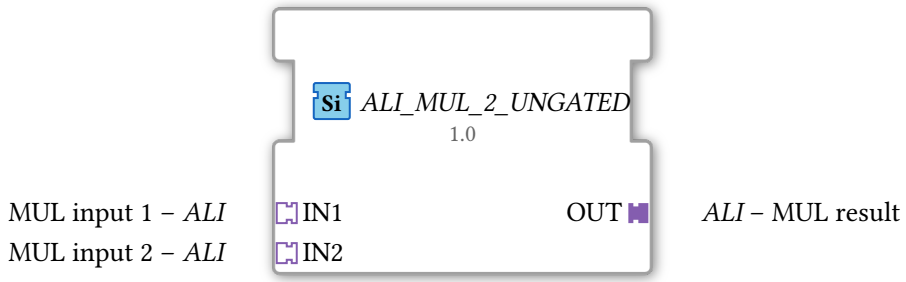
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `ALI_MUL_2` ist ein moderner und effizienter Funktionsbaustein zur Multiplikation. Durch die konsequente Nutzung von unidirektionalen Adaptern eignet er sich hervorragend für serviceorientierte und modular aufgebaute Steuerungssysteme, bei denen Übersichtlichkeit und einfache Wartbarkeit im Vordergrund stehen.

ALI_MUL_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALI_MUL_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ALI_MUL_2_UNGATED` dient zur Durchführung einer arithmetischen Multiplikation zweier Werte. Es handelt sich um einen generischen Funktionsbaustein, der für eine saubere und modulare Architektur in IEC 61499-Systemen konzipiert ist. Anstelle von klassisch diskreten Daten- und Ereignisports nutzt dieser Baustein unidirektionale Adapter, um Steuerungsereignisse und Datenwerte gebündelt zu übertragen.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung ist in den Eingangs-Adaptern integriert.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge vorhanden. Die Ereignissteuerung ist im Ausgangs-Adapter integriert.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübertragung erfolgt gekoppelt über die Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Die Datenübertragung erfolgt gekoppelt über den Ausgangs-Adapter.

Adapter

- **Sockets (Eingangs-Adapter):**
  - `IN1` (Typ: `adapter::types::unidirectional::ALI`): Erster Eingang für die Multiplikation (Multiplikand).

  - `IN2` (Typ: `adapter::types::unidirectional::ALI`): Zweiter Eingang für die Multiplikation (Multiplikator).
- **Plugs (Ausgangs-Adapter):**
  - `OUT` (Typ: `adapter::types::unidirectional::ALI`): Ausgang für das Ergebnis der Multiplikation (Produkt).

Funktionsweise

Der Funktionsbaustein `ALI_MUL_2_UNGATED` multipliziert die über die Adapter-Sockets `IN1` und `IN2` bereitgestellten Werte. Sobald ein entsprechendes Trigger-Ereignis über die Eingangs-Adapter signalisiert wird, führt der Baustein die Berechnung aus:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \tag{2.43}$$

Das berechnete Ergebnis sowie das zugehörige Ausgangs-Ereignis werden anschließend über den Ausgangs-Plug `OUT` an die nachfolgenden Bausteine weitergeleitet.

Technische Besonderheiten

- **Generischer Charakter:** Der Baustein ist über das Attribut `GenericClassName` als `GEN_ALI_MUL` deklariert. Dies ermöglicht eine flexible Handhabung verschiedener Datentypen, die durch das übergeordnete System oder den verwendeten Adaptertyp spezifiziert werden.
- **Adapter-Kopplung:** Durch die Verwendung des unidirektionalen Adaptertyps `ALI` werden Daten und Trigger-Events in einer einzigen Verbindung gebündelt, was den Verdrahtungsaufwand in der Entwicklungsumgebung erheblich minimiert.

Zustandsübersicht

Da es sich um einen rein datenfluss- und adapterbasierten Funktionsbaustein handelt, besitzt der Baustein keine komplexe interne Zustandsmaschine (Execution Control Chart - ECC). Die Abarbeitung erfolgt ereignisgesteuert und direkt gekoppelt an die über die Adapter eingehenden Trigger.

Anwendungsszenarien

- **Messwertskalierung:** Multiplikation von Sensorwerten mit Kalibrierungs- oder Skalierungsfaktoren.
- **Leistungsberechnung:** Ermittlung von physikalischen Größen wie der elektrischen Leistung ($P = U \times I$) in adapterbasierten Steuerungsnetzwerken.
- **Kompakte Berechnungsnetzwerke:** Einsatz in komplexen mathematischen Berechnungen, bei denen die visuelle Übersichtlichkeit des Steuerungsdiagramms durch den Einsatz von Adaptern gewahrt bleiben soll.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu klassischen mathematischen Bausteinen der IEC 61131-3 (wie z. B. einem Standard-`MUL`-Baustein mit diskreten `REQ` / `CNF` Events und separaten Datenleitungen) entfällt beim `ALI_MUL_2_UNGATED` das manuelle Verknüpfen von Event- und Datenpfaden. Dies verhindert Fehler bei der Event-Daten-Zuordnung und führt zu einem wesentlich aufgeräumteren und lesbareren Applikationsdiagramm in 4diac-ide.

- `ALI_MUL_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- `ALI_MUL_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

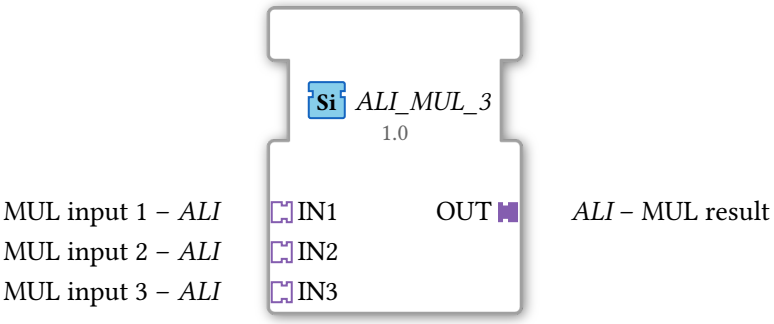
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `ALI_MUL_2_UNGATED` ist ein moderner und effizienter Funktionsbaustein zur Multiplikation. Durch die konsequente Nutzung von unidirektionalen Adaptern eignet er sich hervorragend für serviceorientierte und modular aufgebaute Steuerungssysteme, bei denen Übersichtlichkeit und einfache Wartbarkeit im Vordergrund stehen.

ALI_MUL_3



Kein Bild verfügbar. Der Funktionsblock wird grafisch mit drei eingangsseitigen Sockets (IN1, IN2, IN3) und einem ausgangsseitigen Plug (OUT) dargestellt.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ALI_MUL_3` ist ein generischer Arithmetik-Baustein für die 4diac-IDE, der zur Multiplikation von drei Eingangswerten dient. Er zeichnet sich dadurch aus, dass die Ein- und Ausgänge nicht über klassische Daten- und Ereignis-Pins, sondern vollständig über unidirektionale Adapter des Typs `ALI` realisiert sind. Dies ermöglicht eine strukturierte, modulare und übersichtliche Kapselung von Signal- und Datenflüssen in IEC 61499-Anwendungen.

Schnittstellenstruktur

Da dieser Funktionsblock vollständig auf der Adapter-Technologie basiert, besitzt er keine direkten, klassischen Ereignis- oder Datenkanäle auf der Hauptelebene.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden (Signalverarbeitung erfolgt über die Adapter).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden (Signalweiterleitung erfolgt über den Ausgangs-Adapter).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingänge)

- **IN1** (Typ: `adapter::types::unidirectional::ALI`): Der erste Faktor (Multiplikand 1) für die arithmetische Multiplikation.
- **IN2** (Typ: `adapter::types::unidirectional::ALI`): Der zweite Faktor (Multiplikand 2) für die arithmetische Multiplikation.
- **IN3** (Typ: `adapter::types::unidirectional::ALI`): Der dritte Faktor (Multiplikand 3) für die arithmetische Multiplikation.

Plugs (Ausgänge)

- **OUT** (Typ: `adapter::types::unidirectional::ALI`): Das Ergebnis der Multiplikation (Produkt).

Funktionsweise

Die primäre Aufgabe von `ALI_MUL_3` ist die Berechnung des Produkts aus drei Werten:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \tag{2.44}$$

Die Steuerung des Berechnungszeitpunkts und die Übergabe der Werte erfolgen gekapselt über die `ALI`-Adapter. Sobald ein entsprechendes Trigger-Ereignis über die Eingangs-Adapter signalisiert wird, liest der Baustein die Werte aus den Sockets `IN1`, `IN2` und `IN3` aus, berechnet das mathematische Produkt und stellt das Ergebnis am Plug `OUT` bereit, woraufhin ein entsprechendes Ausgangsereignis initiiert wird.

Technische Besonderheiten

- **Generischer Baustein:** Durch das Attribut `GenericClassName` mit dem Wert `'GEN_ALI_MUL'` ist der Funktionsblock datentypunabhängig konzipiert. Er kann (je nach Ausprägung des verwendeten `ALI`-Adapters) für verschiedene numerische Datentypen (z. B. `INT`, `REAL`, `LREAL`) eingesetzt werden.
- **Sauberes Design:** Die Verwendung von Adaptern reduziert den Verdrahtungsaufwand im Funktionsplan (FBD) erheblich, da Kontroll- und Datenfluss in einer einzigen Verbindung gebündelt werden.
- **Unidirektionale Adapter:** Da es sich um das Paket `adapter::types::unidirectional` handelt, fließen Daten und Trigger gerichtet von den Sockets (`IN1`-`IN3`) zum Plug (`OUT`).

Zustandsübersicht

Der Funktionsblock verhält sich wie ein klassischer, zustandsloser mathematischer Operator (kombinatorisches Verhalten):

- **Bereitschaft:** Der Baustein wartet auf eingehende Aktualisierungsereignisse an den Adaptern `IN1`, `IN2` oder `IN3`.
- **Berechnung:** Bei Triggerung wird die Multiplikation ausgeführt.
- **Ausgabe:** Das Ergebnis wird direkt an den Ausgang `OUT` übergeben und der nachfolgende Baustein getriggert.

Anwendungsszenarien

- **Berechnung physikalischer Größen:** Bestimmung von Volumina (Länge × Breite × Höhe) oder dreidimensionalen Skalierungen in der Fabrikautomatisierung.
- **Kaskadierte Verstärkungsfaktoren:** Multiplikation von drei verschiedenen Gain-Faktoren oder Korrekturwerten in Regelungskreisen (z. B. Sensorwert × Kalibrierfaktor × Skalierungsfaktor).
- **Signalverarbeitung:** Kombination mehrerer Gewichtungsfaktoren in der Prozessüberwachung.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-`MUL`-Baustein der IEC 61131-3:

- **Klassischer `MUL`-Block:** Benötigt explizite `REQ`- und `CNF`-Ereignisse sowie einzelne Daten-Pins für jeden Operanden. Das führt bei mehreren Eingängen schnell zu unübersichtlichen Plänen.
- **`ALI_MUL_3`:** Bündelt die Schnittstellen in vier übersichtliche Adapter-Anschlüsse. Zudem ist er fest auf genau drei Eingänge ausgelegt, was eine Zwischenstufe (Kaskadierung von zwei 2-fach Multiplizierern) einspart.

- `ALI_MUL_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

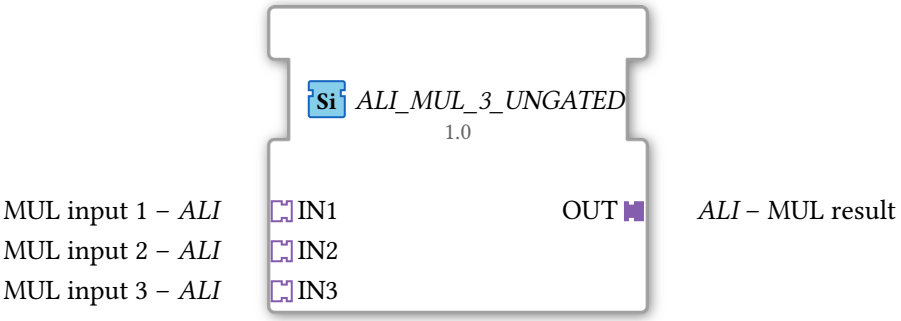
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `ALI_MUL_3` ist ein moderner und hocheffizienter Funktionsblock für mathematische Operationen in der 4diac-Umgebung. Durch die konsequente Nutzung von unidirektionalen Adaptern fördert er ein sauberes, modularisiertes Softwaredesign und eignet sich hervorragend für komplexe Berechnungen mit drei Eingangsvariablen.

ALI_MUL_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALI_MUL_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



Kein Bild verfügbar. Der Funktionsblock wird grafisch mit drei eingangsseitigen Sockets (IN1, IN2, IN3) und einem ausgangsseitigen Plug (OUT) dargestellt.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ALI_MUL_3_UNGATED` ist ein generischer Arithmetik-Baustein für die 4diac-IDE, der zur Multiplikation von drei Eingangswerten dient. Er zeichnet sich dadurch aus, dass die Ein- und Ausgänge nicht über klassische Daten- und Ereignis-Pins, sondern vollständig über unidirektionale Adapter des Typs `ALI` realisiert sind. Dies ermöglicht eine strukturierte, modulare und übersichtliche Kapselung von Signal- und Datenflüssen in IEC 61499-Anwendungen.

Schnittstellenstruktur

Da dieser Funktionsblock vollständig auf der Adapter-Technologie basiert, besitzt er keine direkten, klassischen Ereignis- oder Datenkanäle auf der Hauptelebene.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden (Signalverarbeitung erfolgt über die Adapter).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden (Signalweiterleitung erfolgt über den Ausgangs-Adapter).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingänge)

- **IN1** (Typ: `adapter::types::unidirectional::ALI`): Der erste Faktor (Multiplikand 1) für die arithmetische Multiplikation.
- **IN2** (Typ: `adapter::types::unidirectional::ALI`): Der zweite Faktor (Multiplikand 2) für die arithmetische Multiplikation.
- **IN3** (Typ: `adapter::types::unidirectional::ALI`): Der dritte Faktor (Multiplikand 3) für die arithmetische Multiplikation.

Plugs (Ausgänge)

- **OUT** (Typ: `adapter::types::unidirectional::ALI`): Das Ergebnis der Multiplikation (Produkt).

Funktionsweise

Die primäre Aufgabe von `ALI_MUL_3_UNGATED` ist die Berechnung des Produkts aus drei Werten:

$$\text{OUT} = \text{IN1} \times \text{IN2} \times \text{IN3} \tag{2.45}$$

Die Steuerung des Berechnungszeitpunkts und die Übergabe der Werte erfolgen gekapselt über die `ALI`-Adapter. Sobald ein entsprechendes Trigger-Ereignis über die Eingangs-Adapter signalisiert wird, liest der Baustein die Werte aus den Sockets `IN1`, `IN2` und `IN3` aus, berechnet das mathematische Produkt und stellt das Ergebnis am Plug `OUT` bereit, woraufhin ein entsprechendes Ausgangsereignis initiiert wird.

Technische Besonderheiten

- **Generischer Baustein:** Durch das Attribut `GenericClassName` mit dem Wert `'GEN_ALI_MUL'` ist der Funktionsblock datentypunabhängig konzipiert. Er kann (je nach Ausprägung des verwendeten `ALI`-Adapters) für verschiedene numerische Datentypen (z. B. `INT`, `REAL`, `LREAL`) eingesetzt werden.
- **Sauberes Design:** Die Verwendung von Adaptern reduziert den Verdrahtungsaufwand im Funktionsplan (FBD) erheblich, da Kontroll- und Datenfluss in einer einzigen Verbindung gebündelt werden.
- **Unidirektionale Adapter:** Da es sich um das Paket `adapter::types::unidirectional` handelt, fließen Daten und Trigger gerichtet von den Sockets (`IN1`-`IN3`) zum Plug (`OUT`).

Zustandsübersicht

Der Funktionsblock verhält sich wie ein klassischer, zustandsloser mathematischer Operator (kombinatorisches Verhalten):

- **Bereitschaft:** Der Baustein wartet auf eingehende Aktualisierungsereignisse an den Adaptern `IN1`, `IN2` oder `IN3`.
- **Berechnung:** Bei Triggerung wird die Multiplikation ausgeführt.
- **Ausgabe:** Das Ergebnis wird direkt an den Ausgang `OUT` übergeben und der nachfolgende Baustein getriggert.

Anwendungsszenarien

- **Berechnung physikalischer Größen:** Bestimmung von Volumina (Länge × Breite × Höhe) oder dreidimensionalen Skalierungen in der Fabrikautomatisierung.
- **Kaskadierte Verstärkungsfaktoren:** Multiplikation von drei verschiedenen Gain-Faktoren oder Korrekturwerten in Regelungskreisen (z. B. Sensorwert × Kalibrierfaktor × Skalierungsfaktor).
- **Signalverarbeitung:** Kombination mehrerer Gewichtungsfaktoren in der Prozessüberwachung.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-`MUL`-Baustein der IEC 61131-3:

- **Klassischer `MUL`-Block:** Benötigt explizite `REQ`- und `CNF`-Ereignisse sowie einzelne Daten-Pins für jeden Operanden. Das führt bei mehreren Eingängen schnell zu unübersichtlichen Plänen.
- **`ALI_MUL_3_UNGATED`:** Bündelt die Schnittstellen in vier übersichtliche Adapter-Anschlüsse. Zudem ist er fest auf genau drei Eingänge ausgelegt, was eine Zwischenstufe (Kaskadierung von zwei 2-fach Multiplizierern) einspart.

- `ALI_MUL_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

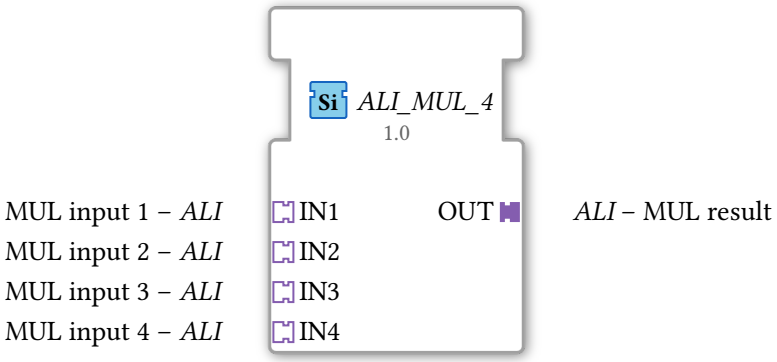
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `ALI_MUL_3_UNGATED` ist ein moderner und hocheffizienter Funktionsblock für mathematische Operationen in der 4diac-Umgebung. Durch die konsequente Nutzung von unidirektionalen Adaptern fördert er ein sauberes, modularisiertes Softwaredesign und eignet sich hervorragend für komplexe Berechnungen mit drei Eingangsvariablen.

ALI_MUL_4



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ALI_MUL_4` dient zur Durchführung einer arithmetischen Multiplikation von vier Eingangswerten. Es handelt sich um einen generischen Funktionsbaustein (Generic FB), der auf der Verwendung von unidirektionalen Adaptern (`ALI`) basiert. Durch die Nutzung von Adaptern anstelle von klassischen Daten- und Ereignispins wird die Verdrahtung in 4diac-ide übersichtlicher und modularer gestaltet.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Eingänge. Die Steuerung und Triggerung erfolgt über die angebundenen Adapter.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Ausgänge. Die Weiterleitung von Ereignissen erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübergabe erfolgt über die Eingangs-Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Das Ergebnis wird über den Ausgangs-Adapter bereitgestellt.

Adapter

Sockets (Eingangs-Adapter)

Die Sockets dienen als Eingänge für die zu multiplizierenden Werte.

- **IN1** (Typ: `adapter::types::unidirectional::ALI`): Erster Faktor für die Multiplikation.
- **IN2** (Typ: `adapter::types::unidirectional::ALI`): Zweiter Faktor für die Multiplikation.
- **IN3** (Typ: `adapter::types::unidirectional::ALI`): Dritter Faktor für die Multiplikation.
- **IN4** (Typ: `adapter::types::unidirectional::ALI`): Vierter Faktor für die Multiplikation.

Plugs (Ausgangs-Adapter)

Der Plug gibt das berechnete Ergebnis aus.

- **OUT** (Typ: `adapter::types::unidirectional::ALI`): Das berechnete Produkt der vier Eingangs-Adapter.

Funktionsweise

Sobald sich die Eingangswerte an den Adaptern `IN1` bis `IN4` ändern oder ein entsprechendes Aktualisierungsereignis über die Adapter getriggert wird, multipliziert der Baustein die vier Werte intern nach folgendem mathematischen Prinzip:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \times \mathrm{IN4} \tag{2.46}$$

Das Ergebnis sowie das zugehörige Aktualisierungsereignis werden anschließend über den Ausgangs-Adapter `OUT` ausgegeben.

Technische Besonderheiten

- **Generischer Baustein:** Durch das Attribut `GenericClassName` mit dem Wert `GEN_ALI_MUL` ist der Baustein datentypunabhängig konzipiert. Er kann je nach Implementierung der `ALI`-Adapter verschiedene numerische Datentypen (z. B. `INT`, `REAL`, `LREAL`) verarbeiten.
- **Adapter-Kopplung:** Durch den Einsatz von unidirektionalen `ALI`-Adaptern wird eine saubere Kapselung von Daten und Trigger-Ereignissen erreicht, was die Komplexität im Systementwurf reduziert.

Zustandsübersicht

Der Funktionsbaustein ist zustandslos (zustandsfreie, rein kombinatorische Verarbeitung). Es gibt keine internen Zustandsmaschinen (Execution Control Chart / ECC). Die Ausgangswerte hängen direkt von den an den Eingangs-Adaptern anliegenden Werten ab.

Anwendungsszenarien

- **Skalierung und Gewichtung:** Berechnung von zusammengesetzten Skalierungsfaktoren in der Prozessautomatisierung.
- **Volumen- und Massenberechnung:** Kontinuierliche Berechnung physikalischer Größen (z. B. $\ddot{L}ange \times Breite \times \ddot{H}ohe \times Dichte$).
- **Mehrstufige Verstärkungsregelungen:** Kaskadierte Signalverstärkung in der Mess- und Regelungstechnik.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-Multiplikationsbaustein (`MUL`) nach IEC 61131-3, der meist nur zwei direkte Dateneingänge besitzt, bietet `ALI_MUL_4`:

1. Die direkte Multiplikation von **vier** Faktoren in einem einzigen Schritt (weniger Bausteine auf dem Control-Canvas nötig).
2. Die Nutzung von **Adaptern** statt Einzelverdrahtungen, was die Übersichtlichkeit des Gesamtprogramms signifikant erhöht.

- **`ALI_MUL_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

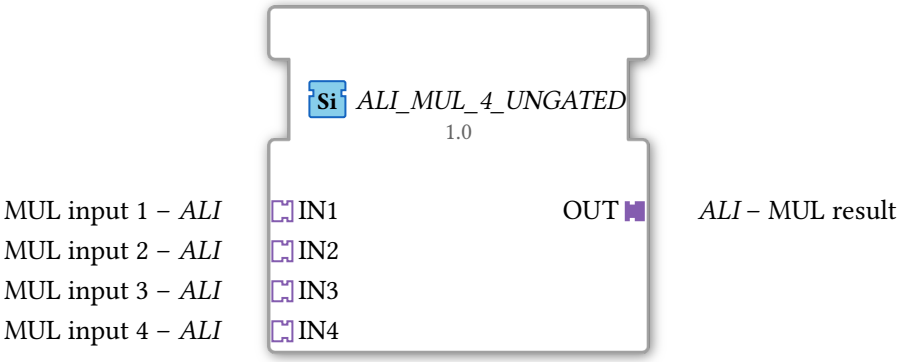
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `ALI_MUL_4` ist ein effizienter Hilfsbaustein für arithmetische Berechnungen in komplexen 4diac-Systemen. Er eignet sich besonders für Anwendungen, bei denen mehrere Werte kompakt und strukturiert miteinander multipliziert werden müssen, ohne die Benutzeroberfläche mit unzähligen Verbindungslinien zu überladen.

ALI_MUL_4_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `ALI_MUL_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ALI_MUL_4_UNGATED` dient zur Durchführung einer arithmetischen Multiplikation von vier Eingangswerten. Es handelt sich um einen generischen Funktionsbaustein (Generic FB), der auf der Verwendung von unidirektionalen Adaptern (`ALI`) basiert. Durch die Nutzung von Adaptern anstelle von klassischen Daten- und Ereignispins wird die Verdrahtung in 4diac-ide übersichtlicher und modularer gestaltet.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Eingänge. Die Steuerung und Triggerung erfolgt über die angebundenen Adapter.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Ausgänge. Die Weiterleitung von Ereignissen erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübergabe erfolgt über die Eingangs-Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Das Ergebnis wird über den Ausgangs-Adapter bereitgestellt.

Adapter

Sockets (Eingangs-Adapter)

Die Sockets dienen als Eingänge für die zu multiplizierenden Werte.

- **IN1** (Typ: `adapter::types::unidirectional::ALI`): Erster Faktor für die Multiplikation.
- **IN2** (Typ: `adapter::types::unidirectional::ALI`): Zweiter Faktor für die Multiplikation.
- **IN3** (Typ: `adapter::types::unidirectional::ALI`): Dritter Faktor für die Multiplikation.
- **IN4** (Typ: `adapter::types::unidirectional::ALI`): Vierter Faktor für die Multiplikation.

Plugs (Ausgangs-Adapter)

Der Plug gibt das berechnete Ergebnis aus.

- **OUT** (Typ: `adapter::types::unidirectional::ALI`): Das berechnete Produkt der vier Eingangs-Adapter.

Funktionsweise

Sobald sich die Eingangswerte an den Adaptern `IN1` bis `IN4` ändern oder ein entsprechendes Aktualisierungsereignis über die Adapter getriggert wird, multipliziert der Baustein die vier Werte intern nach folgendem mathematischen Prinzip:

$$\text{OUT} = \text{IN1} \times \text{IN2} \times \text{IN3} \times \text{IN4} \tag{2.47}$$

Das Ergebnis sowie das zugehörige Aktualisierungsereignis werden anschließend über den Ausgangs-Adapter `OUT` ausgegeben.

Technische Besonderheiten

- **Generischer Baustein:** Durch das Attribut `GenericClassName` mit dem Wert `GEN_ALI_MUL` ist der Baustein datentypunabhängig konzipiert. Er kann je nach Implementierung der `ALI`-Adapter verschiedene numerische Datentypen (z. B. `INT`, `REAL`, `LREAL`) verarbeiten.
- **Adapter-Kopplung:** Durch den Einsatz von unidirektionalen `ALI`-Adaptern wird eine saubere Kapselung von Daten und Trigger-Ereignissen erreicht, was die Komplexität im Systementwurf reduziert.

Zustandsübersicht

Der Funktionsbaustein ist zustandslos (zustandsfreie, rein kombinatorische Verarbeitung). Es gibt keine internen Zustandsmaschinen (Execution Control Chart / ECC). Die Ausgangswerte hängen direkt von den an den Eingangs-Adaptern anliegenden Werten ab.

Anwendungsszenarien

- **Skalierung und Gewichtung:** Berechnung von zusammengesetzten Skalierungsfaktoren in der Prozessautomatisierung.
- **Volumen- und Massenberechnung:** Kontinuierliche Berechnung physikalischer Größen (z. B. $\dot{L}ange \times Breite \times \dot{H}ohe \times Dichte$).
- **Mehrstufige Verstärkungsregelungen:** Kaskadierte Signalverstärkung in der Mess- und Regelungstechnik.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-Multiplikationsbaustein (`MUL`) nach IEC 61131-3, der meist nur zwei direkte Dateneingänge besitzt, bietet `ALI_MUL_4_UNGATED`:

1. Die direkte Multiplikation von **vier** Faktoren in einem einzigen Schritt (weniger Bausteine auf dem Control-Canvas nötig).
2. Die Nutzung von **Adaptern** statt Einzelverdrahtungen, was die Übersichtlichkeit des Gesamtprogramms signifikant erhöht.

- `ALI_MUL_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

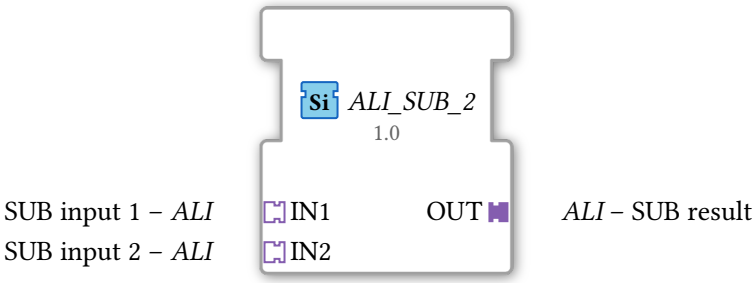
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `ALI_MUL_4_UNGATED` ist ein effizienter Hilfsbaustein für arithmetische Berechnungen in komplexen 4diac-Systemen. Er eignet sich besonders für Anwendungen, bei denen mehrere Werte kompakt und strukturiert miteinander multipliziert werden müssen, ohne die Benutzeroberfläche mit unzähligen Verbindungslinien zu überladen.

ALI_SUB_2



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein (FB) `ALI_SUB_2` ist ein generischer, arithmetischer Funktionsbaustein zur Durchführung einer Subtraktion. Er entspricht dem IEC 61131-3-Standard für standardmäßige arithmetische Funktionen. Der Baustein verwendet Adapter-Verbindungen (`ALI` - Arithmetic Logical Interface), um die Eingangsdaten entgegenzunehmen und das Ergebnis der Subtraktion auszugeben.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Eingänge. Die Steuerung und die Ereignisverarbeitung werden über die angeschlossenen Adapter realisiert.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Ausgänge. Die Ereignisweiterleitung erfolgt über die Ausgangs-Adapter.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübertragung erfolgt gekapselt über die Eingangs-Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Das Ergebnis wird über den Ausgangs-Adapter bereitgestellt.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::ALI`): Der erste Eingang (Minuend) für die Subtraktionsberechnung.
- **IN2** (Typ: `adapter::types::unidirectional::ALI`): Der zweite Eingang (Subtrahend) für die Subtraktionsberechnung.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::ALI`): Der Ausgang (Differenz) der Berechnung ($OUT = IN1 - IN2$).

Funktionsweise

Der Baustein `ALI_SUB_2` führt eine mathematische Subtraktion aus. Sobald Werte über die Eingangs-Adapter anliegen bzw. aktualisiert werden, berechnet der Baustein die Differenz:

$$\text{OUT} = \text{IN1} - \text{IN2} \tag{2.48}$$

Das Ergebnis wird unmittelbar an den Ausgangs-Adapter `OUT` übergeben. Da es sich um einen generischen Funktionsbaustein handelt, der auf der internen Klasse `GEN_ALI_SUB` basiert, kann sich der Baustein flexibel an die im Adapter definierten Datentypen anpassen.

Technische Besonderheiten

- **Generische Implementierung:** Der Baustein nutzt das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `'GEN_ALI_SUB'`. Dies erlaubt eine flexible Handhabung verschiedener Datentypen, die durch die Adapter-Schnittstellen vorgegeben werden.
- **Kapselung durch Adapter:** Durch die Verwendung von unidirektionalen `ALI`-Adaptern werden Daten und die dazugehörigen Steuerungsereignisse (wie z.B. Update-Events) in einer einzigen Verbindung gebündelt. Dies reduziert die Anzahl der sichtbaren Verbindungslinien im 4diac-Applikationseditor und erhöht die Übersichtlichkeit.

Zustandsübersicht

Der Baustein verhält sich rein zustandslos und reaktiv. Es gibt keine interne Zustandsmaschine (Execution Control Chart - ECC). Die Verarbeitung wird direkt durch eingehende Datenänderungen oder Ereignisse auf den Eingangs-Adaptern `IN1` und `IN2` getriggert.

Anwendungsszenarien

- **Signalverarbeitung und Skalierung:** Berechnung von Abweichungen, wie beispielsweise der Differenz zwischen einem Sollwert und einem Istwert (Regelungsabweichung) in Steuerungsanwendungen.
- **Offset-Kompensation:** Abzug von Nullpunktfehlern oder Offsets bei Sensordaten, die über Adapterstrukturen in das System eingespeist werden.
- **Modulare Systemarchitekturen:** Einsatz in komplexen, verteilten Steuerungssystemen, bei denen ein strukturierter und übersichtlicher Datenfluss durch den Einsatz von Adaptern gefordert ist.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zum Standard-Subtraktionsbaustein `SUB` aus der IEC 61131-3 Bibliothek benötigt `ALI_SUB_2` keine explizite Verdrahtung von Datentypen (wie `INT`, `REAL`) und Steuerereignissen (wie `REQ` / `CNF`). Während der Standard-`SUB`-Baustein diskrete Ein- und Ausgänge besitzt, bündelt `ALI_SUB_2` diese in `ALI`-Adaptern, was die Wiederverwendbarkeit und die Übersichtlichkeit im Systemdesign erheblich verbessert.

- `ALI_SUB_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

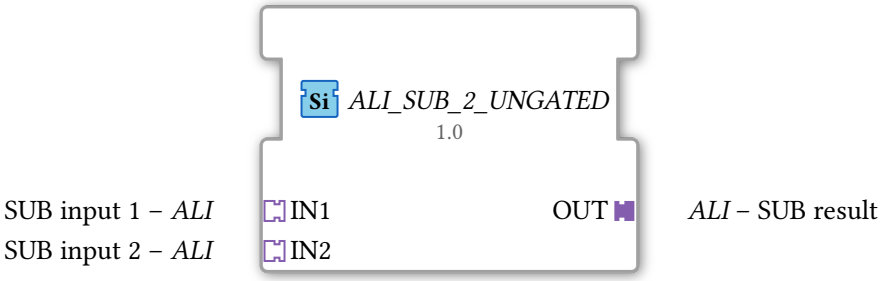
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `ALI_SUB_2` ist ein vielseitiger und wartungsfreundlicher Funktionsbaustein für arithmetische Subtraktionen in 4diac. Durch die konsequente Nutzung von Adaptern fügt er sich ideal in serviceorientierte und modulare Steuerungsarchitekturen ein und minimiert den Verdrahtungsaufwand im Entwicklungswerkzeug.

ALI_SUB_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALI_SUB_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein (FB) `ALI_SUB_2_UNGATED` ist ein generischer, arithmetischer Funktionsbaustein zur Durchführung einer Subtraktion. Er entspricht dem IEC 61131-3-Standard für standardmäßige arithmetische Funktionen. Der Baustein verwendet Adapter-Verbindungen (`ALI` - Arithmetic Logical Interface), um die Eingangsdaten entgegenzunehmen und das Ergebnis der Subtraktion auszugeben.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Eingänge. Die Steuerung und die Ereignisverarbeitung werden über die angeschlossenen Adapter realisiert.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Ausgänge. Die Ereignisweiterleitung erfolgt über die Ausgangs-Adapter.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübertragung erfolgt gekapselt über die Eingangs-Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Das Ergebnis wird über den Ausgangs-Adapter bereitgestellt.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::ALI`): Der erste Eingang (Minuend) für die Subtraktionsberechnung.
- **IN2** (Typ: `adapter::types::unidirectional::ALI`): Der zweite Eingang (Subtrahend) für die Subtraktionsberechnung.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::ALI`): Der Ausgang (Differenz) der Berechnung ($OUT = IN1 - IN2$).

Funktionsweise

Der Baustein `ALI_SUB_2_UNGATED` führt eine mathematische Subtraktion aus. Sobald Werte über die Eingangs-Adapter anliegen bzw. aktualisiert werden, berechnet der Baustein die Differenz:

$$\text{OUT} = \text{IN1} - \text{IN2} \tag{2.49}$$

Das Ergebnis wird unmittelbar an den Ausgangs-Adapter `OUT` übergeben. Da es sich um einen generischen Funktionsbaustein handelt, der auf der internen Klasse `GEN_ALI_SUB` basiert, kann sich der Baustein flexibel an die im Adapter definierten Datentypen anpassen.

Technische Besonderheiten

- **Generische Implementierung:** Der Baustein nutzt das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `'GEN_ALI_SUB'`. Dies erlaubt eine flexible Handhabung verschiedener Datentypen, die durch die Adapter-Schnittstellen vorgegeben werden.
- **Kapselung durch Adapter:** Durch die Verwendung von unidirektionalen `ALI`-Adaptern werden Daten und die dazugehörigen Steuerungsereignisse (wie z.B. Update-Events) in einer einzigen Verbindung gebündelt. Dies reduziert die Anzahl der sichtbaren Verbindungslinien im 4diac-Applikationseditor und erhöht die Übersichtlichkeit.

Zustandsübersicht

Der Baustein verhält sich rein zustandslos und reaktiv. Es gibt keine interne Zustandsmaschine (Execution Control Chart - ECC). Die Verarbeitung wird direkt durch eingehende Datenänderungen oder Ereignisse auf den Eingangs-Adaptern `IN1` und `IN2` getriggert.

Anwendungsszenarien

- **Signalverarbeitung und Skalierung:** Berechnung von Abweichungen, wie beispielsweise der Differenz zwischen einem Sollwert und einem Istwert (Regelungsabweichung) in Steuerungsanwendungen.
- **Offset-Kompensation:** Abzug von Nullpunktfehlern oder Offsets bei Sensordaten, die über Adapterstrukturen in das System eingespeist werden.
- **Modulare Systemarchitekturen:** Einsatz in komplexen, verteilten Steuerungssystemen, bei denen ein strukturierter und übersichtlicher Datenfluss durch den Einsatz von Adaptern gefordert ist.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zum Standard-Subtraktionsbaustein `SUB` aus der IEC 61131-3 Bibliothek benötigt `ALI_SUB_2_UNGATED` keine explizite Verdrahtung von Datentypen (wie `INT`, `REAL`) und Steuerereignissen (wie `REQ` / `CNF`). Während der Standard-`SUB`-Baustein diskrete Ein- und Ausgänge besitzt, bündelt `ALI_SUB_2_UNGATED` diese in `ALI`-Adaptern, was die Wiederverwendbarkeit und die Übersichtlichkeit im Systemdesign erheblich verbessert.

- `ALI_SUB_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

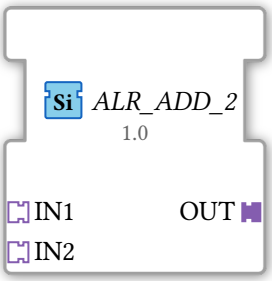
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `ALI_SUB_2_UNGATED` ist ein vielseitiger und wartungsfreundlicher Funktionsbaustein für arithmetische Subtraktionen in 4diac. Durch die konsequente Nutzung von Adaptern fügt er sich ideal in serviceorientierte und modulare Steuerungsarchitekturen ein und minimiert den Verdrahtungsaufwand im Entwicklungswerkzeug.

ALR_ADD_2



ADD input 1 – *ALR*
ADD input 2 – *ALR*
ALR – ADD result

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ALR_ADD_2` ist ein generischer, arithmetischer Additionsbaustein für die 4diac-IDE, der der Norm IEC 61131-3 entspricht. Er dient dazu, zwei Eingangswerte mathematisch zu addieren. Die Besonderheit dieses Bausteins liegt in der Verwendung von unidirektionalen Adaptern (`ALR`), die sowohl Daten als auch die zugehörige Kontrolllogik kapseln, um eine saubere und modulare Signalverarbeitung zu ermöglichen.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten, dedizierten Ereignis-Eingänge. Die Ereignissteuerung wird über die Adapter abgewickelt.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten, dedizierten Ereignis-Ausgänge. Die Ereignissteuerung wird über die Adapter abgewickelt.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge definiert, da die Datenübergabe vollständig über die Adapter erfolgt.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge definiert, da das Ergebnis über den Ausgangs-Adapter bereitgestellt wird.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::ALR`): Der erste Summand der Additionsoperation.
- **IN2** (Typ: `adapter::types::unidirectional::ALR`): Der zweite Summand der Additionsoperation.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::ALR`): Das Ergebnis der Addition ($OUT = IN1 + IN2$).

Funktionsweise

Der `ALR_ADD_2` liest die Werte über die beiden Eingangs-Adapter `IN1` und `IN2` ein. Sobald ein entsprechendes Ereignis über die Adapter signalisiert, dass neue Daten vorliegen, führt der Baustein die Addition aus:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} \tag{2.50}$$

Das berechnete Ergebnis wird anschließend über den Ausgangs-Adapter `OUT` zusammen mit dem entsprechenden Ausgangsereignis an nachfolgende Bausteine weitergeleitet. Da es sich um einen generischen Funktionsbaustein (`GEN_ALR_ADD`) handelt, kann er flexibel mit verschiedenen numerischen Datentypen arbeiten, die durch das ALR-Adapter-Interface unterstützt werden.

Technische Besonderheiten

- **Generischer Typ (`GEN_ALR_ADD`):** Der Baustein ist datentypunabhängig implementiert und passt sich den im ALR-Adapter definierten Datentypen an.
- **Adapter-Kapselung:** Durch die Verwendung unidirektionaler Adapter wird die Anzahl der sichtbaren Verbindungen im Control-Flow-Diagramm der 4diac-IDE drastisch reduziert, was die Übersichtlichkeit und Wartbarkeit des Gesamtsystems erhöht.
- **Konformität:** Entspricht den mathematischen Standards nach IEC 61131-3 für Standard-Arithmetikfunktionen.

Zustandsübersicht

Da die Logik dieses Bausteins rein daten- und ereignisgesteuert über die Adapter abläuft, besitzt der Baustein keine komplexe interne Zustandsmaschine (ECC). Er reagiert unmittelbar auf eingehende Datenänderungen/Ereignisse an den Sockets `IN1` und `IN2` und leitet das Ergebnis an den Plug `OUT` weiter.

Anwendungsszenarien

- **Signalzusammenführung:** Addition von zwei analogen Sensorwerten (z. B. Ermittlung eines Gesamtvolumenstroms aus zwei einzelnen Durchflussmessern).
- **Sollwert-Offset-Berechnung:** Beaufschlagung eines Basis-Sollwerts mit einem Korrekturoffset in verfahrenstechnischen Anlagen.
- **Kompakte Programmierung:** Verwendung in komplexen Regelungsschleifen, bei denen ein übersichtliches Routing mittels Adaptern bevorzugt wird.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-`ADD`-Funktionsbaustein (der separate Event-Eingänge wie `REQ`, Event-Ausgänge wie `CNF` sowie klassische Daten-Pins besitzt) verzichtet `ALR_ADD_2` vollständig auf diese klassischen Pins. Er ersetzt sie durch standardisierte `ALR`-Adapter. Dies vereinfacht die Verdrahtung in der 4diac-IDE erheblich, da Event und Daten in einer einzigen Verbindung gebündelt sind.

- `ALR_ADD_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

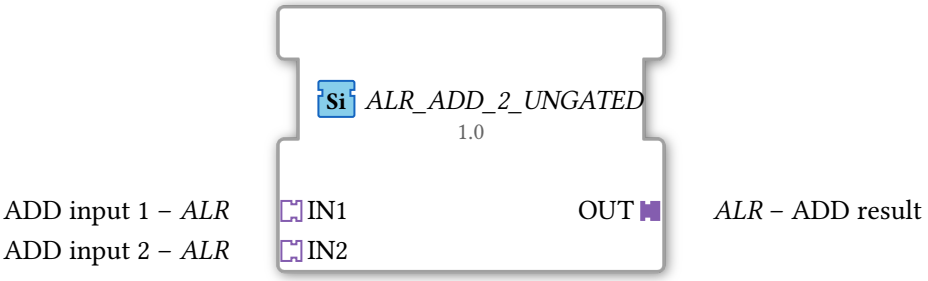
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `ALR_ADD_2` ist eine moderne und hocheffiziente Variante des klassischen Additionsbausteins. Er eignet sich hervorragend für serviceorientierte Architekturen und verteilte Steuerungssysteme innerhalb der 4diac-Umgebung, bei denen Übersichtlichkeit und standardisierte Adapter-Schnittstellen im Vordergrund stehen.

ALR_ADD_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALR_ADD_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ALR_ADD_2_UNGATED` ist ein generischer, arithmetischer Additionsbaustein für die 4diac-IDE, der der Norm IEC 61131-3 entspricht. Er dient dazu, zwei Eingangswerte mathematisch zu addieren. Die Besonderheit dieses Bausteins liegt in der Verwendung von unidirektionalen Adaptern (`ALR`), die sowohl Daten als auch die zugehörige Kontrolllogik kapseln, um eine saubere und modulare Signalverarbeitung zu ermöglichen.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten, dedizierten Ereignis-Eingänge. Die Ereignissteuerung wird über die Adapter abgewickelt.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten, dedizierten Ereignis-Ausgänge. Die Ereignissteuerung wird über die Adapter abgewickelt.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge definiert, da die Datenübergabe vollständig über die Adapter erfolgt.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge definiert, da das Ergebnis über den Ausgangs-Adapter bereitgestellt wird.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::ALR`): Der erste Summand der Additionsoperation.
- **IN2** (Typ: `adapter::types::unidirectional::ALR`): Der zweite Summand der Additionsoperation.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::ALR`): Das Ergebnis der Addition ($OUT = IN1 + IN2$).

Funktionsweise

Der `ALR_ADD_2_UNGATED` liest die Werte über die beiden Eingangs-Adapter `IN1` und `IN2` ein. Sobald ein entsprechendes Ereignis über die Adapter signalisiert, dass neue Daten vorliegen, führt der Baustein die Addition aus:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} \tag{2.51}$$

Das berechnete Ergebnis wird anschließend über den Ausgangs-Adapter `OUT` zusammen mit dem entsprechenden Ausgangsereignis an nachfolgende Bausteine weitergeleitet. Da es sich um einen generischen Funktionsbaustein (`GEN_ALR_ADD`) handelt, kann er flexibel mit verschiedenen numerischen Datentypen arbeiten, die durch das ALR-Adapter-Interface unterstützt werden.

Technische Besonderheiten

- **Generischer Typ (`GEN_ALR_ADD`):** Der Baustein ist datentypunabhängig implementiert und passt sich den im ALR-Adapter definierten Datentypen an.
- **Adapter-Kapselung:** Durch die Verwendung unidirektionaler Adapter wird die Anzahl der sichtbaren Verbindungen im Control-Flow-Diagramm der 4diac-IDE drastisch reduziert, was die Übersichtlichkeit und Wartbarkeit des Gesamtsystems erhöht.
- **Konformität:** Entspricht den mathematischen Standards nach IEC 61131-3 für Standard-Arithmetikfunktionen.

Zustandsübersicht

Da die Logik dieses Bausteins rein daten- und ereignisgesteuert über die Adapter abläuft, besitzt der Baustein keine komplexe interne Zustandsmaschine (ECC). Er reagiert unmittelbar auf eingehende Datenänderungen/Ereignisse an den Sockets `IN1` und `IN2` und leitet das Ergebnis an den Plug `OUT` weiter.

Anwendungsszenarien

- **Signalzusammenführung:** Addition von zwei analogen Sensorwerten (z. B. Ermittlung eines Gesamtvolumenstroms aus zwei einzelnen Durchflussmessern).
- **Sollwert-Offset-Berechnung:** Beaufschlagung eines Basis-Sollwerts mit einem Korrekturoffset in verfahrenstechnischen Anlagen.
- **Kompakte Programmierung:** Verwendung in komplexen Regelungsschleifen, bei denen ein übersichtliches Routing mittels Adaptern bevorzugt wird.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-`ADD`-Funktionsbaustein (der separate Event-Eingänge wie `REQ`, Event-Ausgänge wie `CNF` sowie klassische Daten-Pins besitzt) verzichtet `ALR_ADD_2_UNGATED` vollständig auf diese klassischen Pins. Er ersetzt sie durch standardisierte `ALR`-Adapter. Dies vereinfacht die Verdrahtung in der 4diac-IDE erheblich, da Event und Daten in einer einzigen Verbindung gebündelt sind.

- `ALR_ADD_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

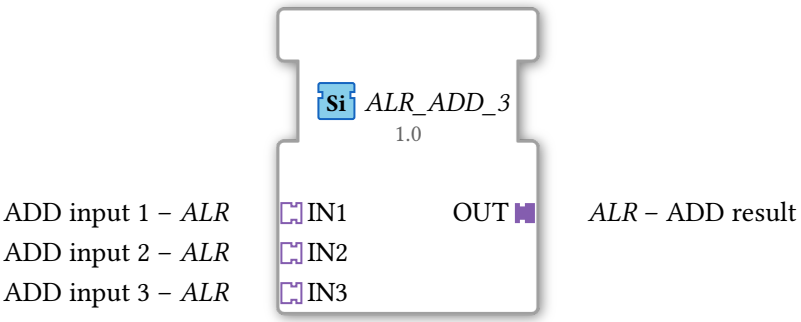
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `ALR_ADD_2_UNGATED` ist eine moderne und hocheffiziente Variante des klassischen Additionsbausteins. Er eignet sich hervorragend für serviceorientierte Architekturen und verteilte Steuerungssysteme innerhalb der 4diac-Umgebung, bei denen Übersichtlichkeit und standardisierte Adapter-Schnittstellen im Vordergrund stehen.

ALR_ADD_3



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ALR_ADD_3` ist ein generischer Arithmetik-Baustein für die 4diac-IDE, der zur Addition von drei Eingangswerten dient. Er entspricht den Standard-Arithmetikfunktionen nach IEC 61131-3, nutzt jedoch ein adapterbasiertes Schnittstellendesign. Anstatt mit einzelnen Daten- und Ereignispins arbeitet dieser Baustein mit unidirektionalen Adaptern, um eine saubere Kapselung und eine übersichtlichere Verdrahtung innerhalb der Anwendung zu ermöglichen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die ereignisbasierte Steuerung wird intern über die Adapter abgewickelt.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Ereignisweiterleitung erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Schnittstellen)

- **IN1** (Typ: `adapter::types::unidirectional::ALR`): Erster Summand der Additionsoperation.
- **IN2** (Typ: `adapter::types::unidirectional::ALR`): Zweiter Summand der Additionsoperation.
- **IN3** (Typ: `adapter::types::unidirectional::ALR`): Dritter Summand der Additionsoperation.

Plugs (Ausgangs-Schnittstellen)

- **OUT** (Typ: `adapter::types::unidirectional::ALR`): Ausgangs-Adapter, der das Ergebnis der Addition (`IN1 + IN2 + IN3`) sowie das zugehörige Trigger-Ereignis bereitstellt.

Funktionsweise

Der Baustein addiert die über die drei Eingangs-Adapter (`IN1`, `IN2` und `IN3`) bereitgestellten Werte. Sobald an einem der Eingangs-Adapter ein neues Ereignis oder ein neuer Wert registriert wird, führt der Funktionsbaustein die Berechnung nach folgendem Schema aus:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} \tag{2.52}$$

Das Ergebnis wird unmittelbar an den Ausgangs-Adapter `OUT` übergeben und ein entsprechendes Aktualisierungsereignis wird ausgelöst, um nachfolgende Bausteine im Datenfluss zu triggern.

Technische Besonderheiten

- **Generischer Baustein:** Durch das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `'GEN_ALR_ADD'` ist der Funktionsbaustein datentypunabhängig (generisch) aufgebaut. Er kann somit flexibel mit verschiedenen numerischen Datentypen (z. B. INT, REAL, LREAL) arbeiten, sofern der verwendete Adaptertyp `ALR` diese unterstützt.
- **Adapter-Kopplung:** Die Verwendung von Adaptern anstelle von klassischen Daten- und Ereignispins reduziert die Anzahl der sichtbaren Verbindungslinien im Application Editor von 4diac drastisch. Dies sorgt für eine wesentlich bessere Lesbarkeit komplexer Steuerungsdiagramme.

Zustandsübersicht

Da es sich bei `ALR_ADD_3` um einen mathematischen, datenflussorientierten Funktionsbaustein handelt, besitzt er keinen komplexen internen Zustandsautomaten (ECC). Die Ausführung erfolgt rein reaktiv (ereignisgesteuert über die Eingangs-Adapter).

Anwendungsszenarien

- **Messwert-Summierung:** Addition von drei analogen Teilströmen zur Ermittlung des Gesamtstroms in einer Energieverteilung.
- **Mischungsrechnungen:** Zusammenführung von drei Durchflussmengen in einer verfahrenstechnischen Anlage zur Berechnung des Gesamt-Zuflusses.
- **Sollwert-Generierung:** Berechnung eines Gesamtsollwerts aus einem Basiswert, einem Korrekturwert und einem Offset.

Vergleich mit ähnlichen Bausteinen

- **Standard `ADD` (IEC 61131-3):** Der klassische `ADD`-Baustein verwendet separate Pins für Daten und Ereignisse (REQ/CNF). `ALR_ADD_3` hingegen bündelt diese Signale in Adaptern, was die Modularität und Wiederverwendbarkeit im projektübergreifenden Design erhöht.
- **`ALR_ADD_2`:** Ein ähnlicher adapterbasierter Baustein, der jedoch nur zwei Eingänge unterstützt. `ALR_ADD_3` spart bei der Addition von exakt drei Werten Kaskadierungen und somit zusätzlichen Speicher- und Berechnungsaufwand.
- `ALR_ADD_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

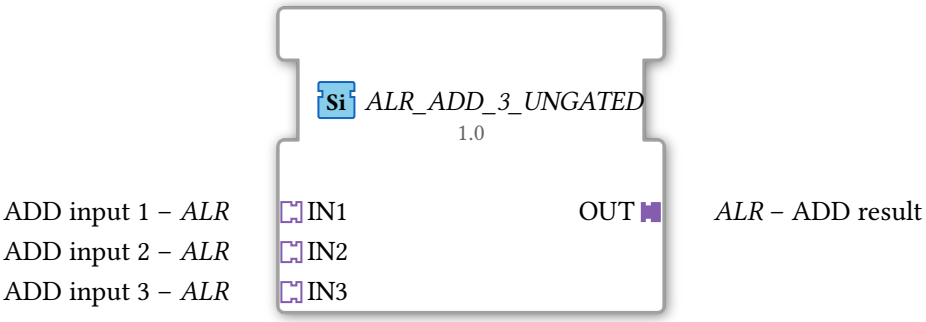
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `ALR_ADD_3` ist ein flexibler und performanter Baustein zur Addition von drei Werten in 4diac-Umgebungen. Durch sein generisches Design und die konsequente Nutzung von unidirektionalen Adaptern trägt er maßgeblich zur Übersichtlichkeit, Strukturierung und Wartbarkeit moderner, verteilter Steuerungsprogramme bei.

ALR_ADD_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALR_ADD_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ALR_ADD_3_UNGATED` ist ein generischer Arithmetik-Baustein für die 4diac-IDE, der zur Addition von drei Eingangswerten dient. Er entspricht den Standard-Arithmetikfunktionen nach IEC 61131-3, nutzt jedoch ein adapterbasiertes Schnittstellendesign. Anstatt mit einzelnen Daten- und Ereignispins arbeitet dieser Baustein mit unidirektionalen Adaptern, um eine saubere Kapselung und eine übersichtlichere Verdrahtung innerhalb der Anwendung zu ermöglichen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die ereignisbasierte Steuerung wird intern über die Adapter abgewickelt.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Ereignisweiterleitung erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Schnittstellen)

- **IN1** (Typ: `adapter::types::unidirectional::ALR`): Erster Summand der Additionsoperation.
- **IN2** (Typ: `adapter::types::unidirectional::ALR`): Zweiter Summand der Additionsoperation.
- **IN3** (Typ: `adapter::types::unidirectional::ALR`): Dritter Summand der Additionsoperation.

Plugs (Ausgangs-Schnittstellen)

- **OUT** (Typ: `adapter::types::unidirectional::ALR`): Ausgangs-Adapter, der das Ergebnis der Addition (`IN1 + IN2 + IN3`) sowie das zugehörige Trigger-Ereignis bereitstellt.

Funktionsweise

Der Baustein addiert die über die drei Eingangs-Adapter (`IN1`, `IN2` und `IN3`) bereitgestellten Werte. Sobald an einem der Eingangs-Adapter ein neues Ereignis oder ein neuer Wert registriert wird, führt der Funktionsbaustein die Berechnung nach folgendem Schema aus:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} \tag{2.53}$$

Das Ergebnis wird unmittelbar an den Ausgangs-Adapter `OUT` übergeben und ein entsprechendes Aktualisierungsereignis wird ausgelöst, um nachfolgende Bausteine im Datenfluss zu triggern.

Technische Besonderheiten

- **Generischer Baustein:** Durch das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `'GEN_ALR_ADD'` ist der Funktionsbaustein datentypunabhängig (generisch) aufgebaut. Er kann somit flexibel mit verschiedenen numerischen Datentypen (z. B. INT, REAL, LREAL) arbeiten, sofern der verwendete Adaptertyp `ALR` diese unterstützt.
- **Adapter-Kopplung:** Die Verwendung von Adaptern anstelle von klassischen Daten- und Ereignispins reduziert die Anzahl der sichtbaren Verbindungslinien im Application Editor von 4diac drastisch. Dies sorgt für eine wesentlich bessere Lesbarkeit komplexer Steuerungsdiagramme.

Zustandsübersicht

Da es sich bei `ALR_ADD_3_UNGATED` um einen mathematischen, datenflussorientierten Funktionsbaustein handelt, besitzt er keinen komplexen internen Zustandsautomaten (ECC). Die Ausführung erfolgt rein reaktiv (ereignisgesteuert über die Eingangs-Adapter).

Anwendungsszenarien

- **Messwert-Summierung:** Addition von drei analogen Teilströmen zur Ermittlung des Gesamtstroms in einer Energieverteilung.
- **Mischungsrechnungen:** Zusammenführung von drei Durchflussmengen in einer verfahrenstechnischen Anlage zur Berechnung des Gesamt-Zuflusses.
- **Sollwert-Generierung:** Berechnung eines Gesamtsollwerts aus einem Basiswert, einem Korrekturwert und einem Offset.

Vergleich mit ähnlichen Bausteinen

- **Standard `ADD` (IEC 61131-3):** Der klassische `ADD`-Baustein verwendet separate Pins für Daten und Ereignisse (REQ/CNF). `ALR_ADD_3_UNGATED` hingegen bündelt diese Signale in Adaptern, was die Modularität und Wiederverwendbarkeit im projektübergreifenden Design erhöht.
- **`ALR_ADD_2`:** Ein ähnlicher adapterbasierter Baustein, der jedoch nur zwei Eingänge unterstützt. `ALR_ADD_3_UNGATED` spart bei der Addition von exakt drei Werten Kaskadierungen und somit zusätzlichen Speicher- und Berechnungsaufwand.

- `ALR_ADD_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

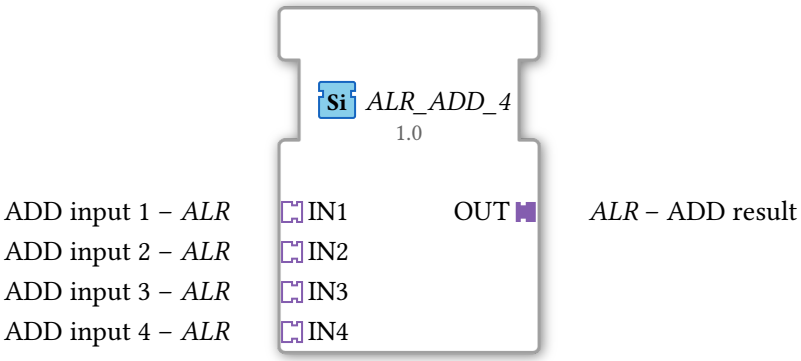
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `ALR_ADD_3_UNGATED` ist ein flexibler und performanter Baustein zur Addition von drei Werten in 4diac-Umgebungen. Durch sein generisches Design und die konsequente Nutzung von unidirektionalen Adaptern trägt er maßgeblich zur Übersichtlichkeit, Strukturierung und Wartbarkeit moderner, verteilter Steuerungsprogramme bei.

ALR_ADD_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein (FB) **ALR_ADD_4** ist ein generischer arithmetischer Baustein zur Durchführung einer Additionsoperation. Er ist gemäß dem Standard IEC 61131-3 als Standard-Arithmetikfunktion klassifiziert. Der Baustein ermöglicht es, bis zu vier Eingangswerte, die über unidirektionale Adapter vom Typ `ALR` bereitgestellt werden, zu addieren und das Ergebnis an einen entsprechenden Ausgangsadapter zu übergeben.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die Synchronisation und Ereignissteuerung erfolgt implizit über die verwendeten Adapter.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Weiterleitung von Ereignissen erfolgt implizit über den Ausgangsadapter.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden. Die Datenübergabe erfolgt vollständig über die Adapter-Schnittstellen.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden. Die Datenübergabe erfolgt vollständig über die Adapter-Schnittstelle.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::ALR`): Erster Eingangswert (Summand 1) für die Addition.

- **IN2** (Typ: `adapter::types::unidirectional::ALR`): Zweiter Eingangswert (Summand 2) für die Addition.
- **IN3** (Typ: `adapter::types::unidirectional::ALR`): Dritter Eingangswert (Summand 3) für die Addition.
- **IN4** (Typ: `adapter::types::unidirectional::ALR`): Vierter Eingangswert (Summand 4) für die Addition.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::ALR`): Das berechnete Ergebnis der Addition (Summe).

Funktionsweise

Der Baustein `ALR_ADD_4` arbeitet intern als generischer Addierer (`GEN_ALR_ADD`). Sobald sich die Werte an den Eingangs-Adaptern `IN1` bis `IN4` ändern, berechnet der Baustein die Summe der anliegenden Werte:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} + \mathrm{IN4} \tag{2.54}$$

Das Ergebnis wird unmittelbar an den Ausgangs-Plug `OUT` übergeben. Da es sich um eine generische Implementierung handelt, hängt der konkrete Datentyp von der Spezifikation des verwendeten `ALR`-Adaptertyps ab.

Technische Besonderheiten

- **Generische Klasse:** Der Baustein ist über das Attribut `GenericClassName` mit dem Wert `'GEN_ALR_ADD'` verknüpft, was ihm eine hohe Flexibilität bei der Verarbeitung verschiedener kompatibler Datentypen innerhalb der Adapterstrukturen verleiht.
- **Adapterbasierte Kopplung:** Durch die ausschließliche Nutzung von Adaptern wird die Anzahl der Verbindungslinien in der 4diac-IDE drastisch reduziert, was zu übersichtlicheren Applikationsdiagrammen führt.

Zustandsübersicht

Da dieser Funktionsbaustein eine rein kombinatorische, mathematische Operation abbildet, besitzt er keine interne Zustandsmaschine (Execution Control Chart, ECC). Der Ausgang folgt direkt den Änderungen der Eingänge.

Anwendungsszenarien

- **Messwert-Aggregation:** Aufsummierung von bis zu vier Energie-, Durchfluss- oder Leistungswerten, die über standardisierte unidirektionale Adapter übertragen werden.
- **Signalzusammenführung:** Strukturierte Bündelung von Sensorsignalen in komplexen Steuerungsarchitekturen vor der weiteren Datenverarbeitung.

Vergleich mit ähnlichen Bausteinen

- **Standard-ADD-Baustein:** Ein klassischer IEC 61131-3 `ADD`-Baustein nutzt direkte Dateneingänge (z. B. `REAL`, `INT`) und benötigt oft explizite Event-Verbindungen (`REQ` / `CNF`). `ALR_ADD_4` kapselt diese Daten und Events in Adaptern, was die Projektierung vereinfacht.
- **Zwei-Wege-Addierer (z.B. ALR_ADD_2):** Bietet nur zwei Eingänge. `ALR_ADD_4` spart bei der Summierung von drei oder vier Werten das Kaskadieren mehrerer kleinerer Addierer-Bausteine ein.
- `ALR_ADD_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

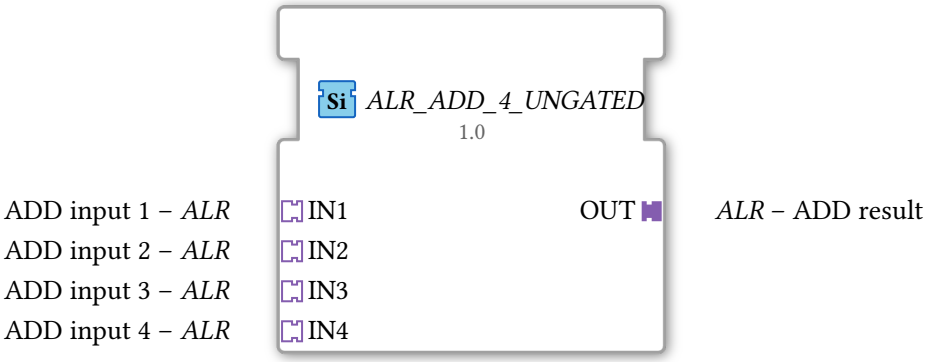
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `ALR_ADD_4`-Baustein ist eine effiziente und saubere Lösung für arithmetische Additionen in komponenten- und adapterbasierten Systemarchitekturen innerhalb von 4diac. Er fördert die Wiederverwendbarkeit und sorgt für ein aufgeräumtes Systemdesign.

ALR_ADD_4_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von ALR_ADD_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein (FB) **ALR_ADD_4_UNGATED** ist ein generischer arithmetischer Baustein zur Durchführung einer Additionsoperation. Er ist gemäß dem Standard IEC 61131-3 als Standard-Arithmetikfunktion klassifiziert. Der Baustein ermöglicht es, bis zu vier Eingangswerte, die über unidirektionale Adapter vom Typ `ALR` bereitgestellt werden, zu addieren und das Ergebnis an einen entsprechenden Ausgangsadapter zu übergeben.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die Synchronisation und Ereignissteuerung erfolgt implizit über die verwendeten Adapter.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Weiterleitung von Ereignissen erfolgt implizit über den Ausgangsadapter.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden. Die Datenübergabe erfolgt vollständig über die Adapter-Schnittstellen.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden. Die Datenübergabe erfolgt vollständig über die Adapter-Schnittstelle.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::ALR`): Erster Eingangswert (Summand 1) für die Addition.
- **IN2** (Typ: `adapter::types::unidirectional::ALR`): Zweiter Eingangswert (Summand 2) für die Addition.
- **IN3** (Typ: `adapter::types::unidirectional::ALR`): Dritter Eingangswert (Summand 3) für die Addition.

- **IN4** (Typ: `adapter::types::unidirectional::ALR`): Vierter Eingangswert (Summand 4) für die Addition.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::ALR`): Das berechnete Ergebnis der Addition (Summe).

Funktionsweise

Der Baustein `ALR_ADD_4_UNGATED` arbeitet intern als generischer Addierer (`GEN_ALR_ADD`). Sobald sich die Werte an den Eingangs-Adaptern `IN1` bis `IN4` ändern, berechnet der Baustein die Summe der anliegenden Werte:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} + \mathrm{IN4} \tag{2.55}$$

Das Ergebnis wird unmittelbar an den Ausgangs-Plug `OUT` übergeben. Da es sich um eine generische Implementierung handelt, hängt der konkrete Datentyp von der Spezifikation des verwendeten `ALR`-Adaptertyps ab.

Technische Besonderheiten

- **Generische Klasse:** Der Baustein ist über das Attribut `GenericClassName` mit dem Wert `'GEN_ALR_ADD'` verknüpft, was ihm eine hohe Flexibilität bei der Verarbeitung verschiedener kompatibler Datentypen innerhalb der Adapterstrukturen verleiht.
- **Adapterbasierte Kopplung:** Durch die ausschließliche Nutzung von Adaptern wird die Anzahl der Verbindungslinien in der 4diac-IDE drastisch reduziert, was zu übersichtlicheren Applikationsdiagrammen führt.

Zustandsübersicht

Da dieser Funktionsbaustein eine rein kombinatorische, mathematische Operation abbildet, besitzt er keine interne Zustandsmaschine (Execution Control Chart, ECC). Der Ausgang folgt direkt den Änderungen der Eingänge.

Anwendungsszenarien

- **Messwert-Aggregation:** Aufsummierung von bis zu vier Energie-, Durchfluss- oder Leistungswerten, die über standardisierte unidirektionale Adapter übertragen werden.
- **Signalzusammenführung:** Strukturierte Bündelung von Sensorsignalen in komplexen Steuerungsarchitekturen vor der weiteren Datenverarbeitung.

Vergleich mit ähnlichen Bausteinen

- **Standard-ADD-Baustein:** Ein klassischer IEC 61131-3 `ADD`-Baustein nutzt direkte Dateneingänge (z. B. `REAL`, `INT`) und benötigt oft explizite Event-Verbindungen (`REQ` / `CNF`). `ALR_ADD_4_UNGATED` kapselt diese Daten und Events in Adaptern, was die Projektierung vereinfacht.
- **Zwei-Wege-Addierer (z.B. ALR_ADD_2):** Bietet nur zwei Eingänge. `ALR_ADD_4_UNGATED` spart bei der Summierung von drei oder vier Werten das Kaskadieren mehrerer kleinerer Addierer-Bausteine ein.
- **`ALR_ADD_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

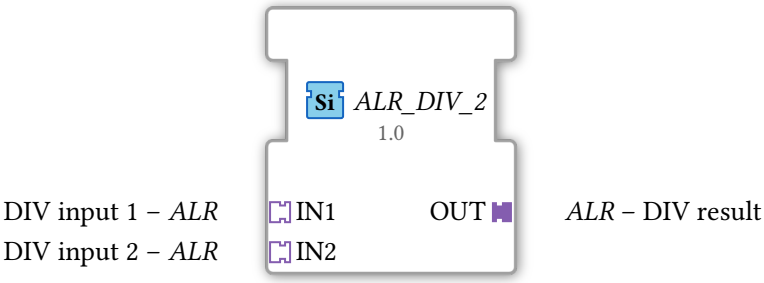
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `ALR_ADD_4_UNGATED`-Baustein ist eine effiziente und saubere Lösung für arithmetische Additionen in komponenten- und adapterbasierten Systemarchitekturen innerhalb von 4diac. Er fördert die Wiederverwendbarkeit und sorgt für ein aufgeräumtes Systemdesign.

ALR_DIV_2



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ALR_DIV_2` ist ein generischer Funktionsbaustein (Generic FB) zur Durchführung von arithmetischen Divisionen. Er ist im Package `adapter::iec61131::arithmetic` definiert und basiert auf der generischen Klasse `GEN_ALR_DIV`.

Im Gegensatz zu klassischen arithmetischen Bausteinen, die mit direkten Daten- und Ereigniseingängen arbeiten, nutzt dieser Baustein Adapter vom Typ `ALR` (unidirektional). Dies ermöglicht eine stark vereinfachte und strukturierte Verdrahtung innerhalb von 4diac-Anwendungen, da Signale und deren zugehörige Kontrollflüsse in Adaptern gebündelt werden.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Baustein besitzt keine direkten Ereignis-Eingänge. Die Steuerung und Triggerung erfolgt implizit über die verbundenen Adapter.

Ereignis-Ausgänge

Dieser Baustein besitzt keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Dieser Baustein besitzt keine direkten Daten-Eingänge.

Daten-Ausgänge

Dieser Baustein besitzt keine direkten Daten-Ausgänge.

Adapter

Typ	Name	Adapter-Typ	Beschreibung
Socket (Eingang)	`IN1`	`adapter::types::unidirectional::ALR`	Erster Operand der Division (Dividend)
Socket (Eingang)	`IN2`	`adapter::types::unidirectional::ALR`	Zweiter Operand der Division (Divisor)
Plug (Ausgang)	`OUT`	`adapter::types::unidirectional::ALR`	Ergebnis der Division (Quotient)

Funktionsweise

Der Baustein `ALR_DIV_2` führt eine mathematische Division der über die Adapter bereitgestellten Werte aus. Die Berechnung folgt der Formel:

$$\mathrm{OUT} = \frac{\mathrm{IN1}}{\mathrm{IN2}} \tag{2.56}$$

Da es sich um einen generischen Funktionsbaustein handelt, wird der genaue Datentyp der mathematischen Operation durch den konkret verwendeten Datentyp des zugrundeliegenden `ALR`-Adapters zur

Laufzeit bzw. bei der Instanziierung bestimmt. Die Datenübertragung erfolgt unidirektional von den Sockets (`IN1` und `IN2`) hin zum Plug (`OUT`).

Technische Besonderheiten

- **Generischer Baustein:** Durch die Eigenschaft `GenericClassName = "GEN_ALR_DIV"` ist der Baustein flexibel für verschiedene Datentypen einsetzbar, sofern diese vom Adaptertyp unterstützt werden.
- **Adapter-Kopplung:** Durch die Verwendung von Adaptern anstelle von einzelnen Pins wird der grafische Verdrahtungsaufwand in der 4diac-IDE minimiert und die Übersichtlichkeit im Applikationseditor erhöht.
- **Division durch Null:** Bei der Implementierung und Anwendung ist darauf zu achten, dass der Wert am Adapter `IN2` ungleich Null ist, um Laufzeitfehler oder ein undefiniertes Systemverhalten zu vermeiden.

Zustandsübersicht

Der Baustein besitzt kein komplexes internes Zustandsdiagramm (ECC). Er arbeitet als rein funktionaler Verarbeitungsbaustein, der bei Wertänderungen an den Eingangs-Adaptern (`IN1` oder `IN2`) das Ergebnis berechnet und über den Ausgangs-Adapter (`OUT`) weiterleitet.

Anwendungsszenarien

- **Signal-Skalierung:** Division von analogen Messwerten durch konstante Faktoren zur Einheitenumrechnung innerhalb einer Adapter-basierten Signalverarbeitungskette.
- **Mittelwertbildung und Verhältnissteuerung:** Berechnung von Verhältnissen (z. B. Luft-Brennstoff-Verhältnis) in Regelungssystemen, bei denen die Signalpfade modular über Adapter realisiert sind.
- **Standardisierung:** Verwendung in komplexen Steuerungsarchitekturen zur Reduzierung von Verbindungslinien.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zum Standard-IEC-61131-3-Kompaktbaustein `DIV` besitzt der `ALR_DIV_2` keine direkten Datenpins (wie `IN1`, `IN2` als `ANY_NUM`). Während der Standard-`DIV`-Baustein für einfache, punktuelle Berechnungen optimiert ist, eignet sich der `ALR_DIV_2` primär für serviceorientierte Architekturen und Anwendungen, die konsequent auf eine lose Kopplung mittels Adapterstrukturen setzen.

- `ALR_DIV_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

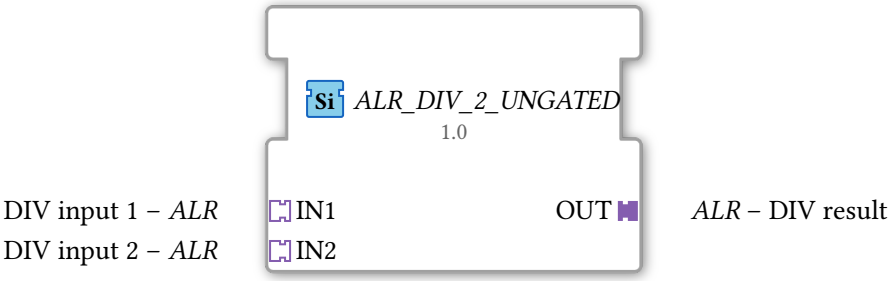
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `ALR_DIV_2` ist ein spezialisierter, aber dennoch flexibler Divisionsbaustein für die 4diac-IDE. Er eignet sich hervorragend für anspruchsvolle Projekte, die Wert auf eine saubere Kapselung von Datenflüssen mithilfe des unidirektionalen `ALR`-Adapters legen.

ALR_DIV_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALR_DIV_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ALR_DIV_2_UNGATED` ist ein generischer Funktionsbaustein (Generic FB) zur Durchführung von arithmetischen Divisionen. Er ist im Package `adapter::iec61131::arithmetic` definiert und basiert auf der generischen Klasse `GEN_ALR_DIV`.

Im Gegensatz zu klassischen arithmetischen Bausteinen, die mit direkten Daten- und Ereigniseingängen arbeiten, nutzt dieser Baustein Adapter vom Typ `ALR` (unidirektional). Dies ermöglicht eine stark vereinfachte und strukturierte Verdrahtung innerhalb von 4diac-Anwendungen, da Signale und deren zugehörige Kontrollflüsse in Adaptern gebündelt werden.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Baustein besitzt keine direkten Ereignis-Eingänge. Die Steuerung und Triggerung erfolgt implizit über die verbundenen Adapter.

Ereignis-Ausgänge

Dieser Baustein besitzt keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Dieser Baustein besitzt keine direkten Daten-Eingänge.

Daten-Ausgänge

Dieser Baustein besitzt keine direkten Daten-Ausgänge.

Adapter

Typ	Name	Adapter-Typ	Beschreibung
Socket (Eingang)	`IN1`	`adapter::types::unidirectional::ALR`	Erster Operand der Division (Dividend)
Socket (Eingang)	`IN2`	`adapter::types::unidirectional::ALR`	Zweiter Operand der Division (Divisor)
Plug (Ausgang)	`OUT`	`adapter::types::unidirectional::ALR`	Ergebnis der Division (Quotient)

Funktionsweise

Der Baustein `ALR_DIV_2_UNGATED` führt eine mathematische Division der über die Adapter bereitgestellten Werte aus. Die Berechnung folgt der Formel:

$$\mathrm{OUT} = \frac{\mathrm{IN1}}{\mathrm{IN2}} \tag{2.57}$$

Da es sich um einen generischen Funktionsbaustein handelt, wird der genaue Datentyp der mathematischen Operation durch den konkret verwendeten Datentyp des zugrundeliegenden `ALR`-Adapters zur Laufzeit bzw. bei der Instanziierung bestimmt. Die Datenübertragung erfolgt unidirektional von den Sockets (`IN1` und `IN2`) hin zum Plug (`OUT`).

Technische Besonderheiten

- **Generischer Baustein:** Durch die Eigenschaft `GenericClassName = "GEN_ALR_DIV"` ist der Baustein flexibel für verschiedene Datentypen einsetzbar, sofern diese vom Adaptertyp unterstützt werden.
- **Adapter-Kopplung:** Durch die Verwendung von Adaptern anstelle von einzelnen Pins wird der grafische Verdrahtungsaufwand in der 4diac-IDE minimiert und die Übersichtlichkeit im Applikationseditor erhöht.
- **Division durch Null:** Bei der Implementierung und Anwendung ist darauf zu achten, dass der Wert am Adapter `IN2` ungleich Null ist, um Laufzeitfehler oder ein undefiniertes Systemverhalten zu vermeiden.

Zustandsübersicht

Der Baustein besitzt kein komplexes internes Zustandsdiagramm (ECC). Er arbeitet als rein funktionaler Verarbeitungsbaustein, der bei Wertänderungen an den Eingangs-Adaptern (`IN1` oder `IN2`) das Ergebnis berechnet und über den Ausgangs-Adapter (`OUT`) weiterleitet.

Anwendungsszenarien

- **Signal-Skalierung:** Division von analogen Messwerten durch konstante Faktoren zur Einheitenumrechnung innerhalb einer Adapter-basierten Signalverarbeitungskette.
- **Mittelwertbildung und Verhältnissteuerung:** Berechnung von Verhältnissen (z. B. Luft-Brennstoff-Verhältnis) in Regelungssystemen, bei denen die Signalpfade modular über Adapter realisiert sind.
- **Standardisierung:** Verwendung in komplexen Steuerungsarchitekturen zur Reduzierung von Verbindungslinien.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zum Standard-IEC-61131-3-Kompaktbaustein `DIV` besitzt der `ALR_DIV_2_UNGATED` keine direkten Datenpins (wie `IN1`, `IN2` als `ANY_NUM`). Während der Standard-`DIV`-Baustein für einfache, punktuelle Berechnungen optimiert ist, eignet sich der `ALR_DIV_2_UNGATED` primär für serviceorientierte Architekturen und Anwendungen, die konsequent auf eine lose Kopplung mittels Adapterstrukturen setzen.

- `ALR_DIV_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- `ALR_DIV_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

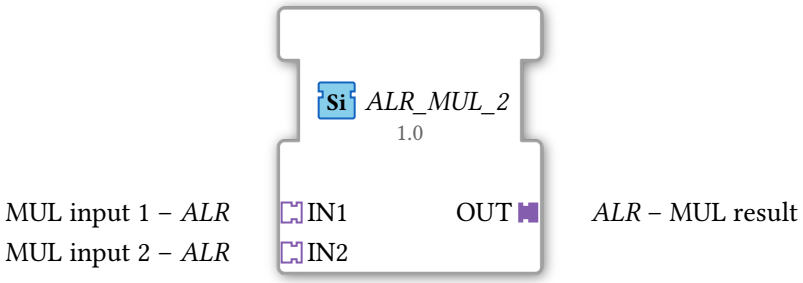
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `ALR_DIV_2_UNGATED` ist ein spezialisierter, aber dennoch flexibler Divisionsbaustein für die 4diac-IDE. Er eignet sich hervorragend für anspruchsvolle Projekte, die Wert auf eine saubere Kapselung von Datenflüssen mithilfe des unidirektionalen `ALR`-Adapters legen.

ALR_MUL_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ALR_MUL_2` ist ein generischer, arithmetischer Baustein, der für die Multiplikation zweier Werte konzipiert ist. Er basiert auf der IEC 61499-Architektur und nutzt Adapter-Schnittstellen

zur Kapselung und Übertragung von Daten und Ereignissen. Dadurch ermöglicht er eine saubere Strukturierung und verringert den Verdrahtungsaufwand in 4diac-IDE-Anwendungen, da Daten und zugehörige Kontrollflüsse in gemeinsamen Verbindungen (Adaptern) gebündelt werden.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Eingänge. Die Steuerung des Kontrollflusses erfolgt implizit über die verwendeten Adapter.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Ausgänge. Die Weiterleitung von Ereignissen erfolgt implizit über den Ausgangs-Adapter.

Daten-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Eingänge. Die Eingangsdaten werden über die Adapter-Schnittstellen bereitgestellt.

Daten-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Ausgänge. Das Berechnungsergebnis wird über den Ausgangs-Adapter bereitgestellt.

Adapter

Der Baustein kommuniziert ausschließlich über Adapter-Verbindungen des Typs `ALR` (unidirektional):

- **IN1 (Socket / Buchse)**:
  - **Typ**: `adapter::types::unidirectional::ALR`
  - **Beschreibung**: Erster Eingang (Multiplikand 1) für die Multiplikation.
- **IN2 (Socket / Buchse)**:
  - **Typ**: `adapter::types::unidirectional::ALR`
  - **Beschreibung**: Zweiter Eingang (Multiplikand 2) für die Multiplikation.
- **OUT (Plug / Stecker)**:
  - **Typ**: `adapter::types::unidirectional::ALR`
  - **Beschreibung**: Ausgang für das berechnete Multiplikationsergebnis.

Funktionsweise

Der Baustein `ALR_MUL_2` führt eine mathematische Multiplikation nach folgendem Prinzip aus:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \tag{2.58}$$

Sobald an den Eingangs-Adaptern `IN1` und/oder `IN2` ein neues Ereignis signalisiert wird, das auf die Aktualisierung der Daten hinweist, nimmt der Baustein die entsprechenden Werte entgegen, multipliziert sie und gibt das Ergebnis sowie ein entsprechendes Aktualisierungsereignis über den Ausgangs-Adapter `OUT` weiter.

Da es sich um einen generischen Baustein (`GEN_ALR_MUL`) handelt, passt sich die zugrundeliegende Berechnung an den im Adapter definierten Datentyp an.

Technische Besonderheiten

- **Generische Implementierung**: Der Baustein ist als `GEN_ALR_MUL` klassifiziert. Dies erlaubt eine flexible Nutzung mit verschiedenen numerischen Datentypen, die durch das Adapterprofil `ALR` definiert sind.
- **Kapselung durch Adapter**: Durch die Verwendung von unidirektionalen Adaptern wird der Kontrollfluss (Events) direkt mit dem Datenfluss (Values) verknüpft. Dies verhindert Synchronisationsfehler zwischen Daten und Ereignissen.
- **Compiler-Paket**: Der Baustein ist im Paket `adapter::iec61131::arithmetic` deklariert und nutzt die Kernbibliothek `eclipse4diac::core::GenericClassName`.

Zustandsübersicht

Der Baustein verhält sich wie ein klassischer, zustandsloser (bzw. rein funktionaler) Baustein. Es gibt keine internen Zustände, die über einen Berechnungszyklus hinaus gespeichert werden:

1. **Wartezustand**: Baustein wartet auf ein Ereignis an `IN1` oder `IN2`.
2. **Berechnung**: Bei Eintreffen eines Ereignisses werden die Werte aus den Adaptern gelesen und multipliziert.
3. **Ausgabe**: Das Ergebnis wird an `OUT` angelegt und das Ausgangsereignis getriggert. Der Baustein kehrt sofort in den Wartezustand zurück.

Anwendungsszenarien

- **Signalverarbeitung in verteilten Systemen**: Wenn Sensorwerte bereits als Adapterstrukturen vorliegen und skaliert oder miteinander verrechnet werden müssen (z. B. Berechnung von Leistung aus Strom und Spannung).
- **Modularer Steuerungsentwurf**: Reduzierung von Linienkreuzungen im Funktionsplan von 4diac-Anwendungen durch den Einsatz von Adaptern anstelle von getrennten Event- und Datenleitungen.
- **Kaskadierte Berechnungen**: Einfache Verkettung mehrerer mathematischer Operationen ohne zusätzlichen Overhead für die Event-Verdrahtung.

Vergleich mit ähnlichen Bausteinen

- **Standard `MUL` Baustein (IEC 61131-3)**: Der klassische `MUL`-Baustein nutzt separate Event-Eingänge (z.B. `REQ`) und Daten-Eingänge (z.B. `IN1`, `IN2` als REAL/INT). `ALR_MUL_2` vereinfacht dies, indem er diese Kanäle in Adaptern bündelt.
- **ALR_ADD_2 / ALR_SUB_2**: Diese Bausteine arbeiten nach demselben Adapter-Prinzip, führen jedoch Additionen oder Subtraktionen anstelle einer Multiplikation aus.

- `ALR_MUL_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

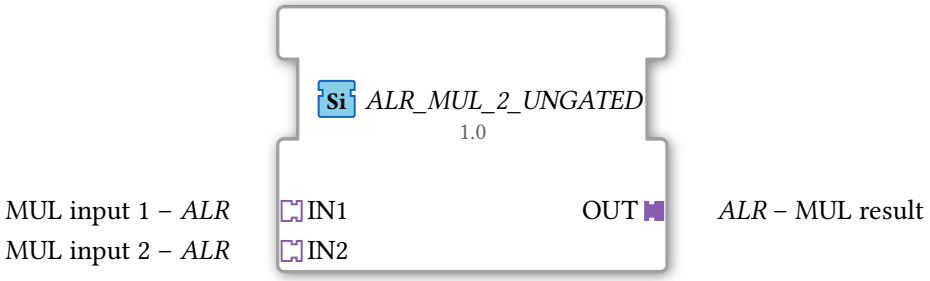
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

`ALR_MUL_2` ist ein spezialisierter, hocheffizienter Funktionsbaustein für moderne IEC 61499-Architekturen. Durch die konsequente Nutzung von Adaptern fördert er ein sauberes, objektorientiertes Softwaredesign in der industriellen Automatisierung und eignet sich hervorragend für komplexe, verteilte Steuerungsaufgaben.

ALR_MUL_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALR_MUL_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ALR_MUL_2_UNGATED` ist ein generischer, arithmetischer Baustein, der für die Multiplikation zweier Werte konzipiert ist. Er basiert auf der IEC 61499-Architektur und nutzt Adapter-Schnittstellen zur Kapselung und Übertragung von Daten und Ereignissen. Dadurch ermöglicht er eine saubere Strukturierung und verringert den Verdrahtungsaufwand in 4diac-IDE-Anwendungen, da Daten und zugehörige Kontrollflüsse in gemeinsamen Verbindungen (Adaptern) gebündelt werden.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Eingänge. Die Steuerung des Kontrollflusses erfolgt implizit über die verwendeten Adapter.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Ausgänge. Die Weiterleitung von Ereignissen erfolgt implizit über den Ausgangs-Adapter.

Daten-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Eingänge. Die Eingangsdaten werden über die Adapter-Schnittstellen bereitgestellt.

Daten-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Ausgänge. Das Berechnungsergebnis wird über den Ausgangs-Adapter bereitgestellt.

Adapter

Der Baustein kommuniziert ausschließlich über Adapter-Verbindungen des Typs `ALR` (unidirektional):

- **IN1 (Socket / Buchse)**:
  - **Typ**: `adapter::types::unidirectional::ALR`
  - **Beschreibung**: Erster Eingang (Multiplikand 1) für die Multiplikation.
- **IN2 (Socket / Buchse)**:
  - **Typ**: `adapter::types::unidirectional::ALR`
  - **Beschreibung**: Zweiter Eingang (Multiplikand 2) für die Multiplikation.
- **OUT (Plug / Stecker)**:
  - **Typ**: `adapter::types::unidirectional::ALR`
  - **Beschreibung**: Ausgang für das berechnete Multiplikationsergebnis.

Funktionsweise

Der Baustein `ALR_MUL_2_UNGATED` führt eine mathematische Multiplikation nach folgendem Prinzip aus:

$$\text{OUT} = \text{IN1} \times \text{IN2} \tag{2.59}$$

Sobald an den Eingangs-Adaptern `IN1` und/oder `IN2` ein neues Ereignis signalisiert wird, das auf die Aktualisierung der Daten hinweist, nimmt der Baustein die entsprechenden Werte entgegen, multipliziert sie und gibt das Ergebnis sowie ein entsprechendes Aktualisierungsereignis über den Ausgangs-Adapter `OUT` weiter.

Da es sich um einen generischen Baustein (GEN_ALR_MUL) handelt, passt sich die zugrundeliegende Berechnung an den im Adapter definierten Datentyp an.

Technische Besonderheiten

- **Generische Implementierung**: Der Baustein ist als GEN_ALR_MUL klassifiziert. Dies erlaubt eine flexible Nutzung mit verschiedenen numerischen Datentypen, die durch das Adapterprofil ALR definiert sind.
- **Kapselung durch Adapter**: Durch die Verwendung von unidirektionalen Adaptern wird der Kontrollfluss (Events) direkt mit dem Datenfluss (Values) verknüpft. Dies verhindert Synchronisationsfehler zwischen Daten und Ereignissen.
- **Compiler-Paket**: Der Baustein ist im Paket adapter::iec61131::arithmetic deklariert und nutzt die Kernbibliothek eclipse4diac::core::GenericClassName.

Zustandsübersicht

Der Baustein verhält sich wie ein klassischer, zustandsloser (bzw. rein funktionaler) Baustein. Es gibt keine internen Zustände, die über einen Berechnungszyklus hinaus gespeichert werden:

1. **Wartezustand**: Baustein wartet auf ein Ereignis an IN1 oder IN2.
2. **Berechnung**: Bei Eintreffen eines Ereignisses werden die Werte aus den Adaptern gelesen und multipliziert.
3. **Ausgabe**: Das Ergebnis wird an OUT angelegt und das Ausgangsereignis getriggert. Der Baustein kehrt sofort in den Wartezustand zurück.

Anwendungsszenarien

- **Signalverarbeitung in verteilten Systemen**: Wenn Sensorwerte bereits als Adapterstrukturen vorliegen und skaliert oder miteinander verrechnet werden müssen (z. B. Berechnung von Leistung aus Strom und Spannung).
- **Modularer Steuerungsentwurf**: Reduzierung von Linienkreuzungen im Funktionsplan von 4diac-Anwendungen durch den Einsatz von Adaptern anstelle von getrennten Event- und Datenleitungen.
- **Kaskadierte Berechnungen**: Einfache Verkettung mehrerer mathematischer Operationen ohne zusätzlichen Overhead für die Event-Verdrahtung.

Vergleich mit ähnlichen Bausteinen

- **Standard MUL Baustein (IEC 61131-3)**: Der klassische MUL-Baustein nutzt separate Event-Eingänge (z.B. REQ) und Daten-Eingänge (z.B. IN1, IN2 als REAL/INT). ALR_MUL_2_UNGATED vereinfacht dies, indem er diese Kanäle in Adaptern bündelt.
- **ALR_ADD_2 / ALR_SUB_2**: Diese Bausteine arbeiten nach demselben Adapter-Prinzip, führen jedoch Additionen oder Subtraktionen anstelle einer Multiplikation aus.

- ALR_MUL_2: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

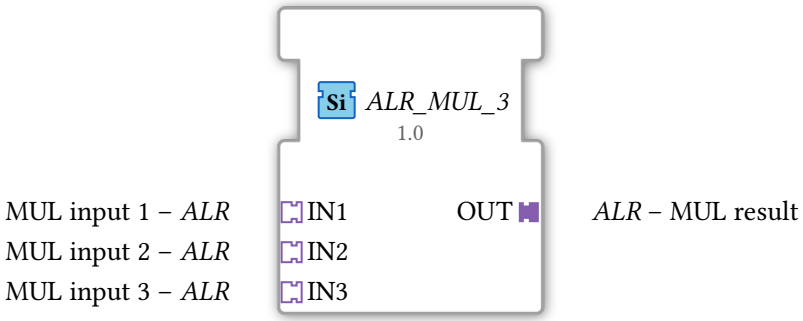
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

ALR_MUL_2_UNGATED ist ein spezialisierter, hocheffizienter Funktionsbaustein für moderne IEC 61499-Architekturen. Durch die konsequente Nutzung von Adaptern fördert er ein sauberes, objektorientiertes Softwaredesign in der industriellen Automatisierung und eignet sich hervorragend für komplexe, verteilte Steuerungsaufgaben.

ALR_MUL_3



(Symbolische Darstellung des Funktionsbausteins)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ALR_MUL_3` ist ein generischer Baustein aus der Bibliothek `adapter::iec61131::arithmetic`, der für die arithmetische Multiplikation von drei Eingangswerten entwickelt wurde. Anstelle von klassischen, diskreten Dateneingängen nutzt dieser Baustein unidirektionale Adapter des Typs `ALR` zur Kapselung und Übertragung der Daten und Steuerungsereignisse. Dies ermöglicht eine strukturierte, modulare und übersichtliche Verdrahtung innerhalb der 4diac-IDE.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die ereignisbasierte Steuerung wird vollständig über die eingebundenen Adapter abgewickelt.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die ereignisbasierte Steuerung wird vollständig über die eingebundenen Adapter abgewickelt.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden. Die Datenwerte sind in den Eingangs-Adaptern gekapselt.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden. Das Ergebnis ist im Ausgangs-Adapter gekapselt.

Adapter

Sockets (Eingangsschnittstellen)

- **IN1** (Typ: `adapter::types::unidirectional::ALR`): Der erste Faktor für die Multiplikation.
- **IN2** (Typ: `adapter::types::unidirectional::ALR`): Der zweite Faktor für die Multiplikation.
- **IN3** (Typ: `adapter::types::unidirectional::ALR`): Der dritte Faktor für die Multiplikation.

Plugs (Ausgangsschnittstellen)

- **OUT** (Typ: `adapter::types::unidirectional::ALR`): Das berechnete Produkt der drei Eingangswerte.

Funktionsweise

Der Baustein `ALR_MUL_3` realisiert die mathematische Multiplikation von drei Eingangsvariablen. Sobald an den Sockets neue Werte bzw. Triggerereignisse anliegen, führt der Baustein die Berechnung nach folgender Formel aus:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \tag{2.60}$$

Das Ergebnis der Berechnung sowie das zugehörige Aktualisierungsereignis werden über den Plug `OUT` ausgegeben.

Da es sich um einen generischen Baustein (Generic-Klasse: `GEN_ALR_MUL`) handelt, hängt der tatsächliche Datentyp (z. B. `REAL`, `LREAL`, `INT`) von der Definition des verwendeten `ALR`-Adapters ab.

Technische Besonderheiten

- **Generisches Design:** Durch die Zuweisung des Attributs `GenericClassName = 'GEN_ALR_MUL'` ist der Baustein flexibel für verschiedene Datentypen einsetzbar, sofern die verwendeten Adapter diese unterstützen.
- **Adapterbasierte Kopplung:** Die Verwendung von Adaptern statt loser Event-/Data-Verbindungen reduziert den Verdrahtungsaufwand (Routing) innerhalb der 4diac-Anwendung drastisch und erhöht die Übersichtlichkeit.
- **Unidirektionalität:** Die verwendeten `ALR`-Adapter sind als unidirektional definiert, was einen klaren und rückwirkungsfreien Datenfluss von den Quellen (Sockets) zur Senke (Plug) gewährleistet.

Zustandsübersicht

Der Baustein besitzt keine komplexe interne Zustandsmaschine (ECC). Er arbeitet als rein kombinatorischer/mathematischer Baustein, der direkt auf Ereignisse an den Eingangs-Adaptern reagiert und das Ergebnis unmittelbar an den Ausgang weiterleitet.

Anwendungsszenarien

- **Sensorskalierung und -korrektur:** Multiplikation eines Rohwerts (IN1) mit einem Kalibrierungsfaktor (IN2) und einem anwendungsbezogenen Gewichtungsfaktor (IN3).
- **Dreidimensionale Berechnungen:** Berechnung von Volumen oder Durchsätzen, bei denen drei physikalische Einflussgrößen miteinander multipliziert werden müssen.
- **Kaskadierte Verstärkungsglieder:** Berechnung von kombinierten Verstärkungen in Regelungskreisen.

Vergleich mit ähnlichen Bausteinen

- **Standard MUL-Baustein (IEC 61131-3):** Klassische Multiplizierer arbeiten mit direkten Elementardatentypen (z. B. `REAL`). `ALR_MUL_3` hingegen bündelt Daten und Ereignisse in Adaptern, was die Modularität und Wiederverwendbarkeit in verteilten Systemen nach IEC 61499 verbessert.
- **ALR_MUL_2 (2-fach Multiplizierer):** Für die Multiplikation von nur zwei Werten wird ein entsprechender 2-fach-Baustein bevorzugt. `ALR_MUL_3` spart bei drei Faktoren die zusätzliche Kaskadierung von zwei separaten Bausteinen ein.

- `ALR_MUL_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

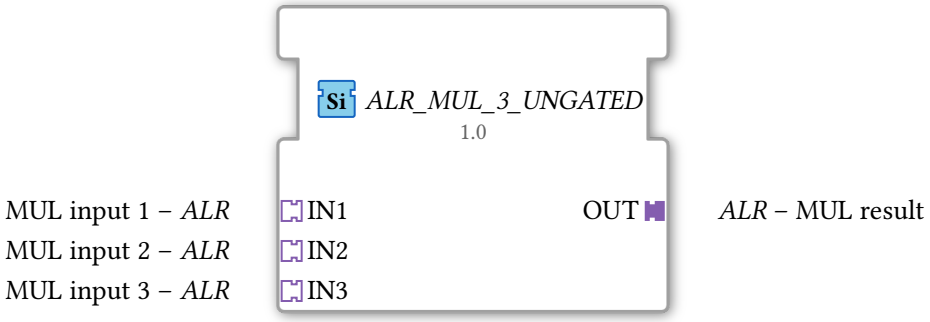
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der Baustein `ALR_MUL_3` ist eine effiziente und saubere Lösung für dreifache Multiplikationsaufgaben in IEC 61499-Anwendungen. Durch die konsequente Nutzung von unidirektionalen Adaptern fördert er ein sauberes Software-Design und sorgt für gut strukturierte Datenflüsse in der 4diac-IDE.

ALR_MUL_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALR_MUL_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Symbolische Darstellung des Funktionsbausteins)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ALR_MUL_3_UNGATED` ist ein generischer Baustein aus der Bibliothek `adapter::iec61131::arithmetic`, der für die arithmetische Multiplikation von drei Eingangswerten entwickelt wurde. Anstelle von klassischen, diskreten Dateneingängen nutzt dieser Baustein unidirektionale Adapter des Typs `ALR` zur Kapselung und Übertragung der Daten und Steuerungsereignisse. Dies ermöglicht eine strukturierte, modulare und übersichtliche Verdrahtung innerhalb der 4diac-IDE.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die ereignisbasierte Steuerung wird vollständig über die eingebundenen Adapter abgewickelt.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die ereignisbasierte Steuerung wird vollständig über die eingebundenen Adapter abgewickelt.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden. Die Datenwerte sind in den Eingangs-Adaptern gekapselt.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden. Das Ergebnis ist im Ausgangs-Adapter gekapselt.

Adapter

Sockets (Eingangsschnittstellen)

- **IN1** (Typ: `adapter::types::unidirectional::ALR`): Der erste Faktor für die Multiplikation.
- **IN2** (Typ: `adapter::types::unidirectional::ALR`): Der zweite Faktor für die Multiplikation.
- **IN3** (Typ: `adapter::types::unidirectional::ALR`): Der dritte Faktor für die Multiplikation.

Plugs (Ausgangsschnittstellen)

- **OUT** (Typ: `adapter::types::unidirectional::ALR`): Das berechnete Produkt der drei Eingangswerte.

Funktionsweise

Der Baustein `ALR_MUL_3_UNGATED` realisiert die mathematische Multiplikation von drei Eingangsvariablen. Sobald an den Sockets neue Werte bzw. Triggerereignisse anliegen, führt der Baustein die Berechnung nach folgender Formel aus:

$$\text{OUT} = \text{IN1} \times \text{IN2} \times \text{IN3} \tag{2.61}$$

Das Ergebnis der Berechnung sowie das zugehörige Aktualisierungsereignis werden über den Plug `OUT` ausgegeben.

Da es sich um einen generischen Baustein (Generic-Klasse: `GEN_ALR_MUL`) handelt, hängt der tatsächliche Datentyp (z. B. `REAL`, `LREAL`, `INT`) von der Definition des verwendeten `ALR`-Adapters ab.

Technische Besonderheiten

- **Generisches Design:** Durch die Zuweisung des Attributs `GenericClassName = 'GEN_ALR_MUL'` ist der Baustein flexibel für verschiedene Datentypen einsetzbar, sofern die verwendeten Adapter diese unterstützen.
- **Adapterbasierte Kopplung:** Die Verwendung von Adaptern statt loser Event-/Data-Verbindungen reduziert den Verdrahtungsaufwand (Routing) innerhalb der 4diac-Anwendung drastisch und erhöht die Übersichtlichkeit.
- **Unidirektionalität:** Die verwendeten `ALR`-Adapter sind als unidirektional definiert, was einen klaren und rückwirkungsfreien Datenfluss von den Quellen (Sockets) zur Senke (Plug) gewährleistet.

Zustandsübersicht

Der Baustein besitzt keine komplexe interne Zustandsmaschine (ECC). Er arbeitet als rein kombinatorischer/mathematischer Baustein, der direkt auf Ereignisse an den Eingangs-Adaptern reagiert und das Ergebnis unmittelbar an den Ausgang weiterleitet.

Anwendungsszenarien

- **Sensorskalierung und -korrektur:** Multiplikation eines Rohwerts (IN1) mit einem Kalibrierungsfaktor (IN2) und einem anwendungsbezogenen Gewichtungsfaktor (IN3).
- **Dreidimensionale Berechnungen:** Berechnung von Volumen oder Durchsätzen, bei denen drei physikalische Einflussgrößen miteinander multipliziert werden müssen.
- **Kaskadierte Verstärkungsglieder:** Berechnung von kombinierten Verstärkungen in Regelungskreisen.

Vergleich mit ähnlichen Bausteinen

- **Standard MUL-Baustein (IEC 61131-3):** Klassische Multiplizierer arbeiten mit direkten Elementardatentypen (z. B. `REAL`). `ALR_MUL_3_UNGATED` hingegen bündelt Daten und Ereignisse in Adaptern, was die Modularität und Wiederverwendbarkeit in verteilten Systemen nach IEC 61499 verbessert.
- **ALR_MUL_2 (2-fach Multiplizierer):** Für die Multiplikation von nur zwei Werten wird ein entsprechender 2-fach-Baustein bevorzugt. `ALR_MUL_3_UNGATED` spart bei drei Faktoren die zusätzliche Kaskadierung von zwei separaten Bausteinen ein.

- `ALR_MUL_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der Baustein `ALR_MUL_3_UNGATED` ist eine effiziente und saubere Lösung für dreifache Multiplikationsaufgaben in IEC 61499-Anwendungen. Durch die konsequente Nutzung von unidirektionalen Adaptern fördert er ein sauberes Software-Design und sorgt für gut strukturierte Datenflüsse in der 4diac-IDE.

ALR_MUL_4

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ALR_MUL_4` dient der Durchführung einer arithmetischen Multiplikation von vier Eingangswerten. Es handelt sich hierbei um einen generischen Funktionsbaustein (Generic FB) für die 4diac-IDE, der auf dem IEC 61499-Standard basiert. Die Signalübertragung und -verarbeitung erfolgt über spezielle unidirektionale Analogadapter, was eine saubere Kapselung der Datenströme ermöglicht.

Schnittstellenstruktur

Ereignis-Eingänge

Für diesen Funktionsbaustein sind keine direkten Ereignis-Eingänge definiert, da die Steuerung und Aktualisierung über die Adapter erfolgt.

Ereignis-Ausgänge

Für diesen Funktionsbaustein sind keine direkten Ereignis-Ausgänge definiert.

Daten-Eingänge

Für diesen Funktionsbaustein sind keine direkten Daten-Eingänge definiert.

Daten-Ausgänge

Für diesen Funktionsbaustein sind keine direkten Daten-Ausgänge definiert.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::ALR`): Erster Eingangswert (Multiplikand 1) für die Multiplikation.
- **IN2** (Typ: `adapter::types::unidirectional::ALR`): Zweiter Eingangswert (Multiplikand 2) für die Multiplikation.
- **IN3** (Typ: `adapter::types::unidirectional::ALR`): Dritter Eingangswert (Multiplikand 3) für die Multiplikation.
- **IN4** (Typ: `adapter::types::unidirectional::ALR`): Vierter Eingangswert (Multiplikand 4) für die Multiplikation.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::ALR`): Ausgang für das berechnete Produkt der vier Eingangswerte.

Funktionsweise

Der Baustein `ALR_MUL_4` multipliziert die analogen Werte, die über die vier Eingangs-Adapter (`IN1` bis `IN4`) bereitgestellt werden, miteinander. Das mathematische Ergebnis wird über den Ausgangs-Adapter `OUT` ausgegeben.

Die zugrundeliegende Berechnungsformel lautet:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \times \mathrm{IN4} \tag{2.62}$$

Da es sich um einen generischen Funktionsbaustein handelt, der die Klasse `GEN_ALR_MUL` verwendet, ist die Implementierung flexibel gegenüber den im Adapter genutzten Datentypen (z. B. `REAL` oder `LREAL`).

Technische Besonderheiten

- **Generischer Baustein**: Durch das Attribut `GenericClassName` mit dem Wert `GEN_ALR_MUL` kann sich der Baustein zur Laufzeit bzw. beim Kompilieren flexibel an die konkrete Datentyp-Ausprägung der verwendeten Adapter anpassen.
- **Kapselung durch Adapter**: Die Verwendung des unidirektionalen Adapters `ALR` (`adapter::types::unidirectional::ALR`) sorgt dafür, dass Datenwerte und gegebenenfalls zugehörige Status- oder Triggerereignisse kompakt in einer einzigen Verbindung gebündelt werden. Dies reduziert den Verdrahtungsaufwand im Funktionsplan erheblich.

Zustandsübersicht

Der Baustein besitzt kein internes Zustandsverhalten (keine State Machine / ECC) und arbeitet rein datenflussorientiert. Sobald sich die Werte an den Eingangs-Adaptern ändern oder ein entsprechendes Aktualisierungsereignis über die Adapter getriggert wird, wird die Multiplikation ausgeführt und das Ergebnis am Ausgang `OUT` bereitgestellt.

Anwendungsszenarien

- **Physikalische Berechnungen**: Berechnung von komplexeren Größen, die das Produkt mehrerer Variablen darstellen (z. B. Leistung, Energieberechnungen oder Volumenströme unter Berücksichtigung von Korrekturfaktoren).
- **Kaskadierte Skalierung**: Anwendung von mehreren Skalierungs- oder Korrekturfaktoren auf ein analoges Eingangssignal in einem einzigen Schritt.
- **Signalverarbeitung**: Vorverarbeitung von Sensorwerten in Steuerungssystemen, bevor die Daten an Visualisierungen oder Aktoren weitergegeben werden.

Vergleich mit ähnlichen Bausteinen

- **Standard-MUL-Bausteine (IEC 61131-3)**: Klassische Multiplikationsbausteine arbeiten in der Regel direkt mit Standard-Datentypen (z. B. `REAL`) und erfordern separate Event-Verbindungen (`REQ`/`CNF`). `ALR_MUL_4` vereinfacht dies durch die Nutzung von Adaptern.
- **ALR_MUL_2 / ALR_MUL_3**: Diese Bausteine sind für die Multiplikation von nur zwei oder drei Werten ausgelegt. Der `ALR_MUL_4` spart bei der Notwendigkeit von vier Multiplikanden zusätzliche Zwischenschritte und zusätzliche Bausteinkaskaden ein, was die Performance und Übersichtlichkeit des Steuerungsprogramms verbessert.
- `ALR_MUL_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

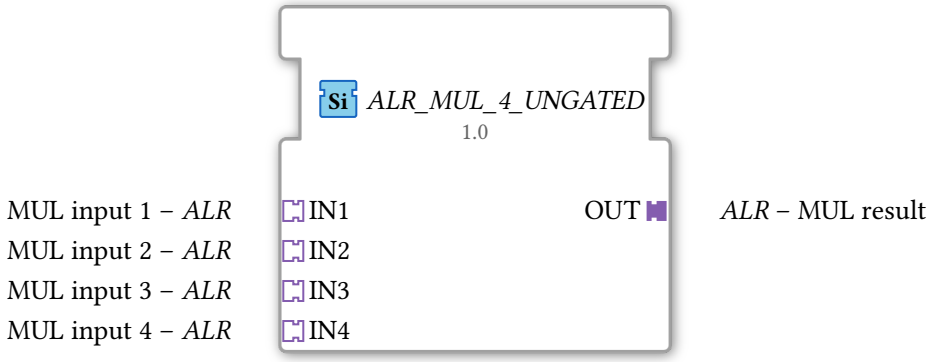
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `ALR_MUL_4`-Funktionsbaustein ist ein praktischer Hilfsbaustein für die mathematische Signalverarbeitung in 4diac. Durch die konsequente Nutzung des Adapterkonzepts trägt er maßgeblich zur Reduzierung von Verbindungslinien im Anwendungsdiagramm bei und bietet gleichzeitig die Flexibilität eines generischen Funktionsbausteins.

ALR_MUL_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALR_MUL_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ALR_MUL_4_UNGATED` dient der Durchführung einer arithmetischen Multiplikation von vier Eingangswerten. Es handelt sich hierbei um einen generischen Funktionsbaustein (Generic FB) für die 4diac-IDE, der auf dem IEC 61499-Standard basiert. Die Signalübertragung und -verarbeitung erfolgt über spezielle unidirektionale Analogadapter, was eine saubere Kapselung der Datenströme ermöglicht.

Schnittstellenstruktur

Ereignis-Eingänge

Für diesen Funktionsbaustein sind keine direkten Ereignis-Eingänge definiert, da die Steuerung und Aktualisierung über die Adapter erfolgt.

Ereignis-Ausgänge

Für diesen Funktionsbaustein sind keine direkten Ereignis-Ausgänge definiert.

Daten-Eingänge

Für diesen Funktionsbaustein sind keine direkten Daten-Eingänge definiert.

Daten-Ausgänge

Für diesen Funktionsbaustein sind keine direkten Daten-Ausgänge definiert.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::ALR`): Erster Eingangswert (Multiplikand 1) für die Multiplikation.
- **IN2** (Typ: `adapter::types::unidirectional::ALR`): Zweiter Eingangswert (Multiplikand 2) für die Multiplikation.
- **IN3** (Typ: `adapter::types::unidirectional::ALR`): Dritter Eingangswert (Multiplikand 3) für die Multiplikation.
- **IN4** (Typ: `adapter::types::unidirectional::ALR`): Vierter Eingangswert (Multiplikand 4) für die Multiplikation.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::ALR`): Ausgang für das berechnete Produkt der vier Eingangswerte.

Funktionsweise

Der Baustein `ALR_MUL_4_UNGATED` multipliziert die analogen Werte, die über die vier Eingangs-Adapter (`IN1` bis `IN4`) bereitgestellt werden, miteinander. Das mathematische Ergebnis wird über den Ausgangs-Adapter `OUT` ausgegeben.

Die zugrundeliegende Berechnungsformel lautet:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \times \mathrm{IN4} \tag{2.63}$$

Da es sich um einen generischen Funktionsbaustein handelt, der die Klasse `GEN_ALR_MUL` verwendet, ist die Implementierung flexibel gegenüber den im Adapter genutzten Datentypen (z. B. `REAL` oder `LREAL`).

Technische Besonderheiten

- **Generischer Baustein**: Durch das Attribut `GenericClassName` mit dem Wert `GEN_ALR_MUL` kann sich der Baustein zur Laufzeit bzw. beim Kompilieren flexibel an die konkrete Datentyp-Ausprägung der verwendeten Adapter anpassen.
- **Kapselung durch Adapter**: Die Verwendung des unidirektionalen Adapters `ALR` (`adapter::types::unidirectional::ALR`) sorgt dafür, dass Datenwerte und gegebenenfalls zugehörige Status- oder Triggerereignisse kompakt in einer einzigen Verbindung gebündelt werden. Dies reduziert den Verdrahtungsaufwand im Funktionsplan erheblich.

Zustandsübersicht

Der Baustein besitzt kein internes Zustandsverhalten (keine State Machine / ECC) und arbeitet rein datenflussorientiert. Sobald sich die Werte an den Eingangs-Adaptern ändern oder ein entsprechendes Aktualisierungsereignis über die Adapter getriggert wird, wird die Multiplikation ausgeführt und das Ergebnis am Ausgang `OUT` bereitgestellt.

Anwendungsszenarien

- **Physikalische Berechnungen**: Berechnung von komplexeren Größen, die das Produkt mehrerer Variablen darstellen (z. B. Leistung, Energieberechnungen oder Volumenströme unter Berücksichtigung von Korrekturfaktoren).
- **Kaskadierte Skalierung**: Anwendung von mehreren Skalierungs- oder Korrekturfaktoren auf ein analoges Eingangssignal in einem einzigen Schritt.
- **Signalverarbeitung**: Vorverarbeitung von Sensorwerten in Steuerungssystemen, bevor die Daten an Visualisierungen oder Aktoren weitergegeben werden.

Vergleich mit ähnlichen Bausteinen

- **Standard-MUL-Bausteine (IEC 61131-3)**: Klassische Multiplikationsbausteine arbeiten in der Regel direkt mit Standard-Datentypen (z. B. `REAL`) und erfordern separate Event-Verbindungen (`REQ`/`CNF`). `ALR_MUL_4_UNGATED` vereinfacht dies durch die Nutzung von Adaptern.
- **ALR_MUL_2 / ALR_MUL_3**: Diese Bausteine sind für die Multiplikation von nur zwei oder drei Werten ausgelegt. Der `ALR_MUL_4_UNGATED` spart bei der Notwendigkeit von vier Multiplikanden zusätzliche Zwischenschritte und zusätzliche Bausteinkaskaden ein, was die Performance und Übersichtlichkeit des Steuerungsprogramms verbessert.
- `ALR_MUL_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

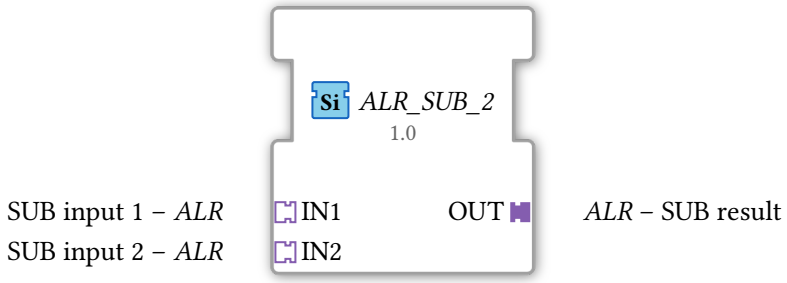
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `ALR_MUL_4_UNGATED`-Funktionsbaustein ist ein praktischer Hilfsbaustein für die mathematische Signalverarbeitung in 4diac. Durch die konsequente Nutzung des Adapterkonzepts trägt er maßgeblich zur Reduzierung von Verbindungslinien im Anwendungsdiagramm bei und bietet gleichzeitig die Flexibilität eines generischen Funktionsbausteins.

ALR_SUB_2



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ALR_SUB_2` ist ein generischer Baustein zur Durchführung von mathematischen Subtraktionen innerhalb einer IEC 61499-Entwicklungsumgebung (wie 4diac IDE). Anstatt direkte Daten- und Ereignispins zu nutzen, basiert dieser Baustein auf einer adaptergesteuerten Architektur. Dies ermöglicht eine strukturierte und saubere Kapselung von Signalen und vereinfacht die Verdrahtung in komplexen Steuerungsanwendungen.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Baustein besitzt keine direkten Ereignis-Eingänge. Die Steuerung und Triggerung erfolgt über die angeschlossenen Adapter.

Ereignis-Ausgänge

Dieser Baustein besitzt keine direkten Ereignis-Ausgänge. Die Ereignisweiterleitung erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::ALR`): Der erste Eingangswert (Minuend) der Subtraktion. Über diesen Adapter werden sowohl der Wert als auch das auslösende Ereignis empfangen.
- **IN2** (Typ: `adapter::types::unidirectional::ALR`): Der zweite Eingangswert (Subtrahend), der vom ersten Wert abgezogen werden soll.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::ALR`): Das Ergebnis der Subtraktion (Differenz). Dieser Adapter gibt das berechnete Ergebnis mitsamt dem zugehörigen Ausgangsereignis aus.

Funktionsweise

Die interne Logik des Bausteins berechnet die Differenz aus den beiden über die Sockets bereitgestellten Werten. Die mathematische Formel lautet:

$$\mathrm{OUT} = \mathrm{IN1} - \mathrm{IN2} \tag{2.64}$$

Sobald sich die Werte an den Eingangs-Adaptern ändern oder ein entsprechendes Trigger-Ereignis an `IN1` oder `IN2` empfangen wird, führt der Baustein die Berechnung aus und aktualisiert den Wert am Ausgangs-Plug `OUT`, gefolgt von einem Ausgangsereignis.

Technische Besonderheiten

- **Generisches Verhalten**: Der Baustein ist als generischer Typ (`GEN_ALR_SUB`) definiert. Er ist somit nicht starr auf einen Datentyp (wie z. B. nur `REAL` oder nur `INT`) festgelegt, sondern passt sich flexibel an die im Adapter `ALR` definierten Datentypen an.
- **Unidirektionale Adapter**: Durch die Verwendung von unidirektionalen Adaptern (`unidirectional::ALR`) wird der Datenfluss klar in eine Richtung gelenkt, was die Systemstabilität erhöht und unnötigen Kommunikations-Overhead minimiert.

Zustandsübersicht

Da es sich bei `ALR_SUB_2` um einen rein mathematischen, datenflussorientierten Funktionsbaustein handelt, besitzt er keinen internen Zustandsautomaten (ECC - Execution Control Chart). Die Abarbeitung erfolgt rein ereignis- und datengesteuert.

Anwendungsszenarien

- **Regelungstechnik (Berechnung der Regelabweichung)**: Ermittlung der Differenz zwischen einem Sollwert (Führungsgröße an `IN1`) und einem Istwert (Regelgröße an `IN2`), um die Regelabweichung am Ausgang `OUT` für einen nachgelagerten PID-Regler bereitzustellen.
- **Differenzmessungen**: Berechnung von physikalischen Differenzwerten, beispielsweise zur Ermittlung des Druckabfalls über einem Filter (Druck vor dem Filter an `IN1` minus Druck nach dem Filter an `IN2`).
- **Offset-Korrektur**: Abzug eines konstanten oder variablen Korrekturwertes von einem analogen Rohsignal.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu klassischen IEC 61131-3 Subtraktionsbausteinen (wie z. B. `SUB`), die mit Standard-Datentypen und direkten Pins arbeiten, bietet der `ALR_SUB_2` durch die Kapselung in Adaptern erhebliche Vorteile bei der Übersichtlichkeit des Programmcodes. Komplexe Signaladern (Daten + Ereignis) werden in einer einzigen Adapterverbindung gebündelt.

Gegenüber bidirektionalen Berechnungsbausteinen zeichnet sich der `ALR_SUB_2` durch eine deterministische, rückwirkungsfreie Signalverarbeitung aus.

- `ALR_SUB_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

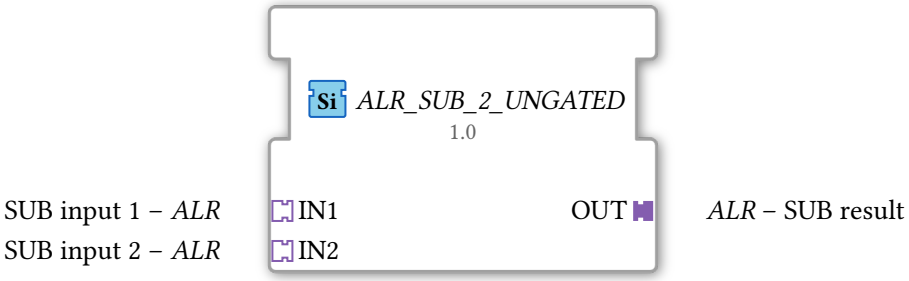
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `ALR_SUB_2` ist ein effizienter und flexibel einsetzbarer Standardbaustein für arithmetische Berechnungen in modernen, serviceorientierten Steuerungssystemen. Durch die Nutzung der Adapter-Technologie unterstützt er ein modulares und wartungsfreundliches Softwaredesign nach dem IEC 61499-Standard.

ALR_SUB_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALR_SUB_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ALR_SUB_2_UNGATED` ist ein generischer Baustein zur Durchführung von mathematischen Subtraktionen innerhalb einer IEC 61499-Entwicklungsumgebung (wie 4diac IDE). Anstatt direkte Daten- und Ereignispins zu nutzen, basiert dieser Baustein auf einer adaptergesteuerten Architektur. Dies ermöglicht eine strukturierte und saubere Kapselung von Signalen und vereinfacht die Verdrahtung in komplexen Steuerungsanwendungen.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Baustein besitzt keine direkten Ereignis-Eingänge. Die Steuerung und Triggerung erfolgt über die angeschlossenen Adapter.

Ereignis-Ausgänge

Dieser Baustein besitzt keine direkten Ereignis-Ausgänge. Die Ereignisweiterleitung erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::ALR`): Der erste Eingangswert (Minuend) der Subtraktion. Über diesen Adapter werden sowohl der Wert als auch das auslösende Ereignis empfangen.
- **IN2** (Typ: `adapter::types::unidirectional::ALR`): Der zweite Eingangswert (Subtrahend), der vom ersten Wert abgezogen werden soll.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::ALR`): Das Ergebnis der Subtraktion (Differenz). Dieser Adapter gibt das berechnete Ergebnis mitsamt dem zugehörigen Ausgangsereignis aus.

Funktionsweise

Die interne Logik des Bausteins berechnet die Differenz aus den beiden über die Sockets bereitgestellten Werten. Die mathematische Formel lautet:

$$\mathrm{OUT} = \mathrm{IN1} - \mathrm{IN2} \tag{2.65}$$

Sobald sich die Werte an den Eingangs-Adaptern ändern oder ein entsprechendes Trigger-Ereignis an `IN1` oder `IN2` empfangen wird, führt der Baustein die Berechnung aus und aktualisiert den Wert am Ausgangs-Plug `OUT`, gefolgt von einem Ausgangsereignis.

Technische Besonderheiten

- **Generisches Verhalten**: Der Baustein ist als generischer Typ (`GEN_ALR_SUB`) definiert. Er ist somit nicht starr auf einen Datentyp (wie z. B. nur `REAL` oder nur `INT`) festgelegt, sondern passt sich flexibel an die im Adapter `ALR` definierten Datentypen an.
- **Unidirektionale Adapter**: Durch die Verwendung von unidirektionalen Adaptern (`unidirectional::ALR`) wird der Datenfluss klar in eine Richtung gelenkt, was die Systemstabilität erhöht und unnötigen Kommunikations-Overhead minimiert.

Zustandsübersicht

Da es sich bei `ALR_SUB_2_UNGATED` um einen rein mathematischen, datenflussorientierten Funktionsbaustein handelt, besitzt er keinen internen Zustandsautomaten (ECC - Execution Control Chart). Die Abarbeitung erfolgt rein ereignis- und datengesteuert.

Anwendungsszenarien

- **Regelungstechnik (Berechnung der Regelabweichung)**: Ermittlung der Differenz zwischen einem Sollwert (Führungsgröße an `IN1`) und einem Istwert (Regelgröße an `IN2`), um die Regelabweichung am Ausgang `OUT` für einen nachgelagerten PID-Regler bereitzustellen.
- **Differenzmessungen**: Berechnung von physikalischen Differenzwerten, beispielsweise zur Ermittlung des Druckabfalls über einem Filter (Druck vor dem Filter an `IN1` minus Druck nach dem Filter an `IN2`).
- **Offset-Korrektur**: Abzug eines konstanten oder variablen Korrekturwertes von einem analogen Rohsignal.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu klassischen IEC 61131-3 Subtraktionsbausteinen (wie z. B. `SUB`), die mit Standard-Datentypen und direkten Pins arbeiten, bietet der `ALR_SUB_2_UNGATED` durch die Kapselung in Adaptern erhebliche Vorteile bei der Übersichtlichkeit des Programmcodes. Komplexe Signaladern (Daten + Ereignis) werden in einer einzigen Adapterverbindung gebündelt.

Gegenüber bidirektionalen Berechnungsbausteinen zeichnet sich der `ALR_SUB_2_UNGATED` durch eine deterministische, rückwirkungsfreie Signalverarbeitung aus.

- `ALR_SUB_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

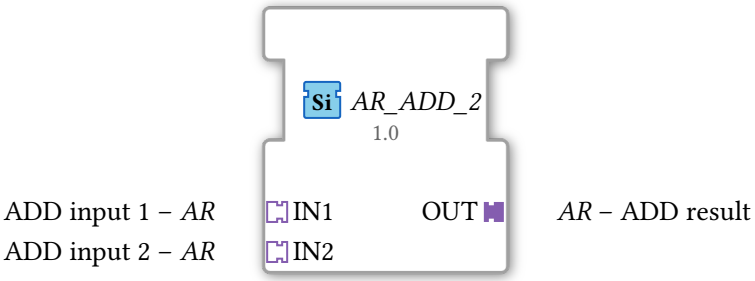
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `ALR_SUB_2_UNGATED` ist ein effizienter und flexibel einsetzbarer Standardbaustein für arithmetische Berechnungen in modernen, serviceorientierten Steuerungssystemen. Durch die Nutzung der Adapter-Technologie unterstützt er ein modulares und wartungsfreundliches Softwaredesign nach dem IEC 61499-Standard.

AR_ADD_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AR_ADD_2** dient der Berechnung der arithmetischen Addition zweier Werte. Er ist als generischer FB ausgelegt und realisiert die grundlegende Rechenoperation über Adapter-Schnittstellen. Der Baustein ist plattformunabhängig und folgt dem IEC-61499-Standard.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Bezeichnung	Richtung	Typ	Kommentar
`IN1`	Socket	`adapter::types::unidirectional::AR`	ADD input 1
`IN2`	Socket	`adapter::types::unidirectional::AR`	ADD input 2
`OUT`	Plug	`adapter::types::unidirectional::AR`	ADD result

Die Adapter sind vom Typ `unidirectional::AR` und ermöglichen die typsichere Verbindung mit anderen Bausteinen, die denselben Adaptertyp unterstützen.

Funktionsweise

Der FB führt die Addition der beiden über die Adapter `IN1` und `IN2` anliegenden Werte durch. Das Ergebnis wird am Adapter `OUT` bereitgestellt. Die genaue Dateninterpretation (Datentyp, Wertebereich) wird durch den verwendeten Adaptertyp bestimmt; der Baustein selbst arbeitet generisch und delegiert die Typinformation an das zugrundeliegende Framework.

Die eigentliche Berechnung erfolgt im Hintergrund über die konfigurierten Laufzeitumgebungen – der FB definiert lediglich die Schnittstellen und die Semantik der Addition.

Technische Besonderheiten

- **Generischer Aufbau** – Der FB verwendet zur Laufzeit einen generischen Klassennamen (`GEN_AR_ADD`), der durch das Attribut `eclipse4diac::core::GenericClassName` festgelegt wird. Dies ermöglicht eine spätere Spezialisierung auf konkrete Datentypen (z. B. Ganzzahlen oder Gleitkommazahlen).
- **Adapterbasiert** – Statt einzelner Daten-Ein- und Ausgänge werden alle Signale über unidirektionale Adapter geführt. Das vereinfacht die Kapselung und Wiederverwendung in komplexen Systemen.
- **Paketangabe** – Der FB ist im Paket `adapter::iec61131::arithmetic` organisiert, was auf eine Herkunft aus einer IEC-61131-Adapterbibliothek hindeutet.
- **Keine Zustandslogik** – Die Addition erfolgt zustandslos; es existiert kein interner Zustandsautomat.

Zustandsübersicht

Der Baustein besitzt keinen Zustandsautomaten. Die Berechnung wird rein datengetrieben durchgeführt – bei Änderung der Eingangsadapter wird das Ergebnis neu berechnet.

Anwendungsspezifische Szenarien

- **Signalverarbeitung** – Kombination zweier analoger oder digitaler Messwerte zu einer Summe.
- **Steuerungstechnik** – Aufsummieren von Zählerständen oder Zeitwerten.
- **Generische Bibliotheksbausteine** – Einsatz als Basis für anwendungsspezifische, getypte Additionsbausteine (z. B. `AR_ADD_INT`, `AR_ADD_REAL`).

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu klassischen, Daten-eingabebasierten Additionsbausteinen (z. B. `ADD`) zeichnet sich `AR_ADD_2` durch seine Adapterschnittstellen aus. Während Standard-Addierer oft mit einzelnen Eingängen (`IN1`, `IN2`) und einem Ausgang (`OUT`) arbeiten, ermöglichen Adapter die direkte Verbindung zu komplexeren Datenstrukturen und eine höhere Wiederverwendbarkeit durch Typsicherheit. Der Baustein ist bewusst generisch gehalten, was eine dynamische Typanpassung zur Laufzeit erlaubt.

- **`AR_ADD_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

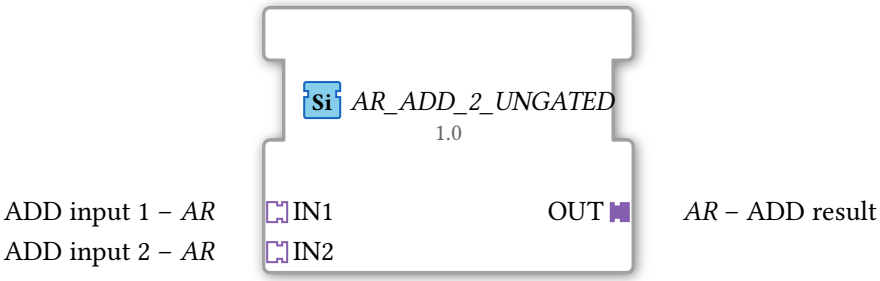
`AR_ADD_2` ist ein kompakter, generischer Funktionsbaustein für die Addition von zwei Werten mittels Adapter-Schnittstellen. Er eignet sich besonders für systemübergreifende Anwendungen, bei denen Typflexibilität und lose Kopplung durch Adapter gefragt sind. Die einfache, zustandslose Architektur macht ihn robust und leicht in bestehende Automatisierungsprojekte integrierbar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AR_ADD_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AR_ADD_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AR_ADD_2_UNGATED** dient der Berechnung der arithmetischen Addition zweier Werte. Er ist als generischer FB ausgelegt und realisiert die grundlegende Rechenoperation über Adapter-Schnittstellen. Der Baustein ist plattformunabhängig und folgt dem IEC-61499-Standard.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Bezeichnung	Richtung	Typ	Kommentar
`IN1`	Socket	`adapter::types::unidirectional::AR`	ADD input 1
`IN2`	Socket	`adapter::types::unidirectional::AR`	ADD input 2
`OUT`	Plug	`adapter::types::unidirectional::AR`	ADD result

Die Adapter sind vom Typ `unidirectional::AR` und ermöglichen die typsichere Verbindung mit anderen Bausteinen, die denselben Adaptertyp unterstützen.

Funktionsweise

Der FB führt die Addition der beiden über die Adapter `IN1` und `IN2` anliegenden Werte durch. Das Ergebnis wird am Adapter `OUT` bereitgestellt. Die genaue Dateninterpretation (Datentyp, Wertebereich) wird durch den verwendeten Adaptertyp bestimmt; der Baustein selbst arbeitet generisch und delegiert die Typinformation an das zugrundeliegende Framework.

Die eigentliche Berechnung erfolgt im Hintergrund über die konfigurierten Laufzeitumgebungen – der FB definiert lediglich die Schnittstellen und die Semantik der Addition.

Technische Besonderheiten

- **Generischer Aufbau** – Der FB verwendet zur Laufzeit einen generischen Klassennamen (`GEN_AR_ADD`), der durch das Attribut `eclipse4diac::core::GenericClassName` festgelegt wird. Dies ermöglicht eine spätere Spezialisierung auf konkrete Datentypen (z. B. Ganzzahlen oder Gleitkommazahlen).
- **Adapterbasiert** – Statt einzelner Daten-Ein- und Ausgänge werden alle Signale über unidirektionale Adapter geführt. Das vereinfacht die Kapselung und Wiederverwendung in komplexen Systemen.
- **Paketangabe** – Der FB ist im Paket `adapter::iec61131::arithmetic` organisiert, was auf eine Herkunft aus einer IEC-61131-Adapterbibliothek hindeutet.
- **Keine Zustandslogik** – Die Addition erfolgt zustandslos; es existiert kein interner Zustandsautomat.

Zustandsübersicht

Der Baustein besitzt keinen Zustandsautomaten. Die Berechnung wird rein datengetrieben durchgeführt – bei Änderung der Eingangsadapter wird das Ergebnis neu berechnet.

Anwendungsspezifische Szenarien

- **Signalverarbeitung** – Kombination zweier analoger oder digitaler Messwerte zu einer Summe.
- **Steuerungstechnik** – Aufsummieren von Zählerständen oder Zeitwerten.
- **Generische Bibliotheksbausteine** – Einsatz als Basis für anwendungsspezifische, getypte Additionsbausteine (z. B. `AR_ADD_INT`, `AR_ADD_REAL`).

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu klassischen, Daten-eingabebasierten Additionsbausteinen (z. B. `ADD`) zeichnet sich `AR_ADD_2_UNGATED` durch seine Adapterschnittstellen aus. Während Standard-Addierer oft mit einzelnen Eingängen (`IN1`, `IN2`) und einem Ausgang (`OUT`) arbeiten, ermöglichen Adapter die direkte Verbindung zu komplexeren Datenstrukturen und eine höhere Wiederverwendbarkeit durch Typsicherheit. Der Baustein ist bewusst generisch gehalten, was eine dynamische Typanpassung zur Laufzeit erlaubt.

- `AR_ADD_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

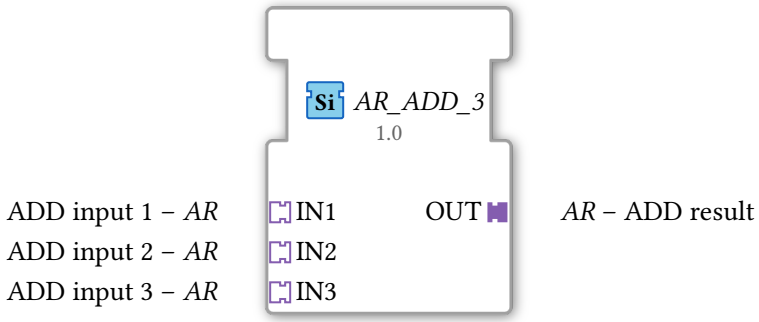
Fazit

`AR_ADD_2_UNGATED` ist ein kompakter, generischer Funktionsbaustein für die Addition von zwei Werten mittels Adapter-Schnittstellen. Er eignet sich besonders für systemübergreifende Anwendungen, bei denen Typflexibilität und lose Kopplung durch Adapter gefragt sind. Die einfache, zustandslose Architektur macht ihn robust und leicht in bestehende Automatisierungsprojekte integrierbar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AR_ADD_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AR_ADD_3` dient der Berechnung einer arithmetischen Addition von drei Werten. Es handelt sich um einen generischen Baustein (generic FB) gemäß IEC 61499-2, der über Adapter-Schnittstellen arbeitet. Der Block ist so konzipiert, dass er in Verbindung mit dem Adapter `adapter::types::unidirectional::AR` der Bibliothek `adapter::iec61131::arithmetic` eingesetzt wird.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine direkten Dateneingänge. Die Eingabewerte werden über Adapter-Schnittstellen bereitgestellt.

Daten-Ausgänge

Keine direkten Datenausgänge. Das Ergebnis wird über einen Adapter-Ausgang bereitgestellt.

Adapter

Richtung	Name	Typ	Kommentar
Sockets	IN1	`adapter::types::unidirectional::AR`	ADD input 1
Sockets	IN2	`adapter::types::unidirectional::AR`	ADD input 2
Sockets	IN3	`adapter::types::unidirectional::AR`	ADD input 3

Plugs	OUT	`adapter::types::unidirectional::AR`	ADD result

Funktionsweise

Der Baustein addiert die drei über die Adapter-Sockets `IN1`, `IN2` und `IN3` eingehenden Werte und stellt das Ergebnis über den Adapter-Plug `OUT` bereit. Die genaue Semantik der Addition (z. B. unterstützte Datentypen, Überlaufverhalten) wird durch die Implementierung des verwendeten Adapter-Typs `AR` festgelegt. Der Baustein ist generisch, d. h. er kann für verschiedene Datentypen parametrisiert werden (z. B. durch Type-Hash oder generischen Klassennamen).

Technische Besonderheiten

- **Generischer Baustein:** Die tatsächliche Datentypbehandlung wird über das Attribut `eclipse4diac::core::GenericClassName` (Wert: `'GEN_AR_ADD'`) und `eclipse4diac::core::TypeHash` gesteuert.
- **Adapterbasierte Kommunikation:** Der Baustein verwendet ausschließlich Adapter (`adapter::types::unidirectional::AR`) für Ein- und Ausgabe. Dies ermöglicht eine lose Kopplung und Wiederverwendbarkeit in unterschiedlichen Kontexten.
- **Lizenz:** Unter der Eclipse Public License 2.0 (EPL-2.0) veröffentlicht.
- **Kompatibilität:** Entwickelt für die 4diac-IDE und die Eclipse 4diac Laufzeitumgebung.

Zustandsübersicht

Der Baustein besitzt kein explizites Zustandsdiagramm (ECC). Es handelt sich um eine rein kombinatorische Funktion: Die Ausgabe ist jederzeit die Summe der drei Eingänge. Es sind keine internen Zustände oder zeitlichen Abhängigkeiten vorhanden.

Anwendungsszenarien

- **Mehrfachaddition in Steuerungssystemen:** Wenn drei analoge oder digitale Werte (z. B. Sensordaten, Sollwerte) aufsummiert werden müssen.
- **Generische Bibliotheksbausteine:** Einsatz in wiederverwendbaren Funktionsbaustein-Bibliotheken, bei denen die Datentypen zur Konfigurationszeit festgelegt werden.
- **Adapterbasierte Architekturen:** In Systemen, die auf standardisierte Adapter-Schnittstellen setzen, um eine flexible Verkettung von Funktionen zu ermöglichen.

Vergleich mit ähnlichen Bausteinen

AR_ADD_2 / AR_ADD_4 etc.: Diese Bausteine unterscheiden sich lediglich in der Anzahl der Eingänge (zwei bzw. vier). `AR_ADD_3` ist speziell für die Addition von genau drei Werten optimiert. Der generische Ansatz erlaubt es, alle diese Varianten bei Bedarf durch Parametrisierung eines gemeinsamen Basistyps zu realisieren.

- `AR_ADD_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

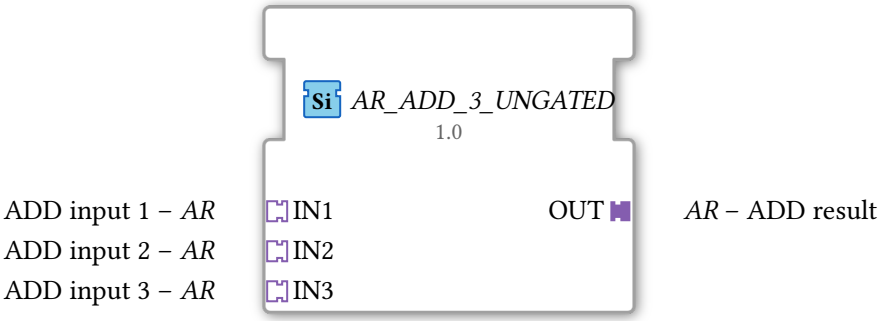
`AR_ADD_3` ist ein übersichtlicher, generischer Funktionsblock für die Addition dreier Werte unter Verwendung von Adaptern. Durch die Abstraktion über generische Attribute und Adapter eignet er sich gut für IEC-61499-Anwendungen, die eine flexible und wiederverwendbare Signalverarbeitung erfordern.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AR_ADD_3_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `AR_ADD_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AR_ADD_3_UNGATED` dient der Berechnung einer arithmetischen Addition von drei Werten. Es handelt sich um einen generischen Baustein (generic FB) gemäß IEC 61499-2, der über Adapter-Schnittstellen arbeitet. Der Block ist so konzipiert, dass er in Verbindung mit dem Adapter `adapter::types::unidirectional::AR` der Bibliothek `adapter::iec61131::arithmetic` eingesetzt wird.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine direkten Dateneingänge. Die Eingabewerte werden über Adapter-Schnittstellen bereitgestellt.

Daten-Ausgänge

Keine direkten Datenausgänge. Das Ergebnis wird über einen Adapter-Ausgang bereitgestellt.

Adapter

Richtung	Name	Typ	Kommentar
Sockets	IN1	`adapter::types::unidirectional::AR`	ADD input 1
Sockets	IN2	`adapter::types::unidirectional::AR`	ADD input 2
Sockets	IN3	`adapter::types::unidirectional::AR`	ADD input 3
Plugs	OUT	`adapter::types::unidirectional::AR`	ADD result

Funktionsweise

Der Baustein addiert die drei über die Adapter-Sockets `IN1`, `IN2` und `IN3` eingehenden Werte und stellt das Ergebnis über den Adapter-Plug `OUT` bereit. Die genaue Semantik der Addition (z. B. unterstützte Datentypen, Überlaufverhalten) wird durch die Implementierung des verwendeten Adapter-Typs `AR` festgelegt.

Der Baustein ist generisch, d. h. er kann für verschiedene Datentypen parametrisiert werden (z. B. durch Type-Hash oder generischen Klassennamen).

Technische Besonderheiten

- **Generischer Baustein:** Die tatsächliche Datentypbehandlung wird über das Attribut `eclipse4diac::core::GenericClassName` (Wert: `'GEN_AR_ADD'`) und `eclipse4diac::core::TypeHash` gesteuert.
- **Adapterbasierte Kommunikation:** Der Baustein verwendet ausschließlich Adapter (`adapter::types::unidirectional::AR`) für Ein- und Ausgabe. Dies ermöglicht eine lose Kopplung und Wiederverwendbarkeit in unterschiedlichen Kontexten.
- **Lizenz:** Unter der Eclipse Public License 2.0 (EPL-2.0) veröffentlicht.
- **Kompatibilität:** Entwickelt für die 4diac-IDE und die Eclipse 4diac Laufzeitumgebung.

Zustandsübersicht

Der Baustein besitzt kein explizites Zustandsdiagramm (ECC). Es handelt sich um eine rein kombinatorische Funktion: Die Ausgabe ist jederzeit die Summe der drei Eingänge. Es sind keine internen Zustände oder zeitlichen Abhängigkeiten vorhanden.

Anwendungsszenarien

- **Mehrfachaddition in Steuerungssystemen:** Wenn drei analoge oder digitale Werte (z. B. Sensordaten, Sollwerte) aufsummiert werden müssen.
- **Generische Bibliotheksbausteine:** Einsatz in wiederverwendbaren Funktionsbaustein-Bibliotheken, bei denen die Datentypen zur Konfigurationszeit festgelegt werden.
- **Adapterbasierte Architekturen:** In Systemen, die auf standardisierte Adapter-Schnittstellen setzen, um eine flexible Verkettung von Funktionen zu ermöglichen.

Vergleich mit ähnlichen Bausteinen

AR_ADD_2 / AR_ADD_4 etc.: Diese Bausteine unterscheiden sich lediglich in der Anzahl der Eingänge (zwei bzw. vier). `AR_ADD_3_UNGATED` ist speziell für die Addition von genau drei Werten optimiert. Der generische Ansatz erlaubt es, alle diese Varianten bei Bedarf durch Parametrisierung eines gemeinsamen Basistyps zu realisieren.

- **`AR_ADD_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

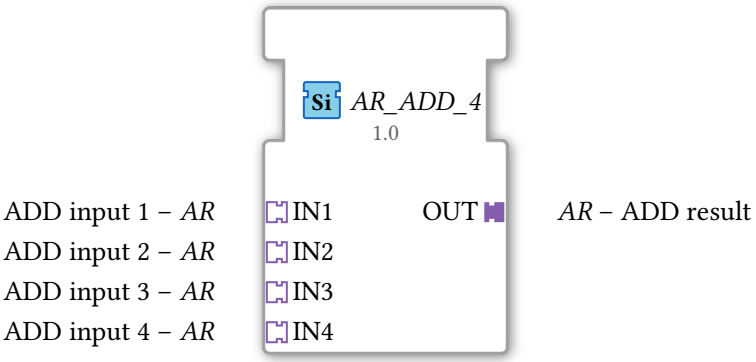
Fazit

`AR_ADD_3_UNGATED` ist ein übersichtlicher, generischer Funktionsblock für die Addition dreier Werte unter Verwendung von Adaptern. Durch die Abstraktion über generische Attribute und Adapter eignet er sich gut für IEC-61499-Anwendungen, die eine flexible und wiederverwendbare Signalverarbeitung erfordern.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AR_ADD_4



Hinweis: Ein grafisches Symbol des Funktionsblocks liegt nicht vor.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AR_ADD_4** ist ein generischer Baustein zur arithmetischen Addition mehrerer Werte. Er ist nach dem Standard IEC 61499-2 ausgelegt und erlaubt die flexible Verarbeitung von bis zu vier additiven Operanden über Adapter-Schnittstellen. Der Baustein wird typischerweise in Automatisierungssystemen eingesetzt, in denen eine numerische Summenbildung mehrerer Signale erforderlich ist.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine (Der Baustein arbeitet ohne explizite Ereignissteuerung.)

Ereignis-Ausgänge

- Keine

Daten-Eingänge

- Keine (Alle Eingabeoperanden werden über Adapter bereitgestellt.)

Daten-Ausgänge

- Keine (Das Ergebnis wird über einen Adapter ausgegeben.)

Adapter

Name	Typ	Richtung	Beschreibung
IN1	`adapter::types::unidirectional::AR`	Socket (Eingang)	Erster Summand der Addition
IN2	`adapter::types::unidirectional::AR`	Socket (Eingang)	Zweiter Summand
IN3	`adapter::types::unidirectional::AR`	Socket (Eingang)	Dritter Summand
IN4	`adapter::types::unidirectional::AR`	Socket (Eingang)	Vierter Summand
OUT	`adapter::types::unidirectional::AR`	Plug (Ausgang)	Ergebnis der Addition (Summe der Eingänge)

Die Adapter sind vom Typ `unidirectional::AR`, was darauf hindeutet, dass sie einen arithmetischen Wert (z. B. einen Zahlenwert) als gerichtete Verbindung bereitstellen oder verarbeiten.

Funktionsweise

Der Baustein **AR_ADD_4** führt die Summenbildung der vier an den Eingangsadaptern anliegenden Werte durch und gibt das Ergebnis am Ausgangsadapter **OUT** aus. Die Berechnung erfolgt nach der allgemeinen Formel:

[\text{OUT} = \text{IN1} + \text{IN2} + \text{IN3} + \text{IN4}]

Da der Baustein als generischer Typ deklariert ist (`eclipse4diac::core::GenericClassName = 'GEN_AR_ADD'`), wird der zugrunde liegende Datentyp (z. B. Integer, Real oder benutzerdefinierte arith-

metische Typen) erst zur Laufzeit durch die konkrete Konfiguration festgelegt. Die Verarbeitung erfolgt datengesteuert – sobald an allen vier Eingängen valide Werte anliegen, wird die Summe berechnet und am Ausgang aktualisiert.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generisch markiert (`eclipse4diac::core::GenericClassName`). Dadurch kann der konkrete arithmetische Datentyp erst beim Einsatz im Projekt festgelegt werden, was eine hohe Wiederverwendbarkeit ermöglicht.
- **Adapterbasierte Kommunikation**: Statt klassischer Daten-Ein-/Ausgänge werden alle Werte über Adapter ausgetauscht. Dies erlaubt eine lose Kopplung mit anderen Bausteinen und fördert modulare Strukturen.
- **Keine Ereignissteuerung**: Der Baustein besitzt keine Ereigniseingänge oder -ausgänge. Die Berechnung und Datenweitergabe erfolgt automatisch, sobald alle Eingangswerte vorliegen (ähnlich einer kontinuierlichen Funktion).
- **Typische Paketstruktur**: Der Baustein ist im Paket `adapter::iec61131::arithmetic` organisiert, was auf eine IEC 61131-nahe Adapterimplementierung hinweist.

Zustandsübersicht

Aufgrund der rein datengetriebenen und ereignislosen Arbeitsweise besitzt der Baustein **keine internen Zustände**. Es gibt keine sequenziellen Abläufe oder Zustandsmaschinen. Die Ausgabe ist immer der aktuelle Summenwert der vier Eingänge.

Anwendungsszenarien

1. **Bildung einer Summe mehrerer Prozesswerte** (z. B. Addition von Durchflusssignalen mehrerer Sensoren).
2. **Skalierung und Summation** – in Kombination mit skalierenden Adaptern kann der Baustein zur gewichteten Summenbildung eingesetzt werden.
3. **Berechnung von Gesamtverbrauch oder Gesamtenergie** aus mehreren Teilmessungen.
4. **Generische Summenbildung in modularen Automatisierungslösungen**, bei denen die Anzahl der Summanden fest, aber der Datentyp variabel ist.

Vergleich mit ähnlichen Bausteinen

Baustein	Anzahl Eingänge	Besonderheit
AR_ADD_4	4	Adapterbasiert, generisch, keine Ereignisse
AR_ADD_2 (hypothetisch)	2	Reduzierte Eingänge, gleiche Konzeption
F_ADD (aus IEC 61131)	2	Standard-Datentypen, ereignisgesteuert (via ENABLE/ENO)
AR_SUM (hypothetisch)	variabel	Flexiblere Anzahl, aber aufwändiger

Der Baustein **AR_ADD_4** hebt sich durch die reine Adapter-Kommunikation und die generische Auslegung von klassischen IEC 61499-Arithmetikbausteinen ab. Er eignet sich besonders für Systeme, die bereits auf Adapter-Technologie setzen und keine explizite Ereignissteuerung benötigen.

- `AR_ADD_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der **AR_ADD_4** ist ein kompakter, generischer Funktionsblock zur Addition von vier Werten über Adapter-Schnittstellen. Seine einfache und robuste datengetriebene Funktionsweise macht ihn zu einem nützlichen Grundbaustein für modulare Automatisierungsanwendungen. Die Abwesenheit von Ereignissen und die generische Typisierung ermöglichen eine flexible Integration in heterogene Systeme. Für

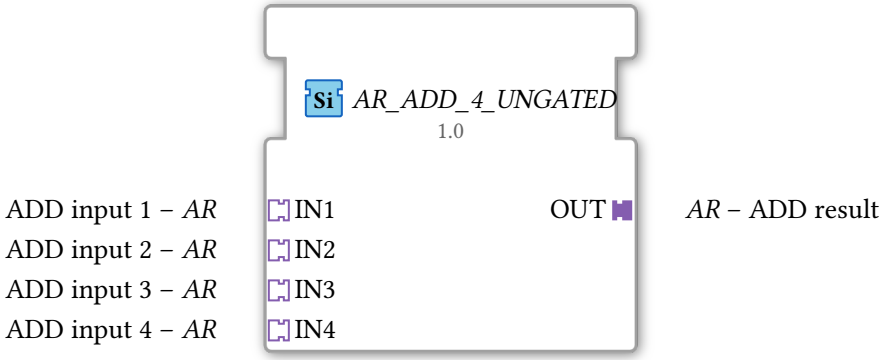
Anwendungen mit mehr oder weniger als vier Summanden muss auf angepasste Varianten oder erweiterte Bausteine zurückgegriffen werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AR_ADD_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AR_ADD_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



Hinweis: Ein grafisches Symbol des Funktionsblocks liegt nicht vor.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AR_ADD_4_UNGATED** ist ein generischer Baustein zur arithmetischen Addition mehrerer Werte. Er ist nach dem Standard IEC 61499-2 ausgelegt und erlaubt die flexible Verarbeitung von bis zu vier additiven Operanden über Adapter-Schnittstellen. Der Baustein wird typischerweise in Automatisierungssystemen eingesetzt, in denen eine numerische Summenbildung mehrerer Signale erforderlich ist.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine (Der Baustein arbeitet ohne explizite Ereignissteuerung.)

Ereignis-Ausgänge

- Keine

Daten-Eingänge

- Keine (Alle Eingabeoperanden werden über Adapter bereitgestellt.)

Daten-Ausgänge

- Keine (Das Ergebnis wird über einen Adapter ausgegeben.)

Adapter

Name	Typ	Richtung	Beschreibung
IN1	`adapter::types::unidirectional::AR`	Socket (Eingang)	Erster Summand der Addition

IN2	`adapter::types::unidirectional::AR`	Socket (Eingang)	Zweiter Summand
IN3	`adapter::types::unidirectional::AR`	Socket (Eingang)	Dritter Summand
IN4	`adapter::types::unidirectional::AR`	Socket (Eingang)	Vierter Summand
OUT	`adapter::types::unidirectional::AR`	Plug (Ausgang)	Ergebnis der Addition (Summe der Eingänge)

Die Adapter sind vom Typ `unidirectional::AR`, was darauf hindeutet, dass sie einen arithmetischen Wert (z. B. einen Zahlenwert) als gerichtete Verbindung bereitstellen oder verarbeiten.

Funktionsweise

Der Baustein **AR_ADD_4_UNGATED** führt die Summenbildung der vier an den Eingangsadaptern anliegenden Werte durch und gibt das Ergebnis am Ausgangsadapter **OUT** aus. Die Berechnung erfolgt nach der allgemeinen Formel:

[\text{OUT} = \text{IN1} + \text{IN2} + \text{IN3} + \text{IN4}]

Da der Baustein als generischer Typ deklariert ist (`eclipse4diac::core::GenericClassName = 'GEN_AR_ADD'`), wird der zugrunde liegende Datentyp (z. B. Integer, Real oder benutzerdefinierte arithmetische Typen) erst zur Laufzeit durch die konkrete Konfiguration festgelegt. Die Verarbeitung erfolgt datengesteuert – sobald an allen vier Eingängen valide Werte anliegen, wird die Summe berechnet und am Ausgang aktualisiert.

Technische Besonderheiten

- **Generischer Baustein**: Der FB ist als generisch markiert (`eclipse4diac::core::GenericClassName`). Dadurch kann der konkrete arithmetische Datentyp erst beim Einsatz im Projekt festgelegt werden, was eine hohe Wiederverwendbarkeit ermöglicht.
- **Adapterbasierte Kommunikation**: Statt klassischer Daten-Ein-/Ausgänge werden alle Werte über Adapter ausgetauscht. Dies erlaubt eine lose Kopplung mit anderen Bausteinen und fördert modulare Strukturen.
- **Keine Ereignissteuerung**: Der Baustein besitzt keine Ereigniseingänge oder -ausgänge. Die Berechnung und Datenweitergabe erfolgt automatisch, sobald alle Eingangswerte vorliegen (ähnlich einer kontinuierlichen Funktion).
- **Typische Paketstruktur**: Der Baustein ist im Paket `adapter::iec61131::arithmetic` organisiert, was auf eine IEC 61131-nahe Adapterimplementierung hinweist.

Zustandsübersicht

Aufgrund der rein datengetriebenen und ereignislosen Arbeitsweise besitzt der Baustein **keine internen Zustände**. Es gibt keine sequenziellen Abläufe oder Zustandsmaschinen. Die Ausgabe ist immer der aktuelle Summenwert der vier Eingänge.

Anwendungsszenarien

1. **Bildung einer Summe mehrerer Prozesswerte** (z. B. Addition von Durchflusssignalen mehrerer Sensoren).
2. **Skalierung und Summation** – in Kombination mit skalierenden Adaptern kann der Baustein zur gewichteten Summenbildung eingesetzt werden.
3. **Berechnung von Gesamtverbrauch oder Gesamtenergie** aus mehreren Teilmessungen.
4. **Generische Summenbildung in modularen Automatisierungslösungen**, bei denen die Anzahl der Summanden fest, aber der Datentyp variabel ist.

Vergleich mit ähnlichen Bausteinen

Baustein	Anzahl Eingänge	Besonderheit
AR_ADD_4_UNGATED	4	Adapterbasiert, generisch, keine Ereignisse
AR_ADD_2 (hypothetisch)	2	Reduzierte Eingänge, gleiche Konzeption
F_ADD (aus IEC 61131)	2	Standard-Datentypen, ereignisgesteuert (via ENABLE/ENO)
AR_SUM (hypothetisch)	variabel	Flexiblere Anzahl, aber aufwändiger

Der Baustein **AR_ADD_4_UNGATED** hebt sich durch die reine Adapter-Kommunikation und die generische Auslegung von klassischen IEC 61499-Arithmetikbausteinen ab. Er eignet sich besonders für Systeme, die bereits auf Adapter-Technologie setzen und keine explizite Ereignissteuerung benötigen.

- `AR_ADD_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

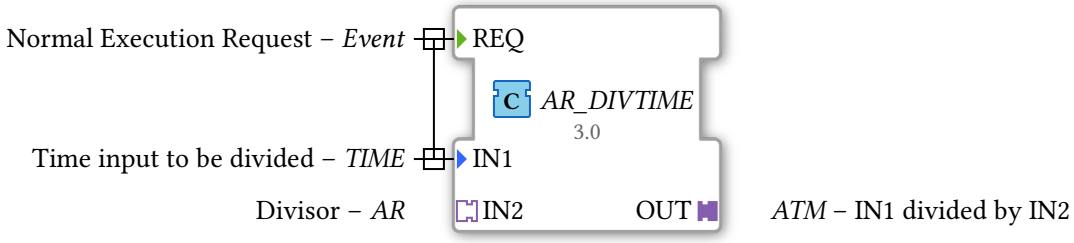
Fazit

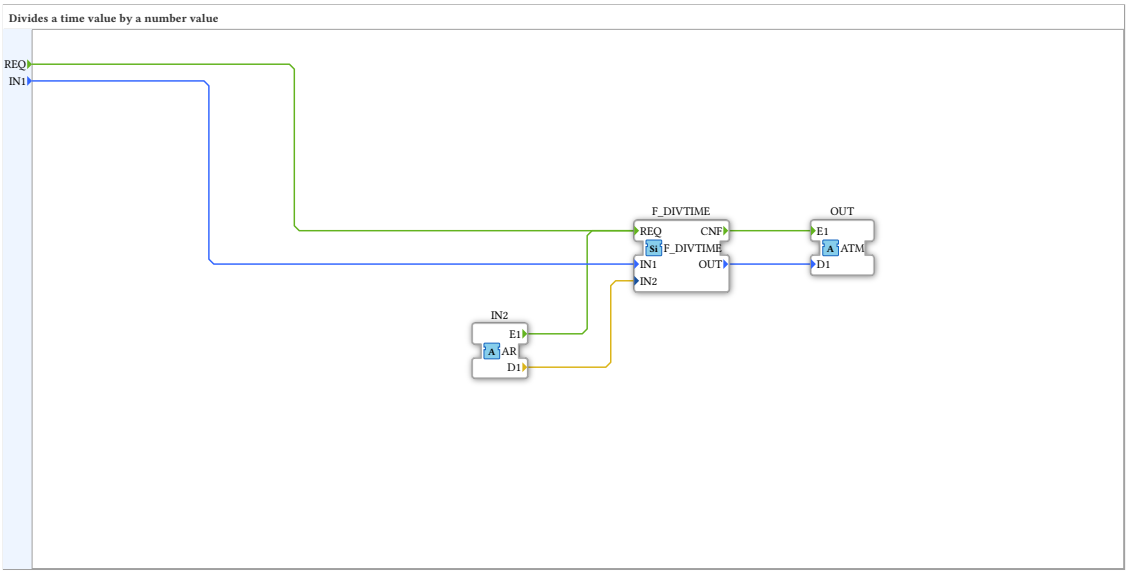
Der **AR_ADD_4_UNGATED** ist ein kompakter, generischer Funktionsblock zur Addition von vier Werten über Adapter-Schnittstellen. Seine einfache und robuste datengetriebene Funktionsweise macht ihn zu einem nützlichen Grundbaustein für modulare Automatisierungsanwendungen. Die Abwesenheit von Ereignissen und die generische Typisierung ermöglichen eine flexible Integration in heterogene Systeme. Für Anwendungen mit mehr oder weniger als vier Summanden muss auf angepasste Varianten oder erweiterte Bausteine zurückgegriffen werden.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AR_DIVTIME





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AR_DIVTIME** dient der Division eines Zeitwertes (Typ TIME) durch einen numerischen Wert. Die Eingabe des Divisors erfolgt über einen Adapter, sodass der Baustein flexibel in verschiedenen Kontexten eingesetzt werden kann. Die Ausgabe des Ergebnisses geschieht ebenfalls über einen Adapter. Der Baustein ist gemäß IEC 61131-3 als standard arithmetic function klassifiziert.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Event): Normaler Ausführungsanstoß. Bei einem Ereignis an REQ wird die Division durchgeführt. Der Eingang IN1 muss dabei mit dem REQ-Ereignis verbunden sein (WITH).

Ereignis-Ausgänge

Der Baustein besitzt keine expliziten Ereignisausgänge in der Interface-Liste. Das Ergebnis wird jedoch über den Adapter-Ausgang **OUT** bereitgestellt, welcher ein Ereignis (E1) sendet, sobald die Berechnung abgeschlossen ist (wie aus der internen Verdrahtung ersichtlich).

Daten-Eingänge

- **IN1** (TIME): Zu dividierender Zeitwert (Dividend).

Daten-Ausgänge

Der Baustein hat keine direkten Datenausgänge. Das Divisionsergebnis wird über den Adapter-Ausgang **OUT** als Datenwert (D1) ausgegeben.

Adapter

- **IN2** (Adapter, Typ `adapter::types::unidirectional::AR`): Divisor. Dieser Adapter liefert ein Ereignis (E1) zur Auslösung sowie den Datenwert (D1).
- **OUT** (Adapter, Typ `adapter::types::unidirectional::ATM`): Ausgang für das Divisionsergebnis (TIME). Bereitstellung von Ereignis (E1) und Daten (D1).

Funktionsweise

Der Funktionsblock AR_DIVTIME kapselt die Funktionalität des internen Bausteins **F_DIVTIME** (Typ `iec61131::arithmetic::F_DIVTIME`). Bei einem Ereignis am Eingang **REQ** oder alternativ am Adaptereingang **IN2.E1** wird die Division gestartet. Der interne FB dividiert **IN1** (TIME) durch **IN2.D1** (numerischer Wert) und gibt das Ergebnis auf **OUT.D1** aus. Sobald die Berechnung abgeschlossen ist, wird ein Ereignis an **OUT.E1** gesendet, welches dem nachfolgenden Empfänger signalisiert, dass das Ergebnis gültig ist.

Technische Besonderheiten

- **Adapter-basierte Schnittstelle:** Statt klassischer Daten-Ein-/Ausgänge werden Adapter verwendet, was eine flexible Verschaltung in komplexen Netzwerken erlaubt. Die Adapter sind als unidirektional definiert und erfordern kompatible Gegenstücke.
- **Standardkonformität:** Der Baustein erfüllt die Anforderungen der IEC 61131-3 (Klassifikation: standard arithmetic function).
- **Interne Kapselung:** Die eigentliche Berechnung erfolgt durch den wiederverwendbaren Baustein F_DIVTIME, sodass AR_DIVTIME als eine Art Wrapper mit adaptergestützter Kommunikation fungiert.
- **Gemischte Schnittstelle:** Anders als vollständig adapterbasierte Bausteine erhält AR_DIVTIME den Dividenden (IN1) als klassischen Dateneingang, nicht über einen Adapter – siehe `ATM_AR_DIVTIME` für die vollständig adapterbasierte Variante.

Zustandsübersicht

Der Baustein selbst besitzt keine explizite Zustandsmaschine. Die ereignisgesteuerte Abarbeitung erfolgt deterministisch:

- Warten auf Ereignis an **REQ** oder **IN2.E1**
- Ausführen der Division
- Senden des Ergebnisses an **OUT.E1** und **OUT.D1**
- Zurückkehren in den Wartezustand

Anwendungsszenarien

- Aufteilen eines Zeitwertes in gleiche Abschnitte, z. B. Umrechnung einer Gesamtlaufzeit auf einzelne Zyklen.
- In Steuerungsanwendungen, bei denen ein Zeitparameter dynamisch durch einen variablen Faktor geteilt werden muss.

- Als Bestandteil von Bibliotheken für IEC 61131-3-konforme automatische Steuerung, insbesondere wenn Adapter zur Entkopplung der Signalpfade eingesetzt werden.

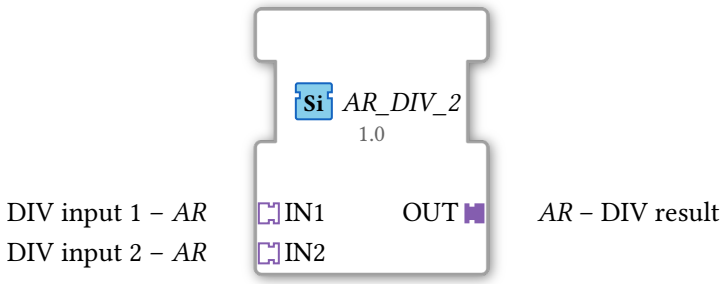
Vergleich mit ähnlichen Bausteinen

- **F_DIVTIME:** Der direkte, adapterfreie Baustein zur Division von TIME durch numerische Werte. Er bietet klassische Ereignis- und Datenschnittstellen. AR_DIVTIME kapselt diesen Baustein und stellt den Divisor über einen Adapter bereit.
- **AR_MULTIME:** Multipliziert statt zu dividieren, ansonsten strukturell identisch (gemischte Schnittstelle: IN1 als Dateneingang, IN2 als Adapter).
- **ATM_AR_DIVTIME:** Die vollständig adapterbasierte Variante – hier ist auch IN1 ein Adapter-Socket (Typ `ATM`) statt eines festen Dateneingangs.

Fazit

Der Funktionsblock **AR_DIVTIME** ermöglicht die Division eines Zeitwertes durch einen numerischen Faktor über eine adaptergestützte Schnittstelle für den Divisor. Er kombiniert die bewährte IEC 61131-3-Arithmetik mit flexibler Konnektivität und eignet sich besonders für modulare Automatisierungssysteme. Die interne Verwendung des Bausteins F_DIVTIME garantiert eine zuverlässige und effiziente Berechnung.

AR_DIV_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AR_DIV_2` ist ein generischer arithmetischer Baustein, der für die Durchführung von Divisionsoperationen (`DIV`) in IEC 61499-basierten Systemen entwickelt wurde. Im Gegensatz zu klassischen Funktionsbausteinen nutzt dieser Baustein Adapter (`unidirectional::AR`), um Daten und Ereignisse zu bündeln. Dies sorgt für eine übersichtlichere und modularere visuelle Programmierung innerhalb der 4diac-IDE.

Schnittstellenstruktur

Da es sich um einen rein adapterbasierten Funktionsbaustein handelt, verfügt `AR_DIV_2` über keine direkten, klassischen Daten- oder Ereignis-Ein- und Ausgänge auf der Hauptebene. Die gesamte Kommunikation wird über Adapter abgewickelt.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. (Die Ereignissteuerung erfolgt implizit über die Eingangs-Adapter).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. (Die Ereignissteuerung erfolgt implizit über den Ausgangs-Adapter).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden. (Die Datenübergabe erfolgt über die Eingangs-Adapter).

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden. (Die Datenübergabe erfolgt über den Ausgangs-Adapter).

Adapter

Typ	Name	Adaptertyp	Beschreibung
Socket (Eingang)	`IN1`	`adapter::types::unidirectional::AR`	Erster Eingangswert (Dividend) für die Division.
Socket (Eingang)	`IN2`	`adapter::types::unidirectional::AR`	Zweiter Eingangswert (Divisor) für die Division.
Plug (Ausgang)	`OUT`	`adapter::types::unidirectional::AR`	Das Ergebnis der Division (Quotient).

Funktionsweise

Der Baustein führt eine mathematische Division der über die Adapter bereitgestellten Werte aus. Sobald ein Aktualisierungsereignis an den Eingangs-Adaptern (`IN1` oder `IN2`) signalisiert, dass neue Daten vorliegen, wird die Berechnung getriggert:

$$\mathrm{OUT} = \frac{\mathrm{IN1}}{\mathrm{IN2}} \tag{2.66}$$

Das berechnete Ergebnis wird anschließend an den Ausgangs-Adapter `OUT` übergeben und das entsprechende Sende-Ereignis des Adapters ausgelöst, um nachfolgende Bausteine über den neuen Wert zu informieren.

Technische Besonderheiten

- **Generischer Typ (`GEN_AR_DIV`):** Der Baustein ist intern als generischer Typ deklariert. Dies ermöglicht eine flexible Anpassung an verschiedene Datentypen (z. B. `REAL`, `LREAL`, `INT`), je nachdem, wie der zugrunde liegende `AR`-Adaptertyp definiert ist.
- **Adapter-Kopplung:** Durch die Verwendung von unidirektionalen Adaptern (`unidirectional::AR`) wird die Anzahl der Verbindungslinien im Funktionsplan drastisch reduziert, was die Lesbarkeit komplexer mathematischer Berechnungen erhöht.
- **Schutz vor Division durch Null:** Systemintegrierte Schutzmaßnahmen hängen von der spezifischen Laufzeitumgebung (Runtime) ab. Bei der Verwendung des Bausteins ist darauf zu achten, dass der Divisor (`IN2`) ungleich Null ist, um Berechnungsfehler oder Laufzeitabstürze zu vermeiden.

Zustandsübersicht

Der Baustein verhält sich im Wesentlichen zustandslos (stateless) und reagiert ereignisgesteuert:

1. **Wartezustand (Idle):** Der Baustein wartet auf ein Aktualisierungsereignis an den Adaptern `IN1` oder `IN2`.
2. **Berechnung:** Nach Eintreffen eines Ereignisses wird die Division ausgeführt.
3. **Aktualisierung:** Das Ergebnis wird an `OUT` angelegt und das Ausgangsereignis getriggert. Der Baustein kehrt sofort in den Wartezustand zurück.

Anwendungsszenarien

- **Messwertskalierung:** Division von Sensorrohdaten durch einen festen Skalierungsfaktor zur Umrechnung in physikalische Einheiten.
- **Mittelwertberechnung:** Division einer Summe durch die Anzahl der Messwerte in komplexeren Berechnungsnetzwerken.
- **Verhältnisberechnungen:** Bestimmung von prozentualen Anteilen oder Verhältnissen in verfahrenstechnischen Anlagen.

Vergleich mit ähnlichen Bausteinen

- **Klassischer `DIV`-Baustein:** Der klassische IEC 61131-3 `DIV`-Baustein besitzt dedizierte Pins für `IN1`, `IN2` und `OUT` sowie die dazugehörigen `REQ`- und `CNF`-Events. `AR_DIV_2` bündelt diese Signale in Adaptern, was die Verdrahtung in großen Projekten vereinfacht.
- **`AR_MUL_2` / `AR_ADD_2`:** Diese Bausteine teilen sich dieselbe Adapter-Philosophie, führen jedoch Multiplikationen bzw. Additionen durch. Sie lassen sich nahtlos mit `AR_DIV_2` zu mathematischen Berechnungsbäumen verketten.

- `AR_DIV_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

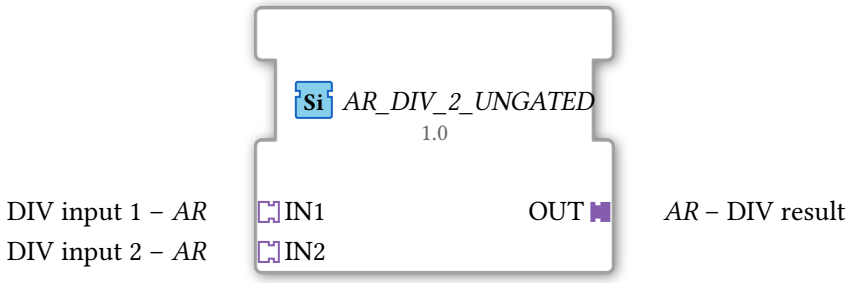
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `AR_DIV_2` ist ein hocheffizienter, modularer Baustein für arithmetische Divisionen. Durch konsequenten Einsatz von Adaptern eignet er sich hervorragend für saubere, übersichtliche Software-Architekturen in der verteilten Automatisierung nach dem IEC 61499 Standard.

AR_DIV_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AR_DIV_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AR_DIV_2_UNGATED` ist ein generischer arithmetischer Baustein, der für die Durchführung von Divisionsoperationen (`DIV`) in IEC 61499-basierten Systemen entwickelt wurde. Im Gegensatz zu klassischen Funktionsbausteinen nutzt dieser Baustein Adapter (`unidirectional::AR`), um Daten und Ereignisse zu bündeln. Dies sorgt für eine übersichtlichere und modularere visuelle Programmierung innerhalb der 4diac-IDE.

Schnittstellenstruktur

Da es sich um einen rein adapterbasierten Funktionsbaustein handelt, verfügt `AR_DIV_2_UNGATED` über keine direkten, klassischen Daten- oder Ereignis-Ein- und Ausgänge auf der Hauptebene. Die gesamte Kommunikation wird über Adapter abgewickelt.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. (Die Ereignissteuerung erfolgt implizit über die Eingangs-Adapter).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. (Die Ereignissteuerung erfolgt implizit über den Ausgangs-Adapter).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden. (Die Datenübergabe erfolgt über die Eingangs-Adapter).

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden. (Die Datenübergabe erfolgt über den Ausgangs-Adapter).

Adapter

Typ	Name	Adaptertyp	Beschreibung
Socket (Eingang)	`IN1`	`adapter::types::unidirectional::AR`	Erster Eingangswert (Dividend) für die Division.
Socket (Eingang)	`IN2`	`adapter::types::unidirectional::AR`	Zweiter Eingangswert (Divisor) für die Division.
Plug (Ausgang)	`OUT`	`adapter::types::unidirectional::AR`	Das Ergebnis der Division (Quotient).

Funktionsweise

Der Baustein führt eine mathematische Division der über die Adapter bereitgestellten Werte aus. Sobald ein Aktualisierungsereignis an den Eingangs-Adaptern (`IN1` oder `IN2`) signalisiert, dass neue Daten vorliegen, wird die Berechnung getriggert:

$$\mathrm{OUT} = \frac{\mathrm{IN1}}{\mathrm{IN2}} \tag{2.67}$$

Das berechnete Ergebnis wird anschließend an den Ausgangs-Adapter `OUT` übergeben und das entsprechende Sende-Ereignis des Adapters ausgelöst, um nachfolgende Bausteine über den neuen Wert zu informieren.

Technische Besonderheiten

- **Generischer Typ (`GEN_AR_DIV`):** Der Baustein ist intern als generischer Typ deklariert. Dies ermöglicht eine flexible Anpassung an verschiedene Datentypen (z. B. `REAL`, `LREAL`, `INT`), je nachdem, wie der zugrunde liegende `AR`-Adaptertyp definiert ist.
- **Adapter-Kopplung:** Durch die Verwendung von unidirektionalen Adaptern (`unidirectional::AR`) wird die Anzahl der Verbindungslinien im Funktionsplan drastisch reduziert, was die Lesbarkeit komplexer mathematischer Berechnungen erhöht.
- **Schutz vor Division durch Null:** Systemintegrierte Schutzmaßnahmen hängen von der spezifischen Laufzeitumgebung (Runtime) ab. Bei der Verwendung des Bausteins ist darauf zu achten, dass der Divisor (`IN2`) ungleich Null ist, um Berechnungsfehler oder Laufzeitabstürze zu vermeiden.

Zustandsübersicht

Der Baustein verhält sich im Wesentlichen zustandslos (stateless) und reagiert ereignisgesteuert:

1. **Wartezustand (Idle):** Der Baustein wartet auf ein Aktualisierungsereignis an den Adaptern `IN1` oder `IN2`.
2. **Berechnung:** Nach Eintreffen eines Ereignisses wird die Division ausgeführt.
3. **Aktualisierung:** Das Ergebnis wird an `OUT` angelegt und das Ausgangsereignis getriggert. Der Baustein kehrt sofort in den Wartezustand zurück.

Anwendungsszenarien

- **Messwertskalierung:** Division von Sensorrohdaten durch einen festen Skalierungsfaktor zur Umrechnung in physikalische Einheiten.
- **Mittelwertberechnung:** Division einer Summe durch die Anzahl der Messwerte in komplexeren Berechnungsnetzwerken.
- **Verhältnisberechnungen:** Bestimmung von prozentualen Anteilen oder Verhältnissen in verfahrenstechnischen Anlagen.

Vergleich mit ähnlichen Bausteinen

- **Klassischer `DIV`-Baustein:** Der klassische IEC 61131-3 `DIV`-Baustein besitzt dedizierte Pins für `IN1`, `IN2` und `OUT` sowie die dazugehörigen `REQ`- und `CNF`-Events. `AR_DIV_2_UNGATED` bündelt diese Signale in Adaptern, was die Verdrahtung in großen Projekten vereinfacht.

- **AR_MUL_2 / AR_ADD_2:** Diese Bausteine teilen sich dieselbe Adapter-Philosophie, führen jedoch Multiplikationen bzw. Additionen durch. Sie lassen sich nahtlos mit AR_DIV_2_UNGATED zu mathematischen Berechnungsbäumen verketten.
- **AR_DIV_2_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- **AR_DIV_2**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

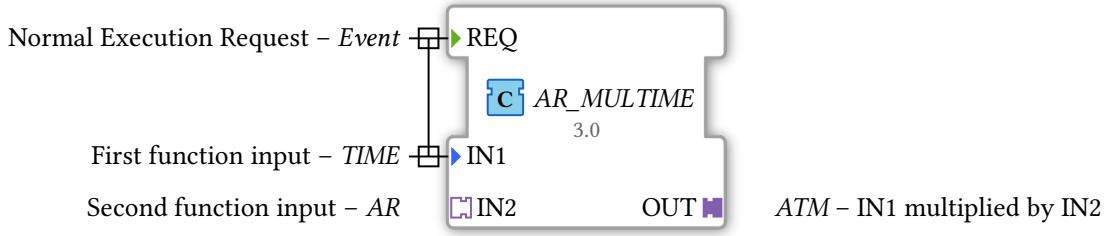
Änderungserkennung

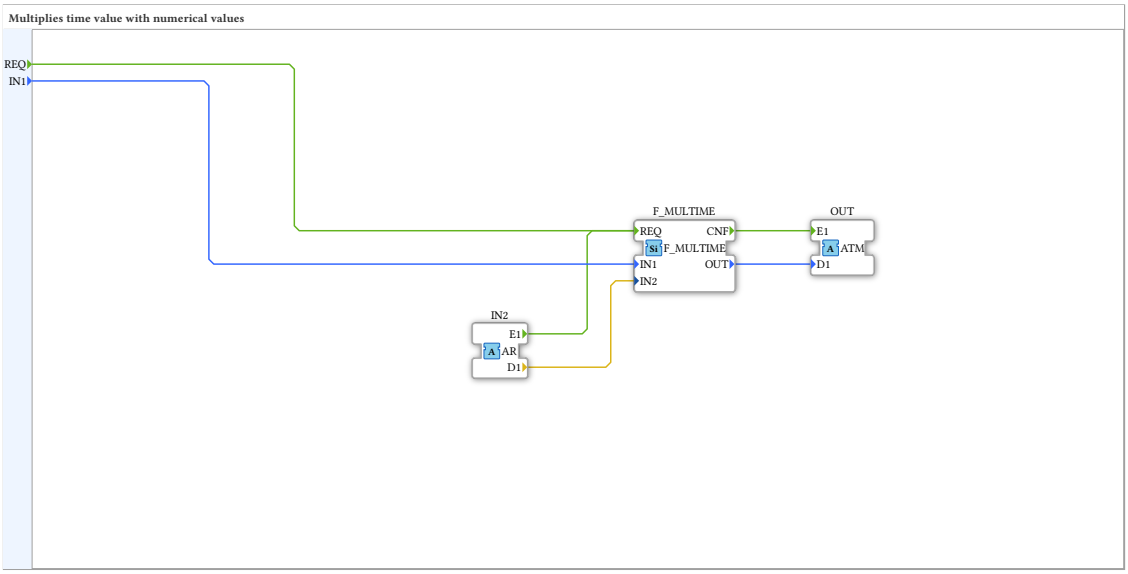
Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der AR_DIV_2_UNGATED ist ein hocheffizienter, modularer Baustein für arithmetische Divisionen. Durch konsequenten Einsatz von Adaptern eignet er sich hervorragend für saubere, übersichtliche Software-Architekturen in der verteilten Automatisierung nach dem IEC 61499 Standard.

AR_MULTIME





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AR_MULTIME** dient der Multiplikation eines Zeitwertes (Typ TIME) mit einem numerischen Wert. Die Eingabe des numerischen Faktors erfolgt über einen Adapter, sodass der Baustein flexibel in verschiedenen Kontexten eingesetzt werden kann. Die Ausgabe des Ergebnisses geschieht ebenfalls über einen Adapter. Der Baustein ist gemäß IEC 61131-3 als standard arithmetic function klassifiziert.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Event): Normaler Ausführungsanstoß. Bei einem Ereignis an REQ wird die Multiplikation durchgeführt. Der Eingang IN1 muss dabei mit dem REQ-Ereignis verbunden sein (WITH).

Ereignis-Ausgänge

Der Baustein besitzt keine expliziten Ereignisausgänge in der Interface-Liste. Das Ergebnis wird jedoch über den Adapter-Ausgang **OUT** bereitgestellt, welcher ein Ereignis (E1) sendet, sobald die Berechnung abgeschlossen ist (wie aus der internen Verdrahtung ersichtlich).

Daten-Eingänge

- **IN1** (TIME): Erster Funktionsinput – der zu multiplizierende Zeitwert.

Daten-Ausgänge

Der Baustein hat keine direkten Datenausgänge. Das Multiplikationsergebnis wird über den Adapter-Ausgang **OUT** als Datenwert (D1) ausgegeben.

Adapter

- **IN2** (Adapter, Typ `adapter::types::unidirectional::AR`): Zweiter Funktionsinput – der numerische Multiplikator. Dieser Adapter liefert ein Ereignis (E1) zur Auslösung sowie den Datenwert (D1).
- **OUT** (Adapter, Typ `adapter::types::unidirectional::ATM`): Ausgang für das multiplizierte Ergebnis (TIME). Bereitstellung von Ereignis (E1) und Daten (D1).

Funktionsweise

Der Funktionsblock AR_MULTIME kapselt die Funktionalität des internen Bausteins **F_MULTIME** (Typ `iec61131::arithmetic::F_MULTIME`).
Bei einem Ereignis am Eingang **REQ** oder alternativ am Adaptereingang **IN2.E1** wird die Multiplikation gestartet. Der interne FB multipliziert **IN1** (TIME) mit **IN2.D1** (numerischer Wert) und gibt das Ergebnis auf **OUT.D1** aus.
Sobald die Berechnung abgeschlossen ist, wird ein Ereignis an **OUT.E1** gesendet, welches dem nachfolgenden Empfänger signalisiert, dass das Ergebnis gültig ist.

Technische Besonderheiten

- **Adapter-basierte Schnittstelle:** Statt klassischer Daten-Ein-/Ausgänge werden Adapter verwendet, was eine flexible Verschaltung in komplexen Netzwerken erlaubt. Die Adapter sind als unidirektional definiert und erfordern kompatible Gegenstücke.
- **Standardkonformität:** Der Baustein erfüllt die Anforderungen der IEC 61131-3 (Klassifikation: standard arithmetic function).
- **Interne Kapselung:** Die eigentliche Berechnung erfolgt durch den wiederverwendbaren Baustein F_MULTIME, sodass AR_MULTIME als eine Art Wrapper mit adaptergestützter Kommunikation fungiert.

Zustandsübersicht

Der Baustein selbst besitzt keine explizite Zustandsmaschine. Die ereignisgesteuerte Abarbeitung erfolgt deterministisch:

- Warten auf Ereignis an **REQ** oder **IN2.E1**
- Ausführen der Multiplikation
- Senden des Ergebnisses an **OUT.E1** und **OUT.D1**
- Zurückkehren in den Wartezustand

Anwendungsszenarien

- Skalierung von Zeitwerten durch einen konstanten oder variablen Faktor, z. B. Umrechnung von Sekunden in Millisekunden oder Verlängerung von Verzögerungszeiten.
- In Steuerungsanwendungen, bei denen Zeitparameter dynamisch angepasst werden müssen, etwa in Verbindung mit PID-Reglern oder Zeitplanungen.
- Als Bestandteil von Bibliotheken für IEC 61131-3-konforme automatische Steuerung, insbesondere wenn Adapter zur Entkopplung der Signalpfade eingesetzt werden.

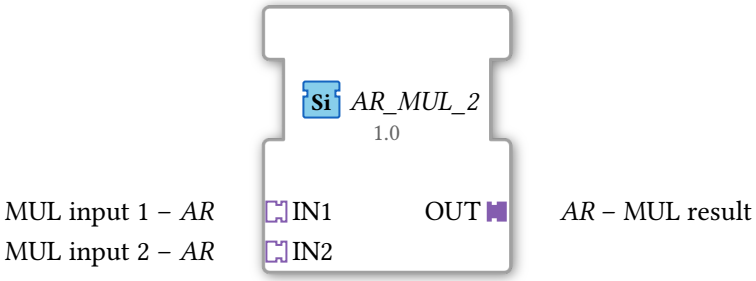
Vergleich mit ähnlichen Bausteinen

- **F_MULTIME:** Der direkte, adapterfreie Baustein zur Multiplikation von TIME mit numerischen Werten. Er bietet klassische Ereignis- und Datenschnittstellen. AR_MULTIME hingegen kapselt diesen Baustein und stellt die Schnittstellen über Adapter bereit, was die Wiederverwendbarkeit in adapterbasierten Architekturen erhöht.
- **AR_MUL:** Ein allgemeiner Multiplikationsbaustein mit Adaptern, der für beliebige numerische Typen ausgelegt sein könnte. AR_MULTIME ist speziell auf TIME-Werte spezialisiert.

Fazit

Der Funktionsblock **AR_MULTIME** ermöglicht die Multiplikation eines Zeitwertes mit einem numerischen Faktor über eine moderne, adaptergestützte Schnittstelle. Er kombiniert die bewährte IEC 61131-3-Arithmetik mit flexibler Konnektivität und eignet sich besonders für modulare Automatisierungssysteme, die auf standardisierte Adapter setzen. Die interne Verwendung des Bausteins F_MULTIME garantiert eine zuverlässige und effiziente Berechnung.

AR_MUL_2



(Kein Bild vorhanden)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein (FB) **AR_MUL_2** ist ein generischer Arithmetik-Baustein für die 4diac-IDE. Er dient zur Multiplikation von zwei Werten, die über unidirektionale Adapter übertragen werden. Durch die Kapselung der mathematischen Operation in eine Adapter-Schnittstelle ermöglicht dieser Baustein eine saubere, modulare und übersichtliche Modellierung von Berechnungen innerhalb von IEC 61499 Anwendungen.

Schnittstellenstruktur

Der Baustein besitzt keine klassischen ereignis- oder datenbasierten Ein- und Ausgänge. Die gesamte Kommunikation wird über Adapter abgewickelt.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. (Die Ereignissteuerung wird über die Adapter realisiert.)

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AR`): Erster Eingangswert (Multiplikand 1) für die Multiplikation.

- **IN2** (Typ: `adapter::types::unidirectional::AR`): Zweiter Eingangswert (Multiplikand 2) für die Multiplikation.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AR`): Ausgang für das berechnete Produkt der Multiplikation ($OUT = IN1 \times IN2$).

Funktionsweise

Sobald über die Eingangs-Adapter `IN1` und `IN2` neue Werte oder entsprechende Trigger-Ereignisse eintreffen, multipliziert der Funktionsbaustein die Werte der beiden Eingänge. Das Ergebnis der mathematischen Operation ($IN1 \cdot IN2$) wird unmittelbar an den Ausgangs-Adapter `OUT` übergeben und für nachfolgende Bausteine bereitgestellt.

Da es sich um einen generischen Funktionsbaustein (`GEN_AR_MUL`) handelt, ist die Implementierung flexibel gegenüber den konkret verwendeten Datentypen innerhalb der `AR`-Adapterstruktur.

Technische Besonderheiten

- **Generischer Baustein:** Durch das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `'GEN_AR_MUL'` ist der Baustein für verschiedene numerische Datentypen einsetzbar, sofern die Adapter diese unterstützen.
- **Unidirektionale Adapter:** Die Verwendung des Typs `unidirectional::AR` sorgt für einen klaren Datenfluss ohne Rückkopplungsschleifen, was die Stabilität der Applikation erhöht und die Latenz minimiert.
- **Compiler-Zugehörigkeit:** Der FB ist dem Package `adapter::iec61131::arithmetic` zugeordnet und konform zum Standard IEC 61499-2.

Zustandsübersicht

Da dieser Funktionsbaustein primär datenflussgesteuert über Adapter arbeitet, besitzt er kein komplexes internes Zustandsdiagramm (ECC). Er agiert als zustandsloses Übertragungsglied:

1. **Warten:** Baustein wartet auf Datenaktualisierungen an den Adaptern `IN1` und/oder `IN2`.
2. **Berechnung:** Bei Wertänderung wird das Produkt gebildet.
3. **Ausgabe:** Das Ergebnis wird direkt an den Ausgang `OUT` weitergeleitet.

Anwendungsszenarien

- **Messwertskalierung:** Multiplikation eines analogen Sensorwertes (z. B. einer Spannung) mit einem Skalierungsfaktor zur Umrechnung in eine physikalische Einheit.
- **Signalverstärkung:** Proportionale Verstärkung von Regelsignalen in der Prozessautomatisierung.
- **Mathematische Berechnungen:** Einsatz als modularer Baustein in komplexeren Berechnungsnetzwerken innerhalb von 4diac-Applikationen.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-Multiplikationsbausteinen (wie dem klassischen `MUL`-Baustein nach IEC 61131-3), die mit elementaren Datentypen (z.B. `INT`, `REAL`) arbeiten, nutzt `AR_MUL_2` strukturierte **Adapter**. Dies hat folgende Vorteile:

- **Geringerer Verdrahtungsaufwand:** Signale und dazugehörige Steuerungsereignisse werden in einer einzigen Adapterverbindung gebündelt.
- **Bessere Lesbarkeit:** Komplexe Steuerungsdiagramme bleiben übersichtlich, da weniger Einzelverbindungen auf der Arbeitsfläche gezogen werden müssen.

- `AR_MUL_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das

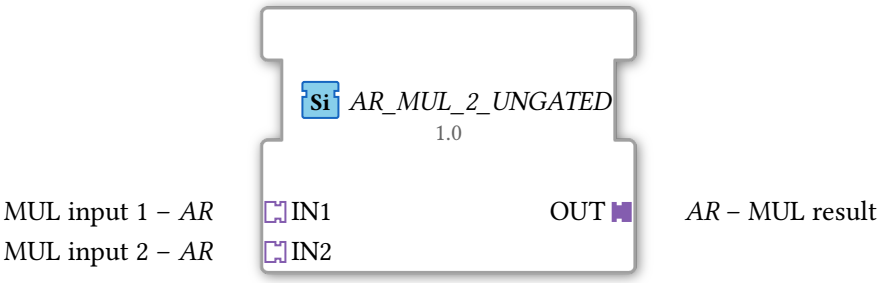
Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `AR_MUL_2`-Funktionsbaustein ist ein effizientes Werkzeug für arithmetische Operationen in modernen, adapterbasierten Steuerungsprogrammen. Durch die Kapselung in unidirektionale Schnittstellen bietet er ein hohes Maß an Wiederverwendbarkeit und sorgt für ein sauberes und strukturiertes Design innerhalb der 4diac-Umgebung.

AR_MUL_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AR_MUL_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild vorhanden)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein (FB) **AR_MUL_2_UNGATED** ist ein generischer Arithmetik-Baustein für die 4diac-IDE. Er dient zur Multiplikation von zwei Werten, die über unidirektionale Adapter übertragen werden. Durch die Kapselung der mathematischen Operation in eine Adapter-Schnittstelle ermöglicht dieser Baustein eine saubere, modulare und übersichtliche Modellierung von Berechnungen innerhalb von IEC 61499 Anwendungen.

Schnittstellenstruktur

Der Baustein besitzt keine klassischen ereignis- oder datenbasierten Ein- und Ausgänge. Die gesamte Kommunikation wird über Adapter abgewickelt.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. (Die Ereignissteuerung wird über die Adapter realisiert.)

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AR`): Erster Eingangswert (Multiplikand 1) für die Multiplikation.
- **IN2** (Typ: `adapter::types::unidirectional::AR`): Zweiter Eingangswert (Multiplikand 2) für die Multiplikation.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AR`): Ausgang für das berechnete Produkt der Multiplikation ($OUT = IN1 \times IN2$).

Funktionsweise

Sobald über die Eingangs-Adapter `IN1` und `IN2` neue Werte oder entsprechende Trigger-Ereignisse eintreffen, multipliziert der Funktionsbaustein die Werte der beiden Eingänge. Das Ergebnis der mathematischen Operation ($IN1 \cdot IN2$) wird unmittelbar an den Ausgangs-Adapter `OUT` übergeben und für nachfolgende Bausteine bereitgestellt.

Da es sich um einen generischen Funktionsbaustein (`GEN_AR_MUL`) handelt, ist die Implementierung flexibel gegenüber den konkret verwendeten Datentypen innerhalb der `AR`-Adapterstruktur.

Technische Besonderheiten

- **Generischer Baustein:** Durch das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `'GEN_AR_MUL'` ist der Baustein für verschiedene numerische Datentypen einsetzbar, sofern die Adapter diese unterstützen.
- **Unidirektionale Adapter:** Die Verwendung des Typs `unidirectional::AR` sorgt für einen klaren Datenfluss ohne Rückkopplungsschleifen, was die Stabilität der Applikation erhöht und die Latenz minimiert.
- **Compiler-Zugehörigkeit:** Der FB ist dem Package `adapter::iec61131::arithmetic` zugeordnet und konform zum Standard IEC 61499-2.

Zustandsübersicht

Da dieser Funktionsbaustein primär datenflussgesteuert über Adapter arbeitet, besitzt er kein komplexes internes Zustandsdiagramm (ECC). Er agiert als zustandsloses Übertragungsglied:

1. **Warten:** Baustein wartet auf Datenaktualisierungen an den Adaptern `IN1` und/oder `IN2`.
2. **Berechnung:** Bei Wertänderung wird das Produkt gebildet.
3. **Ausgabe:** Das Ergebnis wird direkt an den Ausgang `OUT` weitergeleitet.

Anwendungsszenarien

- **Messwertskalierung:** Multiplikation eines analogen Sensorwertes (z. B. einer Spannung) mit einem Skalierungsfaktor zur Umrechnung in eine physikalische Einheit.
- **Signalverstärkung:** Proportionale Verstärkung von Regelsignalen in der Prozessautomatisierung.
- **Mathematische Berechnungen:** Einsatz als modularer Baustein in komplexeren Berechnungsnetzwerken innerhalb von 4diac-Applikationen.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-Multiplikationsbausteinen (wie dem klassischen `MUL`-Baustein nach IEC 61131-3), die mit elementaren Datentypen (z.B. `INT`, `REAL`) arbeiten, nutzt `AR_MUL_2_UNGATED` strukturierte **Adapter**. Dies hat folgende Vorteile:

- **Geringerer Verdrahtungsaufwand:** Signale und dazugehörige Steuerungsereignisse werden in einer einzigen Adapterverbindung gebündelt.
- **Bessere Lesbarkeit:** Komplexe Steuerungsdiagramme bleiben übersichtlich, da weniger Einzelverbindungen auf der Arbeitsfläche gezogen werden müssen.

- `AR_MUL_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

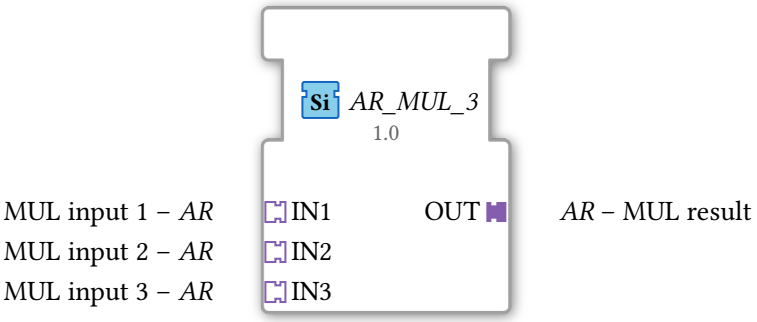
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AR_MUL_2_UNGATED`-Funktionsbaustein ist ein effizientes Werkzeug für arithmetische Operationen in modernen, adapterbasierten Steuerungsprogrammen. Durch die Kapselung in unidirektionale Schnittstellen bietet er ein hohes Maß an Wiederverwendbarkeit und sorgt für ein sauberes und strukturiertes Design innerhalb der 4diac-Umgebung.

AR_MUL_3



(Kein Bild vorhanden)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AR_MUL_3` ist ein generischer arithmetischer Baustein, der zur Multiplikation von drei Eingangswerten dient. Er basiert auf der Verwendung von unidirektionalen Adaptern des Typs `AR` (Arithmetic), was eine strukturierte und übersichtliche Signalübertragung innerhalb der 4diac-IDE ermöglicht. Da es sich um einen generischen Baustein handelt, ist er flexibel für verschiedene numerische Datentypen einsetzbar.

Schnittstellenstruktur

Die Schnittstellen dieses Funktionsbausteins sind vollständig über Adapter realisiert. Es gibt keine direkt herausgeführten Standard-Ereignis- oder Datenkanäle.

Ereignis-Eingänge

Es sind keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung wird über die Eingangs-Adapter abgewickelt.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge vorhanden. Die Ereignissteuerung wird über den Ausgangs-Adapter abgewickelt.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübertragung erfolgt gekoppelt über die Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Die Datenübertragung erfolgt gekoppelt über die Adapter.

Adapter

- **Sockets (Eingangs-Adapter):**
  - `IN1` (Typ: `adapter::types::unidirectional::AR`): Erster Multiplikand (Eingang 1).
  - `IN2` (Typ: `adapter::types::unidirectional::AR`): Zweiter Multiplikand (Eingang 2).
  - `IN3` (Typ: `adapter::types::unidirectional::AR`): Dritter Multiplikand (Eingang 3).
- **Plugs (Ausgangs-Adapter):**

– `OUT` (Typ: `adapter::types::unidirectional::AR`): Ergebnis der Multiplikation ($OUT = IN1 \cdot IN2 \cdot IN3$).

Funktionsweise

Der Baustein führt eine mathematische Multiplikation der an den Adaptern `IN1`, `IN2` und `IN3` anliegenden Werte aus:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \tag{2.68}$$

Sobald an den Eingangs-Adaptern ein Berechnungsereignis (z. B. eine Werteaktualisierung) signalisiert wird, liest der Baustein die aktuellen Werte aus, berechnet das Produkt und gibt das Ergebnis sowie ein entsprechendes Aktualisierungsereignis über den Ausgangs-Adapter `OUT` weiter.

Technische Besonderheiten

- **Generisches Verhalten (`GEN_AR_MUL`):** Der Baustein ist als generischer Typ deklariert. Dadurch kann er zur Entwicklungs- oder Laufzeit auf verschiedene numerische Datentypen (z. B. `INT`, `REAL`, `LREAL`) angewendet werden, sofern die verwendeten Adapter diesen Datentyp unterstützen.
- **Adapter-Struktur:** Die Verwendung von `unidirectional::AR`-Adaptern reduziert die Anzahl der sichtbaren Verbindungslinien im Funktionsplan (FBD) drastisch, da Daten und Steuerungsereignisse in einer einzigen Verbindung gebündelt sind.

Zustandsübersicht

Der Baustein verhält sich rein funktional und besitzt im Wesentlichen folgende logische Zustände:

- **Warten (Idle):** Der Baustein wartet auf ein Trigger-Ereignis über die Eingangs-Adapter.
- **Berechnen (Evaluation):** Nach dem Eintreffen eines Ereignisses werden die Eingangsdaten gelesen und multipliziert.
- **Aktualisieren (Update):** Das berechnete Produkt wird an den Ausgang `OUT` angelegt und das Ausgangsereignis getriggert.

Anwendungsszenarien

- **Volumenberechnungen:** Multiplikation von drei Dimensionen (Länge × Breite × Höhe) zur Ermittlung eines Volumens.
- **Skalierung und Gewichtung:** Anwendung von zwei aufeinanderfolgenden Skalierungsfaktoren auf einen Rohwert (z. B. Sensorwert × Kalibrierfaktor × Einheitenumrechnung).
- **Physikalische Formeln:** Berechnung von Größen, die direkt von drei Variablen abhängen (z. B. Leistung $P = U \cdot I \cdot \cos(\varphi)$ bei vereinfachter Betrachtung).

Vergleich mit ähnlichen Bausteinen

- **Standard-`MUL` (IEC 61131-3):** Klassische Multiplikationsbausteine arbeiten mit diskreten Daten- und Ereignis-Pins. `AR_MUL_3` nutzt stattdessen Adapter, was das Design übersichtlicher macht.
- **`AR_MUL_2`:** Multipliziert lediglich zwei Werte. `AR_MUL_3` spart einen zusätzlichen Kaskadierungs-Baustein ein, wenn drei Variablen multipliziert werden müssen, was die Performance und Übersichtlichkeit optimiert.
- `AR_MUL_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

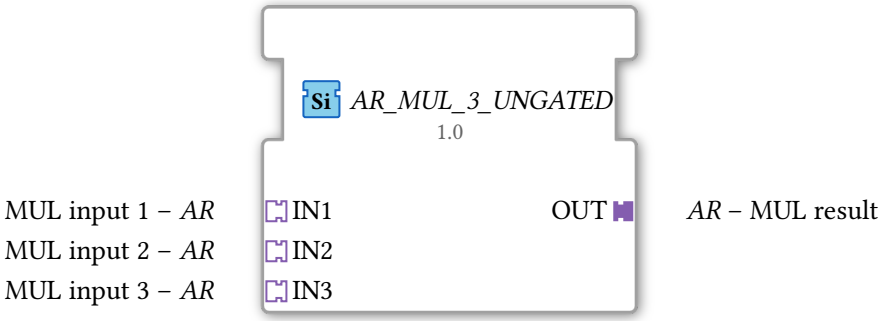
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `AR_MUL_3` ist ein praktischer und wiederverwendbarer Funktionsbaustein für die moderne IEC 61499 Entwicklung in 4diac. Durch die Kapselung der mathematischen Logik in einer adapterbasierten Struktur trägt er maßgeblich zur Übersichtlichkeit komplexer Steuerungsapplikationen bei.

AR_MUL_3_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von AR_MUL_3. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild vorhanden)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein AR_MUL_3_UNGATED ist ein generischer arithmetischer Baustein, der zur Multiplikation von drei Eingangswerten dient. Er basiert auf der Verwendung von unidirektionalen Adaptern des Typs AR (Arithmetic), was eine strukturierte und übersichtliche Signalübertragung innerhalb der 4diac-IDE ermöglicht. Da es sich um einen generischen Baustein handelt, ist er flexibel für verschiedene numerische Datentypen einsetzbar.

Schnittstellenstruktur

Die Schnittstellen dieses Funktionsbausteins sind vollständig über Adapter realisiert. Es gibt keine direkt herausgeführten Standard-Ereignis- oder Datenkanäle.

Ereignis-Eingänge

Es sind keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung wird über die Eingangs-Adapter abgewickelt.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge vorhanden. Die Ereignissteuerung wird über den Ausgangs-Adapter abgewickelt.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübertragung erfolgt gekoppelt über die Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Die Datenübertragung erfolgt gekoppelt über die Adapter.

Adapter

- **Sockets (Eingangs-Adapter):**
  - IN1 (Typ: adapter::types::unidirectional::AR): Erster Multiplikand (Eingang 1).
  - IN2 (Typ: adapter::types::unidirectional::AR): Zweiter Multiplikand (Eingang 2).
  - IN3 (Typ: adapter::types::unidirectional::AR): Dritter Multiplikand (Eingang 3).
- **Plugs (Ausgangs-Adapter):**
  - OUT (Typ: adapter::types::unidirectional::AR): Ergebnis der Multiplikation ($OUT = IN1 \cdot IN2 \cdot IN3$).

Funktionsweise

Der Baustein führt eine mathematische Multiplikation der an den Adaptern `IN1`, `IN2` und `IN3` anliegenden Werte aus:

$$\text{OUT} = \text{IN1} \times \text{IN2} \times \text{IN3} \tag{2.69}$$

Sobald an den Eingangs-Adaptern ein Berechnungsereignis (z. B. eine Werteaktualisierung) signalisiert wird, liest der Baustein die aktuellen Werte aus, berechnet das Produkt und gibt das Ergebnis sowie ein entsprechendes Aktualisierungsereignis über den Ausgangs-Adapter `OUT` weiter.

Technische Besonderheiten

- **Generisches Verhalten (`GEN_AR_MUL`):** Der Baustein ist als generischer Typ deklariert. Dadurch kann er zur Entwicklungs- oder Laufzeit auf verschiedene numerische Datentypen (z. B. `INT`, `REAL`, `LREAL`) angewendet werden, sofern die verwendeten Adapter diesen Datentyp unterstützen.
- **Adapter-Struktur:** Die Verwendung von `unidirectional::AR`-Adaptern reduziert die Anzahl der sichtbaren Verbindungslinien im Funktionsplan (FBD) drastisch, da Daten und Steuerungsereignisse in einer einzigen Verbindung gebündelt sind.

Zustandsübersicht

Der Baustein verhält sich rein funktional und besitzt im Wesentlichen folgende logische Zustände:

- **Warten (Idle):** Der Baustein wartet auf ein Trigger-Ereignis über die Eingangs-Adapter.
- **Berechnen (Evaluation):** Nach dem Eintreffen eines Ereignisses werden die Eingangsdaten gelesen und multipliziert.
- **Aktualisieren (Update):** Das berechnete Produkt wird an den Ausgang `OUT` angelegt und das Ausgangsereignis getriggert.

Anwendungsszenarien

- **Volumenberechnungen:** Multiplikation von drei Dimensionen (Länge × Breite × Höhe) zur Ermittlung eines Volumens.
- **Skalierung und Gewichtung:** Anwendung von zwei aufeinanderfolgenden Skalierungsfaktoren auf einen Rohwert (z. B. Sensorwert × Kalibrierfaktor × Einheitenumrechnung).
- **Physikalische Formeln:** Berechnung von Größen, die direkt von drei Variablen abhängen (z. B. Leistung $P = U \cdot I \cdot \cos(\varphi)$ bei vereinfachter Betrachtung).

Vergleich mit ähnlichen Bausteinen

- **Standard-`MUL` (IEC 61131-3):** Klassische Multiplikationsbausteine arbeiten mit diskreten Daten- und Ereignis-Pins. `AR_MUL_3_UNGATED` nutzt stattdessen Adapter, was das Design übersichtlicher macht.
- **`AR_MUL_2`:** Multipliziert lediglich zwei Werte. `AR_MUL_3_UNGATED` spart einen zusätzlichen Kaskadierungs-Baustein ein, wenn drei Variablen multipliziert werden müssen, was die Performance und Übersichtlichkeit optimiert.
- `AR_MUL_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

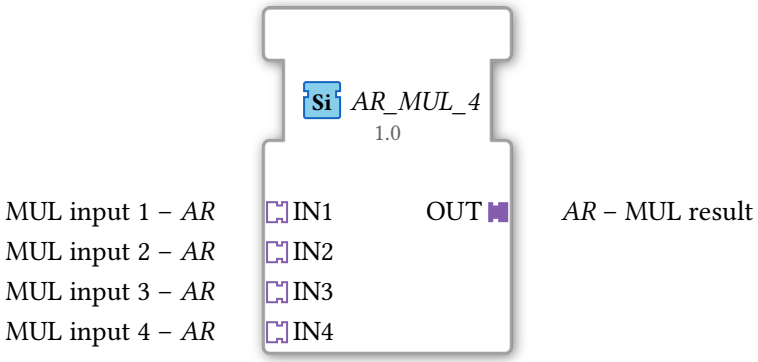
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AR_MUL_3_UNGATED` ist ein praktischer und wiederverwendbarer Funktionsbaustein für die moderne IEC 61499 Entwicklung in 4diac. Durch die Kapselung der mathematischen Logik in einer adapterbasierten Struktur trägt er maßgeblich zur Übersichtlichkeit komplexer Steuerungsapplikationen bei.

AR_MUL_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AR_MUL_4` ist ein generischer arithmetischer Baustein, der zur Multiplikation von vier Eingangswerten dient. Er basiert auf einer Adapter-Schnittstelle, die eine saubere Kapselung und Strukturierung von Daten und dazugehörigen Ereignissen ermöglicht. Durch seine generische Natur (`GEN_AR_MUL`) kann sich der Baustein flexibel an verschiedene numerische Datentypen anpassen.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine direkten, dedizierten Ereignis-Eingänge. Die Synchronisation und die Auslösung der Berechnung erfolgen implizit über die angeschlossenen Eingangs-Adapter.

Ereignis-Ausgänge

Der Baustein besitzt keine direkten, dedizierten Ereignis-Ausgänge. Die Weiterleitung von Berechnungs- und Aktualisierungsereignissen erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Werte für die Multiplikation werden über die Adapter-Schnittstellen eingelesen.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Das Ergebnis der Multiplikation wird über den Ausgangs-Adapter bereitgestellt.

Adapter

Die Kommunikation mit anderen Bausteinen erfolgt ausschließlich über Adapter vom Typ `adapter::types::unidirectional::AR`:

- **Sockets (Eingangs-Adapter):**
  - `IN1`: Erster Multiplikand (Eingang 1).
  - `IN2`: Zweiter Multiplikand (Eingang 2).
  - `IN3`: Dritter Multiplikand (Eingang 3).
  - `IN4`: Vierter Multiplikand (Eingang 4).
- **Plugs (Ausgangs-Adapter):**
  - `OUT`: Das berechnete Produkt der vier Eingänge.

Funktionsweise

Sobald über die Eingangs-Adapter (`IN1` bis `IN4`) neue Werte und die entsprechenden Trigger-Ereignisse eintreffen, multipliziert der Baustein die Werte miteinander. Das mathematische Verhalten entspricht der Formel:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \times \mathrm{IN4} \tag{2.70}$$

Das Ergebnis wird zusammen mit einem entsprechenden Aktualisierungsereignis über den Ausgangs-Adapter `OUT` ausgegeben.

Technische Besonderheiten

- **Generisches Verhalten:** Durch das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `GEN_AR_MUL` ist der Baustein datentypunabhängig konzipiert. Er kann je nach Anwendung mit verschiedenen numerischen Datentypen (z. B. `INT`, `REAL`, `LREAL`) arbeiten, die vom Adaptertyp `AR` unterstützt werden.
- **Adapter-Kopplung:** Die Verwendung von unidirektionalen Adaptern (`AR`) reduziert den Verdrahtungsaufwand in der 4diac-IDE erheblich, da Daten und Steuerungsereignisse in einer einzigen Verbindung gebündelt werden.

Zustandsübersicht

Da es sich um einen rein mathematischen Berechnungsbaustein handelt, ist der Baustein zustandslos. Jede Aktivierung führt direkt zur Berechnung des aktuellen Produkts basierend auf den anliegenden Werten der Eingangs-Adapter.

Anwendungsszenarien

- **Signal-Skalierung:** Kalibrierung oder Anpassung von Sensorwerten, bei denen mehrere Faktoren (z. B. Grundwert, Verstärkung, Korrekturfaktor, Einheiten-Umrechnung) miteinander multipliziert werden müssen.
- **Physikalische Berechnungen:** Berechnung von komplexeren Größen wie Volumenströmen, elektrischer Leistung oder Energieflüssen, bei denen mehrere Messgrößen und konstante Faktoren einfließen.
- **Kaskaden-Vermeidung:** Zusammenfassung mehrerer Multiplikationsschritte in einem einzigen Baustein zur Erhöhung der Übersichtlichkeit im Applikationsdiagramm.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu herkömmlichen IEC 61131-3 `MUL`-Bausteinen, die oft nur zwei Eingänge besitzen und direkte Daten- und Ereignispins nutzen, bietet der `AR_MUL_4` durch die Zusammenfassung von vier Eingängen und die Verwendung von Adaptern eine wesentlich aufgeräumtere visuelle Darstellung im Steuerungsprogramm. Er verhindert die Notwendigkeit, mehrere Multiplizierer hintereinander zu schalten (Kaskadierung).

- `AR_MUL_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

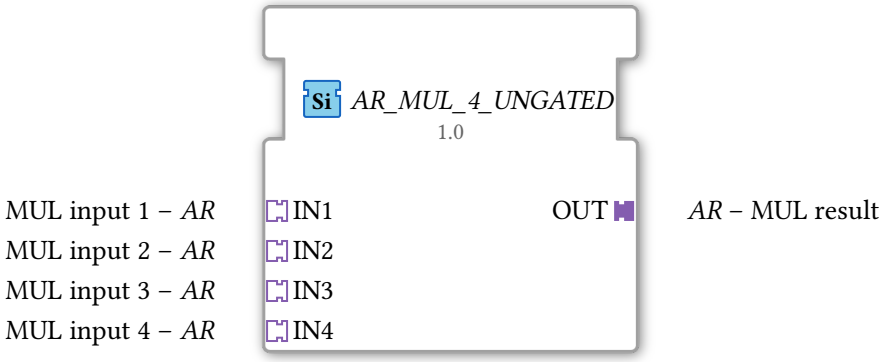
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `AR_MUL_4` ist ein praktischer Hilfsbaustein für arithmetische Operationen in der IEC 61499-Umgebung. Durch die konsequente Nutzung von Adaptern trägt er maßgeblich zur Übersichtlichkeit und Modularität von Steuerungsprogrammen bei, insbesondere bei der Verarbeitung komplexerer mathematischer Formeln.

AR_MUL_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AR_MUL_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AR_MUL_4_UNGATED` ist ein generischer arithmetischer Baustein, der zur Multiplikation von vier Eingangswerten dient. Er basiert auf einer Adapter-Schnittstelle, die eine saubere Kapselung und Strukturierung von Daten und dazugehörigen Ereignissen ermöglicht. Durch seine generische Natur (`GEN_AR_MUL`) kann sich der Baustein flexibel an verschiedene numerische Datentypen anpassen.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine direkten, dedizierten Ereignis-Eingänge. Die Synchronisation und die Auslösung der Berechnung erfolgen implizit über die angeschlossenen Eingangs-Adapter.

Ereignis-Ausgänge

Der Baustein besitzt keine direkten, dedizierten Ereignis-Ausgänge. Die Weiterleitung von Berechnungs- und Aktualisierungsereignissen erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Werte für die Multiplikation werden über die Adapter-Schnittstellen eingelesen.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Das Ergebnis der Multiplikation wird über den Ausgangs-Adapter bereitgestellt.

Adapter

Die Kommunikation mit anderen Bausteinen erfolgt ausschließlich über Adapter vom Typ `adapter::types::unidirectional::AR`:

- **Sockets (Eingangs-Adapter):**
  - `IN1`: Erster Multiplikand (Eingang 1).
  - `IN2`: Zweiter Multiplikand (Eingang 2).
  - `IN3`: Dritter Multiplikand (Eingang 3).
  - `IN4`: Vierter Multiplikand (Eingang 4).
- **Plugs (Ausgangs-Adapter):**
  - `OUT`: Das berechnete Produkt der vier Eingänge.

Funktionsweise

Sobald über die Eingangs-Adapter (`IN1` bis `IN4`) neue Werte und die entsprechenden Trigger-Ereignisse eintreffen, multipliziert der Baustein die Werte miteinander. Das mathematische Verhalten entspricht der Formel:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \times \mathrm{IN4} \tag{2.71}$$

Das Ergebnis wird zusammen mit einem entsprechenden Aktualisierungsereignis über den Ausgangs-Adapter `OUT` ausgegeben.

Technische Besonderheiten

- **Generisches Verhalten:** Durch das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `GEN_AR_MUL` ist der Baustein datentypunabhängig konzipiert. Er kann je nach Anwendung mit verschiedenen numerischen Datentypen (z. B. `INT`, `REAL`, `LREAL`) arbeiten, die vom Adaptertyp `AR` unterstützt werden.
- **Adapter-Kopplung:** Die Verwendung von unidirektionalen Adaptern (`AR`) reduziert den Verdrahtungsaufwand in der 4diac-IDE erheblich, da Daten und Steuerungsereignisse in einer einzigen Verbindung gebündelt werden.

Zustandsübersicht

Da es sich um einen rein mathematischen Berechnungsbaustein handelt, ist der Baustein zustandslos. Jede Aktivierung führt direkt zur Berechnung des aktuellen Produkts basierend auf den anliegenden Werten der Eingangs-Adapter.

Anwendungsszenarien

- **Signal-Skalierung:** Kalibrierung oder Anpassung von Sensorwerten, bei denen mehrere Faktoren (z. B. Grundwert, Verstärkung, Korrekturfaktor, Einheiten-Umrechnung) miteinander multipliziert werden müssen.
- **Physikalische Berechnungen:** Berechnung von komplexeren Größen wie Volumenströmen, elektrischer Leistung oder Energieflüssen, bei denen mehrere Messgrößen und konstante Faktoren einfließen.
- **Kaskaden-Vermeidung:** Zusammenfassung mehrerer Multiplikationsschritte in einem einzigen Baustein zur Erhöhung der Übersichtlichkeit im Applikationsdiagramm.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu herkömmlichen IEC 61131-3 `MUL`-Bausteinen, die oft nur zwei Eingänge besitzen und direkte Daten- und Ereignispins nutzen, bietet der `AR_MUL_4_UNGATED` durch die Zusammenfassung von vier Eingängen und die Verwendung von Adaptern eine wesentlich aufgeräumtere visuelle Darstellung im Steuerungsprogramm. Er verhindert die Notwendigkeit, mehrere Multiplizierer hintereinander zu schalten (Kaskadierung).

- `AR_MUL_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- `AR_MUL_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

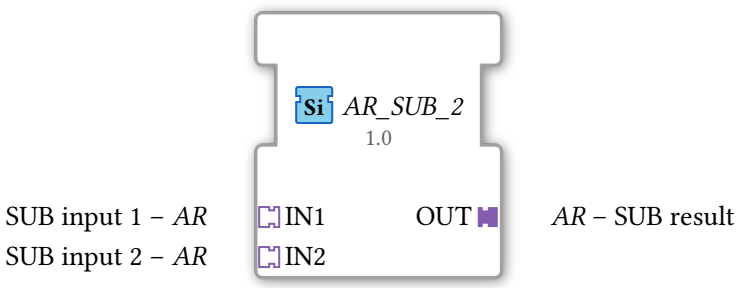
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AR_MUL_4_UNGATED` ist ein praktischer Hilfsbaustein für arithmetische Operationen in der IEC 61499-Umgebung. Durch die konsequente Nutzung von Adaptern trägt er maßgeblich zur Übersichtlichkeit und Modularität von Steuerungsprogrammen bei, insbesondere bei der Verarbeitung komplexerer mathematischer Formeln.

AR_SUB_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AR_SUB_2` dient zur Durchführung einer arithmetischen Subtraktion (Subtrahend von Minuend). Es handelt sich um einen generischen Funktionsbaustein (`GEN_AR_SUB`), der für die Berechnung keine klassischen diskreten Ein- und Ausgänge nutzt, sondern vollständig auf einer adapterbasierten Schnittstellenstruktur aufbaut. Die Werte und die dazugehörige Steuerungslogik werden über unidirektionale Adapter vom Typ `AR` übertragen.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten, diskreten Ereignis-Eingänge. Die Ereignissteuerung wird über die angeschlossenen Adapter realisiert.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten, diskreten Ereignis-Ausgänge. Die Ereignisweiterleitung erfolgt gekapselt über den Ausgangs-Adapter.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Eingangsdaten werden über die Adapter-Eingänge empfangen.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Das Berechnungsergebnis wird über den Adapter-Ausgang bereitgestellt.

Adapter

- **Sockets (Eingangs-Adapter):**
  - **IN1** (Typ: `adapter::types::unidirectional::AR`): Erster Eingang der Subtraktion (Minuend).
  - **IN2** (Typ: `adapter::types::unidirectional::AR`): Zweiter Eingang der Subtraktion (Subtrahend).
- **Plugs (Ausgangs-Adapter):**
  - **OUT** (Typ: `adapter::types::unidirectional::AR`): Ausgang für das Ergebnis der Subtraktion ($OUT = IN1 - IN2$).

Funktionsweise

Der Baustein subtrahiert den Wert, der am Adapter `IN2` anliegt, von dem Wert des Adapters `IN1`. Das Ergebnis dieser arithmetischen Operation wird an den Ausgangs-Plug `OUT` übergeben.

Da es sich um unidirektionale Adapter handelt, triggern eingehende Ereignisse auf den Sockets `IN1` oder `IN2` die interne Berechnung. Nach erfolgter Subtraktion wird das entsprechende Aktualisierungsereignis über den Plug `OUT` an die nachfolgenden Bausteine signalisiert.

Technische Besonderheiten

- **Generischer Typ (`GEN_AR_SUB`)**: Der Baustein ist generisch implementiert. Dies ermöglicht eine flexible Handhabung verschiedener numerischer Datentypen, sofern diese innerhalb der Definition der `AR`-Adapterstruktur zulässig sind.
- **Kapselung**: Durch die Verwendung von Adaptern werden Daten- und Event-Leitungen gebündelt. Dies reduziert die Anzahl der sichtbaren Verbindungslinien im Funktionsplan der 4diac-IDE erheblich.

Zustandsübersicht

Der Baustein besitzt keine komplexe interne Zustandsmaschine (kein Execution Control Chart - ECC), da es sich um einen rein datenflussorientierten Berechnungsbaustein handelt. Die Verarbeitung verhält sich wie folgt:

1. **Warten auf Event**: Der Baustein wartet auf ein Aktualisierungsereignis an `IN1` oder `IN2`.
2. **Berechnung**: Bei Erhalt eines Events werden die aktuellen Werte aus `IN1` und `IN2` gelesen und voneinander abgezogen.
3. **Ausgabe**: Das Ergebnis wird in `OUT` geschrieben und ein Ausgangs-Event am Plug `OUT` ausgelöst.

Anwendungsszenarien

- **Soll-Istwert-Vergleich**: Berechnung der Regelabweichung ($e = w - x$) in Regelungskreisen, bei denen die Signale bereits als Adapterstrukturen vorliegen.
- **Offset-Kompensation**: Subtraktion eines Nullpunktfehlers oder Offsets von einem analogen Sensorwert.
- **Strukturierte Signalverarbeitung**: Mathematische Berechnungen in komplexen, verteilten Steuerungssystemen zur Wahrung einer übersichtlichen Software-Architektur.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zum Standard-Subtraktionsbaustein `SUB` (aus den klassischen IEC 61131-3 Elementen), welcher mit diskreten Eingängen wie `IN1` und `IN2` sowie expliziten Events (`REQ` / `CNF`) arbeitet, bietet `AR_SUB_2` durch die Nutzung von Adaptern eine wesentlich höhere Abstraktion. Dies spart Entwicklungszeit bei der Kopplung komplexer Signalglieder, erfordert jedoch eine konsistente Verwendung von Adaptern im gesamten Projekt.

- `AR_SUB_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

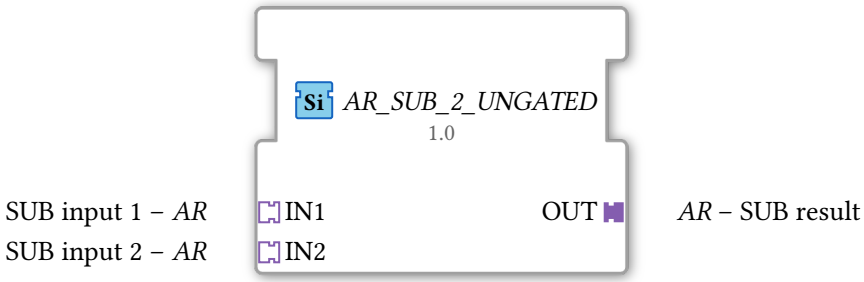
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `AR_SUB_2` ist ein spezialisierter und moderner Rechenbaustein für die 4diac-IDE. Er eignet sich hervorragend für serviceorientierte Architekturen innerhalb der IEC 61499, bei denen Übersichtlichkeit und standardisierte Adapter-Schnittstellen im Vordergrund stehen.

AR_SUB_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AR_SUB_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AR_SUB_2_UNGATED` dient zur Durchführung einer arithmetischen Subtraktion (Subtrahend von Minuend). Es handelt sich um einen generischen Funktionsbaustein (`GEN_AR_SUB`), der für die Berechnung keine klassischen diskreten Ein- und Ausgänge nutzt, sondern vollständig auf einer adapterbasierten Schnittstellenstruktur aufbaut. Die Werte und die dazugehörige Steuerungslogik werden über unidirektionale Adapter vom Typ `AR` übertragen.

Schnittstellenstruktur

Ereignis-Eingänge
Dieser Funktionsbaustein besitzt keine direkten, diskreten Ereignis-Eingänge. Die Ereignissteuerung wird über die angeschlossenen Adapter realisiert.

Ereignis-Ausgänge
Dieser Funktionsbaustein besitzt keine direkten, diskreten Ereignis-Ausgänge. Die Ereignisweiterleitung erfolgt gekapselt über den Ausgangs-Adapter.

Daten-Eingänge
Es sind keine direkten Daten-Eingänge vorhanden. Die Eingangsdaten werden über die Adapter-Eingänge empfangen.

Daten-Ausgänge
Es sind keine direkten Daten-Ausgänge vorhanden. Das Berechnungsergebnis wird über den Adapter-Ausgang bereitgestellt.

Adapter
- **Sockets (Eingangs-Adapter):**
  - **IN1** (Typ: `adapter::types::unidirectional::AR`): Erster Eingang der Subtraktion (Minuend).
  - **IN2** (Typ: `adapter::types::unidirectional::AR`): Zweiter Eingang der Subtraktion (Subtrahend).
- **Plugs (Ausgangs-Adapter):**
  - **OUT** (Typ: `adapter::types::unidirectional::AR`): Ausgang für das Ergebnis der Subtraktion ($OUT = IN1 - IN2$).

Funktionsweise
Der Baustein subtrahiert den Wert, der am Adapter `IN2` anliegt, von dem Wert des Adapters `IN1`. Das Ergebnis dieser arithmetischen Operation wird an den Ausgangs-Plug `OUT` übergeben.

Da es sich um unidirektionale Adapter handelt, triggern eingehende Ereignisse auf den Sockets `IN1` oder `IN2` die interne Berechnung. Nach erfolgter Subtraktion wird das entsprechende Aktualisierungsereignis über den Plug `OUT` an die nachfolgenden Bausteine signalisiert.

Technische Besonderheiten
- **Generischer Typ (`GEN_AR_SUB`)**: Der Baustein ist generisch implementiert. Dies ermöglicht eine flexible Handhabung verschiedener numerischer Datentypen, sofern diese innerhalb der Definition der `AR`-Adapterstruktur zulässig sind.
- **Kapselung**: Durch die Verwendung von Adaptern werden Daten- und Event-Leitungen gebündelt. Dies reduziert die Anzahl der sichtbaren Verbindungslinien im Funktionsplan der 4diac-IDE erheblich.

Zustandsübersicht
Der Baustein besitzt keine komplexe interne Zustandsmaschine (kein Execution Control Chart - ECC), da es sich um einen rein datenflussorientierten Berechnungsbaustein handelt. Die Verarbeitung verhält sich wie folgt:

1. **Warten auf Event**: Der Baustein wartet auf ein Aktualisierungsereignis an `IN1` oder `IN2`.
2. **Berechnung**: Bei Erhalt eines Events werden die aktuellen Werte aus `IN1` und `IN2` gelesen und voneinander abgezogen.
3. **Ausgabe**: Das Ergebnis wird in `OUT` geschrieben und ein Ausgangs-Event am Plug `OUT` ausgelöst.

Anwendungsszenarien
- **Soll-Istwert-Vergleich**: Berechnung der Regelabweichung ($e = w - x$) in Regelungskreisen, bei denen die Signale bereits als Adapterstrukturen vorliegen.
- **Offset-Kompensation**: Subtraktion eines Nullpunktfehlers oder Offsets von einem analogen Sensorwert.
- **Strukturierte Signalverarbeitung**: Mathematische Berechnungen in komplexen, verteilten Steuerungssystemen zur Wahrung einer übersichtlichen Software-Architektur.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zum Standard-Subtraktionsbaustein `SUB` (aus den klassischen IEC 61131-3 Elementen), welcher mit diskreten Eingängen wie `IN1` und `IN2` sowie expliziten Events (`REQ` / `CNF`) arbeitet, bietet `AR_SUB_2_UNGATED` durch die Nutzung von Adaptern eine wesentlich höhere Abstraktion. Dies spart Entwicklungszeit bei der Kopplung komplexer Signalglieder, erfordert jedoch eine konsistente Verwendung von Adaptern im gesamten Projekt.

- **`AR_SUB_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

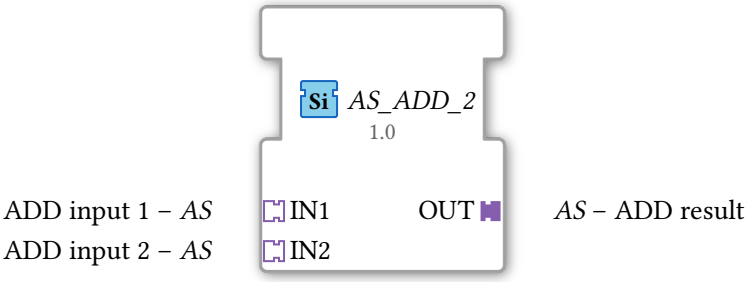
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AR_SUB_2_UNGATED` ist ein spezialisierter und moderner Rechenbaustein für die 4diac-IDE. Er eignet sich hervorragend für serviceorientierte Architekturen innerhalb der IEC 61499, bei denen Übersichtlichkeit und standardisierte Adapter-Schnittstellen im Vordergrund stehen.

AS_ADD_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AS_ADD_2` ist ein generischer arithmetischer Baustein, der für die Addition zweier Werte konzipiert ist. Im Gegensatz zu klassischen mathematischen Funktionsbausteinen nutzt dieser Baustein ein adapterbasiertes Schnittstellenkonzept. Durch die Verwendung von unidirektionalen Adaptern werden Daten und die dazugehörigen Steuerungsereignisse gekapselt übertragen, was zu einem aufgeräumten und modularen Anwendungsdesign in der IEC 61499 Entwicklungsumgebung (wie 4diac IDE) beiträgt.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung ist in den Eingangs-Adaptern integriert.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Ereignissteuerung ist im Ausgangs-Adapter integriert.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden. Die Datenübertragung erfolgt gekapselt über die Eingangs-Adapter.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden. Die Datenübertragung erfolgt gekapselt über den Ausgangs-Adapter.

Adapter

- **IN1 (Socket / Buchse):** Typ `adapter::types::unidirectional::AS`
  - Schnittstelle für den ersten Summanden der Addition.

- **IN2 (Socket / Buchse):** Typ `adapter::types::unidirectional::AS`
  - Schnittstelle für den zweiten Summanden der Addition.
- **OUT (Plug / Stecker):** Typ `adapter::types::unidirectional::AS`
  - Schnittstelle zur Ausgabe des berechneten Additionsergebnisses.

Funktionsweise

Der Baustein `AS_ADD_2` realisiert die mathematische Operation:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} \tag{2.72}$$

Sobald an den Eingangs-Adaptern (`IN1` oder `IN2`) neue Daten signalisiert werden, triggert der Baustein intern die Addition der übertragenen Werte. Das Ergebnis wird anschließend unmittelbar berechnet und über den Ausgangs-Adapter `OUT` zusammen mit einem entsprechenden Aktualisierungsereignis an die nachfolgenden Logikglieder weitergegeben.

Aufgrund seiner generischen Natur (`GEN_AS_ADD`) kann der Baustein flexibel mit verschiedenen numerischen Datentypen arbeiten, sofern der zugrunde liegende Adaptertyp `AS` diese unterstützt.

Technische Besonderheiten

- **Generische Implementierung:** Der Baustein basiert auf der generischen Klasse `GEN_AS_ADD`, wodurch er für unterschiedliche Datentypen wiederverwendbar ist.
- **Adapter-Kopplung:** Die Kapselung von Signalen in Adaptern reduziert die Anzahl der sichtbaren Verbindungslinien im Funktionsbausteindiagramm (FBD) drastisch und verbessert die Lesbarkeit komplexer Anwendungen.
- **Unidirektionaler Datenfluss:** Die Verwendung des Typs `unidirectional::AS` stellt sicher, dass der Informationsfluss klar definiert von den Signalquellen (Sockets) zur Signalsenke (Plug) verläuft.

Zustandsübersicht

Als rein mathematischer Kombinationsbaustein besitzt `AS_ADD_2` kein komplexes internes Zustandsdiagramm (ECC). Sein Verhalten lässt sich in drei zyklische Schritte unterteilen:

1. **Warten (Idle):** Der Baustein wartet auf ein Ereignis an einem der Eingangs-Adapter (`IN1` oder `IN2`).
2. **Berechnen:** Bei Event-Eingang werden die aktuellen Werte aus beiden Adaptern gelesen und addiert.
3. **Senden:** Das Additionsergebnis wird in den Ausgangs-Adapter `OUT` geschrieben und ein Ausgangsereignis getriggert.

Anwendungsszenarien

- **Messwert-Offset-Berechnung:** Aufaddieren eines Kalibrierungs- oder Korrekturwerts (Offset) auf einen analogen Sensorwert innerhalb einer adapterbasierten Signalverarbeitungskette.
- **Signalzusammenführung:** Summierung von zwei unabhängig voneinander erfassten physikalischen Größen (z. B. zwei Teilströme zur Ermittlung des Gesamtstroms).
- **Kaskadierte Berechnungen:** Einfache Erweiterung für mehr als zwei Summanden durch kaskadiertes Hintereinanderschalten mehrerer `AS_ADD_2`-Bausteine.

Vergleich mit ähnlichen Bausteinen

- **Standard ADD (z. B. F_ADD):** Der klassische IEC 61131-3 bzw. IEC 61499 ADD-Baustein arbeitet mit diskreten Variablen (z. B. `ANY_NUM`) und separaten Event-Ports (`REQ` / `CNF`). `AS_ADD_2` hingegen bündelt diese Signale in Adaptern, was die Verdrahtung vereinfacht, jedoch die Verwendung des spezifischen Adaptertyps `AS` voraussetzt.
- **Multi-Addierer (z. B. ADD_3):** Ermöglicht die Addition von drei oder mehr Werten in einem einzigen Baustein, ist jedoch oft unhandlicher, wenn Datenstrukturen konsistent über Adapter transportiert werden sollen.
- `AS_ADD_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

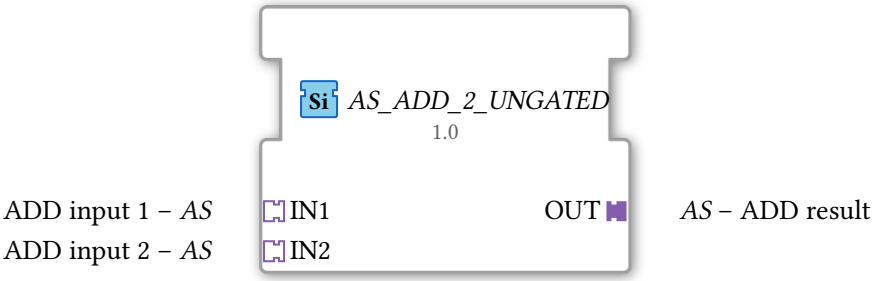
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `AS_ADD_2` ist ein spezialisierter Hilfsbaustein für moderne, modular aufgebaute IEC 61499 Steuerungsprogramme. Durch die konsequente Nutzung von Adaptern fügt er sich nahtlos in serviceorientierte Architekturen ein und minimiert den Design- und Verdrahtungsaufwand in der 4diac IDE.

AS_ADD_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AS_ADD_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AS_ADD_2_UNGATED` ist ein generischer arithmetischer Baustein, der für die Addition zweier Werte konzipiert ist. Im Gegensatz zu klassischen mathematischen Funktionsbausteinen nutzt dieser Baustein ein adapterbasiertes Schnittstellenkonzept. Durch die Verwendung von unidirektionalen Adaptern werden Daten und die dazugehörigen Steuerungsereignisse gekapselt übertragen, was zu einem aufgeräumten und modularen Anwendungsdesign in der IEC 61499 Entwicklungsumgebung (wie 4diac IDE) beiträgt.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung ist in den Eingangs-Adaptern integriert.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Ereignissteuerung ist im Ausgangs-Adapter integriert.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden. Die Datenübertragung erfolgt gekapselt über die Eingangs-Adapter.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden. Die Datenübertragung erfolgt gekapselt über den Ausgangs-Adapter.

Adapter

- **IN1 (Socket / Buchse):** Typ `adapter::types::unidirectional::AS`
  - Schnittstelle für den ersten Summanden der Addition.
- **IN2 (Socket / Buchse):** Typ `adapter::types::unidirectional::AS`
  - Schnittstelle für den zweiten Summanden der Addition.
- **OUT (Plug / Stecker):** Typ `adapter::types::unidirectional::AS`
  - Schnittstelle zur Ausgabe des berechneten Additionsergebnisses.

Funktionsweise

Der Baustein AS_ADD_2_UNGATED realisiert die mathematische Operation:

$$\text{OUT} = \text{IN1} + \text{IN2} \tag{2.73}$$

Sobald an den Eingangs-Adaptern (IN1 oder IN2) neue Daten signalisiert werden, triggert der Baustein intern die Addition der übertragenen Werte. Das Ergebnis wird anschließend unmittelbar berechnet und über den Ausgangs-Adapter OUT zusammen mit einem entsprechenden Aktualisierungsereignis an die nachfolgenden Logikglieder weitergegeben.

Aufgrund seiner generischen Natur (GEN_AS_ADD) kann der Baustein flexibel mit verschiedenen numerischen Datentypen arbeiten, sofern der zugrunde liegende Adaptertyp AS diese unterstützt.

Technische Besonderheiten

- **Generische Implementierung:** Der Baustein basiert auf der generischen Klasse GEN_AS_ADD, wodurch er für unterschiedliche Datentypen wiederverwendbar ist.
- **Adapter-Kopplung:** Die Kapselung von Signalen in Adaptern reduziert die Anzahl der sichtbaren Verbindungslinien im Funktionsbausteindiagramm (FBD) drastisch und verbessert die Lesbarkeit komplexer Anwendungen.
- **Unidirektionaler Datenfluss:** Die Verwendung des Typs unidirectional::AS stellt sicher, dass der Informationsfluss klar definiert von den Signalquellen (Sockets) zur Signalsenke (Plug) verläuft.

Zustandsübersicht

Als rein mathematischer Kombinationsbaustein besitzt AS_ADD_2_UNGATED kein komplexes internes Zustandsdiagramm (ECC). Sein Verhalten lässt sich in drei zyklische Schritte unterteilen:

1. **Warten (Idle):** Der Baustein wartet auf ein Ereignis an einem der Eingangs-Adapter (IN1 oder IN2).
2. **Berechnen:** Bei Event-Eingang werden die aktuellen Werte aus beiden Adaptern gelesen und addiert.
3. **Senden:** Das Additionsergebnis wird in den Ausgangs-Adapter OUT geschrieben und ein Ausgangsereignis getriggert.

Anwendungsszenarien

- **Messwert-Offset-Berechnung:** Aufaddieren eines Kalibrierungs- oder Korrekturwerts (Offset) auf einen analogen Sensorwert innerhalb einer adapterbasierten Signalverarbeitungskette.
- **Signalzusammenführung:** Summierung von zwei unabhängig voneinander erfassten physikalischen Größen (z. B. zwei Teilströme zur Ermittlung des Gesamtstroms).
- **Kaskadierte Berechnungen:** Einfache Erweiterung für mehr als zwei Summanden durch kaskadiertes Hintereinanderschalten mehrerer AS_ADD_2_UNGATED-Bausteine.

Vergleich mit ähnlichen Bausteinen

- **Standard ADD (z. B. F_ADD):** Der klassische IEC 61131-3 bzw. IEC 61499 ADD-Baustein arbeitet mit diskreten Variablen (z. B. ANY_NUM) und separaten Event-Ports (REQ / CNF). AS_ADD_2_UNGATED hingegen bündelt diese Signale in Adaptern, was die Verdrahtung vereinfacht, jedoch die Verwendung des spezifischen Adaptertyps AS voraussetzt.
- **Multi-Addierer (z. B. ADD_3):** Ermöglicht die Addition von drei oder mehr Werten in einem einzigen Baustein, ist jedoch oft unhandlicher, wenn Datenstrukturen konsistent über Adapter transportiert werden sollen.
- **AS_ADD_2_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- **AS_ADD_2**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

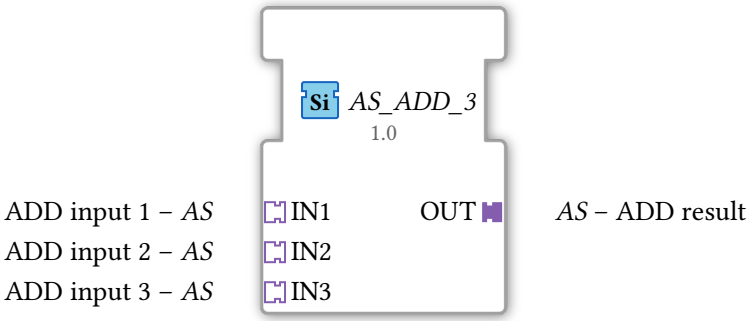
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der AS_ADD_2_UNGATED ist ein spezialisierter Hilfsbaustein für moderne, modular aufgebaute IEC 61499 Steuerungsprogramme. Durch die konsequente Nutzung von Adaptern fügt er sich nahtlos in serviceorientierte Architekturen ein und minimiert den Design- und Verdrahtungsaufwand in der 4diac IDE.

AS_ADD_3



Einleitung

Der Funktionsbaustein `AS_ADD_3` ist ein generischer Funktionsbaustein (Generic FB) für die IEC 61499, der für die Durchführung einer arithmetischen Addition von drei Eingangswerten entwickelt wurde. Die Besonderheit dieses Bausteins liegt darin, dass er keine direkten Daten- und Ereignis-Ein-/Ausgänge besitzt, sondern vollständig auf der Verwendung von unidirektionalen Adaptern basiert. Dies vereinfacht die Modellierung und Verdrahtung in komplexeren Systemen erheblich.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Eingänge. Die Ereignissteuerung wird über die angeschlossenen Adapter abgewickelt.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Ausgänge. Die Weiterleitung von Ereignissen erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden.

Adapter

Der Baustein kommuniziert ausschließlich über Adapter-Schnittstellen.

Sockets (Buchsen - Eingänge):

- **IN1** (Typ: `adapter::types::unidirectional::AS`): Erster Eingang für die Addition.
- **IN2** (Typ: `adapter::types::unidirectional::AS`): Zweiter Eingang für die Addition.
- **IN3** (Typ: `adapter::types::unidirectional::AS`): Dritter Eingang für die Addition.

Plugs (Stecker - Ausgänge):

- **OUT** (Typ: `adapter::types::unidirectional::AS`): Ausgang, der das Ergebnis der Addition (`IN1 + IN2 + IN3`) bereitstellt.

Funktionsweise

Der Funktionsbaustein `AS_ADD_3` führt eine klassische Addition aus:

$$\text{OUT} = \text{IN1} + \text{IN2} + \text{IN3} \tag{2.74}$$

Da der Baustein als generischer Funktionsbaustein (`GEN_AS_ADD`) definiert ist, kann er je nach konkreter Instanziierung und dem im Adapter verwendeten Datentyp verschiedene numerische Datentypen (z. B. `INT`, `REAL`, `DINT`) verarbeiten.

Sobald an den Eingangs-Adaptern (IN1, IN2, IN3) neue Werte und die dazugehörigen Ereignisse signalisiert werden, berechnet der Baustein die Summe und triggert das entsprechende Ausgangsereignis am OUT-Adapter, um den neuen Wert an nachfolgende Bausteine weiterzugeben.

Technische Besonderheiten

- **Generischer Typ (Generic FB):** Der Baustein nutzt das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `'GEN_AS_ADD'`. Dies ermöglicht eine flexible Anpassung an unterschiedliche Datentypen, ohne dass für jeden Typ ein eigener Baustein erstellt werden muss.
- **Kapselung durch Adapter:** Durch die Verwendung von unidirektionalen Adaptern (AS) werden Daten und die dazugehörigen Trigger-Ereignisse in einer einzigen Verbindung gekapselt. Dies sorgt für ein sauberes und übersichtliches Oberflächendesign in der 4diac IDE.

Zustandsübersicht

Der Baustein arbeitet ereignisgesteuert und besitzt kein komplexes internes Zustandsdiagramm (ECC). Seine Ausführung wird durch die an den Sockets eingehenden Adapter-Ereignisse getriggert:

1. **Warten:** Der Baustein wartet auf ein Aktualisierungsereignis an den Adaptern IN1, IN2 oder IN3.
2. **Berechnung:** Bei Empfang eines Ereignisses werden die aktuellen Werte der drei Eingänge addiert.
3. **Ausgabe:** Das Ergebnis wird an den Plug OUT angelegt und ein Aktualisierungsereignis über den Adapter gesendet.

Anwendungsszenarien

- **Messwert-Summierung:** Addition von drei analogen Sensorwerten (z. B. drei Durchflussmesser, um einen Gesamtdurchfluss zu berechnen).
- **Sollwert-Offset-Berechnung:** Berechnung eines Gesamtsollwerts basierend auf einem Basissollwert, einem Benutzer-Offset und einem Korrekturwert aus einer übergeordneten Steuerung.
- **Kombination von Signalquellen:** Zusammenführung von drei Teilsignalen in der Prozessautomatisierung unter Verwendung standardisierter Adapter-Strukturen.

Vergleich mit ähnlichen Bausteinen

- **Standard-ADD-Bausteine (IEC 61131-3):** Klassische ADD-Bausteine nutzen dedizierte Daten-Pins (z. B. IN1, IN2) und separate Event-Pins (REQ, CNF). AS_ADD_3 reduziert den Verdrahtungsaufwand im FBD (Function Block Diagram) durch die Nutzung von Adaptern auf ein Minimum.
- **AS_ADD_2 (Zwei-Wege-Addierer):** Im Vergleich zu einer Variante mit nur zwei Eingängen spart der AS_ADD_3 bei der Addition von drei Werten einen kompletten Baustein sowie die dazugehörige Zwischenverdrahtung ein.

- AS_ADD_3_UNGATED: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

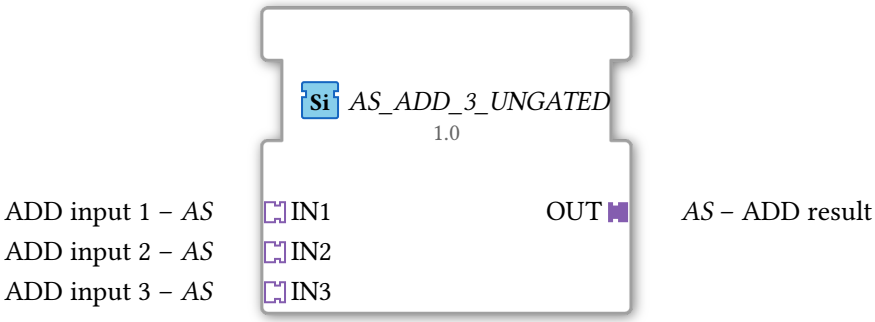
Fazit

Der AS_ADD_3 ist ein hocheffizienter, modularer Baustein zur Addition von drei Werten. Durch die konsequente Nutzung von unidirektionalen Adaptern fügt er sich perfekt in moderne, serviceorientierte und sauber strukturierte IEC 61499 Steuerungsanwendungen ein.

AS_ADD_3_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von AS_ADD_3. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weiter-

gegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



Einleitung

Der Funktionsbaustein `AS_ADD_3_UNGATED` ist ein generischer Funktionsbaustein (Generic FB) für die IEC 61499, der für die Durchführung einer arithmetischen Addition von drei Eingangswerten entwickelt wurde. Die Besonderheit dieses Bausteins liegt darin, dass er keine direkten Daten- und Ereignis-Ein-/Ausgänge besitzt, sondern vollständig auf der Verwendung von unidirektionalen Adaptern basiert. Dies vereinfacht die Modellierung und Verdrahtung in komplexeren Systemen erheblich.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Eingänge. Die Ereignissteuerung wird über die angeschlossenen Adapter abgewickelt.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Ausgänge. Die Weiterleitung von Ereignissen erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden.

Adapter

Der Baustein kommuniziert ausschließlich über Adapter-Schnittstellen.

Sockets (Buchsen - Eingänge):

- **IN1** (Typ: `adapter::types::unidirectional::AS`): Erster Eingang für die Addition.
- **IN2** (Typ: `adapter::types::unidirectional::AS`): Zweiter Eingang für die Addition.
- **IN3** (Typ: `adapter::types::unidirectional::AS`): Dritter Eingang für die Addition.

Plugs (Stecker - Ausgänge):

- **OUT** (Typ: `adapter::types::unidirectional::AS`): Ausgang, der das Ergebnis der Addition (`IN1 + IN2 + IN3`) bereitstellt.

Funktionsweise

Der Funktionsbaustein `AS_ADD_3_UNGATED` führt eine klassische Addition aus:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} \tag{2.75}$$

Da der Baustein als generischer Funktionsbaustein (GEN_AS_ADD) definiert ist, kann er je nach konkreter Instanziierung und dem im Adapter verwendeten Datentyp verschiedene numerische Datentypen (z. B. INT, REAL, DINT) verarbeiten.

Sobald an den Eingangs-Adaptern (IN1, IN2, IN3) neue Werte und die dazugehörigen Ereignisse signalisiert werden, berechnet der Baustein die Summe und triggert das entsprechende Ausgangsereignis am OUT-Adapter, um den neuen Wert an nachfolgende Bausteine weiterzugeben.

Technische Besonderheiten

- **Generischer Typ (Generic FB):** Der Baustein nutzt das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `'GEN_AS_ADD'`. Dies ermöglicht eine flexible Anpassung an unterschiedliche Datentypen, ohne dass für jeden Typ ein eigener Baustein erstellt werden muss.
- **Kapselung durch Adapter:** Durch die Verwendung von unidirektionalen Adaptern (AS) werden Daten und die dazugehörigen Trigger-Ereignisse in einer einzigen Verbindung gekapselt. Dies sorgt für ein sauberes und übersichtliches Oberflächendesign in der 4diac IDE.

Zustandsübersicht

Der Baustein arbeitet ereignisgesteuert und besitzt kein komplexes internes Zustandsdiagramm (ECC). Seine Ausführung wird durch die an den Sockets eingehenden Adapter-Ereignisse getriggert:

1. **Warten:** Der Baustein wartet auf ein Aktualisierungsereignis an den Adaptern IN1, IN2 oder IN3.
2. **Berechnung:** Bei Empfang eines Ereignisses werden die aktuellen Werte der drei Eingänge addiert.
3. **Ausgabe:** Das Ergebnis wird an den Plug OUT angelegt und ein Aktualisierungsereignis über den Adapter gesendet.

Anwendungsszenarien

- **Messwert-Summierung:** Addition von drei analogen Sensorwerten (z. B. drei Durchflussmesser, um einen Gesamtdurchfluss zu berechnen).
- **Sollwert-Offset-Berechnung:** Berechnung eines Gesamtsollwerts basierend auf einem Basissollwert, einem Benutzer-Offset und einem Korrekturwert aus einer übergeordneten Steuerung.
- **Kombination von Signalquellen:** Zusammenführung von drei Teilsignalen in der Prozessautomatisierung unter Verwendung standardisierter Adapter-Strukturen.

Vergleich mit ähnlichen Bausteinen

- **Standard-ADD-Bausteine (IEC 61131-3):** Klassische ADD-Bausteine nutzen dedizierte Daten-Pins (z. B. IN1, IN2) und separate Event-Pins (REQ, CNF). AS_ADD_3_UNGATED reduziert den Verdrahtungsaufwand im FBD (Function Block Diagram) durch die Nutzung von Adaptern auf ein Minimum.
- **AS_ADD_2 (Zwei-Wege-Addierer):** Im Vergleich zu einer Variante mit nur zwei Eingängen spart der AS_ADD_3_UNGATED bei der Addition von drei Werten einen kompletten Baustein sowie die dazugehörige Zwischenverdrahtung ein.

- AS_ADD_3: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

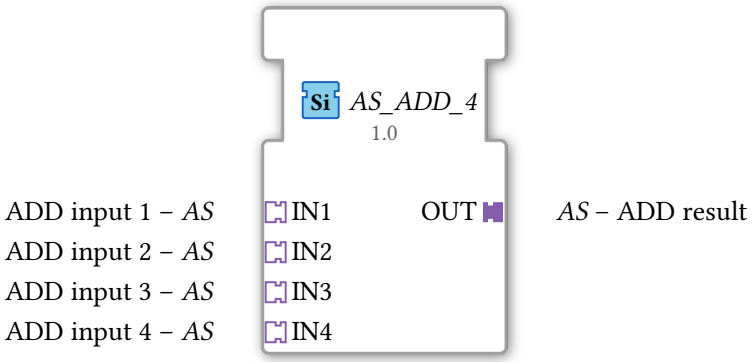
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der AS_ADD_3_UNGATED ist ein hocheffizienter, modularer Baustein zur Addition von drei Werten. Durch die konsequente Nutzung von unidirektionalen Adaptern fügt er sich perfekt in moderne, serviceorientierte und sauber strukturierte IEC 61499 Steuerungsanwendungen ein.

AS_ADD_4



(Bild des Funktionsbausteins nicht vorhanden)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AS_ADD_4` ist ein generischer Funktionsbaustein (FB), der für die Durchführung einer arithmetischen Addition von vier Eingangswerten entwickelt wurde. Die Besonderheit dieses Bausteins liegt in der Verwendung von unidirektionalen Adaptern (`AS`) für die Ein- und Ausgänge, was eine strukturierte und saubere Kapselung von Daten und zugehörigen Ereignissen in IEC 61499-Anwendungen ermöglicht.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung erfolgt implizit über die Adapter-Schnittstellen.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge vorhanden. Die Ereignisweiterleitung erfolgt implizit über den Ausgangs-Adapter.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübergabe erfolgt über die Eingangs-Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Das Ergebnis wird über den Ausgangs-Adapter bereitgestellt.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AS`): Erster Eingangswert (Summand 1) für die Addition.
- **IN2** (Typ: `adapter::types::unidirectional::AS`): Zweiter Eingangswert (Summand 2) für die Addition.
- **IN3** (Typ: `adapter::types::unidirectional::AS`): Dritter Eingangswert (Summand 3) für die Addition.
- **IN4** (Typ: `adapter::types::unidirectional::AS`): Vierter Eingangswert (Summand 4) für die Addition.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AS`): Ausgangsadapter zur Ausgabe des berechneten Additionsergebnisses.

Funktionsweise

Der Baustein führt eine mathematische Addition der über die vier Eingangs-Adapter (`IN1` bis `IN4`) empfangenen Werte aus. Die mathematische Formel lautet:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} + \mathrm{IN4} \tag{2.76}$$

Die Berechnung wird typischerweise getriggert, sobald an einem oder mehreren der Eingangs-Adapter ein entsprechendes Aktualisierungsereignis eintrifft. Nach erfolgreicher Berechnung wird das Ergebnis zusammen mit einem Trigger-Ereignis über den Ausgangs-Adapter `OUT` weitergeleitet.

Technische Besonderheiten

- **Generischer Charakter:** Der Baustein ist als generischer Baustein (`GEN_AS_ADD`) deklariert. Dies ermöglicht eine flexible Handhabung unterschiedlicher Datentypen (z. B. `INT`, `REAL`, `DINT`), je nachdem, wie die zugrundeliegenden Adaptertypen definiert sind.
- **Adapter-Kopplung:** Durch die Nutzung von unidirektionalen Adaptern wird das Risiko von fehlerhafter Pin-Verdrahtung in der 4diac-IDE reduziert, da Daten und Ereignisse in einer einzigen Verbindunglinie gebündelt übertragen werden.

Zustandsübersicht

Da es sich um einen algorithmischen Baustein zur Berechnung handelt, arbeitet der FB zustandslos bzw. ereignisgesteuert:

1. **Warten:** Der Baustein wartet auf ein Aktualisierungsereignis an den Eingangs-Adaptern `IN1` bis `IN4`.
2. **Berechnung:** Bei Eintreffen eines Ereignisses werden die aktuellen Werte der Eingänge addiert.
3. **Ausgabe:** Der neue Summenwert wird auf den Adapter `OUT` geschrieben, und das zugehörige Ausgangsereignis wird ausgelöst.

Anwendungsszenarien

- **Messwert-Summierung:** Zusammenfassung von vier analogen Sensorwerten (z. B. Durchflussmengen oder Leistungen) zu einem Gesamtwert.
- **Sollwert-Generierung:** Addition von Basis-Sollwerten mit verschiedenen Korrektur- oder Offset-Werten in einer Kaskadenregelung.
- **Strukturierte Signalverarbeitung:** Einsatz in komplexen Steuerungsarchitekturen, bei denen eine strikte Trennung von Logik und Datenfluss durch Adapter-Strukturen gefordert ist.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem klassischen, standardmäßigen `ADD`-Baustein (nach IEC 61131-3):

- **Vorteil:** Weniger Verdrahtungsaufwand in der grafischen Oberfläche, da keine separaten Event- und Datenlinien gezogen werden müssen.
- **Nachteil:** Erfordert die Definition und Verwendung des spezifischen Adaptertyps `adapter::types::unidirectional::AS`. Ein direkter Anschluss von Standard-Datentypen (wie einfachen `REAL`- oder `INT`-Variablen) ohne Adapter ist nicht möglich.

- **`AS_ADD_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

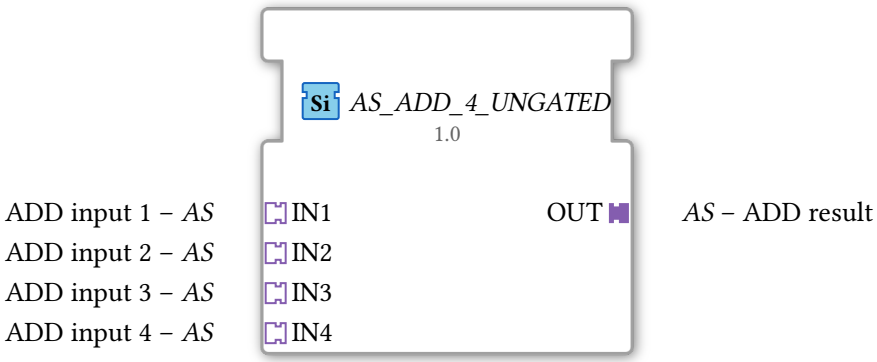
Fazit

Der `AS_ADD_4` ist ein spezialisierter, hocheffizienter Funktionsbaustein für die moderne, adapterbasierte Programmierung in 4diac-IDE. Er eignet sich hervorragend für Anwendungen, die ein hohes Maß an

Modularität und Übersichtlichkeit im Signalfluss erfordern und vier Summanden sauber und performant addieren müssen.

AS_ADD_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AS_ADD_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Bild des Funktionsbausteins nicht vorhanden)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AS_ADD_4_UNGATED` ist ein generischer Funktionsbaustein (FB), der für die Durchführung einer arithmetischen Addition von vier Eingangswerten entwickelt wurde. Die Besonderheit dieses Bausteins liegt in der Verwendung von unidirektionalen Adaptern (`AS`) für die Ein- und Ausgänge, was eine strukturierte und saubere Kapselung von Daten und zugehörigen Ereignissen in IEC 61499-Anwendungen ermöglicht.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung erfolgt implizit über die Adapter-Schnittstellen.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge vorhanden. Die Ereignisweiterleitung erfolgt implizit über den Ausgangs-Adapter.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübergabe erfolgt über die Eingangs-Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Das Ergebnis wird über den Ausgangs-Adapter bereitgestellt.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AS`): Erster Eingangswert (Summand 1) für die Addition.
- **IN2** (Typ: `adapter::types::unidirectional::AS`): Zweiter Eingangswert (Summand 2) für die Addition.

- **IN3** (Typ: `adapter::types::unidirectional::AS`): Dritter Eingangswert (Summand 3) für die Addition.
- **IN4** (Typ: `adapter::types::unidirectional::AS`): Vierter Eingangswert (Summand 4) für die Addition.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AS`): Ausgangsadapter zur Ausgabe des berechneten Additionsergebnisses.

Funktionsweise

Der Baustein führt eine mathematische Addition der über die vier Eingangs-Adapter (`IN1` bis `IN4`) empfangenen Werte aus. Die mathematische Formel lautet:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} + \mathrm{IN4} \tag{2.77}$$

Die Berechnung wird typischerweise getriggert, sobald an einem oder mehreren der Eingangs-Adapter ein entsprechendes Aktualisierungsereignis eintrifft. Nach erfolgreicher Berechnung wird das Ergebnis zusammen mit einem Trigger-Ereignis über den Ausgangs-Adapter `OUT` weitergeleitet.

Technische Besonderheiten

- **Generischer Charakter:** Der Baustein ist als generischer Baustein (`GEN_AS_ADD`) deklariert. Dies ermöglicht eine flexible Handhabung unterschiedlicher Datentypen (z. B. `INT`, `REAL`, `DINT`), je nachdem, wie die zugrundeliegenden Adaptertypen definiert sind.
- **Adapter-Kopplung:** Durch die Nutzung von unidirektionalen Adaptern wird das Risiko von fehlerhafter Pin-Verdrahtung in der 4diac-IDE reduziert, da Daten und Ereignisse in einer einzigen Verbindunglinie gebündelt übertragen werden.

Zustandsübersicht

Da es sich um einen algorithmischen Baustein zur Berechnung handelt, arbeitet der FB zustandslos bzw. ereignisgesteuert:

1. **Warten:** Der Baustein wartet auf ein Aktualisierungsereignis an den Eingangs-Adaptern `IN1` bis `IN4`.
2. **Berechnung:** Bei Eintreffen eines Ereignisses werden die aktuellen Werte der Eingänge addiert.
3. **Ausgabe:** Der neue Summenwert wird auf den Adapter `OUT` geschrieben, und das zugehörige Ausgangsereignis wird ausgelöst.

Anwendungsszenarien

- **Messwert-Summierung:** Zusammenfassung von vier analogen Sensorwerten (z. B. Durchflussmengen oder Leistungen) zu einem Gesamtwert.
- **Sollwert-Generierung:** Addition von Basis-Sollwerten mit verschiedenen Korrektur- oder Offset-Werten in einer Kaskadenregelung.
- **Strukturierte Signalverarbeitung:** Einsatz in komplexen Steuerungsarchitekturen, bei denen eine strikte Trennung von Logik und Datenfluss durch Adapter-Strukturen gefordert ist.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem klassischen, standardmäßigen `ADD`-Baustein (nach IEC 61131-3):

- **Vorteil:** Weniger Verdrahtungsaufwand in der grafischen Oberfläche, da keine separaten Event- und Datenlinien gezogen werden müssen.
- **Nachteil:** Erfordert die Definition und Verwendung des spezifischen Adaptertyps `adapter::types::unidirectional::AS`. Ein direkter Anschluss von Standard-Datentypen (wie einfachen `REAL`- oder `INT`-Variablen) ohne Adapter ist nicht möglich.

- **`AS_ADD_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

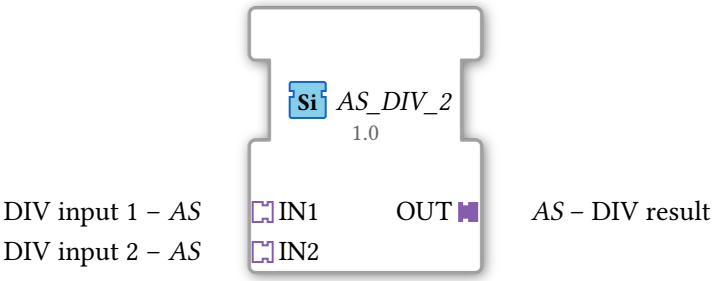
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AS_ADD_4_UNGATED` ist ein spezialisierter, hocheffizienter Funktionsbaustein für die moderne, adapterbasierte Programmierung in 4diac-IDE. Er eignet sich hervorragend für Anwendungen, die ein hohes Maß an Modularität und Übersichtlichkeit im Signalfluss erfordern und vier Summanden sauber und performant addieren müssen.

AS_DIV_2



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein (FB) `AS_DIV_2` ist ein generischer Baustein zur Durchführung einer arithmetischen Division im Kontext der IEC 61499-Architektur. Er nutzt ein adapterbasiertes Konzept zur Datenübertragung und ermöglicht eine saubere und strukturierte Signalverarbeitung, indem er Eingangsdaten über standardisierte Adapter entgegennimmt, die Division berechnet und das Ergebnis an einen Ausgangsadapter übergibt.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Eingänge. Die Steuerung und Triggerung erfolgt implizit über die angebundenen Adapter.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Eingänge.

Daten-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Ausgänge.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AS`):
  - Repräsentiert den ersten Eingangswert der Division (Dividend).
- **IN2** (Typ: `adapter::types::unidirectional::AS`):
  - Repräsentiert den zweiten Eingangswert der Division (Divisor).

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AS`):
  - Gibt das Ergebnis der Division (Quotient) aus.

Funktionsweise

Der Funktionsbaustein `AS_DIV_2` führt eine mathematische Division der übergebenen Eingangswerte durch. Die grundlegende Berechnungsformel lautet:

$$\mathrm{OUT} = \frac{\mathrm{IN1}}{\mathrm{IN2}} \tag{2.78}$$

Sobald sich die Werte an den Eingangs-Adaptern `IN1` oder `IN2` ändern bzw. ein entsprechendes Event über die Adapter übertragen wird, wird die Division durchgeführt und das Ergebnis über den Ausgangs-Adapter `OUT` bereitgestellt.

Technische Besonderheiten

- **Generischer Baustein:** Durch das Attribut `GenericClassName = 'GEN_AS_DIV'` ist der Baustein als generischer Typ definiert. Dies erlaubt eine hohe Flexibilität bei der Verarbeitung unterschiedlicher numerischer Datentypen.
- **Kapselung durch Adapter:** Anstelle von losen Daten- und Ereignisleitungen nutzt dieser Baustein ausschließlich unidirektionale Adapter des Typs `AS`. Dies sorgt für ein aufgeräumtes und übersichtliches Anwendungsdiagramm in der 4diac-IDE.
- **Division durch Null:** Bei der Implementierung und Verwendung ist darauf zu achten, dass der Divisor (`IN2`) ungleich Null sein muss, um einen Laufzeitfehler (Division by Zero) zu vermeiden.

Zustandsübersicht

Da es sich um einen rein mathematischen/arithmetischen Funktionsbaustein handelt, besitzt der `AS_DIV_2` keinen komplexen internen Zustandsautomaten (ECC). Die Ausgangswerte werden direkt in Abhängigkeit der an den Adaptern anliegenden Eingangssignale berechnet.

Anwendungsszenarien

- **Messwert-Skalierung:** Umrechnung und Normierung von analogen Sensorwerten (z. B. Division eines Rohwerts zur Umrechnung in eine physikalische Einheit).
- **Prozentuale Berechnungen:** Bestimmung von Verhältnissen oder prozentualen Anteilen in industriellen Prozessen.
- **Signalvorverarbeitung:** Mathematische Reduktion von Prozessdaten direkt an der Feldebene vor der Weiterleitung an übergeordnete Systeme.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu klassischen IEC 61131-3 `DIV`-Bausteinen, die mit direkten Daten-Pins arbeiten, bietet der `AS_DIV_2` durch die Nutzung von Adaptern eine wesentlich bessere Kapselung. Die Anzahl der Verbindungslinien im CFC (Continuous Function Chart) wird signifikant reduziert, was besonders bei großen und komplexen Steuerungsprogrammen die Wartbarkeit erhöht.

- **`AS_DIV_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

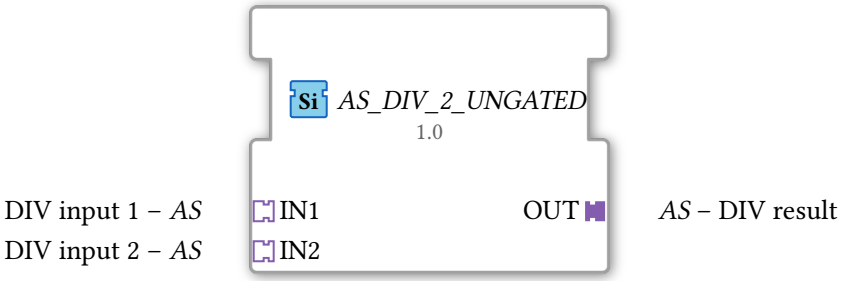
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `AS_DIV_2` ist ein robuster und flexibel einsetzbarer Baustein für arithmetische Divisionen in verteilten Steuerungssystemen. Durch die konsequente Nutzung des Adapter-Konzepts unterstützt er modernes, modulares Softwaredesign nach dem IEC 61499-Standard.

AS_DIV_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AS_DIV_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein (FB) `AS_DIV_2_UNGATED` ist ein generischer Baustein zur Durchführung einer arithmetischen Division im Kontext der IEC 61499-Architektur. Er nutzt ein adapterbasiertes Konzept zur Datenübertragung und ermöglicht eine saubere und strukturierte Signalverarbeitung, indem er Eingangsdaten über standardisierte Adapter entgegennimmt, die Division berechnet und das Ergebnis an einen Ausgangsadapter übergibt.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Eingänge. Die Steuerung und Triggerung erfolgt implizit über die angebundenen Adapter.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Ausgänge.

Daten-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Eingänge.

Daten-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Ausgänge.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AS`):
  - Repräsentiert den ersten Eingangswert der Division (Dividend).
- **IN2** (Typ: `adapter::types::unidirectional::AS`):
  - Repräsentiert den zweiten Eingangswert der Division (Divisor).

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AS`):
  - Gibt das Ergebnis der Division (Quotient) aus.

Funktionsweise

Der Funktionsbaustein `AS_DIV_2_UNGATED` führt eine mathematische Division der übergebenen Eingangswerte durch. Die grundlegende Berechnungsformel lautet:

$$\mathrm{OUT} = \frac{\mathrm{IN1}}{\mathrm{IN2}} \tag{2.79}$$

Sobald sich die Werte an den Eingangs-Adaptern `IN1` oder `IN2` ändern bzw. ein entsprechendes Event über die Adapter übertragen wird, wird die Division durchgeführt und das Ergebnis über den Ausgangs-Adapter `OUT` bereitgestellt.

Technische Besonderheiten

- **Generischer Baustein:** Durch das Attribut `GenericClassName = 'GEN_AS_DIV'` ist der Baustein als generischer Typ definiert. Dies erlaubt eine hohe Flexibilität bei der Verarbeitung unterschiedlicher numerischer Datentypen.
- **Kapselung durch Adapter:** Anstelle von losen Daten- und Ereignisleitungen nutzt dieser Baustein ausschließlich unidirektionale Adapter des Typs `AS`. Dies sorgt für ein aufgeräumtes und übersichtliches Anwendungsdiagramm in der 4diac-IDE.
- **Division durch Null:** Bei der Implementierung und Verwendung ist darauf zu achten, dass der Divisor (`IN2`) ungleich Null sein muss, um einen Laufzeitfehler (Division by Zero) zu vermeiden.

Zustandsübersicht

Da es sich um einen rein mathematischen/arithmetischen Funktionsbaustein handelt, besitzt der `AS_DIV_2_UNGATED` keinen komplexen internen Zustandsautomaten (ECC). Die Ausgangswerte werden direkt in Abhängigkeit der an den Adaptern anliegenden Eingangssignale berechnet.

Anwendungsszenarien

- **Messwert-Skalierung:** Umrechnung und Normierung von analogen Sensorwerten (z. B. Division eines Rohwerts zur Umrechnung in eine physikalische Einheit).
- **Prozentuale Berechnungen:** Bestimmung von Verhältnissen oder prozentualen Anteilen in industriellen Prozessen.
- **Signalvorverarbeitung:** Mathematische Reduktion von Prozessdaten direkt an der Feldebene vor der Weiterleitung an übergeordnete Systeme.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu klassischen IEC 61131-3 `DIV`-Bausteinen, die mit direkten Daten-Pins arbeiten, bietet der `AS_DIV_2_UNGATED` durch die Nutzung von Adaptern eine wesentlich bessere Kapselung. Die Anzahl der Verbindungslinien im CFC (Continuous Function Chart) wird signifikant reduziert, was besonders bei großen und komplexen Steuerungsprogrammen die Wartbarkeit erhöht.

- **`AS_DIV_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

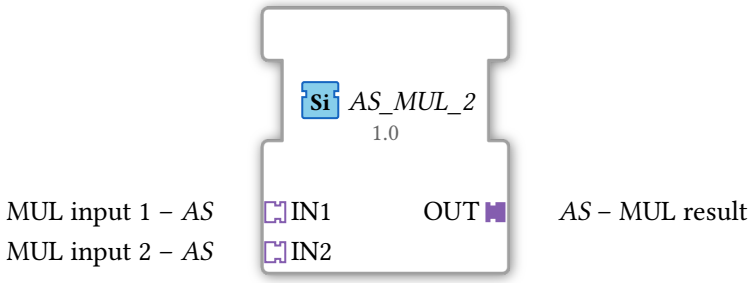
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AS_DIV_2_UNGATED` ist ein robuster und flexibel einsetzbarer Baustein für arithmetische Divisionen in verteilten Steuerungssystemen. Durch die konsequente Nutzung des Adapter-Konzepts unterstützt er modernes, modulares Softwaredesign nach dem IEC 61499-Standard.

AS_MUL_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AS_MUL_2` ist ein generischer, arithmetischer Multiplikationsbaustein für IEC 61499-Anwendungen in der 4diac-IDE. Er dient dazu, zwei Eingangswerte miteinander zu multiplizieren. Die Besonderheit dieses Bausteins liegt in der Verwendung von Adapterschnittstellen anstelle von klassischen, diskreten Event- und Datenkanälen. Dies ermöglicht eine strukturierte und übersichtliche Signalübertragung.

Schnittstellenstruktur

Der Baustein besitzt keine direkt herausgeführten standardmäßigen Event- oder Daten-Pins, sondern kapselt diese vollständig in Adaptern.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden (Ereignisse werden über die Adapter empfangen).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden (Ereignisse werden über die Adapter gesendet).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

- **Sockets (Eingangs-Schnittstellen):**
  - `IN1` (Typ: `adapter::types::unidirectional::AS`): Der erste Eingang (Multiplikand) für die Berechnung.
  - `IN2` (Typ: `adapter::types::unidirectional::AS`): Der zweite Eingang (Multiplikator) für die Berechnung.
- **Plugs (Ausgangs-Schnittstellen):**
  - `OUT` (Typ: `adapter::types::unidirectional::AS`): Das Ergebnis der Multiplikation ($OUT = IN1 \times IN2$).

Funktionsweise

Sobald über die Eingangsadapter `IN1` und `IN2` Daten sowie ein auslösendes Ereignis eingehen, führt der Baustein die Multiplikation aus. Das mathematische Produkt der beiden Eingangswerte wird berechnet und zusammen mit dem entsprechenden Trigger-Ereignis über den Ausgangsadapter `OUT` zur weiteren Verwendung im System bereitgestellt.

Da der Baustein generisch ausgelegt ist (`GenericClassName = 'GEN_AS_MUL'`), passt er sich flexibel an die zugrundeliegenden Datentypen der verbundenen Adapter an (z. B. `INT`, `REAL`, `DINT`).

Technische Besonderheiten

- **Generischer Baustein:** Durch die Definition als `GEN_AS_MUL` ist der Baustein nicht auf einen festen Datentyp fixiert.

- **Unidirektionale Adapterkopplung:** Die Verwendung des Adaptertyps `adapter::types::unidirectional::AS` sorgt für einen klaren, gerichteten Daten- und Signalfluss, wodurch Rückkopplungsschleifen vermieden werden.
- **Reduzierte Komplexität im Editor:** Durch die Kapselung von Daten und Events in Adaptern wird das visuelle „Spaghetticode"-Problem (zu viele Verbindungslinien) in der 4diac-IDE minimiert.

Zustandsübersicht

Der Baustein verhält sich rein ereignisgesteuert:

1. **Wartestatus:** Der Baustein wartet auf ein Ereignis an den Sockets `IN1` oder `IN2`.
2. **Berechnung:** Nach Eintreffen eines Ereignisses werden die aktuellen Werte aus `IN1` und `IN2` gelesen und multipliziert.
3. **Ausgabe:** Das Ergebnis wird an `OUT` angelegt, und ein Ausgangsereignis wird über den Plug `OUT` emittiert.

Anwendungsszenarien

- **Signal-Skalierung:** Multiplikation von analogen Sensorwerten mit einem Skalierungsfaktor, der über einen Adapter eingespeist wird.
- **Modulare Berechnungen:** Verwendung in komplexeren mathematischen Berechnungsnetzwerken, bei denen Datenströme strukturiert über Adapterbusse transportiert werden sollen.
- **Leistungsberechnung:** Multiplikation von Strom- und Spannungswerten zur Ermittlung der elektrischen Leistung in Echtzeitsystemen.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-`MUL`-Baustein der IEC 61131-3 / IEC 61499, welcher mit einzelnen Pins für `REQ`, `CNF`, `IN1`, `IN2` und `OUT` arbeitet, bündelt `AS_MUL_2` diese Signale in jeweils einem einzigen logischen Kanal pro Anschluss. Dies erhöht die Wiederverwendbarkeit und sorgt für ein saubereres Anwendungsdesign, erfordert jedoch die Definition und Verwendung passender Adaptertypen im Projekt.

- `AS_MUL_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

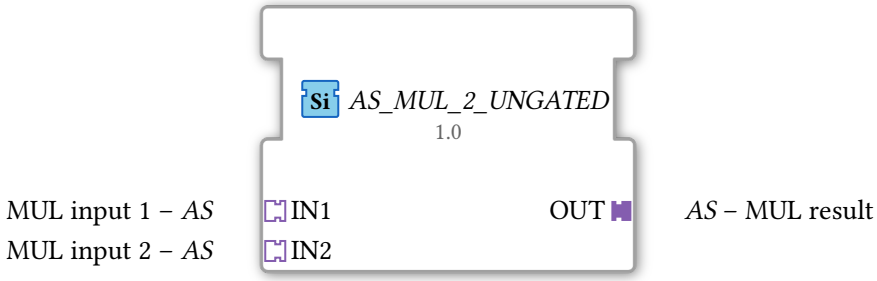
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `AS_MUL_2` ist ein moderner und flexibler Funktionsbaustein für arithmetische Operationen in IEC 61499-Steuerungsprogrammen. Dank der generischen Struktur und der konsequenten Nutzung von Adaptern eignet er sich hervorragend für saubere, modulare und gut wartbare Software-Architekturen in der industriellen Automatisierung.

AS_MUL_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AS_MUL_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AS_MUL_2_UNGATED` ist ein generischer, arithmetischer Multiplikationsbaustein für IEC 61499-Anwendungen in der 4diac-IDE. Er dient dazu, zwei Eingangswerte miteinander zu multiplizieren. Die Besonderheit dieses Bausteins liegt in der Verwendung von Adapterschnittstellen anstelle von klassischen, diskreten Event- und Datenkanälen. Dies ermöglicht eine strukturierte und übersichtliche Signalübertragung.

Schnittstellenstruktur

Der Baustein besitzt keine direkt herausgeführten standardmäßigen Event- oder Daten-Pins, sondern kapselt diese vollständig in Adaptern.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden (Ereignisse werden über die Adapter empfangen).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden (Ereignisse werden über die Adapter gesendet).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

- **Sockets (Eingangs-Schnittstellen):**
  - `IN1` (Typ: `adapter::types::unidirectional::AS`): Der erste Eingang (Multiplikand) für die Berechnung.
  - `IN2` (Typ: `adapter::types::unidirectional::AS`): Der zweite Eingang (Multiplikator) für die Berechnung.
- **Plugs (Ausgangs-Schnittstellen):**
  - `OUT` (Typ: `adapter::types::unidirectional::AS`): Das Ergebnis der Multiplikation ($OUT = IN1 \times IN2$).

Funktionsweise

Sobald über die Eingangsadapter `IN1` und `IN2` Daten sowie ein auslösendes Ereignis eingehen, führt der Baustein die Multiplikation aus. Das mathematische Produkt der beiden Eingangswerte wird berechnet und zusammen mit dem entsprechenden Trigger-Ereignis über den Ausgangsadapter `OUT` zur weiteren Verwendung im System bereitgestellt.

Da der Baustein generisch ausgelegt ist (`GenericClassName = 'GEN_AS_MUL'`), passt er sich flexibel an die zugrundeliegenden Datentypen der verbundenen Adapter an (z. B. `INT`, `REAL`, `DINT`).

Technische Besonderheiten

- **Generischer Baustein:** Durch die Definition als `GEN_AS_MUL` ist der Baustein nicht auf einen festen Datentyp fixiert.

- **Unidirektionale Adapterkopplung:** Die Verwendung des Adaptertyps `adapter::types::unidirectional::AS` sorgt für einen klaren, gerichteten Daten- und Signalfluss, wodurch Rückkopplungsschleifen vermieden werden.
- **Reduzierte Komplexität im Editor:** Durch die Kapselung von Daten und Events in Adaptern wird das visuelle „Spaghetticode"-Problem (zu viele Verbindungslinien) in der 4diac-IDE minimiert.

Zustandsübersicht

Der Baustein verhält sich rein ereignisgesteuert:

1. **Wartestatus:** Der Baustein wartet auf ein Ereignis an den Sockets `IN1` oder `IN2`.
2. **Berechnung:** Nach Eintreffen eines Ereignisses werden die aktuellen Werte aus `IN1` und `IN2` gelesen und multipliziert.
3. **Ausgabe:** Das Ergebnis wird an `OUT` angelegt, und ein Ausgangsereignis wird über den Plug `OUT` emittiert.

Anwendungsszenarien

- **Signal-Skalierung:** Multiplikation von analogen Sensorwerten mit einem Skalierungsfaktor, der über einen Adapter eingespeist wird.
- **Modulare Berechnungen:** Verwendung in komplexeren mathematischen Berechnungsnetzwerken, bei denen Datenströme strukturiert über Adapterbusse transportiert werden sollen.
- **Leistungsberechnung:** Multiplikation von Strom- und Spannungswerten zur Ermittlung der elektrischen Leistung in Echtzeitsystemen.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-`MUL`-Baustein der IEC 61131-3 / IEC 61499, welcher mit einzelnen Pins für `REQ`, `CNF`, `IN1`, `IN2` und `OUT` arbeitet, bündelt `AS_MUL_2_UNGATED` diese Signale in jeweils einem einzigen logischen Kanal pro Anschluss. Dies erhöht die Wiederverwendbarkeit und sorgt für ein saubereres Anwendungsdesign, erfordert jedoch die Definition und Verwendung passender Adaptertypen im Projekt.

- `AS_MUL_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

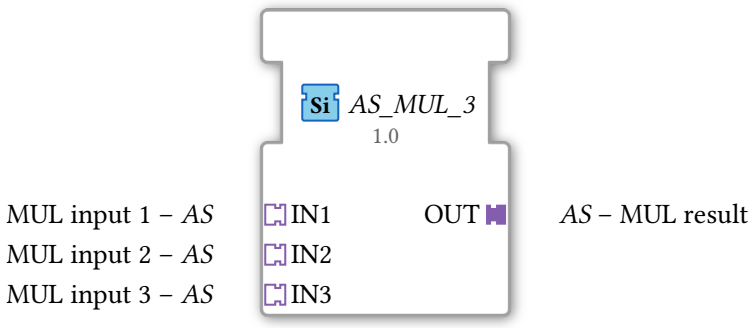
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AS_MUL_2_UNGATED` ist ein moderner und flexibler Funktionsbaustein für arithmetische Operationen in IEC 61499-Steuerungsprogrammen. Dank der generischen Struktur und der konsequenten Nutzung von Adaptern eignet er sich hervorragend für saubere, modulare und gut wartbare Software-Architekturen in der industriellen Automatisierung.

AS_MUL_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AS_MUL_3` ist ein generischer Funktionsblock (Generic FB) für die 4diac-IDE, der zur Durchführung einer arithmetischen Multiplikation mit drei Eingangswerten dient. Er basiert auf der Verwendung von unidirektionalen Adaptern (`adapter::types::unidirectional::AS`), was eine saubere Strukturierung und Kapselung von Daten und Kontrollflüssen in IEC 61499-Anwendungen ermöglicht.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Eingänge. Die Steuerung und Ereignisverarbeitung erfolgt implizit über die Adapter-Schnittstellen.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Ausgänge. Die Ereignisweiterleitung erfolgt implizit über die Adapter-Schnittstellen.

Daten-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Eingänge. Die Datenübergabe erfolgt über die Sockets der Adapter-Schnittstellen.

Daten-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Ausgänge. Die Datenausgabe erfolgt über den Plug der Adapter-Schnittstelle.

Adapter

Die gesamte Kommunikation des Bausteins wird über Adapter abgewickelt:

- **Sockets (Eingangs-Adapter):**
  - **IN1** (Typ: `adapter::types::unidirectional::AS`): Der erste Eingangswert (Multiplikand 1) für die Multiplikation.
  - **IN2** (Typ: `adapter::types::unidirectional::AS`): Der zweite Eingangswert (Multiplikand 2) für die Multiplikation.
  - **IN3** (Typ: `adapter::types::unidirectional::AS`): Der dritte Eingangswert (Multiplikand 3) für die Multiplikation.
- **Plugs (Ausgangs-Adapter):**
  - **OUT** (Typ: `adapter::types::unidirectional::AS`): Der Ausgangs-Adapter, der das Ergebnis der Multiplikation bereitstellt.

Funktionsweise

Der Funktionsbaustein `AS_MUL_3` multipliziert die über die Eingangs-Adapter `IN1`, `IN2` und `IN3` empfangenen Werte miteinander. Sobald sich Werte an den Eingängen ändern oder ein entsprechendes Trigger-Ereignis über die Adapter eingeht, wird die Berechnung ausgeführt und das Ergebnis an den Ausgangs-Adapter `OUT` übergeben.

Die mathematische Formel lautet:

$$\text{OUT} = \text{IN1} \times \text{IN2} \times \text{IN3} \tag{2.80}$$

Technische Besonderheiten

- **Generischer Typ:** Der Baustein ist als generischer Typ (`GEN_AS_MUL`) definiert. Dies ermöglicht eine flexible Anpassung an unterschiedliche numerische Datentypen (z. B. `INT`, `REAL`, `LREAL`), die vom zugrundeliegenden Adaptertyp unterstützt werden.
- **Adapter-basierte Architektur:** Durch die Nutzung von unidirektionalen Adaptern (`AS`) wird der Verkabelungsaufwand in der 4diac-IDE drastisch reduziert, da Ereignis- und Datenleitungen in einer einzigen Verbindung gebündelt sind.

Zustandsübersicht

Der Baustein verhält sich wie ein klassischer zustandsloser Berechnungsbaustein. Die Ausführung wird durch eingehende Ereignisse auf den Sockets (`IN1`, `IN2`, `IN3`) getriggert. Nach erfolgreicher Berechnung des Produkts wird das Ausgangsereignis direkt über den Plug `OUT` zusammen mit dem berechneten Wert emittiert.

Anwendungsszenarien

- **Sensorskalierung und Kalibrierung:** Berechnung von korrigierten Messwerten, bei denen ein Rohwert mit zwei verschiedenen Korrekturfaktoren multipliziert werden muss.
- **Volumenberechnungen:** Multiplikation von drei Dimensionen (Länge × Breite × Höhe) zur Bestimmung eines Volumens in der Prozesstechnik.
- **Mehrstufige Verstärkungsregelung:** Kaskadierte Signalverstärkung in der Regelungstechnik.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-Multiplikationsbausteinen (wie `MUL`), die mit separaten Event- und Datenleitungen (z. B. `REQ`/`CNF` und Standard-Datentypen) arbeiten, bietet `AS_MUL_3` durch die Kapselung in Adaptern eine deutlich aufgeräumtere visuelle Darstellung im Application Editor der 4diac-IDE. Es entfällt das manuelle Verknüpfen von „With“-Verbindungen zwischen Ereignissen und Daten.

- `AS_MUL_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

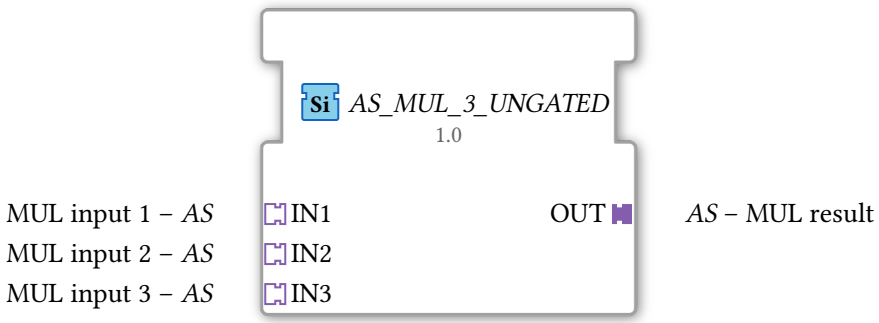
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

`AS_MUL_3` ist ein effizienter, modularer und wiederverwendbarer Funktionsbaustein für arithmetische Dreifach-Multiplikationen. Er eignet sich hervorragend für moderne, adapter-basierte Softwarearchitekturen innerhalb der IEC 61499 und trägt zur Übersichtlichkeit komplexer Steuerungsanwendungen bei.

AS_MUL_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AS_MUL_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AS_MUL_3_UNGATED` ist ein generischer Funktionsblock (Generic FB) für die 4diac-IDE, der zur Durchführung einer arithmetischen Multiplikation mit drei Eingangswerten dient. Er basiert auf der Verwendung von unidirektionalen Adaptern (`adapter::types::unidirectional::AS`), was eine saubere Strukturierung und Kapselung von Daten und Kontrollflüssen in IEC 61499-Anwendungen ermöglicht.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Eingänge. Die Steuerung und Ereignisverarbeitung erfolgt implizit über die Adapter-Schnittstellen.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Ausgänge. Die Ereignisweiterleitung erfolgt implizit über die Adapter-Schnittstellen.

Daten-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Eingänge. Die Datenübergabe erfolgt über die Sockets der Adapter-Schnittstellen.

Daten-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Ausgänge. Die Datenausgabe erfolgt über den Plug der Adapter-Schnittstelle.

Adapter

Die gesamte Kommunikation des Bausteins wird über Adapter abgewickelt:

- **Sockets (Eingangs-Adapter):**
  - **IN1** (Typ: `adapter::types::unidirectional::AS`): Der erste Eingangswert (Multiplikand 1) für die Multiplikation.
  - **IN2** (Typ: `adapter::types::unidirectional::AS`): Der zweite Eingangswert (Multiplikand 2) für die Multiplikation.
  - **IN3** (Typ: `adapter::types::unidirectional::AS`): Der dritte Eingangswert (Multiplikand 3) für die Multiplikation.
- **Plugs (Ausgangs-Adapter):**
  - **OUT** (Typ: `adapter::types::unidirectional::AS`): Der Ausgangs-Adapter, der das Ergebnis der Multiplikation bereitstellt.

Funktionsweise

Der Funktionsbaustein `AS_MUL_3_UNGATED` multipliziert die über die Eingangs-Adapter `IN1`, `IN2` und `IN3` empfangenen Werte miteinander. Sobald sich Werte an den Eingängen ändern oder ein entsprechendes Trigger-Ereignis über die Adapter eingeht, wird die Berechnung ausgeführt und das Ergebnis an den Ausgangs-Adapter `OUT` übergeben.

Die mathematische Formel lautet:

$$\text{OUT} = \text{IN1} \times \text{IN2} \times \text{IN3} \tag{2.81}$$

Technische Besonderheiten

- **Generischer Typ:** Der Baustein ist als generischer Typ (`GEN_AS_MUL`) definiert. Dies ermöglicht eine flexible Anpassung an unterschiedliche numerische Datentypen (z. B. `INT`, `REAL`, `LREAL`), die vom zugrundeliegenden Adaptertyp unterstützt werden.
- **Adapter-basierte Architektur:** Durch die Nutzung von unidirektionalen Adaptern (`AS`) wird der Verkabelungsaufwand in der 4diac-IDE drastisch reduziert, da Ereignis- und Datenleitungen in einer einzigen Verbindung gebündelt sind.

Zustandsübersicht

Der Baustein verhält sich wie ein klassischer zustandsloser Berechnungsbaustein. Die Ausführung wird durch eingehende Ereignisse auf den Sockets (`IN1`, `IN2`, `IN3`) getriggert. Nach erfolgreicher Berechnung des Produkts wird das Ausgangsereignis direkt über den Plug `OUT` zusammen mit dem berechneten Wert emittiert.

Anwendungsszenarien

- **Sensorskalierung und Kalibrierung:** Berechnung von korrigierten Messwerten, bei denen ein Rohwert mit zwei verschiedenen Korrekturfaktoren multipliziert werden muss.
- **Volumenberechnungen:** Multiplikation von drei Dimensionen (Länge × Breite × Höhe) zur Bestimmung eines Volumens in der Prozesstechnik.
- **Mehrstufige Verstärkungsregelung:** Kaskadierte Signalverstärkung in der Regelungstechnik.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-Multiplikationsbausteinen (wie `MUL`), die mit separaten Event- und Datenleitungen (z. B. `REQ`/`CNF` und Standard-Datentypen) arbeiten, bietet `AS_MUL_3_UNGATED` durch die Kapselung in Adaptern eine deutlich aufgeräumtere visuelle Darstellung im Application Editor der 4diac-IDE. Es entfällt das manuelle Verknüpfen von „With“-Verbindungen zwischen Ereignissen und Daten.

- `AS_MUL_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

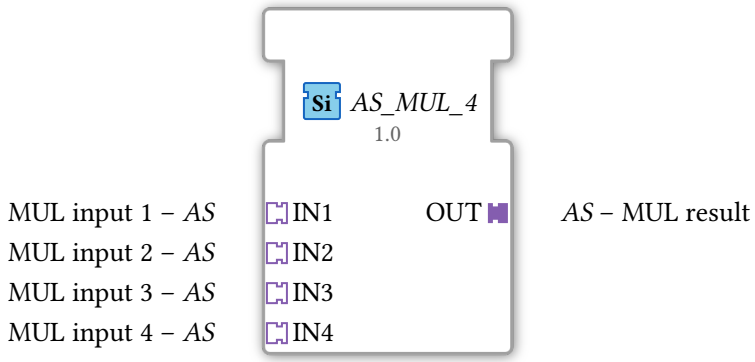
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

`AS_MUL_3_UNGATED` ist ein effizienter, modularer und wiederverwendbarer Funktionsbaustein für arithmetische Dreifach-Multiplikationen. Er eignet sich hervorragend für moderne, adapter-basierte Softwarearchitekturen innerhalb der IEC 61499 und trägt zur Übersichtlichkeit komplexer Steuerungsanwendungen bei.

AS_MUL_4



(Kein Bild vorhanden)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AS_MUL_4` ist ein generischer Funktionsbaustein (Generic FB) für die IEC 61499, der für die arithmetische Multiplikation von vier Eingangswerten entwickelt wurde. Anstelle von klassischen Daten- und Ereignispins nutzt dieser Baustein Adapterverbindungen (`AS` - unidirektional), um Daten und die zugehörige Ausführungssteuerung gekapselt zu übertragen. Dies sorgt für ein übersichtlicheres Design in der 4diac-IDE, da die Anzahl der sichtbaren Verbindungslinien drastisch reduziert wird.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten, eigenständigen Ereignis-Eingänge. Die Ereignissteuerung wird vollständig über die angeschlossenen Adapter abgewickelt.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten, eigenständigen Ereignis-Ausgänge. Die Ereignissteuerung für nachfolgende Bausteine wird über den Ausgangs-Adapter übertragen.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübergabe erfolgt über die Eingangs-Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Das Ergebnis wird über den Ausgangs-Adapter bereitgestellt.

Adapter

Der Baustein basiert vollständig auf einer adapterbasierten Kommunikation.

Sockets (Steckdosen / Eingänge)

- **IN1** (Typ: `adapter::types::unidirectional::AS`): Erster Multiplikand (Eingang 1).
- **IN2** (Typ: `adapter::types::unidirectional::AS`): Zweiter Multiplikand (Eingang 2).
- **IN3** (Typ: `adapter::types::unidirectional::AS`): Dritter Multiplikand (Eingang 3).
- **IN4** (Typ: `adapter::types::unidirectional::AS`): Vierter Multiplikand (Eingang 4).

Plugs (Stecker / Ausgänge)

- **OUT** (Typ: `adapter::types::unidirectional::AS`): Ergebnis der Multiplikation (`IN1 * IN2 * IN3 * IN4`).

Funktionsweise

Sobald über die Adapter-Sockets neue Werte bzw. ein entsprechendes Auslösungsereignis empfangen werden, führt der Baustein die arithmetische Multiplikation der vier Eingangswerte durch:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \times \mathrm{IN4} \tag{2.82}$$

Das Ergebnis sowie das zugehörige Ausgangsereignis werden anschließend über den Adapter-Plug `OUT` an die nachfolgenden Logikglieder weitergegeben.

Technische Besonderheiten

- **Generischer Charakter:** Durch die Zuordnung zur generischen Klasse `GEN_AS_MUL` kann sich der Baustein flexibel an verschiedene numerische Datentypen (z. B. `INT`, `REAL`, `LREAL`) anpassen, sofern die angeschlossenen Adapter vom Typ `AS` denselben Datentyp führen.
- **Unidirektionale Adapter:** Die Verwendung des Typs `adapter::types::unidirectional::AS` stellt sicher, dass der Datenfluss klar und gerichtet von den Eingängen zum Ausgang verläuft.
- **Kompaktes Design:** Durch das Kapseln von Daten- und Event-Kanälen in Adaptern bleibt das Anwendungsdiagramm in 4diac übersichtlich.

Zustandsübersicht

Da es sich bei `AS_MUL_4` um einen mathematischen, zustandslosen Funktionsbaustein handelt, existiert kein komplexes internes Zustandsdiagramm (ECC).

- **Bereit / Idle:** Der Baustein wartet auf eingehende Ereignisse an den Sockets `IN1` bis `IN4`.
- **Berechnung:** Bei einem eingehenden Trigger-Ereignis werden die Werte aktualisiert, multipliziert und direkt an `OUT` übergeben.

Anwendungsszenarien

- **Skalierung von Messwerten:** Multiplikation eines Sensorwerts mit mehreren Kalibrierungs- und Korrekturfaktoren.
- **Volumenberechnungen:** Berechnung eines Volumens aus drei Dimensionen (Länge × Breite × Höhe) multipliziert mit einem Dichtefaktor.
- **Kaskadierte Verstärkungsglieder:** Berechnung von Gesamtkreisverstärkungen in der Regelungstechnik, bei denen vier unterschiedliche Verstärkungsfaktoren (Gains) multipliziert werden müssen.

Vergleich mit ähnlichen Bausteinen

- **Standard-`MUL`-Baustein (IEC 61131-3):** Dieser benötigt für jeden Wert separate Datenleitungen und mindestens ein Trigger-Ereignis (REQ/CNF). `AS_MUL_4` vereinfacht dies durch die Nutzung von Adaptern auf nur fünf Verbindungen (4 Eingangs-Adapter, 1 Ausgangs-Adapter).
- **`AS_MUL_2` / `AS_MUL_3`:** Bieten dieselbe adapterbasierte Funktionalität, sind jedoch auf zwei bzw. drei Eingangswerte begrenzt. `AS_MUL_4` spart bei der Verknüpfung von vier Faktoren zusätzliche Zwischenschritte und temporäre Hilfsvariablen.

- `AS_MUL_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

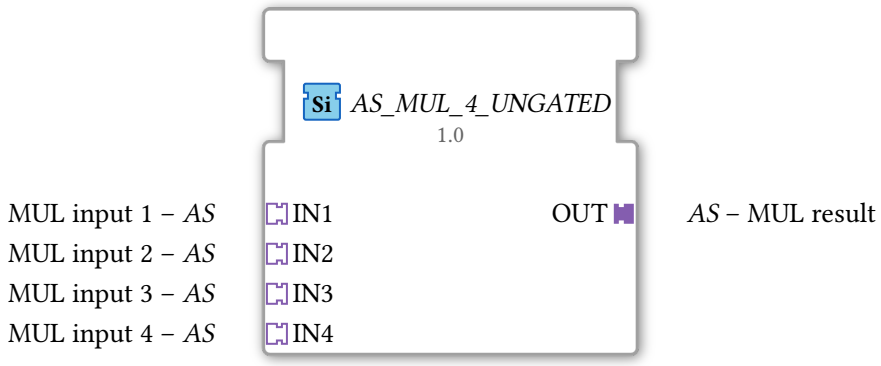
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `AS_MUL_4` ist ein hocheffizienter, übersichtlicher und moderner Funktionsbaustein für arithmetische Berechnungen in der IEC 61499. Durch die konsequente Nutzung von unidirektionalen Adaptern wird das Application-Design aufgeräumt gehalten, während gleichzeitig die volle Flexibilität eines generischen Datentyps erhalten bleibt.

AS_MUL_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AS_MUL_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild vorhanden)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AS_MUL_4_UNGATED` ist ein generischer Funktionsbaustein (Generic FB) für die IEC 61499, der für die arithmetische Multiplikation von vier Eingangswerten entwickelt wurde. Anstelle von klassischen Daten- und Ereignispins nutzt dieser Baustein Adapterverbindungen (`AS` - unidirektional), um Daten und die zugehörige Ausführungssteuerung gekapselt zu übertragen. Dies sorgt für ein übersichtlicheres Design in der 4diac-IDE, da die Anzahl der sichtbaren Verbindungslinien drastisch reduziert wird.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten, eigenständigen Ereignis-Eingänge. Die Ereignissteuerung wird vollständig über die angeschlossenen Adapter abgewickelt.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten, eigenständigen Ereignis-Ausgänge. Die Ereignissteuerung für nachfolgende Bausteine wird über den Ausgangs-Adapter übertragen.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübergabe erfolgt über die Eingangs-Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Das Ergebnis wird über den Ausgangs-Adapter bereitgestellt.

Adapter

Der Baustein basiert vollständig auf einer adapterbasierten Kommunikation.

Sockets (Steckdosen / Eingänge)

- **IN1** (Typ: `adapter::types::unidirectional::AS`): Erster Multiplikand (Eingang 1).
- **IN2** (Typ: `adapter::types::unidirectional::AS`): Zweiter Multiplikand (Eingang 2).
- **IN3** (Typ: `adapter::types::unidirectional::AS`): Dritter Multiplikand (Eingang 3).
- **IN4** (Typ: `adapter::types::unidirectional::AS`): Vierter Multiplikand (Eingang 4).

Plugs (Stecker / Ausgänge)

- **OUT** (Typ: `adapter::types::unidirectional::AS`): Ergebnis der Multiplikation (`IN1 * IN2 * IN3 * IN4`).

Funktionsweise

Sobald über die Adapter-Sockets neue Werte bzw. ein entsprechendes Auslösungsereignis empfangen werden, führt der Baustein die arithmetische Multiplikation der vier Eingangswerte durch:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \times \mathrm{IN4} \tag{2.83}$$

Das Ergebnis sowie das zugehörige Ausgangsereignis werden anschließend über den Adapter-Plug `OUT` an die nachfolgenden Logikglieder weitergegeben.

Technische Besonderheiten

- **Generischer Charakter:** Durch die Zuordnung zur generischen Klasse `GEN_AS_MUL` kann sich der Baustein flexibel an verschiedene numerische Datentypen (z. B. `INT`, `REAL`, `LREAL`) anpassen, sofern die angeschlossenen Adapter vom Typ `AS` denselben Datentyp führen.
- **Unidirektionale Adapter:** Die Verwendung des Typs `adapter::types::unidirectional::AS` stellt sicher, dass der Datenfluss klar und gerichtet von den Eingängen zum Ausgang verläuft.
- **Kompaktes Design:** Durch das Kapseln von Daten- und Event-Kanälen in Adaptern bleibt das Anwendungsdiagramm in 4diac übersichtlich.

Zustandsübersicht

Da es sich bei `AS_MUL_4_UNGATED` um einen mathematischen, zustandslosen Funktionsbaustein handelt, existiert kein komplexes internes Zustandsdiagramm (ECC).

- **Bereit / Idle:** Der Baustein wartet auf eingehende Ereignisse an den Sockets `IN1` bis `IN4`.
- **Berechnung:** Bei einem eingehenden Trigger-Ereignis werden die Werte aktualisiert, multipliziert und direkt an `OUT` übergeben.

Anwendungsszenarien

- **Skalierung von Messwerten:** Multiplikation eines Sensorwerts mit mehreren Kalibrierungs- und Korrekturfaktoren.
- **Volumenberechnungen:** Berechnung eines Volumens aus drei Dimensionen (Länge × Breite × Höhe) multipliziert mit einem Dichtefaktor.
- **Kaskadierte Verstärkungsglieder:** Berechnung von Gesamtkreisverstärkungen in der Regelungstechnik, bei denen vier unterschiedliche Verstärkungsfaktoren (Gains) multipliziert werden müssen.

Vergleich mit ähnlichen Bausteinen

- **Standard-`MUL`-Baustein (IEC 61131-3):** Dieser benötigt für jeden Wert separate Datenleitungen und mindestens ein Trigger-Ereignis (REQ/CNF). `AS_MUL_4_UNGATED` vereinfacht dies durch die Nutzung von Adaptern auf nur fünf Verbindungen (4 Eingangs-Adapter, 1 Ausgangs-Adapter).
- **`AS_MUL_2` / `AS_MUL_3`:** Bieten dieselbe adapterbasierte Funktionalität, sind jedoch auf zwei bzw. drei Eingangswerte begrenzt. `AS_MUL_4_UNGATED` spart bei der Verknüpfung von vier Faktoren zusätzliche Zwischenschritte und temporäre Hilfsvariablen.

- `AS_MUL_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

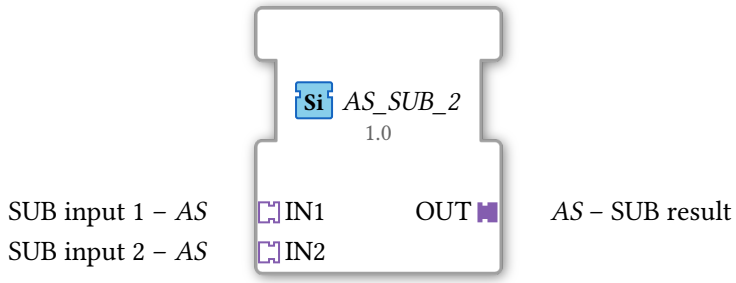
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AS_MUL_4_UNGATED` ist ein hocheffizienter, übersichtlicher und moderner Funktionsbaustein für arithmetische Berechnungen in der IEC 61499. Durch die konsequente Nutzung von unidirektionalen Adaptern wird das Application-Design aufgeräumt gehalten, während gleichzeitig die volle Flexibilität eines generischen Datentyps erhalten bleibt.

AS_SUB_2



(Kein Bild vorhanden)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AS_SUB_2` ist ein generischer Funktionsbaustein (Generic FB) für die Entwicklungsumgebung 4diac-ide. Er dient der Durchführung einer arithmetischen Subtraktion ($OUT = IN1 - IN2$).

Im Gegensatz zu klassischen arithmetischen Bausteinen, die direkt mit standardmäßigen Daten- und Ereignis-Schnittstellen arbeiten, nutzt dieser Baustein unidirektionale Adapterverbindungen zur Kapselung von Daten und Steuerungsereignissen. Dies ermöglicht eine strukturierte und übersichtliche Verdrahtung in komplexen IEC 61499-Anwendungen.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsbaustein besitzt keine direkten, eigenständigen Ereignis-Eingänge. Die Ereignissteuerung wird vollständig über die angeschlossenen Adapter abgewickelt.

Ereignis-Ausgänge

Der Funktionsbaustein besitzt keine direkten, eigenständigen Ereignis-Ausgänge. Die Weiterleitung von Ereignissen erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge auf dem Baustein definiert. Die Eingangsdaten werden über die Adapter-Schnittstellen bereitgestellt.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge auf dem Baustein definiert. Das Ergebnis wird über den Ausgangs-Adapter übergeben.

Adapter

Die gesamte Kommunikation des Funktionsbausteins wird über Adapter realisiert:

- **Sockets (Eingangs-Adapter):**
  - `IN1` (Typ: `adapter::types::unidirectional::AS`): Der erste Eingang (Minuend) für die Subtraktion.
  - `IN2` (Typ: `adapter::types::unidirectional::AS`): Der zweite Eingang (Subtrahend) für die Subtraktion.
- **Plugs (Ausgangs-Adapter):**
  - `OUT` (Typ: `adapter::types::unidirectional::AS`): Das Ergebnis (Differenz) der arithmetischen Subtraktion.

Funktionsweise

Sobald an den Eingangs-Adaptern `IN1` und `IN2` gültige Werte und die entsprechenden Trigger-Ereignisse anliegen, führt der Baustein die Berechnung aus.

Die mathematische Operation lautet:

$$\text{Ergebnis} = \text{Wert von } \ IN1 - \text{Wert von } \ IN2 \tag{2.84}$$

Das Ergebnis der Berechnung sowie das zugehörige Ausgangs-Ereignis werden anschließend über den Ausgangs-Adapter `OUT` an nachfolgende Bausteine weitergeleitet. Da es sich um einen generischen Baustein (`GenericClassName="GEN_AS_SUB"`) handelt, ist er flexibel bezüglich der zugrundeliegenden Datentypen, sofern diese vom Adaptertyp unterstützt werden.

Technische Besonderheiten

- **Generisches Verhalten:** Durch die Deklaration als generischer Baustein kann er für verschiedene numerische Datentypen (z. B. INT, REAL, LREAL) eingesetzt werden, abhängig von der Definition des verwendeten Adapters `adapter::types::unidirectional::AS`.
- **Kompakte Struktur:** Durch die Verwendung von Adaptern wird die visuelle Komplexität im 4diac-Application-Editor stark reduziert, da Ereignis- und Datenleitungen in einer einzigen Adapterverbindung gebündelt sind.
- **Unidirektionaler Datenfluss:** Die verwendeten Adapter weisen einen strikt unidirektionalen Signalfluss auf (von den Sockets zum Plug).

Zustandsübersicht

Die Ausführungslogik wird typischerweise durch Ereignisse auf den Eingangs-Adaptern getriggert:

1. **Wartezustand (Idle):** Der Baustein wartet auf ein Ereignis an `IN1` oder `IN2`.
2. **Berechnung (Calculation):** Nach dem Eintreffen eines Triggers werden die Daten aus den Adaptern gelesen und die Differenz gebildet.
3. **Ausgabe (Dispatch):** Das Ergebnis wird in den Ausgangs-Adapter geschrieben und das Ausgangs-Ereignis an `OUT` ausgelöst. Der Baustein kehrt in den Wartezustand zurück.

Anwendungsszenarien

- **Messwertkorrektur:** Subtraktion von Nullpunktfehlern oder Offsets bei analogen Sensorwerten.
- **Soll-Ist-Vergleich:** Berechnung der Regelabweichung ($e = w - x$) in Regelungskreisen, bei denen Soll- und Istwerte über standardisierte Adapterstrukturen übertragen werden.
- **Füllstands- und Mengenberechnung:** Ermittlung von Differenzmengen in Prozessanlagen.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-Subtraktionsbaustein (z. B. `SUB` aus der IEC 61131-3-Bibliothek), der mit expliziten Eingängen wie `REQ`, `IN1` und `IN2` arbeitet, bietet der `AS_SUB_2` durch die Kapselung in Adaptern eine wesentlich höhere Modularität. Er eignet sich besonders für fortgeschrittene Softwarearchitekturen in IEC 61499, bei denen Daten- und Signalflüsse als logische Einheiten (Schnittstellen) zusammengefasst werden sollen.

- `AS_SUB_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

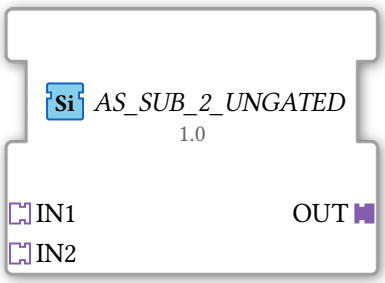
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `AS_SUB_2` ist ein hocheffizienter, wiederverwendbarer und übersichtlicher Funktionsbaustein zur Realisierung von Subtraktionsaufgaben. Seine Stärke liegt in der konsequenten Nutzung von Adaptern, was zu saubereren Software-Designs und einer verbesserten Wartbarkeit innerhalb der 4diac-ide führt.

AS_SUB_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AS_SUB_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



SUB input 1 – *AS*
SUB input 2 – *AS*
AS – SUB result

(Kein Bild vorhanden)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AS_SUB_2_UNGATED` ist ein generischer Funktionsbaustein (Generic FB) für die Entwicklungsumgebung 4diac-ide. Er dient der Durchführung einer arithmetischen Subtraktion ($OUT = IN1 - IN2$).

Im Gegensatz zu klassischen arithmetischen Bausteinen, die direkt mit standardmäßigen Daten- und Ereignis-Schnittstellen arbeiten, nutzt dieser Baustein unidirektionale Adapterverbindungen zur Kapselung von Daten und Steuerungsereignissen. Dies ermöglicht eine strukturierte und übersichtliche Verdrahtung in komplexen IEC 61499-Anwendungen.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsbaustein besitzt keine direkten, eigenständigen Ereignis-Eingänge. Die Ereignissteuerung wird vollständig über die angeschlossenen Adapter abgewickelt.

Ereignis-Ausgänge

Der Funktionsbaustein besitzt keine direkten, eigenständigen Ereignis-Ausgänge. Die Weiterleitung von Ereignissen erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge auf dem Baustein definiert. Die Eingangsdaten werden über die Adapter-Schnittstellen bereitgestellt.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge auf dem Baustein definiert. Das Ergebnis wird über den Ausgangs-Adapter übergeben.

Adapter

Die gesamte Kommunikation des Funktionsbausteins wird über Adapter realisiert:

- **Sockets (Eingangs-Adapter):**
  - `IN1` (Typ: `adapter::types::unidirectional::AS`): Der erste Eingang (Minuend) für die Subtraktion.
  - `IN2` (Typ: `adapter::types::unidirectional::AS`): Der zweite Eingang (Subtrahend) für die Subtraktion.
- **Plugs (Ausgangs-Adapter):**
  - `OUT` (Typ: `adapter::types::unidirectional::AS`): Das Ergebnis (Differenz) der arithmetischen Subtraktion.

Funktionsweise

Sobald an den Eingangs-Adaptern `IN1` und `IN2` gültige Werte und die entsprechenden Trigger-Ereignisse anliegen, führt der Baustein die Berechnung aus.

Die mathematische Operation lautet:

$$\text{Ergebnis} = \text{Wert von } \ IN1 - \text{Wert von } \ IN2 \tag{2.85}$$

Das Ergebnis der Berechnung sowie das zugehörige Ausgangs-Ereignis werden anschließend über den Ausgangs-Adapter `OUT` an nachfolgende Bausteine weitergeleitet. Da es sich um einen generischen Baustein (`GenericClassName="GEN_AS_SUB"`) handelt, ist er flexibel bezüglich der zugrundeliegenden Datentypen, sofern diese vom Adaptertyp unterstützt werden.

Technische Besonderheiten

- **Generisches Verhalten:** Durch die Deklaration als generischer Baustein kann er für verschiedene numerische Datentypen (z. B. INT, REAL, LREAL) eingesetzt werden, abhängig von der Definition des verwendeten Adapters `adapter::types::unidirectional::AS`.
- **Kompakte Struktur:** Durch die Verwendung von Adaptern wird die visuelle Komplexität im 4diac-Application-Editor stark reduziert, da Ereignis- und Datenleitungen in einer einzigen Adapterverbindung gebündelt sind.
- **Unidirektionaler Datenfluss:** Die verwendeten Adapter weisen einen strikt unidirektionalen Signalfluss auf (von den Sockets zum Plug).

Zustandsübersicht

Die Ausführungslogik wird typischerweise durch Ereignisse auf den Eingangs-Adaptern getriggert:

1. **Wartezustand (Idle):** Der Baustein wartet auf ein Ereignis an `IN1` oder `IN2`.
2. **Berechnung (Calculation):** Nach dem Eintreffen eines Triggers werden die Daten aus den Adaptern gelesen und die Differenz gebildet.
3. **Ausgabe (Dispatch):** Das Ergebnis wird in den Ausgangs-Adapter geschrieben und das Ausgangs-Ereignis an `OUT` ausgelöst. Der Baustein kehrt in den Wartezustand zurück.

Anwendungsszenarien

- **Messwertkorrektur:** Subtraktion von Nullpunktfehlern oder Offsets bei analogen Sensorwerten.
- **Soll-Ist-Vergleich:** Berechnung der Regelabweichung ($e = w - x$) in Regelungskreisen, bei denen Soll- und Istwerte über standardisierte Adapterstrukturen übertragen werden.
- **Füllstands- und Mengenberechnung:** Ermittlung von Differenzmengen in Prozessanlagen.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-Subtraktionsbaustein (z. B. `SUB` aus der IEC 61131-3-Bibliothek), der mit expliziten Eingängen wie `REQ`, `IN1` und `IN2` arbeitet, bietet der `AS_SUB_2_UNGATED` durch die Kapselung in Adaptern eine wesentlich höhere Modularität. Er eignet sich besonders für fortgeschrittene Softwarearchitekturen in IEC 61499, bei denen Daten- und Signalflüsse als logische Einheiten (Schnittstellen) zusammengefasst werden sollen.

- `AS_SUB_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

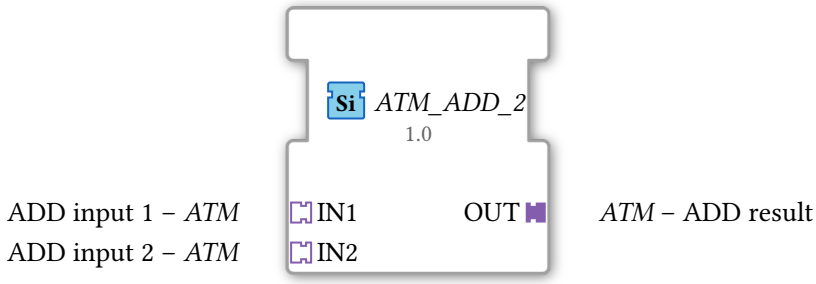
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AS_SUB_2_UNGATED` ist ein hocheffizienter, wiederverwendbarer und übersichtlicher Funktionsbaustein zur Realisierung von Subtraktionsaufgaben. Seine Stärke liegt in der konsequenten Nutzung von Adaptern, was zu saubereren Software-Designs und einer verbesserten Wartbarkeit innerhalb der 4diac-ide führt.

ATM_ADD_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **ATM_ADD_2** dient der Berechnung der arithmetischen Addition zweier Zeitwerte (Typ `TIME`). Er ist als generischer FB ausgelegt und realisiert die grundlegende Rechenoperation über Adapter-Schnittstellen. Der Baustein ist plattformunabhängig und folgt dem IEC-61499-Standard.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Bezeichnung	Richtung	Typ	Kommentar
IN1	Socket	adapter::types::unidirectional::ATM	ADD input 1
IN2	Socket	adapter::types::unidirectional::ATM	ADD input 2
OUT	Plug	adapter::types::unidirectional::ATM	ADD result

Die Adapter sind vom Typ unidirectional::ATM und ermöglichen die typsichere Verbindung mit anderen Bausteinen, die denselben Adaptertyp unterstützen.

Funktionsweise

Der FB summiert die über die Adapter IN1 und IN2 anliegenden Zeitwerte auf und stellt das Ergebnis am Adapter OUT bereit. Ausgangspunkt der Summation ist der neutrale Wert TIME#0s, sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird. Der Baustein arbeitet generisch – die Berechnung wird bei jedem Ereignis an einem der Eingangsadapter neu ausgeführt.

Technische Besonderheiten

- **Generischer Aufbau** – Der FB verwendet zur Laufzeit einen generischen Klassennamen (GEN_ATM_ADD), der durch das Attribut eclipse4diac::core::GenericClassName festgelegt wird. Dieselbe Klasse deckt über den GenericClassName-Mechanismus die Aritäten ATM_ADD_2, ATM_ADD_3 und ATM_ADD_4 ab.
- **Adapterbasiert** – Statt einzelner Daten-Ein- und Ausgänge werden alle Signale über unidirektionale Adapter geführt.
- **Paketangabe** – Der FB ist im Paket adapter::iec61131::arithmetic organisiert.
- **Keine Zustandslogik** – Die Addition erfolgt zustandslos; es existiert kein interner Zustandsautomat.

Zustandsübersicht

Der Baustein besitzt keinen Zustandsautomaten. Die Berechnung wird rein ereignisgetrieben durchgeführt – bei einem Ereignis an einem der Eingangsadapter wird die Summe neu berechnet.

Anwendungsspezifische Szenarien

- **Zeitakkumulation** – Aufsummieren mehrerer Verzögerungs- oder Laufzeitwerte zu einer Gesamtdauer.
- **Steuerungstechnik** – Kombination zweier dynamisch ermittelter Zeitwerte, z. B. Basiszeit plus Zuschlag.
- **Generische Bibliotheksbausteine** – Einsatz als adapterbasiertes Pendant zur klassischen ADD-Funktion, spezialisiert auf TIME.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu AR_ADD_2 (Addition zweier REAL-Werte über Adapter) ist ATM_ADD_2 auf den Datentyp TIME spezialisiert. Anders als AR_MULTIME/ATM_AR_MULTIME, die einen Zeitwert mit einem numerischen Faktor multiplizieren, addiert ATM_ADD_2 zwei gleichartige Zeitwerte.

- ATM_ADD_2_UNGATED: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

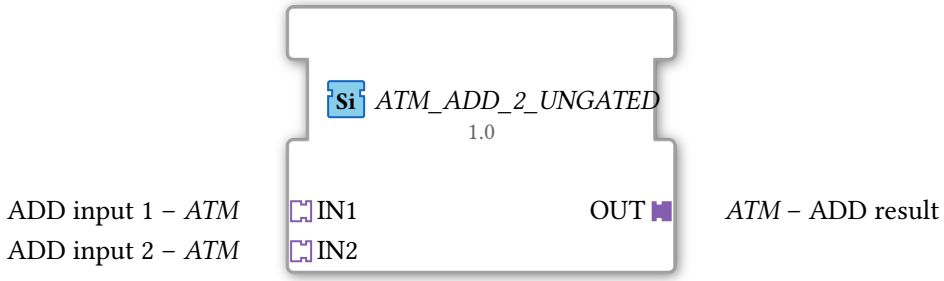
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

ATM_ADD_2 ist ein kompakter, generischer Funktionsbaustein für die Addition von zwei Zeitwerten mittels Adapter-Schnittstellen. Er eignet sich besonders für Anwendungen, in denen mehrere Zeitdauern zu einer Gesamtdauer zusammengeführt werden müssen.

ATM_ADD_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ATM_ADD_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **ATM_ADD_2_UNGATED** dient der Berechnung der arithmetischen Addition zweier Zeitwerte (Typ `TIME`). Er ist als generischer FB ausgelegt und realisiert die grundlegende Rechenoperation über Adapter-Schnittstellen. Der Baustein ist plattformunabhängig und folgt dem IEC-61499-Standard.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Bezeichnung	Richtung	Typ	Kommentar
`IN1`	Socket	`adapter::types::unidirectional::ATM`	ADD input 1
`IN2`	Socket	`adapter::types::unidirectional::ATM`	ADD input 2
`OUT`	Plug	`adapter::types::unidirectional::ATM`	ADD result

Die Adapter sind vom Typ `unidirectional::ATM` und ermöglichen die typsichere Verbindung mit anderen Bausteinen, die denselben Adaptertyp unterstützen.

Funktionsweise

Der FB summiert die über die Adapter `IN1` und `IN2` anliegenden Zeitwerte auf und stellt das Ergebnis am Adapter `OUT` bereit. Ausgangspunkt der Summation ist der neutrale Wert `TIME#0s`, sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird. Der Baustein arbeitet generisch – die Berechnung wird bei jedem Ereignis an einem der Eingangsadapter neu ausgeführt.

Technische Besonderheiten

- **Generischer Aufbau** – Der FB verwendet zur Laufzeit einen generischen Klassennamen (`GEN_ATM_ADD`), der durch das Attribut `eclipse4diac::core::GenericClassName` festgelegt wird. Dieselbe Klasse deckt über den GenericClassName-Mechanismus die Aritäten `ATM_ADD_2_UNGATED`, `ATM_ADD_3` und `ATM_ADD_4` ab.
- **Adapterbasiert** – Statt einzelner Daten-Ein- und Ausgänge werden alle Signale über unidirektionale Adapter geführt.
- **Paketangabe** – Der FB ist im Paket `adapter::iec61131::arithmetic` organisiert.
- **Keine Zustandslogik** – Die Addition erfolgt zustandslos; es existiert kein interner Zustandsautomat.

Zustandsübersicht

Der Baustein besitzt keinen Zustandsautomaten. Die Berechnung wird rein ereignisgetrieben durchgeführt – bei einem Ereignis an einem der Eingangsadapter wird die Summe neu berechnet.

Anwendungsspezifische Szenarien

- **Zeitakkumulation** – Aufsummieren mehrerer Verzögerungs- oder Laufzeitwerte zu einer Gesamtdauer.
- **Steuerungstechnik** – Kombination zweier dynamisch ermittelter Zeitwerte, z. B. Basiszeit plus Zuschlag.
- **Generische Bibliotheksbausteine** – Einsatz als adapterbasiertes Pendant zur klassischen `ADD`-Funktion, spezialisiert auf `TIME`.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu `AR_ADD_2` (Addition zweier `REAL`-Werte über Adapter) ist `ATM_ADD_2_UNGATED` auf den Datentyp `TIME` spezialisiert. Anders als `AR_MULTIME`/`ATM_AR_MULTIME`, die einen Zeitwert mit einem numerischen Faktor multiplizieren, addiert `ATM_ADD_2_UNGATED` zwei gleichartige Zeitwerte.

- `ATM_ADD_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

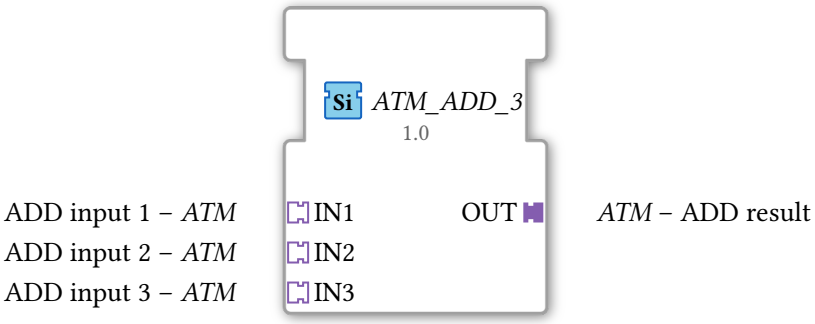
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

`ATM_ADD_2_UNGATED` ist ein kompakter, generischer Funktionsbaustein für die Addition von zwei Zeitwerten mittels Adapter-Schnittstellen. Er eignet sich besonders für Anwendungen, in denen mehrere Zeitdauern zu einer Gesamtdauer zusammengeführt werden müssen.

ATM_ADD_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **ATM_ADD_3** dient der Berechnung der arithmetischen Addition zweier Zeitwerte (Typ `TIME`). Er ist als generischer FB ausgelegt und realisiert die grundlegende Rechenoperation über Adapter-Schnittstellen. Der Baustein ist plattformunabhängig und folgt dem IEC-61499-Standard.

Schnittstellenstruktur

Ereignis-Eingänge
Keine

Ereignis-Ausgänge
Keine

Daten-Eingänge
Keine

Daten-Ausgänge
Keine

Adapter

Bezeichnung	Richtung	Typ	Kommentar
IN1	Socket	adapter::types::unidirectional::ATM	ADD input 1
IN2	Socket	adapter::types::unidirectional::ATM	ADD input 2
IN3	Socket	adapter::types::unidirectional::ATM	ADD input 3
OUT	Plug	adapter::types::unidirectional::ATM	ADD result

Die Adapter sind vom Typ unidirectional::ATM und ermöglichen die typsichere Verbindung mit anderen Bausteinen, die denselben Adaptertyp unterstützen.

Funktionsweise
Der FB summiert die über die Adapter IN1 und IN2 anliegenden Zeitwerte auf und stellt das Ergebnis am Adapter OUT bereit. Ausgangspunkt der Summation ist der neutrale Wert TIME#0s, sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird. Der Baustein arbeitet generisch – die Berechnung wird bei jedem Ereignis an einem der Eingangsadapter neu ausgeführt.

Technische Besonderheiten
- **Generischer Aufbau** – Der FB verwendet zur Laufzeit einen generischen Klassennamen (GEN_ATM_ADD), der durch das Attribut eclipse4diac::core::GenericClassName festgelegt wird. Dieselbe Klasse deckt über den GenericClassName-Mechanismus die Aritäten ATM_ADD_3, ATM_ADD_3 und ATM_ADD_4 ab.
- **Adapterbasiert** – Statt einzelner Daten-Ein- und Ausgänge werden alle Signale über unidirektionale Adapter geführt.
- **Paketangabe** – Der FB ist im Paket adapter::iec61131::arithmetic organisiert.
- **Keine Zustandslogik** – Die Addition erfolgt zustandslos; es existiert kein interner Zustandsautomat.

Zustandsübersicht
Der Baustein besitzt keinen Zustandsautomaten. Die Berechnung wird rein ereignisgetrieben durchgeführt – bei einem Ereignis an einem der Eingangsadapter wird die Summe neu berechnet.

Anwendungsspezifische Szenarien
- **Zeitakkumulation** – Aufsummieren mehrerer Verzögerungs- oder Laufzeitwerte zu einer Gesamtdauer.
- **Steuerungstechnik** – Kombination zweier dynamisch ermittelter Zeitwerte, z. B. Basiszeit plus Zuschlag.
- **Generische Bibliotheksbausteine** – Einsatz als adapterbasiertes Pendant zur klassischen ADD-Funktion, spezialisiert auf TIME.

Vergleich mit ähnlichen Bausteinen
Im Vergleich zu AR_ADD_2 (Addition zweier REAL-Werte über Adapter) ist ATM_ADD_3 auf den Datentyp TIME spezialisiert. Anders als AR_MULTIME/ATM_AR_MULTIME, die einen Zeitwert mit einem numerischen Faktor multiplizieren, addiert ATM_ADD_3 zwei gleichartige Zeitwerte.

- `ATM_ADD_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

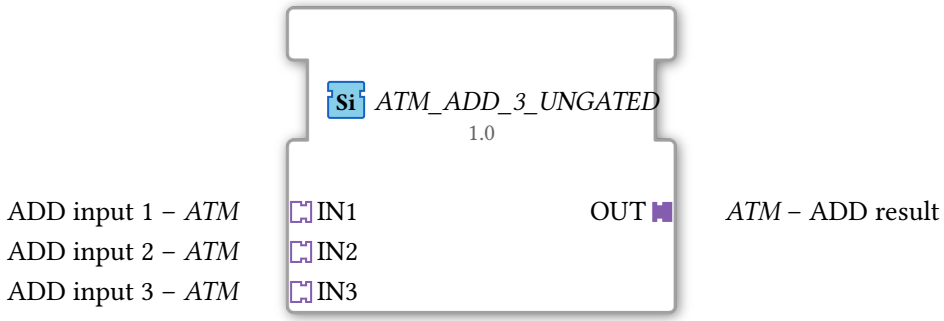
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

`ATM_ADD_3` ist ein kompakter, generischer Funktionsbaustein für die Addition von zwei Zeitwerten mittels Adapter-Schnittstellen. Er eignet sich besonders für Anwendungen, in denen mehrere Zeitdauern zu einer Gesamtdauer zusammengeführt werden müssen.

ATM_ADD_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ATM_ADD_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **ATM_ADD_3_UNGATED** dient der Berechnung der arithmetischen Addition zweier Zeitwerte (Typ `TIME`). Er ist als generischer FB ausgelegt und realisiert die grundlegende Rechenoperation über Adapter-Schnittstellen. Der Baustein ist plattformunabhängig und folgt dem IEC-61499-Standard.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Bezeichnung	Richtung	Typ	Kommentar
IN1	Socket	adapter::types::unidirectional::ATM	ADD input 1
IN2	Socket	adapter::types::unidirectional::ATM	ADD input 2
IN3	Socket	adapter::types::unidirectional::ATM	ADD input 3
OUT	Plug	adapter::types::unidirectional::ATM	ADD result

Die Adapter sind vom Typ `unidirectional::ATM` und ermöglichen die typsichere Verbindung mit anderen Bausteinen, die denselben Adaptertyp unterstützen.

Funktionsweise

Der FB summiert die über die Adapter `IN1` und `IN2` anliegenden Zeitwerte auf und stellt das Ergebnis am Adapter `OUT` bereit. Ausgangspunkt der Summation ist der neutrale Wert `TIME#0s`, sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird. Der Baustein arbeitet generisch – die Berechnung wird bei jedem Ereignis an einem der Eingangsadapter neu ausgeführt.

Technische Besonderheiten

- **Generischer Aufbau** – Der FB verwendet zur Laufzeit einen generischen Klassennamen (`GEN_ATM_ADD`), der durch das Attribut `eclipse4diac::core::GenericClassName` festgelegt wird. Dieselbe Klasse deckt über den GenericClassName-Mechanismus die Aritäten `ATM_ADD_3_UNGATED`, `ATM_ADD_3_UNGATED` und `ATM_ADD_4` ab.
- **Adapterbasiert** – Statt einzelner Daten-Ein- und Ausgänge werden alle Signale über unidirektionale Adapter geführt.
- **Paketangabe** – Der FB ist im Paket `adapter::iec61131::arithmetic` organisiert.
- **Keine Zustandslogik** – Die Addition erfolgt zustandslos; es existiert kein interner Zustandsautomat.

Zustandsübersicht

Der Baustein besitzt keinen Zustandsautomaten. Die Berechnung wird rein ereignisgetrieben durchgeführt – bei einem Ereignis an einem der Eingangsadapter wird die Summe neu berechnet.

Anwendungsspezifische Szenarien

- **Zeitakkumulation** – Aufsummieren mehrerer Verzögerungs- oder Laufzeitwerte zu einer Gesamtdauer.
- **Steuerungstechnik** – Kombination zweier dynamisch ermittelter Zeitwerte, z. B. Basiszeit plus Zuschlag.
- **Generische Bibliotheksbausteine** – Einsatz als adapterbasiertes Pendant zur klassischen `ADD`-Funktion, spezialisiert auf `TIME`.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu `AR_ADD_2` (Addition zweier `REAL`-Werte über Adapter) ist `ATM_ADD_3_UNGATED` auf den Datentyp `TIME` spezialisiert. Anders als `AR_MULTIME`/`ATM_AR_MULTIME`, die einen Zeitwert mit einem numerischen Faktor multiplizieren, addiert `ATM_ADD_3_UNGATED` zwei gleichartige Zeitwerte.

- `ATM_ADD_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

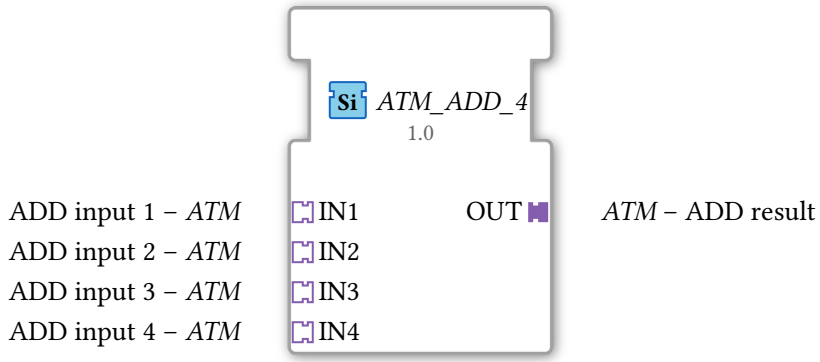
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

`ATM_ADD_3_UNGATED` ist ein kompakter, generischer Funktionsbaustein für die Addition von zwei Zeitwerten mittels Adapter-Schnittstellen. Er eignet sich besonders für Anwendungen, in denen mehrere Zeitdauern zu einer Gesamtdauer zusammengeführt werden müssen.

ATM_ADD_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **ATM_ADD_4** dient der Berechnung der arithmetischen Addition zweier Zeitwerte (Typ `TIME`). Er ist als generischer FB ausgelegt und realisiert die grundlegende Rechenoperation über Adapter-Schnittstellen. Der Baustein ist plattformunabhängig und folgt dem IEC-61499-Standard.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Bezeichnung	Richtung	Typ	Kommentar
`IN1`	Socket	`adapter::types::unidirectional::ATM`	ADD input 1
`IN2`	Socket	`adapter::types::unidirectional::ATM`	ADD input 2
`IN3`	Socket	`adapter::types::unidirectional::ATM`	ADD input 3
`IN4`	Socket	`adapter::types::unidirectional::ATM`	ADD input 4
`OUT`	Plug	`adapter::types::unidirectional::ATM`	ADD result

Die Adapter sind vom Typ `unidirectional::ATM` und ermöglichen die typsichere Verbindung mit anderen Bausteinen, die denselben Adaptertyp unterstützen.

Funktionsweise

Der FB summiert die über die Adapter `IN1` und `IN2` anliegenden Zeitwerte auf und stellt das Ergebnis am Adapter `OUT` bereit. Ausgangspunkt der Summation ist der neutrale Wert `TIME#0s`, sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird. Der Baustein arbeitet generisch – die Berechnung wird bei jedem Ereignis an einem der Eingangsadapter neu ausgeführt.

Technische Besonderheiten

- **Generischer Aufbau** – Der FB verwendet zur Laufzeit einen generischen Klassennamen (`GEN_ATM_ADD`), der durch das Attribut `eclipse4diac::core::GenericClassName` festgelegt wird. Die-

selbe Klasse deckt über den GenericClassName-Mechanismus die Aritäten ATM_ADD_4, ATM_ADD_3 und ATM_ADD_4 ab.
- **Adapterbasiert** – Statt einzelner Daten-Ein- und Ausgänge werden alle Signale über unidirektionale Adapter geführt.
- **Paketangabe** – Der FB ist im Paket adapter::iec61131::arithmetic organisiert.
- **Keine Zustandslogik** – Die Addition erfolgt zustandslos; es existiert kein interner Zustandsautomat.

Zustandsübersicht

Der Baustein besitzt keinen Zustandsautomaten. Die Berechnung wird rein ereignisgetrieben durchgeführt – bei einem Ereignis an einem der Eingangsadapter wird die Summe neu berechnet.

Anwendungsspezifische Szenarien

- **Zeitakkumulation** – Aufsummieren mehrerer Verzögerungs- oder Laufzeitwerte zu einer Gesamtdauer.
- **Steuerungstechnik** – Kombination zweier dynamisch ermittelter Zeitwerte, z. B. Basiszeit plus Zuschlag.
- **Generische Bibliotheksbausteine** – Einsatz als adapterbasiertes Pendant zur klassischen ADD-Funktion, spezialisiert auf TIME.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu AR_ADD_2 (Addition zweier REAL-Werte über Adapter) ist ATM_ADD_4 auf den Datentyp TIME spezialisiert. Anders als AR_MULTIME/ATM_AR_MULTIME, die einen Zeitwert mit einem numerischen Faktor multiplizieren, addiert ATM_ADD_4 zwei gleichartige Zeitwerte.

- ATM_ADD_4_UNGATED: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

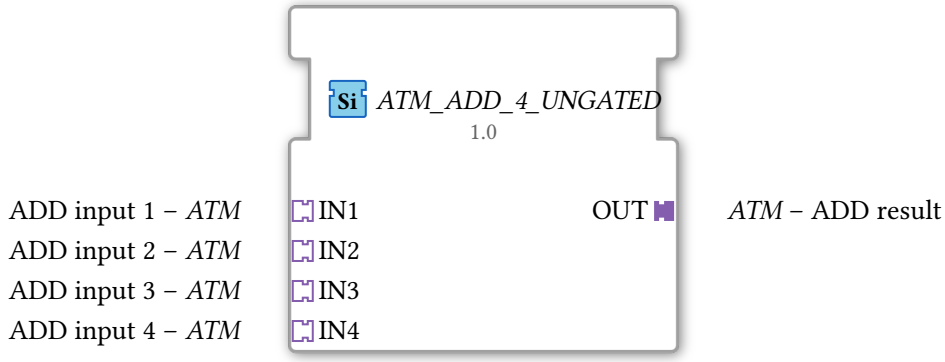
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

ATM_ADD_4 ist ein kompakter, generischer Funktionsbaustein für die Addition von zwei Zeitwerten mittels Adapter-Schnittstellen. Er eignet sich besonders für Anwendungen, in denen mehrere Zeitdauern zu einer Gesamtdauer zusammengeführt werden müssen.

ATM_ADD_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von ATM_ADD_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **ATM_ADD_4_UNGATED** dient der Berechnung der arithmetischen Addition zweier Zeitwerte (Typ `TIME`). Er ist als generischer FB ausgelegt und realisiert die grundlegende Rechenoperation über Adapter-Schnittstellen. Der Baustein ist plattformunabhängig und folgt dem IEC-61499-Standard.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Bezeichnung	Richtung	Typ	Kommentar
`IN1`	Socket	`adapter::types::unidirectional::ATM`	ADD input 1
`IN2`	Socket	`adapter::types::unidirectional::ATM`	ADD input 2
`IN3`	Socket	`adapter::types::unidirectional::ATM`	ADD input 3
`IN4`	Socket	`adapter::types::unidirectional::ATM`	ADD input 4
`OUT`	Plug	`adapter::types::unidirectional::ATM`	ADD result

Die Adapter sind vom Typ `unidirectional::ATM` und ermöglichen die typsichere Verbindung mit anderen Bausteinen, die denselben Adaptertyp unterstützen.

Funktionsweise

Der FB summiert die über die Adapter `IN1` und `IN2` anliegenden Zeitwerte auf und stellt das Ergebnis am Adapter `OUT` bereit. Ausgangspunkt der Summation ist der neutrale Wert `TIME#0s`, sodass bei nur einem tatsächlich angeschlossenen Eingang dessen Wert unverändert durchgereicht wird. Der Baustein arbeitet generisch – die Berechnung wird bei jedem Ereignis an einem der Eingangsadapter neu ausgeführt.

Technische Besonderheiten

- **Generischer Aufbau** – Der FB verwendet zur Laufzeit einen generischen Klassennamen (`GEN_ATM_ADD`), der durch das Attribut `eclipse4diac::core::GenericClassName` festgelegt wird.

Dieselbe Klasse deckt über den GenericClassName-Mechanismus die Aritäten `ATM_ADD_4_UNGATED`, `ATM_ADD_3` und `ATM_ADD_4_UNGATED` ab.
- **Adapterbasiert** – Statt einzelner Daten-Ein- und Ausgänge werden alle Signale über unidirektionale Adapter geführt.
- **Paketangabe** – Der FB ist im Paket `adapter::iec61131::arithmetic` organisiert.
- **Keine Zustandslogik** – Die Addition erfolgt zustandslos; es existiert kein interner Zustandsautomat.

Zustandsübersicht

Der Baustein besitzt keinen Zustandsautomaten. Die Berechnung wird rein ereignisgetrieben durchgeführt – bei einem Ereignis an einem der Eingangsadapter wird die Summe neu berechnet.

Anwendungsspezifische Szenarien

- **Zeitakkumulation** – Aufsummieren mehrerer Verzögerungs- oder Laufzeitwerte zu einer Gesamtdauer.
- **Steuerungstechnik** – Kombination zweier dynamisch ermittelter Zeitwerte, z. B. Basiszeit plus Zuschlag.
- **Generische Bibliotheksbausteine** – Einsatz als adapterbasiertes Pendant zur klassischen `ADD`-Funktion, spezialisiert auf `TIME`.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu `AR_ADD_2` (Addition zweier `REAL`-Werte über Adapter) ist `ATM_ADD_4_UNGATED` auf den Datentyp `TIME` spezialisiert. Anders als `AR_MULTIME`/`ATM_AR_MULTIME`, die einen Zeitwert mit einem numerischen Faktor multiplizieren, addiert `ATM_ADD_4_UNGATED` zwei gleichartige Zeitwerte.

- `ATM_ADD_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

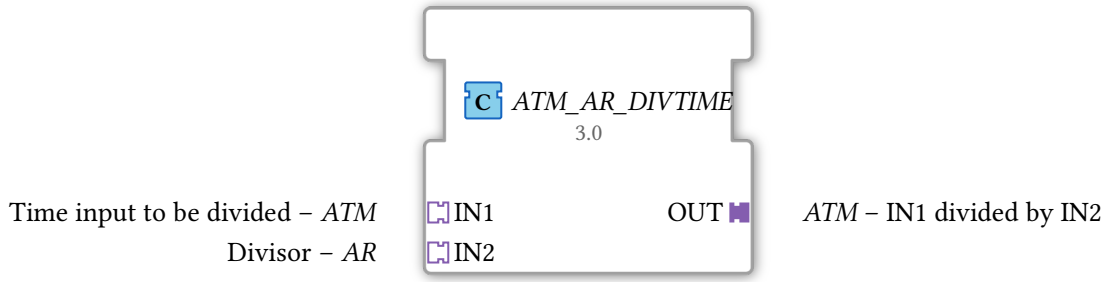
Änderungserkennung

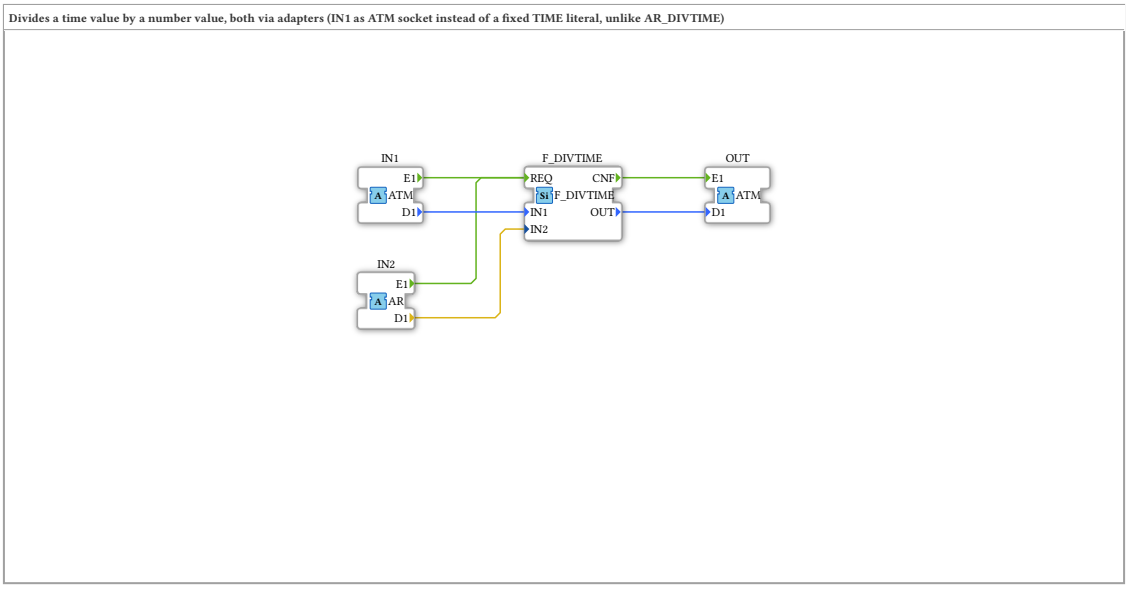
Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

`ATM_ADD_4_UNGATED` ist ein kompakter, generischer Funktionsbaustein für die Addition von zwei Zeitwerten mittels Adapter-Schnittstellen. Er eignet sich besonders für Anwendungen, in denen mehrere Zeitdauern zu einer Gesamtdauer zusammengeführt werden müssen.

ATM_AR_DIVTIME





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ATM_AR_DIVTIME** dividiert einen Zeitwert (Typ TIME) durch einen numerischen Wert – die vollständig adapterbasierte Variante von AR_DIVTIME. Anders als bei AR_DIVTIME wird auch der zu dividierende Zeitwert (IN1) über einen Adapter (Typ `ATM`) statt über einen festen Dateneingang eingespeist. Der Baustein ist gemäß IEC 61131-3 als standard arithmetic function klassifiziert.

Schnittstellenstruktur

Ereignis-Eingänge

Keine – der Baustein besitzt keine klassischen Ereigniseingänge, die Auslösung erfolgt ausschließlich über die Adapter-Sockets.

Ereignis-Ausgänge

Keine expliziten Ereignisausgänge in der Interface-Liste. Das Ergebnis wird über den Adapter-Ausgang **OUT** bereitgestellt, welcher ein Ereignis (E1) sendet, sobald die Berechnung abgeschlossen ist.

Daten-Eingänge

Keine direkten Daten-Eingänge.

Daten-Ausgänge

Keine direkten Daten-Ausgänge. Das Divisionsergebnis wird über den Adapter-Ausgang **OUT** als Datenwert (D1) ausgegeben.

Adapter

- **IN1** (Adapter, Typ `adapter::types::unidirectional::ATM`): Zu dividierender Zeitwert (Dividend).
- **IN2** (Adapter, Typ `adapter::types::unidirectional::AR`): Divisor.
- **OUT** (Adapter, Typ `adapter::types::unidirectional::ATM`): Ausgang für das Divisionsergebnis (TIME).

Funktionsweise

Der Funktionsblock kapselt die Funktionalität des internen Bausteins **F_DIVTIME** (Typ `iec61131::arithmetic::F_DIVTIME`). Ein Ereignis an **IN1.E1** oder **IN2.E1** löst die Division aus: Der interne FB dividiert **IN1.D1** (TIME) durch **IN2.D1** (numerischer Wert) und gibt das Ergebnis auf **OUT.D1** aus. Anschließend wird ein Ereignis an **OUT.E1** gesendet.

Technische Besonderheiten

- **Vollständig adapterbasiert:** Im Gegensatz zu `AR_DIVTIME`, das IN1 als klassischen Dateneingang erwartet, sind bei `ATM_AR_DIVTIME` beide Operanden als Adapter-Sockets ausgeführt (IN1 vom Typ `ATM`, IN2 vom Typ `AR`).
- **Standardkonformität:** Der Baustein erfüllt die Anforderungen der IEC 61131-3 (Klassifikation: standard arithmetic function).
- **Interne Kapselung:** Die eigentliche Berechnung erfolgt durch den wiederverwendbaren Baustein F_DIVTIME.

Zustandsübersicht

Der Baustein besitzt keine explizite Zustandsmaschine. Die ereignisgesteuerte Abarbeitung erfolgt deterministisch: Warten auf ein Ereignis an IN1.E1 oder IN2.E1, Ausführen der Division, Senden des Ergebnisses.

Anwendungsszenarien

- Aufteilen von Zeitwerten, bei denen sowohl der Zeitwert als auch der Divisor bereits als Adapterverbindungen aus vorgeschalteten Bausteinen vorliegen.
- Modulare Netzwerke, in denen konsequent auf Adapter statt klassischer Datenverdrahtung gesetzt wird.

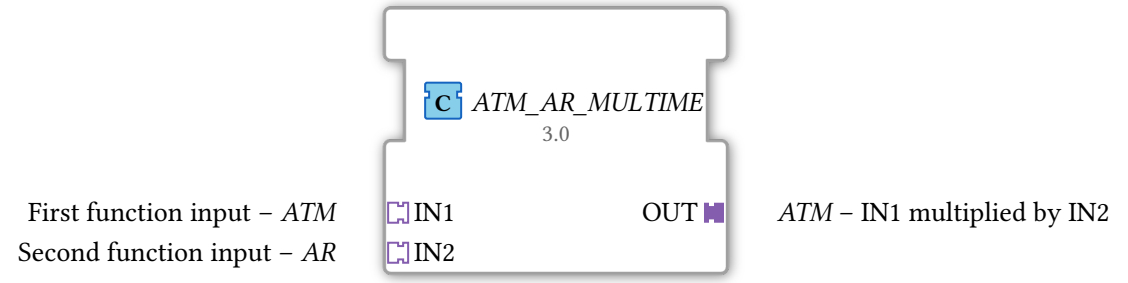
Vergleich mit ähnlichen Bausteinen

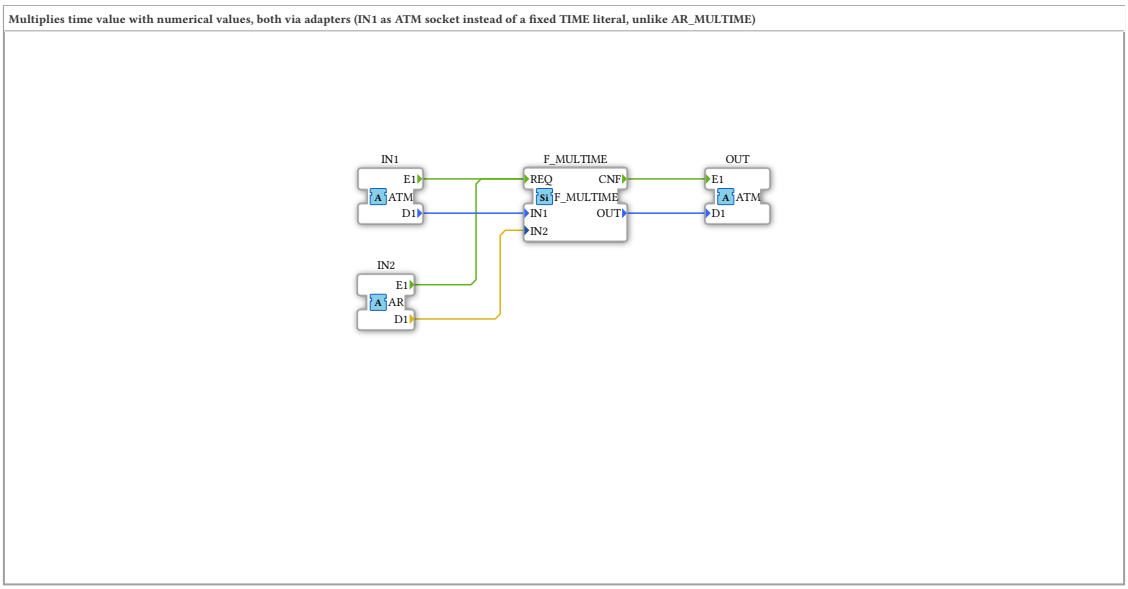
- **AR_DIVTIME:** Die gemischte Variante mit IN1 als klassischem Dateneingang statt Adapter-Socket.
- **ATM_AR_MULTIME:** Multipliziert statt zu dividieren, ansonsten strukturell identisch.

Fazit

`ATM_AR_DIVTIME` bietet eine konsequent adapterbasierte Division von Zeitwerten durch numerische Faktoren und eignet sich besonders für Netzwerke, die durchgängig auf Adapterverbindungen aufbauen.

ATM_AR_MULTIME





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ATM_AR_MULTIME** multipliziert einen Zeitwert (Typ TIME) mit einem numerischen Wert – die vollständig adapterbasierte Variante von AR_MULTIME. Anders als bei AR_MULTIME wird auch der Zeitwert (IN1) über einen Adapter (Typ `ATM`) statt über einen festen Dateneingang eingespeist. Der Baustein ist gemäß IEC 61131-3 als standard arithmetic function klassifiziert.

Schnittstellenstruktur

Ereignis-Eingänge

Keine – der Baustein besitzt keine klassischen Ereigniseingänge, die Auslösung erfolgt ausschließlich über die Adapter-Sockets.

Ereignis-Ausgänge

Keine expliziten Ereignisausgänge in der Interface-Liste. Das Ergebnis wird über den Adapter-Ausgang **OUT** bereitgestellt, welcher ein Ereignis (E1) sendet, sobald die Berechnung abgeschlossen ist.

Daten-Eingänge

Keine direkten Daten-Eingänge.

Daten-Ausgänge

Keine direkten Daten-Ausgänge. Das Multiplikationsergebnis wird über den Adapter-Ausgang **OUT** als Datenwert (D1) ausgegeben.

Adapter

- **IN1** (Adapter, Typ `adapter::types::unidirectional::ATM`): Erster Funktionsinput – der zu multiplizierende Zeitwert.
- **IN2** (Adapter, Typ `adapter::types::unidirectional::AR`): Zweiter Funktionsinput – der numerische Multiplikator.
- **OUT** (Adapter, Typ `adapter::types::unidirectional::ATM`): Ausgang für das multiplizierte Ergebnis (TIME).

Funktionsweise

Der Funktionsblock kapselt die Funktionalität des internen Bausteins **F_MULTIME** (Typ `iec61131::arithmetic::F_MULTIME`). Ein Ereignis an **IN1.E1** oder **IN2.E1** löst die Multiplikation aus: Der interne FB multipliziert **IN1.D1** (TIME) mit **IN2.D1** (numerischer Wert) und gibt das Ergebnis auf **OUT.D1** aus. Anschließend wird ein Ereignis an **OUT.E1** gesendet.

Technische Besonderheiten

- **Vollständig adapterbasiert:** Im Gegensatz zu `AR_MULTIME`, das IN1 als klassischen Dateneingang erwartet, sind bei `ATM_AR_MULTIME` beide Operanden als Adapter-Sockets ausgeführt (IN1 vom Typ `ATM`, IN2 vom Typ `AR`).
- **Standardkonformität:** Der Baustein erfüllt die Anforderungen der IEC 61131-3 (Klassifikation: standard arithmetic function).
- **Interne Kapselung:** Die eigentliche Berechnung erfolgt durch den wiederverwendbaren Baustein F_MULTIME.

Zustandsübersicht

Der Baustein besitzt keine explizite Zustandsmaschine. Die ereignisgesteuerte Abarbeitung erfolgt deterministisch: Warten auf ein Ereignis an IN1.E1 oder IN2.E1, Ausführen der Multiplikation, Senden des Ergebnisses.

Anwendungsszenarien

- Skalierung von Zeitwerten, bei denen sowohl der Zeitwert als auch der Faktor bereits als Adapterverbindungen aus vorgeschalteten Bausteinen vorliegen.
- Modulare Netzwerke, in denen konsequent auf Adapter statt klassischer Datenverdrahtung gesetzt wird.

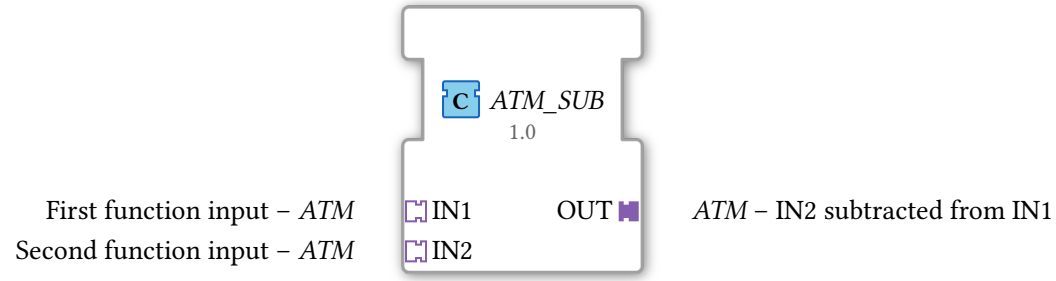
Vergleich mit ähnlichen Bausteinen

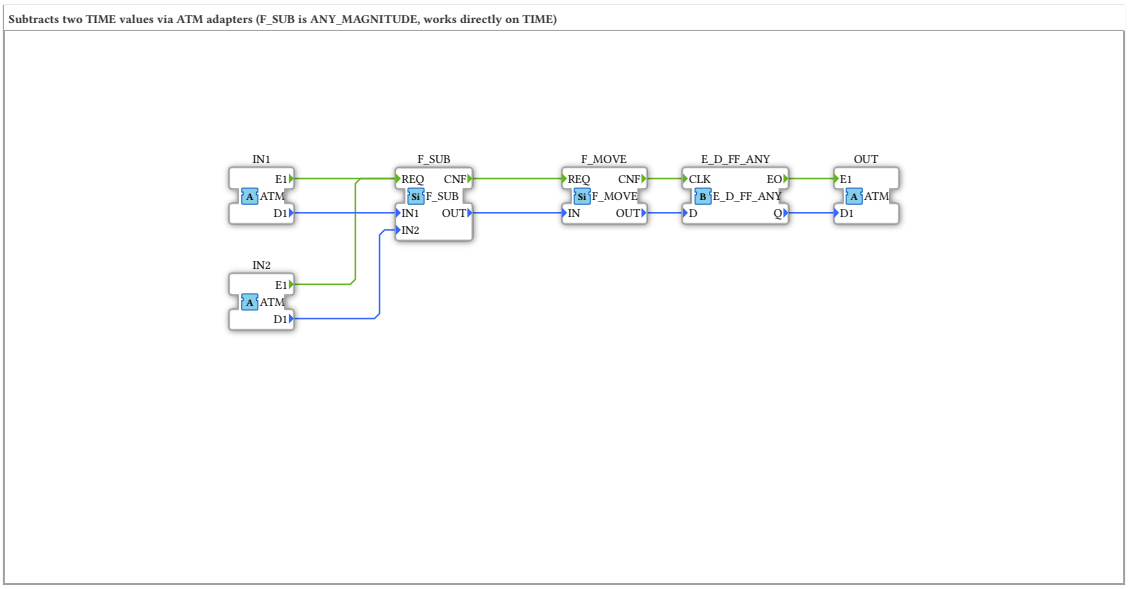
- **AR_MULTIME:** Die gemischte Variante mit IN1 als klassischem Dateneingang statt Adapter-Socket.
- **ATM_AR_DIVTIME:** Dividiert statt zu multiplizieren, ansonsten strukturell identisch.

Fazit

`ATM_AR_MULTIME` bietet eine konsequent adapterbasierte Multiplikation von Zeitwerten mit numerischen Faktoren und eignet sich besonders für Netzwerke, die durchgängig auf Adapterverbindungen aufbauen.

ATM_SUB





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ATM_SUB** subtrahiert zwei Zeitwerte (Typ TIME) über vollständig adapterbasierte Schnittstellen: `IN2` wird von `IN1` abgezogen, das Ergebnis wird nur bei tatsächlicher Wertänderung auf den Ausgangsadapter geschrieben. Im Gegensatz zu den generischen `ATM_ADD_*`-Bausteinen ist ATM_SUB als handverdrahtetes Composite-FB realisiert, das die Änderungserkennung explizit über eigene Bausteine abbildet.

Schnittstellenstruktur

Ereignis-Eingänge

Keine – die Auslösung erfolgt ausschließlich über die Adapter-Sockets `IN1`/`IN2`.

Ereignis-Ausgänge

Keine expliziten Ereignisausgänge in der Interface-Liste. Das Ergebnis wird über den Adapter-Ausgang **OUT** bereitgestellt, dessen Ereignis (E1) nur bei tatsächlicher Wertänderung gesendet wird.

Daten-Eingänge

Keine direkten Daten-Eingänge.

Daten-Ausgänge

Keine direkten Daten-Ausgänge. Das Subtraktionsergebnis wird über den Adapter-Ausgang **OUT** als Datenwert (D1) ausgegeben.

Adapter

- **IN1** (Adapter, Typ `adapter::types::unidirectional::ATM`): Minuend.
- **IN2** (Adapter, Typ `adapter::types::unidirectional::ATM`): Subtrahend.
- **OUT** (Adapter, Typ `adapter::types::unidirectional::ATM`): Ergebnis von `IN1 - IN2`.

Funktionsweise

Ein Ereignis an **IN1.E1** oder **IN2.E1** löst intern den Baustein **F_SUB** (Typ `iec61131::arithmetic::F_SUB`) aus, der `IN1.D1 - IN2.D1` berechnet. Das Ergebnis läuft anschließend über **F_MOVE** (Typ `iec61131::selection::F_MOVE`, mit `DataType="TIME"` fest verdrahtet) auf einen **E_D_FF_ANY**-Baustein (Typ `iec61499::events::E_D_FF_ANY`). Dieser vergleicht den neuen Wert mit dem zuletzt gespeicherten und lässt das Ereignis an **OUT.E1** nur dann durch, wenn sich der Wert tatsächlich geändert hat – der neue Wert wird in jedem Fall an **OUT.D1** übernommen.

Technische Besonderheiten

- **Explizite Änderungserkennung als Composite-Struktur:** Anders als bei den generischen `GEN_ATM_*`-Bausteinen, die die Änderungserkennung in C++ implementieren, bildet ATM_SUB sie über die Kombination `F_SUB → F_MOVE → E_D_FF_ANY` innerhalb des FBNetworks ab.
- **F_MOVE als Typanker:** Der `F_MOVE`-Baustein trägt das Attribut `DataType="TIME"`, wodurch der generische `ANY`-Datenpfad auf `TIME` festgelegt wird, bevor er dem `E_D_FF_ANY` zugeführt wird.
- **Vollständig adapterbasiert:** Beide Operanden und das Ergebnis sind als `ATM`-Adapter ausgeführt, es gibt keine klassischen Daten-Ein-/Ausgänge.

Zustandsübersicht

Der Baustein hält implizit den zuletzt ausgegebenen Wert im internen `E_D_FF_ANY`-Speicher:

- **Warten** auf ein Ereignis an IN1.E1 oder IN2.E1
- **Berechnen** von IN1 – IN2 über F_SUB
- **Vergleichen** mit dem gespeicherten Wert in E_D_FF_ANY
- **Ausgeben**: OUT.D1 wird immer aktualisiert, OUT.E1 nur bei tatsächlicher Änderung gesendet

Anwendungsszenarien

- Berechnung von Restlaufzeiten oder Differenzen zwischen zwei Zeitwerten, z. B. Soll- minus Ist-Zeit.
- Reduktion nachgeschalteter Event-Last, wenn sich das Ergebnis über mehrere Zyklen hinweg nicht ändert.

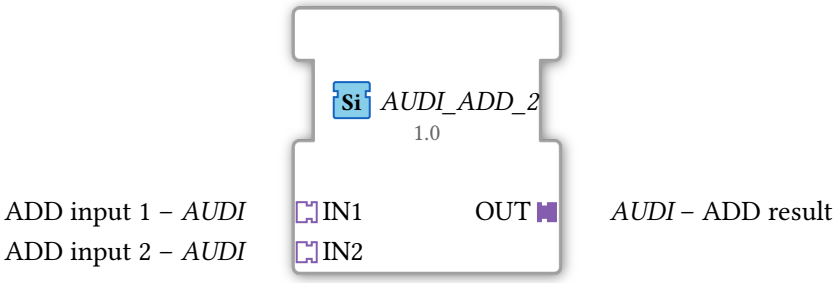
Vergleich mit ähnlichen Bausteinen

- **ATM_ADD_2/3/4:** Addieren statt zu subtrahieren; als generische `GEN_ATM_ADD`-Bausteine implementiert (Änderungserkennung in C++, nicht als sichtbares FBNetwork).
- **F_SUB:** Der direkte, adapterfreie Subtraktionsbaustein ohne eigene Änderungserkennung, den ATM_SUB intern verwendet.

Fazit

`ATM_SUB` liefert eine vollständig adapterbasierte Subtraktion zweier Zeitwerte mit expliziter, im FBNetwork sichtbarer Änderungserkennung. Er eignet sich besonders dort, wo Differenzbildung zwischen Zeitwerten benötigt wird und unnötige Folge-Events bei unverändertem Ergebnis vermieden werden sollen.

AUDI_ADD_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUDI_ADD_2** führt eine arithmetische Addition zweier Eingangswerte durch und stellt das Ergebnis am Ausgang bereit. Der Baustein ist generisch ausgelegt und arbeitet über eine einheitliche Adapter-Schnittstelle vom Typ `AUDI`. Dadurch kann er für unterschiedliche Datentypen (z. B. Ganzzahlen, Gleitkommazahlen) eingesetzt werden, ohne dass der FB selbst angepasst werden muss.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Typ	Richtung	Name	Kommentar
`adapter::types::unidirectional::AUDI`	Socket (Eingang)	**IN1**	ADD input 1
`adapter::types::unidirectional::AUDI`	Socket (Eingang)	**IN2**	ADD input 2
`adapter::types::unidirectional::AUDI`	Plug (Ausgang)	**OUT**	ADD result

Die drei Adapter sind alle vom gleichen Typ `AUDI`. Dieser Adaptertyp kapselt den eigentlichen Datenwert (z. B. einen numerischen Wert) und ermöglicht die unidirektionale Übertragung vom Socket zum Plug. Die Addition erfolgt zwischen den Werten von `IN1` und `IN2`; das Ergebnis wird an `OUT` bereitgestellt.

Funktionsweise

Der FB **AUDI_ADD_2** besitzt keine klassischen Daten-Eingänge oder Ereignisse. Stattdessen kommuniziert er ausschließlich über Adapter. Sobald beide Eingangsadapter (`IN1` und `IN2`) gültige Werte enthalten, wird intern die Summe `IN1 + IN2` berechnet und der resultierende Wert über den Ausgangsadapter `OUT` ausgegeben.

Der Baustein ist generisch: Der tatsächlich verwendete Datentyp wird durch die Konfiguration des Adapters und die generischen Attribute (`GenericClassName`, `TypeHash`) bestimmt. Dies erlaubt den Einsatz mit verschiedenen numerischen Typen (z. B. `INT`, `REAL`, `LREAL`), ohne dass der FB selbst mehrfach instanziiert werden muss.

Technische Besonderheiten

- **Reine Adapter-Schnittstelle**: Keine Ereignisse oder direkten Datenein-/ausgänge – die gesamte Kommunikation erfolgt über Adapter.

- **Generische Ausführung**: Durch die Attribute `eclipse4diac::core::GenericClassName` und `eclipse4diac::core::TypeHash` kann der FB zur Laufzeit auf unterschiedliche Datentypen spezialisiert werden.
- **Unidirektionale Adapter**: Die verwendeten Adapter vom Typ `AUDI` sind unidirektional, d. h. sie übertragen Werte nur in eine Richtung (Socket → Plug).
- **Keine Zustandsautomatik**: Der FB enthält kein ECC (Execution Control Chart) und arbeitet ereignislos; die Berechnung erfolgt sofort bei Vorliegen der Eingangswerte.

Zustandsübersicht

Der **AUDI_ADD_2** besitzt keine Zustände (kein ECC). Die Addition wird unverzögert bei gültigen Eingangswerten durchgeführt.

Anwendungsszenarien

- **Arithmetische Verarbeitung in adapterbasierten Systemen**: Wenn eine Architektur konsequent auf Adapter setzt, bietet dieser FB eine saubere Möglichkeit, zwei Zahlenwerte zu addieren.
- **Generische Berechnungen**: Dank der generischen Auslegung kann der FB in Algorithmen eingesetzt werden, die mit verschiedenen Datentypen arbeiten (z. B. Integer- und Fließkomma-Mischumgebungen).
- **Datenflusssysteme**: In rein datenflussorientierten Applikationen, bei denen keine Ereignisse benötigt werden, fügt sich der FB nahtlos ein.

Vergleich mit ähnlichen Bausteinen

- **Klassische ADD-FBs (z. B. `ADD` aus IEC 61131-3)**: Diese besitzen meist konkrete Dateneingänge und -ausgänge sowie Ereignissteuerung. Der `AUDI_ADD_2` ersetzt dies durch eine Adapter-Schnittstelle, was eine höhere Abstraktion und Kapselung ermöglicht.
- **Generische Addierer (z. B. `GEN_ADD`)**: Solche Bausteine sind ebenfalls generisch, verwenden aber häufig direkte Datenports. Der `AUDI_ADD_2` ist speziell für adapterbasierte Schnittstellen optimiert.
- `AUDI_ADD_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

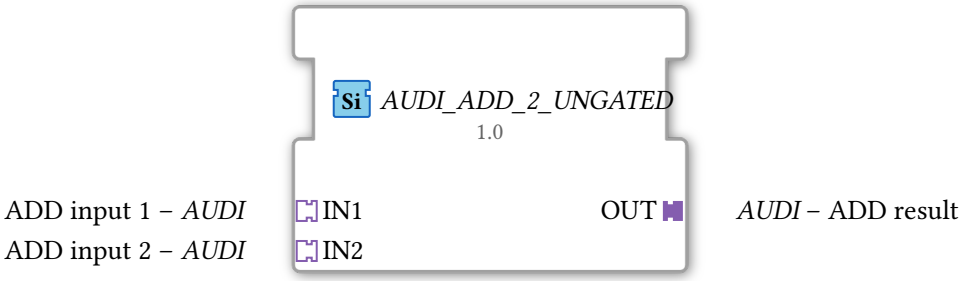
Der **AUDI_ADD_2** ist ein spezialisierter, generischer Additions-FB für adapterbasierte 4diac-IDE-Umgebungen. Durch den Verzicht auf klassische Ports und die Nutzung des `AUDI`-Adaptertyps eignet er sich besonders für datenflussorientierte und stark modularisierte Steuerungsanwendungen. Seine generische Ausführung erhöht die Wiederverwendbarkeit über verschiedene Datentypen hinweg.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUDI_ADD_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUDI_ADD_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUDI_ADD_2_UNGATED** führt eine arithmetische Addition zweier Eingangswerte durch und stellt das Ergebnis am Ausgang bereit. Der Baustein ist generisch ausgelegt und arbeitet über eine einheitliche Adapter-Schnittstelle vom Typ `AUDI`. Dadurch kann er für unterschiedliche Datentypen (z. B. Ganzzahlen, Gleitkommazahlen) eingesetzt werden, ohne dass der FB selbst angepasst werden muss.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Typ	Richtung	Name	Kommentar
`adapter::types::unidirectional::AUDI`	Socket (Eingang)	**IN1**	ADD input 1
`adapter::types::unidirectional::AUDI`	Socket (Eingang)	**IN2**	ADD input 2
`adapter::types::unidirectional::AUDI`	Plug (Ausgang)	**OUT**	ADD result

Die drei Adapter sind alle vom gleichen Typ `AUDI`. Dieser Adaptertyp kapselt den eigentlichen Datenwert (z. B. einen numerischen Wert) und ermöglicht die unidirektionale Übertragung vom Socket zum Plug. Die Addition erfolgt zwischen den Werten von `IN1` und `IN2`; das Ergebnis wird an `OUT` bereitgestellt.

Funktionsweise

Der FB **AUDI_ADD_2_UNGATED** besitzt keine klassischen Daten-Eingänge oder Ereignisse. Stattdessen kommuniziert er ausschließlich über Adapter. Sobald beide Eingangsadapter (`IN1` und `IN2`) gültige Werte enthalten, wird intern die Summe `IN1 + IN2` berechnet und der resultierende Wert über den Ausgangsadapter `OUT` ausgegeben.

Der Baustein ist generisch: Der tatsächlich verwendete Datentyp wird durch die Konfiguration des Adapters und die generischen Attribute (`GenericClassName`, `TypeHash`) bestimmt. Dies erlaubt den Einsatz mit verschiedenen numerischen Typen (z. B. `INT`, `REAL`, `LREAL`), ohne dass der FB selbst mehrfach instanziiert werden muss.

Technische Besonderheiten

- **Reine Adapter-Schnittstelle**: Keine Ereignisse oder direkten Datenein-/ausgänge – die gesamte Kommunikation erfolgt über Adapter.

- **Generische Ausführung**: Durch die Attribute `eclipse4diac::core::GenericClassName` und `eclipse4diac::core::TypeHash` kann der FB zur Laufzeit auf unterschiedliche Datentypen spezialisiert werden.
- **Unidirektionale Adapter**: Die verwendeten Adapter vom Typ `AUDI` sind unidirektional, d. h. sie übertragen Werte nur in eine Richtung (Socket → Plug).
- **Keine Zustandsautomatik**: Der FB enthält kein ECC (Execution Control Chart) und arbeitet ereignislos; die Berechnung erfolgt sofort bei Vorliegen der Eingangswerte.

Zustandsübersicht

Der **AUDI_ADD_2_UNGATED** besitzt keine Zustände (kein ECC). Die Addition wird unverzögert bei gültigen Eingangswerten durchgeführt.

Anwendungsszenarien

- **Arithmetische Verarbeitung in adapterbasierten Systemen**: Wenn eine Architektur konsequent auf Adapter setzt, bietet dieser FB eine saubere Möglichkeit, zwei Zahlenwerte zu addieren.
- **Generische Berechnungen**: Dank der generischen Auslegung kann der FB in Algorithmen eingesetzt werden, die mit verschiedenen Datentypen arbeiten (z. B. Integer- und Fließkomma-Mischumgebungen).
- **Datenflusssysteme**: In rein datenflussorientierten Applikationen, bei denen keine Ereignisse benötigt werden, fügt sich der FB nahtlos ein.

Vergleich mit ähnlichen Bausteinen

- **Klassische ADD-FBs (z. B. `ADD` aus IEC 61131-3)**: Diese besitzen meist konkrete Dateneingänge und -ausgänge sowie Ereignissteuerung. Der `AUDI_ADD_2_UNGATED` ersetzt dies durch eine Adapter-Schnittstelle, was eine höhere Abstraktion und Kapselung ermöglicht.
- **Generische Addierer (z. B. `GEN_ADD`)**: Solche Bausteine sind ebenfalls generisch, verwenden aber häufig direkte Datenports. Der `AUDI_ADD_2_UNGATED` ist speziell für adapterbasierte Schnittstellen optimiert.
- **`AUDI_ADD_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

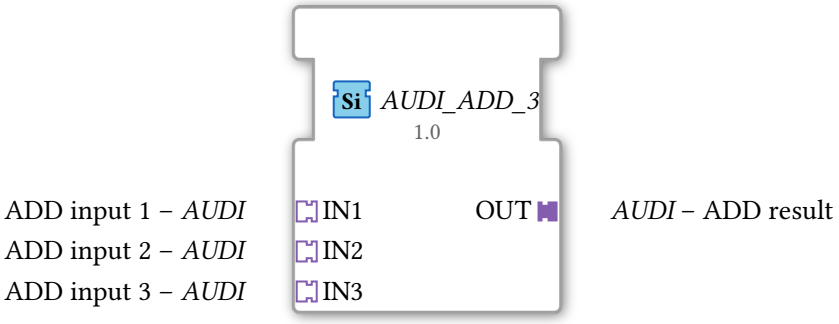
Fazit

Der **AUDI_ADD_2_UNGATED** ist ein spezialisierter, generischer Additions-FB für adapterbasierte 4diac-IDE-Umgebungen. Durch den Verzicht auf klassische Ports und die Nutzung des `AUDI`-Adaptertyps eignet er sich besonders für datenflussorientierte und stark modularisierte Steuerungsanwendungen. Seine generische Ausführung erhöht die Wiederverwendbarkeit über verschiedene Datentypen hinweg.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUDI_ADD_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AUDI_ADD_3` führt eine arithmetische Addition von drei Eingangswerten durch. Es handelt sich um einen generischen Baustein, der über Adapter-Schnittstellen die Werte aufnimmt und das Ergebnis über einen weiteren Adapter ausgibt. Der FB ist als Teil der 4diac-IDE für IEC 61499-2 konzipiert.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden. Die Eingangswerte werden über die Adapter-Schnittstellen bereitgestellt.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden. Das Ergebnis wird über den Adapter-Ausgang bereitgestellt.

Adapter

Name	Typ	Richtung	Kommentar
IN1	`adapter::types::unidirectional::AUDI`	Socket (Eingang)	ADD input 1
IN2	`adapter::types::unidirectional::AUDI`	Socket (Eingang)	ADD input 2
IN3	`adapter::types::unidirectional::AUDI`	Socket (Eingang)	ADD input 3
OUT	`adapter::types::unidirectional::AUDI`	Plug (Ausgang)	ADD result

Funktionsweise

Der FB addiert die drei Werte, die über die Adapter `IN1`, `IN2` und `IN3` anliegen. Das Ergebnis wird kontinuierlich über den Adapter `OUT` ausgegeben. Da es sich um einen generischen Baustein handelt, wird der konkrete Datentyp der Addition durch die Attribute `eclipse4diac::core::GenericClassName` und `eclipse4diac::core::TypeHash` zur Laufzeit festgelegt.

Technische Besonderheiten

- Der FB ist zustandslos (rein kombinatorisch) – kein ECC erforderlich.
- Er nutzt das Eclipse 4diac Generic-Type-System, um eine flexible Wiederverwendbarkeit für verschiedene arithmetische Datentypen (z. B. INT, REAL, LREAL) zu ermöglichen.
- Die Verbindung erfolgt ausschließlich über Adapter des Typs `adapter::types::unidirectional::AUDI`, die eine standardisierte unidirektionale Schnittstelle für arithmetische Operationen bereitstellen.

Zustandsübersicht

Der FB besitzt keine Zustandsmaschine (keine ECC-Elemente in der XML). Das Verhalten ist rein datengetrieben: Bei jeder Änderung eines Eingangswerts wird das Ausgangsergebnis neu berechnet.

Anwendungsszenarien

- **Summierung von Messwerten:** Drei analoge Sensorwerte (z. B. Temperaturen) aufsummieren.
- **Mischungsberechnung:** Addieren von drei Mengen oder Volumenströmen in der Verfahrenstechnik.
- **Generische Addition:** Verwendung in Bibliotheken, bei denen der Datentyp erst zur Projektierungszeit festgelegt wird.

Vergleich mit ähnlichen Bausteinen

- **ADD_2:** Führt eine Addition von zwei Werten durch – `AUDI_ADD_3` erweitert dies auf drei Eingänge.
- **ADD_4:** Addiert vier Werte – `AUDI_ADD_3` spezifiziert drei Eingänge.

- **MUL_3:** Multipliziert drei Werte – im Gegensatz zur Addition. Alle genannten Bausteine verwenden das gleiche Adapter-Konzept und sind generisch.
- `AUDI_ADD_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

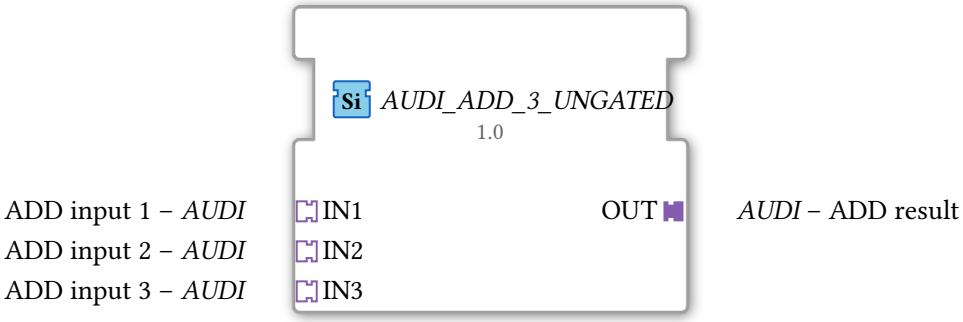
Der `AUDI_ADD_3` ist ein flexibler, generischer Funktionsblock für die Addition von drei Werten. Seine reine Adapter-Schnittstelle und die Zustandslosigkeit machen ihn einfach in der Handhabung und ideal für modulare Automatisierungslösungen auf Basis von IEC 61499.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUDI_ADD_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUDI_ADD_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AUDI_ADD_3_UNGATED` führt eine arithmetische Addition von drei Eingangswerten durch. Es handelt sich um einen generischen Baustein, der über Adapter-Schnittstellen die Werte aufnimmt und das Ergebnis über einen weiteren Adapter ausgibt. Der FB ist als Teil der 4diac-IDE für IEC 61499-2 konzipiert.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden. Die Eingangswerte werden über die Adapter-Schnittstellen bereitgestellt.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden. Das Ergebnis wird über den Adapter-Ausgang bereitgestellt.

Adapter

Name	Typ	Richtung	Kommentar
IN1	`adapter::types::unidirectional::AUDI`	Socket (Eingang)	ADD input 1
IN2	`adapter::types::unidirectional::AUDI`	Socket (Eingang)	ADD input 2
IN3	`adapter::types::unidirectional::AUDI`	Socket (Eingang)	ADD input 3
OUT	`adapter::types::unidirectional::AUDI`	Plug (Ausgang)	ADD result

Funktionsweise

Der FB addiert die drei Werte, die über die Adapter `IN1`, `IN2` und `IN3` anliegen. Das Ergebnis wird kontinuierlich über den Adapter `OUT` ausgegeben. Da es sich um einen generischen Baustein handelt, wird der konkrete Datentyp der Addition durch die Attribute `eclipse4diac::core::GenericClassName` und `eclipse4diac::core::TypeHash` zur Laufzeit festgelegt.

Technische Besonderheiten

- Der FB ist zustandslos (rein kombinatorisch) – kein ECC erforderlich.
- Er nutzt das Eclipse 4diac Generic-Type-System, um eine flexible Wiederverwendbarkeit für verschiedene arithmetische Datentypen (z. B. INT, REAL, LREAL) zu ermöglichen.
- Die Verbindung erfolgt ausschließlich über Adapter des Typs `adapter::types::unidirectional::AUDI`, die eine standardisierte unidirektionale Schnittstelle für arithmetische Operationen bereitstellen.

Zustandsübersicht

Der FB besitzt keine Zustandsmaschine (keine ECC-Elemente in der XML). Das Verhalten ist rein datengetrieben: Bei jeder Änderung eines Eingangswerts wird das Ausgangsergebnis neu berechnet.

Anwendungsszenarien

- **Summierung von Messwerten:** Drei analoge Sensorwerte (z. B. Temperaturen) aufsummieren.
- **Mischungsberechnung:** Addieren von drei Mengen oder Volumenströmen in der Verfahrenstechnik.
- **Generische Addition:** Verwendung in Bibliotheken, bei denen der Datentyp erst zur Projektierungszeit festgelegt wird.

Vergleich mit ähnlichen Bausteinen

- **ADD_2:** Führt eine Addition von zwei Werten durch – `AUDI_ADD_3_UNGATED` erweitert dies auf drei Eingänge.
- **ADD_4:** Addiert vier Werte – `AUDI_ADD_3_UNGATED` spezifiziert drei Eingänge.
- **MUL_3:** Multipliziert drei Werte – im Gegensatz zur Addition. Alle genannten Bausteine verwenden das gleiche Adapter-Konzept und sind generisch.
- `AUDI_ADD_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

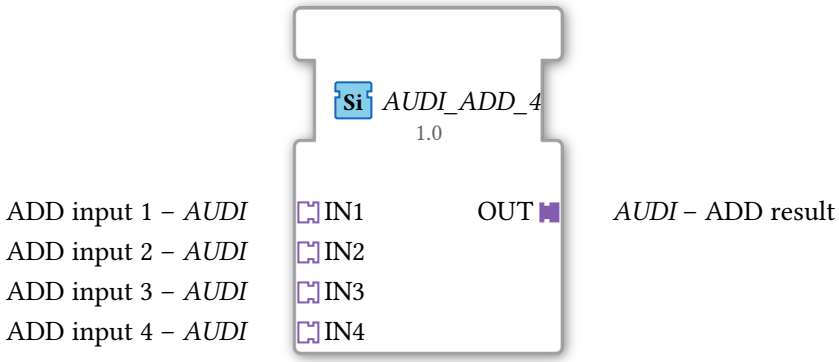
Fazit

Der `AUDI_ADD_3_UNGATED` ist ein flexibler, generischer Funktionsblock für die Addition von drei Werten. Seine reine Adapter-Schnittstelle und die Zustandslosigkeit machen ihn einfach in der Handhabung und ideal für modulare Automatisierungslösungen auf Basis von IEC 61499.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUDI_ADD_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AUDI_ADD_4` ist ein generischer Baustein zur Berechnung der arithmetischen Addition von vier Eingangswerten. Er ist als reiner Adapter-basierter Baustein ohne Ereignis- oder Datenschnittstellen realisiert. Die Verwendung von Adaptern ermöglicht eine flexible Kopplung mit anderen Bausteinen, die den gleichen Adaptertyp `adapter::types::unidirectional::AUDI` nutzen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Der Baustein besitzt keine direkten Dateneingänge. Die zu addierenden Werte werden über Adaptereingänge bereitgestellt.

Daten-Ausgänge

Der Baustein besitzt keine direkten Datenausgänge. Das Ergebnis wird über einen Adapterausgang weitergegeben.

Adapter

Bezeichnung	Richtung	Typ	Beschreibung
`IN1`	Socket (Eingang)	`adapter::types::unidirectional::AUDI`	Erster Summand
`IN2`	Socket (Eingang)	`adapter::types::unidirectional::AUDI`	Zweiter Summand
`IN3`	Socket (Eingang)	`adapter::types::unidirectional::AUDI`	Dritter Summand
`IN4`	Socket (Eingang)	`adapter::types::unidirectional::AUDI`	Vierter Summand
`OUT`	Plug (Ausgang)	`adapter::types::unidirectional::AUDI`	Ergebnis der Addition

Funktionsweise

Der Baustein wartet auf gültige Werte an den Adaptereingängen `IN1` .. `IN4`. Sobald alle vier Eingänge einen Wert liefern, wird die Summe `IN1 + IN2 + IN3 + IN4` gebildet und über den `OUT`-Adapter ausgegeben. Die tatsächliche Datentypspezifikation erfolgt durch das generische Attribut `eclipse4diac::core::GenericClassName`, das auf `'GEN_AUDI_ADD'` gesetzt ist. Dadurch kann der Baustein für verschiedene numerische Datentypen (z. B. INT, REAL, LREAL) instanziiert werden, sofern der verwendete Adaptertyp `AUDI` diese unterstützt.

Technische Besonderheiten

- **Reiner Adapterbaustein**: Der FB tauscht keine Ereignisse aus. Die Synchronisation erfolgt implizit durch die angeschlossenen Adapterverbindungen.
- **Generischer Datentyp**: Über die Attribute `GenericClassName` und `TypeHash` wird der konkrete Datentyp zur Laufzeit festgelegt. Dies ermöglicht eine wiederverwendbare Implementierung ohne Änderung der Logik.
- **Keine Zustandsautomaten**: Der FB enthält kein ECC (Execution Control Chart) – die Addition wird kontinuierlich oder auf Anforderung des Datenflusses durchgeführt.

Zustandsübersicht

Der Baustein besitzt keine expliziten Zustände. Die Verarbeitung erfolgt datengetrieben: Sobald alle vier Eingangswerte verfügbar sind, wird das Ergebnis berechnet und ausgegeben.

Anwendungsszenarien

- **Mittelwertbildung**: In Kombination mit einem nachgeschalteten Divisionsbaustein kann die Summe zur Berechnung eines Durchschnitts verwendet werden.
- **Summierung von Messwerten**: Zum Aufsummieren von vier analogen Eingangssignalen (z. B. Temperatur, Druck, Durchfluss) in einer Automatisierungslösung.
- **Kaskadierte Addition**: Mehrere `AUDI_ADD_4`-Bausteine lassen sich hintereinanderschalten, um eine größere Anzahl von Summanden zu verarbeiten.

Vergleich mit ähnlichen Bausteinen

- **`ADD` (Standard-61499)**: Ein typischer ADD-Baustein besitzt Ereigniseingänge und Dateneingänge/-ausgänge. Der `AUDI_ADD_4` hingegen ist komplett adapterbasiert und besitzt keine Ereignisse, was eine stärkere Kopplung zwischen Bausteinen über Adapter erfordert.
- **`AUDI_ADD_2`**: Ein hypothetischer Baustein mit nur zwei Adaptereingängen – `AUDI_ADD_4` erweitert dies auf vier Summanden.
- **Generische FBs**: Durch das Attribut `GenericClassName` ähnelt der Baustein dem generischen Ansatz von IEC 61499, bei dem der Datentyp erst zur Laufzeit festgelegt wird.
- **`AUDI_ADD_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

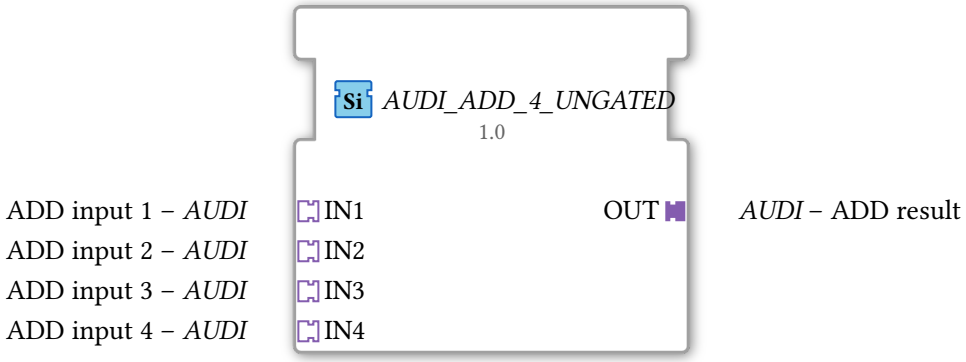
Der `AUDI_ADD_4` ist ein flexibler, reiner Adapterbaustein zur Addition von vier Werten. Dank seiner generischen Auslegung eignet er sich für verschiedene numerische Datentypen und kann in modularen Automatisierungsprojekten eingesetzt werden, die auf Adapter-basierte Kommunikation setzen. Der Verzicht auf Ereignisse vereinfacht die Handhabung in datengetriebenen Systemen, erfordert jedoch eine korrekte Adapterverkabelung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUDI_ADD_4_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `AUDI_ADD_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AUDI_ADD_4_UNGATED` ist ein generischer Baustein zur Berechnung der arithmetischen Addition von vier Eingangswerten. Er ist als reiner Adapter-basierter Baustein ohne Ereignis- oder Datenschnittstellen realisiert. Die Verwendung von Adaptern ermöglicht eine flexible Kopplung mit anderen Bausteinen, die den gleichen Adaptertyp `adapter::types::unidirectional::AUDI` nutzen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Der Baustein besitzt keine direkten Dateneingänge. Die zu addierenden Werte werden über Adaptereingänge bereitgestellt.

Daten-Ausgänge

Der Baustein besitzt keine direkten Datenausgänge. Das Ergebnis wird über einen Adapterausgang weitergegeben.

Adapter

Bezeichnung	Richtung	Typ	Beschreibung
`IN1`	Socket (Eingang)	`adapter::types::unidirectional::AUDI`	Erster Summand
`IN2`	Socket (Eingang)	`adapter::types::unidirectional::AUDI`	Zweiter Summand
`IN3`	Socket (Eingang)	`adapter::types::unidirectional::AUDI`	Dritter Summand
`IN4`	Socket (Eingang)	`adapter::types::unidirectional::AUDI`	Vierter Summand
`OUT`	Plug (Ausgang)	`adapter::types::unidirectional::AUDI`	Ergebnis der Addition

Funktionsweise

Der Baustein wartet auf gültige Werte an den Adaptereingängen `IN1` .. `IN4`. Sobald alle vier Eingänge einen Wert liefern, wird die Summe `IN1 + IN2 + IN3 + IN4` gebildet und über den `OUT`-Adapter ausgegeben. Die tatsächliche Datentypspezifikation erfolgt durch das generische Attribut `eclipse4diac::core::GenericClassName`, das auf `'GEN_AUDI_ADD'` gesetzt ist. Dadurch kann der Baustein für verschiedene numerische Datentypen (z. B. INT, REAL, LREAL) instanziiert werden, sofern der verwendete Adaptertyp `AUDI` diese unterstützt.

Technische Besonderheiten

- **Reiner Adapterbaustein**: Der FB tauscht keine Ereignisse aus. Die Synchronisation erfolgt implizit durch die angeschlossenen Adapterverbindungen.
- **Generischer Datentyp**: Über die Attribute `GenericClassName` und `TypeHash` wird der konkrete Datentyp zur Laufzeit festgelegt. Dies ermöglicht eine wiederverwendbare Implementierung ohne Änderung der Logik.
- **Keine Zustandsautomaten**: Der FB enthält kein ECC (Execution Control Chart) – die Addition wird kontinuierlich oder auf Anforderung des Datenflusses durchgeführt.

Zustandsübersicht

Der Baustein besitzt keine expliziten Zustände. Die Verarbeitung erfolgt datengetrieben: Sobald alle vier Eingangswerte verfügbar sind, wird das Ergebnis berechnet und ausgegeben.

Anwendungsszenarien

- **Mittelwertbildung**: In Kombination mit einem nachgeschalteten Divisionsbaustein kann die Summe zur Berechnung eines Durchschnitts verwendet werden.
- **Summierung von Messwerten**: Zum Aufsummieren von vier analogen Eingangssignalen (z. B. Temperatur, Druck, Durchfluss) in einer Automatisierungslösung.
- **Kaskadierte Addition**: Mehrere `AUDI_ADD_4_UNGATED`-Bausteine lassen sich hintereinanderschalten, um eine größere Anzahl von Summanden zu verarbeiten.

Vergleich mit ähnlichen Bausteinen

- **`ADD` (Standard-61499)**: Ein typischer ADD-Baustein besitzt Ereigniseingänge und Dateneingänge/-ausgänge. Der `AUDI_ADD_4_UNGATED` hingegen ist komplett adapterbasiert und besitzt keine Ereignisse, was eine stärkere Kopplung zwischen Bausteinen über Adapter erfordert.
- **`AUDI_ADD_2`**: Ein hypothetischer Baustein mit nur zwei Adaptereingängen – `AUDI_ADD_4_UNGATED` erweitert dies auf vier Summanden.
- **Generische FBs**: Durch das Attribut `GenericClassName` ähnelt der Baustein dem generischen Ansatz von IEC 61499, bei dem der Datentyp erst zur Laufzeit festgelegt wird.
- **`AUDI_ADD_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

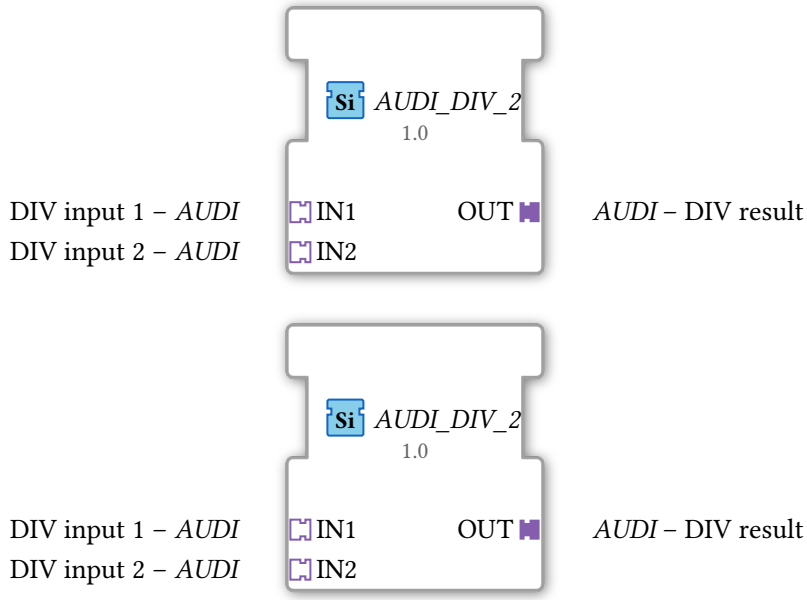
Fazit

Der `AUDI_ADD_4_UNGATED` ist ein flexibler, reiner Adapterbaustein zur Addition von vier Werten. Dank seiner generischen Auslegung eignet er sich für verschiedene numerische Datentypen und kann in modularen Automatisierungsprojekten eingesetzt werden, die auf Adapter-basierte Kommunikation setzen. Der Verzicht auf Ereignisse vereinfacht die Handhabung in datengetriebenen Systemen, erfordert jedoch eine korrekte Adapterverkabelung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUDI_DIV_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein (FB) **AUDI_DIV_2** dient zur Durchführung einer arithmetischen Division. Es handelt sich hierbei um einen generischen Funktionsbaustein (`GEN_AUDI_DIV`), der flexibel mit verschiedenen Datentypen innerhalb der Adapter-Struktur arbeiten kann. Im Gegensatz zu klassischen mathematischen Bausteinen, die über dedizierte Dateneingänge verfügen, nutzt dieser Baustein Adapterverbindungen zur Kapselung von Signalen und Steuerungsflüssen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten klassischen Ereignis- oder Dateneingänge auf der Gehäuseoberfläche. Die gesamte Kommunikation wird über Adapter abgewickelt.

```
        +--------------------------------------+
        |              AUDI_DIV_2              |
        |                                      |
[IN1] =|=====================================>|=[OUT]
        | (unidirectional::AUDI)  (unidirect..) |
[IN2] =|=====================================>|
        | (unidirectional::AUDI)               |
        +--------------------------------------+
```

Ereignis-Eingänge

Keine direkten Ereigniseingänge vorhanden. Die Ereignissteuerung wird über die Adapter abgewickelt.

Ereignis-Ausgänge

Keine direkten Ereignisausgänge vorhanden. Die Ereignissteuerung wird über die Adapter abgewickelt.

Daten-Eingänge

Keine direkten Dateneingänge vorhanden.

Daten-Ausgänge

Keine direkten Datenausgänge vorhanden.

Adapter

Typ	Name	Adaptertyp	Beschreibung

Socket (Buchse)	`IN1`	`adapter::types::unidirectional::AUDI`	Erster Eingangswert für die Division (Dividend).
Socket (Buchse)	`IN2`	`adapter::types::unidirectional::AUDI`	Zweiter Eingangswert für die Division (Divisor).
Plug (Stecker)	`OUT`	`adapter::types::unidirectional::AUDI`	Ergebnis der Division (Quotient).

Funktionsweise

Der Baustein berechnet den Quotienten aus den über die Adapter `IN1` und `IN2` bereitgestellten Werten:

$$\mathrm{OUT} = \frac{\mathrm{IN1}}{\mathrm{IN2}} \tag{2.86}$$

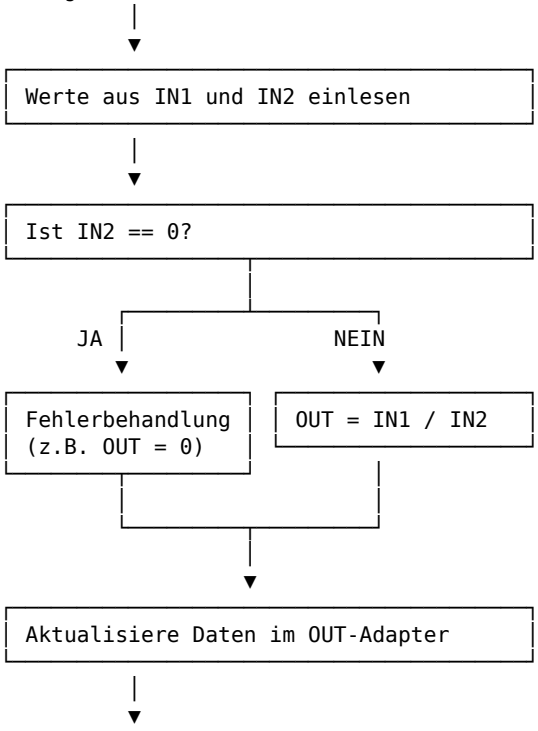
Da die Schnittstellen als unidirektionale `AUDI`-Adapter ausgeführt sind, triggert eine Aktualisierung der Eingangswerte an `IN1` oder `IN2` (über die im Adapter integrierten Ereigniskanäle) die Berechnung im Funktionsbaustein. Das Ergebnis wird anschließend unmittelbar über das im `OUT`-Adapter gekapselte Ereignis signalisiert und an nachfolgende Bausteine weitergegeben.

Technische Besonderheiten

- **Generischer Typ:** Der Baustein ist über das Attribut `eclipse4diac::core::GenericClassName` als `GEN_AUDI_DIV` deklariert. Dies ermöglicht eine automatische Typanpassung an die konkret verwendeten Datenstrukturen innerhalb der `AUDI`-Adapter.
- **Saubere Verdrahtung:** Durch die Kapselung von Daten- und Event-Signalen in Adaptern wird das visuelle „Spaghetticode"-Problem in 4diac-ide minimiert.
- **Fehlerbehandlung (Division durch Null):** Da es sich um eine generische Division handelt, fängt die zugrundeliegende Laufzeitumgebung (z.B. forte) eine Division durch Null ab, um einen Absturz der Steuerung zu verhindern. In einem solchen Fall wird der Ausgang typischerweise auf einen Standardwert (z.B. `0` oder `NaN`/`Infinity` je nach Datentyp) gesetzt.

Zustandsübersicht

Die Ausführung folgt einem einfachen, ereignisgesteuerten Schema:



Anwendungsszenarien

- **Skalierungs- und Normierungsaufgaben:** Division von Messwerten, die in standardisierten `AUDI`-Datenstrukturen übertragen werden.
- **Verhältnisberechnungen:** Bestimmung von Verhältnissen in der Prozessindustrie (z.B. Durchflussverhältnisse zweier Medien).
- **Signalverarbeitung:** Modularer Einsatz in Steuerungsarchitekturen, bei denen eine strikte Trennung durch Adapterstrukturen gefordert ist.

Vergleich mit ähnlichen Bausteinen

Baustein-Typ	Schnittstellenart	Verdrahtungsaufwand	Flexibilität
Standard-`DIV`	Einzelne Pins (REQ, IN1, IN2, OUT, CNF)	Hoch (viele Linien)	Gering (feste Datentypen pro Instanz)
`AUDI_DIV_2`	Adapter (`IN1`, `IN2`, `OUT`)	Sehr gering	Hoch (generische Anpassung über Adapter)

- `AUDI_DIV_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

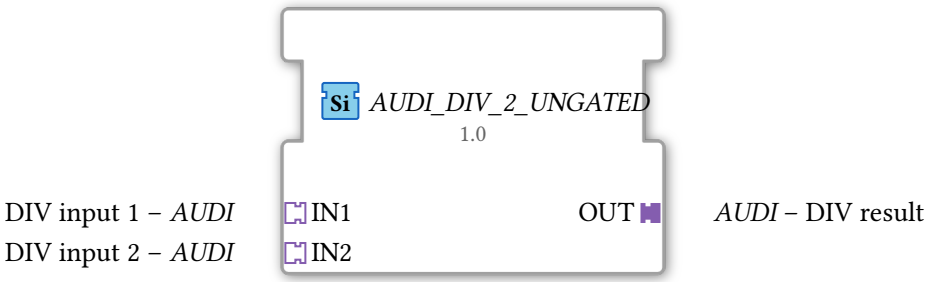
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

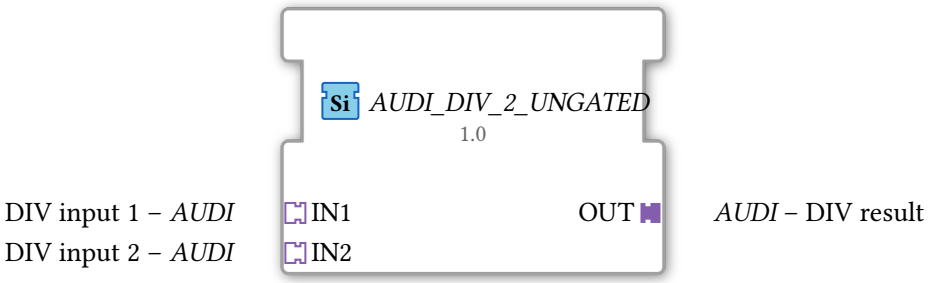
Fazit

Der `AUDI_DIV_2` ist ein moderner, adapterbasierter Funktionsbaustein für die IEC 61499. Er eignet sich hervorragend für komplexe Projekte in der 4diac-ide, bei denen ein hoher Wert auf Übersichtlichkeit, Kapselung und einfache Wartbarkeit des Applikationsdiagramms gelegt wird.

AUDI_DIV_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUDI_DIV_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein (FB) **AUDI_DIV_2_UNGATED** dient zur Durchführung einer arithmetischen Division. Es handelt sich hierbei um einen generischen Funktionsbaustein (`GEN_AUDI_DIV`), der flexibel mit verschiedenen Datentypen innerhalb der Adapter-Struktur arbeiten kann. Im Gegensatz zu klassischen mathematischen Bausteinen, die über dedizierte Dateneingänge verfügen, nutzt dieser Baustein Adapterverbindungen zur Kapselung von Signalen und Steuerungsflüssen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten klassischen Ereignis- oder Dateneingänge auf der Gehäuseoberfläche. Die gesamte Kommunikation wird über Adapter abgewickelt.

```
        +--------------------------------------+
        |              AUDI_DIV_2_UNGATED              |
        |                                      |
 [IN1] =|=====================================>|=[OUT]
        | (unidirectional::AUDI)  (unidirect..) |
 [IN2] =|=====================================>|
        | (unidirectional::AUDI)               |
        +--------------------------------------+
```

Ereignis-Eingänge

Keine direkten Ereigniseingänge vorhanden. Die Ereignissteuerung wird über die Adapter abgewickelt.

Ereignis-Ausgänge

Keine direkten Ereignisausgänge vorhanden. Die Ereignissteuerung wird über die Adapter abgewickelt.

Daten-Eingänge

Keine direkten Dateneingänge vorhanden.

Daten-Ausgänge

Keine direkten Datenausgänge vorhanden.

Adapter

Typ	Name	Adaptertyp	Beschreibung
Socket (Buchse)	`IN1`	`adapter::types::unidirectional::AUDI`	Erster Eingangswert für die Division (Dividend).
Socket (Buchse)	`IN2`	`adapter::types::unidirectional::AUDI`	Zweiter Eingangswert für die Division (Divisor).
Plug (Stecker)	`OUT`	`adapter::types::unidirectional::AUDI`	Ergebnis der Division (Quotient).

Funktionsweise

Der Baustein berechnet den Quotienten aus den über die Adapter `IN1` und `IN2` bereitgestellten Werten:

$$\mathrm{OUT} = \frac{\mathrm{IN1}}{\mathrm{IN2}} \tag{2.87}$$

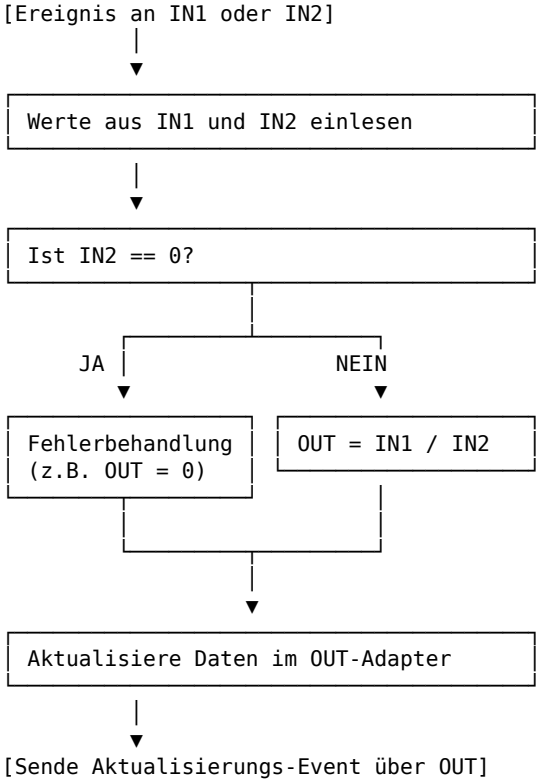
Da die Schnittstellen als unidirektionale `AUDI`-Adapter ausgeführt sind, triggert eine Aktualisierung der Eingangswerte an `IN1` oder `IN2` (über die im Adapter integrierten Ereigniskanäle) die Berechnung im Funktionsbaustein. Das Ergebnis wird anschließend unmittelbar über das im `OUT`-Adapter gekapselte Ereignis signalisiert und an nachfolgende Bausteine weitergegeben.

Technische Besonderheiten

- **Generischer Typ:** Der Baustein ist über das Attribut `eclipse4diac::core::GenericClassName` als `GEN_AUDI_DIV` deklariert. Dies ermöglicht eine automatische Typanpassung an die konkret verwendeten Datenstrukturen innerhalb der `AUDI`-Adapter.
- **Saubere Verdrahtung:** Durch die Kapselung von Daten- und Event-Signalen in Adaptern wird das visuelle „Spaghetticode"-Problem in 4diac-ide minimiert.
- **Fehlerbehandlung (Division durch Null):** Da es sich um eine generische Division handelt, fängt die zugrundeliegende Laufzeitumgebung (z.B. forte) eine Division durch Null ab, um einen Absturz der Steuerung zu verhindern. In einem solchen Fall wird der Ausgang typischerweise auf einen Standardwert (z.B. `0` oder `NaN`/`Infinity` je nach Datentyp) gesetzt.

Zustandsübersicht

Die Ausführung folgt einem einfachen, ereignisgesteuerten Schema:



Anwendungsszenarien

- **Skalierungs- und Normierungsaufgaben:** Division von Messwerten, die in standardisierten `AUDI`-Datenstrukturen übertragen werden.
- **Verhältnisberechnungen:** Bestimmung von Verhältnissen in der Prozessindustrie (z.B. Durchflussverhältnisse zweier Medien).
- **Signalverarbeitung:** Modularer Einsatz in Steuerungsarchitekturen, bei denen eine strikte Trennung durch Adapterstrukturen gefordert ist.

Vergleich mit ähnlichen Bausteinen

Baustein-Typ	Schnittstellenart	Verdrahtungsaufwand	Flexibilität

Standard-DIV	Einzelne Pins (REQ, IN1, IN2, OUT, CNF)	Hoch (viele Linien)	Gering (feste Datentypen pro Instanz)
AUDI_DIV_2_UNGATED	Adapter (IN1, IN2, OUT)	Sehr gering	Hoch (generische Anpassung über Adapter)

- **AUDI_DIV_2**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der AUDI_DIV_2_UNGATED ist ein moderner, adapterbasierter Funktionsbaustein für die IEC 61499. Er eignet sich hervorragend für komplexe Projekte in der 4diac-ide, bei denen ein hoher Wert auf Übersichtlichkeit, Kapselung und einfache Wartbarkeit des Applikationsdiagramms gelegt wird.

AUDI_MUL_2

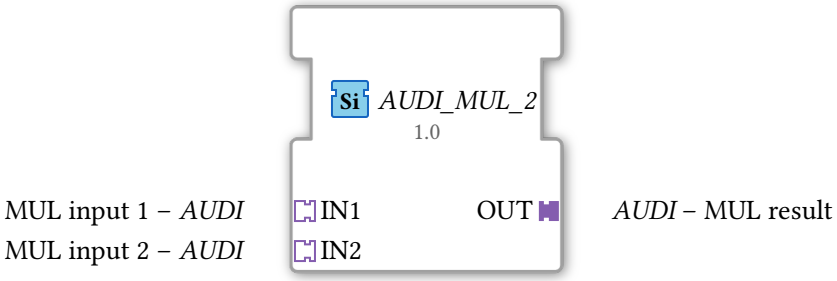


Bild: Kein Bild vorhanden.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AUDI_MUL_2 ist ein generischer Baustein zur Berechnung des arithmetischen Produkts zweier numerischer Werte. Er empfängt seine Operanden über zwei Eingangsadapter und gibt das Ergebnis über einen Ausgangsadapter aus. Die generische Ausprägung erlaubt die Verwendung unterschiedlicher Datentypen (z. B. Ganzzahlen, Gleitkommazahlen), die durch den konkreten Adaptertyp festgelegt werden.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine direkten Datenports. Die Eingangswerte werden über die Adapter IN1 und IN2 übertragen.

Daten-Ausgänge

Keine direkten Datenports. Das Ergebnis wird über den Adapter OUT ausgegeben.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	IN1	adapter::types::unidirectional::AUDI	Erster Multiplikationsoperand

Socket (Eingang)	IN2	adapter::types::unidirectional::AUDI	Zweiter Multiplikationsoperand
Plug (Ausgang)	OUT	adapter::types::unidirectional::AUDI	Produkt der beiden Eingangswerte

Anmerkung: Der Adaptertyp AUDI ist ein unidirektionaler, typisierter Adapter, der den eigentlichen numerischen Wert kapselt.

Funktionsweise

Der Baustein führt die Multiplikation OUT = IN1 * IN2 aus, sobald beide Eingangswerte über die Adapter anliegen. Da keine Ereignisse vorhanden sind, erfolgt die Verarbeitung rein datengetrieben: Sobald die Socket-Adapter IN1 und IN2 gültige Werte bereitstellen, wird das Produkt berechnet und über den Plug-Adapter OUT ausgegeben.

Der Funktionsblock ist generisch ausgelegt. Der konkrete Datentyp (z. B. REAL, INT, LINT) wird durch den jeweils verwendeten Adaptertyp vorgegeben. Die Multiplikation erfolgt mit der für diesen Typ definierten Operation.

Technische Besonderheiten

- **Adapterbasierte Kommunikation:** Der Baustein verwendet keine klassischen Ereignis-/Datenports, sondern ausschließlich Adapter. Dies ermöglicht eine lose Kopplung und Wiederverwendung in verschiedenen Kontexten.
- **Generische Auslegung:** Die Attribute eclipse4diac::core::GenericClassName und eclipse4diac::core::TypeHash kennzeichnen den Baustein als generisch. Der tatsächliche Datentyp wird erst im konkreten Projekt festgelegt, z. B. durch Typisierung des Adapters.
- **Kein Execution Control Chart (ECC):** Da keine Ereignisse existieren, wird kein Zustandsautomat benötigt. Die Verarbeitung erfolgt kontinuierlich bei gültigen Daten.

Zustandsübersicht

Der Baustein besitzt keine Zustände oder ECC-Zustandsdiagramme. Die Ausführung ergibt sich ausschließlich aus dem Datenfluss.

Anwendungsszenarien

- Arithmetische Berechnungen in industriellen Steuerungsanwendungen, bei denen die Multiplikation zweier Werte benötigt wird (z. B. Skalierung, Umrechnung von Einheiten).
- Verwendung in komplexen Funktionsbausteinen, in denen die Operanden über Adapter aus anderen Komponenten stammen.
- Ersatz für klassische MUL-Funktionsbausteine, wenn eine lose Kopplung über Adapter gewünscht ist.

Vergleich mit ähnlichen Bausteinen

- **Klassischer MUL-FB** (mit Ereignis-/Datenports): Arbeitet ereignisgesteuert und benötigt explizite Dateneingänge/-ausgänge. AUDI_MUL_2 hingegen verwendet Adapter, was eine flexiblere Verschaltung ermöglicht.
- **Andere generische Arithmetik-Bausteine** (z. B. AUDI_ADD_2): Gleiche Struktur, aber mit anderer Operation. Das Adapter-Interface bleibt identisch, sodass eine einfache Austauschbarkeit gegeben ist.
- **AUDI_MUL_2_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

AUDI_MUL_2 ist ein generischer, adapterbasierter Multiplikationsbaustein für IEC 61499-Anwendungen. Durch den Verzicht auf Ereignisse und die Nutzung typisierter Adapter eignet er sich besonders für datenflussorientierte und lose gekoppelte Systeme. Die generische Ausprägung erlaubt den Einsatz mit verschiedenen numerischen Datentypen, ohne dass der Baustein selbst angepasst werden muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUDI_MUL_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUDI_MUL_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

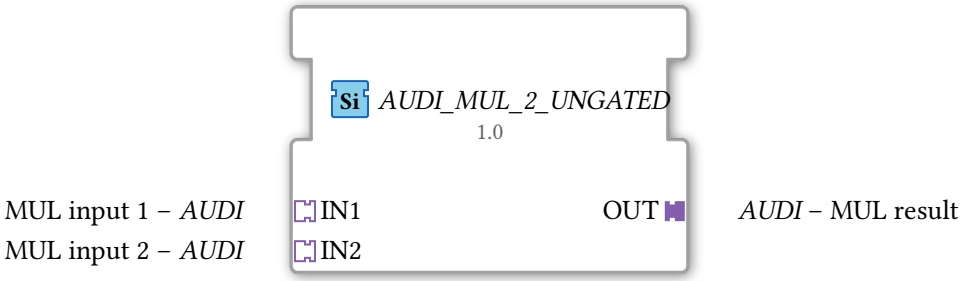


Bild: Kein Bild vorhanden.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AUDI_MUL_2_UNGATED` ist ein generischer Baustein zur Berechnung des arithmetischen Produkts zweier numerischer Werte. Er empfängt seine Operanden über zwei Eingangsadapter und gibt das Ergebnis über einen Ausgangsadapter aus. Die generische Ausprägung erlaubt die Verwendung unterschiedlicher Datentypen (z. B. Ganzzahlen, Gleitkommazahlen), die durch den konkreten Adaptertyp festgelegt werden.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine direkten Datenports. Die Eingangswerte werden über die Adapter `IN1` und `IN2` übertragen.

Daten-Ausgänge

Keine direkten Datenports. Das Ergebnis wird über den Adapter `OUT` ausgegeben.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	`IN1`	`adapter::types::unidirectional::AUDI`	Erster Multiplikationsoperand
Socket (Eingang)	`IN2`	`adapter::types::unidirectional::AUDI`	Zweiter Multiplikationsoperand
Plug (Ausgang)	`OUT`	`adapter::types::unidirectional::AUDI`	Produkt der beiden Eingangswerte

Anmerkung: Der Adaptertyp `AUDI` ist ein unidirektionaler, typisierter Adapter, der den eigentlichen numerischen Wert kapselt.

Funktionsweise

Der Baustein führt die Multiplikation `OUT = IN1 * IN2` aus, sobald beide Eingangswerte über die Adapter anliegen. Da keine Ereignisse vorhanden sind, erfolgt die Verarbeitung rein datengetrieben: Sobald die Socket-Adapter `IN1` und `IN2` gültige Werte bereitstellen, wird das Produkt berechnet und über den Plug-Adapter `OUT` ausgegeben.

Der Funktionsblock ist generisch ausgelegt. Der konkrete Datentyp (z. B. `REAL`, `INT`, `LINT`) wird durch den jeweils verwendeten Adaptertyp vorgegeben. Die Multiplikation erfolgt mit der für diesen Typ definierten Operation.

Technische Besonderheiten

- **Adapterbasierte Kommunikation:** Der Baustein verwendet keine klassischen Ereignis-/Datenports, sondern ausschließlich Adapter. Dies ermöglicht eine lose Kopplung und Wiederverwendung in verschiedenen Kontexten.
- **Generische Auslegung:** Die Attribute `eclipse4diac::core::GenericClassName` und `eclipse4diac::core::TypeHash` kennzeichnen den Baustein als generisch. Der tatsächliche Datentyp wird erst im konkreten Projekt festgelegt, z. B. durch Typisierung des Adapters.
- **Kein Execution Control Chart (ECC):** Da keine Ereignisse existieren, wird kein Zustandsautomat benötigt. Die Verarbeitung erfolgt kontinuierlich bei gültigen Daten.

Zustandsübersicht

Der Baustein besitzt keine Zustände oder ECC-Zustandsdiagramme. Die Ausführung ergibt sich ausschließlich aus dem Datenfluss.

Anwendungsszenarien

- Arithmetische Berechnungen in industriellen Steuerungsanwendungen, bei denen die Multiplikation zweier Werte benötigt wird (z. B. Skalierung, Umrechnung von Einheiten).
- Verwendung in komplexen Funktionsbausteinen, in denen die Operanden über Adapter aus anderen Komponenten stammen.
- Ersatz für klassische MUL-Funktionsbausteine, wenn eine lose Kopplung über Adapter gewünscht ist.

Vergleich mit ähnlichen Bausteinen

- **Klassischer MUL-FB** (mit Ereignis-/Datenports): Arbeitet ereignisgesteuert und benötigt explizite Dateneingänge/-ausgänge. `AUDI_MUL_2_UNGATED` hingegen verwendet Adapter, was eine flexiblere Verschaltung ermöglicht.
- **Andere generische Arithmetik-Bausteine** (z. B. `AUDI_ADD_2`): Gleiche Struktur, aber mit anderer Operation. Das Adapter-Interface bleibt identisch, sodass eine einfache Austauschbarkeit gegeben ist.
- `AUDI_MUL_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- `AUDI_MUL_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

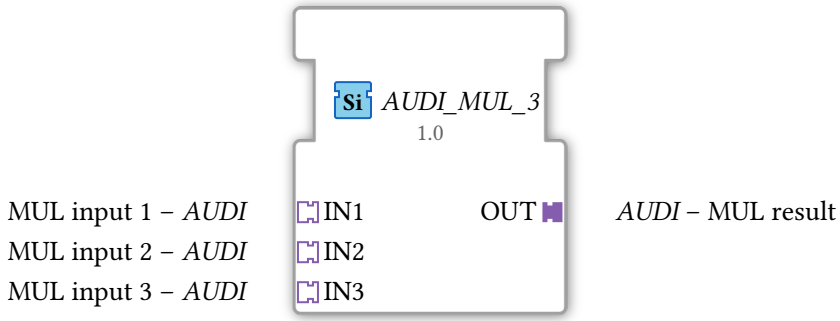
Fazit

`AUDI_MUL_2_UNGATED` ist ein generischer, adapterbasierter Multiplikationsbaustein für IEC 61499-Anwendungen. Durch den Verzicht auf Ereignisse und die Nutzung typisierter Adapter eignet er sich besonders für datenflussorientierte und lose gekoppelte Systeme. Die generische Ausprägung erlaubt den Einsatz mit verschiedenen numerischen Datentypen, ohne dass der Baustein selbst angepasst werden muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUDI_MUL_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUDI_MUL_3** ist ein generischer Baustein zur Multiplikation von drei Eingangswerten. Die Werte werden über unidirektionale Adapter vom Typ **AUDI** bereitgestellt und das Ergebnis über einen gleichartigen Adapter ausgegeben. Der Baustein ist für den Einsatz in IEC 61499-basierten Steuerungssystemen konzipiert.

Schnittstellenstruktur

Ereignis-Eingänge

Nicht vorhanden – die gesamte Datenübertragung erfolgt adapterbasiert ohne explizite Ereignissteuerung.

Ereignis-Ausgänge

Nicht vorhanden.

Daten-Eingänge

Nicht vorhanden – die Eingangswerte werden über die **Sockets IN1, IN2 und IN3** als Adapter bereitgestellt.

Daten-Ausgänge

Nicht vorhanden – das Ergebnis wird über den **Plug OUT** als Adapter ausgegeben.

Adapter

Name	Richtung	Typ	Beschreibung
IN1	Socket (Eingang)	`adapter::types::unidirectional::AUDI`	Erster Multiplikationsfaktor.
IN2	Socket (Eingang)	`adapter::types::unidirectional::AUDI`	Zweiter Multiplikationsfaktor.
IN3	Socket (Eingang)	`adapter::types::unidirectional::AUDI`	Dritter Multiplikationsfaktor.
OUT	Plug (Ausgang)	`adapter::types::unidirectional::AUDI`	Ergebnis der Multiplikation `IN1 * IN2 * IN3`.

Funktionsweise

Der FB **AUDI_MUL_3** führt eine arithmetische Multiplikation der drei über die Adapter **IN1, IN2 und IN3** bereitgestellten Werte durch. Das Produkt wird kontinuierlich berechnet und über den Adapter **OUT** zur Verfügung gestellt. Der zugrundeliegende Datentyp **AUDI** ist generisch definiert (über das Attribut `eclipse4diac::core::GenericClassName`) und kann zur Konfigurationszeit durch einen konkreten numerischen Typ (z. B. INT, REAL) ersetzt werden. Eine explizite Triggerung durch Ereignisse ist nicht erforderlich, da die Adapter die Datenübertragung eigenständig abwickeln.

Technische Besonderheiten

- **Generischer Datentyp**: Der FB ist als generischer Baustein (`GenericClassName = 'GEN_AUDI_MUL'`) ausgelegt. Der verwendete Adaptertyp **AUDI** kann durch einen beliebigen numerischen Datentyp ersetzt werden, solange der gleiche Typ an allen Adaptern verwendet wird.
- **Adapterbasiertes Interface**: Statt diskreter Daten- und Ereignisports kommuniziert der FB ausschließlich über unidirektionale Adapter. Dies ermöglicht eine flexible Verkettung mit anderen Bausteinen desselben Adaptertyps.
- **Keine Explizite Zustandslogik**: Der FB benötigt keinen inneren Zustandsautomaten – die Multiplikation erfolgt rein datenflussgesteuert.

Zustandsübersicht

Der **AUDI_MUL_3** besitzt keinen expliziten Zustandsautomaten. Die Ausgabe **OUT** entspricht stets dem aktuellen Produkt der drei Eingänge. Sobald an allen Sockets gültige Daten anliegen, wird das Ergebnis ohne Verzögerung berechnet.

Anwendungsszenarien

- **Mehrdimensionale Berechnungen**: Kombination mehrerer physikalischer Größen, die multipliziert werden müssen (z. B. Volumen aus Länge, Breite, Höhe).
- **Skalierung von Signalen**: Multiplikation eines Sensorwerts mit zwei Kalibrierungsfaktoren.
- **Verschachtelte Arithmetik**: Einsatz als Teil einer größeren Berechnungskette, bei der das Produkt dreier Eingänge benötigt wird.

Vergleich mit ähnlichen Bausteinen

- **AUDI_MUL_2**: Führt die Multiplikation von nur zwei Eingängen durch. Der **AUDI_MUL_3** erweitert diese Funktionalität auf drei Faktoren.
- **AUDI_ADD_3**: Führt eine Addition dreier Werte durch, während der **AUDI_MUL_3** eine Multiplikation realisiert – beide nutzen die gleiche Adapterstruktur.
- **Konventionelle IEC 61499-Bausteine**: Übliche Multifunktionsblöcke verwenden oft separate Daten- und Ereignisports. Der adapterbasierte Ansatz vereinfacht die Typisierung und Wiederverwendbarkeit in generischen Frameworks.
- `AUDI_MUL_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

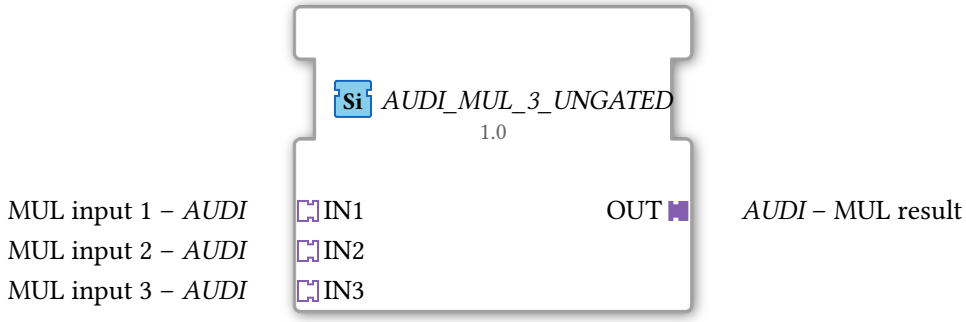
Der **AUDI_MUL_3** ist ein kompakter, generischer Funktionsblock zur Multiplikation von drei Werten. Durch die adaptergesteuerte Schnittstelle und die generische Typisierung eignet er sich hervorragend für modulare, wiederverwendbare Steuerungsapplikationen. Die einfache datenflussorientierte Arbeitsweise ohne Zustandsautomaten macht ihn intuitiv einsetzbar und erweiterbar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUDI_MUL_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUDI_MUL_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUDI_MUL_3_UNGATED** ist ein generischer Baustein zur Multiplikation von drei Eingangswerten. Die Werte werden über unidirektionale Adapter vom Typ **AUDI** bereitgestellt und das Ergebnis über einen gleichartigen Adapter ausgegeben. Der Baustein ist für den Einsatz in IEC 61499-basierten Steuerungssystemen konzipiert.

Schnittstellenstruktur

Ereignis-Eingänge

Nicht vorhanden – die gesamte Datenübertragung erfolgt adapterbasiert ohne explizite Ereignissteuerung.

Ereignis-Ausgänge

Nicht vorhanden.

Daten-Eingänge

Nicht vorhanden – die Eingangswerte werden über die **Sockets IN1, IN2 und IN3** als Adapter bereitgestellt.

Daten-Ausgänge

Nicht vorhanden – das Ergebnis wird über den **Plug OUT** als Adapter ausgegeben.

Adapter

Name	Richtung	Typ	Beschreibung
IN1	Socket (Eingang)	`adapter::types::unidirectional::AUDI`	Erster Multiplikationsfaktor.
IN2	Socket (Eingang)	`adapter::types::unidirectional::AUDI`	Zweiter Multiplikationsfaktor.
IN3	Socket (Eingang)	`adapter::types::unidirectional::AUDI`	Dritter Multiplikationsfaktor.
OUT	Plug (Ausgang)	`adapter::types::unidirectional::AUDI`	Ergebnis der Multiplikation `IN1 * IN2 * IN3`.

Funktionsweise

Der FB **AUDI_MUL_3_UNGATED** führt eine arithmetische Multiplikation der drei über die Adapter **IN1, IN2 und IN3** bereitgestellten Werte durch. Das Produkt wird kontinuierlich berechnet und über den Adapter **OUT** zur Verfügung gestellt. Der zugrundeliegende Datentyp **AUDI** ist generisch definiert (über das Attribut `eclipse4diac::core::GenericClassName`) und kann zur Konfigurationszeit durch einen konkreten numerischen Typ (z. B. INT, REAL) ersetzt werden. Eine explizite Triggerung durch Ereignisse ist nicht erforderlich, da die Adapter die Datenübertragung eigenständig abwickeln.

Technische Besonderheiten

- **Generischer Datentyp**: Der FB ist als generischer Baustein (`GenericClassName = 'GEN_AUDI_MUL'`) ausgelegt. Der verwendete Adaptertyp **AUDI** kann durch einen beliebigen numerischen Datentyp ersetzt werden, solange der gleiche Typ an allen Adaptern verwendet wird.

- **Adapterbasiertes Interface**: Statt diskreter Daten- und Ereignisports kommuniziert der FB ausschließlich über unidirektionale Adapter. Dies ermöglicht eine flexible Verkettung mit anderen Bausteinen desselben Adaptertyps.
- **Keine Explizite Zustandslogik**: Der FB benötigt keinen inneren Zustandsautomaten – die Multiplikation erfolgt rein datenflussgesteuert.

Zustandsübersicht

Der **AUDI_MUL_3_UNGATED** besitzt keinen expliziten Zustandsautomaten. Die Ausgabe **OUT** entspricht stets dem aktuellen Produkt der drei Eingänge. Sobald an allen Sockets gültige Daten anliegen, wird das Ergebnis ohne Verzögerung berechnet.

Anwendungsszenarien

- **Mehrdimensionale Berechnungen**: Kombination mehrerer physikalischer Größen, die multipliziert werden müssen (z. B. Volumen aus Länge, Breite, Höhe).
- **Skalierung von Signalen**: Multiplikation eines Sensorwerts mit zwei Kalibrierungsfaktoren.
- **Verschachtelte Arithmetik**: Einsatz als Teil einer größeren Berechnungskette, bei der das Produkt dreier Eingänge benötigt wird.

Vergleich mit ähnlichen Bausteinen

- **AUDI_MUL_2**: Führt die Multiplikation von nur zwei Eingängen durch. Der **AUDI_MUL_3_UNGATED** erweitert diese Funktionalität auf drei Faktoren.
- **AUDI_ADD_3**: Führt eine Addition dreier Werte durch, während der **AUDI_MUL_3_UNGATED** eine Multiplikation realisiert – beide nutzen die gleiche Adapterstruktur.
- **Konventionelle IEC 61499-Bausteine**: Übliche Multifunktionsblöcke verwenden oft separate Daten- und Ereignisports. Der adapterbasierte Ansatz vereinfacht die Typisierung und Wiederverwendbarkeit in generischen Frameworks.
- `AUDI_MUL_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

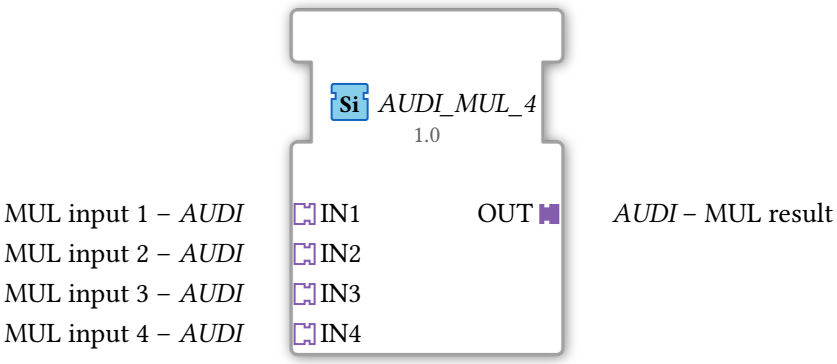
Fazit

Der **AUDI_MUL_3_UNGATED** ist ein kompakter, generischer Funktionsblock zur Multiplikation von drei Werten. Durch die adaptergesteuerte Schnittstelle und die generische Typisierung eignet er sich hervorragend für modulare, wiederverwendbare Steuerungsapplikationen. Die einfache datenflussorientierte Arbeitsweise ohne Zustandsautomaten macht ihn intuitiv einsetzbar und erweiterbar.

🌐 **Passende Themen-Unterseiten auf ms-muc-docs.de**

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUDI_MUL_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AUDI_MUL_4** führt eine arithmetische Multiplikation auf Basis von vier Eingangswerten durch. Es handelt sich um einen generischen Baustein (Generic FB), der über Adapter-Schnittstellen mit einer unidirektionalen Datenstruktur vom Typ `adapter::types::unidirectional::AUDI` arbeitet. Der Baustein ist gemäß IEC 61499-2 spezifiziert und wird insbesondere in Automatisierungslösungen eingesetzt, bei denen eine skalierbare Multiplikation mehrerer Eingangsgrößen erforderlich ist.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Der Baustein verfügt über keine klassischen Dateneingänge. Alle Eingangswerte werden über Adapter-Schnittstellen bereitgestellt.

Daten-Ausgänge

Der Baustein verfügt über keine klassischen Datenausgänge. Das Multiplikationsergebnis wird über einen Adapter-Plug ausgegeben.

Adapter

Typ	Name	Richtung	Kommentar
`adapter::types::unidirectional::AUDI`	IN1	Eingang (Socket)	MUL input 1
`adapter::types::unidirectional::AUDI`	IN2	Eingang (Socket)	MUL input 2
`adapter::types::unidirectional::AUDI`	IN3	Eingang (Socket)	MUL input 3
`adapter::types::unidirectional::AUDI`	IN4	Eingang (Socket)	MUL input 4
`adapter::types::unidirectional::AUDI`	OUT	Ausgang (Plug)	MUL result

Der Adapter `AUDI` ist als unidirektionaler Datentyp definiert und transportiert die jeweiligen numerischen Werte sowie das Ergebnis.

Funktionsweise

Der FB multipliziert die an den vier Adaptern **IN1**, **IN2**, **IN3** und **IN4** anliegenden Werte und stellt das Produkt am Ausgangsadapter **OUT** bereit. Die Verarbeitung erfolgt asynchron, sobald alle erforderlichen Eingangsdaten anliegen. Der Baustein ist als generischer FB ausgelegt, d.h. der verwendete Datentyp kann über das Attribut `eclipse4diac::core::GenericClassName` zur Laufzeit oder beim Deployment festgelegt werden. Standardmäßig wird der Name `'GEN_AUDI_MUL'` verwendet.

Technische Besonderheiten

- **Generische Implementierung**: Der FB ist generisch und kann an verschiedene numerische Datentypen angepasst werden (z. B. INT, REAL, LREAL). Die Konkretisierung erfolgt über das Klassenattribut.
- **Adapterbasierte Kommunikation**: Anstelle von direkten Ein-/Ausgängen werden Adapter genutzt. Dies erlaubt eine flexible Kapselung der Daten und ermöglicht die Wiederverwendung in unterschiedlichen Kontexten.
- **Typ-Hash**: Der Attributwert `eclipse4diac::core::TypeHash` ist leer und kann zur Laufzeit durch den tatsächlichen Hash des verwendeten Datentyps ergänzt werden.
- **Keine Zustandsautomaten**: Der FB besitzt kein ECC (Ereignisgesteuerter Zustandsautomat), da die Multiplikation rein datengetrieben erfolgt.

Zustandsübersicht

Nicht zutreffend – der Baustein arbeitet ohne Zustandsmaschine. Die Multiplikation wird durch die Verfügbarkeit der Eingangsdaten ausgelöst.

Anwendungsszenarien

- **Prozessautomatisierung**: Multiplikation von vier analogen Messwerten (z. B. Durchfluss, Druck, Temperatur, Dichte) zur Berechnung einer Massenstromkorrektur.
- **Maschinensteuerung**: Berechnung von Gesamtlasten oder Drehmomenten aus mehreren Einzelgrößen.
- **Datenvorverarbeitung**: Skalierung oder Gewichtung von Sensordaten in einem übergeordneten Regelungssystem.
- **Bildverarbeitung**: Kombination mehrerer Kanäle (z. B. Rot, Grün, Blau, Intensität) durch Multiplikation.

Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu klassischen IEC-61131-Multiplikationsbausteinen (z. B. `MUL` mit festen Datentypen) bietet `AUDI_MUL_4` eine generische, adapterbasierte Schnittstelle, die eine flexible Wiederverwendung ermöglicht. Andere Multiplikatoren wie `MUL_2` oder `AUDI_MUL_2` verarbeiten nur zwei Eingänge, während `AUDI_MUL_4` vier Eingänge gleichzeitig multipliziert. Bausteine mit ereignisgesteuerter Verarbeitung (z. B. `MUL_E`) erfordern zusätzliche Trigger, während dieser FB rein datengesteuert arbeitet.

- `AUDI_MUL_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

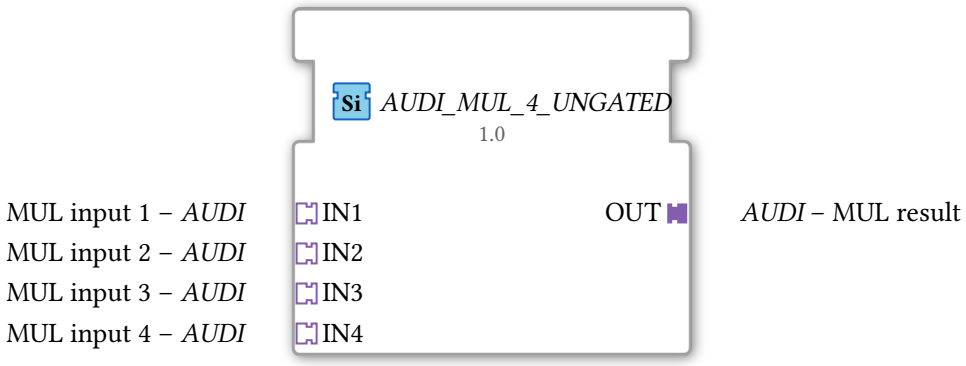
Der FB **AUDI_MUL_4** ist ein spezialisierter, generischer Multiplikationsbaustein für vier Eingangswerte. Durch die Nutzung von Adaptern und die generische Auslegung eignet er sich besonders für modulare, wiederverwendbare Automatisierungslösungen, bei denen Flexibilität und Skalierbarkeit im Vordergrund stehen. Die einfache Schnittstelle ohne Ereignisse macht die Integration in datenorientierte Architekturen unkompliziert.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUDI_MUL_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUDI_MUL_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AUDI_MUL_4_UNGATED** führt eine arithmetische Multiplikation auf Basis von vier Eingangswerten durch. Es handelt sich um einen generischen Baustein (Generic FB), der über Adapter-Schnittstellen mit einer unidirektionalen Datenstruktur vom Typ `adapter::types::unidirectional::AUDI` arbeitet. Der Baustein ist gemäß IEC 61499-2 spezifiziert und wird insbesondere in Automatisierungslösungen eingesetzt, bei denen eine skalierbare Multiplikation mehrerer Eingangsgrößen erforderlich ist.

Schnittstellenstruktur

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Der Baustein verfügt über keine klassischen Dateneingänge. Alle Eingangswerte werden über Adapter-Schnittstellen bereitgestellt.

Daten-Ausgänge

Der Baustein verfügt über keine klassischen Datenausgänge. Das Multiplikationsergebnis wird über einen Adapter-Plug ausgegeben.

Adapter

Typ	Name	Richtung	Kommentar
`adapter::types::unidirectional::AUDI`	IN1	Eingang (Socket)	MUL input 1
`adapter::types::unidirectional::AUDI`	IN2	Eingang (Socket)	MUL input 2
`adapter::types::unidirectional::AUDI`	IN3	Eingang (Socket)	MUL input 3
`adapter::types::unidirectional::AUDI`	IN4	Eingang (Socket)	MUL input 4
`adapter::types::unidirectional::AUDI`	OUT	Ausgang (Plug)	MUL result

Der Adapter `AUDI` ist als unidirektionaler Datentyp definiert und transportiert die jeweiligen numerischen Werte sowie das Ergebnis.

Funktionsweise

Der FB multipliziert die an den vier Adaptern **IN1**, **IN2**, **IN3** und **IN4** anliegenden Werte und stellt das Produkt am Ausgangsadapter **OUT** bereit. Die Verarbeitung erfolgt asynchron, sobald alle erforderlichen Eingangsdaten anliegen. Der Baustein ist als generischer FB ausgelegt, d.h. der verwendete Datentyp kann über das Attribut `eclipse4diac::core::GenericClassName` zur Laufzeit oder beim Deployment festgelegt werden. Standardmäßig wird der Name `'GEN_AUDI_MUL'` verwendet.

Technische Besonderheiten

- **Generische Implementierung**: Der FB ist generisch und kann an verschiedene numerische Datentypen angepasst werden (z. B. INT, REAL, LREAL). Die Konkretisierung erfolgt über das Klassenattribut.
- **Adapterbasierte Kommunikation**: Anstelle von direkten Ein-/Ausgängen werden Adapter genutzt. Dies erlaubt eine flexible Kapselung der Daten und ermöglicht die Wiederverwendung in unterschiedlichen Kontexten.
- **Typ-Hash**: Der Attributwert `eclipse4diac::core::TypeHash` ist leer und kann zur Laufzeit durch den tatsächlichen Hash des verwendeten Datentyps ergänzt werden.
- **Keine Zustandsautomaten**: Der FB besitzt kein ECC (Ereignisgesteuerter Zustandsautomat), da die Multiplikation rein datengetrieben erfolgt.

Zustandsübersicht

Nicht zutreffend – der Baustein arbeitet ohne Zustandsmaschine. Die Multiplikation wird durch die Verfügbarkeit der Eingangsdaten ausgelöst.

Anwendungsszenarien

- **Prozessautomatisierung**: Multiplikation von vier analogen Messwerten (z. B. Durchfluss, Druck, Temperatur, Dichte) zur Berechnung einer Massenstromkorrektur.
- **Maschinensteuerung**: Berechnung von Gesamtlasten oder Drehmomenten aus mehreren Einzelgrößen.
- **Datenvorverarbeitung**: Skalierung oder Gewichtung von Sensordaten in einem übergeordneten Regelungssystem.
- **Bildverarbeitung**: Kombination mehrerer Kanäle (z. B. Rot, Grün, Blau, Intensität) durch Multiplikation.

Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu klassischen IEC-61131-Multiplikationsbausteinen (z. B. `MUL` mit festen Datentypen) bietet `AUDI_MUL_4_UNGATED` eine generische, adapterbasierte Schnittstelle, die eine flexible Wiederverwendung ermöglicht. Andere Multiplikatoren wie `MUL_2` oder `AUDI_MUL_2` verarbeiten nur zwei Eingänge, während `AUDI_MUL_4_UNGATED` vier Eingänge gleichzeitig multipliziert. Bausteine mit ereignisgesteuerter Verarbeitung (z. B. `MUL_E`) erfordern zusätzliche Trigger, während dieser FB rein datengesteuert arbeitet.

- **`AUDI_MUL_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- **`AUDI_MUL_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

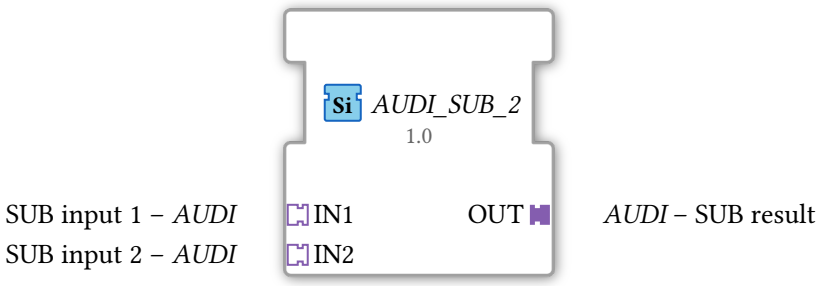
Fazit

Der FB **AUDI_MUL_4_UNGATED** ist ein spezialisierter, generischer Multiplikationsbaustein für vier Eingangswerte. Durch die Nutzung von Adaptern und die generische Auslegung eignet er sich besonders für modulare, wiederverwendbare Automatisierungslösungen, bei denen Flexibilität und Skalierbarkeit im Vordergrund stehen. Die einfache Schnittstelle ohne Ereignisse macht die Integration in datenorientierte Architekturen unkompliziert.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUDI_SUB_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein (FB) `AUDI_SUB_2` ist ein generischer Funktionsbaustein zur Durchführung einer arithmetischen Subtraktion. Er basiert auf der generischen Klasse `GEN_AUDI_SUB` und nutzt unidirektionale Adapter vom Typ `AUDI` zur strukturierten und ereignisgesteuerten Datenübertragung. Dadurch wird eine

saubere Kapselung von Daten und Events erreicht, was die Komplexität der Verkabelung in IEC 61499 Anwendungen reduziert.

Schnittstellenstruktur

Da der Baustein vollständig auf Adaptern basiert, besitzt er keine klassischen, direkt sichtbaren Ereignis- oder Daten-Pins auf der obersten Ebene. Die gesamte Kommunikation wird über die Adapter abgewickelt.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung ist in den Adaptern gekapselt.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Ereignissteuerung ist in den Adaptern gekapselt.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden. Die Datenübertragung erfolgt über die Adapter-Schnittstellen.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden. Die Datenübertragung erfolgt über die Adapter-Schnittstellen.

Adapter

Sockets (Eingangs-Adapter / Steckdosen)

- **IN1** (Typ: `adapter::types::unidirectional::AUDI`):
  - Erster Eingang der Subtraktion (Minuend).
- **IN2** (Typ: `adapter::types::unidirectional::AUDI`):
  - Zweiter Eingang der Subtraktion (Subtrahend).

Plugs (Ausgangs-Adapter / Stecker)

- **OUT** (Typ: `adapter::types::unidirectional::AUDI`):
  - Ausgang des Bausteins, der das Ergebnis der Subtraktion (Differenz) bereitstellt.

Funktionsweise

Der Funktionsbaustein berechnet die mathematische Differenz der beiden über die Eingangs-Adapter bereitgestellten Werte nach folgendem Prinzip:

$$\mathrm{OUT} = \mathrm{IN1} - \mathrm{IN2} \tag{2.88}$$

Sobald sich Werte an den Eingangs-Adaptern `IN1` oder `IN2` ändern und ein entsprechendes Trigger-Ereignis über den Adapter eingeht, wird die Berechnung intern ausgeführt. Das Ergebnis sowie das dazugehörige Update-Ereignis werden anschließend über den Ausgangs-Adapter `OUT` weitergeleitet.

Technische Besonderheiten

- **Generische Implementierung:** Durch das Attribut `eclipse4diac::core::GenericClassName = 'GEN_AUDI_SUB'` ist der Baustein flexibel einsetzbar. Je nach Implementierung des zugrundeliegenden Adapters kann er verschiedene Datentypen unterstützen.
- **Kapselung durch Adapter:** Die Verwendung des unidirektionalen Adapters `AUDI` bündelt Daten- und Eventleitungen. Dies erhöht die Übersichtlichkeit im 4diac-Applikationseditor erheblich, da weniger Linien gezogen werden müssen.

Zustandsübersicht

Der Baustein verhält sich wie ein zustandsloser (stateless) mathematischer Operator. Es gibt keine interne Zustandsschleife (ECC-Zustände im klassischen Sinne), die über längere Zeit gehalten wird. Jede Aktivierung durch ein Ereignis an den Eingängen führt zu einer direkten Berechnung und Aktualisierung des Ausgangs.

Anwendungsszenarien

- **Signalverarbeitung:** Subtraktion von Sensorwerten (z. B. Offset-Kompensation oder Berechnung von Differenzdrücken/-temperaturen) in Systemen, die konsequent auf der `AUDI`-Adapterarchitektur aufbauen.
- **Regelungstechnik:** Berechnung der Regeldifferenz ($e = w - x$) durch Subtraktion des Istwerts vom Sollwert.

Vergleich mit ähnlichen Bausteinen

- **Standard SUB-Baustein (IEC 61131-3 / IEC 61499):** Ein klassischer `SUB`-Baustein besitzt dedizierte Daten-Eingänge (`IN1`, `IN2`), einen Daten-Ausgang (`OUT`) sowie Event-Eingänge und -Ausgänge (z.B. `REQ` / `CNF`). Der `AUDI_SUB_2` hingegen bündelt diese Signale in Adaptern, was die Wiederverwendbarkeit und Modularität in komplexen Architekturen verbessert.
- **AUDI_ADD_2:** Das Gegenstück für die Addition. Verwendet dieselbe Adapter-Schnittstelle, addiert jedoch die Eingangswerte ($IN1 + IN2$).

- `AUDI_SUB_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

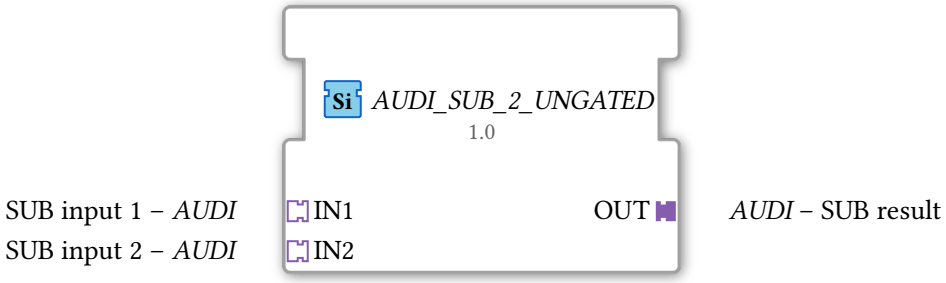
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `AUDI_SUB_2` ist ein spezialisierter, aber durch seine generische Natur dennoch flexibler Funktionsbaustein zur Subtraktion. Er eignet sich hervorragend für serviceorientierte Architekturen innerhalb von 4diac, bei denen ein einheitliches Schnittstellenkonzept mittels unidirektionaler Adapter im Vordergrund steht.

AUDI_SUB_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUDI_SUB_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein (FB) `AUDI_SUB_2_UNGATED` ist ein generischer Funktionsbaustein zur Durchführung einer arithmetischen Subtraktion. Er basiert auf der generischen Klasse `GEN_AUDI_SUB` und nutzt unidirektionale Adapter vom Typ `AUDI` zur strukturierten und ereignisgesteuerten Datenübertragung.

Dadurch wird eine saubere Kapselung von Daten und Events erreicht, was die Komplexität der Verkabelung in IEC 61499 Anwendungen reduziert.

Schnittstellenstruktur

Da der Baustein vollständig auf Adaptern basiert, besitzt er keine klassischen, direkt sichtbaren Ereignis- oder Daten-Pins auf der obersten Ebene. Die gesamte Kommunikation wird über die Adapter abgewickelt.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung ist in den Adaptern gekapselt.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Ereignissteuerung ist in den Adaptern gekapselt.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden. Die Datenübertragung erfolgt über die Adapter-Schnittstellen.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden. Die Datenübertragung erfolgt über die Adapter-Schnittstellen.

Adapter

Sockets (Eingangs-Adapter / Steckdosen)

- **IN1** (Typ: `adapter::types::unidirectional::AUDI`):
  - Erster Eingang der Subtraktion (Minuend).
- **IN2** (Typ: `adapter::types::unidirectional::AUDI`):
  - Zweiter Eingang der Subtraktion (Subtrahend).

Plugs (Ausgangs-Adapter / Stecker)

- **OUT** (Typ: `adapter::types::unidirectional::AUDI`):
  - Ausgang des Bausteins, der das Ergebnis der Subtraktion (Differenz) bereitstellt.

Funktionsweise

Der Funktionsbaustein berechnet die mathematische Differenz der beiden über die Eingangs-Adapter bereitgestellten Werte nach folgendem Prinzip:

$$\mathrm{OUT} = \mathrm{IN1} - \mathrm{IN2} \tag{2.89}$$

Sobald sich Werte an den Eingangs-Adaptern `IN1` oder `IN2` ändern und ein entsprechendes Trigger-Ereignis über den Adapter eingeht, wird die Berechnung intern ausgeführt. Das Ergebnis sowie das dazugehörige Update-Ereignis werden anschließend über den Ausgangs-Adapter `OUT` weitergeleitet.

Technische Besonderheiten

- **Generische Implementierung:** Durch das Attribut `eclipse4diac::core::GenericClassName = 'GEN_AUDI_SUB'` ist der Baustein flexibel einsetzbar. Je nach Implementierung des zugrundeliegenden Adapters kann er verschiedene Datentypen unterstützen.
- **Kapselung durch Adapter:** Die Verwendung des unidirektionalen Adapters `AUDI` bündelt Daten- und Eventleitungen. Dies erhöht die Übersichtlichkeit im 4diac-Applikationseditor erheblich, da weniger Linien gezogen werden müssen.

Zustandsübersicht

Der Baustein verhält sich wie ein zustandsloser (stateless) mathematischer Operator. Es gibt keine interne Zustandsschleife (ECC-Zustände im klassischen Sinne), die über längere Zeit gehalten wird. Jede Aktivierung durch ein Ereignis an den Eingängen führt zu einer direkten Berechnung und Aktualisierung des Ausgangs.

Anwendungsszenarien

- **Signalverarbeitung:** Subtraktion von Sensorwerten (z. B. Offset-Kompensation oder Berechnung von Differenzdrücken/-temperaturen) in Systemen, die konsequent auf der `AUDI`-Adapterarchitektur aufbauen.
- **Regelungstechnik:** Berechnung der Regeldifferenz ($e = w - x$) durch Subtraktion des Istwerts vom Sollwert.

Vergleich mit ähnlichen Bausteinen

- **Standard SUB-Baustein (IEC 61131-3 / IEC 61499):** Ein klassischer `SUB`-Baustein besitzt dedizierte Daten-Eingänge (`IN1`, `IN2`), einen Daten-Ausgang (`OUT`) sowie Event-Eingänge und -Ausgänge (z.B. `REQ` / `CNF`). Der `AUDI_SUB_2_UNGATED` hingegen bündelt diese Signale in Adaptern, was die Wiederverwendbarkeit und Modularität in komplexen Architekturen verbessert.
- **AUDI_ADD_2:** Das Gegenstück für die Addition. Verwendet dieselbe Adapter-Schnittstelle, addiert jedoch die Eingangswerte ($IN1 + IN2$).

- `AUDI_SUB_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

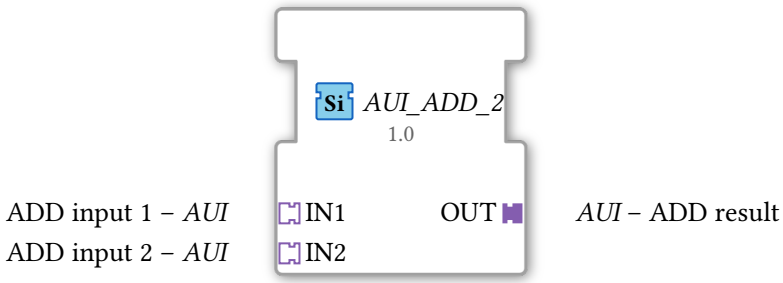
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AUDI_SUB_2_UNGATED` ist ein spezialisierter, aber durch seine generische Natur dennoch flexibler Funktionsbaustein zur Subtraktion. Er eignet sich hervorragend für serviceorientierte Architekturen innerhalb von 4diac, bei denen ein einheitliches Schnittstellenkonzept mittels unidirektionaler Adapter im Vordergrund steht.

AUI_ADD_2



Keine Abbildung verfügbar

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AUI_ADD_2` ist ein generischer Funktionsbaustein (Generic FB), der für die Durchführung einer arithmetischen Addition entwickelt wurde. Er nutzt unidirektionale Adapter vom Typ `AUI` (Adapter Unidirectional Interface), um zwei Eingangswerte zu addieren und das Ergebnis auszugeben. Da es sich um einen generischen Baustein handelt, der auf der Klasse `GEN_AUI_ADD` basiert, kann er flexibel für verschiedene Datentypen eingesetzt werden, die von den entsprechenden Adaptern unterstützt werden.

Schnittstellenstruktur

Da dieser Funktionsbaustein vollständig auf Adaptern basiert, besitzt er keine direkten, diskreten Daten- oder Ereigniskanäle auf der obersten Ebene. Die gesamte Kommunikation und Steuerung wird über die Adapter-Schnittstellen abgewickelt.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden (die Steuerung erfolgt über die angebundenen Adapter).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden (die Weiterleitung erfolgt über die angebundenen Adapter).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

- **Sockets (Eingangs-Adapter):**
  - **IN1** (Typ: `adapter::types::unidirectional::AUI`): Erster Eingang (Summand) für die arithmetische Addition.
  - **IN2** (Typ: `adapter::types::unidirectional::AUI`): Zweiter Eingang (Summand) für die arithmetische Addition.
- **Plugs (Ausgangs-Adapter):**
  - **OUT** (Typ: `adapter::types::unidirectional::AUI`): Ausgang für das Ergebnis der Addition (`IN1 + IN2`).

Funktionsweise

Sobald an den Eingangsadaptern `IN1` und `IN2` Werte anliegen bzw. ein entsprechendes Aktualisierungsereignis über die Adapter getriggert wird, führt der Baustein die Addition aus. Die mathematische Operation lautet:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} \tag{2.90}$$

Das Ergebnis sowie das zugehörige Ausgangsereignis werden unmittelbar über den Ausgangsadapter `OUT` an die nachfolgenden Bausteine weitergegeben.

Durch die Definition als generischer Baustein (`GEN_AUI_ADD`) passt sich die interne Additionslogik dynamisch an die Datentypen der verbundenen Adapter an (z. B. Ganzzahl- oder Gleitkomma-Typen).

Technische Besonderheiten

- **Generisches Design:** Der Baustein ist über das Attribut `eclipse4diac::core::GenericClassName` als `GEN_AUI_ADD` klassifiziert. Dies ermöglicht eine hohe Wiederverwendbarkeit für unterschiedliche numerische Datentypen.
- **Kapselung durch Adapter:** Die Verwendung des unidirektionalen Adapters `AUI` sorgt für ein sehr aufgeräumtes Design im 4diac-Application-Editor, da Ereignis- und Datenflüsse in einer einzigen Verbindung gebündelt sind.
- **Unidirektionaler Fluss:** Die Schnittstellen sind als rein unidirektionale Verbindungen konzipiert, was eine klare Richtung des Signal- und Datenflusses von den Sockets (`IN1`, `IN2`) zum Plug (`OUT`) vorgibt.

Zustandsübersicht

Der Baustein besitzt keinen komplexen internen Zustandsautomaten (ECC). Die Verarbeitung erfolgt ereignisgesteuert und zustandslos (kombinatorisch) basierend auf den über die Adapter eingehenden Triggern.

Anwendungsszenarien

- **Signal-Addition:** Zusammenführen von zwei Sensorwerten (z. B. zur Berechnung eines Gesamtvolumenstroms aus zwei einzelnen Durchflussmessern).
- **Sollwert-Offset:** Aufschalten eines konstanten oder variablen Korrekturwertes (Offsets) auf einen bestehenden Steuerungsparameter.
- **Strukturierte Signalverarbeitung:** Einsatz in komplexeren Regelkreisen innerhalb von IEC 61499, bei denen ein sauberer, adapterbasierter Verdrahtungsstil bevorzugt wird.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-Additionsbausteinen der IEC 61131-3 (wie z. B. einem klassischen `ADD`-Baustein mit diskreten Eingängen) entfällt beim `AUI_ADD_2` das separate Verdrahten von REQ/CNF-Ereignissen und den eigentlichen Datenleitungen. Dies reduziert die Fehleranfälligkeit bei der manuellen Erstellung von Event-Daten-Assoziationen (With-Verbindungen).

- `AUI_ADD_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

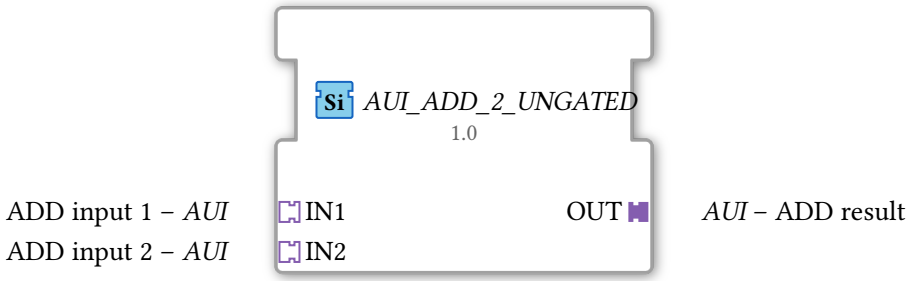
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `AUI_ADD_2` ist ein hocheffizienter, modularer Baustein für die mathematische Addition in modernen IEC 61499-Systemen. Durch den konsequenten Einsatz von Adaptern fördert er ein übersichtliches, fehlerresistentes und gut wartbares Anwendungsdesign in der 4diac-IDE.

AUI_ADD_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUI_ADD_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



Keine Abbildung verfügbar

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AUI_ADD_2_UNGATED` ist ein generischer Funktionsbaustein (Generic FB), der für die Durchführung einer arithmetischen Addition entwickelt wurde. Er nutzt unidirektionale Adapter vom Typ `AUI` (Adapter Unidirectional Interface), um zwei Eingangswerte zu addieren und das Ergebnis auszugeben. Da es sich um einen generischen Baustein handelt, der auf der Klasse `GEN_AUI_ADD` basiert, kann er flexibel für verschiedene Datentypen eingesetzt werden, die von den entsprechenden Adaptern unterstützt werden.

Schnittstellenstruktur

Da dieser Funktionsbaustein vollständig auf Adaptern basiert, besitzt er keine direkten, diskreten Daten- oder Ereigniskanäle auf der obersten Ebene. Die gesamte Kommunikation und Steuerung wird über die Adapter-Schnittstellen abgewickelt.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden (die Steuerung erfolgt über die angebundenen Adapter).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden (die Weiterleitung erfolgt über die angebundenen Adapter).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

- **Sockets (Eingangs-Adapter):**
  - **IN1** (Typ: `adapter::types::unidirectional::AUI`): Erster Eingang (Summand) für die arithmetische Addition.
  - **IN2** (Typ: `adapter::types::unidirectional::AUI`): Zweiter Eingang (Summand) für die arithmetische Addition.
- **Plugs (Ausgangs-Adapter):**
  - **OUT** (Typ: `adapter::types::unidirectional::AUI`): Ausgang für das Ergebnis der Addition (`IN1 + IN2`).

Funktionsweise

Sobald an den Eingangsadaptern `IN1` und `IN2` Werte anliegen bzw. ein entsprechendes Aktualisierungsereignis über die Adapter getriggert wird, führt der Baustein die Addition aus. Die mathematische Operation lautet:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} \tag{2.91}$$

Das Ergebnis sowie das zugehörige Ausgangsereignis werden unmittelbar über den Ausgangsadapter `OUT` an die nachfolgenden Bausteine weitergegeben.

Durch die Definition als generischer Baustein (`GEN_AUI_ADD`) passt sich die interne Additionslogik dynamisch an die Datentypen der verbundenen Adapter an (z. B. Ganzzahl- oder Gleitkomma-Typen).

Technische Besonderheiten

- **Generisches Design:** Der Baustein ist über das Attribut `eclipse4diac::core::GenericClassName` als `GEN_AUI_ADD` klassifiziert. Dies ermöglicht eine hohe Wiederverwendbarkeit für unterschiedliche numerische Datentypen.
- **Kapselung durch Adapter:** Die Verwendung des unidirektionalen Adapters `AUI` sorgt für ein sehr aufgeräumtes Design im 4diac-Application-Editor, da Ereignis- und Datenflüsse in einer einzigen Verbindung gebündelt sind.
- **Unidirektionaler Fluss:** Die Schnittstellen sind als rein unidirektionale Verbindungen konzipiert, was eine klare Richtung des Signal- und Datenflusses von den Sockets (`IN1`, `IN2`) zum Plug (`OUT`) vorgibt.

Zustandsübersicht

Der Baustein besitzt keinen komplexen internen Zustandsautomaten (ECC). Die Verarbeitung erfolgt ereignisgesteuert und zustandslos (kombinatorisch) basierend auf den über die Adapter eingehenden Triggern.

Anwendungsszenarien

- **Signal-Addition:** Zusammenführen von zwei Sensorwerten (z. B. zur Berechnung eines Gesamtvolumenstroms aus zwei einzelnen Durchflussmessern).
- **Sollwert-Offset:** Aufschalten eines konstanten oder variablen Korrekturwertes (Offsets) auf einen bestehenden Steuerungsparameter.
- **Strukturierte Signalverarbeitung:** Einsatz in komplexeren Regelkreisen innerhalb von IEC 61499, bei denen ein sauberer, adapterbasierter Verdrahtungsstil bevorzugt wird.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-Additionsbausteinen der IEC 61131-3 (wie z. B. einem klassischen `ADD`-Baustein mit diskreten Eingängen) entfällt beim `AUI_ADD_2_UNGATED` das separate Verdrahten von REQ/CNF-Ereignissen und den eigentlichen Datenleitungen. Dies reduziert die Fehleranfälligkeit bei der manuellen Erstellung von Event-Daten-Assoziationen (With-Verbindungen).

- **`AUI_ADD_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

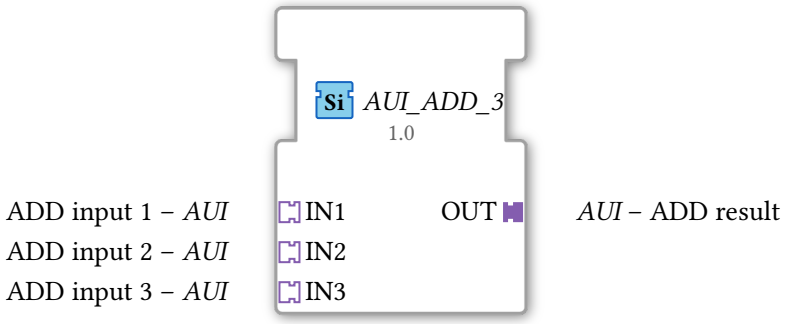
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AUI_ADD_2_UNGATED` ist ein hocheffizienter, modularer Baustein für die mathematische Addition in modernen IEC 61499-Systemen. Durch den konsequenten Einsatz von Adaptern fördert er ein übersichtliches, fehlerresistentes und gut wartbares Anwendungsdesign in der 4diac-IDE.

AUI_ADD_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AUI_ADD_3` ist ein generischer arithmetischer Funktionsbaustein für die IEC 61499 Entwicklungsumgebung (4diac-ide). Seine primäre Aufgabe ist die Addition von drei Eingangswerten, die über standardisierte, unidirektionale Adapterschnittstellen übergeben werden. Das Ergebnis der Addition wird ebenfalls über einen entsprechenden Adapter ausgegeben. Durch die Kapselung von Daten und Steuerungsereignissen in Adaptern wird die Verdrahtung im Applikationseditor erheblich vereinfacht und die Übersichtlichkeit im Systemdesign verbessert.

Schnittstellenstruktur

Die Schnittstellen von `AUI_ADD_3` basieren vollständig auf Adaptern, um eine saubere Strukturierung der Signale zu gewährleisten.

Ereignis-Eingänge

Es sind keine direkten Ereignis-Eingänge definiert. Die Ereignissteuerung wird implizit über die Eingangs-Adapter (Sockets) abgewickelt.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge definiert. Die Ereignisweiterleitung erfolgt implizit über den Ausgangs-Adapter (Plug).

Daten-Eingänge

Es sind keine direkten Daten-Eingänge definiert. Die Daten werden über die Eingangs-Adapter bereitgestellt.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge definiert. Das Ergebnis wird über den Ausgangs-Adapter bereitgestellt.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AUI`): Erster Eingangswert (Summand 1) für die arithmetische Addition.
- **IN2** (Typ: `adapter::types::unidirectional::AUI`): Zweiter Eingangswert (Summand 2) für die arithmetische Addition.
- **IN3** (Typ: `adapter::types::unidirectional::AUI`): Dritter Eingangswert (Summand 3) für die arithmetische Addition.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AUI`): Ausgang für das berechnete Ergebnis (Summe aus `IN1 + IN2 + IN3`).

Funktionsweise

Der Funktionsbaustein realisiert eine dreistellige Addition nach folgender mathematischer Formel:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} \tag{2.92}$$

Sobald an einem der Eingangs-Adapter (`IN1`, `IN2` oder `IN3`) ein Aktualisierungsereignis eintrifft, liest der Funktionsbaustein die aktuellen Werte aller drei Eingänge ein. Anschließend berechnet er die Summe und stellt das Ergebnis am Ausgangs-Adapter `OUT` bereit. Gleichzeitig wird das zugehörige Aktualisierungsereignis über den `OUT`-Adapter weitergegeben, um nachfolgende Bausteine über den neuen Wert zu informieren.

Da es sich um einen generischen Baustein (`GEN_AUI_ADD`) handelt, richtet sich der unterstützte Datentyp (z. B. `INT`, `REAL`, `LREAL`) nach der konkreten Implementierung und Definition des verwendeten Adaptertyps `AUI`.

Technische Besonderheiten

- **Generischer Baustein:** Durch das Attribut `eclipse4diac::core::GenericClassName = 'GEN_AUI_ADD'` ist der Baustein für verschiedene numerische Datentypen flexibel einsetzbar, sofern die verwendeten Adapter dies unterstützen.
- **Kompaktes Design:** Durch die Verwendung von Adaptern anstelle von separaten Event- und Daten-Ports wird das visuelle „Spaghetticode"-Problem in 4diac-Systemen minimiert.
- **Unidirektionale Kommunikation:** Die Verwendung von unidirektionalen Adaptern (`unidirectional::AUI`) stellt sicher, dass der Datenfluss klar definiert von den Eingängen zum Ausgang verläuft.

Zustandsübersicht

Der Baustein arbeitet ereignisgesteuert und zustandslos (bzw. rein reaktiv):

1. **Wartezustand:** Der Baustein wartet auf ein Trigger-Ereignis an einem der Sockets (`IN1`, `IN2`, `IN3`).
2. **Berechnung:** Bei Ereigniseingang werden die Datenwerte ausgelesen und addiert.
3. **Ausgabe:** Der berechnete Wert wird an den Plug `OUT` geschrieben und das Trigger-Ereignis wird gefeuert.
4. **Rückkehr:** Der Baustein kehrt sofort in den Wartezustand zurück.

Anwendungsszenarien

- **Sensorwert-Aggregation:** Addition von drei analogen Messwerten (z. B. drei Temperatursensoren zur Ermittlung einer Gesamtwärme oder drei Durchflussmesser).
- **Sollwert-Generierung:** Zusammenführung von Basis-Sollwert, Hand-Offset und Automatik-Korrekturwert in der Prozesstechnik.
- **Leistungsberechnung:** Summierung der Wirkleistungen von drei einzelnen Phasen eines elektrischen Netzes zu einer Gesamtleistung.

Vergleich mit ähnlichen Bausteinen

- **Standard-ADD-Baustein (IEC 61131-3):** Ein klassischer `ADD`-Baustein besitzt diskrete Eingänge für Daten und Events. `AUI_ADD_3` bündelt diese logisch in Adaptern, was die Wiederverwendbarkeit und Übersichtlichkeit steigert.
- **AUI_ADD_2 (2-fach Addierer):** Während bei einem 2-fach Addierer zur Summierung von drei Werten zwei Bausteine kaskadiert werden müssten, erledigt `AUI_ADD_3` dies in einem einzigen Schritt. Dies spart Ressourcen und reduziert die Latenzzeit im System.

- `AUI_ADD_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

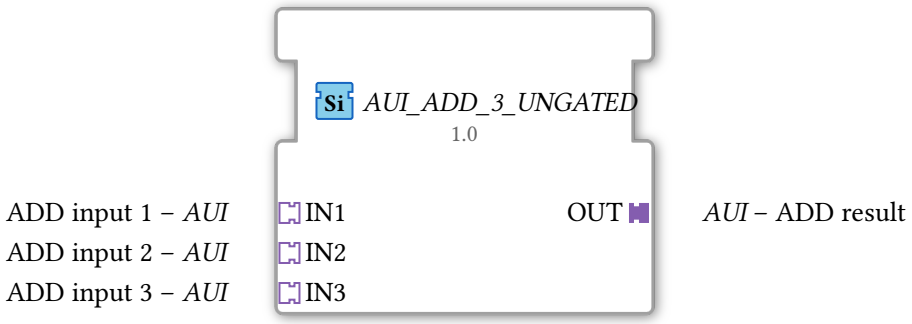
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `AUI_ADD_3` ist ein effizienter Hilfsbaustein für arithmetische Operationen in modernen IEC 61499 Steuerungsanwendungen. Durch die konsequente Nutzung des Adapter-Konzepts fügt er sich nahtlos in serviceorientierte und modular aufgebaute Software-Architekturen ein und trägt zur Reduzierung der Komplexität in der grafischen Programmierung bei.

AUI_ADD_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUI_ADD_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AUI_ADD_3_UNGATED` ist ein generischer arithmetischer Funktionsbaustein für die IEC 61499 Entwicklungsumgebung (4diac-ide). Seine primäre Aufgabe ist die Addition von drei Eingangswerten, die über standardisierte, unidirektionale Adapterschnittstellen übergeben werden. Das Ergebnis der Addition wird ebenfalls über einen entsprechenden Adapter ausgegeben. Durch die Kapselung von Daten und Steuerungsereignissen in Adaptern wird die Verdrahtung im Applikationseditor erheblich vereinfacht und die Übersichtlichkeit im Systemdesign verbessert.

Schnittstellenstruktur

Die Schnittstellen von `AUI_ADD_3_UNGATED` basieren vollständig auf Adaptern, um eine saubere Strukturierung der Signale zu gewährleisten.

Ereignis-Eingänge

Es sind keine direkten Ereignis-Eingänge definiert. Die Ereignissteuerung wird implizit über die Eingangs-Adapter (Sockets) abgewickelt.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge definiert. Die Ereignisweiterleitung erfolgt implizit über den Ausgangs-Adapter (Plug).

Daten-Eingänge

Es sind keine direkten Daten-Eingänge definiert. Die Daten werden über die Eingangs-Adapter bereitgestellt.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge definiert. Das Ergebnis wird über den Ausgangs-Adapter bereitgestellt.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AUI`): Erster Eingangswert (Summand 1) für die arithmetische Addition.
- **IN2** (Typ: `adapter::types::unidirectional::AUI`): Zweiter Eingangswert (Summand 2) für die arithmetische Addition.
- **IN3** (Typ: `adapter::types::unidirectional::AUI`): Dritter Eingangswert (Summand 3) für die arithmetische Addition.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AUI`): Ausgang für das berechnete Ergebnis (Summe aus `IN1 + IN2 + IN3`).

Funktionsweise

Der Funktionsbaustein realisiert eine dreistellige Addition nach folgender mathematischer Formel:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} \tag{2.93}$$

Sobald an einem der Eingangs-Adapter (`IN1`, `IN2` oder `IN3`) ein Aktualisierungsereignis eintrifft, liest der Funktionsbaustein die aktuellen Werte aller drei Eingänge ein. Anschließend berechnet er die Summe und stellt das Ergebnis am Ausgangs-Adapter `OUT` bereit. Gleichzeitig wird das zugehörige Aktualisierungsereignis über den `OUT`-Adapter weitergegeben, um nachfolgende Bausteine über den neuen Wert zu informieren.

Da es sich um einen generischen Baustein (`GEN_AUI_ADD`) handelt, richtet sich der unterstützte Datentyp (z. B. `INT`, `REAL`, `LREAL`) nach der konkreten Implementierung und Definition des verwendeten Adaptertyps `AUI`.

Technische Besonderheiten

- **Generischer Baustein:** Durch das Attribut `eclipse4diac::core::GenericClassName = 'GEN_AUI_ADD'` ist der Baustein für verschiedene numerische Datentypen flexibel einsetzbar, sofern die verwendeten Adapter dies unterstützen.
- **Kompaktes Design:** Durch die Verwendung von Adaptern anstelle von separaten Event- und Daten-Ports wird das visuelle „Spaghetticode"-Problem in 4diac-Systemen minimiert.
- **Unidirektionale Kommunikation:** Die Verwendung von unidirektionalen Adaptern (`unidirectional::AUI`) stellt sicher, dass der Datenfluss klar definiert von den Eingängen zum Ausgang verläuft.

Zustandsübersicht

Der Baustein arbeitet ereignisgesteuert und zustandslos (bzw. rein reaktiv):

1. **Wartezustand:** Der Baustein wartet auf ein Trigger-Ereignis an einem der Sockets (`IN1`, `IN2`, `IN3`).
2. **Berechnung:** Bei Ereigniseingang werden die Datenwerte ausgelesen und addiert.
3. **Ausgabe:** Der berechnete Wert wird an den Plug `OUT` geschrieben und das Trigger-Ereignis wird gefeuert.
4. **Rückkehr:** Der Baustein kehrt sofort in den Wartezustand zurück.

Anwendungsszenarien

- **Sensorwert-Aggregation:** Addition von drei analogen Messwerten (z. B. drei Temperatursensoren zur Ermittlung einer Gesamtwärme oder drei Durchflussmesser).
- **Sollwert-Generierung:** Zusammenführung von Basis-Sollwert, Hand-Offset und Automatik-Korrekturwert in der Prozesstechnik.

- **Leistungsberechnung:** Summierung der Wirkleistungen von drei einzelnen Phasen eines elektrischen Netzes zu einer Gesamtleistung.

Vergleich mit ähnlichen Bausteinen

- **Standard-ADD-Baustein (IEC 61131-3):** Ein klassischer `ADD`-Baustein besitzt diskrete Eingänge für Daten und Events. `AUI_ADD_3_UNGATED` bündelt diese logisch in Adaptern, was die Wiederverwendbarkeit und Übersichtlichkeit steigert.
- **AUI_ADD_2 (2-fach Addierer):** Während bei einem 2-fach Addierer zur Summierung von drei Werten zwei Bausteine kaskadiert werden müssten, erledigt `AUI_ADD_3_UNGATED` dies in einem einzigen Schritt. Dies spart Ressourcen und reduziert die Latenzzeit im System.

- `AUI_ADD_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

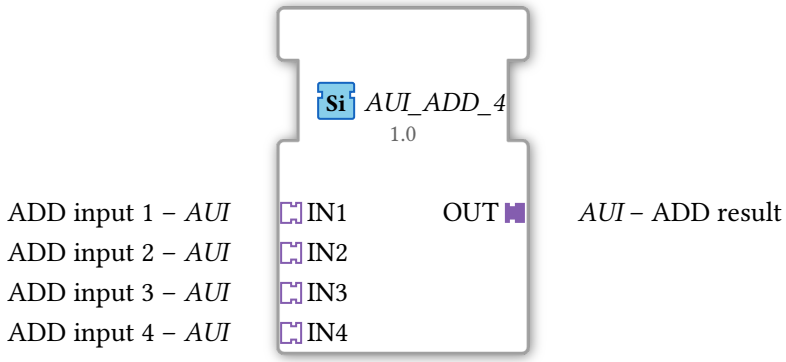
Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AUI_ADD_3_UNGATED` ist ein effizienter Hilfsbaustein für arithmetische Operationen in modernen IEC 61499 Steuerungsanwendungen. Durch die konsequente Nutzung des Adapter-Konzepts fügt er sich nahtlos in serviceorientierte und modular aufgebaute Software-Architekturen ein und trägt zur Reduzierung der Komplexität in der grafischen Programmierung bei.

AUI_ADD_4

```
        +-----------------------------------+
        |             AUI_ADD_4             |
        +-----------------------------------+
==#==>  | [Socket] IN1          [Plug] OUT  | ==#==>
==#==>  | [Socket] IN2                      |
==#==>  | [Socket] IN3                      |
==#==>  | [Socket] IN4                      |
        +-----------------------------------+
```



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AUI_ADD_4` ist ein generischer Arithmetik-Baustein für die Entwicklungsumgebung 4diac-IDE. Er dient zur Addition von bis zu vier numerischen Eingangswerten, die über standardisierte, unidirektionale Adapter vom Typ `AUI` (Analog Universal Interface) bereitgestellt werden. Das berechnete Summengebnis wird über einen entsprechenden Ausgangs-Adapter ausgegeben.

Durch die Verwendung von Adaptern wird die Anzahl der sichtbaren Verbindungen im Steuerungsdiagramm minimiert, was zu einer übersichtlicheren und modulareren Applikationsarchitektur führt.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine direkten Ereignis-Eingänge. Die Ereignissteuerung erfolgt implizit über die integrierten Ereigniskanäle der Eingangs-Adapter (`IN1` bis `IN4`).

Ereignis-Ausgänge

Der Baustein besitzt keine direkten Ereignis-Ausgänge. Die Weiterleitung von Aktualisierungsereignissen erfolgt implizit über den Ausgangs-Adapter (`OUT`).

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübergabe ist vollständig in den Eingangs-Adaptern gekapselt.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Die Datenausgabe ist im Ausgangs-Adapter gekapselt.

Adapter

Sockets (Eingangs-Schnittstellen)

- **IN1** (Typ: `adapter::types::unidirectional::AUI`): Erster Summand der Additionsoperation.
- **IN2** (Typ: `adapter::types::unidirectional::AUI`): Zweiter Summand der Additionsoperation.
- **IN3** (Typ: `adapter::types::unidirectional::AUI`): Dritter Summand der Additionsoperation.
- **IN4** (Typ: `adapter::types::unidirectional::AUI`): Vierter Summand der Additionsoperation.

Plugs (Ausgangs-Schnittstellen)

- **OUT** (Typ: `adapter::types::unidirectional::AUI`): Ausgangs-Adapter, der das mathematische Ergebnis der Addition bereitstellt.

Funktionsweise

Der Funktionsbaustein arbeitet nach den Prinzipien der IEC 61499 für daten- und ereignisgesteuerte Ausführung:

1. **Triggerung:** Sobald an einem der Eingangs-Adapter (`IN1` bis `IN4`) ein neues Datenereignis signalisiert wird, wird der interne Berechnungsalgorithmus aufgerufen.
2. **Berechnung:** Der Baustein liest die numerischen Werte aus allen vier angeschlossenen Adaptern aus und addiert diese nach folgender mathematischer Formel:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} + \mathrm{IN4} \tag{2.94}$$

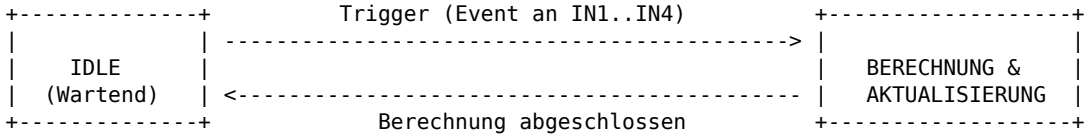
3. **Ausgabe:** Der berechnete Summenwert wird in den Ausgangs-Adapter `OUT` geschrieben, und es wird ein Aktualisierungsereignis über diesen Adapter getriggert, um nachfolgende Bausteine über den neuen Wert zu informieren.

Technische Besonderheiten

- **Generischer Typ (`GEN_AUI_ADD`):** Dank der generischen Definition ist der Baustein flexibel einsetzbar. Die genaue Datentyp-Auflösung (z. B. `INT`, `REAL`, `LREAL`) erfolgt dynamisch zur Laufzeit oder während der Instanziierung in der 4diac-IDE, passend zur Konfiguration des `AUI`-Adapters.
- **Kapselung durch Adapter:** Durch die Verwendung von Adaptern anstelle von einzelnen Event-/Data-Pins wird die visuelle Komplexität im Funktionsplan drastisch reduziert („Clean Design“).
- **Unidirektionaler Datenfluss:** Die Nutzung von unidirektionalen Adaptern stellt sicher, dass Daten und Ereignisse streng gerichtet von den Eingängen zum Ausgang fließen, was unbeabsichtigte Rückkopplungsschleifen verhindert.

Zustandsübersicht

Der Baustein besitzt eine einfache, ereignisgesteuerte Ausführungslogik (Execution Control Chart - ECC):



1. **IDLE (Standardzustand):** Der Baustein wartet passiv auf eingehende Datenänderungen an den Sockets.
2. **BERECHNUNG:** Ein Event an einem der Eingänge stößt die Summation an. Das Ergebnis wird berechnet, an `OUT` übergeben und ein Ausgangs-Event erzeugt. Anschließend kehrt der Baustein sofort in den Zustand `IDLE` zurück.

Anwendungsszenarien

- **Sensordaten-Fusion:** Aufsummierung von bis zu vier analogen Messwerten (z. B. Erfassung des Gesamtvolumenstroms aus vier einzelnen Teilströmen).
- **Leistungsberechnung:** Addition der Wirkleistungen von vier separaten Verbrauchern zur Ermittlung der Gesamtleistung in einem Energiemonitorsystem.
- **Sollwert-Generierung:** Zusammenführung eines Basis-Sollwerts mit bis zu drei Korrektur- oder Offsetwerten in Regelungssystemen.

Vergleich mit ähnlichen Bausteinen

- **Vergleich mit Standard-`ADD` (IEC 61131-3):** Klassische `ADD`-Bausteine arbeiten mit expliziten Daten- und Event-Pins. `AUI_ADD_4` bündelt diese Signale in Adaptern, was die Verkabelung vereinfacht und standardisiert.
- **Vergleich mit `AUI_ADD_2`:** Für die Addition von drei oder vier Werten müssten bei einem 2-fach-Addierer mehrere Bausteine kaskadiert werden. `AUI_ADD_4` spart hierdurch Systemressourcen, verringert die Latenzzeit und sorgt für ein saubereres Applikationslayout.
- `AUI_ADD_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

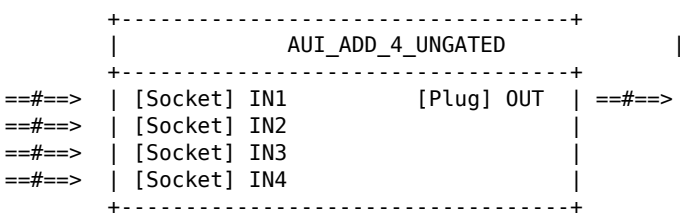
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

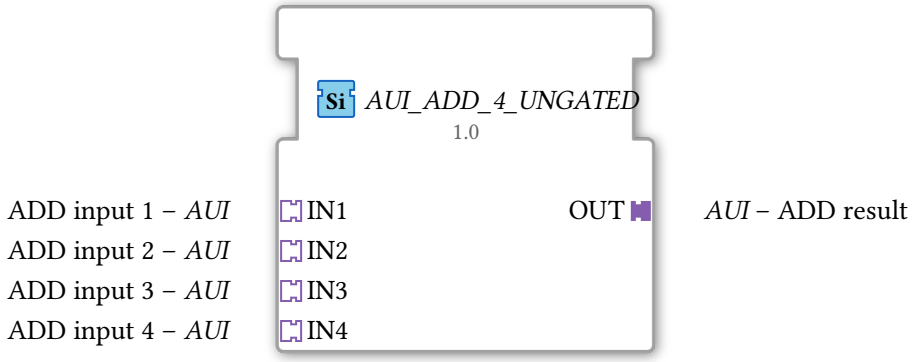
Fazit

Der `AUI_ADD_4` ist ein praktischer Hilfsbaustein für fortgeschrittene Steuerungsprojekte in der 4diac-IDE. Er kombiniert die mathematische Grundfunktion der Addition mit den strukturellen Vorteilen moderner Adapter-Verbindungen und eignet sich ideal für saubere, übersichtliche Signalverarbeitungsketten.

AUI_ADD_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUI_ADD_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AUI_ADD_4_UNGATED` ist ein generischer Arithmetik-Baustein für die Entwicklungsumgebung 4diac-IDE. Er dient zur Addition von bis zu vier numerischen Eingangswerten, die über standardisierte, unidirektionale Adapter vom Typ `AUI` (Analog Universal Interface) bereitgestellt werden. Das berechnete Summengebnis wird über einen entsprechenden Ausgangs-Adapter ausgegeben.

Durch die Verwendung von Adaptern wird die Anzahl der sichtbaren Verbindungen im Steuerungsdiagramm minimiert, was zu einer übersichtlicheren und modulareren Applikationsarchitektur führt.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine direkten Ereignis-Eingänge. Die Ereignissteuerung erfolgt implizit über die integrierten Ereigniskanäle der Eingangs-Adapter (`IN1` bis `IN4`).

Ereignis-Ausgänge

Der Baustein besitzt keine direkten Ereignis-Ausgänge. Die Weiterleitung von Aktualisierungsereignissen erfolgt implizit über den Ausgangs-Adapter (`OUT`).

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübergabe ist vollständig in den Eingangs-Adaptern gekapselt.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Die Datenausgabe ist im Ausgangs-Adapter gekapselt.

Adapter

Sockets (Eingangs-Schnittstellen)

- **IN1** (Typ: `adapter::types::unidirectional::AUI`): Erster Summand der Additionsoperation.
- **IN2** (Typ: `adapter::types::unidirectional::AUI`): Zweiter Summand der Additionsoperation.
- **IN3** (Typ: `adapter::types::unidirectional::AUI`): Dritter Summand der Additionsoperation.
- **IN4** (Typ: `adapter::types::unidirectional::AUI`): Vierter Summand der Additionsoperation.

Plugs (Ausgangs-Schnittstellen)

- **OUT** (Typ: `adapter::types::unidirectional::AUI`): Ausgangs-Adapter, der das mathematische Ergebnis der Addition bereitstellt.

Funktionsweise

Der Funktionsbaustein arbeitet nach den Prinzipien der IEC 61499 für daten- und ereignisgesteuerte Ausführung:

1. **Triggerung:** Sobald an einem der Eingangs-Adapter (`IN1` bis `IN4`) ein neues Datenereignis signalisiert wird, wird der interne Berechnungsalgorithmus aufgerufen.
2. **Berechnung:** Der Baustein liest die numerischen Werte aus allen vier angeschlossenen Adaptern aus und addiert diese nach folgender mathematischer Formel:

$$\text{OUT} = \text{IN1} + \text{IN2} + \text{IN3} + \text{IN4} \tag{2.95}$$

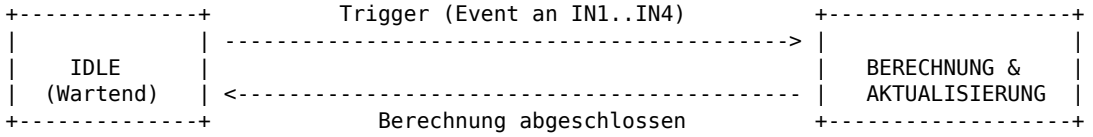
3. **Ausgabe:** Der berechnete Summenwert wird in den Ausgangs-Adapter `OUT` geschrieben, und es wird ein Aktualisierungsereignis über diesen Adapter getriggert, um nachfolgende Bausteine über den neuen Wert zu informieren.

Technische Besonderheiten

- **Generischer Typ (`GEN_AUI_ADD`):** Dank der generischen Definition ist der Baustein flexibel einsetzbar. Die genaue Datentyp-Auflösung (z. B. `INT`, `REAL`, `LREAL`) erfolgt dynamisch zur Laufzeit oder während der Instanziierung in der 4diac-IDE, passend zur Konfiguration des `AUI`-Adapters.
- **Kapselung durch Adapter:** Durch die Verwendung von Adaptern anstelle von einzelnen Event-/Data-Pins wird die visuelle Komplexität im Funktionsplan drastisch reduziert („Clean Design“).
- **Unidirektionaler Datenfluss:** Die Nutzung von unidirektionalen Adaptern stellt sicher, dass Daten und Ereignisse streng gerichtet von den Eingängen zum Ausgang fließen, was unbeabsichtigte Rückkopplungsschleifen verhindert.

Zustandsübersicht

Der Baustein besitzt eine einfache, ereignisgesteuerte Ausführungslogik (Execution Control Chart - ECC):



1. **IDLE (Standardzustand):** Der Baustein wartet passiv auf eingehende Datenänderungen an den Sockets.
2. **BERECHNUNG:** Ein Event an einem der Eingänge stößt die Summation an. Das Ergebnis wird berechnet, an `OUT` übergeben und ein Ausgangs-Event erzeugt. Anschließend kehrt der Baustein sofort in den Zustand `IDLE` zurück.

Anwendungsszenarien

- **Sensordaten-Fusion:** Aufsummierung von bis zu vier analogen Messwerten (z. B. Erfassung des Gesamtvolumenstroms aus vier einzelnen Teilströmen).
- **Leistungsberechnung:** Addition der Wirkleistungen von vier separaten Verbrauchern zur Ermittlung der Gesamtleistung in einem Energiemonitorsystem.
- **Sollwert-Generierung:** Zusammenführung eines Basis-Sollwerts mit bis zu drei Korrektur- oder Offsetwerten in Regelungssystemen.

Vergleich mit ähnlichen Bausteinen

- **Vergleich mit Standard-`ADD` (IEC 61131-3):** Klassische `ADD`-Bausteine arbeiten mit expliziten Daten- und Event-Pins. `AUI_ADD_4_UNGATED` bündelt diese Signale in Adaptern, was die Verkabelung vereinfacht und standardisiert.
- **Vergleich mit `AUI_ADD_2`:** Für die Addition von drei oder vier Werten müssten bei einem 2-fach-Addierer mehrere Bausteine kaskadiert werden. `AUI_ADD_4_UNGATED` spart hierdurch Systemressourcen, verringert die Latenzzeit und sorgt für ein saubereres Applikationslayout.
- `AUI_ADD_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

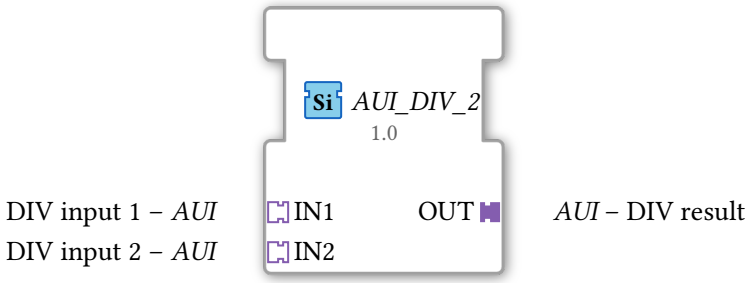
Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AUI_ADD_4_UNGATED` ist ein praktischer Hilfsbaustein für fortgeschrittene Steuerungsprojekte in der 4diac-IDE. Er kombiniert die mathematische Grundfunktion der Addition mit den strukturellen Vorteilen moderner Adapter-Verbindungen und eignet sich ideal für saubere, übersichtliche Signalverarbeitungsketten.

AUI_DIV_2

```
      +----------------------------------+
      |             AUI_DIV_2            |
      +----------------------------------+
=====>| [AUI] IN1                OUT [AUI]|=====>
=====>| [AUI] IN2                        |
      +----------------------------------+
```



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AUI_DIV_2` ist ein generischer Funktionsbaustein zur Durchführung einer arithmetischen Division zweier Eingangswerte. Der Baustein nutzt ein adapterbasiertes Konzept, bei dem die Daten- und Ereignisströme in unidirektionalen Adaptern gebündelt werden. Dies ermöglicht eine saubere, modularisierte und übersichtliche Modellierung innerhalb von IEC 61499 Anwendungen in 4diac.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine direkten Ereignis-Eingänge definiert. Die Ereignissteuerung erfolgt implizit über die Adapter-Schnittstellen.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge definiert. Die Ereignissteuerung erfolgt implizit über die Adapter-Schnittstellen.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge definiert. Die Datenübergabe erfolgt über die Eingangs-Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge definiert. Die Datenübergabe erfolgt über den Ausgangs-Adapter.

Adapter

Sockets (Eingangs-Schnittstellen)

- **IN1** (Typ: `adapter::types::unidirectional::AUI`): Der erste Eingang-Adapter. Er liefert den Dividenden (Zähler) für die Divisionsoperation sowie die zugehörigen Triggereignisse.
- **IN2** (Typ: `adapter::types::unidirectional::AUI`): Der zweite Eingang-Adapter. Er liefert den Divisor (Nenner) für die Divisionsoperation sowie die zugehörigen Triggereignisse.

Plugs (Ausgangs-Schnittstellen)

- **OUT** (Typ: `adapter::types::unidirectional::AUI`): Der Ausgangs-Adapter. Er gibt das Ergebnis der Division (Quotient) und das entsprechende Berechnungsereignis aus.

Funktionsweise

Der Baustein führt die mathematische Operation `OUT = IN1 / IN2` aus.

Sobald an den Eingangs-Adaptern (`IN1` oder `IN2`) ein entsprechendes Trigger-Ereignis registriert wird, liest der Baustein die aktuellen Werte aus den Adaptern aus, berechnet den Quotienten und gibt diesen zusammen mit einem Ausgangsereignis über den Adapter `OUT` weiter.

Da es sich um einen generischen Funktionsbaustein (GEN_AUI_DIV) handelt, kann sich der Baustein je nach Implementierung und Systemkonfiguration an unterschiedliche numerische Datentypen (z. B. INT, REAL, LREAL) anpassen, sofern die zugrundeliegenden Adapter denselben Datentyp unterstützen.

Technische Besonderheiten

- **Generischer Typ:** Durch das Attribut GenericClassName mit dem Wert 'GEN_AUI_DIV' ist der Baustein flexibel einsetzbar und nicht auf einen festen Datentyp fixiert.
- **Schnittstellen-Kapselung:** Die Verwendung von unidirektionalen Adaptern (AUI) reduziert den Verdrahtungsaufwand in der 4diac-IDE erheblich, da Event- und Datenleitungen in einer einzigen Verbindung gebündelt werden.
- **Division durch Null:** Bei der Verwendung dieses Bausteins muss anwendungstechnisch sichergestellt werden, dass der Wert von IN2 ungleich Null ist, um Laufzeitfehler oder unendliche Werte (NaN/Inf) im Steuerungssystem zu vermeiden.

Zustandsübersicht

1. **Wartestatus (IDLE):** Der Baustein wartet auf eingehende Ereignisse über die Adapter IN1 oder IN2.
2. **Berechnung (EVALUATE):** Nach dem Empfang eines Ereignisses werden die Daten der Adapter ausgelesen und die Division durchgeführt.
3. **Ausgabe (DISPATCH):** Das Ergebnis und das verknüpfte Ereignis werden über den Adapter OUT bereitgestellt.

Anwendungsszenarien

- **Skalierung von Sensorwerten:** Division von Rohwerten durch Skalierungsfaktoren in modular aufgebauten Steuerungsarchitekturen.
- **Berechnung von Mittelwerten:** Einsatz in mathematischen Berechnungsnetzwerken, bei denen Datenströme strukturiert über Adapter übertragen werden.
- **Signalverarbeitung:** Reduzierung von Signalstärken oder Frequenzteilungen in verteilten Systemen.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem klassischen Standard-DIV-Baustein (z. B. aus der Standardbibliothek der IEC 61131-3), welcher mit einzelnen Pins für REQ, CNF, IN1, IN2 und OUT arbeitet, bietet der AUI_DIV_2 eine signifikant sauberere visuelle Darstellung im Funktionsplan. Durch die Kapselung in Adaptern wird die Fehleranfälligkeit bei der manuellen Verdrahtung von Event-zu-Daten-Zuordnungen minimiert.

- AUI_DIV_2_UNGATED: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

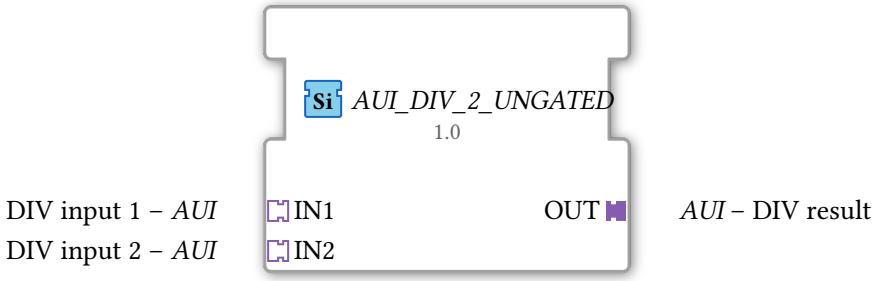
Fazit

Der AUI_DIV_2 ist ein moderner, modularer und wiederverwendbarer Funktionsbaustein zur Division. Durch die konsequente Nutzung von unidirektionalen Adaptern eignet er sich hervorragend für komplexe, verteilte Steuerungsanwendungen in 4diac, bei denen Übersichtlichkeit und einfache Wartbarkeit des Codes im Vordergrund stehen.

AUI_DIV_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AUI_DIV_2. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

```
      +----------------------------------+
      |             AUI_DIV_2_UNGATED            |
      +----------------------------------+
=====>| [AUI] IN1                 OUT [AUI]|=====>
=====>| [AUI] IN2                        |
      +----------------------------------+
```



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AUI_DIV_2_UNGATED` ist ein generischer Funktionsbaustein zur Durchführung einer arithmetischen Division zweier Eingangswerte. Der Baustein nutzt ein adapterbasiertes Konzept, bei dem die Daten- und Ereignisströme in unidirektionalen Adaptern gebündelt werden. Dies ermöglicht eine saubere, modularisierte und übersichtliche Modellierung innerhalb von IEC 61499 Anwendungen in 4diac.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine direkten Ereignis-Eingänge definiert. Die Ereignissteuerung erfolgt implizit über die Adapter-Schnittstellen.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge definiert. Die Ereignissteuerung erfolgt implizit über die Adapter-Schnittstellen.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge definiert. Die Datenübergabe erfolgt über die Eingangs-Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge definiert. Die Datenübergabe erfolgt über den Ausgangs-Adapter.

Adapter

Sockets (Eingangs-Schnittstellen)

- **IN1** (Typ: `adapter::types::unidirectional::AUI`): Der erste Eingang-Adapter. Er liefert den Dividenden (Zähler) für die Divisionsoperation sowie die zugehörigen Triggereignisse.
- **IN2** (Typ: `adapter::types::unidirectional::AUI`): Der zweite Eingang-Adapter. Er liefert den Divisor (Nenner) für die Divisionsoperation sowie die zugehörigen Triggereignisse.

Plugs (Ausgangs-Schnittstellen)

- **OUT** (Typ: `adapter::types::unidirectional::AUI`): Der Ausgangs-Adapter. Er gibt das Ergebnis der Division (Quotient) und das entsprechende Berechnungsereignis aus.

Funktionsweise

Der Baustein führt die mathematische Operation `OUT = IN1 / IN2` aus.

Sobald an den Eingangs-Adaptern (`IN1` oder `IN2`) ein entsprechendes Trigger-Ereignis registriert wird, liest der Baustein die aktuellen Werte aus den Adaptern aus, berechnet den Quotienten und gibt diesen zusammen mit einem Ausgangsereignis über den Adapter `OUT` weiter.

Da es sich um einen generischen Funktionsbaustein (`GEN_AUI_DIV`) handelt, kann sich der Baustein je nach Implementierung und Systemkonfiguration an unterschiedliche numerische Datentypen (z. B. INT, REAL, LREAL) anpassen, sofern die zugrundeliegenden Adapter denselben Datentyp unterstützen.

Technische Besonderheiten

- **Generischer Typ:** Durch das Attribut `GenericClassName` mit dem Wert `'GEN_AUI_DIV'` ist der Baustein flexibel einsetzbar und nicht auf einen festen Datentyp fixiert.
- **Schnittstellen-Kapselung:** Die Verwendung von unidirektionalen Adaptern (`AUI`) reduziert den Verdrahtungsaufwand in der 4diac-IDE erheblich, da Event- und Datenleitungen in einer einzigen Verbindung gebündelt werden.
- **Division durch Null:** Bei der Verwendung dieses Bausteins muss anwendungstechnisch sichergestellt werden, dass der Wert von `IN2` ungleich Null ist, um Laufzeitfehler oder unendliche Werte (NaN/Inf) im Steuerungssystem zu vermeiden.

Zustandsübersicht

1. **Wartestatus (IDLE):** Der Baustein wartet auf eingehende Ereignisse über die Adapter `IN1` oder `IN2`.
2. **Berechnung (EVALUATE):** Nach dem Empfang eines Ereignisses werden die Daten der Adapter ausgelesen und die Division durchgeführt.
3. **Ausgabe (DISPATCH):** Das Ergebnis und das verknüpfte Ereignis werden über den Adapter `OUT` bereitgestellt.

Anwendungsszenarien

- **Skalierung von Sensorwerten:** Division von Rohwerten durch Skalierungsfaktoren in modular aufgebauten Steuerungsarchitekturen.
- **Berechnung von Mittelwerten:** Einsatz in mathematischen Berechnungsnetzwerken, bei denen Datenströme strukturiert über Adapter übertragen werden.
- **Signalverarbeitung:** Reduzierung von Signalstärken oder Frequenzteilungen in verteilten Systemen.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem klassischen Standard-DIV-Baustein (z. B. aus der Standardbibliothek der IEC 61131-3), welcher mit einzelnen Pins für `REQ`, `CNF`, `IN1`, `IN2` und `OUT` arbeitet, bietet der `AUI_DIV_2_UNGATED` eine signifikant sauberere visuelle Darstellung im Funktionsplan. Durch die Kapselung in Adaptern wird die Fehleranfälligkeit bei der manuellen Verdrahtung von Event-zu-Daten-Zuordnungen minimiert.

- `AUI_DIV_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

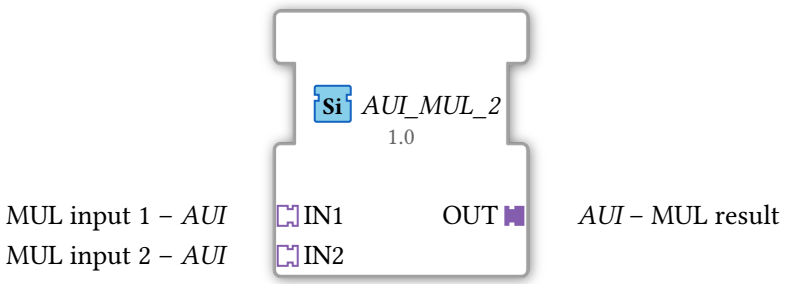
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AUI_DIV_2_UNGATED` ist ein moderner, modularer und wiederverwendbarer Funktionsbaustein zur Division. Durch die konsequente Nutzung von unidirektionalen Adaptern eignet er sich hervorragend für komplexe, verteilte Steuerungsanwendungen in 4diac, bei denen Übersichtlichkeit und einfache Wartbarkeit des Codes im Vordergrund stehen.

AUI_MUL_2



(Kein Bild vorhanden)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AUI_MUL_2` ist ein generischer arithmetischer Funktionsblock für die Entwicklungsumgebung 4diac-ide, der zur Durchführung einer Multiplikation von zwei Eingangswerten dient.

Die Besonderheit dieses Bausteins liegt in seiner adapterbasierten Architektur. Statt klassischer, einzelner Daten- und Ereignisleitungen nutzt dieser Baustein unidirektionale Adapter (`AUI`), um Daten und die dazugehörigen Steuerungsereignisse gekapselt zu übertragen. Dies reduziert die visuelle Komplexität von Steuerungsanwendungen erheblich.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung wird vollständig über die Eingangsadapter abgewickelt.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Ereignisweiterleitung wird vollständig über den Ausgangsadapter abgewickelt.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

- **`IN1` (Typ: `adapter::types::unidirectional::AUI` - Socket / Buchse):** Der erste Eingangsadapter, der den ersten Faktor für die Multiplikation liefert.
- **`IN2` (Typ: `adapter::types::unidirectional::AUI` - Socket / Buchse):** Der zweite Eingangsadapter, der den zweiten Faktor für die Multiplikation liefert.
- **`OUT` (Typ: `adapter::types::unidirectional::AUI` - Plug / Stecker):** Der Ausgangsadapter, der das berechnete Produkt der beiden Eingangswerte sowie das entsprechende Aktualisierungsereignis ausgibt.

Funktionsweise

Sobald an einem der Eingangsadapter (`IN1` oder `IN2`) ein Ereignis signalisiert, dass ein neuer Wert vorliegt, liest der Funktionsblock die aktuellen Werte aus beiden Adaptern aus.

Die Berechnung erfolgt nach der klassischen mathematischen Formel: `OUT = IN1 * IN2`

Nach erfolgreicher Berechnung wird das Ergebnis an den Ausgangsadapter `OUT` übergeben und gleichzeitig ein Ausgangsereignis ausgelöst, um nachfolgende Bausteine über das Vorhandensein eines neuen Werts zu informieren.

Technische Besonderheiten

- **Generisches Verhalten:** Der Baustein ist über das Compiler-Attribut `GenericClassName` mit dem Wert `'GEN_AUI_MUL'` verknüpft. Das bedeutet, dass der Baustein datentypunabhängig (z. B. für `INT`, `REAL`, `LREAL` etc.) agieren kann, je nachdem, welcher konkrete Datentyp durch die angeschlossenen Adapter definiert wird.
- **Effiziente Verdrahtung:** Durch die Kapselung in `AUI`-Adapter (Adapter Unidirectional Interface) wird die Anzahl der Verbindungen im Funktionsplan drastisch reduziert, da Daten- und Ereignisflüsse zusammengefasst sind.

Zustandsübersicht

Der Funktionsbaustein arbeitet rein ereignisgesteuert und zustandslos (reaktiv):

1. **Bereitschaftszustand:** Der Baustein wartet auf eingehende Ereignisse über `IN1` oder `IN2`.
2. **Berechnungsphase:** Ein eingehendes Ereignis triggert die interne Multiplikation der aktuellen Adapterdaten.

3. **Ausgabephase:** Das berechnete Produkt wird an `OUT` angelegt und das Ausgabeereignis getriggert. Der Baustein kehrt sofort in den Bereitschaftszustand zurück.

Anwendungsszenarien

- **Messwertskalierung:** Multiplikation eines analogen Sensorwerts (z. B. Spannung) mit einem Skalierungsfaktor zur Berechnung des physikalischen Istwerts.
- **Leistungsberechnung:** Kontinuierliche Multiplikation von Strom (I) und Spannung (U) zur Ermittlung der aktuellen Wirkleistung (P).
- **Signalverstärkung:** Anwendung im Bereich der Signalverarbeitung zur einfachen Pegelanpassung.

Vergleich mit ähnlichen Bausteinen

- **Standard `MUL` Baustein (IEC 61131-3):** Ein Standard-Multiplikationsbaustein benötigt separate Pins für jedes Datensignal sowie für die Ausführungssteuerung (`REQ`/`CNF`). `AUI_MUL_2` löst dies eleganter und platzsparender über Adapterverbindungen.
- **Multi-Input-Multiplizierer (`MUL_3`, `MUL_4`):** Während Bausteine wie `AUI_MUL_2` fest auf zwei Eingänge ausgelegt sind, erlauben Multi-Input-Bausteine die Multiplikation von drei oder mehr Faktoren in einem einzigen Schritt, weisen dafür jedoch eine höhere Komplexität auf.

- `AUI_MUL_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

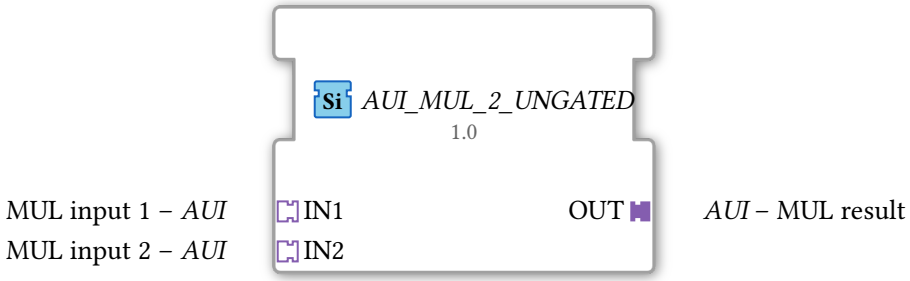
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `AUI_MUL_2` ist ein hochgradig modularer und moderner Funktionsbaustein für die 4diac-ide. Durch die konsequente Nutzung von unidirektionalen Adaptern fördert er ein sauberes Softwaredesign in verteilten Steuerungssystemen nach IEC 61499, ohne dabei an Flexibilität bei den verarbeiteten Datentypen einzubüßen.

AUI_MUL_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUI_MUL_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild vorhanden)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AUI_MUL_2_UNGATED` ist ein generischer arithmetischer Funktionsblock für die Entwicklungsumgebung 4diac-ide, der zur Durchführung einer Multiplikation von zwei Eingangswerten dient.

Die Besonderheit dieses Bausteins liegt in seiner adapterbasierten Architektur. Statt klassischer, einzelner Daten- und Ereignisleitungen nutzt dieser Baustein unidirektionale Adapter (`AUI`), um Daten und die dazugehörigen Steuerungsereignisse gekapselt zu übertragen. Dies reduziert die visuelle Komplexität von Steuerungsanwendungen erheblich.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung wird vollständig über die Eingangsadapter abgewickelt.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Ereignisweiterleitung wird vollständig über den Ausgangsadapter abgewickelt.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

- **`IN1` (Typ: `adapter::types::unidirectional::AUI` - Socket / Buchse):** Der erste Eingangsadapter, der den ersten Faktor für die Multiplikation liefert.
- **`IN2` (Typ: `adapter::types::unidirectional::AUI` - Socket / Buchse):** Der zweite Eingangsadapter, der den zweiten Faktor für die Multiplikation liefert.
- **`OUT` (Typ: `adapter::types::unidirectional::AUI` - Plug / Stecker):** Der Ausgangsadapter, der das berechnete Produkt der beiden Eingangswerte sowie das entsprechende Aktualisierungsereignis ausgibt.

Funktionsweise

Sobald an einem der Eingangsadapter (`IN1` oder `IN2`) ein Ereignis signalisiert, dass ein neuer Wert vorliegt, liest der Funktionsblock die aktuellen Werte aus beiden Adaptern aus.

Die Berechnung erfolgt nach der klassischen mathematischen Formel: `OUT = IN1 * IN2`

Nach erfolgreicher Berechnung wird das Ergebnis an den Ausgangsadapter `OUT` übergeben und gleichzeitig ein Ausgangsereignis ausgelöst, um nachfolgende Bausteine über das Vorhandensein eines neuen Werts zu informieren.

Technische Besonderheiten

- **Generisches Verhalten:** Der Baustein ist über das Compiler-Attribut `GenericClassName` mit dem Wert `'GEN_AUI_MUL'` verknüpft. Das bedeutet, dass der Baustein datentypunabhängig (z. B. für `INT`, `REAL`, `LREAL` etc.) agieren kann, je nachdem, welcher konkrete Datentyp durch die angeschlossenen Adapter definiert wird.
- **Effiziente Verdrahtung:** Durch die Kapselung in `AUI`-Adapter (Adapter Unidirectional Interface) wird die Anzahl der Verbindungen im Funktionsplan drastisch reduziert, da Daten- und Ereignisflüsse zusammengefasst sind.

Zustandsübersicht

Der Funktionsbaustein arbeitet rein ereignisgesteuert und zustandslos (reaktiv):

1. **Bereitschaftszustand:** Der Baustein wartet auf eingehende Ereignisse über `IN1` oder `IN2`.
2. **Berechnungsphase:** Ein eingehendes Ereignis triggert die interne Multiplikation der aktuellen Adapterdaten.

3. **Ausgabephase:** Das berechnete Produkt wird an `OUT` angelegt und das Ausgabeereignis getriggert. Der Baustein kehrt sofort in den Bereitschaftszustand zurück.

Anwendungsszenarien

- **Messwertskalierung:** Multiplikation eines analogen Sensorwerts (z. B. Spannung) mit einem Skalierungsfaktor zur Berechnung des physikalischen Istwerts.
- **Leistungsberechnung:** Kontinuierliche Multiplikation von Strom (I) und Spannung (U) zur Ermittlung der aktuellen Wirkleistung (P).
- **Signalverstärkung:** Anwendung im Bereich der Signalverarbeitung zur einfachen Pegelanpassung.

Vergleich mit ähnlichen Bausteinen

- **Standard `MUL` Baustein (IEC 61131-3):** Ein Standard-Multiplikationsbaustein benötigt separate Pins für jedes Datensignal sowie für die Ausführungssteuerung (`REQ`/`CNF`). `AUI_MUL_2_UNGATED` löst dies eleganter und platzsparender über Adapterverbindungen.
- **Multi-Input-Multiplizierer (`MUL_3`, `MUL_4`):** Während Bausteine wie `AUI_MUL_2_UNGATED` fest auf zwei Eingänge ausgelegt sind, erlauben Multi-Input-Bausteine die Multiplikation von drei oder mehr Faktoren in einem einzigen Schritt, weisen dafür jedoch eine höhere Komplexität auf.

- `AUI_MUL_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

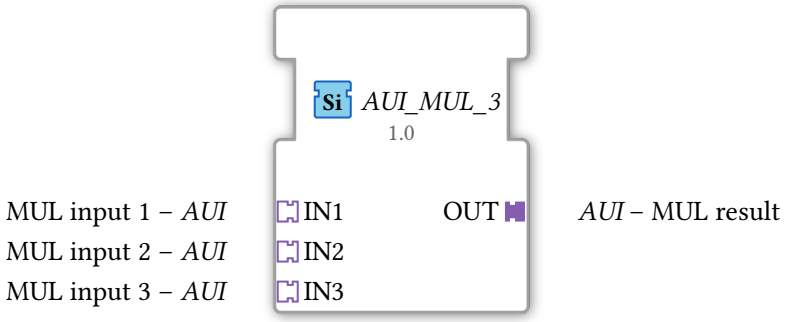
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AUI_MUL_2_UNGATED` ist ein hochgradig modularer und moderner Funktionsbaustein für die 4diac-ide. Durch die konsequente Nutzung von unidirektionalen Adaptern fördert er ein sauberes Softwaredesign in verteilten Steuerungssystemen nach IEC 61499, ohne dabei an Flexibilität bei den verarbeiteten Datentypen einzubüßen.

AUI_MUL_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AUI_MUL_3` dient der mathematischen Multiplikation von drei Eingangswerten. Es handelt sich um einen generischen Funktionsbaustein (Generic FB), der auf der übergeordneten Klasse `GEN_AUI_MUL` basiert. Die Besonderheit dieses Bausteins liegt in der Verwendung von unidirektionalen Adaptern des Typs `AUI` (Adapter Unidirectional Interface), welche sowohl die Daten als auch die dazugehörigen Ereignisse kapseln. Dies sorgt für eine übersichtliche und modulare Strukturierung innerhalb von 4diac-Anwendungen.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten, dedizierten Ereignis-Eingänge. Die Ereignissteuerung wird vollständig über die angeschlossenen Adapter abgewickelt.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten, dedizierten Ereignis-Ausgänge. Die Ereignisweiterleitung erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübergabe erfolgt über die Eingangs-Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Die Datenausgabe erfolgt über den Ausgangs-Adapter.

Adapter

Sockets (Eingangs-Schnittstellen)

- **IN1** (Typ: `adapter::types::unidirectional::AUI`): Erster Multiplikand.
- **IN2** (Typ: `adapter::types::unidirectional::AUI`): Zweiter Multiplikand.
- **IN3** (Typ: `adapter::types::unidirectional::AUI`): Dritter Multiplikand.

Plugs (Ausgangs-Schnittstelle)

- **OUT** (Typ: `adapter::types::unidirectional::AUI`): Ergebnis der Multiplikation ($OUT = IN1 \cdot IN2 \cdot IN3$).

Funktionsweise

Sobald an einem der Eingangs-Adapter (`IN1`, `IN2` oder `IN3`) ein neues Ereignis signalisiert wird, liest der Funktionsbaustein die aktuellen Werte der drei Eingänge aus. Es wird die arithmetische Multiplikation berechnet:

$$\text{Ergebnis} = \text{Wert } (IN1) \cdot \text{Wert } (IN2) \cdot \text{Wert } (IN3) \tag{2.96}$$

Das Ergebnis wird an den Ausgangs-Adapter `OUT` übergeben, und ein entsprechendes Ausgangsereignis wird über diesen Adapter getriggert, um nachfolgende Bausteine über die Aktualisierung zu informieren.

Da der Baustein als generisch (`GEN_AUI_MUL`) deklariert ist, passt er sich flexibel an die Datentypen der verbundenen Adapter an (z. B. `INT`, `REAL`, `LREAL`), solange diese vom genutzten Adaptertyp unterstützt werden.

Technische Besonderheiten

- **Generischer Typ:** Durch die Definition als generischer Baustein kann er flexibel für verschiedene numerische Datentypen eingesetzt werden, ohne dass für jeden Typ ein eigener Baustein erstellt werden muss.
- **Kapselung durch Adapter:** Die Verwendung der unidirektionalen `AUI`-Adapter reduziert die Anzahl der sichtbaren Verbindungslinien im 4diac-Editor signifikant, da Daten und Trigger-Ereignisse in einer einzigen Verbindung gebündelt sind.

Zustandsübersicht

Der Baustein arbeitet rein ereignisgesteuert:

1. **Wartezustand:** Der Baustein wartet auf ein Trigger-Ereignis an einem der Sockets (`IN1`, `IN2`, `IN3`).
2. **Berechnung:** Nach dem Eintreffen eines Ereignisses werden die Daten ausgelesen und multipliziert.
3. **Ausgabe:** Das Ergebnis wird an den Plug `OUT` angelegt, das Sende-Ereignis des Plugs wird ausgelöst, und der Baustein kehrt in den Wartezustand zurück.

Anwendungsszenarien

- **Volumenberechnung:** Berechnung eines dreidimensionalen Volumens (Länge · Breite · Höhe) in Förder- oder Abfüllanlagen.
- **Mehrstufige Skalierung:** Multiplikation eines Sensor-Rohwertes mit einem Kalibrierungsfaktor und einem zusätzlichen Umrechnungsfaktor (z. B. für Maßeinheiten).
- **Leistungsberechnungen:** Berechnung von physikalischen Größen im dreiphasigen System oder unter Einbezug von Wirkungsgraden.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem klassischen, nicht-generischen IEC 61131-3 `MUL`-Standardbaustein bietet der `AUI_MUL_3`:

- **Weniger Kaskadierung:** Standard-Multiplizierer besitzen oft nur zwei Eingänge. Um drei Werte zu multiplizieren, müssten zwei Bausteine kaskadiert werden. `AUI_MUL_3` erledigt dies in einem Schritt.
- **Bessere Übersicht:** Während klassische Bausteine getrennte Event- und Datenleitungen benötigen, vereinfacht das hier genutzte Adapterkonzept das visuelle Applikationsdesign in der 4diac-IDE erheblich.

- `AUI_MUL_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

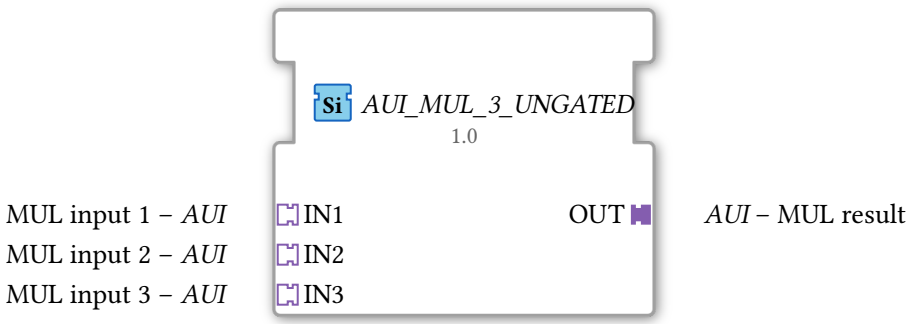
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `AUI_MUL_3` ist ein hocheffizienter und wiederverwendbarer Funktionsbaustein für arithmetische Berechnungen. Durch die Kombination aus generischer Datenverarbeitung und moderner Adapter-Technologie eignet er sich ideal für saubere, übersichtliche und wartbare Steuerungsarchitekturen in IEC 61499-Umgebungen.

AUI_MUL_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUI_MUL_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AUI_MUL_3_UNGATED` dient der mathematischen Multiplikation von drei Eingangswerten. Es handelt sich um einen generischen Funktionsbaustein (Generic FB), der auf der übergeordneten Klasse `GEN_AUI_MUL` basiert. Die Besonderheit dieses Bausteins liegt in der Verwendung von unidirektionalen Adaptern des Typs `AUI` (Adapter Unidirectional Interface), welche sowohl die Daten als auch die dazugehörigen Ereignisse kapseln. Dies sorgt für eine übersichtliche und modulare Strukturierung innerhalb von 4diac-Anwendungen.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten, dedizierten Ereignis-Eingänge. Die Ereignissteuerung wird vollständig über die angeschlossenen Adapter abgewickelt.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten, dedizierten Ereignis-Ausgänge. Die Ereignisweiterleitung erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübergabe erfolgt über die Eingangs-Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Die Datenausgabe erfolgt über den Ausgangs-Adapter.

Adapter

Sockets (Eingangs-Schnittstellen)

- **IN1** (Typ: `adapter::types::unidirectional::AUI`): Erster Multiplikand.
- **IN2** (Typ: `adapter::types::unidirectional::AUI`): Zweiter Multiplikand.
- **IN3** (Typ: `adapter::types::unidirectional::AUI`): Dritter Multiplikand.

Plugs (Ausgangs-Schnittstelle)

- **OUT** (Typ: `adapter::types::unidirectional::AUI`): Ergebnis der Multiplikation ($OUT = IN1 \cdot IN2 \cdot IN3$).

Funktionsweise

Sobald an einem der Eingangs-Adapter (`IN1`, `IN2` oder `IN3`) ein neues Ereignis signalisiert wird, liest der Funktionsbaustein die aktuellen Werte der drei Eingänge aus. Es wird die arithmetische Multiplikation berechnet:

$$\text{Ergebnis} = \text{Wert } (IN1) \cdot \text{Wert } (IN2) \cdot \text{Wert } (IN3) \tag{2.97}$$

Das Ergebnis wird an den Ausgangs-Adapter `OUT` übergeben, und ein entsprechendes Ausgangsereignis wird über diesen Adapter getriggert, um nachfolgende Bausteine über die Aktualisierung zu informieren.

Da der Baustein als generisch (`GEN_AUI_MUL`) deklariert ist, passt er sich flexibel an die Datentypen der verbundenen Adapter an (z. B. `INT`, `REAL`, `LREAL`), solange diese vom genutzten Adaptertyp unterstützt werden.

Technische Besonderheiten

- **Generischer Typ:** Durch die Definition als generischer Baustein kann er flexibel für verschiedene numerische Datentypen eingesetzt werden, ohne dass für jeden Typ ein eigener Baustein erstellt werden muss.
- **Kapselung durch Adapter:** Die Verwendung der unidirektionalen `AUI`-Adapter reduziert die Anzahl der sichtbaren Verbindungslinien im 4diac-Editor signifikant, da Daten und Trigger-Ereignisse in einer einzigen Verbindung gebündelt sind.

Zustandsübersicht

Der Baustein arbeitet rein ereignisgesteuert:

1. **Wartezustand:** Der Baustein wartet auf ein Trigger-Ereignis an einem der Sockets (`IN1`, `IN2`, `IN3`).
2. **Berechnung:** Nach dem Eintreffen eines Ereignisses werden die Daten ausgelesen und multipliziert.
3. **Ausgabe:** Das Ergebnis wird an den Plug `OUT` angelegt, das Sende-Ereignis des Plugs wird ausgelöst, und der Baustein kehrt in den Wartezustand zurück.

Anwendungsszenarien

- **Volumenberechnung:** Berechnung eines dreidimensionalen Volumens (Länge · Breite · Höhe) in Förder- oder Abfüllanlagen.
- **Mehrstufige Skalierung:** Multiplikation eines Sensor-Rohwertes mit einem Kalibrierungsfaktor und einem zusätzlichen Umrechnungsfaktor (z. B. für Maßeinheiten).
- **Leistungsberechnungen:** Berechnung von physikalischen Größen im dreiphasigen System oder unter Einbezug von Wirkungsgraden.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem klassischen, nicht-generischen IEC 61131-3 `MUL`-Standardbaustein bietet der `AUI_MUL_3_UNGATED`:

- **Weniger Kaskadierung:** Standard-Multiplizierer besitzen oft nur zwei Eingänge. Um drei Werte zu multiplizieren, müssten zwei Bausteine kaskadiert werden. `AUI_MUL_3_UNGATED` erledigt dies in einem Schritt.
- **Bessere Übersicht:** Während klassische Bausteine getrennte Event- und Datenleitungen benötigen, vereinfacht das hier genutzte Adapterkonzept das visuelle Applikationsdesign in der 4diac-IDE erheblich.

- `AUI_MUL_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

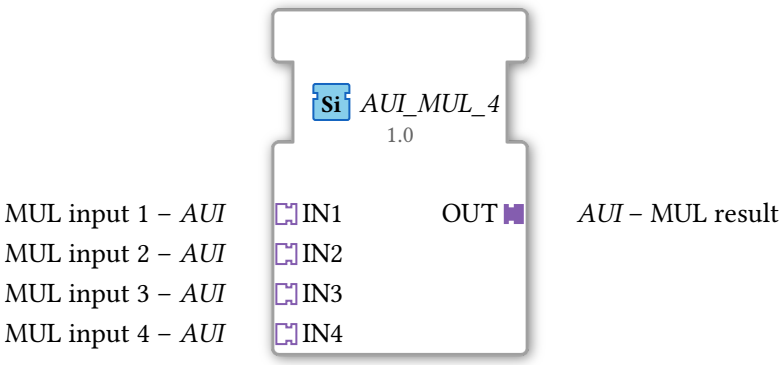
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AUI_MUL_3_UNGATED` ist ein hocheffizienter und wiederverwendbarer Funktionsbaustein für arithmetische Berechnungen. Durch die Kombination aus generischer Datenverarbeitung und moderner Adapter-Technologie eignet er sich ideal für saubere, übersichtliche und wartbare Steuerungsarchitekturen in IEC 61499-Umgebungen.

AUI_MUL_4



Kein Bild verfügbar

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AUI_MUL_4` ist ein generischer Baustein zur Durchführung einer arithmetischen Multiplikation von vier Eingangswerten. Er nutzt unidirektionale Adapter vom Typ `AUI` (Adapter Unit

Interface), um Daten und zugehörige Ereignisse gebündelt zu übertragen. Dies ermöglicht eine saubere und übersichtliche Modellierung in der 4diac-ide.

Schnittstellenstruktur

Ereignis-Eingänge
Es sind keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung wird über die Adapter-Schnittstellen abgewickelt.

Ereignis-Ausgänge
Es sind keine direkten Ereignis-Ausgänge vorhanden. Die Ereignissteuerung wird über die Adapter-Schnittstellen abgewickelt.

Daten-Eingänge
Es sind keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge
Es sind keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingänge)
- **IN1** (Typ: `adapter::types::unidirectional::AUI`): Erster Multiplikand.
- **IN2** (Typ: `adapter::types::unidirectional::AUI`): Zweiter Multiplikand.
- **IN3** (Typ: `adapter::types::unidirectional::AUI`): Dritter Multiplikand.
- **IN4** (Typ: `adapter::types::unidirectional::AUI`): Vierter Multiplikand.

Plugs (Ausgänge)
- **OUT** (Typ: `adapter::types::unidirectional::AUI`): Ausgang für das Ergebnis der Multiplikation ($OUT = IN1 \cdot IN2 \cdot IN3 \cdot IN4$).

Funktionsweise
Sobald an einem der Eingangs-Adapter (`IN1` bis `IN4`) ein neues Ereignis signalisiert wird, liest der Funktionsbaustein die aktuellen Werte der vier Eingänge aus. Diese Werte werden miteinander multipliziert:

$$\text{Ergebnis} = \text{Wert von IN1} \cdot \text{Wert von IN2} \cdot \text{Wert von IN3} \cdot \text{Wert von IN4} \tag{2.98}$$

Das berechnete Ergebnis wird an den Ausgangs-Adapter `OUT` übergeben, und ein entsprechendes Aktualisierungsereignis wird über diesen Plug ausgegeben.

Technische Besonderheiten
- **Generischer Baustein:** Der Baustein basiert auf der generischen Klasse `GEN_AUI_MUL`. Dies ermöglicht eine flexible Anpassung an unterschiedliche Datentypen innerhalb des verwendeten Adaptertyps.
- **Verwendung von Adaptern:** Durch die Kapselung von Daten und Ereignissen in unidirektionalen Adaptern (`AUI`) wird die Anzahl der Verbindungslinien im Application-Editor von 4diac-ide drastisch reduziert, was die Übersichtlichkeit komplexer Anwendungen deutlich erhöht.

Zustandsübersicht
Der Baustein arbeitet rein ereignisgesteuert und besitzt keinen internen Zustand, der über die Dauer eines Berechnungszyklus hinaus bestehen bleibt (zustandsloser / statischer Baustein).

- **Wartezustand:** Der Baustein wartet auf ein Aktualisierungsereignis an einem der Sockets (`IN1` bis `IN4`).
- **Berechnung:** Nach Eintreffen eines Ereignisses wird die Multiplikation durchgeführt.
- **Ausgabe:** Das Ergebnis wird an `OUT` angelegt, ein Ereignis am Plug ausgelöst und der Baustein kehrt sofort in den Wartezustand zurück.

Anwendungsszenarien
- **Signal-Skalierung:** Kaskadierte Skalierung von Messwerten, bei denen ein Rohwert mit mehreren Korrektur- und Kalibrierungsfaktoren multipliziert werden muss.
- **Leistungsberechnung:** Multiplikation von verschiedenen physikalischen Größen zur Ermittlung einer Zielgröße (z. B. Berechnung einer Gesamtleistung aus verschiedenen Faktoren).

- **Modulare Steuerungskonzepte:** Anwendungen, bei denen strukturierte Datenübergaben mittels standardisierter Adapter-Schnittstellen realisiert werden.

Vergleich mit ähnlichen Bausteinen

- **Standard-MUL-Bausteine (IEC 61131-3):** Klassische Multiplizierer arbeiten mit direkten elementaren Datentypen (wie `REAL` oder `INT`) und benötigen separate Event-Eingänge (`REQ`) und Event-Ausgänge (`CNF`). `AUI_MUL_4` vereinfacht dies durch die Kapselung in Adaptern.
- **AUI_MUL_2 / AUI_MUL_3:** Ähnliche Bausteine mit weniger Eingängen. `AUI_MUL_4` eignet sich speziell dann, wenn exakt vier Faktoren multipliziert werden müssen, ohne mehrere Multiplikationsbausteine hintereinander schalten zu müssen (was Ressourcen und Latenz im Ausführungsmodell spart).
- `AUI_MUL_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

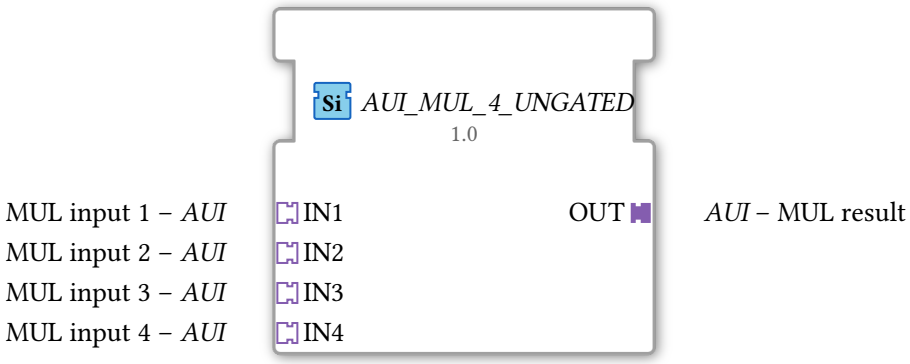
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der Funktionsbaustein `AUI_MUL_4` stellt eine effiziente und saubere Lösung für arithmetische Multiplikationen von vier Faktoren innerhalb einer adapterbasierten 4diac-Architektur dar. Er fördert das modulare Design und die Übersichtlichkeit der IEC 61499 Applikation.

AUI_MUL_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUI_MUL_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



Kein Bild verfügbar

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AUI_MUL_4_UNGATED` ist ein generischer Baustein zur Durchführung einer arithmetischen Multiplikation von vier Eingangswerten. Er nutzt unidirektionale Adapter vom Typ `AUI` (Adapter Unit Interface), um Daten und zugehörige Ereignisse gebündelt zu übertragen. Dies ermöglicht eine saubere und übersichtliche Modellierung in der 4diac-ide.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung wird über die Adapter-Schnittstellen abgewickelt.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge vorhanden. Die Ereignissteuerung wird über die Adapter-Schnittstellen abgewickelt.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingänge)

- **IN1** (Typ: `adapter::types::unidirectional::AUI`): Erster Multiplikand.
- **IN2** (Typ: `adapter::types::unidirectional::AUI`): Zweiter Multiplikand.
- **IN3** (Typ: `adapter::types::unidirectional::AUI`): Dritter Multiplikand.
- **IN4** (Typ: `adapter::types::unidirectional::AUI`): Vierter Multiplikand.

Plugs (Ausgänge)

- **OUT** (Typ: `adapter::types::unidirectional::AUI`): Ausgang für das Ergebnis der Multiplikation ($OUT = IN1 \cdot IN2 \cdot IN3 \cdot IN4$).

Funktionsweise

Sobald an einem der Eingangs-Adapter (`IN1` bis `IN4`) ein neues Ereignis signalisiert wird, liest der Funktionsbaustein die aktuellen Werte der vier Eingänge aus. Diese Werte werden miteinander multipliziert:

$$\text{Ergebnis} = \text{Wert von IN1} \cdot \text{Wert von IN2} \cdot \text{Wert von IN3} \cdot \text{Wert von IN4} \tag{2.99}$$

Das berechnete Ergebnis wird an den Ausgangs-Adapter `OUT` übergeben, und ein entsprechendes Aktualisierungsereignis wird über diesen Plug ausgegeben.

Technische Besonderheiten

- **Generischer Baustein:** Der Baustein basiert auf der generischen Klasse `GEN_AUI_MUL`. Dies ermöglicht eine flexible Anpassung an unterschiedliche Datentypen innerhalb des verwendeten Adaptertyps.
- **Verwendung von Adaptern:** Durch die Kapselung von Daten und Ereignissen in unidirektionalen Adaptern (`AUI`) wird die Anzahl der Verbindungslinien im Application-Editor von 4diac-ide drastisch reduziert, was die Übersichtlichkeit komplexer Anwendungen deutlich erhöht.

Zustandsübersicht

Der Baustein arbeitet rein ereignisgesteuert und besitzt keinen internen Zustand, der über die Dauer eines Berechnungszyklus hinaus bestehen bleibt (zustandsloser / statischer Baustein).

- **Wartezustand:** Der Baustein wartet auf ein Aktualisierungsereignis an einem der Sockets (`IN1` bis `IN4`).
- **Berechnung:** Nach Eintreffen eines Ereignisses wird die Multiplikation durchgeführt.
- **Ausgabe:** Das Ergebnis wird an `OUT` angelegt, ein Ereignis am Plug ausgelöst und der Baustein kehrt sofort in den Wartezustand zurück.

Anwendungsszenarien

- **Signal-Skalierung:** Kaskadierte Skalierung von Messwerten, bei denen ein Rohwert mit mehreren Korrektur- und Kalibrierungsfaktoren multipliziert werden muss.
- **Leistungsberechnung:** Multiplikation von verschiedenen physikalischen Größen zur Ermittlung einer Zielgröße (z. B. Berechnung einer Gesamtleistung aus verschiedenen Faktoren).
- **Modulare Steuerungskonzepte:** Anwendungen, bei denen strukturierte Datenübergaben mittels standardisierter Adapter-Schnittstellen realisiert werden.

Vergleich mit ähnlichen Bausteinen

- **Standard-MUL-Bausteine (IEC 61131-3):** Klassische Multiplizierer arbeiten mit direkten elementaren Datentypen (wie `REAL` oder `INT`) und benötigen separate Event-Eingänge (`REQ`) und Event-Ausgänge (`CNF`). `AUI_MUL_4_UNGATED` vereinfacht dies durch die Kapselung in Adaptern.
- **AUI_MUL_2 / AUI_MUL_3:** Ähnliche Bausteine mit weniger Eingängen. `AUI_MUL_4_UNGATED` eignet sich speziell dann, wenn exakt vier Faktoren multipliziert werden müssen, ohne mehrere Multiplikationsbausteine hintereinander schalten zu müssen (was Ressourcen und Latenz im Ausführungsmodell spart).
- `AUI_MUL_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

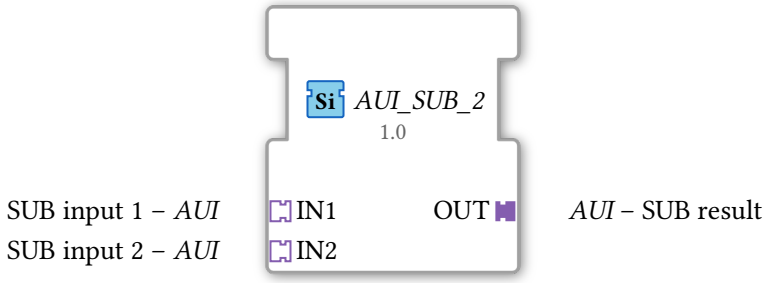
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der Funktionsbaustein `AUI_MUL_4_UNGATED` stellt eine effiziente und saubere Lösung für arithmetische Multiplikationen von vier Faktoren innerhalb einer adapterbasierten 4diac-Architektur dar. Er fördert das modulare Design und die Übersichtlichkeit der IEC 61499 Applikation.

AUI_SUB_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AUI_SUB_2` dient zur Durchführung einer arithmetischen Subtraktion innerhalb einer IEC 61499-Anwendung. Es handelt sich hierbei um einen generischen Funktionsbaustein (Generic FB), der speziell für die Interaktion mit Adapter-Schnittstellen entwickelt wurde. Er ermöglicht es, Subtraktionsoperationen modular und strukturiert über standardisierte Adapterverbindungen abzubilden, anstatt diskrete Einzeldrähte für Daten und Ereignisse zu verwenden.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Eingänge. Die Ereignissteuerung wird intern über die angebundenen Adapter abgewickelt.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Ausgänge. Die Ereignisweiterleitung erfolgt gekapselt über den Ausgangs-Adapter.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübergabe erfolgt über die Eingangs-Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Das Ergebnis wird über den Ausgangs-Adapter bereitgestellt.

Adapter

Sockets (Steckdosen / Eingänge)

- **IN1** (Typ: `adapter::types::unidirectional::AUI`): Erster Eingangsadapter für die Subtraktion. Dieser repräsentiert den Minuenden (Wert, von dem subtrahiert wird).
- **IN2** (Typ: `adapter::types::unidirectional::AUI`): Zweiter Eingangsadapter für die Subtraktion. Dieser repräsentiert den Subtrahenden (Wert, der subtrahiert wird).

Plugs (Stecker / Ausgänge)

- **OUT** (Typ: `adapter::types::unidirectional::AUI`): Ausgangsadapter, der das Ergebnis der Subtraktion (die Differenz) zur weiteren Verwendung bereitstellt.

Funktionsweise

Der Baustein führt die grundlegende mathematische Operation der Subtraktion auf den über die Adapter eingehenden Werten aus:

$$\mathrm{OUT} = \mathrm{IN1} - \mathrm{IN2} \tag{2.100}$$

Sobald an den Eingangs-Adaptern `IN1` oder `IN2` eine Wertänderung signalisiert wird (getriggert durch die ereignisgesteuerte Natur des zugrundeliegenden `AUI`-Adapters), berechnet der Baustein die Differenz und gibt das Ergebnis sowie ein entsprechendes Aktualisierungsereignis über den Ausgangs-Adapter `OUT` aus.

Technische Besonderheiten

- **Generischer Typ:** Der Baustein basiert auf der generischen Klasse `GEN_AUI_SUB`. Dies erlaubt eine flexible Handhabung unterschiedlicher Datentypen, sofern diese vom zugrundeliegenden Adaptertyp `AUI` (Analog User Interface / Unidirectional) unterstützt werden.
- **Kapselung:** Durch die Verwendung von unidirektionalen Adaptern werden Signalflüsse und die dazugehörigen Trigger-Events sauber gebündelt. Dies vereinfacht das Applikationsdesign in der 4diac IDE signifikant und reduziert die Anzahl der sichtbaren Verbindungslinien.

Zustandsübersicht

Der Baustein selbst verwaltet keinen komplexen internen Zustand (zustandslos im Sinne einer State Machine). Seine Ausführung verhält sich rein reaktiv:

1. **Warten:** Der Baustein wartet auf ein Aktualisierungsereignis an `IN1` oder `IN2`.
2. **Berechnung:** Nach dem Eintreffen eines Ereignisses werden die Datenwerte ausgelesen und die Subtraktion durchgeführt.
3. **Ausgabe:** Der neue Differenzwert wird an `OUT` angelegt und das Ausgangsereignis des Adapters getriggert.

Anwendungsszenarien

- **Differenzdruck- / Differenztemperaturmessung:** Berechnung der Abweichung zwischen zwei analogen Sensoren, deren Werte bereits als strukturierte Adapter-Signale vorliegen.
- **Soll-Ist-Wert-Vergleich:** Subtraktion eines Istwerts von einem Sollwert zur Ermittlung der Regeldifferenz in Regelungskreisen.
- **Nullpunktkompensation (Offset-Berechnung):** Abzug eines statischen oder dynamischen Offset-Wertes (über `IN2`) von einem Rohsignal (über `IN1`).

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-Subtraktionsblock (wie dem klassischen `SUB`-Baustein aus der IEC 61131-3 Bibliothek) benötigt der `AUI_SUB_2` keine explizite Verdrahtung von separaten Event- und Data-Ports (z.B. `REQ` und `IN1`, `IN2`). Die gesamte Logik der Wertübertragung und -aktualisierung ist im `AUI`-Adapter gekapselt. Dies macht den `AUI_SUB_2` wartungsfreundlicher und robuster gegenüber Verdrahtungsfehlern in komplexen Systemarchitekturen.

- **`AUI_SUB_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

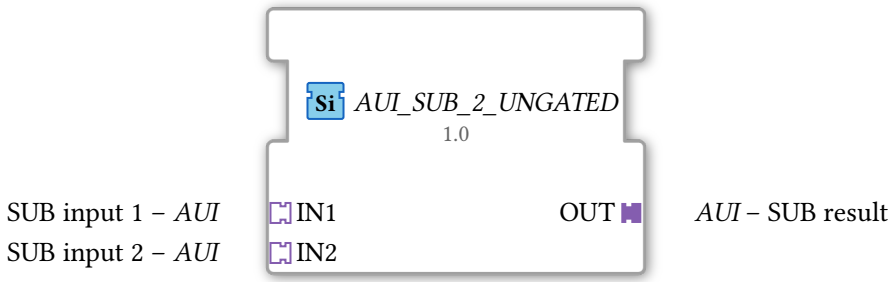
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `AUI_SUB_2` ist ein hocheffizienter, übersichtlicher und moderner Funktionsbaustein zur Durchführung von Subtraktionen. Durch die konsequente Nutzung von Adaptern fügt er sich nahtlos in serviceorientierte und modularisierte Steuerungskonzepte ein und trägt zu einem sauberen Software-Design in der 4diac IDE bei.

AUI_SUB_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUI_SUB_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AUI_SUB_2_UNGATED` dient zur Durchführung einer arithmetischen Subtraktion innerhalb einer IEC 61499-Anwendung. Es handelt sich hierbei um einen generischen Funktionsbaustein (Generic FB), der speziell für die Interaktion mit Adapter-Schnittstellen entwickelt wurde. Er ermöglicht es, Subtraktionsoperationen modular und strukturiert über standardisierte Adapterverbindungen abzubilden, anstatt diskrete Einzeldrähte für Daten und Ereignisse zu verwenden.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Eingänge. Die Ereignissteuerung wird intern über die angebundenen Adapter abgewickelt.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Ausgänge. Die Ereignisweiterleitung erfolgt gekapselt über den Ausgangs-Adapter.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübergabe erfolgt über die Eingangs-Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Das Ergebnis wird über den Ausgangs-Adapter bereitgestellt.

Adapter

Sockets (Steckdosen / Eingänge)

- **IN1** (Typ: `adapter::types::unidirectional::AUI`): Erster Eingangsadapter für die Subtraktion. Dieser repräsentiert den Minuenden (Wert, von dem subtrahiert wird).
- **IN2** (Typ: `adapter::types::unidirectional::AUI`): Zweiter Eingangsadapter für die Subtraktion. Dieser repräsentiert den Subtrahenden (Wert, der subtrahiert wird).

Plugs (Stecker / Ausgänge)

- **OUT** (Typ: `adapter::types::unidirectional::AUI`): Ausgangsadapter, der das Ergebnis der Subtraktion (die Differenz) zur weiteren Verwendung bereitstellt.

Funktionsweise

Der Baustein führt die grundlegende mathematische Operation der Subtraktion auf den über die Adapter eingehenden Werten aus:

$$\mathrm{OUT} = \mathrm{IN1} - \mathrm{IN2} \tag{2.101}$$

Sobald an den Eingangs-Adaptern `IN1` oder `IN2` eine Wertänderung signalisiert wird (getriggert durch die ereignisgesteuerte Natur des zugrundeliegenden `AUI`-Adapters), berechnet der Baustein die Differenz und gibt das Ergebnis sowie ein entsprechendes Aktualisierungsereignis über den Ausgangs-Adapter `OUT` aus.

Technische Besonderheiten

- **Generischer Typ:** Der Baustein basiert auf der generischen Klasse `GEN_AUI_SUB`. Dies erlaubt eine flexible Handhabung unterschiedlicher Datentypen, sofern diese vom zugrundeliegenden Adaptertyp `AUI` (Analog User Interface / Unidirectional) unterstützt werden.
- **Kapselung:** Durch die Verwendung von unidirektionalen Adaptern werden Signalflüsse und die dazugehörigen Trigger-Events sauber gebündelt. Dies vereinfacht das Applikationsdesign in der 4diac IDE signifikant und reduziert die Anzahl der sichtbaren Verbindungslinien.

Zustandsübersicht

Der Baustein selbst verwaltet keinen komplexen internen Zustand (zustandslos im Sinne einer State Machine). Seine Ausführung verhält sich rein reaktiv:

1. **Warten:** Der Baustein wartet auf ein Aktualisierungsereignis an `IN1` oder `IN2`.
2. **Berechnung:** Nach dem Eintreffen eines Ereignisses werden die Datenwerte ausgelesen und die Subtraktion durchgeführt.
3. **Ausgabe:** Der neue Differenzwert wird an `OUT` angelegt und das Ausgangsereignis des Adapters getriggert.

Anwendungsszenarien

- **Differenzdruck- / Differenztemperaturmessung:** Berechnung der Abweichung zwischen zwei analogen Sensoren, deren Werte bereits als strukturierte Adapter-Signale vorliegen.
- **Soll-Ist-Wert-Vergleich:** Subtraktion eines Istwerts von einem Sollwert zur Ermittlung der Regeldifferenz in Regelungskreisen.
- **Nullpunktkompensation (Offset-Berechnung):** Abzug eines statischen oder dynamischen Offset-Wertes (über `IN2`) von einem Rohsignal (über `IN1`).

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-Subtraktionsblock (wie dem klassischen `SUB`-Baustein aus der IEC 61131-3 Bibliothek) benötigt der `AUI_SUB_2_UNGATED` keine explizite Verdrahtung von separaten Event- und Data-Ports (z.B. `REQ` und `IN1`, `IN2`). Die gesamte Logik der Wertübertragung und -aktualisierung ist im `AUI`-Adapter gekapselt. Dies macht den `AUI_SUB_2_UNGATED` wartungsfreundlicher und robuster gegenüber Verdrahtungsfehlern in komplexen Systemarchitekturen.

- `AUI_SUB_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

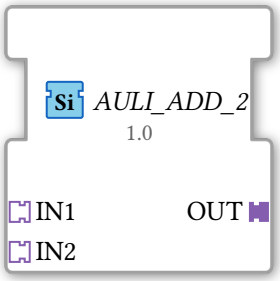
Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AUI_SUB_2_UNGATED` ist ein hocheffizienter, übersichtlicher und moderner Funktionsbaustein zur Durchführung von Subtraktionen. Durch die konsequente Nutzung von Adaptern fügt er sich nahtlos in serviceorientierte und modularisierte Steuerungskonzepte ein und trägt zu einem sauberen Software-Design in der 4diac IDE bei.

AULI_ADD_2

```
       +-------------------------------+
       |          AULI_ADD_2           |
       |                               |
IN1 ==| (Socket)               (Plug) |== OUT
       |                               |
IN2 ==| (Socket)                      |
       +-------------------------------+
```



ADD input 1 – *AULI*

ADD input 2 – *AULI*

AULI – ADD result

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AULI_ADD_2` ist ein generischer Arithmetikbaustein für die IEC 61499-Umgebung (4diac-ide). Er dient zur Durchführung einer mathematischen Addition zweier Eingangswerte, die über standardisierte, unidirektionale Adapter übertragen werden. Das berechnete Ergebnis wird an einen entsprechenden Ausgangs-Adapter übergeben.

Schnittstellenstruktur

Der Baustein besitzt keine klassischen, diskreten Ereignis- oder Dateneingänge auf der obersten Ebene. Die gesamte Kommunikation und Werteübertragungen werden über Adapter abgewickelt.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. (Die Ereignissteuerung ist in den Adaptern gekapselt).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. (Die Ereignissteuerung ist in den Adaptern gekapselt).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AULI`): Der erste Summand für die Additionsoperation.
- **IN2** (Typ: `adapter::types::unidirectional::AULI`): Der zweite Summand für die Additionsoperation.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AULI`): Das Ergebnis der Addition ($OUT = IN1 + IN2$).

Funktionsweise

Der Funktionsbaustein `AULI_ADD_2` führt eine klassische Addition aus. Sobald sich Werte an den Eingangs-Adaptern `IN1` oder `IN2` ändern bzw. ein entsprechendes Aktualisierungsereignis über die Adapter getriggert wird, berechnet der Baustein die Summe der übertragenen Werte:

$$\text{Ergebnis} = \text{Wert } (IN1) + \text{Wert } (IN2) \tag{2.102}$$

Dieses Ergebnis sowie das dazugehörige Aktualisierungsereignis werden anschließend über den Ausgangs-Plug `OUT` an nachfolgende Bausteine weitergeleitet.

Technische Besonderheiten

- **Generischer Charakter:** Der Baustein ist über das Attribut `GenericClassName` als `GEN_AULI_ADD` deklariert. Dies ermöglicht eine flexible Typisierung je nach konkreter Implementierung der verwendeten `AULI`-Adapterdaten.
- **Adapter-Kapselung:** Durch die Verwendung von Adaptern des Typs `unidirectional::AULI` wird die Komplexität der Steuerungsschaltpläne reduziert, da Daten und zugehörige Trigger-Ereignisse in einer einzigen Verbindung gebündelt sind.

Zustandsübersicht

Da es sich bei diesem Baustein um einen rein funktionalen/mathematischen Koppler handelt, besitzt er im Regelfall keine komplexe interne Zustandsmaschine (ECC). Seine Ausführung ist rein daten- bzw. ereignisgetrieben basierend auf den eingehenden Adapter-Signalen:

1. **Wartezustand (Idle):** Baustein wartet auf Signalaktualisierungen an `IN1` oder `IN2`.
2. **Berechnung (Evaluate):** Bei Event-Eingang an einem der Sockets werden die Datenwerte addiert.
3. **Ausgabe (Propagate):** Der addierte Wert wird über den Plug `OUT` ausgegeben und das zugehörige Event getriggert.

Anwendungsszenarien

- **Messwert-Aggregation:** Addition von zwei analogen Prozesswerten (z. B. zwei parallele Durchflusssensoren zur Ermittlung des Gesamtdurchflusses).
- **Sollwert-Offset:** Beaufschlagung eines Hauptsollwerts (über `IN1`) mit einem Korrekturwert oder Offset (über `IN2`).
- **Signalbündelung:** Strukturierte mathematische Signalverarbeitung in komplexen Anlagensteuerungen unter Verwendung von standardisierten Adapter-Schnittstellen zur Reduzierung des Verdrahtungsaufwands.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-Additionsbaustein (wie dem IEC 61131-3 konformen `ADD`-Baustein), welcher mit diskreten Pins für `REQ`, `CNF`, `IN1`, `IN2` und `OUT` arbeitet, entfällt beim `AULI_ADD_2` die explizite Verdrahtung von Kontrollflüssen (Events) und Datenleitungen. Dies erhöht die Übersichtlichkeit in der 4diac-Entwicklungsumgebung signifikant, da pro Signalquelle nur noch eine einzige Adapterlinie gezogen werden muss.

- **`AULI_ADD_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

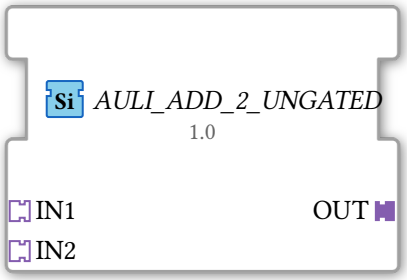
Fazit

Der `AULI_ADD_2` ist ein hocheffizienter, modularer Hilfsbaustein für die analoge Signalverarbeitung. Durch die konsequente Nutzung von unidirektionalen Adaptern erleichtert er den Entwurf sauber strukturierter und wartungsfreundlicher Steuerungsanwendungen nach dem IEC 61499 Standard.

AULI_ADD_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AULI_ADD_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

```
      +------------------------------+
      |          AULI_ADD_2_UNGATED          |
      |                              |
IN1 ==| (Socket)               (Plug) |== OUT
      |                              |
IN2 ==| (Socket)                     |
      +------------------------------+
```



ADD input 1 – *AULI*
ADD input 2 – *AULI*
AULI – ADD result

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AULI_ADD_2_UNGATED` ist ein generischer Arithmetikbaustein für die IEC 61499-Umgebung (4diac-ide). Er dient zur Durchführung einer mathematischen Addition zweier Eingangswerte, die über standardisierte, unidirektionale Adapter übertragen werden. Das berechnete Ergebnis wird an einen entsprechenden Ausgangs-Adapter übergeben.

Schnittstellenstruktur

Der Baustein besitzt keine klassischen, diskreten Ereignis- oder Dateneingänge auf der obersten Ebene. Die gesamte Kommunikation und Werteübertragungen werden über Adapter abgewickelt.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. (Die Ereignissteuerung ist in den Adaptern gekapselt).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. (Die Ereignissteuerung ist in den Adaptern gekapselt).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AULI`): Der erste Summand für die Additionsoperation.

- **IN2** (Typ: `adapter::types::unidirectional::AULI`): Der zweite Summand für die Additionsoperation.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AULI`): Das Ergebnis der Addition ($OUT = IN1 + IN2$).

Funktionsweise

Der Funktionsbaustein `AULI_ADD_2_UNGATED` führt eine klassische Addition aus. Sobald sich Werte an den Eingangs-Adaptern `IN1` oder `IN2` ändern bzw. ein entsprechendes Aktualisierungsereignis über die Adapter getriggert wird, berechnet der Baustein die Summe der übertragenen Werte:

$$\text{Ergebnis} = \text{Wert } (IN1) + \text{Wert } (IN2) \tag{2.103}$$

Dieses Ergebnis sowie das dazugehörige Aktualisierungsereignis werden anschließend über den Ausgangs-Plug `OUT` an nachfolgende Bausteine weitergeleitet.

Technische Besonderheiten

- **Generischer Charakter:** Der Baustein ist über das Attribut `GenericClassName` als `GEN_AULI_ADD` deklariert. Dies ermöglicht eine flexible Typisierung je nach konkreter Implementierung der verwendeten `AULI`-Adapterdaten.
- **Adapter-Kapselung:** Durch die Verwendung von Adaptern des Typs `unidirectional::AULI` wird die Komplexität der Steuerungsschaltpläne reduziert, da Daten und zugehörige Trigger-Ereignisse in einer einzigen Verbindung gebündelt sind.

Zustandsübersicht

Da es sich bei diesem Baustein um einen rein funktionalen/mathematischen Koppler handelt, besitzt er im Regelfall keine komplexe interne Zustandsmaschine (ECC). Seine Ausführung ist rein daten- bzw. ereignisgetrieben basierend auf den eingehenden Adapter-Signalen:

1. **Wartezustand (Idle):** Baustein wartet auf Signalaktualisierungen an `IN1` oder `IN2`.
2. **Berechnung (Evaluate):** Bei Event-Eingang an einem der Sockets werden die Datenwerte addiert.
3. **Ausgabe (Propagate):** Der addierte Wert wird über den Plug `OUT` ausgegeben und das zugehörige Event getriggert.

Anwendungsszenarien

- **Messwert-Aggregation:** Addition von zwei analogen Prozesswerten (z. B. zwei parallele Durchflusssensoren zur Ermittlung des Gesamtdurchflusses).
- **Sollwert-Offset:** Beaufschlagung eines Hauptsollwerts (über `IN1`) mit einem Korrekturwert oder Offset (über `IN2`).
- **Signalbündelung:** Strukturierte mathematische Signalverarbeitung in komplexen Anlagensteuerungen unter Verwendung von standardisierten Adapter-Schnittstellen zur Reduzierung des Verdrahtungsaufwands.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-Additionsbaustein (wie dem IEC 61131-3 konformen `ADD`-Baustein), welcher mit diskreten Pins für `REQ`, `CNF`, `IN1`, `IN2` und `OUT` arbeitet, entfällt beim `AULI_ADD_2_UNGATED` die explizite Verdrahtung von Kontrollflüssen (Events) und Datenleitungen. Dies erhöht die Übersichtlichkeit in der 4diac-Entwicklungsumgebung signifikant, da pro Signalquelle nur noch eine einzige Adapterlinie gezogen werden muss.

- **`AULI_ADD_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

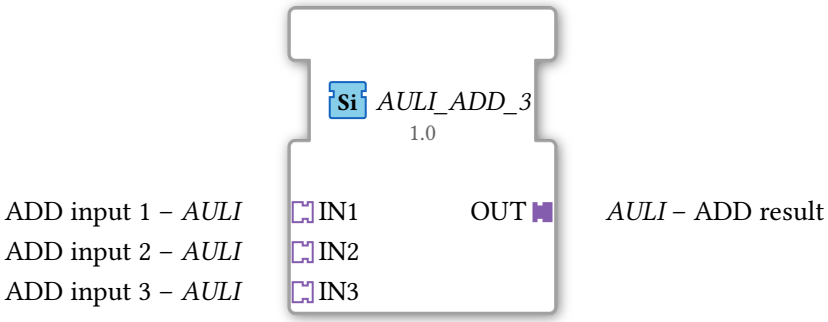
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AULI_ADD_2_UNGATED` ist ein hocheffizienter, modularer Hilfsbaustein für die analoge Signalverarbeitung. Durch die konsequente Nutzung von unidirektionalen Adaptern erleichtert er den Entwurf sauber strukturierter und wartungsfreundlicher Steuerungsanwendungen nach dem IEC 61499 Standard.

AULI_ADD_3



(Bild des Funktionsbausteins temporär nicht verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AULI_ADD_3` ist ein generischer Arithmetik-Baustein für IEC 61499-Systeme, der speziell für die Addition von drei Eingangswerten entwickelt wurde. Im Gegensatz zu klassischen mathematischen Funktionsbausteinen nutzt dieser Baustein Adapterverbindungen des Typs `AULI` (unidirektional), um Daten und zugehörige Kontrollsignale gebündelt zu übertragen. Dies reduziert den Verdrahtungsaufwand innerhalb der Entwicklungsumgebung (4diac IDE) erheblich.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine direkten Ereignis-Eingänge definiert. Die ereignisbasierte Steuerung wird implizit über die angeschlossenen Adapter (`Sockets`) abgewickelt.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge definiert. Die Ereignisweiterleitung erfolgt gekapselt über den Ausgangs-Adapter (`Plug`).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden. Der Datentransfer erfolgt über die Adapter-Schnittstellen.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden. Das Berechnungsergebnis wird über den Ausgangs-Adapter bereitgestellt.

Adapter

Sockets (Eingangs-Adapter)

Die Sockets dienen als Empfänger für die zu addierenden Werte.

- **IN1** (Typ: `adapter::types::unidirectional::AULI`): Erster Summand der arithmetischen Addition.
- **IN2** (Typ: `adapter::types::unidirectional::AULI`): Zweiter Summand der arithmetischen Addition.
- **IN3** (Typ: `adapter::types::unidirectional::AULI`): Dritter Summand der arithmetischen Addition.

Plugs (Ausgangs-Adapter)

Der Plug dient als Sender für das berechnete Ergebnis.

- **OUT** (Typ: `adapter::types::unidirectional::AULI`): Ausgang für das Ergebnis der Addition ($IN1 + IN2 + IN3$).

Funktionsweise

Sobald an den Eingangs-Adaptern (`IN1`, `IN2`, `IN3`) neue Daten signalisiert werden, führt der Baustein die mathematische Addition aus.

Die zugrunde liegende Berechnungsformel lautet:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} \tag{2.104}$$

Das berechnete Ergebnis wird unmittelbar an den Ausgangs-Adapter `OUT` übergeben und steht nachfolgenden Bausteinen zur Verfügung. Da es sich um unidirektionale Adapter handelt, fließt der Datenstrom ausschließlich von den Sockets zum Plug.

Technische Besonderheiten

- **Generischer Baustein:** Der Baustein ist in der 4diac-IDE als generischer Typ deklariert (`GenericClassName = 'GEN_AULI_ADD'`). Dies ermöglicht eine flexible Verarbeitung verschiedener Datentypen (z. B. INT, REAL, LREAL), die durch die Definition des `AULI`-Adapters gestützt werden.
- **Adapter-Kapselung:** Durch die Verwendung von Adaptern anstelle von Standard-Daten- und Eventkanälen wird das Systemdesign modularer und übersichtlicher.
- **Ressourceneffizienz:** Die Berechnung erfolgt direkt auf Basis der Adapterereignisse, was eine performante Ausführung gewährleistet.

Zustandsübersicht

Der Funktionsbaustein arbeitet als zustandsloser (bzw. rein ereignisgesteuerter) arithmetischer Baustein:

1. **Bereitschaft (Idle):** Baustein wartet auf Wertänderungen oder Trigger-Signale an den Eingangs-Adaptern `IN1`, `IN2` oder `IN3`.
2. **Berechnung (Evaluation):** Nach dem Eintreffen eines Signals werden die Werte ausgelesen und summiert.
3. **Aktualisierung (Output):** Die Summe wird an `OUT` übergeben und das zugehörige Event im Adapter getriggert. Der Baustein kehrt sofort in den Zustand *Bereitschaft* zurück.

Anwendungsszenarien

- **Messwert-Summierung:** Zusammenfassung von drei analogen Messwerten (z. B. drei Teilstromstärken zur Ermittlung des Gesamtstroms) in der Prozessautomatisierung.
- **Sollwert-Generierung:** Addition von Basis-Sollwerten mit zwei verschiedenen Korrektur- oder Offset-Werten.
- **Strukturierte Datenverarbeitung:** Einsatz in komplexen IEC 61499 Steuerungsanwendungen, bei denen ein konsistenter Einsatz von Adaptern zur Erhöhung der Übersichtlichkeit gefordert ist.

Vergleich mit ähnlichen Bausteinen

- **Standard-ADD (IEC 61131-3):** Der klassische `ADD`-Baustein benötigt explizite `REQ`- und `CNF`-Event-Leitungen sowie einzelne Datenleitungen. `AULI_ADD_3` ersetzt diese durch nur vier Adapter-Verbindungen, was das visuelle Programmierbild aufräumt.
- **Kaskadierte Addierer (z. B. 2x `AULI_ADD_2`):** Um drei Werte mit klassischen Zwei-Wege-Addierern zu addieren, müssten zwei Bausteine in Reihe geschaltet werden. `AULI_ADD_3` spart hierbei einen kompletten Funktionsblock sowie die dazugehörige Zwischenverdrahtung ein.

- **`AULI_ADD_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

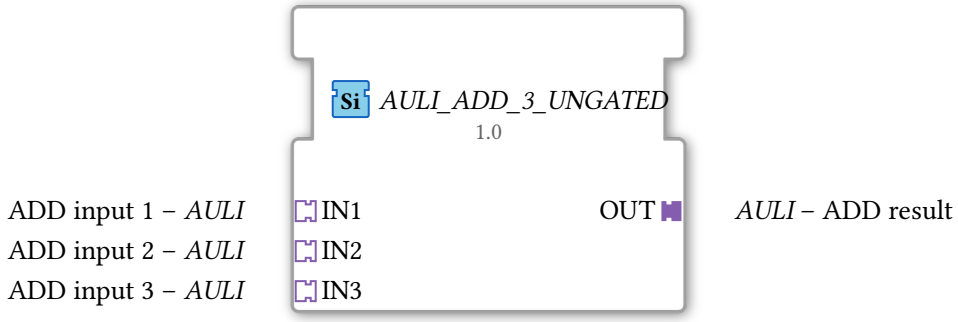
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `AULI_ADD_3` ist ein praktischer und moderner Funktionsbaustein für die strukturierte Programmierung in 4diac. Durch die konsequente Nutzung des unidirektionalen `AULI`-Adapters eignet er sich ideal für saubere, übersichtliche und wartbare Architekturen in verteilten Steuerungssystemen nach dem IEC 61499-Standard.

AULI_ADD_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AULI_ADD_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Bild des Funktionsbausteins temporär nicht verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AULI_ADD_3_UNGATED` ist ein generischer Arithmetik-Baustein für IEC 61499-Systeme, der speziell für die Addition von drei Eingangswerten entwickelt wurde. Im Gegensatz zu klassischen mathematischen Funktionsbausteinen nutzt dieser Baustein Adapterverbindungen des Typs `AULI` (unidirektional), um Daten und zugehörige Kontrollsignale gebündelt zu übertragen. Dies reduziert den Verdrahtungsaufwand innerhalb der Entwicklungsumgebung (4diac IDE) erheblich.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine direkten Ereignis-Eingänge definiert. Die ereignisbasierte Steuerung wird implizit über die angeschlossenen Adapter (`Sockets`) abgewickelt.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge definiert. Die Ereignisweiterleitung erfolgt gekapselt über den Ausgangs-Adapter (`Plug`).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden. Der Datentransfer erfolgt über die Adapter-Schnittstellen.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden. Das Berechnungsergebnis wird über den Ausgangs-Adapter bereitgestellt.

Adapter

Sockets (Eingangs-Adapter)

Die Sockets dienen als Empfänger für die zu addierenden Werte.

- **IN1** (Typ: `adapter::types::unidirectional::AULI`): Erster Summand der arithmetischen Addition.
- **IN2** (Typ: `adapter::types::unidirectional::AULI`): Zweiter Summand der arithmetischen Addition.
- **IN3** (Typ: `adapter::types::unidirectional::AULI`): Dritter Summand der arithmetischen Addition.

Plugs (Ausgangs-Adapter)

Der Plug dient als Sender für das berechnete Ergebnis.

- **OUT** (Typ: `adapter::types::unidirectional::AULI`): Ausgang für das Ergebnis der Addition ($IN1 + IN2 + IN3$).

Funktionsweise

Sobald an den Eingangs-Adaptern (`IN1`, `IN2`, `IN3`) neue Daten signalisiert werden, führt der Baustein die mathematische Addition aus.

Die zugrunde liegende Berechnungsformel lautet:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} \tag{2.105}$$

Das berechnete Ergebnis wird unmittelbar an den Ausgangs-Adapter `OUT` übergeben und steht nachfolgenden Bausteinen zur Verfügung. Da es sich um unidirektionale Adapter handelt, fließt der Datenstrom ausschließlich von den Sockets zum Plug.

Technische Besonderheiten

- **Generischer Baustein:** Der Baustein ist in der 4diac-IDE als generischer Typ deklariert (`GenericClassName = 'GEN_AULI_ADD'`). Dies ermöglicht eine flexible Verarbeitung verschiedener Datentypen (z. B. INT, REAL, LREAL), die durch die Definition des `AULI`-Adapters gestützt werden.
- **Adapter-Kapselung:** Durch die Verwendung von Adaptern anstelle von Standard-Daten- und Eventkanälen wird das Systemdesign modularer und übersichtlicher.
- **Ressourceneffizienz:** Die Berechnung erfolgt direkt auf Basis der Adapterereignisse, was eine performante Ausführung gewährleistet.

Zustandsübersicht

Der Funktionsbaustein arbeitet als zustandsloser (bzw. rein ereignisgesteuerter) arithmetischer Baustein:

1. **Bereitschaft (Idle):** Baustein wartet auf Wertänderungen oder Trigger-Signale an den Eingangs-Adaptern `IN1`, `IN2` oder `IN3`.
2. **Berechnung (Evaluation):** Nach dem Eintreffen eines Signals werden die Werte ausgelesen und summiert.
3. **Aktualisierung (Output):** Die Summe wird an `OUT` übergeben und das zugehörige Event im Adapter getriggert. Der Baustein kehrt sofort in den Zustand *Bereitschaft* zurück.

Anwendungsszenarien

- **Messwert-Summierung:** Zusammenfassung von drei analogen Messwerten (z. B. drei Teilstromstärken zur Ermittlung des Gesamtstroms) in der Prozessautomatisierung.
- **Sollwert-Generierung:** Addition von Basis-Sollwerten mit zwei verschiedenen Korrektur- oder Offset-Werten.
- **Strukturierte Datenverarbeitung:** Einsatz in komplexen IEC 61499 Steuerungsanwendungen, bei denen ein konsistenter Einsatz von Adaptern zur Erhöhung der Übersichtlichkeit gefordert ist.

Vergleich mit ähnlichen Bausteinen

- **Standard-ADD (IEC 61131-3):** Der klassische `ADD`-Baustein benötigt explizite `REQ`- und `CNF`-Event-Leitungen sowie einzelne Datenleitungen. `AULI_ADD_3_UNGATED` ersetzt diese durch nur vier Adapter-Verbindungen, was das visuelle Programmierbild aufräumt.
- **Kaskadierte Addierer (z. B. 2x `AULI_ADD_2`):** Um drei Werte mit klassischen Zwei-Wege-Addierern zu addieren, müssten zwei Bausteine in Reihe geschaltet werden. `AULI_ADD_3_UNGATED` spart hierbei einen kompletten Funktionsblock sowie die dazugehörige Zwischenverdrahtung ein.

- **`AULI_ADD_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

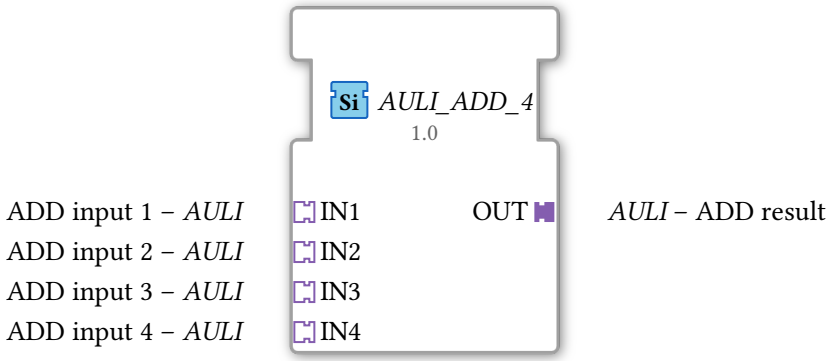
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AULI_ADD_3_UNGATED` ist ein praktischer und moderner Funktionsbaustein für die strukturierte Programmierung in 4diac. Durch die konsequente Nutzung des unidirektionalen `AULI`-Adapters eignet er sich ideal für saubere, übersichtliche und wartbare Architekturen in verteilten Steuerungssystemen nach dem IEC 61499-Standard.

AULI_ADD_4



(Kein Bild vorhanden)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AULI_ADD_4` ist ein generischer Funktionsbaustein (FB), der für die mathematische Addition von vier Werten konzipiert ist. Die Besonderheit dieses Bausteins liegt in der Verwendung von Adaptern des Typs `AULI` (unidirektional). Durch die Kapselung der Daten- und Ereignisübertragung in Adaptern ermöglicht der Baustein eine übersichtliche und modulare Verdrahtung innerhalb von IEC 61499-Anwendungen in der 4diac IDE.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung erfolgt implizit über die angeschlossenen Adapter-Sockets.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge vorhanden. Die Ereignissteuerung erfolgt implizit über den angeschlossenen Adapter-Plug.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübergabe erfolgt über die Eingangs-Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Die Datenausgabe erfolgt über den Ausgangs-Adapter.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AULI`): Erster Summand für die Additionsoperation.
- **IN2** (Typ: `adapter::types::unidirectional::AULI`): Zweiter Summand für die Additionsoperation.
- **IN3** (Typ: `adapter::types::unidirectional::AULI`): Dritter Summand für die Additionsoperation.
- **IN4** (Typ: `adapter::types::unidirectional::AULI`): Vierter Summand für die Additionsoperation.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AULI`): Enthält das Ergebnis der Addition (`IN1 + IN2 + IN3 + IN4`) sowie das dazugehörige Ausgabe-Ereignis.

Funktionsweise

Sobald an einem oder mehreren der Eingangs-Adapter (`IN1` bis `IN4`) ein neues Wert-Ereignis signalisiert wird, liest der Funktionsbaustein die aktuellen Werte aus allen vier Adaptern aus. Diese Werte werden mathematisch addiert:

$$\text{Ergebnis} = \text{Wert}_{IN1} + \text{Wert}_{IN2} + \text{Wert}_{IN3} + \text{Wert}_{IN4} \tag{2.106}$$

Das berechnete Ergebnis wird an den Ausgangs-Adapter `OUT` übergeben und gleichzeitig wird das zugehörige Ereignis am Ausgangs-Plug getriggert, um nachfolgende Bausteine über die Aktualisierung des Werts zu informieren.

Technische Besonderheiten

- **Generischer Baustein:** Der FB basiert auf der generischen Klasse `GEN_AULI_ADD`. Das bedeutet, dass er flexibel auf verschiedene Datentypen innerhalb der `AULI`-Adapterstruktur reagieren kann.
- **Unidirektionale Adapter:** Es werden unidirektionale Adapter (`unidirectional::AULI`) verwendet. Dies vereinfacht den Datenfluss, da die Informationen und Ereignisse ausschließlich in eine Richtung (von den Eingängen zum Ausgang) fließen.
- **Reduzierter Verdrahtungsaufwand:** Durch die Verwendung von Adaptern müssen Ereignis- und Datenleitungen nicht einzeln gezogen werden. Ein einziger Adapter-Kanal bündelt alle relevanten Signale.

Zustandsübersicht

Der Funktionsbaustein verhält sich ereignisgesteuert und zustandslos (bzw. besitzt keinen internen Speicher für historische Werte):

1. **Wartezustand:** Der FB wartet auf ein Ereignis an einem der Eingänge (`IN1` bis `IN4`).
2. **Berechnung:** Bei Eintreffen eines Ereignisses werden die Daten aller vier Eingänge summiert.
3. **Ausgabe:** Der berechnete Wert wird an `OUT` angelegt und das Ausgabeereignis ausgelöst. Der FB kehrt sofort in den Wartezustand zurück.

Anwendungsszenarien

- **Messwert-Summierung:** Zusammenfassung von vier einzelnen Sensorwerten (z. B. Energiemessungen von vier Verbrauchern, Durchflussmengen von vier Rohren) zu einem Gesamtwert.
- **Mittelwertvorbereitung:** Vorsummierung von vier Datenpunkten vor einer anschließenden Division zur Mittelwertbildung.
- **Modulare Steuerungsarchitekturen:** Einsatz in komplexen Systemen, in denen Signale bereits standardisiert als `AULI`-Adapter vorliegen.

Vergleich mit ähnlichen Bausteinen

- **Standard-ADD-Bausteine (z. B. `ADD` aus der IEC 61131-3):** Diese nutzen direkte Daten- und Ereignisports (wie `REQ`, `CNF`, `IN1`, `IN2`). `AULI_ADD_4` hingegen kapselt diese Schnittstellen vollständig in Adaptern, was zu einem aufgeräumteren Control-Flow-Diagramm führt.
- **AULI_ADD_2:** Ein ähnlicher Baustein, der jedoch nur zwei Eingänge besitzt. `AULI_ADD_4` spart bei der Addition von vier Werten die Kaskadierung mehrerer einzelner Additions-Bausteine ein.
- `AULI_ADD_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

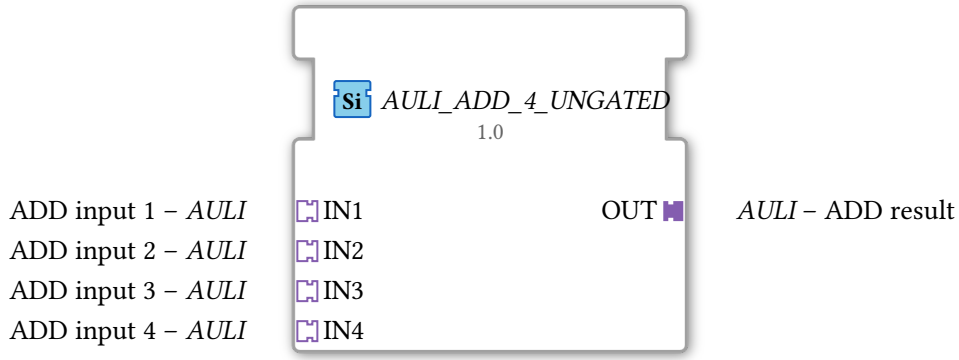
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `AULI_ADD_4`-Funktionsbaustein ist ein hocheffizienter, übersichtlicher und moderner Baustein für arithmetische Operationen in IEC 61499. Durch die konsequente Nutzung der `AULI`-Adapterstruktur sorgt er für eine signifikante Reduzierung von Verbindungslinien im Applikationseditor der 4diac IDE und eignet sich hervorragend für modulare Automatisierungslösungen.

AULI_ADD_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AULI_ADD_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild vorhanden)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AULI_ADD_4_UNGATED` ist ein generischer Funktionsbaustein (FB), der für die mathematische Addition von vier Werten konzipiert ist. Die Besonderheit dieses Bausteins liegt in der Verwendung von Adaptern des Typs `AULI` (unidirektional). Durch die Kapselung der Daten- und Ereignisübertragung in Adaptern ermöglicht der Baustein eine übersichtliche und modulare Verdrahtung innerhalb von IEC 61499-Anwendungen in der 4diac IDE.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung erfolgt implizit über die angeschlossenen Adapter-Sockets.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge vorhanden. Die Ereignissteuerung erfolgt implizit über den angeschlossenen Adapter-Plug.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübergabe erfolgt über die Eingangs-Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Die Datenausgabe erfolgt über den Ausgangs-Adapter.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AULI`): Erster Summand für die Additionsoperation.

- **IN2** (Typ: `adapter::types::unidirectional::AULI`): Zweiter Summand für die Additionsoperation.
- **IN3** (Typ: `adapter::types::unidirectional::AULI`): Dritter Summand für die Additionsoperation.
- **IN4** (Typ: `adapter::types::unidirectional::AULI`): Vierter Summand für die Additionsoperation.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AULI`): Enthält das Ergebnis der Addition (`IN1 + IN2 + IN3 + IN4`) sowie das dazugehörige Ausgabe-Ereignis.

Funktionsweise

Sobald an einem oder mehreren der Eingangs-Adapter (`IN1` bis `IN4`) ein neues Wert-Ereignis signalisiert wird, liest der Funktionsbaustein die aktuellen Werte aus allen vier Adaptern aus. Diese Werte werden mathematisch addiert:

$$\text{Ergebnis} = \text{Wert}_{IN1} + \text{Wert}_{IN2} + \text{Wert}_{IN3} + \text{Wert}_{IN4} \tag{2.107}$$

Das berechnete Ergebnis wird an den Ausgangs-Adapter `OUT` übergeben und gleichzeitig wird das zugehörige Ereignis am Ausgangs-Plug getriggert, um nachfolgende Bausteine über die Aktualisierung des Werts zu informieren.

Technische Besonderheiten

- **Generischer Baustein:** Der FB basiert auf der generischen Klasse `GEN_AULI_ADD`. Das bedeutet, dass er flexibel auf verschiedene Datentypen innerhalb der `AULI`-Adapterstruktur reagieren kann.
- **Unidirektionale Adapter:** Es werden unidirektionale Adapter (`unidirectional::AULI`) verwendet. Dies vereinfacht den Datenfluss, da die Informationen und Ereignisse ausschließlich in eine Richtung (von den Eingängen zum Ausgang) fließen.
- **Reduzierter Verdrahtungsaufwand:** Durch die Verwendung von Adaptern müssen Ereignis- und Datenleitungen nicht einzeln gezogen werden. Ein einziger Adapter-Kanal bündelt alle relevanten Signale.

Zustandsübersicht

Der Funktionsbaustein verhält sich ereignisgesteuert und zustandslos (bzw. besitzt keinen internen Speicher für historische Werte):

1. **Wartezustand:** Der FB wartet auf ein Ereignis an einem der Eingänge (`IN1` bis `IN4`).
2. **Berechnung:** Bei Eintreffen eines Ereignisses werden die Daten aller vier Eingänge summiert.
3. **Ausgabe:** Der berechnete Wert wird an `OUT` angelegt und das Ausgabeereignis ausgelöst. Der FB kehrt sofort in den Wartezustand zurück.

Anwendungsszenarien

- **Messwert-Summierung:** Zusammenfassung von vier einzelnen Sensorwerten (z. B. Energiemessungen von vier Verbrauchern, Durchflussmengen von vier Rohren) zu einem Gesamtwert.
- **Mittelwertvorbereitung:** Vorsummierung von vier Datenpunkten vor einer anschließenden Division zur Mittelwertbildung.
- **Modulare Steuerungsarchitekturen:** Einsatz in komplexen Systemen, in denen Signale bereits standardisiert als `AULI`-Adapter vorliegen.

Vergleich mit ähnlichen Bausteinen

- **Standard-ADD-Bausteine (z. B. `ADD` aus der IEC 61131-3):** Diese nutzen direkte Daten- und Ereignisports (wie `REQ`, `CNF`, `IN1`, `IN2`). `AULI_ADD_4_UNGATED` hingegen kapselt diese Schnittstellen vollständig in Adaptern, was zu einem aufgeräumteren Control-Flow-Diagramm führt.
- **AULI_ADD_2:** Ein ähnlicher Baustein, der jedoch nur zwei Eingänge besitzt. `AULI_ADD_4_UNGATED` spart bei der Addition von vier Werten die Kaskadierung mehrerer einzelner Additions-Bausteine ein.
- **`AULI_ADD_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

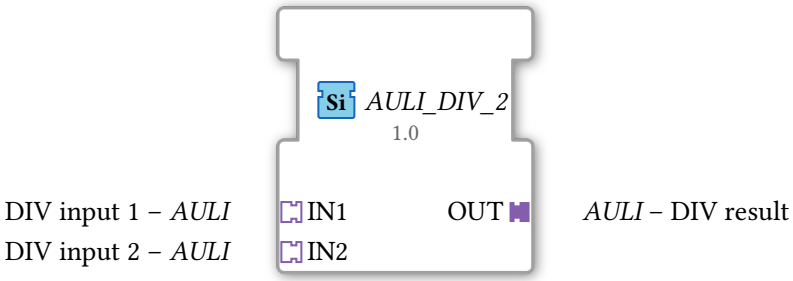
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AULI_ADD_4_UNGATED`-Funktionsbaustein ist ein hocheffizienter, übersichtlicher und moderner Baustein für arithmetische Operationen in IEC 61499. Durch die konsequente Nutzung der `AULI`-Adapterstruktur sorgt er für eine signifikante Reduzierung von Verbindungslinien im Applikationseditor der 4diac IDE und eignet sich hervorragend für modulare Automatisierungslösungen.

AULI_DIV_2



Kein Bild vorhanden

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AULI_DIV_2` ist ein generischer Funktionsbaustein (Generic FB), der für die Durchführung einer arithmetischen Division (DIV) entwickelt wurde. Er basiert auf der IEC 61499-Norm und nutzt das Konzept von Adaptern, um Daten und die dazugehörige Steuerungslogik effizient zu übertragen. Der Baustein ist im Namensraum `adapter::iec61131::arithmetic` definiert und verwendet die generische Klasse `GEN_AULI_DIV`.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten, dedizierten Ereignis-Eingänge. Die Steuerung des Signalflusses und die Ereignisverarbeitung werden vollständig über die angeschlossenen Adapter abgewickelt.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten, dedizierten Ereignis-Ausgänge. Die Ereignisweiterleitung erfolgt gekoppelt über den Ausgangs-Adapter.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Werte für die Berechnung werden über die Eingangs-Adapter bereitgestellt.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Das Berechnungsergebnis wird über den Ausgangs-Adapter bereitgestellt.

Adapter

Die gesamte Kommunikation des Bausteins wird über Adapter vom Typ `AULI` (unidirektional) realisiert:

- **IN1 (Socket / Buchse)**:
  - **Typ**: `adapter::types::unidirectional::AULI`
  - **Beschreibung**: Erster Eingang der Division (Dividend / Zähler).
- **IN2 (Socket / Buchse)**:
  - **Typ**: `adapter::types::unidirectional::AULI`
  - **Beschreibung**: Zweiter Eingang der Division (Divisor / Nenner).
- **OUT (Plug / Stecker)**:
  - **Typ**: `adapter::types::unidirectional::AULI`
  - **Beschreibung**: Ausgang, der das Ergebnis der Division (Quotient) liefert.

Funktionsweise

Der Baustein führt eine mathematische Division der über die Adapter bereitgestellten Eingangswerte aus. Die Formel lautet:

$$\mathrm{OUT} = \frac{\mathrm{IN1}}{\mathrm{IN2}} \tag{2.108}$$

Da es sich um einen generischen Funktionsbaustein (`GEN_AULI_DIV`) handelt, kann er sich zur Laufzeit oder während der Instanziierung in der 4diac-IDE an verschiedene numerische Datentypen anpassen, sofern diese vom Typ `AULI` unterstützt werden. Die Ausführung der Berechnung wird typischerweise durch ein Aktualisierungsereignis auf einem der Eingangs-Adapter (`IN1` oder `IN2`) getriggert. Das Ergebnis wird anschließend am Ausgangs-Adapter `OUT` zusammen mit einem entsprechenden Ausgabeereignis bereitgestellt.

Technische Besonderheiten

- **Generische Natur**: Durch das Attribut `GenericClassName` mit dem Wert `'GEN_AULI_DIV'` ist der Baustein flexibel einsetzbar und nicht starr auf einen einzigen Datentyp (wie z. B. nur `REAL` oder `INT`) festgelegt.
- **Kapselung durch Adapter**: Da keine klassischen Event- und Data-Ports genutzt werden, bleibt das visuelle Design im Funktionsplan (FBD) sehr übersichtlich. Die logische Zusammengehörigkeit von Daten und Triggern ist im `AULI`-Adapter gekapselt.
- **Unidirektionale Adapter**: Die Verwendung von `unidirectional::AULI` deutet darauf hin, dass der Informationsfluss strikt in eine Richtung (von den Sockets zum Baustein und vom Baustein zum Plug) verläuft.

Zustandsübersicht

Da es sich bei diesem Baustein um einen grundlegenden arithmetischen Baustein handelt, besitzt er in der Regel ein einfaches, zustandsloses Verhalten (bzw. verhält sich wie ein klassischer Execution Control Chart (ECC) mit einem Standard-Zustand, der auf Eingangsereignisse reagiert):

1. **Warten auf Ereignis**: Der Baustein wartet auf ein Aktualisierungsereignis an `IN1` oder `IN2`.
2. **Berechnung**: Bei Eintreffen eines Ereignisses wird die Division durchgeführt.
3. **Ergebnisübergabe**: Der berechnete Wert wird an `OUT` angelegt und das entsprechende Ausgangsereignis am Plug getriggert.
4. **Rückkehr**: Der Baustein kehrt in den Wartezustand zurück.

Hinweis zur Division durch Null: Je nach Implementierung der Laufzeitumgebung (Runtime) für den generischen Typ `GEN_AULI_DIV` sollten Mechanismen zur Vermeidung einer Division durch Null (z. B. Ausgabe von `NaN`, Unendlich oder das Abfangen über den Adapterstatus) beachtet werden.

Anwendungsszenarien

- **Skalierung von Sensorwerten**: Division von analogen Rohwerten durch einen Skalierungsfaktor zur Umrechnung in physikalische Einheiten.
- **Verhältnismessung**: Berechnung von Verhältnissen in der Prozessautomatisierung (z. B. Durchflussverhältnis von zwei Medien).
- **Mittelwertbildung**: Verwendung in Berechnungsnetzwerken, bei denen Summenwerte durch eine Anzahl geteilt werden müssen.
- **Strukturierte Architekturen**: Einsatz in Systemen, die konsequent auf Adapter-basierte Kommunikation setzen, um die Anzahl der sichtbaren Verbindungslinien im Systemmodell zu minimieren.

Vergleich mit ähnlichen Bausteinen

- **Standard `DIV`-Baustein**: Ein klassischer IEC 61131-3 / IEC 61499 `DIV`-Baustein nutzt direkte Dateneingänge (`IN1`, `IN2`) und Event-Ports (`REQ`, `CNF`). Der `AULI_DIV_2` hingegen bündelt diese Signale in Adaptern, was die Verdrahtung bei komplexen Signalstrukturen erheblich vereinfacht.
- **Andere AULI-Arithmetikbausteine** (z. B. `AULI_ADD`, `AULI_SUB`): Nutzen dieselbe Schnittstellenphilosophie mit Sockets und Plugs, führen jedoch andere mathematische Operationen aus.
- **`AULI_DIV_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

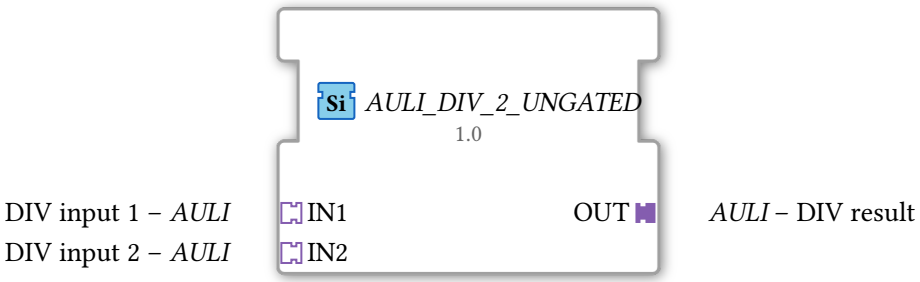
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `AULI_DIV_2` ist ein moderner, modularer Funktionsbaustein für arithmetische Divisionen in Eclipse 4diac. Durch die konsequente Nutzung von unidirektionalen Adaptern fördert er ein sauberes und übersichtliches Softwaredesign in verteilten Steuerungssystemen nach IEC 61499, indem er Daten und Steuerereignisse elegant bündelt.

AULI_DIV_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AULI_DIV_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



Kein Bild vorhanden

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AULI_DIV_2_UNGATED` ist ein generischer Funktionsbaustein (Generic FB), der für die Durchführung einer arithmetischen Division (DIV) entwickelt wurde. Er basiert auf der IEC 61499-Norm und nutzt das Konzept von Adaptern, um Daten und die dazugehörige Steuerungslogik effizient zu übertragen. Der Baustein ist im Namensraum `adapter::iec61131::arithmetic` definiert und verwendet die generische Klasse `GEN_AULI_DIV`.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten, dedizierten Ereignis-Eingänge. Die Steuerung des Signalflusses und die Ereignisverarbeitung werden vollständig über die angeschlossenen Adapter abgewickelt.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten, dedizierten Ereignis-Ausgänge. Die Ereignisweiterleitung erfolgt gekoppelt über den Ausgangs-Adapter.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Werte für die Berechnung werden über die Eingangs-Adapter bereitgestellt.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Das Berechnungsergebnis wird über den Ausgangs-Adapter bereitgestellt.

Adapter

Die gesamte Kommunikation des Bausteins wird über Adapter vom Typ `AULI` (unidirektional) realisiert:

- **IN1 (Socket / Buchse)**:
  - **Typ**: `adapter::types::unidirectional::AULI`
  - **Beschreibung**: Erster Eingang der Division (Dividend / Zähler).
- **IN2 (Socket / Buchse)**:
  - **Typ**: `adapter::types::unidirectional::AULI`
  - **Beschreibung**: Zweiter Eingang der Division (Divisor / Nenner).
- **OUT (Plug / Stecker)**:
  - **Typ**: `adapter::types::unidirectional::AULI`
  - **Beschreibung**: Ausgang, der das Ergebnis der Division (Quotient) liefert.

Funktionsweise

Der Baustein führt eine mathematische Division der über die Adapter bereitgestellten Eingangswerte aus. Die Formel lautet:

$$\mathrm{OUT} = \frac{\mathrm{IN1}}{\mathrm{IN2}} \tag{2.109}$$

Da es sich um einen generischen Funktionsbaustein (`GEN_AULI_DIV`) handelt, kann er sich zur Laufzeit oder während der Instanziierung in der 4diac-IDE an verschiedene numerische Datentypen anpassen, sofern diese vom Typ `AULI` unterstützt werden. Die Ausführung der Berechnung wird typischerweise durch ein Aktualisierungsereignis auf einem der Eingangs-Adapter (`IN1` oder `IN2`) getriggert. Das Ergebnis wird anschließend am Ausgangs-Adapter `OUT` zusammen mit einem entsprechenden Ausgabeereignis bereitgestellt.

Technische Besonderheiten

- **Generische Natur**: Durch das Attribut `GenericClassName` mit dem Wert `'GEN_AULI_DIV'` ist der Baustein flexibel einsetzbar und nicht starr auf einen einzigen Datentyp (wie z. B. nur `REAL` oder `INT`) festgelegt.
- **Kapselung durch Adapter**: Da keine klassischen Event- und Data-Ports genutzt werden, bleibt das visuelle Design im Funktionsplan (FBD) sehr übersichtlich. Die logische Zusammengehörigkeit von Daten und Triggern ist im `AULI`-Adapter gekapselt.
- **Unidirektionale Adapter**: Die Verwendung von `unidirectional::AULI` deutet darauf hin, dass der Informationsfluss strikt in eine Richtung (von den Sockets zum Baustein und vom Baustein zum Plug) verläuft.

Zustandsübersicht

Da es sich bei diesem Baustein um einen grundlegenden arithmetischen Baustein handelt, besitzt er in der Regel ein einfaches, zustandsloses Verhalten (bzw. verhält sich wie ein klassischer Execution Control Chart (ECC) mit einem Standard-Zustand, der auf Eingangsereignisse reagiert):

1. **Warten auf Ereignis**: Der Baustein wartet auf ein Aktualisierungsereignis an `IN1` oder `IN2`.
2. **Berechnung**: Bei Eintreffen eines Ereignisses wird die Division durchgeführt.
3. **Ergebnisübergabe**: Der berechnete Wert wird an `OUT` angelegt und das entsprechende Ausgangsereignis am Plug getriggert.
4. **Rückkehr**: Der Baustein kehrt in den Wartezustand zurück.

Hinweis zur Division durch Null: Je nach Implementierung der Laufzeitumgebung (Runtime) für den generischen Typ `GEN_AULI_DIV` sollten Mechanismen zur Vermeidung einer Division durch Null (z. B. Ausgabe von `NaN`, Unendlich oder das Abfangen über den Adapterstatus) beachtet werden.

Anwendungsszenarien

- **Skalierung von Sensorwerten**: Division von analogen Rohwerten durch einen Skalierungsfaktor zur Umrechnung in physikalische Einheiten.
- **Verhältnismessung**: Berechnung von Verhältnissen in der Prozessautomatisierung (z. B. Durchflussverhältnis von zwei Medien).
- **Mittelwertbildung**: Verwendung in Berechnungsnetzwerken, bei denen Summenwerte durch eine Anzahl geteilt werden müssen.

- **Strukturierte Architekturen**: Einsatz in Systemen, die konsequent auf Adapter-basierte Kommunikation setzen, um die Anzahl der sichtbaren Verbindungslinien im Systemmodell zu minimieren.

Vergleich mit ähnlichen Bausteinen

- **Standard `DIV`-Baustein**: Ein klassischer IEC 61131-3 / IEC 61499 `DIV`-Baustein nutzt direkte Dateneingänge (`IN1`, `IN2`) und Event-Ports (`REQ`, `CNF`). Der `AULI_DIV_2_UNGATED` hingegen bündelt diese Signale in Adaptern, was die Verdrahtung bei komplexen Signalstrukturen erheblich vereinfacht.
- **Andere AULI-Arithmetikbausteine** (z. B. `AULI_ADD`, `AULI_SUB`): Nutzen dieselbe Schnittstellenphilosophie mit Sockets und Plugs, führen jedoch andere mathematische Operationen aus.
- **`AULI_DIV_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- **`AULI_DIV_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

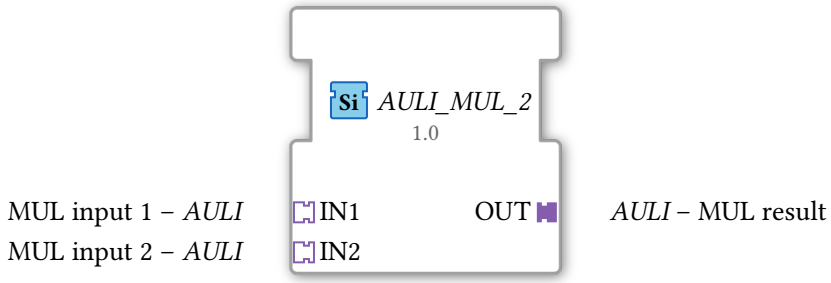
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AULI_DIV_2_UNGATED` ist ein moderner, modularer Funktionsbaustein für arithmetische Divisionen in Eclipse 4diac. Durch die konsequente Nutzung von unidirektionalen Adaptern fördert er ein sauberes und übersichtliches Softwaredesign in verteilten Steuerungssystemen nach IEC 61499, indem er Daten und Steuerereignisse elegant bündelt.

AULI_MUL_2



Kein Bild verfügbar

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AULI_MUL_2` dient zur Durchführung einer arithmetischen Multiplikation von zwei Werten. Es handelt sich hierbei um einen generischen Funktionsbaustein (`GEN_AULI_MUL`), dessen Schnittstellen vollständig über Adapter des Typs `AULI` (unidirektional) realisiert sind. Dies ermöglicht eine saubere Kapselung von Daten und Events und reduziert den Verdrahtungsaufwand innerhalb der 4diac-IDE.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung erfolgt implizit über die Adapter-Schnittstellen.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge vorhanden. Die Ereignissteuerung erfolgt implizit über die Adapter-Schnittstellen.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübergabe erfolgt über die Eingangs-Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Die Datenausgabe erfolgt über den Ausgangs-Adapter.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AULI`):
  - Erster Eingangswert (Multiplikand) für die Multiplikation.
- **IN2** (Typ: `adapter::types::unidirectional::AULI`):
  - Zweiter Eingangswert (Multiplikator) für die Multiplikation.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AULI`):
  - Ergebnis der Multiplikation ($OUT = IN1 \times IN2$).

Funktionsweise

Sobald an den Eingangs-Adaptern `IN1` oder `IN2` neue Daten signalisiert werden, führt der Baustein die Multiplikation der beiden Werte aus:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \tag{2.110}$$

Das Ergebnis sowie das zugehörige Aktualisierungsereignis werden anschließend über den Ausgangs-Adapter `OUT` ausgegeben. Da es sich um einen generischen Baustein handelt, richtet sich der genaue Datentyp nach der Definition des verwendeten `AULI`-Adapters (z. B. REAL, INT, LREAL).

Technische Besonderheiten

- **Generischer Baustein:** Der Typ ist als `GEN_AULI_MUL` deklariert, was eine flexible Wiederverwendbarkeit für verschiedene numerische Datentypen ermöglicht, sofern diese vom Adaptertyp unterstützt werden.
- **Kapselung durch Adapter:** Durch die Verwendung von unidirektionalen Adaptern (`AULI`) werden Daten und das dazugehörige Trigger-Event in einer einzigen Verbindung gebündelt. Dies verhindert „Spaghetti-Code“ und lose Event-Daten-Paare im Funktionsplan.
- **Package-Zugehörigkeit:** Der Baustein ist im Namensraum `adapter::iec61131::arithmetic` organisiert.

Zustandsübersicht

Da es sich um einen reinen Rechenbaustein (arithmetische Funktion) handelt, besitzt der Baustein keinen internen Zustandsautomaten (ECC) im klassischen Sinne.

1. **Wartend:** Der Baustein wartet auf ein Aktualisierungsereignis an den Adaptern `IN1` oder `IN2`.
2. **Berechnung:** Bei Eintreffen eines Events werden die Werte aus `IN1` und `IN2` ausgelesen und multipliziert.
3. **Ausgabe:** Das Ergebnis wird an `OUT` angelegt und das Ausgangs-Event des Adapters getriggert. Der Baustein kehrt sofort in den Zustand *Wartend* zurück.

Anwendungsszenarien

- **Signalverarbeitung:** Skalierung von Sensorwerten, bei denen ein Messwert mit einem konstanten oder dynamischen Faktor multipliziert werden muss.
- **Regelungstechnik:** Berechnung von Stellgrößen in Regelkreisen (z. B. P-Anteil eines PID-Reglers).
- **Strukturierte Steuerungsarchitekturen:** Einsatz in komplexen Systemen, in denen einheitlich mit Adaptern gearbeitet wird, um die Lesbarkeit des Gesamtsystems zu wahren.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-Multiplikationsbaustein (wie dem IEC 61199 `MUL`-Baustein) bietet der `AULI_MUL_2` folgende Vor- und Nachteile:

Merkmal	Standard MUL Baustein	AULI_MUL_2
Schnittstellen	Einzelne Event- und Datenpins	Gebündelte Adapter (AULI)
Verdrahtungsaufwand	Hoch (separate Event- und Datenlinien)	Minimal (nur eine Adapterlinie pro Signal)
Flexibilität	Hoch (direktes Mapping von Variablen)	Erfordert die Verwendung des AULI-Adaptertyps
Lesbarkeit	Kann bei vielen Bausteinen unübersichtlich werden	Sehr sauber und modular

- AULI_MUL_2_UNGATED: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

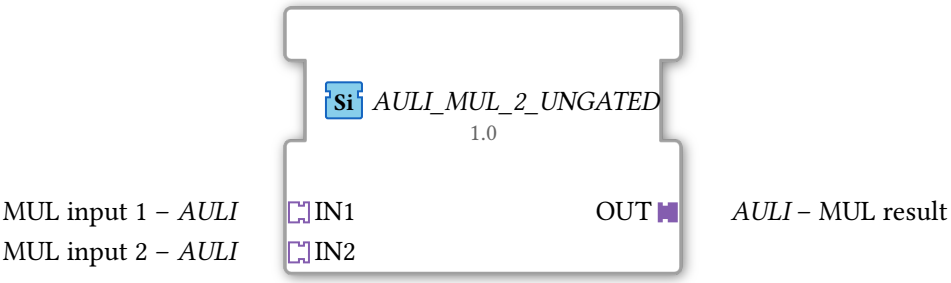
Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der AULI_MUL_2 ist eine spezialisierte und moderne Variante eines Multiplikationsbausteins für die IEC 61499. Durch die konsequente Nutzung von Adaptern eignet er sich hervorragend für anspruchsvolle, modular aufgebaute Applikationen, bei denen Übersichtlichkeit und Standardisierung im Vordergrund stehen.

AULI_MUL_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AULI_MUL_2. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



Kein Bild verfügbar

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein AULI_MUL_2_UNGATED dient zur Durchführung einer arithmetischen Multiplikation von zwei Werten. Es handelt sich hierbei um einen generischen Funktionsbaustein (GEN_AULI_MUL), dessen Schnittstellen vollständig über Adapter des Typs AULI (unidirektional) realisiert sind. Dies ermöglicht eine saubere Kapselung von Daten und Events und reduziert den Verdrahtungsaufwand innerhalb der 4diac-IDE.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung erfolgt implizit über die Adapter-Schnittstellen.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge vorhanden. Die Ereignissteuerung erfolgt implizit über die Adapter-Schnittstellen.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübergabe erfolgt über die Eingangs-Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Die Datenausgabe erfolgt über den Ausgangs-Adapter.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AULI`):
  - Erster Eingangswert (Multiplikand) für die Multiplikation.
- **IN2** (Typ: `adapter::types::unidirectional::AULI`):
  - Zweiter Eingangswert (Multiplikator) für die Multiplikation.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AULI`):
  - Ergebnis der Multiplikation ($OUT = IN1 \times IN2$).

Funktionsweise

Sobald an den Eingangs-Adaptern `IN1` oder `IN2` neue Daten signalisiert werden, führt der Baustein die Multiplikation der beiden Werte aus:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \tag{2.111}$$

Das Ergebnis sowie das zugehörige Aktualisierungsereignis werden anschließend über den Ausgangs-Adapter `OUT` ausgegeben. Da es sich um einen generischen Baustein handelt, richtet sich der genaue Datentyp nach der Definition des verwendeten `AULI`-Adapters (z. B. REAL, INT, LREAL).

Technische Besonderheiten

- **Generischer Baustein:** Der Typ ist als `GEN_AULI_MUL` deklariert, was eine flexible Wiederverwendbarkeit für verschiedene numerische Datentypen ermöglicht, sofern diese vom Adaptertyp unterstützt werden.
- **Kapselung durch Adapter:** Durch die Verwendung von unidirektionalen Adaptern (`AULI`) werden Daten und das dazugehörige Trigger-Event in einer einzigen Verbindung gebündelt. Dies verhindert „Spaghetti-Code“ und lose Event-Daten-Paare im Funktionsplan.
- **Package-Zugehörigkeit:** Der Baustein ist im Namensraum `adapter::iec61131::arithmetic` organisiert.

Zustandsübersicht

Da es sich um einen reinen Rechenbaustein (arithmetische Funktion) handelt, besitzt der Baustein keinen internen Zustandsautomaten (ECC) im klassischen Sinne.

1. **Wartend:** Der Baustein wartet auf ein Aktualisierungsereignis an den Adaptern `IN1` oder `IN2`.
2. **Berechnung:** Bei Eintreffen eines Events werden die Werte aus `IN1` und `IN2` ausgelesen und multipliziert.
3. **Ausgabe:** Das Ergebnis wird an `OUT` angelegt und das Ausgangs-Event des Adapters getriggert. Der Baustein kehrt sofort in den Zustand *Wartend* zurück.

Anwendungsszenarien

- **Signalverarbeitung:** Skalierung von Sensorwerten, bei denen ein Messwert mit einem konstanten oder dynamischen Faktor multipliziert werden muss.
- **Regelungstechnik:** Berechnung von Stellgrößen in Regelkreisen (z. B. P-Anteil eines PID-Reglers).
- **Strukturierte Steuerungsarchitekturen:** Einsatz in komplexen Systemen, in denen einheitlich mit Adaptern gearbeitet wird, um die Lesbarkeit des Gesamtsystems zu wahren.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-Multiplikationsbaustein (wie dem IEC 61199 `MUL`-Baustein) bietet der `AULI_MUL_2_UNGATED` folgende Vor- und Nachteile:

Merkmal	Standard `MUL` Baustein	`AULI_MUL_2_UNGATED`
Schnittstellen	Einzelne Event- und Datenpins	Gebündelte Adapter (`AULI`)
Verdrahtungsaufwand	Hoch (separate Event- und Datenlinien)	Minimal (nur eine Adapterlinie pro Signal)
Flexibilität	Hoch (direktes Mapping von Variablen)	Erfordert die Verwendung des `AULI`-Adaptertyps
Lesbarkeit	Kann bei vielen Bausteinen unübersichtlich werden	Sehr sauber und modular

---

- `AULI_MUL_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

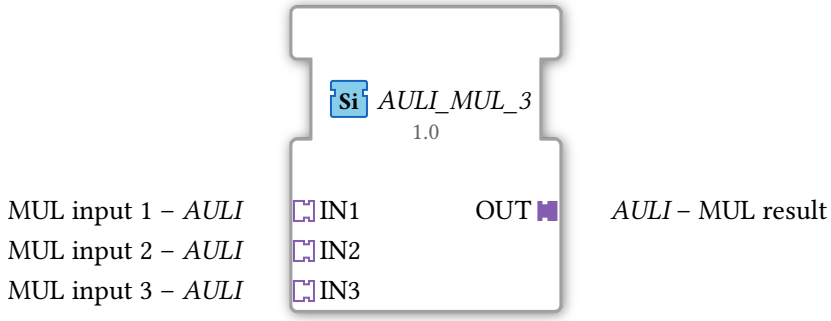
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AULI_MUL_2_UNGATED` ist eine spezialisierte und moderne Variante eines Multiplikationsbausteins für die IEC 61499. Durch die konsequente Nutzung von Adaptern eignet er sich hervorragend für anspruchsvolle, modular aufgebaute Applikationen, bei denen Übersichtlichkeit und Standardisierung im Vordergrund stehen.

AULI_MUL_3



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AULI_MUL_3` ist ein generischer Funktionsbaustein (Generic FB), der für die arithmetische Multiplikation von drei Eingangswerten über unidirektionale Adapter entwickelt wurde. Er ist im Package `adapter::iec61131::arithmetic` definiert und ermöglicht eine saubere, adapterbasierte Datenverarbeitung innerhalb der 4diac-Umgebung.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Eingänge. Die Steuerung und Ereignisverarbeitung erfolgt über die angebundenen Adapter.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Ausgänge. Die Weiterleitung von Ereignissen erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AULI`): Erster Eingangswert (Multiplikand 1) für die Multiplikation.
- **IN2** (Typ: `adapter::types::unidirectional::AULI`): Zweiter Eingangswert (Multiplikand 2) für die Multiplikation.
- **IN3** (Typ: `adapter::types::unidirectional::AULI`): Dritter Eingangswert (Multiplikand 3) für die Multiplikation.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AULI`): Ausgang für das berechnete Produkt der drei Eingangswerte.

Funktionsweise

Der Baustein `AULI_MUL_3` führt eine klassische Multiplikation der Werte durch, die über die drei Eingangs-Adapter (`IN1`, `IN2` und `IN3`) bereitgestellt werden. Die Berechnung folgt der mathematischen Formel:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \tag{2.112}$$

Da es sich um einen generischen Baustein (`GEN_AULI_MUL`) handelt, der auf unidirektionalen Adaptern (`AULI`) basiert, werden die Werte und die dazugehörigen Aktualisierungsereignisse innerhalb der Adapterverbindungen übertragen. Sobald an den Eingängen neue Daten anliegen und signalisiert werden, wird die Berechnung durchgeführt und das Ergebnis über den Ausgangs-Adapter `OUT` ausgegeben.

Technische Besonderheiten

- **Generischer Typ:** Der Baustein ist als generischer Typ deklariert (`GenericClassName = "GEN_AULI_MUL"`). Dies erlaubt eine flexible Handhabung je nach zugrundeliegendem Datentyp der verwendeten Adapter.
- **Adapter-Kapselung:** Durch die Verwendung von Adaptern des Typs `AULI` werden Daten und zugehörige Kontrollflüsse (Events) in einer einzigen Verbindung gebündelt, was den Verdrahtungsaufwand im Funktionsplan erheblich reduziert.
- **Unidirektionalität:** Die Schnittstellen nutzen das unidirektionale Profil, wodurch ein klarer Informationsfluss von den Eingängen zum Ausgang gewährleistet ist.

Zustandsübersicht

Der Baustein arbeitet ereignisgesteuert und besitzt keinen internen, persistenten Zustand (zustandslos / kombinatorisch).

1. **Warten auf Aktualisierung:** Der Baustein wartet auf ein Aktualisierungsereignis an den Eingangs-Sockets (`IN1`, `IN2` oder `IN3`).
2. **Berechnung:** Nach Erhalt eines Ereignisses werden die aktuellen Werte gelesen und multipliziert.

3. **Ausgabe:** Das Ergebnis wird an den Plug `OUT` übergeben und das entsprechende Ausgangsereignis ausgelöst.

Anwendungsszenarien

- **Volumenberechnung:** Berechnung von dreidimensionalen Größen (z. B. Länge × Breite × Höhe) in der Prozessautomatisierung.
- **Skalierung und Gewichtung:** Anwendung von zwei aufeinanderfolgenden Skalierungsfaktoren auf einen Messwert (z. B. Sensorwert × Kalibrierungsfaktor × Einheitenumrechnung).
- **Leistungsberechnung:** Dreiphasige oder mehrstufige Berechnungen, bei denen drei physikalische Größen multiplikativ verknüpft werden müssen.

Vergleich mit ähnlichen Bausteinen

- **Standard-MUL (IEC 61131-3):** Der klassische `MUL`-Baustein arbeitet mit direkten Daten- und Ereignispins. `AULI_MUL_3` unterscheidet sich dadurch, dass er genau drei Eingänge besitzt und diese vollständig über Adapter (`AULI`) kapselt, was zu einer übersichtlicheren Architektur führt.
- **AULI_MUL_2 (falls vorhanden):** Ein hypothetischer Baustein mit nur zwei Eingängen. `AULI_MUL_3` spart bei der Multiplikation von drei Werten einen kompletten Kaskadierungsbaustein ein.

- `AULI_MUL_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

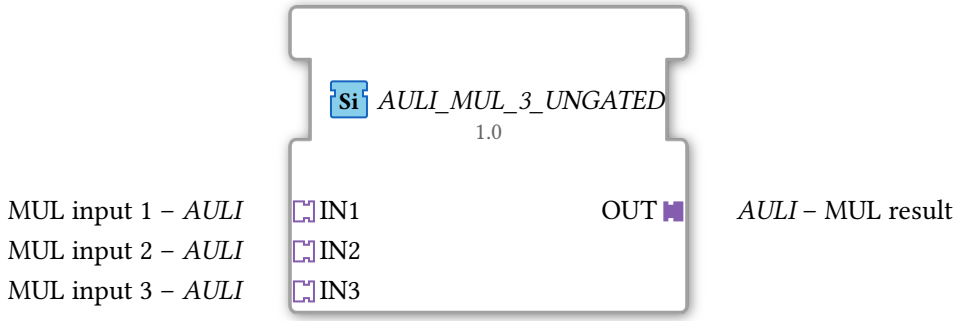
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

`AULI_MUL_3` ist ein effizienter Hilfsbaustein für Anwendungen, die ein hohes Maß an Modularität erfordern. Durch die konsequente Nutzung von unidirektionalen Adaptern trägt er zur Reduzierung von „Signal-Spaghetti“ in komplexen 4diac-Anwendungen bei und vereinfacht die Multiplikation von drei Variablen.

AULI_MUL_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AULI_MUL_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AULI_MUL_3_UNGATED` ist ein generischer Funktionsbaustein (Generic FB), der für die arithmetische Multiplikation von drei Eingangswerten über unidirektionale Adapter entwickelt wurde. Er ist im Package `adapter::iec61131::arithmetic` definiert und ermöglicht eine saubere, adapterbasierte Datenverarbeitung innerhalb der 4diac-Umgebung.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Eingänge. Die Steuerung und Ereignisverarbeitung erfolgt über die angebundenen Adapter.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Ausgänge. Die Weiterleitung von Ereignissen erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AULI`): Erster Eingangswert (Multiplikand 1) für die Multiplikation.
- **IN2** (Typ: `adapter::types::unidirectional::AULI`): Zweiter Eingangswert (Multiplikand 2) für die Multiplikation.
- **IN3** (Typ: `adapter::types::unidirectional::AULI`): Dritter Eingangswert (Multiplikand 3) für die Multiplikation.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AULI`): Ausgang für das berechnete Produkt der drei Eingangswerte.

Funktionsweise

Der Baustein `AULI_MUL_3_UNGATED` führt eine klassische Multiplikation der Werte durch, die über die drei Eingangs-Adapter (`IN1`, `IN2` und `IN3`) bereitgestellt werden. Die Berechnung folgt der mathematischen Formel:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \tag{2.113}$$

Da es sich um einen generischen Baustein (`GEN_AULI_MUL`) handelt, der auf unidirektionalen Adaptern (`AULI`) basiert, werden die Werte und die dazugehörigen Aktualisierungsereignisse innerhalb der Adapterverbindungen übertragen. Sobald an den Eingängen neue Daten anliegen und signalisiert werden, wird die Berechnung durchgeführt und das Ergebnis über den Ausgangs-Adapter `OUT` ausgegeben.

Technische Besonderheiten

- **Generischer Typ:** Der Baustein ist als generischer Typ deklariert (`GenericClassName = "GEN_AULI_MUL"`). Dies erlaubt eine flexible Handhabung je nach zugrundeliegendem Datentyp der verwendeten Adapter.
- **Adapter-Kapselung:** Durch die Verwendung von Adaptern des Typs `AULI` werden Daten und zugehörige Kontrollflüsse (Events) in einer einzigen Verbindung gebündelt, was den Verdrahtungsaufwand im Funktionsplan erheblich reduziert.
- **Unidirektionalität:** Die Schnittstellen nutzen das unidirektionale Profil, wodurch ein klarer Informationsfluss von den Eingängen zum Ausgang gewährleistet ist.

Zustandsübersicht

Der Baustein arbeitet ereignisgesteuert und besitzt keinen internen, persistenten Zustand (zustandslos / kombinatorisch).

1. **Warten auf Aktualisierung:** Der Baustein wartet auf ein Aktualisierungsereignis an den Eingangs-Sockets (`IN1`, `IN2` oder `IN3`).
2. **Berechnung:** Nach Erhalt eines Ereignisses werden die aktuellen Werte gelesen und multipliziert.
3. **Ausgabe:** Das Ergebnis wird an den Plug `OUT` übergeben und das entsprechende Ausgangsereignis ausgelöst.

Anwendungsszenarien

- **Volumenberechnung:** Berechnung von dreidimensionalen Größen (z. B. Länge × Breite × Höhe) in der Prozessautomatisierung.
- **Skalierung und Gewichtung:** Anwendung von zwei aufeinanderfolgenden Skalierungsfaktoren auf einen Messwert (z. B. Sensorwert × Kalibrierungsfaktor × Einheitenumrechnung).
- **Leistungsberechnung:** Dreiphasige oder mehrstufige Berechnungen, bei denen drei physikalische Größen multiplikativ verknüpft werden müssen.

Vergleich mit ähnlichen Bausteinen

- **Standard-MUL (IEC 61131-3):** Der klassische `MUL`-Baustein arbeitet mit direkten Daten- und Ereignispins. `AULI_MUL_3_UNGATED` unterscheidet sich dadurch, dass er genau drei Eingänge besitzt und diese vollständig über Adapter (`AULI`) kapselt, was zu einer übersichtlicheren Architektur führt.
- **AULI_MUL_2 (falls vorhanden):** Ein hypothetischer Baustein mit nur zwei Eingängen. `AULI_MUL_3_UNGATED` spart bei der Multiplikation von drei Werten einen kompletten Kaskadierungsbaustein ein.

- `AULI_MUL_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

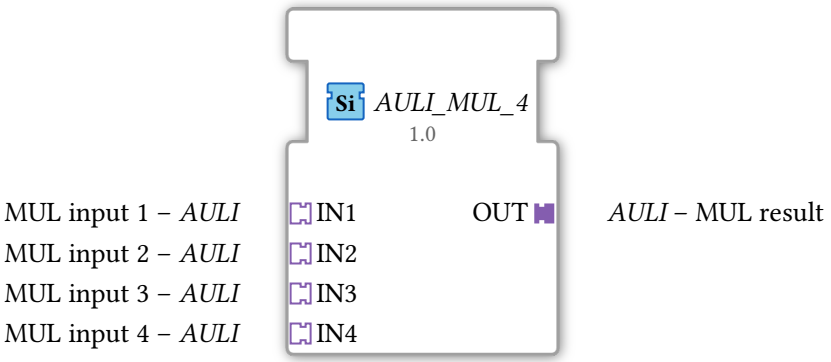
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

`AULI_MUL_3_UNGATED` ist ein effizienter Hilfsbaustein für Anwendungen, die ein hohes Maß an Modularität erfordern. Durch die konsequente Nutzung von unidirektionalen Adaptern trägt er zur Reduzierung von „Signal-Spaghetti“ in komplexen 4diac-Anwendungen bei und vereinfacht die Multiplikation von drei Variablen.

AULI_MUL_4



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AULI_MUL_4` ist ein generischer arithmetischer Baustein für die 4diac-ide. Er dient zur Multiplikation von vier Eingangswerten. Der Baustein nutzt ein adapterbasiertes Schnittstellenkonzept, um die Anzahl der einzelnen Event- und Datenverbindungen im Applikationsdiagramm zu minimieren und eine saubere Kapselung zu gewährleisten.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung wird vollständig über die Adapter abgewickelt.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge vorhanden. Die Ereignissteuerung wird vollständig über die Adapter abgewickelt.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübertragung erfolgt gekapselt über die Eingangs-Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Die Datenübertragung erfolgt gekapselt über den Ausgangs-Adapter.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AULI`): Erster Multiplikand.
- **IN2** (Typ: `adapter::types::unidirectional::AULI`): Zweiter Multiplikand.
- **IN3** (Typ: `adapter::types::unidirectional::AULI`): Dritter Multiplikand.
- **IN4** (Typ: `adapter::types::unidirectional::AULI`): Vierter Multiplikand.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AULI`): Ergebnis der Multiplikation.

Funktionsweise

Sobald an den Eingangs-Adaptern (`IN1` bis `IN4`) neue Werte signalisiert werden, führt der Baustein die Multiplikation der vier Werte durch.

Die mathematische Formel lautet:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \times \mathrm{IN4} \tag{2.114}$$

Das Ergebnis sowie das zugehörige Verarbeitungsereignis werden anschließend über den Ausgangs-Adapter `OUT` ausgegeben. Da es sich um unidirektionale Adapter des Typs `AULI` handelt, fließen Daten und Trigger-Signale gerichtet von den Sockets zum Plug.

Technische Besonderheiten

- **Generischer Baustein:** Der Baustein ist als generischer Typ deklariert (`GenericClassName = 'GEN_AULI_MUL'`). Dies ermöglicht eine flexible Anpassung an unterschiedliche numerische Datentypen, die vom zugrundeliegenden `AULI`-Adaptertyp unterstützt werden.
- **Kapselung durch Adapter:** Durch die Verwendung von Adaptern anstelle von Standard-Event-/Daten-Pins wird das Systemdesign stark vereinfacht („Spaghetti-Verdrahtung“ im Control-Flow wird vermieden).

Zustandsübersicht

Der Funktionsbaustein verhält sich im Wesentlichen zustandslos (*stateless*). Die Berechnung erfolgt rein reaktiv auf Basis der an den Eingangs-Adaptern anliegenden Werte und Events. Es werden keine internen historischen Zustände gespeichert.

Anwendungsszenarien

- **Sensorwert-Skalierung:** Berechnung von korrigierten Messwerten, bei denen ein Rohwert mit mehreren Kalibrierungs-, Korrektur- oder Umrechnungsfaktoren multipliziert werden muss.
- **Volumen- und Massenberechnung:** Physikalische Berechnungen in der Prozesstechnik, die das Produkt mehrerer Variablen erfordern (z. B. $V = l \times b \times h$ unter Berücksichtigung eines zusätzlichen Dichtefaktors).
- **Strukturierte Signalverarbeitung:** Einsatz in komplexeren Steuerungsanwendungen, bei denen Daten modular über Adapterstrukturen verteilt werden.

Vergleich mit ähnlichen Bausteinen

- **Standard-MUL-Baustein (IEC 61131-3 / IEC 61499):** Standard-Multiplizierer besitzen meist dedizierte Pins wie `REQ`, `CNF` sowie klassische Daten-Eingänge (z. B. `IN1`, `IN2`). `AULI_MUL_4` vereinfacht das Schnittstellendesign durch die Nutzung von Adaptern erheblich.
- **Kaskadierte 2-fach Multiplizierer:** Um vier Werte mit Standardbausteinen zu multiplizieren, müssten drei herkömmliche `MUL`-Bausteine kaskadiert werden. `AULI_MUL_4` bündelt diese Logik in einem einzigen Baustein, was Ressourcen spart und die Übersichtlichkeit erhöht.
- `AULI_MUL_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

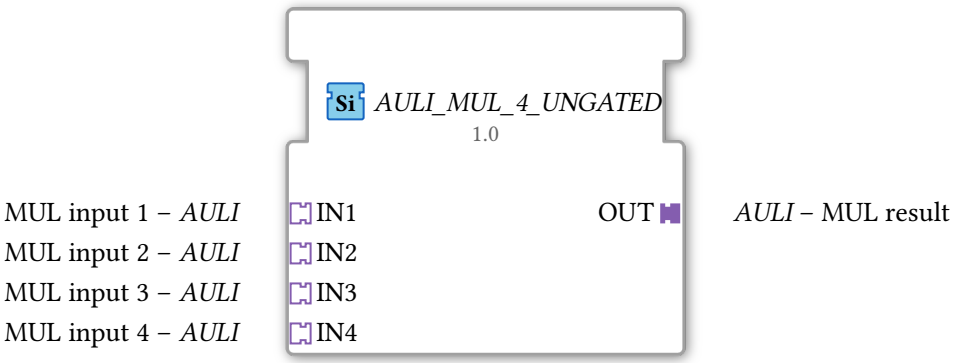
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `AULI_MUL_4` ist ein praktischer und moderner Funktionsbaustein zur Multiplikation von vier numerischen Werten unter Verwendung des IEC 61499 Adapterkonzepts. Er eignet sich hervorragend für sauber strukturierte, lesbare und wartungsfreundliche Steuerungsapplikationen in der 4diac-ide.

AULI_MUL_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AULI_MUL_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AULI_MUL_4_UNGATED` ist ein generischer arithmetischer Baustein für die 4diac-ide. Er dient zur Multiplikation von vier Eingangswerten. Der Baustein nutzt ein adapterbasiertes Schnittstellenkonzept, um die Anzahl der einzelnen Event- und Datenverbindungen im Applikationsdiagramm zu minimieren und eine saubere Kapselung zu gewährleisten.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung wird vollständig über die Adapter abgewickelt.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge vorhanden. Die Ereignissteuerung wird vollständig über die Adapter abgewickelt.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübertragung erfolgt gekapselt über die Eingangs-Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Die Datenübertragung erfolgt gekapselt über den Ausgangs-Adapter.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AULI`): Erster Multiplikand.
- **IN2** (Typ: `adapter::types::unidirectional::AULI`): Zweiter Multiplikand.
- **IN3** (Typ: `adapter::types::unidirectional::AULI`): Dritter Multiplikand.
- **IN4** (Typ: `adapter::types::unidirectional::AULI`): Vierter Multiplikand.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AULI`): Ergebnis der Multiplikation.

Funktionsweise

Sobald an den Eingangs-Adaptern (`IN1` bis `IN4`) neue Werte signalisiert werden, führt der Baustein die Multiplikation der vier Werte durch.

Die mathematische Formel lautet:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \times \mathrm{IN4} \tag{2.115}$$

Das Ergebnis sowie das zugehörige Verarbeitungsereignis werden anschließend über den Ausgangs-Adapter `OUT` ausgegeben. Da es sich um unidirektionale Adapter des Typs `AULI` handelt, fließen Daten und Trigger-Signale gerichtet von den Sockets zum Plug.

Technische Besonderheiten

- **Generischer Baustein:** Der Baustein ist als generischer Typ deklariert (`GenericClassName = 'GEN_AULI_MUL'`). Dies ermöglicht eine flexible Anpassung an unterschiedliche numerische Datentypen, die vom zugrundeliegenden `AULI`-Adaptertyp unterstützt werden.
- **Kapselung durch Adapter:** Durch die Verwendung von Adaptern anstelle von Standard-Event-/Daten-Pins wird das Systemdesign stark vereinfacht („Spaghetti-Verdrahtung“ im Control-Flow wird vermieden).

Zustandsübersicht

Der Funktionsbaustein verhält sich im Wesentlichen zustandslos (*stateless*). Die Berechnung erfolgt rein reaktiv auf Basis der an den Eingangs-Adaptern anliegenden Werte und Events. Es werden keine internen historischen Zustände gespeichert.

Anwendungsszenarien

- **Sensorwert-Skalierung:** Berechnung von korrigierten Messwerten, bei denen ein Rohwert mit mehreren Kalibrierungs-, Korrektur- oder Umrechnungsfaktoren multipliziert werden muss.
- **Volumen- und Massenberechnung:** Physikalische Berechnungen in der Prozesstechnik, die das Produkt mehrerer Variablen erfordern (z. B. $V = l \times b \times h$ unter Berücksichtigung eines zusätzlichen Dichtefaktors).
- **Strukturierte Signalverarbeitung:** Einsatz in komplexeren Steuerungsanwendungen, bei denen Daten modular über Adapterstrukturen verteilt werden.

Vergleich mit ähnlichen Bausteinen

- **Standard-MUL-Baustein (IEC 61131-3 / IEC 61499):** Standard-Multiplizierer besitzen meist dedizierte Pins wie `REQ`, `CNF` sowie klassische Daten-Eingänge (z. B. `IN1`, `IN2`). `AULI_MUL_4_UNGATED` vereinfacht das Schnittstellendesign durch die Nutzung von Adaptern erheblich.
- **Kaskadierte 2-fach Multiplizierer:** Um vier Werte mit Standardbausteinen zu multiplizieren, müssten drei herkömmliche `MUL`-Bausteine kaskadiert werden. `AULI_MUL_4_UNGATED` bündelt diese Logik in einem einzigen Baustein, was Ressourcen spart und die Übersichtlichkeit erhöht.
- `AULI_MUL_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- `AULI_MUL_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

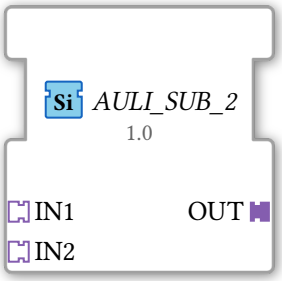
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AULI_MUL_4_UNGATED` ist ein praktischer und moderner Funktionsbaustein zur Multiplikation von vier numerischen Werten unter Verwendung des IEC 61499 Adapterkonzepts. Er eignet sich hervorragend für sauber strukturierte, lesbare und wartungsfreundliche Steuerungsapplikationen in der 4diac-ide.

AULI_SUB_2



SUB input 1 – *AULI*

SUB input 2 – *AULI*

AULI – SUB result

(Kein Bild vorhanden)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AULI_SUB_2` ist ein generischer Funktionsbaustein (Generic FB), der für die Durchführung einer arithmetischen Subtraktion (`OUT = IN1 - IN2`) entwickelt wurde. Anstatt klassischer Daten- und Ereignispins nutzt dieser Baustein Adapter-Schnittstellen des Typs `unidirectional::AULI`. Dies ermöglicht eine strukturierte und modularisierte Signalübertragung in IEC 61499 Applikationen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung wird vollständig über die Adapter abgewickelt.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Ereignissteuerung wird vollständig über die Adapter abgewickelt.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AULI`):
  - Beschreibung: Der erste Eingang (Minuend) für die Subtraktion.
- **IN2** (Typ: `adapter::types::unidirectional::AULI`):
  - Beschreibung: Der zweite Eingang (Subtrahend) für die Subtraktion.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AULI`):
  - Beschreibung: Der Ausgang (Differenz) der arithmetischen Subtraktion.

Funktionsweise

Sobald über die Eingangs-Adapter `IN1` und `IN2` Werte und die dazugehörigen Ereignisse eintreffen, führt der Baustein die mathematische Operation `OUT = IN1 - IN2` aus. Da es sich um einen generischen Baustein (`GEN_AULI_SUB`) handelt, kann sich die konkrete Datentyp-Verarbeitung nach den im Adapter definierten Datentypen richten (z. B. Ganzzahlen oder Gleitkommazahlen). Das Ergebnis wird zusammen mit dem entsprechenden Trigger-Ereignis über den Ausgangs-Adapter `OUT` weitergeleitet.

Technische Besonderheiten

- **Generischer Baustein:** Durch das Attribut `GenericClassName` mit dem Wert `'GEN_AULI_SUB'` ist der Baustein flexibel für verschiedene Datentypen einsetzbar, sofern die verwendeten Adapter dies unterstützen.
- **Unidirektionale Adapter:** Die Verwendung des Typs `AULI` sorgt für eine klare Trennung der Signalflussrichtungen (unidirektional) und reduziert den Verdrahtungsaufwand im 4diac-Editor drastisch.

Zustandsübersicht

Der Baustein verhält sich rein kombinatorisch bzw. ereignisgesteuert:

1. **Warten auf Input:** Der Baustein verbleibt im Ruhezustand, bis ein Ereignis an einem der Eingangs-Adapter (`IN1` oder `IN2`) signalisiert wird.
2. **Berechnung:** Bei Signaländerung/Ereignis wird die Subtraktion durchgeführt.
3. **Ausgabe:** Das Ergebnis wird unmittelbar an den Plug `OUT` übergeben und das Ausgangsereignis ausgelöst.

Anwendungsszenarien

- **Soll-Ist-Wert-Vergleich:** Berechnung von Regeldifferenzen in der Prozesstechnik (z. B. `Regeldifferenz = Sollwert - Istwert`).
- **Offset-Kompensation:** Abzug von Nullpunktfehlern oder Offsets bei analogen Sensorwerten.
- **Kaskadierte Berechnungen:** Mathematische Berechnungen in modularen Anlagensteuerungen, bei denen Messwerte strukturiert über Adapter transportiert werden.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-Subtraktionsbaustein (z. B. `SUB` aus der IEC 61131-3 Standardbibliothek) bietet `AULI_SUB_2` folgende Vorteile:

- **Reduzierte Komplexität:** Statt einzelner Leitungen für Event (REQ/CNF) und Daten (IN1, IN2, OUT) bündeln die `AULI`-Adapter alle relevanten Informationen in einer einzigen Verbindung.
- **Modularität:** Das Design eignet sich hervorragend für serviceorientierte Architekturen in verteilten Systemen.

- `AULI_SUB_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

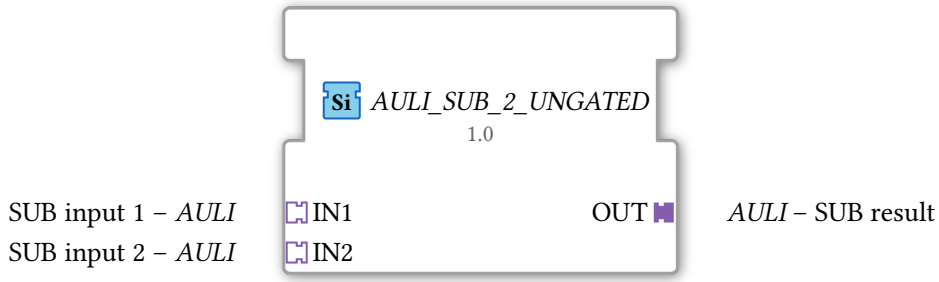
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `AULI_SUB_2` ist ein robuster und wiederverwendbarer Funktionsbaustein zur Subtraktion. Durch den konsequenten Einsatz von Adaptern fördert er ein sauberes, übersichtliches und wartungsfreundliches Applikationsdesign in der 4diac-IDE.

AULI_SUB_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AULI_SUB_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild vorhanden)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AULI_SUB_2_UNGATED` ist ein generischer Funktionsbaustein (Generic FB), der für die Durchführung einer arithmetischen Subtraktion (`OUT = IN1 - IN2`) entwickelt wurde. Anstatt klassischer Daten- und Ereignispins nutzt dieser Baustein Adapter-Schnittstellen des Typs `unidirectional::AULI`. Dies ermöglicht eine strukturierte und modularisierte Signalübertragung in IEC 61499 Applikationen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung wird vollständig über die Adapter abgewickelt.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Ereignissteuerung wird vollständig über die Adapter abgewickelt.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AULI`):
  - Beschreibung: Der erste Eingang (Minuend) für die Subtraktion.
- **IN2** (Typ: `adapter::types::unidirectional::AULI`):
  - Beschreibung: Der zweite Eingang (Subtrahend) für die Subtraktion.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AULI`):
  - Beschreibung: Der Ausgang (Differenz) der arithmetischen Subtraktion.

Funktionsweise

Sobald über die Eingangs-Adapter `IN1` und `IN2` Werte und die dazugehörigen Ereignisse eintreffen, führt der Baustein die mathematische Operation `OUT = IN1 - IN2` aus. Da es sich um einen generischen Baustein (`GEN_AULI_SUB`) handelt, kann sich die konkrete Datentyp-Verarbeitung nach den im Adapter definierten Datentypen richten (z. B. Ganzzahlen oder Gleitkommazahlen). Das Ergebnis wird zusammen mit dem entsprechenden Trigger-Ereignis über den Ausgangs-Adapter `OUT` weitergeleitet.

Technische Besonderheiten

- **Generischer Baustein:** Durch das Attribut `GenericClassName` mit dem Wert `'GEN_AULI_SUB'` ist der Baustein flexibel für verschiedene Datentypen einsetzbar, sofern die verwendeten Adapter dies unterstützen.
- **Unidirektionale Adapter:** Die Verwendung des Typs `AULI` sorgt für eine klare Trennung der Signalflussrichtungen (unidirektional) und reduziert den Verdrahtungsaufwand im 4diac-Editor drastisch.

Zustandsübersicht

Der Baustein verhält sich rein kombinatorisch bzw. ereignisgesteuert:

1. **Warten auf Input:** Der Baustein verbleibt im Ruhezustand, bis ein Ereignis an einem der Eingangs-Adapter (`IN1` oder `IN2`) signalisiert wird.
2. **Berechnung:** Bei Signaländerung/Ereignis wird die Subtraktion durchgeführt.
3. **Ausgabe:** Das Ergebnis wird unmittelbar an den Plug `OUT` übergeben und das Ausgangsereignis ausgelöst.

Anwendungsszenarien

- **Soll-Ist-Wert-Vergleich:** Berechnung von Regeldifferenzen in der Prozesstechnik (z. B. `Regeldifferenz = Sollwert - Istwert`).
- **Offset-Kompensation:** Abzug von Nullpunktfehlern oder Offsets bei analogen Sensorwerten.
- **Kaskadierte Berechnungen:** Mathematische Berechnungen in modularen Anlagensteuerungen, bei denen Messwerte strukturiert über Adapter transportiert werden.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-Subtraktionsbaustein (z. B. `SUB` aus der IEC 61131-3 Standardbibliothek) bietet `AULI_SUB_2_UNGATED` folgende Vorteile:

- **Reduzierte Komplexität:** Statt einzelner Leitungen für Event (REQ/CNF) und Daten (IN1, IN2, OUT) bündeln die `AULI`-Adapter alle relevanten Informationen in einer einzigen Verbindung.
- **Modularität:** Das Design eignet sich hervorragend für serviceorientierte Architekturen in verteilten Systemen.

- **AULI_SUB_2**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

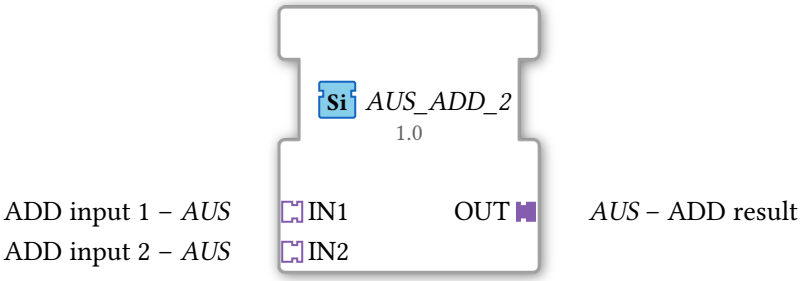
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AULI_SUB_2_UNGATED` ist ein robuster und wiederverwendbarer Funktionsbaustein zur Subtraktion. Durch den konsequenten Einsatz von Adaptern fördert er ein sauberes, übersichtliches und wartungsfreundliches Applikationsdesign in der 4diac-IDE.

AUS_ADD_2



(Grafische Darstellung des Funktionsblocks nicht verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AUS_ADD_2` ist ein generischer Baustein (Generic FB) zur Durchführung einer arithmetischen Addition. Er dient dazu, zwei Eingangssignale zu addieren und das Ergebnis auszugeben. Die Besonderheit dieses Bausteins liegt in der Verwendung von standardisierten, unidirektionalen Adaptern (`AUS`), wodurch die Signal- und Datenverdrahtung innerhalb von Eclipse 4diac stark vereinfacht und gekapselt wird.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Eingänge. Die Ereignissteuerung wird über die Adapter abgewickelt.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Ausgänge. Die Ereignissteuerung wird über die Adapter abgewickelt.

Daten-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Eingänge.

Daten-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Ausgänge.

Adapter

Sockets (Eingangs-Adapter / Buchsen)

- **IN1** (Typ: `adapter::types::unidirectional::AUS`): Der erste Summand der Addition.
- **IN2** (Typ: `adapter::types::unidirectional::AUS`): Der zweite Summand der Addition.

Plugs (Ausgangs-Adapter / Stecker)

- **OUT** (Typ: `adapter::types::unidirectional::AUS`): Das Ergebnis der Addition (Summe aus `IN1` und `IN2`).

Funktionsweise

Der Baustein AUS_ADD_2 arbeitet als arithmetischer Addierer auf Basis von Adaptern. Sobald an den Eingangs-Adaptern IN1 oder IN2 ein neues Ereignis samt Datenwert ankommt, führt der Baustein die Addition der beiden Werte durch:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} \tag{2.116}$$

Das Ergebnis wird zusammen mit dem entsprechenden Trigger-Ereignis über den Ausgangs-Adapter OUT ausgegeben. Da es sich um unidirektionale Adapter handelt, fließt der Daten- und Kontrollfluss ausschließlich von den Eingängen zum Ausgang.

Technische Besonderheiten

- **Generischer Baustein:** Durch das Attribut eclipse4diac::core::GenericClassName mit dem Wert 'GEN_AUS_ADD' ist der Baustein als generischer Typ definiert. Dies ermöglicht eine flexible Anpassung an unterschiedliche Datentypen, die durch die Adapterstrukturen definiert werden.
- **Kapselung:** Durch die Verwendung von Adaptern anstelle von einzelnen Event- und Data-Pins wird das Systemdesign übersichtlicher und weniger fehleranfällig für Verdrahtungsfehler.

Zustandsübersicht

Da es sich bei diesem Baustein im Wesentlichen um eine mathematische Verknüpfung handelt, besitzt er kein komplexes internes Zustandsdiagramm (ECC). Er reagiert ereignisgesteuert auf eintreffende Daten an den Adaptern IN1 und IN2, berechnet die Summe und leitet das Ergebnis direkt an OUT weiter.

Anwendungsszenarien

- **Messwertverarbeitung:** Addition von zwei Sensorwerten oder die Beaufschlagung eines Messwerts mit einem Offset, der über einen Adapter eingespeist wird.
- **Sollwertberechnung:** Zusammenführung von Basis-Sollwerten und Korrekturwerten in der Prozessautomatisierung.
- **Strukturierte Programmierung:** Einsatz in komplexen Steuerungen, bei denen Daten- und Eventleitungen zur Erhöhung der Übersichtlichkeit in Adaptern gebündelt sind.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-Addition-Baustein (wie z. B. ADD aus der IEC 61131-3 Bibliothek), welcher diskrete Daten- und Ereignis-Pins (wie REQ, CNF, IN1, IN2, OUT) besitzt, kapselt AUS_ADD_2 diese Schnittstellen vollständig in Adaptern. Dies reduziert die Anzahl der sichtbaren Verbindungslinien im 4diac-Editor drastisch und fördert ein objektorientiertes Design.

- **AUS_ADD_2_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (OUT) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf OUT gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

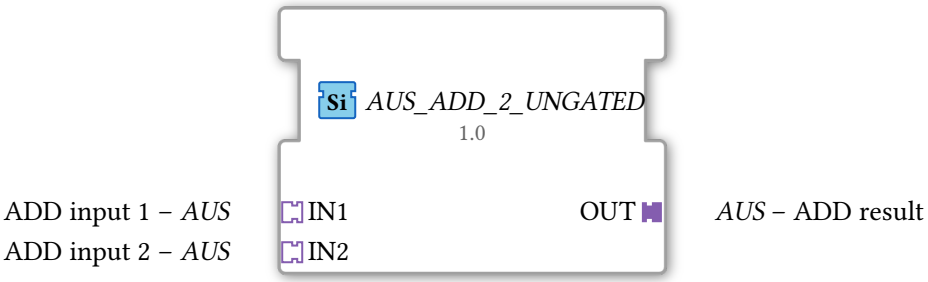
Fazit

AUS_ADD_2 ist ein effizienter, generischer Hilfsbaustein für die arithmetische Addition in IEC 61499-Anwendungen. Durch die konsequente Nutzung von unidirektionalen Adaptern eignet er sich hervorragend für moderne, modulare Softwarearchitekturen in der industriellen Automatisierungstechnik.

AUS_ADD_2_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von AUS_ADD_2. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weiter-

gegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Grafische Darstellung des Funktionsblocks nicht verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AUS_ADD_2_UNGATED` ist ein generischer Baustein (Generic FB) zur Durchführung einer arithmetischen Addition. Er dient dazu, zwei Eingangssignale zu addieren und das Ergebnis auszugeben. Die Besonderheit dieses Bausteins liegt in der Verwendung von standardisierten, unidirektionalen Adaptern (`AUS`), wodurch die Signal- und Datenverdrahtung innerhalb von Eclipse 4diac stark vereinfacht und gekapselt wird.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Eingänge. Die Ereignissteuerung wird über die Adapter abgewickelt.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Ausgänge. Die Ereignissteuerung wird über die Adapter abgewickelt.

Daten-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Eingänge.

Daten-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Ausgänge.

Adapter

Sockets (Eingangs-Adapter / Buchsen)

- **IN1** (Typ: `adapter::types::unidirectional::AUS`): Der erste Summand der Addition.
- **IN2** (Typ: `adapter::types::unidirectional::AUS`): Der zweite Summand der Addition.

Plugs (Ausgangs-Adapter / Stecker)

- **OUT** (Typ: `adapter::types::unidirectional::AUS`): Das Ergebnis der Addition (Summe aus `IN1` und `IN2`).

Funktionsweise

Der Baustein `AUS_ADD_2_UNGATED` arbeitet als arithmetischer Addierer auf Basis von Adaptern. Sobald an den Eingangs-Adaptern `IN1` oder `IN2` ein neues Ereignis samt Datenwert ankommt, führt der Baustein die Addition der beiden Werte durch:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} \tag{2.117}$$

Das Ergebnis wird zusammen mit dem entsprechenden Trigger-Ereignis über den Ausgangs-Adapter `OUT` ausgegeben. Da es sich um unidirektionale Adapter handelt, fließt der Daten- und Kontrollfluss ausschließlich von den Eingängen zum Ausgang.

Technische Besonderheiten

- **Generischer Baustein:** Durch das Attribut `eclipse4diac::core::GenericClassName` mit dem Wert `'GEN_AUS_ADD'` ist der Baustein als generischer Typ definiert. Dies ermöglicht eine flexible Anpassung an unterschiedliche Datentypen, die durch die Adapterstrukturen definiert werden.
- **Kapselung:** Durch die Verwendung von Adaptern anstelle von einzelnen Event- und Data-Pins wird das Systemdesign übersichtlicher und weniger fehleranfällig für Verdrahtungsfehler.

Zustandsübersicht

Da es sich bei diesem Baustein im Wesentlichen um eine mathematische Verknüpfung handelt, besitzt er kein komplexes internes Zustandsdiagramm (ECC). Er reagiert ereignisgesteuert auf eintreffende Daten an den Adaptern `IN1` und `IN2`, berechnet die Summe und leitet das Ergebnis direkt an `OUT` weiter.

Anwendungsszenarien

- **Messwertverarbeitung:** Addition von zwei Sensorwerten oder die Beaufschlagung eines Messwerts mit einem Offset, der über einen Adapter eingespeist wird.
- **Sollwertberechnung:** Zusammenführung von Basis-Sollwerten und Korrekturwerten in der Prozessautomatisierung.
- **Strukturierte Programmierung:** Einsatz in komplexen Steuerungen, bei denen Daten- und Eventleitungen zur Erhöhung der Übersichtlichkeit in Adaptern gebündelt sind.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-Addition-Baustein (wie z. B. `ADD` aus der IEC 61131-3 Bibliothek), welcher diskrete Daten- und Ereignis-Pins (wie `REQ`, `CNF`, `IN1`, `IN2`, `OUT`) besitzt, kapselt `AUS_ADD_2_UNGATED` diese Schnittstellen vollständig in Adaptern. Dies reduziert die Anzahl der sichtbaren Verbindungslinien im 4diac-Editor drastisch und fördert ein objektorientiertes Design.

- `AUS_ADD_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

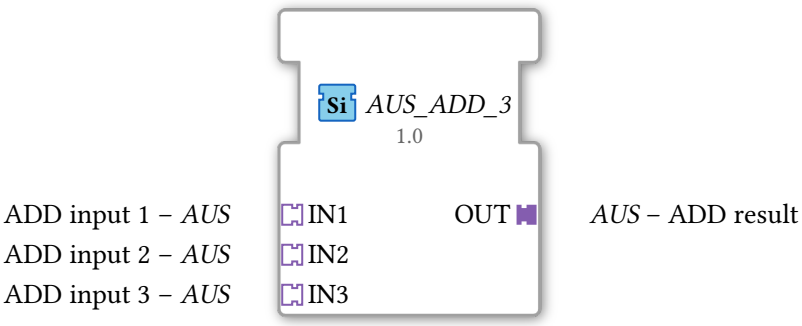
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

`AUS_ADD_2_UNGATED` ist ein effizienter, generischer Hilfsbaustein für die arithmetische Addition in IEC 61499-Anwendungen. Durch die konsequente Nutzung von unidirektionalen Adaptern eignet er sich hervorragend für moderne, modulare Softwarearchitekturen in der industriellen Automatisierungstechnik.

AUS_ADD_3



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AUS_ADD_3` ist ein generischer arithmetischer Additionsbaustein für die 4diac-IDE. Er dient dazu, die Werte von drei Eingangs-Adaptern (`IN1`, `IN2` und `IN3`) zu addieren und das Ergebnis über einen Ausgangs-Adapter (`OUT`) bereitzustellen. Der Baustein nutzt dafür unidirektionale Adapter vom Typ `AUS`.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die Steuerung und Triggerung erfolgt über die Adapter-Schnittstellen.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Ereignisweiterleitung erfolgt über die Adapter-Schnittstellen.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Der Baustein kommuniziert ausschließlich über Adapter-Schnittstellen:
Sockets (Eingangs-Adapter):

- **IN1** (Typ: `adapter::types::unidirectional::AUS`): Erster Summand für die Addition.
- **IN2** (Typ: `adapter::types::unidirectional::AUS`): Zweiter Summand für die Addition.
- **IN3** (Typ: `adapter::types::unidirectional::AUS`): Dritter Summand für die Addition.

Plugs (Ausgangs-Adapter):

- **OUT** (Typ: `adapter::types::unidirectional::AUS`): Ausgangsadapter, der das Ergebnis der Addition (`IN1 + IN2 + IN3`) führt.

Funktionsweise

Sobald an einem der Eingangs-Adapter (`IN1`, `IN2` oder `IN3`) ein neues Ereignis oder ein geänderter Datenwert signalisiert wird, führt der Baustein intern eine Addition der Werte aus. Die Berechnung folgt der mathematischen Formel:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} \tag{2.118}$$

Nach erfolgreicher Berechnung wird das Ergebnis sowie das zugehörige Aktualisierungsereignis über den Ausgangs-Adapter `OUT` an nachfolgende Bausteine weitergeleitet.

Technische Besonderheiten

- **Generischer Baustein:** Der Baustein basiert auf der generischen Klasse `GEN_AUS_ADD` (Attribut `eclipse4diac::core::GenericClassName`). Dies ermöglicht eine flexible Anpassung an unterschiedliche Datentypen innerhalb des `AUS`-Adapters.
- **Adapterbasierte Architektur:** Durch die Verwendung von Adaptern anstelle von Standard-Event/Data-Verbindungen wird der Verdrahtungsaufwand im Funktionsplan drastisch reduziert, da Events und Daten in einem einzigen Kanal (dem Adapter) gebündelt sind.
- **Unidirektionaler Datenfluss:** Die verwendeten Adapter sind unidirektional, was eine klare Richtung der Datenverarbeitung von den Eingängen zum Ausgang vorgibt.

Zustandsübersicht

Da es sich um einen kombinatorischen (bzw. zustandslosen) Funktionsbaustein handelt, besitzt `AUS_ADD_3` keine komplexen internen Zustände (keine State Machine). Die Verarbeitung erfolgt rein ereignisgesteuert:

1. **Wartezustand:** Der Baustein wartet auf ein Ereignis an einem der Eingänge (`IN1`, `IN2`, `IN3`).
2. **Berechnung:** Bei Eintreffen eines Ereignisses werden die Daten ausgelesen und addiert.

3. **Aktualisierung:** Das Ergebnis wird an `OUT` geschrieben und das entsprechende Ausgangsereignis am Adapter ausgelöst.

Anwendungsszenarien

- **Messwert-Aggregat:** Zusammenführung und Summation von drei analogen Sensorwerten (z. B. drei Durchflussmesser, Temperaturfühler oder Stromverbraucher), die über Adapter-Schnittstellen im System verteilt sind.
- **Sollwert-Generierung:** Addition von Basis-Sollwerten, Korrekturwerten und Offsets in Regelungssystemen.

Vergleich mit ähnlichen Bausteinen

- **Standard-ADD (IEC 61131-3):** Klassische `ADD`-Bausteine nutzen dedizierte Event- und Datenleitungen. `AUS_ADD_3` hingegen kapselt diese in Adaptern, was zu saubereren Software-Architekturen führt.
- **AUS_ADD_2 / AUS_ADD_4:** Im Vergleich zu Varianten mit zwei oder vier Eingängen ist dieser Baustein speziell auf genau drei Eingangskanäle optimiert, um ungenutzte Schnittstellen im Programmcode zu vermeiden.
- `AUS_ADD_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

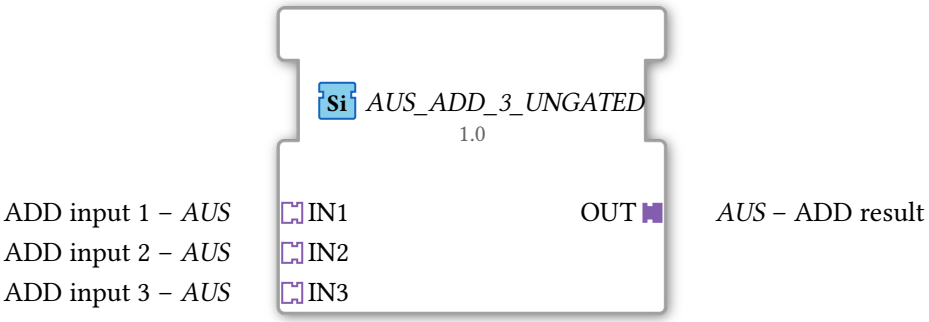
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `AUS_ADD_3` ist ein effizienter und wiederverwendbarer Funktionsbaustein zur Addition von drei Signalen. Durch die konsequente Nutzung von unidirektionalen Adaptern fördert er ein modulares, übersichtliches und wartungsfreundliches Anwendungsdesign innerhalb der IEC 61499 Entwicklungsumgebung.

AUS_ADD_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUS_ADD_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AUS_ADD_3_UNGATED` ist ein generischer arithmetischer Additionsbaustein für die 4diac-IDE. Er dient dazu, die Werte von drei Eingangs-Adaptern (`IN1`, `IN2` und `IN3`) zu addieren und das

Ergebnis über einen Ausgangs-Adapter (`OUT`) bereitzustellen. Der Baustein nutzt dafür unidirektionale Adapter vom Typ `AUS`.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die Steuerung und Triggerung erfolgt über die Adapter-Schnittstellen.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Ereignisweiterleitung erfolgt über die Adapter-Schnittstellen.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Der Baustein kommuniziert ausschließlich über Adapter-Schnittstellen:
Sockets (Eingangs-Adapter):

- **IN1** (Typ: `adapter::types::unidirectional::AUS`): Erster Summand für die Addition.
- **IN2** (Typ: `adapter::types::unidirectional::AUS`): Zweiter Summand für die Addition.
- **IN3** (Typ: `adapter::types::unidirectional::AUS`): Dritter Summand für die Addition.

Plugs (Ausgangs-Adapter):

- **OUT** (Typ: `adapter::types::unidirectional::AUS`): Ausgangsadapter, der das Ergebnis der Addition (`IN1 + IN2 + IN3`) führt.

Funktionsweise

Sobald an einem der Eingangs-Adapter (`IN1`, `IN2` oder `IN3`) ein neues Ereignis oder ein geänderter Datenwert signalisiert wird, führt der Baustein intern eine Addition der Werte aus. Die Berechnung folgt der mathematischen Formel:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} \tag{2.119}$$

Nach erfolgreicher Berechnung wird das Ergebnis sowie das zugehörige Aktualisierungsereignis über den Ausgangs-Adapter `OUT` an nachfolgende Bausteine weitergeleitet.

Technische Besonderheiten

- **Generischer Baustein:** Der Baustein basiert auf der generischen Klasse `GEN_AUS_ADD` (Attribut `eclipse4diac::core::GenericClassName`). Dies ermöglicht eine flexible Anpassung an unterschiedliche Datentypen innerhalb des `AUS`-Adapters.
- **Adapterbasierte Architektur:** Durch die Verwendung von Adaptern anstelle von Standard-Event/Data-Verbindungen wird der Verdrahtungsaufwand im Funktionsplan drastisch reduziert, da Events und Daten in einem einzigen Kanal (dem Adapter) gebündelt sind.
- **Unidirektionaler Datenfluss:** Die verwendeten Adapter sind unidirektional, was eine klare Richtung der Datenverarbeitung von den Eingängen zum Ausgang vorgibt.

Zustandsübersicht

Da es sich um einen kombinatorischen (bzw. zustandslosen) Funktionsbaustein handelt, besitzt `AUS_ADD_3_UNGATED` keine komplexen internen Zustände (keine State Machine). Die Verarbeitung erfolgt rein ereignisgesteuert:

1. **Wartezustand:** Der Baustein wartet auf ein Ereignis an einem der Eingänge (`IN1`, `IN2`, `IN3`).
2. **Berechnung:** Bei Eintreffen eines Ereignisses werden die Daten ausgelesen und addiert.
3. **Aktualisierung:** Das Ergebnis wird an `OUT` geschrieben und das entsprechende Ausgangsereignis am Adapter ausgelöst.

Anwendungsszenarien

- **Messwert-Aggregat:** Zusammenführung und Summation von drei analogen Sensorwerten (z. B. drei Durchflussmesser, Temperaturfühler oder Stromverbraucher), die über Adapter-Schnittstellen im System verteilt sind.
- **Sollwert-Generierung:** Addition von Basis-Sollwerten, Korrekturwerten und Offsets in Regelungssystemen.

Vergleich mit ähnlichen Bausteinen

- **Standard-ADD (IEC 61131-3):** Klassische `ADD`-Bausteine nutzen dedizierte Event- und Datenleitungen. `AUS_ADD_3_UNGATED` hingegen kapselt diese in Adaptern, was zu saubereren Software-Architekturen führt.
- **AUS_ADD_2 / AUS_ADD_4:** Im Vergleich zu Varianten mit zwei oder vier Eingängen ist dieser Baustein speziell auf genau drei Eingangskanäle optimiert, um ungenutzte Schnittstellen im Programmcode zu vermeiden.
- `AUS_ADD_3`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AUS_ADD_3_UNGATED` ist ein effizienter und wiederverwendbarer Funktionsbaustein zur Addition von drei Signalen. Durch die konsequente Nutzung von unidirektionalen Adaptern fördert er ein modulares, übersichtliches und wartungsfreundliches Anwendungsdesign innerhalb der IEC 61499 Entwicklungsumgebung.

AUS_ADD_4

```
       +----------------------------------------+
       |                AUS_ADD_4               |
IN1 ==# (Socket)                          (Plug) #== OUT
IN2 ==# (Socket)                                 |
IN3 ==# (Socket)                                 |
IN4 ==# (Socket)                                 |
       +----------------------------------------+
```

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AUS_ADD_4` dient der arithmetischen Addition von vier Eingangswerten. Es handelt sich hierbei um einen generischen Funktionsbaustein (Generic FB), der speziell für die Verwendung mit unidirektionalen Adaptern entwickelt wurde. Durch die Kapselung der Signale in Adaptern ermöglicht der Baustein eine saubere und übersichtliche Strukturierung von Datenflüssen innerhalb von IEC 61499 Applikationen.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten, klassischen Ereignis-Eingänge. Die Event-Steuerung wird implizit über die Adapter-Schnittstellen abgewickelt.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten, klassischen Ereignis-Ausgänge. Die Event-Weiterleitung erfolgt über die Adapter-Schnittstellen.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübertragung erfolgt ausschließlich über die zugeordneten Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Das Ergebnis wird über den Ausgangs-Adapter bereitgestellt.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AUS`): Erster Summand der Addition.
- **IN2** (Typ: `adapter::types::unidirectional::AUS`): Zweiter Summand der Addition.
- **IN3** (Typ: `adapter::types::unidirectional::AUS`): Dritter Summand der Addition.
- **IN4** (Typ: `adapter::types::unidirectional::AUS`): Vierter Summand der Addition.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AUS`): Schnittstelle zur Ausgabe des berechneten Additionsergebnisses.

Funktionsweise

Der Baustein `AUS_ADD_4` empfängt über seine vier Eingangs-Adapter (`IN1` bis `IN4`) kontinuierlich oder ereignisgesteuert numerische Werte. Sobald an den Sockets Daten anliegen oder ein Update-Event eintrifft, führt der Baustein die arithmetische Addition der vier Werte aus:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} + \mathrm{IN4} \tag{2.120}$$

Das Ergebnis dieser Berechnung wird unmittelbar an den Ausgangs-Plug `OUT` übergeben und für nachfolgende Programmteile bereitgestellt. Da es sich um unidirektionale Verbindungen handelt, fließen die Daten strikt gerichtet von den Eingängen zum Ausgang.

Technische Besonderheiten

- **Generischer Baustein:** Durch die Zuordnung zur generischen Klasse `GEN_AUS_ADD` ist der Baustein hochgradig wiederverwendbar und passt sich flexibel an die zugrundeliegenden Datentypen der Adapter an.
- **Reduzierte Routing-Komplexität:** Durch die Verwendung von Adaptern statt einzelner Event- und Datenleitungen wird die Anzahl der Verbindungslinien in der 4diac-IDE drastisch reduziert, was die Lesbarkeit von großen Applikationsdiagrammen signifikant erhöht.
- **Compiler-Zugehörigkeit:** Der Baustein ist im Package `adapter::iec61131::arithmetic` organisiert.

Zustandsübersicht

Der Baustein verhält sich wie ein kombinatorisches Glied (bzw. ein zustandsloser Funktionsblock). Er speichert keine historischen Werte zwischen den Berechnungszyklen. Jede Aktualisierung an einem der Eingangs-Adapter stößt die Berechnung direkt an und aktualisiert den Ausgang.

Anwendungsszenarien

- **Signal-Summierung:** Zusammenführung und Addition von vier analogen Sensorwerten (z. B. Ermittlung des Gesamtvolumenstroms aus vier einzelnen Durchflussmessern oder der Gesamtleistung von vier elektrischen Verbrauchern).

- **Mittelwertvorbereitung:** Aufsummierung von vier Messstellen als vorbereitender Schritt für eine anschließende Division zur Mittelwertberechnung.
- **Kombinatorische Steuerungslogik:** Aggregation von gewichteten Steuersignalen in komplexeren verteilten Systemen.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-Addierer (z. B. dem klassischen `ADD`-Baustein aus der IEC 61131-3-Bibliothek), der direkt mit elementaren Datentypen (wie `REAL` oder `INT`) arbeitet, setzt `AUS_ADD_4` vollständig auf Adapter-Kopplung. Dies spart Zeit bei der Instanziierung und Verdrahtung, setzt jedoch voraus, dass die Quell- und Zielsignale im Adaptertyp `AUS` gekapselt vorliegen.

- `AUS_ADD_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

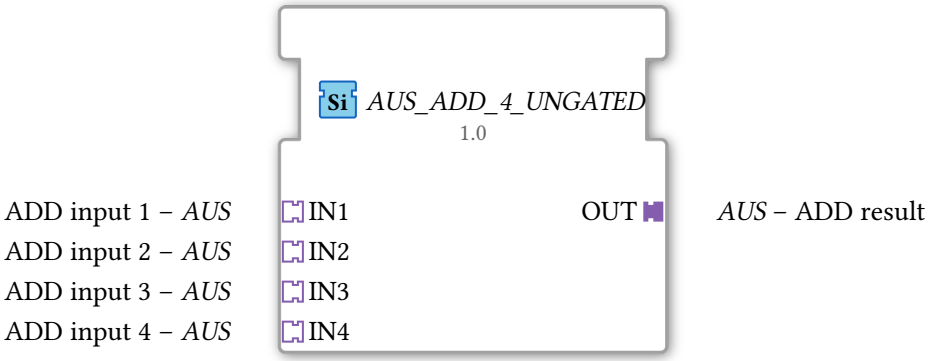
Fazit

Der `AUS_ADD_4` ist ein spezialisierter und effizienter Hilfsbaustein für die strukturierte Applikationsentwicklung in 4diac. Er eignet sich hervorragend für sauber strukturierte Steuerungsanwendungen, bei denen die Signalverarbeitung konsistent über Adapter-Pipelines abgebildet werden soll.

AUS_ADD_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUS_ADD_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

```
      +----------------------------------------+
      |               AUS_ADD_4_UNGATED        |
IN1 ==# (Socket)                        (Plug) #== OUT
IN2 ==# (Socket)                               |
IN3 ==# (Socket)                               |
IN4 ==# (Socket)                               |
      +----------------------------------------+
```



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AUS_ADD_4_UNGATED` dient der arithmetischen Addition von vier Eingangswerten. Es handelt sich hierbei um einen generischen Funktionsbaustein (Generic FB), der speziell für die Verwendung mit unidirektionalen Adaptern entwickelt wurde. Durch die Kapselung der Signale in Adaptern

ermöglicht der Baustein eine saubere und übersichtliche Strukturierung von Datenflüssen innerhalb von IEC 61499 Applikationen.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten, klassischen Ereignis-Eingänge. Die Event-Steuerung wird implizit über die Adapter-Schnittstellen abgewickelt.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten, klassischen Ereignis-Ausgänge. Die Event-Weiterleitung erfolgt über die Adapter-Schnittstellen.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübertragung erfolgt ausschließlich über die zugeordneten Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Das Ergebnis wird über den Ausgangs-Adapter bereitgestellt.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AUS`): Erster Summand der Addition.
- **IN2** (Typ: `adapter::types::unidirectional::AUS`): Zweiter Summand der Addition.
- **IN3** (Typ: `adapter::types::unidirectional::AUS`): Dritter Summand der Addition.
- **IN4** (Typ: `adapter::types::unidirectional::AUS`): Vierter Summand der Addition.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AUS`): Schnittstelle zur Ausgabe des berechneten Additionsergebnisses.

Funktionsweise

Der Baustein `AUS_ADD_4_UNGATED` empfängt über seine vier Eingangs-Adapter (`IN1` bis `IN4`) kontinuierlich oder ereignisgesteuert numerische Werte. Sobald an den Sockets Daten anliegen oder ein Update-Event eintrifft, führt der Baustein die arithmetische Addition der vier Werte aus:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} + \mathrm{IN4} \tag{2.121}$$

Das Ergebnis dieser Berechnung wird unmittelbar an den Ausgangs-Plug `OUT` übergeben und für nachfolgende Programmteile bereitgestellt. Da es sich um unidirektionale Verbindungen handelt, fließen die Daten strikt gerichtet von den Eingängen zum Ausgang.

Technische Besonderheiten

- **Generischer Baustein:** Durch die Zuordnung zur generischen Klasse `GEN_AUS_ADD` ist der Baustein hochgradig wiederverwendbar und passt sich flexibel an die zugrundeliegenden Datentypen der Adapter an.
- **Reduzierte Routing-Komplexität:** Durch die Verwendung von Adaptern statt einzelner Event- und Datenleitungen wird die Anzahl der Verbindungslinien in der 4diac-IDE drastisch reduziert, was die Lesbarkeit von großen Applikationsdiagrammen signifikant erhöht.
- **Compiler-Zugehörigkeit:** Der Baustein ist im Package `adapter::iec61131::arithmetic` organisiert.

Zustandsübersicht

Der Baustein verhält sich wie ein kombinatorisches Glied (bzw. ein zustandsloser Funktionsblock). Er speichert keine historischen Werte zwischen den Berechnungszyklen. Jede Aktualisierung an einem der Eingangs-Adapter stößt die Berechnung direkt an und aktualisiert den Ausgang.

Anwendungsszenarien

- **Signal-Summierung:** Zusammenführung und Addition von vier analogen Sensorwerten (z. B. Ermittlung des Gesamtvolumenstroms aus vier einzelnen Durchflussmessern oder der Gesamtleistung von vier elektrischen Verbrauchern).
- **Mittelwertvorbereitung:** Aufsummierung von vier Messstellen als vorbereitender Schritt für eine anschließende Division zur Mittelwertberechnung.
- **Kombinatorische Steuerungslogik:** Aggregation von gewichteten Steuersignalen in komplexeren verteilten Systemen.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-Addierer (z. B. dem klassischen `ADD`-Baustein aus der IEC 61131-3-Bibliothek), der direkt mit elementaren Datentypen (wie `REAL` oder `INT`) arbeitet, setzt `AUS_ADD_4_UNGATED` vollständig auf Adapter-Kopplung. Dies spart Zeit bei der Instanziierung und Verdrahtung, setzt jedoch voraus, dass die Quell- und Zielsignale im Adaptertyp `AUS` gekapselt vorliegen.

- `AUS_ADD_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- `AUS_ADD_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

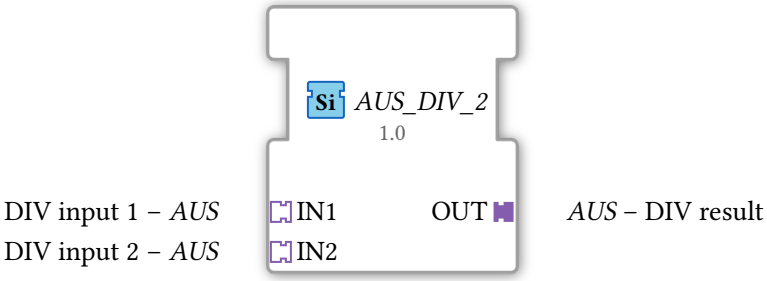
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AUS_ADD_4_UNGATED` ist ein spezialisierter und effizienter Hilfsbaustein für die strukturierte Applikationsentwicklung in 4diac. Er eignet sich hervorragend für sauber strukturierte Steuerungsanwendungen, bei denen die Signalverarbeitung konsistent über Adapter-Pipelines abgebildet werden soll.

AUS_DIV_2



(Kein Bild vorhanden)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein (FB) **AUS_DIV_2** ist ein generischer Baustein zur Durchführung der arithmetischen Division (DIV). Er ist für den Einsatz in IEC 61499-konformen Systemen (wie Eclipse 4diac) konzipiert. Der Baustein nutzt unidirektionale Adapter des Typs `AUS`, um die Eingangsdaten zu empfangen, die Division durchzuführen und das Ergebnis über einen entsprechenden Ausgangsadapter weiterzuleiten. Durch die Kapselung der Signale in Adaptern wird die Verdrahtung innerhalb der Entwicklungsumgebung erheblich vereinfacht.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine direkten Ereignis-Eingänge auf der Bausteinoberfläche vorhanden. Die ereignisbasierte Steuerung und Triggerung der Berechnung erfolgt intern über die angeschlossenen Eingangs-Adapter.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge auf der Bausteinoberfläche vorhanden. Die Weiterleitung von Berechnungs- und Aktualisierungsereignissen erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Der Baustein besitzt keine direkten elementaren Daten-Eingänge. Alle Eingangsdaten werden über die Adapter-Schnittstellen übergeben.

Daten-Ausgänge

Der Baustein besitzt keine direkten elementaren Daten-Ausgänge. Das Berechnungsergebnis wird über den Ausgangs-Adapter bereitgestellt.

Adapter

Typ	Name	Adaptertyp	Beschreibung
Socket (Eingang)	`IN1`	`adapter::types::unidirectional::AUS`	Erster Eingangswert für die Division (Dividend).
Socket (Eingang)	`IN2`	`adapter::types::unidirectional::AUS`	Zweiter Eingangswert für die Division (Divisor).
Plug (Ausgang)	`OUT`	`adapter::types::unidirectional::AUS`	Ausgang für das Ergebnis der Division (Quotient).

Funktionsweise

Der Baustein `AUS_DIV_2` führt eine klassische Division nach folgendem mathematischen Prinzip aus:

$$\mathrm{OUT} = \frac{\mathrm{IN1}}{\mathrm{IN2}} \tag{2.122}$$

Da es sich um einen generischen Funktionsbaustein handelt (gekennzeichnet durch das Attribut `GEN_AUS_DIV`), ist der Baustein nicht auf einen festen Datentyp (wie z. B. `INT` oder `REAL`) festgelegt. Der tatsächliche Datentyp wird bei der Instanziierung in der 4diac-IDE basierend auf den verbundenen Adaptern aufgelöst.

Sobald an den Eingangs-Adaptern `IN1` oder `IN2` ein neues Datenereignis signalisiert wird, führt der Baustein die Division durch und aktualisiert den Wert am Ausgangs-Adapter `OUT`, gefolgt von einem entsprechenden Freigabeereignis über den Ausgangs-Plug.

Technische Besonderheiten

- **Generischer Typ (`GEN_AUS_DIV`):** Ermöglicht eine flexible Wiederverwendung für unterschiedliche numerische Datentypen, ohne dass separate Bausteine für `INT`, `REAL` oder `LREAL` angelegt werden müssen.
- **Unidirektionale Adapter:** Die Verwendung des Typs `adapter::types::unidirectional::AUS` sorgt für eine saubere Daten- und Event-Kapselung. Dies reduziert die visuelle Komplexität im Systementwurf (weniger „Spaghetti-Code“ durch lose Event- und Datenlinien).
- **Division durch Null:** Bei der Implementierung auf der Zielplattform (Runtime) ist darauf zu achten, wie der Baustein auf einen Divisor von `0` reagiert (z. B. Ausgabe von `NaN`/`INF` bei Gleitkommazahlen oder Systemfehler bei Ganzzahlen).

Zustandsübersicht

Der Baustein besitzt kein komplexes internes Zustandsdiagramm (ECC). Seine Ausführung ist rein transaktional:

1. **Wartezustand:** Der Baustein wartet auf ein Aktualisierungsereignis an den Sockets `IN1` oder `IN2`.
2. **Berechnung:** Bei Ereigniseingang werden die Werte ausgelesen und dividiert.
3. **Ausgabe:** Das Ergebnis wird an den Plug `OUT` geschrieben und das Ausgangsereignis getriggert. Der Baustein kehrt sofort in den Wartezustand zurück.

Anwendungsszenarien

- **Messwertskalierung:** Division von Sensorwerten durch konstante Faktoren, die über Adapterstrukturen im System verteilt werden.
- **Mittelwertberechnung:** Einsatz in mathematischen Berechnungsnetzwerken innerhalb von verteilten Steuerungssystemen.
- **Verhältnissteuerung:** Berechnung von Verhältnissen (z. B. Luft-Brennstoff-Verhältnis in Brennersteuerungen), bei denen die Eingangssignale bereits als standardisierte `AUS`-Adapter vorliegen.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-Ganzzahl-Divisionsbaustein (wie dem IEC 61131-3 `DIV` Baustein) bietet `AUS_DIV_2` folgende Vorteile:

- **Adapterbasierte Kopplung:** Standard-FBs benötigen separate Pins für Daten und Events (REQ/CNF). `AUS_DIV_2` bündelt diese logisch in den `AUS`-Adaptern.
- **Datentyp-Flexibilität:** Während klassische Bausteine oft fest für z. B. `REAL` (e.g., `R_DIV`) definiert sind, passt sich dieser generische Baustein dem verwendeten Adaptertyp an.
- **`AUS_DIV_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

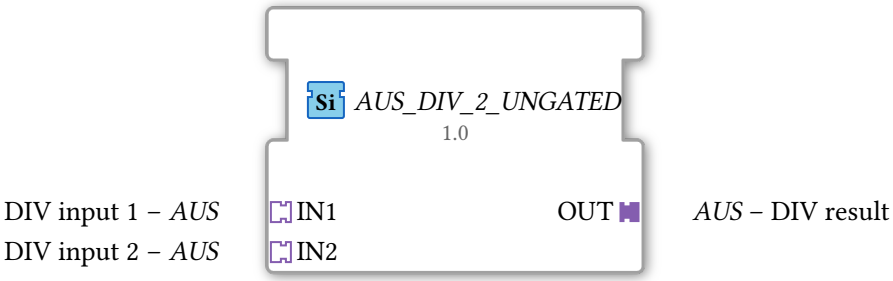
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der Funktionsbaustein `AUS_DIV_2` ist ein hocheffizienter, wiederverwendbarer Hilfsbaustein für arithmetische Berechnungen in modernen, adapterbasierten IEC 61499-Architekturen. Er trägt maßgeblich zur Übersichtlichkeit von Applikationsdiagrammen bei, indem er die Signal- und Eventflüsse für mathematische Divisionen in standardisierten Adaptern konsolidiert.

AUS_DIV_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUS_DIV_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild vorhanden)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein (FB) **AUS_DIV_2_UNGATED** ist ein generischer Baustein zur Durchführung der arithmetischen Division (DIV). Er ist für den Einsatz in IEC 61499-konformen Systemen (wie Eclipse 4diac) konzipiert. Der Baustein nutzt unidirektionale Adapter des Typs `AUS`, um die Eingangsdaten zu

empfangen, die Division durchzuführen und das Ergebnis über einen entsprechenden Ausgangsadapter weiterzuleiten. Durch die Kapselung der Signale in Adaptern wird die Verdrahtung innerhalb der Entwicklungsumgebung erheblich vereinfacht.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine direkten Ereignis-Eingänge auf der Bausteinoberfläche vorhanden. Die ereignisbasierte Steuerung und Triggerung der Berechnung erfolgt intern über die angeschlossenen Eingangs-Adapter.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge auf der Bausteinoberfläche vorhanden. Die Weiterleitung von Berechnungs- und Aktualisierungsereignissen erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Der Baustein besitzt keine direkten elementaren Daten-Eingänge. Alle Eingangsdaten werden über die Adapter-Schnittstellen übergeben.

Daten-Ausgänge

Der Baustein besitzt keine direkten elementaren Daten-Ausgänge. Das Berechnungsergebnis wird über den Ausgangs-Adapter bereitgestellt.

Adapter

Typ	Name	Adaptertyp	Beschreibung
Socket (Eingang)	`IN1`	`adapter::types::unidirectional::AUS`	Erster Eingangswert für die Division (Dividend).
Socket (Eingang)	`IN2`	`adapter::types::unidirectional::AUS`	Zweiter Eingangswert für die Division (Divisor).
Plug (Ausgang)	`OUT`	`adapter::types::unidirectional::AUS`	Ausgang für das Ergebnis der Division (Quotient).

Funktionsweise

Der Baustein `AUS_DIV_2_UNGATED` führt eine klassische Division nach folgendem mathematischen Prinzip aus:

$$\mathrm{OUT} = \frac{\mathrm{IN1}}{\mathrm{IN2}} \tag{2.123}$$

Da es sich um einen generischen Funktionsbaustein handelt (gekennzeichnet durch das Attribut `GEN_AUS_DIV`), ist der Baustein nicht auf einen festen Datentyp (wie z. B. `INT` oder `REAL`) festgelegt. Der tatsächliche Datentyp wird bei der Instanziierung in der 4diac-IDE basierend auf den verbundenen Adaptern aufgelöst.

Sobald an den Eingangs-Adaptern `IN1` oder `IN2` ein neues Datenereignis signalisiert wird, führt der Baustein die Division durch und aktualisiert den Wert am Ausgangs-Adapter `OUT`, gefolgt von einem entsprechenden Freigabeereignis über den Ausgangs-Plug.

Technische Besonderheiten

- **Generischer Typ (`GEN_AUS_DIV`):** Ermöglicht eine flexible Wiederverwendung für unterschiedliche numerische Datentypen, ohne dass separate Bausteine für `INT`, `REAL` oder `LREAL` angelegt werden müssen.
- **Unidirektionale Adapter:** Die Verwendung des Typs `adapter::types::unidirectional::AUS` sorgt für eine saubere Daten- und Event-Kapselung. Dies reduziert die visuelle Komplexität im Systementwurf (weniger „Spaghetti-Code“ durch lose Event- und Datenlinien).
- **Division durch Null:** Bei der Implementierung auf der Zielplattform (Runtime) ist darauf zu achten, wie der Baustein auf einen Divisor von `0` reagiert (z. B. Ausgabe von `NaN`/`INF` bei Gleitkommazahlen oder Systemfehler bei Ganzzahlen).

Zustandsübersicht

Der Baustein besitzt kein komplexes internes Zustandsdiagramm (ECC). Seine Ausführung ist rein transaktional:

1. **Wartezustand:** Der Baustein wartet auf ein Aktualisierungsereignis an den Sockets `IN1` oder `IN2`.
2. **Berechnung:** Bei Ereigniseingang werden die Werte ausgelesen und dividiert.
3. **Ausgabe:** Das Ergebnis wird an den Plug `OUT` geschrieben und das Ausgangsereignis getriggert. Der Baustein kehrt sofort in den Wartezustand zurück.

Anwendungsszenarien

- **Messwertskalierung:** Division von Sensorwerten durch konstante Faktoren, die über Adapterstrukturen im System verteilt werden.
- **Mittelwertberechnung:** Einsatz in mathematischen Berechnungsnetzwerken innerhalb von verteilten Steuerungssystemen.
- **Verhältnissteuerung:** Berechnung von Verhältnissen (z. B. Luft-Brennstoff-Verhältnis in Brennersteuerungen), bei denen die Eingangssignale bereits als standardisierte `AUS`-Adapter vorliegen.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-Ganzzahl-Divisionsbaustein (wie dem IEC 61131-3 `DIV` Baustein) bietet `AUS_DIV_2_UNGATED` folgende Vorteile:

- **Adapterbasierte Kopplung:** Standard-FBs benötigen separate Pins für Daten und Events (REQ/CNF). `AUS_DIV_2_UNGATED` bündelt diese logisch in den `AUS`-Adaptern.
- **Datentyp-Flexibilität:** Während klassische Bausteine oft fest für z. B. `REAL` (e.g., `R_DIV`) definiert sind, passt sich dieser generische Baustein dem verwendeten Adaptertyp an.
- **`AUS_DIV_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- **`AUS_DIV_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

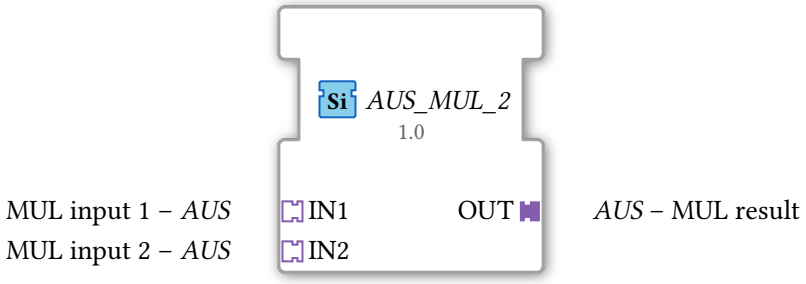
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der Funktionsbaustein `AUS_DIV_2_UNGATED` ist ein hocheffizienter, wiederverwendbarer Hilfsbaustein für arithmetische Berechnungen in modernen, adapterbasierten IEC 61499-Architekturen. Er trägt maßgeblich zur Übersichtlichkeit von Applikationsdiagrammen bei, indem er die Signal- und Eventflüsse für mathematische Divisionen in standardisierten Adaptern konsolidiert.

AUS_MUL_2



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AUS_MUL_2` ist ein generischer Arithmetikbaustein für die 4diac-IDE, der zur Multiplikation zweier Werte dient. Im Gegensatz zu klassischen mathematischen Funktionsbausteinen nutzt

dieser Baustein ein adapterbasiertes Konzept zur Datenübertragung. Dadurch wird eine strukturierte, modulare und übersichtliche Signalverdrahtung in komplexen IEC 61499 Anwendungen ermöglicht.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Eingänge. Die Ereignissteuerung wird über die angeschlossenen Adapter abgewickelt.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Ausgänge. Die Ereignisweiterleitung erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Eingänge.

Daten-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Ausgänge.

Adapter

Die gesamte Kommunikation (Daten und zugehörige Trigger-Ereignisse) wird über unidirektionale Adapter des Typs `AUS` realisiert.

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AUS`): Der erste Multiplikand (Eingang 1).
- **IN2** (Typ: `adapter::types::unidirectional::AUS`): Der zweite Multiplikand (Eingang 2).

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AUS`): Das Ergebnis der Multiplikation ($IN1 \times IN2$).

Funktionsweise

Sobald an den Eingangs-Adaptern `IN1` oder `IN2` ein neues Ereignis signalisiert wird, das auf neue Daten hinweist, führt der Funktionsbaustein intern die Multiplikation aus:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \tag{2.124}$$

Das Ergebnis der Berechnung sowie das entsprechende Bestätigungsereignis werden anschließend über den Ausgangs-Plug `OUT` an nachfolgende Bausteine weitergegeben. Da es sich um einen generischen Baustein (`GEN_AUS_MUL`) handelt, passt sich die interne Verarbeitung flexibel an den im Adapter definierten Datentyp an.

Technische Besonderheiten

- **Generischer Charakter:** Der Baustein ist als generischer Typ (`GenericClassName = 'GEN_AUS_MUL'`) deklariert. Dies ermöglicht eine flexible Handhabung verschiedener numerischer Datentypen (z. B. `INT`, `REAL`, `LREAL`), die durch die Adapterstruktur vorgegeben werden.
- **Adapter-Fokus:** Die Reduktion von klassischen Pins hin zu Adaptern minimiert den Verdrahtungsaufwand im Funktionsplan erheblich und sorgt für ein sauberes, objektorientiertes Design.

Zustandsübersicht

Der Baustein arbeitet ereignisgesteuert basierend auf den Zustandsänderungen der Adapter:

1. **IDLE (Bereitschaft):** Der Baustein wartet auf eingehende Werte/Ereignisse an den Sockets `IN1` und `IN2`.
2. **BERECHNUNG:** Bei Empfang eines Triggers wird die Multiplikation der in den Adaptern enthaltenen Datenwerte ausgeführt.
3. **AUSGABE:** Das Produkt wird in den Adapter `OUT` geschrieben und ein Sende-Ereignis ausgelöst. Der Baustein kehrt in den Zustand *IDLE* zurück.

Anwendungsszenarien

- **Messwertskalierung:** Multiplikation eines Rohwertes (z. B. aus einem Sensor-Adapter) mit einem Kalibrierungsfaktor.
- **Leistungsberechnung:** Multiplikation von Strom- und Spannungswerten, die über standardisierte Adapter-Schnittstellen eingelesen werden.
- **Modulare Signalverarbeitung:** Einsatz in komplexen Regelungsschleifen, bei denen Signalketten sauber durch Adapter gekapselt sind, um die Übersicht im Steuerungsdiagramm zu wahren.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zum Standard-IEC-61131-3-Multiplikationsbaustein (`MUL`), der mit einzelnen Pins für `REQ`, `CNF` sowie direkten Daten-Ein- und Ausgängen arbeitet, bündelt `AUS_MUL_2` all diese Signale in Adaptern. Dies verhindert „Kabelsalat“ im Funktionsplan, erfordert jedoch, dass die angebundenen Signale bereits im `AUS`-Adapterformat vorliegen.

- `AUS_MUL_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

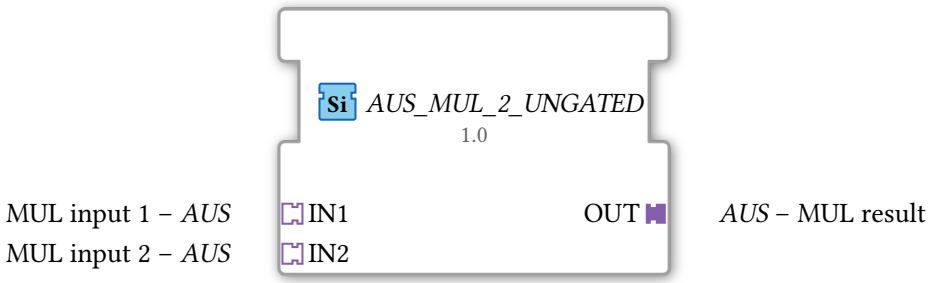
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `AUS_MUL_2` ist ein moderner, robuster und wiederverwendbarer Funktionsbaustein zur Multiplikation in der 4diac-IDE. Durch die konsequente Nutzung von unidirektionalen Adaptern eignet er sich hervorragend für serviceorientierte Architekturen und strukturierte Applikationsdesigns im industriellen Umfeld.

AUS_MUL_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUS_MUL_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AUS_MUL_2_UNGATED` ist ein generischer Arithmetikbaustein für die 4diac-IDE, der zur Multiplikation zweier Werte dient. Im Gegensatz zu klassischen mathematischen Funktionsbausteinen nutzt dieser Baustein ein adapterbasiertes Konzept zur Datenübertragung. Dadurch wird eine strukturierte, modulare und übersichtliche Signalverdrahtung in komplexen IEC 61499 Anwendungen ermöglicht.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Eingänge. Die Ereignissteuerung wird über die angeschlossenen Adapter abgewickelt.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Ausgänge. Die Ereignisweiterleitung erfolgt über den Ausgangs-Adapter.

Daten-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Eingänge.

Daten-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Ausgänge.

Adapter

Die gesamte Kommunikation (Daten und zugehörige Trigger-Ereignisse) wird über unidirektionale Adapter des Typs `AUS` realisiert.

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AUS`): Der erste Multiplikand (Eingang 1).
- **IN2** (Typ: `adapter::types::unidirectional::AUS`): Der zweite Multiplikand (Eingang 2).

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AUS`): Das Ergebnis der Multiplikation ($IN1 \times IN2$).

Funktionsweise

Sobald an den Eingangs-Adaptern `IN1` oder `IN2` ein neues Ereignis signalisiert wird, das auf neue Daten hinweist, führt der Funktionsbaustein intern die Multiplikation aus:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \tag{2.125}$$

Das Ergebnis der Berechnung sowie das entsprechende Bestätigungsereignis werden anschließend über den Ausgangs-Plug `OUT` an nachfolgende Bausteine weitergegeben. Da es sich um einen generischen Baustein (`GEN_AUS_MUL`) handelt, passt sich die interne Verarbeitung flexibel an den im Adapter definierten Datentyp an.

Technische Besonderheiten

- **Generischer Charakter:** Der Baustein ist als generischer Typ (`GenericClassName = 'GEN_AUS_MUL'`) deklariert. Dies ermöglicht eine flexible Handhabung verschiedener numerischer Datentypen (z. B. `INT`, `REAL`, `LREAL`), die durch die Adapterstruktur vorgegeben werden.
- **Adapter-Fokus:** Die Reduktion von klassischen Pins hin zu Adaptern minimiert den Verdrahtungsaufwand im Funktionsplan erheblich und sorgt für ein sauberes, objektorientiertes Design.

Zustandsübersicht

Der Baustein arbeitet ereignisgesteuert basierend auf den Zustandsänderungen der Adapter:

1. **IDLE (Bereitschaft):** Der Baustein wartet auf eingehende Werte/Ereignisse an den Sockets `IN1` und `IN2`.
2. **BERECHNUNG:** Bei Empfang eines Triggers wird die Multiplikation der in den Adaptern enthaltenen Datenwerte ausgeführt.
3. **AUSGABE:** Das Produkt wird in den Adapter `OUT` geschrieben und ein Sende-Ereignis ausgelöst. Der Baustein kehrt in den Zustand *IDLE* zurück.

Anwendungsszenarien

- **Messwertskalierung:** Multiplikation eines Rohwertes (z. B. aus einem Sensor-Adapter) mit einem Kalibrierungsfaktor.
- **Leistungsberechnung:** Multiplikation von Strom- und Spannungswerten, die über standardisierte Adapter-Schnittstellen eingelesen werden.
- **Modulare Signalverarbeitung:** Einsatz in komplexen Regelungsschleifen, bei denen Signalketten sauber durch Adapter gekapselt sind, um die Übersicht im Steuerungsdiagramm zu wahren.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zum Standard-IEC-61131-3-Multiplikationsbaustein (`MUL`), der mit einzelnen Pins für `REQ`, `CNF` sowie direkten Daten-Ein- und Ausgängen arbeitet, bündelt `AUS_MUL_2_UNGATED` all diese Signale in Adaptern. Dies verhindert „Kabelsalat“ im Funktionsplan, erfordert jedoch, dass die angebundenen Signale bereits im `AUS`-Adapterformat vorliegen.

- `AUS_MUL_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

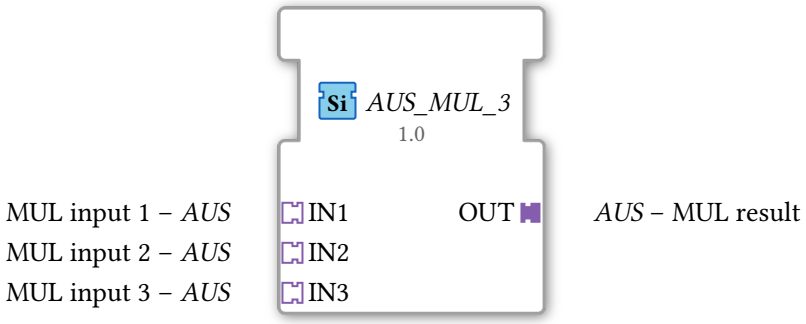
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AUS_MUL_2_UNGATED` ist ein moderner, robuster und wiederverwendbarer Funktionsbaustein zur Multiplikation in der 4diac-IDE. Durch die konsequente Nutzung von unidirektionalen Adaptern eignet er sich hervorragend für serviceorientierte Architekturen und strukturierte Applikationsdesigns im industriellen Umfeld.

AUS_MUL_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein (FB) `AUS_MUL_3` dient der arithmetischen Multiplikation von drei Eingangswerten. Es handelt sich hierbei um einen generischen Funktionsbaustein (`Generic FB`), der auf der Verwendung von unidirektionalen Adaptern des Typs `AUS` basiert. Der Baustein berechnet das Produkt aus drei Eingangsgrößen und stellt das Ergebnis an einem Ausgangs-Adapter zur Verfügung.

Schnittstellenstruktur

Da dieser Funktionsbaustein vollständig auf Adaptern basiert, verfügt er über keine direkten, klassischen Ereignis- oder Datenschnittstellen auf der Bausteinebene. Die gesamte Kommunikation und Synchronisation erfolgt über die deklarierten Adapter.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. (Die Ereignissteuerung wird über die Adapter-Sockets abgewickelt).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. (Die Ereignissteuerung wird über den Adapter-Plug abgewickelt).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AUS`): Der erste Multiplikand für die arithmetische Operation.
- **IN2** (Typ: `adapter::types::unidirectional::AUS`): Der zweite Multiplikand für die arithmetische Operation.
- **IN3** (Typ: `adapter::types::unidirectional::AUS`): Der dritte Multiplikand für die arithmetische Operation.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AUS`): Das berechnete Ergebnis der Multiplikation ($OUT = IN1 \times IN2 \times IN3$).

Funktionsweise

Sobald an den Eingangs-Adaptern (`IN1`, `IN2`, `IN3`) neue Werte anliegen und ein entsprechendes Trigger-Ereignis über die Adapter übertragen wird, führt der Baustein die Multiplikation aus.

Die mathematische Formel lautet:

$$\text{Ergebnis} = \text{Wert von IN1} \times \text{Wert von IN2} \times \text{Wert von IN3} \tag{2.126}$$

Das berechnete Ergebnis wird zusammen mit einem entsprechenden Bestätigungsereignis über den Ausgangs-Plug `OUT` an die nachfolgenden Programmteile weitergeleitet.

Technische Besonderheiten

- **Generischer Baustein:** Durch das Attribut `GenericClassName` mit dem Wert `'GEN_AUS_MUL'` ist der Baustein flexibel einsetzbar. Je nach konkreter Implementierung des Adaptertyps `AUS` kann sich der Baustein auf verschiedene Datentypen (z. B. `INT`, `REAL`, `LREAL`) anpassen.
- **Kapselung durch Adapter:** Die Verwendung von unidirektionalen Adaptern (`unidirectional::AUS`) reduziert das Routing von einzelnen Event- und Datenleitungen in der 4diac-IDE drastisch. Dies sorgt für ein saubereres und übersichtlicheres Applikationsdesign.

Zustandsübersicht

Der Baustein verhält sich rein zustandslos (bzw. reagiert ereignisgesteuert auf die eingehenden Adapter-Signale):

1. **Wartezustand:** Der Baustein wartet auf ein Aktualisierungsereignis an den Sockets `IN1`, `IN2` oder `IN3`.
2. **Berechnung:** Bei Eintreffen eines Ereignisses werden die Datenwerte der drei Eingangs-Adapter gelesen und miteinander multipliziert.
3. **Ausgabe:** Das Ergebnis wird an den Plug `OUT` geschrieben und das dazugehörige Sende-Ereignis des Plugs ausgelöst.

Anwendungsszenarien

- **Skalierung und Gewichtung:** Multiplikation eines Sensorwertes mit zwei aufeinanderfolgenden Skalierungsfaktoren (z. B. physikalische Umrechnung und Kalibrierungsfaktor).
- **Volumenberechnung:** Berechnung eines dreidimensionalen Rauminhalts oder Durchflusses, bei dem drei Parameter (z. B. Länge × Breite × Höhe oder drei verschiedene Raten) direkt miteinander multipliziert werden müssen.
- **Strukturierte Signalverarbeitung:** Einsatz in komplexeren Regelkreisen, die eine saubere Signalarchitektur mittels Adaptern verlangen.

Vergleich mit ähnlichen Bausteinen

- **Standard-MUL-Baustein (IEC 61131-3):** Der klassische `MUL`-Baustein nutzt direkte Daten- und Ereignis-Pins. `AUS_MUL_3` hingegen bündelt diese Signale in Adaptern, was die Verdrahtung in großen Projekten vereinfacht.
- **AUS_MUL_2 (Zwei-Eingangs-Variante):** Während eine hypothetische Variante mit zwei Eingängen nur zwei Faktoren multiplizieren kann, spart `AUS_MUL_3` bei der Multiplikation von drei Werten einen zusätzlichen Kaskadierungs-Baustein ein.
- `AUS_MUL_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

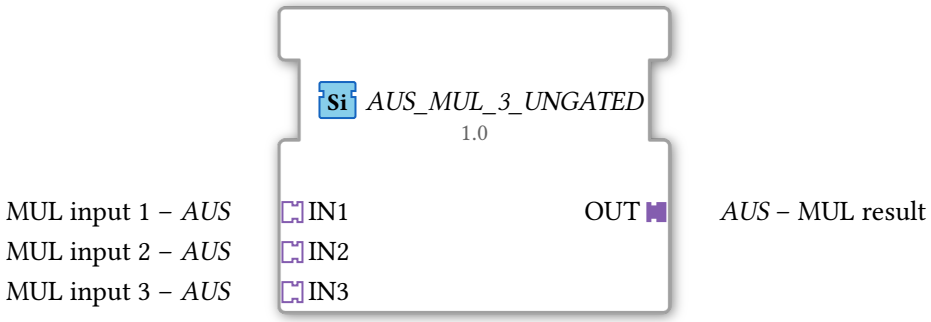
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `AUS_MUL_3` ist ein hocheffizienter, generischer Funktionsbaustein zur Multiplikation von drei Werten. Durch die konsequente Nutzung von unidirektionalen Adaptern eignet er sich hervorragend für moderne, adapterbasierte Softwarearchitekturen in der 4diac-Umgebung, da er die visuelle Komplexität von Steuerungsdiagrammen minimiert.

AUS_MUL_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUS_MUL_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein (FB) `AUS_MUL_3_UNGATED` dient der arithmetischen Multiplikation von drei Eingangswerten. Es handelt sich hierbei um einen generischen Funktionsbaustein (`Generic FB`), der auf der Verwendung von unidirektionalen Adaptern des Typs `AUS` basiert. Der Baustein berechnet das Produkt aus drei Eingangsgrößen und stellt das Ergebnis an einem Ausgangs-Adapter zur Verfügung.

Schnittstellenstruktur

Da dieser Funktionsbaustein vollständig auf Adaptern basiert, verfügt er über keine direkten, klassischen Ereignis- oder Datenschnittstellen auf der Bausteinebene. Die gesamte Kommunikation und Synchronisation erfolgt über die deklarierten Adapter.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. (Die Ereignissteuerung wird über die Adapter-Sockets abgewickelt).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. (Die Ereignissteuerung wird über den Adapter-Plug abgewickelt).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AUS`): Der erste Multiplikand für die arithmetische Operation.
- **IN2** (Typ: `adapter::types::unidirectional::AUS`): Der zweite Multiplikand für die arithmetische Operation.
- **IN3** (Typ: `adapter::types::unidirectional::AUS`): Der dritte Multiplikand für die arithmetische Operation.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AUS`): Das berechnete Ergebnis der Multiplikation ($OUT = IN1 \times IN2 \times IN3$).

Funktionsweise

Sobald an den Eingangs-Adaptern (`IN1`, `IN2`, `IN3`) neue Werte anliegen und ein entsprechendes Trigger-Ereignis über die Adapter übertragen wird, führt der Baustein die Multiplikation aus.

Die mathematische Formel lautet:

$$\text{Ergebnis} = \text{Wert von IN1} \times \text{Wert von IN2} \times \text{Wert von IN3} \tag{2.127}$$

Das berechnete Ergebnis wird zusammen mit einem entsprechenden Bestätigungsereignis über den Ausgangs-Plug `OUT` an die nachfolgenden Programmteile weitergeleitet.

Technische Besonderheiten

- **Generischer Baustein:** Durch das Attribut `GenericClassName` mit dem Wert `'GEN_AUS_MUL'` ist der Baustein flexibel einsetzbar. Je nach konkreter Implementierung des Adaptertyps `AUS` kann sich der Baustein auf verschiedene Datentypen (z. B. `INT`, `REAL`, `LREAL`) anpassen.
- **Kapselung durch Adapter:** Die Verwendung von unidirektionalen Adaptern (`unidirectional::AUS`) reduziert das Routing von einzelnen Event- und Datenleitungen in der 4diac-IDE drastisch. Dies sorgt für ein saubereres und übersichtlicheres Applikationsdesign.

Zustandsübersicht

Der Baustein verhält sich rein zustandslos (bzw. reagiert ereignisgesteuert auf die eingehenden Adapter-Signale):

1. **Wartezustand:** Der Baustein wartet auf ein Aktualisierungsereignis an den Sockets `IN1`, `IN2` oder `IN3`.
2. **Berechnung:** Bei Eintreffen eines Ereignisses werden die Datenwerte der drei Eingangs-Adapter gelesen und miteinander multipliziert.
3. **Ausgabe:** Das Ergebnis wird an den Plug `OUT` geschrieben und das dazugehörige Sende-Ereignis des Plugs ausgelöst.

Anwendungsszenarien

- **Skalierung und Gewichtung:** Multiplikation eines Sensorwertes mit zwei aufeinanderfolgenden Skalierungsfaktoren (z. B. physikalische Umrechnung und Kalibrierungsfaktor).

- **Volumenberechnung:** Berechnung eines dreidimensionalen Rauminhalts oder Durchflusses, bei dem drei Parameter (z. B. Länge × Breite × Höhe oder drei verschiedene Raten) direkt miteinander multipliziert werden müssen.
- **Strukturierte Signalverarbeitung:** Einsatz in komplexeren Regelkreisen, die eine saubere Signalarchitektur mittels Adaptern verlangen.

Vergleich mit ähnlichen Bausteinen

- **Standard-MUL-Baustein (IEC 61131-3):** Der klassische `MUL`-Baustein nutzt direkte Daten- und Ereignis-Pins. `AUS_MUL_3_UNGATED` hingegen bündelt diese Signale in Adaptern, was die Verdrahtung in großen Projekten vereinfacht.
- **AUS_MUL_2 (Zwei-Eingangs-Variante):** Während eine hypothetische Variante mit zwei Eingängen nur zwei Faktoren multiplizieren kann, spart `AUS_MUL_3_UNGATED` bei der Multiplikation von drei Werten einen zusätzlichen Kaskadierungs-Baustein ein.
- **`AUS_MUL_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.
- **`AUS_MUL_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

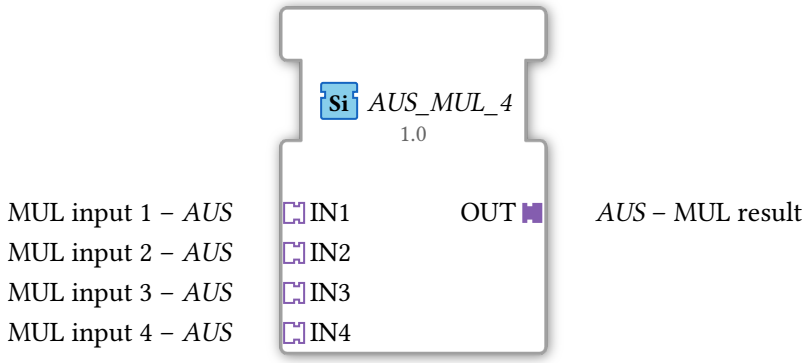
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AUS_MUL_3_UNGATED` ist ein hocheffizienter, generischer Funktionsbaustein zur Multiplikation von drei Werten. Durch die konsequente Nutzung von unidirektionalen Adaptern eignet er sich hervorragend für moderne, adapterbasierte Softwarearchitekturen in der 4diac-Umgebung, da er die visuelle Komplexität von Steuerungsdiagrammen minimiert.

AUS_MUL_4



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AUS_MUL_4` ist ein generischer Baustein (Generic FB) zur Durchführung einer arithmetischen Multiplikation (`MUL`) mit bis zu vier Eingangswerten. Er basiert auf der IEC 61499-Architektur und nutzt unidirektionale Adapter, um Daten und die dazugehörigen Steuerungsereignisse sauber gekapselt zu übertragen.

Durch seinen generischen Charakter (`GEN_AUS_MUL`) ist der Baustein flexibel einsetzbar und kann auf verschiedene numerische Datentypen angewendet werden.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Eingänge. Die Ereignissteuerung wird vollständig über die angeschlossenen Adapter abgewickelt.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Ausgänge. Die Ereignissteuerung wird vollständig über den Ausgangs-Adapter abgewickelt.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübergabe erfolgt über die Eingangs-Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Die Datenübergabe erfolgt über den Ausgangs-Adapter.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AUS`): Erster Eingangswert (Multiplikand 1) für die Multiplikation.
- **IN2** (Typ: `adapter::types::unidirectional::AUS`): Zweiter Eingangswert (Multiplikand 2) für die Multiplikation.
- **IN3** (Typ: `adapter::types::unidirectional::AUS`): Dritter Eingangswert (Multiplikand 3) für die Multiplikation.
- **IN4** (Typ: `adapter::types::unidirectional::AUS`): Vierter Eingangswert (Multiplikand 4) für die Multiplikation.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AUS`): Das berechnete Produkt (Ergebnis der Multiplikation).

Funktionsweise

Sobald an den Eingangs-Adaptern (`IN1` bis `IN4`) neue Datenwerte und die entsprechenden Triggereignisse anliegen, führt der Baustein die Multiplikation durch.

Die mathematische Berechnung folgt der Formel:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \times \mathrm{IN4} \tag{2.128}$$

Das berechnete Ergebnis wird zusammen mit einem entsprechenden Ausgabe-Ereignis über den Ausgangs-Adapter `OUT` bereitgestellt. Da es sich um einen generischen Funktionsbaustein handelt, passt sich der tatsächliche Datentyp (z. B. `INT`, `REAL`, `LREAL`) der Instanziierung in der 4diac-IDE an.

Technische Besonderheiten

- **Generischer Baustein:** Durch die Definition als `GEN_AUS_MUL` ist der Baustein nicht auf einen festen Datentyp fixiert, was die Wiederverwendbarkeit in verschiedenen Projekten drastisch erhöht.
- **Adapterbasierte Kommunikation:** Die Verwendung von unidirektionalen `AUS`-Adaptern sorgt für ein übersichtliches Anwendungsdiagramm, da Daten- und Eventleitungen in einer einzigen Verbindung gebündelt werden. Dies reduziert Verdrahtungsfehler in der Entwicklungsumgebung.

Zustandsübersicht

Der Funktionsbaustein verhält sich wie ein klassischer, zustandsloser mathematischer Operator (kombinatorische Logik). Er speichert keine internen Zustände zwischen den Berechnungszyklen. Jede Aktualisierung der Eingangswerte über die Adapter führt zu einer Neuberechnung und Aktualisierung des Ausgangs.

Anwendungsszenarien

- **Skalierung und Kalibrierung:** Berechnung von komplexeren physikalischen Werten, bei denen mehrere Faktoren (z. B. Sensorwert × Kalibrierungsfaktor × Temperaturkompensation × Einheitenumrechnung) multipliziert werden müssen.
- **Leistungsberechnung:** Kombination von mehreren Messgrößen in der Prozesstechnik.
- **Mehrstufige Verstärkungsregelungen:** Kaskadierung von Verstärkungsfaktoren in Steuerungsalgorithmen.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-Multiplikationsbausteinen (wie dem klassischen `MUL` der IEC 61131-3), die mit einzelnen Pins für Events und Daten arbeiten, bietet der `AUS_MUL_4` folgende Vorteile:

- **Weniger Verbindungen:** Vier separate Eingänge und ein Ausgang würden im klassischen Modell mindestens 10 Verbindungen (5 Events, 5 Datenleitungen) erfordern. Durch die Adapter-Technologie wird dies auf 5 Adapter-Verbindungen reduziert.
- **Kompaktheit:** Das Kaskadieren von mehreren 2-fach Multiplizierern entfällt, da direkt vier Operanden in einem einzigen Baustein verarbeitet werden können.

- `AUS_MUL_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

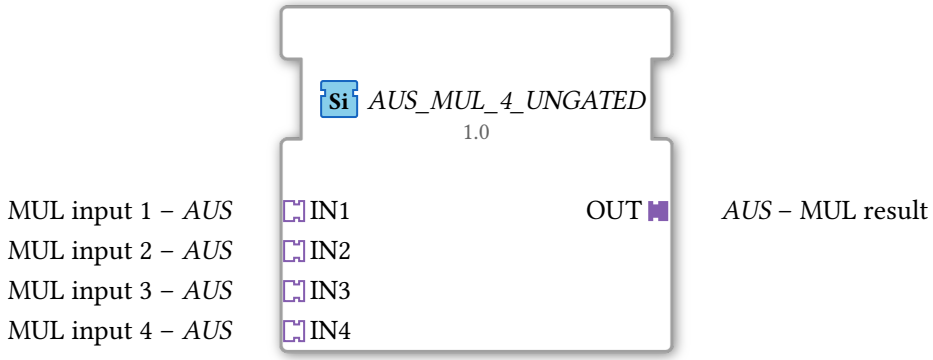
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `AUS_MUL_4` ist ein hocheffizienter und moderner Funktionsbaustein für die 4diac-IDE. Er kombiniert die mathematische Grundfunktion der Multiplikation mit den modernen Strukturierungsvorteilen von IEC 61499-Adaptern, was zu saubereren, wartungsfreundlicheren und übersichtlicheren Steuerungsanwendungen führt.

AUS_MUL_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUS_MUL_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AUS_MUL_4_UNGATED` ist ein generischer Baustein (Generic FB) zur Durchführung einer arithmetischen Multiplikation (`MUL`) mit bis zu vier Eingangswerten. Er basiert auf der IEC 61499-Architektur und nutzt unidirektionale Adapter, um Daten und die dazugehörigen Steuerungsereignisse sauber gekapselt zu übertragen.

Durch seinen generischen Charakter (`GEN_AUS_MUL`) ist der Baustein flexibel einsetzbar und kann auf verschiedene numerische Datentypen angewendet werden.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Eingänge. Die Ereignissteuerung wird vollständig über die angeschlossenen Adapter abgewickelt.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Ausgänge. Die Ereignissteuerung wird vollständig über den Ausgangs-Adapter abgewickelt.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Datenübergabe erfolgt über die Eingangs-Adapter.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Die Datenübergabe erfolgt über den Ausgangs-Adapter.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AUS`): Erster Eingangswert (Multiplikand 1) für die Multiplikation.
- **IN2** (Typ: `adapter::types::unidirectional::AUS`): Zweiter Eingangswert (Multiplikand 2) für die Multiplikation.
- **IN3** (Typ: `adapter::types::unidirectional::AUS`): Dritter Eingangswert (Multiplikand 3) für die Multiplikation.
- **IN4** (Typ: `adapter::types::unidirectional::AUS`): Vierter Eingangswert (Multiplikand 4) für die Multiplikation.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AUS`): Das berechnete Produkt (Ergebnis der Multiplikation).

Funktionsweise

Sobald an den Eingangs-Adaptern (`IN1` bis `IN4`) neue Datenwerte und die entsprechenden Triggereignisse anliegen, führt der Baustein die Multiplikation durch.

Die mathematische Berechnung folgt der Formel:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \times \mathrm{IN4} \tag{2.129}$$

Das berechnete Ergebnis wird zusammen mit einem entsprechenden Ausgabe-Ereignis über den Ausgangs-Adapter `OUT` bereitgestellt. Da es sich um einen generischen Funktionsbaustein handelt, passt sich der tatsächliche Datentyp (z. B. `INT`, `REAL`, `LREAL`) der Instanziierung in der 4diac-IDE an.

Technische Besonderheiten

- **Generischer Baustein:** Durch die Definition als `GEN_AUS_MUL` ist der Baustein nicht auf einen festen Datentyp fixiert, was die Wiederverwendbarkeit in verschiedenen Projekten drastisch erhöht.
- **Adapterbasierte Kommunikation:** Die Verwendung von unidirektionalen `AUS`-Adaptern sorgt für ein übersichtliches Anwendungsdiagramm, da Daten- und Eventleitungen in einer einzigen Verbindung gebündelt werden. Dies reduziert Verdrahtungsfehler in der Entwicklungsumgebung.

Zustandsübersicht

Der Funktionsbaustein verhält sich wie ein klassischer, zustandsloser mathematischer Operator (kombinatorische Logik). Er speichert keine internen Zustände zwischen den Berechnungszyklen. Jede Aktualisierung der Eingangswerte über die Adapter führt zu einer Neuberechnung und Aktualisierung des Ausgangs.

Anwendungsszenarien

- **Skalierung und Kalibrierung:** Berechnung von komplexeren physikalischen Werten, bei denen mehrere Faktoren (z. B. Sensorwert × Kalibrierungsfaktor × Temperaturkompensation × Einheitenumrechnung) multipliziert werden müssen.
- **Leistungsberechnung:** Kombination von mehreren Messgrößen in der Prozesstechnik.
- **Mehrstufige Verstärkungsregelungen:** Kaskadierung von Verstärkungsfaktoren in Steuerungsalgorithmen.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-Multiplikationsbausteinen (wie dem klassischen `MUL` der IEC 61131-3), die mit einzelnen Pins für Events und Daten arbeiten, bietet der `AUS_MUL_4_UNGATED` folgende Vorteile:

- **Weniger Verbindungen:** Vier separate Eingänge und ein Ausgang würden im klassischen Modell mindestens 10 Verbindungen (5 Events, 5 Datenleitungen) erfordern. Durch die Adapter-Technologie wird dies auf 5 Adapter-Verbindungen reduziert.
- **Kompaktheit:** Das Kaskadieren von mehreren 2-fach Multiplizierern entfällt, da direkt vier Operanden in einem einzigen Baustein verarbeitet werden können.

- `AUS_MUL_4`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

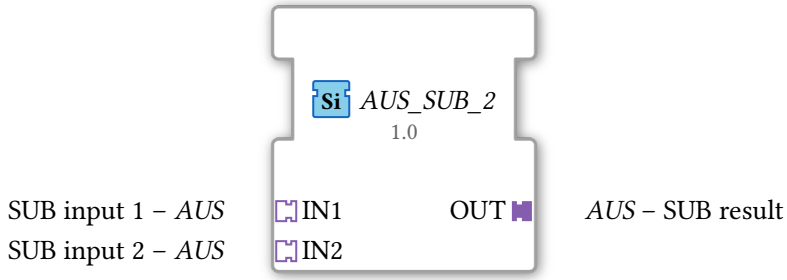
Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AUS_MUL_4_UNGATED` ist ein hocheffizienter und moderner Funktionsbaustein für die 4diac-IDE. Er kombiniert die mathematische Grundfunktion der Multiplikation mit den modernen Strukturierungsvorteilen von IEC 61499-Adaptern, was zu saubereren, wartungsfreundlicheren und übersichtlicheren Steuerungsanwendungen führt.

AUS_SUB_2



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein (FB) `AUS_SUB_2` ist ein generischer Baustein zur Durchführung von arithmetischen Subtraktionsoperationen innerhalb von 4diac-basierten Steuerungssystemen. Der Baustein nutzt unidirektionale Adapter vom Typ `AUS` zur Kapselung und Übertragung von Eingangs- und Ausgangssignalen. Durch sein generisches Design (Kompilierungsklasse `GEN_AUS_SUB`) kann er flexibel in verschiedenen Steuerungsszenarien eingesetzt werden, um die Differenz zweier Werte zu bilden.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten, dedizierten Ereignis-Eingänge. Die Ereignissteuerung wird vollständig über die verwendeten Adapter abgewickelt.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten, dedizierten Ereignis-Ausgänge. Die Ereignissteuerung wird vollständig über die verwendeten Adapter abgewickelt.

Daten-Eingänge

Es sind keine direkten, elementaren Daten-Eingänge vorhanden. Die Datenübergabe erfolgt über die Eingangs-Adapter.

Daten-Ausgänge

Es sind keine direkten, elementaren Daten-Ausgänge vorhanden. Das Ergebnis wird über den Ausgangs-Adapter bereitgestellt.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AUS`): Der erste Eingang (Minuend) für die Subtraktionsberechnung.
- **IN2** (Typ: `adapter::types::unidirectional::AUS`): Der zweite Eingang (Subtrahend) für die Subtraktionsberechnung.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AUS`): Der Ausgang (Differenz), der das berechnete Ergebnis der Subtraktion liefert.

Funktionsweise

Die primäre Aufgabe von `AUS_SUB_2` ist die arithmetische Subtraktion:

$$\text{OUT} = \text{IN1} - \text{IN2} \tag{2.130}$$

Da der Funktionsbaustein auf Adaptern basiert, empfängt er die Werte und die dazugehörigen Auslöseereignisse über die Sockets `IN1` und `IN2`. Sobald relevante Daten über die Eingangs-Adapter eintreffen, wird die mathematische Operation ausgeführt. Das Ergebnis der Berechnung sowie das entsprechende Ausgabereignis werden anschließend über den Plug `OUT` an die nachfolgenden Bausteine weitergeleitet.

Da es sich um einen generischen Baustein (`GEN_AUS_SUB`) handelt, passt sich die mathematische Verarbeitung dynamisch an die im `AUS`-Adapter definierten Datentypen an.

Technische Besonderheiten

- **Generischer Typ (`GEN_AUS_SUB`):** Der Baustein ist nicht auf einen festen Datentyp (wie z. B. nur `INT` oder `REAL`) beschränkt, sondern unterstützt die durch die Adapterstruktur vorgegebenen Datentypen.
- **Unidirektionale Adapter:** Die Verwendung des Typs `unidirectional::AUS` sorgt für eine klare, einseitige Daten- und Signalflussrichtung. Dies minimiert die Komplexität bei der Signalverfolgung im System.
- **Kapselung:** Durch den Verzicht auf klassische Event- und Daten-Pins bleibt das visuelle Layout im Application Editor der 4diac-ide extrem kompakt und übersichtlich.

Zustandsübersicht

Da dieser Baustein als rein funktionaler bzw. zustandsloser Berechnungsblock konzipiert ist, besitzt er keine komplexe interne Execution Control Chart (ECC). Seine Ausführung ist rein daten- und ereignisgesteuert basierend auf den Interaktionen der angeschlossenen Adapter.

Anwendungsszenarien

- **Prozesswertkorrektur:** Subtraktion von Nullpunkt-Offsets oder Kalibrierungswerten von einem gemessenen Sensorwert.
- **Soll-Ist-Wert-Vergleich:** Berechnung der Regelabweichung ($e = w - x$) in Regelungskreisen, bei denen die Signale bereits als strukturierte Adapter-Kanäle vorliegen.
- **Füllstandsberechnung:** Ermittlung von Differenzmengen in Behältern oder Systemen durch Subtraktion des Abflusses vom Zufluss.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem klassischen IEC 61131-3 `SUB`-Baustein, der direkt mit elementaren Datentypen arbeitet, bietet der `AUS_SUB_2` folgende Vorteile:

- **Weniger Verdrahtungsaufwand:** Ereignis- und Datenleitungen müssen nicht separat gezogen werden, da sie im `AUS`-Adapter gebündelt sind.
- **Höhere Modularität:** Er eignet sich ideal für serviceorientierte Architekturen in der IEC 61499, in denen Subsysteme standardmäßig über Adapter kommunizieren.

- `AUS_SUB_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Änderungserkennung

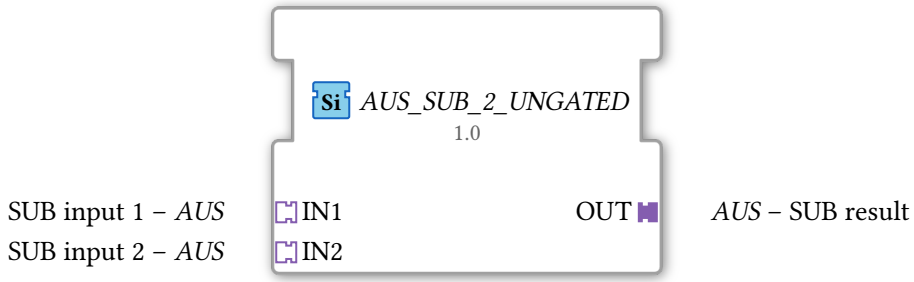
Das Ergebnis wird nur auf den Ausgangs-Plug (`OUT`) geschrieben und dessen Adapter-Event nur gesendet, wenn sich der neu berechnete Wert vom aktuell auf `OUT` gehaltenen Wert unterscheidet. Bleibt das Ergebnis unverändert, wird kein Adapter-Event gesendet – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Fazit

Der `AUS_SUB_2` ist ein spezialisierter, aber dennoch flexibel einsetzbarer Subtraktionsbaustein. Durch die konsequente Nutzung von unidirektionalen Adaptern fördert er ein sauberes, strukturiertes und übersichtliches Design von Steuerungsprogrammen in der 4diac-ide.

AUS_SUB_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUS_SUB_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein (FB) `AUS_SUB_2_UNGATED` ist ein generischer Baustein zur Durchführung von arithmetischen Subtraktionsoperationen innerhalb von 4diac-basierten Steuerungssystemen. Der Baustein nutzt unidirektionale Adapter vom Typ `AUS` zur Kapselung und Übertragung von Eingangs- und Ausgangssignalen. Durch sein generisches Design (Kompilierungsklasse `GEN_AUS_SUB`) kann er flexibel in verschiedenen Steuerungsszenarien eingesetzt werden, um die Differenz zweier Werte zu bilden.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten, dedizierten Ereignis-Eingänge. Die Ereignissteuerung wird vollständig über die verwendeten Adapter abgewickelt.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten, dedizierten Ereignis-Ausgänge. Die Ereignissteuerung wird vollständig über die verwendeten Adapter abgewickelt.

Daten-Eingänge

Es sind keine direkten, elementaren Daten-Eingänge vorhanden. Die Datenübergabe erfolgt über die Eingangs-Adapter.

Daten-Ausgänge

Es sind keine direkten, elementaren Daten-Ausgänge vorhanden. Das Ergebnis wird über den Ausgangs-Adapter bereitgestellt.

Adapter

Sockets (Eingangs-Adapter)

- **IN1** (Typ: `adapter::types::unidirectional::AUS`): Der erste Eingang (Minuend) für die Subtraktionsberechnung.
- **IN2** (Typ: `adapter::types::unidirectional::AUS`): Der zweite Eingang (Subtrahend) für die Subtraktionsberechnung.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AUS`): Der Ausgang (Differenz), der das berechnete Ergebnis der Subtraktion liefert.

Funktionsweise

Die primäre Aufgabe von `AUS_SUB_2_UNGATED` ist die arithmetische Subtraktion:

$$\mathrm{OUT} = \mathrm{IN1} - \mathrm{IN2} \tag{2.131}$$

Da der Funktionsbaustein auf Adaptern basiert, empfängt er die Werte und die dazugehörigen Auslöseereignisse über die Sockets `IN1` und `IN2`. Sobald relevante Daten über die Eingangs-Adapter eintreffen, wird die mathematische Operation ausgeführt. Das Ergebnis der Berechnung sowie das entsprechende Ausgabereignis werden anschließend über den Plug `OUT` an die nachfolgenden Bausteine weitergeleitet.

Da es sich um einen generischen Baustein (`GEN_AUS_SUB`) handelt, passt sich die mathematische Verarbeitung dynamisch an die im `AUS`-Adapter definierten Datentypen an.

Technische Besonderheiten

- **Generischer Typ (`GEN_AUS_SUB`):** Der Baustein ist nicht auf einen festen Datentyp (wie z. B. nur `INT` oder `REAL`) beschränkt, sondern unterstützt die durch die Adapterstruktur vorgegebenen Datentypen.
- **Unidirektionale Adapter:** Die Verwendung des Typs `unidirectional::AUS` sorgt für eine klare, einseitige Daten- und Signalflussrichtung. Dies minimiert die Komplexität bei der Signalverfolgung im System.
- **Kapselung:** Durch den Verzicht auf klassische Event- und Daten-Pins bleibt das visuelle Layout im Application Editor der 4diac-ide extrem kompakt und übersichtlich.

Zustandsübersicht

Da dieser Baustein als rein funktionaler bzw. zustandsloser Berechnungsblock konzipiert ist, besitzt er keine komplexe interne Execution Control Chart (ECC). Seine Ausführung ist rein daten- und ereignisgesteuert basierend auf den Interaktionen der angeschlossenen Adapter.

Anwendungsszenarien

- **Prozesswertkorrektur:** Subtraktion von Nullpunkt-Offsets oder Kalibrierungswerten von einem gemessenen Sensorwert.
- **Soll-Ist-Wert-Vergleich:** Berechnung der Regelabweichung ($e = w - x$) in Regelungskreisen, bei denen die Signale bereits als strukturierte Adapter-Kanäle vorliegen.
- **Füllstandsberechnung:** Ermittlung von Differenzmengen in Behältern oder Systemen durch Subtraktion des Abflusses vom Zufluss.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem klassischen IEC 61131-3 `SUB`-Baustein, der direkt mit elementaren Datentypen arbeitet, bietet der `AUS_SUB_2_UNGATED` folgende Vorteile:

- **Weniger Verdrahtungsaufwand:** Ereignis- und Datenleitungen müssen nicht separat gezogen werden, da sie im `AUS`-Adapter gebündelt sind.
- **Höhere Modularität:** Er eignet sich ideal für serviceorientierte Architekturen in der IEC 61499, in denen Subsysteme standardmäßig über Adapter kommunizieren.

- `AUS_SUB_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Änderungserkennung

Dieser Baustein führt **keine** Änderungserkennung durch. Jedes neu berechnete Ergebnis wird bedingungslos auf den Ausgang geschrieben und das zugehörige Adapter-Event gesendet, unabhängig davon, ob sich der Wert gegenüber dem vorherigen Durchlauf geändert hat.

Fazit

Der `AUS_SUB_2_UNGATED` ist ein spezialisierter, aber dennoch flexibel einsetzbarer Subtraktionsbaustein. Durch die konsequente Nutzung von unidirektionalen Adaptern fördert er ein sauberes, strukturiertes und übersichtliches Design von Steuerungsprogrammen in der 4diac-ide.

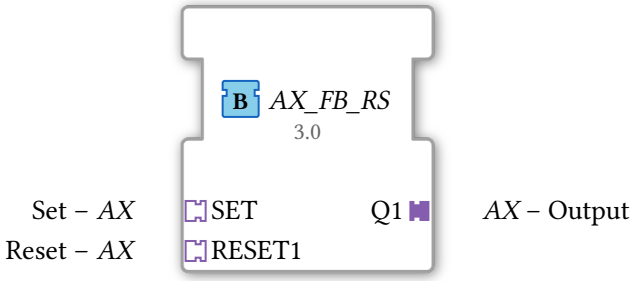
bistableElements

AX_FB_RS

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **AX_FB_RS** realisiert ein bistabiles Kippglied mit vorrangigem Rücksetzen (Reset-Dominanz), basierend auf der IEC 61131-3 Norm. Im Gegensatz zum Standard-`RS`-Baustein nutzt dieser FB Adapter-Schnittstellen (`AX`), um Ereignisse und Daten gebündelt zu übertragen. Dies ermöglicht eine kompaktere Darstellung in Steuerungsdiagrammen, da Daten- und Ereignisverbindungen in einer einzigen Linie zusammengefasst werden.



Schnittstellenstruktur

Da dieser Baustein ausschließlich auf Adaptern basiert, sind die klassischen Ereignis- und Daten-Ein-/Ausgänge leer. Die Kommunikation erfolgt vollständig über die definierten Plugs und Sockets.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Der Baustein verwendet den Adapter-Typ `adapter::types::unidirectional::AX`. Dieser Adaptertyp bündelt typischerweise ein boolesches Signal (`D1`) und ein zugehöriges Ereignis (`E1`).

Name	Typ	Richtung	Beschreibung
Q1	AX	Plug (Ausgang)	Der Ausgang des Flip-Flops (Status).
SET	AX	Socket (Eingang)	Der Setz-Eingang. Wenn aktiv, wird der Ausgang `Q1` auf TRUE gesetzt (sofern `RESET1` nicht aktiv ist).
RESET1	AX	Socket (Eingang)	Der Rücksetz-Eingang. Wenn aktiv, wird der Ausgang `Q1` auf FALSE gesetzt. Dieser Eingang hat Priorität vor `SET`.

Funktionsweise

Der **AX_FB_RS** implementiert die Logik eines RS-Flip-Flops mit Reset-Dominanz. Der interne Zustand und der Ausgang `Q1` werden basierend auf den Eingangswerten der Adapter `SET` und `RESET1` berechnet.
Die logische Funktion lautet: `Q1 := (NOT RESET1) AND (SET OR Q1)`
Das Verhalten lässt sich wie folgt zusammenfassen:

1. **Rücksetzen:** Wenn `RESET1.D1` TRUE ist, wird `Q1.D1` sofort FALSE, unabhängig vom Zustand von `SET.D1`.
2. **Setzen:** Wenn `SET.D1` TRUE ist und `RESET1.D1` FALSE ist, wird `Q1.D1` TRUE.
3. **Speichern:** Wenn sowohl `SET.D1` als auch `RESET1.D1` FALSE sind, behält `Q1.D1` seinen vorherigen Zustand bei.

Sobald ein Ereignis (`E1`) an einem der Eingangs-Adapter (`SET` oder `RESET1`) empfangen wird, führt der Baustein den Algorithmus aus und feuert anschließend ein Ereignis am Ausgangs-Adapter `Q1.E1`.

Technische Besonderheiten

- **Adapter-Technologie:** Durch die Verwendung von `AX`-Adaptern reduziert der Baustein die visuelle Komplexität in 4diac-Anwendungen, da für das Paar aus Ereignis und booleschem Wert nur eine Verbindungslinie gezogen werden muss.

- **Ereignisgesteuert:** Der Algorithmus wird bei Eintreffen eines Events über den `SET`- oder `RESET1`-Adapter ausgelöst.
- **Reset-Dominanz:** Dies ist ein sicherheitsrelevantes Merkmal. Im Falle eines Konflikts (beide Eingänge TRUE) schaltet der Ausgang sicher ab (FALSE).

Zustandsübersicht

Der Baustein basiert auf einem einfachen ECC (Execution Control Chart) mit einem einzigen Hauptzustand.

- **Zustandsname:** `REQ`
- **Aktion:** Führt den Algorithmus `REQ` aus und sendet ein Ereignis über `Q1.E1`.
- **Transitionen:**
  - Der Zustand `REQ` hat Selbstübergänge (Self-Loops), die durch das Eintreffen von Ereignissen an den Adaptern `SET.E1` oder `RESET1.E1` ausgelöst werden.

Dies bedeutet, dass der Baustein immer bereit ist, Eingangssignale zu verarbeiten und sofort darauf zu reagieren.

Anwendungsszenarien

- **Motorsteuerungen (Start/Stop):** Typische Anwendung für Start- (Set) und Stop-Taster (Reset), wobei der Stop-Taster aus Sicherheitsgründen Vorrang hat (Not-Halt-Logik).
- **Alarm-Speicher:** Ein Alarm wird gesetzt und bleibt bestehen, bis er explizit zurückgesetzt (quittiert) wird.
- **Zustandsmerker:** Speichern von binären Zuständen innerhalb einer adapterbasierten Steuerungslogik.
- **Modulare Systeme:** Einsatz in Systemen, die konsequent auf Adapter-Schnittstellen setzen, um „Spaghetti-Code“ in der grafischen Darstellung zu vermeiden.

⚖️ Vergleich mit ähnlichen Bausteinen

Merkmal	AX_FB_RS	AX_FB_SR	RS (Standard)
Dominanz	**Reset-Dominant** (Ausgang FALSE bei S=1, R=1)	Set-Dominant (Ausgang TRUE bei S=1, R=1)	Reset-Dominant
Schnittstelle	Adapter (`AX`)	Adapter (`AX`)	Boolesche Ereignis-/Datenports
Verdrahtung	Minimalistisch (gebündelt)	Minimalistisch (gebündelt)	Klassisch (getrennte Leitungen)

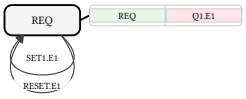
Fazit

Der **AX_FB_RS** ist eine moderne, adapterbasierte Implementierung des klassischen RS-Flip-Flops. Er eignet sich hervorragend für komplexe Steuerungsanwendungen in 4diac, bei denen Übersichtlichkeit und Kapselung von Signalen im Vordergrund stehen. Durch die integrierte Reset-Dominanz ist er besonders für Anwendungen geeignet, bei denen das sichere Abschalten Vorrang vor dem Einschalten haben muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AX_FB_SR

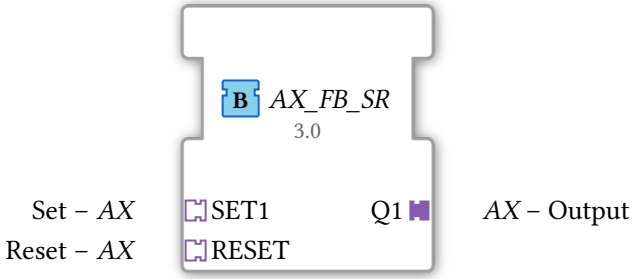


• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **AX_FB_SR** realisiert ein bistabiles Kippglied (Latch) mit **Setz-Dominanz** (SR-Flip-Flop). Im Gegensatz zu herkömmlichen IEC 61131-3 Bausteinen verwendet dieser FB jedoch keine diskreten Ereignis- und Datenanschlüsse, sondern **Adapter** für die Signalübertragung. Dies ermöglicht eine kompaktere Darstellung und vereinfachte Verdrahtung in komplexen Steuerungsanwendungen, die auf dem Adapter-Konzept basieren.

Der Baustein speichert einen binären Zustand. Wenn sowohl das Setz- (SET) als auch das Rücksetzsignal (RESET) gleichzeitig aktiv sind, hat das Setzsignal Vorrang (Setz-Dominanz).



Schnittstellenstruktur

Der Baustein kommuniziert ausschließlich über Adapter-Schnittstellen. Klassische diskrete Ein- und Ausgänge sind nicht vorhanden, da diese innerhalb der Adapter gekapselt sind.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden (diese sind in den Sockets `SET1` und `RESET` enthalten).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden (diese sind im Plug `Q1` enthalten).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden (diese sind in den Sockets `SET1` und `RESET` enthalten).

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden (diese sind im Plug `Q1` enthalten).

Adapter

Name	Typ	Richtung	Beschreibung
Q1	`adapter::types::unidirectional::AX`	Plug (Ausgang)	Der Ausgangsadapter, der den aktuellen Zustand (Q) und das zugehörige Ereignis bereitstellt.
SET1	`adapter::types::unidirectional::AX`	Socket (Eingang)	Der Setz-Eingangsadapter. Ein `TRUE`-Signal hier setzt den Ausgang auf `TRUE`.
RESET	`adapter::types::unidirectional::AX`	Socket (Eingang)	Der Rücksetz-Eingangsadapter. Ein `TRUE`-Signal hier setzt den Ausgang auf `FALSE` (sofern SET1 nicht aktiv ist).

Hinweis: Der Adapter-Typ `AX` impliziert typischerweise eine Kombination aus einem booleschen Datenpunkt (`D1`) und einem Ereignis (`E1`).

Funktionsweise

Der interne Algorithmus implementiert die Logik eines SR-Flip-Flops mit Vorrang auf Setzen. Die Logik wird ausgeführt, sobald ein Ereignis an einem der Adapter-Eingänge (`SET1.E1` oder `RESET.E1`) empfangen wird.

Die logische Gleichung lautet (in ST-Syntax):

```
Q1.D1 := SET1.D1 OR ((NOT RESET.D1) AND Q1.D1);
```

Dies bedeutet:

1. Ist der Dateneingang von **SET1** `TRUE`, wird der Ausgang **Q1** auf `TRUE` gesetzt (unabhängig von RESET).
2. Ist **SET1** `FALSE` und **RESET** `TRUE`, wird der Ausgang **Q1** auf `FALSE` gesetzt.
3. Sind beide Eingänge `FALSE`, behält **Q1** seinen vorherigen Zustand bei (Speicherfunktion).

Nach der Berechnung des neuen Zustands wird das Ereignis am Ausgangsadapter (`Q1.E1`) ausgelöst, um nachfolgende Bausteine über die Aktualisierung zu informieren.

Technische Besonderheiten

- **Adapter-basiert:** Der Hauptvorteil dieses Bausteins ist die Reduktion von Verbindungslinien im Funktionsplan, da Daten und Ereignisse über eine einzige Adapter-Verbindung gebündelt werden.
- **Ereignisgesteuert:** Der Baustein reagiert sofort auf eingehende Ereignisse an den `SET1` oder `RESET` Adaptern und leitet das Ergebnis sofort weiter.
- **Zustandsmaschine (ECC):** Die Execution Control Chart (ECC) ist minimal gehalten. Sie besteht aus einem einzigen Zustand `REQ`, der auf Eingangsereignisse wartet und den Algorithmus `REQ` ausführt.

Zustandsübersicht

Die folgende Wahrheitstabelle beschreibt das Verhalten des Bausteins (wobei `Q_n` der alte Zustand und `Q_n+1` der neue Zustand ist):

SET1.D1	RESET.D1	Q_n (Alter Zustand)	Q_n+1 (Neuer Zustand)	Beschreibung
0	0	0	0	Zustand halten
0	0	1	1	Zustand halten
0	1	X	0	Rücksetzen
1	0	X	1	Setzen
1	1	X	1	**Setzen (Dominant)**

Anwendungsszenarien

- **Adapter-basierte Steuerungssysteme:** Ideal für Projekte, die konsequent die Adapter-Technologie von 4diac nutzen, um den „Spaghetti-Code“ von parallelen Event- und Datenleitungen zu vermeiden.
- **Start/Stop-Logik:** Realisierung von Maschinensteuerungen, bei denen das „Ein“-Signal (Setzen) Priorität hat oder eine dauerhafte Aktivierung sichergestellt werden muss, solange der Taster gedrückt ist.
- **Alarm-Handling:** Speichern von Alarmzuständen, die manuell quittiert (zurückgesetzt) werden müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **SR (Standard):** Der `AX_FB_SR` entspricht logisch dem Standard-`SR` Baustein der IEC 61131-3, unterscheidet sich jedoch durch die Nutzung von `AX`-Adaptern anstelle von diskreten `BOOL`- und `EVENT`-Pins.
- **AX_FB_RS:** Dies ist das Gegenstück mit **Rücksetz-Dominanz**. Beim `AX_FB_RS` würde bei gleichzeitigem Anliegen von SET und RESET der Ausgang auf `FALSE` gehen.
- **E_SR:** Ein ereignisgesteuertes Standard-SR-Flip-Flop ohne Adapter.

Fazit

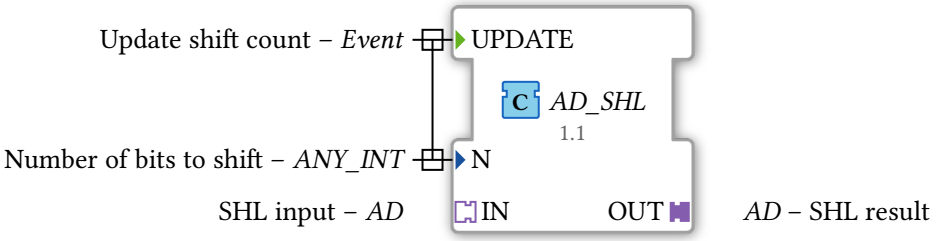
Der **AX_FB_SR** ist ein spezialisierter Baustein für die strukturierte, adapterbasierte Programmierung in 4diac. Er bietet die klassische Funktionalität eines setz-dominanten Speicherglieds, optimiert jedoch die grafische Darstellung und Handhabung durch die Kapselung von Signalen in Adaptern. Er ist unverzichtbar in Anwendungen, die auf modularen und übersichtlichen Schnittstellendefinitionen basieren.

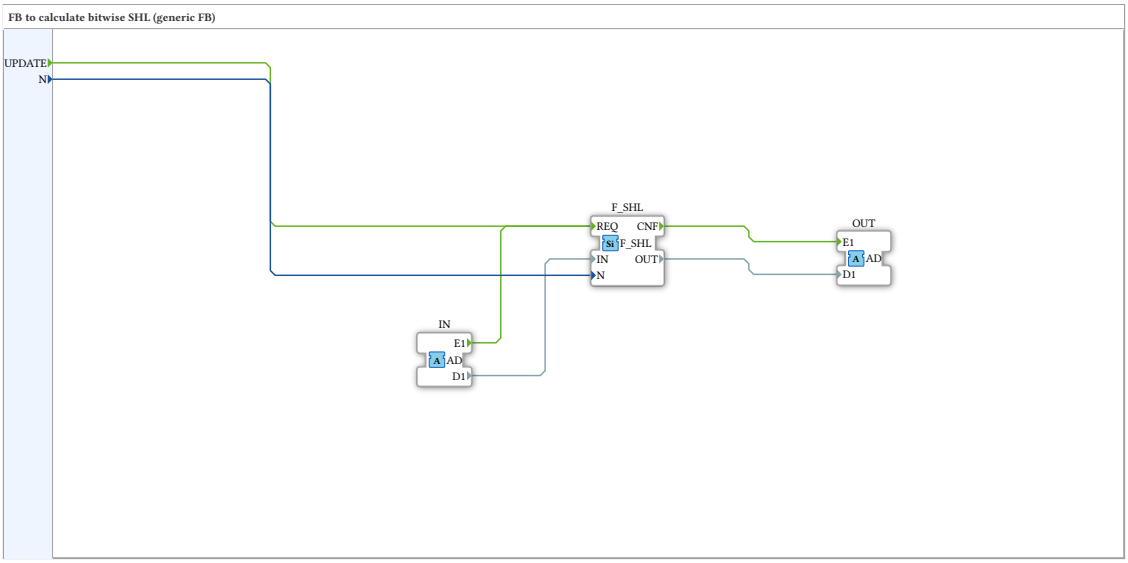
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

bitwiseOperators

AD_SHL





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AD_SHL** führt eine bitweise Linksverschiebung (Shift Left) eines Eingangswertes durch. Er ist als generischer Baustein konzipiert, der über Adapter-Schnittstellen kommuniziert und beliebige ganzzahlige Datentypen (ANY_INT) unterstützt. Die eigentliche Verschiebeoperation wird intern durch einen vordefinierten Baustein `F_SHL` realisiert.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung	Mit-Parameter
`UPDATE`	Aktualisiert die Schiebeanzahl und startet die Berechnung.	`N`

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Das Ereignis zur Weitergabe des Ergebnisses erfolgt über den Adapter `OUT` (Ereignis `OUT.E1`).

Daten-Eingänge

Name	Typ	Beschreibung
`N`	`ANY_INT`	Anzahl der Bit-Positionen, um die verschoben werden soll.

Daten-Ausgänge

Keine direkten Daten-Ausgänge. Das Ergebnis der Verschiebung wird über den Adapter `OUT` (Daten `OUT.D1`) bereitgestellt.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	`IN`	`adapter::types::unidirectional::AD`	Empfängt den zu verschiebenden Wert über den Adapter-Kanal.
Plug	`OUT`	`adapter::types::unidirectional::AD`	Stellt das Ergebnis der Linksverschiebung zur Verfügung.

Funktionsweise

Der FB arbeitet als kompositiver Funktionsblock, der die eigentliche Operation an den Baustein `F_SHL` delegiert. Intern wird folgende Logik umgesetzt:

1. **Ereignisauslösung**
   - Ein eingehendes Ereignis von `UPDATE` oder vom Adapter `IN.E1` triggert den Eingang `REQ` des internen `F_SHL`-Bausteins.
   - Der Wert von `N` wird direkt an den `F_SHL.N`-Eingang übergeben.
   - Der über den Adapter `IN` bereitgestellte Datenwert (`IN.D1`) wird an den `F_SHL.IN`-Eingang weitergeleitet.
2. **Berechnung**
   Der interne Baustein `F_SHL` (Typ `iec61131::bitwiseOperators::F_SHL`) führt die bitweise Linksverschiebung des Eingangswertes um die in `N` angegebene Anzahl durch.
3. **Ausgabe**
   Das Ergebnis der Berechnung (`F_SHL.OUT`) wird auf den Datenausgang des Plug-Adapter `OUT.D1` übertragen. Gleichzeitig signalisiert das Ereignis `F_SHL.CNF` dem Plug-Adapter (`OUT.E1`) das Vorliegen des neuen Ergebnisses.

Der Baustein ist ereignisgesteuert: Jedes Eintreffen eines gültigen Ereignisses (entweder extern über `UPDATE` oder vom Socket-Adapter `IN`) löst eine vollständige Berechnung aus.

Technische Besonderheiten

- **Adapter-Schnittstelle**
  Anstelle fester Ein- und Ausgänge verwendet der FB Adapter (`IN` als Socket, `OUT` als Plug), was eine flexible Anbindung in verschiedenen Architekturkontexten ermöglicht.
- **Generischer Datentyp**
  Der Parameter `N` ist als `ANY_INT` deklariert, sodass der Baustein mit unterschiedlichen ganzzahligen Typen (z. B. `INT`, `DINT`, `LINT`) arbeiten kann, solange der interne `F_SHL` diese unterstützt.
- **Kombinierte Ereignissteuerung**
  Sowohl das externe Ereignis `UPDATE` als auch das vom Socket-Adapter empfangene Ereignis `IN.E1` können die Berechnung anstoßen. Dies ermöglicht sowohl zyklische als auch datengetriebene Aktualisierungen.
- **Lizenzierung**
  Der Baustein ist unter der Eclipse Public License 2.0 lizenziert und als generischer FB konzipiert.

Zustandsübersicht

Der FB ist als **kompositives Netzwerk** realisiert und besitzt keinen eigenen internen Zustandsautomaten (ECC). Die Ablaufsteuerung wird vollständig durch die Ereignisverbindungen innerhalb des Netzwerks und den verwendeten `F_SHL`-Baustein bestimmt. Eine explizite Zustandsmaschine ist nicht erforderlich.

Anwendungsszenarien

- **Bitmanipulation in der Steuerungstechnik**
  Verschieben von Bitmustern zur Maskierung, Skalierung oder Kodierung von Signalen.
- **Datenaufbereitung in Kommunikationsprotokollen**
  Anpassung von Datenworten durch Linksverschiebung, z. B. für serielle Schnittstellen oder Bussysteme.
- **Arithmetische Berechnungen**
  Multiplikation mit Zweierpotenzen durch Linksverschiebung, insbesondere in rechenzeitkritischen Umgebungen.

Vergleich mit ähnlichen Bausteinen

- **Direkter `F_SHL`-Baustein**
  Der interne `F_SHL` bietet die gleiche Funktionalität, jedoch ohne Adapter-Schnittstelle. `AD_SHL` kapselt diesen Baustein und erweitert ihn um eine adapterbasierte Anbindung, was die Wiederverwendung in modularen Systemen erleichtert.
- **Adapterbasierte Rechenbausteine**
  Ähnliche Konzepte existieren für andere bitweise Operationen (z. B. `AD_SHR`, `AD_AND`). `AD_SHL` spezifiziert die Linksverschiebung und kann in einer Bibliothek adaptergekoppelter Logikbausteine verwendet werden.

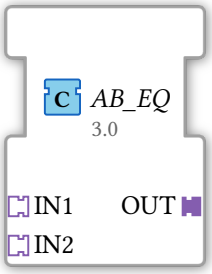
Fazit

Der Funktionsblock `AD_SHL` realisiert eine bitweise Linksverschiebung mittels eines gekapselten `F_SHL`-Bausteins und stellt die gesamte Funktionalität über eine standardisierte Adapter-Schnittstelle zur Verfügung. Durch die generische Auslegung und die flexible Ereignissteuerung eignet er sich für vielfältige Anwendungen in der IEC-61499-basierten Automatisierungstechnik. Seine kompositive Struktur macht ihn wartbar und leicht erweiterbar.

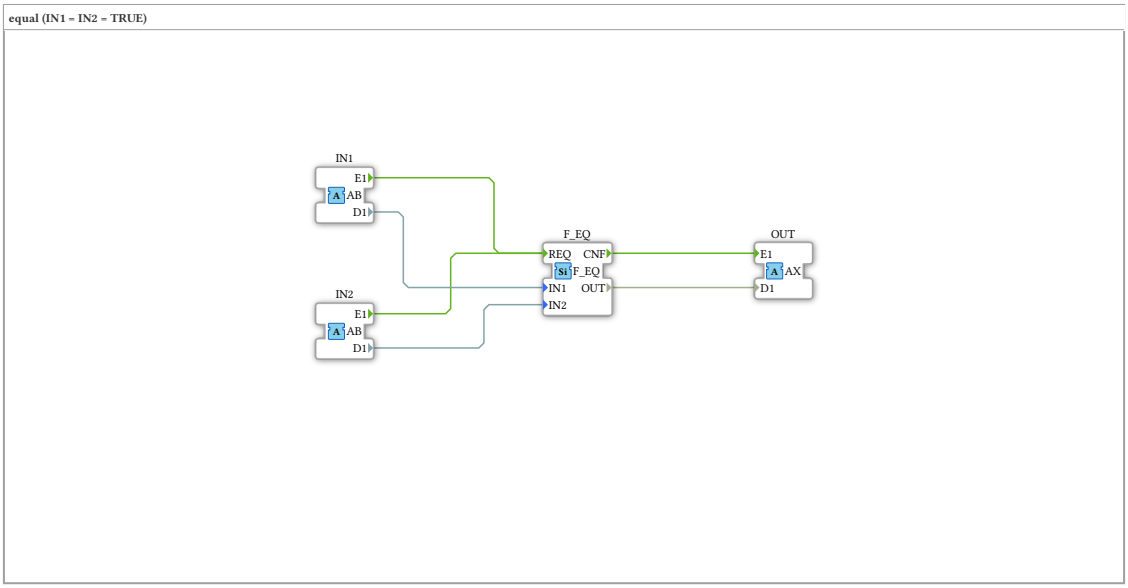
comparison

AB_EQ

Input value 1 -- left operand (IN1 = IN2) – *AB*
Input value 2 -- right operand (IN1 = IN2) – *AB*



AX – IN1 = IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AB_EQ** führt einen Vergleich zweier Werte vom Typ `BYTE` (8-Bit-Bitmuster (Byte)) durch und prüft, ob der erste Wert (IN1) gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `BYTE`, 8-Bit-Bitmuster (Byte)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 = IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	AB (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	AB (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	AX (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter AB und AX sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein F_EQ (Typ iec61131::comparison::F_EQ) weitergeleitet. Dort werden die aktuellen Werte von IN1.D1 und IN2.D1 verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (CNF) und gibt das Boolesche Ergebnis (OUT) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert OUT.D1 wird auf TRUE oder FALSE gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (AB/AX), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („equal to“) für den Datentyp BYTE.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Statusregister gegen einen Referenz- oder Schwellwert vom Typ BYTE prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ BYTE die Bedingung „gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf BYTE-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

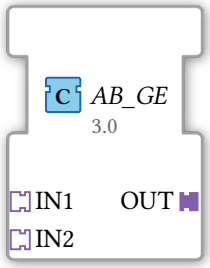
- **AB_NE**: Prüft auf „ungleich“ – invers zur Funktion von AB_EQ.
- **AB_NE**: Prüft auf Ungleichheit der Werte.

Der AB_EQ ist Teil der vollständigen AB-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp BYTE.

Fazit

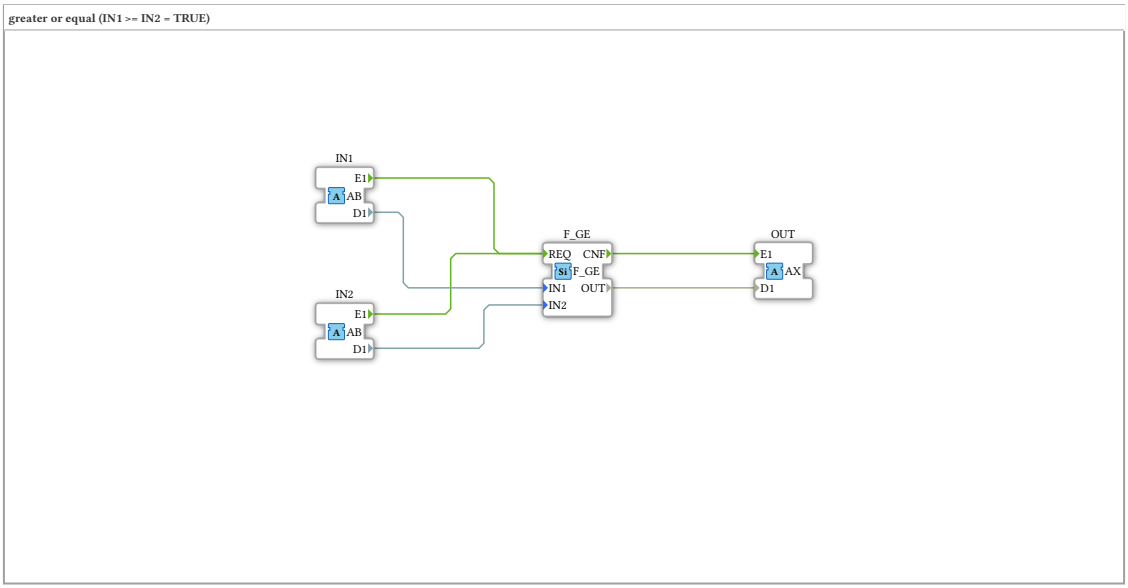
Der **AB_EQ** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp BYTE. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AB_GE



Input value 1 -- left operand (IN1 >= IN2) – *AB*
Input value 2 -- right operand (IN1 >= IN2) – *AB*

AX – IN1 >= IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AB_GE** führt einen Vergleich zweier Werte vom Typ `BYTE` (8-Bit-Bitmuster (Byte)) durch und prüft, ob der erste Wert (IN1) größer gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `BYTE`, 8-Bit-Bitmuster (Byte)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 >= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	AB (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	AB (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	AX (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter AB und AX sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein F_GE (Typ iec61131::comparison::F_GE) weitergeleitet. Dort werden die aktuellen Werte von IN1.D1 und IN2.D1 verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (CNF) und gibt das Boolesche Ergebnis (OUT) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert OUT.D1 wird auf TRUE oder FALSE gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (AB/AX), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than or equal to“) für den Datentyp BYTE.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Statusregister gegen einen Referenz- oder Schwellwert vom Typ BYTE prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ BYTE die Bedingung „größer gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf BYTE-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AB_LE**: Prüft auf „kleiner gleich“ – invers zur Funktion von AB_GE.
- **AB_EQ**: Prüft auf Gleichheit der Werte.
- **AB_GT**: Führt den Vergleich „größer als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version AB_GE ist.

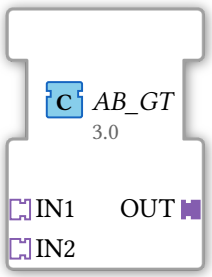
Der AB_GE ergänzt die AB-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

Der **AB_GE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp BYTE. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise

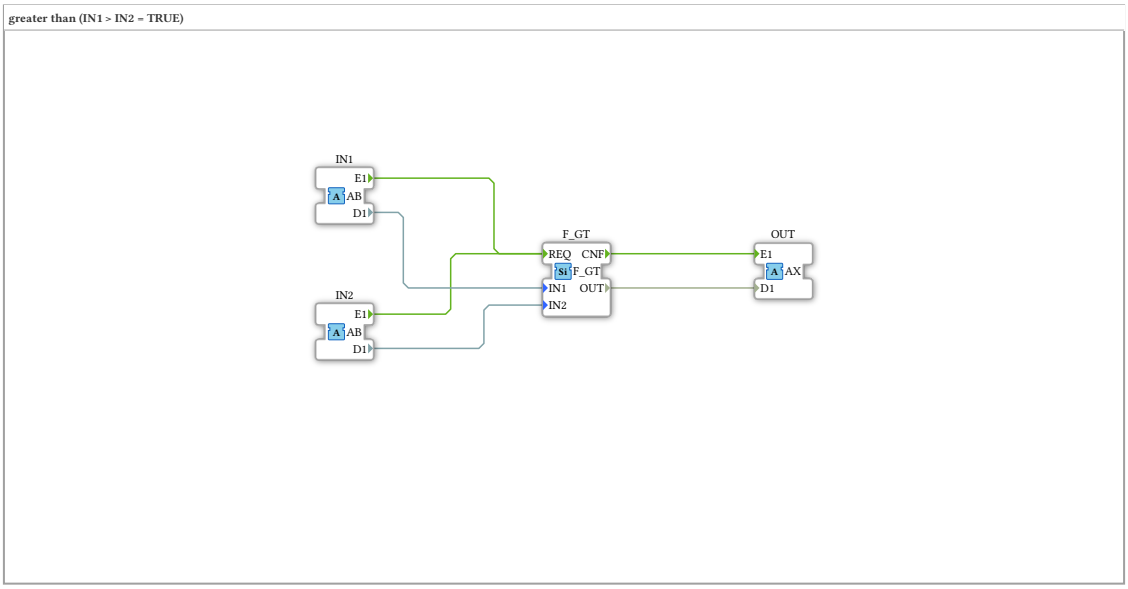
eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AB_GT



Input value 1 -- left operand (IN1 > IN2) – *AB*
Input value 2 -- right operand (IN1 > IN2) – *AB*

AX – IN1 > IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AB_GT** führt einen Vergleich zweier Werte vom Typ `BYTE` (8-Bit-Bitmuster (Byte)) durch und prüft, ob der erste Wert (IN1) größer als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `BYTE`, 8-Bit-Bitmuster (Byte)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 > IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AB` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AB` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AB` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GT` (Typ `iec61131::comparison::F_GT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AB`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than“) für den Datentyp `BYTE`.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Statusregister gegen einen Referenz- oder Schwellwert vom Typ `BYTE` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `BYTE` die Bedingung „größer als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `BYTE`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AB_LT**: Prüft auf „kleiner als“ – invers zur Funktion von AB_GT.
- **AB_EQ**: Prüft auf Gleichheit der Werte.
- **AB_GE**: Führt den Vergleich „größer gleich“ durch – die um den Gleichheitsfall erweiterte Version von AB_GT.

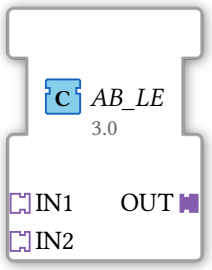
Der AB_GT ist die grundlegende „größer als“-Komponente ohne Gleichheitsfall, ergänzt durch AB_GE.

Fazit

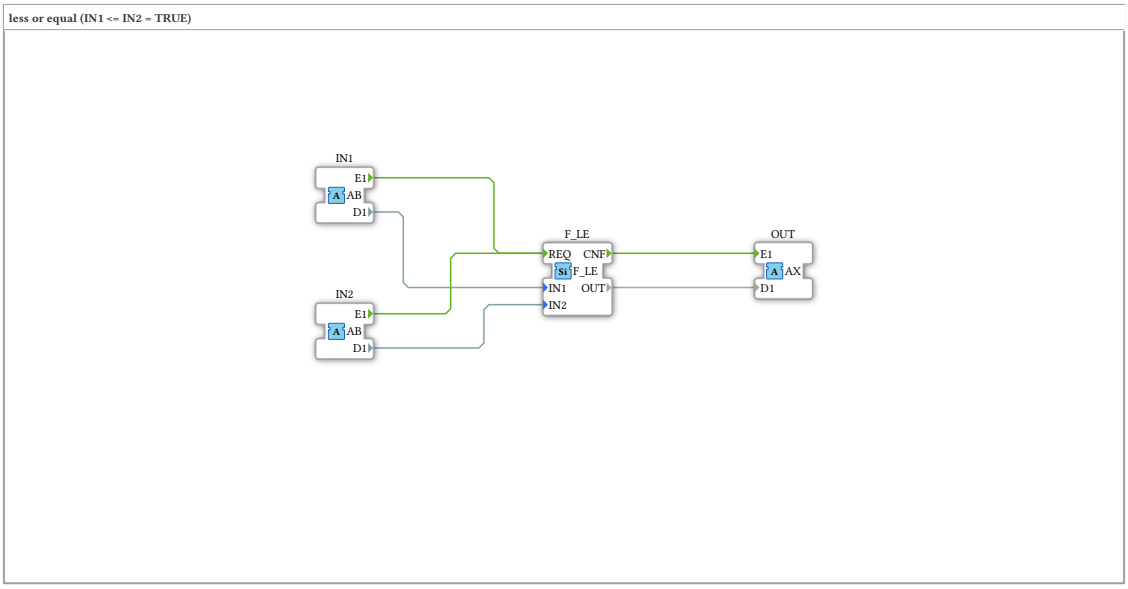
Der **AB_GT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `BYTE`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AB_LE

Input value 1 -- left operand (IN1 <= IN2) – *AB*
Input value 2 -- right operand (IN1 <= IN2) – *AB*



AX – IN1 <= IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AB_LE** führt einen Vergleich zweier Werte vom Typ `BYTE` (8-Bit-Bitmuster (Byte)) durch und prüft, ob der erste Wert (IN1) kleiner gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `BYTE`, 8-Bit-Bitmuster (Byte)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 <= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AB` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AB` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AB` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LE` (Typ `iec61131::comparison::F_LE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AB`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than or equal to“) für den Datentyp `BYTE`.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Statusregister gegen einen Referenz- oder Schwellwert vom Typ `BYTE` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `BYTE` die Bedingung „kleiner gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `BYTE`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AB_GE**: Prüft auf „größer gleich“ – invers zur Funktion von AB_LE.
- **AB_EQ**: Prüft auf Gleichheit der Werte.

- **AB_LT**: Führt den Vergleich „kleiner als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version AB_LE ist.

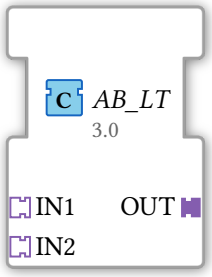
Der AB_LE ergänzt die AB-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

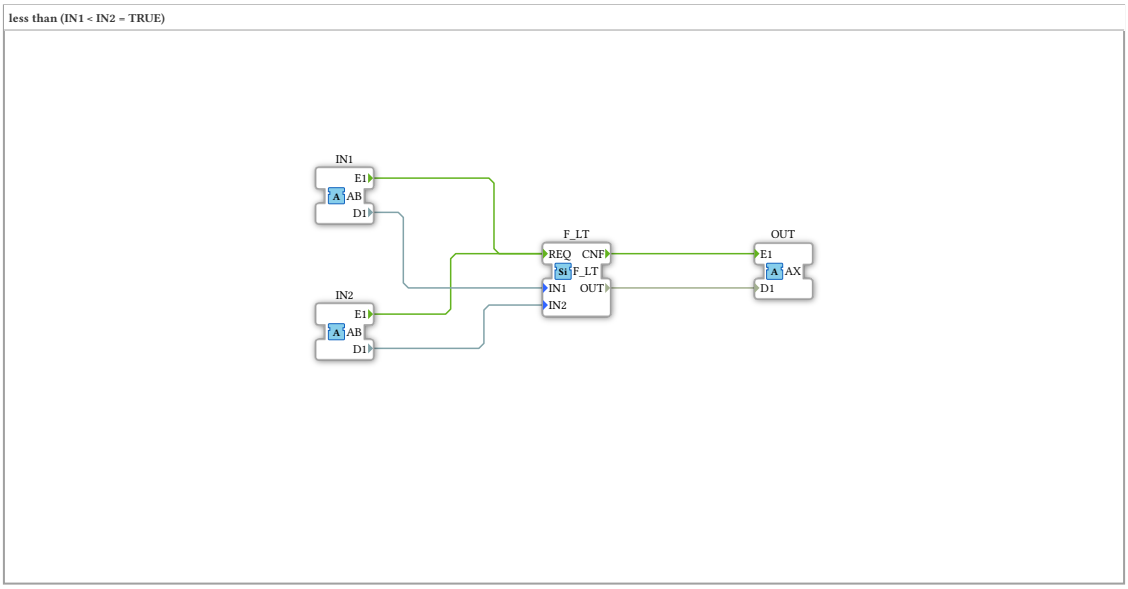
Der **AB_LE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `BYTE`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AB_LT

Input value 1 -- left operand (IN1 < IN2) – *AB*
Input value 2 -- right operand (IN1 < IN2) – *AB*



AX – IN1 < IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AB_LT** führt einen Vergleich zweier Werte vom Typ `BYTE` (8-Bit-Bitmuster (Byte)) durch und prüft, ob der erste Wert (IN1) kleiner als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `BYTE`, 8-Bit-Bitmuster (Byte)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 < IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AB` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AB` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AB` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LT` (Typ `iec61131::comparison::F_LT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AB`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than“) für den Datentyp `BYTE`.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Statusregister gegen einen Referenz- oder Schwellwert vom Typ `BYTE` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `BYTE` die Bedingung „kleiner als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `BYTE`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AB_GT**: Prüft auf „größer als“ – invers zur Funktion von AB_LT.
- **AB_EQ**: Prüft auf Gleichheit der Werte.
- **AB_LE**: Führt den Vergleich „kleiner gleich“ durch – die um den Gleichheitsfall erweiterte Version von AB_LT.

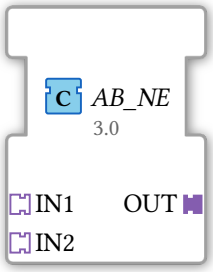
Der AB_LT ist die grundlegende „kleiner als“-Komponente ohne Gleichheitsfall, ergänzt durch AB_LE.

Fazit

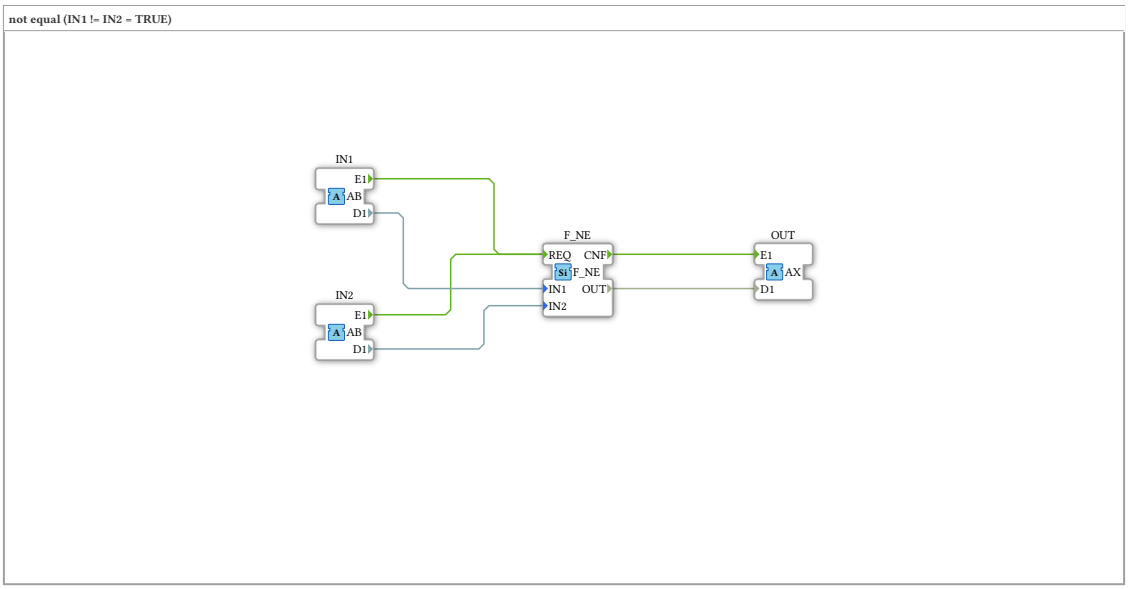
Der **AB_LT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `BYTE`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AB_NE

Input value 1 -- left operand (IN1 != IN2) – *AB*
Input value 2 -- right operand (IN1 != IN2) – *AB*



AX – IN1 != IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AB_NE** führt einen Vergleich zweier Werte vom Typ `BYTE` (8-Bit-Bitmuster (Byte)) durch und prüft, ob der erste Wert (IN1) ungleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `BYTE`, 8-Bit-Bitmuster (Byte)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 != IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AB` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AB` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AB` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_NE` (Typ `iec61131::comparison::F_NE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AB`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („not equal to“) für den Datentyp `BYTE`.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Statusregister gegen einen Referenz- oder Schwellwert vom Typ `BYTE` prüfen.

- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `BYTE` die Bedingung „ungleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `BYTE`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AB_EQ**: Prüft auf „gleich“ – invers zur Funktion von AB_NE.
- **AB_EQ**: Prüft auf Gleichheit der Werte.

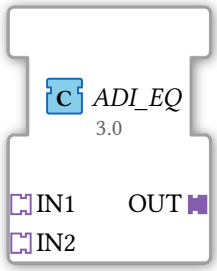
Der AB_NE ist Teil der vollständigen AB-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `BYTE`.

Fazit

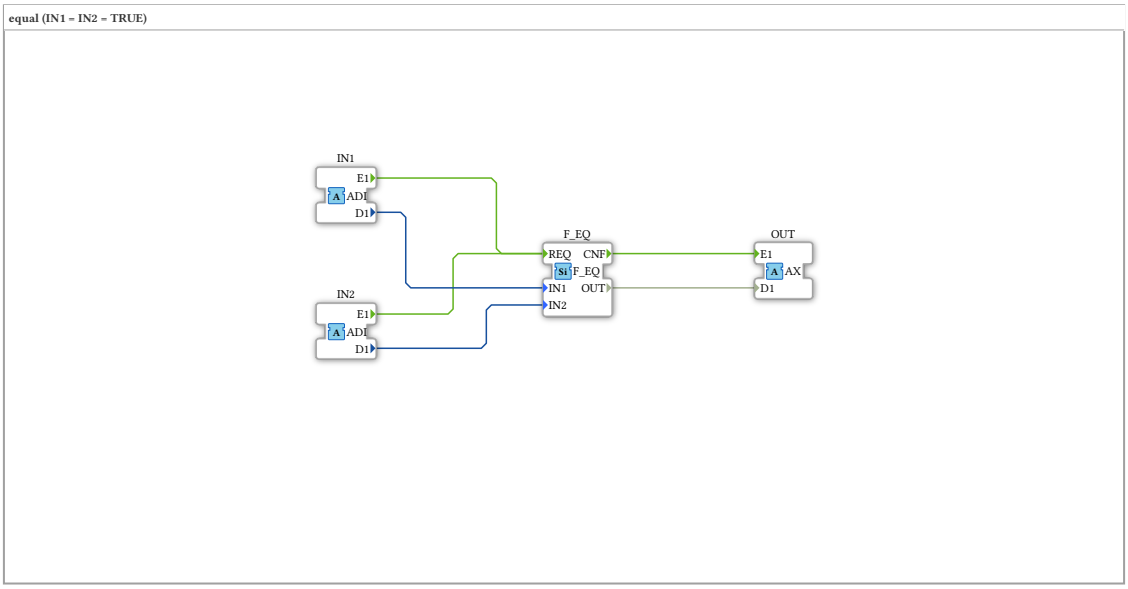
Der **AB_NE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `BYTE`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

ADI_EQ

Input value 1 -- left operand (IN1 = IN2) – *ADI*
Input value 2 -- right operand (IN1 = IN2) – *ADI*



AX – IN1 = IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **ADI_EQ** führt einen Vergleich zweier Werte vom Typ `DINT` (32-Bit-Ganzzahl (vorzeichenbehaftet)) durch und prüft, ob der erste Wert (IN1) gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `DINT`, 32-Bit-Ganzzahl (vorzeichenbehaftet)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 = IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`ADI` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`ADI` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `ADI` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_EQ` (Typ `iec61131::comparison::F_EQ`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`ADI`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („equal to“) für den Datentyp `DINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Zähl- und Positionswerte gegen einen Referenz- oder Schwellwert vom Typ `DINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `DINT` die Bedingung „gleich“ erfüllen.

- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `DINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **ADI_NE**: Prüft auf „ungleich“ – invers zur Funktion von ADI_EQ.
- **ADI_NE**: Prüft auf Ungleichheit der Werte.

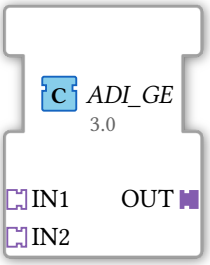
Der ADI_EQ ist Teil der vollständigen ADI-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `DINT`.

Fazit

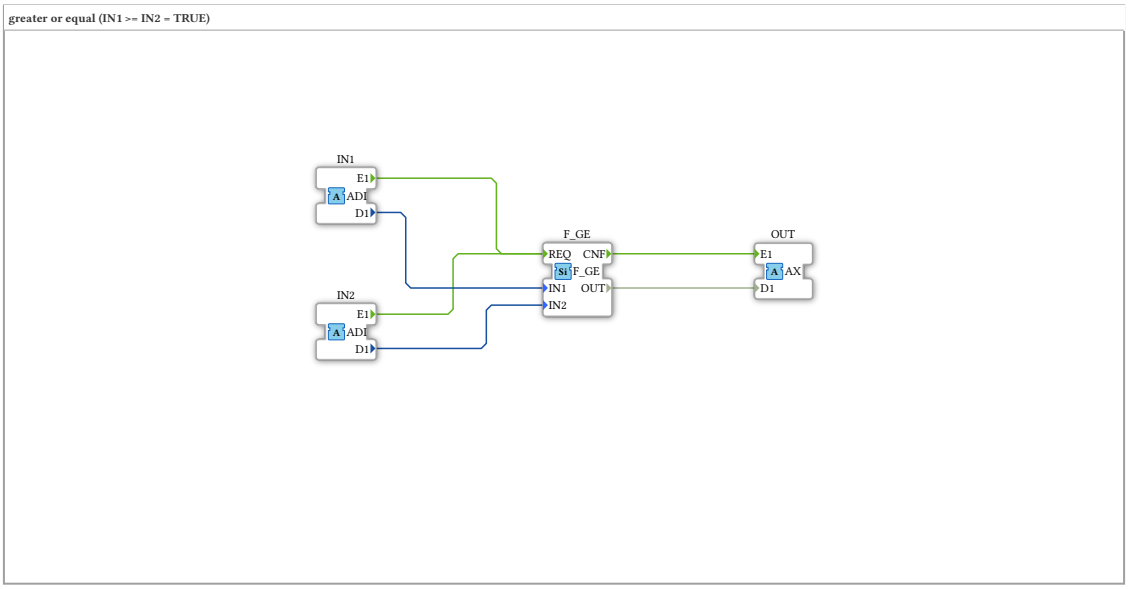
Der **ADI_EQ** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `DINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

ADI_GE

Input value 1 -- left operand (IN1 >= IN2) – *ADI*
Input value 2 -- right operand (IN1 >= IN2) – *ADI*



AX – IN1 >= IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **ADI_GE** führt einen Vergleich zweier Werte vom Typ `DINT` (32-Bit-Ganzzahl (vorzeichenbehaftet)) durch und prüft, ob der erste Wert (IN1) größer gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `DINT`, 32-Bit-Ganzzahl (vorzeichenbehaftet)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 >= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`ADI` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`ADI` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `ADI` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GE` (Typ `iec61131::comparison::F_GE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`ADI`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than or equal to“) für den Datentyp `DINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Zähl- und Positionswerte gegen einen Referenz- oder Schwellwert vom Typ `DINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `DINT` die Bedingung „größer gleich“ erfüllen.

- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `DINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

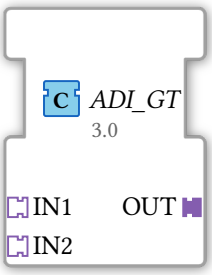
- **ADI_LE**: Prüft auf „kleiner gleich“ – invers zur Funktion von ADI_GE.
- **ADI_EQ**: Prüft auf Gleichheit der Werte.
- **ADI_GT**: Führt den Vergleich „größer als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version ADI_GE ist.

Der ADI_GE ergänzt die ADI-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

Der **ADI_GE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `DINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

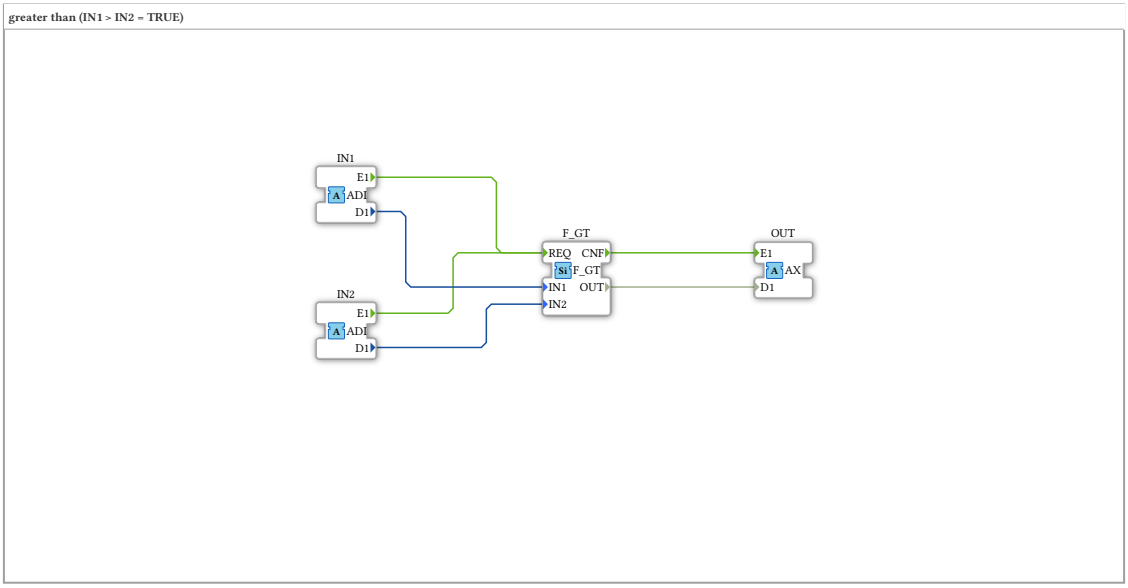
ADI_GT



Input value 1 -- left operand (IN1 > IN2) – *ADI*

Input value 2 -- right operand (IN1 > IN2) – *ADI*

AX – IN1 > IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **ADI_GT** führt einen Vergleich zweier Werte vom Typ `DINT` (32-Bit-Ganzzahl (vorzeichenbehaftet)) durch und prüft, ob der erste Wert (IN1) größer als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `DINT`, 32-Bit-Ganzzahl (vorzeichenbehaftet)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 > IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`ADI` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`ADI` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `ADI` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GT` (Typ `iec61131::comparison::F_GT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`ADI`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than“) für den Datentyp `DINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Zähl- und Positionswerte gegen einen Referenz- oder Schwellwert vom Typ `DINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `DINT` die Bedingung „größer als“ erfüllen.

- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `DINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **ADI_LT**: Prüft auf „kleiner als“ – invers zur Funktion von ADI_GT.
- **ADI_EQ**: Prüft auf Gleichheit der Werte.
- **ADI_GE**: Führt den Vergleich „größer gleich“ durch – die um den Gleichheitsfall erweiterte Version von ADI_GT.

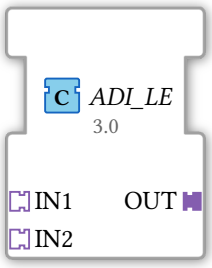
Der ADI_GT ist die grundlegende „größer als“-Komponente ohne Gleichheitsfall, ergänzt durch ADI_GE.

Fazit

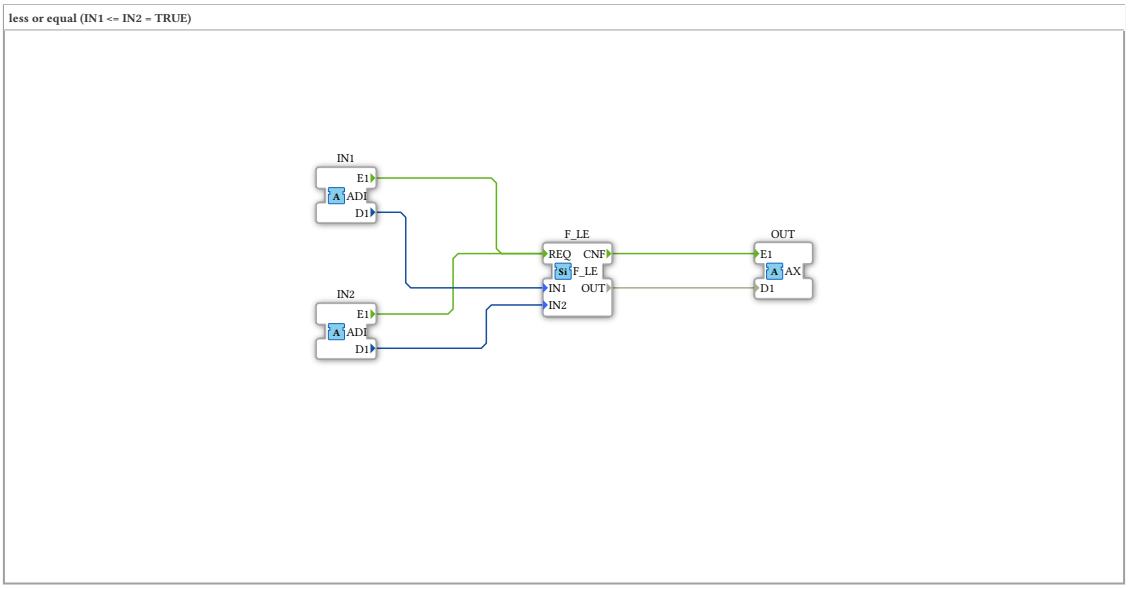
Der **ADI_GT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `DINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

ADI_LE

Input value 1 -- left operand (IN1 <= IN2) – *ADI*
Input value 2 -- right operand (IN1 <= IN2) – *ADI*



AX – IN1 <= IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **ADI_LE** führt einen Vergleich zweier Werte vom Typ `DINT` (32-Bit-Ganzzahl (vorzeichenbehaftet)) durch und prüft, ob der erste Wert (IN1) kleiner gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `DINT`, 32-Bit-Ganzzahl (vorzeichenbehaftet)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 <= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`ADI` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`ADI` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `ADI` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LE` (Typ `iec61131::comparison::F_LE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`ADI`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than or equal to“) für den Datentyp `DINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Zähl- und Positionswerte gegen einen Referenz- oder Schwellwert vom Typ `DINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `DINT` die Bedingung „kleiner gleich“ erfüllen.

- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `DINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

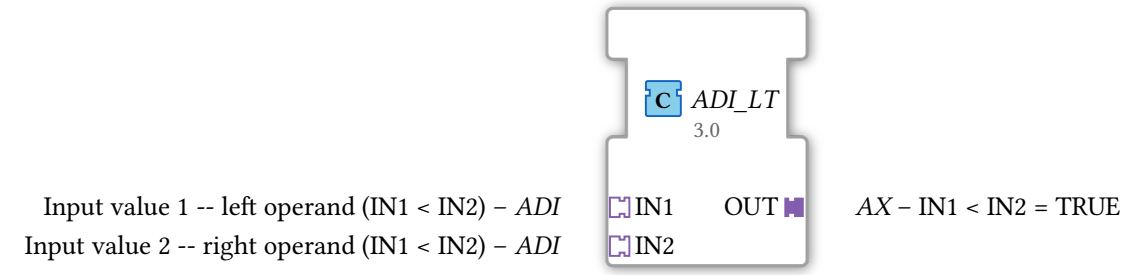
- **ADI_GE**: Prüft auf „größer gleich“ – invers zur Funktion von ADI_LE.
- **ADI_EQ**: Prüft auf Gleichheit der Werte.
- **ADI_LT**: Führt den Vergleich „kleiner als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version ADI_LE ist.

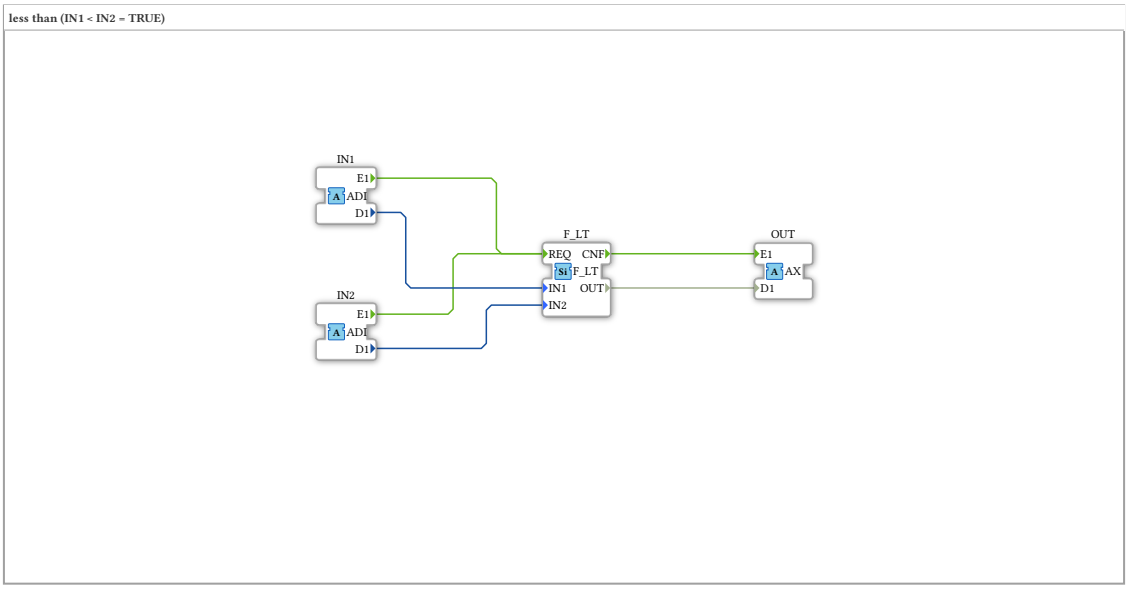
Der ADI_LE ergänzt die ADI-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

Der **ADI_LE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `DINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

ADI_LT





• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **ADI_LT** führt einen Vergleich zweier Werte vom Typ `DINT` (32-Bit-Ganzzahl (vorzeichenbehaftet)) durch und prüft, ob der erste Wert (IN1) kleiner als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `DINT`, 32-Bit-Ganzzahl (vorzeichenbehaftet)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 < IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`ADI` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`ADI` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `ADI` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LT` (Typ `iec61131::comparison::F_LT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`ADI`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than“) für den Datentyp `DINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Zähl- und Positionswerte gegen einen Referenz- oder Schwellwert vom Typ `DINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `DINT` die Bedingung „kleiner als“ erfüllen.

- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `DINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **ADI_GT**: Prüft auf „größer als“ – invers zur Funktion von ADI_LT.
- **ADI_EQ**: Prüft auf Gleichheit der Werte.
- **ADI_LE**: Führt den Vergleich „kleiner gleich“ durch – die um den Gleichheitsfall erweiterte Version von ADI_LT.

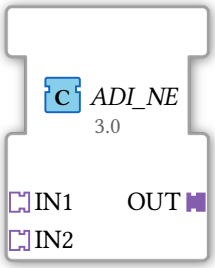
Der ADI_LT ist die grundlegende „kleiner als“-Komponente ohne Gleichheitsfall, ergänzt durch ADI_LE.

Fazit

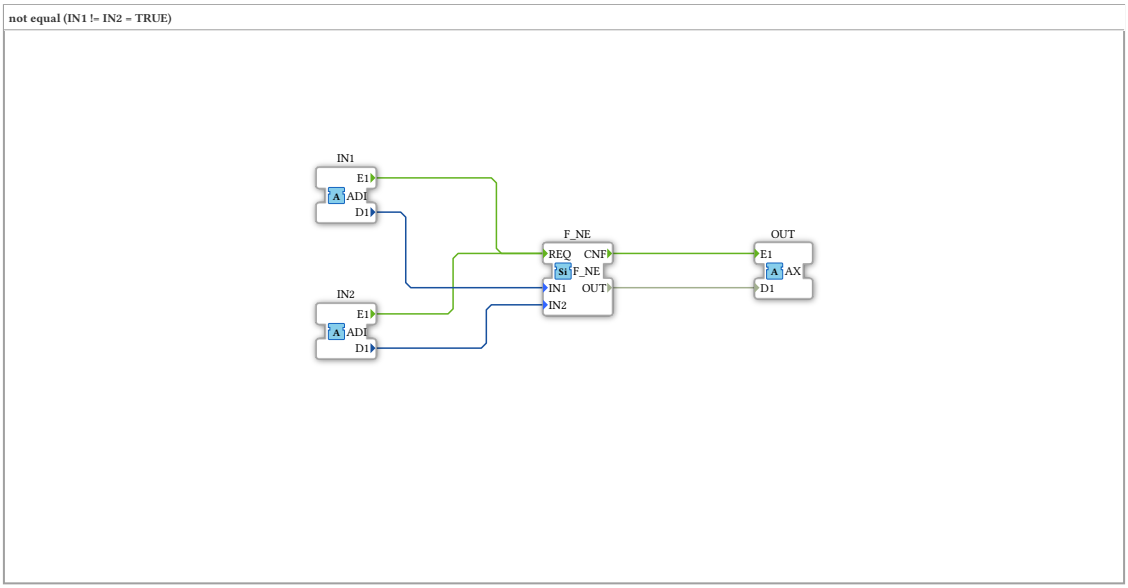
Der **ADI_LT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `DINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

ADI_NE

Input value 1 -- left operand (IN1 != IN2) – *ADI*
Input value 2 -- right operand (IN1 != IN2) – *ADI*



AX – IN1 != IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **ADI_NE** führt einen Vergleich zweier Werte vom Typ `DINT` (32-Bit-Ganzzahl (vorzeichenbehaftet)) durch und prüft, ob der erste Wert (IN1) ungleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `DINT`, 32-Bit-Ganzzahl (vorzeichenbehaftet)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 != IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`ADI` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`ADI` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `ADI` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_NE` (Typ `iec61131::comparison::F_NE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`ADI`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („not equal to“) für den Datentyp `DINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Zähl- und Positionswerte gegen einen Referenz- oder Schwellwert vom Typ `DINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `DINT` die Bedingung „ungleich“ erfüllen.

- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `DINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **ADI_EQ**: Prüft auf „gleich“ – invers zur Funktion von ADI_NE.
- **ADI_EQ**: Prüft auf Gleichheit der Werte.

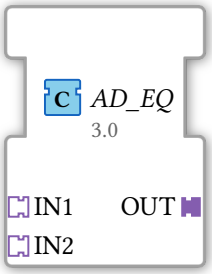
Der ADI_NE ist Teil der vollständigen ADI-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `DINT`.

Fazit

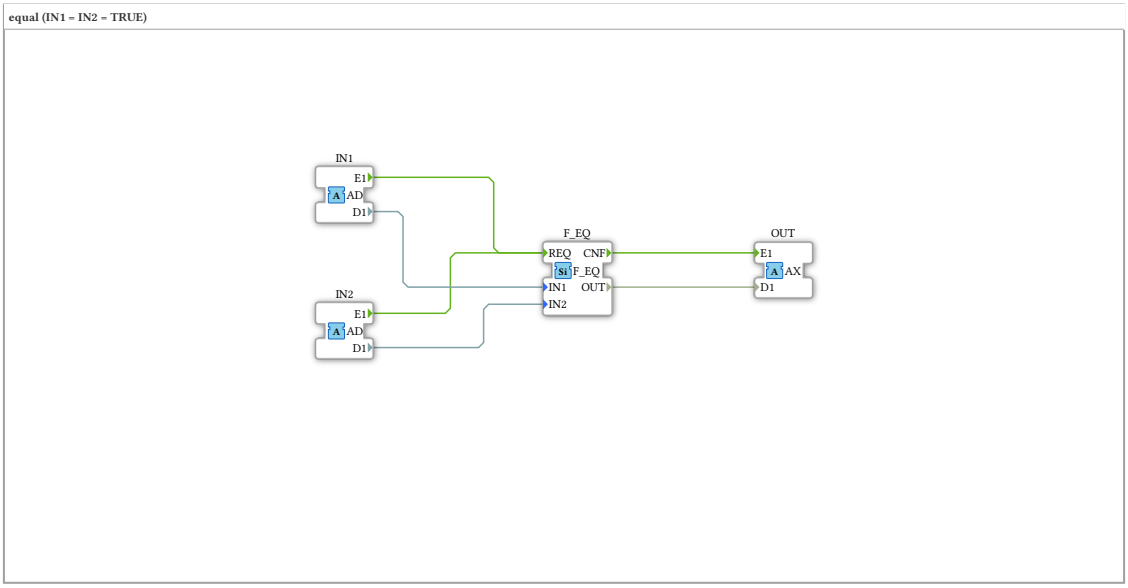
Der **ADI_NE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `DINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AD_EQ

Input value 1 -- left operand (IN1 = IN2) – *AD*
Input value 2 -- right operand (IN1 = IN2) – *AD*



AX – IN1 = IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AD_EQ** führt einen Vergleich zweier Werte vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)) durch und prüft, ob der erste Wert (IN1) gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `DWORD`, 32-Bit-Bitmuster (Doppelwort)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 = IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AD` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AD` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AD` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_EQ` (Typ `iec61131::comparison::F_EQ`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AD`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („equal to“) für den Datentyp `DWORD`.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Bitmasken gegen einen Referenz- oder Schwellwert vom Typ `DWORD` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `DWORD` die Bedingung „gleich“ erfüllen.

- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `DWORD`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AD_NE**: Prüft auf „ungleich“ – invers zur Funktion von AD_EQ.
- **AD_NE**: Prüft auf Ungleichheit der Werte.

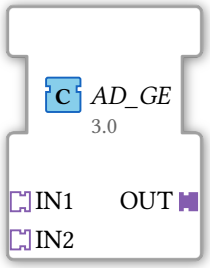
Der AD_EQ ist Teil der vollständigen AD-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `DWORD`.

Fazit

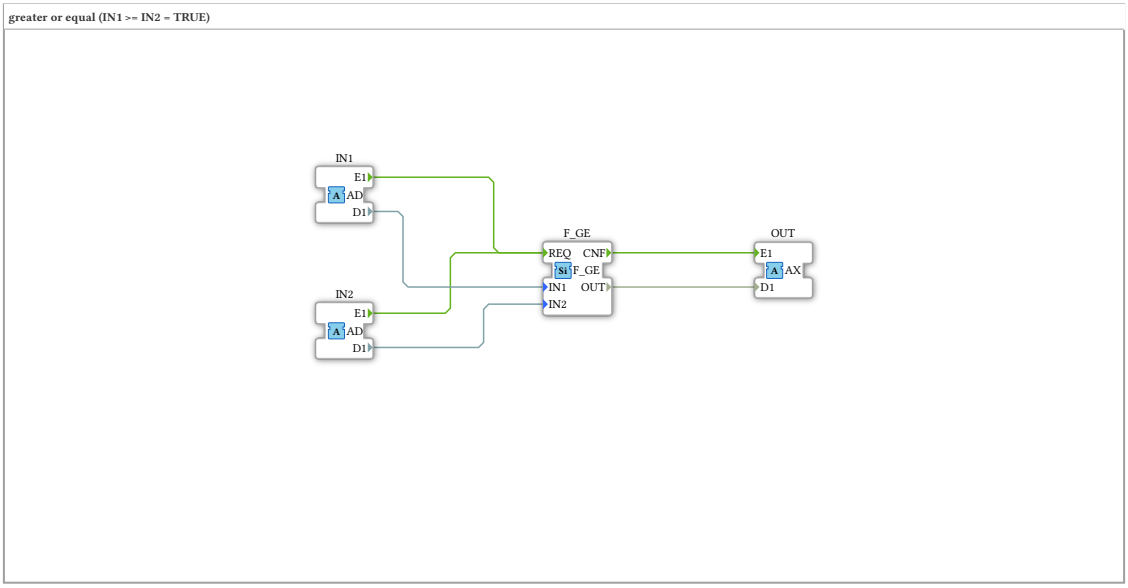
Der **AD_EQ** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `DWORD`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AD_GE

Input value 1 -- left operand (IN1 >= IN2) – *AD*
Input value 2 -- right operand (IN1 >= IN2) – *AD*



AX – IN1 >= IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AD_GE** führt einen Vergleich zweier Werte vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)) durch und prüft, ob der erste Wert (IN1) größer gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `DWORD`, 32-Bit-Bitmuster (Doppelwort)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 >= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AD` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AD` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AD` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GE` (Typ `iec61131::comparison::F_GE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AD`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than or equal to“) für den Datentyp `DWORD`.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Bitmasken gegen einen Referenz- oder Schwellwert vom Typ `DWORD` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `DWORD` die Bedingung „größer gleich“ erfüllen.

- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `DWORD`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

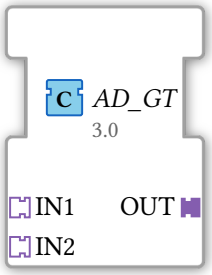
- **AD_LE**: Prüft auf „kleiner gleich“ – invers zur Funktion von AD_GE.
- **AD_EQ**: Prüft auf Gleichheit der Werte.
- **AD_GT**: Führt den Vergleich „größer als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version AD_GE ist.

Der AD_GE ergänzt die AD-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

Der **AD_GE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `DWORD`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

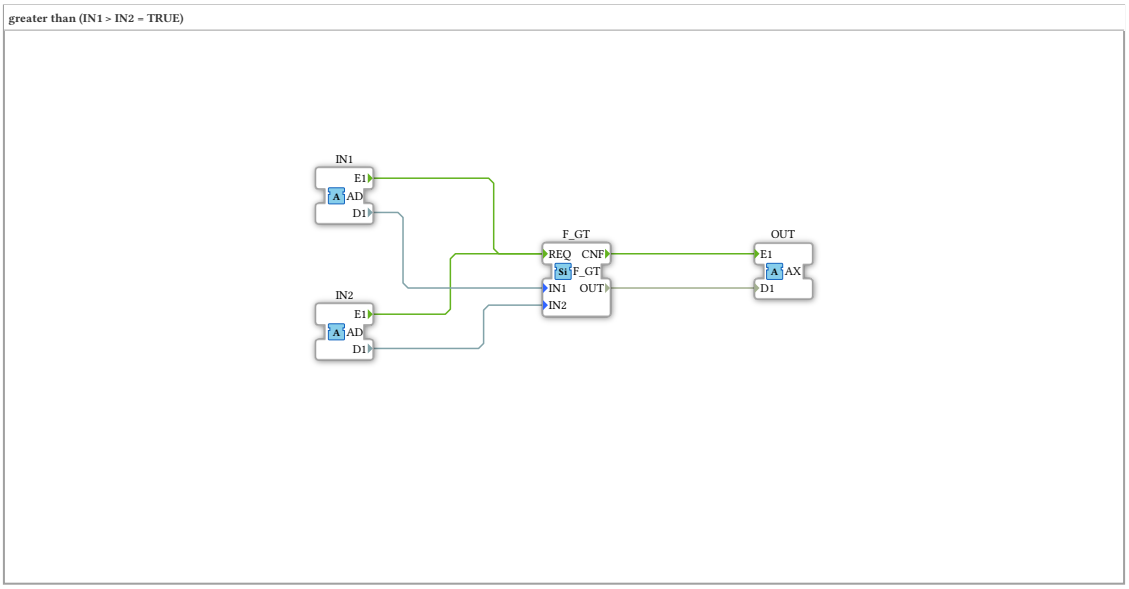
AD_GT



Input value 1 -- left operand (IN1 > IN2) – *AD*

Input value 2 -- right operand (IN1 > IN2) – *AD*

AX – IN1 > IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AD_GT** führt einen Vergleich zweier Werte vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)) durch und prüft, ob der erste Wert (IN1) größer als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `DWORD`, 32-Bit-Bitmuster (Doppelwort)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 > IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AD` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AD` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AD` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GT` (Typ `iec61131::comparison::F_GT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AD`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than“) für den Datentyp `DWORD`.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Bitmasken gegen einen Referenz- oder Schwellwert vom Typ `DWORD` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `DWORD` die Bedingung „größer als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `DWORD`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AD_LT**: Prüft auf „kleiner als“ – invers zur Funktion von AD_GT.
- **AD_EQ**: Prüft auf Gleichheit der Werte.
- **AD_GE**: Führt den Vergleich „größer gleich“ durch – die um den Gleichheitsfall erweiterte Version von AD_GT.

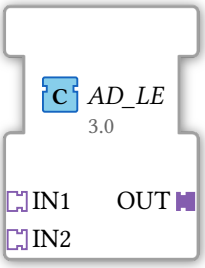
Der AD_GT ist die grundlegende „größer als“-Komponente ohne Gleichheitsfall, ergänzt durch AD_GE.

Fazit

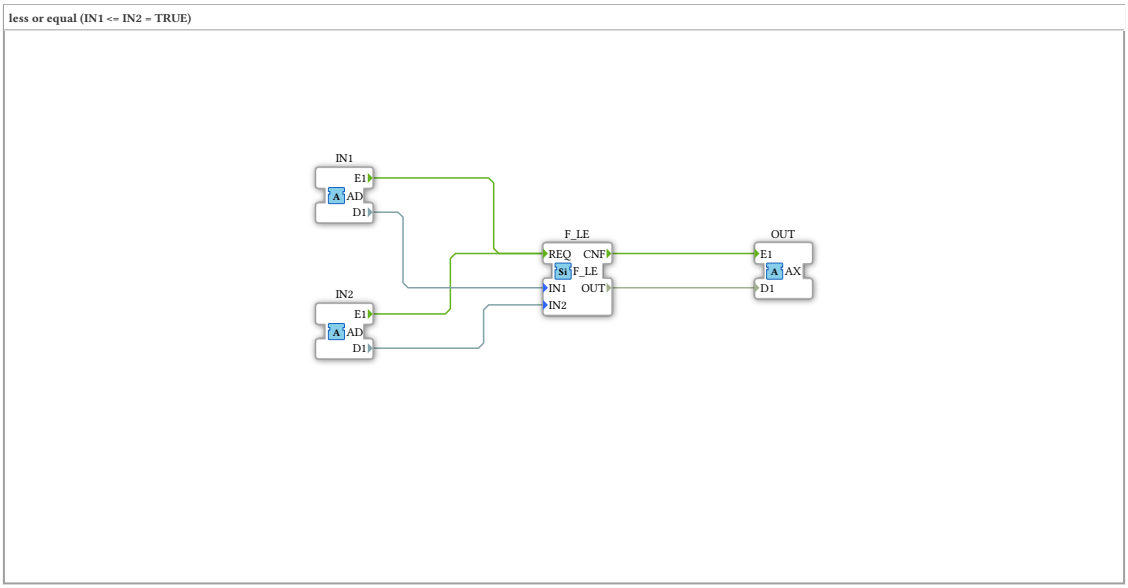
Der **AD_GT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `DWORD`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AD_LE

Input value 1 -- left operand (IN1 <= IN2) – *AD*
Input value 2 -- right operand (IN1 <= IN2) – *AD*



AX – IN1 <= IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AD_LE** führt einen Vergleich zweier Werte vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)) durch und prüft, ob der erste Wert (IN1) kleiner gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und

arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `DWORD`, 32-Bit-Bitmuster (Doppelwort)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 <= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AD` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AD` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AD` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LE` (Typ `iec61131::comparison::F_LE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AD`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than or equal to“) für den Datentyp `DWORD`.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Bitmasken gegen einen Referenz- oder Schwellwert vom Typ `DWORD` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `DWORD` die Bedingung „kleiner gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `DWORD`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

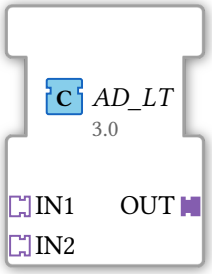
- **AD_GE**: Prüft auf „größer gleich“ – invers zur Funktion von AD_LE.
- **AD_EQ**: Prüft auf Gleichheit der Werte.
- **AD_LT**: Führt den Vergleich „kleiner als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version AD_LE ist.

Der AD_LE ergänzt die AD-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

Der **AD_LE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `DWORD`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

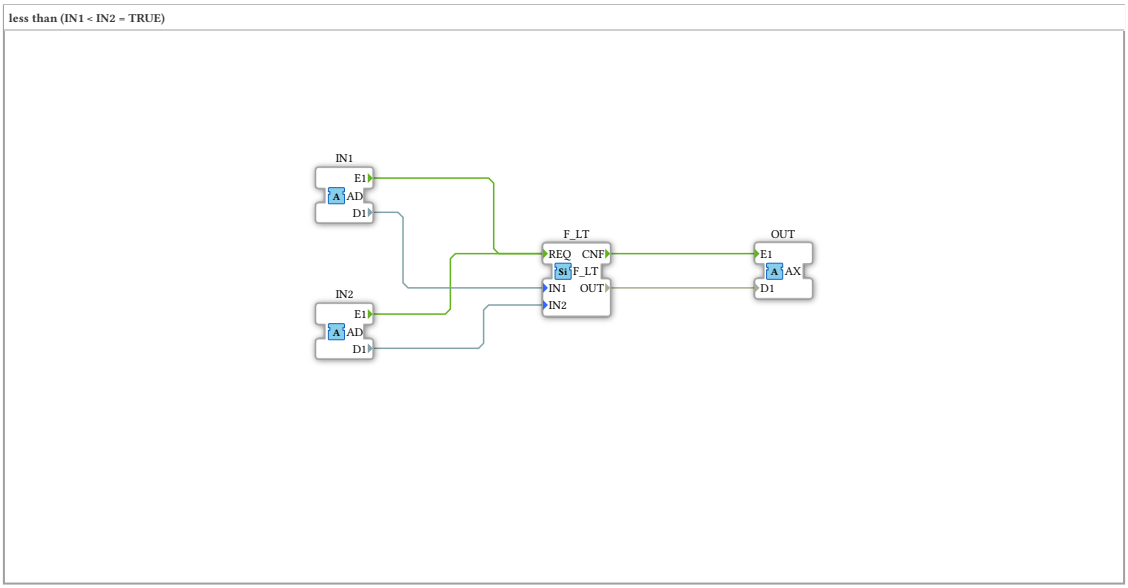
AD_LT



Input value 1 -- left operand (IN1 < IN2) – *AD*

Input value 2 -- right operand (IN1 < IN2) – *AD*

AX – IN1 < IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AD_LT** führt einen Vergleich zweier Werte vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)) durch und prüft, ob der erste Wert (IN1) kleiner als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `DWORD`, 32-Bit-Bitmuster (Doppelwort)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 < IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AD` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AD` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AD` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LT` (Typ `iec61131::comparison::F_LT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AD`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than“) für den Datentyp `DWORD`.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Bitmasken gegen einen Referenz- oder Schwellwert vom Typ `DWORD` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `DWORD` die Bedingung „kleiner als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `DWORD`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

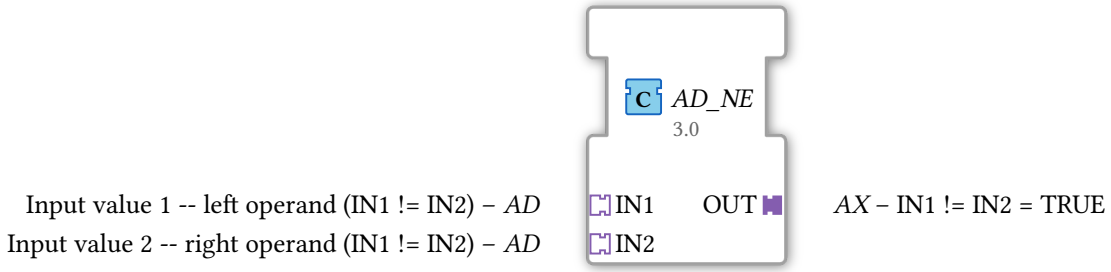
- **AD_GT**: Prüft auf „größer als“ – invers zur Funktion von AD_LT.
- **AD_EQ**: Prüft auf Gleichheit der Werte.
- **AD_LE**: Führt den Vergleich „kleiner gleich“ durch – die um den Gleichheitsfall erweiterte Version von AD_LT.

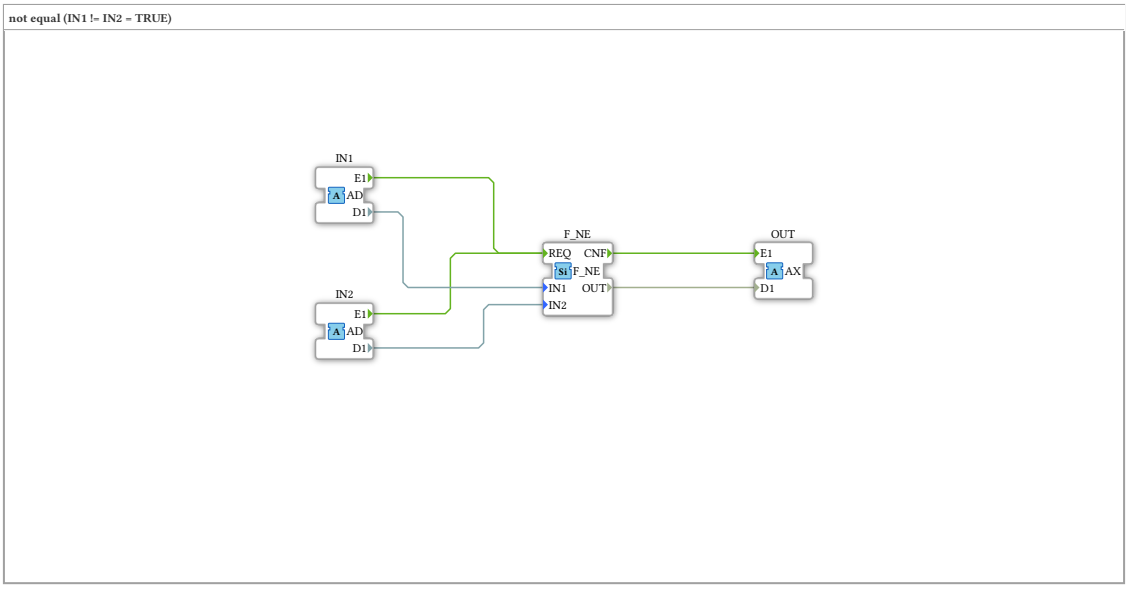
Der AD_LT ist die grundlegende „kleiner als“-Komponente ohne Gleichheitsfall, ergänzt durch AD_LE.

Fazit

Der **AD_LT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `DWORD`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AD_NE





• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AD_NE** führt einen Vergleich zweier Werte vom Typ `DWORD` (32-Bit-Bitmuster (Doppelwort)) durch und prüft, ob der erste Wert (IN1) ungleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `DWORD`, 32-Bit-Bitmuster (Doppelwort)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 != IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AD` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AD` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter AD und AX sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein F_NE (Typ iec61131::comparison::F_NE) weitergeleitet. Dort werden die aktuellen Werte von IN1.D1 und IN2.D1 verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (CNF) und gibt das Boolesche Ergebnis (OUT) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert OUT.D1 wird auf TRUE oder FALSE gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (AD/AX), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („not equal to“) für den Datentyp DWORD.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Bitmasken gegen einen Referenz- oder Schwellwert vom Typ DWORD prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ DWORD die Bedingung „ungleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf DWORD-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

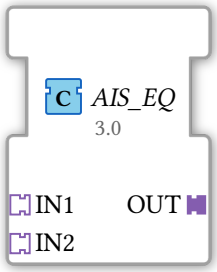
- **AD_EQ**: Prüft auf „gleich“ – invers zur Funktion von AD_NE.
- **AD_EQ**: Prüft auf Gleichheit der Werte.

Der AD_NE ist Teil der vollständigen AD-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp DWORD.

Fazit

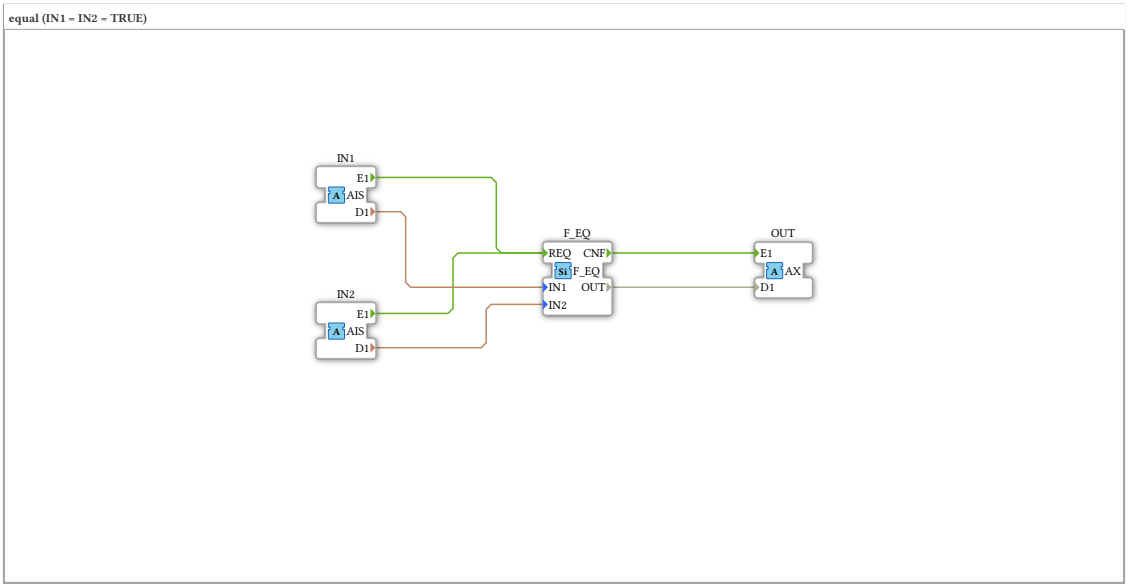
Der **AD_NE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp DWORD. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AIS_EQ



Input value 1 -- left operand (IN1 = IN2) – *AIS*
Input value 2 -- right operand (IN1 = IN2) – *AIS*

AX – IN1 = IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AIS_EQ** führt einen Vergleich zweier Werte vom Typ `STRING` (Zeichenkette) durch und prüft, ob der erste Wert (IN1) gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt lexikografisch (zeichenweise) gemäß der Zeichenkodierung.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `STRING`, Zeichenkette).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 = IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AIS` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AIS` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AIS` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_EQ` (Typ `iec61131::comparison::F_EQ`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AIS`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („equal to“) für den Datentyp `STRING`.
- **Vergleichslogik**: Der Vergleich erfolgt lexikografisch (zeichenweise) gemäß der Zeichenkodierung.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Rezept- und Textkennungen gegen einen Referenz- oder Schwellwert vom Typ `STRING` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `STRING` die Bedingung „gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `STRING`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

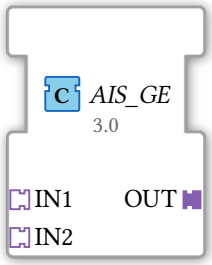
- **AIS_NE**: Prüft auf „ungleich“ – invers zur Funktion von AIS_EQ.
- **AIS_NE**: Prüft auf Ungleichheit der Werte.

Der AIS_EQ ist Teil der vollständigen AIS-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `STRING`.

Fazit

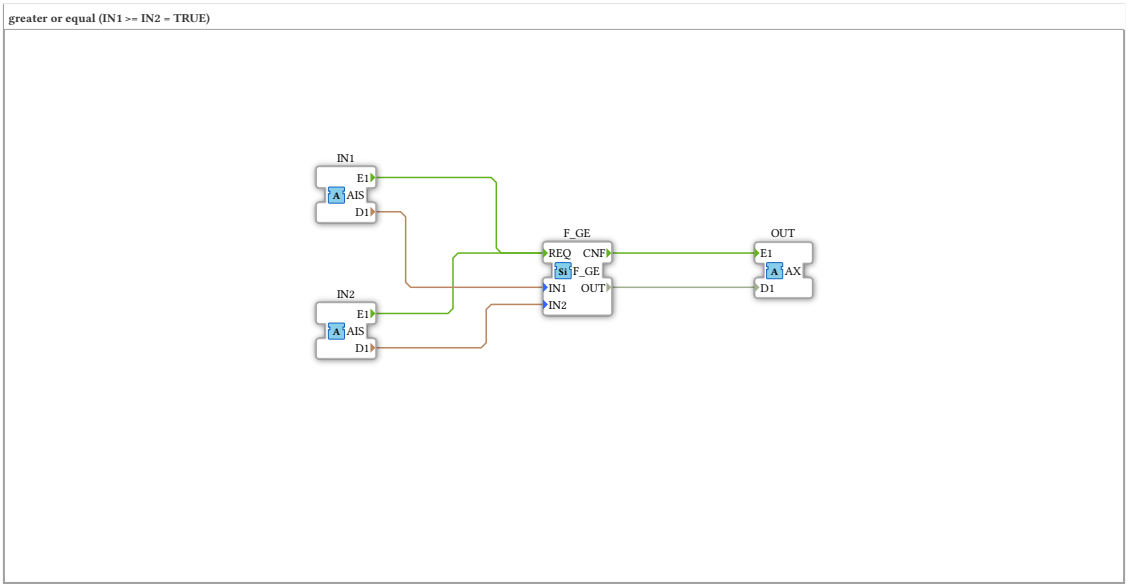
Der **AIS_EQ** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `STRING`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AIS_GE



Input value 1 -- left operand (IN1 >= IN2) – *AIS*
Input value 2 -- right operand (IN1 >= IN2) – *AIS*

AX – IN1 >= IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AIS_GE** führt einen Vergleich zweier Werte vom Typ `STRING` (Zeichenkette) durch und prüft, ob der erste Wert (IN1) größer gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt lexikografisch (zeichenweise) gemäß der Zeichenkodierung.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `STRING`, Zeichenkette).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 >= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AIS` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AIS` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AIS` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GE` (Typ `iec61131::comparison::F_GE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AIS`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than or equal to“) für den Datentyp `STRING`.
- **Vergleichslogik**: Der Vergleich erfolgt lexikografisch (zeichenweise) gemäß der Zeichenkodierung.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Rezept- und Textkennungen gegen einen Referenz- oder Schwellwert vom Typ `STRING` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `STRING` die Bedingung „größer gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `STRING`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AIS_LE**: Prüft auf „kleiner gleich“ – invers zur Funktion von AIS_GE.
- **AIS_EQ**: Prüft auf Gleichheit der Werte.
- **AIS_GT**: Führt den Vergleich „größer als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version AIS_GE ist.

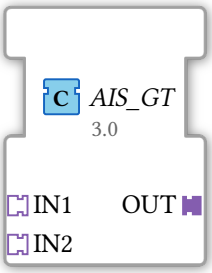
Der AIS_GE ergänzt die AIS-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

Der **AIS_GE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `STRING`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeits-

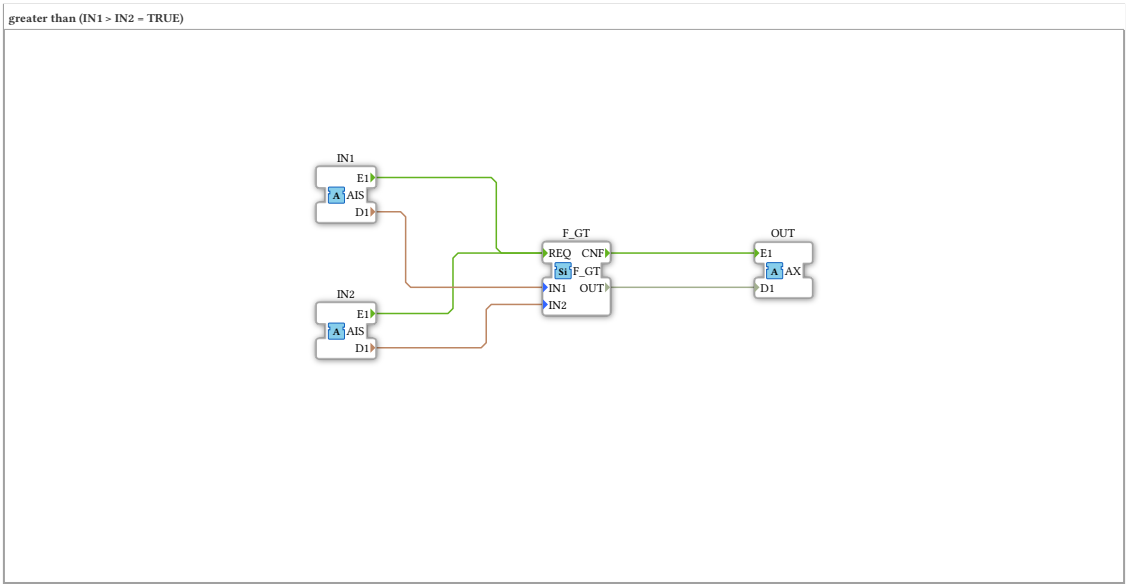
weise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AIS_GT



Input value 1 -- left operand (IN1 > IN2) – *AIS*
Input value 2 -- right operand (IN1 > IN2) – *AIS*

AX – IN1 > IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AIS_GT** führt einen Vergleich zweier Werte vom Typ `STRING` (Zeichenkette) durch und prüft, ob der erste Wert (IN1) größer als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt lexikografisch (zeichenweise) gemäß der Zeichenkodierung.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `STRING`, Zeichenkette).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 > IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AIS` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AIS` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AIS` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GT` (Typ `iec61131::comparison::F_GT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AIS`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than“) für den Datentyp `STRING`.
- **Vergleichslogik**: Der Vergleich erfolgt lexikografisch (zeichenweise) gemäß der Zeichenkodierung.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Rezept- und Textkennungen gegen einen Referenz- oder Schwellwert vom Typ `STRING` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `STRING` die Bedingung „größer als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `STRING`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AIS_LT**: Prüft auf „kleiner als“ – invers zur Funktion von AIS_GT.
- **AIS_EQ**: Prüft auf Gleichheit der Werte.
- **AIS_GE**: Führt den Vergleich „größer gleich“ durch – die um den Gleichheitsfall erweiterte Version von AIS_GT.

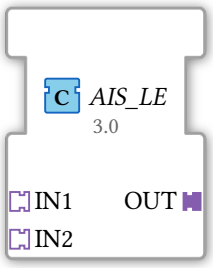
Der AIS_GT ist die grundlegende „größer als“-Komponente ohne Gleichheitsfall, ergänzt durch AIS_GE.

Fazit

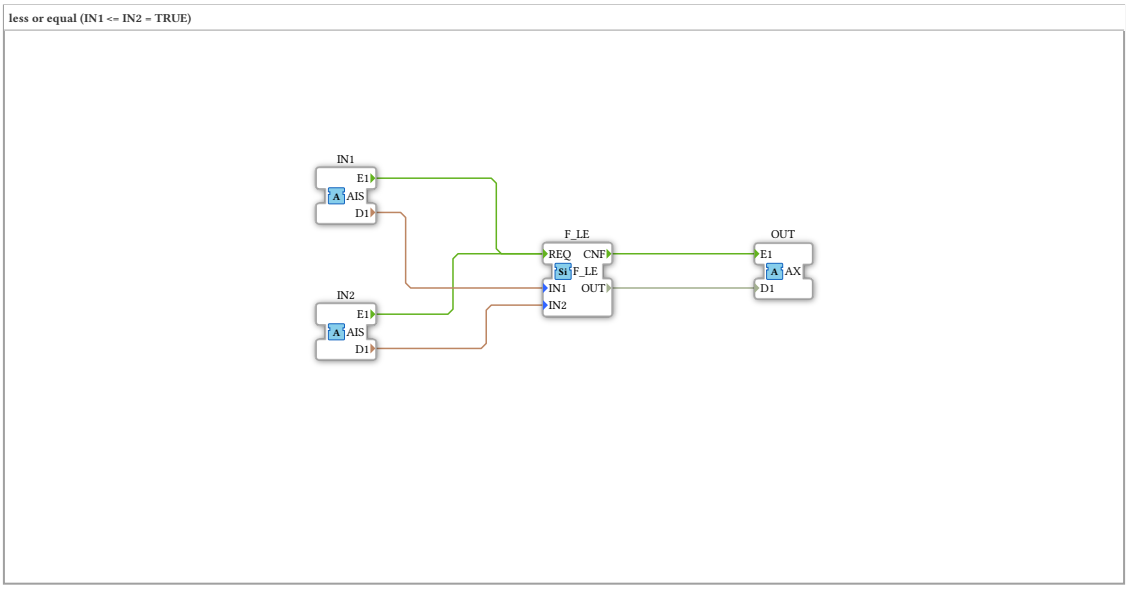
Der **AIS_GT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `STRING`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AIS_LE

Input value 1 -- left operand (IN1 <= IN2) – *AIS*
Input value 2 -- right operand (IN1 <= IN2) – *AIS*



AX – IN1 <= IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AIS_LE** führt einen Vergleich zweier Werte vom Typ `STRING` (Zeichenkette) durch und prüft, ob der erste Wert (IN1) kleiner gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt lexikografisch (zeichenweise) gemäß der Zeichenkodierung.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `STRING`, Zeichenkette).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 <= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AIS` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AIS` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AIS` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LE` (Typ `iec61131::comparison::F_LE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AIS`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than or equal to“) für den Datentyp `STRING`.
- **Vergleichslogik**: Der Vergleich erfolgt lexikografisch (zeichenweise) gemäß der Zeichenkodierung.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Rezept- und Textkennungen gegen einen Referenz- oder Schwellwert vom Typ `STRING` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `STRING` die Bedingung „kleiner gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `STRING`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

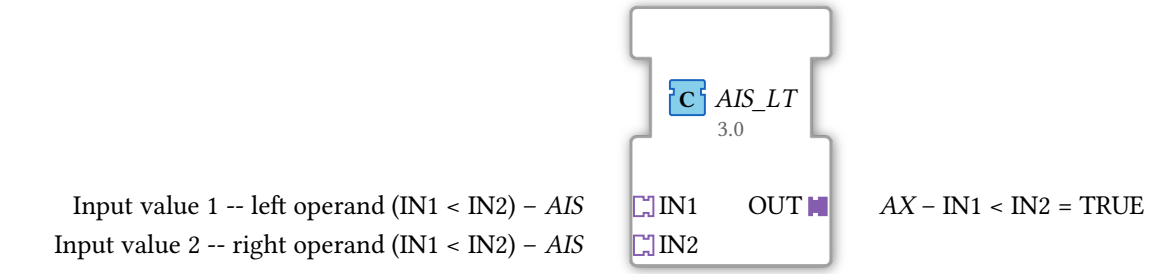
- **AIS_GE**: Prüft auf „größer gleich“ – invers zur Funktion von AIS_LE.
- **AIS_EQ**: Prüft auf Gleichheit der Werte.
- **AIS_LT**: Führt den Vergleich „kleiner als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version AIS_LE ist.

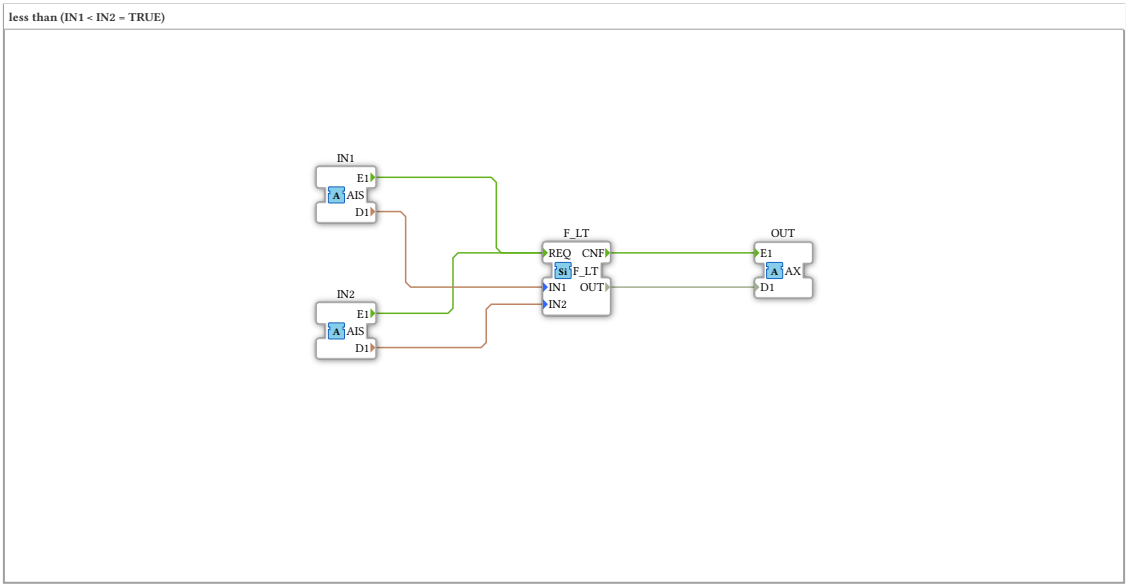
Der AIS_LE ergänzt die AIS-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

Der **AIS_LE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `STRING`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AIS_LT





• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AIS_LT** führt einen Vergleich zweier Werte vom Typ `STRING` (Zeichenkette) durch und prüft, ob der erste Wert (IN1) kleiner als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt lexikografisch (zeichenweise) gemäß der Zeichenkodierung.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `STRING`, Zeichenkette).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 < IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AIS` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AIS` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AIS` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LT` (Typ `iec61131::comparison::F_LT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AIS`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than“) für den Datentyp `STRING`.
- **Vergleichslogik**: Der Vergleich erfolgt lexikografisch (zeichenweise) gemäß der Zeichenkodierung.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Rezept- und Textkennungen gegen einen Referenz- oder Schwellwert vom Typ `STRING` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `STRING` die Bedingung „kleiner als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `STRING`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AIS_GT**: Prüft auf „größer als“ – invers zur Funktion von AIS_LT.
- **AIS_EQ**: Prüft auf Gleichheit der Werte.
- **AIS_LE**: Führt den Vergleich „kleiner gleich“ durch – die um den Gleichheitsfall erweiterte Version von AIS_LT.

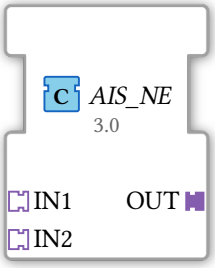
Der AIS_LT ist die grundlegende „kleiner als“-Komponente ohne Gleichheitsfall, ergänzt durch AIS_LE.

Fazit

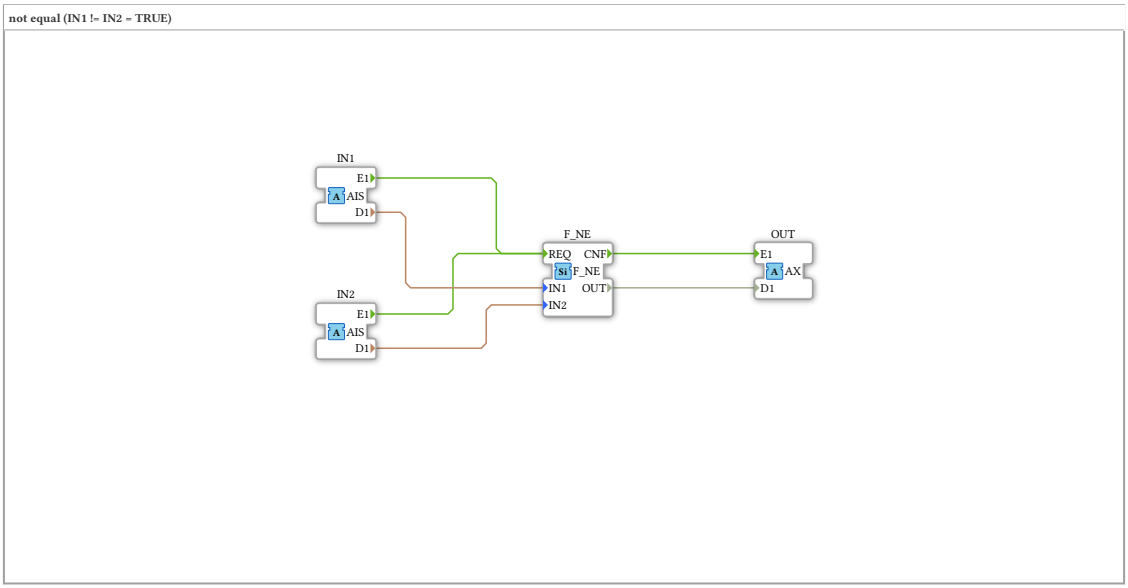
Der **AIS_LT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `STRING`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AIS_NE

Input value 1 -- left operand (IN1 != IN2) – *AIS*
Input value 2 -- right operand (IN1 != IN2) – *AIS*



AX – IN1 != IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AIS_NE** führt einen Vergleich zweier Werte vom Typ `STRING` (Zeichenkette) durch und prüft, ob der erste Wert (IN1) ungleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt lexikografisch (zeichenweise) gemäß der Zeichenkodierung.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `STRING`, Zeichenkette).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 != IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AIS` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AIS` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AIS` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_NE` (Typ `iec61131::comparison::F_NE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AIS`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („not equal to“) für den Datentyp `STRING`.
- **Vergleichslogik**: Der Vergleich erfolgt lexikografisch (zeichenweise) gemäß der Zeichenkodierung.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Rezept- und Textkennungen gegen einen Referenz- oder Schwellwert vom Typ `STRING` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `STRING` die Bedingung „ungleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `STRING`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

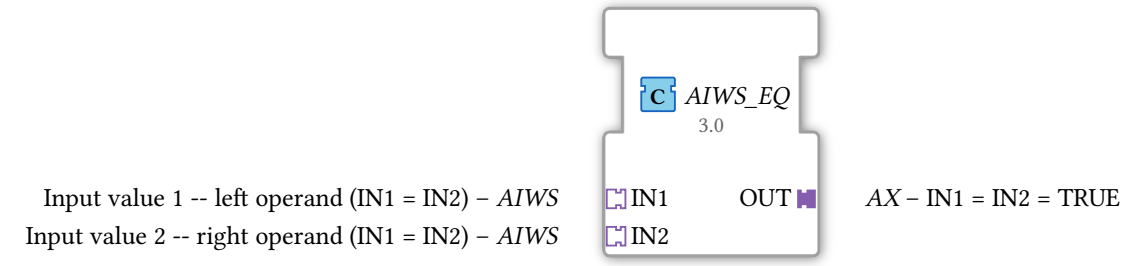
- **AIS_EQ**: Prüft auf „gleich“ – invers zur Funktion von AIS_NE.
- **AIS_EQ**: Prüft auf Gleichheit der Werte.

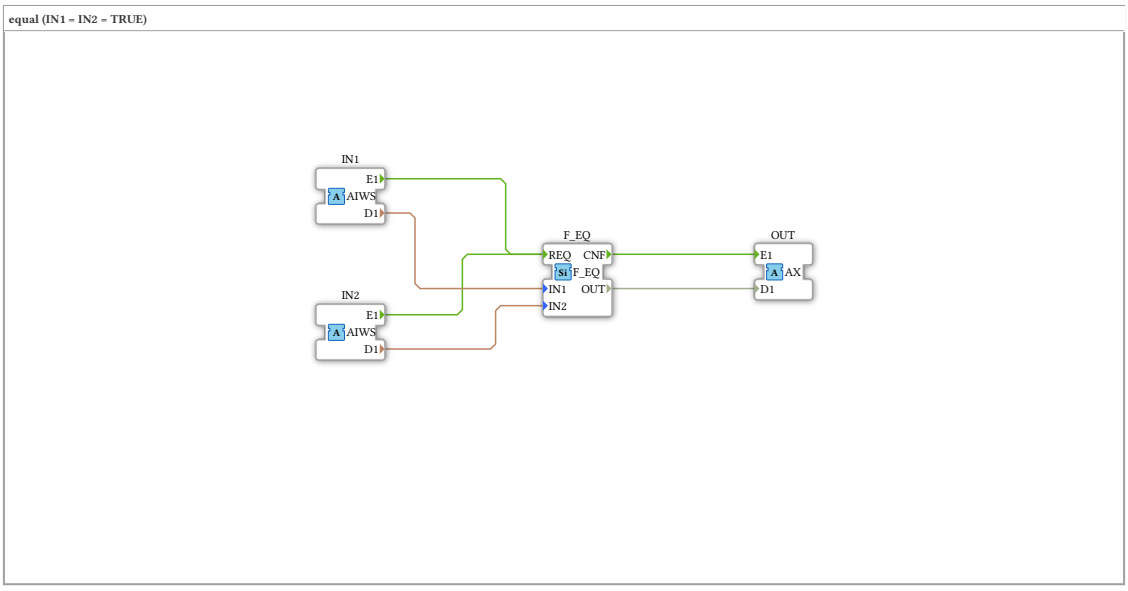
Der AIS_NE ist Teil der vollständigen AIS-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `STRING`.

Fazit

Der **AIS_NE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `STRING`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AIWS_EQ





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AIWS_EQ** führt einen Vergleich zweier Werte vom Typ `WSTRING` (Wide-Zeichenkette (Unicode)) durch und prüft, ob der erste Wert (IN1) gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt lexikografisch (zeichenweise) gemäß der Zeichenkodierung.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `WSTRING`, Wide-Zeichenkette (Unicode)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 = IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AIWS` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AIWS` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AIWS` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_EQ` (Typ `iec61131::comparison::F_EQ`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AIWS`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („equal to“) für den Datentyp `WSTRING`.
- **Vergleichslogik**: Der Vergleich erfolgt lexikografisch (zeichenweise) gemäß der Zeichenkodierung.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: mehrsprachige Textkennungen gegen einen Referenz- oder Schwellwert vom Typ `WSTRING` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `WSTRING` die Bedingung „gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `WSTRING`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

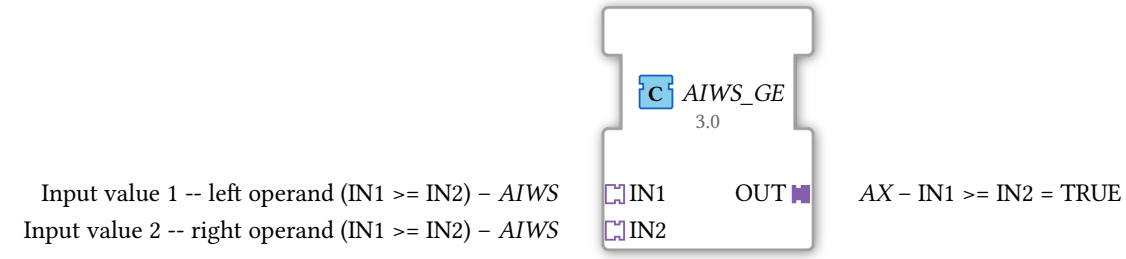
- **AIWS_NE**: Prüft auf „ungleich“ – invers zur Funktion von AIWS_EQ.
- **AIWS_NE**: Prüft auf Ungleichheit der Werte.

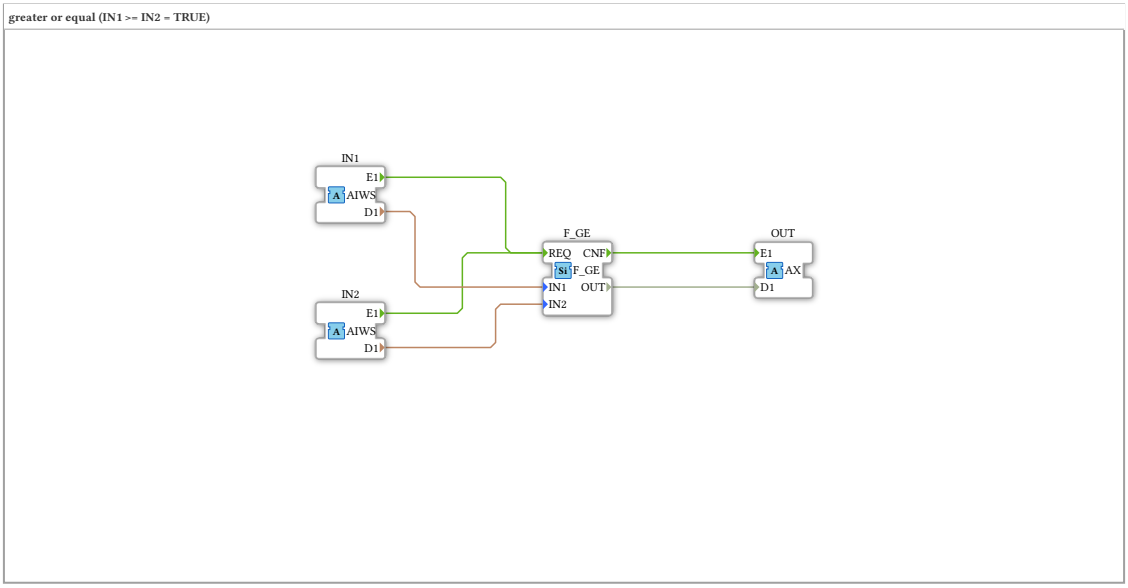
Der AIWS_EQ ist Teil der vollständigen AIWS-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `WSTRING`.

Fazit

Der **AIWS_EQ** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `WSTRING`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AIWS_GE





• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AIWS_GE** führt einen Vergleich zweier Werte vom Typ `WSTRING` (Wide-Zeichenkette (Unicode)) durch und prüft, ob der erste Wert (IN1) größer gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt lexikografisch (zeichenweise) gemäß der Zeichenkodierung.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `WSTRING`, Wide-Zeichenkette (Unicode)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 >= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AIWS` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AIWS` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AIWS` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GE` (Typ `iec61131::comparison::F_GE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AIWS`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than or equal to“) für den Datentyp `WSTRING`.
- **Vergleichslogik**: Der Vergleich erfolgt lexikografisch (zeichenweise) gemäß der Zeichenkodierung.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: mehrsprachige Textkennungen gegen einen Referenz- oder Schwellwert vom Typ `WSTRING` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `WSTRING` die Bedingung „größer gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `WSTRING`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AIWS_LE**: Prüft auf „kleiner gleich“ – invers zur Funktion von AIWS_GE.
- **AIWS_EQ**: Prüft auf Gleichheit der Werte.
- **AIWS_GT**: Führt den Vergleich „größer als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version AIWS_GE ist.

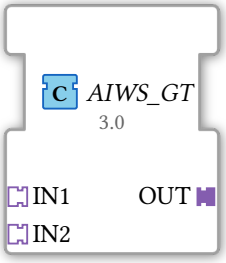
Der AIWS_GE ergänzt die AIWS-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

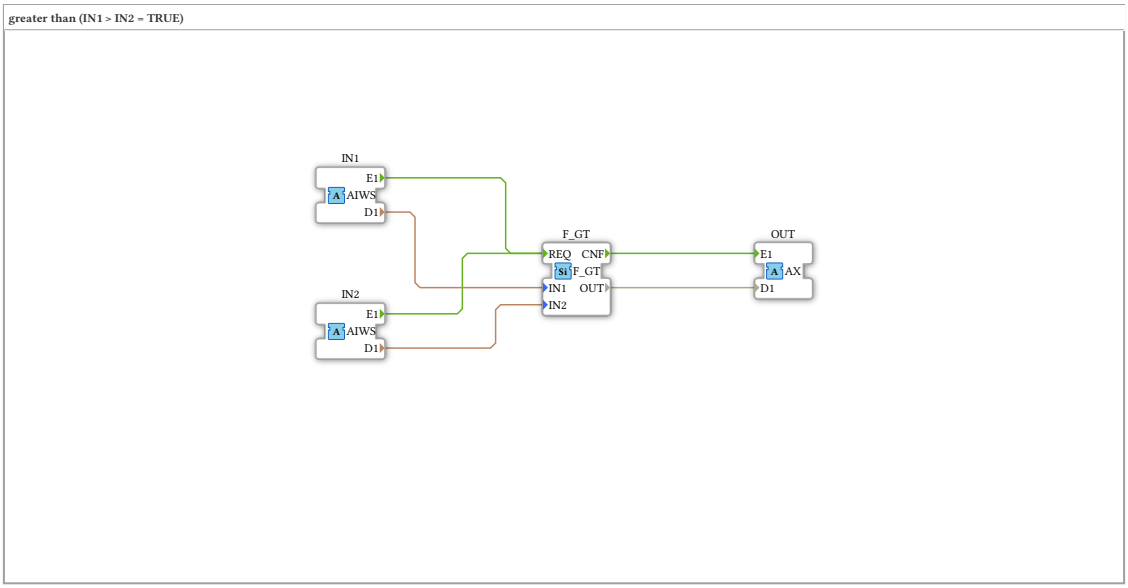
Der **AIWS_GE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `WSTRING`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AIWS_GT

Input value 1 -- left operand (IN1 > IN2) – *AIWS*
Input value 2 -- right operand (IN1 > IN2) – *AIWS*



AX – IN1 > IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AIWS_GT** führt einen Vergleich zweier Werte vom Typ `WSTRING` (Wide-Zeichenkette (Unicode)) durch und prüft, ob der erste Wert (IN1) größer als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt lexikografisch (zeichenweise) gemäß der Zeichenkodierung.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `WSTRING`, Wide-Zeichenkette (Unicode)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 > IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AIWS` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AIWS` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AIWS` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GT` (Typ `iec61131::comparison::F_GT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AIWS`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than“) für den Datentyp `WSTRING`.
- **Vergleichslogik**: Der Vergleich erfolgt lexikografisch (zeichenweise) gemäß der Zeichenkodierung.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: mehrsprachige Textkennungen gegen einen Referenz- oder Schwellwert vom Typ `WSTRING` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `WSTRING` die Bedingung „größer als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `WSTRING`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

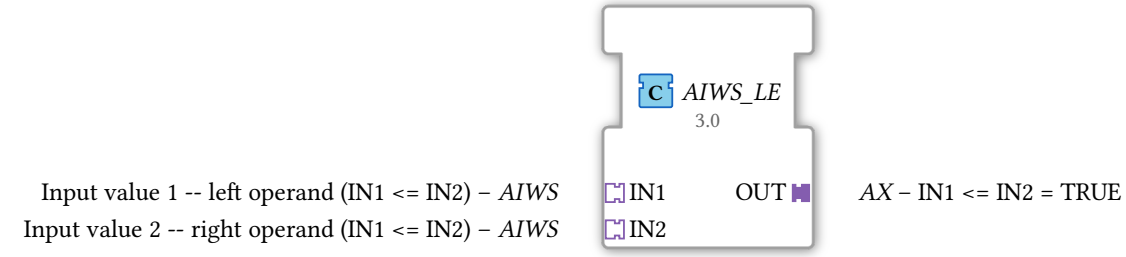
- **AIWS_LT**: Prüft auf „kleiner als“ – invers zur Funktion von AIWS_GT.
- **AIWS_EQ**: Prüft auf Gleichheit der Werte.
- **AIWS_GE**: Führt den Vergleich „größer gleich“ durch – die um den Gleichheitsfall erweiterte Version von AIWS_GT.

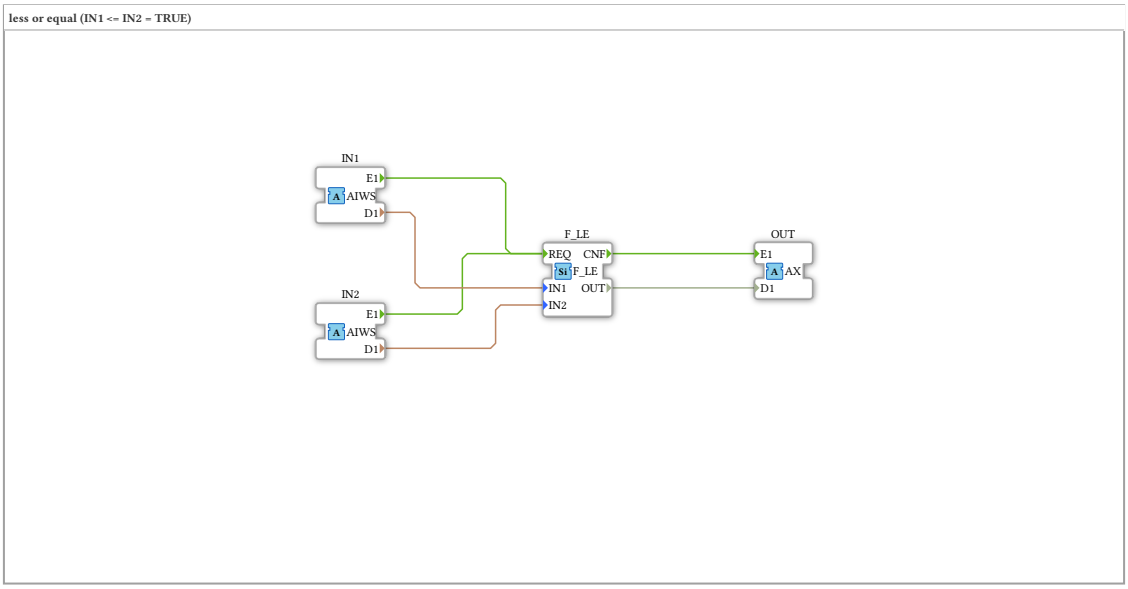
Der AIWS_GT ist die grundlegende „größer als“-Komponente ohne Gleichheitsfall, ergänzt durch AIWS_GE.

Fazit

Der **AIWS_GT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `WSTRING`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AIWS_LE





• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AIWS_LE** führt einen Vergleich zweier Werte vom Typ `WSTRING` (Wide-Zeichenkette (Unicode)) durch und prüft, ob der erste Wert (IN1) kleiner gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt lexikografisch (zeichenweise) gemäß der Zeichenkodierung.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `WSTRING`, Wide-Zeichenkette (Unicode)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 <= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AIWS` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AIWS` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AIWS` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LE` (Typ `iec61131::comparison::F_LE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AIWS`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than or equal to“) für den Datentyp `WSTRING`.
- **Vergleichslogik**: Der Vergleich erfolgt lexikografisch (zeichenweise) gemäß der Zeichenkodierung.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: mehrsprachige Textkennungen gegen einen Referenz- oder Schwellwert vom Typ `WSTRING` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `WSTRING` die Bedingung „kleiner gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `WSTRING`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AIWS_GE**: Prüft auf „größer gleich“ – invers zur Funktion von AIWS_LE.
- **AIWS_EQ**: Prüft auf Gleichheit der Werte.
- **AIWS_LT**: Führt den Vergleich „kleiner als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version AIWS_LE ist.

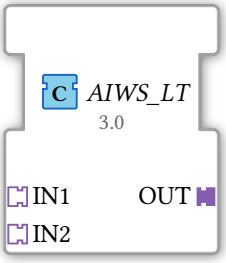
Der AIWS_LE ergänzt die AIWS-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

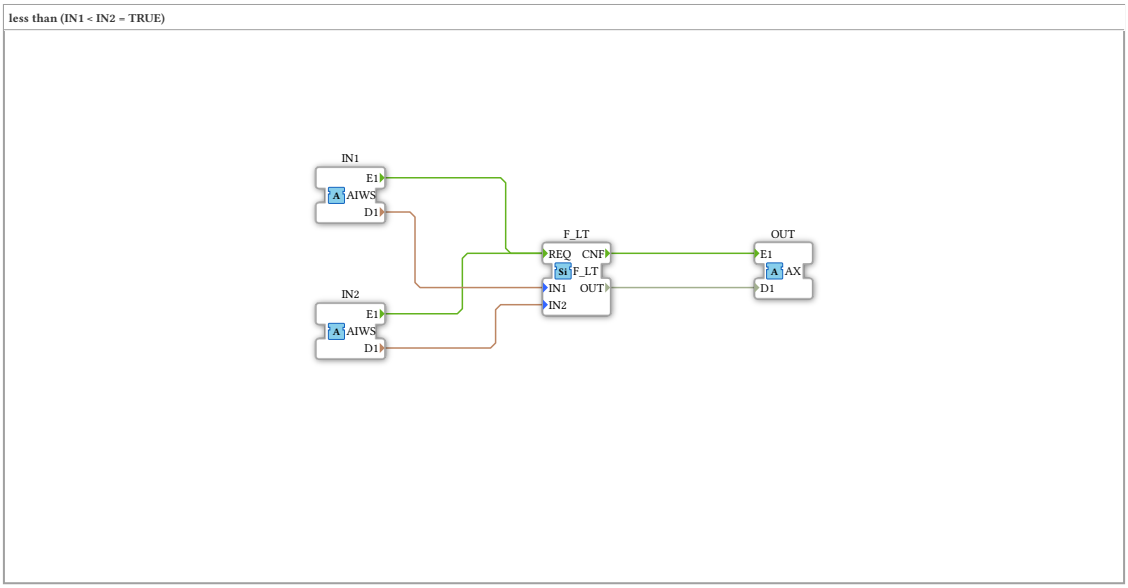
Der **AIWS_LE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `WSTRING`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AIWS_LT

Input value 1 -- left operand (IN1 < IN2) – *AIWS*
Input value 2 -- right operand (IN1 < IN2) – *AIWS*



AX – IN1 < IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AIWS_LT** führt einen Vergleich zweier Werte vom Typ `WSTRING` (Wide-Zeichenkette (Unicode)) durch und prüft, ob der erste Wert (IN1) kleiner als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt lexikografisch (zeichenweise) gemäß der Zeichenkodierung.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `WSTRING`, Wide-Zeichenkette (Unicode)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 < IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AIWS` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AIWS` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AIWS` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LT` (Typ `iec61131::comparison::F_LT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AIWS`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than“) für den Datentyp `WSTRING`.
- **Vergleichslogik**: Der Vergleich erfolgt lexikografisch (zeichenweise) gemäß der Zeichenkodierung.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: mehrsprachige Textkennungen gegen einen Referenz- oder Schwellwert vom Typ `WSTRING` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `WSTRING` die Bedingung „kleiner als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `WSTRING`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AIWS_GT**: Prüft auf „größer als“ – invers zur Funktion von AIWS_LT.
- **AIWS_EQ**: Prüft auf Gleichheit der Werte.
- **AIWS_LE**: Führt den Vergleich „kleiner gleich“ durch – die um den Gleichheitsfall erweiterte Version von AIWS_LT.

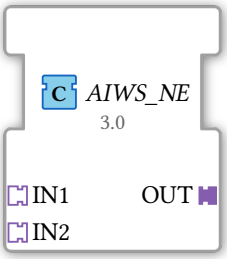
Der AIWS_LT ist die grundlegende „kleiner als“-Komponente ohne Gleichheitsfall, ergänzt durch AIWS_LE.

Fazit

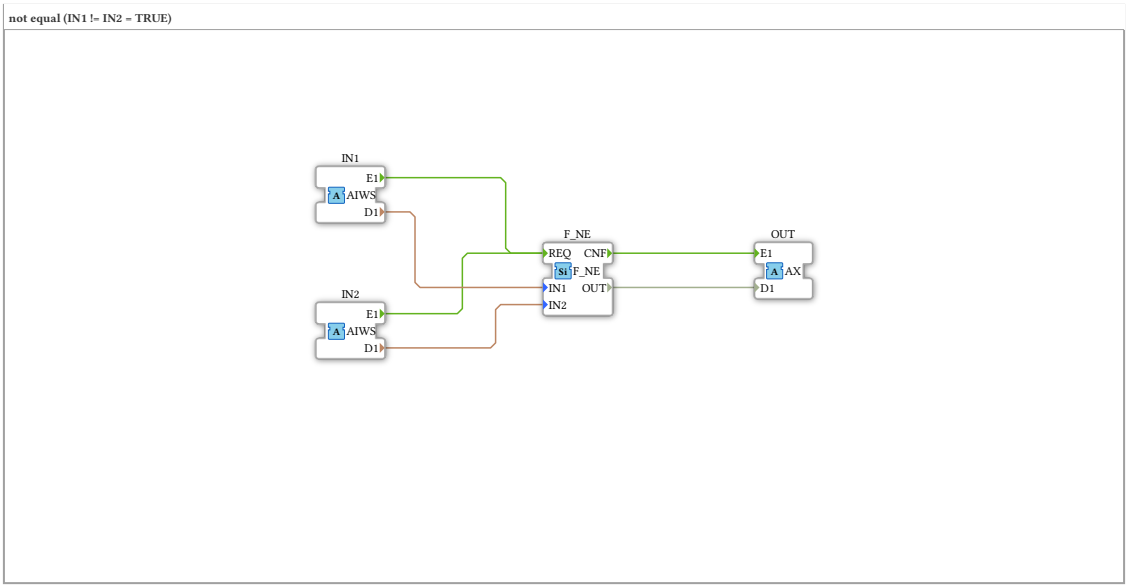
Der **AIWS_LT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `WSTRING`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AIWS_NE

Input value 1 -- left operand (IN1 != IN2) – *AIWS*
Input value 2 -- right operand (IN1 != IN2) – *AIWS*



AX – IN1 != IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AIWS_NE** führt einen Vergleich zweier Werte vom Typ `WSTRING` (Wide-Zeichenkette (Unicode)) durch und prüft, ob der erste Wert (IN1) ungleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt lexikografisch (zeichenweise) gemäß der Zeichenkodierung.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `WSTRING`, Wide-Zeichenkette (Unicode)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 != IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AIWS` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AIWS` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AIWS` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_NE` (Typ `iec61131::comparison::F_NE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AIWS`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („not equal to“) für den Datentyp `WSTRING`.
- **Vergleichslogik**: Der Vergleich erfolgt lexikografisch (zeichenweise) gemäß der Zeichenkodierung.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: mehrsprachige Textkennungen gegen einen Referenz- oder Schwellwert vom Typ `WSTRING` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `WSTRING` die Bedingung „ungleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `WSTRING`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AIWS_EQ**: Prüft auf „gleich“ – invers zur Funktion von AIWS_NE.
- **AIWS_EQ**: Prüft auf Gleichheit der Werte.

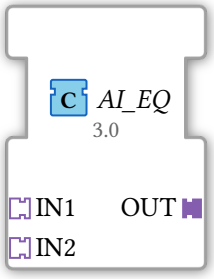
Der AIWS_NE ist Teil der vollständigen AIWS-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `WSTRING`.

Fazit

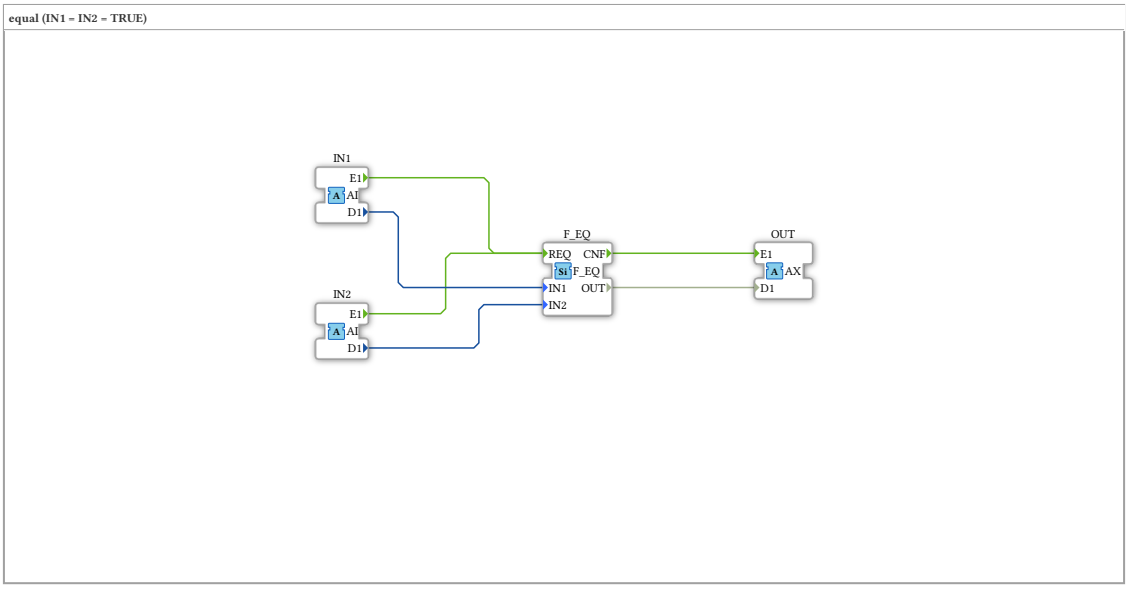
Der **AIWS_NE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `WSTRING`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AI_EQ

Input value 1 -- left operand (IN1 = IN2) – *AI*
Input value 2 -- right operand (IN1 = IN2) – *AI*



AX – IN1 = IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AI_EQ** führt einen Vergleich zweier Werte vom Typ `INT` (16-Bit-Ganzzahl (vorzeichenbehaftet)) durch und prüft, ob der erste Wert (IN1) gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `INT`, 16-Bit-Ganzzahl (vorzeichenbehaftet)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 = IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AI` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AI` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AI` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_EQ` (Typ `iec61131::comparison::F_EQ`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AI`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („equal to“) für den Datentyp `INT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Zählerstände gegen einen Referenz- oder Schwellwert vom Typ `INT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `INT` die Bedingung „gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `INT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AI_NE**: Prüft auf „ungleich“ – invers zur Funktion von AI_EQ.
- **AI_NE**: Prüft auf Ungleichheit der Werte.

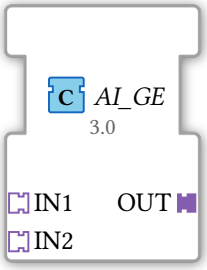
Der AI_EQ ist Teil der vollständigen AI-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `INT`.

Fazit

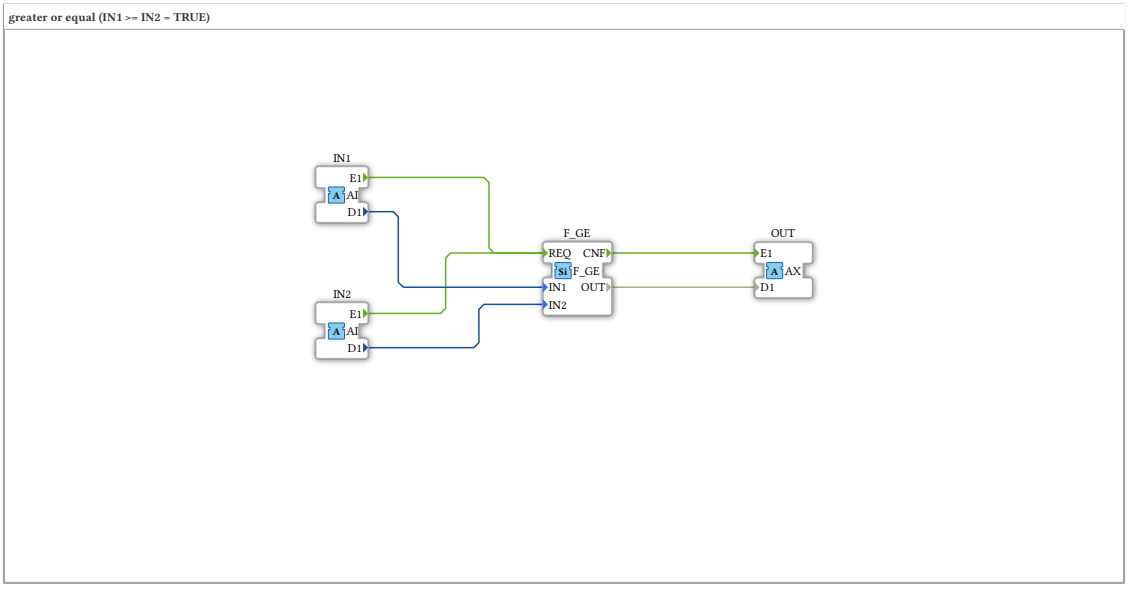
Der **AI_EQ** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `INT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AI_GE

Input value 1 -- left operand (IN1 >= IN2) – *AI*
Input value 2 -- right operand (IN1 >= IN2) – *AI*



AX – IN1 >= IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AI_GE** führt einen Vergleich zweier Werte vom Typ `INT` (16-Bit-Ganzzahl (vorzeichenbehaftet)) durch und prüft, ob der erste Wert (IN1) größer gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `INT`, 16-Bit-Ganzzahl (vorzeichenbehaftet)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 >= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AI` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AI` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AI` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GE` (Typ `iec61131::comparison::F_GE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AI`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than or equal to“) für den Datentyp `INT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Zählerstände gegen einen Referenz- oder Schwellwert vom Typ `INT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `INT` die Bedingung „größer gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `INT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AI_LE**: Prüft auf „kleiner gleich“ – invers zur Funktion von AI_GE.
- **AI_EQ**: Prüft auf Gleichheit der Werte.
- **AI_GT**: Führt den Vergleich „größer als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version AI_GE ist.

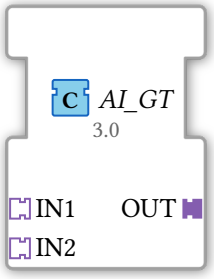
Der AI_GE ergänzt die AI-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

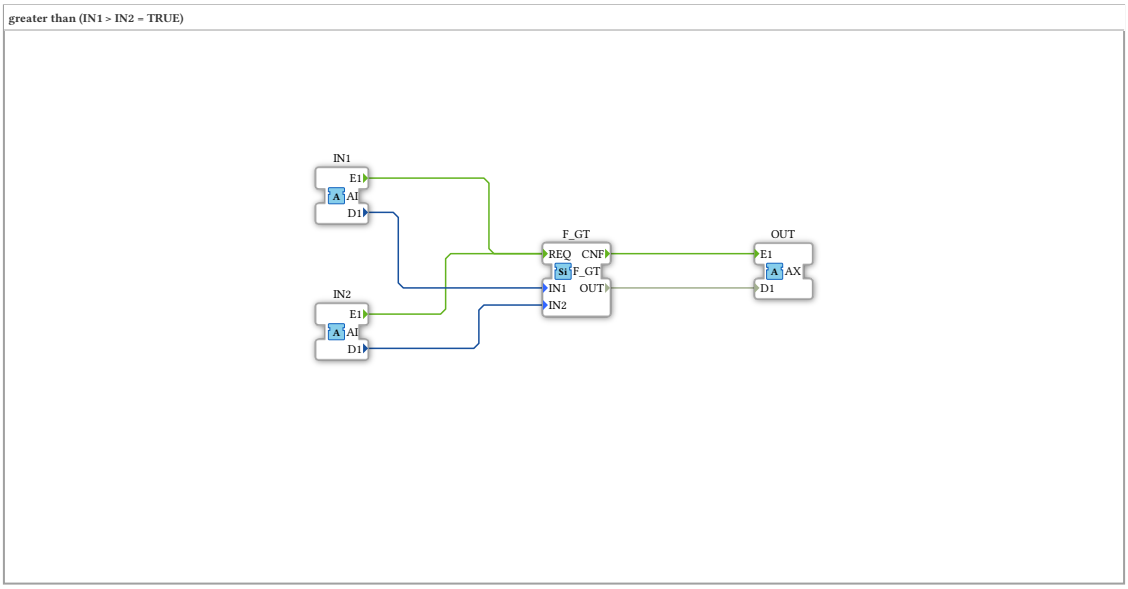
Der **AI_GE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `INT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AI_GT

Input value 1 -- left operand (IN1 > IN2) – *AI*
Input value 2 -- right operand (IN1 > IN2) – *AI*



AX – IN1 > IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AI_GT** führt einen Vergleich zweier Werte vom Typ `INT` (16-Bit-Ganzzahl (vorzeichenbehaftet)) durch und prüft, ob der erste Wert (IN1) größer als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `INT`, 16-Bit-Ganzzahl (vorzeichenbehaftet)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 > IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AI` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AI` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AI` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GT` (Typ `iec61131::comparison::F_GT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AI`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than“) für den Datentyp `INT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Zählerstände gegen einen Referenz- oder Schwellwert vom Typ `INT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `INT` die Bedingung „größer als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `INT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AI_LT**: Prüft auf „kleiner als“ – invers zur Funktion von AI_GT.
- **AI_EQ**: Prüft auf Gleichheit der Werte.
- **AI_GE**: Führt den Vergleich „größer gleich“ durch – die um den Gleichheitsfall erweiterte Version von AI_GT.

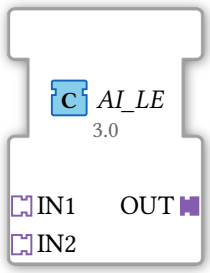
Der AI_GT ist die grundlegende „größer als“-Komponente ohne Gleichheitsfall, ergänzt durch AI_GE.

Fazit

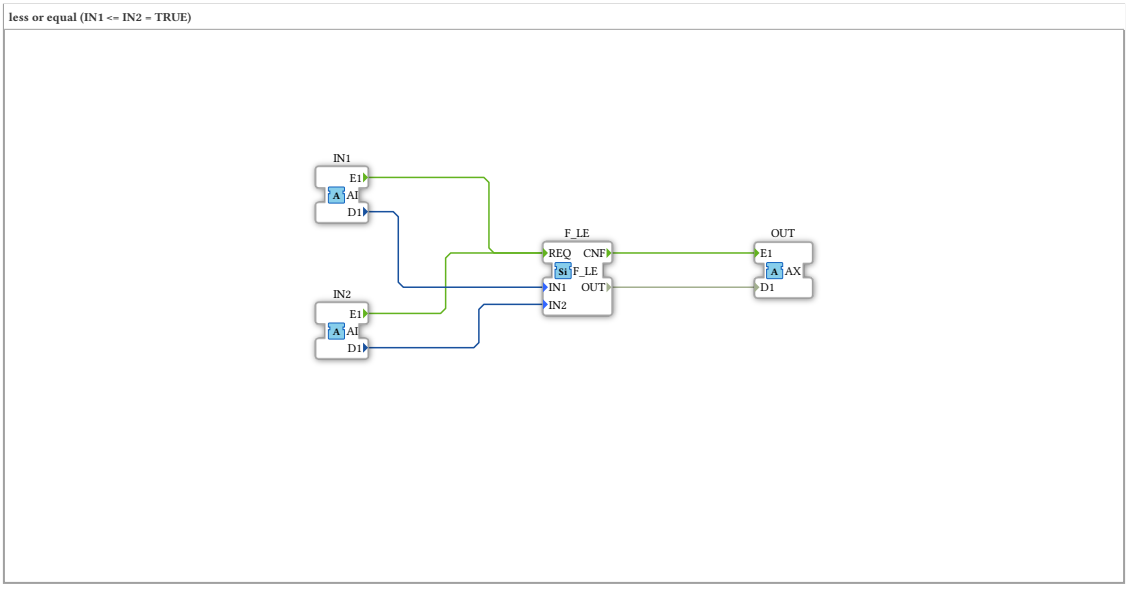
Der **AI_GT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `INT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AI_LE

Input value 1 -- left operand (IN1 <= IN2) – *AI*
Input value 2 -- right operand (IN1 <= IN2) – *AI*



AX – IN1 <= IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AI_LE** führt einen Vergleich zweier Werte vom Typ `INT` (16-Bit-Ganzzahl (vorzeichenbehaftet)) durch und prüft, ob der erste Wert (IN1) kleiner gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `INT`, 16-Bit-Ganzzahl (vorzeichenbehaftet)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 <= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AI` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AI` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AI` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LE` (Typ `iec61131::comparison::F_LE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AI`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than or equal to“) für den Datentyp `INT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Zählerstände gegen einen Referenz- oder Schwellwert vom Typ `INT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `INT` die Bedingung „kleiner gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `INT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AI_GE**: Prüft auf „größer gleich“ – invers zur Funktion von AI_LE.
- **AI_EQ**: Prüft auf Gleichheit der Werte.
- **AI_LT**: Führt den Vergleich „kleiner als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version AI_LE ist.

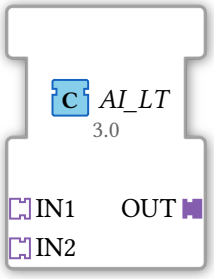
Der AI_LE ergänzt die AI-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

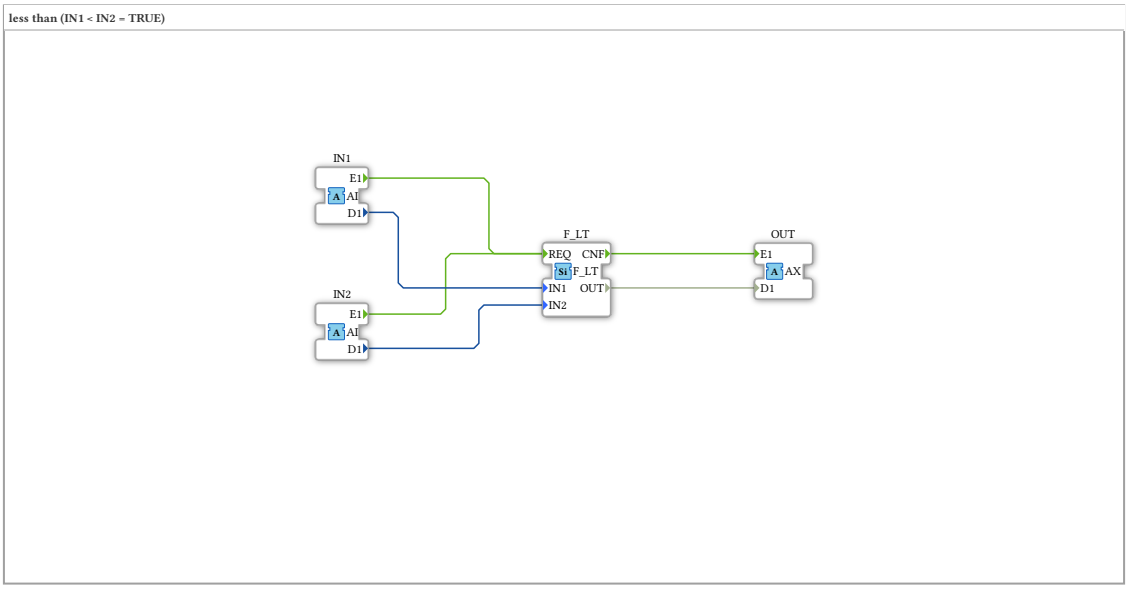
Der **AI_LE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `INT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AI_LT

Input value 1 -- left operand (IN1 < IN2) – *AI*
Input value 2 -- right operand (IN1 < IN2) – *AI*



AX – IN1 < IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AI_LT** führt einen Vergleich zweier Werte vom Typ `INT` (16-Bit-Ganzzahl (vorzeichenbehaftet)) durch und prüft, ob der erste Wert (IN1) kleiner als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `INT`, 16-Bit-Ganzzahl (vorzeichenbehaftet)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 < IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AI` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AI` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AI` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LT` (Typ `iec61131::comparison::F_LT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AI`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than“) für den Datentyp `INT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Zählerstände gegen einen Referenz- oder Schwellwert vom Typ `INT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `INT` die Bedingung „kleiner als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `INT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AI_GT**: Prüft auf „größer als“ – invers zur Funktion von AI_LT.
- **AI_EQ**: Prüft auf Gleichheit der Werte.
- **AI_LE**: Führt den Vergleich „kleiner gleich“ durch – die um den Gleichheitsfall erweiterte Version von AI_LT.

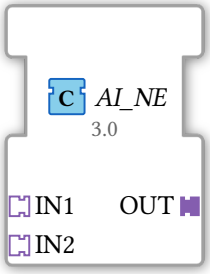
Der AI_LT ist die grundlegende „kleiner als“-Komponente ohne Gleichheitsfall, ergänzt durch AI_LE.

Fazit

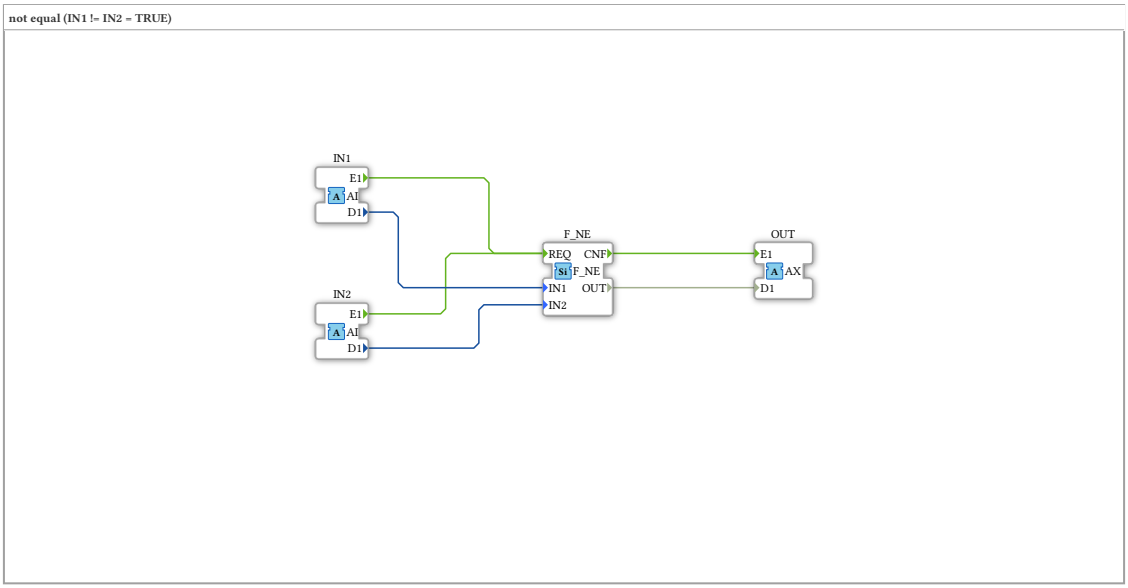
Der **AI_LT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `INT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AI_NE

Input value 1 -- left operand (IN1 != IN2) – *AI*
Input value 2 -- right operand (IN1 != IN2) – *AI*



AX – IN1 != IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AI_NE** führt einen Vergleich zweier Werte vom Typ `INT` (16-Bit-Ganzzahl (vorzeichenbehaftet)) durch und prüft, ob der erste Wert (IN1) ungleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `INT`, 16-Bit-Ganzzahl (vorzeichenbehaftet)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 != IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AI` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AI` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AI` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_NE` (Typ `iec61131::comparison::F_NE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AI`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („not equal to“) für den Datentyp `INT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Zählerstände gegen einen Referenz- oder Schwellwert vom Typ `INT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `INT` die Bedingung „ungleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `INT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AI_EQ**: Prüft auf „gleich“ – invers zur Funktion von AI_NE.
- **AI_EQ**: Prüft auf Gleichheit der Werte.

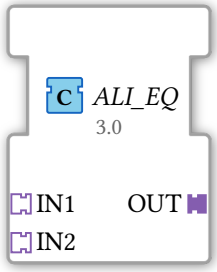
Der AI_NE ist Teil der vollständigen AI-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `INT`.

Fazit

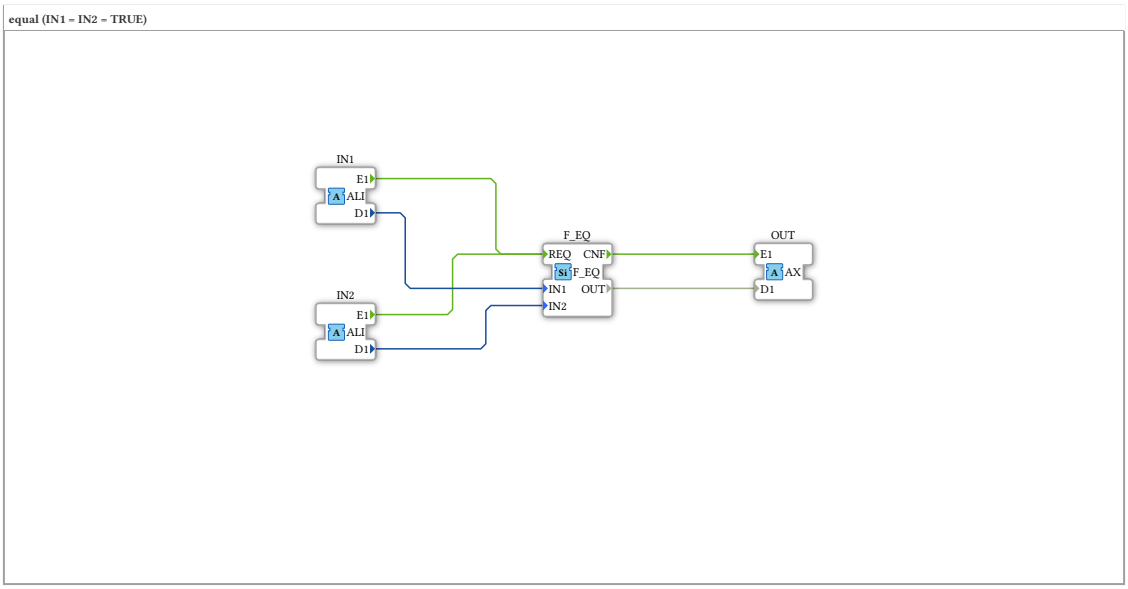
Der **AI_NE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `INT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

ALI_EQ

Input value 1 -- left operand (IN1 = IN2) – *ALI*
Input value 2 -- right operand (IN1 = IN2) – *ALI*



AX – IN1 = IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **ALI_EQ** führt einen Vergleich zweier Werte vom Typ `LINT` (64-Bit-Ganzzahl (vorzeichenbehaftet)) durch und prüft, ob der erste Wert (IN1) gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `LINT`, 64-Bit-Ganzzahl (vorzeichenbehaftet)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 = IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`ALI` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`ALI` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `ALI` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_EQ` (Typ `iec61131::comparison::F_EQ`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`ALI`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („equal to“) für den Datentyp `LINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: große Zähl- und Summenwerte gegen einen Referenz- oder Schwellwert vom Typ `LINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `LINT` die Bedingung „gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `LINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

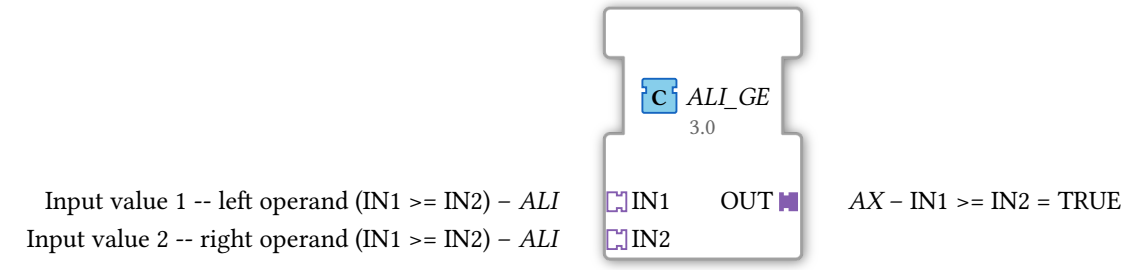
- **ALI_NE**: Prüft auf „ungleich“ – invers zur Funktion von ALI_EQ.
- **ALI_NE**: Prüft auf Ungleichheit der Werte.

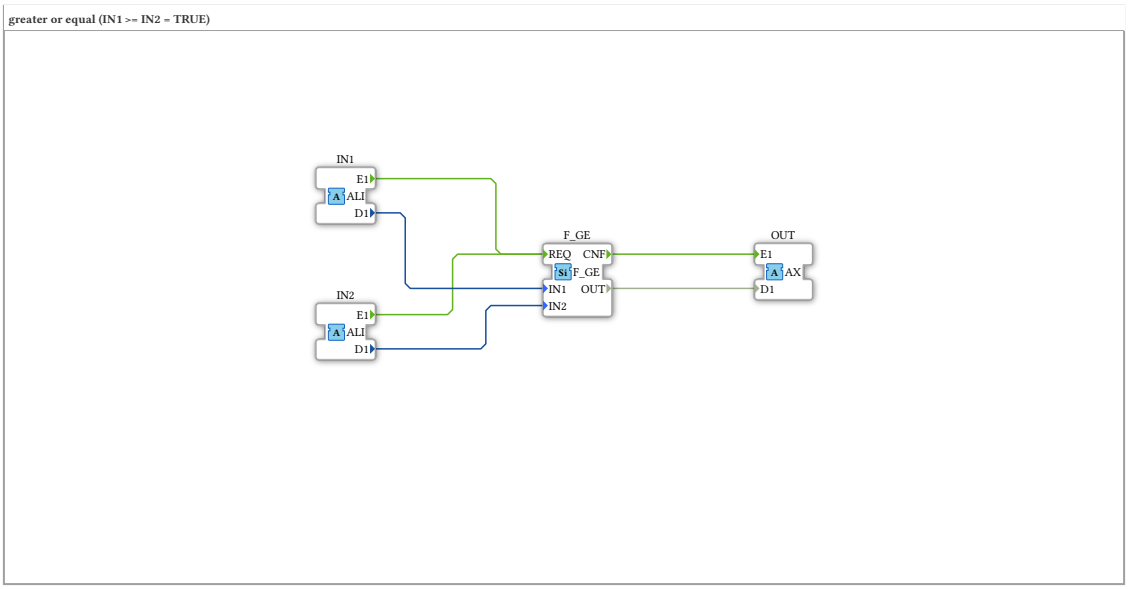
Der ALI_EQ ist Teil der vollständigen ALI-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `LINT`.

Fazit

Der **ALI_EQ** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `LINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

ALI_GE





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **ALI_GE** führt einen Vergleich zweier Werte vom Typ `LINT` (64-Bit-Ganzzahl (vorzeichenbehaftet)) durch und prüft, ob der erste Wert (IN1) größer gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `LINT`, 64-Bit-Ganzzahl (vorzeichenbehaftet)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 >= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`ALI` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`ALI` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `ALI` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GE` (Typ `iec61131::comparison::F_GE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`ALI`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than or equal to“) für den Datentyp `LINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: große Zähl- und Summenwerte gegen einen Referenz- oder Schwellwert vom Typ `LINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `LINT` die Bedingung „größer gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `LINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

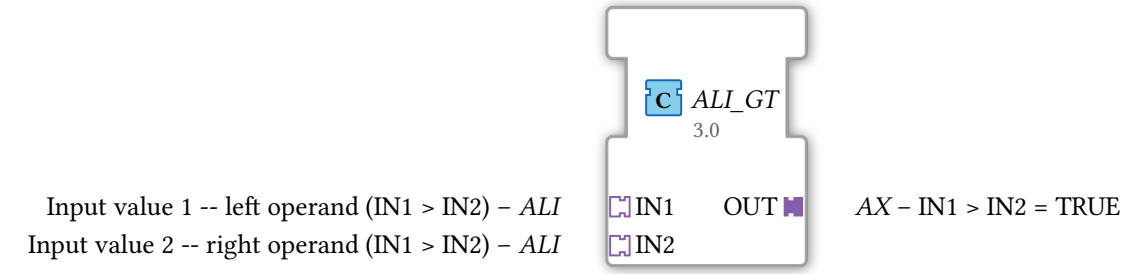
- **ALI_LE**: Prüft auf „kleiner gleich“ – invers zur Funktion von ALI_GE.
- **ALI_EQ**: Prüft auf Gleichheit der Werte.
- **ALI_GT**: Führt den Vergleich „größer als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version ALI_GE ist.

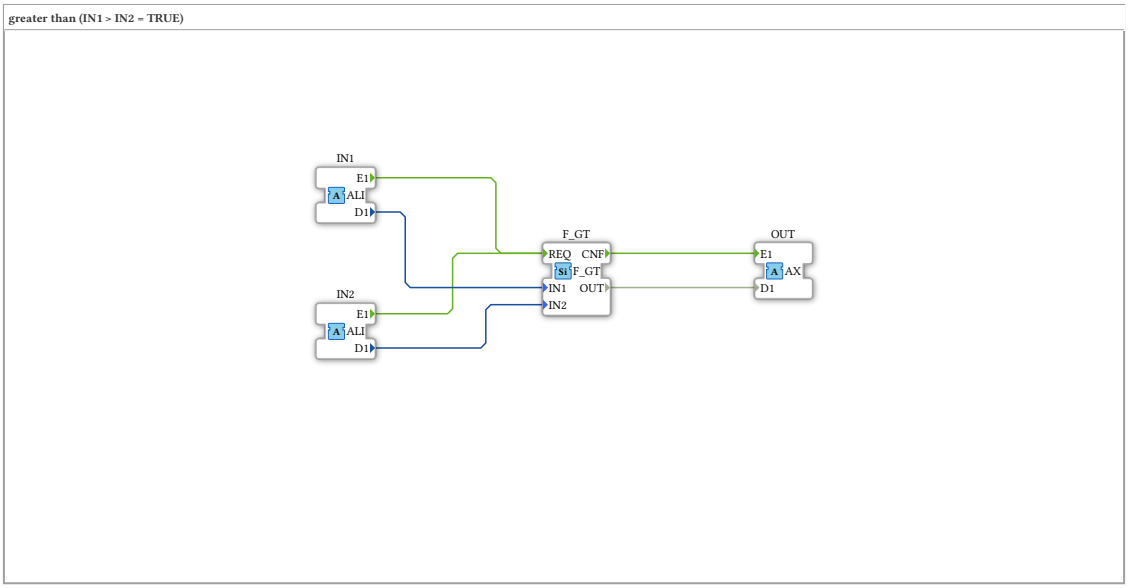
Der ALI_GE ergänzt die ALI-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

Der **ALI_GE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `LINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

ALI_GT





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **ALI_GT** führt einen Vergleich zweier Werte vom Typ `LINT` (64-Bit-Ganzzahl (vorzeichenbehaftet)) durch und prüft, ob der erste Wert (IN1) größer als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `LINT`, 64-Bit-Ganzzahl (vorzeichenbehaftet)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 > IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`ALI` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`ALI` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `ALI` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GT` (Typ `iec61131::comparison::F_GT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`ALI`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than“) für den Datentyp `LINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: große Zähl- und Summenwerte gegen einen Referenz- oder Schwellwert vom Typ `LINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `LINT` die Bedingung „größer als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `LINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

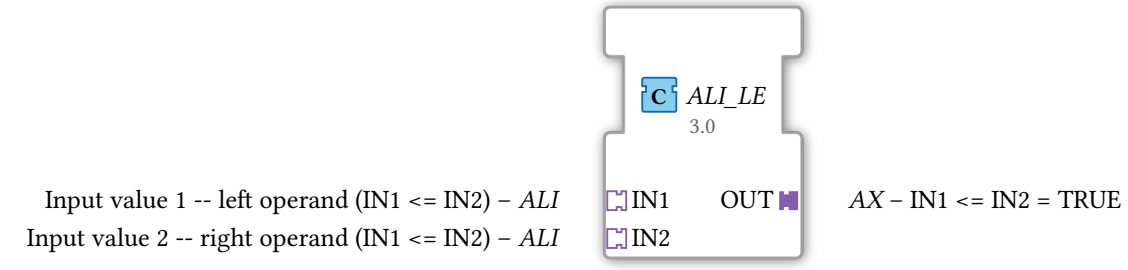
- **ALI_LT**: Prüft auf „kleiner als“ – invers zur Funktion von ALI_GT.
- **ALI_EQ**: Prüft auf Gleichheit der Werte.
- **ALI_GE**: Führt den Vergleich „größer gleich“ durch – die um den Gleichheitsfall erweiterte Version von ALI_GT.

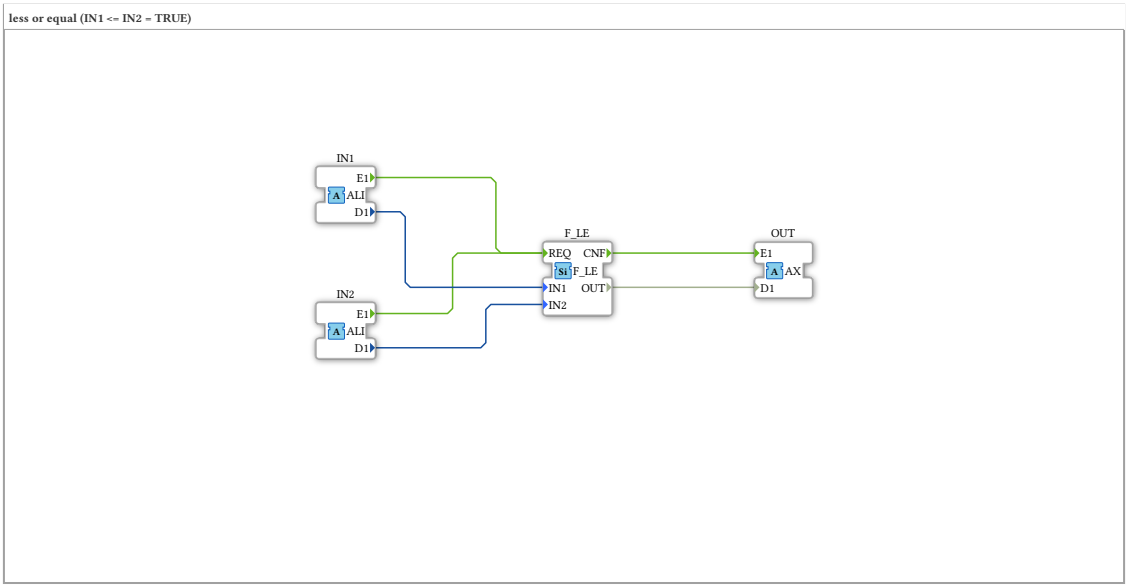
Der ALI_GT ist die grundlegende „größer als“-Komponente ohne Gleichheitsfall, ergänzt durch ALI_GE.

Fazit

Der **ALI_GT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `LINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

ALI_LE





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **ALI_LE** führt einen Vergleich zweier Werte vom Typ `LINT` (64-Bit-Ganzzahl (vorzeichenbehaftet)) durch und prüft, ob der erste Wert (IN1) kleiner gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `LINT`, 64-Bit-Ganzzahl (vorzeichenbehaftet)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 <= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`ALI` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`ALI` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `ALI` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LE` (Typ `iec61131::comparison::F_LE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`ALI`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than or equal to“) für den Datentyp `LINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: große Zähl- und Summenwerte gegen einen Referenz- oder Schwellwert vom Typ `LINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `LINT` die Bedingung „kleiner gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `LINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **ALI_GE**: Prüft auf „größer gleich“ – invers zur Funktion von ALI_LE.
- **ALI_EQ**: Prüft auf Gleichheit der Werte.
- **ALI_LT**: Führt den Vergleich „kleiner als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version ALI_LE ist.

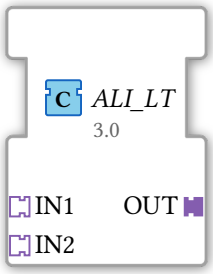
Der ALI_LE ergänzt die ALI-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

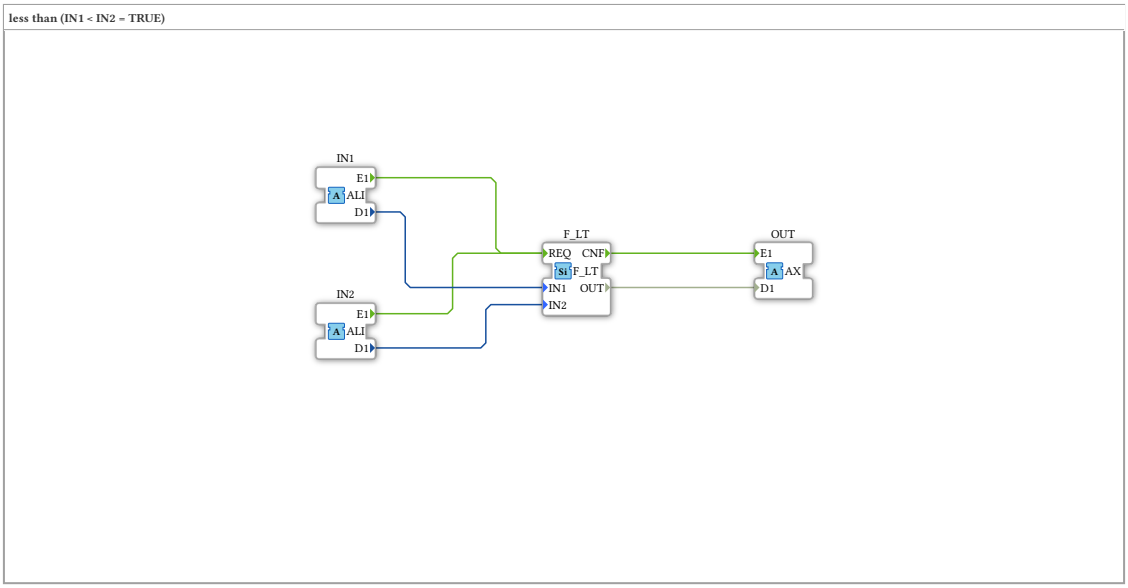
Der **ALI_LE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `LINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

ALI_LT

Input value 1 -- left operand (IN1 < IN2) – *ALI*
Input value 2 -- right operand (IN1 < IN2) – *ALI*



AX – IN1 < IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **ALI_LT** führt einen Vergleich zweier Werte vom Typ `LINT` (64-Bit-Ganzzahl (vorzeichenbehaftet)) durch und prüft, ob der erste Wert (IN1) kleiner als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `LINT`, 64-Bit-Ganzzahl (vorzeichenbehaftet)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 < IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`ALI` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`ALI` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `ALI` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LT` (Typ `iec61131::comparison::F_LT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`ALI`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than“) für den Datentyp `LINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: große Zähl- und Summenwerte gegen einen Referenz- oder Schwellwert vom Typ `LINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `LINT` die Bedingung „kleiner als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `LINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **ALI_GT**: Prüft auf „größer als“ – invers zur Funktion von ALI_LT.
- **ALI_EQ**: Prüft auf Gleichheit der Werte.
- **ALI_LE**: Führt den Vergleich „kleiner gleich“ durch – die um den Gleichheitsfall erweiterte Version von ALI_LT.

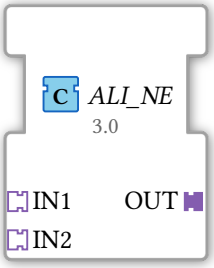
Der ALI_LT ist die grundlegende „kleiner als“-Komponente ohne Gleichheitsfall, ergänzt durch ALI_LE.

Fazit

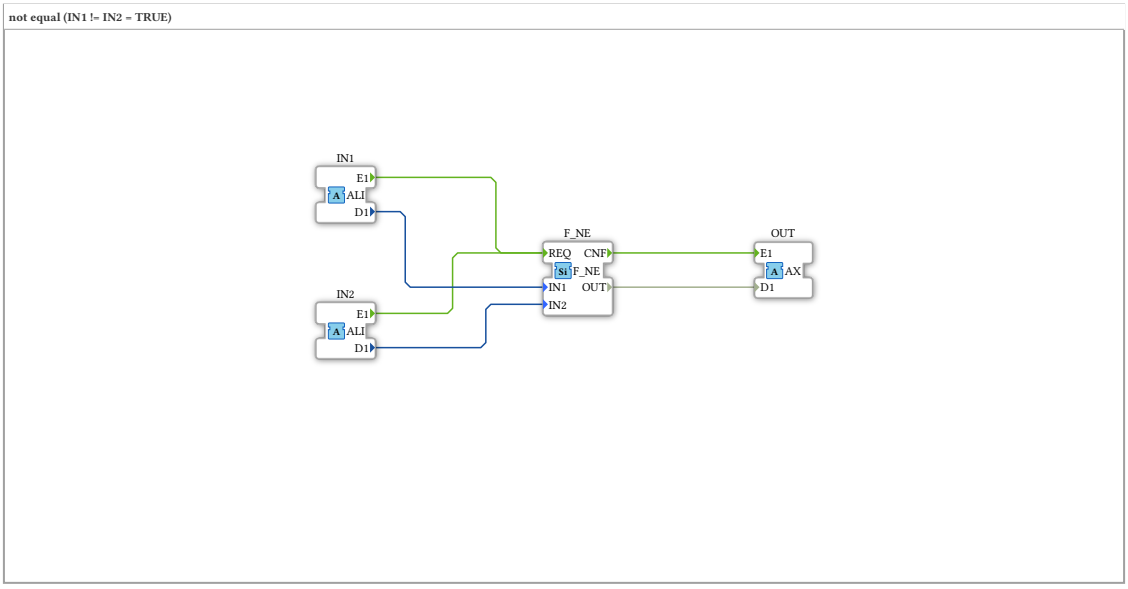
Der **ALI_LT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `LINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

ALI_NE

Input value 1 -- left operand (IN1 != IN2) – *ALI*
Input value 2 -- right operand (IN1 != IN2) – *ALI*



AX – IN1 != IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **ALI_NE** führt einen Vergleich zweier Werte vom Typ `LINT` (64-Bit-Ganzzahl (vorzeichenbehaftet)) durch und prüft, ob der erste Wert (IN1) ungleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `LINT`, 64-Bit-Ganzzahl (vorzeichenbehaftet)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 != IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`ALI` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`ALI` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `ALI` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_NE` (Typ `iec61131::comparison::F_NE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`ALI`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („not equal to“) für den Datentyp `LINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: große Zähl- und Summenwerte gegen einen Referenz- oder Schwellwert vom Typ `LINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `LINT` die Bedingung „ungleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `LINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **ALI_EQ**: Prüft auf „gleich“ – invers zur Funktion von ALI_NE.
- **ALI_EQ**: Prüft auf Gleichheit der Werte.

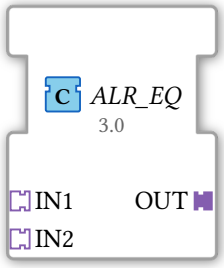
Der ALI_NE ist Teil der vollständigen ALI-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `LINT`.

Fazit

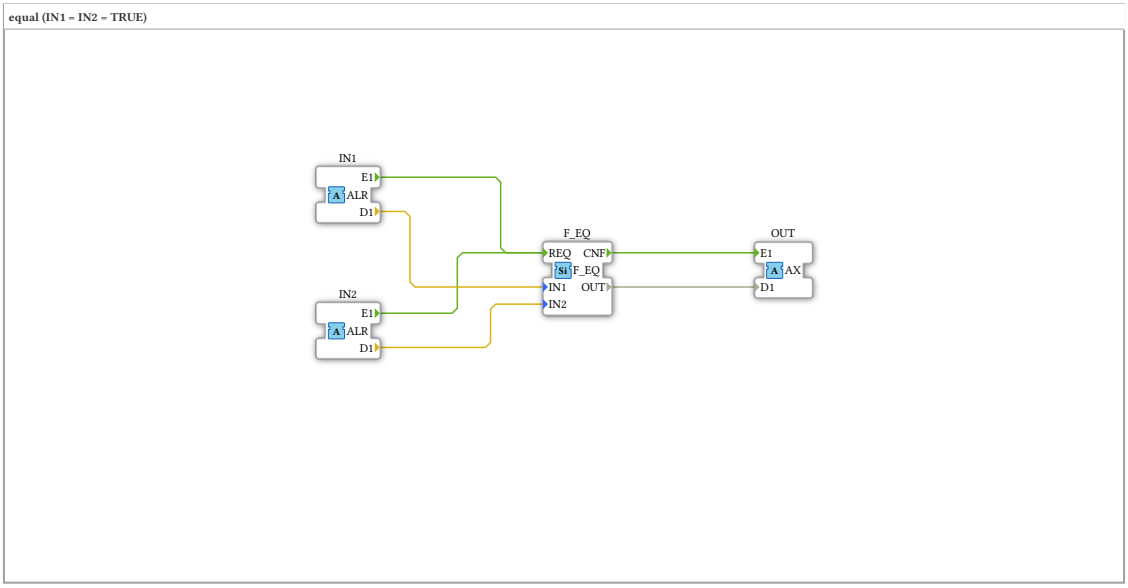
Der **ALI_NE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `LINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

ALR_EQ

Input value 1 -- left operand (IN1 = IN2) – *ALR*
Input value 2 -- right operand (IN1 = IN2) – *ALR*



AX – IN1 = IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **ALR_EQ** führt einen Vergleich zweier Werte vom Typ `LREAL` (64-Bit-Gleitkommazahl (doppelte Genauigkeit)) durch und prüft, ob der erste Wert (IN1) gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `LREAL`, 64-Bit-Gleitkommazahl (doppelte Genauigkeit)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 = IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`ALR` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`ALR` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `ALR` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_EQ` (Typ `iec61131::comparison::F_EQ`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`ALR`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („equal to“) für den Datentyp `LREAL`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: hochpräzise Messwerte gegen einen Referenz- oder Schwellwert vom Typ `LREAL` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `LREAL` die Bedingung „gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `LREAL`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **ALR_NE**: Prüft auf „ungleich“ – invers zur Funktion von ALR_EQ.
- **ALR_NE**: Prüft auf Ungleichheit der Werte.

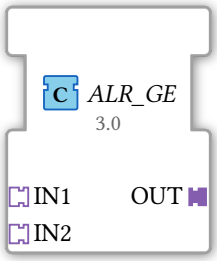
Der ALR_EQ ist Teil der vollständigen ALR-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `LREAL`.

Fazit

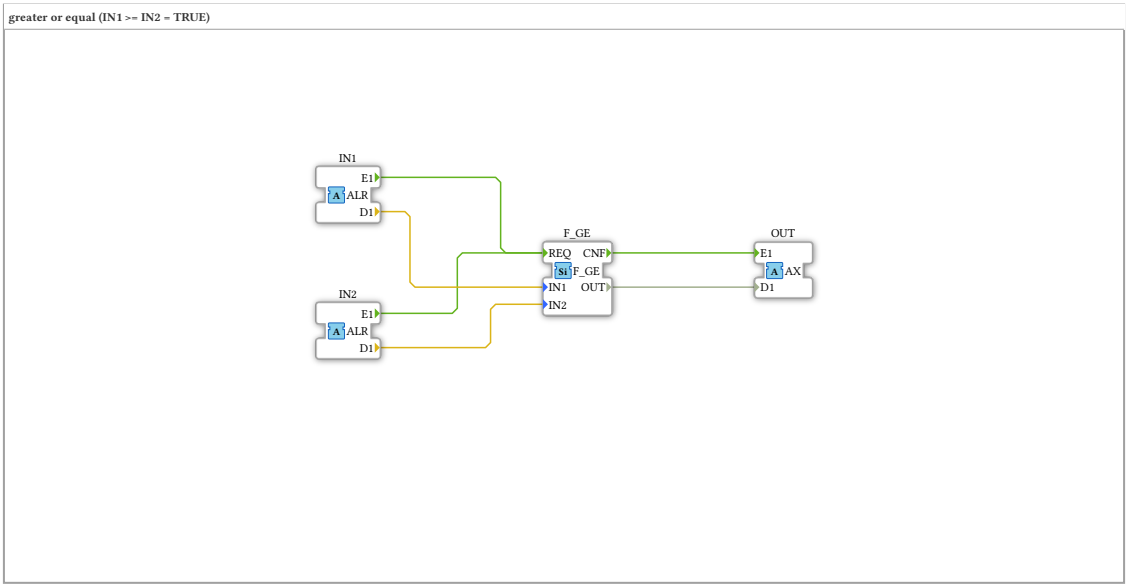
Der **ALR_EQ** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `LREAL`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

ALR_GE

Input value 1 -- left operand (IN1 >= IN2) – *ALR*
Input value 2 -- right operand (IN1 >= IN2) – *ALR*



AX – IN1 >= IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **ALR_GE** führt einen Vergleich zweier Werte vom Typ `LREAL` (64-Bit-Gleitkommazahl (doppelte Genauigkeit)) durch und prüft, ob der erste Wert (IN1) größer gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `LREAL`, 64-Bit-Gleitkommazahl (doppelte Genauigkeit)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 >= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`ALR` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`ALR` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `ALR` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GE` (Typ `iec61131::comparison::F_GE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`ALR`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than or equal to“) für den Datentyp `LREAL`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: hochpräzise Messwerte gegen einen Referenz- oder Schwellwert vom Typ `LREAL` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `LREAL` die Bedingung „größer gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `LREAL`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **ALR_LE**: Prüft auf „kleiner gleich“ – invers zur Funktion von ALR_GE.
- **ALR_EQ**: Prüft auf Gleichheit der Werte.
- **ALR_GT**: Führt den Vergleich „größer als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version ALR_GE ist.

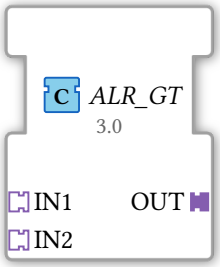
Der ALR_GE ergänzt die ALR-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

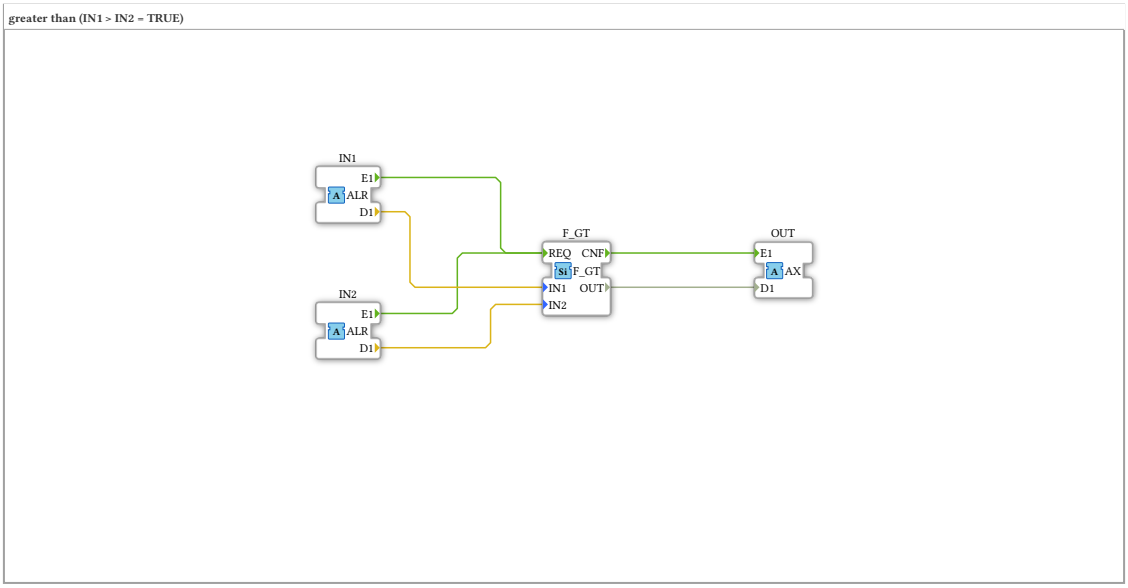
Der **ALR_GE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `LREAL`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

ALR_GT

Input value 1 -- left operand (IN1 > IN2) – *ALR*
Input value 2 -- right operand (IN1 > IN2) – *ALR*



AX – IN1 > IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **ALR_GT** führt einen Vergleich zweier Werte vom Typ `LREAL` (64-Bit-Gleitkommazahl (doppelte Genauigkeit)) durch und prüft, ob der erste Wert (IN1) größer als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.

- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `LREAL`, 64-Bit-Gleitkommazahl (doppelte Genauigkeit)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 > IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`ALR` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`ALR` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `ALR` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GT` (Typ `iec61131::comparison::F_GT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`ALR`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than“) für den Datentyp `LREAL`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: hochpräzise Messwerte gegen einen Referenz- oder Schwellwert vom Typ `LREAL` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `LREAL` die Bedingung „größer als“ erfüllen.

- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf LREAL-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **ALR_LT**: Prüft auf „kleiner als“ – invers zur Funktion von ALR_GT.
- **ALR_EQ**: Prüft auf Gleichheit der Werte.
- **ALR_GE**: Führt den Vergleich „größer gleich“ durch – die um den Gleichheitsfall erweiterte Version von ALR_GT.

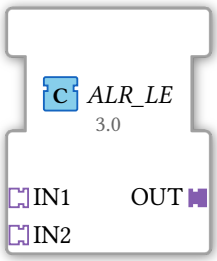
Der ALR_GT ist die grundlegende „größer als“-Komponente ohne Gleichheitsfall, ergänzt durch ALR_GE.

Fazit

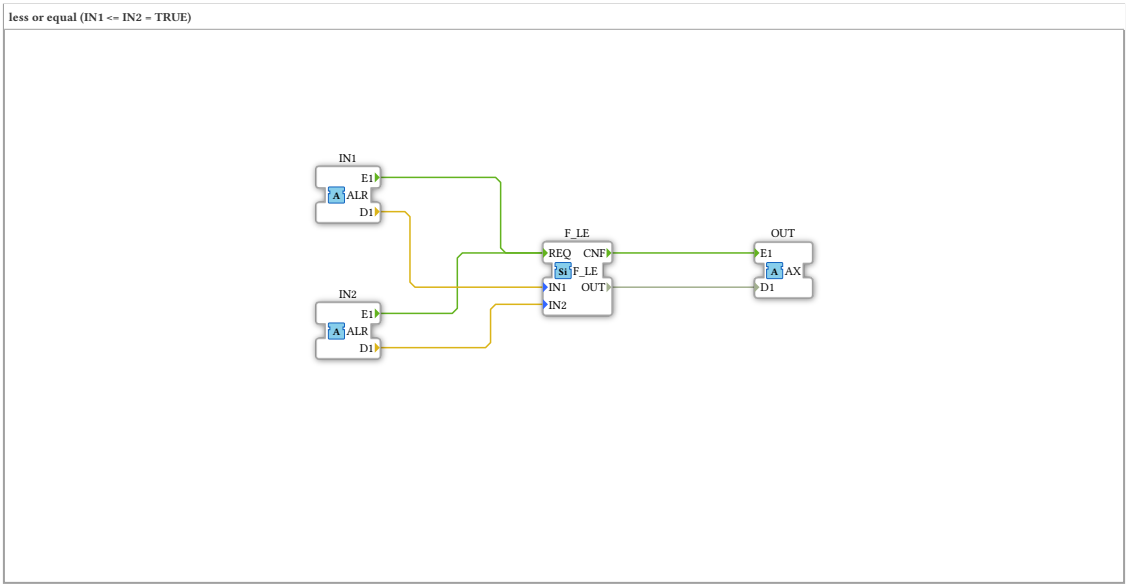
Der **ALR_GT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp LREAL. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

ALR_LE

Input value 1 -- left operand (IN1 <= IN2) – *ALR*
Input value 2 -- right operand (IN1 <= IN2) – *ALR*



AX – IN1 <= IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **ALR_LE** führt einen Vergleich zweier Werte vom Typ LREAL (64-Bit-Gleitkommazahl (doppelte Genauigkeit)) durch und prüft, ob der erste Wert (IN1) kleiner gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `LREAL`, 64-Bit-Gleitkommazahl (doppelte Genauigkeit)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 <= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`ALR` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`ALR` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `ALR` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LE` (Typ `iec61131::comparison::F_LE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`ALR`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than or equal to“) für den Datentyp `LREAL`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: hochpräzise Messwerte gegen einen Referenz- oder Schwellwert vom Typ `LREAL` prüfen.

- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `LREAL` die Bedingung „kleiner gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `LREAL`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

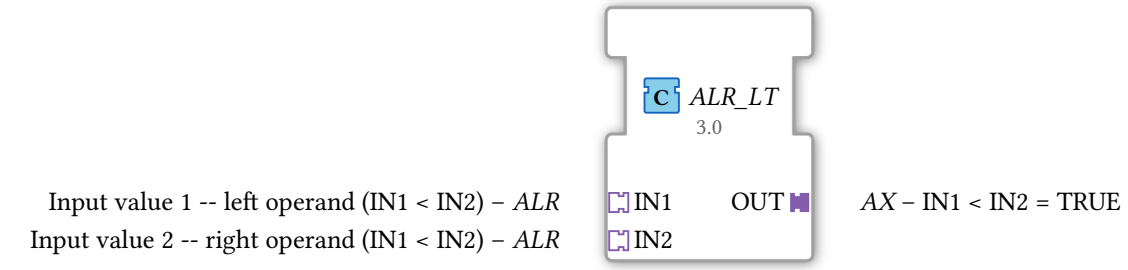
- **ALR_GE**: Prüft auf „größer gleich“ – invers zur Funktion von ALR_LE.
- **ALR_EQ**: Prüft auf Gleichheit der Werte.
- **ALR_LT**: Führt den Vergleich „kleiner als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version ALR_LE ist.

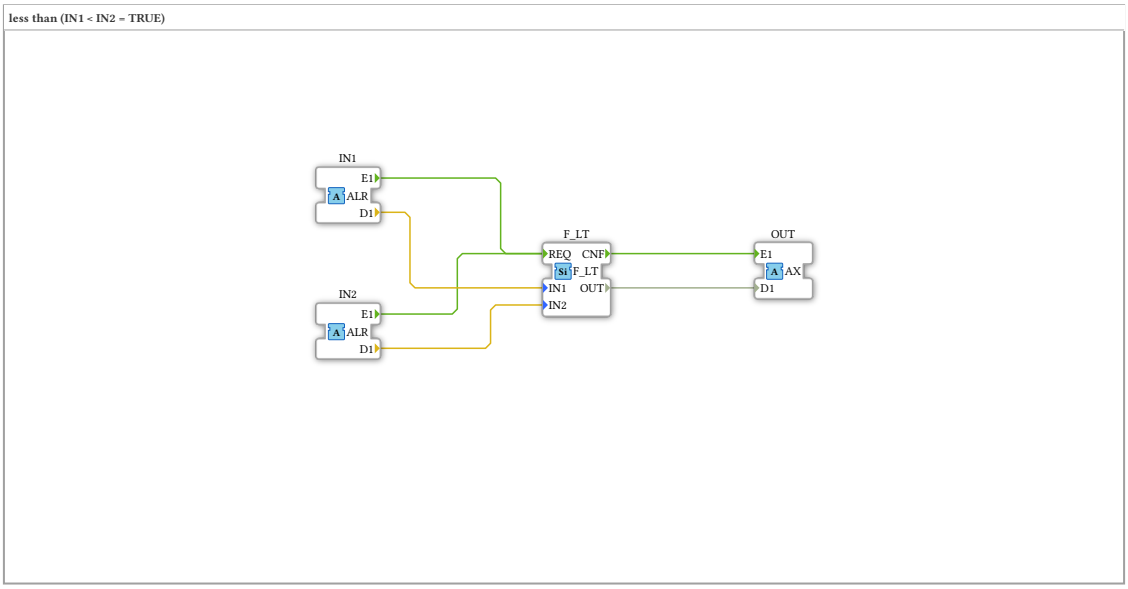
Der ALR_LE ergänzt die ALR-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

Der **ALR_LE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `LREAL`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

ALR_LT





• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **ALR_LT** führt einen Vergleich zweier Werte vom Typ `LREAL` (64-Bit-Gleitkommazahl (doppelte Genauigkeit)) durch und prüft, ob der erste Wert (IN1) kleiner als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `LREAL`, 64-Bit-Gleitkommazahl (doppelte Genauigkeit)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 < IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`ALR` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`ALR` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `ALR` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LT` (Typ `iec61131::comparison::F_LT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`ALR`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than“) für den Datentyp `LREAL`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: hochpräzise Messwerte gegen einen Referenz- oder Schwellwert vom Typ `LREAL` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `LREAL` die Bedingung „kleiner als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `LREAL`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **ALR_GT**: Prüft auf „größer als“ – invers zur Funktion von ALR_LT.
- **ALR_EQ**: Prüft auf Gleichheit der Werte.
- **ALR_LE**: Führt den Vergleich „kleiner gleich“ durch – die um den Gleichheitsfall erweiterte Version von ALR_LT.

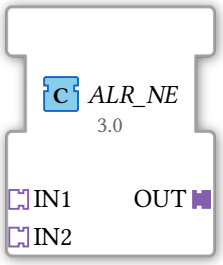
Der ALR_LT ist die grundlegende „kleiner als“-Komponente ohne Gleichheitsfall, ergänzt durch ALR_LE.

Fazit

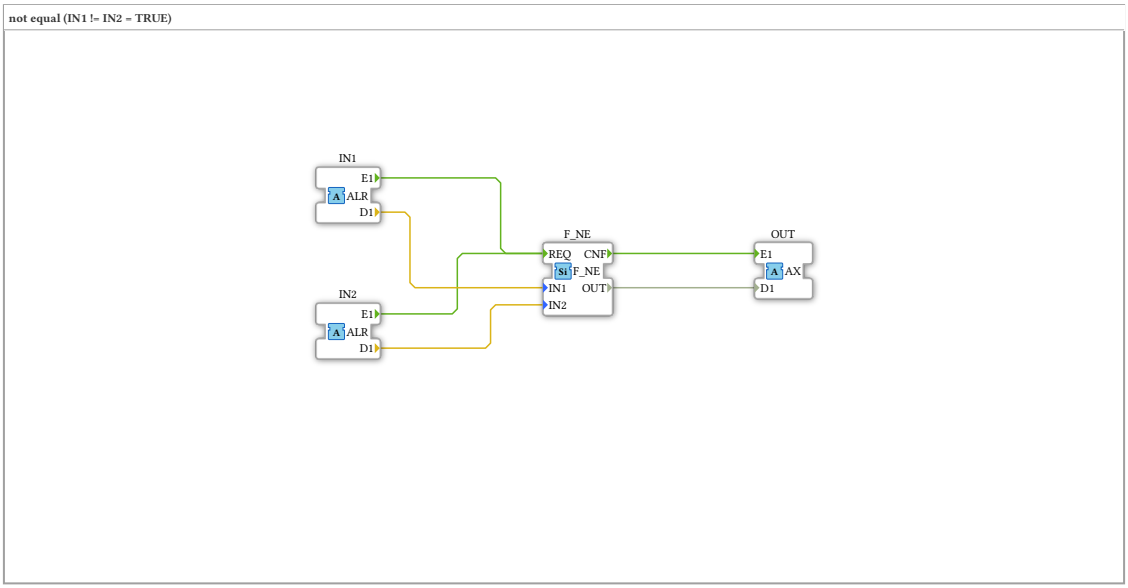
Der **ALR_LT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `LREAL`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

ALR_NE

Input value 1 -- left operand (IN1 != IN2) – *ALR*
Input value 2 -- right operand (IN1 != IN2) – *ALR*



AX – IN1 != IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **ALR_NE** führt einen Vergleich zweier Werte vom Typ `LREAL` (64-Bit-Gleitkommazahl (doppelte Genauigkeit)) durch und prüft, ob der erste Wert (IN1) ungleich dem zweiten Wert (IN2) ist. Das

Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `LREAL`, 64-Bit-Gleitkommazahl (doppelte Genauigkeit)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 != IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`ALR` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`ALR` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `ALR` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_NE` (Typ `iec61131::comparison::F_NE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`ALR`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („not equal to“) für den Datentyp `LREAL`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: hochpräzise Messwerte gegen einen Referenz- oder Schwellwert vom Typ `LREAL` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `LREAL` die Bedingung „ungleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `LREAL`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **ALR_EQ**: Prüft auf „gleich“ – invers zur Funktion von ALR_NE.
- **ALR_EQ**: Prüft auf Gleichheit der Werte.

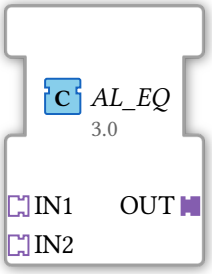
Der ALR_NE ist Teil der vollständigen ALR-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `LREAL`.

Fazit

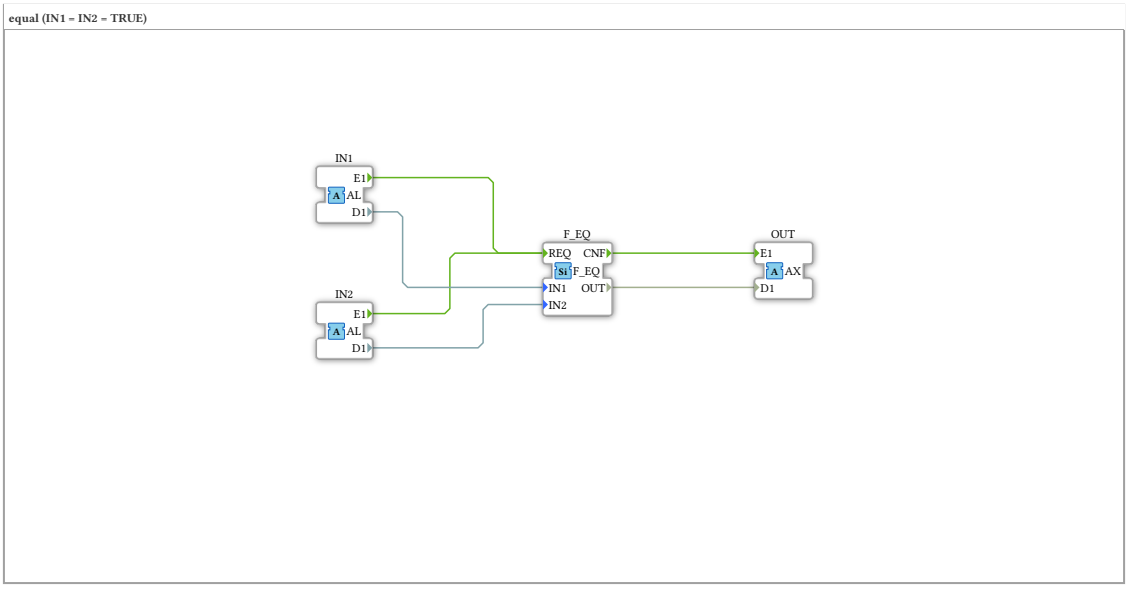
Der **ALR_NE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `LREAL`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AL_EQ

Input value 1 -- left operand (IN1 = IN2) – *AL*
Input value 2 -- right operand (IN1 = IN2) – *AL*



AX – IN1 = IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AL_EQ** führt einen Vergleich zweier Werte vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)) durch und prüft, ob der erste Wert (IN1) gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `LWORD`, 64-Bit-Bitmuster (Langwort)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 = IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AL` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AL` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AL` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_EQ` (Typ `iec61131::comparison::F_EQ`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AL`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („equal to“) für den Datentyp `LWORD`.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: erweiterte Bitmasken gegen einen Referenz- oder Schwellwert vom Typ `LWORD` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `LWORD` die Bedingung „gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `LWORD`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AL_NE**: Prüft auf „ungleich“ – invers zur Funktion von AL_EQ.
- **AL_NE**: Prüft auf Ungleichheit der Werte.

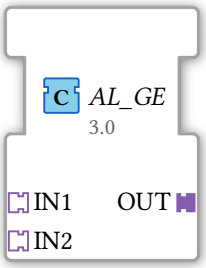
Der AL_EQ ist Teil der vollständigen AL-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `LWORD`.

Fazit

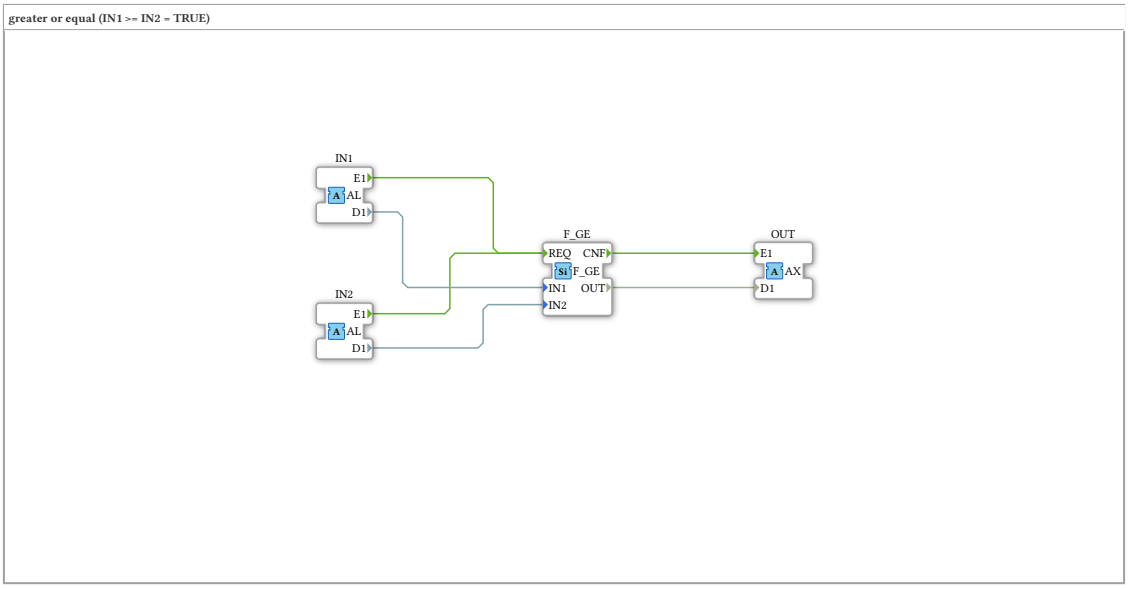
Der **AL_EQ** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `LWORD`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AL_GE

Input value 1 -- left operand (IN1 >= IN2) – *AL*
Input value 2 -- right operand (IN1 >= IN2) – *AL*



AX – IN1 >= IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AL_GE** führt einen Vergleich zweier Werte vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)) durch und prüft, ob der erste Wert (IN1) größer gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `LWORD`, 64-Bit-Bitmuster (Langwort)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 >= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AL` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AL` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter AL und AX sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein F_GE (Typ iec61131::comparison::F_GE) weitergeleitet. Dort werden die aktuellen Werte von IN1.D1 und IN2.D1 verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (CNF) und gibt das Boolesche Ergebnis (OUT) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert OUT.D1 wird auf TRUE oder FALSE gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (AL/AX), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than or equal to“) für den Datentyp LWORD.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: erweiterte Bitmasken gegen einen Referenz- oder Schwellwert vom Typ LWORD prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ LWORD die Bedingung „größer gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf LWORD-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

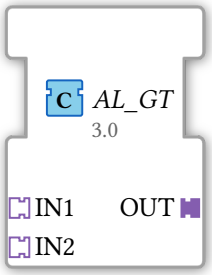
- **AL_LE**: Prüft auf „kleiner gleich“ – invers zur Funktion von AL_GE.
- **AL_EQ**: Prüft auf Gleichheit der Werte.
- **AL_GT**: Führt den Vergleich „größer als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version AL_GE ist.

Der AL_GE ergänzt die AL-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

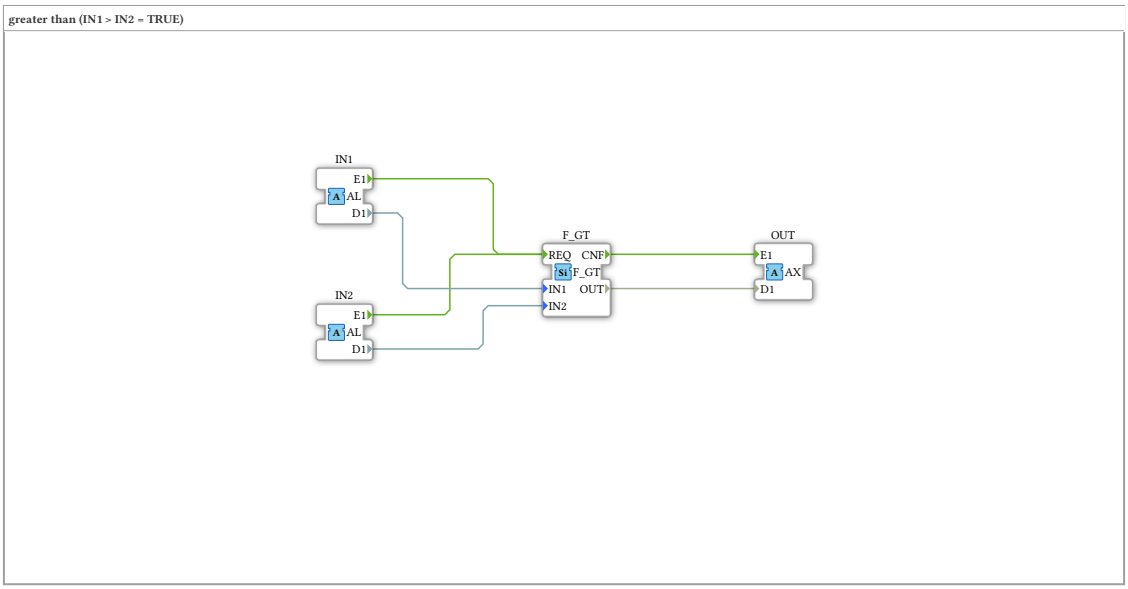
Der **AL_GE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp LWORD. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AL_GT



Input value 1 -- left operand (IN1 > IN2) – *AL*
Input value 2 -- right operand (IN1 > IN2) – *AL*

AX – IN1 > IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AL_GT** führt einen Vergleich zweier Werte vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)) durch und prüft, ob der erste Wert (IN1) größer als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `LWORD`, 64-Bit-Bitmuster (Langwort)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 > IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	AL (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	AL (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	AX (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter AL und AX sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein F_GT (Typ iec61131::comparison::F_GT) weitergeleitet. Dort werden die aktuellen Werte von IN1.D1 und IN2.D1 verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (CNF) und gibt das Boolesche Ergebnis (OUT) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert OUT.D1 wird auf TRUE oder FALSE gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (AL/AX), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than“) für den Datentyp LWORD.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: erweiterte Bitmasken gegen einen Referenz- oder Schwellwert vom Typ LWORD prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ LWORD die Bedingung „größer als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf LWORD-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AL_LT**: Prüft auf „kleiner als“ – invers zur Funktion von AL_GT.
- **AL_EQ**: Prüft auf Gleichheit der Werte.
- **AL_GE**: Führt den Vergleich „größer gleich“ durch – die um den Gleichheitsfall erweiterte Version von AL_GT.

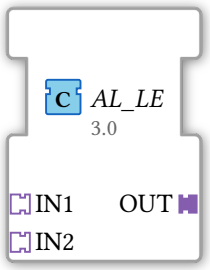
Der AL_GT ist die grundlegende „größer als“-Komponente ohne Gleichheitsfall, ergänzt durch AL_GE.

Fazit

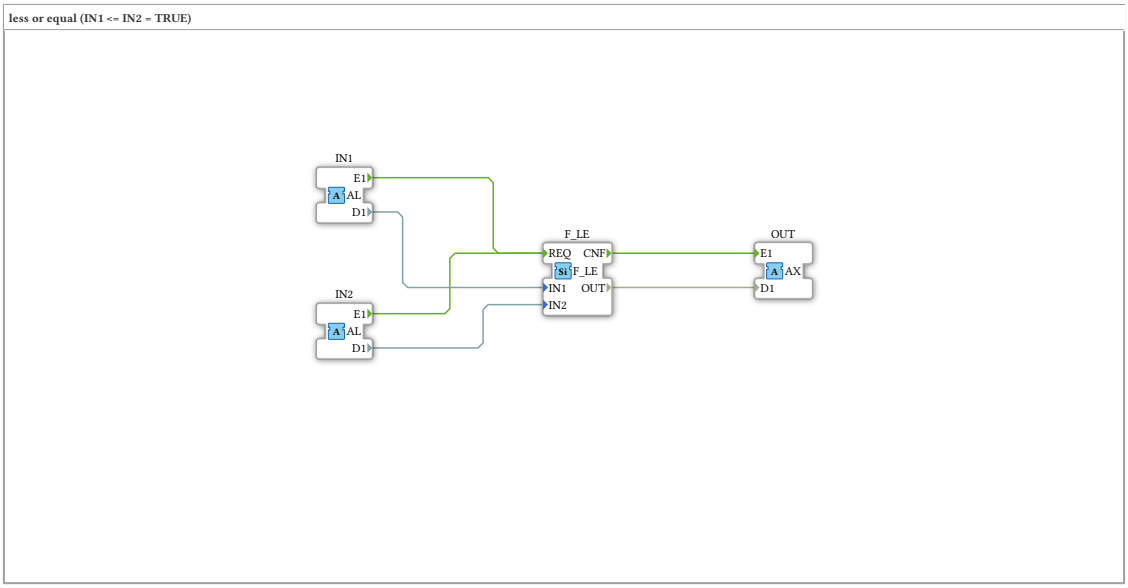
Der **AL_GT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp LWORD. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise

eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AL_LE



Input value 1 -- left operand (IN1 <= IN2) – *AL*
Input value 2 -- right operand (IN1 <= IN2) – *AL*
AX – IN1 <= IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AL_LE** führt einen Vergleich zweier Werte vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)) durch und prüft, ob der erste Wert (IN1) kleiner gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `LWORD`, 64-Bit-Bitmuster (Langwort)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 <= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AL` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AL` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AL` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LE` (Typ `iec61131::comparison::F_LE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AL`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than or equal to“) für den Datentyp `LWORD`.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: erweiterte Bitmasken gegen einen Referenz- oder Schwellwert vom Typ `LWORD` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `LWORD` die Bedingung „kleiner gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `LWORD`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AL_GE**: Prüft auf „größer gleich“ – invers zur Funktion von AL_LE.
- **AL_EQ**: Prüft auf Gleichheit der Werte.
- **AL_LT**: Führt den Vergleich „kleiner als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version AL_LE ist.

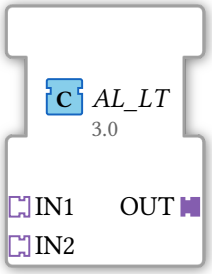
Der AL_LE ergänzt die AL-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

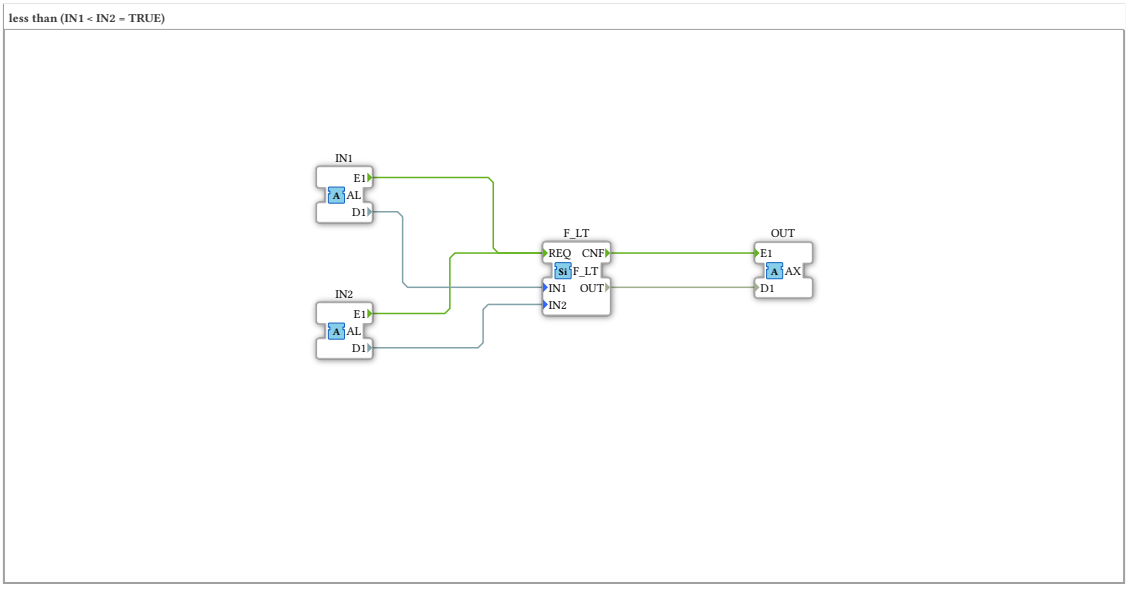
Der **AL_LE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `LWORD`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AL_LT

Input value 1 -- left operand (IN1 < IN2) – *AL*
Input value 2 -- right operand (IN1 < IN2) – *AL*



AX – IN1 < IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AL_LT** führt einen Vergleich zweier Werte vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)) durch und prüft, ob der erste Wert (IN1) kleiner als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `LWORD`, 64-Bit-Bitmuster (Langwort)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 < IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AL` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AL` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AL` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LT` (Typ `iec61131::comparison::F_LT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AL`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than“) für den Datentyp `LWORD`.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: erweiterte Bitmasken gegen einen Referenz- oder Schwellwert vom Typ `LWORD` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `LWORD` die Bedingung „kleiner als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `LWORD`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AL_GT**: Prüft auf „größer als“ – invers zur Funktion von AL_LT.
- **AL_EQ**: Prüft auf Gleichheit der Werte.

- **AL_LE**: Führt den Vergleich „kleiner gleich“ durch – die um den Gleichheitsfall erweiterte Version von AL_LT.

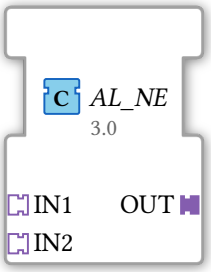
Der AL_LT ist die grundlegende „kleiner als“-Komponente ohne Gleichheitsfall, ergänzt durch AL_LE.

Fazit

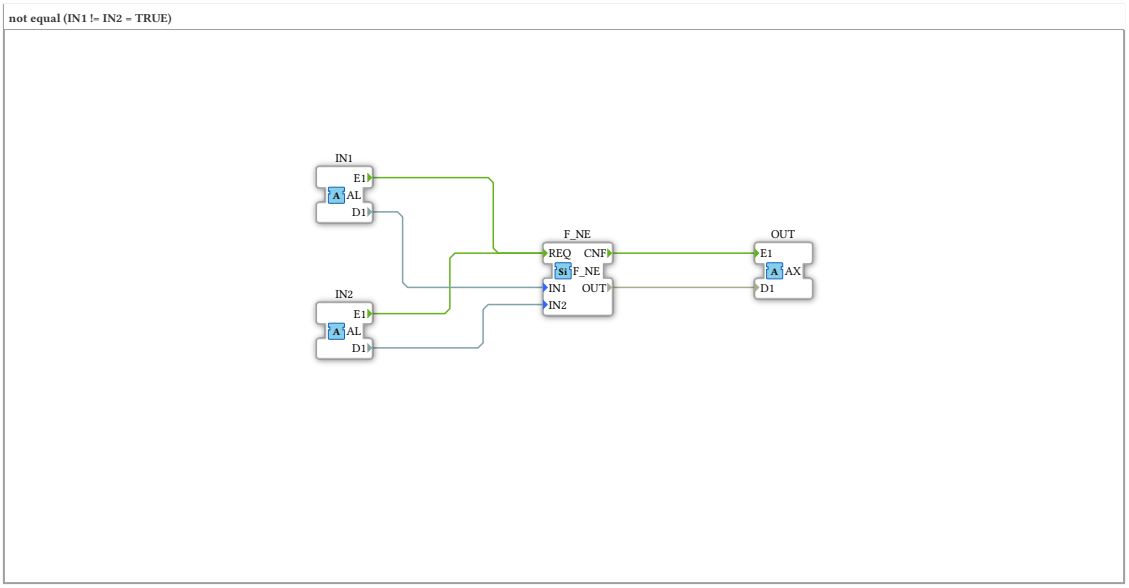
Der **AL_LT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `LWORD`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AL_NE

Input value 1 -- left operand (IN1 != IN2) – *AL*
Input value 2 -- right operand (IN1 != IN2) – *AL*



AX – IN1 != IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AL_NE** führt einen Vergleich zweier Werte vom Typ `LWORD` (64-Bit-Bitmuster (Langwort)) durch und prüft, ob der erste Wert (IN1) ungleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `LWORD`, 64-Bit-Bitmuster (Langwort)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 != IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AL` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AL` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AL` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_NE` (Typ `iec61131::comparison::F_NE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AL`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („not equal to“) für den Datentyp `LWORD`.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: erweiterte Bitmasken gegen einen Referenz- oder Schwellwert vom Typ `LWORD` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `LWORD` die Bedingung „ungleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `LWORD`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AL_EQ**: Prüft auf „gleich“ – invers zur Funktion von AL_NE.
- **AL_EQ**: Prüft auf Gleichheit der Werte.

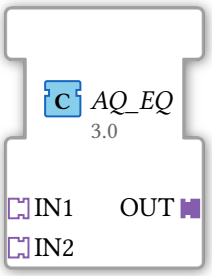
Der AL_NE ist Teil der vollständigen AL-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `LWORD`.

Fazit

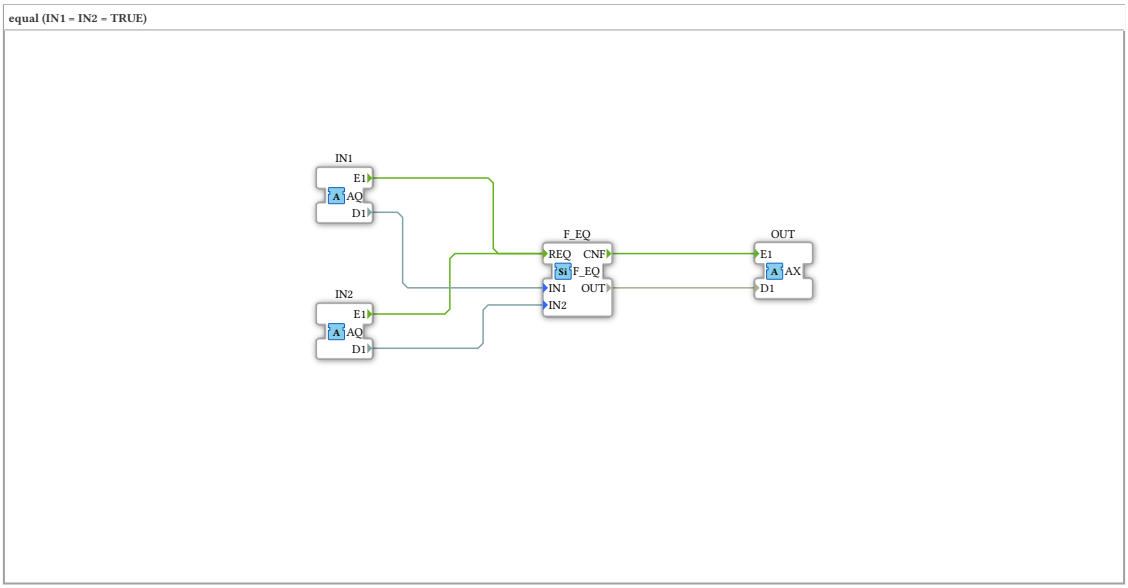
Der **AL_NE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `LWORD`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AQ_EQ

Input value 1 -- left operand (IN1 = IN2) – *AQ*
Input value 2 -- right operand (IN1 = IN2) – *AQ*



AX – IN1 = IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AQ_EQ** führt einen Vergleich zweier Werte vom Typ `QUARTER` (2-Bit-Wert (Viertel-Byte)) durch und prüft, ob der erste Wert (IN1) gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `QUARTER`, 2-Bit-Wert (Viertel-Byte)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 = IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AQ` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AQ` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AQ` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_EQ` (Typ `iec61131::comparison::F_EQ`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AQ`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („equal to“) für den Datentyp `QUARTER`.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: kompakte Statusfelder gegen einen Referenz- oder Schwellwert vom Typ `QUARTER` prüfen.

- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `QUARTER` die Bedingung „gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `QUARTER`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AQ_NE**: Prüft auf „ungleich“ – invers zur Funktion von AQ_EQ.
- **AQ_NE**: Prüft auf Ungleichheit der Werte.

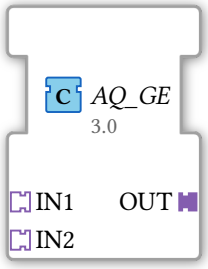
Der AQ_EQ ist Teil der vollständigen AQ-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `QUARTER`.

Fazit

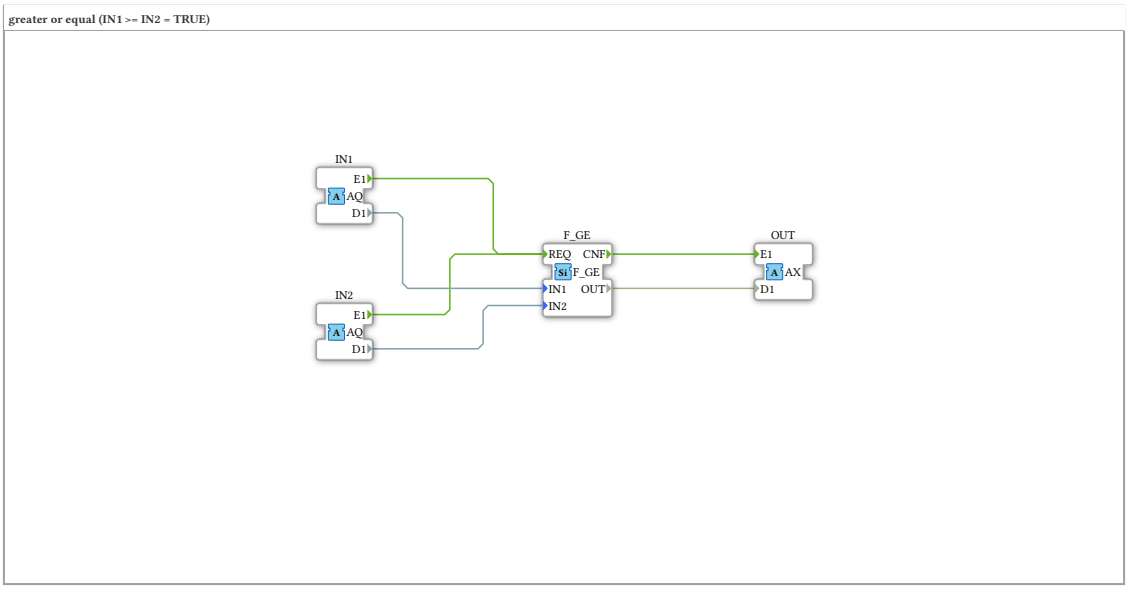
Der **AQ_EQ** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `QUARTER`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AQ_GE

Input value 1 -- left operand (IN1 >= IN2) – *AQ*
Input value 2 -- right operand (IN1 >= IN2) – *AQ*



AX – IN1 >= IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AQ_GE** führt einen Vergleich zweier Werte vom Typ `QUARTER` (2-Bit-Wert (Viertel-Byte)) durch und prüft, ob der erste Wert (IN1) größer gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `QUARTER`, 2-Bit-Wert (Viertel-Byte)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 >= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AQ` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AQ` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AQ` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GE` (Typ `iec61131::comparison::F_GE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AQ`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than or equal to“) für den Datentyp `QUARTER`.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: kompakte Statusfelder gegen einen Referenz- oder Schwellwert vom Typ `QUARTER` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `QUARTER` die Bedingung „größer gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `QUARTER`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AQ_LE**: Prüft auf „kleiner gleich“ – invers zur Funktion von AQ_GE.
- **AQ_EQ**: Prüft auf Gleichheit der Werte.
- **AQ_GT**: Führt den Vergleich „größer als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version AQ_GE ist.

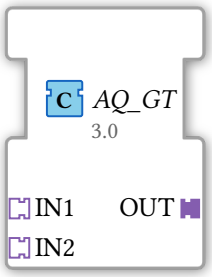
Der AQ_GE ergänzt die AQ-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

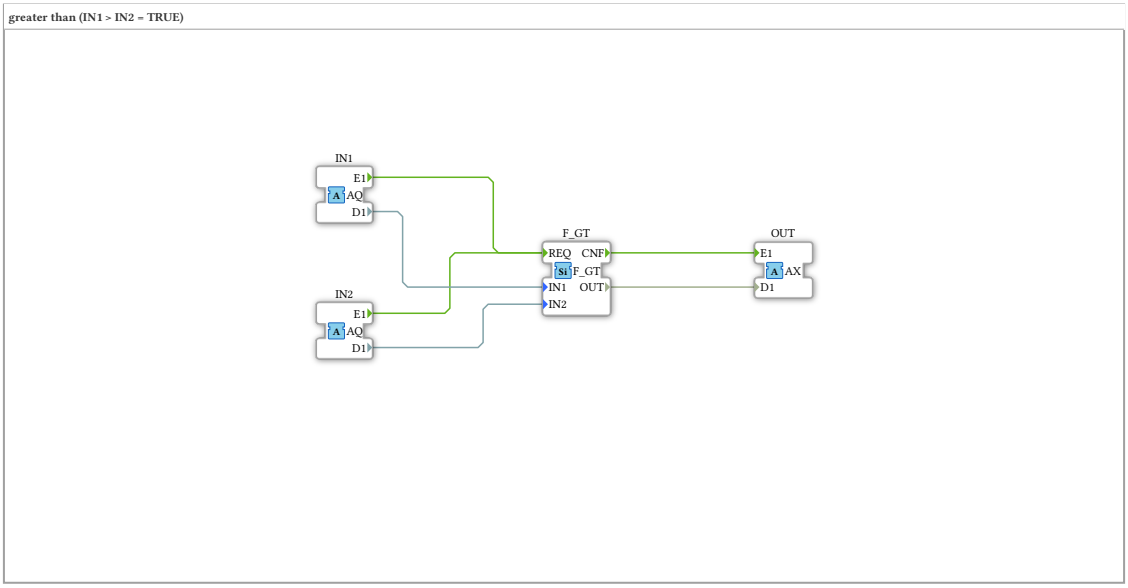
Der **AQ_GE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `QUARTER`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AQ_GT

Input value 1 -- left operand (IN1 > IN2) – *AQ*
Input value 2 -- right operand (IN1 > IN2) – *AQ*



AX – IN1 > IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AQ_GT** führt einen Vergleich zweier Werte vom Typ `QUARTER` (2-Bit-Wert (Viertel-Byte)) durch und prüft, ob der erste Wert (IN1) größer als der zweite Wert (IN2) ist. Das Ergebnis wird

als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `QUARTER`, 2-Bit-Wert (Viertel-Byte)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 > IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AQ` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AQ` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AQ` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GT` (Typ `iec61131::comparison::F_GT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AQ`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than“) für den Datentyp `QUARTER`.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: kompakte Statusfelder gegen einen Referenz- oder Schwellwert vom Typ `QUARTER` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `QUARTER` die Bedingung „größer als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `QUARTER`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AQ_LT**: Prüft auf „kleiner als“ – invers zur Funktion von AQ_GT.
- **AQ_EQ**: Prüft auf Gleichheit der Werte.
- **AQ_GE**: Führt den Vergleich „größer gleich“ durch – die um den Gleichheitsfall erweiterte Version von AQ_GT.

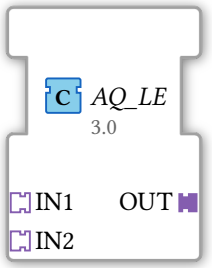
Der AQ_GT ist die grundlegende „größer als“-Komponente ohne Gleichheitsfall, ergänzt durch AQ_GE.

Fazit

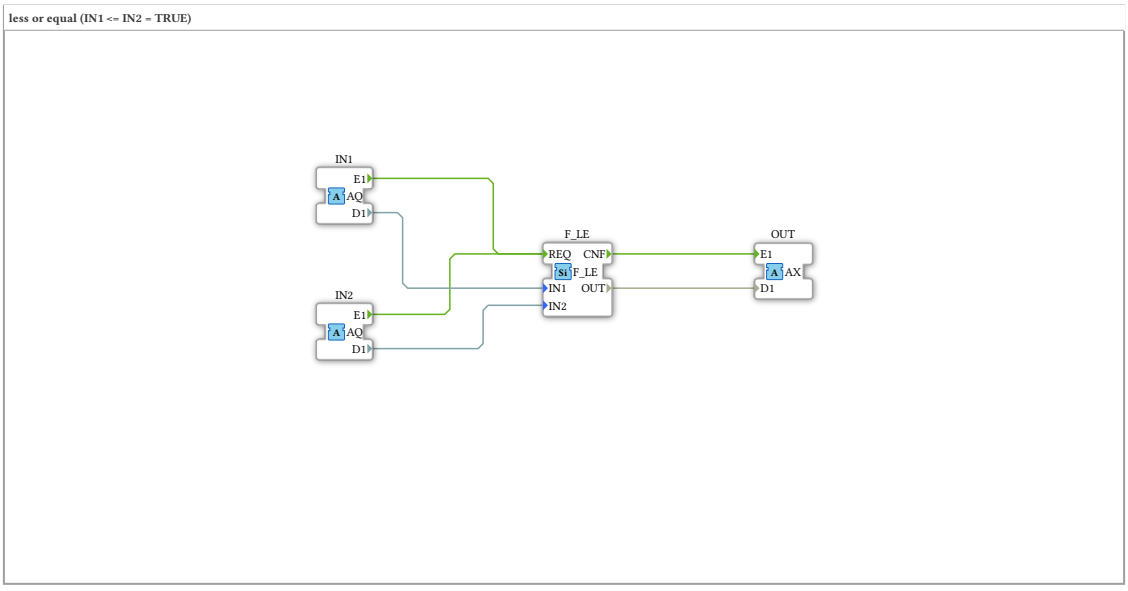
Der **AQ_GT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `QUARTER`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AQ_LE

Input value 1 -- left operand (IN1 <= IN2) – *AQ*
Input value 2 -- right operand (IN1 <= IN2) – *AQ*



AX – IN1 <= IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AQ_LE** führt einen Vergleich zweier Werte vom Typ `QUARTER` (2-Bit-Wert (Viertel-Byte)) durch und prüft, ob der erste Wert (IN1) kleiner gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `QUARTER`, 2-Bit-Wert (Viertel-Byte)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 <= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AQ` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AQ` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AQ` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LE` (Typ `iec61131::comparison::F_LE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AQ`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than or equal to“) für den Datentyp `QUARTER`.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: kompakte Statusfelder gegen einen Referenz- oder Schwellwert vom Typ `QUARTER` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `QUARTER` die Bedingung „kleiner gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `QUARTER`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

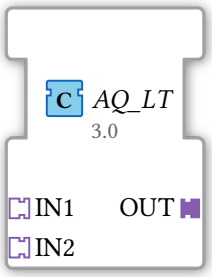
- **AQ_GE**: Prüft auf „größer gleich“ – invers zur Funktion von AQ_LE.
- **AQ_EQ**: Prüft auf Gleichheit der Werte.
- **AQ_LT**: Führt den Vergleich „kleiner als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version AQ_LE ist.

Der AQ_LE ergänzt die AQ-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

Der **AQ_LE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `QUARTER`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

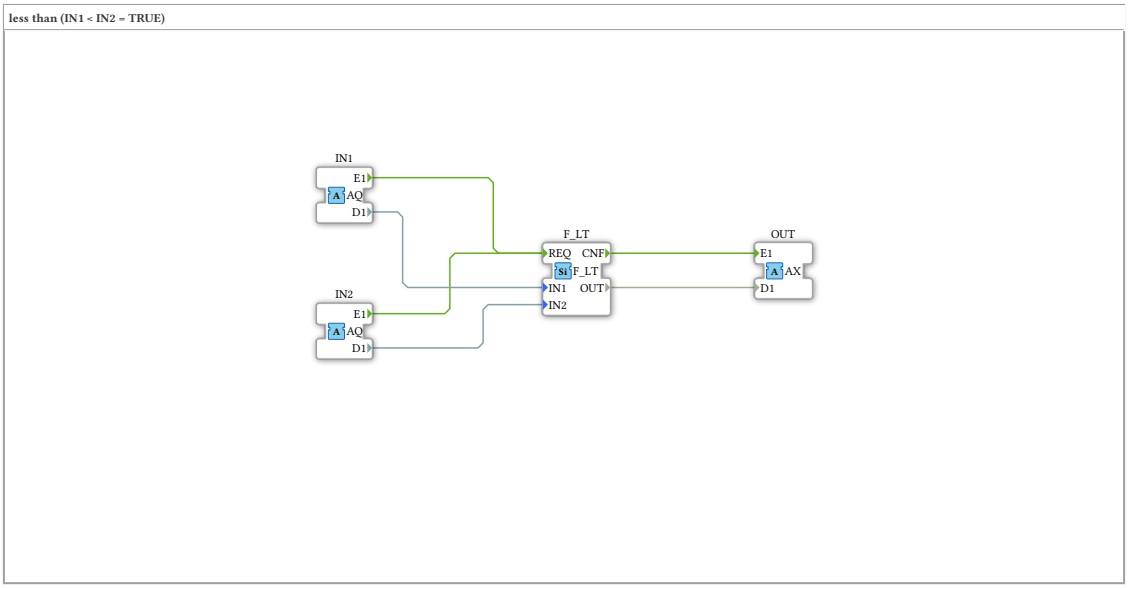
AQ_LT



Input value 1 -- left operand (IN1 < IN2) – *AQ*

Input value 2 -- right operand (IN1 < IN2) – *AQ*

AX – IN1 < IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AQ_LT** führt einen Vergleich zweier Werte vom Typ `QUARTER` (2-Bit-Wert (Viertel-Byte)) durch und prüft, ob der erste Wert (IN1) kleiner als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `QUARTER`, 2-Bit-Wert (Viertel-Byte)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 < IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AQ` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AQ` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AQ` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LT` (Typ `iec61131::comparison::F_LT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AQ`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than“) für den Datentyp `QUARTER`.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: kompakte Statusfelder gegen einen Referenz- oder Schwellwert vom Typ `QUARTER` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `QUARTER` die Bedingung „kleiner als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `QUARTER`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

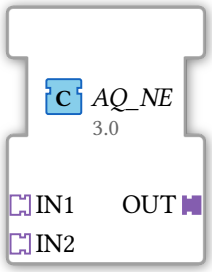
- **AQ_GT**: Prüft auf „größer als“ – invers zur Funktion von AQ_LT.
- **AQ_EQ**: Prüft auf Gleichheit der Werte.
- **AQ_LE**: Führt den Vergleich „kleiner gleich“ durch – die um den Gleichheitsfall erweiterte Version von AQ_LT.

Der AQ_LT ist die grundlegende „kleiner als“-Komponente ohne Gleichheitsfall, ergänzt durch AQ_LE.

Fazit

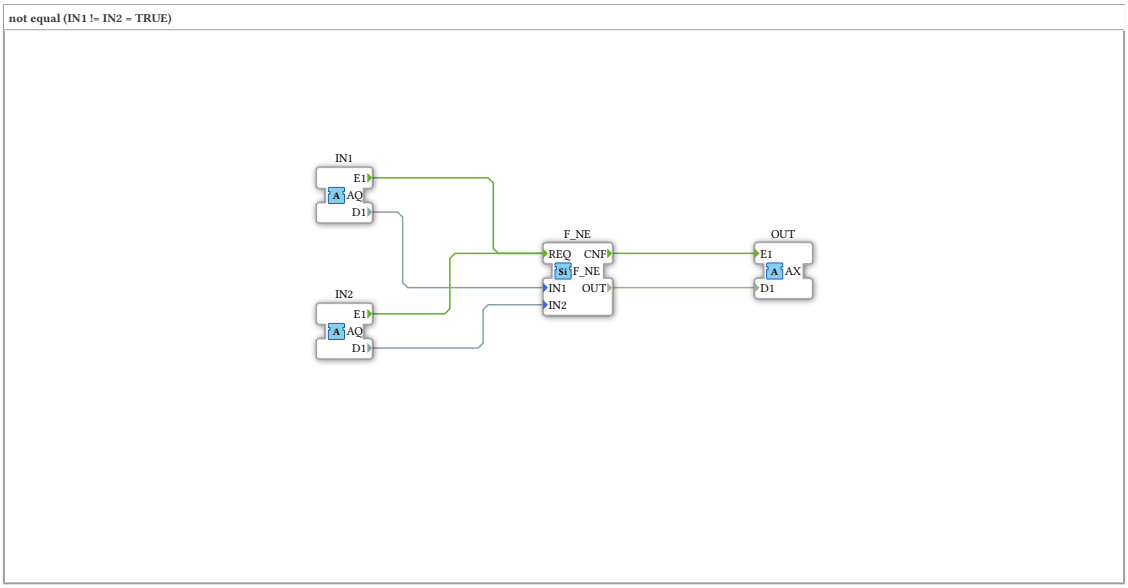
Der **AQ_LT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `QUARTER`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AQ_NE



Input value 1 -- left operand (IN1 != IN2) – *AQ*
Input value 2 -- right operand (IN1 != IN2) – *AQ*

AX – IN1 != IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AQ_NE** führt einen Vergleich zweier Werte vom Typ `QUARTER` (2-Bit-Wert (Viertel-Byte)) durch und prüft, ob der erste Wert (IN1) ungleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `QUARTER`, 2-Bit-Wert (Viertel-Byte)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 != IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	AQ (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	AQ (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	AX (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter AQ und AX sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein F_NE (Typ iec61131::comparison::F_NE) weitergeleitet. Dort werden die aktuellen Werte von IN1.D1 und IN2.D1 verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (CNF) und gibt das Boolesche Ergebnis (OUT) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert OUT.D1 wird auf TRUE oder FALSE gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (AQ/AX), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („not equal to“) für den Datentyp QUARTER.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: kompakte Statusfelder gegen einen Referenz- oder Schwellwert vom Typ QUARTER prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ QUARTER die Bedingung „ungleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf QUARTER-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AQ_EQ**: Prüft auf „gleich“ – invers zur Funktion von AQ_NE.
- **AQ_EQ**: Prüft auf Gleichheit der Werte.

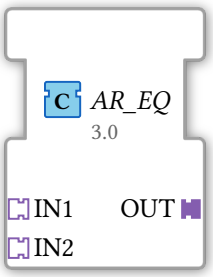
Der AQ_NE ist Teil der vollständigen AQ-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp QUARTER.

Fazit

Der **AQ_NE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp QUARTER. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeits-

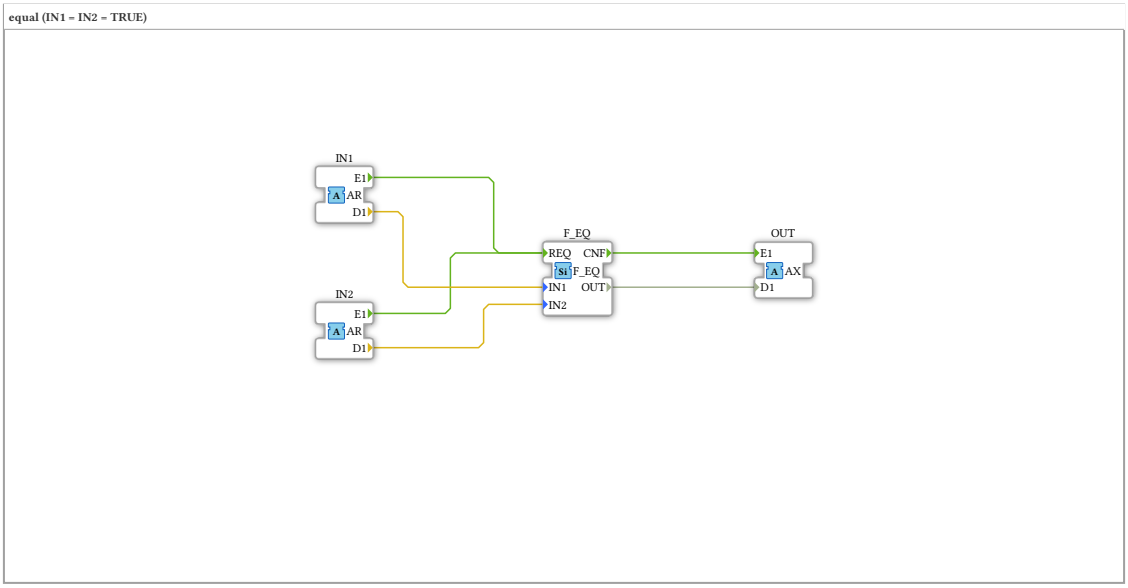
weise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AR_EQ



Input value 1 -- left operand (IN1 = IN2) – *AR*
Input value 2 -- right operand (IN1 = IN2) – *AR*

AX – IN1 = IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AR_EQ** führt einen Vergleich zweier Werte vom Typ `REAL` (32-Bit-Gleitkommazahl (einfache Genauigkeit)) durch und prüft, ob der erste Wert (IN1) gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `REAL`, 32-Bit-Gleitkommazahl (einfache Genauigkeit)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 = IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AR` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AR` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AR` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_EQ` (Typ `iec61131::comparison::F_EQ`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AR`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („equal to“) für den Datentyp `REAL`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Messwerte gegen einen Referenz- oder Schwellwert vom Typ `REAL` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `REAL` die Bedingung „gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `REAL`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AR_NE**: Prüft auf „ungleich“ – invers zur Funktion von AR_EQ.
- **AR_NE**: Prüft auf Ungleichheit der Werte.

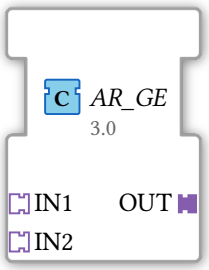
Der AR_EQ ist Teil der vollständigen AR-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `REAL`.

Fazit

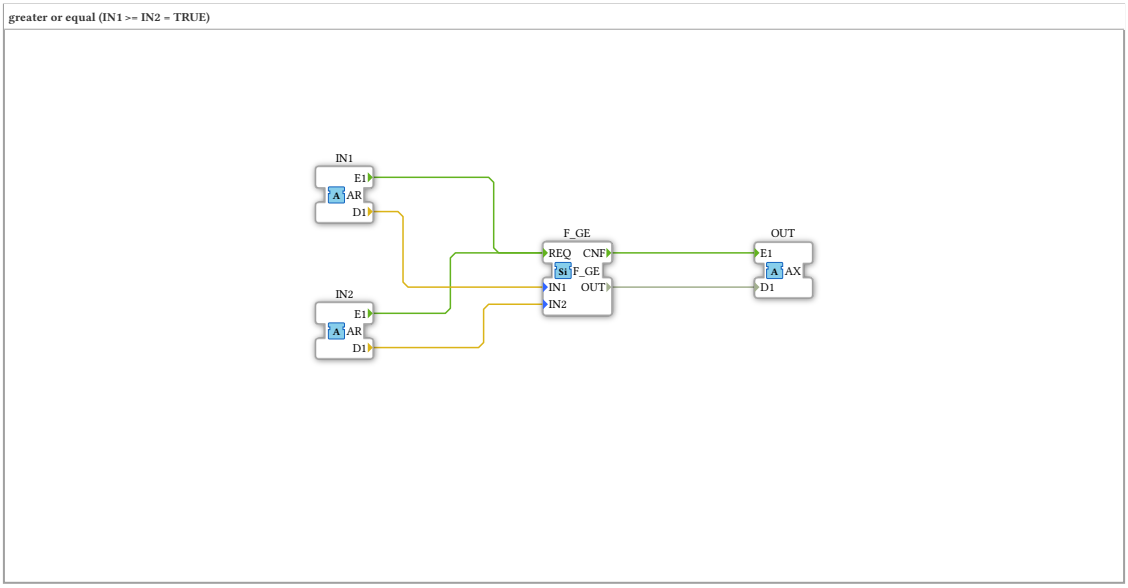
Der **AR_EQ** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `REAL`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise

eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AR_GE



Input value 1 -- left operand (IN1 >= IN2) – *AR*
Input value 2 -- right operand (IN1 >= IN2) – *AR*
AX – IN1 >= IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AR_GE** führt einen Vergleich zweier Werte vom Typ `REAL` (32-Bit-Gleitkommazahl (einfache Genauigkeit)) durch und prüft, ob der erste Wert (IN1) größer gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `REAL`, 32-Bit-Gleitkommazahl (einfache Genauigkeit)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 >= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AR` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AR` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AR` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GE` (Typ `iec61131::comparison::F_GE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AR`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than or equal to“) für den Datentyp `REAL`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Messwerte gegen einen Referenz- oder Schwellwert vom Typ `REAL` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `REAL` die Bedingung „größer gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `REAL`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AR_LE**: Prüft auf „kleiner gleich“ – invers zur Funktion von AR_GE.
- **AR_EQ**: Prüft auf Gleichheit der Werte.
- **AR_GT**: Führt den Vergleich „größer als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version AR_GE ist.

Der AR_GE ergänzt die AR-Vergleichsfamilie um den Grenzfall der Gleichheit.

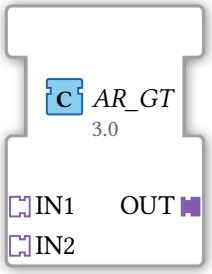
Fazit

Der **AR_GE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `REAL`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermög-

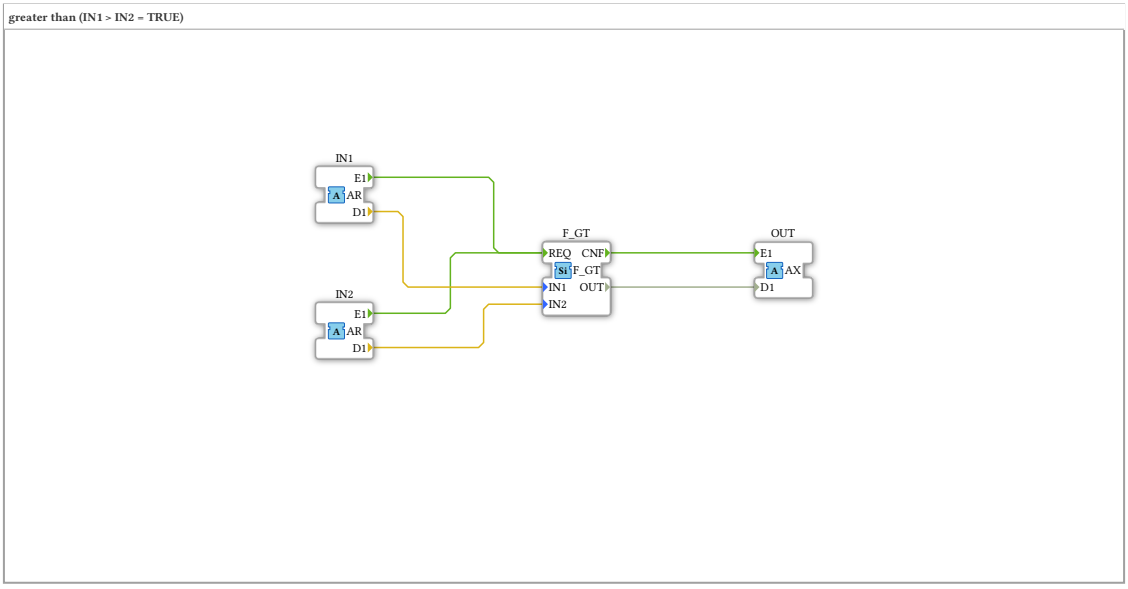
licht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AR_GT

Input value 1 -- left operand (IN1 > IN2) – *AR*
Input value 2 -- right operand (IN1 > IN2) – *AR*



AX – IN1 > IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AR_GT** führt einen Vergleich zweier Werte vom Typ `REAL` (32-Bit-Gleitkommazahl (einfache Genauigkeit)) durch und prüft, ob der erste Wert (IN1) größer als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `REAL`, 32-Bit-Gleitkommazahl (einfache Genauigkeit)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 > IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AR` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AR` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AR` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GT` (Typ `iec61131::comparison::F_GT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AR`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than“) für den Datentyp `REAL`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Messwerte gegen einen Referenz- oder Schwellwert vom Typ `REAL` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `REAL` die Bedingung „größer als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `REAL`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AR_LT**: Prüft auf „kleiner als“ – invers zur Funktion von AR_GT.
- **AR_EQ**: Prüft auf Gleichheit der Werte.
- **AR_GE**: Führt den Vergleich „größer gleich“ durch – die um den Gleichheitsfall erweiterte Version von AR_GT.

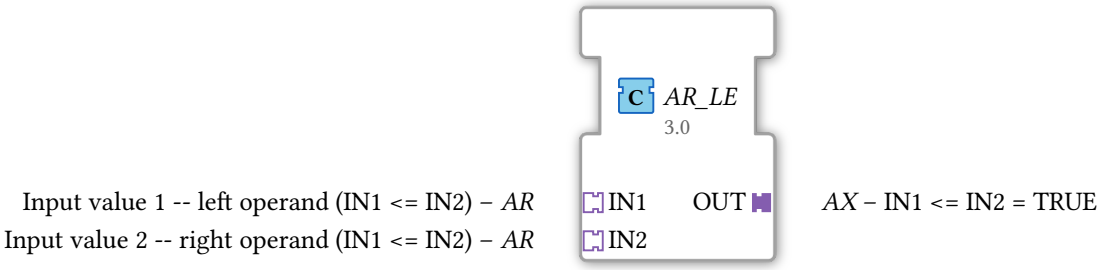
Der AR_GT ist die grundlegende „größer als“-Komponente ohne Gleichheitsfall, ergänzt durch AR_GE.

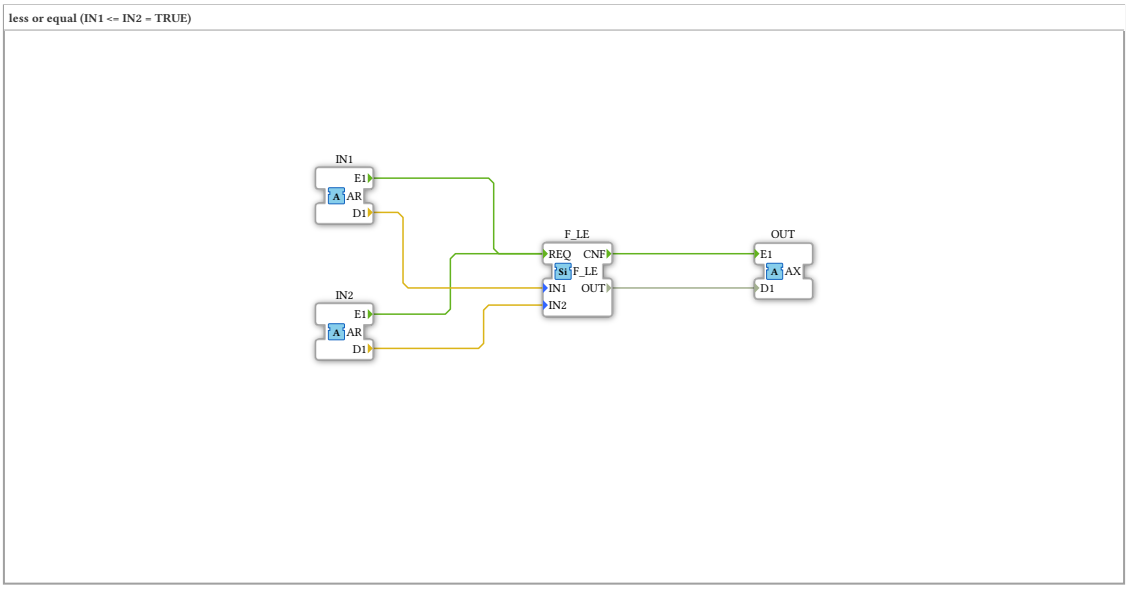
Fazit

Der **AR_GT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `REAL`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermög-

licht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AR_LE





• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AR_LE** führt einen Vergleich zweier Werte vom Typ `REAL` (32-Bit-Gleitkommazahl (einfache Genauigkeit)) durch und prüft, ob der erste Wert (IN1) kleiner gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `REAL`, 32-Bit-Gleitkommazahl (einfache Genauigkeit)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 <= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AR` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AR` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AR` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LE` (Typ `iec61131::comparison::F_LE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AR`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than or equal to“) für den Datentyp `REAL`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Messwerte gegen einen Referenz- oder Schwellwert vom Typ `REAL` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `REAL` die Bedingung „kleiner gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `REAL`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AR_GE**: Prüft auf „größer gleich“ – invers zur Funktion von AR_LE.
- **AR_EQ**: Prüft auf Gleichheit der Werte.
- **AR_LT**: Führt den Vergleich „kleiner als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version AR_LE ist.

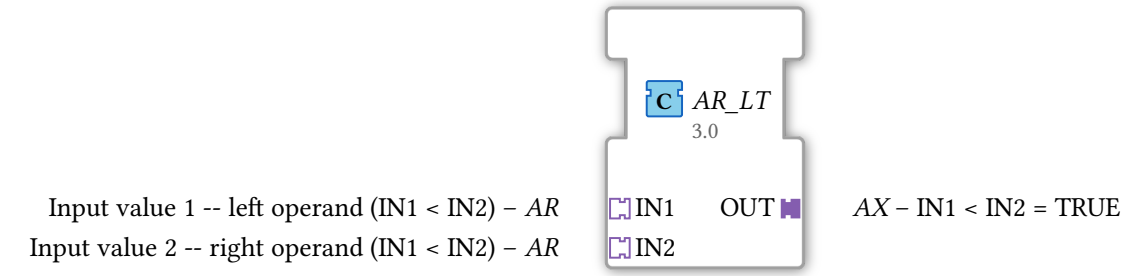
Der AR_LE ergänzt die AR-Vergleichsfamilie um den Grenzfall der Gleichheit.

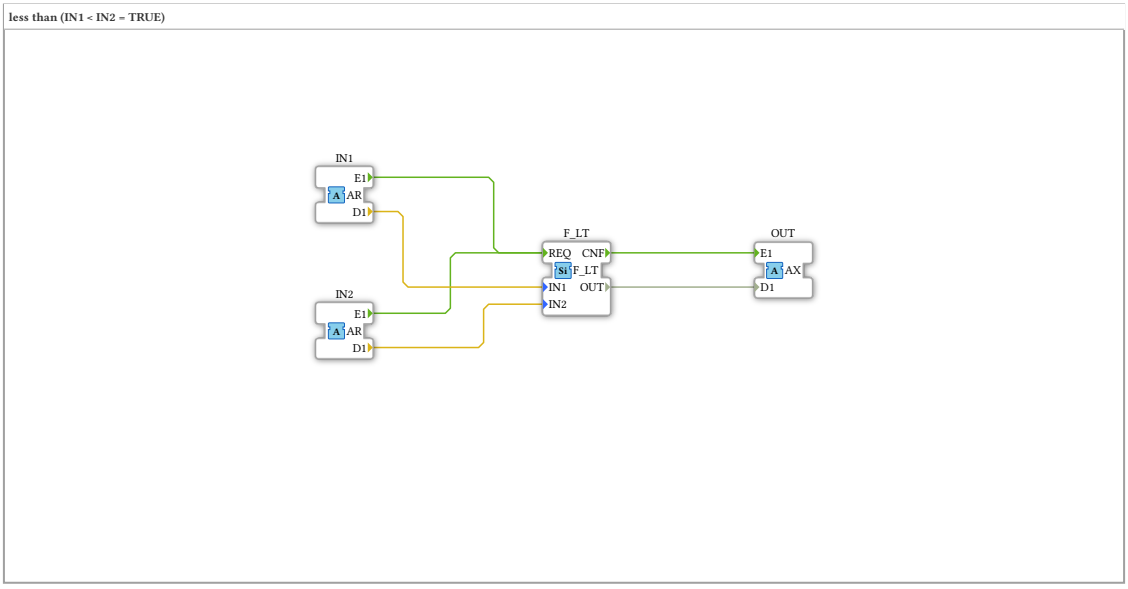
Fazit

Der **AR_LE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `REAL`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermög-

licht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AR_LT





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AR_LT** führt einen Vergleich zweier Werte vom Typ `REAL` (32-Bit-Gleitkommazahl (einfache Genauigkeit)) durch und prüft, ob der erste Wert (IN1) kleiner als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `REAL`, 32-Bit-Gleitkommazahl (einfache Genauigkeit)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 < IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AR` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AR` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AR` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LT` (Typ `iec61131::comparison::F_LT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AR`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than“) für den Datentyp `REAL`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Messwerte gegen einen Referenz- oder Schwellwert vom Typ `REAL` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `REAL` die Bedingung „kleiner als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `REAL`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AR_GT**: Prüft auf „größer als“ – invers zur Funktion von AR_LT.
- **AR_EQ**: Prüft auf Gleichheit der Werte.
- **AR_LE**: Führt den Vergleich „kleiner gleich“ durch – die um den Gleichheitsfall erweiterte Version von AR_LT.

Der AR_LT ist die grundlegende „kleiner als“-Komponente ohne Gleichheitsfall, ergänzt durch AR_LE.

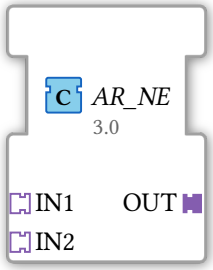
Fazit

Der **AR_LT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `REAL`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermög-

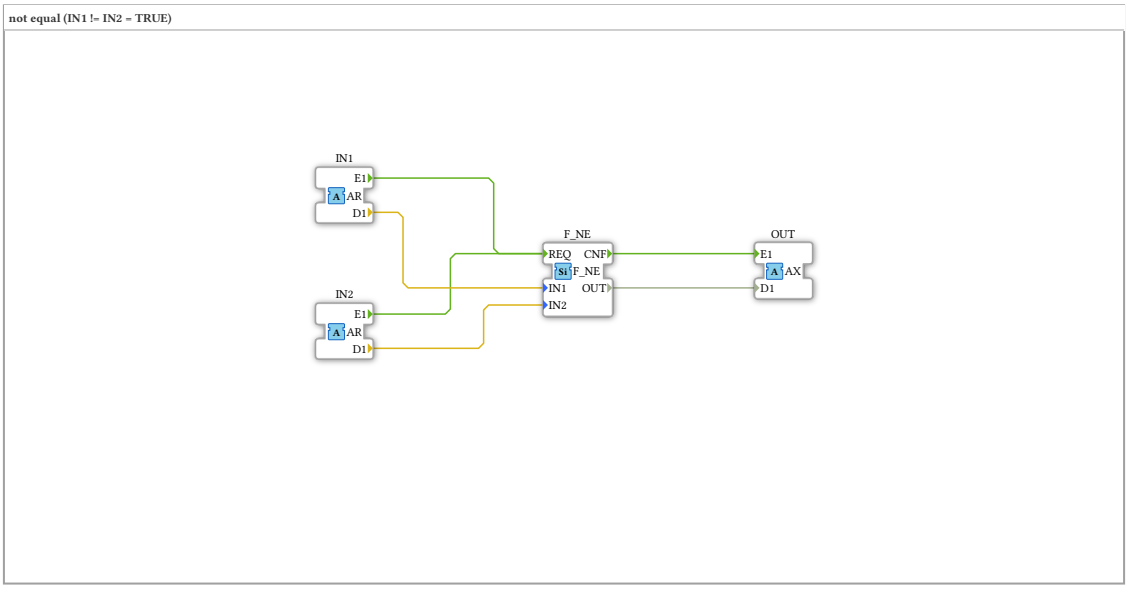
licht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AR_NE

Input value 1 -- left operand (IN1 != IN2) – *AR*
Input value 2 -- right operand (IN1 != IN2) – *AR*



AX – IN1 != IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AR_NE** führt einen Vergleich zweier Werte vom Typ `REAL` (32-Bit-Gleitkommazahl (einfache Genauigkeit)) durch und prüft, ob der erste Wert (IN1) ungleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `REAL`, 32-Bit-Gleitkommazahl (einfache Genauigkeit)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 != IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AR` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AR` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AR` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_NE` (Typ `iec61131::comparison::F_NE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AR`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („not equal to“) für den Datentyp `REAL`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Messwerte gegen einen Referenz- oder Schwellwert vom Typ `REAL` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `REAL` die Bedingung „ungleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `REAL`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AR_EQ**: Prüft auf „gleich“ – invers zur Funktion von AR_NE.
- **AR_EQ**: Prüft auf Gleichheit der Werte.

Der AR_NE ist Teil der vollständigen AR-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `REAL`.

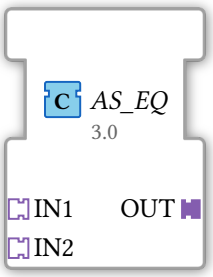
Fazit

Der **AR_NE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `REAL`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise

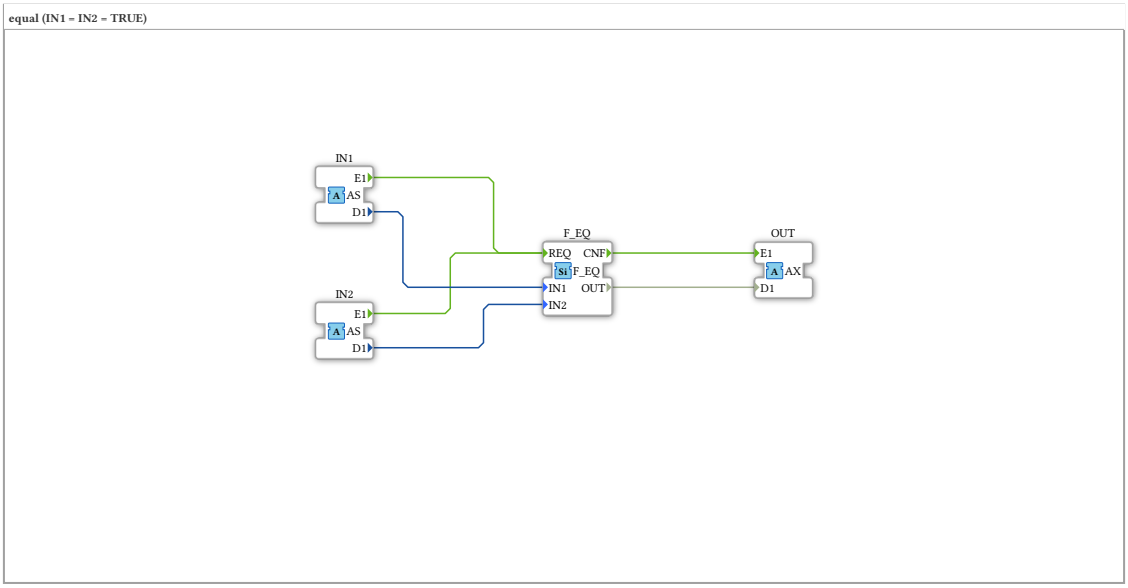
eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AS_EQ

Input value 1 -- left operand (IN1 = IN2) – *AS*
Input value 2 -- right operand (IN1 = IN2) – *AS*



AX – IN1 = IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AS_EQ** führt einen Vergleich zweier Werte vom Typ `SINT` (8-Bit-Ganzzahl (vorzeichenbehaftet)) durch und prüft, ob der erste Wert (IN1) gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `SINT`, 8-Bit-Ganzzahl (vorzeichenbehaftet)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 = IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AS` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AS` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AS` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_EQ` (Typ `iec61131::comparison::F_EQ`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AS`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („equal to“) für den Datentyp `SINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: kleine Zähl- und Indexwerte gegen einen Referenz- oder Schwellwert vom Typ `SINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `SINT` die Bedingung „gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `SINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AS_NE**: Prüft auf „ungleich“ – invers zur Funktion von AS_EQ.
- **AS_NE**: Prüft auf Ungleichheit der Werte.

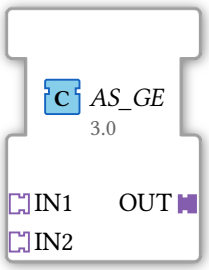
Der AS_EQ ist Teil der vollständigen AS-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `SINT`.

Fazit

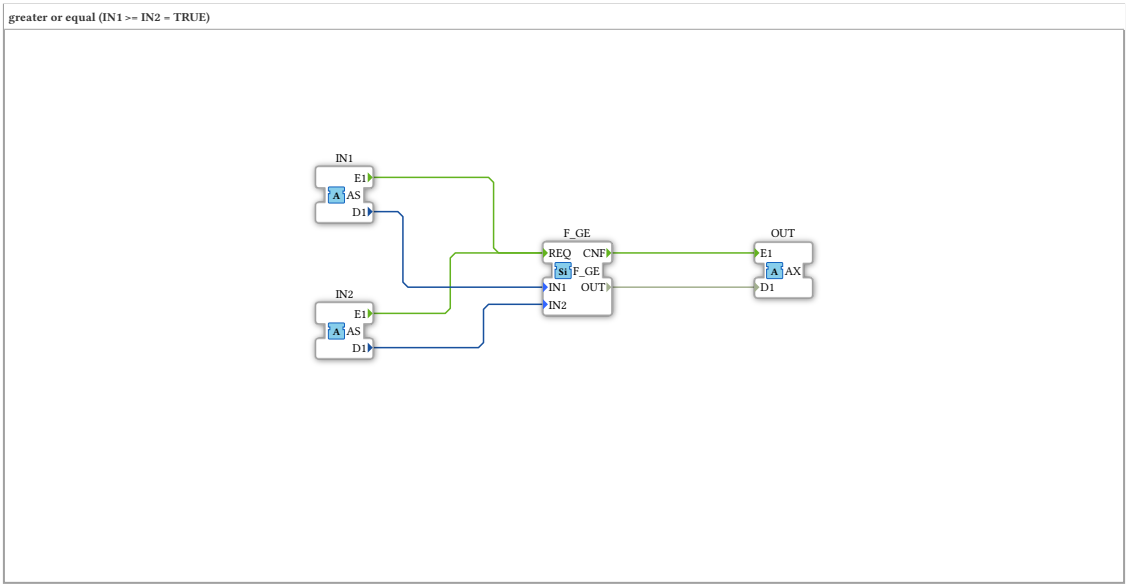
Der **AS_EQ** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `SINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise

eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AS_GE



Input value 1 -- left operand (IN1 >= IN2) – *AS*
Input value 2 -- right operand (IN1 >= IN2) – *AS*
AX – IN1 >= IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AS_GE** führt einen Vergleich zweier Werte vom Typ `SINT` (8-Bit-Ganzzahl (vorzeichenbehaftet)) durch und prüft, ob der erste Wert (IN1) größer gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `SINT`, 8-Bit-Ganzzahl (vorzeichenbehaftet)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 >= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AS` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AS` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AS` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GE` (Typ `iec61131::comparison::F_GE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AS`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than or equal to“) für den Datentyp `SINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: kleine Zähl- und Indexwerte gegen einen Referenz- oder Schwellwert vom Typ `SINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `SINT` die Bedingung „größer gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `SINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AS_LE**: Prüft auf „kleiner gleich“ – invers zur Funktion von AS_GE.
- **AS_EQ**: Prüft auf Gleichheit der Werte.
- **AS_GT**: Führt den Vergleich „größer als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version AS_GE ist.

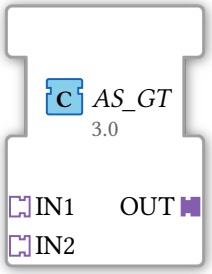
Der AS_GE ergänzt die AS-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

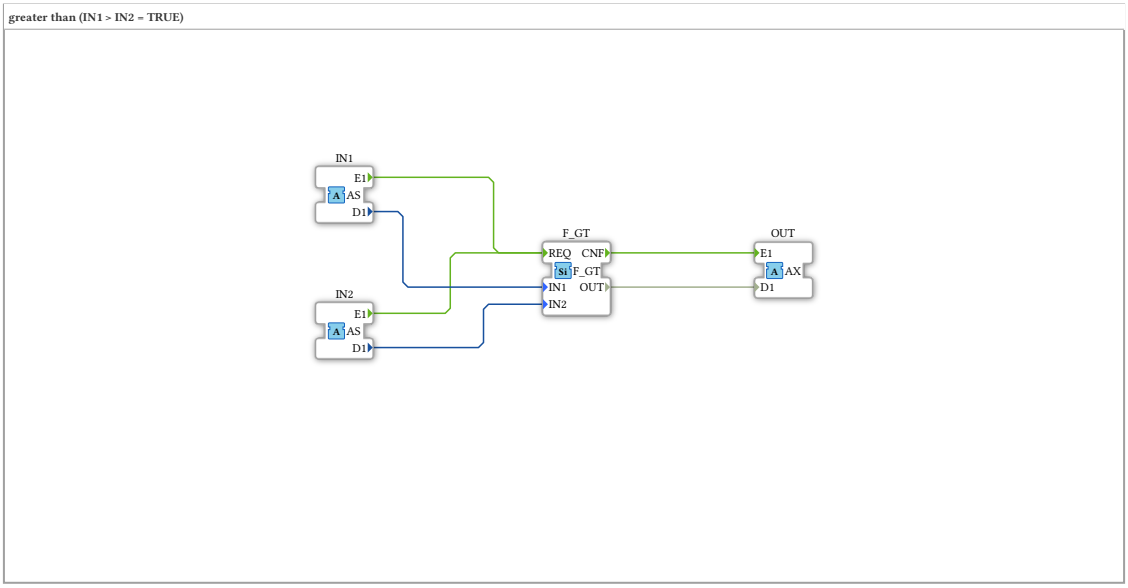
Der **AS_GE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `SINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AS_GT

Input value 1 -- left operand (IN1 > IN2) – *AS*
Input value 2 -- right operand (IN1 > IN2) – *AS*



AX – IN1 > IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AS_GT** führt einen Vergleich zweier Werte vom Typ `SINT` (8-Bit-Ganzzahl (vorzeichenbehaftet)) durch und prüft, ob der erste Wert (IN1) größer als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `SINT`, 8-Bit-Ganzzahl (vorzeichenbehaftet)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 > IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AS` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AS` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AS` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GT` (Typ `iec61131::comparison::F_GT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AS`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than“) für den Datentyp `SINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: kleine Zähl- und Indexwerte gegen einen Referenz- oder Schwellwert vom Typ `SINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `SINT` die Bedingung „größer als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `SINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AS_LT**: Prüft auf „kleiner als“ – invers zur Funktion von AS_GT.
- **AS_EQ**: Prüft auf Gleichheit der Werte.
- **AS_GE**: Führt den Vergleich „größer gleich“ durch – die um den Gleichheitsfall erweiterte Version von AS_GT.

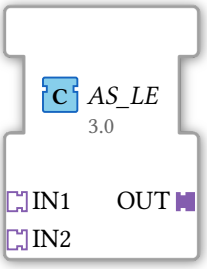
Der AS_GT ist die grundlegende „größer als“-Komponente ohne Gleichheitsfall, ergänzt durch AS_GE.

Fazit

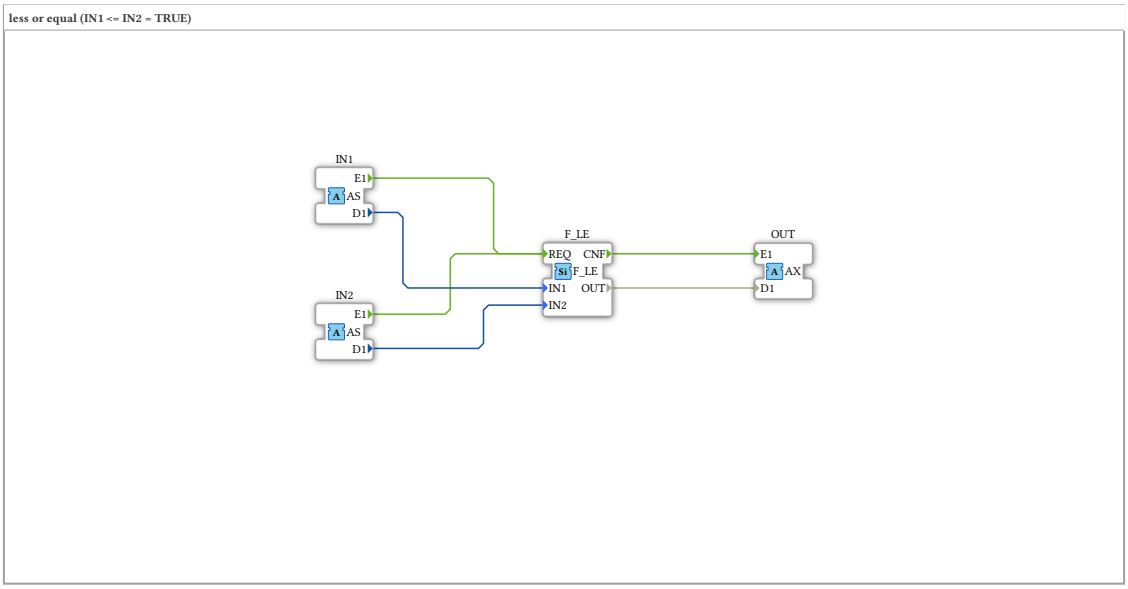
Der **AS_GT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `SINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AS_LE

Input value 1 -- left operand (IN1 <= IN2) – *AS*
Input value 2 -- right operand (IN1 <= IN2) – *AS*



AX – IN1 <= IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AS_LE** führt einen Vergleich zweier Werte vom Typ `SINT` (8-Bit-Ganzzahl (vorzeichenbehaftet)) durch und prüft, ob der erste Wert (IN1) kleiner gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `SINT`, 8-Bit-Ganzzahl (vorzeichenbehaftet)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 <= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AS` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AS` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AS` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LE` (Typ `iec61131::comparison::F_LE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AS`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than or equal to“) für den Datentyp `SINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: kleine Zähl- und Indexwerte gegen einen Referenz- oder Schwellwert vom Typ `SINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `SINT` die Bedingung „kleiner gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `SINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AS_GE**: Prüft auf „größer gleich“ – invers zur Funktion von AS_LE.
- **AS_EQ**: Prüft auf Gleichheit der Werte.
- **AS_LT**: Führt den Vergleich „kleiner als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version AS_LE ist.

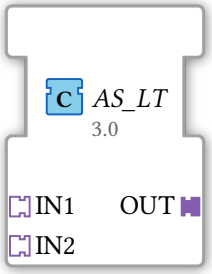
Der AS_LE ergänzt die AS-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

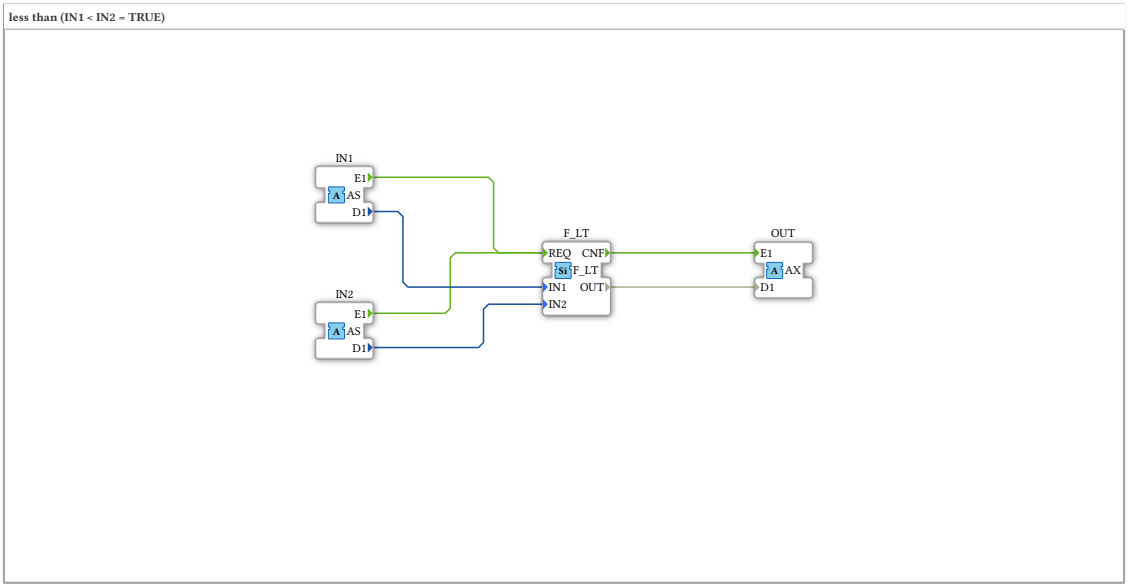
Der **AS_LE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `SINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AS_LT

Input value 1 -- left operand (IN1 < IN2) – *AS*
Input value 2 -- right operand (IN1 < IN2) – *AS*



AX – IN1 < IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AS_LT** führt einen Vergleich zweier Werte vom Typ `SINT` (8-Bit-Ganzzahl (vorzeichenbehaftet)) durch und prüft, ob der erste Wert (IN1) kleiner als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `SINT`, 8-Bit-Ganzzahl (vorzeichenbehaftet)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 < IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AS` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AS` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AS` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LT` (Typ `iec61131::comparison::F_LT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AS`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than“) für den Datentyp `SINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: kleine Zähl- und Indexwerte gegen einen Referenz- oder Schwellwert vom Typ `SINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `SINT` die Bedingung „kleiner als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `SINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AS_GT**: Prüft auf „größer als“ – invers zur Funktion von AS_LT.
- **AS_EQ**: Prüft auf Gleichheit der Werte.
- **AS_LE**: Führt den Vergleich „kleiner gleich“ durch – die um den Gleichheitsfall erweiterte Version von AS_LT.

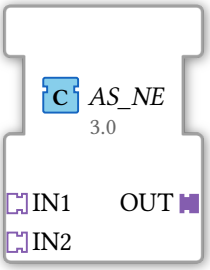
Der AS_LT ist die grundlegende „kleiner als“-Komponente ohne Gleichheitsfall, ergänzt durch AS_LE.

Fazit

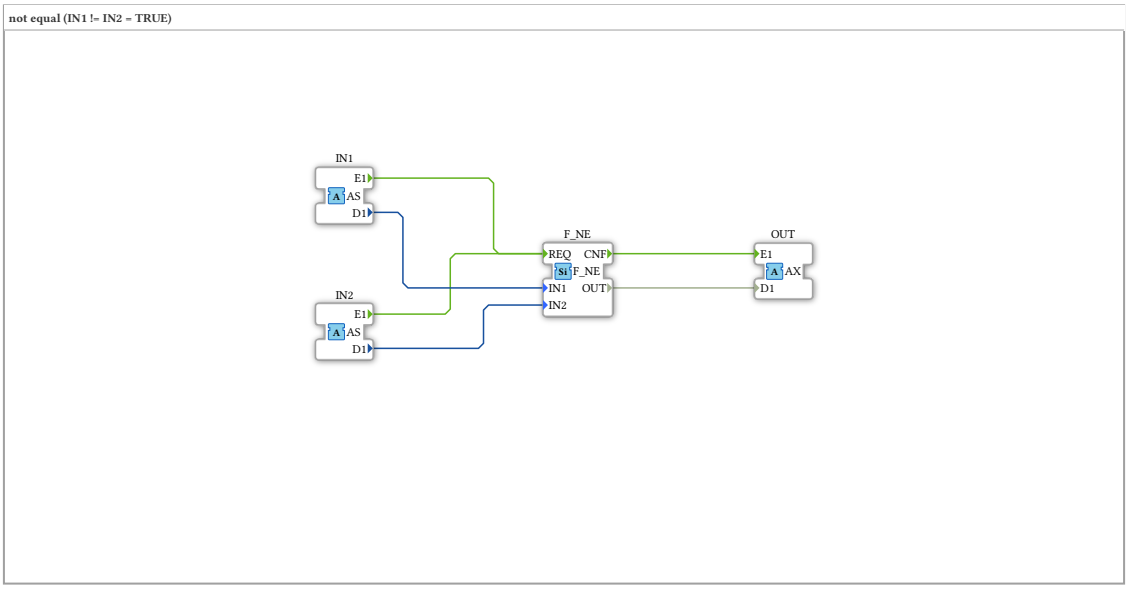
Der **AS_LT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `SINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AS_NE

Input value 1 -- left operand (IN1 != IN2) – *AS*
Input value 2 -- right operand (IN1 != IN2) – *AS*



AX – IN1 != IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AS_NE** führt einen Vergleich zweier Werte vom Typ `SINT` (8-Bit-Ganzzahl (vorzeichenbehaftet)) durch und prüft, ob der erste Wert (IN1) ungleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `SINT`, 8-Bit-Ganzzahl (vorzeichenbehaftet)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 != IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AS` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AS` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AS` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_NE` (Typ `iec61131::comparison::F_NE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AS`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („not equal to“) für den Datentyp `SINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: kleine Zähl- und Indexwerte gegen einen Referenz- oder Schwellwert vom Typ `SINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `SINT` die Bedingung „ungleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `SINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AS_EQ**: Prüft auf „gleich“ – invers zur Funktion von AS_NE.
- **AS_EQ**: Prüft auf Gleichheit der Werte.

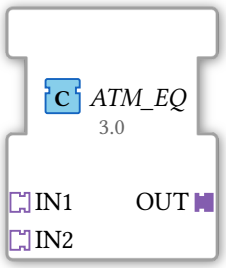
Der AS_NE ist Teil der vollständigen AS-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `SINT`.

Fazit

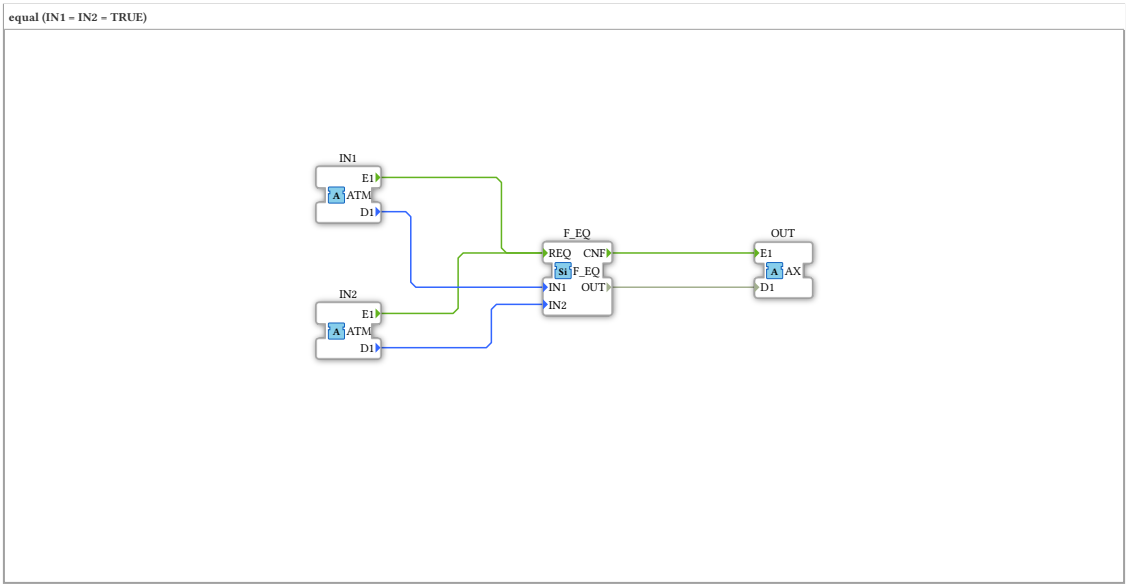
Der **AS_NE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `SINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

ATM_EQ

Input value 1 -- left operand (IN1 = IN2) – *ATM*
Input value 2 -- right operand (IN1 = IN2) – *ATM*



AX – IN1 = IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **ATM_EQ** führt einen Vergleich zweier Werte vom Typ `TIME` (Zeitdauer) durch und prüft, ob der erste Wert (IN1) gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Verglichen wird die Dauer der beiden Zeitwerte.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `TIME`, Zeitdauer).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 = IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`ATM` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`ATM` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `ATM` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_EQ` (Typ `iec61131::comparison::F_EQ`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`ATM`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („equal to“) für den Datentyp `TIME`.
- **Vergleichslogik**: Verglichen wird die Dauer der beiden Zeitwerte.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Zeit- und Verzögerungswerte gegen einen Referenz- oder Schwellwert vom Typ `TIME` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `TIME` die Bedingung „gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `TIME`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

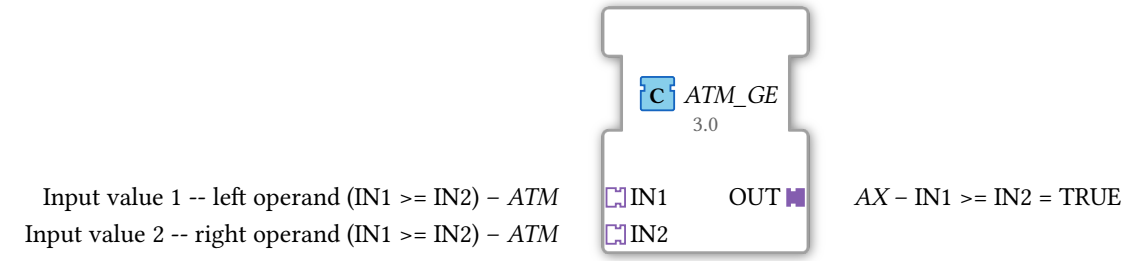
- **ATM_NE**: Prüft auf „ungleich“ – invers zur Funktion von ATM_EQ.
- **ATM_NE**: Prüft auf Ungleichheit der Werte.

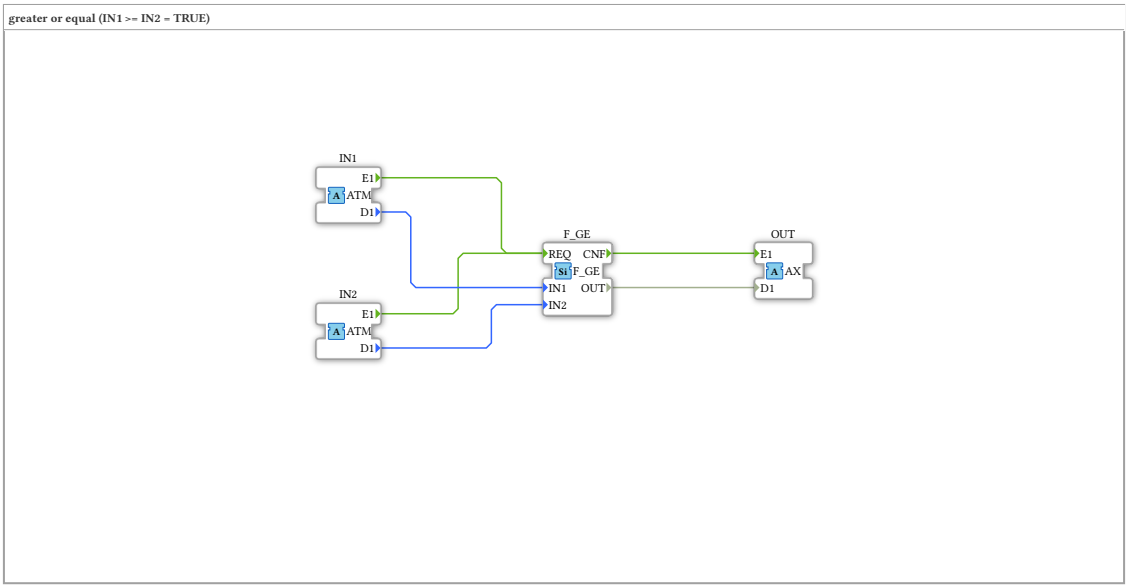
Der ATM_EQ ist Teil der vollständigen ATM-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `TIME`.

Fazit

Der **ATM_EQ** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `TIME`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

ATM_GE





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **ATM_GE** führt einen Vergleich zweier Werte vom Typ `TIME` (Zeitdauer) durch und prüft, ob der erste Wert (IN1) größer gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Verglichen wird die Dauer der beiden Zeitwerte.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `TIME`, Zeitdauer).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 >= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`ATM` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`ATM` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `ATM` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GE` (Typ `iec61131::comparison::F_GE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`ATM`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than or equal to“) für den Datentyp `TIME`.
- **Vergleichslogik**: Verglichen wird die Dauer der beiden Zeitwerte.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Zeit- und Verzögerungswerte gegen einen Referenz- oder Schwellwert vom Typ `TIME` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `TIME` die Bedingung „größer gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `TIME`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

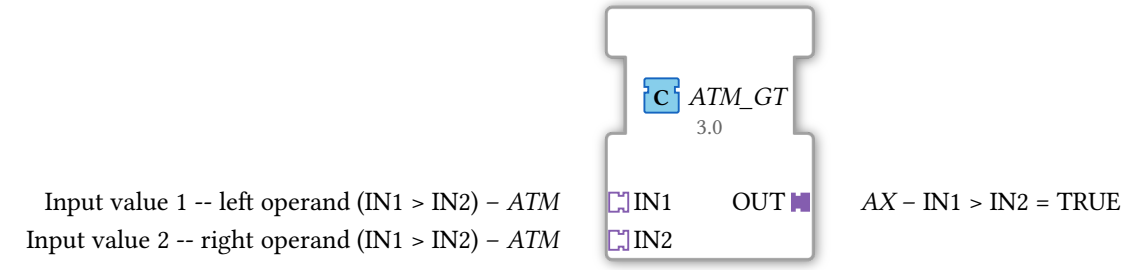
- **ATM_LE**: Prüft auf „kleiner gleich“ – invers zur Funktion von ATM_GE.
- **ATM_EQ**: Prüft auf Gleichheit der Werte.
- **ATM_GT**: Führt den Vergleich „größer als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version ATM_GE ist.

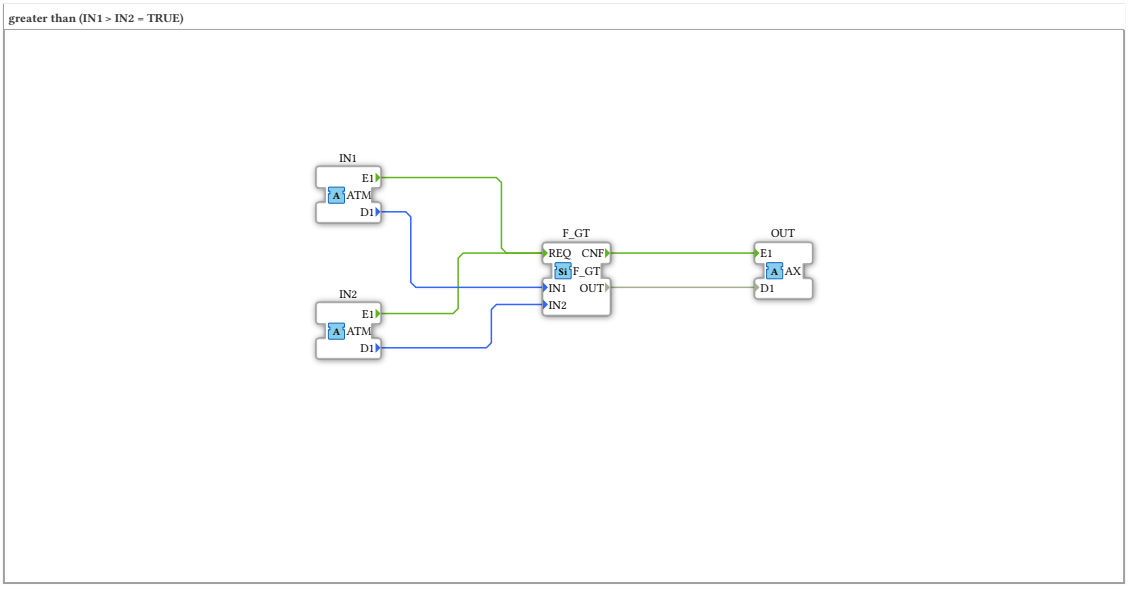
Der ATM_GE ergänzt die ATM-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

Der **ATM_GE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `TIME`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

ATM_GT





• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **ATM_GT** führt einen Vergleich zweier Werte vom Typ `TIME` (Zeitdauer) durch und prüft, ob der erste Wert (IN1) größer als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Verglichen wird die Dauer der beiden Zeitwerte.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `TIME`, Zeitdauer).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 > IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`ATM` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`ATM` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `ATM` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GT` (Typ `iec61131::comparison::F_GT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`ATM`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than“) für den Datentyp `TIME`.
- **Vergleichslogik**: Verglichen wird die Dauer der beiden Zeitwerte.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Zeit- und Verzögerungswerte gegen einen Referenz- oder Schwellwert vom Typ `TIME` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `TIME` die Bedingung „größer als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `TIME`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **ATM_LT**: Prüft auf „kleiner als“ – invers zur Funktion von ATM_GT.
- **ATM_EQ**: Prüft auf Gleichheit der Werte.
- **ATM_GE**: Führt den Vergleich „größer gleich“ durch – die um den Gleichheitsfall erweiterte Version von ATM_GT.

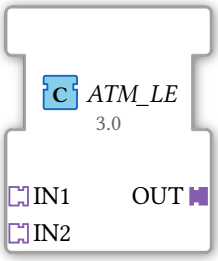
Der ATM_GT ist die grundlegende „größer als“-Komponente ohne Gleichheitsfall, ergänzt durch ATM_GE.

Fazit

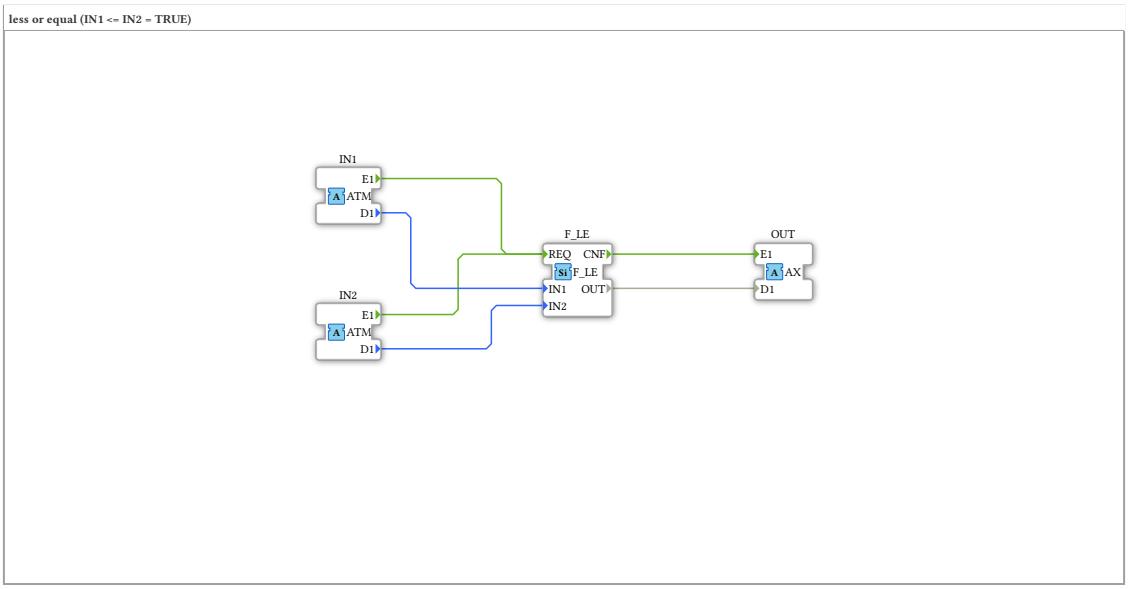
Der **ATM_GT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `TIME`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

ATM_LE

Input value 1 -- left operand (IN1 <= IN2) – *ATM*
Input value 2 -- right operand (IN1 <= IN2) – *ATM*



AX – IN1 <= IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **ATM_LE** führt einen Vergleich zweier Werte vom Typ `TIME` (Zeitdauer) durch und prüft, ob der erste Wert (IN1) kleiner gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Verglichen wird die Dauer der beiden Zeitwerte.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `TIME`, Zeitdauer).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 <= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`ATM` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`ATM` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `ATM` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LE` (Typ `iec61131::comparison::F_LE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`ATM`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than or equal to“) für den Datentyp `TIME`.
- **Vergleichslogik**: Verglichen wird die Dauer der beiden Zeitwerte.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Zeit- und Verzögerungswerte gegen einen Referenz- oder Schwellwert vom Typ `TIME` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `TIME` die Bedingung „kleiner gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `TIME`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **ATM_GE**: Prüft auf „größer gleich“ – invers zur Funktion von ATM_LE.
- **ATM_EQ**: Prüft auf Gleichheit der Werte.
- **ATM_LT**: Führt den Vergleich „kleiner als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version ATM_LE ist.

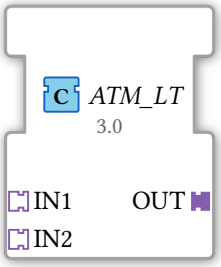
Der ATM_LE ergänzt die ATM-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

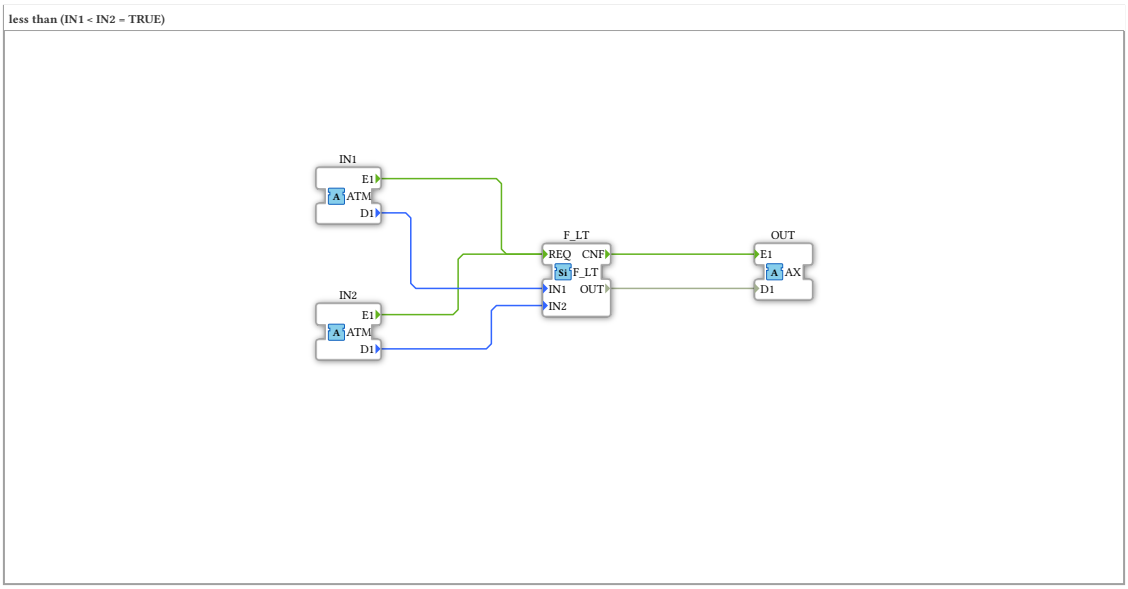
Der **ATM_LE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `TIME`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

ATM_LT

Input value 1 -- left operand (IN1 < IN2) – *ATM*
Input value 2 -- right operand (IN1 < IN2) – *ATM*



AX – IN1 < IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **ATM_LT** führt einen Vergleich zweier Werte vom Typ `TIME` (Zeitdauer) durch und prüft, ob der erste Wert (IN1) kleiner als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Verglichen wird die Dauer der beiden Zeitwerte.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.

- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `TIME`, Zeitdauer).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 < IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`ATM` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`ATM` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `ATM` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LT` (Typ `iec61131::comparison::F_LT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`ATM`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than“) für den Datentyp `TIME`.
- **Vergleichslogik**: Verglichen wird die Dauer der beiden Zeitwerte.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Zeit- und Verzögerungswerte gegen einen Referenz- oder Schwellwert vom Typ `TIME` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `TIME` die Bedingung „kleiner als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `TIME`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **ATM_GT**: Prüft auf „größer als“ – invers zur Funktion von ATM_LT.
- **ATM_EQ**: Prüft auf Gleichheit der Werte.
- **ATM_LE**: Führt den Vergleich „kleiner gleich“ durch – die um den Gleichheitsfall erweiterte Version von ATM_LT.

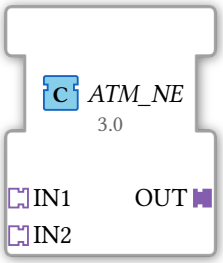
Der ATM_LT ist die grundlegende „kleiner als“-Komponente ohne Gleichheitsfall, ergänzt durch ATM_LE.

Fazit

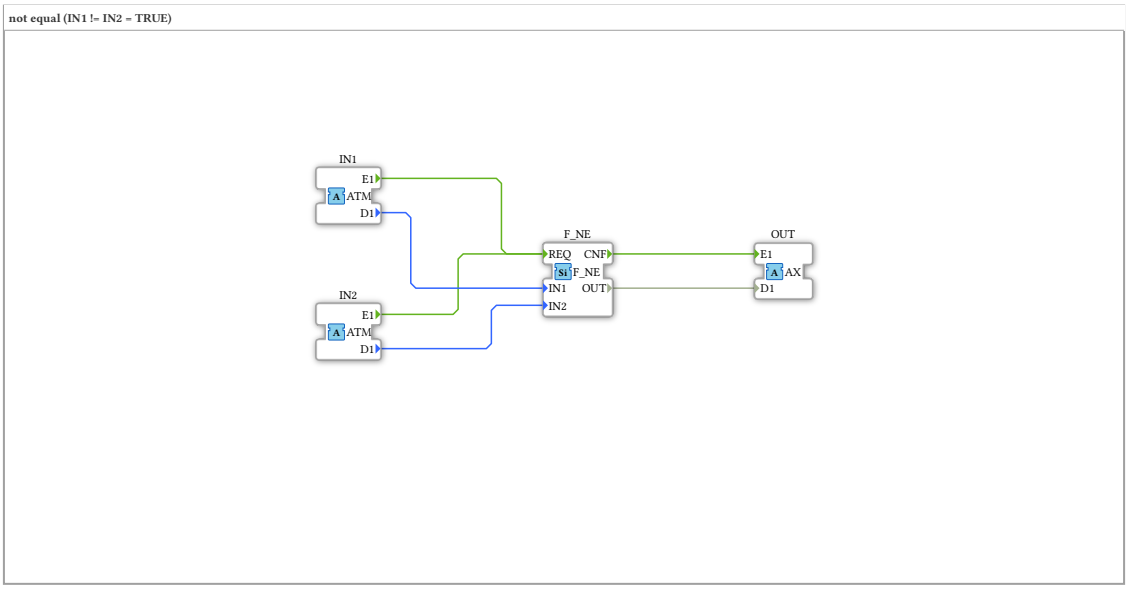
Der **ATM_LT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `TIME`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

ATM_NE

Input value 1 -- left operand (IN1 != IN2) – *ATM*
Input value 2 -- right operand (IN1 != IN2) – *ATM*



AX – IN1 != IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **ATM_NE** führt einen Vergleich zweier Werte vom Typ `TIME` (Zeitdauer) durch und prüft, ob der erste Wert (IN1) ungleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Verglichen wird die Dauer der beiden Zeitwerte.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `TIME`, Zeitdauer).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 != IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`ATM` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`ATM` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `ATM` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_NE` (Typ `iec61131::comparison::F_NE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`ATM`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („not equal to“) für den Datentyp `TIME`.
- **Vergleichslogik**: Verglichen wird die Dauer der beiden Zeitwerte.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Zeit- und Verzögerungswerte gegen einen Referenz- oder Schwellwert vom Typ `TIME` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `TIME` die Bedingung „ungleich“ erfüllen.

- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `TIME`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **ATM_EQ**: Prüft auf „gleich“ – invers zur Funktion von ATM_NE.
- **ATM_EQ**: Prüft auf Gleichheit der Werte.

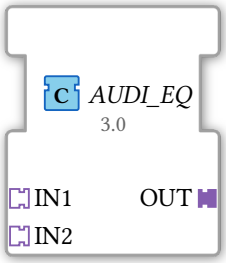
Der ATM_NE ist Teil der vollständigen ATM-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `TIME`.

Fazit

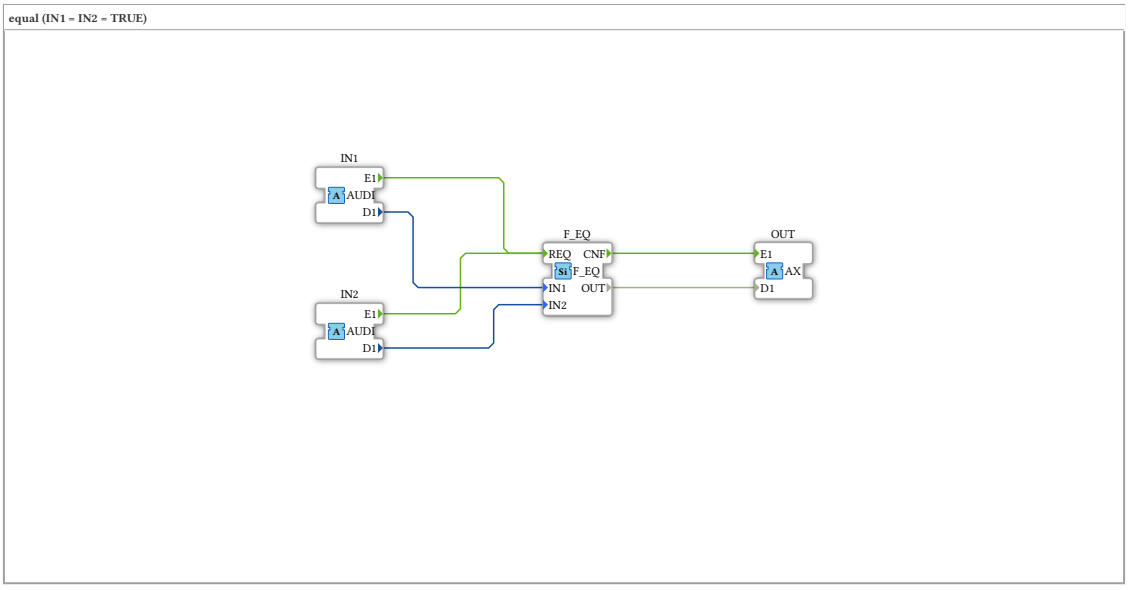
Der **ATM_NE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `TIME`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AUDI_EQ

Input value 1 -- left operand (IN1 = IN2) – *AUDI*
Input value 2 -- right operand (IN1 = IN2) – *AUDI*



AX – IN1 = IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUDI_EQ** dient dem Vergleich zweier analogen Werte auf Gleichheit. Er ist als Composite-FB (zusammengesetzter Baustein) realisiert und verwendet Adapter zur Kommunikation mit der Umgebung. Das Ergebnis ist ein boolescher Wert, der über einen weiteren Adapter ausgegeben wird. Der Baustein entspricht der IEC 61131-3 Klassifikation „Standard comparison function“.

Schnittstellenstruktur

Der FB besitzt ausschließlich Adapter-Schnittstellen. Es sind keine direkten Ereignis- oder Datenein-/ausgänge definiert. Die eigentlichen Ein-/Ausgänge werden über die Adaptertypen realisiert.

Ereignis-Eingänge

- **IN1.E1** (über Adapter `IN1`): Ereigniseingang des ersten zu vergleichenden Wertes.
- **IN2.E1** (über Adapter `IN2`): Ereigniseingang des zweiten zu vergleichenden Wertes.

Ereignis-Ausgänge

- **OUT.E1** (über Adapter `OUT`): Ereignisausgang – wird aktiviert, sobald der Vergleich abgeschlossen ist und das Ergebnis gültig ist.

Daten-Eingänge

- **IN1.D1** (über Adapter `IN1`): Erster Vergleichswert (analog).
- **IN2.D1** (über Adapter `IN2`): Zweiter Vergleichswert (analog).

Daten-Ausgänge

- **OUT.D1** (über Adapter `OUT`): Boolesches Ergebnis des Vergleichs (`TRUE` wenn beide Werte gleich sind, sonst `FALSE`).

Adapter

Adapter	Richtung	Typ	Kurzbeschreibung
`IN1`	Socket	`adapter::types::unidirectional::AUDI`	Liefert den ersten analogen Wert inkl. Ereignis.
`IN2`	Socket	`adapter::types::unidirectional::AUDI`	Liefert den zweiten analogen Wert inkl. Ereignis.
`OUT`	Plug	`adapter::types::unidirectional::AX`	Gibt das Vergleichsergebnis (boolesch) mit Ereignis aus.

Funktionsweise

Der FB **AUDI_EQ** enthält intern einen bereits vorhandenen Funktionsblock `iec61131::comparison::F_EQ` zur reinen Gleichheitsprüfung.

- Sobald an einem der beiden Eingangsadapter (`IN1` oder `IN2`) ein Ereignis eintrifft, wird der interne FB `F_EQ` aufgerufen.
- Dabei werden die aktuellen Datenwerte von `IN1.D1` und `IN2.D1` an `F_EQ` übergeben.
- Der interne FB vergleicht beide Werte und liefert ein boolesches Ergebnis.
- Nach Abschluss der Verarbeitung gibt der interne FB ein Ereignis aus, das über den Ausgangsadapter `OUT` weitergeleitet wird.
- Gleichzeitig wird das Vergleichsergebnis am Datenausgang `OUT.D1` bereitgestellt.

Der Baustein arbeitet rein **kombinatorisch** – er speichert keine Zustände und führt die Vergleichslogik bei jeder Aktivierung neu aus.

Technische Besonderheiten

- **Composite-FB**: Der Baustein ist als Netzwerk aus Sub-FBs realisiert, was eine modulare und wiederverwendbare Struktur ermöglicht.
- **Adapterbasierte Schnittstelle**: Durch die Verwendung der unidirektionalen Adapter `AUDI` und `AX` wird eine flexible Kapselung von Ereignis- und Datenkanälen erreicht. Die Adapter definieren die tatsächlichen Ein- und Ausgänge.
- **Standardkomponente**: Der interne FB `F_EQ` ist Teil der IEC 61131-3 Bibliothek und garantiert eine plattformunabhängige Vergleichslogik.
- **Keine eigene Zustandsmaschine**: Der Baustein besitzt kein internes Zustandsdiagramm, sondern delegiert die Logik vollständig an den eingebetteten FB.

Zustandsübersicht

Der FB **AUDI_EQ** verfügt über **keine eigenständigen Zustände**. Die interne Logik ist rein ereignisgesteuert und resultiert unmittelbar in der Ausgabe. Eine Zustandsübersicht entfällt daher.

Anwendungsszenarien

- **Überwachung von Prozessgrößen**: Zwei analoge Werte (z. B. Drucksensoren, Temperaturfühler) auf Gleichheit prüfen.
- **Redundanzprüfung**: Vergleich zweier unabhängig gemessener Werte, um Sensorausfälle zu erkennen.
- **Signalabgleich**: In Steuerungssystemen, in denen sichergestellt werden muss, dass zwei Eingangssignale den gleichen Wert aufweisen, bevor eine Aktion ausgelöst wird.

Vergleich mit ähnlichen Bausteinen

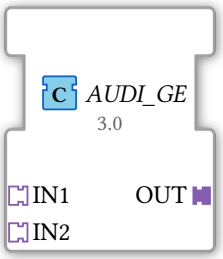
- **`F_EQ` (direkt genutzt)** – Der interne FB `F_EQ` bietet dieselbe Vergleichslogik, jedoch ohne Adapter. Er erfordert explizite Ereignis- und Dateneingänge.
- **`F_NE` (Ungleichheit)** – Führt einen Ungleichheitsvergleich durch; strukturell ähnlich, aber Ergebnis invertiert.
- **`AUDI_GT` (Größer als)** – Analoger Aufbau mit Adaptern, aber anderer Vergleichsoperator. Der **AUDI_EQ** unterscheidet sich durch die vorgefertigte Adapter-Schnittstelle, die eine einfache Integration in bestehende Adapter-Netzwerke ermöglicht. Er ist speziell für Anwendungen konzipiert, die eine standardisierte Ein-/Ausgabe über unidirektionale Adapter erfordern.

Fazit

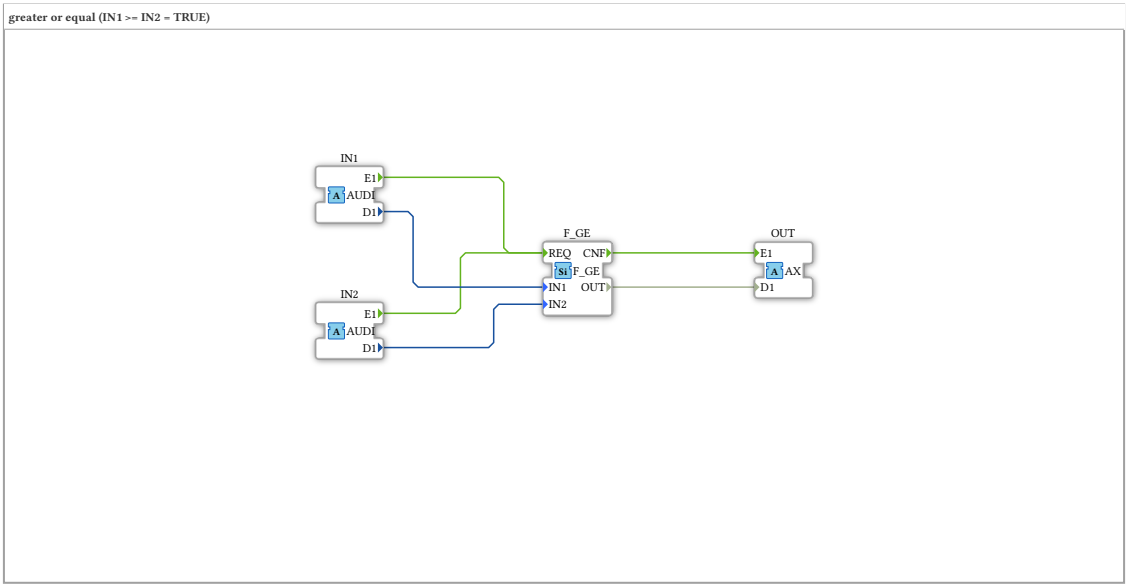
Der **AUDI_EQ** ist ein einfach zu verwendender zusammengesetzter Funktionsblock für Gleichheitsvergleiche mit zwei analogen Werten. Durch die adapterbasierte Schnittstelle fügt er sich nahtlos in modulare Steuerungssysteme ein. Die interne Verwendung der standardisierten IEC-Bausteine gewährleistet eine zuverlässige und nach IEC 61131-3 definierte Funktionalität. Der Baustein eignet sich besonders für Anwendungen, die eine klare Trennung von Ereignis- und Datenflüssen über Adapter erfordern.

AUDI_GE

Input value 1 -- left operand (IN1 >= IN2) – *AUDI*
Input value 2 -- right operand (IN1 >= IN2) – *AUDI*



AX – IN1 >= IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AUDI_GE führt einen Vergleich zweier Werte auf „größer oder gleich“ durch. Er stellt das Ergebnis als booleschen Wert bereit. Der Baustein ist als Komposition realisiert und nutzt intern den Standard-Vergleichsbaustein F_GE aus der IEC 61131-3-Bibliothek. Die Ein- und Ausgabe erfolgt ausschließlich über unidirektionale Adapter.

Schnittstellenstruktur

Ereignis-Eingänge

- **Keine direkten Ereignisseingänge** Die Auslösung des Vergleichs erfolgt über die **Socket-Adapter** IN1 und IN2. Jeder dieser Adapter liefert ein Ereignis an den internen Baustein, sobald gültige Daten anliegen.

Ereignis-Ausgänge

- **Keine direkten Ereignisausgänge** Das Ergebnis des Vergleichs wird über den **Plug-Adapter** OUT als Ereignis weitergereicht, sobald die Auswertung abgeschlossen ist.

Daten-Eingänge

Die Datenwerte, die verglichen werden sollen, werden über die Adapter bezogen:

- **Adapter IN1** – liefert den ersten Vergleichswert (Typ entsprechend dem Adapter AUDI).
- **Adapter IN2** – liefert den zweiten Vergleichswert (Typ entsprechend dem Adapter AUDI).

Daten-Ausgänge

- **Adapter OUT** – gibt den booleschen Wert TRUE aus, wenn IN1 ≥ IN2 ist, andernfalls FALSE (Typ entsprechend dem Adapter AX).

Adapter

Adapter	Richtung	Typ	Beschreibung
IN1	Socket	adapter::types::unidirectional::AUDI	Erster Eingabewert
IN2	Socket	adapter::types::unidirectional::AUDI	Zweiter Eingabewert
OUT	Plug	adapter::types::unidirectional::AX	Ausgabe: IN1 >= IN2 (boolesch)

Funktionsweise

1. Die Ereignisse von IN1.E1 und IN2.E1 werden zum internen F_GE-Baustein geführt (über eine ODER-Verknüpfung – beide Ereignisse triggern denselben REQ-Eingang).
2. Gleichzeitig werden die Daten IN1.D1 und IN2.D1 an die entsprechenden Eingänge F_GE.IN1 und F_GE.IN2 übergeben.
3. Der interne Baustein F_GE berechnet IN1 ≥ IN2.
4. Das Ergebnis erscheint am Ausgang F_GE.OUT und wird an den Adapter OUT.D1 weitergegeben.
5. Mit dem Ereignis F_GE.CNF wird OUT.E1 ausgelöst, um der Umgebung zu signalisieren, dass ein neues Ergebnis vorliegt.

Technische Besonderheiten

- Der Baustein ist vollständig adapterbasiert – er besitzt keine klassischen Ereignis- oder Dateneingänge auf der obersten Ebene.
- Die Adapter AUDI und AX sind unidirektional und müssen im entsprechenden Projekt definiert sein.
- Die Ausführung erfolgt asynchron: Sobald einer der beiden Eingangsadapter ein Ereignis liefert, wird der Vergleich neu berechnet. Eine Synchronisation der Werte ist nicht erforderlich, da beide Daten gleichzeitig anliegen müssen, damit das Ergebnis sinnvoll ist.
- Die Bausteinkonfiguration entspricht der IEC 61131-3-Norm und der Eclipse 4diac-Umgebung.

Zustandsübersicht

Der Baustein selbst besitzt keinen expliziten Zustandsautomaten – die gesamte Steuerlogik liegt im internen F_GE. Sein Verhalten ist deterministisch:

- **Warten auf Trigger**: Solange kein Ereignis an `IN1` oder `IN2` eintrifft, bleibt der Ausgang `OUT` unverändert.
- **Berechnung**: Bei Eintreffen eines Ereignisses wird der Vergleich ausgeführt.
- **Ergebnisausgabe**: Das Ergebnis wird am Ausgang bereitgestellt und ein Ereignis am `OUT`-Adapter gesendet.

Anwendungsszenarien

- **Zustandsüberwachung**: Prüfen, ob ein Messwert einen Schwellwert erreicht oder überschreitet.
- **Aktorsteuerung**: Aktivieren eines Ausgangs, sobald eine Bedingung `Wert1 ≥ Wert2` erfüllt ist.
- **Wertvergleich in Produktionsanlagen**: Vergleichen von Soll-/Ist-Werten in kontinuierlichen Prozessen.
- **Einfache logische Verknüpfungen**: Als Bestandteil komplexerer Sicherheits- oder Steuerungslogik.

Vergleich mit ähnlichen Bausteinen

- **AUDI_GT** (greater than) – vergleicht auf >, liefert `TRUE` nur bei strikt größerem ersten Wert.
- **AUDI_EQ** (equal) – vergleicht auf Gleichheit.
- **AUDI_LE** (less or equal) – das Gegenstück für ≤.
- **AUDI_GE** ist der direkte Partner zu `AUDI_LE` und deckt den Fall „mindestens gleich“ ab. Im Vergleich zu einem Einzelbaustein wie `F_GE` bietet `AUDI_GE` eine adapterbasierte Kapselung, was die Wiederverwendung in verschiedenen Projekten erleichtert.

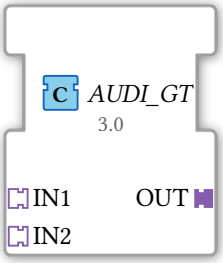
Fazit

`AUDI_GE` ist ein spezialisierter Vergleichsbaustein, der durch die Verwendung von Adaptern eine flexible und saubere Schnittstellendefinition ermöglicht. Er eignet sich besonders für modulare Automatisierungsprojekte, in denen Werte über standardisierte Schnittstellen ausgetauscht werden. Die interne Nutzung des bewährten `F_GE`-Bausteins garantiert korrekte und performante Vergleichsoperationen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

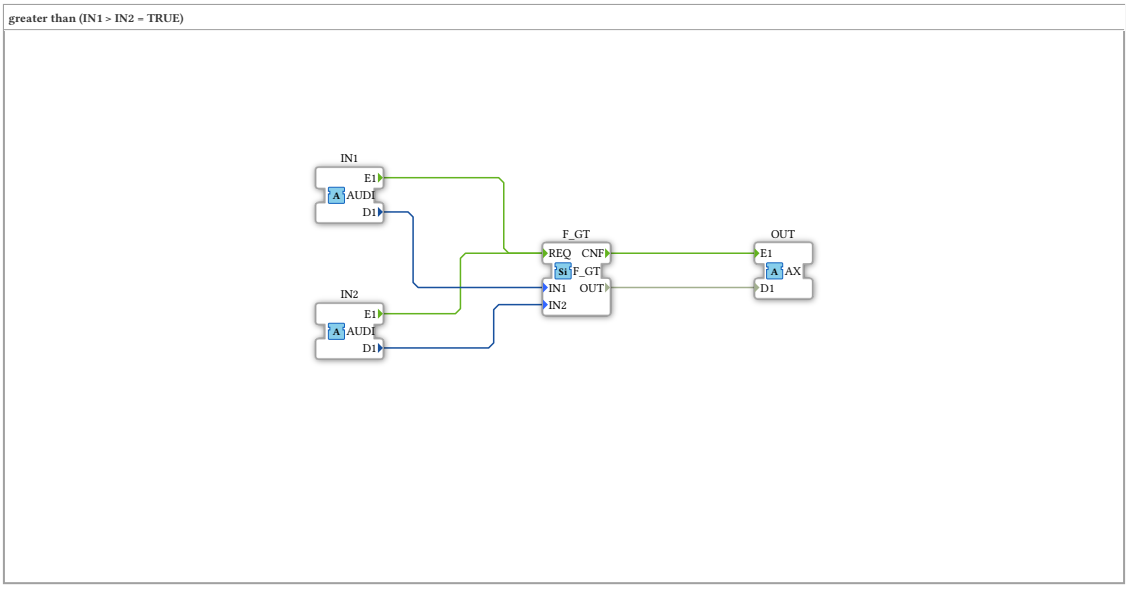
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUDI_GT



Input value 1 -- left operand (IN1 > IN2) – *AUDI*
Input value 2 -- right operand (IN1 > IN2) – *AUDI*

AX – IN1 > IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock AUDI_GT realisiert einen Vergleich zweier Werte auf „Größer als“ (greater than). Er ist als IEC 61131-3 konformer Vergleichsbaustein konzipiert und verwendet ein Adapter-basiertes Interface, um die Werte und Ereignisse zu übertragen. Der Baustein ist Teil der Bibliothek iec61131::comparison.

Schnittstellenstruktur

Ereignis-Eingänge

- **IN1.E1** – Ereigniseingang für den ersten Wert (über Adapter IN1)
- **IN2.E1** – Ereigniseingang für den zweiten Wert (über Adapter IN2) Beide Ereignisse werden auf den internen REQ-Eingang des Bausteins F_GT verbunden.

Ereignis-Ausgänge

- **OUT.E1** – Ereignisausgang, der nach Abschluss des Vergleichs aktiviert wird (über Adapter OUT). Das Ergebnis liegt dann am Datenausgang an.

Daten-Eingänge

- **IN1.D1** – Erster Eingangswert (über Adapter IN1, Typ abhängig vom Adapter AUDI)
- **IN2.D1** – Zweiter Eingangswert (über Adapter IN2, Typ abhängig vom Adapter AUDI)

Daten-Ausgänge

- **OUT.D1** – Ergebnis des Vergleichs: TRUE, wenn IN1 > IN2, sonst FALSE (über Adapter OUT, Typ abhängig vom Adapter AX)

Adapter

- **Socket IN1** (Typ: `adapter::types::unidirectional::AUDI`) – Verbindung für den ersten zu vergleichenden Wert.
- **Socket IN2** (Typ: `adapter::types::unidirectional::AUDI`) – Verbindung für den zweiten zu vergleichenden Wert.
- **Plug OUT** (Typ: `adapter::types::unidirectional::AX`) – Ausgabe des Vergleichsergebnisses.

Die verwendeten Adapter sind unidirektional und stellen je einen Ereignis- und einen Datenport bereit.

Funktionsweise

Der Baustein AUDI_GT kapselt den elementaren Vergleichsbaustein F_GT aus der IEC 61131-3 Bibliothek. Die Eingangsadapter IN1 und IN2 liefern je einen Wert (D1) sowie ein Startsignal (E1). Beide Ereignisse werden auf den REQ-Eingang des F_GT verbunden, sodass der Vergleich ausgelöst wird, sobald mindes-

tens einer der Ereignisse eintritt. Intern führt F_GT den Vergleich der beiden Werte durch. Das Ergebnis (BOOL) wird über den Ausgangsadapter OUT bereitgestellt, wobei gleichzeitig ein Ereignis (E1) generiert wird.

Technische Besonderheiten

- Der Baustein verwendet **Adapter** anstelle von direkten Ein-/Ausgangs-Ports, was eine flexible Kapselung und Wiederverwendung der Schnittstellen ermöglicht.
- Die Ereignisse beider Eingänge werden auf denselben REQ-Eingang des internen F_GT zusammengeführt. Dies bedeutet, dass der Baustein bei jedem eintreffenden Ereignis an IN1 oder IN2 den Vergleich neu durchführt.
- Der Baustein gibt das Ergebnis als BOOL-Wert aus (TRUE/FALSE). Die genauen Datentypen der Adapter (AUDI, AX) sind anwendungsspezifisch und können je nach Kontext variieren.
- Der Baustein ist als Teil der Bibliothek `iec61131::comparison` gekennzeichnet und folgt dem IEC 61131-3 Standard.

Zustandsübersicht

Der Baustein AUDI_GT besitzt keinen eigenen Zustandsautomaten. Die gesamte Logik wird durch den internen F_GT realisiert, der einen einfachen kombinatorischen Vergleich ausführt. Es gibt keine internen Zustände oder Speicherverhalten; das Ergebnis wird bei jeder Aktivierung neu berechnet.

Anwendungsszenarien

- **Prozessüberwachung**: Vergleich von Messwerten (z.B. Temperatur, Druck) mit einem Schwellwert. Wenn Wert1 > Wert2, wird ein Alarm ausgelöst.
- **Logische Entscheidungen**: In Automatisierungsabläufen, z.B. Sortierlogik oder Prioritätssteuerungen.
- **Sicherheitsfunktionen**: Überprüfung, ob ein aktueller Wert einen Grenzwert überschreitet.
- **Adapter-basierte Systeme**: Überall dort, wo modulare, wiederverwendbare Schnittstellen über Adapter gewünscht sind.

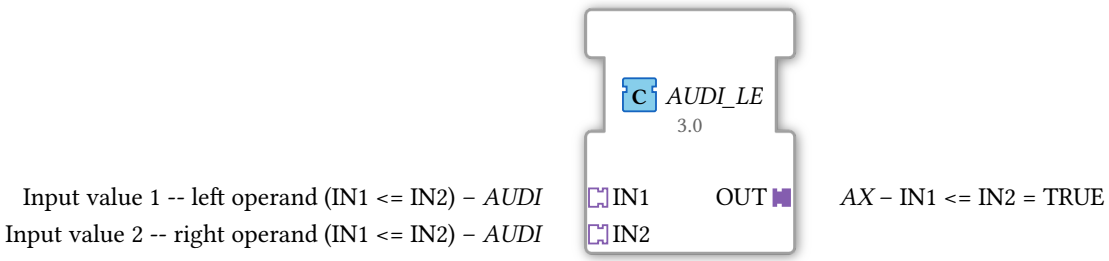
Vergleich mit ähnlichen Bausteinen

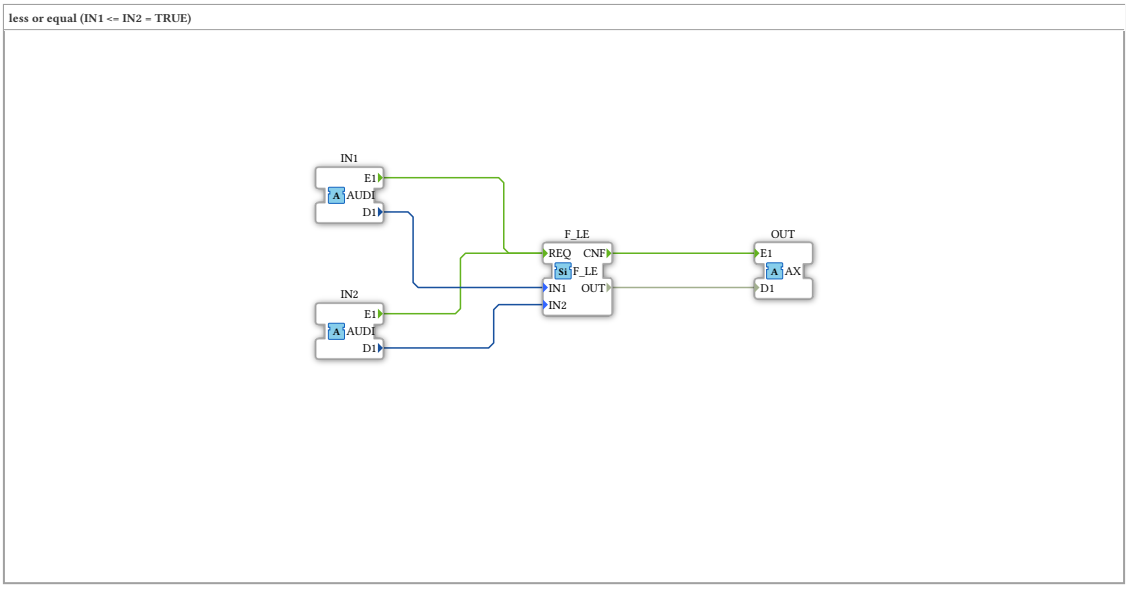
- **GT (Greater Than)**: Einfache Vergleicher wie `F_GT` (IEC 61131-3) haben direkte Eingänge. `AUDI_GT` erweitert dies auf eine adapterbasierte Schnittstelle, was die Integration in heterogene Systeme erleichtert.
- **GE (Greater or Equal)**, **LT (Less Than)**, **EQ (Equal)**: Analoge Bausteine für andere Vergleichsoperationen. Der vorliegende Baustein unterstützt nur „Größer als“.
- **Vergleichskaskaden**: Mehrere AUDI_GT können zu komplexeren Ausdrücken verknüpft werden, indem Ausgänge an Eingänge weiterer Bausteine geschaltet werden.

Fazit

Der AUDI_GT ist ein dedizierter Vergleichsbaustein für die „Größer als“-Funktion mit adaptern basierten Ein- und Ausgängen. Er bietet eine standardisierte, modulare Schnittstelle gemäß IEC 61131-3 und eignet sich besonders für den Einsatz in verteilten Automatisierungssystemen, die eine lose Kopplung der Komponenten erfordern. Dank der Adaptertechnologie kann der Baustein flexibel in verschiedene Kontexte eingebettet werden.

AUDI_LE





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUDI_LE** realisiert einen Vergleich zweier Werte auf „kleiner oder gleich“ (less or equal). Er ist als Composite-FB auf Basis des standardisierten Vergleichsbausteins `F_LE` aus der IEC-61131-Klassifikation aufgebaut. Die Kommunikation mit der Umgebung erfolgt ausschließlich über Adapter-Schnittstellen, sodass der Baustein modular in ereignisgesteuerte Abläufe eingebunden werden kann.

Schnittstellenstruktur

Ereignis-Eingänge

Bezeichnung	Datentyp	Beschreibung
`IN1.E1`	EVENT	Ereignis-Eingang des Adapters IN1 – löst die Verarbeitung des ersten Wertes aus.
`IN2.E1`	EVENT	Ereignis-Eingang des Adapters IN2 – löst die Verarbeitung des zweiten Wertes aus.

Hinweis: Beide Ereignisse müssen eingehen, damit ein Vergleich stattfindet (logische UND-Verknüpfung im internen Netzwerk).

Ereignis-Ausgänge

Bezeichnung	Datentyp	Beschreibung
`OUT.E1`	EVENT	Ereignis-Ausgang des Adapters OUT – wird aktiviert, sobald das Vergleichsergebnis bereitsteht.

Daten-Eingänge

Bezeichnung	Datentyp	Beschreibung
`IN1.D1`	ANY	Erster zu vergleichender Wert (über Adapter IN1).
`IN2.D1`	ANY	Zweiter zu vergleichender Wert (über Adapter IN2).

Daten-Ausgänge

Bezeichnung	Datentyp	Beschreibung
`OUT.D1`	BOOL	Vergleichsergebnis: `TRUE`, wenn `IN1 ≤ IN2`; sonst `FALSE`.

Adapter

Name	Typ	Richtung	Beschreibung
IN1	adapter::types::unidirectional::AUDI	Socket (Eingang)	Aufnahme des ersten Wertes (Ereignis + Daten).
IN2	adapter::types::unidirectional::AUDI	Socket (Eingang)	Aufnahme des zweiten Wertes (Ereignis + Daten).
OUT	adapter::types::unidirectional::AUDI	Plug (Ausgang)	Ausgabe des Vergleichsergebnisses (Ereignis + Daten).

Funktionsweise

1. Der FB erwartet an beiden Eingangsadaptern (`IN1` und `IN2`) je ein Ereignis und einen Datenwert.
2. Intern werden die beiden Ereignisse (über eine implizite UND-Bedingung im Netzwerk) zusammengeführt und der eingebettete Baustein `F_LE` mit den Datenwerten gestartet.
3. `F_LE` führt den Vergleich `IN1 ≤ IN2` durch und legt das Ergebnis (Boolescher Wert) an seinem Datenausgang an.
4. Sobald das Ergebnis vorliegt, wird das Ereignis am Ausgangsadapter `OUT` ausgelöst und der Ergebniswert über `OUT.D1` bereitgestellt.

Technische Besonderheiten

- **Adapterbasierte Schnittstellen:** Der FB besitzt keine klassischen Einzel-Ports, sondern verwendet standardisierte unidirektionale Adapter. Dies erleichtert die Wiederverwendung und die Integration in komplexe Kommunikationsstrukturen.
- **Composite-Aufbau:** Die gesamte Logik wird über einen internen FB (`F_LE`) realisiert. Dadurch ist der Baustein leicht nachvollziehbar und erweiterbar.
- **Ereignissynchronisation:** Beide Eingangsereignisse müssen eintreffen, bevor die Verarbeitung startet. Dies gewährleistet, dass stets aktuelle Wertepaare verglichen werden.

Zustandsübersicht

Der FB besitzt keinen eigenen gerichteten Zustandsautomaten, da er rein kombinatorisch arbeitet. Der interne `F_LE` ist ebenfalls zustandslos. Die einzige relevante Logik ist die UND-Verknüpfung der Ereignisse – nach Erhalt beider Ereignisse wird einmalig das Ergebnis ausgegeben.

Anwendungsszenarien

- **Grenzwertüberwachung:** Prüfen, ob ein Messwert einen Schwellwert nicht überschreitet.
- **Freigabelogik:** Aktivieren einer Aktion nur dann, wenn die aktuelle Anforderung kleiner oder gleich einer maximalen Zulassung ist.
- **Vergleichsketten:** Kombination mehrerer `AUDI_LE`-Bausteine, um komplexe Bedingungen (z. B. `x ≤ y ≤ z`) zu realisieren.

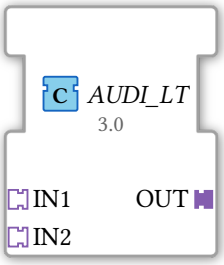
Vergleich mit ähnlichen Bausteinen

FB	Funktion	Unterschied zu AUDI_LE
`AUDI_LT`	Kleiner als (<)	Nicht inklusive Gleichheit.
`AUDI_EQ`	Gleichheit (=)	Prüft nur auf exakte Übereinstimmung.
`AUDI_GE`	Größer oder gleich (≥)	Umgekehrte Vergleichsrichtung.
`AUDI_AND`	Logisches UND	Arbeitet mit Booleschen Werten, nicht mit numerischen Vergleichen.

Fazit

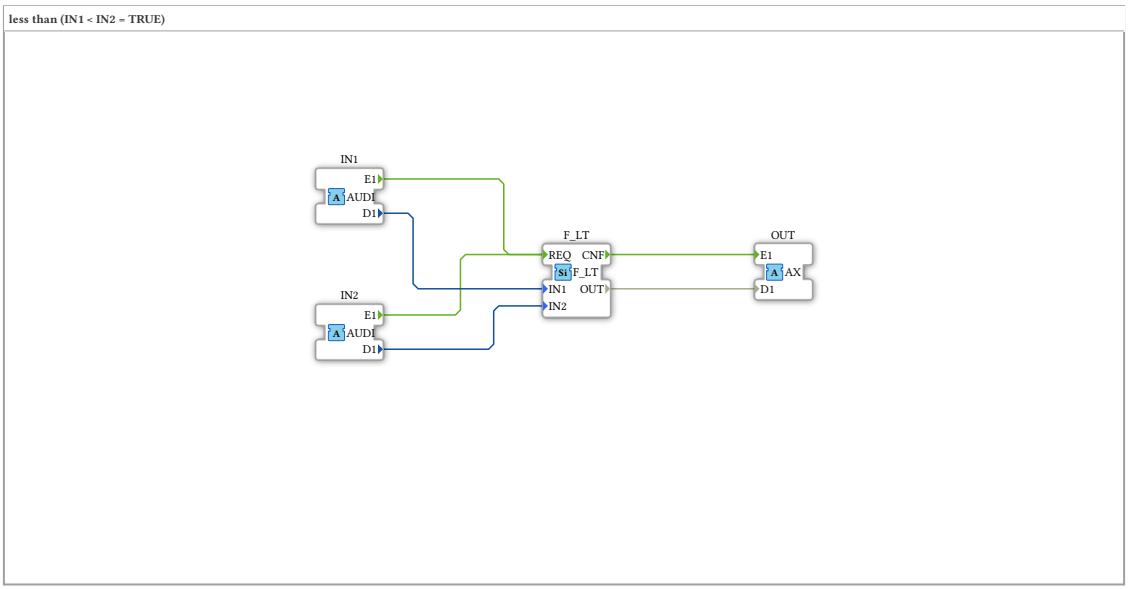
`AUDI_LE` ist ein spezialisierter Vergleichsbaustein, der dank seiner Adapter-Schnittstellen besonders gut in modulare, ereignisgesteuerte Automatisierungslösungen passt. Er bietet eine klare, standardisierte Logik für die „kleiner oder gleich“-Bedingung und lässt sich durch die Composite-Struktur leicht in eigene Applikationen einbinden.

AUDI_LT



Input value 1 -- left operand (IN1 < IN2) – *AUDI*
Input value 2 -- right operand (IN1 < IN2) – *AUDI*

AX – IN1 < IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AUDI_LT` realisiert einen „Kleiner-als“-Vergleich (less than) zwischen zwei Eingangswerten. Er wird in der Automatisierungstechnik häufig eingesetzt, um Schwellwerte zu überwachen oder logische Entscheidungen zu treffen. Der Baustein ist nach IEC 61131-3 als „standard comparison function“ klassifiziert und verwendet Adapter für eine flexible Anbindung unterschiedlicher Datentypen.

Schnittstellenstruktur

Ereignis-Eingänge

Über die Adapter-Sockets werden Ereignisse von außen an den FB weitergeleitet:

- `IN1.E1` – Ereignis, das die Vergleichsoperation anstößt, sobald ein neuer Wert an `IN1` anliegt.
- `IN2.E1` – Ereignis, das ebenfalls die Vergleichsoperation anstößt, sobald ein neuer Wert an `IN2` anliegt.

Ereignis-Ausgänge

- `OUT.E1` – Ereignis, das nach Abschluss des Vergleichs ausgegeben wird und signalisiert, dass das Ergebnis am Datenausgang `OUT.D1` gültig ist.

Daten-Eingänge

- `IN1.D1` – Erster zu vergleichender Wert (Daten vom Typ des Adapters `AUDI`).
- `IN2.D1` – Zweiter zu vergleichender Wert (Daten vom Typ des Adapters `AUDI`).

Daten-Ausgänge

- `OUT.D1` – Ergebnis des Vergleichs: `TRUE`, wenn `IN1 < IN2`, sonst `FALSE` (Daten vom Typ des Adapters `AX`).

Adapter

- `IN1` – Socket (Eingang) vom Typ `adapter::types::unidirectional::AUDI`. Liefert ein Ereignis (`E1`) und einen Datenwert (`D1`) an den FB.
- `IN2` – Socket (Eingang) vom gleichen Typ `AUDI`. Stellt den zweiten Eingangswert bereit.
- `OUT` – Plug (Ausgang) vom Typ `adapter::types::unidirectional::AX`. Übernimmt das Ergebnisereignis (`E1`) und den Ergebnis-Datenwert (`D1`) zur Weitergabe an nachfolgende Bausteine.

Funktionsweise

Der Baustein führt einen numerischen Vergleich der beiden an `IN1` und `IN2` anliegenden Daten durch. Sobald eines der beiden Ereignisse (`IN1.E1` oder `IN2.E1`) eintrifft, wird der interne Funktionsblock `F_LT` (Less Than) aufgerufen. Dieser liest die aktuellen Werte von `IN1` und `IN2` und berechnet deren Größenverhältnis. Das Ergebnis (`TRUE` / `FALSE`) wird am Datenausgang `OUT.D1` bereitgestellt und gleichzeitig das Ereignis `OUT.E1` ausgelöst. Da beide Ereigniseingänge mit dem selben `REQ`-Eingang des internen Bausteins verbunden sind, wird der Vergleich bei jeder Änderung eines der beiden Werte erneut durchgeführt.

Technische Besonderheiten

- **Adapterbasierte Schnittstellen** – Die Verwendung der Adapter `AUDI` und `AX` ermöglicht eine universelle Kopplung an verschiedene Datentypen, ohne dass der FB selbst typisiert werden muss. Die eigentliche Typanpassung erfolgt in den Adapterdefinitionen.
- **Asynchrone Auslösung** – Der Vergleich wird immer dann gestartet, wenn ein Ereignis an `IN1` oder `IN2` eintrifft. Dadurch reagiert der FB sofort auf Änderungen eines der beiden Eingänge.
- **Zustandslosigkeit** – Der FB besitzt keinen internen Zustand; das Ausgangssignal ist ausschließlich von den aktuellen Eingangswerten abhängig.

Zustandsübersicht

Der Baustein ist rein funktional und benötigt keinen eigenen Zustandsautomaten. Er führt nach jedem Ereignis eine einmalige Vergleichsoperation durch und kehrt anschließend in einen leeren Wartezustand zurück.

Anwendungsszenarien

- **Grenzwertüberwachung** – z. B. Prüfen, ob eine Temperatur unter einen Schwellwert fällt.
- **Bereichsprüfung** – Kombination mit weiteren Vergleichsbausteinen (z. B. `AUDI_GT`) zur Realisierung von „innerhalb“ oder „außerhalb“-Logik.
- **Steuerungslogik** – Boolsche Entscheidungen in Abhängigkeit von zwei analogen oder digitalen Größen.

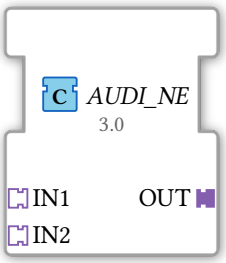
Vergleich mit ähnlichen Bausteinen

- **`AUDI_EQ`** – Prüft auf Gleichheit (`IN1 = IN2`).
- **`AUDI_GT`** – Prüft auf Größer (`IN1 > IN2`).
- **`AUDI_LT`** – Prüft auf Kleiner (`IN1 < IN2`). Alle drei Bausteine haben identische Schnittstellen und unterscheiden sich nur in der Vergleichsoperation. Sie können leicht ausgetauscht werden, um die Logik anzupassen.

Fazit

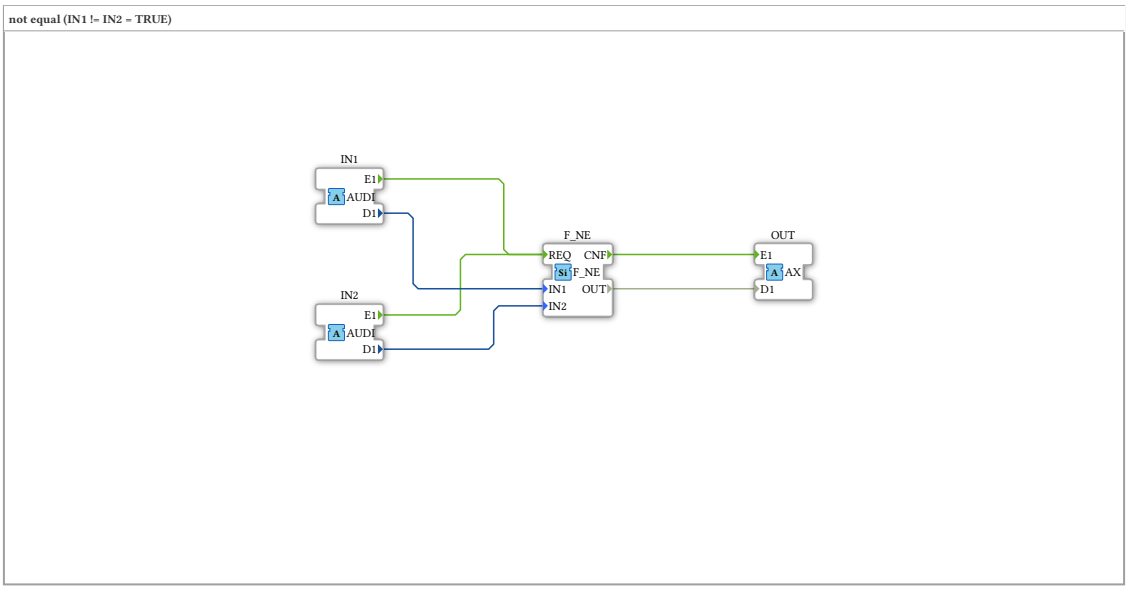
Der `AUDI_LT` ist ein einfacher, aber robuster Vergleichsbaustein, der sich aufgrund seiner Adapter-Schnittstellen flexibel in unterschiedlichste Automatisierungsumgebungen einbinden lässt. Seine asynchrone Arbeitsweise und die klare Trennung von Ereignis und Daten machen ihn zu einem grundlegenden Werkzeug für wertvergleichende Aufgaben in der Steuerungstechnik.

AUDI_NE



Input value 1 -- left operand (IN1 != IN2) – *AUDI*
Input value 2 -- right operand (IN1 != IN2) – *AUDI*

AX – IN1 != IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUDI_NE** realisiert eine Ungleichheitsprüfung („not equal“). Er vergleicht zwei über Adapter bereitgestellte Eingangswerte und gibt das Vergleichsergebnis als booleschen Wert über einen Ausgangsadapter aus. Der Baustein wird vorwiegend in Automatisierungssystemen eingesetzt, die eine adapterbasierte, ereignisgesteuerte Kommunikation unterstützen.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine separaten Ereignis-Eingänge am Interface selbst. Die Ereignisse werden über die Adapter-Sockets **IN1** und **IN2** übertragen. Jeder dieser Sockets enthält einen Ereignis-Eingang (E1), dessen Eintreffen den internen Vergleich anstößt.

Ereignis-Ausgänge

Der Ausgangsadapter **OUT** stellt einen Ereignis-Ausgang (E1) bereit. Dieser signalisiert den Abschluss des Vergleichs und zeigt an, dass das Ergebnis am Datenausgang gültig ist.

Daten-Eingänge

- **IN1.D1** – Erster zu vergleichender Wert (Datentyp abhängig von der Adapterdefinition `adapter::types::unidirectional::AUDI`).
- **IN2.D1** – Zweiter zu vergleichender Wert (gleicher Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1** – Boolesches Ergebnis der Ungleichheitsprüfung: `TRUE` wenn IN1 != IN2, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`adapter::types::unidirectional::AUDI`	Socket	Aufnahme des ersten Eingangswertes
IN2	`adapter::types::unidirectional::AUDI`	Socket	Aufnahme des zweiten Eingangswertes
OUT	`adapter::types::unidirectional::AX`	Plug	Ausgabe des Vergleichsergebnisses

Funktionsweise

Der **AUDI_NE** ist ein zusammengesetzter Funktionsblock. Intern wird der standardisierte Vergleichsbaustein **F_NE (Not Equal)** aus der IEC 61131-Bibliothek verwendet.

- Trifft an **IN1.E1** oder **IN2.E1** ein Ereignis ein, wird das Signal an den internen Baustein **F_NE.REQ** weitergeleitet.
- Gleichzeitig werden die aktuellen Datenwerte von **IN1.D1** und **IN2.D1** an **F_NE.IN1** und **F_NE.IN2** übertragen.
- **F_NE** führt den Ungleich-Vergleich durch und gibt das Ergebnis über **F_NE.OUT** sowie ein Bestätigungsereignis über **F_NE.CNF** aus.
- Das Bestätigungsereignis wird an **OUT.E1** weitergegeben, das Ergebnis an **OUT.D1**.

Der FB arbeitet somit ereignisgesteuert: Jedes Ereignis an einem der beiden Eingänge löst einen neuen Vergleich aus, die Ausgabe wird sofort aktualisiert.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Statt diskreter Ein-/Ausgänge werden Adapter verwendet. Dies ermöglicht eine typsichere und modulare Verbindung mit anderen Bausteinen, die dasselbe Adapterprotokoll nutzen.
- **Zwei unabhängige Ereignisquellen**: Sowohl IN1 als auch IN2 können ein Ereignis liefern. Das interne Verhalten ist so ausgelegt, dass nach jedem Ereignis – unabhängig von der Quelle – ein neuer Vergleich gestartet wird.
- **Wiederverwendung standardisierter Logik**: Die eigentliche Vergleichsfunktion wird durch den eingebetteten FB **F_NE** realisiert, sodass eine hohe Zuverlässigkeit und Konformität zu IEC 61131 gewährleistet ist.

Zustandsübersicht

Der FB besitzt keine expliziten Zustände (kein ECC). Es handelt sich um einen kombinatorischen Baustein ohne Gedächtniseffekt. Die Ausgabe wird rein durch die aktuellen Eingangswerte und das auslösende Ereignis bestimmt.

Anwendungsszenarien

- **Überwachung von Abweichungen**: Prüfung, ob zwei Messwerte (z. B. Soll- und Istwerte) voneinander abweichen. Das Ergebnis kann zur Alarmierung oder Regelung verwendet werden.
- **Redundanzprüfung**: In sicherheitskritischen Systemen kann die Ungleichheit zweier redundanter Sensoren auf einen Fehler hinweisen.
- **Zustandsvergleiche**: Vergleich von Binärzuständen oder Zählwerten, um Änderungen oder Inkonsistenzen zu erkennen.

Vergleich mit ähnlichen Bausteinen

- **AUDI_EQ (Equal)**: Der Gegenpart für Gleichheitsprüfung. Gleiche Schnittstelle, aber invertierte Logik: Ausgabe `TRUE` bei Gleichheit.
- **Direkter Einsatz von F_NE**: Ohne den Adapter-Wrapper könnte man F_NE direkt verwenden. **AUDI_NE** bietet jedoch eine standardisierte Adapter-Schnittstelle, die in Frameworks wie 4diac die Integration und Typensicherheit verbessert.

Fazit

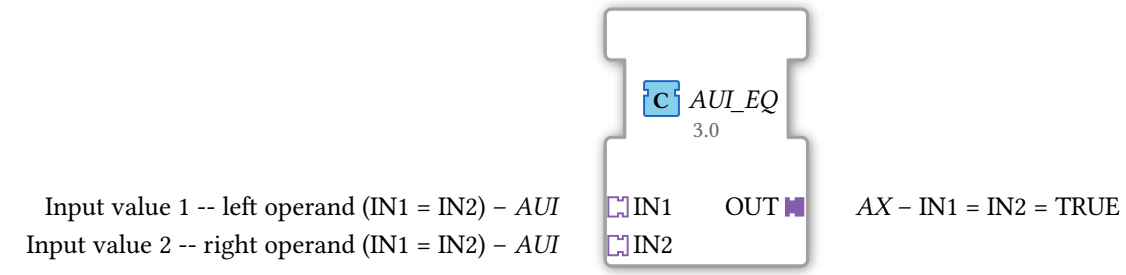
Der **AUDI_NE** ist ein praxisnaher Baustein für den Ungleich-Vergleich in adapterbasierten IEC 61131-Systemen. Durch die Verwendung von Adaptern und die Einbettung des bewährten **F_NE**-Bausteins

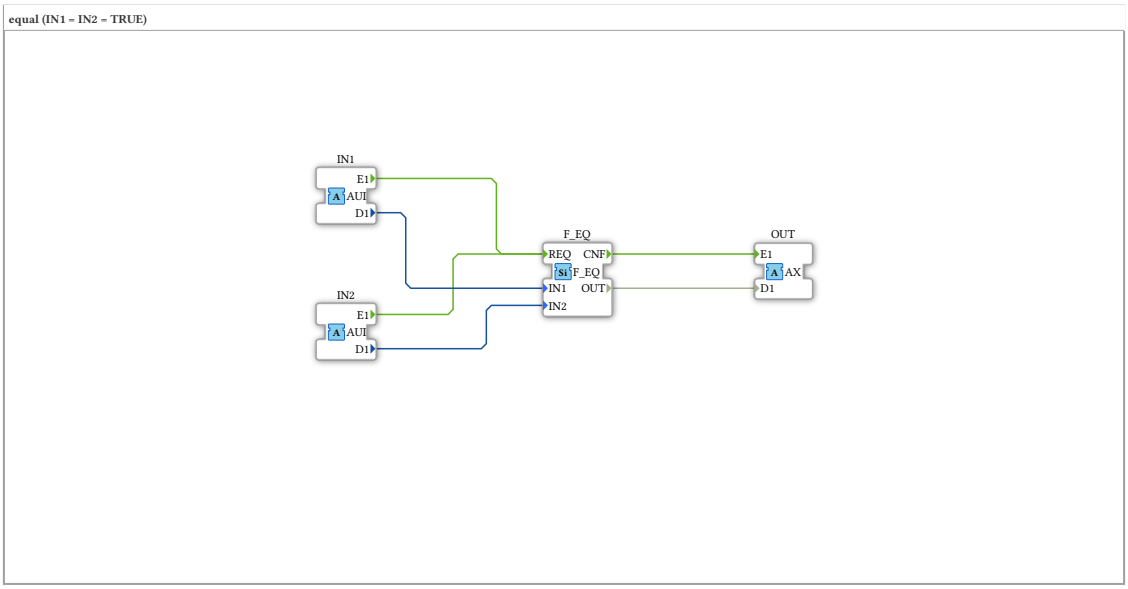
verbindet er Typensicherheit mit bewährter Funktionalität. Er eignet sich besonders für ereignisgesteuerte Automatisierungslösungen, bei denen modulare und wiederverwendbare Komponenten gefragt sind.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUI_EQ





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock AUI_EQ führt einen Gleichheitsvergleich zwischen zwei analogen Werten durch. Er ist als standard comparison function nach IEC 61131-3 klassifiziert und nutzt Adapter für die typsichere Kommunikation mit der Umgebung. Das Ergebnis wird als boolescher Wert ausgegeben, der genau dann `true` ist, wenn beide Eingangswerte gleich sind.

Schnittstellenstruktur

Ereignis-Eingänge

- **IN1.E1** – Auslöser für den Vergleich (wird gesendet, wenn der erste Wert aktualisiert wird).
- **IN2.E1** – Alternativer Auslöser für den Vergleich (wird gesendet, wenn der zweite Wert aktualisiert wird).

Beide Ereignisse sind mit demselben internen Verarbeitungsereignis verbunden; der Vergleich wird bei Eintreffen eines der beiden Ereignisse angestoßen.

Ereignis-Ausgänge

- **OUT.E1** – Bestätigung, dass der Vergleich abgeschlossen ist und das Ergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1** – Erster zu vergleichender analoger Wert.
- **IN2.D1** – Zweiter zu vergleichender analoger Wert.

Daten-Ausgänge

- **OUT.D1** – Ergebnis des Vergleichs: `true`, wenn `IN1.D1 == IN2.D1`, sonst `false`.

Adapter

- **IN1** (Socket) – Adapter vom Typ `adapter::types::unidirectional::AUI`, kapselt den ersten Eingangswert und das zugehörige Ereignis.
- **IN2** (Socket) – Adapter vom Typ `adapter::types::unidirectional::AUI`, kapselt den zweiten Eingangswert und das zugehörige Ereignis.
- **OUT** (Plug) – Adapter vom Typ `adapter::types::unidirectional::AX`, kapselt das Ergebnis und das Ausgangsereignis.

Die Adapter ermöglichen eine lose Kopplung zwischen dem Baustein und seiner Umgebung und vereinfachen die Wiederverwendung in verschiedenen Anwendungen.

Funktionsweise

AUI_EQ delegiert die eigentliche Vergleichslogik an den internen Funktionsblock `F_EQ` (Typ `iec61131::comparison::F_EQ`). Sobald an einem der Eingangsadapter (IN1 oder IN2) ein Ereignis eintrifft, wird die Berechnung gestartet. Der interne `F_EQ` vergleicht die beiden übergebenen Werte und liefert das boolesche Ergebnis. Nach Abschluss der Berechnung wird das Ergebnis über den Ausgangsadapter OUT ausgegeben und ein entsprechendes Ereignis gesendet.

Der Baustein ist ereignisgesteuert – er reagiert auf Eingangsereignisse und benötigt keinen eigenen zyklischen Aufruf.

Technische Besonderheiten

- Der Funktionsblock verwendet ausschließlich Adapter für die Kommunikation, was eine hohe Flexibilität bei der Anbindung an verschiedene Datenquellen und -senken bietet.
- Intern basiert er auf dem Standard-Baustein `F_EQ` aus der IEC 61131-Bibliothek, der eine robuste und getestete Vergleichsfunktion bereitstellt.
- Die Eingangsereignisse sind logisch ODER-verknüpft: Jedes der beiden Ereignisse löst eine Neuberechnung aus. Dies vereinfacht die Ansteuerung, da nur das aktuell aktualisierte Signal gesendet werden muss.
- Der Baustein ist in mehreren Versionen weiterentwickelt worden und wurde zuletzt 2025 aktualisiert.

Zustandsübersicht

AUI_EQ besitzt keinen expliziten Zustandsautomaten. Sein Verhalten wird vollständig durch die Ereignisverarbeitung des internen `F_EQ` bestimmt. Nach dem Eintreffen eines Ereignisses durchläuft der Baustein eine Berechnung und sendet das Ergebnis – es gibt keine wartenden oder blockierenden Zustände.

Anwendungsszenarien

- **Überwachung von Gleichheit**: Prüfen, ob zwei Sensormesswerte (z. B. Temperatur, Druck) denselben Wert aufweisen.
- **Alarmierung**: Auslösen eines Alarms, wenn zwei redundante Messungen voneinander abweichen (logische Negation des Ausgangs).
- **Steuerung**: Schalten von Aktoren, wenn ein Sollwert mit einem Istwert übereinstimmt.
- **Plausibilitätsprüfung**: Vergleichen von Analogwerten aus unterschiedlichen Quellen zur Fehlererkennung.

Vergleich mit ähnlichen Bausteinen

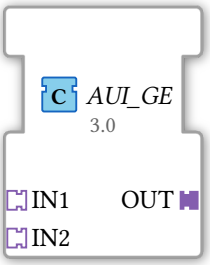
Neben `AUI_EQ` existieren in der IEC 61131-Bibliothek weitere Vergleichsbausteine wie `AUI_GT` (größer), `AUI_LT` (kleiner) und `AUI_NE` (ungleich). Diese teilen sich die gleiche Adapter-Schnittstelle, unterscheiden sich jedoch in der Vergleichsoperation. `AUI_EQ` ist speziell für den Gleichheitstest optimiert und bietet keine anderen Vergleichslogiken. Für umfassendere Vergleiche (z B. größer oder gleich) sind Kombinationen mit logischen Bausteinen erforderlich.

Fazit

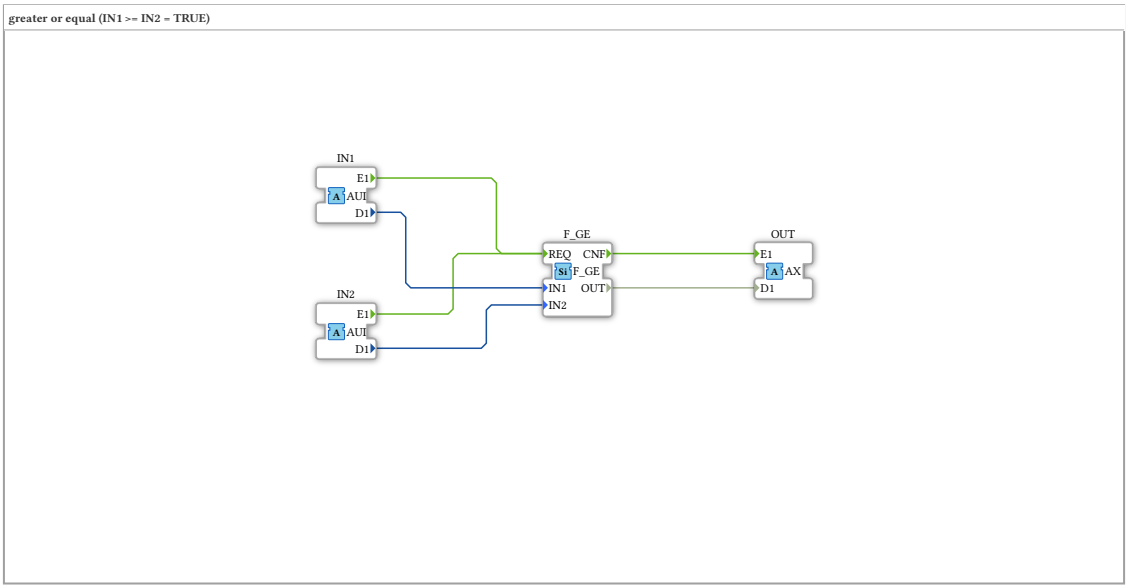
Der Funktionsblock AUI_EQ realisiert eine einfache und zuverlässige Gleichheitsprüfung für analoge Werte. Die Verwendung von Adaptern macht ihn äußerst anpassungsfähig für verschiedene Automatisierungsumgebungen. Seine ereignisgesteuerte Arbeitsweise und die interne Nutzung eines standardisierten Vergleichsbausteins gewährleisten eine konsistente und performante Ausführung. AUI_EQ ist damit ein grundlegender Baustein für Überwachungs- und Steuerungsaufgaben, bei denen Wertegleichheit festgestellt werden muss.

AUI_GE

Input value 1 -- left operand (IN1 >= IN2) – *AUI*
Input value 2 -- right operand (IN1 >= IN2) – *AUI*



AX – IN1 >= IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AUI_GE** führt einen Vergleich zweier Werte durch und prüft, ob der erste Wert (IN1) größer oder gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp, abhängig vom Adaptertyp `AUI`, üblicherweise ein numerischer Wert).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 >= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AUI` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AUI` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AUI` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GE` (Typ `iec61131::comparison::F_GE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter, wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater or equal“).
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Prüfen, ob ein Messwert (z. B. Füllstand, Drehzahl) einen vorgegebenen Schwellwert erreicht oder überschreitet.
- **Steuerungslogik**: Aktivieren einer Aktion (z. B. Pumpe stoppen), wenn ein Istwert einen Sollwert erreicht oder überschreitet.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Mindestbedingung erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AUI_LE**: Prüft auf „kleiner gleich“ – invers zur Funktion von AUI_GE.
- **AUI_EQ**: Prüft auf Gleichheit der Werte.
- **AUI_GT**: Prüft auf „größer als“ – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version AUI_GE ist.

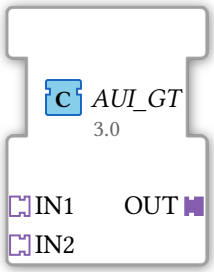
Der AUI_GE ergänzt die AUI-Vergleichsfamilie um den Grenzfall der Gleichheit und ist damit die passende Wahl, wenn ein Schwellwert eingeschlossen werden soll.

Fazit

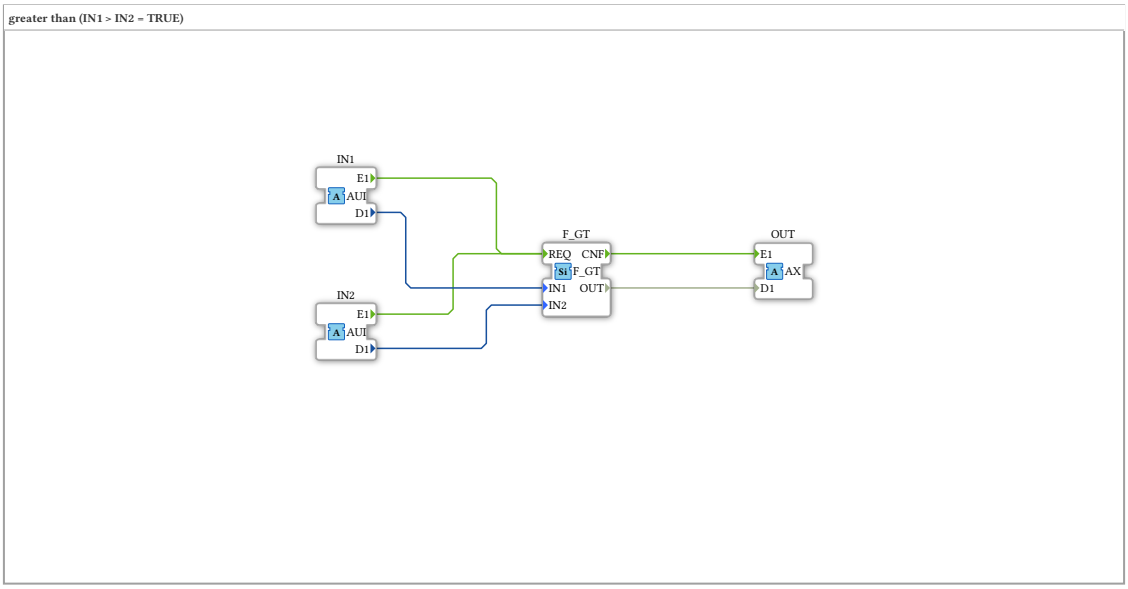
Der **AUI_GE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AUI_GT

Input value 1 -- left operand (IN1 > IN2) – *AUI*
Input value 2 -- right operand (IN1 > IN2) – *AUI*



AX – IN1 > IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AUI_GT** führt einen Vergleich zweier Werte durch und prüft, ob der erste Wert (IN1) größer als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp, abhängig vom Adaptertyp `AUI`, üblicherweise ein numerischer Wert).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 > IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AUI` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AUI` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AUI` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GT` (Typ `iec61131::comparison::F_GT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter, wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than“).
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Prüfen, ob ein Messwert (z. B. Temperatur, Druck) einen vorgegebenen Schwellwert überschreitet.
- **Steuerungslogik**: Aktivieren einer Aktion (z. B. Ventil öffnen), wenn ein Sollwert größer als der Istwert ist.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine bestimmte Bedingung erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AUI_LT**: Prüft auf „kleiner als“ – invers zur Funktion von AUI_GT.
- **AUI_EQ**: Prüft auf Gleichheit der Werte.
- **AUI_GE** / **AUI_LE**: Führen die Vergleiche „größer gleich“ bzw. „kleiner gleich“ durch – sind erweiterte Versionen von AUI_GT und AUI_LT.

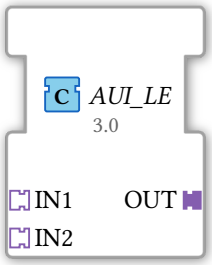
Der AUI_GT ist die grundlegende „größer“-Komponente, die über die beiden Socket-Adapter eine einfache Integration in Adapter-basierte Netzwerke ermöglicht.

Fazit

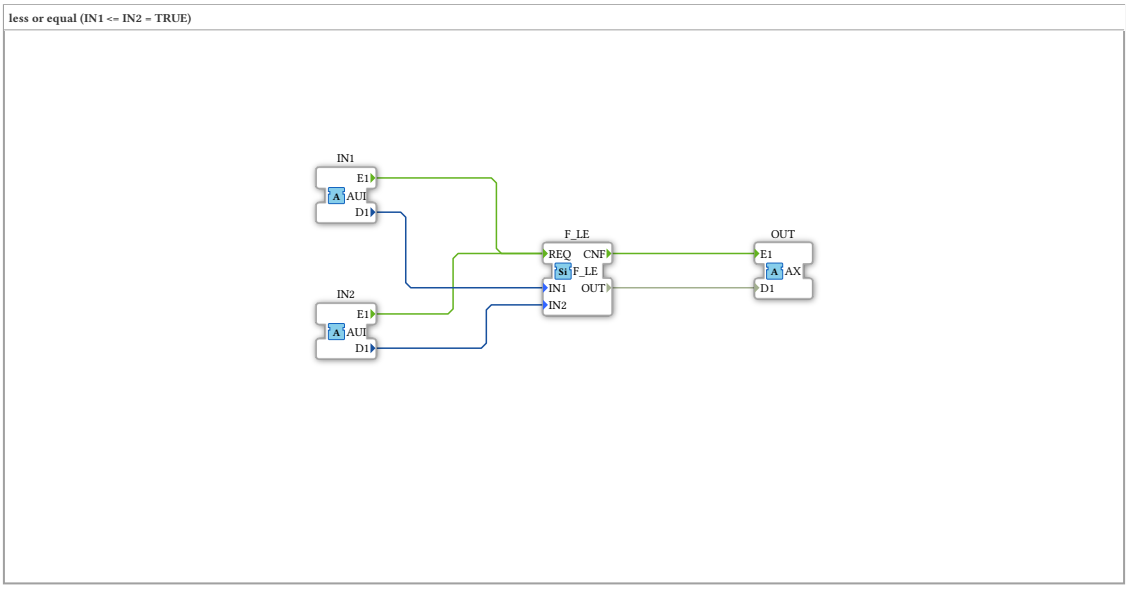
Der **AUI_GT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AUI_LE

Input value 1 -- left operand (IN1 <= IN2) – *AUI*
Input value 2 -- right operand (IN1 <= IN2) – *AUI*



AX – IN1 <= IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUI_LE** führt einen Vergleich zweier analoger Eingangswerte auf „kleiner oder gleich“ (less or equal) durch. Er ist als Composite-Baustein realisiert, der intern den IEC-61131-3 Baustein `F_LE` verwendet. Das Ergebnis (ein BOOL-Wert) wird über einen Ausgangsadapter bereitgestellt. Der FB eignet sich für ereignisgesteuerte Grenzwertüberwachungen in Automatisierungssystemen.

Schnittstellenstruktur

Ereignis-Eingänge

- **IN1.E1** (über Adapter IN1) – Ereignis, das den Vergleich auslöst, sobald der erste Eingangswert bereitsteht.
- **IN2.E1** (über Adapter IN2) – Ereignis, das den Vergleich auslöst, sobald der zweite Eingangswert bereitsteht.

Ereignis-Ausgänge

- **OUT.E1** (über Adapter OUT) – Ereignis, das nach erfolgreichem Vergleich ausgegeben wird und signalisiert, dass das Ergebnis gültig ist.

Daten-Eingänge

- **IN1.D1** (über Adapter IN1) – Erster Eingangswert (analog, Typ gemäß Adapter `AUI`).
- **IN2.D1** (über Adapter IN2) – Zweiter Eingangswert (analog, Typ gemäß Adapter `AUI`).

Daten-Ausgänge

- **OUT.D1** (über Adapter OUT) – Ergebnis des Vergleichs (BOOL): `TRUE`, wenn IN1 ≤ IN2, sonst `FALSE`.

Adapter

- **IN1** – Socket (Typ: `adapter::types::unidirectional::AUI`) für den ersten Analogwert.
- **IN2** – Socket (Typ: `adapter::types::unidirectional::AUI`) für den zweiten Analogwert.
- **OUT** – Plug (Typ: `adapter::types::unidirectional::AX`) für das Vergleichsergebnis.

Funktionsweise

Der Baustein enthält ein internes Netzwerk mit einem `F_LE`-Baustein. Die Ereignisse der beiden Eingangsadapter (`IN1.E1` und `IN2.E1`) sind mit dem `REQ`-Eingang des `F_LE` verbunden. Jedes eintreffende Ereignis (egal an welchem Eingang) startet den Vergleich der aktuellen Datenwerte `IN1.D1` und `IN2.D1`. Der interne `F_LE` berechnet `(IN1 ≤ IN2)` und legt das Ergebnis auf seinem Ausgang `OUT` ab. Dieses wird über den Adapter `OUT.D1` ausgegeben und gleichzeitig ein Ereignis über `OUT.E1` gesendet.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Die Ein- und Ausgänge sind als unidirektionale Adapter realisiert, was eine flexible Kopplung mit anderen Bausteinen ermöglicht.
- **Ereignissteuerung**: Der Vergleich wird durch jedes anliegende Ereignis an einem der beiden Eingänge ausgelöst – es muss nicht zwingend ein gleichzeitiges Ereignis an beiden Eingängen vorliegen.
- **Interner Standardbaustein**: Die Vergleichslogik nutzt den geprüften IEC-61131-3 Baustein `F_LE`.
- **Lizenz**: Der Baustein steht unter der Eclipse Public License 2.0 (EPL-2.0), Copyright (c) 2013 TU Wien ACIN.

Zustandsübersicht

Der Baustein besitzt keine explizite Zustandsmaschine. Er arbeitet ereignisgesteuert und seine Ausgabe ist das direkte Ergebnis des internen kombinatorischen Vergleichs. Nach der Ereignisbearbeitung liegt das Ergebnis bis zum nächsten Ereignis stabil an.

Anwendungsszenarien

- **Grenzwertüberwachung**: Prüfen, ob ein Messwert (z. B. Temperatur, Druck) einen eingestellten Schwellwert nicht überschreitet.
- **Steuerungslogik**: Auslösen einer Aktion, wenn zwei analoge Signale in einer festgelegten Relation stehen (z. B. Füllstand ≤ Maximalfüllstand).
- **Sicherheitsfunktionen**: Vergleich von redundanten Sensorwerten auf „kleiner gleich“ zur Fehlererkennung.

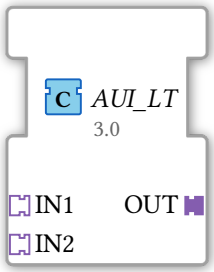
Vergleich mit ähnlichen Bausteinen

- **AUI_LT**: Führt den Vergleich „kleiner“ (strict less) durch.
- **AUI_EQ**: Prüft auf Gleichheit der beiden Werte.
- **AUI_GT**: Prüft auf „größer“.
- **AUI_GE**: Prüft auf „größer oder gleich“.
  Alle diese Bausteine besitzen die gleiche Adapterschnittstelle (AUI für Eingänge, AX für Ausgang) und unterscheiden sich nur in der Vergleichsoperation.

Fazit

Der Baustein `AUI_LE` bietet eine einfache und zuverlässige Möglichkeit, analoge Signale auf „kleiner oder gleich“ zu prüfen. Dank der adapterbasierten Schnittstelle und der ereignisgesteuerten Ausführung lässt er sich nahtlos in Automatisierungsprojekte integrieren, die standardisierte Vergleichsfunktionen benötigen.

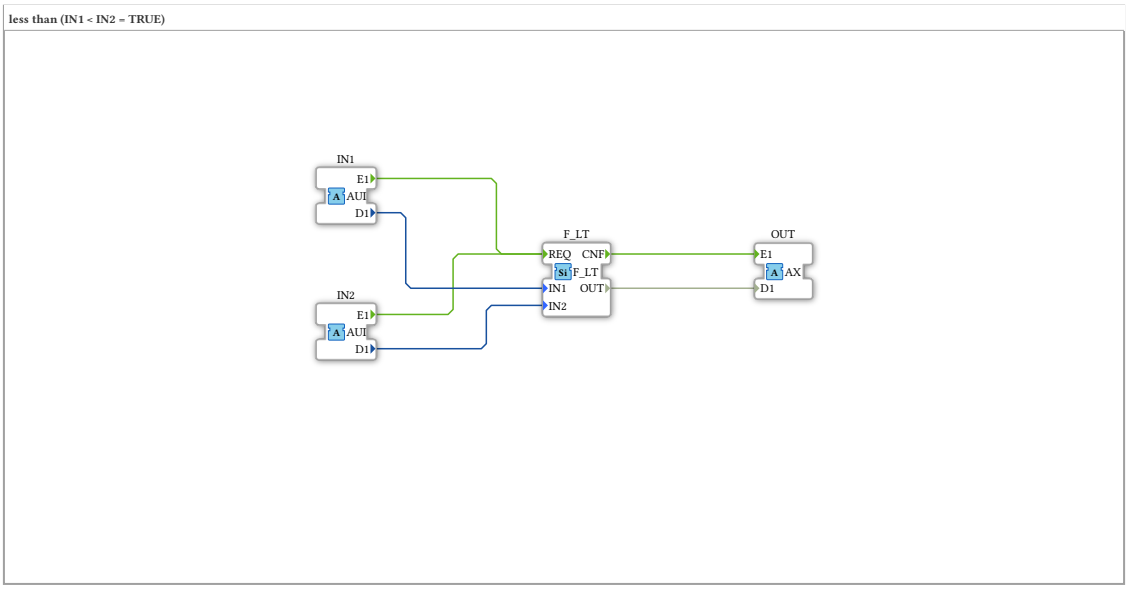
AUI_LT



Input value 1 -- left operand (IN1 < IN2) – *AUI*

Input value 2 -- right operand (IN1 < IN2) – *AUI*

AX – IN1 < IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `AUI_LT` realisiert einen Kleiner-als-Vergleich (<) zwischen zwei Eingangswerten. Er wird über Adapter-Schnittstellen angesprochen und eignet sich für den Einsatz in komponentenbasierten Automatisierungssystemen nach IEC 61131-3. Der Baustein führt den Vergleich asynchron durch und gibt das Ergebnis als booleschen Wert über einen Ausgangsadapter aus.

Schnittstellenstruktur

Ereignis-Eingänge

Bezeichnung	Beschreibung
IN1.E1	Ereignis am Socket-Adapter IN1; löst den Vergleich aus, sobald ein neuer Wert an IN1 anliegt.
IN2.E1	Ereignis am Socket-Adapter IN2; löst den Vergleich aus, sobald ein neuer Wert an IN2 anliegt.

Ereignis-Ausgänge

Bezeichnung	Beschreibung
OUT.E1	Ereignis am Plug-Adapter OUT; signalisiert, dass nach einem Vergleich ein gültiges Ergebnis an OUT.D1 anliegt.

Daten-Eingänge

Bezeichnung	Beschreibung
IN1.D1	Erster Wert des Vergleichs (ANY_BIT-kompatibel).
IN2.D1	Zweiter Wert des Vergleichs (ANY_BIT-kompatibel).

Daten-Ausgänge

Bezeichnung	Beschreibung
OUT.D1	Ergebnis des Vergleichs: `true`, wenn IN1.D1 < IN2.D1; sonst `false`.

Adapter

Bezeichnung	Typ	Richtung	Beschreibung
IN1	AUI (unidirectional)	Socket	Eingangsadapter für den ersten Wert und dessen Ereignis.
IN2	AUI (unidirectional)	Socket	Eingangsadapter für den zweiten Wert und dessen Ereignis.
OUT	AX (unidirectional)	Plug	Ausgangsadapter für das Vergleichsergebnis und das zugehörige Ereignis.

Funktionsweise

Der `AUI_LT`-Baustein ist als Composite-FB realisiert. Intern wird ein einzelner `F_LT`-Funktionsblock (aus der IEC 61131-3 Vergleichsbibliothek) verwendet.

- Die **Ereignis-Eingänge** `IN1.E1` und `IN2.E1` sind beide mit dem `REQ`-Eingang des internen `F_LT` verbunden. Sobald an einem der beiden Adapter ein Ereignis eintrifft, wird der Vergleich angestoßen.
- Die **Datenwerte** `IN1.D1` und `IN2.D1` werden direkt an die entsprechenden Eingänge `IN1` und `IN2` des `F_LT` weitergeleitet.
- Nach Abschluss der Berechnung signalisiert der `F_LT` über seinen Ausgang `CNF` das Ereignis `OUT.E1` und stellt das Vergleichsergebnis am Datenausgang `OUT.D1` bereit.

Der Vergleich erfolgt nach der Logik: **IN1.D1 < IN2.D1 ⟹ OUT.D1 = true**, andernfalls `false`.

Technische Besonderheiten

- **Adapterbasierte Kommunikation**: Alle Ein- und Ausgänge erfolgen über unidirektionale Adapter (Typen `AUI` und `AX`). Dies ermöglicht eine lose Kopplung und Wiederverwendung in verschiedenen Umgebungen.
- **Asynchrone Auslösung**: Da beide Ereigniseingänge auf den gleichen `REQ` des internen `F_LT` führen, kann der Vergleich durch ein Ereignis an **einem** der beiden Eingänge gestartet werden. Es ist nicht erforderlich, dass beide Ereignisse gleichzeitig eintreffen.
- **Typkompatibilität**: Der Baustein arbeitet mit beliebigen `ANY_BIT`-Datentypen (z. B. `BOOL`, `BYTE`, `WORD`, `DWORD`, `LWORD`), sofern die angeschlossenen Adapter dies unterstützen.
- **Keine Zustandsspeicherung**: Der FB ist rein kombinatorisch; er speichert keine historischen Werte.

Zustandsübersicht

Da der `AUI_LT` keine explizite Zustandsmaschine besitzt, existiert lediglich ein **impliziter Zustand**:

Zustand	Beschreibung
Bereit	Wartet auf ein Ereignis an IN1.E1 oder IN2.E1. Nach Auslösung wird der Vergleich durchgeführt und das Ergebnis über OUT ausgegeben. Der FB kehrt sofort in den Bereit-Zustand zurück.

Anwendungsszenarien

- **Grenzwertüberwachung**: Prüfen, ob ein Messwert unter einem Schwellwert liegt, z. B. `Temperatur < Grenzwert`.

- **Steuerungslogik**: Bedingte Aktivierung von Ausgängen, z. B. Ventil öffnen, wenn `Druck < Sollwert`.
- **Interlock-Bedingungen**: Sicherheitsabfragen, bei denen ein Betrieb nur erlaubt ist, wenn eine Größe kleiner als eine andere ist.
- **Vergleichsketten**: Kombination mit anderen Vergleichsbausteinen (z. B. `AUI_EQ`, `AUI_GT`) zur Realisierung komplexer logischer Bedingungen.

Vergleich mit ähnlichen Bausteinen

Baustein	Funktion	Schnittstelle	Besonderheit
AUI_LT	Kleiner als (<)	Adapter (AUI, AX)	Asynchron, zwei separate Ereigniseingänge
F_LT	Kleiner als (<)	Standard-Ports (REQ/CNF, ANY_BIT)	Direkte Ereignis-/Datenports, kein Adapter
AUI_EQ	Gleich (=)	Adapter (AUI, AX)	Gleiche Struktur, anderer Vergleichsoperator
AUI_GT	Größer als (>)	Adapter (AUI, AX)	Gleiche Struktur, anderer Vergleichsoperator

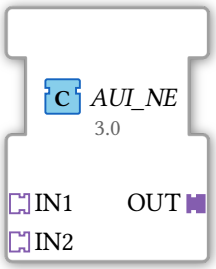
Der `AUI_LT` bietet durch die Adapter-Schnittstellen eine besonders flexible Integration in IEC 61131-3-konforme Systeme, während der interne `F_LT` die standardisierte Vergleichslogik bereitstellt.

Fazit

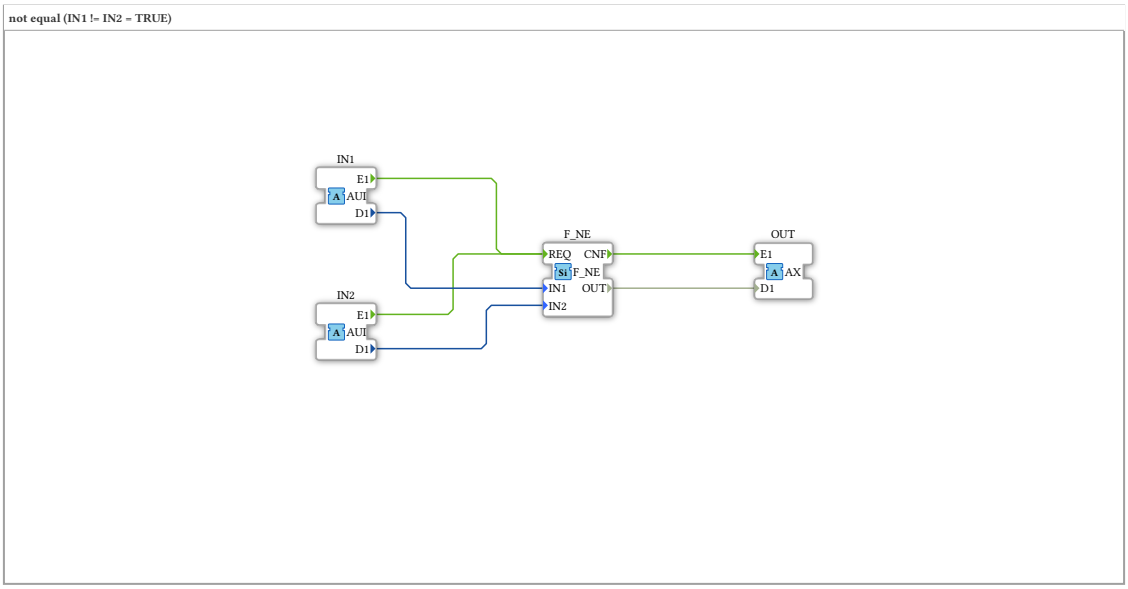
Der `AUI_LT` ist ein kompakter und zuverlässiger Funktionsblock für Kleiner-als-Vergleiche in adapterbasierten Automatisierungsumgebungen. Seine asynchrone Ereignisverarbeitung und die lose Kopplung über Adapter machen ihn ideal für modulare Steuerungsanwendungen, bei denen einfache Bedingungen dezentral und ressourcenschonend ausgewertet werden müssen.

AUI_NE

Input value 1 -- left operand (IN1 != IN2) – *AUI*
Input value 2 -- right operand (IN1 != IN2) – *AUI*



AX – IN1 != IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AUI_NE` realisiert einen Ungleich-Vergleich (not equal) für zwei Eingangswerte. Er ist als Composite-FB (zusammengesetzter Baustein) aus dem IEC-61131-3-Vergleichsbaustein `F_NE` aufgebaut. Die Kommunikation mit der Umgebung erfolgt über standardisierte Adapter-Schnittstellen (`AUI` für die Eingänge, `AX` für den Ausgang). Dadurch lässt er sich in adapterbasierte Architekturen der 4diac-IDE nahtlos integrieren.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine direkten Ereignis-Eingänge. Ereignisse werden über die **Socket-Adapter** `IN1` und `IN2` empfangen:

Adapter	Ereignis	Beschreibung
`IN1`	Ereignis über Adapter-Eingang E1	Startet den Vergleich, wenn der Wert an `IN1` bereitsteht.
`IN2`	Ereignis über Adapter-Eingang E1	Startet den Vergleich, wenn der Wert an `IN2` bereitsteht.

Beide Ereignisse führen zur Aktivierung des internen `F_NE`-Bausteins.

Ereignis-Ausgänge

Der FB besitzt keine direkten Ereignis-Ausgänge. Das Ergebnis wird über den **Plug-Adapter** `OUT` ausgegeben:

Adapter	Ereignis	Beschreibung
`OUT`	Ereignis über Adapter-Ausgang E1	Zeigt an, dass der Vergleich abgeschlossen ist und der Ergebniswert gültig ist.

Daten-Eingänge

Daten werden ebenfalls über die Socket-Adapter bereitgestellt:

Adapter	Datenpunkt	Typ	Beschreibung
`IN1`	D1	ANY (über AUI)	Erster Vergleichswert
`IN2`	D2	ANY (über AUI)	Zweiter Vergleichswert

Die genauen Datentypen werden durch die verwendete Adapter-Instanz (`adapter::types::unidirectional::AUI`) festgelegt.

Daten-Ausgänge

Der Ergebniswert wird über den Plug-Adapter ausgegeben:

Adapter	Datenpunkt	Typ	Beschreibung
`OUT`	D1	BOOL (über AX)	`TRUE`, wenn IN1 ≠ IN2; `FALSE` sonst.

Adapter

Adapter	Richtung	Typ	Beschreibung
`IN1`	Socket	`adapter::types::unidirectional::AUI`	Aufnehmender Adapter für den ersten Eingangswert.
`IN2`	Socket	`adapter::types::unidirectional::AUI`	Aufnehmender Adapter für den zweiten Eingangswert.
`OUT`	Plug	`adapter::types::unidirectional::AX`	Ausgebender Adapter für das Vergleichsergebnis.

Funktionsweise

Bei jedem eingehenden Ereignis über `IN1.E1` oder `IN2.E1` wird der interne Baustein `F_NE` (IEC 61131-3: `NOT EQUAL`) getriggert. `F_NE` vergleicht die Werte an den Datenpunkten `IN1.D1` und `IN2.D2`. Das Ergebnis (BOOL) wird an den Ausgangsadapter `OUT.D1` weitergeleitet und gleichzeitig ein Ereignis an `OUT.E1` gesendet, um den Ergebnisabruf zu signalisieren.

Der FB arbeitet **ereignisgesteuert** – ein Vergleich wird nur dann neu berechnet, wenn mindestens einer der beiden Eingangsadapter ein Ereignis empfängt. Dadurch wird unnötige Rechenlast vermieden.

Technische Besonderheiten

- **Adapter-Kapselung**: Die gesamte Ein-/Ausgabe erfolgt über standardisierte Adapter, wodurch der FB in komplexe Adapter-Netzwerke eingebunden werden kann.
- **Wiederverwendung**: Intern wird der bewährte IEC-61131-3-Baustein `F_NE` genutzt, der einen effizienten, typunabhängigen Ungleich-Vergleich implementiert.
- **Unidirektionale Adapter**: Die verwendeten Adapter (`AUI`, `AX`) sind unidirektional – sie unterstützen jeweils nur eine Daten- und Ereignisrichtung, was die Klarheit der Datenflüsse erhöht.
- **Keine Zustandsspeicherung**: Der FB ist zustandslos; jedes Ereignis führt zu einer Neuberechnung.

Zustandsübersicht

Der FB `AUI_NE` selbst besitzt keinen eigenen Zustandsautomaten. Das Verhalten wird vollständig durch den internen `F_NE`-Baustein bestimmt, der einen einfachen funktionalen Ablauf ohne Zustände realisiert. Die Ereignissteuerung (Eingang → Berechnung → Ausgang) ist strikt kausal.

Anwendungsszenarien

- **Abweichungsüberwachung**: Prüfen, ob zwei Messwerte (z. B. Temperatur, Druck) voneinander abweichen.
- **Schwellwertverletzung**: Kombination mit anderen Bausteinen, um Ungleichheit als Teil einer Plausibilitätsprüfung zu nutzen.
- **Datenvalidierung**: Erkennen, ob zwei redundante Signale unterschiedliche Werte liefern (Vergleich von Kopien).
- **Ereignisgesteuerte Vergleiche**: Einbettung in Applikationen, die nur bei Änderung eines der Eingänge einen Vergleich benötigen.

Vergleich mit ähnlichen Bausteinen

Baustein	Vergleichsfunktion	Besonderheit
`AUI_EQ`	Gleichheit (equal)	Ergebnis `TRUE` bei gleichen Werten
`AUI_NE`	Ungleichheit (not equal)	Ergebnis `TRUE` bei unterschiedlichen Werten
`AUI_GT`	Größer (greater than)	Ergebnis `TRUE` wenn IN1 > IN2
`AUI_LT`	Kleiner (less than)	Ergebnis `TRUE` wenn IN1 < IN2

Alle genannten Bausteine teilen sich die gleiche Adapter-Schnittstelle und lassen sich daher leicht austauschen oder kombinieren.

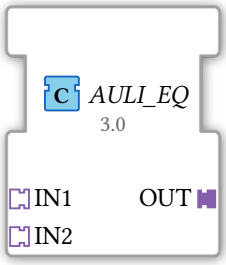
Fazit

Der `AUI_NE` ist ein einfacher, aber zuverlässiger Funktionsblock für den Ungleich-Vergleich zweier Werte innerhalb einer adapterbasierten 4diac-Umgebung. Seine Stärken liegen in der sauberen Kapselung durch Adapter, der ereignisgesteuerten Ausführung und der hohen Wiederverwendbarkeit dank des zugrundeliegenden IEC-61131-3-Bausteins. Er eignet sich besonders für modulare, erweiterbare Automatisierungslösungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

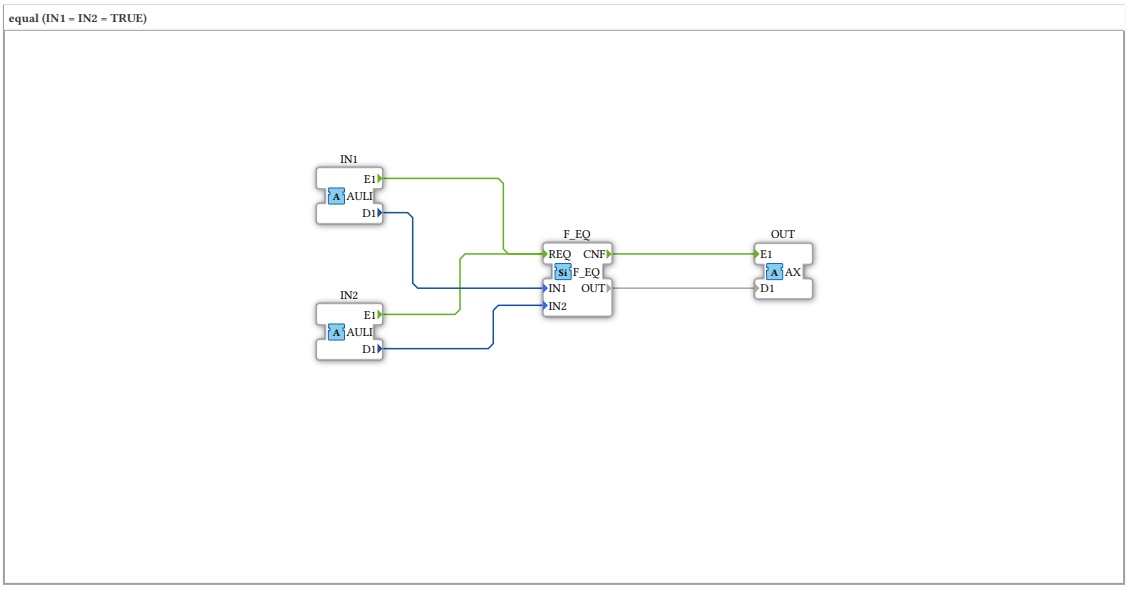
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AULI_EQ



Input value 1 -- left operand (IN1 = IN2) – *AULI*
Input value 2 -- right operand (IN1 = IN2) – *AULI*

AX – IN1 = IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AULI_EQ** führt einen Vergleich zweier Werte vom Typ `ULINT` (64-Bit-Ganzzahl (vorzeichenlos)) durch und prüft, ob der erste Wert (IN1) gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `ULINT`, 64-Bit-Ganzzahl (vorzeichenlos)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 = IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AULI` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AULI` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AULI` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_EQ` (Typ `iec61131::comparison::F_EQ`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AULI`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („equal to“) für den Datentyp `ULINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: große Zähl- und Mengenwerte gegen einen Referenz- oder Schwellwert vom Typ `ULINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `ULINT` die Bedingung „gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `ULINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

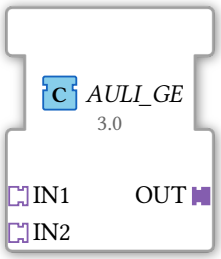
- **AULI_NE**: Prüft auf „ungleich“ – invers zur Funktion von AULI_EQ.
- **AULI_NE**: Prüft auf Ungleichheit der Werte.

Der AULI_EQ ist Teil der vollständigen AULI-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `ULINT`.

Fazit

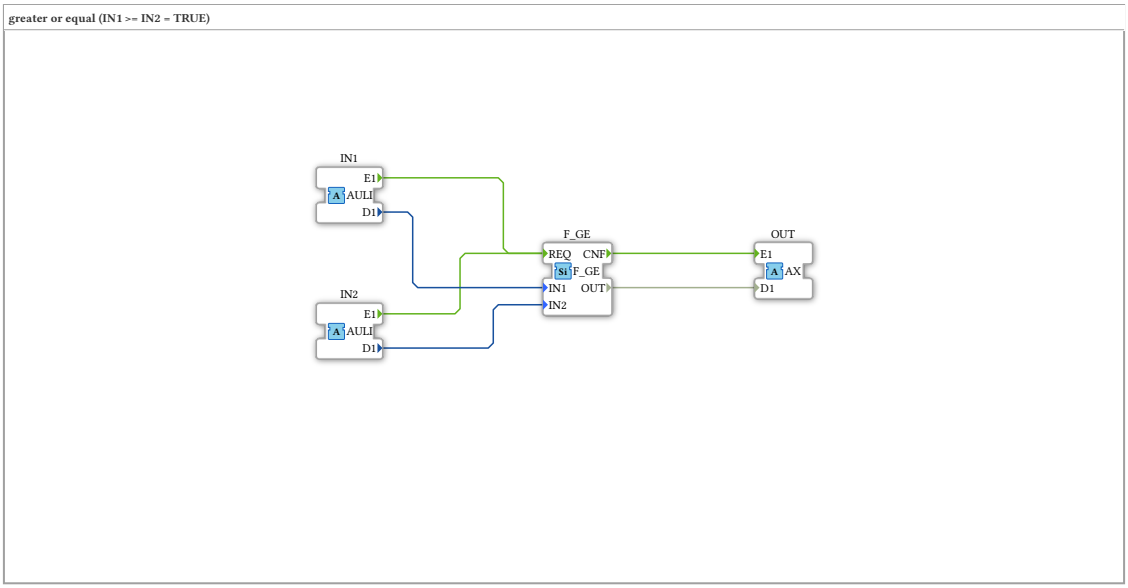
Der **AULI_EQ** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `ULINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AULI_GE



Input value 1 -- left operand (IN1 >= IN2) – *AULI*
Input value 2 -- right operand (IN1 >= IN2) – *AULI*

AX – IN1 >= IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AULI_GE** führt einen Vergleich zweier Werte vom Typ `ULINT` (64-Bit-Ganzzahl (vorzeichenlos)) durch und prüft, ob der erste Wert (IN1) größer gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `ULINT`, 64-Bit-Ganzzahl (vorzeichenlos)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 >= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AULI` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AULI` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AULI` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GE` (Typ `iec61131::comparison::F_GE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AULI`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than or equal to“) für den Datentyp `ULINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: große Zähl- und Mengenwerte gegen einen Referenz- oder Schwellwert vom Typ `ULINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `ULINT` die Bedingung „größer gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `ULINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AULI_LE**: Prüft auf „kleiner gleich“ – invers zur Funktion von AULI_GE.
- **AULI_EQ**: Prüft auf Gleichheit der Werte.
- **AULI_GT**: Führt den Vergleich „größer als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version AULI_GE ist.

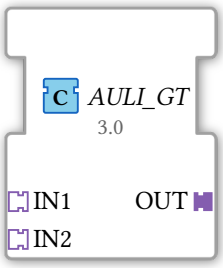
Der AULI_GE ergänzt die AULI-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

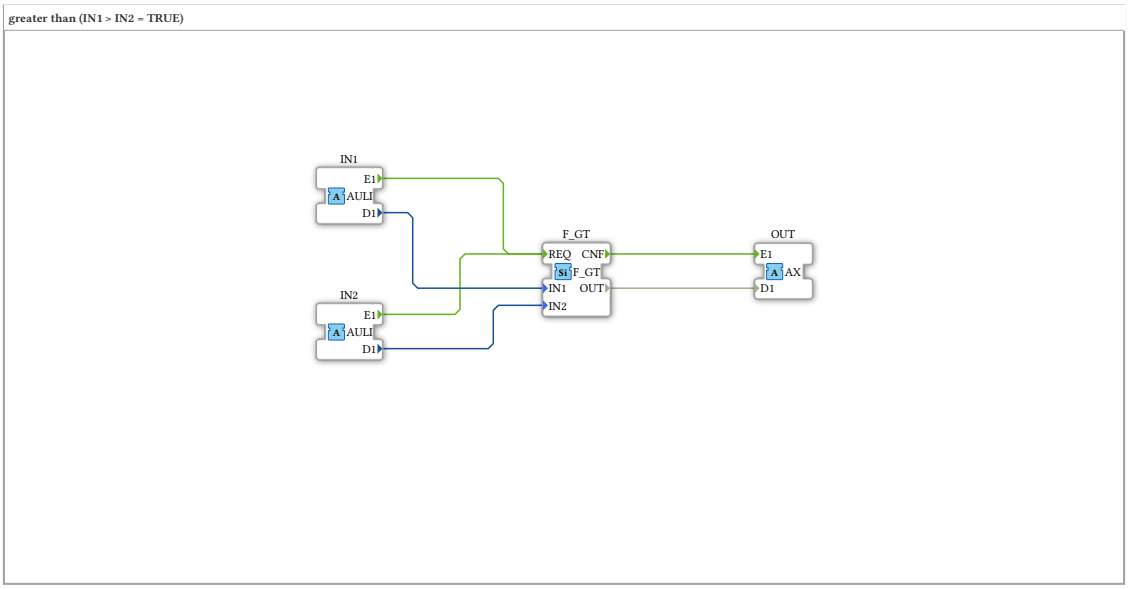
Der **AULI_GE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `ULINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeits-

weise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AULI_GT



Input value 1 -- left operand (IN1 > IN2) – *AULI*
Input value 2 -- right operand (IN1 > IN2) – *AULI*
AX – IN1 > IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AULI_GT** führt einen Vergleich zweier Werte vom Typ `ULINT` (64-Bit-Ganzzahl (vorzeichenlos)) durch und prüft, ob der erste Wert (IN1) größer als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `ULINT`, 64-Bit-Ganzzahl (vorzeichenlos)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 > IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AULI` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AULI` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AULI` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GT` (Typ `iec61131::comparison::F_GT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AULI`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than“) für den Datentyp `ULINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: große Zähl- und Mengenwerte gegen einen Referenz- oder Schwellwert vom Typ `ULINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `ULINT` die Bedingung „größer als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `ULINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

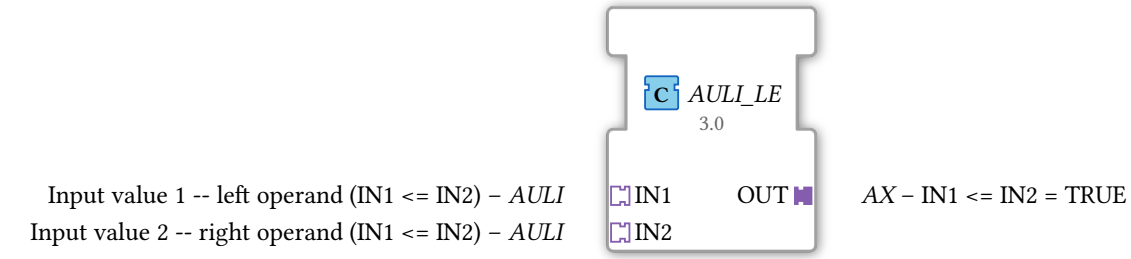
- **AULI_LT**: Prüft auf „kleiner als“ – invers zur Funktion von AULI_GT.
- **AULI_EQ**: Prüft auf Gleichheit der Werte.
- **AULI_GE**: Führt den Vergleich „größer gleich“ durch – die um den Gleichheitsfall erweiterte Version von AULI_GT.

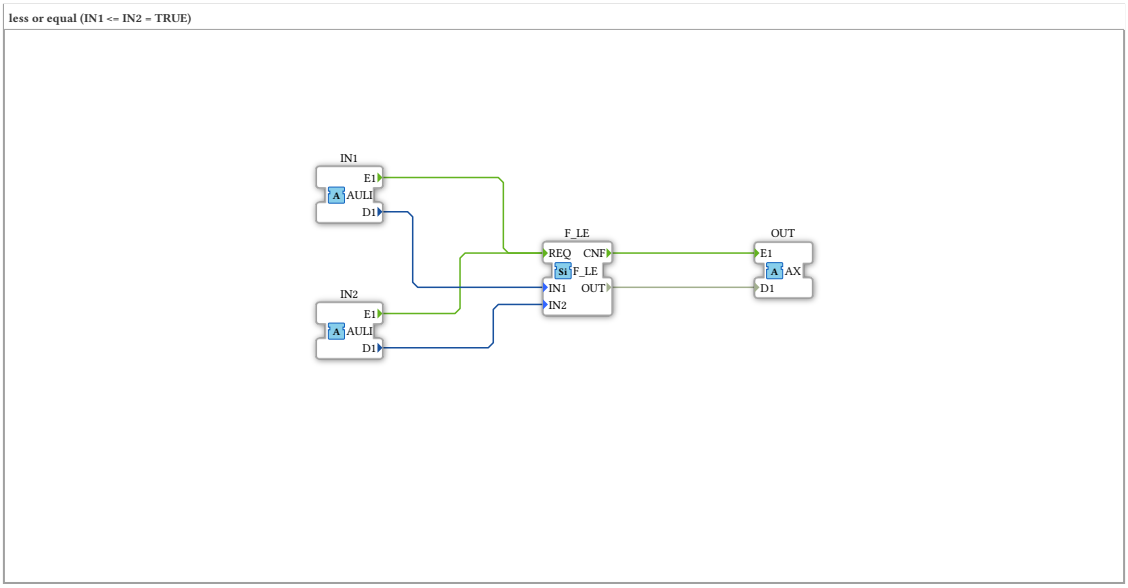
Der AULI_GT ist die grundlegende „größer als“-Komponente ohne Gleichheitsfall, ergänzt durch AULI_GE.

Fazit

Der **AULI_GT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `ULINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AULI_LE





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AULI_LE** führt einen Vergleich zweier Werte vom Typ `ULINT` (64-Bit-Ganzzahl (vorzeichenlos)) durch und prüft, ob der erste Wert (IN1) kleiner gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `ULINT`, 64-Bit-Ganzzahl (vorzeichenlos)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 <= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AULI` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AULI` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AULI` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LE` (Typ `iec61131::comparison::F_LE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AULI`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than or equal to“) für den Datentyp `ULINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: große Zähl- und Mengenwerte gegen einen Referenz- oder Schwellwert vom Typ `ULINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `ULINT` die Bedingung „kleiner gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `ULINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

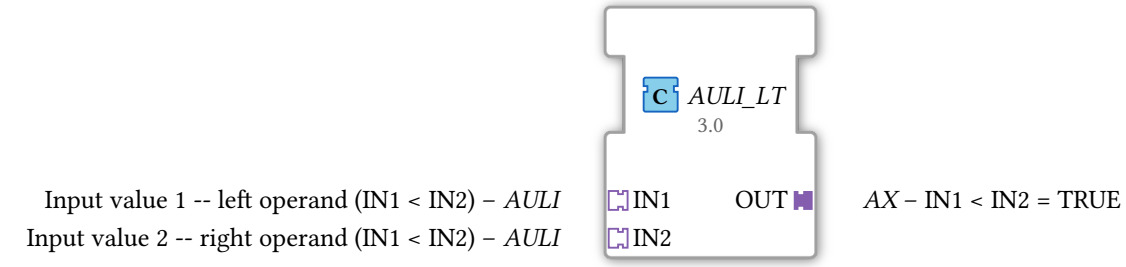
- **AULI_GE**: Prüft auf „größer gleich“ – invers zur Funktion von AULI_LE.
- **AULI_EQ**: Prüft auf Gleichheit der Werte.
- **AULI_LT**: Führt den Vergleich „kleiner als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version AULI_LE ist.

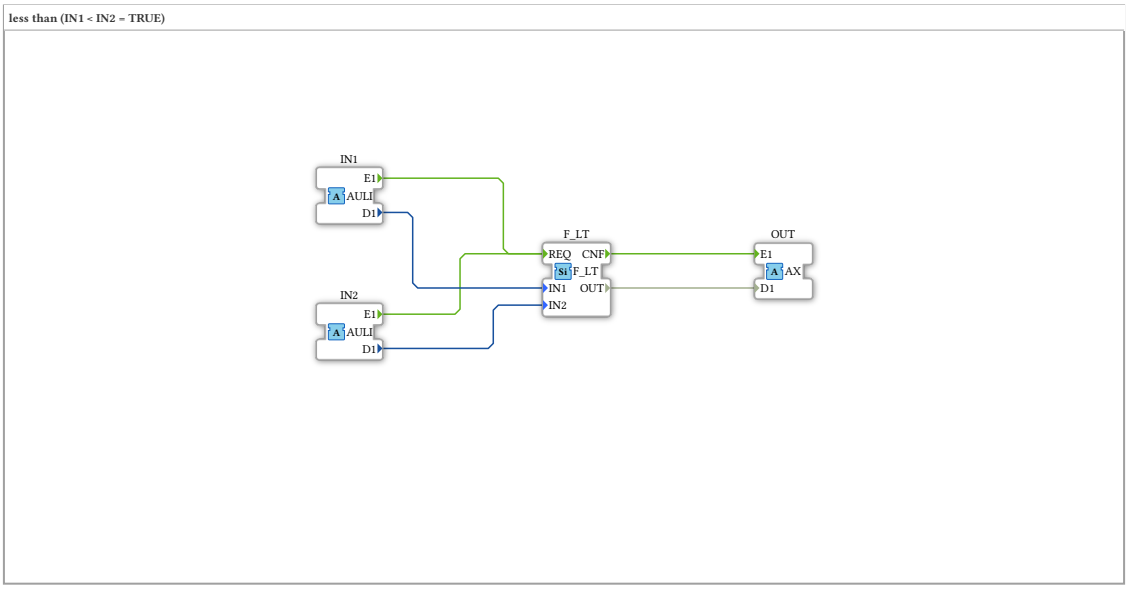
Der AULI_LE ergänzt die AULI-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

Der **AULI_LE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `ULINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AULI_LT





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AULI_LT** führt einen Vergleich zweier Werte vom Typ `ULINT` (64-Bit-Ganzzahl (vorzeichenlos)) durch und prüft, ob der erste Wert (IN1) kleiner als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `ULINT`, 64-Bit-Ganzzahl (vorzeichenlos)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 < IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AULI` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AULI` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AULI` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LT` (Typ `iec61131::comparison::F_LT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AULI`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than“) für den Datentyp `ULINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: große Zähl- und Mengenwerte gegen einen Referenz- oder Schwellwert vom Typ `ULINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `ULINT` die Bedingung „kleiner als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `ULINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AULI_GT**: Prüft auf „größer als“ – invers zur Funktion von AULI_LT.
- **AULI_EQ**: Prüft auf Gleichheit der Werte.
- **AULI_LE**: Führt den Vergleich „kleiner gleich“ durch – die um den Gleichheitsfall erweiterte Version von AULI_LT.

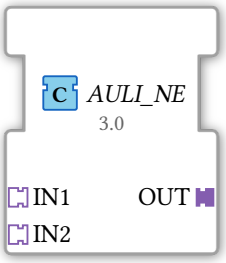
Der AULI_LT ist die grundlegende „kleiner als“-Komponente ohne Gleichheitsfall, ergänzt durch AULI_LE.

Fazit

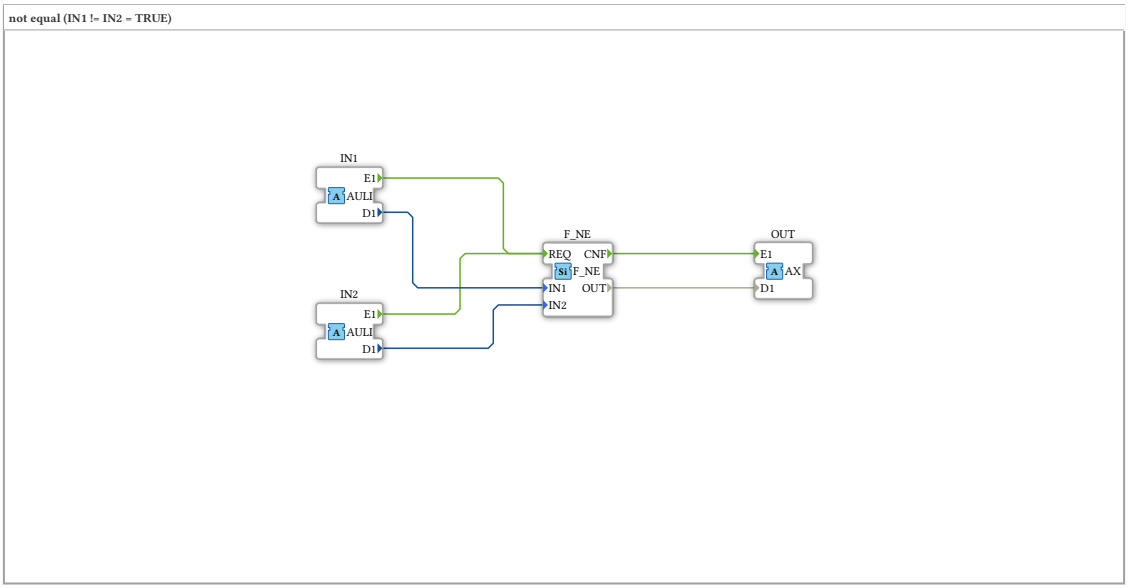
Der **AULI_LT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `ULINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AULI_NE

Input value 1 -- left operand (IN1 != IN2) – *AULI*
Input value 2 -- right operand (IN1 != IN2) – *AULI*



AX – IN1 != IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AULI_NE** führt einen Vergleich zweier Werte vom Typ `ULINT` (64-Bit-Ganzzahl (vorzeichenlos)) durch und prüft, ob der erste Wert (IN1) ungleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `ULINT`, 64-Bit-Ganzzahl (vorzeichenlos)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 != IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AULI` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AULI` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AULI` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_NE` (Typ `iec61131::comparison::F_NE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AULI`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („not equal to“) für den Datentyp `ULINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: große Zähl- und Mengenwerte gegen einen Referenz- oder Schwellwert vom Typ `ULINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `ULINT` die Bedingung „ungleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `ULINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AULI_EQ**: Prüft auf „gleich“ – invers zur Funktion von AULI_NE.
- **AULI_EQ**: Prüft auf Gleichheit der Werte.

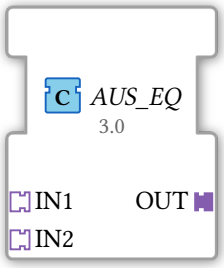
Der AULI_NE ist Teil der vollständigen AULI-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `ULINT`.

Fazit

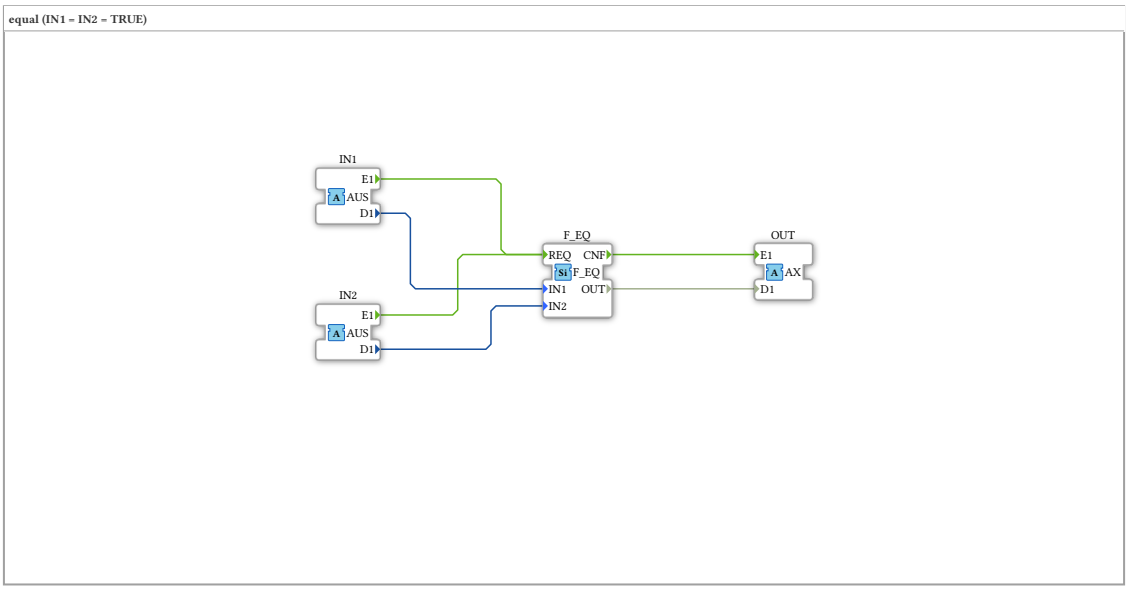
Der **AULI_NE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `ULINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AUS_EQ

Input value 1 -- left operand (IN1 = IN2) – *AUS*
Input value 2 -- right operand (IN1 = IN2) – *AUS*



AX – IN1 = IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AUS_EQ** führt einen Vergleich zweier Werte vom Typ `USINT` (8-Bit-Ganzzahl (vorzeichenlos)) durch und prüft, ob der erste Wert (IN1) gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `USINT`, 8-Bit-Ganzzahl (vorzeichenlos)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 = IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AUS` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AUS` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AUS` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_EQ` (Typ `iec61131::comparison::F_EQ`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AUS`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („equal to“) für den Datentyp `USINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: kleine Zähl- und Indexwerte gegen einen Referenz- oder Schwellwert vom Typ `USINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `USINT` die Bedingung „gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `USINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

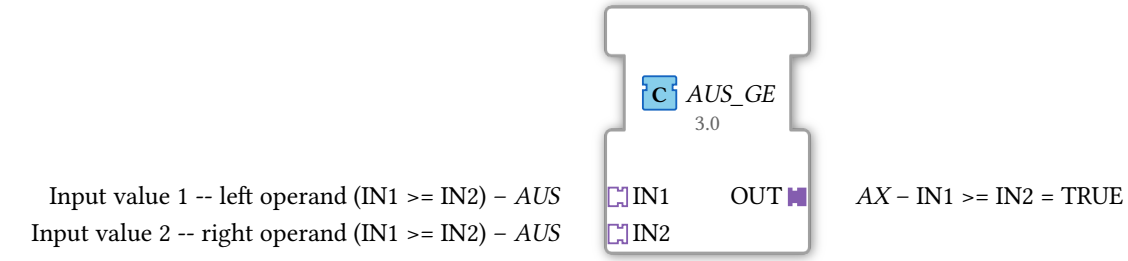
- **AUS_NE**: Prüft auf „ungleich“ – invers zur Funktion von AUS_EQ.
- **AUS_NE**: Prüft auf Ungleichheit der Werte.

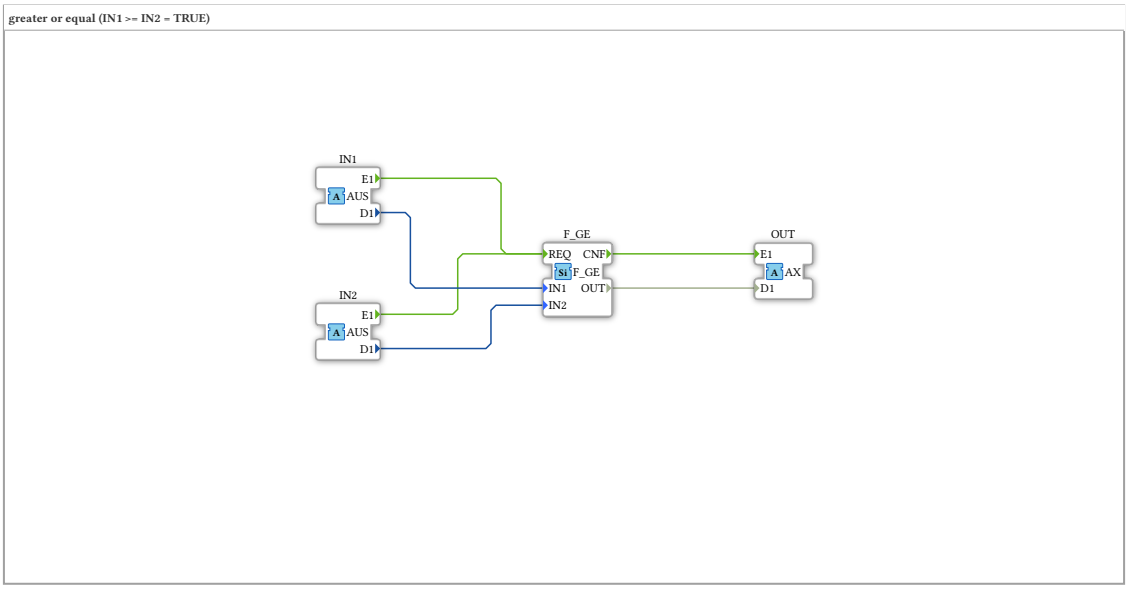
Der AUS_EQ ist Teil der vollständigen AUS-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `USINT`.

Fazit

Der **AUS_EQ** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `USINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AUS_GE





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AUS_GE** führt einen Vergleich zweier Werte vom Typ `USINT` (8-Bit-Ganzzahl (vorzeichenlos)) durch und prüft, ob der erste Wert (IN1) größer gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `USINT`, 8-Bit-Ganzzahl (vorzeichenlos)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 >= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AUS` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AUS` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AUS` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GE` (Typ `iec61131::comparison::F_GE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AUS`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than or equal to“) für den Datentyp `USINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: kleine Zähl- und Indexwerte gegen einen Referenz- oder Schwellwert vom Typ `USINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `USINT` die Bedingung „größer gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `USINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AUS_LE**: Prüft auf „kleiner gleich“ – invers zur Funktion von AUS_GE.
- **AUS_EQ**: Prüft auf Gleichheit der Werte.
- **AUS_GT**: Führt den Vergleich „größer als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version AUS_GE ist.

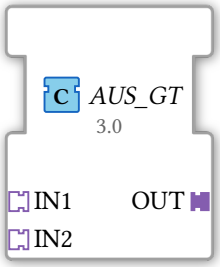
Der AUS_GE ergänzt die AUS-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

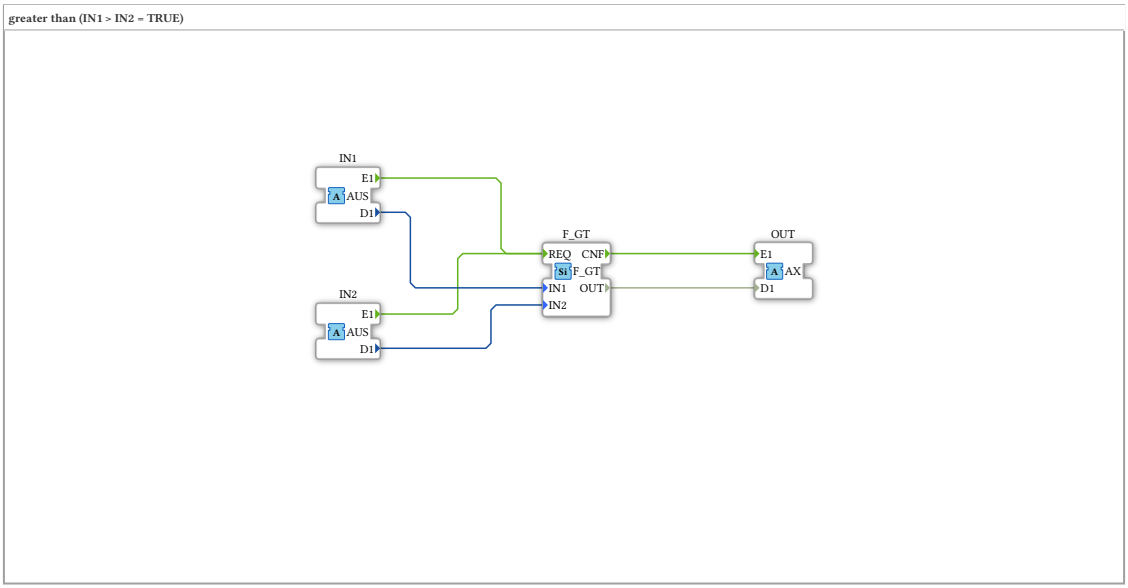
Der **AUS_GE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `USINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AUS_GT

Input value 1 -- left operand (IN1 > IN2) – *AUS*
Input value 2 -- right operand (IN1 > IN2) – *AUS*



AX – IN1 > IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AUS_GT** führt einen Vergleich zweier Werte vom Typ `USINT` (8-Bit-Ganzzahl (vorzeichenlos)) durch und prüft, ob der erste Wert (IN1) größer als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.

- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `USINT`, 8-Bit-Ganzzahl (vorzeichenlos)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 > IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AUS` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AUS` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AUS` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GT` (Typ `iec61131::comparison::F_GT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AUS`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than“) für den Datentyp `USINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: kleine Zähl- und Indexwerte gegen einen Referenz- oder Schwellwert vom Typ `USINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `USINT` die Bedingung „größer als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `USINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AUS_LT**: Prüft auf „kleiner als“ – invers zur Funktion von AUS_GT.
- **AUS_EQ**: Prüft auf Gleichheit der Werte.
- **AUS_GE**: Führt den Vergleich „größer gleich“ durch – die um den Gleichheitsfall erweiterte Version von AUS_GT.

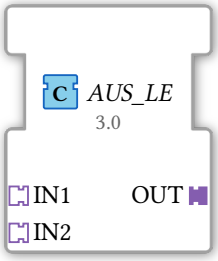
Der AUS_GT ist die grundlegende „größer als“-Komponente ohne Gleichheitsfall, ergänzt durch AUS_GE.

Fazit

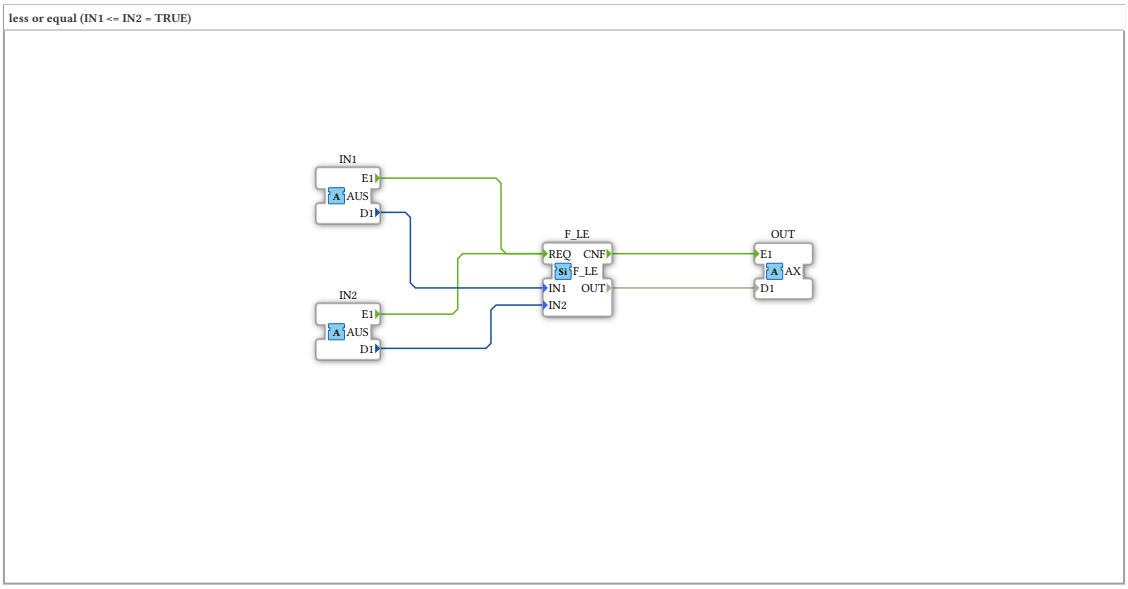
Der **AUS_GT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `USINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AUS_LE

Input value 1 -- left operand (IN1 <= IN2) – *AUS*
Input value 2 -- right operand (IN1 <= IN2) – *AUS*



AX – IN1 <= IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AUS_LE** führt einen Vergleich zweier Werte vom Typ `USINT` (8-Bit-Ganzzahl (vorzeichenlos)) durch und prüft, ob der erste Wert (IN1) kleiner gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.

- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `USINT`, 8-Bit-Ganzzahl (vorzeichenlos)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 <= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AUS` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AUS` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AUS` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LE` (Typ `iec61131::comparison::F_LE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AUS`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than or equal to“) für den Datentyp `USINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: kleine Zähl- und Indexwerte gegen einen Referenz- oder Schwellwert vom Typ `USINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `USINT` die Bedingung „kleiner gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `USINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AUS_GE**: Prüft auf „größer gleich“ – invers zur Funktion von AUS_LE.
- **AUS_EQ**: Prüft auf Gleichheit der Werte.
- **AUS_LT**: Führt den Vergleich „kleiner als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version AUS_LE ist.

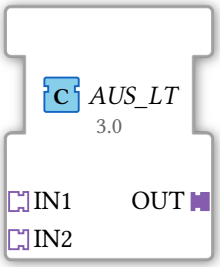
Der AUS_LE ergänzt die AUS-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

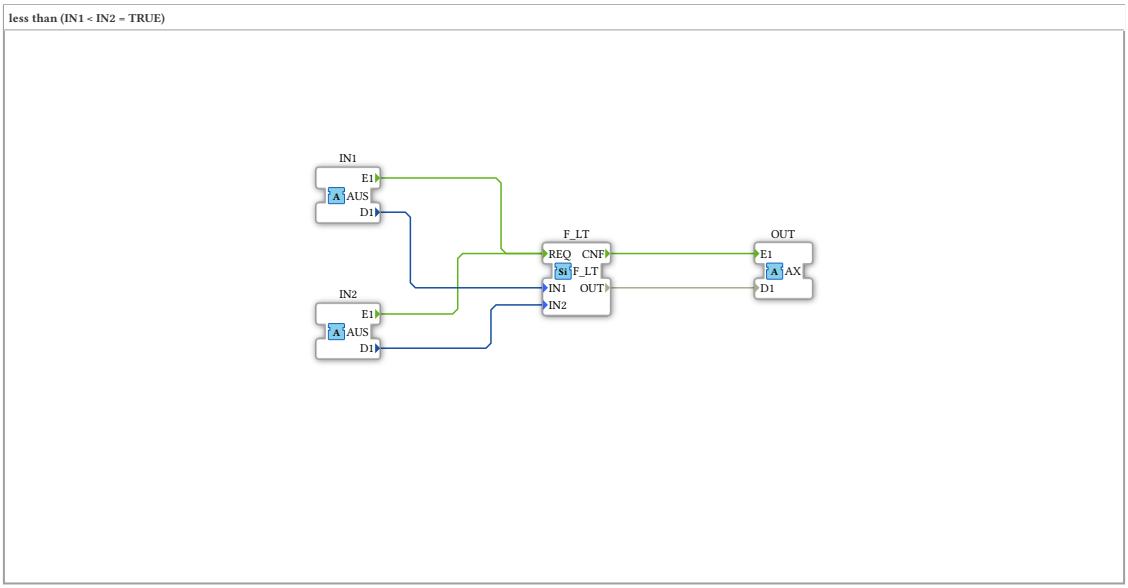
Der **AUS_LE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `USINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AUS_LT

Input value 1 -- left operand (IN1 < IN2) – *AUS*
Input value 2 -- right operand (IN1 < IN2) – *AUS*



AX – IN1 < IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AUS_LT** führt einen Vergleich zweier Werte vom Typ `USINT` (8-Bit-Ganzzahl (vorzeichenlos)) durch und prüft, ob der erste Wert (IN1) kleiner als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.

- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `USINT`, 8-Bit-Ganzzahl (vorzeichenlos)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 < IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AUS` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AUS` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AUS` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LT` (Typ `iec61131::comparison::F_LT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AUS`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than“) für den Datentyp `USINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: kleine Zähl- und Indexwerte gegen einen Referenz- oder Schwellwert vom Typ `USINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `USINT` die Bedingung „kleiner als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `USINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AUS_GT**: Prüft auf „größer als“ – invers zur Funktion von AUS_LT.
- **AUS_EQ**: Prüft auf Gleichheit der Werte.
- **AUS_LE**: Führt den Vergleich „kleiner gleich“ durch – die um den Gleichheitsfall erweiterte Version von AUS_LT.

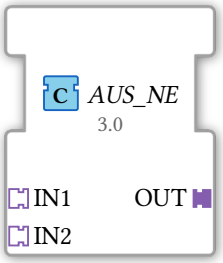
Der AUS_LT ist die grundlegende „kleiner als“-Komponente ohne Gleichheitsfall, ergänzt durch AUS_LE.

Fazit

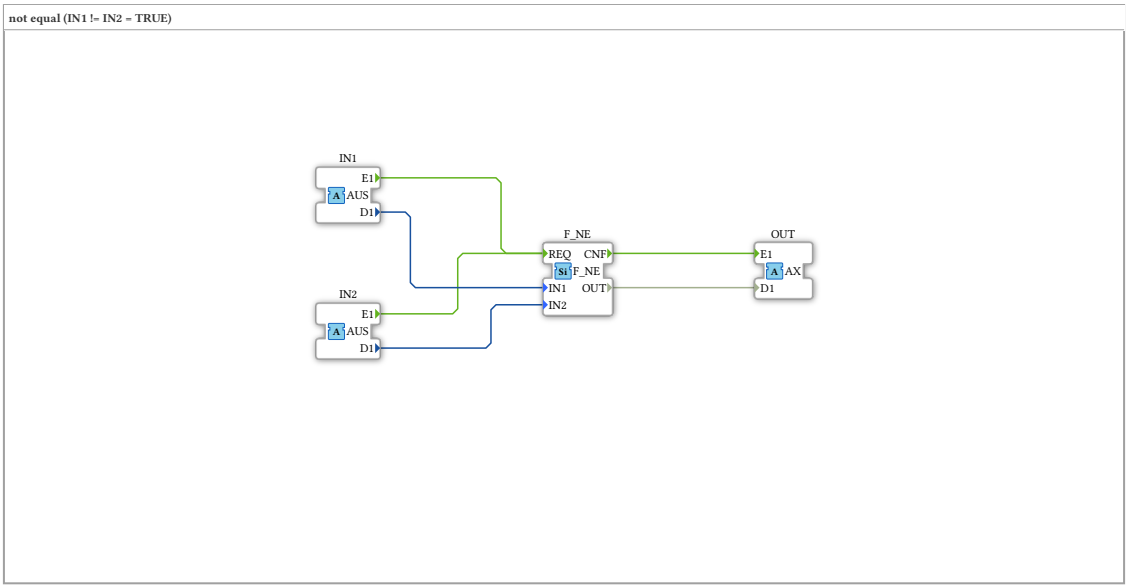
Der **AUS_LT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `USINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AUS_NE

Input value 1 -- left operand (IN1 != IN2) – *AUS*
Input value 2 -- right operand (IN1 != IN2) – *AUS*



AX – IN1 != IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AUS_NE** führt einen Vergleich zweier Werte vom Typ `USINT` (8-Bit-Ganzzahl (vorzeichenlos)) durch und prüft, ob der erste Wert (IN1) ungleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Der Vergleich erfolgt im üblichen numerischen Sinn.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.

- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `USINT`, 8-Bit-Ganzzahl (vorzeichenlos)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 != IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AUS` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AUS` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AUS` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_NE` (Typ `iec61131::comparison::F_NE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AUS`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („not equal to“) für den Datentyp `USINT`.
- **Vergleichslogik**: Der Vergleich erfolgt im üblichen numerischen Sinn.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: kleine Zähl- und Indexwerte gegen einen Referenz- oder Schwellwert vom Typ `USINT` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `USINT` die Bedingung „ungleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `USINT`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AUS_EQ**: Prüft auf „gleich“ – invers zur Funktion von AUS_NE.
- **AUS_EQ**: Prüft auf Gleichheit der Werte.

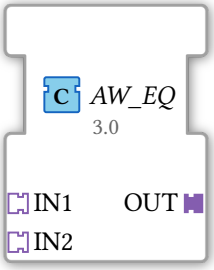
Der AUS_NE ist Teil der vollständigen AUS-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `USINT`.

Fazit

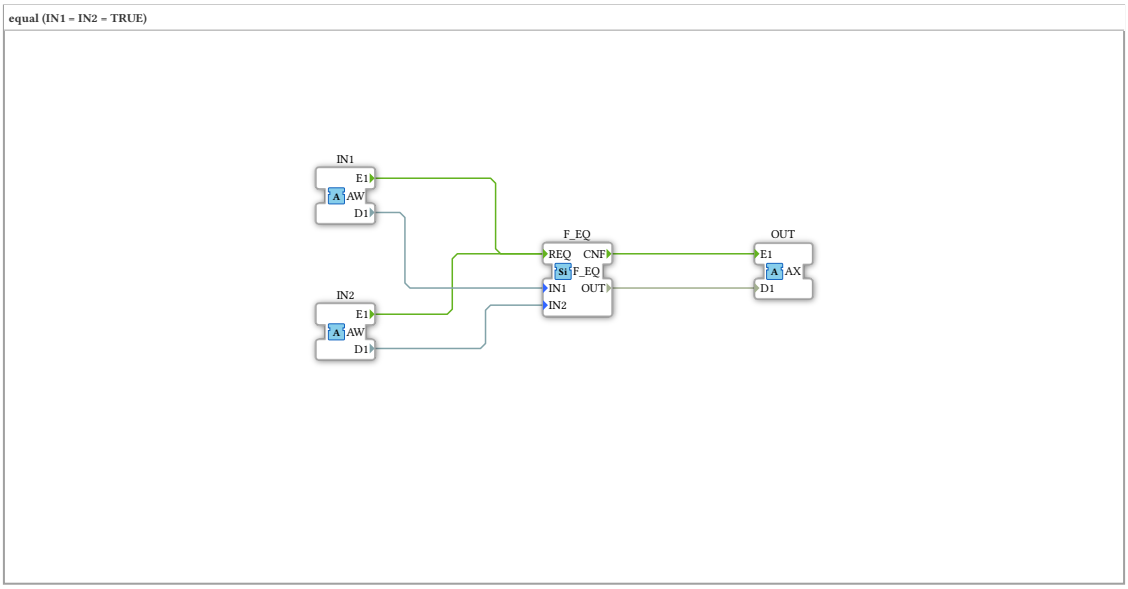
Der **AUS_NE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `USINT`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AW_EQ

Input value 1 -- left operand (IN1 = IN2) – *AW*
Input value 2 -- right operand (IN1 = IN2) – *AW*



AX – IN1 = IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AW_EQ** führt einen Vergleich zweier Werte vom Typ `WORD` (16-Bit-Bitmuster (Wort)) durch und prüft, ob der erste Wert (IN1) gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.

- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `WORD`, 16-Bit-Bitmuster (Wort)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 = IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AW` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AW` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AW` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_EQ` (Typ `iec61131::comparison::F_EQ`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AW`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („equal to“) für den Datentyp `WORD`.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Statuswörter gegen einen Referenz- oder Schwellwert vom Typ `WORD` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `WORD` die Bedingung „gleich“ erfüllen.

- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `WORD`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AW_NE**: Prüft auf „ungleich“ – invers zur Funktion von AW_EQ.
- **AW_NE**: Prüft auf Ungleichheit der Werte.

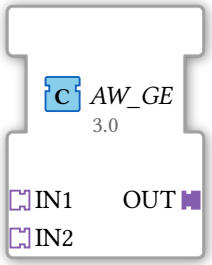
Der AW_EQ ist Teil der vollständigen AW-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `WORD`.

Fazit

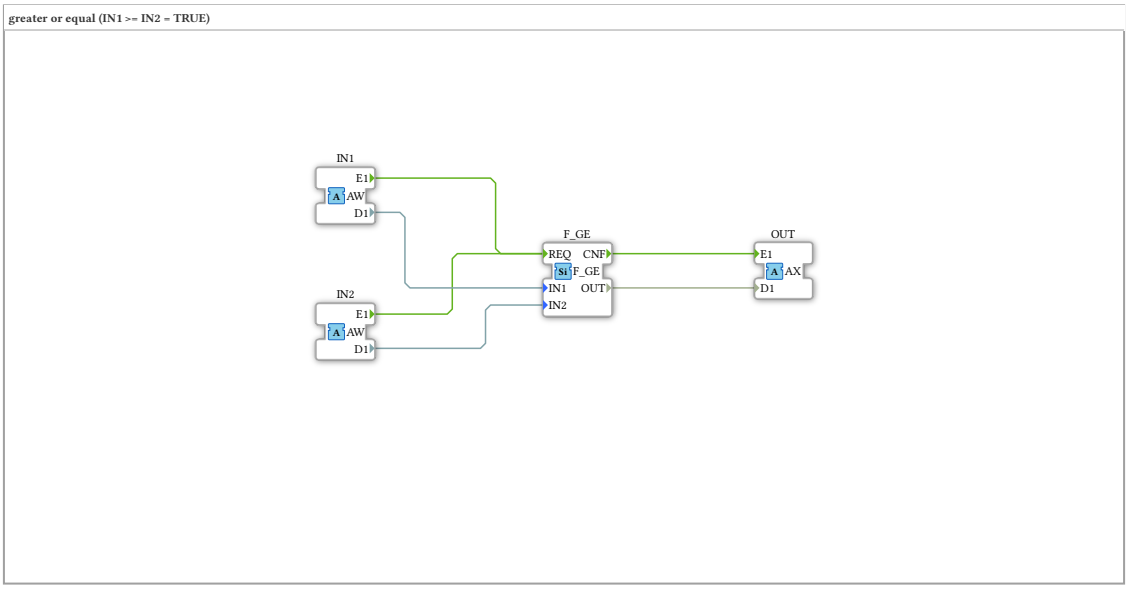
Der **AW_EQ** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `WORD`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AW_GE

Input value 1 -- left operand (IN1 >= IN2) – *AW*
Input value 2 -- right operand (IN1 >= IN2) – *AW*



AX – IN1 >= IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AW_GE** führt einen Vergleich zweier Werte vom Typ `WORD` (16-Bit-Bitmuster (Wort)) durch und prüft, ob der erste Wert (IN1) größer gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `WORD`, 16-Bit-Bitmuster (Wort)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 >= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AW` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AW` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AW` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GE` (Typ `iec61131::comparison::F_GE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AW`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than or equal to“) für den Datentyp `WORD`.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Statuswörter gegen einen Referenz- oder Schwellwert vom Typ `WORD` prüfen.

- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `WORD` die Bedingung „größer gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `WORD`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AW_LE**: Prüft auf „kleiner gleich“ – invers zur Funktion von AW_GE.
- **AW_EQ**: Prüft auf Gleichheit der Werte.
- **AW_GT**: Führt den Vergleich „größer als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version AW_GE ist.

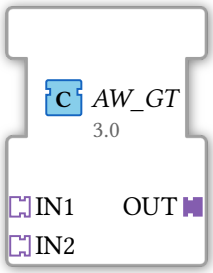
Der AW_GE ergänzt die AW-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

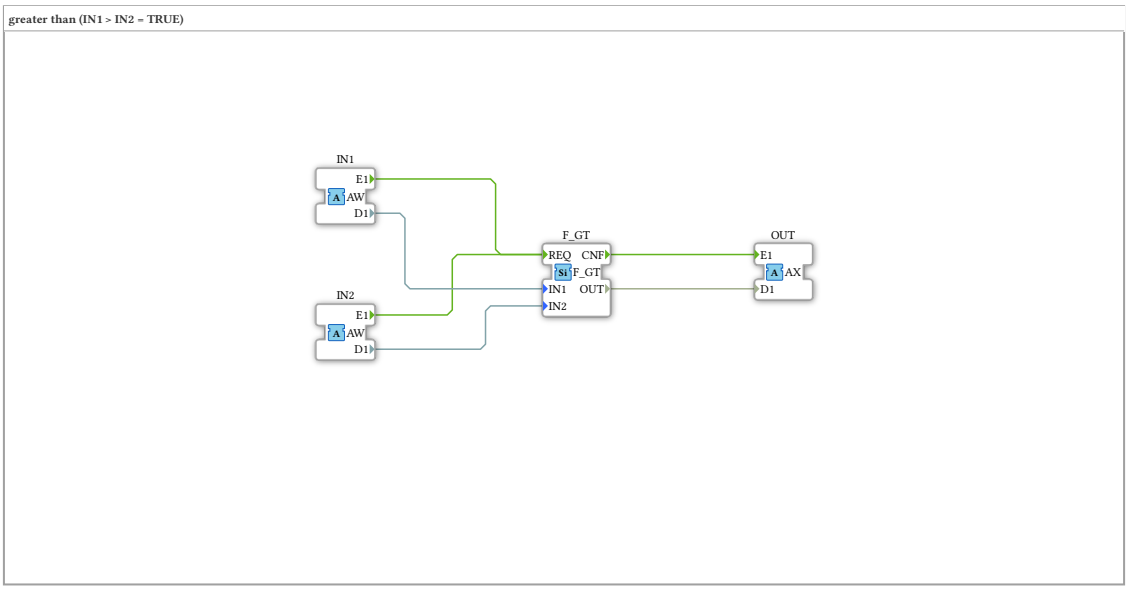
Der **AW_GE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `WORD`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AW_GT

Input value 1 -- left operand (IN1 > IN2) – *AW*
Input value 2 -- right operand (IN1 > IN2) – *AW*



AX – IN1 > IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AW_GT** führt einen Vergleich zweier Werte vom Typ `WORD` (16-Bit-Bitmuster (Wort)) durch und prüft, ob der erste Wert (IN1) größer als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `WORD`, 16-Bit-Bitmuster (Wort)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 > IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AW` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AW` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AW` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GT` (Typ `iec61131::comparison::F_GT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AW`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than“) für den Datentyp `WORD`.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Statuswörter gegen einen Referenz- oder Schwellwert vom Typ `WORD` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `WORD` die Bedingung „größer als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `WORD`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AW_LT**: Prüft auf „kleiner als“ – invers zur Funktion von AW_GT.
- **AW_EQ**: Prüft auf Gleichheit der Werte.
- **AW_GE**: Führt den Vergleich „größer gleich“ durch – die um den Gleichheitsfall erweiterte Version von AW_GT.

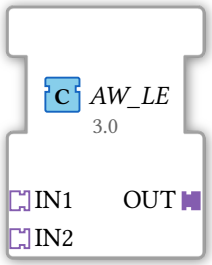
Der AW_GT ist die grundlegende „größer als“-Komponente ohne Gleichheitsfall, ergänzt durch AW_GE.

Fazit

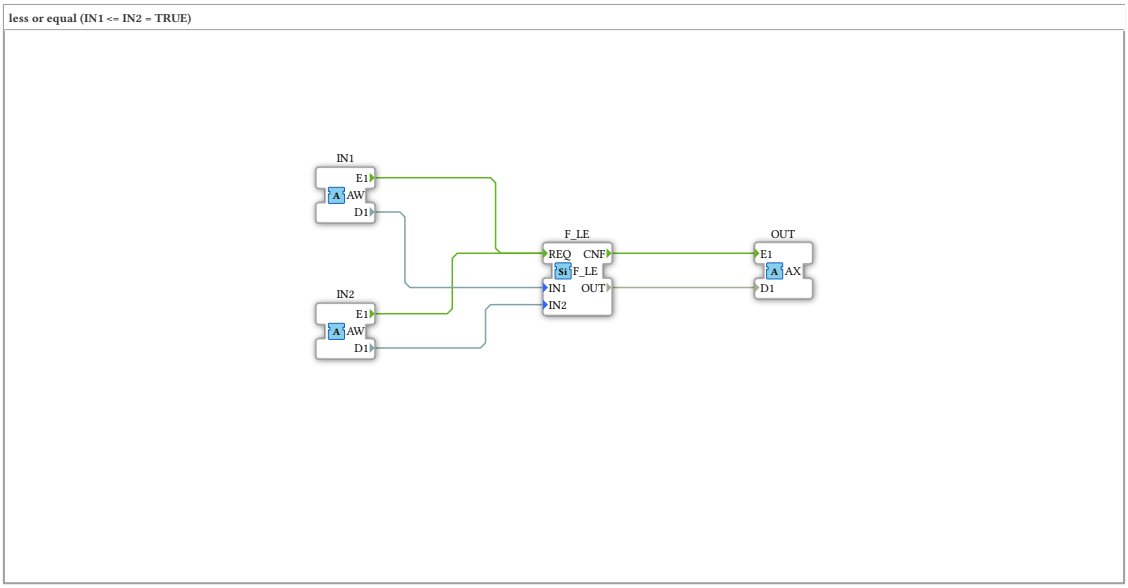
Der **AW_GT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `WORD`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AW_LE

Input value 1 -- left operand (IN1 <= IN2) – *AW*
Input value 2 -- right operand (IN1 <= IN2) – *AW*



AX – IN1 <= IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AW_LE** führt einen Vergleich zweier Werte vom Typ `WORD` (16-Bit-Bitmuster (Wort)) durch und prüft, ob der erste Wert (IN1) kleiner gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und

arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `WORD`, 16-Bit-Bitmuster (Wort)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 <= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AW` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AW` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AW` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LE` (Typ `iec61131::comparison::F_LE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AW`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than or equal to“) für den Datentyp `WORD`.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Statuswörter gegen einen Referenz- oder Schwellwert vom Typ `WORD` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `WORD` die Bedingung „kleiner gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `WORD`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

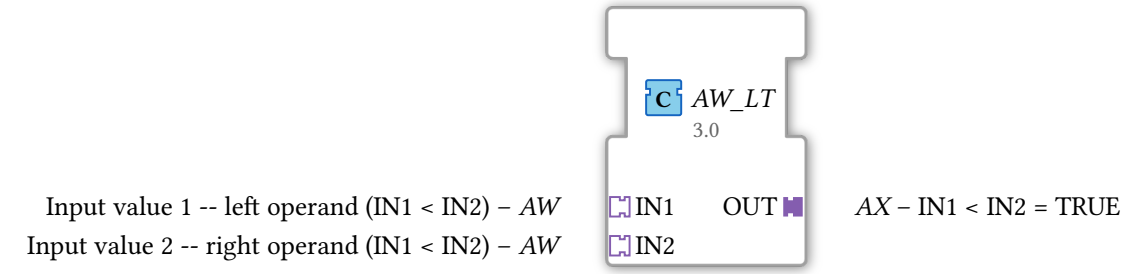
- **AW_GE**: Prüft auf „größer gleich“ – invers zur Funktion von AW_LE.
- **AW_EQ**: Prüft auf Gleichheit der Werte.
- **AW_LT**: Führt den Vergleich „kleiner als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version AW_LE ist.

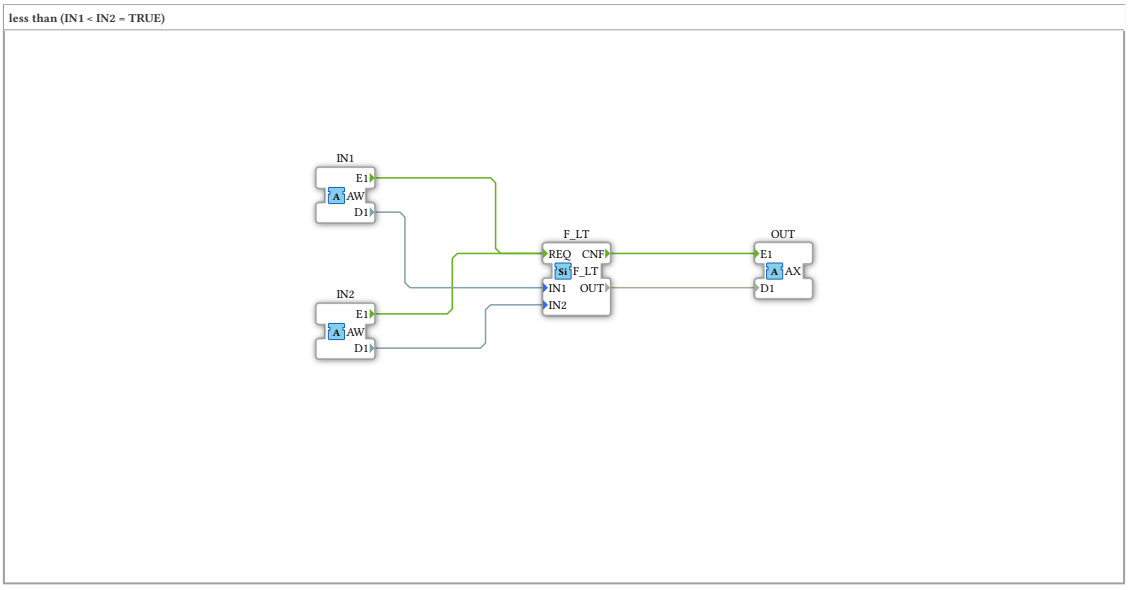
Der AW_LE ergänzt die AW-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

Der **AW_LE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `WORD`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AW_LT





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AW_LT** führt einen Vergleich zweier Werte vom Typ `WORD` (16-Bit-Bitmuster (Wort)) durch und prüft, ob der erste Wert (IN1) kleiner als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `WORD`, 16-Bit-Bitmuster (Wort)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 < IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AW` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AW` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AW` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LT` (Typ `iec61131::comparison::F_LT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AW`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than“) für den Datentyp `WORD`.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Statuswörter gegen einen Referenz- oder Schwellwert vom Typ `WORD` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `WORD` die Bedingung „kleiner als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `WORD`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AW_GT**: Prüft auf „größer als“ – invers zur Funktion von AW_LT.
- **AW_EQ**: Prüft auf Gleichheit der Werte.
- **AW_LE**: Führt den Vergleich „kleiner gleich“ durch – die um den Gleichheitsfall erweiterte Version von AW_LT.

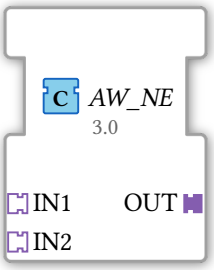
Der AW_LT ist die grundlegende „kleiner als“-Komponente ohne Gleichheitsfall, ergänzt durch AW_LE.

Fazit

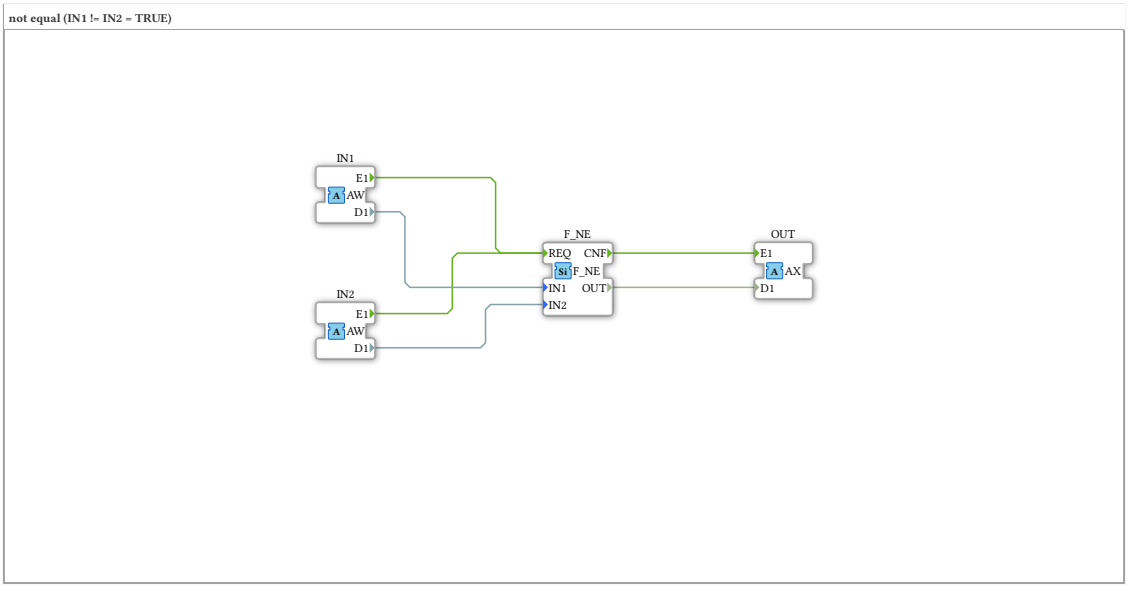
Der **AW_LT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `WORD`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AW_NE

Input value 1 -- left operand (IN1 != IN2) – *AW*
Input value 2 -- right operand (IN1 != IN2) – *AW*



AX – IN1 != IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AW_NE** führt einen Vergleich zweier Werte vom Typ `WORD` (16-Bit-Bitmuster (Wort)) durch und prüft, ob der erste Wert (IN1) ungleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `WORD`, 16-Bit-Bitmuster (Wort)).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 != IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AW` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AW` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AW` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_NE` (Typ `iec61131::comparison::F_NE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AW`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („not equal to“) für den Datentyp `WORD`.
- **Vergleichslogik**: Die Bitmuster werden dabei als vorzeichenlose Ganzzahl interpretiert und numerisch verglichen.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Statuswörter gegen einen Referenz- oder Schwellwert vom Typ `WORD` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `WORD` die Bedingung „ungleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `WORD`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AW_EQ**: Prüft auf „gleich“ – invers zur Funktion von AW_NE.
- **AW_EQ**: Prüft auf Gleichheit der Werte.

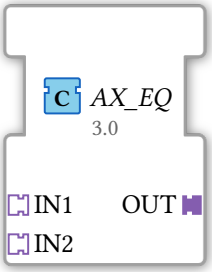
Der AW_NE ist Teil der vollständigen AW-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `WORD`.

Fazit

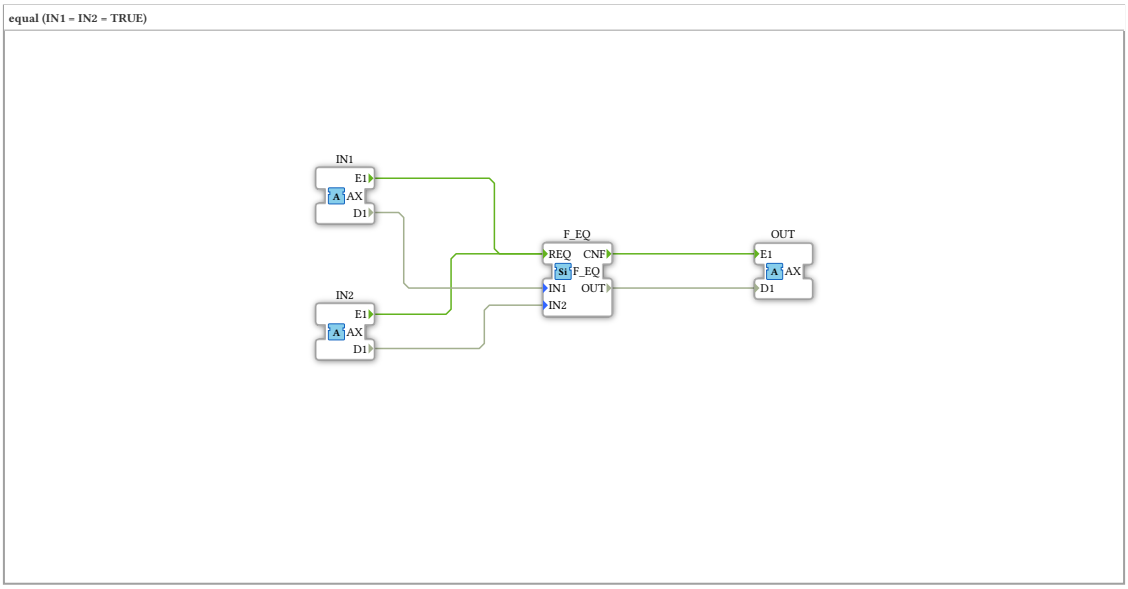
Der **AW_NE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `WORD`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AX_EQ

Input value 1 -- left operand (IN1 = IN2) – *AX*
Input value 2 -- right operand (IN1 = IN2) – *AX*



AX – IN1 = IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AX_EQ** führt einen Vergleich zweier Werte vom Typ `BOOL` (Boolescher Wert) durch und prüft, ob der erste Wert (IN1) gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. `FALSE` gilt dabei als der kleinere, `TRUE` als der größere Wert.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `BOOL`, Boolescher Wert).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 = IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AX` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AX` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AX` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_EQ` (Typ `iec61131::comparison::F_EQ`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AX`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („equal to“) für den Datentyp `BOOL`.
- **Vergleichslogik**: `FALSE` gilt dabei als der kleinere, `TRUE` als der größere Wert.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Freigabe- und Zustandsbits gegen einen Referenz- oder Schwellwert vom Typ `BOOL` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `BOOL` die Bedingung „gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `BOOL`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AX_NE**: Prüft auf „ungleich“ – invers zur Funktion von AX_EQ.
- **AX_NE**: Prüft auf Ungleichheit der Werte.

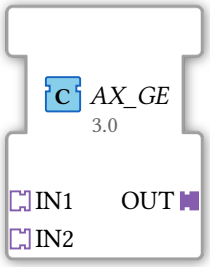
Der AX_EQ ist Teil der vollständigen AX-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `BOOL`.

Fazit

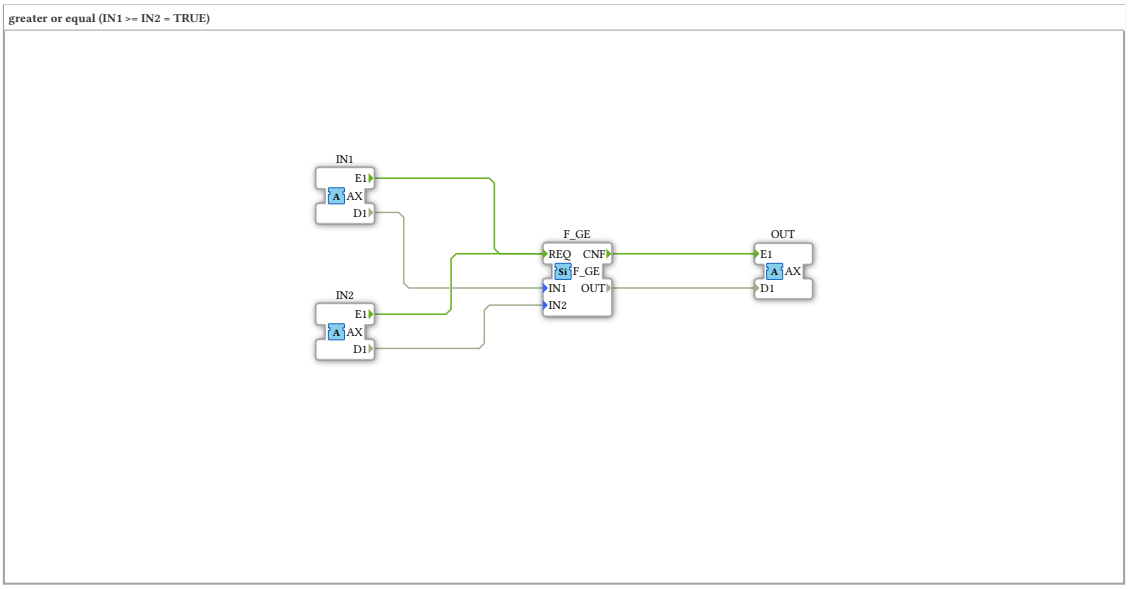
Der **AX_EQ** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `BOOL`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AX_GE

Input value 1 -- left operand (IN1 >= IN2) – *AX*
Input value 2 -- right operand (IN1 >= IN2) – *AX*



AX – IN1 >= IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AX_GE** führt einen Vergleich zweier Werte vom Typ `BOOL` (Boolescher Wert) durch und prüft, ob der erste Wert (IN1) größer gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. `FALSE` gilt dabei als der kleinere, `TRUE` als der größere Wert.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `BOOL`, Boolescher Wert).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 >= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AX` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AX` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AX` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GE` (Typ `iec61131::comparison::F_GE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AX`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than or equal to“) für den Datentyp `BOOL`.
- **Vergleichslogik**: `FALSE` gilt dabei als der kleinere, `TRUE` als der größere Wert.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Freigabe- und Zustandsbits gegen einen Referenz- oder Schwellwert vom Typ `BOOL` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `BOOL` die Bedingung „größer gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `BOOL`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AX_LE**: Prüft auf „kleiner gleich“ – invers zur Funktion von AX_GE.
- **AX_EQ**: Prüft auf Gleichheit der Werte.
- **AX_GT**: Führt den Vergleich „größer als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version AX_GE ist.

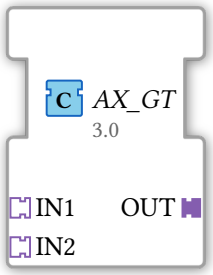
Der AX_GE ergänzt die AX-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

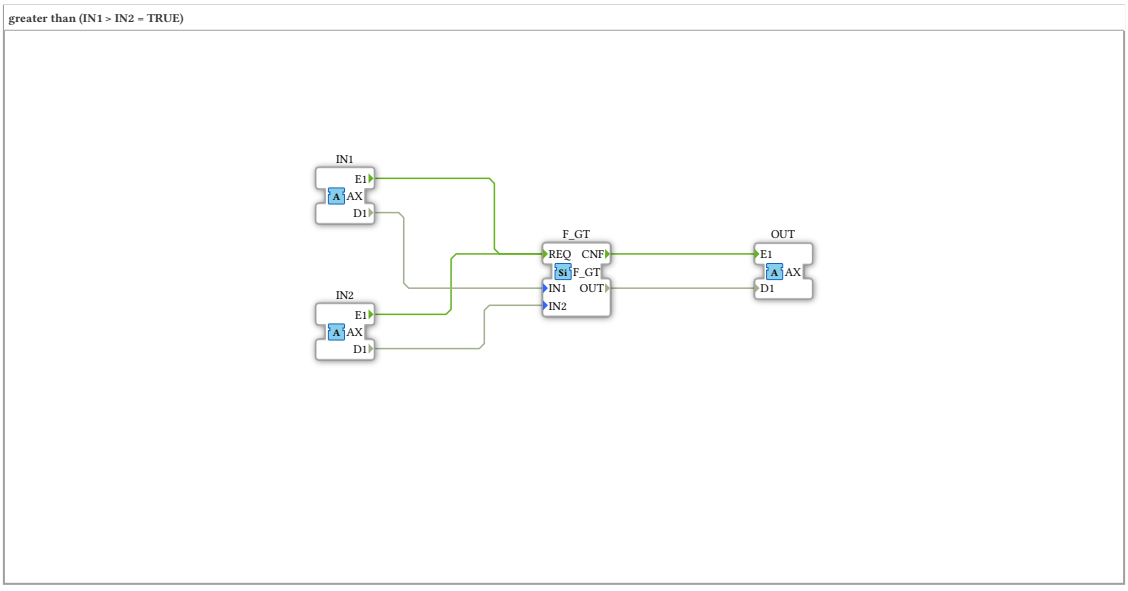
Der **AX_GE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `BOOL`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AX_GT

Input value 1 -- left operand (IN1 > IN2) – *AX*
Input value 2 -- right operand (IN1 > IN2) – *AX*



AX – IN1 > IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AX_GT** führt einen Vergleich zweier Werte vom Typ `BOOL` (Boolescher Wert) durch und prüft, ob der erste Wert (IN1) größer als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. `FALSE` gilt dabei als der kleinere, `TRUE` als der größere Wert.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `BOOL`, Boolescher Wert).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 > IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AX` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AX` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AX` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_GT` (Typ `iec61131::comparison::F_GT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AX`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („greater than“) für den Datentyp `BOOL`.
- **Vergleichslogik**: `FALSE` gilt dabei als der kleinere, `TRUE` als der größere Wert.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Freigabe- und Zustandsbits gegen einen Referenz- oder Schwellwert vom Typ `BOOL` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `BOOL` die Bedingung „größer als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `BOOL`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AX_LT**: Prüft auf „kleiner als“ – invers zur Funktion von AX_GT.
- **AX_EQ**: Prüft auf Gleichheit der Werte.
- **AX_GE**: Führt den Vergleich „größer gleich“ durch – die um den Gleichheitsfall erweiterte Version von AX_GT.

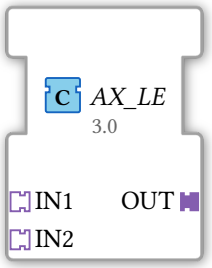
Der AX_GT ist die grundlegende „größer als“-Komponente ohne Gleichheitsfall, ergänzt durch AX_GE.

Fazit

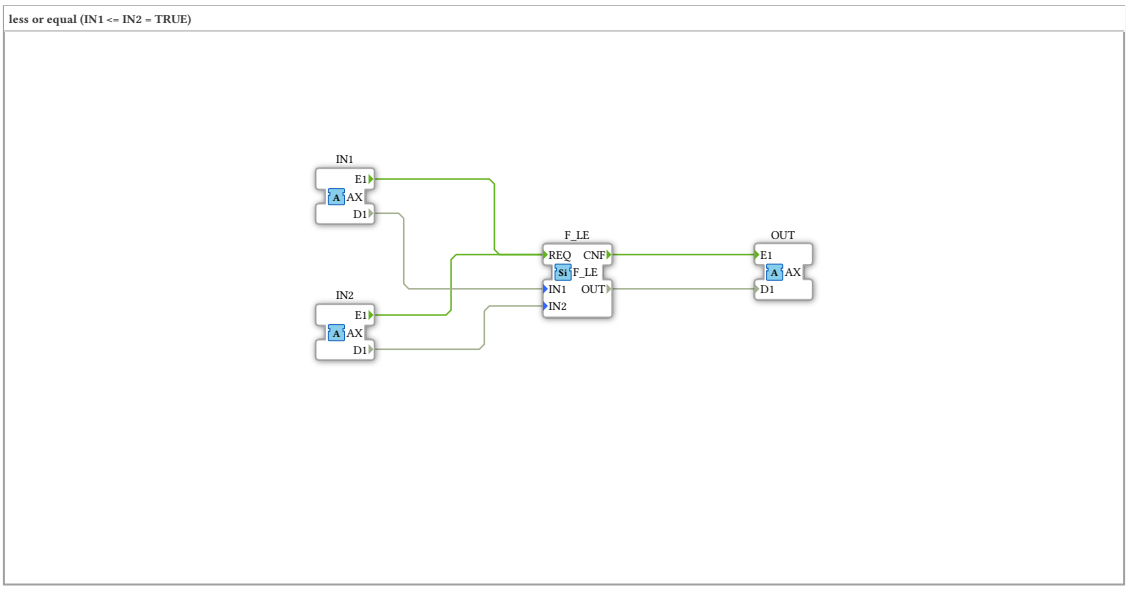
Der **AX_GT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `BOOL`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AX_LE

Input value 1 -- left operand (IN1 <= IN2) – *AX*
Input value 2 -- right operand (IN1 <= IN2) – *AX*



AX – IN1 <= IN2 = TRUE



• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AX_LE** führt einen Vergleich zweier Werte vom Typ `BOOL` (Boolescher Wert) durch und prüft, ob der erste Wert (IN1) kleiner gleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. `FALSE` gilt dabei als der kleinere, `TRUE` als der größere Wert.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `BOOL`, Boolescher Wert).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 <= IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AX` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AX` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AX` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LE` (Typ `iec61131::comparison::F_LE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AX`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than or equal to“) für den Datentyp `BOOL`.
- **Vergleichslogik**: `FALSE` gilt dabei als der kleinere, `TRUE` als der größere Wert.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Freigabe- und Zustandsbits gegen einen Referenz- oder Schwellwert vom Typ `BOOL` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `BOOL` die Bedingung „kleiner gleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `BOOL`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AX_GE**: Prüft auf „größer gleich“ – invers zur Funktion von AX_LE.
- **AX_EQ**: Prüft auf Gleichheit der Werte.
- **AX_LT**: Führt den Vergleich „kleiner als“ durch – die Basisvariante ohne Gleichheitsfall, deren erweiterte Version AX_LE ist.

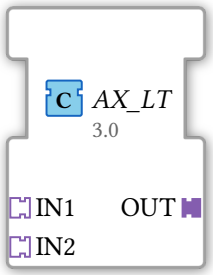
Der AX_LE ergänzt die AX-Vergleichsfamilie um den Grenzfall der Gleichheit.

Fazit

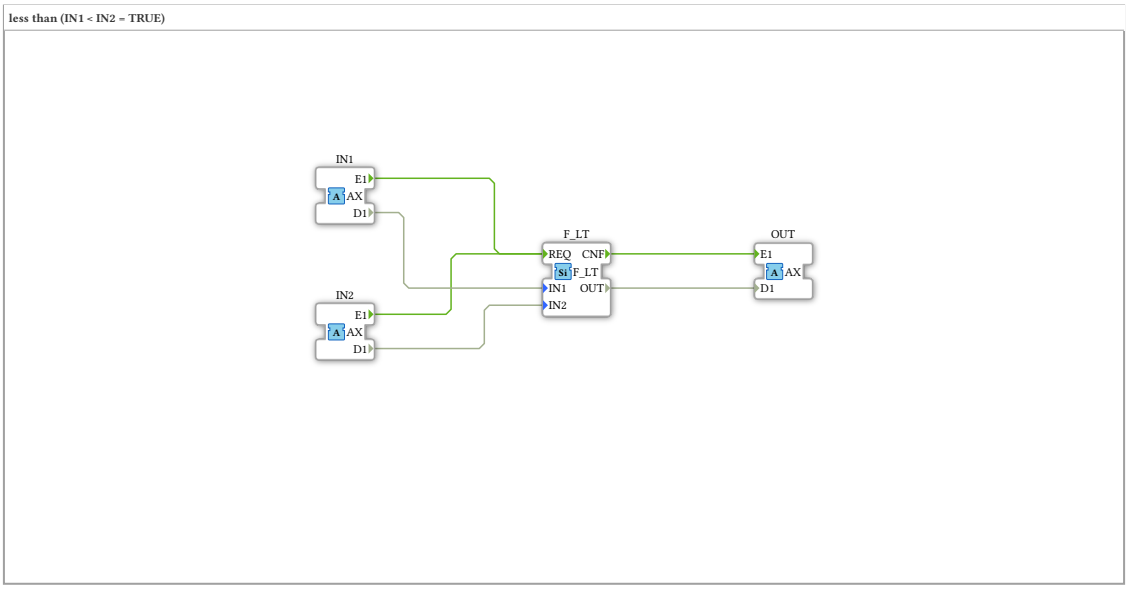
Der **AX_LE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `BOOL`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AX_LT

Input value 1 -- left operand (IN1 < IN2) – *AX*
Input value 2 -- right operand (IN1 < IN2) – *AX*



AX – IN1 < IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AX_LT** führt einen Vergleich zweier Werte vom Typ `BOOL` (Boolescher Wert) durch und prüft, ob der erste Wert (IN1) kleiner als der zweite Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. `FALSE` gilt dabei als der kleinere, `TRUE` als der größere Wert.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `BOOL`, Boolescher Wert).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 < IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AX` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AX` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AX` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_LT` (Typ `iec61131::comparison::F_LT`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AX`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („less than“) für den Datentyp `BOOL`.
- **Vergleichslogik**: `FALSE` gilt dabei als der kleinere, `TRUE` als der größere Wert.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Freigabe- und Zustandsbits gegen einen Referenz- oder Schwellwert vom Typ `BOOL` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `BOOL` die Bedingung „kleiner als“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `BOOL`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AX_GT**: Prüft auf „größer als“ – invers zur Funktion von AX_LT.
- **AX_EQ**: Prüft auf Gleichheit der Werte.
- **AX_LE**: Führt den Vergleich „kleiner gleich“ durch – die um den Gleichheitsfall erweiterte Version von AX_LT.

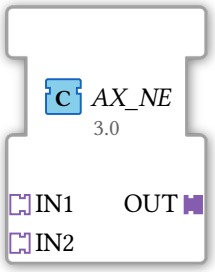
Der AX_LT ist die grundlegende „kleiner als“-Komponente ohne Gleichheitsfall, ergänzt durch AX_LE.

Fazit

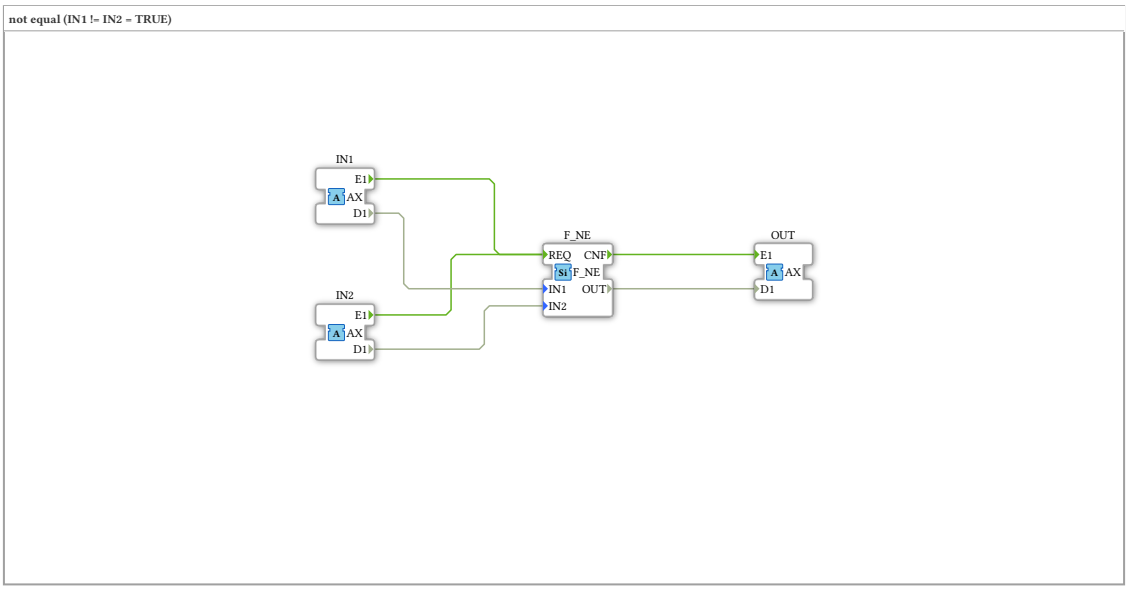
Der **AX_LT** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `BOOL`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

AX_NE

Input value 1 -- left operand (IN1 != IN2) – *AX*
Input value 2 -- right operand (IN1 != IN2) – *AX*



AX – IN1 != IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AX_NE** führt einen Vergleich zweier Werte vom Typ `BOOL` (Boolescher Wert) durch und prüft, ob der erste Wert (IN1) ungleich dem zweiten Wert (IN2) ist. Das Ergebnis wird als Boolescher Wert über den Ausgangsadapter bereitgestellt. Der Baustein ist ereignisgesteuert und arbeitet asynchron. `FALSE` gilt dabei als der kleinere, `TRUE` als der größere Wert.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Dateneingänge und -ausgänge, sondern kommuniziert ausschließlich über Adapter.

Ereignis-Eingänge

- **IN1.E1**: Ereignisseingang des Socket-Adapters IN1 – löst den Vergleich aus.
- **IN2.E1**: Ereignisseingang des Socket-Adapters IN2 – ebenfalls als Trigger für den Vergleich.

Ereignis-Ausgänge

- **OUT.E1**: Ereignisausgang des Plug-Adapters OUT – signalisiert, dass das Vergleichsergebnis am Datenausgang anliegt.

Daten-Eingänge

- **IN1.D1**: Erster zu vergleichender Wert (Datentyp `BOOL`, Boolescher Wert).
- **IN2.D1**: Zweiter zu vergleichender Wert (Datentyp wie IN1).

Daten-Ausgänge

- **OUT.D1**: Boolesches Ergebnis des Vergleichs – `TRUE`, wenn `IN1 != IN2`, sonst `FALSE`.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN1	`AX` (Socket)	Eingang	Adapter für den ersten Vergleichswert und das zugehörige Ereignis
IN2	`AX` (Socket)	Eingang	Adapter für den zweiten Vergleichswert und das zugehörige Ereignis
OUT	`AX` (Plug)	Ausgang	Adapter für das Vergleichsergebnis und das Ausgangsereignis

Die Adapter `AX` und `AX` sind unidirektionale Schnittstellen, die je einen Ereignis- und einen Datenkanal bereitstellen.

Funktionsweise

1. Der Baustein wartet auf ein Ereignis an einem der Socket-Adapter **IN1.E1** oder **IN2.E1**.
2. Sobald ein solches Ereignis eintrifft, wird es an den internen Funktionsbaustein `F_NE` (Typ `iec61131::comparison::F_NE`) weitergeleitet. Dort werden die aktuellen Werte von `IN1.D1` und `IN2.D1` verglichen.
3. Der interne Baustein erzeugt ein Ausgangsereignis (`CNF`) und gibt das Boolesche Ergebnis (`OUT`) aus.
4. Dieses Ereignis wird auf den Plug-Adapter **OUT.E1** propagiert, und der Datenwert `OUT.D1` wird auf `TRUE` oder `FALSE` gesetzt.
5. Die Ausgabe erfolgt nur bei einer Änderung der Eingangswerte oder bei einem erneuten Ereignis.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (`AX`/`AX`), wodurch er flexibel in verschiedene Kommunikationskontexte eingebunden werden kann.
- **Asynchrone Verarbeitung**: Die Vergleichsoperation wird durch ein Ereignis an einem der Eingangsadapter getriggert; das Ausgangssignal wird ereignisgesteuert geliefert.
- **Normkonformität**: Der Baustein implementiert die Vergleichsfunktion aus IEC 61131-3 („not equal to“) für den Datentyp `BOOL`.
- **Vergleichslogik**: `FALSE` gilt dabei als der kleinere, `TRUE` als der größere Wert.
- **Keine Zustandsautomaten**: Der Baustein ist rein kombinatorisch – das Ergebnis wird unmittelbar nach dem Ereignis gebildet, es gibt keinen internen Zustand.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten (ECC). Die interne Logik besteht aus einer direkten Verbindung der Adapter zu einem standardisierten Vergleichsbaustein. Die Ausgabe erfolgt sofort bei Ereigniseingang.

Anwendungsszenarien

- **Grenzwertüberwachung**: Freigabe- und Zustandsbits gegen einen Referenz- oder Schwellwert vom Typ `BOOL` prüfen.
- **Steuerungslogik**: Auslösen einer Aktion, sobald zwei Werte vom Typ `BOOL` die Bedingung „ungleich“ erfüllen.
- **Datenfilterung**: Auswählen von Datensätzen oder Signalen, die eine Vergleichsbedingung auf `BOOL`-Basis erfüllen.

Vergleich mit ähnlichen Bausteinen

- **AX_EQ**: Prüft auf „gleich“ – invers zur Funktion von AX_NE.
- **AX_EQ**: Prüft auf Gleichheit der Werte.

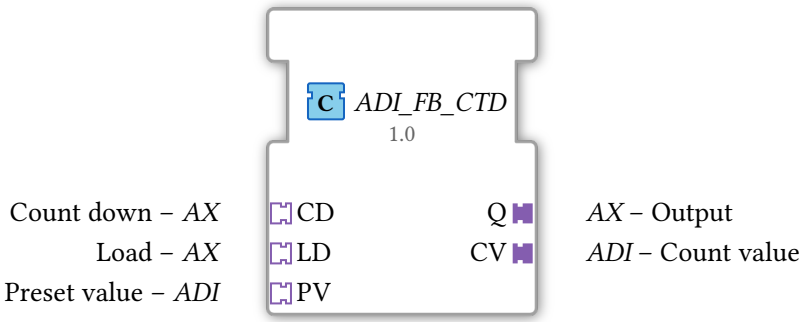
Der AX_NE ist Teil der vollständigen AX-Vergleichsfamilie (EQ/NE/GT/GE/LT/LE) für den Datentyp `BOOL`.

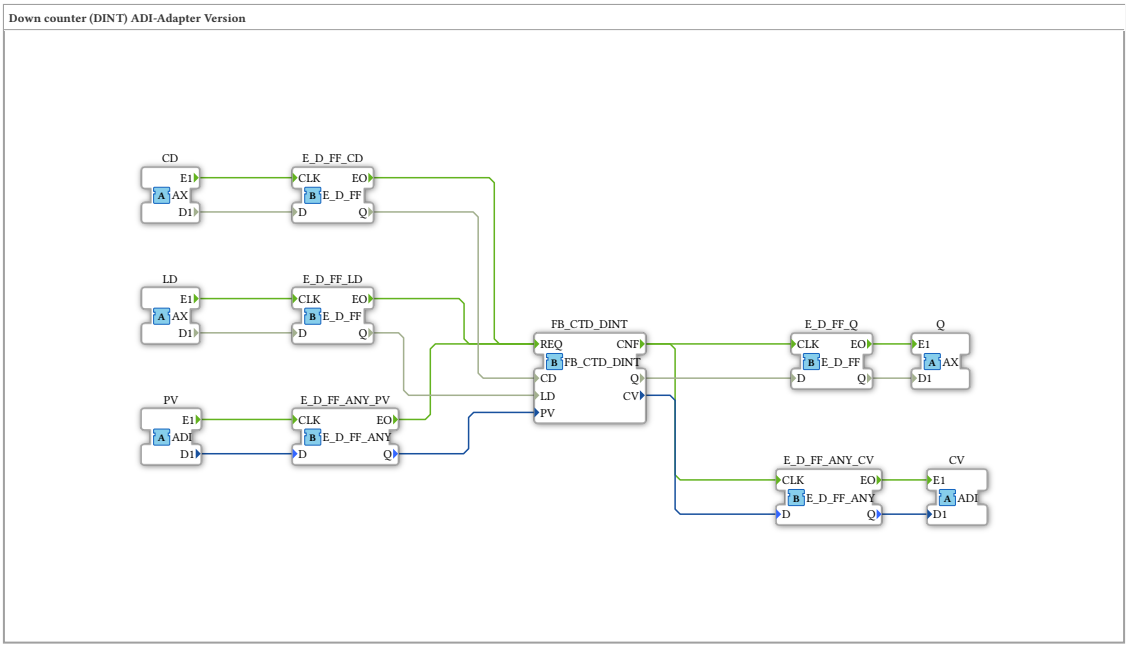
Fazit

Der **AX_NE** ist ein einfacher, aber wichtiger Vergleichsbaustein für IEC 61131-3 Steuerungsanwendungen mit dem Datentyp `BOOL`. Seine adapterorientierte Schnittstelle macht ihn vielseitig einsetzbar und ermöglicht eine klare Trennung von Ereignis- und Datenpfaden. Durch die ereignisgesteuerte Arbeitsweise eignet er sich besonders für Echtzeitsysteme, in denen Bedingungen sofort auf Werteänderungen reagiert werden müssen.

counters

ADI_FB_CTD





• – ○ • – ○ • – ○ •

Einleitung

Der **ADI_FB_CTD** ist ein Abwärtszähler (Down Counter) für Werte vom Typ `DINT`, der ausschließlich über ADI-Adapter kommuniziert. Er kapselt einen internen Standardzähler und ermöglicht das Herunterzählen, Laden eines Vorgabewerts sowie die Ausgabe des aktuellen Zählerstands – alles über einheitliche Adapterschnittstellen.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine direkten Ereignis-Eingänge. Ereignisse werden über die Adapter-Sockets **CD**, **LD** und **PV** bereitgestellt.

Ereignis-Ausgänge

Name	Typ	Beschreibung
CNF	Event	Bestätigungssignal, das nach jeder Verarbeitung eines eingehenden Ereignisses ausgegeben wird.

Daten-Eingänge

Keine direkten Daten-Eingänge; Daten werden über die Adapter-Sockets bereitgestellt.

Daten-Ausgänge

Keine direkten Daten-Ausgänge; Daten werden über die Adapter-Plugs bereitgestellt.

Adapter

Name	Typ	Richtung	Beschreibung
CD	`adapter::types::unidirectional::AX`	Socket (Eingang)	Ereignis zum Herunterzählen; der Datenwert (D1) wird als CD-Impuls an den internen Zähler weitergegeben.
LD	`adapter::types::unidirectional::AX`	Socket (Eingang)	Ereignis zum Laden des Zählers mit dem aktuellen Wert von **PV** der

			Datenwert (D1) signalisiert den Ladevorgang.
PV	adapter::types::unidirectional::ADI	Socket (Eingang)	Vorgabewert (Preset Value) als DINT; das Ereignis (E1) löst eine Aktualisierung des internen PV-Registers aus.
Q	adapter::types::unidirectional::AX	Plug (Ausgang)	Ausgangssignal, das anzeigt, ob der Zählerstand den Wert Null erreicht hat (z. B. TRUE bei Null).
CV	adapter::types::unidirectional::ADI	Plug (Ausgang)	Aktueller Zählerstand als DINT; wird bei jeder Aktualisierung ausgegeben.

Funktionsweise

Der ADI_FB_CTD verwendet einen internen IEC-61131-Zähler (FB_CTD_DINT). Die gesamte Kommunikation erfolgt über ADI-Adapter, die ein Ereignis (E1) und einen Datenwert (D1) kombinieren:

- Ein Ereignis am **CD**-Socket veranlasst den internen Zähler, den aktuellen Wert um **1** zu dekrementieren.
- Ein Ereignis am **LD**-Socket lädt den Zähler auf den Wert, der zuletzt über **PV** bereitgestellt wurde.
- Ein Ereignis am **PV**-Socket aktualisiert den Vorgabewert im internen Register; der Zähler selbst wird dabei nicht verändert. Erst ein nachfolgendes LD-Ereignis übernimmt den neuen Wert.

Alle eingehenden Ereignisse (CD, LD und PV) werden auf den gemeinsamen REQ-Eingang des internen Zählers geführt. Der interne Baustein unterscheidet anhand der Datenwerte, ob gezählt oder geladen werden soll. Nach Abschluss der Aktion wird das Ereignis CNF ausgelöst, das gleichzeitig die Ausgangsadapter **Q** und **CV** bedient und den CNF-Ereignisausgang setzt.

Technische Besonderheiten

- **Immer aktive Ausgabe:** Der FB gibt bei **jeder** Aktualisierung (durch CD, LD oder PV) ein Ereignis auf dem **Q**-Adapter aus – auch wenn sich der zugehörige Datenwert nicht ändert. Ist eine auslösende Ausgabe nur bei tatsächlicher Zustandsänderung erwünscht, muss ein Filtrier-Baustein wie AX_D_FF vorgeschaltet werden.
- **Adapter-basierte Schnittstelle:** Alle Ein- und Ausgänge sind als ADI-Adapter realisiert. Dies ermöglicht eine lose Kopplung und einfache Wiederverwendbarkeit in modularen Steuerungsarchitekturen.
- **Interner Standardzähler:** Die Zähllogik basiert auf einem etablierten IEC-61131-Baustein und ist daher robust und zuverlässig.

Zustandsübersicht

Der FB besitzt keinen eigenen Zustandsautomaten. Sein Verhalten wird durch den internen FB_CTD_DINT bestimmt, der sich in folgenden Betriebsarten befinden kann:

- **Laden:** Bei einem Ereignis am **LD**-Adapter wird der Zähler auf den aktuellen PV-Wert gesetzt.
- **Zählen:** Bei einem Ereignis am **CD**-Adapter wird der Zähler dekrementiert.
- **Warten:** Kein Ereignis liegt an; der Zähler behält seinen aktuellen Stand.

Nach jeder Verarbeitung ist der FB sofort bereit für das nächste Ereignis (keine Blockierung).

Anwendungsszenarien

- **Werkstückzählung:** Ein Sensor meldet jedes durchlaufende Werkstück als Ereignis auf **CD**. Der Zähler dekrementiert von einem vorgegebenen Startwert (**PV**) bis auf Null. Bei Erreichen von Null kann über **Q** ein Signal (z. B. „Behälter voll“) ausgegeben werden.
- **Zeitverzögerung:** In Kombination mit einem Taktgeber fungiert der Zähler als Abwärtstimer, dessen Restzeit über **CV** ausgelesen wird.
- **Steuerung von Abläufen:** Der aktuelle Zählerstand (**CV**) dient als Parameter für nachgeschaltete Schritte (z. B. Positionierung, Mengensteuerung).

Vergleich mit ähnlichen Bausteinen

- **CTU (Aufwärtszähler)** – zählt nur aufwärts; ADI_FB_CTD nur abwärts.

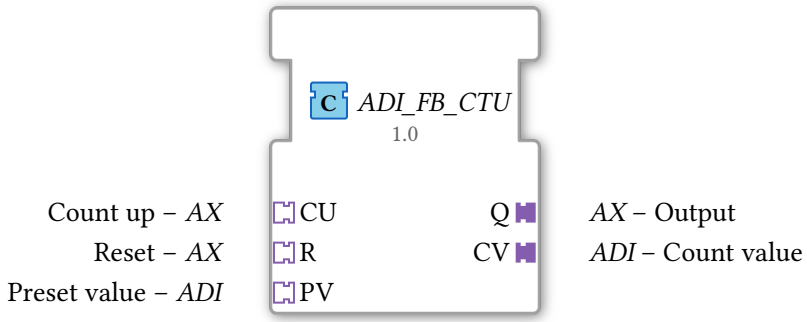
- **CTUD (Aufwärts/Abwärtszähler)** – kann in beide Richtungen zählen; ADI_FB_CTD ist auf eine Richtung spezialisiert.
- **Standard-IEC-61499-Zähler** – verwenden separate Ereignis- und Datenports, während ADI_FB_CTD die Schnittstellen über Adapter vereinheitlicht.
- **ADI_FB_CTU** – strukturell identisch, jedoch mit Aufwärtszähllogik (ebenfalls als Adapterversion).

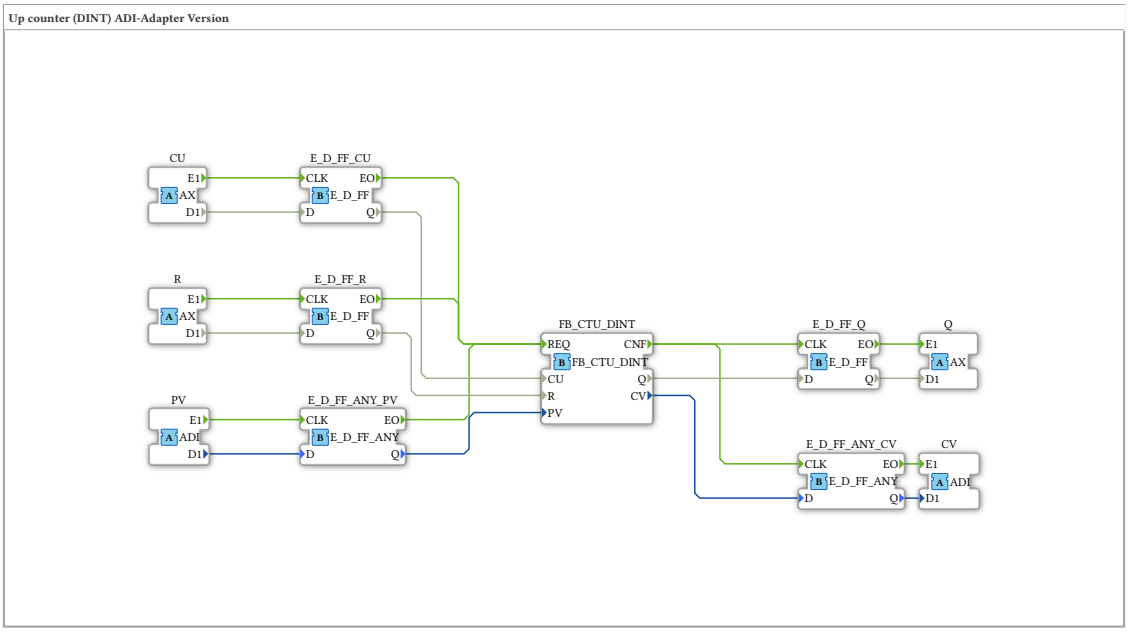
Der Vorteil der Adapterversion liegt in der modulareren, wiederverwendbaren Schnittstelle; der Nachteil ist der höhere Abstraktionsgrad und die erforderliche Filterung bei Bedarf.

Fazit

Der ADI_FB_CTD ist ein zuverlässiger Abwärtszähler für `DINT`-Werte, der sich durch seine durchgängig adapterbasierte Schnittstelle ideal in lose gekoppelte Steuerungsarchitekturen einfügt. Die Tatsache, dass der **Q**-Ausgang bei jeder Aktualisierung auslöst, muss bei der Systemauslegung berücksichtigt werden. Dank des internen Standardzählers ist die Zähllogik robust und erprobt – ein bewährter Baustein für Abwärtszählszenarien.

ADI_FB_CTU





• – ○ • – ○ • – ○ •

Einleitung

Der ADI_FB_CTU ist ein Aufwärtszähler (Up-Counter) für Ganzzahlen vom Typ DINT, dessen Ein- und Ausgänge über standardisierte Adapter (AX und ADI) bereitgestellt werden. Er kapselt den Standard-Funktionsblock `FB_CTU_DINT` und ermöglicht dessen Integration in modulare Adapter-basierte Systeme. Der Baustein eignet sich für allgemeine Zählaufgaben in der Automatisierungstechnik.

Schnittstellenstruktur

Der Funktionsblock besitzt keine direkten Ereignis- oder Datenschnittstellen, sondern ausschließlich Adapter für die Anbindung. Die nachfolgende Tabelle erläutert die verfügbaren Adapter, deren Typ und Bedeutung.

Richtung	Name	Adapter-Typ	Beschreibung
Eingang (Socket)	`CU`	`AX`	Zählimpuls-Eingang (Ereignis + Daten)
Eingang (Socket)	`R`	`AX`	Rücksetz-Eingang (Ereignis + Daten)
Eingang (Socket)	`PV`	`ADI`	Vorgabewert (Preset Value) für den Vergleich
Ausgang (Plug)	`Q`	`AX`	Ausgangssignal – aktiv, wenn Zählerstand ≥ PV
Ausgang (Plug)	`CV`	`ADI`	Aktueller Zählerstand

Die Adapter `AX` (Ereignis-Adapter) und `ADI` (Daten-Adapter) sind unidirektional. Über die Adapter werden sowohl die Ereignisse als auch die zugehörigen Datenwerte übertragen.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge. Die Ereignisse werden über die Adapter `CU` und `R` (vom Typ `AX`) zugeführt.

Ereignis-Ausgänge

Ein direkter Ereignis-Ausgang `CNF` signalisiert die Bestätigung einer Verarbeitung. Zusätzlich wird über den Adapter `Q` (Typ `AX`) ein Ausgangsereignis bei jeder Aktualisierung ausgegeben.

Daten-Eingänge

Keine direkten Daten-Eingänge. Der Vorgabewert wird über den Adapter `PV` (Typ `ADI`) bereitgestellt.

Daten-Ausgänge

Keine direkten Daten-Ausgänge. Der aktuelle Zählerstand wird über den Adapter `CV` (Typ `ADI`) ausgegeben.

Adapter

Der Funktionsblock verwendet drei Sockets (Eingangsadapter) und zwei Plugs (Ausgangsadapter):

- **`CU` (Socket, `AX`)**: Zählimpuls – bei jedem Ereignis wird der interne Zähler inkrementiert.
- **`R` (Socket, `AX`)**: Reset – setzt den Zähler auf Null zurück.
- **`PV` (Socket, `ADI`)**: Preset Value – legt den Schwellwert fest, ab dem der Ausgang `Q` aktiv wird.
- **`Q` (Plug, `AX`)**: Ausgang – wird aktiv, sobald der Zählerstand den Wert von `PV` erreicht oder überschreitet.
- **`CV` (Plug, `ADI`)**: Aktueller Zählerstand – kann von nachgeschalteten Bausteinen gelesen werden.

Funktionsweise

Der ADI_FB_CTU realisiert einen einfachen Aufwärtszähler mit Vorgabewert-Vergleich.

- Bei jedem Ereignis am Eingang `CU` wird der interne Zähler um 1 erhöht.
- Ein Ereignis am Eingang `R` setzt den Zähler auf 0 zurück.
- Der Vorgabewert `PV` wird beim Eintreffen eines Ereignisses am Eingang `PV` aktualisiert.

Nach jeder Verarbeitung (unabhängig davon, ob `CU`, `R` oder `PV` ausgelöst hat) wird das Bestätigungsereignis `CNF` ausgegeben. Gleichzeitig wird auch der Ausgangsadapter `Q` mit einem Ereignis bedient und der aktuelle Zählerstand über den Adapter `CV` bereitgestellt.

> **Wichtig:** Der Baustein feuert das Ausgangsereignis `Q.E1` bei **jeder** Aktualisierung – also auch bei Reset oder Vorgabewert-Änderung – nicht nur bei einer Änderung des Zählerstands. Soll das Ausgangssignal nur bei einer tatsächlichen Änderung des Vergleichsergebnisses ausgelöst werden, empfehlen die Entwickler die Verwendung eines AX_D_FF als nachgeschalteten Filter.

Intern wird ein Standard-Funktionsblock `FB_CTU_DINT` verwendet, dessen Ein- und Ausgänge über die Adapter verdrahtet sind. Der Zählwert ist vom Typ `DINT` (32-Bit-Ganzzahl).

Technische Besonderheiten

- **Adapter-basierte Schnittstelle** – ermöglicht eine lose Kopplung und einfache Integration in Adapter-basierte Architekturen (z. B. gemäß IEC 61499).
- **Unidirektionale Adapter** – die Adapter `AX` und `ADI` übertragen jeweils nur in eine Richtung.
- **Bestätigungsereignis `CNF`** – jedes Ereignis an einem Eingang löst eine sofortige Bestätigung aus.
- **Keine Flankenerkennung** – der Baustein reagiert auf jedes Ereignis, nicht auf steigende oder fallende Flanken eines digitalen Signals.
- **Hinweis im Quelltext** – die häufige Ausgabe des `Q.E1`-Ereignisses kann bei zeitkritischen Anwendungen zu unnötigen Lasten führen; ggf. ist eine Filterung erforderlich.

Zustandsübersicht

Der Funktionsblock besitzt einen einzigen internen Zustand: den aktuellen Zählerstand (Initialwert = 0). Abhängig von den eingehenden Ereignissen ergeben sich folgende Zustandsübergänge:

Ereignis	Bedingung	Neuer Zustand (Zähler)	Ausgabe
`CU`	–	Zähler + 1	`CNF`, `Q.E1`, `CV`
`R`	–	0	`CNF`, `Q.E1`, `CV`
`PV`	–	unverändert	`CNF`, `Q.E1`, `CV` (PV-Wert wird intern gespeichert)

Der Ausgang `Q` (über den Adapter) wird gesetzt, sobald `Zähler ≥ PV` ist. Der aktuelle Wert von `Q` wird bei jeder Ausgabe mitgeliefert.

Anwendungsszenarien

- **Ereigniszählung** – Zählen von Impulsen, z. B. Teiledurchlauf, Maschinenzyklen.
- **Füllstandüberwachung** – Erfassen der Anzahl von Behältern oder Chargen.
- **Produktionssteuerung** – Auslösen einer Aktion, wenn eine bestimmte Stückzahl erreicht ist.
- **Zeitmessung** – in Kombination mit einem Impulsgeber kann die Anzahl der Impulse als Zeitmaß genutzt werden.
- **Adapter-basierte Automatisierungssysteme** – überall dort, wo eine einheitliche Adapter-Schnittstelle gefordert ist.

Vergleich mit ähnlichen Bausteinen

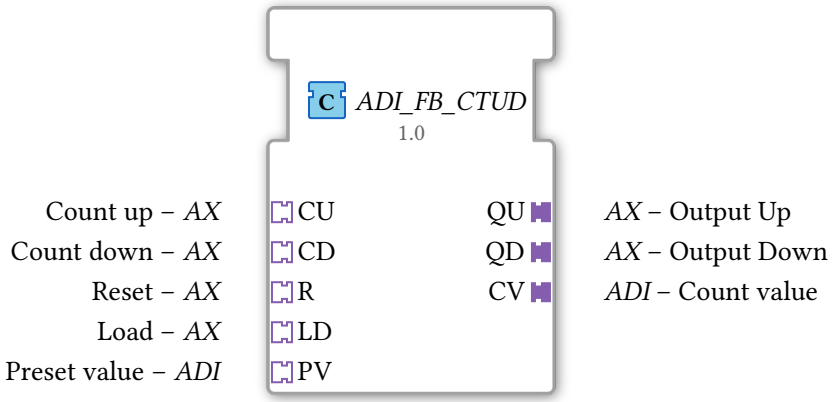
Baustein	Eigenschaften
`CTU` (Standard, ohne Adapter)	Gleiche Zählfunktion, aber mit direkten Ereignis- und Dateneingängen/-ausgängen. Einfacher in klassischen IEC-61499-Netzwerken nutzbar.
`ADI_FB_CTUD`	Auf-/Abwärtszähler, ebenfalls Adapter-basiert. Bietet zusätzlich einen Abwärtszähleingang.
`FB_CTU_DINT` (intern)	Dieselbe Zähllogik, jedoch ohne Adapterkapselung. Die Adapterversion bietet eine einheitliche, modulare Schnittstelle.
`CTU` mit AX-Filter	Wenn eine Flankenauswertung erforderlich ist, kann durch Vorschalten eines AX_D_FF eine reine Änderungserkennung realisiert werden.

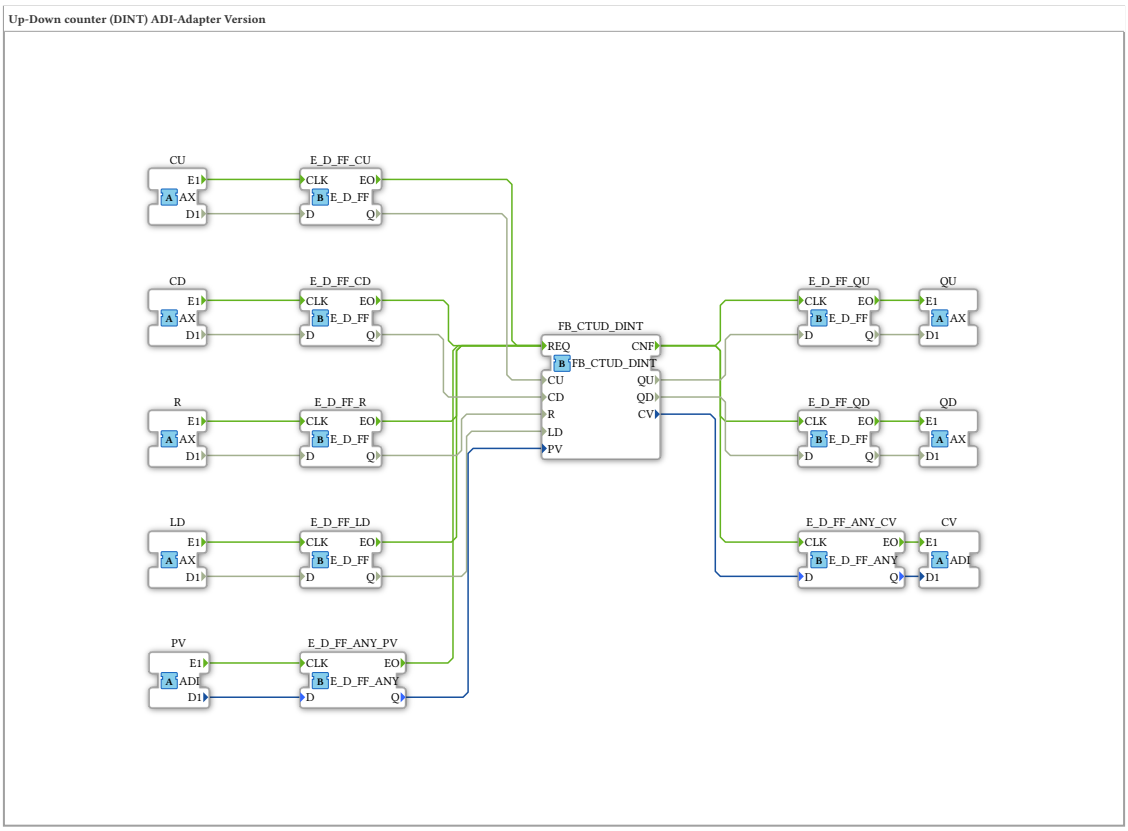
Der ADI_FB_CTU ist als „Wrapper“ für den Standardzähler konzipiert und erleichtert die Wiederverwendung in Adapter-basierten Frameworks.

Fazit

Der ADI_FB_CTU ist ein flexibler, adapterbasierter Aufwärtszähler für DINT-Werte. Er kapselt die bewährte Zähllogik des `FB_CTU_DINT` und stellt sie über standardisierte Adapter (AX und ADI) bereit. Die Besonderheit, dass Ausgangsereignisse bei jeder Aktualisierung generiert werden, sollte bei der Systemauslegung berücksichtigt werden. Der Baustein eignet sich hervorragend für modulare, erweiterbare Automatisierungslösungen, in denen eine einheitliche Adapter-Schnittstelle gefordert wird.

ADI_FB_CTUD





• – ○ • – ○ • – ○ •

Einleitung

Der **ADI_FB_CTUD** ist ein aufwärts/abwärts zählender Funktionsbaustein für ganzzahlige Werte (DINT). Er realisiert einen Vorwärts-/Rückwärtszähler, der über standardisierte **Adapterschnittstellen** gesteuert wird. Der Baustein kapselt die IEC-61131-3-Komponente `FB_CTUD_DINT` und stellt deren Funktionalität über ereignisgesteuerte Adapter bereit.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine direkten Ereigniseingänge. Die Ansteuerung erfolgt ausschließlich über die **Sockets**:

- **CU (Count Up)** – Ereignis zum Inkrementieren des Zählwerts
- **CD (Count Down)** – Ereignis zum Dekrementieren des Zählwerts
- **R (Reset)** – Ereignis zum Zurücksetzen des Zählwerts auf Null

- **LD (Load)** – Ereignis zum Laden des Presetwerts
- **PV (Preset Value)** – Ereignis zum Setzen des Presetwerts (Datenwert)

Diese Adapter liefern sowohl ein Ereignis als auch einen Datenwert (bei AX-Adapter über den Datenausgang `D1`).

Ereignis-Ausgänge

- **CNF** – Bestätigungsereignis, das nach jeder Verarbeitung eines der Eingangsereignisse ausgegeben wird.

Daten-Eingänge

Die Datenwerte der Eingangsadapter werden über die Verbindungen des internen FB `FB_CTUD_DINT` verarbeitet. Die Adapter liefern folgende Daten:

- **CU.D1** – Zählrichtung (Bool, 1 = Aufwärtszählen)
- **CD.D1** – Zählrichtung (Bool, 1 = Abwärtszählen)
  (Hinweis: Im Standard-CTUD-DINT wird üblicherweise nur ein Zählereignis mit Richtungsauswahl verwendet; hier werden CU/CD als separate Ereignisse behandelt.)
- **R.D1** – Reset-Signal (Bool)
- **LD.D1** – Load-Signal (Bool)
- **PV.D1** – Presetwert (DINT)

Daten-Ausgänge

Die Ausgangsdaten werden über die Plug-Adapter bereitgestellt:

- **QU.D1** – Zähler hat aufwärtsgeschaltet (Bool, nach CU-Ereignis)
- **QD.D1** – Zähler hat abwärtsgeschaltet (Bool, nach CD-Ereignis)
- **CV.D1** – Aktueller Zählwert (DINT)

Adapter

Der Baustein verwendet drei verschiedene Adaptertypen:

- **unidirectional::AX** – für Ereignis- und Bool-Daten (CU, CD, R, LD als Sockets; QU, QD als Plugs)
- **unidirectional::ADI** – für Wertübergabe (PV als Socket, CV als Plug)

Die Adapter sind so konzipiert, dass sie sowohl das Ereignis als auch den zugehörigen Datenwert in einem Zug übertragen.

Funktionsweise

Der interne FB `FB_CTUD_DINT` realisiert die klassische Aufwärts-/Abwärtszählerlogik:

- Bei einem Ereignis auf **CU** wird der Zählwert um 1 erhöht, sofern `CU.D1` = TRUE (oder das Ereignis allein als Aufwärtsimpuls gilt).
- Bei einem Ereignis auf **CD** wird der Zählwert um 1 verringert, sofern `CD.D1` = TRUE.
- Ein Ereignis auf **R** setzt den Zählwert auf 0 zurück.
- Ein Ereignis auf **LD** lädt den aktuellen Zählwert mit dem über **PV** übergebenen Wert.
- Nach jeder Verarbeitung wird **CNF** ausgegeben, sowie die Ereignisse **QU.E1** und **QD.E1** über die entsprechenden Adapter, wenn sich der Zählwert geändert hat.

Wichtig: Der Baustein feuert die Ausgangsereignisse (QU.E1, QD.E1) bei *jedem* Update (CU, CD, R, LD, PV) – nicht nur bei einer tatsächlichen Wertänderung. Soll eine reine Flankenauswertung (on-change) erfolgen, wird der Einsatz eines **AX_D_FF** als Filter empfohlen.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle:** Alle Ein- und Ausgänge erfolgen über standardisierte unidirektionale Adapter, was die Wiederverwendbarkeit und Kapselung erhöht.
- **IEC-61131-3-Kompatibilität:** Der interne Zähler entspricht der Norm und erlaubt einfache Migration zwischen verschiedenen Steuerungssystemen.
- **Immer aktive Ereignisse:** Wie oben beschrieben, werden Ausgangsereignisse bei jedem Eingangsereignis generiert – dies kann in zeitkritischen Anwendungen zu hoher Buslast führen.

- **Kein Zustandsautomat auf oberster Ebene:** Der FB besitzt kein eigenes ECC; die gesamte Logik wird durch den internen FB ausgeführt.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten. Die Verarbeitung erfolgt strikt ereignisgesteuert:

1. Warten auf ein Ereignis an einem der Sockets.
2. Verarbeitung des zugehörigen Datenwerts (sofern vorhanden) und des internen Zählerzustands.
3. Ausgabe von **CNF** sowie ggf. **QU.E1** und **QD.E1**.
4. Zurück zu Schritt 1.

Anwendungsszenarien

- **Zählen von Impulsen** in Fertigungsanlagen (z. B. Werkstückzähler).
- **Positionserfassung** mit Inkrementalgebern (Auf-/Abwärtszählung).
- **Stücklisten- und Bestandszählung** mit Rückstellmöglichkeit.
- **Ereignisgesteuerte Steuerungen** in der Automatisierungstechnik, bei denen Zählerstände über Adapter ausgetauscht werden.

Vergleich mit ähnlichen Bausteinen

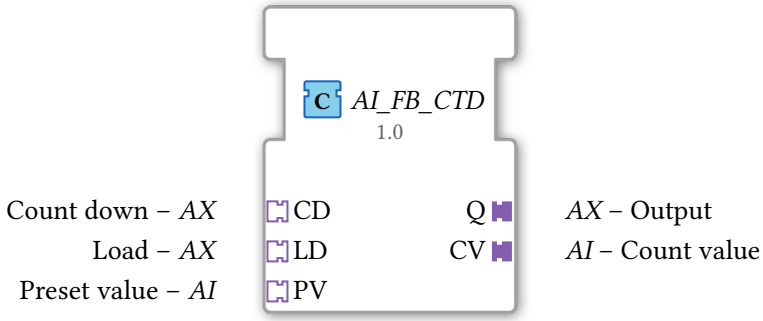
Eigenschaft	ADI_FB_CTUD	Standard CTUD (IEC 61131-3)
Schnittstelle	Adapter-basiert (AX/ADI)	Direkte Ein-/Ausgänge (Events, Daten)
Ereignisverhalten	Ausgang bei jedem Update	Meist Flanken- oder wertabhängig
Wiederverwendbarkeit	Leicht durch Adapter-Kapselung	Abhängig von der konkreten Steuerung
Kompatibilität	IEC-61131-3-konform (intern)	Direkte Normimplementierung
Filterung	Empfehlung: AX_D_FF	Teilweise integriert

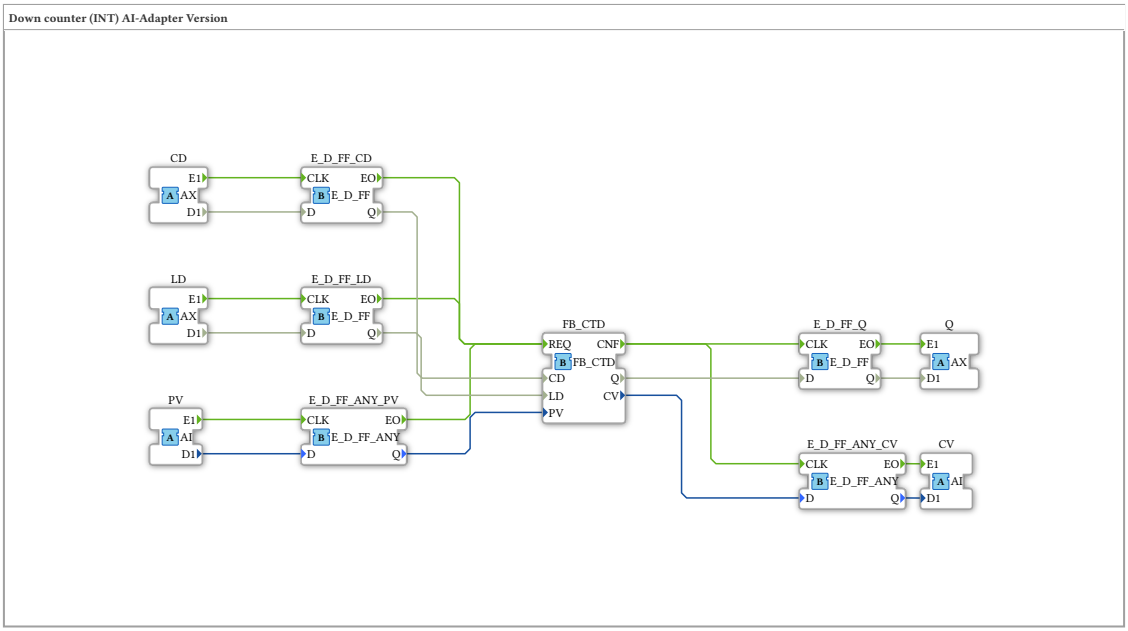
Der ADI_FB_CTUD bietet eine saubere adapterbasierte Kapselung, erfordert jedoch bei Bedarf nach on-change-Triggering zusätzliche Maßnahmen.

Fazit

Der **ADI_FB_CTUD** ist ein flexibler und normenkonformer Aufwärts-/Abwärtszähler mit DINT-Wertebereich, der sich durch seine moderne Adapterschnittstelle auszeichnet. Er eignet sich besonders für modulare Steuerungsarchitekturen, in denen Komponenten über standardisierte Adapter kommunizieren. Die Besonderheit des immer aktiven Ereignisausgangs sollte bei der Integration in Echtzeitsysteme berücksichtigt werden.

AI_FB_CTD





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AI_FB_CTD** ist ein nach IEC 61499-2 standardisierter Abwärtszähler (**Down Counter**) für Integer-Werte. Er nutzt durchgängig Adapter-Schnittstellen (AX, AI), um Ereignisse und Daten zu transportieren. Der Baustein ist als Wrapper um den klassischen IEC 61131-FB `FB_CTD` realisiert und ermöglicht eine modulare, adapterbasierte Verbindung in 4diac-IDE-Netzwerken. Jedes eingehende Ereignis über die Zählersteuerungen (CD, LD, PV) führt zur Aktualisierung des internen Zählers und löst ein Ausgangsereignis aus.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine direkten, separaten Ereignis-Eingänge. Die erforderlichen Ereignisse werden über die **Socket-Adapter** (CD, LD, PV) bereitgestellt – jedes dieser Module führt ein Ereignis (E1) mit sich, das die Zählerlogik triggert.

Socket	Ereignis (im Adapter)	Beschreibung
CD	E1	Zählereignis (Abwärtszähler)
LD	E1	Ladeereignis (Setzen auf PV)
PV	E1	Vorgabeaktualisierung

Ereignis-Ausgänge

Name	Typ	Beschreibung
CNF	Event	Bestätigungsereignis nach jeder Zähleraktualisierung (CD, LD, PV)

Zusätzlich werden die Ausgangsereignisse auch über die **Plug-Adapter** Q und CV (jeweils E1) weitergegeben.

Daten-Eingänge

Es gibt keine direkten Daten-Eingänge. Alle Datenwerte werden über die Socket-Adapter übertragen:

- **CD** liefert den Abwärtszählimpuls (D1) – logischer Wert, der den Zähler dekrementiert.
- **LD** liefert das Ladesignal (D1) – logischer Wert, der den Zähler auf den aktuellen PV-Wert setzt.
- **PV** liefert den Vorgabewert (D1) – Integer, der als neuer Zählerstand geladen wird.

Daten-Ausgänge

Es gibt keine direkten Daten-Ausgänge. Die Ausgangsdaten werden über die Plug-Adapter bereitgestellt:

- **Q** liefert den Zählerstatus (D1) – BOOL, TRUE wenn Zählerstand ≤ 0.
- **CV** liefert den aktuellen Zählerstand (D1) – INT.

Adapter

Bezeichnung	Richtung	Typ	Beschreibung
CD	Socket	`adapter::types::unidirectional::AX`	Zählereingang (Ereignis + Daten) für Abwärtszählen
LD	Socket	`adapter::types::unidirectional::AX`	Ladeeingang (Ereignis + Daten) zum Setzen auf PV
PV	Socket	`adapter::types::unidirectional::AI`	Vorgabeeingang (Ereignis + Daten) für den Preset-Wert
Q	Plug	`adapter::types::unidirectional::AX`	Ausgangssignal (Ereignis + Daten) – Zähler ≤ 0
CV	Plug	`adapter::types::unidirectional::AI`	Ausgangszählerwert (Ereignis + aktueller Zählerstand)

Funktionsweise

Der Baustein enthält intern einen `iec61131::counters::FB_CTD`, der die eigentliche Zähllogik implementiert. Die Funktion ist wie folgt:

1. Ein Ereignis an **CD** (über `CD.E1`) oder **LD** (`LD.E1`) oder **PV** (`PV.E1`) triggert den internen FB über seinen `REQ`-Ereigniseingang.
2. Der interne FB wertet die zugehörigen Daten aus:
   - Bei einem **CD**-Ereignis wird der Zähler um 1 dekrementiert (wenn das zugehörige Datum TRUE ist).
   - Bei einem **LD**-Ereignis wird der Zähler auf den aktuellen Wert von **PV** gesetzt (wenn das Ladedatum TRUE ist).
   - Bei einem **PV**-Ereignis wird der Vorgabewert intern aktualisiert (der Zähler bleibt unverändert).
3. Nach der Verarbeitung erzeugt der interne FB sein `CNF`-Ereignis. Dieses wird auf die externen Ausgänge verteilt:
   - `CNF` (direkter Event-Ausgang)
   - `Q.E1` (Ereignis des Plug-Adapters Q)
   - `CV.E1` (Ereignis des Plug-Adapters CV)
4. Zeitgleich werden die Daten:
   - `FB_CTD.Q` (BOOL) ⟶ `Q.D1`
   - `FB_CTD.CV` (INT) ⟶ `CV.D1` übertragen.

Hinweis: Das Ausgangsereignis wird **bei jeder Aktualisierung** (also bei CD, LD oder PV) ausgelöst. Eine reine Wertänderung ohne Ereignis (On-Change) ist nicht implementiert.

Technische Besonderheiten

- **Lizenz und Herkunft**: Der Baustein steht unter der **Eclipse Public License 2.0** (EPL 2.0). Er wurde als Lehrmaterial für die Meisterschulen am Ostbahnhof entwickelt.
- **Adapter-basierte Kommunikation**: Anders als klassische FB-Schnittstellen werden alle Ein- und Ausgänge über Adapter (AX, AI) realisiert. Dies ermöglicht eine flexible, hierarchische Verdrahtung und Wiederverwendung in komplexen Steuerungsarchitekturen.
- **On-Change-Filter**: Der Baustein feuert das Ausgangsereignis bei **jedem** Eingangsereignis, unabhängig von einer tatsächlichen Wertänderung. Soll eine Änderungsdetektion erfolgen, wird der Einsatz eines **AX_D_FF** (Differentiator-Flipflop) als Vorfilter empfohlen (siehe auch Copyright-Hinweis in der XML).
- **Interner FB**: Die Zähllogik basiert auf dem Standardbaustein `FB_CTD` aus IEC 61131 (adaptiert für 4diac). Dadurch ist das Verhalten deterministisch und kompatibel zu SPS-Programmen.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten. Die einzige innere Variable ist der **aktuelle Zählerwert** (Integer). Dieser wird durch die Ereignisse verändert:

- Initialwert: 0 (oder durch LD auf PV gesetzt)
- CD-Ereignis: Zähler = Zähler - 1 (wenn zugehöriges Datum TRUE, sonst keine Änderung)
- LD-Ereignis: Zähler = PV (wenn Ladedatum TRUE)
- PV-Ereignis: PV-Wert wird intern gespeichert, Zähler bleibt unverändert.

Eine grafische Zustandsmaschine ist daher nicht erforderlich.

Anwendungsszenarien

- **Bestandszähler**: Dekrementierung eines Lagerbestands bei Entnahme (CD), Neusetzen über LD nach Auffüllung.
- **Ereigniszähler mit Grenzwert**: Zählen von Ereignissen bis zum Erreichen von 0, dann Auslösen eines Signals (Q).
- **Zeit-/Impulssteuerung**: Verwendung als Pulszähler in Verbindung mit einem Taktgeber (CD) und Vorgabe der gewünschten Anzahl (PV).
- **Adapter-Integration**: In modularen Steuerungssystemen, bei denen Ereignisse und Daten über standardisierte Adapter-Schnittstellen gekapselt werden sollen.

Vergleich mit ähnlichen Bausteinen

Baustein	Zählrichtung	Schnittstellen	Besonderheit
AI_FB_CTD	Abwärts	Nur Adapter (AX, AI)	Ausgang bei jeder Aktualisierung
AI_FB_CTU	Aufwärts	Nur Adapter (AX, AI)	Aufwärtszähler, analoge Struktur
FB_CTD (Standard)	Abwärts	Direkte Events/Daten	Klassischer Variablenanschluss, ohne Adapter
FB_CTUD	Beide	Direkt	Kombinierter Auf-/Abwärtszähler, erweiterte Logik

Der **AI_FB_CTD** hebt sich durch die konsequente Adapter-Nutzung hervor, was ihn ideal für hierarchische und wiederverwendbare Steuerungsmodelle macht. Im Unterschied zum Standard-FB_CTD entfallen separate Daten- und Eventeingänge – alles wird über die Adapter geführt.

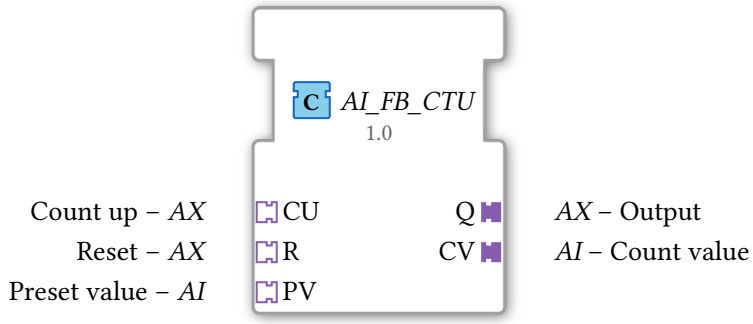
Fazit

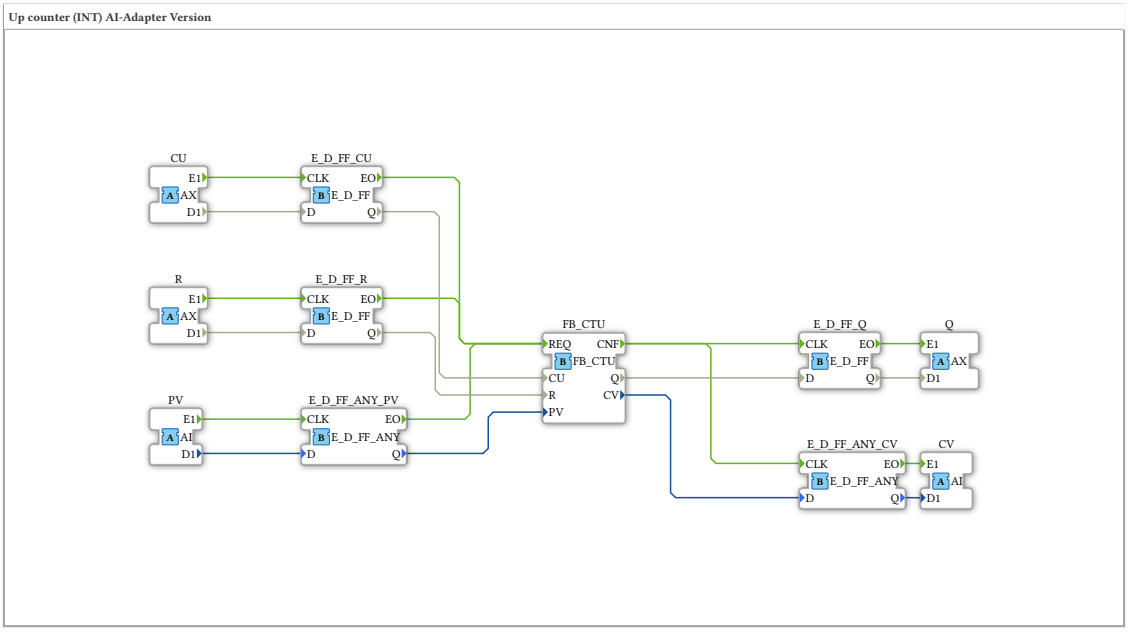
Der `AI_FB_CTD` ist ein flexibler, adapterbasierter Abwärtszähler für Integer-Werte, der die bewährte Zähllogik aus IEC 61131 in eine moderne, modulare Schnittstellenlandschaft überführt. Er eignet sich hervorragend für den Einsatz in verteilten Steuerungssystemen nach IEC 61499, bei denen Kapselung und Wiederverwendbarkeit im Vordergrund stehen. Die einfache Handhabung (drei Eingangsereignisse, zwei Ausgangsadapter) und die klare Dokumentation machen ihn zu einem soliden Grundbaustein für Zähleranwendungen. Empfohlen wird bei Bedarf an Änderungsdetektion die Vorschaltung eines `AX_D_FF`.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AI_FB_CTU





• – ◦ • – ◦ • – ◦ •

Einleitung

Der **AI_FB_CTU** ist ein Aufwärtszähler (Up Counter) für Ganzzahlen (INT), der die IEC 61131-3 Funktionalität eines CTU (Counter Up) in einer adapterbasierten Bauform kapselt. Er ist speziell für die Verwendung in der 4diac-IDE konzipiert und erlaubt eine modulare Anbindung über unidirektionale Adapter-Schnittstellen. Der Baustein feuert bei jeder Aktualisierung seiner Eingänge (CU, R, PV) ein Bestätigungsereignis, wodurch er sich für zeitgesteuerte oder ereignisgesteuerte Zählaufgaben eignet.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine direkten Ereignis-Eingänge. Die Ereignissteuerung erfolgt ausschließlich über die Adapter-Sockets **CU**, **R** und **PV**. Jeder dieser Sockets liefert ein Ereignis (E1), das den internen Ablauf auslöst.

- **CU.E1** – Zählimpuls (Ereignis vom counting-up Adapter)
- **R.E1** – Rücksetzen (Ereignis vom Reset Adapter)
- **PV.E1** – Vorgabe des Preset-Werts (Ereignis vom Preset-Value Adapter)

Ereignis-Ausgänge

- **CNF** (Typ: Event) – Bestätigungsereignis, das nach jeder erfolgreichen Verarbeitung aller drei möglichen Ereignisse ausgelöst wird.

Zusätzlich werden die Ausgabeadapter **Q** und **CV** mit demselben Ereignis bedient:

- **Q.E1** – Ereignis für den Ausgangsadapter (Zählerstand erreicht oder übersteigt den Preset-Wert)

- **CV.E1** – Ereignis für den aktuellen Zählwert

Daten-Eingänge

Alle Dateneingänge werden über die Adapter-Sockets bereitgestellt:

Adapter	Daten-Eingang	Typ	Beschreibung
CU.D1	CU	INT	Count Up – Zählimpuls (wird bei jedem Ereignis inkrementiert)
R.D1	R	INT	Reset – Wert auf den der Zähler zurückgesetzt wird (typischerweise 0)
PV.D1	PV	INT	Preset Value – Schwellwert, bei dessen Erreichen der Ausgang Q aktiv wird

Daten-Ausgänge

- **Q.D1** (über Adapter Q, Typ AX) – Ausgangssignal (BOOL), wird TRUE, wenn der Zählerstand ≥ PV ist.
- **CV.D1** (über Adapter CV, Typ AI) – Aktueller Zählerstand (INT).

Adapter

Name	Typ	Richtung	Beschreibung
CU	adapter::types::unidirectional::AX	Socket (Eingang)	Ereignis- und Datenadapter für den Zählimpuls
R	adapter::types::unidirectional::AX	Socket (Eingang)	Ereignis- und Datenadapter für das Rücksetzen
PV	adapter::types::unidirectional::AI	Socket (Eingang)	Datenadapter (nur Wert, kein Ereignis) für den Preset-Wert
Q	adapter::types::unidirectional::AX	Plug (Ausgang)	Ereignis- und Datenadapter für den Zählerausgang
CV	adapter::types::unidirectional::AI	Plug (Ausgang)	Datenadapter (nur Wert) für den aktuellen Zählerstand

Funktionsweise

Der **AI_FB_CTU** verwendet intern einen standardisierten IEC 61131-3 CTU-Baustein (`iec61131::counters::FB_CTU`). Bei jedem eingehenden Ereignis (CU.E1, R.E1 oder PV.E1) wird der interne CTU durch Aufruf seines REQ-Eingangs verarbeitet. Die Daten aus den Adaptern (CU.D1, R.D1, PV.D1) werden direkt an die entsprechenden Eingänge des internen CTU weitergeleitet.

Nach der Verarbeitung wird das Ergebnis (aktueller Zählerstand CV und Ausgang Q) an die Ausgabeadapter gesendet und gleichzeitig das Bestätigungsereignis CNF ausgelöst. Wichtig: **Der Baustein führt bei jedem beliebigen der drei Ereignisse einen vollständigen Durchlauf durch**, d.h. CU, R und PV werden stets gemeinsam ausgewertet. Dieses Verhalten kann zu unerwarteten Zählimpulsen führen, wenn nicht alle Eingänge gleichzeitig relevant sind. Für reine Änderungsauslösung empfiehlt sich die Verwendung eines AX_D_FF (D-Flipflop) als Filter.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Alle Ein- und Ausgänge sind als Adapter realisiert, was eine flexible Verschaltung in Composite-FBs oder Subapplikationen ermöglicht.
- **Kapselung von IEC 61131-3**: Der Baustein verpackt die bewährte Zählerlogik aus der IEC 61131-3 in eine 4diac-konforme Komponente.
- **Simultane Triggerung**: Jedes Ereignis (CU, R, PV) führt zu einer vollständigen Neuberechnung – auch wenn nur ein Parameter geändert wurde.
- **Lizenz**: veröffentlicht unter der Eclipse Public License 2.0.

Zustandsübersicht

Der interne Zustand wird durch den IEC 61131-3 CTU bestimmt:

- **CV** (aktueller Zählerstand) – Integer-Wert, der bei jedem CU-Ereignis inkrementiert wird (sofern kein Reset erfolgt).
- **Q** (Ausgang) – Boolescher Wert, der TRUE wird, sobald CV >= PV.
- Bei einem **Reset** (R) wird CV auf den Wert von R.D1 gesetzt (meist 0) und Q zurückgesetzt.
- Bei einem neuen **PV** wird lediglich der Schwellwert aktualisiert; Q kann sich sofort ändern, wenn CV >= neuer PV.

Der Baustein hat keine eigenen sequentiellen Zustände jenseits dieser Datenabhängigkeiten.

Anwendungsszenarien

- **Produktionszählung**: Erfassung von Werkstücken auf einem Förderband (CU = Impulsgeber, PV = Chargengröße, Q = Chargenende).
- **Ereigniszähler**: Zählen von Sensorsignalen in Kombination mit zeitlicher Auswertung.
- **Batch-Prozesse**: Steuerung von Dosier- oder Abfüllvorgängen mit einstellbarem Sollwert (PV).
- **Modulare Automatisierung**: Einbettung in größere Funktionsbausteine über einheitliche Adapter-Schnittstellen (AX/AI).

Vergleich mit ähnlichen Bausteinen

Baustein	Eigenschaften
AI_FB_CTU	Adapter-basiert, verwendet IEC 61131-3 CTU, löst bei jedem Eingang aus
Standard CTU (IEC 61131)	Eingänge als Events + Daten, kein Adapterkonzept, oft direkt an Hardware gebunden
CTUD (Auf-/Abwärtszähler)	Bietet zusätzlich Abwärtszählen, hat komplexere Schnittstelle
AX_CTU (hypothetisch)	Könnte optimierte Ereignisauslösung (nur bei Änderung) bieten

Der **AI_FB_CTU** besticht durch seine einfache Adapter-Anbindung, erfordert aber ggf. einen externen Filter, um unnötige Aufrufe zu vermeiden.

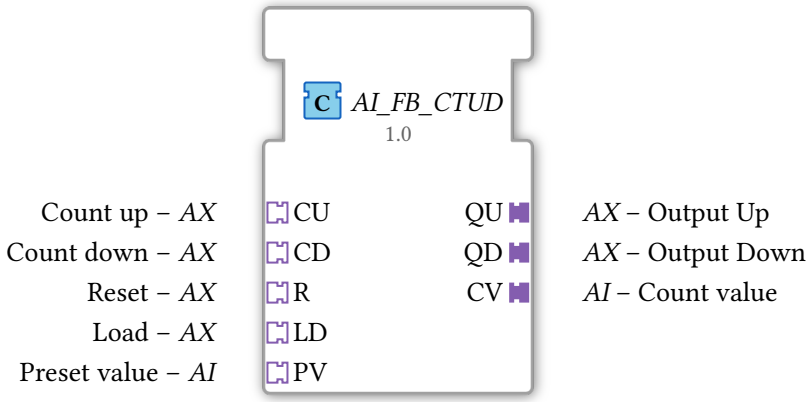
Fazit

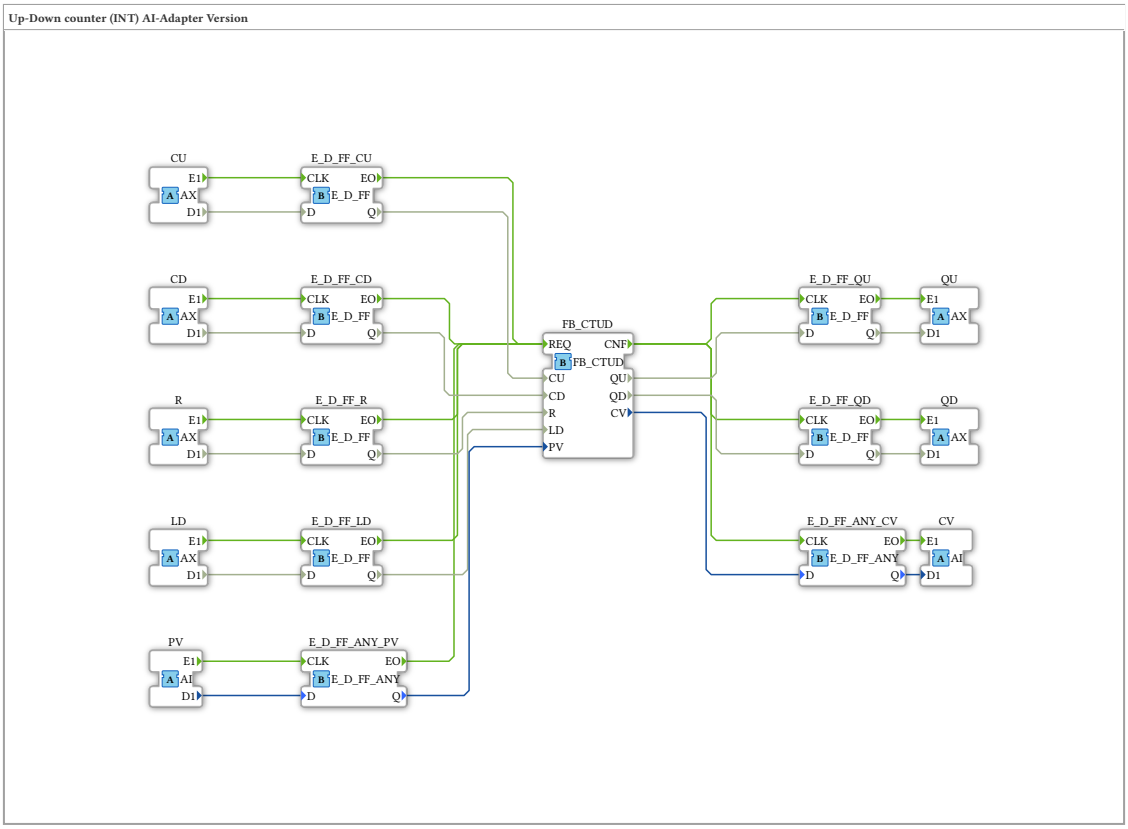
Der **AI_FB_CTU** ist ein praktischer Zählerbaustein für die adapterbasierte Automatisierung mit 4diac. Er kombiniert die bewährte IEC 61131-3 Logik mit moderner, modularer Schnittstellentechnik. Seine einfache Struktur und klare Funktionsweise machen ihn zur ersten Wahl für alle Aufwärtszählaufgaben, bei denen eine lose Kopplung über Adapter gewünscht ist. Anwender sollten jedoch das Triggern auf jedes Ereignis beachten und bei Bedarf eine Differentialfilterung vorsehen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AI_FB_CTUD





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AI_FB_CTUD** realisiert einen Aufwärts-/Abwärtszähler (Up-Down-Counter) vom Datentyp **INT** in einer Adapter-basierten Variante. Er kapselt den IEC 61131-3 Standardbaustein `FB_CTUD` und stellt dessen Funktionalität über die Adapter-Schnittstellen (`AX` und `AI`) der 4diac-IDE zur Verfügung. Der Baustein eignet sich besonders für den Einsatz in industriellen Steuerungssystemen, die auf ereignisgesteuerte Kommunikation setzen.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine separaten Ereignis-Eingänge im klassischen Sinne. Stattdessen werden Ereignisse über die **Sockets** (Adapter) empfangen:

- **CU** (AX) – Zählereignis aufwärts (Count Up)
- **CD** (AX) – Zählereignis abwärts (Count Down)
- **R** (AX) – Rücksetzen (Reset)
- **LD** (AX) – Laden des Presetwertes (Load)
- **PV** (AI) – Übergabe des Presetwertes (Preset Value)

Jedes eingehende Ereignis löst eine Aktualisierung des internen Zählers aus.

Ereignis-Ausgänge

- **CNF** – Bestätigung der Ausführung (Execution Confirmation), wird bei jeder Aktualisierung ausgegeben.
- **QU** (AX) – Ausgangsereignis für den Aufwärtszähler (Up Output)
- **QD** (AX) – Ausgangsereignis für den Abwärtszähler (Down Output)
- **CV** (AI) – Ausgangsereignis für den aktuellen Zählwert (Count Value)

Daten-Eingänge

Alle Daten werden über die **Sockets** bereitgestellt:

- **CU.D** (BOOL) – Zählimpuls aufwärts
- **CD.D** (BOOL) – Zählimpuls abwärts
- **R.D** (BOOL) – Rücksetzsignal
- **LD.D** (BOOL) – Freigabe zum Laden des Presetwerts
- **PV.D** (INT) – Vorgabewert (Preset Value)

Daten-Ausgänge

- **QU.D** (BOOL) – Aufwärtszähl-Flag (TRUE, wenn Zähler den PV-Wert erreicht hat)
- **QD.D** (BOOL) – Abwärtszähl-Flag (TRUE, wenn Zähler den Wert 0 erreicht hat)
- **CV.D** (INT) – Aktueller Zählerstand

Adapter

- **Sockets (Eingänge):**
  - `CU` (Typ `adapter::types::unidirectional::AX`)
  - `CD` (Typ `adapter::types::unidirectional::AX`)
  - `R` (Typ `adapter::types::unidirectional::AX`)
  - `LD` (Typ `adapter::types::unidirectional::AX`)
  - `PV` (Typ `adapter::types::unidirectional::AI`)
- **Plugs (Ausgänge):**
  - `QU` (Typ `adapter::types::unidirectional::AX`)
  - `QD` (Typ `adapter::types::unidirectional::AX`)
  - `CV` (Typ `adapter::types::unidirectional::AI`)

Funktionsweise

Der Baustein arbeitet als reiner **Adapter-Wrapper** für den internen IEC 61131-3 FB `FB_CTUD`. Die eingehenden Adapter-Ereignisse (über die Sockets) werden auf den Ereignis-Eingang `FB_CTUD.REQ` zusammengeführt, während die zugehörigen Datenwerte (z. B. `CU.D`, `PV.D`) an die entsprechenden Eingänge des internen Zählers weitergeleitet werden. Bei jeder Ausführung des internen FBs werden dessen Ausgänge (`CV`, `QU`, `QD`) gelesen und über die Adapter-Plugs ausgegeben. Gleichzeitig wird das Ereignis `CNF` erzeugt.

Der interne Zähler verhält sich wie ein klassischer IEC-61131-3 Up-Down-Counter:

- Ein positiver Flanke an `CU` erhöht den Zählwert um 1.
- Ein positiver Flanke an `CD` verringert den Zählwert um 1.
- Ein positiver Flanke an `R` setzt den Zählwert auf 0 zurück.
- Die Kombination `LD = TRUE` und ein Ereignis an `PV` lädt den aktuellen Zählwert auf den Wert von `PV`.
- `QU` wird TRUE, wenn der Zählwert den Presetwert erreicht oder überschreitet.
- `QD` wird TRUE, wenn der Zählwert den Wert 0 erreicht oder unterschreitet.

Technische Besonderheiten

- **Keine Änderungsdetektion:** Der Baustein gibt **bei jedem empfangenen Ereignis** (unabhängig von einer tatsächlichen Wertänderung) die Ausgänge `CNF`, `QU`, `QD` und `CV` aus. Falls ein Verhalten gewünscht ist, bei dem die Ausgänge nur bei einer Änderung feuern, sollte ein vorgeschalteter **AX_D_FF** (Differentiator/Filter) verwendet werden. Dies wird auch im Quellcode des Bausteins ausdrücklich empfohlen.
- **Adapter-Nutzung:** Sämtliche Kommunikation erfolgt über unidirektionale Adapter (`AX` für boolesche Ereignisse, `AI` für analoge Werte). Dies ermöglicht eine lose Kopplung und Wiederverwendbarkeit in verschiedenen Systemtopologien.
- **Einheitliche Triggerung:** Alle fünf Ereignisquellen (CU, CD, R, LD, PV) führen zur Ausführung des internen Zählers über den gemeinsamen `REQ`-Eingang – es gibt keine separate Abarbeitungslogik pro Ereignis.

Zustandsübersicht

Der Baustein selbst besitzt keinen expliziten Zustandsautomaten, sondern spiegelt den Zustand des internen `FB_CTUD` wider:

- **Zählerstand (INT):** Initial 0, verändert sich durch CU, CD, R oder LD.
- **Flags:** `QU` und `QD` werden basierend auf dem Vergleich von Zählerstand und Presetwert (bzw. 0) gesetzt.
- **Presetwert:** Wird über den `PV`-Adapter gespeichert und bei jedem `LD`-Ereignis geladen.

Eine detaillierte Zustandsmaschine des IEC-61131-3 Counters kann der entsprechenden Norm entnommen werden.

Anwendungsszenarien

- **Ereigniszählung in beide Richtungen:** Z. B. Erfassung von Werkstücken auf einem Förderband, wobei Sensoren für Einlauf (CU) und Auslauf (CD) verwendet werden.
- **Lagerbestandsverwaltung:** Auf- und Abbuchung von Materialien in einem Puffer.
- **Positionierung mit Preset:** Ein System kann auf eine vorgegebene Position (PV) gesetzt und dann mittels CU/CD nachgeführt werden.
- **Resets nach Zyklusende:** Rücksetzen des Zählers nach Erreichen einer bestimmten Stückzahl.

Vergleich mit ähnlichen Bausteinen

- **FB_CTUD (Standard IEC-61131-3):** Der `AI_FB_CTUD` ist eine **Adapter-Version** des gleichen Logikbausteins. Während der Standard-FB klassische Ein-/Ausgangspins verwendet, kommuniziert die Adapter-Version über Ereignis- und Daten-Adapter. Dies vereinfacht die Einbettung in 4diac-Systeme mit adapterbasierter Kommunikation.
- **AI_FB_CTU / AI_FB_CTD:** Einfache Aufwärts- bzw. Abwärtszähler ohne Gegenrichtung. `AI_FB_CTUD` bietet beide Zählrichtungen in einem Baustein.
- **AX_D_FF:** Ein Differentiator/Filter, der nur bei Änderungen feuert. Kann als Vorschaltbaustein eingesetzt werden, um das Ausgabeverhalten von `AI_FB_CTUD` zu optimieren.

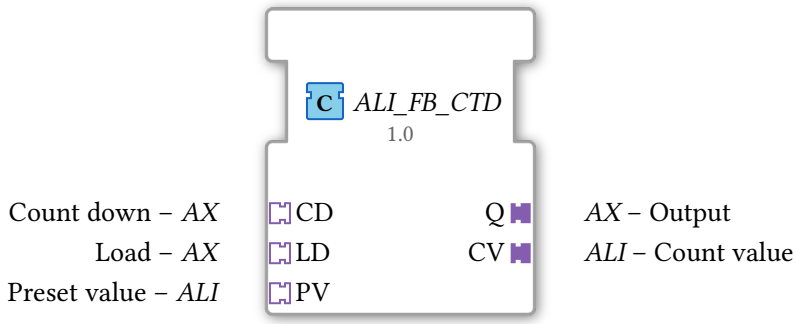
Fazit

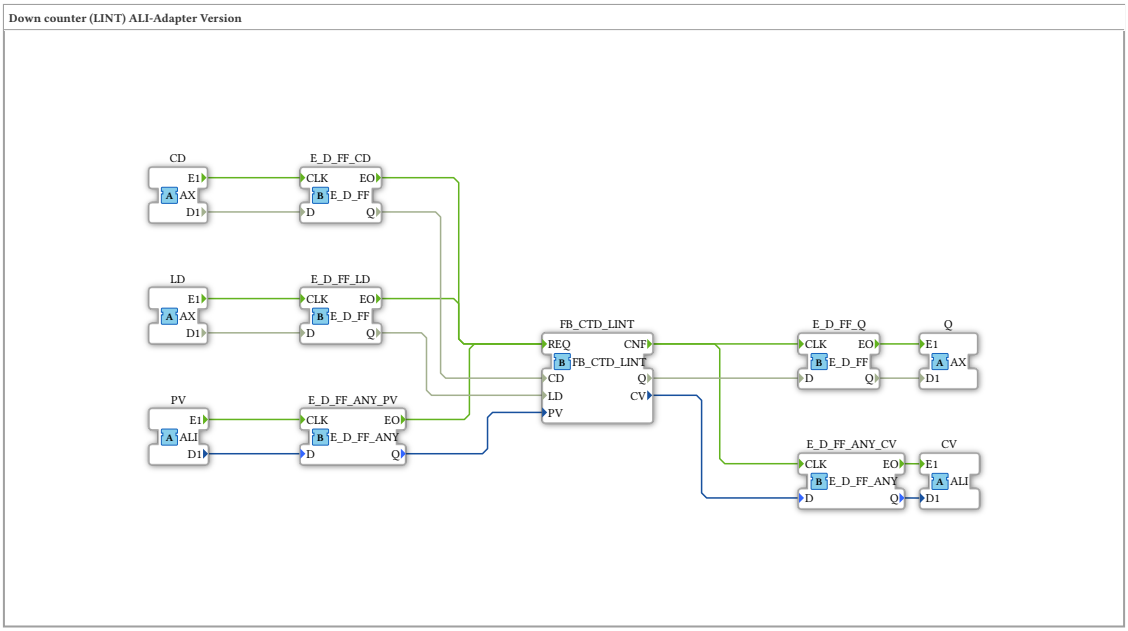
Der `AI_FB_CTUD` ist ein leistungsfähiger, adapterbasierter Aufwärts-/Abwärtszähler, der die etablierte IEC-61131-3 Funktionalität in die ereignisgesteuerte Welt von 4diac überführt. Durch die saubere Kapselung und die Verwendung unidirektionaler Adapter ist er flexibel einsetzbar und gut in modulare Automatisierungslösungen integrierbar. Die explizite Empfehlung zur Nutzung eines `AX_D_FF` für änderungsgesteuerte Ausgaben zeigt, dass der Baustein bewusst einfach gehalten wurde – eine Stärke für transparente und vorhersagbare Systeme.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ALI_FB_CTD





• – ◦ • – ◦ • – ◦ •

Einleitung

Der **ALI_FB_CTD** ist ein Abwärtszähler (Down Counter) für LINT-Werte (64-Bit Integer) mit Adapter-Schnittstellen nach IEC 61499. Er ist speziell für die Verwendung mit den unidirektionalen Adaptern `AX` (Ereignis-/Bool-Träger) und `ALI` (LINT-Wert-Träger) ausgelegt. Der Baustein kapselt den Standard-FB `FB_CTD_LINT` aus der Bibliothek `iec61131::counters` und erweitert ihn um eine adapterbasierte Anbindung, die eine lose Kopplung zwischen den Funktionsbausteinen ermöglicht.

Der Baustein feuert seine Ereignisausgänge bei jeder Aktualisierung – unabhängig davon, ob sich der Zählerwert tatsächlich ändert. Für eine änderungsgetriggerte Weiterverarbeitung wird empfohlen, einen `AX_D_FF` als Filter vorzuschalten.

Schnittstellenstruktur

Ereignis-Eingänge

Keine – alle Ereignis- und Dateninformationen werden über Adapter-Sockets (CD, LD, PV) übertragen.

Ereignis-Ausgänge

- **CNF** (Event): Bestätigung der ausgeführten Operation (wird nach jeder Aktualisierung gesendet).

Daten-Eingänge

Es existieren keine separaten DataInputs. Die Eingabedaten werden über die folgenden Adapter-Sockets bereitgestellt:

- **CD** (Socket, Typ `adapter::types::unidirectional::AX`): Abwärtszählimpuls (aktiviert den Zählvorgang).
- **LD** (Socket, Typ `adapter::types::unidirectional::AX`): Ladeimpuls (aktiviert das Laden des Preset-Werts).
- **PV** (Socket, Typ `adapter::types::unidirectional::ALI`): Preset-Wert (LINT); der Wert wird beim Laden (LD) übernommen.

Daten-Ausgänge

Es existieren keine separaten DataOutputs. Die Ausgabedaten werden über die folgenden Adapter-Plugs bereitgestellt:

- **Q** (Plug, Typ `adapter::types::unidirectional::AX`): Ausgangssignal (Bool) – zeigt an, ob der Zählerstand **Null** erreicht hat.
- **CV** (Plug, Typ `adapter::types::unidirectional::ALI`): Aktueller Zählerwert (LINT).

Adapter

Bezeichnung	Richtung	Typ	Beschreibung
CD	Socket	AX	Abwärtszählimpuls (Ereignis + Bool)
LD	Socket	AX	Ladeimpuls (Ereignis + Bool)
PV	Socket	ALI	Preset-Wert (LINT)
Q	Plug	AX	Ausgangssignal bei Zählerstand = 0
CV	Plug	ALI	Aktueller Zählerwert (LINT)

Funktionsweise

Der ALI_FB_CTD realisiert einen Abwärtszähler mit Laden. Die interne Logik wird durch den FB `FB_CTD_LINT` abgebildet:

1. **Laden (Load)**
   Ein Ereignis am Socket **LD** (über `LD.E1`) überträgt den Datenwert von **PV** (über `PV.D1`) in den internen Zähler. Der Zählerstand wird auf den Preset-Wert gesetzt.
2. **Abwärtszählen (Count Down)**
   Ein Ereignis am Socket **CD** (über `CD.E1`) dekrementiert den aktuellen Zählerstand um 1, sofern er größer als 0 ist.
3. **Ausgangssignalisierung**
   - Der aktuelle Zählerwert wird stets über den Plug **CV** ausgegeben (als LINT).
   - Der Plug **Q** liefert `true`, wenn der Zählerstand **Null** erreicht hat (Bool-Wert über den AX-Adapter).
   - Das Ereignis **CNF** wird nach **jeder** Verarbeitung (sowohl nach CD, LD als auch PV) gesendet.
4. **Ereignisweitergabe**
   Die eingehenden Ereignisse von CD, LD und PV werden alle auf den **REQ**-Eingang des internen FB vereinigt. Dessen **CNF**-Ausgang wird dann auf die Ausgangsadapter (Q.E1, CV.E1) und auf den externen Ereignisausgang CNF verteilt.

Technische Besonderheiten

- **Immer-Auslösen**: Der Baustein sendet die Ausgangsereignisse (Q.E1, CV.E1, CNF) bei **jeder** Aktualisierung – auch wenn sich der Zählerwert nicht ändert (z.B. bei wiederholtem CD auf 0). Für eine änderungsbasierte Auslösung muss ein `AX_D_FF` (Flanken-Detektor) zwischengeschaltet werden.
- **Adapter-Kopplung**: Die Verwendung von unidirektionalen Adaptern (AX, ALI) ermöglicht eine flexible Verschaltung in IEC-61499-Netzwerken, ohne starre Daten- und Ereignisverbindungen.
- **LINT-Werte**: Der Zähler arbeitet mit 64-Bit-Integer, geeignet für große Zählbereiche.
- **Interne Struktur**: Der Baustein delegiert die gesamte Zähllogik an den erprobten Standard-FB `FB_CTD_LINT` der Bibliothek `iec61131::counters`.

Zustandsübersicht

Der ALI_FB_CTD besitzt **keinen expliziten Zustandsautomaten** im FB-Netzwerk. Sein Verhalten ist rein ereignisgesteuert und kombinatorisch:

Eingangsereignis	Aktion
LD aktiviert	Zählerstand = Preset-Wert
CD aktiviert	Zählerstand dekrementieren (falls > 0)
PV aktiviert	Keine Zähleränderung, aber Ereignis CNF wird gesendet. Der Datenwert von PV wird nicht unmittelbar übernommen – ein LD ist erforderlich.

Der aktuelle Zählerstand und der Bool-Ausgang werden nach jedem Schritt aktualisiert.

Anwendungsszenarien

- **Stückzähler in der Fertigung**: Am Ende eines Produktionsschritts wird ein Impuls auf CD gegeben; der Zähler zählt von einer vorgegebenen Stückzahl (PV) abwärts. Bei Null wird Q aktiv und signalisiert z.B. „Batch fertig“.

- **Zeitsteuerung mit Pulszählung**: Ein Taktgeber liefert Impulse auf CD; nach einer vordefinierten Anzahl (PV) wird ein Ereignis ausgelöst.
- **Wartungsintervall**: Der Zähler dient als Count-Down in Wartungszyklen; nach Erreichen von 0 wird ein Alarm ausgelöst.

Vergleich mit ähnlichen Bausteinen

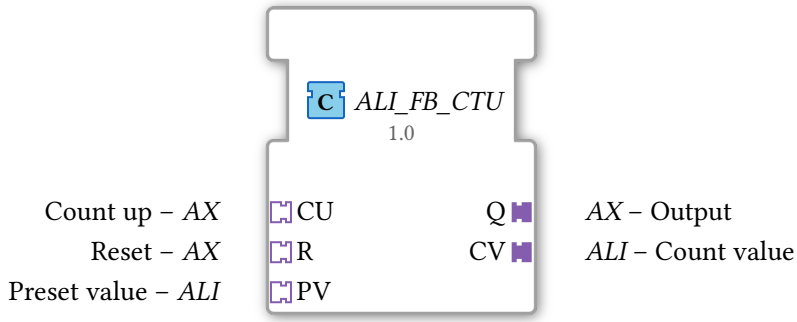
Baustein	Besonderheit
ALI_FB_CTD (dieser Baustein)	Verwendet AX- und ALI-Adapter; immer-auslösend; geeignet für adapterbasierte Architekturen.
FB_CTD_LINT (Standard)	Reiner Daten-/Ereignis-FB ohne Adapter; Ereignisauslösung direkt über die Schnittstellen.
CTU (Aufwärtszähler)	Zählt aufwärts, anstatt abwärts; andere Anwendungslogik.
Zählbausteine mit Flankenunterdrückung	Sie besitzen eigene Filterlogik (z.B. AX_D_FF) und lösen nur bei tatsächlichen Wertänderungen aus.

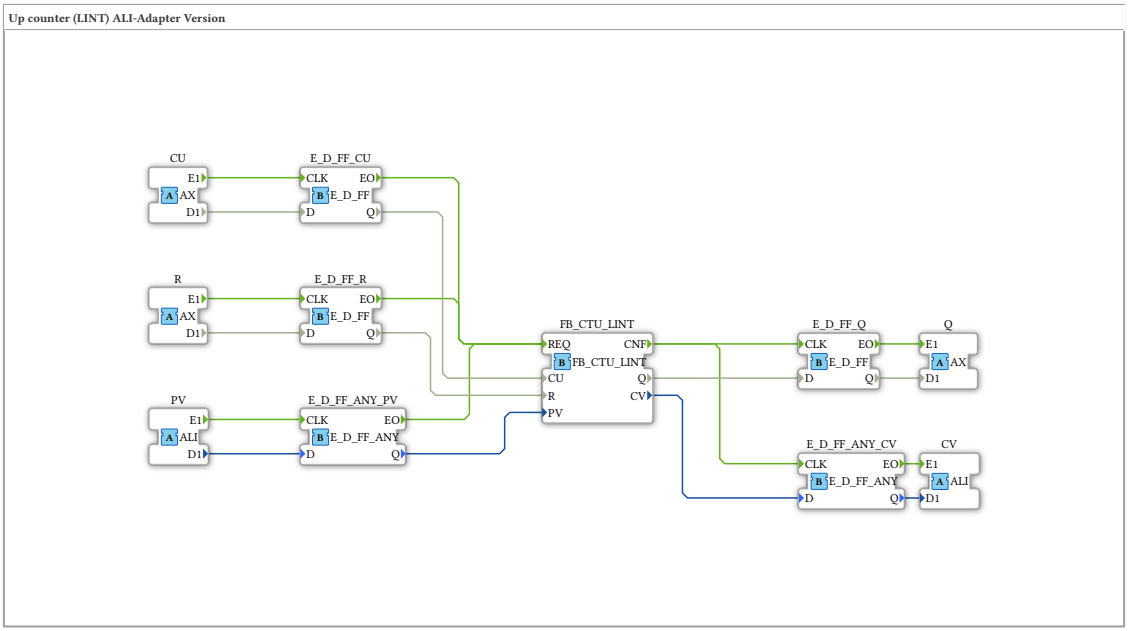
Der ALI_FB_CTD eignet sich besonders in Umgebungen, die auf Adaptern basieren und bei denen ein einfaches, zuverlässiges Abwärtszählen ohne zusätzliche Filter gewünscht ist.

Fazit

Der **ALI_FB_CTD** ist ein robuster Abwärtszähler für LINT-Werte, der speziell für die Adapter-Technik in IEC-61499-Systemen entwickelt wurde. Er bietet eine klare, auf Adaptern basierende Schnittstelle und ist dank der internen Nutzung von `FB_CTD_LINT` zuverlässig und getestet. Durch seine „Immer-Auslösen"-Eigenschaft eignet er sich für Anwendungen, bei denen jede Zustandsänderung am Eingang sofort weitergeleitet werden soll. Wo eine änderungsbasierte Auslösung erforderlich ist, kann ein einfacher Filterbaustein ergänzt werden. Insgesamt stellt er eine wertvolle Ergänzung für adapterorientierte Automatisierungslösungen dar.

ALI_FB_CTU





• – ○ • – ○ • – ○ •

Einleitung

Der **ALI_FB_CTU** ist ein Aufwärtszähler (Counter Up) mit 64-Bit-Ganzzahlauflösung (LINT). Er dient als Adapter-Wrapper für den IEC-61131-Zählerbaustein *FB_CTU_LINT* und stellt alle Ein- und Ausgangssignale über standardisierte Adapter-Schnittstellen (AX und ALI) bereit. Dadurch wird eine klare Trennung von Ereignis- und Datenflüssen erreicht und die Wiederverwendung in unterschiedlichen Projektumgebungen erleichtert.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine separaten, diskreten Ereignis-Eingänge. Die auslösenden Ereignisse werden über die Adapter **CU**, **R** und **PV** empfangen. Jeder dieser Adapter stellt ein Ereignis (E1) sowie einen zugehörigen Datenwert (D1) bereit. Ein eingehendes Ereignis an einem dieser Adapter startet die Verarbeitung.

Ereignis-Ausgänge

- **CNF** (Execution Confirmation): Wird nach jedem vollständigen Verarbeitungsdurchlauf ausgegeben – unabhängig davon, ob CU, R oder PV das Ereignis ausgelöst hat.
- **Q** (Adapter AX): Stellt das Ausgangsereignis (E1) zur Verfügung, das parallel zum CNF-Ereignis ausgegeben wird.
- **CV** (Adapter ALI): Gibt ebenfalls ein Ereignis (E1) aus, synchron zum CNF-Ereignis.

Daten-Eingänge

Der Baustein besitzt keine separaten, diskreten Daten-Eingänge. Alle Eingangsdaten werden über die Adapter-Schnittstellen bereitgestellt:

- **CU.D1** – Count-Up-Signal (BOOL)
- **R.D1** – Reset-Signal (BOOL)
- **PV.D1** – Preset-Wert (LINT)

Daten-Ausgänge

Der Baustein besitzt keine separaten, diskreten Daten-Ausgänge. Die Ausgangsdaten werden über die Adapter-Schnittstellen ausgegeben:

- **Q.D1** – Zählerausgang (BOOL): wird TRUE, sobald der aktuelle Zählerstand ≥ PV ist.
- **CV.D1** – Aktueller Zählerstand (LINT)

Adapter

Richtung	Name	Typ	Beschreibung
Sockets (Eingänge)	CU	AX	Count-Up: Ereignis + BOOL-Signal
	R	AX	Reset: Ereignis + BOOL-Signal
	PV	ALI	Preset-Wert: Ereignis + LINT-Wert
Plugs (Ausgänge)	Q	AX	Ausgangssignal: Ereignis + BOOL-Wert
	CV	ALI	Zählerstand: Ereignis + LINT-Wert

Funktionsweise

Der **ALI_FB_CTU** kapselt einen internen *FB_CTU_LINT*. Jedes eingehende Ereignis an einem der drei Adapter **CU**, **R** oder **PV** wird an den REQ-Event des internen Bausteins weitergeleitet. Gleichzeitig werden die jeweiligen Datenwerte an die entsprechenden Eingänge (*CU*, *R*, *PV*) übergeben. Der interne Zähler arbeitet gemäß der IEC-61131-Definition eines CTU:

- **Beim Ereignis an CU**: Der Zählerstand wird um 1 erhöht, sofern das zugehörige Datenbit TRUE ist.
- **Beim Ereignis an R**: Der Zählerstand wird auf 0 zurückgesetzt.
- **Beim Ereignis an PV**: Der übergebene Wert wird als neuer Preset-Wert gespeichert.

Nach jeder Verarbeitung gibt der interne Baustein einen CNF-Event aus. Dieser wird an alle drei Ausgangsadapter (CNF, Q, CV) weitergegeben, sodass alle Ausgänge synchron aktualisiert werden. Der Ausgang **Q** wird genau dann TRUE, wenn der aktuelle Zählerstand größer oder gleich dem Preset-Wert ist. Der aktuelle Zählerstand ist über **CV** jederzeit ablesbar.

Technische Besonderheiten

- **Adapter-basierte Schnittstelle**: Alle Signale werden über standardisierte Adapter (AX für binäre, ALI für numerische Werte) ausgetauscht. Dies vereinfacht die Anbindung an heterogene Systeme und fördert die Wiederverwendbarkeit.
- **64-Bit-Auflösung**: Der Zähler verwendet den Datentyp LINT (Signed 64-Bit Integer), wodurch sehr große Zählbereiche abgedeckt werden.
- **Gemeinsamer Trigger**: Sowohl CU, R als auch PV lösen denselben internen REQ aus. Werden mehrere Adapter gleichzeitig aktiv, werden alle zugehörigen Datenwerte in einem Schritt verarbeitet – der interne Baustein wertet die Signale parallel aus.
- **Hinweis zum Ausgangsverhalten**: Der Baustein feuert das AX-Ausgangsereignis (Q.E1) bei *jedem* Update – also auch dann, wenn sich der Ausgangszustand nicht ändert. Falls eine reine „On-Change“-Auslösung gewünscht ist, muss dem Ausgang ein AX_D_FF (Einschaltverzögerung/Ereignisfilter) nachgeschaltet werden.

Zustandsübersicht

Der interne Zustand wird durch den Zählerstand (64-Bit-Integer) und den booleschen Ausgang Q gebildet. Eine explizite Zustandsmaschine existiert nicht; der Baustein arbeitet ereignisgesteuert:

Zustandskomponente	Mögliche Werte	Beschreibung
Zählerstand (CV)	0 … 2^{63}-1	Aktueller Zählwert
Ausgang Q	FALSE / TRUE	TRUE, wenn CV ≥ PV

Anwendungsszenarien

- **Ereigniszählung**: Zählen von Impulsen in Fertigungsanlagen, Förderbändern oder Energiezählern.
- **Mengenüberwachung**: Erfassen produzierter Stückzahlen und Auslösen eines Signals bei Erreichen eines Schwellwerts.
- **Reset-gesteuerte Zähler**: Rücksetzen des Zählers auf Null nach einem manuellen oder automatischen Reset.
- **Preset-basierte Steuerung**: Dynamisches Ändern des Sollwerts während des Betriebs (z. B. Chargenwechsel).

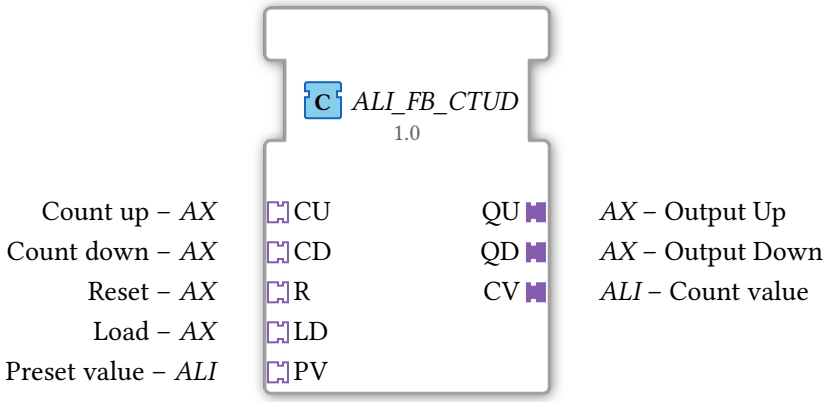
Vergleich mit ähnlichen Bausteinen

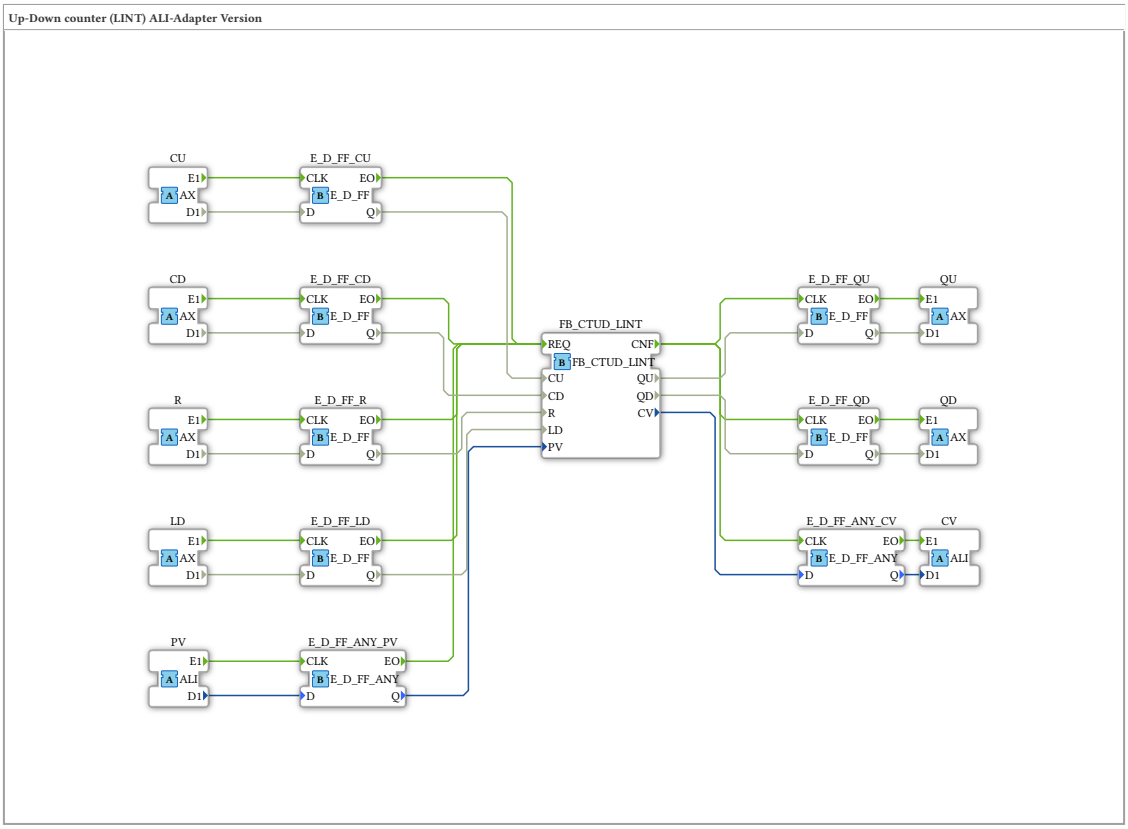
- **Gegenüber einem direkten FB_CTU_LINT**: Der ALI_FB_CTU abstrahiert die Schnittstellen auf Adapter-Ebene. Dadurch können die Signale unabhängig von der konkreten Implementierung mit anderen Bausteinen verbunden werden, die ebenfalls AX/ALI-Adapter verwenden.
- **Gegenüber einem Standard-CTU mit BOOL-Eingängen**: Der ALI_FB_CTU erlaubt den Anschluss von Adaptern, die sowohl Ereignisse als auch Daten bündeln. Dies reduziert die Anzahl separater Leitungen und macht den Entwurf übersichtlicher.
- **Gegenüber einem CTU mit INT/DINT**: Durch die Verwendung von LINT wird ein wesentlich größerer Wertebereich abgedeckt, was für hochauflösende Zählaufgaben oder Langzeitzähler vorteilhaft ist.

Fazit

Der **ALI_FB_CTU** ist ein leistungsfähiger, adapterbasierter Aufwärtszähler für industrielle Automatisierungsumgebungen. Er kombiniert die bewährte IEC-61131-Zählerlogik mit modernen Adapter-Schnittstellen und bietet eine einfache, lesbare Anbindung an andere Komponenten. Die 64-Bit-Auflösung, die gemeinsame Ereignisverarbeitung aller Eingänge und die Möglichkeit, Schwellwerte dynamisch zu setzen, machen ihn zu einer flexiblen Lösung für vielfältige Zähl- und Überwachungsaufgaben.

ALI_FB_CTUD





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ALI_FB_CTUD** realisiert einen Auf-/Abwärtszähler (Up-Down Counter) mit einem Wertebereich des Typs **LINT** (64-Bit-Ganzzahl). Er ist speziell für die Verwendung mit **ALI-Adaptern** ausgelegt und kapselt einen standardkonformen Zähler gemäß IEC 61131-3 (FB_CTUD_LINT). Die Steuerung und Ausgabe erfolgt ausschließlich über Adapterschnittstellen, was eine flexible Einbindung in adapterbasierte Architekturen erlaubt.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine direkten Ereignis-Eingänge. Alle Steuerereignisse werden über die **Sockets** (Adapter) empfangen:

- **CU.E1** – Ereignis zum Erhöhen des Zählerstands (Count Up)
- **CD.E1** – Ereignis zum Verringern des Zählerstands (Count Down)
- **R.E1** – Ereignis zum Zurücksetzen des Zählerstands (Reset)
- **LD.E1** – Ereignis zum Laden des Preset-Werts (Load)
- **PV.E1**– Ereignis zum Übernehmen des Preset-Werts (Preset Value)

Hinweis: Alle fünf Ereignisse lösen einen gemeinsamen internen Verarbeitungszyklus aus.

Ereignis-Ausgänge

- **CNF** (Event) – Bestätigung der Ausführung (Execution Confirmation)

Zusätzlich werden über die **Plugs** (Ausgangsadapter) Ereignisse ausgegeben:

- **QU.E1** – Ereignis bei steigendem Zählerstand (Output Up)
- **QD.E1** – Ereignis bei fallendem Zählerstand (Output Down)
- **CV.E1** – Ereignis bei jeder Änderung des Zählerwerts (Count Value)

Besonderheit: Diese Ereignisse werden bei **jeder** Aktualisierung (unabhängig vom Eingangsereignis) ausgelöst. Für eine änderungsgesteuerte Auslösung empfiehlt sich die Vorschaltung eines AX_D_FF-Bausteins.

Daten-Eingänge

Die Datenwerte werden über die **Sockets** bereitgestellt:

- **CU.D1** (BOOL) – Freigabe zum Hochzählen (Count Up Enable)
- **CD.D1** (BOOL) – Freigabe zum Runterzählen (Count Down Enable)
- **R.D1** (BOOL) – Rücksetzsignal (Reset)
- **LD.D1** (BOOL) – Ladesignal (Load)
- **PV.D1** (LINT) – Preset-Wert (Preset Value)

Daten-Ausgänge

Die Ergebnisdaten werden über die **Plugs** bereitgestellt:

- **QU.D1** (BOOL) – Signal: Zählerstand > 0 (z. B. für „Up“-Ausgang)
- **QD.D1** (BOOL) – Signal: Zählerstand < 0 (für „Down“-Ausgang, abhängig von interner Logik)
- **CV.D1** (LINT) – aktueller Zählerwert

Adapter

Typ	Richtung	Adapter-Typ	Beschreibung
CU	Socket	`adapter::types::unidirectional::AX`	Aufwärtszähl-Eingang (Ereignis+BOOL)
CD	Socket	`adapter::types::unidirectional::AX`	Abwärtszähl-Eingang (Ereignis+BOOL)
R	Socket	`adapter::types::unidirectional::AX`	Reset-Eingang (Ereignis+BOOL)
LD	Socket	`adapter::types::unidirectional::AX`	Load-Eingang (Ereignis+BOOL)
PV	Socket	`adapter::types::unidirectional::ALI`	Preset-Wert-Eingang (Ereignis+LINT)
QU	Plug	`adapter::types::unidirectional::AX`	Aufwärts-Ausgang (Ereignis+BOOL)
QD	Plug	`adapter::types::unidirectional::AX`	Abwärts-Ausgang (Ereignis+BOOL)
CV	Plug	`adapter::types::unidirectional::ALI`	Zählerwert-Ausgang (Ereignis+LINT)

Funktionsweise

Der Baustein enthält einen internen Funktionsblock **FB_CTUD_LINT**, der die eigentliche Zählerlogik nach IEC 61131-3 implementiert. Jedes Ereignis an einem der fünf Sockets (CU.E1, CD.E1, R.E1, LD.E1, PV.E1) löst eine Verarbeitung aus: Die zugehörigen booleschen Daten (CU.D1, CD.D1, R.D1, LD.D1) und der Preset-Wert (PV.D1) werden an den internen Baustein weitergeleitet und dort synchron ausgewertet. Der interne Baustein berechnet daraufhin den neuen Zählerstand und die Ausgangssignale QU, QD und CV. Nach Abschluss der Berechnung wird das Ausgangsereignis **CNF** gesendet und gleichzeitig werden die Ereignisse **QU.E1**, **QD.E1** und **CV.E1** auf den entsprechenden Plugs ausgegeben.

Die Verwendung von Adaptern ermöglicht eine lose Kopplung: Die eigentlichen Signalquellen (z. B. Sensoren oder Bedienelemente) und -senken (z. B. Aktoren oder Anzeigen) werden über Adapterverbindungen angebunden, ohne dass die internen Daten- und Ereignisleitungen direkt sichtbar sind.

Technische Besonderheiten

- **Datenbereich**: 64-Bit vorzeichenbehaftete Ganzzahl (LINT), geeignet für große Zählerstände.
- **Adapter-getriebene Schnittstelle**: Keine direkten Ereignis- oder Dateneingänge; alle Steuerungen erfolgen über AX- und ALI-Adapter.
- **Immerwährende Ereignisausgabe**: Die Ausgangsereignisse (QU.E1, QD.E1, CV.E1) werden bei **jedem** Verarbeitungszyklus ausgegeben, unabhängig davon, ob sich der Zählerstand tatsächlich geändert hat. Dieses Verhalten ist im Quellcode explizit dokumentiert – für eine änderungsbasierte Auslösung wird die Verwendung eines AX_D_FF-Filters empfohlen.
- **Interner Standard-Baustein**: Der FB_CTUD_LINT entspricht der IEC 61131-3-Definition eines Auf-/Abwärtszählers.

Zustandsübersicht

Eine formale Zustandsmaschine ist nicht extern sichtbar. Der interne Zustand besteht aus dem aktuellen **Zählerwert** (CV) und den internen Flags **QU** (z. B. „Wert > 0“) und **QD** (z. B. „Wert < 0“). Der Zustand wird bei jedem Eingangsereignis gemäß folgender Priorität aktualisiert:

1. **Reset (R)**: setzt CV auf 0.
2. **Load (LD)**: übernimmt PV als neuen CV.
3. **Count Up (CU)**: erhöht CV um 1, wenn CU.D1 = TRUE.
4. **Count Down (CD)**: verringert CV um 1, wenn CD.D1 = TRUE. Die Flags QU und QD werden danach aus dem neuen CV berechnet.

Anwendungsszenarien

- **Produktionszählung**: Erfassen von Werkstücken oder Zyklen – Aufwärtszählen bei Eintritt, Abwärtszählen bei Austritt.
- **Positionsüberwachung**: Hoch-/Runterzählen von Schritten in einem Lineartrieb.
- **Bestandsverwaltung**: Zählen von Lagereinheiten mit manueller Korrektur über Load/Reset.
- **Adapter-basierte Steuerungssysteme**: Einbindung in Architekturen, die auf unidirektionale Adapter (AX/ALI) setzen, z. B. verteilte Automatisierungsknoten.

Vergleich mit ähnlichen Bausteinen

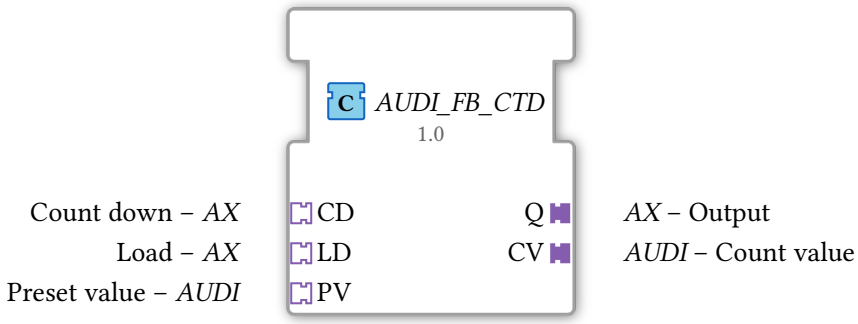
Der Standard-IEC-61131-3-Baustein **CTUD** (z. B. `FB_CTUD_INT`) arbeitet meist mit kleineren Datentypen (INT, DINT) und bietet direkte Ereignis- und Dateneingänge. Der **ALI_FB_CTUD** erweitert dieses Konzept um:

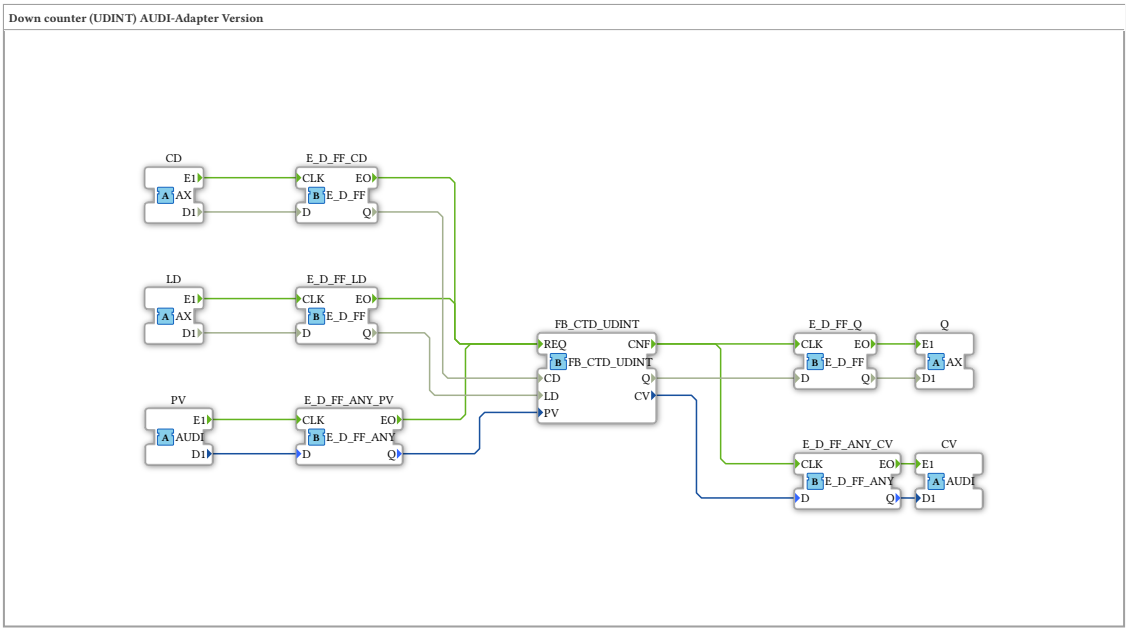
- **LINT-Datentyp** für sehr große Zählerbereiche.
- **Adapter-Schnittstellen** (AX/ALI) statt freier Ein-/Ausgänge.
- **Immerwährende Ereignisausgabe** im Gegensatz zu einer rein änderungsgesteuerten Ausgabe.

Fazit

Der **ALI_FB_CTUD** ist ein leistungsfähiger Auf-/Abwärtszähler für 64-Bit-Zahlen, der sich nahtlos in adapterbasierte Automatisierungsumgebungen einfügt. Seine Struktur ermöglicht eine klare Trennung von Steuer- und Datenflüssen und erleichtert die Wiederverwendung in modularen Steuerungssystemen. Die Besonderheit der immerwährenden Ereignisausgabe sollte bei der Projektierung beachtet werden – gegebenenfalls ist ein nachgeschalteter Filterbaustein erforderlich, um unerwünschte Flanken zu vermeiden.

AUDI_FB_CTD





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUDI_FB_CTD** realisiert einen Abwärtszähler (Down Counter) auf Basis des Datentyps UDINT. Er ist speziell für den Einsatz mit Adapter-Schnittstellen ausgelegt und verwendet die Typen `AX` und `AUDI` zur ereignisgesteuerten Kommunikation. Die eigentliche Zähllogik wird durch einen internen Funktionsblock des Typs `iec61131::counters::FB_CTD_UDINT` ausgeführt.

Dieser Baustein feuert bei jeder Aktualisierung – unabhängig davon, ob der Eingang `CD`, `LD` oder `PV` ausgelöst wurde – ein Ereignis auf dem Ausgangskanal `Q`. Für eine auf Änderungen basierte Auslösung wird der Einsatz eines nachgeschalteten `AX_D_FF`-Filters empfohlen.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine direkten Ereignis-Eingänge. Alle Ereignisse werden über die angeschlossenen **Sockets** (Adapter-Eingänge) zugeführt.

Socket (Adapter)	Typ	Beschreibung
`CD`	`AX`	Count-Down-Ereignis (Abwärtszählen)
`LD`	`AX`	Load-Ereignis (Laden des Preset-Wertes)
`PV`	`AUDI`	Preset-Wert-Ereignis (Vorgabewert setzen)

Ereignis-Ausgänge

Name	Typ	Beschreibung
`CNF`	Event	Bestätigungsereignis nach jeder Verarbeitung

Daten-Eingänge

Die Daten werden über die Sockets als Teil des Adapter-Protokolls übertragen.

Socket (Adapter)	Daten-Typ	Bedeutung
`CD.D1`	BOOL	Befehl zum Herunterzählen (TRUE = zählen)
`LD.D1`	BOOL	Befehl zum Laden des Preset-Wertes
`PV.D1`	UDINT	Vorgabewert für den Zähler (Preset)

Daten-Ausgänge

Die Ausgangsdaten werden über die Plugs ausgegeben.

Plug (Adapter)	Daten-Typ	Bedeutung
`Q.D1`	BOOL	Zählerstand = 0 (TRUE wenn erreicht)
`CV.D1`	UDINT	Aktueller Zählerwert

Adapter

Richtung	Name	Typ	Beschreibung
Socket	`CD`	`AX`	Ereignisgesteuerter Zählimpuls (abwärts)
Socket	`LD`	`AX`	Ereignisgesteuertes Laden des Presets
Socket	`PV`	`AUDI`	Ereignisgesteuerte Vorgabe des Presets
Plug	`Q`	`AX`	Ausgangssignal: Zählerstand = 0
Plug	`CV`	`AUDI`	Ausgangssignal: aktueller Zählerwert

Funktionsweise

Intern wird ein standardisierter IEC-61131-Abwärtszähler (`FB_CTD_UDINT`) verwendet. Die Adapter-Ereignisse `CD.E1`, `LD.E1` und `PV.E1` werden zu einem einzigen `REQ`-Ereignis des internen Bausteins zusammengeführt. Dadurch wird der interne Zähler bei jedem beliebigen äußeren Ereignis neu berechnet.

- **CD (Count Down)**: Liegt das Datensignal `CD.D1 = TRUE` an, wird der aktuelle Zählerwert um 1 verringert. Bei `FALSE` bleibt der Wert unverändert.
- **LD (Load)**: Ist `LD.D1 = TRUE`, wird der Zähler auf den Wert von `PV.D1` gesetzt.
- **PV (Preset Value)**: Das Ereignis aktualisiert den gespeicherten Preset-Wert; ein darauf folgendes `LD`-Ereignis übernimmt diesen.

Nach jeder Berechnung werden die Ausgangsdaten `CV` (aktueller Zählerwert) und `Q` (TRUE wenn `CV = 0`) über die entsprechenden Plugs ausgegeben. Gleichzeitig wird das Bestätigungsereignis `CNF` ausgelöst.

Technische Besonderheiten

- **Ereignisverhalten**: Bei jedem Eingangsereignis (`CD`, `LD` oder `PV`) wird der Ausgangskanal `Q` bedient. Das bedeutet, dass ein `Q.E1`-Ereignis auch dann gesendet wird, wenn sich der Zählerwert oder der `Q`-Zustand nicht geändert hat. Dies kann zu unnötig vielen Ereignissen in der nachfolgenden Logik führen.
- **Empfehlung zur Filterung**: Soll das Ausgangsereignis nur bei tatsächlicher Zustandsänderung von `Q` ausgelöst werden, ist ein **AX_D_FF** (Flip-Flop) als Filter zwischenzuschalten. Dieser unterdrückt Wiederholungen.
- **Adapterbasierte Anbindung**: Alle Ein- und Ausgänge erfolgen über genormte Adapter (`AX`, `AUDI`). Dies vereinfacht die modulare Verdrahtung und Wiederverwendung in unterschiedlichen Umgebungen.

Zustandsübersicht

Der Baustein besitzt keine expliziten Zustände. Der interne Zähler arbeitet nach den Regeln eines IEC-61131-CTD-Bausteins. Der aktuelle Zählerstand wird durch die Datensignale bestimmt.

Anwendungsszenarien

- **Ereignisgesteuerte Abwärtszählung**: Einsatz in Automatisierungssystemen, bei denen ein Zählerstand durch äußere Ereignisse dekrementiert wird (z. B. Anzahl verarbeiteter Werkstücke).
- **Modulare Steuerungskomponenten**: Integration in adapterbasierte Architekturen, bei denen standardisierte Schnittstellen den Datenaustausch zwischen verschiedenen Funktionsblöcken erleichtern.
- **Timer-Nachbildung**: Kombination mit einem Taktgeber, um eine definierte Anzahl von Zyklen zu zählen und bei Erreichen von Null eine Aktion auszulösen.

Vergleich mit ähnlichen Bausteinen

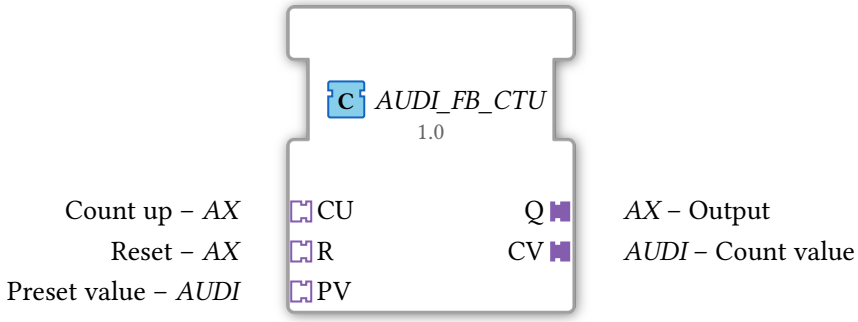
- **Standard-CTD** (z. B. `FB_CTD_UDINT`): Besitzt direkte Ein-/Ausgangsvariablen, keine Adapter. Der `AUDI_FB_CTD` hingegen kapselt die Logik und stellt die Signale über Adapter bereit, was eine saubere Trennung von Ereignis- und Datenkanälen ermöglicht.

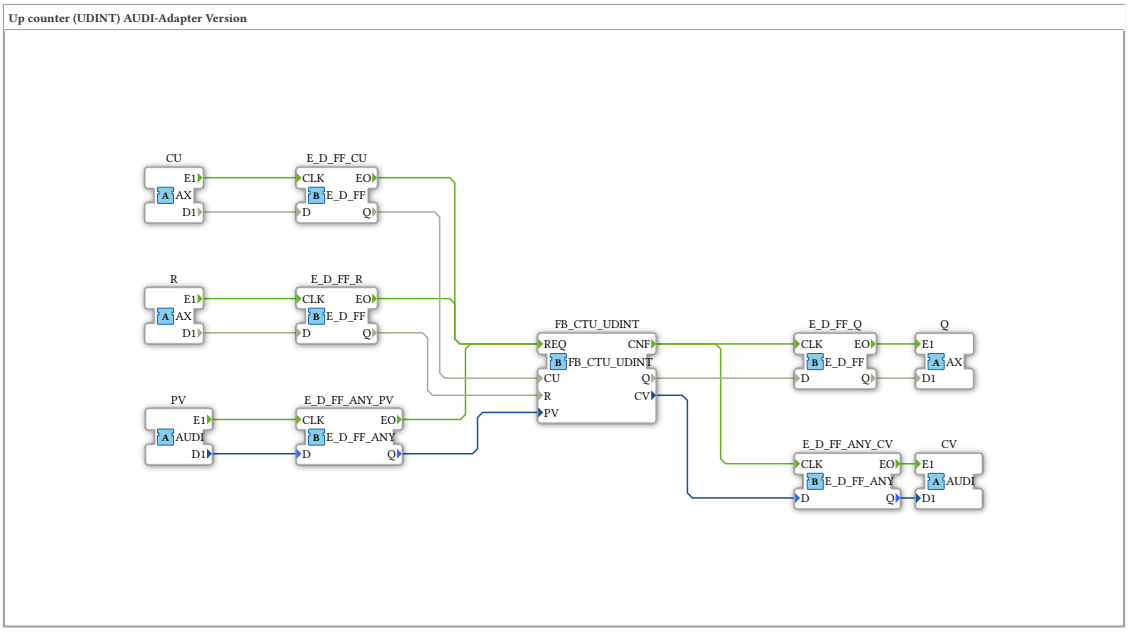
- **Aufwärtszähler (CTU)**: Im Gegensatz zum `AUDI_FB_CTD` zählt ein CTU aufwärts. Beide können adapterbasiert realisiert werden, sind aber für unterschiedliche Zählrichtungen ausgelegt.
- **Eigenschaft „immer feuern“**: Viele Standardzähler geben nur dann ein Ausgangsereignis, wenn sich der Ausgangszustand ändert. Der `AUDI_FB_CTD` tut dies bei jeder Verarbeitung, was eine Besonderheit darstellt und je nach Anwendung Vor- oder Nachteil sein kann.

Fazit

Der `AUDI_FB_CTD` ist ein flexibler adaptergestützter Abwärtszähler für UDINT-Werte. Seine Stärke liegt in der modularen Anbindung über standardisierte Schnittstellen. Entwickler sollten jedoch das spezielle Ereignisverhalten (Ausgangsimplus bei jedem Update) beachten und bei Bedarf durch einen Filter korrigieren. Der Baustein eignet sich gut für ereignisgesteuerte Zählaufgaben in heterogenen Steuerungslandschaften.

AUDI_FB_CTU





• – ○ • – ○ • – ○ •

Einleitung

Der **AUDI_FB_CTU** ist ein Aufwärtszähler (Up Counter) für den Datentyp **UDINT** (32-Bit vorzeichenlos), der über standardisierte **AUDI-Adapter** angebunden wird. Der Baustein kapselt die IEC 61131-Zählfunktion in einer adapterbasierten Architektur und erlaubt das Triggern von Zählimpulsen, Reset- und Preset-Operationen über Adapter-Events. Jeder zählrelevante Vorgang (Zählimpuls, Reset, Setzen des Presetwerts) löst zwingend einen Ausgangs-Event aus – unabhängig davon, ob sich der Zählerstand oder der Ausgangswert ändert. Soll eine reine Änderungsauslösung (on-change) realisiert werden, muss ein nachgeschalteter AX_D_FF-Baustein verwendet werden.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine klassischen Event-Eingänge (EventInputs). Stattdessen werden die auslösenden Ereignisse **über die Adapter-Sockets** hereingeführt:

- **CU.E1** – Ereignis vom *Count up*-Adapter: ein ansteigender Impuls erhöht den Zählerstand um 1.
- **R.E1** – Ereignis vom *Reset*-Adapter: setzt den Zählerstand auf 0 zurück.
- **PV.E1** – Ereignis vom *Preset value*-Adapter: übernimmt den aktuell am Preset-Eingang anliegenden Wert als neuen Vorwahlwert (wirkt sofort, ohne den Zähler zu ändern).

Alle drei Ereignisse werden intern auf den **REQ**-Event des eingebetteten IEC-Funktionsblocks `FB_CTU_UDINT` gemultiplext. Das bedeutet, dass jeder dieser externen Impulse eine Verarbeitung anstößt.

Ereignis-Ausgänge

- **CNF** – Bestätigungs-Event (EventOutput): wird nach jeder Verarbeitung eines eingehenden Ereignisses ausgegeben. Ist mit den Ausgangsadaptern mitverbunden, sodass die angeschlossenen Komponenten über eine erfolgte Zähloperation informiert werden.

Zusätzlich werden folgende **Adapter-Events** ausgelöst (über interne Verdrahtung):

- **Q.E1** – Event des *Output*-Adapters (Typ AX): signalisiert die Verfügbarkeit eines neuen Ausgangswerts Q.
- **CV.E1** – Event des *Count value*-Adapters (Typ AUDI): signalisiert die Verfügbarkeit eines neuen Zählerstands CV.

Diese Adapter-Events feuern **gleichzeitig mit CNF**, d. h. bei jedem zählrelevanten Ereignis werden alle drei Ausgangs-Events gleichzeitig aktiviert.

Daten-Eingänge

Auch die Daten werden **über die Adapter-Sockets** bereitgestellt:

- **CU.D1** (UDINT, über Adapter CU) – Zählimpuls: Nur der Wert 0 oder 1 wird ausgewertet; ein positiver Flankenwechsel (0⟶1) führt zu einer Erhöhung. (Der Adapter liefert das Ereignis selbst, zusätzlich kann der Datenwert z. B. als flankenbewertetes Signal genutzt werden.)
- **R.D1** (UDINT, über Adapter R) – Reset-Signal: Ein Wert ≠0 setzt den Zähler zurück.
- **PV.D1** (UDINT, über Adapter PV) – Preset-Wert: Der hier anliegende Wert wird beim Eintreffen des PV.E1-Ereignisses als neue Vorwahl übernommen. (Der Datenwert wird also nur bei einem expliziten PV-Ereignis eingelesen.)

Daten-Ausgänge

- **Q.D1** (BOOL, über Adapter Q) – Ausgangssignal: `TRUE`, wenn der aktuelle Zählerstand ≥ **PV** ist (Pre-set Output).
- **CV.D1** (UDINT, über Adapter CV) – Aktueller Zählerstand (Count Value).

Adapter

Adapter-Typ	Richtung	Name	Kurzbeschreibung
`AX` (unidirectional)	Socket (Eingang)	CU	Liefert das Zählimpuls-Ereignis und einen flankenbewerteten Datenwert.
`AX` (unidirectional)	Socket (Eingang)	R	Liefert das Reset-Ereignis und den Resetschalter.
`AUDI` (unidirectional)	Socket (Eingang)	PV	Liefert das Preset-Ereignis und den Preset-Zahlenwert.
`AX` (unidirectional)	Plug (Ausgang)	Q	Stellt das Ausgangsbinary (Q) als Ereignis und Datenwert bereit.
`AUDI` (unidirectional)	Plug (Ausgang)	CV	Stellt den aktuellen Zählerstand als Ereignis und Datenwert bereit.

Funktionsweise

Der **AUDI_FB_CTU** delegiert seine gesamte Zähllogik an den inneren IEC-Baustein `FB_CTU_UDINT`. Bei jedem eingehenden Ereignis (CU.E1, R.E1 oder PV.E1) wird dessen **REQ**-Eingang aktiviert. Der innere Zähler führt dann die folgende Logik aus:

1. **Reset zuerst**: Falls **R** ≠ 0, wird der Zähler auf 0 gesetzt.
2. **Zählimpuls**: Falls **CU** ≠ 0 und ein positiver Flankenübergang seit dem letzten Aufruf vorliegt, wird CV um 1 erhöht.
3. **Preset-Übernahme**: Falls **PV.E1** ausgelöst wurde, wird der an PV anliegende Wert als neuer Schwellwert (PV) gespeichert.
4. **Vergleich**: Ist CV ≥ PV (gespeicherter Presetwert), wird **Q** = TRUE, sonst FALSE.

Nach der Verarbeitung werden die Ereignisse **CNF**, **Q.E1** und **CV.E1** gleichzeitig ausgegeben, unabhängig davon, ob sich Q oder CV geändert haben.

Technische Besonderheiten

- **Adapterbasierung**: Sämtliche Ein- und Ausgänge sind als standardisierte unidirektionale Adapter (`AX` für Binär-, `AUDI` für Zahlenwerte) realisiert. Dies ermöglicht eine flexible Verschaltung in adapterorientierten Architekturen.
- **Multiplexing der Ereignisse**: Die drei Eingangsereignisse (CU, R, PV) werden auf einen gemeinsamen REQ gemultiplext. Dies vereinfacht die interne Verdrahtung, bedeutet aber, dass auch ein Reset- oder Presetvorgang den Ausgangs-Event auslöst.
- **Keine on-change-Filterung**: Wie im Header-Kommentar beschrieben, feuert der Baustein **immer** bei jeder Verarbeitung. Für eine reine Änderungsauslösung muss ein AX_D_FF nachgeschaltet werden (z. B. auf Q und CV).
- **Datenbreite**: Alle Zähler- und Vorwahlwerte sind als **UDINT** (32-Bit vorzeichenlos) ausgeführt.
- **Zustandshaltung**: Der innere Zähler speichert den aktuellen Zählerstand und die Preset-Schwelle über mehrere Aufrufe hinweg.

Zustandsübersicht

Der Baustein verfügt über **keinen eigenen Zustandsautomaten** (ECC). Die Zustandslogik liegt vollständig im eingebetteten IEC-Funktionsblock `FB_CTU_UDINT`. Dessen wesentliche Zustände sind:

- **Idle**: Warten auf ein Ereignis (CU.E1, R.E1 oder PV.E1).
- **Processing** (während des internen Durchlaufs): Auswerten der Eingangsdaten (CU, R, PV), Aktualisieren des Zählers und der Ausgänge Q und CV.
- **Emitting**: Ausgabe der Ereignisse CNF, Q.E1, CV.E1.

Ein expliziter Fehlerzustand ist nicht vorgesehen.

Anwendungsszenarien

- **Ereigniszählung in adapterbasierten Automatisierungssystemen** – z. B. Zählen von Werkstücken auf einem Förderband, wobei der Sensor über einen AX-Adapter angebunden ist.
- **Zeit- oder Impulszähler mit Reset** – Der Reset-Adapter kann z. B. von einem manuellen Taster kommen.
- **Erreichen einer Schwellwertmarke** – Der PV-Eingang erlaubt eine dynamische Grenzwertvorgabe; sobald der Zählerstand diesen Wert erreicht oder überschreitet, wird Q aktiv.
- **Integration in Modulsysteme** – Durch die standardisierten Adapter (AUDI, AX) lässt sich der Baustein leicht in bestehende adapterbasierte Bibliotheken einfügen.

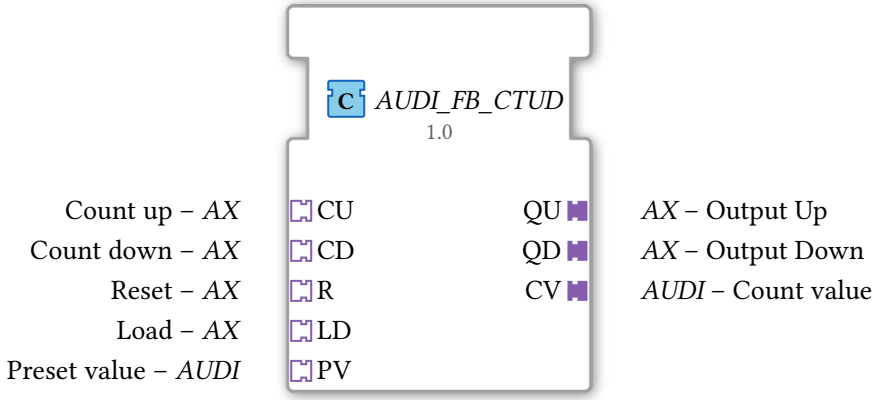
Vergleich mit ähnlichen Bausteinen

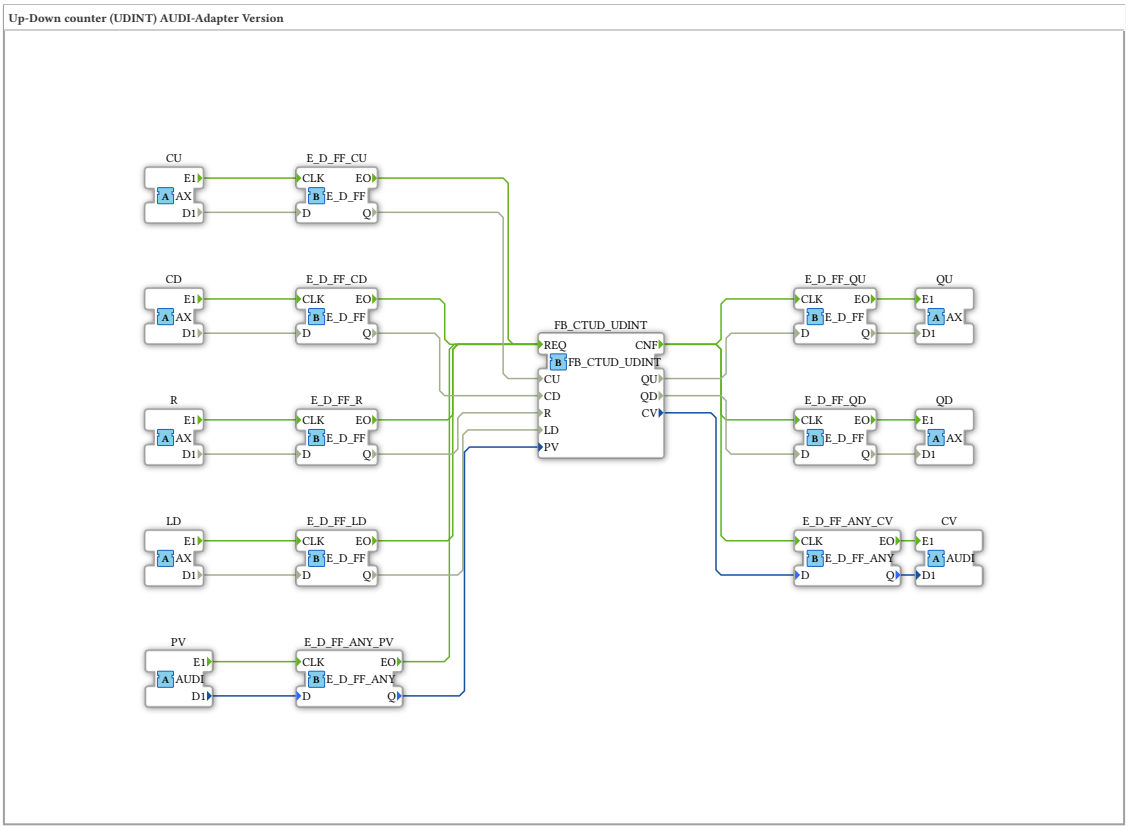
- **Gegensatz zu klassischen IEC-CTU-Bausteinen** (z. B. `CTU_UDINT` ohne Adapter): Diese haben direkte Event- und Daten-Eingänge, benötigen aber keine Adapter-Wrapper. Der AUDI_FB_CTU bietet den Vorteil einer adaptereinheitlichen Schnittstelle, erfordert aber ein spezifisches Adapter-Mapping.
- **Gegensatz zu CTU-Bausteinen mit On-Change-Auslösung**: Bausteine, die nur bei tatsächlichen Wertänderungen feuern, vermeiden unnötige Ereignisfluten. Der AUDI_FB_CTU feuert immer, was durch den Zusatz eines AX_D_FF kompensiert werden kann.
- **Vergleich zum AUDI_FB_CTD** (Abwärtszähler): Der interne IEC-Baustein wäre dann `FB_CTD_UDINT`; Schnittstellen und Verhalten sind analog, nur Zählrichtung und Bedingung für Q unterscheiden sich (Q aktiv bei CV ≤ 0).

Fazit

Der **AUDI_FB_CTU** stellt einen robusten, adapterbasierten Aufwärtszähler dar, der sich nahtlos in adapterorientierte Automatisierungslösungen integrieren lässt. Durch die Kapselung eines standardisierten IEC-Zählers und die Verwendung von AUDI/AX-Adaptern wird eine hohe Interoperabilität erreicht. Die Eigenschaft, bei jedem Ereignis zu feuern, ist für viele Anwendungen akzeptabel oder kann durch eine nachgeschaltete Filterung korrigiert werden. Der Baustein eignet sich besonders für modulare, erweiterbare Steuerungsarchitekturen, die auf Adapterprotokollen basieren.

AUDI_FB_CTUD





• – ○ • – ○ • – ○ •

Einleitung

Der **AUDI_FB_CTUD** ist ein Aufwärts-/Abwärtszähler (Up-Down Counter) mit einem Wertebereich für vorzeichenlose 32-Bit-Ganzzahlen (UDINT). Er ist als reine Adapter-Version realisiert, d. h. alle Ein- und Ausgänge werden über Adapter-Schnittstellen bereitgestellt. Der Baustein kapselt den standardisierten IEC-61131-Funktionsblock `FB_CTUD_UDINT` und erweitert dessen Funktionalität um eine adapterbasierte Anbindung. Bei jedem auslösenden Ereignis (CU, CD, R, LD oder PV) werden alle Ausgangsadapter

zyklisch aktualisiert. Wenn eine änderungsgesteuerte (flankensensitive) Auslösung benötigt wird, ist die Verwendung eines AX_D_FF als Filterbaustein empfohlen.

Schnittstellenstruktur

Ereignis-Eingänge
Der Baustein besitzt auf oberster Ebene keine diskreten Ereignis-Eingänge. Alle Ereignisse werden über die eingehenden Adapter-Schnittstellen (Sockets) vom Typ AX übertragen.

Ereignis-Ausgänge

Name	Typ	Kommentar
CNF	Event	Ausführungsbestätigung – wird bei jeder Bearbeitung eines eingehenden Ereignisses ausgegeben.

Zusätzlich werden Ereignisse über die ausgehenden Adapter (Plugs) gesendet.

Daten-Eingänge
Auf oberster Ebene existieren keine direkten Daten-Eingänge. Alle Datenwerte werden über die Sockets CU.D1, CD.D1, R.D1, LD.D1 und PV.D1 bereitgestellt.

Daten-Ausgänge
Auf oberster Ebene existieren keine direkten Daten-Ausgänge. Die Ergebnisdaten werden über die Plugs QU.D1, QD.D1 und CV.D1 ausgegeben.

Adapter

Richtung	Name	Typ	Kommentar
Socket (Eingang)	CU	adapter::types::unidirectional::AX	Count up – Ereignis und Daten (BOOL) zum Hochzählen
Socket (Eingang)	CD	adapter::types::unidirectional::AX	Count down – Ereignis und Daten (BOOL) zum Herunterzählen
Socket (Eingang)	R	adapter::types::unidirectional::AX	Reset – Ereignis und Daten (BOOL) zum Rücksetzen des Zählers
Socket (Eingang)	LD	adapter::types::unidirectional::AX	Load – Ereignis und Daten (BOOL) zum Laden des Preset-Wertes
Socket (Eingang)	PV	adapter::types::unidirectional::AUDI	Preset value – Ereignis und Daten (UDINT) für den Vorgabewert
Plug (Ausgang)	QU	adapter::types::unidirectional::AX	Output Up – Ereignis und Daten (BOOL), aktiv wenn der aktuelle Zählwert den Wert 0 erreicht
Plug (Ausgang)	QD	adapter::types::unidirectional::AX	Output Down – Ereignis und Daten (BOOL), aktiv wenn der aktuelle Zählwert den Wert >= PV erreicht
Plug (Ausgang)	CV	adapter::types::unidirectional::AUDC	Count value – Ereignis und Daten (UDINT) mit dem aktuellen Zählwert

Funktionsweise
Der AUDI_FB_CTUD nutzt intern den Baustein FB_CTUD_UDINT aus der IEC-61131-Bibliothek. Alle über die Sockets eingehenden Ereignisse (von CU, CD, R, LD und PV) werden auf den REQ-Eingang des internen Bausteins geführt. Die zugehörigen Daten werden auf die entsprechenden Eingänge (CU, CD, R, LD, PV) geleitet. Der interne Baustein verarbeitet die Anforderung und aktualisiert seine Ausgänge. Nach erfolg-

reicher Bearbeitung wird das Ereignis CNF ausgelöst und gleichzeitig werden die Ausgabeadapter QU, QD und CV mit ihren neuen Daten und Ereignissen versorgt.

Wichtig: Der Baustein feuert die Ausgangsereignisse (QU.E1, QD.E1, CV.E1) bei **jeder** Verarbeitung eines eingehenden Ereignisses. Eine filternde Flankenauswertung findet nicht statt.

Technische Besonderheiten

- **Reine Adapter-Schnittstelle:** Der Baustein besitzt keine klassischen Ein-/Ausgänge, sondern ausschließlich Adapter (Sockets und Plugs). Dies ermöglicht eine flexible Kopplung in adapterbasierten Automatisierungsarchitekturen.
- **Verwendung unidirektionaler Adapter:** Die Adapter AX und AUDI sind als unidirektional definiert. Damit wird eine klare Richtung der Daten- und Ereignisflüsse erzwungen.
- **Kapselung des IEC-61131-Standards:** Die Zähllogik entspricht dem genormten CTUD-Funktionsblock, was die Übertragbarkeit und Wartbarkeit erhöht.
- **Bereich:** Der Zählwert ist als UDINT (vorzeichenlos, $0 \ldots 2^{32}-1$) implementiert.

Zustandsübersicht

Der interne FB_CTUD_UDINT durchläuft folgende Zustände, abhängig von den eingehenden Befehlen:

Bedingung	Aktion
CU = TRUE (steigende Flanke) und CD = FALSE	Zähler erhöht sich um 1
CD = TRUE (steigende Flanke) und CU = FALSE	Zähler verringert sich um 1
R = TRUE (steigende Flanke)	Zähler wird auf 0 zurückgesetzt
LD = TRUE (steigende Flanke)	Zähler wird auf den Wert von PV gesetzt
Gleichzeitige CU- und CD-Flanken	Zählerstand bleibt unverändert (Priorität: keine)
QU = TRUE	Aktueller Wert = 0
QD = TRUE	Aktueller Wert ≥ PV

Anwendungsszenarien

- **Stückzähler in Produktionsanlagen:** Erfassung von Produkten an einer Förderstrecke, wobei CU und CD für korrekte Zählung bei Vor- und Rücklauf genutzt werden.
- **Lagerbestandsverwaltung:** Zählen von Ein- und Auslagerungen, mit Reset bei Inventur und Load für Sollvorgaben.
- **Ereigniszähler in der Gebäudeautomation:** Erfassen von Personenströmen (z. B. in Aufzügen) mit zusätzlicher Rücksetzfunktion.
- **Adapter-basierte Systeme:** Integration in verteilte Steuerungslandschaften, die auf dem Adapter-Konzept der 4diac-IDE aufbauen (z. B. AUTOSAR-ähnliche Kommunikation).

Vergleich mit ähnlichen Bausteinen

Merkmal	AUDI_FB_CTUD	Standard CTUD (IEC 61131)
Schnittstelle	Reine Adapter (Sockets/Plugs)	Direkte Ereignis- und Datenein-/ausgänge
Datenformat	UDINT (über AUDI-Adapter)	UDINT (direkt)
Ereignisverhalten	Jedes eingehende Event feuert alle Ausgänge	Ereignis wird nur bei relevanten Änderungen ausgegeben
Flexibilität	Höher durch adapterbasierte Verkabelung	Geringer, aber einfacher in klassischen SPS-Umgebungen
Empfohlener Filter	Zusätzlicher AX_D_FF für flankengesteuerte Ausgabe	Kein Filter nötig, da flankenempfindlich

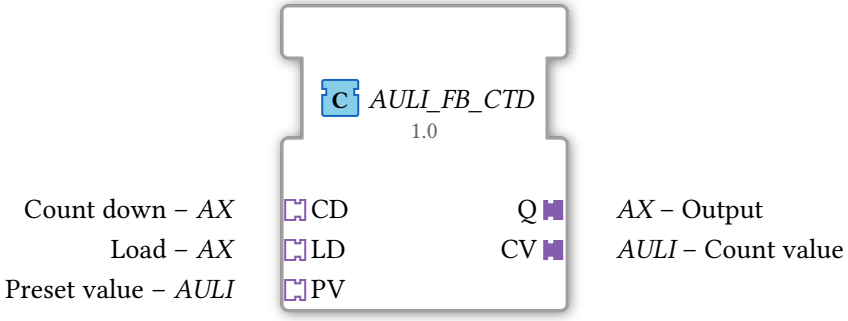
Fazit

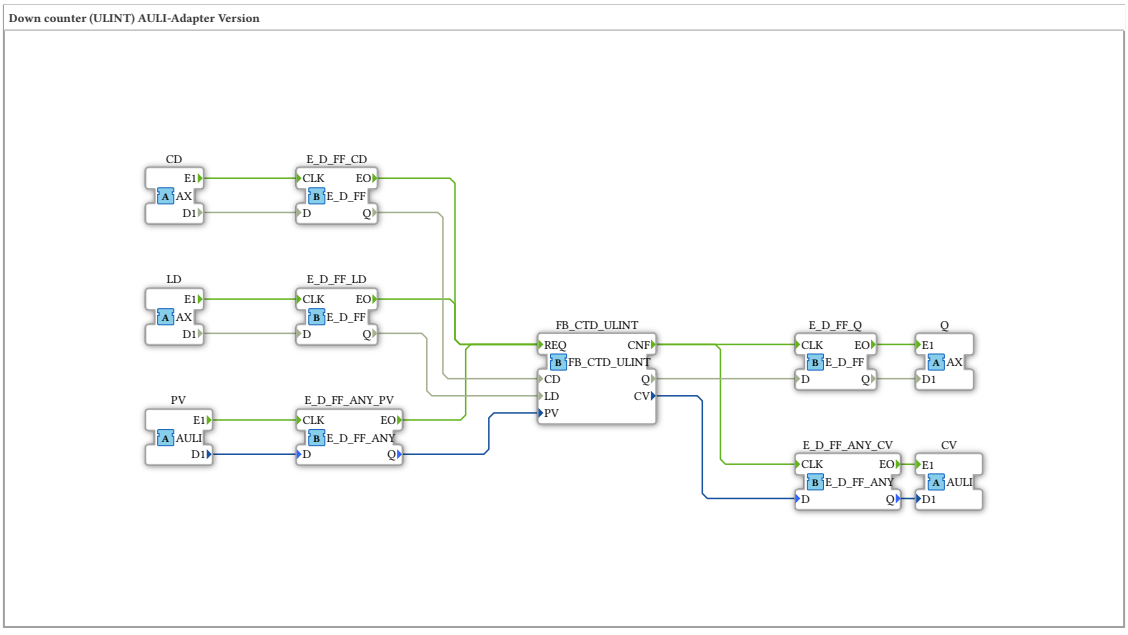
Der `AUDI_FB_CTUD` ist ein leistungsfähiger Aufwärts-/Abwärtszähler, der durch seine adapterbasierte Schnittstelle besonders für moderne, modulare Automatisierungslösungen geeignet ist. Er kombiniert die bewährte IEC-61131-Zähllogik mit der Flexibilität der 4diac-Adaptertechnologie. Anwender sollten jedoch beachten, dass die Ausgangsereignisse bei jeder Verarbeitung und nicht nur bei Änderungen ausgelöst werden – gegebenenfalls ist ein nachgeschalteter Flankenfilter erforderlich. Mit seiner klaren Struktur und den standardisierten Datentypen stellt er eine robuste Komponente für Zählaufgaben in industriellen Steuerungssystemen dar.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AULI_FB_CTD





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AULI_FB_CTD** realisiert einen **Abwärtszähler (Down Counter)** auf Basis des Datentyps `ULINT` (unsigned long integer). Er ist als **Adapter-Version** ausgeführt und kapselt den Standardbaustein `FB_CTD_ULINT` aus der IEC 61131-3-Bibliothek. Der Baustein ermöglicht eine modulare Anbindung über die Adapter-Schnittstellen CD (Count Down), LD (Load) und PV (Preset Value) sowie die Ausgabe des aktuellen Zählwerts (CV) und eines binären Signals (Q) über entsprechende Plug-Adapter.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt **keine direkten Ereignis-Eingänge**. Die Triggerung erfolgt ausschließlich über die **Event-Kanäle der Socket-Adapter**:

Adapter	Event-Port	Beschreibung
CD	CD.E1	Zählimpuls (Count Down)
LD	LD.E1	Ladeimpuls (Load des Preset-Werts)
PV	PV.E1	Aktualisierung des Preset-Werts

Hinweis: Jedes dieser Ereignisse löst eine Verarbeitung des internen Zählers aus. Der Zähler wird dabei immer neu berechnet – unabhängig davon, ob sich der Wert tatsächlich ändert.

Ereignis-Ausgänge

Name	Beschreibung
CNF	Bestätigung (Confirmation) nach jeder Verarbeitung

Zusätzlich werden über die Plug-Adapter **zwei Ereignis-Ausgänge** bereitgestellt:

Adapter	Event-Port	Beschreibung
Q	Q.E1	Wird bei jeder Verarbeitung ausgegeben
CV	CV.E1	Wird bei jeder Verarbeitung ausgegeben

Anmerkung: Da die Ereignisse bei jedem Update (CD, LD, PV) feuern, wird empfohlen, bei Bedarf eine Änderungsflanke mit einem `AX_D_FF` zu filtern (siehe Technische Besonderheiten).

Daten-Eingänge

Daten werden ebenfalls über die **Socket-Adapter** bereitgestellt:

Adapter	Daten-Port	Datentyp	Beschreibung
CD	CD.D1	`BOOL`	Zählimpuls (steigende Flanke)
LD	LD.D1	`BOOL`	Ladebefehl (steigende Flanke)
PV	PV.D1	`ULINT`	Preset-Wert (bei LD oder PV-Update geladen)

Daten-Ausgänge

Daten werden über die **Plug-Adapter** ausgegeben:

Adapter	Daten-Port	Datentyp	Beschreibung
Q	Q.D1	`BOOL`	Zählerstand = 0 (Ausgangssignal)
CV	CV.D1	`ULINT`	Aktueller Zählerstand

Adapter

Richtung	Adapter-Typ	Kurzbeschreibung
Socket	`AX` (bidirektional)	Count-Down-Steuerung (Event + BOOL)
Socket	`AX` (bidirektional)	Load-Steuerung (Event + BOOL)
Socket	`AULI` (bidirektional)	Preset-Wert (Event + ULINT)

Plug	AX (bidirektional)	Ausgang Q (Event + BOOL)
Plug	AULI (bidirektional)	Ausgang Zählerstand (Event + ULINT)

Funktionsweise

Der Baustein kapselt die IEC 61131-3-Funktion FB_CTD_ULINT. Die interne Logik wird durch die Ereignisse der drei Socket-Adapter getriggert:

1. **CD-Ereignis (Count Down):** Bei einer steigenden Flanke von CD.D1 und einem gleichzeitigen Event auf CD.E1 wird der Zählerstand um 1 dekrementiert (sofern er > 0 ist). Der neue Wert wird am Plug CV ausgegeben.
2. **LD-Ereignis (Load):** Bei einer steigenden Flanke von LD.D1 und einem Event auf LD.E1 wird der aktuelle Preset-Wert (PV.D1) in den Zähler geladen. Der Zählerstand wird auf den Preset-Wert gesetzt.
3. **PV-Ereignis (Preset Value Update):** Ein Event auf PV.E1 aktualisiert den intern gespeicherten Preset-Wert (ohne den Zählerstand zu verändern). Dies ist nützlich, um den Preset dynamisch während des Betriebs zu ändern.

Nach jeder Verarbeitung wird das Bestätigungsereignis CNF sowie die Ereignisse auf den Plug-Adaptern Q.E1 und CV.E1 gesendet. Die Daten Q.D1 und CV.D1 werden entsprechend aktualisiert.

Technische Besonderheiten

- **Datentyp ULINT:** Der Baustein arbeitet mit vorzeichenlosen 64-Bit-Ganzzahlen (ULINT), wodurch Zählbereiche von 0 bis 2^{64}-1 möglich sind – geeignet für sehr große Zählaufgaben.
- **Adapter-basierte Anbindung:** Alle Ein- und Ausgänge erfolgen über Adapter (AX für binäre Signale, AULI für ULINT-Werte). Dies ermöglicht eine saubere Kapselung und modulare Verdrahtung in der 4diac-IDE.
- **Ereignisausgabe bei jedem Update:** Der Baustein feuert die Ausgangsereignisse bei jedem eingehenden Ereignis (CD, LD, PV) – auch wenn sich der Zählerstand oder Ausgangswert nicht ändert. Dadurch entsteht ein **permanentes Triggern** des nachgeschalteten Netzwerks. → **Empfehlung:** Verwenden Sie einen AX_D_FF (Differentiator/Filter) an den Ausgängen, wenn nur bei Wertänderung reagiert werden soll.
- **Kein interner Zustandsautomat:** Der Baustein selbst besitzt keinen eigenen Zustandsautomaten; die Zustandslogik (z. B. Erkennung der steigenden Flanke) wird vom internen FB_CTD_ULINT übernommen.

Zustandsübersicht

Der Baustein verwaltet intern lediglich den **Zählerstand** (CV) und den **aktuellen Preset-Wert** (PV). Eine explizite Zustandsmaschine liegt nicht vor. Die möglichen Aktionen sind:

Zustand / Aktion	Auslöser	Ergebnis
Count Down	CD-Ereignis & CD.D1=TRUE	CV := CV - 1 (falls CV>0)
Load	LD-Ereignis & LD.D1=TRUE	CV := PV (aktueller Preset)
Preset Update	PV-Ereignis	PV wird überschrieben (CV bleibt unverändert)

Der Ausgang Q wird auf TRUE gesetzt, sobald CV = 0 ist; andernfalls ist er FALSE.

Anwendungsszenarien

- **Stückzähler mit großem Bereich:** Erfassung von Produktionsmengen mit Wertebereich > 32 Bit (z. B. 10 Mrd. Teile).
- **Vorwahl- oder Ablaufsteuerung:** Einsatz als Abwärtszähler in einer Ablaufkette, bei der nach Erreichen des Werts 0 ein Signal ausgelöst wird (z. B. Chargenende).
- **Dynamische Preset-Werte:** Änderung des Zähl-Endwerts während des Betriebs über das PV-Ereignis, ohne den aktuellen Zählerstand zu beeinflussen.
- **Modulare Systeme:** Integration in größere Steuerungsarchitekturen, die durchgängig auf Adapter-Kommunikation setzen (z. B. über den Eclipse 4diac-Adapter-Mechanismus).

Vergleich mit ähnlichen Bausteinen

Baustein	Datentyp	Schnittstelle	Besonderheit
`FB_CTD_ULINT`	ULINT	Standard-EA	Basis-Abwärtszähler ohne Adapter
AULI_FB_CTD	ULINT	Adapter (AX, AULI)	Adapter-gekapselt, alle Events führen zu Update
`FB_CTD` (Standard)	INT/UINT	Standard-EA	Meist 16-Bit oder 32-Bit, feste Ereignislogik

Der **AULI_FB_CTD** bietet durch die Adapter-Kopplung eine flexiblere Einbindung in komplexe Netzwerke, hat aber den „NebenEffekt“, dass Ausgangsereignisse auch bei unveränderten Werten gesendet werden. Für Anwendungen, die nur bei Wertänderung feuern sollen, ist der Basisbaustein `FB_CTD_ULINT` oder eine Kombination mit einem Flankendetektor (`AX_D_FF`) zu bevorzugen.

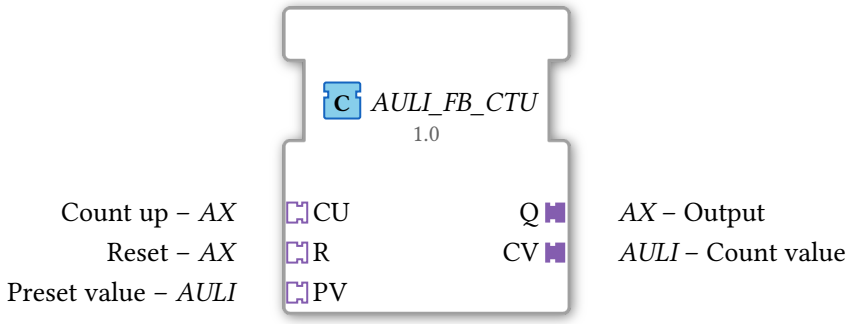
Fazit

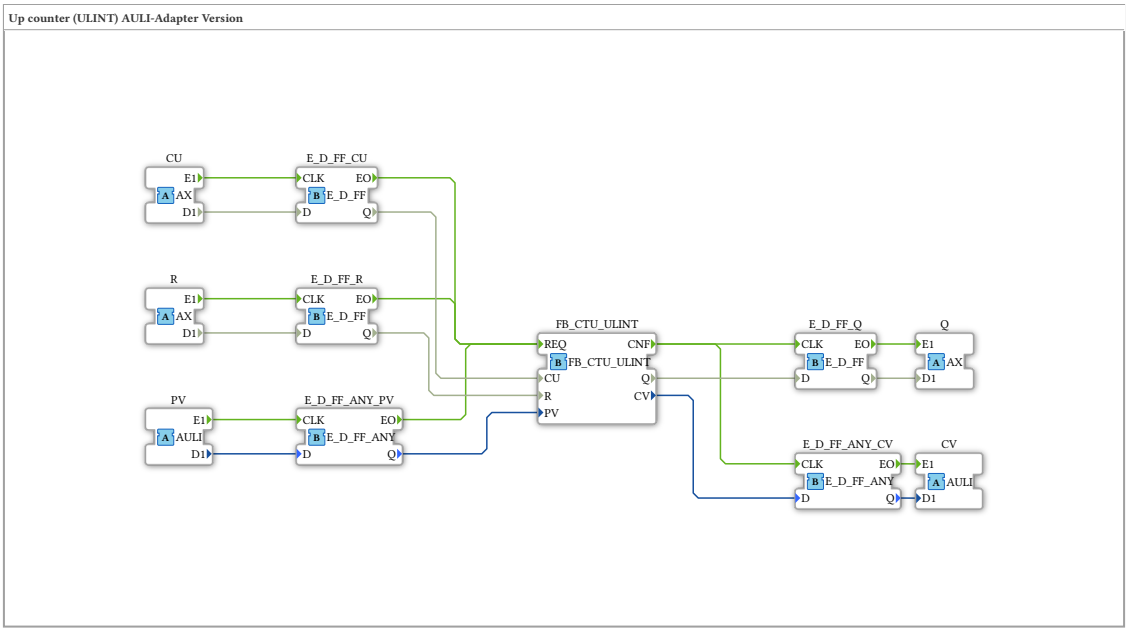
Der **AULI_FB_CTD** ist ein leistungsfähiger Abwärtszähler für 64-Bit-Werte, der über Adapter-Schnittstellen in 4diac-Projekte integriert wird. Er eignet sich besonders für große Zählbereiche und modulare Steuerungstopologien. Die ständige Ereignisausgabe erfordert eine sorgfältige Handhabung des nachfolgenden Netzwerks, kann aber durch geeignete Filterbausteine (z. B. `AX_D_FF`) gezähmt werden. Dank der Kapselung des bewährten `FB_CTD_ULINT` ist die Funktionsweise robust und standardkonform.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AULI_FB_CTU





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AULI_FB_CTU** realisiert einen Aufwärtszähler (Up Counter) auf Basis des Datentyps ULINT (unsigned long integer). Er ist als IEC 61499-konformer Baustein mit einer **AULI-Adapter-Schnittstelle** ausgelegt. Der Zähler verwendet intern einen standardisierten IEC 61131-3-CTU-Baustein (`FB_CTU_ULINT`) und stellt dessen Funktionalität über Adapterports zur Verfügung. Die Ereignisauslösung erfolgt bei jedem Update – unabhängig von einer Wertänderung.

Schnittstellenstruktur

Ereignis-Eingänge

Keine eigenständigen Ereignis-Eingänge. Ereignisse werden ausschließlich über die **AX-Adapter** der Sockets (CU, R, PV) empfangen.

Ereignis-Ausgänge

Name	Typ	Kommentar
CNF	Event	Bestätigung der Ausführung (Execution Confirmation) – wird bei jeder Aktualisierung des Zählers ausgelöst

Daten-Eingänge

Keine direkten Daten-Eingänge. Die benötigten Daten (CU-Impuls, Reset-Signal, Preset-Wert) werden über die **AULI-bzw. AX-Adapter** der Sockets bereitgestellt.

Daten-Ausgänge

Keine direkten Daten-Ausgänge. Die Ergebniswerte (aktueller Zählerstand CV und Schwellwert-Status Q) werden über die **AULI-bzw. AX-Adapter** der Plugs ausgegeben.

Adapter

Richtung	Adapter-Name	Typ	Kommentar
Socket (Eingang)	CU	`adapter::types::unidirectional::AX`	Count-Up-Impuls (Ereignis + Daten)
Socket (Eingang)	R	`adapter::types::unidirectional::AX`	Reset-Impuls (Ereignis + Daten)
Socket (Eingang)	PV	`adapter::types::unidirectional::AULI`	Preset-Wert (Vergleichswert für Q-Ausgang)
Plug (Ausgang)	Q	`adapter::types::unidirectional::AX`	Zählerstand ≥ Preset? (Bool-Status)
Plug (Ausgang)	CV	`adapter::types::unidirectional::AULI`	Aktueller Zählerstand (ULINT)

Funktionsweise

Der Baustein kapselt einen IEC 61131-3-konformen CTU-Zähler (`FB_CTU_ULINT`). Über die **Socket-Adapter** werden sowohl Ereignisse als auch Daten empfangen:

- **CU-Aktualisierung**: Ein positiver Flanke an CU löst einen Zählschritt aus (CV = CV + 1).
- **R-Aktualisierung**: Ein Reset setzt den Zählerstand zurück (CV = 0).
- **PV-Aktualisierung**: Der Preset-Wert wird übernommen; bei jedem Zählschritt wird CV mit PV verglichen.

Das Ereignis an einem Socket wird intern auf den `REQ`-Eingang des eingebetteten CTU-Bausteins geführt. Nach der Verarbeitung wird das `CNF`-Ereignis des CTU-Blocks an alle drei Ausgänge weitergeleitet:

- **Q-Plug**: Gibt das Status-Ereignis (Q.E1) aus – wird **bei jedem Update** gesendet, unabhängig davon, ob sich der boolsche Wert geändert hat.
- **CV-Plug**: Gibt das Ereignis (CV.E1) aus – ebenfalls **bei jedem Update**.
- **CNF-Ereignis**: Wird als äußeres Ereignis bereitgestellt.

Die Datenflüsse sind analog: Die Daten der Sockets (CU.D1, R.D1, PV.D1) werden an die entsprechenden Daten-Eingänge des internen Zählers übergeben. Die Ausgabedaten (Q.D1, CV.D1) werden über die Plugs ausgegeben.

Technische Besonderheiten

- **Immer-Aktualisierung**: Der Baustein löst die Ausgangsereignisse (Q.E1, CV.E1, CNF) bei **jedem** eingehenden Ereignis (CU, R, PV) aus – auch wenn sich der Zählerstand oder der Q-Wert nicht ändert.
- **Filterempfehlung**: Soll nur bei tatsächlicher Wertänderung getriggert werden, wird der Einsatz eines **AX_D_FF**-Bausteins als Vorfilter empfohlen (siehe Kommentar in der Identifikation).
- **Datenbreite**: Der Zähler arbeitet mit dem Datentyp ULINT (64-Bit unsigned integer), geeignet für große Zählbereiche.
- **Adapter-Schnittstelle**: Alle Ein- und Ausgaben erfolgen über genormte Adapter (`AX` für boolesche/ereignisbasierte, `AULI` für ULINT-Werte). Dies erleichtert den modularen Aufbau und die Wiederverwendung in Adapter-basierten Projekten.
- **Keine Zustandsmaschine auf oberster Ebene**: Der Baustein ist rein transaktionsbasiert: Jedes eingehende Ereignis führt zu einer sofortigen Verarbeitung.

Zustandsübersicht

Der FB besitzt keinen eigenen Zustandsautomaten. Der interne `FB_CTU_ULINT` kann folgende Zustände durchlaufen (vereinfacht):

- **Idle**: Warten auf ein Ereignis.
- **Zählen (Increment)**: Bei CU-Ereignis wird der interne Zähler erhöht.
- **Reset**: Bei R-Ereignis wird der Zähler auf 0 gesetzt.
- **Preset-Update**: Bei PV-Ereignis wird der Vergleichswert aktualisiert.

Nach jeder Verarbeitung werden die Ausgänge aktualisiert und das CNF-Ereignis gesendet.

Anwendungsszenarien

- **Hochfrequente Impulszählung**: Erfassung von Zählimpulsen (z. B. Encoder, Taktsignale) mit 64-Bit-Auflösung.
- **Schwellwertüberwachung**: Zählen von Produkten oder Ereignissen bis zu einem einstellbaren Grenzwert (PV); bei Erreichen wird der Q-Ausgang aktiv.
- **Reset-gesteuerte Batch-Zählung**: Nach einem R-Impuls beginnt der Zähler von Null; die aktuelle Anzahl wird über CV ausgegeben.
- **Adapter-basierte Automatisierungsmodule**: Einfache Integration in Systeme, die auf AX/AULI-Adapter setzen (z. B. standardisierte E/A-Module).

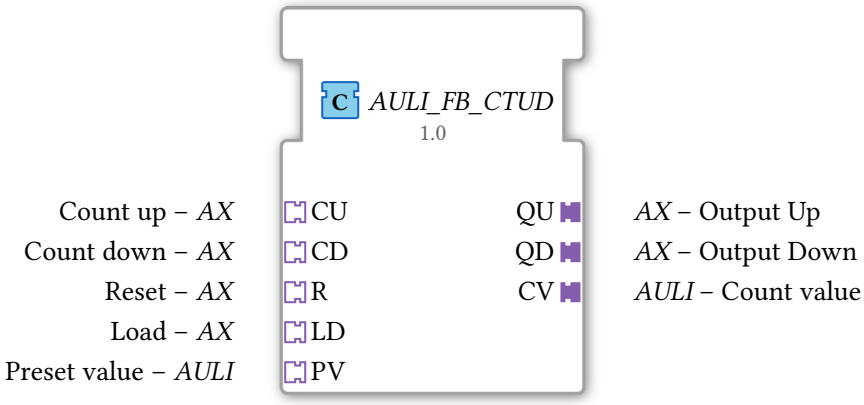
Vergleich mit ähnlichen Bausteinen

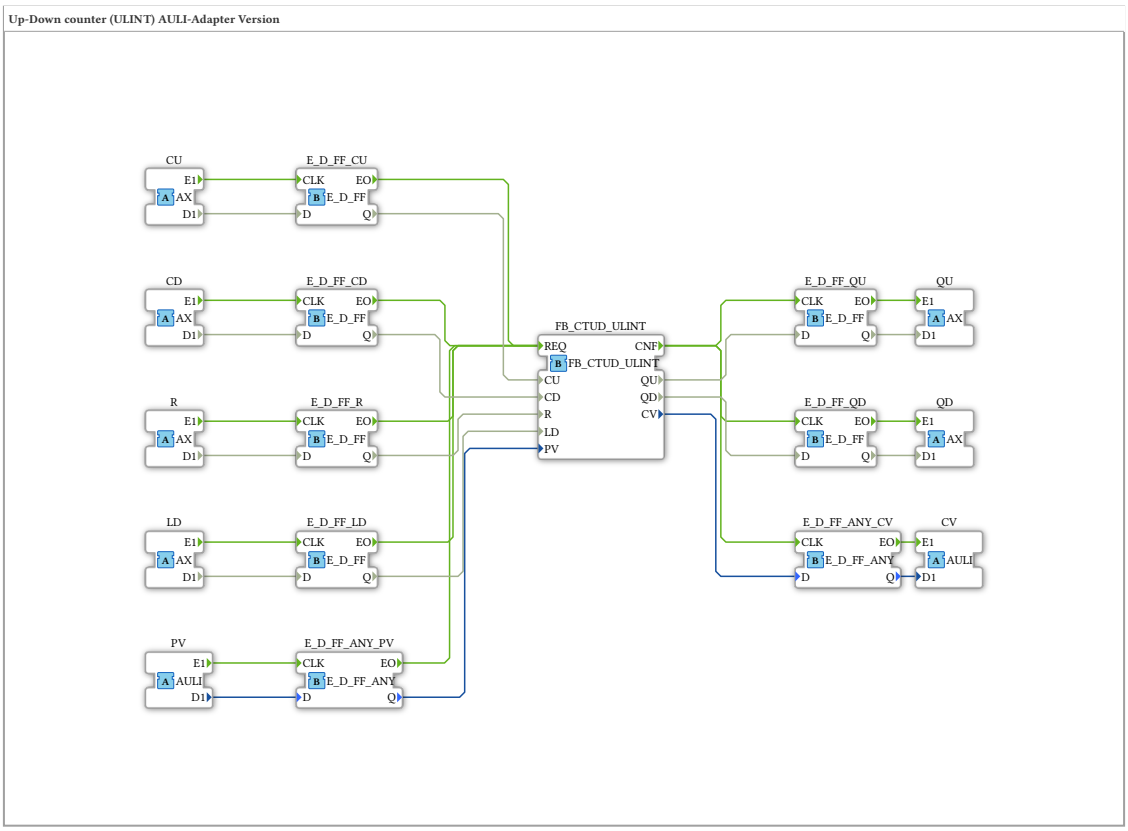
- **Standard-CTU (IEC 61131-3)**: Bietet in der Regel separate Ereignis- und Datenports. Der `AULI_FB_CTU` kapselt diese in Adaptern, was die Verkabelung auf Bus-Ebene verschlankt.
- **CTU mit Change-Detection**: Manche Zähler-Bausteine feuern Ausgangsereignisse nur bei Wertänderung. Der vorliegende FB löst bei jedem Update aus – dadurch entsteht ein deterministisches Verhalten (Ereignis pro Eingangsereignis), erfordert aber ggf. nachgeschaltete Filter.
- **CTU als Basic-FB**: Andere Implementierungen nutzen einen eigenen Zustandsautomaten; dieser FB delegiert die Logik an einen eingebetteten Block und bleibt dadurch schlank.

Fazit

Der **AULI_FB_CTU** bietet eine IEC 61499-kompatible, adapterbasierte Aufwärtszähler-Funktion mit ULINT-Datenbreite. Seine Stärken liegen in der modularen Schnittstelle und der zuverlässigen Ereignisauslösung bei jedem Update. Für Anwendungen, die eine reine Änderungsdetektion benötigen, ist ein externer Filterbaustein (AX_D_FF) empfehlenswert. Insgesamt eignet sich der Baustein für vielfältige Zählszenarien in der industriellen Automatisierung, besonders in Adapter-orientierten Architekturen.

AULI_FB_CTUD





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AULI_FB_CTUD** implementiert einen Aufwärts-/Abwärtszähler für vorzeichenlose 64-Bit-Ganzzahlen (ULINT). Er nutzt ausschließlich Adapter-Schnittstellen nach dem IEC 61499-2 Standard für Ereignis- und Datenverbindungen. Der Baustein kapselt den Standard-FB `FB_CTUD_ULINT` und erweitert dessen Ein- und Ausgänge um adapterbasierte Anschlüsse, was eine flexible und standardisierte Integration in adapterorientierte Architekturen ermöglicht.

Schnittstellenstruktur

Ereignis-Eingänge

Der Block verfügt über keine klassischen Ereigniseingänge. Stattdessen werden die Zählereignisse über die **Adapter-Sockets** übertragen:

- **CU** (Count Up) – Ereignis über `CU.E1`
- **CD** (Count Down) – Ereignis über `CD.E1`

- **R** (Reset) – Ereignis über `R.E1`
- **LD** (Load) – Ereignis über `LD.E1`
- **PV** (Preset Value) – Ereignis über `PV.E1`

Ereignis-Ausgänge

- **CNF** – Standard-Ereignisausgang, der bei jeder Zähleraktualisierung ausgelöst wird.

Zusätzlich werden Ereignisse über die **Adapter-Plugs** ausgegeben:

- **QU.E1** – Ereignis bei Zählerüberlauf (Maximalwert erreicht)
- **QD.E1** – Ereignis bei Zählerunterlauf (Wert 0)
- **CV.E1** – Ereignis zur Aktualisierung des aktuellen Zählerstands

Daten-Eingänge

Der Block besitzt keine separaten Dateneingänge. Alle Eingangsdaten werden über die Adapter-Sockets bereitgestellt:

- **CU.D1** (BOOL) – Aktivierung Inkrement (TRUE steigert den Zähler bei CU.E1)
- **CD.D1** (BOOL) – Aktivierung Dekrement
- **R.D1** (BOOL) – Aktivierung Reset
- **LD.D1** (BOOL) – Aktivierung Load
- **PV.D1** (ULINT) – Wert für den Load-Vorgang

Daten-Ausgänge

Es gibt keine separaten Datenausgänge. Die Ausgangsdaten werden über die Adapter-Plugs ausgegeben:

- **QU.D1** (BOOL) – Signalisiert Erreichen des Maximalwerts
- **QD.D1** (BOOL) – Signalisiert Erreichen von 0
- **CV.D1** (ULINT) – Aktueller Zählerstand

Adapter

Adapter	Richtung	Typ	Zweck
CU	Socket	AX	Count Up – Ereignis + Boolesche Freigabe
CD	Socket	AX	Count Down – Ereignis + Boolesche Freigabe
R	Socket	AX	Reset – Ereignis + Boolesche Freigabe
LD	Socket	AX	Load – Ereignis + Boolesche Freigabe
PV	Socket	AULI	Preset Value – Ereignis + ULINT-Wert
QU	Plug	AX	Output Up – Ereignis + Boolescher Zustand
QD	Plug	AX	Output Down – Ereignis + Boolescher Zustand
CV	Plug	AULI	Current Value – Ereignis + ULINT-Zählerstand

Funktionsweise

Der Baustein reagiert auf Ereignisse an den Eingangsadaptern (`CU.E1`, `CD.E1`, `R.E1`, `LD.E1`, `PV.E1`). Für jedes Ereignis wird der zugehörige Datenwert (`DX.D1`) ausgewertet:

- **CU**: Wenn `CU.E1` kommt und `CU.D1 = TRUE`, wird der Zähler inkrementiert.
- **CD**: Wenn `CD.E1` kommt und `CD.D1 = TRUE`, wird der Zähler dekrementiert.
- **R**: Wenn `R.E1` kommt und `R.D1 = TRUE`, wird der Zähler auf 0 zurückgesetzt.
- **LD**: Wenn `LD.E1` kommt und `LD.D1 = TRUE`, wird der Zähler auf den Wert von `PV.D1` gesetzt.

Nach jeder Verarbeitung wird das Ereignis `CNF` ausgegeben. Gleichzeitig werden die Ausgangsadapter aktualisiert:

- `QU.D1` wird `TRUE`, wenn der Zähler den Maximalwert (`2^64 - 1`) erreicht hat.
- `QD.D1` wird `TRUE`, wenn der Zählerstand 0 ist.
- `CV.D1` liefert den aktuellen Zählerstand.

Der gesamte Ablauf erfolgt synchron – jedes eingehende Ereignis löst eine Berechnung und anschließend die Ausgabe der Ergebnisse aus.

Technische Besonderheiten

- **Adapterbasierte Kommunikation**: Alle Ein- und Ausgaben erfolgen über Adapter (Sockets/Plugs). Dies ermöglicht eine losgelöste Verbindung zwischen Bausteinen und vereinfacht die Wiederverwendung in verschiedenen Kontexten.
- **Unidirektionale Adapter**: Die verwendeten Adapter (AX, AULI) sind unidirektional – Sockets empfangen, Plugs senden.
- **Triggerverhalten**: Der Baustein feuert die Ausgangsereignisse (`QU.E1`, `QD.E1`, `CV.E1`) bei **jeder** Zähleraktualisierung (auch bei Reset oder Load). Für eine änderungsbasierte Auslösung muss ein AX_D_FF (Differentiator) vorgeschaltet werden.
- **Wertebereich**: Der Zähler arbeitet im Bereich 0 … 2^64-1 (ULINT). Überläufe werden durch `QU` signalisiert, Unterläufe durch `QD`.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten; das Zählverhalten wird durch den internen `FB_CTUD_ULINT` realisiert. Im Wesentlichen werden drei Zustände unterschieden:

1. **Normalbetrieb** – Der Zählerwert liegt zwischen 1 und `2^64-2`. Weder `QU` noch `QD` sind aktiv.
2. **Überlauf** – Der Zähler hat den Maximalwert erreicht. `QU.D1 = TRUE`.
3. **Unterlauf** – Der Zählerstand ist 0. `QD.D1 = TRUE`.

Nach Reset (`R`) oder Load (`LD`) kann der Zähler sofort in einen dieser Zustände springen.

Anwendungsszenarien

- **Industrielle Stückzähler**: Erfassung von Produktionsmengen mit Auf- und Abwärtszählung (z. B. Gut-/Schlechtteile).
- **Paletten- oder Werkstückverfolgung**: Zählen von Ein- und Ausgängen in einem Pufferspeicher.
- **Ereignisgesteuerte Systeme**: Kombination mit Sensoren (Lichtschranken, Näherungsschalter) über die Adapter-Schnittstellen.
- **Adapter-basierte Steuerungsarchitekturen**: Nahtlose Integration in Projekte, die das Socket/Plug-Konzept von IEC 61499-2 nutzen.

Vergleich mit ähnlichen Bausteinen

Baustein	Eigenschaften
`FB_CTUD_ULINT`	Gleiche Zählerlogik, aber mit separaten Ereignis- und Dateneingängen/-ausgängen (keine Adapter).
`AULI_FB_CTU`	Nur Aufwärtszähler, als Adapterversion.
`AULI_FB_CTD`	Nur Abwärtszähler, als Adapterversion.
`CTUD` (Standard IEC 61499)	Ähnliche Funktionalität, aber meist mit anderen Datentypen (z. B. INT) und ohne Adapter.

Der `AULI_FB_CTUD` vereint Auf- und Abwärtszählung in einem Block und bietet durch die Adapter-Schnittstellen eine besonders flexible Anbindung an andere adapterbasierte Komponenten.

Fazit

Der `AULI_FB_CTUD` ist ein leistungsfähiger Aufwärts-/Abwärtszähler für ULINT-Werte, der sich durch seine vollständige Adapter-Schnittstelle auszeichnet. Er kombiniert bewährte Zähllogik mit der Flexibilität des IEC 61499-2 Adapterkonzepts. Dank der einheitlichen Ereignis- und Datenübertragung über Plugs und Sockets eignet er sich besonders für modulare, wiederverwendbare Steuerungsanwendungen. Die einfache Handhabung und die integrierte Überlauf-/Unterlauferkennung machen ihn zu einem zuverlässigen Grundbaustein in vielen industriellen Zählaufgaben.

edgeDetection

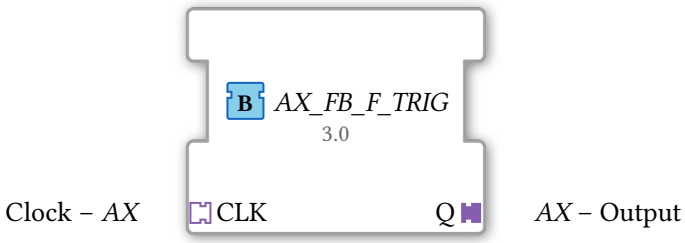
AX_FB_F_TRIG

Hier Platzhalter für das Bild des Funktionsblocks einfügen, falls vorhanden

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AX_FB_F_TRIG** (Falling Edge Trigger mit AX-Adapter) dient zur Erkennung einer fallenden Flanke (Signalwechsel von TRUE auf FALSE) innerhalb einer adapterbasierten Architektur. Er implementiert die Logik eines Standard-IEC 61131-3 `F_TRIG`-Bausteins, nutzt jedoch **AX-Adapter** zur Kapselung von Daten- und Ereignissignalen, um die Verdrahtungskomplexität in IEC 61499-Anwendungen zu reduzieren.



Schnittstellenstruktur

Der Baustein verzichtet auf klassische separate Ereignis- und Daten-Pins und nutzt stattdessen Adapter-Schnittstellen.

Ereignis-Eingänge

- *Keine direkten Ereignis-Eingänge (gekapselt im Adapter `CLK`).*

Ereignis-Ausgänge

- *Keine direkten Ereignis-Ausgänge (gekapselt im Adapter `Q`).*

Daten-Eingänge

- *Keine direkten Daten-Eingänge (gekapselt im Adapter `CLK`).*

Daten-Ausgänge

- *Keine direkten Daten-Ausgänge (gekapselt im Adapter `Q`).*

Adapter

Name	Typ	Komment	Beschreibung
CLK	`adapter::types::unidirectional::AX`	Clock	**Socket (Eingang):** Der Eingangsadapter, der das zu überwachende Signal liefert. Enthält typischerweise ein Ereignis (`E1`) und einen Booleschen Datenwert (`D1`).
Q	`adapter::types::unidirectional::AX`	Output	**Plug (Ausgang):** Der Ausgangsadapter, der das Ergebnis der Flankenerkennung sendet.

Funktionsweise

Der Baustein überwacht den logischen Zustand des Dateneingangs im Adapter `CLK` (Feld `D1`). Bei jedem Ereignis am Eingang `CLK` (`E1`) wird der Algorithmus ausgeführt:

1. Der Baustein prüft, ob ein Wechsel von **TRUE** (High) auf **FALSE** (Low) stattgefunden hat.
2. Dazu wird ein interner Speicher (`MEM`) genutzt, der den invertierten Zustand des letzten Zyklus speichert.
3. Die Logik lautet: `Q.D1 := NOT CLK.D1 AND NOT MEM`.
4. Der Speicher wird aktualisiert: `MEM := NOT CLK.D1`.

Verhalten:

- Wenn das Eingangssignal von 1 auf 0 wechselt, wird der Ausgang `Q.D1` für genau einen Zyklus auf TRUE gesetzt.
- In allen anderen Fällen (0->0, 0->1, 1->1) bleibt der Ausgang FALSE.
- Das Ergebnis wird über das Ereignis `Q.E1` signalisiert.

Technische Besonderheiten

- **Adapter-Technologie:** Durch die Verwendung des `AX`-Adapters (allgemeiner Datentyp, hier boolesch interpretiert) werden Ereignisfluss und Datenfluss gebündelt. Dies führt zu übersichtlicheren Funktionsplan-Diagrammen.
- **Interne Speicherung:** Der Baustein besitzt eine interne Variable `MEM` (Initialwert TRUE), um den vorherigen Zustand des Eingangssignals zu referenzieren.
- **Minimalistisches ECC:** Die Zustandsmaschine (ECC) besteht nur aus einem einzigen Zustand, was eine sehr schnelle und direkte Abarbeitung des Algorithmus gewährleistet.

Zustandsübersicht

Die Execution Control Chart (ECC) des Bausteins ist sehr einfach gehalten:

- **REQ (Zustand):** Dies ist der einzige Zustand.
  - **Transition:** Er wird durch das Ereignis `CLK.E1` (vom Eingangsadapter) ausgelöst.
  - **Aktion:** Führt den Algorithmus `REQ` aus und triggert anschließend das Ausgabeereignis `Q.E1`.

Anwendungsszenarien

Der **AX_FB_F_TRIG** eignet sich für diverse Steuerungsaufgaben, bei denen das „Abschalten“ oder „Loslassen“ erkannt werden muss:

- **Taster-Auswertung:** Erkennen, wann ein Taster losgelassen wird (negative Flanke).
- **Sensorik:** Auslösen einer Aktion, wenn ein Werkstück einen Sensorbereich verlässt (Lichtschranke wird frei).
- **Alarm-Rücksetzung:** Erkennung, wenn ein Fehlersignal nicht mehr anliegt.
- **Strukturierte Programmierung:** Einsatz in komplexen Anwendungen, die intensiv auf Adapter setzen, um „Spaghetti-Code“ (viele einzelne Verbindungslinien) zu vermeiden.

⚖️ Vergleich mit ähnlichen Bausteinen

Baustein	Unterschied
F_TRIG (Standard)	Nutzt separate `CLK` (BOOL) und `Q` (BOOL) Pins sowie separate `REQ`/`CNF` Ereignisse. Funktional identisch, aber andere Schnittstelle.
AX_FB_R_TRIG	Das Gegenstück zu diesem Baustein. Es erkennt die **steigende** Flanke (Rising Edge, 0 auf 1).
E_F_TRIG	Ereignisbasierter Trigger in IEC 61499, arbeitet oft rein auf Ereignisebene ohne gekapselte Datenadapter.

Fazit

Der **AX_FB_F_TRIG** ist ein essenzieller Baustein für Entwickler, die innerhalb der 4diac IDE und IEC 61499 auf adapterbasierte Designs setzen. Er bietet die verlässliche Funktionalität der klassischen Flankenerkennung, verpackt in einer modernen, verbindungssparenden Schnittstelle.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AX_FB_R_TRIG

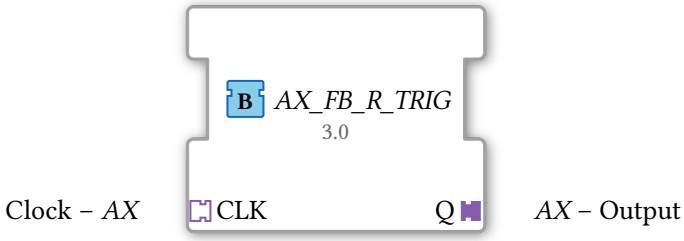
ECC: AX_FB_R_TRIG

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **AX_FB_R_TRIG** (Rising Trigger mit AX-Adapter) dient zur Erkennung einer steigenden Flanke eines booleschen Signals. Im Gegensatz zum Standard-`R_TRIG`-Baustein nach IEC 61131-3 verwendet dieser Baustein **AX-Adapter** für die Ein- und Ausgabe. Er kapselt die Logik zur Flankenerkennung in einer adapterbasierten Architektur, was die Verschaltung in Systemen vereinfacht, die `adapter::types::unidirectional::AX` verwenden.

Das Ziel ist es, ein `TRUE`-Signal am Ausgang zu erzeugen, wenn das Eingangssignal von `FALSE` auf `TRUE` wechselt.



Schnittstellenstruktur

Ereignis-Eingänge

Dieser Baustein verfügt über keine direkten Ereignis-Eingänge. Die Ereignissteuerung erfolgt über den Adapter `CLK`.

Ereignis-Ausgänge

Dieser Baustein verfügt über keine direkten Ereignis-Ausgänge. Die Ereignisweitergabe erfolgt über den Adapter `Q`.

Daten-Eingänge

Dieser Baustein verfügt über keine direkten Daten-Eingänge. Die Daten werden über den Adapter `CLK` empfangen.

Daten-Ausgänge

Dieser Baustein verfügt über keine direkten Daten-Ausgänge. Die Daten werden über den Adapter `Q` ausgegeben.

Adapter

Name	Typ	Kommentar
CLK	`adapter::types::unidirectional::AX`	**Socket (Eingang):** Der Takteingang (Clock), der überwacht wird. Enthält das Datensignal (D1) und das zugehörige Ereignis (E1).
Q	`adapter::types::unidirectional::AX`	**Plug (Ausgang):** Der Impulsausgang. Liefert `TRUE` nur bei einer positiven Flanke am Eingang `CLK`.

Funktionsweise

Der Baustein implementiert die klassische Logik eines „Rising Edge Detector“ (Erkennung steigender Flanken):

1. Der Baustein wartet auf ein Ereignis am Adapter-Eingang `CLK` (Signal `CLK.E1`).
2. Sobald das Ereignis eintritt, wird der Algorithmus `REQ` ausgeführt.
3. Der Algorithmus prüft den aktuellen Wert des Dateneingangs (`CLK.D1`) und vergleicht ihn mit dem gespeicherten Zustand aus dem vorherigen Zyklus (`MEM`).
   - **Logik:** `Q.D1 := CLK.D1 AND NOT MEM`
4. Wenn `CLK.D1` wahr (`TRUE`) ist und der gespeicherte Wert `MEM` falsch (`FALSE`) war, wird der Ausgang `Q.D1` auf `TRUE` gesetzt. In allen anderen Fällen ist `Q.D1` `FALSE`.
5. Der interne Speicher `MEM` wird anschließend mit dem aktuellen Wert von `CLK.D1` aktualisiert.
6. Abschließend wird das Ereignis am Ausgangsadapter `Q` (`Q.E1`) ausgelöst, um nachfolgende Bausteine zu informieren.

Technische Besonderheiten

- **Adapter-Kapselung:** Der Baustein ist speziell für den Typ `adapter::types::unidirectional::AX` ausgelegt. Dieser Adaptertyp bündelt typischerweise ein Ereignis (`E1`) und ein boolesches Datum (`D1`). Dies reduziert die sichtbare Verdrahtung im Anwendungsdiagramm, da Event- und Datenfluss in einer einzigen Verbindungslinie zusammengefasst sind.

- **Interner Speicher:** Der Baustein nutzt eine interne boolesche Variable `MEM`, um den Zustand des Eingangssignals zwischen zwei Ausführungen zu speichern.

Zustandsübersicht

Der Baustein basiert auf einer sehr einfachen Execution Control Chart (ECC) mit nur einem Zustand:

- **Zustand `REQ`:** Dies ist der einzige Zustand.
  - **Transition:** Der Baustein verbleibt in diesem Zustand und reagiert auf das Eingangsadapter-Ereignis `CLK.E1`.
  - **Aktion:** Bei jedem Ereignis wird der Algorithmus `REQ` ausgeführt und das Ausgangsadapter-Ereignis `Q.E1` gesendet.

Anwendungsszenarien

- **Signalkonditionierung:** Umwandlung eines dauerhaften Signals (z.B. Schalter ist AN) in einen einzelnen Impuls, um eine Aktion genau einmal beim Einschalten auszulösen.
- **Verdrahtungsvereinfachung:** Einsatz in Systemen, die konsequent auf Adapter-Technologie setzen, um die Anzahl der sichtbaren Verbindungen zwischen Funktionsblöcken zu minimieren.
- **Start-Trigger:** Erkennung, wann ein Prozess oder eine Maschine gestartet wird (Übergang von 0 auf 1).

⚖️ Vergleich mit ähnlichen Bausteinen

- **`R_TRIG` (Standard):** Der funktionale Kern ist identisch. Der Unterschied liegt in der Schnittstelle: `R_TRIG` nutzt separate `CLK` (BOOL) und `REQ` (Event) Eingänge sowie `Q` (BOOL) und `CNF` (Event) Ausgänge. `AX_FB_R_TRIG` nutzt Adapter.
- **`AX_FB_F_TRIG`:** Das Gegenstück zu diesem Baustein, welches auf fallende Flanken (High-to-Low) reagiert.

Fazit

Der **AX_FB_R_TRIG** ist ein essenzieller Utility-Baustein für Entwickler, die innerhalb der 4diac-Umgebung mit AX-Adaptern arbeiten. Er bietet die robuste Funktionalität der Standard-Flankenerkennung, verpackt in einer modernen, adapterbasierten Schnittstelle, was zu aufgeräumteren und modulareren Steuerungsanwendungen führt.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

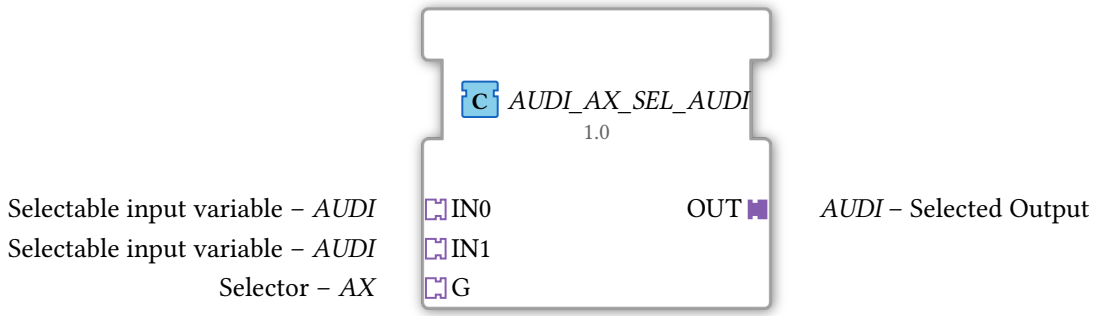
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

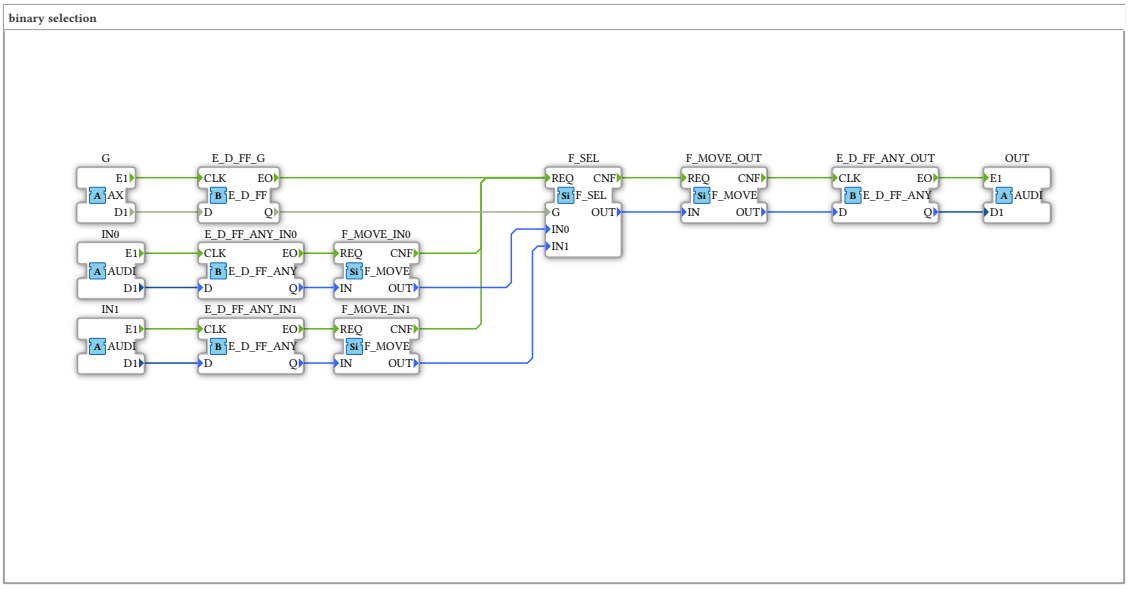
selection

In dieser Kategorie finden Sie Bausteine zur Signalauswahl und Extremwertbildung für Adapter-Schnittstellen:

- **AR_AX_SEL_AR**: Binäre Auswahl zwischen zwei AR-Eingängen basierend auf einem AX-Selektor.
- **AR_MAX**: Ermittlung des Höchstwerts (Maximum) aus zwei AR-Eingängen.
- **AR_MIN**: Ermittlung des Mindestwerts (Minimum) aus zwei AR-Eingängen.

AUDI_AX_SEL_AUDI





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUDI_AX_SEL_AUDI** realisiert eine binäre Auswahl (binary selection) zwischen zwei Eingabewerten. Er fungiert als Multiplexer, der abhängig von einem Selektorsignal einen der beiden Eingänge an den Ausgang durchschaltet. Die Kommunikation erfolgt ausschließlich über unidirektionale Adapter, die sowohl Ereignis- als auch Datensignale kapseln. Dadurch wird eine flexible und typengebundene Einbindung in existierende Signalpfade ermöglicht.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine dedizierten Ereignis-Eingänge. Ereignisse werden jedoch über den **Selektor-Adapter G** bereitgestellt:

- **G.E1**: Ereignissignal, das die Auswahl logik auslöst.

Ereignis-Ausgänge

Der FB besitzt keine dedizierten Ereignis-Ausgänge. Ereignisse werden jedoch über den **Ausgangs-Adapter OUT** ausgegeben:

- **OUT.E1**: Quittierungssignal, das nach erfolgreicher Durchschaltung des gewählten Eingangs ausgelöst wird.

Daten-Eingänge

Der FB besitzt keine dedizierten Daten-Eingänge. Datenwerte werden über die folgenden Adapter bereitgestellt:

- **IN0.D1** (Typ: AUDI) – erster auswählbarer Datenwert
- **IN1.D1** (Typ: AUDI) – zweiter auswählbarer Datenwert
- **G.D1** (Typ: AX) – Selektorfeld zur Bestimmung der aktiven Quelle

Daten-Ausgänge

Der FB besitzt keine dedizierten Daten-Ausgänge. Der ausgewählte Datenwert wird über den **Ausgangs-Adapter OUT** ausgegeben:

- **OUT.D1** (Typ: AUDI) – ausgewählter Datenwert (entspricht IN0.D1 oder IN1.D1)

Adapter

Adapter	Richtung	Typ	Kommentar

IN0	Socket (Eingang)	`adapter::types::unidirectional::AUDI`	Erster auswählbarer Eingang
IN1	Socket (Eingang)	`adapter::types::unidirectional::AUDI`	Zweiter auswählbarer Eingang
G	Socket (Eingang)	`adapter::types::unidirectional::AX`	Selektor (Ereignis + Daten)
OUT	Plug (Ausgang)	`adapter::types::unidirectional::AUDI`	Ausgangsfeld mit Ereignis und Daten

Funktionsweise

Der Baustein enthält intern eine Instanz des IEC-61131-Funktionsbausteins `F_SEL` (aus der Bibliothek `iec61131::selection`). Die Logik läuft wie folgt ab:

1. **Triggerung**: Ein Ereignis am **G.E1** (über den Selektordapter) startet die Verarbeitung.
2. **Datensammlung**: Die anliegenden Werte von **IN0.D1**, **IN1.D1** und **G.D1** werden an den `F_SEL`-Baustein weitergeleitet.
3. **Auswahl**: `F_SEL` prüft den Wert von **G.D1**. Typische Semantik (abhängig vom Datentyp `AX`):
   - Ist der Selektorfeld **False** (oder 0), wird **IN0** ausgewählt.
   - Ist das Selektorfeld **True** (oder ungleich 0), wird **IN1** ausgewählt.
4. **Ausgabe**: Das ausgewählte Signal wird an **OUT.D1** weitergegeben. Gleichzeitig wird ein Ereignis an **OUT.E1** gesendet, um die nachgeschaltete Verarbeitung zu informieren.

Der interne Ablauf ist ereignisgesteuert und vermeidet kontinuierliche Lesezugriffe, was eine effiziente Kommunikation in zeitgesteuerten Umgebungen ermöglicht.

Technische Besonderheiten

- **Reine Adapter-Kommunikation**: Alle Schnittstellen sind als unidirektionale Adapter definiert. Ereignis- und Datenpfade sind innerhalb eines Adapters zusammengefasst, was die Konfiguration in der 4diac-IDE vereinfacht.
- **Wiederverwendung von IEC-61131-Bibliotheken**: Die Auswahllogik wird vollständig durch den Standard-Baustein `F_SEL` realisiert, der keine zusätzliche Zustandsautomaten oder komplexe Algorithmen erfordert.
- **Typensicherheit**: Die Adapter sind auf die benutzerdefinierten Typen `AUDI` und `AX` spezialisiert. Dadurch wird eine Fehlkopplung mit inkompatiblen Signalen bereits zur Entwurfszeit verhindert.
- **Kein eigener Zustand**: Der Baustein selbst ist zustandslos – er leitet lediglich die Signale an den internen `F_SEL` weiter. Der interne FB verwaltet die Auswahl logik autonom.

Zustandsübersicht

Der FB **AUDI_AX_SEL_AUDI** besitzt **keinen eigenen Zustandsautomaten**. Die gesamte auswahl logik wird von dem inneren `F_SEL`-Baustein ausgeführt, der seinerseits einen reinen Funktionsblock ohne Zustandspeicher darstellt. Das Verhalten ist therefore **zustandslos**:

- Nach dem Empfang eines Ereignisses an **G.E1** wird die Auswahl sofort getroffen und das Ergebnis in **OUT.D1** ausgegeben.
- Es existieren keine Initialisierungs-, Warte- oder Fehlerzustände; der Baustein ist jederzeit bereit für eine neue Selektoranforderung.

Anwendungsszenarien

Der Baustein eignet sich ideal für folgende Einsatzfälle in der Automatisierungstechnik:

- **Signalumschaltung** zwischen zwei Sensoren (z. B. Temperaturfühler, Druckaufnehmer) abhängig von einer Betriebsart oder einem manuellen Schalter.
- **Bypass-Selektion** in Steuerungspfaden: bei Störung eines Kanals wird automatisch auf einen Ersatzkanal umgeschaltet.
- **Parameterauswahl**: zwischen zwei festen Parametersätzen (z. B. für unterschiedliche Produkte) wird anhand eines Selektorfeldes gewählt.
- **Testmodus**: Umschaltung zwischen einem realen und einem simulierten Signal für Inbetriebnahme oder Wartung.

Die Verwendung benutzerdefinierter Adaptertypen (`AUDI`, `AX`) erlaubt eine einfache Anpassung an projektspezifische Datenstrukturen.

Vergleich mit ähnlichen Bausteinen

Kriterium	**AUDI_AX_SEL_AUDI**	Einfacher Bool-MUX (z. B. `F_SEL`)
Schnittstelle	Reine Adapter (Ereignis+Daten gekapselt)	Getrennte Ein-/Ausgangspins
Datentyp	Benutzerdefinierte Typen (`AUDI`, `AX`)	Beliebig (parametrierbar)
Selektor	Ereignisgesteuert (über Adapter)	Datenbasiert, ohne extra Ereignis
Wiederverwendbarkeit	Höher durch Typensicherheit und Kapselung	Flexibler, aber lose Kopplung
Komplexität	Mittel (Adapter erfordern Vorwissen)	Niedrig (direkte Pins)

Der **AUDI_AX_SEL_AUDI** bietet eine höhere Integration in adapterbasierte Architekturen und eignet sich besonders für modulare, typengebundene Systeme, während ein reiner `F_SEL` universeller ist.

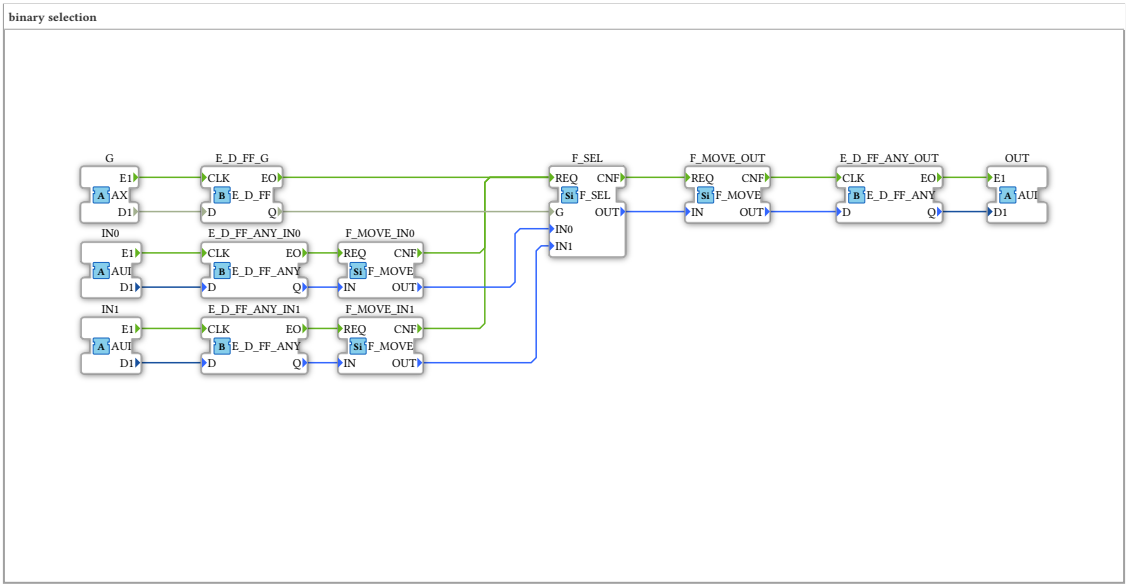
Fazit

Der **AUDI_AX_SEL_AUDI** ist ein spezialisierter binärer Selektor, der die Vorteile der adapterbasierten Kommunikation in 4diac nutzt. Durch die Kapselung von Ereignis- und Datenpfaden in einem Adapter wird die Signalverdrahtung im Gesamtsystem reduziert und die Typsicherheit erhöht. Die interne Delegation an den bewährten `F_SEL`-Baustein der IEC-61131-Bibliothek gewährleistet eine robuste, zustandslose Auswahl logik. Der Baustein eignet sich hervorragend für den Einsatz in modularen Steuerungsprojekten, bei denen klare Schnittstellen und einfache Wiederverwendbarkeit im Vordergrund stehen.

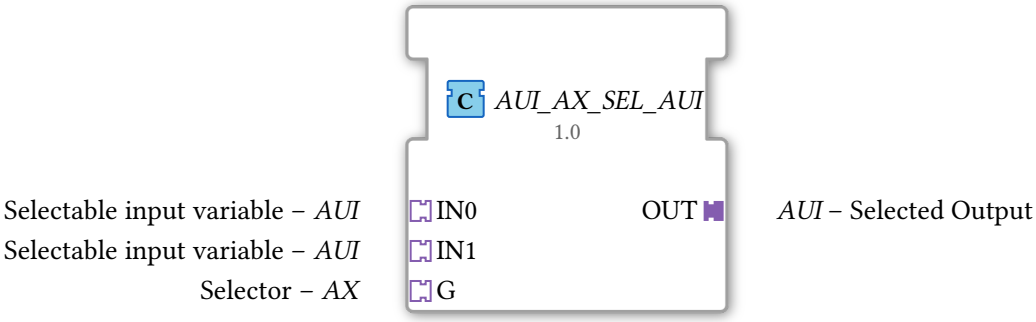
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUI_AX_SEL_AUI



binary selection



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUI_AX_SEL_AUI** realisiert eine binäre Auswahl zwischen zwei Eingangswerten vom Typ **AUI**. Die Entscheidung, welcher Eingang an den Ausgang weitergeleitet wird, erfolgt über einen dritten Adapter vom Typ **AX**, der als Selektor dient. Der Baustein kapselt die in IEC 61131-3 definierte Funktion **F_SEL** und stellt sie als adapterbasierte Komponente bereit.

Schnittstellenstruktur

Ereignis-Eingänge

- **G.E1** (über Adapter G): Triggert die Auswahlverarbeitung, sobald ein Ereignis am Selektor-Adapter anliegt.

Ereignis-Ausgänge

- **OUT.E1** (über Adapter OUT): Wird ausgelöst, nachdem der ausgewählte Wert am Ausgang anliegt.

Daten-Eingänge

- **G.D1** (über Adapter G): Der selektierende Wert (z. B. BOOL oder numerisch), der bestimmt, ob **IN0** oder **IN1** ausgewählt wird.
- **IN0.D1** (über Adapter IN0): Erster Eingangswert vom Typ **AUI**.
- **IN1.D1** (über Adapter IN1): Zweiter Eingangswert vom Typ **AUI**.

Daten-Ausgänge

- **OUT.D1** (über Adapter OUT): Der ausgewählte Wert (Typ **AUI**), entsprechend der Selektionslogik.

Adapter

- **G** (Socket, Typ `adapter::types::unidirectional::AX`): Liefert das Selektionssignal.
- **IN0** (Socket, Typ `adapter::types::unidirectional::AUI`): Erster Datenkanal.
- **IN1** (Socket, Typ `adapter::types::unidirectional::AUI`): Zweiter Datenkanal.
- **OUT** (Plug, Typ `adapter::types::unidirectional::AUI`): Ergebniskanal.

Funktionsweise

Der Baustein arbeitet ereignisgesteuert über die Adapter-Schnittstellen. Sobald ein Ereignis am Selektor **G.E1** eintrifft, wird der interne Funktionsblock **F_SEL** aktiviert. **F_SEL** prüft den Wert **G.D1**:

- Ist **G.D1** = `FALSE` (oder 0), wird der Wert von **IN0.D1** an den Ausgang **OUT.D1** weitergegeben.
- Ist **G.D1** = `TRUE` (oder ungleich 0), wird **IN1.D1** ausgewählt.

Nach Abschluss der Selektion signalisiert **F_SEL.CNF** das Ereignis **OUT.E1**, um den Verbraucher über den neuen gültigen Ausgangswert zu informieren.

Technische Besonderheiten

- Der Baustein ist als **Composite-FB** realisiert – das Verhalten ergibt sich aus der inneren Verschaltung eines standardisierten **F_SEL**-Blocks (IEC 61131-3).
- Die Daten werden über Adapter vom Typ **AUI** und **AX** übertragen. Diese Adaptertypen sind als unidirektionale Kanäle definiert, deren interne Datenstruktur nicht näher spezifiziert ist (z. B. Integer, Real, oder benutzerdefinierte Structs).

- Die Auswahl erfolgt rein kombinatorisch, jedoch erst nach einem auslösenden Ereignis am Selektor. Ein reiner Datenwechsel ohne Ereignis führt nicht zu einer Aktualisierung des Ausgangs.

Zustandsübersicht

Da es sich um einen Composite-FB ohne eigenen Zustandsautomaten handelt, existieren keine expliziten Zustände. Die Logik ist rein ereignisgesteuert:

1. **Warte auf Ereignis** – solange kein **G.E1** eintrifft, bleibt der Ausgang unverändert.
2. **Auswahl durchführen** – nach **G.E1** wird der entsprechende Eingang ausgewählt und der Ausgang gesetzt.
3. **Ausgang signalisieren** – **OUT.E1** wird nach erfolgreicher Auswahl generiert.

Anwendungsszenarien

- **Umschaltung zwischen zwei Messwerten** (z. B. Temperatur von Sensor A oder B) basierend auf einem Steuersignal.
- **Betriebsartumschaltung** in der Automatisierungstechnik, bei der je nach Modus unterschiedliche Stellgrößen aktiviert werden.
- **Redundante Signalquellen** – falls der primäre Kanal (IN0) ausfällt, kann über den Selektor auf den sekundären Kanal (IN1) umgeschaltet werden.

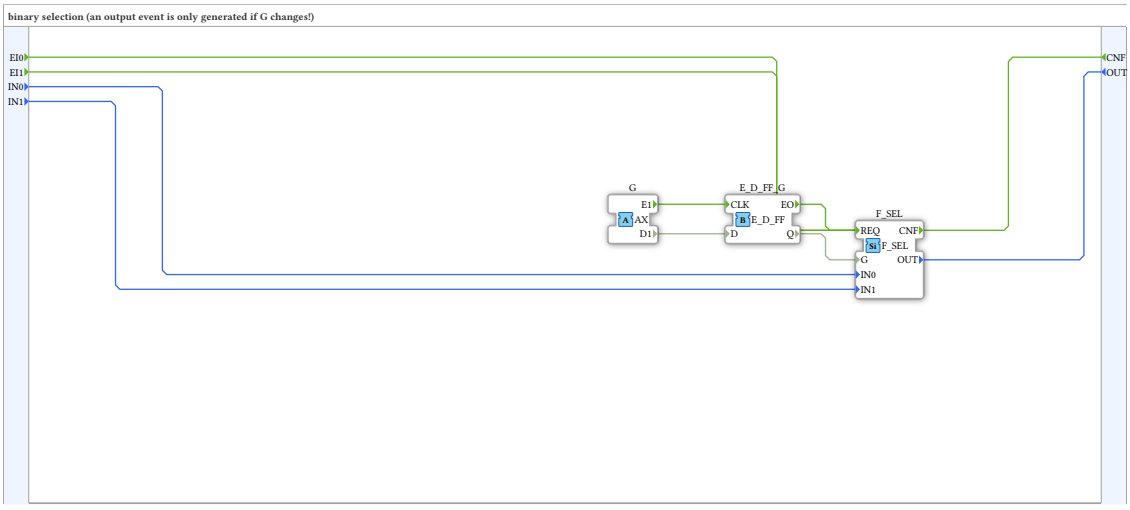
Vergleich mit ähnlichen Bausteinen

- **MUX (Multiplexer)**: Ein Multiplexer wählt aus mehr als zwei Eingängen aus; **AUI_AX_SEL_AUI** ist auf zwei Eingänge spezialisiert und verwendet Adapter statt einzelner Signale.
- **F_SEL (Standard IEC 61131)**: Der innere Baustein bietet identische Funktionalität, jedoch ohne Adapter-Kapselung. Der vorliegende FB erweitert ihn um eine adapterbasierte Schnittstelle, was die Wiederverwendung in adapterorientierten Architekturen erleichtert.
- **SEL (Selection)**: Ähnliche Logik, aber meist als reiner Funktionsbaustein ohne Ereignissteuerung definiert. Hier wird die Auswahl explizit durch ein Ereignis ausgelöst.

Fazit

AUI_AX_SEL_AUI ist ein adapterbasierter Composite-FB für die einfache binäre Auswahl zwischen zwei AUI-Datenströmen. Durch die Verwendung von `F_SEL` im Inneren wird eine robuste und standardkonforme Selektionslogik gewährleistet. Der Baustein eignet sich besonders für modulare Automatisierungslösungen, bei denen Daten über unidirektionale Adapter ausgetauscht werden.

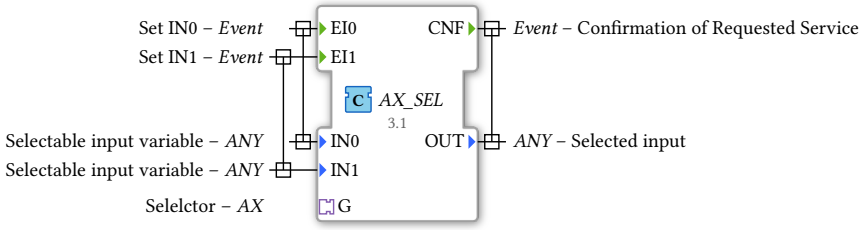
AX_SEL



• – ○ • – ○ • – ○ •

Einleitung

Der AX_SEL Funktionsblock ist ein binärer Selektor, der zur Auswahl zwischen zwei Eingangswerten dient. Er gehört zur Kategorie der Standard-Selektionsfunktionen gemäß IEC 61131-3 und ermöglicht die flexible Auswahl von Daten basierend auf einem Steuersignal.



Schnittstellenstruktur

Ereignis-Eingänge

- **EI0** - Setzt IN0 (Datenzuordnung: `IN0`)
- **EI1** - Setzt IN1 (Datenzuordnung: `IN1`)

Ereignis-Ausgänge

- **CNF** - Bestätigung des angeforderten Dienstes (Confirmation of Requested Service)

Daten-Eingänge

- **IN0** (ANY) - Selektierbare Eingangsvariable
- **IN1** (ANY) - Selektierbare Eingangsvariable

Daten-Ausgänge

- **OUT** (ANY) - Ausgewählter Eingang

Adapter

- **G** (adapter::types::unidirectional::AX) - Unidirektionaler Adapter für Steuerung

Funktionsweise

Der AX_SEL Funktionsblock wählt zwischen zwei Eingangswerten (IN0 und IN1) aus und gibt den selektierten Wert am Ausgang OUT aus. Die Auswahl wird über den angeschlossenen Adapter G gesteuert: `G.E1` taktet ein internes D-Flipflop (`E_D_FF_G`), das `G.D1` übernimmt und als Auswahl `G` an `F_SEL` liefert. `F_SEL` wird durch das Flipflop-Ereignis (G hat sich geändert) sowie direkt durch `EI0` und `EI1` angestoßen; sein `CNF` wird als `CNF` des Bausteins ausgegeben und der ausgewählte Wert steht dann an OUT bereit. Ein Ausgangsereignis entsteht also nur bei einem Ereignis an `G.E1`, `EI0` oder `EI1`.

Technische Besonderheiten

- Verwendet den ANY-Datentyp für maximale Flexibilität bei den Ein- und Ausgängen
- Implementiert als unidirektionaler Adapter für einfache Steuerung
- Unterstützt verschiedene Datentypen durch die Verwendung von ANY

Zustandsübersicht

Der Funktionsblock besitzt einen einfachen Zustand:

- **Bereit**: Wartet auf Steuersignal vom Adapter G
- Bei Empfang eines Ereignisses (`EI0`, `EI1` oder `G.E1`) wird der entsprechende Eingang ausgewählt und das CNF-Ereignis ausgelöst

Anwendungsszenarien

- Auswahl zwischen zwei verschiedenen Sensoren
- Umschaltung zwischen Betriebsmodi
- Selektion von Datenquellen in Abhängigkeit von Betriebszuständen
- Wechsel zwischen manueller und automatischer Steuerung

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Selektionsbausteinen bietet AX_SEL:

- Größere Flexibilität durch ANY-Datentyp
- Einfache Integration durch unidirektionalen Adapter
- Standardisierte Schnittstelle gemäß IEC 61131-3

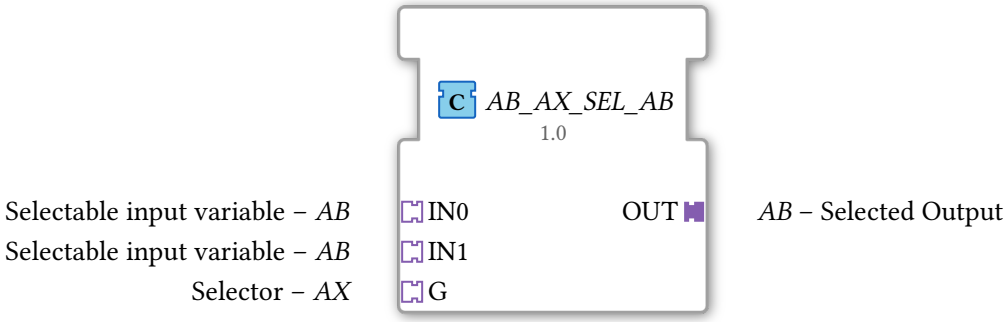
Vergleich mit F_SEL

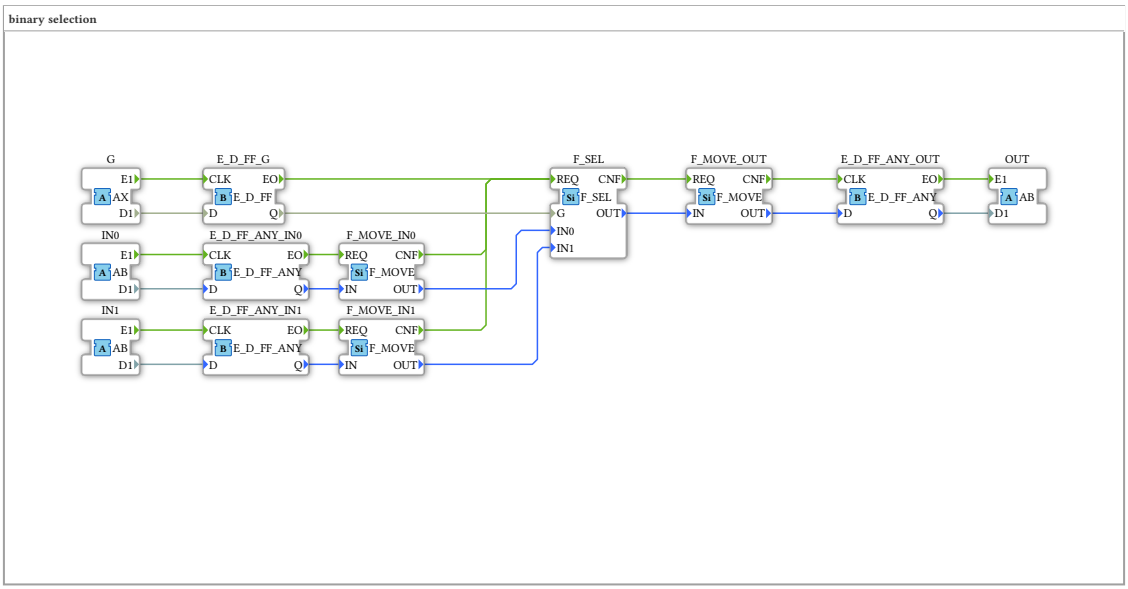
Fazit

Der AX_SEL Funktionsblock stellt eine einfache und flexible Lösung für binäre Auswahlaufgaben in Steuerungssystemen dar. Durch die Verwendung des ANY-Datentyps und standardisierter Adapter-Schnittstellen ermöglicht er eine vielseitige Anwendung in verschiedenen Automatisierungsprojekten.

Adapter

AB_AX_SEL_AB





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AB_AX_SEL_AB` ist ein zusammengesetzter Funktionsbaustein (Composite Function Block) für die IEC 61499, der eine binäre Auswahl (Selektion) auf Adapter-Ebene realisiert. Er dient dazu, basierend auf einem Steuersignal eines Selektor-Adapters (`G`), zwischen den Signalen zweier Eingangs-Adapter (`IN0` und `IN1`) auszuwählen und das ausgewählte Signal an einen Ausgangs-Adapter (`OUT`) weiterzuleiten.

Dieser Baustein eignet sich besonders für ereignisgesteuerte Architekturen, bei denen Datenströme flexibel geroutet werden müssen, ohne die Kapselung der Adapter-Schnittstellen aufzubrechen.

Schnittstellenstruktur

Da es sich um einen adapterbasierten Funktionsbaustein handelt, sind die klassischen Ereignis- und Dateneingänge vollständig in den Adapterschnittstellen gekapselt.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die Steuerung erfolgt über die Ereignisse innerhalb der Adapter.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Signalisierung erfolgt über die Ereignisse innerhalb des Ausgangs-Adapters.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Name	Richtung	Typ	Beschreibung
OUT	Plug (Consumer)	`adapter::types::unidirectional::AB`	Der ausgewählte Ausgang (Selected Output). Gibt das weitergeleitete Signal aus.
IN0	Socket (Provider)	`adapter::types::unidirectional::AB`	Erster auswählbarer Eingang (wird selektiert, wenn der Selektor `G` den Zustand `FALSE` hat).
IN1	Socket (Provider)	`adapter::types::unidirectional::AB`	Zweiter auswählbarer Eingang (wird selektiert, wenn der Selektor `G` den Zustand `TRUE` hat).
G	Socket (Provider)	`adapter::types::unidirectional::AX`	Selektor-Adapter (Steuersignal für die Auswahl).

Hinweis zu den Adaptertypen:

- `AB` steht für einen unidirektionalen Adapter, der ein Byte-Datensignal (`D1`/`D`) und ein Ereignis (`E1`) führt.
- `AX` steht für einen unidirektionalen Adapter, der ein boolesches Datensignal (`D1`/`D`) und ein Ereignis (`E1`) führt.

Funktionsweise

Im Inneren des Funktionsblocks befindet sich ein Netzwerk aus Standard-Funktionsbausteinen (IEC 61131-3 und IEC 61499), das die Logik abbildet:

1. **Ereignis- und Datenerfassung:**
   - Trifft an einem der Eingangs-Adapter (`IN0`, `IN1`, `G`) ein Ereignis (`E1`) ein, wird der jeweilige Wert (`D1`) über ein flankengetriggertes Speicherglied (D-Flip-Flop `E_D_FF` bzw. `E_D_FF_ANY`) zwischengespeichert.
2. **Datenkonvertierung / Pufferung:**
   - Die Bausteine `F_MOVE_IN0` und `F_MOVE_IN1` kopieren die Byte-Werte der Eingänge an das Auswahl-Element.
3. **Auswahl-Logik (`F_SEL`):**
   - Der Standard-Auswahlbaustein `F_SEL` (aus der IEC 61131-3 Library) übernimmt die eigentliche Selektion:
     - Ist das Signal aus dem Selektor-Adapter `G` gleich `FALSE`, wird das Signal von `IN0` an den Ausgang durchgeschaltet.
     - Ist das Signal aus dem Selektor-Adapter `G` gleich `TRUE`, wird das Signal von `IN1` an den Ausgang durchgeschaltet.
4. **Ausgabe:**
   - Der ausgewählte Wert wird über einen weiteren Konvertierungsbaustein (`F_MOVE_OUT` mit Datentyp `BYTE`) an ein Ausgangs-Flipflop (`E_D_FF_ANY_OUT`) übergeben.
   - Dieses generiert am Plug `OUT` das entsprechende Ausgangsereignis (`OUT.E1`) und stellt den selektierten Datenwert (`OUT.D1`) bereit.

Technische Besonderheiten

- **Ereignisgesteuertes Daten-Routing:** Jede Änderung an den Eingängen `IN0` oder `IN1` sowie am Selektor `G` triggert eine Aktualisierung und Berechnung des Ausgangs. Es erfolgt keine zyklische Abfrage (Polling).

- **Entkopplung durch Speicherglieder:** Durch den Einsatz von `E_D_FF` und `E_D_FF_ANY` wird sichergestellt, dass die Daten konsistent übernommen werden und keine unkontrollierten Signaländerungen (Race Conditions) am Ausgang entstehen.
- **Typisierung:** Die interne Verarbeitung der Daten ist fest auf den Datentyp `BYTE` ausgelegt (definiert über das Attribut `DataType="BYTE"` an den `F_MOVE`-Bausteinen).

Zustandsübersicht

Zustand Selektor `G`	Ereignis an Eingang	Verhalten am Ausgang `OUT`
`FALSE`	Ereignis an `IN0`	Wert von `IN0` wird an `OUT` übergeben; Ereignis `OUT.E1` wird ausgelöst.
`FALSE`	Ereignis an `IN1`	Wert von `IN1` wird intern aktualisiert, jedoch **nicht** an `OUT` weitergeleitet.
`TRUE`	Ereignis an `IN0`	Wert von `IN0` wird intern aktualisiert, jedoch **nicht** an `OUT` weitergeleitet.
`TRUE`	Ereignis an `IN1`	Wert von `IN1` wird an `OUT` übergeben; Ereignis `OUT.E1` wird ausgelöst.
Wechsel `FALSE` → `TRUE`	Ereignis an `G`	Der aktuell gespeicherte Wert von `IN1` wird an `OUT` durchgeschaltet; `OUT.E1` wird ausgelöst.
Wechsel `TRUE` → `FALSE`	Ereignis an `G`	Der aktuell gespeicherte Wert von `IN0` wird an `OUT` durchgeschaltet; `OUT.E1` wird ausgelöst.

Anwendungsszenarien

- **Signal-Umschaltung in der Feldebene:** Umschalten zwischen dem Signal eines aktiven Sensors (`IN1`) und einem vordefinierten Ersatzwert/Fehlerwert (`IN0`) im Falle einer Störung.
- **Hand-/Automatik-Umschaltung:** Weiterleitung von Steuerbytes, wobei im Automatikbetrieb (`G = TRUE`) Werte aus der automatischen Regelung verwendet werden, während im Handbetrieb (`G = FALSE`) manuell vorgegebene Werte an die Aktoren gesendet werden.
- **Rezeptur- oder Modusauswahl:** Dynamische Pfadauswahl für Konfigurations- oder Zustandsdaten in komplexen Prozesssteuerungen.

Vergleich mit ähnlichen Bausteinen

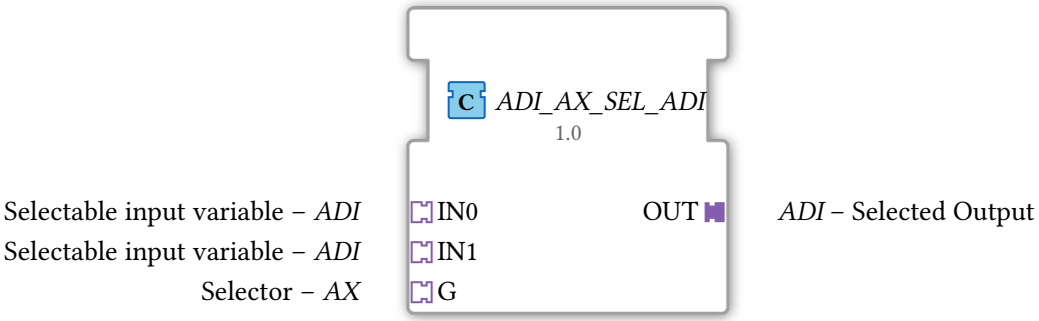
Im Vergleich zum Standard-Funktionsbaustein `F_SEL` arbeitet `AB_AX_SEL_AB` direkt auf Adapter-Ebene.

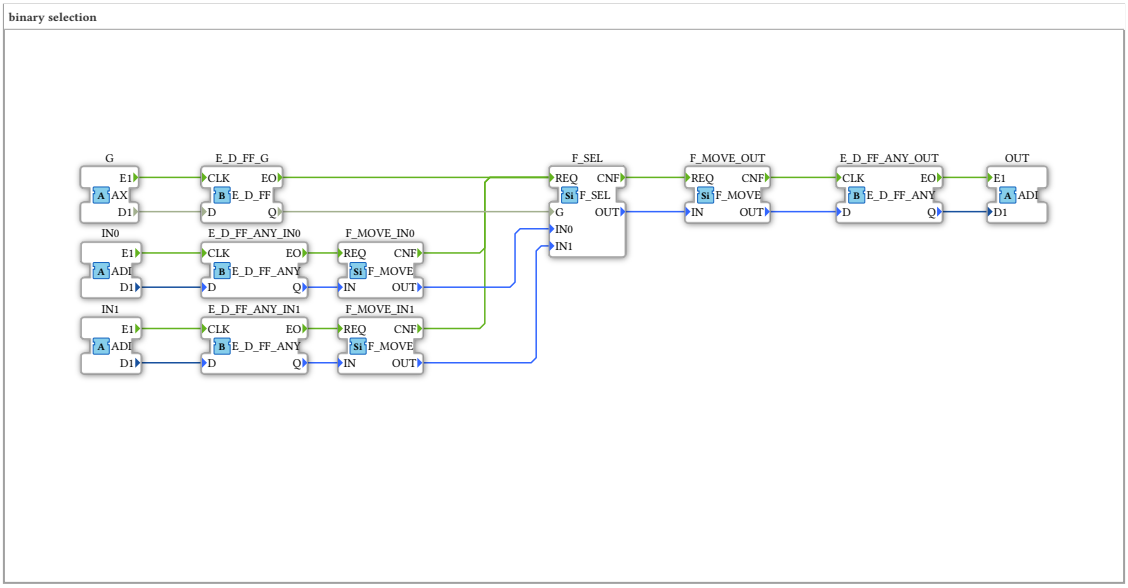
- Während bei `F_SEL` diskrete Datenleitungen und separate Trigger-Ereignisse manuell verdrahtet werden müssen, kapselt `AB_AX_SEL_AB` diese Logik.
- Das spart Engineering-Zeit, reduziert die Anzahl sichtbarer Verbindungen im 4diac-IDE Application Editor und minimiert Fehlerquellen bei der Verknüpfung von Daten- und Ereignisströmen.

Fazit

Der `AB_AX_SEL_AB` ist ein robuster und wiederverwendbarer Hilfsbaustein zur Signalroutung auf Basis von Byte-Adaptern. Durch die konsequente Kapselung der Auswahl- und Synchronisationslogik trägt er maßgeblich zur Übersichtlichkeit und Modularisierung von Steuerungsprogrammen in 4diac-IDE bei.

ADI_AX_SEL_ADI





Kein Bild verfügbar

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein `ADI_AX_SEL_ADI` dient der binären Auswahl (Selektion) zwischen zwei analogen bzw. ganzzahligen Eingangssignalen, die über unidirektionale Adapter übertragen werden. Basierend auf dem Zustand eines digitalen Auswahlsignals (Selectors) wird entweder der Wert des ersten oder des zweiten Eingangs auf den Ausgang durchgeschaltet.

Der Baustein ist als zusammengesetzter Funktionsbaustein (Composite FB) realisiert und nutzt intern standardisierte Auswahl- und Kopierfunktionen nach IEC 61131-3, die für die ereignisgesteuerte IEC 61499-Adapter-Architektur angepasst wurden.

Schnittstellenstruktur

Die Schnittstelle dieses Funktionsbausteins ist vollständig über Adapter realisiert. Es gibt keine direkten elementaren Daten- oder Ereigniseingänge außerhalb der Adapterstrukturen.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden (Ereignisse werden über die Adapter-Schnittstellen empfangen).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden (Ereignisse werden über die Adapter-Schnittstellen gesendet).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Plugs (Stecker - Ausgänge)

- **OUT** (Typ: `adapter::types::unidirectional::ADI`): Der ausgewählte Ausgang (Selected Output). Sendet das ausgewählte Datensignal (`D1`) und das zugehörige Aktualisierungsereignis (`E1`).

Sockets (Buchsen - Eingänge)

- **IN0** (Typ: `adapter::types::unidirectional::ADI`): Erster auswählbarer Eingang. Dieser Wert wird zum Ausgang durchgeschaltet, wenn das Auswahlsignal `G` auf `FALSE` (0) steht.
- **IN1** (Typ: `adapter::types::unidirectional::ADI`): Zweiter auswählbarer Eingang. Dieser Wert wird zum Ausgang durchgeschaltet, wenn das Auswahlsignal `G` auf `TRUE` (1) steht.
- **G** (Typ: `adapter::types::unidirectional::AX`): Der Selector (Gate) zur Steuerung des Auswahlprozesses. Ein Wechsel des Zustands steuert die Umschaltung zwischen `IN0` und `IN1`.

Funktionsweise

Die interne Struktur von `ADI_AX_SEL_ADI` arbeitet rein ereignisgesteuert:

1. **Eingangserfassung**: Trifft an einem der Eingänge (`IN0`, `IN1`) oder am Selector (`G`) ein Ereignis (`E1`) ein, wird der dazugehörige interne Datenwert (`D1`) über ereignisgesteuerte Speicherglieder (`E_D_FF_ANY` bzw. `E_D_FF`) erfasst.
2. **Datenpufferung**: Die Bausteine `F_MOVE_IN0` und `F_MOVE_IN1` kopieren die Daten (interpretiert als Datentyp `DINT`) an den zentralen Auswahlbaustein `F_SEL`.
3. **Auswahllogik (Selection)**: Der Baustein `F_SEL` (entspricht dem klassischen `SEL`-Standardbaustein aus der IEC 61131-3) wertet das Steuersignal `G` aus:
   - Ist das Signal des Adapters `G` gleich `FALSE`, wird das Signal von `IN0` gewählt.
   - Ist das Signal des Adapters `G` gleich `TRUE`, wird das Signal von `IN1` gewählt.
4. **Ausgabe**: Der ausgewählte Wert wird über `F_MOVE_OUT` an das Ausgangs-Flip-Flop `E_D_FF_ANY_OUT` übergeben. Dieses setzt den Wert `D1` am Ausgangsadapter `OUT` und triggert sofort das dazugehörige Ereignis `OUT.E1`, um nachgelagerte Bausteine über die Datenänderung zu informieren.

Technische Besonderheiten

- **Adapter-Kapselung**: Durch die Verwendung von Adaptern (`ADI` und `AX`) wird die Anzahl der sichtbaren Verbindungslinien im 4diac-Anwendungsdiagramm drastisch reduziert, was die Übersichtlichkeit und Wartbarkeit erhöht.
- **Datentyp-Spezifizierung**: Die internen Kopiervorgänge (`F_MOVE_IN0`, `F_MOVE_IN1` und `F_MOVE_OUT`) sind fest für den Datentyp `DINT` (Double Integer) konfiguriert.
- **Echtzeit-Durchreichung**: Der Baustein reagiert verzögerungsfrei auf jede Flanke der Eingangssignale sowie des Auswahlsignals.

Zustandsübersicht

Da es sich um einen rein datenfluss- und ereignisgesteuerten Netzwerk-Baustein (Composite FB) handelt, besitzt er keine interne Zustandsmaschine (ECC). Das Verhalten lässt sich statisch wie folgt beschreiben:

Zustand Selector `G`	Ereignis an Eingang	Verhalten am Ausgang `OUT`
`FALSE`	Beliebiges Ereignis	`OUT.D1` erhält den Wert von `IN0.D1`; `OUT.E1` wird ausgelöst.
`TRUE`	Beliebiges Ereignis	`OUT.D1` erhält den Wert von `IN1.D1`; `OUT.E1` wird ausgelöst.

Anwendungsszenarien

- **Hand-/Automatik-Umschaltung**: Auswahl zwischen einem manuell vorgegebenen Sollwert (z. B. über ein HMI an `IN1`) und einem automatisch berechneten Regler-Sollwert (an `IN0`).

- **Sensor-Redundanz**: Umschaltung auf einen Backup-Sensor (`IN1`), falls eine Diagnose-Logik einen Fehler beim Hauptsensor (`IN0`) erkennt und das Signal `G` auf `TRUE` setzt.
- **Rezeptursteuerung**: Dynamische Auswahl verschiedener vordefinierter Prozessparameter während des Betriebs.

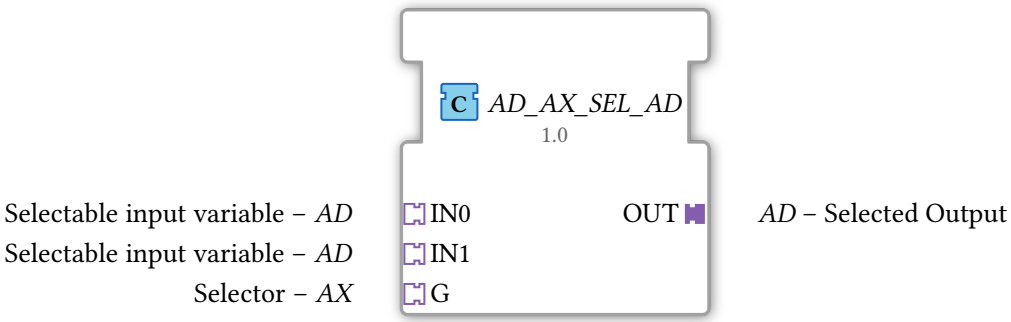
Vergleich mit ähnlichen Bausteinen

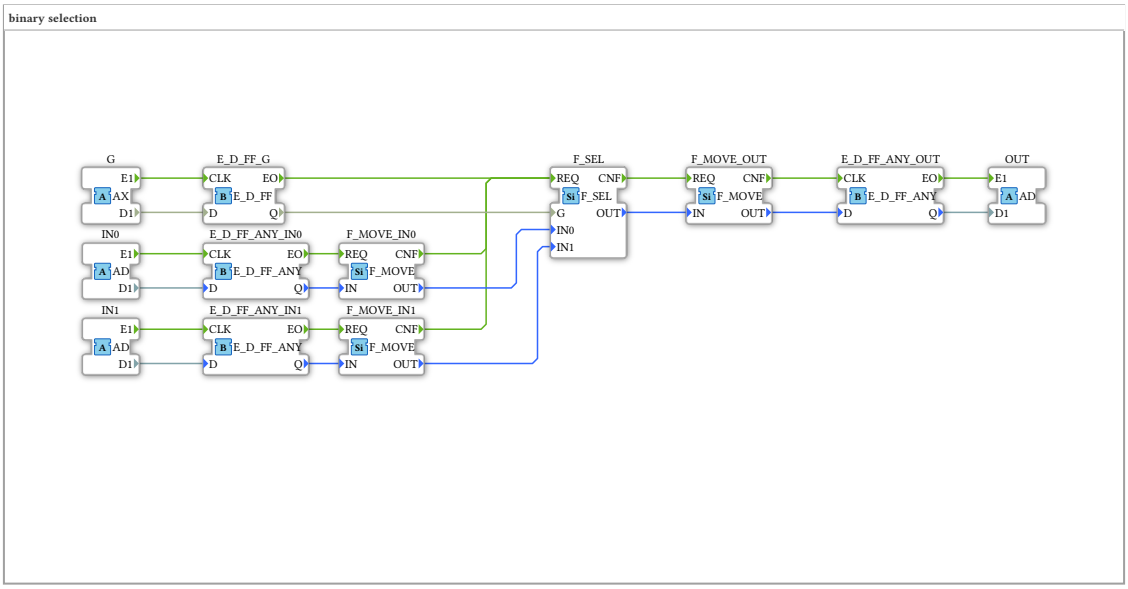
- **Standard `F_SEL`**: Der Standard-Auswahlbaustein arbeitet mit elementaren Datentypen und benötigt explizite Event-Verbindungen für Trigger und Confirmation. `ADI_AX_SEL_ADI` automatisiert diesen Prozess vollständig über strukturierte Adapter-Kanäle.
- **Analoge Multiplexer**: Im Gegensatz zu Multiplexern mit einer unbestimmten Anzahl an Kanälen ist dieser Baustein auf die schnelle und effiziente binäre Auswahl (1 aus 2) spezialisiert.

Fazit

Der `ADI_AX_SEL_ADI` ist ein äußerst nützlicher Hilfsbaustein für IEC 61499-Anwendungen in der 4diac IDE. Er verbindet die bewährte Selektionslogik der IEC 61131-3 mit den modernen, sauberen Design-Prinzipien der adapterbasierten ereignisorientierten Kommunikation.

AD_AX_SEL_AD





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein `AD_AX_SEL_AD` dient der binären Auswahl (Selektion) zwischen zwei Eingangssignalen, die über Adapterstrukturen bereitgestellt werden. Basierend auf dem Zustand eines Selektions-Signals (G) wird entweder der Wert des Eingangs `IN0` oder des Eingangs `IN1` an den Ausgang `OUT` weitergeleitet. Der Baustein ist intern als Composite-Funktionsbaustein realisiert und nutzt standardisier-

te IEC 61499- und IEC 61131-Mechanismen, um eine ereignisgesteuerte, typsichere Signalweiterleitung zu gewährleisten.

Schnittstellenstruktur

Die Schnittstellen dieses Funktionsbausteins sind vollständig über Adapter realisiert, was eine modulare und übersichtliche Verdrahtung im Anwendungsnetzwerk ermöglicht.

Ereignis-Eingänge

Dieser Baustein besitzt keine direkten Ereignis-Eingänge am Gehäuse. Die Ereignissteuerung erfolgt implizit über die Adapter.

Ereignis-Ausgänge

Dieser Baustein besitzt keine direkten Ereignis-Ausgänge am Gehäuse. Die Ereignisweiterleitung erfolgt implizit über die Adapter.

Daten-Eingänge

Dieser Baustein besitzt keine direkten Daten-Eingänge am Gehäuse.

Daten-Ausgänge

Dieser Baustein besitzt keine direkten Daten-Ausgänge am Gehäuse.

Adapter

- **Sockets (Eingangs-Adapter):**
  - `IN0` (Typ: `adapter::types::unidirectional::AD`): Erster auswählbarer Eingangskanal (wird gewählt, wenn `G` den Zustand `FALSE` hat).
  - `IN1` (Typ: `adapter::types::unidirectional::AD`): Zweiter auswählbarer Eingangskanal (wird gewählt, wenn `G` den Zustand `TRUE` hat).
  - `G` (Typ: `adapter::types::unidirectional::AX`): Selektor-Signal zur Steuerung der Auswahl.
- **Plugs (Ausgangs-Adapter):**
  - `OUT` (Typ: `adapter::types::unidirectional::AD`): Der ausgewählte Ausgangskanal, der den Wert des selektierten Eingangs führt.

Funktionsweise

Der Baustein kapselt ein internes Netzwerk aus Steuer- und Konvertierungsbausteinen:

1. **Signalpufferung und Erfassung:** Sobald sich an den Adaptern `IN0`, `IN1` oder `G` Daten ändern, triggert das entsprechende Adapter-Ereignis `E1` die internen Flip-Flops (`E_D_FF` bzw. `E_D_FF_ANY`).
2. **Datenweiterleitung:** Die Datenwerte (`D1`) der Eingänge `IN0` und `IN1` werden über interne Datenkonverter (`F_MOVE` vom Typ `DWORD`) eingelesen und dem Auswahlbaustein `F_SEL` zur Verfügung gestellt.
3. **Auswahllogik (`F_SEL`):**
   - Liegt am Selektor `G` das Signal `FALSE` an, schaltet der Baustein das Signal von `IN0` auf den Ausgang.
   - Liegt am Selektor `G` das Signal `TRUE` an, schaltet der Baustein das Signal von `IN1` auf den Ausgang.
4. **Ausgangssynchronisation:** Der ausgewählte Wert wird über einen weiteren `F_MOVE`-Baustein an ein Ausgangs-Flip-Flop (`E_D_FF_ANY_OUT`) übergeben. Dieses generiert das Ausgangsereignis `E1` am Ausgangsadapter `OUT` und legt den selektierten Wert an `D1` an.

Technische Besonderheiten

- **Adapter-Kapselung:** Durch die Verwendung von unidirektionalen Adaptern (`AD` und `AX`) wird das Signalrouting vereinfacht und lose Daten- und Ereignisleitungen vermieden.
- **Typunabhängigkeit über DWORD:** Die internen Datenpfade verwenden den Datentyp `DWORD` zur Repräsentation der Adapterdaten, was eine flexible Übertragung unterschiedlicher Datenformate ermöglicht.
- **Echtzeitnahe Reaktion:** Jede Flanke oder Wertänderung an den Eingangs- oder Selektor-Adaptern führt unverzüglich zu einer Aktualisierung des Ausgangs.

Zustandsübersicht

Der Baustein besitzt als Composite-FB kein eigenes Execution Control Chart (ECC). Das Verhalten ist rein daten- und ereignisgesteuert durch das interne Zusammenspiel der Standard-FBs:

- **Ereignis auf `IN0` oder `IN1`:** Aktualisiert den jeweiligen internen Wertpuffer. Falls der betroffene Eingang gerade aktiv geschaltet ist, wird der neue Wert sofort an `OUT` ausgegeben.
- **Ereignis auf `G`:** Wechselt den aktiven Kanal und gibt den Wert des neu gewählten Kanals an `OUT` aus.

Anwendungsszenarien

- **Sensorredundanz:** Umschalten zwischen einem primären Sensor (`IN0`) und einem redundanten Backup-Sensor (`IN1`) bei Signalverlust oder Fehlfunktion.
- **Sollwert-Umschaltung:** Wechseln zwischen Handbetrieb-Sollwert und Automatik-Sollwert in industriellen Prozessen.
- **Signal-Multiplexing:** Strukturierte Weiterleitung von Steuerungsdaten in modular aufgebauten Gesamtanlagen.

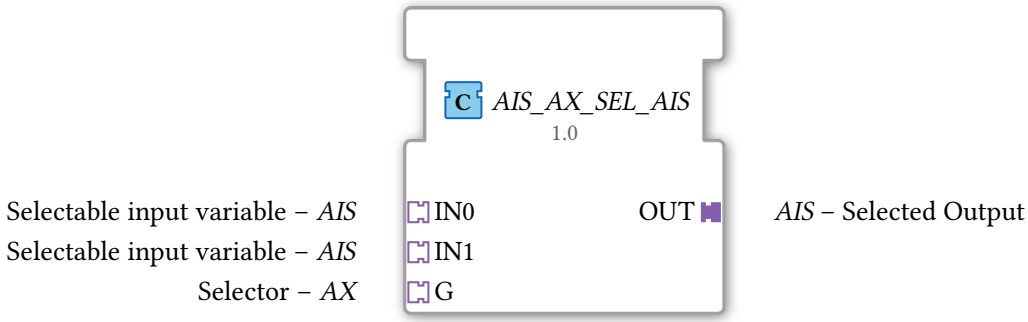
Vergleich mit ähnlichen Bausteinen

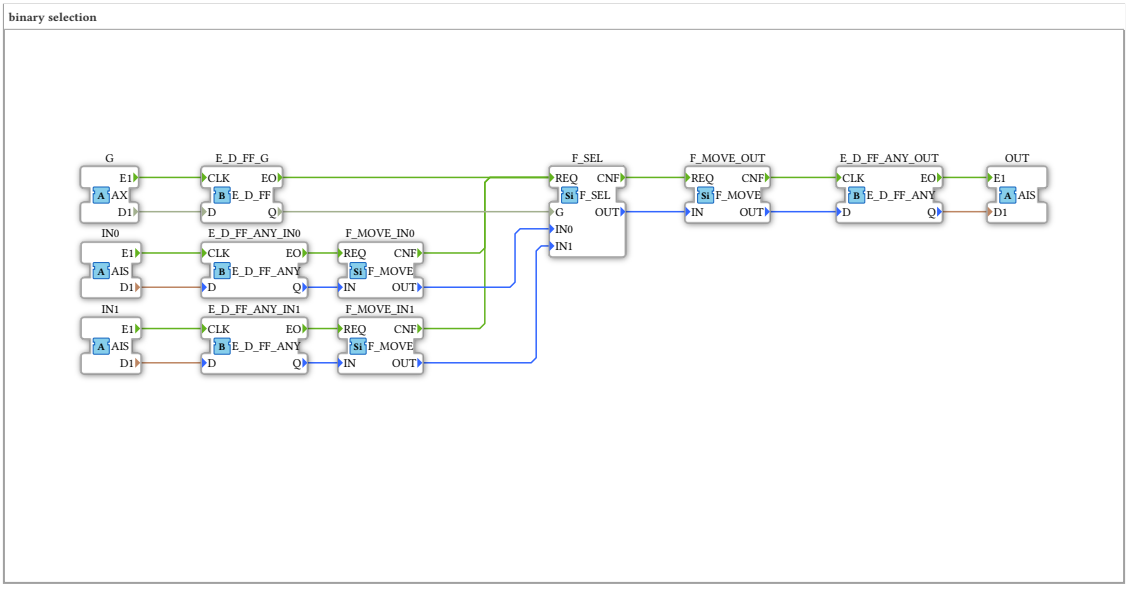
- **Standard `F_SEL`:** Der klassische `F_SEL`-Baustein nach IEC 61131-3 arbeitet direkt auf elementaren Datentypen und erfordert eine manuelle Verdrahtung von Trigger-Ereignissen. `AD_AX_SEL_AD` kapselt diese Logik und stellt sie direkt für adapterbasierte Architekturen bereit.
- **Multiplexer (MUX):** Ein klassischer MUX erlaubt die Auswahl aus mehr als zwei Kanälen mittels eines Integer-Werts. `AD_AX_SEL_AD` ist für einfache binäre Entscheidungen (Entweder-Oder) optimiert und spart dadurch Verarbeitungsressourcen.

Fazit

Der `AD_AX_SEL_AD` ist ein robuster und effizienter Hilfsbaustein zur ereignisgesteuerten Signalumschaltung. Durch die konsequente Nutzung von Adaptern fügt er sich ideal in moderne, serviceorientierte Steuerungsarchitekturen innerhalb von 4diac ein.

AIS_AX_SEL_AIS





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **AIS_AX_SEL_AIS** ist ein binärer Auswahlbaustein (Selection) für die IEC 61499 Architektur in 4diac-IDE. Er dient dazu, basierend auf dem Zustand eines Selektionssignals (Gate) dynamisch zwischen zwei unidirektionalen analogen Eingangssignalen auszuwählen und das gewählte Signal an den Ausgang weiterzuleiten.

Der Baustein nutzt intern die klassische Auswahl-Logik des standardisierten `SEL`-Bausteins (bekannt aus der IEC 61131-3), ist jedoch vollständig für die ereignisgesteuerte Ausführung der IEC 61499 optimiert und kapselt die Daten- und Ereignisströme mithilfe von standardisierten Adaptern.

Schnittstellenstruktur

Der Funktionsbaustein verwendet ein adapterbasiertes Schnittstellendesign. Er besitzt keine direkt herausgeführten ereignis- oder datenbasierten Ein- und Ausgänge auf der Hauptebene, sondern wickelt die gesamte Kommunikation über Plugs (Stecker) und Sockets (Buchsen) ab.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die Ereignisverarbeitung erfolgt gekapselt über die Adapter-Schnittstellen.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Ereignisausgabe erfolgt gekapselt über den Ausgangs-Adapter.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Schnittstellen)

- **G** (Typ: `adapter::types::unidirectional::AX`): Der Selector-Eingang (Gate). Bestimmt, welcher der beiden Eingänge (`IN0` oder `IN1`) an den Ausgang durchgeschaltet wird.
- **IN0** (Typ: `adapter::types::unidirectional::AIS`): Der erste auswählbare Eingangskanal. Dieser Kanal wird aktiv geschaltet, wenn der Selektor `G` den Zustand `FALSE` (0) aufweist.
- **IN1** (Typ: `adapter::types::unidirectional::AIS`): Der zweite auswählbare Eingangskanal. Dieser Kanal wird aktiv geschaltet, wenn der Selektor `G` den Zustand `TRUE` (1) aufweist.

Plugs (Ausgangs-Schnittstellen)

- **OUT** (Typ: `adapter::types::unidirectional::AIS`): Der ausgewählte Ausgangskanal. Er liefert den Wert des jeweils aktiven Eingangskanals sowie das zugehörige Aktualisierungsereignis.

Funktionsweise

Im Inneren des FBs arbeitet ein Netzwerk aus Standard-Funktionsbausteinen, welches die Adapter-Ereignisse synchronisiert und die logische Auswahl trifft:

1. **Ereignissynchronisation & Signalspeicherung:** Die an den Adaptern `IN0`, `IN1` und `G` ankommenden Daten werden bei jedem Eintreffen eines entsprechenden Adapter-Ereignisses (`E1`) in Daten-Flipflops (`E_D_FF_ANY` bzw. `E_D_FF`) zwischengespeichert. Dadurch wird sichergestellt, dass die Datenwerte für die Weiterverarbeitung stabil zur Verfügung stehen.
2. **Datenbereitstellung:** Die Bausteine `F_MOVE_IN0` und `F_MOVE_IN1` kopieren die zwischengespeicherten Daten (interpretiert als Datentyp `STRING`) und leiten sie an den eigentlichen Auswahlkern weiter.
3. **Auswahlprozess (F_SEL):** Der Kern-Baustein `F_SEL` (Typ `iec61131::selection::F_SEL`) wertet den Zustand des Selektors `G` aus:
   - Ist `G` = `FALSE`, wird der Wert von `IN0` gewählt.
   - Ist `G` = `TRUE`, wird der Wert von `IN1` gewählt.
4. **Ausgabeübertragung:** Das Auswahlergebnis wird über den Baustein `F_MOVE_OUT` an das Ausgangs-Flipflop `E_D_FF_ANY_OUT` übertragen. Dieses triggert das Ereignis `E1` am Ausgangs-Adapter `OUT` und legt den selektierten Datenwert an den Ausgang `D1` an.

Technische Besonderheiten

- **Adapter-Kapselung:** Durch die Verwendung von unidirektionalen Adaptern (`AIS` und `AX`) wird das Applikationsdiagramm in der 4diac-IDE übersichtlich gehalten, da Daten- und Ereignisleitungen in einer einzigen Verbindung gebündelt sind.
- **Asynchrone Ereignisbehandlung:** Jede Änderung an einem der Eingänge (`IN0`, `IN1`) oder am Selektor (`G`) stößt den Auswahlprozess automatisch neu an und aktualisiert den Ausgang wert- und ereignisgetreu.
- **Datentyp-Spezifizierung:** Die internen Datenkopierer (`F_MOVE`) sind fest auf den Datentyp `STRING` parametriert, was darauf hindeutet, dass die zu schaltenden Signale im `AIS`-Adapter als Zeichenketten übertragen werden.

Zustandsübersicht

Zustand des Selektors (`G`)	Letztes Ereignis an…	Aktiver Ausgang (`OUT.D1`)	Beschreibung
`FALSE`	`IN0.E1` oder `G.E1`	Wert von `IN0.D1`	Der Eingang 0 ist aktiv geschaltet. Änderungen an `IN1` haben keinen Einfluss auf den Ausgang.
`TRUE`	`IN1.E1` oder `G.E1`	Wert von `IN1.D1`	Der Eingang 1 ist aktiv geschaltet. Änderungen an `IN0` haben keinen Einfluss auf den Ausgang.

Anwendungsszenarien

- **Sensorredundanz / Failsafe-Systeme:** Umschalten zwischen einem Hauptsensor und einem Ersatzsensor bei Signalverlust oder Fehlfunktion.
- **Betriebsmodus-Auswahl:** Dynamische Weiterleitung unterschiedlicher Parametersätze (z.B. Automatik- vs. Handbetrieb-Sollwerte) an ein Stellglied.
- **Signalrouting in landwirtschaftlichen Maschinen (ISOBUS-Kontext):** Kanalsteuerung für analoge Prozesswerte oder Zustandsmeldungen in komplexen Steuerungsnetzwerken.

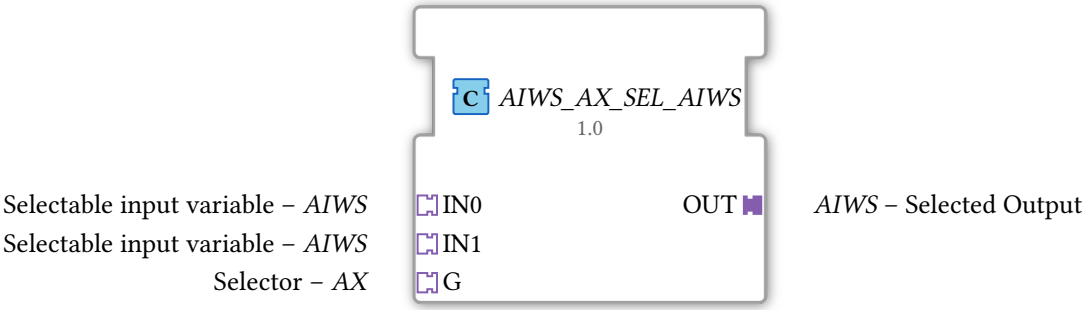
Vergleich mit ähnlichen Bausteinen

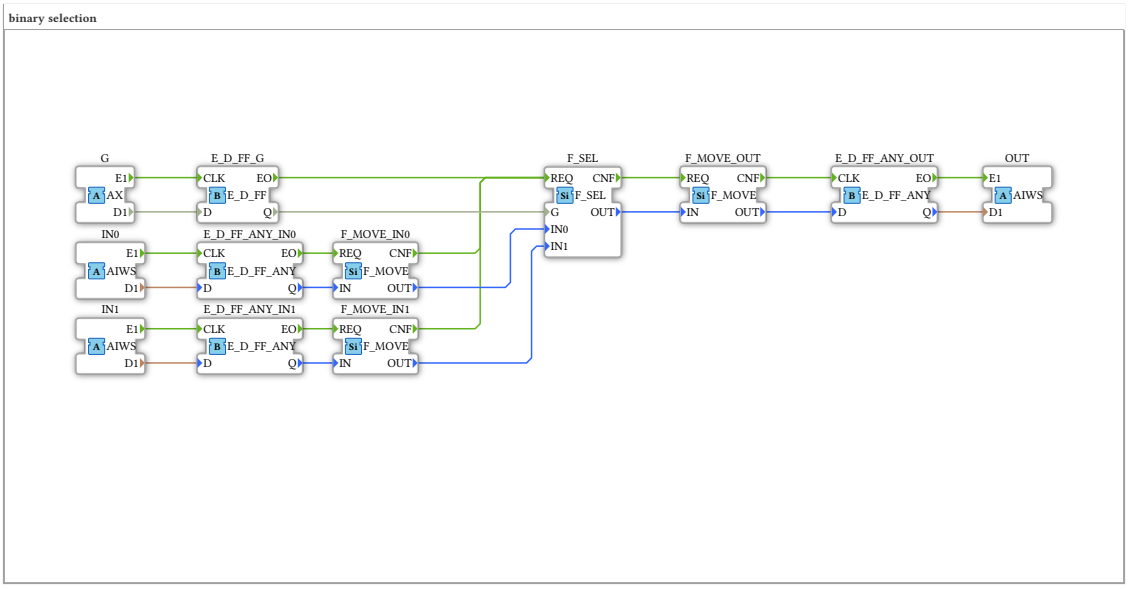
- **Klassischer `SEL` (IEC 61131-3):** Der klassische `SEL`-Baustein besitzt keine Ereignissteuerung und ist rein datenflussorientiert. `AIS_AX_SEL_AIS` erweitert dieses Prinzip um eine ereignisbasierte Steuerung für verteilte Systeme.
- **Standard-Auswahlbausteine ohne Adapter:** Typische IEC 61499 Auswahlbausteine arbeiten oft mit vielen einzelnen Event- und Daten-Pins. Dieser Baustein bietet durch die Adapter-Schnittstellen eine deutlich höhere Wartbarkeit und Modularität im Systemdesign.

Fazit

Der **AIS_AX_SEL_AIS** ist eine spezialisierte und performante Lösung für die Signalumschaltung in der 4diac-IDE. Durch die strikte Verwendung von Adaptern und die zuverlässige interne Ereignisverarbeitung eignet er sich ideal für saubere, modulare Softwarearchitekturen im Bereich der industriellen Automatisierung und Agrartechnik.

AIWS_AX_SEL_AIWS





(Kein Bild vorhanden)

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein `AIWS_AX_SEL_AIWS` dient der binären Auswahl (Selektion) zwischen zwei unidirektionalen Adaptersignalen vom Typ `AIWS`. Die Steuerung, welcher der beiden Eingänge auf den Ausgang geschaltet wird, erfolgt über einen Steueradapter vom Typ `AX`.

Der Baustein kapselt die klassische Selektionslogik (vergleichbar mit dem `SEL`-Standardbaustein aus der IEC 61131-3) in der ereignisgesteuerten Umgebung von IEC 61499. Dies ermöglicht eine saubere, modulare und übersichtliche Signalumschaltung direkt auf Adapterebene, ohne dass Ereignis- und Datenleitungen im übergeordneten System einzeln verdrahtet werden müssen.

Schnittstellenstruktur

Da es sich um einen adapterbasierten Funktionsbaustein handelt, besitzt der Baustein selbst keine direkten, klassischen Ereignis- oder Daten-Ein- und Ausgänge auf seiner äußeren Hülle. Die gesamte Kommunikation wird über die Adapter-Schnittstellen abgewickelt.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden (Ereignisse werden über die Adapter-Sockets empfangen).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden (Ereignisse werden über den Adapter-Plug gesendet).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Adapter)

- **G** (Typ: adapter::types::unidirectional::AX): Der Selektor-Adapter. Das darin enthaltene Datensignal bestimmt, welcher Eingang aktiv geschaltet wird.
- **IN0** (Typ: adapter::types::unidirectional::AIWS): Der erste auswählbare Eingangs-Adapter. Dieser wird an den Ausgang durchgereicht, wenn das Selektorsignal den Zustand FALSE (0) aufweist.
- **IN1** (Typ: adapter::types::unidirectional::AIWS): Der zweite auswählbare Eingangs-Adapter. Dieser wird an den Ausgang durchgereicht, wenn das Selektorsignal den Zustand TRUE (1) aufweist.

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: adapter::types::unidirectional::AIWS): Der Ausgangs-Adapter. Er gibt die Daten und Ereignisse des jeweils selektierten Eingangs (IN0 oder IN1) aus.

Funktionsweise

Das interne Netzwerk des Funktionsbausteins realisiert die Signalweiterleitung und Ereignissynchronisation wie folgt:

1. **Ereigniserfassung und Pufferung:** Sobald an einem der Sockets (G, IN0 oder IN1) das Ereignis E1 eintrifft, wird der dazugehörige Datenwert D1 über ein internes ereignisgesteuertes Flipflop (E_D_FF für den Selektor bzw. E_D_FF_ANY für die Daten) zwischengespeichert.
2. **Datenkonvertierung / Zuweisung:** Die Daten der Eingänge IN0 und IN1 werden über interne Zuweisungsglieder (F_MOVE) vom Typ WSTRING an den Auswahlbaustein übergeben.
3. **Auswahlprozess (F_SEL):** Der interne Baustein F_SEL (IEC 61131-3 SEL) wertet den Zustand des Selektors aus dem Adapter G aus:
   - Ist das Signal aus G im Zustand FALSE, wird das Signal von IN0 an den Ausgang weitergeleitet.
   - Ist das Signal aus G im Zustand TRUE, wird das Signal von IN1 an den Ausgang weitergeleitet.
4. **Ausgabe:** Das ausgewählte Signal wird über einen weiteren F_MOVE-Baustein und ein Ausgangs-Flipflop (E_D_FF_ANY_OUT) an den Datenpunkt D1 des Ausgangs-Plugs OUT übergeben. Gleichzeitig wird am Ausgang das Ereignis E1 ausgelöst, um nachfolgende Bausteine über die Datenaktualisierung zu informieren.

Technische Besonderheiten

- **Datentyp WSTRING:** Die interne Verarbeitung der Adapterdaten erfolgt über F_MOVE-Bausteine mit dem Datentyp WSTRING. Dies bedeutet, dass die Nutzdaten innerhalb der AIWS-Adapter als Zeichenketten (Wide Strings) übertragen und umgeschaltet werden.
- **Ereignisentkopplung:** Durch die Verwendung von Flipflops (E_D_FF) wird sichergestellt, dass jede Änderung an den Eingängen oder am Selektor eine sofortige und konsistente Aktualisierung des Ausgangs zur Folge hat.

Zustandsübersicht

Die logische Zuordnung des Ausgangs verhält sich in Abhängigkeit vom Selektor G wie folgt:

Zustand von G (Selektor)	Aktiver Pfad	Datenwert an OUT.D1	Ereignis an OUT.E1
FALSE	IN0	Wert von IN0.D1	Wird bei Ereignissen an IN0.E1 oder G.E1 ausgelöst
TRUE	IN1	Wert von IN1.D1	Wird bei Ereignissen an IN1.E1 oder G.E1 ausgelöst

Anwendungsszenarien

- **Umschaltung von Analogwerten mit Status (formatiert als WSTRING):** Redundante Sensorik, bei der im Fehlerfall von Sensor 1 (IN0) auf einen Ersatzsensor 2 (IN1) umgeschaltet werden soll.
- **Dynamische Text- oder Rezepturauswahl:** Umschaltung von Konfigurations-Strings, Logmeldungen oder Sollwerten, die über Adapterstrukturen an ein HMI oder eine Steuereinheit geleitet werden.

- **Signal-Routing:** Flexibles Routing in modular aufgebauten Industrieanlagen, bei denen Datenströme je nach Betriebsmodus (z. B. Automatik vs. Handbetrieb) umgeleitet werden müssen.

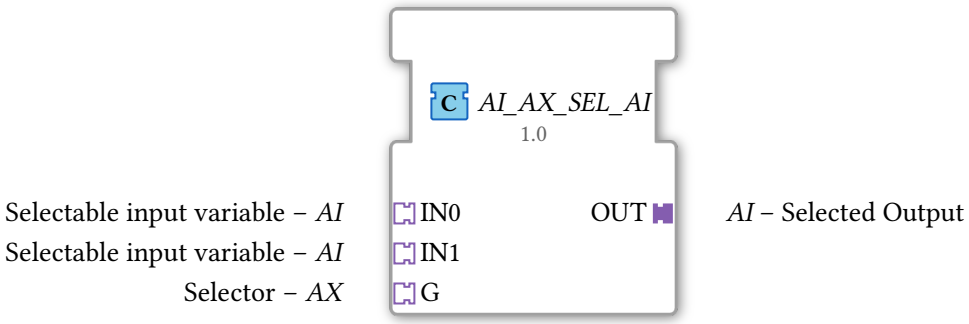
Vergleich mit ähnlichen Bausteinen

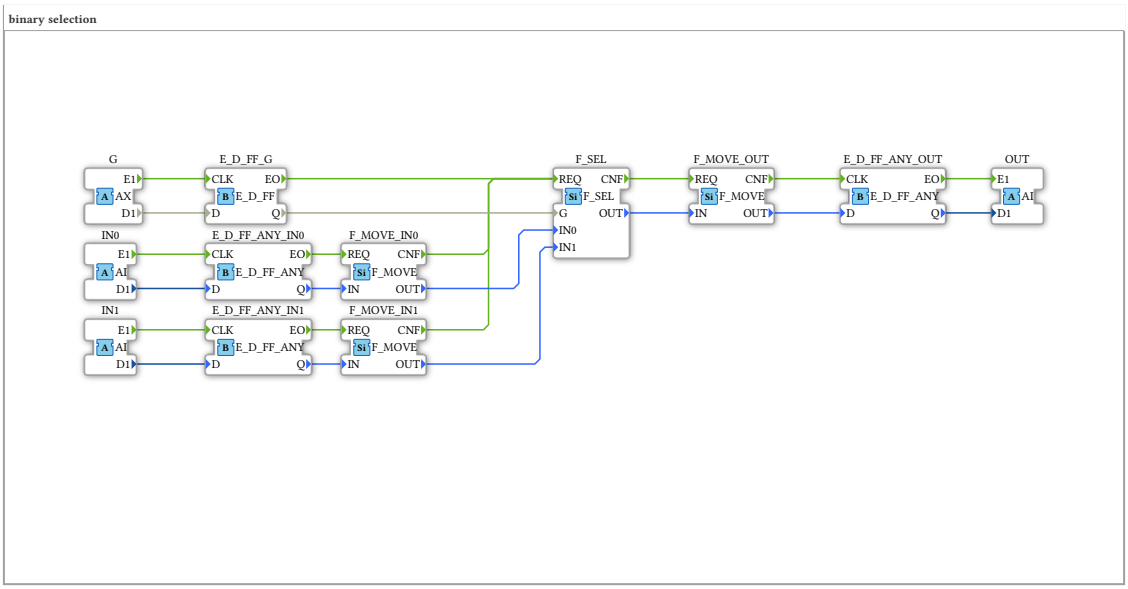
- **Standard `SEL` (IEC 61131-3):** Der klassische `SEL`-Baustein arbeitet auf elementaren Datentypen (z.B. `INT`, `REAL`, `STRING`) und besitzt keine native Ereignissteuerung oder Adapterunterstützung. `AIWS_AX_SEL_AIWS` erweitert dieses Prinzip für die IEC 61499, indem er die Umschaltung direkt für komplexe Adaptertypen bereitstellt.
- **Multiplexer (`MUX`):** Ein klassischer Multiplexer erlaubt die Auswahl aus mehr als zwei Kanälen über einen Integer-Index. Der `AIWS_AX_SEL_AIWS` ist für die schnelle und ressourcenschonende binäre Auswahl (2 Kanäle) optimiert.

Fazit

Der `AIWS_AX_SEL_AIWS` ist ein hochspezialisierter Hilfsbaustein für IEC 61499-Systeme (wie 4diac-ide), der komplexe Daten- und Ereignisströme zweier Adapter elegant und deterministisch umschaltet. Er trägt maßgeblich zur Reduzierung von „Spaghetti-Code“ und unübersichtlicher Verdrahtung in grafischen Programmierumgebungen bei.

AI_AX_SEL_AI





(Kein Bild vorhanden)

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein `AI_AX_SEL_AI` ist ein binärer Selektor (Multiplexer) für analoge Signale, der auf der IEC 61499-Architektur basiert. Er dient dazu, basierend auf einem Steuersignal (Selektor) eines von zwei analogen Eingangssignalen auszuwählen und an den Ausgang weiterzuleiten. Die Besonderheit dieses Bausteins liegt in der konsequenten Nutzung von Adaptern, was die Verkabelung innerhalb der Entwicklungsumgebung 4diac-ide erheblich vereinfacht und strukturiert.

Schnittstellenstruktur

Der Funktionsbaustein besitzt keine klassischen diskreten Ein- und Ausgänge für Daten und Ereignisse auf seiner Hauptebene. Stattdessen ist die gesamte Kommunikation über standardisierte Adapter realisiert.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden (vollständig über Adapter gekapselt).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden (vollständig über Adapter gekapselt).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden (vollständig über Adapter gekapselt).

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden (vollständig über Adapter gekapselt).

Adapter

Sockets (Eingangs-Schnittstellen)

- **G** (Typ: `adapter::types::unidirectional::AX`): Der Selektor-Eingang. Dieser Adapter liefert das Steuersignal (boolescher Zustand), das bestimmt, welcher der beiden analogen Eingänge auf den Ausgang geschaltet wird.
- **IN0** (Typ: `adapter::types::unidirectional::AI`): Der auswählbare Analogeingang 0. Dieser Wert wird durchgeschaltet, wenn das Selektorsignal `G` den Zustand `FALSE` (0) aufweist.
- **IN1** (Typ: `adapter::types::unidirectional::AI`): Der auswählbare Analogeingang 1. Dieser Wert wird durchgeschaltet, wenn das Selektorsignal `G` den Zustand `TRUE` (1) aufweist.

Plugs (Ausgangs-Schnittstellen)

- **OUT** (Typ: `adapter::types::unidirectional::AI`): Der ausgewählte Analogausgang. Er gibt das Signal des jeweils aktiven Eingangs (`IN0` oder `IN1`) aus.

Funktionsweise

Die interne Logik des Bausteins steuert den Daten- und Ereignisfluss wie folgt:

1. **Ereignissteuerung & Datenerfassung:**
   - Jedes Mal, wenn an einem der Eingangs-Adapter (`IN0`, `IN1` oder `G`) ein Ereignis (`E1`) eintrifft, wird das zugehörige Datenelement (`D1`) über interne flanken- bzw. ereignisgesteuerte Speicherbausteine (`E_D_FF` und `E_D_FF_ANY`) zwischengespeichert.
   - Interne `F_MOVE`-Bausteine konvertieren und übergeben die Werte (als Datentyp `INT`) an den zentralen Auswahlbaustein.
2. **Auswahllogik (Selection):**
   - Der Kernbaustein `F_SEL` (ein standardisierter IEC 61131-3 Auswahl-Funktionsbaustein) wertet das Steuersignal `G` aus.
   - Ist das Steuersignal `G` auf `FALSE`, wird der Wert von `IN0` gewählt.
   - Ist das Steuersignal `G` auf `TRUE`, wird der Wert von `IN1` gewählt.
3. **Ausgabe:**
   - Nach erfolgreicher Auswahl wird der ausgewählte Wert über einen weiteren `F_MOVE`-Baustein und ein Ausgangs-Flip-Flop (`E_D_FF_ANY_OUT`) an den Ausgangs-Plug `OUT` übergeben.
   - Gleichzeitig wird das Ausgangsereignis `OUT.E1` ausgelöst, um nachfolgende Bausteine über die Wertänderung zu informieren.

Technische Besonderheiten

- **Datentyp-Spezifizierung:** Intern arbeitet der Baustein mit dem Datentyp `INT` (Ganzzahl) für die analogen Werte, was durch die Parametrierung der internen `F_MOVE`-Instanzen festgelegt ist.

- **Adapterbasierte Architektur:** Durch die Kapselung von Ereignis und Daten in den Adaptern `AI` (Analog Input) und `AX` (Auxiliary Control) wird die Anzahl der Verbindungsleitungen im übergeordneten Systemdiagramm drastisch reduziert.
- **Event-Driven Propagation:** Jede Aktualisierung eines Eingangs (unabhängig davon, ob es sich um die Daten oder den Selektor handelt) stößt die Neuberechnung und die Ausgabe eines Ereignisses am Ausgang an.

Zustandsübersicht

Der Baustein verhält sich rein ereignis- und datenflussgesteuert. Es gibt keine komplexen internen Zustände (keine Zustandsmaschine / ECC im klassischen Sinne), sondern eine direkte Abhängigkeit des Ausgangs von den Eingängen:

Selektor-Eingang (`G.D1`)	Aktiver Pfad	Ausgangswert (`OUT.D1`)	Ausgangsereignis (`OUT.E1`)
`FALSE`	`IN0`	Wert von `IN0.D1`	Wird bei jedem Trigger ausgelöst
`TRUE`	`IN1`	Wert von `IN1.D1`	Wird bei jedem Trigger ausgelöst

Anwendungsszenarien

- **Redundante Sensorik:** Umschalten zwischen einem primären und einem sekundären (Backup-) Analogsensor bei einem detektierten Ausfall.
- **Hand-/Automatik-Umschaltung:** Auswahl zwischen einem manuell vorgegebenen Sollwert (z.B. über ein HMI an `IN0`) und einem automatisiert berechneten Regler-Sollwert (an `IN1`).
- **Rezepturgesteuerte Signalumleitung:** Dynamische Zuweisung von analogen Prozessparametern basierend auf dem aktuellen Betriebszustand einer Anlage.

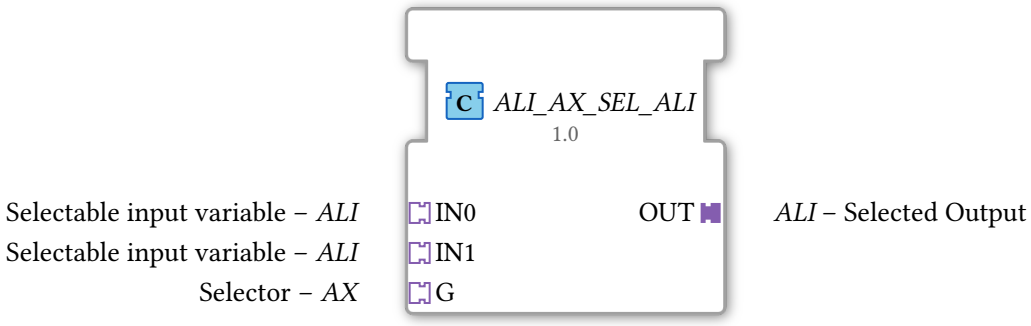
Vergleich mit ähnlichen Bausteinen

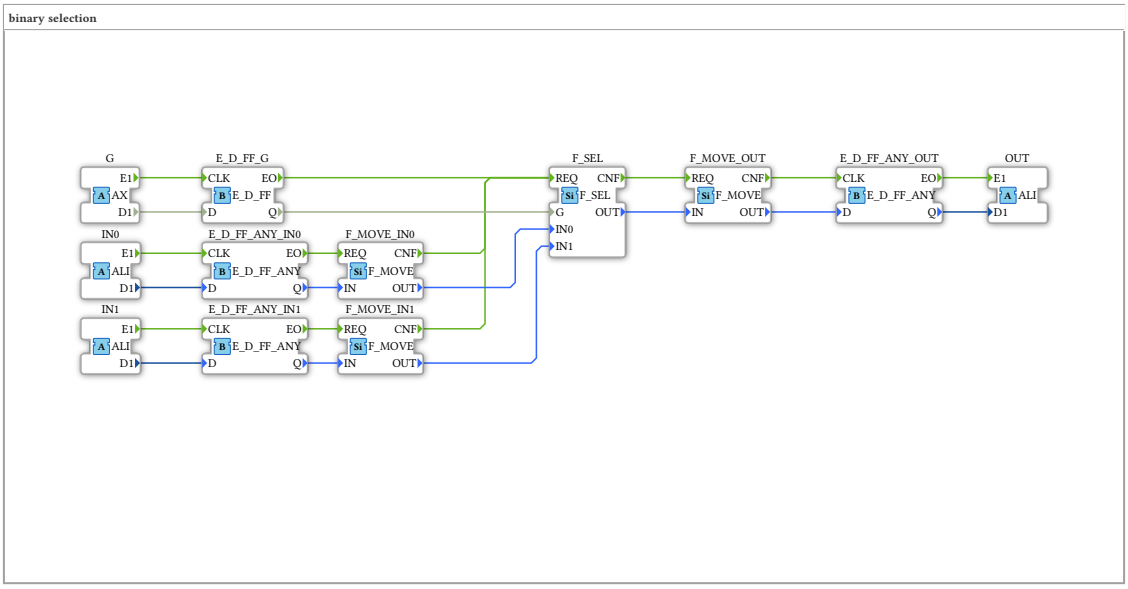
- **Standard `F_SEL` (IEC 61131-3):** Der klassische `F_SEL`-Baustein benötigt diskrete Pins für Ereignisse und Daten. `AI_AX_SEL_AI` kapselt diese Funktionalität in anwendungsfreundliche Adapterstrukturen, was die Wiederverwendbarkeit und Übersichtlichkeit in komplexen IEC 61499-Anwendungen erhöht.
- **Klassische Multiplexer (z.B. MUX):** Während ein Multiplexer oft mehrere Kanäle (z.B. 4 oder 8) bedient, ist dieser Baustein als schlanker, hocheffizienter 2-Wege-Auswahlschalter für dedizierte Analogsignale optimiert.

Fazit

Der `AI_AX_SEL_AI`-Baustein stellt eine elegante, adapterbasierte Lösung für die binäre Auswahl von Analogsignalen in verteilten Steuerungssystemen dar. Durch die saubere Trennung und Kapselung der Schnittstellen in standardisierte Adapter eignet er sich hervorragend für den Einsatz in modularen und serviceorientierten Automatisierungsarchitekturen innerhalb von Eclipse 4diac.

ALI_AX_SEL_ALI





[Bild des Funktionsblocks]

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `ALI_AX_SEL_ALI` ist ein binärer Auswahlbaustein (Selection Block), der für die Verwendung mit unidirektionalen Adaptern konzipiert ist. Er ermöglicht es, basierend auf einem Auswahlsignal (Selector), zwischen zwei Eingangssignalen zu wählen und das ausgewählte Signal an einen Ausgang weiterzuleiten. Dieser Baustein kapselt die Standard-Auswahllogik und passt sie an eine ereignisgesteuerte Adapter-Schnittstelle an.

Schnittstellenstruktur

Da es sich um einen adapterbasierten Funktionsblock handelt, besitzt er keine direkten, klassischen Ereignis- oder Dateneingänge auf der obersten Ebene. Die gesamte Kommunikation wird über Adapter abgewickelt.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden (Steuerung erfolgt über die Adapter).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden (Steuerung erfolgt über die Adapter).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingang-Schnittstellen)

- **G** (`adapter::types::unidirectional::AX`): Der Selektor-Adapter (Gate). Er steuert, welcher der beiden Eingänge auf den Ausgang durchgeschaltet wird.
- **IN0** (`adapter::types::unidirectional::ALI`): Der erste auswählbare Eingangs-Adapter (wird standardmäßig bei `G = FALSE` gewählt).
- **IN1** (`adapter::types::unidirectional::ALI`): Der zweite auswählbare Eingangs-Adapter (wird bei `G = TRUE` gewählt).

Plugs (Ausgang-Schnittstellen)

- **OUT** (`adapter::types::unidirectional::ALI`): Der Ausgangs-Adapter, der das ausgewählte Signal und das dazugehörige Ereignis bereitstellt.

Funktionsweise

Der Funktionsblock arbeitet als Multiplexer für die Adapter-Signale:

1. **Ereignis-Erfassung**: Sobald an einem der Eingangs-Adapter (`IN0`, `IN1`) oder dem Selektor-Adapter (`G`) ein Ereignis (`E1`) eingeht, wird dieses über interne flankengetriggerte Flip-Flops (`E_D_FF` bzw. `E_D_FF_ANY`) abgefangen.
2. **Datenpufferung**: Die Daten der Eingänge (`IN0.D1` und `IN1.D1`) werden über interne `F_MOVE`-Bausteine (konfiguriert auf den Datentyp `LINT`) an den zentralen Auswahlbaustein `F_SEL` weitergegeben.
3. **Auswahl**: Der Baustein `F_SEL` wertet den Zustand des Selektors `G.D1` aus:
   - Ist `G.D1` auf `FALSE`, wird der Wert von `IN0` ausgewählt.
   - Ist `G.D1` auf `TRUE`, wird der Wert von `IN1` ausgewählt.
4. **Ausgabe**: Der ausgewählte Wert wird an das Flip-Flop `E_D_FF_ANY_OUT` übergeben. Dieses aktualisiert den Ausgangs-Datenpunkt `OUT.D1` und triggert gleichzeitig das Ausgangsereignis `OUT.E1`.

Technische Besonderheiten

- **Datentyp-Spezifität**: Die internen Verschiebungs-Bausteine (`F_MOVE`) sind fest auf den Datentyp `LINT` (Long Integer) eingestellt. Das bedeutet, dass die über die `ALI`-Adapter übertragenen Daten als `LINT` interpretiert und verarbeitet werden.
- **Ereignisentkopplung**: Durch die Verwendung von Flip-Flops (`E_D_FF`) wird sichergestellt, dass Datenänderungen an den Eingängen konsistent und synchronisiert mit den entsprechenden Ereignissen am Ausgang ankommen.

Zustandsübersicht

Der Funktionsblock besitzt keine eigene Zustandshierarchie (kein ECC), da er als zusammengesetzter Netzwerk-Funktionsblock (Composite FB) realisiert ist. Sein Verhalten ist rein daten- und ereignisflussgesteuert:

Zustand Selektor (`G.D1`)	Trigger-Ereignis	Aktion / Ergebnis
`FALSE`	Ereignis an `IN0.E1`, `IN1.E1` oder `G.E1`	Wert von `IN0.D1` wird an `OUT.D1` übergeben; `OUT.E1` wird ausgelöst.
`TRUE`	Ereignis an `IN0.E1`, `IN1.E1` oder `G.E1`	Wert von `IN1.D1` wird an `OUT.D1` übergeben; `OUT.E1` wird ausgelöst.

Anwendungsszenarien

- **Sollwert-Umschaltung**: Auswahl zwischen einem automatischen (z.B. von einem Regler berechneten) Sollwert und einem manuell vorgegebenen Sollwert in einer industriellen Anlage.
- **Sensor-Redundanz**: Umschaltung zwischen zwei redundanten Sensoren (z.B. bei Ausfall oder zur Kalibrierung), die ihre Daten über standardisierte Adapter-Schnittstellen bereitstellen.
- **Rezepturgesteuerte Pfadauswahl**: Dynamische Auswahl von Prozessparametern basierend auf einem binären Steuersignal im Ablaufprogramm.

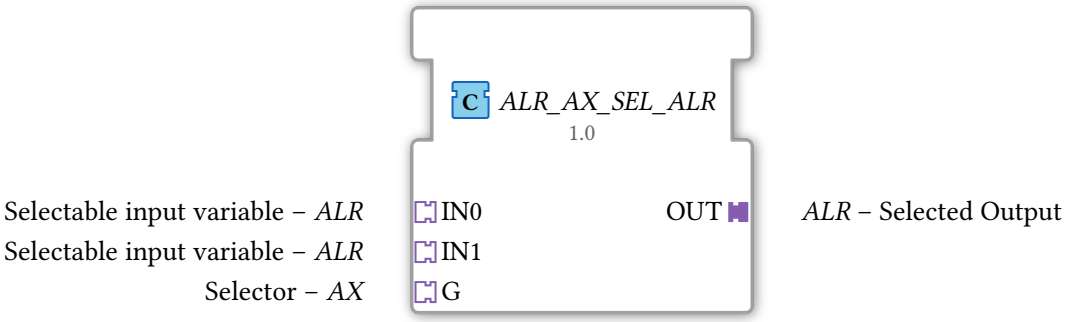
Vergleich mit ähnlichen Bausteinen

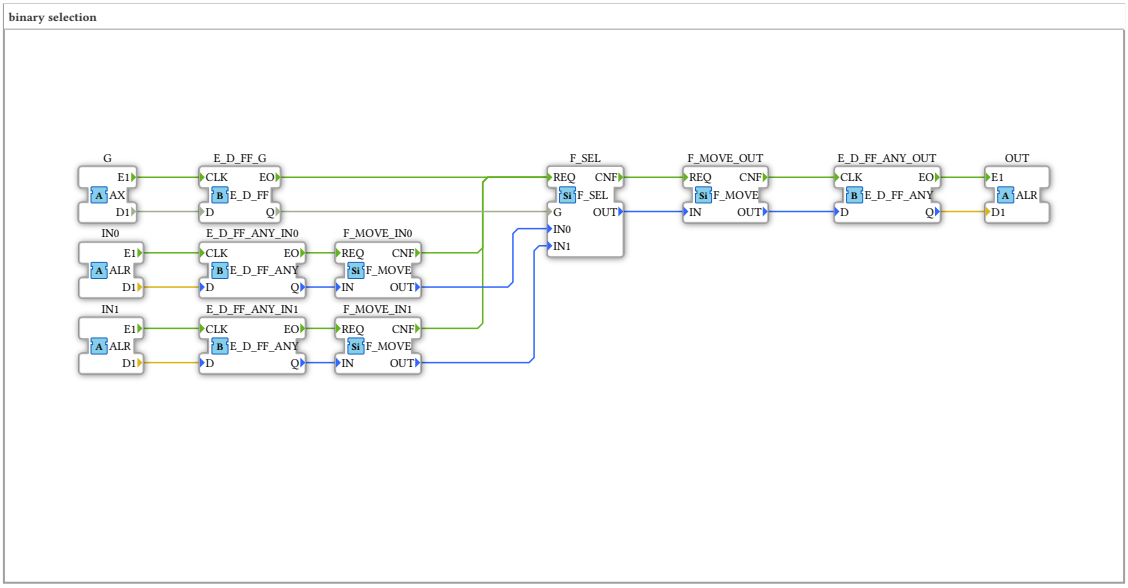
- **F_SEL (Standard IEC 61131-3)**: Der standardmäßige Auswahleffekt-Block arbeitet direkt mit elementaren Datentypen und benötigt explizite Event-Verbindungen. `ALI_AX_SEL_ALI` kapselt diesen Baustein und erweitert ihn um die direkte Unterstützung für die strukturiertere und sauberere Verwendung von unidirektionalen Adaptern (`ALI`/`AX`).
- **Klassische Multiplexer (MUX)**: Während ein MUX oft mehrere Kanäle (z.B. 4, 8) bedienen kann, ist dieser Baustein strikt auf eine binäre Auswahl (2 Kanäle) optimiert, bietet dafür jedoch eine schlanke Adapter-Integration.

Fazit

Der `ALI_AX_SEL_ALI` ist ein spezialisierter Hilfsbaustein, der die Signal- und Ereignisweiterleitung in komplexen 4diac-Systemen vereinfacht. Durch die Kapselung der Adapterlogik trägt er maßgeblich zur Übersichtlichkeit im Applikations-Netzwerk bei, indem er das manuelle Entpacken und Neuverpacken von Adapter-Signalen überflüssig macht.

ALR_AX_SEL_ALR





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ALR_AX_SEL_ALR` dient zur binären Auswahl (Selektion) zwischen zwei analogen Eingangssignalen, die über Adapter des Typs `ALR` bereitgestellt werden. Gesteuert wird diese Auswahl durch einen Selektor-Adapter des Typs `AX`. Das ausgewählte Signal wird an einen Ausgangs-Adapter des Typs `ALR` weitergegeben.

Der Baustein basiert intern auf der IEC 61131-3 Auswahlfunktion `F_SEL` und ist für ereignisgesteuerte IEC 61499-Architekturen optimiert.

Schnittstellenstruktur

Da dieser Baustein als Adapter-Koppler aufgebaut ist, besitzt er keine klassischen, direkten Ereignis- oder Daten-Schnittstellen auf der obersten Ebene. Die gesamte Kommunikation wird über Sockets (Eingangs-Adapter) und Plugs (Ausgangs-Adapter) abgewickelt.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

- **Sockets (Eingangs-Schnittstellen):**
  - `IN0` (Typ: `adapter::types::unidirectional::ALR`): Erster auswählbarer Eingangskanal (wird auf den Ausgang geschaltet, wenn der Selektor `G` den Wert `FALSE` aufweist).
  - `IN1` (Typ: `adapter::types::unidirectional::ALR`): Zweiter auswählbarer Eingangskanal (wird auf den Ausgang geschaltet, wenn der Selektor `G` den Wert `TRUE` aufweist).
  - `G` (Typ: `adapter::types::unidirectional::AX`): Selektor-Eingang zur Steuerung der Kanalauswahl.
- **Plugs (Ausgangs-Schnittstellen):**
  - `OUT` (Typ: `adapter::types::unidirectional::ALR`): Ausgang, der das ausgewählte Signal führt.

Funktionsweise

Der Baustein realisiert eine softwaregesteuerte 1-aus-2-Auswahl auf Basis von Adaptern:

1. **Ereignispufferung:** Eintreffende Ereignisse und Daten an den Sockets `IN0`, `IN1` und `G` werden über interne Ereignis-D-Flip-Flops (`E_D_FF` bzw. `E_D_FF_ANY`) zwischengespeichert, um eine konsistente Datenübergabe zu gewährleisten.
2. **Wertübertragung:** Die Datenwerte werden über interne Zuweisungsglieder (`F_MOVE`) für den Datentyp `LREAL` aufbereitet.
3. **Auswahllogik (`F_SEL`):** Die Auswahl erfolgt über den Standard-Selektionsbaustein:
   - Ist das Signal an `G.D1` gleich `FALSE`, wird der Wert von `IN0.D1` an den Ausgang weitergegeben.
   - Ist das Signal an `G.D1` gleich `TRUE`, wird der Wert von `IN1.D1` an den Ausgang weitergegeben.
4. **Ausgabe-Triggerung:** Jede Änderung an einem der Eingänge oder am Selektor führt zu einer Aktualisierung des Ausgangs-Adapters `OUT`, signalisiert durch das Ereignis `OUT.E1` und den neuen Datenwert `OUT.D1`.

Technische Besonderheiten

- **Präzise Datenverarbeitung (LREAL):** Die internen Datenpfade sind für den Datentyp `LREAL` ausgelegt, was eine verlustfreie Übertragung hochgenauer analoger Messwerte ermöglicht.
- **Kapselung durch Adapter:** Durch die Verwendung von standardisierten Adaptern (`ALR`, `AX`) entfällt die komplexe Einzelverdrahtung von Daten- und Ereignis-Pins im Applikationseditor.
- **Ereignisgesteuert:** Der Ausgang reagiert dynamisch und verzögerungsfrei auf Wertänderungen und Trigger an den Eingängen.

Zustandsübersicht

Das Verhalten des Bausteins wird durch den Daten- und Ereignisfluss im internen Netzwerk bestimmt:

Zustand Selektor `G.D1`	Durchgeschalteter Kanal	Ausgangswert `OUT.D1`	Ausgangsereignis `OUT.E1`
`FALSE`	Kanal 0 (`IN0`)	Wert von `IN0.D1`	Triggert bei Ereignissen auf `IN0.E1` oder Änderung von `G`
`TRUE`	Kanal 1 (`IN1`)	Wert von `IN1.D1`	Triggert bei Ereignissen auf `IN1.E1` oder Änderung von `G`

Anwendungsszenarien

- **Sensor-Redundanz / Failover:** Umschaltung zwischen einem Hauptsensor (`IN0`) und einem Backup-Sensor (`IN1`) bei Ausfall oder Signalstörung, gesteuert durch ein Diagnosesignal an `G`.
- **Sollwertumschaltung:** Wechsel zwischen einem Automatik-Sollwert und einem manuellen Vorgabewert in verfahrenstechnischen Regelkreisen.
- **Rezeptur- und Parametersatz-Auswahl:** Dynamische Zuweisung unterschiedlicher analoger Prozessparameter basierend auf dem aktuellen Anlagenzustand.

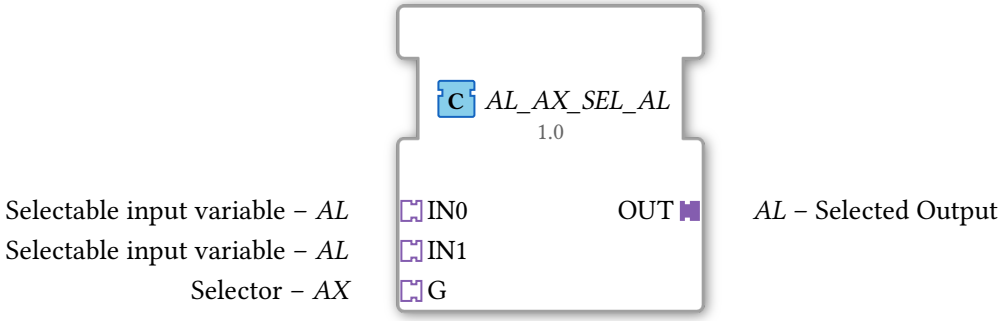
Vergleich mit ähnlichen Bausteinen

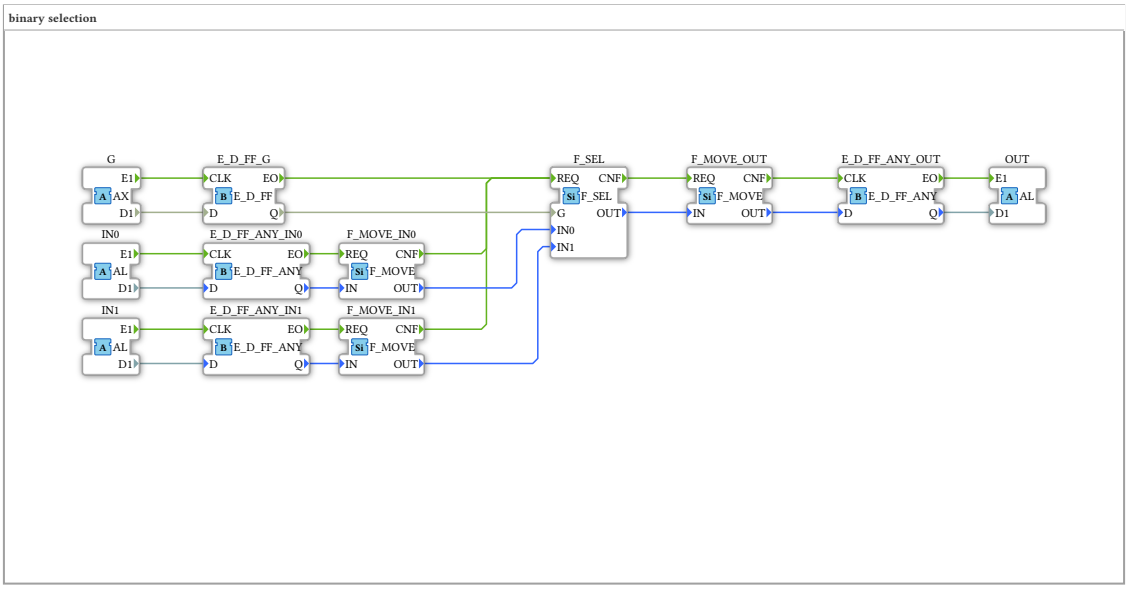
Im Vergleich zu einem klassischen IEC 61131-3 SEL-Baustein kapselt ALR_AX_SEL_ALR die Signalwege komplett in Adapterstrukturen. Dies reduziert die visuelle Komplexität in 4diac-Applikationsdiagrammen erheblich, da statt mehrerer einzelner Verbindungen nur noch strukturierte Adapterlinien gezogen werden müssen.

Fazit

Der ALR_AX_SEL_ALR ist ein robuster und wiederverwendbarer Baustein für die strukturierte Signalumschaltung in der IEC 61499. Durch die konsequente Nutzung von Adaptern und die Unterstützung von hochauflösenden LREAL-Daten eignet er sich ideal für moderne Steuerungs- und Regelungskonzepte in der industriellen Automatisierung.

AL_AX_SEL_AL





(Kein Bild vorhanden)

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein AL_AX_SEL_AL ist ein ereignisgesteuerter Selektor (Binäre Auswahl) für IEC 61499-Systeme in der 4diac-IDE. Er dient als Multiplexer, der basierend auf dem Zustand eines Auswahlsignals (Selector) Daten von einem von zwei Eingangs-Adaptern an einen Ausgangs-Adapter weiterleitet. Der Baustein nutzt konsequent die Vorteile von Adaptern, um Daten- und Ereignisströme sauber zu kapseln und die Komplexität der Verkabelung im Anwendungsdiagramm zu minimieren.

Schnittstellenstruktur

Da dieser Funktionsbaustein intern als Netzwerk aufgebaut ist und vollständig auf Adapter-Schnittstellen basiert, besitzt er keine direkt herausgeführten klassischen Ein- oder Ausgangsvariablen.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung erfolgt implizit über die angeschlossenen Adapter.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Ereignisweiterleitung erfolgt implizit über den Ausgangs-Adapter.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Plugs (Stecker)

- **`OUT`** (Typ: `adapter::types::unidirectional::AL`): Der ausgewählte Ausgang. Über diesen Adapter werden die selektierten Daten (`D1`) und das dazugehörige Aktualisierungsereignis (`E1`) ausgegeben.

Sockets (Buchsen)

- **`G`** (Typ: `adapter::types::unidirectional::AX`): Der Selektor-Eingang (Gate/Selector). Steuert, welcher der beiden Eingänge (`IN0` oder `IN1`) auf den Ausgang durchgeschaltet wird.
- **`IN0`** (Typ: `adapter::types::unidirectional::AL`): Der erste auswählbare Eingangskanal. Dieser ist aktiv und wird an den Ausgang weitergeleitet, wenn der Selektor `G` den Zustand `FALSE` (bzw. logisch 0) aufweist.
- **`IN1`** (Typ: `adapter::types::unidirectional::AL`): Der zweite auswählbare Eingangskanal. Dieser ist aktiv und wird an den Ausgang weitergeleitet, wenn der Selektor `G` den Zustand `TRUE` (bzw. logisch 1) aufweist.

Funktionsweise

Der Baustein realisiert eine klassische Multiplexer-Logik (Äquivalent zu `SEL` in IEC 61131-3) im IEC 61499-Kontext:

1. **Ereignisverarbeitung & Pufferung:** Trifft ein Ereignis auf einem der Eingangs-Adapter (`IN0`, `IN1`) oder dem Selektor-Adapter (`G`) ein, fangen interne D-Flipflops (`E_D_FF` bzw. `E_D_FF_ANY`) die Datenwerte ab und speichern diese zwischen.
2. **Daten-Typkonvertierung:** Die gepufferten Datenwerte werden über interne Transfer-Bausteine (`F_MOVE` mit dem Datentyp `LWORD`) an den zentralen Auswahlbaustein weitergeleitet.
3. **Auswahllogik (`F_SEL`):** Das Kernstück des Netzwerks wertet den Zustand des Selektors `G` aus:
   - Ist `G` gleich `FALSE`, wird das Signal von `IN0` an den Ausgang weitergegeben.
   - Ist `G` gleich `TRUE`, wird das Signal von `IN1` an den Ausgang weitergegeben.
4. **Ausgabe:** Nach erfolgreicher Auswahl wird das Ergebnis über ein weiteres Flipflop an den Plug `OUT` übertragen und dort zeitgleich mit einem Ereignis (`OUT.E1`) zur Verfügung gestellt.

Technische Besonderheiten

- **Typoffenheit durch LWORD-Abstraktion:** Die interne Verwendung von `LWORD` (64-Bit) für die Kopieroperationen (`F_MOVE`) ermöglicht eine hohe Flexibilität bei den transportierten Daten, da verschiedene Bit- und Ganzzahlmuster ohne Informationsverlust übertragen werden können.
- **Vollständige Ereignissteuerung:** Jede Änderung an einem der Eingänge oder dem Selektor triggert eine sofortige Neuberechnung. Es wird sichergestellt, dass am Ausgang stets konsistente und aktuelle Daten anliegen.

Zustandsübersicht

Selektor-Zustand (`G.D1`)	Ausgangswert (`OUT.D1`)	Trigger-Ereignis

FALSE (0)	Wert von IN0.D1	Ereignis auf IN0 oder G
TRUE (1)	Wert von IN1.D1	Ereignis auf IN1 oder G

Anwendungsszenarien

- **Sollwert-Umschaltung:** Wechseln zwischen einem Standard-Sollwert (IN0) und einem alternativen Sollwert (IN1) basierend auf einem externen Steuersignal.
- **Hand-/Automatikbetrieb:** Umschalten eines Stellsignals für ein Ventil oder einen Motor zwischen einem manuell vorgegebenen Wert (z. B. aus einer Visualisierung) und dem berechneten Wert eines Automatik-Reglers.
- **Sensor-Redundanz:** Dynamisches Umschalten auf einen Ersatzsensor (IN1), falls der Hauptsensor (IN0) über eine Diagnosefunktion als fehlerhaft markiert wird (Umschaltung gesteuert durch Fehlerbit auf G).

Vergleich mit ähnlichen Bausteinen

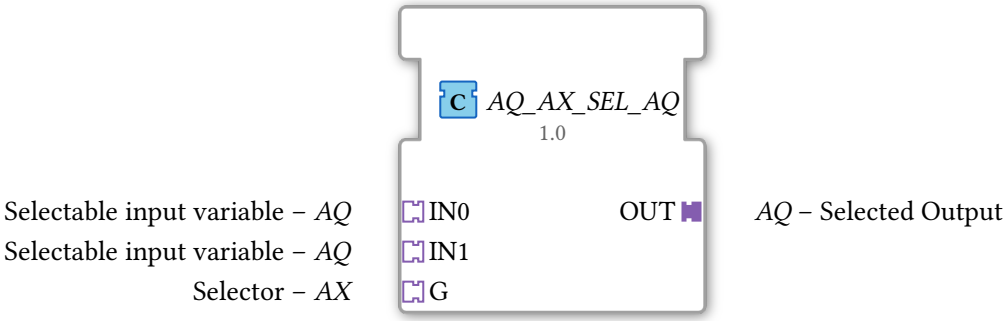
Im Vergleich zu einem Standard-SEL-Funktionsbaustein aus der IEC 61131-3 Bibliothek bietet AL_AX_SEL_AL folgende Vorteile:

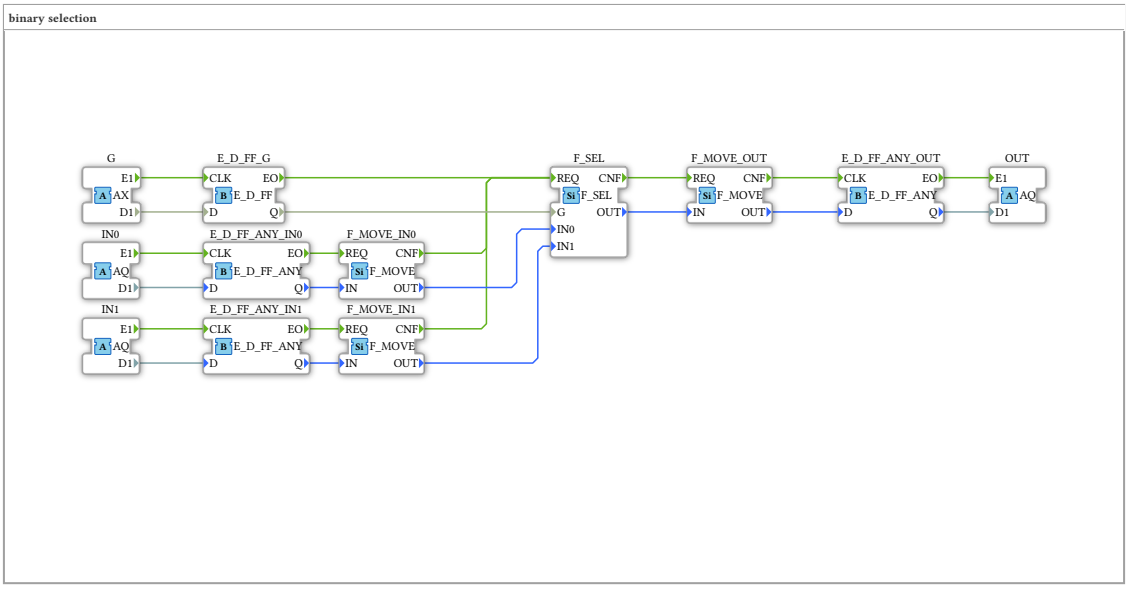
- **Weniger Verdrahtungsaufwand:** Durch die Kapselung in Adaptern müssen Ereignis- und Datenleitungen nicht einzeln gezogen werden. Ein einziger Verbindungspfad pro Kanal genügt.
- **Ereignis-Kopplung:** Der Baustein reagiert inhärent auf die in den Adaptern integrierten Event-Trigger, was eine manuelle ereignisseitige Verknüpfung im FBD (Function Block Diagram) überflüssig macht.

Fazit

Der AL_AX_SEL_AL-Baustein ist eine effiziente und saubere Lösung für Umschaltaufgaben in komplexen 4diac-Anwendungen. Durch die konsequente Nutzung von unidirektionalen Adaptern fördert er ein modulares Softwaredesign und sorgt für übersichtliche Steuerungstopologien bei gleichzeitig robuster, ereignisgesteuerter Signalverarbeitung.

AQ_AX_SEL_AQ





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein `AQ_AX_SEL_AQ` dient der binären Auswahl (Selektion) zwischen zwei analogen oder wertbasierten Eingangssignalen, die über Adapter bereitgestellt werden. Die Steuerung, welcher der beiden Eingänge an den Ausgang durchgeschaltet wird, erfolgt über einen Selektor-Adapter. Der Baustein kapselt die klassische Auswahl-Logik und bereitet sie für die ereignisgesteuerte Ausführung in IEC 61499-Umgebungen auf.

Schnittstellenstruktur

Der Funktionsbaustein kommuniziert ausschließlich über standardisierte Adapter-Schnittstellen. Dies vereinfacht die Modellierung im Applikationsnetzwerk, da zusammengehörige Daten- und Ereignislinien gebündelt übertragen werden.

Ereignis-Eingänge

Der Baustein besitzt keine direkten Ereignis-Eingänge. Die Ereignisverarbeitung erfolgt implizit über die Ereignisse der angeschlossenen Eingangs-Adapter (`IN0`, `IN1`, `G`).

Ereignis-Ausgänge

Der Baustein besitzt keine direkten Ereignis-Ausgänge. Die Weiterleitung von Ereignissen erfolgt über den Ausgangs-Adapter (`OUT`).

Daten-Eingänge

Es sind keine direkten Daten-Eingänge auf der Bausteinoberfläche vorhanden. Die Datenübergabe erfolgt gekapselt über die Sockets.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Die Datenbereitstellung erfolgt gekapselt über den Plug.

Adapter

Sockets (Eingangs-Schnittstellen)

- **G** (Typ: `adapter::types::unidirectional::AX`):
  - **Beschreibung**: Selektor (Auswahlsignal).
  - **Daten**: `D1` (Zustand des Selektors).
  - **Ereignis**: `E1` (Trigger bei Wertänderung des Selektors).
- **IN0** (Typ: `adapter::types::unidirectional::AQ`):
  - **Beschreibung**: Erster wählbarer Eingangskanal (wird bei `G = FALSE` gewählt).

  - **Daten**: `D1` (Eingangswert).
  - **Ereignis**: `E1` (Trigger bei Wertänderung an IN0).
- **IN1** (Typ: `adapter::types::unidirectional::AQ`):
  - **Beschreibung**: Zweiter wählbarer Eingangskanal (wird bei `G = TRUE` gewählt).
  - **Daten**: `D1` (Eingangswert).
  - **Ereignis**: `E1` (Trigger bei Wertänderung an IN1).

Plugs (Ausgangs-Schnittstellen)

- **OUT** (Typ: `adapter::types::unidirectional::AQ`):
  - **Beschreibung**: Der ausgewählte Ausgangskanal.
  - **Daten**: `D1` (Ausgangswert des selektierten Eingangs).
  - **Ereignis**: `E1` (Ausgangs-Ereignis zur Signalisierung eines neuen gültigen Wertes).

Funktionsweise

Der Baustein ist intern als FB-Netzwerk realisiert und arbeitet nach folgendem Ablauf:

1. **Ereigniserfassung**: Sobald ein Ereignis `E1` an einem der Eingänge (`IN0`, `IN1`) oder dem Selektor (`G`) eintrifft, wird dieses über flanken- bzw. wertgesteuerte Hilfsbausteine (`E_D_FF` und `E_D_FF_ANY`) abgefangen und synchronisiert.
2. **Datenpufferung**: Die aktuellen Werte der Adapter-Eingänge werden mittels `F_MOVE`-Funktionsbausteinen im internen Netzwerk zwischengespeichert und typkonform aufbereitet.
3. **Auswahl-Logik (Selection)**: Der interne Kernbaustein `F_SEL` (basierend auf der standardisierten Auswahlfunktion nach IEC 61131-3) wertet den Zustand des Selektors aus:
   - Ist `G.D1` auf `FALSE` gesetzt, wird der Wert von `IN0` durchgeschaltet.
   - Ist `G.D1` auf `TRUE` gesetzt, wird der Wert von `IN1` durchgeschaltet.
4. **Ausgabe**: Der selektierte Wert wird an den Ausgangs-Plug `OUT.D1` übergeben. Gleichzeitig wird das Ausgangsereignis `OUT.E1` gefeuert, um nachfolgende Programmteile im Steuerungsnetzwerk über die Wertänderung zu informieren.

Technische Besonderheiten

- **Ereignis-Daten-Konsistenz**: Durch die Verwendung von internen D-Flipflops wird sichergestellt, dass die Datenwerte zum exakten Zeitpunkt des Eintreffens des entsprechenden Änderungsereignisses gelesen und verarbeitet werden.
- **Typisierung**: Der Baustein nutzt intern `F_MOVE` mit dem Datentyp `BYTE` zur Kapselung und Zuweisung der Adapterdaten. Das bedeutet, dass die genutzten `AQ`-Schnittstellen binär- bzw. bytekompatible Datenstrukturen übertragen.

Zustandsübersicht

Da es sich bei `AQ_AX_SEL_AQ` um einen rein daten- und ereignisgesteuerten Kombinationsbaustein ohne eigenen internen Zustandsautomaten (ECC) handelt, lässt sich das Verhalten über folgende logische Tabelle beschreiben:

Selektor (`G.D1`)	Wert `IN0.D1`	Wert `IN1.D1`	Ausgang (`OUT.D1`)	Verhalten des Ausgangsereignisses (`OUT.E1`)
FALSE	`Wert_A`	`Wert_B`	**`Wert_A`**	Wird gefeuert, sobald ein Ereignis an `IN0.E1`, `IN1.E1` oder `G.E1` auftritt.
TRUE	`Wert_A`	`Wert_B`	**`Wert_B`**	Wird gefeuert, sobald ein Ereignis an `IN0.E1`, `IN1.E1` oder `G.E1` auftritt.

Anwendungsszenarien

- **Sensor-Redundanz / Umschaltung**: Automatisches oder manuelles Umschalten zwischen einem Primärsensor (`IN0`) und einem Sekundär-/Backupsensor (`IN1`) im Fehlerfall über das Steuersignal `G`.
- **Hand-/Automatikbetrieb**: Auswahl von Sollwerten für Aktoren. Im Automatikmodus (`G = TRUE`) wird der berechnete Reglerwert (`IN1`) verwendet, im Handbetrieb (`G = FALSE`) ein vordefinierter Festwert oder ein manuell vorgegebener Parameter (`IN0`).

- **Rezepturgesteuerte Parametrierung**: Dynamische Umschaltung von Prozessgrenzwerten während des laufenden Betriebs.

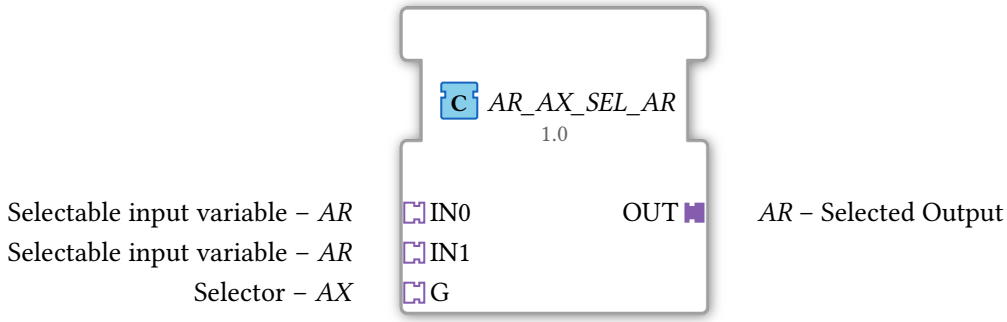
Vergleich mit ähnlichen Bausteinen

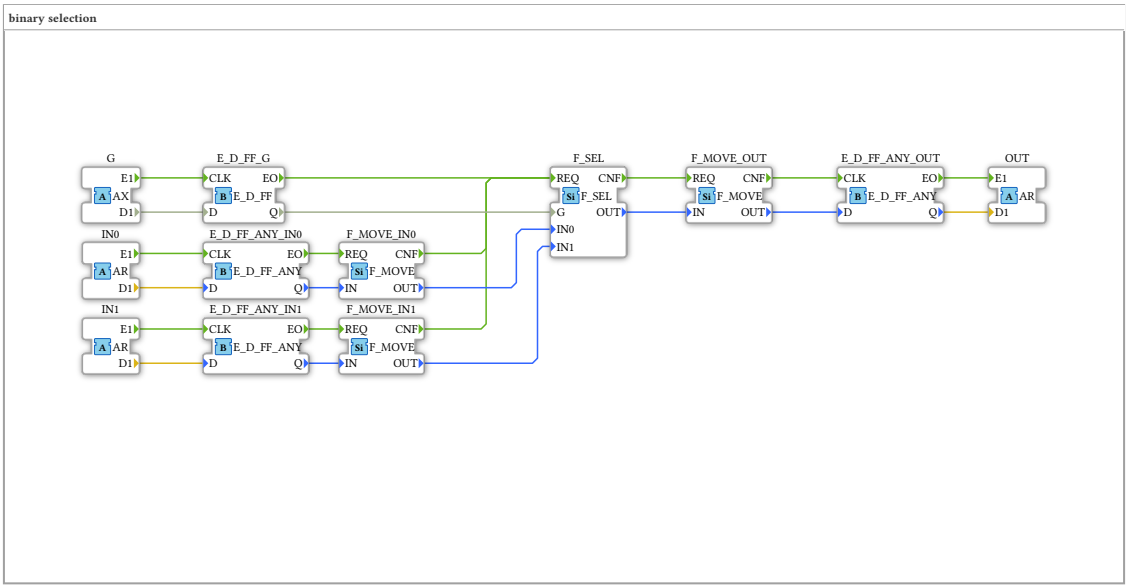
- **Standard `SEL` (IEC 61131-3)**: Der klassische `SEL`-Baustein besitzt keine Ereignissteuerung und reagiert rein zyklisch auf anstehende Daten. `AQ_AX_SEL_AQ` erweitert diese Funktionalität um das ereignisbasierte Paradigma von IEC 61499.
- **Klassische Multiplexer (MUX)**: Während ein Multiplexer meist aus einer Vielzahl von Eingängen wählt, beschränkt sich dieser Baustein gezielt auf eine effiziente 1-aus-2-Auswahl (binäre Selektion), was den Parametrierungs- und Verdrahtungsaufwand minimiert.

Fazit

Der `AQ_AX_SEL_AQ` ist ein nützlicher Infrastruktur-Baustein für modulare 4diac-Anwendungen. Durch die vollständige Kapselung der Signale in Adapter-Verbindungen trägt er maßgeblich zur Übersichtlichkeit im Application-Window bei und sorgt für eine robuste, ereignisgesteuerte Signalweiterleitung.

AR_AX_SEL_AR





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein `AR_AX_SEL_AR` dient der binären Auswahl (Selektion) zwischen zwei analogen Eingangssignalen, die über Adapter übertragen werden. Basierend auf dem Zustand eines Auswahlsignals (Selector) wird einer der beiden Eingänge auf den Ausgang durchgeschaltet.

Durch die konsequente Verwendung von Adaptern anstelle von klassischen diskreten Daten- und Ereignispins wird die Komplexität der Verkabelung im übergeordneten IEC 61499 Applikationsdiagramm signifikant reduziert.

Schnittstellenstruktur

Da dieser Funktionsbaustein vollständig auf adapterbasierte Kommunikation setzt, verfügt er auf oberster Ebene über keine direkten, klassischen Event- oder Daten-Schnittstellen. Die gesamte Kommunikation wird über die deklarierten Adapter abgewickelt.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden (Ereignisse werden über die Adapter-Schnittstellen empfangen).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden (Ereignisse werden über die Adapter-Schnittstellen gesendet).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangsschnittstellen)

- **G** (Typ: `adapter::types::unidirectional::AX`): Der Auswahladapter (Selector). Das hier anliegende Signal steuert, welcher der beiden Eingänge (`IN0` oder `IN1`) an den Ausgang weitergeleitet wird.
- **IN0** (Typ: `adapter::types::unidirectional::AR`): Der erste auswählbare Signal-Eingang. Dieser Wert wird an den Ausgang `OUT` weitergegeben, wenn der Selektor `G` den Zustand `FALSE` (0) aufweist.
- **IN1** (Typ: `adapter::types::unidirectional::AR`): Der zweite auswählbare Signal-Eingang. Dieser Wert wird an den Ausgang `OUT` weitergegeben, wenn der Selektor `G` den Zustand `TRUE` (1) aufweist.

Plugs (Ausgangsschnittstellen)

- **OUT** (Typ: `adapter::types::unidirectional::AR`): Der selektierte Ausgangsadapter. Er liefert den Wert des jeweils aktiven Eingangs inklusive des dazugehörigen Aktualisierungsereignisses.

Funktionsweise

Im Inneren des Funktionsbausteins `AR_AX_SEL_AR` befindet sich ein Netzwerk aus Standard-IEC-61131-3- und IEC-61499-Bausteinen, welches die Adapter-Signale verarbeitet:

1. **Ereignis- und Datenerfassung:** Sobald ein Ereignis `E1` an einem der Eingangs-Adapter (`IN0`, `IN1` oder `G`) eintrifft, wird dieses durch interne flankengetriggerte Flip-Flops (`E_D_FF` bzw. `E_D_FF_ANY`) abgefangen. Die entsprechenden Datenwerte (`D1`) werden zwischengespeichert.
2. **Datenkonvertierung:** Die Werte der analogen Eingänge werden über Konvertierungsbausteine des Typs `F_MOVE` (konfiguriert auf den Datentyp `REAL`) geleitet, um eine konsistente Datenverarbeitung zu gewährleisten.
3. **Auswahllogik (Multiplexing):** Der Standard-Auswahlbaustein `F_SEL` übernimmt die eigentliche Selektion:
   - Ist der Wert von `G.D1` gleich `FALSE`, wird das Signal von `IN0` an den Ausgang weitergeleitet.
   - Ist der Wert von `G.D1` gleich `TRUE`, wird das Signal von `IN1` weitergeleitet.
4. **Ausgabe:** Das ausgewählte Signal wird über einen weiteren `F_MOVE`-Baustein an das Ausgangs-Flip-Flop `E_D_FF_ANY_OUT` übergeben. Dieses generiert das Ausgangsereignis `E1` am Plug `OUT` und stellt den ausgewählten Wert an `OUT.D1` bereit.

Technische Besonderheiten

- **Unidirektionale Adapterstruktur:** Der Baustein nutzt unidirektionale Adaptertypen (`AR` für analoge Werte, `AX` für binäre Werte), was eine klare und störungsfreie Signalrichtung im System gewährleistet.
- **Typkonsistenz:** Intern arbeitet der Baustein mit dem Datentyp `REAL` für die analogen Signale (konfiguriert über die Attribute der `F_MOVE`-Bausteine).
- **Ereignisgesteuert:** Eine Neuberechnung und Aktualisierung des Ausgangs erfolgt sofort, sobald sich entweder die Selektion `G` ändert oder neue Werte an den Eingängen `IN0` bzw. `IN1` signalisiert werden.

Zustandsübersicht

Das Verhalten lässt sich über folgende einfache Logiktabelle beschreiben:

Zustand Selektor (`G.D1`)	Wert am Ausgang (`OUT.D1`)	Triggerndes Ereignis
`FALSE`	Wert von `IN0.D1`	Ereignis an `IN0.E1` oder Änderung an `G.E1`
`TRUE`	Wert von `IN1.D1`	Ereignis an `IN1.E1` oder Änderung an `G.E1`

Anwendungsszenarien

- **Sensorredundanz:** Umschalten zwischen einem primären und einem sekundären analogen Sensor (z. B. Temperatursensor oder Drucksensor) bei Ausfall des Primärsensors.
- **Hand-/Automatik-Umschaltung:** Wahlweise Aufschaltung eines automatischen Regelwerts (z. B. berechnet durch einen PID-Regler) oder eines manuell vorgegebenen Sollwerts auf ein Stellglied.
- **Signalrouting:** Dynamische Pfadauswahl in komplexeren verfahrenstechnischen oder agrartechnischen Steuerungsanwendungen.

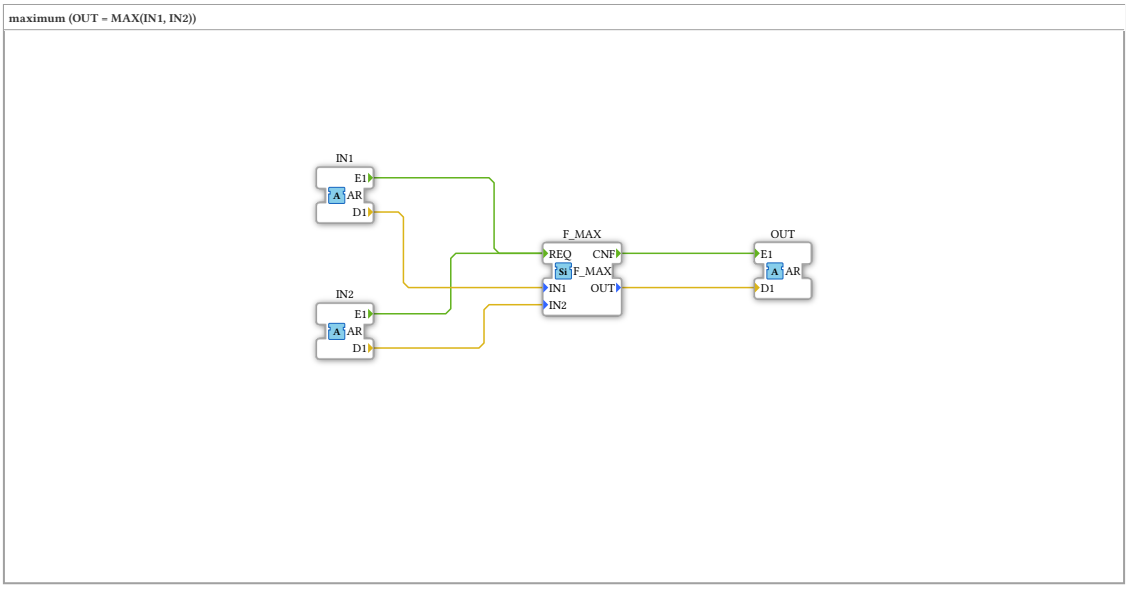
Vergleich mit ähnlichen Bausteinen

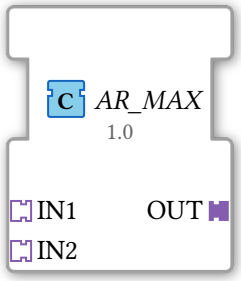
Im Vergleich zum Standard-IEC-61131-Baustein `F_SEL` bietet `AR_AX_SEL_AR` den Vorteil der vollständigen Kapselung über Adapter. Während beim Standard-`F_SEL` Events und Daten getrennt im Netzwerk verdrahtet und synchronisiert werden müssen, reduziert `AR_AX_SEL_AR` den Verdrahtungsaufwand im übergeordneten System auf nur drei Verbindungen (zwei Eingangs-Adapter, ein Ausgangs-Adapter) plus das Selektionssignal.

Fazit

Der `AR_AX_SEL_AR` ist ein äußerst nützlicher Hilfsbaustein für IEC 61499 Applikationen. Durch die Abstraktion der Daten- und Eventströme in standardisierte Adapterkanäle sorgt er für ein aufgeräumtes Applikationsdesign und vereinfacht die Implementierung von Umschalt- und Redundanzlogiken für analoge Signale.

AR_MAX





Input value 1 – *AR*
Input value 2 – *AR*
AR – MAX(IN1, IN2)

Einleitung

Der Funktionsbaustein `AR_MAX` dient der Ermittlung des Maximalwerts aus zwei analogen Eingangssignalen, die über Adapter (`AR`) übertragen werden. Er vergleicht die Datenwerte der beiden Eingangs-Sockets `IN0` und `IN1` und gibt den größeren der beiden Werte an den Ausgangs-Plug `OUT` weiter.

Durch die konsequente Verwendung von Adaptern anstelle von diskreten Daten- und Ereignispins wird der Verdrahtungsaufwand im übergeordneten IEC 61499 Applikationsdiagramm signifikant reduziert.

Schnittstellenstruktur

Da dieser Funktionsbaustein vollständig auf adapterbasierte Kommunikation setzt, verfügt er auf oberster Ebene über keine direkten, klassischen Event- oder Daten-Schnittstellen. Die gesamte Kommunikation wird über die deklarierten Adapter abgewickelt.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden (Ereignisse werden über die Adapter-Schnittstellen empfangen).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden (Ereignisse werden über die Adapter-Schnittstellen gesendet).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangsschnittstellen)

- **IN0** (Typ: `adapter::types::unidirectional::AR`): Der erste analoge Eingangssignal-Adapter.
- **IN1** (Typ: `adapter::types::unidirectional::AR`): Der zweite analoge Eingangssignal-Adapter.

Plugs (Ausgangsschnittstellen)

- **OUT** (Typ: `adapter::types::unidirectional::AR`): Der Ausgangsadapter. Er liefert den Maximalwert $\max(\text{IN0.D1}, \text{IN1.D1})$ inklusive des dazugehörigen Aktualisierungsereignisses.

Funktionsweise

Sobald ein Ereignis `E1` an einem der beiden Eingangs-Adapter (`IN0` oder `IN1`) eintrifft, vergleicht der Baustein die aktuellen Datenwerte `IN0.D1` und `IN1.D1`:

$$\text{OUT.D1} = \max(\text{IN0.D1}, \text{IN1.D1}) \tag{2.132}$$

Der ermittelte Maximalwert wird auf `OUT.D1` gelegt und zusammen mit dem Ausgangsereignis `OUT.E1` emittiert.

Technische Besonderheiten

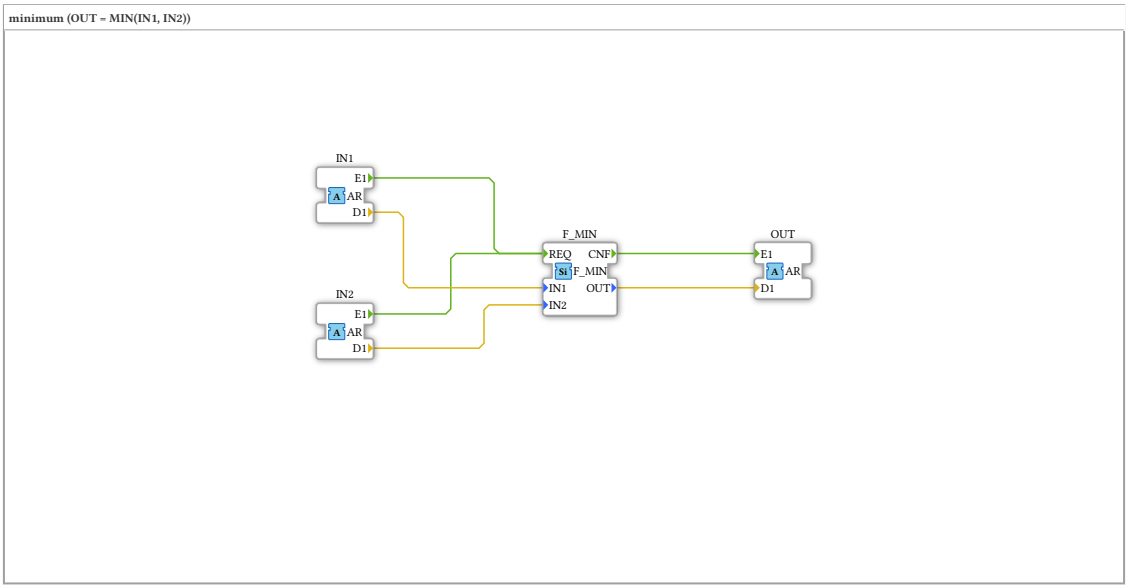
- **Unidirektionale Adapterstruktur**: Nutzt standardisierte `AR`-Adapterkanäle für eine saubere Signalrichtung.
- **Ereignisgesteuert**: Die Neuberechnung und Ausgabe erfolgt unmittelbar beim Eintreffen eines neuen Messwerts an `IN0` oder `IN1`.

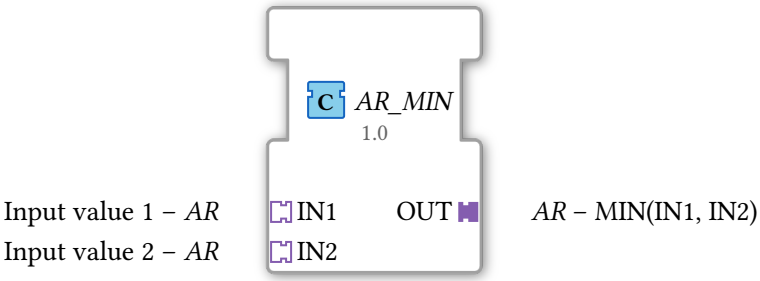
- **Typkonformität**: Arbeitet intern auf Basis des Fließkommadatentyps `REAL`.

Fazit

Der `AR_MAX` ist ein nützlicher Auswahlbaustein zur Bildung von Extremwerten (Höchstwertauswahl) in adapterbasierten IEC 61499 Anwendungen.

AR_MIN





Einleitung

Der Funktionsbaustein `AR_MIN` dient der Ermittlung des Minimalwerts aus zwei analogen Eingangssignalen, die über Adapter (`AR`) übertragen werden. Er vergleicht die Datenwerte der beiden Eingangs-Sockets `IN0` und `IN1` und gibt den kleineren der beiden Werte an den Ausgangs-Plug `OUT` weiter.

Durch die konsequente Verwendung von Adaptern anstelle von diskreten Daten- und Ereignispins wird der Verdrahtungsaufwand im übergeordneten IEC 61499 Applikationsdiagramm signifikant reduziert.

Schnittstellenstruktur

Da dieser Funktionsbaustein vollständig auf adapterbasierte Kommunikation setzt, verfügt er auf oberster Ebene über keine direkten, klassischen Event- oder Daten-Schnittstellen. Die gesamte Kommunikation wird über die deklarierten Adapter abgewickelt.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden (Ereignisse werden über die Adapter-Schnittstellen empfangen).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden (Ereignisse werden über die Adapter-Schnittstellen gesendet).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangsschnittstellen)

- **IN0** (Typ: `adapter::types::unidirectional::AR`): Der erste analoge Eingangssignal-Adapter.
- **IN1** (Typ: `adapter::types::unidirectional::AR`): Der zweite analoge Eingangssignal-Adapter.

Plugs (Ausgangsschnittstellen)

- **OUT** (Typ: `adapter::types::unidirectional::AR`): Der Ausgangsadapter. Er liefert den Minimalwert $\min(\text{IN0.D1}, \text{IN1.D1})$ inklusive des dazugehörigen Aktualisierungsereignisses.

Funktionsweise

Sobald ein Ereignis `E1` an einem der beiden Eingangs-Adapter (`IN0` oder `IN1`) eintrifft, vergleicht der Baustein die aktuellen Datenwerte `IN0.D1` und `IN1.D1`:

$$\text{OUT.D1} = \min(\text{IN0.D1}, \text{IN1.D1}) \tag{2.133}$$

Der ermittelte Minimalwert wird auf `OUT.D1` gelegt und zusammen mit dem Ausgangsereignis `OUT.E1` emittiert.

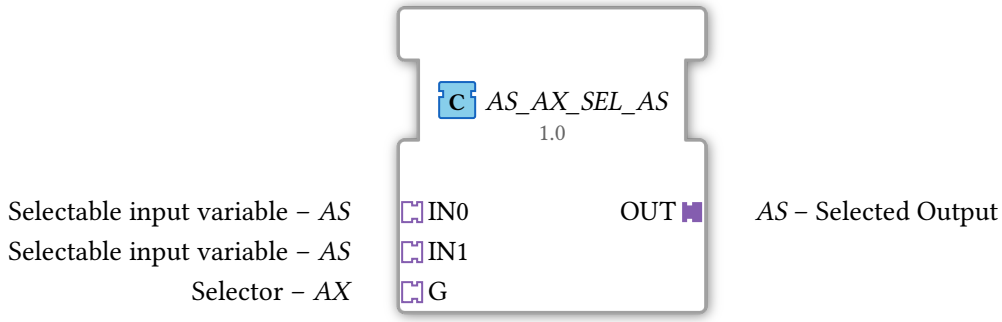
Technische Besonderheiten

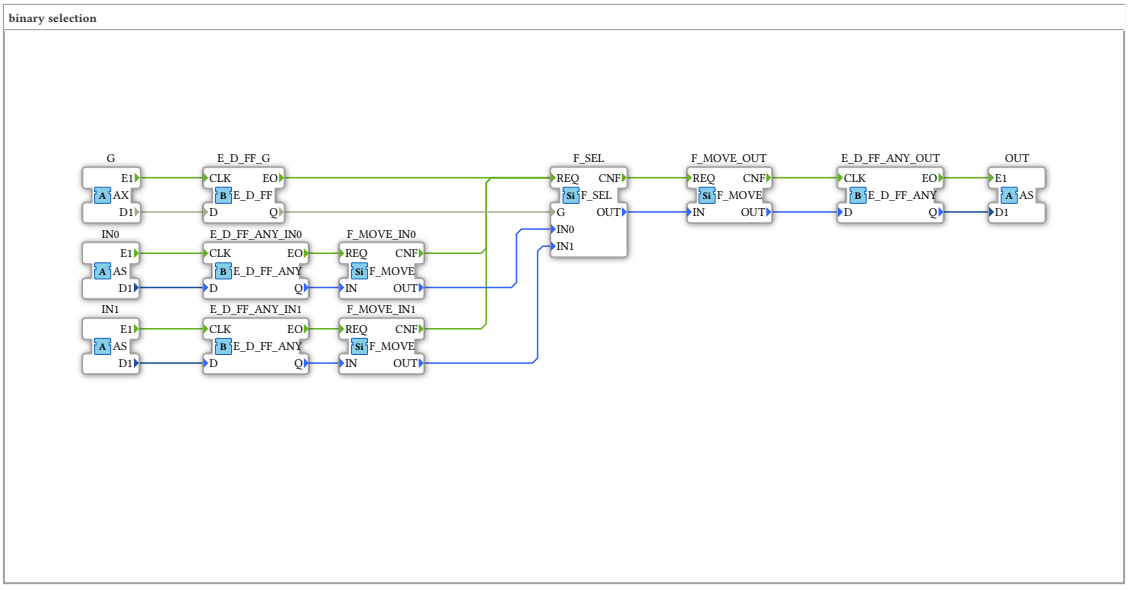
- **Unidirektionale Adapterstruktur**: Nutzt standardisierte `AR`-Adapterkanäle für eine saubere Signalrichtung.
- **Ereignisgesteuert**: Die Neuberechnung und Ausgabe erfolgt unmittelbar beim Eintreffen eines neuen Messwerts an `IN0` oder `IN1`.
- **Typkonformität**: Arbeitet intern auf Basis des Fließkommadatentyps `REAL`.

Fazit

Der `AR_MIN` ist ein nützlicher Auswahlbaustein zur Bildung von Extremwerten (Mindestwertauswahl) in adapterbasierten IEC 61499 Anwendungen.

AS_AX_SEL_AS





(Kein Bild vorhanden)

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein `AS_AX_SEL_AS` dient der binären Auswahl (Selektion) zwischen zwei über Adapter bereitgestellten Eingangssignalen. Basierend auf dem Zustand eines Selektions-Adapters (`G`) wird entweder der Wert des Eingangs `IN0` oder des Eingangs `IN1` an den Ausgang `OUT` weitergeleitet.

Dieser Baustein kapselt die klassische Multiplexer-Funktion (entsprechend `SEL` aus der IEC 61131-3) in einer eventgesteuerten IEC 61499-Architektur unter Verwendung von unidirektionalen Adaptern. Dies vereinfacht die Signalverdrahtung in komplexen Steuerungsanwendungen erheblich.

Schnittstellenstruktur

Der Funktionsbaustein besitzt keine klassischen ereignis- oder datenbasierten Ein- und Ausgänge auf der Hauptebene. Die gesamte Kommunikation wird über Adapter-Schnittstellen abgewickelt.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die Ereignisverarbeitung erfolgt gekapselt über die Sockets der Adapter.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Ereignisweiterleitung erfolgt gekapselt über den Plug des Ausgangs-Adapters.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Schnittstellen)

- **G** (Typ: `adapter::types::unidirectional::AX`):
  - **Beschreibung:** Selektor-Schnittstelle. Bestimmt, welcher Eingang auf den Ausgang geschaltet wird.
  - **Ereignis:** `E1` (Trigger für die Aktualisierung des Selektionssignals).
  - **Daten:** `D1` (Boolean-Wert; bestimmt den aktiven Pfad).
- **IN0** (Typ: `adapter::types::unidirectional::AS`):
  - **Beschreibung:** Erster auswählbarer Datenkanal (wird gewählt, wenn `G` den Wert `FALSE` hat).
  - **Ereignis:** `E1` (Signalisiert neue Daten an `IN0`).

  - **Daten:** `D1` (Datenwert des ersten Kanals).
- **IN1** (Typ: `adapter::types::unidirectional::AS`):
  - **Beschreibung:** Zweiter auswählbarer Datenkanal (wird gewählt, wenn `G` den Wert `TRUE` hat).
  - **Ereignis:** `E1` (Signalisiert neue Daten an `IN1`).
  - **Daten:** `D1` (Datenwert des zweiten Kanals).

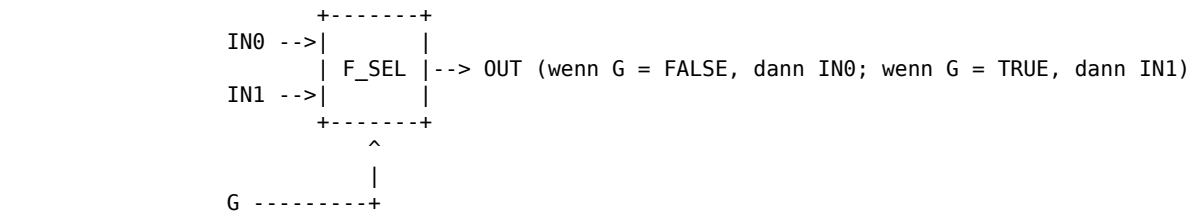
Plugs (Ausgangs-Schnittstellen)

- **OUT** (Typ: `adapter::types::unidirectional::AS`):
  - **Beschreibung:** Der ausgewählte Ausgangskanal, welcher die Daten des aktiven Eingangs spiegelt.
  - **Ereignis:** `E1` (Wird ausgelöst, wenn ein neuer selektierter Wert ausgegeben wird).
  - **Daten:** `D1` (Der weitergeleitete Datenwert von `IN0` oder `IN1`).

Funktionsweise

Im Inneren des Funktionsbausteins befindet sich ein Netzwerk aus Standard-Funktionsbausteinen, welches die Adapter-Signale verarbeitet:

1. **Signalspeicherung (Latch):** Die über die Adapter eingehenden Daten (`G.D1`, `IN0.D1`, `IN1.D1`) werden bei Eintreffen des jeweiligen Ereignisses (`E1`) in ereignisgesteuerten D-Flip-Flops (`E_D_FF` bzw. `E_D_FF_ANY`) zwischengespeichert.
2. **Daten-Konvertierung & Weiterleitung:** Über `F_MOVE`-Bausteine werden die zwischengespeicherten Daten der Typen `IN0` und `IN1` an den zentralen Auswahlbaustein übergeben. Intern sind diese Bewegungsoperationen für den Datentyp `SINT` (Short Integer) konfiguriert.
3. **Auswahl-Logik (`F_SEL`):** Der Baustein `F_SEL` (IEC 61131-3 Selection) führt die eigentliche Auswahl durch:
   - Ist der Zustand am Selektor `G` gleich `FALSE`, wird der Wert von `IN0` durchgeschaltet.
   - Ist der Zustand am Selektor `G` gleich `TRUE`, wird der Wert von `IN1` durchgeschaltet.
4. **Ausgabe-Triggerung:** Nach erfolgreicher Selektion wird das Ergebnis über einen weiteren `F_MOVE`-Baustein an das Ausgangs-Flip-Flop `E_D_FF_ANY_OUT` geleitet. Dieses erzeugt das Ausgangsereignis `OUT.E1` und legt den selektierten Wert an `OUT.D1` an.



Technische Besonderheiten

- **Datentyp-Spezifität:** Obwohl die Adapter flexibel ausgelegt sein können, sind die internen Konvertierungsglieder (`F_MOVE`) fest auf den Datentyp **`SINT`** parametriert. Der Baustein ist somit speziell für die Auswahl von vorzeichenbehafteten 8-Bit-Ganzzahlen (Short Integers) konzipiert.
- **Ereignisgesteuert:** Jede Änderung oder jedes Ereignis an den Eingängen `IN0`, `IN1` oder `G` stößt die Kette neu an und aktualisiert den Ausgang `OUT`.

Zustandsübersicht

Die folgende Tabelle zeigt das logische Verhalten des Bausteins basierend auf dem Selektionssignal `G.D1`:

Zustand G.D1	Ausgewählter Pfad	Ausgangswert (OUT.D1)	Ausgangsereignis (OUT.E1)
`FALSE`	Kanal 0 (`IN0`)	Wert von `IN0.D1`	Wird bei Wertänderung/Ereignis getriggert
`TRUE`	Kanal 1 (`IN1`)	Wert von `IN1.D1`	Wird bei Wertänderung/Ereignis getriggert

Anwendungsszenarien

- **Umschaltung von Betriebsmodi:** Routing von Sollwerten (z. B. Umschalten zwischen einem manuell vorgegebenen SINT-Wert und einem Automatik-Sollwert).
- **Signal-Quellen-Selektion:** Dynamisches Auswählen zwischen zwei Sensorwerten oder Parameterkonfigurationen, die über standardisierte unidirektionale Adapterstrukturen im System verteilt werden.

- **Kanalbündelung:** Reduzierung des Verdrahtungsaufwands in 4diac-Applikationen durch die Kapselung von Daten und Events in Adaptern.

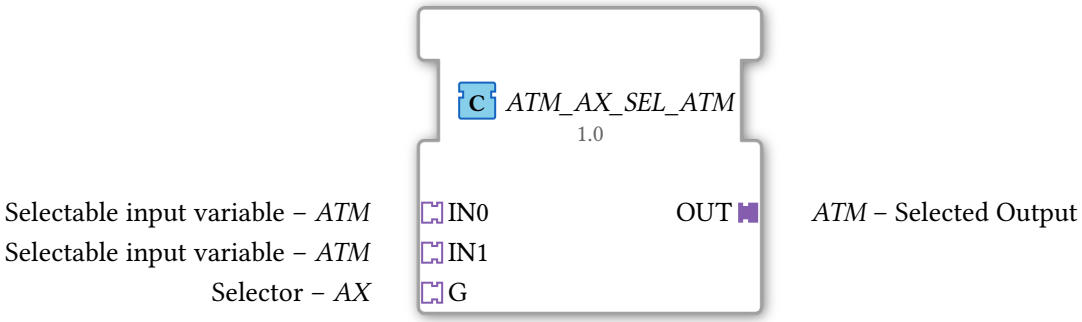
Vergleich mit ähnlichen Bausteinen

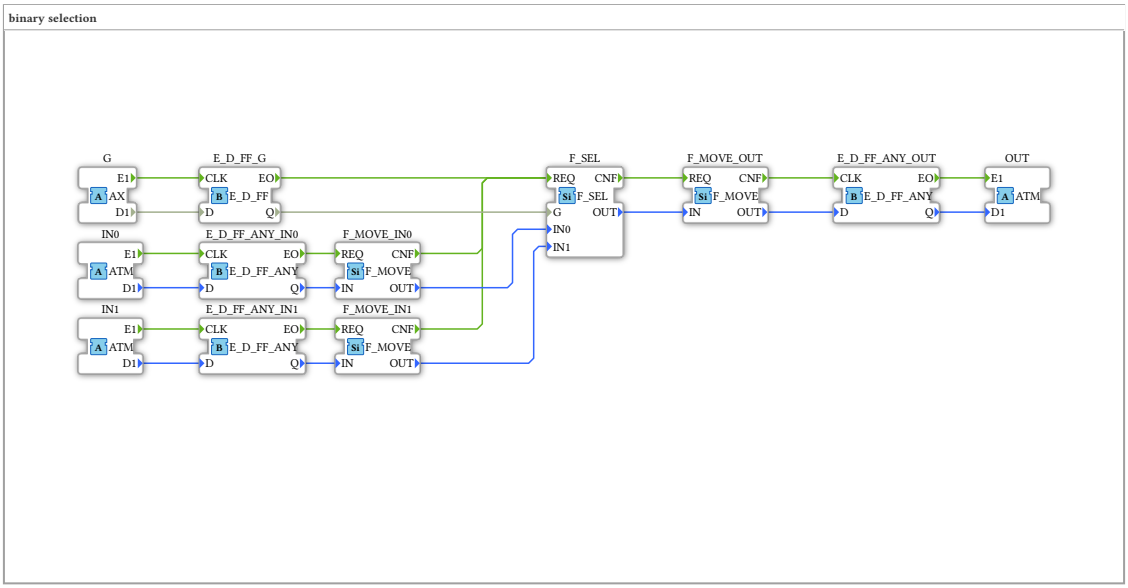
- **`F_SEL` (Standard-Auswahlbaustein):** Arbeitet direkt mit Standard-Datentypen und benötigt explizite Event-Verbindungen auf Applikationsebene. `AS_AX_SEL_AS` hingegen kapselt diese Logik vollständig und nutzt Adapter, was das Anwendungsdiagramm übersichtlicher macht.
- **Andere adapterbasierte Selektoren:** Unterscheiden sich im Wesentlichen durch die unterstützten Datentypen in den Adaptern (z. B. `INT`, `REAL` statt `SINT`).

Fazit

Der Baustein `AS_AX_SEL_AS` stellt eine elegante, saubere und wiederverwendbare Lösung dar, um eine binäre Signal-Umschaltung auf Basis von unidirektionalen Adaptern in 4diac-IDE zu realisieren. Durch die interne Verwendung standardisierter Komponenten ist ein deterministisches Verhalten im ereignisgesteuerten System gewährleistet.

ATM_AX_SEL_ATM





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein `ATM_AX_SEL_ATM` dient als binärer Selektor (Multiplexer) für Adapter-Schnittstellen gemäß dem IEC 61499 Standard. Er ermöglicht es, basierend auf dem Zustand eines Auswahladapters (`G` vom Typ `AX`), dynamisch zwischen zwei Eingangsadaptern (`IN0` und `IN1` vom Typ `ATM`) zu wählen und das ausgewählte Signal an den Ausgangsadapter (`OUT` vom Typ `ATM`) weiterzuleiten. Der Baustein arbeitet rein ereignisgesteuert und ist intern für die Verarbeitung von zeitbasierten Signalen (`TIME`) ausgelegt.

Schnittstellenstruktur

Der Funktionsbaustein besitzt keine direkt herausgeführten Standard-Ereignis- oder Datenpins. Die gesamte Kommunikation wird elegant über Sockets (Eingangs-Adapter) und Plugs (Ausgangs-Adapter) gekapselt.

Ereignis-Eingänge

Keine direkten Ereigniseingänge am FB-Körper vorhanden. Die Ereignissteuerung erfolgt implizit über die Adapter-Sockets.

Ereignis-Ausgänge

Keine direkten Ereignisausgänge am FB-Körper vorhanden. Die Ereignissignalisierung erfolgt implizit über den Adapter-Plug.

Daten-Eingänge

Keine direkten Dateneingänge am FB-Körper vorhanden.

Daten-Ausgänge

Keine direkten Datenausgänge am FB-Körper vorhanden.

Adapter

Sockets (Eingangs-Schnittstellen)

- **IN0** (Typ: `adapter::types::unidirectional::ATM`): Der erste auswählbare Eingangsadapter (entspricht der Selektion bei logisch `FALSE`).
- **IN1** (Typ: `adapter::types::unidirectional::ATM`): Der zweite auswählbare Eingangsadapter (entspricht der Selektion bei logisch `TRUE`).
- **G** (Typ: `adapter::types::unidirectional::AX`): Der Auswahladapter (Selector). Das hier anliegende Signal bestimmt, welcher der beiden Eingänge an den Ausgang durchgeschaltet wird.

Plugs (Ausgangs-Schnittstellen)

- **OUT** (Typ: `adapter::types::unidirectional::ATM`): Der Ausgangsadapter, der die Daten und Ereignisse des jeweils selektierten Eingangs (`IN0` oder `IN1`) führt.

Funktionsweise

Die interne Struktur des `ATM_AX_SEL_ATM` besteht aus einem Netzwerk standardisierter Hilfsbausteine, die eine zuverlässige und konsistente Signalweiterleitung sicherstellen:

1. **Ereigniserfassung und Pufferung**: Sobald an einem der Sockets (`IN0`, `IN1` oder `G`) ein Ereignis (`E1`) eintrifft, wird dieses über flankengetriggerte Flip-Flops (`E_D_FF_ANY` bzw. `E_D_FF`) erfasst. Die Datenwerte (`D1`) werden stabilisiert und an interne Puffer weitergegeben.
2. **Datenkonvertierung**: Die Daten der Eingänge `IN0` und `IN1` werden über zwei `F_MOVE`-Bausteine (konfiguriert auf den Datentyp `TIME`) gepuffert und für die Selektionslogik bereitgestellt.
3. **Auswahl (Selektion)**: Der Baustein `F_SEL` (ein Standard-Auswahlbaustein der IEC 61131-3) wertet den Zustand des Selektors `G` aus:
   - Ist das Signal von `G` gleich `FALSE`, wird der Wert von `IN0` gewählt.
   - Ist das Signal von `G` gleich `TRUE`, wird der Wert von `IN1` gewählt.
4. **Ausgabe**: Der ausgewählte Wert wird über einen weiteren `F_MOVE`-Baustein an den Ausgangsadapter `OUT` übergeben, woraufhin zeitgleich das Ausgangsereignis (`OUT.E1`) ausgelöst wird.

Technische Besonderheiten

- **Adapter-Kapselung**: Durch die Verwendung von Adaptern anstelle von Einzelpins wird die Anzahl der Verbindungslinien im übergeordneten Steuerungsdiagramm drastisch reduziert, was die Lesbarkeit des Gesamtsystems verbessert.
- **Datentyp-Fokus**: Der Baustein ist intern durch die Parametrierung der `F_MOVE`-Bausteine speziell für den Datentyp `TIME` (Zeitwerte) optimiert.
- **Ereignisintegrität**: Die Verwendung von Ereignis-Flip-Flops stellt sicher, dass keine Signaländerung verloren geht und der Ausgang nur dann aktualisiert wird, wenn sich tatsächlich relevante Eingangsdaten oder der Selektor ändern.

Zustandsübersicht

Da es sich um ein zusammengesetztes Funktionsbaustein-Netzwerk (Composite FB) handelt, besitzt der Baustein keine eigene klassische ECC (Event Execution Control) Zustandsmaschine. Das Verhalten ist rein datenfluss- und ereignisgesteuert:

Zustand Selector `G.D1`	Triggerndes Ereignis	Zustand Ausgang `OUT.D1`	Ausgegebenes Ereignis
`FALSE`	Änderung an `IN0.E1` oder `G.E1`	Wert von `IN0.D1`	`OUT.E1`
`TRUE`	Änderung an `IN1.E1` oder `G.E1`	Wert von `IN1.D1`	`OUT.E1`

Anwendungsszenarien

- **Umschaltung von Prozesszeiten**: In industriellen oder landwirtschaftlichen Prozessen, bei denen zwischen zwei unterschiedlichen Takt-, Spül- oder Verzögerungszeiten (z. B. Sommer-/Winterbetrieb oder Rezeptur A/Rezeptur B) umgeschaltet werden muss.
- **Sollwert-Umschaltung**: Dynamische Auswahl eines zeitbasierten Sollwerts aus zwei unterschiedlichen Quellen basierend auf einer übergeordneten Betriebsart.

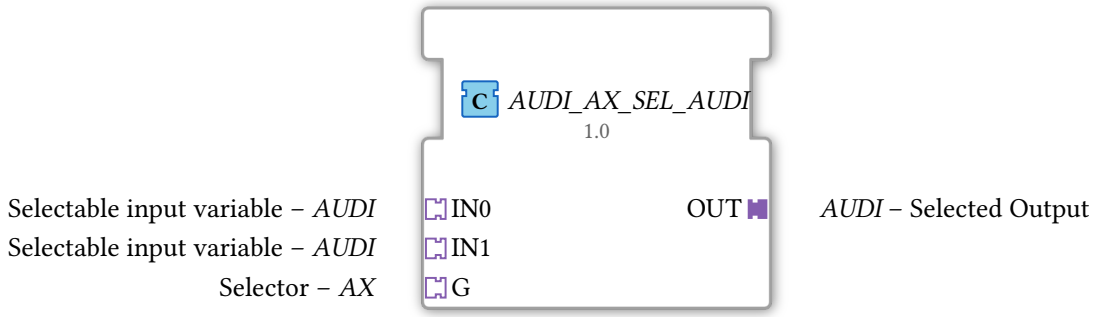
Vergleich mit ähnlichen Bausteinen

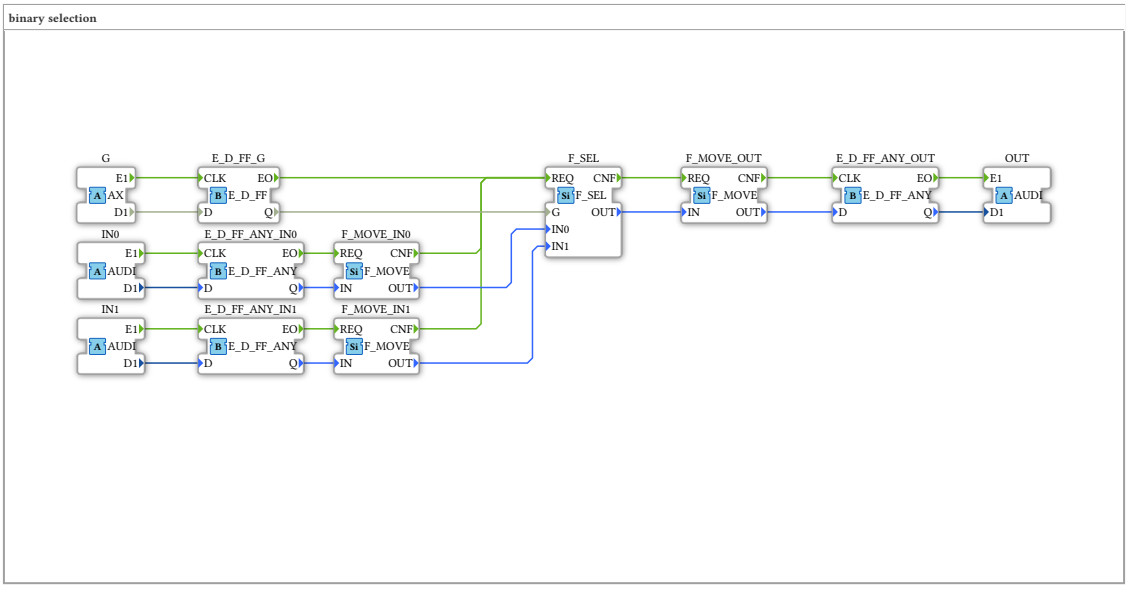
- **Standard `F_SEL`**: Der klassische `F_SEL`-Baustein arbeitet direkt auf elementaren Datentypen und besitzt keine integrierte Ereignissteuerung für Adapter. `ATM_AX_SEL_ATM` verlagert diese Logik auf die strukturierte Adapter-Ebene.
- **Standard-Multiplexer**: Im Gegensatz zu universellen Multiplexern, die oft komplexe Bus-Strukturen aufweisen, ist dieser Baustein hochspezialisiert, extrem leichtgewichtig und perfekt auf die Kombination aus `ATM`- und `AX`-Adaptern abgestimmt.

Fazit

Der `ATM_AX_SEL_ATM` ist ein robuster und wiederverwendbarer Hilfsbaustein für moderne Steuerungsarchitekturen nach IEC 61499. Er löst die Aufgabe der binären Signalumschaltung für zeitbasierte Datenströme auf elegante Weise unter Ausnutzung der Vorteile der Adapter-Technologie.

AUDI_AX_SEL_AUDI





(Kein Bild vorhanden)

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein `AUDI_AX_SEL_AUDI` dient als binärer Selektor (Auswahlschalter) für unidirektionale Kommunikationsadapter. Er ermöglicht es, basierend auf dem Zustand eines Steuersignals an einem Selektor-Adapter, den Daten- und Ereignisstrom zwischen zwei alternativen Eingangs-Adaptern auf einen einzigen Ausgangs-Adapter umzuleiten.

Der Baustein kapselt die dafür notwendige Synchronisations- und Konvertierungslogik, um eine saubere Trennung von Ereignissen und Datenflüssen innerhalb von IEC 61499 Anwendungen zu gewährleisten.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine direkten Ereignis-Eingänge auf der Bausteinoberfläche definiert. Die Ereignisverarbeitung erfolgt vollständig gekapselt über die angeschlossenen Adapter.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge auf der Bausteinoberfläche definiert. Die Ereignisweiterleitung erfolgt vollständig über den Ausgangs-Adapter.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Anschlüsse)

- **IN0** (Typ: `adapter::types::unidirectional::AUDI`): Erster auswählbarer Eingangs-Adapter. Die dort ankommenden Daten und Ereignisse werden an den Ausgang durchgereicht, wenn das Auswahlsignal `G` den logischen Zustand `FALSE` (bzw. `0`) besitzt.
- **IN1** (Typ: `adapter::types::unidirectional::AUDI`): Zweiter auswählbarer Eingangs-Adapter. Die dort ankommenden Daten und Ereignisse werden an den Ausgang durchgereicht, wenn das Auswahlsignal `G` den logischen Zustand `TRUE` (bzw. `1`) besitzt.
- **G** (Typ: `adapter::types::unidirectional::AX`): Selektor-Adapter. Das Signal auf diesem Adapter bestimmt, welcher der beiden Eingänge (`IN0` oder `IN1`) auf den Ausgang geschaltet wird.

Plugs (Ausgangs-Anschlüsse)

- **OUT** (Typ: `adapter::types::unidirectional::AUDI`): Ausgangs-Adapter. Gibt die Daten und das dazugehörige Triggerereignis des aktuell ausgewählten Eingangs aus.

Funktionsweise

Das interne Verhalten des Bausteins wird durch ein Netzwerk aus Standard-Funktionsbausteinen realisiert:

1. **Ereignis- und Datensynchronisation**: Die über die Sockets eingehenden Signale (`IN0`, `IN1`, `G`) werden zunächst über flanken- bzw. pegelgesteuerte D-Flip-Flops (`E_D_FF` und `E_D_FF_ANY`) geführt. Dies stellt sicher, dass Datenwerte (`D1`) und Ereignisse (`E1`) synchron zueinander verarbeitet werden.
2. **Datenkonvertierung**: Die synchronisierten Datenwerte der beiden Eingänge werden über zwei Transfer-Bausteine (`F_MOVE` mit dem Datentyp `UDINT`) an den Auswahlbaustein übergeben.
3. **Auswahlsteuerung**: Der Baustein `F_SEL` (IEC 61131-3 Selection) führt die eigentliche logische Umschaltung durch:
   - Ist das Steuersignal des Adapters `G` auf `0` (`FALSE`), wird der Wert von `IN0` gewählt.
   - Ist das Steuersignal des Adapters `G` auf `1` (`TRUE`), wird der Wert von `IN1` gewählt.
4. **Ausgabe**: Das selektierte Ergebnis wird über einen weiteren Transfer-Baustein (`F_MOVE_OUT`) an das Ausgangs-Flip-Flop `E_D_FF_ANY_OUT` übertragen. Dieses generiert das Ausgangsereignis `OUT.E1` und stellt den selektierten Datenwert an `OUT.D1` bereit.

Technische Besonderheiten

- **UDINT-Datenverarbeitung**: Obwohl die Adapter vom generischen Typ `AUDI` sind, werden die Nutzdaten (`D1`) intern explizit als `UDINT` (Unsigned Double Integer / 32-Bit Ganzzahl) verarbeitet und übertragen.
- **Ereignisgesteuertes Verhalten**: Jede Änderung an den Eingängen oder am Selektor führt über das interne Koppelnetzwerk zu einer Neuberechnung und triggert das Ausgangsereignis des Adapters `OUT`.
- **Kapselung**: Durch die Verwendung von Adaptern statt loser Event- und Daten-Ports wird der Verdrahtungsaufwand im übergeordneten Systemdiagramm extrem minimiert.

Zustandsübersicht

Da der Baustein als FB-Netzwerk aufgebaut ist, besitzt er keine klassische Zustandsmaschine (ECC). Das logische Verhalten lässt sich stattdessen über folgende Funktionstabelle beschreiben:

Zustand Selektor (`G.D1`)	Triggerndes Ereignis	Aktiver Datenpfad	Wert am Ausgang (`OUT.D1`)	Ereignis am Ausgang (`OUT.E1`)
`FALSE`	Event an `IN0.E1` oder `G.E1`	`IN0 -> OUT`	Entspricht `IN0.D1`	Wird ausgelöst
`TRUE`	Event an `IN1.E1` oder `G.E1`	`IN1 -> OUT`	Entspricht `IN1.D1`	Wird ausgelöst

---

Anwendungsszenarien

- **Sollwert-Umschaltung**: Wechsel zwischen einem Automatik-Sollwert (z.B. von einem PID-Regler an `IN1`) und einem Hand-Sollwert (z.B. von einer Visualisierung an `IN0`) über ein Auswahlsignal an `G`.
- **Sensor-Redundanz**: Ausfallsichere Umschaltung zwischen einem Hauptsensor und einem Ersatzsensor bei Signalstörungen.
- **Rezeptursteuerung**: Auswahl unterschiedlicher vordefinierter Parameter-Profile im laufenden Betrieb.

Vergleich mit ähnlichen Bausteinen

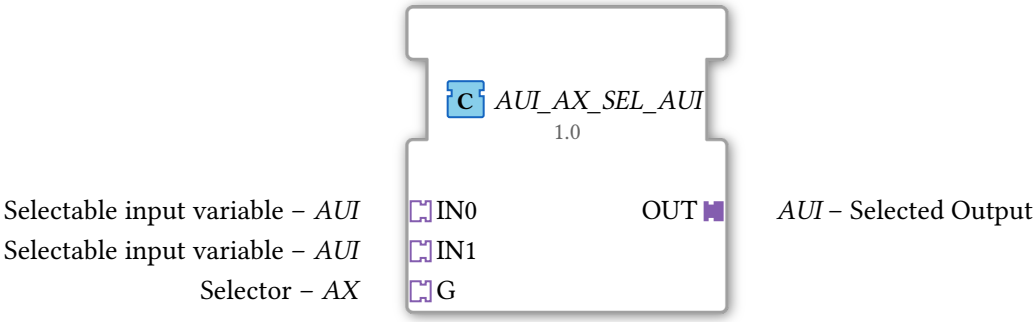
Im Vergleich zum Standard-Auswahlbaustein `F_SEL` aus der IEC 61131-3 Bibliothek bietet `AUDI_AX_SEL_AUDI` den Vorteil, dass er direkt mit strukturierten, unidirektionalen Adaptern (`AUDI` und `AX`) interagieren kann. Bei der Verwendung des Standard-`F_SEL`-Bausteins müssten alle Event- und Datenlei-

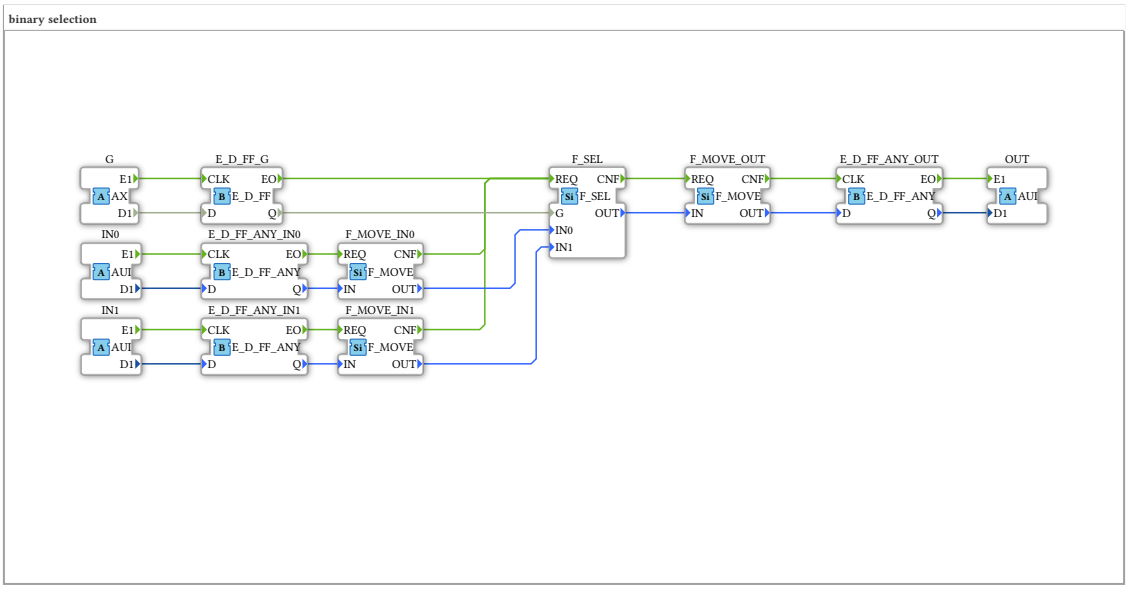
tungen der Adapter manuell aufgesplittet, einzeln verschaltet und anschließend wieder zusammengeführt werden, was fehleranfällig ist und das Anwendungsdiagramm unübersichtlich macht.

Fazit

Der Baustein `AUDI_AX_SEL_AUDI` bietet eine elegante, saubere und wiederverwendbare Lösung für die Signalumschaltung in 4diac-basierten Steuerungsarchitekturen. Durch die konsequente Nutzung von Adaptern trägt er maßgeblich zur Modularität und Übersichtlichkeit komplexer Steuerungsprogramme bei.

AUI_AX_SEL_AUI





Kein Bild verfügbar

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein `AUI_AX_SEL_AUI` ist ein zusammengesetzter Funktionsbaustein (Composite Function Block) für 4diac-ide. Er dient der binären Auswahl (Selektion) zwischen zwei analogen bzw. numerischen Eingangssignalen, die über Adapter bereitgestellt werden. Basierend auf dem Zustand eines Selektions-Adapters schaltet der Baustein eines der beiden Eingangssignale auf den Ausgang auf.

Schnittstellenstruktur

Der Funktionsbaustein verfügt über keine klassischen, direkten Ereignis- oder Datenkanäle an seiner Hauptoberfläche. Die gesamte Kommunikation wird über Adapter abgewickelt.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge
Keine direkten Ereignis-Ausgänge vorhanden.

Daten-Eingänge
Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge
Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Adapter)
- **IN0** (Typ: `adapter::types::unidirectional::AUI`): Unidirektionaler Adapter für das erste auswählbare Eingangssignal. Dieses Signal wird an den Ausgang weitergeleitet, wenn der Selektor `G` den Wert `FALSE` aufweist.
- **IN1** (Typ: `adapter::types::unidirectional::AUI`): Unidirektionaler Adapter für das zweite auswählbare Eingangssignal. Dieses Signal wird an den Ausgang weitergeleitet, wenn der Selektor `G` den Wert `TRUE` aufweist.
- **G** (Typ: `adapter::types::unidirectional::AX`): Unidirektionaler Selektor-Adapter. Bestimmt, welches der beiden Eingangssignale durchgeschaltet wird.

Plugs (Ausgangs-Adapter)
- **OUT** (Typ: `adapter::types::unidirectional::AUI`): Unidirektionaler Adapter für das ausgewählte Ausgangssignal.

Funktionsweise

Im Inneren des Bausteins befindet sich ein definiertes Netzwerk aus Standard-Funktionsbausteinen, das die Adapter-Signale verarbeitet:

1. **Pufferung & Synchronisation:** Die über die Adapter `IN0`, `IN1` und `G` eintreffenden Ereignisse (`E1`) und Daten (`D1`) werden mithilfe von Ereignis-Daten-Flipflops (`E_D_FF` und `E_D_FF_ANY`) zwischengespeichert und synchronisiert.
2. **Datenkonvertierung:** Über interne `F_MOVE`-Bausteine (konfiguriert auf den Datentyp `UINT`) werden die gepufferten numerischen Werte der Eingänge an den eigentlichen Auswahl-Baustein weitergeleitet.
3. **Selektion:** Der Standard-Auswahlbaustein `F_SEL` (gemäß IEC 61131-3) führt die eigentliche Auswahl durch:
   - Ist der Wert am Selektor `G` gleich `FALSE`, wird das Signal von `IN0` gewählt.
   - Ist der Wert am Selektor `G` gleich `TRUE`, wird das Signal von `IN1` gewählt.
4. **Ausgabe:** Das Ergebnis der Selektion wird über einen weiteren `F_MOVE`- und `E_D_FF_ANY`-Puffer an den Ausgangs-Plug `OUT` übertragen. Gleichzeitig wird das Ausgangsereignis am Plug ausgelöst, um nachfolgende Bausteine über den neuen Wert zu informieren.

Technische Besonderheiten
- **Reine Adapter-Schnittstelle:** Der Baustein ist speziell für die Verwendung in modernen, adapterbasierten Architekturen konzipiert. Dies sorgt für ein sehr cleanes und übersichtliches Applikationsdiagramm, da lose Daten- und Event-Leitungen in Adaptern gebündelt werden.
- **Ereignisgesteuert:** Jede Änderung (Senden eines Ereignisses) an den Eingängen `IN0`, `IN1` oder dem Selektor `G` triggert eine sofortige Neuberechnung und führt zu einer Aktualisierung des Ausgangs `OUT`.
- **Typisierung:** Die internen Konvertierungs- und Weiterleitungsbausteine arbeiten primär mit dem Datentyp `UINT`.

Zustandsübersicht

Da es sich um einen Composite-FB handelt, besitzt der Baustein keine eigene Zustandsmaschine (ECC). Das Verhalten ist rein daten- und ereignisgesteuert:

Zustand Selektor `G` (Wert von D1)	Ausgang `OUT` (Wert von D1)	Triggerung von `OUT.E1`

FALSE (0)	Entspricht dem Wert von IN0	Ja, bei Änderung/Ereignis an IN0 oder G
TRUE (1)	Entspricht dem Wert von IN1	Ja, bei Änderung/Ereignis an IN1 oder G

Anwendungsszenarien

- **Sollwert-Umschaltung:** Wechseln zwischen einem manuell vorgegebenen Sollwert (z. B. über ein HMI an IN0) und einem Automatik-Sollwert (z. B. aus einer Berechnungslogik an IN1).
- **Sensorredundanz:** Umschalten zwischen zwei analogen Messwertgebern (z. B. Primärsensor an IN1 und Backup-Sensor an IN0) im Fehlerfall über ein Steuersignal an G.
- **Betriebsartenwahl:** Auswahl unterschiedlicher analoger Steuerparameter in Abhängigkeit vom aktuellen Maschinenzustand.

Vergleich mit ähnlichen Bausteinen

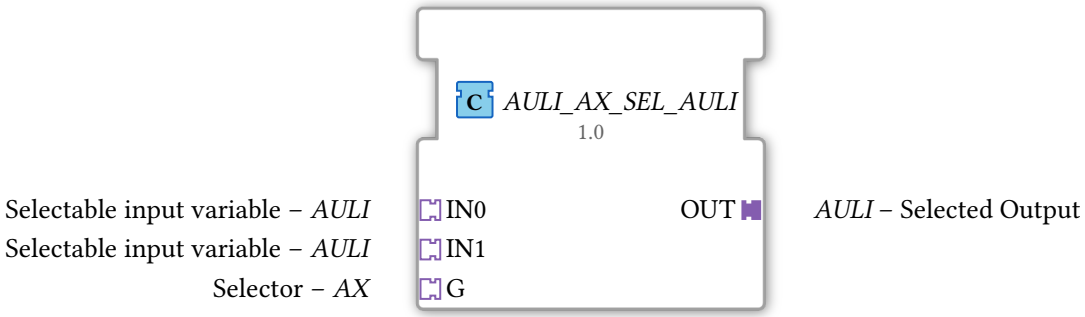
Der klassische Standard-Baustein F_SEL benötigt direkte Datenleitungen (z. B. ANY_ELEMENTARY) und eine manuelle Event-Verkabelung für die Berechnungstriggerung (REQ/CNF).

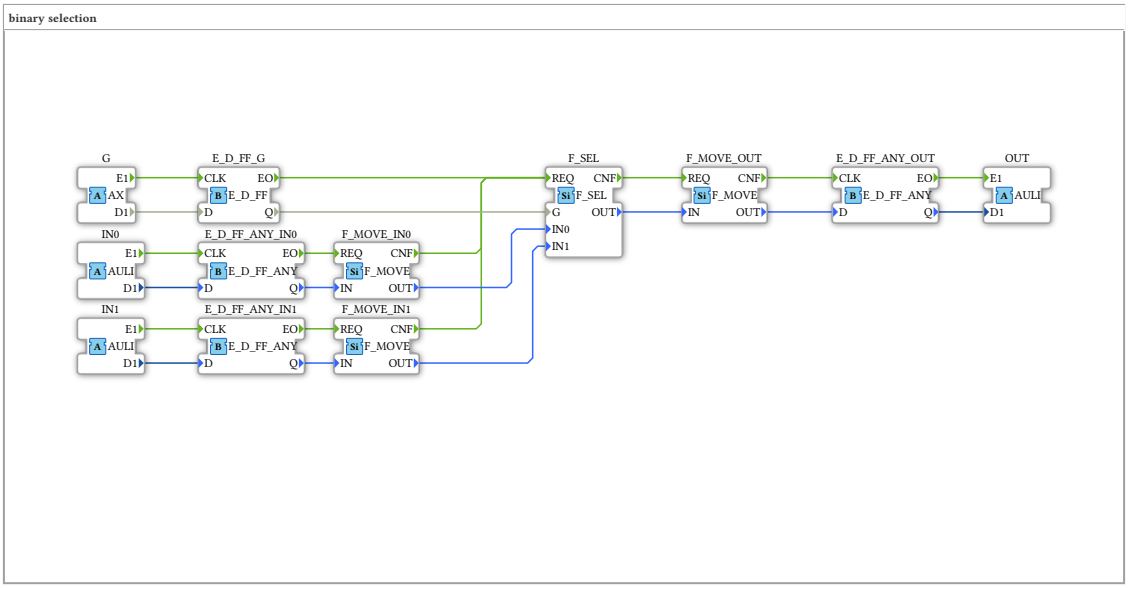
Der Baustein AUI_AX_SEL_AUI kapselt diese komplette Logik inklusive der Event-Synchronisation und Pufferung. Er eignet sich daher ideal für modularisierte Software-Architekturen, bei denen Signalgruppen bereits standardmäßig als Adapter vorliegen.

Fazit

AUI_AX_SEL_AUI vereinfacht das Signalrouting in 4diac-Anwendungen erheblich. Er kombiniert die bewährte Auswahl-Logik der IEC 61131-3 mit den strukturellen Vorteilen ereignisgesteuerter Adapter in der IEC 61499.

AULI_AX_SEL_AULI





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein `AULI_AX_SEL_AULI` dient als binärer Selektor (Multiplexer) im 4diac-Framework. Er ermöglicht es, basierend auf einem Steuersignal eines Selektor-Adapters (`G`), zwischen zwei Eingangssignalen von Adapter-Eingängen (`IN0` und `IN1`) auszuwählen und das ausgewählte Signal an einen Ausgangs-Adapter (`OUT`) weiterzuleiten.

Die Besonderheit dieses Bausteins liegt in der konsequenten Verwendung von unidirektionalen Adaptern zur Kapselung von Ereignissen und Daten, was zu einer aufgeräumten und modularen Struktur im übergeordneten Anwendungsnetzwerk führt.

Schnittstellenstruktur

Da dieser Funktionsbaustein vollständig auf Adaptern basiert, besitzt er keine direkten, klassischen Ereignis- oder Datenkanäle auf seiner äußeren Grenze. Die gesamte Kommunikation ist in den Adaptern gekapselt.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden. Die Ereignissteuerung erfolgt implizit über die Eingangsadapter.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden. Die Ereignisausgabe erfolgt implizit über den Ausgangsadapter.

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangsadapter)

- **G** (Typ: `adapter::types::unidirectional::AX`): Der Selektor-Kanal. Das hier anliegende Signal steuert, welcher der beiden Eingänge auf den Ausgang durchgeschaltet wird.
- **IN0** (Typ: `adapter::types::unidirectional::AULI`): Der erste auswählbare Eingangskanal. Dieser Wert wird an den Ausgang weitergegeben, wenn der Selektor `G` den Zustand `FALSE` (0) aufweist.
- **IN1** (Typ: `adapter::types::unidirectional::AULI`): Der zweite auswählbare Eingangskanal. Dieser Wert wird an den Ausgang weitergegeben, wenn der Selektor `G` den Zustand `TRUE` (1) aufweist.

Plugs (Ausgangsadapter)

- **OUT** (Typ: `adapter::types::unidirectional::AULI`): Der ausgewählte Ausgangskanal, welcher das Signal des ausgewählten Eingangs (`IN0` oder `IN1`) führt.

Funktionsweise

Intern basiert der Baustein auf einem Netzwerk aus Standard-Funktionsbausteinen der IEC 61131 und IEC 61499:

1. **Ereigniserfassung:** Sobald an einem der Eingänge (`IN0.E1`, `IN1.E1`) oder am Selektor (`G.E1`) ein Ereignis eintrifft, wird dieses über flankengetriggerte Ereignis-Flip-Flops (`E_D_FF` bzw. `E_D_FF_ANY`) abgefangen und der jeweils aktuelle Datenwert (`D1`) zwischengespeichert.
2. **Datenkonvertierung & Transport:** Die Datenwerte der Eingänge werden über Konvertierungsbausteine (`F_MOVE` mit dem Datentyp `ULINT`) an den zentralen Auswahlbaustein übergeben.
3. **Auswahllogik (Multiplexing):** Der Baustein `F_SEL` (Standard Selection) wertet den Zustand des Selektors aus:
   - Ist der Wert von `G` gleich `FALSE`, wird der Wert von `IN0` gewählt.
   - Ist der Wert von `G` gleich `TRUE`, wird der Wert von `IN1` gewählt.
4. **Ausgabe:** Nach erfolgreicher Selektion wird das Ergebnis über einen weiteren Konvertierungsblock (`F_MOVE_OUT`) an das Ausgangs-Flip-Flop (`E_D_FF_ANY_OUT`) gesendet. Dieses triggert das Ausgangsereignis `OUT.E1` und legt den ausgewählten Wert an `OUT.D1` an.

Technische Besonderheiten

- **Datentyp-Spezifität:** Intern arbeitet die Selektion auf dem Datentyp `ULINT` (Unsigned Long Integer / 64-Bit). Dies macht den Baustein besonders geeignet für die Verarbeitung von hochauflösenden Zählerwerten, Zeitstempeln oder großen ID-Nummern.
- **Adapter-Kapselung:** Durch die Verwendung von Adaptern (`AULI` und `AX`) werden Signal- und Eventleitungen gebündelt. Dies verhindert „Spaghetti-Code“ im Systementwurf.
- **Event-Driven:** Der Baustein arbeitet rein ereignisgesteuert. Es erfolgt keine zyklische Berechnung, was Ressourcen schont.

Zustandsübersicht

Der Baustein besitzt keine eigene explizite Zustandskette (Execution Control Chart, ECC), da er als reines Netzwerk (FBNetwork) realisiert ist. Sein Verhalten ist rein daten- und ereignisflussgesteuert:

Trigger-Ereignis	Zustand Selektor (`G.D1`)	Resultierendes Verhalten	Ausgangsereignis
Event an `G.E1`, `IN0.E1` oder `IN1.E1`	`FALSE`	Wert von `IN0.D1` wird nach `OUT.D1` kopiert	`OUT.E1`
Event an `G.E1`, `IN0.E1` oder `IN1.E1`	`TRUE`	Wert von `IN1.D1` wird nach `OUT.D1` kopiert	`OUT.E1`

Anwendungsszenarien

- **Sensor-Redundanz:** Umschalten zwischen einem Hauptsensor (`IN0`) und einem Backup-Sensor (`IN1`) basierend auf einem Statussignal (`G`).
- **Sollwert-Umschaltung:** Wechseln zwischen einem automatischen Sollwert (z.B. aus einer Rezepturesteuerung) und einem manuell vorgegebenen Benutzer-Sollwert.
- **Betriebsarten-Umschaltung:** Routing von unterschiedlichen Datenströmen je nach aktivem Anlagenstatus (z. B. Normalbetrieb vs. Wartungsmodus).

Vergleich mit ähnlichen Bausteinen

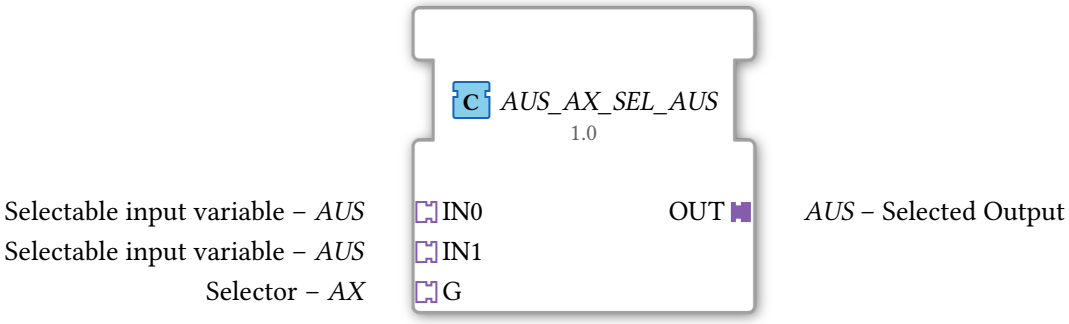
Im Vergleich zum klassischen Standard-Auswahlbaustein `F_SEL` bietet `AULI_AX_SEL_AULI` folgende Vorteile:

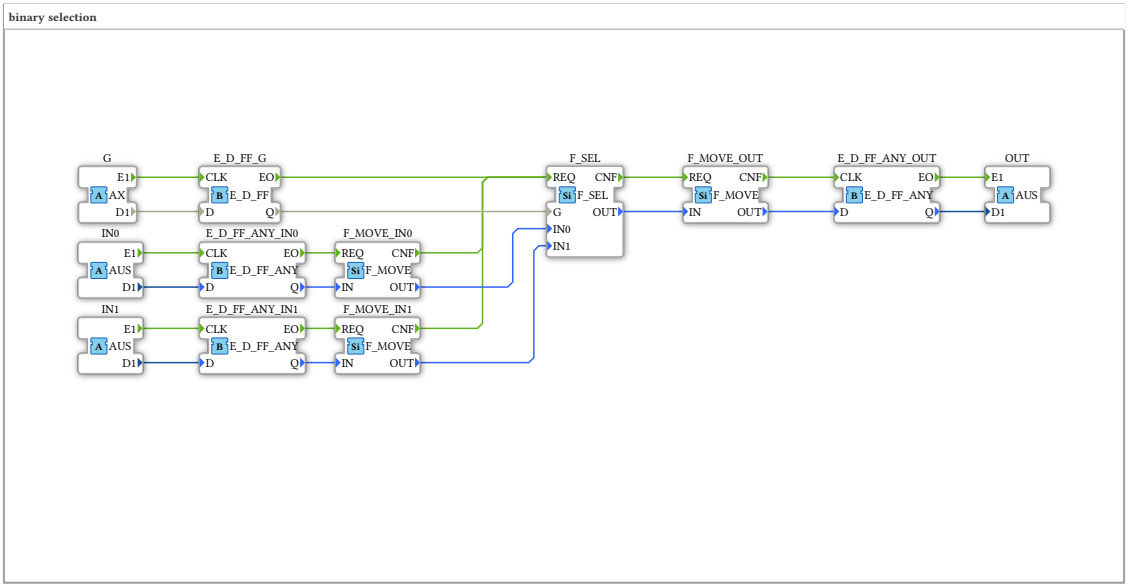
- **Kein manuelles Verdrahten von Events:** Bei `F_SEL` müssen Daten- und Eventleitungen manuell synchronisiert werden. Hier geschieht dies automatisch im Inneren des Bausteins durch die Verwendung der Adapter.
- **Erhöhte Typsicherheit & Übersichtlichkeit:** Verbindungsfehler im übergeordneten Steuerungsdiagramm werden minimiert, da Adapter nur mit passenden Gegenstücken verbunden werden können.

Fazit

Der `AULI_AX_SEL_AULI` ist ein robuster, wiederverwendbarer und ereignisgesteuerter Selektorbaustein für 64-Bit-Ganzzahlwerte. Er eignet sich hervorragend für moderne, komponentenbasierte Softwarearchitekturen in der IEC 61499, bei denen Übersichtlichkeit und standardisierte Schnittstellen im Vordergrund stehen.

AUS_AX_SEL_AUS





(Kein Bild verfügbar)

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **AUS_AX_SEL_AUS** ist ein binärer Selektor (Multiplexer) für IEC 61499-Systeme. Er dient dazu, basierend auf dem Zustand eines Auswahlsignals (Selector `G`), zwischen zwei Eingangsadaptern (`IN0` und `IN1`) zu wählen und den ausgewählten Wert an den Ausgangsadapter (`OUT`) weiterzuleiten.

Der Baustein nutzt unidirektionale Adapterstrukturen, was eine saubere Kapselung von Daten- und Ereignisströmen ermöglicht und das visuelle Routing innerhalb der Entwicklungsumgebung vereinfacht.

Schnittstellenstruktur

Da dieser Funktionsbaustein als Composite-Netzwerk realisiert ist und vollständig auf Adaptern basiert, besitzt er keine direkten, klassischen Ereignis- oder Datenkanäle auf seiner Hauptebene. Die gesamte Kommunikation ist in den Adaptern gebündelt.

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge vorhanden (Ereignisse werden über die Adapter-Schnittstellen empfangen).

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden (Ereignisse werden über die Adapter-Schnittstellen gesendet).

Daten-Eingänge

Keine direkten Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine direkten Daten-Ausgänge vorhanden.

Adapter

Sockets (Eingangs-Adapter)

- **G** (Typ: `adapter::types::unidirectional::AX`):
  - **Beschreibung:** Auswahlsignal (Selector).
  - **Inhalt:** Enthält das Auswahlerleignis `E1` und das dazugehörige Datensignal `D1` (üblicherweise boolesch interpretiert), das bestimmt, welcher Eingang durchgeschaltet wird.
- **IN0** (Typ: `adapter::types::unidirectional::AUS`):
  - **Beschreibung:** Erster auswählbarer Eingang. Wird an den Ausgang weitergeleitet, wenn der Selektor `G` den Zustand `FALSE` (bzw. `0`) aufweist.
  - **Inhalt:** Ereignis `E1` und Datensignal `D1` (Daten-Typ: `USINT`).
- **IN1** (Typ: `adapter::types::unidirectional::AUS`):
  - **Beschreibung:** Zweiter auswählbarer Eingang. Wird an den Ausgang weitergeleitet, wenn der Selektor `G` den Zustand `TRUE` (bzw. `1`) aufweist.
  - **Inhalt:** Ereignis `E1` und Datensignal `D1` (Daten-Typ: `USINT`).

Plugs (Ausgangs-Adapter)

- **OUT** (Typ: `adapter::types::unidirectional::AUS`):
  - **Beschreibung:** Der ausgewählte Ausgang. Erhält die Daten und Ereignisse des jeweils durchgeschalteten Eingangs.
  - **Inhalt:** Ereignis `E1` und Datensignal `D1` (Daten-Typ: `USINT`).

Funktionsweise

Der Baustein kapsele ein internes Netzwerk, welches die Adapter-Signale extrahiert, verarbeitet und über standardisierte IEC 61131-3- und IEC 61499-Bausteine routet:

1. **Datenerfassung:** Sobald an einem der Eingangs-Adapter (`G`, `IN0` oder `IN1`) ein Ereignis (`E1`) eintrifft, wird der entsprechende Datenwert (`D1`) mithilfe von Flip-Flops (`E_D_FF` bzw. `E_D_FF_ANY`) zwischengespeichert.
2. **Wertübergabe:** Die Werte der Eingänge `IN0` und `IN1` werden über Kopiervorgänge (`F_MOVE` mit dem Datentyp `USINT`) an den zentralen Selektionsbaustein weitergegeben.
3. **Selektion:** Der interne Baustein `F_SEL` (standardmäßiger IEC 61131-3 Selektor) entscheidet anhand des Zustands von `G.D1`:
   - Ist `G.D1` = `0` (bzw. `FALSE`), wird der Wert von `IN0` auf den Ausgang gelegt.
   - Ist `G.D1` = `1` (bzw. `TRUE`), wird der Wert von `IN1` auf den Ausgang gelegt.
4. **Ausgabe:** Das Ergebnis wird über ein weiteres Flip-Flop (`E_D_FF_ANY_OUT`) an den Ausgangs-Adapter `OUT` übergeben und triggert dort das Ereignis `OUT.E1` zeitgleich mit der Bereitstellung der Daten auf `OUT.D1`.

Technische Besonderheiten

- **Ereignisgesteuert:** Der Ausgang wird bei jeder Änderung an den Eingängen oder des Selektors sofort aktualisiert und mit einem neuen Ereignis versehen.
- **Datentyp-Spezifisch:** Intern ist die Verarbeitung der Datenkanäle auf den Typ `USINT` ausgelegt (festgelegt über die Attribute der internen `F_MOVE`-Instanzen).
- **Adapter-Kapselung:** Erleichtert das saubere Design von Applikationsdiagrammen, da komplexe Daten- und Event-Paare als eine einzige Verbindung (Bus) dargestellt werden.

Zustandsübersicht

Da es sich um einen Composite-Funktionsbaustein ohne eigene Execution Control Chart (ECC) handelt, wird das Verhalten rein durch den Signalfluss bestimmt:

Zustand Selektor (`G.D1`)	Trigger-Ereignis	Aktivität am Ausgang (`OUT`)

`0` / `FALSE`	Beliebig (`G.E1`, `IN0.E1`, `IN1.E1`)	Gibt den Wert von `IN0.D1` aus; triggert `OUT.E1`.
`1` / `TRUE`	Beliebig (`G.E1`, `IN0.E1`, `IN1.E1`)	Gibt den Wert von `IN1.D1` aus; triggert `OUT.E1`.

Anwendungsszenarien

- **Sollwert-Umschaltung:** Dynamisches Umschalten einer Zielgröße (z. B. einer Drehzahl oder Stufe vom Typ `USINT`) zwischen einem Automatikbetrieb (`IN1`) und einem Handbetrieb (`IN0`).
- **Rezepturwahl:** Auswahl zwischen zwei fest vorgegebenen Betriebsmodi oder Parametersätzen in einer Maschine über eine übergeordnete Steuerung.
- **Fehlersicherungen:** Schnelles Umschalten auf einen Ersatzwert (`IN0`), falls ein Hauptsignalweg gestört ist.

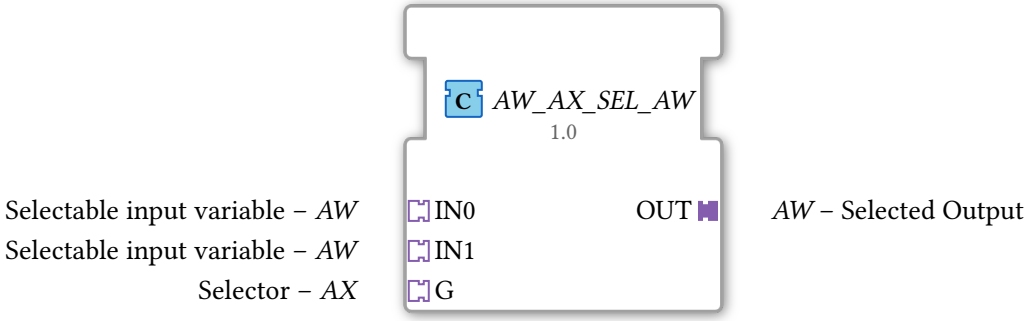
Vergleich mit ähnlichen Bausteinen

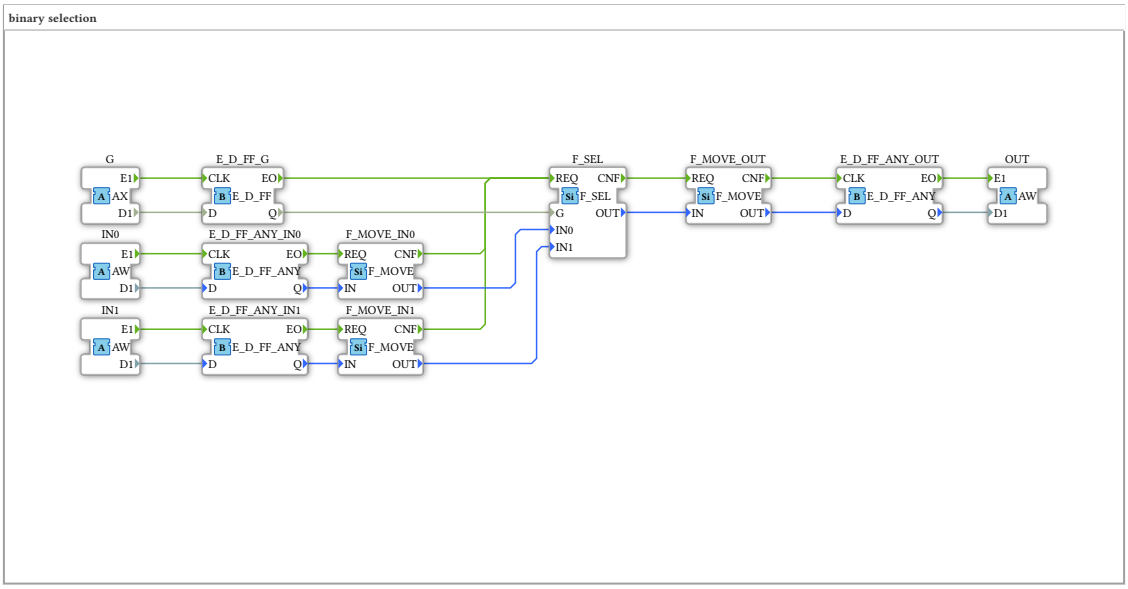
Im Vergleich zum Standard-Auswahlbaustein `F_SEL` aus der IEC 61131-3 Bibliothek bietet `AUS_AX_SEL_AUS` den entscheidenden Vorteil, direkt mit den herstellerspezifischen oder standardisierten unidirektionalen Adaptern (`AUS` und `AX`) arbeiten zu können. Während bei `F_SEL` Ereignis- und Datenleitungen in IEC 61499 mühsam einzeln verdrahtet werden müssen, geschieht dies hier vollautomatisch im Inneren des Blocks.

Fazit

Der Baustein **AUS_AX_SEL_AUS** stellt eine wichtige Brücke für moderne, adapterbasierte IEC 61499-Architekturen dar. Er vereint die bewährte Selektionslogik der IEC 61131-3 mit den Vorteilen der ereignisgesteuerten Adapterkopplung und trägt so signifikant zur Übersichtlichkeit und Wartbarkeit von Steuerungssoftware bei.

AW_AX_SEL_AW





(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `AW_AX_SEL_AW` dient der binären Auswahl (Selektion) zwischen zwei analogen oder wortbasierten Eingangssignalen über einen binären Selektor. Der Baustein ist speziell für die Verwendung mit adapterbasierten Schnittstellen in 4diac-ide (IEC 61499) konzipiert. Er kapselt die klassische Selektionslogik (`SEL`) und stellt diese über unidirektionale Adapter für `WORD`-Daten (`AW`) und binäre Steuersignale (`AX`) zur Verfügung.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine direkten Ereignis-Eingänge auf Bausteinebene vorhanden. Die Ereignisverarbeitung erfolgt vollständig gekapselt über die Adapter-Schnittstellen.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge auf Bausteinebene vorhanden. Die Ereignisausgabe erfolgt vollständig gekapselt über die Adapter-Schnittstellen.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge auf Bausteinebene vorhanden.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge auf Bausteinebene vorhanden.

Adapter

Sockets (Eingang-Schnittstellen)

- **G** (Typ: `adapter::types::unidirectional::AX`):
  - Selektions-Eingang (Selector). Bestimmt, welcher der beiden Eingänge auf den Ausgang durchgeschaltet wird.
- **IN0** (Typ: `adapter::types::unidirectional::AW`):
  - Erster auswählbarer Eingangskanal (wird selektiert, wenn `G` den Zustand `FALSE` bzw. `0` hat).
- **IN1** (Typ: `adapter::types::unidirectional::AW`):
  - Zweiter auswählbarer Eingangskanal (wird selektiert, wenn `G` den Zustand `TRUE` bzw. `1` hat).

Plugs (Ausgang-Schnittstellen)

- **OUT** (Typ: `adapter::types::unidirectional::AW`):
  - Selektierter Ausgangskanal, der den Wert des ausgewählten Eingangs führt.

Funktionsweise

Der Baustein basiert intern auf einem Netzwerk aus Standard-Funktionsbausteinen, um die ereignisgesteuerte Selektion zu realisieren:

1. **Signalpufferung:** Die über die Adapter `IN0`, `IN1` und `G` eingehenden Ereignisse (`E1`) und Daten (`D1`) werden über D-Flipflops (`E_D_FF` bzw. `E_D_FF_ANY`) zwischengespeichert. Dies stellt sicher, dass die Daten konsistent zur Verfügung stehen, sobald ein Ereignis eintrifft.
2. **Datenweiterleitung:** Die Daten aus den WORD-Adaptern werden über `F_MOVE`-Bausteine (konfiguriert auf den Datentyp `WORD`) an den zentralen Auswahlbaustein übergeben.
3. **Auswahllogik (F_SEL):** Der Baustein `F_SEL` (Standard-Auswahlfunktion) wertet den Zustand des Selektors `G` aus:
   - Ist `G.Q` = `FALSE`, wird der Wert von `IN0` an den Ausgang weitergegeben.
   - Ist `G.Q` = `TRUE`, wird der Wert von `IN1` an den Ausgang weitergegeben.
4. **Ausgabe-Triggerung:** Jede Änderung an den Eingängen oder dem Selektor triggert über die Event-Verbindungen den Durchlauf. Der selektierte Wert wird über den Ausgangs-Zwischespeicher `E_D_FF_ANY_OUT` an den Ausgangs-Plug `OUT` übergeben und das zugehörige Ausgangsereignis `OUT.E1` ausgelöst.

Technische Besonderheiten

- **Adapter-Kapselung:** Durch die ausschließliche Verwendung von Adaptern wird der Verkabelungsaufwand im übergeordneten System drastisch reduziert, da Daten- und Eventleitungen in einem einzigen Adapterkanal gebündelt sind.
- **Unidirektionale Datenübertragung:** Die verwendeten Adaptertypen `AW` (Adapter Word) und `AX` (Adapter Bit/Boolean) sind unidirektional aufgebaut, was eine klare Signalflussrichtung definiert.
- **Datentyp:** Die interne Verarbeitung der analogen Werte ist fest auf den Datentyp `WORD` ausgelegt.

Zustandsübersicht

Da es sich um einen zusammengesetzten Funktionsbaustein (Composite FB) ohne eigene State-Machine (ECC) handelt, wird das Verhalten direkt durch den Daten- und Ereignisfluss der internen Komponenten bestimmt:

- **Ereignis auf G.E1:** Der Selektorwert wird aktualisiert. Die Auswahllogik schaltet entsprechend um und aktualisiert den Ausgang `OUT`.
- **Ereignis auf IN0.E1 / IN1.E1:** Die Eingangsdaten werden aktualisiert. Wenn der geänderte Eingang dem aktuell über `G` selektierten Kanal entspricht, wird der neue Wert sofort an `OUT` weitergeleitet und das Ereignis `OUT.E1` getriggert.

Anwendungsszenarien

- **Umschaltung von Sollwerten:** Wahlweise Vorgabe eines Sollwerts (z. B. Drehzahl, Druck, Temperatur als `WORD` codiert) aus einem Automatikprogramm (`IN1`) oder einer manuellen Bedienung (`IN0`).
- **Signal-Routing:** Dynamische Umleitung von Datenströmen innerhalb modularer Steuerungsarchitekturen.
- **Redundante Sensorik:** Umschalten zwischen einem Hauptsensor und einem Ersatzsensor bei Signalfehlern.

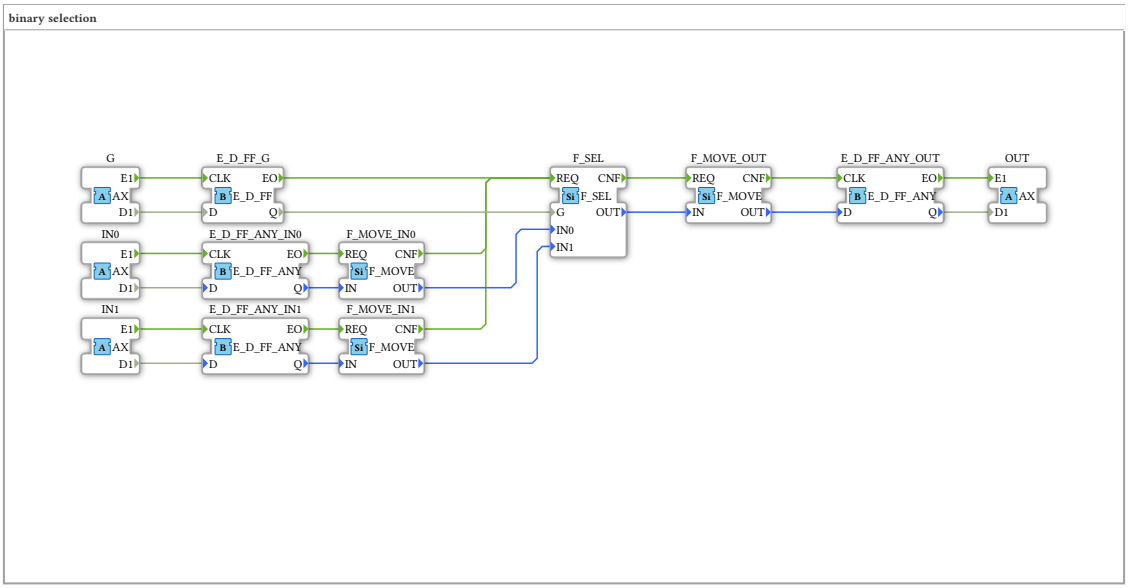
Vergleich mit ähnlichen Bausteinen

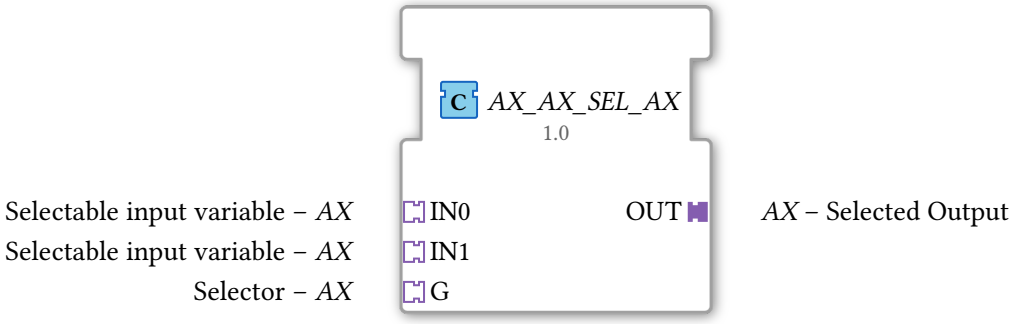
- **Standard `F_SEL` (IEC 61131-3):** Der Standard-Auswahlbaustein arbeitet mit direkten Elementardatentypen (z.B. WORD, BOOL) und besitzt keine integrierte Event-Steuerung für IEC 61499. `AW_AX_SEL_AW` erweitert diese Funktionalität um die Event-Kopplung und verpackt sie in standardisierte Adapter.
- **Multiplexer (MUX):** Ein Multiplexer erlaubt die Auswahl aus mehr als zwei Kanälen über einen ganzzahligen Index (INT). `AW_AX_SEL_AW` ist hingegen für die schnelle, binäre Auswahl (2 Kanäle via BOOL/AX) optimiert.

Fazit

Der `AW_AX_SEL_AW` ist ein robuster und wiederverwendbarer Hilfsbaustein für die strukturierte Programmierung in IEC 61499. Er eignet sich hervorragend für Applikationen, die auf eine saubere Schnittstellenarchitektur mittels Adaptern setzen, und vereinfacht die Signalumschaltung im industriellen Umfeld.

AX_AX_SEL_AX





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein `AX_AX_SEL_AX` dient der binären Auswahl (Selektion) zwischen zwei booleschen Signalen, die über standardisierte Adapter übertragen werden. Er stellt eine IEC 61499-konforme Kapselung der klassischen IEC 61131-3 `SEL`-Funktion dar. Durch die Verwendung von Adaptern werden sowohl die Datenwerte als auch die dazugehörigen Ereignisse gebündelt übertragen, was den Verdrahtungsaufwand im Control-Application-Design erheblich reduziert.

Schnittstellenstruktur

Ereignis-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Eingänge. Die Ereignissteuerung wird vollständig über die Adapter abgewickelt.

Ereignis-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Ereignis-Ausgänge. Die Ereignisausgabe wird vollständig über die Adapter abgewickelt.

Daten-Eingänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Eingänge.

Daten-Ausgänge

Dieser Funktionsbaustein besitzt keine direkten Daten-Ausgänge.

Adapter

Sockets (Eingang-Schnittstellen)

- **IN0** (Typ: `adapter::types::unidirectional::AX`): Auswählbarer Eingangsadapter 0. Der hier anliegende Wert wird an den Ausgang durchgereicht, wenn das Auswahlsignal `G` den Zustand `FALSE` aufweist.
- **IN1** (Typ: `adapter::types::unidirectional::AX`): Auswählbarer Eingangsadapter 1. Der hier anliegende Wert wird an den Ausgang durchgereicht, wenn das Auswahlsignal `G` den Zustand `TRUE` aufweist.
- **G** (Typ: `adapter::types::unidirectional::AX`): Selektor-Adapter. Bestimmt, welcher der beiden Eingänge (`IN0` oder `IN1`) auf den Ausgang geschaltet wird.

Plugs (Ausgang-Schnittstellen)

- **OUT** (Typ: `adapter::types::unidirectional::AX`): Ausgangsadapter, der das ausgewählte Signal (`D1`) und das dazugehörige Bestätigungsereignis (`E1`) ausgibt.

Funktionsweise

Intern basiert der `AX_AX_SEL_AX` auf einem ereignisgesteuerten Netzwerk, welches die Eingangssignale der Sockets verarbeitet und über den Plug ausgibt:

1. **Ereigniserfassung:** Jedes Eintreffen eines Ereignisses (`E1`) an den Adaptern `IN0`, `IN1` oder `G` triggert ein internes flanken-gesteuertes Flip-Flop (`E_D_FF` bzw. `E_D_FF_ANY`).
2. **Datenpufferung & Transport:** Die Datenwerte (`D1`) der Adapter werden über Konvertierungsbausteine des Typs `F_MOVE` (konfiguriert auf den Datentyp `BOOL`) an den zentralen Selektionsbaustein `F_SEL` weitergeleitet.
3. **Auswahllogik:**
   - Ist das Signal am Adapter `G` gleich `FALSE`, schaltet der interne Baustein `F_SEL` den Wert von `IN0` auf den Ausgang.
   - Ist das Signal am Adapter `G` gleich `TRUE`, wird der Wert von `IN1` auf den Ausgang geschaltet.
4. **Ausgabe:** Der ausgewählte Wert wird an das Flip-Flop des Ausgangs-Adapters übergeben, welches das Ereignis `OUT.E1` auslöst und den Datenpunkt `OUT.D1` aktualisiert.

Technische Besonderheiten

- **Typisierung auf BOOL:** Die internen Datenkopierer (`F_MOVE`) sind fest auf den Datentyp `BOOL` parametriert. Daher ist dieser spezifische Baustein für die Auswahl von booleschen Zuständen optimiert.
- **Ereignissynchronisation:** Durch den internen Einsatz von `E_D_FF` und `E_D_FF_ANY` ist sichergestellt, dass Datenänderungen an den Eingängen sofort und ohne Zyklusverzögerung konsistent an den Ausgang übertragen werden.

Zustandsübersicht

Da es sich um einen zusammengesetzten Funktionsbaustein (Composite FB) handelt, wird das Verhalten direkt durch den Daten- und Ereignisfluss bestimmt:

Zustand Selektor `G.D1`	Ausgang `OUT.D1`	Verhalten bei Ereignis an den Eingängen
`FALSE`	Wert von `IN0.D1`	Ereignisse an `IN0.E1` oder `G.E1` triggern eine Aktualisierung von `OUT.E1`.
`TRUE`	Wert von `IN1.D1`	Ereignisse an `IN1.E1` oder `G.E1` triggern eine Aktualisierung von `OUT.E1`.

Anwendungsszenarien

- **Betriebsartenumschaltung:** Umschalten zwischen einem Automatik- und einem Hand-Befehl (z. B. Ventil öffnen/schließen).
- **Signal-Redundanz:** Auswahl zwischen einem primären Steuersignal und einem Backup-Signal im Fehlerfall.
- **Kompakte Applikationserstellung:** Reduzierung von „Spaghetti-Code“ und unübersichtlichen Linienverbindungen in 4diac-ide durch Nutzung der Adapter-Technologie.

Vergleich mit ähnlichen Bausteinen

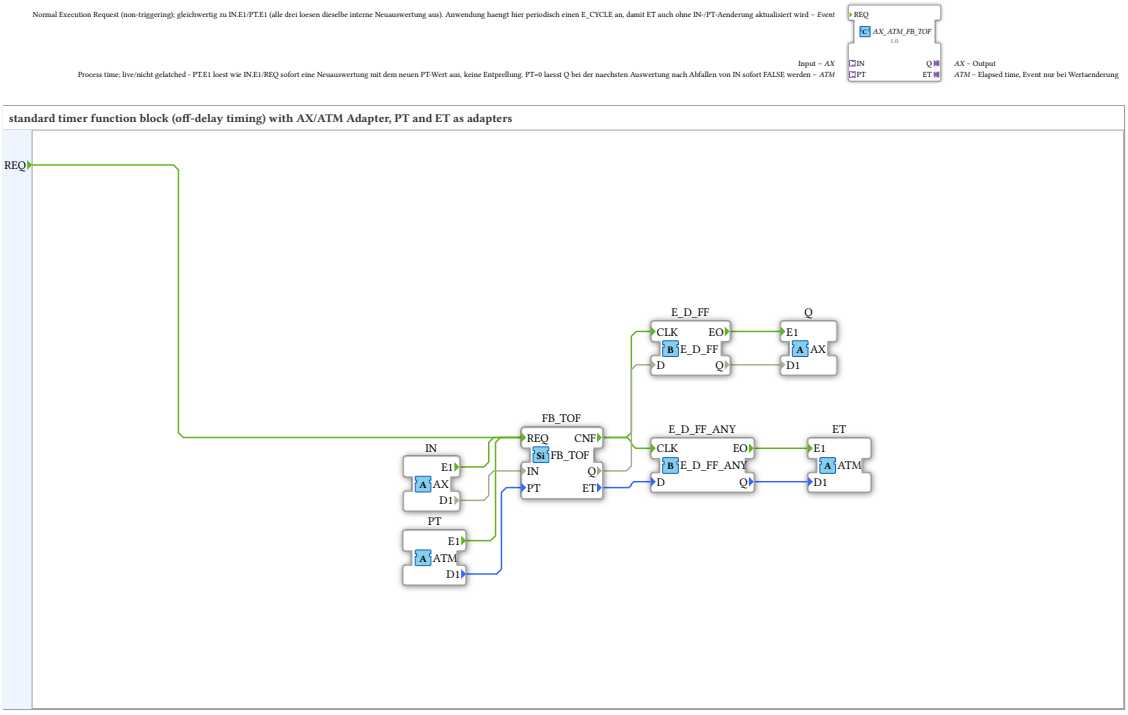
- **Standard `F_SEL`:** Der klassische `F_SEL`-Baustein benötigt separate Event-Verbindungen (`REQ` / `CNF`) und Datenleitungen für jeden Kanal. `AX_AX_SEL_AX` vereinfacht dies durch die Kapselung in drei Eingangs- und einen Ausgangsadapter.
- **Multiplexer (MUX):** Während ein Multiplexer zwischen einer Vielzahl von Kanälen umschalten kann, ist der `AX_AX_SEL_AX` als binärer Selektor auf exakt zwei Kanäle spezialisiert und dadurch ressourcenschonender und einfacher zu beschalten.

Fazit

Der `AX_AX_SEL_AX` ist ein praktischer Hilfsbaustein für die strukturierte Anwendungsentwicklung in der IEC 61499. Er kombiniert die bewährte Selektionslogik der IEC 61131 mit den modernen Vorteilen der adapterbasierten Signalübertragung in der 4diac-ide.

timers

AX_ATM_FB_TOF



• — ○ • — ○ • — ○ •

Einleitung

`AX_ATM_FB_TOF` ist die adapterbasierte Variante von AX_FB_TOF: derselbe IEC-61131-3-Ausschaltverzögerungstimer (`iec61131::timers::FB_TOF`), aber mit `PT` (Sollzeit) als Socket und `ET` (abgelaufene Zeit) als Plug — beide vom Typ ATM — statt als einfache `TIME`-Datenvariablen. Ebenso wie AX_ATM_FB_TON erlaubt das, den Baustein direkt in adapterbasierte Ketten einzuhängen, ohne an jeder Verwendungsstelle eigene Datenverbindungen für `PT`/`ET` zu ziehen.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Event, nicht auslösend): Normaler Ausführungsaufruf. Die Anwendung hängt hier typischerweise einen `E_CYCLE` an, damit `ET` periodisch aktualisiert wird, während der Timer läuft.

Ereignis-Ausgänge

Keine eigenen Ereignis-Ausgänge; Ereignisse laufen ausschließlich über die Adapter-Plugs `Q` und `ET`.

Daten-Eingänge

Keine eigenen Daten-Eingänge. Alle Daten werden über die Adapter-Sockets übertragen.

Daten-Ausgänge

Keine eigenen Daten-Ausgänge. Alle Daten werden über die Adapter-Plugs übertragen.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	`IN`	`adapter::types::unidirectional::AX`	Startsignal
Socket (Eingang)	`PT`	`adapter::types::unidirectional::ATM`	Sollzeit
Plug (Ausgang)	`Q`	`adapter::types::unidirectional::AX`	Ausgang, bleibt TRUE bis `PT` nach dem Abfallen von `IN`
Plug (Ausgang)	`ET`	`adapter::types::unidirectional::ATM`	Abgelaufene Zeit, Event nur bei Wertänderung

Funktionsweise

Intern wird derselbe `FB_TOF`-Standardbaustein wie in `AX_FB_TOF` verwendet, ergänzt um zwei `E_D_FF`/`E_D_FF_ANY`-Puffer:

1. `IN.E1`, das eigene `REQ`-Ereignis und `PT.E1` lösen alle gemeinsam `FB_TOF.REQ` aus.
2. `FB_TOF.Q` läuft über ein `E_D_FF` (Change-Detection) auf `Q.D1`/`Q.E1`.
3. `FB_TOF.ET` läuft über ein `E_D_FF_ANY` (generische Change-Detection) auf `ET.D1`/`ET.E1` — nur bei tatsächlicher Wertänderung, nicht bei jedem `REQ`-Zyklus.

Im Unterschied zum Einschaltverzögerungstimer setzt `IN = TRUE` den Ausgang `Q` sofort auf TRUE; erst wenn `IN` wieder auf FALSE fällt, läuft die Zeit `PT` ab, bevor `Q` auf FALSE zurückfällt.

Technische Besonderheiten

- **`REQ` bleibt bewusst ein einfaches Event, keine Adapterverbindung**, wie bei AX_ATM_FB_TON — die Anwendung bestimmt selbst die Aktualisierungsrate von `ET` über einen extern angeschlossenen `E_CYCLE`.
- **`ET`-Events nur bei Wertänderung**, nach demselben Prinzip wie ASSEMBLE_AB_FROM_AX.
- **`PT` ist live, nicht gelatched — `IN.E1`, `PT.E1` und `REQ` sind für `FB_TOF` gleichwertige Auslöser**, genau wie bei AX_ATM_FB_TON. `PT.D1` liegt als reine Datenverbindung dauerhaft an `FB_TOF.PT` an, keine Pufferung/Entprellung im Adapter:
  - **`PT = 0`**: Nachdem `IN` auf FALSE gefallen ist, wird `Q` bei der nächsten Auswertung sofort FALSE (kein sichtbarer Nachlauf).
  - **`PT` wird während des Nachlaufs verkleinert**, sodass das bereits gelaufene `ET` die neue Sollzeit übersteigt: `Q` wird sofort FALSE.
  - **`PT` wird während des Nachlaufs vergrößert**: `Q` bleibt entsprechend länger TRUE.
  - **Schnell aufeinanderfolgende `PT.E1`-Ereignisse** lösen jeweils eine eigene Auswertung aus, keine Zusammenfassung.
  - **Vor dem allerersten `PT.E1`** liegt an `FB_TOF.PT` der `TIME`-Standardwert `T#0s` an.
- **Identisches Verhalten in AX_ATM_FB_TON und AX_ATM_FB_TP** — alle drei sind strukturell identisch verdrahtet, nur der intern gewrappte Standardbaustein unterscheidet sich.

Zustandsübersicht

1. **Ruhezustand**: `IN` ist FALSE, `Q` ist FALSE, `ET` ist 0.
2. **`IN` wird TRUE**: `Q` wird sofort TRUE, `ET` bleibt 0.
3. **`IN` fällt auf FALSE**: Timer startet, `ET` steigt bis `PT`, `Q` bleibt TRUE.
4. **Abgelaufen**: `ET` hat `PT` erreicht, `Q` wird FALSE.

Anwendungsszenarien

- Nachlaufzeiten, bei denen ein Ausgang nach dem Abschalten eines Eingangs noch eine definierte Zeit lang aktiv bleiben soll, mit sichtbarer Restlaufzeit über `ET`.
- Jede Stelle, an der bereits `AX_FB_TOF` verwendet wird, aber `PT`/`ET` über adapterbasierte Ketten fließen sollen.

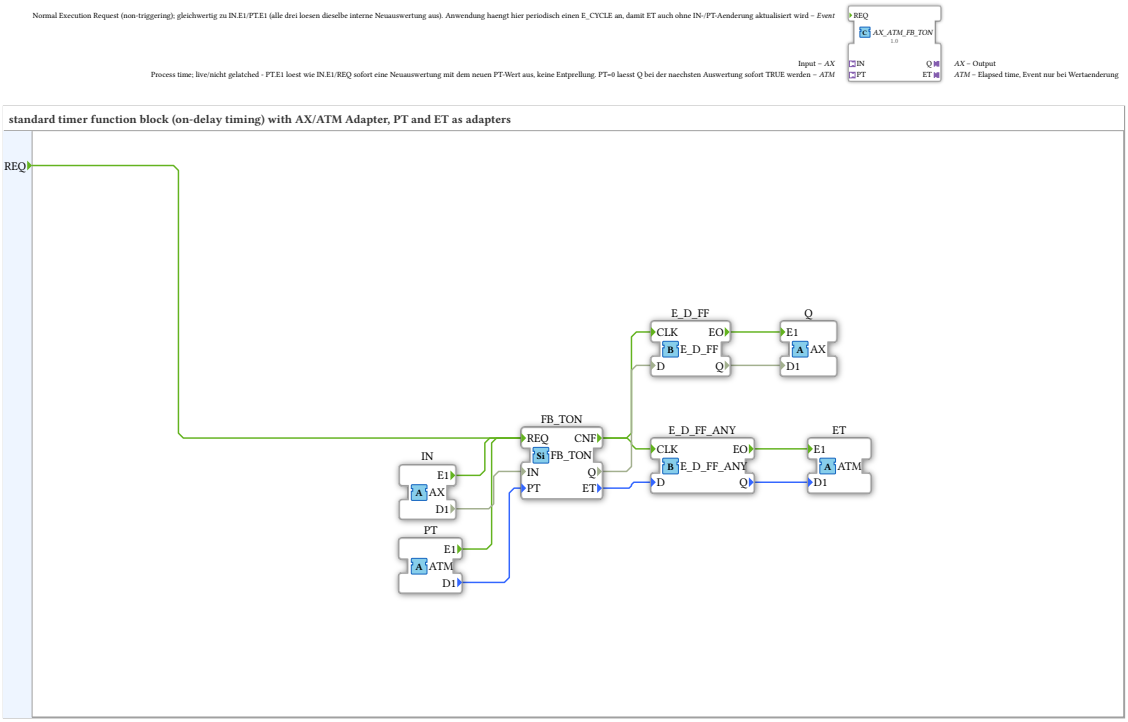
⚖️ Vergleich mit ähnlichen Bausteinen

- **AX_FB_TOF**: identisches Timing-Verhalten, aber `PT`/`ET` als einfache `TIME`-Datenvariablen statt Adapter.
- **AX_TOF**: die `events`-Variante ohne `ET`-Ausgang.
- **AX_ATM_FB_TON** / **AX_ATM_FB_TP**: dieselbe Adapter- Erweiterung für Ein-Verzögerung bzw. Impuls statt Aus-Verzögerung.

Fazit

`AX_ATM_FB_TOF` überträgt das bewährte `AX_FB_TOF`-Timing-Verhalten in eine vollständig adapterbasierte Schnittstelle, mit Change-Detection auf dem `ET`-Ausgang, um unnötige Ereignisse zu vermeiden.

AX_ATM_FB_TON



• — ○ • — ○ • — ○ •

Einleitung

`AX_ATM_FB_TON` ist die adapterbasierte Variante von AX_FB_TON: derselbe IEC-61131-3-Einschaltverzögerungstimer (`iec61131::timers::FB_TON`), aber mit `PT` (Sollzeit) als Socket und `ET` (abgelaufene Zeit) als Plug — beide vom Typ ATM — statt als einfache `TIME`-Datenvariablen. Das erlaubt, den Baustein direkt in adapterbasierte Ketten einzuhängen (z. B. hinter ein `AR_SUBSCRIBE_1`, das eine per OPC-UA geschriebene REAL-Sekundenzahl liefert, umgewandelt in `TIME`), ohne an jeder Verwendungsstelle eigene Datenverbindungen für `PT`/`ET` zu ziehen.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Event, nicht auslösend): Normaler Ausführungsaufruf. Die Anwendung hängt hier typischerweise einen `E_CYCLE` an, damit `ET` periodisch aktualisiert wird, während der Timer läuft — ohne einen externen `E_CYCLE` bliebe `ET` zwischen `IN`- bzw. `PT`-Ereignissen stehen.

Ereignis-Ausgänge

Keine eigenen Ereignis-Ausgänge; Ereignisse laufen ausschließlich über die Adapter-Plugs `Q` und `ET`.

Daten-Eingänge

Keine eigenen Daten-Eingänge. Alle Daten werden über die Adapter-Sockets übertragen.

Daten-Ausgänge

Keine eigenen Daten-Ausgänge. Alle Daten werden über die Adapter-Plugs übertragen.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	IN	adapter::types::unidirectional::AX	Startsignal
Socket (Eingang)	PT	adapter::types::unidirectional::ATM	Sollzeit
Plug (Ausgang)	Q	adapter::types::unidirectional::AX	Ausgang, TRUE nachdem IN seit PT ansteht
Plug (Ausgang)	ET	adapter::types::unidirectional::ATM	Abgelaufene Zeit, Event nur bei Wertänderung

Funktionsweise

Intern wird derselbe FB_TON-Standardbaustein wie in AX_FB_TON verwendet, ergänzt um zwei E_D_FF/E_D_FF_ANY-Puffer:

1. IN.E1, das eigene REQ-Ereignis und PT.E1 lösen alle gemeinsam FB_TON.REQ aus — jede der drei Quellen (Eingang ändert sich, Sollzeit ändert sich, oder ein zyklischer Anstoß von außen) soll die aktuelle Zeitbasis neu berechnen.
2. FB_TON.Q läuft über ein E_D_FF (Change-Detection) auf Q.D1/Q.E1 — der Q-Plug feuert sein Ereignis nur, wenn sich der boolesche Ausgang tatsächlich ändert.
3. FB_TON.ET läuft über ein E_D_FF_ANY (generische Change-Detection) auf ET.D1/ET.E1 — ebenso nur bei tatsächlicher Wertänderung, nicht bei jedem REQ-Zyklus.

Technische Besonderheiten

- **REQ bleibt bewusst ein einfaches Event, keine Adapterverbindung.** Damit kann die Anwendung selbst entscheiden, mit welcher Zykluszeit ET aktualisiert wird (z. B. per E_CYCLE), statt dass der Baustein eine feste Aktualisierungsrate vorgibt.
- **ET-Events nur bei Wertänderung**, nach demselben Prinzip wie ASSEMBLE_AB_FROM_AX und AB_AX_SEL_AB — verhindert Event-Flut bei jedem E_CYCLE-Tick, wenn sich der Zeitwert (je nach Auflösung) gar nicht geändert hat.
- **PT ist live, nicht gelatched — IN.E1, PT.E1 und REQ sind für FB_TON gleichwertige Auslöser.** PT.D1 liegt als reine Datenverbindung dauerhaft an FB_TON.PT an; der Adapter puffert oder entprellt PT nicht. FB_TON selbst unterscheidet nicht, welches der drei Ereignisse (IN.E1, PT.E1, REQ) die Neuauswertung ausgelöst hat — es wertet in jedem Fall mit dem *aktuellen* PT gegen das *aktuelle* ET neu aus. Eine zur Laufzeit geänderte Sollzeit ist also nicht an der steigenden Flanke von IN eingefroren, sondern wirkt bei der nächsten Auswertung sofort:
  - **PT = 0**: Q wird bei der nächsten Auswertung sofort TRUE, da ET (≥ 0) die Sollzeit bereits erreicht/überschritten hat.
  - **PT wird während des Timings verkleinert**, sodass das bereits gelaufene ET die neue, kleinere Sollzeit übersteigt: Q wird bei der nächsten Auswertung sofort TRUE, statt bis zur ursprünglich größeren Sollzeit zu warten.
  - **PT wird während des Timings vergrößert**: Q bleibt entsprechend länger FALSE.
  - **Schnell aufeinanderfolgende PT.E1-Ereignisse werden nicht zusammengefasst** — jedes einzelne Ereignis löst eine eigene FB_TON.REQ-Auswertung aus, es gibt keine Debounce-Logik im Adapter.
  - **Vor dem allerersten PT.E1** liegt an FB_TON.PT der TIME-Standardwert T#0s an, nicht ein projektspezifischer Default — entspricht dann dem Fall PT = 0 oben, falls IN vorher schon TRUE wird.
- **Identisches Verhalten in AX_ATM_FB_TOF und AX_ATM_FB_TP.** Alle drei Bausteine sind strukturell identisch verdrahtet (gleiches IN.E1/PT.E1/REQ → REQ-Muster, gleiches E_D_FF/E_D_FF_ANY-Change-Detection- Muster für Q/ET) — nur der intern gewrappte Standardbaustein (FB_TON/FB_TOF/FB_TP) unterscheidet sich. Das oben beschriebene PT-Verhalten gilt für alle drei gleichermaßen; siehe deren jeweilige Seiten für die timerspezifischen PT=0-Grenzfälle.
- **adapter::events::unidirectional::timers-Familie (AX_TON etc.) hat keinen ET-Ausgang.** Wird die verbleibende/abgelaufene Zeit gebraucht (z. B. für einen Countdown in der Visu), ist diese iec61131-3-Familie erforderlich, nicht die einfachere events-Familie.

Zustandsübersicht

Wie `AX_FB_TON`: kein eigenes ECC, reines Composite-Verdrahtungsmuster um den Standard-`FB_TON`.

1. **Ruhezustand**: `IN` ist FALSE, `Q` ist FALSE, `ET` ist 0.
2. **Timing-Zustand**: `IN` ist TRUE, Timer läuft, `ET` steigt bis `PT`, `Q` bleibt FALSE.
3. **Ausgelöst**: `ET` hat `PT` erreicht, `Q` wird TRUE.
4. **Reset**: `IN` wird FALSE, `Q` wird FALSE, `ET` wird 0.

Anwendungsszenarien

- **Countdown-Anzeige über OPC-UA**: `PT` aus einem abonnierten REAL-Sekundenwert (z. B. `Override.Dauer`) speisen, `ET` (bzw. `PT - ET` für die Restzeit) über einen Publish-Baustein an die Visu weiterreichen.
- Jede Stelle, an der bereits `AX_FB_TON` verwendet wird, aber zusätzlich `PT`/`ET` über adapterbasierte Ketten statt einzelner Datenverbindungen fließen sollen.

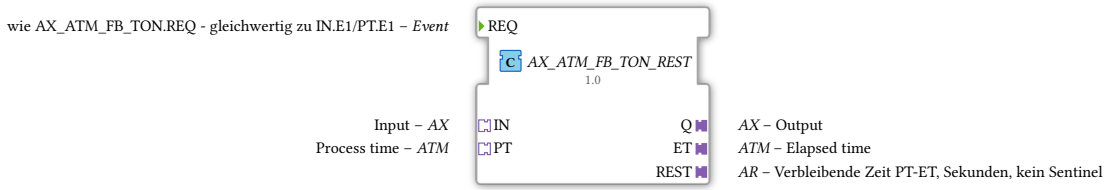
⚖️ Vergleich mit ähnlichen Bausteinen

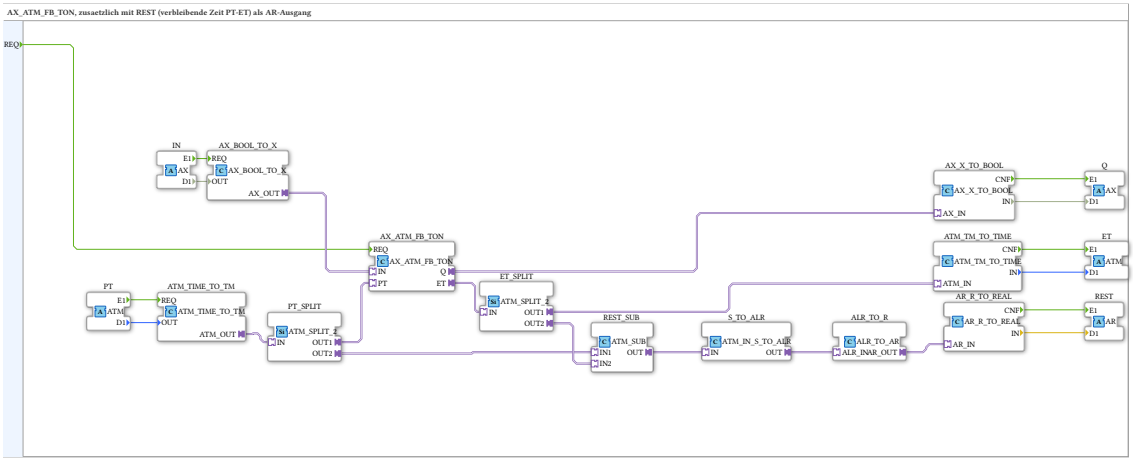
- **AX_FB_TON**: identisches Timing-Verhalten, aber `PT`/`ET` als einfache `TIME`-Datenvariablen statt Adapter.
- **AX_TON**: die `events`-Variante ohne `ET`-Ausgang überhaupt — geeignet, wenn nur das Timing-Ergebnis (`Q`), nicht aber die verstrichene Zeit gebraucht wird.
- **AX_ATM_FB_TOF** / **AX_ATM_FB_TP**: dieselbe Adapter- Erweiterung für Aus-Verzögerung bzw. Impuls statt Ein-Verzögerung.

Fazit

`AX_ATM_FB_TON` überträgt das bewährte `AX_FB_TON`-Timing-Verhalten in eine vollständig adapterbasierte Schnittstelle, sodass sowohl Sollzeit als auch abgelaufene Zeit ohne zusätzliche Datenverbindungen durch adapterbasierte Netzwerke fließen können — inklusive Change-Detection auf dem `ET`-Ausgang, um unnötige Ereignisse zu vermeiden.

AX_ATM_FB_TON_REST





• — ○ • — ○ • — ○ •

Einleitung

`AX_ATM_FB_TON_REST` umhüllt AX_ATM_FB_TON mit einem dritten Adapter-Plug, `REST`, der die verbleibende Zeit (`PT - ET`) als REAL-Sekundenwert führt. Mehrere Anwendungen brauchen nicht nur „ist der Timer abgelaufen“ (`Q`) oder „wie viel Zeit ist vergangen“ (`ET`), sondern „wie viel Zeit bleibt noch“ für eine Countdown-Anzeige — das extern aus `PT`/`ET` zu berechnen bedeutet, an jeder Verwendungsstelle dieselbe

Adapterkette (ATM_SUB ⟶ ATM_IN_S_TO_ALR ⟶ ALR_TO_AR) zu wiederholen. Dieser Baustein erledigt das einmal, intern, sodass der Aufrufer REST als einen einzigen, publish-fertigen Adapter-Plug erhält.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Event, nicht auslösend): dieselbe Rolle wie bei AX_ATM_FB_TON — die Anwendung hängt hier typischerweise einen E_CYCLE an, damit ET/REST aktualisiert bleiben, während der Timer läuft.

Ereignis-Ausgänge

Keine eigenen; Ereignisse laufen ausschließlich über die Adapter-Plugs Q, ET und REST.

Daten-Eingänge

Keine eigenen. Alle Daten werden über die Adapter-Sockets übertragen.

Daten-Ausgänge

Keine eigenen. Alle Daten werden über die Adapter-Plugs übertragen.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	IN	adapter::types::unidirectional::AX	Startsignal
Socket (Eingang)	PT	adapter::types::unidirectional::ATM	Sollzeit
Plug (Ausgang)	Q	adapter::types::unidirectional::AX	Ausgang, TRUE sobald IN seit PT ansteht
Plug (Ausgang)	ET	adapter::types::unidirectional::ATM	Abgelaufene Zeit, Event nur bei Wertänderung
Plug (Ausgang)	REST	adapter::types::unidirectional::AR	Verbleibende Zeit PT - ET in Sekunden, kein Sentinel-Wert

Funktionsweise

Intern übernimmt eine einfache AX_ATM_FB_TON-Instanz das eigentliche Timing. Darum herum:

1. PT wird aufgeteilt (ATM_SPLIT_2), sodass der Wert sowohl das PT des gewrappten Timers selbst als auch ein ATM_SUB erreicht, das PT - ET berechnet.
2. Das ET des gewrappten Timers wird ebenso aufgeteilt, sodass es sowohl den eigenen ET-Plug dieses Bausteins als auch dasselbe ATM_SUB (als Subtrahend) erreicht.
3. ATM_SUB.OUT (ein ATM, also TIME) wird über ATM_IN_S_TO_ALR ⟶ ALR_TO_AR in Sekunden umgerechnet und ergibt den REST-Wert.
4. Q wird unverändert vom gewrappten Timer durchgereicht.

Warum einfaches Socket/Plug-Durchreichen hier nicht funktioniert, und was der Baustein stattdessen tut: Ein Socket kann nur *von einem Plug gespeist* werden, und ein Plug kann nur *einen Socket speisen* — eine simple <Connection Source="IN" Destination="InnerTimer.IN"/> zwischen dem eigenen IN-Socket dieses Bausteins und dem IN-Socket des gewrappten Timers (beides Sockets) ist keine legale Adapterverbindung, und dasselbe symmetrische Problem besteht bei Q (beides Plugs). Weil AX_ATM_FB_TON_REST selbst ein FBType ist (kein SubAppType), ist der lesende Dot-Zugriff auf den aktuellen Wert eines Sockets via IN.D1/IN.E1 **hier legal** — die Lösung besteht daher darin, an jeder Grenze über ein Paar reiner Datenvariablen zu überbrücken: AX_BOOL_TO_X liest IN.D1/IN.E1 und gibt den Wert als Plug wieder aus, der den IN-Socket des gewrappten Timers speist; symmetrisch dazu liest AX_X_TO_BOOL den Q-Plug des gewrappten Timers und gibt ihn auf den eigenen Q-Plug dieses Bausteins aus. Dasselbe Muster gilt für PT/ET (ATM_TIME_TO_TM/ATM_TM_TO_TIME) — jedes Paar überbrückt eine Richtung hinein und eine Richtung hinaus, weil sowohl Socket als auch Plug des gewrappten Timers einen gebundenen Wert brauchen. REST braucht nur die ausgehende Hälfte dieses Musters — er wird rein intern berechnet und nie von außen gespeist — daher kommt nur AR_R_TO_REAL zum Einsatz: Er liest den intern berechneten AR-Wert (über seinen eigenen AR_IN-Socket, gespeist vom Plug von ALR_TO_R)

und gibt den reinen REAL-Wert auf die eigenen Plug-Pins von REST aus. Dieselbe Regel — ein Socket kann nur von einem Plug gespeist werden — gilt für jeden Adaptertyp, nicht nur für AX/ATM/AR.

Technische Details

- **REST führt keine Sentinel-Konvention (kein „−1 = inaktiv“).** Ein Aufrufer, der eine solche braucht (z. B. „−1, solange der Timer nicht läuft“), gated REST selbst mit einem nachgeschalteten AR_AX_SEL_AR, der abhängig von seiner eigenen „aktiv“-Bedingung zwischen REST und einer Konstante auswählt — dieser Baustein meldet immer nur die rohe PT - ET-Differenz.
- **REST aktualisiert sich mit derselben Auflösung wie ET**, also nur bei tatsächlicher Wertänderung (geerbt von AX_ATM_FB_TONs E_D_FF_ANY auf ET), nicht bei jedem REQ-/ E_CYCLE-Tick.
- **Sämtliche PT-Semantik von AX_ATM_FB_TON gilt unverändert** — PT ist live, nicht gelatched; PT = 0 lässt Q (und damit ein REST von 0) bereits bei der nächsten Auswertung wahr/erreicht werden. Siehe AX_ATM_FB_TON für die vollständige Aufschlüsselung dieser Grenzfälle.

Zustandsübersicht

Dieselben vier Zustände wie AX_ATM_FB_TON, wobei REST durchgehend PT - ET folgt:

1. **Ruhezustand**: IN ist FALSE, Q ist FALSE, ET ist 0, REST entspricht PT in Sekunden.
2. **Ausgelöst**: ET hat PT erreicht, REST erreicht 0, Q wird TRUE.
3. **Reset**: IN wird FALSE, Q wird FALSE, ET wird 0, REST kehrt zu PT zurück.

Anwendungsszenarien

- **Countdown-Anzeige über OPC-UA**: PT aus einem abonnierten REAL-Sekundenwert speisen, REST direkt an die Visu weitergeben — keine externe ATM_SUB/ATM_IN_S_TO_ALR/ALR_TO_AR-Kette an der Verwendungsstelle nötig.
- Jede Stelle, die heute AX_ATM_FB_TON zusammen mit einer eigenen Kopie der ATM_SUB/ATM_IN_S_TO_ALR/ALR_TO_AR-Kette verdrahtet, um einen Restzeit-Wert zu erhalten — dieser Baustein ersetzt alle vier Bausteine durch einen einzigen.

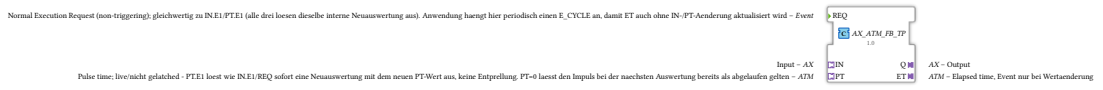
⚖️ Vergleich mit ähnlichen Bausteinen

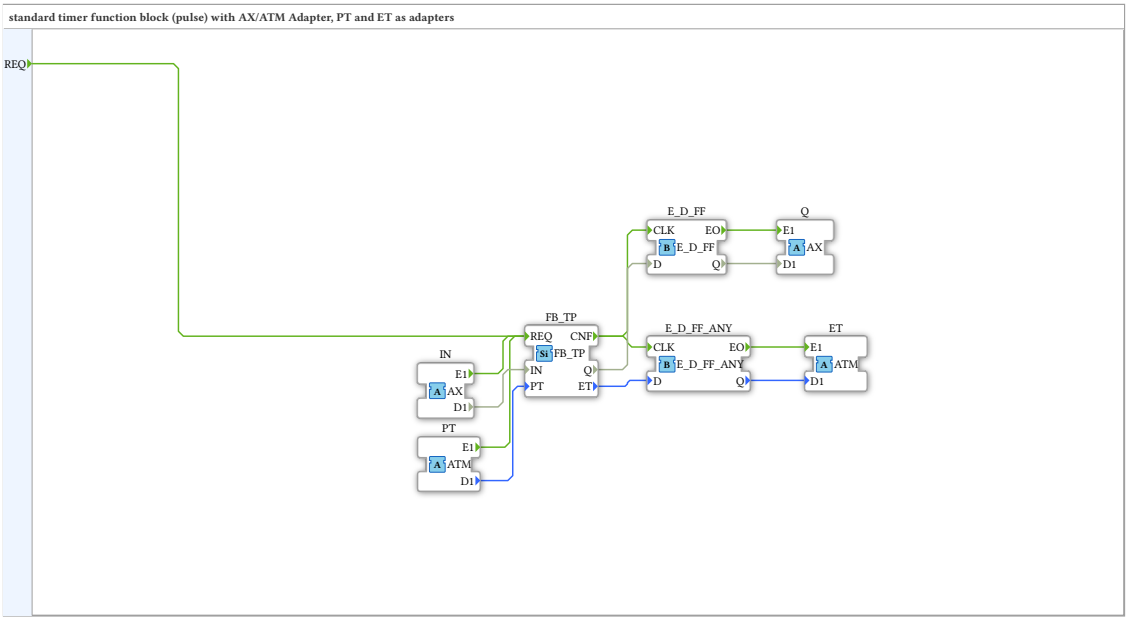
- **AX_ATM_FB_TON**: identisches Timing-Verhalten und identische Schnittstelle, minus den REST-Plug — verwenden, wenn nur Q/ET gebraucht werden.
- **AR_AX_SEL_AR**: kein Ersatz, sondern der Begleitbaustein, um einen Sentinel-Wert (z. B. „−1 = inaktiv“) auf den rohen REST-Ausgang dieses Bausteins aufzusetzen.

Fazit

AX_ATM_FB_TON_REST fasst das wiederkehrende Muster „Einschaltverzögerungstimer mit Restzeit-Anzeige“ — bisher vier separat verdrahtete Bausteine pro Verwendungsstelle — in einem einzigen adapterbasierten Baustein zusammen. Dafür nimmt er eine etwas komplexere interne Verdrahtung (Socket/Plug-Überbrückung an jeder Grenze) in Kauf, zugunsten einer deutlich einfacheren Verwendungsstelle.

AX_ATM_FB_TP





• – ○ • – ○ • – ○ •

Einleitung

`AX_ATM_FB_TP` ist die adapterbasierte Variante von AX_FB_TP: derselbe IEC-61131-3-Impulstimer (`iec61131::timers::FB_TP`), aber mit `PT` (Impulszeit) als Socket und `ET` (abgelaufene Zeit) als Plug — beide vom Typ ATM — statt als einfache `TIME`-Datenvariablen. Ebenso wie AX_ATM_FB_TON erlaubt das, den Baustein direkt in adapterbasierte Ketten einzuhängen, ohne an jeder Verwendungsstelle eigene Datenverbindungen für `PT`/`ET` zu ziehen.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Event, nicht auslösend): Normaler Ausführungsaufruf. Die Anwendung hängt hier typischerweise einen `E_CYCLE` an, damit `ET` periodisch aktualisiert wird, während der Puls läuft.

Ereignis-Ausgänge

Keine eigenen Ereignis-Ausgänge; Ereignisse laufen ausschließlich über die Adapter-Plugs `Q` und `ET`.

Daten-Eingänge

Keine eigenen Daten-Eingänge. Alle Daten werden über die Adapter-Sockets übertragen.

Daten-Ausgänge

Keine eigenen Daten-Ausgänge. Alle Daten werden über die Adapter-Plugs übertragen.

Adapter

Richtung	Name	Typ	Beschreibung
Socket (Eingang)	`IN`	`adapter::types::unidirectional::AX`	Startsignal
Socket (Eingang)	`PT`	`adapter::types::unidirectional::ATM`	Impulszeit
Plug (Ausgang)	`Q`	`adapter::types::unidirectional::AX`	Ausgang, TRUE für die Dauer `PT` nach steigender Flanke von `IN`
Plug (Ausgang)	`ET`	`adapter::types::unidirectional::ATM`	Abgelaufene Zeit, Event nur bei Wertänderung

Funktionsweise

Intern wird derselbe FB_TP-Standardbaustein wie in AX_FB_TP verwendet, ergänzt um zwei E_D_FF/E_D_FF_ANY-Puffer:

1. IN.E1, das eigene REQ-Ereignis und PT.E1 lösen alle gemeinsam FB_TP.REQ aus.
2. FB_TP.Q läuft über ein E_D_FF (Change-Detection) auf Q.D1/Q.E1.
3. FB_TP.ET läuft über ein E_D_FF_ANY (generische Change-Detection) auf ET.D1/ET.E1 — nur bei tatsächlicher Wertänderung, nicht bei jedem REQ-Zyklus.

Eine steigende Flanke an IN setzt Q sofort auf TRUE und startet den Zeitablauf; Q bleibt für die Dauer PT TRUE, unabhängig davon, ob IN währenddessen wieder auf FALSE fällt (echter Impuls, nicht retriggerbar).

Technische Besonderheiten

- **REQ bleibt bewusst ein einfaches Event, keine Adapterverbindung**, wie bei AX_ATM_FB_TON — die Anwendung bestimmt selbst die Aktualisierungsrate von ET über einen extern angeschlossenen E_CYCLE.
- **ET-Events nur bei Wertänderung**, nach demselben Prinzip wie ASSEMBLE_AB_FROM_AX.
- **PT ist live, nicht gelatched — IN.E1, PT.E1 und REQ sind für FB_TP gleichwertige Auslöser**, genau wie bei AX_ATM_FB_TON. PT.D1 liegt als reine Datenverbindung dauerhaft an FB_TP.PT an, keine Pufferung/Entprellung im Adapter:
  - **PT = 0**: Der Impuls gilt bei der nächsten Auswertung bereits als abgelaufen — für den Beobachter kann Q praktisch keine sichtbare TRUE-Phase haben.
  - **PT wird während des laufenden Impulses verkleinert**, sodass das bereits gelaufene ET die neue Sollzeit übersteigt: Q wird sofort FALSE.
  - **PT wird während des laufenden Impulses vergrößert**: Q bleibt entsprechend länger TRUE.
  - **Schnell aufeinanderfolgende PT.E1-Ereignisse** lösen jeweils eine eigene Auswertung aus, keine Zusammenfassung.
  - **Vor dem allerersten PT.E1** liegt an FB_TP.PT der TIME-Standardwert T#0s an.
- **Identisches Verhalten in AX_ATM_FB_TON und AX_ATM_FB_TOF** — alle drei sind strukturell identisch verdrahtet, nur der intern gewrappte Standardbaustein unterscheidet sich.

Zustandsübersicht

1. **Ruhezustand**: IN ist FALSE, Q ist FALSE, ET ist 0.
2. **Steigende Flanke an IN**: Q wird sofort TRUE, ET beginnt zu laufen.
3. **Impuls aktiv**: ET steigt bis PT, Q bleibt TRUE — unabhängig vom weiteren Verlauf von IN.
4. **Abgelaufen**: ET hat PT erreicht, Q wird FALSE.

Anwendungsszenarien

- Definierte, nicht retriggerbare Impulslängen (z. B. Mindest-Einschaltzeit eines Ventils) mit sichtbarem Zeitfortschritt über ET.
- Jede Stelle, an der bereits AX_FB_TP verwendet wird, aber PT/ET über adapterbasierte Ketten fließen sollen.

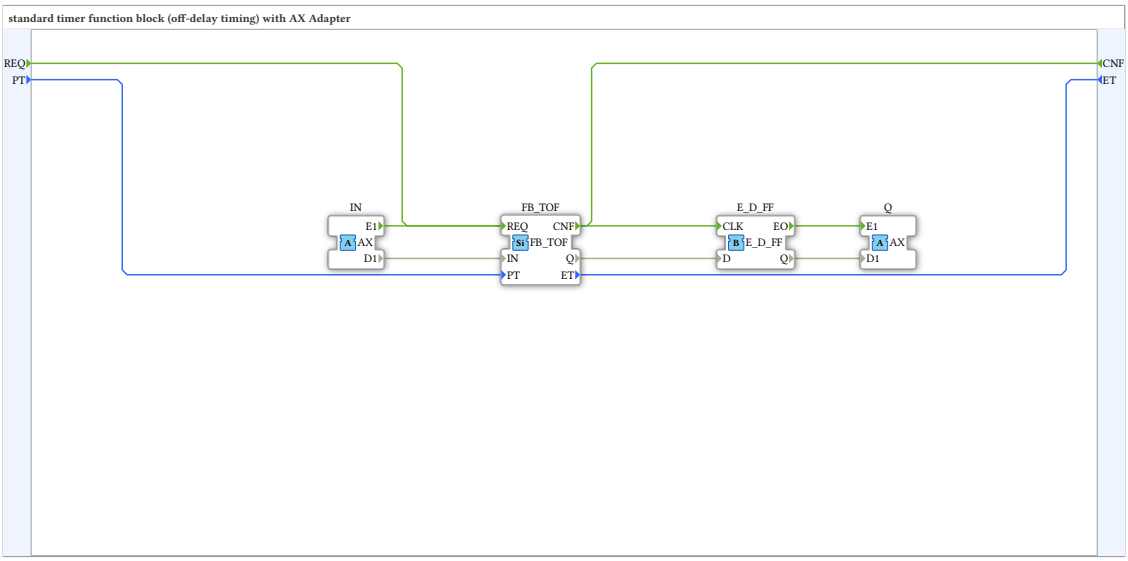
⚖️ Vergleich mit ähnlichen Bausteinen

- AX_FB_TP: identisches Timing-Verhalten, aber PT/ET als einfache TIME-Datenvariablen statt Adapter.
- AX_TP: die events-Variante ohne ET-Ausgang.
- AX_ATM_FB_TON / AX_ATM_FB_TOF: dieselbe Adapter- Erweiterung für Ein- bzw. Aus-Verzögerung statt Impuls.

Fazit

AX_ATM_FB_TP überträgt das bewährte AX_FB_TP-Timing-Verhalten in eine vollständig adapterbasierte Schnittstelle, mit Change-Detection auf dem ET-Ausgang, um unnötige Ereignisse zu vermeiden.

AX_FB_TOF

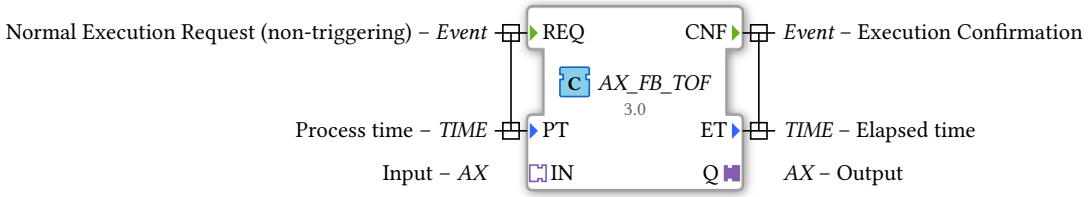


• – ○ • – ○ • – ○ •

Einleitung

Wichtiger Hinweis: Dieser Baustein funktioniert nur korrekt, wenn er zyklisch aufgerufen wird.

AX_FB_TOF ist ein standardisierter Timer-Funktionsblock gemäß IEC 61131-3, der eine Ausschaltverzögerung (Off-Delay Timing) realisiert. Der Baustein hält das Ausgangssignal nach dem Abschalten des Eingangssignals für eine definierte Zeitdauer aufrecht.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Event): Normal Execution Request (übernimmt den Wert von PT); ruft den Baustein auf (muss zyklisch aufgerufen werden)

Ereignis-Ausgänge

- **CNF** (Event): Execution Confirmation (mit `ET`)

Daten-Eingänge

- **PT** (TIME): Prozesszeit - definiert die Dauer der Ausschaltverzögerung

Daten-Ausgänge

- **ET** (TIME): Abgelaufene Zeit (Elapsed Time)

Adapter

- **IN** (Socket): Empfängt Eingangsereignisse und -daten (Typ: `adapter::types::unidirectional::AX`)
- **Q** (Plug): Sendet Ausgangsereignisse und -daten (Typ: `adapter::types::unidirectional::AX`)

Funktionsweise

Der AX_FB_TOF-Block arbeitet als Ausschaltverzögerungstimer:

1. Wenn der Eingang am IN-Adapter TRUE wird, schaltet der Ausgang am Q-Adapter sofort auf TRUE.

2. Wenn der Eingang am IN-Adapter von TRUE auf FALSE wechselt, startet der interne Timer.
3. Der Ausgang am Q-Adapter bleibt TRUE, solange der Timer läuft und PT noch nicht erreicht wurde.
4. Nach Ablauf der Zeit PT wechselt der Ausgang am Q-Adapter auf FALSE.
5. Der Datenausgang ET zeigt die Zeit an, die seit dem Abfallen des Eingangssignals vergangen ist.
6. Wird der Eingang am IN-Adapter wieder TRUE, während der Timer läuft, wird der Timer zurückgesetzt und Q bleibt TRUE.

Technische Besonderheiten

- Implementiert auf Basis des IEC 61131-3 TOF-Standards
- Verwendet unidirektionale Adapter für die Integration in moderne Steuerungskonzepte
- Bietet einen ET-Ausgang zur Überwachung der verbleibenden Verzögerungszeit
- Robuste Implementierung für zuverlässige Nachlaufsteuerungen

Zustandsübersicht

1. **Inaktiv**: IN ist FALSE, Q ist FALSE, ET ist 0.
2. **Aktiv**: IN ist TRUE, Q ist TRUE, ET ist 0.
3. **Verzögerungsphase**: IN wechselt auf FALSE, Timer läuft, Q bleibt TRUE, ET steigt.
4. **Abgelaufen**: ET erreicht PT, Q wechselt auf FALSE.

Anwendungsszenarien

- Nachlaufsteuerungen (z. B. Lüftersteuerung nach dem Abschalten eines Geräts)
- Entprellung von Signalen, die kurzzeitig abfallen könnten
- Sicherstellung von Mindestlaufzeiten nach einem Stopp-Befehl

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu ereignisbasierten Timern wie AX_TOF folgt dieser Baustein dem klassischen SPS-Verhalten nach IEC 61131-3 und bietet zusätzlich den ET-Statusausgang.

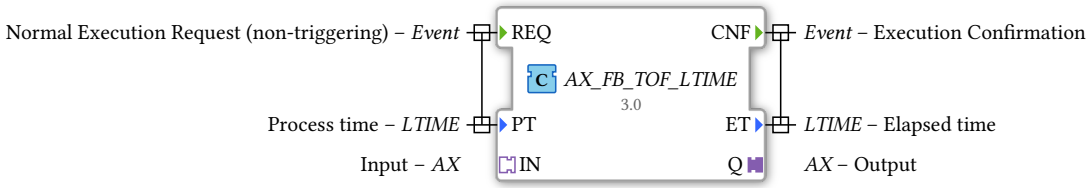
🛠️ Zugehörige Übungen

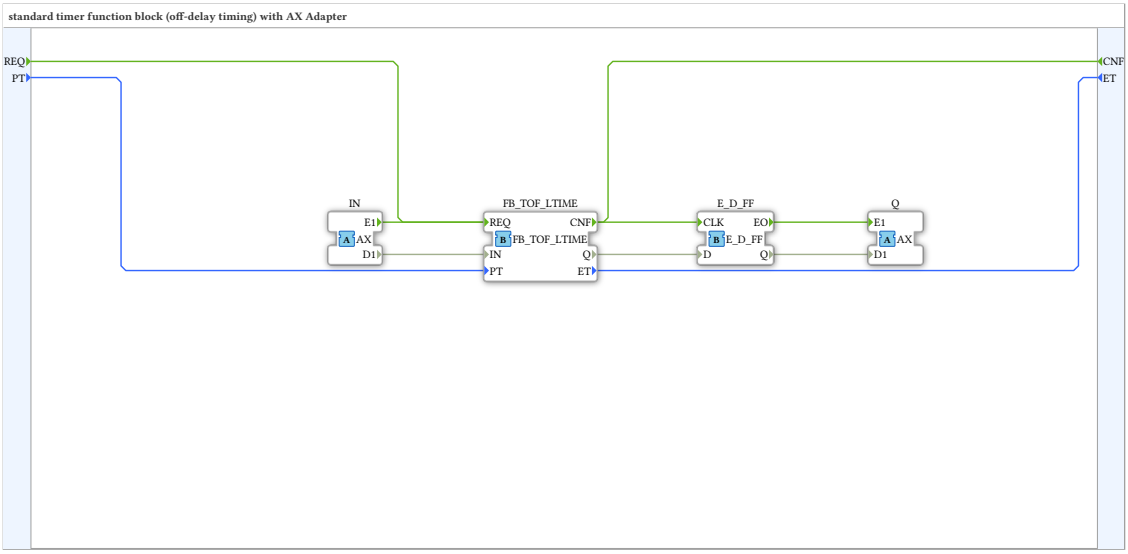
- Uebung_020e2_AX

Fazit

AX_FB_TOF ist der ideale Baustein für alle Anwendungen, die eine präzise Ausschaltverzögerung mit Statusrückmeldung über die abgelaufene Zeit erfordern.

AX_FB_TOF_LTIME





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AX_FB_TOF_LTIME** realisiert einen Standard-Timer für die Ausschaltverzögerung (Off-Delay) nach IEC 61131-3. Er ist speziell für den Einsatz mit einem AX-Adapter ausgelegt und verwendet dabei einen internen Timerbaustein sowie ein D-Flipflop, um das Zeitverhalten präzise abzubilden. Die Zeitbasis arbeitet mit dem Datentyp `LTIME`, wodurch eine hohe zeitliche Auflösung erreicht wird.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Kommentar
`REQ`	Normaler Ausführungsauftrag (nicht auslösend) – startet bzw. aktualisiert die Timerberechnung. Der zugehörige Daten-Eingang ist `PT`.

Ereignis-Ausgänge

Ereignis	Kommentar
`CNF`	Bestätigung der Ausführung – signalisiert den Abschluss einer Timeraktualisierung. Der zugehörige Daten-Ausgang ist `ET`.

Daten-Eingänge

Variable	Typ	Kommentar
`PT`	`LTIME`	Prozesszeit – die gewünschte Ausschaltverzögerungsdauer.

Daten-Ausgänge

Variable	Typ	Kommentar
`ET`	`LTIME`	Verstrichene Zeit – die bereits abgelaufene Zeit seit Start der Verzögerung.

Adapter

Richtung	Adaptertyp	Kommentar

Socket `IN`	`adapter::types::unidirectional::AX`	Eingangsadapter – nimmt das binäre Steuersignal entgegen (z. B. über Ereignis `E1` und Daten `D1`).
Plug `Q`	`adapter::types::unidirectional::AX`	Ausgangsadapter – gibt das zeitverzögerte Ausgangssignal weiter (über Ereignis `E1` und Daten `D1`).

Funktionsweise

Der Baustein verhält sich wie eine klassische Ausschaltverzögerung (TOF):

- Solange der Eingang `IN` (über den Adapter) **TRUE** ist, ist der Ausgang `Q` ebenfalls **TRUE** und die verstrichene Zeit `ET` wird auf Null gesetzt.
- Sobald `IN` von **TRUE** auf **FALSE** wechselt, startet die Zeitmessung. Der Ausgang `Q` bleibt für die Dauer der vorgegebenen Zeit `PT` **TRUE** und fällt erst danach auf **FALSE**.
- Wird `IN` vor Ablauf der Verzögerungszeit wieder **TRUE**, wird der Timer zurückgesetzt und `Q` bleibt **TRUE**.

Die ereignisgesteuerte Verarbeitung erfolgt wie folgt:

1. Das Ereignis `REQ` (entweder direkt oder vom Adaptereingang `IN.E1`) triggert den internen Timer `FB_TOF_LTIME`.
2. Der Timer berechnet die verstrichene Zeit und gibt bei Abschluss das Ereignis `CNF` aus.
3. Dieses `CNF` taktet das D-Flipflop `E_D_FF`, das den aktuellen Timer-Ausgang (`Q` des Timers) speichert und auf den Ausgangsadapter `Q.D1` durchschaltet.
4. Gleichzeitig wird das `CNF`-Ereignis an den Ausgangsadapter `Q.E1` weitergeleitet, so dass der nachgeschaltete Adapter über die Änderung informiert wird.

Die Variable `ET` wird direkt vom Timer übernommen und zeigt jederzeit die aktuell abgelaufene Verzögerungszeit an.

Technische Besonderheiten

- **Hohe Zeitauflösung** durch Verwendung von `LTIME` (Long Time) gemäß IEC 61131-3.
- **Adapter-basierte Ein-/Ausgabe** über den unidirektionalen `AX`-Adapter ermöglicht eine lose Kopplung zwischen verschiedenen Ressourcen (z. B. zwischen Anwendung und E/A-Treiber).
- **Standardkonformität** nach IEC 61131-3 – das Verhalten entspricht exakt dem eines TOF-Timers.
- **Interne Struktur**: Der Baustein kombiniert einen reinen Timer (`FB_TOF_LTIME`) mit einem flankengetriggerten D-Flipflop, um das Ausschaltverzögerungsverhalten zu realisieren.

Zustandsübersicht

Zustand	Eingang `IN`	Ausgang `Q`	Verstrichene Zeit `ET`
Inaktiv / Bereit	TRUE	TRUE	0 (Null)
Verzögerung läuft	FALSE ⟶ TRUE-Rückkehr möglich	TRUE	0 … PT
Verzögerung abgelaufen	FALSE	FALSE	PT (konstant)

Ein erneuter Wechsel von `IN` auf **TRUE** setzt den Timer zurück und führt sofort wieder in den Zustand „Inaktiv / Bereit“. Ein Wechsel auf **FALSE** startet die Verzögerung von Neuem.

Anwendungsszenarien

- **Nachlaufsteuerung** von Lüftern, Pumpen oder Förderbändern nach dem Abschalten eines Signals.
- **Entprellung** von Signalen: ein kurzzeitiger Pegelwechsel wird durch die Verzögerung ausgefiltert.
- **Schutzzeiten** beim Schalten von induktiven Lasten: das Ausschalten wird um eine definierte Zeit verzögert.
- **Zeitgesteuerte Logik** in Produktionsanlagen, bei der ein Ausgang für eine bestimmte Dauer nach dem Eingangsereignis aktiv bleiben soll.

Vergleich mit ähnlichen Bausteinen

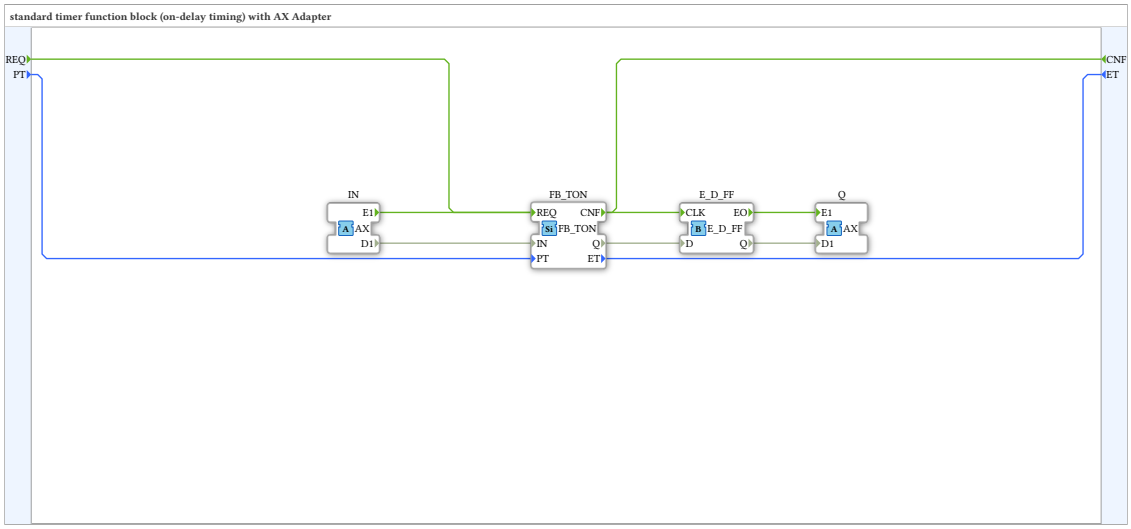
- **TON (Einschaltverzögerung)**: Der Ausgang wird erst nach einer Verzögerung nach dem Einschalten aktiv. – TOF verzögert das Ausschalten.

- **TP (Impuls)**: Der Ausgang gibt einen Impuls vorgegebener Länge, unabhängig von der Dauer des Eingangssignals. – TOF reagiert nur auf die fallende Flanke.
- **RTRIG / FTRIG (Flankenerkennung)**: Erkennen Signalwechsel, ohne Zeitkomponente. – TOF kombiniert Flankenerkennung mit Zeitsteuerung.

Fazit

Der `AX_FB_TOF_LTIME` ist ein leistungsfähiger, standardisierter Funktionsblock für zeitkritische Ausschaltverzögerungen in der Automatisierungstechnik. Durch die Verwendung des AX-Adapters lässt er sich flexibel in verteilte Systeme einbinden, während die interne Kombination aus Timer und Flipflop ein präzises und reproduzierbares Zeitverhalten gewährleistet. Er eignet sich besonders für Anwendungen, die eine genaue Zeitbasis und eine modulare Schnittstelle erfordern.

AX_FB_TON

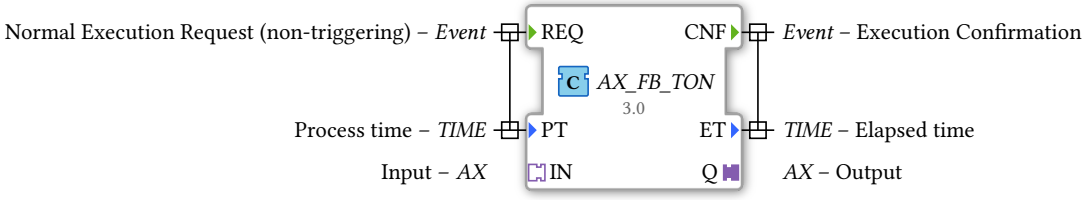


• – ◦ • – ◦ • – ◦ •

Einleitung

Wichtiger Hinweis: Dieser Baustein funktioniert nur korrekt, wenn er zyklisch aufgerufen wird.

AX_FB_TON ist ein standardisierter Timer-Funktionsblock gemäß IEC 61131-3, der eine Einschaltverzögerung (On-Delay Timing) realisiert und über Adapter-Schnittstellen verfügt. Der Baustein wartet nach Aktivierung eine definierte Zeit ab, bevor er das Ausgangssignal setzt, und stellt die abgelaufene Zeit zur Verfügung.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Event): Normal Execution Request (übernimmt den Wert von PT); ruft den Baustein auf (muss zyklisch aufgerufen werden)

Ereignis-Ausgänge

- **CNF** (Event): Execution Confirmation (Ausführungsbestätigung); trägt `ET`

Daten-Eingänge

- **PT** (TIME): Prozesszeit - definiert die Verzögerungsdauer

Daten-Ausgänge

- **ET** (TIME): Abgelaufene Zeit (Elapsed Time)

Adapter

- **IN** (Socket): Empfängt Eingangsereignisse und -daten (Typ: `adapter::types::unidirectional::AX`)
- **Q** (Plug): Sendet Ausgangsereignisse und -daten (Typ: `adapter::types::unidirectional::AX`)

Funktionsweise

Der AX_FB_TON-Block arbeitet als Einschaltverzögerungstimer. Bei Empfang eines Signals über den IN-Adapter startet der interne Timer.

1. Wenn der Eingang am IN-Adapter TRUE wird, beginnt der Timer zu laufen.
2. Nach Ablauf der eingestellten Prozesszeit (PT) wird der Ausgang am Q-Adapter TRUE.
3. Der Datenausgang ET zeigt die Zeit an, die seit dem Start des Timers vergangen ist, bis PT erreicht ist.
4. Wird der Eingang am IN-Adapter FALSE, wird der Timer zurückgesetzt und der Ausgang am Q-Adapter wird ebenfalls FALSE.

Technische Besonderheiten

- Implementiert auf Basis des IEC 61131-3 TON-Standards
- Verwendet unidirektionale Adapter für die Integration in adapterbasierte Architekturen
- Bietet einen ET-Ausgang zur Überwachung des Zeitverlaufs
- Kompatibel mit IEC 61131-3 und IEC 61499

Zustandsübersicht

1. **Ruhezustand**: IN ist FALSE, Timer inaktiv, Q ist FALSE, ET ist 0.
2. **Timing-Zustand**: IN ist TRUE, Timer läuft, ET steigt an, Q ist FALSE.
3. **Ausgelöst**: IN ist TRUE, ET hat PT erreicht, Q ist TRUE.
4. **Reset**: IN wird FALSE, Timer stoppt, Q wird FALSE, ET wird 0.

Anwendungsszenarien

- Klassische Einschaltverzögerungen in der SPS-Programmierung
- Überwachung von Prozesszeiten mit Visualisierung der Restlaufzeit (über ET)
- Kaskadierung von Zeitabläufen in adapterbasierten Systemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zum AX_TON bietet AX_FB_TON zusätzlich den Ausgang **ET** für die abgelaufene Zeit und folgt strikt dem Verhalten des IEC 61131-3 TON Blocks, während AX_TON eher auf den IEC 61499 E_TON optimiert ist.

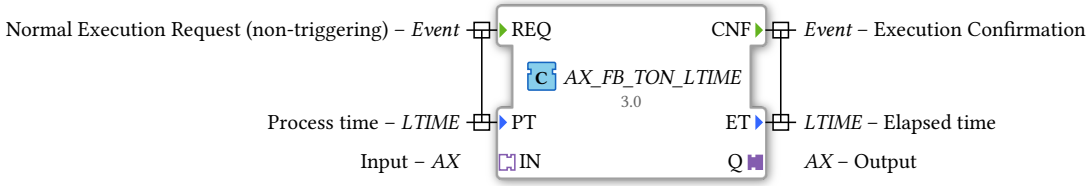
🛠️ Zugehörige Übungen

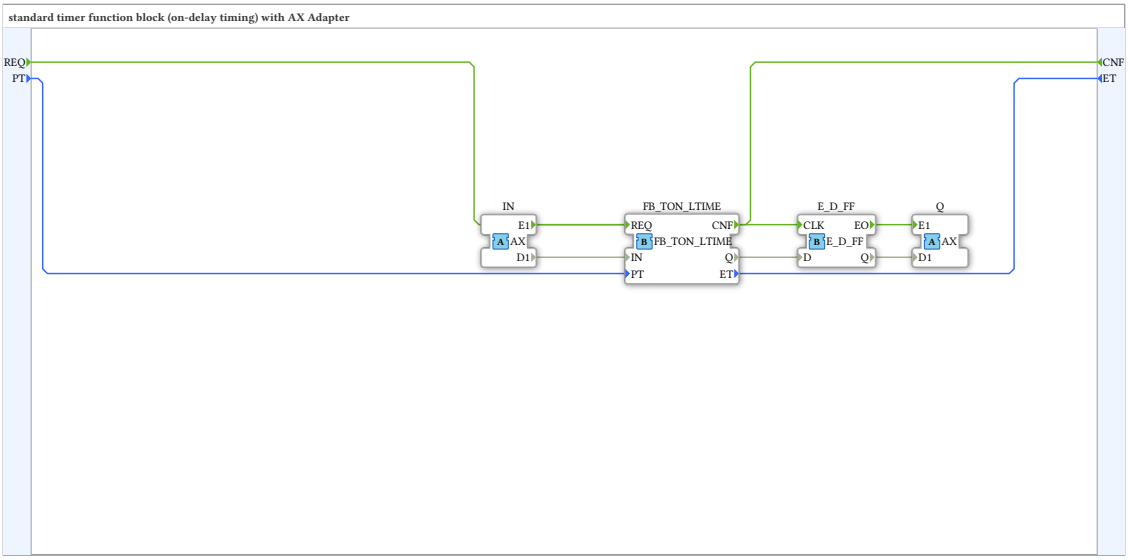
- Uebung_020c3_AX

Fazit

AX_FB_TON kombiniert das bewährte Verhalten des IEC 61131-3 TON-Timers mit der Flexibilität von Adapter-Schnittstellen und ermöglicht so eine präzise Zeitsteuerung mit Rückmeldung des aktuellen Zeitstatus.

AX_FB_TON_LTIME





• – ○ • – ○ • – ○ •

Einleitung

Der Baustein **AX_FB_TON_LTIME** realisiert eine Einschaltverzögerung (On-Delay Timer) nach IEC 61131-3 mit dem Datentyp `LTIME` und integriertem **AX-Adapter** Interface. Er eignet sich besonders für zeitkritische Steuerungsaufgaben, bei denen sowohl Ereignisse als auch Daten über standardisierte Adapter ausgetauscht werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung
REQ	`Event`	Normaler Ausführungsanstoß (nicht flankengetriggert); startet die Zeitmessung, wenn das Signal am Dateneingang `IN` oder am Adapter `IN.E1` aktiv ist.

Ereignis-Ausgänge

Name	Typ	Beschreibung
CNF	`Event`	Bestätigung der Ausführung. Wird ausgelöst, sobald der Timer abläuft oder die Bedingung erfüllt ist.

Daten-Eingänge

Name	Typ	Beschreibung
PT	`LTIME`	Vorgabezeit (Preset Time) für die Einschaltverzögerung.

Daten-Ausgänge

Name	Typ	Beschreibung
ET	`LTIME`	Abgelaufene Zeit (Elapsed Time) seit dem Start der Zeitmessung.

Adapter

Typ	Richtung	Beschreibung

IN	Socket (Input)	Unidirektionaler AX-Adapter für Eingangsdaten. Enthält ein Ereignis (E1) und einen Datenwert (D1). Das Ereignis startet die Zeitmessung; der Datenwert bestimmt den Timer-Eingang (IN).
Q	Plug (Output)	Unidirektionaler AX-Adapter für Ausgangsdaten. Gibt das Ergebnis der Zeitprüfung aus: 1, wenn die abgelaufene Zeit die Vorgabezeit erreicht hat, sonst 0.

Funktionsweise

Der Baustein ist als Composite realisiert und enthält intern den IEC-61131-3 Timer **FB_TON_LTIME** sowie ein flankengesteuertes D-Flip-Flop (**E_D_FF**).

- **Startbedingungen**: Die Zeitmessung wird gestartet, sobald ein Ereignis am Adapter IN.E1 oder am Ereigniseingang REQ eintrifft. Gleichzeitig muss der Datenwert IN.D1 (oder der interne Timer-Eingang) auf TRUE gesetzt sein.
- **Ablauf**: Der interne Timer zählt die Zeit hoch. Der Ausgang ET gibt die aktuell abgelaufene Zeit aus.
- **Ende**: Wenn die abgelaufene Zeit ET die Vorgabezeit PT erreicht, wird der Timer-Ausgang (Q) auf TRUE gesetzt. Dies löst das Ereignis CNF aus und taktet das D-Flip-Flop.
- **Ausgangsadapter**: Das D-Flip-Flop speichert den Zustand und gibt ihn über den Adapter Q.D1 aus. Das Ereignis Q.E1 signalisiert die Ausgangsänderung.

Technische Besonderheiten

- **LTIME-Datentyp**: Ermöglicht eine sehr hohe Zeitauflösung (Nanosekundenbereich) und große Zeitbereiche.
- **Adapter-Schnittstelle**: Der Baustein kann über standardisierte AX-Adapter in bestehende Systeme eingebunden werden, ohne direkte Ereignis- oder Datenverbindungen zu benötigen.
- **Flankengetriggerte Ausgabe**: Durch das interne D-Flip-Flop wird der Ausgangszustand erst bei einer positiven Flanke des Timer-Abschlusses aktualisiert – dies vermeidet unerwünschtes Flattern.
- **Internes Netzwerk**: Die Funktionalität wird durch die Kombination von Standard-Bausteinen erreicht, was die Implementierung transparent und testbar macht.

Zustandsübersicht

Der Baustein selbst besitzt kein eigenes Zustandsdiagramm, da er aus einem internen FBNetzwerk besteht. Das Verhalten kann aber aus dem zugrunde liegenden IEC-Timer-Baustein abgeleitet werden:

1. **Inaktiv** – Timer läuft nicht, ET = 0, Ausgang Q = FALSE.
2. **Zeit läuft** – Eingang IN = TRUE, Timer zählt ET hoch.
3. **Abgelaufen** – ET ≥ PT; Q wechselt auf TRUE und bleibt, bis IN auf FALSE fällt oder ein erneuter REQ die Zeitmessung zurücksetzt.

Anwendungsszenarien

- **Verzögerte Einschaltung von Aktoren**: Ein Ventil oder Motor soll erst nach einer definierten Verzögerungszeit eingeschaltet werden.
- **Synchronisation von Prozessen**: Warten auf das Eintreffen eines Signals über den Adapter, bevor eine zeitkritische Aktion ausgelöst wird.
- **Überwachung von Reaktionszeiten**: Erfassen der verstrichenen Zeit zwischen einem Ereignis und einer bestimmten Bedingung.
- **Erweiterung bestehender AX-basierter Systeme**: Austausch von Standard-Timerbausteinen durch die Adapter-Variante ohne Änderung der übergeordneten Struktur.

Vergleich mit ähnlichen Bausteinen

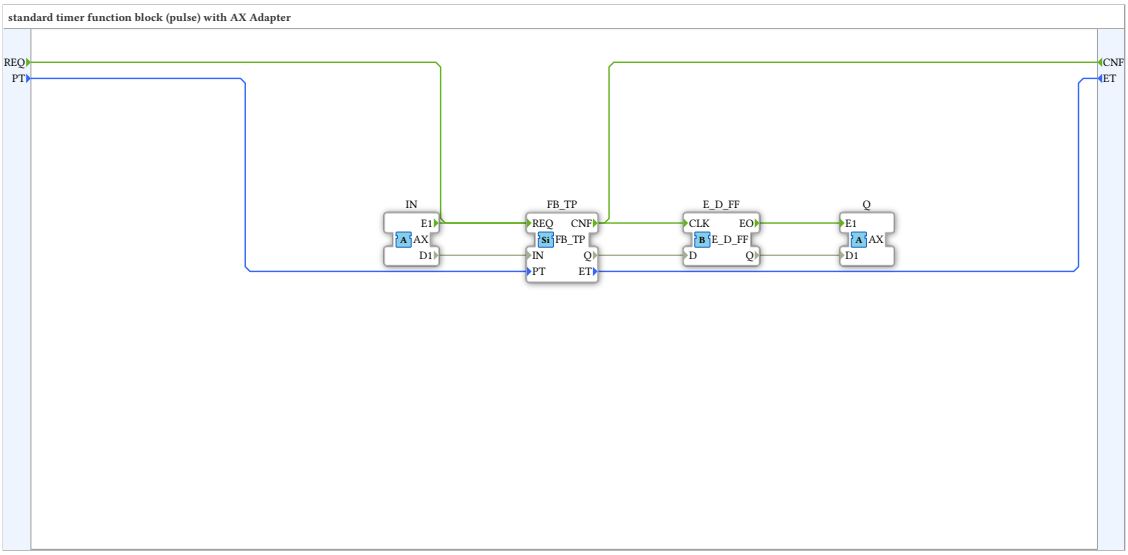
- **FB_TON_LTIME** (reiner IEC-Timer): Besitzt keine Adapter, benötigt direkte Ereignis- und Datenverbindungen. Der AX_FB_TON_LTIME erweitert diesen um eine standardisierte, leicht koppelbare Schnittstelle.
- **Ereignisgesteuerte Timer** (z.B. E_TON): Arbeiten oft mit Booleschen Eingängen und besitzen keine native LTIME-Unterstützung. Der hier vorgestellte Baustein kombiniert präzise Langzeitmessung mit einer flexiblen Adapter-Anbindung.

Fazit

Der **AX_FB_TON_LTIME** ist ein leistungsfähiger, adapterbasierter Einschaltverzögerungs-Baustein für die IEC-61499-Umgebung. Er vereint die Vorteile des standardisierten IEC-61131-3-Timers (LTIME, exakte

Zeitmessung) mit der Modularität und Wiederverwendbarkeit von AX-Adaptern. Durch die interne Kombination mit einem D-Flip-Flop wird eine zuverlässige, flankengetriggerte Ausgabe gewährleistet. Er eignet sich ideal für zeitkritische Steuerungen, die in ein Adapter-basiertes Architekturmodell eingebettet sind.

AX_FB_TP

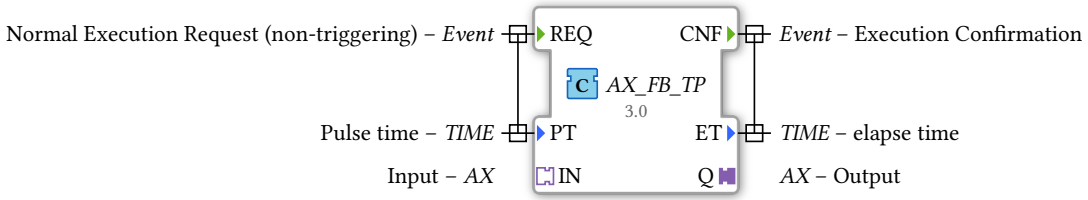


• – ○ • – ○ • – ○ •

Einleitung

Wichtiger Hinweis: Dieser Baustein funktioniert nur korrekt, wenn er zyklisch aufgerufen wird.

AX_FB_TP ist ein standardisierter Timer-Funktionsblock gemäß IEC 61131-3, der einen Impuls (Pulse) mit definierter Länge erzeugt. Bei einer steigenden Flanke am Eingang wird ein Ausgangsimpuls für die festgelegte Zeitdauer generiert.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Event): Normal Execution Request (übernimmt den Wert von PT); ruft den Baustein auf (muss zyklisch aufgerufen werden)

Ereignis-Ausgänge

- **CNF** (Event): Execution Confirmation (mit ET)

Daten-Eingänge

- **PT** (TIME): Pulszeit - definiert die Dauer des Ausgangsimpulses

Daten-Ausgänge

- **ET** (TIME): Abgelaufene Zeit (Elapsed Time)

Adapter

- **IN** (Socket): Empfängt Eingangsereignisse und -daten (Typ: `adapter::types::unidirectional::AX`)
- **Q** (Plug): Sendet Ausgangsereignisse und -daten (Typ: `adapter::types::unidirectional::AX`)

Funktionsweise

Der AX_FB_TP-Block arbeitet als Impulsgeber:

1. Bei einer steigenden Flanke (Wechsel von FALSE auf TRUE) am IN-Adapter wechselt der Ausgang am Q-Adapter sofort auf TRUE.
2. Der interne Timer startet und läuft für die Dauer PT.
3. Der Ausgang am Q-Adapter bleibt für die gesamte Zeit PT auf TRUE, unabhängig davon, ob das Eingangssignal am IN-Adapter zwischenzeitlich wieder auf FALSE wechselt.
4. Während der Impuls läuft (solange ET < PT), haben weitere Flanken am Eingang keinen Einfluss auf den laufenden Timer (nicht retriggerbar).
5. Nach Ablauf der Zeit PT wechselt der Ausgang am Q-Adapter zurück auf FALSE.
6. Der Datenausgang ET zeigt die Zeit an, die seit dem Start des Impulses vergangen ist.

Technische Besonderheiten

- Implementiert auf Basis des IEC 61131-3 TP-Standards
- Verwendet unidirektionale Adapter für eine saubere Kapselung in adapterbasierten Architekturen
- Nicht retriggerbar während der aktiven Pulsphase
- Präzise Zeitsteuerung durch Verwendung des TIME-Datentyps

Zustandsübersicht

1. **Ruhezustand**: IN ist FALSE, Q ist FALSE, ET ist 0.
2. **Impulsphase**: Steigende Flanke an IN detektiert, Q ist TRUE, ET steigt bis PT.
3. **Abschluss**: ET hat PT erreicht, Q wechselt auf FALSE.
4. **Warten auf Reset**: Wenn IN noch TRUE ist, bleibt Q FALSE und ET bleibt auf PT, bis IN wieder FALSE wird.

Anwendungsszenarien

- Erzeugung von zeitlich begrenzten Steuersignalen (z. B. Startimpuls für einen Motor)
- Zeitliche Begrenzung von Bedienvorgängen
- Generierung von definierten Signalpulsen in der Prozessautomatisierung

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zum AX_PULSE bietet AX_FB_TP die klassische IEC 61131-3 TP-Charakteristik inklusive dem ET-Statusausgang.

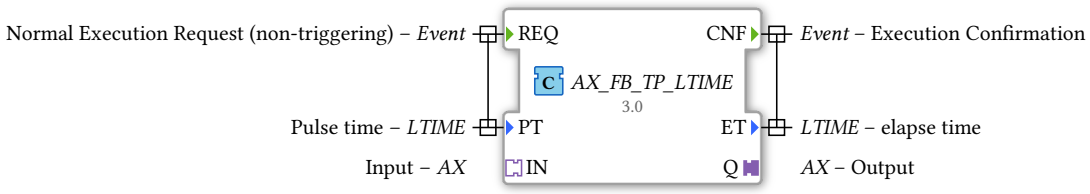
🛠️ Zugehörige Übungen

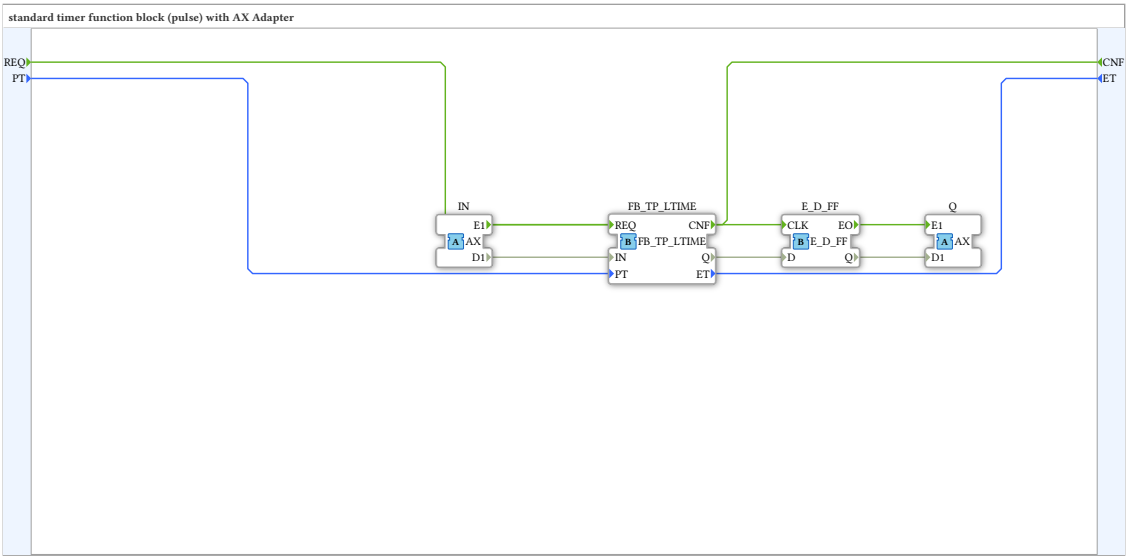
- Uebung_020f2_AX

Fazit

AX_FB_TP ist der Standardbaustein für die Erzeugung stabiler, zeitlich exakt definierter Impulse in einer adapterbasierten Umgebung.

AX_FB_TP_LTIME





• – ○ • – ○ • – ○ •

Einleitung

Der **AX_FB_TP_LTIME** ist ein standardisierter Puls-Timer-Funktionsblock (Timer-Pulse) gemäß IEC 61131-3, erweitert um einen uni-direktionalen **AX-Adapter**. Er wird in der 4diac-IDE verwendet, um zeitlich begrenzte Impulse zu erzeugen und die Ausgangssignale über einen Adapter an die Anwendung zu übergeben. Das Verhalten entspricht einem **TP**-Timer, der bei einem steigenden Flanke am Eingang den Ausgang für eine vorgegebene Zeit auf TRUE setzt.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
REQ	Event	Normaler Ausführungsanforderung (nicht-triggernd)

Ereignis-Ausgänge

Name	Typ	Kommentar
CNF	Event	Ausführungsbestätigung

Daten-Eingänge

Name	Typ	Kommentar
PT	LTIME	Pulszeit (Impulsdauer)

Daten-Ausgänge

Name	Typ	Kommentar
ET	LTIME	Verstrichene Zeit (Elapsed Time)

Adapter

- **Stecker (Plugs):**
  - **Q**: Ausgangsadapter vom Typ `adapter::types::unidirectional::AX`. Über diesen Adapter wird das boolesche Ausgangssignal `Q.D1` sowie das Ereignis `Q.E1` an die Anwendung weitergegeben.
- **Buchse (Sockets):**

– **IN**: Eingangsadapter vom Typ `adapter::types::unidirectional::AX`. Über diesen Adapter empfängt der Baustein das Start-Ereignis (über `IN.E1`) und ggf. ein boolesches Signal (über `IN.D1`), das intern als Freigabe oder Zusatzbedingung genutzt wird (hier nicht direkt mit der Timerlogik verbunden, sondern zu einem internen Standard-Timer geführt).

Funktionsweise

Der Baustein besteht intern aus zwei Komponenten:

1. **FB_TP_LTIME** – ein reiner Puls-Timer (IEC 61131-3 TP) mit Zeitbasis LTIME.
2. **E_D_FF** – ein flankengesteuertes D-Flipflop (E-D-FF gemäß IEC 61499), das den Q-Ausgang des Timers zwischenspeichert.

Ablauf:

- Ein Ereignis über den Adaptereingang `IN.E1` oder den Standard-Eventeingang `REQ` startet den Timer `FB_TP_LTIME` (die Ereignisse sind intern zusammengeführt – beide lösen denselben Start aus).
- Der Timer zählt die verstrichene Zeit `ET` hoch und setzt seinen internen boolschen Ausgang `Q` auf TRUE, sobald `PT` erreicht ist. Nach Ablauf der Pulszeit wird `Q` wieder FALSE und ein `CNF`-Ereignis ausgegeben.
- Das `CNF`-Ereignis taktet das D-Flipflop `E_D_FF`, welches den aktuellen Zustand von `Q` (also TRUE während der Pulsdauer, FALSE danach) übernimmt und am Ausgang `Q.D1` des Adapters bereitstellt.
- Gleichzeitig wird das `CNF`-Ereignis auch direkt an den Ausgang `CNF` des Gesamtbausteins weitergeleitet, sodass die Anwendung über das Ende des Impulses informiert wird.
- Der Ausgang `ET` gibt die aktuell verstrichene Zeit an und ist direkt mit dem Ausgang des internen Timers verbunden.

Technische Besonderheiten

- **Adapter-Schnittstelle:** Der FB kapselt die Timerlogik in eine Adapter-Schnittstelle, sodass er in modularen, adapterbasierten Architekturen einfach eingebunden werden kann.
- **D-Flipflop zur Signalhaltung:** Der Q-Ausgang des Timers wird über ein Flipflop synchron zum `CNF`-Ereignis übernommen, um Glitches oder asynchrone Änderungen zu vermeiden.
- **Doppelter Ereignispfad:** Der Start des Timers kann sowohl über den Adapter (`IN.E1`) als auch über den Standard-Eventeingang `REQ` erfolgen – beide sind intern verbunden, was eine flexible Ansteuerung erlaubt.
- **LTIME-Unterstützung:** Die Pulszeit wird als `LTIME` (Long Time, 64-Bit Nanosekunden) verarbeitet, was eine extrem hohe Zeitauflösung und große Zeitbereiche ermöglicht.

Zustandsübersicht

Da der Funktionsblock aus zwei Teilen besteht, ergibt sich folgendes Zustandsverhalten für den Gesamtbaustein:

Zustand	Beschreibung
IDLE	Kein Impuls aktiv. Der Adapterausgang `Q.D1` ist FALSE.
TIMING	Ein Puls läuft – der interne Timer zählt hoch. `ET` steigt, `Q.D1` ist TRUE (sobald das Flipflop beim Start oder nach dem ersten CNF getaktet wurde).
DONE	Der Puls ist abgeschlossen. `CNF` wurde ausgegeben, `ET` ist gleich `PT`, `Q.D1` wird mit dem nächsten `CNF` auf FALSE gesetzt. Bei einem neuen `REQ` oder `IN.E1` beginnt der Zyklus von vorn.

Anwendungsszenarien

- **Impulsausgabe mit definierter Länge:** Steuerung von Aktoren (z. B. Ventile, Lampen), die für eine bestimmte Zeit eingeschaltet werden sollen.
- **Adapter-basierte Subsysteme:** Einbinden des Timers in eine modulare Automatisierungslösung, bei der Ein- und Ausgänge über Adapter (AX-Protokoll) kommunizieren.
- **Zeitmessung mit hoher Auflösung:** Nutzung von `LTIME` für präzise Zeitsteuerungen im Mikro- oder Nanosekundenbereich.

Vergleich mit ähnlichen Bausteinen

Baustein	Typ	Besonderheit

FB_TP_LTIME (Standard)	TP-Timer	Kein Adapter, nur Standard-Events und Signale.
AX_FB_TP_LTIME (dieser FB)	TP-Timer mit AX-Adapter	Ermöglicht Einbindung in adapterbasierte Architekturen, enthält D-Flipflop zur Signalstabilisierung.
FB_TON_LTIME	Einschaltverzögerung	Verzögert das Einschalten eines Ausgangs.
FB_TOF_LTIME	Ausschaltverzögerung	Verzögert das Ausschalten eines Ausgangs.

Der vorliegende Baustein kombiniert die Timerfunktion eines TP mit der Flexibilität eines Adapters.

Fazit

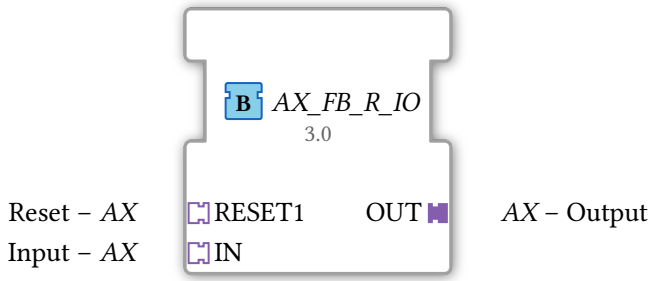
Der **AX_FB_TP_LTIME** ist ein robuster und präziser Puls-Timer für die 4diac-IDE, der durch die Adapter-Schnittstelle und die Verwendung von `LTIME` besonders geeignet für hochauflösende, modulare Automatisierungslösungen ist. Die interne Flipflop-Struktur sorgt für saubere Signalübergaben, und die doppelte Ereignisbeschaltung bietet Flexibilität in der Ansteuerung.

Passende Themen-Unterseiten auf ms-muc-docs.de

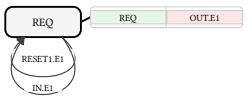
- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.1.11.12 monostableElements

AX_FB_R_IO



ECC: AX_FB_R_IO



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AX_FB_R_IO** realisiert ein monostabiles, reset-dominantes Verhalten. Er verfügt über drei Adapter-Schnittstellen, über die er ein Eingangssignal (IN) an den Ausgang (OUT) weiterleitet, sofern kein aktives Rücksetzsignal (RESET1) anliegt. Der Baustein arbeitet nach dem Prinzip: **Reset hat Priorität**.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Bezeichnung	Typ	Richtung	Kommentar
IN	unidirectional::AX	Socket	Eingangssignal
RESET1	unidirectional::AX	Socket	Rücksetzsignal (priorisiert)
OUT	unidirectional::AX	Plug	Ausgangssignal

Die Adapter vom Typ `adapter::types::unidirectional::AX` verfügen jeweils über einen Ereignisausgang `E1` und einen Datenausgang `D1` (vom Typ BOOL). Der Datenwert des Adapters wird über `D1` bereitgestellt, die Verarbeitung wird durch das Ereignis `E1` ausgelöst.

Funktionsweise

Die Kernlogik des Bausteins wird durch eine einfache Zustandsmaschine mit einem einzigen Zustand `REQ` abgebildet. Sobald ein Ereignis an einem der beiden Eingangsadapter (`IN.E1` oder `RESET1.E1`) eintrifft, wird der Algorithmus `REQ` ausgeführt. Dieser Algorithmus prüft den Rücksetzeingang:

- **Wenn `RESET1.D1 = TRUE`**: Der Ausgang `OUT.D1` wird auf `FALSE` gesetzt, unabhängig vom Eingangssignal.
- **Andernfalls**: Der Ausgang `OUT.D1` erhält direkt den Wert des Eingangssignals `IN.D1`.

Das Ereignis am Ausgangsadapter (`OUT.E1`) wird bei jeder Ausführung des Algorithmus ausgelöst, sodass nachfolgende Bausteine über die Wertänderung informiert werden.

Technische Besonderheiten

- **Reset-Dominanz**: Das Rücksetzsignal `RESET1` hat stets Vorrang vor dem Eingangssignal `IN`. Solange `RESET1.D1` aktiv ist, bleibt der Ausgang `FALSE`.
- **Monostabiles Verhalten**: Der Baustein kehrt nach jeder Verarbeitung automatisch in den Bereitschaftszustand zurück (ein Zustand). Es gibt keinen gespeicherten Zustand, jede Ausführung reagiert sofort auf die aktuellen Eingänge.
- **Adapterbasierte Kommunikation**: Sämtliche Ein- und Ausgänge erfolgen über bidirektionale Adapter, die sowohl Daten als auch Ereignisse transportieren. Die Ereignissteuerung erfolgt implizit über die Adapter-Protokolle.
- **Paketname**: `adapter::monostableElements` (definiert im CompilerInfo).

Zustandsübersicht

Der Funktionsblock besitzt genau einen Zustand:

Zustand	Beschreibung
REQ	Start- und Endzustand. Wartet auf ein Ereignis von `IN` oder `RESET1`. Nach Ausführung des Algorithmus kehrt der Zustand immer zu `REQ` zurück (keine weiteren Zustände).

Es gibt zwei Transitionen:

- Von `REQ` zu `REQ` bei Ereignis `IN.E1`
- Von `REQ` zu `REQ` bei Ereignis `RESET1.E1`

Beide Transitionen führen zurück in den gleichen Zustand und lösen die Ausführung des Algorithmus aus.

Anwendungsszenarien

- **Freigabeschaltung in der Automatisierungstechnik**: Ein Signal (z. B. Maschinenstart) wird nur dann an den Ausgang weitergegeben, wenn kein Rücksetzkommando (z. B. Stopp) aktiv ist.
- **Sicherheitsverriegelung**: In einer Steuerungskette wird ein Rücksetzsignal prioritär behandelt, um gefährliche Zustände zuverlässig zu eliminieren.
- **Signalweiche mit Priorität**: Zwei Quellen konkurrieren um einen Ausgang – hier hat die Rücksetzfunktion stets Vorrang.

Vergleich mit ähnlichen Bausteinen

- **AX_FB_RS** (Set-Dominant): Im Gegensatz zu diesem Baustein hat dort der Set-Eingang Priorität. `AX_FB_R_IO` ist reset-dominant.
- **AX_FB_SR**: Set-Reset-Flipflop mit gleichberechtigten Eingängen, oft ohne Priorität (kann zu undefinierten Zuständen führen). `AX_FB_R_IO` vermeidet dies durch klare Dominanz.
- **AX_FB_MONOSTABLE**: Reine monostabile Kippglieder mit Zeitparametern; `AX_FB_R_IO` hat kein Zeitverhalten, sondern reagiert sofort auf Signale.

Fazit

Der Funktionsblock `AX_FB_R_IO` ist ein einfacher, aber zuverlässiger Baustein zur realisierungsgetreuen Abbildung eines priorisierten Rücksetzverhaltens. Dank der Adapter-Schnittstelle fügt er sich nahtlos in ereignisgesteuerte Automatisierungssysteme ein. Die klare Reset-Dominanz macht ihn besonders geeignet für Sicherheits- und Freigabefunktionen, bei denen ein Rücksetzsignal unbedingten Vorrang haben muss.

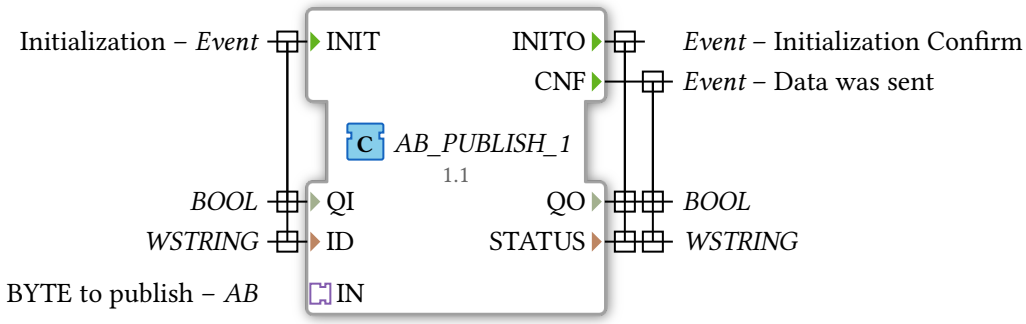
2.1.11.13 net

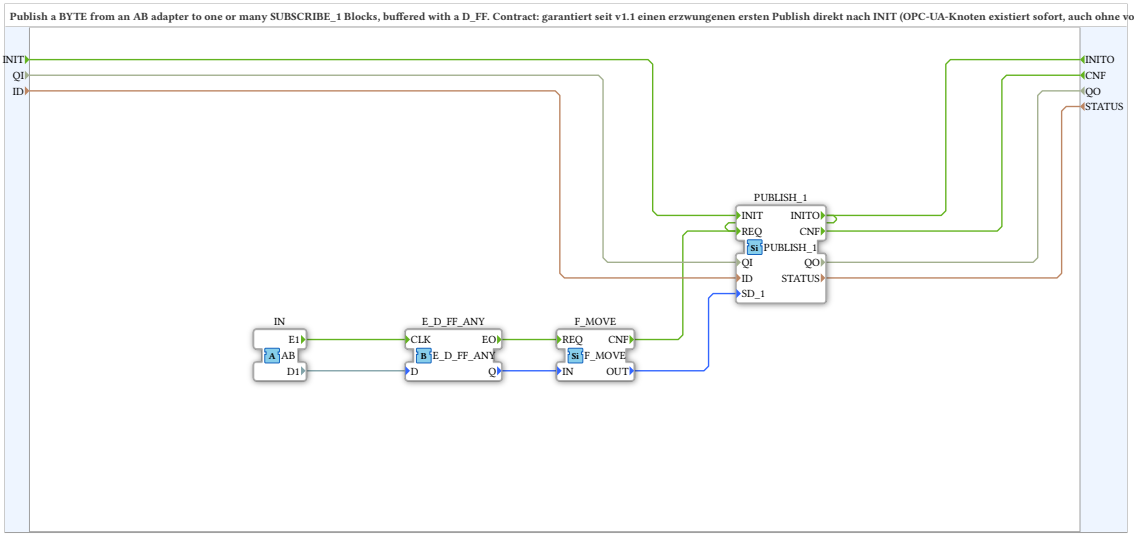
Hier finden Sie Funktionsbausteine, die die Netzwerk-Kommunikation (Publish/Subscribe sowie OPC-UA Client/Server) auf die unidirektionalen Adapter abbilden. Die Bausteine kapseln die entsprechenden `iec61499::net`-Funktionsbausteine und puffern die übertragenen Werte mit einem D-Flipflop.

- AB_PUBLISH_1
- AB_SUBSCRIBE_1
- AD_PUBLISH_1
- AD_SUBSCRIBE_1
- ADI_PUBLISH_1
- ADI_SUBSCRIBE_1
- AI_PUBLISH_1
- AI_SUBSCRIBE_1
- AIS_PUBLISH_1
- AIS_SUBSCRIBE_1
- AIWS_PUBLISH_1
- AIWS_SUBSCRIBE_1
- AL_PUBLISH_1
- AL_SUBSCRIBE_1
- ALI_PUBLISH_1
- ALI_SUBSCRIBE_1
- ALR_PUBLISH_1
- ALR_SUBSCRIBE_1
- AQ_PUBLISH_1
- AQ_SUBSCRIBE_1
- AR_CLIENT_1_0
- AR_CLIENT_1_0_UNGATED
- AR_PUBLISH_1
- AR_SERVER_0_1
- AR_SUBSCRIBE_1
- AS_PUBLISH_1
- AS_SUBSCRIBE_1
- ATM_CLIENT_1_0
- ATM_PUBLISH_1
- ATM_SERVER_0_1
- ATM_SUBSCRIBE_1
- AUDI_PUBLISH_1
- AUDI_SUBSCRIBE_1
- AUI_PUBLISH_1
- AUI_SUBSCRIBE_1
- AULI_PUBLISH_1
- AULI_SUBSCRIBE_1
- AUS_PUBLISH_1

- AUS_SUBSCRIBE_1
- AW_PUBLISH_1
- AW_SUBSCRIBE_1
- AX_CLIENT_1_0
- AX_PUBLISH_1
- AX_SERVER_0_1
- AX_SUBSCRIBE_1
- AE_AX_CLIENT_0_SUBSCRIBE_1
- AE_AX_SERVER_0_CLIENT_1_0
- AE_CLIENT_0
- ASR_AX_CLIENT_0_SUBSCRIBE_1
- ASR_AX_SERVER_0_CLIENT_1_0
- ASR_CLIENT_0
- ASRT_AX_CLIENT_0_SUBSCRIBE_1
- ASRT_AX_SERVER_0_CLIENT_1_0
- ASRT_CLIENT_0

AB_PUBLISH_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AB_PUBLISH_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `PUBLISH_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf

einen unidirektionalen **AB-Adapter** abbildet. Ein am Adapter-Socket `IN` anliegender **BYTE**-Wert wird über ein internes D-Flipflop (E_D_FF_ANY) gepuffert und anschließend über den konfigurierten Publish-Kanal an alle zugehörigen `SUBSCRIBE_1`-Empfänger gesendet.

Schnittstellenstruktur

Ereignis-Eingänge
- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`

Ereignis-Ausgänge
- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`
- **CNF** (Event): Bestätigung, dass die Daten gesendet wurden, verbunden mit `QO` und `STATUS`

Daten-Eingänge
- **QI** (BOOL): Qualifier-Eingang, aktiviert den Publish-Dienst
- **ID** (WSTRING): Identifikator des Publish-Kanals

Daten-Ausgänge
- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::AB	Socket (Eingang)	Zu veröffentlichender BYTE-Wert

Funktionsweise
1. Über das `INIT`-Ereignis wird der interne `PUBLISH_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Sobald der AB-Socket `IN` ein Ereignis auf `IN.E1` liefert, wird der gleichzeitig an `IN.D1` anliegende BYTE-Wert in das interne **E_D_FF_ANY**-Flipflop übernommen (`CLK` bzw. `D`).
3. Das Flipflop hält den Wert stabil an seinem Ausgang `Q` und erzeugt bei einer Wertänderung das Ereignis `EO`.
4. `EO` löst das Sendeereignis `REQ` des internen `PUBLISH_1`-Blocks aus; der gepufferte Wert `Q` wird über die Datenverbindung als `SD_1` übergeben und auf dem konfigurierten Kanal veröffentlicht.
5. Nach erfolgreichem Senden bestätigt der `PUBLISH_1`-Block mit `CNF`, das nach außen geführt wird.

Technische Besonderheiten
- **Pufferung mit D-Flipflop**: Der zu sendende BYTE-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert. Dadurch wird verhindert, dass ein sich während der Verarbeitung ändernder Eingangswert den Sendevorgang stört.
- **Ereignisbasierte Entkopplung**: Durch das echte ereignisbasierte `E_D_FF_ANY` werden unmittelbare Rückkopplungsschleifen zwischen `PUBLISH_1` und `SUBSCRIBE_1` (z. B. auf demselben OPC-UA-Knoten) entkoppelt – der saubere Fix gegenüber dem Interim-Fix mit `AX_D_FF`.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `PUBLISH_1` (inkl. `REQ`/`SD_1`) wird nach innen verlegt; nach außen ist nur noch die AB-Adapter-Schnittstelle sichtbar.

Zustandsübersicht
1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Der Block ist bereit, Daten über den konfigurierten Kanal zu veröffentlichen.
3. **Sendeaktiv**: Ein am AB-Socket eintreffendes Ereignis puffert den Wert und veranlasst das Senden über den internen `PUBLISH_1`-Block.

Anwendungsszenarien
- **Netzwerkkopplung in AB-Anwendungen**: Senden von BYTE-Signalen (z. B. Schaltbefehle oder Statusbits) aus der Adapter-basierten Logik über Publish/Subscribe an entfernte Empfänger.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_SUBSCRIBE_1` werden Rückkopplungen zwischen eigenen Veröffentlichungen und dem eigenen Abonnement entkoppelt.

- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AB-Adapter setzen.

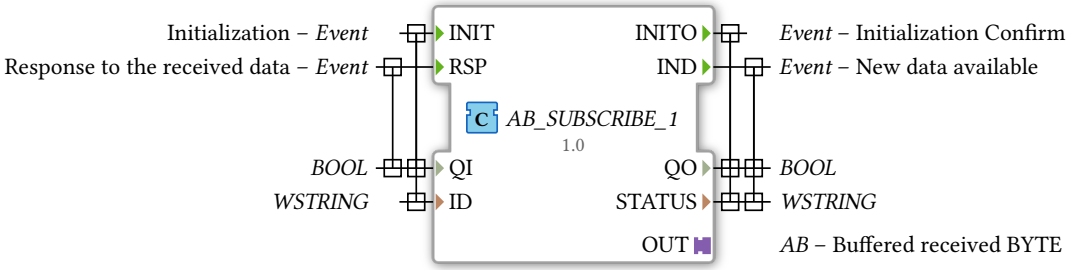
Vergleich mit ähnlichen Bausteinen

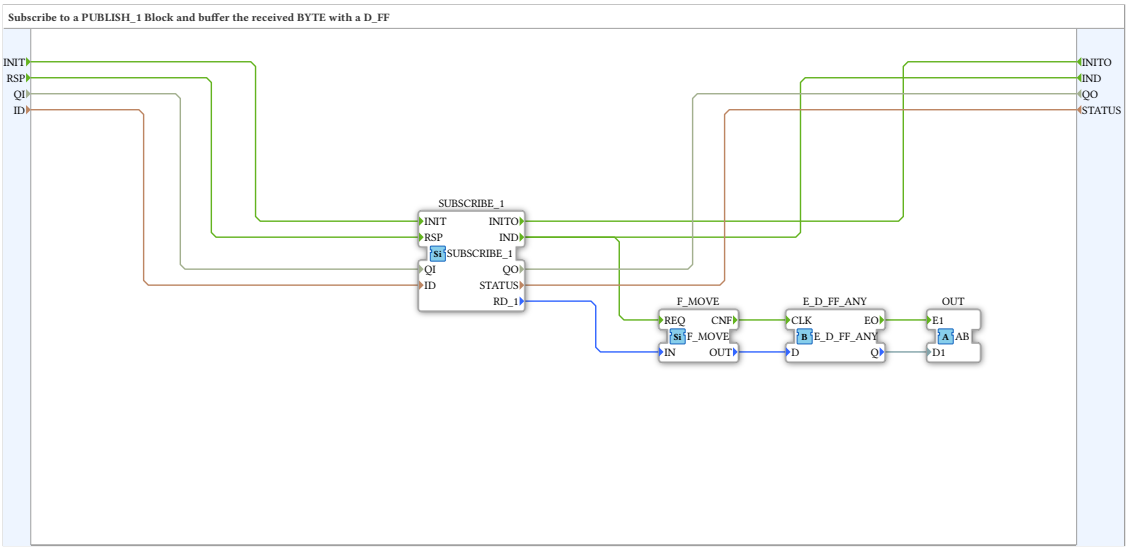
- **PUBLISH_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`REQ`, `SD_1`, ...). **AB_PUBLISH_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den AB-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen BYTE-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AB_PUBLISH_1** vereint Puffern und Veröffentlichen in einem Baustein.

Fazit

AB_PUBLISH_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `PUBLISH_1` mit der Adapter-basierten BYTE-Verarbeitung. Die interne D-Flipflop-Pufferung stabilisiert den Sendewert und entkoppelt Rückkopplungsschleifen; die reduzierte AB-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

AB_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AB_SUBSCRIBE_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `SUBSCRIBE_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AB-Adapter** abbildet. Er abonniert einen über die `ID` konfigurierten Publish-Kanal und stellt den empfangenen **BYTE**-Wert gepuffert über einen D-Flipflop (E_D_FF_ANY) am Adapter-Plug `OUT` bereit. Damit lassen sich Netzkommunikation und BYTE-Signalverarbeitung in der AB-Adapter-Welt sauber verbinden.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`
- **RSP** (Event): Response-Ereignis, verbunden mit `QI`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Abonnement-Dienst
- **ID** (WSTRING): Identifikator des zu abonnierenden Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
OUT	adapter::types::unidirectional::AB	Plug (Ausgang)	Gepufferter empfangener BYTE-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `SUBSCRIBE_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Der interne `SUBSCRIBE_1`-Block überwacht den konfigurierten Kanal. Sobald der zugehörige `PUBLISH_1`-Block Daten veröffentlicht, erzeugt er das Indikationsereignis `IND` und legt den empfangenen Wert an `RD_1` an.
3. `IND` taktet den internen **E_D_FF_ANY**-Flipflop (`CLK`), der den Wert von `RD_1` an seinem Dateneingang `D` übernimmt.
4. Der gespeicherte Wert erscheint stabil am Ausgang `Q` und wird über die Datenverbindung auf `OUT.D1` des AX-Plugs gelegt.
5. Das Ereignis `EO` des Flipflops wird auf den Ereignis-Ausgang `OUT.E1` gelegt, sodass nachgeschaltete Bausteine über die neue Information benachrichtigt werden.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der empfangene BYTE-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert und bleibt stabil am Ausgang, bis der nächste Wert empfangen wird. `EO` feuert nur bei einer Änderung von `Q`.
- **RSP ist ein No-Op bei Subscribern**: Laut Quellcode-Analyse von `4diac-forte` (`CCommFB::sendData()`) führt das `RSP`-Ereignis bei `SUBSCRIBE_1` (Service-Typ Subscriber) keine Aktion aus – der Zweig `!= e_Subscriber` wird übersprungen. Das Event-Input-Paar `INIT`/`RSP` wird nur für die Quittierung durchgeführt und ist für die Quittierung im Subscriber-Fall funktionslos. Es wird dennoch nach außen geführt, damit ein konfigurierbarer Anschluss besteht.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `SUBSCRIBE_1` (inkl. `IND`/`RD_1`) wird nach innen verlegt; nach außen ist nur noch die AB-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Das Abonnement ist aktiv; der Block wartet auf Daten vom Publisher.
3. **Datenempfang**: Bei `IND` wird der empfangene BYTE-Wert in das interne D-Flipflop übernommen und über `OUT` ausgegeben.

Anwendungsszenarien

- **Netzwerkkopplung in AB-Anwendungen**: Empfang von BYTE-Signalen (z. B. Schaltbefehle oder Statusbits) über Publish/Subscribe und Weiterleitung über AB-Adapter an die Adapter-basierte Logik.

- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_PUBLISH_1` und einem echten ereignisbasierten `E_D_FF_ANY` lässt sich die Rückkopplung zwischen `SUBSCRIBE_1` und `PUBLISH_1` entkoppeln (sauberer Fix gegenüber dem Interim-Fix mit `AX_D_FF`).
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AB-Adapter setzen.

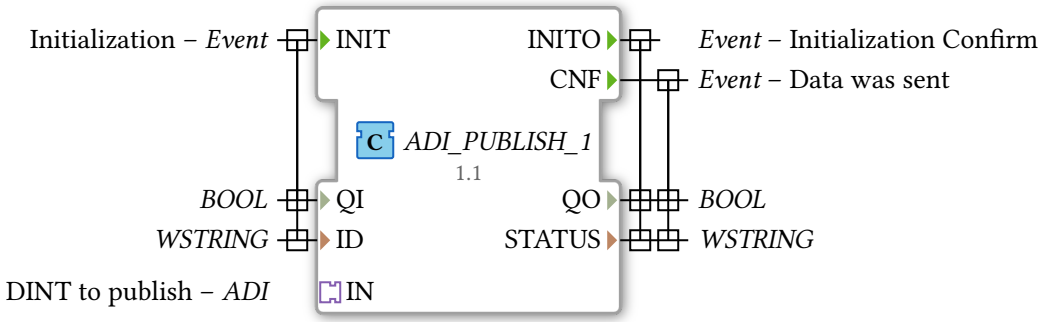
Vergleich mit ähnlichen Bausteinen

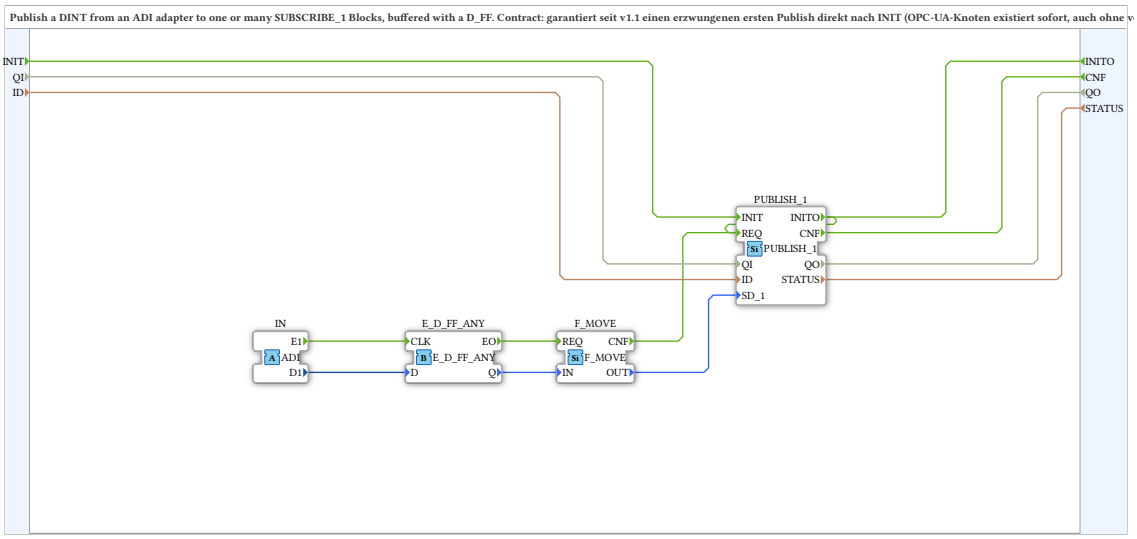
- **SUBSCRIBE_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`IND`, `RD_1`, ...). **AB_SUBSCRIBE_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den AB-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen BYTE-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AB_SUBSCRIBE_1** vereint Subscribing und Pufferung in einem Baustein.

Fazit

AB_SUBSCRIBE_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `SUBSCRIBE_1` mit der Adapter-basierten BYTE-Verarbeitung. Die interne D-Flipflop-Pufferung hält den empfangenen Wert stabil, bis neue Daten eintreffen, und die reduzierte AB-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

ADI_PUBLISH_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ADI_PUBLISH_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `PUBLISH_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **ADI-Adapter** abbildet. Ein am Adapter-Socket `IN` anliegender **DINT**-Wert wird über ein internes D-Flipflop (E_D_FF_ANY) gepuffert und anschließend über den konfigurierten Publish-Kanal an alle zugehörigen `SUBSCRIBE_1`-Empfänger gesendet.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`
- **CNF** (Event): Bestätigung, dass die Daten gesendet wurden, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Publish-Dienst
- **ID** (WSTRING): Identifikator des Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::ADI	Socket (Eingang)	Zu veröffentlichender DINT-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `PUBLISH_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Sobald der ADI-Socket `IN` ein Ereignis auf `IN.E1` liefert, wird der gleichzeitig an `IN.D1` anliegende DINT-Wert in das interne **E_D_FF_ANY**-Flipflop übernommen (`CLK` bzw. `D`).
3. Das Flipflop hält den Wert stabil an seinem Ausgang `Q` und erzeugt bei einer Wertänderung das Ereignis `EO`.
4. `EO` löst das Sendeereignis `REQ` des internen `PUBLISH_1`-Blocks aus; der gepufferte Wert `Q` wird über die Datenverbindung als `SD_1` übergeben und auf dem konfigurierten Kanal veröffentlicht.
5. Nach erfolgreichem Senden bestätigt der `PUBLISH_1`-Block mit `CNF`, das nach außen geführt wird.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der zu sendende DINT-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert. Dadurch wird verhindert, dass ein sich während der Verarbeitung ändernder Eingangswert den Sendevorgang stört.
- **Ereignisbasierte Entkopplung**: Durch das echte ereignisbasierte `E_D_FF_ANY` werden unmittelbare Rückkopplungsschleifen zwischen `PUBLISH_1` und `SUBSCRIBE_1` (z. B. auf demselben OPC-UA-Knoten) entkoppelt – der saubere Fix gegenüber dem Interim-Fix mit `AX_D_FF`.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `PUBLISH_1` (inkl. `REQ`/`SD_1`) wird nach innen verlegt; nach außen ist nur noch die ADI-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Der Block ist bereit, Daten über den konfigurierten Kanal zu veröffentlichen.
3. **Sendeaktiv**: Ein am ADI-Socket eintreffendes Ereignis puffert den Wert und veranlasst das Senden über den internen `PUBLISH_1`-Block.

Anwendungsszenarien

- **Netzwerkkopplung in ADI-Anwendungen**: Senden von DINT-Signalen (z. B. Schaltbefehle oder Statusbits) aus der Adapter-basierten Logik über Publish/Subscribe an entfernte Empfänger.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_SUBSCRIBE_1` werden Rückkopplungen zwischen eigenen Veröffentlichungen und dem eigenen Abonnement entkoppelt.
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf ADI-Adapter setzen.

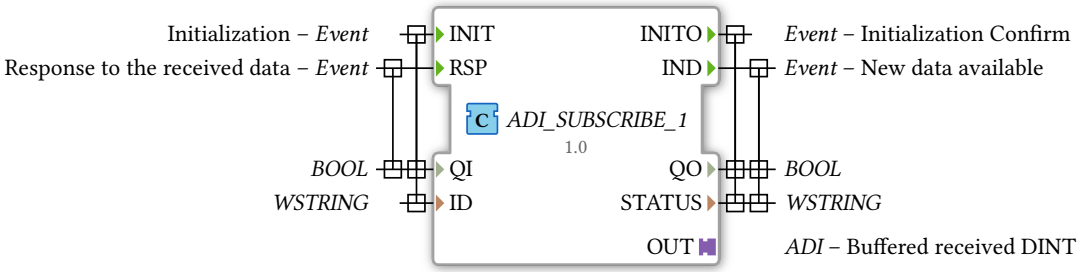
Vergleich mit ähnlichen Bausteinen

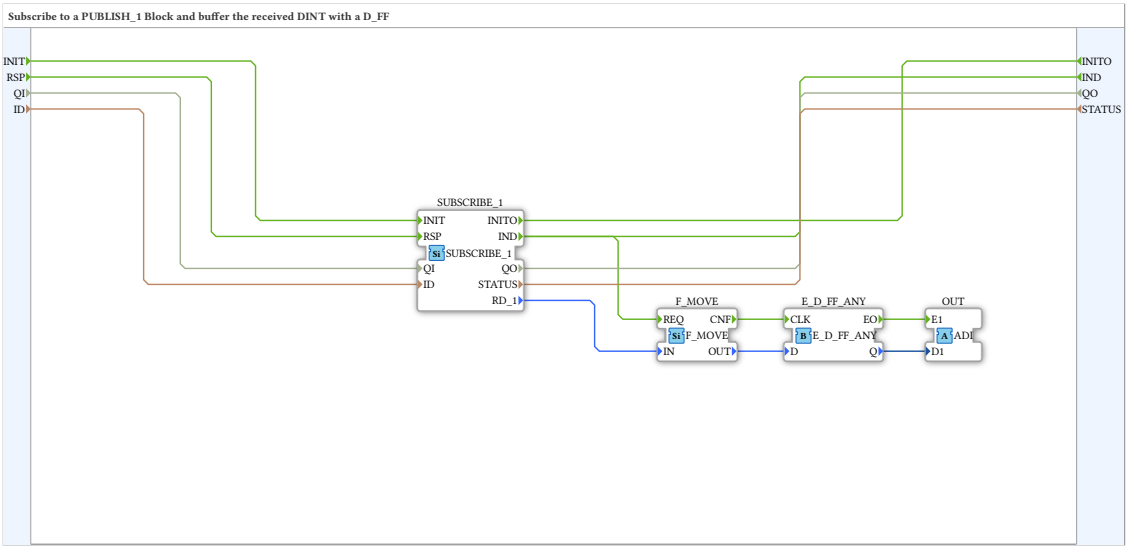
- **PUBLISH_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`REQ`, `SD_1`, ...). **ADI_PUBLISH_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den ADI-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen DINT-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **ADI_PUBLISH_1** vereint Puffern und Veröffentlichen in einem Baustein.

Fazit

ADI_PUBLISH_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `PUBLISH_1` mit der Adapter-basierten DINT-Verarbeitung. Die interne D-Flipflop-Pufferung stabilisiert den Sendewert und entkoppelt Rückkopplungsschleifen; die reduzierte ADI-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

ADI_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ADI_SUBSCRIBE_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `SUBSCRIBE_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **ADI-Adapter** abbildet. Er abonniert einen über die `ID` konfigurierten Publish-Kanal und stellt den empfangenen **DINT**-Wert gepuffert über einen D-Flipflop (E_D_FF_ANY) am Adapter-Plug `OUT` bereit. Damit lassen sich Netzkommunikation und DINT-Signalverarbeitung in der ADI-Adapter-Welt sauber verbinden.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`
- **RSP** (Event): Response-Ereignis, verbunden mit `QI`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Abonnement-Dienst
- **ID** (WSTRING): Identifikator des zu abonnierenden Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
OUT	adapter::types::unidirectional::ADI	Plug (Ausgang)	Gepufferter empfangener DINT-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `SUBSCRIBE_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Der interne `SUBSCRIBE_1`-Block überwacht den konfigurierten Kanal. Sobald der zugehörige `PUBLISH_1`-Block Daten veröffentlicht, erzeugt er das Indikationsereignis `IND` und legt den empfangenen Wert an `RD_1` an.
3. `IND` taktet den internen **E_D_FF_ANY**-Flipflop (`CLK`), der den Wert von `RD_1` an seinem Dateneingang `D` übernimmt.
4. Der gespeicherte Wert erscheint stabil am Ausgang `Q` und wird über die Datenverbindung auf `OUT.D1` des AX-Plugs gelegt.
5. Das Ereignis `EO` des Flipflops wird auf den Ereignis-Ausgang `OUT.E1` gelegt, sodass nachgeschaltete Bausteine über die neue Information benachrichtigt werden.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der empfangene DINT-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert und bleibt stabil am Ausgang, bis der nächste Wert empfangen wird. `EO` feuert nur bei einer Änderung von `Q`.
- **RSP ist ein No-Op bei Subscribern**: Laut Quellcode-Analyse von `4diac-forte` (`CCommFB::sendData()`) führt das `RSP`-Ereignis bei `SUBSCRIBE_1` (Service-Typ Subscriber) keine Aktion aus – der Zweig `!= e_Subscriber` wird übersprungen. Das Event-Input-Paar `INIT`/`RSP` wird nur für die Quittierung durchgeführt und ist für die Quittierung im Subscriber-Fall funktionslos. Es wird dennoch nach außen geführt, damit ein konfigurierbarer Anschluss besteht.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `SUBSCRIBE_1` (inkl. `IND`/`RD_1`) wird nach innen verlegt; nach außen ist nur noch die ADI-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Das Abonnement ist aktiv; der Block wartet auf Daten vom Publisher.
3. **Datenempfang**: Bei `IND` wird der empfangene DINT-Wert in das interne D-Flipflop übernommen und über `OUT` ausgegeben.

Anwendungsszenarien

- **Netzwerkkopplung in ADI-Anwendungen**: Empfang von DINT-Signalen (z. B. Schaltbefehle oder Statusbits) über Publish/Subscribe und Weiterleitung über ADI-Adapter an die Adapter-basierte Logik.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_PUBLISH_1` und einem echten ereignisbasierten `E_D_FF_ANY` lässt sich die Rückkopplung zwischen `SUBSCRIBE_1` und `PUBLISH_1` entkoppeln (sauberer Fix gegenüber dem Interim-Fix mit `AX_D_FF`).
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf ADI-Adapter setzen.

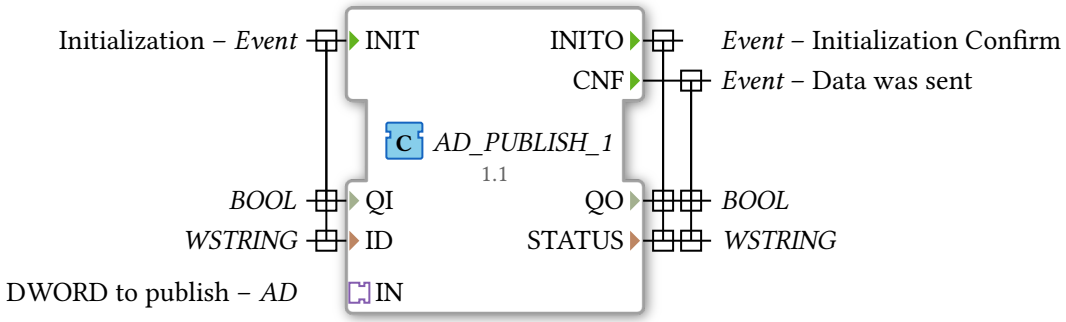
Vergleich mit ähnlichen Bausteinen

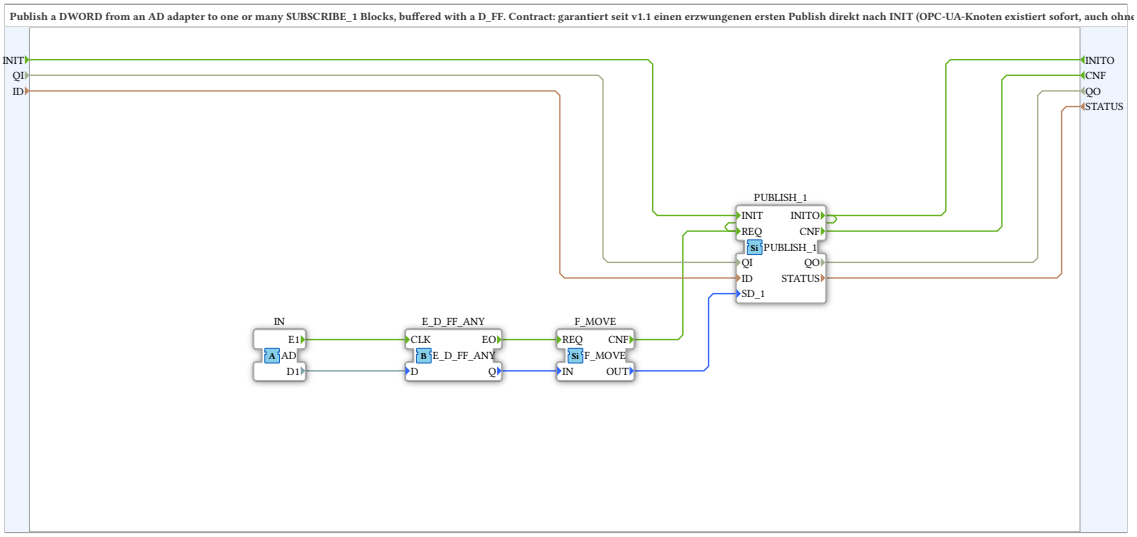
- **SUBSCRIBE_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`IND`, `RD_1`, ...). **ADI_SUBSCRIBE_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den ADI-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen DINT-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **ADI_SUBSCRIBE_1** vereint Subscribing und Pufferung in einem Baustein.

Fazit

ADI_SUBSCRIBE_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `SUBSCRIBE_1` mit der Adapter-basierten DINT-Verarbeitung. Die interne D-Flipflop-Pufferung hält den empfangenen Wert stabil, bis neue Daten eintreffen, und die reduzierte ADI-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

AD_PUBLISH_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AD_PUBLISH_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `PUBLISH_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AD-Adapter** abbildet. Ein am Adapter-Socket `IN` anliegender **DWORD**-Wert wird über ein internes D-Flipflop (E_D_FF_ANY) gepuffert und anschließend über den konfigurierten Publish-Kanal an alle zugehörigen `SUBSCRIBE_1`-Empfänger gesendet.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`
- **CNF** (Event): Bestätigung, dass die Daten gesendet wurden, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Publish-Dienst
- **ID** (WSTRING): Identifikator des Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::AD	Socket (Eingang)	Zu veröffentlichender DWORD-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `PUBLISH_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Sobald der AD-Socket `IN` ein Ereignis auf `IN.E1` liefert, wird der gleichzeitig an `IN.D1` anliegende DWORD-Wert in das interne **E_D_FF_ANY**-Flipflop übernommen (`CLK` bzw. `D`).
3. Das Flipflop hält den Wert stabil an seinem Ausgang `Q` und erzeugt bei einer Wertänderung das Ereignis `EO`.
4. `EO` löst das Sendeereignis `REQ` des internen `PUBLISH_1`-Blocks aus; der gepufferte Wert `Q` wird über die Datenverbindung als `SD_1` übergeben und auf dem konfigurierten Kanal veröffentlicht.
5. Nach erfolgreichem Senden bestätigt der `PUBLISH_1`-Block mit `CNF`, das nach außen geführt wird.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der zu sendende DWORD-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert. Dadurch wird verhindert, dass ein sich während der Verarbeitung ändernder Eingangswert den Sendevorgang stört.
- **Ereignisbasierte Entkopplung**: Durch das echte ereignisbasierte `E_D_FF_ANY` werden unmittelbare Rückkopplungsschleifen zwischen `PUBLISH_1` und `SUBSCRIBE_1` (z. B. auf demselben OPC-UA-Knoten) entkoppelt – der saubere Fix gegenüber dem Interim-Fix mit `AX_D_FF`.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `PUBLISH_1` (inkl. `REQ`/`SD_1`) wird nach innen verlegt; nach außen ist nur noch die AD-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Der Block ist bereit, Daten über den konfigurierten Kanal zu veröffentlichen.
3. **Sendeaktiv**: Ein am AD-Socket eintreffendes Ereignis puffert den Wert und veranlasst das Senden über den internen `PUBLISH_1`-Block.

Anwendungsszenarien

- **Netzwerkkopplung in AD-Anwendungen**: Senden von DWORD-Signalen (z. B. Schaltbefehle oder Statusbits) aus der Adapter-basierten Logik über Publish/Subscribe an entfernte Empfänger.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_SUBSCRIBE_1` werden Rückkopplungen zwischen eigenen Veröffentlichungen und dem eigenen Abonnement entkoppelt.
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AD-Adapter setzen.

Vergleich mit ähnlichen Bausteinen

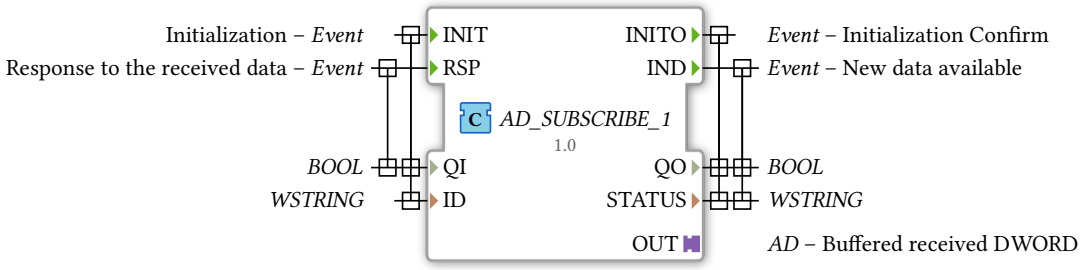
- **PUBLISH_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`REQ`, `SD_1`, …). **AD_PUBLISH_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den AD-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.

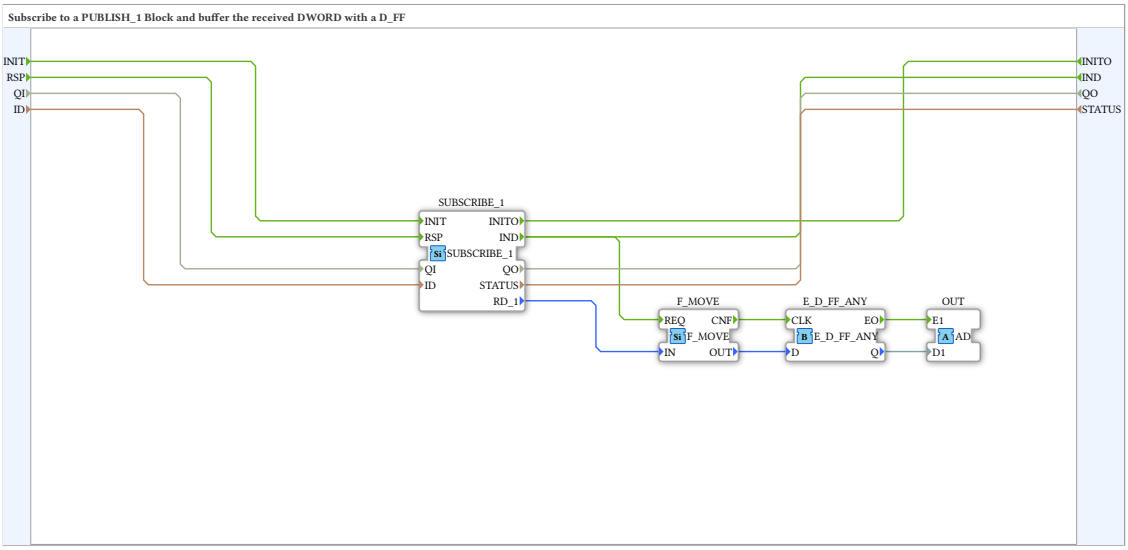
- **AX_D_FF**: Puffert einen DWORD-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AD_PUBLISH_1** vereint Puffern und Veröffentlichen in einem Baustein.

Fazit

AD_PUBLISH_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `PUBLISH_1` mit der Adapter-basierten DWORD-Verarbeitung. Die interne D-Flipflop-Pufferung stabilisiert den Sendewert und entkoppelt Rückkopplungsschleifen; die reduzierte AD-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

AD_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AD_SUBSCRIBE_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `SUBSCRIBE_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AD-Adapter** abbildet. Er abonniert einen über die `ID` konfigurierten Publish-Kanal und stellt den empfangenen **DWORD**-Wert gepuffert über einen D-Flipflop (E_D_FF_ANY) am Adapter-Plug `OUT` bereit. Damit lassen sich Netzkommunikation und DWORD-Signalverarbeitung in der AD-Adapter-Welt sauber verbinden.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`
- **RSP** (Event): Response-Ereignis, verbunden mit `QI`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Abonnement-Dienst

- **ID** (WSTRING): Identifikator des zu abonnierenden Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
OUT	adapter::types::unidirectional::AD	Plug (Ausgang)	Gepufferter empfangener DWORD-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `SUBSCRIBE_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Der interne `SUBSCRIBE_1`-Block überwacht den konfigurierten Kanal. Sobald der zugehörige `PUBLISH_1`-Block Daten veröffentlicht, erzeugt er das Indikationsereignis `IND` und legt den empfangenen Wert an `RD_1` an.
3. `IND` taktet den internen **E_D_FF_ANY**-Flipflop (`CLK`), der den Wert von `RD_1` an seinem Dateneingang `D` übernimmt.
4. Der gespeicherte Wert erscheint stabil am Ausgang `Q` und wird über die Datenverbindung auf `OUT.D1` des AX-Plugs gelegt.
5. Das Ereignis `EO` des Flipflops wird auf den Ereignis-Ausgang `OUT.E1` gelegt, sodass nachgeschaltete Bausteine über die neue Information benachrichtigt werden.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der empfangene DWORD-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert und bleibt stabil am Ausgang, bis der nächste Wert empfangen wird. `EO` feuert nur bei einer Änderung von `Q`.
- **RSP ist ein No-Op bei Subscribern**: Laut Quellcode-Analyse von `4diac-forte` (`CCommFB::sendData()`) führt das `RSP`-Ereignis bei `SUBSCRIBE_1` (Service-Typ Subscriber) keine Aktion aus – der Zweig `!= e_Subscriber` wird übersprungen. Das Event-Input-Paar `INIT`/`RSP` wird nur für die Quittierung durchgeführt und ist für die Quittierung im Subscriber-Fall funktionslos. Es wird dennoch nach außen geführt, damit ein konfigurierbarer Anschluss besteht.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `SUBSCRIBE_1` (inkl. `IND`/`RD_1`) wird nach innen verlegt; nach außen ist nur noch die AD-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Das Abonnement ist aktiv; der Block wartet auf Daten vom Publisher.
3. **Datenempfang**: Bei `IND` wird der empfangene DWORD-Wert in das interne D-Flipflop übernommen und über `OUT` ausgegeben.

Anwendungsszenarien

- **Netzwerkkopplung in AD-Anwendungen**: Empfang von DWORD-Signalen (z. B. Schaltbefehle oder Statusbits) über Publish/Subscribe und Weiterleitung über AD-Adapter an die Adapter-basierte Logik.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_PUBLISH_1` und einem echten ereignisbasierten `E_D_FF_ANY` lässt sich die Rückkopplung zwischen `SUBSCRIBE_1` und `PUBLISH_1` entkoppeln (sauberer Fix gegenüber dem Interim-Fix mit `AX_D_FF`).
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AD-Adapter setzen.

Vergleich mit ähnlichen Bausteinen

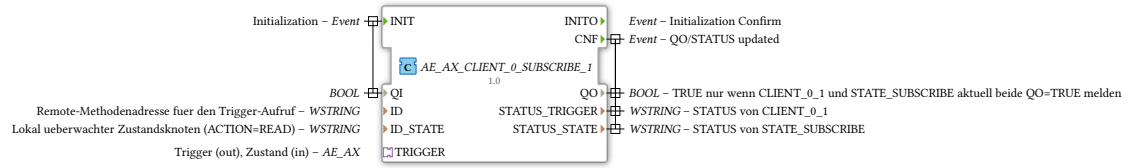
- **SUBSCRIBE_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`IND`, `RD_1`, …). **AD_SUBSCRIBE_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den AD-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.

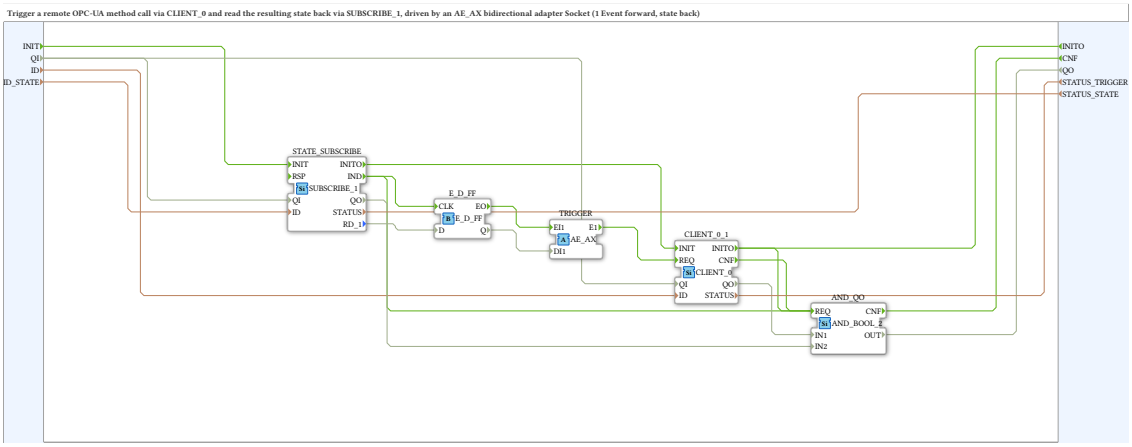
- **AX_D_FF**: Puffert einen DWORD-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AD_SUBSCRIBE_1** vereint Subscribing und Pufferung in einem Baustein.

Fazit

AD_SUBSCRIBE_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `SUBSCRIBE_1` mit der Adapter-basierten DWORD-Verarbeitung. Die interne D-Flipflop-Pufferung hält den empfangenen Wert stabil, bis neue Daten eintreffen, und die reduzierte AD-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

AE_AX_CLIENT_0_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AE_AX_CLIENT_0_SUBSCRIBE_1** ist ein Composite-Funktionsblock, der einen entfernten OPC-UA-**Methodenaufruf** über `CLIENT_0` auslöst und den dazugehörigen Zustand über `SUBSCRIBE_1` zurückliest – beides hinter einem einzigen bidirektionalen **AE_AX-Adapter-Socket**. Er kollabiert damit das Muster „ein händisch verdrahteter `CLIENT_0` plus ein `SUBSCRIBE_1`“ zu einem einzigen Baustein mit einem Adapterport.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO`, `STATUS_TRIGGER` und `STATUS_STATE`
- **CNF** (Event): `QO`/`STATUS` aktualisiert, verbunden mit `QO`, `STATUS_TRIGGER` und `STATUS_STATE`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang für beide interne Verbindungen
- **ID** (WSTRING): Remote-Methodenadresse für den Trigger-Aufruf (`CLIENT_0_1`)
- **ID_STATE** (WSTRING): Lokal überwachter Zustandsknoten für `SUBSCRIBE_1` (ACTION=READ)

Daten-Ausgänge

- **QO** (BOOL): TRUE nur wenn `CLIENT_0_1` und `STATE_SUBSCRIBE` aktuell beide `QO = TRUE` melden
- **STATUS_TRIGGER** (WSTRING): Statusinformationen von `CLIENT_0_1`
- **STATUS_STATE** (WSTRING): Statusinformationen von `STATE_SUBSCRIBE`

Adapter

Adapter	Typ	Richtung	Beschreibung
TRIGGER	adapter::types::bidirectional::AE_AX	Socket – Trigger (Ausgang), Zustand (Eingang)	Auslösendes Ereignis nach außen, Zustand zurück

Funktionsweise

1. Über das `INIT`-Ereignis wird zuerst `STATE_SUBSCRIBE` (`SUBSCRIBE_1`) mit `QI` und `ID_STATE` initialisiert; nach dessen `INITO` wird `CLIENT_0_1` mit `QI` und `ID` initialisiert. Nach beiden Bestätigungen wird `INITO` nach außen gemeldet.
2. Ein am Adapter-Socket eintreffendes Ereignis `TRIGGER.E1` löst direkt den Methodenaufruf `CLIENT_0_1.REQ` aus.
3. Sobald `STATE_SUBSCRIBE` per `IND` einen neuen Zustandswert an `RD_1` liefert, taktet dieses Ereignis das interne **E_D_FF**-Flipflop, das den Wert übernimmt und stabil hält.
4. Der Ausgang `EO` des Flipflops feuert `TRIGGER.EI1` mit dem gepufferten Wert an `TRIGGER.DI1` – der Zustand wird also über denselben Adapter zurückgemeldet, über den auch der Trigger nach außen geht.
5. `AND_QO` (AND_BOOL_2) verundet die `QO`-Ausgänge von `CLIENT_0_1` und `STATE_SUBSCRIBE`; jede Bestätigung (`CLIENT_0_1.CNF` oder `STATE_SUBSCRIBE.IND`) löst zusätzlich `CNF` nach außen aus.

Technische Besonderheiten

- **Zwei Netzverbindungen hinter einem Adapter**: `CLIENT_0` (Trigger, Schreibrichtung) und `SUBSCRIBE_1` (Zustand, Leserichtung) werden zu einer einzigen bidirektionalen AE_AX-Schnittstelle zusammengefasst.
- **Pufferung mit D-Flipflop**: Der über `SUBSCRIBE_1` empfangene Zustand wird über ein internes `iec61499::events::E_D_FF` stabilisiert, bevor er über den Adapter zurückgemeldet wird.
- **Sequentielle Initialisierung**: `STATE_SUBSCRIBE` wird vor `CLIENT_0_1` initialisiert (Composite-FBTypes verketten `INIT`/`INITO` seriell durch jede Instanz).
- **Kapselung**: Nach außen ist nur noch die AE_AX-Adapter-Schnittstelle sichtbar; `CLIENT_0`s `REQ` und `SUBSCRIBE_1`s `IND`/`RD_1` bleiben intern.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Beide Verbindungen (Trigger und Zustandsabo) sind aufgebaut.
3. **Sendeaktiv**: Ein am Adapter eintreffendes Trigger-Ereignis löst den Remote-Methodenaufruf aus.
4. **Zustandsaktualisierung**: Ein neuer Wert vom Abonnement wird gepuffert und über denselben Adapter zurückgemeldet.

Anwendungsszenarien

- Fernauslösen einer entfernten Aktion und gleichzeitiges Rücklesen des resultierenden Zustands über eine einzige Adapterverbindung, z. B. für ein Bedienelement, das sofort den tatsächlichen Zustand des entfernten Aktors anzeigt
- Reduktion von händischer CLIENT_0/SUBSCRIBE_1-Verdrahtung auf einen einzigen wiederverwendbaren Baustein

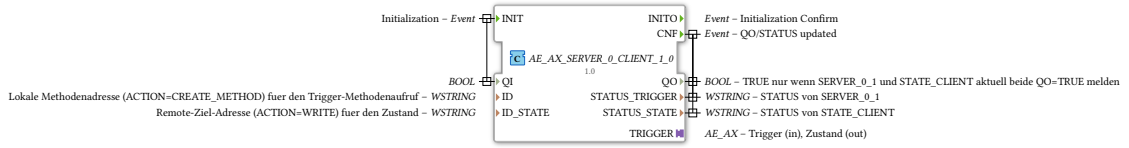
Vergleich mit ähnlichen Bausteinen

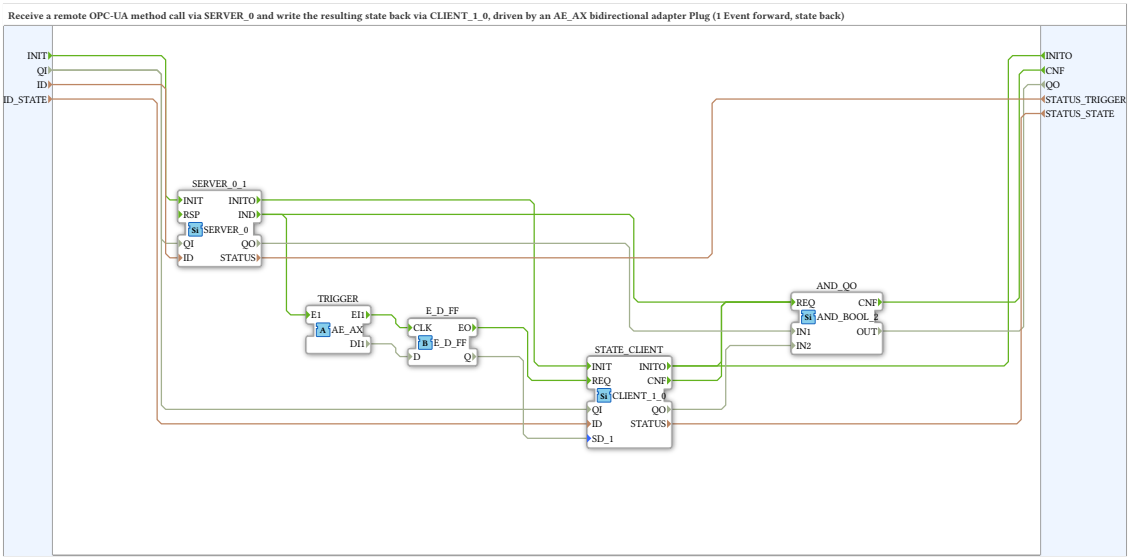
- **AE_CLIENT_0**: Reiner Trigger ohne Zustands-Rücklesung.
- **AI_SUBSCRIBE_1**: Reines Zustands-Abonnement ohne Trigger-Funktion.
- **ASR_AX_CLIENT_0_SUBSCRIBE_1** / **ASRT_AX_CLIENT_0_SUBSCRIBE_1**: Erweitern dasselbe Muster auf 2 bzw. 3 parallele Trigger.
- **AE_AX_SERVER_0_CLIENT_1_0**: Die serverseitige Gegenstelle – empfängt statt zu triggern und schreibt statt zu abonnieren.

Fazit

AE_AX_CLIENT_0_SUBSCRIBE_1 bündelt einen Remote-Trigger und ein Zustands-Abonnement hinter einem einzigen bidirektionalen AE_AX-Adapter und macht damit händisch verdrahtete CLIENT_0/SUBSCRIBE_1-Kombinationen zu einem einzigen wiederverwendbaren Baustein.

AE_AX_SERVER_0_CLIENT_1_0





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AE_AX_SERVER_0_CLIENT_1_0** ist ein Composite-Funktionsblock, der einen entfernten OPC-UA-**Methodenaufruf** über `SERVER_0` empfängt und den resultierenden Zustand über `CLIENT_1_0` an einen entfernten Knoten zurückschreibt – beides hinter einem einzigen bidirektionalen **AE_AX-Adapter-Plug**. Er ist die serverseitige Gegenstelle zu AE_AX_CLIENT_0_SUBSCRIBE_1.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO`, `STATUS_TRIGGER` und `STATUS_STATE`
- **CNF** (Event): `QO`/`STATUS` aktualisiert, verbunden mit `QO`, `STATUS_TRIGGER` und `STATUS_STATE`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang für beide interne Verbindungen
- **ID** (WSTRING): Lokale Methodenadresse (ACTION=CREATE_METHOD) für den Trigger-Methodenaufruf (`SERVER_0_1`)
- **ID_STATE** (WSTRING): Remote-Ziel-Adresse (ACTION=WRITE) für den Zustand (`STATE_CLIENT`)

Daten-Ausgänge

- **QO** (BOOL): TRUE nur wenn `SERVER_0_1` und `STATE_CLIENT` aktuell beide `QO = TRUE` melden
- **STATUS_TRIGGER** (WSTRING): Statusinformationen von `SERVER_0_1`
- **STATUS_STATE** (WSTRING): Statusinformationen von `STATE_CLIENT`

Adapter

Adapter	Typ	Richtung	Beschreibung
TRIGGER	adapter::types::bidirectional::AE_AX	Plug – Trigger (Eingang), Zustand (Ausgang)	Empfangenes Ereignis, Zustand nach außen

Funktionsweise

1. Über das `INIT`-Ereignis wird zuerst `SERVER_0_1` mit `QI` und `ID` initialisiert; nach dessen `INITO` wird `STATE_CLIENT` mit `QI` und `ID_STATE` initialisiert. Nach beiden Bestätigungen wird `INITO` nach außen gemeldet.
2. Ein entfernter Methodenaufruf auf `SERVER_0_1` erzeugt das Indikationsereignis `IND`, das direkt auf `TRIGGER.E1` gelegt wird – der Adapter meldet den Aufruf also nach innen an die weiterverarbeitende Logik.
3. Sobald die weiterverarbeitende Logik über den Adapter ein Rückmeldeereignis `TRIGGER.EI1` mit Datum `TRIGGER.DI1` liefert, taktet dies das interne **E_D_FF**-Flipflop, das den Wert übernimmt und stabil hält.
4. Der Ausgang `EO` des Flipflops löst `STATE_CLIENT.REQ` aus; der gepufferte Wert wird als `SD_1` übergeben und per OPC-UA-Write an den unter `ID_STATE` konfigurierten entfernten Knoten gesendet.
5. `AND_QO` (AND_BOOL_2) verundet die `QO`-Ausgänge von `SERVER_0_1` und `STATE_CLIENT`; jede Bestätigung löst zusätzlich `CNF` nach außen aus.

Technische Besonderheiten

- **Zwei Netzverbindungen hinter einem Adapter**: `SERVER_0` (Empfang, Server-Rolle) und `CLIENT_1_0` (Zustand, Remote-Write) werden zu einer einzigen bidirektionalen AE_AX-Schnittstelle zusammengefasst.
- **Pufferung mit D-Flipflop**: Der über den Adapter gemeldete Rückmeldewert wird über ein internes `iec61499::events::E_D_FF` stabilisiert, bevor er per `CLIENT_1_0` verschickt wird.
- **Sequentielle Initialisierung**: `SERVER_0_1` wird vor `STATE_CLIENT` initialisiert (Composite-FBTypes verketten `INIT`/`INITO` seriell durch jede Instanz).
- **Kapselung**: Nach außen ist nur noch die AE_AX-Adapter-Schnittstelle sichtbar; `SERVER_0`s `IND` und `CLIENT_1_0`s `REQ`/`SD_1` bleiben intern.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Beide Verbindungen (Server und Zustands-Client) sind aufgebaut.
3. **Empfangsaktiv**: Ein entfernter Methodenaufruf wird über den Adapter an die interne Logik gemeldet.
4. **Zustandsschreibvorgang**: Ein Rückmeldewert der internen Logik wird gepuffert und per Remote-Write an den konfigurierten Zielknoten gesendet.

Anwendungsszenarien

- Entgegennahme eines entfernten Trigger-Aufrufs und gleichzeitiges Zurückschreiben des resultierenden Zustands an einen anderen entfernten Knoten über eine einzige Adapterverbindung
- Serverseitiges Gegenstück zu AE_AX_CLIENT_0_SUBSCRIBE_1 in verteilten Steuerungsarchitekturen

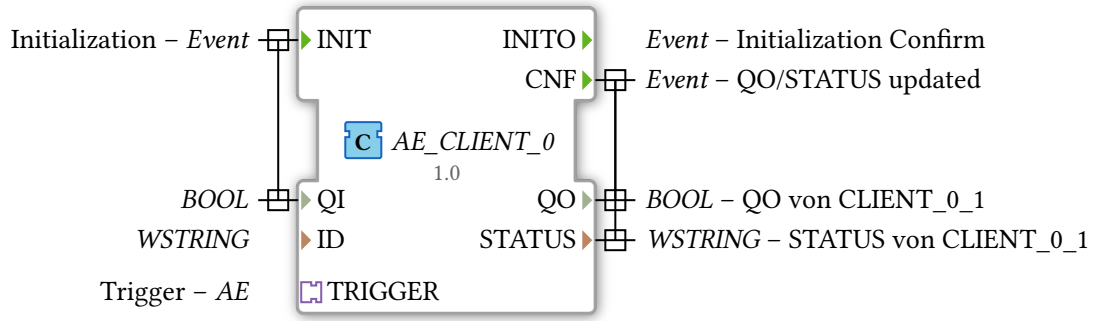
Vergleich mit ähnlichen Bausteinen

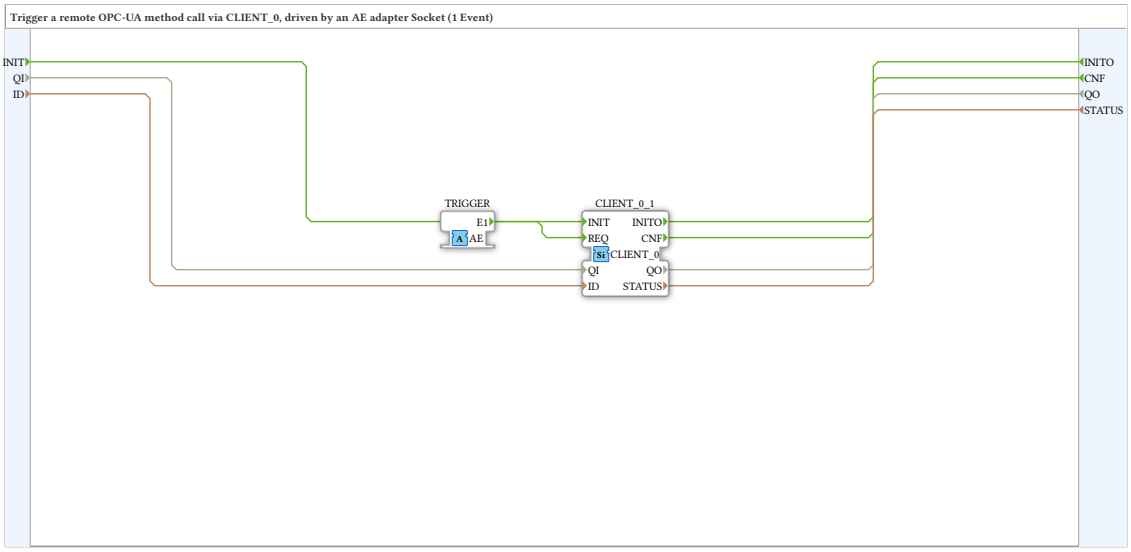
- **AE_AX_CLIENT_0_SUBSCRIBE_1**: Die client-/aufrufende Gegenstelle – triggert aktiv statt zu empfangen, abonniert statt zu schreiben.
- **AX_CLIENT_1_0**: Reiner Remote-Write ohne Empfangsseite.
- **ASR_AX_SERVER_0_CLIENT_1_0** / **ASRT_AX_SERVER_0_CLIENT_1_0**: Erweitern dasselbe Muster auf 2 bzw. 3 parallele Empfangs-Server.

Fazit

AE_AX_SERVER_0_CLIENT_1_0 bündelt einen empfangenden Server und einen zustandsschreibenden Client hinter einem einzigen bidirektionalen AE_AX-Adapter und bildet damit die serverseitige Entsprechung zu AE_AX_CLIENT_0_SUBSCRIBE_1.

AE_CLIENT_0





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AE_CLIENT_0** ist ein Composite-Funktionsblock, der den netzwerkbasierten `CLIENT_0`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AE-Adapter** abbildet. Jedes am Adapter-Socket `TRIGGER` eintreffende Ereignis löst über `CLIENT_0` einen entfernten OPC-UA-**Methodenaufruf** unter der unter `ID` konfigurierten Adresse aus.

Im Unterschied zu `CLIENT_1_0` (Remote-Write eines Datenwerts) ruft `CLIENT_0` eine entfernte **Methode ohne Nutzdaten** auf – der passende Baustein für ereignisgesteuerte Trigger-Aktionen wie „Set“, „Reset“ oder „Toggle“ auf einem entfernten Knoten.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`
- **CNF** (Event): Bestätigung, dass der Methodenaufruf gesendet wurde, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, öffnet (TRUE) bzw. schließt (FALSE) die Verbindung zum Server
- **ID** (WSTRING): Remote-Methodenadresse für den Aufruf

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Verbindungsstatus (identisch mit `CLIENT_0_1.QO`)
- **STATUS** (WSTRING): Statusinformationen als Unicode-String (identisch mit `CLIENT_0_1.STATUS`)

Adapter

Adapter	Typ	Richtung	Beschreibung
TRIGGER	adapter::types::unidirectional::AE	Socket (Eingang)	Auslösendes Ereignis für den Methodenaufruf

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `CLIENT_0_1`-Block mit `QI` und `ID` initialisiert; er baut die Verbindung zum entfernten Server auf. Bei Erfolg quittiert er mit `INITO`.
2. Jedes am Adapter-Socket `TRIGGER` eintreffende Ereignis (`TRIGGER.E1`) löst direkt das Sendeereignis `REQ` des internen `CLIENT_0_1`-Blocks aus.
3. `CLIENT_0_1` ruft daraufhin die unter `ID` konfigurierte Remote-Methode auf und bestätigt mit `CNF`, das nach außen geführt wird.

Technische Besonderheiten

- **Reiner Trigger, keine Nutzdaten**: Anders als `AX_CLIENT_1_0` überträgt `AE_CLIENT_0` keinen Datenwert – der Adapter liefert lediglich das auslösende Ereignis.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `CLIENT_0` (inkl. `REQ`) wird nach innen verlegt; nach außen ist nur noch die AE-Adapter-Schnittstelle sichtbar.
- Baustein der Grundfamilie, aus der ASR_CLIENT_0 (2 Trigger) und ASRT_CLIENT_0 (3 Trigger) durch Parallelschaltung mehrerer `CLIENT_0`-Instanzen hervorgehen.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Die Verbindung zum entfernten Server ist aufgebaut, der Block ist bereit, Methodenaufrufe auszulösen.
3. **Sendeaktiv**: Ein am AE-Socket eintreffendes Ereignis löst den Remote-Methodenaufruf über den internen `CLIENT_0`-Block aus.

Anwendungsszenarien

- Fernauslösen einer einzelnen entfernten Aktion (z. B. „Start“, „Trigger“) ohne zugehörigen Datenwert
- Baustein für zusammengesetzte Mehrfach-Trigger-Bausteine wie ASR_CLIENT_0 und ASRT_CLIENT_0
- Netzwerkkopplung in AE-Adapter-basierten Anwendungen

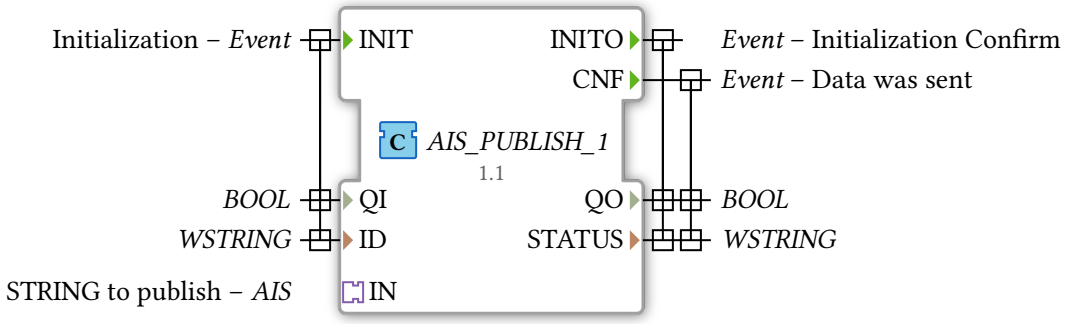
Vergleich mit ähnlichen Bausteinen

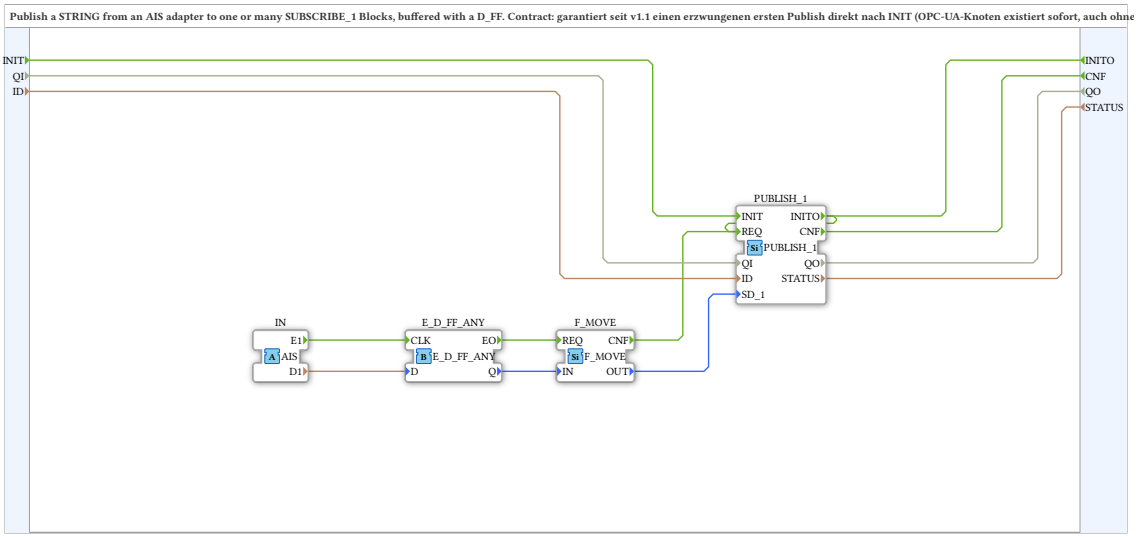
- **CLIENT_0**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`REQ`, …). **AE_CLIENT_0** kapselt diesen Baustein und stellt nur noch den AE-Adapter bereit.
- **AX_CLIENT_1_0**: Strukturell ähnlich aufgebaut, kapselt aber `CLIENT_1_0` (Remote-Write eines Datenwerts) statt `CLIENT_0` (reiner Methodenaufruf ohne Daten).
- **ASR_CLIENT_0** / **ASRT_CLIENT_0**: Erweitern dasselbe Muster auf 2 bzw. 3 parallele Trigger (Set/Reset bzw. Set/Reset/Toggle).

Fazit

AE_CLIENT_0 verbindet die Trigger-Semantik des Standardbausteins `CLIENT_0` mit der Adapter-basierten Ereignisverarbeitung. Die reduzierte AE-Schnittstelle macht den Baustein ideal, um in durchgängig adapterbasierten IEC 61499-Anwendungen einen entfernten OPC-UA-Methodenaufruf direkt auszulösen.

AIS_PUBLISH_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AIS_PUBLISH_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `PUBLISH_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AIS-Adapter** abbildet. Ein am Adapter-Socket `IN` anliegender **STRING**-Wert wird über ein internes D-Flipflop (E_D_FF_ANY) gepuffert und anschließend über den konfigurierten Publish-Kanal an alle zugehörigen `SUBSCRIBE_1`-Empfänger gesendet.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`
- **CNF** (Event): Bestätigung, dass die Daten gesendet wurden, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Publish-Dienst
- **ID** (WSTRING): Identifikator des Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::AIS	Socket (Eingang)	Zu veröffentlichender STRING-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `PUBLISH_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Sobald der AIS-Socket `IN` ein Ereignis auf `IN.E1` liefert, wird der gleichzeitig an `IN.D1` anliegende STRING-Wert in das interne **E_D_FF_ANY**-Flipflop übernommen (`CLK` bzw. `D`).
3. Das Flipflop hält den Wert stabil an seinem Ausgang `Q` und erzeugt bei einer Wertänderung das Ereignis `EO`.
4. `EO` löst das Sendeereignis `REQ` des internen `PUBLISH_1`-Blocks aus; der gepufferte Wert `Q` wird über die Datenverbindung als `SD_1` übergeben und auf dem konfigurierten Kanal veröffentlicht.
5. Nach erfolgreichem Senden bestätigt der `PUBLISH_1`-Block mit `CNF`, das nach außen geführt wird.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der zu sendende STRING-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert. Dadurch wird verhindert, dass ein sich während der Verarbeitung ändernder Eingangswert den Sendevorgang stört.
- **Ereignisbasierte Entkopplung**: Durch das echte ereignisbasierte `E_D_FF_ANY` werden unmittelbare Rückkopplungsschleifen zwischen `PUBLISH_1` und `SUBSCRIBE_1` (z. B. auf demselben OPC-UA-Knoten) entkoppelt – der saubere Fix gegenüber dem Interim-Fix mit `AX_D_FF`.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `PUBLISH_1` (inkl. `REQ`/`SD_1`) wird nach innen verlegt; nach außen ist nur noch die AIS-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Der Block ist bereit, Daten über den konfigurierten Kanal zu veröffentlichen.
3. **Sendeaktiv**: Ein am AIS-Socket eintreffendes Ereignis puffert den Wert und veranlasst das Senden über den internen `PUBLISH_1`-Block.

Anwendungsszenarien

- **Netzwerkkopplung in AIS-Anwendungen**: Senden von STRING-Signalen (z. B. Schaltbefehle oder Statusbits) aus der Adapter-basierten Logik über Publish/Subscribe an entfernte Empfänger.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_SUBSCRIBE_1` werden Rückkopplungen zwischen eigenen Veröffentlichungen und dem eigenen Abonnement entkoppelt.
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AIS-Adapter setzen.

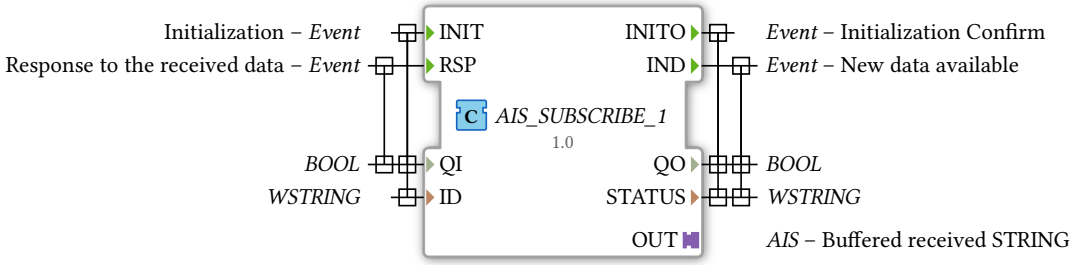
Vergleich mit ähnlichen Bausteinen

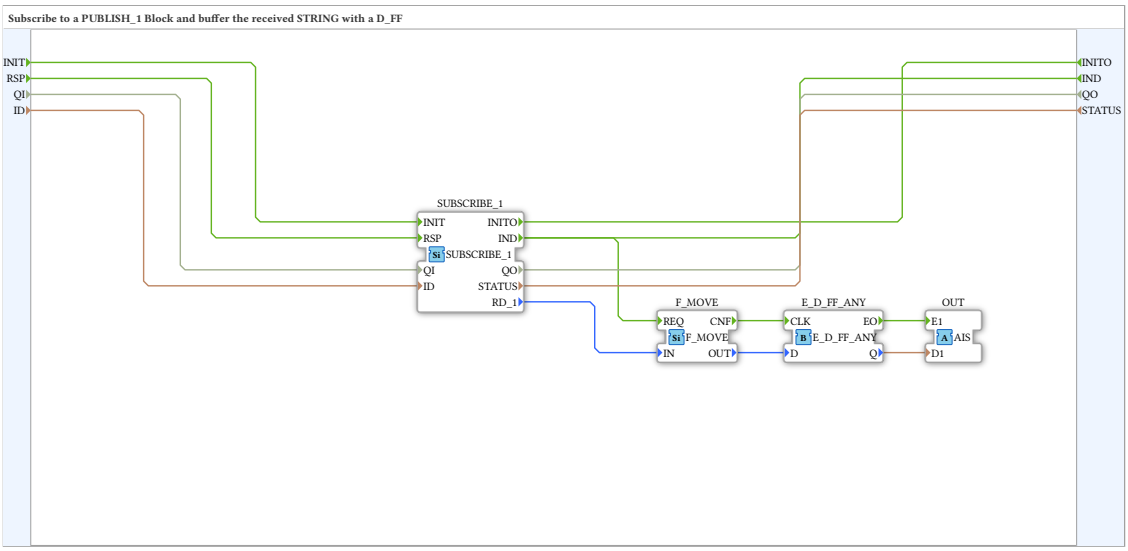
- **PUBLISH_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`REQ`, `SD_1`, …). **AIS_PUBLISH_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den AIS-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen STRING-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AIS_PUBLISH_1** vereint Puffern und Veröffentlichen in einem Baustein.

Fazit

AIS_PUBLISH_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `PUBLISH_1` mit der Adapter-basierten STRING-Verarbeitung. Die interne D-Flipflop-Pufferung stabilisiert den Sendewert und entkoppelt Rückkopplungsschleifen; die reduzierte AIS-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

AIS_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AIS_SUBSCRIBE_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `SUBSCRIBE_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AIS-Adapter** abbildet. Er abonniert einen über die `ID` konfigurierten Publish-Kanal und stellt den empfangenen **STRING**-Wert gepuffert über einen D-Flipflop (E_D_FF_ANY) am Adapter-Plug `OUT` bereit. Damit lassen sich Netzkommunikation und STRING-Signalverarbeitung in der AIS-Adapter-Welt sauber verbinden.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`
- **RSP** (Event): Response-Ereignis, verbunden mit `QI`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Abonnement-Dienst
- **ID** (WSTRING): Identifikator des zu abonnierenden Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung

OUT	adapter::types::unidirectional::AIS	Plug (Ausgang)	Gepufferter empfangener STRING-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `SUBSCRIBE_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Der interne `SUBSCRIBE_1`-Block überwacht den konfigurierten Kanal. Sobald der zugehörige `PUBLISH_1`-Block Daten veröffentlicht, erzeugt er das Indikationsereignis `IND` und legt den empfangenen Wert an `RD_1` an.
3. `IND` taktet den internen **E_D_FF_ANY**-Flipflop (`CLK`), der den Wert von `RD_1` an seinem Dateneingang `D` übernimmt.
4. Der gespeicherte Wert erscheint stabil am Ausgang `Q` und wird über die Datenverbindung auf `OUT.D1` des AX-Plugs gelegt.
5. Das Ereignis `EO` des Flipflops wird auf den Ereignis-Ausgang `OUT.E1` gelegt, sodass nachgeschaltete Bausteine über die neue Information benachrichtigt werden.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der empfangene STRING-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert und bleibt stabil am Ausgang, bis der nächste Wert empfangen wird. `EO` feuert nur bei einer Änderung von `Q`.
- **RSP ist ein No-Op bei Subscribern**: Laut Quellcode-Analyse von `4diac-forte` (`CCommFB::sendData()`) führt das `RSP`-Ereignis bei `SUBSCRIBE_1` (Service-Typ Subscriber) keine Aktion aus – der Zweig `!= e_Subscriber` wird übersprungen. Das Event-Input-Paar `INIT`/`RSP` wird nur für die Quittierung durchgeführt und ist für die Quittierung im Subscriber-Fall funktionslos. Es wird dennoch nach außen geführt, damit ein konfigurierbarer Anschluss besteht.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `SUBSCRIBE_1` (inkl. `IND`/`RD_1`) wird nach innen verlegt; nach außen ist nur noch die AIS-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Das Abonnement ist aktiv; der Block wartet auf Daten vom Publisher.
3. **Datenempfang**: Bei `IND` wird der empfangene STRING-Wert in das interne D-Flipflop übernommen und über `OUT` ausgegeben.

Anwendungsszenarien

- **Netzwerkkopplung in AIS-Anwendungen**: Empfang von STRING-Signalen (z. B. Schaltbefehle oder Statusbits) über Publish/Subscribe und Weiterleitung über AIS-Adapter an die Adapter-basierte Logik.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_PUBLISH_1` und einem echten ereignisbasierten `E_D_FF_ANY` lässt sich die Rückkopplung zwischen `SUBSCRIBE_1` und `PUBLISH_1` entkoppeln (sauberer Fix gegenüber dem Interim-Fix mit `AX_D_FF`).
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AIS-Adapter setzen.

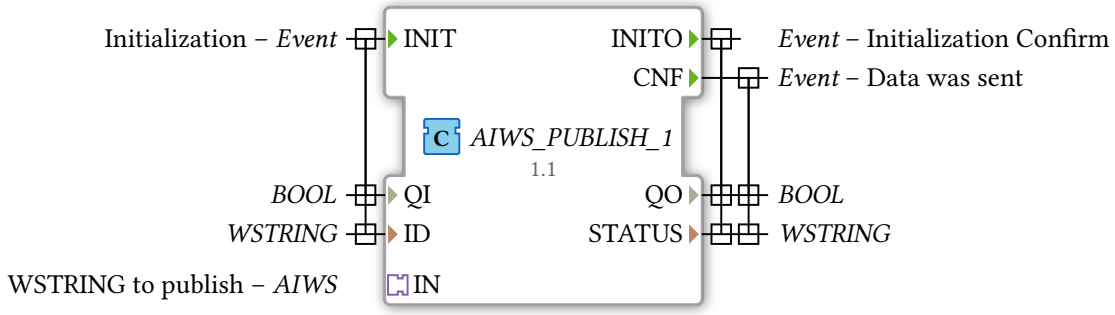
Vergleich mit ähnlichen Bausteinen

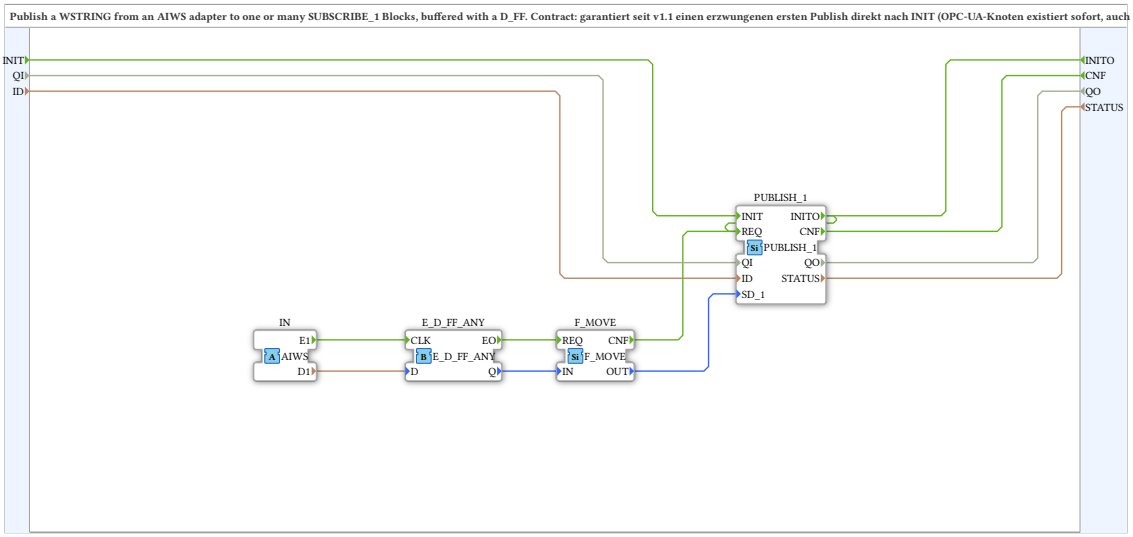
- **SUBSCRIBE_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`IND`, `RD_1`, …). **AIS_SUBSCRIBE_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den AIS-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen STRING-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AIS_SUBSCRIBE_1** vereint Subscribing und Pufferung in einem Baustein.

Fazit

AIS_SUBSCRIBE_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `SUBSCRIBE_1` mit der Adapter-basierten STRING-Verarbeitung. Die interne D-Flipflop-Pufferung hält den empfangenen Wert stabil, bis neue Daten eintreffen, und die reduzierte AIS-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

AIWS_PUBLISH_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AIWS_PUBLISH_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `PUBLISH_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AIWS-Adapter** abbildet. Ein am Adapter-Socket `IN` anliegender **WSTRING**-Wert wird über ein internes D-Flipflop (E_D_FF_ANY) gepuffert und anschließend über den konfigurierten Publish-Kanal an alle zugehörigen `SUBSCRIBE_1`-Empfänger gesendet.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`
- **CNF** (Event): Bestätigung, dass die Daten gesendet wurden, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Publish-Dienst
- **ID** (WSTRING): Identifikator des Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::AIWS	Socket (Eingang)	Zu veröffentlichender WSTRING-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `PUBLISH_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Sobald der AIWS-Socket `IN` ein Ereignis auf `IN.E1` liefert, wird der gleichzeitig an `IN.D1` anliegende WSTRING-Wert in das interne **E_D_FF_ANY**-Flipflop übernommen (`CLK` bzw. `D`).
3. Das Flipflop hält den Wert stabil an seinem Ausgang `Q` und erzeugt bei einer Wertänderung das Ereignis `EO`.
4. `EO` löst das Sendeereignis `REQ` des internen `PUBLISH_1`-Blocks aus; der gepufferte Wert `Q` wird über die Datenverbindung als `SD_1` übergeben und auf dem konfigurierten Kanal veröffentlicht.
5. Nach erfolgreichem Senden bestätigt der `PUBLISH_1`-Block mit `CNF`, das nach außen geführt wird.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der zu sendende WSTRING-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert. Dadurch wird verhindert, dass ein sich während der Verarbeitung ändernder Eingangswert den Sendevorgang stört.
- **Ereignisbasierte Entkopplung**: Durch das echte ereignisbasierte `E_D_FF_ANY` werden unmittelbare Rückkopplungsschleifen zwischen `PUBLISH_1` und `SUBSCRIBE_1` (z. B. auf demselben OPC-UA-Knoten) entkoppelt – der saubere Fix gegenüber dem Interim-Fix mit `AX_D_FF`.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `PUBLISH_1` (inkl. `REQ`/`SD_1`) wird nach innen verlegt; nach außen ist nur noch die AIWS-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Der Block ist bereit, Daten über den konfigurierten Kanal zu veröffentlichen.
3. **Sendeaktiv**: Ein am AIWS-Socket eintreffendes Ereignis puffert den Wert und veranlasst das Senden über den internen `PUBLISH_1`-Block.

Anwendungsszenarien

- **Netzwerkkopplung in AIWS-Anwendungen**: Senden von WSTRING-Signalen (z. B. Schaltbefehle oder Statusbits) aus der Adapter-basierten Logik über Publish/Subscribe an entfernte Empfänger.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_SUBSCRIBE_1` werden Rückkopplungen zwischen eigenen Veröffentlichungen und dem eigenen Abonnement entkoppelt.
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AIWS-Adapter setzen.

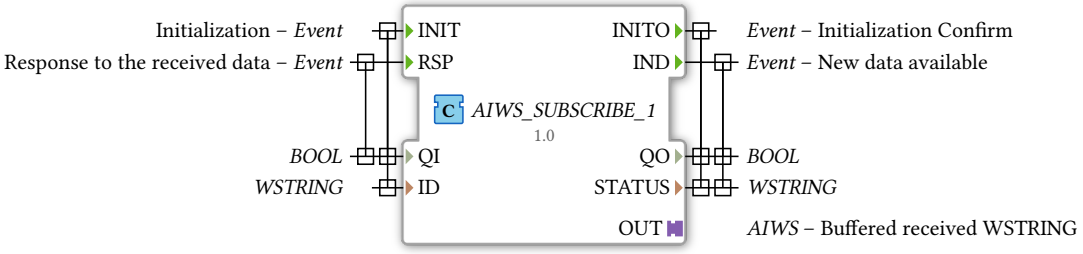
Vergleich mit ähnlichen Bausteinen

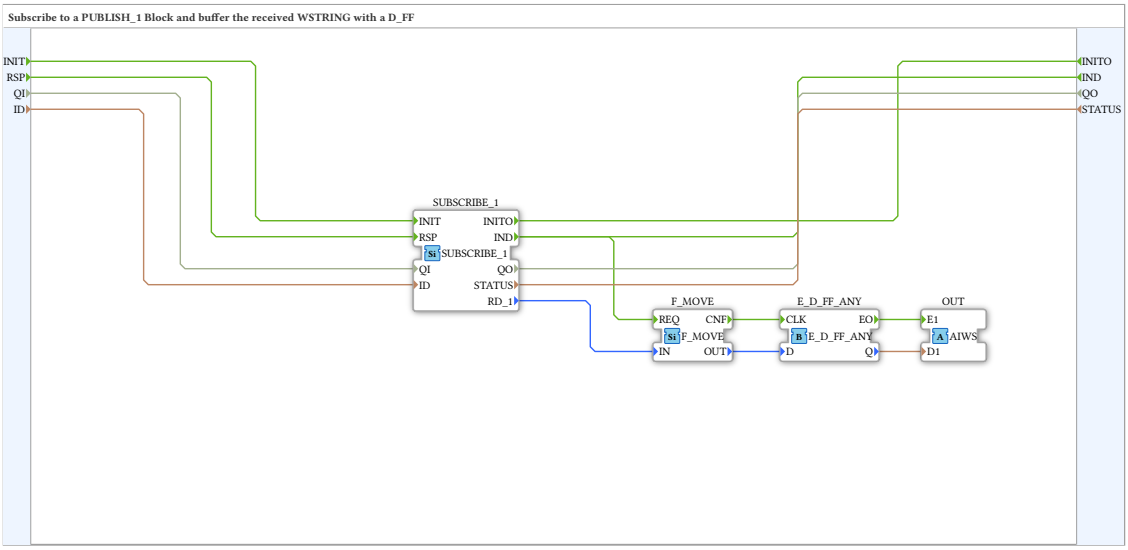
- **PUBLISH_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`REQ`, `SD_1`, …). **AIWS_PUBLISH_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den AIWS-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen WSTRING-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AIWS_PUBLISH_1** vereint Puffern und Veröffentlichen in einem Baustein.

Fazit

AIWS_PUBLISH_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `PUBLISH_1` mit der Adapter-basierten WSTRING-Verarbeitung. Die interne D-Flipflop-Pufferung stabilisiert den Sendewert und entkoppelt Rückkopplungsschleifen; die reduzierte AIWS-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

AIWS_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AIWS_SUBSCRIBE_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `SUBSCRIBE_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AIWS-Adapter** abbildet. Er abonniert einen über die `ID` konfigurierten Publish-Kanal und stellt den empfangenen **WSTRING**-Wert gepuffert über einen D-Flipflop (E_D_FF_ANY) am Adapter-Plug `OUT` bereit. Damit lassen sich Netzkommunikation und WSTRING-Signalverarbeitung in der AIWS-Adapter-Welt sauber verbinden.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`
- **RSP** (Event): Response-Ereignis, verbunden mit `QI`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Abonnement-Dienst
- **ID** (WSTRING): Identifikator des zu abonnierenden Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung

OUT	adapter::types::unidirectional::AIWS	Plug (Ausgang)	Gepufferter empfangener WSTRING-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `SUBSCRIBE_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Der interne `SUBSCRIBE_1`-Block überwacht den konfigurierten Kanal. Sobald der zugehörige `PUBLISH_1`-Block Daten veröffentlicht, erzeugt er das Indikationsereignis `IND` und legt den empfangenen Wert an `RD_1` an.
3. `IND` taktet den internen **E_D_FF_ANY**-Flipflop (`CLK`), der den Wert von `RD_1` an seinem Dateneingang `D` übernimmt.
4. Der gespeicherte Wert erscheint stabil am Ausgang `Q` und wird über die Datenverbindung auf `OUT.D1` des AX-Plugs gelegt.
5. Das Ereignis `EO` des Flipflops wird auf den Ereignis-Ausgang `OUT.E1` gelegt, sodass nachgeschaltete Bausteine über die neue Information benachrichtigt werden.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der empfangene WSTRING-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert und bleibt stabil am Ausgang, bis der nächste Wert empfangen wird. `EO` feuert nur bei einer Änderung von `Q`.
- **RSP ist ein No-Op bei Subscribern**: Laut Quellcode-Analyse von `4diac-forte` (`CCommFB::sendData()`) führt das `RSP`-Ereignis bei `SUBSCRIBE_1` (Service-Typ Subscriber) keine Aktion aus – der Zweig `!= e_Subscriber` wird übersprungen. Das Event-Input-Paar `INIT`/`RSP` wird nur für die Quittierung durchgeführt und ist für die Quittierung im Subscriber-Fall funktionslos. Es wird dennoch nach außen geführt, damit ein konfigurierbarer Anschluss besteht.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `SUBSCRIBE_1` (inkl. `IND`/`RD_1`) wird nach innen verlegt; nach außen ist nur noch die AIWS-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Das Abonnement ist aktiv; der Block wartet auf Daten vom Publisher.
3. **Datenempfang**: Bei `IND` wird der empfangene WSTRING-Wert in das interne D-Flipflop übernommen und über `OUT` ausgegeben.

Anwendungsszenarien

- **Netzwerkkopplung in AIWS-Anwendungen**: Empfang von WSTRING-Signalen (z. B. Schaltbefehle oder Statusbits) über Publish/Subscribe und Weiterleitung über AIWS-Adapter an die Adapter-basierte Logik.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_PUBLISH_1` und einem echten ereignisbasierten `E_D_FF_ANY` lässt sich die Rückkopplung zwischen `SUBSCRIBE_1` und `PUBLISH_1` entkoppeln (sauberer Fix gegenüber dem Interim-Fix mit `AX_D_FF`).
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AIWS-Adapter setzen.

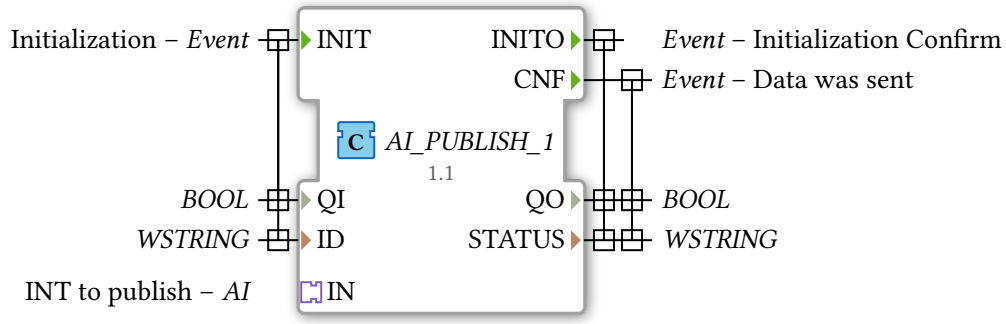
Vergleich mit ähnlichen Bausteinen

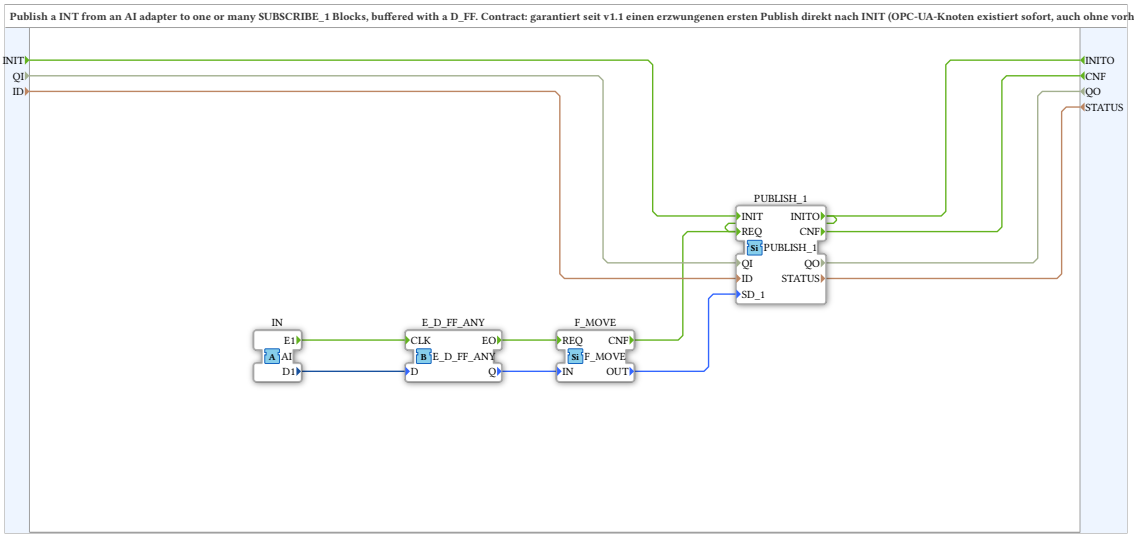
- **SUBSCRIBE_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`IND`, `RD_1`, …). **AIWS_SUBSCRIBE_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den AIWS-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen WSTRING-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AIWS_SUBSCRIBE_1** vereint Subscribing und Pufferung in einem Baustein.

Fazit

AIWS_SUBSCRIBE_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `SUBSCRIBE_1` mit der Adapter-basierten WSTRING-Verarbeitung. Die interne D-Flipflop-Pufferung hält den empfangenen Wert stabil, bis neue Daten eintreffen, und die reduzierte AIWS-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

AI_PUBLISH_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AI_PUBLISH_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `PUBLISH_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AI-Adapter** abbildet. Ein am Adapter-Socket `IN` anliegender **INT**-Wert wird über ein internes D-Flipflop (E_D_FF_ANY) gepuffert und anschließend über den konfigurierten Publish-Kanal an alle zugehörigen `SUBSCRIBE_1`-Empfänger gesendet.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`
- **CNF** (Event): Bestätigung, dass die Daten gesendet wurden, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Publish-Dienst
- **ID** (WSTRING): Identifikator des Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::AI	Socket (Eingang)	Zu veröffentlichender INT-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `PUBLISH_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Sobald der AI-Socket `IN` ein Ereignis auf `IN.E1` liefert, wird der gleichzeitig an `IN.D1` anliegende INT-Wert in das interne **E_D_FF_ANY**-Flipflop übernommen (`CLK` bzw. `D`).
3. Das Flipflop hält den Wert stabil an seinem Ausgang `Q` und erzeugt bei einer Wertänderung das Ereignis `EO`.
4. `EO` löst das Sendeereignis `REQ` des internen `PUBLISH_1`-Blocks aus; der gepufferte Wert `Q` wird über die Datenverbindung als `SD_1` übergeben und auf dem konfigurierten Kanal veröffentlicht.
5. Nach erfolgreichem Senden bestätigt der `PUBLISH_1`-Block mit `CNF`, das nach außen geführt wird.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der zu sendende INT-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert. Dadurch wird verhindert, dass ein sich während der Verarbeitung ändernder Eingangswert den Sendevorgang stört.
- **Ereignisbasierte Entkopplung**: Durch das echte ereignisbasierte `E_D_FF_ANY` werden unmittelbare Rückkopplungsschleifen zwischen `PUBLISH_1` und `SUBSCRIBE_1` (z. B. auf demselben OPC-UA-Knoten) entkoppelt – der saubere Fix gegenüber dem Interim-Fix mit `AX_D_FF`.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `PUBLISH_1` (inkl. `REQ`/`SD_1`) wird nach innen verlegt; nach außen ist nur noch die AI-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Der Block ist bereit, Daten über den konfigurierten Kanal zu veröffentlichen.
3. **Sendeaktiv**: Ein am AI-Socket eintreffendes Ereignis puffert den Wert und veranlasst das Senden über den internen `PUBLISH_1`-Block.

Anwendungsszenarien

- **Netzwerkkopplung in AI-Anwendungen**: Senden von INT-Signalen (z. B. Schaltbefehle oder Statusbits) aus der Adapter-basierten Logik über Publish/Subscribe an entfernte Empfänger.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_SUBSCRIBE_1` werden Rückkopplungen zwischen eigenen Veröffentlichungen und dem eigenen Abonnement entkoppelt.
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AI-Adapter setzen.

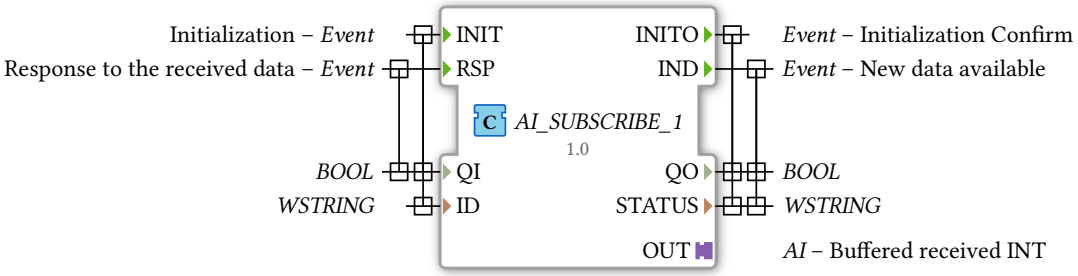
Vergleich mit ähnlichen Bausteinen

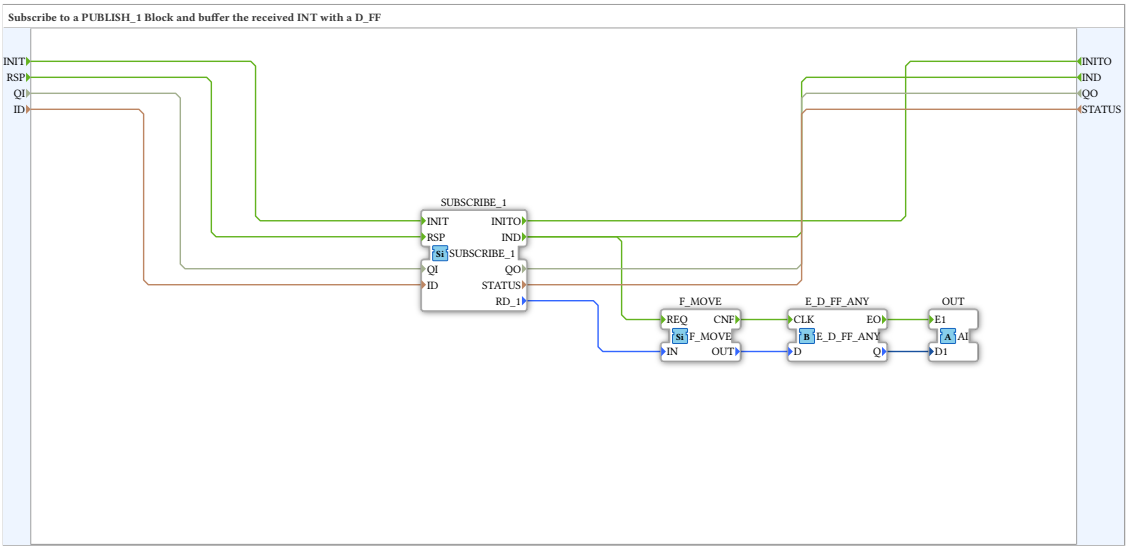
- **PUBLISH_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`REQ`, `SD_1`, …). **AI_PUBLISH_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den AI-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen INT-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AI_PUBLISH_1** vereint Puffern und Veröffentlichen in einem Baustein.

Fazit

AI_PUBLISH_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `PUBLISH_1` mit der Adapter-basierten INT-Verarbeitung. Die interne D-Flipflop-Pufferung stabilisiert den Sendewert und entkoppelt Rückkopplungsschleifen; die reduzierte AI-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

AI_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AI_SUBSCRIBE_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `SUBSCRIBE_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AI-Adapter** abbildet. Er abonniert einen über die `ID` konfigurierten Publish-Kanal und stellt den empfangenen **INT**-Wert gepuffert über einen D-Flipflop (E_D_FF_ANY) am Adapter-Plug `OUT` bereit. Damit lassen sich Netzkommunikation und INT-Signalverarbeitung in der AI-Adapter-Welt sauber verbinden.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`
- **RSP** (Event): Response-Ereignis, verbunden mit `QI`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Abonnement-Dienst
- **ID** (WSTRING): Identifikator des zu abonnierenden Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
OUT	adapter::types::unidirectional::AI	Plug (Ausgang)	Gepufferter empfangener INT-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `SUBSCRIBE_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Der interne `SUBSCRIBE_1`-Block überwacht den konfigurierten Kanal. Sobald der zugehörige `PUBLISH_1`-Block Daten veröffentlicht, erzeugt er das Indikationsereignis `IND` und legt den empfangenen Wert an `RD_1` an.
3. `IND` taktet den internen **E_D_FF_ANY**-Flipflop (`CLK`), der den Wert von `RD_1` an seinem Dateneingang `D` übernimmt.
4. Der gespeicherte Wert erscheint stabil am Ausgang `Q` und wird über die Datenverbindung auf `OUT.D1` des AX-Plugs gelegt.
5. Das Ereignis `EO` des Flipflops wird auf den Ereignis-Ausgang `OUT.E1` gelegt, sodass nachgeschaltete Bausteine über die neue Information benachrichtigt werden.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der empfangene INT-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert und bleibt stabil am Ausgang, bis der nächste Wert empfangen wird. `EO` feuert nur bei einer Änderung von `Q`.
- **RSP ist ein No-Op bei Subscribern**: Laut Quellcode-Analyse von `4diac-forte` (`CCommFB::sendData()`) führt das `RSP`-Ereignis bei `SUBSCRIBE_1` (Service-Typ Subscriber) keine Aktion aus – der Zweig `!= e_Subscriber` wird übersprungen. Das Event-Input-Paar `INIT`/`RSP` wird nur für die Quittierung durchgeführt und ist für die Quittierung im Subscriber-Fall funktionslos. Es wird dennoch nach außen geführt, damit ein konfigurierbarer Anschluss besteht.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `SUBSCRIBE_1` (inkl. `IND`/`RD_1`) wird nach innen verlegt; nach außen ist nur noch die AI-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Das Abonnement ist aktiv; der Block wartet auf Daten vom Publisher.
3. **Datenempfang**: Bei `IND` wird der empfangene INT-Wert in das interne D-Flipflop übernommen und über `OUT` ausgegeben.

Anwendungsszenarien

- **Netzwerkkopplung in AI-Anwendungen**: Empfang von INT-Signalen (z. B. Schaltbefehle oder Statusbits) über Publish/Subscribe und Weiterleitung über AI-Adapter an die Adapter-basierte Logik.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_PUBLISH_1` und einem echten ereignisbasierten `E_D_FF_ANY` lässt sich die Rückkopplung zwischen `SUBSCRIBE_1` und `PUBLISH_1` entkoppeln (sauberer Fix gegenüber dem Interim-Fix mit `AX_D_FF`).
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AI-Adapter setzen.

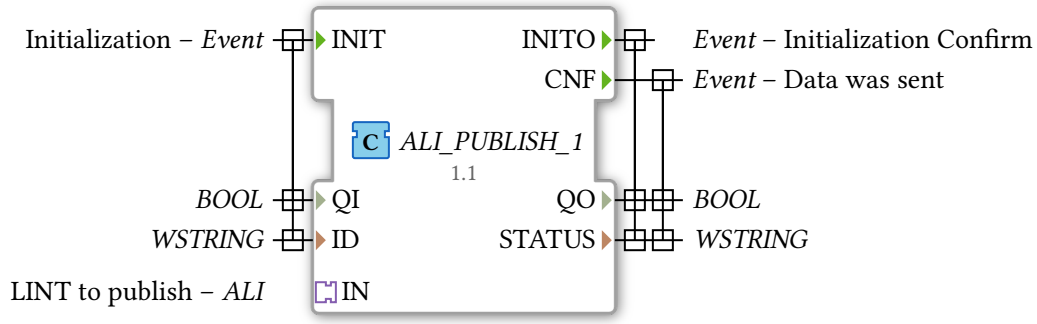
Vergleich mit ähnlichen Bausteinen

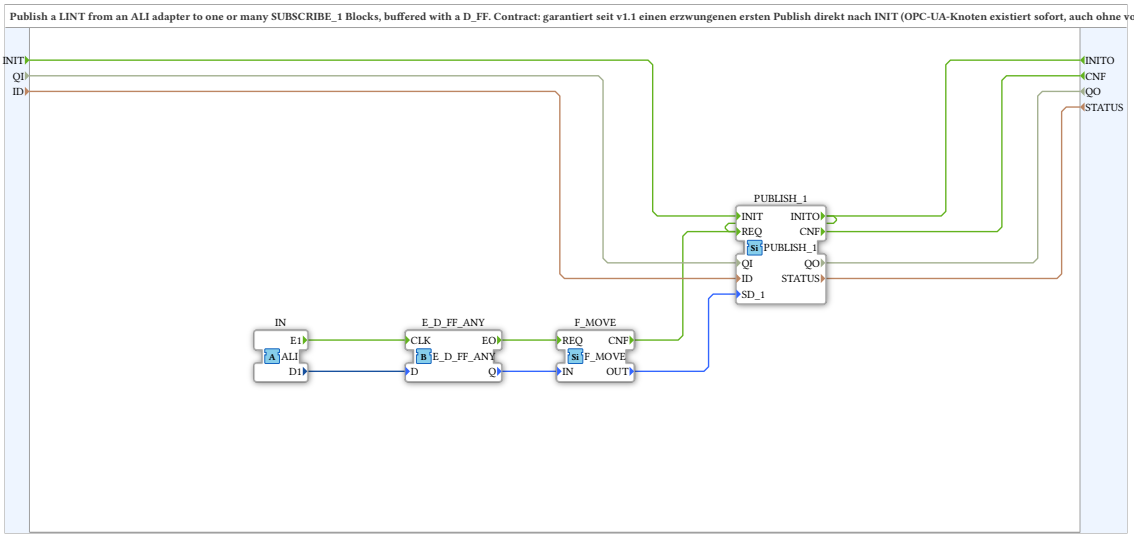
- **SUBSCRIBE_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`IND`, `RD_1`, …). **AI_SUBSCRIBE_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den AI-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen INT-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AI_SUBSCRIBE_1** vereint Subscribing und Pufferung in einem Baustein.

Fazit

AI_SUBSCRIBE_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `SUBSCRIBE_1` mit der Adapter-basierten INT-Verarbeitung. Die interne D-Flipflop-Pufferung hält den empfangenen Wert stabil, bis neue Daten eintreffen, und die reduzierte AI-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

ALI_PUBLISH_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ALI_PUBLISH_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `PUBLISH_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **ALI-Adapter** abbildet. Ein am Adapter-Socket `IN` anliegender **LINT**-Wert wird über ein internes D-Flipflop (E_D_FF_ANY) gepuffert und anschließend über den konfigurierten Publish-Kanal an alle zugehörigen `SUBSCRIBE_1`-Empfänger gesendet.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`
- **CNF** (Event): Bestätigung, dass die Daten gesendet wurden, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Publish-Dienst
- **ID** (WSTRING): Identifikator des Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::ALI	Socket (Eingang)	Zu veröffentlichender LINT-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `PUBLISH_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Sobald der ALI-Socket `IN` ein Ereignis auf `IN.E1` liefert, wird der gleichzeitig an `IN.D1` anliegende LINT-Wert in das interne **E_D_FF_ANY**-Flipflop übernommen (`CLK` bzw. `D`).
3. Das Flipflop hält den Wert stabil an seinem Ausgang `Q` und erzeugt bei einer Wertänderung das Ereignis `EO`.
4. `EO` löst das Sendeereignis `REQ` des internen `PUBLISH_1`-Blocks aus; der gepufferte Wert `Q` wird über die Datenverbindung als `SD_1` übergeben und auf dem konfigurierten Kanal veröffentlicht.
5. Nach erfolgreichem Senden bestätigt der `PUBLISH_1`-Block mit `CNF`, das nach außen geführt wird.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der zu sendende LINT-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert. Dadurch wird verhindert, dass ein sich während der Verarbeitung ändernder Eingangswert den Sendevorgang stört.
- **Ereignisbasierte Entkopplung**: Durch das echte ereignisbasierte `E_D_FF_ANY` werden unmittelbare Rückkopplungsschleifen zwischen `PUBLISH_1` und `SUBSCRIBE_1` (z. B. auf demselben OPC-UA-Knoten) entkoppelt – der saubere Fix gegenüber dem Interim-Fix mit `AX_D_FF`.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `PUBLISH_1` (inkl. `REQ`/`SD_1`) wird nach innen verlegt; nach außen ist nur noch die ALI-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Der Block ist bereit, Daten über den konfigurierten Kanal zu veröffentlichen.
3. **Sendeaktiv**: Ein am ALI-Socket eintreffendes Ereignis puffert den Wert und veranlasst das Senden über den internen `PUBLISH_1`-Block.

Anwendungsszenarien

- **Netzwerkkopplung in ALI-Anwendungen**: Senden von LINT-Signalen (z. B. Schaltbefehle oder Statusbits) aus der Adapter-basierten Logik über Publish/Subscribe an entfernte Empfänger.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_SUBSCRIBE_1` werden Rückkopplungen zwischen eigenen Veröffentlichungen und dem eigenen Abonnement entkoppelt.
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf ALI-Adapter setzen.

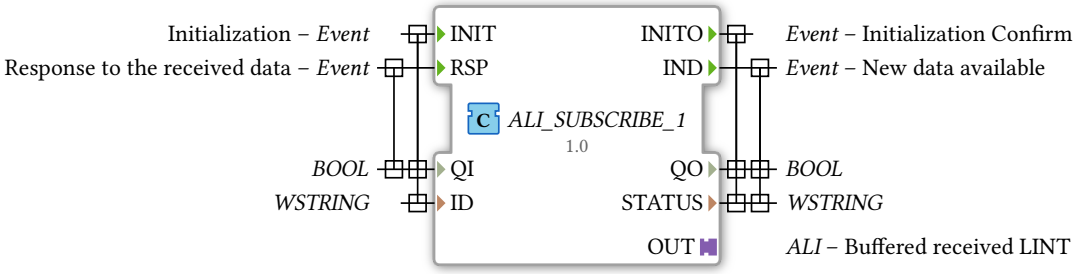
Vergleich mit ähnlichen Bausteinen

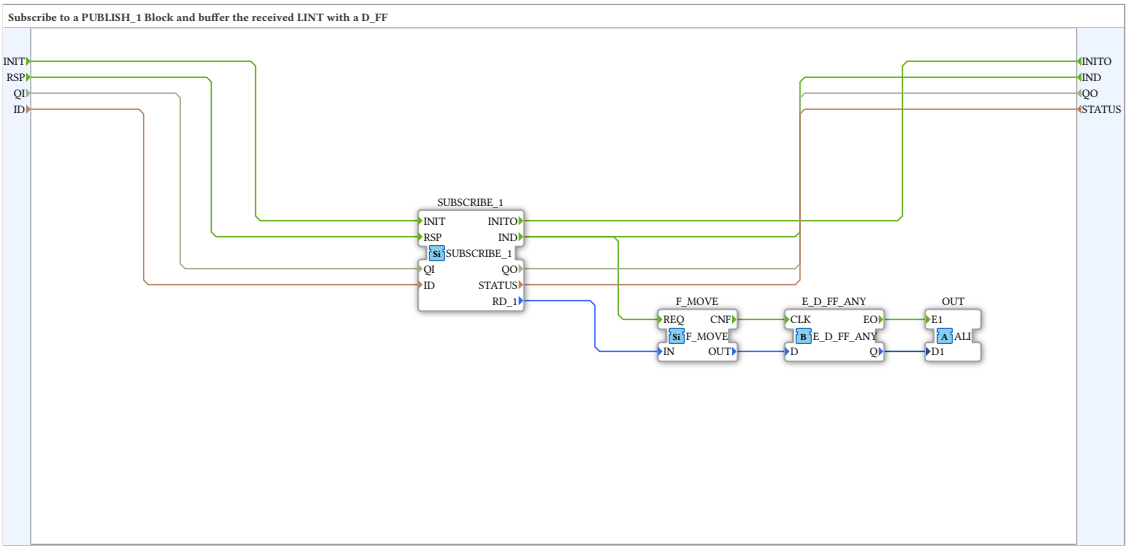
- **PUBLISH_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`REQ`, `SD_1`, …). **ALI_PUBLISH_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den ALI-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen LINT-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **ALI_PUBLISH_1** vereint Puffern und Veröffentlichen in einem Baustein.

Fazit

ALI_PUBLISH_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `PUBLISH_1` mit der Adapter-basierten LINT-Verarbeitung. Die interne D-Flipflop-Pufferung stabilisiert den Sendewert und entkoppelt Rückkopplungsschleifen; die reduzierte ALI-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

ALI_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ALI_SUBSCRIBE_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `SUBSCRIBE_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **ALI-Adapter** abbildet. Er abonniert einen über die `ID` konfigurierten Publish-Kanal und stellt den empfangenen **LINT**-Wert gepuffert über einen D-Flipflop (E_D_FF_ANY) am Adapter-Plug `OUT` bereit. Damit lassen sich Netzkommunikation und LINT-Signalverarbeitung in der ALI-Adapter-Welt sauber verbinden.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`
- **RSP** (Event): Response-Ereignis, verbunden mit `QI`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Abonnement-Dienst
- **ID** (WSTRING): Identifikator des zu abonnierenden Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
OUT	adapter::types::unidirectional::ALI	Plug (Ausgang)	Gepufferter empfangener LINT-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `SUBSCRIBE_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Der interne `SUBSCRIBE_1`-Block überwacht den konfigurierten Kanal. Sobald der zugehörige `PUBLISH_1`-Block Daten veröffentlicht, erzeugt er das Indikationsereignis `IND` und legt den empfangenen Wert an `RD_1` an.
3. `IND` taktet den internen **E_D_FF_ANY**-Flipflop (`CLK`), der den Wert von `RD_1` an seinem Dateneingang `D` übernimmt.
4. Der gespeicherte Wert erscheint stabil am Ausgang `Q` und wird über die Datenverbindung auf `OUT.D1` des AX-Plugs gelegt.
5. Das Ereignis `EO` des Flipflops wird auf den Ereignis-Ausgang `OUT.E1` gelegt, sodass nachgeschaltete Bausteine über die neue Information benachrichtigt werden.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der empfangene LINT-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert und bleibt stabil am Ausgang, bis der nächste Wert empfangen wird. `EO` feuert nur bei einer Änderung von `Q`.
- **RSP ist ein No-Op bei Subscribern**: Laut Quellcode-Analyse von `4diac-forte` (`CCommFB::sendData()`) führt das `RSP`-Ereignis bei `SUBSCRIBE_1` (Service-Typ Subscriber) keine Aktion aus – der Zweig `!= e_Subscriber` wird übersprungen. Das Event-Input-Paar `INIT`/`RSP` wird nur für die Quittierung durchgeführt und ist für die Quittierung im Subscriber-Fall funktionslos. Es wird dennoch nach außen geführt, damit ein konfigurierbarer Anschluss besteht.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `SUBSCRIBE_1` (inkl. `IND`/`RD_1`) wird nach innen verlegt; nach außen ist nur noch die ALI-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Das Abonnement ist aktiv; der Block wartet auf Daten vom Publisher.
3. **Datenempfang**: Bei `IND` wird der empfangene LINT-Wert in das interne D-Flipflop übernommen und über `OUT` ausgegeben.

Anwendungsszenarien

- **Netzwerkkopplung in ALI-Anwendungen**: Empfang von LINT-Signalen (z. B. Schaltbefehle oder Statusbits) über Publish/Subscribe und Weiterleitung über ALI-Adapter an die Adapter-basierte Logik.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_PUBLISH_1` und einem echten ereignisbasierten `E_D_FF_ANY` lässt sich die Rückkopplung zwischen `SUBSCRIBE_1` und `PUBLISH_1` entkoppeln (sauberer Fix gegenüber dem Interim-Fix mit `AX_D_FF`).
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf ALI-Adapter setzen.

Vergleich mit ähnlichen Bausteinen

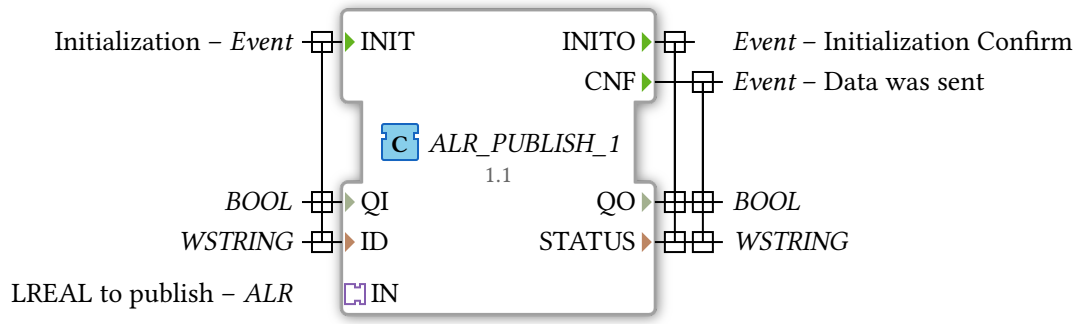
- **SUBSCRIBE_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`IND`, `RD_1`, …). **ALI_SUBSCRIBE_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den ALI-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen LINT-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **ALI_SUBSCRIBE_1** vereint Subscribing und Pufferung in einem Baustein.

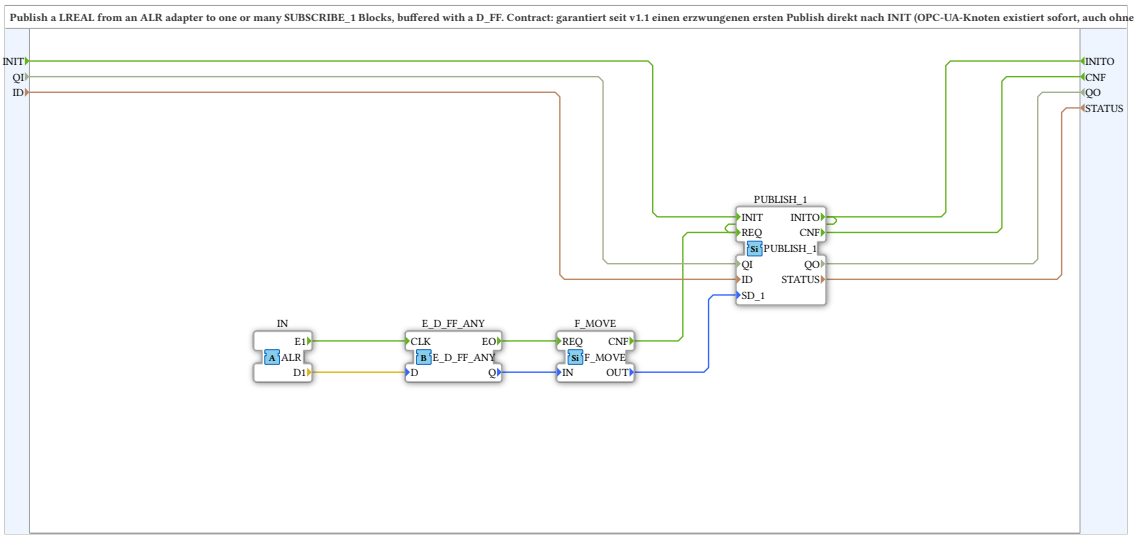
Fazit

ALI_SUBSCRIBE_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `SUBSCRIBE_1` mit der Adapter-basierten LINT-Verarbeitung. Die interne D-Flipflop-Pufferung hält den

empfangenen Wert stabil, bis neue Daten eintreffen, und die reduzierte ALI-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

ALR_PUBLISH_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ALR_PUBLISH_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `PUBLISH_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **ALR-Adapter** abbildet. Ein am Adapter-Socket `IN` anliegender **LREAL**-Wert wird über ein internes D-Flipflop (E_D_FF_ANY) gepuffert und anschließend über den konfigurierten Publish-Kanal an alle zugehörigen `SUBSCRIBE_1`-Empfänger gesendet.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`
- **CNF** (Event): Bestätigung, dass die Daten gesendet wurden, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Publish-Dienst
- **ID** (WSTRING): Identifikator des Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::ALR	Socket (Eingang)	Zu veröffentlichender LREAL-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `PUBLISH_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Sobald der ALR-Socket `IN` ein Ereignis auf `IN.E1` liefert, wird der gleichzeitig an `IN.D1` anliegende LREAL-Wert in das interne **E_D_FF_ANY**-Flipflop übernommen (`CLK` bzw. `D`).
3. Das Flipflop hält den Wert stabil an seinem Ausgang `Q` und erzeugt bei einer Wertänderung das Ereignis `EO`.
4. `EO` löst das Sendeereignis `REQ` des internen `PUBLISH_1`-Blocks aus; der gepufferte Wert `Q` wird über die Datenverbindung als `SD_1` übergeben und auf dem konfigurierten Kanal veröffentlicht.
5. Nach erfolgreichem Senden bestätigt der `PUBLISH_1`-Block mit `CNF`, das nach außen geführt wird.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der zu sendende LREAL-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert. Dadurch wird verhindert, dass ein sich während der Verarbeitung ändernder Eingangswert den Sendevorgang stört.
- **Ereignisbasierte Entkopplung**: Durch das echte ereignisbasierte `E_D_FF_ANY` werden unmittelbare Rückkopplungsschleifen zwischen `PUBLISH_1` und `SUBSCRIBE_1` (z. B. auf demselben OPC-UA-Knoten) entkoppelt – der saubere Fix gegenüber dem Interim-Fix mit `AX_D_FF`.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `PUBLISH_1` (inkl. `REQ`/`SD_1`) wird nach innen verlegt; nach außen ist nur noch die ALR-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Der Block ist bereit, Daten über den konfigurierten Kanal zu veröffentlichen.
3. **Sendeaktiv**: Ein am ALR-Socket eintreffendes Ereignis puffert den Wert und veranlasst das Senden über den internen `PUBLISH_1`-Block.

Anwendungsszenarien

- **Netzwerkkopplung in ALR-Anwendungen**: Senden von LREAL-Signalen (z. B. Schaltbefehle oder Statusbits) aus der Adapter-basierten Logik über Publish/Subscribe an entfernte Empfänger.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_SUBSCRIBE_1` werden Rückkopplungen zwischen eigenen Veröffentlichungen und dem eigenen Abonnement entkoppelt.
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf ALR-Adapter setzen.

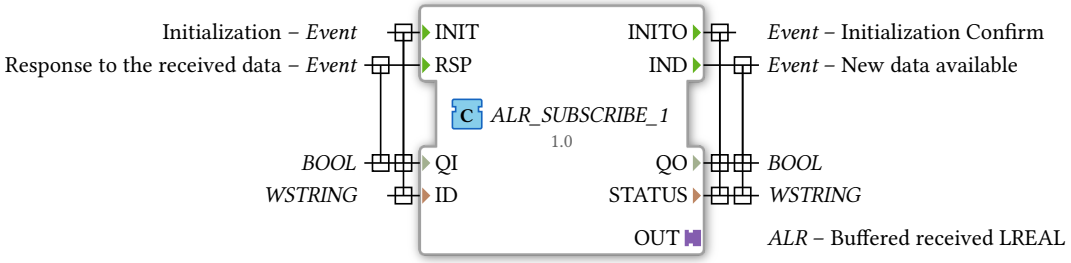
Vergleich mit ähnlichen Bausteinen

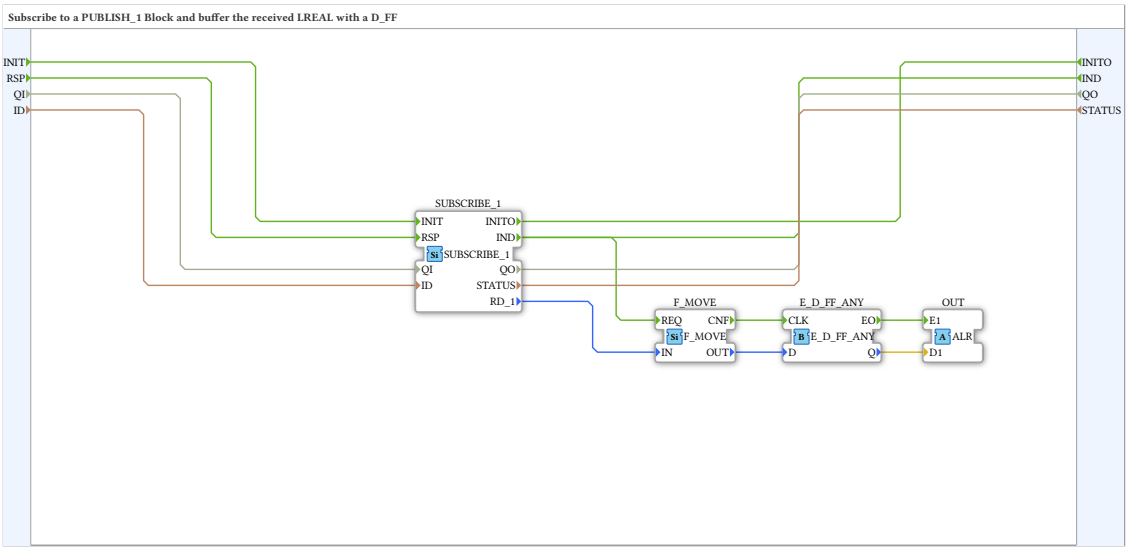
- **PUBLISH_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`REQ`, `SD_1`, …). **ALR_PUBLISH_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den ALR-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen LREAL-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **ALR_PUBLISH_1** vereint Puffern und Veröffentlichen in einem Baustein.

Fazit

ALR_PUBLISH_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `PUBLISH_1` mit der Adapter-basierten LREAL-Verarbeitung. Die interne D-Flipflop-Pufferung stabilisiert den Sendewert und entkoppelt Rückkopplungsschleifen; die reduzierte ALR-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

ALR_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ALR_SUBSCRIBE_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `SUBSCRIBE_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **ALR-Adapter** abbildet. Er abonniert einen über die `ID` konfigurierten Publish-Kanal und stellt den empfangenen **LREAL**-Wert gepuffert über einen D-Flipflop (E_D_FF_ANY) am Adapter-Plug `OUT` bereit. Damit lassen sich Netzkommunikation und LREAL-Signalverarbeitung in der ALR-Adapter-Welt sauber verbinden.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`
- **RSP** (Event): Response-Ereignis, verbunden mit `QI`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Abonnement-Dienst
- **ID** (WSTRING): Identifikator des zu abonnierenden Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung

OUT	adapter::types::unidirectional::ALR	Plug (Ausgang)	Gepufferter empfangener LREAL-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `SUBSCRIBE_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Der interne `SUBSCRIBE_1`-Block überwacht den konfigurierten Kanal. Sobald der zugehörige `PUBLISH_1`-Block Daten veröffentlicht, erzeugt er das Indikationsereignis `IND` und legt den empfangenen Wert an `RD_1` an.
3. `IND` taktet den internen **E_D_FF_ANY**-Flipflop (`CLK`), der den Wert von `RD_1` an seinem Dateneingang `D` übernimmt.
4. Der gespeicherte Wert erscheint stabil am Ausgang `Q` und wird über die Datenverbindung auf `OUT.D1` des AX-Plugs gelegt.
5. Das Ereignis `EO` des Flipflops wird auf den Ereignis-Ausgang `OUT.E1` gelegt, sodass nachgeschaltete Bausteine über die neue Information benachrichtigt werden.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der empfangene LREAL-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert und bleibt stabil am Ausgang, bis der nächste Wert empfangen wird. `EO` feuert nur bei einer Änderung von `Q`.
- **RSP ist ein No-Op bei Subscribern**: Laut Quellcode-Analyse von `4diac-forte` (`CCommFB::sendData()`) führt das `RSP`-Ereignis bei `SUBSCRIBE_1` (Service-Typ Subscriber) keine Aktion aus – der Zweig `!= e_Subscriber` wird übersprungen. Das Event-Input-Paar `INIT`/`RSP` wird nur für die Quittierung durchgeführt und ist für die Quittierung im Subscriber-Fall funktionslos. Es wird dennoch nach außen geführt, damit ein konfigurierbarer Anschluss besteht.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `SUBSCRIBE_1` (inkl. `IND`/`RD_1`) wird nach innen verlegt; nach außen ist nur noch die ALR-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Das Abonnement ist aktiv; der Block wartet auf Daten vom Publisher.
3. **Datenempfang**: Bei `IND` wird der empfangene LREAL-Wert in das interne D-Flipflop übernommen und über `OUT` ausgegeben.

Anwendungsszenarien

- **Netzwerkkopplung in ALR-Anwendungen**: Empfang von LREAL-Signalen (z. B. Schaltbefehle oder Statusbits) über Publish/Subscribe und Weiterleitung über ALR-Adapter an die Adapter-basierte Logik.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_PUBLISH_1` und einem echten ereignisbasierten `E_D_FF_ANY` lässt sich die Rückkopplung zwischen `SUBSCRIBE_1` und `PUBLISH_1` entkoppeln (sauberer Fix gegenüber dem Interim-Fix mit `AX_D_FF`).
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf ALR-Adapter setzen.

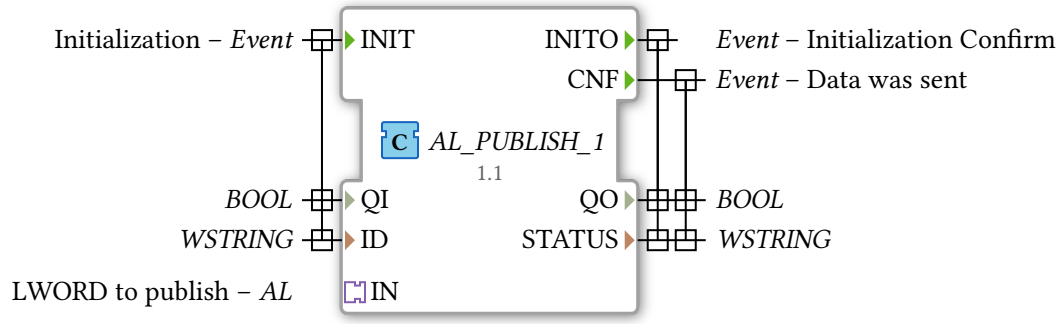
Vergleich mit ähnlichen Bausteinen

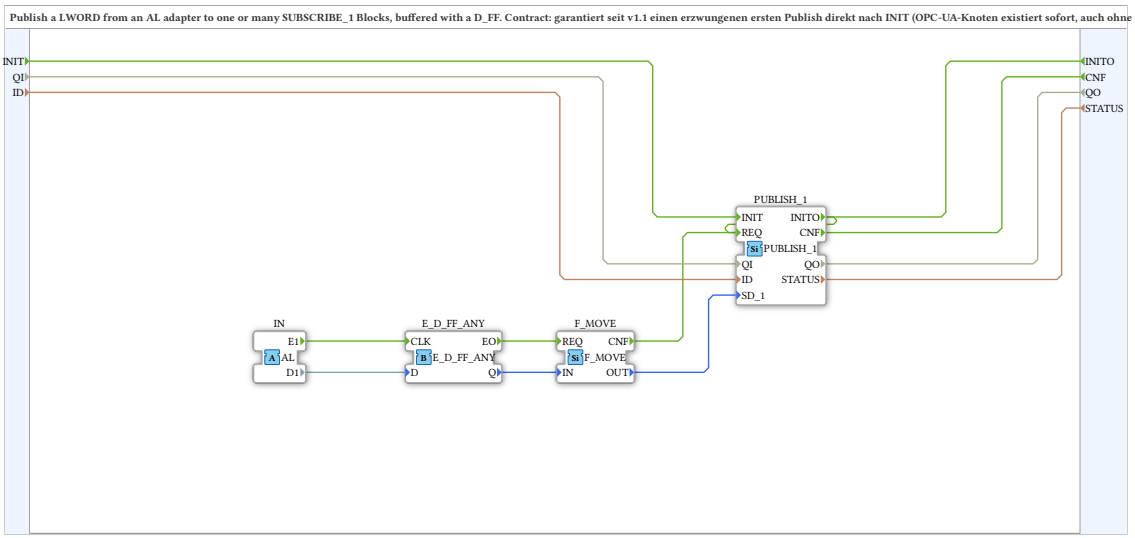
- **SUBSCRIBE_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`IND`, `RD_1`, …). **ALR_SUBSCRIBE_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den ALR-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen LREAL-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **ALR_SUBSCRIBE_1** vereint Subscribing und Pufferung in einem Baustein.

Fazit

ALR_SUBSCRIBE_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `SUBSCRIBE_1` mit der Adapter-basierten LREAL-Verarbeitung. Die interne D-Flipflop-Pufferung hält den empfangenen Wert stabil, bis neue Daten eintreffen, und die reduzierte ALR-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

AL_PUBLISH_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AL_PUBLISH_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `PUBLISH_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AL-Adapter** abbildet. Ein am Adapter-Socket `IN` anliegender **LWORD**-Wert wird über ein internes D-Flipflop (E_D_FF_ANY) gepuffert und anschließend über den konfigurierten Publish-Kanal an alle zugehörigen `SUBSCRIBE_1`-Empfänger gesendet.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`
- **CNF** (Event): Bestätigung, dass die Daten gesendet wurden, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Publish-Dienst
- **ID** (WSTRING): Identifikator des Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::AL	Socket (Eingang)	Zu veröffentlichender LWORD-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `PUBLISH_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Sobald der AL-Socket `IN` ein Ereignis auf `IN.E1` liefert, wird der gleichzeitig an `IN.D1` anliegende LWORD-Wert in das interne **E_D_FF_ANY**-Flipflop übernommen (`CLK` bzw. `D`).
3. Das Flipflop hält den Wert stabil an seinem Ausgang `Q` und erzeugt bei einer Wertänderung das Ereignis `EO`.
4. `EO` löst das Sendeereignis `REQ` des internen `PUBLISH_1`-Blocks aus; der gepufferte Wert `Q` wird über die Datenverbindung als `SD_1` übergeben und auf dem konfigurierten Kanal veröffentlicht.
5. Nach erfolgreichem Senden bestätigt der `PUBLISH_1`-Block mit `CNF`, das nach außen geführt wird.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der zu sendende LWORD-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert. Dadurch wird verhindert, dass ein sich während der Verarbeitung ändernder Eingangswert den Sendevorgang stört.
- **Ereignisbasierte Entkopplung**: Durch das echte ereignisbasierte `E_D_FF_ANY` werden unmittelbare Rückkopplungsschleifen zwischen `PUBLISH_1` und `SUBSCRIBE_1` (z. B. auf demselben OPC-UA-Knoten) entkoppelt – der saubere Fix gegenüber dem Interim-Fix mit `AX_D_FF`.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `PUBLISH_1` (inkl. `REQ`/`SD_1`) wird nach innen verlegt; nach außen ist nur noch die AL-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Der Block ist bereit, Daten über den konfigurierten Kanal zu veröffentlichen.
3. **Sendeaktiv**: Ein am AL-Socket eintreffendes Ereignis puffert den Wert und veranlasst das Senden über den internen `PUBLISH_1`-Block.

Anwendungsszenarien

- **Netzwerkkopplung in AL-Anwendungen**: Senden von LWORD-Signalen (z. B. Schaltbefehle oder Statusbits) aus der Adapter-basierten Logik über Publish/Subscribe an entfernte Empfänger.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_SUBSCRIBE_1` werden Rückkopplungen zwischen eigenen Veröffentlichungen und dem eigenen Abonnement entkoppelt.
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AL-Adapter setzen.

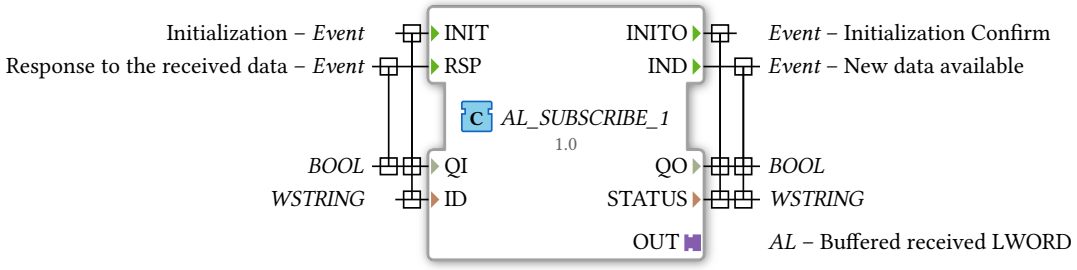
Vergleich mit ähnlichen Bausteinen

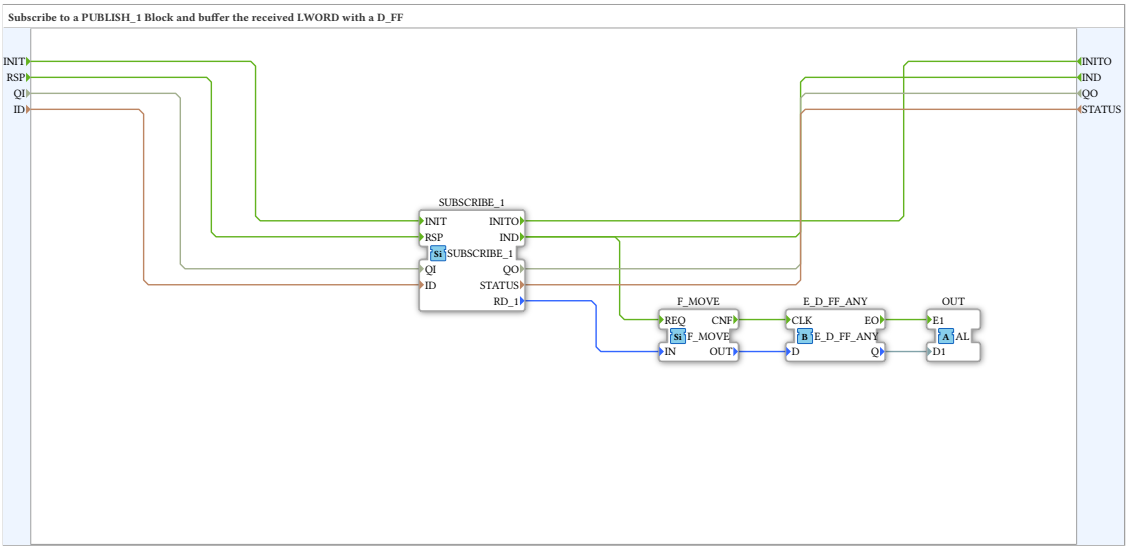
- **PUBLISH_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`REQ`, `SD_1`, …). **AL_PUBLISH_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den AL-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen LWORD-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AL_PUBLISH_1** vereint Puffern und Veröffentlichen in einem Baustein.

Fazit

AL_PUBLISH_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `PUBLISH_1` mit der Adapter-basierten LWORD-Verarbeitung. Die interne D-Flipflop-Pufferung stabilisiert den Sendewert und entkoppelt Rückkopplungsschleifen; die reduzierte AL-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

AL_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AL_SUBSCRIBE_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `SUBSCRIBE_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AL-Adapter** abbildet. Er abonniert einen über die `ID` konfigurierten Publish-Kanal und stellt den empfangenen **LWORD**-Wert gepuffert über einen D-Flipflop (E_D_FF_ANY) am Adapter-Plug `OUT` bereit. Damit lassen sich Netzkommunikation und LWORD-Signalverarbeitung in der AL-Adapter-Welt sauber verbinden.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`
- **RSP** (Event): Response-Ereignis, verbunden mit `QI`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Abonnement-Dienst
- **ID** (WSTRING): Identifikator des zu abonnierenden Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung

OUT	adapter::types::unidirectional::AL	Plug (Ausgang)	Gepufferter empfangener LWORD-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `SUBSCRIBE_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Der interne `SUBSCRIBE_1`-Block überwacht den konfigurierten Kanal. Sobald der zugehörige `PUBLISH_1`-Block Daten veröffentlicht, erzeugt er das Indikationsereignis `IND` und legt den empfangenen Wert an `RD_1` an.
3. `IND` taktet den internen **E_D_FF_ANY**-Flipflop (`CLK`), der den Wert von `RD_1` an seinem Dateneingang `D` übernimmt.
4. Der gespeicherte Wert erscheint stabil am Ausgang `Q` und wird über die Datenverbindung auf `OUT.D1` des AX-Plugs gelegt.
5. Das Ereignis `EO` des Flipflops wird auf den Ereignis-Ausgang `OUT.E1` gelegt, sodass nachgeschaltete Bausteine über die neue Information benachrichtigt werden.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der empfangene LWORD-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert und bleibt stabil am Ausgang, bis der nächste Wert empfangen wird. `EO` feuert nur bei einer Änderung von `Q`.
- **RSP ist ein No-Op bei Subscribern**: Laut Quellcode-Analyse von `4diac-forte` (`CCommFB::sendData()`) führt das `RSP`-Ereignis bei `SUBSCRIBE_1` (Service-Typ Subscriber) keine Aktion aus – der Zweig `!= e_Subscriber` wird übersprungen. Das Event-Input-Paar `INIT`/`RSP` wird nur für die Quittierung durchgeführt und ist für die Quittierung im Subscriber-Fall funktionslos. Es wird dennoch nach außen geführt, damit ein konfigurierbarer Anschluss besteht.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `SUBSCRIBE_1` (inkl. `IND`/`RD_1`) wird nach innen verlegt; nach außen ist nur noch die AL-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Das Abonnement ist aktiv; der Block wartet auf Daten vom Publisher.
3. **Datenempfang**: Bei `IND` wird der empfangene LWORD-Wert in das interne D-Flipflop übernommen und über `OUT` ausgegeben.

Anwendungsszenarien

- **Netzwerkkopplung in AL-Anwendungen**: Empfang von LWORD-Signalen (z. B. Schaltbefehle oder Statusbits) über Publish/Subscribe und Weiterleitung über AL-Adapter an die Adapter-basierte Logik.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_PUBLISH_1` und einem echten ereignisbasierten `E_D_FF_ANY` lässt sich die Rückkopplung zwischen `SUBSCRIBE_1` und `PUBLISH_1` entkoppeln (sauberer Fix gegenüber dem Interim-Fix mit `AX_D_FF`).
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AL-Adapter setzen.

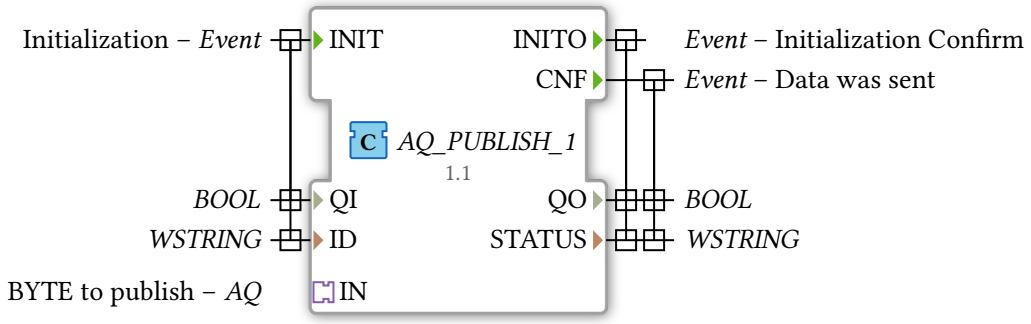
Vergleich mit ähnlichen Bausteinen

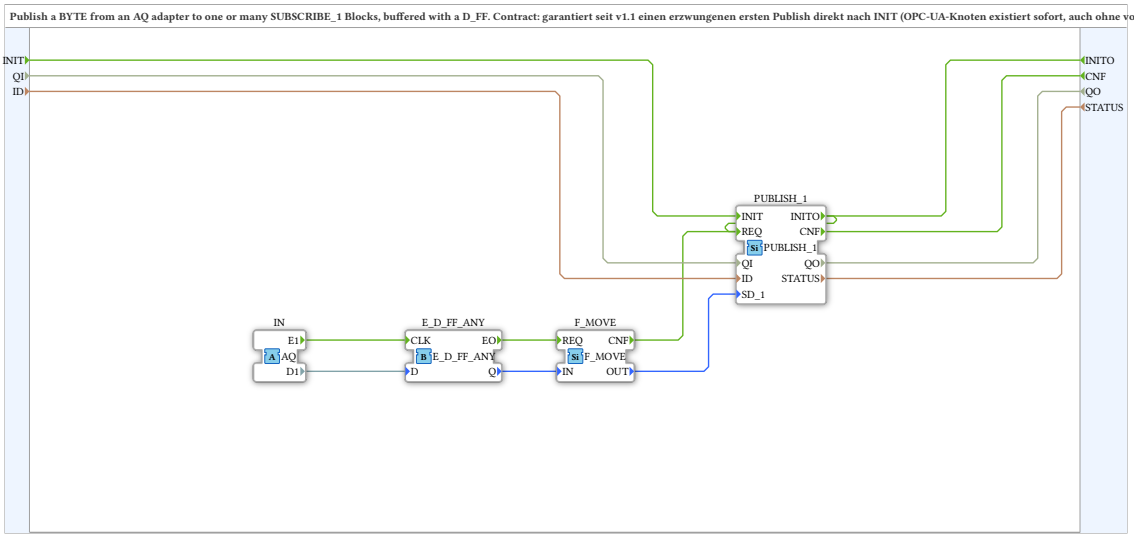
- **SUBSCRIBE_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`IND`, `RD_1`, …). **AL_SUBSCRIBE_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den AL-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen LWORD-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AL_SUBSCRIBE_1** vereint Subscribing und Pufferung in einem Baustein.

Fazit

AL_SUBSCRIBE_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `SUBSCRIBE_1` mit der Adapter-basierten LWORD-Verarbeitung. Die interne D-Flipflop-Pufferung hält den empfangenen Wert stabil, bis neue Daten eintreffen, und die reduzierte AL-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

AQ_PUBLISH_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AQ_PUBLISH_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `PUBLISH_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AQ-Adapter** abbildet. Ein am Adapter-Socket `IN` anliegender **BYTE**-Wert wird über ein internes D-Flipflop (E_D_FF_ANY) gepuffert und anschließend über den konfigurierten Publish-Kanal an alle zugehörigen `SUBSCRIBE_1`-Empfänger gesendet.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`
- **CNF** (Event): Bestätigung, dass die Daten gesendet wurden, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Publish-Dienst
- **ID** (WSTRING): Identifikator des Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::AQ	Socket (Eingang)	Zu veröffentlichender BYTE-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `PUBLISH_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Sobald der AQ-Socket `IN` ein Ereignis auf `IN.E1` liefert, wird der gleichzeitig an `IN.D1` anliegende BYTE-Wert in das interne **E_D_FF_ANY**-Flipflop übernommen (`CLK` bzw. `D`).
3. Das Flipflop hält den Wert stabil an seinem Ausgang `Q` und erzeugt bei einer Wertänderung das Ereignis `EO`.
4. `EO` löst das Sendeereignis `REQ` des internen `PUBLISH_1`-Blocks aus; der gepufferte Wert `Q` wird über die Datenverbindung als `SD_1` übergeben und auf dem konfigurierten Kanal veröffentlicht.
5. Nach erfolgreichem Senden bestätigt der `PUBLISH_1`-Block mit `CNF`, das nach außen geführt wird.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der zu sendende BYTE-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert. Dadurch wird verhindert, dass ein sich während der Verarbeitung ändernder Eingangswert den Sendevorgang stört.
- **Ereignisbasierte Entkopplung**: Durch das echte ereignisbasierte `E_D_FF_ANY` werden unmittelbare Rückkopplungsschleifen zwischen `PUBLISH_1` und `SUBSCRIBE_1` (z. B. auf demselben OPC-UA-Knoten) entkoppelt – der saubere Fix gegenüber dem Interim-Fix mit `AX_D_FF`.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `PUBLISH_1` (inkl. `REQ`/`SD_1`) wird nach innen verlegt; nach außen ist nur noch die AQ-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Der Block ist bereit, Daten über den konfigurierten Kanal zu veröffentlichen.
3. **Sendeaktiv**: Ein am AQ-Socket eintreffendes Ereignis puffert den Wert und veranlasst das Senden über den internen `PUBLISH_1`-Block.

Anwendungsszenarien

- **Netzwerkkopplung in AQ-Anwendungen**: Senden von BYTE-Signalen (z. B. Schaltbefehle oder Statusbits) aus der Adapter-basierten Logik über Publish/Subscribe an entfernte Empfänger.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_SUBSCRIBE_1` werden Rückkopplungen zwischen eigenen Veröffentlichungen und dem eigenen Abonnement entkoppelt.
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AQ-Adapter setzen.

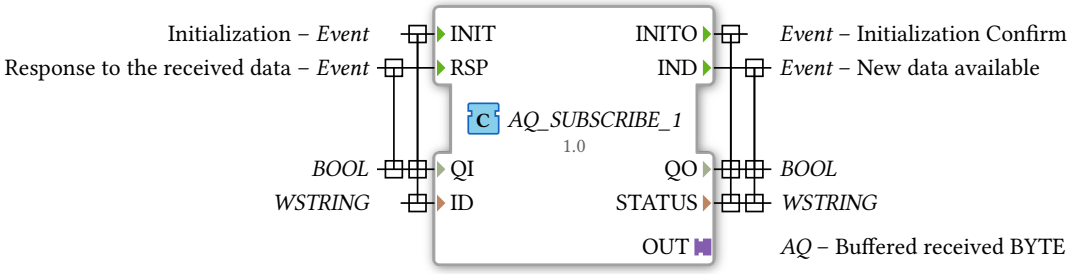
Vergleich mit ähnlichen Bausteinen

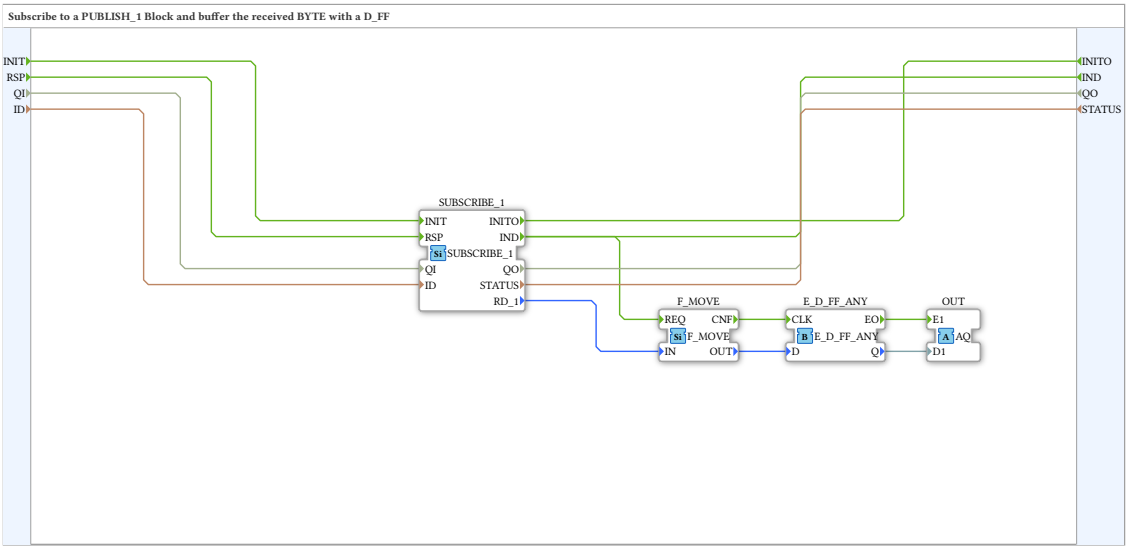
- **PUBLISH_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`REQ`, `SD_1`, …). **AQ_PUBLISH_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den AQ-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen BYTE-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AQ_PUBLISH_1** vereint Puffern und Veröffentlichen in einem Baustein.

Fazit

AQ_PUBLISH_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `PUBLISH_1` mit der Adapter-basierten BYTE-Verarbeitung. Die interne D-Flipflop-Pufferung stabilisiert den Sendewert und entkoppelt Rückkopplungsschleifen; die reduzierte AQ-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

AQ_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AQ_SUBSCRIBE_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `SUBSCRIBE_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AQ-Adapter** abbildet. Er abonniert einen über die `ID` konfigurierten Publish-Kanal und stellt den empfangenen **BYTE**-Wert gepuffert über einen D-Flipflop (E_D_FF_ANY) am Adapter-Plug `OUT` bereit. Damit lassen sich Netzkommunikation und BYTE-Signalverarbeitung in der AQ-Adapter-Welt sauber verbinden.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`
- **RSP** (Event): Response-Ereignis, verbunden mit `QI`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Abonnement-Dienst
- **ID** (WSTRING): Identifikator des zu abonnierenden Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
OUT	adapter::types::unidirectional::AQ	Plug (Ausgang)	Gepufferter empfangener BYTE-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `SUBSCRIBE_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Der interne `SUBSCRIBE_1`-Block überwacht den konfigurierten Kanal. Sobald der zugehörige `PUBLISH_1`-Block Daten veröffentlicht, erzeugt er das Indikationsereignis `IND` und legt den empfangenen Wert an `RD_1` an.
3. `IND` taktet den internen **E_D_FF_ANY**-Flipflop (`CLK`), der den Wert von `RD_1` an seinem Dateneingang `D` übernimmt.
4. Der gespeicherte Wert erscheint stabil am Ausgang `Q` und wird über die Datenverbindung auf `OUT.D1` des AX-Plugs gelegt.
5. Das Ereignis `EO` des Flipflops wird auf den Ereignis-Ausgang `OUT.E1` gelegt, sodass nachgeschaltete Bausteine über die neue Information benachrichtigt werden.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der empfangene BYTE-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert und bleibt stabil am Ausgang, bis der nächste Wert empfangen wird. `EO` feuert nur bei einer Änderung von `Q`.
- **RSP ist ein No-Op bei Subscribern**: Laut Quellcode-Analyse von `4diac-forte` (`CCommFB::sendData()`) führt das `RSP`-Ereignis bei `SUBSCRIBE_1` (Service-Typ Subscriber) keine Aktion aus – der Zweig `!= e_Subscriber` wird übersprungen. Das Event-Input-Paar `INIT`/`RSP` wird nur für die Quittierung durchgeführt und ist für die Quittierung im Subscriber-Fall funktionslos. Es wird dennoch nach außen geführt, damit ein konfigurierbarer Anschluss besteht.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `SUBSCRIBE_1` (inkl. `IND`/`RD_1`) wird nach innen verlegt; nach außen ist nur noch die AQ-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Das Abonnement ist aktiv; der Block wartet auf Daten vom Publisher.
3. **Datenempfang**: Bei `IND` wird der empfangene BYTE-Wert in das interne D-Flipflop übernommen und über `OUT` ausgegeben.

Anwendungsszenarien

- **Netzwerkkopplung in AQ-Anwendungen**: Empfang von BYTE-Signalen (z. B. Schaltbefehle oder Statusbits) über Publish/Subscribe und Weiterleitung über AQ-Adapter an die Adapter-basierte Logik.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_PUBLISH_1` und einem echten ereignisbasierten `E_D_FF_ANY` lässt sich die Rückkopplung zwischen `SUBSCRIBE_1` und `PUBLISH_1` entkoppeln (sauberer Fix gegenüber dem Interim-Fix mit `AX_D_FF`).
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AQ-Adapter setzen.

Vergleich mit ähnlichen Bausteinen

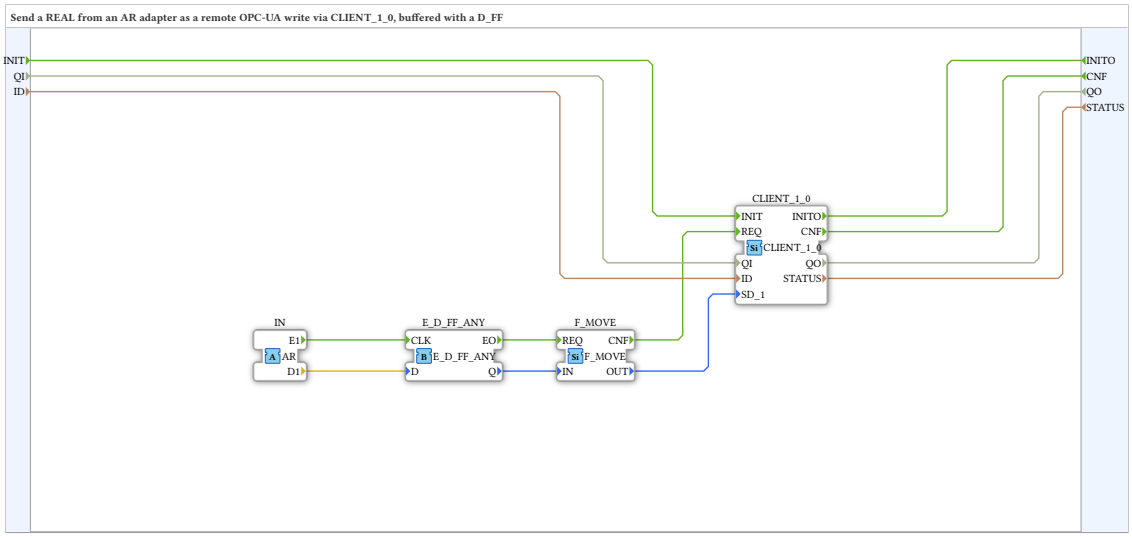
- **SUBSCRIBE_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`IND`, `RD_1`, …). **AQ_SUBSCRIBE_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den AQ-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen BYTE-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AQ_SUBSCRIBE_1** vereint Subscribing und Pufferung in einem Baustein.

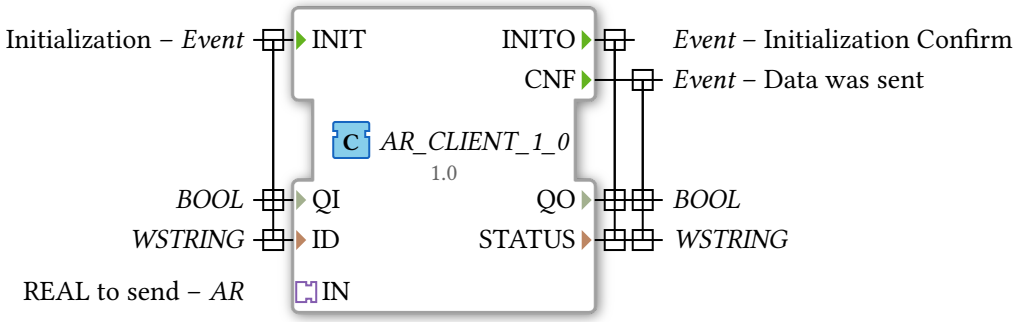
Fazit

AQ_SUBSCRIBE_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `SUBSCRIBE_1` mit der Adapter-basierten BYTE-Verarbeitung. Die interne D-Flipflop-Pufferung hält den

empfangenen Wert stabil, bis neue Daten eintreffen, und die reduzierte AQ-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

AR_CLIENT_1_0





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AR_CLIENT_1_0** ist ein Composite-Funktionsblock, der den netzwerkbasierten `CLIENT_1_0`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AR-Adapter** (Datentyp `REAL`) abbildet. Ein am Adapter-Socket `IN` anliegender **REAL**-Wert wird über ein internes `E_D_FF_ANY`-Flipflop gepuffert und anschließend über `CLIENT_1_0` als OPC-UA-**Write** oder Methodenaufruf (`CALL_METHOD`) an den unter `ID` konfigurierten entfernten Knoten gesendet.

Im Unterschied zu **AR_PUBLISH_1** (lokales Publish/Subscribe) schreibt `CLIENT_1_0` aktiv auf einen **entfernten** Server.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`
- **CNF** (Event): Bestätigung, dass die Daten gesendet wurden, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, öffnet (TRUE) bzw. schließt (FALSE) die Verbindung zum Server
- **ID** (WSTRING): Identifikator der Verbindung (OPC-UA-Adresse des Zielknotens, z. B. `opc_ua[WRITE;opc.tcp://192.168.1.12:4840#;...]`)

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Verbindungsstatus
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::AR	Socket (Eingang)	Zu sendender REAL-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `CLIENT_1_0`-Block mit `QI` und `ID` initialisiert; er baut die Verbindung zum entfernten Server auf und quittiert bei Erfolg mit `INITO`.
2. Sobald der AR-Socket `IN` ein Ereignis auf `IN.E1` liefert, wird der an `IN.D1` anliegende REAL-Wert in das interne `E_D_FF_ANY`-Flipflop übernommen.
3. Das Flipflop hält den Wert stabil und erzeugt das Ereignis `EO`.
4. `EO` stößt über ein `F_MOVE` (Datentyp `REAL`) die Zuweisung an `CLIENT_1_0.SD_1` an und löst das Sendeereignis `REQ` des `CLIENT_1_0`-Blocks aus.
5. Nach erfolgreichem Senden bestätigt der `CLIENT_1_0`-Block mit `CNF`.

Technische Besonderheiten

- **Pufferung mit E_D_FF_ANY**: Der zu sendende REAL-Wert wird über ein internes `E_D_FF_ANY` gepuffert, um Störungen durch sich während des Sendevorgangs ändernde Eingangswerte zu verhindern.
- **Remote-Write / Call-Method**: `CLIENT_1_0` adressiert einen entfernten OPC-UA-Server direkt, um analoge Messwerte oder Sollwerte zu übertragen.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `CLIENT_1_0` wird nach innen verlegt; nach außen ist nur noch die AR-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Die Verbindung zum entfernten Server ist aufgebaut, der Block ist bereit zu senden.
3. **Sendeaktiv**: Ein am AR-Socket eintreffendes Ereignis puffert den REAL-Wert und führt den Remote-Write aus.

Anwendungsszenarien

- **Fernübertragung von Messwerten**: Übertragung von analogen Sensorwerten (z. B. Druck, Temperatur, Position) an eine entfernte Steuerung oder ein HMI via OPC UA.
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilten Analogwerten in Bibliotheken, die durchgängig auf AR-Adapter setzen.

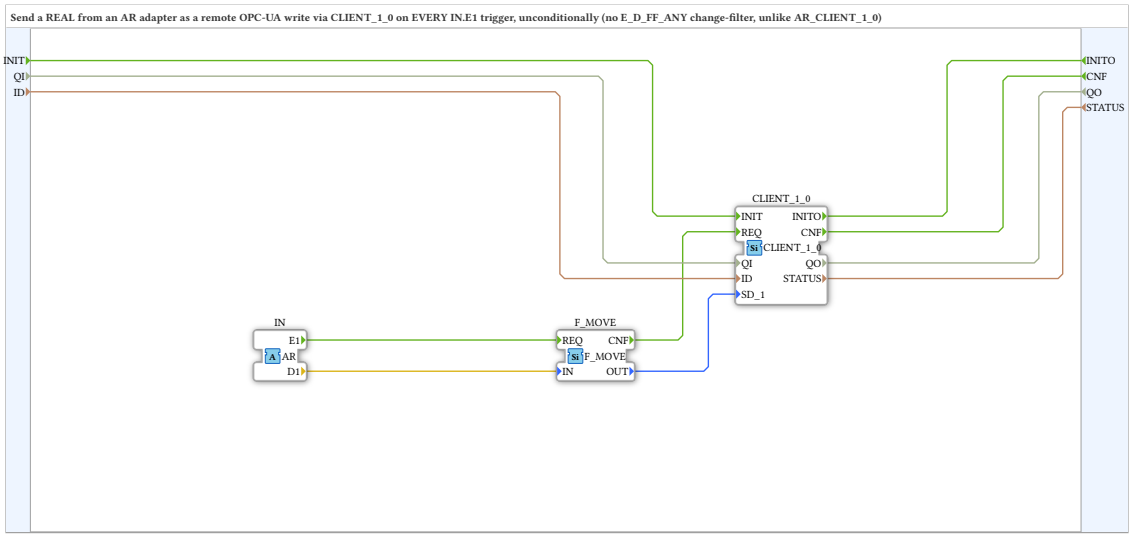
Vergleich mit ähnlichen Bausteinen

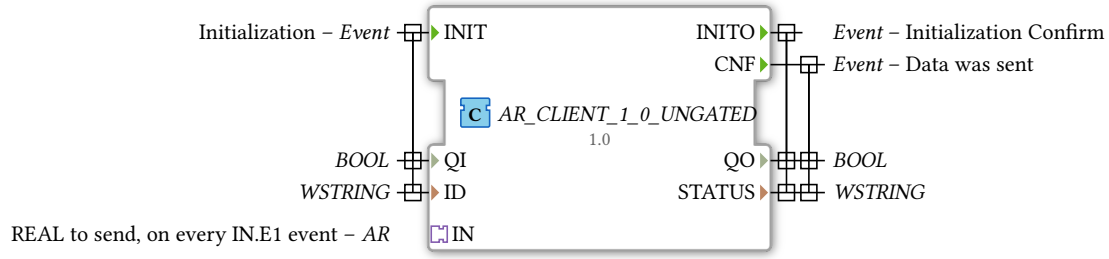
- **AR_CLIENT_1_0_UNGATED**: Verzichtet auf das interne `E_D_FF_ANY`-Flipflop (Change-Filter) und löst bei jedem `IN.E1`-Ereignis bedingungslos den Remote-Write bzw. Methodenaufruf aus — ideal für wiederholte Taster-/VT-Eingaben desselben Werts.
- **AX_CLIENT_1_0**: Identischer Aufbau, verarbeitet jedoch BOOL-Werte über einen AX-Adapter.
- **ATM_CLIENT_1_0**: Identischer Aufbau, verarbeitet TIME-Werte über einen ATM-Adapter.
- **AR_PUBLISH_1**: Kapselt `PUBLISH_1` statt `CLIENT_1_0` für lokale Publish/Subscribe-Verbindungen.

Fazit

AR_CLIENT_1_0 verbindet die verbindungsorientierte Remote-Write-Kommunikation des Standardbausteins `CLIENT_1_0` mit der Adapter-basierten Real-Wert-Verarbeitung.

AR_CLIENT_1_0_UNGATED





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AR_CLIENT_1_0_UNGATED** ist ein Composite-Funktionsblock aus dem Paket `adapter::net`. Er kapselt den netzwerkbasierten `CLIENT_1_0`-Funktionsblock aus der IEC 61499-Standardbibliothek und stellt dessen Schnittstelle auf einen unidirektionalen **AR-Adapter** (`IN`, Datentyp `REAL`) bereit.

Im Gegensatz zu **AR_CLIENT_1_0** enthält dieser Baustein **keinen** internen `E_D_FF_ANY`-Wert-Filter (Send-on-Change). Jedes eintreffende Ereignis an `IN.E1` löst **ungefiltert und bedingungslos** ein Sendeereignis an `CLIENT_1_0.REQ` aus.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`
- **CNF** (Event): Bestätigung, dass die Daten gesendet wurden, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, öffnet (TRUE) bzw. schließt (FALSE) die Verbindung zum Server
- **ID** (WSTRING): Identifikator der Verbindung (OPC-UA-Adresse des Zielknotens, z. B. `opc_ua[CALL_METHOD;opc.tcp://192.168.1.12:4840#;...]`)

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Verbindungsstatus
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::AR	Socket (Eingang)	Zu sendender REAL-Wert (wird bei jedem `IN.E1` gesendet)

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `CLIENT_1_0`-Block mit `QI` und `ID` initialisiert und quittiert bei Erfolg mit `INITO`.
2. Sobald der AR-Socket `IN` ein Ereignis auf `IN.E1` liefert, wird das Ereignis **direkt** an `F_MOVE.REQ` weitergeleitet.
3. `F_MOVE` (`REAL`) übernimmt den an `IN.D1` anliegenden REAL-Wert, leitet ihn an `CLIENT_1_0.SD_1` weiter und löst `CLIENT_1_0.REQ` aus.
4. Der `CLIENT_1_0`-Block führt den OPC-UA-Remote-Write bzw. den Methodenaufruf aus und bestätigt mit `CNF`.

Technische Besonderheiten

- **Ungefilterte Ereignisauslösung (Ungated)**: Da kein `E_D_FF_ANY`-Flipflop vorgeschaltet ist, wird kein Change-Filter angewendet. Selbst wenn der Wert `IN.D1` identisch zum zuletzt gesendeten Wert ist, führt ein erneutes `IN.E1`-Ereignis zur Ausführung.
- **Speziell für OPC-UA CALL_METHOD / Remote-Input**: Unverzichtbar für Bedienkomponenten (z. B. VT-Eingabefelder oder Web-HMIs), bei denen die wiederholte Eingabe desselben Zahlenwerts zuverlässig einen erneuten Remote-Methodenaufruf auslösen muss.
- **Entkopplung vom Change-Filter**: Während `AR_CLIENT_1_0` für kontinuierliche Sende-bei-Änderung-Anwendungsfälle optimiert ist, dient `AR_CLIENT_1_0_UNGATED` der ereignisgesteuerten Aufrufausführung.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Verbindung zum Server steht; bereit für Sendeereignisse.
3. **Sendeaktiv**: Jedes `IN.E1`-Ereignis führt unmittelbar zur Übertragung via `CLIENT_1_0`.

Anwendungsszenarien

- **Remote-Methodenaufrufe über VT/HMI**: Z. B. `NumericValue_TO_CLIENT_1_0_OPC.SUB` — Wenn der Bediener einen Zahlenwert am VT bestätigt, muss die OPC-UA-Methode auch dann aufgerufen werden, wenn der Wert unverändert geblieben ist.
- **Ereignisgesteuerte Befehlsübermittlung**: Übertragung von REAL-Parametern, deren Sende-Zeitpunkt als Ereignis eine eigenständige Bedeutung hat.

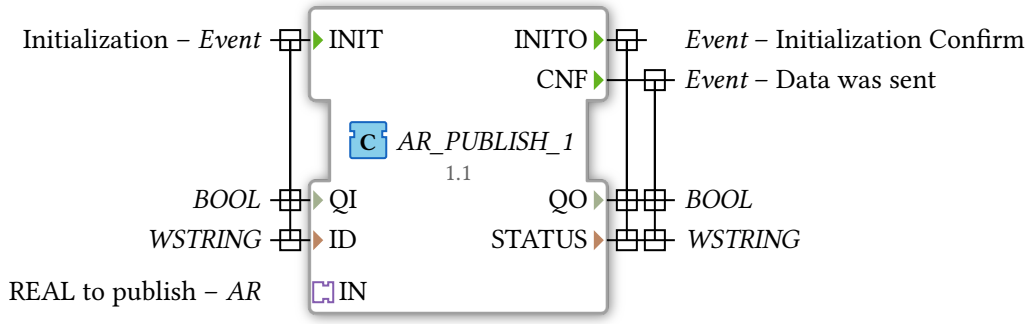
Vergleich mit ähnlichen Bausteinen

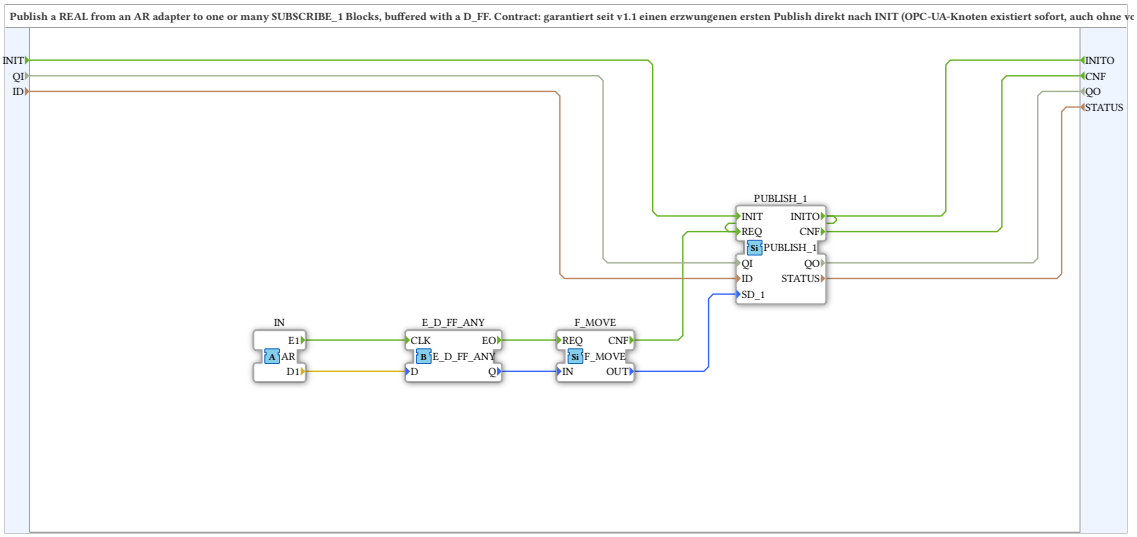
- **AR_CLIENT_1_0**: Enthält ein internes `E_D_FF_ANY`-Flipflop, das Werteänderungen filtert (Send-on-Change). `AR_CLIENT_1_0_UNGATED` verzichtet bewusst auf diesen Filter.
- **AX_CLIENT_1_0**: Verarbeitet BOOL-Werte mit Change-Filter.
- **ATM_CLIENT_1_0**: Verarbeitet TIME-Werte mit Change-Filter.

Fazit

AR_CLIENT_1_0_UNGATED stellt sicher, dass jedes Trigger-Ereignis an der AR-Adapter-Schnittstelle zu einem OPC-UA-Sendevorgang führt — ideal für Remote-Methodenaufrufe in HMI- und VT-Anwendungen.

AR_PUBLISH_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AR_PUBLISH_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `PUBLISH_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AR-Adapter** abbildet. Ein am Adapter-Socket `IN` anliegender **REAL**-Wert wird über ein internes D-Flipflop (E_D_FF_ANY) gepuffert und anschließend über den konfigurierten Publish-Kanal an alle zugehörigen `SUBSCRIBE_1`-Empfänger gesendet.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`
- **CNF** (Event): Bestätigung, dass die Daten gesendet wurden, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Publish-Dienst
- **ID** (WSTRING): Identifikator des Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::AR	Socket (Eingang)	Zu veröffentlichender REAL-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `PUBLISH_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Sobald der AR-Socket `IN` ein Ereignis auf `IN.E1` liefert, wird der gleichzeitig an `IN.D1` anliegende REAL-Wert in das interne **E_D_FF_ANY**-Flipflop übernommen (`CLK` bzw. `D`).
3. Das Flipflop hält den Wert stabil an seinem Ausgang `Q` und erzeugt bei einer Wertänderung das Ereignis `EO`.
4. `EO` löst das Sendeereignis `REQ` des internen `PUBLISH_1`-Blocks aus; der gepufferte Wert `Q` wird über die Datenverbindung als `SD_1` übergeben und auf dem konfigurierten Kanal veröffentlicht.
5. Nach erfolgreichem Senden bestätigt der `PUBLISH_1`-Block mit `CNF`, das nach außen geführt wird.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der zu sendende REAL-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert. Dadurch wird verhindert, dass ein sich während der Verarbeitung ändernder Eingangswert den Sendevorgang stört.
- **Ereignisbasierte Entkopplung**: Durch das echte ereignisbasierte `E_D_FF_ANY` werden unmittelbare Rückkopplungsschleifen zwischen `PUBLISH_1` und `SUBSCRIBE_1` (z. B. auf demselben OPC-UA-Knoten) entkoppelt – der saubere Fix gegenüber dem Interim-Fix mit `AX_D_FF`.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `PUBLISH_1` (inkl. `REQ`/`SD_1`) wird nach innen verlegt; nach außen ist nur noch die AR-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Der Block ist bereit, Daten über den konfigurierten Kanal zu veröffentlichen.
3. **Sendeaktiv**: Ein am AR-Socket eintreffendes Ereignis puffert den Wert und veranlasst das Senden über den internen `PUBLISH_1`-Block.

Anwendungsszenarien

- **Netzwerkkopplung in AR-Anwendungen**: Senden von REAL-Signalen (z. B. Schaltbefehle oder Statusbits) aus der Adapter-basierten Logik über Publish/Subscribe an entfernte Empfänger.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_SUBSCRIBE_1` werden Rückkopplungen zwischen eigenen Veröffentlichungen und dem eigenen Abonnement entkoppelt.
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AR-Adapter setzen.

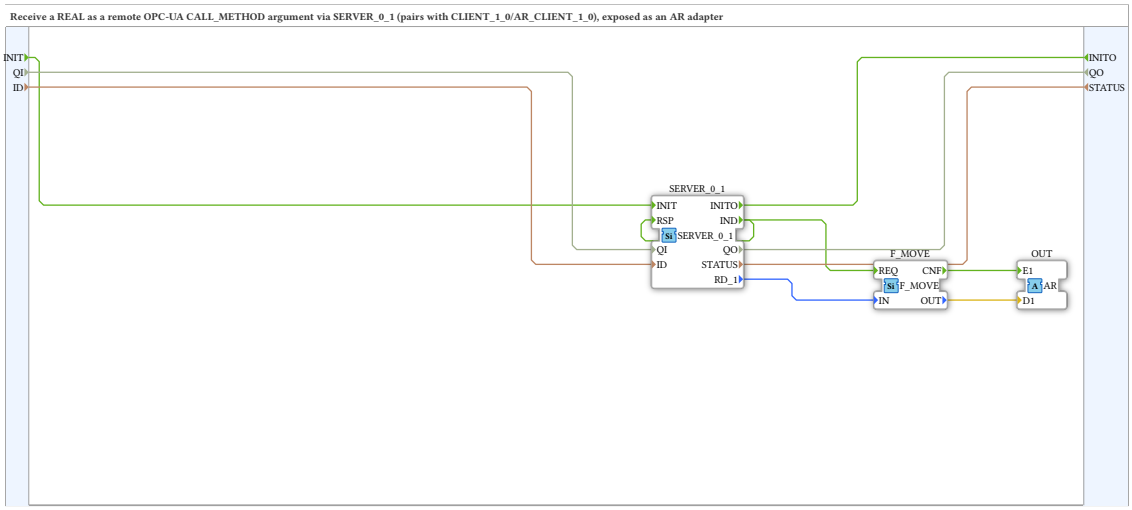
Vergleich mit ähnlichen Bausteinen

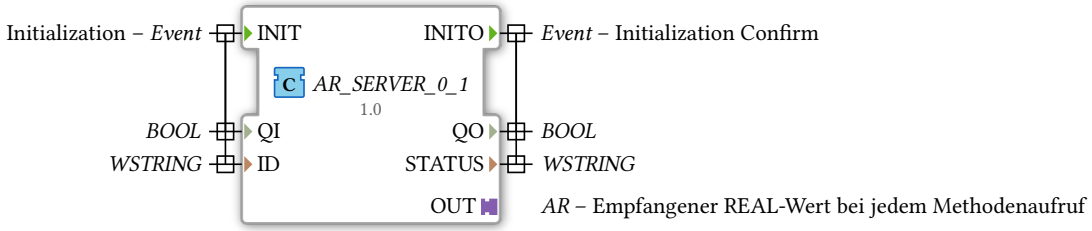
- **PUBLISH_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`REQ`, `SD_1`, …). **AR_PUBLISH_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den AR-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen REAL-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AR_PUBLISH_1** vereint Puffern und Veröffentlichen in einem Baustein.

Fazit

AR_PUBLISH_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `PUBLISH_1` mit der Adapter-basierten REAL-Verarbeitung. Die interne D-Flipflop-Pufferung stabilisiert den Sendewert und entkoppelt Rückkopplungsschleifen; die reduzierte AR-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

AR_SERVER_0_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AR_SERVER_0_1** ist ein Composite-Funktionsblock aus dem Paket `adapter::net`. Er kapselt den netzwerkbasierten Server-Baustein `iec61499::net::SERVER_0_1` und stellt empfangene OPC-UA-Methodenaufrufe (`CALL_METHOD`) für **REAL**-Argumente als unidirektionalen **AR-Adapter** (`OUT`) bereit. Er bildet das Empfangs-Gegenstück zu **AR_CLIENT_1_0** (bzw. `CLIENT_1_0`).

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID` (leitet an `SERVER_0_1.INIT` weiter)

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung (`SERVER_0_1.INITO`), verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, schaltet den Server aktiv (TRUE) bzw. inaktiv (FALSE)
- **ID** (WSTRING): Server-Identifikator (OPC-UA-Serveradresse und Methodenspezifikation)

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Server-Betriebsstatus
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
OUT	adapter::types::unidirectional::AR	Plug (Ausgang)	Empfangener REAL-Wert bei jedem Methodenaufruf

Funktionsweise

1. Ein `INIT`-Ereignis an `AR_SERVER_0_1` wird an den internen `SERVER_0_1.INIT`-Eingang weitergeleitet. Der Server startet den OPC-UA-Endpunkt und quittiert bei Erfolg mit `INITO`.
2. Bei einem eintreffenden Methodenaufruf von einem entfernten Client (`CLIENT_1_0/AR_CLIENT_1_0`) erzeugt `SERVER_0_1` das Indikations-Ereignis `IND` und stellt den empfangenen REAL-Wert an `RD_1` bereit.
3. Zur Vermeidung von Antwort-Latenz-Sperren (RSP-Latenz-Bug) ist `SERVER_0_1.IND` im internen Netzwerk direkt auf `SERVER_0_1.RSP` zurückverdrahtet. Der Server quittiert die Methode dadurch sofort ohne Verzögerung.
4. Gleichzeitig leitet `IND` den empfangenen Wert über `F_MOVE` (Datentyp `REAL`) an den AR-Plug `OUT.D1` weiter und löst das Ausgangs-Ereignis `OUT.E1` aus.

Technische Besonderheiten

- **Direkte RSP-Rückverdrahtung**: Um 4-Sekunden-Server-Sperren durch verzögerte Antworten zu vermeiden, wird `SERVER_0_1.IND` intern unmittelbar auf `SERVER_0_1.RSP` zurückgekoppelt.
- **Nahtlose Adapter-Integration**: Der empfangene Rohwert `RD_1` von `SERVER_0_1` wird über ein `F_MOVE` (REAL) direkt auf die AR-Adapter-Struktur (`OUT.D1` / `OUT.E1`) abgebildet.
- **Empfangs-Gegenstück zu `AR_CLIENT_1_0`**: In IEC 61499 pairt ein `CLIENT_1_0`-Baustein („Connect to a SERVER_0_1 Block“) mit einem `SERVER_0_1`-Baustein.

Zustandsübersicht

1. **Nicht initialisiert**: Der Server ist inaktiv und wartet auf das `INIT`-Ereignis.
2. **Bereit (Listening)**: Der OPC-UA-Server ist aktiv und wartet auf Methodenaufrufe von entfernten Clients.
3. **Empfang (Indikation)**: Ein Methodenaufruf trifft ein; der REAL-Wert wird an den `OUT`-Adapter übergeben und die Quittierung sofort gesendet.

Anwendungsszenarien

- **Empfang von Fernsteuerungs-Sollwerten**: Empfangen von analogen Einstellwerten (z. B. Soll-Druck, Soll-Drehzahl) von einem entfernten Steuerungsmodul via OPC UA `CALL_METHOD`.
- **Modulare Server-Architekturen**: Bereitstellung von OPC-UA-Methoden-Endpunkten in durchgängig adapterbasierten Anwendungen.

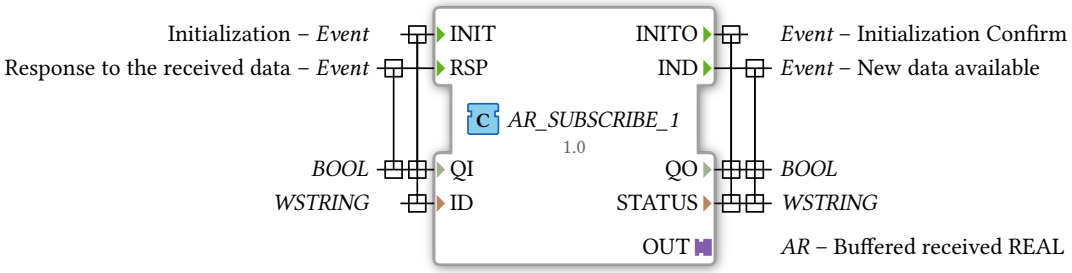
Vergleich mit ähnlichen Bausteinen

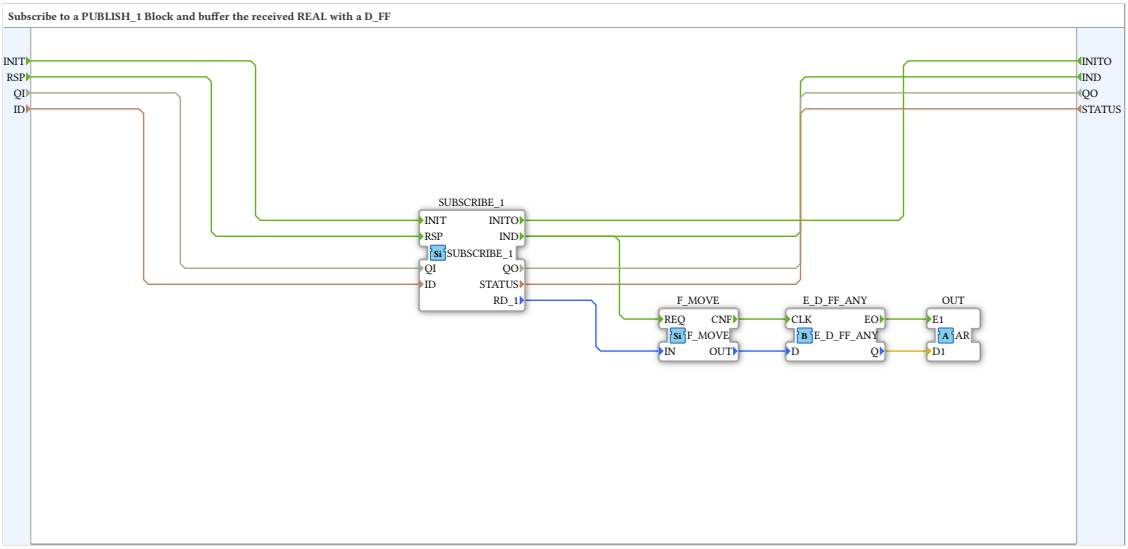
- **AX_SERVER_0_1**: Empfängt BOOL-Werte über einen AX-Adapter.
- **ATM_SERVER_0_1**: Empfängt TIME-Werte über einen ATM-Adapter.
- **AR_SUBSCRIBE_1**: Kapselt `SUBSCRIBE_1` für lokales Publish/Subscribe statt verbindungsorientierter Methodenaufrufe.

Fazit

AR_SERVER_0_1 stellt empfangene OPC-UA-Methodenaufrufe für REAL-Werte als sauberen AR-Adapter bereit und verhindert durch direkte RSP-Rückkopplung Server-Lockups.

AR_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AR_SUBSCRIBE_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `SUBSCRIBE_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AR-Adapter** abbildet. Er abonniert einen über die `ID` konfigurierten Publish-Kanal und stellt den empfangenen **REAL**-Wert gepuffert über einen D-Flipflop (E_D_FF_ANY) am Adapter-Plug `OUT` bereit. Damit lassen sich Netzkommunikation und REAL-Signalverarbeitung in der AR-Adapter-Welt sauber verbinden.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`
- **RSP** (Event): Response-Ereignis, verbunden mit `QI`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Abonnement-Dienst
- **ID** (WSTRING): Identifikator des zu abonnierenden Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
OUT	adapter::types::unidirectional::AR	Plug (Ausgang)	Gepufferter empfangener REAL-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `SUBSCRIBE_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Der interne `SUBSCRIBE_1`-Block überwacht den konfigurierten Kanal. Sobald der zugehörige `PUBLISH_1`-Block Daten veröffentlicht, erzeugt er das Indikationsereignis `IND` und legt den empfangenen Wert an `RD_1` an.

3. `IND` taktet den internen **E_D_FF_ANY**-Flipflop (`CLK`), der den Wert von `RD_1` an seinem Dateneingang `D` übernimmt.
4. Der gespeicherte Wert erscheint stabil am Ausgang `Q` und wird über die Datenverbindung auf `OUT.D1` des AX-Plugs gelegt.
5. Das Ereignis `EO` des Flipflops wird auf den Ereignis-Ausgang `OUT.E1` gelegt, sodass nachgeschaltete Bausteine über die neue Information benachrichtigt werden.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der empfangene REAL-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert und bleibt stabil am Ausgang, bis der nächste Wert empfangen wird. `EO` feuert nur bei einer Änderung von `Q`.
- **RSP ist ein No-Op bei Subscribern**: Laut Quellcode-Analyse von `4diac-forte` (`CCommFB::sendData()`) führt das `RSP`-Ereignis bei `SUBSCRIBE_1` (Service-Typ Subscriber) keine Aktion aus – der Zweig `!= e_Subscriber` wird übersprungen. Das Event-Input-Paar `INIT`/`RSP` wird nur für die Quittierung durchgeführt und ist für die Quittierung im Subscriber-Fall funktionslos. Es wird dennoch nach außen geführt, damit ein konfigurierbarer Anschluss besteht.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `SUBSCRIBE_1` (inkl. `IND`/`RD_1`) wird nach innen verlegt; nach außen ist nur noch die AR-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Das Abonnement ist aktiv; der Block wartet auf Daten vom Publisher.
3. **Datenempfang**: Bei `IND` wird der empfangene REAL-Wert in das interne D-Flipflop übernommen und über `OUT` ausgegeben.

Anwendungsszenarien

- **Netzwerkkopplung in AR-Anwendungen**: Empfang von REAL-Signalen (z. B. Schaltbefehle oder Statusbits) über Publish/Subscribe und Weiterleitung über AR-Adapter an die Adapter-basierte Logik.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_PUBLISH_1` und einem echten ereignisbasierten `E_D_FF_ANY` lässt sich die Rückkopplung zwischen `SUBSCRIBE_1` und `PUBLISH_1` entkoppeln (sauberer Fix gegenüber dem Interim-Fix mit `AX_D_FF`).
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AR-Adapter setzen.

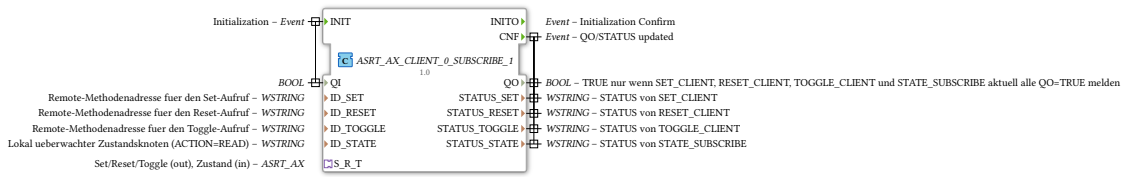
Vergleich mit ähnlichen Bausteinen

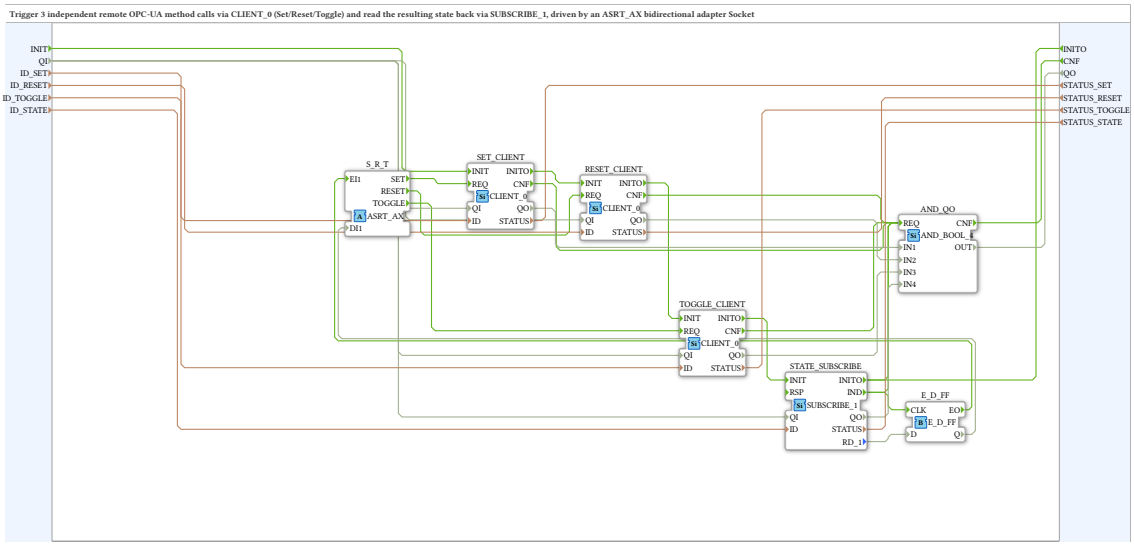
- **SUBSCRIBE_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`IND`, `RD_1`, …). **AR_SUBSCRIBE_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den AR-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen REAL-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AR_SUBSCRIBE_1** vereint Subscribing und Pufferung in einem Baustein.

Fazit

AR_SUBSCRIBE_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `SUBSCRIBE_1` mit der Adapter-basierten REAL-Verarbeitung. Die interne D-Flipflop-Pufferung hält den empfangenen Wert stabil, bis neue Daten eintreffen, und die reduzierte AR-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

ASRT_AX_CLIENT_0_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ASRT_AX_CLIENT_0_SUBSCRIBE_1** ist ein Composite-Funktionsblock, der drei unabhängige entfernte OPC-UA-**Methodenaufrufe** (Set, Reset und Toggle) über je einen eigenen `CLIENT_0`-Baustein auslöst und den dazugehörigen Zustand über `SUBSCRIBE_1` zurückliest – alles hinter einem einzigen bidirektionalen **ASRT_AX-Adapter-Socket**. Er ist die um einen Zustandskanal erweiterte Variante von ASRT_CLIENT_0.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO`, `STATUS_SET`, `STATUS_RESET`, `STATUS_TOGGLE` und `STATUS_STATE`
- **CNF** (Event): `QO`/`STATUS` aktualisiert, verbunden mit denselben Variablen

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang für alle vier internen Verbindungen
- **ID_SET** (WSTRING): Remote-Methodenadresse für den Set-Aufruf
- **ID_RESET** (WSTRING): Remote-Methodenadresse für den Reset-Aufruf
- **ID_TOGGLE** (WSTRING): Remote-Methodenadresse für den Toggle-Aufruf
- **ID_STATE** (WSTRING): Lokal überwachter Zustandsknoten (ACTION=READ)

Daten-Ausgänge

- **QO** (BOOL): TRUE nur wenn `SET_CLIENT`, `RESET_CLIENT`, `TOGGLE_CLIENT` und `STATE_SUBSCRIBE` aktuell alle `QO = TRUE` melden
- **STATUS_SET** (WSTRING): Statusinformationen von `SET_CLIENT`
- **STATUS_RESET** (WSTRING): Statusinformationen von `RESET_CLIENT`
- **STATUS_TOGGLE** (WSTRING): Statusinformationen von `TOGGLE_CLIENT`
- **STATUS_STATE** (WSTRING): Statusinformationen von `STATE_SUBSCRIBE`

Adapter

Adapter	Typ	Richtung	Beschreibung
S_R_T	adapter::types::bidirectional::ASRT_AX	Socket – Set/Reset/Toggle (Ausgang), Zustand (Eingang)	Set/Reset/Toggle nach außen, Zustand zurück

Funktionsweise

1. Über das `INIT`-Ereignis werden nacheinander `SET_CLIENT`, `RESET_CLIENT`, `TOGGLE_CLIENT` und `STATE_SUBSCRIBE` initialisiert. Nach allen vier Bestätigungen wird `INITO` nach außen gemeldet.
2. `S_R_T.SET`, `S_R_T.RESET` bzw. `S_R_T.TOGGLE` lösen direkt die jeweils zugehörigen `CLIENT_0`-Methodenaufrufe aus.
3. Sobald `STATE_SUBSCRIBE` per `IND` einen neuen Zustandswert liefert, taktet dies das interne **E_D_FF**-Flipflop, das den Wert übernimmt und stabil hält.
4. Der Ausgang `EO` des Flipflops feuert `S_R_T.EI1` mit dem gepufferten Wert an `S_R_T.DI1` – der Zustand wird über denselben Adapter zurückgemeldet, über den auch Set/Reset/Toggle ausgelöst werden.
5. `AND_QO` (AND_BOOL_4) verundet die `QO`-Ausgänge aller vier internen Bausteine; jede Bestätigung löst zusätzlich `CNF` nach außen aus.

Technische Besonderheiten

- **Vier Netzverbindungen hinter einem Adapter**: Drei `CLIENT_0`-Instanzen (Set, Reset, Toggle) und ein `SUBSCRIBE_1` (Zustand) werden zu einer einzigen bidirektionalen ASRT_AX-Schnittstelle zusammengefasst.
- **Pufferung mit D-Flipflop**: Der über `SUBSCRIBE_1` empfangene Zustand wird über ein internes `iec61499::events::E_D_FF` stabilisiert, bevor er über den Adapter zurückgemeldet wird.
- **Sequentielle Initialisierung**: `SET_CLIENT` → `RESET_CLIENT` → `TOGGLE_CLIENT` → `STATE_SUBSCRIBE` (Composite-FBTypes verketten `INIT`/`INITO` seriell durch jede Instanz).
- **Kapselung**: Nach außen ist nur noch die ASRT_AX-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Alle vier Verbindungen (Set, Reset, Toggle, Zustandsabo) sind aufgebaut.
3. **Sendeaktiv**: Ein am Adapter eintreffendes Set-, Reset- oder Toggle-Ereignis löst den entsprechenden Remote-Methodenaufruf aus.
4. **Zustandsaktualisierung**: Ein neuer Wert vom Abonnement wird gepuffert und über denselben Adapter zurückgemeldet.

Anwendungsszenarien

- Fernauslösen von Set/Reset/Toggle auf einem entfernten Aktor und gleichzeitiges Rücklesen des resultierenden Zustands über eine einzige Adapterverbindung
- Reduktion von händischer CLIENT_0/SUBSCRIBE_1-Verdrahtung auf einen einzigen wiederverwendbaren Baustein

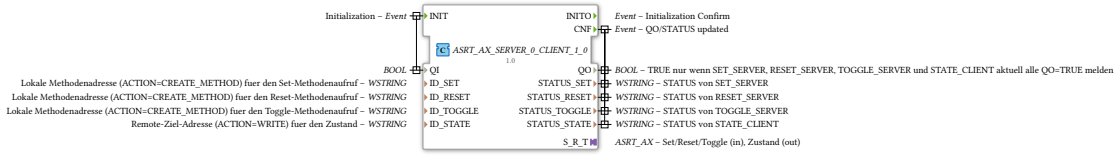
Vergleich mit ähnlichen Bausteinen

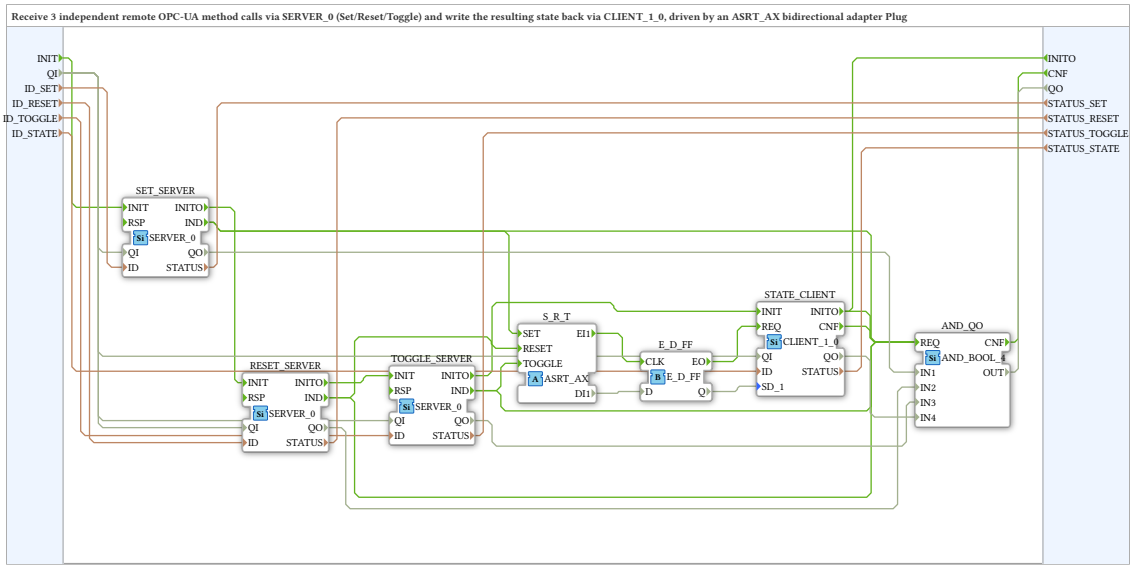
- **ASRT_CLIENT_0**: Dieselben drei Trigger, aber ohne Zustands-Rücklesung.
- **ASR_AX_CLIENT_0_SUBSCRIBE_1**: Dasselbe Muster mit nur zwei Triggern (ohne Toggle).
- **ASRT_AX_SERVER_0_CLIENT_1_0**: Die serverseitige Gegenstelle.

Fazit

ASRT_AX_CLIENT_0_SUBSCRIBE_1 bündelt drei Remote-Trigger (Set/Reset/Toggle) und ein Zustands-Abonnement hinter einem einzigen bidirektionalen ASRT_AX-Adapter und macht damit händisch verdrahtete Kombinationen zu einem einzigen wiederverwendbaren Baustein.

ASRT_AX_SERVER_0_CLIENT_1_0





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ASRT_AX_SERVER_0_CLIENT_1_0** ist ein Composite-Funktionsblock, der drei unabhängige entfernte OPC-UA-**Methodenaufrufe** (Set, Reset und Toggle) über je einen eigenen `SERVER_0`-Baustein empfängt und den resultierenden Zustand über `CLIENT_1_0` an einen entfernten Knoten zurückschreibt – alles hinter einem einzigen bidirektionalen **ASRT_AX-Adapter-Plug**. Er ist die serverseitige Gegenstelle zu ASRT_AX_CLIENT_0_SUBSCRIBE_1.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO`, `STATUS_SET`, `STATUS_RESET`, `STATUS_TOGGLE` und `STATUS_STATE`
- **CNF** (Event): `QO`/`STATUS` aktualisiert, verbunden mit denselben Variablen

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang für alle vier internen Verbindungen
- **ID_SET** (WSTRING): Lokale Methodenadresse (ACTION=CREATE_METHOD) für den Set-Methodenaufruf
- **ID_RESET** (WSTRING): Lokale Methodenadresse (ACTION=CREATE_METHOD) für den Reset-Methodenaufruf
- **ID_TOGGLE** (WSTRING): Lokale Methodenadresse (ACTION=CREATE_METHOD) für den Toggle-Methodenaufruf
- **ID_STATE** (WSTRING): Remote-Ziel-Adresse (ACTION=WRITE) für den Zustand

Daten-Ausgänge

- **QO** (BOOL): TRUE nur wenn `SET_SERVER`, `RESET_SERVER`, `TOGGLE_SERVER` und `STATE_CLIENT` aktuell alle `QO = TRUE` melden
- **STATUS_SET** (WSTRING): Statusinformationen von `SET_SERVER`
- **STATUS_RESET** (WSTRING): Statusinformationen von `RESET_SERVER`
- **STATUS_TOGGLE** (WSTRING): Statusinformationen von `TOGGLE_SERVER`
- **STATUS_STATE** (WSTRING): Statusinformationen von `STATE_CLIENT`

Adapter

Adapter	Typ	Richtung	Beschreibung

S_R_T	adapter::types::bidirectional::ASRT_AX	Plug – Set/Reset/Toggle (Eingang), Zustand (Ausgang)	Empfangenes Set/Reset/Toggle, Zustand nach außen

Funktionsweise

1. Über das `INIT`-Ereignis werden nacheinander `SET_SERVER`, `RESET_SERVER`, `TOGGLE_SERVER` und `STATE_CLIENT` initialisiert. Nach allen vier Bestätigungen wird `INITO` nach außen gemeldet.
2. Ein entfernter Set-, Reset- bzw. Toggle-Aufruf auf dem jeweiligen `SERVER_0`-Baustein löst `S_R_T.SET`, `S_R_T.RESET` bzw. `S_R_T.TOGGLE` aus – alle drei werden also über den Adapter nach innen gemeldet.
3. Sobald die weiterverarbeitende Logik über den Adapter ein Rückmeldeereignis `S_R_T.EI1` mit Datum `S_R_T.DI1` liefert, taktet dies das interne **E_D_FF**-Flipflop, das den Wert übernimmt und stabil hält.
4. Der Ausgang `EO` des Flipflops löst `STATE_CLIENT.REQ` aus; der gepufferte Wert wird per OPC-UA-Write an den unter `ID_STATE` konfigurierten entfernten Knoten gesendet.
5. `AND_QO` (AND_BOOL_4) verundet die `QO`-Ausgänge aller vier internen Bausteine; jede Bestätigung löst zusätzlich `CNF` nach außen aus.

Technische Besonderheiten

- **Vier Netzverbindungen hinter einem Adapter**: Drei `SERVER_0`-Instanzen (Set, Reset, Toggle) und ein `CLIENT_1_0` (Zustand) werden zu einer einzigen bidirektionalen ASRT_AX-Schnittstelle zusammengefasst.
- **Pufferung mit D-Flipflop**: Der über den Adapter gemeldete Rückmeldewert wird über ein internes `iec61499::events::E_D_FF` stabilisiert, bevor er per `CLIENT_1_0` verschickt wird.
- **Sequentielle Initialisierung**: `SET_SERVER` → `RESET_SERVER` → `TOGGLE_SERVER` → `STATE_CLIENT` (Composite-FBTypes verketten `INIT`/`INITO` seriell durch jede Instanz).
- **Kapselung**: Nach außen ist nur noch die ASRT_AX-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Alle vier Verbindungen (Set-, Reset-, Toggle-Server, Zustands-Client) sind aufgebaut.
3. **Empfangsaktiv**: Ein entfernter Set-, Reset- oder Toggle-Aufruf wird über den Adapter an die interne Logik gemeldet.
4. **Zustandsschreibvorgang**: Ein Rückmeldewert der internen Logik wird gepuffert und per Remote-Write an den konfigurierten Zielknoten gesendet.

Anwendungsszenarien

- Entgegennahme entfernter Set/Reset/Toggle-Aufrufe und gleichzeitiges Zurückschreiben des resultierenden Zustands an einen anderen entfernten Knoten über eine einzige Adapterverbindung
- Serverseitiges Gegenstück zu ASRT_AX_CLIENT_0_SUBSCRIBE_1 in verteilten Steuerungsarchitekturen

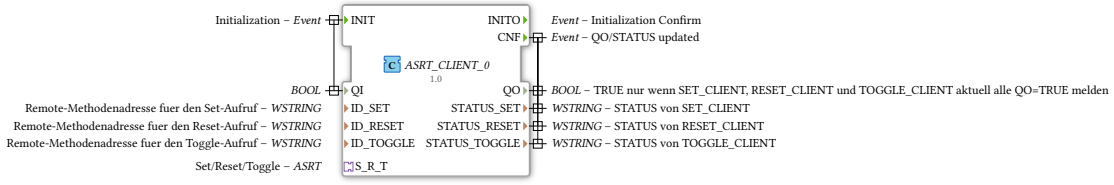
Vergleich mit ähnlichen Bausteinen

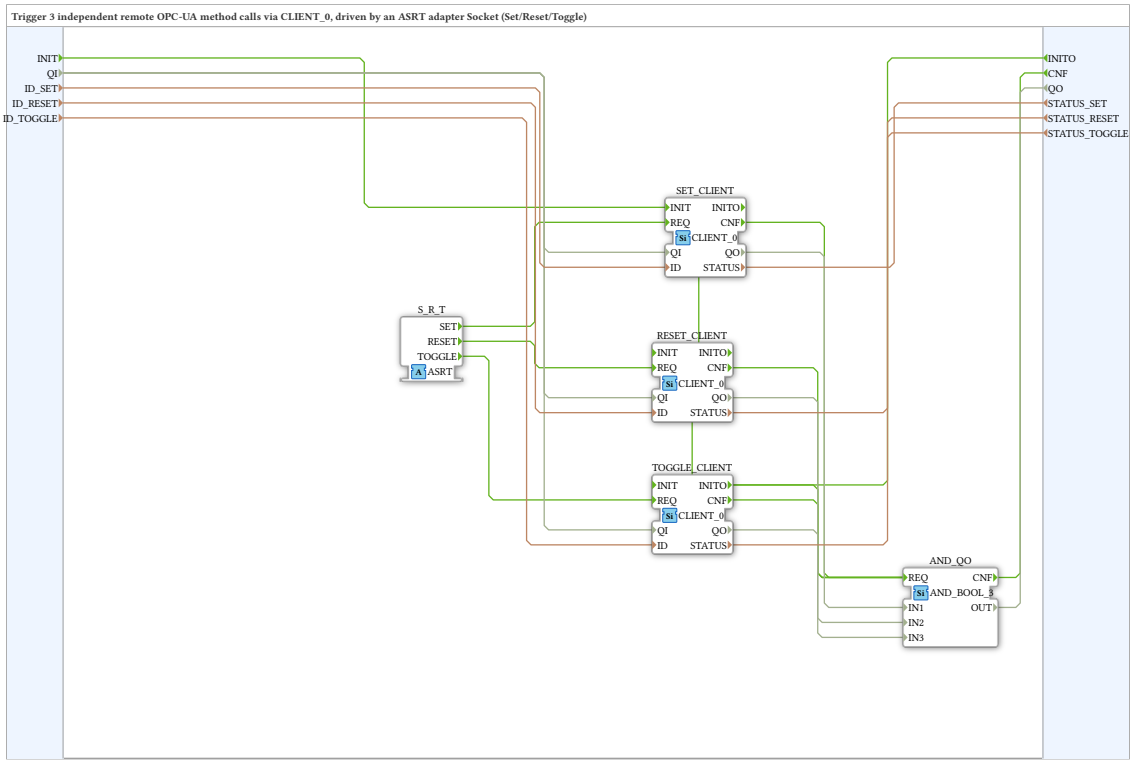
- **ASRT_AX_CLIENT_0_SUBSCRIBE_1**: Die client-/aufrufende Gegenstelle.
- **ASR_AX_SERVER_0_CLIENT_1_0**: Dasselbe Muster mit nur zwei empfangenden Servern (ohne Toggle).

Fazit

ASRT_AX_SERVER_0_CLIENT_1_0 bündelt drei empfangende Server (Set/Reset/Toggle) und einen zustandsschreibenden Client hinter einem einzigen bidirektionalen ASRT_AX-Adapter und bildet damit die serverseitige Entsprechung zu ASRT_AX_CLIENT_0_SUBSCRIBE_1.

ASRT_CLIENT_0





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ASRT_CLIENT_0** ist ein Composite-Funktionsblock, der drei unabhängige entfernte OPC-UA-**Methodenaufrufe** (Set, Reset und Toggle) über je einen eigenen `CLIENT_0`-Baustein auslöst, gesteuert über einen einzigen unidirektionalen **ASRT-Adapter-Socket**. Er ist die um einen dritten Trigger erweiterte Variante von ASR_CLIENT_0.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO`, `STATUS_SET`, `STATUS_RESET` und `STATUS_TOGGLE`
- **CNF** (Event): Bestätigung, dass einer der Methodenaufrufe gesendet wurde, verbunden mit denselben Statusvariablen

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, öffnet (TRUE) bzw. schließt (FALSE) alle drei Verbindungen
- **ID_SET** (WSTRING): Remote-Methodenadresse für den Set-Aufruf
- **ID_RESET** (WSTRING): Remote-Methodenadresse für den Reset-Aufruf
- **ID_TOGGLE** (WSTRING): Remote-Methodenadresse für den Toggle-Aufruf

Daten-Ausgänge

- **QO** (BOOL): TRUE nur wenn `SET_CLIENT`, `RESET_CLIENT` und `TOGGLE_CLIENT` aktuell alle `QO = TRUE` melden
- **STATUS_SET** (WSTRING): Statusinformationen von `SET_CLIENT`
- **STATUS_RESET** (WSTRING): Statusinformationen von `RESET_CLIENT`
- **STATUS_TOGGLE** (WSTRING): Statusinformationen von `TOGGLE_CLIENT`

Adapter

Adapter	Typ	Richtung	Beschreibung
S_R_T	adapter::types::unidirectional::ASRT	Socket (Eingang)	Set/Reset/Toggle

Funktionsweise

1. Über das `INIT`-Ereignis werden die internen `SET_CLIENT`-, `RESET_CLIENT`- und `TOGGLE_CLIENT`-Blöcke nacheinander (seriell verkettet über `INIT`/`INITO`) mit `QI` und ihrer jeweiligen `ID` initialisiert. Nach allen drei Bestätigungen wird `INITO` nach außen gemeldet.
2. Ein Ereignis am Adapter-Socket `S_R_T.SET`, `S_R_T.RESET` bzw. `S_R_T.TOGGLE` löst direkt den jeweils zugehörigen `CLIENT_0`-Methodenaufruf aus.
3. Alle drei `CLIENT_0`-Instanzen bestätigen unabhängig voneinander mit `CNF`, das jeweils nach außen weitergegeben wird.
4. Der Baustein `AND_QO` (AND_BOOL_3) verundet die `QO`-Ausgänge aller drei `CLIENT_0`-Instanzen zu einem gemeinsamen `QO`.

Technische Besonderheiten

- **Drei parallele CLIENT_0-Instanzen**: Set, Reset und Toggle werden über getrennte `CLIENT_0`-Bausteine (unterschiedliche `ID`, unabhängige Verbindungen) abgewickelt.
- **Sequentielle Initialisierung**: Die drei internen Blöcke werden nacheinander initialisiert (`SET_CLIENT.INITO` ⟶ `RESET_CLIENT.INIT` ⟶ `TOGGLE_CLIENT.INIT`), da Composite-FBTypes ihr `INIT`/`INITO` explizit durch jede Instanz seriell verketten müssen.
- **Kapselung**: Nach außen ist nur noch die ASRT-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Alle drei Verbindungen sind aufgebaut, der Block ist bereit, Set-, Reset- oder Toggle-Aufrufe auszulösen.
3. **Sendeaktiv**: Ein am ASRT-Socket eintreffendes Ereignis löst den entsprechenden Remote-Methodenaufruf aus.

Anwendungsszenarien

- Fernauslösen dreier unabhängiger entfernter Aktionen (Set/Reset/Toggle) über einen einzigen Adapter, z. B. zum Schalten eines entfernten Aktors mit zusätzlicher Toggle-Funktion
- Netzwerkkopplung in ASRT-Adapter-basierten Anwendungen

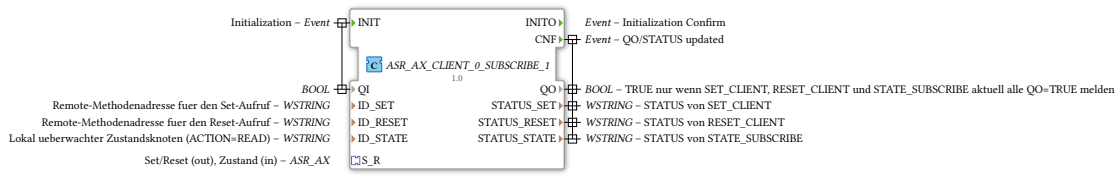
Vergleich mit ähnlichen Bausteinen

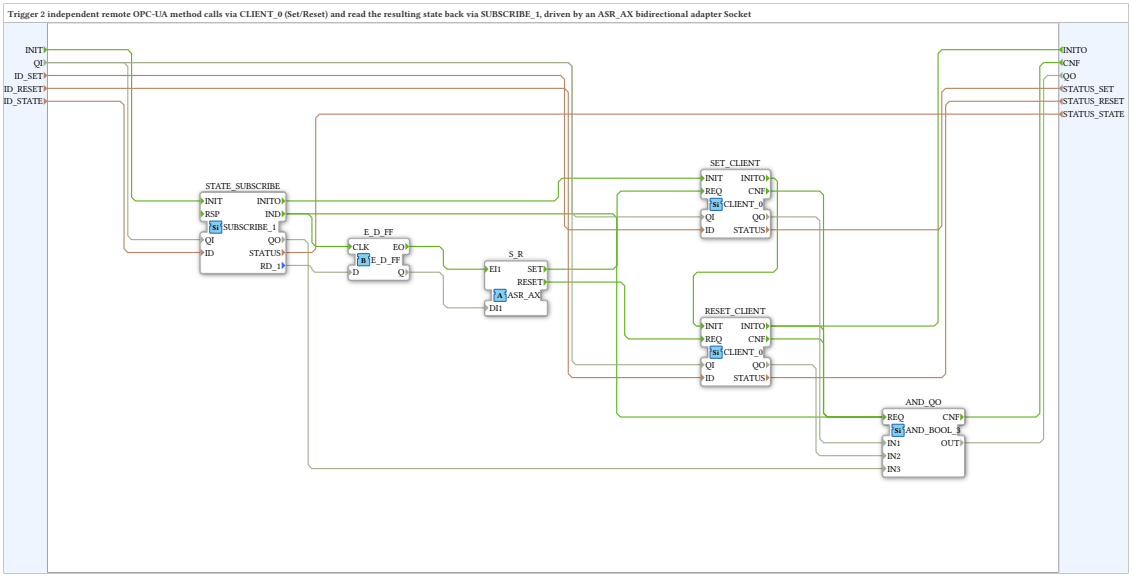
- **ASR_CLIENT_0**: Dieselbe Grundstruktur mit nur zwei Triggern (ohne Toggle).
- **AE_CLIENT_0**: Reduziert dasselbe Muster auf einen einzelnen Trigger.
- **ASRT_AX_CLIENT_0_SUBSCRIBE_1**: Kombiniert dieselben drei `CLIENT_0`-Trigger zusätzlich mit einem `SUBSCRIBE_1`-Zustandskanal hinter einem bidirektionalen Adapter.

Fazit

ASRT_CLIENT_0 bündelt drei unabhängige Remote-Methodenaufrufe (Set/Reset/Toggle) hinter einem einzigen ASRT-Adapter-Socket und eignet sich damit für Anwendungen, in denen ein entfernter Aktor über drei getrennte OPC-UA-Methoden geschaltet wird.

ASR_AX_CLIENT_0_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ASR_AX_CLIENT_0_SUBSCRIBE_1** ist ein Composite-Funktionsblock, der zwei unabhängige entfernte OPC-UA-**Methodenaufrufe** (Set und Reset) über je einen eigenen `CLIENT_0`-Baustein auslöst und den dazugehörigen Zustand über `SUBSCRIBE_1` zurückliest – alles hinter einem einzigen bidirektionalen **ASR_AX-Adapter-Socket**. Er ist die um einen Zustandskanal erweiterte Variante von ASR_CLIENT_0.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO`, `STATUS_SET`, `STATUS_RESET` und `STATUS_STATE`
- **CNF** (Event): `QO`/`STATUS` aktualisiert, verbunden mit denselben Variablen

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang für alle drei internen Verbindungen
- **ID_SET** (WSTRING): Remote-Methodenadresse für den Set-Aufruf
- **ID_RESET** (WSTRING): Remote-Methodenadresse für den Reset-Aufruf
- **ID_STATE** (WSTRING): Lokal überwachter Zustandsknoten (ACTION=READ)

Daten-Ausgänge

- **QO** (BOOL): TRUE nur wenn `SET_CLIENT`, `RESET_CLIENT` und `STATE_SUBSCRIBE` aktuell alle `QO = TRUE` melden
- **STATUS_SET** (WSTRING): Statusinformationen von `SET_CLIENT`
- **STATUS_RESET** (WSTRING): Statusinformationen von `RESET_CLIENT`
- **STATUS_STATE** (WSTRING): Statusinformationen von `STATE_SUBSCRIBE`

Adapter

Adapter	Typ	Richtung	Beschreibung
S_R	adapter::types::bidirectional::ASR_AX	Socket – Set/Reset (Ausgang), Zustand (Eingang)	Set/Reset nach außen, Zustand zurück

Funktionsweise

1. Über das `INIT`-Ereignis wird zuerst `STATE_SUBSCRIBE` initialisiert, danach nacheinander `SET_CLIENT` und `RESET_CLIENT`. Nach allen Bestätigungen wird `INITO` nach außen gemeldet.
2. `S_R.SET` bzw. `S_R.RESET` lösen direkt die Methodenaufrufe `SET_CLIENT.REQ` bzw. `RESET_CLIENT.REQ` aus.
3. Sobald `STATE_SUBSCRIBE` per `IND` einen neuen Zustandswert liefert, taktet dies das interne **E_D_FF**-Flipflop, das den Wert übernimmt und stabil hält.
4. Der Ausgang `EO` des Flipflops feuert `S_R.EI1` mit dem gepufferten Wert an `S_R.DI1` – der Zustand wird über denselben Adapter zurückgemeldet, über den auch Set/Reset ausgelöst werden.
5. `AND_QO` (AND_BOOL_3) verundet die `QO`-Ausgänge aller drei internen Bausteine; jede Bestätigung löst zusätzlich `CNF` nach außen aus.

Technische Besonderheiten

- **Drei Netzverbindungen hinter einem Adapter**: Zwei `CLIENT_0`-Instanzen (Set, Reset) und ein `SUBSCRIBE_1` (Zustand) werden zu einer einzigen bidirektionalen ASR_AX-Schnittstelle zusammengefasst.
- **Pufferung mit D-Flipflop**: Der über `SUBSCRIBE_1` empfangene Zustand wird über ein internes `iec61499::events::E_D_FF` stabilisiert, bevor er über den Adapter zurückgemeldet wird.
- **Sequentielle Initialisierung**: `STATE_SUBSCRIBE` ⟶ `SET_CLIENT` ⟶ `RESET_CLIENT` (Composite-FBTypes verketten `INIT`/`INITO` seriell durch jede Instanz).
- **Kapselung**: Nach außen ist nur noch die ASR_AX-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Alle drei Verbindungen (Set, Reset, Zustandsabo) sind aufgebaut.
3. **Sendeaktiv**: Ein am Adapter eintreffendes Set- oder Reset-Ereignis löst den entsprechenden Remote-Methodenaufruf aus.
4. **Zustandsaktualisierung**: Ein neuer Wert vom Abonnement wird gepuffert und über denselben Adapter zurückgemeldet.

Anwendungsszenarien

- Fernauslösen von Set/Reset auf einem entfernten Aktor und gleichzeitiges Rücklesen des resultierenden Zustands über eine einzige Adapterverbindung
- Reduktion von händischer CLIENT_0/SUBSCRIBE_1-Verdrahtung auf einen einzigen wiederverwendbaren Baustein

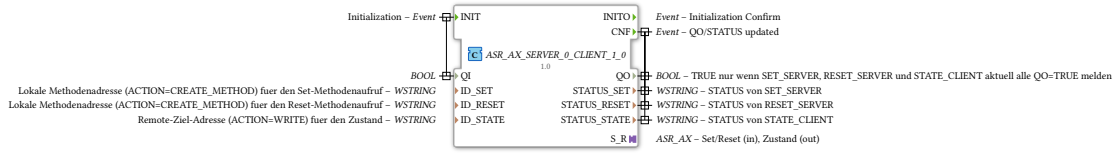
Vergleich mit ähnlichen Bausteinen

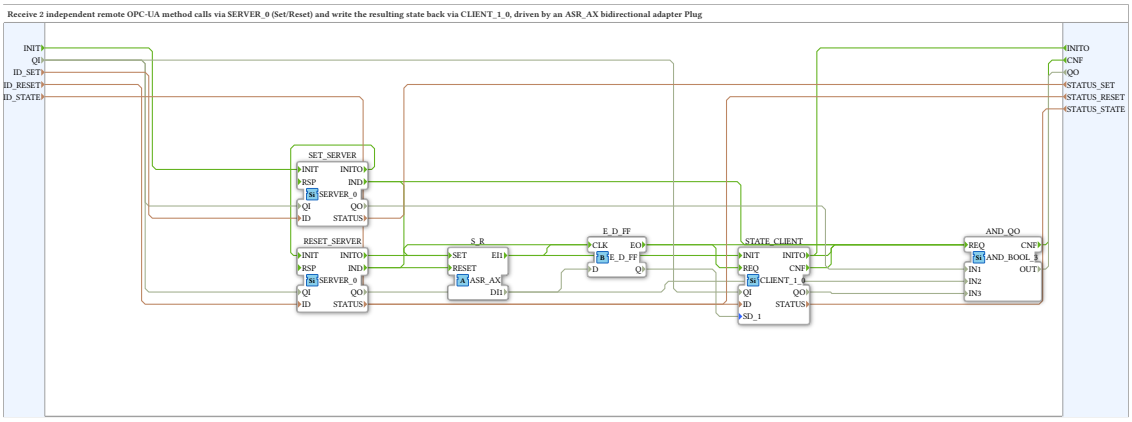
- **ASR_CLIENT_0**: Dieselben zwei Trigger, aber ohne Zustands-Rücklesung.
- **AE_AX_CLIENT_0_SUBSCRIBE_1**: Dasselbe Muster mit nur einem Trigger statt zwei.
- **ASRT_AX_CLIENT_0_SUBSCRIBE_1**: Erweitert dasselbe Muster um einen dritten Trigger (Toggle).
- **ASR_AX_SERVER_0_CLIENT_1_0**: Die serverseitige Gegenstelle.

Fazit

ASR_AX_CLIENT_0_SUBSCRIBE_1 bündelt zwei Remote-Trigger (Set/Reset) und ein Zustands-Abonnement hinter einem einzigen bidirektionalen ASR_AX-Adapter und macht damit händisch verdrahtete Kombinationen zu einem einzigen wiederverwendbaren Baustein.

ASR_AX_SERVER_0_CLIENT_1_0





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ASR_AX_SERVER_0_CLIENT_1_0** ist ein Composite-Funktionsblock, der zwei unabhängige entfernte OPC-UA-**Methodenaufrufe** (Set und Reset) über je einen eigenen `SERVER_0`-Baustein empfängt und den resultierenden Zustand über `CLIENT_1_0` an einen entfernten Knoten zurückschreibt – alles hinter einem einzigen bidirektionalen **ASR_AX-Adapter-Plug**. Er ist die serverseitige Gegenstelle zu ASR_AX_CLIENT_0_SUBSCRIBE_1.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO`, `STATUS_SET`, `STATUS_RESET` und `STATUS_STATE`
- **CNF** (Event): `QO`/`STATUS` aktualisiert, verbunden mit denselben Variablen

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang für alle drei internen Verbindungen
- **ID_SET** (WSTRING): Lokale Methodenadresse (ACTION=CREATE_METHOD) für den Set-Methodenaufruf
- **ID_RESET** (WSTRING): Lokale Methodenadresse (ACTION=CREATE_METHOD) für den Reset-Methodenaufruf
- **ID_STATE** (WSTRING): Remote-Ziel-Adresse (ACTION=WRITE) für den Zustand

Daten-Ausgänge

- **QO** (BOOL): TRUE nur wenn `SET_SERVER`, `RESET_SERVER` und `STATE_CLIENT` aktuell alle `QO = TRUE` melden
- **STATUS_SET** (WSTRING): Statusinformationen von `SET_SERVER`
- **STATUS_RESET** (WSTRING): Statusinformationen von `RESET_SERVER`
- **STATUS_STATE** (WSTRING): Statusinformationen von `STATE_CLIENT`

Adapter

Adapter	Typ	Richtung	Beschreibung
S_R	adapter::types::bidirectional::ASR_AX	Plug – Set/Reset (Eingang), Zustand (Ausgang)	Empfangenes Set/Reset, Zustand nach außen

Funktionsweise

1. Über das `INIT`-Ereignis werden nacheinander `SET_SERVER`, `RESET_SERVER` und `STATE_CLIENT` initialisiert. Nach allen drei Bestätigungen wird `INITO` nach außen gemeldet.

2. Ein entfernter Set-Aufruf auf `SET_SERVER` löst `S_R.SET` aus, ein entfernter Reset-Aufruf auf `RESET_SERVER` löst `S_R.RESET` aus – beide werden also über den Adapter nach innen gemeldet.
3. Sobald die weiterverarbeitende Logik über den Adapter ein Rückmeldeereignis `S_R.EI1` mit Datum `S_R.DI1` liefert, taktet dies das interne **E_D_FF**-Flipflop, das den Wert übernimmt und stabil hält.
4. Der Ausgang `EO` des Flipflops löst `STATE_CLIENT.REQ` aus; der gepufferte Wert wird per OPC-UA-Write an den unter `ID_STATE` konfigurierten entfernten Knoten gesendet.
5. `AND_QO` (AND_BOOL_3) verundet die `QO`-Ausgänge aller drei internen Bausteine; jede Bestätigung löst zusätzlich `CNF` nach außen aus.

Technische Besonderheiten

- **Drei Netzverbindungen hinter einem Adapter**: Zwei `SERVER_0`-Instanzen (Set, Reset) und ein `CLIENT_1_0` (Zustand) werden zu einer einzigen bidirektionalen ASR_AX-Schnittstelle zusammengefasst.
- **Pufferung mit D-Flipflop**: Der über den Adapter gemeldete Rückmeldewert wird über ein internes `iec61499::events::E_D_FF` stabilisiert, bevor er per `CLIENT_1_0` verschickt wird.
- **Sequentielle Initialisierung**: `SET_SERVER` → `RESET_SERVER` → `STATE_CLIENT` (Composite-FBTypes verketten `INIT`/`INITO` seriell durch jede Instanz).
- **Kapselung**: Nach außen ist nur noch die ASR_AX-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Alle drei Verbindungen (Set-Server, Reset-Server, Zustands-Client) sind aufgebaut.
3. **Empfangsaktiv**: Ein entfernter Set- oder Reset-Aufruf wird über den Adapter an die interne Logik gemeldet.
4. **Zustandsschreibvorgang**: Ein Rückmeldewert der internen Logik wird gepuffert und per Remote-Write an den konfigurierten Zielknoten gesendet.

Anwendungsszenarien

- Entgegennahme entfernter Set/Reset-Aufrufe und gleichzeitiges Zurückschreiben des resultierenden Zustands an einen anderen entfernten Knoten über eine einzige Adapterverbindung
- Serverseitiges Gegenstück zu ASR_AX_CLIENT_0_SUBSCRIBE_1 in verteilten Steuerungsarchitekturen

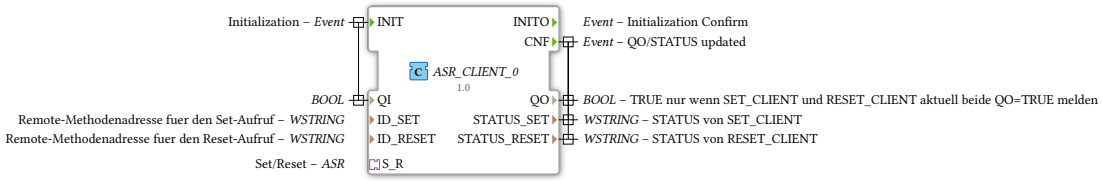
Vergleich mit ähnlichen Bausteinen

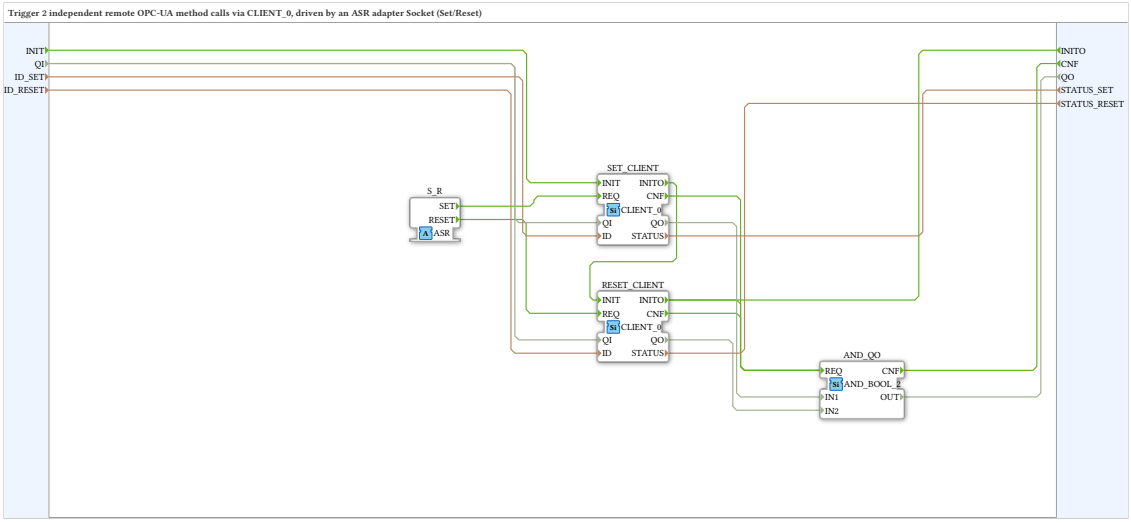
- **ASR_AX_CLIENT_0_SUBSCRIBE_1**: Die client-/aufrufende Gegenstelle.
- **AE_AX_SERVER_0_CLIENT_1_0**: Dasselbe Muster mit nur einem empfangenden Server statt zwei.
- **ASRT_AX_SERVER_0_CLIENT_1_0**: Erweitert dasselbe Muster um einen dritten empfangenden Server (Toggle).

Fazit

ASR_AX_SERVER_0_CLIENT_1_0 bündelt zwei empfangende Server (Set/Reset) und einen zustandsschreibenden Client hinter einem einzigen bidirektionalen ASR_AX-Adapter und bildet damit die serverseitige Entsprechung zu ASR_AX_CLIENT_0_SUBSCRIBE_1.

ASR_CLIENT_0





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ASR_CLIENT_0** ist ein Composite-Funktionsblock, der zwei unabhängige entfernte OPC-UA-**Methodenaufrufe** (Set und Reset) über je einen eigenen `CLIENT_0`-Baustein auslöst, gesteuert über einen einzigen unidirektionalen **ASR-Adapter-Socket**. Er ist die um einen zweiten Trigger erweiterte Variante von AE_CLIENT_0.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO`, `STATUS_SET` und `STATUS_RESET`
- **CNF** (Event): Bestätigung, dass einer der Methodenaufrufe gesendet wurde, verbunden mit `QO`, `STATUS_SET` und `STATUS_RESET`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, öffnet (TRUE) bzw. schließt (FALSE) beide Verbindungen
- **ID_SET** (WSTRING): Remote-Methodenadresse für den Set-Aufruf
- **ID_RESET** (WSTRING): Remote-Methodenadresse für den Reset-Aufruf

Daten-Ausgänge

- **QO** (BOOL): TRUE nur wenn `SET_CLIENT` und `RESET_CLIENT` aktuell beide `QO = TRUE` melden
- **STATUS_SET** (WSTRING): Statusinformationen von `SET_CLIENT`
- **STATUS_RESET** (WSTRING): Statusinformationen von `RESET_CLIENT`

Adapter

Adapter	Typ	Richtung	Beschreibung
S_R	adapter::types::unidirectional::ASR	Socket (Eingang)	Set/Reset

Funktionsweise

1. Über das `INIT`-Ereignis werden die internen `SET_CLIENT`- und `RESET_CLIENT`-Blöcke nacheinander (`INIT` → `INITO` → `INIT`) mit `QI` und ihrer jeweiligen `ID` initialisiert. Nach beiden Bestätigungen wird `INITO` nach außen gemeldet.
2. Ein Ereignis am Adapter-Socket `S_R.SET` löst direkt den internen `SET_CLIENT`-Methodenaufruf aus; `S_R.RESET` löst entsprechend `RESET_CLIENT` aus.
3. Beide `CLIENT_0`-Instanzen bestätigen unabhängig voneinander mit `CNF`, das jeweils nach außen weitergegeben wird.

4. Der Baustein `AND_QO` verundet die `QO`-Ausgänge beider `CLIENT_0`-Instanzen zu einem gemeinsamen `QO`.

Technische Besonderheiten

- **Zwei parallele CLIENT_0-Instanzen**: Set und Reset werden über getrennte `CLIENT_0`-Bausteine (unterschiedliche `ID`, unabhängige Verbindungen) abgewickelt.
- **Sequentielle Initialisierung**: Die beiden internen Blöcke werden nacheinander initialisiert (`SET_CLIENT.INITO` ⟶ `RESET_CLIENT.INIT`), da Composite-FBTypes ihr `INIT`/`INITO` explizit durch jede Instanz seriell verketten müssen.
- **Kapselung**: Nach außen ist nur noch die ASR-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Beide Verbindungen sind aufgebaut, der Block ist bereit, Set- oder Reset-Aufrufe auszulösen.
3. **Sendeaktiv**: Ein am ASR-Socket eintreffendes Set- oder Reset-Ereignis löst den entsprechenden Remote-Methodenaufruf aus.

Anwendungsszenarien

- Fernauslösen zweier unabhängiger entfernter Aktionen (Set/Reset) über einen einzigen Adapter, z. B. zum Schalten eines entfernten Aktors
- Netzwerkkopplung in ASR-Adapter-basierten Anwendungen

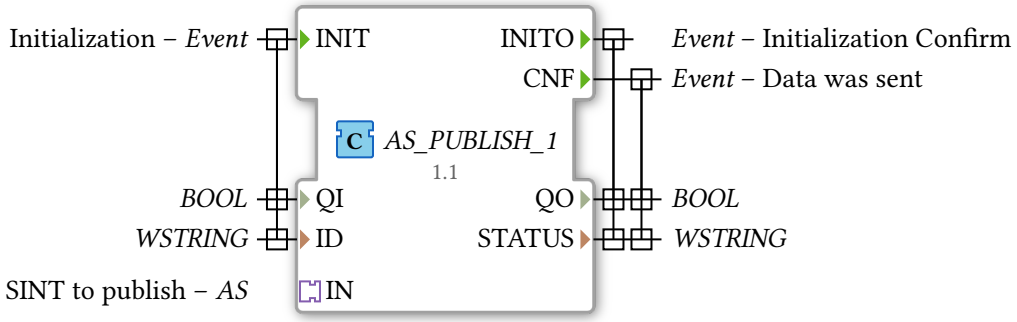
Vergleich mit ähnlichen Bausteinen

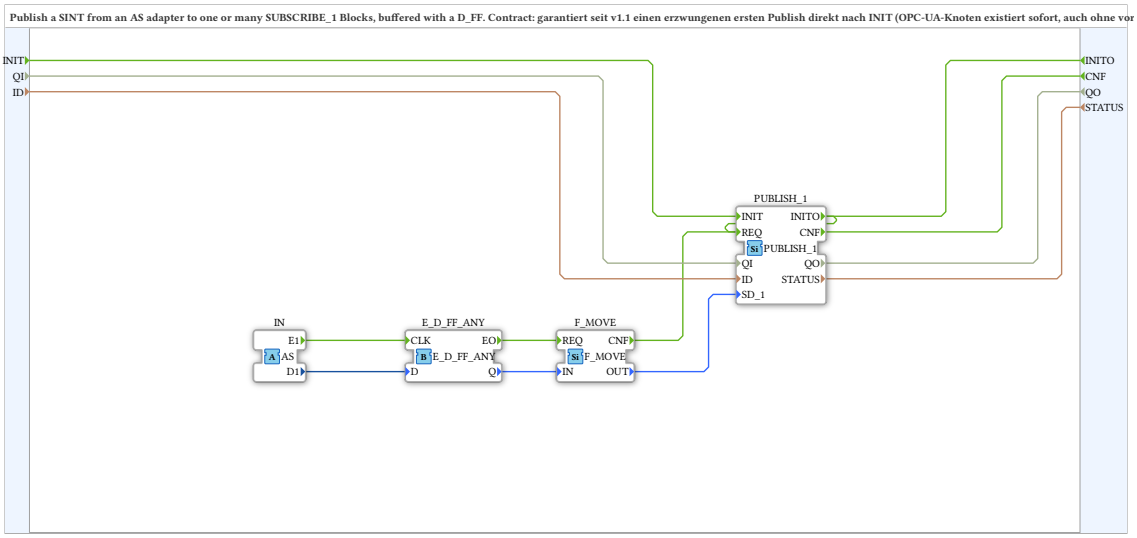
- **AE_CLIENT_0**: Dieselbe Grundstruktur mit nur einem Trigger statt zwei.
- **ASRT_CLIENT_0**: Erweitert dasselbe Muster um einen dritten Trigger (Toggle).
- **ASR_AX_CLIENT_0_SUBSCRIBE_1**: Kombiniert dieselben zwei `CLIENT_0`-Trigger zusätzlich mit einem `SUBSCRIBE_1`-Zustandskanal hinter einem bidirektionalen Adapter.

Fazit

ASR_CLIENT_0 bündelt zwei unabhängige Remote-Methodenaufrufe (Set/Reset) hinter einem einzigen ASR-Adapter-Socket und eignet sich damit für Anwendungen, in denen ein entfernter Aktor über zwei getrennte OPC-UA-Methoden geschaltet wird.

AS_PUBLISH_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AS_PUBLISH_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `PUBLISH_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AS-Adapter** abbildet. Ein am Adapter-Socket `IN` anliegender **SINT**-Wert wird über ein internes D-Flipflop (E_D_FF_ANY) gepuffert und anschließend über den konfigurierten Publish-Kanal an alle zugehörigen `SUBSCRIBE_1`-Empfänger gesendet.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`
- **CNF** (Event): Bestätigung, dass die Daten gesendet wurden, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Publish-Dienst
- **ID** (WSTRING): Identifikator des Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::AS	Socket (Eingang)	Zu veröffentlichender SINT-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `PUBLISH_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Sobald der AS-Socket `IN` ein Ereignis auf `IN.E1` liefert, wird der gleichzeitig an `IN.D1` anliegende SINT-Wert in das interne **E_D_FF_ANY**-Flipflop übernommen (`CLK` bzw. `D`).
3. Das Flipflop hält den Wert stabil an seinem Ausgang `Q` und erzeugt bei einer Wertänderung das Ereignis `EO`.
4. `EO` löst das Sendeereignis `REQ` des internen `PUBLISH_1`-Blocks aus; der gepufferte Wert `Q` wird über die Datenverbindung als `SD_1` übergeben und auf dem konfigurierten Kanal veröffentlicht.

5. Nach erfolgreichem Senden bestätigt der `PUBLISH_1`-Block mit `CNF`, das nach außen geführt wird.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der zu sendende SINT-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert. Dadurch wird verhindert, dass ein sich während der Verarbeitung ändernder Eingangswert den Sendevorgang stört.
- **Ereignisbasierte Entkopplung**: Durch das echte ereignisbasierte `E_D_FF_ANY` werden unmittelbare Rückkopplungsschleifen zwischen `PUBLISH_1` und `SUBSCRIBE_1` (z. B. auf demselben OPC-UA-Knoten) entkoppelt – der saubere Fix gegenüber dem Interim-Fix mit `AX_D_FF`.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `PUBLISH_1` (inkl. `REQ`/`SD_1`) wird nach innen verlegt; nach außen ist nur noch die AS-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Der Block ist bereit, Daten über den konfigurierten Kanal zu veröffentlichen.
3. **Sendeaktiv**: Ein am AS-Socket eintreffendes Ereignis puffert den Wert und veranlasst das Senden über den internen `PUBLISH_1`-Block.

Anwendungsszenarien

- **Netzwerkkopplung in AS-Anwendungen**: Senden von SINT-Signalen (z. B. Schaltbefehle oder Statusbits) aus der Adapter-basierten Logik über Publish/Subscribe an entfernte Empfänger.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_SUBSCRIBE_1` werden Rückkopplungen zwischen eigenen Veröffentlichungen und dem eigenen Abonnement entkoppelt.
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AS-Adapter setzen.

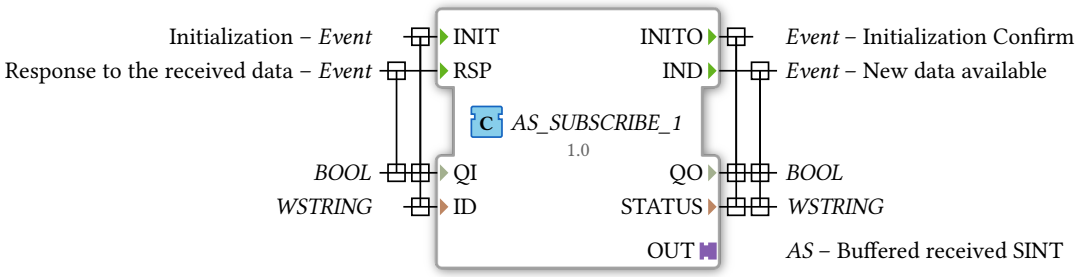
Vergleich mit ähnlichen Bausteinen

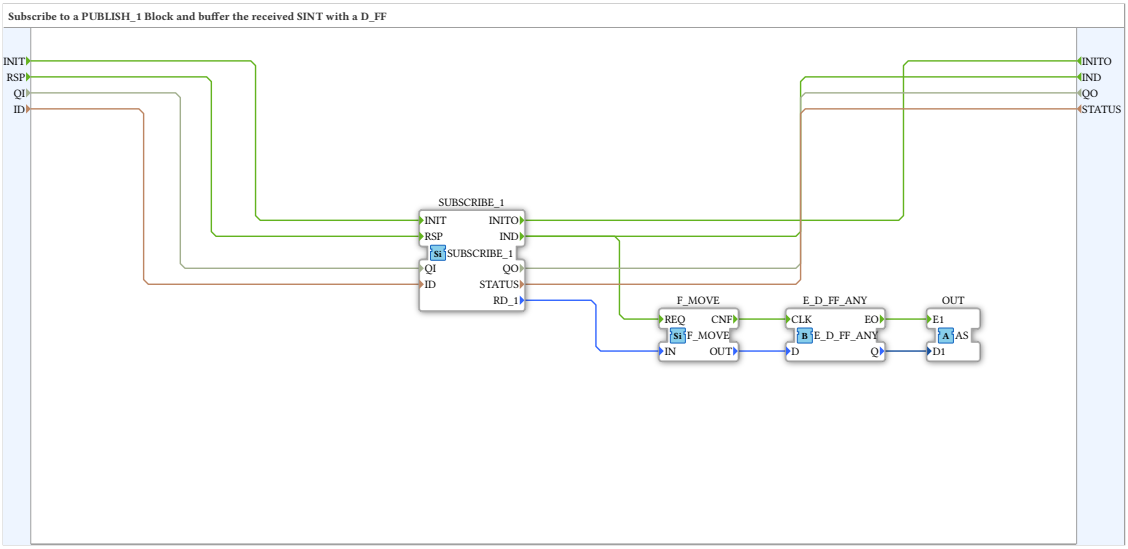
- **PUBLISH_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`REQ`, `SD_1`, …). **AS_PUBLISH_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den AS-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen SINT-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AS_PUBLISH_1** vereint Puffern und Veröffentlichen in einem Baustein.

Fazit

AS_PUBLISH_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `PUBLISH_1` mit der Adapter-basierten SINT-Verarbeitung. Die interne D-Flipflop-Pufferung stabilisiert den Sendewert und entkoppelt Rückkopplungsschleifen; die reduzierte AS-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

AS_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AS_SUBSCRIBE_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `SUBSCRIBE_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AS-Adapter** abbildet. Er abonniert einen über die `ID` konfigurierten Publish-Kanal und stellt den empfangenen **SINT**-Wert gepuffert über einen D-Flipflop (E_D_FF_ANY) am Adapter-Plug `OUT` bereit. Damit lassen sich Netzkommunikation und SINT-Signalverarbeitung in der AS-Adapter-Welt sauber verbinden.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`
- **RSP** (Event): Response-Ereignis, verbunden mit `QI`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Abonnement-Dienst
- **ID** (WSTRING): Identifikator des zu abonnierenden Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
OUT	adapter::types::unidirectional::AS	Plug (Ausgang)	Gepufferter empfangener SINT-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `SUBSCRIBE_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Der interne `SUBSCRIBE_1`-Block überwacht den konfigurierten Kanal. Sobald der zugehörige `PUBLISH_1`-Block Daten veröffentlicht, erzeugt er das Indikationsereignis `IND` und legt den empfangenen Wert an `RD_1` an.

3. IND taktet den internen **E_D_FF_ANY**-Flipflop (CLK), der den Wert von RD_1 an seinem Dateneingang D übernimmt.
4. Der gespeicherte Wert erscheint stabil am Ausgang Q und wird über die Datenverbindung auf OUT.D1 des AX-Plugs gelegt.
5. Das Ereignis EO des Flipflops wird auf den Ereignis-Ausgang OUT.E1 gelegt, sodass nachgeschaltete Bausteine über die neue Information benachrichtigt werden.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der empfangene SINT-Wert wird über ein internes iec61499::events::E_D_FF_ANY gepuffert und bleibt stabil am Ausgang, bis der nächste Wert empfangen wird. EO feuert nur bei einer Änderung von Q.
- **RSP ist ein No-Op bei Subscribern**: Laut Quellcode-Analyse von 4diac-forte (CCommFB::sendData()) führt das RSP-Ereignis bei SUBSCRIBE_1 (Service-Typ Subscriber) keine Aktion aus – der Zweig != e_Subscriber wird übersprungen. Das Event-Input-Paar INIT/RSP wird nur für die Quittierung durchgeführt und ist für die Quittierung im Subscriber-Fall funktionslos. Es wird dennoch nach außen geführt, damit ein konfigurierbarer Anschluss besteht.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von SUBSCRIBE_1 (inkl. IND/RD_1) wird nach innen verlegt; nach außen ist nur noch die AS-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das INIT-Ereignis.
2. **Initialisiert**: Das Abonnement ist aktiv; der Block wartet auf Daten vom Publisher.
3. **Datenempfang**: Bei IND wird der empfangene SINT-Wert in das interne D-Flipflop übernommen und über OUT ausgegeben.

Anwendungsszenarien

- **Netzwerkkopplung in AS-Anwendungen**: Empfang von SINT-Signalen (z. B. Schaltbefehle oder Statusbits) über Publish/Subscribe und Weiterleitung über AS-Adapter an die Adapter-basierte Logik.
- **Event-Storm-Vermeidung**: Gemeinsam mit AX_PUBLISH_1 und einem echten ereignisbasierten E_D_FF_ANY lässt sich die Rückkopplung zwischen SUBSCRIBE_1 und PUBLISH_1 entkoppeln (sauberer Fix gegenüber dem Interim-Fix mit AX_D_FF).
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AS-Adapter setzen.

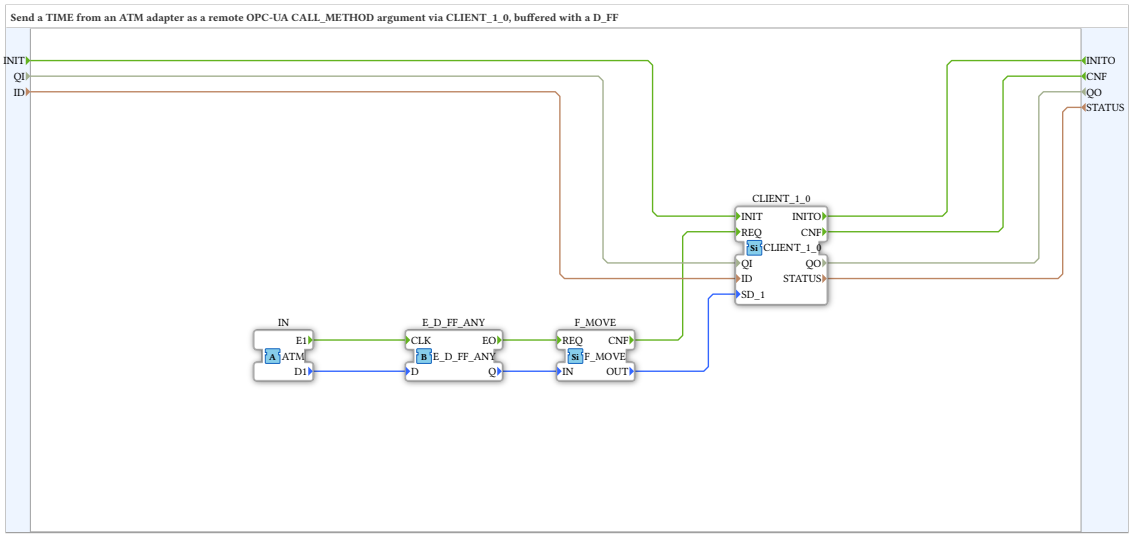
Vergleich mit ähnlichen Bausteinen

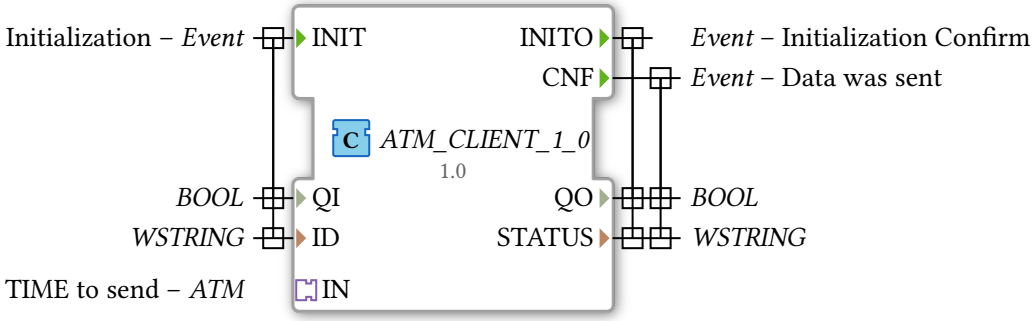
- **SUBSCRIBE_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (IND, RD_1, …). **AS_SUBSCRIBE_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (E_D_FF_ANY) und stellt nur noch den AS-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen SINT-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AS_SUBSCRIBE_1** vereint Subscribing und Pufferung in einem Baustein.

Fazit

AS_SUBSCRIBE_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins SUBSCRIBE_1 mit der Adapter-basierten SINT-Verarbeitung. Die interne D-Flipflop-Pufferung hält den empfangenen Wert stabil, bis neue Daten eintreffen, und die reduzierte AS-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

ATM_CLIENT_1_0





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ATM_CLIENT_1_0** ist ein Composite-Funktionsblock, der den netzwerkbasierten `CLIENT_1_0`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **ATM-Adapter** (Datentyp `TIME`) abbildet. Ein am Adapter-Socket `IN` anliegender **TIME**-Wert wird über ein internes `E_D_FF_ANY`-Flipflop gepuffert und anschließend über `CLIENT_1_0` als OPC-UA-**Write** oder Methodenaufruf (`CALL_METHOD`) an den unter `ID` konfigurierten entfernten Knoten gesendet.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`
- **CNF** (Event): Bestätigung, dass die Daten gesendet wurden, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, öffnet (TRUE) bzw. schließt (FALSE) die Verbindung zum Server
- **ID** (WSTRING): Identifikator der Verbindung (OPC-UA-Adresse des Zielknotens, z. B. `opc_ua[WRITE;opc.tcp://192.168.1.12:4840#;...]`)

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Verbindungsstatus
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::ATM	Socket (Eingang)	Zu sendender TIME-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `CLIENT_1_0`-Block mit `QI` und `ID` initialisiert; er baut die Verbindung zum entfernten Server auf und quittiert mit `INITO`.
2. Sobald der ATM-Socket `IN` ein Ereignis auf `IN.E1` liefert, wird der an `IN.D1` anliegende TIME-Wert in das interne `E_D_FF_ANY`-Flipflop übernommen.
3. Das Flipflop hält den Wert stabil und erzeugt das Ereignis `EO`.
4. `EO` stößt über ein `F_MOVE` (Datentyp `TIME`) die Zuweisung an `CLIENT_1_0.SD_1` an und löst das Sendeereignis `REQ` des `CLIENT_1_0`-Blocks aus.
5. Nach erfolgreichem Senden bestätigt der `CLIENT_1_0`-Block mit `CNF`.

Technische Besonderheiten

- **Pufferung mit E_D_FF_ANY**: Der zu sendende TIME-Wert wird über ein internes `E_D_FF_ANY` gepuffert, um Störungen während des Sendevorgangs zu vermeiden.
- **Remote-Write / Call-Method**: `CLIENT_1_0` adressiert einen entfernten OPC-UA-Server direkt, um Zeitwerte (z. B. Timer-Sollwerte, Verzögerungszeiten) zu übertragen.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `CLIENT_1_0` wird nach innen verlegt.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Die Verbindung zum entfernten Server steht, der Block ist bereit zu senden.
3. **Sendeaktiv**: Ein am ATM-Socket eintreffendes Ereignis puffert den TIME-Wert und führt den Remote-Write aus.

Anwendungsszenarien

- **Fernübertragung von Zeitparametern**: Übertragung von Zeitdauer-Werten (z. B. Ventilschaltzeiten, Spülzeiten) an ein entferntes Steuerungsmodul via OPC UA.
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilten Zeitwerten in Bibliotheken, die auf ATM-Adapter setzen.

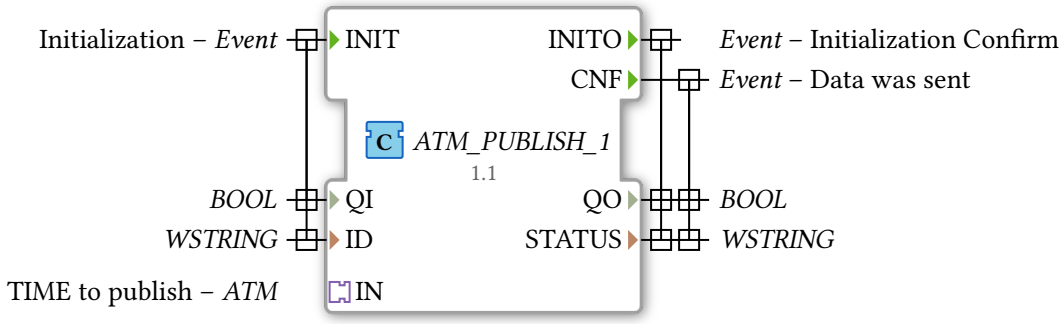
Vergleich mit ähnlichen Bausteinen

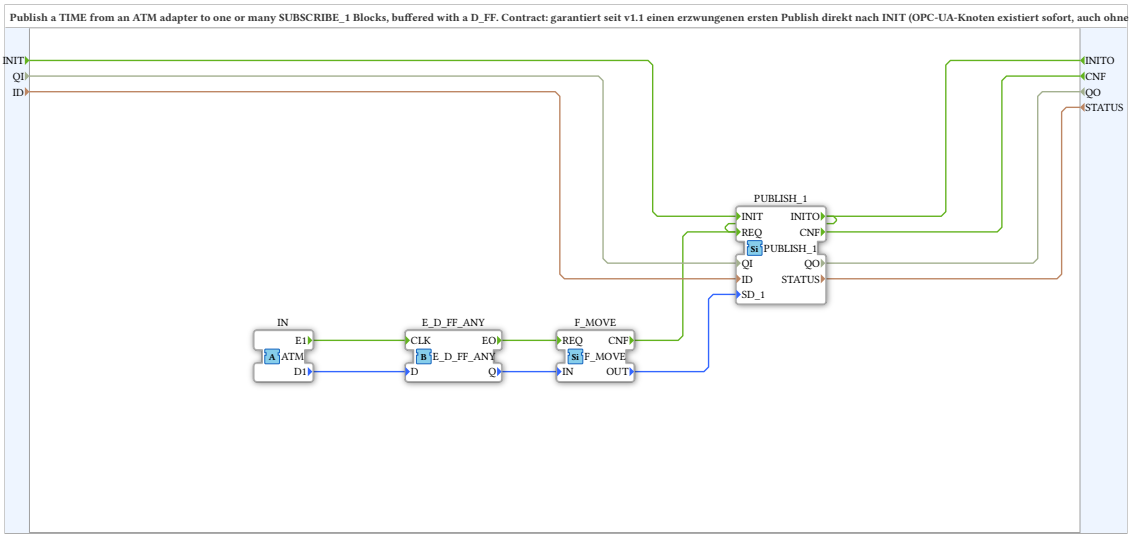
- **AX_CLIENT_1_0**: Verarbeitet BOOL-Werte über einen AX-Adapter.
- **AR_CLIENT_1_0**: Verarbeitet REAL-Werte über einen AR-Adapter.
- **ATM_PUBLISH_1**: Kapselt `PUBLISH_1` für lokale Publish/Subscribe-Verbindungen.

Fazit

ATM_CLIENT_1_0 ermöglicht das direkte Schreiben von TIME-Werten auf einen entfernten OPC-UA-Server über eine saubere ATM-Adapter-Schnittstelle.

ATM_PUBLISH_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ATM_PUBLISH_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `PUBLISH_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **ATM-Adapter** abbildet. Ein am Adapter-Socket `IN` anliegender **TIME**-Wert wird über ein internes D-Flipflop (E_D_FF_ANY) gepuffert und anschließend über den konfigurierten Publish-Kanal an alle zugehörigen `SUBSCRIBE_1`-Empfänger gesendet.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`
- **CNF** (Event): Bestätigung, dass die Daten gesendet wurden, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Publish-Dienst
- **ID** (WSTRING): Identifikator des Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::ATM	Socket (Eingang)	Zu veröffentlichender TIME-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `PUBLISH_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Sobald der ATM-Socket `IN` ein Ereignis auf `IN.E1` liefert, wird der gleichzeitig an `IN.D1` anliegende TIME-Wert in das interne **E_D_FF_ANY**-Flipflop übernommen (`CLK` bzw. `D`).
3. Das Flipflop hält den Wert stabil an seinem Ausgang `Q` und erzeugt bei einer Wertänderung das Ereignis `EO`.

4. `EO` löst das Sendeereignis `REQ` des internen `PUBLISH_1`-Blocks aus; der gepufferte Wert `Q` wird über die Datenverbindung als `SD_1` übergeben und auf dem konfigurierten Kanal veröffentlicht.
5. Nach erfolgreichem Senden bestätigt der `PUBLISH_1`-Block mit `CNF`, das nach außen geführt wird.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der zu sendende TIME-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert. Dadurch wird verhindert, dass ein sich während der Verarbeitung ändernder Eingangswert den Sendevorgang stört.
- **Ereignisbasierte Entkopplung**: Durch das echte ereignisbasierte `E_D_FF_ANY` werden unmittelbare Rückkopplungsschleifen zwischen `PUBLISH_1` und `SUBSCRIBE_1` (z. B. auf demselben OPC-UA-Knoten) entkoppelt – der saubere Fix gegenüber dem Interim-Fix mit `AX_D_FF`.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `PUBLISH_1` (inkl. `REQ`/`SD_1`) wird nach innen verlegt; nach außen ist nur noch die ATM-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Der Block ist bereit, Daten über den konfigurierten Kanal zu veröffentlichen.
3. **Sendeaktiv**: Ein am ATM-Socket eintreffendes Ereignis puffert den Wert und veranlasst das Senden über den internen `PUBLISH_1`-Block.

Anwendungsszenarien

- **Netzwerkkopplung in ATM-Anwendungen**: Senden von TIME-Signalen (z. B. Schaltbefehle oder Statusbits) aus der Adapter-basierten Logik über Publish/Subscribe an entfernte Empfänger.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_SUBSCRIBE_1` werden Rückkopplungen zwischen eigenen Veröffentlichungen und dem eigenen Abonnement entkoppelt.
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf ATM-Adapter setzen.

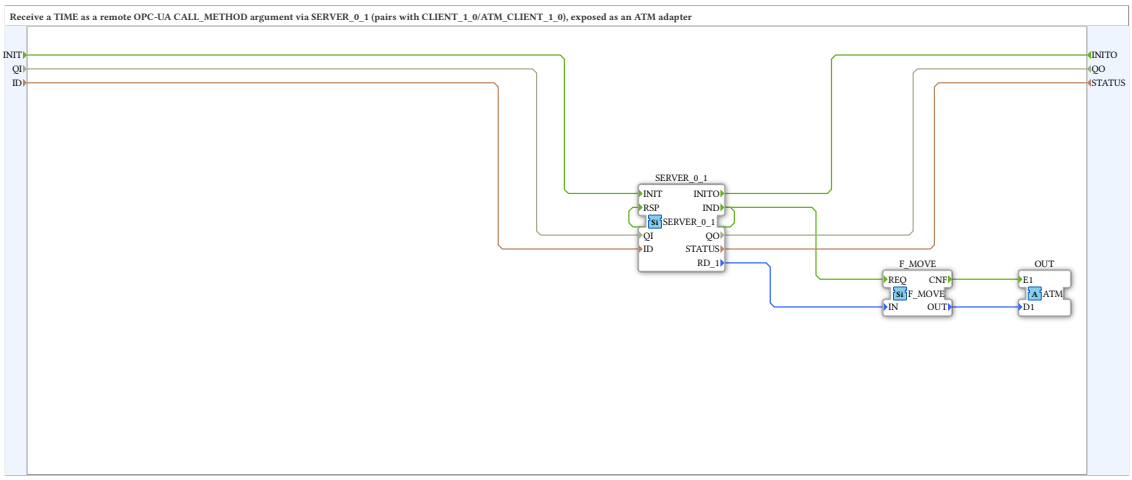
Vergleich mit ähnlichen Bausteinen

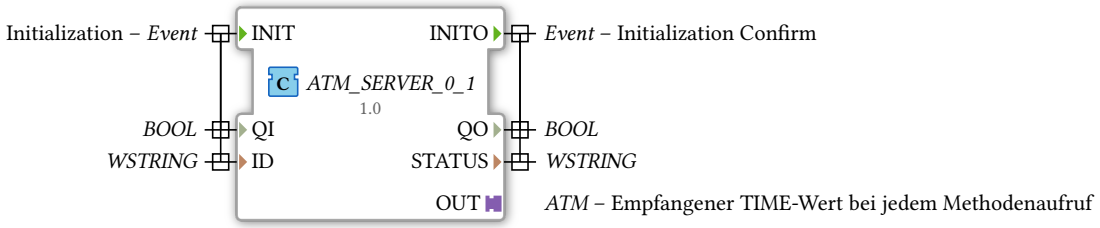
- **PUBLISH_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`REQ`, `SD_1`, ...). **ATM_PUBLISH_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den ATM-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen TIME-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **ATM_PUBLISH_1** vereint Puffern und Veröffentlichen in einem Baustein.

Fazit

ATM_PUBLISH_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `PUBLISH_1` mit der Adapter-basierten TIME-Verarbeitung. Die interne D-Flipflop-Pufferung stabilisiert den Sendewert und entkoppelt Rückkopplungsschleifen; die reduzierte ATM-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

ATM_SERVER_0_1





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ATM_SERVER_0_1** ist ein Composite-Funktionsblock aus dem Paket `adapter::net`. Er kapselt den netzwerkbasierten Server-Baustein `iec61499::net::SERVER_0_1` und stellt empfangene OPC-UA-Methodenaufrufe (`CALL_METHOD`) für **TIME**-Argumente als unidirektionalen **ATM-Adapter** (`OUT`) bereit. Er bildet das Empfangs-Gegenstück zu **ATM_CLIENT_1_0** (bzw. `CLIENT_1_0`).

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID` (leitet an `SERVER_0_1.INIT` weiter)

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung (`SERVER_0_1.INITO`), verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, schaltet den Server aktiv (TRUE) bzw. inaktiv (FALSE)
- **ID** (WSTRING): Server-Identifikator (OPC-UA-Serveradresse und Methodenspezifikation)

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Server-Betriebsstatus
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
OUT	adapter::types::unidirectional::ATM	Plug (Ausgang)	Empfangener TIME-Wert bei jedem Methodenaufruf

Funktionsweise

1. Ein `INIT`-Ereignis an `ATM_SERVER_0_1` wird an den internen `SERVER_0_1.INIT`-Eingang weitergeleitet. Der Server startet den OPC-UA-Endpunkt und quittiert bei Erfolg mit `INITO`.
2. Bei einem eintreffenden Methodenaufruf von einem entfernten Client (`CLIENT_1_0`/`ATM_CLIENT_1_0`) erzeugt `SERVER_0_1` das Indikations-Ereignis `IND` und stellt den empfangenen TIME-Wert an `RD_1` bereit.
3. Zur Vermeidung von Antwort-Latenz-Sperren (RSP-Latenz-Bug) ist `SERVER_0_1.IND` im internen Netzwerk direkt auf `SERVER_0_1.RSP` zurückverdrahtet. Der Server quittiert die Methode dadurch sofort ohne Verzögerung.
4. Gleichzeitig leitet `IND` den empfangenen Wert über `F_MOVE` (Datentyp `TIME`) an den ATM-Plug `OUT.D1` weiter und löst das Ausgangs-Ereignis `OUT.E1` aus.

Technische Besonderheiten

- **Direkte RSP-Rückverdrahtung**: Um 4-Sekunden-Server-Sperren durch verzögerte Antworten zu vermeiden, wird `SERVER_0_1.IND` intern unmittelbar auf `SERVER_0_1.RSP` zurückgekoppelt.
- **Nahtlose Adapter-Integration**: Der empfangene Rohwert `RD_1` von `SERVER_0_1` wird über ein `F_MOVE` (TIME) direkt auf die ATM-Adapter-Struktur (`OUT.D1` / `OUT.E1`) abgebildet.
- **Empfangs-Gegenstück zu `ATM_CLIENT_1_0`**: In IEC 61499 pairt ein `CLIENT_1_0`-Baustein („Connect to a SERVER_0_1 Block“) mit einem `SERVER_0_1`-Baustein.

Zustandsübersicht

1. **Nicht initialisiert**: Der Server ist inaktiv und wartet auf das `INIT`-Ereignis.
2. **Bereit (Listening)**: Der OPC-UA-Server ist aktiv und wartet auf Methodenaufrufe von entfernten Clients.
3. **Empfang (Indikation)**: Ein Methodenaufruf trifft ein; der TIME-Wert wird an den `OUT`-Adapter übergeben und die Quittierung sofort gesendet.

Anwendungsszenarien

- **Empfang von Zeitparametern**: Empfangen von Schaltzeiten oder Zeitdauern (z. B. Spülzeit, Verzögerungszeit) von einem entfernten Steuerungsmodul via OPC UA `CALL_METHOD`.
- **Modulare Server-Architekturen**: Bereitstellung von OPC-UA-Methoden-Endpunkten für Zeitparameter in adapterbasierten Anwendungen.

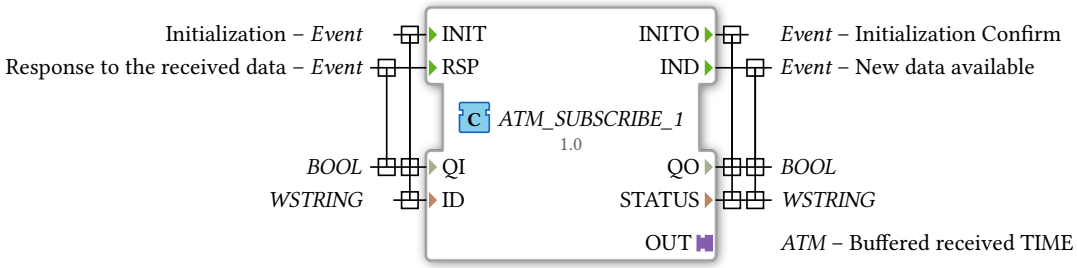
Vergleich mit ähnlichen Bausteinen

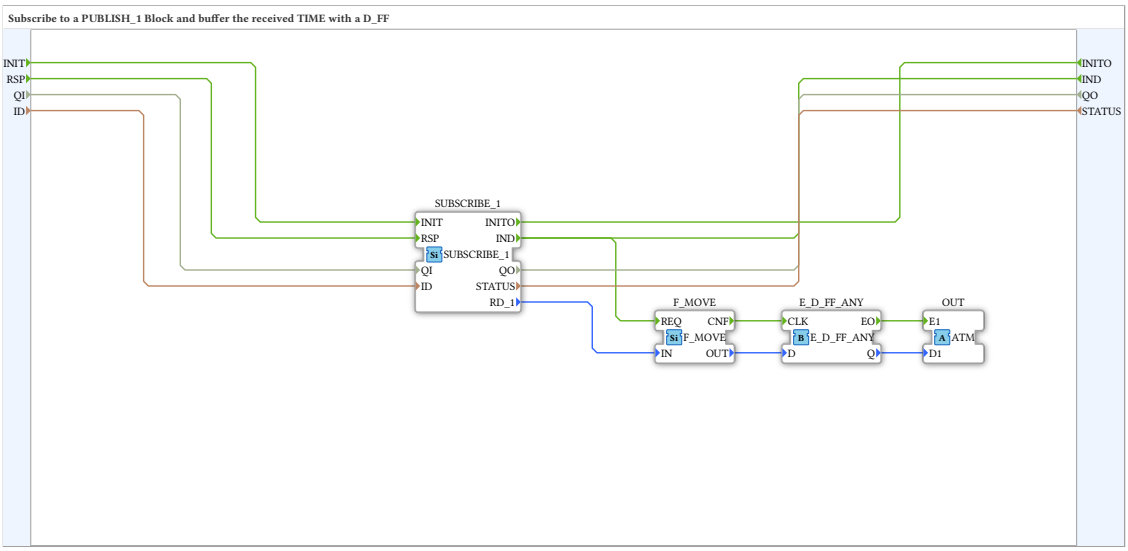
- **AR_SERVER_0_1**: Empfängt REAL-Werte über einen AR-Adapter.
- **AX_SERVER_0_1**: Empfängt BOOL-Werte über einen AX-Adapter.
- **ATM_SUBSCRIBE_1**: Kapselt `SUBSCRIBE_1` für lokales Publish/Subscribe statt verbindungsorientierter Methodenaufrufe.

Fazit

ATM_SERVER_0_1 stellt empfangene OPC-UA-Methodenaufrufe für TIME-Werte als sauberen ATM-Adapter bereit und verhindert durch direkte RSP-Rückkopplung Server-Lockups.

ATM_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ATM_SUBSCRIBE_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `SUBSCRIBE_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **ATM-Adapter** abbildet. Er abonniert einen über die `ID` konfigurierten Publish-

Kanal und stellt den empfangenen **TIME**-Wert gepuffert über einen D-Flipflop (E_D_FF_ANY) am Adapter-Plug `OUT` bereit. Damit lassen sich Netzkommunikation und TIME-Signalverarbeitung in der ATM-Adapter-Welt sauber verbinden.

Schnittstellenstruktur

Ereignis-Eingänge
- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`
- **RSP** (Event): Response-Ereignis, verbunden mit `QI`

Ereignis-Ausgänge
- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`

Daten-Eingänge
- **QI** (BOOL): Qualifier-Eingang, aktiviert den Abonnement-Dienst
- **ID** (WSTRING): Identifikator des zu abonnierenden Publish-Kanals

Daten-Ausgänge
- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
OUT	adapter::types::unidirectional::ATM	Plug (Ausgang)	Gepufferter empfangener TIME-Wert

Funktionsweise
1. Über das `INIT`-Ereignis wird der interne `SUBSCRIBE_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Der interne `SUBSCRIBE_1`-Block überwacht den konfigurierten Kanal. Sobald der zugehörige `PUBLISH_1`-Block Daten veröffentlicht, erzeugt er das Indikationsereignis `IND` und legt den empfangenen Wert an `RD_1` an.
3. `IND` taktet den internen **E_D_FF_ANY**-Flipflop (`CLK`), der den Wert von `RD_1` an seinem Dateneingang `D` übernimmt.
4. Der gespeicherte Wert erscheint stabil am Ausgang `Q` und wird über die Datenverbindung auf `OUT.D1` des AX-Plugs gelegt.
5. Das Ereignis `EO` des Flipflops wird auf den Ereignis-Ausgang `OUT.E1` gelegt, sodass nachgeschaltete Bausteine über die neue Information benachrichtigt werden.

Technische Besonderheiten
- **Pufferung mit D-Flipflop**: Der empfangene TIME-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert und bleibt stabil am Ausgang, bis der nächste Wert empfangen wird. `EO` feuert nur bei einer Änderung von `Q`.
- **RSP ist ein No-Op bei Subscribern**: Laut Quellcode-Analyse von `4diac-forte` (`CCommFB::sendData()`) führt das `RSP`-Ereignis bei `SUBSCRIBE_1` (Service-Typ Subscriber) keine Aktion aus – der Zweig `!= e_Subscriber` wird übersprungen. Das Event-Input-Paar `INIT`/`RSP` wird nur für die Quittierung durchgeführt und ist für die Quittierung im Subscriber-Fall funktionslos. Es wird dennoch nach außen geführt, damit ein konfigurierbarer Anschluss besteht.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `SUBSCRIBE_1` (inkl. `IND`/`RD_1`) wird nach innen verlegt; nach außen ist nur noch die ATM-Adapter-Schnittstelle sichtbar.

Zustandsübersicht
1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Das Abonnement ist aktiv; der Block wartet auf Daten vom Publisher.
3. **Datenempfang**: Bei `IND` wird der empfangene TIME-Wert in das interne D-Flipflop übernommen und über `OUT` ausgegeben.

Anwendungsszenarien

- **Netzwerkkopplung in ATM-Anwendungen**: Empfang von TIME-Signalen (z. B. Schaltbefehle oder Statusbits) über Publish/Subscribe und Weiterleitung über ATM-Adapter an die Adapter-basierte Logik.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_PUBLISH_1` und einem echten ereignisbasierten `E_D_FF_ANY` lässt sich die Rückkopplung zwischen `SUBSCRIBE_1` und `PUBLISH_1` entkoppeln (sauberer Fix gegenüber dem Interim-Fix mit `AX_D_FF`).
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf ATM-Adapter setzen.

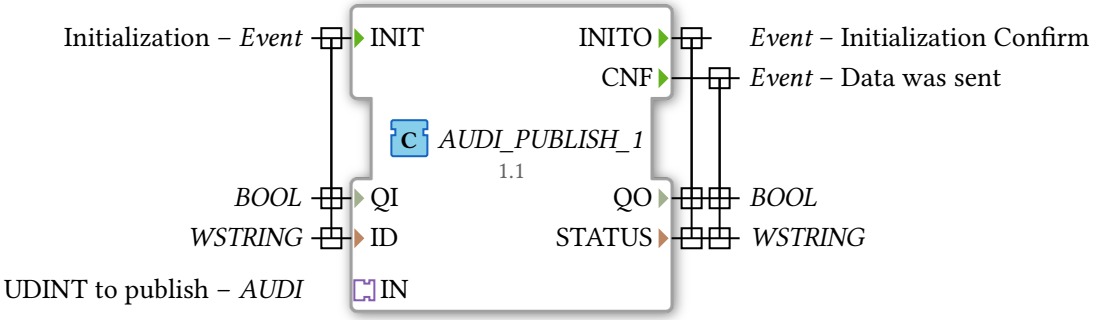
Vergleich mit ähnlichen Bausteinen

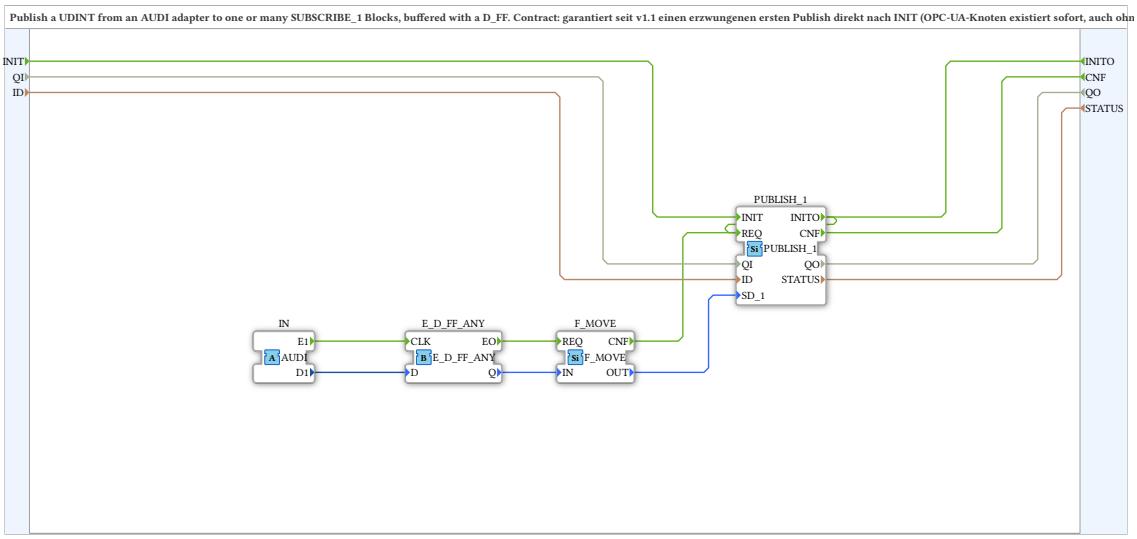
- **SUBSCRIBE_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`IND`, `RD_1`, ...). **ATM_SUBSCRIBE_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den ATM-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen TIME-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **ATM_SUBSCRIBE_1** vereint Subscribing und Pufferung in einem Baustein.

Fazit

ATM_SUBSCRIBE_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `SUBSCRIBE_1` mit der Adapter-basierten TIME-Verarbeitung. Die interne D-Flipflop-Pufferung hält den empfangenen Wert stabil, bis neue Daten eintreffen, und die reduzierte ATM-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

AUDI_PUBLISH_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUDI_PUBLISH_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `PUBLISH_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf

einen unidirektionalen **AUDI-Adapter** abbildet. Ein am Adapter-Socket `IN` anliegender **UDINT**-Wert wird über ein internes D-Flipflop (E_D_FF_ANY) gepuffert und anschließend über den konfigurierten Publish-Kanal an alle zugehörigen `SUBSCRIBE_1`-Empfänger gesendet.

Schnittstellenstruktur

Ereignis-Eingänge
- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`

Ereignis-Ausgänge
- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`
- **CNF** (Event): Bestätigung, dass die Daten gesendet wurden, verbunden mit `QO` und `STATUS`

Daten-Eingänge
- **QI** (BOOL): Qualifier-Eingang, aktiviert den Publish-Dienst
- **ID** (WSTRING): Identifikator des Publish-Kanals

Daten-Ausgänge
- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::AUDI	Socket (Eingang)	Zu veröffentlichender UDINT-Wert

Funktionsweise
1. Über das `INIT`-Ereignis wird der interne `PUBLISH_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Sobald der AUDI-Socket `IN` ein Ereignis auf `IN.E1` liefert, wird der gleichzeitig an `IN.D1` anliegende UDINT-Wert in das interne **E_D_FF_ANY**-Flipflop übernommen (`CLK` bzw. `D`).
3. Das Flipflop hält den Wert stabil an seinem Ausgang `Q` und erzeugt bei einer Wertänderung das Ereignis `EO`.
4. `EO` löst das Sendeereignis `REQ` des internen `PUBLISH_1`-Blocks aus; der gepufferte Wert `Q` wird über die Datenverbindung als `SD_1` übergeben und auf dem konfigurierten Kanal veröffentlicht.
5. Nach erfolgreichem Senden bestätigt der `PUBLISH_1`-Block mit `CNF`, das nach außen geführt wird.

Technische Besonderheiten
- **Pufferung mit D-Flipflop**: Der zu sendende UDINT-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert. Dadurch wird verhindert, dass ein sich während der Verarbeitung ändernder Eingangswert den Sendevorgang stört.
- **Ereignisbasierte Entkopplung**: Durch das echte ereignisbasierte `E_D_FF_ANY` werden unmittelbare Rückkopplungsschleifen zwischen `PUBLISH_1` und `SUBSCRIBE_1` (z. B. auf demselben OPC-UA-Knoten) entkoppelt – der saubere Fix gegenüber dem Interim-Fix mit `AX_D_FF`.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `PUBLISH_1` (inkl. `REQ`/`SD_1`) wird nach innen verlegt; nach außen ist nur noch die AUDI-Adapter-Schnittstelle sichtbar.

Zustandsübersicht
1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Der Block ist bereit, Daten über den konfigurierten Kanal zu veröffentlichen.
3. **Sendeaktiv**: Ein am AUDI-Socket eintreffendes Ereignis puffert den Wert und veranlasst das Senden über den internen `PUBLISH_1`-Block.

Anwendungsszenarien
- **Netzwerkkopplung in AUDI-Anwendungen**: Senden von UDINT-Signalen (z. B. Schaltbefehle oder Statusbits) aus der Adapter-basierten Logik über Publish/Subscribe an entfernte Empfänger.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_SUBSCRIBE_1` werden Rückkopplungen zwischen eigenen Veröffentlichungen und dem eigenen Abonnement entkoppelt.

- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AUDI-Adapter setzen.

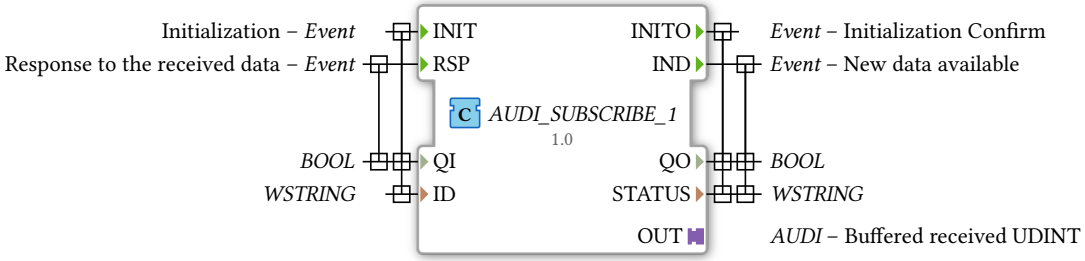
Vergleich mit ähnlichen Bausteinen

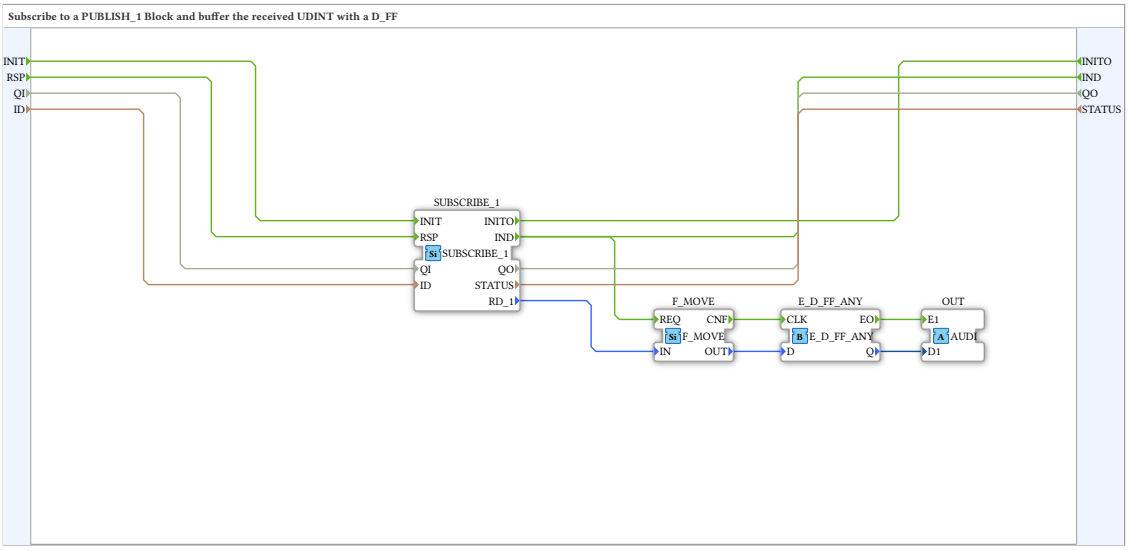
- **PUBLISH_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`REQ`, `SD_1`, ...). **AUDI_PUBLISH_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den AUDI-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen UDINT-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AUDI_PUBLISH_1** vereint Puffern und Veröffentlichen in einem Baustein.

Fazit

AUDI_PUBLISH_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `PUBLISH_1` mit der Adapter-basierten UDINT-Verarbeitung. Die interne D-Flipflop-Pufferung stabilisiert den Sendewert und entkoppelt Rückkopplungsschleifen; die reduzierte AUDI-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

AUDI_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUDI_SUBSCRIBE_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `SUBSCRIBE_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AUDI-Adapter** abbildet. Er abonniert einen über die `ID` konfigurierten Publish-Kanal und stellt den empfangenen **UDINT**-Wert gepuffert über einen D-Flipflop (E_D_FF_ANY) am Adapter-Plug `OUT` bereit. Damit lassen sich Netzkommunikation und UDINT-Signalverarbeitung in der AUDI-Adapter-Welt sauber verbinden.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`
- **RSP** (Event): Response-Ereignis, verbunden mit `QI`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Abonnement-Dienst
- **ID** (WSTRING): Identifikator des zu abonnierenden Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
OUT	adapter::types::unidirectional::AUDI	Plug (Ausgang)	Gepufferter empfangener UDINT-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `SUBSCRIBE_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Der interne `SUBSCRIBE_1`-Block überwacht den konfigurierten Kanal. Sobald der zugehörige `PUBLISH_1`-Block Daten veröffentlicht, erzeugt er das Indikationsereignis `IND` und legt den empfangenen Wert an `RD_1` an.
3. `IND` taktet den internen **E_D_FF_ANY**-Flipflop (`CLK`), der den Wert von `RD_1` an seinem Dateneingang `D` übernimmt.
4. Der gespeicherte Wert erscheint stabil am Ausgang `Q` und wird über die Datenverbindung auf `OUT.D1` des AX-Plugs gelegt.
5. Das Ereignis `EO` des Flipflops wird auf den Ereignis-Ausgang `OUT.E1` gelegt, sodass nachgeschaltete Bausteine über die neue Information benachrichtigt werden.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der empfangene UDINT-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert und bleibt stabil am Ausgang, bis der nächste Wert empfangen wird. `EO` feuert nur bei einer Änderung von `Q`.
- **RSP ist ein No-Op bei Subscribern**: Laut Quellcode-Analyse von `4diac-forte` (`CCommFB::sendData()`) führt das `RSP`-Ereignis bei `SUBSCRIBE_1` (Service-Typ Subscriber) keine Aktion aus – der Zweig `!= e_Subscriber` wird übersprungen. Das Event-Input-Paar `INIT`/`RSP` wird nur für die Quittierung durchgeführt und ist für die Quittierung im Subscriber-Fall funktionslos. Es wird dennoch nach außen geführt, damit ein konfigurierbarer Anschluss besteht.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `SUBSCRIBE_1` (inkl. `IND`/`RD_1`) wird nach innen verlegt; nach außen ist nur noch die AUDI-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Das Abonnement ist aktiv; der Block wartet auf Daten vom Publisher.
3. **Datenempfang**: Bei `IND` wird der empfangene UDINT-Wert in das interne D-Flipflop übernommen und über `OUT` ausgegeben.

Anwendungsszenarien

- **Netzwerkkopplung in AUDI-Anwendungen**: Empfang von UDINT-Signalen (z. B. Schaltbefehle oder Statusbits) über Publish/Subscribe und Weiterleitung über AUDI-Adapter an die Adapter-basierte Logik.

- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_PUBLISH_1` und einem echten ereignisbasierten `E_D_FF_ANY` lässt sich die Rückkopplung zwischen `SUBSCRIBE_1` und `PUBLISH_1` entkoppeln (sauberer Fix gegenüber dem Interim-Fix mit `AX_D_FF`).
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AUDI-Adapter setzen.

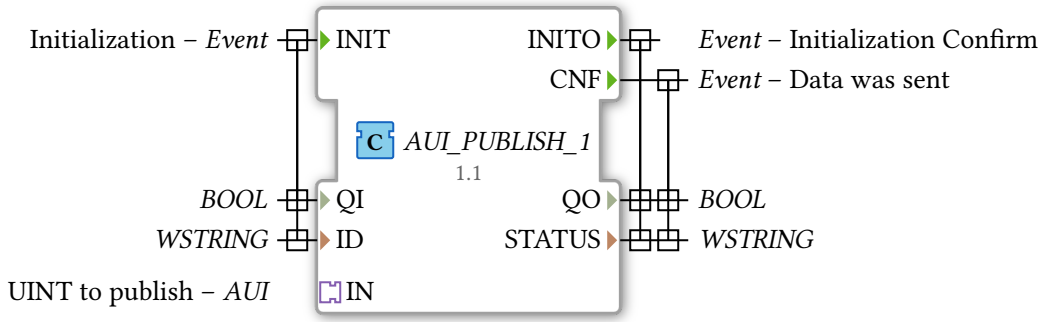
Vergleich mit ähnlichen Bausteinen

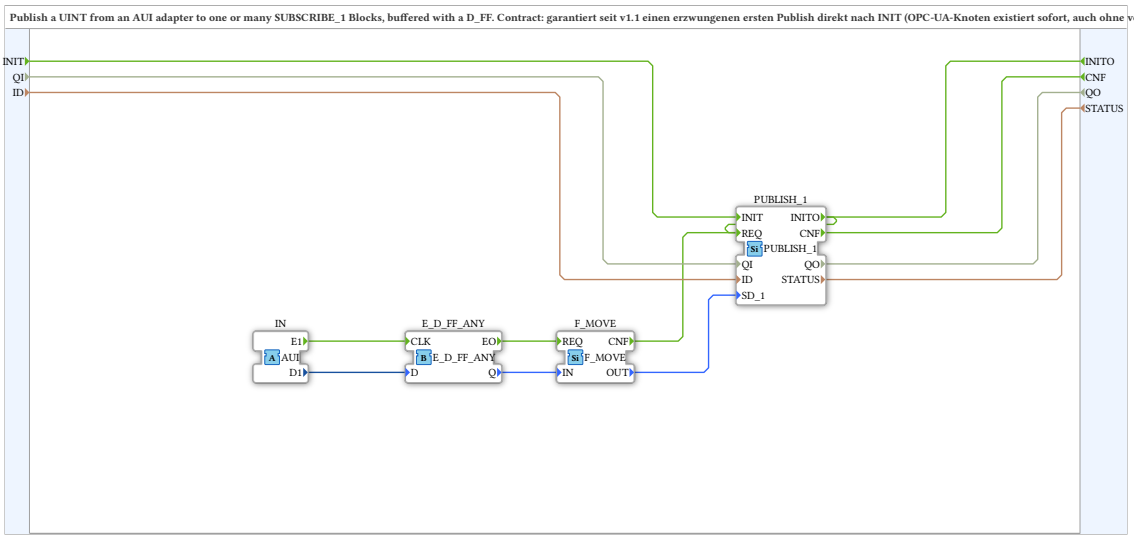
- **SUBSCRIBE_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`IND`, `RD_1`, ...). **AUDI_SUBSCRIBE_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den AUDI-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen UDINT-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AUDI_SUBSCRIBE_1** vereint Subscribing und Pufferung in einem Baustein.

Fazit

AUDI_SUBSCRIBE_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `SUBSCRIBE_1` mit der Adapter-basierten UDINT-Verarbeitung. Die interne D-Flipflop-Pufferung hält den empfangenen Wert stabil, bis neue Daten eintreffen, und die reduzierte AUDI-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

AUI_PUBLISH_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUI_PUBLISH_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `PUBLISH_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AUI-Adapter** abbildet. Ein am Adapter-Socket `IN` anliegender **UINT**-Wert wird über ein internes D-Flipflop (E_D_FF_ANY) gepuffert und anschließend über den konfigurierten Publish-Kanal an alle zugehörigen `SUBSCRIBE_1`-Empfänger gesendet.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`
- **CNF** (Event): Bestätigung, dass die Daten gesendet wurden, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Publish-Dienst
- **ID** (WSTRING): Identifikator des Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::AUI	Socket (Eingang)	Zu veröffentlichender UINT-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `PUBLISH_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Sobald der AUI-Socket `IN` ein Ereignis auf `IN.E1` liefert, wird der gleichzeitig an `IN.D1` anliegende UINT-Wert in das interne **E_D_FF_ANY**-Flipflop übernommen (`CLK` bzw. `D`).
3. Das Flipflop hält den Wert stabil an seinem Ausgang `Q` und erzeugt bei einer Wertänderung das Ereignis `EO`.
4. `EO` löst das Sendeereignis `REQ` des internen `PUBLISH_1`-Blocks aus; der gepufferte Wert `Q` wird über die Datenverbindung als `SD_1` übergeben und auf dem konfigurierten Kanal veröffentlicht.
5. Nach erfolgreichem Senden bestätigt der `PUBLISH_1`-Block mit `CNF`, das nach außen geführt wird.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der zu sendende UINT-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert. Dadurch wird verhindert, dass ein sich während der Verarbeitung ändernder Eingangswert den Sendevorgang stört.
- **Ereignisbasierte Entkopplung**: Durch das echte ereignisbasierte `E_D_FF_ANY` werden unmittelbare Rückkopplungsschleifen zwischen `PUBLISH_1` und `SUBSCRIBE_1` (z. B. auf demselben OPC-UA-Knoten) entkoppelt – der saubere Fix gegenüber dem Interim-Fix mit `AX_D_FF`.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `PUBLISH_1` (inkl. `REQ`/`SD_1`) wird nach innen verlegt; nach außen ist nur noch die AUI-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Der Block ist bereit, Daten über den konfigurierten Kanal zu veröffentlichen.
3. **Sendeaktiv**: Ein am AUI-Socket eintreffendes Ereignis puffert den Wert und veranlasst das Senden über den internen `PUBLISH_1`-Block.

Anwendungsszenarien

- **Netzwerkkopplung in AUI-Anwendungen**: Senden von UINT-Signalen (z. B. Schaltbefehle oder Statusbits) aus der Adapter-basierten Logik über Publish/Subscribe an entfernte Empfänger.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_SUBSCRIBE_1` werden Rückkopplungen zwischen eigenen Veröffentlichungen und dem eigenen Abonnement entkoppelt.
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AUI-Adapter setzen.

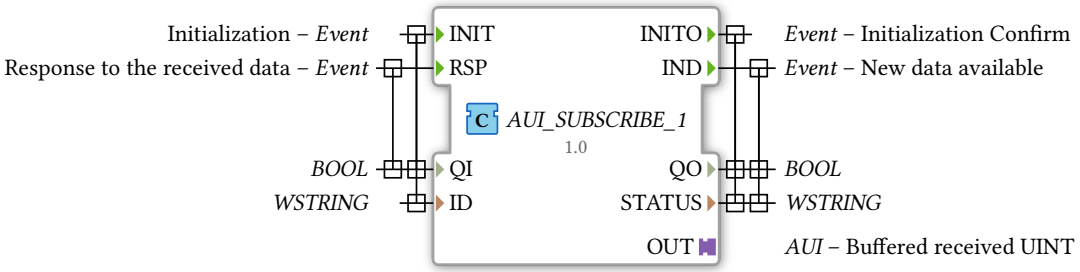
Vergleich mit ähnlichen Bausteinen

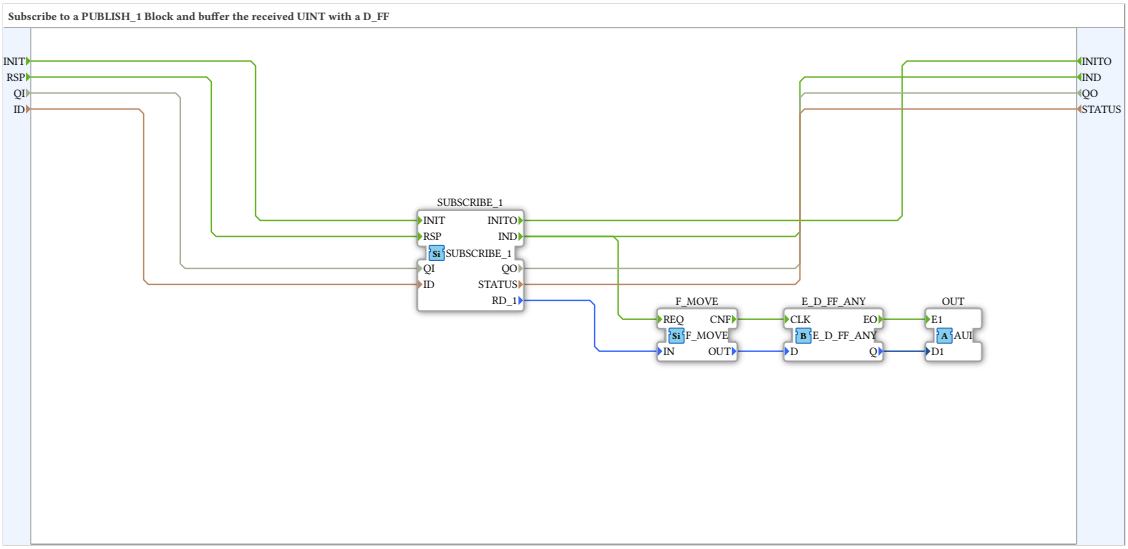
- **PUBLISH_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`REQ`, `SD_1`, ...). **AUI_PUBLISH_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den AUI-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen UINT-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AUI_PUBLISH_1** vereint Puffern und Veröffentlichen in einem Baustein.

Fazit

AUI_PUBLISH_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `PUBLISH_1` mit der Adapter-basierten UINT-Verarbeitung. Die interne D-Flipflop-Pufferung stabilisiert den Sendewert und entkoppelt Rückkopplungsschleifen; die reduzierte AUI-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

AUI_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUI_SUBSCRIBE_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `SUBSCRIBE_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AUI-Adapter** abbildet. Er abonniert einen über die `ID` konfigurierten Publish-Kanal und stellt den empfangenen **UINT**-Wert gepuffert über einen D-Flipflop (E_D_FF_ANY) am Adapter-Plug `OUT` bereit. Damit lassen sich Netzkommunikation und UINT-Signalverarbeitung in der AUI-Adapter-Welt sauber verbinden.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`
- **RSP** (Event): Response-Ereignis, verbunden mit `QI`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Abonnement-Dienst
- **ID** (WSTRING): Identifikator des zu abonnierenden Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
OUT	adapter::types::unidirectional::AUI	Plug (Ausgang)	Gepufferter empfangener UINT-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `SUBSCRIBE_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Der interne `SUBSCRIBE_1`-Block überwacht den konfigurierten Kanal. Sobald der zugehörige `PUBLISH_1`-Block Daten veröffentlicht, erzeugt er das Indikationsereignis `IND` und legt den empfangenen Wert an `RD_1` an.
3. `IND` taktet den internen **E_D_FF_ANY**-Flipflop (`CLK`), der den Wert von `RD_1` an seinem Dateneingang `D` übernimmt.
4. Der gespeicherte Wert erscheint stabil am Ausgang `Q` und wird über die Datenverbindung auf `OUT.D1` des AX-Plugs gelegt.
5. Das Ereignis `EO` des Flipflops wird auf den Ereignis-Ausgang `OUT.E1` gelegt, sodass nachgeschaltete Bausteine über die neue Information benachrichtigt werden.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der empfangene UINT-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert und bleibt stabil am Ausgang, bis der nächste Wert empfangen wird. `EO` feuert nur bei einer Änderung von `Q`.
- **RSP ist ein No-Op bei Subscribern**: Laut Quellcode-Analyse von `4diac-forte` (`CCommFB::sendData()`) führt das `RSP`-Ereignis bei `SUBSCRIBE_1` (Service-Typ Subscriber) keine Aktion aus – der Zweig `!= e_Subscriber` wird übersprungen. Das Event-Input-Paar `INIT`/`RSP` wird nur für die Quittierung durchgeführt und ist für die Quittierung im Subscriber-Fall funktionslos. Es wird dennoch nach außen geführt, damit ein konfigurierbarer Anschluss besteht.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `SUBSCRIBE_1` (inkl. `IND`/`RD_1`) wird nach innen verlegt; nach außen ist nur noch die AUI-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Das Abonnement ist aktiv; der Block wartet auf Daten vom Publisher.
3. **Datenempfang**: Bei `IND` wird der empfangene UINT-Wert in das interne D-Flipflop übernommen und über `OUT` ausgegeben.

Anwendungsszenarien

- **Netzwerkkopplung in AUI-Anwendungen**: Empfang von UINT-Signalen (z. B. Schaltbefehle oder Statusbits) über Publish/Subscribe und Weiterleitung über AUI-Adapter an die Adapter-basierte Logik.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_PUBLISH_1` und einem echten ereignisbasierten `E_D_FF_ANY` lässt sich die Rückkopplung zwischen `SUBSCRIBE_1` und `PUBLISH_1` entkoppeln (sauberer Fix gegenüber dem Interim-Fix mit `AX_D_FF`).
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AUI-Adapter setzen.

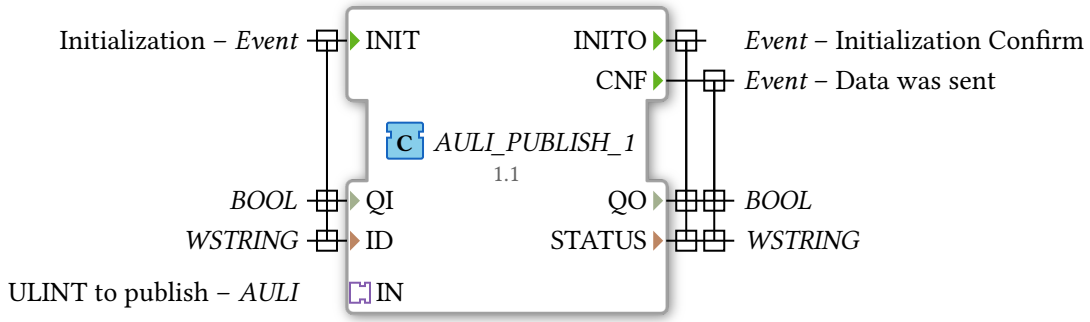
Vergleich mit ähnlichen Bausteinen

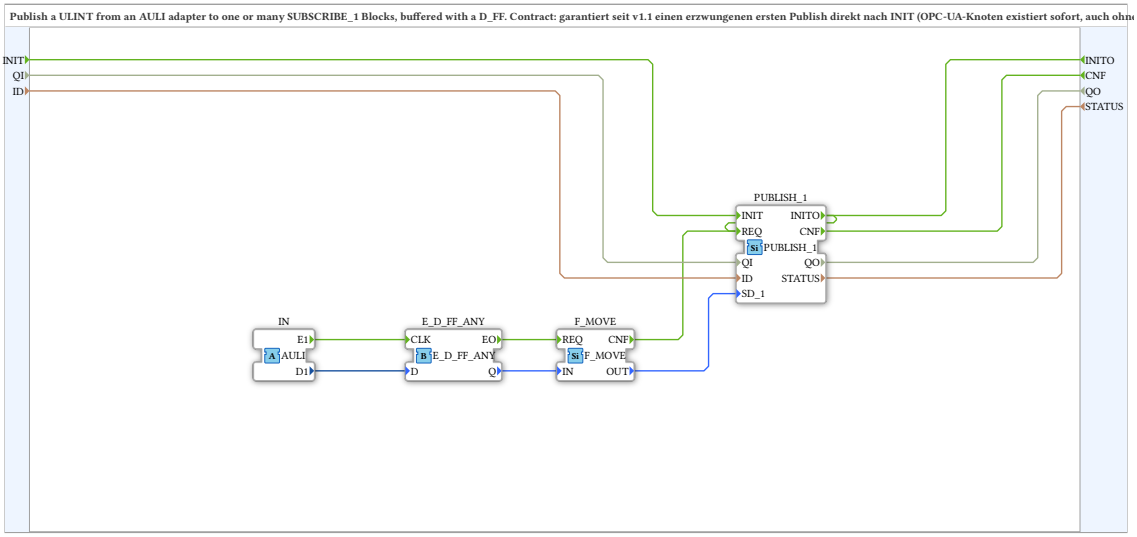
- **SUBSCRIBE_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`IND`, `RD_1`, ...). **AUI_SUBSCRIBE_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den AUI-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen UINT-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AUI_SUBSCRIBE_1** vereint Subscribing und Pufferung in einem Baustein.

Fazit

AUI_SUBSCRIBE_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `SUBSCRIBE_1` mit der Adapter-basierten UINT-Verarbeitung. Die interne D-Flipflop-Pufferung hält den empfangenen Wert stabil, bis neue Daten eintreffen, und die reduzierte AUI-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

AULI_PUBLISH_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AULI_PUBLISH_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `PUBLISH_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AULI-Adapter** abbildet. Ein am Adapter-Socket `IN` anliegender **ULINT**-Wert wird über ein internes D-Flipflop (E_D_FF_ANY) gepuffert und anschließend über den konfigurierten Publish-Kanal an alle zugehörigen `SUBSCRIBE_1`-Empfänger gesendet.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`
- **CNF** (Event): Bestätigung, dass die Daten gesendet wurden, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Publish-Dienst
- **ID** (WSTRING): Identifikator des Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::AULI	Socket (Eingang)	Zu veröffentlichender ULINT-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `PUBLISH_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Sobald der AULI-Socket `IN` ein Ereignis auf `IN.E1` liefert, wird der gleichzeitig an `IN.D1` anliegende ULINT-Wert in das interne **E_D_FF_ANY**-Flipflop übernommen (`CLK` bzw. `D`).
3. Das Flipflop hält den Wert stabil an seinem Ausgang `Q` und erzeugt bei einer Wertänderung das Ereignis `EO`.
4. `EO` löst das Sendeereignis `REQ` des internen `PUBLISH_1`-Blocks aus; der gepufferte Wert `Q` wird über die Datenverbindung als `SD_1` übergeben und auf dem konfigurierten Kanal veröffentlicht.
5. Nach erfolgreichem Senden bestätigt der `PUBLISH_1`-Block mit `CNF`, das nach außen geführt wird.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der zu sendende ULINT-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert. Dadurch wird verhindert, dass ein sich während der Verarbeitung ändernder Eingangswert den Sendevorgang stört.
- **Ereignisbasierte Entkopplung**: Durch das echte ereignisbasierte `E_D_FF_ANY` werden unmittelbare Rückkopplungsschleifen zwischen `PUBLISH_1` und `SUBSCRIBE_1` (z. B. auf demselben OPC-UA-Knoten) entkoppelt – der saubere Fix gegenüber dem Interim-Fix mit `AX_D_FF`.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `PUBLISH_1` (inkl. `REQ`/`SD_1`) wird nach innen verlegt; nach außen ist nur noch die AULI-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Der Block ist bereit, Daten über den konfigurierten Kanal zu veröffentlichen.
3. **Sendeaktiv**: Ein am AULI-Socket eintreffendes Ereignis puffert den Wert und veranlasst das Senden über den internen `PUBLISH_1`-Block.

Anwendungsszenarien

- **Netzwerkkopplung in AULI-Anwendungen**: Senden von ULINT-Signalen (z. B. Schaltbefehle oder Statusbits) aus der Adapter-basierten Logik über Publish/Subscribe an entfernte Empfänger.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_SUBSCRIBE_1` werden Rückkopplungen zwischen eigenen Veröffentlichungen und dem eigenen Abonnement entkoppelt.
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AULI-Adapter setzen.

Vergleich mit ähnlichen Bausteinen

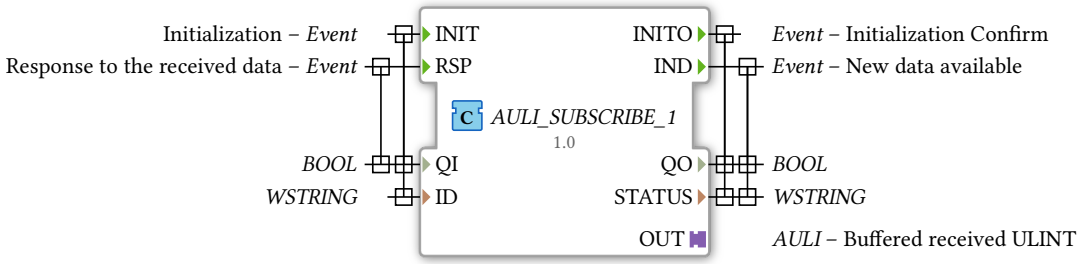
- **PUBLISH_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`REQ`, `SD_1`, …). **AULI_PUBLISH_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den AULI-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.

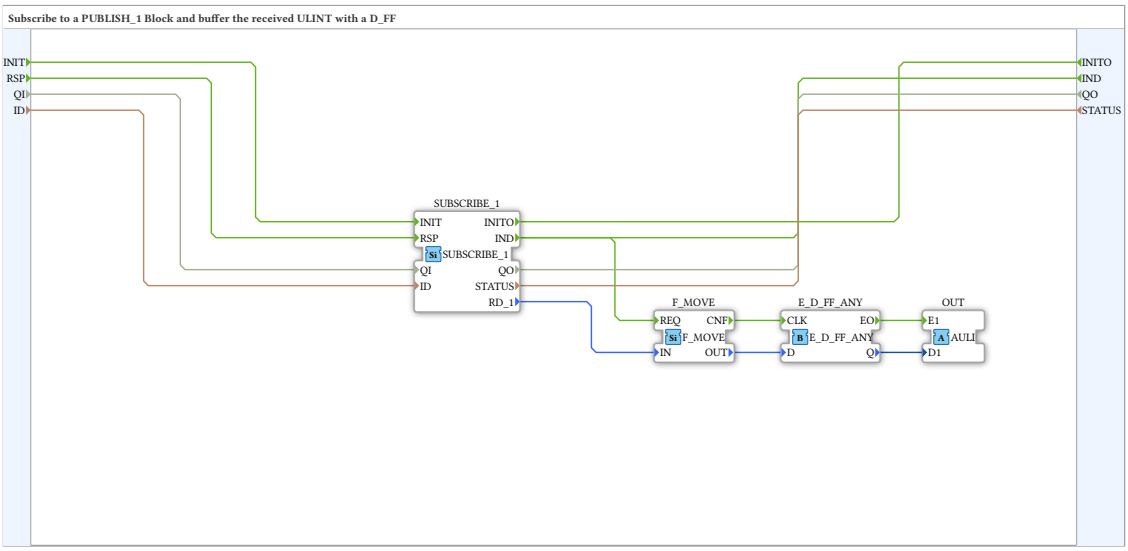
- **AX_D_FF**: Puffert einen ULINT-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AULI_PUBLISH_1** vereint Puffern und Veröffentlichen in einem Baustein.

Fazit

AULI_PUBLISH_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `PUBLISH_1` mit der Adapter-basierten ULINT-Verarbeitung. Die interne D-Flipflop-Pufferung stabilisiert den Sendewert und entkoppelt Rückkopplungsschleifen; die reduzierte AULI-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

AULI_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AULI_SUBSCRIBE_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `SUBSCRIBE_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AULI-Adapter** abbildet. Er abonniert einen über die `ID` konfigurierten Publish-Kanal und stellt den empfangenen **ULINT**-Wert gepuffert über einen D-Flipflop (E_D_FF_ANY) am Adapter-Plug `OUT` bereit. Damit lassen sich Netzkommunikation und ULINT-Signalverarbeitung in der AULI-Adapter-Welt sauber verbinden.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`
- **RSP** (Event): Response-Ereignis, verbunden mit `QI`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Abonnement-Dienst

- **ID** (WSTRING): Identifikator des zu abonnierenden Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
OUT	adapter::types::unidirectional::AULI	Plug (Ausgang)	Gepufferter empfangener ULINT-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `SUBSCRIBE_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Der interne `SUBSCRIBE_1`-Block überwacht den konfigurierten Kanal. Sobald der zugehörige `PUBLISH_1`-Block Daten veröffentlicht, erzeugt er das Indikationsereignis `IND` und legt den empfangenen Wert an `RD_1` an.
3. `IND` taktet den internen **E_D_FF_ANY**-Flipflop (`CLK`), der den Wert von `RD_1` an seinem Dateneingang `D` übernimmt.
4. Der gespeicherte Wert erscheint stabil am Ausgang `Q` und wird über die Datenverbindung auf `OUT.D1` des AX-Plugs gelegt.
5. Das Ereignis `EO` des Flipflops wird auf den Ereignis-Ausgang `OUT.E1` gelegt, sodass nachgeschaltete Bausteine über die neue Information benachrichtigt werden.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der empfangene ULINT-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert und bleibt stabil am Ausgang, bis der nächste Wert empfangen wird. `EO` feuert nur bei einer Änderung von `Q`.
- **RSP ist ein No-Op bei Subscribern**: Laut Quellcode-Analyse von `4diac-forte` (`CCommFB::sendData()`) führt das `RSP`-Ereignis bei `SUBSCRIBE_1` (Service-Typ Subscriber) keine Aktion aus – der Zweig `!= e_Subscriber` wird übersprungen. Das Event-Input-Paar `INIT`/`RSP` wird nur für die Quittierung durchgeführt und ist für die Quittierung im Subscriber-Fall funktionslos. Es wird dennoch nach außen geführt, damit ein konfigurierbarer Anschluss besteht.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `SUBSCRIBE_1` (inkl. `IND`/`RD_1`) wird nach innen verlegt; nach außen ist nur noch die AULI-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Das Abonnement ist aktiv; der Block wartet auf Daten vom Publisher.
3. **Datenempfang**: Bei `IND` wird der empfangene ULINT-Wert in das interne D-Flipflop übernommen und über `OUT` ausgegeben.

Anwendungsszenarien

- **Netzwerkkopplung in AULI-Anwendungen**: Empfang von ULINT-Signalen (z. B. Schaltbefehle oder Statusbits) über Publish/Subscribe und Weiterleitung über AULI-Adapter an die Adapter-basierte Logik.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_PUBLISH_1` und einem echten ereignisbasierten `E_D_FF_ANY` lässt sich die Rückkopplung zwischen `SUBSCRIBE_1` und `PUBLISH_1` entkoppeln (sauberer Fix gegenüber dem Interim-Fix mit `AX_D_FF`).
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AULI-Adapter setzen.

Vergleich mit ähnlichen Bausteinen

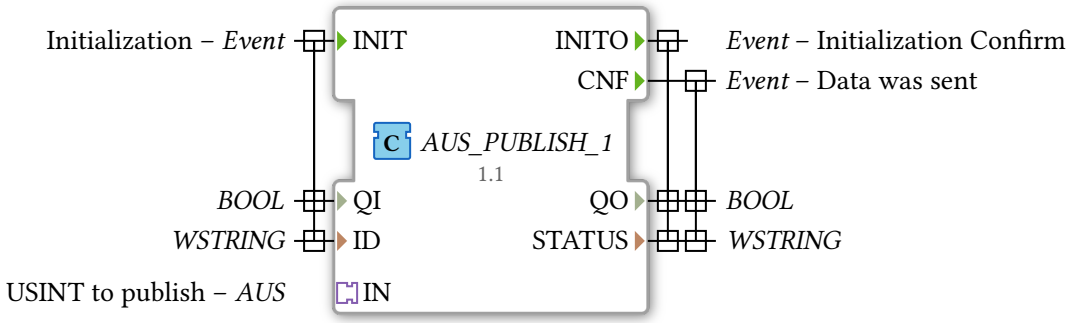
- **SUBSCRIBE_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`IND`, `RD_1`, …). **AULI_SUBSCRIBE_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den AULI-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.

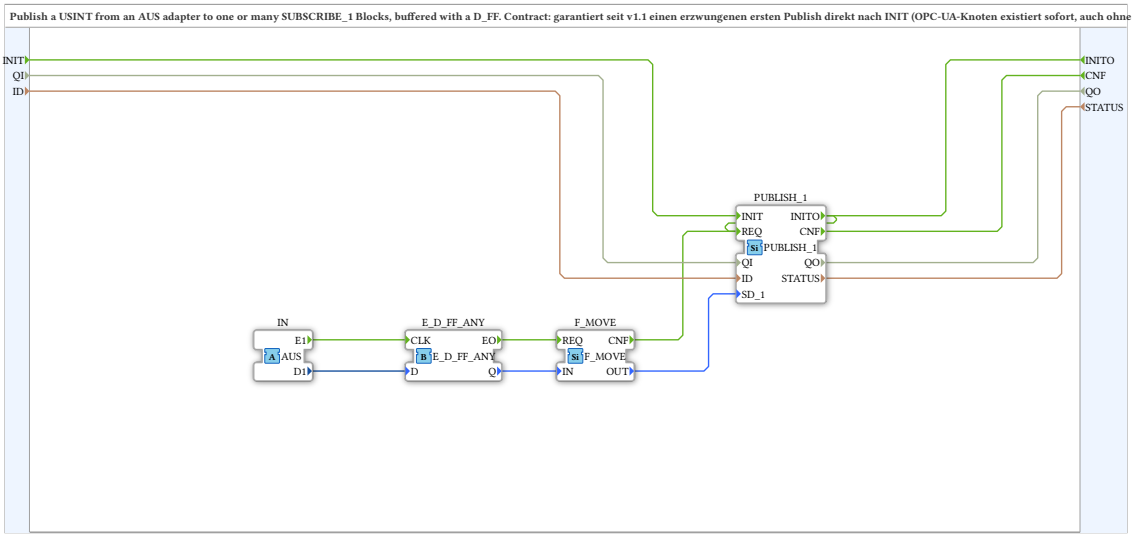
- **AX_D_FF**: Puffert einen ULINT-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AULI_SUBSCRIBE_1** vereint Subscribing und Pufferung in einem Baustein.

Fazit

AULI_SUBSCRIBE_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `SUBSCRIBE_1` mit der Adapter-basierten ULINT-Verarbeitung. Die interne D-Flipflop-Pufferung hält den empfangenen Wert stabil, bis neue Daten eintreffen, und die reduzierte AULI-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

AUS_PUBLISH_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUS_PUBLISH_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `PUBLISH_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AUS-Adapter** abbildet. Ein am Adapter-Socket `IN` anliegender **USINT**-Wert wird über ein internes D-Flipflop (E_D_FF_ANY) gepuffert und anschließend über den konfigurierten Publish-Kanal an alle zugehörigen `SUBSCRIBE_1`-Empfänger gesendet.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`
- **CNF** (Event): Bestätigung, dass die Daten gesendet wurden, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Publish-Dienst
- **ID** (WSTRING): Identifikator des Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::AUS	Socket (Eingang)	Zu veröffentlichender USINT-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `PUBLISH_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Sobald der AUS-Socket `IN` ein Ereignis auf `IN.E1` liefert, wird der gleichzeitig an `IN.D1` anliegende USINT-Wert in das interne **E_D_FF_ANY**-Flipflop übernommen (`CLK` bzw. `D`).
3. Das Flipflop hält den Wert stabil an seinem Ausgang `Q` und erzeugt bei einer Wertänderung das Ereignis `EO`.
4. `EO` löst das Sendeereignis `REQ` des internen `PUBLISH_1`-Blocks aus; der gepufferte Wert `Q` wird über die Datenverbindung als `SD_1` übergeben und auf dem konfigurierten Kanal veröffentlicht.
5. Nach erfolgreichem Senden bestätigt der `PUBLISH_1`-Block mit `CNF`, das nach außen geführt wird.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der zu sendende USINT-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert. Dadurch wird verhindert, dass ein sich während der Verarbeitung ändernder Eingangswert den Sendevorgang stört.
- **Ereignisbasierte Entkopplung**: Durch das echte ereignisbasierte `E_D_FF_ANY` werden unmittelbare Rückkopplungsschleifen zwischen `PUBLISH_1` und `SUBSCRIBE_1` (z. B. auf demselben OPC-UA-Knoten) entkoppelt – der saubere Fix gegenüber dem Interim-Fix mit `AX_D_FF`.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `PUBLISH_1` (inkl. `REQ`/`SD_1`) wird nach innen verlegt; nach außen ist nur noch die AUS-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Der Block ist bereit, Daten über den konfigurierten Kanal zu veröffentlichen.
3. **Sendeaktiv**: Ein am AUS-Socket eintreffendes Ereignis puffert den Wert und veranlasst das Senden über den internen `PUBLISH_1`-Block.

Anwendungsszenarien

- **Netzwerkkopplung in AUS-Anwendungen**: Senden von USINT-Signalen (z. B. Schaltbefehle oder Statusbits) aus der Adapter-basierten Logik über Publish/Subscribe an entfernte Empfänger.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_SUBSCRIBE_1` werden Rückkopplungen zwischen eigenen Veröffentlichungen und dem eigenen Abonnement entkoppelt.
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AUS-Adapter setzen.

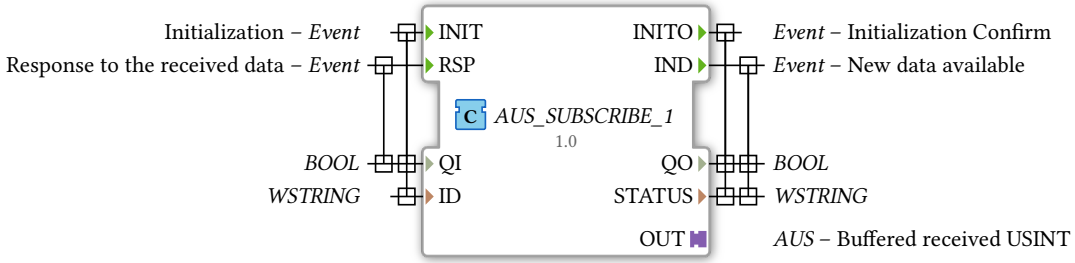
Vergleich mit ähnlichen Bausteinen

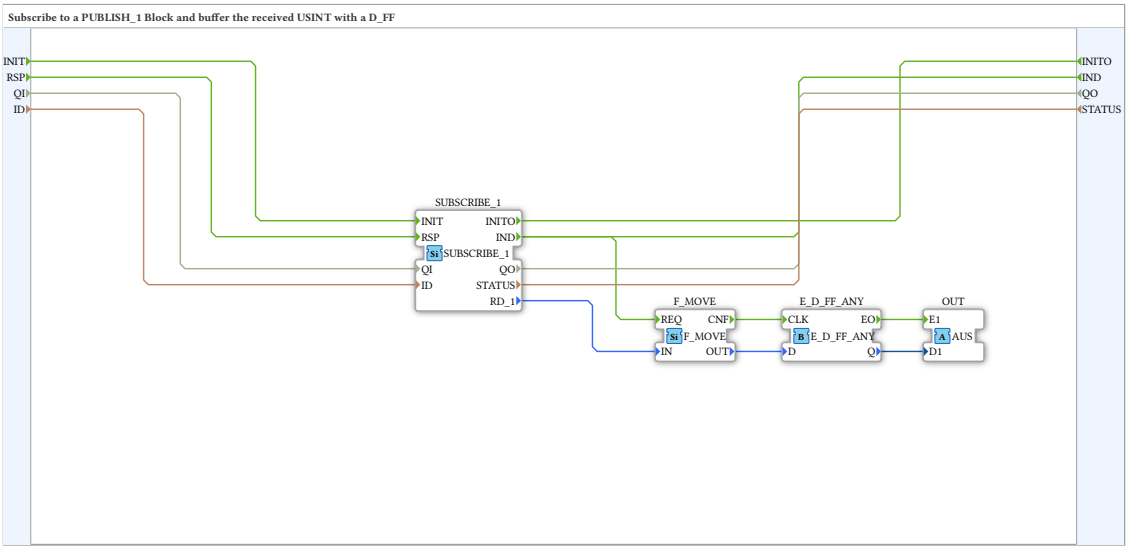
- **PUBLISH_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`REQ`, `SD_1`, …). **AUS_PUBLISH_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den AUS-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen USINT-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AUS_PUBLISH_1** vereint Puffern und Veröffentlichen in einem Baustein.

Fazit

AUS_PUBLISH_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `PUBLISH_1` mit der Adapter-basierten USINT-Verarbeitung. Die interne D-Flipflop-Pufferung stabilisiert den Sendewert und entkoppelt Rückkopplungsschleifen; die reduzierte AUS-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

AUS_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUS_SUBSCRIBE_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `SUBSCRIBE_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AUS-Adapter** abbildet. Er abonniert einen über die `ID` konfigurierten Publish-Kanal und stellt den empfangenen **USINT**-Wert gepuffert über einen D-Flipflop (E_D_FF_ANY) am Adapter-Plug `OUT` bereit. Damit lassen sich Netzkommunikation und USINT-Signalverarbeitung in der AUS-Adapter-Welt sauber verbinden.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`
- **RSP** (Event): Response-Ereignis, verbunden mit `QI`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Abonnement-Dienst
- **ID** (WSTRING): Identifikator des zu abonnierenden Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
OUT	adapter::types::unidirectional::AUS	Plug (Ausgang)	Gepufferter empfangener USINT-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `SUBSCRIBE_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Der interne `SUBSCRIBE_1`-Block überwacht den konfigurierten Kanal. Sobald der zugehörige `PUBLISH_1`-Block Daten veröffentlicht, erzeugt er das Indikationsereignis `IND` und legt den empfangenen Wert an `RD_1` an.
3. `IND` taktet den internen **E_D_FF_ANY**-Flipflop (`CLK`), der den Wert von `RD_1` an seinem Dateneingang `D` übernimmt.
4. Der gespeicherte Wert erscheint stabil am Ausgang `Q` und wird über die Datenverbindung auf `OUT.D1` des AX-Plugs gelegt.
5. Das Ereignis `EO` des Flipflops wird auf den Ereignis-Ausgang `OUT.E1` gelegt, sodass nachgeschaltete Bausteine über die neue Information benachrichtigt werden.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der empfangene USINT-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert und bleibt stabil am Ausgang, bis der nächste Wert empfangen wird. `EO` feuert nur bei einer Änderung von `Q`.
- **RSP ist ein No-Op bei Subscribern**: Laut Quellcode-Analyse von `4diac-forte` (`CCommFB::sendData()`) führt das `RSP`-Ereignis bei `SUBSCRIBE_1` (Service-Typ Subscriber) keine Aktion aus – der Zweig `!= e_Subscriber` wird übersprungen. Das Event-Input-Paar `INIT`/`RSP` wird nur für die Quittierung durchgeführt und ist für die Quittierung im Subscriber-Fall funktionslos. Es wird dennoch nach außen geführt, damit ein konfigurierbarer Anschluss besteht.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `SUBSCRIBE_1` (inkl. `IND`/`RD_1`) wird nach innen verlegt; nach außen ist nur noch die AUS-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Das Abonnement ist aktiv; der Block wartet auf Daten vom Publisher.
3. **Datenempfang**: Bei `IND` wird der empfangene USINT-Wert in das interne D-Flipflop übernommen und über `OUT` ausgegeben.

Anwendungsszenarien

- **Netzwerkkopplung in AUS-Anwendungen**: Empfang von USINT-Signalen (z. B. Schaltbefehle oder Statusbits) über Publish/Subscribe und Weiterleitung über AUS-Adapter an die Adapter-basierte Logik.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_PUBLISH_1` und einem echten ereignisbasierten `E_D_FF_ANY` lässt sich die Rückkopplung zwischen `SUBSCRIBE_1` und `PUBLISH_1` entkoppeln (sauberer Fix gegenüber dem Interim-Fix mit `AX_D_FF`).
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AUS-Adapter setzen.

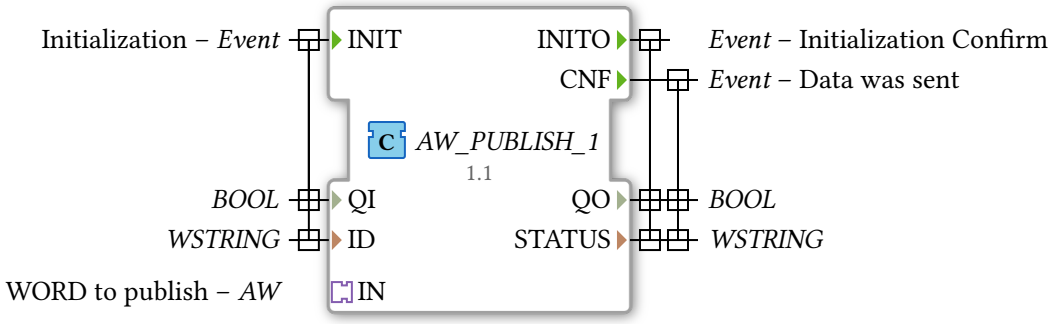
Vergleich mit ähnlichen Bausteinen

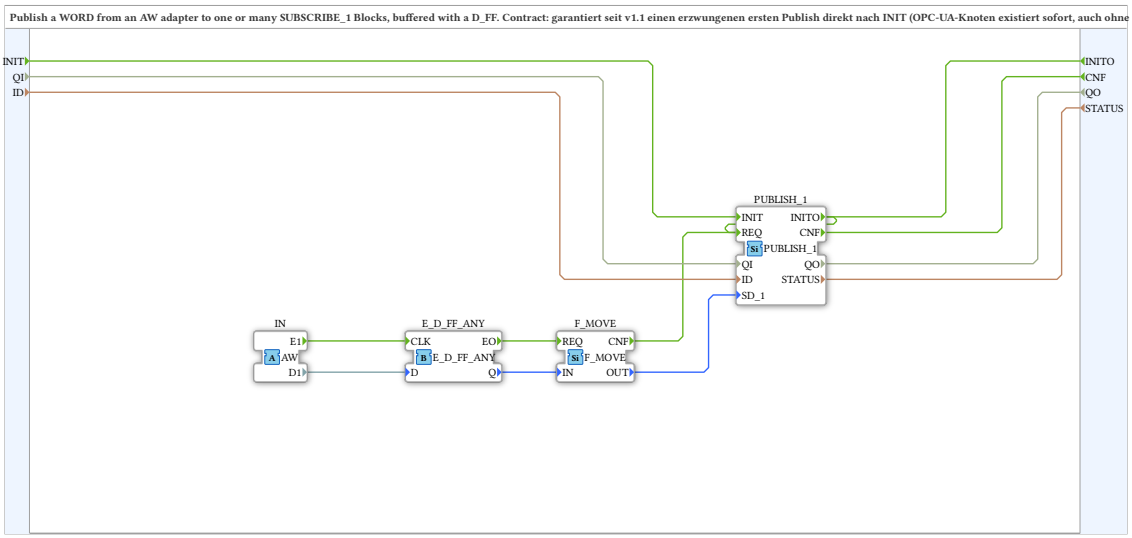
- **SUBSCRIBE_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`IND`, `RD_1`, …). **AUS_SUBSCRIBE_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den AUS-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen USINT-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AUS_SUBSCRIBE_1** vereint Subscribing und Pufferung in einem Baustein.

Fazit

AUS_SUBSCRIBE_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `SUBSCRIBE_1` mit der Adapter-basierten USINT-Verarbeitung. Die interne D-Flipflop-Pufferung hält den empfangenen Wert stabil, bis neue Daten eintreffen, und die reduzierte AUS-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

AW_PUBLISH_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AW_PUBLISH_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `PUBLISH_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AW-Adapter** abbildet. Ein am Adapter-Socket `IN` anliegender **WORD**-Wert wird über ein internes D-Flipflop (E_D_FF_ANY) gepuffert und anschließend über den konfigurierten Publish-Kanal an alle zugehörigen `SUBSCRIBE_1`-Empfänger gesendet.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`
- **CNF** (Event): Bestätigung, dass die Daten gesendet wurden, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Publish-Dienst
- **ID** (WSTRING): Identifikator des Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::AW	Socket (Eingang)	Zu veröffentlichender WORD-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `PUBLISH_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Sobald der AW-Socket `IN` ein Ereignis auf `IN.E1` liefert, wird der gleichzeitig an `IN.D1` anliegende WORD-Wert in das interne **E_D_FF_ANY**-Flipflop übernommen (`CLK` bzw. `D`).
3. Das Flipflop hält den Wert stabil an seinem Ausgang `Q` und erzeugt bei einer Wertänderung das Ereignis `EO`.
4. `EO` löst das Sendeereignis `REQ` des internen `PUBLISH_1`-Blocks aus; der gepufferte Wert `Q` wird über die Datenverbindung als `SD_1` übergeben und auf dem konfigurierten Kanal veröffentlicht.
5. Nach erfolgreichem Senden bestätigt der `PUBLISH_1`-Block mit `CNF`, das nach außen geführt wird.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der zu sendende WORD-Wert wird über ein internes `iec61499::events::E_D_FF_ANY` gepuffert. Dadurch wird verhindert, dass ein sich während der Verarbeitung ändernder Eingangswert den Sendevorgang stört.
- **Ereignisbasierte Entkopplung**: Durch das echte ereignisbasierte `E_D_FF_ANY` werden unmittelbare Rückkopplungsschleifen zwischen `PUBLISH_1` und `SUBSCRIBE_1` (z. B. auf demselben OPC-UA-Knoten) entkoppelt – der saubere Fix gegenüber dem Interim-Fix mit `AX_D_FF`.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `PUBLISH_1` (inkl. `REQ`/`SD_1`) wird nach innen verlegt; nach außen ist nur noch die AW-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Der Block ist bereit, Daten über den konfigurierten Kanal zu veröffentlichen.
3. **Sendeaktiv**: Ein am AW-Socket eintreffendes Ereignis puffert den Wert und veranlasst das Senden über den internen `PUBLISH_1`-Block.

Anwendungsszenarien

- **Netzwerkkopplung in AW-Anwendungen**: Senden von WORD-Signalen (z. B. Schaltbefehle oder Statusbits) aus der Adapter-basierten Logik über Publish/Subscribe an entfernte Empfänger.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_SUBSCRIBE_1` werden Rückkopplungen zwischen eigenen Veröffentlichungen und dem eigenen Abonnement entkoppelt.
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AW-Adapter setzen.

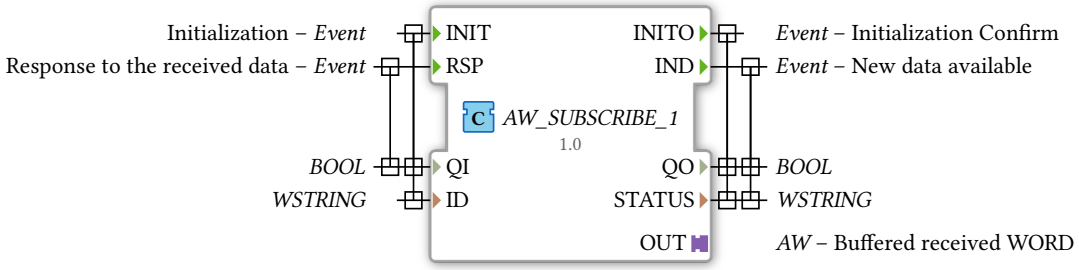
Vergleich mit ähnlichen Bausteinen

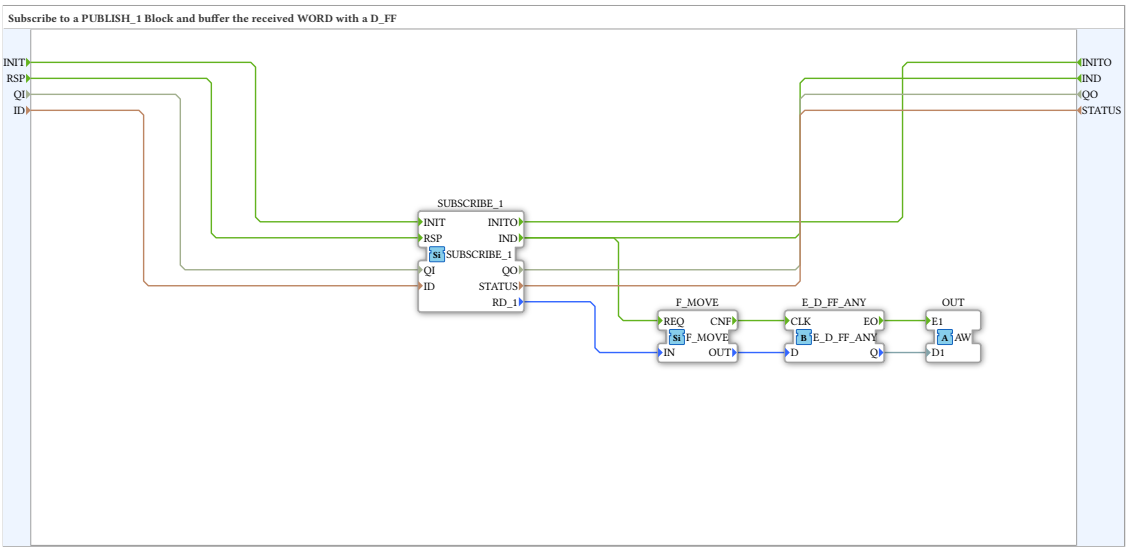
- **PUBLISH_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`REQ`, `SD_1`, …). **AW_PUBLISH_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (`E_D_FF_ANY`) und stellt nur noch den AW-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen WORD-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AW_PUBLISH_1** vereint Puffern und Veröffentlichen in einem Baustein.

Fazit

AW_PUBLISH_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `PUBLISH_1` mit der Adapter-basierten WORD-Verarbeitung. Die interne D-Flipflop-Pufferung stabilisiert den Sendewert und entkoppelt Rückkopplungsschleifen; die reduzierte AW-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

AW_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AW_SUBSCRIBE_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `SUBSCRIBE_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AW-Adapter** abbildet. Er abonniert einen über die `ID` konfigurierten Publish-Kanal und stellt den empfangenen **WORD**-Wert gepuffert über einen D-Flipflop (E_D_FF_ANY) am Adapter-Plug `OUT` bereit. Damit lassen sich Netzkommunikation und WORD-Signalverarbeitung in der AW-Adapter-Welt sauber verbinden.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`
- **RSP** (Event): Response-Ereignis, verbunden mit `QI`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Abonnement-Dienst
- **ID** (WSTRING): Identifikator des zu abonnierenden Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung

OUT	adapter::types::unidirectional::AW	Plug (Ausgang)	Gepufferter empfangener WORD-Wert

Funktionsweise

1. Über das INIT-Ereignis wird der interne SUBSCRIBE_1-Block mit QI und ID initialisiert. Bei Erfolg quittiert er mit INITO.
2. Der interne SUBSCRIBE_1-Block überwacht den konfigurierten Kanal. Sobald der zugehörige PUBLISH_1-Block Daten veröffentlicht, erzeugt er das Indikationsereignis IND und legt den empfangenen Wert an RD_1 an.
3. IND taktet den internen **E_D_FF_ANY**-Flipflop (CLK), der den Wert von RD_1 an seinem Dateneingang D übernimmt.
4. Der gespeicherte Wert erscheint stabil am Ausgang Q und wird über die Datenverbindung auf OUT.D1 des AX-Plugs gelegt.
5. Das Ereignis EO des Flipflops wird auf den Ereignis-Ausgang OUT.E1 gelegt, sodass nachgeschaltete Bausteine über die neue Information benachrichtigt werden.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der empfangene WORD-Wert wird über ein internes iec61499::events::E_D_FF_ANY gepuffert und bleibt stabil am Ausgang, bis der nächste Wert empfangen wird. EO feuert nur bei einer Änderung von Q.
- **RSP ist ein No-Op bei Subscribern**: Laut Quellcode-Analyse von 4diac-forte (CCommFB::sendData()) führt das RSP-Ereignis bei SUBSCRIBE_1 (Service-Typ Subscriber) keine Aktion aus – der Zweig != e_Subscriber wird übersprungen. Das Event-Input-Paar INIT/RSP wird nur für die Quittierung durchgeführt und ist für die Quittierung im Subscriber-Fall funktionslos. Es wird dennoch nach außen geführt, damit ein konfigurierbarer Anschluss besteht.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von SUBSCRIBE_1 (inkl. IND/RD_1) wird nach innen verlegt; nach außen ist nur noch die AW-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das INIT-Ereignis.
2. **Initialisiert**: Das Abonnement ist aktiv; der Block wartet auf Daten vom Publisher.
3. **Datenempfang**: Bei IND wird der empfangene WORD-Wert in das interne D-Flipflop übernommen und über OUT ausgegeben.

Anwendungsszenarien

- **Netzwerkkopplung in AW-Anwendungen**: Empfang von WORD-Signalen (z. B. Schaltbefehle oder Statusbits) über Publish/Subscribe und Weiterleitung über AW-Adapter an die Adapter-basierte Logik.
- **Event-Storm-Vermeidung**: Gemeinsam mit AX_PUBLISH_1 und einem echten ereignisbasierten E_D_FF_ANY lässt sich die Rückkopplung zwischen SUBSCRIBE_1 und PUBLISH_1 entkoppeln (sauberer Fix gegenüber dem Interim-Fix mit AX_D_FF).
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AW-Adapter setzen.

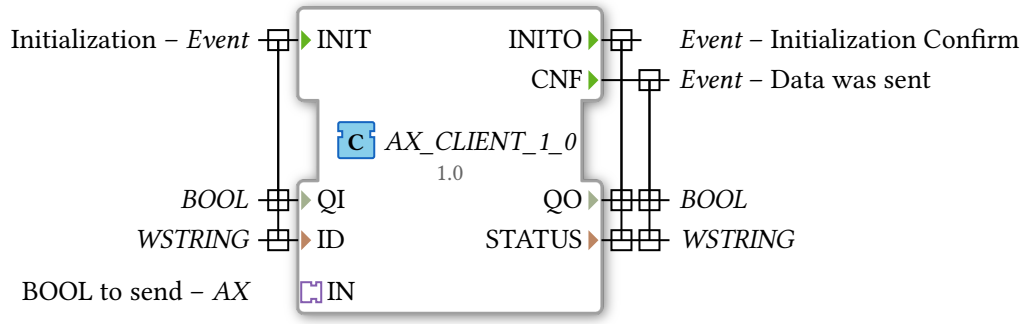
Vergleich mit ähnlichen Bausteinen

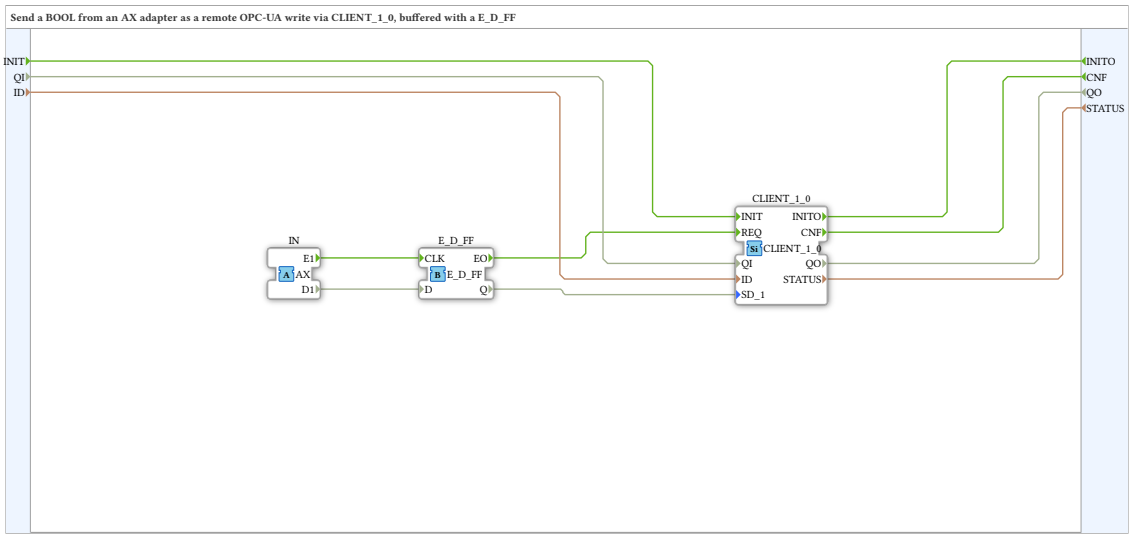
- **SUBSCRIBE_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (IND, RD_1, …). **AW_SUBSCRIBE_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop (E_D_FF_ANY) und stellt nur noch den AW-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen WORD-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AW_SUBSCRIBE_1** vereint Subscribing und Pufferung in einem Baustein.

Fazit

AW_SUBSCRIBE_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins SUBSCRIBE_1 mit der Adapter-basierten WORD-Verarbeitung. Die interne D-Flipflop-Pufferung hält den empfangenen Wert stabil, bis neue Daten eintreffen, und die reduzierte AW-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

AX_CLIENT_1_0





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AX_CLIENT_1_0** ist ein Composite-Funktionsblock, der den netzwerkbasierten `CLIENT_1_0`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AX-Adapter** abbildet. Ein am Adapter-Socket `IN` anliegender **BOOL**-Wert wird über ein internes D-Flipflop (E_D_FF) gepuffert und anschließend über `CLIENT_1_0` als OPC-UA-**Write** an den unter `ID` konfigurierten entfernten Knoten gesendet.

Im Unterschied zu **AX_PUBLISH_1** (lokales Publish/Subscribe) schreibt `CLIENT_1_0` aktiv auf einen **entfernten** Server – laut der OPC-UA-Adressierungstabelle in `opcua.adoc` ist das der korrekte Baustein für ein Remote-`WRITE`, während `PUBLISH_1`/`AX_PUBLISH_1` nur lokal veröffentlichen können.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`
- **CNF** (Event): Bestätigung, dass die Daten gesendet wurden, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, öffnet (TRUE) bzw. schließt (FALSE) die Verbindung zum Server
- **ID** (WSTRING): Identifikator der Verbindung (OPC-UA-Adresse des Zielknotens, z. B. `opc_ua[WRITE;opc.tcp://192.168.1.12:4840#;...]`)

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Verbindungsstatus

- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::AX	Socket (Eingang)	Zu sendender BOOL-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `CLIENT_1_0`-Block mit `QI` und `ID` initialisiert; er baut die Verbindung zum entfernten Server auf. Bei Erfolg quittiert er mit `INITO`.
2. Sobald der AX-Socket `IN` ein Ereignis auf `IN.E1` liefert, wird der gleichzeitig an `IN.D1` anliegende BOOL-Wert in das interne **E_D_FF**-Flipflop übernommen (`CLK` bzw. `D`).
3. Das Flipflop hält den Wert stabil an seinem Ausgang `Q` und erzeugt bei einer Wertänderung das Ereignis `EO`.
4. `EO` löst das Sendeereignis `REQ` des internen `CLIENT_1_0`-Blocks aus; der gepufferte Wert `Q` wird über die Datenverbindung als `SD_1` übergeben und per OPC-UA-Write an den Server gesendet.
5. Nach erfolgreichem Senden bestätigt der `CLIENT_1_0`-Block mit `CNF`, das nach außen geführt wird.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der zu sendende BOOL-Wert wird über ein internes `iec61499::events::E_D_FF` gepuffert. Dadurch wird verhindert, dass ein sich während der Verarbeitung ändernder Eingangswert den Sendevorgang stört.
- **Remote-Write statt lokalem Publish**: `CLIENT_1_0` adressiert einen entfernten OPC-UA-Server direkt (ACTION=WRITE mit vollständigem ENDPOINT in der `ID`), im Gegensatz zu `PUBLISH_1`/`SUBSCRIBE_1`, die nur lokal auf demselben FORTE-Knoten funktionieren.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `CLIENT_1_0` (inkl. `REQ`/`SD_1`) wird nach innen verlegt; nach außen ist nur noch die AX-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Die Verbindung zum entfernten Server ist aufgebaut, der Block ist bereit zu senden.
3. **Sendeaktiv**: Ein am AX-Socket eintreffendes Ereignis puffert den Wert und veranlasst den Remote-Write über den internen `CLIENT_1_0`-Block.

Anwendungsszenarien

- **Fernsteuerung anderer Module**: Ein VT-Tastendruck auf einem Modul mit eigener ISOBUS/VT-Verbindung löst einen OPC-UA-Write auf ein anderes, entferntes Steuerungsmodul aus (z. B. STG1 schaltet einen Ausgang auf STG2–STG5).
- **Netzwerkkopplung in AX-Anwendungen**: Senden von Bool-Signalen (Schaltbefehle) aus der Adapter-basierten Logik direkt an einen entfernten OPC-UA-Server.
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Fernsteuerung in Bibliotheken, die durchgängig auf AX-Adapter setzen.

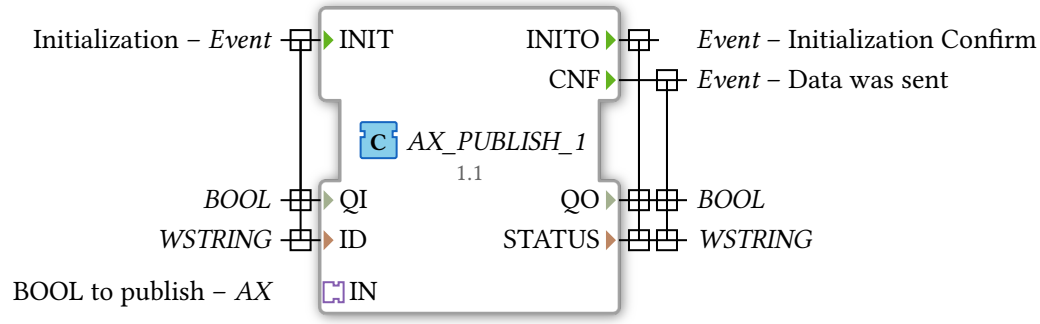
Vergleich mit ähnlichen Bausteinen

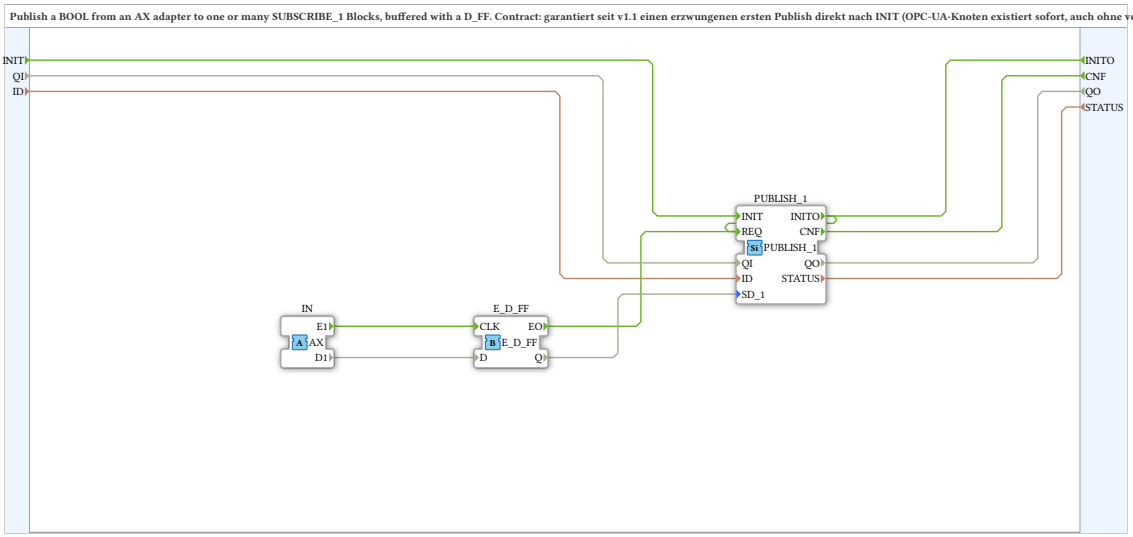
- **CLIENT_1_0**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`REQ`, `SD_1`, …). **AX_CLIENT_1_0** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop und stellt nur noch den AX-Adapter bereit.
- **AX_PUBLISH_1**: Strukturell identisch aufgebaut (gleiche Ereignisse/Variablen/Adapter), kapselt aber `PUBLISH_1` statt `CLIENT_1_0` – veröffentlicht also nur lokal, statt aktiv auf einen entfernten Knoten zu schreiben.

Fazit

AX_CLIENT_1_0 verbindet die verbindungsorientierte Remote-Write-Kommunikation des Standardbausteins `CLIENT_1_0` mit der Adapter-basierten Bool-Verarbeitung. Die interne D-Flipflop-Pufferung stabilisiert den Sendewert; die reduzierte AX-Schnittstelle macht den Baustein ideal, um in durchgängig adapterbasierten IEC 61499-Anwendungen einen entfernten OPC-UA-Server direkt zu schreiben.

AX_PUBLISH_1





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AX_PUBLISH_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `PUBLISH_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AX-Adapter** abbildet. Ein am Adapter-Socket `IN` anliegender **BOOL**-Wert wird über ein internes D-Flipflop (E_D_FF) gepuffert und anschließend über den konfigurierten Publish-Kanal an alle zugehörigen `SUBSCRIBE_1`-Empfänger gesendet.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`
- **CNF** (Event): Bestätigung, dass die Daten gesendet wurden, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Publish-Dienst
- **ID** (WSTRING): Identifikator des Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::AX	Socket (Eingang)	Zu veröffentlichender BOOL-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `PUBLISH_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Sobald der AX-Socket `IN` ein Ereignis auf `IN.E1` liefert, wird der gleichzeitig an `IN.D1` anliegende BOOL-Wert in das interne **E_D_FF**-Flipflop übernommen (`CLK` bzw. `D`).
3. Das Flipflop hält den Wert stabil an seinem Ausgang `Q` und erzeugt bei einer Wertänderung das Ereignis `EO`.
4. `EO` löst das Sendeereignis `REQ` des internen `PUBLISH_1`-Blocks aus; der gepufferte Wert `Q` wird über die Datenverbindung als `SD_1` übergeben und auf dem konfigurierten Kanal veröffentlicht.
5. Nach erfolgreichem Senden bestätigt der `PUBLISH_1`-Block mit `CNF`, das nach außen geführt wird.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der zu sendende BOOL-Wert wird über ein internes `iec61499::events::E_D_FF` gepuffert. Dadurch wird verhindert, dass ein sich während der Verarbeitung ändernder Eingangswert den Sendevorgang stört.
- **Ereignisbasierte Entkopplung**: Durch das echte ereignisbasierte `E_D_FF` werden unmittelbare Rückkopplungsschleifen zwischen `PUBLISH_1` und `SUBSCRIBE_1` (z. B. auf demselben OPC-UA-Knoten) entkoppelt – der saubere Fix gegenüber dem Interim-Fix mit `AX_D_FF`.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von `PUBLISH_1` (inkl. `REQ`/`SD_1`) wird nach innen verlegt; nach außen ist nur noch die AX-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das `INIT`-Ereignis.
2. **Initialisiert**: Der Block ist bereit, Daten über den konfigurierten Kanal zu veröffentlichen.
3. **Sendeaktiv**: Ein am AX-Socket eintreffendes Ereignis puffert den Wert und veranlasst das Senden über den internen `PUBLISH_1`-Block.

Anwendungsszenarien

- **Netzwerkkopplung in AX-Anwendungen**: Senden von Bool-Signalen (z. B. Schaltbefehle oder Statusbits) aus der Adapter-basierten Logik über Publish/Subscribe an entfernte Empfänger.
- **Event-Storm-Vermeidung**: Gemeinsam mit `AX_SUBSCRIBE_1` werden Rückkopplungen zwischen eigenen Veröffentlichungen und dem eigenen Abonnement entkoppelt.
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AX-Adapter setzen.

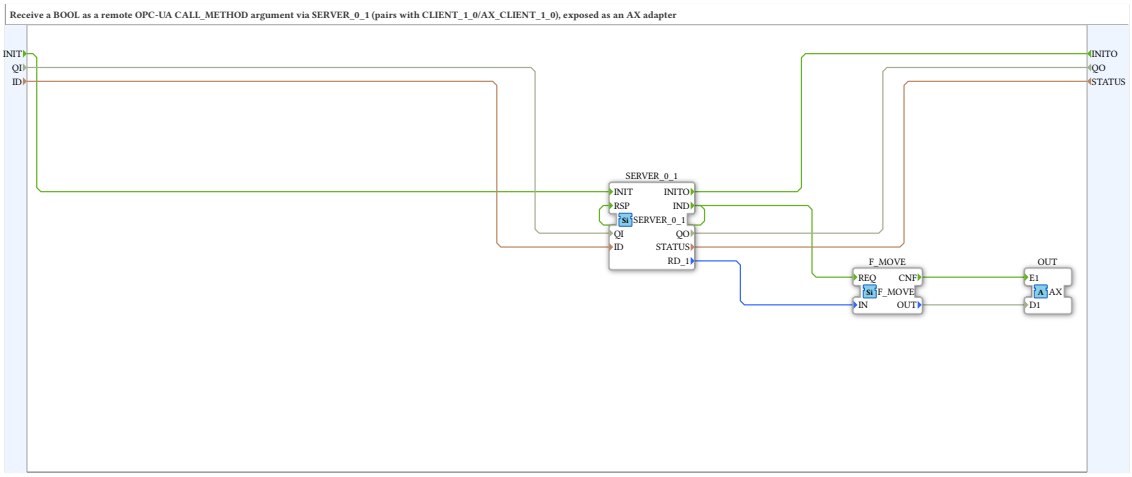
Vergleich mit ähnlichen Bausteinen

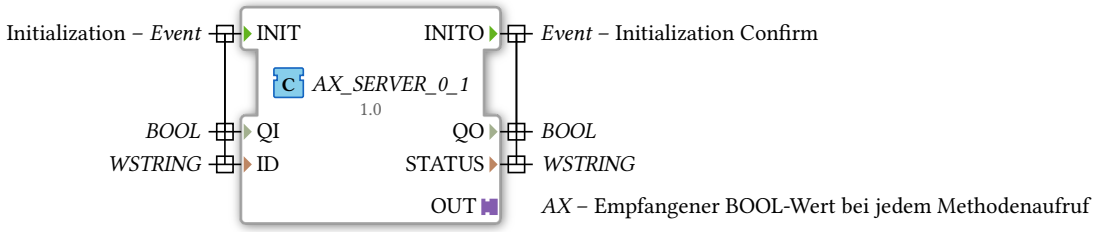
- **PUBLISH_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (`REQ`, `SD_1`, …). **AX_PUBLISH_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop und stellt nur noch den AX-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen Bool-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AX_PUBLISH_1** vereint Puffern und Veröffentlichen in einem Baustein.

Fazit

AX_PUBLISH_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins `PUBLISH_1` mit der Adapter-basierten Bool-Verarbeitung. Die interne D-Flipflop-Pufferung stabilisiert den Sendewert und entkoppelt Rückkopplungsschleifen; die reduzierte AX-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

AX_SERVER_0_1





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AX_SERVER_0_1** ist ein Composite-Funktionsblock aus dem Paket `adapter::net`. Er kapselt den netzwerkbasierten Server-Baustein `iec61499::net::SERVER_0_1` und stellt empfangene OPC-UA-Methodenaufrufe (`CALL_METHOD`) für **BOOL**-Argumente als unidirektionalen **AX-Adapter** (`OUT`) bereit. Er bildet das Empfangs-Gegenstück zu **AX_CLIENT_1_0** (bzw. `CLIENT_1_0`).

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID` (leitet an `SERVER_0_1.INIT` weiter)

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung (`SERVER_0_1.INITO`), verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, schaltet den Server aktiv (TRUE) bzw. inaktiv (FALSE)
- **ID** (WSTRING): Server-Identifikator (OPC-UA-Serveradresse und Methodenspezifikation)

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Server-Betriebsstatus
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
OUT	adapter::types::unidirectional::AX	Plug (Ausgang)	Empfangener BOOL-Wert bei jedem Methodenaufruf

Funktionsweise

1. Ein `INIT`-Ereignis an `AX_SERVER_0_1` wird an den internen `SERVER_0_1.INIT`-Eingang weitergeleitet. Der Server startet den OPC-UA-Endpunkt und quittiert bei Erfolg mit `INITO`.

2. Bei einem eintreffenden Methodenaufruf von einem entfernten Client (`CLIENT_1_0`/`AX_CLIENT_1_0`) erzeugt `SERVER_0_1` das Indikations-Ereignis `IND` und stellt den empfangenen BOOL-Wert an `RD_1` bereit.
3. Zur Vermeidung von Antwort-Latenz-Sperren (RSP-Latenz-Bug) ist `SERVER_0_1.IND` im internen Netzwerk direkt auf `SERVER_0_1.RSP` zurückverdrahtet. Der Server quittiert die Methode dadurch sofort ohne Verzögerung.
4. Gleichzeitig leitet `IND` den empfangenen Wert über `F_MOVE` (Datentyp `BOOL`) an den AX-Plug `OUT.D1` weiter und löst das Ausgangs-Ereignis `OUT.E1` aus.

Technische Besonderheiten

- **Direkte RSP-Rückverdrahtung**: Um 4-Sekunden-Server-Sperren durch verzögerte Antworten zu vermeiden, wird `SERVER_0_1.IND` intern unmittelbar auf `SERVER_0_1.RSP` zurückgekoppelt.
- **Nahtlose Adapter-Integration**: Der empfangene Rohwert `RD_1` von `SERVER_0_1` wird über ein `F_MOVE` (BOOL) direkt auf die AX-Adapter-Struktur (`OUT.D1` / `OUT.E1`) abgebildet.
- **Empfangs-Gegenstück zu `AX_CLIENT_1_0`**: In IEC 61499 pairt ein `CLIENT_1_0`-Baustein („Connect to a SERVER_0_1 Block“) mit einem `SERVER_0_1`-Baustein.

Zustandsübersicht

1. **Nicht initialisiert**: Der Server ist inaktiv und wartet auf das `INIT`-Ereignis.
2. **Bereit (Listening)**: Der OPC-UA-Server ist aktiv und wartet auf Methodenaufrufe von entfernten Clients.
3. **Empfang (Indikation)**: Ein Methodenaufruf trifft ein; der BOOL-Wert wird an den `OUT`-Adapter übergeben und die Quittierung sofort gesendet.

Anwendungsszenarien

- **Empfang von Schaltbefehlen**: Empfangen von Schaltsignalen (z. B. Start/Stopp, Freigaben) von einem entfernten Steuerungsmodul via OPC UA `CALL_METHOD`.
- **Modulare Server-Architekturen**: Bereitstellung von OPC-UA-Methoden-Endpunkten für Bool-Signale in adapterbasierten Anwendungen.

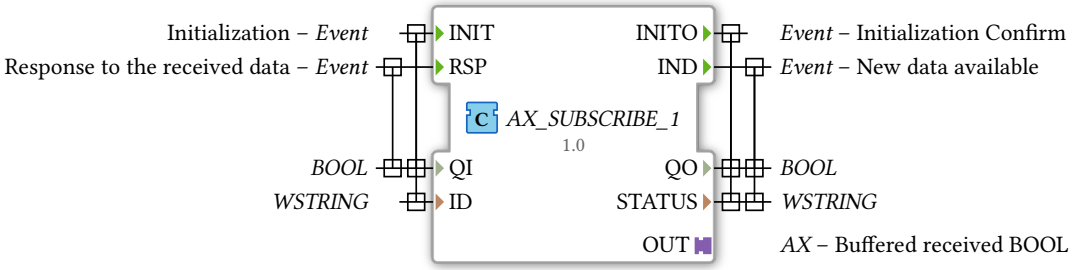
Vergleich mit ähnlichen Bausteinen

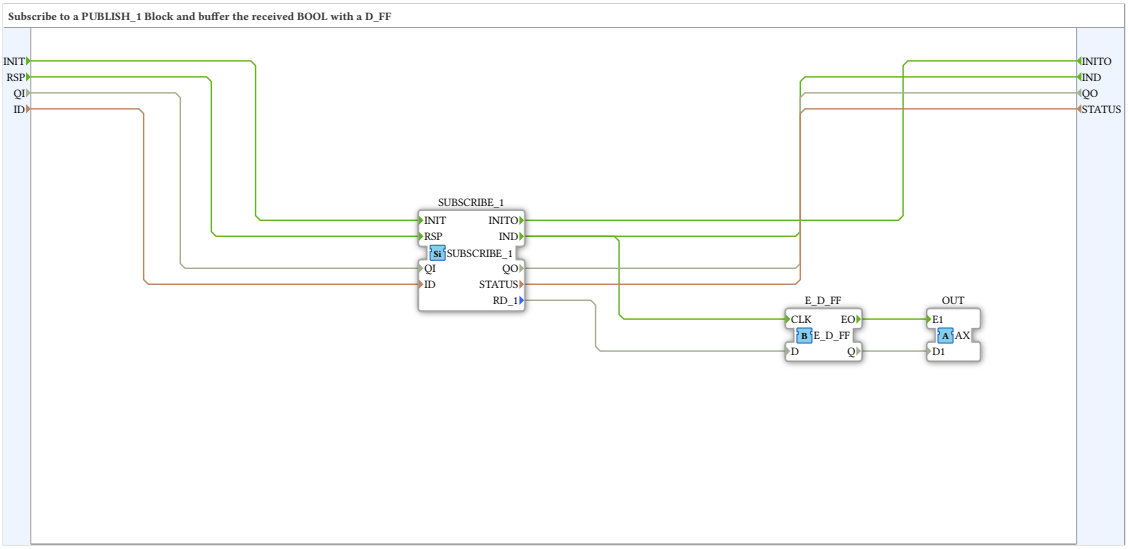
- **AR_SERVER_0_1**: Empfängt REAL-Werte über einen AR-Adapter.
- **ATM_SERVER_0_1**: Empfängt TIME-Werte über einen ATM-Adapter.
- **AX_SUBSCRIBE_1**: Kapselt `SUBSCRIBE_1` für lokales Publish/Subscribe statt verbindungsorientierter Methodenaufrufe.

Fazit

AX_SERVER_0_1 stellt empfangene OPC-UA-Methodenaufrufe für BOOL-Werte als sauberen AX-Adapter bereit und verhindert durch direkte RSP-Rückkopplung Server-Lockups.

AX_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AX_SUBSCRIBE_1** ist ein Composite-Funktionsblock, der den netzwerkbasierten `SUBSCRIBE_1`-Funktionsblock aus der IEC 61499-Standardbibliothek kapselt und dessen Schnittstelle auf einen unidirektionalen **AX-Adapter** abbildet. Er abonniert einen über die `ID` konfigurierten Publish-Kanal und stellt den empfangenen **BOOL**-Wert gepuffert über einen D-Flipflop (E_D_FF) am Adapter-Plug `OUT` bereit. Damit lassen sich Netzkommunikation und Bool-Signalverarbeitung in der AX-Adapter-Welt sauber verbinden.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsereignis, verbunden mit `QI` und `ID`
- **RSP** (Event): Response-Ereignis, verbunden mit `QI`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung, verbunden mit `QO` und `STATUS`

Daten-Eingänge

- **QI** (BOOL): Qualifier-Eingang, aktiviert den Abonnement-Dienst
- **ID** (WSTRING): Identifikator des zu abonnierenden Publish-Kanals

Daten-Ausgänge

- **QO** (BOOL): Qualifier-Ausgang, Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Adapter	Typ	Richtung	Beschreibung
OUT	adapter::types::unidirectional::AX	Plug (Ausgang)	Gepufferter empfangener BOOL-Wert

Funktionsweise

1. Über das `INIT`-Ereignis wird der interne `SUBSCRIBE_1`-Block mit `QI` und `ID` initialisiert. Bei Erfolg quittiert er mit `INITO`.
2. Der interne `SUBSCRIBE_1`-Block überwacht den konfigurierten Kanal. Sobald der zugehörige `PUBLISH_1`-Block Daten veröffentlicht, erzeugt er das Indikationsereignis `IND` und legt den empfangenen Wert an `RD_1` an.

3. IND taktet den internen **E_D_FF**-Flipflop (CLK), der den Wert von RD_1 an seinem Dateneingang D übernimmt.
4. Der gespeicherte Wert erscheint stabil am Ausgang Q und wird über die Datenverbindung auf OUT.D1 des AX-Plugs gelegt.
5. Das Ereignis EO des Flipflops wird auf den Ereignis-Ausgang OUT.E1 gelegt, sodass nachgeschaltete Bausteine über die neue Information benachrichtigt werden.

Technische Besonderheiten

- **Pufferung mit D-Flipflop**: Der empfangene BOOL-Wert wird über ein internes iec61499::events::E_D_FF gepuffert und bleibt stabil am Ausgang, bis der nächste Wert empfangen wird. EO feuert nur bei einer Änderung von Q.
- **RSP ist ein No-Op bei Subscribern**: Laut Quellcode-Analyse von 4diac-forte (CCommFB::sendData()) führt das RSP-Ereignis bei SUBSCRIBE_1 (Service-Typ Subscriber) keine Aktion aus – der Zweig != e_Subscriber wird übersprungen. Das Event-Input-Paar INIT/RSP wird nur für die Quittierung durchgeführt und ist für die Quittierung im Subscriber-Fall funktionslos. Es wird dennoch nach außen geführt, damit ein konfigurierbarer Anschluss besteht.
- **Kapselung**: Die ursprüngliche Ereignis-/Daten-Schnittstelle von SUBSCRIBE_1 (inkl. IND/RD_1) wird nach innen verlegt; nach außen ist nur noch die AX-Adapter-Schnittstelle sichtbar.

Zustandsübersicht

1. **Nicht initialisiert**: Der Block wartet auf das INIT-Ereignis.
2. **Initialisiert**: Das Abonnement ist aktiv; der Block wartet auf Daten vom Publisher.
3. **Datenempfang**: Bei IND wird der empfangene BOOL-Wert in das interne D-Flipflop übernommen und über OUT ausgegeben.

Anwendungsszenarien

- **Netzwerkkopplung in AX-Anwendungen**: Empfang von Bool-Signalen (z. B. Schaltbefehle oder Statusbits) über Publish/Subscribe und Weiterleitung über AX-Adapter an die Adapter-basierte Logik.
- **Event-Storm-Vermeidung**: Gemeinsam mit AX_PUBLISH_1 und einem echten ereignisbasierten E_D_FF lässt sich die Rückkopplung zwischen SUBSCRIBE_1 und PUBLISH_1 entkoppeln (sauberer Fix gegenüber dem Interim-Fix mit AX_D_FF).
- **Modulare Steuerungsarchitekturen**: Einbindung von verteilter Kommunikation in Bibliotheken, die durchgängig auf AX-Adapter setzen.

Vergleich mit ähnlichen Bausteinen

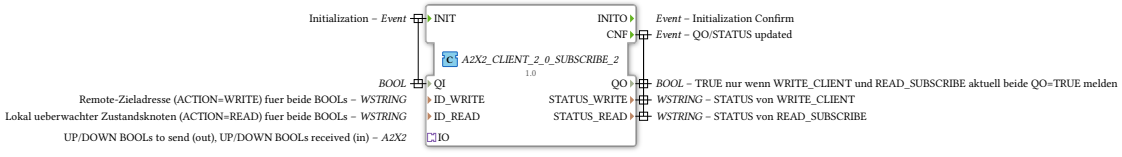
- **SUBSCRIBE_1**: Der reine Standardbaustein exponiert seine Schnittstelle als Events/Variablen (IND, RD_1, …). **AX_SUBSCRIBE_1** kapselt diesen Baustein, puffert den Wert mit einem D-Flipflop und stellt nur noch den AX-Adapter bereit – dadurch ist die Schnittstelle erheblich kompakter und in Adapter-Netzwerke integrierbar.
- **AX_D_FF**: Puffert einen Bool-Wert auf Adapter-Ebene, kennt aber keine Netzwerkfunktionalität. **AX_SUBSCRIBE_1** vereint Subscribing und Pufferung in einem Baustein.

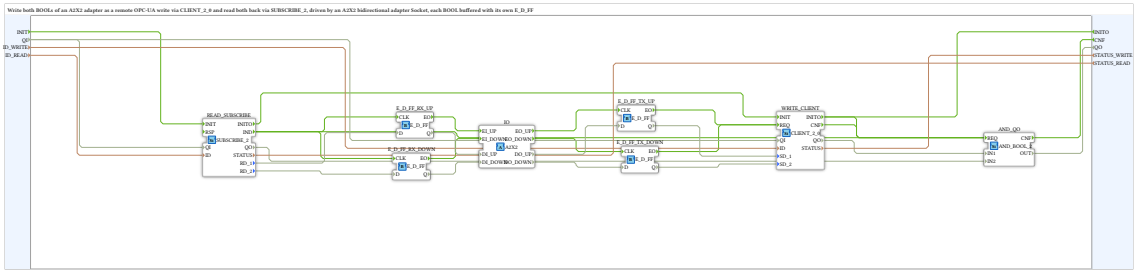
Fazit

AX_SUBSCRIBE_1 verbindet die Publish/Subscribe-Kommunikation des Standardbausteins SUBSCRIBE_1 mit der Adapter-basierten Bool-Verarbeitung. Die interne D-Flipflop-Pufferung hält den empfangenen Wert stabil, bis neue Daten eintreffen, und die reduzierte AX-Schnittstelle macht den Baustein ideal für durchgängig adapterbasierte IEC 61499-Anwendungen.

BOOL

A2X2_CLIENT_2_0_SUBSCRIBE_2





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `A2X2_CLIENT_2_0_SUBSCRIBE_2` realisiert eine bidirektionale Datenübertragung über OPC-UA zwischen einem A2X2-Adapter und zwei externen OPC-UA-Knoten. Er kombiniert einen OPC-UA-Client (`CLIENT_2_0`) zum Schreiben von zwei Bool-Werten mit einem OPC-UA-Subscriber (`SUBSCRIBE_2`) zum Lesen von zwei Bool-Werten. Die Werte werden über `E_D_FF`-Flipflops gepuffert, um eine entkoppelte und zuverlässige Kommunikation zu gewährleisten. Der Baustein eignet sich insbesondere für Anwendungen, bei denen ein Steuerungssystem mit einer OPC-UA-basierten Leitebene kommunizieren muss und dabei sowohl Schreib- als auch Leseoperationen auf einfache Weise gebündelt werden sollen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
`INIT`	EInit	Initialisierungsereignis; startet die Initialisierung des Subscribers und anschließend des Clients.

Ereignis-Ausgänge

Name	Typ	Kommentar
`INITO`	EInit	Bestätigung der erfolgreichen Initialisierung beider internen Bausteine.
`CNF`	Event	Wird ausgelöst, wenn sowohl der Write-Client als auch der Read-Subscriber eine Bestätigung liefern (QO = TRUE).

Daten-Eingänge

Name	Typ	Kommentar
`QI`	BOOL	Aktiviert die Kommunikation (TRUE = aktiv). Wird an beide internen Bausteine weitergegeben.
`ID_WRITE`	WSTRING	Remote-Zieladresse (ACTION=WRITE) für das Schreiben der beiden BOOL-Werte.
`ID_READ`	WSTRING	Lokal überwachter Zustandsknoten (ACTION=READ) für das Lesen der beiden BOOL-Werte.

Daten-Ausgänge

Name	Typ	Kommentar
`QO`	BOOL	TRUE nur wenn sowohl der Write-Client als auch der Read-Subscriber aktuell `QO = TRUE` melden.
`STATUS_WRITE`	WSTRING	Statusmeldung des internen `CLIENT_2_0`-Bausteins.
`STATUS_READ`	WSTRING	Statusmeldung des internen `SUBSCRIBE_2`-Bausteins.

Adapter

Name	Typ	Kommentar
IO	adapter::types::bidirectional::A2X2 (Socket)	Bidirektionaler A2X2-Adapter: vier Datenleitungen (DO_UP, DO_DOWN, DI_UP, DI_DOWN) und vier Ereignisleitungen (EI_UP, EI_DOWN, EO_UP, EO_DOWN) für den Austausch von Bool-Werten.

Funktionsweise

Der Funktionsblock verbindet einen OPC-UA-Client und einen OPC-UA-Subscriber mit einem A2X2-Adapter. Die internen Abläufe lassen sich in zwei getrennte Datenpfade unterteilen:

Schreibpfad (TX):

- Die über den Adapter eingehenden Datenwerte IO.DO_UP und IO.DO_DOWN werden durch die Ereignisse IO.EO_UP und IO.EO_DOWN in die jeweiligen E_D_FF_TX_*-Flipflops übernommen.
- Die Ausgänge dieser Flipflops (Q) werden an die Daten-Eingänge SD_1 und SD_2 des Clients (WRITE_CLIENT) gelegt.
- Jedes eingehende Ereignis am Adapter triggert den Client-Baustein (REQ), der die aktuellen Werte an die konfigurierte OPC-UA-Adresse (ID_WRITE) sendet.

Lesepfad (RX):

- Der Subscriber (READ_SUBSCRIBE) empfängt kontinuierlich die beiden Bool-Werte von der OPC-UA-Adresse (ID_READ) über seine Daten-Ausgänge RD_1 und RD_2.
- Diese Werte werden bei jedem empfangenen Ereignis (IND) in die E_D_FF_RX_*-Flipflops übernommen.
- Die Ausgänge dieser Flipflops speisen die Adapter-Datenausgänge IO.DI_UP und IO.DI_DOWN und werden über die Ereignisse IO.EI_UP und IO.EI_DOWN an den Adapter signalisiert.

Initialisierung:

- Der INIT-Eingang initialisiert zuerst den Subscriber (READ_SUBSCRIBE.INIT).
- Nach erfolgreicher Initialisierung des Subscribers (Ereignis INITO) wird der Client initialisiert (WRITE_CLIENT.INIT).
- Der Client bestätigt die Initialisierung mit INITO, woraufhin der Baustein das Ereignis INITO nach außen abgibt.

Fehler- und Statusüberwachung:

- Die Ereignisse WRITE_CLIENT.CNF und READ_SUBSCRIBE.IND werden gemeinsam auf den Eingang REQ des UND-Glieds AND_QO geführt.
- Das UND-Glied setzt QO nur dann auf TRUE, wenn sowohl der Client als auch der Subscriber ihren Betriebszustand als aktiv melden (beide QO = TRUE).
- Der Ausgang des UND-Glieds erzeugt das Ereignis CNF, um den Aufrufer über den aktuellen Zustand zu informieren.
- Die Statuswerte des Clients und des Subscribers werden direkt auf STATUS_WRITE und STATUS_READ ausgegeben.

Technische Besonderheiten

- **Bidirektionalität:** Der Adapter A2X2 unterstützt sowohl das Senden als auch das Empfangen von Bool-Werten, was eine symmetrische Kommunikation ermöglicht.
- **OPC-UA-Integration:** Der Baustein nutzt die standardisierten 4diac-Bausteine CLIENT_2_0 und SUBSCRIBE_2 für die OPC-UA-Kommunikation. ID_WRITE und ID_READ müssen als vollständige OPC-UA-Knotenadressen (z.B. ns=2;s=MyNode) angegeben werden.
- **Pufferung:** Die E_D_FF-Flipflops entkoppeln die asynchronen Ereignisse der Adapter- und OPC-UA-Seite. Dadurch werden kurze Signalimpulse zuverlässig zwischengespeichert und spätere Datenänderungen korrekt weitergegeben.
- **Kombinierte Statusauswertung:** QO ist nur aktiv, wenn beide Kommunikationspfade (Senden und Empfangen) gleichzeitig in Ordnung sind. Dies ermöglicht eine einfache Überwachung der Gesamtverbindung.

- **Reihenfolge der Initialisierung:** Die Initialisierung erfolgt strikt sequenziell (erst Subscriber, dann Client). Dies stellt sicher, dass die Verbindung zum OPC-UA-Server erst nach erfolgreichem Abonnement aufgebaut wird.

Zustandsübersicht

Der Funktionsblock besitzt keine expliziten Zustandsautomaten, da er vollständig aus konfigurierten Standardbausteinen besteht. Dennoch lassen sich logische Betriebszustände ableiten:

Zustand	Beschreibung
Initialisierung	Nach einem `INIT`-Impuls wird der Subscriber initialisiert, danach der Client. Nach erfolgreichem Abschluss wird `INITO` ausgegeben.
Aktiv / Normalbetrieb	`QI = TRUE`, Client und Subscriber sind erfolgreich verbunden. `QO` wird TRUE, wenn beide Bausteine ihre Bereitschaft signalisieren.
Fehler / Nicht bereit	Mindestens einer der internen Bausteine meldet `QO = FALSE` oder einen Status ungleich `"Idle"`. `QO` wird FALSE, `STATUS_WRITE` bzw. `STATUS_READ` enthalten die Fehlerdetails.
Deaktiviert	Bei `QI = FALSE` werden beide internen Bausteine deaktiviert; es werden keine Daten mehr übertragen.

Anwendungsszenarien

- **Fernsteuerung von Binärsignalen:** Ein Anlagensteuerungs-PLC sendet zwei Schaltzustände (z.B. Ventil auf/zu) und empfängt zwei Rückmeldungen über eine OPC-UA-Verbindung. Der Baustein bündelt beide Richtungen in einer einzigen Komponente.
- **Überwachung von Feldgeräten:** Ein Feldgerät mit A2X2-Schnittstelle wird über OPC-UA angebunden. Die vom Gerät gelieferten Bool-Werte werden per Subscription gelesen und auf der Steuerung verfügbar gemacht; umgekehrt können Sollwerte an das Gerät gesendet werden.
- **Integration in Leitsysteme:** Da `CLIENT_2_0` und `SUBSCRIBE_2` standardisierte 4diac-Bausteine sind, lässt sich der FB nahtlos in bestehende OPC-UA-basierte Automatisierungsarchitekturen integrieren.

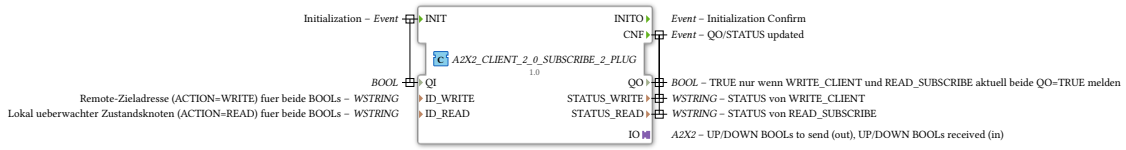
Vergleich mit ähnlichen Bausteinen

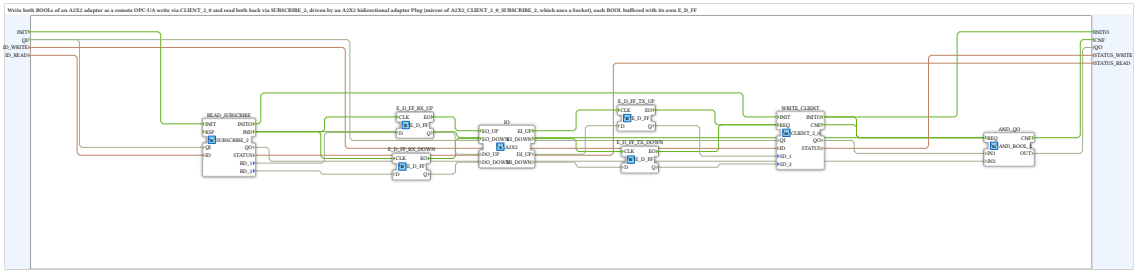
Ähnliche Bausteine, die nur eine Kommunikationsrichtung abdecken, sind z.B. `CLIENT_2_0` (nur Schreiben) oder `SUBSCRIBE_2` (nur Lesen). Der Kombinationsbaustein `A2X2_CLIENT_2_0_SUBSCRIBE_2` vereint beide Funktionen und bietet zusätzlich eine konfigurierte Pufferung sowie eine aggregierte Statusanzeige. Gegenüber einer manuellen Verschaltung der Standardbausteine spart er Entwicklungszeit und reduziert die Fehleranfälligkeit durch eine klar definierte Schnittstelle.

Fazit

Der Funktionsblock `A2X2_CLIENT_2_0_SUBSCRIBE_2` ist eine wiederverwendbare, kompakte Lösung für die bidirektionale OPC-UA-Kommunikation mit einem A2X2-Adapter. Durch die Kombination von Client und Subscriber, die Pufferung der Datenpfade und die gemeinsame Statusüberwachung bietet er eine robuste und leicht integrierbare Komponente für Automatisierungsprojekte. Die strikte Trennung von Schreib- und Lesepfad sowie die klare Initialisierungsreihenfolge machen ihn sowohl für einfache Steuerungsaufgaben als auch für komplexere Leitsystemanbindungen geeignet.

A2X2_CLIENT_2_0_SUBSCRIBE_2_PLUG





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `A2X2_CLIENT_2_0_SUBSCRIBE_2_PLUG` dient als bidirektionale Schnittstelle zwischen einem A2X2-Adapter (Plug) und einem OPC-UA-Server. Er schreibt die beiden BOOL-Signale UP und DOWN des Adapters über einen `CLIENT_2_0` als Remote-Write an eine definierte OPC-UA-Adresse und liest gleichzeitig die gleichen Signale über einen `SUBSCRIBE_2` zurück. Der FB puffert jeden BOOL-Wert mit einem eigenen `E_D_FF`-Baustein, um einen sauberen Signalwechsel zu gewährleisten. Er ist das Pendant zum FB `A2X2_CLIENT_2_0_SUBSCRIBE_2`, der einen Socket verwendet – hier wird die Kommunikation über einen Adapter-Plug realisiert.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar
`INIT`	EInit	Initialisierung des FB, ausgelöst durch `QI`.

Ereignis-Ausgänge

Ereignis	Typ	Kommentar
`INITO`	EInit	Bestätigung der erfolgreichen Initialisierung.
`CNF`	Event	Wird ausgelöst, wenn `QO`, `STATUS_WRITE` und `STATUS_READ` aktualisiert wurden.

Daten-Eingänge

Name	Typ	Kommentar
`QI`	BOOL	Steuert die Initialisierung. Bei `TRUE` wird der FB aktiviert.
`ID_WRITE`	WSTRING	OPC-UA-Knotenadresse für die Schreib-Operationen (ACTION=WRITE).
`ID_READ`	WSTRING	OPC-UA-Knotenadresse für die Lese-Operationen (ACTION=READ).

Daten-Ausgänge

Name	Typ	Kommentar
`QO`	BOOL	`TRUE`, wenn sowohl `WRITE_CLIENT.QO` als auch `READ_SUBSCRIBE.QO` aktiv sind.
`STATUS_WRITE`	WSTRING	Statusmeldung des `WRITE_CLIENT` (Schreibvorgang).
`STATUS_READ`	WSTRING	Statusmeldung des `READ_SUBSCRIBE` (Lesevorgang).

Adapter

Name	Typ	Kommentar
`IO`	A2X2	Bidirektionaler Adapter-Plug für die BOOL-Signale `UP` und `DOWN`.

Funktionsweise

Der FB besteht aus mehreren internen Funktionsbausteinen: `WRITE_CLIENT` (CLIENT_2_0), `READ_SUBSCRIBE` (SUBSCRIBE_2), `AND_QO` (Zwei-Eingang-UND) sowie vier `E_D_FF`-Bausteinen zur Pufferung der BOOL-Werte.

1. **Initialisierung:**
   Beim Anlegen des FB wird `INIT` ausgelöst. Über eine Sequenz wird zuerst `READ_SUBSCRIBE.INIT` und dann – nach dessen Bestätigung `INITO` – `WRITE_CLIENT.INIT` aktiviert. Nach erfolgreicher Initialisierung beider Netzwerkbausteine wird das Ereignis `INITO` des Gesamt-FB ausgegeben.
2. **Schreiben (TX):**
   Die Daten-Eingänge `DI_UP` und `DI_DOWN` des Adapters werden über `E_D_FF_TX_UP` bzw. `E_D_FF_TX_DOWN` gepuffert. Wenn sich ein Wert ändert (`EI_UP` oder `EI_DOWN` Ereignis), wird der neue Wert am Ausgang `Q` des jeweiligen `E_D_FF` an die Daten-Eingänge `SD_1` bzw. `SD_2` des `WRITE_CLIENT` übergeben und ein Schreibauftrag (`REQ`) ausgelöst.
3. **Lesen (RX):**
   Der `READ_SUBSCRIBE` empfängt kontinuierlich die Daten `RD_1` und `RD_2` vom OPC-UA-Server. Bei jeder Aktualisierung (`IND`) werden diese Werte in die `E_D_FF_RX_*` Bausteine übernommen. Erst wenn ein Flankenwechsel am `CLK`-Eingang auftritt, wird der jeweilige Ausgang `Q` aktualisiert und das Ergebnis an `DO_UP` bzw. `DO_DOWN` des Adapters ausgegeben (Ereignisse `EO_UP`/`EO_DOWN`).
4. **Zusammenführung von Qualität und Status:**
   Die Ausgänge `QO` der beiden Netzwerkbausteine (`WRITE_CLIENT` und `READ_SUBSCRIBE`) werden über den UND-Baustein `AND_QO` verknüpft. Das Ergebnis wird am Ausgang `QO` des FB bereitgestellt und das Ereignis `CNF` ausgelöst (zusammen mit den Statuswerten). Dadurch signalisiert der FB, dass sowohl der Schreib- als auch der Lesevorgang fehlerfrei funktionieren.

Technische Besonderheiten

- **Pufferung mit `E_D_FF`:**
  Jeder BOOL-Wert wird durch einen eigenen `E_D_FF` geführt. Dies verhindert, dass bei schnellen Signalwechseln Daten verloren gehen oder unerwünschte Mehrfach-Sendeaufträge entstehen.
- **Eigenständige Initialisierungskette:**
  Die Reihenfolge `READ_SUBSCRIBE` ⟶ `WRITE_CLIENT` stellt sicher, dass der Abonnent vor dem Client aktiviert wird, was für OPC-UA-Subscriptions notwendig ist.
- **Asynchrone Aktualisierung:**
  Da Lesen und Schreiben asynchron über OPC-UA erfolgen, wird `CNF` erst nach Abschluss beider Operationen (oder einer Aktualisierung vom Lesen) ausgelöst.
- **Plug statt Socket:**
  Im Gegensatz zum Schwestern-FB `A2X2_CLIENT_2_0_SUBSCRIBE_2`, der einen Socket verwendet, nutzt dieser FB einen `Plug`. Dadurch kann er direkt in ein übergeordnetes FB-Netzwerk eingebettet werden, das über einen kompatiblen Socket verfügt.

Zustandsübersicht

Da der FB keine expliziten Zustände in der XML definiert, lassen sich aus dem internen Netzwerk folgende Phasen ableiten:

- **Initialisierungsphase:** `QI = TRUE` ⟶ `INIT` ⟶ `READ_SUBSCRIBE.INIT` ⟶ `WRITE_CLIENT.INIT` ⟶ `INITO`
- **Betriebsphase:**
  - *Senden:* Bei Änderung an `DI_UP`/`DI_DOWN` wird gepuffert und ein Write ausgelöst.
  - *Empfangen:* Bei Änderung vom Server wird gepuffert und an den Adapter ausgegeben.
  - *Überwachung:* Die `QO`-Signale der Netzwerk-FBs werden über UND verknüpft.

Eine detaillierte Zustandsmaschine ist im FB selbst nicht sichtbar, die internen `E_D_FF` können jedoch in die Zustände „Set“ und „Reset“ wechseln, je nach Datenwert und Taktsignal.

Anwendungsszenarien

- **Fernsteuerung von Binärausgängen:**
  Der FB eignet sich, um zwei digitale Ausgänge eines (z. B. industriellen) Geräts über OPC-UA zu steuern und den aktuellen Zustand gleichzeitig in die Gegenrichtung zurückzulesen.

- **Steuerung mit Rückmeldung:**
  Wenn die Schreib- und Leseadresse auf denselben OPC-UA-Knoten zeigen, kann der FB als Spiegel für eine wechselseitige Kommunikation zwischen zwei Steuerungen dienen.
- **Integration in IEC 61499-Systeme:**
  Durch die Verwendung eines Adapter-Plugs ist der FB einfach in größere Applikationen einbindbar, die bidirektionale A2X2-Schnittstellen verwenden.

Vergleich mit ähnlichen Bausteinen

Der direkte Verwandte ist `A2X2_CLIENT_2_0_SUBSCRIBE_2`, der statt eines Plugs einen Socket verwendet. Der vorliegende FB (`_PLUG`) bietet die gleiche Funktionalität, setzt aber auf die Adapter-Mechanik von 4diac. Dadurch ist er flexibler in der Topologie, da er an beliebigen Stellen eines FB-Netzwerks über einen Socket eingebunden werden kann.

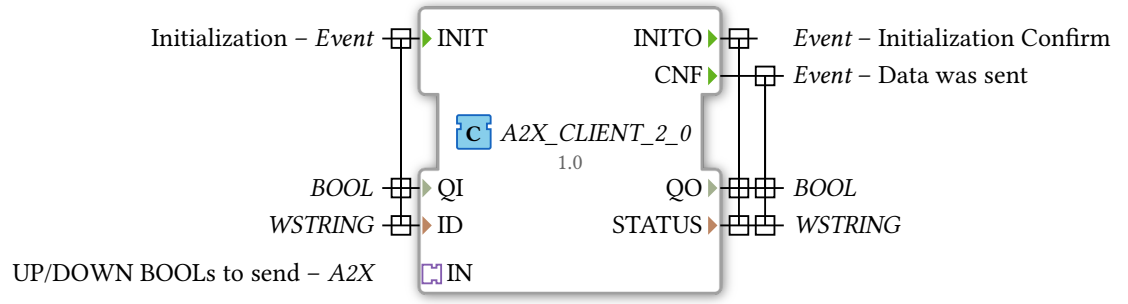
Weitere Alternativen könnten sein:

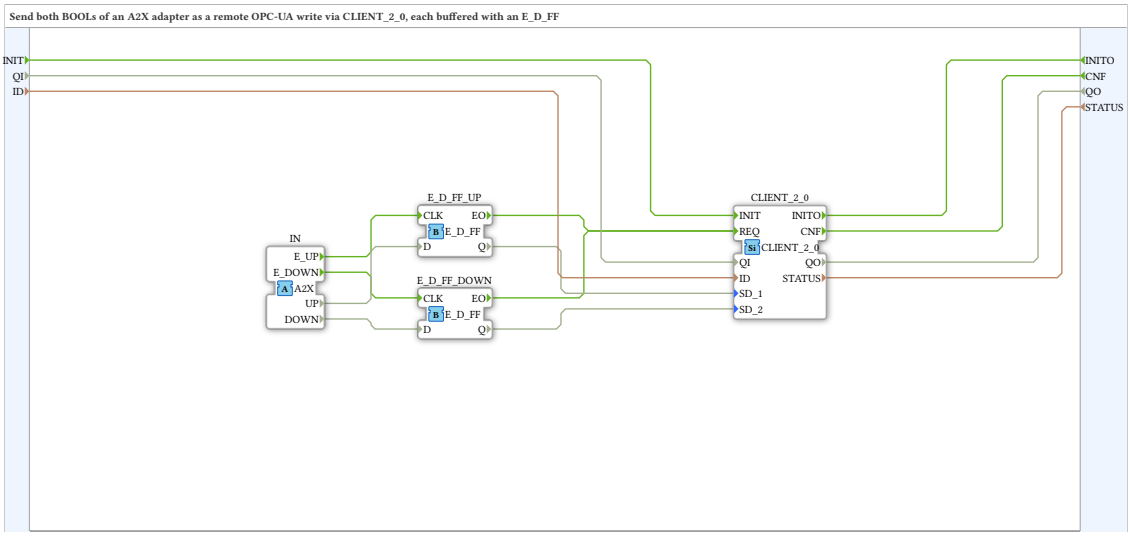
- Verwendung von zwei getrennten FBs für Schreiben und Lesen, was jedoch eine manuelle Synchronisation erfordert.
- Einsatz von `E_SR`-Bausteinen zur Pufferung, die jedoch weniger Flanken-orientiert arbeiten.

Fazit

Der `A2X2_CLIENT_2_0_SUBSCRIBE_2_PLUG` implementiert eine robuste bidirektionale OPC-UA-Anbindung für zwei BOOL-Signale, gekapselt in einen einzelnen, gut wiederverwendbaren Funktionsblock. Die Kombination aus `CLIENT_2_0` und `SUBSCRIBE_2` mit `E_D_FF`-Puffern gewährleistet eine zuverlässige Datenübertragung und eine klare Statusanzeige. Dank der Adapter-Schnittstelle lässt sich der FB direkt in bestehende 4diac-Applikationen integrieren und eignet sich besonders für Remote-Steuerungs- und Regelungsaufgaben in Automatisierungsumgebungen.

A2X_CLIENT_2_0





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **A2X_CLIENT_2_0** dient dazu, zwei binäre Signale eines A2X-Adapters (UP und DOWN) als OPC-UA-Write über einen CLIENT_2_0-Baustein an ein entferntes System zu übertragen. Die Signale werden dabei jeweils über einen E_D_FF-Flipflop zwischengespeichert, sodass auch bei kurzen Ereignissen ein zuverlässiges Senden gewährleistet ist. Der Baustein kapselt die komplette Kommunikationslogik und bietet eine einfache Schnittstelle für die Anbindung von binären Prozessdaten an ein OPC-UA-Netzwerk.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar
INIT	EInit	Initialisierung des Bausteins (initiiert Verbindungsaufbau)

Ereignis-Ausgänge

Ereignis	Typ	Kommentar
INITO	EInit	Bestätigung der erfolgreichen Initialisierung
CNF	Event	Bestätigung, dass die Daten gesendet wurden

Daten-Eingänge

Name	Typ	Kommentar
QI	BOOL	Steuersignal für die Aktivierung des Bausteins (Quality Indicator)
ID	WSTRING	Adress- oder Identifikationsinformation für die OPC-UA-Verbindung

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Qualitätssignal – zeigt den Betriebszustand des Bausteins an
STATUS	WSTRING	Statusmeldung oder Fehlercode der Kommunikation

Adapter

Socket	Typ	Kommentar
IN	adapter::types::unidirectional::A2X	Bietet die Eingänge UP und DOWN (BOOL) sowie die Ereignisse E_UP und E_DOWN

Funktionsweise

Der Baustein nutzt intern zwei E_D_FF-Flipflops (UP und DOWN) sowie einen CLIENT_2_0-Baustein für die OPC-UA-Kommunikation. Die Ereignisse `E_UP` und `E_DOWN` vom Adapter setzen jeweils das zugehörige Flipflop, wobei der aktuelle Wert von `UP` bzw. `DOWN` als Dateneingang dient. Die Ausgänge `Q` der Flipflops werden als Datenwerte `SD_1` und `SD_2` an den CLIENT_2_0 übergeben. Sobald eines der Flipflops einen neuen Wert erfasst hat, wird ein `REQ`-Ereignis an den CLIENT_2_0 ausgelöst, der daraufhin die beiden Werte als OPC-UA-Write an das konfigurierte Ziel sendet.

Die Initialisierung (`INIT`) wird direkt an den CLIENT_2_0 weitergeleitet. Nach erfolgreichem Verbindungsaufbau antwortet der CLIENT_2_0 mit `INITO`. Nach jedem Sendevorgang wird ein `CNF`-Ereignis emittiert. Die Ausgänge `QO` und `STATUS` spiegeln die entsprechenden Ausgänge des CLIENT_2_0 wider.

Technische Besonderheiten

- **Pufferung durch E_D_FF**: Jedes Binärsignal wird in einem flankengesteuerten Flipflop gespeichert. Dadurch wird verhindert, dass ein kurzes Ereignis verloren geht, falls der CLIENT_2_0 gerade nicht bereit ist.

- **Einheitliche Datenübertragung**: Die beiden Bool-Werte werden in einem einzigen OPC-UA-Write übertragen, was die Netzwerklast reduziert und die Konsistenz der Daten gewährleistet.
- **Direkte Durchreichung**: Die Status- und Qualitätssignale (`QO`, `STATUS`) werden unverändert vom CLIENT_2_0 übernommen, sodass der Anwender den Kommunikationsstatus genau verfolgen kann.
- **Ereignisgesteuerte Aktualisierung**: Es werden nur dann Daten gesendet, wenn sich der Wert eines der Binärsignale ändert – kein kontinuierliches Polling erforderlich.

Zustandsübersicht

Der Baustein besitzt keine explizit modellierten Zustände, das Verhalten wird durch die internen Komponenten bestimmt:

- **Initialisierungsphase**: Nach dem `INIT`-Ereignis versucht der CLIENT_2_0 eine Verbindung zum Zielsystem aufzubauen. Erfolgreicher Abschluss wird durch `INITO` signalisiert.
- **Bereitschaft**: Nach erfolgreicher Initialisierung wartet der Baustein auf Ereignisse an den Adaptereingängen.
- **Sendeoperation**: Tritt ein Ereignis `E_UP` oder `E_DOWN` auf, wird der aktuelle Wert übernommen und ein `REQ` an den CLIENT_2_0 gesendet. Nach Abschluss des Schreibvorgangs wird `CNF` ausgegeben.
- **Fehlerfall**: Wenn die Verbindung verloren geht oder der CLIENT_2_0 einen Fehler meldet, wird dies über `STATUS` und ggfs. `QO` wiedergegeben.

Anwendungsszenarien

- **Fernsteuerung**: Übertragung von zwei Schaltzuständen (z. B. Auf/Ab, Ein/Aus) von einer SPS an eine übergeordnete Leitstelle via OPC-UA.
- **Datenbrücke**: Anbindung eines A2X-Adapters an ein OPC-UA-basiertes Überwachungssystem, ohne dass separate Kommunikationsbausteine verdrahtet werden müssen.
- **Zuverlässige Ereignisübertragung**: Einsatz in Umgebungen, in denen kurze Impulse oder Flanken sicher erfasst und übertragen werden müssen, z. B. bei Zählern oder Positionssignalen.

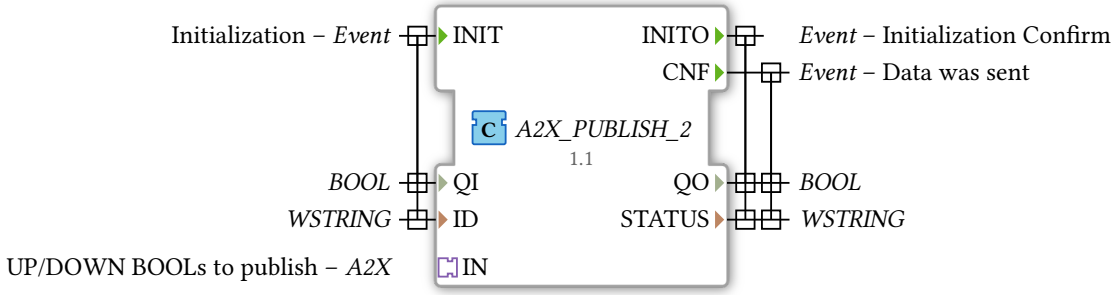
Vergleich mit ähnlichen Bausteinen

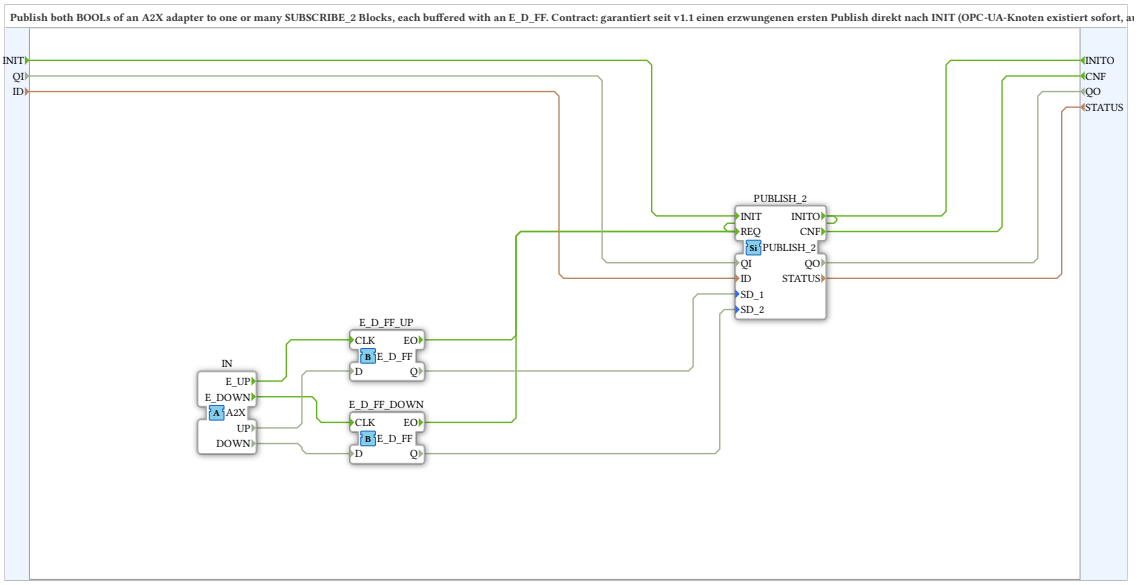
Im Gegensatz zu einem direkten Einsatz eines `CLIENT_2_0` ohne vorgeschaltete Pufferung bietet `A2X_CLIENT_2_0` eine integrierte Ereignisbehandlung. Ein einfacher `CLIENT_2_0` erfordert eine manuelle Zuordnung von Ereignissen und Datenwerten, während dieser Baustein die komplette Logik kapselt und dem Anwender eine kompakte, wiederverwendbare Schnittstelle bietet. Zudem werden durch die Flipflops Datenverluste durch zeitliche Verzögerungen zwischen Ereignis und Sendebereitschaft vermieden. Gegenüber einem generischen OPC-UA-Client zeichnet sich dieser Baustein durch die spezifische Anpassung an die A2X-Adapter-Semantik aus.

Fazit

Der Funktionsblock **A2X_CLIENT_2_0** stellt eine robuste und komfortable Lösung dar, um zwei binäre Signale über OPC-UA zu übertragen. Durch die eingebaute Pufferung und die ereignisgesteuerte Verarbeitung werden auch zeitkritische Signale zuverlässig versendet. Die klare Schnittstelle und die Wiederverwendbarkeit machen ihn zu einem nützlichen Baustein für Automatisierungs- und Visualisierungsanwendungen, die eine OPC-UA-Anbindung erfordern.

A2X_PUBLISH_2





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `A2X_PUBLISH_2` ist ein zusammengesetzter Baustein (Composite FB), der dazu dient, zwei boolesche Werte eines A2X-Adapters über ein Netzwerk zu publizieren. Er basiert auf dem Standard-Funktionsblock `PUBLISH_2` und ergänzt diesen um eine Adapter-Schnittstelle vom Typ `adapter::types::unidirectional::A2X`. Durch die Verwendung von flankengetriggerten Speichern (`E_D_FF`) werden Daten nur dann gesendet, wenn sich einer der beiden Werte ändert, wodurch unnötige Netzwerklast vermieden wird.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Datentyp	Beschreibung
`INIT`	`EInit`	Initialisierung des Bausteins. Parameter `QI` und `ID` werden übergeben und an den internen `PUBLISH_2` weitergeleitet.

Ereignis-Ausgänge

Ereignis	Datentyp	Beschreibung
`INITO`	`EInit`	Bestätigung der erfolgreichen Initialisierung. Wird ausgelöst, wenn der `PUBLISH_2` die Initialisierung abgeschlossen hat.
`CNF`	`Event`	Bestätigung, dass Daten erfolgreich gesendet wurden. Ausgelöst nach jedem Sendevorgang.

Daten-Eingänge

Name	Datentyp	Beschreibung
`QI`	`BOOL`	Aktivierungssignal für den Baustein. Bei `FALSE` wird keine Kommunikation durchgeführt.
`ID`	`WSTRING`	Kennung (z.B. IP-Adresse und Port) für den Publikationskanal. Wird an `PUBLISH_2` übergeben.

Daten-Ausgänge

Name	Datentyp	Beschreibung
QO	BOOL	Status der letzten Operation (siehe PUBLISH_2): TRUE bei Erfolg, FALSE bei Fehler.
STATUS	WSTRING	Detaillierte Fehlermeldung oder Statusinformation.

Adapter

Typ	Richtung	Name	Beschreibung
adapter::types::unidirectional::A2X	Socket	IN	Empfängt die beiden booleschen Werte UP und DOWN sowie die zugehörigen Ereignisse E_UP und E_DOWN von einer übergeordneten Anwendung.

Funktionsweise

Der Baustein verwendet intern drei Funktionsblöcke:

- E_D_FF_UP und E_D_FF_DOWN: flankengetriggerte D-Flip-Flops (Typ iec61499::events::E_D_FF), die jeweils einen der beiden Eingangswerte (IN.UP bzw. IN.DOWN) puffern und bei einem Ereignis (E_UP bzw. E_DOWN) einen Ausgangsimpuls erzeugen.
- PUBLISH_2: der eigentliche Netzwerk-Publisher, der die gepufferten Werte an ein Ziel sendet.

Ablauf:

1. **Initialisierung**: Über das Ereignis INIT werden QI und ID an den PUBLISH_2 übergeben und dessen Initialisierung gestartet. Nach erfolgreicher Initialisierung wird INITO ausgelöst.
2. **Datenänderung**: Wenn am Adapter IN entweder das Ereignis E_UP oder E_DOWN eintrifft, wird das entsprechende Flip-Flop mit dem aktuellen Wert (IN.UP bzw. IN.DOWN) übernommen. Dieses Flip-Flop erzeugt seinerseits ein Ereignis an seinem Ausgang EO, das mit dem REQ-Eingang des PUBLISH_2 verbunden ist.
3. **Senden**: Der PUBLISH_2 sendet dann die beiden gespeicherten Werte SD_1 (vom Flip-Flop UP) und SD_2 (vom Flip-Flop DOWN) als zusammengehörigen Datensatz über das Netzwerk. Nach erfolgreichem Senden wird das Ereignis CNF ausgelöst und die Statusausgänge QO und STATUS aktualisiert.

Hinweis: Die beiden Flip-Flops sind unabhängig voneinander. Wenn nur einer der beiden Werte geändert wird, wird trotzdem ein komplettes Datenpaket mit beiden aktuellen Werten gesendet.

Technische Besonderheiten

- Der Baustein ist ein **Composite FB** und kapselt die Logik, sodass der Anwender lediglich die Adapter-Schnittstelle und die Parameter QI/ID verwenden muss.
- Die Verwendung der **E_D_FF**-Bausteine sorgt dafür, dass Daten nur bei **Signalflanken** gesendet werden – es wird also keine kontinuierliche Übertragung durchgeführt, sondern nur bei Änderungen.
- Der Baustein ist für **unidirektionale** Kommunikation ausgelegt (nur Senden, kein Empfangen). Der Socket IN liefert die Daten, die publiziert werden.
- Der interne PUBLISH_2 basiert auf dem Standard der IEC 61499 und unterstützt verschiedene Netzwerkprotokolle (z.B. UDP, TCP – abhängig von der Implementierung).
- Die Software ist unter der **Eclipse Public License 2.0** lizenziert und wurde für die 4diac-IDE entwickelt.

Zustandsübersicht

Der Baustein hat keinen expliziten Zustandsautomaten, sondern nutzt die internen Zustände des PUBLISH_2. Exemplarisch:

- **Initialisierung läuft**: Nach dem INIT-Ereignis wartet der PUBLISH_2 auf die Bestätigung durch das INITO-Ereignis.
- **Bereit**: Nach der Initialisierung kann der Baustein Daten senden. Solange QI den Wert TRUE hat, werden eingehende E_UP/E_DOWN-Ereignisse verarbeitet.
- **Senden aktiv**: Während eines Sendevorgangs werden weitere Ereignisse gepuffert (durch die Flip-Flops), bis der aktuelle Vorgang abgeschlossen ist.

- **Fehlerzustand**: Bei Fehlern (z.B. Netzwerk nicht erreichbar) wird `STATUS` entsprechend gesetzt und `QO` auf `FALSE` gesetzt. Nach Behebung des Fehlers kann der Baustein über `INIT` erneut initialisiert werden.

Anwendungsszenarien

- **Maschinensteuerung**: Veröffentlichung von zwei Status-Bits (z.B. „Bewegung oben/unten“) an eine übergeordnete Leitebene, wobei nur bei Änderungen gesendet wird.
- **Gebäudeautomation**: Publizieren von Schaltzuständen (z.B. Licht an/aus) über ein Netzwerk an ein Visualisierungssystem.
- **Industrie 4.0 / IoT**: Integration in verteilte Steuerungssysteme, bei denen eine minimale Netzwerklast gewünscht ist.

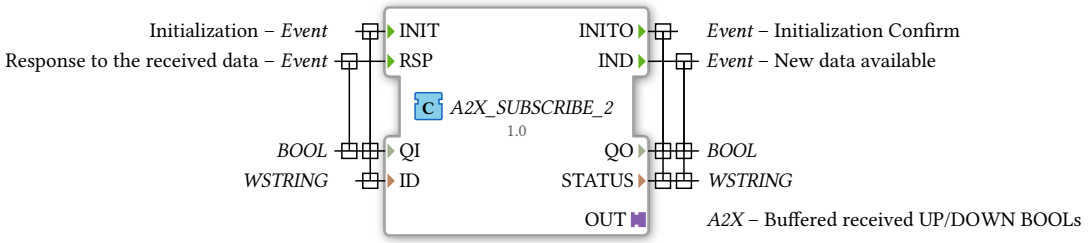
Vergleich mit ähnlichen Bausteinen

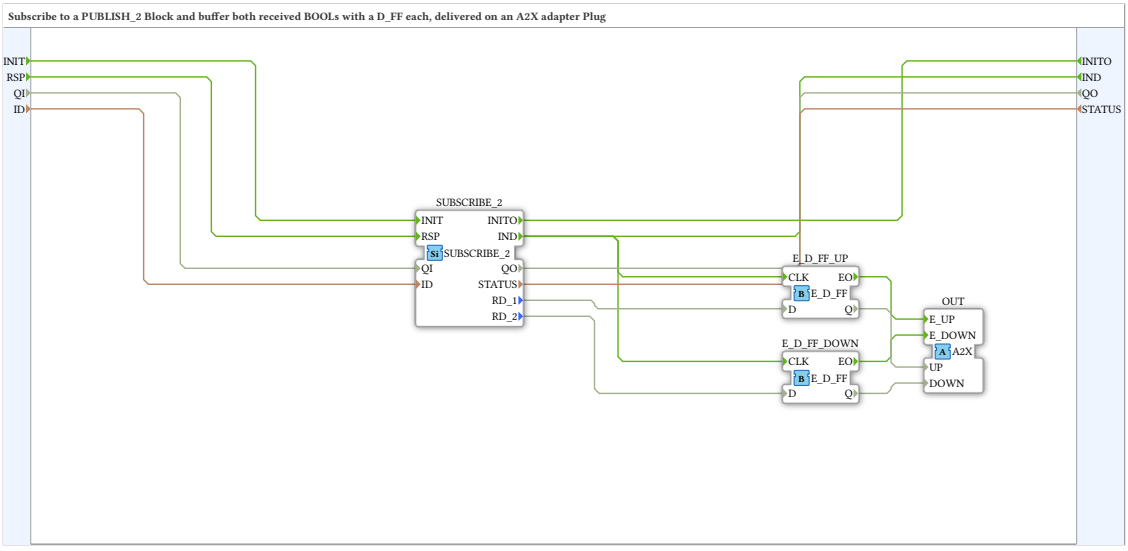
- **`A2X_PUBLISH`** (ohne Suffix „2“): Publiziert nur einen einzelnen BOOL-Wert über einen Adapter. `A2X_PUBLISH_2` erweitert dies auf zwei Werte, was für Anwendungen mit zwei zusammengehörigen Signalen (z.B. Richtung und Status) nützlich ist.
- **`PUBLISH`** (Standard): Bietet eine generische Publikationsfunktion für mehrere Datenwerte, erfordert aber eine manuelle Fehlerbehandlung und Ereignissteuerung. `A2X_PUBLISH_2` kapselt diese Logik und bietet eine komfortablere Adapter-Anbindung.
- **`PUBLISH_2`** (Standard): Kann zwei Datenwerte senden, benötigt aber zusätzliche Logik, um nur bei Änderungen zu senden. Der vorliegende Baustein ergänzt diese Funktionalität durch die integrierten Flip-Flops.

Fazit

`A2X_PUBLISH_2` ist ein spezialisierter Baustein für die effiziente Veröffentlichung von zwei booleschen Signalen über ein Netzwerk. Durch die Kombination von Adapter, flankengetriggerten Speichern und dem Standard-Publisher wird eine robuste und einfach zu integrierende Lösung für verteilte Steuerungssysteme bereitgestellt. Der Baustein reduziert die Netzwerkkommunikation auf tatsächliche Änderungen und erleichtert die Anbindung an höhere Ebenen. Dank der klaren Schnittstellen und der integrierten Statusbehandlung ist er für eine Vielzahl industrieller Anwendungen geeignet.

A2X_SUBSCRIBE_2





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **A2X_SUBSCRIBE_2** dient als Subscriber in einem industriellen Kommunikationsnetzwerk. Er empfängt zwei boolesche Werte von einem korrespondierenden **PUBLISH_2**-Block, puffert diese jeweils über ein D-Flipflop und stellt sie über einen unidirektionalen **A2X**-Adapter zur Verfügung. Dadurch können empfangene Signale stabil gehalten und über standardisierte Adapterschnittstellen an nachgelagerte Logik weitergegeben werden.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierung des Bausteins. Wird typischerweise mit einem Initialisierungsereignis ausgelöst und setzt den Subscriber in einen definierten Zustand. Mit den Daten `QI` und `ID` verbunden.
- **RSP** (Event): Antwort auf ein zuvor gesendetes Request-Ereignis. Dient zur Quittierung von Kommunikationsvorgängen. Mit dem Daten-Eingang `QI` verbunden.

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der erfolgreichen Initialisierung. Wird zusammen mit den Daten `QO` und `STATUS` ausgegeben.
- **IND** (Event): Signalisiert, dass neue Daten vom Netzwerk empfangen wurden. Zusammen mit `QO` und `STATUS` ausgegeben.

Daten-Eingänge

- **QI** (BOOL): Qualifizierer für die Initialisierung. Steuert den Initialisierungsvorgang.
- **ID** (WSTRING): Identifikator (z.B. Netzwerkadresse oder Topic) zur eindeutigen Zuordnung des Subscribers.

Daten-Ausgänge

- **QO** (BOOL): Qualifizierer, ob der Baustein betriebsbereit ist.
- **STATUS** (WSTRING): Statusmeldung über den letzten Vorgang (z.B. Fehler oder Erfolg).

Adapter

- **OUT** (Plug, Typ `adapter::types::unidirectional::A2X`): Unidirektionaler Ausgangs-Adapter. Stellt die beiden gepufferten booleschen Werte (`UP` und `DOWN`) sowie die zugehörigen Ereignisse (`E_UP`, `E_DOWN`) bereit.

Funktionsweise

Der Baustein integriert intern einen **SUBSCRIBE_2**-Netzwerkbaustein (aus `iec61499::net`), der über die Eingänge `QI` und `ID` konfiguriert wird. Beim Eintreffen eines neuen Datensatzes (Ereignis `IND` des SUBSCRIBE_2) werden die beiden empfangenen Werte (`RD_1` und `RD_2`) jeweils an ein D-Flipflop (`E_D_FF_UP` und `E_D_FF_DOWN`) übergeben. Das Ereignis `IND` taktet gleichzeitig beide Flipflops, sodass die aktuellen Werte übernommen werden. Anschließend werden die Ausgänge der Flipflops über den Adapter `OUT` bereitgestellt. Die Adapter-Ereignisse `E_UP` und `E_DOWN` signalisieren, dass der jeweilige Wert aktualisiert wurde. Zusätzlich wird das `IND`-Ereignis des Subscribers direkt an den Ausgang `IND` weitergeleitet, sodass auch externe Bausteine über neue Daten informiert werden.

Technische Besonderheiten

- **D-Flipflops zur Entkopplung**: Die zwei empfangenen BOOL-Werte werden zwischengespeichert, um kurze Signalpegel zu stabilisieren und eine saubere Übergabe über den Adapter zu gewährleisten.
- **Unidirektionaler Adapter**: Der Ausgangs-Adapter ist als Plug konzipiert, d.h. er gibt Daten nur aus und empfängt keine Rückmeldungen – ideal für reine Sensor-/Statusübertragungen.
- **Wiederverwendbarkeit**: Durch die Kapselung der Subscriber-Funktionalität und Pufferung in einem FB kann er leicht in verschiedene Automatisierungsprojekte integriert werden.
- **Ereignisgesteuerte Aktualisierung**: Die Flipflops werden nur bei einem neuen Datensatz getaktet (über `IND`), sodass keine unnötige CPU-Last entsteht.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten, folgt aber einem impliziten Ablauf:

1. **Initialisierung**: Über `INIT` wird der interne `SUBSCRIBE_2` mit `QI` und `ID` konfiguriert. Nach erfolgreicher Initialisierung wird `INITO` mit `QO=TRUE` und `STATUS` entsprechend ausgegeben.

2. **Betrieb**: Nach der Initialisierung lauscht der Baustein auf eingehende Daten. Bei jedem `IND`-Ereignis werden die neuen Werte in die Flipflops übernommen und über den Adapter ausgegeben.
3. **Fehlerzustand**: Falls die Kommunikation fehlschlägt, wird `STATUS` mit einer Fehlermeldung belegt und `QO` auf `FALSE` gesetzt.

Anwendungsszenarien

- **Fernsteuerung von Aktoren**: Empfang von zwei booleschen Steuersignalen (z.B. Auf/Ab, Ein/Aus) von einer zentralen Steuerung und Weiterleitung an lokale Logik über den A2X-Adapter.
- **Statusüberwachung**: Übertragung von zwei Sensorzuständen (z.B. Grenzwertüberschreitung, Fehler) in ein lokales Leitsystem.
- **Modulare Automatisierung**: Einsatz in modularen Maschinenkonzepten, bei denen standardisierte Adapterschnittstellen für die Kommunikation zwischen verschiedenen Teilsystemen verwendet werden.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem einfachen `SUBSCRIBE_2`-Baustein, der die empfangenen Daten direkt an Datenausgänge weiterreicht, bietet `A2X_SUBSCRIBE_2` zusätzlich:

- **Zwischenspeicherung** der Werte über D-Flipflops, wodurch die Daten auch nach dem Empfang stabil bleiben.
- **Adapter-Schnittstelle** statt direkter Datenausgänge – dies erleichtert die Integration in bestehende Adapter-basierte Architekturen.
- **Separate Ereignisse** für jeden gepufferten Wert (`E_UP`, `E_DOWN`), was eine präzise Synchronisation mit nachgelagerten Bausteinen ermöglicht. Gegenüber einer Lösung mit zwei getrennten Subscriber-Blöcken (einer pro BOOL) reduziert dieser Baustein den Konfigurationsaufwand und vereinheitlicht die Datenaufnahme.

Fazit

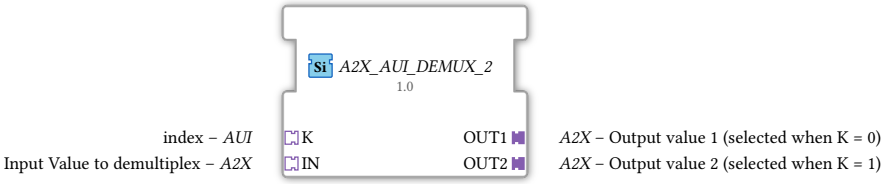
Der Funktionsblock **A2X_SUBSCRIBE_2** stellt eine robuste und kompakte Lösung dar, um zwei boolesche Signale über ein Netzwerk zu empfangen und gepuffert über eine standardisierte Adapterschnittstelle bereitzustellen. Durch die Kombination von Subscriber-Funktionalität, D-Flipflop-Pufferung und Adapter-Integration bietet er eine hohe Wiederverwendbarkeit und erleichtert die Entwicklung verteilter Automatisierungssysteme. Insbesondere für Anwendungen, die eine stabile und ereignisgesteuerte Übergabe von Binärsignalen erfordern, ist dieser Baustein eine empfehlenswerte Wahl.

2.1.11.14 selection

unidirectional

BOOL

A2X_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **A2X_AUI_DEMUX_2** ist ein generischer Demultiplexer, der einen über einen Adapter anliegenden Wert in Abhängigkeit eines Indexes an einen von zwei Ausgängen weiterleitet. Die Schnittstellen sind vollständig als unidirektionale Adapter realisiert. Der Baustein aktualisiert einen Adapter-Ausgang nur bei einer tatsächlichen Wertänderung; das zugehörige Ereignis wird ebenfalls nur dann ausgelöst.

Schnittstellenstruktur

Der Baustein besitzt keine klassischen Datenports. Der Datenaustausch erfolgt über Adapter-Sockets und Adapter-Plugs. Es sind keine Ereignisausgänge vorhanden.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge.

Daten-Eingänge

Keine. Der Datentransport erfolgt über Adapter.

Daten-Ausgänge

Keine. Der Datentransport erfolgt über Adapter.

Adapter

Richtung	Name	Typ	Beschreibung
Socket	`IN`	`adapter::types::unidirectional::A2X`	Eingangswert, der demultiplext wird.
Socket	`K`	`adapter::types::unidirectional::AUI`	Auswahlindex: `K = 0` wählt `OUT1`, `K = 1` wählt `OUT2`.
Plug	`OUT1`	`adapter::types::unidirectional::A2X`	Ausgang 1, aktiv bei `K = 0`.
Plug	`OUT2`	`adapter::types::unidirectional::A2X`	Ausgang 2, aktiv bei `K = 1`.

Funktionsweise

Der Demultiplexer arbeitet wie ein umschaltbarer Verteiler:

- Der am Socket `IN` anliegende Wert wird unverändert an den ausgewählten Adapter-Ausgang weitergeleitet.
- Bei `K = 0` wird `OUT1` aktualisiert.
- Bei `K = 1` wird `OUT2` aktualisiert.
- Der jeweils nicht ausgewählte Ausgang behält seinen bisherigen Wert.

Eine Aktualisierung eines Adapter-Ausgangs erfolgt nur bei einer tatsächlichen Wertänderung. Dadurch werden unnötige Ereignisse und Datenübertragungen vermieden.

Technische Besonderheiten

- Vollständig adapterbasierte Schnittstelle ohne klassische Daten-Ein-/Ausgänge.
- Verwendung unidirektionaler Adaptertypen `A2X` und `AUI`.
- Der Baustein ist als generischer Funktionsblock mit dem ClassName `GEN_A2X_AUI_DEMUX` hinterlegt.
- Der Baustein ist dem Compiler-Paket `adapter::selection::unidirectional` zugeordnet.
- Laut Versionsinformation ist das zugehörige Adapter-Backend `GEN_A2X_AUI_DEMUX` noch zu implementieren.

Zustandsübersicht

Eine explizite Zustandsmaschine ist in der Typdefinition nicht enthalten. Das Verhalten lässt sich konzeptionell wie folgt beschreiben:

Zustand	Bedeutung
Warten	Es liegt keine Wertänderung vor; `OUT1` und `OUT2` behalten ihre Werte.
Auswahl	Der Index `K` wird ausgewertet und der zugehörige Ausgang wird mit dem Wert von `IN` aktualisiert.

Anwendungsszenarien

- Verteilung eines Wertes an zwei unterschiedliche Verbraucher, z. B. Steuerung und Visualisierung.

- Umschalten zwischen zwei Betriebsmodi anhand eines Auswahlindexes.
- Einsatz in IEC-61499-Systemen mit unidirektionalen Adapterverbindungen zur Reduzierung der Ereignislast.

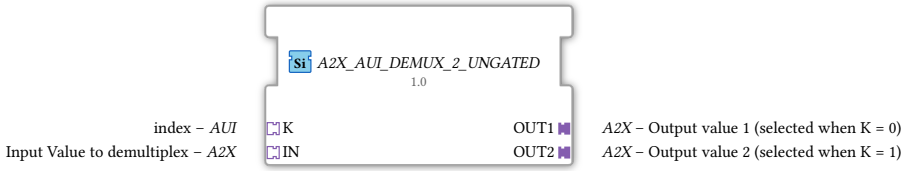
Vergleich mit ähnlichen Bausteinen

Gegenüber klassischen Demultiplexer-Bausteinen, die separate Daten- und Ereignisports verwenden, kapselt `A2X_AUI_DEMUX_2` die Datenschnittstellen vollständig in Adapter. Eine verwandte Variante ist `AX_AUI_DEMUX_2`, auf der dieser Baustein basiert. Der wesentliche Unterschied liegt in der Adapterbasierten Anbindung und der integrierten Änderungserkennung, durch die Adapter-Ausgänge nur bei tatsächlichen Wertänderungen aktualisiert werden.

Fazit

`A2X_AUI_DEMUX_2` ist ein kompakter, adapter-basierter Demultiplexer für zwei unidirektionale Ausgänge. Er ermöglicht eine klare modulare Verdrahtung und vermeidet unnötige Ereignisse durch seine Änderungserkennung. Damit eignet er sich gut für selektive Wertverteilungen in IEC-61499-Anwendungen.

A2X_AUI_DEMUX_2_UNGATED



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **A2X_AUI_DEMUX_2_UNGATED** ist ein generischer, adapterbasierter 1-aus-2-Demultiplexer. Er leitet einen über den Adapter `IN` ankommenden A2X-Wert an einen der beiden Ausgänge `OUT1` oder `OUT2` weiter. Die Auswahl des Ausgangs erfolgt über den Index-Adapter `K`.

Die Erweiterung **UNGATED** bedeutet, dass keine Änderungserkennung stattfindet. Jedes neu berechnete Ergebnis wird bedingungslos und ohne Filterung an den ausgewählten Ausgang weitergereicht – auch dann, wenn sich der Wert gegenüber dem vorherigen Zyklus nicht geändert hat.

Schnittstellenstruktur

Ereignis-Eingänge

Keine separaten Ereignis-Eingänge vorhanden.
Die auslösenden Ereignisse werden über die Adapter-Schnittstellen transportiert.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Adapter	Typ	Richtung	Kommentar
`OUT1`	`adapter::types::unidirectional::A2X`	Plug	Ausgangswert 1, ausgewählt wenn `K = 0`
`OUT2`	`adapter::types::unidirectional::A2X`	Plug	Ausgangswert 2, ausgewählt wenn `K = 1`
`K`	`adapter::types::unidirectional::AUI`	Socket	Auswahlindex

IN	adapter::types::unidirectional::A2X	Socket	Eingangswert zum Demultiplexen

Funktionsweise

Der Baustein arbeitet als Demultiplexer:

- Der aktuelle A2X-Wert liegt am Socket `IN` an.
- Der Socket `K` liefert den Auswahlindex.
- Bei `K = 0` wird der Wert an `OUT1` weitergeleitet.
- Bei `K = 1` wird der Wert an `OUT2` weitergeleitet.

Der Zusatz **UNGATED** ist die entscheidende Eigenschaft: Es findet keine Erkennung statt, ob sich der Wert geändert hat. Jeder neue Berechnungsschritt beziehungsweise jedes neue Ergebnis wird unmittelbar und bedingungslos an den ausgewählten Ausgang weitergegeben.

Technische Besonderheiten

- Der Baustein ist als generischer Funktionsbaustein angelegt.
  Über das Attribut `eclipse4diac::core::GenericClassName = 'GEN_A2X_AUI_DEMUX'` wird die generische Implementierung referenziert.
- Die Adapter gehören zum Paket `adapter::selection::unidirectional`.
- Es gibt keine direkten Daten-Eingänge oder Daten-Ausgänge. Die Daten werden vollständig über die unidirektionalen Adapter übertragen.
- `UNGATED` bedeutet: keine Unterdrückung unveränderter Werte, keine Flankenerkennung, keine Änderungsfilterung.
- Die Versionsinformation weist darauf hin, dass das Adapter-Backend `GEN_A2X_AUI_DEMUX` in dieser Version noch zu implementieren ist.

Zustandsübersicht

Der Baustein besitzt keine explizite Zustandsmaschine beziehungsweise keinen eigenen ECC. Das Verhalten ist reaktiv und wird über die Adapter angestoßen. Logisch lassen sich folgende Phasen unterscheiden:

Phase	Beschreibung
Indexübernahme	Der Socket `K` liefert den Auswahlindex `0` oder `1`.
Wertweiterleitung	Der aktuelle `IN`-Wert wird an `OUT1` oder `OUT2` übergeben.

Da keine Änderungserkennung erfolgt, wird auch ein unveränderter Wert in jedem Zyklus erneut weitergeleitet.

Anwendungsszenarien

- **Periodische Ableitungsberechnung:** Ein Messwert wird zyklisch an einen Differenzierer weitergegeben. Auch unveränderte Werte müssen ankommen, damit die zeitliche Kadenz erhalten bleibt.
- **Frequenzberechnung:** Gleiche aufeinanderfolgende Werte dürfen nicht ausgeblendet werden, weil die Frequenzbestimmung auf der Zählung aller Abtastwerte beruht.
- **Signalverteilung:** Ein A2X-Wert wird je nach Betriebsmodus an einen von zwei Verbrauchern geroutet, ohne dass gleiche Werte unterdrückt werden.

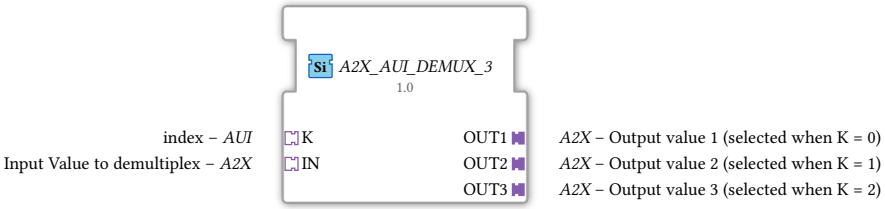
Vergleich mit ähnlichen Bausteinen

Baustein	Eigenschaft
`A2X_AUI_DEMUX_2`	Gleiche Struktur, jedoch mit Änderungserkennung: Es werden nur Werte weitergegeben, die sich gegenüber dem vorherigen Wert geändert haben.
`A2X_AUI_DEMUX_2_UNGATED`	Ungefilterte Weitergabe: Jedes neu berechnete Ergebnis wird unabhängig von einer Wertänderung weitergeleitet.
`AX_AUI_DEMUX_2_UNGATED`	Frühere Variante mit AX-Adapter statt A2X; die Demultiplexerlogik ist grundsätzlich vergleichbar.

Fazit

`A2X_AUI_DEMUX_2_UNGATED` ist ein generischer Demultiplexer für adapterbasierte A2X-Werte. Seine Stärke liegt in der kadenztreuen Weitergabe jedes Ergebnisses ohne Änderungsfilterung. Damit ist er besonders geeignet für Verbraucher, die eine periodische Aktualisierung benötigen, wie etwa Ableitungs- oder Frequenzberechnungen. Vor dem Einsatz sollte jedoch der Implementierungsstand des Backends `GEN_A2X_AUI_DEMUX` geprüft werden.

A2X_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `A2X_AUI_DEMUX_3` ist ein generischer Demultiplexer auf Adapterbasis. Er empfängt über einen `A2X`-Adapter einen Eingangswert und leitet diesen in Abhängigkeit eines über den `AUI`-Adapter anliegenden Index an einen von drei Ausgängen weiter. Der Baustein ist so optimiert, dass ein Adapter-Ausgang nur bei einer tatsächlichen Wertänderung aktualisiert wird und Ereignisse nur bei echten Änderungen ausgelöst werden.

Schnittstellenstruktur

Der Baustein besitzt keine klassischen Ereignis- oder Datenein-/ausgänge. Die gesamte Kommunikation erfolgt über unidirektionale Adapter.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Sockets / Eingänge

Name	Typ	Beschreibung
`K`	`adapter::types::unidirectional::AUI`	Index zur Auswahl des Ausgangs. Gültige Werte: 0, 1, 2.
`IN`	`adapter::types::unidirectional::A2X`	Eingangswert, der auf einen der Ausgänge verteilt werden soll.

Plugs / Ausgänge

Name	Typ	Beschreibung
`OUT1`	`adapter::types::unidirectional::A2X`	Ausgang 1, ausgewählt bei `K = 0`.
`OUT2`	`adapter::types::unidirectional::A2X`	Ausgang 2, ausgewählt bei `K = 1`.
`OUT3`	`adapter::types::unidirectional::A2X`	Ausgang 3, ausgewählt bei `K = 2`.

Funktionsweise

Der Baustein arbeitet als Demultiplexer auf Adapterebene. Ein über den IN-Adapter eintreffender Wert wird an denjenigen Ausgang weitergegeben, der durch den aktuellen Wert des K-Adapters bestimmt wird.

Der Index K wird über den gleichnamigen Socket empfangen. Sobald sich der Index ändert, wird die Zuordnung angepasst.

Der eigentliche Datenwert wird über den A2X-Adapter IN empfangen. Nur der aktuell ausgewählte Ausgang erhält den Wert. Die nicht ausgewählten Ausgänge bleiben unverändert.

Eine Besonderheit ist die änderungsbasierte Aktualisierung: Wenn ein neuer Wert empfangen wird, wird geprüft, ob sich der Wert tatsächlich gegenüber dem zuletzt übertragenen Wert geändert hat. Nur in diesem Fall wird der Ausgang aktualisiert und ein Ereignis ausgelöst. Dadurch werden redundante Verarbeitungsschritte und unnötige Ereignisse vermieden.

Technische Besonderheiten

- **Generischer Funktionsbaustein:** Die konkrete Laufzeitlogik wird über die GenericClassName GEN_A2X_AUI_DEMUX bereitgestellt. Der vorliegende Baustein ist die konkrete Ausprägung mit drei Ausgängen.
- **Reine Adapter-Schnittstelle:** Es gibt keine direkten Ereignis- oder Datenanschlüsse. Die Kommunikation erfolgt vollständig über die unidirektionalen Adapter A2X und AUI.
- **Änderungsbasierte Ausgabe:** Der Adapter-Ausgang wird nur bei einer tatsächlichen Wertänderung aktualisiert. Ein Ereignis wird ebenfalls nur bei einer echten Änderung erzeugt.
- **Drei Ausgangskanäle:** OUT1, OUT2 und OUT3 decken die Indexwerte 0, 1 und 2 ab.
- **Bereitstellungshinweis:** In der vorliegenden Definition ist das Adapter-Backend GEN_A2X_AUI_DEMUX als noch zu implementieren gekennzeichnet. Die Schnittstellenbeschreibung ist vollständig; die Laufzeitlogik muss vor dem Einsatz bereitgestellt oder generiert werden.

Zustandsübersicht

Der generische Baustein besitzt in der XML-Definition keine explizit sichtbare ECC. Logisch lässt sich das Verhalten in folgende Zustände unterteilen:

Zustand	Auslöser	Verhalten
IDLE	Keine Änderung	Wartet auf neue Werte oder einen neuen Index. Es findet keine Ausgabeänderung statt.
INDEX_CHANGE	Neuer Wert am Adapter K	Die Auswahl wird auf OUT1, OUT2 oder OUT3 umgestellt.
VALUE_CHANGE	Neuer Wert am Adapter IN	Der Wert wird an den aktuell ausgewählten Ausgang übergeben. Die Aktualisierung erfolgt nur bei tatsächlicher Wertänderung.

Anwendungsszenarien

- Verteilung eines zentralen A2X-Datenstroms an genau eine von drei möglichen Senken, z. B. an verschiedene Anzeigen, Visualisierungen oder Steuerungsmodule.
- Umschaltung zwischen mehreren Verbrauchern über einen AUI-Index, etwa zur Laufzeit- oder Stationsauswahl.
- Entkoppelte Signalverteilung in modularen IEC-61499-Anlagen, bei denen Ereignisse nur bei tatsächlichen Wertänderungen benötigt werden.
- Wiederverwendung als generischer Adapter-Demultiplexer in einer Bibliothek für adapterbasierte Kommunikationsstrukturen.

Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu einem klassischen Demultiplexer mit getrennten Ereignis- und Dateneingängen besitzt A2X_AUI_DEMUX_3 keine direkten Datenpins. Die Verbindung erfolgt ausschließlich über unidirektionale Adapter. Dadurch lassen sich typisierte Datenströme direkt an andere Adapter anbinden, ohne zusätzliche Verdrahtung auf Ereignis- und Datenebene.

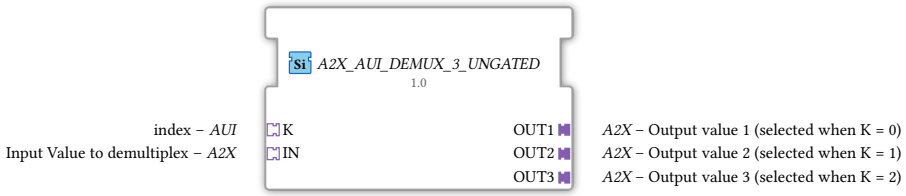
Der Baustein ist eine A2X-Variante des AX_AUI_DEMUX_3. Während der Grundaufbau – ein Eingang, drei Ausgänge und ein Index – gleich bleibt, verwendet diese Variante den Adaptertyp A2X für den Datentransport. Sie ist damit für Umgebungen geeignet, die durchgängig auf A2X-Adapter setzen.

Gegenüber einem Multiplexer, der mehrere Eingänge auf einen Ausgang schaltet, stellt `A2X_AUI_DEMUX_3` die umgekehrte Richtung dar: Ein Eingang wird wahlweise auf einen von drei Ausgängen geschaltet.

Fazit

`A2X_AUI_DEMUX_3` ist ein kompakter, adapterbasierter Demultiplexer für unidirektionale `A2X`-Datenströme. Seine Stärke liegt in der klaren Trennung zwischen Indexauswahl und Datenweiterleitung sowie in der effizienten, änderungsbasierten Ausgabe. Die generische Definition ermöglicht eine saubere Einbindung in IEC-61499-Systeme, sobald das zugehörige Backend `GEN_A2X_AUI_DEMUX` implementiert ist.

A2X_AUI_DEMUX_3_UNGATED



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **A2X_AUI_DEMUX_3_UNGATED** ist ein generischer, adapterbasierter 1-zu-3-Demultiplexer. Er gehört zur A2X/AUI-Familie und leitet einen am Eingangs-Adapter `IN` anliegenden Wert abhängig vom Index `K` an einen der drei Ausgangs-Adapter `OUT1`, `OUT2` oder `OUT3` weiter.

Die Variante **_UNGATED** besitzt **keine Änderungserkennung**. Das bedeutet: Jedes neu berechnete Ergebnis wird bedingungslos an den ausgewählten Ausgang weitergegeben – unabhängig davon, ob sich der Wert gegenüber dem vorherigen geändert hat.

Schnittstellenstruktur

Der Baustein besitzt keine klassischen Daten-Ein- oder Daten-Ausgänge. Sämtliche Werte und Steuerinformationen werden über die Adapter-Sockets und Adapter-Plugs übertragen.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge.

Daten-Eingänge

Keine. Alle Eingangsdaten werden über die Adapter-Sockets `IN` und `K` transportiert.

Daten-Ausgänge

Keine. Alle Ausgangsdaten werden über die Adapter-Plugs `OUT1`, `OUT2` und `OUT3` transportiert.

Adapter

Name	Richtung	Typ	Kommentar
`K`	Socket (Eingang)	`adapter::types::unidirectional::AUI`	Index zur Auswahl des Ausgangs
`IN`	Socket (Eingang)	`adapter::types::unidirectional::A2X`	Eingangswert, der demultiplext wird
`OUT1`	Plug (Ausgang)	`adapter::types::unidirectional::A2X`	Ausgangswert 1, ausgewählt bei K = 0
`OUT2`	Plug (Ausgang)	`adapter::types::unidirectional::A2X`	Ausgangswert 2, ausgewählt bei K = 1

OUT3	Plug (Ausgang)	adapter::types::unidirectional::A2X	Ausgangswert 3, ausgewählt bei K = 2

Funktionsweise

Der Baustein arbeitet als Demultiplexer auf Adapterebene:

1. Über den Socket K wird der gewünschte Ausgangskanal ausgewählt.
2. Über den Socket IN wird der weiterzuleitende Wert empfangen.
3. Der Wert wird an den ausgewählten Ausgang weitergegeben:
   - K = 0 ⟶ OUT1
   - K = 1 ⟶ OUT2
   - K = 2 ⟶ OUT3

Da es sich um die **_UNGATED**-Variante handelt, wird keine Prüfung durchgeführt, ob sich der Eingangswert geändert hat. Jeder ankommende Wert wird unabhängig von einer Wertänderung weitergeleitet. Dadurch eignet sich der Baustein besonders für Verbraucher, die eine periodische Kadenz benötigen – etwa für Ableitungs- oder Frequenzberechnungen.

Technische Besonderheiten

- Der Baustein ist als generische Klasse gekennzeichnet:
  `eclipse4diac::core::GenericClassName = 'GEN_A2X_AUI_DEMUX'`
- Er liegt im Paket `adapter::selection::unidirectional`.
- Der Baustein besitzt keine klassischen Ereignis-Eingänge. Die Ansteuerung erfolgt über die verbundenen Adapter-Sockets.
- Alle Adapter sind unidirektional ausgelegt.
- Es gibt keine Änderungserkennung und keine Filterung des Eingangswerts.
- Laut Versionsinformation ist das eigentliche Adapter-Backend `GEN_A2X_AUI_DEMUX` in dieser Version noch zu implementieren. Der Baustein ist daher als generische Vorlage beziehungsweise als Entwurf zu betrachten.

Zustandsübersicht

In der übergebenen XML ist keine explizite ECC-Zustandsmaschine hinterlegt. Das erwartete konzeptionelle Verhalten lässt sich wie folgt beschreiben:

Zustand	Bedeutung
IDLE	Warten auf gültigen Index K und/oder Eingangswert IN
SELECT	Auswahl des Zielausgangs anhand von K
FORWARD	Bedingungslose Weiterleitung von IN an den gewählten Ausgang

Anwendungsszenarien

- **Periodische Frequenz- oder Ableitungsberechnung:**
  Verbraucher benötigen in jedem Zyklus einen neuen Wert, auch wenn der Zahlenwert unverändert bleibt.
- **Signalverteilung auf mehrere Auswertepfade:**
  Ein Eingangswert wird abhängig vom Index an einen von drei Verarbeitungspfaden weitergeleitet.
- **Zyklische Messwertübertragung:**
  Jeder berechnete Messwert wird ohne vorherige Änderungsprüfung an den ausgewählten Ausgang gesendet.
- **Generischer Aufbau:**
  Durch die Verwendung von Adaptern kann der Baustein mit unterschiedlichen A2X- und AUI-Datentypen verbunden werden.

Vergleich mit ähnlichen Bausteinen

Baustein	Änderungserkennung	Verhalten

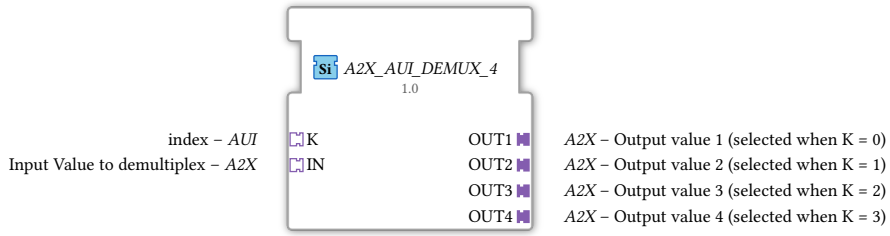
A2X_AUI_DEMUX_3	Ja	Gibt Werte nur bei Änderung weiter
A2X_AUI_DEMUX_3_UNGATED	Nein	Gibt jeden Wert bedingungslos weiter

Gegenüber Varianten mit Änderungserkennung verzichtet der *UNGATED*-Baustein bewusst auf Logik zur Unterdrückung unveränderter Werte. Dadurch wird sichergestellt, dass Verbraucher mit einer festen zeitlichen Kadenz immer ein aktuelles Ergebnis erhalten.

Fazit

A2X_AUI_DEMUX_3_UNGATED ist ein generischer, adapterbasierter 1-zu-3-Demultiplexer ohne Änderungserkennung. Er leitet jeden ankommenden Wert unabhängig von dessen Änderung an den durch K gewählten Ausgang weiter. Damit eignet er sich besonders für periodische und taktgebundene Anwendungen. Da das Adapter-Backend GEN_A2X_AUI_DEMUX laut Versionsinformation noch implementiert werden muss, sollte der Baustein aktuell nur als Entwurf betrachtet werden.

A2X_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein A2X_AUI_DEMUX_4 ist ein Demultiplexer auf Basis unidirektionaler Adapter. Er leitet den am Socket IN anliegenden A2X-Wert an genau einen der vier Ausgänge OUT1 bis OUT4 weiter. Welcher Ausgang aktiviert wird, bestimmt der Socket K.

Besonderheit dieses Bausteins ist das änderungsbasierte Verhalten: Ein Adapter-Ausgang wird nur dann aktualisiert, wenn sich der Wert tatsächlich ändert. Nur in diesem Fall wird auch ein Ereignis an den nachgeschalteten Baustein weitergegeben.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine eigenständigen Ereignis-Eingänge vorhanden. Der Baustein wird über die eingehenden Adapter-Sockets K und IN angestoßen.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge.

Daten-Eingänge

Es sind keine direkten Dateneingänge deklariert. Die Eingangsdaten werden über die Adapter-Sockets K und IN übertragen.

Daten-Ausgänge

Es sind keine direkten Datenausgänge vorhanden. Die Ausgangsdaten werden ausschließlich über die Adapter-Plugs OUT1 bis OUT4 bereitgestellt.

Adapter

Adapter	Richtung	Typ	Kommentar
K	Socket	adapter::types::unidirectional::AUI	Auswahlindex
IN	Socket	adapter::types::unidirectional::A2X	Eingangswert, der demultiplext werden soll

OUT1	Plug	adapter::types::unidirectional::A2X	Ausgang 1, aktiv bei K = 0
OUT2	Plug	adapter::types::unidirectional::A2X	Ausgang 2, aktiv bei K = 1
OUT3	Plug	adapter::types::unidirectional::A2X	Ausgang 3, aktiv bei K = 2
OUT4	Plug	adapter::types::unidirectional::A2X	Ausgang 4, aktiv bei K = 3

Funktionsweise

Der Baustein arbeitet als 1-zu-4-Demultiplexer. Der Wert am Socket IN wird abhängig vom Auswahlwert am Socket K auf genau einen der vier Ausgänge OUT1 bis OUT4 geleitet:

- K = 0 ⟶ OUT1
- K = 1 ⟶ OUT2
- K = 2 ⟶ OUT3
- K = 3 ⟶ OUT4

Wird ein neuer Wert am IN-Socket empfangen, prüft der Baustein, ob der Wert dem zuletzt ausgegebenen Wert des ausgewählten Ausgangs entspricht. Nur bei einer tatsächlichen Wertänderung wird der Ausgang aktualisiert und ein Ereignis über den entsprechenden A2X-Adapter ausgelöst. Nicht ausgewählte Ausgänge behalten ihren letzten Wert.

Technische Besonderheiten

- Der Baustein ist als generischer FB deklariert. Der generische Klassenname lautet GEN_A2X_AUI_DEMUX.
- Die Kommunikation erfolgt vollständig über unidirektionale Adapter; klassische Datenein- und Datenausgänge sind nicht vorhanden.
- Die Ausgänge werden nur bei Wertänderung aktualisiert. Dadurch werden unnötige Ereignisse und Verarbeitungsschritte in nachgelagerten Bausteinen vermieden.
- Die Bezeichnung _4 kennzeichnet die Ausführung mit vier Ausgängen.
- Der Baustein ist eine A2X-Variante des AX_AUI_DEMUX_4 und verwendet die Adapter-Typen A2X und AUI.
- Die eigentliche Logik liegt im generischen Backend GEN_A2X_AUI_DEMUX; die Typdefinition selbst stellt die Schnittstellen und die generische Zuordnung bereit.

Zustandsübersicht

In der XML-Definition ist kein expliziter ECC hinterlegt. Das Verhalten lässt sich logisch wie folgt beschreiben:

Zustand	Bedeutung
Warten	Der Baustein wartet auf einen neuen Wert am Socket K oder IN.
Auswahl	Der Auswahlindex K wird ausgewertet und der zugehörige Ausgang OUT1 bis OUT4 bestimmt.
Prüfen	Der an IN anliegende Wert wird mit dem aktuellen Wert des ausgewählten Ausgangs verglichen.
Aktualisieren	Bei Wertänderung wird der A2X-Ausgang aktualisiert und das Ereignis ausgelöst.

Anwendungsszenarien

- Verteilung eines A2X-Signals an mehrere Regler oder Aktoren, wobei nur ein Empfänger zur gleichen Zeit aktiv ist.
- Betriebsarten-Umschaltung in modularen Automatisierungsanlagen: Ein Sollwert wird je nach Modus an das aktive Modul geroutet.
- Diagnose- und Monitoring-Systeme: Ein Messwert wird wahlweise an Anzeige, Speicher oder Alarmauswertung geleitet.
- Ereignisarme Systeme, in denen die Kommunikationslast durch ausschließliche Aktualisierung bei Wertänderungen reduziert werden soll.

Vergleich mit ähnlichen Bausteinen

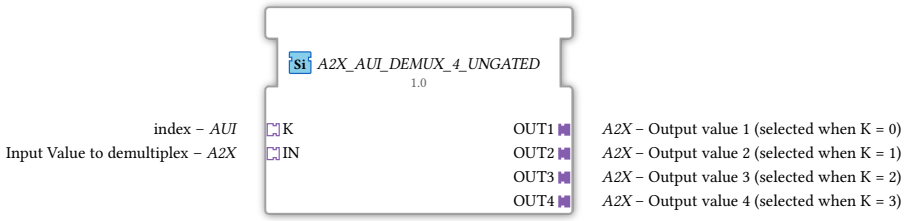
Baustein	Eigenschaft

A2X_AUI_DEMUX_4	Vier A2X-Ausgänge, Auswahl über AUI, ereignisoptimierte Wertänderung
AX_AUI_DEMUX_4	Ursprüngliche Variante mit AX-Adaptern; A2X_AUI_DEMUX_4 ist die A2X-basierte Weiterentwicklung
Konventioneller Demultiplexer	Besitzt direkte Daten-Ein- und Ausgänge; kein Adapterkonzept und keine änderungsbasierte Eventweiterleitung

Fazit

A2X_AUI_DEMUX_4 ist ein flexibler und ereignisoptimierter Adapter-Demultiplexer für vier Ausgänge. Er eignet sich besonders für modulare IEC-61499-Systeme, in denen ein Wert wahlweise an mehrere Senken verteilt werden soll und unnötige Ereignisse vermieden werden müssen. Durch die generische Definition und die Verwendung unidirektionaler Adapter ist er gut wiederverwendbar und in die 4diac-IDE integrierbar.

A2X_AUI_DEMUX_4_UNGATED



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein A2X_AUI_DEMUX_4_UNGATED ist ein adapterbasierter 1-aus-4-Demultiplexer. Er empfängt über einen A2X-Adapter einen Eingangswert und leitet diesen über einen von vier A2X-Ausgangsadaptern weiter. Die Auswahl des aktiven Ausgangs erfolgt über einen AUI-Adapter, der den Index K transportiert.

Die Besonderheit dieser Variante ist der Zusatz UNGATED: Im Gegensatz zu einem Baustein mit Änderungserkennung wird hier **jedes** neu berechnete Ergebnis bedingungslos weitergegeben – auch wenn sich der Wert gegenüber dem vorherigen Zyklus nicht geändert hat. Dadurch eignet sich der Baustein besonders für Verbraucher, die eine periodische Kadenz benötigen, zum Beispiel für Ableitungs- oder Frequenzberechnungen.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ereignis- oder Datenports. Alle Eingangs- und Ausgangswerte werden über Adapter übertragen.

Ereignis-Eingänge

Keine eigenständigen Ereignis-Eingänge definiert.

Die Steuerung und Datenübertragung erfolgt über die Adapter K und IN.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge.

Daten-Eingänge

Keine direkten Daten-Eingänge.

Die Eingangsdaten sind in den Adapter-Sockets K und IN gekapselt.

Daten-Ausgänge

Keine direkten Daten-Ausgänge.

Die Ausgangsdaten werden über die Adapter-Plugs OUT1 bis OUT4 bereitgestellt.

Adapter

Name	Richtung / Rolle	Adaptertyp	Beschreibung
K	Socket / Eingang	adapter::types::unidirectional::AUI	Index zur Auswahl des aktiven Ausgangs
IN	Socket / Eingang	adapter::types::unidirectional::A2X	Eingangswert, der demultiplext wird
OUT1	Plug / Ausgang	adapter::types::unidirectional::A2X	Ausgangswert 1, ausgewählt bei K = 0
OUT2	Plug / Ausgang	adapter::types::unidirectional::A2X	Ausgangswert 2, ausgewählt bei K = 1
OUT3	Plug / Ausgang	adapter::types::unidirectional::A2X	Ausgangswert 3, ausgewählt bei K = 2
OUT4	Plug / Ausgang	adapter::types::unidirectional::A2X	Ausgangswert 4, ausgewählt bei K = 3

Funktionsweise

Der Baustein arbeitet als Adapter-basierter Demultiplexer:

1. Über den Adapter K wird der Auswahlindex empfangen.
2. Über den Adapter IN wird der aktuelle Eingangswert empfangen.
3. Der Baustein leitet diesen Wert unverändert an den Ausgang weiter, der dem Index K entspricht.
4. Da die Variante UNGATED keine Änderungserkennung besitzt, wird jeder ankommende Ergebniszyklus weitergegeben – auch bei unveränderten Werten.

Nicht ausgewählte Ausgänge werden in der Regel nicht bedient und behalten ihren bisherigen Zustand.

Technische Besonderheiten

- **Keine Änderungserkennung:** Jedes neu berechnete Ergebnis wird ungefiltert weitergeleitet.
- **Reine Adapter-Anbindung:** Es gibt keine direkten Daten- oder Ereignis-Eingänge; der Datenaustausch erfolgt über die unidirektionalen Adapter A2X und AUI.
- **Generischer Funktionsbaustein:** Der Baustein ist als generischer FB mit GenericClassName = 'GEN_A2X_AUI_DEMUX' deklariert.
- **Erweiterungskonzept:** Die Bezeichnung A2X verweist auf die Verwendung des A2X-Adaptertyps für die eigentlichen Nutzdaten.
- **Backend-Hinweis:** Die Versionsinformation weist darauf hin, dass das Adapter-Backend GEN_A2X_AUI_DEMUX noch zu implementieren ist. Vor dem Einsatz sollte geprüft werden, ob das verwendete Laufzeitsystem diesen generischen Baustein bereits unterstützt.
- **Lizenz:** Der Baustein ist unter der Eclipse Public License 2.0 verfügbar.

Zustandsübersicht

Der Baustein besitzt keinen explizit sichtbaren Zustandsautomaten. Der relevante Zustand ist der aktuell eingestellte Index K.

Index K	aktiver Ausgang
0	OUT1
1	OUT2
2	OUT3
3	OUT4

Für Indexwerte außerhalb von 0..3 ist in der Typdefinition kein Verhalten festgelegt.

Anwendungsszenarien

- **Ableitungs- und Frequenzberechnungen:** Verbraucher, die jeden Takt benötigen, auch wenn sich der Messwert nicht ändert.

- **Parallele Auswertung:** Ein analoger oder generischer Datenwert wird auf vier unterschiedliche Auswertungsmodule verteilt.
- **Datenstrom-Umschaltung:** Ein ankommender Wert wird abhängig vom Index `K` an genau eine von vier Senken weitergereicht.
- **Periodische Diagnose:** Rohdaten werden ohne Filterung und ohne Verlust von Wiederholungen an Beobachter oder Analysefunktionen übergeben.
- **Adapterbasierte Systeme:** Der Baustein eignet sich für modulare Architekturen, die mit unidirektionalen `A2X`- und `AUI`-Adaptern arbeiten.

Vergleich mit ähnlichen Bausteinen

Baustein	Eigenschaft
`A2X_AUI_DEMUX_4`	Leitet Werte nur bei Änderung des Eingangswerts weiter.
`A2X_AUI_DEMUX_4_UNGATED`	Leitet jeden Ergebniszyklus bedingungslos weiter, unabhängig von einer Wertänderung.
`AX_AUI_DEMUX_4_UNGATED`	Vergleichbare Demultiplexer-Variante mit anderem Adapter-Backend; der vorliegende Baustein ist die `A2X`-basierte Variante.

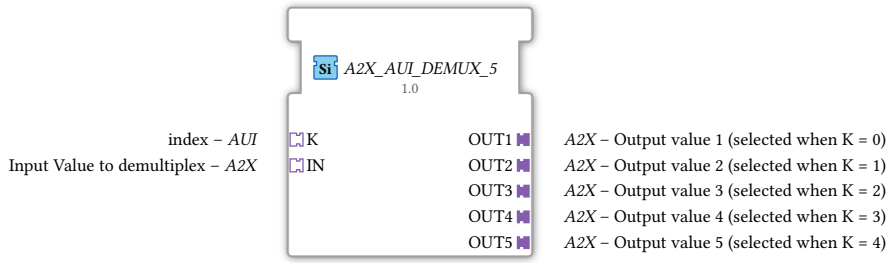
Der entscheidende Unterschied zu nicht-`UNGATED`-Varianten ist das bewusste Fehlen der Änderungserkennung. Dadurch ist die `UNGATED`-Variante besser geeignet, wenn eine gleichbleibende, periodische Weitergabe von Werten wichtiger ist als die Unterdrückung redundanter Daten.

Fazit

`A2X_AUI_DEMUX_4_UNGATED` ist ein kompakter, adapterbasierter 1-aus-4-Demultiplexer ohne Wertänderungsfilterung. Er eignet sich besonders für Anwendungen, die eine kontinuierliche und periodische Datenweitergabe benötigen, wie etwa Ableitungs- oder Frequenzberechnungen.

Durch die rein adapterbasierte Schnittstelle lässt sich der Baustein gut in modulare 4diac-Projekte integrieren. Vor dem Einsatz muss sichergestellt werden, dass das zugehörige generische Backend `GEN_A2X_AUI_DEMUX` in der verwendeten Laufzeitumgebung verfügbar ist.

A2X_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `A2X_AUI_DEMUX_5` ist ein Demultiplexer mit fünf Ausgängen. Er leitet den am Adapter-Socket `IN` anliegenden Wert an einen der fünf Adapter-Plugs `OUT1` bis `OUT5` weiter. Die Auswahl des aktiven Ausgangs erfolgt über den Indexwert am Adapter-Socket `K`. Es handelt sich um eine A2X-Variante von `AX_AUI_DEMUX_5` mit fester Ausgangsanzahl von 5. Der Baustein ist als generischer Funktionsblock auf Basis des Generik-Gerüsts `GEN_A2X_AUI_DEMUX` modelliert.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine Ereignis-Eingänge.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge.

Daten-Eingänge

Der Baustein besitzt keine Daten-Eingänge.

Daten-Ausgänge

Der Baustein besitzt keine Daten-Ausgänge.

Adapter

Der Baustein verwendet ausschließlich Adapter-Schnittstellen zur Übertragung von Werten und Steuerinformationen.

Plugs (Ausgänge):

Name	Typ	Kommentar
OUT1	adapter::types::unidirectional::A2X	Ausgangswert 1 (aktiv bei K = 0)
OUT2	adapter::types::unidirectional::A2X	Ausgangswert 2 (aktiv bei K = 1)
OUT3	adapter::types::unidirectional::A2X	Ausgangswert 3 (aktiv bei K = 2)
OUT4	adapter::types::unidirectional::A2X	Ausgangswert 4 (aktiv bei K = 3)
OUT5	adapter::types::unidirectional::A2X	Ausgangswert 5 (aktiv bei K = 4)

Sockets (Eingänge):

Name	Typ	Kommentar
K	adapter::types::unidirectional::AUI	Index zur Auswahl des Ausgangs
IN	adapter::types::unidirectional::A2X	Eingangswert, der demultiplext wird

Funktionsweise

Der Demultiplexer leitet den über IN empfangenen Wert an den durch K bestimmten Ausgang weiter. Dabei gilt:

- K = 0 ⟶ Ausgabe auf OUT1
- K = 1 ⟶ Ausgabe auf OUT2
- K = 2 ⟶ Ausgabe auf OUT3
- K = 3 ⟶ Ausgabe auf OUT4
- K = 4 ⟶ Ausgabe auf OUT5

Der Baustein arbeitet rein wertänderungsbasiert. Ein Adapter-Ausgang wird nur dann aktualisiert, wenn sich der Wert tatsächlich ändert.

Technische Besonderheiten

- Der Baustein ist eine **A2X-Variante** von AX_AUI_DEMUX_5 und nutzt den Adaptertyp A2X für Ein- und Ausgänge sowie AUI für den Index.
- Er ist als **generischer Funktionsblock** angelegt. Die generische Klasse lautet GEN_A2X_AUI_DEMUX. Die vorliegende Instanz A2X_AUI_DEMUX_5 hat fest fünf Ausgänge.
- Der Baustein enthält keine klassischen Daten- oder Ereignis-Ein-/Ausgänge; die gesamte Kommunikation erfolgt über Adapter.
- Das Ausgabe-Verhalten ist ereignisgesteuert, jedoch werden die Adapter-Ausgänge nur bei tatsächlicher Wertänderung aktualisiert.
- Die Paketbezeichnung adapter::selection::unidirectional ordnet den Baustein in eine Bibliothek für unidirektionale Selektions-Adapter ein.

Zustandsübersicht

Eine explizite Zustandsmaschine ist für diesen Funktionsblock nicht hinterlegt. Das Verhalten wird durch den generischen Backend-Baustein GEN_A2X_AUI_DEMUX realisiert. Konzeptionell lässt sich der Ablauf in zwei Phasen beschreiben:

1. **Bereitschaft**: Der Baustein wartet auf einen neuen Wert an IN oder einen neuen Indexwert an K.

Da keine expliziten Zustände definiert sind, beschreibt dieses Modell das erwartete Kommunikationsverhalten.

Anwendungsszenarien

- **Auswahl eines von mehreren Zielen**: Ein Messwert oder Datentelegramm wird über den Eingang `IN` empfangen und je nach Index an einen von fünf nachgeschalteten Bausteinen weitergeleitet.
- **Routing in modularen Automatisierungssystemen**: In einer Steuerungsarchitektur können Datenkanäle abhängig von einem Index auf verschiedene Verbraucher oder Senken geschaltet werden.
- **Test- und Simulationsaufbauten**: Der Baustein kann verwendet werden, um Werte gezielt an verschiedene Testpfade zu verteilen, ohne die Verdrahtung zu ändern.

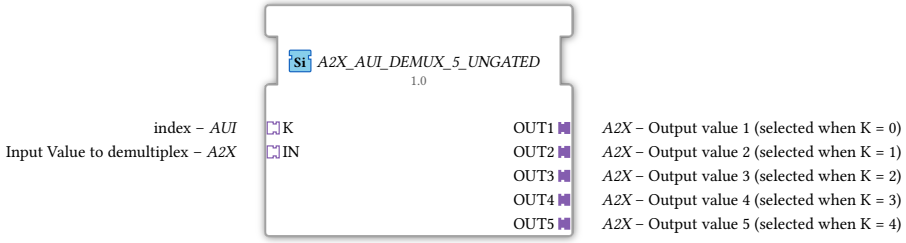
Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu einem klassischen Multiplexer (mehrere Eingänge, ein Ausgang) führt dieser Baustein eine Demultiplex-Funktion aus. Während Varianten wie `AX_AUI_DEMUX_5` möglicherweise herkömmliche Daten- und Ereignisschnittstellen verwenden, setzt `A2X_AUI_DEMUX_5` konsequent auf die unidirektionalen Adaptertypen `A2X` und `AUI`. Dadurch wird die Integration in adapterbasierte Kommunikationsstrukturen erleichtert. Durch die wertänderungsbasierte Aktualisierung und die daraus folgende bedingte Ausgabe unterscheidet er sich von Bausteinen, die bei jedem Empfang eines Ereignisses einen Ausgang unabhängig von einer Wertänderung aktualisieren.

Fazit

`A2X_AUI_DEMUX_5` ist ein spezialisierter Demultiplexer mit fünf Ausgängen für unidirektionale Adapterverbindungen. Er eignet sich für den wertabhängigen Datentransfer in modularen Automatisierungs- und Kommunikationsarchitekturen. Die Beschränkung auf Adapterschnittstellen und die wertänderungsbasierte Informationsweitergabe ermöglichen eine präzise und effiziente Weiterleitung von Daten, ohne unnötige Ereignisse zu erzeugen. Der Baustein stellt damit eine nützliche Komponente für adapterbasierte Selektions- und Verteilaufgaben dar.

A2X_AUI_DEMUX_5_UNGATED



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **A2X_AUI_DEMUX_5_UNGATED** ist ein unidirektionaler Demultiplexer mit fünf Ausgängen. Er empfängt über den Adapter-Socket `IN` einen Wert und über den Adapter-Socket `K` einen Index. Abhängig vom Index wird der empfangene Wert an einen der fünf Ausgangs-Adapter `OUT1` bis `OUT5` weitergeleitet.

Die Variante **UNGATED** verzichtet auf eine Änderungserkennung. Das bedeutet: Jedes neu ankommende Ergebnis wird bedingungslos an den ausgewählten Ausgang weitergegeben – auch dann, wenn sich der Wert gegenüber der vorherigen Übertragung nicht geändert hat.

• – ◦ • – ◦ • – ◦ •

Schnittstellenstruktur

Ereignis-Eingänge

Keine eigenständigen Ereignis-Eingänge. Die verarbeitungsrelevanten Ereignisse werden über die Adapter-Sockets transportiert.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge.

Daten-Eingänge

Keine direkten Daten-Eingänge. Die Daten werden über die Adapter-Sockets `IN` und `K` geführt.

Daten-Ausgänge

Keine direkten Daten-Ausgänge. Die Ausgangsdaten werden über die Adapter-Plugs `OUT1` bis `OUT5` bereitgestellt.

Adapter

Adapter	Richtung	Typ	Beschreibung
`K`	Socket	`adapter::types::unidirectional::AUI`	Index zur Auswahl eines Ausgangs
`IN`	Socket	`adapter::types::unidirectional::A2X`	Eingangswert, der demultiplexiert werden soll
`OUT1`	Plug	`adapter::types::unidirectional::A2X`	Ausgang 1, aktiv bei K = 0
`OUT2`	Plug	`adapter::types::unidirectional::A2X`	Ausgang 2, aktiv bei K = 1
`OUT3`	Plug	`adapter::types::unidirectional::A2X`	Ausgang 3, aktiv bei K = 2
`OUT4`	Plug	`adapter::types::unidirectional::A2X`	Ausgang 4, aktiv bei K = 3
`OUT5`	Plug	`adapter::types::unidirectional::A2X`	Ausgang 5, aktiv bei K = 4

• – ◦ • – ◦ • – ◦ •

Funktionsweise

Der Baustein arbeitet als 1-aus-5-Demultiplexer auf Adapterbasis.

1. Über den Socket `K` wird der gewünschte Ausgangskanal als Index empfangen.
2. Über den Socket `IN` trifft der zu verteilende Wert ein.
3. Der Wert wird an den durch `K` ausgewählten Ausgangs-Adapter `OUT1` bis `OUT5` weitergeleitet.

Die Zuordnung lautet:

- K = 0 ⟶ `OUT1`
- K = 1 ⟶ `OUT2`
- K = 2 ⟶ `OUT3`
- K = 3 ⟶ `OUT4`
- K = 4 ⟶ `OUT5`

Durch die **UNGATED**-Eigenschaft gibt es keine Filterung auf Wertänderungen. Ein konstanter Wert wird also bei jedem Ereignis erneut an den gewählten Ausgang übertragen.

• – ◦ • – ◦ • – ◦ •

Technische Besonderheiten

- **Keine Änderungserkennung:** Der Baustein leitet jedes neu berechnete Ergebnis ohne Bedingung weiter.
- **Adapterbasierte Ein-/Ausgabe:** Die Datenübertragung erfolgt über unidirektionale Adapter vom Typ `A2X` und `AUI`.
- **Generischer Baustein:** Der Baustein ist als generischer FB deklariert und verwendet die generische Klasse `GEN_A2X_AUI_DEMUX`.
- **Kein ECC:** Es ist kein internes Zustandsdiagramm beziehungsweise keine Ablaufsteuerung (ECC) vorhanden.
- **Hinweis:** Laut Versionsinformation ist das zugehörige Adapter-Backend `GEN_A2X_AUI_DEMUX` noch zu implementieren.

• – ◦ • – ◦ • – ◦ •

Zustandsübersicht

Der Baustein besitzt keine explizite Zustandsmaschine. Die logische Auswahl wird über den Index `K` bestimmt.

K	Aktiver Ausgang

0	OUT1
1	OUT2
2	OUT3
3	OUT4
4	OUT5
Sonstige Werte	Nicht definiert / kein Ausgang spezifiziert

• – ◦ • – ◦ • – ◦ •

Anwendungsszenarien

- **Ableitungs- und Frequenzberechnungen:** Verbraucher, die eine periodische Kadenz benötigen, erhalten auch bei unveränderten Werten weiterhin ein Ereignis.
- **Kanalwahl:** Ein einzelner Messwert kann abhängig von einer Steuergröße gezielt an verschiedene Ausgänge verteilt werden.
- **Zyklische Datenverteilung:** Der FB eignet sich für Systeme, bei denen jeder Takt eine aktualisierte Ausgabe erwartet wird, unabhängig davon, ob sich der Datenwert geändert hat.
- **Test- und Simulationsumgebungen:** Zur gezielten Umschaltung zwischen mehreren Datenempfängern über einen zentralen Index.

• – ◦ • – ◦ • – ◦ •

Vergleich mit ähnlichen Bausteinen

Baustein	Änderungserkennung	Ausgabeverhalten
A2X_AUI_DEMUX_5	Ja	Werte werden nur bei Änderung weitergegeben
A2X_AUI_DEMUX_5_UNGATED	Nein	Jeder Wert wird ungefiltert weitergegeben

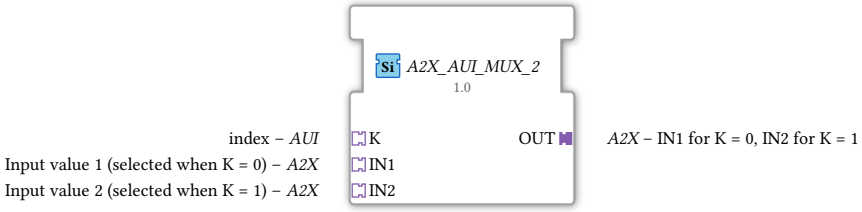
Gegenüber einem klassischen Demultiplexer mit getrennten Daten- und Ereignisports kapselt dieser Baustein die Datenpfade in Adaptern. Dadurch lassen sich Verbindungen modularer und typischerweise einfacher in der Applikation einsetzen.

• – ◦ • – ◦ • – ◦ •

Fazit

Der Baustein **A2X_AUI_DEMUX_5_UNGATED** ist ein adapterbasierter 5-fach-Demultiplexer ohne Änderungserkennung. Er eignet sich besonders für Verbraucher, die auch bei unveränderten Werten eine feste Versorgung mit Ereignissen und Daten benötigen. Die generische Ausführung macht ihn flexibel, solange das Backend GEN_A2X_AUI_DEMUX in der Zielumgebung verfügbar ist.

A2X_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **A2X_AUI_MUX_2** ist ein generischer 2-zu-1-Multiplexer auf Basis von Adapter-Schnittstellen. Er wählt zwischen zwei A2X-Adapter-Eingängen anhand eines über einen AUI-Adapter bereitgestellten Index aus und leitet den gewählten Wert an den A2X-Adapter-Ausgang weiter. Eine

Besonderheit ist die aktualisierungsoptimierte Arbeitsweise: Der Ausgang wird nur dann mit einem neuen Wert versorgt, wenn sich der ausgewählte Wert tatsächlich ändert.

Der Baustein ist als generischer Funktionsbaustein (Generic FB) angelegt. Die eigentliche Ausführungslogik wird über die zugeordnete generische Klasse `GEN_A2X_AUI_MUX` bereitgestellt und kann an die konkreten Adapter-Typen angepasst werden.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Name	Richtung	Typ	Kommentar
K	Socket	`adapter::types::unidirectional::AUI`	Index zur Auswahl des Eingangs.
IN1	Socket	`adapter::types::unidirectional::A2X`	Eingangswert 1, aktiv wenn K = 0.
IN2	Socket	`adapter::types::unidirectional::A2X`	Eingangswert 2, aktiv wenn K = 1.
OUT	Plug	`adapter::types::unidirectional::A2X`	Ausgang, liefert den ausgewählten Wert.

Funktionsweise

Der Baustein überwacht den über den Adapter **K** bereitgestellten Index. Der Indexwert bestimmt, welcher der beiden A2X-Eingänge zur Auswahl kommt:

- Bei **K = 0** wird der Wert von **IN1** zum Ausgang **OUT** durchgeschaltet.
- Bei **K = 1** wird der Wert von **IN2** zum Ausgang **OUT** durchgeschaltet.

Sobald sich entweder der Index K oder der aktuell ausgewählte Eingangswert ändert, wird der neue Wert mit dem zuletzt am Ausgang anliegenden Wert verglichen. Nur wenn sich der Wert tatsächlich unterscheidet, wird der Ausgangs-Adapter **OUT** mit dem neuen Wert aktualisiert. Bleibt der Wert unverändert, unterbleibt die Aktualisierung des Ausgangs. Dadurch werden unnötige Datenübertragungen und Ereignisfluten im nachgelagerten Netzwerk vermieden.

Technische Besonderheiten

- **Generischer Funktionsbaustein:** Die konkrete Verarbeitungslogik wird über die generische Klasse `GEN_A2X_AUI_MUX` bereitgestellt. Die Implementierung ist noch nicht vollständig im Standardumfang enthalten und muss für die jeweilige Zielumgebung bereitgestellt werden.
- **Änderungsbasierte Ausgabe:** Der Ausgang wird nur bei einer tatsächlichen Wertänderung aktualisiert. Dadurch wird das nachgeschaltete System entlastet und die Datenkonsistenz bei unveränderten Werten sichergestellt.
- **Adapterbasierte Schnittstellen:** Sämtliche Ein- und Ausgänge sind als unidirektionale Adapter ausgeführt. Dadurch ist der Baustein flexibel in verschiedene Adapter-Topologien integrierbar.
- **Zwei Eingänge, ein Index:** Die Auswahl erfolgt ausschließlich über den adapterbasierten Index K. Die Wertebereiche der Adapter-Typen A2X und AUI werden durch die jeweiligen Adapterdefinitionen festgelegt.

Zustandsübersicht

Der Baustein besitzt keine explizit modellierten Zustände. Die Funktionsweise kann jedoch als impliziter Ablauf beschrieben werden:

1. **Warten:** Der Baustein wartet auf Änderungen am Index K oder an den Eingangsadaptern IN1/IN2.
2. **Auswahl:** Der aktuelle Index K wird ausgewertet und der entsprechende Eingang ausgewählt.
3. **Vergleich:** Der neue Wert wird mit dem aktuell am Ausgang anliegenden Wert verglichen.
4. **Aktualisierung:** Bei Abweichung wird OUT aktualisiert. Bei Übereinstimmung bleibt der Zustand unverändert.

Anwendungsszenarien

- **Datenweiche:** Umschaltung zwischen zwei Datenquellen anhand eines externen Steuersignals, beispielsweise zur redundanten Signalzuführung.
- **Sensorumschaltung:** Auswahl zwischen zwei gleichartigen Sensoren, wobei nur eine Änderung des tatsächlich aktiven Sensors weitergereicht wird.
- **Effiziente Weiterleitung:** Einsatz in verzweigten Kommunikationsstrukturen, in denen unnötige Aktualisierungen vermieden werden sollen, etwa bei kontinuierlich gleichbleibenden Prozesswerten.
- **Generische Adapter-Integration:** Verwendung in Systemen, die auf die Adapter-Typen AUI und A2X aufbauen und einen austauschbaren Multiplexer benötigen.

Vergleich mit ähnlichen Bausteinen

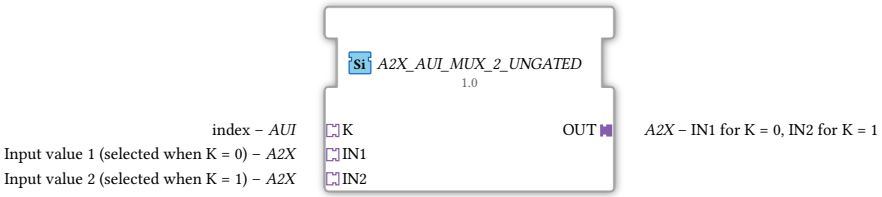
Im Gegensatz zu einfachen Multiplexern, die bei jeder Änderung eines Eingangs oder des Index ein Ereignis auslösen, arbeitet **A2X_AUI_MUX_2** ereignisoptimiert. Ein herkömmlicher Multiplexer würde den Ausgang bei jeder Neuberechnung aktualisieren, auch wenn sich der Wert nicht ändert. Dieser Baustein unterdrückt solche redundanten Ausgaben und aktualisiert den Ausgang nur dann, wenn der tatsächliche Wert wechselt.

Gegenüber einer nicht-generischen Variante ist dieser Baustein über die generische Klasse `GEN_A2X_AUI_MUX` an verschiedene Adapterausprägungen anpassbar. Dadurch ist er flexibler, erfordert jedoch eine angepasste Implementierung in der Zielumgebung.

Fazit

Der **A2X_AUI_MUX_2** ist ein zweikanaliger, adapterbasierter Multiplexer mit dem Fokus auf effizienter und ereignisreduzierter Werteweitergabe. Durch die Auswahl über den AUI-Adapter und die Ausgabe über einen A2X-Adapter fügt er sich nahtlos in adapterorientierte 4diac-Lösungen ein. Die Kombination aus generischer Struktur und änderungsbasierter Aktualisierung macht ihn besonders geeignet für Systeme, bei denen Kommunikationsaufkommen und Signalaktualität sorgfältig balanciert werden müssen.

A2X_AUI_MUX_2_UNGATED



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein `A2X_AUI_MUX_2_UNGATED` ist ein generischer Multiplexer, der zwei A2X-Eingänge über einen AUI-Index auswählt und das gewählte Signal über einen A2X-Ausgang bereitstellt. Im Gegensatz zur Variante `A2X_AUI_MUX_2` besitzt dieser Baustein keine Änderungserkennung: Jedes neu berechnete Ergebnis wird unabhängig davon, ob sich der Wert geändert hat, bedingungslos weitergegeben. Dadurch eignet er sich besonders für Anwendungen, die eine periodische Kadenz benötigen, etwa Ableitungs- oder Frequenzberechnungen.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine Ereignis-Eingänge. Die Steuerung erfolgt ausschließlich über die Adapter-Schnittstellen.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge.

Daten-Eingänge

Der Baustein besitzt keine direkten Daten-Eingänge. Alle Daten werden über die Adapter-Schnittstellen `IN1` und `IN2` sowie den Auswahl-Adapter `K` übergeben.

Daten-Ausgänge

Der Baustein besitzt keine direkten Daten-Ausgänge. Der ausgewählte Wert wird über den Adapter-Ausgang `OUT` bereitgestellt.

Adapter

Adapter	Richtung	Typ	Kommentar
`K`	Socket (Eingang)	`adapter::types::unidirectional::AUI`	Auswahl-Index: Bei K = 0 wird IN1 gewählt, bei K = 1 wird IN2 gewählt.
`IN1`	Socket (Eingang)	`adapter::types::unidirectional::A2X`	Erster Eingangswert (ausgewählt bei K = 0).
`IN2`	Socket (Eingang)	`adapter::types::unidirectional::A2X`	Zweiter Eingangswert (ausgewählt bei K = 1).
`OUT`	Plug (Ausgang)	`adapter::types::unidirectional::A2X`	Ausgang mit dem ausgewählten Eingangswert.

Funktionsweise

Der Baustein empfängt über den Adapter `K` einen Auswahl-Index. Abhängig von diesem Index wird der Wert von `IN1` (bei K = 0) oder von `IN2` (bei K = 1) an den Ausgang `OUT` durchgereicht.

Die Kennzeichnung `UNGATED` bedeutet, dass keine Änderungserkennung zwischen altem und neuem Wert stattfindet. Jeder am ausgewählten Eingang anliegende neue Datenwert wird sofort und vollständig an den Ausgang weitergegeben. Es findet keine Filterung identischer Werte statt. Dadurch wird sichergestellt, dass nachfolgende Verbraucher, die auf eine gleichmäßige zeitliche Abfolge angewiesen sind, kontinuierlich mit Daten versorgt werden.

Technische Besonderheiten

- **Generischer Baustein**: Der Baustein ist als generischer FB mit dem Klassennamen `GEN_A2X_AUI_MUX` definiert. Der tatsächliche Typ wird erst bei der Instanziierung im Projekt aufgelöst.
- **Reine Adapter-Schnittstelle**: Es existieren keine klassischen Ereignis- oder Daten-Pins. Die gesamte Kommunikation läuft über die Adapter `K`, `IN1`, `IN2` und `OUT`.
- **Unidirektionale Adapter**: Alle verwendeten Adapter sind vom Typ `unidirectional`, d.h. die Datenflussrichtung ist fest vorgegeben und es findet keine Rückkanal-Kommunikation statt.
- **Keine Änderungserkennung**: Identische Datenwerte werden dennoch als Ereignis weitergegeben. Dies unterscheidet den Baustein von der Variante mit Änderungserkennung.

Zustandsübersicht

Der Baustein lässt sich konzeptionell in drei Zustände gliedern:

1. **Bereit**: Der Baustein wartet auf einen neuen Auswahl-Index über den Adapter `K`.
2. **Auswahl**: Der an `K` anliegende Index wird ausgewertet und der entsprechende Eingang (`IN1` oder `IN2`) ausgewählt.

Eine interne Speicherung vorheriger Werte oder ein Vergleich mit dem aktuellen Wert findet nicht statt.

Anwendungsszenarien

- **Periodische Datenweitergabe**: Wenn ein nachgeschalteter Baustein eine feste Datenrate benötigt, unabhängig davon, ob sich der Messwert ändert.
- **Ableitungs- und Frequenzberechnungen**: Hier werden zeitliche Änderungen benötigt; ein Auslassen identischer Werte würde die Berechnung verfälschen.
- **Umschaltung zwischen zwei Signalquellen**: Der Baustein kann verwendet werden, um zwischen zwei Sensoren oder Datenströmen umzuschalten, wobei jede neue Messung am ausgewählten Eingang sofort weitergeleitet wird.

- **Test- und Diagnosezwecke**: Wenn sichergestellt werden soll, dass jede Berechnung am Ausgang ankommt, auch wenn sich der Wert nicht ändert.

Vergleich mit ähnlichen Bausteinen

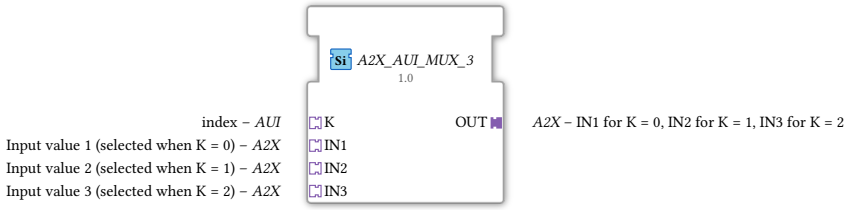
Baustein	Änderungserkennung	Ausgabe identischer Werte
`A2X_AUI_MUX_2`	Ja, nur bei Wertänderung wird ausgegeben.	Nein, identische Werte werden unterdrückt.
`A2X_AUI_MUX_2_UNGATED`	Nein, jede neue Berechnung wird ausgegeben.	Ja, auch bei unverändertem Wert wird ein Ereignis erzeugt.

Der wesentliche Unterschied liegt also in der Behandlung unveränderter Werte. Während der normale Multiplexer die Ausgabe bei unverändertem Wert unterdrückt, gibt die `UNGATED`-Variante das Ergebnis immer aus.

Fazit

Der Funktionsbaustein `A2X_AUI_MUX_2_UNGATED` ist eine spezialisierte Variante eines 2-zu-1-Multiplexers mit Adapter-basierter Kommunikation. Durch den Verzicht auf eine Änderungserkennung wird eine lückenlose und kadenzgenaue Datenweitergabe ermöglicht. Er ist damit ideal für Anwendungen, die auf eine kontinuierliche Verarbeitung angewiesen sind und keine Werte auslassen dürfen.

A2X_AUI_MUX_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der A2X_AUI_MUX_3 ist ein generischer Multiplexer-Baustein auf Basis unidirektionaler Adapter. Er wählt anhand eines Auswahlindexes K einen von drei A2X-Eingängen aus und stellt den zugehörigen Wert am Ausgang OUT bereit. Der Ausgang wird nur bei einer tatsächlichen Wertänderung aktualisiert. Dadurch eignet er sich für wartungsarme und ereignisreduzierte Signalumschaltungen in IEC-61499-Applikationen.

Schnittstellenstruktur

Der Baustein besitzt keine separaten Daten-Ein- oder Daten-Ausgänge. Alle Werte und der Auswahlindex werden über unidirektionale Adapteranschlüsse übertragen.

Ereignis-Eingänge

Es sind keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge.

Daten-Eingänge

Keine. Die Eingangswerte werden über die Adapter IN1, IN2 und IN3 eingelesen.

Daten-Ausgänge

Keine. Der Ausgangswert wird über den Adapter OUT bereitgestellt.

Adapter

Name	Richtung	Typ	Beschreibung

K	Socket	adapter::types::unidirectional::AUI	Auswahlindex: K = 0 ⟶ IN1, K = 1 ⟶ IN2, K = 2 ⟶ IN3
IN1	Socket	adapter::types::unidirectional::A2X	Erster Eingangswert, ausgewählt bei K = 0
IN2	Socket	adapter::types::unidirectional::A2X	Zweiter Eingangswert, ausgewählt bei K = 1
IN3	Socket	adapter::types::unidirectional::A2X	Dritter Eingangswert, ausgewählt bei K = 2
OUT	Plug	adapter::types::unidirectional::A2X	Ausgang; liefert den Wert des ausgewählten Eingangs

Funktionsweise

Der A2X_AUI_MUX_3 arbeitet als gesteuerter Multiplexer:

1. Über den AUI-Adapter K wird der gewünschte Eingangskanal ausgewählt.
2. Der Wert des gewählten A2X-Eingangs wird mit dem aktuellen Ausgangswert an OUT verglichen.
3. Bei einer Abweichung wird OUT auf den neuen Wert gesetzt.
4. Bleibt der Wert unverändert, wird OUT nicht aktualisiert.

Die Werteübertragung erfolgt ausschließlich über die unidirektionalen Adapter. Der FB benötigt keinen separaten Ereignis-Eingang, da die Verarbeitung über die an den Adapter-Sockets anliegenden Werte angestoßen wird.

Technische Besonderheiten

- Der Baustein ist als generischer Funktionsblock deklariert und verwendet den GenericClassName `'GEN_A2X_AUI_MUX'`.
- Er ist dem Paket `adapter::selection::unidirectional` zugeordnet.
- Die Kommunikation erfolgt vollständig über Adapter; klassische Datenports werden nicht verwendet.
- Der Ausgang wird nur bei tatsächlicher Wertänderung aktualisiert.
- Der Auswahlindex K ist für die Werte 0, 1 und 2 definiert. Werte außerhalb dieses Bereichs müssen in der Anwendung vermieden werden.

Zustandsübersicht

Die XML-Deklaration des Bausteins enthält keine explizite Zustandsmaschine. Das Laufzeitverhalten wird durch das generische Backend `GEN_A2X_AUI_MUX` bereitgestellt. Ablauflogisch lässt sich das Verhalten wie folgt beschreiben:

Zustand	Beschreibung	Aktion
IDLE	Warten auf einen neuen Auswahlindex oder einen geänderten Eingangswert	Keine Ausgabe
COMPARE	Vergleich des gewählten Eingangswerts mit dem aktuellen OUT-Wert	Keine Ausgabe
UPDATE	Der neue Wert wird an OUT übernommen	OUT wird aktualisiert
NO_UPDATE	Kein Unterschied zwischen gewähltem Eingangswert und OUT	OUT bleibt unverändert

Anwendungsszenarien

- **Signalumschaltung:** Auswahl zwischen drei verschiedenen Sensor- oder Prozesssignalen über den Index K.
- **Betriebsartenwahl:** K = 0 für Normalbetrieb, K = 1 für Wartungsmodus, K = 2 für Testsignal.
- **Redundanzumschaltung:** Bei Ausfall eines Sensors kann per K auf einen Ersatzsensor umgeschaltet werden.
- **Simulation und Test:** Umschaltung zwischen realen Prozesswerten und simulierten Werten in einer Anlage.

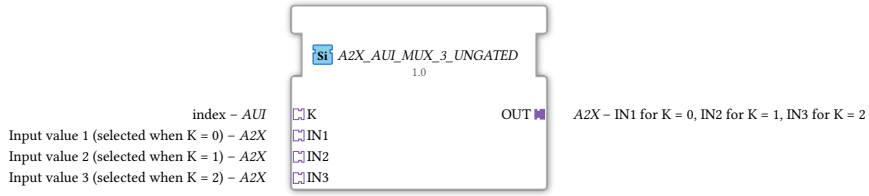
Vergleich mit ähnlichen Bausteinen

Baustein	Eigenschaft
A2X_AUI_MUX_2	Multiplexer-Variante mit zwei A2X-Eingängen
A2X_AUI_MUX_3	Drei Eingänge, AUI-Indexauswahl, Change-only-Ausgabe
A2X_AUI_MUX_4	Multiplexer-Variante mit vier A2X-Eingängen
AX_AUI_MUX_3	Variante mit anderem Adaptertyp; ohne Change-only-Verhalten oder mit abweichender Ereignislogik
A2X_AUI_DEMUX	Inverse Funktion: Verteilt einen Eingangswert auf mehrere Ausgänge

Fazit

Der A2X_AUI_MUX_3 realisiert eine kompakte, adapterbasierte 3-zu-1-Auswahl für IEC-61499-Anwendungen. Durch die Aktualisierung nur bei tatsächlicher Wertänderung eignet er sich besonders für effiziente, ereignisarme und modular aufgebaute Automatisierungslösungen.

A2X_AUI_MUX_3_UNGATED



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **A2X_AUI_MUX_3_UNGATED** ist ein generischer Multiplexer auf Adapterbasis. Er wählt aus drei A2X-Eingangsadaptern einen Datenstrom aus und leitet ihn an einen A2X-Ausgangsadapter weiter. Die Auswahl erfolgt über den AUI-Adapter `K`.

Im Gegensatz zur Variante mit Änderungserkennung besitzt dieser Baustein keine „Gating“-Logik: Jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, unabhängig davon, ob sich der Wert geändert hat. Dadurch eignet er sich besonders für Verbraucher, die eine periodische Kadenz benötigen, etwa für Ableitungs- oder Frequenzberechnungen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge.

Daten-Eingänge

Keine. Die Datenübertragung erfolgt vollständig über Adapter.

Daten-Ausgänge

Keine. Die Datenübertragung erfolgt vollständig über Adapter.

Adapter

Name	Typ	Richtung	Kommentar
`OUT`	`adapter::types::unidirectional::A2X`	Plug	Ausgangsadapter; liefert `IN1` bei `K = 0`, `IN2` bei `K = 1`, `IN3` bei `K = 2`

K	adapter::types::unidirectional::AUI	Socket	Indexadapter zur Auswahl des aktiven Eingangs
IN1	adapter::types::unidirectional::A2X	Socket	Eingangswert 1, ausgewählt bei K = 0
IN2	adapter::types::unidirectional::A2X	Socket	Eingangswert 2, ausgewählt bei K = 1
IN3	adapter::types::unidirectional::A2X	Socket	Eingangswert 3, ausgewählt bei K = 2

Funktionsweise

Der Baustein arbeitet als Adapter-Multiplexer:

1. Über den AUI-Adapter K wird ein Index empfangen.
2. Abhängig vom Index wird einer der drei A2X-Eingänge ausgewählt:
   - K = 0 ⟶ IN1
   - K = 1 ⟶ IN2
   - K = 2 ⟶ IN3
3. Die Daten des ausgewählten Eingangs werden auf den Ausgangsadapter OUT durchgeschaltet.

Jedes an einem aktiven Eingang ankommende Ergebnis wird weitergeleitet. Eine Prüfung, ob sich der Wert gegenüber der vorherigen Übertragung geändert hat, findet nicht statt.

Technische Besonderheiten

- **Keine Änderungserkennung:** Die Bezeichnung UNGATED bedeutet, dass keine Wertänderungserkennung erfolgt.
- **Reine Adapter-Schnittstelle:** Es gibt weder Daten-Ein- noch Daten-Ausgänge; der Datentransport erfolgt ausschließlich über unidirektionale Adapter.
- **Generischer Funktionsbaustein:** Der Baustein ist als generischer FB deklariert und verwendet die generische Klasse GEN_A2X_AUI_MUX.
- **Kein Ereignis-Eingang:** Die Steuerung erfolgt nicht über klassische Event-Eingänge, sondern über die Adapterverbindungen.
- **Periodische Weitergabe:** Durch das Fehlen einer Änderungserkennung eignet sich der Baustein für Anwendungen, die eine kontinuierliche, periodische Datenweitergabe benötigen.

Zustandsübersicht

Der Baustein besitzt keine explizit ausgeprägte Zustandsmaschine. Der Auswahlzustand wird durch den Wert von K bestimmt:

K	ausgewählter Eingang
0	IN1
1	IN2
2	IN3

Gültige Indexwerte sind 0, 1 und 2. Werte außerhalb dieses Bereichs sind in der Schnittstellenbeschreibung nicht definiert.

Anwendungsszenarien

- **Sensorumschaltung:** Auswahl eines von drei analogen Messwerten für einen gemeinsamen Verarbeitungspfad.
- **Zyklische Datenweitergabe:** Weitergabe jedes Berechnungsergebnisses mit fester Kadenz, auch wenn sich der Wert nicht ändert.
- **Ableitungs- und Frequenzberechnung:** Einsatz vor Bausteinen, die aus zeitlichen Wertfolgen Differenzen oder Frequenzen berechnen.
- **Adapterbasiertes Multiplexing:** Strukturierte Kopplung mehrerer A2X-Datenquellen an einen einzelnen Ausgangsadapter.

Vergleich mit ähnlichen Bausteinen

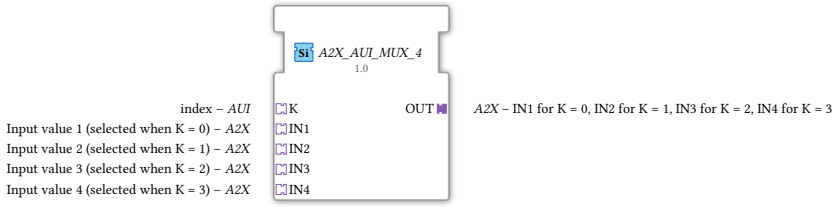
Baustein	Änderungserkennung	Datenadapter	Besonderheit

A2X_AUI_MUX_3	Ja	A2X	Gibt Werte nur bei Änderung weiter.
A2X_AUI_MUX_3_UNGATED	Nein	A2X	Gibt jedes neu berechnete Ergebnis bedingungslos weiter.
AX_AUI_MUX_3_UNGATED	Nein	AX	Variante mit anderem Datenadaptertyp.

Fazit

A2X_AUI_MUX_3_UNGATED ist ein flexibler, adapterbasierter Multiplexer für drei A2X-Datenströme. Durch den Verzicht auf eine Änderungserkennung liefert er jedes neue Ergebnis sofort weiter. Damit ist er ideal für Anwendungen, die eine kontinuierliche, periodische Datenverarbeitung erfordern und keine Stillstandszeiten durch unveränderte Werte erlauben.

A2X_AUI_MUX_4



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock **A2X_AUI_MUX_4** ist ein generischer 4-zu-1-Multiplexer auf Basis unidirektionaler IEC-61499-Adapter. Er wählt über den Indexadapter **K** genau einen der vier Eingangsadapter **IN1** bis **IN4** aus und stellt dessen Wert am Ausgangsadapter **OUT** bereit.

Der Baustein ist als generischer FB mit dem Klassennamen GEN_A2X_AUI_MUX angelegt. Er ist speziell für den Einsatz mit **A2X**-Adaptern vorgesehen und aktualisiert den Adapter-Ausgang nur bei tatsächlicher Wertänderung des ausgewählten Eingangs.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge.

Daten-Eingänge

Keine direkten Dateneingänge. Die Eingangsdaten werden ausschließlich über Adapter-Sockets übertragen.

Daten-Ausgänge

Keine direkten Datenausgänge. Die Ausgangsdaten werden ausschließlich über den Adapter-Plug OUT bereitgestellt.

Adapter

Name	Richtung	Typ	Beschreibung
OUT	Plug	adapter::types::unidirectional::A2X	Ausgangsadapter mit dem gewählten Wert.
K	Socket	adapter::types::unidirectional::AUI	Indexadapter zur Auswahl des aktiven Eingangs.

IN1	Socket	adapter::types::unidirectional::A2X	Eingangswert 1, aktiv für K = 0.
IN2	Socket	adapter::types::unidirectional::A2X	Eingangswert 2, aktiv für K = 1.
IN3	Socket	adapter::types::unidirectional::A2X	Eingangswert 3, aktiv für K = 2.
IN4	Socket	adapter::types::unidirectional::A2X	Eingangswert 4, aktiv für K = 3.

Funktionsweise

Der Baustein arbeitet als adapterbasierter Multiplexer:

1. Der Index K legt fest, welcher der vier Eingänge ausgewählt wird.
2. Bei K = 0 wird IN1 ausgewählt, bei K = 1 IN2, bei K = 2 IN3 und bei K = 3 IN4.
3. Der Wert des ausgewählten Eingangsadapter wird auf den Ausgangsadapter OUT übertragen.
4. OUT wird nur dann aktualisiert, wenn sich der Wert gegenüber der letzten Ausgabe tatsächlich geändert hat.

Technische Besonderheiten

- Der FB ist als generischer Baustein mit dem Attribut eclipse4diac::core::GenericClassName vom Typ GEN_A2X_AUI_MUX definiert.
- Die Schnittstellen bestehen vollständig aus unidirektionalen Adaptern der Typen A2X und AUI; es gibt keine klassischen Datenein- und -ausgänge.
- Die Ausgabe erfolgt ausschließlich bei Wertänderung. Dadurch wird unnötige Kommunikation auf dem Adapter-Ausgang vermieden.
- Die Versionsinformation weist darauf hin, dass das konkrete Adapter-Backend GEN_A2X_AUI_MUX noch zu implementieren ist.
- Der Baustein ist für die Paketstruktur adapter::selection::unidirectional vorgesehen.

Zustandsübersicht

In der XML-Definition ist keine explizite ECC-Zustandsmaschine enthalten. Funktional lässt sich das Verhalten jedoch in folgende logische Zustände gliedern:

Zustand	Beschreibung	Ausgangsverhalten
Bereit	Der FB wartet auf einen gültigen Index K und prüft die Eingänge.	Kein Ereignis.
Vergleich	Der Wert des ausgewählten Eingangs wird mit dem zuletzt ausgegebenen Wert verglichen.	Kein Ereignis.
Aktualisierung	Eine Wertänderung wurde erkannt; OUT wird gesetzt.	OUT wird aktualisiert.
Unverändert	Der Wert entspricht dem letzten Ausgangswert.	Kein Ereignis, OUT bleibt unverändert.

Anwendungsszenarien

- **Signalumschaltung**: Auswahl einer von vier A2X-Datenquellen, z. B. unterschiedliche Sensoren, auf einen gemeinsamen Ausgang.
- **Redundanzumschaltung**: Umschalten zwischen mehreren redundanten Datenpfaden, ohne permanente Ausgangsereignisse zu erzeugen.
- **Parametrierbare Datenauswahl**: Nutzung eines AUI-Indexes zur dynamischen Auswahl eines Datenstroms in einer Produktions- oder Steuerungsanlage.
- **Generischer Einsatz**: Durch die generische Klassenbindung kann der FB für verschiedene Datenformate verwendet werden, sobald das Backend GEN_A2X_AUI_MUX implementiert ist.

Vergleich mit ähnlichen Bausteinen

Gegenüber einem klassischen Daten-MUX besitzt A2X_AUI_MUX_4 keine direkten Eingangs- und Ausgangsdaten, sondern nutzt ausschließlich unidirektionale Adapter. Dadurch können auch komplexe oder typisierte Datenstrukturen übertragen werden.

Im Unterschied zu einer einfachen AX_AUI_MUX_4-Variante ist dieser Baustein speziell für **A2X**-Adapter angepasst. Zusätzlich wird der Ausgang nur bei tatsächlicher Änderung aktualisiert, wodurch Ereignis- und Kommunikationsaufwand reduziert werden.

Fazit

A2X_AUI_MUX_4 ist ein flexibler, adapterbasierter 4-zu-1-Multiplexer für die IEC-61499-Umgebung. Er reduziert durch die Wertänderungserkennung unnötige Ausgangsereignisse und eignet sich besonders für die dynamische Auswahl von A2X-Datenströmen. Vor dem produktiven Einsatz muss das zugehörige generische Backend GEN_A2X_AUI_MUX implementiert werden.

A2X_AUI_MUX_4_UNGATED

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **A2X_AUI_MUX_4_UNGATED** ist ein 4-zu-1-Multiplexer auf Adapterbasis. Er wählt über den als Index dienenden **AUI**-Adapter **K** einen von vier **A2X**-Eingängen aus und leitet dessen Daten beziehungsweise Ereignisse an den Ausgangs-Adapter **OUT** weiter.

Die Bezeichnung **UNGATED** kennzeichnet eine Variante ohne Änderungserkennung: Im Gegensatz zu einem gated Multiplexer wird hier **jedes neu berechnete Ergebnis** bedingungslos weitergegeben. Das ist für Verbraucher gedacht, die eine periodische Kadenz benötigen, unabhängig davon, ob sich ein Wert tatsächlich geändert hat – beispielsweise bei Ableitungs- oder Frequenzberechnungen.

• – ◦ • – ◦ • – ◦ •

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine direkten Ereignis-Eingänge. Die auslösenden Ereignisse werden über die Adapter-Sockets transportiert.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Nutzdaten werden über die **A2X**-Adapter-Sockets IN1 bis IN4 sowie über den **AUI**-Adapter K eingelesen.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Der ausgewählte Datenstrom wird über den **A2X**-Adapter-Plug OUT ausgegeben.

Adapter

Name	Richtung	Typ	Bedeutung
K	Socket	adapter::types::unidirectional::AUI	Auswahlindex.
IN1	Socket	adapter::types::unidirectional::A2X	Eingangswert 1, ausgewählt bei K = 0.
IN2	Socket	adapter::types::unidirectional::A2X	Eingangswert 2, ausgewählt bei K = 1.
IN3	Socket	adapter::types::unidirectional::A2X	Eingangswert 3, ausgewählt bei K = 2.
IN4	Socket	adapter::types::unidirectional::A2X	Eingangswert 4, ausgewählt bei K = 3.

OUT	Plug	adapter::types::unidirectional::A2X	Ausgang mit dem ausgewählten A2X-Datenstrom.

Alle Adapter sind unidirektional. Der Datenfluss verläuft von den Sockets IN1–IN4 und K zum Plug OUT.

• – ◦ • – ◦ • – ◦ •

Funktionsweise

Der Baustein verhält sich wie ein klassischer 4-zu-1-Multiplexer:

1. Der **AUI**-Socket K liefert einen Indexwert.
2. Anhand dieses Indexes wird einer der vier **A2X**-Sockets IN1–IN4 ausgewählt.
3. Das über diesen Socket ankommende Adapter-Paket wird unverändert an den **A2X**-Plug OUT weitergegeben.

Die entscheidende Eigenschaft ist das **Fehlen einer Änderungserkennung**. Der Baustein vergleicht keinen alten mit einem neuen Wert. Jedes ankommende Ereignis und jedes neu berechnete Ergebnis wird sofort und ohne Prüfung weitergereicht. Dadurch bleibt auch dann ein kontinuierlicher Datenstrom erhalten, wenn sich der Wert über mehrere Zyklen nicht ändert.

• – ◦ • – ◦ • – ◦ •

Technische Besonderheiten

- **Keine Änderungserkennung:** Der Baustein unterdrückt keine unveränderten Werte. Jedes Ereignis wird durchgeschaltet.
- **Adapterbasierte Ein-/Ausgabe:** Es gibt keine klassischen Datenein- oder Datenausgänge; alle Werte laufen über die A2X- beziehungsweise AUI-Adapter.
- **Generischer Baustein:** Die tatsächliche Implementierung wird über das Attribut eclipse4diac::core::GenericClassName mit GEN_A2X_AUI_MUX referenziert.
- **Unidirektionale Adapter:** Der Baustein verwendet ausschließlich unidirektionale Adapter, was eine klare, gerichtete Datenflussstruktur ermöglicht.
- **Indexbereich:** Der Auswahlindex K wird entsprechend den Kommentaren als Wert 0 bis 3 erwartet. Werte außerhalb dieses Bereichs sind nicht definiert.

• – ◦ • – ◦ • – ◦ •

Zustandsübersicht

Der Baustein ist als generischer Funktionsblock ohne eigenes, in der XML abgebildetes ECC definiert. Konzeptionell lassen sich folgende Phasen beschreiben:

Zustand	Beschreibung
Bereit	Der Baustein wartet auf einen gültigen Index über den AUI-Adapter K.
Auswahl	Der Index wird ausgewertet und der zugehörige Eingang IN1–IN4 ausgewählt.
Durchreichen	Das Datenpaket des gewählten Eingangs wird unabhängig von einer Wertänderung an OUT weitergegeben.

Es gibt keinen Zustand, der eine Weitergabe aufgrund unveränderter Werte blockiert.

• – ◦ • – ◦ • – ◦ •

Anwendungsszenarien

- **Ableitungsberechnung:** Für die Bildung eines Differenzenquotienten muss jeder Abtastwert verarbeitet werden, auch wenn der Messwert konstant bleibt. Nur so kann eine saubere Ableitung von Null taktrichtig ausgegeben werden.
- **Frequenz- und Periodendauerermittlung:** Verbraucher benötigen eine lückenlose Ereignisfolge, um Zeitintervalle zuverlässig zu messen.
- **Datenquellen-Umschaltung:** In einer Regelung oder Visualisierung können verschiedene A2X-Datenquellen über den Index K umgeschaltet werden, ohne dass die Ausgabe bei gleichen Werten verzögert wird.

- **Test- und Prüfumgebungen:** Jede Berechnung eines vorgelagerten Bausteins soll protokolliert oder weiterverarbeitet werden, unabhängig davon, ob sich der Wert geändert hat.

• – ◦ • – ◦ • – ◦ •

Vergleich mit ähnlichen Bausteinen

Baustein	Verhalten	Geeignet für
`A2X_AUI_MUX_4`	Multiplexer mit Änderungserkennung; leitet nur bei Wertänderung weiter.	Verbraucher, die ausschließlich auf geänderte Werte reagieren sollen.
`A2X_AUI_MUX_4_UNGATED`	Multiplexer ohne Änderungserkennung; leitet jedes Ereignis bedingungslos weiter.	Zeitkontinuierliche Auswertungen, Ableitungen, Frequenzberechnungen, lückenlose Protokollierung.

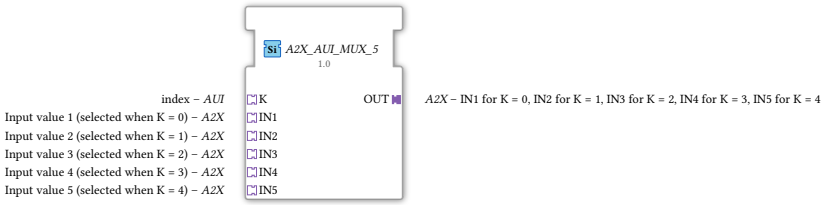
Die grundlegende Multiplexer-Funktion ist identisch. Der einzige wesentliche Unterschied liegt in der Filterung durch die Änderungserkennung.

• – ◦ • – ◦ • – ◦ •

Fazit

Der Funktionsblock **A2X_AUI_MUX_4_UNGATED** ist eine sinnvolle Ergänzung zur variantenreichen A2X-Multiplexer-Familie. Er bietet eine einfache, adapterbasierte 4-zu-1-Auswahl und verzichtet bewusst auf jede Änderungserkennung. Dadurch eignet er sich besonders für Anwendungen, die eine periodische, lückenlose und ungefilterte Weitergabe von berechneten Ergebnissen benötigen.

A2X_AUI_MUX_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **A2X_AUI_MUX_5** ist ein generischer Multiplexer auf Basis unidirektionaler Adapter. Er wählt anhand eines über den **AUI**-Adapter zugeführten Indexes einen von fünf **A2X**-Adaptereingängen aus und leitet dessen Wert über einen **A2X**-Adapterausgang weiter. Der Ausgang wird dabei nur bei einer tatsächlichen Wertänderung aktualisiert.

Schnittstellenstruktur

Der Baustein besitzt keine klassischen Datenein- oder Datenausgänge. Alle Werte werden über Adapter übertragen. Ereignis-Ein- und -Ausgänge sind nicht vorhanden.

Ereignis-Eingänge

Es sind keine Ereignis-Eingänge definiert.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge.

Daten-Eingänge

Es sind keine Daten-Eingänge vorhanden. Die Eingabedaten werden ausschließlich über die Adapter `IN1` bis `IN5` bereitgestellt.

Daten-Ausgänge

Es sind keine Daten-Ausgänge vorhanden. Der Ausgabewert wird ausschließlich über den Adapter `OUT` bereitgestellt.

Adapter

Adapter	Richtung	Typ	Kommentar
OUT	Ausgang (Plug)	adapter::types::unidirectional::A2X	Ausgangswert; entspricht dem über K ausgewählten Eingang
K	Eingang (Socket)	adapter::types::unidirectional::AUI	Indexauswahl: 0 bis 4
IN1	Eingang (Socket)	adapter::types::unidirectional::A2X	Eingangswert 1, ausgewählt bei K = 0
IN2	Eingang (Socket)	adapter::types::unidirectional::A2X	Eingangswert 2, ausgewählt bei K = 1
IN3	Eingang (Socket)	adapter::types::unidirectional::A2X	Eingangswert 3, ausgewählt bei K = 2
IN4	Eingang (Socket)	adapter::types::unidirectional::A2X	Eingangswert 4, ausgewählt bei K = 3
IN5	Eingang (Socket)	adapter::types::unidirectional::A2X	Eingangswert 5, ausgewählt bei K = 4

Funktionsweise

Der Baustein arbeitet als adapterbasierter Multiplexer:

1. Der K-Adapter liefert einen Indexwert im Bereich 0 bis 4.
2. Anhand dieses Indexes wird einer der fünf A2X-Eingänge ausgewählt:
   - K = 0 ⟶ IN1
   - K = 1 ⟶ IN2
   - K = 2 ⟶ IN3
   - K = 3 ⟶ IN4
   - K = 4 ⟶ IN5
3. Der Wert des ausgewählten Eingangs wird auf den OUT-Adapter übertragen.
4. Die Übertragung auf OUT erfolgt nur dann, wenn sich der resultierende Wert gegenüber dem zuletzt gesendeten Wert tatsächlich geändert hat.

Der Baustein reagiert damit nicht nur auf eine Änderung des Indexes, sondern auch auf eine Wertänderung am aktuell ausgewählten Eingang.

Technische Besonderheiten

- Der Baustein ist als **generischer Funktionsbaustein** ausgelegt. Über das Attribut eclipse4diac::core::GenericClassName wird die generische Implementierungsklasse GEN_A2X_AUI_MUX referenziert.
- Die Datenübertragung erfolgt vollständig über **unidirektionale Adapter** vom Typ A2X und AUI. Es sind keine separaten Datenein- oder Datenausgänge vorhanden.
- Die Ausgangsaktualisierung erfolgt **ereignisgesteuert bei Wertänderung**. Dadurch werden unnötige Aktualisierungen am Adapterausgang vermieden.
- Der zulässige Indexbereich ist durch die Adapterkommentare mit 0 bis 4 festgelegt; für jeden Index ist genau einer der fünf Eingänge vorgesehen.
- In der vorliegenden Version ist die generische Backend-Implementierung GEN_A2X_AUI_MUX als noch zu implementieren gekennzeichnet. Für den Einsatz des Bausteins muss dieses Backend in der Zielumgebung verfügbar sein.

Zustandsübersicht

Ein explizites Zustandsdiagramm ist in der XML nicht hinterlegt. Logisch lässt sich das Verhalten des Bausteins in zwei Zustände unterteilen:

Zustand	Beschreibung

WARTEN	Der Baustein wartet auf eine Änderung des Indexes K oder auf eine Wertänderung des aktuell ausgewählten Eingangs.
AKTUALISIEREN	Bei einer tatsächlichen Wertänderung wird der neue Wert auf OUT übertragen.

Ein Wechsel in den Zustand AKTUALISIEREN findet nur statt, wenn sich der auszugebende Wert tatsächlich ändert. Andernfalls verbleibt der Baustein im Zustand WARTEN.

Anwendungsszenarien

- **Adapterbasierte Signalumschaltung:** Auswahl eines von fünf analogen oder digitalen A2X-Signalen über einen zentralen Indexadapter.
- **Modulare Maschinensteuerung:** Dynamisches Durchschalten verschiedener Sensor- oder Steuerwerte an einen gemeinsamen Ausgangsadapter.
- **Redundante Wertauswahl:** Umschaltung zwischen mehreren Datenquellen, ohne dass die angeschlossenen Bausteine explizite Ereigniseingänge bereitstellen müssen.
- **Vermeidung unnötiger Ausgangsaktivität:** Durch die Aktualisierung nur bei Wertänderung bleiben nachgeschaltete Bausteine weitgehend ruhig, solange sich der ausgewählte Wert nicht ändert.

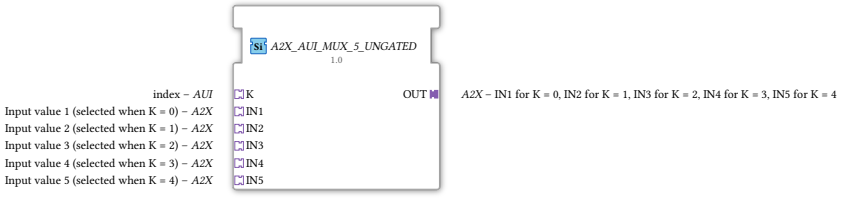
Vergleich mit ähnlichen Bausteinen

Baustein	Eigenschaft
A2X_AUI_MUX_5	Adapterbasierter Multiplexer mit unidirektionalen A2X- und AUI-Adaptern; Ausgang nur bei Wertänderung aktiv.
AX_AUI_MUX_5	Verwandte Variante ohne die spezielle A2X-Adapteranbindung. A2X_AUI_MUX_5 stellt eine darauf aufbauende Variante dar.
Konventioneller MUX-Funktionsbaustein	Klassische Multiplexer verwenden in der Regel diskrete Dateneingänge und Ereigniseingänge. Der vorliegende Baustein nutzt dagegen ausschließlich Adapter zur Datenübertragung.

Fazit

A2X_AUI_MUX_5 ist ein kompakter, adapterbasierter Multiplexer für bis zu fünf A2X-Eingänge. Durch die Auswahl über einen AUI-Index und die Aktualisierung nur bei tatsächlicher Wertänderung eignet er sich besonders für modulare, ereignisarme Adapterverbindungen in IEC-61499-Systemen. Der generische Aufbau ermöglicht eine flexible Wiederverwendung, sobald das zugehörige Backend GEN_A2X_AUI_MUX verfügbar ist.

A2X_AUI_MUX_5_UNGATED



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **A2X_AUI_MUX_5_UNGATED** ist ein unidirektionaler 5:1-Multiplexer auf Basis von IEC-61499-Adaptern. Er wählt anhand des Indexes K einen von fünf A2X-Eingängen (IN1 bis IN5) aus und gibt dessen Wert unverändert über den A2X-Ausgang OUT weiter.

Die Bezeichnung UNGATED kennzeichnet die besondere Eigenschaft dieses Bausteins: Er besitzt **keine Änderungserkennung**. Jedes neu berechnete Ergebnis des ausgewählten Eingangs wird bedingungslos weitergegeben, auch wenn der Wert gegenüber dem vorherigen Zyklus unverändert ist. Dadurch eignet er sich für Verbraucher, die eine periodische Kadenz benötigen, beispielsweise für Ableitungs- oder Frequenzberechnungen.

Schnittstellenstruktur

Ereignis-Eingänge

In der XML sind keine direkten Ereignis-Eingänge deklariert. Ereignisse zur Übertragung von Werten werden über die verwendeten Adaptertypen transportiert.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Nutzdaten werden über die A2X-Adapter `IN1` bis `IN5` sowie über den AUI-Adapter `K` eingelesen.

Daten-Ausgänge

Es sind keine direkten Daten-Ausgänge vorhanden. Der Ausgangswert wird über den A2X-Adapter `OUT` bereitgestellt.

Adapter

Name	Richtung	Typ	Beschreibung
`OUT`	Plug / Ausgang	`adapter::types::unidirectional::A2X`	Ausgangswert; liefert den Wert des angewählten Eingangs
`K`	Socket / Eingang	`adapter::types::unidirectional::AUI`	Index zur Auswahl des aktiven Eingangs
`IN1`	Socket / Eingang	`adapter::types::unidirectional::A2X`	Eingangswert 1, ausgewählt bei `K = 0`
`IN2`	Socket / Eingang	`adapter::types::unidirectional::A2X`	Eingangswert 2, ausgewählt bei `K = 1`
`IN3`	Socket / Eingang	`adapter::types::unidirectional::A2X`	Eingangswert 3, ausgewählt bei `K = 2`
`IN4`	Socket / Eingang	`adapter::types::unidirectional::A2X`	Eingangswert 4, ausgewählt bei `K = 3`
`IN5`	Socket / Eingang	`adapter::types::unidirectional::A2X`	Eingangswert 5, ausgewählt bei `K = 4`

Funktionsweise

Der Baustein arbeitet als 5:1-Multiplexer:

1. Der Index `K` legt fest, welcher der fünf Eingänge aktiv ist:
   - `K = 0` ⟶ `IN1`
   - `K = 1` ⟶ `IN2`
   - `K = 2` ⟶ `IN3`
   - `K = 3` ⟶ `IN4`
   - `K = 4` ⟶ `IN5`
2. Der Wert des aktiven Eingangs wird über den Ausgang `OUT` weitergegeben.
3. Es findet keine Prüfung statt, ob sich der Eingangswert gegenüber dem vorherigen Zyklus verändert hat. Jedes neu berechnete Ergebnis wird unmittelbar weitergeleitet.

Durch das Fehlen einer Änderungserkennung bleibt die Ausgangskadenz vollständig von der Berechnungskadenz der Eingangsquelle abhängig. Auch unveränderte Werte erzeugen eine Ausgabe.

Technische Besonderheiten

- Der Baustein ist als generischer Funktionsblock deklariert. Das Attribut `eclipse4diac::core::GenericClassName` verweist auf die generische Implementierung `GEN_A2X_AUI_MUX`.

- Die Schnittstelle verwendet ausschließlich unidirektionale Adapter des Typs `adapter::types::unidirectional`.
- Es gibt keine direkten IEC-61499-Daten-Ein- oder -Ausgänge; der Datentransport erfolgt vollständig über Adapter.
- Der Baustein ist im Compiler-Paket `adapter::selection::unidirectional` abgelegt.
- Der `TypeHash` ist in der XML noch leer.
- Laut Versionshinweis ist das zugehörige Adapter-Backend `GEN_A2X_AUI_MUX` noch zu implementieren.

Zustandsübersicht

Die XML enthält keinen expliziten Ausführungszustandsautomaten (ECC). Der Baustein verhält sich als zustandsarmer Multiplexer:

Phase	Beschreibung
Initial	Warten auf einen gültigen Index `K` und anliegende Eingangswerte
Auswahl	Der zu `K` gehörende Eingang wird an den Ausgang `OUT` gekoppelt
Weiterleitung	Jeder neu berechnete Wert des aktiven Eingangs wird unmittelbar an `OUT` weitergegeben
Umschaltung	Bei Änderung von `K` wird die Auswahl aktualisiert

Anwendungsszenarien

- **Periodische Frequenz- oder Ableitungsberechnung:** Der nachgelagerte Verbraucher benötigt einen kontinuierlichen Datenstrom, auch wenn sich der Wert nicht ändert.
- **Sensormultiplexing:** Fünf Messwerte werden über A2X-Adapter eingelesen und je nach Index gemeinsam auf einen Ausgang geschaltet.
- **Adapterbasierte Signalverarbeitung:** Der Baustein lässt sich in IEC-61499-Systeme integrieren, die durchgängig mit unidirektionalen Adaptern arbeiten.
- **Umschaltung zwischen mehreren Berechnungsquellen:** Eine nachgeschaltete Funktion soll immer das aktuellste Ergebnis der ausgewählten Quelle erhalten.

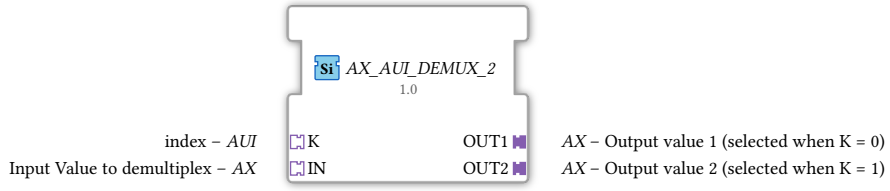
Vergleich mit ähnlichen Bausteinen

- **A2X_AUI_MUX_5:** Besitzt eine Änderungserkennung und gibt nur dann einen Wert weiter, wenn sich das Ergebnis tatsächlich geändert hat. Der `UNGATED`-Baustein verzichtet auf diese Erkennung.
- **AX_AUI_MUX_5_UNGATED:** Bietet dieselbe Multiplex-Logik, verwendet jedoch einen anderen Adaptertyp (`AX` statt `A2X`). `A2X_AUI_MUX_5_UNGATED` ist die A2X-Variante davon.
- **Klassischer MUX mit Daten-Eingängen:** Ein Standard-Multiplexer besitzt sichtbare Daten-Eingänge und einen skalareren Index-Eingang. Der vorliegende Baustein kapselt die Signale in Adaptertypen und ist dadurch für adapterbasierte Verbindungen besser geeignet.

Fazit

A2X_AUI_MUX_5_UNGATED ist ein flexibler 5:1-Multiplexer für adapterbasierte 4diac-Systeme. Seine besondere Stärke ist die bedingungslose Weitergabe jedes neu berechneten Ergebnisses. Dadurch ist er ideal für Anwendungen, die eine lückenlose, periodische Datenversorgung benötigen und keine unveränderten Werte unterdrücken dürfen.

AX_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_AUI_DEMUX_2 ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp BOOL. Anders als AX_DEMUX_2 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AX_AUI_DEMUX_2 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (BOOL, boolescher Wert), der an einen der n Ausgänge weitergeleitet wird (AX-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AX-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AX-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 2 Ausgangs-Adapter (OUT1 … OUT2) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AX_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AX_AUI_DEMUX_2 … AX_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen IN-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`BOOL`, boolescher Wert) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AX_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

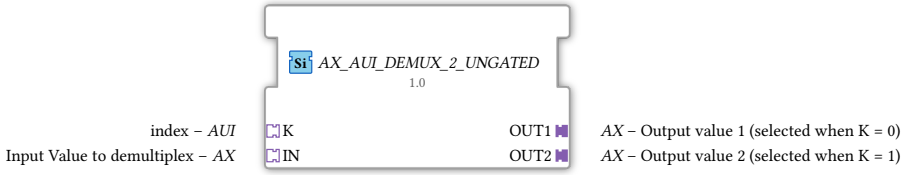
- **`AX_AUI_DEMUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AX_AUI_DEMUX_2` überträgt die Demultiplexer-Logik von `AX_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AX_AUI_DEMUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AX_AUI_DEMUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AX_AUI_DEMUX_2_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `BOOL`. Anders als `AX_DEMUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AX_AUI_DEMUX_2_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`BOOL`, boolescher Wert), der an einen der `n` Ausgänge weitergeleitet wird (AX-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AX-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AX-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Ausgangs-Adapter (`OUT1` … `OUT2`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AX_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AX_AUI_DEMUX_2_UNGATED … AX_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`BOOL`, boolescher Wert) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AX_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AX_AUI_DEMUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AX_AUI_DEMUX_2_UNGATED` überträgt die Demultiplexer-Logik von `AX_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AX_AUI_DEMUX_3

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AX_AUI_DEMUX_3` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `BOOL`. Anders als `AX_DEMUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AX_AUI_DEMUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`BOOL`, boolescher Wert), der an einen der `n` Ausgänge weitergeleitet wird (AX-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AX-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AX-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AX-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Ausgangs-Adapter (`OUT1` … `OUT3`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AX_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AX_AUI_DEMUX_2 … AX_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`BOOL`, boolescher Wert) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AX_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

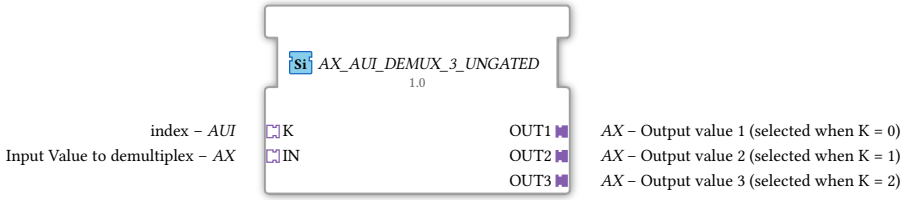
- **`AX_AUI_DEMUX_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AX_AUI_DEMUX_3` überträgt die Demultiplexer-Logik von `AX_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AX_AUI_DEMUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AX_AUI_DEMUX_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_AUI_DEMUX_3_UNGATED ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp BOOL. Anders als AX_DEMUX_3 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AX_AUI_DEMUX_3_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (BOOL, boolescher Wert), der an einen der n Ausgänge weitergeleitet wird (AX-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AX-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AX-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AX-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 3 Ausgangs-Adapter (OUT1 … OUT3) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AX_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AX_AUI_DEMUX_2 … AX_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`BOOL`, boolescher Wert) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

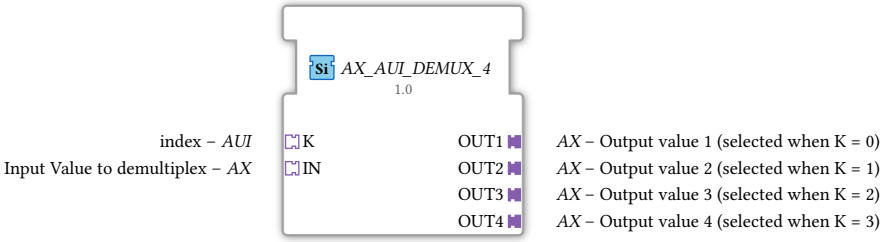
Vergleich mit `AX_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AX_AUI_DEMUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AX_AUI_DEMUX_3_UNGATED` überträgt die Demultiplexer-Logik von `AX_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AX_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AX_AUI_DEMUX_4` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `BOOL`. Anders als `AX_DEMUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AX_AUI_DEMUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`BOOL`, boolescher Wert), der an einen der `n` Ausgänge weitergeleitet wird (AX-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AX-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AX-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AX-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AX-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AX_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AX_AUI_DEMUX_2 … AX_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`BOOL`, boolescher Wert) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AX_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

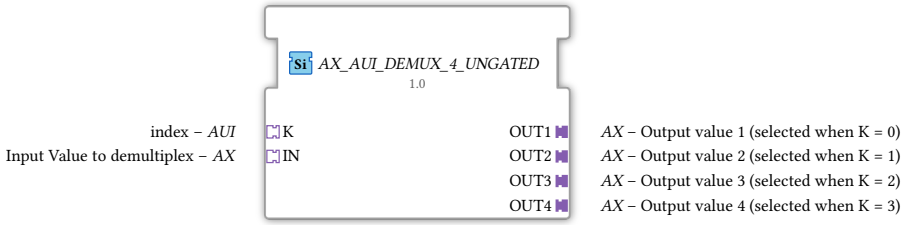
- **`AX_AUI_DEMUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AX_AUI_DEMUX_4` überträgt die Demultiplexer-Logik von `AX_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AX_AUI_DEMUX_4_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von AX_AUI_DEMUX_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_AUI_DEMUX_4_UNGATED ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp BOOL. Anders als AX_DEMUX_4 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AX_AUI_DEMUX_4_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (BOOL, boolescher Wert), der an einen der n Ausgänge weitergeleitet wird (AX-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AX-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AX-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AX-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AX-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AX_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AX_AUI_DEMUX_2 … AX_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`BOOL`, boolescher Wert) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AX_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

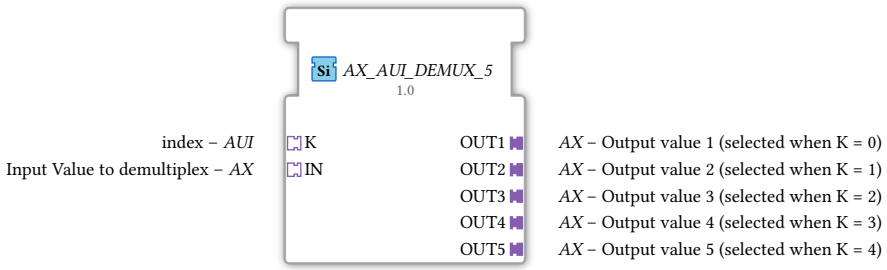
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AX_AUI_DEMUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AX_AUI_DEMUX_4_UNGATED` überträgt die Demultiplexer-Logik von `AX_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AX_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_AUI_DEMUX_5 ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp BOOL. Anders als AX_DEMUX_5 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AX_AUI_DEMUX_5 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (BOOL, boolescher Wert), der an einen der n Ausgänge weitergeleitet wird (AX-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AX-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AX-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AX-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AX-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AX-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 5 Ausgangs-Adapter (OUT1 … OUT5) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AX_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AX_AUI_DEMUX_2 … AX_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`BOOL`, boolescher Wert) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AX_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

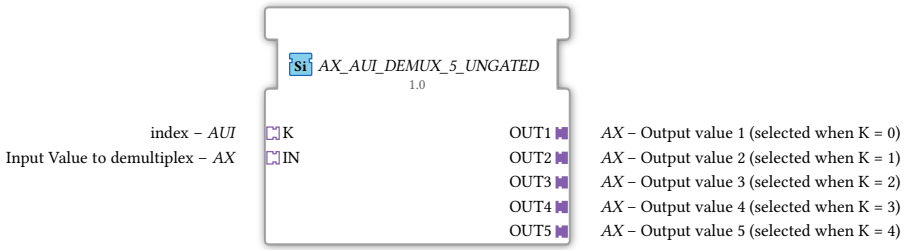
- **`AX_AUI_DEMUX_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AX_AUI_DEMUX_5` überträgt die Demultiplexer-Logik von `AX_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AX_AUI_DEMUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AX_AUI_DEMUX_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `AX_AUI_DEMUX_5_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `BOOL`. Anders als `AX_DEMUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AX_AUI_DEMUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`BOOL`, boolescher Wert), der an einen der `n` Ausgänge weitergeleitet wird (AX-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AX-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AX-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AX-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AX-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AX-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der `5` Ausgangs-Adapter (`OUT1` … `OUT5`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AX_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AX_AUI_DEMUX_2 … AX_AUI_DEMUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`BOOL`, boolescher Wert) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AX_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

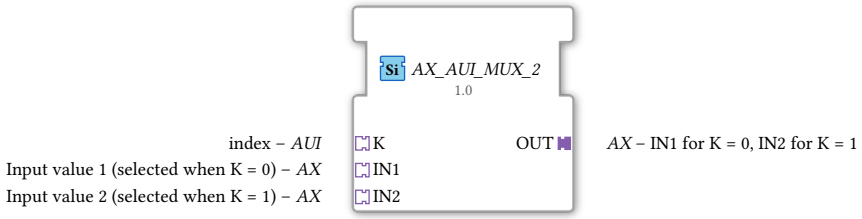
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AX_AUI_DEMUX_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AX_AUI_DEMUX_5_UNGATED` überträgt die Demultiplexer-Logik von `AX_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AX_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AX_AUI_MUX_2` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `BOOL`. Anders als `AX_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AX_AUI_MUX_2` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.

- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (AX-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (AX-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AX-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AX_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AX_AUI_MUX_2 … AX_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (boolescher Wert) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AX_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AX_AUI_MUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

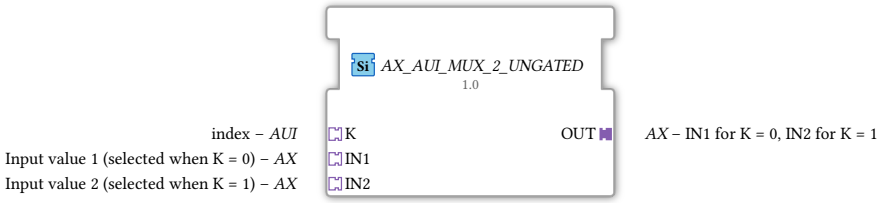
Fazit

Der `AX_AUI_MUX_2` überträgt die Multiplexer-Logik von `AX_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AX_AUI_MUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AX_AUI_MUX_2. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos

weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AX_AUI_MUX_2_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `BOOL`. Anders als `AX_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AX_AUI_MUX_2_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (AX-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (AX-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AX-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AX_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AX_AUI_MUX_2_UNGATED … AX_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (boolescher Wert) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AX_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

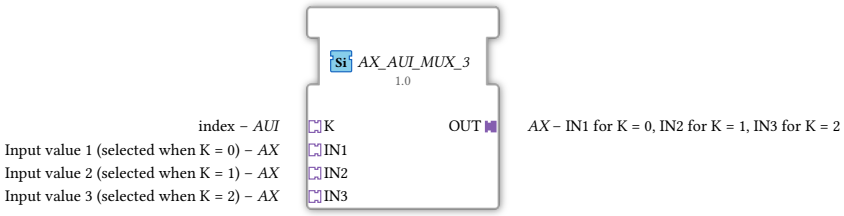
Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AX_AUI_MUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AX_AUI_MUX_2_UNGATED` überträgt die Multiplexer-Logik von `AX_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AX_AUI_MUX_3



• – ○ • – ○ • – ○ •

Einleitung

Der `AX_AUI_MUX_3` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `BOOL`. Anders als `AX_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AX_AUI_MUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (AX-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (AX-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (AX-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AX-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN3`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Eingangs-Adapter (`IN1` … `IN3`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AX_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AX_AUI_MUX_2 … AX_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (boolescher Wert) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird

- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit AX_MUX_3 (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

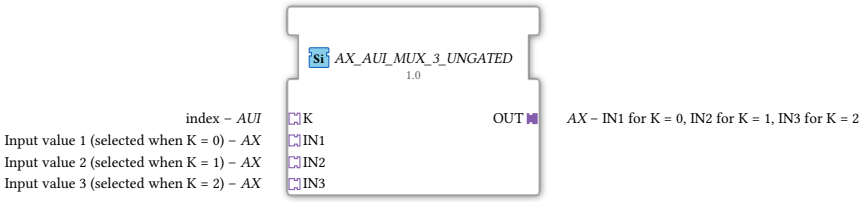
- **AX_AUI_MUX_3_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der AX_AUI_MUX_3 überträgt die Multiplexer-Logik von AX_MUX_3 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AX_AUI_MUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AX_AUI_MUX_3. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AX_AUI_MUX_3_UNGATED ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp BOOL. Anders als AX_MUX_3 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AX_AUI_MUX_3_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (AX-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (AX-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (AX-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AX-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN3`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Eingangs-Adapter (`IN1` … `IN3`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AX_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AX_AUI_MUX_2 … AX_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (boolescher Wert) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AX_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AX_AUI_MUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AX_AUI_MUX_3_UNGATED` überträgt die Multiplexer-Logik von `AX_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AX_AUI_MUX_4

• – ○ • – ○ • – ○ •

Einleitung

Der `AX_AUI_MUX_4` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `BOOL`. Anders als `AX_MUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AX_AUI_MUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index `K = 0` an `OUT` durchgereicht (AX-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index `K = 1` an `OUT` durchgereicht (AX-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index `K = 2` an `OUT` durchgereicht (AX-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index `K = 3` an `OUT` durchgereicht (AX-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AX-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN4`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Eingangs-Adapter (`IN1` … `IN4`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AX_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AX_AUI_MUX_2 … AX_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (boolescher Wert) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AX_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

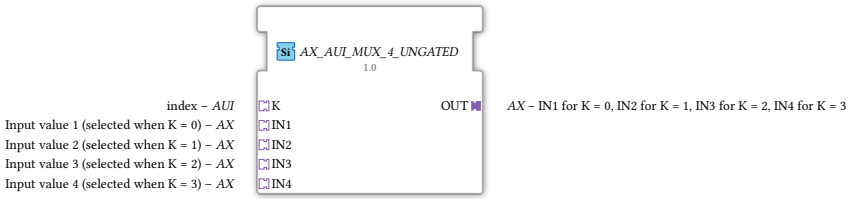
- **`AX_AUI_MUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AX_AUI_MUX_4` überträgt die Multiplexer-Logik von `AX_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AX_AUI_MUX_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AX_AUI_MUX_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der AX_AUI_MUX_4_UNGATED ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp BOOL. Anders als AX_MUX_4 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AX_AUI_MUX_4_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index K = 0 an OUT durchgereicht (AX-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index K = 1 an OUT durchgereicht (AX-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index K = 2 an OUT durchgereicht (AX-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index K = 3 an OUT durchgereicht (AX-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AX-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN4:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 4 Eingangs-Adapter (IN1 … IN4) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf OUT gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AX_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AX_AUI_MUX_2 … AX_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (boolescher Wert) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AX_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

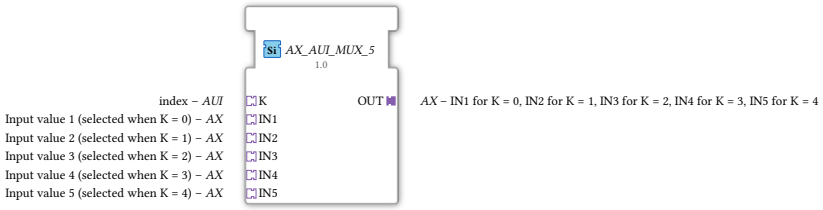
Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AX_AUI_MUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AX_AUI_MUX_4_UNGATED` überträgt die Multiplexer-Logik von `AX_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AX_AUI_MUX_5



• – ○ • – ○ • – ○ •

Einleitung

Der `AX_AUI_MUX_5` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `BOOL`. Anders als `AX_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AX_AUI_MUX_5` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index `K = 0` an `OUT` durchgereicht (AX-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index `K = 1` an `OUT` durchgereicht (AX-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index `K = 2` an `OUT` durchgereicht (AX-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index `K = 3` an `OUT` durchgereicht (AX-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index `K = 4` an `OUT` durchgereicht (AX-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AX-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN5`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Eingangs-Adapter (`IN1` … `IN5`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AX_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AX_AUI_MUX_2 … AX_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (boolescher Wert) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AX_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

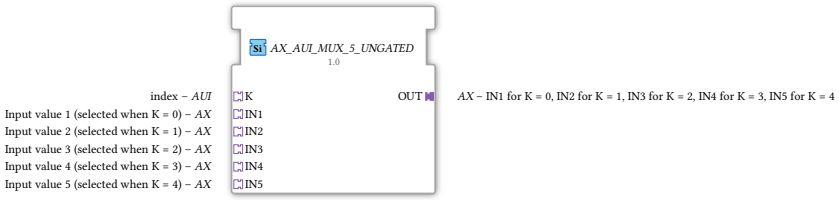
- **`AX_AUI_MUX_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AX_AUI_MUX_5` überträgt die Multiplexer-Logik von `AX_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AX_AUI_MUX_5_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von AX_AUI_MUX_5. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der AX_AUI_MUX_5_UNGATED ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp BOOL. Anders als AX_MUX_5 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AX_AUI_MUX_5_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index K = 0 an OUT durchgereicht (AX-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index K = 1 an OUT durchgereicht (AX-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index K = 2 an OUT durchgereicht (AX-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index K = 3 an OUT durchgereicht (AX-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index K = 4 an OUT durchgereicht (AX-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AX-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN5:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 5 Eingangs-Adapter (IN1 … IN5) gerade ausgewählt ist.

2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AX_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AX_AUI_MUX_2 … AX_AUI_MUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (boolescher Wert) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AX_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

- **`AX_AUI_MUX_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AX_AUI_MUX_5_UNGATED` überträgt die Multiplexer-Logik von `AX_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

BOOL

🎧 Podcast

- George Boole: Wie ein Autodidakt mit 0 und 1 das Fundament unserer digitalen Welt legte

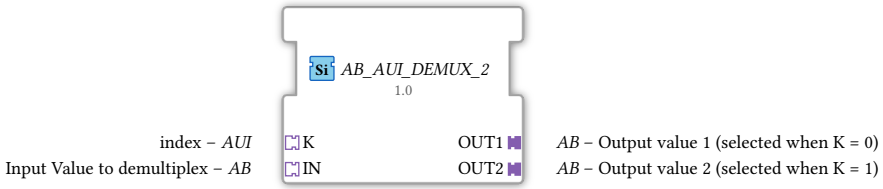
🛠️ Zugehörige Übungen

- Uebung_001_AX_b
- Uebung_002a2_AX
- Uebung_002a4
- Uebung_002a5b
- Uebung_002b3
- Uebung_004b_AX
- Uebung_006a3
- LinksRechts_T_FF_Event
- Uebung_035a2
- Uebung_035a3

- SoftkeyOverride_TO_QX_BG
- Uebung_055
- Uebung_056
- Bool8_TO_logiBUS_QX8
- Uebung_094a_AX
- Uebung_160

BYTE

AB_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AB_AUI_DEMUX_2` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `BYTE`. Anders als `AB_DEMUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AB_AUI_DEMUX_2` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`BYTE`, Byte-Wert (8 Bit Bitmuster)), der an einen der `n` Ausgänge weitergeleitet wird (AB-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AB-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AB-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Ausgangs-Adapter (`OUT1` … `OUT2`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AB_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AB_AUI_DEMUX_2 … AB_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`BYTE`, Byte-Wert (8 Bit Bitmuster)) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AB_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

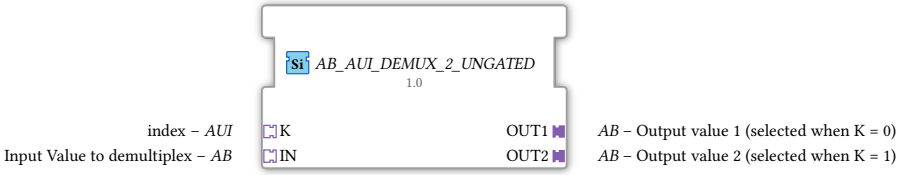
- **`AB_AUI_DEMUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AB_AUI_DEMUX_2` überträgt die Demultiplexer-Logik von `AB_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AB_AUI_DEMUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AB_AUI_DEMUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AB_AUI_DEMUX_2_UNGATED ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp BYTE. Anders als AB_DEMUX_2 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AB_AUI_DEMUX_2_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (BYTE, Byte-Wert (8 Bit Bitmuster)), der an einen der n Ausgänge weitergeleitet wird (AB-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AB-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AB-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 2 Ausgangs-Adapter (OUT1 … OUT2) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AB_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AB_AUI_DEMUX_2_UNGATED … AB_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`BYTE`, Byte-Wert (8 Bit Bitmuster)) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

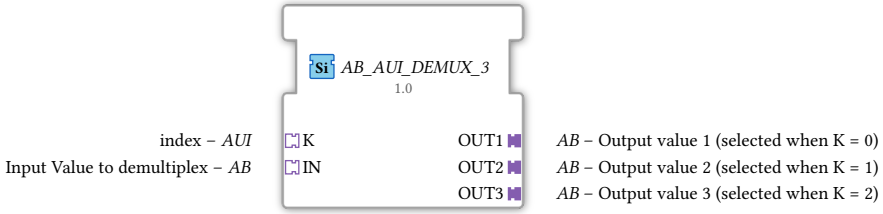
Vergleich mit `AB_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AB_AUI_DEMUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AB_AUI_DEMUX_2_UNGATED` überträgt die Demultiplexer-Logik von `AB_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AB_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AB_AUI_DEMUX_3` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `BYTE`. Anders als `AB_DEMUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AB_AUI_DEMUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`BYTE`, Byte-Wert (8 Bit Bitmuster)), der an einen der `n` Ausgänge weitergeleitet wird (AB-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AB-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AB-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AB-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Ausgangs-Adapter (`OUT1` … `OUT3`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AB_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AB_AUI_DEMUX_2 … AB_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`BYTE`, Byte-Wert (8 Bit Bitmuster)) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AB_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

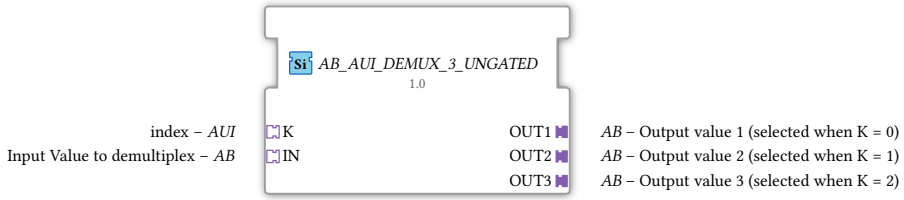
- **`AB_AUI_DEMUX_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AB_AUI_DEMUX_3` überträgt die Demultiplexer-Logik von `AB_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AB_AUI_DEMUX_3_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von AB_AUI_DEMUX_3. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AB_AUI_DEMUX_3_UNGATED ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp BYTE. Anders als AB_DEMUX_3 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AB_AUI_DEMUX_3_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (BYTE, Byte-Wert (8 Bit Bitmuster)), der an einen der n Ausgänge weitergeleitet wird (AB-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AB-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AB-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AB-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 3 Ausgangs-Adapter (OUT1 … OUT3) gerade ausgewählt ist.

2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AB_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AB_AUI_DEMUX_2 … AB_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`BYTE`, Byte-Wert (8 Bit Bitmuster)) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AB_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

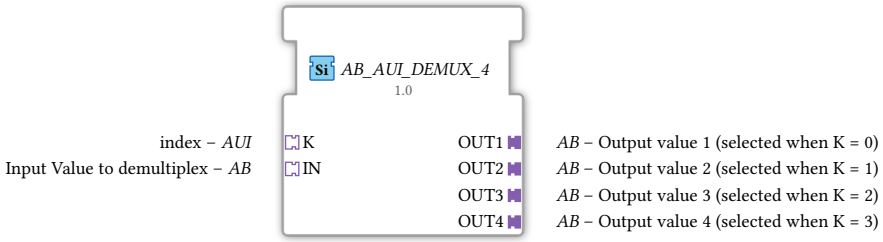
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AB_AUI_DEMUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AB_AUI_DEMUX_3_UNGATED` überträgt die Demultiplexer-Logik von `AB_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AB_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AB_AUI_DEMUX_4` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `BYTE`. Anders als `AB_DEMUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem

K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AB_AUI_DEMUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`BYTE`, Byte-Wert (8 Bit Bitmuster)), der an einen der `n` Ausgänge weitergeleitet wird (AB-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AB-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AB-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AB-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AB-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AB_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AB_AUI_DEMUX_2 … AB_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`BYTE`, Byte-Wert (8 Bit Bitmuster)) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AB_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

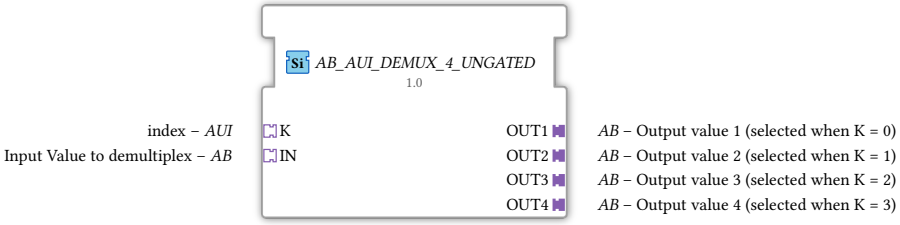
- **`AB_AUI_DEMUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AB_AUI_DEMUX_4` überträgt die Demultiplexer-Logik von `AB_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AB_AUI_DEMUX_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AB_AUI_DEMUX_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AB_AUI_DEMUX_4_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `BYTE`. Anders als `AB_DEMUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AB_AUI_DEMUX_4_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`BYTE`, Byte-Wert (8 Bit Bitmuster)), der an einen der `n` Ausgänge weitergeleitet wird (AB-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AB-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AB-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AB-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AB-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AB_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AB_AUI_DEMUX_2 … AB_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`BYTE`, Byte-Wert (8 Bit Bitmuster)) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AB_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

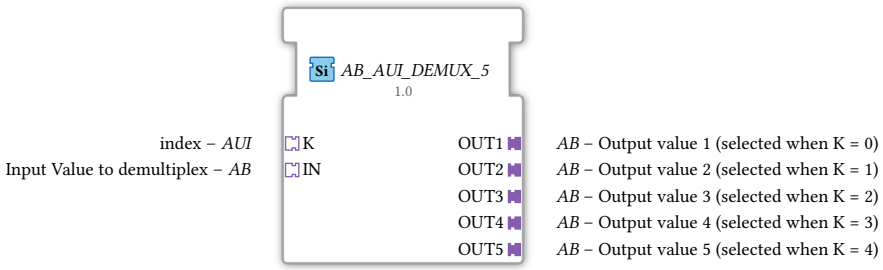
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AB_AUI_DEMUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AB_AUI_DEMUX_4_UNGATED` überträgt die Demultiplexer-Logik von `AB_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AB_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AB_AUI_DEMUX_5` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `BYTE`. Anders als `AB_DEMUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AB_AUI_DEMUX_5` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`BYTE`, Byte-Wert (8 Bit Bitmuster)), der an einen der `n` Ausgänge weitergeleitet wird (AB-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AB-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AB-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AB-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AB-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AB-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Ausgangs-Adapter (`OUT1` … `OUT5`) gerade ausgewählt ist.

2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AB_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AB_AUI_DEMUX_2 … AB_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`BYTE`, Byte-Wert (8 Bit Bitmuster)) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AB_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

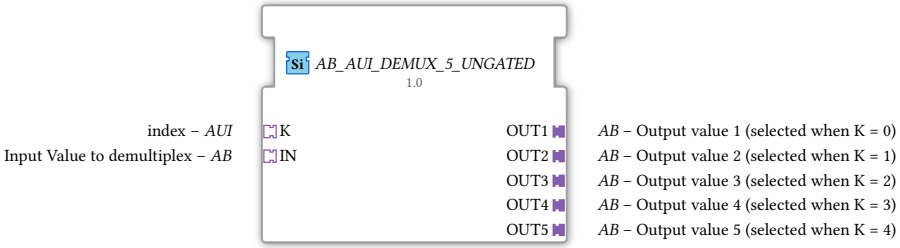
- **`AB_AUI_DEMUX_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AB_AUI_DEMUX_5` überträgt die Demultiplexer-Logik von `AB_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AB_AUI_DEMUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AB_AUI_DEMUX_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AB_AUI_DEMUX_5_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `BYTE`. Anders als `AB_DEMUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AB_AUI_DEMUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`BYTE`, Byte-Wert (8 Bit Bitmuster)), der an einen der `n` Ausgänge weitergeleitet wird (AB-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AB-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AB-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AB-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AB-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AB-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Ausgangs-Adapter (`OUT1` … `OUT5`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AB_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AB_AUI_DEMUX_2 … AB_AUI_DEMUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`BYTE`, Byte-Wert (8 Bit Bitmuster)) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AB_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

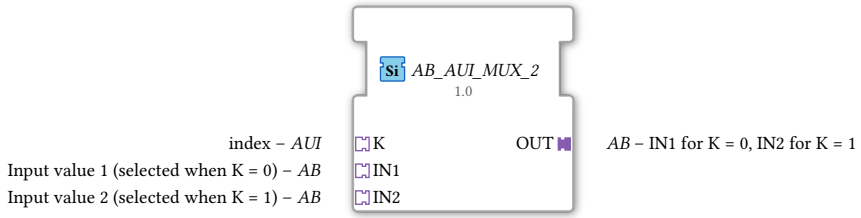
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AB_AUI_DEMUX_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AB_AUI_DEMUX_5_UNGATED` überträgt die Demultiplexer-Logik von `AB_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AB_AUI_MUX_2



• – ○ • – ○ • – ○ •

Einleitung

Der `AB_AUI_MUX_2` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `BYTE`. Anders als `AB_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AB_AUI_MUX_2` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (AB-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (AB-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AB-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AB_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AB_AUI_MUX_2 … AB_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (Byte-Wert (8 Bit Bitmuster)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit AB_MUX_2 (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

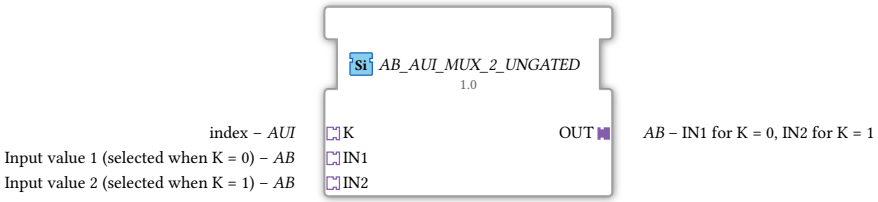
- **AB_AUI_MUX_2_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der AB_AUI_MUX_2 überträgt die Multiplexer-Logik von AB_MUX_2 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AB_AUI_MUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AB_AUI_MUX_2. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der AB_AUI_MUX_2_UNGATED ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp BYTE. Anders als AB_MUX_2 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AB_AUI_MUX_2_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (AB-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (AB-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AB-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AB_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AB_AUI_MUX_2_UNGATED … AB_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (Byte-Wert (8 Bit Bitmuster)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AB_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AB_AUI_MUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AB_AUI_MUX_2_UNGATED` überträgt die Multiplexer-Logik von `AB_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AB_AUI_MUX_3

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AB_AUI_MUX_3` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `BYTE`. Anders als `AB_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AB_AUI_MUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (AB-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (AB-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (AB-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AB-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN3`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Eingangs-Adapter (`IN1` … `IN3`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.

- Generische Implementierung (`GEN_AB_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AB_AUI_MUX_2 … AB_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (Byte-Wert (8 Bit Bitmuster)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AB_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

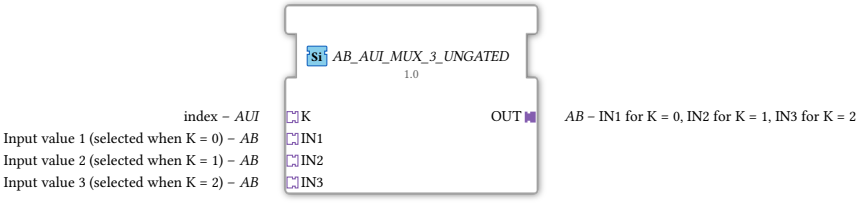
- **`AB_AUI_MUX_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AB_AUI_MUX_3` überträgt die Multiplexer-Logik von `AB_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AB_AUI_MUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AB_AUI_MUX_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AB_AUI_MUX_3_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `BYTE`. Anders als `AB_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional

Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge
- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AB_AUI_MUX_3_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge
- Keine Ereignis-Ausgänge.

Daten-Eingänge
- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge
- Keine direkten Datenausgänge vorhanden.

Adapter
- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index K = 0 an OUT durchgereicht (AB-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index K = 1 an OUT durchgereicht (AB-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index K = 2 an OUT durchgereicht (AB-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AB-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN3:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 3 Eingangs-Adapter (IN1 … IN3) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf OUT gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten
- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AB_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AB_AUI_MUX_2 … AB_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN1…IN3) ⟶ aktuell selektierten Eingang auslesen, OUT bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (Byte-Wert (8 Bit Bitmuster)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AB_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AB_AUI_MUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AB_AUI_MUX_3_UNGATED` überträgt die Multiplexer-Logik von `AB_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AB_AUI_MUX_4

• – ○ • – ○ • – ○ •

Einleitung

Der `AB_AUI_MUX_4` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `BYTE`. Anders als `AB_MUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AB_AUI_MUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.

- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index `K = 0` an `OUT` durchgereicht (AB-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index `K = 1` an `OUT` durchgereicht (AB-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index `K = 2` an `OUT` durchgereicht (AB-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index `K = 3` an `OUT` durchgereicht (AB-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AB-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN4`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Eingangs-Adapter (`IN1` … `IN4`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AB_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AB_AUI_MUX_2 … AB_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (Byte-Wert (8 Bit Bitmuster)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AB_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

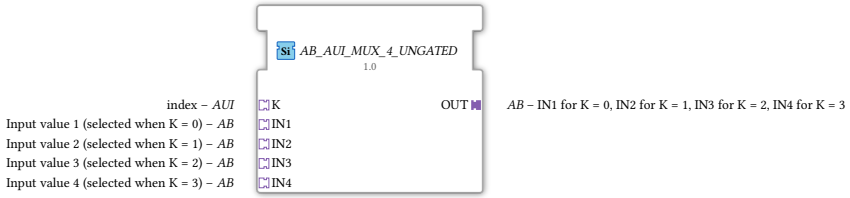
- **`AB_AUI_MUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AB_AUI_MUX_4` überträgt die Multiplexer-Logik von `AB_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AB_AUI_MUX_4_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von AB_AUI_MUX_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der AB_AUI_MUX_4_UNGATED ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp BYTE. Anders als AB_MUX_4 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AB_AUI_MUX_4_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index K = 0 an OUT durchgereicht (AB-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index K = 1 an OUT durchgereicht (AB-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index K = 2 an OUT durchgereicht (AB-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index K = 3 an OUT durchgereicht (AB-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AB-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN4:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 4 Eingangs-Adapter (IN1 … IN4) gerade ausgewählt ist.

2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AB_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AB_AUI_MUX_2 … AB_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (Byte-Wert (8 Bit Bitmuster)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AB_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

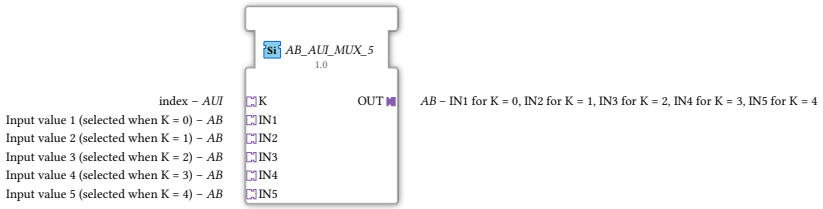
Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AB_AUI_MUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AB_AUI_MUX_4_UNGATED` überträgt die Multiplexer-Logik von `AB_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AB_AUI_MUX_5



• – ○ • – ○ • – ○ •

Einleitung

Der `AB_AUI_MUX_5` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `BYTE`. Anders als `AB_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“).

Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge
- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AB_AUI_MUX_5 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge
- Keine Ereignis-Ausgänge.

Daten-Eingänge
- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge
- Keine direkten Datenausgänge vorhanden.

Adapter
- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index K = 0 an OUT durchgereicht (AB-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index K = 1 an OUT durchgereicht (AB-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index K = 2 an OUT durchgereicht (AB-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index K = 3 an OUT durchgereicht (AB-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index K = 4 an OUT durchgereicht (AB-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AB-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN5:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 5 Eingangs-Adapter (IN1 … IN5) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf OUT gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten
- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AB_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AB_AUI_MUX_2 … AB_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN1…IN5) ⟶ aktuell selektierten Eingang auslesen, OUT bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (Byte-Wert (8 Bit Bitmuster)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AB_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

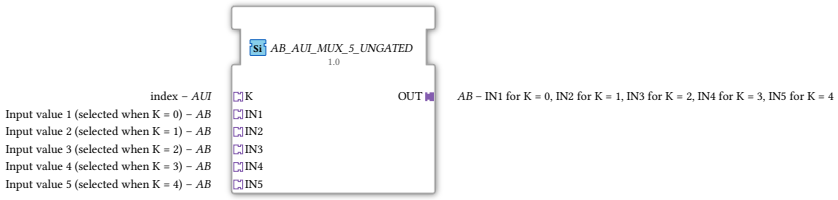
- **`AB_AUI_MUX_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AB_AUI_MUX_5` überträgt die Multiplexer-Logik von `AB_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AB_AUI_MUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AB_AUI_MUX_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `AB_AUI_MUX_5_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `BYTE`. Anders als `AB_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AB_AUI_MUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index `K = 0` an `OUT` durchgereicht (AB-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index `K = 1` an `OUT` durchgereicht (AB-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index `K = 2` an `OUT` durchgereicht (AB-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index `K = 3` an `OUT` durchgereicht (AB-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index `K = 4` an `OUT` durchgereicht (AB-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AB-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN5`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Eingangs-Adapter (`IN1` … `IN5`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AB_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AB_AUI_MUX_2 … AB_AUI_MUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (Byte-Wert (8 Bit Bitmuster)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AB_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

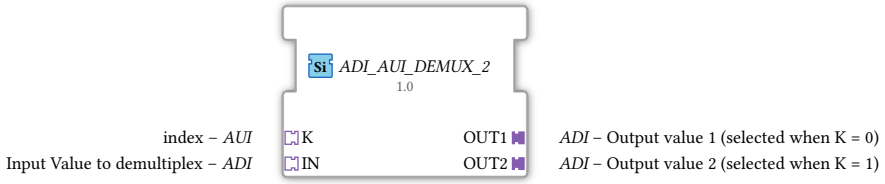
- `AB_AUI_MUX_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AB_AUI_MUX_5_UNGATED` überträgt die Multiplexer-Logik von `AB_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

DINT

ADI_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ADI_AUI_DEMUX_2` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `DINT`. Anders als `ADI_DEMUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ADI_AUI_DEMUX_2` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`DINT`, 32-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der `n` Ausgänge weitergeleitet wird (ADI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (ADI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (ADI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Ausgangs-Adapter (`OUT1` … `OUT2`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ADI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ADI_AUI_DEMUX_2 … ADI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`DINT`, 32-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ADI_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

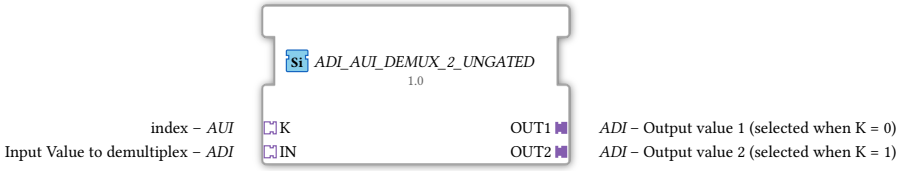
- **`ADI_AUI_DEMUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `ADI_AUI_DEMUX_2` überträgt die Demultiplexer-Logik von `ADI_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ADI_AUI_DEMUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ADI_AUI_DEMUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der ADI_AUI_DEMUX_2_UNGATED ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp DINT. Anders als ADI_DEMUX_2 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet ADI_AUI_DEMUX_2_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (DINT, 32-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der n Ausgänge weitergeleitet wird (ADI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (ADI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (ADI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 2 Ausgangs-Adapter (OUT1 … OUT2) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_ADI_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ADI_AUI_DEMUX_2_UNGATED … ADI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`DINT`, 32-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

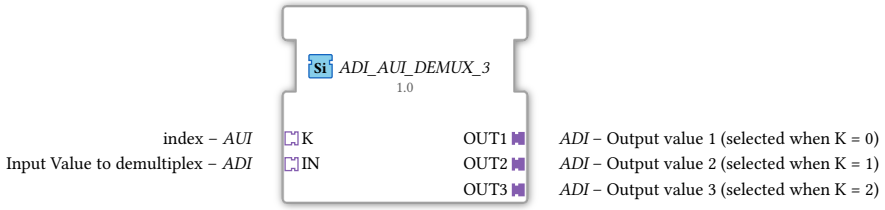
Vergleich mit `ADI_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`ADI_AUI_DEMUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `ADI_AUI_DEMUX_2_UNGATED` überträgt die Demultiplexer-Logik von `ADI_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ADI_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ADI_AUI_DEMUX_3` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `DINT`. Anders als `ADI_DEMUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ADI_AUI_DEMUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`DINT`, 32-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der `n` Ausgänge weitergeleitet wird (ADI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (ADI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (ADI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (ADI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Ausgangs-Adapter (`OUT1` … `OUT3`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ADI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ADI_AUI_DEMUX_2 … ADI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`DINT`, 32-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ADI_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

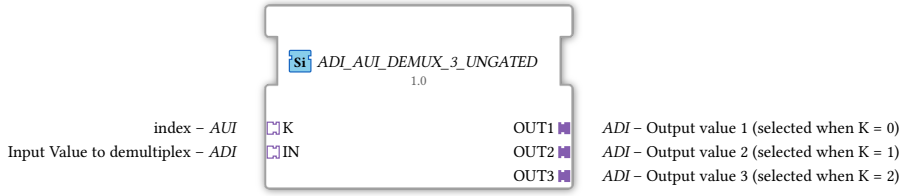
- **`ADI_AUI_DEMUX_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `ADI_AUI_DEMUX_3` überträgt die Demultiplexer-Logik von `ADI_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ADI_AUI_DEMUX_3_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `ADI_AUI_DEMUX_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `ADI_AUI_DEMUX_3_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `DINT`. Anders als `ADI_DEMUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ADI_AUI_DEMUX_3_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`DINT`, 32-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der `n` Ausgänge weitergeleitet wird (ADI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (ADI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (ADI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (ADI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Ausgangs-Adapter (`OUT1` … `OUT3`) gerade ausgewählt ist.

2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ADI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ADI_AUI_DEMUX_2 … ADI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`DINT`, 32-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ADI_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`ADI_AUI_DEMUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `ADI_AUI_DEMUX_3_UNGATED` überträgt die Demultiplexer-Logik von `ADI_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ADI_AUI_DEMUX_4

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ADI_AUI_DEMUX_4` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `DINT`. Anders als `ADI_DEMUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem

K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ADI_AUI_DEMUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`DINT`, 32-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der `n` Ausgänge weitergeleitet wird (ADI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (ADI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (ADI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (ADI-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (ADI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ADI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ADI_AUI_DEMUX_2 … ADI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`DINT`, 32-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ADI_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

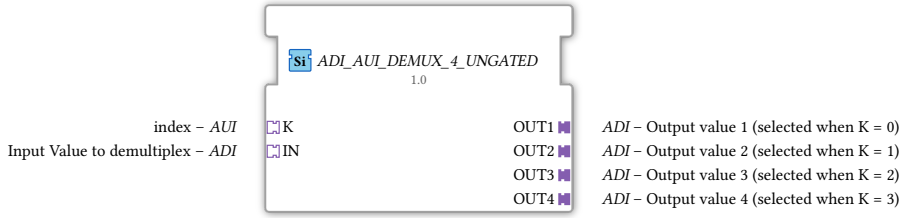
- **`ADI_AUI_DEMUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `ADI_AUI_DEMUX_4` überträgt die Demultiplexer-Logik von `ADI_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ADI_AUI_DEMUX_4_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `ADI_AUI_DEMUX_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ADI_AUI_DEMUX_4_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `DINT`. Anders als `ADI_DEMUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ADI_AUI_DEMUX_4_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`DINT`, 32-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der `n` Ausgänge weitergeleitet wird (ADI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (ADI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (ADI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (ADI-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (ADI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ADI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ADI_AUI_DEMUX_2 … ADI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`DINT`, 32-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ADI_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

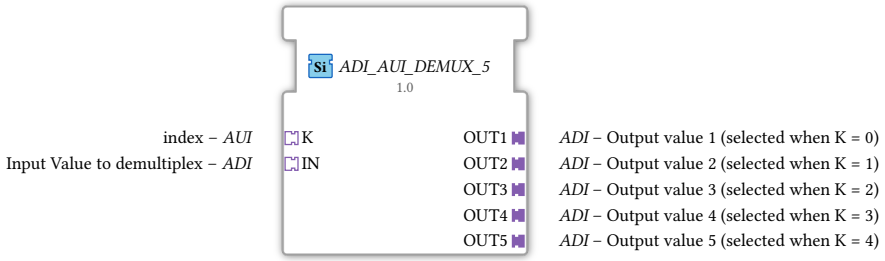
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **ADI_AUI_DEMUX_4**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der ADI_AUI_DEMUX_4_UNGATED überträgt die Demultiplexer-Logik von ADI_DEMUX_4 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ADI_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der ADI_AUI_DEMUX_5 ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp DINT. Anders als ADI_DEMUX_5 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet ADI_AUI_DEMUX_5 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (DINT, 32-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der n Ausgänge weitergeleitet wird (ADI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (ADI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (ADI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (ADI-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (ADI-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (ADI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Ausgangs-Adapter (`OUT1` … `OUT5`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ADI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ADI_AUI_DEMUX_2 … ADI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`DINT`, 32-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ADI_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

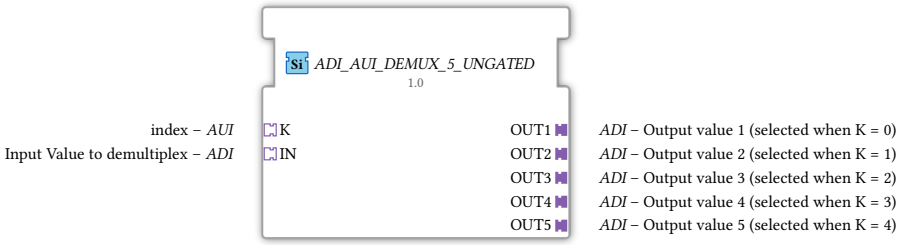
- **`ADI_AUI_DEMUX_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `ADI_AUI_DEMUX_5` überträgt die Demultiplexer-Logik von `ADI_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ADI_AUI_DEMUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ADI_AUI_DEMUX_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ADI_AUI_DEMUX_5_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `DINT`. Anders als `ADI_DEMUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ADI_AUI_DEMUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`DINT`, 32-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der `n` Ausgänge weitergeleitet wird (ADI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (ADI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (ADI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (ADI-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (ADI-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (ADI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Ausgangs-Adapter (`OUT1` … `OUT5`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ADI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ADI_AUI_DEMUX_2 … ADI_AUI_DEMUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`DINT`, 32-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

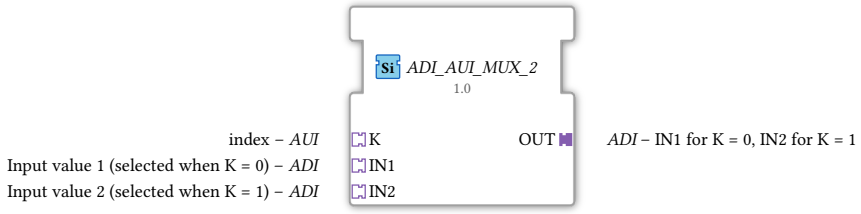
Vergleich mit `ADI_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`ADI_AUI_DEMUX_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `ADI_AUI_DEMUX_5_UNGATED` überträgt die Demultiplexer-Logik von `ADI_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ADI_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ADI_AUI_MUX_2` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `DINT`. Anders als `ADI_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ADI_AUI_MUX_2` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (ADI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (ADI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ADI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ADI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ADI_AUI_MUX_2 … ADI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (32-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ADI_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

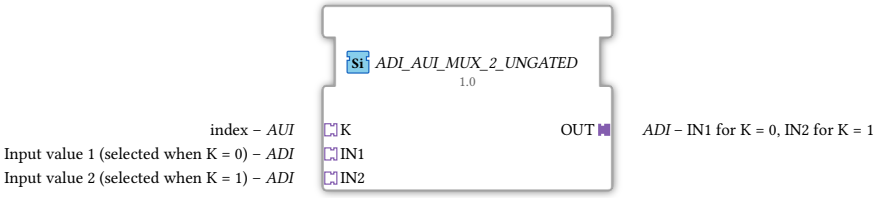
- **`ADI_AUI_MUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `ADI_AUI_MUX_2` überträgt die Multiplexer-Logik von `ADI_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ADI_AUI_MUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ADI_AUI_MUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ADI_AUI_MUX_2_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `DINT`. Anders als `ADI_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ADI_AUI_MUX_2_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (ADI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (ADI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ADI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ADI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ADI_AUI_MUX_2_UNGATED … ADI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (32-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ADI_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

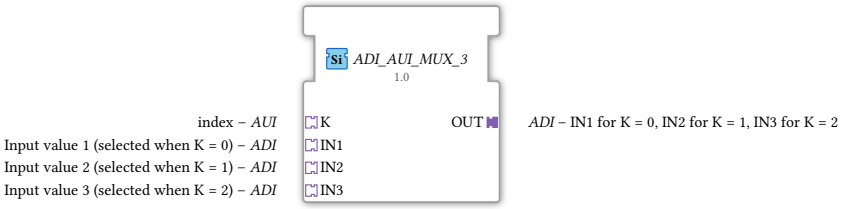
Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`ADI_AUI_MUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `ADI_AUI_MUX_2_UNGATED` überträgt die Multiplexer-Logik von `ADI_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ADI_AUI_MUX_3



• – ○ • – ○ • – ○ •

Einleitung

Der `ADI_AUI_MUX_3` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `DINT`. Anders als `ADI_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ADI_AUI_MUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (ADI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (ADI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (ADI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ADI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN3`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Eingangs-Adapter (`IN1` … `IN3`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.

- Generische Implementierung (`GEN_ADI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ADI_AUI_MUX_2 … ADI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (32-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ADI_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

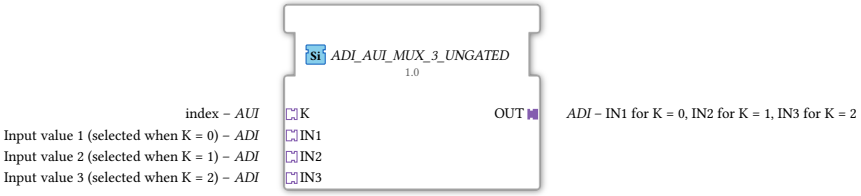
- **`ADI_AUI_MUX_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `ADI_AUI_MUX_3` überträgt die Multiplexer-Logik von `ADI_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ADI_AUI_MUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ADI_AUI_MUX_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ADI_AUI_MUX_3_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `DINT`. Anders als `ADI_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter

Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet ADI_AUI_MUX_3_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index K = 0 an OUT durchgereicht (ADI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index K = 1 an OUT durchgereicht (ADI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index K = 2 an OUT durchgereicht (ADI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ADI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN3:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 3 Eingangs-Adapter (IN1 … IN3) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf OUT gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_ADI_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ADI_AUI_MUX_2 … ADI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN1…IN3) ⟶ aktuell selektierten Eingang auslesen, OUT bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (32-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ADI_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`ADI_AUI_MUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `ADI_AUI_MUX_3_UNGATED` überträgt die Multiplexer-Logik von `ADI_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ADI_AUI_MUX_4

• – ○ • – ○ • – ○ •

Einleitung

Der `ADI_AUI_MUX_4` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `DINT`. Anders als `ADI_MUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ADI_AUI_MUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index `K = 0` an `OUT` durchgereicht (ADI-Adaptertyp).

- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index `K = 1` an `OUT` durchgereicht (ADI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index `K = 2` an `OUT` durchgereicht (ADI-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index `K = 3` an `OUT` durchgereicht (ADI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ADI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN4`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Eingangs-Adapter (`IN1` … `IN4`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ADI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ADI_AUI_MUX_2 … ADI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (32-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ADI_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

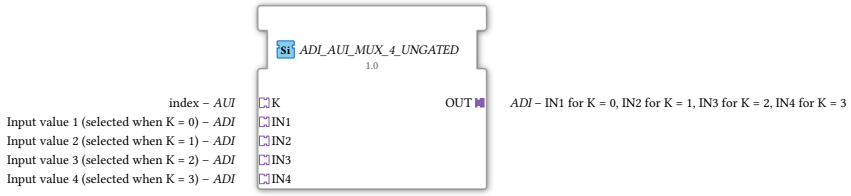
- **`ADI_AUI_MUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `ADI_AUI_MUX_4` überträgt die Multiplexer-Logik von `ADI_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ADI_AUI_MUX_4_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von ADI_AUI_MUX_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der ADI_AUI_MUX_4_UNGATED ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp DINT. Anders als ADI_MUX_4 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet ADI_AUI_MUX_4_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index K = 0 an OUT durchgereicht (ADI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index K = 1 an OUT durchgereicht (ADI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index K = 2 an OUT durchgereicht (ADI-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index K = 3 an OUT durchgereicht (ADI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ADI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN4:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 4 Eingangs-Adapter (IN1 … IN4) gerade ausgewählt ist.

2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ADI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ADI_AUI_MUX_2 … ADI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (32-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ADI_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`ADI_AUI_MUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `ADI_AUI_MUX_4_UNGATED` überträgt die Multiplexer-Logik von `ADI_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ADI_AUI_MUX_5

• – ○ • – ○ • – ○ •

Einleitung

Der `ADI_AUI_MUX_5` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `DINT`. Anders als `ADI_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“).

Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge
- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet ADI_AUI_MUX_5 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge
- Keine Ereignis-Ausgänge.

Daten-Eingänge
- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge
- Keine direkten Datenausgänge vorhanden.

Adapter
- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index K = 0 an OUT durchgereicht (ADI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index K = 1 an OUT durchgereicht (ADI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index K = 2 an OUT durchgereicht (ADI-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index K = 3 an OUT durchgereicht (ADI-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index K = 4 an OUT durchgereicht (ADI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ADI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN5:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 5 Eingangs-Adapter (IN1 … IN5) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf OUT gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten
- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_ADI_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ADI_AUI_MUX_2 … ADI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN1…IN5) → aktuell selektierten Eingang auslesen, OUT bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (32-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ADI_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

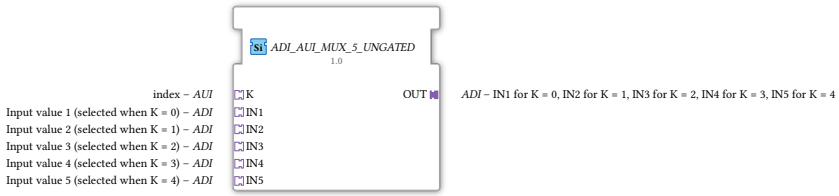
- **`ADI_AUI_MUX_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `ADI_AUI_MUX_5` überträgt die Multiplexer-Logik von `ADI_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ADI_AUI_MUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ADI_AUI_MUX_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `ADI_AUI_MUX_5_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `DINT`. Anders als `ADI_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ADI_AUI_MUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index `K = 0` an `OUT` durchgereicht (ADI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index `K = 1` an `OUT` durchgereicht (ADI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index `K = 2` an `OUT` durchgereicht (ADI-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index `K = 3` an `OUT` durchgereicht (ADI-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index `K = 4` an `OUT` durchgereicht (ADI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ADI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN5`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Eingangs-Adapter (`IN1` … `IN5`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ADI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ADI_AUI_MUX_2 … ADI_AUI_MUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (32-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ADI_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

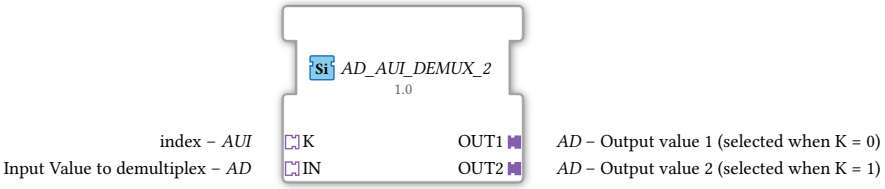
- **`ADI_AUI_MUX_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `ADI_AUI_MUX_5_UNGATED` überträgt die Multiplexer-Logik von `ADI_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

DWORD

AD_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AD_AUI_DEMUX_2` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `DWORD`. Anders als `AD_DEMUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AD_AUI_DEMUX_2` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`DWORD`, 32-Bit-Bitmuster), der an einen der `n` Ausgänge weitergeleitet wird (AD-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AD-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AD-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Ausgangs-Adapter (`OUT1` … `OUT2`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AD_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AD_AUI_DEMUX_2 … AD_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`DWORD`, 32-Bit-Bitmuster) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AD_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

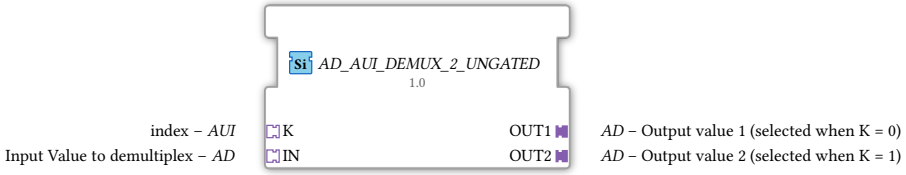
- **`AD_AUI_DEMUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AD_AUI_DEMUX_2` überträgt die Demultiplexer-Logik von `AD_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AD_AUI_DEMUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AD_AUI_DEMUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AD_AUI_DEMUX_2_UNGATED ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp DWORD. Anders als AD_DEMUX_2 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AD_AUI_DEMUX_2_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (DWORD, 32-Bit-Bitmuster), der an einen der n Ausgänge weitergeleitet wird (AD-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AD-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AD-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 2 Ausgangs-Adapter (OUT1 … OUT2) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AD_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AD_AUI_DEMUX_2_UNGATED … AD_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`DWORD`, 32-Bit-Bitmuster) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

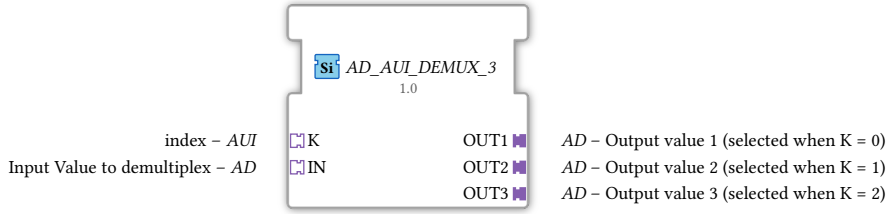
Vergleich mit `AD_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AD_AUI_DEMUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AD_AUI_DEMUX_2_UNGATED` überträgt die Demultiplexer-Logik von `AD_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AD_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AD_AUI_DEMUX_3` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `DWORD`. Anders als `AD_DEMUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AD_AUI_DEMUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`DWORD`, 32-Bit-Bitmuster), der an einen der `n` Ausgänge weitergeleitet wird (AD-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AD-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AD-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AD-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Ausgangs-Adapter (`OUT1` … `OUT3`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AD_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AD_AUI_DEMUX_2 … AD_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`DWORD`, 32-Bit-Bitmuster) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AD_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

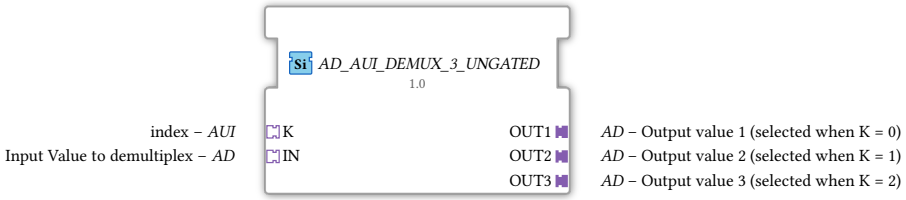
- **`AD_AUI_DEMUX_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AD_AUI_DEMUX_3` überträgt die Demultiplexer-Logik von `AD_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AD_AUI_DEMUX_3_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von AD_AUI_DEMUX_3. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AD_AUI_DEMUX_3_UNGATED ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp DWORD. Anders als AD_DEMUX_3 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AD_AUI_DEMUX_3_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (DWORD, 32-Bit-Bitmuster), der an einen der n Ausgänge weitergeleitet wird (AD-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AD-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AD-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AD-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 3 Ausgangs-Adapter (OUT1 … OUT3) gerade ausgewählt ist.

2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AD_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AD_AUI_DEMUX_2 … AD_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`DWORD`, 32-Bit-Bitmuster) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AD_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

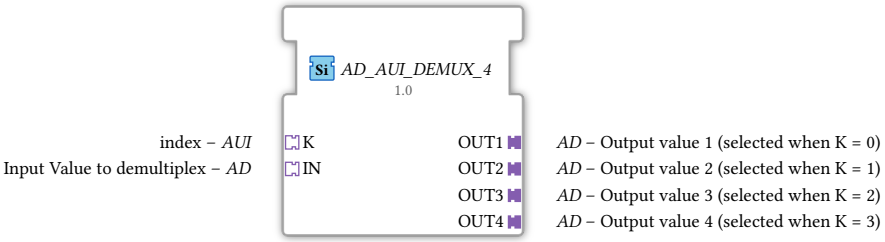
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AD_AUI_DEMUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AD_AUI_DEMUX_3_UNGATED` überträgt die Demultiplexer-Logik von `AD_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AD_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AD_AUI_DEMUX_4` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `DWORD`. Anders als `AD_DEMUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem

K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AD_AUI_DEMUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`DWORD`, 32-Bit-Bitmuster), der an einen der `n` Ausgänge weitergeleitet wird (AD-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AD-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AD-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AD-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AD-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AD_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AD_AUI_DEMUX_2 … AD_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`DWORD`, 32-Bit-Bitmuster) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AD_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

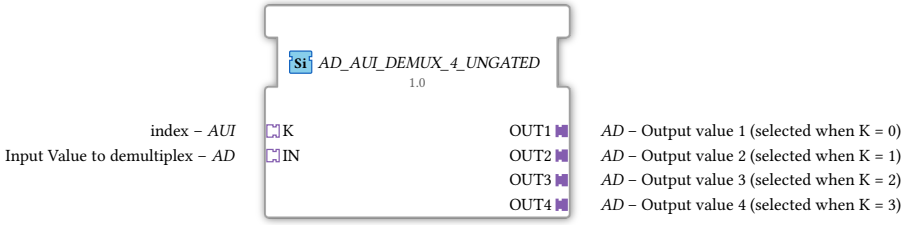
- **`AD_AUI_DEMUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AD_AUI_DEMUX_4` überträgt die Demultiplexer-Logik von `AD_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AD_AUI_DEMUX_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AD_AUI_DEMUX_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `AD_AUI_DEMUX_4_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `DWORD`. Anders als `AD_DEMUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AD_AUI_DEMUX_4_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`DWORD`, 32-Bit-Bitmuster), der an einen der `n` Ausgänge weitergeleitet wird (AD-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AD-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AD-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AD-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AD-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AD_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AD_AUI_DEMUX_2 … AD_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`DWORD`, 32-Bit-Bitmuster) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AD_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

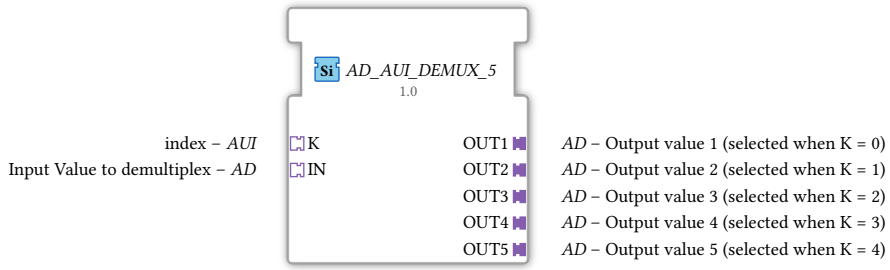
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AD_AUI_DEMUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der AD_AUI_DEMUX_4_UNGATED überträgt die Demultiplexer-Logik von AD_DEMUX_4 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AD_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AD_AUI_DEMUX_5 ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp DWORD. Anders als AD_DEMUX_5 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AD_AUI_DEMUX_5 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (DWORD, 32-Bit-Bitmuster), der an einen der n Ausgänge weitergeleitet wird (AD-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AD-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AD-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AD-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AD-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AD-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 5 Ausgangs-Adapter (OUT1 … OUT5) gerade ausgewählt ist.

2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AD_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AD_AUI_DEMUX_2 … AD_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen IN-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (DWORD, 32-Bit-Bitmuster) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit AD_DEMUX_5 (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

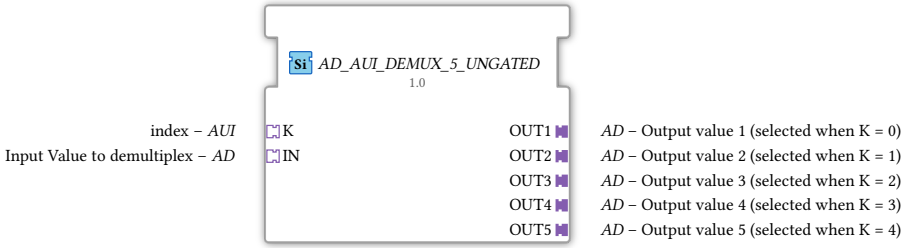
- **AD_AUI_DEMUX_5_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der AD_AUI_DEMUX_5 überträgt die Demultiplexer-Logik von AD_DEMUX_5 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AD_AUI_DEMUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AD_AUI_DEMUX_5. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AD_AUI_DEMUX_5_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `DWORD`. Anders als `AD_DEMUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AD_AUI_DEMUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`DWORD`, 32-Bit-Bitmuster), der an einen der `n` Ausgänge weitergeleitet wird (AD-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AD-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AD-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AD-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AD-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AD-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Ausgangs-Adapter (`OUT1` … `OUT5`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AD_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AD_AUI_DEMUX_2 … AD_AUI_DEMUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`DWORD`, 32-Bit-Bitmuster) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AD_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

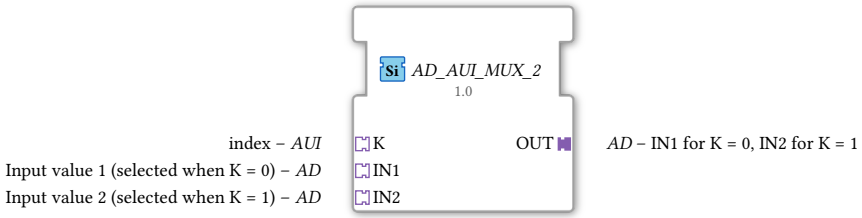
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AD_AUI_DEMUX_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AD_AUI_DEMUX_5_UNGATED` überträgt die Demultiplexer-Logik von `AD_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AD_AUI_MUX_2



• – ○ • – ○ • – ○ •

Einleitung

Der `AD_AUI_MUX_2` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `DWORD`. Anders als `AD_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AD_AUI_MUX_2` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (AD-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (AD-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AD-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AD_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AD_AUI_MUX_2 … AD_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (32-Bit-Bitmuster) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit AD_MUX_2 (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

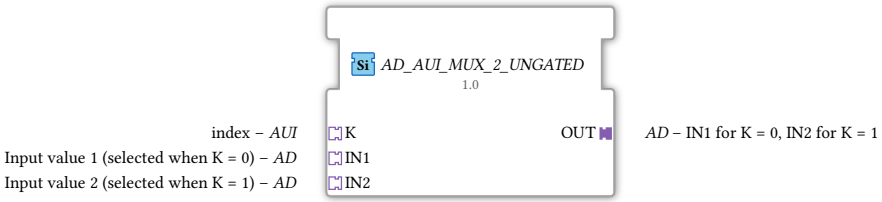
- **AD_AUI_MUX_2_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der AD_AUI_MUX_2 überträgt die Multiplexer-Logik von AD_MUX_2 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AD_AUI_MUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AD_AUI_MUX_2. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der AD_AUI_MUX_2_UNGATED ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp DWORD. Anders als AD_MUX_2 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AD_AUI_MUX_2_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (AD-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (AD-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AD-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AD_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AD_AUI_MUX_2_UNGATED … AD_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (32-Bit-Bitmuster) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AD_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

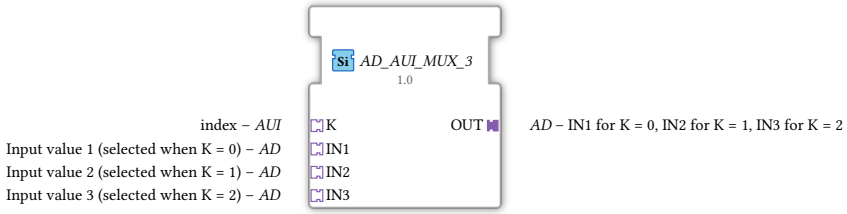
Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AD_AUI_MUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AD_AUI_MUX_2_UNGATED` überträgt die Multiplexer-Logik von `AD_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AD_AUI_MUX_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AD_AUI_MUX_3` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `DWORD`. Anders als `AD_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AD_AUI_MUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (AD-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (AD-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (AD-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AD-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN3`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Eingangs-Adapter (`IN1` … `IN3`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.

- Generische Implementierung (`GEN_AD_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AD_AUI_MUX_2 … AD_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (32-Bit-Bitmuster) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AD_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

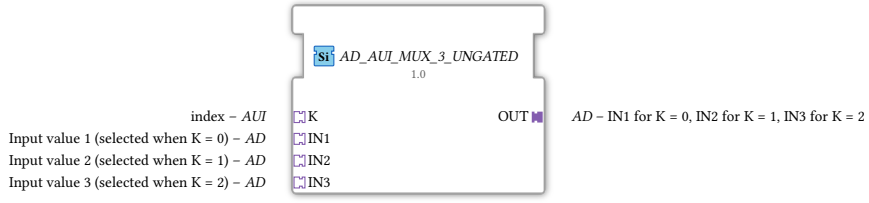
- **`AD_AUI_MUX_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AD_AUI_MUX_3` überträgt die Multiplexer-Logik von `AD_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AD_AUI_MUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AD_AUI_MUX_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AD_AUI_MUX_3_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `DWORD`. Anders als `AD_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional

Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge
- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AD_AUI_MUX_3_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge
- Keine Ereignis-Ausgänge.

Daten-Eingänge
- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge
- Keine direkten Datenausgänge vorhanden.

Adapter
- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index K = 0 an OUT durchgereicht (AD-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index K = 1 an OUT durchgereicht (AD-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index K = 2 an OUT durchgereicht (AD-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AD-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN3:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 3 Eingangs-Adapter (IN1 … IN3) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf OUT gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten
- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AD_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AD_AUI_MUX_2 … AD_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN1…IN3) ⟶ aktuell selektierten Eingang auslesen, OUT bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (32-Bit-Bitmuster) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AD_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

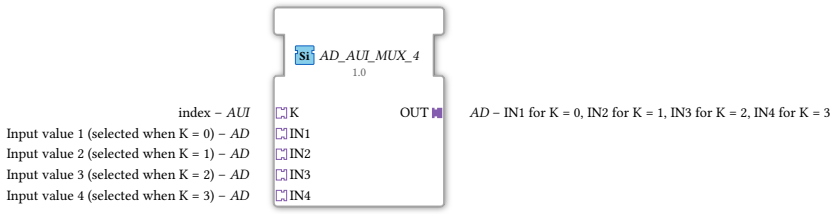
Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AD_AUI_MUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AD_AUI_MUX_3_UNGATED` überträgt die Multiplexer-Logik von `AD_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AD_AUI_MUX_4



• – ○ • – ○ • – ○ •

Einleitung

Der `AD_AUI_MUX_4` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `DWORD`. Anders als `AD_MUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AD_AUI_MUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.

- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index `K = 0` an `OUT` durchgereicht (AD-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index `K = 1` an `OUT` durchgereicht (AD-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index `K = 2` an `OUT` durchgereicht (AD-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index `K = 3` an `OUT` durchgereicht (AD-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AD-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN4`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Eingangs-Adapter (`IN1` … `IN4`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AD_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AD_AUI_MUX_2 … AD_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (32-Bit-Bitmuster) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AD_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

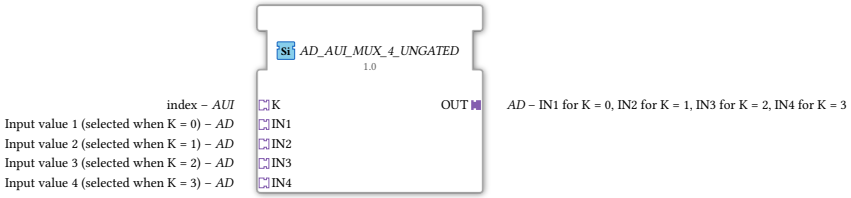
- **`AD_AUI_MUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AD_AUI_MUX_4` überträgt die Multiplexer-Logik von `AD_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AD_AUI_MUX_4_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von AD_AUI_MUX_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der AD_AUI_MUX_4_UNGATED ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp DWORD. Anders als AD_MUX_4 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AD_AUI_MUX_4_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index K = 0 an OUT durchgereicht (AD-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index K = 1 an OUT durchgereicht (AD-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index K = 2 an OUT durchgereicht (AD-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index K = 3 an OUT durchgereicht (AD-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AD-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN4:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 4 Eingangs-Adapter (IN1 … IN4) gerade ausgewählt ist.

2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AD_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AD_AUI_MUX_2 … AD_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (32-Bit-Bitmuster) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AD_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

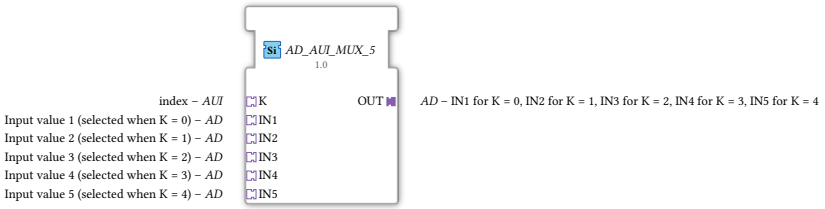
Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AD_AUI_MUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AD_AUI_MUX_4_UNGATED` überträgt die Multiplexer-Logik von `AD_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AD_AUI_MUX_5



• – ○ • – ○ • – ○ •

Einleitung

Der `AD_AUI_MUX_5` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `DWORD`. Anders als `AD_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“).

Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge
- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AD_AUI_MUX_5 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge
- Keine Ereignis-Ausgänge.

Daten-Eingänge
- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge
- Keine direkten Datenausgänge vorhanden.

Adapter
- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index K = 0 an OUT durchgereicht (AD-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index K = 1 an OUT durchgereicht (AD-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index K = 2 an OUT durchgereicht (AD-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index K = 3 an OUT durchgereicht (AD-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index K = 4 an OUT durchgereicht (AD-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AD-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN5:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 5 Eingangs-Adapter (IN1 … IN5) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf OUT gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten
- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AD_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AD_AUI_MUX_2 … AD_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN1…IN5) ⟶ aktuell selektierten Eingang auslesen, OUT bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (32-Bit-Bitmuster) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AD_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

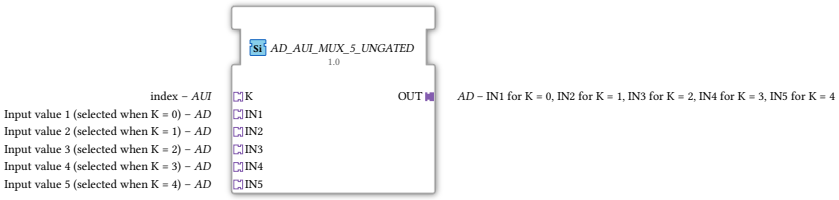
- **`AD_AUI_MUX_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AD_AUI_MUX_5` überträgt die Multiplexer-Logik von `AD_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AD_AUI_MUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AD_AUI_MUX_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `AD_AUI_MUX_5_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `DWORD`. Anders als `AD_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AD_AUI_MUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index `K = 0` an `OUT` durchgereicht (AD-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index `K = 1` an `OUT` durchgereicht (AD-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index `K = 2` an `OUT` durchgereicht (AD-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index `K = 3` an `OUT` durchgereicht (AD-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index `K = 4` an `OUT` durchgereicht (AD-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AD-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN5`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Eingangs-Adapter (`IN1` … `IN5`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AD_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AD_AUI_MUX_2 … AD_AUI_MUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (32-Bit-Bitmuster) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AD_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

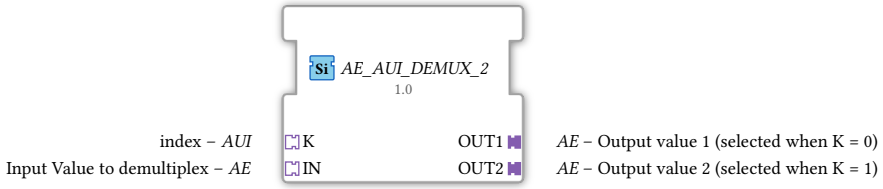
- `AD_AUI_MUX_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AD_AUI_MUX_5_UNGATED` überträgt die Multiplexer-Logik von `AD_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

EVENT

AE_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AE_AUI_DEMUX_2` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `EVENT`. Anders als `AE_DEMUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AE_AUI_DEMUX_2` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für das durchgereichte Ereignis, der an einen der `n` Ausgänge weitergeleitet wird (AE-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AE-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AE-Adaptertyp).

Funktionsweise

Der `AE_AUI_DEMUX_2` empfängt über den Socket **IN** das durchgereichte Ereignis und leitet ihn an genau einen der 2 Ausgangs-Plugs `OUT1` … `OUT2` weiter. Welcher Ausgang beschrieben wird, bestimmt der Index, der über den Adapter-Socket **K** (Typ `AUI`) hereinkommt: Sobald der am Socket K angeschlossene `AUI`-Plug sein internes `E1`-Ereignis mit dem Datenwert `D1` sendet, wertet der `AE_AUI_DEMUX_2` diesen Wert als Index (0-basiert, `0` … `1`) aus, schreibt Ereignis auf den entsprechenden `OUT`-Plug und löst dort dessen Adapter-Event aus.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AE_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AE_AUI_DEMUX_2 … AE_AUI_DEMUX_5).
- Der `AE`-Adapter überträgt ausschließlich ein Ereignis, keinen Datenwert – `AE_AUI_DEMUX_2` schaltet also reine Ereignisimpulse, keine Nutzdaten.

Zustandsübersicht

Der Baustein ist zustandslos bezüglich einer Ablaufsteuerung: Er wartet auf das `E1`-Ereignis am Socket **K**, wertet bei dessen Eintreffen den mitgelieferten Index aus und aktualisiert den betroffenen Adapter-Plug. Zwischen zwei Aufrufen wird kein Zustand außer dem zuletzt geschriebenen Adapterwert gehalten.

Anwendungsszenarien

- Verteilen eines Steuer-Ereignisses auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AE_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

Fazit

Der `AE_AUI_DEMUX_2` überträgt die Demultiplexer-Logik von `AE_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AE_AUI_DEMUX_3

• – ○ • – ○ • – ○ •

Einleitung

Der `AE_AUI_DEMUX_3` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `EVENT`. Anders als `AE_DEMUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AE_AUI_DEMUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für das durchgereichte Ereignis, der an einen der `n` Ausgänge weitergeleitet wird (AE-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AE-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AE-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AE-Adaptertyp).

Funktionsweise

Der `AE_AUI_DEMUX_3` empfängt über den Socket **IN** das durchgereichte Ereignis und leitet ihn an genau einen der 3 Ausgangs-Plugs `OUT1` … `OUT3` weiter. Welcher Ausgang beschrieben wird, bestimmt der Index, der über den Adapter-Socket **K** (Typ `AUI`) hereinkommt: Sobald der am Socket K angeschlossene `AUI`-Plug sein internes `E1`-Ereignis mit dem Datenwert `D1` sendet, wertet der `AE_AUI_DEMUX_3` diesen Wert als Index (0-basiert, `0` … `2`) aus, schreibt Ereignis auf den entsprechenden `OUT`-Plug und löst dort dessen Adapter-Event aus.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AE_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AE_AUI_DEMUX_2 … AE_AUI_DEMUX_5).
- Der `AE`-Adapter überträgt ausschließlich ein Ereignis, keinen Datenwert – `AE_AUI_DEMUX_3` schaltet also reine Ereignisimpulse, keine Nutzdaten.

Zustandsübersicht

Der Baustein ist zustandslos bezüglich einer Ablaufsteuerung: Er wartet auf das `E1`-Ereignis am Socket **K**, wertet bei dessen Eintreffen den mitgelieferten Index aus und aktualisiert den betroffenen Adapter-Plug. Zwischen zwei Aufrufen wird kein Zustand außer dem zuletzt geschriebenen Adapterwert gehalten.

Anwendungsszenarien

- Verteilen eines Steuer-Ereignisses auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

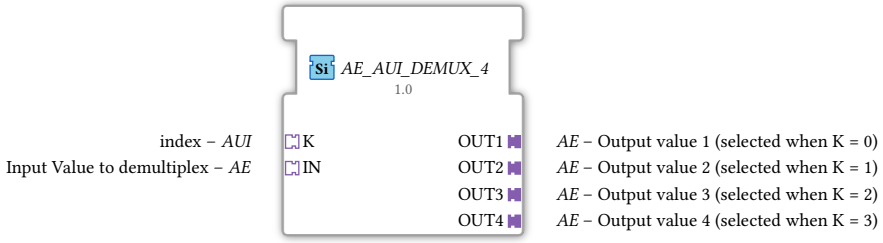
Vergleich mit `AE_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

Fazit

Der `AE_AUI_DEMUX_3` überträgt die Demultiplexer-Logik von `AE_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AE_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AE_AUI_DEMUX_4` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `EVENT`. Anders als `AE_DEMUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AE_AUI_DEMUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für das durchgereichte Ereignis, der an einen der `n` Ausgänge weitergeleitet wird (AE-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AE-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AE-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AE-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AE-Adaptertyp).

Funktionsweise

Der `AE_AUI_DEMUX_4` empfängt über den Socket **IN** das durchgereichte Ereignis und leitet ihn an genau einen der 4 Ausgangs-Plugs `OUT1` … `OUT4` weiter. Welcher Ausgang beschrieben wird, bestimmt der Index, der über den Adapter-Socket **K** (Typ `AUI`) hereinkommt: Sobald der am Socket K angeschlossene `AUI`-Plug sein internes `E1`-Ereignis mit dem Datenwert `D1` sendet, wertet der `AE_AUI_DEMUX_4` diesen Wert als Index (0-basiert, `0` … `3`) aus, schreibt Ereignis auf den entsprechenden `OUT`-Plug und löst dort dessen Adapter-Event aus.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AE_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AE_AUI_DEMUX_2 … AE_AUI_DEMUX_5).

- Der AE-Adapter überträgt ausschließlich ein Ereignis, keinen Datenwert – AE_AUI_DEMUX_4 schaltet also reine Ereignisimpulse, keine Nutzdaten.

Zustandsübersicht

Der Baustein ist zustandslos bezüglich einer Ablaufsteuerung: Er wartet auf das E1-Ereignis am Socket **K**, wertet bei dessen Eintreffen den mitgelieferten Index aus und aktualisiert den betroffenen Adapter-Plug. Zwischen zwei Aufrufen wird kein Zustand außer dem zuletzt geschriebenen Adapterwert gehalten.

Anwendungsszenarien

- Verteilen eines Steuer-Ereignisses auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

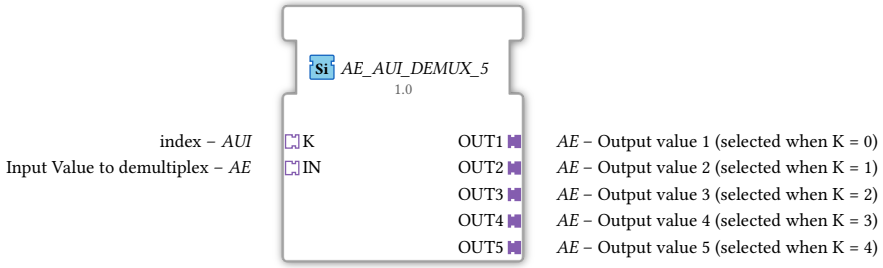
Vergleich mit AE_DEMUX_4 (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

Fazit

Der AE_AUI_DEMUX_4 überträgt die Demultiplexer-Logik von AE_DEMUX_4 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AE_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AE_AUI_DEMUX_5 ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp EVENT. Anders als AE_DEMUX_5 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AE_AUI_DEMUX_5 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für das durchgereichte Ereignis, der an einen der `n` Ausgänge weitergeleitet wird (AE-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AE-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AE-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AE-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AE-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AE-Adaptertyp).

Funktionsweise

Der `AE_AUI_DEMUX_5` empfängt über den Socket **IN** das durchgereichte Ereignis und leitet ihn an genau einen der 5 Ausgangs-Plugs `OUT1` … `OUT5` weiter. Welcher Ausgang beschrieben wird, bestimmt der Index, der über den Adapter-Socket **K** (Typ `AUI`) hereinkommt: Sobald der am Socket K angeschlossene `AUI`-Plug sein internes `E1`-Ereignis mit dem Datenwert `D1` sendet, wertet der `AE_AUI_DEMUX_5` diesen Wert als Index (0-basiert, `0` … `4`) aus, schreibt Ereignis auf den entsprechenden `OUT`-Plug und löst dort dessen Adapter-Event aus.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AE_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AE_AUI_DEMUX_2 … AE_AUI_DEMUX_5).
- Der `AE`-Adapter überträgt ausschließlich ein Ereignis, keinen Datenwert – `AE_AUI_DEMUX_5` schaltet also reine Ereignisimpulse, keine Nutzdaten.

Zustandsübersicht

Der Baustein ist zustandslos bezüglich einer Ablaufsteuerung: Er wartet auf das `E1`-Ereignis am Socket **K**, wertet bei dessen Eintreffen den mitgelieferten Index aus und aktualisiert den betroffenen Adapter-Plug. Zwischen zwei Aufrufen wird kein Zustand außer dem zuletzt geschriebenen Adapterwert gehalten.

Anwendungsszenarien

- Verteilen eines Steuer-Ereignisses auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

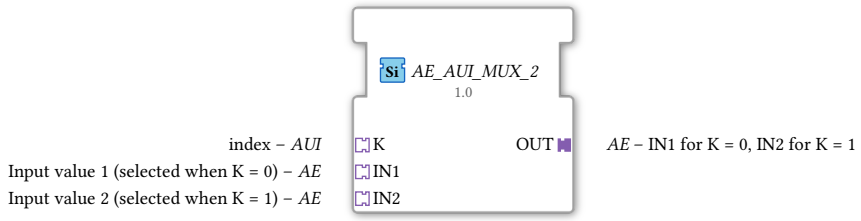
Vergleich mit `AE_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

Fazit

Der `AE_AUI_DEMUX_5` überträgt die Demultiplexer-Logik von `AE_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AE_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AE_AUI_MUX_2` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `EVENT`. Anders als `AE_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AE_AUI_MUX_2` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (AE-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (AE-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AE-Adaptertyp).

Funktionsweise

Der `AE_AUI_MUX_2` wählt über den Adapter-Socket **K** (Typ `AUI`) einen von 2 Eingangs-Sockets `IN1` … `IN2` aus und reicht dessen Wert an den einzigen Ausgangs-Plug `OUT` weiter. Sobald der an K angeschlossene `AUI`-Plug sein internes `E1`-Ereignis mit dem Datenwert `D1` sendet, interpretiert `AE_AUI_MUX_2` diesen Wert als 0-basierten Index (`0` … `1`), kopiert das durchgereichte Ereignis auf `OUT` und löst dort das Adapter-Event aus.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AE_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AE_AUI_MUX_2 … AE_AUI_MUX_5).
- Der `AE`-Adapter überträgt ausschließlich ein Ereignis, keinen Datenwert – `AE_AUI_MUX_2` schaltet also reine Ereignisimpulse, keine Nutzdaten.

Zustandsübersicht

Der Baustein ist zustandslos bezüglich einer Ablaufsteuerung: Er wartet auf das `E1`-Ereignis am Socket **K**, wertet bei dessen Eintreffen den mitgelieferten Index aus und aktualisiert den betroffenen Adapter-Plug. Zwischen zwei Aufrufen wird kein Zustand außer dem zuletzt geschriebenen Adapterwert gehalten.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Ereignisquellen über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AE_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

Fazit

Der `AE_AUI_MUX_2` überträgt die Multiplexer-Logik von `AE_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AE_AUI_MUX_3

• – ○ • – ○ • – ○ •

Einleitung

Der `AE_AUI_MUX_3` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `EVENT`. Anders als `AE_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AE_AUI_MUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (AE-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (AE-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (AE-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AE-Adaptertyp).

Funktionsweise

Der `AE_AUI_MUX_3` wählt über den Adapter-Socket **K** (Typ `AUI`) einen von 3 Eingangs-Sockets `IN1` … `IN3` aus und reicht dessen Wert an den einzigen Ausgangs-Plug `OUT` weiter. Sobald der an K angeschlossene `AUI`-Plug sein internes `E1`-Ereignis mit dem Datenwert `D1` sendet, interpretiert `AE_AUI_MUX_3` diesen Wert als 0-basierten Index (`0` … `2`), kopiert das durchgereichte Ereignis auf `OUT` und löst dort das Adapter-Event aus.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AE_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AE_AUI_MUX_2 … AE_AUI_MUX_5).
- Der `AE`-Adapter überträgt ausschließlich ein Ereignis, keinen Datenwert – `AE_AUI_MUX_3` schaltet also reine Ereignisimpulse, keine Nutzdaten.

Zustandsübersicht

Der Baustein ist zustandslos bezüglich einer Ablaufsteuerung: Er wartet auf das `E1`-Ereignis am Socket **K**, wertet bei dessen Eintreffen den mitgelieferten Index aus und aktualisiert den betroffenen Adapter-Plug. Zwischen zwei Aufrufen wird kein Zustand außer dem zuletzt geschriebenen Adapterwert gehalten.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Ereignisquellen über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

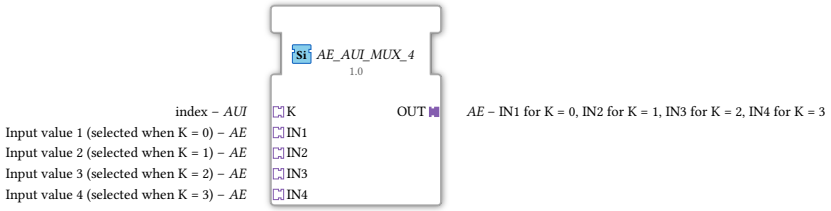
Vergleich mit `AE_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

Fazit

Der `AE_AUI_MUX_3` überträgt die Multiplexer-Logik von `AE_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AE_AUI_MUX_4



• – ○ • – ○ • – ○ •

Einleitung

Der AE_AUI_MUX_4 ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp EVENT. Anders als AE_MUX_4 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AE_AUI_MUX_4 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index K = 0 an OUT durchgereicht (AE-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index K = 1 an OUT durchgereicht (AE-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index K = 2 an OUT durchgereicht (AE-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index K = 3 an OUT durchgereicht (AE-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AE-Adaptertyp).

Funktionsweise

Der AE_AUI_MUX_4 wählt über den Adapter-Socket **K** (Typ AUI) einen von 4 Eingangs-Sockets IN1 … IN4 aus und reicht dessen Wert an den einzigen Ausgangs-Plug OUT weiter. Sobald der an K angeschlossene AUI-Plug sein internes E1-Ereignis mit dem Datenwert D1 sendet, interpretiert AE_AUI_MUX_4 diesen Wert als 0-basierten Index (0 … 3), kopiert das durchgereichte Ereignis auf OUT und löst dort das Adapter-Event aus.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AE_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AE_AUI_MUX_2 … AE_AUI_MUX_5).
- Der AE-Adapter überträgt ausschließlich ein Ereignis, keinen Datenwert – AE_AUI_MUX_4 schaltet also reine Ereignisimpulse, keine Nutzdaten.

Zustandsübersicht

Der Baustein ist zustandslos bezüglich einer Ablaufsteuerung: Er wartet auf das E1-Ereignis am Socket **K**, wertet bei dessen Eintreffen den mitgelieferten Index aus und aktualisiert den betroffenen Adapter-Plug. Zwischen zwei Aufrufen wird kein Zustand außer dem zuletzt geschriebenen Adapterwert gehalten.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Ereignisquellen über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

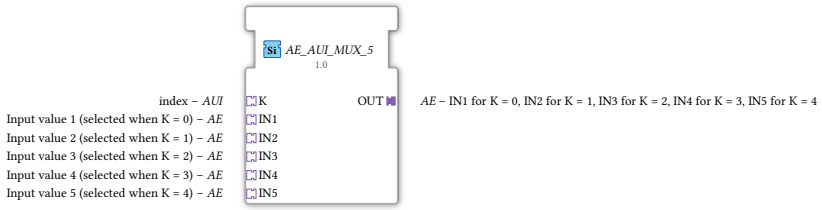
Vergleich mit `AE_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

Fazit

Der `AE_AUI_MUX_4` überträgt die Multiplexer-Logik von `AE_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AE_AUI_MUX_5



• – ○ • – ○ • – ○ •

Einleitung

Der `AE_AUI_MUX_5` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `EVENT`. Anders als `AE_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AE_AUI_MUX_5` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index `K = 0` an `OUT` durchgereicht (AE-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index `K = 1` an `OUT` durchgereicht (AE-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index `K = 2` an `OUT` durchgereicht (AE-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index `K = 3` an `OUT` durchgereicht (AE-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index `K = 4` an `OUT` durchgereicht (AE-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AE-Adaptertyp).

Funktionsweise

Der AE_AUI_MUX_5 wählt über den Adapter-Socket **K** (Typ AUI) einen von 5 Eingangs-Sockets IN1 … IN5 aus und reicht dessen Wert an den einzigen Ausgangs-Plug OUT weiter. Sobald der an K angeschlossene AUI-Plug sein internes E1-Ereignis mit dem Datenwert D1 sendet, interpretiert AE_AUI_MUX_5 diesen Wert als 0-basierten Index (0 … 4), kopiert das durchgereichte Ereignis auf OUT und löst dort das Adapter-Event aus.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AE_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AE_AUI_MUX_2 … AE_AUI_MUX_5).
- Der AE-Adapter überträgt ausschließlich ein Ereignis, keinen Datenwert – AE_AUI_MUX_5 schaltet also reine Ereignisimpulse, keine Nutzdaten.

Zustandsübersicht

Der Baustein ist zustandslos bezüglich einer Ablaufsteuerung: Er wartet auf das E1-Ereignis am Socket **K**, wertet bei dessen Eintreffen den mitgelieferten Index aus und aktualisiert den betroffenen Adapter-Plug. Zwischen zwei Aufrufen wird kein Zustand außer dem zuletzt geschriebenen Adapterwert gehalten.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Ereignisquellen über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit AE_MUX_5 (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

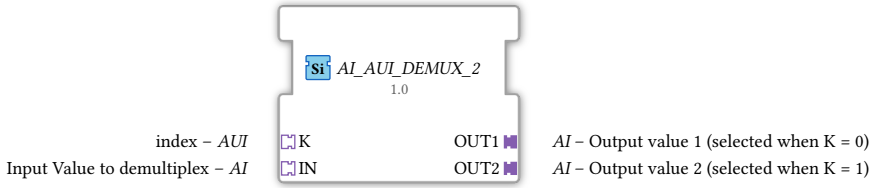
Für die Anzahl der Eingänge existiert kein datenbasiertes F_MUX-Pendant in der iec61131-3-Bibliothek.

Fazit

Der AE_AUI_MUX_5 überträgt die Multiplexer-Logik von AE_MUX_5 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

INT

AI_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AI_AUI_DEMUX_2 ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp INT. Anders als AI_DEMUX_2 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AI_AUI_DEMUX_2` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`INT`, 16-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der `n` Ausgänge weitergeleitet wird (AI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Ausgangs-Adapter (`OUT1` … `OUT2`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AI_AUI_DEMUX_2 … AI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`INT`, 16-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)

- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AI_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

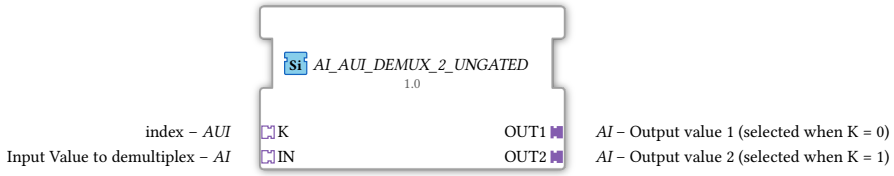
- **`AI_AUI_DEMUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AI_AUI_DEMUX_2` überträgt die Demultiplexer-Logik von `AI_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AI_AUI_DEMUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AI_AUI_DEMUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AI_AUI_DEMUX_2_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `INT`. Anders als `AI_DEMUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AI_AUI_DEMUX_2_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`INT`, 16-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der `n` Ausgänge weitergeleitet wird (AI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Ausgangs-Adapter (`OUT1` … `OUT2`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AI_AUI_DEMUX_2_UNGATED … AI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`INT`, 16-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AI_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

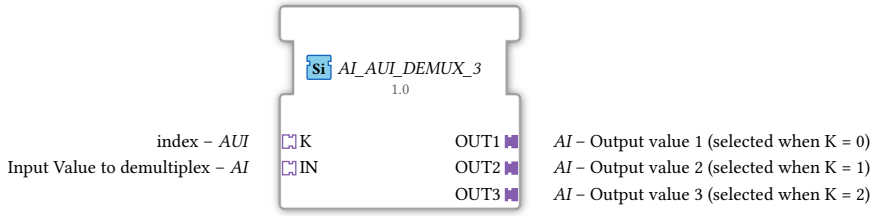
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AI_AUI_DEMUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AI_AUI_DEMUX_2_UNGATED` überträgt die Demultiplexer-Logik von `AI_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AI_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AI_AUI_DEMUX_3` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `INT`. Anders als `AI_DEMUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AI_AUI_DEMUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`INT`, 16-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der `n` Ausgänge weitergeleitet wird (AI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Ausgangs-Adapter (`OUT1` … `OUT3`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.

- Generische Implementierung (`GEN_AI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AI_AUI_DEMUX_2 … AI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`INT`, 16-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AI_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

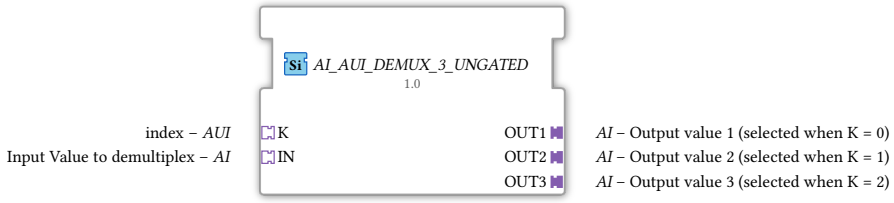
- **`AI_AUI_DEMUX_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AI_AUI_DEMUX_3` überträgt die Demultiplexer-Logik von `AI_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AI_AUI_DEMUX_3_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `AI_AUI_DEMUX_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AI_AUI_DEMUX_3_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `INT`. Anders als `AI_DEMUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter

Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AI_AUI_DEMUX_3_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (INT, 16-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der n Ausgänge weitergeleitet wird (AI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 3 Ausgangs-Adapter (OUT1 … OUT3) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AI_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AI_AUI_DEMUX_2 … AI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen IN-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`INT`, 16-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AI_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

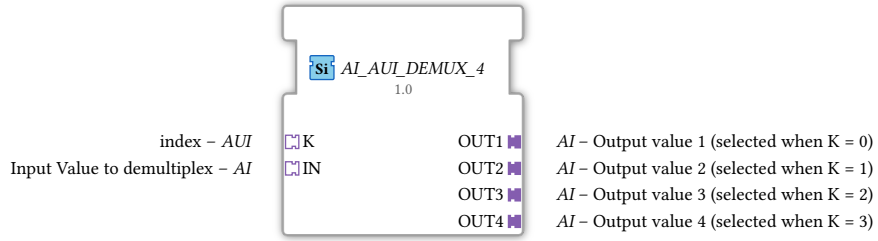
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AI_AUI_DEMUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AI_AUI_DEMUX_3_UNGATED` überträgt die Demultiplexer-Logik von `AI_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AI_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AI_AUI_DEMUX_4` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `INT`. Anders als `AI_DEMUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AI_AUI_DEMUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`INT`, 16-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der `n` Ausgänge weitergeleitet wird (AI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AI-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AI_AUI_DEMUX_2 … AI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`INT`, 16-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AI_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

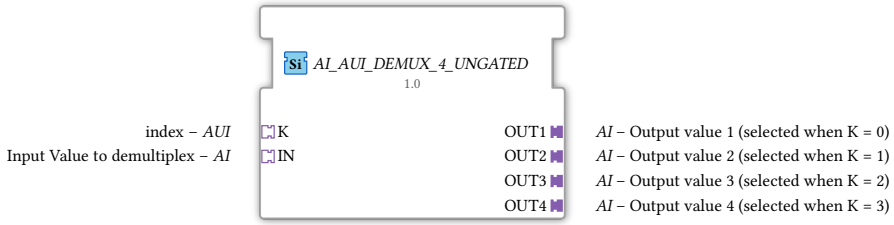
- **`AI_AUI_DEMUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AI_AUI_DEMUX_4` überträgt die Demultiplexer-Logik von `AI_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AI_AUI_DEMUX_4_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von AI_AUI_DEMUX_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AI_AUI_DEMUX_4_UNGATED ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp INT. Anders als AI_DEMUX_4 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AI_AUI_DEMUX_4_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (INT, 16-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der n Ausgänge weitergeleitet wird (AI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AI-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AI_AUI_DEMUX_2 … AI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`INT`, 16-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AI_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

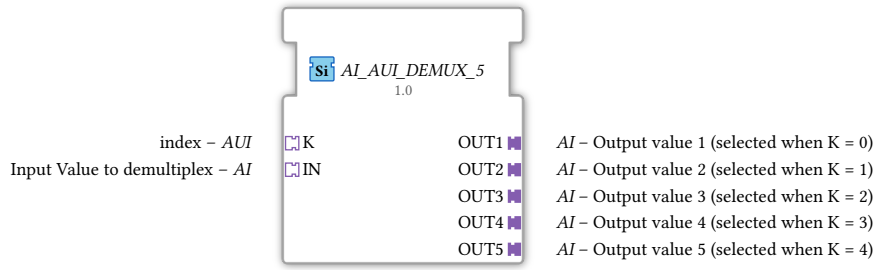
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AI_AUI_DEMUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AI_AUI_DEMUX_4_UNGATED` überträgt die Demultiplexer-Logik von `AI_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AI_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AI_AUI_DEMUX_5 ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp INT. Anders als AI_DEMUX_5 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AI_AUI_DEMUX_5 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (INT, 16-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der n Ausgänge weitergeleitet wird (AI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AI-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AI-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 5 Ausgangs-Adapter (OUT1 … OUT5) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AI_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AI_AUI_DEMUX_2 … AI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`INT`, 16-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AI_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

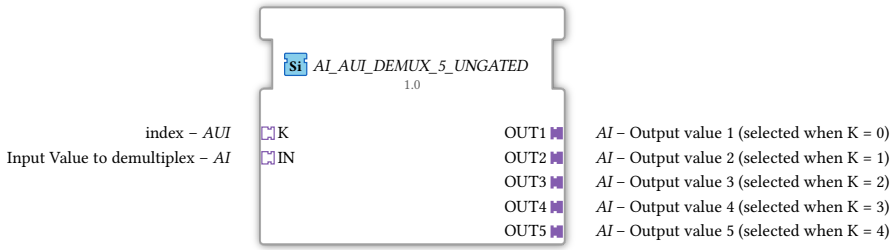
- **`AI_AUI_DEMUX_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AI_AUI_DEMUX_5` überträgt die Demultiplexer-Logik von `AI_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AI_AUI_DEMUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AI_AUI_DEMUX_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `AI_AUI_DEMUX_5_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `INT`. Anders als `AI_DEMUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AI_AUI_DEMUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`INT`, 16-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der `n` Ausgänge weitergeleitet wird (AI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AI-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AI-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der `5` Ausgangs-Adapter (`OUT1` … `OUT5`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AI_AUI_DEMUX_2 … AI_AUI_DEMUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`INT`, 16-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AI_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

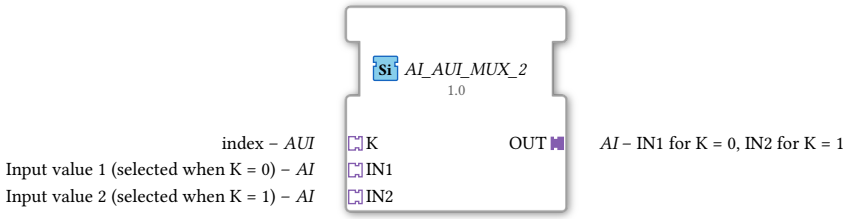
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AI_AUI_DEMUX_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AI_AUI_DEMUX_5_UNGATED` überträgt die Demultiplexer-Logik von `AI_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AI_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AI_AUI_MUX_2` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `INT`. Anders als `AI_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AI_AUI_MUX_2` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.

- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (AI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (AI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AI_AUI_MUX_2 … AI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (16-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AI_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AI_AUI_MUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

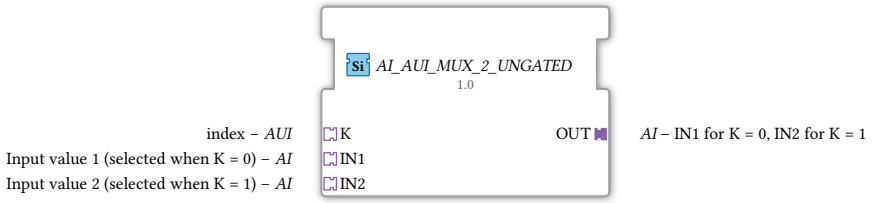
Fazit

Der `AI_AUI_MUX_2` überträgt die Multiplexer-Logik von `AI_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AI_AUI_MUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AI_AUI_MUX_2. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos

weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AI_AUI_MUX_2_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `INT`. Anders als `AI_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AI_AUI_MUX_2_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (AI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (AI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AI_AUI_MUX_2_UNGATED … AI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (16-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AI_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AI_AUI_MUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AI_AUI_MUX_2_UNGATED` überträgt die Multiplexer-Logik von `AI_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AI_AUI_MUX_3

• – ○ • – ○ • – ○ •

Einleitung

Der `AI_AUI_MUX_3` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `INT`. Anders als `AI_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AI_AUI_MUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (AI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (AI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (AI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN3`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Eingangs-Adapter (`IN1` … `IN3`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AI_AUI_MUX_2 … AI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (16-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird

- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AI_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

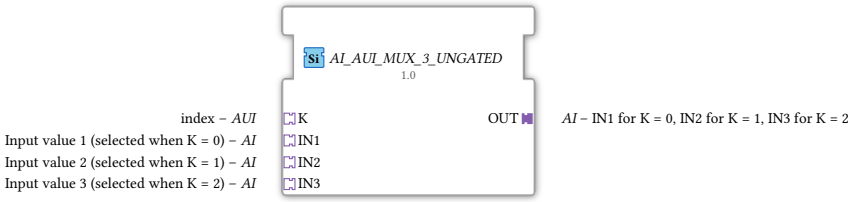
- **`AI_AUI_MUX_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AI_AUI_MUX_3` überträgt die Multiplexer-Logik von `AI_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AI_AUI_MUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AI_AUI_MUX_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `AI_AUI_MUX_3_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `INT`. Anders als `AI_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AI_AUI_MUX_3_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (AI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (AI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (AI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN3`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Eingangs-Adapter (`IN1` … `IN3`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AI_AUI_MUX_2 … AI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (16-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AI_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

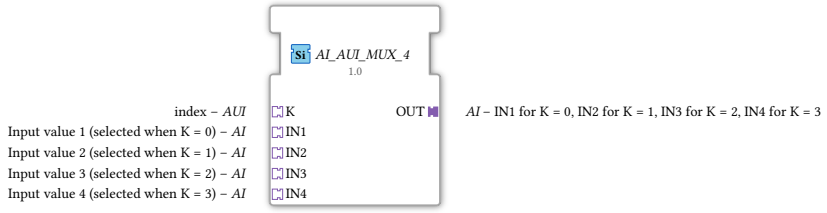
Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AI_AUI_MUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AI_AUI_MUX_3_UNGATED` überträgt die Multiplexer-Logik von `AI_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AI_AUI_MUX_4



• – ○ • – ○ • – ○ •

Einleitung

Der `AI_AUI_MUX_4` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `INT`. Anders als `AI_MUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AI_AUI_MUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index `K = 0` an `OUT` durchgereicht (AI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index `K = 1` an `OUT` durchgereicht (AI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index `K = 2` an `OUT` durchgereicht (AI-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index `K = 3` an `OUT` durchgereicht (AI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN4`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Eingangs-Adapter (`IN1` … `IN4`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AI_AUI_MUX_2 … AI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (16-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AI_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

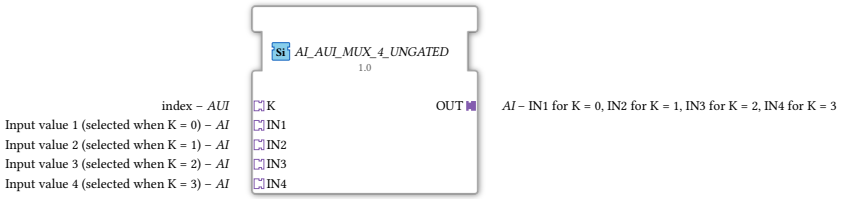
- **`AI_AUI_MUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AI_AUI_MUX_4` überträgt die Multiplexer-Logik von `AI_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AI_AUI_MUX_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AI_AUI_MUX_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der AI_AUI_MUX_4_UNGATED ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp INT. Anders als AI_MUX_4 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AI_AUI_MUX_4_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index K = 0 an OUT durchgereicht (AI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index K = 1 an OUT durchgereicht (AI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index K = 2 an OUT durchgereicht (AI-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index K = 3 an OUT durchgereicht (AI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN4:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 4 Eingangs-Adapter (IN1 … IN4) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf OUT gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AI_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AI_AUI_MUX_2 … AI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (16-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AI_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

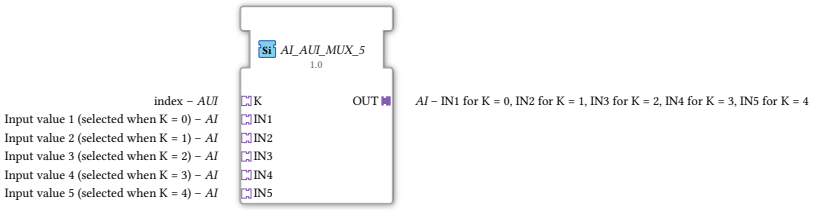
Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AI_AUI_MUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AI_AUI_MUX_4_UNGATED` überträgt die Multiplexer-Logik von `AI_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AI_AUI_MUX_5



• – ○ • – ○ • – ○ •

Einleitung

Der `AI_AUI_MUX_5` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `INT`. Anders als `AI_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AI_AUI_MUX_5` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index `K = 0` an `OUT` durchgereicht (AI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index `K = 1` an `OUT` durchgereicht (AI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index `K = 2` an `OUT` durchgereicht (AI-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index `K = 3` an `OUT` durchgereicht (AI-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index `K = 4` an `OUT` durchgereicht (AI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN5`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Eingangs-Adapter (`IN1` … `IN5`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AI_AUI_MUX_2 … AI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (16-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AI_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

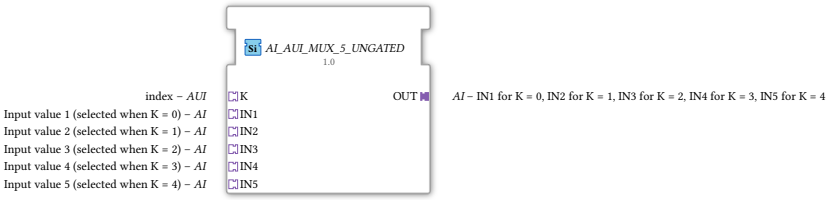
- **`AI_AUI_MUX_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AI_AUI_MUX_5` überträgt die Multiplexer-Logik von `AI_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AI_AUI_MUX_5_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von AI_AUI_MUX_5. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `AI_AUI_MUX_5_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `INT`. Anders als `AI_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AI_AUI_MUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index `K = 0` an `OUT` durchgereicht (AI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index `K = 1` an `OUT` durchgereicht (AI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index `K = 2` an `OUT` durchgereicht (AI-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index `K = 3` an `OUT` durchgereicht (AI-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index `K = 4` an `OUT` durchgereicht (AI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN5`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Eingangs-Adapter (`IN1` … `IN5`) gerade ausgewählt ist.

2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AI_AUI_MUX_2 … AI_AUI_MUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (16-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AI_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

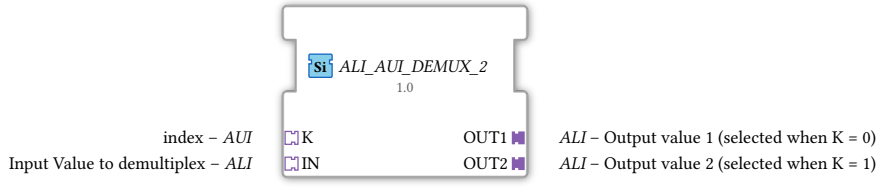
- `AI_AUI_MUX_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AI_AUI_MUX_5_UNGATED` überträgt die Multiplexer-Logik von `AI_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

LINT

ALI_AUI_DEMUX_2



• – ○ • – ○ • – ○ •

Einleitung

Der `ALI_AUI_DEMUX_2` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `LINT`. Anders als `ALI_DEMUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional

Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet ALI_AUI_DEMUX_2 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (LINT, 64-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der n Ausgänge weitergeleitet wird (ALI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (ALI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (ALI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 2 Ausgangs-Adapter (OUT1 … OUT2) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_ALI_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALI_AUI_DEMUX_2 … ALI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen IN-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (LINT, 64-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index

- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ALI_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

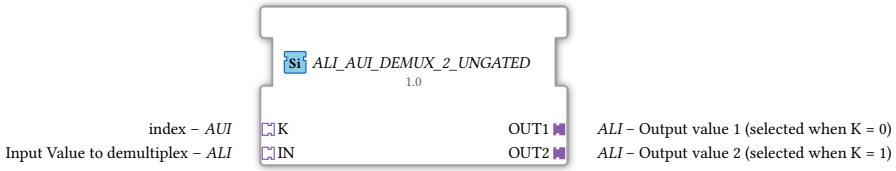
- **`ALI_AUI_DEMUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `ALI_AUI_DEMUX_2` überträgt die Demultiplexer-Logik von `ALI_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALI_AUI_DEMUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALI_AUI_DEMUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ALI_AUI_DEMUX_2_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `LINT`. Anders als `ALI_DEMUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ALI_AUI_DEMUX_2_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`LINT`, 64-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der `n` Ausgänge weitergeleitet wird (ALI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (ALI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (ALI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Ausgangs-Adapter (`OUT1` … `OUT2`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ALI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALI_AUI_DEMUX_2_UNGATED … ALI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`LINT`, 64-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ALI_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

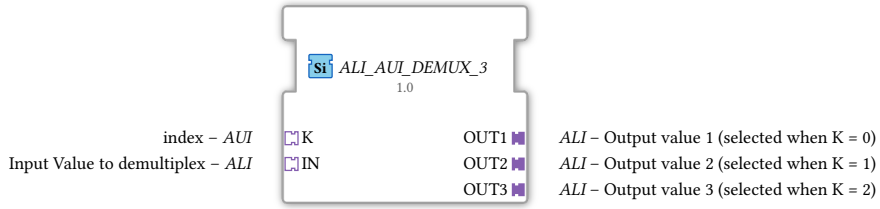
- **`ALI_AUI_DEMUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `ALI_AUI_DEMUX_2_UNGATED` überträgt die Demultiplexer-Logik von `ALI_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als

AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALI_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der ALI_AUI_DEMUX_3 ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp LINT. Anders als ALI_DEMUX_3 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet ALI_AUI_DEMUX_3 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (LINT, 64-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der n Ausgänge weitergeleitet wird (ALI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (ALI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (ALI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (ALI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 3 Ausgangs-Adapter (OUT1 … OUT3) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ALI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALI_AUI_DEMUX_2 … ALI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`LINT`, 64-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ALI_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

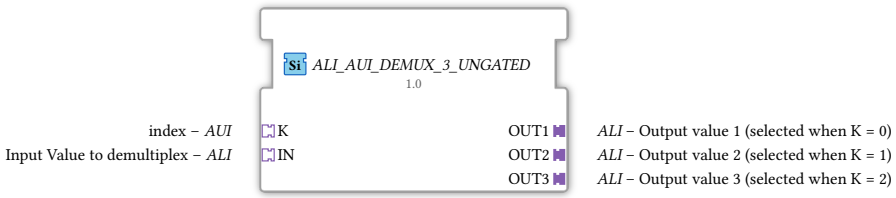
- **`ALI_AUI_DEMUX_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `ALI_AUI_DEMUX_3` überträgt die Demultiplexer-Logik von `ALI_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALI_AUI_DEMUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALI_AUI_DEMUX_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der ALI_AUI_DEMUX_3_UNGATED ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp LINT. Anders als ALI_DEMUX_3 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet ALI_AUI_DEMUX_3_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (LINT, 64-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der n Ausgänge weitergeleitet wird (ALI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (ALI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (ALI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (ALI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 3 Ausgangs-Adapter (OUT1 … OUT3) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_ALI_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALI_AUI_DEMUX_2 … ALI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`LINT`, 64-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

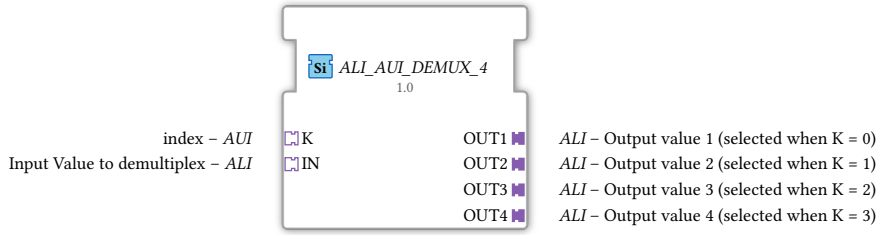
Vergleich mit `ALI_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`ALI_AUI_DEMUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `ALI_AUI_DEMUX_3_UNGATED` überträgt die Demultiplexer-Logik von `ALI_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALI_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ALI_AUI_DEMUX_4` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `LINT`. Anders als `ALI_DEMUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ALI_AUI_DEMUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`LINT`, 64-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der `n` Ausgänge weitergeleitet wird (ALI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (ALI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (ALI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (ALI-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (ALI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ALI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALI_AUI_DEMUX_2 … ALI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`LINT`, 64-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ALI_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

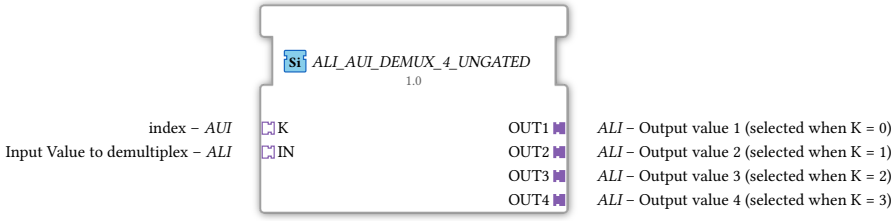
- **`ALI_AUI_DEMUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `ALI_AUI_DEMUX_4` überträgt die Demultiplexer-Logik von `ALI_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALI_AUI_DEMUX_4_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von ALI_AUI_DEMUX_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der ALI_AUI_DEMUX_4_UNGATED ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp LINT. Anders als ALI_DEMUX_4 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet ALI_AUI_DEMUX_4_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (LINT, 64-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der n Ausgänge weitergeleitet wird (ALI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (ALI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (ALI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (ALI-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (ALI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ALI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALI_AUI_DEMUX_2 … ALI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`LINT`, 64-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ALI_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

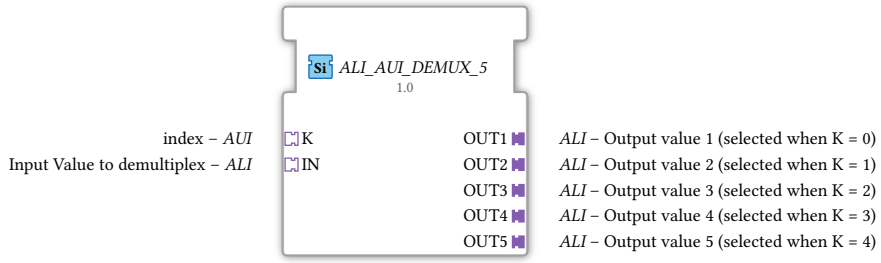
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`ALI_AUI_DEMUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `ALI_AUI_DEMUX_4_UNGATED` überträgt die Demultiplexer-Logik von `ALI_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALI_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der ALI_AUI_DEMUX_5 ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp LINT. Anders als ALI_DEMUX_5 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet ALI_AUI_DEMUX_5 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (LINT, 64-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der n Ausgänge weitergeleitet wird (ALI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (ALI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (ALI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (ALI-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (ALI-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (ALI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 5 Ausgangs-Adapter (OUT1 … OUT5) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_ALI_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALI_AUI_DEMUX_2 … ALI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`LINT`, 64-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ALI_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

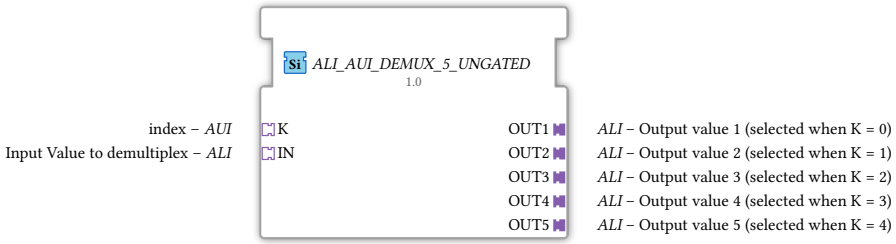
- **`ALI_AUI_DEMUX_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `ALI_AUI_DEMUX_5` überträgt die Demultiplexer-Logik von `ALI_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALI_AUI_DEMUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALI_AUI_DEMUX_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ALI_AUI_DEMUX_5_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `LINT`. Anders als `ALI_DEMUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ALI_AUI_DEMUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`LINT`, 64-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der `n` Ausgänge weitergeleitet wird (ALI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (ALI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (ALI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (ALI-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (ALI-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (ALI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der `5` Ausgangs-Adapter (`OUT1` … `OUT5`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ALI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALI_AUI_DEMUX_2 … ALI_AUI_DEMUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`LINT`, 64-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ALI_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

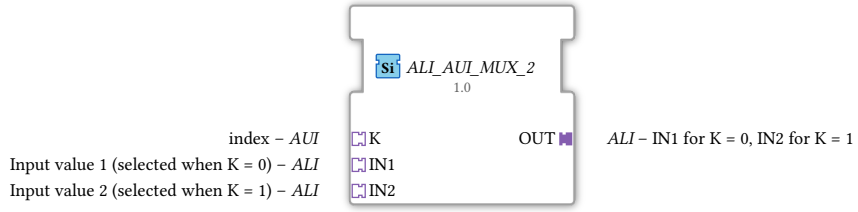
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`ALI_AUI_DEMUX_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `ALI_AUI_DEMUX_5_UNGATED` überträgt die Demultiplexer-Logik von `ALI_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALI_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ALI_AUI_MUX_2` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `LINT`. Anders als `ALI_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ALI_AUI_MUX_2` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (ALI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (ALI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ALI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ALI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALI_AUI_MUX_2 … ALI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (64-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ALI_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

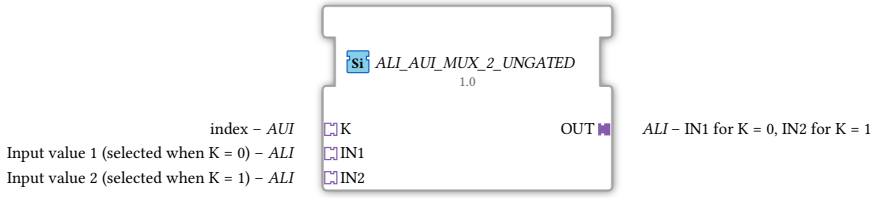
- **`ALI_AUI_MUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `ALI_AUI_MUX_2` überträgt die Multiplexer-Logik von `ALI_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALI_AUI_MUX_2_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von ALI_AUI_MUX_2. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der ALI_AUI_MUX_2_UNGATED ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp LINT. Anders als ALI_MUX_2 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet ALI_AUI_MUX_2_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index K = 0 an OUT durchgereicht (ALI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index K = 1 an OUT durchgereicht (ALI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ALI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN2:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 2 Eingangs-Adapter (IN1 … IN2) gerade ausgewählt ist.

2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ALI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALI_AUI_MUX_2_UNGATED … ALI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (64-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ALI_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`ALI_AUI_MUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `ALI_AUI_MUX_2_UNGATED` überträgt die Multiplexer-Logik von `ALI_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALI_AUI_MUX_3

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ALI_AUI_MUX_3` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `LINT`. Anders als `ALI_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“).

Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge
- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet ALI_AUI_MUX_3 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge
- Keine Ereignis-Ausgänge.

Daten-Eingänge
- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge
- Keine direkten Datenausgänge vorhanden.

Adapter
- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index K = 0 an OUT durchgereicht (ALI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index K = 1 an OUT durchgereicht (ALI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index K = 2 an OUT durchgereicht (ALI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ALI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN3:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 3 Eingangs-Adapter (IN1 … IN3) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf OUT gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten
- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_ALI_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALI_AUI_MUX_2 … ALI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN1…IN3) ⟶ aktuell selektierten Eingang auslesen, OUT bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (64-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ALI_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

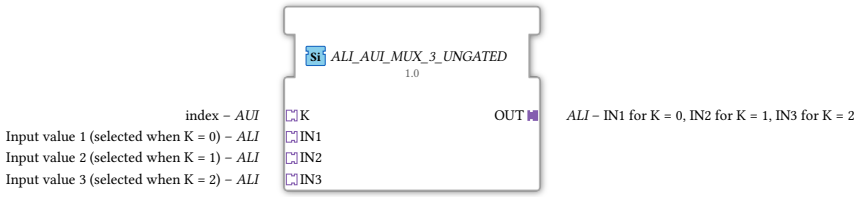
- **`ALI_AUI_MUX_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `ALI_AUI_MUX_3` überträgt die Multiplexer-Logik von `ALI_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALI_AUI_MUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALI_AUI_MUX_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `ALI_AUI_MUX_3_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `LINT`. Anders als `ALI_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ALI_AUI_MUX_3_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (ALI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (ALI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (ALI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ALI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN3`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Eingangs-Adapter (`IN1` … `IN3`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ALI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALI_AUI_MUX_2 … ALI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (64-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ALI_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`ALI_AUI_MUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `ALI_AUI_MUX_3_UNGATED` überträgt die Multiplexer-Logik von `ALI_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALI_AUI_MUX_4

• — ○ • — ○ • — ○ •

Einleitung

Der `ALI_AUI_MUX_4` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `LINT`. Anders als `ALI_MUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ALI_AUI_MUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index `K = 0` an `OUT` durchgereicht (ALI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index `K = 1` an `OUT` durchgereicht (ALI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index `K = 2` an `OUT` durchgereicht (ALI-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index `K = 3` an `OUT` durchgereicht (ALI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ALI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN4`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Eingangs-Adapter (`IN1` … `IN4`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_ALI_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALI_AUI_MUX_2 … ALI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN1…IN4) ⟶ aktuell selektierten Eingang auslesen, OUT bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (64-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit ALI_MUX_4 (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **ALI_AUI_MUX_4_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der ALI_AUI_MUX_4 überträgt die Multiplexer-Logik von ALI_MUX_4 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALI_AUI_MUX_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von ALI_AUI_MUX_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

Einleitung

Der `ALI_AUI_MUX_4_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `LINT`. Anders als `ALI_MUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ALI_AUI_MUX_4_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index `K = 0` an `OUT` durchgereicht (ALI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index `K = 1` an `OUT` durchgereicht (ALI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index `K = 2` an `OUT` durchgereicht (ALI-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index `K = 3` an `OUT` durchgereicht (ALI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ALI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN4`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Eingangs-Adapter (`IN1` … `IN4`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ALI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALI_AUI_MUX_2 … ALI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (64-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ALI_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`ALI_AUI_MUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `ALI_AUI_MUX_4_UNGATED` überträgt die Multiplexer-Logik von `ALI_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALI_AUI_MUX_5

• – ○ • – ○ • – ○ •

Einleitung

Der `ALI_AUI_MUX_5` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `LINT`. Anders als `ALI_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ALI_AUI_MUX_5` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index `K = 0` an `OUT` durchgereicht (ALI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index `K = 1` an `OUT` durchgereicht (ALI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index `K = 2` an `OUT` durchgereicht (ALI-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index `K = 3` an `OUT` durchgereicht (ALI-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index `K = 4` an `OUT` durchgereicht (ALI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ALI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN5`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Eingangs-Adapter (`IN1` … `IN5`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ALI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALI_AUI_MUX_2 … ALI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (64-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ALI_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

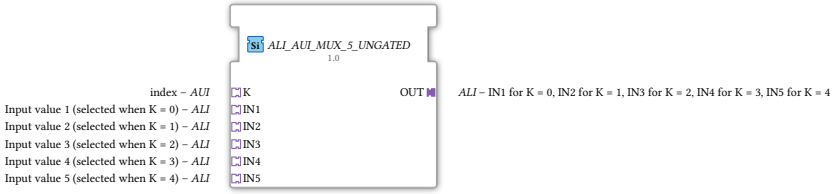
- **`ALI_AUI_MUX_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `ALI_AUI_MUX_5` überträgt die Multiplexer-Logik von `ALI_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALI_AUI_MUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALI_AUI_MUX_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `ALI_AUI_MUX_5_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `LINT`. Anders als `ALI_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ALI_AUI_MUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index `K = 0` an `OUT` durchgereicht (ALI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index `K = 1` an `OUT` durchgereicht (ALI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index `K = 2` an `OUT` durchgereicht (ALI-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index `K = 3` an `OUT` durchgereicht (ALI-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index `K = 4` an `OUT` durchgereicht (ALI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ALI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`...`IN5`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Eingangs-Adapter (`IN1` ... `IN5`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ALI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALI_AUI_MUX_2 ... ALI_AUI_MUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`...`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (64-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ALI_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

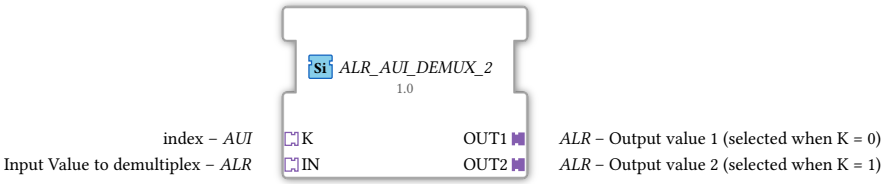
- `ALI_AUI_MUX_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `ALI_AUI_MUX_5_UNGATED` überträgt die Multiplexer-Logik von `ALI_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

LREAL

ALR_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der ALR_AUI_DEMUX_2 ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp LREAL. Anders als ALR_DEMUX_2 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet ALR_AUI_DEMUX_2 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (LREAL, 64-Bit-Fließkommazahl), der an einen der n Ausgänge weitergeleitet wird (ALR-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (ALR-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (ALR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 2 Ausgangs-Adapter (OUT1 … OUT2) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_ALR_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALR_AUI_DEMUX_2 … ALR_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`LREAL`, 64-Bit-Fließkommazahl) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ALR_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

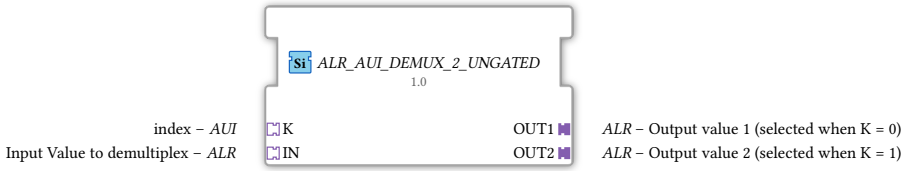
- **`ALR_AUI_DEMUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `ALR_AUI_DEMUX_2` überträgt die Demultiplexer-Logik von `ALR_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALR_AUI_DEMUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALR_AUI_DEMUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ALR_AUI_DEMUX_2_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `LREAL`. Anders als `ALR_DEMUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ALR_AUI_DEMUX_2_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`LREAL`, 64-Bit-Fließkommazahl), der an einen der `n` Ausgänge weitergeleitet wird (ALR-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (ALR-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (ALR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Ausgangs-Adapter (`OUT1` … `OUT2`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ALR_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALR_AUI_DEMUX_2_UNGATED … ALR_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`LREAL`, 64-Bit-Fließkommazahl) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ALR_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

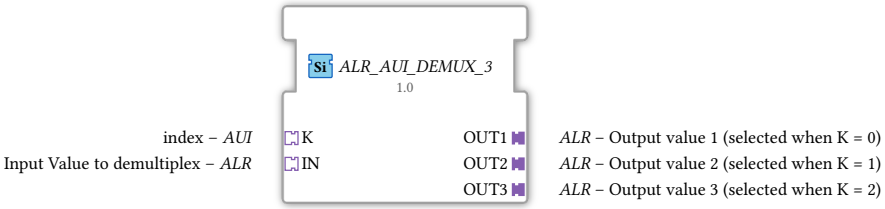
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **ALR_AUI_DEMUX_2**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der ALR_AUI_DEMUX_2_UNGATED überträgt die Demultiplexer-Logik von ALR_DEMUX_2 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALR_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der ALR_AUI_DEMUX_3 ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp LREAL. Anders als ALR_DEMUX_3 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet ALR_AUI_DEMUX_3 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (LREAL, 64-Bit-Fließkommazahl), der an einen der n Ausgänge weitergeleitet wird (ALR-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (ALR-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (ALR-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (ALR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 3 Ausgangs-Adapter (OUT1 … OUT3) gerade ausgewählt ist.

2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_ALR_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALR_AUI_DEMUX_2 … ALR_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen IN-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (LREAL, 64-Bit-Fließkommazahl) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit ALR_DEMUX_3 (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

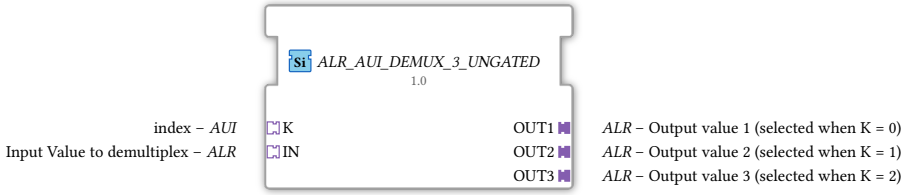
- **ALR_AUI_DEMUX_3_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der ALR_AUI_DEMUX_3 überträgt die Demultiplexer-Logik von ALR_DEMUX_3 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALR_AUI_DEMUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von ALR_AUI_DEMUX_3. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ALR_AUI_DEMUX_3_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `LREAL`. Anders als `ALR_DEMUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ALR_AUI_DEMUX_3_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`LREAL`, 64-Bit-Fließkommazahl), der an einen der `n` Ausgänge weitergeleitet wird (ALR-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (ALR-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (ALR-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (ALR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Ausgangs-Adapter (`OUT1` … `OUT3`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.

- Generische Implementierung (`GEN_ALR_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALR_AUI_DEMUX_2 … ALR_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`LREAL`, 64-Bit-Fließkommazahl) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ALR_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

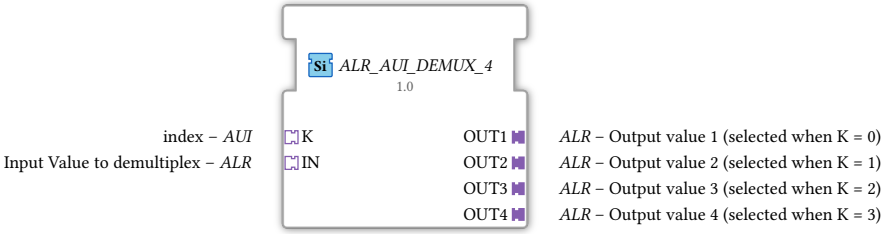
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`ALR_AUI_DEMUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `ALR_AUI_DEMUX_3_UNGATED` überträgt die Demultiplexer-Logik von `ALR_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALR_AUI_DEMUX_4



• – ○ • – ○ • – ○ •

Einleitung

Der `ALR_AUI_DEMUX_4` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `LREAL`. Anders als `ALR_DEMUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ALR_AUI_DEMUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`LREAL`, 64-Bit-Fließkommazahl), der an einen der `n` Ausgänge weitergeleitet wird (ALR-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (ALR-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (ALR-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (ALR-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (ALR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ALR_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALR_AUI_DEMUX_2 … ALR_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`LREAL`, 64-Bit-Fließkommazahl) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index

- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ALR_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

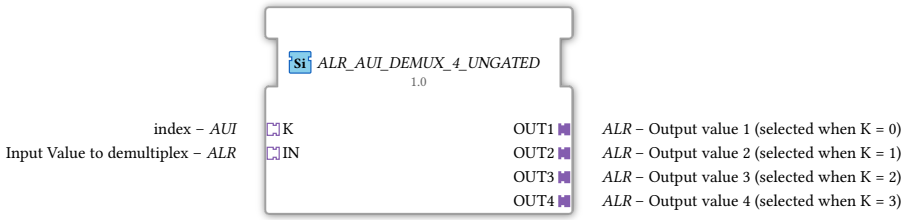
- **`ALR_AUI_DEMUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `ALR_AUI_DEMUX_4` überträgt die Demultiplexer-Logik von `ALR_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALR_AUI_DEMUX_4_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `ALR_AUI_DEMUX_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ALR_AUI_DEMUX_4_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `LREAL`. Anders als `ALR_DEMUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ALR_AUI_DEMUX_4_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`LREAL`, 64-Bit-Fließkommazahl), der an einen der `n` Ausgänge weitergeleitet wird (ALR-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (ALR-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (ALR-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (ALR-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (ALR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ALR_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALR_AUI_DEMUX_2 … ALR_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`LREAL`, 64-Bit-Fließkommazahl) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

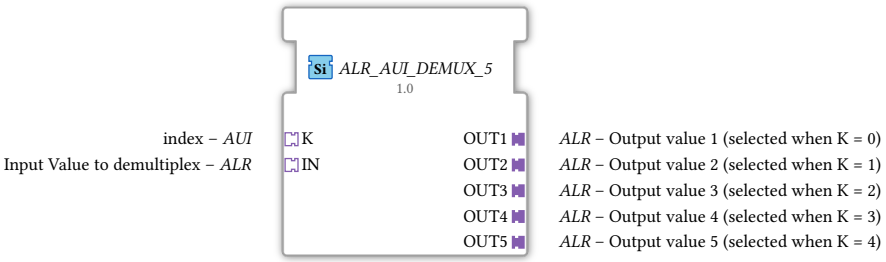
Vergleich mit `ALR_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`ALR_AUI_DEMUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `ALR_AUI_DEMUX_4_UNGATED` überträgt die Demultiplexer-Logik von `ALR_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALR_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ALR_AUI_DEMUX_5` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `LREAL`. Anders als `ALR_DEMUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ALR_AUI_DEMUX_5` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`LREAL`, 64-Bit-Fließkommazahl), der an einen der `n` Ausgänge weitergeleitet wird (ALR-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (ALR-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (ALR-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (ALR-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (ALR-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (ALR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Ausgangs-Adapter (`OUT1` … `OUT5`) gerade ausgewählt ist.

2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ALR_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALR_AUI_DEMUX_2 … ALR_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`LREAL`, 64-Bit-Fließkommazahl) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ALR_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

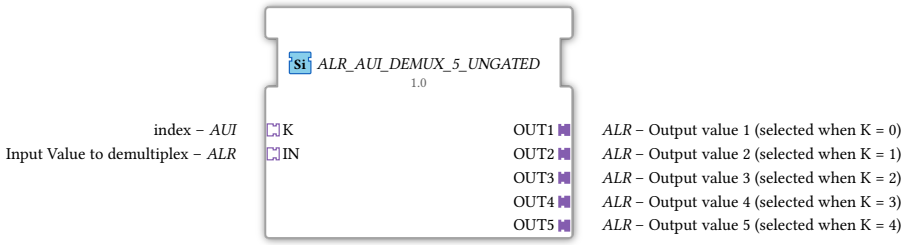
- **`ALR_AUI_DEMUX_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `ALR_AUI_DEMUX_5` überträgt die Demultiplexer-Logik von `ALR_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALR_AUI_DEMUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALR_AUI_DEMUX_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ALR_AUI_DEMUX_5_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `LREAL`. Anders als `ALR_DEMUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ALR_AUI_DEMUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`LREAL`, 64-Bit-Fließkommazahl), der an einen der `n` Ausgänge weitergeleitet wird (ALR-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (ALR-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (ALR-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (ALR-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (ALR-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (ALR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Ausgangs-Adapter (`OUT1` … `OUT5`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ALR_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALR_AUI_DEMUX_2 … ALR_AUI_DEMUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`LREAL`, 64-Bit-Fließkommazahl) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

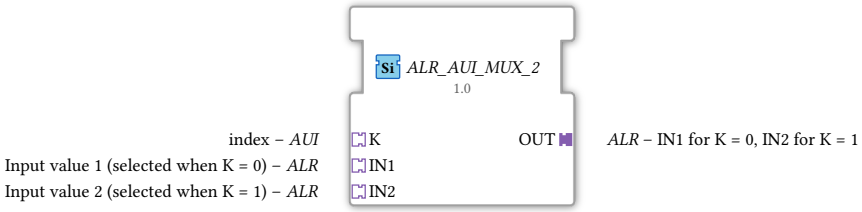
Vergleich mit `ALR_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`ALR_AUI_DEMUX_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `ALR_AUI_DEMUX_5_UNGATED` überträgt die Demultiplexer-Logik von `ALR_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALR_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ALR_AUI_MUX_2` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `LREAL`. Anders als `ALR_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ALR_AUI_MUX_2` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (ALR-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (ALR-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ALR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ALR_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALR_AUI_MUX_2 … ALR_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (64-Bit-Fließkommazahl) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird

- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ALR_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

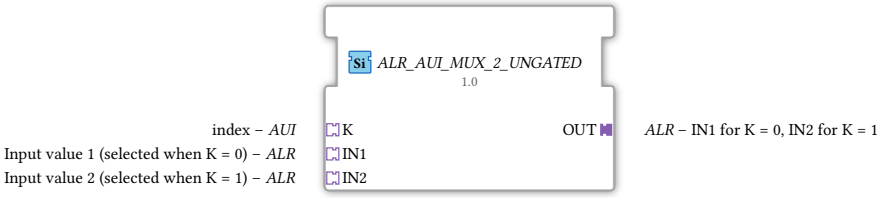
- **`ALR_AUI_MUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `ALR_AUI_MUX_2` überträgt die Multiplexer-Logik von `ALR_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALR_AUI_MUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALR_AUI_MUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ALR_AUI_MUX_2_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `LREAL`. Anders als `ALR_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ALR_AUI_MUX_2_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (ALR-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (ALR-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ALR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ALR_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALR_AUI_MUX_2_UNGATED … ALR_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (64-Bit-Fließkommazahl) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ALR_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

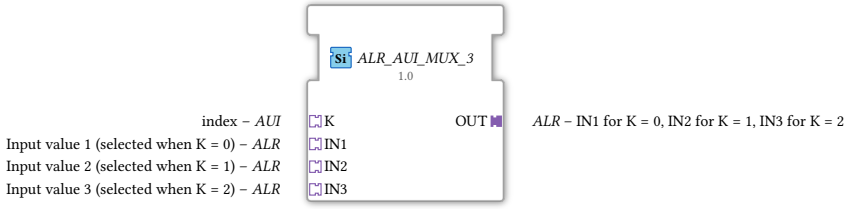
- **`ALR_AUI_MUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `ALR_AUI_MUX_2_UNGATED` überträgt die Multiplexer-Logik von `ALR_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter

aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALR_AUI_MUX_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ALR_AUI_MUX_3` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `LREAL`. Anders als `ALR_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ALR_AUI_MUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (ALR-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (ALR-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (ALR-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ALR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN3`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Eingangs-Adapter (`IN1` … `IN3`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ALR_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALR_AUI_MUX_2 … ALR_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (64-Bit-Fließkommazahl) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ALR_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

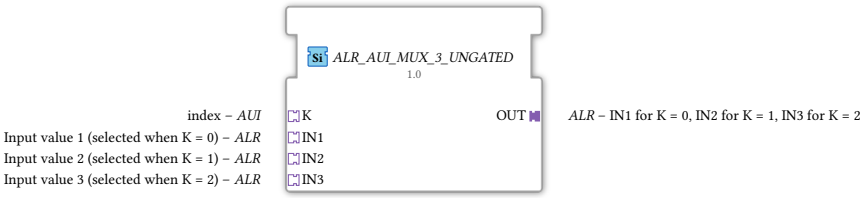
- **`ALR_AUI_MUX_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `ALR_AUI_MUX_3` überträgt die Multiplexer-Logik von `ALR_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALR_AUI_MUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALR_AUI_MUX_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der ALR_AUI_MUX_3_UNGATED ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp LREAL. Anders als ALR_MUX_3 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet ALR_AUI_MUX_3_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index K = 0 an OUT durchgereicht (ALR-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index K = 1 an OUT durchgereicht (ALR-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index K = 2 an OUT durchgereicht (ALR-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ALR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN3:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 3 Eingangs-Adapter (IN1 … IN3) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf OUT gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_ALR_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALR_AUI_MUX_2 … ALR_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (64-Bit-Fließkommazahl) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ALR_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`ALR_AUI_MUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `ALR_AUI_MUX_3_UNGATED` überträgt die Multiplexer-Logik von `ALR_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALR_AUI_MUX_4

• – ○ • – ○ • – ○ •

Einleitung

Der `ALR_AUI_MUX_4` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `LREAL`. Anders als `ALR_MUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ALR_AUI_MUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index `K = 0` an `OUT` durchgereicht (ALR-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index `K = 1` an `OUT` durchgereicht (ALR-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index `K = 2` an `OUT` durchgereicht (ALR-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index `K = 3` an `OUT` durchgereicht (ALR-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ALR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN4`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Eingangs-Adapter (`IN1` … `IN4`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ALR_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALR_AUI_MUX_2 … ALR_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (64-Bit-Fließkommazahl) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ALR_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

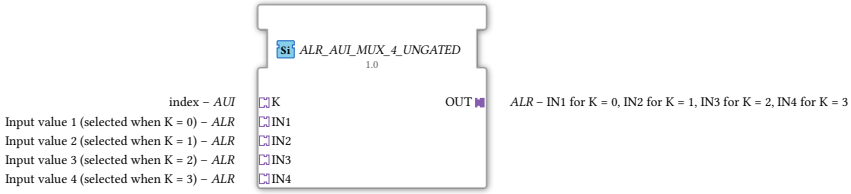
- **`ALR_AUI_MUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der ALR_AUI_MUX_4 überträgt die Multiplexer-Logik von ALR_MUX_4 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALR_AUI_MUX_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von ALR_AUI_MUX_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der ALR_AUI_MUX_4_UNGATED ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp LREAL. Anders als ALR_MUX_4 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet ALR_AUI_MUX_4_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index K = 0 an OUT durchgereicht (ALR-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index K = 1 an OUT durchgereicht (ALR-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index K = 2 an OUT durchgereicht (ALR-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index K = 3 an OUT durchgereicht (ALR-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ALR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`...`IN4`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Eingangs-Adapter (`IN1` ... `IN4`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ALR_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALR_AUI_MUX_2 ... ALR_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`...`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (64-Bit-Fließkommazahl) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ALR_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

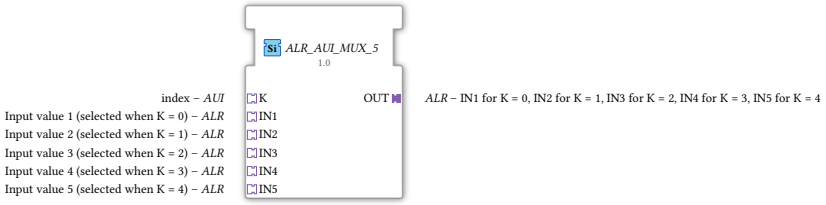
Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`ALR_AUI_MUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `ALR_AUI_MUX_4_UNGATED` überträgt die Multiplexer-Logik von `ALR_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALR_AUI_MUX_5



• — ○ • — ○ • — ○ •

Einleitung

Der ALR_AUI_MUX_5 ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp LREAL. Anders als ALR_MUX_5 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet ALR_AUI_MUX_5 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index K = 0 an OUT durchgereicht (ALR-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index K = 1 an OUT durchgereicht (ALR-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index K = 2 an OUT durchgereicht (ALR-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index K = 3 an OUT durchgereicht (ALR-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index K = 4 an OUT durchgereicht (ALR-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ALR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN5:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 5 Eingangs-Adapter (IN1 … IN5) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf OUT gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_ALR_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALR_AUI_MUX_2 … ALR_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (64-Bit-Fließkommazahl) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ALR_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

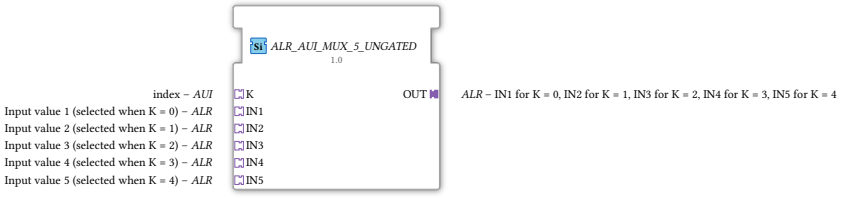
- **`ALR_AUI_MUX_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `ALR_AUI_MUX_5` überträgt die Multiplexer-Logik von `ALR_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ALR_AUI_MUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ALR_AUI_MUX_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `ALR_AUI_MUX_5_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `LREAL`. Anders als `ALR_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem

Datenwert D1 sendet, wertet ALR_AUI_MUX_5_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index K = 0 an OUT durchgereicht (ALR-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index K = 1 an OUT durchgereicht (ALR-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index K = 2 an OUT durchgereicht (ALR-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index K = 3 an OUT durchgereicht (ALR-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index K = 4 an OUT durchgereicht (ALR-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ALR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN5:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 5 Eingangs-Adapter (IN1 … IN5) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf OUT gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_ALR_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ALR_AUI_MUX_2 … ALR_AUI_MUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN1…IN5) → aktuell selektierten Eingang auslesen, OUT bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (64-Bit-Fließkommazahl) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ALR_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

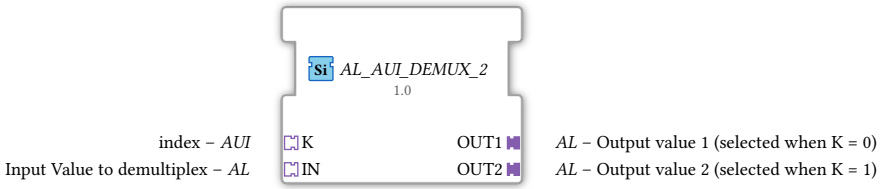
- **`ALR_AUI_MUX_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `ALR_AUI_MUX_5_UNGATED` überträgt die Multiplexer-Logik von `ALR_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

LWORD

AL_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AL_AUI_DEMUX_2` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `LWORD`. Anders als `AL_DEMUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AL_AUI_DEMUX_2` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`LWORD`, 64-Bit-Bitmuster), der an einen der `n` Ausgänge weitergeleitet wird (AL-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AL-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AL-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Ausgangs-Adapter (`OUT1` … `OUT2`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AL_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AL_AUI_DEMUX_2 … AL_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`LWORD`, 64-Bit-Bitmuster) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AL_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

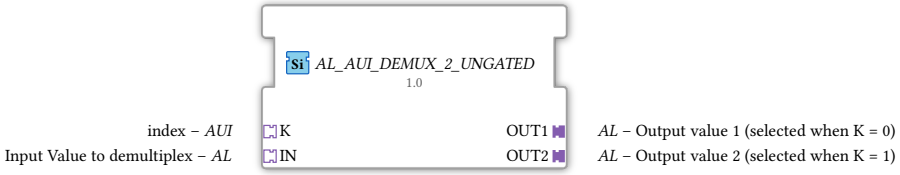
- **`AL_AUI_DEMUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AL_AUI_DEMUX_2` überträgt die Demultiplexer-Logik von `AL_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AL_AUI_DEMUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AL_AUI_DEMUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AL_AUI_DEMUX_2_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `LWORD`. Anders als `AL_DEMUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AL_AUI_DEMUX_2_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`LWORD`, 64-Bit-Bitmuster), der an einen der `n` Ausgänge weitergeleitet wird (AL-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AL-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AL-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Ausgangs-Adapter (`OUT1` … `OUT2`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AL_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AL_AUI_DEMUX_2_UNGATED … AL_AUI_DEMUX_5).

- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`LWORD`, 64-Bit-Bitmuster) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AL_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AL_AUI_DEMUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AL_AUI_DEMUX_2_UNGATED` überträgt die Demultiplexer-Logik von `AL_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AL_AUI_DEMUX_3

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AL_AUI_DEMUX_3` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `LWORD`. Anders als `AL_DEMUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AL_AUI_DEMUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`LWORD`, 64-Bit-Bitmuster), der an einen der `n` Ausgänge weitergeleitet wird (AL-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AL-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AL-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AL-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Ausgangs-Adapter (`OUT1` … `OUT3`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AL_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AL_AUI_DEMUX_2 … AL_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`LWORD`, 64-Bit-Bitmuster) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AL_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

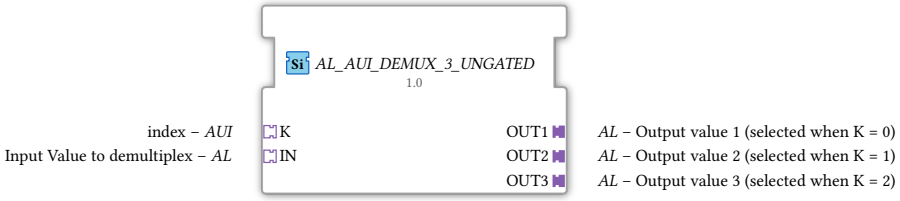
- **`AL_AUI_DEMUX_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der AL_AUI_DEMUX_3 überträgt die Demultiplexer-Logik von AL_DEMUX_3 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AL_AUI_DEMUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AL_AUI_DEMUX_3. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AL_AUI_DEMUX_3_UNGATED ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp LWORD. Anders als AL_DEMUX_3 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AL_AUI_DEMUX_3_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (LWORD, 64-Bit-Bitmuster), der an einen der n Ausgänge weitergeleitet wird (AL-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AL-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AL-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AL-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Ausgangs-Adapter (`OUT1` … `OUT3`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AL_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AL_AUI_DEMUX_2 … AL_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`LWORD`, 64-Bit-Bitmuster) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AL_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

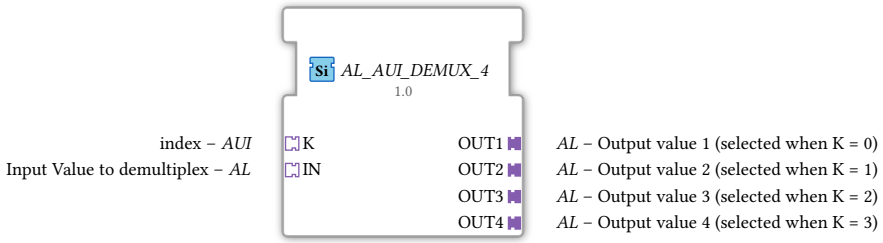
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AL_AUI_DEMUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AL_AUI_DEMUX_3_UNGATED` überträgt die Demultiplexer-Logik von `AL_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AL_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AL_AUI_DEMUX_4 ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp LWORD. Anders als AL_DEMUX_4 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AL_AUI_DEMUX_4 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (LWORD, 64-Bit-Bitmuster), der an einen der n Ausgänge weitergeleitet wird (AL-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AL-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AL-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AL-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AL-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 4 Ausgangs-Adapter (OUT1 … OUT4) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AL_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AL_AUI_DEMUX_2 … AL_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`LWORD`, 64-Bit-Bitmuster) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AL_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

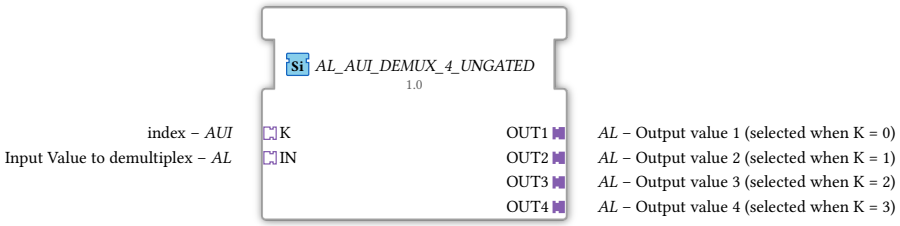
- **`AL_AUI_DEMUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AL_AUI_DEMUX_4` überträgt die Demultiplexer-Logik von `AL_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AL_AUI_DEMUX_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AL_AUI_DEMUX_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `AL_AUI_DEMUX_4_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `LWORD`. Anders als `AL_DEMUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AL_AUI_DEMUX_4_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`LWORD`, 64-Bit-Bitmuster), der an einen der `n` Ausgänge weitergeleitet wird (AL-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AL-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AL-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AL-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AL-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AL_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AL_AUI_DEMUX_2 … AL_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`LWORD`, 64-Bit-Bitmuster) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index

- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AL_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

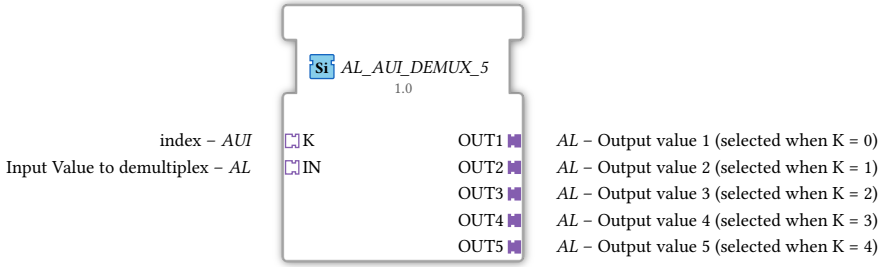
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AL_AUI_DEMUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AL_AUI_DEMUX_4_UNGATED` überträgt die Demultiplexer-Logik von `AL_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AL_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AL_AUI_DEMUX_5` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `LWORD`. Anders als `AL_DEMUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AL_AUI_DEMUX_5` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`LWORD`, 64-Bit-Bitmuster), der an einen der `n` Ausgänge weitergeleitet wird (AL-Adaptertyp).

- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AL-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AL-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AL-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AL-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AL-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Ausgangs-Adapter (`OUT1` … `OUT5`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AL_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AL_AUI_DEMUX_2 … AL_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`LWORD`, 64-Bit-Bitmuster) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AL_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

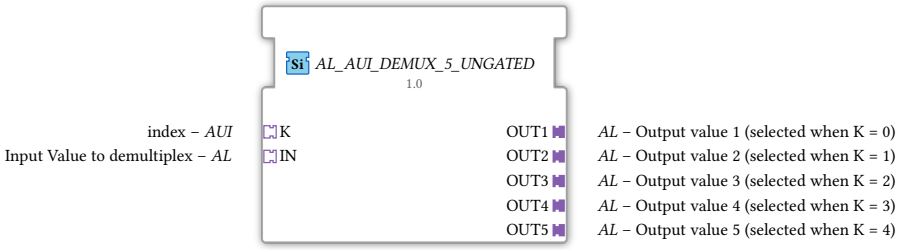
- **`AL_AUI_DEMUX_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AL_AUI_DEMUX_5` überträgt die Demultiplexer-Logik von `AL_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AL_AUI_DEMUX_5_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von AL_AUI_DEMUX_5. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AL_AUI_DEMUX_5_UNGATED ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp LWORD. Anders als AL_DEMUX_5 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AL_AUI_DEMUX_5_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (LWORD, 64-Bit-Bitmuster), der an einen der n Ausgänge weitergeleitet wird (AL-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AL-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AL-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AL-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AL-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AL-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Ausgangs-Adapter (`OUT1` … `OUT5`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AL_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AL_AUI_DEMUX_2 … AL_AUI_DEMUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`LWORD`, 64-Bit-Bitmuster) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AL_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

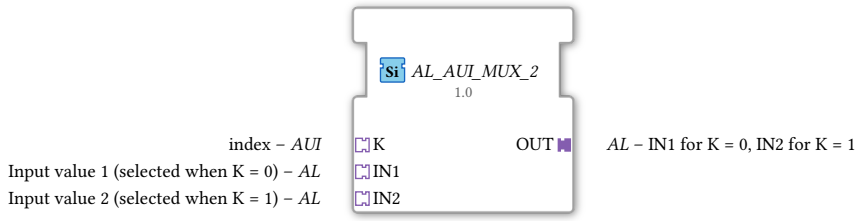
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AL_AUI_DEMUX_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AL_AUI_DEMUX_5_UNGATED` überträgt die Demultiplexer-Logik von `AL_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AL_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AL_AUI_MUX_2 ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp LWORD. Anders als AL_MUX_2 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AL_AUI_MUX_2 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index K = 0 an OUT durchgereicht (AL-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index K = 1 an OUT durchgereicht (AL-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AL-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN2:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 2 Eingangs-Adapter (IN1 … IN2) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf OUT gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AL_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AL_AUI_MUX_2 … AL_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN1…IN2) ⟶ aktuell selektierten Eingang auslesen, OUT bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (64-Bit-Bitmuster) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AL_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

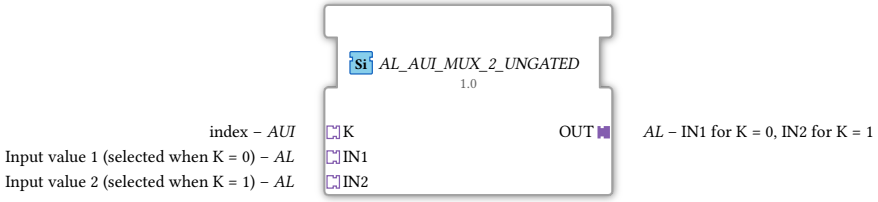
- **`AL_AUI_MUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AL_AUI_MUX_2` überträgt die Multiplexer-Logik von `AL_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AL_AUI_MUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AL_AUI_MUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AL_AUI_MUX_2_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `LWORD`. Anders als `AL_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AL_AUI_MUX_2_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (AL-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (AL-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AL-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AL_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AL_AUI_MUX_2_UNGATED … AL_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (64-Bit-Bitmuster) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AL_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AL_AUI_MUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AL_AUI_MUX_2_UNGATED` überträgt die Multiplexer-Logik von `AL_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AL_AUI_MUX_3

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AL_AUI_MUX_3` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `LWORD`. Anders als `AL_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AL_AUI_MUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (AL-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (AL-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (AL-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AL-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN3`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Eingangs-Adapter (`IN1` … `IN3`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AL_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AL_AUI_MUX_2 … AL_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN1…IN3) ⟶ aktuell selektierten Eingang auslesen, OUT bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (64-Bit-Bitmuster) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit AL_MUX_3 (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

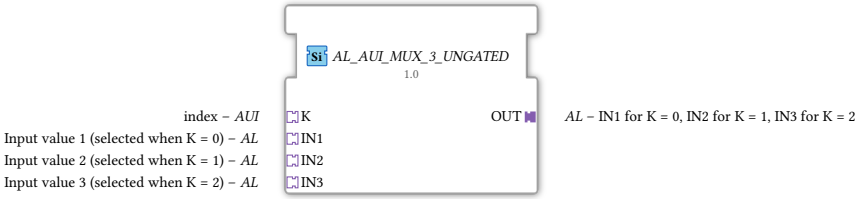
- **AL_AUI_MUX_3_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der AL_AUI_MUX_3 überträgt die Multiplexer-Logik von AL_MUX_3 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AL_AUI_MUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AL_AUI_MUX_3. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AL_AUI_MUX_3_UNGATED ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp LWORD. Anders als AL_MUX_3 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AL_AUI_MUX_3_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index K = 0 an OUT durchgereicht (AL-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index K = 1 an OUT durchgereicht (AL-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index K = 2 an OUT durchgereicht (AL-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AL-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN3:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 3 Eingangs-Adapter (IN1 … IN3) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf OUT gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AL_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AL_AUI_MUX_2 … AL_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (64-Bit-Bitmuster) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AL_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AL_AUI_MUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AL_AUI_MUX_3_UNGATED` überträgt die Multiplexer-Logik von `AL_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AL_AUI_MUX_4

• – ○ • – ○ • – ○ •

Einleitung

Der `AL_AUI_MUX_4` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `LWORD`. Anders als `AL_MUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AL_AUI_MUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index `K = 0` an `OUT` durchgereicht (AL-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index `K = 1` an `OUT` durchgereicht (AL-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index `K = 2` an `OUT` durchgereicht (AL-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index `K = 3` an `OUT` durchgereicht (AL-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AL-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN4`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Eingangs-Adapter (`IN1` … `IN4`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AL_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AL_AUI_MUX_2 … AL_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (64-Bit-Bitmuster) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AL_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

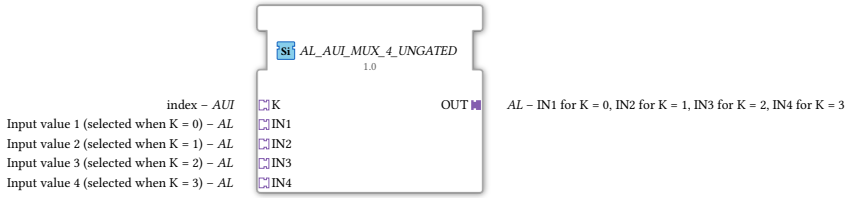
- **`AL_AUI_MUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AL_AUI_MUX_4` überträgt die Multiplexer-Logik von `AL_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AL_AUI_MUX_4_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von AL_AUI_MUX_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der AL_AUI_MUX_4_UNGATED ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp LWORD. Anders als AL_MUX_4 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AL_AUI_MUX_4_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index K = 0 an OUT durchgereicht (AL-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index K = 1 an OUT durchgereicht (AL-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index K = 2 an OUT durchgereicht (AL-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index K = 3 an OUT durchgereicht (AL-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AL-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN4:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 4 Eingangs-Adapter (IN1 … IN4) gerade ausgewählt ist.

2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AL_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AL_AUI_MUX_2 … AL_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (64-Bit-Bitmuster) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AL_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

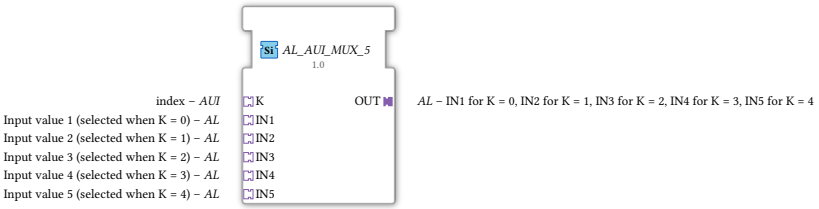
Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AL_AUI_MUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AL_AUI_MUX_4_UNGATED` überträgt die Multiplexer-Logik von `AL_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AL_AUI_MUX_5



• – ○ • – ○ • – ○ •

Einleitung

Der `AL_AUI_MUX_5` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `LWORD`. Anders als `AL_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“).

Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge
- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AL_AUI_MUX_5 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge
- Keine Ereignis-Ausgänge.

Daten-Eingänge
- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge
- Keine direkten Datenausgänge vorhanden.

Adapter
- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index K = 0 an OUT durchgereicht (AL-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index K = 1 an OUT durchgereicht (AL-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index K = 2 an OUT durchgereicht (AL-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index K = 3 an OUT durchgereicht (AL-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index K = 4 an OUT durchgereicht (AL-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AL-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN5:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 5 Eingangs-Adapter (IN1 … IN5) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf OUT gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten
- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AL_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AL_AUI_MUX_2 … AL_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN1…IN5) ⟶ aktuell selektierten Eingang auslesen, OUT bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (64-Bit-Bitmuster) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AL_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

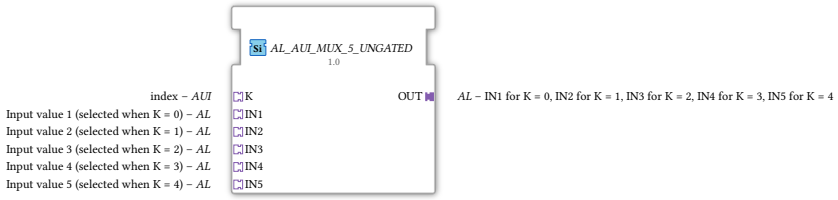
- **`AL_AUI_MUX_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AL_AUI_MUX_5` überträgt die Multiplexer-Logik von `AL_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AL_AUI_MUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AL_AUI_MUX_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `AL_AUI_MUX_5_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `LWORD`. Anders als `AL_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AL_AUI_MUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index `K = 0` an `OUT` durchgereicht (AL-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index `K = 1` an `OUT` durchgereicht (AL-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index `K = 2` an `OUT` durchgereicht (AL-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index `K = 3` an `OUT` durchgereicht (AL-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index `K = 4` an `OUT` durchgereicht (AL-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AL-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN5`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Eingangs-Adapter (`IN1` … `IN5`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AL_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AL_AUI_MUX_2 … AL_AUI_MUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (64-Bit-Bitmuster) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AL_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

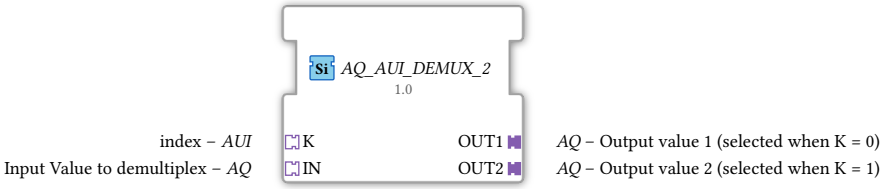
- `AL_AUI_MUX_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AL_AUI_MUX_5_UNGATED` überträgt die Multiplexer-Logik von `AL_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

QUARTER

AQ_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AQ_AUI_DEMUX_2` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `BYTE`. Anders als `AQ_DEMUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AQ_AUI_DEMUX_2` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`BYTE`, Byte-Wert (8 Bit Bitmuster)), der an einen der `n` Ausgänge weitergeleitet wird (AQ-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AQ-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AQ-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Ausgangs-Adapter (`OUT1` … `OUT2`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AQ_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AQ_AUI_DEMUX_2 … AQ_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Der `QUARTER`-Wert wird technisch als `BYTE` mit 2 gültigen Bits übertragen; die Interpretation als Viertelkreis-/Quadrantenwert erfolgt im aufrufenden Kontext.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`BYTE`, Byte-Wert (8 Bit Bitmuster)) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AQ_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

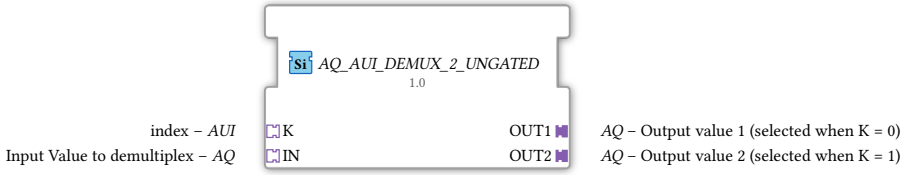
- **`AQ_AUI_DEMUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AQ_AUI_DEMUX_2` überträgt die Demultiplexer-Logik von `AQ_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AQ_AUI_DEMUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AQ_AUI_DEMUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AQ_AUI_DEMUX_2_UNGATED ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp BYTE. Anders als AQ_DEMUX_2 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AQ_AUI_DEMUX_2_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (BYTE, Byte-Wert (8 Bit Bitmuster)), der an einen der n Ausgänge weitergeleitet wird (AQ-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AQ-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AQ-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 2 Ausgangs-Adapter (OUT1 … OUT2) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AQ_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AQ_AUI_DEMUX_2_UNGATED … AQ_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Der QUARTER-Wert wird technisch als BYTE mit 2 gültigen Bits übertragen; die Interpretation als Viertelkreis-/Quadrantenwert erfolgt im aufrufenden Kontext.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`BYTE`, Byte-Wert (8 Bit Bitmuster)) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AQ_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

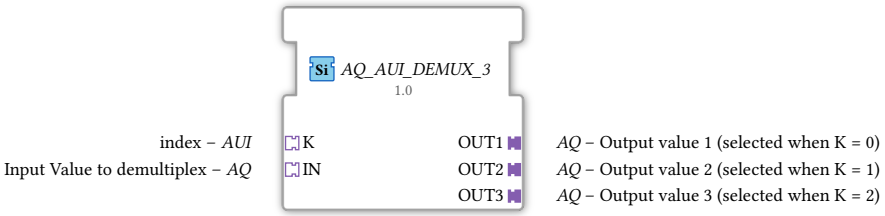
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AQ_AUI_DEMUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AQ_AUI_DEMUX_2_UNGATED` überträgt die Demultiplexer-Logik von `AQ_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AQ_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AQ_AUI_DEMUX_3` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `BYTE`. Anders als `AQ_DEMUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AQ_AUI_DEMUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`BYTE`, Byte-Wert (8 Bit Bitmuster)), der an einen der `n` Ausgänge weitergeleitet wird (AQ-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AQ-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AQ-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AQ-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Ausgangs-Adapter (`OUT1` … `OUT3`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AQ_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AQ_AUI_DEMUX_2 … AQ_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Der `QUARTER`-Wert wird technisch als `BYTE` mit 2 gültigen Bits übertragen; die Interpretation als Viertelkreis-/Quadrantenwert erfolgt im aufrufenden Kontext.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`BYTE`, Byte-Wert (8 Bit Bitmuster)) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AQ_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

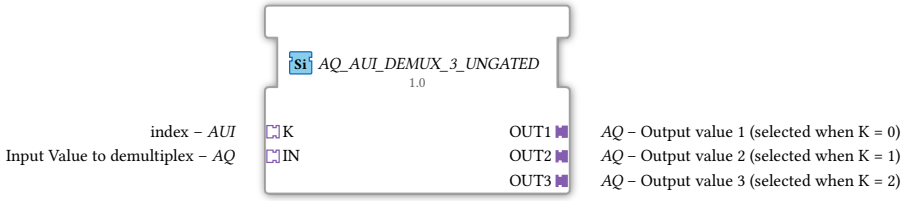
- **`AQ_AUI_DEMUX_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der AQ_AUI_DEMUX_3 überträgt die Demultiplexer-Logik von AQ_DEMUX_3 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AQ_AUI_DEMUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AQ_AUI_DEMUX_3. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AQ_AUI_DEMUX_3_UNGATED ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp BYTE. Anders als AQ_DEMUX_3 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AQ_AUI_DEMUX_3_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (BYTE, Byte-Wert (8 Bit Bitmuster)), der an einen der n Ausgänge weitergeleitet wird (AQ-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AQ-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AQ-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AQ-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Ausgangs-Adapter (`OUT1` … `OUT3`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AQ_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AQ_AUI_DEMUX_2 … AQ_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Der `QUARTER`-Wert wird technisch als `BYTE` mit 2 gültigen Bits übertragen; die Interpretation als Viertelkreis-/Quadrantenwert erfolgt im aufrufenden Kontext.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`BYTE`, Byte-Wert (8 Bit Bitmuster)) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AQ_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

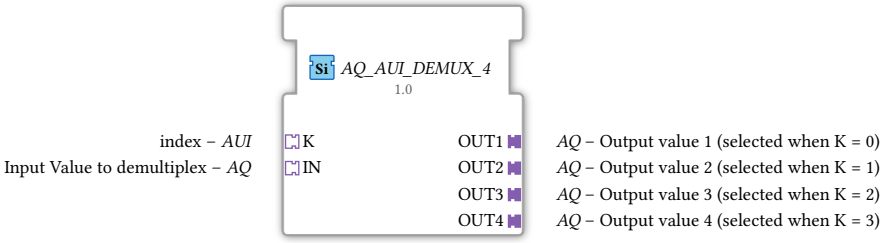
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AQ_AUI_DEMUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AQ_AUI_DEMUX_3_UNGATED` überträgt die Demultiplexer-Logik von `AQ_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AQ_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AQ_AUI_DEMUX_4` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `BYTE`. Anders als `AQ_DEMUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AQ_AUI_DEMUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`BYTE`, Byte-Wert (8 Bit Bitmuster)), der an einen der `n` Ausgänge weitergeleitet wird (AQ-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AQ-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AQ-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AQ-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AQ-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AQ_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AQ_AUI_DEMUX_2 … AQ_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Der `QUARTER`-Wert wird technisch als `BYTE` mit 2 gültigen Bits übertragen; die Interpretation als Viertelkreis-/Quadrantenwert erfolgt im aufrufenden Kontext.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`BYTE`, Byte-Wert (8 Bit Bitmuster)) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AQ_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

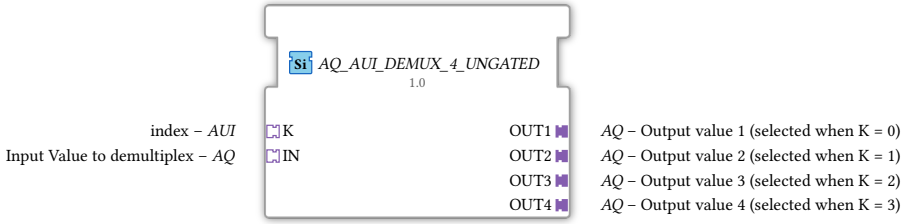
- **`AQ_AUI_DEMUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AQ_AUI_DEMUX_4` überträgt die Demultiplexer-Logik von `AQ_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AQ_AUI_DEMUX_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AQ_AUI_DEMUX_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AQ_AUI_DEMUX_4_UNGATED ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp BYTE. Anders als AQ_DEMUX_4 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AQ_AUI_DEMUX_4_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (BYTE, Byte-Wert (8 Bit Bitmuster)), der an einen der n Ausgänge weitergeleitet wird (AQ-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AQ-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AQ-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AQ-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AQ-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 4 Ausgangs-Adapter (OUT1 … OUT4) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AQ_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AQ_AUI_DEMUX_2 … AQ_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Der QUARTER-Wert wird technisch als BYTE mit 2 gültigen Bits übertragen; die Interpretation als Viertelkreis-/Quadrantenwert erfolgt im aufrufenden Kontext.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`BYTE`, Byte-Wert (8 Bit Bitmuster)) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AQ_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

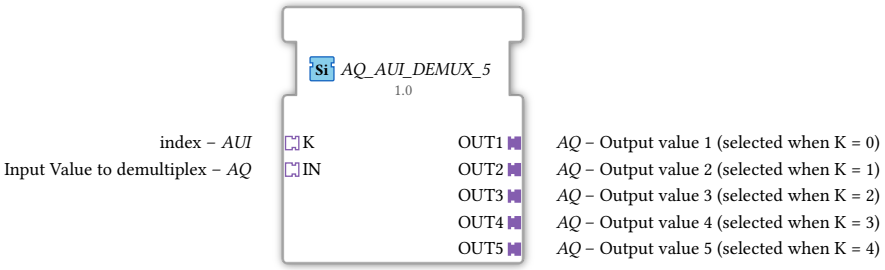
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AQ_AUI_DEMUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AQ_AUI_DEMUX_4_UNGATED` überträgt die Demultiplexer-Logik von `AQ_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AQ_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AQ_AUI_DEMUX_5` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `BYTE`. Anders als `AQ_DEMUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AQ_AUI_DEMUX_5` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`BYTE`, Byte-Wert (8 Bit Bitmuster)), der an einen der `n` Ausgänge weitergeleitet wird (AQ-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AQ-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AQ-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AQ-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AQ-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AQ-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Ausgangs-Adapter (`OUT1` … `OUT5`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AQ_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AQ_AUI_DEMUX_2 … AQ_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Der `QUARTER`-Wert wird technisch als `BYTE` mit 2 gültigen Bits übertragen; die Interpretation als Viertelkreis-/Quadrantenwert erfolgt im aufrufenden Kontext.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`BYTE`, Byte-Wert (8 Bit Bitmuster)) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AQ_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

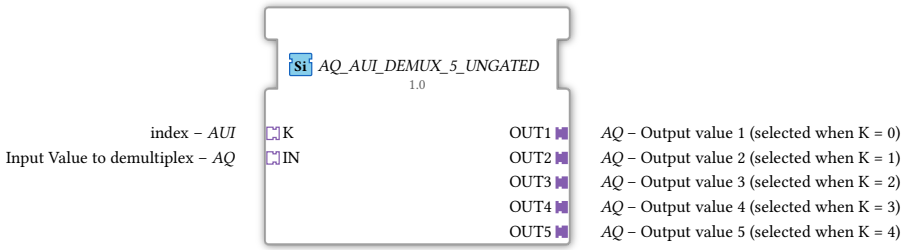
- **`AQ_AUI_DEMUX_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AQ_AUI_DEMUX_5` überträgt die Demultiplexer-Logik von `AQ_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AQ_AUI_DEMUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AQ_AUI_DEMUX_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `AQ_AUI_DEMUX_5_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `BYTE`. Anders als `AQ_DEMUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AQ_AUI_DEMUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`BYTE`, Byte-Wert (8 Bit Bitmuster)), der an einen der `n` Ausgänge weitergeleitet wird (AQ-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AQ-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AQ-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AQ-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AQ-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AQ-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Ausgangs-Adapter (`OUT1` … `OUT5`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AQ_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AQ_AUI_DEMUX_2 … AQ_AUI_DEMUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Der `QUARTER`-Wert wird technisch als `BYTE` mit 2 gültigen Bits übertragen; die Interpretation als Viertelkreis-/Quadrantenwert erfolgt im aufrufenden Kontext.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`BYTE`, Byte-Wert (8 Bit Bitmuster)) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AQ_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

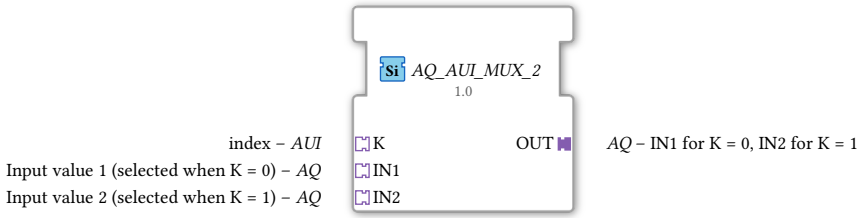
- **`AQ_AUI_DEMUX_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AQ_AUI_DEMUX_5_UNGATED` überträgt die Demultiplexer-Logik von `AQ_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als

AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AQ_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AQ_AUI_MUX_2 ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp BYTE. Anders als AQ_MUX_2 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AQ_AUI_MUX_2 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index K = 0 an OUT durchgereicht (AQ-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index K = 1 an OUT durchgereicht (AQ-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AQ-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN2:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 2 Eingangs-Adapter (IN1 … IN2) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf OUT gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AQ_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AQ_AUI_MUX_2 … AQ_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Der `QUARTER`-Wert wird technisch als `BYTE` mit 2 gültigen Bits übertragen; die Interpretation als Viertelkreis-/Quadrantenwert erfolgt im aufrufenden Kontext.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (Byte-Wert (8 Bit Bitmuster)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AQ_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

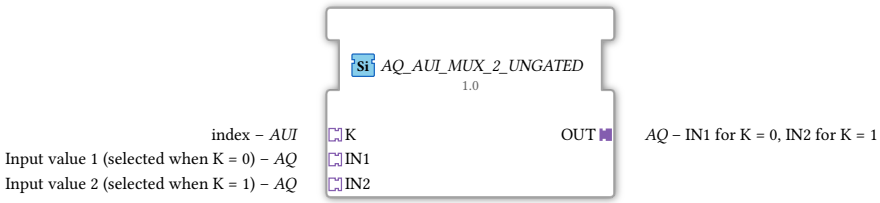
- **`AQ_AUI_MUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AQ_AUI_MUX_2` überträgt die Multiplexer-Logik von `AQ_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AQ_AUI_MUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AQ_AUI_MUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AQ_AUI_MUX_2_UNGATED ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp BYTE. Anders als AQ_MUX_2 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AQ_AUI_MUX_2_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index K = 0 an OUT durchgereicht (AQ-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index K = 1 an OUT durchgereicht (AQ-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AQ-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN2:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 2 Eingangs-Adapter (IN1 … IN2) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf OUT gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AQ_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AQ_AUI_MUX_2_UNGATED … AQ_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Der QUARTER-Wert wird technisch als BYTE mit 2 gültigen Bits übertragen; die Interpretation als Viertelkreis-/Quadrantenwert erfolgt im aufrufenden Kontext.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (Byte-Wert (8 Bit Bitmuster)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AQ_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

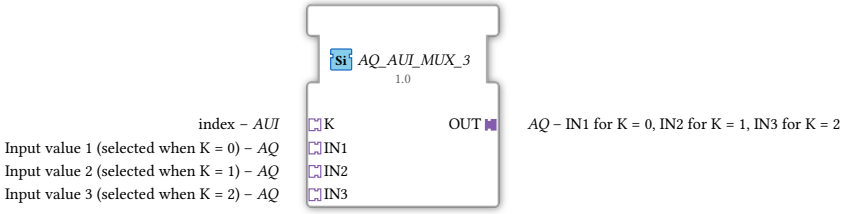
Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AQ_AUI_MUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AQ_AUI_MUX_2_UNGATED` überträgt die Multiplexer-Logik von `AQ_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AQ_AUI_MUX_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AQ_AUI_MUX_3` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `BYTE`. Anders als `AQ_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AQ_AUI_MUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (AQ-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (AQ-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (AQ-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AQ-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN3`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Eingangs-Adapter (`IN1` … `IN3`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AQ_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AQ_AUI_MUX_2 … AQ_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Der `QUARTER`-Wert wird technisch als `BYTE` mit 2 gültigen Bits übertragen; die Interpretation als Viertelkreis-/Quadrantenwert erfolgt im aufrufenden Kontext.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (Byte-Wert (8 Bit Bitmuster)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AQ_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AQ_AUI_MUX_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

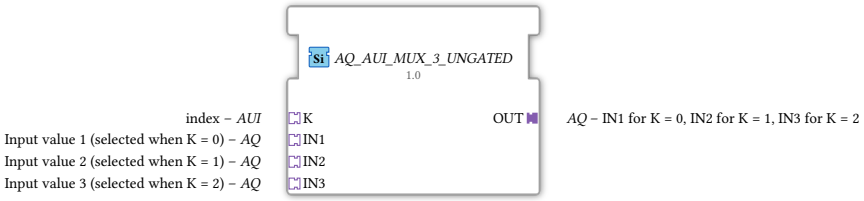
Fazit

Der `AQ_AUI_MUX_3` überträgt die Multiplexer-Logik von `AQ_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem

anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AQ_AUI_MUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AQ_AUI_MUX_3. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AQ_AUI_MUX_3_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `BYTE`. Anders als `AQ_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AQ_AUI_MUX_3_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (AQ-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (AQ-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (AQ-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AQ-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN3`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Eingangs-Adapter (`IN1` … `IN3`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AQ_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AQ_AUI_MUX_2 … AQ_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Der `QUARTER`-Wert wird technisch als `BYTE` mit 2 gültigen Bits übertragen; die Interpretation als Viertelkreis-/Quadrantenwert erfolgt im aufrufenden Kontext.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (Byte-Wert (8 Bit Bitmuster)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AQ_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AQ_AUI_MUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AQ_AUI_MUX_3_UNGATED` überträgt die Multiplexer-Logik von `AQ_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AQ_AUI_MUX_4

• – ○ • – ○ • – ○ •

Einleitung

Der AQ_AUI_MUX_4 ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp BYTE. Anders als AQ_MUX_4 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AQ_AUI_MUX_4 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index K = 0 an OUT durchgereicht (AQ-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index K = 1 an OUT durchgereicht (AQ-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index K = 2 an OUT durchgereicht (AQ-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index K = 3 an OUT durchgereicht (AQ-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AQ-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN4:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 4 Eingangs-Adapter (IN1 … IN4) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf OUT gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AQ_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AQ_AUI_MUX_2 … AQ_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Der QUARTER-Wert wird technisch als BYTE mit 2 gültigen Bits übertragen; die Interpretation als Viertelkreis-/Quadrantenwert erfolgt im aufrufenden Kontext.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (Byte-Wert (8 Bit Bitmuster)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AQ_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

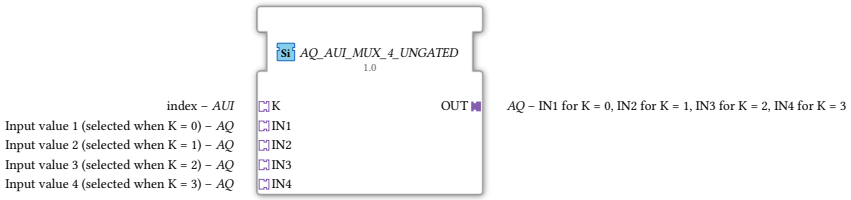
- **`AQ_AUI_MUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AQ_AUI_MUX_4` überträgt die Multiplexer-Logik von `AQ_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AQ_AUI_MUX_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AQ_AUI_MUX_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• — ○ • — ○ • — ○ •

Einleitung

Der `AQ_AUI_MUX_4_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `BYTE`. Anders als `AQ_MUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AQ_AUI_MUX_4_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index `K = 0` an `OUT` durchgereicht (AQ-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index `K = 1` an `OUT` durchgereicht (AQ-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index `K = 2` an `OUT` durchgereicht (AQ-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index `K = 3` an `OUT` durchgereicht (AQ-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AQ-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN4`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Eingangs-Adapter (`IN1` … `IN4`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AQ_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AQ_AUI_MUX_2 … AQ_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Der `QUARTER`-Wert wird technisch als `BYTE` mit 2 gültigen Bits übertragen; die Interpretation als Viertelkreis-/Quadrantenwert erfolgt im aufrufenden Kontext.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (Byte-Wert (8 Bit Bitmuster)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AQ_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

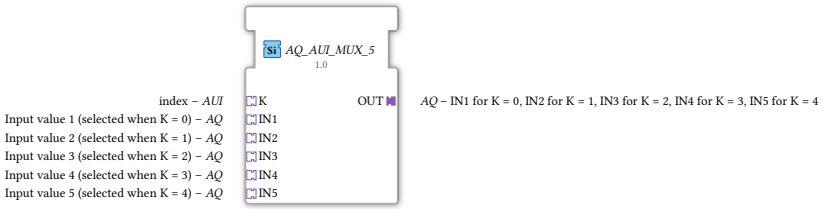
Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AQ_AUI_MUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AQ_AUI_MUX_4_UNGATED` überträgt die Multiplexer-Logik von `AQ_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AQ_AUI_MUX_5



• – ○ • – ○ • – ○ •

Einleitung

Der `AQ_AUI_MUX_5` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `BYTE`. Anders als `AQ_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AQ_AUI_MUX_5` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index `K = 0` an `OUT` durchgereicht (AQ-Adaptertyp).

- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index `K = 1` an `OUT` durchgereicht (AQ-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index `K = 2` an `OUT` durchgereicht (AQ-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index `K = 3` an `OUT` durchgereicht (AQ-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index `K = 4` an `OUT` durchgereicht (AQ-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AQ-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN5`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Eingangs-Adapter (`IN1` … `IN5`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AQ_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AQ_AUI_MUX_2 … AQ_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Der `QUARTER`-Wert wird technisch als `BYTE` mit 2 gültigen Bits übertragen; die Interpretation als Viertelkreis-/Quadrantenwert erfolgt im aufrufenden Kontext.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (Byte-Wert (8 Bit Bitmuster)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AQ_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

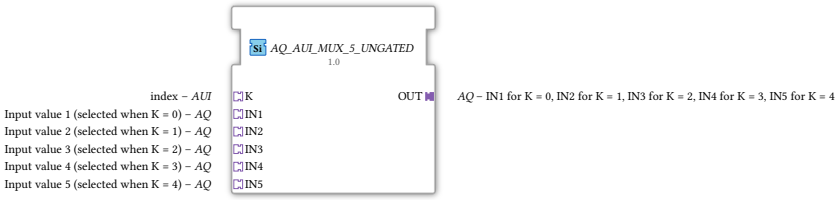
- **`AQ_AUI_MUX_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AQ_AUI_MUX_5` überträgt die Multiplexer-Logik von `AQ_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AQ_AUI_MUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AQ_AUI_MUX_5. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der AQ_AUI_MUX_5_UNGATED ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp BYTE. Anders als AQ_MUX_5 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AQ_AUI_MUX_5_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index K = 0 an OUT durchgereicht (AQ-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index K = 1 an OUT durchgereicht (AQ-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index K = 2 an OUT durchgereicht (AQ-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index K = 3 an OUT durchgereicht (AQ-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index K = 4 an OUT durchgereicht (AQ-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AQ-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN5:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 5 Eingangs-Adapter (IN1 … IN5) gerade ausgewählt ist.

2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AQ_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AQ_AUI_MUX_2 … AQ_AUI_MUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Der `QUARTER`-Wert wird technisch als `BYTE` mit 2 gültigen Bits übertragen; die Interpretation als Viertelkreis-/Quadrantenwert erfolgt im aufrufenden Kontext.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (Byte-Wert (8 Bit Bitmuster)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AQ_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

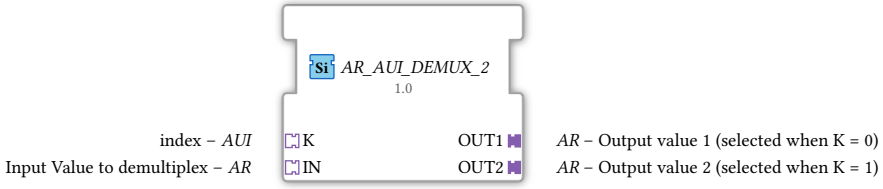
- `AQ_AUI_MUX_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AQ_AUI_MUX_5_UNGATED` überträgt die Multiplexer-Logik von `AQ_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

REAL

AR_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AR_AUI_DEMUX_2 ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp REAL. Anders als AR_DEMUX_2 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AR_AUI_DEMUX_2 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (REAL, 32-Bit-Fließkommazahl), der an einen der n Ausgänge weitergeleitet wird (AR-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AR-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 2 Ausgangs-Adapter (OUT1 … OUT2) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AR_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AR_AUI_DEMUX_2 … AR_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen IN-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`REAL`, 32-Bit-Fließkommazahl) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AR_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

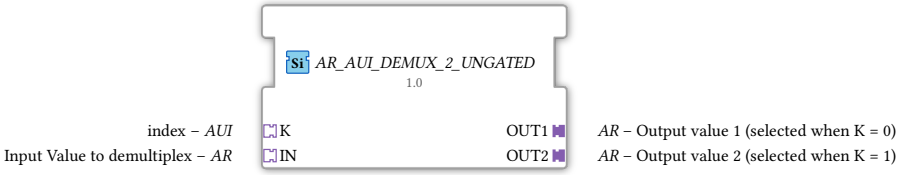
- **`AR_AUI_DEMUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AR_AUI_DEMUX_2` überträgt die Demultiplexer-Logik von `AR_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AR_AUI_DEMUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AR_AUI_DEMUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AR_AUI_DEMUX_2_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `REAL`. Anders als `AR_DEMUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AR_AUI_DEMUX_2_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`REAL`, 32-Bit-Fließkommazahl), der an einen der `n` Ausgänge weitergeleitet wird (AR-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AR-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Ausgangs-Adapter (`OUT1` … `OUT2`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AR_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AR_AUI_DEMUX_2_UNGATED … AR_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`REAL`, 32-Bit-Fließkommazahl) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AR_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

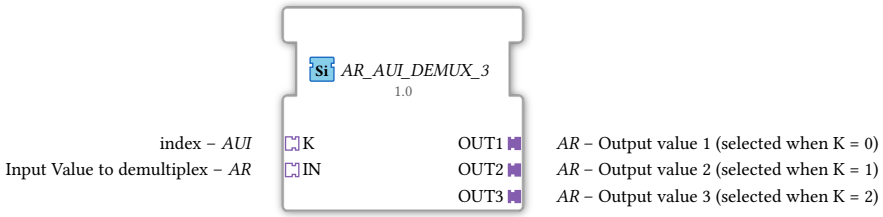
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AR_AUI_DEMUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AR_AUI_DEMUX_2_UNGATED` überträgt die Demultiplexer-Logik von `AR_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AR_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AR_AUI_DEMUX_3` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `REAL`. Anders als `AR_DEMUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AR_AUI_DEMUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`REAL`, 32-Bit-Fließkommazahl), der an einen der `n` Ausgänge weitergeleitet wird (AR-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AR-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AR-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Ausgangs-Adapter (`OUT1` … `OUT3`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AR_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AR_AUI_DEMUX_2 … AR_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`REAL`, 32-Bit-Fließkommazahl) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AR_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

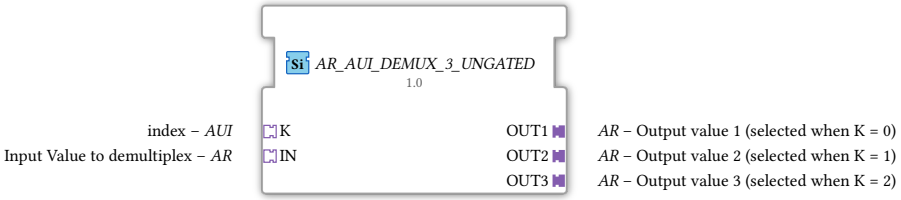
- **`AR_AUI_DEMUX_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AR_AUI_DEMUX_3` überträgt die Demultiplexer-Logik von `AR_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AR_AUI_DEMUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AR_AUI_DEMUX_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AR_AUI_DEMUX_3_UNGATED ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp REAL. Anders als AR_DEMUX_3 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AR_AUI_DEMUX_3_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (REAL, 32-Bit-Fließkommazahl), der an einen der n Ausgänge weitergeleitet wird (AR-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AR-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AR-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 3 Ausgangs-Adapter (OUT1 … OUT3) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AR_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AR_AUI_DEMUX_2 … AR_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`REAL`, 32-Bit-Fließkommazahl) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

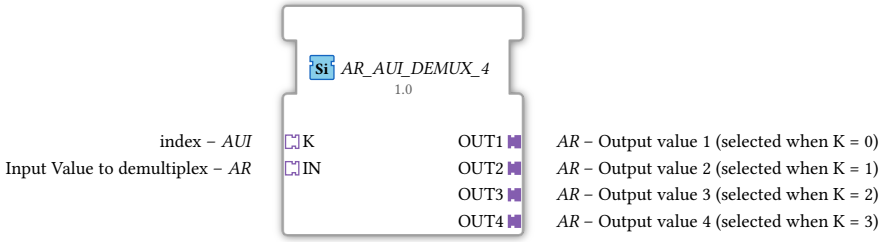
Vergleich mit `AR_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AR_AUI_DEMUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AR_AUI_DEMUX_3_UNGATED` überträgt die Demultiplexer-Logik von `AR_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AR_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AR_AUI_DEMUX_4` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `REAL`. Anders als `AR_DEMUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AR_AUI_DEMUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`REAL`, 32-Bit-Fließkommazahl), der an einen der `n` Ausgänge weitergeleitet wird (AR-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AR-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AR-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AR-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AR_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AR_AUI_DEMUX_2 … AR_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`REAL`, 32-Bit-Fließkommazahl) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AR_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

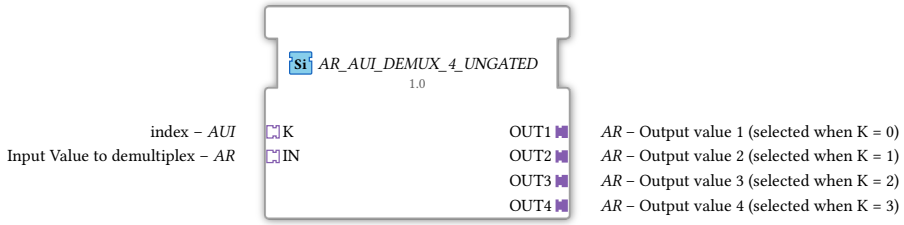
- **`AR_AUI_DEMUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AR_AUI_DEMUX_4` überträgt die Demultiplexer-Logik von `AR_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AR_AUI_DEMUX_4_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von AR_AUI_DEMUX_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AR_AUI_DEMUX_4_UNGATED ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp REAL. Anders als AR_DEMUX_4 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AR_AUI_DEMUX_4_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (REAL, 32-Bit-Fließkommazahl), der an einen der n Ausgänge weitergeleitet wird (AR-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AR-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AR-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AR-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AR_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AR_AUI_DEMUX_2 … AR_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`REAL`, 32-Bit-Fließkommazahl) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AR_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

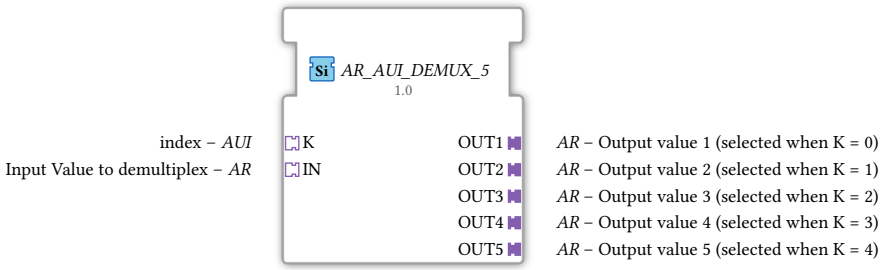
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AR_AUI_DEMUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AR_AUI_DEMUX_4_UNGATED` überträgt die Demultiplexer-Logik von `AR_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AR_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AR_AUI_DEMUX_5 ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp REAL. Anders als AR_DEMUX_5 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AR_AUI_DEMUX_5 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (REAL, 32-Bit-Fließkommazahl), der an einen der n Ausgänge weitergeleitet wird (AR-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AR-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AR-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AR-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AR-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 5 Ausgangs-Adapter (OUT1 … OUT5) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AR_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AR_AUI_DEMUX_2 … AR_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`REAL`, 32-Bit-Fließkommazahl) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AR_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

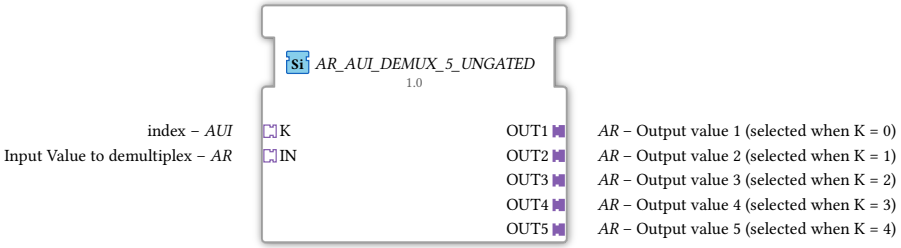
- **`AR_AUI_DEMUX_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AR_AUI_DEMUX_5` überträgt die Demultiplexer-Logik von `AR_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AR_AUI_DEMUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AR_AUI_DEMUX_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `AR_AUI_DEMUX_5_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `REAL`. Anders als `AR_DEMUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AR_AUI_DEMUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`REAL`, 32-Bit-Fließkommazahl), der an einen der `n` Ausgänge weitergeleitet wird (AR-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AR-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AR-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AR-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AR-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der `5` Ausgangs-Adapter (`OUT1` … `OUT5`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AR_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AR_AUI_DEMUX_2 … AR_AUI_DEMUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`REAL`, 32-Bit-Fließkommazahl) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AR_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

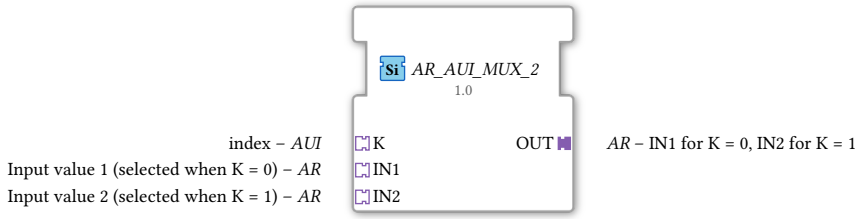
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AR_AUI_DEMUX_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AR_AUI_DEMUX_5_UNGATED` überträgt die Demultiplexer-Logik von `AR_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AR_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AR_AUI_MUX_2` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `REAL`. Anders als `AR_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AR_AUI_MUX_2` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.

- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (AR-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (AR-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AR_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AR_AUI_MUX_2 … AR_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (32-Bit-Fließkommazahl) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AR_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- `AR_AUI_MUX_2_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

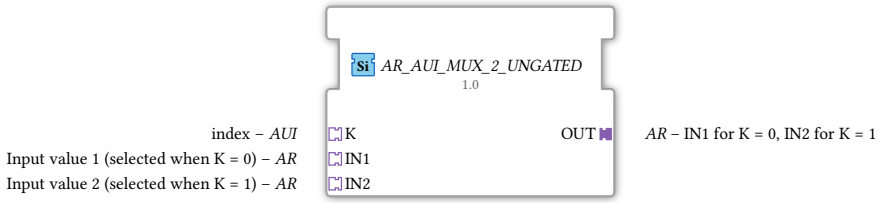
Fazit

Der `AR_AUI_MUX_2` überträgt die Multiplexer-Logik von `AR_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AR_AUI_MUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AR_AUI_MUX_2. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos

weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AR_AUI_MUX_2_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `REAL`. Anders als `AR_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AR_AUI_MUX_2_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (AR-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (AR-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AR_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AR_AUI_MUX_2_UNGATED … AR_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (32-Bit-Fließkommazahl) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AR_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AR_AUI_MUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AR_AUI_MUX_2_UNGATED` überträgt die Multiplexer-Logik von `AR_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AR_AUI_MUX_3

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AR_AUI_MUX_3` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `REAL`. Anders als `AR_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AR_AUI_MUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (AR-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (AR-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (AR-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN3`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Eingangs-Adapter (`IN1` … `IN3`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AR_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AR_AUI_MUX_2 … AR_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (32-Bit-Fließkommazahl) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird

- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit AR_MUX_3 (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

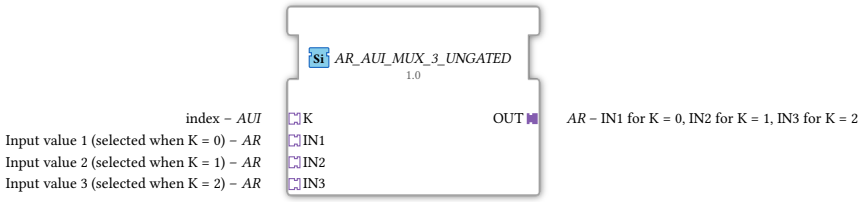
- **AR_AUI_MUX_3_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der AR_AUI_MUX_3 überträgt die Multiplexer-Logik von AR_MUX_3 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AR_AUI_MUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AR_AUI_MUX_3. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der AR_AUI_MUX_3_UNGATED ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp REAL. Anders als AR_MUX_3 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AR_AUI_MUX_3_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (AR-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (AR-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (AR-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN3`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Eingangs-Adapter (`IN1` … `IN3`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AR_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AR_AUI_MUX_2 … AR_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (32-Bit-Fließkommazahl) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AR_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AR_AUI_MUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AR_AUI_MUX_3_UNGATED` überträgt die Multiplexer-Logik von `AR_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AR_AUI_MUX_4

• – ○ • – ○ • – ○ •

Einleitung

Der `AR_AUI_MUX_4` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `REAL`. Anders als `AR_MUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AR_AUI_MUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index `K = 0` an `OUT` durchgereicht (AR-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index `K = 1` an `OUT` durchgereicht (AR-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index `K = 2` an `OUT` durchgereicht (AR-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index `K = 3` an `OUT` durchgereicht (AR-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN4`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Eingangs-Adapter (`IN1` … `IN4`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AR_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AR_AUI_MUX_2 … AR_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (32-Bit-Fließkommazahl) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AR_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

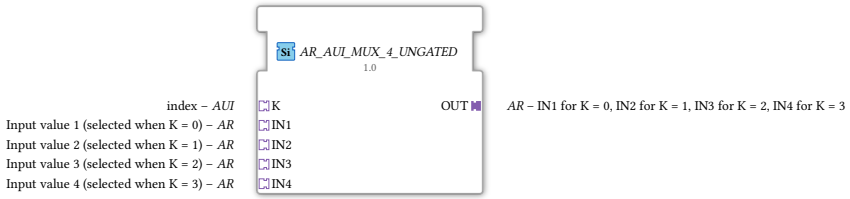
- **`AR_AUI_MUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AR_AUI_MUX_4` überträgt die Multiplexer-Logik von `AR_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AR_AUI_MUX_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AR_AUI_MUX_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der AR_AUI_MUX_4_UNGATED ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp REAL. Anders als AR_MUX_4 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AR_AUI_MUX_4_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index K = 0 an OUT durchgereicht (AR-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index K = 1 an OUT durchgereicht (AR-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index K = 2 an OUT durchgereicht (AR-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index K = 3 an OUT durchgereicht (AR-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN4:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 4 Eingangs-Adapter (IN1 … IN4) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf OUT gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AR_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AR_AUI_MUX_2 … AR_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (32-Bit-Fließkommazahl) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AR_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

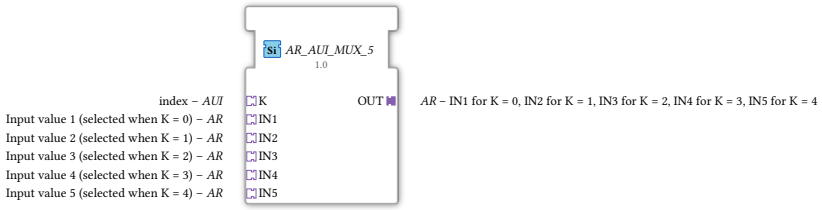
Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AR_AUI_MUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AR_AUI_MUX_4_UNGATED` überträgt die Multiplexer-Logik von `AR_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AR_AUI_MUX_5



• – ○ • – ○ • – ○ •

Einleitung

Der `AR_AUI_MUX_5` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `REAL`. Anders als `AR_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AR_AUI_MUX_5` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index `K = 0` an `OUT` durchgereicht (AR-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index `K = 1` an `OUT` durchgereicht (AR-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index `K = 2` an `OUT` durchgereicht (AR-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index `K = 3` an `OUT` durchgereicht (AR-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index `K = 4` an `OUT` durchgereicht (AR-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN5`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Eingangs-Adapter (`IN1` … `IN5`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AR_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AR_AUI_MUX_2 … AR_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (32-Bit-Fließkommazahl) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AR_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

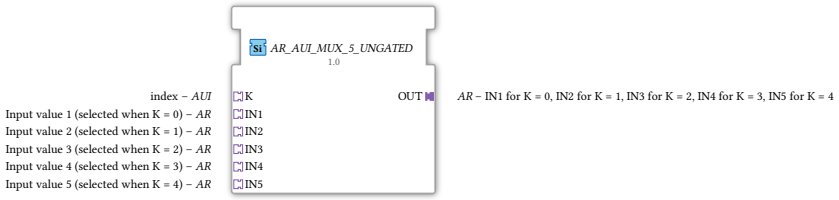
- **`AR_AUI_MUX_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AR_AUI_MUX_5` überträgt die Multiplexer-Logik von `AR_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AR_AUI_MUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AR_AUI_MUX_5. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `AR_AUI_MUX_5_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `REAL`. Anders als `AR_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AR_AUI_MUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index `K = 0` an `OUT` durchgereicht (AR-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index `K = 1` an `OUT` durchgereicht (AR-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index `K = 2` an `OUT` durchgereicht (AR-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index `K = 3` an `OUT` durchgereicht (AR-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index `K = 4` an `OUT` durchgereicht (AR-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AR-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN5`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Eingangs-Adapter (`IN1` … `IN5`) gerade ausgewählt ist.

2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AR_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AR_AUI_MUX_2 … AR_AUI_MUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (32-Bit-Fließkommazahl) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AR_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

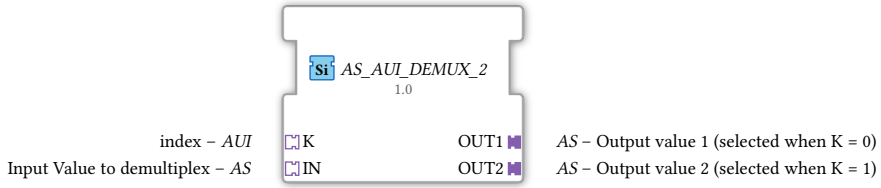
- `AR_AUI_MUX_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AR_AUI_MUX_5_UNGATED` überträgt die Multiplexer-Logik von `AR_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

SINT

AS_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AS_AUI_DEMUX_2` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `SINT`. Anders als `AS_DEMUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional

Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AS_AUI_DEMUX_2 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (SINT, 8-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der n Ausgänge weitergeleitet wird (AS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 2 Ausgangs-Adapter (OUT1 … OUT2) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AS_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AS_AUI_DEMUX_2 … AS_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen IN-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (SINT, 8-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index

- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AS_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

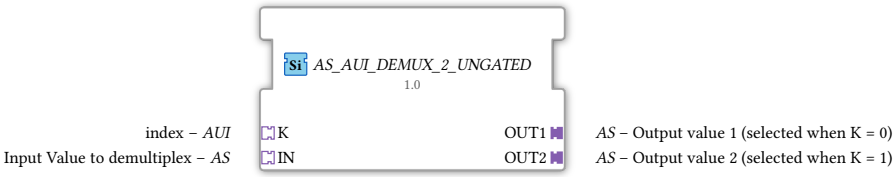
- **`AS_AUI_DEMUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AS_AUI_DEMUX_2` überträgt die Demultiplexer-Logik von `AS_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AS_AUI_DEMUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AS_AUI_DEMUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AS_AUI_DEMUX_2_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `SINT`. Anders als `AS_DEMUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AS_AUI_DEMUX_2_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`SINT`, 8-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der `n` Ausgänge weitergeleitet wird (AS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Ausgangs-Adapter (`OUT1` … `OUT2`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AS_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AS_AUI_DEMUX_2_UNGATED … AS_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`SINT`, 8-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AS_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

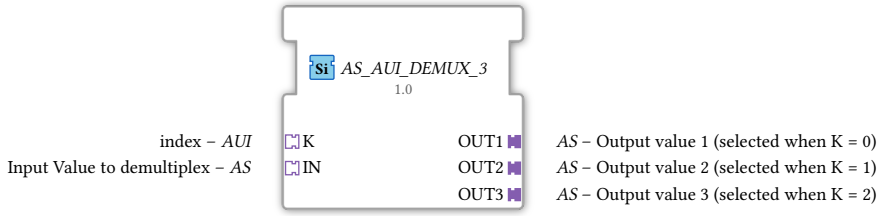
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AS_AUI_DEMUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AS_AUI_DEMUX_2_UNGATED` überträgt die Demultiplexer-Logik von `AS_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AS_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AS_AUI_DEMUX_3` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `SINT`. Anders als `AS_DEMUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AS_AUI_DEMUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`SINT`, 8-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der `n` Ausgänge weitergeleitet wird (AS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AS-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Ausgangs-Adapter (`OUT1` … `OUT3`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.

- Generische Implementierung (`GEN_AS_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AS_AUI_DEMUX_2 … AS_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`SINT`, 8-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AS_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

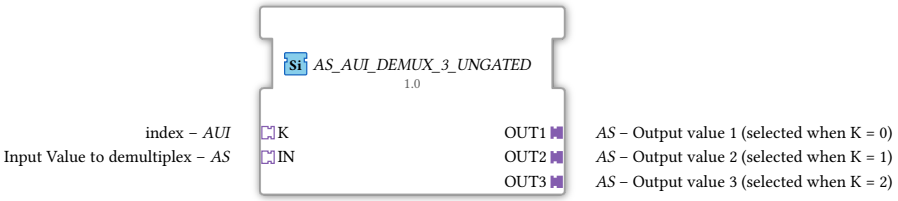
- **`AS_AUI_DEMUX_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AS_AUI_DEMUX_3` überträgt die Demultiplexer-Logik von `AS_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AS_AUI_DEMUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AS_AUI_DEMUX_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AS_AUI_DEMUX_3_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `SINT`. Anders als `AS_DEMUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter

Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge
- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AS_AUI_DEMUX_3_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge
- Keine Ereignis-Ausgänge.

Daten-Eingänge
- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge
- Keine direkten Datenausgänge vorhanden.

Adapter
- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (SINT, 8-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der n Ausgänge weitergeleitet wird (AS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AS-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 3 Ausgangs-Adapter (OUT1 … OUT3) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten
- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AS_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AS_AUI_DEMUX_2 … AS_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen IN-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`SINT`, 8-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AS_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

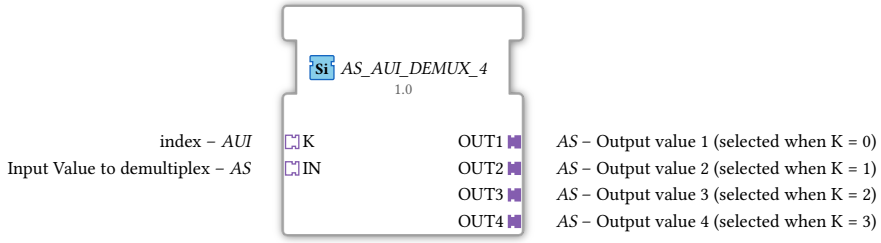
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AS_AUI_DEMUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AS_AUI_DEMUX_3_UNGATED` überträgt die Demultiplexer-Logik von `AS_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AS_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AS_AUI_DEMUX_4` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `SINT`. Anders als `AS_DEMUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AS_AUI_DEMUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`SINT`, 8-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der `n` Ausgänge weitergeleitet wird (AS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AS-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AS-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AS_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AS_AUI_DEMUX_2 … AS_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`SINT`, 8-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AS_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

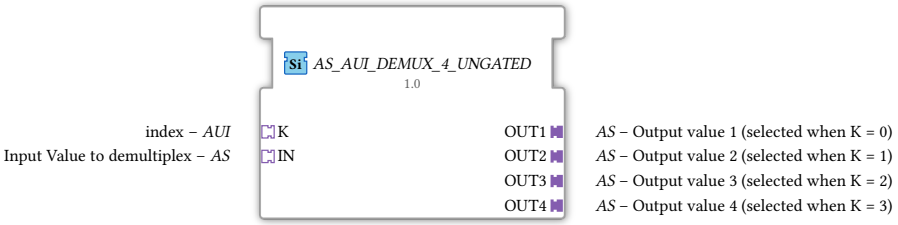
- **`AS_AUI_DEMUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AS_AUI_DEMUX_4` überträgt die Demultiplexer-Logik von `AS_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AS_AUI_DEMUX_4_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von AS_AUI_DEMUX_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AS_AUI_DEMUX_4_UNGATED ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp SINT. Anders als AS_DEMUX_4 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AS_AUI_DEMUX_4_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (SINT, 8-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der n Ausgänge weitergeleitet wird (AS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AS-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AS-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AS_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AS_AUI_DEMUX_2 … AS_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`SINT`, 8-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AS_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

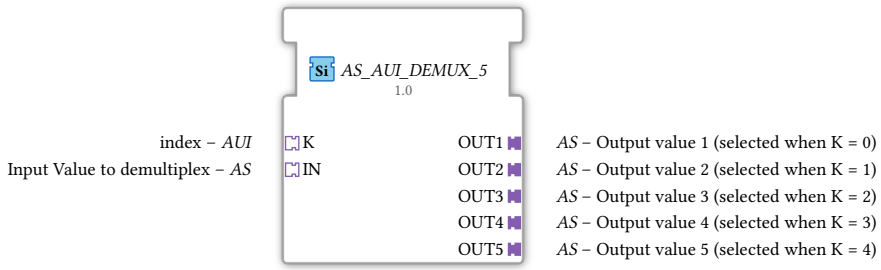
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AS_AUI_DEMUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AS_AUI_DEMUX_4_UNGATED` überträgt die Demultiplexer-Logik von `AS_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AS_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AS_AUI_DEMUX_5 ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp SINT. Anders als AS_DEMUX_5 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AS_AUI_DEMUX_5 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (SINT, 8-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der n Ausgänge weitergeleitet wird (AS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AS-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AS-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AS-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 5 Ausgangs-Adapter (OUT1 … OUT5) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AS_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AS_AUI_DEMUX_2 … AS_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`SINT`, 8-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AS_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

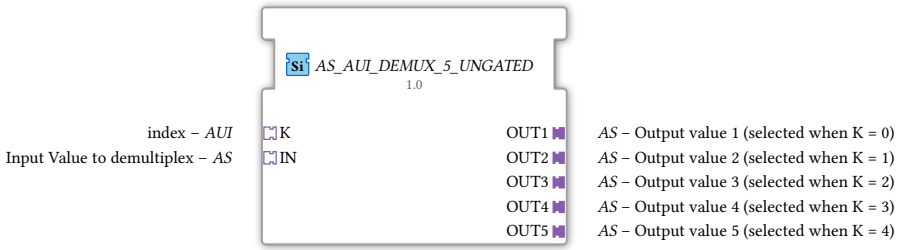
- **`AS_AUI_DEMUX_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AS_AUI_DEMUX_5` überträgt die Demultiplexer-Logik von `AS_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AS_AUI_DEMUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AS_AUI_DEMUX_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `AS_AUI_DEMUX_5_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `SINT`. Anders als `AS_DEMUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AS_AUI_DEMUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`SINT`, 8-Bit-Ganzzahl (vorzeichenbehaftet)), der an einen der `n` Ausgänge weitergeleitet wird (AS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AS-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AS-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AS-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der `5` Ausgangs-Adapter (`OUT1` … `OUT5`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AS_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AS_AUI_DEMUX_2 … AS_AUI_DEMUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`SINT`, 8-Bit-Ganzzahl (vorzeichenbehaftet)) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AS_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

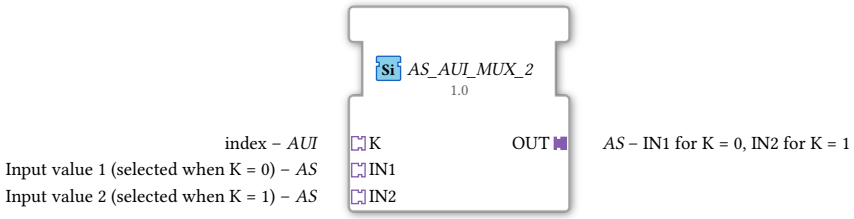
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AS_AUI_DEMUX_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AS_AUI_DEMUX_5_UNGATED` überträgt die Demultiplexer-Logik von `AS_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AS_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AS_AUI_MUX_2` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `SINT`. Anders als `AS_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AS_AUI_MUX_2` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.

- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (AS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (AS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AS_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AS_AUI_MUX_2 … AS_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (8-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AS_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AS_AUI_MUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

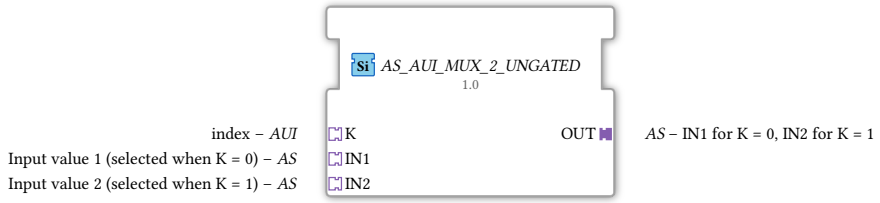
Fazit

Der `AS_AUI_MUX_2` überträgt die Multiplexer-Logik von `AS_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AS_AUI_MUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AS_AUI_MUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos

weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AS_AUI_MUX_2_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `SINT`. Anders als `AS_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AS_AUI_MUX_2_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (AS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (AS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AS_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AS_AUI_MUX_2_UNGATED … AS_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (8-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AS_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AS_AUI_MUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AS_AUI_MUX_2_UNGATED` überträgt die Multiplexer-Logik von `AS_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AS_AUI_MUX_3

• – ○ • – ○ • – ○ •

Einleitung

Der `AS_AUI_MUX_3` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `SINT`. Anders als `AS_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AS_AUI_MUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (AS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (AS-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (AS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN3`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Eingangs-Adapter (`IN1` … `IN3`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AS_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AS_AUI_MUX_2 … AS_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (8-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird

- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AS_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

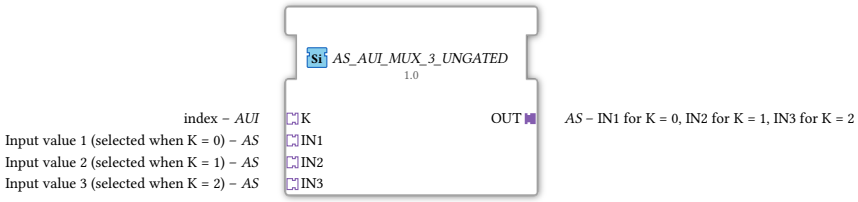
- **`AS_AUI_MUX_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AS_AUI_MUX_3` überträgt die Multiplexer-Logik von `AS_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AS_AUI_MUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AS_AUI_MUX_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `AS_AUI_MUX_3_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `SINT`. Anders als `AS_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AS_AUI_MUX_3_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (AS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (AS-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (AS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN3`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Eingangs-Adapter (`IN1` … `IN3`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AS_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AS_AUI_MUX_2 … AS_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (8-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AS_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

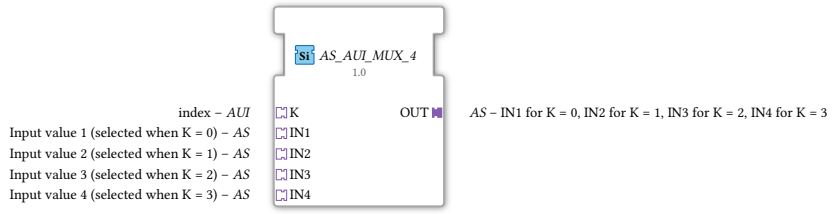
Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AS_AUI_MUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AS_AUI_MUX_3_UNGATED` überträgt die Multiplexer-Logik von `AS_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AS_AUI_MUX_4



• – ○ • – ○ • – ○ •

Einleitung

Der `AS_AUI_MUX_4` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `SINT`. Anders als `AS_MUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AS_AUI_MUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index `K = 0` an `OUT` durchgereicht (AS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index `K = 1` an `OUT` durchgereicht (AS-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index `K = 2` an `OUT` durchgereicht (AS-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index `K = 3` an `OUT` durchgereicht (AS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN4`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Eingangs-Adapter (`IN1` … `IN4`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AS_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AS_AUI_MUX_2 … AS_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (8-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AS_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

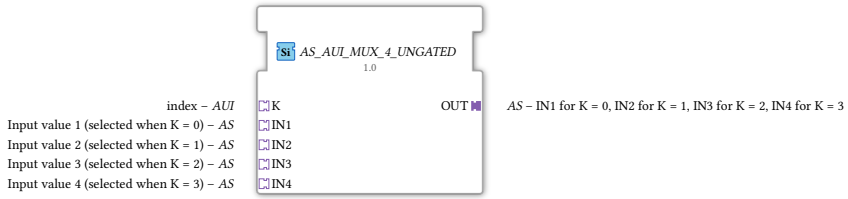
- **`AS_AUI_MUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AS_AUI_MUX_4` überträgt die Multiplexer-Logik von `AS_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AS_AUI_MUX_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AS_AUI_MUX_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der AS_AUI_MUX_4_UNGATED ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp SINT. Anders als AS_MUX_4 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AS_AUI_MUX_4_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index K = 0 an OUT durchgereicht (AS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index K = 1 an OUT durchgereicht (AS-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index K = 2 an OUT durchgereicht (AS-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index K = 3 an OUT durchgereicht (AS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN4:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 4 Eingangs-Adapter (IN1 … IN4) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf OUT gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AS_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AS_AUI_MUX_2 … AS_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (8-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AS_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

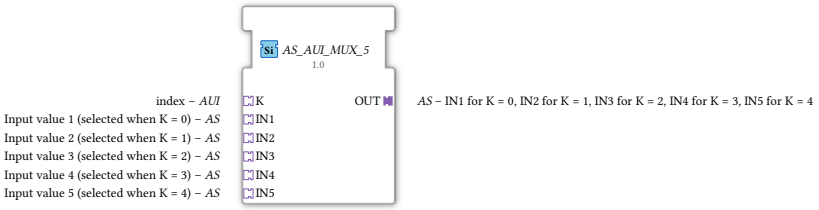
Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AS_AUI_MUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AS_AUI_MUX_4_UNGATED` überträgt die Multiplexer-Logik von `AS_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AS_AUI_MUX_5



• – ○ • – ○ • – ○ •

Einleitung

Der `AS_AUI_MUX_5` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `SINT`. Anders als `AS_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AS_AUI_MUX_5` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index `K = 0` an `OUT` durchgereicht (AS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index `K = 1` an `OUT` durchgereicht (AS-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index `K = 2` an `OUT` durchgereicht (AS-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index `K = 3` an `OUT` durchgereicht (AS-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index `K = 4` an `OUT` durchgereicht (AS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN5`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Eingangs-Adapter (`IN1` … `IN5`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AS_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AS_AUI_MUX_2 … AS_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (8-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AS_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

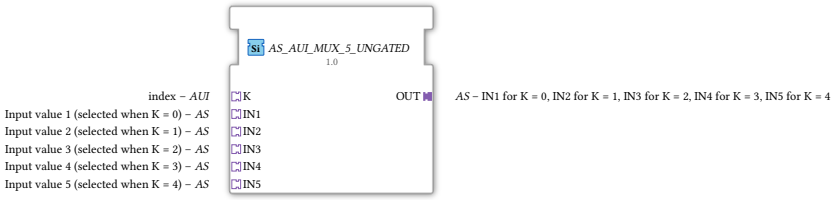
- `AS_AUI_MUX_5_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AS_AUI_MUX_5` überträgt die Multiplexer-Logik von `AS_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AS_AUI_MUX_5_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `AS_AUI_MUX_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `AS_AUI_MUX_5_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `SINT`. Anders als `AS_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AS_AUI_MUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index `K = 0` an `OUT` durchgereicht (AS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index `K = 1` an `OUT` durchgereicht (AS-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index `K = 2` an `OUT` durchgereicht (AS-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index `K = 3` an `OUT` durchgereicht (AS-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index `K = 4` an `OUT` durchgereicht (AS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN5`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Eingangs-Adapter (`IN1` … `IN5`) gerade ausgewählt ist.

2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AS_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AS_AUI_MUX_2 … AS_AUI_MUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (8-Bit-Ganzzahl (vorzeichenbehaftet)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AS_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

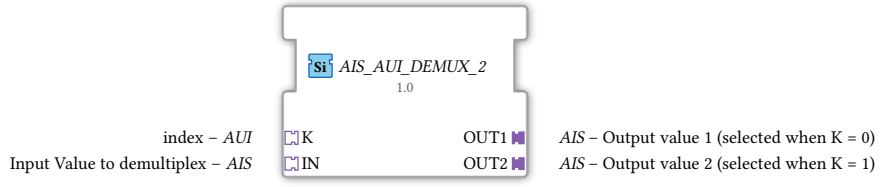
- `AS_AUI_MUX_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AS_AUI_MUX_5_UNGATED` überträgt die Multiplexer-Logik von `AS_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

STRING

AIS_AUI_DEMUX_2



• – ○ • – ○ • – ○ •

Einleitung

Der `AIS_AUI_DEMUX_2` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `STRING`. Anders als `AIS_DEMUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter

Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge
- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AIS_AUI_DEMUX_2 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge
- Keine Ereignis-Ausgänge.

Daten-Eingänge
- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge
- Keine direkten Datenausgänge vorhanden.

Adapter
- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (STRING, Zeichenkette (8-Bit-Zeichen)), der an einen der n Ausgänge weitergeleitet wird (AIS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AIS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AIS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 2 Ausgangs-Adapter (OUT1 … OUT2) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten
- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AIS_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIS_AUI_DEMUX_2 … AIS_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen IN-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien
- Verteilen eines Wertes (STRING, Zeichenkette (8-Bit-Zeichen)) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index

- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AIS_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

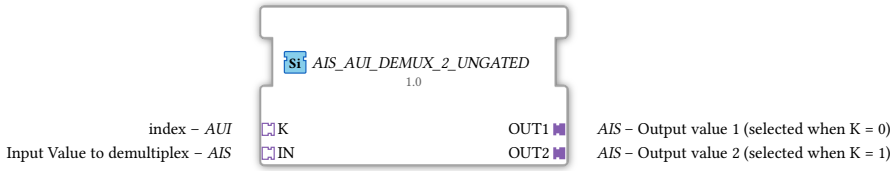
- **`AIS_AUI_DEMUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AIS_AUI_DEMUX_2` überträgt die Demultiplexer-Logik von `AIS_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIS_AUI_DEMUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AIS_AUI_DEMUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AIS_AUI_DEMUX_2_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `STRING`. Anders als `AIS_DEMUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AIS_AUI_DEMUX_2_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`STRING`, Zeichenkette (8-Bit-Zeichen)), der an einen der `n` Ausgänge weitergeleitet wird (AIS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AIS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AIS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Ausgangs-Adapter (`OUT1` … `OUT2`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AIS_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIS_AUI_DEMUX_2_UNGATED … AIS_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`STRING`, Zeichenkette (8-Bit-Zeichen)) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AIS_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

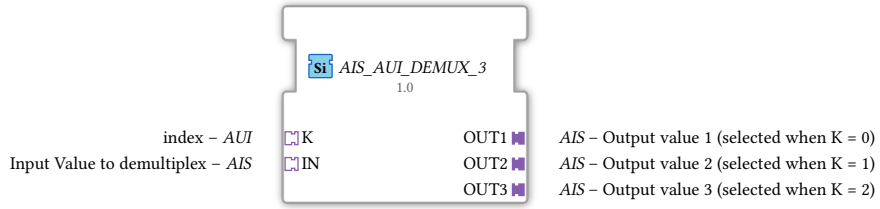
- **`AIS_AUI_DEMUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AIS_AUI_DEMUX_2_UNGATED` überträgt die Demultiplexer-Logik von `AIS_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als

AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIS_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AIS_AUI_DEMUX_3 ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp STRING. Anders als AIS_DEMUX_3 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AIS_AUI_DEMUX_3 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (STRING, Zeichenkette (8-Bit-Zeichen)), der an einen der n Ausgänge weitergeleitet wird (AIS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AIS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AIS-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AIS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 3 Ausgangs-Adapter (OUT1 … OUT3) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AIS_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIS_AUI_DEMUX_2 … AIS_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`STRING`, Zeichenkette (8-Bit-Zeichen)) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AIS_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

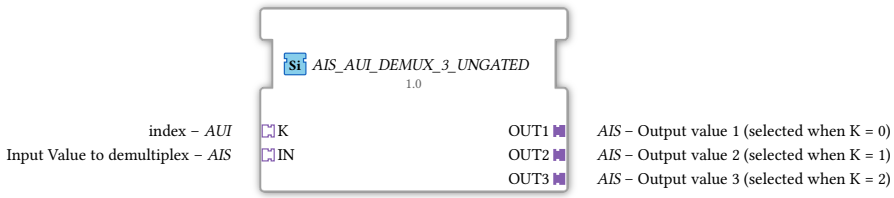
- **`AIS_AUI_DEMUX_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AIS_AUI_DEMUX_3` überträgt die Demultiplexer-Logik von `AIS_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIS_AUI_DEMUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AIS_AUI_DEMUX_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AIS_AUI_DEMUX_3_UNGATED ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp STRING. Anders als AIS_DEMUX_3 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AIS_AUI_DEMUX_3_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (STRING, Zeichenkette (8-Bit-Zeichen)), der an einen der n Ausgänge weitergeleitet wird (AIS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AIS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AIS-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AIS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 3 Ausgangs-Adapter (OUT1 … OUT3) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AIS_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIS_AUI_DEMUX_2 … AIS_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`STRING`, Zeichenkette (8-Bit-Zeichen)) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

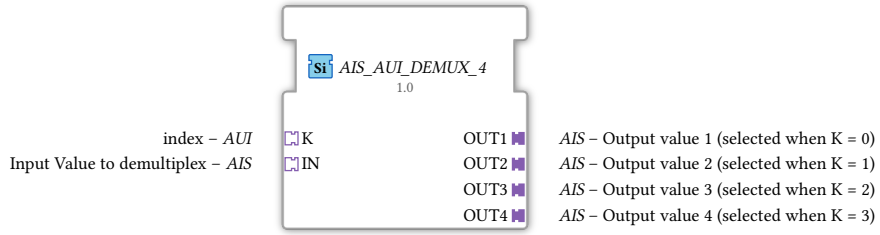
Vergleich mit `AIS_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AIS_AUI_DEMUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AIS_AUI_DEMUX_3_UNGATED` überträgt die Demultiplexer-Logik von `AIS_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIS_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AIS_AUI_DEMUX_4` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `STRING`. Anders als `AIS_DEMUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AIS_AUI_DEMUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`STRING`, Zeichenkette (8-Bit-Zeichen)), der an einen der `n` Ausgänge weitergeleitet wird (AIS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AIS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AIS-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AIS-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AIS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AIS_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIS_AUI_DEMUX_2 … AIS_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`STRING`, Zeichenkette (8-Bit-Zeichen)) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AIS_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

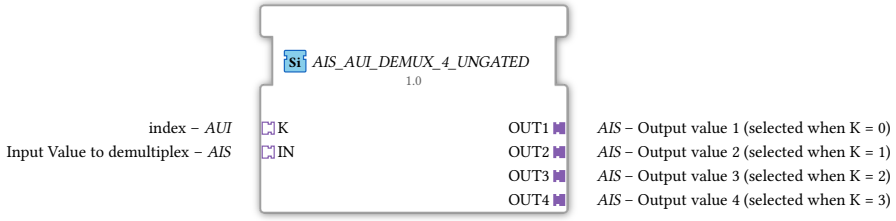
- **`AIS_AUI_DEMUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AIS_AUI_DEMUX_4` überträgt die Demultiplexer-Logik von `AIS_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIS_AUI_DEMUX_4_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `AIS_AUI_DEMUX_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AIS_AUI_DEMUX_4_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `STRING`. Anders als `AIS_DEMUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AIS_AUI_DEMUX_4_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`STRING`, Zeichenkette (8-Bit-Zeichen)), der an einen der `n` Ausgänge weitergeleitet wird (AIS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AIS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AIS-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AIS-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AIS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AIS_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIS_AUI_DEMUX_2 … AIS_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`STRING`, Zeichenkette (8-Bit-Zeichen)) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AIS_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

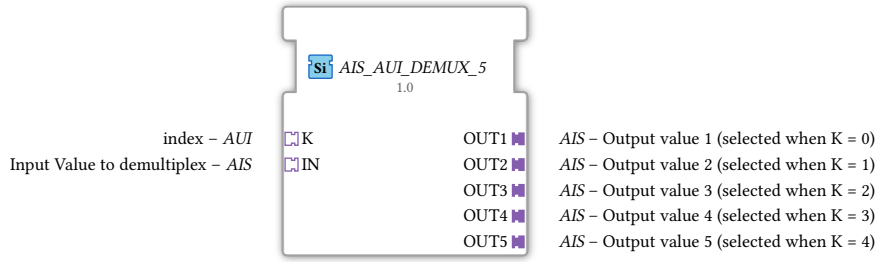
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AIS_AUI_DEMUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AIS_AUI_DEMUX_4_UNGATED` überträgt die Demultiplexer-Logik von `AIS_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIS_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AIS_AUI_DEMUX_5 ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp STRING. Anders als AIS_DEMUX_5 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AIS_AUI_DEMUX_5 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (STRING, Zeichenkette (8-Bit-Zeichen)), der an einen der n Ausgänge weitergeleitet wird (AIS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AIS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AIS-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AIS-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AIS-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AIS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 5 Ausgangs-Adapter (OUT1 … OUT5) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AIS_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIS_AUI_DEMUX_2 … AIS_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`STRING`, Zeichenkette (8-Bit-Zeichen)) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AIS_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

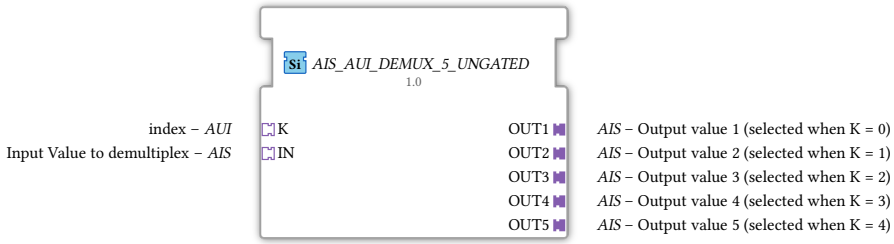
- **`AIS_AUI_DEMUX_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AIS_AUI_DEMUX_5` überträgt die Demultiplexer-Logik von `AIS_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIS_AUI_DEMUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AIS_AUI_DEMUX_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AIS_AUI_DEMUX_5_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `STRING`. Anders als `AIS_DEMUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AIS_AUI_DEMUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`STRING`, Zeichenkette (8-Bit-Zeichen)), der an einen der `n` Ausgänge weitergeleitet wird (AIS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AIS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AIS-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AIS-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AIS-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AIS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der `5` Ausgangs-Adapter (`OUT1` … `OUT5`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AIS_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIS_AUI_DEMUX_2 … AIS_AUI_DEMUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`STRING`, Zeichenkette (8-Bit-Zeichen)) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AIS_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

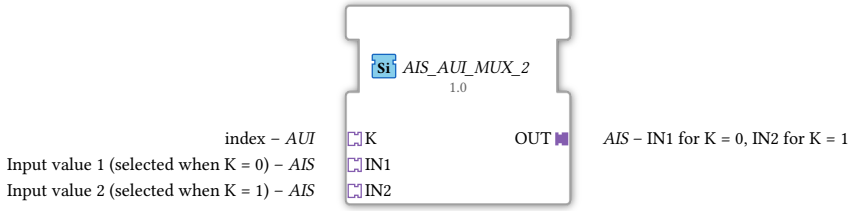
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AIS_AUI_DEMUX_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AIS_AUI_DEMUX_5_UNGATED` überträgt die Demultiplexer-Logik von `AIS_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIS_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AIS_AUI_MUX_2` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `STRING`. Anders als `AIS_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AIS_AUI_MUX_2` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (AIS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (AIS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AIS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AIS_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIS_AUI_MUX_2 … AIS_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (Zeichenkette (8-Bit-Zeichen)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AIS_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

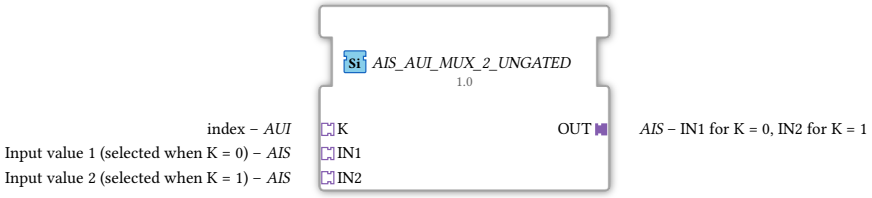
- **`AIS_AUI_MUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AIS_AUI_MUX_2` überträgt die Multiplexer-Logik von `AIS_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIS_AUI_MUX_2_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von AIS_AUI_MUX_2. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AIS_AUI_MUX_2_UNGATED ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp STRING. Anders als AIS_MUX_2 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AIS_AUI_MUX_2_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index K = 0 an OUT durchgereicht (AIS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index K = 1 an OUT durchgereicht (AIS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AIS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN2:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 2 Eingangs-Adapter (IN1 … IN2) gerade ausgewählt ist.

2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AIS_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIS_AUI_MUX_2_UNGATED … AIS_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (Zeichenkette (8-Bit-Zeichen)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AIS_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

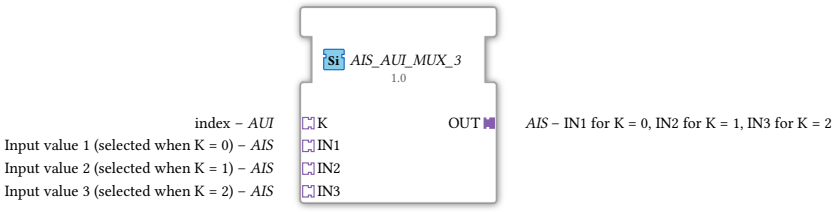
Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AIS_AUI_MUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AIS_AUI_MUX_2_UNGATED` überträgt die Multiplexer-Logik von `AIS_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIS_AUI_MUX_3



• – ○ • – ○ • – ○ •

Einleitung

Der `AIS_AUI_MUX_3` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `STRING`. Anders als `AIS_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional

Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AIS_AUI_MUX_3 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index K = 0 an OUT durchgereicht (AIS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index K = 1 an OUT durchgereicht (AIS-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index K = 2 an OUT durchgereicht (AIS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AIS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN3:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 3 Eingangs-Adapter (IN1 … IN3) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf OUT gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AIS_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIS_AUI_MUX_2 … AIS_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN1…IN3) ⟶ aktuell selektierten Eingang auslesen, OUT bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (Zeichenkette (8-Bit-Zeichen)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AIS_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

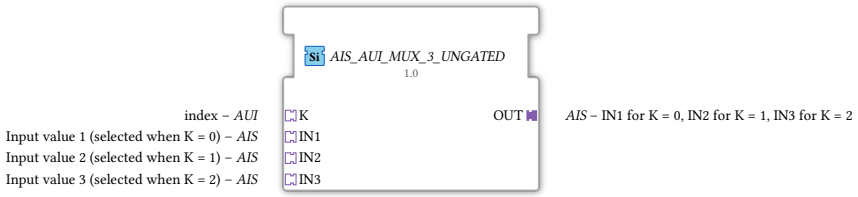
- **`AIS_AUI_MUX_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AIS_AUI_MUX_3` überträgt die Multiplexer-Logik von `AIS_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIS_AUI_MUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AIS_AUI_MUX_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `AIS_AUI_MUX_3_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `STRING`. Anders als `AIS_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AIS_AUI_MUX_3_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (AIS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (AIS-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (AIS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AIS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN3`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Eingangs-Adapter (`IN1` … `IN3`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AIS_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIS_AUI_MUX_2 … AIS_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (Zeichenkette (8-Bit-Zeichen)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AIS_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AIS_AUI_MUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AIS_AUI_MUX_3_UNGATED` überträgt die Multiplexer-Logik von `AIS_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIS_AUI_MUX_4

• — ○ • — ○ • — ○ •

Einleitung

Der `AIS_AUI_MUX_4` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `STRING`. Anders als `AIS_MUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AIS_AUI_MUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index `K = 0` an `OUT` durchgereicht (AIS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index `K = 1` an `OUT` durchgereicht (AIS-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index `K = 2` an `OUT` durchgereicht (AIS-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index `K = 3` an `OUT` durchgereicht (AIS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AIS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN4`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Eingangs-Adapter (`IN1` … `IN4`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AIS_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIS_AUI_MUX_2 … AIS_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN1…IN4) ⟶ aktuell selektierten Eingang auslesen, OUT bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (Zeichenkette (8-Bit-Zeichen)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit AIS_MUX_4 (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **AIS_AUI_MUX_4_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der AIS_AUI_MUX_4 überträgt die Multiplexer-Logik von AIS_MUX_4 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIS_AUI_MUX_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AIS_AUI_MUX_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• — ○ • — ○ • — ○ •

Einleitung

Der AIS_AUI_MUX_4_UNGATED ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp STRING. Anders als AIS_MUX_4 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AIS_AUI_MUX_4_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index K = 0 an OUT durchgereicht (AIS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index K = 1 an OUT durchgereicht (AIS-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index K = 2 an OUT durchgereicht (AIS-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index K = 3 an OUT durchgereicht (AIS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AIS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN4:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 4 Eingangs-Adapter (IN1 … IN4) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf OUT gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AIS_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIS_AUI_MUX_2 … AIS_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (Zeichenkette (8-Bit-Zeichen)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AIS_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

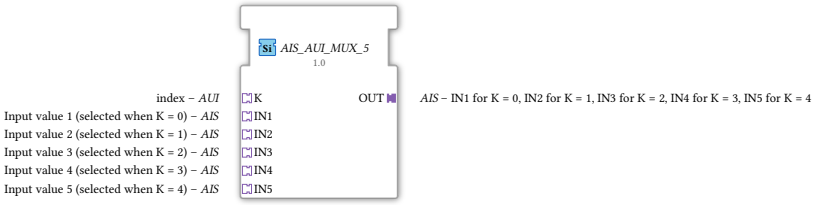
Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AIS_AUI_MUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AIS_AUI_MUX_4_UNGATED` überträgt die Multiplexer-Logik von `AIS_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIS_AUI_MUX_5



• – ○ • – ○ • – ○ •

Einleitung

Der `AIS_AUI_MUX_5` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `STRING`. Anders als `AIS_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AIS_AUI_MUX_5` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index `K = 0` an `OUT` durchgereicht (AIS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index `K = 1` an `OUT` durchgereicht (AIS-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index `K = 2` an `OUT` durchgereicht (AIS-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index `K = 3` an `OUT` durchgereicht (AIS-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index `K = 4` an `OUT` durchgereicht (AIS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AIS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN5`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Eingangs-Adapter (`IN1` … `IN5`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AIS_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIS_AUI_MUX_2 … AIS_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (Zeichenkette (8-Bit-Zeichen)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AIS_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

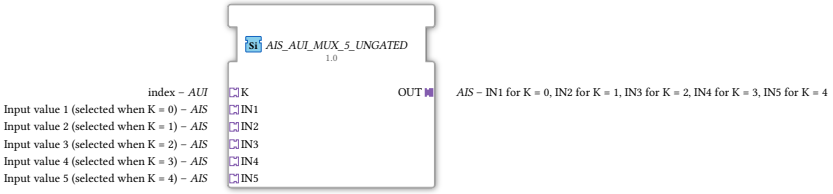
- **`AIS_AUI_MUX_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AIS_AUI_MUX_5` überträgt die Multiplexer-Logik von `AIS_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIS_AUI_MUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AIS_AUI_MUX_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `AIS_AUI_MUX_5_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `STRING`. Anders als `AIS_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AIS_AUI_MUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index `K = 0` an `OUT` durchgereicht (AIS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index `K = 1` an `OUT` durchgereicht (AIS-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index `K = 2` an `OUT` durchgereicht (AIS-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index `K = 3` an `OUT` durchgereicht (AIS-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index `K = 4` an `OUT` durchgereicht (AIS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AIS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN5`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Eingangs-Adapter (`IN1` … `IN5`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AIS_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIS_AUI_MUX_2 … AIS_AUI_MUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (Zeichenkette (8-Bit-Zeichen)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AIS_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

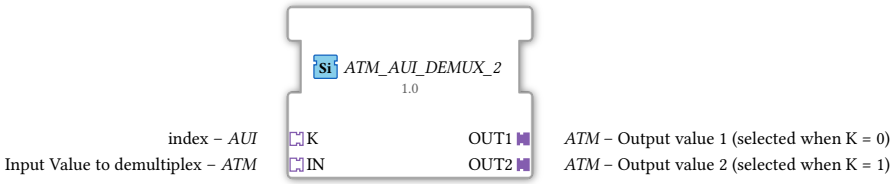
- **`AIS_AUI_MUX_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AIS_AUI_MUX_5_UNGATED` überträgt die Multiplexer-Logik von `AIS_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

TIME

ATM_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der ATM_AUI_DEMUX_2 ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp TIME. Anders als ATM_DEMUX_2 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet ATM_AUI_DEMUX_2 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (TIME, Zeitdauer), der an einen der n Ausgänge weitergeleitet wird (ATM-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (ATM-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (ATM-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 2 Ausgangs-Adapter (OUT1 … OUT2) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_ATM_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ATM_AUI_DEMUX_2 … ATM_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`TIME`, Zeitdauer) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ATM_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

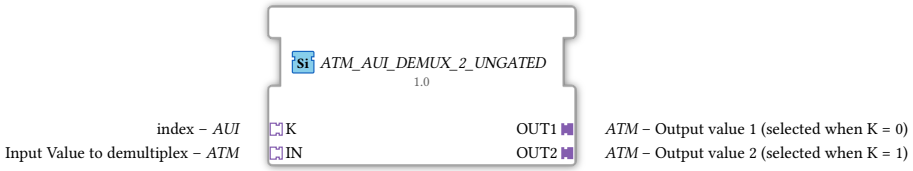
- **`ATM_AUI_DEMUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `ATM_AUI_DEMUX_2` überträgt die Demultiplexer-Logik von `ATM_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ATM_AUI_DEMUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ATM_AUI_DEMUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ATM_AUI_DEMUX_2_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `TIME`. Anders als `ATM_DEMUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ATM_AUI_DEMUX_2_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`TIME`, Zeitdauer), der an einen der `n` Ausgänge weitergeleitet wird (ATM-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (ATM-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (ATM-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Ausgangs-Adapter (`OUT1` … `OUT2`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ATM_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ATM_AUI_DEMUX_2_UNGATED … ATM_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`TIME`, Zeitdauer) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ATM_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

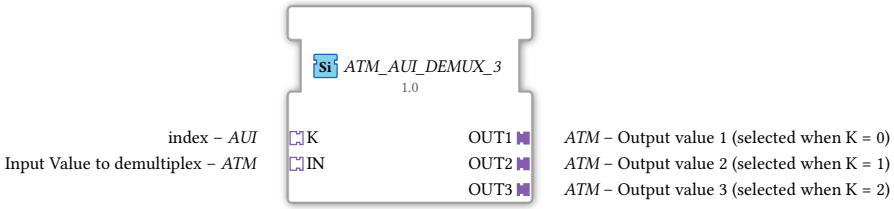
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- `ATM_AUI_DEMUX_2`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `ATM_AUI_DEMUX_2_UNGATED` überträgt die Demultiplexer-Logik von `ATM_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ATM_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ATM_AUI_DEMUX_3` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `TIME`. Anders als `ATM_DEMUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ATM_AUI_DEMUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`TIME`, Zeitdauer), der an einen der `n` Ausgänge weitergeleitet wird (ATM-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (ATM-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (ATM-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (ATM-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Ausgangs-Adapter (`OUT1` … `OUT3`) gerade ausgewählt ist.

2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_ATM_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ATM_AUI_DEMUX_2 … ATM_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen IN-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (TIME, Zeitdauer) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit ATM_DEMUX_3 (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

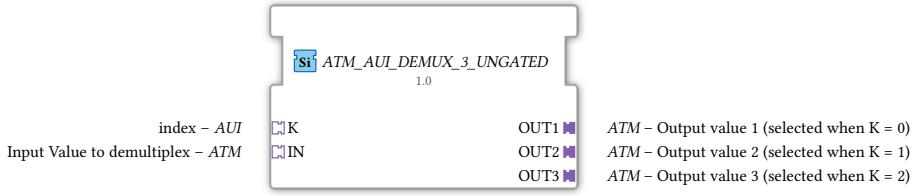
- **ATM_AUI_DEMUX_3_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der ATM_AUI_DEMUX_3 überträgt die Demultiplexer-Logik von ATM_DEMUX_3 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ATM_AUI_DEMUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von ATM_AUI_DEMUX_3. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ATM_AUI_DEMUX_3_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `TIME`. Anders als `ATM_DEMUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ATM_AUI_DEMUX_3_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`TIME`, Zeitdauer), der an einen der `n` Ausgänge weitergeleitet wird (ATM-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (ATM-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (ATM-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (ATM-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Ausgangs-Adapter (`OUT1` … `OUT3`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.

- Generische Implementierung (`GEN_ATM_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ATM_AUI_DEMUX_2 … ATM_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`TIME`, Zeitdauer) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ATM_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

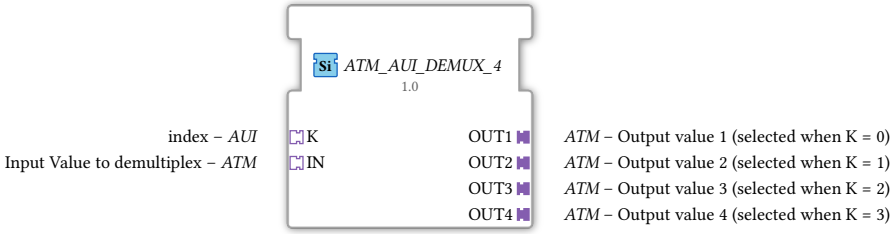
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`ATM_AUI_DEMUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `ATM_AUI_DEMUX_3_UNGATED` überträgt die Demultiplexer-Logik von `ATM_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ATM_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ATM_AUI_DEMUX_4` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `TIME`. Anders als `ATM_DEMUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ATM_AUI_DEMUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`TIME`, Zeitdauer), der an einen der `n` Ausgänge weitergeleitet wird (ATM-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (ATM-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (ATM-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (ATM-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (ATM-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ATM_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ATM_AUI_DEMUX_2 … ATM_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`TIME`, Zeitdauer) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index

- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ATM_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

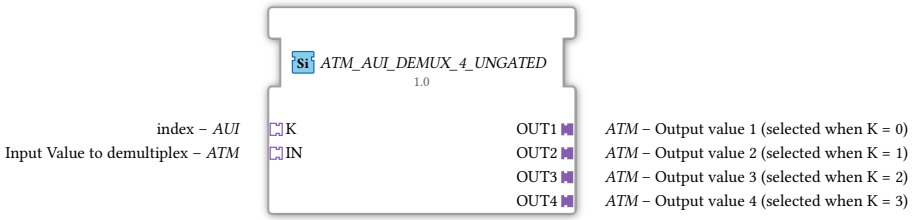
- **`ATM_AUI_DEMUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `ATM_AUI_DEMUX_4` überträgt die Demultiplexer-Logik von `ATM_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ATM_AUI_DEMUX_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ATM_AUI_DEMUX_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ATM_AUI_DEMUX_4_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `TIME`. Anders als `ATM_DEMUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ATM_AUI_DEMUX_4_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`TIME`, Zeitdauer), der an einen der `n` Ausgänge weitergeleitet wird (ATM-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (ATM-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (ATM-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (ATM-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (ATM-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ATM_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ATM_AUI_DEMUX_2 … ATM_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`TIME`, Zeitdauer) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

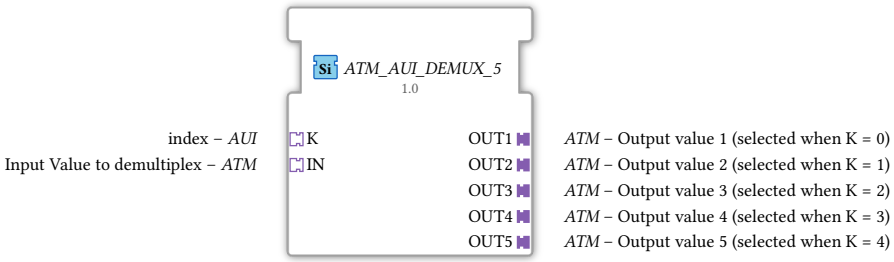
Vergleich mit `ATM_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`ATM_AUI_DEMUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `ATM_AUI_DEMUX_4_UNGATED` überträgt die Demultiplexer-Logik von `ATM_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ATM_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ATM_AUI_DEMUX_5` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `TIME`. Anders als `ATM_DEMUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ATM_AUI_DEMUX_5` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`TIME`, Zeitdauer), der an einen der `n` Ausgänge weitergeleitet wird (ATM-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (ATM-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (ATM-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (ATM-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (ATM-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (ATM-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Ausgangs-Adapter (`OUT1` … `OUT5`) gerade ausgewählt ist.

2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_ATM_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ATM_AUI_DEMUX_2 … ATM_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen IN-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (TIME, Zeitdauer) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit ATM_DEMUX_5 (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

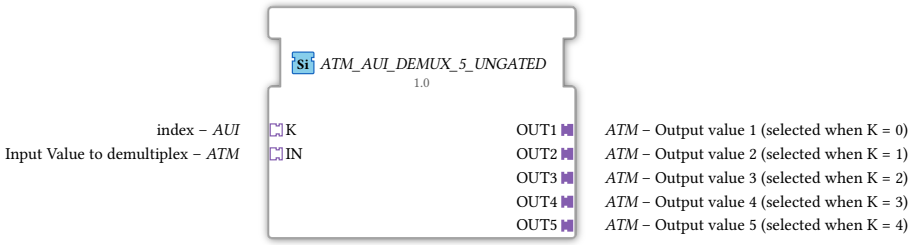
- **ATM_AUI_DEMUX_5_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der ATM_AUI_DEMUX_5 überträgt die Demultiplexer-Logik von ATM_DEMUX_5 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ATM_AUI_DEMUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von ATM_AUI_DEMUX_5. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ATM_AUI_DEMUX_5_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `TIME`. Anders als `ATM_DEMUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ATM_AUI_DEMUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`TIME`, Zeitdauer), der an einen der `n` Ausgänge weitergeleitet wird (ATM-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (ATM-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (ATM-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (ATM-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (ATM-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (ATM-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Ausgangs-Adapter (`OUT1` … `OUT5`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ATM_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ATM_AUI_DEMUX_2 … ATM_AUI_DEMUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`TIME`, Zeitdauer) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

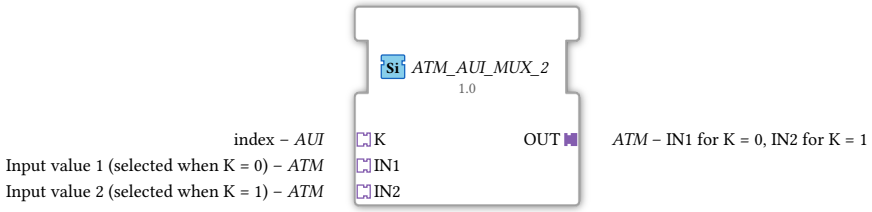
Vergleich mit `ATM_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`ATM_AUI_DEMUX_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `ATM_AUI_DEMUX_5_UNGATED` überträgt die Demultiplexer-Logik von `ATM_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ATM_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ATM_AUI_MUX_2` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `TIME`. Anders als `ATM_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ATM_AUI_MUX_2` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (ATM-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (ATM-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ATM-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ATM_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ATM_AUI_MUX_2 … ATM_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (Zeitdauer) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ATM_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

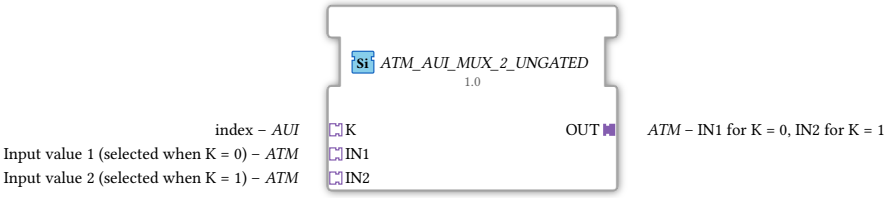
- **`ATM_AUI_MUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `ATM_AUI_MUX_2` überträgt die Multiplexer-Logik von `ATM_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ATM_AUI_MUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ATM_AUI_MUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ATM_AUI_MUX_2_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `TIME`. Anders als `ATM_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ATM_AUI_MUX_2_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (ATM-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (ATM-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ATM-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ATM_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ATM_AUI_MUX_2_UNGATED … ATM_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (Zeitdauer) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ATM_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

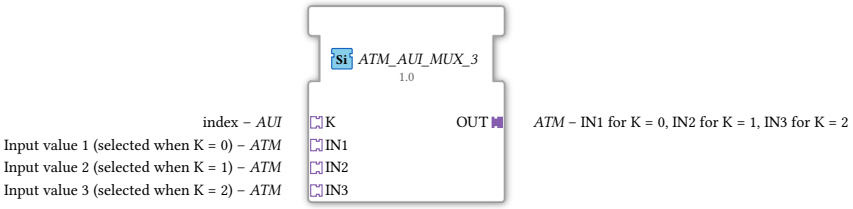
Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`ATM_AUI_MUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `ATM_AUI_MUX_2_UNGATED` überträgt die Multiplexer-Logik von `ATM_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ATM_AUI_MUX_3



• – ○ • – ○ • – ○ •

Einleitung

Der `ATM_AUI_MUX_3` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `TIME`. Anders als `ATM_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ATM_AUI_MUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (ATM-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (ATM-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (ATM-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ATM-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN3`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Eingangs-Adapter (`IN1` … `IN3`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ATM_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ATM_AUI_MUX_2 … ATM_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (Zeitdauer) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ATM_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

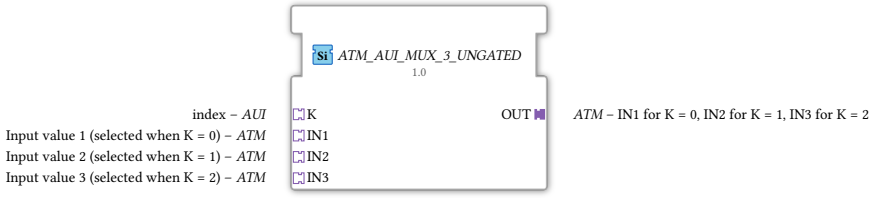
- `ATM_AUI_MUX_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `ATM_AUI_MUX_3` überträgt die Multiplexer-Logik von `ATM_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ATM_AUI_MUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ATM_AUI_MUX_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `ATM_AUI_MUX_3_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `TIME`. Anders als `ATM_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit

zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ATM_AUI_MUX_3_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (ATM-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (ATM-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (ATM-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ATM-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN3`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Eingangs-Adapter (`IN1` … `IN3`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ATM_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ATM_AUI_MUX_2 … ATM_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (Zeitdauer) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ATM_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

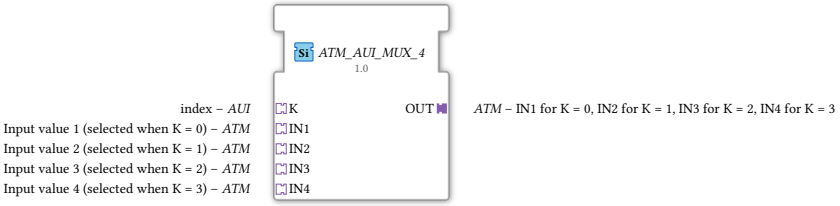
Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`ATM_AUI_MUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `ATM_AUI_MUX_3_UNGATED` überträgt die Multiplexer-Logik von `ATM_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ATM_AUI_MUX_4



• – ○ • – ○ • – ○ •

Einleitung

Der `ATM_AUI_MUX_4` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `TIME`. Anders als `ATM_MUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ATM_AUI_MUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index `K = 0` an `OUT` durchgereicht (ATM-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index `K = 1` an `OUT` durchgereicht (ATM-Adaptertyp).

- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index `K = 2` an `OUT` durchgereicht (ATM-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index `K = 3` an `OUT` durchgereicht (ATM-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ATM-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN4`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Eingangs-Adapter (`IN1` … `IN4`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ATM_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ATM_AUI_MUX_2 … ATM_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (Zeitdauer) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ATM_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- `ATM_AUI_MUX_4_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `ATM_AUI_MUX_4` überträgt die Multiplexer-Logik von `ATM_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ATM_AUI_MUX_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ATM_AUI_MUX_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos

weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ATM_AUI_MUX_4_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `TIME`. Anders als `ATM_MUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ATM_AUI_MUX_4_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index `K = 0` an `OUT` durchgereicht (ATM-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index `K = 1` an `OUT` durchgereicht (ATM-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index `K = 2` an `OUT` durchgereicht (ATM-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index `K = 3` an `OUT` durchgereicht (ATM-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ATM-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN4`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Eingangs-Adapter (`IN1` … `IN4`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_ATM_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ATM_AUI_MUX_2 … ATM_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN1…IN4) ⟶ aktuell selektierten Eingang auslesen, OUT bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (Zeitdauer) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit ATM_MUX_4 (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

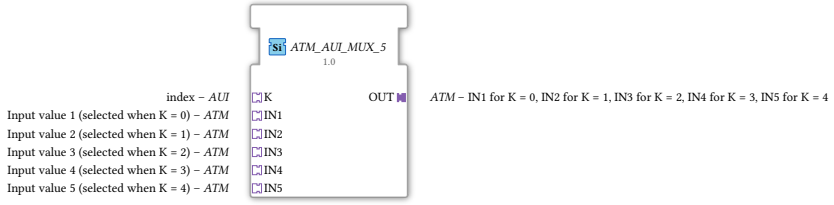
Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **ATM_AUI_MUX_4**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der ATM_AUI_MUX_4_UNGATED überträgt die Multiplexer-Logik von ATM_MUX_4 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ATM_AUI_MUX_5



• – ○ • – ○ • – ○ •

Einleitung

Der ATM_AUI_MUX_5 ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp TIME. Anders als ATM_MUX_5 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ATM_AUI_MUX_5` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index `K = 0` an `OUT` durchgereicht (ATM-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index `K = 1` an `OUT` durchgereicht (ATM-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index `K = 2` an `OUT` durchgereicht (ATM-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index `K = 3` an `OUT` durchgereicht (ATM-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index `K = 4` an `OUT` durchgereicht (ATM-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ATM-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN5`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Eingangs-Adapter (`IN1` … `IN5`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ATM_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ATM_AUI_MUX_2 … ATM_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (Zeitdauer) über einen zentral verwalteten Adapter-Index

- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ATM_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

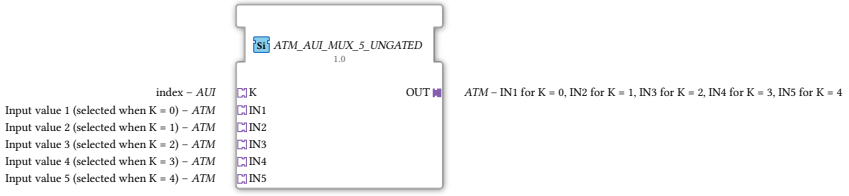
- `ATM_AUI_MUX_5_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `ATM_AUI_MUX_5` überträgt die Multiplexer-Logik von `ATM_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

ATM_AUI_MUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `ATM_AUI_MUX_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `ATM_AUI_MUX_5_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `TIME`. Anders als `ATM_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `ATM_AUI_MUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index `K = 0` an `OUT` durchgereicht (ATM-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index `K = 1` an `OUT` durchgereicht (ATM-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index `K = 2` an `OUT` durchgereicht (ATM-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index `K = 3` an `OUT` durchgereicht (ATM-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index `K = 4` an `OUT` durchgereicht (ATM-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (ATM-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN5`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Eingangs-Adapter (`IN1` … `IN5`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_ATM_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (ATM_AUI_MUX_2 … ATM_AUI_MUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (Zeitdauer) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `ATM_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

- `ATM_AUI_MUX_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

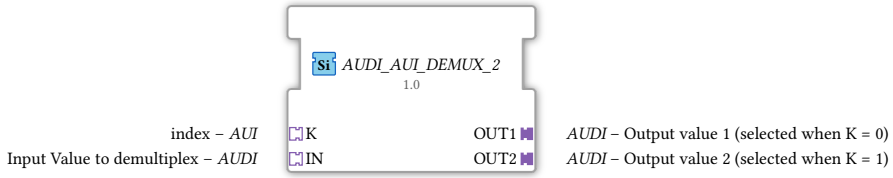
Fazit

Der `ATM_AUI_MUX_5_UNGATED` überträgt die Multiplexer-Logik von `ATM_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter

aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

UDINT

AUDI_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AUDI_AUI_DEMUX_2` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `UDINT`. Anders als `AUDI_DEMUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUDI_AUI_DEMUX_2` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`UDINT`, 32-Bit-Ganzzahl (vorzeichenlos)), der an einen der `n` Ausgänge weitergeleitet wird (AUDI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AUDI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AUDI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Ausgangs-Adapter (`OUT1` … `OUT2`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUDI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUDI_AUI_DEMUX_2 … AUDI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`UDINT`, 32-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUDI_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

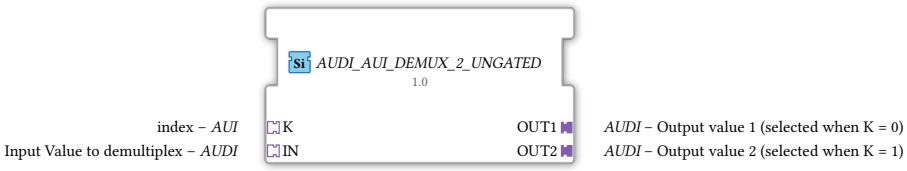
- **`AUDI_AUI_DEMUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AUDI_AUI_DEMUX_2` überträgt die Demultiplexer-Logik von `AUDI_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUDI_AUI_DEMUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUDI_AUI_DEMUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `AUDI_AUI_DEMUX_2_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `UDINT`. Anders als `AUDI_DEMUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit

zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUDI_AUI_DEMUX_2_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`UDINT`, 32-Bit-Ganzzahl (vorzeichenlos)), der an einen der `n` Ausgänge weitergeleitet wird (AUDI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AUDI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AUDI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Ausgangs-Adapter (`OUT1` … `OUT2`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUDI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUDI_AUI_DEMUX_2_UNGATED … AUDI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`UDINT`, 32-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUDI_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

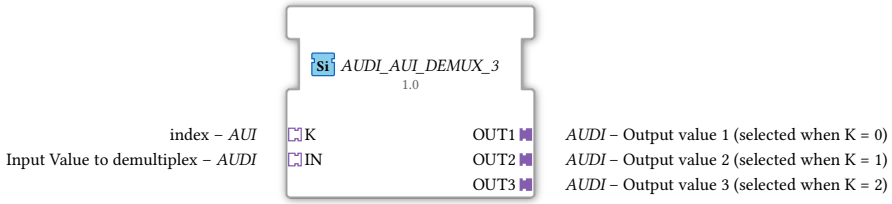
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AUDI_AUI_DEMUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AUDI_AUI_DEMUX_2_UNGATED` überträgt die Demultiplexer-Logik von `AUDI_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUDI_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AUDI_AUI_DEMUX_3` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `UDINT`. Anders als `AUDI_DEMUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUDI_AUI_DEMUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.

- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`UDINT`, 32-Bit-Ganzzahl (vorzeichenlos)), der an einen der `n` Ausgänge weitergeleitet wird (AUDI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AUDI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AUDI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AUDI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Ausgangs-Adapter (`OUT1` … `OUT3`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUDI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUDI_AUI_DEMUX_2 … AUDI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`UDINT`, 32-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUDI_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

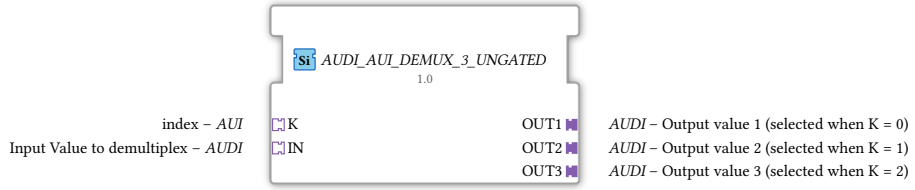
- **`AUDI_AUI_DEMUX_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AUDI_AUI_DEMUX_3` überträgt die Demultiplexer-Logik von `AUDI_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUDI_AUI_DEMUX_3_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von AUDI_AUI_DEMUX_3. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AUDI_AUI_DEMUX_3_UNGATED ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp UDINT. Anders als AUDI_DEMUX_3 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AUDI_AUI_DEMUX_3_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (UDINT, 32-Bit-Ganzzahl (vorzeichenlos)), der an einen der n Ausgänge weitergeleitet wird (AUDI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AUDI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AUDI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AUDI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 3 Ausgangs-Adapter (OUT1 … OUT3) gerade ausgewählt ist.

2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUDI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUDI_AUI_DEMUX_2 … AUDI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`UDINT`, 32-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

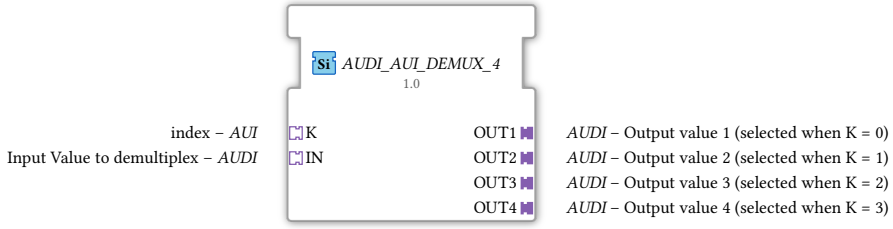
Vergleich mit `AUDI_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AUDI_AUI_DEMUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AUDI_AUI_DEMUX_3_UNGATED` überträgt die Demultiplexer-Logik von `AUDI_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUDI_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AUDI_AUI_DEMUX_4` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `UDINT`. Anders als `AUDI_DEMUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit

zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUDI_AUI_DEMUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`UDINT`, 32-Bit-Ganzzahl (vorzeichenlos)), der an einen der `n` Ausgänge weitergeleitet wird (AUDI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AUDI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AUDI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AUDI-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AUDI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUDI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUDI_AUI_DEMUX_2 … AUDI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`UDINT`, 32-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUDI_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

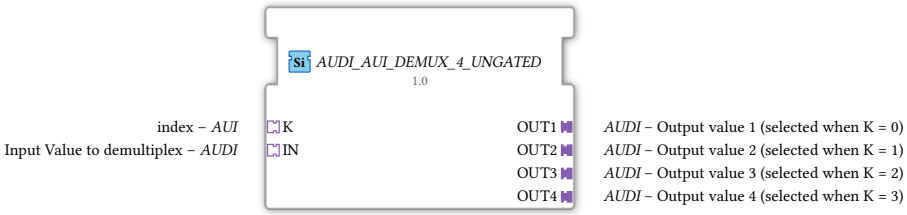
- **`AUDI_AUI_DEMUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AUDI_AUI_DEMUX_4` überträgt die Demultiplexer-Logik von `AUDI_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUDI_AUI_DEMUX_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUDI_AUI_DEMUX_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AUDI_AUI_DEMUX_4_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `UDINT`. Anders als `AUDI_DEMUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUDI_AUI_DEMUX_4_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`UDINT`, 32-Bit-Ganzzahl (vorzeichenlos)), der an einen der `n` Ausgänge weitergeleitet wird (AUDI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AUDI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AUDI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AUDI-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AUDI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUDI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUDI_AUI_DEMUX_2 … AUDI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`UDINT`, 32-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUDI_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

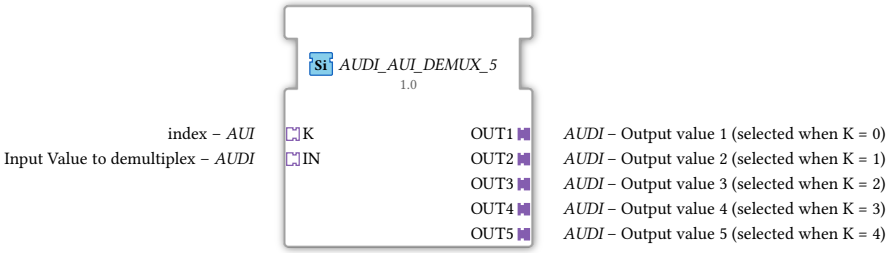
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **AUDI_AUI_DEMUX_4**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der AUDI_AUI_DEMUX_4_UNGATED überträgt die Demultiplexer-Logik von AUDI_DEMUX_4 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUDI_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AUDI_AUI_DEMUX_5 ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp UDINT. Anders als AUDI_DEMUX_5 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AUDI_AUI_DEMUX_5 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (UDINT, 32-Bit-Ganzzahl (vorzeichenlos)), der an einen der n Ausgänge weitergeleitet wird (AUDI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AUDI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AUDI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AUDI-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AUDI-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AUDI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Ausgangs-Adapter (`OUT1` … `OUT5`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUDI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUDI_AUI_DEMUX_2 … AUDI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`UDINT`, 32-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUDI_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

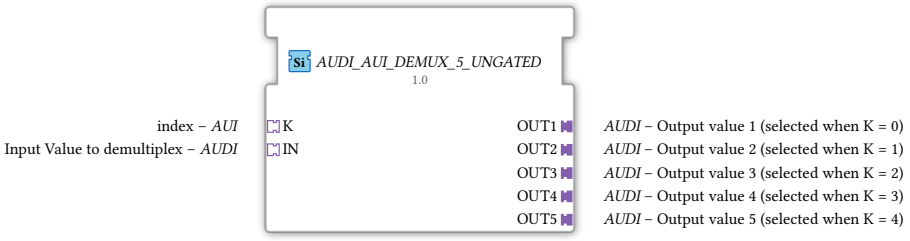
- **`AUDI_AUI_DEMUX_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AUDI_AUI_DEMUX_5` überträgt die Demultiplexer-Logik von `AUDI_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUDI_AUI_DEMUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUDI_AUI_DEMUX_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AUDI_AUI_DEMUX_5_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `UDINT`. Anders als `AUDI_DEMUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUDI_AUI_DEMUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`UDINT`, 32-Bit-Ganzzahl (vorzeichenlos)), der an einen der `n` Ausgänge weitergeleitet wird (AUDI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AUDI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AUDI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AUDI-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AUDI-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AUDI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Ausgangs-Adapter (`OUT1` … `OUT5`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUDI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUDI_AUI_DEMUX_2 … AUDI_AUI_DEMUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`UDINT`, 32-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

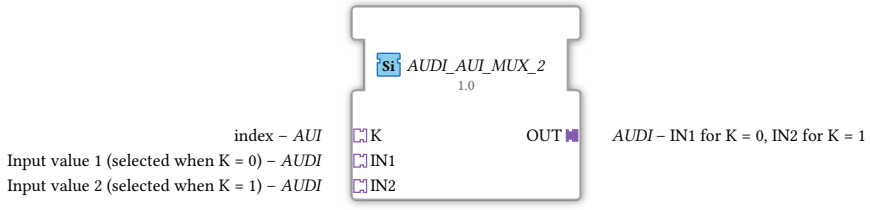
Vergleich mit `AUDI_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AUDI_AUI_DEMUX_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AUDI_AUI_DEMUX_5_UNGATED` überträgt die Demultiplexer-Logik von `AUDI_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUDI_AUI_MUX_2



• – ○ • – ○ • – ○ •

Einleitung

Der `AUDI_AUI_MUX_2` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `UDINT`. Anders als `AUDI_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUDI_AUI_MUX_2` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (AUDI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (AUDI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AUDI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUDI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUDI_AUI_MUX_2 … AUDI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (32-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird

- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUDI_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

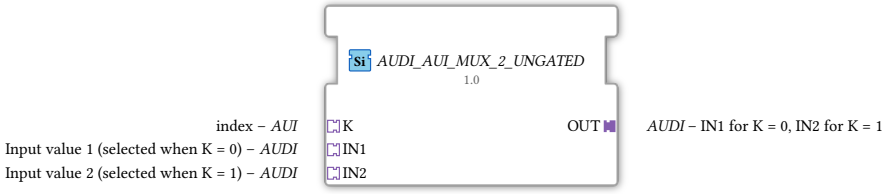
- **`AUDI_AUI_MUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AUDI_AUI_MUX_2` überträgt die Multiplexer-Logik von `AUDI_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUDI_AUI_MUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUDI_AUI_MUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `AUDI_AUI_MUX_2_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `UDINT`. Anders als `AUDI_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUDI_AUI_MUX_2_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (AUDI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (AUDI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AUDI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUDI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUDI_AUI_MUX_2_UNGATED … AUDI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (32-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUDI_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

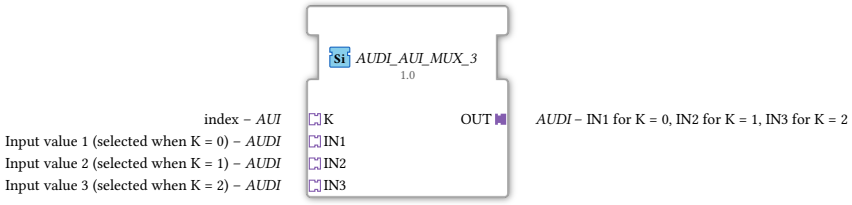
Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AUDI_AUI_MUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AUDI_AUI_MUX_2_UNGATED` überträgt die Multiplexer-Logik von `AUDI_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUDI_AUI_MUX_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AUDI_AUI_MUX_3` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `UDINT`. Anders als `AUDI_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUDI_AUI_MUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (AUDI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (AUDI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (AUDI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AUDI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN3`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Eingangs-Adapter (`IN1` … `IN3`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUDI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUDI_AUI_MUX_2 … AUDI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (32-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUDI_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

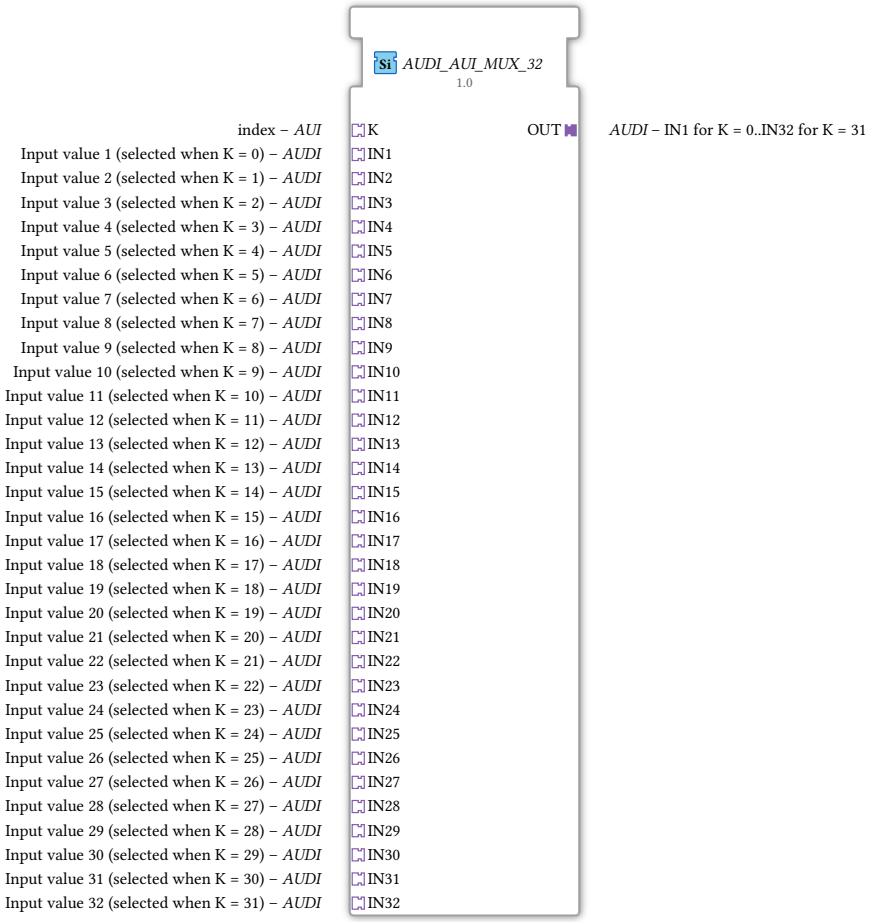
Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AUDI_AUI_MUX_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AUDI_AUI_MUX_3` überträgt die Multiplexer-Logik von `AUDI_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUDI_AUI_MUX_32



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `AUDI_AUI_MUX_32` ist ein generischer Multiplexer, der es ermöglicht, aus 32 unidirektionalen Eingangssignalen (`IN1` bis `IN32`) genau eines auszuwählen und auf einen unidirektionalen Ausgang (`OUT`) zu übertragen. Die Auswahl wird über einen separaten Indexadapter `K` gesteuert, der den Wert des gewünschten Kanals (0 bis 31) liefert. Ein besonderes Merkmal ist, dass das ausgegebene Signal nur dann aktualisiert wird, wenn sich der Wert des ausgewählten Eingangs tatsächlich ändert. Der Baustein ist als generischer Typ (`GEN_AUDI_AUI_MUX`) implementiert und kann dadurch mit unterschiedlichen Datentypen verwendet werden, solange die verwendeten Adapter vom entsprechenden Typ sind.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock besitzt **keine** Ereignis-Eingänge.

Ereignis-Ausgänge

Der Funktionsblock besitzt **keine** Ereignis-Ausgänge.

Daten-Eingänge

Der Funktionsblock besitzt **keine** direkten Dateneingänge. Die Datenübertragung erfolgt ausschließlich über Adapter.

Daten-Ausgänge

Der Funktionsblock besitzt **keine** direkten Datenausgänge. Die Datenausgabe erfolgt über den Adapter `OUT`.

Adapter

Der Funktionsblock verwendet ausschließlich Adapter für seine Ein- und Ausgaben. Alle Adapter sind **unidirektional** und stammen aus den Typen `AUDI` bzw. `AUI`.

Name	Richtung	Typ	Kommentar
`OUT`	Plug	`adapter::types::unidirectional::AUDI`	Ausgang: liefert den ausgewählten Eingangswert.
`K`	Socket	`adapter::types::unidirectional::AUI`	Index (0 bis 31), der den aktiven Eingang bestimmt.
`IN1`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 1, aktiv bei K = 0.
`IN2`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 2, aktiv bei K = 1.
`IN3`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 3, aktiv bei K = 2.
`IN4`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 4, aktiv bei K = 3.
`IN5`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 5, aktiv bei K = 4.
`IN6`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 6, aktiv bei K = 5.
`IN7`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 7, aktiv bei K = 6.
`IN8`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 8, aktiv bei K = 7.
`IN9`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 9, aktiv bei K = 8.
`IN10`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 10, aktiv bei K = 9.
`IN11`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 11, aktiv bei K = 10.
`IN12`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 12, aktiv bei K = 11.
`IN13`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 13, aktiv bei K = 12.
`IN14`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 14, aktiv bei K = 13.
`IN15`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 15, aktiv bei K = 14.
`IN16`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 16, aktiv bei K = 15.
`IN17`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 17, aktiv bei K = 16.
`IN18`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 18, aktiv bei K = 17.
`IN19`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 19, aktiv bei K = 18.
`IN20`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 20, aktiv bei K = 19.
`IN21`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 21, aktiv bei K = 20.
`IN22`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 22, aktiv bei K = 21.
`IN23`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 23, aktiv bei K = 22.
`IN24`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 24, aktiv bei K = 23.
`IN25`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 25, aktiv bei K = 24.
`IN26`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 26, aktiv bei K = 25.
`IN27`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 27, aktiv bei K = 26.
`IN28`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 28, aktiv bei K = 27.
`IN29`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 29, aktiv bei K = 28.
`IN30`	Socket	`adapter::types::unidirectional::AUDI`	Eingang 30, aktiv bei K = 29.

IN31	Socket	adapter::types::unidirectional::AUDI	Eingang 31, aktiv bei K = 30.
IN32	Socket	adapter::types::unidirectional::AUDI	Eingang 32, aktiv bei K = 31.

Funktionsweise

Der AUDI_AUI_MUX_32 arbeitet als klassischer Multiplexer mit 32 Kanälen. Der über den Adapter K anliegende Wert bestimmt, welcher der 32 Eingänge (IN1 bis IN32) auf den Ausgang OUT durchgeschaltet wird. Der Index K wird als ganzzahliger Wert erwartet, wobei gültige Werte von 0 bis 31 reichen (bei Werten außerhalb dieses Bereichs wird das Verhalten nicht definiert und das ausgewählte Signal bleibt unverändert).

Die Besonderheit dieses Bausteins liegt in der auslösenden Bedingung für die Ausgangsaktualisierung: Der Ausgang OUT wird nur dann mit einem neuen Wert belegt, wenn sich der Wert des aktuell ausgewählten Eingangs gegenüber dem zuletzt übertragenen Wert geändert hat. Eine Änderung des Index K allein löst noch keine Aktualisierung aus, es sei denn, der neu ausgewählte Eingang hat einen anderen Wert als der zuvor aktive. Sobald eine solche Änderung erkannt wird, wird der Ausgang aktualisiert. Dies reduziert unnötigen Datenverkehr und Ereignisfrequenzen im nachgeschalteten System.

Die Kommunikation über die Adapter ist unidirektional: Die Eingangsadapter (IN*) liefern Daten an den Baustein, der Indexadapter K liefert den Steuerwert, und der Ausgangsadapter OUT überträgt das ausgewählte Signal an die aufnehmende Einheit. Intern basiert der Baustein auf dem generischen Typ GEN_AUDI_AUI_MUX, sodass er mit unterschiedlichen Datentypen wie z. B. INT, REAL, BOOL oder auch strukturierten Typen arbeiten kann, sofern die jeweils verwendeten Adapter vom Typ AUDI und AUI passend instanziiert sind.

Technische Besonderheiten

- **Generischer Aufbau:** Der FB ist als generischer Baustein implementiert (Attribut GenericClassName = 'GEN_AUDI_AUI_MUX'), wodurch er für verschiedene Datentypen verwendet werden kann. Die konkrete Datentypbindung erfolgt über die Typisierung der verwendeten Adapter.
- **Änderungsgesteuerte Aktualisierung:** Der Ausgang wird nur dann aktualisiert, wenn sich der Wert des ausgewählten Eingangs tatsächlich ändert. Bei gleichbleibenden Werten erfolgt keine Aktualisierung, was die Systemlast minimiert.
- **Unidirektionale Adapter:** Alle Adapter sind als unidirektional deklariert. Dadurch wird eine klare Datenflussrichtung ohne Rückwärtskommunikation gewährleistet.
- **32 Kanäle:** Der Baustein unterstützt 32 Eingänge und einen einzigen Ausgang – ideal für Anwendungen, bei denen viele Signale auf einen gemeinsamen Bus ausgewählt werden müssen.

Zustandsübersicht

Der Baustein besitzt **keinen** expliziten endlichen Zustandsautomaten im klassischen Sinn. Intern kann man sich eine Speicherung des letzten aktiven Index und des letzten ausgegebenen Werts vorstellen. Es gibt keine Zustände wie „IDLE“ oder „BUSY“, sondern lediglich die datenflussbasierte Verarbeitung. Der Baustein ist permanent aktiv und reagiert auf Änderungen an den Eingängen bzw. am Index.

Anwendungsszenarien

- **Sensorauswahl:** In einer Maschine mit vielen analogen oder digitalen Sensoren (z. B. Temperaturen, Drücke, Positionen) kann über den Indexadapter K jeweils der gewünschte Sensorwert auf einen einzigen Ausgang gelegt und an ein zentrales Steuerungssystem weitergeleitet werden.
- **Datenpriorisierung:** In Kommunikationsprotokollen kann der Multiplexer dazu verwendet werden, je nach Betriebsmodus den relevanten Datenkanal auszuwählen.
- **Test- und Überwachungsanwendungen:** Zur Überprüfung einzelner Signalpfade können nacheinander verschiedene Eingänge durchgeschaltet werden, ohne dass mehrere parallele Ausgänge benötigt werden.
- **Konfigurierbare Verschaltung:** In Automatisierungssystemen, bei denen die Zuordnung von Eingangssignalen zu einem Ausgang zur Laufzeit geändert werden muss, bietet dieser FB eine flexible Lösung.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem Standard-Multiplexer, der bei *jeder* Änderung an einem Eingang oder am Index ein Ereignis auslöst, zeichnet sich dieser Baustein durch seine **änderungsbasierte** Ausgangsaktualisierung aus. Dadurch wird verhindert, dass unnötige Ereignisse gesendet werden, wenn sich das ausgewählte

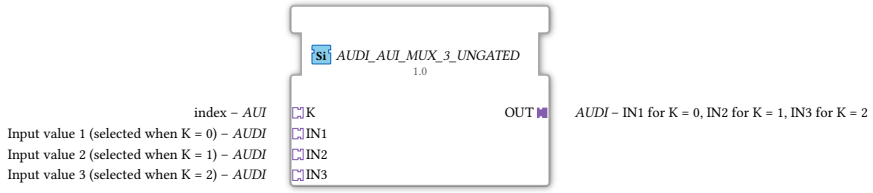
Signal nicht ändert. Ein weiterer Unterschied ist die Verwendung von **unidirektionalen Adaptern** anstelle von direkten Datenein-/-ausgängen, was die Wiederverwendbarkeit und Flexibilität erhöht. Typische mehrstufig aufgebaute Multiplexer mit mehreren Bausteinen (z. B. Baumstruktur) würden bei gleicher Kanalzahl mehr Bausteine und Verdrahtung benötigen; `AUDI_AUI_MUX_32` integriert hingegen die komplette 32-zu-1-Auswahl in einem einzigen FB.

Fazit

Der `AUDI_AUI_MUX_32` ist ein leistungsfähiger, generischer Multiplexer für 32 unidirektionale Signale. Seine Stärke liegt in der effizienten, ereignisgesteuerten Auswahl von Eingängen auf einen gemeinsamen Ausgang, wobei nur tatsächliche Wertänderungen kommuniziert werden. Die Verwendung von Adaptern erlaubt eine flexible Anpassung an unterschiedliche Datentypen und sorgt für eine saubere Schnittstellenstruktur. Der Baustein eignet sich insbesondere für Automatisierungslösungen, in denen eine dynamische Kanalwahl bei minimierter Ereignislast gefordert ist.

AUDI_AUI_MUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUDI_AUI_MUX_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `AUDI_AUI_MUX_3_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `UDINT`. Anders als `AUDI_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUDI_AUI_MUX_3_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (AUDI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (AUDI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (AUDI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AUDI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN3`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Eingangs-Adapter (`IN1` … `IN3`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUDI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUDI_AUI_MUX_2 … AUDI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (32-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUDI_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AUDI_AUI_MUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AUDI_AUI_MUX_3_UNGATED` überträgt die Multiplexer-Logik von `AUDI_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUDI_AUI_MUX_4

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AUDI_AUI_MUX_4` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `UDINT`. Anders als `AUDI_MUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUDI_AUI_MUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index `K = 0` an `OUT` durchgereicht (AUDI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index `K = 1` an `OUT` durchgereicht (AUDI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index `K = 2` an `OUT` durchgereicht (AUDI-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index `K = 3` an `OUT` durchgereicht (AUDI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AUDI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN4`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Eingangs-Adapter (`IN1` … `IN4`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUDI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUDI_AUI_MUX_2 … AUDI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (32-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUDI_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AUDI_AUI_MUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AUDI_AUI_MUX_4` überträgt die Multiplexer-Logik von `AUDI_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUDI_AUI_MUX_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUDI_AUI_MUX_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ○ • – ○ • – ○ •

Einleitung

Der AUDI_AUI_MUX_4_UNGATED ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp UDINT. Anders als AUDI_MUX_4 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AUDI_AUI_MUX_4_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index K = 0 an OUT durchgereicht (AUDI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index K = 1 an OUT durchgereicht (AUDI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index K = 2 an OUT durchgereicht (AUDI-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index K = 3 an OUT durchgereicht (AUDI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AUDI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN4:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 4 Eingangs-Adapter (IN1 … IN4) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf OUT gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AUDI_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUDI_AUI_MUX_2 … AUDI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (32-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUDI_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AUDI_AUI_MUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AUDI_AUI_MUX_4_UNGATED` überträgt die Multiplexer-Logik von `AUDI_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUDI_AUI_MUX_5

• – ○ • – ○ • – ○ •

Einleitung

Der `AUDI_AUI_MUX_5` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `UDINT`. Anders als `AUDI_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUDI_AUI_MUX_5` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index `K = 0` an `OUT` durchgereicht (AUDI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index `K = 1` an `OUT` durchgereicht (AUDI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index `K = 2` an `OUT` durchgereicht (AUDI-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index `K = 3` an `OUT` durchgereicht (AUDI-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index `K = 4` an `OUT` durchgereicht (AUDI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AUDI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN5`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Eingangs-Adapter (`IN1` … `IN5`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUDI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUDI_AUI_MUX_2 … AUDI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (32-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUDI_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

- `AUDI_AUI_MUX_5_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der AUDI_AUI_MUX_5 überträgt die Multiplexer-Logik von AUDI_MUX_5 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUDI_AUI_MUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AUDI_AUI_MUX_5. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ○ • – ○ • – ○ •

Einleitung

Der AUDI_AUI_MUX_5_UNGATED ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp UDINT. Anders als AUDI_MUX_5 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AUDI_AUI_MUX_5_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index K = 0 an OUT durchgereicht (AUDI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index K = 1 an OUT durchgereicht (AUDI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index K = 2 an OUT durchgereicht (AUDI-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index K = 3 an OUT durchgereicht (AUDI-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index K = 4 an OUT durchgereicht (AUDI-Adaptertyp).

- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AUDI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN5`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Eingangs-Adapter (`IN1` … `IN5`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUDI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUDI_AUI_MUX_2 … AUDI_AUI_MUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (32-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUDI_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

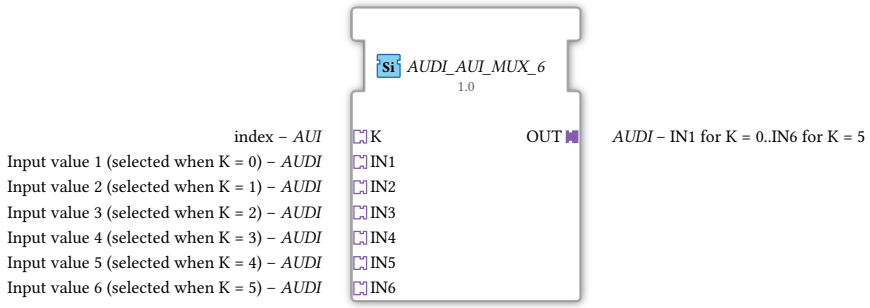
Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

- **`AUDI_AUI_MUX_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AUDI_AUI_MUX_5_UNGATED` überträgt die Multiplexer-Logik von `AUDI_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUDI_AUI_MUX_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AUDI_AUI_MUX_6** realisiert einen generischen Multiplexer, der aus sechs unidirektionalen Adaptereingängen (IN1 bis IN6) den aktuell ausgewählten Wert über den Adapterausgang OUT bereitstellt. Die Auswahl des aktiven Eingangs erfolgt über einen separaten Indexadapter K, der Werte von 0 bis 5 annimmt. Der Ausgangswert wird nur dann aktualisiert, wenn er sich tatsächlich ändert, wodurch unnötige Aktualisierungen vermieden werden.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge.

Daten-Eingänge

Keine direkten Dateneingänge definiert. Die Daten werden ausschließlich über die Adaptereingänge übertragen.

Daten-Ausgänge

Keine direkten Datenausgänge definiert. Die Daten werden ausschließlich über den Adapterausgang bereitgestellt.

Adapter

Adapter	Typ	Richtung	Beschreibung
`OUT`	`adapter::types::unidirectional::AUDI`	Plug	Ausgangsadapter, stellt den ausgewählten Eingangswert bereit.
`K`	`adapter::types::unidirectional::AUI`	Socket	Indexauswahl (0–5).
`IN1`	`adapter::types::unidirectional::AUDI`	Socket	Eingangswert 1 (wird bei K=0 ausgewählt).
`IN2`	`adapter::types::unidirectional::AUDI`	Socket	Eingangswert 2 (K=1).
`IN3`	`adapter::types::unidirectional::AUDI`	Socket	Eingangswert 3 (K=2).
`IN4`	`adapter::types::unidirectional::AUDI`	Socket	Eingangswert 4 (K=3).
`IN5`	`adapter::types::unidirectional::AUDI`	Socket	Eingangswert 5 (K=4).
`IN6`	`adapter::types::unidirectional::AUDI`	Socket	Eingangswert 6 (K=5).

Funktionsweise

Der Baustein arbeitet als Adapter-basierter Multiplexer: Der über den Socket `K` ankommende Indexwert bestimmt, welcher der sechs Eingangsadapter (`IN1`–`IN6`) an den Ausgangsadapter `OUT` weitergereicht wird. Die Adapter sind unidirektional, d.h. die Datenübertragung erfolgt von den Sockets zum Plug.

Bei jeder Änderung des Indexwerts `K` wird der entsprechende Eingang ausgewählt. Der Ausgangswert wird nur dann aktualisiert, wenn sich der vom neuen Eingang gelieferte Wert vom vorherigen Ausgangswert unterscheidet. Bei gleichen Werten bleibt der Ausgang unverändert.

Technische Besonderheiten

- **Generische Implementierung:** Der Baustein wird als generischer FB (GenericClassName `'GEN_AUDI_AUI_MUX'`) bereitgestellt, was eine flexible Nutzung in verschiedenen Kontexten ermöglicht.
- **Adapterbasiert:** Die Kommunikation erfolgt ausschließlich über Adapter, wodurch eine klare und lose Kopplung ermöglicht wird.
- **Unidirektionale Adapter:** Sowohl die Eingangs- als auch die Ausgangsadapter sind unidirektional, d.h. es werden nur Daten vom Socket zum Plug übertragen.
- **Änderungserkennung:** Der Ausgang wird nur bei tatsächlicher Wertänderung aktualisiert, was die Ereignislast im System reduziert.
- **Sechs Eingänge:** Bietet die Auswahl aus sechs unterschiedlichen Datenquellen, wobei der Indexbereich 0–5 abgedeckt wird.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, dennoch lässt sich das Verhalten in folgende Phasen strukturieren:

1. **Warten auf Indexänderung:** Der FB verharrt in einem Ruhezustand, bis der Wert am Adapter `K` einen neuen gültigen Index (0–5) annimmt.
2. **Auswahl und Prüfung:** Der neue Index wird verarbeitet, der entsprechende Eingang wird an den Ausgang gekoppelt. Der FB vergleicht den neuen Wert mit dem zuletzt gesendeten Ausgangswert.

Anwendungsszenarien

- **Multiplexen analoger Messwerte:** Auswahl eines von sechs Sensoren (z.B. Temperatur, Druck, Feuchte) für die Weiterverarbeitung in einem Automatisierungssystem.
- **Kanalumschaltung:** Dynamische Umschaltung zwischen mehreren Datenquellen (z.B. in der Agrartechnik für die Auswahl verschiedener Betriebsmodi).
- **Reduzierung der Ereignislast:** Einsatz in Systemen, in denen unnötige Aktualisierungen vermieden werden sollen, da der Baustein nur bei echten Wertänderungen Ereignisse erzeugt.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu klassischen Multiplexern, die direkt über Dateneingänge arbeiten, nutzt dieser FB ausschließlich Adapter, was die Einbindung in adapterbasierte Architekturen erleichtert. Die Eigenschaft, nur bei Wertänderungen ein Ereignis auszulösen, unterscheidet ihn von einfachen Multiplexern, die bei jeder Indexänderung ein Ereignis senden. Dadurch wird die Kommunikation effizienter, insbesondere bei häufig wechselnden Indizes, aber selten wechselnden Datenwerten.

Fazit

Der **AUDI_AUI_MUX_6** stellt einen flexiblen und effizienten Multiplexer für adapterbasierte Systeme dar. Durch seine generische Struktur und die intelligente Triggerung nur bei tatsächlichen Änderungen ist er ideal für Anwendungen, in denen mehrere Datenquellen verwaltet und unnötige Ereignisflut vermieden werden muss. Die klare Trennung über Adapter erleichtert die Wiederverwendung und Integration in bestehende 4diac-Architekturen.

AUDI_AUI_MUX_7

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AUDI_AUI_MUX_7** realisiert einen generischen Multiplexer, der über einen Index (K) einen von sieben Eingängen (IN1 bis IN7) auf einen Ausgang (OUT) durchschaltet. Der Baustein ist speziell für die Verwendung mit den Adaptertypen `adapter::types::unidirectional::AUDI` und `adapter::types::unidirectional::AUI` ausgelegt und arbeitet rein ereignisgesteuert. Der Ausgang wird dabei nur dann aktualisiert, wenn sich der tatsächliche Wert am ausgewählten Eingang ändert.

Schnittstellenstruktur

Ereignis-Eingänge

Keine – der Baustein besitzt keine Ereignis-Eingänge.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge.

Daten-Eingänge

Keine direkten Dateneingänge; die Eingangsdaten werden ausschließlich über die Adapter-Schnittstellen bereitgestellt.

Daten-Ausgänge

Keine direkten Datenausgänge; die Ausgangsdaten werden ausschließlich über die Adapter-Schnittstelle bereitgestellt.

Adapter

Richtung	Name	Typ	Kommentar
Ausgang (Plug)	`OUT`	`adapter::types::unidirectional::AUDI`	Ausgangswert, der dem ausgewählten Eingang entspricht.
Eingang (Socket)	`K`	`adapter::types::unidirectional::AUI`	Indexauswahl (0 bis 6) für die Eingangswahl.
Eingang (Socket)	`IN1`	`adapter::types::unidirectional::AUDI`	Eingangswert 1, ausgewählt bei K = 0.
Eingang (Socket)	`IN2`	`adapter::types::unidirectional::AUDI`	Eingangswert 2, ausgewählt bei K = 1.
Eingang (Socket)	`IN3`	`adapter::types::unidirectional::AUDI`	Eingangswert 3, ausgewählt bei K = 2.
Eingang (Socket)	`IN4`	`adapter::types::unidirectional::AUDI`	Eingangswert 4, ausgewählt bei K = 3.
Eingang (Socket)	`IN5`	`adapter::types::unidirectional::AUDI`	Eingangswert 5, ausgewählt bei K = 4.

Eingang (Socket)	IN6	adapter::types::unidirectional::AUDI	Eingangswert 6, ausgewählt bei K = 5.
Eingang (Socket)	IN7	adapter::types::unidirectional::AUDI	Eingangswert 7, ausgewählt bei K = 6.

Funktionsweise

Der Baustein arbeitet als Multiplexer mit sieben Datenquellen und einer Indexsteuerung. Der Wert am Adapter `K` wird als Index interpretiert. Abhängig vom Wert (0 für `IN1`, 1 für `IN2`, ... bis 6 für `IN7`) wird der korrespondierende Eingang ausgewählt. Der aktuelle Wert des ausgewählten Eingangs wird auf den Ausgang `OUT` übertragen.

Die Besonderheit liegt in der Ereignisbehandlung: Der Ausgang wird nur dann aktualisiert, wenn sich der Wert am Ausgang tatsächlich ändert. Dies reduziert unnötige Ereignisflüsse und verbessert die Effizienz in der Datenweiterleitung.

Der Baustein ist als generischer Funktionsbaustein (Generic FB) deklariert (`eclipse4diac::core::GenericClassName = 'GEN_AUDI_AUI_MUX'`). Dadurch kann er in unterschiedlichen Kontexten wiederverwendet werden, ohne die internen Typen fest zu verdrahten.

Technische Besonderheiten

- **Generischer Typ:** Der Baustein ist als generischer FB implementiert, was eine flexible Anpassung an verschiedene Datentypen ermöglicht (hier speziell die Adaptertypen `AUDI` und `AUI`).
- **Ereignisreduzierung:** Der Ausgang wird nur bei einer Wertänderung aktualisiert.
- **Adapterbasierte Kommunikation:** Alle Ein- und Ausgänge sind über unidirektionale Adapter (`AdapterDeclaration`) realisiert, was eine lose Kopplung und einfache Integration in bestehende Adaptertopologien ermöglicht.
- **Kein Zustandsautomat:** Es ist kein ECC (Execution Control Chart) definiert; die Verarbeitung erfolgt rein ereignisorientiert ohne interne Zustände.

Zustandsübersicht

Da kein ECC vorhanden ist, existiert keine explizite Zustandsmaschine. Der Baustein führt bei Aktivierung die Multiplexfunktion deterministisch aus.

Anwendungsszenarien

- **Datenwegebündelung:** In Systemen, in denen mehrere Sensoren oder Datenquellen über einen einzigen Kanal ausgelesen werden sollen, kann der Multiplexer verwendet werden, um sequenziell verschiedene Eingänge zu selektieren.
- **Konfigurierbare Signalauswahl:** In Steuerungen mit wechselnden Signalquellen (z. B. Messumformer, Diagnosekanäle) wird über den Index `K` dynamisch die gewünschte Quelle ausgewählt.
- **Energieeffiziente Kommunikation:** Durch die ereignisbasierte Aktualisierung nur bei Wertänderungen eignet sich der Baustein für Applikationen mit begrenzter Bandbreite oder bei denen unnötige Datenübertragungen vermieden werden sollen.

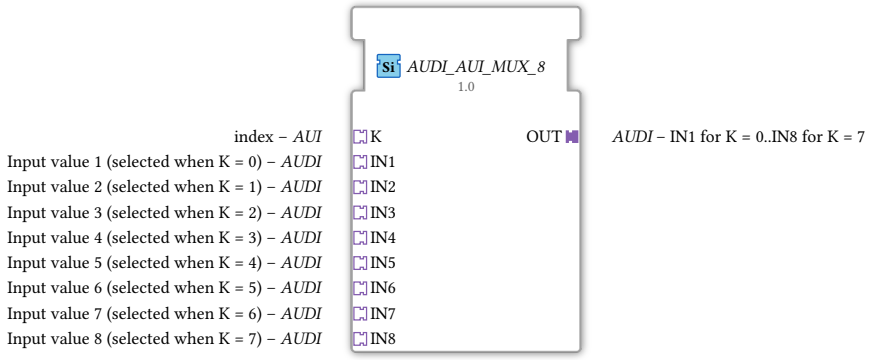
Vergleich mit ähnlichen Bausteinen

Typische Multiplexer in der 4diac-Umgebung verwenden oft direkte Dateneingänge und einen numerischen Index. Der `AUDI_AUI_MUX_7` zeichnet sich durch die Adapter-basierte Schnittstelle und die ereignisreduzierte Ausgabe aus. Gegenüber einem klassischen Datenmultiplexer (z. B. `SEL` oder `MUX`) bietet er eine höhere Flexibilität in der Datenmodellierung über Adapter und minimiert die Ereignislast durch die Wertänderungserkennung. Andere Bausteine erzeugen bei jedem Indexwechsel ein Ereignis, auch wenn sich der Datenwert nicht ändert – hier wird dies gezielt vermieden.

Fazit

Der **AUDI_AUI_MUX_7** ist ein leistungsfähiger, generischer Multiplexer für Adapter-basierte Kommunikation. Seine Fähigkeit, den Ausgang nur bei tatsächlichen Wertänderungen zu aktualisieren, macht ihn besonders für Anwendungen geeignet, bei denen Effizienz und Datenreduktion im Vordergrund stehen. Die klare Trennung von Eingängen, Ausgang und Indexsteuerung ermöglicht eine einfache Integration in bestehende Automatisierungsstrukturen und bietet eine flexible, wiederverwendbare Lösung für vielfältige Selektionsaufgaben.

AUDI_AUI_MUX_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AUDI_AUI_MUX_8` ist ein generischer Multiplexer, der über einen Adapter-Index `K` einen von acht Eingangswerten (`IN1` bis `IN8`) auswählt und diesen über den Adapter-Ausgang `OUT` bereitstellt. Der Baustein wurde speziell für die effiziente Weitergabe von Datenströmen entworfen und aktualisiert den Ausgang nur bei einer tatsächlichen Wertänderung. Dadurch werden unnötige Ereignisse vermieden und die Netzwerklast reduziert.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine vorhanden.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- **K** (Socket, Typ `AUI`): Index zur Auswahl des aktiven Eingangs. Gültige Werte sind 0 bis 7.
- **IN1** (Socket, Typ `AUDI`): Eingangswert 1, aktiv bei `K = 0`.
- **IN2** (Socket, Typ `AUDI`): Eingangswert 2, aktiv bei `K = 1`.
- **IN3** (Socket, Typ `AUDI`): Eingangswert 3, aktiv bei `K = 2`.
- **IN4** (Socket, Typ `AUDI`): Eingangswert 4, aktiv bei `K = 3`.
- **IN5** (Socket, Typ `AUDI`): Eingangswert 5, aktiv bei `K = 4`.
- **IN6** (Socket, Typ `AUDI`): Eingangswert 6, aktiv bei `K = 5`.
- **IN7** (Socket, Typ `AUDI`): Eingangswert 7, aktiv bei `K = 6`.
- **IN8** (Socket, Typ `AUDI`): Eingangswert 8, aktiv bei `K = 7`.

Daten-Ausgänge

- **OUT** (Plug, Typ `AUDI`): Gibt den Wert des aktuell ausgewählten Eingangs weiter.

Adapter

- **Ausgangsadapter (Plug):** `OUT`
- **Eingangsadapter (Sockets):** `K`, `IN1` – `IN8`

Funktionsweise

Der Baustein wartet auf Aktualisierungen der angeschlossenen Adapter. Sobald sich entweder der Index `K` oder der Wert des momentan ausgewählten Eingangs ändert, wird der neue Wert an den Ausgang `OUT` übertragen. Diese Logik stellt sicher, dass keine unnötigen Ereignisse generiert werden, wenn sich nicht relevante Eingänge ändern oder wenn sich der Index zwar ändert, aber der ausgewählte Wert identisch bleibt (z. B. wenn zwei Eingänge denselben Wert liefern). Der Baustein ist als generischer Funktionsblock definiert, wodurch er mit verschiedenen Datentypen wiederverwendet werden kann, die den jeweiligen Adapterdefinitionen entsprechen.

Technische Besonderheiten

- **Generische Implementierung:** Der FB verwendet das Attribut `GenericClassName = 'GEN_AUDI_AUI_MUX'`, was eine typlose, wiederverwendbare Ausführung ermöglicht.
- **Adapter-basierte Schnittstelle:** Alle Werte werden über Adapter übertragen, die eine lose Kopplung zwischen den Bausteinen in einem verteilten System ermöglichen.
- **Effiziente Aktualisierung:** Der Ausgang wird nur bei tatsächlicher Wertänderung aktualisiert, was die Netzwerkkommunikation reduziert.
- **Keine Ereignis-Eingänge:** Die Steuerung des Bausteins erfolgt rein über die Datenaktualisierung der Adapter; ein explizites Trigger-Ereignis ist nicht erforderlich.

Zustandsübersicht

Der FB besitzt keine extern sichtbaren Zustände im klassischen Sinne. Im internen Verhalten wird jedoch implizit der aktuelle Index `K` und der zuletzt ausgegebene Wert gespeichert, um eine Änderung zu erkennen. Diese internen Zustände sind:

- **Wartezustand:** Es wird auf eine Änderung des Index oder des ausgewählten Eingangswerts gewartet.
- **Aktualisierungszustand:** Bei Erkennung einer relevanten Änderung wird der Ausgangswert übernommen, danach wechselt der FB zurück in den Wartezustand.

Anwendungsszenarien

- **Datenauswahl in Messsystemen:** Auswahl eines von mehreren Sensoren (z. B. Temperatur, Druck) auf Grundlage eines Steuerindex.
- **Umschaltung von Kommunikationskanälen:** Flexibles Routing von Datenpaketen zwischen verschiedenen Quellen und einem Ziel.
- **Parametrierung:** Dynamisches Einspielen von Konfigurationswerten aus mehreren Quellen in eine Steuerung.
- **Redundanzverwaltung:** Umschalten auf redundante Eingänge bei Ausfall eines Sensors (durch externe Logik, die den Index anpasst).

Vergleich mit ähnlichen Bausteinen

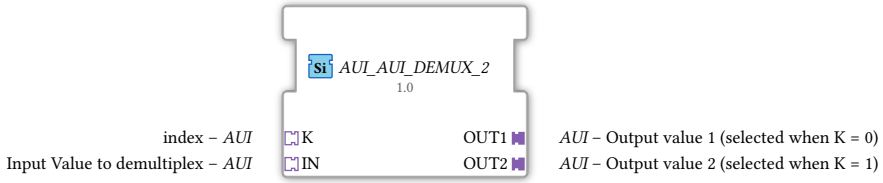
Gegenüber klassischen MUX-Bausteinen, die über separate Daten- und Ereignis-Eingänge verfügen, zeichnet sich dieser FB durch seine Adapter-Schnittstelle und die ereignisoptimierte Ausgabe aus. Konventionelle Multiplexer geben oft bei jeder Indexänderung ein Ereignis aus, unabhängig davon, ob sich der Wert wirklich ändert. `AUDI_AUI_MUX_8` eliminiert unnötige Events und ist dadurch für datengetriebene industrielle Anwendungen besser geeignet. Der Verzicht auf Ereignis-Eingänge vereinfacht die Einbindung in Systeme, in denen Datenwerte kontinuierlich aktualisiert werden.

Fazit

Der `AUDI_AUI_MUX_8` ist ein leistungsfähiger und flexibler Multiplexer-Baustein, der durch seine Adapterbasierte Architektur und die intelligente Ereignissteuerung besonders für moderne, verteilte Automatisierungssysteme geeignet ist. Er reduziert unnötige Kommunikation und ermöglicht eine klare, modulare Struktur bei der Auswahl und Weitergabe von Datenwerten. Mit seiner generischen Natur lässt er sich vielfältig in verschiedenen Anwendungskontexten einsetzen.

UINT

AUI_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AUI_AUI_DEMUX_2` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `UINT`. Anders als `AUI_DEMUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem

K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUI_AUI_DEMUX_2` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`UINT`, 16-Bit-Ganzzahl (vorzeichenlos)), der an einen der `n` Ausgänge weitergeleitet wird (AUI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AUI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AUI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Ausgangs-Adapter (`OUT1` … `OUT2`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUI_AUI_DEMUX_2 … AUI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Da der Nutzdatentyp dieses Bausteins selbst `UINT` ist, hat `AUI_AUI_DEMUX_2` denselben Adaptertyp (`AUI`) sowohl für die Nutzdaten (`IN`/`OUT`) als auch für den Index-Socket **K** – beide Rollen sind unabhängig voneinander verdrahtet, auch wenn sie denselben Adaptertyp verwenden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`UINT`, 16-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUI_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

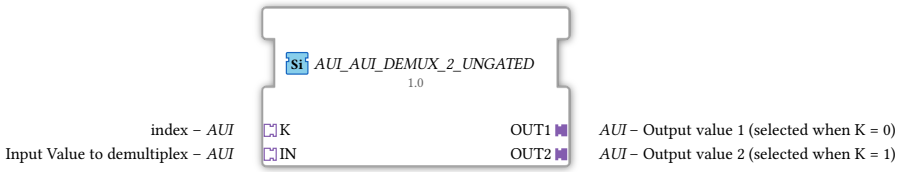
- **`AUI_AUI_DEMUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AUI_AUI_DEMUX_2` überträgt die Demultiplexer-Logik von `AUI_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUI_AUI_DEMUX_2_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `AUI_AUI_DEMUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AUI_AUI_DEMUX_2_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `UINT`. Anders als `AUI_DEMUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUI_AUI_DEMUX_2_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`UINT`, 16-Bit-Ganzzahl (vorzeichenlos)), der an einen der `n` Ausgänge weitergeleitet wird (AUI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AUI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AUI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Ausgangs-Adapter (`OUT1` … `OUT2`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUI_AUI_DEMUX_2_UNGATED … AUI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Da der Nutzdatentyp dieses Bausteins selbst `UINT` ist, hat `AUI_AUI_DEMUX_2_UNGATED` denselben Adaptertyp (`AUI`) sowohl für die Nutzdaten (`IN`/`OUT`) als auch für den Index-Socket **K** – beide Rollen sind unabhängig voneinander verdrahtet, auch wenn sie denselben Adaptertyp verwenden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`UINT`, 16-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUI_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

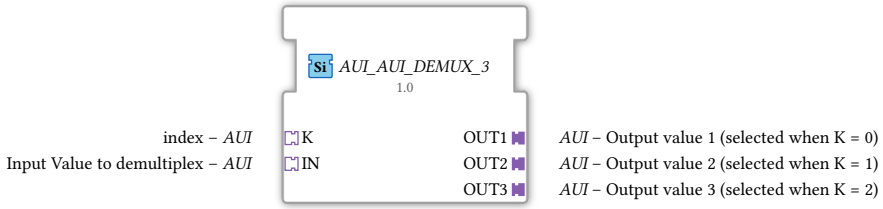
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **AUI_AUI_DEMUX_2**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der AUI_AUI_DEMUX_2_UNGATED überträgt die Demultiplexer-Logik von AUI_DEMUX_2 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUI_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AUI_AUI_DEMUX_3 ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp UINT. Anders als AUI_DEMUX_3 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AUI_AUI_DEMUX_3 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (UINT, 16-Bit-Ganzzahl (vorzeichenlos)), der an einen der n Ausgänge weitergeleitet wird (AUI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AUI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AUI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AUI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 3 Ausgangs-Adapter (OUT1 … OUT3) gerade ausgewählt ist.

2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AUI_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUI_AUI_DEMUX_2 … AUI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Da der Nutzdatentyp dieses Bausteins selbst UINT ist, hat AUI_AUI_DEMUX_3 denselben Adaptertyp (AUI) sowohl für die Nutzdaten (IN/OUT) als auch für den Index-Socket **K** – beide Rollen sind unabhängig voneinander verdrahtet, auch wenn sie denselben Adaptertyp verwenden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen IN-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (UINT, 16-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit AUI_DEMUX_3 (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

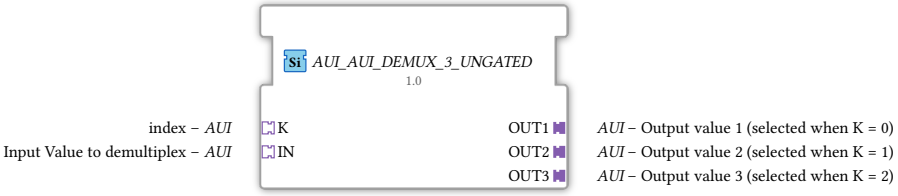
- **AUI_AUI_DEMUX_3_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der AUI_AUI_DEMUX_3 überträgt die Demultiplexer-Logik von AUI_DEMUX_3 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUI_AUI_DEMUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AUI_AUI_DEMUX_3. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AUI_AUI_DEMUX_3_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `UINT`. Anders als `AUI_DEMUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUI_AUI_DEMUX_3_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`UINT`, 16-Bit-Ganzzahl (vorzeichenlos)), der an einen der `n` Ausgänge weitergeleitet wird (AUI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AUI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AUI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AUI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Ausgangs-Adapter (`OUT1` … `OUT3`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.

- Generische Implementierung (`GEN_AUI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUI_AUI_DEMUX_2 … AUI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Da der Nutzdatentyp dieses Bausteins selbst `UINT` ist, hat `AUI_AUI_DEMUX_3_UNGATED` denselben Adaptertyp (`AUI`) sowohl für die Nutzdaten (`IN`/`OUT`) als auch für den Index-Socket **K** – beide Rollen sind unabhängig voneinander verdrahtet, auch wenn sie denselben Adaptertyp verwenden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`UINT`, 16-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUI_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

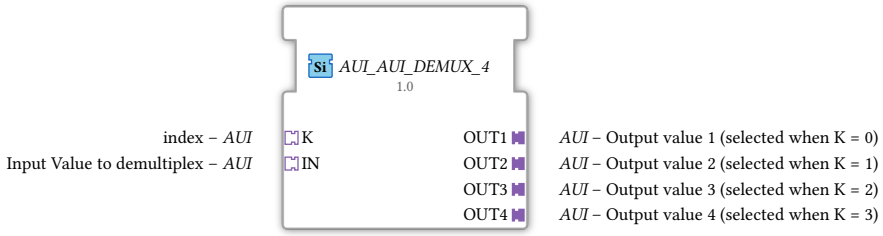
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AUI_AUI_DEMUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AUI_AUI_DEMUX_3_UNGATED` überträgt die Demultiplexer-Logik von `AUI_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUI_AUI_DEMUX_4



• – ○ • – ○ • – ○ •

Einleitung

Der `AUI_AUI_DEMUX_4` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `UINT`. Anders als `AUI_DEMUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUI_AUI_DEMUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`UINT`, 16-Bit-Ganzzahl (vorzeichenlos)), der an einen der `n` Ausgänge weitergeleitet wird (AUI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AUI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AUI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AUI-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AUI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUI_AUI_DEMUX_2 … AUI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Da der Nutzdatentyp dieses Bausteins selbst `UINT` ist, hat `AUI_AUI_DEMUX_4` denselben Adaptertyp (`AUI`) sowohl für die Nutzdaten (`IN`/`OUT`) als auch für den Index-Socket **K** – beide Rollen sind unabhängig voneinander verdrahtet, auch wenn sie denselben Adaptertyp verwenden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) → aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`UINT`, 16-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUI_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

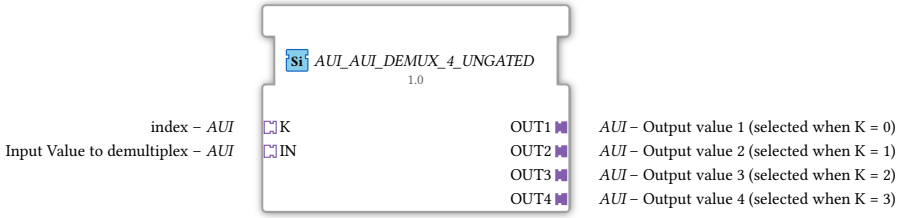
- **`AUI_AUI_DEMUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AUI_AUI_DEMUX_4` überträgt die Demultiplexer-Logik von `AUI_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUI_AUI_DEMUX_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUI_AUI_DEMUX_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AUI_AUI_DEMUX_4_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `UINT`. Anders als `AUI_DEMUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUI_AUI_DEMUX_4_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`UINT`, 16-Bit-Ganzzahl (vorzeichenlos)), der an einen der `n` Ausgänge weitergeleitet wird (AUI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AUI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AUI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AUI-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AUI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUI_AUI_DEMUX_2 … AUI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Da der Nutzdatentyp dieses Bausteins selbst `UINT` ist, hat `AUI_AUI_DEMUX_4_UNGATED` denselben Adaptertyp (`AUI`) sowohl für die Nutzdaten (`IN`/`OUT`) als auch für den Index-Socket **K** – beide Rollen sind unabhängig voneinander verdrahtet, auch wenn sie denselben Adaptertyp verwenden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`UINT`, 16-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUI_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

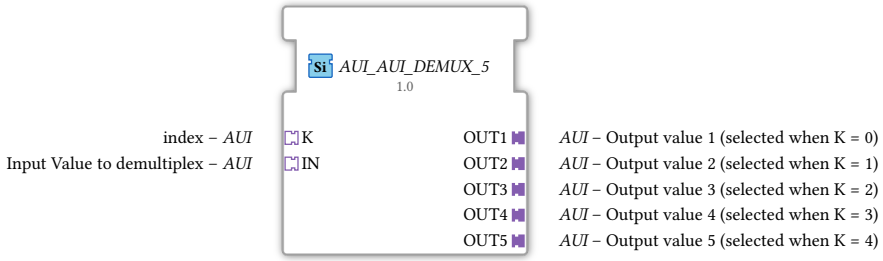
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AUI_AUI_DEMUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AUI_AUI_DEMUX_4_UNGATED` überträgt die Demultiplexer-Logik von `AUI_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUI_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AUI_AUI_DEMUX_5` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `UINT`. Anders als `AUI_DEMUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUI_AUI_DEMUX_5` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`UINT`, 16-Bit-Ganzzahl (vorzeichenlos)), der an einen der `n` Ausgänge weitergeleitet wird (AUI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AUI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AUI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AUI-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AUI-Adaptertyp).

- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AUI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Ausgangs-Adapter (`OUT1` … `OUT5`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUI_AUI_DEMUX_2 … AUI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Da der Nutzdatentyp dieses Bausteins selbst `UINT` ist, hat `AUI_AUI_DEMUX_5` denselben Adaptertyp (`AUI`) sowohl für die Nutzdaten (`IN`/`OUT`) als auch für den Index-Socket **K** – beide Rollen sind unabhängig voneinander verdrahtet, auch wenn sie denselben Adaptertyp verwenden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`UINT`, 16-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUI_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AUI_AUI_DEMUX_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

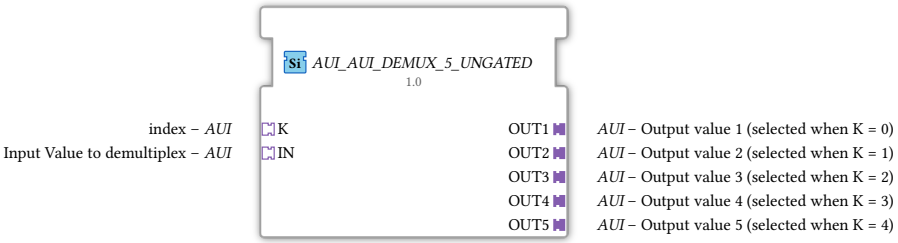
Fazit

Der `AUI_AUI_DEMUX_5` überträgt die Demultiplexer-Logik von `AUI_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUI_AUI_DEMUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUI_AUI_DEMUX_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos

weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AUI_AUI_DEMUX_5_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `UINT`. Anders als `AUI_DEMUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUI_AUI_DEMUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`UINT`, 16-Bit-Ganzzahl (vorzeichenlos)), der an einen der `n` Ausgänge weitergeleitet wird (AUI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AUI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AUI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AUI-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AUI-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AUI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Ausgangs-Adapter (`OUT1` … `OUT5`) gerade ausgewählt ist.

2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUI_AUI_DEMUX_2 … AUI_AUI_DEMUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Da der Nutzdatentyp dieses Bausteins selbst `UINT` ist, hat `AUI_AUI_DEMUX_5_UNGATED` denselben Adaptertyp (`AUI`) sowohl für die Nutzdaten (`IN`/`OUT`) als auch für den Index-Socket **K** – beide Rollen sind unabhängig voneinander verdrahtet, auch wenn sie denselben Adaptertyp verwenden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`UINT`, 16-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUI_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

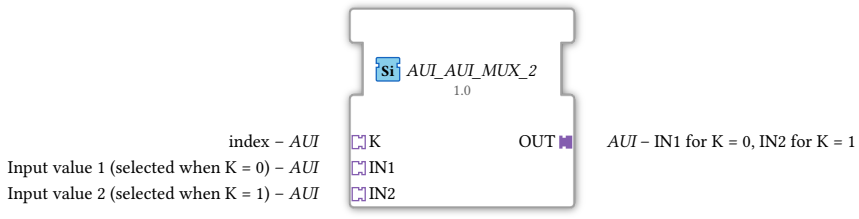
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AUI_AUI_DEMUX_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AUI_AUI_DEMUX_5_UNGATED` überträgt die Demultiplexer-Logik von `AUI_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUI_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AUI_AUI_MUX_2 ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp UINT. Anders als AUI_MUX_2 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AUI_AUI_MUX_2 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index K = 0 an OUT durchgereicht (AUI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index K = 1 an OUT durchgereicht (AUI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AUI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN2:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 2 Eingangs-Adapter (IN1 … IN2) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf OUT gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AUI_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUI_AUI_MUX_2 … AUI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Da der Nutzdatentyp dieses Bausteins selbst UINT ist, hat AUI_AUI_MUX_2 denselben Adaptertyp (AUI) sowohl für die Nutzdaten (IN/OUT) als auch für den Index-Socket **K** – beide Rollen sind unabhängig voneinander verdrahtet, auch wenn sie denselben Adaptertyp verwenden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (16-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUI_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

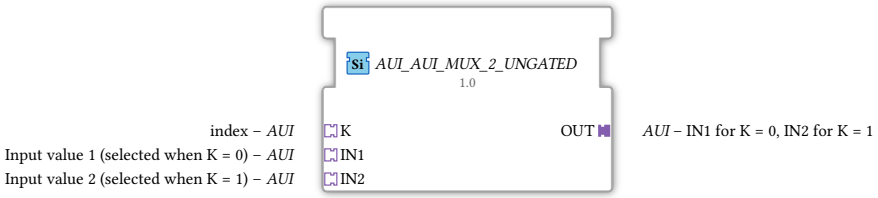
- **`AUI_AUI_MUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AUI_AUI_MUX_2` überträgt die Multiplexer-Logik von `AUI_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUI_AUI_MUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUI_AUI_MUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `AUI_AUI_MUX_2_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `UINT`. Anders als `AUI_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem

Datenwert D1 sendet, wertet AUI_AUI_MUX_2_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index K = 0 an OUT durchgereicht (AUI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index K = 1 an OUT durchgereicht (AUI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AUI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN2:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 2 Eingangs-Adapter (IN1 … IN2) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf OUT gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AUI_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUI_AUI_MUX_2_UNGATED … AUI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Da der Nutzdatentyp dieses Bausteins selbst UINT ist, hat AUI_AUI_MUX_2_UNGATED denselben Adaptertyp (AUI) sowohl für die Nutzdaten (IN/OUT) als auch für den Index-Socket **K** – beide Rollen sind unabhängig voneinander verdrahtet, auch wenn sie denselben Adaptertyp verwenden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN1…IN2) ⟶ aktuell selektierten Eingang auslesen, OUT bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (16-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUI_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

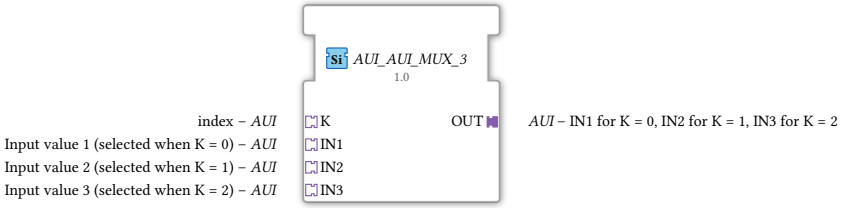
Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AUI_AUI_MUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AUI_AUI_MUX_2_UNGATED` überträgt die Multiplexer-Logik von `AUI_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUI_AUI_MUX_3



• – ○ • – ○ • – ○ •

Einleitung

Der `AUI_AUI_MUX_3` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `UINT`. Anders als `AUI_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUI_AUI_MUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (AUI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (AUI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (AUI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AUI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`...`IN3`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Eingangs-Adapter (`IN1` ... `IN3`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUI_AUI_MUX_2 ... AUI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Da der Nutzdatentyp dieses Bausteins selbst `UINT` ist, hat `AUI_AUI_MUX_3` denselben Adaptertyp (`AUI`) sowohl für die Nutzdaten (`IN`/`OUT`) als auch für den Index-Socket **K** – beide Rollen sind unabhängig voneinander verdrahtet, auch wenn sie denselben Adaptertyp verwenden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`...`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (16-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUI_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AUI_AUI_MUX_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

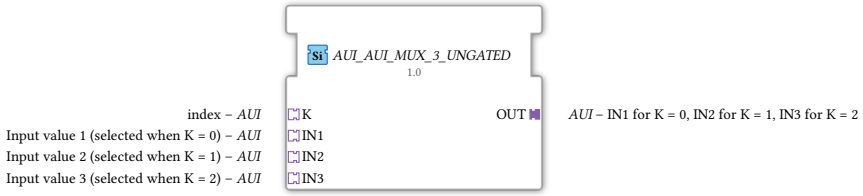
Fazit

Der `AUI_AUI_MUX_3` überträgt die Multiplexer-Logik von `AUI_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUI_AUI_MUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUI_AUI_MUX_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos

weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AUI_AUI_MUX_3_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `UINT`. Anders als `AUI_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUI_AUI_MUX_3_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (AUI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (AUI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (AUI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AUI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN3`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Eingangs-Adapter (`IN1` … `IN3`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUI_AUI_MUX_2 … AUI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Da der Nutzdatentyp dieses Bausteins selbst `UINT` ist, hat `AUI_AUI_MUX_3_UNGATED` denselben Adaptertyp (`AUI`) sowohl für die Nutzdaten (`IN`/`OUT`) als auch für den Index-Socket **K** – beide Rollen sind unabhängig voneinander verdrahtet, auch wenn sie denselben Adaptertyp verwenden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (16-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUI_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AUI_AUI_MUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AUI_AUI_MUX_3_UNGATED` überträgt die Multiplexer-Logik von `AUI_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUI_AUI_MUX_4

• – ○ • – ○ • – ○ •

Einleitung

Der `AUI_AUI_MUX_4` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `UINT`. Anders als `AUI_MUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUI_AUI_MUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index `K = 0` an `OUT` durchgereicht (AUI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index `K = 1` an `OUT` durchgereicht (AUI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index `K = 2` an `OUT` durchgereicht (AUI-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index `K = 3` an `OUT` durchgereicht (AUI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AUI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN4`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Eingangs-Adapter (`IN1` … `IN4`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUI_AUI_MUX_2 … AUI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Da der Nutzdatentyp dieses Bausteins selbst `UINT` ist, hat `AUI_AUI_MUX_4` denselben Adaptertyp (`AUI`) sowohl für die Nutzdaten (`IN`/`OUT`) als auch für den Index-Socket **K** – beide Rollen sind unabhängig voneinander verdrahtet, auch wenn sie denselben Adaptertyp verwenden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (16-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUI_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

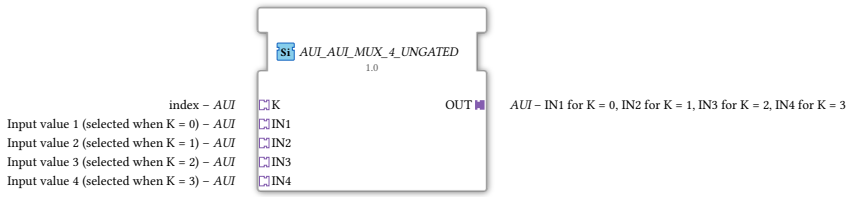
- **`AUI_AUI_MUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AUI_AUI_MUX_4` überträgt die Multiplexer-Logik von `AUI_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUI_AUI_MUX_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUI_AUI_MUX_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `AUI_AUI_MUX_4_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `UINT`. Anders als `AUI_MUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUI_AUI_MUX_4_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index `K = 0` an `OUT` durchgereicht (AUI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index `K = 1` an `OUT` durchgereicht (AUI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index `K = 2` an `OUT` durchgereicht (AUI-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index `K = 3` an `OUT` durchgereicht (AUI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AUI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN4`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Eingangs-Adapter (`IN1` … `IN4`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUI_AUI_MUX_2 … AUI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Da der Nutzdatentyp dieses Bausteins selbst `UINT` ist, hat `AUI_AUI_MUX_4_UNGATED` denselben Adaptertyp (`AUI`) sowohl für die Nutzdaten (`IN`/`OUT`) als auch für den Index-Socket **K** – beide Rollen sind unabhängig voneinander verdrahtet, auch wenn sie denselben Adaptertyp verwenden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (16-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUI_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **AUI_AUI_MUX_4**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der AUI_AUI_MUX_4_UNGATED überträgt die Multiplexer-Logik von AUI_MUX_4 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUI_AUI_MUX_5

• – ○ • – ○ • – ○ •

Einleitung

Der AUI_AUI_MUX_5 ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp UINT. Anders als AUI_MUX_5 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AUI_AUI_MUX_5 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index K = 0 an OUT durchgereicht (AUI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index K = 1 an OUT durchgereicht (AUI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index K = 2 an OUT durchgereicht (AUI-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index K = 3 an OUT durchgereicht (AUI-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index K = 4 an OUT durchgereicht (AUI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AUI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN5:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Eingangs-Adapter (`IN1` … `IN5`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUI_AUI_MUX_2 … AUI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Da der Nutzdatentyp dieses Bausteins selbst `UINT` ist, hat `AUI_AUI_MUX_5` denselben Adaptertyp (`AUI`) sowohl für die Nutzdaten (`IN`/`OUT`) als auch für den Index-Socket **K** – beide Rollen sind unabhängig voneinander verdrahtet, auch wenn sie denselben Adaptertyp verwenden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (16-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUI_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

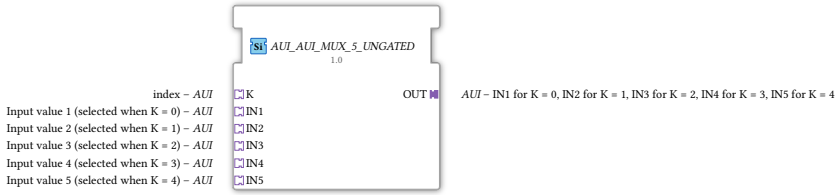
- `AUI_AUI_MUX_5_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AUI_AUI_MUX_5` überträgt die Multiplexer-Logik von `AUI_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUI_AUI_MUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUI_AUI_MUX_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• — ○ • — ○ • — ○ •

Einleitung

Der `AUI_AUI_MUX_5_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `UINT`. Anders als `AUI_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUI_AUI_MUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index `K = 0` an `OUT` durchgereicht (AUI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index `K = 1` an `OUT` durchgereicht (AUI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index `K = 2` an `OUT` durchgereicht (AUI-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index `K = 3` an `OUT` durchgereicht (AUI-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index `K = 4` an `OUT` durchgereicht (AUI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AUI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN5`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Eingangs-Adapter (`IN1` … `IN5`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUI_AUI_MUX_2 … AUI_AUI_MUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.
- Da der Nutzdatentyp dieses Bausteins selbst `UINT` ist, hat `AUI_AUI_MUX_5_UNGATED` denselben Adaptertyp (`AUI`) sowohl für die Nutzdaten (`IN`/`OUT`) als auch für den Index-Socket **K** – beide Rollen sind unabhängig voneinander verdrahtet, auch wenn sie denselben Adaptertyp verwenden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (16-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUI_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

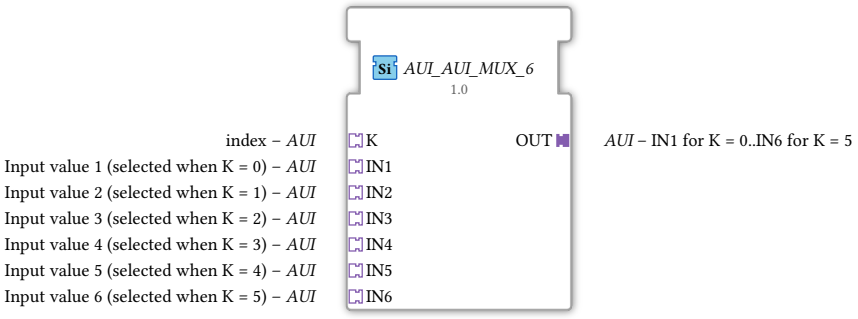
Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

- **`AUI_AUI_MUX_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AUI_AUI_MUX_5_UNGATED` überträgt die Multiplexer-Logik von `AUI_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUI_AUI_MUX_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AUI_AUI_MUX_6` ist ein generischer Multiplexer, der über sechs unidirektionale AUI-Adapter (Application User Interface) Eingangswerte entgegennimmt und einen davon – basierend auf einem über einen weiteren AUI-Adapter bereitgestellten Index – an seinen Ausgang weiterleitet. Der Baustein ist speziell für die Auswahl eines von sechs Kanälen konzipiert und arbeitet ereignisgesteuert:

Das Ausgangsereignis wird nur dann ausgelöst, wenn sich der am Ausgang anliegende Wert tatsächlich ändert. Dadurch werden unnötige Übertragungen und Prozesslast vermieden. Der FB ist als generischer Baustein mit dem Klassennamen `GEN_AUI_AUI_MUX` definiert und wird über Plug & Socket-Verbindungen in 4diac-IDE integriert.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Der FB verwendet ausschließlich unidirektionale AUI-Adapter (Typ `adapter::types::unidirectional::AUI`) für den Datenaustausch.

Ausgang (Plug)

Name	Typ (Adapter)	Kommentar
`OUT`	`unidirectional::AUI`	Ausgangswert, entspricht dem ausgewählten Eingang (IN1 – IN6).

Eingänge (Sockets)

Name	Typ (Adapter)	Kommentar
`K`	`unidirectional::AUI`	Index-Steuerung: wählt den aktiven Eingang (Wert 0–5).
`IN1`	`unidirectional::AUI`	Eingang 1 – aktiv bei K = 0
`IN2`	`unidirectional::AUI`	Eingang 2 – aktiv bei K = 1
`IN3`	`unidirectional::AUI`	Eingang 3 – aktiv bei K = 2
`IN4`	`unidirectional::AUI`	Eingang 4 – aktiv bei K = 3
`IN5`	`unidirectional::AUI`	Eingang 5 – aktiv bei K = 4
`IN6`	`unidirectional::AUI`	Eingang 6 – aktiv bei K = 5

Funktionsweise

Der Baustein `AUI_AUI_MUX_6` implementiert eine 6-zu-1-Auswahl (Multiplexer) auf Basis von AUI-Adaptern. Der über den Adapter `K` übertragene Indexwert bestimmt, welcher der sechs Eingangsadapter (`IN1` bis `IN6`) aktuell durchgeschaltet wird. Der Wert des ausgewählten Eingangs wird permanent am Ausgangsadapter `OUT` bereitgestellt.

Wesentlich ist die ereignisgesteuerte Aktualisierung: Der Ausgang `OUT` wird nur dann mit einem neuen Wert versehen, wenn sich der Wert des gewählten Eingangs gegenüber dem zuletzt gesendeten Ausgangswert geändert hat. Dies gilt sowohl bei einem Wechsel des Index (K) als auch bei einer Änderung des aktiven Eingangswerts. Bleibt der Wert unverändert, wird kein Ereignis erzeugt.

Die Kommunikation erfolgt asynchron und unidirektional über die jeweiligen Adapter-Schnittstellen. Der Baustein besitzt keine eigenen Daten- oder Ereignis-Ein-/Ausgänge, sondern ausschließlich Adapter, die eine lose Kopplung und Wiederverwendbarkeit ermöglichen.

Technische Besonderheiten

- **Generischer Aufbau**: Der Baustein ist als generischer FB mit dem Klassennamen `GEN_AUI_AUI_MUX` definiert. Dadurch kann er parametrisiert und in unterschiedlichen Kontexten eingesetzt werden.

- **Nur Adapter-Schnittstelle**: Sämtliche Ein- und Ausgänge sind als unidirektionale AUI-Adapter ausgeführt. Dies erlaubt eine flexible Verbindung mit anderen AUI-basierten Bausteinen und eine modulare Systemarchitektur.
- **Ereignis-Minimierung**: Durch die interne Erkennung von Wertänderungen wird der Ausgang nur bei tatsächlichen Änderungen aktualisiert. Dies reduziert die Netzwerk- und Prozessorlast in verteilten Systemen.
- **Keine Zustandsautomaten**: Die Logik ist rein funktional; es existiert kein explizit modellierter Zustandsautomat. Die Auswahl und Aktualisierung erfolgt vollständig datengetrieben.
- **Lizenzierung**: Der Baustein ist unter der Eclipse Public License 2.0 veröffentlicht.

Zustandsübersicht

Da der Baustein keinen internen Zustandsautomaten besitzt, entfällt eine klassische Zustandsdarstellung. Die Funktionsweise lässt sich vielmehr durch folgende logische Abläufe beschreiben:

1. **Initialisierung**: Beim ersten Durchlauf wird der Eingang `IN1` als Ausgang gesetzt, falls keine gültige Indexvorgabe vorliegt.
2. **Indexauswahl**: Durch Änderung des Adapters `K` wird der entsprechende Eingang aktiviert.
3. **Keine Änderung**: Liegt keine Änderung vor, bleibt der Ausgang unverändert und es wird kein Ereignis ausgelöst.

Somit existiert nur der funktionale Zustand „aktiv“ ohne spezifische Unterzustände.

Anwendungsszenarien

- **Kanalumschaltung in Automatisierungssystemen**: Auswahl eines von sechs analogen oder digitalen Messwerten zur Weiterverarbeitung.
- **Redundante Datenquellen**: Umschaltung auf einen Ersatzeingang bei Ausfall des Primärsignals.
- **Konfigurierbare Datenweiterleitung**: In modularen Anlagen kann über den Indexadapter dynamisch festgelegt werden, welcher Datenstrom an den Ausgang gelangt.
- **Ereignisbasierte Kommunikation**: Aufgrund der Änderungserkennung eignet sich der FB besonders für Netzwerke mit begrenzter Bandbreite, da nur relevante Änderungen übertragen werden.

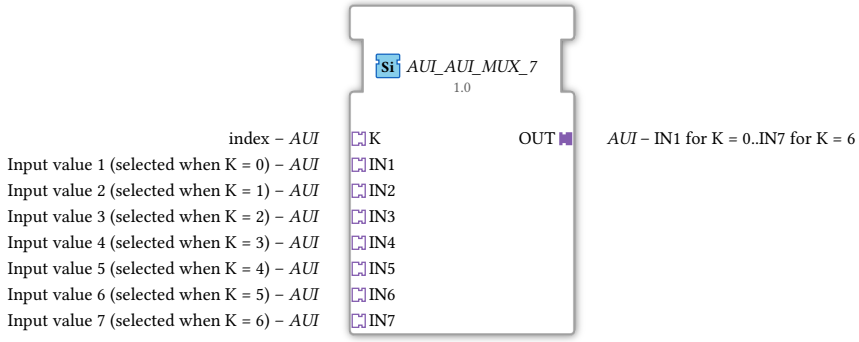
Vergleich mit ähnlichen Bausteinen

Gegenüber klassischen Multiplexern, die über separate Daten-Eingänge (z. B. INT, REAL) und einen Auswahl-Datenwert verfügen, zeichnet sich `AUI_AUI_MUX_6` durch die Verwendung von Adapter-Schnittstellen aus. Adapter ermöglichen eine semantische Bündelung von Daten und Ereignissen und vereinfachen die Verbindung zu komplexen Komponenten. Im Vergleich zu einer direkten Implementierung mit `MUX`-Funktionsbausteinen (z. B. nach IEC 61499) bietet dieser FB eine höhere Abstraktionsebene und eine genauere Anpassung an AUI-basierte Architekturen. Andere Multiplexer verfügen oft über feste Eingangsanzahlen oder erfordern zusätzliche Ereignis-Steuerung; hier erfolgt die Auswahl rein über den Indexadapter ohne separate Enable-Signale.

Fazit

Der `AUI_AUI_MUX_6` ist ein flexibler und effizienter Multiplexer für die Auswahl aus sechs AUI-Datenquellen. Durch die vollständige Adapter-Schnittstelle und die ausschließlich ereignisgesteuerte Aktualisierung bei Wertänderungen bietet er eine optimale Lösung für verteilte Automatisierungssysteme, die Wert auf geringe Netzlast und modulare Wiederverwendbarkeit legen. Die generische Implementierung und die klare Definition als 6-zu-1-Auswahl machen ihn zu einem nützlichen Baustein in vielfältigen Anwendungsumgebungen.

AUI_AUI_MUX_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AUI_AUI_MUX_7** ist ein Multiplexer, der es ermöglicht, einen von sieben Eingangsadaptern (IN1 bis IN7) auf einen Ausgangsadapter (OUT) zu schalten. Die Auswahl des aktiven Eingangs erfolgt über den Indexadapter **K**. Der Baustein ist als generischer Typ definiert und wird für die Auswahl von AUI-Datenströmen verwendet. Der Adapter-Ausgang wird nur bei einer tatsächlichen Wertänderung aktualisiert.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge.

Daten-Eingänge

Es sind keine direkten Dateneingänge vorhanden; die Eingänge erfolgen über die Adapter.

Daten-Ausgänge

Es sind keine direkten Datenausgänge vorhanden; der Ausgang wird über den Adapter OUT realisiert.

Adapter

Sockets (Eingangsadapter):

Name	Typ	Beschreibung
K	AUI (unidirectional)	Index zur Auswahl eines der Eingänge IN1..IN7 (0..6)
IN1	AUI (unidirectional)	Eingangswert 1, ausgewählt bei K = 0
IN2	AUI (unidirectional)	Eingangswert 2, ausgewählt bei K = 1
IN3	AUI (unidirectional)	Eingangswert 3, ausgewählt bei K = 2
IN4	AUI (unidirectional)	Eingangswert 4, ausgewählt bei K = 3
IN5	AUI (unidirectional)	Eingangswert 5, ausgewählt bei K = 4
IN6	AUI (unidirectional)	Eingangswert 6, ausgewählt bei K = 5
IN7	AUI (unidirectional)	Eingangswert 7, ausgewählt bei K = 6

Plugs (Ausgangsadapter):

Name	Typ	Beschreibung
OUT	AUI (unidirectional)	Ausgang, der den aktuell ausgewählten Eingangswert liefert

Funktionsweise

Der Baustein überwacht den Indexadapter **K** sowie alle Eingangsadapter. Sobald sich der Wert von **K** oder der Wert des aktuell ausgewählten Eingangs (IN1..IN7) ändert, wird der Ausgang **OUT** aktualisiert. Die Auswahl erfolgt gemäß dem Index **K** mit folgender Zuordnung:

- K = 0 ⟶ IN1
- K = 1 ⟶ IN2
- K = 2 ⟶ IN3
- K = 3 ⟶ IN4
- K = 4 ⟶ IN5
- K = 5 ⟶ IN6
- K = 6 ⟶ IN7

Wenn **K** außerhalb des gültigen Bereichs liegt (größer als 6), wird kein Eingang ausgewählt und der Ausgang bleibt unverändert. Die Aktualisierung des Ausgangs erfolgt nur bei tatsächlicher Wertänderung, was unnötige Ereignisse vermeidet.

Technische Besonderheiten

- Der Baustein ist als **generischer Typ** (**GenericClassName** = `GEN_AUI_AUI_MUX`) implementiert, wodurch er für andere Anzahlen von Eingängen erweiterbar ist.
- Der Baustein verwendet ausschließlich Adapter für die Ein-/Ausgabe (unidirektionaler AUI-Typ). Dadurch werden die datenflussorientierten Verbindungen vereinfacht.
- Die Erkennung von Wertänderungen basiert auf einem Vergleich der aktuellen und vorherigen Werte; nur bei Änderung wird **OUT** aktualisiert.
- Es gibt keine ereignisgesteuerten Eingänge; die gesamte Logik wird durch Datenänderungen an den Adaptern getriggert.
- Der Baustein wurde für den Einsatz in der Agrartechnik entwickelt und unter der Eclipse Public License 2.0 veröffentlicht.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten. Die Funktionsweise ist rein reaktiv: Bei jeder Änderung an **K** oder dem ausgewählten Eingang wird der Ausgang neu berechnet und aktualisiert, falls sich der Ausgangswert tatsächlich geändert hat. Es gibt keine internen Zustände außer den aktuell gespeicherten Werten.

Anwendungsszenarien

- **Datenquellenumschaltung**: Der Baustein kann verwendet werden, um zwischen mehreren Sensoren oder Datenquellen umzuschalten, die über AUI-Schnittstellen angebunden sind.
- **Redundanzsteuerung**: In Systemen mit redundanten Pfaden kann der Multiplexer den aktiven Pfad basierend auf einem Steuersignal wählen.
- **Flexible Konfiguration**: Durch die Adapterstruktur lassen sich verschiedene AUI-basierte Komponenten dynamisch auswählen.

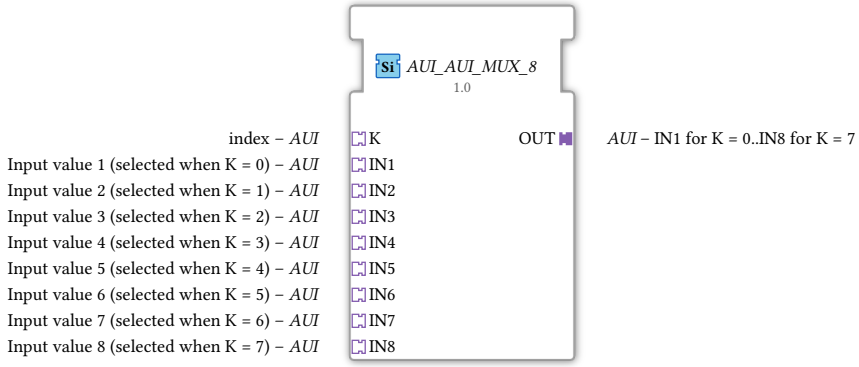
Vergleich mit ähnlichen Bausteinen

Andere Multiplexer-Bausteine bieten oft einzelne Dateneingänge (z.B. INTEGER) anstelle von Adaptern. Der Vorteil dieses Bausteins liegt in der Verwendung von Adaptern, die komplexe Datentypen und ereignisbasierte Kommunikation integrieren können. Nachteil könnte die fixe Anzahl der Eingänge sein (hier 7). Im Vergleich zu einem generischen Multiplexer, der über Parameter konfigurierbar ist, ist dieser Baustein spezialisiert, aber durch die Generik erweiterbar.

Fazit

Der **AUI_AUI_MUX_7** ist ein spezialisierter Multiplexer für AUI-basierte Verbindungen. Er ermöglicht eine saubere und effiziente Auswahl von bis zu sieben Eingangsadaptern auf einen Ausgang. Durch die ausschließliche Verwendung von Adaptern und die Ereignisausgabe bei Wertänderungen eignet er sich gut für Echtzeit- und Automatisierungssysteme, insbesondere in der Agrartechnik. Die generische Implementierung erlaubt eine einfache Anpassung an andere Eingangsanzahlen.

AUI_AUI_MUX_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AUI_AUI_MUX_8** ist ein generischer Multiplexer auf Basis von AUI-Adaptern. Er wählt aus acht Eingangsadaptern (IN1 bis IN8) denjenigen aus, dessen Index über den Adapter **K** bestimmt wird, und stellt den Wert über den Ausgangsadapter **OUT** bereit. Dabei wird der Ausgangswert nur bei tatsächlicher Änderung aktualisiert.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

Keine (die Eingabewerte werden ausschließlich über Adapter bereitgestellt).

Daten-Ausgänge

Keine (der Ausgabewert wird ausschließlich über einen Adapter bereitgestellt).

Adapter

Typ	Name	Richtung	Kommentar
Plug	OUT	Ausgang	Ausgewählter AUI-Wert (0..7 je nach Index K)
Socket	K	Eingang	Index zur Auswahl des Eingangs (Wert 0..7)
Socket	IN1	Eingang	Eingabewert 1, wird bei K = 0 ausgewählt
Socket	IN2	Eingang	Eingabewert 2, wird bei K = 1 ausgewählt
Socket	IN3	Eingang	Eingabewert 3, wird bei K = 2 ausgewählt
Socket	IN4	Eingang	Eingabewert 4, wird bei K = 3 ausgewählt
Socket	IN5	Eingang	Eingabewert 5, wird bei K = 4 ausgewählt
Socket	IN6	Eingang	Eingabewert 6, wird bei K = 5 ausgewählt
Socket	IN7	Eingang	Eingabewert 7, wird bei K = 6 ausgewählt
Socket	IN8	Eingang	Eingabewert 8, wird bei K = 7 ausgewählt

Alle Adapter sind vom Typ `adapter::types::unidirectional::AUI` und arbeiten unidirektional.

Funktionsweise

Der Funktionsblock überwacht den Indexadapter **K** sowie alle acht Eingangsadapter **IN1** bis **IN8**. Sobald sich der Wert von **K** ändert, wird der entsprechende Eingang ausgewählt. Der Wert dieses Eingangs wird direkt auf den Ausgangsadapter **OUT** übertragen.
Eine Aktualisierung des Ausgangs erfolgt jedoch nur dann, wenn sich der Wert des ausgewählten Eingangs gegenüber dem zuletzt ausgegebenen Wert tatsächlich geändert hat. Dadurch werden unnötige Ausgangsaktivierungen vermieden und nachgeschaltete Bausteine werden nur bei echten Wertänderungen benachrichtigt.

Technische Besonderheiten

- **Generischer Funktionsblock** – Der Baustein ist als generischer Typ mit dem Klassennamen `GEN_AUI_AUI_MUX` definiert und kann für verschiedene Anwendungen instanziiert werden.
- **Reine Adapter-basierte Kommunikation** – Es gibt keine direkten Daten- oder eventbasierten Ein-/Ausgänge; die gesamte Datenübertragung erfolgt über AUI-Adapter.
- **Änderungsdetektion** – Der Ausgang wird nur bei einer tatsächlichen Wertänderung des ausgewählten Eingangs aktualisiert.
- **Unidirektionale Adapter** – Alle AUI-Adapter sind unidirektional, d.h. die Daten fließen nur in eine Richtung (Sockets liefern Werte, Plugs geben Werte aus).

Zustandsübersicht

Der Funktionsblock besitzt keinen expliziten Zustandsautomaten, sondern verwendet intern einen Speicher für den zuletzt auf **OUT** ausgegebenen Wert. Dieser Speicher dient als Referenz zur Erkennung von Wertänderungen. Der Zustand kann wie folgt beschrieben werden:

- **Bereit** oder **Wartend** – wartet auf eine Änderung von **K** oder eines Eingangswerts.
- **Aktualisierung** – wenn sich der ausgewählte Eingangswert ändert, wird der neue Wert auf **OUT** übernommen. Dieser einfache Zustandsübergang ist implizit und erfordert keine spezielle Konfiguration.

Anwendungsszenarien

- **Signalumschaltung in der Automatisierungstechnik** – Auswahl eines von acht analogen oder digitalen Messwerten (z.B. Temperatur, Druck, Füllstand) für die weitere Verarbeitung.
- **Flexibles Routing** – Dynamische Umleitung von Datensignalen in Produktionsanlagen oder Prozesssteuerungen, ohne fest verdrahtete Verschaltung.
- **Redundanz- und Qualitätsüberwachung** – Durch die Ereignisausgabe nur bei Änderungen können überwachte Werte effizient an Leitsysteme oder Visualisierungen weitergegeben werden.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu klassischen Multiplexern (z.B. in IEC 61499 mit einfachen Datenports) bietet **AUI_AUI_MUX_8** folgende Vorteile:

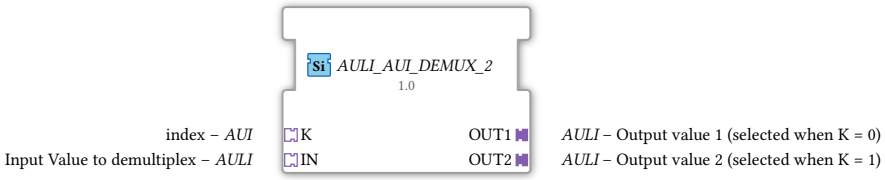
- **Adapter-basierte Schnittstellen** – Einheitliche, wiederverwendbare Kommunikationsstruktur über AUI-Typen.
- **Änderungsbasierte Ausgabe** – Reduziert die Anzahl der Ereignisse im System und senkt die Last in nachfolgenden Funktionsblöcken.
- **Generische Implementierung** – Kann über Klassennamen angepasst und für verschiedene Adaptertypen wiederverwendet werden. Nachteile gegenüber einfachen Multiplexern sind die Abhängigkeit von passenden AUI-Adaptertypen und die etwas komplexere Einbindung in ein Netzwerk.

Fazit

Der **AUI_AUI_MUX_8** ist ein vielseitiger Multiplexer für AUI-basierte Systeme, der eine kompakte und effiziente Auswahl eines von acht Eingangswerten ermöglicht. Dank seiner ereignisgesteuerten Änderungserkennung eignet er sich besonders für Anwendungen, bei denen eine minimale Kommunikationslast und präzise Wertübertragung gefordert sind. Die reine Adapter-Schnittstelle erleichtert die Integration in moderne IEC-61499-Architekturen und fördert die Wiederverwendbarkeit des Bausteins.

ULINT

AULI_AUI_DEMUX_2



• – ○ • – ○ • – ○ •

Einleitung

Der `AULI_AUI_DEMUX_2` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `ULINT`. Anders als `AULI_DEMUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AULI_AUI_DEMUX_2` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`ULINT`, 64-Bit-Ganzzahl (vorzeichenlos)), der an einen der `n` Ausgänge weitergeleitet wird (AULI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AULI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AULI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Ausgangs-Adapter (`OUT1` … `OUT2`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.

- Generische Implementierung (`GEN_AULI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AULI_AUI_DEMUX_2 … AULI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`ULINT`, 64-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AULI_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

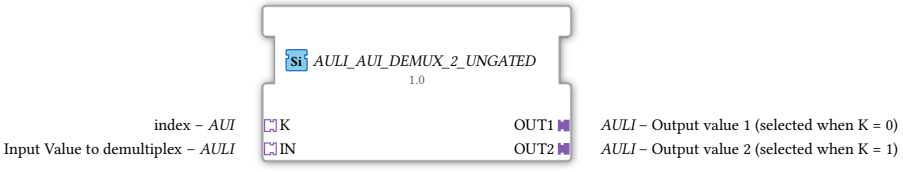
- **`AULI_AUI_DEMUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AULI_AUI_DEMUX_2` überträgt die Demultiplexer-Logik von `AULI_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AULI_AUI_DEMUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AULI_AUI_DEMUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AULI_AUI_DEMUX_2_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `ULINT`. Anders als `AULI_DEMUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AULI_AUI_DEMUX_2_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`ULINT`, 64-Bit-Ganzzahl (vorzeichenlos)), der an einen der `n` Ausgänge weitergeleitet wird (AULI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AULI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AULI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Ausgangs-Adapter (`OUT1` … `OUT2`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AULI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AULI_AUI_DEMUX_2_UNGATED … AULI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`ULINT`, 64-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)

- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AULI_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

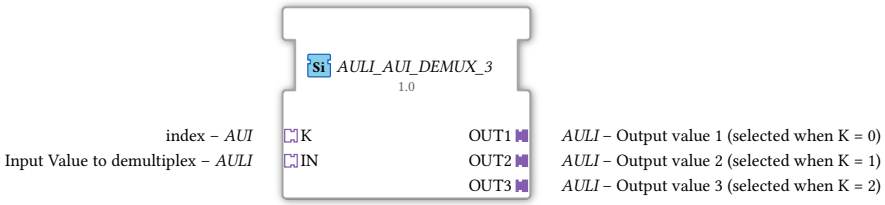
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AULI_AUI_DEMUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AULI_AUI_DEMUX_2_UNGATED` überträgt die Demultiplexer-Logik von `AULI_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AULI_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AULI_AUI_DEMUX_3` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `ULINT`. Anders als `AULI_DEMUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AULI_AUI_DEMUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`ULINT`, 64-Bit-Ganzzahl (vorzeichenlos)), der an einen der `n` Ausgänge weitergeleitet wird (AULI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AULI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AULI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AULI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Ausgangs-Adapter (`OUT1` … `OUT3`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AULI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AULI_AUI_DEMUX_2 … AULI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`ULINT`, 64-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AULI_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

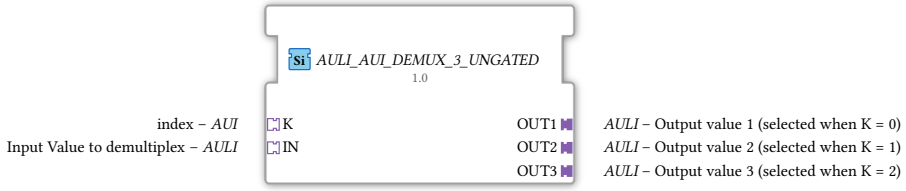
- **`AULI_AUI_DEMUX_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AULI_AUI_DEMUX_3` überträgt die Demultiplexer-Logik von `AULI_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AULI_AUI_DEMUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AULI_AUI_DEMUX_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AULI_AUI_DEMUX_3_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `ULINT`. Anders als `AULI_DEMUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AULI_AUI_DEMUX_3_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`ULINT`, 64-Bit-Ganzzahl (vorzeichenlos)), der an einen der `n` Ausgänge weitergeleitet wird (AULI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AULI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AULI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AULI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Ausgangs-Adapter (`OUT1` … `OUT3`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.

- Generische Implementierung (`GEN_AULI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AULI_AUI_DEMUX_2 … AULI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`ULINT`, 64-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AULI_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

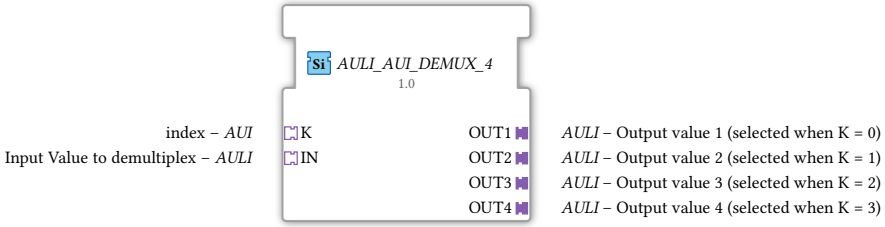
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AULI_AUI_DEMUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AULI_AUI_DEMUX_3_UNGATED` überträgt die Demultiplexer-Logik von `AULI_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AULI_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AULI_AUI_DEMUX_4` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `ULINT`. Anders als `AULI_DEMUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AULI_AUI_DEMUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`ULINT`, 64-Bit-Ganzzahl (vorzeichenlos)), der an einen der `n` Ausgänge weitergeleitet wird (AULI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AULI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AULI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AULI-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AULI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AULI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AULI_AUI_DEMUX_2 … AULI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`ULINT`, 64-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AULI_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

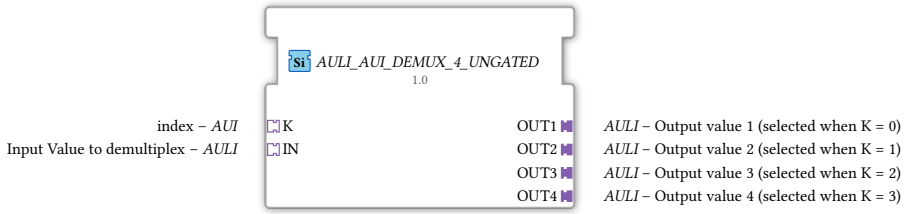
- **`AULI_AUI_DEMUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AULI_AUI_DEMUX_4` überträgt die Demultiplexer-Logik von `AULI_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AULI_AUI_DEMUX_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AULI_AUI_DEMUX_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AULI_AUI_DEMUX_4_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `ULINT`. Anders als `AULI_DEMUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AULI_AUI_DEMUX_4_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`ULINT`, 64-Bit-Ganzzahl (vorzeichenlos)), der an einen der `n` Ausgänge weitergeleitet wird (AULI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AULI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AULI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AULI-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AULI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AULI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AULI_AUI_DEMUX_2 … AULI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`ULINT`, 64-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AULI_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

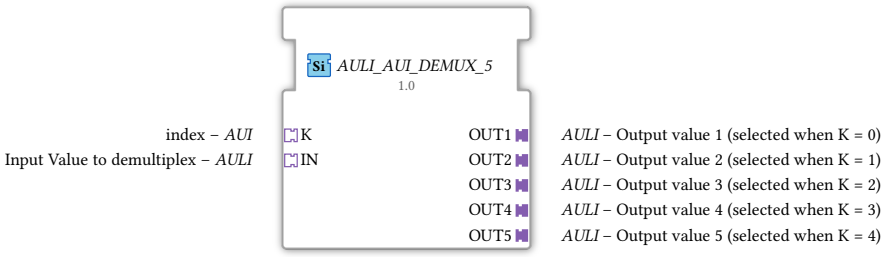
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AULI_AUI_DEMUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AULI_AUI_DEMUX_4_UNGATED` überträgt die Demultiplexer-Logik von `AULI_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AULI_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AULI_AUI_DEMUX_5` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `ULINT`. Anders als `AULI_DEMUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AULI_AUI_DEMUX_5` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`ULINT`, 64-Bit-Ganzzahl (vorzeichenlos)), der an einen der `n` Ausgänge weitergeleitet wird (AULI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AULI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AULI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AULI-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AULI-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AULI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Ausgangs-Adapter (`OUT1` … `OUT5`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AULI_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AULI_AUI_DEMUX_2 … AULI_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`ULINT`, 64-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AULI_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

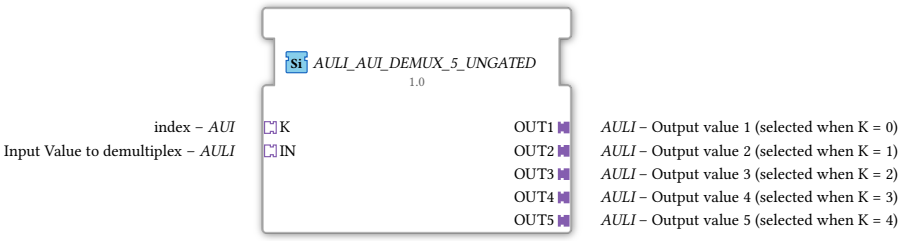
- **`AULI_AUI_DEMUX_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AULI_AUI_DEMUX_5` überträgt die Demultiplexer-Logik von `AULI_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AULI_AUI_DEMUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AULI_AUI_DEMUX_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AULI_AUI_DEMUX_5_UNGATED ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp ULINT. Anders als AULI_DEMUX_5 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AULI_AUI_DEMUX_5_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (ULINT, 64-Bit-Ganzzahl (vorzeichenlos)), der an einen der n Ausgänge weitergeleitet wird (AULI-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AULI-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AULI-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AULI-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AULI-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AULI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 5 Ausgangs-Adapter (OUT1 … OUT5) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AULI_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AULI_AUI_DEMUX_2 … AULI_AUI_DEMUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`ULINT`, 64-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

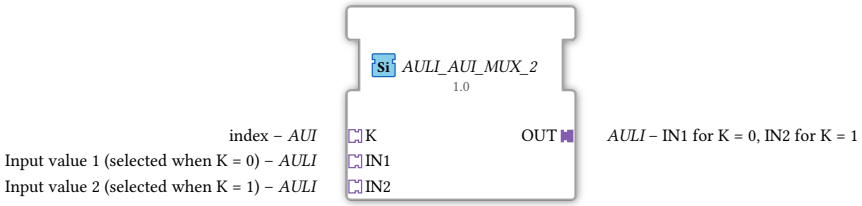
Vergleich mit `AULI_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AULI_AUI_DEMUX_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AULI_AUI_DEMUX_5_UNGATED` überträgt die Demultiplexer-Logik von `AULI_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AULI_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AULI_AUI_MUX_2` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `ULINT`. Anders als `AULI_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AULI_AUI_MUX_2` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (AULI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (AULI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AULI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AULI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AULI_AUI_MUX_2 … AULI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (64-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AULI_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AULI_AUI_MUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

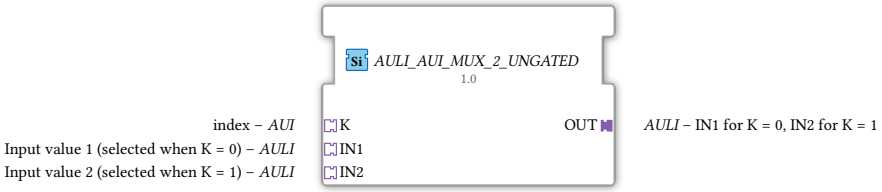
Fazit

Der `AULI_AUI_MUX_2` überträgt die Multiplexer-Logik von `AULI_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter

aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AULI_AUI_MUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AULI_AUI_MUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `AULI_AUI_MUX_2_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `ULINT`. Anders als `AULI_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AULI_AUI_MUX_2_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (AULI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (AULI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AULI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AULI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AULI_AUI_MUX_2_UNGATED … AULI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (64-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AULI_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

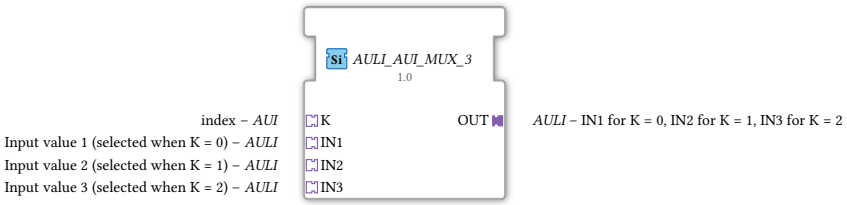
Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AULI_AUI_MUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AULI_AUI_MUX_2_UNGATED` überträgt die Multiplexer-Logik von `AULI_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AULI_AUI_MUX_3



• – ○ • – ○ • – ○ •

Einleitung

Der `AULI_AUI_MUX_3` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `ULINT`. Anders als `AULI_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem

K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AULI_AUI_MUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (AULI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (AULI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (AULI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AULI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN3`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Eingangs-Adapter (`IN1` … `IN3`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AULI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AULI_AUI_MUX_2 … AULI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (64-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AULI_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

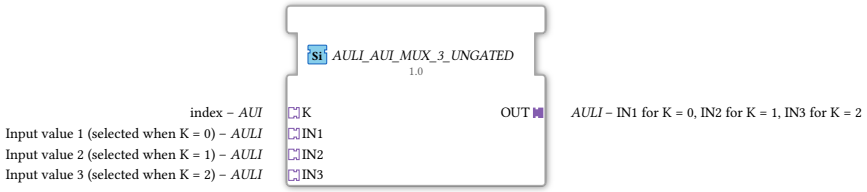
- **`AULI_AUI_MUX_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AULI_AUI_MUX_3` überträgt die Multiplexer-Logik von `AULI_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AULI_AUI_MUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AULI_AUI_MUX_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AULI_AUI_MUX_3_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `ULINT`. Anders als `AULI_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AULI_AUI_MUX_3_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (AULI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (AULI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (AULI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AULI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN3`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Eingangs-Adapter (`IN1` … `IN3`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AULI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AULI_AUI_MUX_2 … AULI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (64-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AULI_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AULI_AUI_MUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AULI_AUI_MUX_3_UNGATED` überträgt die Multiplexer-Logik von `AULI_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AULI_AUI_MUX_4

• – ○ • – ○ • – ○ •

Einleitung

Der `AULI_AUI_MUX_4` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `ULINT`. Anders als `AULI_MUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AULI_AUI_MUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index `K = 0` an `OUT` durchgereicht (AULI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index `K = 1` an `OUT` durchgereicht (AULI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index `K = 2` an `OUT` durchgereicht (AULI-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index `K = 3` an `OUT` durchgereicht (AULI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AULI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN4`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Eingangs-Adapter (`IN1` … `IN4`) gerade ausgewählt ist.

2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AULI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AULI_AUI_MUX_2 … AULI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (64-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AULI_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AULI_AUI_MUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AULI_AUI_MUX_4` überträgt die Multiplexer-Logik von `AULI_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AULI_AUI_MUX_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AULI_AUI_MUX_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ○ • – ○ • – ○ •

Einleitung

Der `AULI_AUI_MUX_4_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `ULINT`. Anders als `AULI_MUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AULI_AUI_MUX_4_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index `K = 0` an `OUT` durchgereicht (AULI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index `K = 1` an `OUT` durchgereicht (AULI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index `K = 2` an `OUT` durchgereicht (AULI-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index `K = 3` an `OUT` durchgereicht (AULI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AULI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN4`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Eingangs-Adapter (`IN1` … `IN4`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AULI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AULI_AUI_MUX_2 … AULI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (64-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AULI_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AULI_AUI_MUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AULI_AUI_MUX_4_UNGATED` überträgt die Multiplexer-Logik von `AULI_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AULI_AUI_MUX_5

• – ○ • – ○ • – ○ •

Einleitung

Der `AULI_AUI_MUX_5` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `ULINT`. Anders als `AULI_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AULI_AUI_MUX_5` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index `K = 0` an `OUT` durchgereicht (AULI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index `K = 1` an `OUT` durchgereicht (AULI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index `K = 2` an `OUT` durchgereicht (AULI-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index `K = 3` an `OUT` durchgereicht (AULI-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index `K = 4` an `OUT` durchgereicht (AULI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AULI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN5`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Eingangs-Adapter (`IN1` … `IN5`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AULI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AULI_AUI_MUX_2 … AULI_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (64-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AULI_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

- **`AULI_AUI_MUX_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AULI_AUI_MUX_5` überträgt die Multiplexer-Logik von `AULI_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AULI_AUI_MUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AULI_AUI_MUX_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ○ • – ○ • – ○ •

Einleitung

Der `AULI_AUI_MUX_5_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `ULINT`. Anders als `AULI_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AULI_AUI_MUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index `K = 0` an `OUT` durchgereicht (AULI-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index `K = 1` an `OUT` durchgereicht (AULI-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index `K = 2` an `OUT` durchgereicht (AULI-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index `K = 3` an `OUT` durchgereicht (AULI-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index `K = 4` an `OUT` durchgereicht (AULI-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AULI-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN5`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Eingangs-Adapter (`IN1` … `IN5`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AULI_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AULI_AUI_MUX_2 … AULI_AUI_MUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (64-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AULI_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

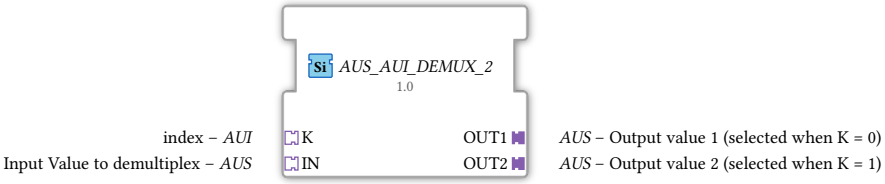
- **AULI_AUI_MUX_5**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der AULI_AUI_MUX_5_UNGATED überträgt die Multiplexer-Logik von AULI_MUX_5 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

USINT

AUS_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AUS_AUI_DEMUX_2 ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp USINT. Anders als AUS_DEMUX_2 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AUS_AUI_DEMUX_2 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (USINT, 8-Bit-Ganzzahl (vorzeichenlos)), der an einen der n Ausgänge weitergeleitet wird (AUS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AUS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AUS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 2 Ausgangs-Adapter (OUT1 … OUT2) gerade ausgewählt ist.

2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUS_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUS_AUI_DEMUX_2 … AUS_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`USINT`, 8-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUS_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

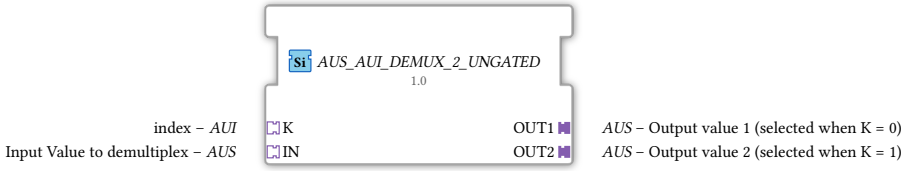
- **`AUS_AUI_DEMUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AUS_AUI_DEMUX_2` überträgt die Demultiplexer-Logik von `AUS_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUS_AUI_DEMUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUS_AUI_DEMUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AUS_AUI_DEMUX_2_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `USINT`. Anders als `AUS_DEMUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUS_AUI_DEMUX_2_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`USINT`, 8-Bit-Ganzzahl (vorzeichenlos)), der an einen der `n` Ausgänge weitergeleitet wird (AUS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AUS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AUS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Ausgangs-Adapter (`OUT1` … `OUT2`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUS_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUS_AUI_DEMUX_2_UNGATED … AUS_AUI_DEMUX_5).

- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`USINT`, 8-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

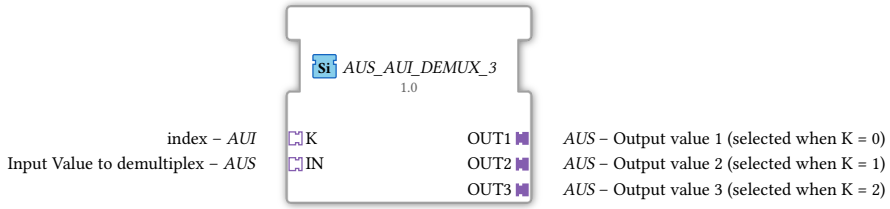
Vergleich mit `AUS_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AUS_AUI_DEMUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AUS_AUI_DEMUX_2_UNGATED` überträgt die Demultiplexer-Logik von `AUS_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUS_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AUS_AUI_DEMUX_3` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `USINT`. Anders als `AUS_DEMUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUS_AUI_DEMUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`USINT`, 8-Bit-Ganzzahl (vorzeichenlos)), der an einen der `n` Ausgänge weitergeleitet wird (AUS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AUS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AUS-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AUS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Ausgangs-Adapter (`OUT1` … `OUT3`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUS_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUS_AUI_DEMUX_2 … AUS_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`USINT`, 8-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUS_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

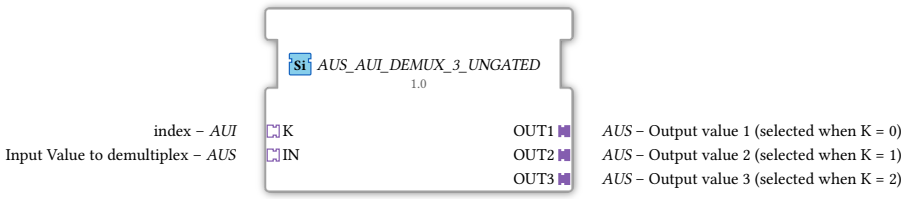
- **AUS_AUI_DEMUX_3_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der AUS_AUI_DEMUX_3 überträgt die Demultiplexer-Logik von AUS_DEMUX_3 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUS_AUI_DEMUX_3_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von AUS_AUI_DEMUX_3. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AUS_AUI_DEMUX_3_UNGATED ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp USINT. Anders als AUS_DEMUX_3 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AUS_AUI_DEMUX_3_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (USINT, 8-Bit-Ganzzahl (vorzeichenlos)), der an einen der n Ausgänge weitergeleitet wird (AUS-Adaptertyp).

- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AUS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AUS-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AUS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Ausgangs-Adapter (`OUT1` … `OUT3`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUS_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUS_AUI_DEMUX_2 … AUS_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`USINT`, 8-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUS_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

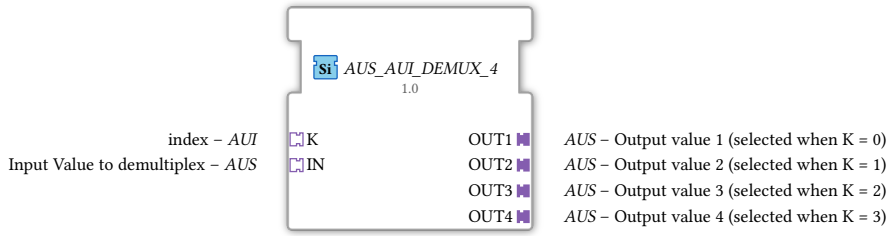
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AUS_AUI_DEMUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AUS_AUI_DEMUX_3_UNGATED` überträgt die Demultiplexer-Logik von `AUS_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUS_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AUS_AUI_DEMUX_4` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `USINT`. Anders als `AUS_DEMUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUS_AUI_DEMUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`USINT`, 8-Bit-Ganzzahl (vorzeichenlos)), der an einen der `n` Ausgänge weitergeleitet wird (AUS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AUS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AUS-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AUS-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AUS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUS_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUS_AUI_DEMUX_2 … AUS_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`USINT`, 8-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUS_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

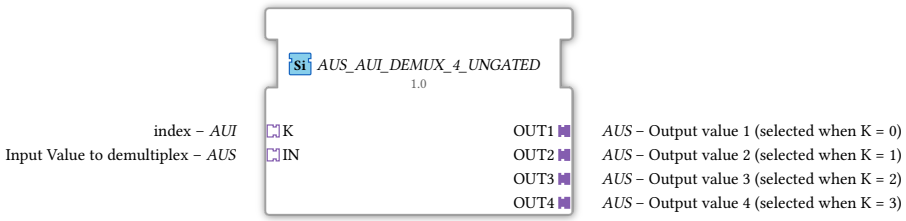
- **`AUS_AUI_DEMUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AUS_AUI_DEMUX_4` überträgt die Demultiplexer-Logik von `AUS_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUS_AUI_DEMUX_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUS_AUI_DEMUX_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der AUS_AUI_DEMUX_4_UNGATED ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp USINT. Anders als AUS_DEMUX_4 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AUS_AUI_DEMUX_4_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (USINT, 8-Bit-Ganzzahl (vorzeichenlos)), der an einen der n Ausgänge weitergeleitet wird (AUS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AUS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AUS-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AUS-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AUS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 4 Ausgangs-Adapter (OUT1 … OUT4) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AUS_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUS_AUI_DEMUX_2 … AUS_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`USINT`, 8-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

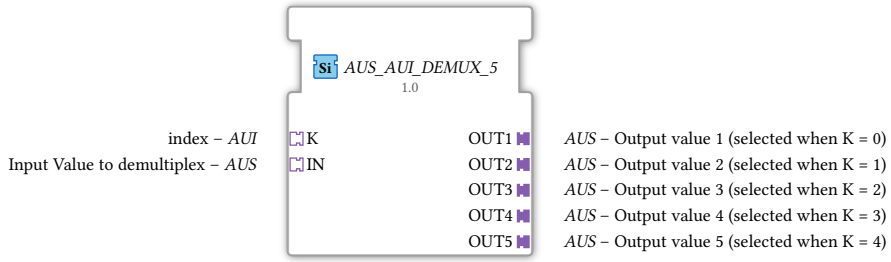
Vergleich mit `AUS_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AUS_AUI_DEMUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AUS_AUI_DEMUX_4_UNGATED` überträgt die Demultiplexer-Logik von `AUS_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUS_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AUS_AUI_DEMUX_5` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `USINT`. Anders als `AUS_DEMUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUS_AUI_DEMUX_5` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`USINT`, 8-Bit-Ganzzahl (vorzeichenlos)), der an einen der `n` Ausgänge weitergeleitet wird (AUS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AUS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AUS-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AUS-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AUS-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AUS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der `5` Ausgangs-Adapter (`OUT1` … `OUT5`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUS_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUS_AUI_DEMUX_2 … AUS_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`USINT`, 8-Bit-Ganzzahl (vorzeichenlos)) auf bis zu `5` nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUS_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

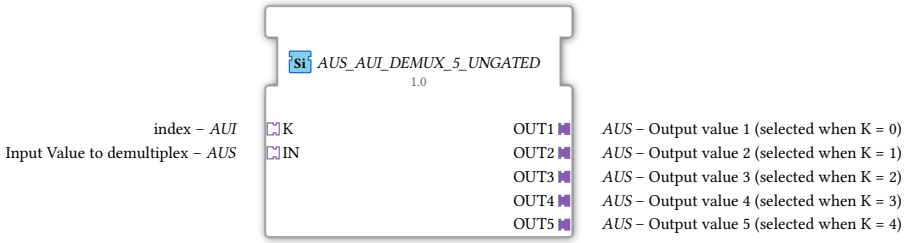
- **AUS_AUI_DEMUX_5_UNGATED**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der AUS_AUI_DEMUX_5 überträgt die Demultiplexer-Logik von AUS_DEMUX_5 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUS_AUI_DEMUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AUS_AUI_DEMUX_5. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AUS_AUI_DEMUX_5_UNGATED ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp USINT. Anders als AUS_DEMUX_5 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AUS_AUI_DEMUX_5_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (USINT, 8-Bit-Ganzzahl (vorzeichenlos)), der an einen der n Ausgänge weitergeleitet wird (AUS-Adaptertyp).

- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AUS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AUS-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AUS-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AUS-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AUS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Ausgangs-Adapter (`OUT1` … `OUT5`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUS_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUS_AUI_DEMUX_2 … AUS_AUI_DEMUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`USINT`, 8-Bit-Ganzzahl (vorzeichenlos)) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUS_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

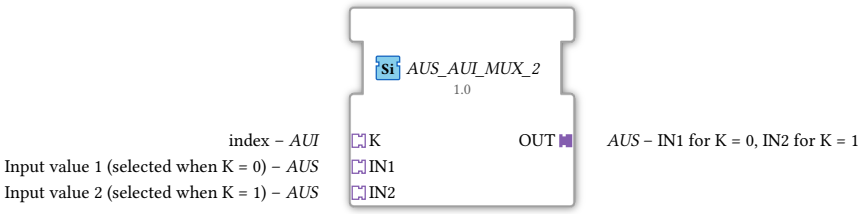
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AUS_AUI_DEMUX_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AUS_AUI_DEMUX_5_UNGATED` überträgt die Demultiplexer-Logik von `AUS_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUS_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AUS_AUI_MUX_2` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `USINT`. Anders als `AUS_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUS_AUI_MUX_2` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (AUS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (AUS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AUS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.

- Generische Implementierung (`GEN_AUS_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUS_AUI_MUX_2 … AUS_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (8-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUS_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

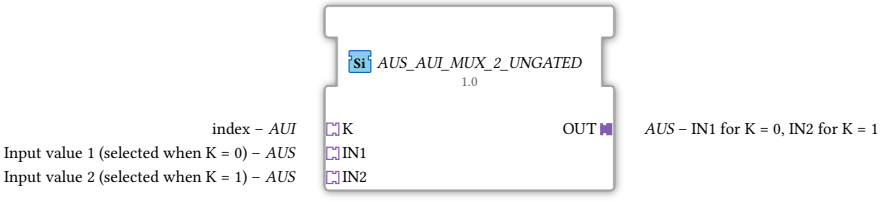
- **`AUS_AUI_MUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AUS_AUI_MUX_2` überträgt die Multiplexer-Logik von `AUS_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUS_AUI_MUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUS_AUI_MUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AUS_AUI_MUX_2_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `USINT`. Anders als `AUS_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter

Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AUS_AUI_MUX_2_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index K = 0 an OUT durchgereicht (AUS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index K = 1 an OUT durchgereicht (AUS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AUS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN2:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 2 Eingangs-Adapter (IN1 … IN2) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf OUT gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AUS_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUS_AUI_MUX_2_UNGATED … AUS_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN1…IN2) ⟶ aktuell selektierten Eingang auslesen, OUT bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (8-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUS_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

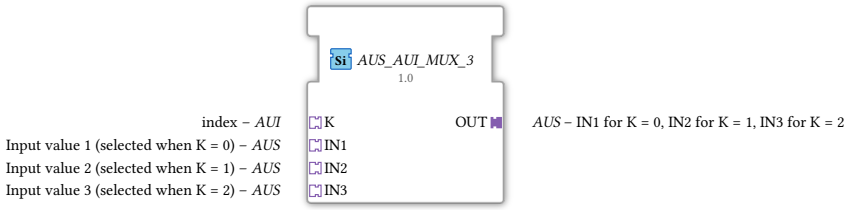
Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AUS_AUI_MUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AUS_AUI_MUX_2_UNGATED` überträgt die Multiplexer-Logik von `AUS_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUS_AUI_MUX_3



• – ○ • – ○ • – ○ •

Einleitung

Der `AUS_AUI_MUX_3` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `USINT`. Anders als `AUS_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUS_AUI_MUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.

- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (AUS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (AUS-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (AUS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AUS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN3`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Eingangs-Adapter (`IN1` … `IN3`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUS_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUS_AUI_MUX_2 … AUS_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (8-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUS_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

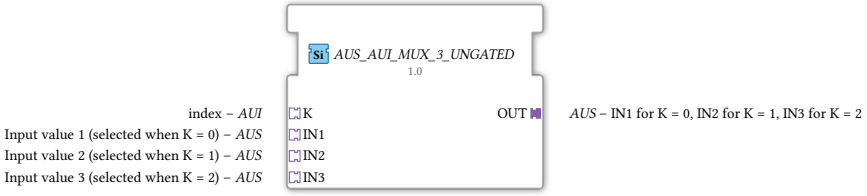
- `AUS_AUI_MUX_3_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AUS_AUI_MUX_3` überträgt die Multiplexer-Logik von `AUS_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUS_AUI_MUX_3_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von AUS_AUI_MUX_3. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der AUS_AUI_MUX_3_UNGATED ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp USINT. Anders als AUS_MUX_3 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AUS_AUI_MUX_3_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index K = 0 an OUT durchgereicht (AUS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index K = 1 an OUT durchgereicht (AUS-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index K = 2 an OUT durchgereicht (AUS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AUS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN3:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 3 Eingangs-Adapter (IN1 … IN3) gerade ausgewählt ist.

2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUS_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUS_AUI_MUX_2 … AUS_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (8-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUS_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AUS_AUI_MUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AUS_AUI_MUX_3_UNGATED` überträgt die Multiplexer-Logik von `AUS_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUS_AUI_MUX_4

• – ○ • – ○ • – ○ •

Einleitung

Der `AUS_AUI_MUX_4` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `USINT`. Anders als `AUS_MUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“).

Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge
- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AUS_AUI_MUX_4 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge
- Keine Ereignis-Ausgänge.

Daten-Eingänge
- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge
- Keine direkten Datenausgänge vorhanden.

Adapter
- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index K = 0 an OUT durchgereicht (AUS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index K = 1 an OUT durchgereicht (AUS-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index K = 2 an OUT durchgereicht (AUS-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index K = 3 an OUT durchgereicht (AUS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AUS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN4:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 4 Eingangs-Adapter (IN1 … IN4) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf OUT gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten
- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AUS_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUS_AUI_MUX_2 … AUS_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN1…IN4) ⟶ aktuell selektierten Eingang auslesen, OUT bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (8-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUS_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AUS_AUI_MUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AUS_AUI_MUX_4` überträgt die Multiplexer-Logik von `AUS_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUS_AUI_MUX_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUS_AUI_MUX_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ○ • – ○ • – ○ •

Einleitung

Der `AUS_AUI_MUX_4_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `USINT`. Anders als `AUS_MUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUS_AUI_MUX_4_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index `K = 0` an `OUT` durchgereicht (AUS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index `K = 1` an `OUT` durchgereicht (AUS-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index `K = 2` an `OUT` durchgereicht (AUS-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index `K = 3` an `OUT` durchgereicht (AUS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AUS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN4`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Eingangs-Adapter (`IN1` … `IN4`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUS_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUS_AUI_MUX_2 … AUS_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (8-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUS_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

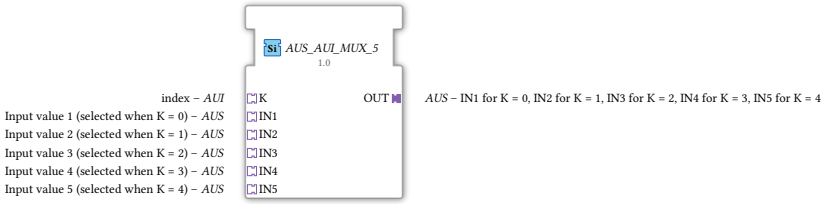
Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **AUS_AUI_MUX_4**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der AUS_AUI_MUX_4_UNGATED überträgt die Multiplexer-Logik von AUS_MUX_4 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUS_AUI_MUX_5



• – ○ • – ○ • – ○ •

Einleitung

Der AUS_AUI_MUX_5 ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp USINT. Anders als AUS_MUX_5 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AUS_AUI_MUX_5 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index K = 0 an OUT durchgereicht (AUS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index K = 1 an OUT durchgereicht (AUS-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index K = 2 an OUT durchgereicht (AUS-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index K = 3 an OUT durchgereicht (AUS-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index K = 4 an OUT durchgereicht (AUS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AUS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN5:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Eingangs-Adapter (`IN1` … `IN5`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUS_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUS_AUI_MUX_2 … AUS_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (8-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUS_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

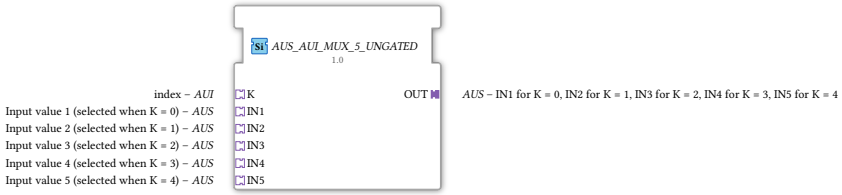
- `AUS_AUI_MUX_5_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AUS_AUI_MUX_5` überträgt die Multiplexer-Logik von `AUS_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AUS_AUI_MUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AUS_AUI_MUX_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `AUS_AUI_MUX_5_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `USINT`. Anders als `AUS_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AUS_AUI_MUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index `K = 0` an `OUT` durchgereicht (AUS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index `K = 1` an `OUT` durchgereicht (AUS-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index `K = 2` an `OUT` durchgereicht (AUS-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index `K = 3` an `OUT` durchgereicht (AUS-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index `K = 4` an `OUT` durchgereicht (AUS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AUS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN5`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Eingangs-Adapter (`IN1` … `IN5`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AUS_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AUS_AUI_MUX_2 … AUS_AUI_MUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (8-Bit-Ganzzahl (vorzeichenlos)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AUS_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

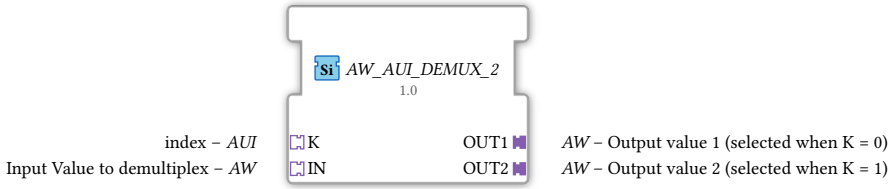
- `AUS_AUI_MUX_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AUS_AUI_MUX_5_UNGATED` überträgt die Multiplexer-Logik von `AUS_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

WORD

AW_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AW_AUI_DEMUX_2` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `WORD`. Anders als `AW_DEMUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AW_AUI_DEMUX_2` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`WORD`, 16-Bit-Bitmuster), der an einen der `n` Ausgänge weitergeleitet wird (AW-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AW-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AW-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Ausgangs-Adapter (`OUT1` … `OUT2`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AW_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AW_AUI_DEMUX_2 … AW_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) → aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`WORD`, 16-Bit-Bitmuster) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)

- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AW_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

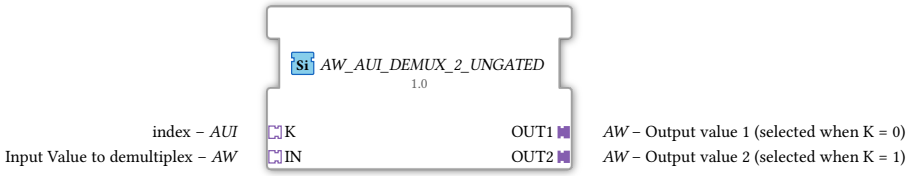
- **`AW_AUI_DEMUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AW_AUI_DEMUX_2` überträgt die Demultiplexer-Logik von `AW_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AW_AUI_DEMUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AW_AUI_DEMUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AW_AUI_DEMUX_2_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `WORD`. Anders als `AW_DEMUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AW_AUI_DEMUX_2_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`WORD`, 16-Bit-Bitmuster), der an einen der `n` Ausgänge weitergeleitet wird (AW-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AW-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AW-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Ausgangs-Adapter (`OUT1` … `OUT2`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AW_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AW_AUI_DEMUX_2_UNGATED … AW_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`WORD`, 16-Bit-Bitmuster) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AW_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

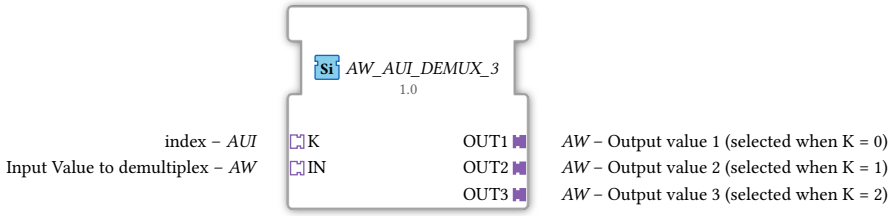
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AW_AUI_DEMUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AW_AUI_DEMUX_2_UNGATED` überträgt die Demultiplexer-Logik von `AW_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AW_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AW_AUI_DEMUX_3` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `WORD`. Anders als `AW_DEMUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AW_AUI_DEMUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`WORD`, 16-Bit-Bitmuster), der an einen der `n` Ausgänge weitergeleitet wird (AW-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AW-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AW-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AW-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Ausgangs-Adapter (`OUT1` … `OUT3`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.

- Generische Implementierung (`GEN_AW_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AW_AUI_DEMUX_2 … AW_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`WORD`, 16-Bit-Bitmuster) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AW_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

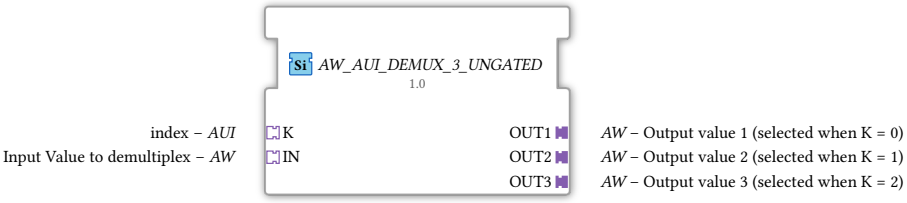
- **`AW_AUI_DEMUX_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AW_AUI_DEMUX_3` überträgt die Demultiplexer-Logik von `AW_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AW_AUI_DEMUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AW_AUI_DEMUX_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AW_AUI_DEMUX_3_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `WORD`. Anders als `AW_DEMUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter

Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge
- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AW_AUI_DEMUX_3_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge
- Keine Ereignis-Ausgänge.

Daten-Eingänge
- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge
- Keine direkten Datenausgänge vorhanden.

Adapter
- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (WORD, 16-Bit-Bitmuster), der an einen der n Ausgänge weitergeleitet wird (AW-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AW-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AW-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AW-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 3 Ausgangs-Adapter (OUT1 … OUT3) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten
- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AW_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AW_AUI_DEMUX_2 … AW_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen IN-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`WORD`, 16-Bit-Bitmuster) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AW_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

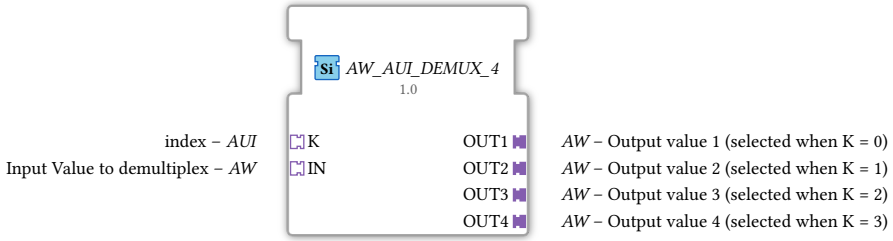
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AW_AUI_DEMUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AW_AUI_DEMUX_3_UNGATED` überträgt die Demultiplexer-Logik von `AW_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AW_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AW_AUI_DEMUX_4` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `WORD`. Anders als `AW_DEMUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AW_AUI_DEMUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`WORD`, 16-Bit-Bitmuster), der an einen der `n` Ausgänge weitergeleitet wird (AW-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AW-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AW-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AW-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AW-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AW_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AW_AUI_DEMUX_2 … AW_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`WORD`, 16-Bit-Bitmuster) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AW_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

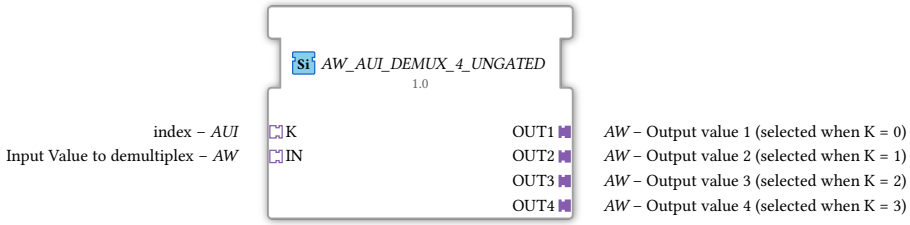
- **`AW_AUI_DEMUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AW_AUI_DEMUX_4` überträgt die Demultiplexer-Logik von `AW_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AW_AUI_DEMUX_4_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von AW_AUI_DEMUX_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AW_AUI_DEMUX_4_UNGATED ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp WORD. Anders als AW_DEMUX_4 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AW_AUI_DEMUX_4_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (WORD, 16-Bit-Bitmuster), der an einen der n Ausgänge weitergeleitet wird (AW-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AW-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AW-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AW-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AW-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AW_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AW_AUI_DEMUX_2 … AW_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`WORD`, 16-Bit-Bitmuster) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AW_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

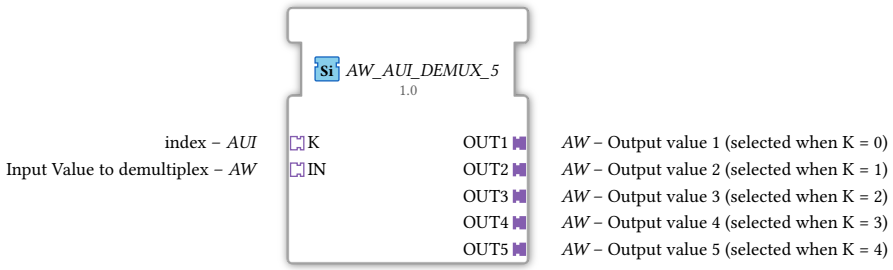
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AW_AUI_DEMUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AW_AUI_DEMUX_4_UNGATED` überträgt die Demultiplexer-Logik von `AW_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AW_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AW_AUI_DEMUX_5 ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp WORD. Anders als AW_DEMUX_5 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AW_AUI_DEMUX_5 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (WORD, 16-Bit-Bitmuster), der an einen der n Ausgänge weitergeleitet wird (AW-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AW-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AW-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AW-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AW-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AW-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 5 Ausgangs-Adapter (OUT1 … OUT5) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AW_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AW_AUI_DEMUX_2 … AW_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`WORD`, 16-Bit-Bitmuster) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AW_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

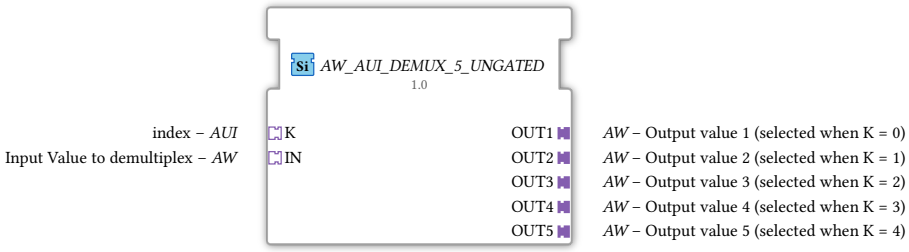
- **`AW_AUI_DEMUX_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AW_AUI_DEMUX_5` überträgt die Demultiplexer-Logik von `AW_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AW_AUI_DEMUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AW_AUI_DEMUX_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `AW_AUI_DEMUX_5_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `WORD`. Anders als `AW_DEMUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AW_AUI_DEMUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`WORD`, 16-Bit-Bitmuster), der an einen der `n` Ausgänge weitergeleitet wird (AW-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AW-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AW-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AW-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AW-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AW-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der `5` Ausgangs-Adapter (`OUT1` … `OUT5`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AW_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AW_AUI_DEMUX_2 … AW_AUI_DEMUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`WORD`, 16-Bit-Bitmuster) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AW_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

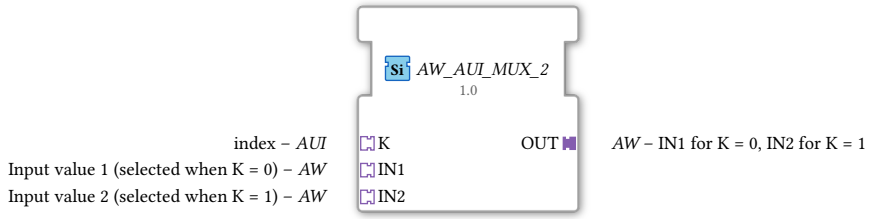
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AW_AUI_DEMUX_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AW_AUI_DEMUX_5_UNGATED` überträgt die Demultiplexer-Logik von `AW_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AW_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AW_AUI_MUX_2` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `WORD`. Anders als `AW_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AW_AUI_MUX_2` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.

- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (AW-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (AW-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AW-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AW_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AW_AUI_MUX_2 … AW_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (16-Bit-Bitmuster) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AW_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AW_AUI_MUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

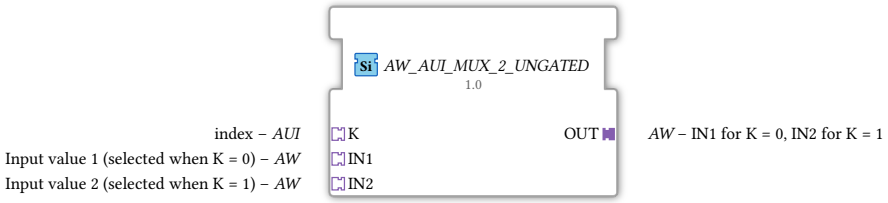
Fazit

Der `AW_AUI_MUX_2` überträgt die Multiplexer-Logik von `AW_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AW_AUI_MUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von AW_AUI_MUX_2. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos

weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AW_AUI_MUX_2_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `WORD`. Anders als `AW_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AW_AUI_MUX_2_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (AW-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (AW-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AW-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AW_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AW_AUI_MUX_2_UNGATED … AW_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (16-Bit-Bitmuster) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AW_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

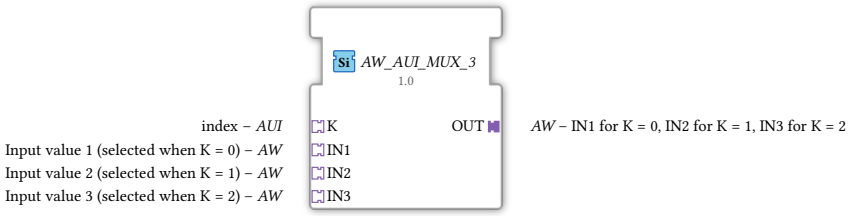
Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AW_AUI_MUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AW_AUI_MUX_2_UNGATED` überträgt die Multiplexer-Logik von `AW_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AW_AUI_MUX_3



• – ○ • – ○ • – ○ •

Einleitung

Der `AW_AUI_MUX_3` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `WORD`. Anders als `AW_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AW_AUI_MUX_3` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (AW-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (AW-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (AW-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AW-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN3`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Eingangs-Adapter (`IN1` … `IN3`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AW_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AW_AUI_MUX_2 … AW_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (16-Bit-Bitmuster) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird

- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AW_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AW_AUI_MUX_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AW_AUI_MUX_3` überträgt die Multiplexer-Logik von `AW_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AW_AUI_MUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AW_AUI_MUX_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.

• – ○ • – ○ • – ○ •

Einleitung

Der `AW_AUI_MUX_3_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `WORD`. Anders als `AW_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AW_AUI_MUX_3_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (AW-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (AW-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (AW-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AW-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN3`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Eingangs-Adapter (`IN1` … `IN3`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AW_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AW_AUI_MUX_2 … AW_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (16-Bit-Bitmuster) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AW_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

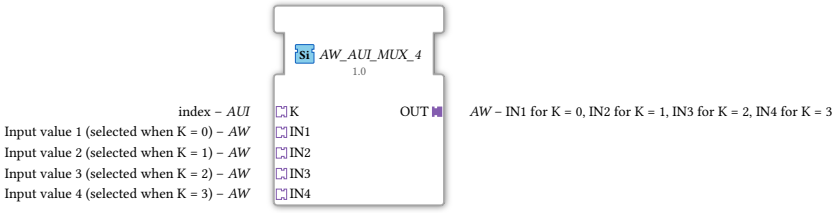
Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AW_AUI_MUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AW_AUI_MUX_3_UNGATED` überträgt die Multiplexer-Logik von `AW_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AW_AUI_MUX_4



• – ○ • – ○ • – ○ •

Einleitung

Der `AW_AUI_MUX_4` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `WORD`. Anders als `AW_MUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AW_AUI_MUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index `K = 0` an `OUT` durchgereicht (AW-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index `K = 1` an `OUT` durchgereicht (AW-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index `K = 2` an `OUT` durchgereicht (AW-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index `K = 3` an `OUT` durchgereicht (AW-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AW-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN4`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Eingangs-Adapter (`IN1` … `IN4`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AW_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AW_AUI_MUX_2 … AW_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (16-Bit-Bitmuster) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AW_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

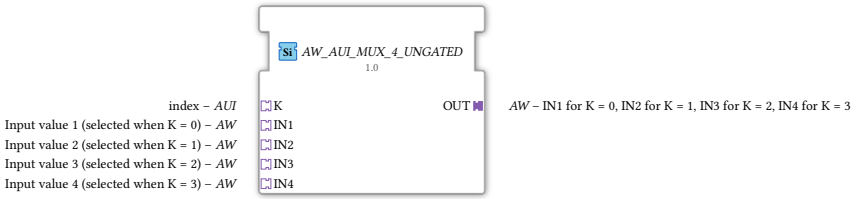
- **`AW_AUI_MUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AW_AUI_MUX_4` überträgt die Multiplexer-Logik von `AW_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AW_AUI_MUX_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AW_AUI_MUX_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der AW_AUI_MUX_4_UNGATED ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp WORD. Anders als AW_MUX_4 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AW_AUI_MUX_4_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index K = 0 an OUT durchgereicht (AW-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index K = 1 an OUT durchgereicht (AW-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index K = 2 an OUT durchgereicht (AW-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index K = 3 an OUT durchgereicht (AW-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AW-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN4:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 4 Eingangs-Adapter (IN1 … IN4) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf OUT gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AW_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AW_AUI_MUX_2 … AW_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (16-Bit-Bitmuster) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AW_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

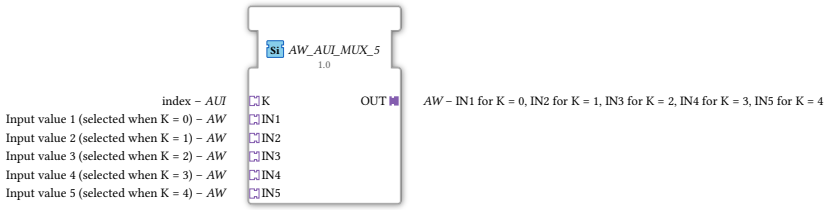
Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AW_AUI_MUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AW_AUI_MUX_4_UNGATED` überträgt die Multiplexer-Logik von `AW_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AW_AUI_MUX_5



• – ○ • – ○ • – ○ •

Einleitung

Der `AW_AUI_MUX_5` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `WORD`. Anders als `AW_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AW_AUI_MUX_5` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index `K = 0` an `OUT` durchgereicht (AW-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index `K = 1` an `OUT` durchgereicht (AW-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index `K = 2` an `OUT` durchgereicht (AW-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index `K = 3` an `OUT` durchgereicht (AW-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index `K = 4` an `OUT` durchgereicht (AW-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AW-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN5`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Eingangs-Adapter (`IN1` … `IN5`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AW_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AW_AUI_MUX_2 … AW_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (16-Bit-Bitmuster) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AW_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

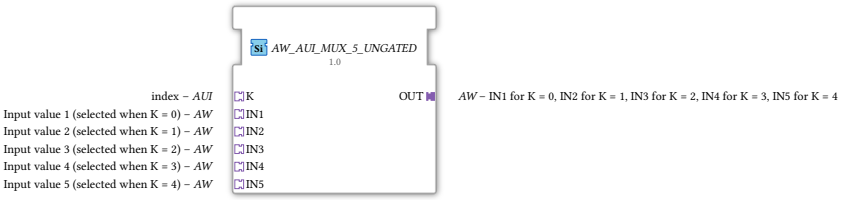
- `AW_AUI_MUX_5_UNGATED`: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AW_AUI_MUX_5` überträgt die Multiplexer-Logik von `AW_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AW_AUI_MUX_5_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von AW_AUI_MUX_5. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der AW_AUI_MUX_5_UNGATED ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp WORD. Anders als AW_MUX_5 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AW_AUI_MUX_5_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index K = 0 an OUT durchgereicht (AW-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index K = 1 an OUT durchgereicht (AW-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index K = 2 an OUT durchgereicht (AW-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index K = 3 an OUT durchgereicht (AW-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index K = 4 an OUT durchgereicht (AW-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AW-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN5:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 5 Eingangs-Adapter (IN1 … IN5) gerade ausgewählt ist.

2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AW_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AW_AUI_MUX_2 … AW_AUI_MUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (16-Bit-Bitmuster) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AW_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

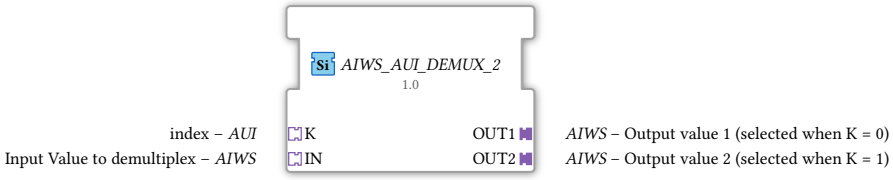
- `AW_AUI_MUX_5`: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AW_AUI_MUX_5_UNGATED` überträgt die Multiplexer-Logik von `AW_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

WSTRING

AIWS_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AIWS_AUI_DEMUX_2` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `WSTRING`. Anders als `AIWS_DEMUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter

Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge
- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AIWS_AUI_DEMUX_2 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge
- Keine Ereignis-Ausgänge.

Daten-Eingänge
- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge
- Keine direkten Datenausgänge vorhanden.

Adapter
- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (WSTRING, Zeichenkette (16-Bit-Wide-Zeichen)), der an einen der n Ausgänge weitergeleitet wird (AIWS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AIWS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AIWS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 2 Ausgangs-Adapter (OUT1 … OUT2) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten
- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AIWS_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIWS_AUI_DEMUX_2 … AIWS_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen IN-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien
- Verteilen eines Wertes (WSTRING, Zeichenkette (16-Bit-Wide-Zeichen)) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index

- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AIWS_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

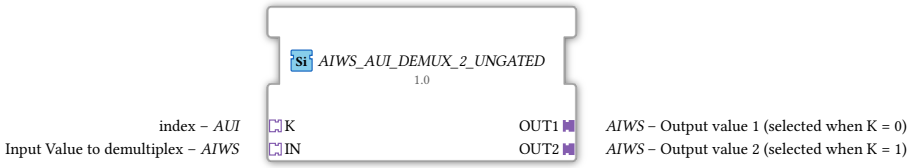
- **`AIWS_AUI_DEMUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AIWS_AUI_DEMUX_2` überträgt die Demultiplexer-Logik von `AIWS_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIWS_AUI_DEMUX_2_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AIWS_AUI_DEMUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `AIWS_AUI_DEMUX_2_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `WSTRING`. Anders als `AIWS_DEMUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AIWS_AUI_DEMUX_2_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`WSTRING`, Zeichenkette (16-Bit-Wide-Zeichen)), der an einen der `n` Ausgänge weitergeleitet wird (AIWS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 2 (AIWS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 2 (AIWS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Ausgangs-Adapter (`OUT1` … `OUT2`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AIWS_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIWS_AUI_DEMUX_2_UNGATED … AIWS_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`WSTRING`, Zeichenkette (16-Bit-Wide-Zeichen)) auf bis zu 2 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AIWS_DEMUX_2` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

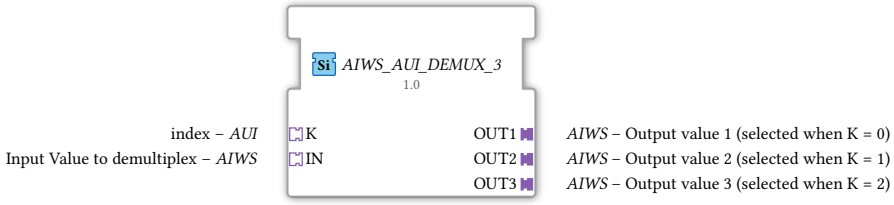
- **`AIWS_AUI_DEMUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AIWS_AUI_DEMUX_2_UNGATED` überträgt die Demultiplexer-Logik von `AIWS_DEMUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als

AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIWS_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AIWS_AUI_DEMUX_3 ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp WSTRING. Anders als AIWS_DEMUX_3 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AIWS_AUI_DEMUX_3 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (WSTRING, Zeichenkette (16-Bit-Wide-Zeichen)), der an einen der n Ausgänge weitergeleitet wird (AIWS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AIWS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AIWS-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AIWS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 3 Ausgangs-Adapter (OUT1 … OUT3) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AIWS_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIWS_AUI_DEMUX_2 … AIWS_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`WSTRING`, Zeichenkette (16-Bit-Wide-Zeichen)) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AIWS_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

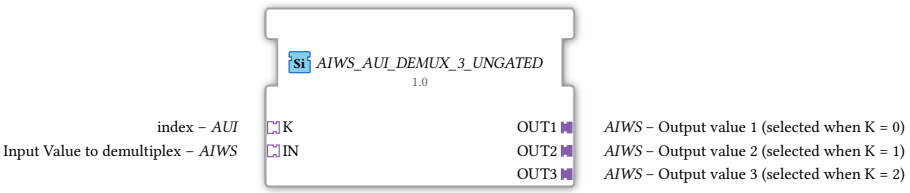
- **`AIWS_AUI_DEMUX_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AIWS_AUI_DEMUX_3` überträgt die Demultiplexer-Logik von `AIWS_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIWS_AUI_DEMUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AIWS_AUI_DEMUX_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der AIWS_AUI_DEMUX_3_UNGATED ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp WSTRING. Anders als AIWS_DEMUX_3 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AIWS_AUI_DEMUX_3_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (WSTRING, Zeichenkette (16-Bit-Wide-Zeichen)), der an einen der n Ausgänge weitergeleitet wird (AIWS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 3 (AIWS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 3 (AIWS-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 3 (AIWS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 3 Ausgangs-Adapter (OUT1 … OUT3) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AIWS_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIWS_AUI_DEMUX_2 … AIWS_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`WSTRING`, Zeichenkette (16-Bit-Wide-Zeichen)) auf bis zu 3 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

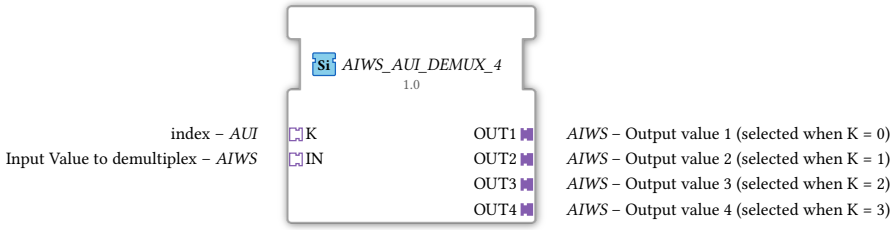
Vergleich mit `AIWS_DEMUX_3` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AIWS_AUI_DEMUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AIWS_AUI_DEMUX_3_UNGATED` überträgt die Demultiplexer-Logik von `AIWS_DEMUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIWS_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AIWS_AUI_DEMUX_4` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `WSTRING`. Anders als `AIWS_DEMUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AIWS_AUI_DEMUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`WSTRING`, Zeichenkette (16-Bit-Wide-Zeichen)), der an einen der `n` Ausgänge weitergeleitet wird (AIWS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AIWS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AIWS-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AIWS-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AIWS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AIWS_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIWS_AUI_DEMUX_2 … AIWS_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`WSTRING`, Zeichenkette (16-Bit-Wide-Zeichen)) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AIWS_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

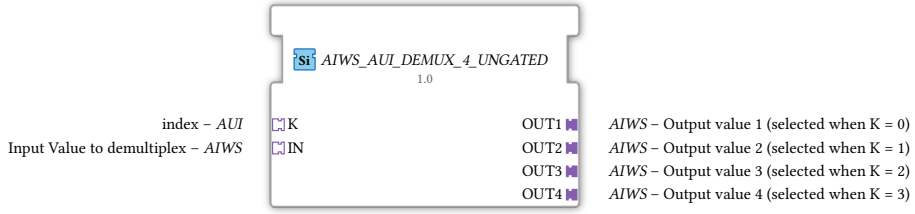
- **`AIWS_AUI_DEMUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AIWS_AUI_DEMUX_4` überträgt die Demultiplexer-Logik von `AIWS_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIWS_AUI_DEMUX_4_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von AIWS_AUI_DEMUX_4. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AIWS_AUI_DEMUX_4_UNGATED ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp WSTRING. Anders als AIWS_DEMUX_4 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AIWS_AUI_DEMUX_4_UNGATED intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (WSTRING, Zeichenkette (16-Bit-Wide-Zeichen)), der an einen der n Ausgänge weitergeleitet wird (AIWS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 4 (AIWS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 4 (AIWS-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 4 (AIWS-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 4 (AIWS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Ausgangs-Adapter (`OUT1` … `OUT4`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AIWS_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIWS_AUI_DEMUX_2 … AIWS_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`WSTRING`, Zeichenkette (16-Bit-Wide-Zeichen)) auf bis zu 4 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AIWS_DEMUX_4` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

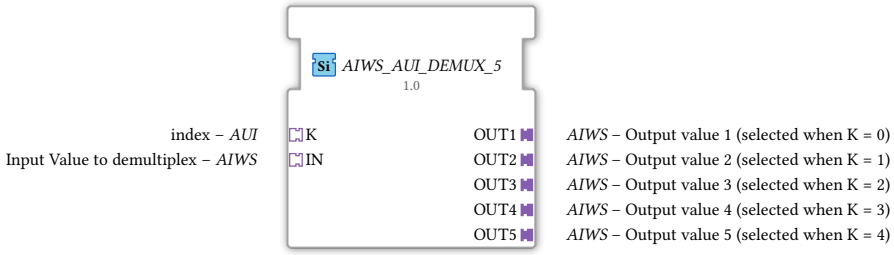
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AIWS_AUI_DEMUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AIWS_AUI_DEMUX_4_UNGATED` überträgt die Demultiplexer-Logik von `AIWS_DEMUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIWS_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AIWS_AUI_DEMUX_5 ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp WSTRING. Anders als AIWS_DEMUX_5 erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ AUI („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AIWS_AUI_DEMUX_5 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (WSTRING, Zeichenkette (16-Bit-Wide-Zeichen)), der an einen der n Ausgänge weitergeleitet wird (AIWS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AIWS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AIWS-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AIWS-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AIWS-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AIWS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis am Eingangs-Adapter IN:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 5 Ausgangs-Adapter (OUT1 … OUT5) gerade ausgewählt ist.
2. Der Datenwert von IN wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von K sofort den aktuellen Wert von IN auf den neu ausgewählten Ausgang, selbst wenn sich IN seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AIWS_AUI_DEMUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIWS_AUI_DEMUX_2 … AIWS_AUI_DEMUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`WSTRING`, Zeichenkette (16-Bit-Wide-Zeichen)) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AIWS_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

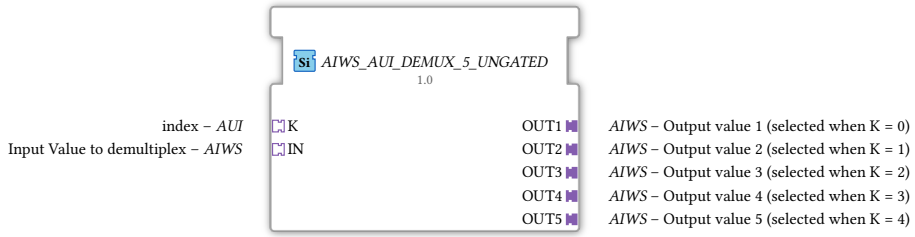
- **`AIWS_AUI_DEMUX_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AIWS_AUI_DEMUX_5` überträgt die Demultiplexer-Logik von `AIWS_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIWS_AUI_DEMUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AIWS_AUI_DEMUX_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AIWS_AUI_DEMUX_5_UNGATED` ist die adapterbasierte Variante des generischen Demultiplexers für den Datentyp `WSTRING`. Anders als `AIWS_DEMUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AIWS_AUI_DEMUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN** (Socket): Eingangsadapter für den am Eingang anliegenden Wert (`WSTRING`, Zeichenkette (16-Bit-Wide-Zeichen)), der an einen der `n` Ausgänge weitergeleitet wird (AIWS-Adaptertyp).
- **OUT1** (Plug): Ausgangsadapter 1 von 5 (AIWS-Adaptertyp).
- **OUT2** (Plug): Ausgangsadapter 2 von 5 (AIWS-Adaptertyp).
- **OUT3** (Plug): Ausgangsadapter 3 von 5 (AIWS-Adaptertyp).
- **OUT4** (Plug): Ausgangsadapter 4 von 5 (AIWS-Adaptertyp).
- **OUT5** (Plug): Ausgangsadapter 5 von 5 (AIWS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis am Eingangs-Adapter `IN`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der `5` Ausgangs-Adapter (`OUT1` … `OUT5`) gerade ausgewählt ist.
2. Der Datenwert von `IN` wird mit dem aktuell auf dem ausgewählten Ausgang gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird dieser Ausgang neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch übernimmt auch eine reine Änderung von `K` sofort den aktuellen Wert von `IN` auf den neu ausgewählten Ausgang, selbst wenn sich `IN` seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AIWS_AUI_DEMUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIWS_AUI_DEMUX_2 … AIWS_AUI_DEMUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN`) ⟶ aktuell selektierten Ausgang bestimmen, ihn bei Wertänderung mit dem aktuellen `IN`-Wert aktualisieren und sein Adapter-Event senden.

Anwendungsszenarien

- Verteilen eines Wertes (`WSTRING`, Zeichenkette (16-Bit-Wide-Zeichen)) auf bis zu 5 nachgeschaltete Verbraucher, ausgewählt über einen Adapter-Index
- Routing in Modulen, die den Auswahlindex bereits als AUI-Adapterverbindung von einem vorgeschalteten Baustein erhalten (kein separates REQ/K-Paar nötig)
- Aufbau kaskadierbarer Verteiler-Netzwerke, bei denen der Index selbst wieder aus einem anderen AUI-Baustein stammt

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AIWS_DEMUX_5` (gleiche Verteillogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

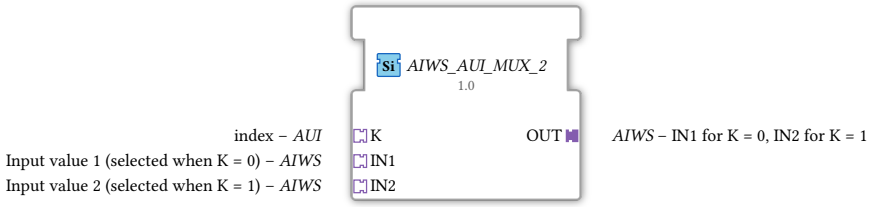
Vergleich mit E_DEMUX, das rein ereignisbasiert ohne Adapterkonzept demultiplext.

- **`AIWS_AUI_DEMUX_5`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AIWS_AUI_DEMUX_5_UNGATED` überträgt die Demultiplexer-Logik von `AIWS_DEMUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIWS_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AIWS_AUI_MUX_2` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `WSTRING`. Anders als `AIWS_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AIWS_AUI_MUX_2` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1/D1`.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (AIWS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (AIWS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AIWS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AIWS_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIWS_AUI_MUX_2 … AIWS_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (Zeichenkette (16-Bit-Wide-Zeichen)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AIWS_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

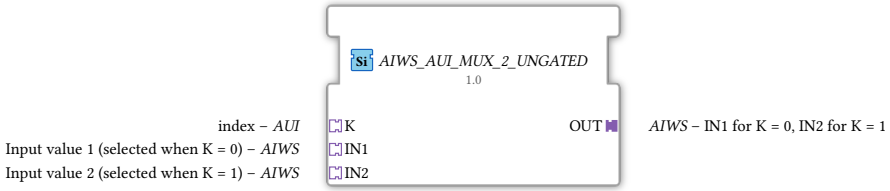
- **`AIWS_AUI_MUX_2_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AIWS_AUI_MUX_2` überträgt die Multiplexer-Logik von `AIWS_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIWS_AUI_MUX_2_UNGATED

i UNGATED-Variante: Dieser Baustein ist die ungegatete Version von `AIWS_AUI_MUX_2`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AIWS_AUI_MUX_2_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `WSTRING`. Anders als `AIWS_MUX_2` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AIWS_AUI_MUX_2_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 2, wird bei Index `K = 0` an `OUT` durchgereicht (AIWS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 2, wird bei Index `K = 1` an `OUT` durchgereicht (AIWS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AIWS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN2`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 2 Eingangs-Adapter (`IN1` … `IN2`) gerade ausgewählt ist.

2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AIWS_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIWS_AUI_MUX_2_UNGATED … AIWS_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN2`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 2 Signalquellen (Zeichenkette (16-Bit-Wide-Zeichen)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AIWS_MUX_2` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

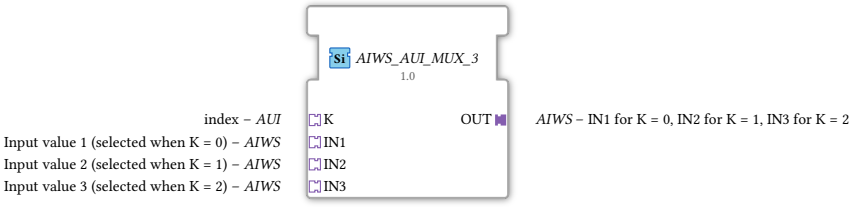
Vergleich mit F_MUX_2, das dieselbe 2:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AIWS_AUI_MUX_2`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AIWS_AUI_MUX_2_UNGATED` überträgt die Multiplexer-Logik von `AIWS_MUX_2` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIWS_AUI_MUX_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AIWS_AUI_MUX_3` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `WSTRING`. Anders als `AIWS_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirec-

tional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem AUI-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge
- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene AUI-Plug sein internes Ereignis E1 mit dem Datenwert D1 sendet, wertet AIWS_AUI_MUX_3 intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge
- Keine Ereignis-Ausgänge.

Daten-Eingänge
- Keine direkten Daten-Eingänge. Der Index (UINT, 0-basiert) kommt ausschließlich über den D1-Wert des am Socket **K** angeschlossenen AUI-Adapters.

Daten-Ausgänge
- Keine direkten Datenausgänge vorhanden.

Adapter
- **K** (Socket, Typ AUI): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis E1/D1.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index K = 0 an OUT durchgereicht (AIWS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index K = 1 an OUT durchgereicht (AIWS-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index K = 2 an OUT durchgereicht (AIWS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AIWS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters K (Typ AUI) als auch bei einem Ereignis an einem der Eingangs-Adapter IN1…IN3:

1. Der aktuelle Wert von K.D1 bestimmt, welcher der 3 Eingangs-Adapter (IN1 … IN3) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf OUT gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird OUT neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von K den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten
- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als AUI-Adapter vorliegt.
- Generische Implementierung (GEN_AIWS_AUI_MUX) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIWS_AUI_MUX_2 … AIWS_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von K.D1 neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter K oder IN1…IN3) ⟶ aktuell selektierten Eingang auslesen, OUT bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (Zeichenkette (16-Bit-Wide-Zeichen)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AIWS_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

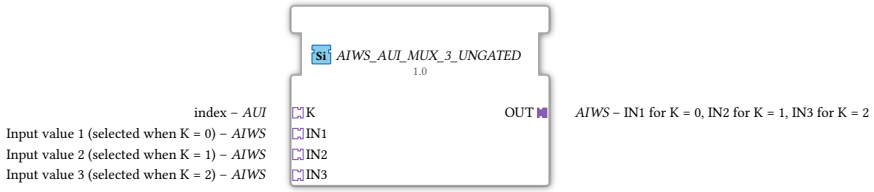
- **`AIWS_AUI_MUX_3_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AIWS_AUI_MUX_3` überträgt die Multiplexer-Logik von `AIWS_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIWS_AUI_MUX_3_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AIWS_AUI_MUX_3`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AIWS_AUI_MUX_3_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `WSTRING`. Anders als `AIWS_MUX_3` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AIWS_AUI_MUX_3_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 3, wird bei Index `K = 0` an `OUT` durchgereicht (AIWS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 3, wird bei Index `K = 1` an `OUT` durchgereicht (AIWS-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 3, wird bei Index `K = 2` an `OUT` durchgereicht (AIWS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AIWS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN3`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 3 Eingangs-Adapter (`IN1` … `IN3`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AIWS_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIWS_AUI_MUX_2 … AIWS_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN3`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 3 Signalquellen (Zeichenkette (16-Bit-Wide-Zeichen)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AIWS_MUX_3` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

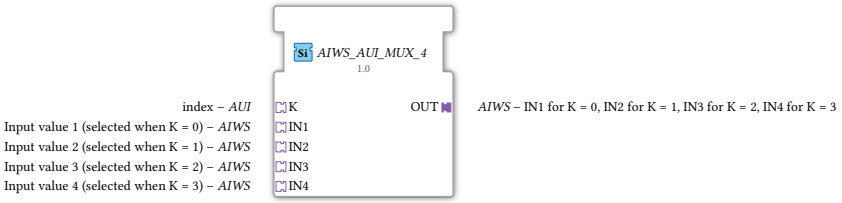
Vergleich mit F_MUX_3, das dieselbe 3:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AIWS_AUI_MUX_3`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AIWS_AUI_MUX_3_UNGATED` überträgt die Multiplexer-Logik von `AIWS_MUX_3` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIWS_AUI_MUX_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AIWS_AUI_MUX_4` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `WSTRING`. Anders als `AIWS_MUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AIWS_AUI_MUX_4` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index `K = 0` an `OUT` durchgereicht (AIWS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index `K = 1` an `OUT` durchgereicht (AIWS-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index `K = 2` an `OUT` durchgereicht (AIWS-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index `K = 3` an `OUT` durchgereicht (AIWS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AIWS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN4`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Eingangs-Adapter (`IN1` … `IN4`) gerade ausgewählt ist.

2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AIWS_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIWS_AUI_MUX_2 … AIWS_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (Zeichenkette (16-Bit-Wide-Zeichen)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AIWS_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

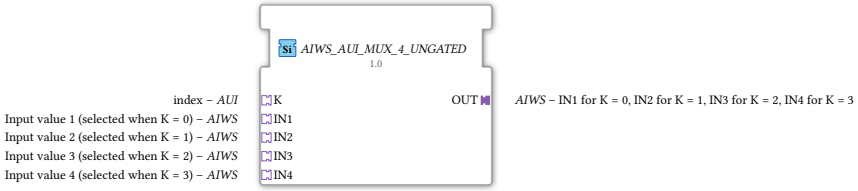
- **`AIWS_AUI_MUX_4_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AIWS_AUI_MUX_4` überträgt die Multiplexer-Logik von `AIWS_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIWS_AUI_MUX_4_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AIWS_AUI_MUX_4`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• — ○ • — ○ • — ○ •

Einleitung

Der `AIWS_AUI_MUX_4_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `WSTRING`. Anders als `AIWS_MUX_4` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AIWS_AUI_MUX_4_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 4, wird bei Index `K = 0` an `OUT` durchgereicht (AIWS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 4, wird bei Index `K = 1` an `OUT` durchgereicht (AIWS-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 4, wird bei Index `K = 2` an `OUT` durchgereicht (AIWS-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 4, wird bei Index `K = 3` an `OUT` durchgereicht (AIWS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AIWS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN4`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 4 Eingangs-Adapter (`IN1` … `IN4`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AIWS_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIWS_AUI_MUX_2 … AIWS_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN4`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 4 Signalquellen (Zeichenkette (16-Bit-Wide-Zeichen)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AIWS_MUX_4` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

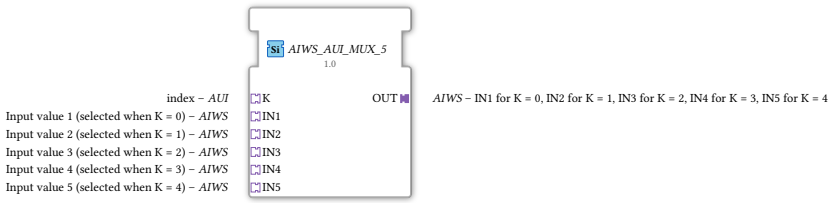
Vergleich mit F_MUX_4, das dieselbe 4:1-Auswahl rein datenbasiert ohne Adapter/Ereigniskonzept ausführt.

- **`AIWS_AUI_MUX_4`**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

Fazit

Der `AIWS_AUI_MUX_4_UNGATED` überträgt die Multiplexer-Logik von `AIWS_MUX_4` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIWS_AUI_MUX_5



• – ○ • – ○ • – ○ •

Einleitung

Der `AIWS_AUI_MUX_5` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `WSTRING`. Anders als `AIWS_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AIWS_AUI_MUX_5` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index `K = 0` an `OUT` durchgereicht (AIWS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index `K = 1` an `OUT` durchgereicht (AIWS-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index `K = 2` an `OUT` durchgereicht (AIWS-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index `K = 3` an `OUT` durchgereicht (AIWS-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index `K = 4` an `OUT` durchgereicht (AIWS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AIWS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN5`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Eingangs-Adapter (`IN1` … `IN5`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AIWS_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIWS_AUI_MUX_2 … AIWS_AUI_MUX_5).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (Zeichenkette (16-Bit-Wide-Zeichen)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AIWS_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

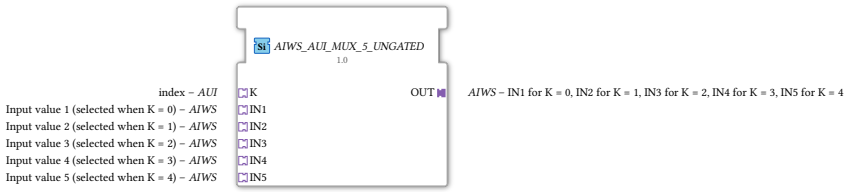
- **`AIWS_AUI_MUX_5_UNGATED`**: Ungegatete Variante – aktualisiert den Ausgang bei jedem Durchlauf, auch ohne Wertänderung.

Fazit

Der `AIWS_AUI_MUX_5` überträgt die Multiplexer-Logik von `AIWS_MUX_5` auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als `AUI`-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

AIWS_AUI_MUX_5_UNGATED

> **i UNGATED-Variante:** Dieser Baustein ist die ungegatete Version von `AIWS_AUI_MUX_5`. Er unterdrückt **keine** unveränderten Wiederholungen – jedes neu berechnete Ergebnis wird bedingungslos weitergegeben, auch ohne Wertänderung. Das ist wichtig für Verbraucher, die eine periodische Kadenz unabhängig von Wertänderung brauchen (z. B. Ableitungs-/Frequenzberechnungen, die sonst nicht gegen Null abklingen). Alle Angaben zu Änderungserkennung/Change-Gating weiter unten auf dieser Seite gelten **nicht** für diesen Baustein.



• – ○ • – ○ • – ○ •

Einleitung

Der `AIWS_AUI_MUX_5_UNGATED` ist die adapterbasierte Variante des generischen Multiplexers für den Datentyp `WSTRING`. Anders als `AIWS_MUX_5` erhält er den Auswahlindex nicht über ein REQ-Ereignis mit zugehörigem K-Dateneingang, sondern über einen eigenen Adapter-Socket **K** vom Typ `AUI` („Adapter Unidirectional Interface“). Das erlaubt es, den Index direkt aus einem anderen Baustein mit passendem `AUI`-Plug einzuspeisen, ohne eigene Verdrahtung von Ereignis- und Datenleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- Keine expliziten Ereignis-Eingänge am Baustein selbst. Der Index wird ausschließlich über den Adapter-Socket **K** empfangen: Sobald der dort angeschlossene `AUI`-Plug sein internes Ereignis `E1` mit dem Datenwert `D1` sendet, wertet `AIWS_AUI_MUX_5_UNGATED` intern den Index aus und löst die Verarbeitung aus.

Ereignis-Ausgänge

- Keine Ereignis-Ausgänge.

Daten-Eingänge

- Keine direkten Daten-Eingänge. Der Index (`UINT`, 0-basiert) kommt ausschließlich über den `D1`-Wert des am Socket **K** angeschlossenen `AUI`-Adapters.

Daten-Ausgänge

- Keine direkten Datenausgänge vorhanden.

Adapter

- **K** (Socket, Typ `AUI`): Index-Eingang – Auswahl des aktiven Ein-/Ausgangs über das adaptereigene Ereignis `E1`/`D1`.
- **IN1** (Socket): Eingangsadapter 1 von 5, wird bei Index `K = 0` an `OUT` durchgereicht (AIWS-Adaptertyp).
- **IN2** (Socket): Eingangsadapter 2 von 5, wird bei Index `K = 1` an `OUT` durchgereicht (AIWS-Adaptertyp).
- **IN3** (Socket): Eingangsadapter 3 von 5, wird bei Index `K = 2` an `OUT` durchgereicht (AIWS-Adaptertyp).
- **IN4** (Socket): Eingangsadapter 4 von 5, wird bei Index `K = 3` an `OUT` durchgereicht (AIWS-Adaptertyp).
- **IN5** (Socket): Eingangsadapter 5 von 5, wird bei Index `K = 4` an `OUT` durchgereicht (AIWS-Adaptertyp).
- **OUT** (Plug): Ausgangsadapter, gibt den über den Index ausgewählten Eingang weiter (AIWS-Adaptertyp).

Funktionsweise

Der Funktionsblock wertet den aktuellen Wert von **K** bei jedem eingehenden Ereignis neu aus – sowohl beim Ereignis des Selektor-Adapters `K` (Typ `AUI`) als auch bei einem Ereignis an einem der Eingangs-Adapter `IN1`…`IN5`:

1. Der aktuelle Wert von `K.D1` bestimmt, welcher der 5 Eingangs-Adapter (`IN1` … `IN5`) gerade ausgewählt ist.
2. Der Datenwert dieses ausgewählten Eingangs wird mit dem aktuell auf `OUT` gehaltenen Wert verglichen. Nur bei tatsächlicher Änderung wird `OUT` neu beschrieben und dessen Adapter-Event gesendet (siehe „Änderungserkennung“ unten).

Dadurch zieht auch eine reine Änderung von `K` den Ausgang sofort nach, selbst wenn sich der Datenwert des neu ausgewählten Eingangs seit dessen letztem eigenen Ereignis nicht verändert hat.

Technische Besonderheiten

- Adapterbasierter Indexeingang statt klassischem REQ/K-Eingangspaar – reduziert die Verdrahtung, wenn der Index bereits als `AUI`-Adapter vorliegt.
- Generische Implementierung (`GEN_AIWS_AUI_MUX`) – gemeinsam für alle Portanzahlen dieser Bausteinfamilie (AIWS_AUI_MUX_2 … AIWS_AUI_MUX_5_UNGATED).
- **Änderungserkennung**: Der jeweils beschriebene Ausgangs-Plug wird nur aktualisiert und sein Adapter-Event nur gesendet, wenn sich der neue Wert vom aktuell gehaltenen Wert unterscheidet. Bleibt der Wert gleich, bleibt auch das Adapter-Event aus – so werden überflüssige Updates bei nachgeschalteten Peers vermieden.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, sondern wertet bei jedem eingehenden Ereignis den aktuellen Wert von `K.D1` neu aus:

- **Beliebiges Ereignis** (Selektor-Adapter `K` oder `IN1`…`IN5`) ⟶ aktuell selektierten Eingang auslesen, `OUT` bei Wertänderung aktualisieren und dessen Adapter-Event senden.

Anwendungsszenarien

- Auswahl zwischen bis zu 5 Signalquellen (Zeichenkette (16-Bit-Wide-Zeichen)) über einen zentral verwalteten Adapter-Index
- Ersatz für ein REQ/K-Eingangspaar durch eine einzelne AUI-Adapterverbindung, wenn der Index bereits von einem anderen Baustein als Adapter bereitgestellt wird
- Aufbau von Auswahlnetzwerken, in denen mehrere MUX-Bausteine denselben Index-Adapter teilen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit `AIWS_MUX_5` (gleiche Auswahllogik, Index jedoch klassisch über **REQ**-Ereignis + **K**-Dateneingang statt über einen Adapter).

Für die Anzahl der Eingänge existiert kein datenbasiertes `F_MUX`-Pendant in der `iec61131-3`-Bibliothek.

- **AIWS_AUI_MUX_5**: Die gegatete Variante – aktualisiert den Ausgang nur bei tatsächlicher Wertänderung.

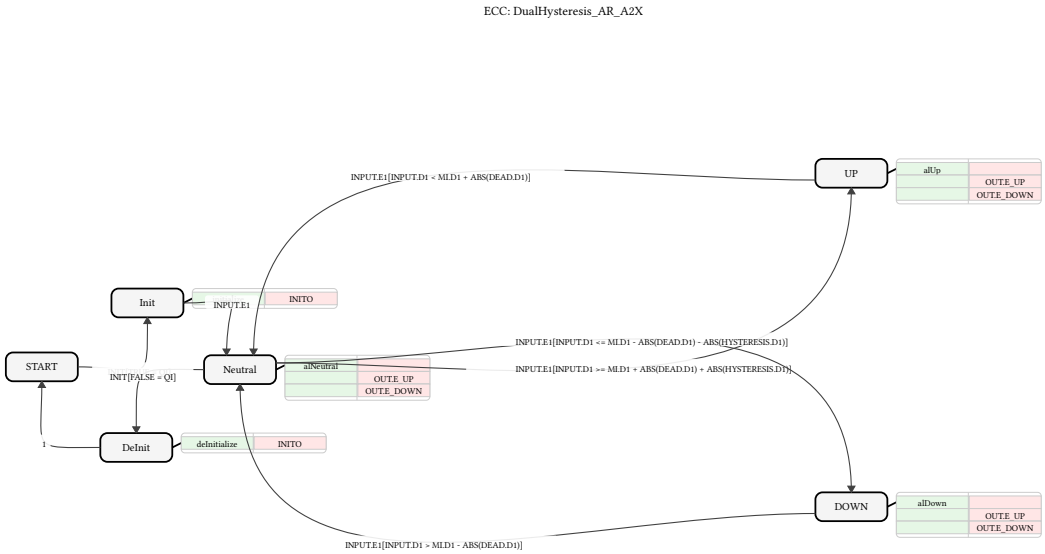
Fazit

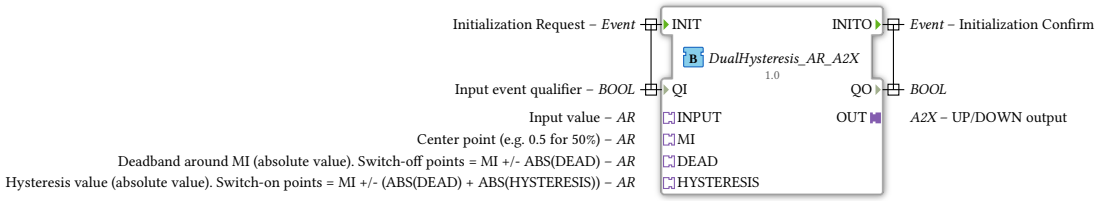
Der AIWS_AUI_MUX_5_UNGATED überträgt die Multiplexer-Logik von AIWS_MUX_5 auf eine rein adapterbasierte Indexversorgung. Das macht ihn zur passenden Wahl, wenn der Auswahlindex bereits als AUI-Adapter aus einem anderen Baustein zur Verfügung steht und keine zusätzliche Ereignis-/Datenverdrahtung für den Index gewünscht ist.

2.1.11.15 Signalprocessing

Hysteresis

DualHysteresis_AR_A2X





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **DualHysteresis_AR_A2X** realisiert eine zweistufige Analog-Digital-Umwandlung mit Hysterese und Totzone. Er vergleicht einen analogen Eingangswert INPUT kontinuierlich mit einem konfigurierbaren Mittelpunkt MI, einer Totzone DEAD und einer Hysterese HYSTERESIS. Basierend auf dem Vergleich werden zwei binäre Ausgangssignale UP und DOWN erzeugt, die über den unidirektionalen Adapter OUT (Typ A2X) ausgegeben werden. Diese Signale können beispielsweise zur Ansteuerung von Aktoren oder zur Überwachung von Prozesswerten dienen, wobei die Hysterese ein häufiges Umschalten bei kleinen Schwankungen verhindert.

Der Baustein ist als grundlegender FB (BasicFB) mit einer ereignisgesteuerten Zustandsmaschine (ECC) implementiert und erfüllt die Anforderungen der Norm IEC 61499-2.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung
INIT	EInit	Initialisierungsanforderung (Aufruf mit QI).

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung
INITO	EInit	Bestätigung der Initialisierung/Deinitialisierung.

Daten-Eingänge

Variable	Typ	Beschreibung
QI	BOOL	Eingangsqualifizierer: Aktiviert die normale Funktionsweise (TRUE = aktiv).

Daten-Ausgänge

Variable	Typ	Beschreibung
QO	BOOL	Ausgangsqualifizierer: Übernimmt den Wert von QI bei normaler Operation.

Adapter

Sockets (Eingänge):

Adapter	Typ	Beschreibung
INPUT	AR	Analoger Eingangswert, der überwacht wird.
MI	AR	Mittelpunkt (z. B. 0,5 für 50 %).
DEAD	AR	Totbreite um MI (absoluter Wert). Ausschaltpunkte: MI ± |DEAD|.
HYSTERESIS	AR	Hysterese (absoluter Wert). Einschaltpunkte: MI ± (|DEAD| + |HYSTERESIS|).

Plugs (Ausgänge):

Adapter	Typ	Beschreibung
OUT	A2X	Unidirektionaler Ausgang mit den Bool-Werten UP und DOWN. Zusätzlich werden Ereignisse E_UP und E_DOWN ausgelöst.

Funktionsweise

Der Funktionsblock durchläuft nach der Initialisierung einen sicheren Neutralzustand. Je nach analogen Eingangswert INPUT und den Parametern MI, DEAD und HYSTERESIS wird in den Zustand UP oder DOWN gewechselt. Die Schaltbedingungen sind:

- **Einschalten UP** (Zustandsübergang von Neutral nach UP):
  `INPUT >= MI + |DEAD| + |HYSTERESIS|`
- **Ausschalten UP** (Übergang von UP nach Neutral):
  `INPUT < MI + |DEAD|` (strikt)
- **Einschalten DOWN** (Übergang von Neutral nach DOWN):
  `INPUT <= MI - |DEAD| - |HYSTERESIS|`
- **Ausschalten DOWN** (Übergang von DOWN nach Neutral):
  `INPUT > MI - |DEAD|` (strikt)

In jedem Zustand wird der Ausgang QO auf den Wert von QI gesetzt. Liegt QI auf TRUE, wird der entsprechende Ausgang (UP oder DOWN) aktiviert; andernfalls werden alle Ausgänge auf FALSE gesetzt (sicherer Zustand). Die Ausgabe erfolgt über den Adapter OUT mit den zugehörigen Ereignissen OUT.E_UP und OUT.E_DOWN.

Die Initialisierung erfolgt über das Ereignis INIT mit QI = TRUE. Dabei werden alle Ausgänge auf FALSE gesetzt und der Baustein wechselt in den Neutral-Zustand. Eine Deinitialisierung wird durch INIT mit QI = FALSE ausgelöst.

Technische Besonderheiten

- **Verwendung von Absolutwerten**: Die Parameter DEAD und HYSTERESIS werden intern als Absolutwerte interpretiert, sodass auch negative Eingangswerte korrekt verarbeitet werden.
- **Hysterese & Totzone**: Die Kombination aus Totzone und Hysterese verhindert ein schnelles Oszillieren der Ausgänge bei kleinen Änderungen in der Nähe der Schaltpunkte.
- **Sicherheitszustand**: Solange QI = FALSE ist, werden alle Ausgänge auf FALSE gehalten (OUT.UP und OUT.DOWN), unabhängig vom Vergleichsergebnis. Dies erhöht die Zuverlässigkeit in sicherheitskritischen Anwendungen.
- **Adapterbasierte Kommunikation**: Die Ein- und Ausgänge sind als unidirektionale Adapter (AR und A2X) definiert, was eine flexible Verkabelung innerhalb von 4diac-Projekten ermöglicht.
- **Ereignisgesteuerte Zustandsmaschine**: Der FB verwendet ein klassisches ECC-Modell, das deterministisches Verhalten garantiert.

Zustandsübersicht

Zustand	Beschreibung
START	Initialer Ruhezustand nach dem Einschalten.
Init	Führt die Initialisierung aus (setzt Ausgänge auf FALSE) und bestätigt mit INITO.
Neutral	Normalbetrieb, keine aktiven Ausgänge (UP/DOWN = FALSE).
UP	Aktiver Zustand: OUT.UP = TRUE und OUT.DOWN = FALSE.
DOWN	Aktiver Zustand: OUT.DOWN = TRUE und OUT.UP = FALSE.
DeInit	Deinitialisierung, setzt alle Ausgänge auf FALSE und bestätigt mit INITO.

Die Übergänge zwischen den Zuständen sind in der ECC definiert und basieren auf den oben genannten Bedingungen. Nach der Deinitialisierung kehrt der FB in den START-Zustand zurück.

Anwendungsszenarien

- **Prozessüberwachung**: Überwachung eines analogen Sensors (z. B. Temperatur, Druck, Füllstand) und Erzeugung von Schaltsignalen bei Überschreiten/Unterschreiten von Grenzwerten mit Hysterese, um Pumpen oder Ventile anzusteuern.
- **Agrartechnik**: Steuerung von landwirtschaftlichen Maschinen, z. B. Erkennung von Positionen oder Füllständen mit Toleranzbereichen.
- **Gebäudeautomation**: Regelung von Heiz-/Kühlsystemen basierend auf Temperaturschwellwerten.
- **Industrielle Automatisierung**: Signalumwandlung in digitale Stellsignale für andere Funktionsblöcke, wobei die Hysterese ein häufiges Umschalten vermeidet.

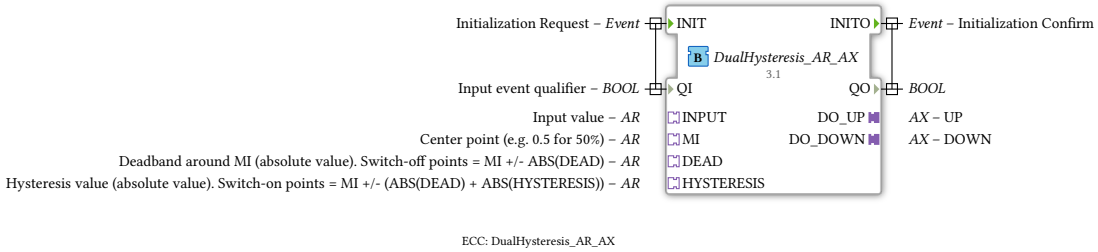
Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen Komparator-Bausteinen (z. B. einem Schwellwertschalter) bietet DualHysteresis_AR_A2X eine deutlich höhere Robustheit gegenüber Rauschen und kleinen Wertschwankungen. Während ein einfacher Komparator direkt bei Erreichen eines Grenzwerts umschaltet, sorgt die Hysterese dafür, dass erst ein deutlicher Über- bzw. Unterschreitung erfolgen muss. Zudem bietet dieser FB zwei getrennte Ausgänge (UP und DOWN), wodurch sowohl Über- als auch Unterschreitungen eines Bereichs erkannt werden können – ähnlich einem Fensterkomparator, jedoch mit einstellbarer Totzone und Hysterese.

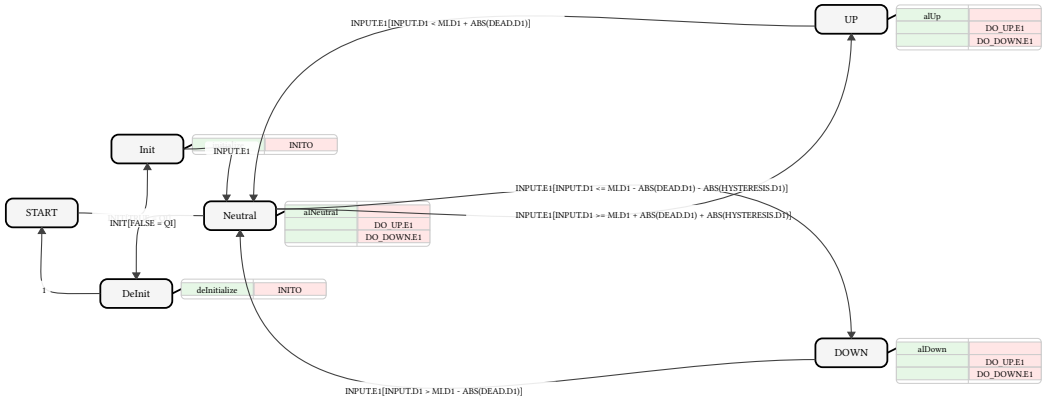
Fazit

Der Funktionsblock DualHysteresis_AR_A2X ist ein vielseitiger Baustein für industrielle und agrarische Anwendungen, bei denen ein analoges Signal in digitale Schaltbefehle umgewandelt werden muss. Durch die konfigurierbare Hysterese und Totzone können zuverlässige und stabile Schaltvorgänge realisiert werden. Die klare Struktur und die ereignisgesteuerte Implementierung machen ihn einfach in bestehende 4diac-Projekte integrierbar. Seine Eigenschaften eignen sich besonders für Regelungs- und Überwachungsaufgaben, bei denen ein präzises, aber gleichzeitig robustes Schaltverhalten gefordert ist.

DualHysteresis_AR_AX



ECC: DualHysteresis_AR_AX



(Kein Bild vorhanden)

• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsbaustein `DualHysteresis_AR_AX` dient der zweitwegigen (dualen) Analog-Digital-Wandlung mit Hystereseverhalten. Er vergleicht ein analoges Eingangssignal mit einem definierten Mittelpunkt (`MI`), um über zwei separate Digitalausgänge (UP und DOWN) Steuerbefehle auszugeben. Dies entspricht der Funktionsweise eines klassischen Dreipunktreglers mit Totband und Hystereseschwellen. Der Baustein ist für die Signalverarbeitung im Rahmen von Regelungs- und Steuerungsaufgaben konzipiert und nutzt Adapter für eine strukturierte Kopplung der Signale.

Schnittstellenstruktur

Ereignis-Eingänge

- `INIT` (Typ: `EInit`): Initialisierungsanfrage. Startet oder stoppt den Baustein in Abhängigkeit des Daten-Eingangs `QI`.

Ereignis-Ausgänge

- `INITO` (Typ: `EInit`): Initialisierungsbestätigung. Signalisiert den erfolgreichen Übergang in den initialisierten bzw. deinitialisierten Zustand.

Daten-Eingänge

- `QI` (Typ: `BOOL`): Eingangshilfsvariable (Input Event Qualifier). Bestimmt, ob der Baustein aktiv arbeiten soll (`TRUE`) oder in den sicheren Deinitialisierungszustand versetzt wird (`FALSE`).

Daten-Ausgänge

- `QO` (Typ: `BOOL`): Ausgangshilfsvariable (Output Event Qualifier). Spiegelt den aktuellen Status und die Betriebsbereitschaft des Bausteins wider.

Adapter

- **Plugs (Stecker):**
  - `DO_UP` (Typ: `adapter::types::unidirectional::AX`): Ausgangsadapter für die Aufwärtsbewegung / den oberen Schwellwert (UP).
  - `DO_DOWN` (Typ: `adapter::types::unidirectional::AX`): Ausgangsadapter für die Abwärtsbewegung / den unteren Schwellwert (DOWN).
- **Sockets (Buchsen):**
  - `INPUT` (Typ: `adapter::types::unidirectional::AR`): Der aktuelle analoge Mess- oder Eingangswert.
  - `MI` (Typ: `adapter::types::unidirectional::AR`): Der Mittelpunkt bzw. Sollwert der Regelung (Center Point, z. B. `0.5` für 50%).
  - `DEAD` (Typ: `adapter::types::unidirectional::AR`): Das Totband um den Mittelpunkt (Absolutwert). Definiert die Ausschaltpunkte.
  - `HYSTERESIS` (Typ: `adapter::types::unidirectional::AR`): Die Hysterese (Absolutwert). Definiert zusammen mit dem Totband die Einschaltpunkte.

Funktionsweise

Die Zustandsumschaltung erfolgt ereignisgesteuert über den Eingang `INPUT.E1`. Die mathematische Auswertung stellt sicher, dass negative Parametereingaben für das Totband (`DEAD`) und die Hysterese (`HYSTERESIS`) durch die Verwendung von Absolutwerten (`ABS`) abgefangen werden.

Die logischen Schwellenwerte verhalten sich wie folgt:

- **Einschalten UP (Wechsel nach UP):**

$$INPUT \geq MI + |DEAD| + |HYSTERESIS| \tag{2.134}$$

(Ausgänge: `DO_UP.D1 = TRUE`, `DO_DOWN.D1 = FALSE`)

- **Ausschalten UP (Wechsel zurück nach Neutral):**

$$INPUT < MI + |DEAD| \tag{2.135}$$

(Ausgänge: `DO_UP.D1 = FALSE`, `DO_DOWN.D1 = FALSE`)

- **Einschalten DOWN (Wechsel nach DOWN):**

$$INPUT \leq MI - |DEAD| - |HYSTERESIS| \tag{2.136}$$

(Ausgänge: `DO_UP.D1 = FALSE`, `DO_DOWN.D1 = TRUE`)

- **Ausschalten DOWN (Wechsel zurück nach Neutral):**

$$INPUT > MI - |DEAD| \tag{2.137}$$

(Ausgänge: `DO_UP.D1 = FALSE`, `DO_DOWN.D1 = FALSE`)

Technische Besonderheiten

- **Fehlersicherheit durch Absolutwerte:** Die Schwellenwertberechnungen nutzen explizit die `ABS`-Funktion für die Parameter `DEAD` und `HYSTERESIS`. Dies verhindert Fehlverhalten bei versehentlich negativ parametrisierten Werten.
- **Sicherer Zustand:** Im deinitialisierten Zustand (`QI = FALSE`) oder im Zustand `Neutral` werden beide Digitalausgänge (`DO_UP.D1` und `DO_DOWN.D1`) zwingend auf `FALSE` gesetzt.
- **Adapter-Anbindung:** Die Verwendung von `AR` (Analog Receiver) und `AX` (Digital Actuator) Adaptern sorgt für eine standardisierte, saubere Kapselung der Signalwege innerhalb der 4diac-IDE.

Zustandsübersicht

Der Baustein basiert auf einer Zustandsmaschine (ECC) mit folgenden Zuständen:

1. `START`: Ruhezustand vor der ersten Initialisierung.
2. `Init`: Initialisierungsphase. Setzt die Ausgänge in den sicheren Zustand.
3. `Neutral`: Der Eingangswert befindet sich innerhalb des zulässigen Bereichs. Beide Ausgänge sind inaktiv.
4. `UP`: Der obere Einschaltpunkt wurde überschritten. Der Ausgang `DO_UP` ist aktiv.
5. `DOWN`: Der untere Einschaltpunkt wurde unterschritten. Der Ausgang `DO_DOWN` ist aktiv.
6. `DeInit`: Übergangszustand zur Deaktivierung des Bausteins.

Anwendungsszenarien

- **Dreipunktregler:** Temperaturregelungen (z. B. Heizen / Neutral / Kühlen) oder Füllstandsregelungen (z. B. Zulaufventil öffnen / Neutral / Ablaufventil öffnen).
- **Spurführungs- und Lenksysteme:** Automatische Lenkungen in der Landtechnik oder Logistik, bei denen eine Abweichung von der Spurmitte (`MI`) nach links (`DOWN`) oder rechts (`UP`) korrigiert werden muss.
- **Sollwert-Bandüberwachung:** Überwachung physikalischer Größen, bei denen erst bei signifikanten Abweichungen (Totband + Hysterese) ein Aktor angesteuert werden soll, um ständiges „Flattern“ des Aktors zu vermeiden.

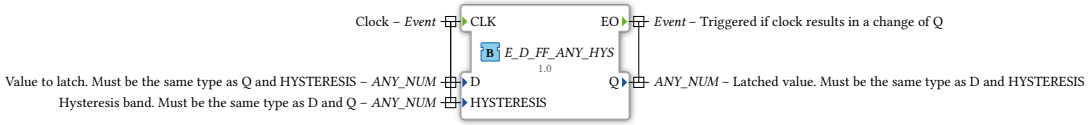
Vergleich mit ähnlichen Bausteinen

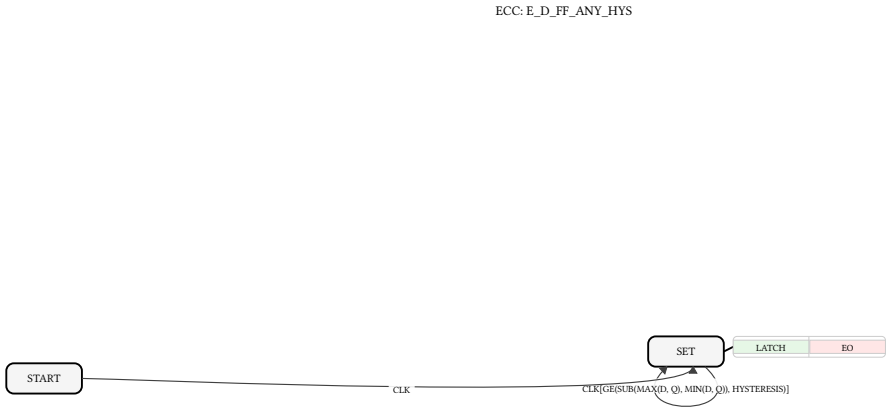
Klassische Hysteresebausteine (z. B. Zweipunktregler) steuern lediglich einen einzelnen Digitalausgang an. Der `DualHysteresis_AR_AX` vereint zwei entgegengesetzte Hystereseschleifen in einem einzigen Baustein (Dreipunktverhalten). Zudem zeichnet er sich im Vergleich zu Standard-FBs mit elementaren Datentypen durch die Verwendung von unidirektionalen Adaptern (`AR` und `AX`) aus, was die Verdrahtung in komplexen Applikationen vereinfacht.

Fazit

Der `DualHysteresis_AR_AX` ist ein robuster und flexibler Funktionsbaustein zur Realisierung von Dreipunkt-Regelungen. Durch die mathematische Absicherung der Grenzwerte und das klar definierte Zustandsverhalten eignet er sich hervorragend für anspruchsvolle Steuerungsaufgaben, bei denen Aktoren materialschonend und flatterfrei angesteuert werden müssen.

E_D_FF_ANY_HYS





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein `E_D_FF_ANY_HYS` ist ein flankengetriggertes D-Flip-Flop (Data Latch) für numerische Datentypen mit integrierter Hysterese. Er dient dazu, einen Eingangswert zu speichern (zu „latchen“) und am Ausgang bereitzustellen. Eine Aktualisierung des Ausgangs und das Auslösen des Ausgangsereignisses erfolgen bei einem Taktsignal jedoch nur dann, wenn die Änderung des Eingangswerts im Vergleich zum aktuellen Ausgangswert die definierte Hysterese-Schwelle erreicht oder überschreitet.

Schnittstellenstruktur

Ereignis-Eingänge

- **CLK**: Takt-Eingang (Clock). Bei Eintreffen dieses Ereignisses wird geprüft, ob sich der Eingangswert `D` im Vergleich zum aktuellen Ausgangswert `Q` um mindestens den Wert von `HYSTERESIS` geändert hat.

Ereignis-Ausgänge

- **EO**: Ereignis-Ausgang (Event Out). Wird ausgelöst, wenn ein Takt-Ereignis (`CLK`) zu einer tatsächlichen Wertänderung und Aktualisierung des Ausgangs `Q` geführt hat.

Daten-Eingänge

- **D** (`ANY_NUM`): Der aktuelle, einzulesende Datenwert.
- **HYSTERESIS** (`ANY_NUM`): Das Hysterese-Band. Gibt die Mindestdifferenz an, die zwischen `D` und `Q` liegen muss, damit der Ausgang aktualisiert wird.

Daten-Ausgänge

- **Q** (`ANY_NUM`): Der gespeicherte (gelatchte) Wert.

Adapter

- *Keine Adapter vorhanden.*

Funktionsweise

Der Funktionsbaustein arbeitet intern mit einer Execution Control Chart (ECC) und einem einfachen Zuweisungs-Algorithmus:

1. **Initialisierung**: Beim ersten Start befindet sich der Baustein im Zustand `START`. Ein erstes `CLK`-Ereignis führt direkt in den Zustand `SET`. Dabei wird der Algorithmus `LATCH` ausgeführt, welcher den aktuellen Wert von `D` auf den Ausgang `Q` schreibt. Gleichzeitig wird das Ausgangsereignis `EO` gesendet.
2. **Hysterese-Prüfung bei Folge-Ereignissen**: Befindet sich der Baustein im Zustand `SET`, führt jedes weitere `CLK`-Ereignis nur dann zu einer erneuten Ausführung von `SET` (und damit zur Aktualisierung von `Q` und dem Feuern von `EO`), wenn folgende Bedingung erfüllt ist:

$$| D - Q | \geq \text{HYSTERESIS} \tag{2.138}$$

   In der ECC-Bedingung wird dies mathematisch über die Formel `GE(SUB(MAX(D, Q), MIN(D, Q)), HYSTERESIS)` realisiert. Dies entspricht der absoluten Differenz zwischen dem Eingangswert `D` und dem aktuell gespeicherten Wert `Q`.
3. Ist die Differenz kleiner als der Wert von `HYSTERESIS`, wird das Ereignis ignoriert, der Ausgang `Q` bleibt unverändert und es wird kein `EO`-Ereignis ausgelöst.

Technische Besonderheiten

- **Generischer Datentyp**: Durch die Verwendung des Datentyps `ANY_NUM` für die Ein- und Ausgänge ist der Baustein hochgradig wiederverwendbar und kann mit verschiedenen numerischen Datentypen (z. B. `INT`, `REAL`, `LREAL`) verschaltet werden.
- **Effiziente Differenzberechnung**: Die Differenzermittlung über `SUB(MAX(D, Q), MIN(D, Q))` stellt sicher, dass immer ein positiver Absolutwert für den Vergleich mit der Hysterese herangezogen wird, unabhängig davon, ob der neue Wert größer oder kleiner als der alte Wert ist.

Zustandsübersicht

```
  [ START ] --( CLK )--> [ SET (Algorithmus: LATCH, Ausgang: EO) ]
                              |
                              +--( CLK [Differenz >= HYSTERESIS] )--> (erneuter Aufruf von
SET)
```

- **START**: Wartezustand vor dem ersten Takt.
- **SET**: Zustand, in dem der Eingangswert übernommen und ausgegeben wird. Wird bei jedem gültigen Takt (unter Berücksichtigung der Hysterese) zyklisch wieder aufgerufen.

Anwendungsszenarien

- **Rauschfilterung für analoge Sensorwerte**: Reduzierung von Event-Fluten in verteilten Systemen. Wenn ein Sensor (z. B. Temperatursensor) kontinuierlich leicht schwankende Werte liefert, verhindert die Hysterese, dass bei minimalen Änderungen ständig neue Steuerungsereignisse im System verteilt werden.
- **Grenzwertüberwachung mit Totband**: Vermeidung von prellenden Signalen bei der Übertragung von Prozesswerten an Visualisierungen (HMI) oder Datenbanken.

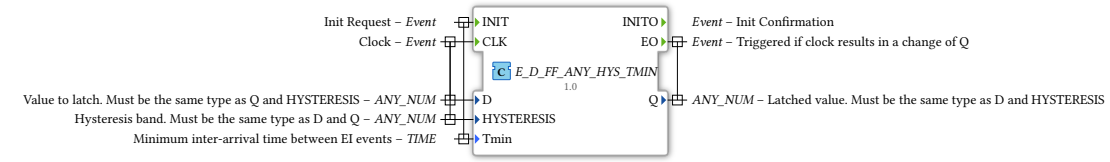
Vergleich mit ähnlichen Bausteinen

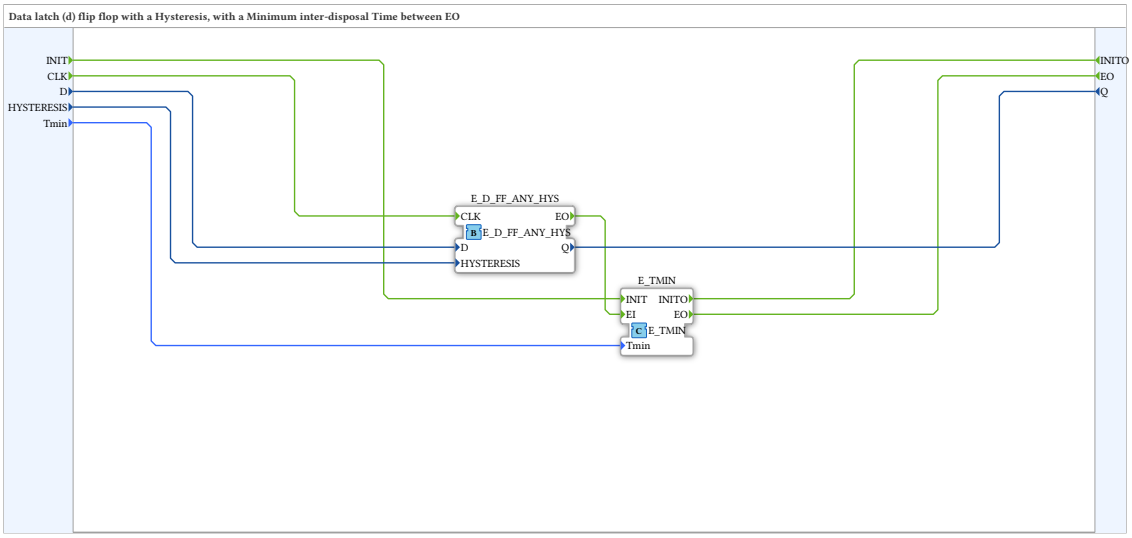
- **E_D_FF**: Das Standard-D-Flip-Flop reagiert meist auf boolescher Ebene oder speichert Werte bei jedem Takt bedingungslos ab. `E_D_FF_ANY_HYS` filtert die Werte zusätzlich durch die Hysterese-Bedingung.
- **Hysterese-Bausteine (z.B. mit booleschem Ausgang)**: Typische Hysterese-Bausteine vergleichen einen Wert mit festen Schwellenwerten und liefern ein `BOOL`-Signal (z. B. Schwellwertschalter). `E_D_FF_ANY_HYS` hingegen gibt den numerischen Wert selbst weiter, sobald dieser sich signifikant geändert hat.

Fazit

Der `E_D_FF_ANY_HYS` ist ein extrem nützlicher Baustein zur daten- und ereignisgesteuerten Signalverarbeitung. Er verbindet die Speicherfunktion eines D-Flip-Flops mit einer intelligenten Rausch- und Änderungsfilterung für numerische Signale und trägt so maßgeblich zur Optimierung der Performance und Reduzierung der Event-Last in IEC 61499 Anwendungen bei.

E_D_FF_ANY_HYS_TMIN





• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsbaustein `E_D_FF_ANY_HYS_TMIN` ist ein zusammengesetzter Funktionsbaustein (Composite Function Block) für IEC 61499-basierte Systeme. Er realisiert ein flankengetriggertes D-Flipflop (Data Latch) mit einer integrierten Schwellwert-Hysterese und einer zeitlichen Mindestabstandsbegrenzung (`Tmin`) für ausgehende Ereignisse.

Der Baustein dient dazu, kontinuierliche oder hochfrequente Änderungen eines numerischen Eingangssignals zu filtern. Ein neues Ereignis am Ausgang wird nur dann erzeugt, wenn die Signaländerung die definierte Hysterese überschreitet und gleichzeitig ein definierter Mindestzeitraum seit dem letzten Ausgangsereignis vergangen ist. Dies verhindert effektiv ein sogenanntes „Event Flooding“ (Ereignis-Überlastung) in verteilten Steuerungssystemen.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung	Zugehörige Daten
INIT	EInit	Initialisiert den Baustein und setzt die Mindestzeitbegrenzung.	`Tmin`
CLK	Event	Taktsignal (Clock). Bei Eintreffen dieses Ereignisses wird der Wert am Eingang `D` ausgewertet.	`D`, `HYSTERESIS`

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung	Zugehörige Daten
INITO	EInit	Bestätigung der erfolgreichen Initialisierung.	Keine
EO	Event	Wird ausgelöst, wenn ein CLK-Ereignis eine gültige Änderung des Ausgangswerts `Q` bewirkt hat und die Mindestzeit `Tmin` abgelaufen ist.	`Q`

Daten-Eingänge

Variable	Typ	Beschreibung
D	ANY_NUM	Der zu speichernde Eingangswert (Data).
HYSTERESIS	ANY_NUM	Das Hystereseband. Bestimmt, wie stark sich `D` vom aktuellen Ausgangswert `Q` unterscheiden muss, um eine Änderung zu bewirken.
Tmin	TIME	Die minimale Zeitspanne, die zwischen zwei aufeinanderfolgenden `EO`-Ereignissen vergehen muss.

Daten-Ausgänge

Variable	Typ	Beschreibung
Q	ANY_NUM	Der gespeicherte und gefilterte Ausgangswert (Latched Value).

Adapter

Dieser Funktionsbaustein verwendet keine Adapter.

Funktionsweise

Der Baustein kombiniert intern zwei spezialisierte Funktionsbausteine:

1. Einen Hysterese-Speicherbaustein (`E_D_FF_ANY_HYS`)
2. Einen zeitlichen Ereignisbegrenzer (`E_TMIN`)

Sobald ein Ereignis am Eingang `CLK` eintrifft, liest der interne Baustein `E_D_FF_ANY_HYS` den aktuellen Wert von `D` ein.

- Liegt die Differenz zwischen dem neuen Wert `D` und dem aktuell gespeicherten Wert `Q` außerhalb des durch `HYSTERESIS` definierten Bereichs, wird `Q` mit dem neuen Wert aktualisiert und ein internes Ereignis an den Begrenzer `E_TMIN` weitergeleitet.
- Liegt die Änderung innerhalb des Hysteresebands, bleibt der Wert unverändert und es wird kein Ereignis weitergeleitet.

Der Baustein `E_TMIN` überwacht die Weiterleitung dieses Ereignisses an den Ausgang `EO`. Er stellt sicher, dass zwischen zwei `EO`-Ereignissen mindestens die Zeitspanne `Tmin` verstreicht. Schneller aufeinanderfolgende Ereignisse werden entsprechend verzögert oder gefiltert, um die nachfolgende Logik vor zu hoher Ereignislast zu schützen.

Technische Besonderheiten

- **Generische Datentypen (`ANY_NUM`):** Durch die Verwendung des generischen Datentyps `ANY_NUM` für `D`, `HYSTERESIS` und `Q` ist der Baustein hochgradig wiederverwendbar und kann für verschiedene numerische Typen (z. B. `INT`, `REAL`, `LREAL`) eingesetzt werden.
- **Kombinierte Filterung:** Der Baustein vereint Amplitudenfilterung (Hysterese) und Zeitfilterung (Tmin) in einer einzigen Komponente.

Zustandsübersicht

1. **Uninitialisiert:** Der Baustein wartet auf das Ereignis `INIT`.
2. **Initialisiert / Bereit:** Der Baustein hat `Tmin` konfiguriert und wartet auf Ereignisse am Eingang `CLK`.
3. **Wertänderung blockiert (Hysterese):** Ein `CLK`-Ereignis trifft ein, aber die Änderung von `D` ist zu gering. `Q` bleibt gleich, kein `EO`.
4. **Wertänderung aktiv (Zeitverzögerung/Filterung):** Ein `CLK`-Ereignis trifft ein und die Änderung ist groß genug. Wenn `Tmin` abgelaufen ist, wird `Q` aktualisiert und `EO` sofort ausgegeben. Falls `Tmin` noch aktiv ist, greift die Schutzfunktion des Zeitglieds.

Anwendungsszenarien

- **Analoge Sensorwertverarbeitung:** Dämpfung und Rauschunterdrückung bei verrauschten analogen Messwerten (z. B. Temperatur, Druck oder Füllstand).
- **Netzwerkschonende Datenübertragung:** Reduzierung von Publish-Ereignissen in IoT- oder SCADA-Systemen (z. B. MQTT). Daten werden nur gesendet, wenn sich ein Wert signifikant geändert hat, jedoch maximal alle „X“ Sekunden.

- **Ressourceneffizienz in SPS-Steuerungen:** Entlastung der CPU von zyklischen Berechnungen, die durch hochfrequente Signaländerungen getriggert würden.

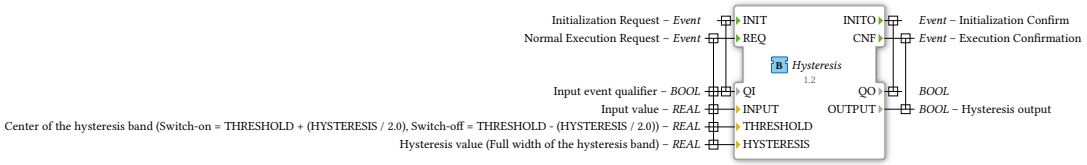
Vergleich mit ähnlichen Bausteinen

- **`E_D_FF`:** Speichert Werte direkt bei jedem Takt ohne Hysterese und Zeitbegrenzung. Führt bei unruhigen Signalen zu extrem hoher Ereignisdichte.
- **`E_D_FF_ANY_HYS`:** Besitzt zwar die Hysteresefunktion, bietet jedoch keinen Schutz vor schnellen, kontinuierlichen Signaländerungen (z. B. bei schnellen Rampen), die dennoch zu viele Ereignisse erzeugen würden. `E_D_FF_ANY_HYS_TMIN` löst dieses Problem durch die integrierte `Tmin`-Komponente.

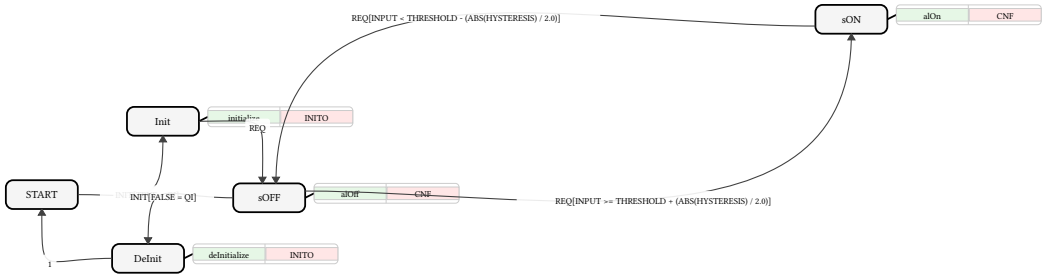
Fazit

Der `E_D_FF_ANY_HYS_TMIN` ist ein äußerst nützlicher Baustein zur Signalaufbereitung im industriellen Umfeld. Er verbindet eine wertbasierte Filterung mit einer zeitbasierten Ratenbegrenzung. Dadurch eignet er sich ideal für stabile Steuerungsarchitekturen, bei denen Sensorwerte präzise erfasst, aber nachfolgende Systeme vor Überlastung geschützt werden müssen.

Hysteresis



ECC: Hysteresis



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein `Hysteresis` dient als analog-digitaler Umsetzer mit integrierter Hysteresekurve. Er wandelt ein analoges Eingangssignal (`INPUT`) in ein binäres Ausgangssignal (`OUTPUT`) um. Durch die Definition eines Schwellenwerts (`THRESHOLD`) und einer Hysteresebreite (`HYSTERESIS`) wird ein unerwünschtes Prellen oder schnelles Oszillieren des Ausgangs bei verrauschten Eingangssignalen effektiv verhindert.

• – ○ • – ○ • – ○ •

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisiert den Funktionsbaustein. Aktualisiert den Ausgangs-Qualifier `QO` basierend auf `QI`.
- **REQ**: Löst eine Berechnung des Hysteresezustands basierend auf den aktuellen Eingangswerten aus.

Ereignis-Ausgänge

- **INITO**: Bestätigt den Abschluss der Initialisierung oder Deinitialisierung.
- **CNF**: Bestätigt die Ausführung der Berechnung und signalisiert, dass ein aktualisierter Ausgangswert vorliegt.

Daten-Eingänge

- **QI** (BOOL): Eingangs-Event-Qualifier. Steuert die Gültigkeit der Eingangsdaten und die Aktivierung des Bausteins.
- **INPUT** (REAL): Das zu überwachende, analoge Eingangssignal.
- **THRESHOLD** (REAL, Standardwert: `0.0`): Der Mittelpunkt des Hysteresebands (Sollwert).
- **HYSTERESIS** (REAL, Standardwert: `0.1`): Die Gesamtbreite des Hysteresebands.

Daten-Ausgänge

- **QO** (BOOL): Ausgangs-Event-Qualifier (gibt an, ob der Baustein aktiv und betriebsbereit ist).
- **OUTPUT** (BOOL): Das digitale Ausgangssignal des Hystereseglieds.

Adapter

Dieser Funktionsbaustein verwendet keine Adapter.

• – ◦ • – ◦ • – ◦ •

Funktionsweise

Der Funktionsbaustein arbeitet mit einem symmetrischen Hystereseband um den definierten Mittelpunkt (`THRESHOLD`). Um auch bei negativen Hysteresewerten ein konsistentes Verhalten zu gewährleisten, wird intern der Absolutwert (`ABS`) der Hysterese verwendet.

Die Schwellenwerte für das Schalten berechnen sich wie folgt:

- **Einschaltpunkt (Switch-on):** $THRESHOLD + \frac{|HYSTERESIS|}{2.0}$
- **Ausschaltpunkt (Switch-off):** $THRESHOLD - \frac{|HYSTERESIS|}{2.0}$

Schaltlogik

1. Befindet sich der Baustein im Zustand **sOFF** (Ausgang ist `FALSE`), schaltet der Ausgang auf `TRUE` (**sON**), sobald das Eingangssignal den oberen Grenzwert erreicht oder überschreitet:

$$\text{INPUT} \geq \text{THRESHOLD} + \frac{|\text{HYSTERESIS}|}{2.0} \tag{2.139}$$

2. Befindet sich der Baustein im Zustand **sON** (Ausgang ist `TRUE`), schaltet der Ausgang erst dann wieder auf `FALSE` (**sOFF**), wenn das Eingangssignal den unteren Grenzwert streng unterschreitet:

$$\text{INPUT} < \text{THRESHOLD} - \frac{|\text{HYSTERESIS}|}{2.0} \tag{2.140}$$

• – ◦ • – ◦ • – ◦ •

Technische Besonderheiten

- **Schwingungsschutz:** Die Ausschaltbedingung verwendet eine strikte Ungleichung ($<$), während die Einschaltbedingung inklusiv ist ($\geq$). Dies verhindert Oszillationen exakt an den Grenzen des Hysteresebands.
- **Sicherheitsverriegelung über QI:** Der Ausgang `OUTPUT` kann nur dann den Zustand `TRUE` annehmen, wenn der Eingangs-Qualifier `QI` auf `TRUE` gesetzt ist. Ist `QI` im Zustand `FALSE`, bleibt der Ausgang gesperrt (`FALSE`).
- **Robustheit:** Durch die Verwendung von `ABS(HYSTERESIS)` im Algorithmus führt die Eingabe einer negativen Hysterese nicht zu einem Fehlverhalten des Bausteins.

• – ◦ • – ◦ • – ◦ •

Zustandsübersicht

Der Funktionsbaustein ist als Basic-FB mit einer Execution Control Chart (ECC) realisiert:

1. **START**: Der Standard-Ruhezustand nach dem Systemstart.
2. **Init**: Wird durch das Ereignis `INIT` bei gesetztem `QI = TRUE` aufgerufen. Initialisiert den Baustein und setzt den Ausgang `OUTPUT` auf `FALSE`.
3. **sOFF**: Der Zustand, in dem die Ausschaltbedingung erfüllt ist (`OUTPUT = FALSE`).
4. **sON**: Der Zustand, in dem die Einschaltbedingung erfüllt ist (`OUTPUT = TRUE` sofern `QI = TRUE`).
5. **DeInit**: Wird durch `INIT` bei `QI = FALSE` aufgerufen. Setzt `QO` und `OUTPUT` zurück auf `FALSE`.

• – ◦ • – ◦ • – ◦ •

Anwendungsszenarien

- **Zweipunktregler:** Temperaturregelung (z. B. Heizung ein bei $< 19^\circ$ C, aus bei $> 21^\circ$ C mit einem Threshold von 20° C und einer Hysterese von 2° C).
- **Füllstandsüberwachung:** Aktivierung einer Pumpe beim Erreichen eines maximalen Füllstands und Deaktivierung beim Unterschreiten des Minimalwerts.
- **Rauschunterdrückung:** Digitalisierung von verrauschten analogen Sensorsignalen (z. B. Druck- oder Helligkeitssensoren), um unruhige Signalwechsel am Ausgang zu eliminieren.

• – ◦ • – ◦ • – ◦ •

Vergleich mit ähnlichen Bausteinen

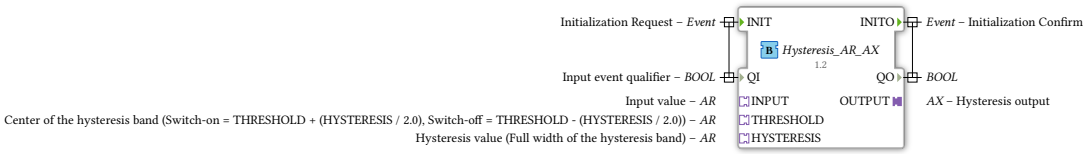
Im Vergleich zu einem einfachen Komparator (z. B. `GT` oder `GE`-Bausteine), die sofort bei jedem minimalen Über- oder Unterschreiten eines Wertes schalten, bietet der `Hysteresis`-Baustein durch die zwei Schaltschwellen eine inhärente Signalstabilisierung. Er vereint die Logik von zwei Komparatoren und einem RS-Flip-Flop in einem einzigen, übersichtlichen Baustein.

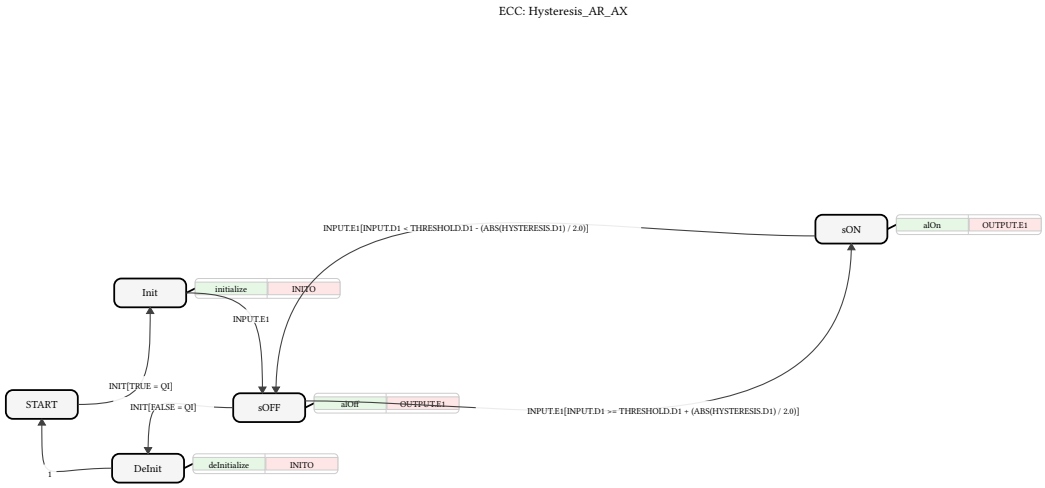
• – ◦ • – ◦ • – ◦ •

Fazit

Der `Hysteresis`-Baustein ist ein essenzielles Werkzeug in der Signalverarbeitung und Regelungstechnik innerhalb der IEC 61499. Er bietet eine robuste, einfach zu konfigurierende Lösung zur Erzeugung stabiler binärer Steuersignale aus kontinuierlichen analogen Eingangswerten.

Hysteresis_AR_AX





• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock `Hysteresis_AR_AX` ist ein Basic-Funktionsblock (BFB) für die ereignisgesteuerte Signalverarbeitung in der IEC 61499. Er dient der Analog-Digital-Wandlung eines analogen Eingangssignals unter Berücksichtigung einer einstellbaren Hysterese. Dadurch wird verhindert, dass kleine Signalschwankungen oder Rauschen um einen Schwellenwert herum zu einem schnellen, unerwünschten Hin- und Herschalten des digitalen Ausgangs (Oszillation) führen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung (`EInit`). Aktiviert oder deaktiviert den Baustein in Kopplung mit dem Eingang `QI`.

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung (`EInit`). Signalisiert den Abschluss der Initialisierung oder Deinitialisierung in Kopplung mit `QO`.

Daten-Eingänge

- **QI** (BOOL): Input Event Qualifier. Bestimmt, ob der Baustein aktiv sein soll (`TRUE`) oder deinitialisiert werden soll (`FALSE`).

Daten-Ausgänge

- **QO** (BOOL): Output Event Qualifier. Zeigt den aktuellen Status der Funktionsblock-Aktivierung an.

Adapter

Der Baustein nutzt standardisierte, unidirektionale Adapter für eine saubere, modulare Kopplung von Daten und Ereignissen.

- **Sockets (Eingangs-Adapter):**
  - **INPUT** (`adapter::types::unidirectional::AR`): Der zu überwachende analoge Eingangswert. Der Wert liegt an `INPUT.D1` an; die Auswertung wird durch das Ereignis `INPUT.E1` getriggert.
  - **THRESHOLD** (`adapter::types::unidirectional::AR`): Der Mittelpunkt des Hysteresebandes (`THRESHOLD.D1`).

  - **HYSTERESIS** (`adapter::types::unidirectional::AR`): Die Gesamtbreite des Hysteresebandes (`HYSTERESIS.D1`).
- **Plugs (Ausgangs-Adapter):**
  - **OUTPUT** (`adapter::types::unidirectional::AX`): Das digitale Ausgangssignal. Gibt das Schaltergebnis am Datenpunkt `OUTPUT.D1` (BOOL) aus und triggert den Ausgang über das Ereignis `OUTPUT.E1`.

Funktionsweise

Die Hysterese ist symmetrisch um den Schwellenwert (`THRESHOLD`) definiert. Die Gesamtbreite des Hysteresebandes wird durch den Parameter `HYSTERESIS` vorgegeben.

Berechnungsformeln

Um Fehlkonfigurationen zu vermeiden, wird intern stets der Absolutwert der Hysteresebreite (`ABS(HYSTERESIS.D1)`) verwendet.

1. **Einschaltbedingung (Switch-on):** Der Ausgang schaltet ein (`OUTPUT.D1 := TRUE`), wenn der Eingangswert die obere Grenze des Hysteresebandes erreicht oder überschreitet:
$$\text{INPUT.D1} \geq \text{THRESHOLD.D1} + \frac{|\text{HYSTERESIS.D1}|}{2.0} \tag{2.141}$$
2. **Ausschaltbedingung (Switch-off):** Der Ausgang schaltet aus (`OUTPUT.D1 := FALSE`), wenn der Eingangswert die untere Grenze des Hysteresebandes streng unterschreitet:
$$\text{INPUT.D1} < \text{THRESHOLD.D1} - \frac{|\text{HYSTERESIS.D1}|}{2.0} \tag{2.142}$$

Technische Besonderheiten

- **Verhinderung von Oszillationen:** Durch die Verwendung einer strikten Ungleichung ($<$) beim Ausschalten und einer weichen Ungleichung ($\geq$) beim Einschalten wird ein stabiles Verhalten an den exakten Grenzen des Hysteresebandes garantiert.
- **Absoluthysterese:** Durch die Verwendung der `ABS`-Funktion im Execution Control Chart (ECC) arbeitet der Baustein auch dann korrekt, wenn für die Hysteresebreite fälschlicherweise ein negativer Wert übergeben wird.
- **Adapterbasierte Architektur:** Die Verwendung von unidirektionalen Adaptern (`AR` / `AX`) bündelt Daten- und Ereignisleitungen. Dies erhöht die Übersichtlichkeit im Applikationsdiagramm signifikant.

Zustandsübersicht

Das Verhalten wird über die folgenden Zustände im Execution Control Chart (ECC) gesteuert:

1. **START**: Ruhezustand vor der Initialisierung.
2. **Init**: Setzt den Ausgang `OUTPUT.D1` zurück und setzt `QO := QI`.
3. **sOFF**: Zustand des ausgeschalteten Ausgangs. Der Algorithmus `alOff` stellt sicher, dass `OUTPUT.D1` auf `FALSE` gesetzt ist. Ein Übergang nach `sON` erfolgt, sobald die Einschaltbedingung erfüllt ist.
4. **sON**: Zustand des eingeschalteten Ausgangs. Der Algorithmus `alOn` setzt `OUTPUT.D1` auf `TRUE` (sofern `QI` aktiv ist). Ein Übergang zurück nach `sOFF` erfolgt, sobald die Ausschaltbedingung erfüllt ist.
5. **DeInit**: Deinitialisierungszustand. Setzt die Ausgänge kontrolliert zurück.

Anwendungsszenarien

- **Zwei-Punkt-Temperaturregelung:** Steuerung einer Heizung. Die Heizung schaltet ein, wenn die Temperatur unter den Sollwert minus der halben Hysterese fällt, und schaltet erst wieder aus, wenn die Temperatur den Sollwert plus der halben Hysterese überschreitet.
- **Füllstandskontrolle:** Ein- und Ausschalten von Pumpen in Tanks. Verhindert ein permanentes Schalten der Pumpen bei unruhigen Flüssigkeitsoberflächen (Wellenbewegungen).
- **Grenzüberwachung im Prozessbereich:** Umwandlung verrauschter analoger Sensorwerte (z.B. Druck, Durchfluss) in stabile digitale Alarmsignale.

Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu klassischen Komparator-Bausteinen (wie `GT` oder `LT`), die sofort bei jeder minimalen Schwellenwertüberschreitung reagieren, puffert `Hysteresis_AR_AX` das Signal durch das definierte Band ab.

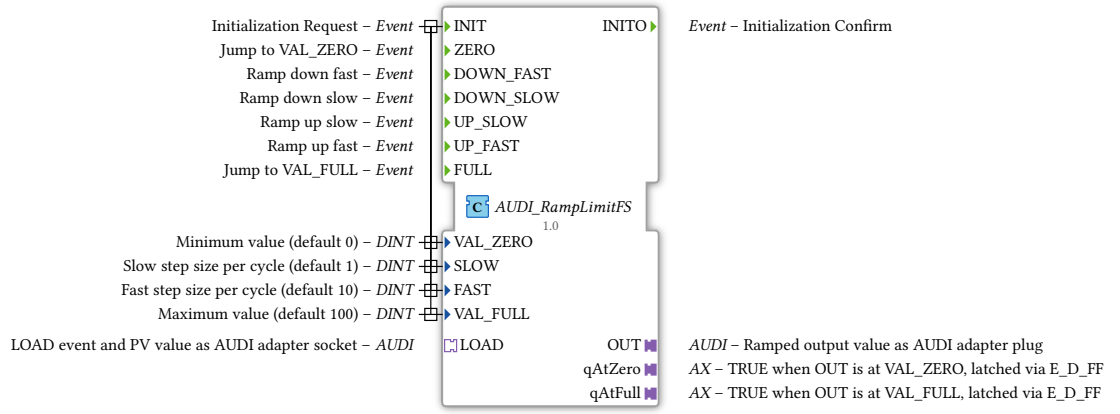
Gegenüber Standard-Hysteresegliedern, die oft mit separaten Pins für Daten und Trigger-Events arbeiten, vereinfacht dieser Baustein durch das standardisierte Adapter-Konzept (AR/AX) das Signalrouting in komplexeren 4diac-Anwendungen erheblich.

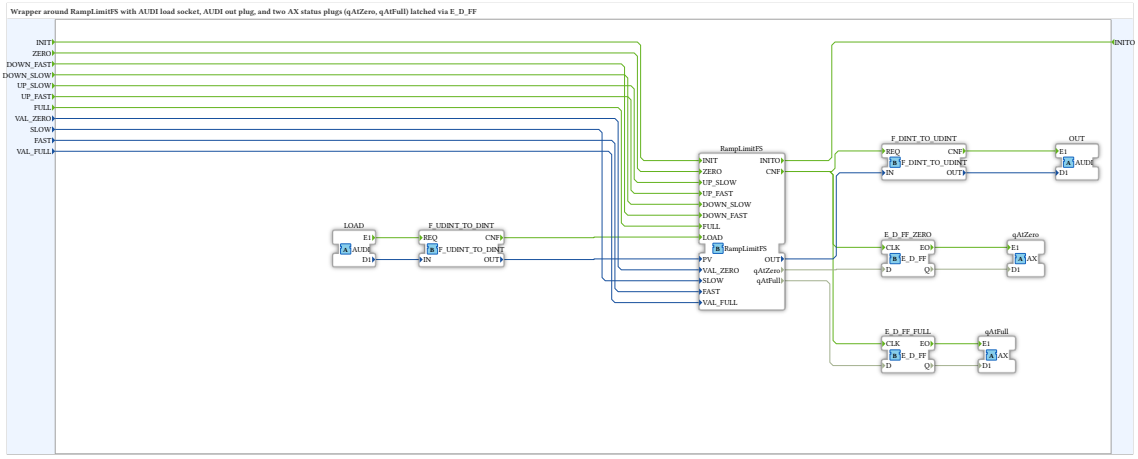
Fazit

Der `Hysteresis_AR_AX` ist ein hocheffizienter und robuster Baustein für die Signalvorverarbeitung. Dank der integrierten Absolutwertbildung der Hysterese und den asymmetrischen Schaltbedingungen bietet er eine hohe Ausfallsicherheit im praktischen Betrieb. Durch das konsequente Nutzen von Adaptern fügt er sich ideal in moderne, serviceorientierte Steuerungsarchitekturen ein.

Ramp

AUDI_RampLimitFS





• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock **AUDI_RampLimitFS** ist ein gekapselter Rampenbegrenzer, der auf dem Basisbaustein `RampLimitFS` aufbaut und eine anwendungsorientierte Schnittstelle über AUDI- und AX-Adapter bereitstellt. Er ermöglicht das schrittweise Anfahren an einen Grenzwert („Hochrampe“) und das schrittweise Verlassen („Rampen abwärts“), jeweils mit unterschiedlichen Rampenraten. Integrierte Statuslatch-Ausgänge melden, sobald der Ausgangswert die untere oder obere Grenze erreicht hat. Durch die Nutzung standardisierter Adapter vereinfacht der Baustein die Einbindung in bestehende Signalverarbeitungsketten.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Kommentar

INIT	Initialisierungsanforderung; übernimmt die Parameter und startet den Baustein. (Mit VAL_ZERO, SLOW, FAST, VAL_FULL)
ZERO	Sprung auf den Minimalwert (VAL_ZERO).
DOWN_FAST	Rampenabwärts mit schneller Rate (FAST).
DOWN_SLOW	Rampenabwärts mit langsamer Rate (SLOW).
UP_SLOW	Rampenaufwärts mit langsamer Rate (SLOW).
UP_FAST	Rampenaufwärts mit schneller Rate (FAST).
FULL	Sprung auf den Maximalwert (VAL_FULL).

Ereignis-Ausgänge

Ereignis	Kommentar
INIT0	Bestätigung der erfolgreichen Initialisierung.

Daten-Eingänge

Variable	Typ	Initialwert	Kommentar
VAL_ZERO	DINT	0	Minimalwert der Rampe.
SLOW	DINT	1	Schrittweite für langsame Rampenbewegung pro Zyklus.
FAST	DINT	10	Schrittweite für schnelle Rampenbewegung pro Zyklus.
VAL_FULL	DINT	100	Maximalwert der Rampe.

Daten-Ausgänge

Direkte Datenausgänge sind nicht vorhanden. Alle Ausgangswerte werden über die **Adapter-Plugs** OUT, qAtZero und qAtFull bereitgestellt.

Adapter

Adapter	Richtung	Typ	Kommentar
LOAD	Socket	adapter::types::unidirectional::AUDI	Empfängt das Zielwert-Signal (PV) und das zugehörige Ereignis (z.B. E1).
OUT	Plug	adapter::types::unidirectional::AUDI	Gibt den gerammpten Ausgangswert aus.
qAtZero	Plug	adapter::types::unidirectional::AX	Meldet (gelatcht), dass der Ausgang den Minimalwert erreicht hat.
qAtFull	Plug	adapter::types::unidirectional::AX	Meldet (gelatcht), dass der Ausgang den Maximalwert erreicht hat.

Funktionsweise

Der Baustein verarbeitet einen Zielwert, der über den LOAD-Adapter eintrifft, und erzeugt daraus einen rampenförmigen Ausgangswert, der über den OUT-Adapter ausgegeben wird. Der interne Kern RampLimitFS führt die eigentliche Rampenberechnung durch.

Über die Ereigniseingänge ZERO, FULL, DOWN_FAST, DOWN_SLOW, UP_SLOW und UP_FAST wird die Richtung und Geschwindigkeit der Rampe gesteuert. Die Werte VAL_ZERO und VAL_FULL definieren die Unter- bzw. Obergrenze. Die Schrittweiten SLOW und FAST bestimmen, wie stark sich der Ausgangswert pro Ereigniszyklus ändert.

Die Konvertierung zwischen UDINT (im AUDI-Adapter) und DINT (intern verwendeter Datentyp) wird durch die eingebetteten Funktionsbausteine F_UDINT_TO_DINT bzw. F_DINT_TO_UDINT automatisch durchgeführt.

Die beiden Statusausgänge `qAtZero` und `qAtFull` werden durch zwei `E_D_FF`-Latchbausteine realisiert. Sobald der interne Rampenbaustein meldet, dass der Ausgang den Minimalwert (bzw. Maximalwert) erreicht hat, wird das entsprechende Latch gesetzt und anschließend über den AX-Adapter signalisiert. Die Latches bleiben solange gesetzt, bis sie durch eine neue Bedingung zurückgesetzt werden.

Technische Besonderheiten

- **Adapterintegration**: Nutzung des unidirektionalen AUDI-Adapters für Werte und Ereignisse sowie des AX-Adapters für Statusinhalte. Dadurch ist eine modulare Einbindung in bestehende Signalpfade ohne proprietäre Schnittstellen möglich.
- **Datentypkonvertierung**: Automatische Umwandlung von `UDINT` (aus dem AUDI-Socket) in `DINT` für die interne Verarbeitung und anschließende Rückumwandlung für den AUDI-Plug.
- **Status-Latching**: Die Erreichung von Unter- und Obergrenze wird in Flip-Flops (`E_D_FF`) festgehalten, sodass auch kurze Zustandswechsel zuverlässig erfasst werden.
- **Mehrere Rampenraten**: Unterstützung von zwei unterschiedlichen Geschwindigkeiten (`SLOW` und `FAST`) ermöglicht feiner abgestufte positive und negative Bewegung.
- **Direkter Sprung** auf die Grenzwerte über die Ereignisse `ZERO` und `FULL`, unabhängig von Rampenzeiten.

Zustandsübersicht

Der Baustein besitzt keine explizit sichtbaren Zustände, jedoch lässt sich der interne Ablauf wie folgt beschreiben:

- **Initialisierung**: Nach `INIT` werden die Grenzen und Schrittweiten übernommen; der Ausgang wird auf `VAL_ZERO` gesetzt (Standardverhalten laut Beschreibung).
- **Betrieb**: Abhängig von den anliegenden Ereignissen wird der Ausgangswert schrittweise erhöht (`UP_*`) oder verringert (`DOWN_*`), wobei die Schrittweite der jeweiligen Geschwindigkeit entspricht.
- **Grenzerreichen**: Wird die untere oder obere Grenze erreicht, aktiviert der Baustein die entsprechenden Statuslatch (`qAtZero`, `qAtFull`).
- **Sprünge**: Die Ereignisse `ZERO` und `FULL` setzen den Ausgang sofort auf den jeweiligen Grenzwert (ohne Rampenbewegung).

Anwendungsszenarien

- **Antriebssteuerung**: Sanftes Hoch- und Herunterfahren von Förderbändern, Lüftern oder Pumpen, um mechanischen Verschleiß zu minimieren.
- **Positionieraufgaben**: Rampenbegrenzung für Servoachsen oder Schrittmotoren, bei denen ein gleichmäßiges Anfahren an eine Zielposition erforderlich ist.
- **Signalglättung**: Glättung von Prozesswerten in der Messtechnik, um Störspitzen abzufangen und einen stabilen Regelkreis zu gewährleisten.
- **Test- und Prüfstände**: Erzeugung definierter Anfahr- und Abschaltkurven für Prüfzyklen.

Vergleich mit ähnlichen Bausteinen

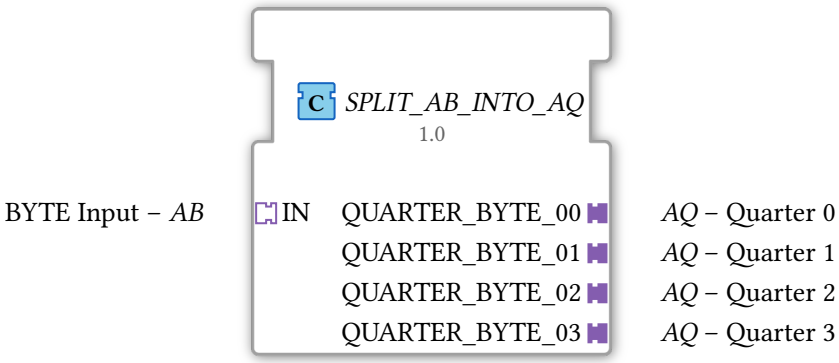
Gegenüber dem ursprünglichen `RampLimitFS` bietet **AUDI_RampLimitFS** eine vereinheitlichte Schnittstelle für den direkten Anschluss an AUDI-basierte Signalpfade und verhindert so manuelle Konvertierungs- und Verdrahtungsaufwände. Zusätzlich werden die Grenzwertstatusinformationen gelatcht, sodass die Signalauswertung unabhängig von der Dauer des Grenzzustands erfolgen kann. Im Vergleich zu einem einfachen PID- oder Integrator-Baustein ist hier nur eine begrenzte Anzahl von Ereignissen und Einstellparametern vorhanden, was den Baustein insbesondere für einfache, robuste Rampenfunktionen in Echtzeitanwendungen prädestiniert.

Fazit

AUDI_RampLimitFS ist ein modular aufgebauter Rampenbaustein, der die Kernfunktionalität eines klassischen Rampengenerators mit modernen Adapterkonzepten und Statuslatch-Funktionen kombiniert. Die klare Schnittstelle, die automatische Datentypkonvertierung und die zwei Rampengeschwindigkeiten machen ihn zu einer vielseitig einsetzbaren Komponente in Automatisierungssystemen. Durch die Kapselung komplexer internen Logik wird der Anwender von typischen Implementierungsdetails befreit und kann den Baustein direkt in seine Signalverarbeitung integrieren.

2.1.11.16 splitting

SPLIT_AB_INTO_AQ



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `SPLIT_AB_INTO_AQ` dient dazu, ein über einen `AB`-Adapter (Byte) eingehendes Datenbyte in vier einzelne 2-Bit-Quartette aufzuteilen und diese jeweils über `AQ`-Adapter (Quarter) auszugeben. Er kapselt die Zerlegung eines Bytes in handhabbare Teilwerte und stellt sie über standardisierte unidirektionale Adapter-Schnittstellen für die weitere Verarbeitung zur Verfügung.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Beschreibung
`IN.E1`	Ereignis am Socket löst die Zerlegung und Ausgabe der Quartette aus.

Ereignis-Ausgänge

Keine direkten Ereignisausgänge; die Ausgabe erfolgt über die Ereignisse der angeschlossenen Adapter (`QUARTER_BYTE_xx.E1`).

Daten-Eingänge

Name	Beschreibung
`IN.D1`	Das eingehende Datenbyte (Typ: BYTE), das in vier Quartette aufgeteilt wird.

Daten-Ausgänge

Keine direkten Datenausgänge; die Quartette werden über die Daten-Schnittstellen der Ausgangs-Adapter (`QUARTER_BYTE_xx.D1`) bereitgestellt.

Adapter

Name	Typ	Richtung	Beschreibung
`IN`	`adapter::types::unidirectional::AB`	Socket (Eingang)	Empfängt das zu zerlegende Byte inkl. Ereignis.
`QUARTER_BYTE_00`	`adapter::types::unidirectional::AQ`	Plug (Ausgang)	Gibt das Quartett 0 (niederwertigste 2 Bits) aus.
`QUARTER_BYTE_01`	`adapter::types::unidirectional::AQ`	Plug (Ausgang)	Gibt das Quartett 1 aus.
`QUARTER_BYTE_02`	`adapter::types::unidirectional::AQ`	Plug (Ausgang)	Gibt das Quartett 2 aus.
`QUARTER_BYTE_03`	`adapter::types::unidirectional::AQ`	Plug (Ausgang)	Gibt das Quartett 3 (höchstwertigste 2 Bits) aus.

Funktionsweise

1. Ein Ereignis am Socket `IN.E1` aktiviert die Verarbeitung.
2. Das am Socket anliegende Byte (`IN.D1`) wird an den internen Baustein `SPLIT_BYTE_INTO_QUARTERS` weitergeleitet. Dieser zerlegt das Byte in vier 2-Bit-Teile (Quartette).
3. Die vier Quartette werden parallel an die Daten-Eingänge von `E_D_FF_ANY`-Flip-Flops gegeben.
4. Gleichzeitig wird das Bestätigungsereignis (`CNF`) des Splitters an die Takt-Eingänge (`CLK`) aller vier Flip-Flops verteilt.
5. Mit der positiven Flanke dieses Ereignisses übernehmen die Flip-Flops die Quartette und geben sie an den jeweiligen Ausgang (`Q`) weiter.
6. Die Ausgänge der Flip-Flops sind direkt mit den Daten-Eingängen der Ausgangs-Adapter `QUARTER_BYTE_00` … `QUARTER_BYTE_03` verbunden. Gleichzeitig wird das Flip-Flop-Ausgangsereignis (`EO`) an den Ereigniseingang (`E1`) des jeweiligen Adapters gesendet, sodass die Daten auf der Adapter-Schnittstelle ausgegeben werden.

Somit wird mit jeder Aktivierung des Sockets ein gesamtes Byte in vier Quartette aufgeteilt und synchron an alle vier Ausgangs-Adapter übermittelt.

Technische Besonderheiten

- **Interne Flip-Flops:** Die Verwendung von `E_D_FF_ANY` stellt sicher, dass die Zerlegung nur einmal pro Ereignis erfolgt und die Werte zwischengespeichert werden, bis ein neues Ereignis eintrifft.
- **Adapter-Kopplung:** Statt direkter Ein-/Ausgänge werden ausschließlich unidirektionale Adapter verwendet, was eine lose Kopplung und Wiederverwendbarkeit in komplexen Adapter-Netzwerken ermöglicht.
- **Synchronisation:** Die Taktung der Flip-Flops erfolgt über das gemeinsame `CNF`-Ereignis, sodass alle Quartette zeitgleich anliegen.
- **Keine Zustandsmaschine:** Der FB arbeitet rein ereignisgesteuert ohne eigenen sequenziellen Zustand – die Logik ist vollständig durch die vernetzten internen Bausteine realisiert.

Zustandsübersicht

Da `SPLIT_AB_INTO_AQ` ein zusammengesetzter FB ohne eigene Zustandsmaschine ist, wird sein Verhalten durch die internen Bausteine bestimmt. Im Wesentlichen existieren zwei Phasen:

1. **Idle:** Es liegt kein Ereignis am Socket an; die Ausgangs-Adapter halten die zuletzt übernommenen Werte.
2. **Verarbeitung:** Ein Ereignis am Socket löst das Zerlegen und Aktualisieren aller Ausgänge aus. Nach Abschluss geht der FB wieder in den Ruhezustand über.

Anwendungsszenarien

- **Steuerung von vier Aktoren:** Ein Byte (z. B. von einem Master-Steuerbus) wird in vier 2-Bit-Werte zerlegt, um separate Stellgrößen für unabhängige Aktuatoren zu liefern.
- **Datenreduktion:** Aufteilung eines Bytes in vier einzelne Signale zur Übergabe an parallele, einfachere Verarbeitungseinheiten.
- **Adapter-basierte Architekturen:** Einsatz in Systemen, die durchgängig mit unidirektionalen Adaptern arbeiten, um die Modularität zu erhöhen.

Vergleich mit ähnlichen Bausteinen

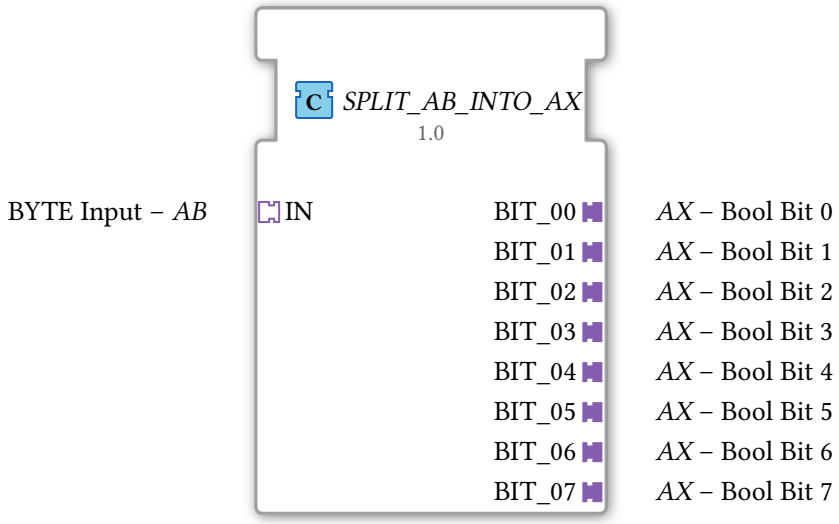
- **`SPLIT_BYTE_INTO_QUARTERS`:** Leistet die reine Zerlegung eines Bytes in vier Quartette, jedoch über direkte Daten- und Ereignis-Ein-/Ausgänge. `SPLIT_AB_INTO_AQ` kapselt diese Logik in einer Adapter-Schnittstelle und fügt Flip-Flops zur stabilen Ausgabe hinzu.
- **`SPLIT_WORD_INTO_BYTES`:** Zerlegt ein 16-Bit-Wort in zwei Bytes – analoges Prinzip, aber für andere Datenbreiten und andere Adaptertypen.
- **Einfache Multiplexer/Demultiplexer:** Während diese Signale umschalten, konzentriert sich `SPLIT_AB_INTO_AQ` auf die feste, parallele Aufteilung eines zusammengesetzten Datentyps.

Fazit

`SPLIT_AB_INTO_AQ` ist ein spezialisierter zusammengesetzter Funktionsblock zur Aufteilung eines Byte-Adapter-Eingangs in vier separate Quarter-Adapter-Ausgänge. Er vereinfacht den Entwurf von Adapter-Netzwerken, indem er die komplexe Bit-Manipulation verbirgt und eine zuverlässige, ereignisgesteuerte Ausgabe der Quartette gewährleistet. Durch die Verwendung von Flip-Flops wird eine stabile

Datenhaltung zwischen den Ereignissen erreicht. Der Baustein eignet sich besonders für modulare Automatisierungslösungen, die mit unidirektionalen Adaptern arbeiten.

SPLIT_AB_INTO_AX



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **SPLIT_AB_INTO_AX** dient dazu, ein Byte, das über einen unidirektionalen AB-Adapter empfangen wird, in seine acht einzelnen Bits aufzuspalten und diese über jeweils separate AX-Adapter bereitzustellen. Dabei erfolgt eine taktgesteuerte Übernahme der Bits mittels D-Flipflops, sodass die Ausgangswerte stabil bleiben, bis ein neues Byte verarbeitet wird.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine expliziten Ereignis-Eingänge auf der obersten Ebene. Die Verarbeitung wird durch das Ereignis ausgelöst, das über den Socket-Adapter **IN** (Typ AB) eintrifft. Der Adapter **IN** stellt einen Ereignis-Eingang (`E1`) und einen Daten-Eingang (`D1`) bereit.

Ereignis-Ausgänge

Der Baustein besitzt keine expliziten Ereignis-Ausgänge auf der obersten Ebene. Die Ergebnisausgabe erfolgt über die Plug-Adapter **BIT_00** bis **BIT_07** (Typ AX), die ebenfalls einen Ereignis-Ausgang (`E1`) enthalten.

Daten-Eingänge

- **IN.D1** – über den Socket-Adapter **IN** (Typ AB) wird das zu verarbeitende Byte (BYTE) empfangen.

Daten-Ausgänge

- **BIT_00.D1 … BIT_07.D1** – über die Plug-Adapter **BIT_00** bis **BIT_07** (Typ AX) werden die einzelnen booleschen Werte der Bits (Bit 0 bis Bit 7) ausgegeben.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::AB	Socket (Eingang)	BYTE-Wert, wird in Einzelbits zerlegt
BIT_00	adapter::types::unidirectional::AX	Plug (Ausgang)	Boolescher Wert von Bit 0 (LSB)

BIT_01	adapter::types::unidirectional::AX	Plug (Ausgang)	Boolescher Wert von Bit 1
BIT_02	adapter::types::unidirectional::AX	Plug (Ausgang)	Boolescher Wert von Bit 2
BIT_03	adapter::types::unidirectional::AX	Plug (Ausgang)	Boolescher Wert von Bit 3
BIT_04	adapter::types::unidirectional::AX	Plug (Ausgang)	Boolescher Wert von Bit 4
BIT_05	adapter::types::unidirectional::AX	Plug (Ausgang)	Boolescher Wert von Bit 5
BIT_06	adapter::types::unidirectional::AX	Plug (Ausgang)	Boolescher Wert von Bit 6
BIT_07	adapter::types::unidirectional::AX	Plug (Ausgang)	Boolescher Wert von Bit 7 (MSB)

Funktionsweise

1. Ein eingehendes Byte wird über den Socket-Adapter **IN** empfangen. Das Ereignis `E1` löst die Verarbeitung aus.
2. Der interne Funktionsblock `SPLIT_BYTE_INTO_BOOLS` zerlegt das Byte in acht boolesche Signale (`BIT_00` … `BIT_07`). Gleichzeitig wird das Ereignis `CNF` erzeugt.
3. Dieses Ereignis wird auf die Takteingänge (`CLK`) von acht **E_D_FF**-Flipflops verteilt. Jedes Flipflop übernimmt beim Takt den zugehörigen booleschen Wert an seinem Dateneingang `D`.
4. Die Ausgänge `Q` der Flipflops liefern die stabilisierten Bits. Sie werden über die Datenverbindungen an die Plug-Adapter **BIT_00** … **BIT_07** weitergegeben.
5. Das Ereignis `EO` jedes Flipflops wird auf den Ereignis-Ausgang des jeweiligen Plug-Adapters gelegt, sodass nachgeschaltete Bausteine über die neue Bit-Information benachrichtigt werden.

Technische Besonderheiten

- **Zustandsspeicherung**: Die Verwendung von D-Flipflops (E_D_FF) sorgt dafür, dass die Ausgangswerte auch nach dem Verarbeitungsereignis erhalten bleiben, bis ein neues Byte verarbeitet wird.
- **Parallele Ausgabe**: Alle acht Bits werden gleichzeitig aktualisiert, sobald das gemeinsam getaktete Ereignis `CNF` von `SPLIT_BYTE_INTO_BOOLS` anliegt.
- **Unidirektionale Adapter**: Sowohl der Eingangs- als auch die Ausgangsadapter sind unidirektional (von der Applikation zur Ressource), daher eignet sich der Baustein für Senderichtungen, in denen nur eine Seite die Datenflüsse steuert.

Zustandsübersicht

Der Baustein selbst besitzt keinen expliziten Zustandsautomaten. Das Verhalten ergibt sich aus der internen Kombination von:

- **SPLIT_BYTE_INTO_BOOLS**: Einzelschritt – erzeugt aus einem BYTE acht BOOL-Werte und ein Bestätigungsereignis.
- **E_D_FF**: Jeder Flipflop speichert einen booleschen Wert bei steigender Taktflanke und hält ihn bis zum nächsten Takt.

Somit lässt sich der Baustein als **reiner kombinatorischer Splitter mit nachgeschalteten Haltegliedern** beschreiben.

Anwendungsszenarien

- **Digitale Ausgabeansteuerung**: Ein Byte (z. B. von einem Kommunikationsadapter) wird in einzelne digitale Ausgänge aufgeteilt, die jeweils über AX-Getter-Adapter mit Aktoren verbunden werden.
- **Zustandsüberwachung**: Ein Byte-Signal (z. B. von einem Feldbus) wird in einzelne Statusbits zerlegt, die getrennt voneinander weiterverarbeitet oder visualisiert werden können.
- **Bitleveldatenvorbereitung**: Vor der Übergabe an bausteininterne Verarbeitungen, die einzelne boolesche Signale erwarten.

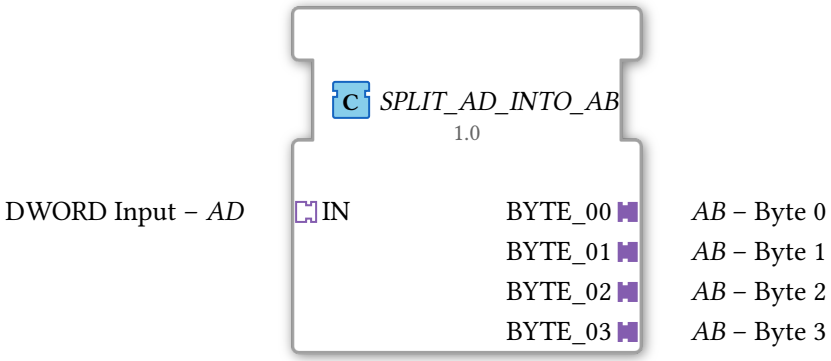
Vergleich mit ähnlichen Bausteinen

- **SPLIT_BYTE_INTO_BOOLS**: Zerlegt ein Byte ebenfalls in boolesche Werte, gibt diese aber direkt als Datenausgänge weiter und erzeugt ein einziges Ereignis.
  SPLIT_AB_INTO_AX erweitert dies um eine taktgesteuerte Zwischenspeicherung und die Ausgabe über AX-Adapter, was eine saubere Ereignis-Synchronisation mit nachfolgenden Bausteinen erlaubt.
- **Einfache Data-Array-Splitter**: Manche Bibliotheken bieten Bausteine, die Arrays in Einzelelemente aufteilen, jedoch ohne die zusätzliche Speicherung und Adapter-Schnittstelle.

Fazit

SPLIT_AB_INTO_AX ist ein spezialisierter Baustein für die Aufteilung eines BYTE-Werts in acht stabil gehaltene boolesche Ausgänge über AX-Adapter. Durch die Kombination von Splitter und Flipflops eignet er sich besonders für Szenarien, in denen die einzelnen Bits asynchron oder zeitlich versetzt abgegriffen werden müssen, ohne dass sich der Wert während der Verarbeitung ändert. Seine klare, adapterbasierte Schnittstelle erleichtert die Einbindung in modulare IEC 61499-Anwendungen.

SPLIT_AD_INTO_AB



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **SPLIT_AD_INTO_AB** dient der Aufteilung eines 32-Bit-Datenworts (AD-Adapter) in vier einzelne Bytes. Die vier Bytes werden über jeweils separate unidirektionale AB-Adapter ausgegeben. Der Baustein ist als Composite-FB realisiert und nutzt dazu einen SPLIT_DWORD_INTO_BYTES sowie mehrere flankengesteuerte D-Flipflops (E_D_FF_ANY).

Schnittstellenstruktur

Der FB besitzt keine klassischen Ereignis- oder Datenschnittstellen. Die gesamte Kommunikation erfolgt über Adapter:

Adapter (Sockets)

Adapter	Typ	Kommentar
IN	`adapter::types::unidirectional::AD`	DWORD-Eingang (32-Bit)

Adapter (Plugs)

Adapter	Typ	Kommentar
BYTE_00	`adapter::types::unidirectional::AB`	Byte 0 (niederwertigstes Byte)
BYTE_01	`adapter::types::unidirectional::AB`	Byte 1
BYTE_02	`adapter::types::unidirectional::AB`	Byte 2
BYTE_03	`adapter::types::unidirectional::AB`	Byte 3 (höchstwertiges Byte)

Jeder AB-Adapter besitzt mindestens einen Ereignisausgang (E1) und einen Datenausgang (D1), die mit den internen Flipflops verbunden sind.

Funktionsweise

1. Ein Ereignis am **IN**-Adapter (E1) triggert den internen Baustein `SPLIT_DWORD_INTO_BYTES`.
2. `SPLIT_DWORD_INTO_BYTES` zerlegt den am **IN.D1** anliegenden DWORD in vier einzelne Bytes (BYTE_00 bis BYTE_03) und gibt sie als Daten aus.
3. Gleichzeitig erzeugt `SPLIT_DWORD_INTO_BYTES` ein Bestätigungsereignis (CNF), das an die Takteingänge (CLK) aller vier `E_D_FF_ANY`-Flipflops weitergeleitet wird.
4. Jedes Flipflop speichert das ihm zugewiesene Byte (D-Eingang) bei der steigenden Flanke des Taktes und gibt es an seinem Ausgang (Q) aus.
5. Die Ausgänge der Flipflops sind mit den Datenausgängen der AB-Adapter (D1) verbunden. Gleichzeitig wird das Ereignis (EO) jedes Flipflops an den Ereigniseingang (E1) des entsprechenden Adapters gesendet, sodass der angeschlossene Consumer über die neuen Daten informiert wird.

Technische Besonderheiten

- **Synchrone Ausgabe**: Alle vier Bytes werden gleichzeitig mit demselben Ereignis (CNF von `SPLIT_DWORD_INTO_BYTES`) ausgegeben. Dadurch bleiben die Konsumenten der einzelnen Bytes stets synchron.
- **Verwendung von E_D_FF_ANY**: Diese flankengesteuerten Flipflops sorgen dafür, dass die Daten erst bei einem neuen Ereignis übernommen werden und bis dahin stabil am Ausgang anliegen.
- **Adapter-Basierte Schnittstelle**: Der Baustein arbeitet vollständig über Adapter, wodurch er sich nahtlos in eine adapterorientierte Architektur einfügt (z.B. bei Verwendung von `adapter::types::unidirectional::AB` und `AD`).

Zustandsübersicht

Der FB selbst besitzt keinen expliziten Zustandsautomaten – er ist ein Composite aus Standard-Funktionsblöcken. Die interne Ablaufsteuerung erfolgt durch die Ereignis- und Datenflüsse:

- **Warten**: Kein Eingangsereignis ⟶ Ausgabe bleibt unverändert.
- **Aufteilen**: Bei einem Ereignis am IN-Adapter wird einmalig der DWORD aufgeteilt und alle vier Bytes ausgegeben.

Anwendungsszenarien

- **Zerlegung von 32-Bit-Sensordaten** (z.B. Encoder-Werte, AD-Wandler) in einzelne Bytes für die Übertragung über Byte-orientierte Protokolle.
- **Ansteuerung von Peripherie**, die einzelne Bytes als Adress- oder Datenwörter erwartet (z.B. Registerzugriffe).
- **Datensammlung in verteilten Systemen**, bei der ein 32-Bit-Wert auf mehrere unabhängige Verarbeiter aufgeteilt werden muss.

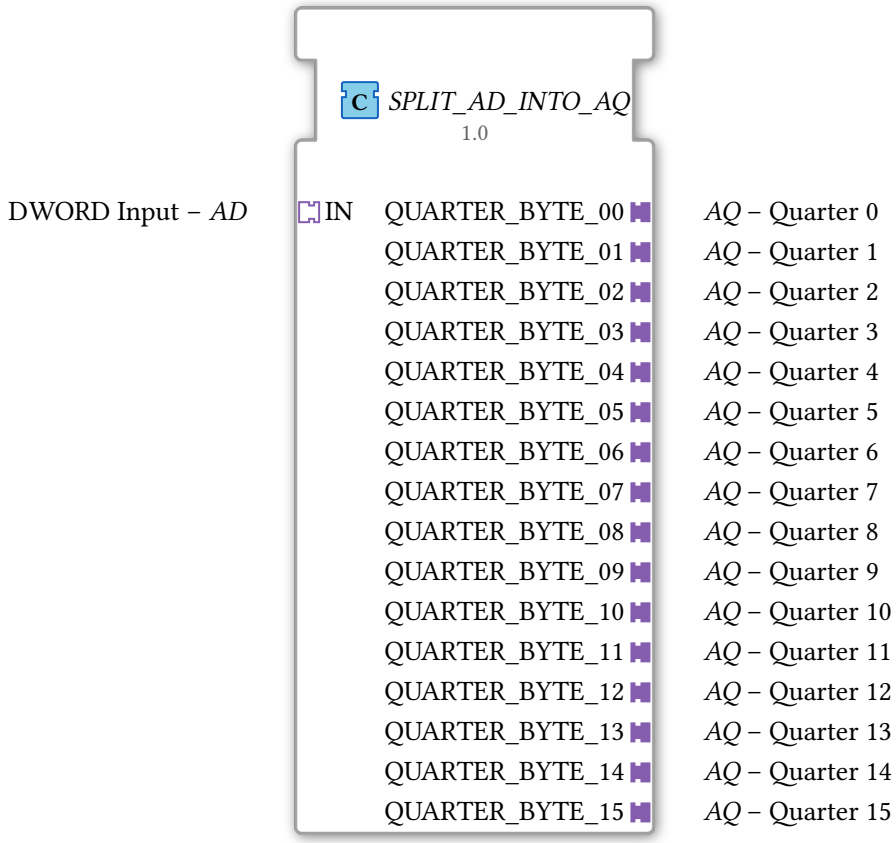
Vergleich mit ähnlichen Bausteinen

- **SPLIT_DWORD_INTO_BYTES**: Zerlegt einen DWORD ebenfalls in Bytes, gibt diese jedoch direkt als Daten-Ausgänge (BYTE_00..BYTE_03) aus, ohne die Daten über Adapter zu leiten.
- **SPLIT_AD_INTO_AB**: Bietet die gleiche Funktionalität, kapselt die Ausgabe jedoch in unidirektionale AB-Adapter. Dies ermöglicht eine saubere Trennung der Schnittstellen und erleichtert die Wiederverwendung in verschiedenen Kontexten.
- **AT** (Adapter-Typ-basierte Bausteine): Andere Adapter-Splitter könnten z.B. WORDs in Nibbles aufteilen, während dieser FB speziell auf DWORD ⟶ 4×BYTE ausgelegt ist.

Fazit

Der **SPLIT_AD_INTO_AB** ist ein nützlicher Composite-FB, der einen 32-Bit-Datenwert synchron und adapterbasiert in vier einzelne Bytes aufspaltet. Durch die Verwendung flankengesteuerter Flipflops bleiben die Ausgänge stabil, bis das nächste Eingangsereignis eintrifft. Die Adapter-Schnittstellen integrieren sich nahtlos in modulare Echtzeitsysteme nach IEC 61499 und erleichtern die strukturierte Datenweitergabe.

SPLIT_AD_INTO_AQ



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `SPLIT_AD_INTO_AQ` zerlegt einen eingehenden AD-Adapter (DWORD) in 16 einzelne AQ-Adapter (QUARTER). Er dient als Schnittstelle, um einen breiten Datenwert (32 Bit) in seine 2-Bit-Quarter-Bestandteile aufzuteilen und diese an separate Ausgabe-Adapter weiterzuleiten. Der Baustein ist als Composite-FB realisiert und nutzt intern einen `SPLIT_DWORD_INTO_QUARTERS`-Baustein sowie 16 Flip-Flops (`E_D_FF_ANY`) zur synchronen Weitergabe.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine eigenständigen Ereignis-Eingänge. Das Ereignis wird über die Adapter-Schnittstelle `IN.E1` empfangen und intern weiterverarbeitet.

Ereignis-Ausgänge

Der FB besitzt keine eigenständigen Ereignis-Ausgänge. Die Ereignisausgabe erfolgt über die AQ-Adapter-Plugs (jeweils über `QUARTER_BYTE_xx.E1`).

Daten-Eingänge

Der FB besitzt keine eigenständigen Daten-Eingänge. Die Daten werden über die Adapter-Schnittstelle `IN.D1` bereitgestellt.

Daten-Ausgänge

Der FB besitzt keine eigenständigen Daten-Ausgänge. Die aufgeteilten Daten werden über die AQ-Adapter-Plugs (jeweils über `QUARTER_BYTE_xx.D1`) ausgegeben.

Adapter

Name	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional:AD	Socket (Eingang)	DWORD-Eingangsadapter (32 Bit). Über E1 und D1 werden Ereignis und Daten empfangen.
QUARTER_BYTE_00 ... QUARTER_BYTE_15	adapter::types::unidirectional:AQ	Plug (Ausgang)	16 Ausgangsadapter, die jeweils ein Quarter (2 Bit) des ursprünglichen DWORDs bereitstellen. Jeder Adapter besitzt einen Ereignisausgang E1 und einen Datenausgang D1.

Funktionsweise

Der Baustein arbeitet rein intern auf Basis eines Composite-Netzwerks. Sobald über den IN-Adapter ein Ereignis (E1) eintrifft, wird dieses an den internen Baustein SPLIT_DWORD_INTO_QUARTERS weitergeleitet. Dieser zerlegt den angelegten DWORD-Wert (IN.D1) in 16 separate Quarter-Bytes (jeweils 2 Bit).

Nach Abschluss der Zerteilung signalisiert SPLIT_DWORD_INTO_QUARTERS.CNF einen Taktimpuls, der gleichzeitig an alle 16 Flip-Flops (E_D_FF_ANY_00 bis E_D_FF_ANY_15) geschickt wird. Die Flip-Flops übernehmen sodann die jeweiligen Quarter-Daten an ihrem Dateneingang (von SPLIT_DWORD_INTO_QUARTERS.QUARTER_BYTE_xx) und geben diese an ihrem Ausgang Q aus. Gleichzeitig feuern sie ein Ereignis an ihrem Ausgangsanschluss EO, welches an den entsprechenden AQ-Adapter-Plug (QUARTER_BYTE_xx.E1) weitergeleitet wird. Dadurch wird der ausgegebene Datenwert (Q) über den Adapter-Datenpfad (D1) zum Zielbaustein transportiert.

Die Verarbeitung erfolgt streng synchron: alle 16 Quarter-Werte werden im selben Takt aktualisiert.

Technische Besonderheiten

- **Composite-Architektur**: Der FB nutzt andere Bausteine (SPLIT_DWORD_INTO_QUARTERS und E_D_FF_ANY) zur Realisierung der Zerlegung und Synchronisation.
- **Keine eigenständigen Ein-/Ausgänge**: Die gesamte Kommunikation erfolgt ausschließlich über Adapter. Dies ermöglicht eine saubere Kapselung und Wiederverwendung in adapterbasierten Architekturen.
- **Synchronisation**: Alle 16 AQ-Ausgänge werden durch einen gemeinsamen Takt (vom SPLIT_DWORD_INTO_QUARTERS.CNF) gleichzeitig aktualisiert. Somit ist gewährleistet, dass die aufgeteilten Daten konsistent zum gleichen Zeitpunkt anliegen.
- **Skalierbarkeit**: Der FB ist auf 16 Quarter (entsprechend 32 Bit) ausgelegt. Eine Anpassung auf andere Bitbreiten wäre durch Änderung der internen Struktur möglich.

Zustandsübersicht

Als Composite-FB besitzt SPLIT_AD_INTO_AQ keinen eigenen Zustandsautomaten. Die interne Datenverarbeitung ist bestimmt durch:

- **Warten auf Ereignis**: Im Leerlauf wird kein internes Ereignis erzeugt.
- **Verarbeitung**: Beim Eintreffen von IN.E1 werden Split und Aktualisierung der Flip-Flops durchgeführt.
- **Ausgabe**: Unmittelbar nach dem Takt liegen alle AQ-Ausgänge an.

Anwendungsszenarien

- **Datenaufteilung in Steuerungssystemen**: Wenn ein Sensor oder ein Kommunikationsmodul ein 32-Bit-Datenwort (z.B. Encoder-Position, Messwert) liefert, das auf 16 separate 2-Bit-Signale (z.B. Helligkeitssensoren, Schaltzustände) aufgeteilt werden muss.
- **Adapter-basierte Kommunikation**: Einsatz in der 4diac-IDE, wenn Adapter nach IEC 61499 genutzt werden, um Daten zwischen verschiedenen Komponenten zu verteilen.
- **Parallelverarbeitung von Teilinformationen**: Zerlegung eines DWORDs für nachgelagerte Bausteine, die jeweils nur 2 Bit des Originals benötigen.

Vergleich mit ähnlichen Bausteinen

Baustein	Beschreibung	Unterschied zu `SPLIT_AD_INTO_AQ`
`SPLIT_DWORD_INTO_QUARTERS`	Zerlegt einen DWORD in 16 Quarter-Werte und gibt diese als direkte Datenausgänge aus.	`SPLIT_AD_INTO_AQ` kapselt diese Zerlegung zusätzlich in Adapter-Schnittstellen und fügt eine Flip-Flop-Synchronisation hinzu.
`SPLIT_INT_INTO_BITS`	Teilt ein Integer in einzelne Bits auf.	Arbeitet auf Bit-Ebene und nicht auf 2-Bit-Quarters; Ausgabe erfolgt typischerweise als boolesche Werte.
Manuelle Aufteilung mit `MUX` oder `DEMUX`	Könnte verwendet werden, um eine Datenaufteilung ohne Adapter zu realisieren.	`SPLIT_AD_INTO_AQ` ist speziell für die Adapter-Kommunikation optimiert und bietet eine gebündelte, synchronisierte Lösung.

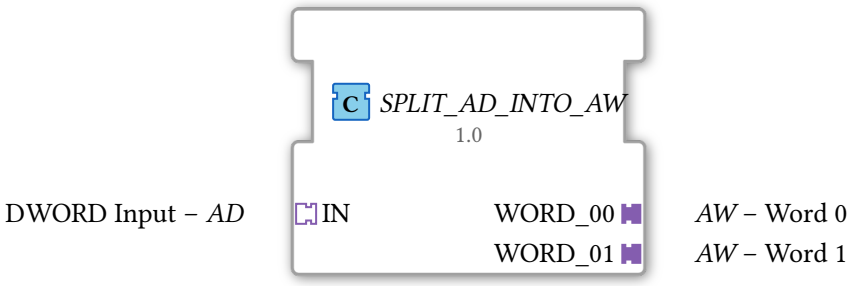
Fazit

`SPLIT_AD_INTO_AQ` ist ein nützlicher Baustein zur Aufteilung eines DWORDs (AD-Adapter) in 16 2-Bit-Quarter-Adapter (AQ). Seine composite Architektur mit interner Synchronisation sorgt für eine konsistente Datenweitergabe und erleichtert die modulare, adapterbasierte Programmierung in der 4diac-IDE. Besonders geeignet ist er für Anwendungen, die eine parallele Verarbeitung von Teilinformationen erfordern, ohne dass die Details der Aufteilung im übergeordneten Netzwerk sichtbar werden müssen.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SPLIT_AD_INTO_AW



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **SPLIT_AD_INTO_AW** dient dazu, einen 32-Bit-DWORD-Wert, der über einen **AD**-Adapter (Unidirectional) anliegt, in zwei 16-Bit-WORD-Werte aufzuteilen und diese über zwei separate **AW**-Adapter (Unidirectional) auszugeben. Die Aufteilung erfolgt ereignisgesteuert, und die Ausgabedaten werden durch Flip-Flops stabil gehalten, bis ein neuer Trigger eintrifft. Der Baustein ist als zusammengesetzter FB (Composite) realisiert und nutzt intern den Baustein **SPLIT_DWORD_INTO_WORDS** sowie zwei **E_D_FF_ANY**-Flip-Flops.

Schnittstellenstruktur

Ereignis-Eingänge

Bezeichnung	Adapter	Ereignis	Beschreibung

IN	AD (Socket)	E1	Startet die Aufteilung des anliegenden DWORD-Wertes.

Ereignis-Ausgänge

Bezeichnung	Adapter	Ereignis	Beschreibung
WORD_00	AW (Plug)	E1	Signalisiert, dass das niederwertige WORD (Low-Word) gültig ist.
WORD_01	AW (Plug)	E1	Signalisiert, dass das höherwertige WORD (High-Word) gültig ist.

Daten-Eingänge

Bezeichnung	Adapter	Datentyp	Beschreibung
IN	AD (Socket)	D1 (DWORD)	32-Bit-Eingangswert, der aufgeteilt wird.

Daten-Ausgänge

Bezeichnung	Adapter	Datentyp	Beschreibung
WORD_00	AW (Plug)	D1 (WORD)	Niederwertiges 16-Bit-Wort des Eingangs-DWORD.
WORD_01	AW (Plug)	D1 (WORD)	Höherwertiges 16-Bit-Wort des Eingangs-DWORD.

Adapter

Die Adapter sind als **unidirectional** ausgelegt:

- **AD** (Eingang): Über einen Socket bereitgestellt, enthält je einen Ereignis- (E1) und Daten- (D1) Kanal.
- **AW** (Ausgang): Über zwei Plugs bereitgestellt, jeweils mit einem Ereignis- (E1) und Daten- (D1) Kanal.

Funktionsweise

1. Ein Ereignis am Eingang **IN.E1** triggert die Verarbeitung.
2. Der aktuelle DWORD-Wert von **IN.D1** wird gelesen und an den internen Baustein **SPLIT_DWORD_INTO_WORDS** weitergeleitet.
3. Dieser teilt den 32-Bit-Wert in zwei 16-Bit-Wörter: **WORD_00** (niederwertig) und **WORD_01** (höherwertig).
4. Die beiden Ergebnisse werden parallel in die Flip-Flops **E_D_FF_ANY_00** und **E_D_FF_ANY_01** übernommen und dort bis zum nächsten Ereignis gehalten.
5. Gleichzeitig werden die Ausgangs-Ereignisse **WORD_00.E1** und **WORD_01.E1** ausgelöst, die anzeigen, dass die entsprechenden Daten (**WORD_00.D1** und **WORD_01.D1**) gültig sind.

Technische Besonderheiten

- **Datensynchronisation:** Durch die Flip-Flops bleiben die Ausgangsdaten stabil, auch wenn der Eingangswert zwischen zwei Ereignissen wechselt. Erst ein neues Ereignis am Eingang aktualisiert beide Ausgänge.
- **Gleichzeitige Ausgabe:** Beide WORD-Ausgänge werden durch dasselbe Eingangsereignis aktualisiert – es gibt keine zeitliche Staffelung.
- **Adapter-Konformität:** Der FB ist speziell für die Verwendung mit **AD-** und **AW-** Adaptern ausgelegt und erlaubt eine nahtlose Einbindung in adapterbasierte Datenflüsse.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten. Das Verhalten ist rein ereignisgesteuert:

- **Idle:** Warten auf ein Ereignis an **IN.E1**.
- **Verarbeitung:** Bei Eintreffen von **IN.E1** wird die Aufteilung durchgeführt und die Flip-Flops gesetzt.
- **Ausgabe:** Die Ereignisse an den Ausgängen werden genau einmal nach der Verarbeitung ausgelöst.

Anwendungsszenarien

- **Adapter-basierte Zerlegung:** Trennen einer 32-Bit-Größe (z. B. aus einem Sensor-Datenwort) in zwei 16-Bit-Teile für getrennte Verarbeitungsketten.
- **Kommunikationsprotokolle:** Aufbereitung von Daten für Schnittstellen, die WORD-weise übertragen (z. B. Modbus-Register oder CAN-Nachrichten).
- **Bit-Manipulation:** Vorverarbeitung für nachfolgende Bausteine, die nur WORD-Adapter erwarten.

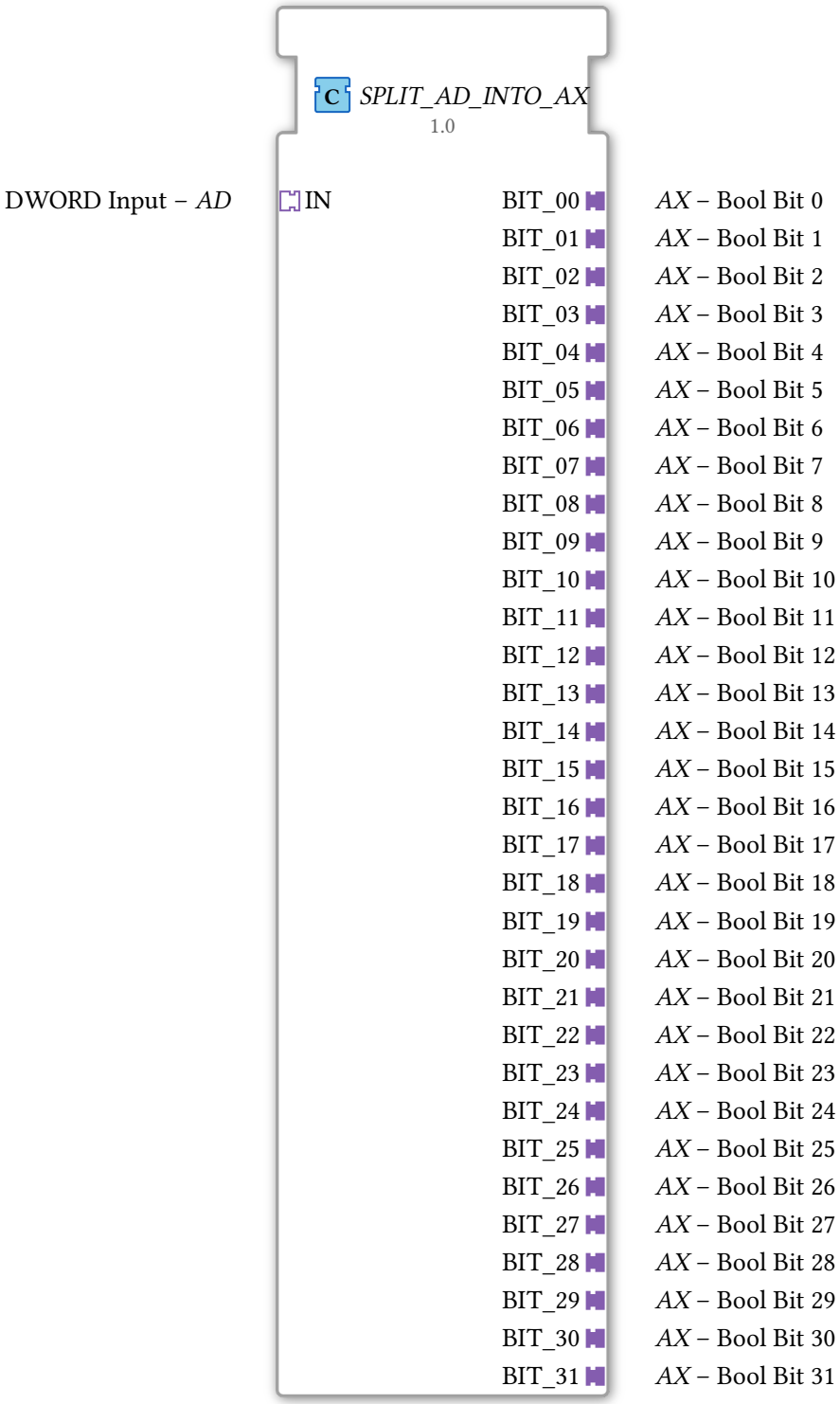
Vergleich mit ähnlichen Bausteinen

Baustein	Beschreibung	Unterschied
SPLIT_DWORD_INTO_WORDS	Reine Datenaufteilung ohne Ereignisausgabe oder Adapter.	Liefert nur Datenausgänge, keine Ereignisse und keine Adapter-Anbindung.
SPLIT_AD_INTO_AW	Adapter-basierte Variante mit stabilisierten Ausgängen.	Integriert Flip-Flops und Ereignisausgabe, speziell für AD-/AW-Adapter.

Fazit

SPLIT_AD_INTO_AW ist ein praktischer zusammengesetzter Funktionsblock, der die Aufteilung eines DWORD-Adapter-Signals in zwei WORD-Adapter-Signale vereinfacht. Durch die eingebauten Flip-Flops und die synchronisierte Ereignisausgabe eignet er sich besonders für zeitkritische und adapterorientierte Anwendungen in industriellen Steuerungen.

SPLIT_AD_INTO_AX



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock SPLIT_AD_INTO_AX dient dazu, ein über einen AD-Adapter (DWORD) eingehendes 32-Bit-Datenwort in 32 separate AX-Adapter (BOOL) aufzutrennen. Jeder AX-Adapter repräsentiert dabei ein einzelnes Bit des ursprünglichen DWORD-Wertes. Der Baustein kapselt die erforderliche Ereignissteuerung und Datenspeicherung in einem modularen, einfach nutzbaren Funktionsblock.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Die Ereignissteuerung erfolgt indirekt über den Adapter-Socket.

Ereignis-Ausgänge

Keine. Die Ausgabe erfolgt über die Adapter-Plugs, deren internes Ereignisverhalten durch die verwendeten Typen (AX) definiert wird.

Daten-Eingänge

Bezeichnung	Typ	Kommentar
IN	AD	DWORD Input (32 Bit)

Die Daten werden über den Adapter-Socket IN empfangen. Der AD-Adapter stellt einen DWORD-Wert sowie ein zugehöriges Ereignis (E1) bereit.

Daten-Ausgänge

Bezeichnung	Typ	Kommentar
BIT_00	AX	Bool Bit 0
BIT_01	AX	Bool Bit 1
…	…	…
BIT_31	AX	Bool Bit 31

Alle 32 Ausgangsadapter sind vom Typ AX (unidirektionaler BOOL-Adapter). Jeder Ausgang gibt den Zustand des entsprechenden Bits des eingehenden DWORD aus.

Adapter

- **Socket**: `IN` (Typ `adapter::types::unidirectional::AD`) – dient dem Empfang des DWORD-Werts und des zugehörigen Ereignisses.
- **Plugs**: 32 Adapter `BIT_00` … `BIT_31` (Typ `adapter::types::unidirectional::AX`) – stellen die einzelnen Bits als boolesche Signale an die verbundene Logik bereit.

Funktionsweise

Der interne Ablauf des FBs gliedert sich in zwei Schritte:

1. **Aufteilung des DWORD in Bool-Werte**
   Ein interner FB `SPLIT_DWORD_INTO_BOOLS` empfängt den DWORD über die Datenverbindung `IN.D1`. Er teilt die 32 Bit in einzelne boolesche Signale (`BIT_00` … `BIT_31`) auf.
2. **Synchronisation und Speicherung**
   Das Ereignis `IN.E1` triggert den Eingang `REQ` des Splitters. Nach vollständiger Verarbeitung sendet dieser das Bestätigungsereignis `CNF`. Dieses Ereignis wird an die Takteingänge (CLK) von 32 Flip-Flops (Typ `E_D_FF`) weitergeleitet. Die Flip-Flops übernehmen gleichzeitig die von `SPLIT_DWORD_INTO_BOOLS` bereitgestellten Bool-Werte auf ihren Dateneingängen `D`.
   Die Ausgänge `Q` der Flip-Flops sind fest mit den Daten-Eingängen `D1` der zugehörigen AX-Adapter verbunden, sodass die gespeicherten Bitwerte unmittelbar an die Ausgangsadapter weitergegeben werden.

Somit wird nach jedem Ereignis am Eingangsadapter der gesamte 32-Bit-Wert parallel auf die 32 Ausgangsadapter übertragen und dort bis zum nächsten Ereignis gehalten.

Technische Besonderheiten

- **Parallele Verarbeitung**: Alle 32 Bits werden gleichzeitig durch die Flip-Flops übernommen, was eine konsistente Momentaufnahme des DWORD gewährleistet.
- **Modularer Aufbau**: Der FB nutzt vorhandene Standardbausteine (`SPLIT_DWORD_INTO_BOOLS` und `E_D_FF`), was Wartbarkeit und Wiederverwendung erleichtert.
- **Adapterbasierte Kommunikation**: Die gesamte Ein- und Ausgabe erfolgt über Adapter, sodass der FB nahtlos in adapterbasierte Anwendungen integriert werden kann.

- **Keine externen Ereignisse**: Der FB besitzt keine eigenen Ereignis-Ein-/Ausgänge, da die Ereignissteuerung vollständig über die Adapter abgewickelt wird.

Zustandsübersicht

Der FB selbst ist zustandslos – seine Funktionalität wird durch die inneren Flip-Flops realisiert. Jeder `E_D_FF` besitzt einen internen Speicherzustand (0 oder 1), der durch das Taktereignis aktualisiert wird. Nach dem Start befinden sich alle Flip-Flops im initialen Zustand (0), bis das erste Ereignis am Eingang eintrifft.

Anwendungsszenarien

- **Digitale Signalverarbeitung**: Zerlegung eines 32-Bit-Datenworts in einzelne boolesche Steuersignale, z. B. für Status- oder Freigabebits.
- **Schnittstellenanpassung**: Verbindung einer DWORD-Quelle (z. B. Bussystem, Register) mit diskreten digitalen Ein-/Ausgängen.
- **Test und Simulation**: Gezielte Analyse einzelner Bits eines Datenworts ohne aufwändige Bit-Manipulation im Anwendungscode.

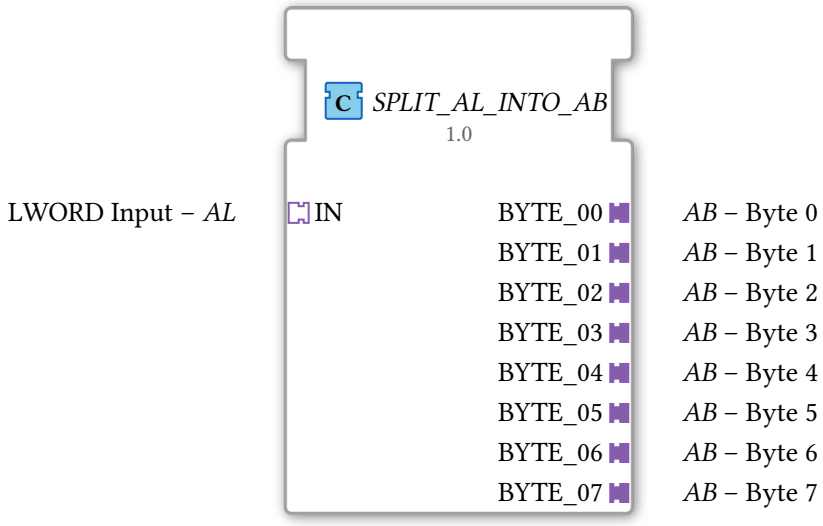
Vergleich mit ähnlichen Bausteinen

- **SPLIT_WORD_INTO_BOOLS / SPLIT_BYTE_INTO_BOOLS**: Diese Bausteine arbeiten auf 16- bzw. 8-Bit-Ebene und sind für kleinere Datenwortbreiten optimiert. SPLIT_AD_INTO_AX ist speziell für 32-Bit-Wörter ausgelegt und nutzt Adapter statt direkter Datenports.
- **Direkte Bit-Extraktion mit Adaptern**: Alternativ könnte man den `SPLIT_DWORD_INTO_BOOLS` direkt in einer Applikation einsetzen und die Ausgänge über Konnektoren verbinden. Der vorliegende FB kapselt jedoch die gesamte Synchronisation und stellt eine einheitliche adapterbasierte Schnittstelle bereit, was die Wiederverwendung vereinfacht.

Fazit

SPLIT_AD_INTO_AX ist ein praktischer Funktionsblock zur Aufteilung eines 32-Bit-Datenworts in 32 einzelne boolesche Adaptersignale. Durch den modularen Aufbau, die parallele Verarbeitung und die adapterbasierte Kommunikation eignet er sich hervorragend für den Einsatz in komplexen Automatisierungslösungen, bei denen Bit-Informationen aus einem kompakten Datenwort in getrennte logische Pfade aufgelöst werden müssen.

SPLIT_AL_INTO_AB



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **SPLIT_AL_INTO_AB** dient dazu, einen über einen **AL-Adapter** (Long Word) eingehenden 64-Bit-Wert (LWORD) in seine acht einzelnen Bytes aufzuteilen und diese über acht separate **AB-Adapter** (Byte) bereitzustellen. Die Aufteilung erfolgt synchron zu einem Ereignis am Eingangsadapter, und jedes Byte wird über ein eigenes Flipflop (E_D_FF_ANY) zwischengespeichert, bevor es am

entsprechenden Ausgangsadapter anliegt. Der Baustein ist als **Adapter-Splitting-Komponente** konzipiert und eignet sich besonders für die strukturierte Datenzerlegung in IEC 61499-Systemen.

Schnittstellenstruktur

Ereignis-Eingänge

- **IN** (Socket, Typ: `adapter::types::unidirectional::AL`): Der integrierte Ereignis-Ausgang **E1** des AL-Adapters löst die Verarbeitung aus. Ein eingehendes Ereignis am Adapter IN startet die Zerlegung des gleichzeitig anliegenden Datenwertes.

Ereignis-Ausgänge

Die Ausgangsadapter **BYTE_00** bis **BYTE_07** (Plugs, Typ: `adapter::types::unidirectional::AB`) verfügen jeweils über einen Ereignis-Eingang **E1**, der nach erfolgter Datenübernahme ausgelöst wird. Diese Ereignisse werden vom internen Netzwerk generiert und signalisieren, dass das zugehörige Byte gültig am Datenausgang des Adapters anliegt.

Daten-Eingänge

- **IN** (Socket, Typ: `adapter::types::unidirectional::AL`): Liefert den zu zerlegenden 64-Bit-Datenwert über den Daten-Eingang **D1** des Adapters (Typ LWORD).

Daten-Ausgänge

- **BYTE_00** bis **BYTE_07** (Plugs, Typ: `adapter::types::unidirectional::AB`): Jeder dieser Adapter stellt über seinen Daten-Ausgang **D1** ein einzelnes Byte (BYTE) zur Verfügung. Die Zuordnung ist:
  - BYTE_00: niederwertigstes Byte (Bit 0…7)
  - BYTE_01: nächstes Byte (Bit 8…15)
  - …
  - BYTE_07: höchstwertigstes Byte (Bit 56…63)

Adapter

- **IN**: Socket (AL – Adapter Long Word) – aufnehmender Adapter
- **BYTE_00 … BYTE_07**: Plugs (AB – Adapter Byte) – abgebende Adapter

Funktionsweise

1. Ein Ereignis am Eingangsadapter **IN** (E1) triggert den internen Funktionsblock **SPLIT_LWORD_INTO_BYTES** (Typ: `eclipse4diac::utils::splitting::SPLIT_LWORD_INTO_BYTES`).
2. Gleichzeitig wird der vom Adapter IN bereitgestellte LWORD-Wert (D1) an den Eingang **IN** dieses internen Bausteins weitergeleitet.
3. Der interne Splitter zerlegt den LWORD in acht separate Bytes (**BYTE_00** … **BYTE_07**) und legt diese an seinen Datenausgängen an.
4. Das Abschlussereignis **CNF** des Splitters wird parallel an acht **E_D_FF_ANY**-Flipflops gesendet. Jedes Flipflop übernimmt mit diesem Takt das ihm zugeordnete Byte und gibt es an seinem Ausgang **Q** aus.
5. Die Flipflops erzeugen ihrerseits ein Ausgangsereignis **EO**, das direkt mit dem Ereigniseingang **E1** des entsprechenden **BYTE_XX**-Adapters verbunden ist.
6. Dadurch werden die einzelnen Bytes zeitgleich über die acht AB-Adapter nach außen gegeben – sowohl als Datenwert (Q → D1) als auch als Bestätigungsereignis (EO → E1).

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet ausschließlich Adapter (AL und AB) und keine herkömmlichen Ereignis-/Datenports. Dies ermöglicht eine modulare Kapselung und Wiederverwendung in Adapter-basierten Architekturen.
- **Gleichzeitige Ausgabe**: Die acht Bytes werden durch die parallele Taktung aller Flipflops gleichzeitig ausgegeben, was für zeitkritische Anwendungen vorteilhaft ist.
- **Zwischenspeicherung**: Jedes Byte wird in einem eigenen Flipflop (E_D_FF_ANY) zwischengespeichert, sodass der Ausgangswert auch dann erhalten bleibt, wenn der Eingangswert wechselt oder ein neuer Zerlegungszyklus beginnt.
- **Unidirektionale Datenfluss**: Die Adapter sind als unidirektional (IN: AL, OUT: AB) ausgelegt – der Datenfluss erfolgt nur vom Eingang zu den Ausgängen.

- **Keine Zustandsautomaten im FB selbst**: Der FB delegiert die Zustandshaltung an die internen Flipflops; er selbst ist rein kombinatorisch mit einer getakteten Übernahme.

Zustandsübersicht

Der Baustein selbst besitzt keinen eigenen, sichtbaren Zustandsautomaten. Die internen Flipflops **E_D_FF_ANY** besitzen jedoch einen Zweizustand (Q = 0 oder 1). Der gesamte Ablauf folgt einem festen Schema:

- **Idle**: Kein Ereignis am Eingang – Ausgänge behalten ihre letzten Werte.
- **Processing**: Ein Ereignis am IN-Adapter wird verarbeitet.
- **Output**: Nach der Taktung der Flipflops stehen alle acht Bytes gleichzeitig an den Ausgangsadaptern an und werden durch die jeweiligen Ereignisse bestätigt.

Anwendungsszenarien

- **Dekomposition von Kommunikationsdaten**: Zerlegung eines empfangenen LWORD-Pakets (z. B. aus einem Feldbus oder einer seriellen Schnittstelle) in einzelne Byte-Signale für die weitere Verarbeitung in der Steuerung.
- **Parallele Ausgabe von Diagnose-/Statusbytes**: Aufteilung eines zusammengesetzten 64-Bit-Statuswortes auf acht getrennte Byte-Ausgänge zur Anzeige oder Weiterleitung.
- **Adapterorientierte Datenverarbeitung**: Einbindung in Architekturen, die auf Adaptern basieren, um eine strikte Trennung von Daten- und Ereignisschnittstellen zu gewährleisten.
- **Schnittstelle zu Unterbausteinen**: Versorgung mehrerer FB, die jeweils ein Byte benötigen (z. B. für sieben Segmentanzeigen, Schrittmotoransteuerung oder D/A-Wandler).

Vergleich mit ähnlichen Bausteinen

- **SPLIT_LWORD_INTO_BYTES**: Dieser interne FB führt die reine Aufteilung ohne Adapter aus und gibt einfache Datenausgänge (BYTE) sowie ein Ereignis (CNF) aus. Im Gegensatz dazu bietet **SPLIT_AL_INTO_AB** eine vollständige Adapterkapselung inklusive Zwischenspeicherung und synchronisierter Ereignisausgabe.
- **SPLIT_DWORD_INTO_BYTES** (analog, falls vorhanden): Zerlegt einen 32-Bit-Wert in vier Bytes. SPLIT_AL_INTO_AB ist auf 64 Bit ausgelegt und liefert acht Bytes.
- **Einfache MOVE-FBs**: Mehrere MOVE-Bausteine mit Bitmasken müssten einzeln konfiguriert werden, während dieser Baustein eine fertige, integrierte Lösung bietet.
- **Adapterbasierte Alternativen**: Andere Splitting-Adapter könnten symetrische oder andere Adaptertypen verwenden; dieser FB ist auf AL⟶AB spezialisiert.

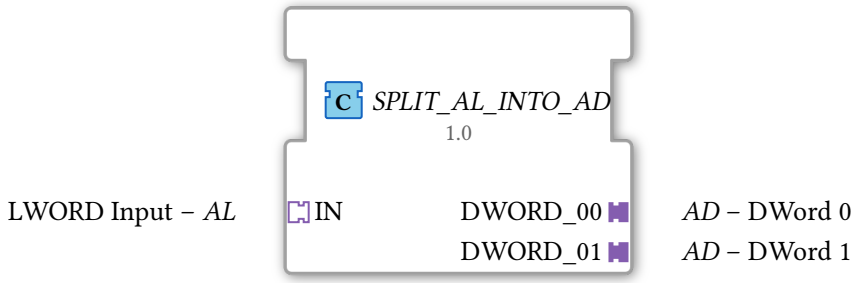
Fazit

SPLIT_AL_INTO_AB ist ein nützlicher und effizienter Funktionsblock zum Aufteilen eines 64-Bit-Datenwortes in acht einzelne Bytes, der speziell für den Einsatz mit IEC 61499-Adaptern (AL/AB) entwickelt wurde. Die Kombination aus internem Splitter und Flipflops gewährleistet eine gleichzeitige, pufferge-stützte Ausgabe aller Bytes. Dank der adapterbasierten Schnittstelle lässt sich der Baustein nahtlos in modulare Steuerungsarchitekturen integrieren, bei denen eine saubere Trennung von Ereignis- und Datenkanälen gewünscht ist. Er eignet sich besonders für Anwendungen, die eine parallele, strukturierte Datenweitergabe erfordern.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SPLIT_AL_INTO_AD



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `SPLIT_AL_INTO_AD` dient dazu, einen eingehenden **AL (LWORD)**-Adapter in zwei separate **AD (DWORD)**-Adapter aufzuteilen. Er realisiert damit die notwendige Schnittstellenwandlung für Anwendungen, bei denen ein großes Datenwort (LWORD) in zwei kleinere DWORDs zerlegt und über unidirektionale Adapter bereitgestellt werden muss.

Schnittstellenstruktur

Ereignis-Eingänge

- **nicht direkt vorhanden**
  Ereignisse werden über den SOCKET-Adapter `IN` (Signale `IN.E1`) empfangen. Intern löst dies die Verarbeitung aus.

Ereignis-Ausgänge

- **nicht direkt vorhanden**
  Die Ausgabeereignisse erfolgen über die PLUG-Adapter `DWORD_00.E1` und `DWORD_01.E1`, sobald ein neuer Wert bereitsteht.

Daten-Eingänge

- **IN.D1** – Eingangsdaten (`LWORD`) aus dem SOCKET-Adapter `IN`.

Daten-Ausgänge

- **DWORD_00.D1** – Erstes aufgeteiltes DWORD (Low-Teil) am PLUG-Adapter `DWORD_00`.
- **DWORD_01.D1** – Zweites aufgeteiltes DWORD (High-Teil) am PLUG-Adapter `DWORD_01`.

Adapter

Typ	Name	Richtung	Kommentar
SOCKET	`IN`	Eingang	AL (LWORD) – Quelladapter
PLUG	`DWORD_00`	Ausgang	AD (DWORD) – erster Zieladapter
PLUG	`DWORD_01`	Ausgang	AD (DWORD) – zweiter Zieladapter

Funktionsweise

Der Baustein arbeitet rein ereignisgesteuert. Sobald am SOCKET `IN` ein Ereignis eintrifft (`IN.E1`), wird das zugehörige Datenwort (`IN.D1`, Typ `LWORD`) intern an den Unterbaustein `SPLIT_LWORD_INTO_DWORDS` weitergeleitet. Dieser zerlegt das 64-Bit-Wort in zwei 32-Bit-DWORDs.

Die beiden Ergebnisse werden parallel an die Eingänge der Flip-Flop-Bausteine `E_D_FF_ANY_00` und `E_D_FF_ANY_01` übergeben. Gleichzeitig löst der Splitter-Baustein ein gemeinsames Bestätigungsereignis (`CNF`) aus, das beide Flip-Flops taktet. Dadurch werden die neuen DWORD-Werte übernommen und als gültige Ausgangsdaten an den jeweiligen PLUG-Adapter (`DWORD_00.D1` und `DWORD_01.D1`) ausgegeben. Gleichzeitig wird ein Ereignis an den entsprechenden Ausgangsadapter gesendet (`DWORD_00.E1` bzw. `DWORD_01.E1`).

Die Flip-Flops stellen sicher, dass die Ausgangsdaten bis zum nächsten gültigen Eingabeereignis stabil bleiben und nur bei einer neuen Verarbeitung aktualisiert werden.

Technische Besonderheiten

- **Vollständige Adapter-Kapselung** – Der Baustein hat keine direkten Events oder Datenports, sondern kommuniziert ausschließlich über standardisierte unidirektionale Adapter (AL/AD).
- **Interne Nutzung von `SPLIT_LWORD_INTO_DWORDS`** – Dieser separate, typisierte Splitter-Baustein übernimmt die reine Datenaufteilung; die Flip-Flops entkoppeln Daten- und Ereignisweitergabe.
- **Null-Latenz bei gleichzeitigen Ereignissen** – Da beide Flip-Flops durch dasselbe `CNF` getaktet werden, werden beide Ausgänge stets synchron aktualisiert.
- **Kein eigenes Zustandsverhalten** – Der Baustein ist rein kombinatorisch mit ereignisgesteuerter Speicherung; es existiert keine interne Zustandsmaschine.

Zustandsübersicht

Der Funktionsblock besitzt keinen expliziten Zustandsautomaten. Sein Verhalten ist rein daten- und ereignisgesteuert:

- **Idle** – Kein Eingabeereignis; Ausgangsdaten bleiben unverändert.
- **Processing** – Nach Eintreffen von `IN.E1` werden die Daten aufgeteilt und an die Ausgänge übergeben.
- **Stable** – Die Flip-Flops halten die Werte bis zum nächsten Ereignis.

Anwendungsszenarien

- **Industrielle Automatisierung** – Wenn ein Sensor/Aktor ein LWORD-Datenwort (z. B. 64-Bit-Zähler oder kombinierte Status-/Steuerwörter) über einen AL-Adapter liefert, die nachfolgenden Module aber mit zwei getrennten DWORD-Adaptern arbeiten.
- **Protokoll-Umsetzung** – Zerlegung großer Datenpakete in handhabbare Teilwörter für Adapter-basierte Kommunikationsprotokolle.
- **Datenvalidierung und -weiterleitung** – Aufteilen eines LWORD in zwei DWORDs, um unterschiedliche Verarbeitungspfade (z. B. Visualisierung und Steuerung) parallel zu bedienen.

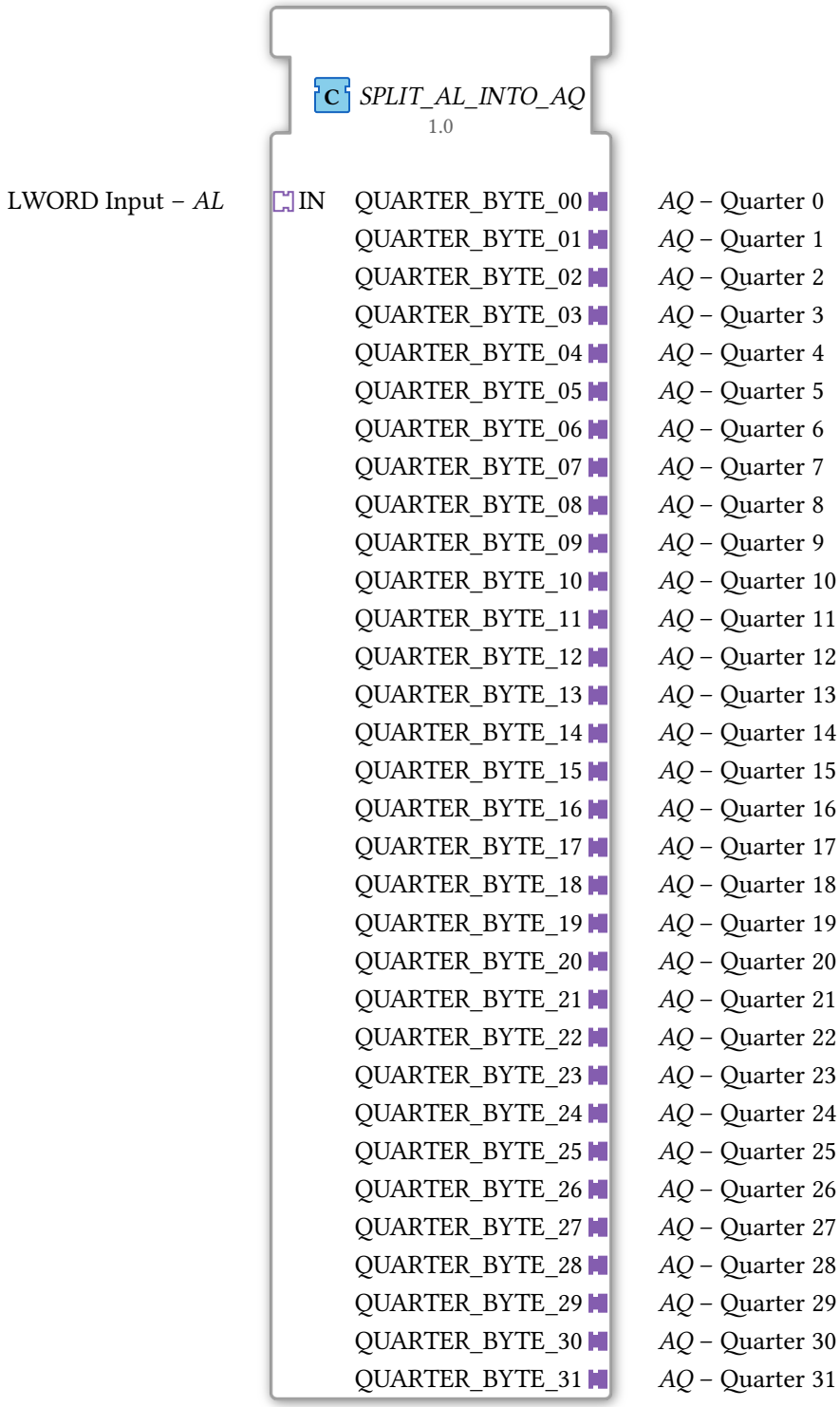
Vergleich mit ähnlichen Bausteinen

- **`SPLIT_LWORD_INTO_DWORDS`** – reiner Datensplitter ohne Adapter-Anbindung und ohne Ereignis-Synchronisation. `SPLIT_AL_INTO_AD` erweitert diesen um die erforderlichen Adapter-Schnittstellen und Flip-Flops zur korrekten Ereignisbehandlung.
- **Manuelle Aufteilung** – Alternativ könnte man die Aufteilung direkt in der Logik des übergeordneten Systems programmieren; der Baustein verringert jedoch den Aufwand und verbessert die Wiederverwendbarkeit.

Fazit

`SPLIT_AL_INTO_AD` ist ein spezialisierter, aber eleganter Adapter-Baustein, der die nahtlose Verbindung zwischen einer LWORD-Quelle und zwei DWORD-Senken ermöglicht. Durch die saubere Trennung von Datenaufteilung und Ereignissynchronisation eignet er sich besonders für modulare, adapterbasierte Steuerungsarchitekturen.

SPLIT_AL_INTO_AQ



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **SPLIT_AL_INTO_AQ** ist ein zusammengesetzter Baustein (Composite-FB), der einen eingehenden LWORD-Wert (über einen `AL`-Adapter) in 32 separate 2-Bit-Werte aufteilt und diese jeweils über einen eigenen `AQ`-Adapter (Quarter-Byte) ausgibt. Die Aufteilung erfolgt synchron zu einem Ereignis, das über den Eingangsadapter bereitgestellt wird. Der Baustein dient als Schnittstelle zwischen einem breiten Datenwort und mehreren schmalen, ereignisgesteuerten Teilsegmenten.

Schnittstellenstruktur

Ereignis-Eingänge

Keine direkten Ereignis-Eingänge. Das auslösende Ereignis wird über den eingehenden Adapter `IN.E1` bereitgestellt.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge. Die Ausgabeereignisse werden über die ausgehenden Adapter `QUARTER_BYTE_xx.E1` weitergegeben.

Daten-Eingänge

Keine direkten Daten-Eingänge. Der LWORD-Datenwert wird über den eingehenden Adapter `IN.D1` eingelesen.

Daten-Ausgänge

Keine direkten Daten-Ausgänge. Die 2-Bit-Datenwerte werden über die ausgehenden Adapter `QUARTER_BYTE_xx.D1` ausgegeben.

Adapter

Richtung	Name	Typ	Kommentar
Socket (Eingang)	`IN`	`adapter::types::unidirectionalLWORD`	LWORD-Eingang (64 Bit)
Plug (Ausgang)	`QUARTER_BYTE_00` .. `QUARTER_BYTE_31`	`adapter::types::unidirectionalQWORD`	32 Ausgänge, jeder liefert einen 2-Bit-Wert (Quarter)

Jeder Adapter verfügt über je einen Ereignis- und einen Datenkanal (`E1`, `D1`).

- Der `AL`-Adapter stellt ein Ereignis (`E1`) und den LWORD-Datenwert (`D1`) bereit.
- Die `AQ`-Adapter empfangen ein Ereignis (`E1`) und den zugehörigen 2-Bit-Wert (`D1`).

Funktionsweise

Der Baustein arbeitet in folgenden Schritten:

1. **Ereignisempfang**: Ein Ereignis am Eingangsadapter `IN.E1` triggert die interne Verarbeitung.
2. **Aufteilung**: Die interne Instanz `SPLIT_LWORD_INTO_QUARTERS` zerlegt den eingehenden LWORD (64 Bit) in 32 aufeinanderfolgende 2-Bit-Segmente (Quarter-Byte 0 bis 31). Jedes Segment wird an den Dateneingang eines von 32 `E_D_FF_ANY`-Flipflops weitergeleitet. Gleichzeitig wird das Ereignis an den Takteingang (`CLK`) aller Flipflops verteilt.
3. **Ausgabe**: Mit der steigenden Flanke des Taktes übernehmen die Flipflops die 2-Bit-Werte. Die Flipflops geben dann über ihre Ausgänge (`Q` und `EO`) die Daten und ein Ausgangsereignis an die entsprechenden `AQ`-Adapter weiter. Damit stehen an den 32 Ausgängen parallel die segmentierten Werte zur Verfügung.

Der gesamte Ablauf erfolgt streng ereignisgesteuert – ein neues Eingangsereignis aktualisiert alle Ausgänge gleichzeitig.

Technische Besonderheiten

- **Verwendung von D-Flipflops**: Die `E_D_FF_ANY`-Bausteine gewährleisten, dass die Ausgabedaten erst nach dem Taktereignis stabil anliegen und nicht durch Zwischenwerte gestört werden.
- **Parallelisierung**: Alle 32 Teilwerte werden in einem Schritt berechnet und ausgegeben. Der Baustein ist damit deterministisch und benötigt keine Schleifen oder sequentielle Abarbeitung.
- **Adapterbasierte Schnittstelle**: Der FB kommuniziert ausschließlich über IEC-61499-Adapterschnittstellen. Dies ermöglicht eine saubere Trennung von Ereignis- und Datenflüssen und erleichtert die Wiederverwendung in unterschiedlichen Kontexten.
- **Einsatz von 32-fach-Verkettung**: Die vertikale Anordnung der Flipflops und Adapter im Netzwerk zeigt eine systematische, aber sehr ausgedehnte Struktur – bei der Implementierung ist auf ausreichende Performance (z. B. Laufzeit des gemeinsamen Taktsignals) zu achten.

Zustandsübersicht

Der Baustein enthält keinen eigenen Zustandsautomaten. Die interne Funktionalität ergibt sich aus der Kombination von:

- **einem** `SPLIT_LWORD_INTO_QUARTERS` (kombinatorische Aufteilung)
- **32** `E_D_FF_ANY` (speichernde Elemente mit den Zuständen gesetzt/rückgesetzt)

Jedes Flipflop speichert den letzten geladenen 2-Bit-Wert. Ein neues Eingangsereignis überschreibt alle 32 Werte gleichzeitig.

Anwendungsszenarien

- **Aufschlüsselung eines Feldbus-Datentelegramms**: Ein LWORD enthält mehrere Status- oder Steuerbits, die auf separate Aktoren oder Sensoren verteilt werden müssen.
- **Parallelisierung von 2-Bit-Signalen**: In der Anbindung von BCD- oder Quadratur-Encodern können mehrere 2-Bit-Informationen kompakt übertragen und dann getrennt weiterverarbeitet werden.
- **Bridge zwischen breitem und schmalem Datenbus**: Wenn ein System mit 64-Bit-Wörtern arbeitet, die Zielkomponenten aber nur 2-Bit-Schnittstellen besitzen, bietet der Baustein eine einfache Aufteilung.

Vergleich mit ähnlichen Bausteinen

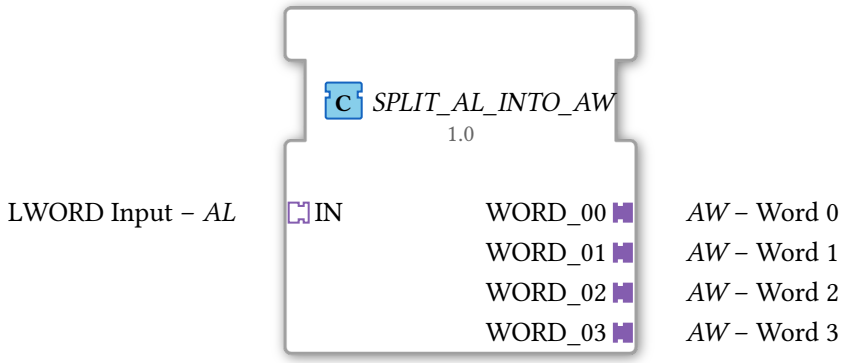
Baustein	Ausgabeformat	Anzahl Ausgänge	Synchronisation
`SPLIT_AL_INTO_AQ`	2-Bit-AQ-Adapter	32	Gemeinsames Ereignis
`SPLIT_LWORD_INTO_BYTES` (hypothetisch)	8-Bit-Adapter	8	Ereignis
`SPLIT_LWORD_INTO_WORDS` (hypothetisch)	16-Bit-Adapter	4	Ereignis

Der vorliegende Baustein ist speziell für die feine Granularität von 2-Bit-Segmenten optimiert und setzt dabei auf die ereignisgetriebene IEC-61499-Adaptertechnik. Der Hauptunterschied zu einfacheren Split-Bausteinen liegt in der Zahl der Ausgänge (32 statt typischen 4 oder 8) sowie der Verwendung von Flipflops zur stabilen Ausgabe.

Fazit

Der `SPLIT_AL_INTO_AQ`-Funktionsblock bietet eine effiziente, parallelisierte Aufteilung eines 64-Bit-LWORD in 32 separate 2-Bit-Kanäle. Dank der strengen Ereignissteuerung und der Speicherung durch D-Flipflops sind deterministische und zeitlich präzise Ausgaben gewährleistet. Der Baustein eignet sich besonders für Anwendungen, bei denen viele schmale Datenströme aus einem kompakten Datenwort abgeleitet werden müssen. Die adapterbasierte Schnittstelle macht ihn flexibel einsetzbar und gut in bestehende IEC-61499-Systeme integrierbar.

SPLIT_AL_INTO_AW



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **SPLIT_AL_INTO_AW** dient dazu, eine über einen AL-Adapter (LWORD) eingehende 64-Bit-Ganzzahl in vier separate 16-Bit-Werte aufzuteilen und diese über vier einzelne AW-Adapter auszugeben. Er realisiert eine festverdrahtete Aufteilung, die durch ein eingehendes Ereignis ausgelöst wird.

Schnittstellenstruktur

Ereignis-Eingänge

- **IN.E1** (Ereignis) – Wird über den Socket **IN** empfangen und startet den Aufteilungsvorgang.

Ereignis-Ausgänge

- **WORD_00.E1** – Wird über den Plug **WORD_00** ausgegeben.
- **WORD_01.E1** – Wird über den Plug **WORD_01** ausgegeben.
- **WORD_02.E1** – Wird über den Plug **WORD_02** ausgegeben.
- **WORD_03.E1** – Wird über den Plug **WORD_03** ausgegeben.

Jeder Ereignisausgang wird parallel aktiviert, sobald die Aufteilung abgeschlossen ist.

Daten-Eingänge

- **IN.D1** (LWORD) – Die 64-Bit-Eingangsgröße, die aufgeteilt werden soll. Bereitgestellt über den Socket **IN**.

Daten-Ausgänge

- **WORD_00.D1** (WORD) – Das niederwertigste Wort (Bits 0–15).
- **WORD_01.D1** (WORD) – Das zweite Wort (Bits 16–31).
- **WORD_02.D1** (WORD) – Das dritte Wort (Bits 32–47).
- **WORD_03.D1** (WORD) – Das höchstwertige Wort (Bits 48–63).

Jeder Ausgang wird durch einen flankengesteuerten Speicher (E_D_FF_ANY) bereitgestellt und bleibt bis zum nächsten Aufteilungsvorgang stabil.

Adapter

- **Socket IN** – Typ: `adapter::types::unidirectional::AL` (LWORD)
- **Plugs WORD_00 … WORD_03** – Typ: `adapter::types::unidirectional::AW` (WORD)

Funktionsweise

1. Ein eingehendes Ereignis an **IN.E1** aktiviert den internen Baustein **SPLIT_LWORD_INTO_WORDS** über dessen Ereigniseingang **REQ**.
2. Gleichzeitig wird das Datum **IN.D1** an den Dateneingang **IN** des Split-Bausteins übergeben.
3. **SPLIT_LWORD_INTO_WORDS** teilt die LWORD in die vier Wortteile (**WORD_00**…**WORD_03**) auf und erzeugt ein Abschlussereignis an **CNF**.
4. Dieses Abschlussereignis wird an alle vier flankengesteuerten Flip-Flops (**E_D_FF_ANY**) weitergeleitet. Jeder Flip-Flop übernimmt das ihm zugeordnete Teilwort an seinem Dateneingang **D** und legt es an seinem Ausgang **Q** an.
5. Gleichzeitig werden die Ereignisausgänge **WORD_00.E1** … **WORD_03.E1** aktiviert, sodass die nachgeschalteten Adapter die neuen Daten gemeinsam übernehmen können.

Technische Besonderheiten

- **Verwendung von E_D_FF_ANY**: Jeder Teilwert wird in einem flankengesteuerten Flip-Flop zwischengespeichert. Dadurch bleibt der Ausgangswert auch dann erhalten, wenn am Eingang kein neues Ereignis anliegt. Erst der nächste Aufteilungsvorgang aktualisiert die Werte.
- **Parallelausgabe**: Alle vier Ausgangsereignisse werden gleichzeitig ausgelöst – die Daten stehen zum gleichen Zeitpunkt an allen Plugs an.
- **Adapterbasierte Schnittstelle**: Der Baustein arbeitet ausschließlich mit Adaptern, was eine saubere Kapselung und Wiederverwendbarkeit in modularen Applikationen ermöglicht.

Zustandsübersicht

Der Baustein selbst besitzt keinen expliziten Zustandsautomaten. Die internen Flip-Flops (E_D_FF_ANY) können folgende Zustände annehmen:

- **Idle**: Kein Ereignis anliegend – die Ausgangsplugs halten den zuletzt gespeicherten Wert.
- **Busy**: Während der Verarbeitung (zwischen **IN.E1** und dem Erscheinen der Ausgangsereignisse) ist der Baustein kurzzeitig belegt.
- **Ausgabe aktiv**: Nach Abschluss werden die neuen Daten an den Plugs ausgegeben und die Ereignisse gesendet.

Anwendungsszenarien

- **De-Serialisierung**: Auftrennen einer 64-Bit-Telegrammkomponente in einzelne 16-Bit-Registerwerte für die Weiterverarbeitung.
- **Steuerungstechnik**: Aufteilen einer langen Bitmaske in mehrere Teilmasken, die an verschiedene Aktoren oder Anzeigeeinheiten gesendet werden.
- **Datenvorbereitung**: Zerlegen eines LWORD-Messwerts (z. B. aus einem Encoder oder einer Analog-Digital-Wandlerkarte) in nutzbare Wort-Kanalwerte.

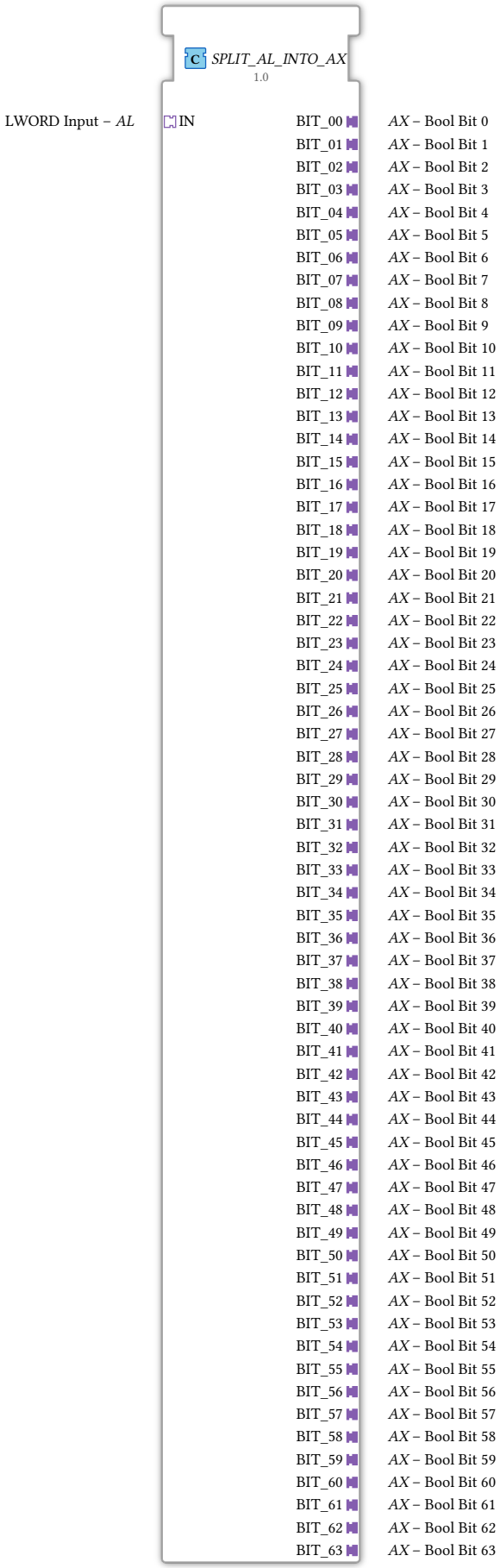
Vergleich mit ähnlichen Bausteinen

- **SPLIT_LWORD_INTO_WORDS** (nicht adapterbasiert): Arbeitet mit reinen Daten- und Ereignisports. **SPLIT_AL_INTO_AW** kapselt diesen Baustein und fügt die Adapterschnittstellen sowie die flankengesteuerte Speicherung hinzu.
- **MUX** oder **DEMUX** (IEC 61499-Standard): Diese trennen oder kombinieren Datenströme meist auf Bit- oder Bitebene, während **SPLIT_AL_INTO_AW** speziell für die feste 64-Bit-Aufteilung ausgelegt ist.

Fazit

SPLIT_AL_INTO_AW bietet eine einfache, robuste Lösung, um eine LWORD über Adapter in vier WORTe aufzuteilen. Durch die integrierte flankengesteuerte Zwischenspeicherung ist der Baustein besonders für ereignisgesteuerte Systeme geeignet, bei denen die Daten parallel und stabil an verschiedenen Ausgängen anliegen müssen. Die adapterbasierte Schnittstelle vereinfacht die Integration in bestehende Komponentenbibliotheken.

SPLIT_AL_INTO_AX


SPLIT_AL_INTO_AX
1.0
LWORD Input – AL
IN
BIT_00 AX – Bool Bit 0
BIT_01 AX – Bool Bit 1
BIT_02 AX – Bool Bit 2
BIT_03 AX – Bool Bit 3
BIT_04 AX – Bool Bit 4
BIT_05 AX – Bool Bit 5
BIT_06 AX – Bool Bit 6
BIT_07 AX – Bool Bit 7
BIT_08 AX – Bool Bit 8
BIT_09 AX – Bool Bit 9
BIT_10 AX – Bool Bit 10
BIT_11 AX – Bool Bit 11
BIT_12 AX – Bool Bit 12
BIT_13 AX – Bool Bit 13
BIT_14 AX – Bool Bit 14
BIT_15 AX – Bool Bit 15
BIT_16 AX – Bool Bit 16
BIT_17 AX – Bool Bit 17
BIT_18 AX – Bool Bit 18
BIT_19 AX – Bool Bit 19
BIT_20 AX – Bool Bit 20
BIT_21 AX – Bool Bit 21
BIT_22 AX – Bool Bit 22
BIT_23 AX – Bool Bit 23
BIT_24 AX – Bool Bit 24
BIT_25 AX – Bool Bit 25
BIT_26 AX – Bool Bit 26
BIT_27 AX – Bool Bit 27
BIT_28 AX – Bool Bit 28
BIT_29 AX – Bool Bit 29
BIT_30 AX – Bool Bit 30
BIT_31 AX – Bool Bit 31
BIT_32 AX – Bool Bit 32
BIT_33 AX – Bool Bit 33
BIT_34 AX – Bool Bit 34
BIT_35 AX – Bool Bit 35
BIT_36 AX – Bool Bit 36
BIT_37 AX – Bool Bit 37
BIT_38 AX – Bool Bit 38
BIT_39 AX – Bool Bit 39
BIT_40 AX – Bool Bit 40
BIT_41 AX – Bool Bit 41
BIT_42 AX – Bool Bit 42
BIT_43 AX – Bool Bit 43
BIT_44 AX – Bool Bit 44
BIT_45 AX – Bool Bit 45
BIT_46 AX – Bool Bit 46
BIT_47 AX – Bool Bit 47
BIT_48 AX – Bool Bit 48
BIT_49 AX – Bool Bit 49
BIT_50 AX – Bool Bit 50
BIT_51 AX – Bool Bit 51
BIT_52 AX – Bool Bit 52
BIT_53 AX – Bool Bit 53
BIT_54 AX – Bool Bit 54
BIT_55 AX – Bool Bit 55
BIT_56 AX – Bool Bit 56
BIT_57 AX – Bool Bit 57
BIT_58 AX – Bool Bit 58
BIT_59 AX – Bool Bit 59
BIT_60 AX – Bool Bit 60
BIT_61 AX – Bool Bit 61
BIT_62 AX – Bool Bit 62
BIT_63 AX – Bool Bit 63

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **SPLIT_AL_INTO_AX** zerlegt einen als LWORD (64 Bit) kodierten Wert in 64 einzelne BOOL-Signale. Der Eingangswert wird über einen **AL-Adapter** (Analog-Longword) bereitgestellt, die Ausgabe erfolgt über 64 separate **AX-Adapter** (Analog-Bit). Jeder Ausgangsadapter liefert den Status eines Bits zusammen mit einem zugehörigen Ereignis. Der Baustein eignet sich besonders für die Aufbereitung digitaler Signale aus einem kompakten Datenwort, z. B. zur parallelen Ansteuerung von 64 diskreten Ausgängen.

Schnittstellenstruktur

Ereignis-Eingänge

- **IN.E1** (im IN-Adapter): Startet die Verarbeitung, sobald ein neuer LWORD-Wert am Adapter anliegt.

Ereignis-Ausgänge

Jeder AX-Adapter stellt einen Ereignisausgang bereit:

- **BIT_00.E1** .. **BIT_63.E1** (in den entsprechenden Adaptern): Wird ausgelöst, sobald das zugehörige Bit im internen Flipflop aktualisiert wurde.

Daten-Eingänge

- **IN.D1** (LWORD): Der zu zerlegende 64-Bit-Datensatz.

Daten-Ausgänge

- **BIT_00.D1** .. **BIT_63.D1** (BOOL): Der Zustand des jeweiligen Bits (0 oder 1).

Adapter

- **IN** (Socket, Typ: `adapter::types::unidirectional::AL`): Liefert den LWORD-Wert und das Startereignis.
- **BIT_00** .. **BIT_63** (Plugs, Typ: `adapter::types::unidirectional::AX`): Geben jeweils ein BOOL-Signal (Bitwert) und ein zugehöriges Ereignis aus.

Funktionsweise

1. Ein eingehender LWORD-Wert über den AL-Adapter **IN** löst über das Ereignis **E1** die Verarbeitung aus.
2. Der interne Baustein **SPLIT_LWORD_INTO_BOOLS** extrahiert alle 64 Bits aus dem LWORD.
3. Jedes Bit wird auf den Dateneingang **D** eines zugehörigen **E_D_FF** (Edge-Triggered D-Flipflop) gelegt.
4. Gleichzeitig erhalten alle 64 Flipflops über das Ereignis **CNF** des Splitters den Taktimpuls **CLK**. Damit werden alle Bits gleichzeitig in die Flipflops übernommen.
5. Die Flipflops geben den gespeicherten Wert an ihrem Ausgang **Q** aus, welcher über den Datenausgang **D1** des jeweiligen AX-Adapters bereitgestellt wird.
6. Jedes Flipflop erzeugt am Ausgang **EO** ein Ereignis, das an den Ereigniseingang **E1** des entsprechenden AX-Adapters weitergeleitet wird.

Technische Besonderheiten

- **Synchronisation**: Sämtliche Bits werden durch ein gemeinsames Ereignis (CNF von SPLIT_LWORD_INTO_BOOLS) zeitgleich in die Flipflops übernommen. So entsteht ein konsistenter Ausgangszustand ohne Wettlaufsituationen.
- **Speicherung**: Die Verwendung von **E_D_FF** bewirkt, dass der Ausgangszustand erhalten bleibt, bis ein neuer LWORD-Wert verarbeitet wird. Der Baustein arbeitet somit als „Sample & Hold“ für alle 64 Bits.
- **Große Anzahl von Ausgängen**: Der FB bietet 64 unabhängige BOOL-Ausgänge, die in der 4diac-IDE auf viele Geräteadapter abgebildet werden können.
- **Adapter-basierte Schnittstelle**: Ein- und Ausgänge sind als unidirektionale Adapter realisiert, was die modulare Kapselung und Wiederverwendung erleichtert.

Zustandsübersicht

Der FB besitzt keinen eigenen expliziten Zustandsautomaten. Der innere Zustand wird durch die 64 **E_D_FF** repräsentiert:

- **Initial**: Alle Flipflops stehen auf FALSE (0).

- **Nach einer Verarbeitung**: Jedes Flipflop speichert den aktuellen Bitwert des zuletzt empfangenen LWORD. Die Zustandsänderung erfolgt nur bei einem neuen Ereignis auf **IN.E1**.

Anwendungsszenarien

- **Parallele Ausgabe von digitalen Signalen** aus einem seriellen oder paketierten Datenwort, z. B. aus einem Bussystem oder einer Kommunikationsschnittstelle.
- **Maskierte Steuerung**: Kombination mit logischen Bausteinen, um einzelne Bits eines Statusworts gezielt auszuwerten.
- **Test- und Diagnose-Interfaces**: Aufbereitung eines 64-Bit-Fehlerworts für die Anzeige auf separaten Leuchten oder in Visualisierungen.
- **Schrittstelle zu diskreten Aktoren**: Jeder AX-Adapter kann direkt mit einer digitalen Ausgangsbaugruppe verbunden werden.

Vergleich mit ähnlichen Bausteinen

- **SPLIT_LWORD_INTO_BOOLS** (rein funktionaler Splitter ohne Speicher): Liefert die Bits nur während der Verarbeitung, hält den Wert nicht. **SPLIT_AL_INTO_AX** ergänzt die Speicherfunktion und stellt einen stabilen Ausgang bis zur nächsten Aktualisierung bereit.
- **Typkonverter wie DWORD_TO_BOOL_ARRAY**: Arbeiten meist auf Arrays und nicht auf Adapterniveau; der vorliegende Baustein ist speziell auf die 4diac-Adapter-Schnittstelle zugeschnitten.
- **Flipflop-basierte Lösungen mit je einem eigenen F_TRIG o. Ä.:** Der FB vereinfacht die Konfiguration, da alle 64 Bits in einem einzigen Baustein zusammengefasst sind.

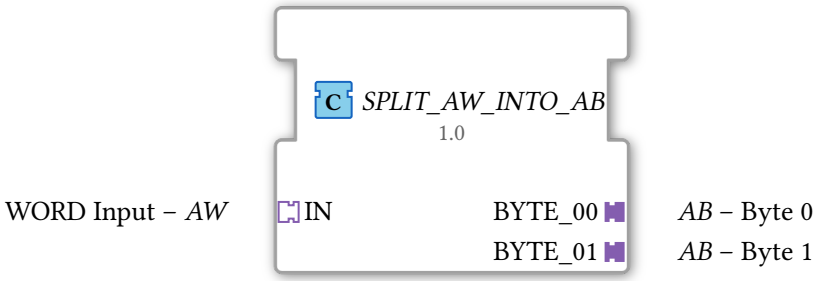
Fazit

SPLIT_AL_INTO_AX ist ein leistungsfähiger, adapterbasierter Funktionsblock zur Aufteilung eines LWORD-Datensatzes in 64 diskrete BOOL-Signale. Durch die integrierten Flipflops bleibt der Ausgangszustand bis zur nächsten Aktualisierung erhalten. Die große Anzahl von Ausgängen und die synchrone Verarbeitung machen ihn ideal für Anwendungen, die eine parallele, zeitlich konsistente Signalerzeugung erfordern. Der Einsatz von unidirektionalen Adaptern sorgt für eine klare Schnittstellendefinition und erleichtert die Integration in bestehende 4diac-Projekte.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SPLIT_AW_INTO_AB



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **SPLIT_AW_INTO_AB** dient zur Aufteilung eines Wort-Datenpakets (Typ AW, 16 Bit) in zwei einzelne Byte-Datenpakete (Typ AB, je 8 Bit). Die Eingangsdaten werden über einen Socket empfangen und über zwei separate Plugs ausgegeben. Der Baustein wird typischerweise in Umgebungen eingesetzt, die auf Adapter-basierte Kommunikation setzen, z. B. in der Automatisierungstechnik beim Zerlegen von Wort-basierten Protokollen.

Schnittstellenstruktur

Ereignis-Eingänge

Der FB besitzt keine eigenen Ereignis-Eingänge. Die Ereignissteuerung erfolgt indirekt über den Adapter **IN**.

Ereignis-Ausgänge

Der FB besitzt keine eigenen Ereignis-Ausgänge. Ereignisse werden über die Ausgangs-Adapter **BYTE_00** und **BYTE_01** nach außen getragen.

Daten-Eingänge

Der FB besitzt keine eigenen Daten-Eingänge. Die Dateneingabe erfolgt indirekt über den Adapter **IN**.

Daten-Ausgänge

Der FB besitzt keine eigenen Daten-Ausgänge. Die Datenausgabe erfolgt indirekt über die Adapter **BYTE_00** und **BYTE_01**.

Adapter

Adapter	Richtung	Typ	Beschreibung
IN	Socket	`adapter::types::unidirectional::AW`	Empfängt ein 16-Bit Wort (WORD). Der Adapter stellt die Daten und ein zugehöriges Ereignis zur Verfügung.
BYTE_00	Plug	`adapter::types::unidirectional::AB`	Gibt das niederwertige Byte (Byte 0) aus.
BYTE_01	Plug	`adapter::types::unidirectional::AB`	Gibt das höherwertige Byte (Byte 1) aus.

Funktionsweise

1. Ein eingehendes Ereignis auf dem Adapter **IN** (E1) triggert den internen Funktionsblock **SPLIT_WORD_INTO_BYTES**.
2. Dieser teilt das empfangene WORD (16 Bit) in zwei Bytes auf: **BYTE_00** (niederwertig) und **BYTE_01** (höherwertig).
3. Das Abschlussereignis (CNF) von **SPLIT_WORD_INTO_BYTES** wird an die Takt-Eingänge zweier **E_D_FF_ANY** Flip-Flops weitergeleitet.
4. Jeder Flip-Flop speichert das zugehörige Byte und gibt es bei einem neuen Taktimpuls über seinen Ausgang **Q** aus.
5. Die Flip-Flops erzeugen jeweils ein Ereignis auf **EO**, das an die Ausgangs-Adapter **BYTE_00** und **BYTE_01** gesendet wird. So werden die aufgeteilten Bytes zusammen mit einem Ereignis nach außen bereitgestellt.

Technische Besonderheiten

- **Adapter-basierte Schnittstelle:** Der FB enthält keine direkten Event- oder Dateneingänge/-ausgänge. Die gesamte Kommunikation erfolgt über Adapter (Socket für Eingang, Plugs für Ausgänge). Er eignet sich daher besonders für Anwendungen, die auf das adapterbasierte 4diac-Typensystem setzen.
- **Datenzwischenspeicherung:** Die beiden **E_D_FF_ANY** Flip-Flops halten die aufgeteilten Bytes stabil, bis neue Daten anliegen. Dadurch werden inkonsistente Zustände während der Aufteilung vermieden.
- **Ereignissynchronisation:** Die Ausgabe der Bytes erfolgt zeitgleich, da das gleiche CNF-Ereignis beide Flip-Flops taktet.

Zustandsübersicht

Der FB besitzt keine explizite Zustandsmaschine. Er verarbeitet Impulse ereignisgesteuert und gibt bei jedem Ereignis auf **IN** die aufgeteilten Bytes aus. Die Flip-Flops haben die typischen zwei Zustände: gespeicherter Wert (0 oder 1 pro Bit, abhängig vom Byte).

Anwendungsszenarien

- **Byte-orientierte Kommunikation:** Wenn ein Sensor oder Aktor Daten im WORD-Format sendet, aber nachgelagerte Bausteine einzelne Bytes erwarten.
- **Protokoll-Aufbereitung:** Zerlegen von Telegrammen, die aus einem 16-Bit Wert bestehen, in getrennte Byte-Adapter für weitere Verarbeitungsschritte.
- **Modbus/IO-Link Integration:** Anpassung von 16-Bit Registerwerten an 8-Bit Schnittstellen.

Vergleich mit ähnlichen Bausteinen

- **SPLIT_WORD_INTO_BYTES:** Ein reiner Daten-Splitter ohne Adapter-Kapselung. Er arbeitet auf Datenebene und benötigt eigene Event- und Datenschnittstellen.
- **MUX/ DEMUX Bausteine:** Multiplexer oder Demultiplexer für mehrere Werte, jedoch meist mit anderen Datentypen und ohne Adapter-Integration.
- **Adapter-Wrapper wie SPLIT_AW_INTO_AB:** Speziell für die adapterbasierte 4diac-Architektur entwickelt. Sie kapseln die Aufteilungslogik hinter einheitlichen Adapter-Schnittstellen.

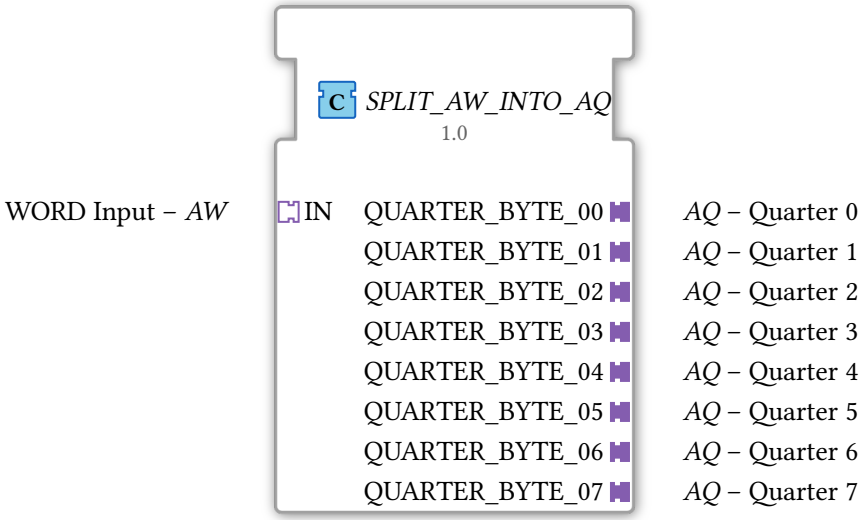
Fazit

SPLIT_AW_INTO_AB ist ein spezialisierter Adapter-Wrapper, der ein einkommendes WORD sauber in zwei getrennte BYTE-Adapter aufteilt. Die Verwendung von Flip-Flops gewährleistet einen stabilen Datenfluss und vermeidet zeitliche Inkonsistenzen. Der Baustein ist ideal für Systeme, die eine adapterbasierte Kommunikation nach IEC 61499-2 verwenden und eine einfache Transformation zwischen 16-Bit und 8-Bit Daten benötigen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SPLIT_AW_INTO_AQ



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **SPLIT_AW_INTO_AQ** teilt die acht Quarter-Adapter (AQ) eines Word-Adapters (AW) auf. Das eingehende Word (16 Bit) wird in acht Quarter-Einheiten (je 2 Bit) zerlegt und über die entsprechenden AQ-Plugs ausgegeben.

Schnittstellenstruktur

Ereignis-Eingänge

- `IN.E1` (über Socket IN) – Lösen die Aufteilung eines neuen Word-Wertes aus.

Ereignis-Ausgänge

- `QUARTER_BYTE_00.E1` bis `QUARTER_BYTE_07.E1` – Signalisiert die Bereitstellung des jeweiligen Quarter-Ergebnisses.

Daten-Eingänge

- `IN.D1` – Empfängt das eingehende Word (16 Bit, Typ passend zum AW-Adapter).

Daten-Ausgänge

- `QUARTER_BYTE_00.D1` bis `QUARTER_BYTE_07.D1` – Liefern die acht aufgeteilten Quarter-Werte (jeweils 2 Bit).

Adapter

- **Socket:** `IN` – Adapter vom Typ `adapter::types::unidirectional::AW`
- **Plugs:** `QUARTER_BYTE_00` ... `QUARTER_BYTE_07` – Adapter vom Typ `adapter::types::unidirectional::AQ`

Funktionsweise

1. Ein Ereignis am Eingang `IN.E1` aktiviert den internen Baustein `SPLIT_WORD_INTO_QUARTERS`.
2. `SPLIT_WORD_INTO_QUARTERS` zerlegt das über `IN.D1` anliegende 16-Bit-Wort in acht separate 2-Bit-Quarter-Werte.
3. Die aufgeteilten Werte werden parallel in acht `E_D_FF_ANY`-Flip-Flops zwischengespeichert (`D`-Eingänge).
4. Nach Abschluss der Verarbeitung (CNF-Ereignis von `SPLIT_WORD_INTO_QUARTERS`) erhalten alle Flip-Flops gleichzeitig einen Taktimpuls (`CLK`).
5. Die Ausgänge `Q` der Flip-Flops werden auf die entsprechenden `QUARTER_BYTE_xx.D1`-Ausgänge gelegt.
6. Gleichzeitig wird jeweils das Ereignis `QUARTER_BYTE_xx.E1` ausgelöst, um die Datenweitergabe an nachgeschaltete Bausteine zu signalisieren.

Technische Besonderheiten

- **Synchronisation durch Flip-Flops:** Alle acht Quarter-Werte werden taktsynchron aus dem internen Splitter übernommen.
- **Keine persistente Zustandshaltung:** Der Baustein puffert die Quarter-Werte nur bis zum nächsten Verarbeitungsdurchlauf.
- **Einheitliche Schnittstellen:** Die Adapter folgen dem unidirektionalen AW/AQ-Protokoll und ermöglichen eine einfache Kopplung in der 4diac-IDE.

Zustandsübersicht

Der Baustein besitzt keinen eigenen Zustandsautomaten. Die internen `E_D_FF_ANY`-Flip-Flops können sich in zwei Zuständen befinden:

- **Speichert vorherigen Wert** – Bis ein neues Ereignis eintrifft.
- **Aktualisierter Wert** – Nach einem Taktimpuls durch das CNF-Ereignis.

Anwendungsszenarien

- **Datenaufbereitung:** Zerlegen eines 16-Bit-Wortes aus einem Kommunikations-Adapter in acht 2-Bit-Quarter-Signale, z.B. für parallele Verarbeitung in Steuerungslogiken.
- **Multiplexing/Demultiplexing:** Trennen von Wort-Datenströmen in separate Quarter-Kanäle.
- **Erweiterung von Busstrukturen:** Einfaches Aufteilen von Adress- oder Steuerwörtern auf mehrere nachfolgende Bausteine.

Vergleich mit ähnlichen Bausteinen

- **SPLIT_BYTE_INTO_NIBBLES** – Teilt ein Byte in zwei 4-Bit-Nibbles.
- **SPLIT_WORD_INTO_BYTES** – Zerlegt ein 16-Bit-Wort in zwei 8-Bit-Bytes.
- **SPLIT_AW_INTO_AQ** ist speziell für die Aufteilung eines AW-Adapters (Word) in acht AQ-Adapter (Quarter) ausgelegt – die Schnittstellen sind direkt auf die Adapter-Typen abgestimmt und erfordern keine manuelle Konfiguration.

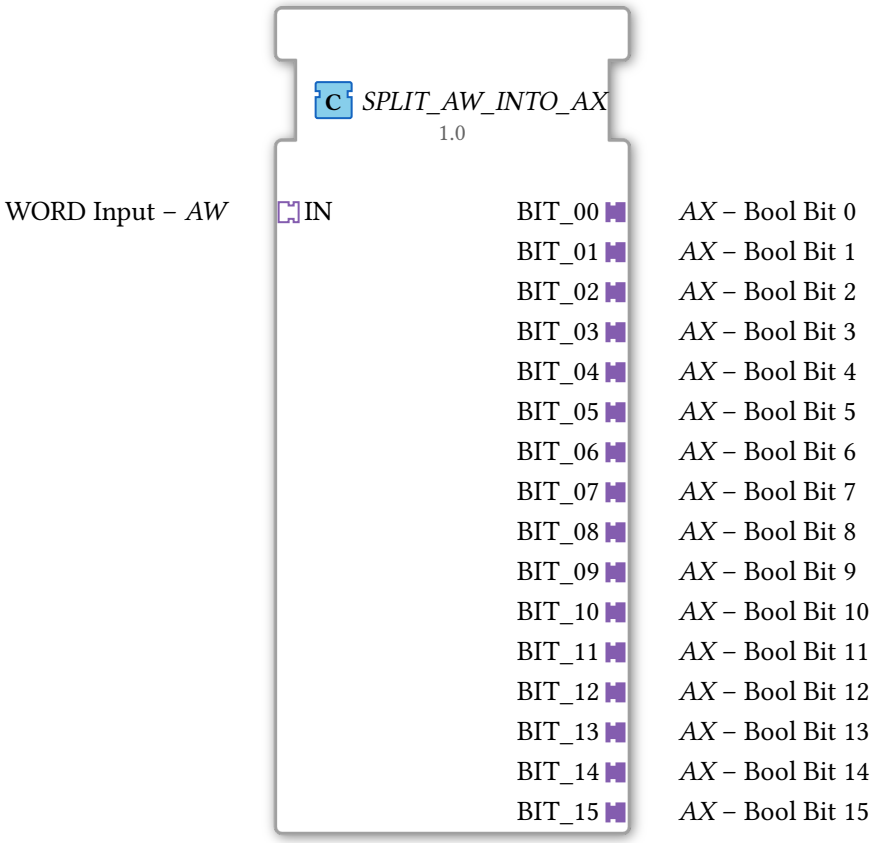
Fazit

Der Baustein **SPLIT_AW_INTO_AQ** bietet eine kompakte und zuverlässige Möglichkeit, ein einkommendes Word in acht Quarter-Signale aufzuteilen. Durch die integrierte Taktsynchronisation und die standardisierten Adapter-Schnittstellen eignet er sich hervorragend für modulare IEC 61499-Anwendungen, die eine gleichzeitige Datenaufbereitung für mehrere Teilnehmer erfordern.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SPLIT_AW_INTO_AX



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `SPLIT_AW_INTO_AX` dient dazu, ein 16-Bit Wort (Typ `AW`) in 16 einzelne Binärsignale (Typ `AX`) aufzuteilen. Jeder der 16 Ausgänge repräsentiert ein Bit des eingehenden Wortes und wird als eigenständiger Adapter mit Ereignis- und Datenleitung bereitgestellt. Die Aufteilung erfolgt synchron bei Eintreffen eines Ereignisses am Eingangsadapter.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine eigenständigen Ereignis-Eingänge auf der Fassade. Das initiale Ereignis wird über den Socket-Adapter `IN` (siehe Adapter) empfangen.

Ereignis-Ausgänge

Der Baustein besitzt keine eigenständigen Ereignis-Ausgänge auf der Fassade. Die Ausgangsereignisse werden über die Plug-Adapter `BIT_00` ... `BIT_15` (siehe Adapter) abgegeben.

Daten-Eingänge

Der Baustein besitzt keine eigenständigen Daten-Eingänge auf der Fassade. Der zu verarbeitende 16-Bit Wert wird über den Socket-Adapter `IN` (siehe Adapter) als Datenwert `D1` (Typ `WORD`) bereitgestellt.

Daten-Ausgänge

Der Baustein besitzt keine eigenständigen Daten-Ausgänge auf der Fassade. Die 16 extrahierten Binärwerte werden über die Plug-Adapter `BIT_00` ... `BIT_15` als Datenwert `D1` (Typ `BOOL`) ausgegeben.

Adapter

Richtung	Name	Typ	Kommentar

Socket (Eingang)	`IN`	`adapter::types::unidirectional::AW`	16-Bit Wort als Eingang, Ereignis-Eingang `E1`, Daten-Eingang `D1`
Plug (Ausgang)	`BIT_00` … `BIT_15`	`adapter::types::unidirectional::AX`	Jeweils ein Bit des Wortes, Ereignis-Ausgang `E1`, Daten-Ausgang `D1`

Funktionsweise

1. Ein externes Ereignis am Socket `IN.E1` löst die Verarbeitung aus.
2. Der aktuelle Wert von `IN.D1` (Typ `WORD`, 16 Bit) wird ausgelesen.
3. Der intern eingebettete Baustein `SPLIT_WORD_INTO_BOOLS` zerlegt das Wort in 16 einzelne Boolesche Werte (`BIT_00` … `BIT_15`).
4. Diese 16 Werte werden taktgleich an jeweils ein `E_D_FF` (D-Flipflop) übergeben. Die Flipflops übernehmen die Daten mit jedem Takt (hier dem Ereignis `CNF` von `SPLIT_WORD_INTO_BOOLS`).
5. Jedes Flipflop gibt seinen gespeicherten Wert über den zugehörigen Plug-Adapter `BIT_xx.D1` sowie ein Ereignis über `BIT_xx.E1` aus.

Dadurch wird der gesamte Wortwert in einem Zyklus auf die 16 Ausgangsadapter verteilt und dort bis zum nächsten Ereignis gehalten.

Technische Besonderheiten

- **Ereignissynchrone Aufteilung:** Die gesamte Aufteilung erfolgt innerhalb eines einzigen Ereigniszyklus – alle 16 Ausgänge werden gleichzeitig aktualisiert.
- **Speicherung:** Jeder Bitwert wird durch ein eigenes `E_D_FF` gehalten, sodass die Ausgänge auch ohne ständig wiederholte Ereignisse stabil bleiben.
- **Adapter-basierte Ein-/Ausgabe:** Der Baustein kommuniziert ausschließlich über Adapter – er kann daher nahtlos in 4diac-Systeme eingebunden werden, die unidirektionale Adapter für Wort- oder Boolsignale nutzen.
- **Keine eigenen Ereignis-/Dateneingänge:** Die gesamte Schnittstelle wird über die Adapter abgebildet; eine direkte Verkabelung auf der Fassade entfällt.

Zustandsübersicht

Der Baustein besitzt keinen eigenen Zustandsautomaten auf oberster Ebene. Der interne Zustand wird durch die 16 D-Flipflops (`E_D_FF_00` … `E_D_FF_15`) bestimmt:

- Jedes Flipflop kann sich in einem der beiden Zustände `Q = 0` oder `Q = 1` befinden.
- Der Zustand wird nur bei einem Takt (d.h. bei jedem neu eintreffenden Ereignis am Eingang) aktualisiert.
- Im Ruhezustand zwischen Ereignissen bleibt der jeweils zuletzt gespeicherte Wert erhalten.

Anwendungsszenarien

- **Steuerungsaufgaben:** Ein übergeordneter Regler oder eine SPS sendet ein 16-Bit Wort (z.B. als Steuerwort) – der Baustein wandelt es in 16 einzelne binäre Ansteuersignale um (z.B. für Relais, Ventile oder Anzeigeelemente).
- **Protokollkonvertierung:** Eingangssignale aus einem Bussystem liegen als Wort vor und müssen auf diskrete Ausgänge verteilt werden.
- **Test und Simulation:** Darstellung eines Wortes als 16 Boolesche Kanäle zur Visualisierung oder Fehlersuche.

Vergleich mit ähnlichen Bausteinen

- **`SPLIT_WORD_INTO_BOOLS`** – Dieser Baustein teilt ein Wort ebenfalls in Boolesche Werte auf, jedoch ohne Adapter und ohne ereignisgesteuerte Ausgabe. Er dient als interne Komponente des vorliegenden Bausteins.
- **`SPLIT_AW_INTO_AX`** – Erweitert die reine Datenteilung um Adapter und Ereignisausgabe, sodass die einzelnen Bits als vollwertige AX-Schnittstellen (mit eigenem Ereignis und Daten) zur Verfügung stehen. Dadurch ist eine direkte Verschaltung mit anderen 4diac-Bausteinen, die AX-Adapter erwarten, möglich.

- **Alternative Eigenentwicklung:** Theoretisch könnte man 16 separate `SLICE`-Bausteine verwenden, um Bits aus einem Wort zu extrahieren – dies wäre jedoch aufwändiger und würde nicht die synchrone Ereignisausgabe bieten.

Fazit

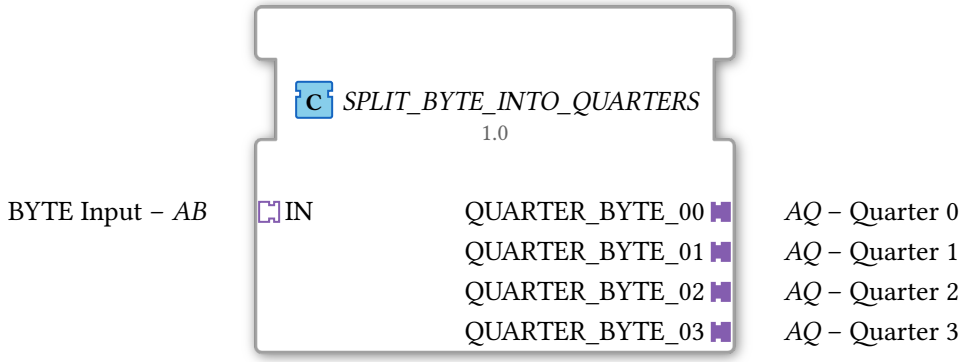
`SPLIT_AW_INTO_AX` ist ein kompakter, aber leistungsfähiger Konverter, der ein 16-Bit Wort zuverlässig und ereignissynchron auf 16 einzelne Binärausgänge aufteilt. Durch die Verwendung von Adaptern und Flipflops ist er besonders gut geeignet für den Einsatz in modularen IEC 61499-Anwendungen, bei denen eine saubere Trennung von Wort- und Bitschnittstellen gefordert wird. Er vereinfacht die Schnittstellenanpassung und erhöht die Lesbarkeit und Wartbarkeit des Applikationsnetzwerks.

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SPLIT_BYTE_INTO_QUARTERS

Bild nicht verfügbar



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `SPLIT_BYTE_INTO_QUARTERS` dient der Aufteilung eines eingehenden Byte-Datenwerts in vier separate Quarter-Ausgänge. Er realisiert eine logische Zerlegung eines Bytes (8 Bit) in vier 2-Bit-Gruppen, die über unidirektionale Adapter nach außen geführt werden. Die interne Implementierung nutzt eine Kombination aus Flipflops und einer rekursiven Instanz des gleichen Bausteins, um die Aufteilung synchron und datengetrieben zu steuern.

Schnittstellenstruktur

Ereignis-Eingänge

Nicht vorhanden. Der Baustein besitzt keine eigenständigen Ereignis-Eingänge; die Steuerung erfolgt ausschließlich über die Adapter-Schnittstellen.

Ereignis-Ausgänge

Nicht vorhanden.

Daten-Eingänge

Nicht vorhanden.

Daten-Ausgänge

Nicht vorhanden.

Adapter

Richtung	Name	Typ	Beschreibung

Socket (Eingang)	`IN`	`adapter::types::unidirectional::AB`	Byte-Eingang, der die aufzuteilenden Daten liefert.
Plug (Ausgang)	`QUARTER_BYTE_00`	`adapter::types::unidirectional::AQ`	Ausgang für das erste Quarter (niederwertigste 2 Bit).
Plug (Ausgang)	`QUARTER_BYTE_01`	`adapter::types::unidirectional::AQ`	Ausgang für das zweite Quarter (Bit 2-3).
Plug (Ausgang)	`QUARTER_BYTE_02`	`adapter::types::unidirectional::AQ`	Ausgang für das dritte Quarter (Bit 4-5).
Plug (Ausgang)	`QUARTER_BYTE_03`	`adapter::types::unidirectional::AQ`	Ausgang für das vierte Quarter (höchstwertigste 2 Bit).

Funktionsweise

Der Baustein arbeitet als **Composite-FB** mit einem inneren Netzwerk. Sobald der Socket `IN` ein Ereignis (über `E1`) und die zugehörigen Daten empfängt, werden diese an eine interne Instanz des gleichen Bausteins (`SPLIT_BYTE_INTO_QUARTERS`) weitergeleitet. Deren Ausgänge (ebenfalls vier Quarter-Adapter) werden in vier parallel geschalteten **E_D_FF_ANY-Flipflops** zwischengespeichert. Die Flipflops übernehmen die Daten mit dem durch die interne `CNF`-Eventkette getakteten Takt und geben sie synchron an die äußeren Plug-Adapter (`QUARTER_BYTE_00` … `QUARTER_BYTE_03`) weiter. Die genaue Abbildung der Bits auf die Quarters erfolgt implizit über die interne Verkabelung; es wird angenommen, dass das Byte in vier aufeinanderfolgende 2-Bit-Gruppen (von LSB nach MSB) unterteilt wird.

Technische Besonderheiten

- **Rekursive Typverwendung**: Die interne Netztopologie enthält eine weitere Instanz des Bausteins `SPLIT_BYTE_INTO_QUARTERS`, wodurch eine mehrstufige oder iterative Zerlegung ermöglicht wird. Die konkrete Tiefe bleibt jedoch durch die feste Verdrahtung auf eine Stufe beschränkt.
- **Adapter basiertes Design**: Statt herkömmlicher Datenports werden ausschließlich unidirektionale Adapter eingesetzt, die sowohl Ereignis- als auch Datenkanäle kapseln. Dies fördert lose Kopplung und Wiederverwendbarkeit.
- **Synchronisation durch Flipflops**: Die vier Ausgangs-Flipflops gewährleisten, dass alle Quarter gleichzeitig mit einem gemeinsamen Takt aktualisiert werden, sobald die interne Aufteilung abgeschlossen ist.

Zustandsübersicht

Der Baustein besitzt **keinen expliziten Zustandsautomaten** (ECC). Seine Funktionsweise ergibt sich allein aus der Daten- und Ereignisverdrahtung der inneren Komponenten. Das Systemverhalten ist daher rein kombinatorisch mit einer Taktflanke der Flipflops.

Anwendungsszenarien

- **Parallelisierung von Byte-Daten**: Zerlegung eines empfangenen Bytes in vier unabhängige 2-Bit-Kanäle, z. B. zur Ansteuerung von vier parallelen Ausgabegeräten oder zur verteilten Verarbeitung.
- **Serielle- zu Parallel-Wandlung**: Umwandlung eines seriellen Byte-Datenstroms in vier parallele Quarter-Signale.
- **Protokoll-Aufbereitung**: Extrahieren von Teilinformationen aus einem Byte, wenn jedes Quarter eine separate Bedeutung trägt (z. B. Status-Flags, Konfigurationswerte).

Vergleich mit ähnlichen Bausteinen

- **SPLIT_WORD_INTO_HALF_WORDS**: Analoger Baustein zur Aufteilung eines 16-Bit-Worts in zwei 8-Bit-Hälften, arbeitet meist mit direkten Datenports.
- **EXTRACT_BITS_4TO1**: Standard-Splitting-Bausteine arbeiten oft bitweise oder mit wählbaren Indexbereichen. Der vorliegende Baustein ist speziell auf die feste Quarter-Aufteilung eines Bytes optimiert und nutzt Adapter anstelle von einfachen Daten-Ein-/Ausgängen.
- **MUX/DEMUX-Bausteine**: Multiplexer und Demultiplexer bieten generische Aufteilung, benötigen jedoch zusätzliche Steuerleitungen. Der `SPLIT_BYTE_INTO_QUARTERS` ist spezialisierter und einfacher konfigurierbar.

Fazit

Der Funktionsblock `SPLIT_BYTE_INTO_QUARTERS` stellt eine kompakte, adapterbasierte Lösung zur Zerlegung eines Bytes in vier Quarter-Ausgänge dar. Durch die Kombination aus rekursiver Struktur und Flipflop-Synchronisation wird eine zuverlässige, parallele Datenbereitstellung erreicht. Er eignet sich für Anwendungen, die eine feste Aufteilung eines 8-Bit-Werts in vier 2-Bit-Einheiten erfordern, und bietet durch die einheitliche Adapter-Schnittstelle eine gute Integrationsfähigkeit in modulare 4diac-Projekte.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

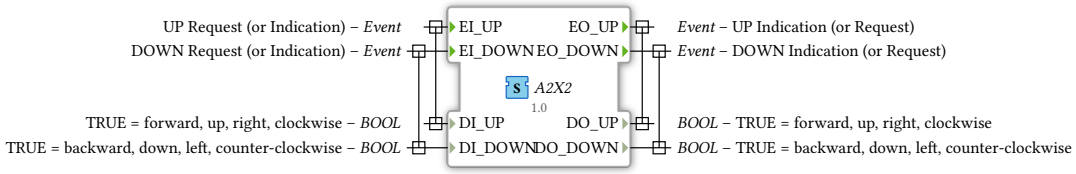
2.1.11.17 types

Bidirectional Adapters

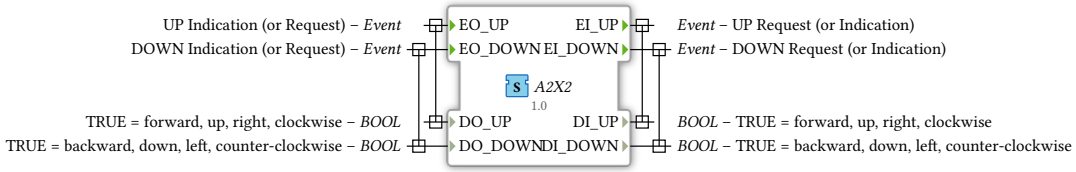
BOOL

A2X2

Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



bidirektionales Adapter-Interface für 2 Ereignisse und 2 Bools

Interface

Ereignis-Eingänge

Name	Kommentar	With
EI_UP	UP Anfrage (oder Indikation)	DI_UP
EI_DOWN	DOWN Anfrage (oder Indikation)	DI_DOWN

Ereignis-Ausgänge

Name	Kommentar	With
EO_UP	UP Indikation (oder Anfrage)	DO_UP
EO_DOWN	DOWN Indikation (oder Anfrage)	DO_DOWN

Eingangsvariablen

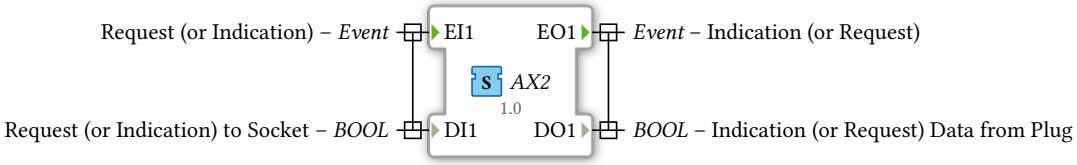
Name	Typ	Kommentar
DI_UP	BOOL	TRUE = vorwärts, hoch, rechts, im Uhrzeigersinn
DI_DOWN	BOOL	TRUE = rückwärts, runter, links, gegen den Uhrzeigersinn

Ausgangsvariablen

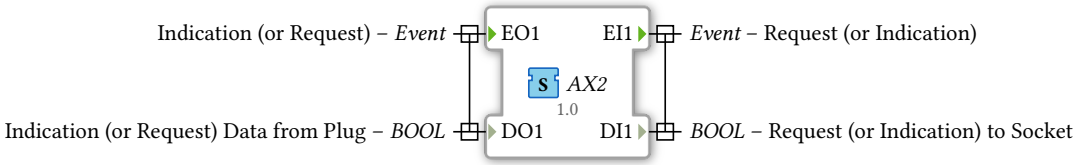
Name	Typ	Kommentar
DO_UP	BOOL	TRUE = vorwärts, hoch, rechts, im Uhrzeigersinn
DO_DOWN	BOOL	TRUE = rückwärts, runter, links, gegen den Uhrzeigersinn

AX2

Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



bidirectional Adapter Interface for 1 Event and 1 Bool

Interface

Event Inputs

Name	Comment	With
EI1	Request (or Indication)	DI1

Event Outputs

Name	Comment	With
EO1	Indication (or Request)	DO1

Input Vars

Name	Type	Comment
DI1	BOOL	Request (or Indication) to Socket

Output Vars

Name	Type	Comment
DO1	BOOL	Indication (or Request) Data from Plug

Bidirectional BOOL Adapter

🎧 Podcast

- George Boole: Wie ein Autodidakt mit 0 und 1 das Fundament unserer digitalen Welt legte

🛠️ Zugehörige Übungen

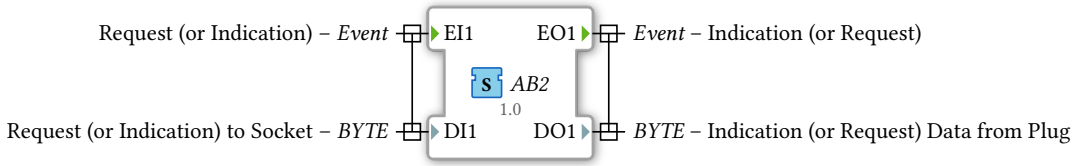
- Uebung_001_AX_b
- Uebung_002a2_AX
- Uebung_002a4

- Uebung_002a5b
- Uebung_002b3
- Uebung_004b_AX
- Uebung_006a3
- LinksRechts_T_FF_Event
- Uebung_035a2
- Uebung_035a3
- SoftkeyOverride_TO_QX_BG
- Uebung_055
- Uebung_056
- Bool8_TO_logiBUS_QX8
- Uebung_094a_AX
- Uebung_160

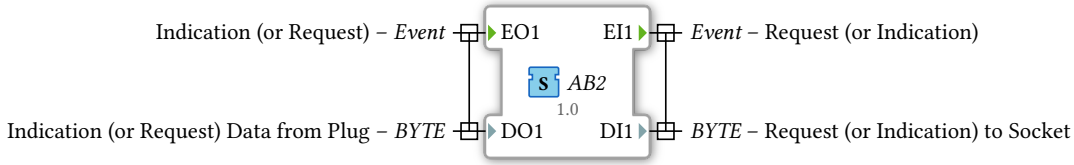
BYTE

AB2

Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



bidirectional Adapter Interface for 1 Event and 1 Byte

Interface

Event Inputs

Name	Comment	With
EI1	Request (or Indication)	DI1

Event Outputs

Name	Comment	With
EO1	Indication (or Request)	DO1

Input Vars

Name	Type	Comment
DI1	BYTE	Request (or Indication) to Socket

Output Vars

Name	Type	Comment
DO1	BYTE	Indication (or Request) Data from Plug

Bidirectional BYTE Adapter

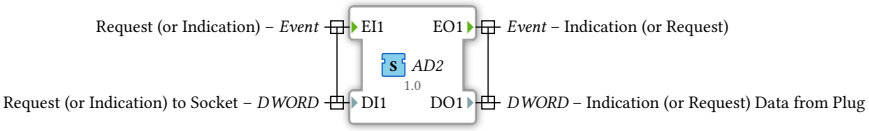
🛠️ Zugehörige Übungen

- SIN_TO_PCAN_Callback_Byte
- Uebung_128b
- Uebung_12x_sub

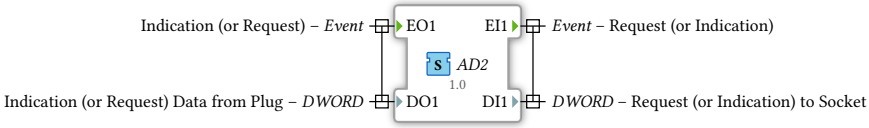
DWORD

AD2

Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



bidirectional Adapter Interface for 1 Event and 1 Dword

Interface

Event Inputs

Name	Comment	With
EI1	Request (or Indication)	DI1

Event Outputs

Name	Comment	With
EO1	Indication (or Request)	DO1

Input Vars

Name	Type	Comment
DI1	DWORD	Request (or Indication) to Socket

Output Vars

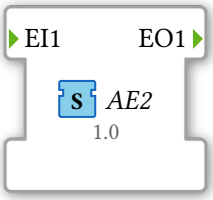
Name	Type	Comment
DO1	DWORD	Indication (or Request) Data from Plug

EVENT

AE2

Plug (Adapter-Ausgang)

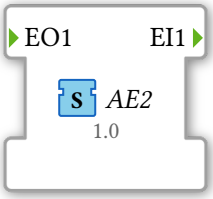
Request (or Indication) – *Event*



Event – Indication (or Request)

Socket (Adapter-Eingang)

Indication (or Request) – *Event*



Event – Request (or Indication)

bidirectional Adapter Interface for 1 Event

Interface

Event Inputs

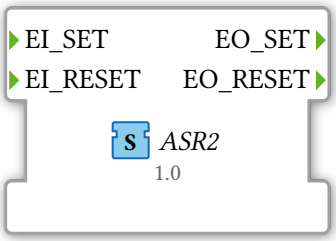
Name	Comment	With
EI1	Request (or Indication)	

Event Outputs

Name	Comment	With
EO1	Indication (or Request)	

ASR2
Plug (Adapter-Ausgang)

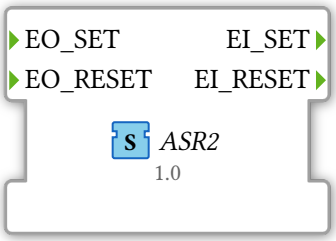
Set / Switch on – *Event*
Reset / Switch off – *Event*



Event – Set / Switch on
Event – Reset / Switch off

Socket (Adapter-Eingang)

Set / Switch on – *Event*
Reset / Switch off – *Event*



Event – Set / Switch on
Event – Reset / Switch off

bidirectional Adapter Interface for 2 Events

Interface

Event Inputs

Name	Comment	With

EI_SET	Set / Switch on	
EI_RESET	Reset / Switch off	

Event Outputs

Name	Comment	With
EO_SET	Set / Switch on	
EO_RESET	Reset / Switch off	

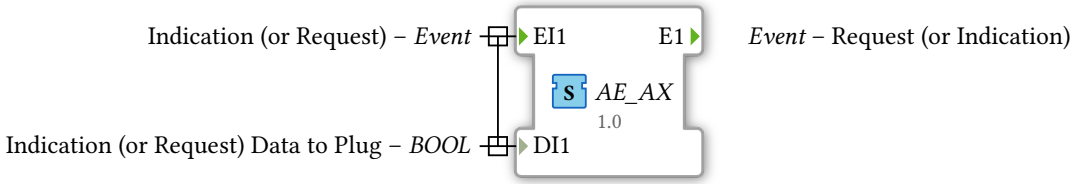
Bidirectional EVENT Adapter

🎧 Podcast

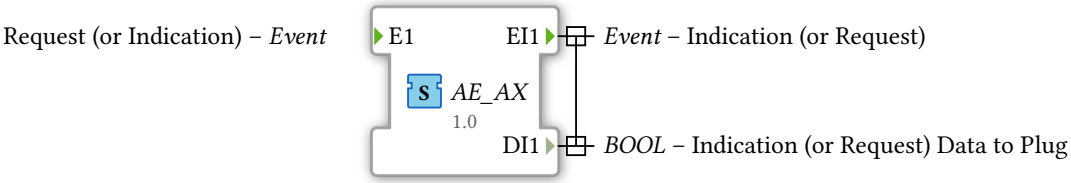
- E_TOF vs. FB_TOF: Der Event-Timer, der nicht zyklisch tickt – Revolution für Automatisierungssysteme?
- EventFBs nach IEC 61499: Legosteine der Automatisierung – So funktionieren Ereignis-Funktionsbausteine
- IEC 61499: Revolution der Automatisierung – Event-gesteuerte FBs und verteilte Systeme erklärt
- E_REND: Event Synchronization in IEC 61499
- The Unstoppable Counter: Why IEC 61499's ECTU Guarantees Safe, Event-Driven Control (and Never Overflows)

EVENT_BOOL

AE_AX
Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



bidirectional Adapter Interface for 1 Event (forward) and 1 Bool (backward, AX-style)

Interface

Event Inputs

Name	Comment	With
EI1	Indication (or Request)	DI1

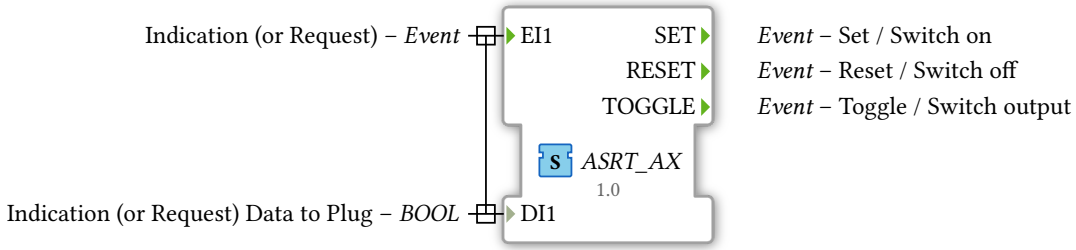
Event Outputs

Name	Comment	With
E1	Request (or Indication)	

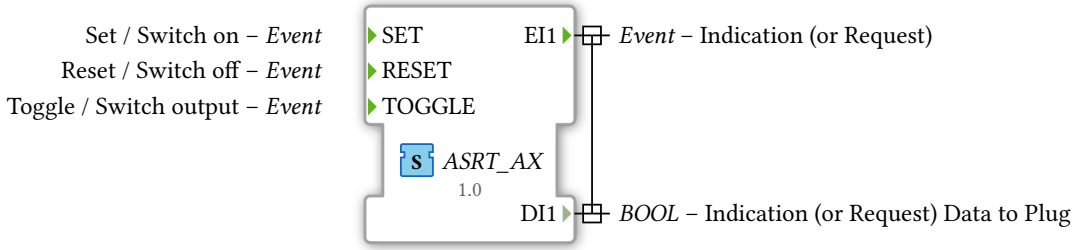
Input Vars

Name	Type	Comment
DI1	BOOL	Indication (or Request) Data to Plug

ASRT_AX
Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



bidirectional Adapter Interface for 3 Events (forward, Set/Reset/Toggle) and 1 Bool (backward, AX-style)

Interface

Event Inputs

Name	Comment	With
EI1	Indication (or Request)	DI1

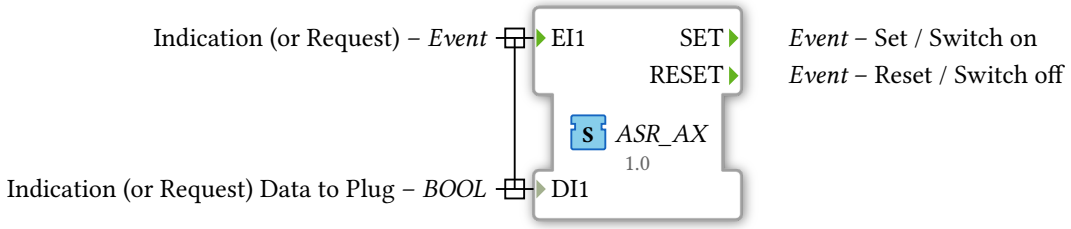
Event Outputs

Name	Comment	With
SET	Set / Switch on	
RESET	Reset / Switch off	
TOGGLE	Toggle / Switch output	

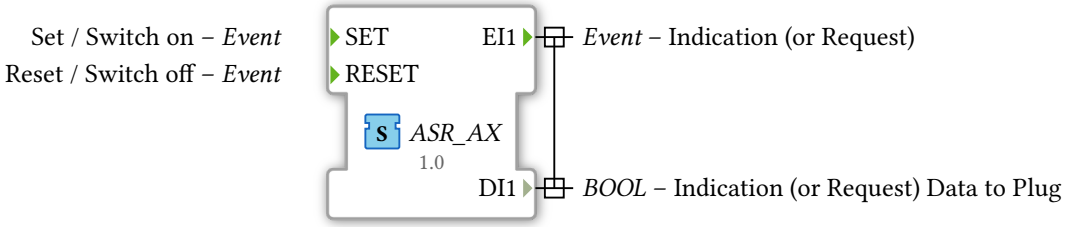
Input Vars

Name	Type	Comment
DI1	BOOL	Indication (or Request) Data to Plug

ASR_AX
Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



bidirectional Adapter Interface for 2 Events (forward, Set/Reset) and 1 Bool (backward, AX-style)

Interface

Event Inputs

Name	Comment	With
EI1	Indication (or Request)	DI1

Event Outputs

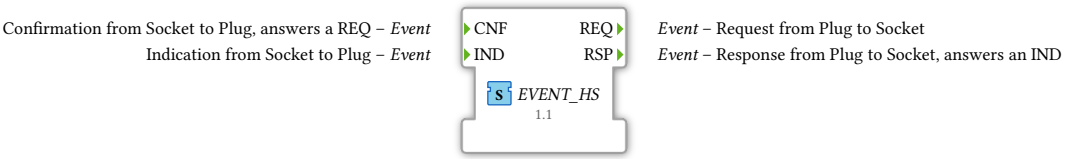
Name	Comment	With
SET	Set / Switch on	
RESET	Reset / Switch off	

Input Vars

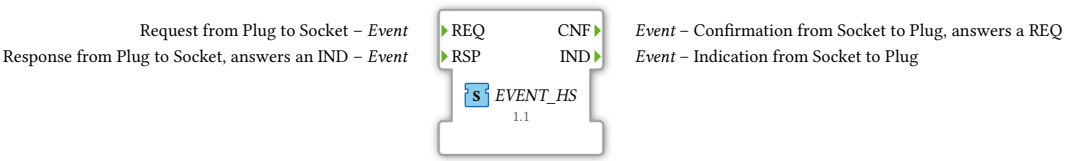
Name	Type	Comment
DI1	BOOL	Indication (or Request) Data to Plug

Handshake

EVENT_HS
Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **EVENT_HS**-Adapter realisiert das **Handshake-Muster** aus dem IEC 61499 Primer (Modul 6 – Designmethoden und -muster, Valeriy Vyatkin). Er bündelt die vier klassischen Service-Primitive *Request*, *Confirm*, *Indication* und *Response* in einer einzigen Adapter-Verbindung und verzichtet dabei vollständig auf Datenvariablen (dataless). Dadurch wird eine reine Ereignissignalisierung zwischen zwei Kommunikationspartnern ermöglicht, ohne dass separate Ereignisverbindungen oder zusätzliche Datenleitungen benötigt werden.

Der Adapter definiert zwei Rollen: den **Plug** (links, Requester/Client) und den **Socket** (rechts, Responder/Server). Der Plug sendet REQ/RSP und empfängt CNF/IND, während der Socket genau umgekehrt arbeitet. Diese Zuordnung entspricht der natürlichen Leserichtung im FBNetwork-Editor und ist seit Version 1.1 des Adapters festgelegt.

Schnittstellenstruktur

Ereignis-Eingänge

- **CNF** – *Confirmation*: Bestätigung vom Socket an den Plug, antwortet auf eine REQ-Anfrage.
- **IND** – *Indication*: Unaufgeforderte Meldung vom Socket an den Plug (z. B. über ein extern aufgetretenes Ereignis).

Ereignis-Ausgänge

- **REQ** – *Request*: Anfrage vom Plug an den Socket („bitte führe X aus“).
- **RSP** – *Response*: Antwort vom Plug an den Socket, beantwortet eine IND („Indication empfangen“).

Daten-Eingänge

Keine – es werden keine Datenwerte übertragen.

Daten-Ausgänge

Keine – der Adapter ist bewusst datenlos.

Adapter

Der EVENT_HS wird als Adaptertyp verwendet und stellt die Verbindung zwischen zwei FBs her. Er besitzt standardmäßig einen **Plug** (linke Seite) und einen **Socket** (rechte Seite). Die Rollen sind fest definiert: Der Plug agiert als Requester, der Socket als Responder.

Funktionsweise

Der Adapter definiert zwei spezifische Service-Sequenzen, die die zeitliche Abfolge der Ereignisse festlegen:

1. **request_confirm**
   - Der Plug sendet ein **REQ**-Ereignis an den Socket.
   - Der Socket verarbeitet die Anfrage und sendet anschließend ein **CNF**-Ereignis zurück an den Plug. Dieses Muster wird typischerweise für synchrone Aufträge verwendet, bei denen der Aufrufer eine Bestätigung erwartet.
2. **indication_response**
   - Der Socket sendet ein **IND**-Ereignis an den Plug (z. B. als Folge eines externen Ereignisses).
   - Der Plug empfängt die Indication und quittiert diese mit einem **RSP**-Ereignis an den Socket. Hierbei handelt es sich um eine asynchrone Benachrichtigung mit anschließender Bestätigung.

Die beiden Sequenzen sind unabhängig voneinander und können zeitlich verschränkt auftreten. Da keine Daten übertragen werden, eignet sich der Adapter für reine Steuer- und Synchronisationsaufgaben.

Technische Besonderheiten

- **Dataless Design**: Es werden keine Datenvariablen deklariert – ausschließlich Ereignisse werden ausgetauscht. Dies reduziert den Ressourcenbedarf und die Komplexität.
- **Eindeutige Rollenverteilung**: Seit Version 1.1 sind die Ereignis-Eingänge und -Ausgänge vertauscht, sodass der Plug die Requester-Rolle und der Socket die Responder-Rolle einnimmt. Diese Anpassung verbessert die Übersichtlichkeit im Netzwerk-Editor.
- **Compiler-Validierung**: Der ECC-Compiler von 4diac prüft lediglich, ob das referenzierte Ereignis (z. B. `HS.REQ`) auf dem Adapter existiert – nicht jedoch, ob die Richtung an Plug/Socket kohärent ist.

Entwickler müssen daher sicherstellen, dass die Verdrahtung und die ECC-Implementierung semantisch korrekt sind.
- **Minimalismus**: Der Adapter bildet exakt das auf Slide 72 des genannten Kurses gezeigte Muster ab, ohne zusätzliche Erweiterungen.

Zustandsübersicht

Der Adapter selbst besitzt keinen internen Zustandsautomaten. Stattdessen definiert er zwei klar abgegrenzte Service-Sequenzen:

- **Zustand A – Request/Confirm**: Nach dem Senden von `REQ` wartet der Plug auf `CNF`. Diese Sequenz ist synchron.
- **Zustand B – Indication/Response**: Nach dem Empfang von `IND` muss der Plug mit `RSP` antworten. Diese Sequenz ist asynchron.

Die beiden Sequenzen können unabhängig voneinander und auch überlappend ausgeführt werden. Es gibt keine feste Reihenfolge zwischen den Sequenzen.

Anwendungsszenarien

- **Auftragsbestätigung**: Ein FB sendet eine Anfrage (REQ) an einen anderen FB und wartet auf dessen Bestätigung (CNF), z. B. für das Auslösen einer Aktion mit Rückmeldung.
- **Ereignis-Benachrichtigung**: Ein überwachter Prozess erzeugt ein unerwartetes Ereignis (IND), das an den Plug gemeldet wird. Dieser quittiert den Empfang mit RSP.
- **Steuerung ohne Daten**: Überall dort, wo nur Signale (ohne Daten) ausgetauscht werden müssen, z. B. Trigger, Alarme oder Synchronisationspunkte.
- **Strukturierte Kommunikation**: Statt mehrere einzelne Ereignisverbindungen zu verwenden, bündelt der Adapter die vier Primitive in einer einzigen Schnittstelle – dies erhöht die Lesbarkeit und Wartbarkeit des Systems.

Vergleich mit ähnlichen Bausteinen

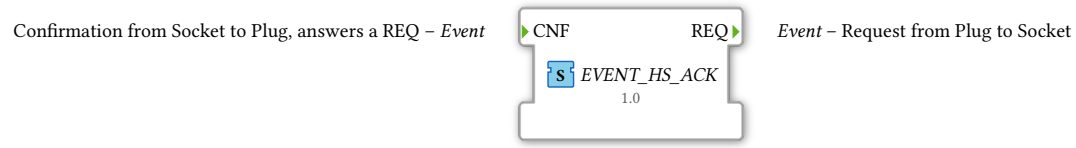
Im Gegensatz zu Adaptern mit Datenübertragung (z. B. `ANY`-basierten Adaptern) überträgt der EVENT_HS ausschließlich Ereignisse. Dadurch entfällt die Notwendigkeit, Datenwerte zu typisieren oder zu serialisieren. Gegenüber der Verwendung von vier separaten Ereignisverbindungen (REQ, CNF, IND, RSP) bietet der Adapter den Vorteil, dass die Zusammengehörigkeit der Primitive explizit gekapselt ist und die Verkabelung im FBNetwork-Editor übersichtlicher wird. Nachteilig ist, dass keine zusätzlichen Informationen (z. B. Fehlercodes oder Nutzdaten) transportiert werden können – dafür sind andere Adapter wie `DATA_HS` oder eigene maßgeschneiderte Typen erforderlich.

Fazit

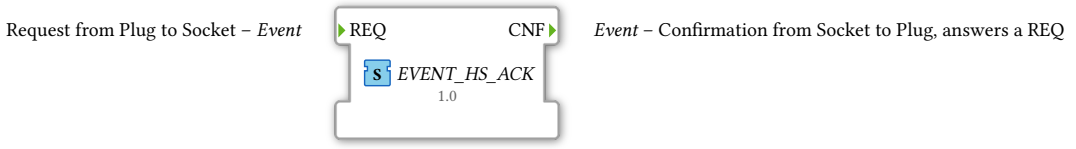
Der **EVENT_HS**-Adapter ist eine kompakte und saubere Lösung für reine Ereignissignalisierung nach dem Handshake-Muster. Er eignet sich besonders für einfache Steuer- und Synchronisationsaufgaben, bei denen keine Daten übertragen werden müssen. Durch die klare Rollenverteilung und die intuitive Anordnung im Editor ist er leicht einzusetzen und zu verstehen. Die dokumentierte Einschränkung der Compiler-Validierung sollte jedoch bei der Entwicklung stets beachtet werden, um logische Fehler frühzeitig zu vermeiden. Insgesamt ist der Adapter ein nützliches Werkzeug für die modulare und strukturierte Kommunikation in IEC-61499-Systemen.

EVENT_HS_ACK

Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Adapter `EVENT_HS_ACK` ist eine reduzierte Variante des Handshake-Adapters `EVENT_HS` aus dem IEC-61499-Entwurfsmuster. Er implementiert ausschließlich die Request-Confirm-Hälfte (REQ/CNF) der vollständigen Handshake-Vokabulars (REQ/CNF/IND/RSP). Damit wird eine zuverlässige Auftragsbestätigung ohne Datenübertragung realisiert – der Socket (Responder) kann auf eine Request (REQ) nur mit einer Confirmation (CNF) antworten, besitzt aber keine Möglichkeit, unaufgefordert Indikationen an den Plug (Requester) zu senden. Dies macht den Adapter ideal für einfache Steuerungs- und Synchronisationsaufgaben, bei denen lediglich eine positive Quittung auf eine Anforderung benötigt wird.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
CNF	Event	Bestätigung vom Socket zum Plug, beantwortet eine REQ

Ereignis-Ausgänge

Name	Typ	Kommentar
REQ	Event	Anforderung (Request) vom Plug zum Socket

Daten-Eingänge

Keine – der Adapter ist datenlos.

Daten-Ausgänge

Keine – der Adapter ist datenlos.

Adapter

Keine – der Baustein ist selbst ein Adapter (Plug/Socket).

Funktionsweise

Der Adapter definiert eine einfache Handshake-Sequenz mit zwei Transaktionen:

1. Der Plug (Requester) sendet ein `REQ`-Ereignis über die linke Schnittstelle. Dieses wird unmittelbar als `REQ`-Ereignis an der rechten Schnittstelle (Socket) ausgegeben.
2. Der Socket (Responder) beantwortet die Anforderung mit einem `CNF`-Ereignis. Dieses wird vom Adapter als `CNF`-Ereignis an die linke Schnittstelle (Plug) weitergeleitet.

Somit erhält jede `REQ`-Anforderung eine zugehörige `CNF`-Bestätigung. Die Rollenverteilung folgt dem Standard des `EVENT_HS`-Adapters: Der Plug agiert als Client/Requester, der Socket als Server/Responder. Der Adapter besitzt keine interne Zustandslogik – er dient lediglich als ereignisbasierte Verbindung und ermöglicht eine lose Kopplung zwischen Komponenten.

Technische Besonderheiten

- **Datenlos**: Es werden keine Daten übertragen – nur Ereignisse. Dadurch wird der Adapter besonders schlank und effizient.
- **Reduziertes Vokabular**: Im Gegensatz zum vollständigen `EVENT_HS` fehlen die Ereignisse `IND` (Indikation) und `RSP` (Response). Der Socket kann also nie unaufgefordert etwas an den Plug melden; er reagiert ausschließlich auf `REQ`.
- **Design-Pattern-Herkunft**: Der Adapter gehört zur Handshake-Familie und ist in einem separaten Dokument (`HandshakePattern.md`) detailliert beschrieben.
- **Eindeutige Schnittstellenrollen**: Durch die klare Trennung von Plug und Socket wird die Verbindungsrichtung im System explizit festgelegt.

Zustandsübersicht

Da der Adapter keine eigenen Zustände besitzt, gibt es keine Zustandsautomaten im Inneren. Die Kommunikation erfolgt rein ereignisgesteuert. Im Rahmen einer Verbindung können jedoch die folgenden Phasen unterschieden werden:

- **Idle**: Es liegt kein Ereignis an.
- **Warten auf Bestätigung**: Nach dem Senden von `REQ` wartet der Plug auf die `CNF`-Antwort.
- **Bestätigt**: Der Plug hat die `CNF` erhalten; die Transaktion ist abgeschlossen.

Diese Phasen werden nicht vom Adapter selbst verwaltet, sondern durch die angeschlossenen Funktionsbausteine.

Anwendungsszenarien

- **Steuerungsquittungen**: Eine SPS sendet einen Befehl an einen Aktor und wartet auf die Bestätigung, dass der Befehl ausgeführt wurde.
- **Synchronisation**: Zwei parallele Prozesse sollen sich gegenseitig per Request/Confirm abstimmen, ohne Daten auszutauschen.
- **Subsystem-Initialisierung**: Ein übergeordnetes System fordert ein Subsystem zum Start auf und erhält eine Bestätigung, sobald der Startvorgang abgeschlossen ist.
- **Redundanz**: In Verbindung mit einer Automatisierungskomponente kann `EVENT_HS_ACK` eine einfache, zuverlässige Übermittlung von Quittungen gewährleisten.

Vergleich mit ähnlichen Bausteinen

Baustein	Ereignisse	Datentyp	Beschreibung
`EVENT_HS`	REQ, CNF, IND, RSP	– (datenlos)	Vollständiger Handshake mit Indikation/Response
`EVENT_HS_UNI`	REQ, CNF	– (datenlos)	Unidirektionaler Handshake (nur Anforderung/Bestätigung)
`EVENT_HS_ACK`	REQ, CNF	– (datenlos)	Wie `EVENT_HS_UNI`, aber jede REQ erhalten eine CNF (aktive Bestätigung)
`EVENT_HS_UNI_WSTRING`	REQ, CNF	WSTRING	Unidirektionaler Handshake mit Datenübertragung
`EVENT_HS_ACK_WSTRING`	REQ, CNF	WSTRING	Handshake mit Daten und garantierter Bestätigung

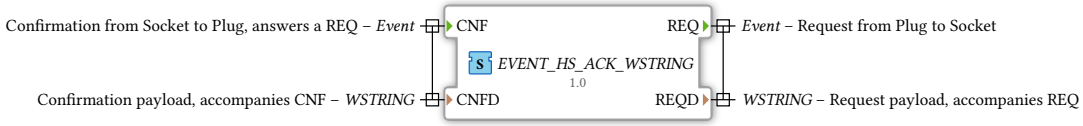
Im Vergleich zu `EVENT_HS` ist `EVENT_HS_ACK` auf die minimal erforderliche Kommunikation reduziert. Gegenüber `EVENT_HS_UNI` stellt `EVENT_HS_ACK` sicher, dass jede Anforderung tatsächlich quittiert wird – eine einfache, aber wichtige Erweiterung für Systeme, die eine positive Rückmeldung verlangen.

Fazit

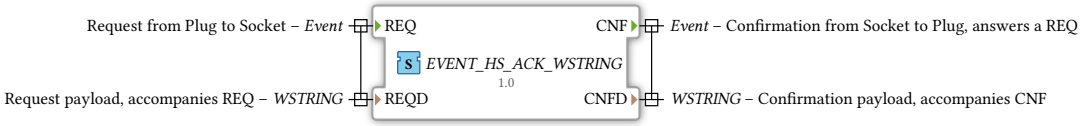
`EVENT_HS_ACK` ist ein schlanker, datenloser Adapter für eine klar umrissene Aufgabe: eine Anforderung senden und eine Bestätigung empfangen. Er eignet sich hervorragend für Anwendungen, die eine robuste, ereignisbasierte Quittung benötigen, ohne den Overhead eines vollständigen Handshakes oder einer Datenübertragung. Durch die klare Plug/Socket-Rollenverteilung und die garantierten REQ-CNF-Paare bietet er eine einfache, aber zuverlässige Grundlage für die Kommunikation zwischen IEC-61499-Komponenten.

EVENT_HS_ACK_WSTRING

Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Adapter `EVENT_HS_ACK_WSTRING` gehört zur Familie der Handshake-Adapter (nach dem IEC 61499-Muster). Er realisiert eine reduzierte Variante des klassischen `EVENT_HS`-Adapters, bei der nur die **Request-/Confirm-Kante** (REQ/CNF) verwendet wird. Beide Ereignisse transportieren jeweils einen Datenparameter vom Typ `WSTRING`. Dadurch eignet sich dieser Adapter für Anwendungen, in denen eine Anfrage und eine Bestätigung mit einer textuellen Nutzlast (z. B. im Format `"name,value"`) ausgetauscht werden müssen.

Der Adapter stellt eine Plug-/Socket-Schnittstelle bereit: Die **Plug**-Seite (links) agiert als Requester/Client, die **Socket**-Seite (rechts) als Responder/Server. Es gibt keine unaufgeforderten Indications (IND) oder Responses (RSP) – die Kommunikation folgt strikt dem Muster: Plug sendet REQ mit REQD, Socket antwortet mit CNF und CNFD.

Schnittstellenstruktur

Ereignis-Eingänge

- **CNF** (Confirmation)
  Wird von der Socket-Seite ausgelöst, um eine zuvor gesendete REQ-Anfrage zu bestätigen.
  Mit Datenparameter: `CNFD` (WSTRING) – die Bestätigungsnutzlast.

Ereignis-Ausgänge

- **REQ** (Request)
  Wird von der Plug-Seite ausgelöst, um eine Anfrage an die Socket-Seite zu senden.
  Mit Datenparameter: `REQD` (WSTRING) – die Anfragenutzlast.

Daten-Eingänge

- **CNFD** (WSTRING) – Begleitdaten zum Ereignis `CNF`. Enthält die Bestätigungsinformation (z. B. `"push,100"`), die vom Socket an den Plug zurückgegeben wird.

Daten-Ausgänge

- **REQD** (WSTRING) – Begleitdaten zum Ereignis `REQ`. Enthält die Anfrageinformation (z. B. `"push,100"`), die vom Plug an den Socket gesendet wird.

Adapter

Der Adapter selbst definiert eine bidirektionale Schnittstelle mit zwei Rollen:

- **PLUG** (links) – kontrolliert die Richtung der Ereignisse: REQ wird **gesendet**, CNF wird **empfangen**.
- **SOCKET** (rechts) – spiegelbildlich: REQ wird **empfangen**, CNF wird **gesendet**.

Die Zuordnung erfolgt beim Verbinden zweier FB-Instanzen, wobei der Adapter entweder als **Plug** oder als **Socket** verdrahtet wird.

Funktionsweise

Der Adapter implementiert einen einfachen Request-Confirm-Handshake mit textuellen Nutzlasten. Die Abfolge ist wie folgt:

1. **Plug** setzt den Ausgangsdatenwert `REQD` und sendet das Ereignis **REQ**.
2. **Socket** empfängt das Ereignis **REQ** und den zugehörigen Wert `REQD`.
3. **Socket** verarbeitet die Anfrage und setzt den Eingangswert `CNFD` (bei der Socket-Instanz) und löst das Ereignis **CNF** aus.
4. **Plug** empfängt das Ereignis **CNF** und den Wert `CNFD`.

Wichtig: Es gibt **keine Indication (IND)** und **keine Response (RSP)**. Das bedeutet, dass die Socket-Seite nicht von sich aus eine unaufgeforderte Nachricht an den Plug senden kann – jede Kommunikation muss mit einem REQ des Plugs beginnen und wird durch ein CNF des Sockets abgeschlossen.

Technische Besonderheiten

- **Nur Request/Confirm-Pfad** – bewusste Reduktion des vollen Handshake-Vokabulars (REQ, CNF, IND, RSP) auf die beiden Ereignisse `REQ` und `CNF`.
- **WSTRING-Nutzlast** – beide Ereignisse sind mit einem Datenparameter vom Typ `WSTRING` verknüpft, was die Übertragung beliebiger Textdaten (z. B. Kommando-Strings) ermöglicht.
- **Klare Rollenverteilung** – Plug ist immer der Initiator, Socket der Responder. Dies erleichtert die Implementierung deterministischer Kommunikationsprotokolle.
- **Einfaches Dienstmodell** – Die Service-Sequenz `request_confirm` definiert exakt eine Transaktionskette, sodass keine zusätzlichen Zustände oder Nebenläufigkeiten behandelt werden müssen.

Zustandsübersicht

Der Adapter besitzt keinen expliziten internen Zustandsautomaten – der Protokollablauf ist durch die Service-Sequenz festgelegt. Es lassen sich jedoch zwei logische Phasen unterscheiden:

Phase	Aktivität
Warten auf REQ	Plug kann einen neuen REQ senden. Sobald REQ gesendet wird, wartet der Plug auf das zugehörige CNF.
Warten auf CNF	Socket hat REQ empfangen und arbeitet die Anfrage ab. Nach Abschluss sendet Socket das CNF. Danach ist der Zyklus abgeschlossen und ein neuer REQ kann beginnen.

Anwendungsszenarien

- **Kommandosteuerung**: Senden von Befehlen wie `"push,100"` an einen Server und Quittierung mit demselben oder einem modifizierten Wert.
- **Parameterabfrage**: Anfordern eines bestimmten Werts per REQ und Empfang des Ergebnisses im CNF.
- **Einfache Synchronisation**: Sicherstellen, dass eine Aktion nur nach erfolgreicher Bestätigung ausgeführt wird.
- **Bidirektionale Punkt-zu-Punkt-Kommunikation** in IEC-61499-Systemen, bei der keine unaufgeforderten Meldungen benötigt werden.

Vergleich mit ähnlichen Bausteinen

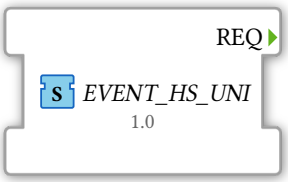
- **`EVENT_HS`** – Vollständiger Handshake mit REQ/CNF und IND/RSP. `EVENT_HS_ACK_WSTRING` ist eine reduzierte Variante ohne IND/RSP, dafür mit WSTRING-Payload.
- **`EVENT_HS_ACK`** – Bietet ebenfalls nur REQ/CNF, jedoch **ohne** Datenparameter. `EVENT_HS_ACK_WSTRING` erweitert diesen um die Übertragung von Nutzdaten.
- **`EVENT_HS_UNI`** – Verwendet nur REQ/CNF (ohne Daten) und ist für unidirektionale Bestätigungen gedacht.
- **`EVENT_HS_UNI_WSTRING`** – Ähnlich wie `EVENT_HS_UNI`, aber mit WSTRING-Payload (ohne CNF-Daten). Im Gegensatz dazu besitzt `EVENT_HS_ACK_WSTRING` auch Daten beim CNF, sodass eine echte Quittung mit Inhalt möglich ist.

Fazit

`EVENT_HS_ACK_WSTRING` ist ein kompakter und klar definierter Adapter für den IEC-61499-Handshake mit textueller Nutzlast. Durch die Reduktion auf Request/Confirm wird die Kommunikation deterministisch und leicht verständlich. Die Integration von WSTRING-Daten macht ihn flexibel für verschiedenste Anwendungen, in denen eine Anfrage mit einer Bestätigung inklusive Rückmeldung verbunden werden soll. Seine einfache Struktur erleichtert die Wiederverwendung und die Einbindung in größere Systeme.

EVENT_HS_UNI

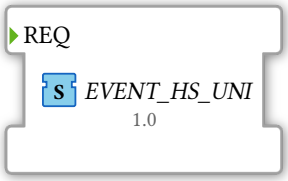
Plug (Adapter-Ausgang)



Event – Request/notification from Plug to Socket, no reply expected

Socket (Adapter-Eingang)

Request/notification from Plug to Socket, no reply expected – *Event*



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Adapter **EVENT_HS_UNI** gehört zur Familie der Handshake-Adapter nach dem Entwurfsmuster aus IEC 61499 (Modul 6, Valeriy Vyatkin). Er stellt eine stark reduzierte, unidirektionale Variante dar, die ausschließlich ein einzelnes Ereignis **REQ** vom Plug zum Socket überträgt. Es gibt keinerlei Bestätigungs- oder Antwortereignisse (CNF, IND, RSP) – der Adapter implementiert bewusst kein echter Handshake, sondern eine reine „Fire-and-Forget"-Benachrichtigung ohne Rückkanal. Dies macht ihn für Anwendungen geeignet, bei denen eine Bestätigung nicht erforderlich oder nicht gewünscht ist, und wo eine bestmögliche Zustellung ausreicht.

Schnittstellenstruktur

Der Adapter besitzt nur eine einzige Ereignis-Schnittstelle. Es gibt weder Daten-Ein- noch Daten-Ausgänge, keine Adapter weiter innerhalb des Bausteins und auch keinen zusätzlichen Ereignis-Eingang oder -Ausgang außer dem einen REQ-Ausgang.

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Name	Typ	Kommentar
`REQ`	Event	Anforderung/Benachrichtigung vom Plug zum Socket, keine Antwort erwartet

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Der Adapter selbst ist ein Plug/Socket-Adapter. Auf der Plug-Seite (linke Schnittstelle) wird das Ereignis `REQ` gesendet, auf der Socket-Seite (rechte Schnittstelle) wird es empfangen. Die Service-Definition dokumentiert diesen Ablauf als Sequenz „notify".

Funktionsweise

Der Adapter überträgt ein einzelnes Ereignis `REQ` unidirektional vom Plug zum Socket. Die Übertragung erfolgt ohne Daten und ohne Bestätigung. Der Plug (Absender) löst das Ereignis aus, der Socket (Empfänger) reagiert darauf. Es gibt keine Möglichkeit für den Socket, den Empfang zu quittieren oder die Verarbeitung zu beeinflussen. Der Plug kann nicht feststellen, ob die Benachrichtigung den Socket erreicht hat oder ob sie dort erfolgreich verarbeitet wurde – es handelt sich um eine reine Einweg-Signalisation mit Best-Effort-Charakter.

Der Service-Sequenz-Diagramm zeigt die Transaktion:

- **Plug** sendet `REQ` an **Socket** (event). Es gibt keine Rücktransaktion.

Technische Besonderheiten

- **Kein Handshake**: Im Gegensatz zu typischen Handshake-Adaptern wie `EVENT_HS` fehlen jegliche Antwortpfade (CNF, IND, RSP). Dadurch ist die Kommunikation extrem schlank und ressourcenschonend, aber auch unzuverlässig im Sinne einer Quittierung.
- **Keine Daten**: Es werden keinerlei Nutzdaten übertragen, nur das Ereignis selbst. Das reduziert die Schnittstellenkomplexität auf ein Minimum.

- **Unidirektionalität**: Die Verbindung erlaubt nur eine Richtung (Plug → Socket). Eine umgekehrte Kommunikation ist nicht vorgesehen.
- **Dokumentationszweck**: Das enthaltene `<Service>`-Element dient lediglich der Dokumentation (optionale Service-Sequenz-Diagramm) und ist für die Funktionalität nicht erforderlich – auch der Standard-Adapter `AE.adp` besitzt kein solches Element.
- **Kompatibilität**: Der Adapter gehört zur Familie der `EVENT_HS`-Adapter und teilt die Rolle Plug/Socket mit diesen. Er ist jedoch nur für den Einsatz als reine Benachrichtigung gedacht und ersetzt nicht einen vollwertigen Handshake.

Zustandsübersicht

Da der Adapter keinerlei Zustandslogik besitzt, existiert kein expliziter Zustandsautomat. Die einzige Aktion ist das Auslösen des `REQ`-Ereignisses. Es gibt keinen internen Speicher oder Zustand zu verwalten.

Anwendungsszenarien

- **Simpelste Signalisierung**: Wenn eine Komponente nur eine Information „Es ist etwas passiert“ an eine andere weitergeben muss, ohne Rückantwort oder Daten.
- **Best-Effort-Notify**: Für lose gekoppelte Systeme, bei denen eine Benachrichtigung nicht quittiert werden muss und ein Verlust tolerierbar ist (z. B. Statusmeldungen, die regelmäßig wiederholt werden).
- **Test- und Bildungszwecke**: Als minimales Beispiel für unidirektionale Kommunikation in IEC-61499-Systemen.
- **Reduzierte Variante im Framework**: Innerhalb der `EVENT_HS`-Familie als schlankstes Mitglied – geeignet, wenn die vollwertigen Handshake-Varianten überdimensioniert sind.

Vergleich mit ähnlichen Bausteinen

Baustein	Ereignisse	Daten	Richtung	Handshake	Verwendung
`EVENT_HS_UNI` (dieser)	Nur `REQ` (Plug → Socket)	Keine	Unidirektional	Nein	Fire-and-Forget-Benachrichtigung
`EVENT_HS`	`REQ`/`CNF`, `IND`/`RSP`	Keine	Bidirektional	Ja	Vollständiger Handshake
`EVENT_HS_UNI_WSTRING`	Nur `REQ` (Plug → Socket)	WSTRING	Unidirektional	Nein	Fire-and-Forget mit Datennutzlast
`EVENT_HS_ACK`	`REQ`/`CNF`	Keine	Bidirektional	Ja	Handshake mit Bestätigung, ohne Daten
`EVENT_HS_ACK_WSTRING`	`REQ`/`CNF`	WSTRING	Bidirektional	Ja	Handshake mit Bestätigung und Daten

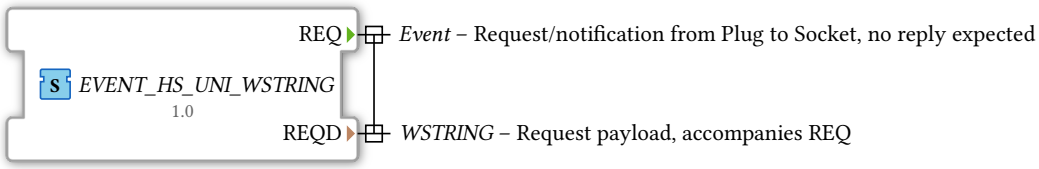
Der Unterschied zu `EVENT_HS` besteht vor allem im Wegfall aller Antwort- und Datenpfade. `EVENT_HS_UNI` ist gezielt für einfache, unidirektionale Benachrichtigungen konzipiert.

Fazit

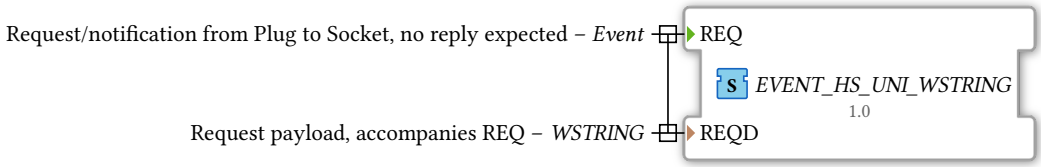
Der Adapter `EVENT_HS_UNI` ist eine extrem reduzierte Schnittstelle für unidirektionale, darlose Benachrichtigungen. Er eignet sich nur für Szenarien, bei denen eine Bestätigung oder Rückantwort nicht erforderlich ist und die Zustellung als „best effort“ akzeptiert wird. Seine Stärke liegt in der Einfachheit und geringen Ressourcenbelastung; er ist jedoch nicht als Ersatz für einen echten Handshake gedacht. Für Systeme, die eine zuverlässige Kommunikation mit Quittung benötigen, sind die anderen Varianten der `EVENT_HS`-Familie zu wählen.

EVENT_HS_UNI_WSTRING

Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Adapter `EVENT_HS_UNI_WSTRING` gehört zur `EVENT_HS`-Familie (Handshake-Designmuster nach IEC 61499) und stellt eine reduzierte, unidirektionale Variante dar. Er dient der Übertragung einer „Fire-and-Forget"-Nachricht (ohne Antwort) mit einem WSTRING-Payload vom Plug zum Socket. Der Name „UNI" weist darauf hin, dass nur eine Richtung (ein Ereignis) existiert – es gibt weder Bestätigungen noch Rückkanäle. Diese Variante erweitert `EVENT_HS_UNI` um ein Datenfeld `REQD`, das zusammen mit dem Ereignis `REQ` gesendet wird.

Der Adapter ist für Szenarien gedacht, in denen eine einseitige, best-effort Benachrichtigung mit einem String-Payload ausreicht und keine Rückmeldung über Empfang oder Verarbeitung erforderlich ist.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Name	Typ	Mit Variablen	Kommentar
REQ	Event	REQD	Anforderung/Benachrichtigung vom Plug zum Socket; keine Antwort erwartet.

Daten-Eingänge

Es sind keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Name	Typ	Kommentar
REQD	WSTRING	Nutzlast, die zusammen mit dem Ereignis `REQ` übertragen wird.

Adapter

Der Baustein ist selbst ein Adapter und besitzt daher keine weiteren Adapter-Schnittstellen.

Funktionsweise

Der Adapter realisiert eine einfache, unidirektionale Ereignisübertragung: Der Plug (linke Schnittstelle) erzeugt das Ereignis `REQ` und übergibt dabei den Wert von `REQD` als Parameter. Der Socket (rechte Schnittstelle) empfängt dieses Ereignis und erhält die zugehörige Nutzlast.

Die Service-Sequenz `notify` im XML beschreibt diesen Ablauf explizit als eine Transaktion:

- **Input**: `PLUG` ⟶ Ereignis `REQ` mit Parameter `REQD`
- **Output**: `SOCKET` ⟶ Ereignis `REQ` mit Parameter `REQD`

Es existiert keinerlei Rückkanal; weder eine Bestätigung (`CNF`), noch eine Anzeige (`IND`) oder eine Antwort (`RSP`). Der Adapter leitet das Ereignis einfach vom Plug zum Socket weiter.

Technische Besonderheiten

- **Kein echter Handshake**: Wie der Name andeutet, handelt es sich um eine „Fire-and-Forget“-Variante. Der Socket hat keine Möglichkeit, die Anfrage oder deren Nutzlast zu bestätigen oder abzulehnen. Der Plug kann nicht feststellen, ob der Socket das Ereignis erhalten oder verarbeitet hat.
- **Datenübertragung**: Die Nutzlast `REQD` ist vom Typ `WSTRING` und wird nur zusammen mit `REQ` übertragen. Es gibt keine separate Datenverbindung.
- **Dokumentarischer Service**: Der `<Service>`-Block im XML ist rein informativ (gemäß XSD optional) und nicht erforderlich, damit die Adapterverbindung Ereignisse weiterleitet. Er wurde dennoch hinzugefügt, um die einzige Transaktion zu verdeutlichen.
- **Rollensplit**: Wie bei allen Adaptoren der `EVENT_HS`-Familie besitzt der Plug die aktive Rolle (Sender), der Socket die passive Rolle (Empfänger). Die Richtung ist im Adaptertyp festgelegt.

Zustandsübersicht

Der Adapter besitzt keine internen Zustände. Er ist eine reine „Durchreiche“ für das Ereignis `REQ` und dessen Nutzlast `REQD`. Es gibt genau eine Übertragung pro Aufruf, ohne Verzögerung oder Zwischenspeicherung.

Anwendungsszenarien

- **Push-Nachrichten**: Senden von Benachrichtigungen oder Statusmeldungen, bei denen keine Bestätigung erforderlich ist (z. B. Log-Einträge, Sensorwerte an einen Anzeigeprozess).
- **Ereignisbasierte Datenweitergabe**: Übertragung eines Strings als Teil eines Ereignisses, wenn nur eine Richtung kommuniziert wird.
- **Integration in bestehende Handshake-Strukturen**: Als Alternative zu `EVENT_HS_UNI`, wenn zusätzlich ein WSTRING-Payload benötigt wird, aber weiterhin auf Rückkanäle verzichtet werden kann.

Vergleich mit ähnlichen Bausteinen

Innerhalb der `EVENT_HS`-Familie gibt es mehrere Varianten:

Baustein	Richtung	Payload	Bestätigung	Zweck
`EVENT_HS`	bidirektional	–	ja	Vollständiger Handshake (REQ/CNF, IND/RSP)
`EVENT_HS_UNI`	unidirektional	–	nein	Fire-and-Forget-Ereignis ohne Nutzlast
`EVENT_HS_ACK`	unidirektional	–	ja	Fire-and-Forget mit Bestätigung (ACK)
`EVENT_HS_ACK_WSTRING`	unidirektional	WSTRING	ja	Fire-and-Forget mit Payload und Bestätigung
`EVENT_HS_UNI_WSTRING`	unidirektional	WSTRING	nein	Fire-and-Forget mit Payload ohne Bestätigung

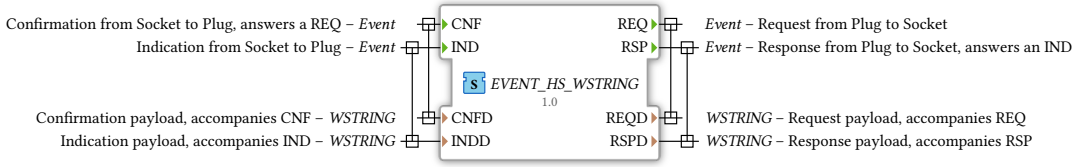
Im Vergleich zu `EVENT_HS_UNI` fügt dieser Baustein die Nutzlast `REQD` hinzu. Gegenüber `EVENT_HS_ACK_WSTRING` entfällt die Bestätigung, wodurch der Kommunikationsaufwand minimal bleibt, aber auch keine Erfolgskontrolle existiert.

Fazit

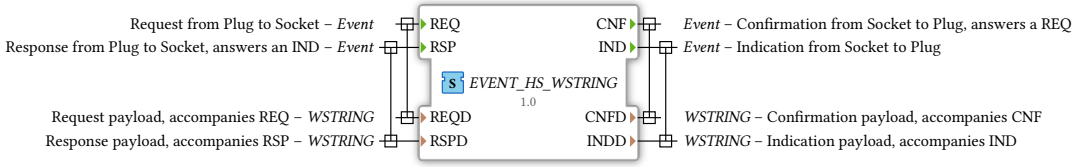
`EVENT_HS_UNI_WSTRING` ist ein schlanker Adapter für unidirektionale, best-effort Ereignisübertragungen mit einem String-Payload. Er eignet sich ausschließlich für Szenarien, in denen eine fehlende Rückmeldung akzeptabel ist – etwa bei reinen Benachrichtigungen oder Statusmeldungen. Aufgrund der fehlenden Bestätigungsmechanismen sollte er nicht verwendet werden, wenn eine verlässliche Zustellung oder Verarbeitung überwacht werden muss. In solchen Fällen ist eine Variante mit Bestätigung (z. B. `EVENT_HS_ACK_WSTRING`) vorzuziehen.

EVENT_HS_WSTRING

Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



• – ○ • – ○ • – ○ •

Einleitung

Der Adapter `EVENT_HS_WSTRING` ist eine datentragende Variante des bekannten **Handshake-Entwurfsmusters** für IEC 61499-Applikationen. Er ergänzt das klassische `EVENT_HS`-Adapter-Interface um eine `WSTRING`-Nutzlast pro Ereignis, sodass bei jedem Handshake-Schritt zusätzliche Informationen übertragen werden können. Das Muster ermöglicht eine zuverlässige, sequenzielle Kommunikation zwischen einem Client (Plug) und einem Server (Socket) und eignet sich besonders für Service-orientierte Architekturen und Prozessdaten-Schnittstellen.

Der Adapter basiert auf dem IEC 61499-Standard und ist für die Verwendung in der 4diac-IDE konzipiert. Er implementiert die Rollenverteilung **Plug** (Client/Requester) und **Socket** (Server/Responder) und unterstützt die beiden Service-Sequenzen *request_confirm* und *indication_response*. Die Datenfelder `CNFD`, `INDD`, `REQD` und `RSPD` transportieren jeweils die zugehörige Payload.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Datentyp	Kommentar
`CNF`	Event	Bestätigung vom Socket zum Plug; beantwortet eine `REQ`-Anfrage.
`IND`	Event	Unaufgeforderte Anzeige vom Socket zum Plug.

Ereignis-Ausgänge

Ereignis	Datentyp	Kommentar
`REQ`	Event	Anforderung vom Plug zum Socket.
`RSP`	Event	Antwort vom Plug zum Socket; beantwortet eine `IND`.

Daten-Eingänge

Variable	Datentyp	Kommentar
`CNFD`	`WSTRING`	Nutzlast, die mit dem `CNF`-Ereignis übertragen wird.
`INDD`	`WSTRING`	Nutzlast, die mit dem `IND`-Ereignis übertragen wird.

Daten-Ausgänge

Variable	Datentyp	Kommentar
`REQD`	`WSTRING`	Nutzlast, die mit dem `REQ`-Ereignis übertragen wird.
`RSPD`	`WSTRING`	Nutzlast, die mit dem `RSP`-Ereignis übertragen wird.

Adapter

Der Adapter besitzt zwei Schnittstellen:

- **Plug** (links): Dies ist der Client/Requester. Er sendet `REQ` und `RSP` und empfängt `CNF` und `IND`.
- **Socket** (rechts): Dies ist der Server/Responder. Er empfängt `REQ` und `RSP` und sendet `CNF` und `IND`.

Die Zuordnung erfolgt über die Service-Definition, in der der Plug als `LeftInterface` und der Socket als `RightInterface` festgelegt sind.

Funktionsweise

Der Adapter realisiert zwei unabhängige Handshake-Sequenzen, die über die Service-Sequenzen definiert werden:

1. **request_confirm** – Dies ist der klassische Request-Confirm-Zyklus:
   - Der Plug sendet ein `REQ`-Ereignis zusammen mit der Nutzlast `REQD` an den Socket.
   - Der Socket empfängt dieses Ereignis und verarbeitet die Daten.
   - Anschließend sendet der Socket ein `CNF`-Ereignis mit `CNFD` zurück an den Plug, um die Ausführung zu bestätigen.
2. **indication_response** – Dies ermöglicht eine Server-initiierte Kommunikation:
   - Der Socket sendet unaufgefordert ein `IND`-Ereignis mit `INDD` an den Plug (z. B. zur Signalisierung eines Ereignisses).
   - Der Plug empfängt dies und antwortet mit einem `RSP`-Ereignis samt `RSPD` an den Socket.

Beide Sequenzen sind vollständig getrennt und können unabhängig voneinander ablaufen. Die Datenvariablen sind jeweils an das zugehörige Ereignis gebunden und werden mit dem Ereignis übertragen. Dadurch ist sichergestellt, dass die Nutzlast immer im korrekten Kontext ankommt.

Technische Besonderheiten

- **Datenvariante des Handshake-Musters**: Im Vergleich zum datenlosen `EVENT_HS`-Adapter trägt jedes Ereignis eine `WSTRING`-Nutzlast. Dies entspricht dem generischen *Service*-Adapter aus dem Lehrbuch von Valeriy Vyatkin (IEC 61499 Primer, Modul 6, Folie 48).
- **Fester Datentyp**: Die Payload ist aktuell auf `WSTRING` festgelegt. Bei Bedarf kann eine typspezifischere Variante abgeleitet werden.
- **Eindeutige Rollentrennung**: Plug und Socket haben klar definierte Aufgaben; dies erleichtert die Wiederverwendung in verschiedenen Applikationen.
- **Paketzuordnung**: Der Adapter ist im Paket `adapter::types::bidirectional::Handshake` organisiert, was eine saubere Einbindung in größere Projekte ermöglicht.
- **Versionierung**: Der Adapter ist mit Version 1.0 und dem Autor Franz Höpfinger (HR Agrartechnik GmbH) hinterlegt. Die Historie enthält Verweise auf das Vorlesungsmaterial und die zugehörige Dokumentation.

Zustandsübersicht

Der Adapter selbst besitzt keinen expliziten Zustandsautomaten, die beiden Service-Sequenzen definieren jedoch klare Zustandsübergänge:

Sequenzname	Schritt	Ereignis / Richtung	Beschreibung
`request_confirm`	1	REQ (Plug → Socket)	Plug sendet Anforderung und Nutzlast.
`request_confirm`	2	CNF (Socket → Plug)	Socket bestätigt die Ausführung.
`indication_response`	1	IND (Socket → Plug)	Socket sendet eine unaufgeforderte Anzeige.
`indication_response`	2	RSP (Plug → Socket)	Plug quittiert die Anzeige.

In der Praxis bedeutet dies: Nach dem Senden eines `REQ` wartet der Plug auf ein `CNF`; nach dem Empfang eines `IND` muss der Plug mit `RSP` antworten. Es gibt keine parallelen Überlappungen innerhalb einer Sequenz, wodurch eine deterministische Kommunikation gewährleistet ist.

Anwendungsszenarien

- **Service-Aufrufe in industriellen Steuerungen**: Ein Steuerungsgerät (Plug) sendet einen Befehl wie `"push,100"` an ein Servicemodul (Socket) und erhält eine Bestätigung.
- **Status- und Ereignisbenachrichtigungen**: Ein Sensor (Socket) sendet unaufgefordert eine Messwertänderung (`IND`) an die Steuerung (Plug), die daraufhin eine Quittung (`RSP`) zurücksendet.
- **Parametrierung und Konfiguration**: Der Adapter eignet sich für den Austausch von Konfigurationsdaten zwischen Engineering-Tool und Laufzeitsystem, wenn eine asynchrone, aber zuverlässige Übertragung gefordert ist.
- **Bildung von Subsystemen**: Innerhalb von Subapplikationen kann der Adapter als klar definierte Schnittstelle zwischen Modulen verwendet werden, um lose Kopplung und Wiederverwendbarkeit zu erhöhen.

Vergleich mit ähnlichen Bausteinen

Adapter	Ereignisse	Nutzlast	Zweck
`EVENT_HS`	REQ, CNF, IND, RSP	–	Reines Handshake-Muster ohne Daten, ideal für einfache Synchronisation.
`EVENT_HS_WSTRING`	REQ, CNF, IND, RSP	`WSTRING` pro Ereignis	Datenhandshake für Text-basierte Nachrichten (z. B. Befehle, Status).
`PUBLISH/SUBSCRIBE`-Adapter	Unterschiedlich	Beliebig	Publish-Subscribe-Muster, eher für unidirektionale Kommunikation.

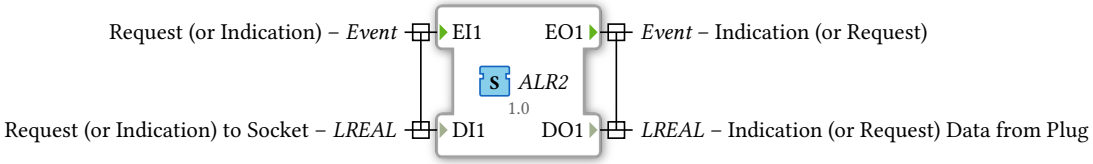
Der Vorteil von `EVENT_HS_WSTRING` gegenüber `EVENT_HS` ist die direkte Mitnahme von Informationen in beiden Richtungen, ohne zusätzliche Datenkanäle aufbauen zu müssen. Gegenüber generischen Publish/Subscribe-Mechanismen bietet er die Vorteile eines strengen Handshake-Protokolls (Request-Confirm und Indication-Response), was für viele Steuerungsanwendungen robuster ist.

Fazit

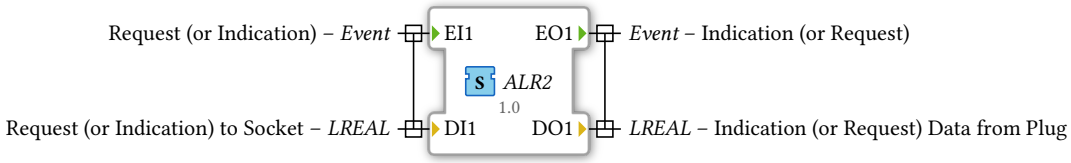
Der Adapter `EVENT_HS_WSTRING` stellt eine praxisgerechte Erweiterung des Handshake-Musters dar. Durch die Integration einer `WSTRING`-Nutzlast in jedes Ereignis wird die ursprünglich reine Synchronisationslogik zu einem vollwertigen Kommunikationsmittel für textuelle Befehle, Statusmeldungen oder Konfigurationsdaten. Die klare Trennung zwischen Plug und Socket sowie die beiden sauber definierten Service-Sequenzen machen den Adapter zu einer zuverlässigen Wahl für serviceorientierte Architekturen in IEC 61499-Anwendungen. Dank der eindeutigen Versionierung und der dokumentierten Herkunft ist er zudem gut nachvollziehbar und wartbar. Für den Einsatz in Projekten, die eine reine Datenübertragung ohne Handshake-Anforderungen benötigen, sollte man aber auf andere Muster wie Publish/Subscribe zurückgreifen. Insgesamt bietet `EVENT_HS_WSTRING` einen guten Kompromiss zwischen Einfachheit und Funktionsumfang.

REAL

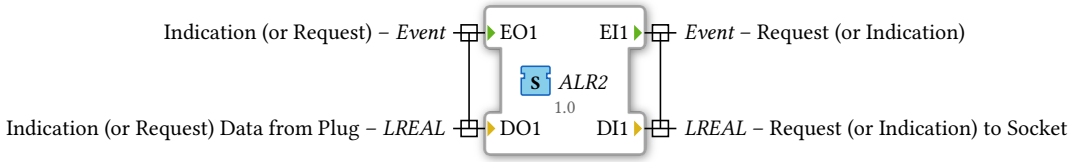
ALR2



Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Adapter-Typ **ALR2** stellt eine bidirektionale Schnittstelle zwischen zwei Funktionsblöcken bereit. Er überträgt genau ein Ereignis zusammen mit einem LREAL-Wert in beide Richtungen. Die Bezeichnung ‚ALR2' steht für ‚Adapter LREAL 2-Wege'. Der Baustein ist als generischer Steckverbinder (Plug/Socket) innerhalb der 4diac-IDE konzipiert und ermöglicht eine saubere, typsichere Kopplung von Komponenten.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar	mit Daten
EI1	Event	Anforderung (oder Indikation)	DI1

Ereignis-Ausgänge

Name	Typ	Kommentar	mit Daten
EO1	Event	Indikation (oder Anforderung)	DO1

Daten-Eingänge

Name	Datentyp	Kommentar
DI1	LREAL	Anforderung (oder Indikation) an Socket

Daten-Ausgänge

Name	Datentyp	Kommentar
DO1	LREAL	Indikation (oder Anforderung) Daten vom Plug

Adapter

Keine definiert.

Funktionsweise

Der ALR2-Adapter arbeitet ereignisgesteuert und bidirektional:

- Wird der Ereigniseingang **EI1** ausgelöst, so wird der aktuell an **DI1** anliegende LREAL-Wert zum gegenüberliegenden Adapterende übertragen. Dort erscheint das Ereignis **EO1** zusammen mit dem Wert auf **DO1**.
- In der Gegenrichtung gilt das Gleiche: Triggert der verbundene Gegenadapter sein Ereignis, so empfängt dieser ALR2 das Ereignis an **EI1** und stellt den empfangenen Wert an **DI1** bereit.

Die Kommentare „Anforderung (oder Indikation)“ und „Indikation (oder Anforderung)“ verdeutlichen, dass der Adapter je nach Einbausituation (Plug oder Socket) die Rolle eines anfordernden oder eines anzeigenden Kanals einnehmen kann.

Technische Besonderheiten

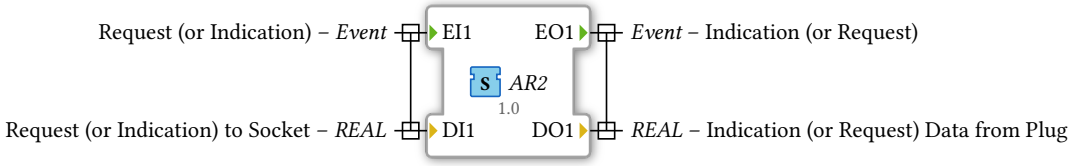
- **Typisierte Datenübertragung:** Es wird ausschließlich der Datentyp **LREAL** (64-Bit-Gleitkommazahl) unterstützt.
- **Bidirektionalität:** Ein einziger Adapter realisiert den Datenaustausch in beide Richtungen.
- **Compiler-Information:** Das Paket ist unter `adapter::types::bidirectional` abgelegt.

Fazit

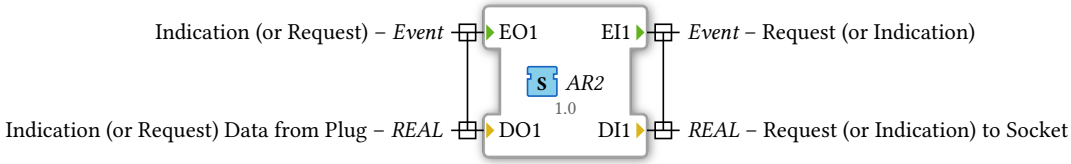
Der ALR2-Adapter ist ein eleganter Baustein für die bidirektionale Übertragung eines LREAL-Werts mit zugehörigem Ereignis.

AR2

Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Adapter-Typ **AR2** stellt eine bidirektionale Schnittstelle zwischen zwei Funktionsblöcken bereit. Er überträgt genau ein Ereignis zusammen mit einem REAL-Wert in beide Richtungen. Die Bezeichnung „AR2“ steht dabei für „Adapter Real 2-Wege“. Der Baustein ist als generischer Steckverbinder (Plug/Socket) innerhalb der 4diac-IDE konzipiert und ermöglicht eine saubere, typsichere Kopplung von Komponenten.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar	mit Daten
EI1	Event	Request (or Indication)	DI1

Ereignis-Ausgänge

Name	Typ	Kommentar	mit Daten
EO1	Event	Indication (or Request)	DO1

Daten-Eingänge

Name	Datentyp	Kommentar
DI1	REAL	Request (or Indication) to Socket

Daten-Ausgänge

Name	Datentyp	Kommentar
DO1	REAL	Indication (or Request) Data from Plug

Adapter

Keine definiert.

Funktionsweise

Der AR2-Adapter arbeitet ereignisgesteuert und bidirektional:

- Wird der Ereigniseingang **EI1** ausgelöst, so wird der aktuell an **DI1** anliegende REAL-Wert zum gegenüberliegenden Adapterende übertragen. Dort erscheint das Ereignis **EO1** zusammen mit dem Wert auf **DO1**.
- In der Gegenrichtung gilt das Gleiche: Triggert der verbundene Gegenadapter sein Ereignis, so empfängt dieser AR2 das Ereignis an **EI1** und stellt den empfangenen Wert an **DI1** bereit.

Die Kommentare „Request (or Indication)“ und „Indication (or Request)“ verdeutlichen, dass der Adapter je nach Einbausituation (Plug oder Socket) die Rolle eines anfordernden oder eines anzeigenden Kanals einnehmen kann. Somit eignet sich AR2 sowohl für Master/Slave- als auch für Peer-to-Peer-Kommunikation.

Technische Besonderheiten

- **Typisierte Datenübertragung:** Es wird ausschließlich der Datentyp **REAL** unterstützt, was eine klare Semantik für physikalische Messwerte oder Steuergrößen bietet.
- **Bidirektionalität:** Ein einziger Adapter realisiert den Datenaustausch in beide Richtungen, was die Anzahl der benötigten Schnittstellen reduziert.
- **Versionierung:** Der Baustein liegt in Version 1.0 vor, erstellt von der HR Agrartechnik GmbH (Copyright 2026, EPL 2.0).
- **Compiler-Information:** Das Paket ist unter `adapter::types::bidirectional` abgelegt und nutzt einen Typ-Hash (`eclipse4diac::core::TypeHash`) für Integritätsprüfungen.
- **Keine Zustandsmaschine:** Der Adapter selbst besitzt keine eigene Zustandslogik; er fungiert als reine Verbindung zwischen zwei Ereignis-/Datenpaaren.

Zustandsübersicht

Der AR2-Adapter implementiert keine explizite Zustandsmaschine. Der Datenfluss erfolgt ohne Verzögerung oder Zwischenspeicherung beim Auftreten der Ereignisse – die Signale werden direkt von Eingang zu Ausgang durchgereicht.

Anwendungsszenarien

- **Sensorwertübertragung:** Ein Funktionsblock erfasst einen analogen Sensorwert (z.B. Temperatur, Druck) und sendet diesen über AR2 an einen Auswerteblock. Gleichzeitig kann der Auswerteblock über die Gegenrichtung einen Sollwert zurückgeben.
- **Regelungskopplung:** In einer modularen Steuerung verbindet AR2 einen PID-Regler mit einem Stellglied. Der Regler sendet den Stellwert (REAL) und empfängt gleichzeitig die aktuelle Regelabweichung.
- **Parametrierung:** Über AR2 können Konfigurationsparameter (z.B. Grenzwerte) zwischen einem Bediengerät und der Steuerungslogik ausgetauscht werden.

Vergleich mit ähnlichen Bausteinen

- **AR1 (unidirektional):** Ein Adapter, der Daten nur in eine Richtung überträgt. AR2 hingegen ermöglicht bidirektionalen Austausch, was den Verkabelungsaufwand reduziert.
- **AE2 (Adapter Event 2-Wege):** Überträgt zwei Ereignisse, aber keine Daten. AR2 kombiniert ein Ereignis mit einem REAL-Wert – ideal für wertbasierte Kommunikation.
- **AR4 (Adapter Real 4-Wege):** Bietet vier voneinander unabhängige REAL-Kanäle. AR2 ist kompakter, wenn nur ein Wert benötigt wird.

Fazit

Der AR2-Adapter ist ein eleganter Baustein für die bidirektionale Übertragung eines REAL-Werts mit zugehörigem Ereignis. Seine klare Schnittstelle und die einfache Handhabung machen ihn besonders

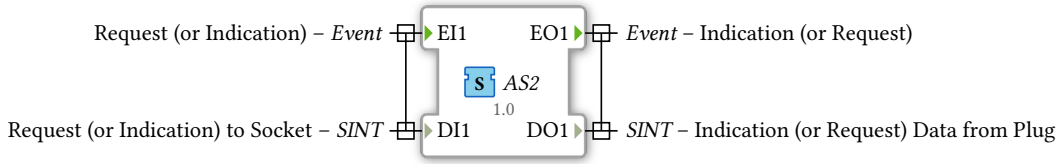
geeignet für modulare Automatisierungslösungen, bei denen Messdaten oder Steuergrößen zwischen Funktionsblöcken ausgetauscht werden müssen. Durch die Einhaltung des 61499-Standards und die Lizenz unter EPL 2.0 ist AR2 offen und erweiterbar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

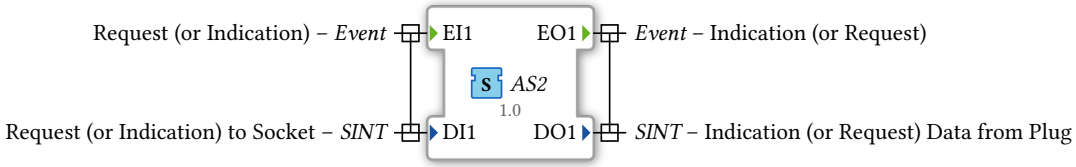
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SINT

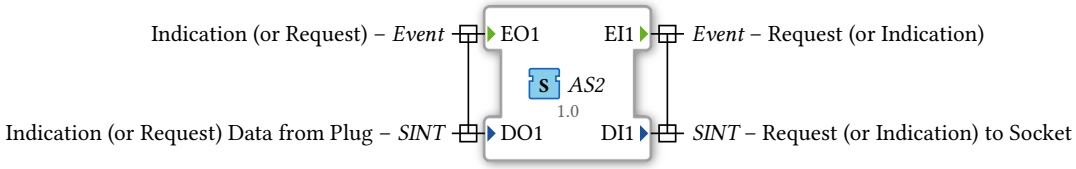
AS2



Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Adapter-Typ **AS2** stellt eine bidirektionale Schnittstelle zwischen zwei Funktionsblöcken bereit. Er überträgt genau ein Ereignis zusammen mit einem SINT-Wert in beide Richtungen. Die Bezeichnung ‚AS2' steht für ‚Adapter SINT 2-Wege'. Der Baustein ist als generischer Steckverbinder (Plug/Socket) innerhalb der 4diac-IDE konzipiert und ermöglicht eine saubere, typsichere Kopplung von Komponenten.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar	mit Daten
EI1	Event	Anforderung (oder Indikation)	DI1

Ereignis-Ausgänge

Name	Typ	Kommentar	mit Daten
EO1	Event	Indikation (oder Anforderung)	DO1

Daten-Eingänge

Name	Datentyp	Kommentar
DI1	SINT	Anforderung (oder Indikation) an Socket

Daten-Ausgänge

Name	Datentyp	Kommentar
DO1	SINT	Indikation (oder Anforderung) Daten vom Plug

Adapter

Keine definiert.

Funktionsweise

Der AS2-Adapter arbeitet ereignisgesteuert und bidirektional:

- Wird der Ereigniseingang **EI1** ausgelöst, so wird der aktuell an **DI1** anliegende SINT-Wert zum gegenüberliegenden Adapterende übertragen. Dort erscheint das Ereignis **EO1** zusammen mit dem Wert auf **DO1**.
- In der Gegenrichtung gilt das Gleiche: Triggert der verbundene Gegenadapter sein Ereignis, so empfängt dieser AS2 das Ereignis an **EI1** und stellt den empfangenen Wert an **DI1** bereit.

Die Kommentare „Anforderung (oder Indikation)“ und „Indikation (oder Anforderung)“ verdeutlichen, dass der Adapter je nach Einbausituation (Plug oder Socket) die Rolle eines anfordernden oder eines anzeigenden Kanals einnehmen kann.

Technische Besonderheiten

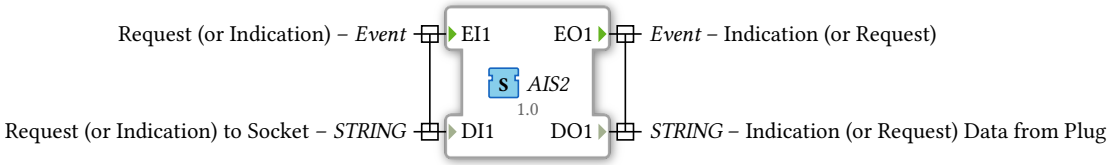
- **Typisierte Datenübertragung:** Es wird ausschließlich der Datentyp **SINT** (vorzeichenbehaftete 8-Bit-Ganzzahl) unterstützt.
- **Bidirektionalität:** Ein einziger Adapter realisiert den Datenaustausch in beide Richtungen.
- **Compiler-Information:** Das Paket ist unter `adapter::types::bidirectional` abgelegt.

Fazit

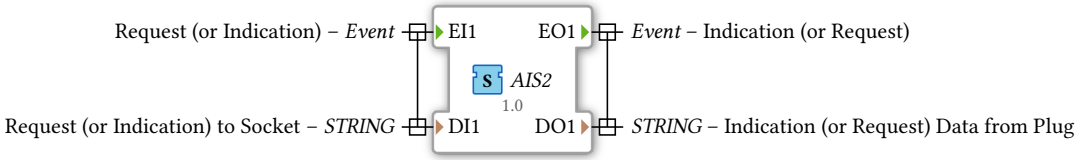
Der AS2-Adapter ist ein eleganter Baustein für die bidirektionale Übertragung eines SINT-Werts mit zugehörigem Ereignis.

STRING

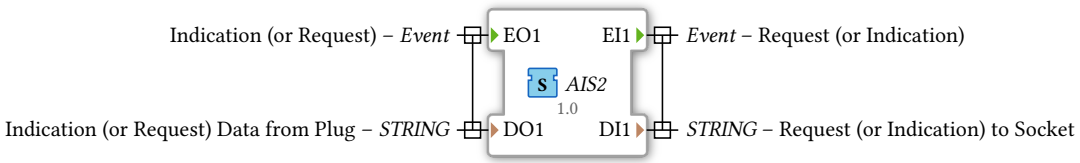
AIS2



Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Adapter-Typ **AIS2** stellt eine bidirektionale Schnittstelle zwischen zwei Funktionsblöcken bereit. Er überträgt genau ein Ereignis zusammen mit einem STRING-Wert in beide Richtungen. Die Bezeichnung ‚AIS2' steht für ‚Adapter STRING 2-Wege'. Der Baustein ist als generischer Steckverbinder (Plug/Socket) innerhalb der 4diac-IDE konzipiert und ermöglicht eine saubere, typsichere Kopplung von Komponenten.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar	mit Daten
EI1	Event	Anforderung (oder Indikation)	DI1

Ereignis-Ausgänge

Name	Typ	Kommentar	mit Daten
EO1	Event	Indikation (oder Anforderung)	DO1

Daten-Eingänge

Name	Datentyp	Kommentar
DI1	STRING	Anforderung (oder Indikation) an Socket

Daten-Ausgänge

Name	Datentyp	Kommentar
DO1	STRING	Indikation (oder Anforderung) Daten vom Plug

Adapter

Keine definiert.

Funktionsweise

Der AIS2-Adapter arbeitet ereignisgesteuert und bidirektional:

- Wird der Ereigniseingang **EI1** ausgelöst, so wird der aktuell an **DI1** anliegende STRING-Wert zum gegenüberliegenden Adapterende übertragen. Dort erscheint das Ereignis **EO1** zusammen mit dem Wert auf **DO1**.
- In der Gegenrichtung gilt das Gleiche: Triggert der verbundene Gegenadapter sein Ereignis, so empfängt dieser AIS2 das Ereignis an **EI1** und stellt den empfangenen Wert an **DI1** bereit.

Die Kommentare „Anforderung (oder Indikation)" und „Indikation (oder Anforderung)" verdeutlichen, dass der Adapter je nach Einbausituation (Plug oder Socket) die Rolle eines anfordernden oder eines anzeigenden Kanals einnehmen kann.

Technische Besonderheiten

- **Typisierte Datenübertragung:** Es wird ausschließlich der Datentyp **STRING** (Zeichenkette) unterstützt.
- **Bidirektionalität:** Ein einziger Adapter realisiert den Datenaustausch in beide Richtungen.
- **Compiler-Information:** Das Paket ist unter `adapter::types::bidirectional` abgelegt.

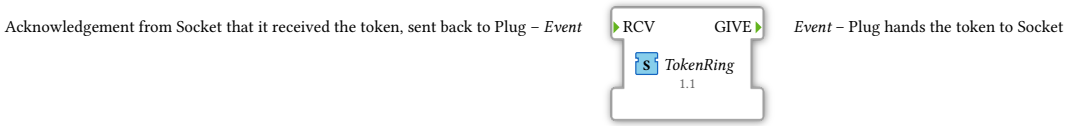
Fazit

Der AIS2-Adapter ist ein eleganter Baustein für die bidirektionale Übertragung eines STRING-Werts mit zugehörigem Ereignis.

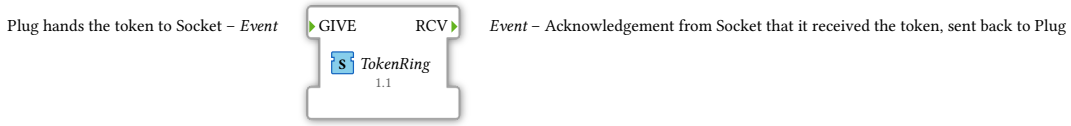
TokenRing

TokenRing

Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



• – ○ • – ○ • – ○ •

Einleitung

Der TokenRing-Adapter implementiert das Token-Ring-Entwurfsmuster zur gegenseitigen Ausschließung (Mutual Exclusion) in verteilten Systemen. Er ist datenlos und dient ausschließlich zur Übergabe eines Tokens zwischen zwei benachbarten Controllern in einem Ring. Nur der aktuelle Token-Inhaber darf auf die geschützte Ressource zugreifen; nach Abschluss gibt er das Token an seinen Nachbarn weiter.

Schnittstellenstruktur

Der Adapter besitzt zwei Ereignisse, keine Daten oder Unter-Adapter.

Ereignis-Eingänge

- **RCV**: Ereignis, das vom Socket (Empfänger) an den Plug (Geber) gesendet wird, um den Empfang des Tokens zu bestätigen.

Ereignis-Ausgänge

- **GIVE**: Ereignis, das vom Plug (Geber) an den Socket (Empfänger) gesendet wird, um das Token zu übergeben.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Der TokenRing-Adapter ist selbst ein bidirektionaler Adapter mit zwei Rollen: Plug (rechte Seite, auch als `MTXOUT` bekannt) und Socket (linke Seite, auch als `MTXIN` bekannt). Er enthält keine weiteren Adapter.

Funktionsweise

Der TokenRing-Adapter implementiert eine einfache Handshake-Sequenz: Der Plug sendet das GIVE-Ereignis an den Socket. Sobald der Socket das Token empfangen hat, antwortet er mit dem RCV-Ereignis an den Plug. Damit ist die Übergabe abgeschlossen. Ein Controller, der in einen Ring eingebunden ist, benötigt zwei Instanzen dieses Adapters: eine als Plug (MTXOUT) für die Verbindung zum nachgeschalteten Nachbarn und eine als Socket (MTXIN) für die Verbindung zum vorgeschalteten Nachbarn. Auf diese Weise wird ein Token im Ring weitergegeben und die gegenseitige Ausschließung sichergestellt.

Technische Besonderheiten

- Der Adapter ist **datenlos**, d.h. er überträgt keine Nutzdaten, sondern nur Steuersignale.
- Die Ereignisreihenfolge ist durch einen Service-Contract festgelegt: `GIVE` vom Plug zum Socket, gefolgt von `RCV` vom Socket zum Plug.
- Die Rollen sind klar getrennt: Der Plug (MTXOUT) ist der aktive Geber, der Socket (MTXIN) der passive Empfänger.
- Der Adapter ist für eine Ringtopologie konzipiert; jeder Knoten besitzt genau eine Plug- und eine Socket-Instanz.

Zustandsübersicht

Da es sich um einen reinen Ereignisaustausch ohne interne Zustände handelt, gibt es keinen expliziten Zustandsautomaten. Der Ablauf ist strikt sequenziell und wird durch den Service-Contract `token_pass` definiert:

1. Senden von `GIVE` (Plug ⟶ Socket)
2. Empfangen von `RCV` (Socket ⟶ Plug)

Anwendungsszenarien

- **Verteilte Steuerungssysteme**: Mehrere Steuerungen müssen exklusiv auf eine gemeinsame Ressource (z.B. eine Maschinenachse, einen Bus) zugreifen.
- **Ringförmige Netzwerktopologien**: Sicherung der gegenseitigen Ausschließung ohne zentrale Instanz.
- **IEC-61499-basierte Automatisierungssysteme**: Verwendung des Token-Ring-Musters zur Koordination paralleler Funktionsbausteine.

Vergleich mit ähnlichen Bausteinen

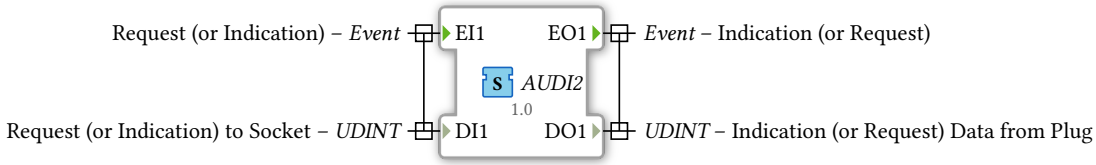
- **Mutual Exclusion (Mutex) Baustein**: Oft als zentrale Semaphore realisiert; der TokenRing-Adapter verteilt die Kontrolle dezentral über eine Ringstruktur.
- **Semaphore-Adapter**: Üblicherweise mit Zähler und Warteschlange; der TokenRing-Adapter arbeitet ohne Zähler und ohne Warteschlange – es gibt immer nur ein Token.
- **Analogie zum Token-Bus**: Der TokenRing-Adapter ist eine logische Ringstruktur, die auf der IEC-61499-Adaptermechanik aufbaut.

Fazit

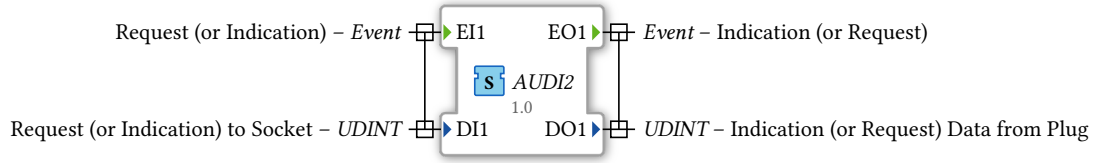
Der TokenRing-Adapter ist ein leichtgewichtiges, effektives Mittel zur Implementierung gegenseitiger Ausschließung in verteilten IEC-61499-Systemen. Durch seine einfache Ereignisschnittstelle und die klare Rollentrennung lässt er sich ohne zusätzliche Datenübertragung in Ringtopologien integrieren. Die Verwendung von zwei Adapterinstanzen pro Knoten gewährleistet die korrekte Token-Weitergabe und verhindert gleichzeitigen Ressourcenzugriff.

UDINT

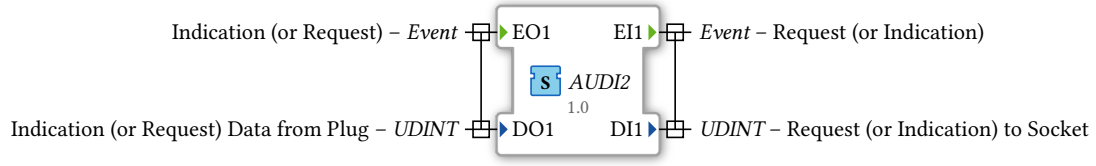
AUDI2



Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Adapter-Typ **AUDI2** stellt eine bidirektionale Schnittstelle zwischen zwei Funktionsblöcken bereit. Er überträgt genau ein Ereignis zusammen mit einem UDINT-Wert in beide Richtungen. Die Bezeichnung ‚AUDI2' steht für ‚Adapter UDINT 2-Wege'. Der Baustein ist als generischer Steckverbinder (Plug/Socket) innerhalb der 4diac-IDE konzipiert und ermöglicht eine saubere, typsichere Kopplung von Komponenten.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar	mit Daten
EI1	Event	Anforderung (oder Indikation)	DI1

Ereignis-Ausgänge

Name	Typ	Kommentar	mit Daten
EO1	Event	Indikation (oder Anforderung)	DO1

Daten-Eingänge

Name	Datentyp	Kommentar
DI1	UDINT	Anforderung (oder Indikation) an Socket

Daten-Ausgänge

Name	Datentyp	Kommentar
DO1	UDINT	Indikation (oder Anforderung) Daten vom Plug

Adapter

Keine definiert.

Funktionsweise

Der AUDI2-Adapter arbeitet ereignisgesteuert und bidirektional:

- Wird der Ereigniseingang **EI1** ausgelöst, so wird der aktuell an **DI1** anliegende UDINT-Wert zum gegenüberliegenden Adapterende übertragen. Dort erscheint das Ereignis **EO1** zusammen mit dem Wert auf **DO1**.
- In der Gegenrichtung gilt das Gleiche: Triggert der verbundene Gegenadapter sein Ereignis, so empfängt dieser AUDI2 das Ereignis an **EI1** und stellt den empfangenen Wert an **DI1** bereit.

Die Kommentare „Anforderung (oder Indikation)" und „Indikation (oder Anforderung)" verdeutlichen, dass der Adapter je nach Einbausituation (Plug oder Socket) die Rolle eines anfordernden oder eines anzeigenden Kanals einnehmen kann.

Technische Besonderheiten

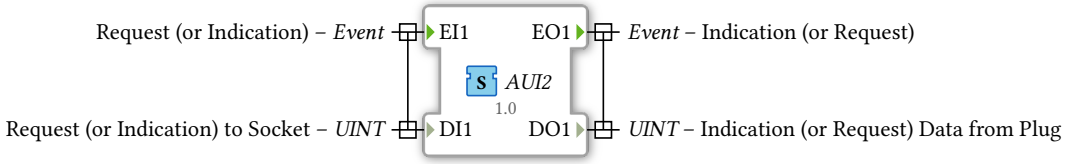
- **Typisierte Datenübertragung:** Es wird ausschließlich der Datentyp **UDINT** (vorzeichenlose 32-Bit-Ganzzahl) unterstützt.
- **Bidirektionalität:** Ein einziger Adapter realisiert den Datenaustausch in beide Richtungen.
- **Compiler-Information:** Das Paket ist unter `adapter::types::bidirectional` abgelegt.

Fazit

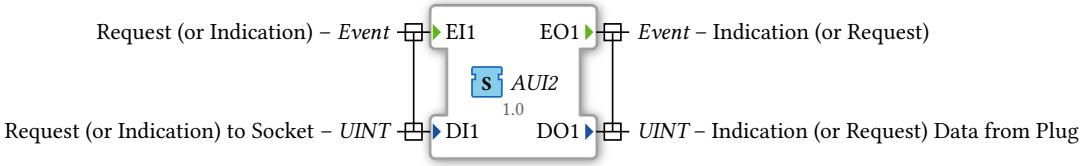
Der AUDI2-Adapter ist ein eleganter Baustein für die bidirektionale Übertragung eines UDINT-Werts mit zugehörigem Ereignis.

UINT

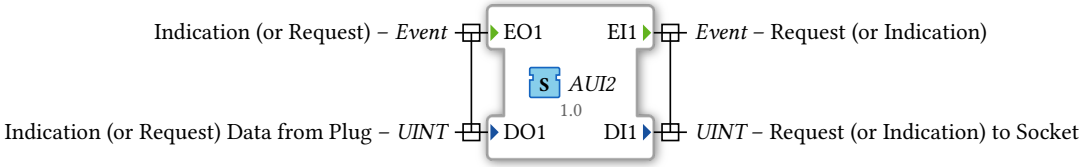
AUI2



Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Adapter-Typ **AUI2** stellt eine bidirektionale Schnittstelle zwischen zwei Funktionsblöcken bereit. Er überträgt genau ein Ereignis zusammen mit einem UINT-Wert in beide Richtungen. Die Bezeichnung ‚AUI2' steht für ‚Adapter UINT 2-Wege'. Der Baustein ist als generischer Steckverbinder (Plug/Socket) innerhalb der 4diac-IDE konzipiert und ermöglicht eine saubere, typsichere Kopplung von Komponenten.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar	mit Daten
EI1	Event	Anforderung (oder Indikation)	DI1

Ereignis-Ausgänge

Name	Typ	Kommentar	mit Daten
EO1	Event	Indikation (oder Anforderung)	DO1

Daten-Eingänge

Name	Datentyp	Kommentar
DI1	UINT	Anforderung (oder Indikation) an Socket

Daten-Ausgänge

Name	Datentyp	Kommentar
DO1	UINT	Indikation (oder Anforderung) Daten vom Plug

Adapter

Keine definiert.

Funktionsweise

Der AUI2-Adapter arbeitet ereignisgesteuert und bidirektional:

- Wird der Ereigniseingang **EI1** ausgelöst, so wird der aktuell an **DI1** anliegende UINT-Wert zum gegenüberliegenden Adapterende übertragen. Dort erscheint das Ereignis **EO1** zusammen mit dem Wert auf **DO1**.
- In der Gegenrichtung gilt das Gleiche: Triggert der verbundene Gegenadapter sein Ereignis, so empfängt dieser AUI2 das Ereignis an **EI1** und stellt den empfangenen Wert an **DI1** bereit.

Die Kommentare „Anforderung (oder Indikation)“ und „Indikation (oder Anforderung)“ verdeutlichen, dass der Adapter je nach Einbausituation (Plug oder Socket) die Rolle eines anfordernden oder eines anzeigenden Kanals einnehmen kann.

Technische Besonderheiten

- **Typisierte Datenübertragung:** Es wird ausschließlich der Datentyp **UINT** (vorzeichenlose 16-Bit-Ganzzahl) unterstützt.
- **Bidirektionalität:** Ein einziger Adapter realisiert den Datenaustausch in beide Richtungen.
- **Compiler-Information:** Das Paket ist unter `adapter::types::bidirectional` abgelegt.

Fazit

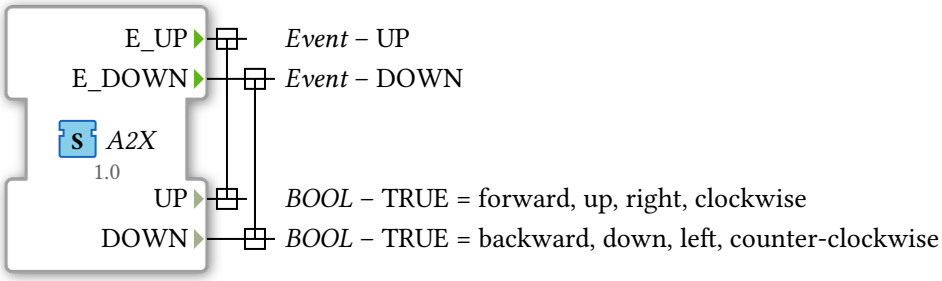
Der AUI2-Adapter ist ein eleganter Baustein für die bidirektionale Übertragung eines UINT-Werts mit zugehörigem Ereignis.

Unidirectional Adapters

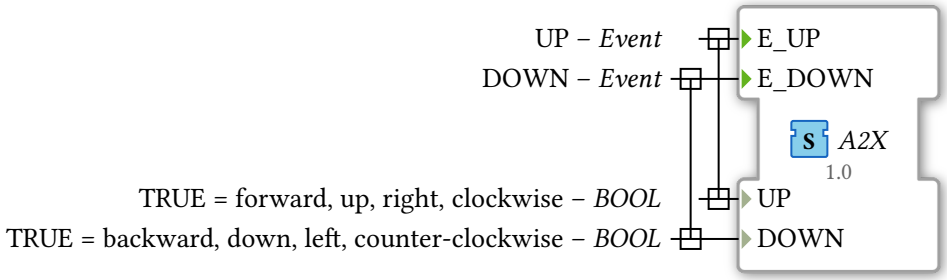
BOOL

A2X (BOOL)

Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



unidirectional Adapter Interface for 2 Events and 2 Bools

Interface

Events

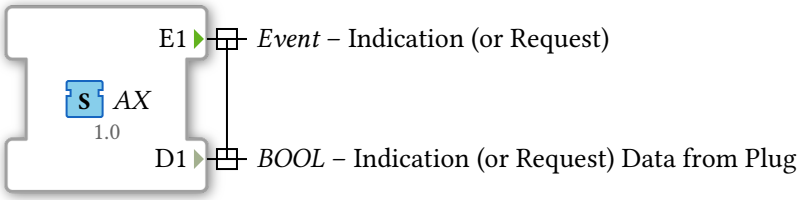
Name	Comment	With
E_UP	UP	UP
E_DOWN	DOWN	DOWN

Data

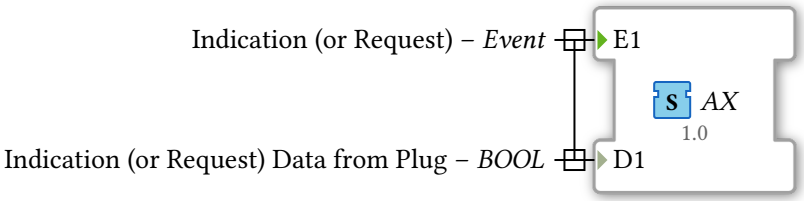
Name	Type	Comment
UP	BOOL	TRUE = forward, up, right, clockwise
DOWN	BOOL	TRUE = backward, down, left, counter-clockwise

AX (BOOL)

Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



🎧 Podcast

- Eclipse 4diac: Wie Open Source Forschung und Praxis in der industriellen Automatisierung verbindet
- Das Working Set Objekt: Das Gehirn der ISOBUS-Bedienoberfläche verstehen – Von der Norm zur Praxis im ISO-Designer
- Code-Renovierung mit AX-Adaptern: Wie Eclipse 4diac™ durch Signal-Bündelung Komplexität besiegt
- Von 1400 Fehlern zum sauberen Code: Die Migration der „Getreidehacke“ auf Eclipse 4diac™ 3.0 und die Macht der AX-Adapter
- Agrar-Revolution 1883: Wie Max Eyth Englands Landwirtschaft modernisierte

Interface

Events

Name	Comment	With
E1	Indication (or Request)	D1

Data

Name	Type	Comment
D1	BOOL	Indication (or Request) Data from Plug

🛠️ Zugehörige Übungen

- Uebung_003b2_AX
- Uebung_003b3_AX
- Uebung_003c_AX
- Uebung_006a3_AX
- Uebung_006a3_sub_AX
- Uebung_006a4_AX
- Uebung_010b4_AX
- Uebung_010b5_AX
- Uebung_010c2_AX

- Uebung_010c3_AX
- AuxIXA_TO_logiBUS_QXA_BG
- Uebung_010c4_AX
- Uebung_010c_AX
- AX_SPLIT_TO_2x_AX_TP
- Uebung_035a1_AX
- Uebung_035a1b_AX
- Uebung_035a2_AX
- Uebung_035a3_AX
- Uebung_038_AX
- Uebung_040_AX

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Unidirectional BOOL Adapter

🎧 Podcast

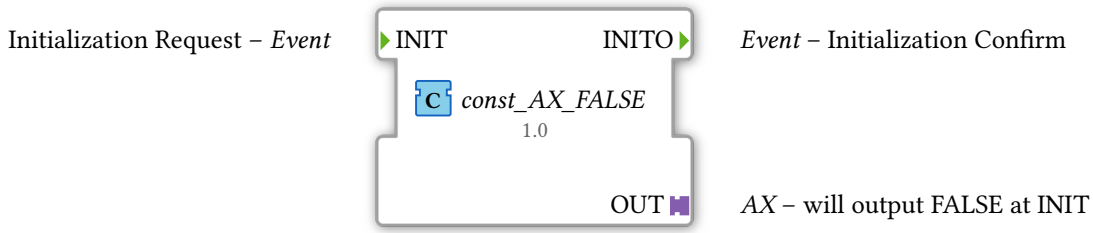
- George Boole: Wie ein Autodidakt mit 0 und 1 das Fundament unserer digitalen Welt legte

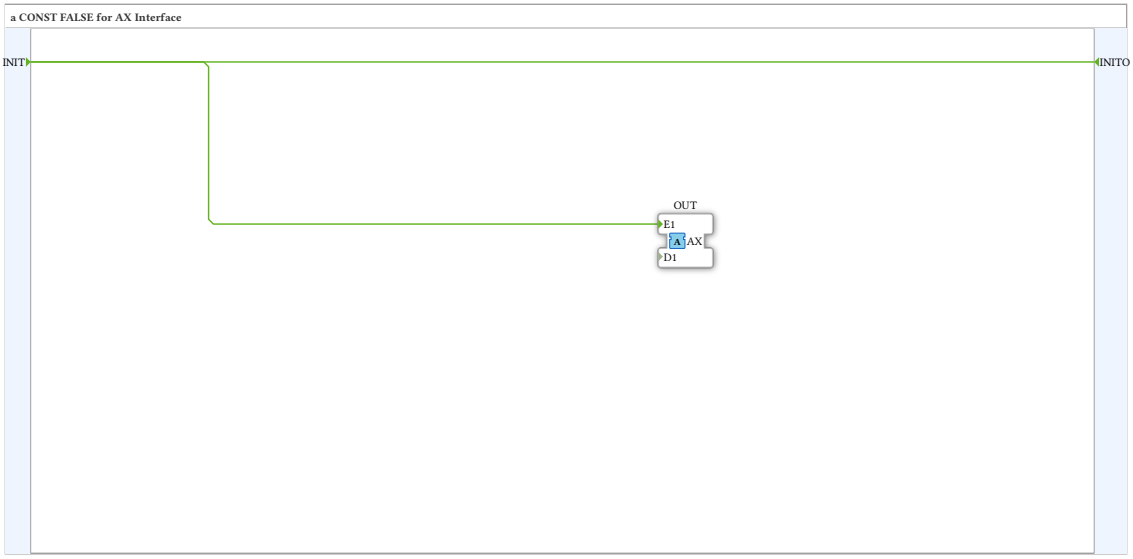
🛠️ Zugehörige Übungen

- Uebung_001_AX_b
- Uebung_002a2_AX
- Uebung_002a4
- Uebung_002a5b
- Uebung_002b3
- Uebung_004b_AX
- Uebung_006a3
- LinksRechts_T_FF_Event
- Uebung_035a2
- Uebung_035a3
- SoftkeyOverride_TO_QX_BG
- Uebung_055
- Uebung_056
- Bool8_TO_logiBUS_QX8
- Uebung_094a_AX
- Uebung_160

constants

const_AX_FALSE





• – ◦ • – ◦ • – ◦ •

Einleitung

`const_AX_FALSE` ist das Gegenstück zu const_AX_TRUE: Ein einfacher Konstanten-Baustein, der bei seiner Initialisierung den festen Wert `FALSE` auf einen AX-Adapter-Plug legt. Er dient dazu, statisch verdrahtete boolesche Konstanten in ein AX-Adapternetzwerk einzuspeisen, ohne dass dafür eine eigene Verdrahtung von Ereignis- und Datenleitung nötig ist.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung. Löst die Ausgabe des Konstantwerts über den Plug `OUT` aus.

Ereignis-Ausgänge

- **INITO**: Bestätigt den Abschluss der Initialisierung.

Daten-Eingänge

- Keine. Der Wert `FALSE` ist fest im Baustein verdrahtet (`Parameter Name="D1" Value="FALSE"` auf dem Plug `OUT`).

Daten-Ausgänge

- Keine direkten Datenausgänge; der Wert wird ausschließlich über den Adapter `OUT` bereitgestellt.

Adapter

- **OUT** (Plug, Typ `AX`): gibt bei `INIT` den festen Wert `FALSE` über sein `E1`/`D1`-Adapterereignis aus.

Funktionsweise

Bei Eintreffen des Ereignisses `INIT` löst `const_AX_FALSE` intern zwei Verbindungen aus: Zum einen wird `INIT` direkt an `OUT.E1` weitergereicht – der `OUT`-Plug feuert damit sein Adapterereignis mit dem fest parametrierten Datenwert `D1 = FALSE`. Zum anderen wird `INIT` an `INITO` weitergereicht, wodurch der Aufrufer den Abschluss der Initialisierung bestätigt bekommt.

Technische Besonderheiten

- Der Wert `FALSE` ist als `Parameter`-Attribut direkt am `AdapterDeclaration`-Element von `OUT` hinterlegt, nicht als Laufzeitdatum – der Baustein besitzt keine Möglichkeit, zur Laufzeit einen anderen Wert auszugeben. Wird ein zur Laufzeit konfigurierbarer Startwert benötigt, ist stattdessen initval_AX zu verwenden.
- Reine Verdrahtungslogik ohne eigene ECC/Zustandsmaschine (Composite-FB ohne innere Funktionsblöcke).

Zustandsübersicht

Der Baustein ist zustandslos: Jedes `INIT`-Ereignis führt unmittelbar zur Ausgabe von `FALSE` auf `OUT` und zur Bestätigung über `INITO`.

Anwendungsszenarien

- Initialisierung von AX-Adapterverbindungen mit einem festen `FALSE`-Startwert, z. B. als Sperr- oder Disable-Konstante
- Testaufbauten, in denen ein AX-Eingang ohne echten Sensor/Vorgänger-Baustein mit einem festen Wert versorgt werden muss
- Ersatz für eine manuell verdrahtete Konstanten-Verbindung in generischen, AX-adapterbasierten Netzwerken

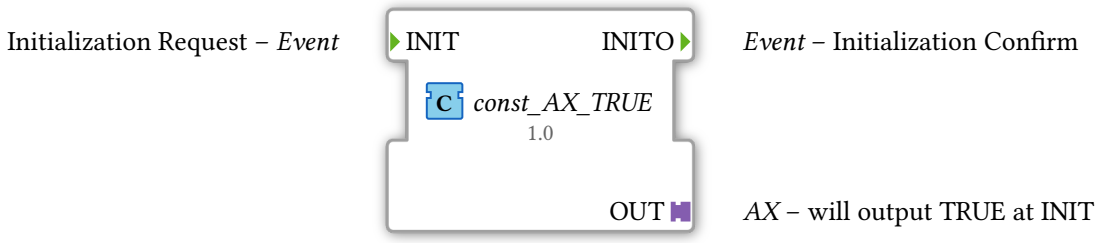
⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit const_AX_TRUE, der denselben Mechanismus für den Wert `TRUE` nutzt, sowie mit initval_AX, der denselben Zweck mit einem zur Laufzeit über `INIT_VAL` konfigurierbaren Wert erfüllt statt eines fest verdrahteten.

Fazit

`const_AX_FALSE` ist der einfachste Weg, einen AX-Adapter-Plug mit einer festen `FALSE`-Konstante zu initialisieren, und eignet sich überall dort, wo der Wert zur Entwurfszeit feststeht und keine Laufzeitkonfiguration erforderlich ist.

const_AX_TRUE



• – ◦ • – ◦ • – ◦ •

Einleitung

`const_AX_TRUE` ist ein einfacher Konstanten-Baustein, der bei seiner Initialisierung den festen Wert `TRUE` auf einen AX-Adapter-Plug legt. Er dient dazu, statisch verdrahtete boolesche Konstanten in ein AX-Adapternetzwerk einzuspeisen, ohne dass dafür eine eigene Verdrahtung von Ereignis- und Datenleitung nötig ist.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung. Löst die Ausgabe des Konstantwerts über den Plug `OUT` aus.

Ereignis-Ausgänge

- **INITO**: Bestätigt den Abschluss der Initialisierung.

Daten-Eingänge

- Keine. Der Wert `TRUE` ist fest im Baustein verdrahtet (`Parameter Name="D1" Value="TRUE"` auf dem Plug `OUT`).

Daten-Ausgänge

- Keine direkten Datenausgänge; der Wert wird ausschließlich über den Adapter `OUT` bereitgestellt.

Adapter

- **OUT** (Plug, Typ `AX`): gibt bei `INIT` den festen Wert `TRUE` über sein `E1`/`D1`-Adapterereignis aus.

Funktionsweise

Bei Eintreffen des Ereignisses `INIT` löst `const_AX_TRUE` intern zwei Verbindungen aus: Zum einen wird `INIT` direkt an `OUT.E1` weitergereicht – der `OUT`-Plug feuert damit sein Adapterereignis mit dem fest parametrierten Datenwert `D1 = TRUE`. Zum anderen wird `INIT` an `INITO` weitergereicht, wodurch der Aufrufer den Abschluss der Initialisierung bestätigt bekommt.

Technische Besonderheiten

- Der Wert `TRUE` ist als `Parameter`-Attribut direkt am `AdapterDeclaration`-Element von `OUT` hinterlegt, nicht als Laufzeitdatum – der Baustein besitzt keine Möglichkeit, zur Laufzeit einen anderen Wert auszugeben. Wird ein zur Laufzeit konfigurierbarer Startwert benötigt, ist stattdessen initval_AX zu verwenden.
- Reine Verdrahtungslogik ohne eigene ECC/Zustandsmaschine (Composite-FB ohne innere Funktionsblöcke).

Zustandsübersicht

Der Baustein ist zustandslos: Jedes `INIT`-Ereignis führt unmittelbar zur Ausgabe von `TRUE` auf `OUT` und zur Bestätigung über `INITO`.

Anwendungsszenarien

- Initialisierung von AX-Adapterverbindungen mit einem festen `TRUE`-Startwert, z. B. als Freigabe- oder Enable-Konstante
- Testaufbauten, in denen ein AX-Eingang ohne echten Sensor/Vorgänger-Baustein mit einem festen Wert versorgt werden muss
- Ersatz für eine manuell verdrahtete Konstanten-Verbindung in generischen, AX-adapterbasierten Netzwerken

⚖️ Vergleich mit ähnlichen Bausteinen

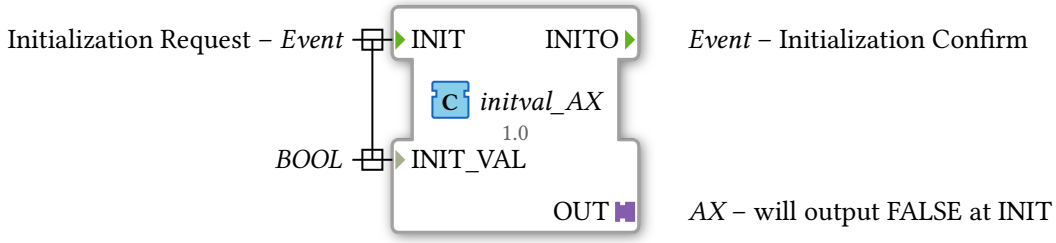
Vergleich mit const_AX_FALSE, der denselben Mechanismus für den Wert `FALSE` nutzt, sowie mit initval_AX, der denselben Zweck mit einem zur Laufzeit über `INIT_VAL` konfigurierbaren Wert erfüllt statt eines fest verdrahteten.

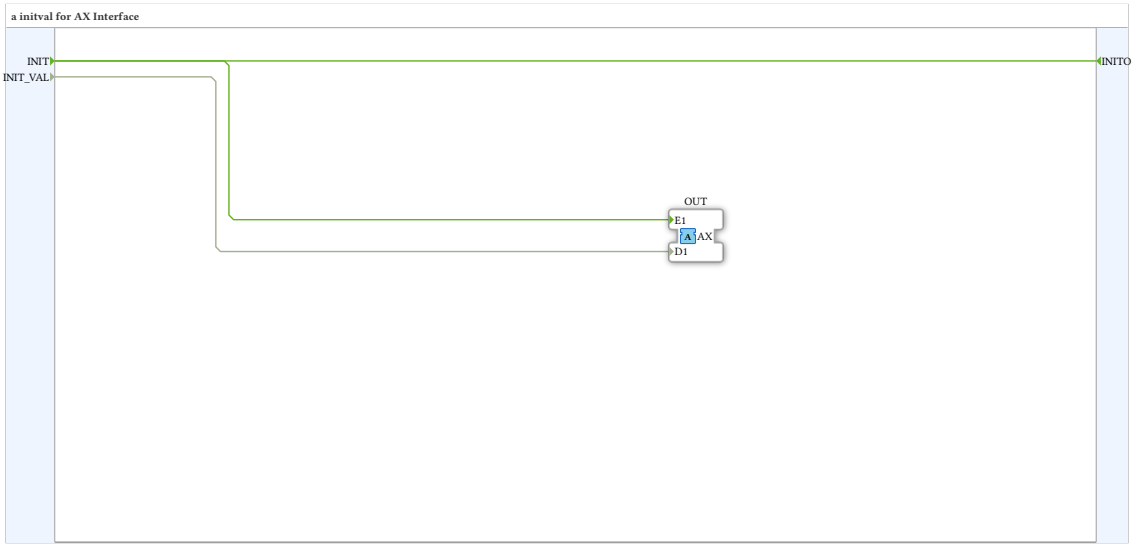
Fazit

`const_AX_TRUE` ist der einfachste Weg, einen AX-Adapter-Plug mit einer festen `TRUE`-Konstante zu initialisieren, und eignet sich überall dort, wo der Wert zur Entwurfszeit feststeht und keine Laufzeitkonfiguration erforderlich ist.

initval

initval_AX





• – ◦ • – ◦ • – ◦ •

Einleitung

`initval_AX` ist die konfigurierbare Variante der const_AX_TRUE/const_AX_FALSE-Bausteine: Statt eines fest verdrahteten Werts nimmt er bei der Initialisierung einen booleschen Startwert als Dateneingang entgegen und legt diesen auf einen AX-Adapter-Plug.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, liefert `INIT_VAL`. Löst die Ausgabe von `INIT_VAL` über den Plug `OUT` aus.

Ereignis-Ausgänge

- **INITO**: Bestätigt den Abschluss der Initialisierung.

Daten-Eingänge

- **INIT_VAL** (BOOL): Der Startwert, der beim Eintreffen von `INIT` auf den Adapter `OUT` geschrieben wird.

Daten-Ausgänge

- Keine direkten Datenausgänge; der Wert wird ausschließlich über den Adapter `OUT` bereitgestellt.

Adapter

- **OUT** (Plug, Typ `AX`): gibt bei `INIT` den über `INIT_VAL` übergebenen Wert über sein `E1`/`D1`-Adapterereignis aus.

Funktionsweise

Bei Eintreffen des Ereignisses `INIT` (mit `With Var="INIT_VAL"`) verknüpft `initval_AX` intern zwei Datenpfade: `INIT_VAL` wird per Datenverbindung auf `OUT.D1` kopiert, und `INIT` wird per Ereignisverbindung

an `OUT.E1` weitergereicht – der `OUT`-Plug feuert damit sein Adapterereignis mit dem gerade übergebenen Wert. Anschließend wird `INIT` zusätzlich an `INITO` weitergereicht, wodurch der Aufrufer den Abschluss der Initialisierung bestätigt bekommt.

Technische Besonderheiten

- Im Unterschied zu `const_AX_TRUE`/`const_AX_FALSE` ist der ausgegebene Wert nicht im Baustein selbst parametriert, sondern kommt bei jedem `INIT`-Aufruf frisch über `INIT_VAL` – der Baustein kann also auch außerhalb einer reinen Initialisierungsphase zum dynamischen Setzen eines AX-Werts verwendet werden.
- Reine Verdrahtungslogik ohne eigene ECC/Zustandsmaschine (Composite-FB ohne innere Funktionsblöcke); die Datenverbindung `INIT_VAL → OUT.D1` muss vor der Ereignisverbindung ausgewertet sein, damit `OUT.E1` den aktuellen Wert trägt – dies wird durch die 61499-Ausführungsreihenfolge der `FBNetwork`-Verbindungen sichergestellt und ist nicht separat zu verdrahten.

Zustandsübersicht

Der Baustein ist zustandslos: Jedes `INIT`-Ereignis führt unmittelbar zur Übernahme von `INIT_VAL` auf `OUT` und zur Bestätigung über `INITO`.

Anwendungsszenarien

- Initialisierung von AX-Adapterverbindungen mit einem projekt- oder parameterabhängigen Startwert (z. B. aus einer Konfigurationsvariable)
- Dynamisches Setzen eines AX-Werts zur Laufzeit, wenn `const_AX_TRUE`/`const_AX_FALSE` wegen des fest verdrahteten Werts nicht ausreichen
- Wiederverwendbare Bausteinvorlage für Testaufbauten, in denen unterschiedliche AX-Startwerte durchprobiert werden sollen

⚖ Vergleich mit ähnlichen Bausteinen

Vergleich mit const_AX_TRUE und const_AX_FALSE, die denselben Verdrahtungsmechanismus nutzen, den Ausgabewert jedoch fest im Baustein parametrieren statt ihn über einen Dateneingang entgegenzunehmen.

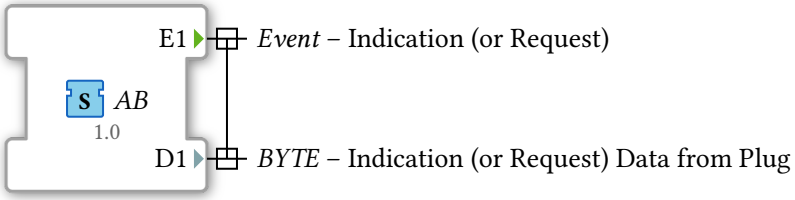
Fazit

`initval_AX` liefert dieselbe einfache Initialisierungslogik wie `const_AX_TRUE`/`const_AX_FALSE`, macht den ausgegebenen Wert aber über `INIT_VAL` frei konfigurierbar und eignet sich daher überall dort, wo der Startwert nicht bereits zur Entwurfszeit feststeht.

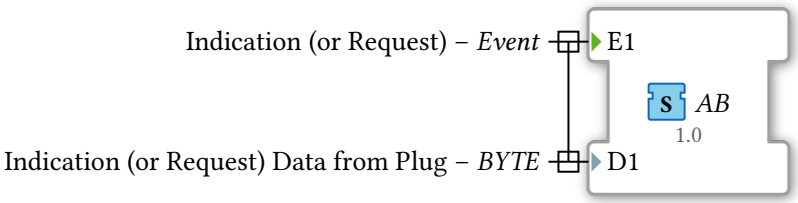
BYTE

AB (BYTE)

Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



🎧 Podcast

- From Pyramid to Plug-and-Play: The Rise of Self-Configurable Industrial Automation
- E_CTD: Ereignisgesteuerter Abwärtszähler nach IEC 61499
- E_DELAY in IEC 61499: Präzise, Abbrechbare Zeitverzögerung in Steuerungssystemen
- E_TON in der Industrieautomation: Wie ein simpler Timer Sicherheit und Stabilität schafft
- Ereignisbehandlung: Das ABC der Automatisierung und wie Systeme „denken“

📺 Video

- Charles Babbage

Interface

Events

Name	Comment	With
E1	Indication (or Request)	D1

Data

Name	Type	Comment
D1	BYTE	Indication (or Request) Data from Plug

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

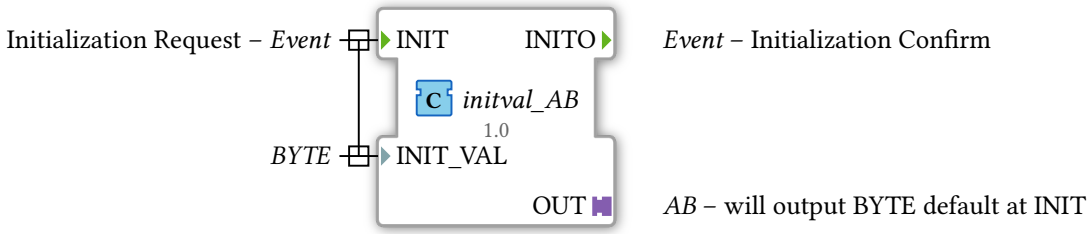
Unidirectional BYTE Adapter

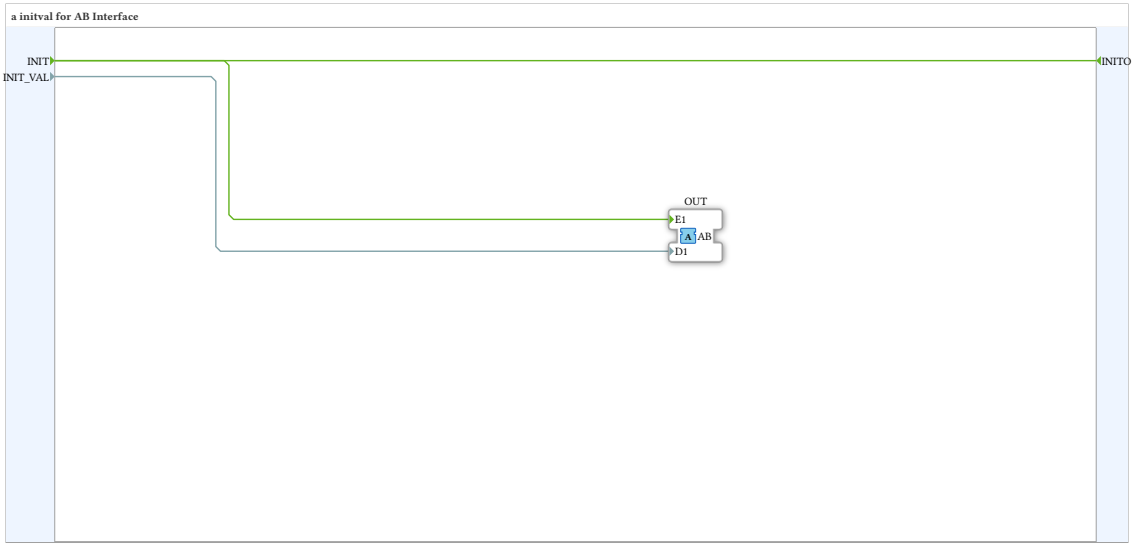
🛠️ Zugehörige Übungen

- SIN_TO_PCAN_Callback_Byte
- Uebung_128b
- Uebung_12x_sub

####### Initval

initval_AB





• – ◦ • – ◦ • – ◦ •

Einleitung

`initval_AB` ist der konfigurierbare Initialisierungsbaustein für den AB-Adapter (Datentyp `BYTE`, 8-Bit-Bitmuster): Er nimmt bei der Initialisierung einen Startwert als Dateneingang entgegen und legt diesen auf einen `AB`-Adapter-Plug. Damit ersetzt er die früher verbreitete Praxis, einen konstanten Vergleichswert über einen gemischten Adapter/Elementar-Baustein einzuspeisen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, liefert `INIT_VAL`. Löst die Ausgabe von `INIT_VAL` über den Plug `OUT` aus.

Ereignis-Ausgänge

- **INITO**: Bestätigt den Abschluss der Initialisierung.

Daten-Eingänge

- **INIT_VAL** (BYTE): Der Startwert, der beim Eintreffen von `INIT` auf den Adapter `OUT` geschrieben wird.

Daten-Ausgänge

- Keine direkten Datenausgänge; der Wert wird ausschließlich über den Adapter `OUT` bereitgestellt.

Adapter

- **OUT** (Plug, Typ `AB`): gibt bei `INIT` den über `INIT_VAL` übergebenen Wert über sein `E1`/`D1`-Adapterereignis aus.

Funktionsweise

Bei Eintreffen des Ereignisses `INIT` (mit `With Var="INIT_VAL"`) verknüpft `initval_AB` intern zwei Datenpfade: `INIT_VAL` wird per Datenverbindung auf `OUT.D1` kopiert, und `INIT` wird per Ereignisverbindung an `OUT.E1` weitergereicht – der `OUT`-Plug feuert damit sein Adapterereignis mit dem gerade übergebenen Wert. Anschließend wird `INIT` zusätzlich an `INITO` weitergereicht, wodurch der Aufrufer den Abschluss der Initialisierung bestätigt bekommt.

Technische Besonderheiten

- Der ausgegebene Wert ist nicht im Baustein selbst parametriert, sondern kommt bei jedem `INIT`-Aufruf frisch über `INIT_VAL` – der Baustein kann also auch außerhalb einer reinen Initialisierungsphase zum dynamischen Setzen eines `AB`-Werts verwendet werden.
- Reine Verdrahtungslogik ohne eigene ECC/Zustandsmaschine (Composite-FB ohne innere Funktionsblöcke); die Datenverbindung `INIT_VAL → OUT.D1` muss vor der Ereignisverbindung ausgewertet

sein, damit `OUT.E1` den aktuellen Wert trägt – dies wird durch die 61499-Ausführungsreihenfolge der `FBNetwork`-Verbindungen sichergestellt und ist nicht separat zu verdrahten.

Zustandsübersicht

Der Baustein ist zustandslos: Jedes `INIT`-Ereignis führt unmittelbar zur Übernahme von `INIT_VAL` auf `OUT` und zur Bestätigung über `INITO`.

Anwendungsszenarien

- Initialisierung von `AB`-Adapterverbindungen mit einem projekt- oder parameterabhängigen Startwert (z. B. aus einer Konfigurationsvariable)
- Ersatz für die entfernten „gemischten" Adapter↔Elementar-Vergleichsbausteine: Ein konstanter `BYTE`-Wert wird über `initval_AB` auf einen `AB`-Adapter gelegt und kann dann mit dem reinen adapterbasierten `AB`-Vergleichsbaustein weiterverwendet werden.
- Wiederverwendbare Bausteinvorlage für Testaufbauten, in denen unterschiedliche `AB`-Startwerte durchprobiert werden sollen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit initval_AX, das denselben Verdrahtungsmechanismus für den Datentyp `BOOL` nutzt. Anders als `const_AX_TRUE`/`const_AX_FALSE` (die es nur für `AX` gibt) parametriert `initval_AB` den Ausgabewert nicht fest im Baustein, sondern nimmt ihn über den Dateneingang `INIT_VAL` entgegen.

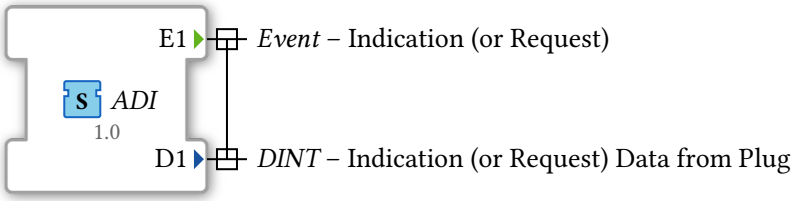
Fazit

`initval_AB` liefert eine einfache, wiederverwendbare Initialisierungslogik für den `AB`-Adapter und macht den ausgegebenen Wert über `INIT_VAL` frei konfigurierbar. Er eignet sich überall dort, wo ein `BYTE`-Startwert nicht bereits zur Entwurfszeit feststeht oder als Ersatz für einen entfernten gemischten Vergleichsbaustein benötigt wird.

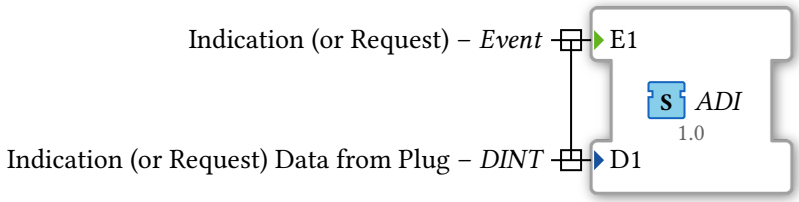
DINT

ADI (DINT)

Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



🎧 Podcast

- DIN EN 61499-1: Funktionsbausteine und der Paradigmenwechsel in der industriellen Softwareentwicklung
- Diesels radikale Vision: Warum der Erfinder alle Motoren seiner Zeit für „prinzipiell falsch" hielt – Der Weg zum Dieselmotor

Interface

Events

Name	Comment	With

E1	Indication (or Request)	D1

Data

Name	Type	Comment
D1	DINT	Indication (or Request) Data from Plug

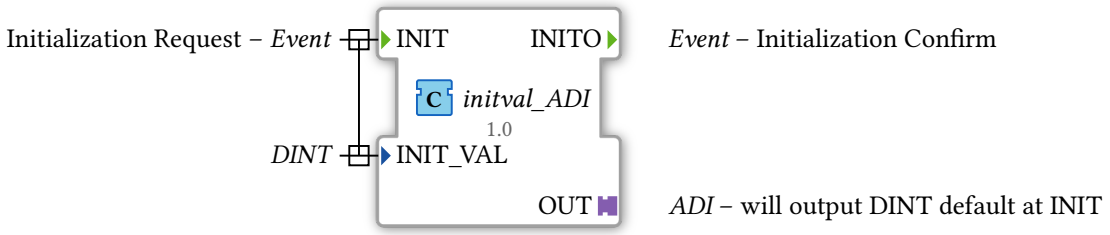
Unidirectional DINT Adapter

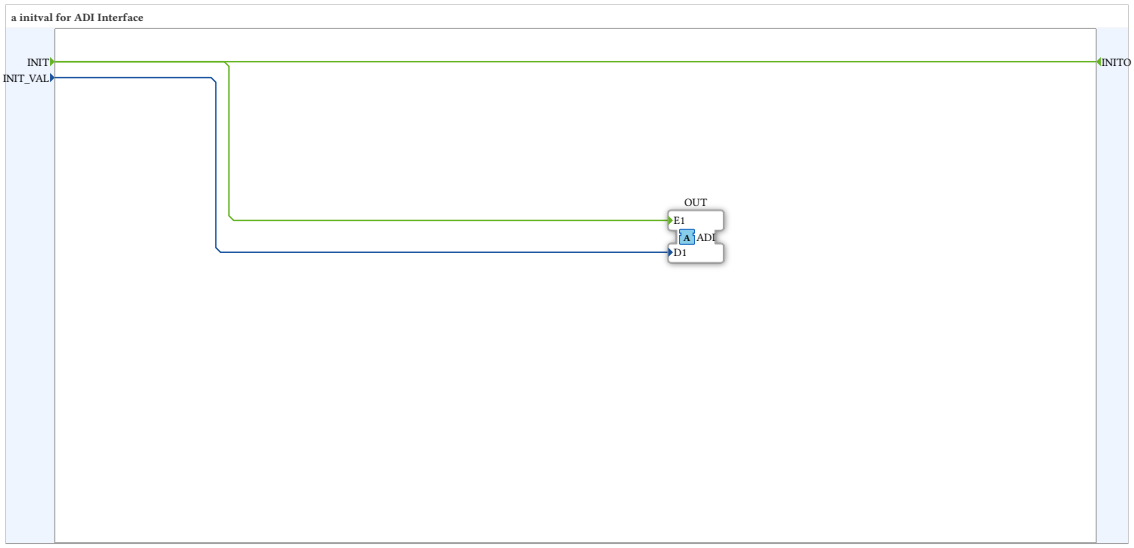
🎧 Podcast

- SINT, INT, DINT: Warum die Wahl des Datentyps über Effizienz und Fehler entscheidet

####### Initval

initval_ADI





• – ○ • – ○ • – ○ •

Einleitung

`initval_ADI` ist der konfigurierbare Initialisierungsbaustein für den ADI-Adapter (Datentyp `DINT`, 32-Bit-Ganzzahl (vorzeichenbehaftet)): Er nimmt bei der Initialisierung einen Startwert als Dateneingang entgegen und legt diesen auf einen `ADI`-Adapter-Plug. Damit ersetzt er die früher verbreitete Praxis, einen konstanten Vergleichswert über einen gemischten Adapter/Elementar-Baustein einzuspeisen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, liefert `INIT_VAL`. Löst die Ausgabe von `INIT_VAL` über den Plug `OUT` aus.

Ereignis-Ausgänge

- **INITO**: Bestätigt den Abschluss der Initialisierung.

Daten-Eingänge

- **INIT_VAL** (DINT): Der Startwert, der beim Eintreffen von `INIT` auf den Adapter `OUT` geschrieben wird.

Daten-Ausgänge

- Keine direkten Datenausgänge; der Wert wird ausschließlich über den Adapter `OUT` bereitgestellt.

Adapter

- **OUT** (Plug, Typ `ADI`): gibt bei `INIT` den über `INIT_VAL` übergebenen Wert über sein `E1`/`D1`-Adapterereignis aus.

Funktionsweise

Bei Eintreffen des Ereignisses `INIT` (mit `With Var="INIT_VAL"`) verknüpft `initval_ADI` intern zwei Datenpfade: `INIT_VAL` wird per Datenverbindung auf `OUT.D1` kopiert, und `INIT` wird per Ereignisverbindung an `OUT.E1` weitergereicht – der `OUT`-Plug feuert damit sein Adapterereignis mit dem gerade übergebenen Wert. Anschließend wird `INIT` zusätzlich an `INITO` weitergereicht, wodurch der Aufrufer den Abschluss der Initialisierung bestätigt bekommt.

Technische Besonderheiten

- Der ausgegebene Wert ist nicht im Baustein selbst parametriert, sondern kommt bei jedem `INIT`-Aufruf frisch über `INIT_VAL` – der Baustein kann also auch außerhalb einer reinen Initialisierungsphase zum dynamischen Setzen eines `ADI`-Werts verwendet werden.
- Reine Verdrahtungslogik ohne eigene ECC/Zustandsmaschine (Composite-FB ohne innere Funktionsblöcke); die Datenverbindung `INIT_VAL → OUT.D1` muss vor der Ereignisverbindung ausgewertet sein, damit `OUT.E1` den aktuellen Wert trägt – dies wird durch die 61499-Ausführungsreihenfolge der `FBNetwork`-Verbindungen sichergestellt und ist nicht separat zu verdrahten.

Zustandsübersicht

Der Baustein ist zustandslos: Jedes `INIT`-Ereignis führt unmittelbar zur Übernahme von `INIT_VAL` auf `OUT` und zur Bestätigung über `INITO`.

Anwendungsszenarien

- Initialisierung von `ADI`-Adapterverbindungen mit einem projekt- oder parameterabhängigen Startwert (z. B. aus einer Konfigurationsvariable)
- Ersatz für die entfernten „gemischten“ Adapter↔Elementar-Vergleichsbausteine: Ein konstanter `DINT`-Wert wird über `initval_ADI` auf einen `ADI`-Adapter gelegt und kann dann mit dem reinen adapterbasierten `ADI`-Vergleichsbaustein weiterverwendet werden.
- Wiederverwendbare Bausteinvorlage für Testaufbauten, in denen unterschiedliche `ADI`-Startwerte durchprobiert werden sollen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit initval_AX, das denselben Verdrahtungsmechanismus für den Datentyp `BOOL` nutzt. Anders als `const_AX_TRUE`/`const_AX_FALSE` (die es nur für `AX` gibt) parametriert `initval_ADI` den Ausgabewert nicht fest im Baustein, sondern nimmt ihn über den Dateneingang `INIT_VAL` entgegen.

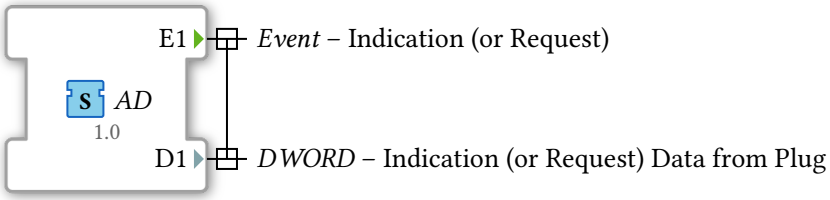
Fazit

`initval_ADI` liefert eine einfache, wiederverwendbare Initialisierungslogik für den `ADI`-Adapter und macht den ausgegebenen Wert über `INIT_VAL` frei konfigurierbar. Er eignet sich überall dort, wo ein `DINT`-Startwert nicht bereits zur Entwurfszeit feststeht oder als Ersatz für einen entfernten gemischten Vergleichsbaustein benötigt wird.

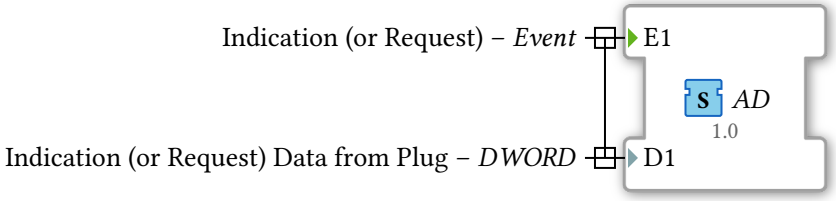
DWORD

AD (DWORD)

Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



🎧 Podcast

- Eclipse 4diac: Ihr Fahrrad für die Industrie 4.0 – Software-Aufwand meistern und verteilte Systeme steuern
- DIN EN 61499-1: Funktionsbausteine und der Paradigmenwechsel in der industriellen Softwareentwicklung
- Industrial Revolution Reloaded: Unpacking Plug & Produce, Data Privacy, and ETFA 2025
- Industrielle Automation verstehen: SPS, PLS, SCADA, MES und ERP entschlüsselt – Eine Reise durch die Smart Factory
- PLS vs. SCADA: Die Nervenzentren der Industrie – Sicherheit, Effizienz und Herausforderungen

📺 Video

- EAGLE CAD: A Legend
- Via in Pad Plated Over

Interface

Events

Name	Comment	With
E1	Indication (or Request)	D1

Data

Name	Type	Comment
D1	DWORD	Indication (or Request) Data from Plug

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

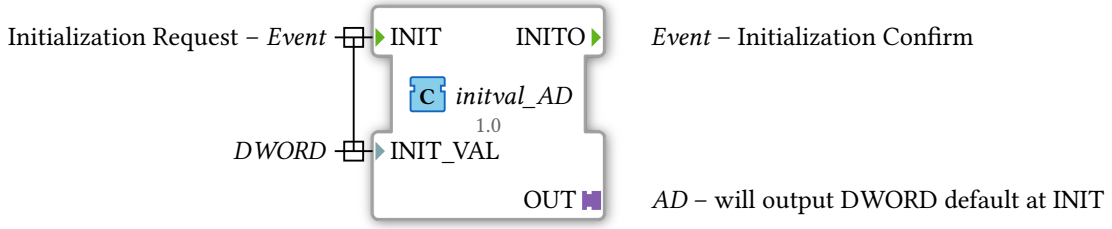
Unidirectional DWORD Adapter

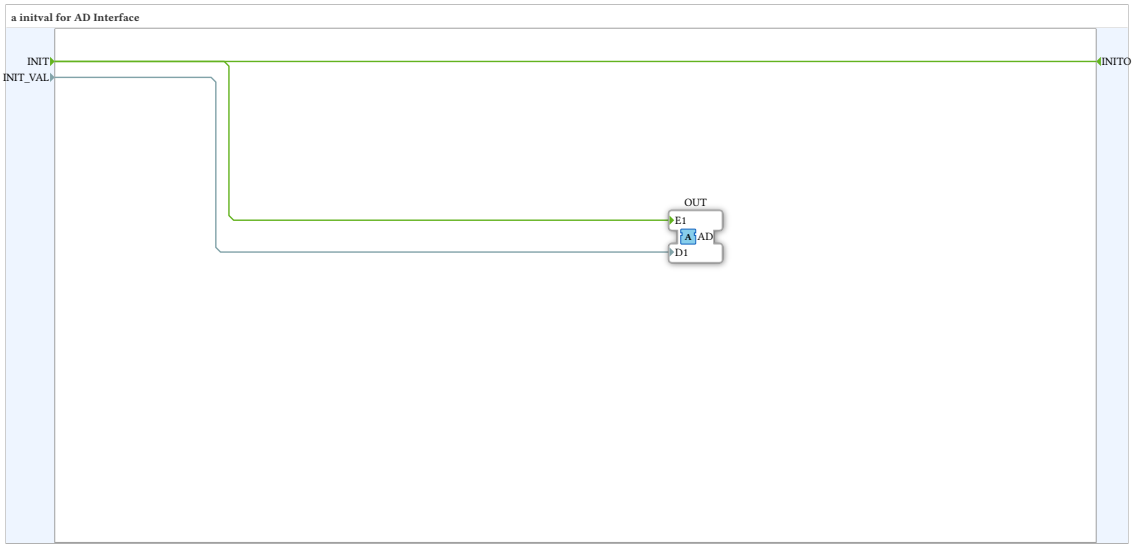
🛠️ Zugehörige Übungen

- Uebung_034b
- SIN_TO_PCAN_Callback_DWord
- Uebung_152
- Uebung_153

####### Initval

initval_AD





• – ○ • – ○ • – ○ •

Einleitung

`initval_AD` ist der konfigurierbare Initialisierungsbaustein für den AD-Adapter (Datentyp `DWORD`, 32-Bit-Bitmuster): Er nimmt bei der Initialisierung einen Startwert als Dateneingang entgegen und legt diesen auf einen `AD`-Adapter-Plug. Damit ersetzt er die früher verbreitete Praxis, einen konstanten Vergleichswert über einen gemischten Adapter/Elementar-Baustein einzuspeisen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, liefert `INIT_VAL`. Löst die Ausgabe von `INIT_VAL` über den Plug `OUT` aus.

Ereignis-Ausgänge

- **INITO**: Bestätigt den Abschluss der Initialisierung.

Daten-Eingänge

- **INIT_VAL** (DWORD): Der Startwert, der beim Eintreffen von `INIT` auf den Adapter `OUT` geschrieben wird.

Daten-Ausgänge

- Keine direkten Datenausgänge; der Wert wird ausschließlich über den Adapter `OUT` bereitgestellt.

Adapter

- **OUT** (Plug, Typ `AD`): gibt bei `INIT` den über `INIT_VAL` übergebenen Wert über sein `E1`/`D1`-Adapterereignis aus.

Funktionsweise

Bei Eintreffen des Ereignisses `INIT` (mit `With Var="INIT_VAL"`) verknüpft `initval_AD` intern zwei Datenpfade: `INIT_VAL` wird per Datenverbindung auf `OUT.D1` kopiert, und `INIT` wird per Ereignisverbindung

an OUT.E1 weitergereicht – der OUT-Plug feuert damit sein Adapterereignis mit dem gerade übergebenen Wert. Anschließend wird INIT zusätzlich an INITO weitergereicht, wodurch der Aufrufer den Abschluss der Initialisierung bestätigt bekommt.

Technische Besonderheiten

- Der ausgegebene Wert ist nicht im Baustein selbst parametriert, sondern kommt bei jedem INIT-Aufruf frisch über INIT_VAL – der Baustein kann also auch außerhalb einer reinen Initialisierungsphase zum dynamischen Setzen eines AD-Werts verwendet werden.
- Reine Verdrahtungslogik ohne eigene ECC/Zustandsmaschine (Composite-FB ohne innere Funktionsblöcke); die Datenverbindung INIT_VAL → OUT.D1 muss vor der Ereignisverbindung ausgewertet sein, damit OUT.E1 den aktuellen Wert trägt – dies wird durch die 61499-Ausführungsreihenfolge der FBNetwork-Verbindungen sichergestellt und ist nicht separat zu verdrahten.

Zustandsübersicht

Der Baustein ist zustandslos: Jedes INIT-Ereignis führt unmittelbar zur Übernahme von INIT_VAL auf OUT und zur Bestätigung über INITO.

Anwendungsszenarien

- Initialisierung von AD-Adapterverbindungen mit einem projekt- oder parameterabhängigen Startwert (z. B. aus einer Konfigurationsvariable)
- Ersatz für die entfernten „gemischten“ Adapter↔Elementar-Vergleichsbausteine: Ein konstanter DWORD-Wert wird über initval_AD auf einen AD-Adapter gelegt und kann dann mit dem reinen adapterbasierten AD-Vergleichsbaustein weiterverwendet werden.
- Wiederverwendbare Bausteinvorlage für Testaufbauten, in denen unterschiedliche AD-Startwerte durchprobiert werden sollen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit initval_AX, das denselben Verdrahtungsmechanismus für den Datentyp BOOL nutzt. Anders als const_AX_TRUE/const_AX_FALSE (die es nur für AX gibt) parametriert initval_AD den Ausgabewert nicht fest im Baustein, sondern nimmt ihn über den Dateneingang INIT_VAL entgegen.

Fazit

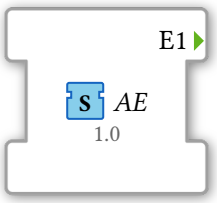
initval_AD liefert eine einfache, wiederverwendbare Initialisierungslogik für den AD-Adapter und macht den ausgegebenen Wert über INIT_VAL frei konfigurierbar. Er eignet sich überall dort, wo ein DWORD-Startwert nicht bereits zur Entwurfszeit feststeht oder als Ersatz für einen entfernten gemischten Vergleichsbaustein benötigt wird.

EVENT

- AE
- ASR
- ASRT
- EVENT_Adapter_Uni

AE (EVENT)

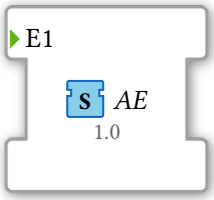
Plug (Adapter-Ausgang)



Event – Indication (or Request)

Socket (Adapter-Eingang)

Indication (or Request) – *Event*

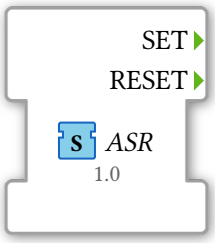


Interface

Events

Name	Comment	With
E1	Indication (or Request)	

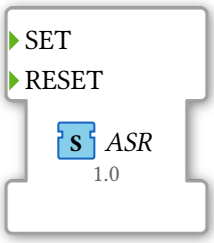
ASR (EVENT)
Plug (Adapter-Ausgang)



Event – Set / Switch on
Event – Reset / Switch off

Socket (Adapter-Eingang)

Set / Switch on – *Event*
Reset / Switch off – *Event*



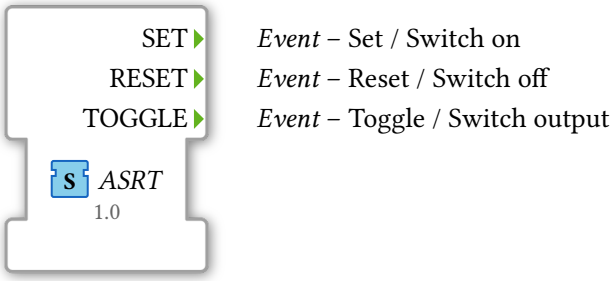
unidirectional Adapter Interface for 2 Events

Interface

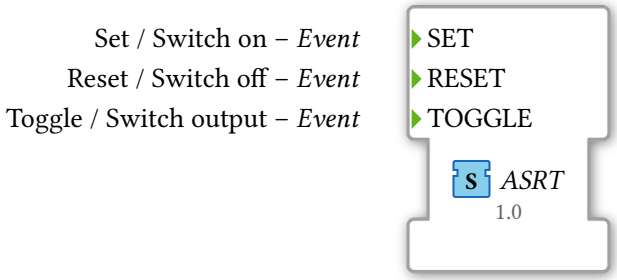
Events

Name	Comment	With
SET	Set / Switch on	
RESET	Reset / Switch off	

ASRT (EVENT)
Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



unidirectional Adapter Interface for 3 Events

Interface

Events

Name	Comment	With
SET	Set / Switch on	
RESET	Reset / Switch off	
TOGGLE	Toggle / Switch output	

Unidirectional EVENT Adapter

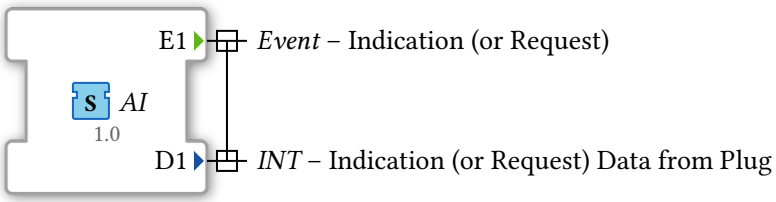
🎧 Podcast

- E_TOF vs. FB_TOF: Der Event-Timer, der nicht zyklisch tickt – Revolution für Automatisierungssysteme?
- EventFBs nach IEC 61499: Legosteine der Automatisierung – So funktionieren Ereignis-Funktionsbausteine
- IEC 61499: Revolution der Automatisierung – Event-gesteuerte FBs und verteilte Systeme erklärt
- E_REND: Event Synchronization in IEC 61499
- The Unstoppable Counter: Why IEC 61499's ECTU Guarantees Safe, Event-Driven Control (and Never Overflows)

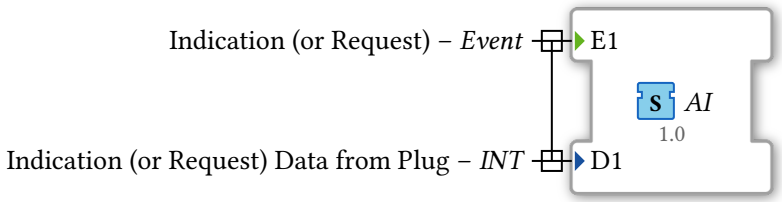
INT

AI (INT)

Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



🎧 Podcast

- Von klickenden Relais zum intelligenten Code: Wie Software die Industriesteuerung revolutionierte
- ISOBUS-Container: Dynamische Bedienfelder für klare Sicht und mehr Effizienz
- Analyse der Novellierung der Meisterprüfungsverordnung im Land- und Baumaschinenmechatroniker-Handwerk: Ein Detaillierter Vergleich der Verordnungen von 2024 und 2001
- Das Relais im Detail: Schaltverstärker, Schutz und die Geheimnisse von A1/A2, 85/86 und der Hysterese
- Das tödliche Dilemma des Relais: Warum Silberkontakte bei Kleinsignalen versagen und Gold bei Last schmilzt – Der Freibrenn-Effekt erklärt

📺 Video

- AI: The Unknown Story
- The silent failure

Interface

Events

Name	Comment	With
E1	Indication (or Request)	D1

Data

Name	Type	Comment
D1	INT	Indication (or Request) Data from Plug

Unidirectional INT Adapter

🎧 Podcast

- Eclipse 4diac 3.0: ST-Interpreter, FBE und 7200 Commits – Der Turbo für verteilte Automatisierung
- Von der SPS zur verteilten Intelligenz: IEC 61499 für Industrie 4.0 verstehen
- Simplifying Industrial Control: Your Deep Dive into Eclipse 4diac and IEC 61499
- Ereignisse und IEC 61499: Der Startschuss für intelligente Systeme
- ERP-Systeme: Das digitale Gehirn – Chancen, Risiken und der Weg zur intelligenten Produktion

🛠️ Zugehörige Übungen

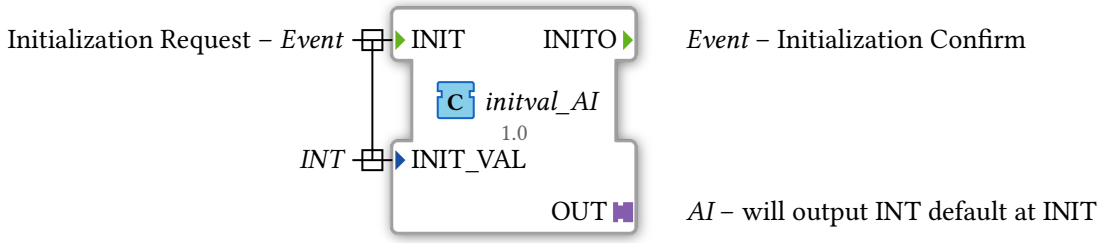
- SR_TO_QX_TRIG_EO
- SINT_TO_Q_NumericValue
- Softkey_T_FF_ILOCK_TO_QX_BG
- Bool8_TO_logiBUS_QX8

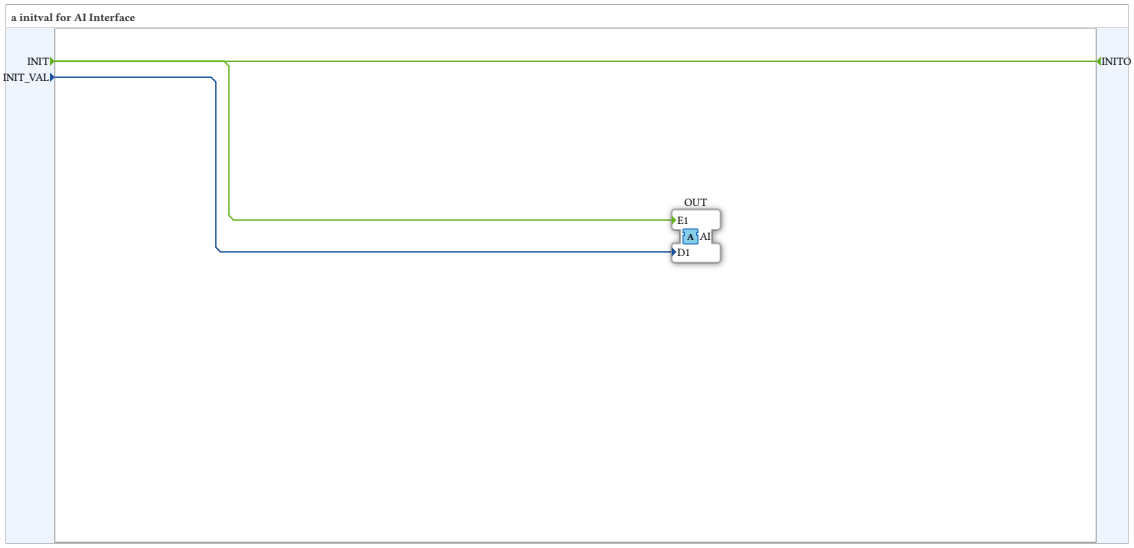
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

####### Initval

initval_AI





• – ◦ • – ◦ • – ◦ •

Einleitung

`initval_AI` ist der konfigurierbare Initialisierungsbaustein für den AI-Adapter (Datentyp `INT`, 16-Bit-Ganzzahl (vorzeichenbehaftet)): Er nimmt bei der Initialisierung einen Startwert als Dateneingang entgegen und legt diesen auf einen `AI`-Adapter-Plug. Damit ersetzt er die früher verbreitete Praxis, einen konstanten Vergleichswert über einen gemischten Adapter/Elementar-Baustein einzuspeisen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, liefert `INIT_VAL`. Löst die Ausgabe von `INIT_VAL` über den Plug `OUT` aus.

Ereignis-Ausgänge

- **INITO**: Bestätigt den Abschluss der Initialisierung.

Daten-Eingänge

- **INIT_VAL** (INT): Der Startwert, der beim Eintreffen von `INIT` auf den Adapter `OUT` geschrieben wird.

Daten-Ausgänge

- Keine direkten Datenausgänge; der Wert wird ausschließlich über den Adapter `OUT` bereitgestellt.

Adapter

- **OUT** (Plug, Typ `AI`): gibt bei `INIT` den über `INIT_VAL` übergebenen Wert über sein `E1`/`D1`-Adapterereignis aus.

Funktionsweise

Bei Eintreffen des Ereignisses `INIT` (mit `With Var="INIT_VAL"`) verknüpft `initval_AI` intern zwei Datenpfade: `INIT_VAL` wird per Datenverbindung auf `OUT.D1` kopiert, und `INIT` wird per Ereignisverbindung

an OUT.E1 weitergereicht – der OUT-Plug feuert damit sein Adapterereignis mit dem gerade übergebenen Wert. Anschließend wird INIT zusätzlich an INITO weitergereicht, wodurch der Aufrufer den Abschluss der Initialisierung bestätigt bekommt.

Technische Besonderheiten

- Der ausgegebene Wert ist nicht im Baustein selbst parametriert, sondern kommt bei jedem INIT-Aufruf frisch über INIT_VAL – der Baustein kann also auch außerhalb einer reinen Initialisierungsphase zum dynamischen Setzen eines AI-Werts verwendet werden.
- Reine Verdrahtungslogik ohne eigene ECC/Zustandsmaschine (Composite-FB ohne innere Funktionsblöcke); die Datenverbindung INIT_VAL → OUT.D1 muss vor der Ereignisverbindung ausgewertet sein, damit OUT.E1 den aktuellen Wert trägt – dies wird durch die 61499-Ausführungsreihenfolge der FBNetwork-Verbindungen sichergestellt und ist nicht separat zu verdrahten.

Zustandsübersicht

Der Baustein ist zustandslos: Jedes INIT-Ereignis führt unmittelbar zur Übernahme von INIT_VAL auf OUT und zur Bestätigung über INITO.

Anwendungsszenarien

- Initialisierung von AI-Adapterverbindungen mit einem projekt- oder parameterabhängigen Startwert (z. B. aus einer Konfigurationsvariable)
- Ersatz für die entfernten „gemischten“ Adapter↔Elementar-Vergleichsbausteine: Ein konstanter INT-Wert wird über initval_AI auf einen AI-Adapter gelegt und kann dann mit dem reinen adapterbasierten AI-Vergleichsbaustein weiterverwendet werden.
- Wiederverwendbare Bausteinvorlage für Testaufbauten, in denen unterschiedliche AI-Startwerte durchprobiert werden sollen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit initval_AX, das denselben Verdrahtungsmechanismus für den Datentyp BOOL nutzt. Anders als const_AX_TRUE/const_AX_FALSE (die es nur für AX gibt) parametriert initval_AI den Ausgabewert nicht fest im Baustein, sondern nimmt ihn über den Dateneingang INIT_VAL entgegen.

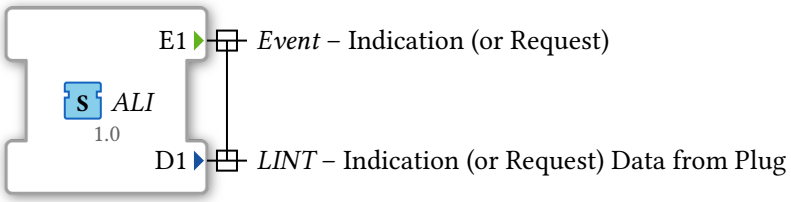
Fazit

initval_AI liefert eine einfache, wiederverwendbare Initialisierungslogik für den AI-Adapter und macht den ausgegebenen Wert über INIT_VAL frei konfigurierbar. Er eignet sich überall dort, wo ein INT-Startwert nicht bereits zur Entwurfszeit feststeht oder als Ersatz für einen entfernten gemischten Vergleichsbaustein benötigt wird.

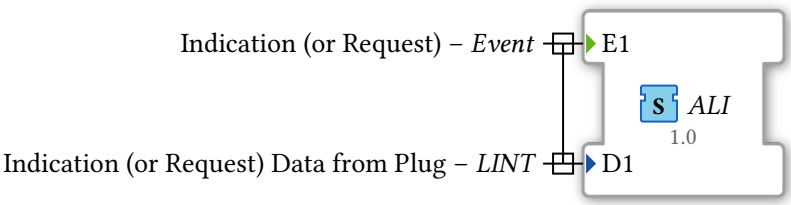
LINT

ALI (LINT)

Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



🎧 Podcast

- Der E_CTU in der IEC 61499: Ereignisgesteuertes Zählen und warum der Minimalist im Maschinenbau überzeugt
- IEC 61499: Fabrikautomation im Umbruch – Von der Vision zur industriellen Realität?
- R_TRIG & F_TRIG: Einmalige Aktionen in der zyklischen SPS-Welt der IEC 61131-3
- ISOBUS Output Meter: Dynamische Anzeigen meistern – vom Zeiger bis zur Visualisierung auf dem VT
- ISOBUS Skalierung: Wenn der Ackerschlepper-Bildschirm nicht passt – Eine Einführung in ISO 11783-6

Interface

Events

Name	Comment	With
E1	Indication (or Request)	D1

Data

Name	Type	Comment
D1	LINT	Indication (or Request) Data from Plug

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de

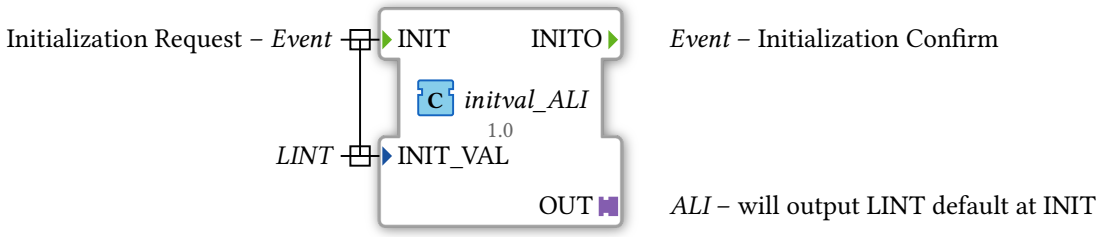
Unidirectional LINT Adapter

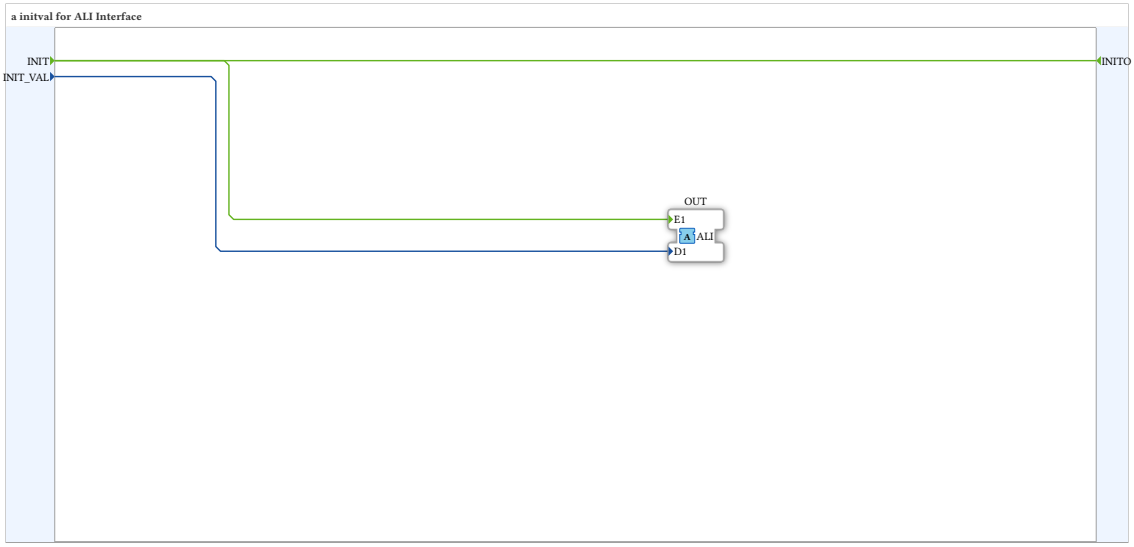
🛠️ Zugehörige Übungen

- Uebung_072c

####### Initval

initval_ALI





• – ◦ • – ◦ • – ◦ •

Einleitung

`initval_ALI` ist der konfigurierbare Initialisierungsbaustein für den ALI-Adapter (Datentyp `LINT`, 64-Bit-Ganzzahl (vorzeichenbehaftet)): Er nimmt bei der Initialisierung einen Startwert als Dateneingang entgegen und legt diesen auf einen `ALI`-Adapter-Plug. Damit ersetzt er die früher verbreitete Praxis, einen konstanten Vergleichswert über einen gemischten Adapter/Elementar-Baustein einzuspeisen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, liefert `INIT_VAL`. Löst die Ausgabe von `INIT_VAL` über den Plug `OUT` aus.

Ereignis-Ausgänge

- **INITO**: Bestätigt den Abschluss der Initialisierung.

Daten-Eingänge

- **INIT_VAL** (LINT): Der Startwert, der beim Eintreffen von `INIT` auf den Adapter `OUT` geschrieben wird.

Daten-Ausgänge

- Keine direkten Datenausgänge; der Wert wird ausschließlich über den Adapter `OUT` bereitgestellt.

Adapter

- **OUT** (Plug, Typ `ALI`): gibt bei `INIT` den über `INIT_VAL` übergebenen Wert über sein `E1`/`D1`-Adaptereereignis aus.

Funktionsweise

Bei Eintreffen des Ereignisses `INIT` (mit `With Var="INIT_VAL"`) verknüpft `initval_ALI` intern zwei Datenpfade: `INIT_VAL` wird per Datenverbindung auf `OUT.D1` kopiert, und `INIT` wird per Ereignisverbindung an `OUT.E1` weitergereicht – der `OUT`-Plug feuert damit sein Adapterereignis mit dem gerade übergebenen Wert. Anschließend wird `INIT` zusätzlich an `INITO` weitergereicht, wodurch der Aufrufer den Abschluss der Initialisierung bestätigt bekommt.

Technische Besonderheiten

- Der ausgegebene Wert ist nicht im Baustein selbst parametriert, sondern kommt bei jedem `INIT`-Aufruf frisch über `INIT_VAL` – der Baustein kann also auch außerhalb einer reinen Initialisierungsphase zum dynamischen Setzen eines `ALI`-Werts verwendet werden.
- Reine Verdrahtungslogik ohne eigene ECC/Zustandsmaschine (Composite-FB ohne innere Funktionsblöcke); die Datenverbindung `INIT_VAL → OUT.D1` muss vor der Ereignisverbindung ausgewertet

sein, damit `OUT.E1` den aktuellen Wert trägt – dies wird durch die 61499-Ausführungsreihenfolge der `FBNetwork`-Verbindungen sichergestellt und ist nicht separat zu verdrahten.

Zustandsübersicht

Der Baustein ist zustandslos: Jedes `INIT`-Ereignis führt unmittelbar zur Übernahme von `INIT_VAL` auf `OUT` und zur Bestätigung über `INITO`.

Anwendungsszenarien

- Initialisierung von `ALI`-Adapterverbindungen mit einem projekt- oder parameterabhängigen Startwert (z. B. aus einer Konfigurationsvariable)
- Ersatz für die entfernten „gemischten" Adapter↔Elementar-Vergleichsbausteine: Ein konstanter `LINT`-Wert wird über `initval_ALI` auf einen `ALI`-Adapter gelegt und kann dann mit dem reinen adapterbasierten `ALI`-Vergleichsbaustein weiterverwendet werden.
- Wiederverwendbare Bausteinvorlage für Testaufbauten, in denen unterschiedliche `ALI`-Startwerte durchprobiert werden sollen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit initval_AX, das denselben Verdrahtungsmechanismus für den Datentyp `BOOL` nutzt. Anders als `const_AX_TRUE`/`const_AX_FALSE` (die es nur für `AX` gibt) parametriert `initval_ALI` den Ausgabewert nicht fest im Baustein, sondern nimmt ihn über den Dateneingang `INIT_VAL` entgegen.

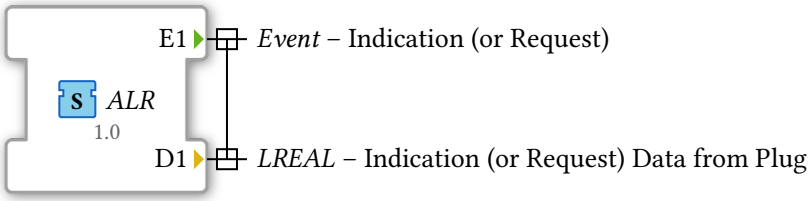
Fazit

`initval_ALI` liefert eine einfache, wiederverwendbare Initialisierungslogik für den `ALI`-Adapter und macht den ausgegebenen Wert über `INIT_VAL` frei konfigurierbar. Er eignet sich überall dort, wo ein `LINT`-Startwert nicht bereits zur Entwurfszeit feststeht oder als Ersatz für einen entfernten gemischten Vergleichsbaustein benötigt wird.

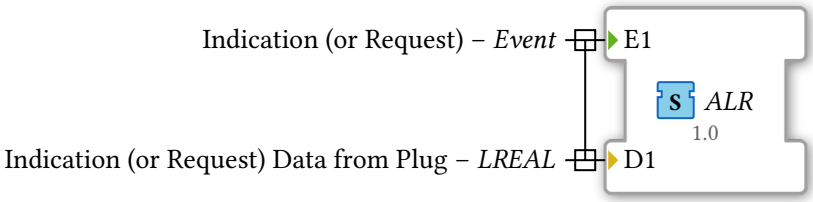
LREAL

ALR (LREAL)

Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



unidirectional Adapter Interface for 1 Event and 1 Lreal

Interface

Events

Name	Comment	With
E1	Indication (or Request)	D1

Data

Name	Type	Comment
D1	LREAL	Indication (or Request) Data from Plug

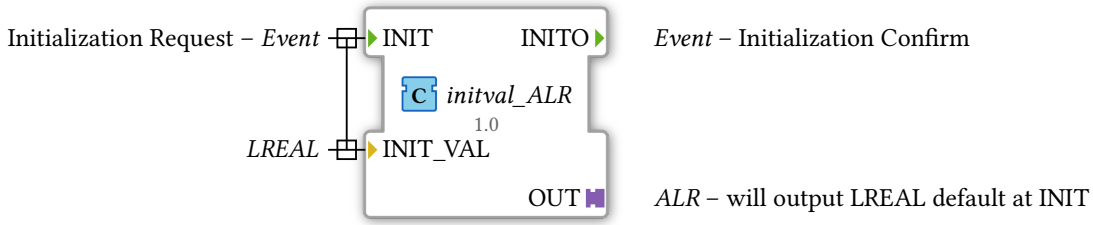
Unidirectional LREAL Adapter

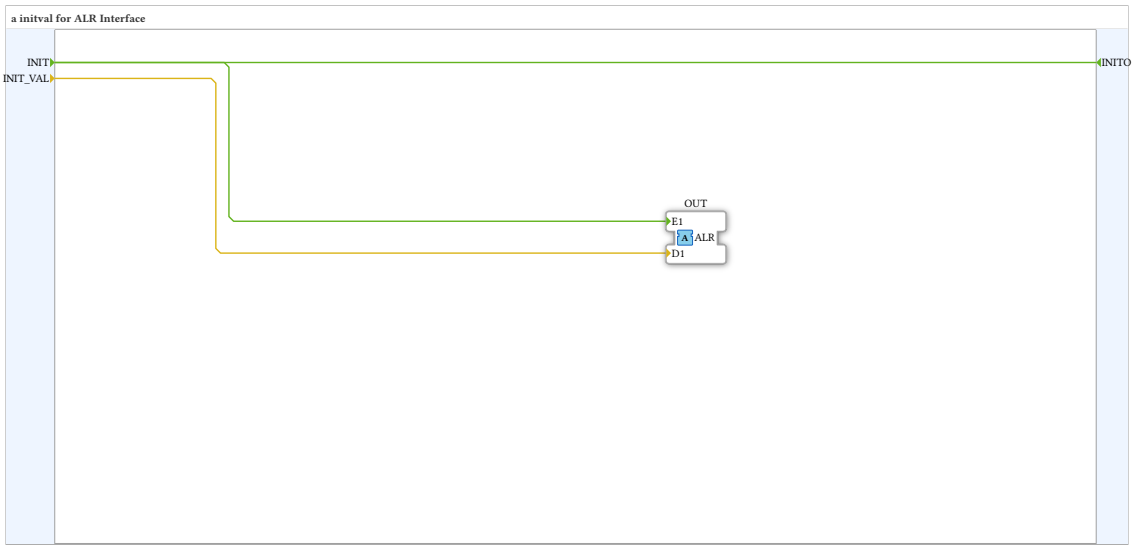
🛠️ Zugehörige Übungen

- Uebung_003a0
- Uebung_003a0_AX
- Uebung_010c2
- Uebung_010c2_AX
- Uebung_035a
- Uebung_035a2
- Uebung_035a3

####### Initval

initval_ALR





• – ○ • – ○ • – ○ •

Einleitung

`initval_ALR` ist der konfigurierbare Initialisierungsbaustein für den ALR-Adapter (Datentyp `LREAL`, 64-Bit-Gleitkommazahl): Er nimmt bei der Initialisierung einen Startwert als Dateneingang entgegen und legt diesen auf einen `ALR`-Adapter-Plug. Damit ersetzt er die früher verbreitete Praxis, einen konstanten Vergleichswert über einen gemischten Adapter/Elementar-Baustein einzuspeisen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, liefert `INIT_VAL`. Löst die Ausgabe von `INIT_VAL` über den Plug `OUT` aus.

Ereignis-Ausgänge

- **INITO**: Bestätigt den Abschluss der Initialisierung.

Daten-Eingänge

- **INIT_VAL** (LREAL): Der Startwert, der beim Eintreffen von `INIT` auf den Adapter `OUT` geschrieben wird.

Daten-Ausgänge

- Keine direkten Datenausgänge; der Wert wird ausschließlich über den Adapter `OUT` bereitgestellt.

Adapter

- **OUT** (Plug, Typ `ALR`): gibt bei `INIT` den über `INIT_VAL` übergebenen Wert über sein `E1`/`D1`-Adapterereignis aus.

Funktionsweise

Bei Eintreffen des Ereignisses `INIT` (mit `With Var="INIT_VAL"`) verknüpft `initval_ALR` intern zwei Datenpfade: `INIT_VAL` wird per Datenverbindung auf `OUT.D1` kopiert, und `INIT` wird per Ereignisverbindung an `OUT.E1` weitergereicht – der `OUT`-Plug feuert damit sein Adapterereignis mit dem gerade übergebenen Wert. Anschließend wird `INIT` zusätzlich an `INITO` weitergereicht, wodurch der Aufrufer den Abschluss der Initialisierung bestätigt bekommt.

Technische Besonderheiten

- Der ausgegebene Wert ist nicht im Baustein selbst parametriert, sondern kommt bei jedem `INIT`-Aufruf frisch über `INIT_VAL` – der Baustein kann also auch außerhalb einer reinen Initialisierungsphase zum dynamischen Setzen eines `ALR`-Werts verwendet werden.
- Reine Verdrahtungslogik ohne eigene ECC/Zustandsmaschine (Composite-FB ohne innere Funktionsblöcke); die Datenverbindung `INIT_VAL → OUT.D1` muss vor der Ereignisverbindung ausgewertet sein, damit `OUT.E1` den aktuellen Wert trägt – dies wird durch die 61499-Ausführungsreihenfolge der `FBNetwork`-Verbindungen sichergestellt und ist nicht separat zu verdrahten.

Zustandsübersicht

Der Baustein ist zustandslos: Jedes `INIT`-Ereignis führt unmittelbar zur Übernahme von `INIT_VAL` auf `OUT` und zur Bestätigung über `INITO`.

Anwendungsszenarien

- Initialisierung von `ALR`-Adapterverbindungen mit einem projekt- oder parameterabhängigen Startwert (z. B. aus einer Konfigurationsvariable)
- Ersatz für die entfernten „gemischten“ Adapter↔Elementar-Vergleichsbausteine: Ein konstanter `LREAL`-Wert wird über `initval_ALR` auf einen `ALR`-Adapter gelegt und kann dann mit dem reinen adapterbasierten `ALR`-Vergleichsbaustein weiterverwendet werden.
- Wiederverwendbare Bausteinvorlage für Testaufbauten, in denen unterschiedliche `ALR`-Startwerte durchprobiert werden sollen

⚖️ Vergleich mit ähnlichen Bausteinen

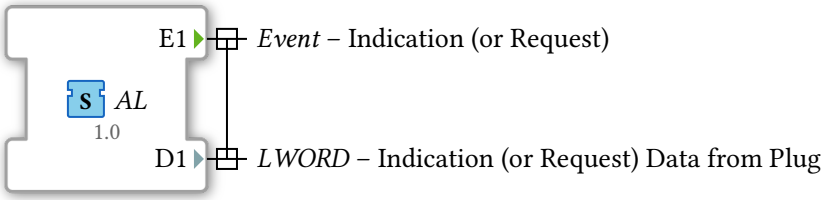
Vergleich mit initval_AX, das denselben Verdrahtungsmechanismus für den Datentyp `BOOL` nutzt. Anders als `const_AX_TRUE`/`const_AX_FALSE` (die es nur für `AX` gibt) parametriert `initval_ALR` den Ausgabewert nicht fest im Baustein, sondern nimmt ihn über den Dateneingang `INIT_VAL` entgegen.

Fazit

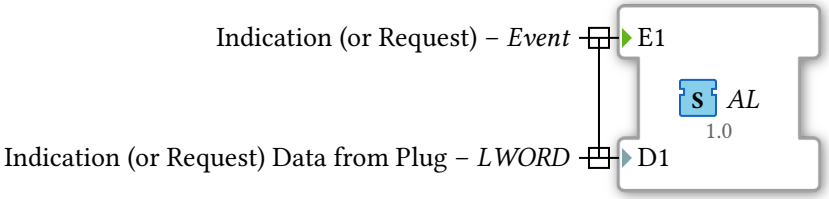
`initval_ALR` liefert eine einfache, wiederverwendbare Initialisierungslogik für den `ALR`-Adapter und macht den ausgegebenen Wert über `INIT_VAL` frei konfigurierbar. Er eignet sich überall dort, wo ein `LREAL`-Startwert nicht bereits zur Entwurfszeit feststeht oder als Ersatz für einen entfernten gemischten Vergleichsbaustein benötigt wird.

LWORD

AL (LWORD)
Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



🎧 Podcast

- 4diac IDE: Dein „Hello World“ der Automatisierung – Das Blinking Tutorial Lokal
- 4diac-Präsentation: Zielgruppen, Struktur und Alleinstellungsmerkmal Schärfen
- Eclipse 4diac: Open Source als Game Changer für industrielle Steuerungen?
- Eclipse 4diac and FORTE: Unlocking Next-Gen Industrial Automation with Open-Source Power
- Eclipse 4diac: Revolutionizing Industrial Control with Open-Source Cyber-Physical Systems

📺 Video

- Der Katalog von 1863
- Der smarte Schalter
- Der unaufhaltsame Chip
- Digital decisions
- Ideale Dioden

Interface

Events

Name	Comment	With
E1	Indication (or Request)	D1

Data

Name	Type	Comment
D1	LWORD	Indication (or Request) Data from Plug

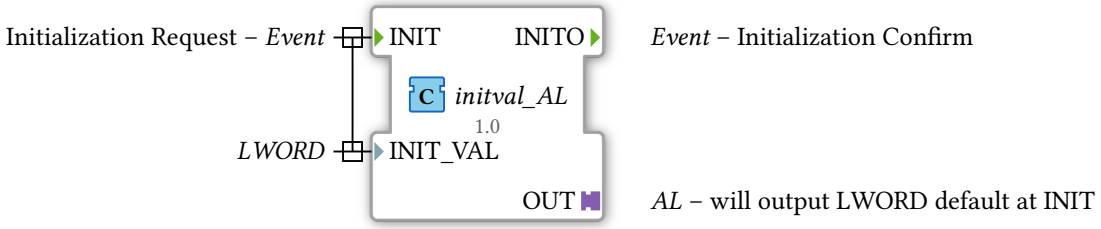
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

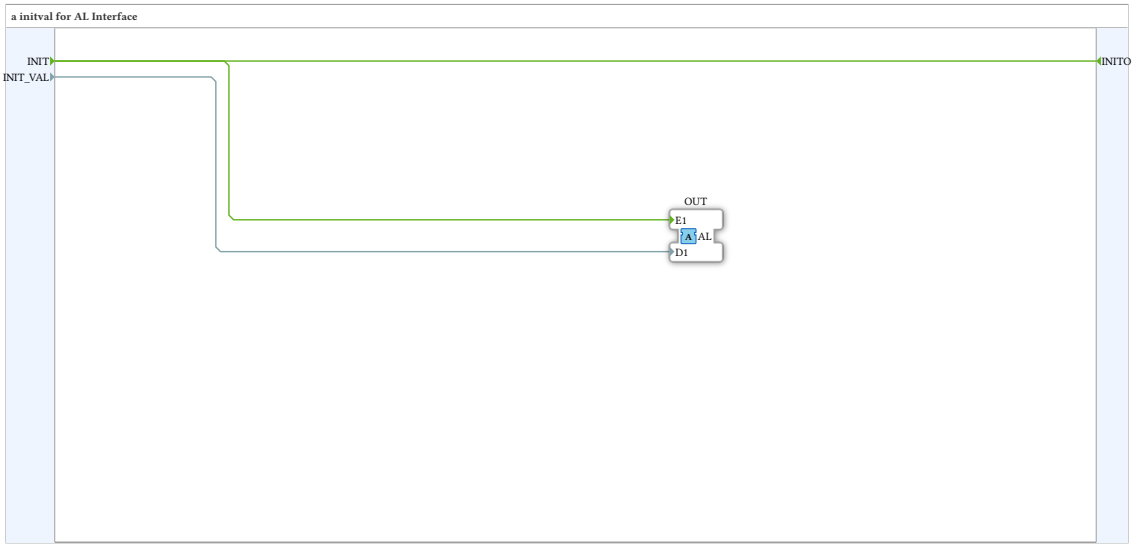
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 Diode & Halbleitergrundlagen auf ms-muc-docs.de
- 🌐 Smart Switch vs. Relais auf ms-muc-docs.de

Unidirectional LWORD Adapter

####### Initval

initval_AL





• – ◦ • – ◦ • – ◦ •

Einleitung

`initval_AL` ist der konfigurierbare Initialisierungsbaustein für den AL-Adapter (Datentyp `LWORD`, 64-Bit-Bitmuster): Er nimmt bei der Initialisierung einen Startwert als Dateneingang entgegen und legt diesen auf einen `AL`-Adapter-Plug. Damit ersetzt er die früher verbreitete Praxis, einen konstanten Vergleichswert über einen gemischten Adapter/Elementar-Baustein einzuspeisen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, liefert `INIT_VAL`. Löst die Ausgabe von `INIT_VAL` über den Plug `OUT` aus.

Ereignis-Ausgänge

- **INITO**: Bestätigt den Abschluss der Initialisierung.

Daten-Eingänge

- **INIT_VAL** (LWORD): Der Startwert, der beim Eintreffen von `INIT` auf den Adapter `OUT` geschrieben wird.

Daten-Ausgänge

- Keine direkten Datenausgänge; der Wert wird ausschließlich über den Adapter `OUT` bereitgestellt.

Adapter

- **OUT** (Plug, Typ `AL`): gibt bei `INIT` den über `INIT_VAL` übergebenen Wert über sein `E1/D1`-Adapterereignis aus.

Funktionsweise

Bei Eintreffen des Ereignisses `INIT` (mit `With Var="INIT_VAL"`) verknüpft `initval_AL` intern zwei Datenpfade: `INIT_VAL` wird per Datenverbindung auf `OUT.D1` kopiert, und `INIT` wird per Ereignisverbindung

an OUT.E1 weitergereicht – der OUT-Plug feuert damit sein Adapterereignis mit dem gerade übergebenen Wert. Anschließend wird INIT zusätzlich an INITO weitergereicht, wodurch der Aufrufer den Abschluss der Initialisierung bestätigt bekommt.

Technische Besonderheiten

- Der ausgegebene Wert ist nicht im Baustein selbst parametriert, sondern kommt bei jedem INIT-Aufruf frisch über INIT_VAL – der Baustein kann also auch außerhalb einer reinen Initialisierungsphase zum dynamischen Setzen eines AL-Werts verwendet werden.
- Reine Verdrahtungslogik ohne eigene ECC/Zustandsmaschine (Composite-FB ohne innere Funktionsblöcke); die Datenverbindung INIT_VAL → OUT.D1 muss vor der Ereignisverbindung ausgewertet sein, damit OUT.E1 den aktuellen Wert trägt – dies wird durch die 61499-Ausführungsreihenfolge der FBNetwork-Verbindungen sichergestellt und ist nicht separat zu verdrahten.

Zustandsübersicht

Der Baustein ist zustandslos: Jedes INIT-Ereignis führt unmittelbar zur Übernahme von INIT_VAL auf OUT und zur Bestätigung über INITO.

Anwendungsszenarien

- Initialisierung von AL-Adapterverbindungen mit einem projekt- oder parameterabhängigen Startwert (z. B. aus einer Konfigurationsvariable)
- Ersatz für die entfernten „gemischten“ Adapter↔Elementar-Vergleichsbausteine: Ein konstanter LWORD-Wert wird über initval_AL auf einen AL-Adapter gelegt und kann dann mit dem reinen adapterbasierten AL-Vergleichsbaustein weiterverwendet werden.
- Wiederverwendbare Bausteinvorlage für Testaufbauten, in denen unterschiedliche AL-Startwerte durchprobiert werden sollen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit initval_AX, das denselben Verdrahtungsmechanismus für den Datentyp BOOL nutzt. Anders als const_AX_TRUE/const_AX_FALSE (die es nur für AX gibt) parametriert initval_AL den Ausgabewert nicht fest im Baustein, sondern nimmt ihn über den Dateneingang INIT_VAL entgegen.

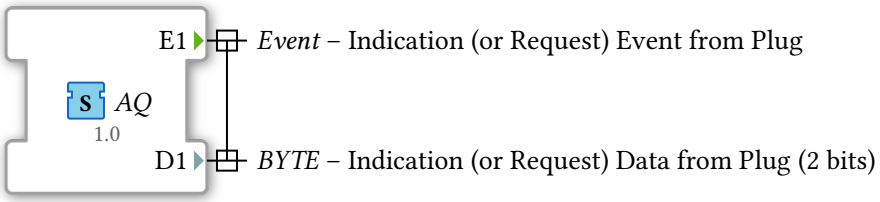
Fazit

initval_AL liefert eine einfache, wiederverwendbare Initialisierungslogik für den AL-Adapter und macht den ausgegebenen Wert über INIT_VAL frei konfigurierbar. Er eignet sich überall dort, wo ein LWORD-Startwert nicht bereits zur Entwurfszeit feststeht oder als Ersatz für einen entfernten gemischten Vergleichsbaustein benötigt wird.

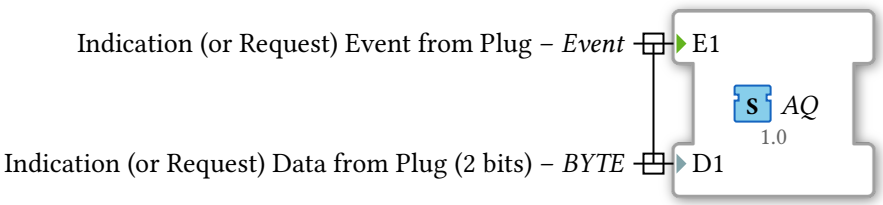
QUARTER

AQ

Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Adapter `AQ` stellt eine unidirektionale Schnittstelle (Plug-Seite) für die Übertragung eines Ereignisses und eines Byte-Wertes (genutzt für 2 Bit) bereit. Er dient als Standarddefinition für die Kommunikation zwischen Funktionsbausteinen nach IEC 61499-1 und ermöglicht eine einfache, gerichtete Datenübergabe.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Name	Typ	Kommentar
E1	Event	Indication (or Request) Event from Plug (wird zusammen mit D1 ausgelöst)

Daten-Eingänge

Keine.

Daten-Ausgänge

Name	Typ	Kommentar
D1	BYTE	Indication (or Request) Data from Plug (2 Bit genutzt)

Adapter

Keine.

Funktionsweise

Der Adapter `AQ` definiert eine unidirektionale Verbindung. Ein angeschlossener Funktionsbaustein kann über das Ereignis `E1` signalisieren, dass der am Datenausgang `D1` anliegende Byte-Wert gültig ist. Der Datenwert ist ein Byte, von dem die unteren 2 Bits die eigentliche Information tragen. Die restlichen 6 Bits sind für die Anwendung nicht definiert. Der Adapter selbst besitzt keine Verarbeitungslogik – er stellt ausschließlich die Schnittstellenspezifikation dar.

Technische Besonderheiten

- **Unidirektional**: Nur Ereignis- und Datenausgänge vorhanden, keine Eingänge. Geeignet für Plug-Seiten in Adapter-Paarungen.
- **Datennutzung**: Von den 8 Bit des Byte-Wertes werden nur die unteren 2 Bit als signifikant angesehen. Die übrigen Bits sollten durch den sendenden FB auf `0` gesetzt oder ignoriert werden.
- **Event + Data**: Das Ereignis `E1` ist mit dem Datenausgang `D1` verknüpft (`With`-Relation). Beide werden gemeinsam ausgesendet.
- **Kompakt**: Minimale Schnittstelle mit einem Ereignis und einem Byte.

Zustandsübersicht

Der Adapter selbst besitzt keinen internen Zustand. Der sendende Funktionsbaustein steuert die Ausgaben basierend auf seiner eigenen Zustandsmaschine. Der Adapter definiert lediglich das Protokoll: Ereignis und Daten werden gemeinsam übertragen.

Anwendungsszenarien

- Übertragung von 2-Bit-Statusinformationen (z. B. Betriebszustand "Ein/Aus/Störung" oder zwei separate Flags).
- Einfache Befehlsübermittlung mit kodiertem Byte (z. B. 2-Bit-Steuerworte in Automatisierungssystemen).
- Als Teil einer übergeordneten Adapter-Paarung für minimale, gerichtete Kommunikation zwischen FBs.

Vergleich mit ähnlichen Bausteinen

Merkmal	AQ (dieser Adapter)	Andere Adapter (z. B. mit mehreren Events)

Ereignisse	1 Ausgang	Oft mehrere Ein-/Ausgänge
Daten	1 Byte (2 Bit genutzt)	Häufig mehrere Variablen oder komplexe Typen
Ausrichtung	Unidirektional (Plug-Seite)	Bidirektional oder reine Socket-Seite
Komplexität	Sehr gering	Mittel bis hoch

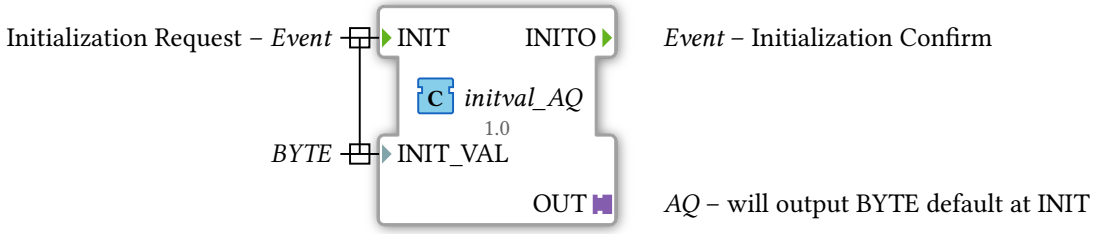
Der `AQ` ist speziell für den Minimalfall “Ein Ereignis + ein Byte” optimiert.

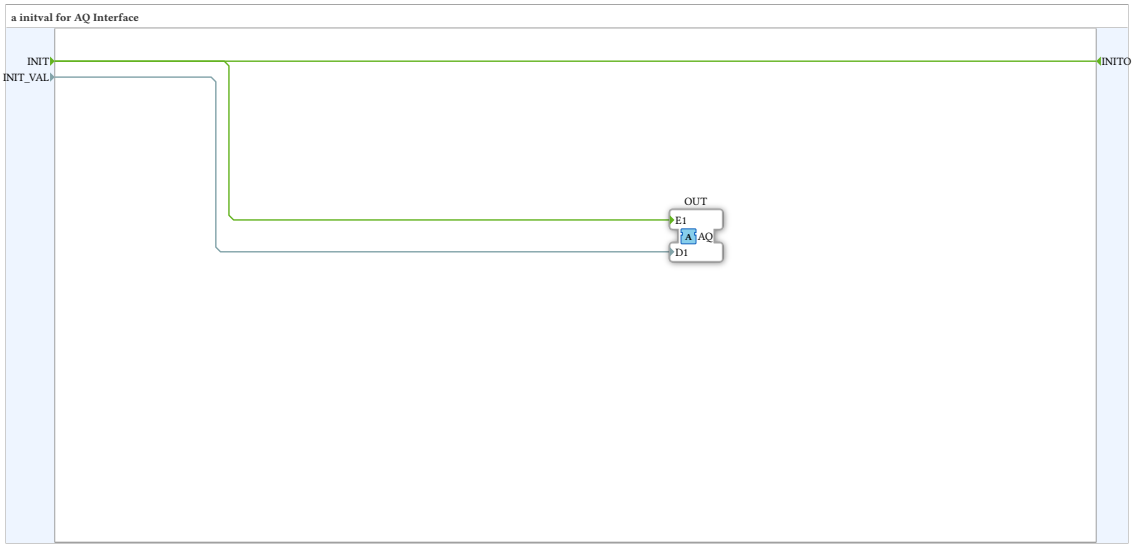
Fazit

Der Adapter `AQ` bietet eine schlanke und standardisierte Schnittstelle für die unidirektionale Übertragung eines Ereignisses und eines 2-Bit-Signals. Seine Einfachheit macht ihn ideal für Anwendungen, die eine reduzierte Kommunikation erfordern und gleichzeitig die IEC-61499-Kompatibilität wahren. Der Adapter ist als Plug-Seite definiert und erwartet einen entsprechenden Socket-Gegenpart.

####### Initval

initval_AQ





• – ◦ • – ◦ • – ◦ •

Einleitung

`initval_AQ` ist der konfigurierbare Initialisierungsbaustein für den AQ-Adapter (Datentyp `BYTE`, 2-Bit-Wert (als Byte übertragen)): Er nimmt bei der Initialisierung einen Startwert als Dateneingang entgegen und legt diesen auf einen `AQ`-Adapter-Plug. Damit ersetzt er die früher verbreitete Praxis, einen konstanten Vergleichswert über einen gemischten Adapter/Elementar-Baustein einzuspeisen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, liefert `INIT_VAL`. Löst die Ausgabe von `INIT_VAL` über den Plug `OUT` aus.

Ereignis-Ausgänge

- **INITO**: Bestätigt den Abschluss der Initialisierung.

Daten-Eingänge

- **INIT_VAL** (BYTE): Der Startwert, der beim Eintreffen von `INIT` auf den Adapter `OUT` geschrieben wird.

Daten-Ausgänge

- Keine direkten Datenausgänge; der Wert wird ausschließlich über den Adapter `OUT` bereitgestellt.

Adapter

- **OUT** (Plug, Typ `AQ`): gibt bei `INIT` den über `INIT_VAL` übergebenen Wert über sein `E1`/`D1`-Adapterereignis aus.

Funktionsweise

Bei Eintreffen des Ereignisses `INIT` (mit `With Var="INIT_VAL"`) verknüpft `initval_AQ` intern zwei Datenpfade: `INIT_VAL` wird per Datenverbindung auf `OUT.D1` kopiert, und `INIT` wird per Ereignisverbindung an `OUT.E1` weitergereicht – der `OUT`-Plug feuert damit sein Adapterereignis mit dem gerade übergebenen Wert. Anschließend wird `INIT` zusätzlich an `INITO` weitergereicht, wodurch der Aufrufer den Abschluss der Initialisierung bestätigt bekommt.

Technische Besonderheiten

- Der ausgegebene Wert ist nicht im Baustein selbst parametriert, sondern kommt bei jedem `INIT`-Aufruf frisch über `INIT_VAL` – der Baustein kann also auch außerhalb einer reinen Initialisierungsphase zum dynamischen Setzen eines `AQ`-Werts verwendet werden.
- Reine Verdrahtungslogik ohne eigene ECC/Zustandsmaschine (Composite-FB ohne innere Funktionsblöcke); die Datenverbindung `INIT_VAL → OUT.D1` muss vor der Ereignisverbindung ausgewertet sein, damit `OUT.E1` den aktuellen Wert trägt – dies wird durch die 61499-Ausführungsreihenfolge der `FBNetwork`-Verbindungen sichergestellt und ist nicht separat zu verdrahten.

Zustandsübersicht

Der Baustein ist zustandslos: Jedes `INIT`-Ereignis führt unmittelbar zur Übernahme von `INIT_VAL` auf `OUT` und zur Bestätigung über `INITO`.

Anwendungsszenarien

- Initialisierung von `AQ`-Adapterverbindungen mit einem projekt- oder parameterabhängigen Startwert (z. B. aus einer Konfigurationsvariable)
- Ersatz für die entfernten „gemischten“ Adapter↔Elementar-Vergleichsbausteine: Ein konstanter `BYTE`-Wert wird über `initval_AQ` auf einen `AQ`-Adapter gelegt und kann dann mit dem reinen adapterbasierten `AQ`-Vergleichsbaustein weiterverwendet werden.
- Wiederverwendbare Bausteinvorlage für Testaufbauten, in denen unterschiedliche `AQ`-Startwerte durchprobiert werden sollen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit initval_AX, das denselben Verdrahtungsmechanismus für den Datentyp `BOOL` nutzt. Anders als `const_AX_TRUE`/`const_AX_FALSE` (die es nur für `AX` gibt) parametriert `initval_AQ` den Ausgabewert nicht fest im Baustein, sondern nimmt ihn über den Dateneingang `INIT_VAL` entgegen.

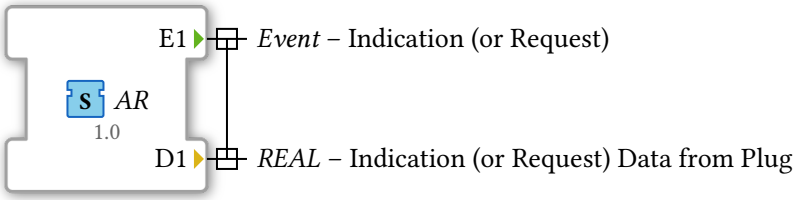
Fazit

`initval_AQ` liefert eine einfache, wiederverwendbare Initialisierungslogik für den `AQ`-Adapter und macht den ausgegebenen Wert über `INIT_VAL` frei konfigurierbar. Er eignet sich überall dort, wo ein `BYTE`-Startwert nicht bereits zur Entwurfszeit feststeht oder als Ersatz für einen entfernten gemischten Vergleichsbaustein benötigt wird.

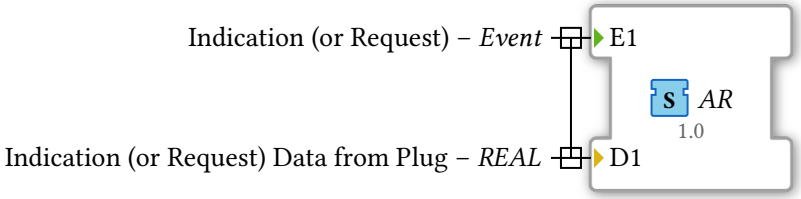
REAL

AR (REAL)

Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



🎧 Podcast

- From „Get Away with Electronics“ to Essential: How Agricultural Tech Earned Trust and Delivered Real Relief
- 4diac IDE: Wie der IEC 61499 Standard die Industrieautomatisierung revolutioniert
- Den Software-Drachen zähmen: Industrielle Automatisierung und die Zukunft der Produktion
- Eclipse 4diac: Ihr Fahrrad für die Industrie 4.0 – Software-Aufwand meistern und verteilte Systeme steuern
- Software-Drachen zähmen: Warum IEC 61499 die Industrie revolutionieren könnte

📺 Video

- Charles Babbage
- Der smarte Schalter
- The high-tech harvest
- Vom Chaos zur Klarheit

Interface

Events

Name	Comment	With
E1	Indication (or Request)	D1

Data

Name	Type	Comment
D1	REAL	Indication (or Request) Data from Plug

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Unidirectional REAL Adapter

🎧 Podcast

- IEC 61499: Fabrikautomation im Umbruch – Von der Vision zur industriellen Realität?
- From „Get Away with Electronics“ to Essential: How Agricultural Tech Earned Trust and Delivered Real Relief

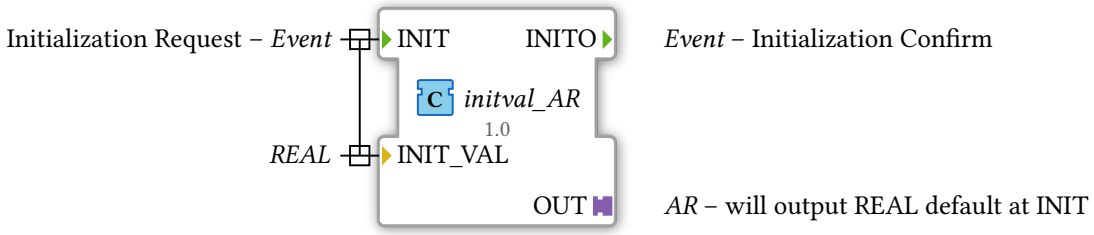
🛠️ Zugehörige Übungen

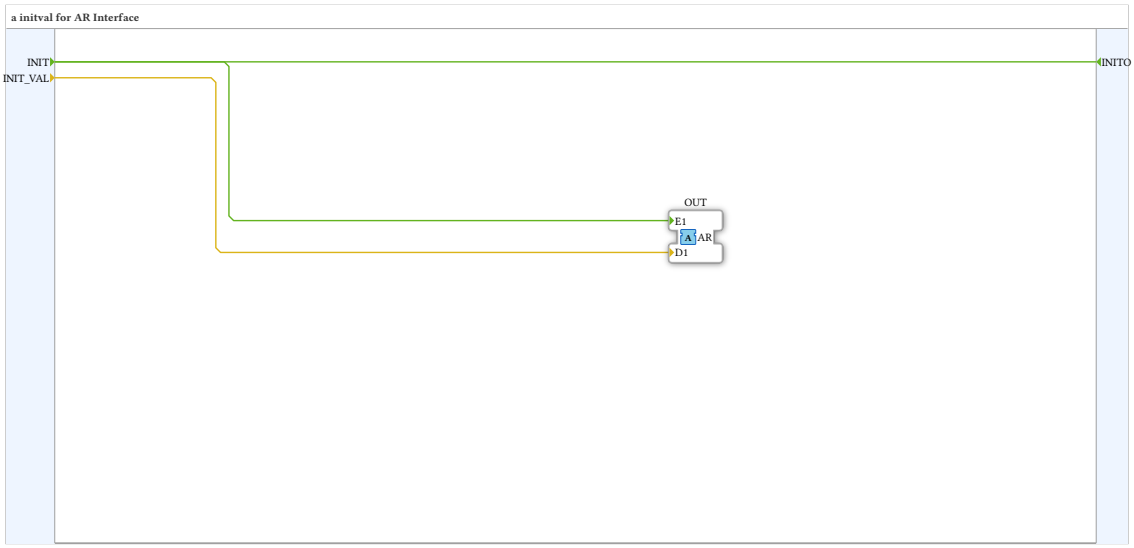
- Uebung_072c

- Uebung_151_AX
- Uebung_152
- Uebung_153

####### Initval

initval_AR





• – ◦ • – ◦ • – ◦ •

Einleitung

`initval_AR` ist der konfigurierbare Initialisierungsbaustein für den AR-Adapter (Datentyp `REAL`, 32-Bit-Gleitkommazahl): Er nimmt bei der Initialisierung einen Startwert als Dateneingang entgegen und legt diesen auf einen `AR`-Adapter-Plug. Damit ersetzt er die früher verbreitete Praxis, einen konstanten Vergleichswert über einen gemischten Adapter/Elementar-Baustein einzuspeisen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, liefert `INIT_VAL`. Löst die Ausgabe von `INIT_VAL` über den Plug `OUT` aus.

Ereignis-Ausgänge

- **INITO**: Bestätigt den Abschluss der Initialisierung.

Daten-Eingänge

- **INIT_VAL** (REAL): Der Startwert, der beim Eintreffen von `INIT` auf den Adapter `OUT` geschrieben wird.

Daten-Ausgänge

- Keine direkten Datenausgänge; der Wert wird ausschließlich über den Adapter `OUT` bereitgestellt.

Adapter

- **OUT** (Plug, Typ `AR`): gibt bei `INIT` den über `INIT_VAL` übergebenen Wert über sein `E1`/`D1`-Adapterereignis aus.

Funktionsweise

Bei Eintreffen des Ereignisses `INIT` (mit `With Var="INIT_VAL"`) verknüpft `initval_AR` intern zwei Datenpfade: `INIT_VAL` wird per Datenverbindung auf `OUT.D1` kopiert, und `INIT` wird per Ereignisverbindung an `OUT.E1` weitergereicht – der `OUT`-Plug feuert damit sein Adapterereignis mit dem gerade übergebenen Wert. Anschließend wird `INIT` zusätzlich an `INITO` weitergereicht, wodurch der Aufrufer den Abschluss der Initialisierung bestätigt bekommt.

Technische Besonderheiten

- Der ausgegebene Wert ist nicht im Baustein selbst parametriert, sondern kommt bei jedem `INIT`-Aufruf frisch über `INIT_VAL` – der Baustein kann also auch außerhalb einer reinen Initialisierungsphase zum dynamischen Setzen eines `AR`-Werts verwendet werden.
- Reine Verdrahtungslogik ohne eigene ECC/Zustandsmaschine (Composite-FB ohne innere Funktionsblöcke); die Datenverbindung `INIT_VAL → OUT.D1` muss vor der Ereignisverbindung ausgewertet sein, damit `OUT.E1` den aktuellen Wert trägt – dies wird durch die 61499-Ausführungsreihenfolge der `FBNetwork`-Verbindungen sichergestellt und ist nicht separat zu verdrahten.

Zustandsübersicht

Der Baustein ist zustandslos: Jedes `INIT`-Ereignis führt unmittelbar zur Übernahme von `INIT_VAL` auf `OUT` und zur Bestätigung über `INITO`.

Anwendungsszenarien

- Initialisierung von `AR`-Adapterverbindungen mit einem projekt- oder parameterabhängigen Startwert (z. B. aus einer Konfigurationsvariable)
- Ersatz für die entfernten „gemischten“ Adapter↔Elementar-Vergleichsbausteine: Ein konstanter `REAL`-Wert wird über `initval_AR` auf einen `AR`-Adapter gelegt und kann dann mit dem reinen adapterbasierten `AR`-Vergleichsbaustein weiterverwendet werden.
- Wiederverwendbare Bausteinvorlage für Testaufbauten, in denen unterschiedliche `AR`-Startwerte durchprobiert werden sollen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit initval_AX, das denselben Verdrahtungsmechanismus für den Datentyp `BOOL` nutzt. Anders als `const_AX_TRUE`/`const_AX_FALSE` (die es nur für `AX` gibt) parametriert `initval_AR` den Ausgabewert nicht fest im Baustein, sondern nimmt ihn über den Dateneingang `INIT_VAL` entgegen.

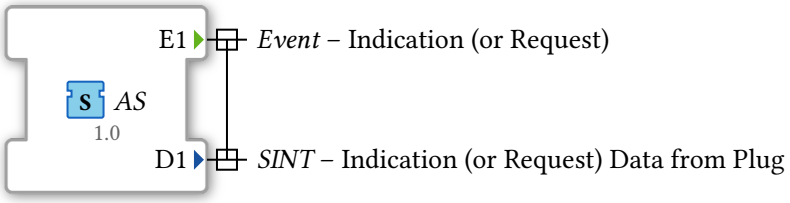
Fazit

`initval_AR` liefert eine einfache, wiederverwendbare Initialisierungslogik für den `AR`-Adapter und macht den ausgegebenen Wert über `INIT_VAL` frei konfigurierbar. Er eignet sich überall dort, wo ein `REAL`-Startwert nicht bereits zur Entwurfszeit feststeht oder als Ersatz für einen entfernten gemischten Vergleichsbaustein benötigt wird.

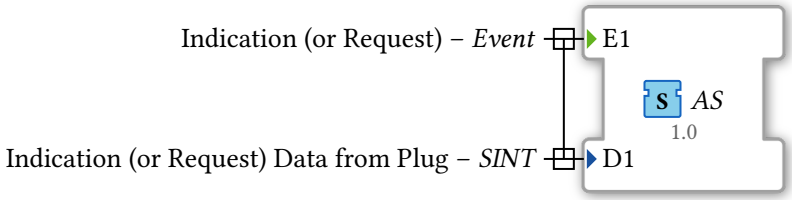
SINT

AS (SINT)

Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



🎧 Podcast

- 4diac IDE: Dein „Hello World“ der Automatisierung – Das Blinking Tutorial Lokal
- 4diac IDE: Dein Open-Source-Werkzeugkasten für verteilte Industrieautomatisierung nach IEC 61499
- Eclipse 4diac FORTE: IEC 61499 verstehen – Der LEGO®-Baukasten für Ihre Industrie 4.0 Steuerung
- Eclipse 4diac: Innovationsmotor Forschung vs. Nutzerbedürfnisse – Was treibt die Entwicklung wirklich voran?
- Automatisierung 4.0: Warum Software die Hardware überholt und was das für deine Skills bedeutet

📺 Video

- Das ESP32-S3-DevKitC-1
- Das Ohmsche Gesetz
- Decoding the datasheet

Interface

Events

Name	Comment	With
E1	Indication (or Request)	D1

Data

Name	Type	Comment
D1	SINT	Indication (or Request) Data from Plug

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 ESP32 & ESP32-S3 DevKit auf ms-muc-docs.de

Unidirectional SINT Adapter

🎧 Podcast

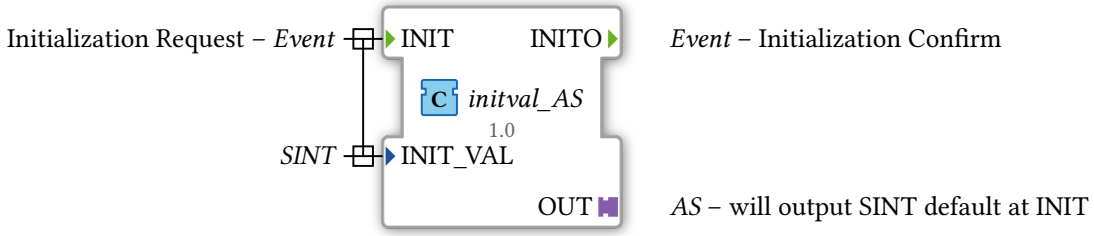
- SINT, INT, DINT: Warum die Wahl des Datentyps über Effizienz und Fehler entscheidet

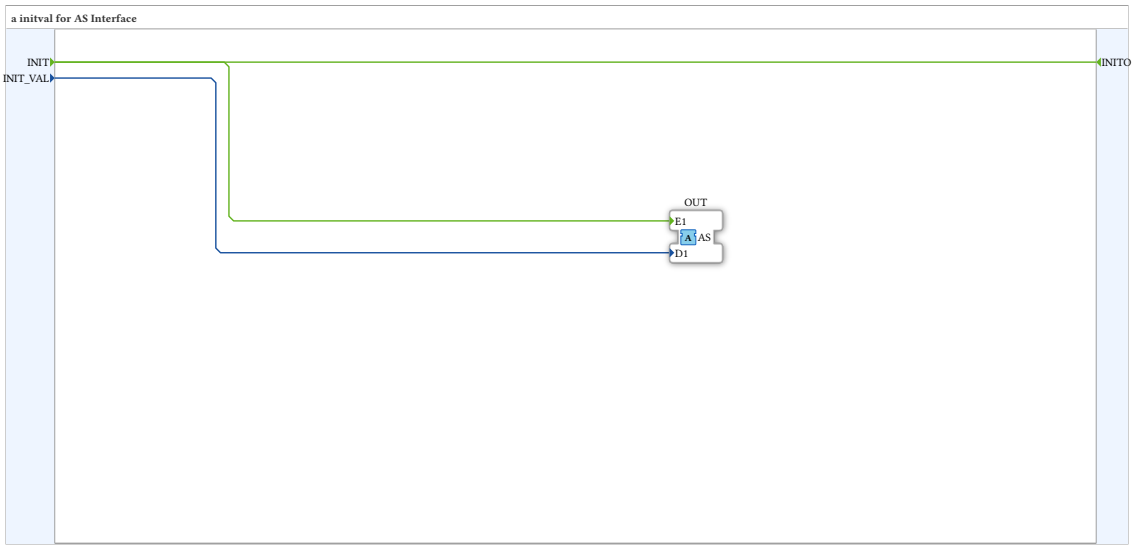
🛠️ Zugehörige Übungen

- SINT_TO_Q_NumericValue

####### Initval

initval_AS





• – ◦ • – ◦ • – ◦ •

Einleitung

`initval_AS` ist der konfigurierbare Initialisierungsbaustein für den AS-Adapter (Datentyp `SINT`, 8-Bit-Ganzzahl (vorzeichenbehaftet)): Er nimmt bei der Initialisierung einen Startwert als Dateneingang entgegen und legt diesen auf einen `AS`-Adapter-Plug. Damit ersetzt er die früher verbreitete Praxis, einen konstanten Vergleichswert über einen gemischten Adapter/Elementar-Baustein einzuspeisen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, liefert `INIT_VAL`. Löst die Ausgabe von `INIT_VAL` über den Plug `OUT` aus.

Ereignis-Ausgänge

- **INITO**: Bestätigt den Abschluss der Initialisierung.

Daten-Eingänge

- **INIT_VAL** (SINT): Der Startwert, der beim Eintreffen von `INIT` auf den Adapter `OUT` geschrieben wird.

Daten-Ausgänge

- Keine direkten Datenausgänge; der Wert wird ausschließlich über den Adapter `OUT` bereitgestellt.

Adapter

- **OUT** (Plug, Typ `AS`): gibt bei `INIT` den über `INIT_VAL` übergebenen Wert über sein `E1`/`D1`-Adapterereignis aus.

Funktionsweise

Bei Eintreffen des Ereignisses `INIT` (mit `With Var="INIT_VAL"`) verknüpft `initval_AS` intern zwei Datenpfade: `INIT_VAL` wird per Datenverbindung auf `OUT.D1` kopiert, und `INIT` wird per Ereignisverbindung an `OUT.E1` weitergereicht – der `OUT`-Plug feuert damit sein Adapterereignis mit dem gerade übergebenen

Wert. Anschließend wird `INIT` zusätzlich an `INITO` weitergereicht, wodurch der Aufrufer den Abschluss der Initialisierung bestätigt bekommt.

Technische Besonderheiten

- Der ausgegebene Wert ist nicht im Baustein selbst parametriert, sondern kommt bei jedem `INIT`-Aufruf frisch über `INIT_VAL` – der Baustein kann also auch außerhalb einer reinen Initialisierungsphase zum dynamischen Setzen eines `AS`-Werts verwendet werden.
- Reine Verdrahtungslogik ohne eigene ECC/Zustandsmaschine (Composite-FB ohne innere Funktionsblöcke); die Datenverbindung `INIT_VAL → OUT.D1` muss vor der Ereignisverbindung ausgewertet sein, damit `OUT.E1` den aktuellen Wert trägt – dies wird durch die 61499-Ausführungsreihenfolge der `FBNetwork`-Verbindungen sichergestellt und ist nicht separat zu verdrahten.

Zustandsübersicht

Der Baustein ist zustandslos: Jedes `INIT`-Ereignis führt unmittelbar zur Übernahme von `INIT_VAL` auf `OUT` und zur Bestätigung über `INITO`.

Anwendungsszenarien

- Initialisierung von `AS`-Adapterverbindungen mit einem projekt- oder parameterabhängigen Startwert (z. B. aus einer Konfigurationsvariable)
- Ersatz für die entfernten „gemischten“ Adapter↔Elementar-Vergleichsbausteine: Ein konstanter `SINT`-Wert wird über `initval_AS` auf einen `AS`-Adapter gelegt und kann dann mit dem reinen adapterbasierten `AS`-Vergleichsbaustein weiterverwendet werden.
- Wiederverwendbare Bausteinvorlage für Testaufbauten, in denen unterschiedliche `AS`-Startwerte durchprobiert werden sollen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit initval_AX, das denselben Verdrahtungsmechanismus für den Datentyp `BOOL` nutzt. Anders als `const_AX_TRUE`/`const_AX_FALSE` (die es nur für `AX` gibt) parametriert `initval_AS` den Ausgabewert nicht fest im Baustein, sondern nimmt ihn über den Dateneingang `INIT_VAL` entgegen.

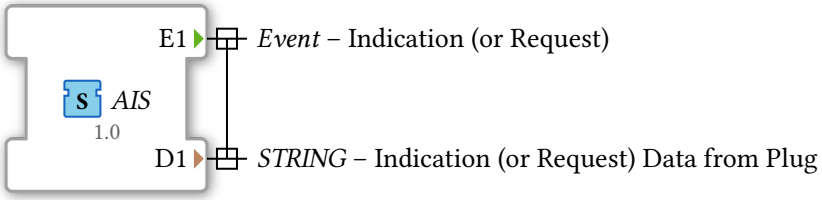
Fazit

`initval_AS` liefert eine einfache, wiederverwendbare Initialisierungslogik für den `AS`-Adapter und macht den ausgegebenen Wert über `INIT_VAL` frei konfigurierbar. Er eignet sich überall dort, wo ein `SINT`-Startwert nicht bereits zur Entwurfszeit feststeht oder als Ersatz für einen entfernten gemischten Vergleichsbaustein benötigt wird.

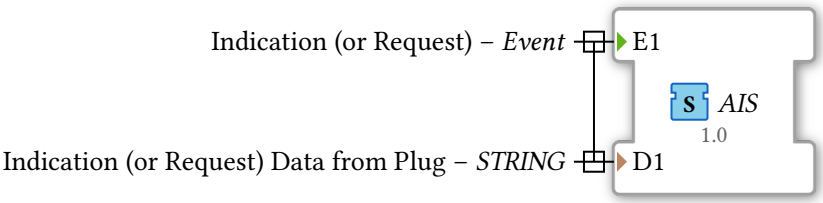
STRING

AIS

Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



Unidirektionaler Adapter für 1 Ereignis und 1 Zeichenkette

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Adapter **AIS** (Adapter Interface String) stellt eine unidirektionale Schnittstelle zur Verfügung, über die genau ein Ereignis (E1) zusammen mit einer zugehörigen Zeichenkette (D1) von einer Plug-Instanz an eine Socket-Instanz übertragen werden kann. Er wird verwendet, um einfache, gerichtete Datenflüsse zu realisieren, bei denen keinerlei Rückmeldung oder zusätzliche Steuerung erforderlich ist.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Name	Typ	Kommentar
E1	Event	Indikation (oder Request) – das Ereignis wird beim Senden der Daten ausgelöst.

Daten-Eingänge

Keine.

Daten-Ausgänge

Name	Typ	Kommentar
D1	STRING	Die zum Ereignis gehörenden Daten (Zeichenkette).

Adapter

Dieser Baustein ist selbst ein Adapter. Er besitzt keine weiteren internen Adapter.

Funktionsweise

Der Adapter arbeitet nach dem Prinzip einer unidirektionalen „Plug- Socket“-Verbindung.

- Die **Plug**-Instanz sendet ein Ereignis an E1 und legt gleichzeitig die Daten auf D1 ab.
- Die **Socket**-Instanz empfängt dieses Ereignis und kann den Wert von D1 auslesen.
  Eine Kommunikation in die Gegenrichtung ist nicht vorgesehen.

Technische Besonderheiten

- **Minimaler Ressourcenverbrauch**: Der Adapter benötigt nur ein Ereignis und eine Zeichenkette – ideal für ressourcenbeschränkte Steuerungen.
- **Typsicherheit**: Der Datentyp ist auf STRING festgelegt; der Adapter kann direkt in Applikationen verwendet werden, die genau diese Signatur benötigen.
- **Einfache Wiederverwendung**: Durch die strikte Trennung in Plug und Socket lassen sich Module leicht kapseln und austauschen.

Zustandsübersicht

Der Adapter besitzt keinen internen Zustandsautomaten. Er verhält sich **zustandslos**: Jedes eingehende Ereignis an der Plug-Seite wird unverzögert als Ausgangsereignis an der Socket-Seite weitergeleitet. Der Datenwert D1 wird dabei zum Zeitpunkt des Ereignisses übernommen.

Anwendungsszenarien

- **Sensor-Anbindung**: Übergabe eines einzelnen Messwerts (z. B. Lauftext eines RFID-Lesers) an eine Verarbeitungseinheit.
- **Monitor-Schnittstelle**: Ausgabe eines Status-Strings aus einer Steuerung an ein Anzeigeelement.
- **Debugging**: Einfaches Herausführen von Diagnoseinformationen aus einem Funktionsblock.

Vergleich mit ähnlichen Bausteinen

- **Komplexere Adapter** wie z. B. Adapter mit mehreren Ereignissen und gemischten Datentypen bieten mehr Flexibilität, benötigen aber auch mehr Konfigurationsaufwand.
- **Bidirektionale Adapter** (z. B. mit je einem Ereignis pro Richtung) erlauben eine Rückmeldung; der AIS-Adapter verzichtet bewusst darauf, um die Schnittstelle so einfach wie möglich zu halten.

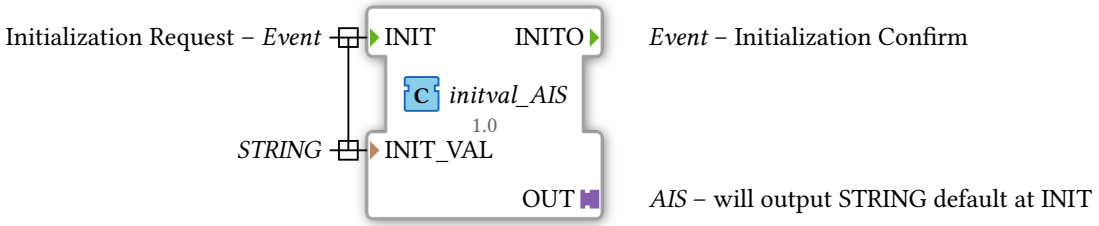
- Der AIS ist speziell auf **genau ein Ereignis + eine Zeichenkette** optimiert – andere Adapter mit gleicher Signatur, aber anderem Namen, könnten funktional identisch sein.

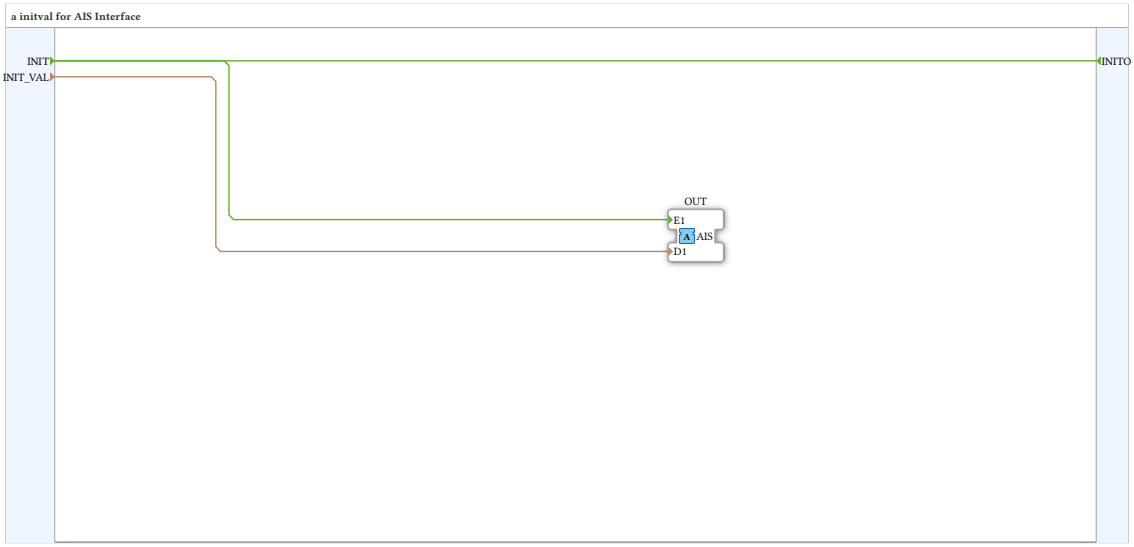
Fazit

Der Adapter **AIS** ist ein schlanker, zustandsloser Baustein für die unidirektionale Übertragung eines Ereignisses zusammen mit einem String. Er eignet sich hervorragend für alle Fälle, in denen eine einfache, klare Signalweiterleitung ohne zusätzliche Logik oder Rückkanal benötigt wird. Durch die minimale Schnittstelle bleibt die Applikation übersichtlich und leicht verständlich.

####### Initval

initval_AIS





• – ◦ • – ◦ • – ◦ •

Einleitung

`initval_AIS` ist der konfigurierbare Initialisierungsbaustein für den AIS-Adapter (Datentyp `STRING`, Zeichenkette): Er nimmt bei der Initialisierung einen Startwert als Dateneingang entgegen und legt diesen auf einen `AIS`-Adapter-Plug. Damit ersetzt er die früher verbreitete Praxis, einen konstanten Vergleichswert über einen gemischten Adapter/Elementar-Baustein einzuspeisen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, liefert `INIT_VAL`. Löst die Ausgabe von `INIT_VAL` über den Plug `OUT` aus.

Ereignis-Ausgänge

- **INITO**: Bestätigt den Abschluss der Initialisierung.

Daten-Eingänge

- **INIT_VAL** (STRING): Der Startwert, der beim Eintreffen von `INIT` auf den Adapter `OUT` geschrieben wird.

Daten-Ausgänge

- Keine direkten Datenausgänge; der Wert wird ausschließlich über den Adapter `OUT` bereitgestellt.

Adapter

- **OUT** (Plug, Typ `AIS`): gibt bei `INIT` den über `INIT_VAL` übergebenen Wert über sein `E1`/`D1`-Adapterereignis aus.

Funktionsweise

Bei Eintreffen des Ereignisses `INIT` (mit `With Var="INIT_VAL"`) verknüpft `initval_AIS` intern zwei Datenpfade: `INIT_VAL` wird per Datenverbindung auf `OUT.D1` kopiert, und `INIT` wird per Ereignisverbindung an `OUT.E1` weitergereicht – der `OUT`-Plug feuert damit sein Adapterereignis mit dem gerade übergebenen Wert. Anschließend wird `INIT` zusätzlich an `INITO` weitergereicht, wodurch der Aufrufer den Abschluss der Initialisierung bestätigt bekommt.

Technische Besonderheiten

- Der ausgegebene Wert ist nicht im Baustein selbst parametriert, sondern kommt bei jedem `INIT`-Aufruf frisch über `INIT_VAL` – der Baustein kann also auch außerhalb einer reinen Initialisierungsphase zum dynamischen Setzen eines `AIS`-Werts verwendet werden.
- Reine Verdrahtungslogik ohne eigene ECC/Zustandsmaschine (Composite-FB ohne innere Funktionsblöcke); die Datenverbindung `INIT_VAL → OUT.D1` muss vor der Ereignisverbindung ausgewertet sein, damit `OUT.E1` den aktuellen Wert trägt – dies wird durch die 61499-Ausführungsreihenfolge der `FBNetwork`-Verbindungen sichergestellt und ist nicht separat zu verdrahten.

Zustandsübersicht

Der Baustein ist zustandslos: Jedes `INIT`-Ereignis führt unmittelbar zur Übernahme von `INIT_VAL` auf `OUT` und zur Bestätigung über `INITO`.

Anwendungsszenarien

- Initialisierung von `AIS`-Adapterverbindungen mit einem projekt- oder parameterabhängigen Startwert (z. B. aus einer Konfigurationsvariable)
- Ersatz für die entfernten „gemischten“ Adapter↔Elementar-Vergleichsbausteine: Ein konstanter `STRING`-Wert wird über `initval_AIS` auf einen `AIS`-Adapter gelegt und kann dann mit dem reinen adapterbasierten `AIS`-Vergleichsbaustein weiterverwendet werden.
- Wiederverwendbare Bausteinvorlage für Testaufbauten, in denen unterschiedliche `AIS`-Startwerte durchprobiert werden sollen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit initval_AX, das denselben Verdrahtungsmechanismus für den Datentyp `BOOL` nutzt. Anders als `const_AX_TRUE`/`const_AX_FALSE` (die es nur für `AX` gibt) parametriert `initval_AIS` den Ausgabewert nicht fest im Baustein, sondern nimmt ihn über den Dateneingang `INIT_VAL` entgegen.

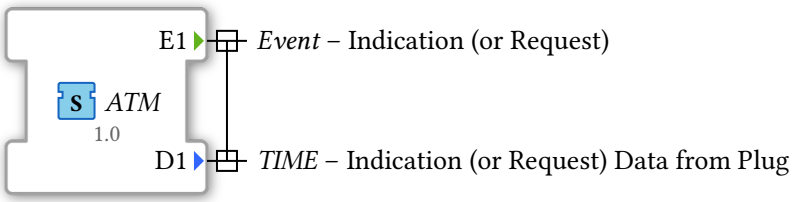
Fazit

`initval_AIS` liefert eine einfache, wiederverwendbare Initialisierungslogik für den `AIS`-Adapter und macht den ausgegebenen Wert über `INIT_VAL` frei konfigurierbar. Er eignet sich überall dort, wo ein `STRING`-Startwert nicht bereits zur Entwurfszeit feststeht oder als Ersatz für einen entfernten gemischten Vergleichsbaustein benötigt wird.

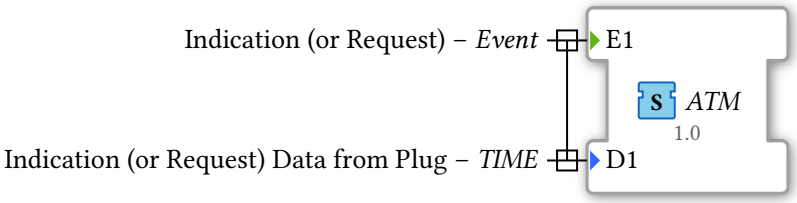
TIME

ATM

Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **ATM-Adapter** (Abkürzung für „Adapter Time Message“) ist ein unidirektionaler Typ nach IEC 61499-1, der eine ereignisgesteuerte Schnittstelle zur Übertragung eines Zeitwertes bereitstellt. Er wird als Adapter verwendet, um zwischen Funktionsbausteinen (FBs) eine standardisierte, lose gekoppelte Kommunikation zu ermöglichen. Entwickelt wurde dieser Adapter von der HR Agrartechnik GmbH und ist unter der Eclipse Public License 2.0 (EPL-2.0) verfügbar.

Schnittstellenstruktur

Ereignis-Eingänge

Der Adapter besitzt keine Ereignis-Eingänge.

Ereignis-Ausgänge

Name	Typ	Kommentar
E1	Event	Indikation (oder Anforderung) – wird zusammen mit dem Datenwert D1 gesendet

Daten-Eingänge

Der Adapter besitzt keine Daten-Eingänge.

Daten-Ausgänge

Name	Typ	Kommentar
D1	TIME	Indikations-/Anforderungsdaten vom Plug

Adapter

Der Baustein selbst ist vom Typ `AdapterType`. Es sind keine weiteren Adapter-Schnittstellen enthalten.

Funktionsweise

Der ATM-Adapter arbeitet ausschließlich als unidirektionale Schnittstelle. Er wird in einen Funktionsbaustein als **Plug** (oder als **Socket**) eingebunden und überträgt bei Auftreten des Ereignisses **E1** den zugehörigen Zeitwert **D1** (Typ `TIME`) an die verbundene Gegenstelle. Die Übertragung erfolgt in einer Richtung: Vom sendenden zum empfangenden Baustein. Die Kopplung wird automatisch durch die 4diac-IDE-Umgebung aufgelöst.

Technische Besonderheiten

- **Unidirektionaler Adapter**: Nur eine Daten- und Ereignisrichtung (Plug ⟶ Socket).
- **Einfachste Ausprägung**: Genau ein Ereignis und eine Variable – minimaler Overhead.
- **Typisierung**: Die übertragene Variable ist vom elementaren Typ `TIME`, geeignet für Zeitstempel, Verzögerungen oder Zeitmessungen.
- **Lizenz**: Veröffentlicht unter EPL-2.0, somit frei nutzbar und erweiterbar.
- Keine Zustandsmaschine oder interne Logik – reines Schnittstelleninterface.

Zustandsübersicht

Da der ATM-Adapter keine interne Zustandsmaschine besitzt (zustandsloser Adapter), ist keine Zustandsübersicht erforderlich. Die Funktionalität beschränkt sich auf die reine Ereignis-Daten-Weitergabe.

Anwendungsszenarien

- **Übermittlung von Zeitstempeln** zwischen verschiedenen Funktionsbausteinen (z. B. Protokollierung, Steuerung mit Zeitbezug).
- **Signal für zeitgesteuerte Aktionen**: Ein Baustein sendet über den Adapter eine Zeitinformation, die von einem anderen Baustein ausgewertet wird.
- **Testumgebungen**: Einfaches, klar definiertes Interface für Unit-Tests von Bausteinen, die Zeitwerte verarbeiten.

Vergleich mit ähnlichen Bausteinen

Merkmal	ATM (dieser Adapter)	Andere unidirektionale Adapter (z. B. mit mehreren Events/Daten)
Ereignisse	1 (E1)	meist mehrere (z. B. INIT, REQ, IND, RSP)
Daten	1 (TIME)	oft mehrere Daten oder verschiedene Typen
Komplexität	gering	mittel bis hoch
Einsatz	einfache Zeitübertragung	komplexe Prozessschriftstellen

Gegenüber allgemeineren Adaptern beschränkt sich ATM auf das Nötigste – ideal, wenn nur ein Ereignis mit einem einzigen Zeitwert ausgetauscht werden muss.

Fazit

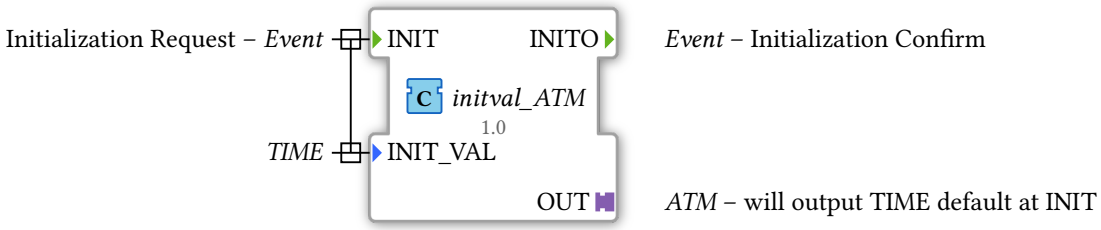
Der ATM-Adapter ist ein minimaler, aber praxisrelevanter Baustein zur unidirektionalen Übertragung von Zeitwerten in IEC 61499-Systemen. Seine klare Schnittstelle und einfache Handhabung machen ihn besonders geeignet für Anwendungen, die einen Fokus auf zeitbezogene Kommunikation legen. Durch die offene Lizenzierung kann der Adapter ohne Einschränkungen in eigene Projekte integriert oder erweitert werden.

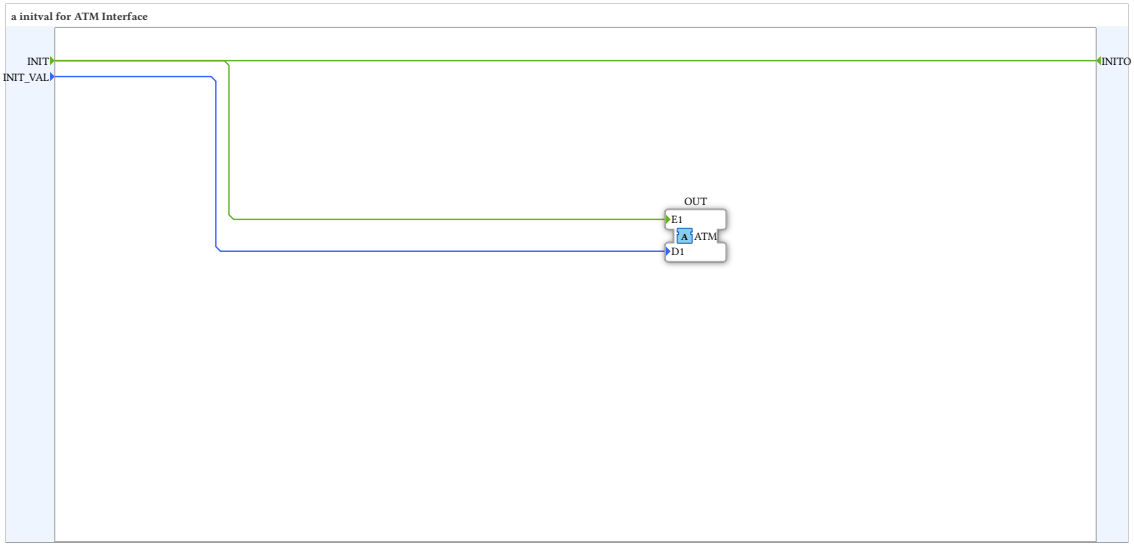
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

####### Initval

initval_ATM





• – ◦ • – ◦ • – ◦ •

Einleitung

`initval_ATM` ist der konfigurierbare Initialisierungsbaustein für den ATM-Adapter (Datentyp `TIME`, Zeitdauer): Er nimmt bei der Initialisierung einen Startwert als Dateneingang entgegen und legt diesen auf einen `ATM`-Adapter-Plug. Damit ersetzt er die früher verbreitete Praxis, einen konstanten Vergleichswert über einen gemischten Adapter/Elementar-Baustein einzuspeisen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, liefert `INIT_VAL`. Löst die Ausgabe von `INIT_VAL` über den Plug `OUT` aus.

Ereignis-Ausgänge

- **INITO**: Bestätigt den Abschluss der Initialisierung.

Daten-Eingänge

- **INIT_VAL** (TIME): Der Startwert, der beim Eintreffen von `INIT` auf den Adapter `OUT` geschrieben wird.

Daten-Ausgänge

- Keine direkten Datenausgänge; der Wert wird ausschließlich über den Adapter `OUT` bereitgestellt.

Adapter

- **OUT** (Plug, Typ `ATM`): gibt bei `INIT` den über `INIT_VAL` übergebenen Wert über sein `E1`/`D1`-Adapterereignis aus.

Funktionsweise

Bei Eintreffen des Ereignisses `INIT` (mit `With Var="INIT_VAL"`) verknüpft `initval_ATM` intern zwei Datenpfade: `INIT_VAL` wird per Datenverbindung auf `OUT.D1` kopiert, und `INIT` wird per Ereignisverbindung an `OUT.E1` weitergereicht – der `OUT`-Plug feuert damit sein Adapterereignis mit dem gerade übergebenen Wert. Anschließend wird `INIT` zusätzlich an `INITO` weitergereicht, wodurch der Aufrufer den Abschluss der Initialisierung bestätigt bekommt.

Technische Besonderheiten

- Der ausgegebene Wert ist nicht im Baustein selbst parametriert, sondern kommt bei jedem `INIT`-Aufruf frisch über `INIT_VAL` – der Baustein kann also auch außerhalb einer reinen Initialisierungsphase zum dynamischen Setzen eines `ATM`-Werts verwendet werden.
- Reine Verdrahtungslogik ohne eigene ECC/Zustandsmaschine (Composite-FB ohne innere Funktionsblöcke); die Datenverbindung `INIT_VAL → OUT.D1` muss vor der Ereignisverbindung ausgewertet

sein, damit `OUT.E1` den aktuellen Wert trägt – dies wird durch die 61499-Ausführungsreihenfolge der `FBNetwork`-Verbindungen sichergestellt und ist nicht separat zu verdrahten.

Zustandsübersicht

Der Baustein ist zustandslos: Jedes `INIT`-Ereignis führt unmittelbar zur Übernahme von `INIT_VAL` auf `OUT` und zur Bestätigung über `INITO`.

Anwendungsszenarien

- Initialisierung von `ATM`-Adapterverbindungen mit einem projekt- oder parameterabhängigen Startwert (z. B. aus einer Konfigurationsvariable)
- Ersatz für die entfernten „gemischten" Adapter↔Elementar-Vergleichsbausteine: Ein konstanter `TIME`-Wert wird über `initval_ATM` auf einen `ATM`-Adapter gelegt und kann dann mit dem reinen adapterbasierten `ATM`-Vergleichsbaustein weiterverwendet werden.
- Wiederverwendbare Bausteinvorlage für Testaufbauten, in denen unterschiedliche `ATM`-Startwerte durchprobiert werden sollen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit initval_AX, das denselben Verdrahtungsmechanismus für den Datentyp `BOOL` nutzt. Anders als `const_AX_TRUE`/`const_AX_FALSE` (die es nur für `AX` gibt) parametriert `initval_ATM` den Ausgabewert nicht fest im Baustein, sondern nimmt ihn über den Dateneingang `INIT_VAL` entgegen.

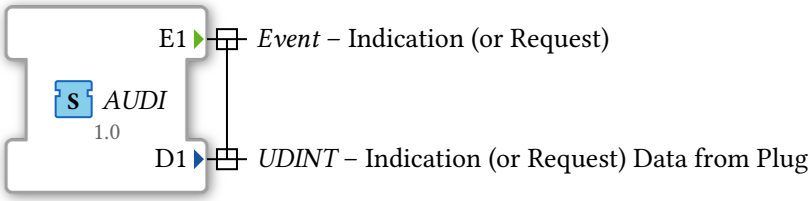
Fazit

`initval_ATM` liefert eine einfache, wiederverwendbare Initialisierungslogik für den `ATM`-Adapter und macht den ausgegebenen Wert über `INIT_VAL` frei konfigurierbar. Er eignet sich überall dort, wo ein `TIME`-Startwert nicht bereits zur Entwurfszeit feststeht oder als Ersatz für einen entfernten gemischten Vergleichsbaustein benötigt wird.

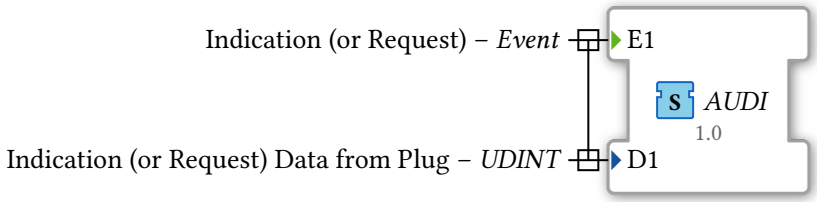
UDINT

AUDI (UDINT)

Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



unidirectional Adapter Interface for 1 Event and 1 Udint

Interface

Events

Name	Comment	With
E1	Indication (or Request)	D1

Data

Name	Type	Comment
D1	UDINT	Indication (or Request) Data from Plug

🛠️ Zugehörige Übungen

- Uebung_009_AX
- Uebung_011_AUDI
- Uebung_083_AX

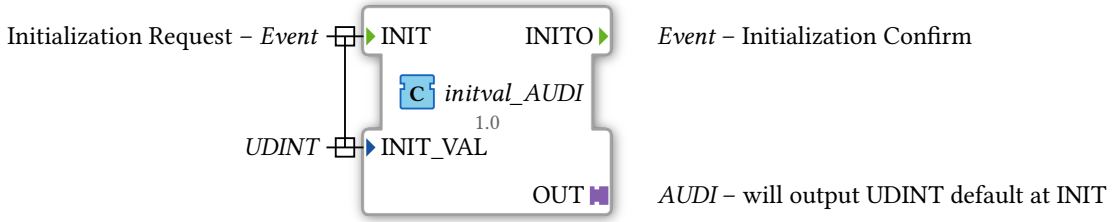
Unidirectional UDINT Adapter

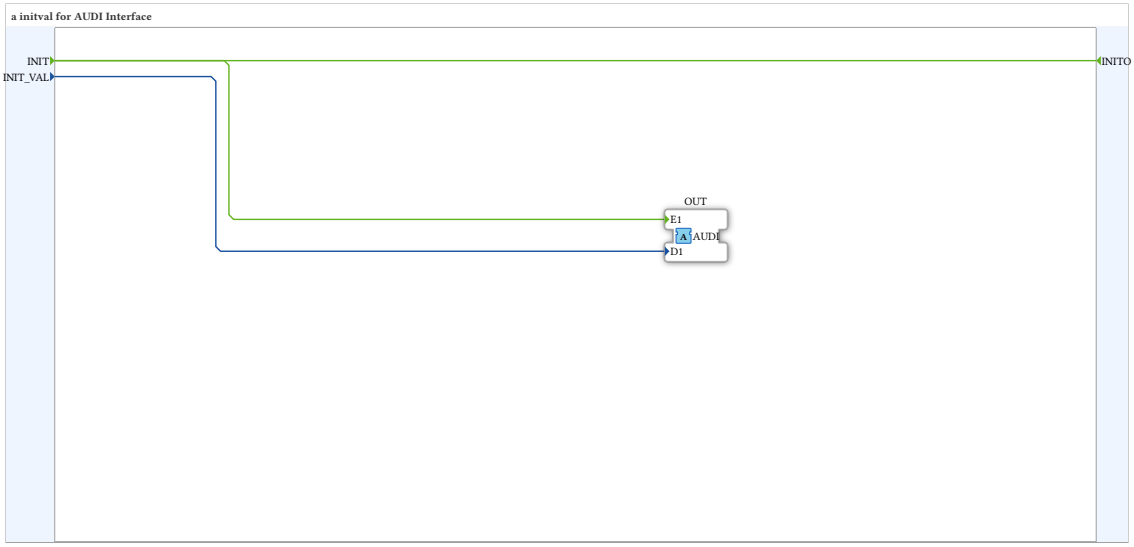
🛠️ Zugehörige Übungen

- Uebung_009
- Uebung_009_AX
- Uebung_009a
- Uebung_011
- Uebung_011a
- Uebung_011a2
- Uebung_012
- NumericValue_ID_TO_NVS
- Uebung_012b
- Uebung_012c
- Uebung_015
- Uebung_015a
- Uebung_020c2_sub
- Uebung_028
- Uebung_034
- Uebung_034a1_Q1
- Uebung_034a1_Q2
- Uebung_034a1_Q4
- Uebung_034b
- Uebung_072c
- Uebung_083
- Uebung_083_AX
- Uebung_103
- Uebung_111
- Uebung_151_AX
- Uebung_152
- Uebung_153

####### Initval

initval_AUDI





• – ◦ • – ◦ • – ◦ •

Einleitung

`initval_AUDI` ist der konfigurierbare Initialisierungsbaustein für den AUDI-Adapter (Datentyp `UDINT`, 32-Bit-Ganzzahl (vorzeichenlos)): Er nimmt bei der Initialisierung einen Startwert als Dateneingang entgegen und legt diesen auf einen `AUDI`-Adapter-Plug. Damit ersetzt er die früher verbreitete Praxis, einen konstanten Vergleichswert über einen gemischten Adapter/Elementar-Baustein einzuspeisen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, liefert `INIT_VAL`. Löst die Ausgabe von `INIT_VAL` über den Plug `OUT` aus.

Ereignis-Ausgänge

- **INITO**: Bestätigt den Abschluss der Initialisierung.

Daten-Eingänge

- **INIT_VAL** (UDINT): Der Startwert, der beim Eintreffen von `INIT` auf den Adapter `OUT` geschrieben wird.

Daten-Ausgänge

- Keine direkten Datenausgänge; der Wert wird ausschließlich über den Adapter `OUT` bereitgestellt.

Adapter

- **OUT** (Plug, Typ `AUDI`): gibt bei `INIT` den über `INIT_VAL` übergebenen Wert über sein `E1`/`D1`-Adapterereignis aus.

Funktionsweise

Bei Eintreffen des Ereignisses `INIT` (mit `With Var="INIT_VAL"`) verknüpft `initval_AUDI` intern zwei Datenpfade: `INIT_VAL` wird per Datenverbindung auf `OUT.D1` kopiert, und `INIT` wird per Ereignisverbindung an `OUT.E1` weitergereicht – der `OUT`-Plug feuert damit sein Adapterereignis mit dem gerade übergebenen Wert. Anschließend wird `INIT` zusätzlich an `INITO` weitergereicht, wodurch der Aufrufer den Abschluss der Initialisierung bestätigt bekommt.

Technische Besonderheiten

- Der ausgegebene Wert ist nicht im Baustein selbst parametriert, sondern kommt bei jedem `INIT`-Aufruf frisch über `INIT_VAL` – der Baustein kann also auch außerhalb einer reinen Initialisierungsphase zum dynamischen Setzen eines `AUDI`-Werts verwendet werden.
- Reine Verdrahtungslogik ohne eigene ECC/Zustandsmaschine (Composite-FB ohne innere Funktionsblöcke); die Datenverbindung `INIT_VAL → OUT.D1` muss vor der Ereignisverbindung ausgewertet

sein, damit `OUT.E1` den aktuellen Wert trägt – dies wird durch die 61499-Ausführungsreihenfolge der `FBNetwork`-Verbindungen sichergestellt und ist nicht separat zu verdrahten.

Zustandsübersicht

Der Baustein ist zustandslos: Jedes `INIT`-Ereignis führt unmittelbar zur Übernahme von `INIT_VAL` auf `OUT` und zur Bestätigung über `INITO`.

Anwendungsszenarien

- Initialisierung von `AUDI`-Adapterverbindungen mit einem projekt- oder parameterabhängigen Startwert (z. B. aus einer Konfigurationsvariable)
- Ersatz für die entfernten „gemischten“ Adapter↔Elementar-Vergleichsbausteine: Ein konstanter `UDINT`-Wert wird über `initval_AUDI` auf einen `AUDI`-Adapter gelegt und kann dann mit dem reinen adapterbasierten `AUDI`-Vergleichsbaustein weiterverwendet werden.
- Wiederverwendbare Bausteinvorlage für Testaufbauten, in denen unterschiedliche `AUDI`-Startwerte durchprobiert werden sollen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit initval_AX, das denselben Verdrahtungsmechanismus für den Datentyp `BOOL` nutzt. Anders als `const_AX_TRUE`/`const_AX_FALSE` (die es nur für `AX` gibt) parametriert `initval_AUDI` den Ausgabewert nicht fest im Baustein, sondern nimmt ihn über den Dateneingang `INIT_VAL` entgegen.

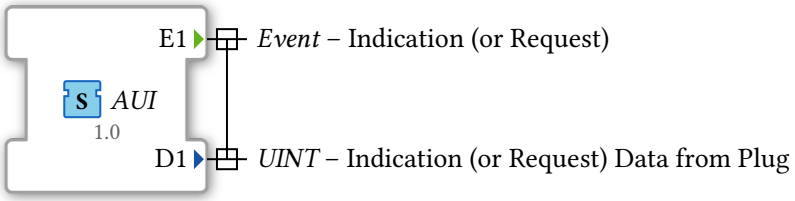
Fazit

`initval_AUDI` liefert eine einfache, wiederverwendbare Initialisierungslogik für den `AUDI`-Adapter und macht den ausgegebenen Wert über `INIT_VAL` frei konfigurierbar. Er eignet sich überall dort, wo ein `UDINT`-Startwert nicht bereits zur Entwurfszeit feststeht oder als Ersatz für einen entfernten gemischten Vergleichsbaustein benötigt wird.

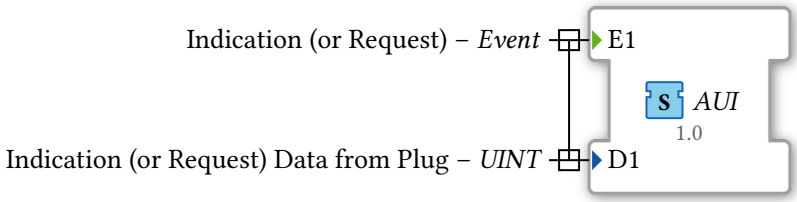
UINT

AUI (UINT)

Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



unidirectional Adapter Interface for 1 Event and 1 Uint

Interface

Events

Name	Comment	With
E1	Indication (or Request)	D1

Data

Name	Type	Comment
D1	UINT	Indication (or Request) Data from Plug

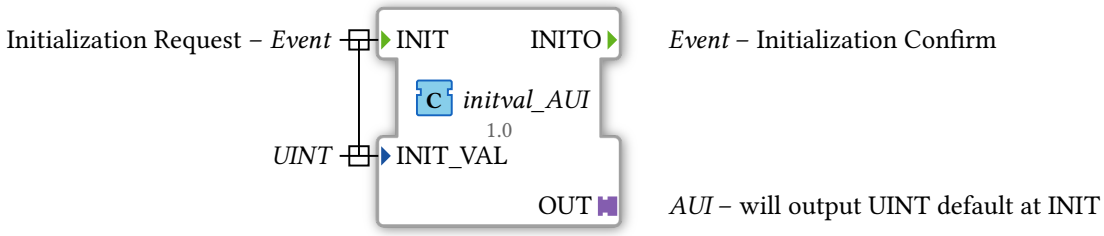
Unidirectional UINT Adapter

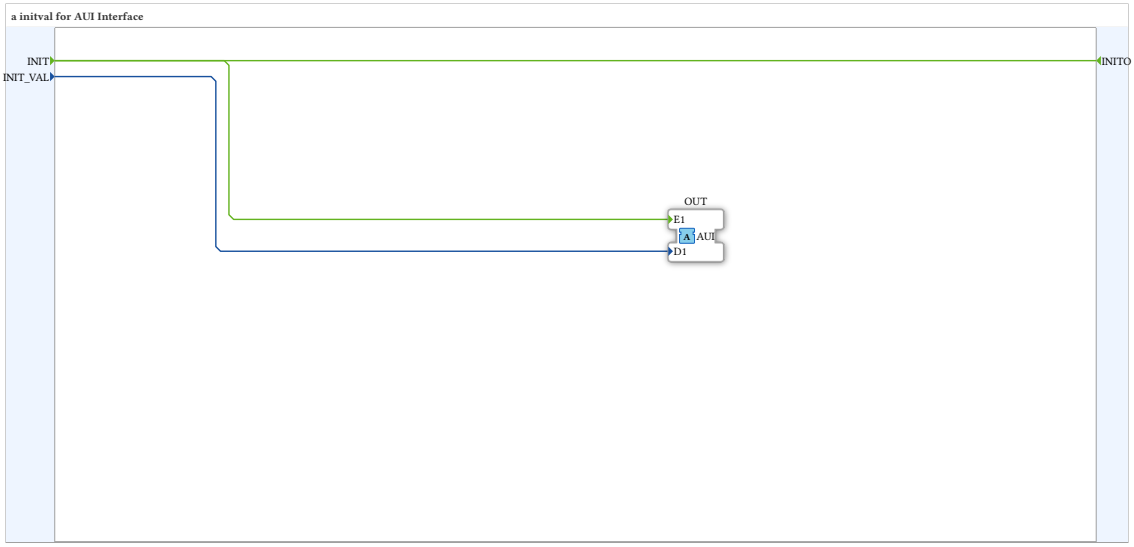
🛠️ Zugehörige Übungen

- Uebung_006c
- Uebung_010b4_sub_AX
- Uebung_010b5_sub_AX
- Uebung_010c2
- Uebung_010c2_AX
- AuxIXA_TO_logiBUS_QXA_BG
- NumericValue_ID_TO_NVS
- Uebung_020c2_sub
- IX_TO_logiBUS_LED_strip_QX
- Uebung_035
- Uebung_036
- Uebung_037
- Uebung_038
- Uebung_038_AX
- SINT_TO_Q_NumericValue
- SoftkeyOverride_TO_QX_BG
- Softkey_T_FF_ILOCK_TO_QX_BG
- Uebung_040
- Uebung_040_AX
- Uebung_041
- Uebung_087
- Uebung_087a1
- Uebung_090a1
- Uebung_090a1_AX
- Uebung_090a2
- Uebung_090a2_AX
- Uebung_103

####### Initval

initval_AUI





• – ◦ • – ◦ • – ◦ •

Einleitung

`initval_AUI` ist der konfigurierbare Initialisierungsbaustein für den AUI-Adapter (Datentyp `UINT`, 16-Bit-Ganzzahl (vorzeichenlos)): Er nimmt bei der Initialisierung einen Startwert als Dateneingang entgegen und legt diesen auf einen `AUI`-Adapter-Plug. Damit ersetzt er die früher verbreitete Praxis, einen konstanten Vergleichswert über einen gemischten Adapter/Elementar-Baustein einzuspeisen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, liefert `INIT_VAL`. Löst die Ausgabe von `INIT_VAL` über den Plug `OUT` aus.

Ereignis-Ausgänge

- **INITO**: Bestätigt den Abschluss der Initialisierung.

Daten-Eingänge

- **INIT_VAL** (UINT): Der Startwert, der beim Eintreffen von `INIT` auf den Adapter `OUT` geschrieben wird.

Daten-Ausgänge

- Keine direkten Datenausgänge; der Wert wird ausschließlich über den Adapter `OUT` bereitgestellt.

Adapter

- **OUT** (Plug, Typ `AUI`): gibt bei `INIT` den über `INIT_VAL` übergebenen Wert über sein `E1`/`D1`-Adapterereignis aus.

Funktionsweise

Bei Eintreffen des Ereignisses `INIT` (mit `With Var="INIT_VAL"`) verknüpft `initval_AUI` intern zwei Datenpfade: `INIT_VAL` wird per Datenverbindung auf `OUT.D1` kopiert, und `INIT` wird per Ereignisverbindung an `OUT.E1` weitergereicht – der `OUT`-Plug feuert damit sein Adapterereignis mit dem gerade übergebenen Wert. Anschließend wird `INIT` zusätzlich an `INITO` weitergereicht, wodurch der Aufrufer den Abschluss der Initialisierung bestätigt bekommt.

Technische Besonderheiten

- Der ausgegebene Wert ist nicht im Baustein selbst parametriert, sondern kommt bei jedem `INIT`-Aufruf frisch über `INIT_VAL` – der Baustein kann also auch außerhalb einer reinen Initialisierungsphase zum dynamischen Setzen eines `AUI`-Werts verwendet werden.
- Reine Verdrahtungslogik ohne eigene ECC/Zustandsmaschine (Composite-FB ohne innere Funktionsblöcke); die Datenverbindung `INIT_VAL → OUT.D1` muss vor der Ereignisverbindung ausgewertet

sein, damit `OUT.E1` den aktuellen Wert trägt – dies wird durch die 61499-Ausführungsreihenfolge der `FBNetwork`-Verbindungen sichergestellt und ist nicht separat zu verdrahten.

Zustandsübersicht

Der Baustein ist zustandslos: Jedes `INIT`-Ereignis führt unmittelbar zur Übernahme von `INIT_VAL` auf `OUT` und zur Bestätigung über `INITO`.

Anwendungsszenarien

- Initialisierung von `AUI`-Adapterverbindungen mit einem projekt- oder parameterabhängigen Startwert (z. B. aus einer Konfigurationsvariable)
- Ersatz für die entfernten „gemischten" Adapter↔Elementar-Vergleichsbausteine: Ein konstanter `UINT`-Wert wird über `initval_AUI` auf einen `AUI`-Adapter gelegt und kann dann mit dem reinen adapterbasierten `AUI`-Vergleichsbaustein weiterverwendet werden.
- Wiederverwendbare Bausteinvorlage für Testaufbauten, in denen unterschiedliche `AUI`-Startwerte durchprobiert werden sollen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit initval_AX, das denselben Verdrahtungsmechanismus für den Datentyp `BOOL` nutzt. Anders als `const_AX_TRUE`/`const_AX_FALSE` (die es nur für `AX` gibt) parametriert `initval_AUI` den Ausgabewert nicht fest im Baustein, sondern nimmt ihn über den Dateneingang `INIT_VAL` entgegen.

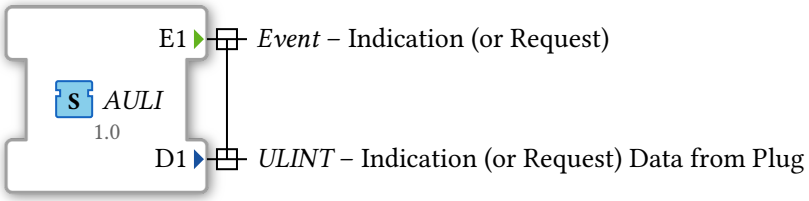
Fazit

`initval_AUI` liefert eine einfache, wiederverwendbare Initialisierungslogik für den `AUI`-Adapter und macht den ausgegebenen Wert über `INIT_VAL` frei konfigurierbar. Er eignet sich überall dort, wo ein `UINT`-Startwert nicht bereits zur Entwurfszeit feststeht oder als Ersatz für einen entfernten gemischten Vergleichsbaustein benötigt wird.

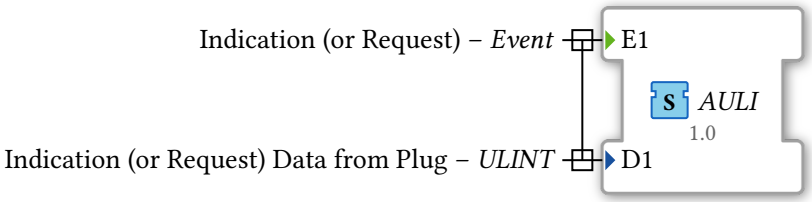
ULINT

AULI (ULINT)

Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



🎧 Podcast

- Außenzahnradmaschinen: Vom Arbeitstier zur Intelligenz der Hydraulik – Herausforderungen, Innovationen & Keplers Erbe
- Danfoss und Eaton Hydraulik Zwillingsgeschichte endet
- Der unterschätzte Held: Warum der Ölbehälter Ihr Hydrauliksystem revolutioniert
- Druckbegrenzungsventile: Lebensversicherung der Hydraulik – Arten, Funktion und Systemintegration
- Kraftpakete im Einsatz: Das Geheimnis der Hydraulikzylinder – Von Baggern bis Hightech-Maschinen

Interface

Events

Name	Comment	With
E1	Indication (or Request)	D1

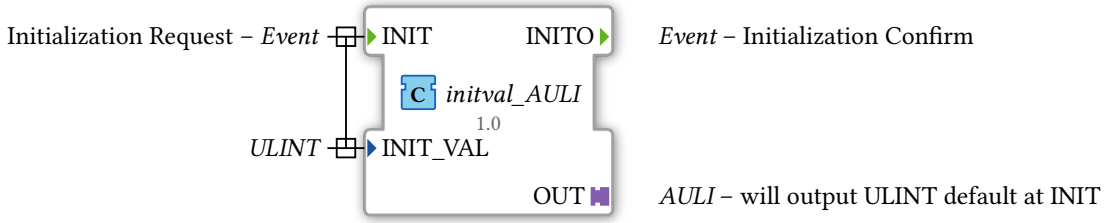
Data

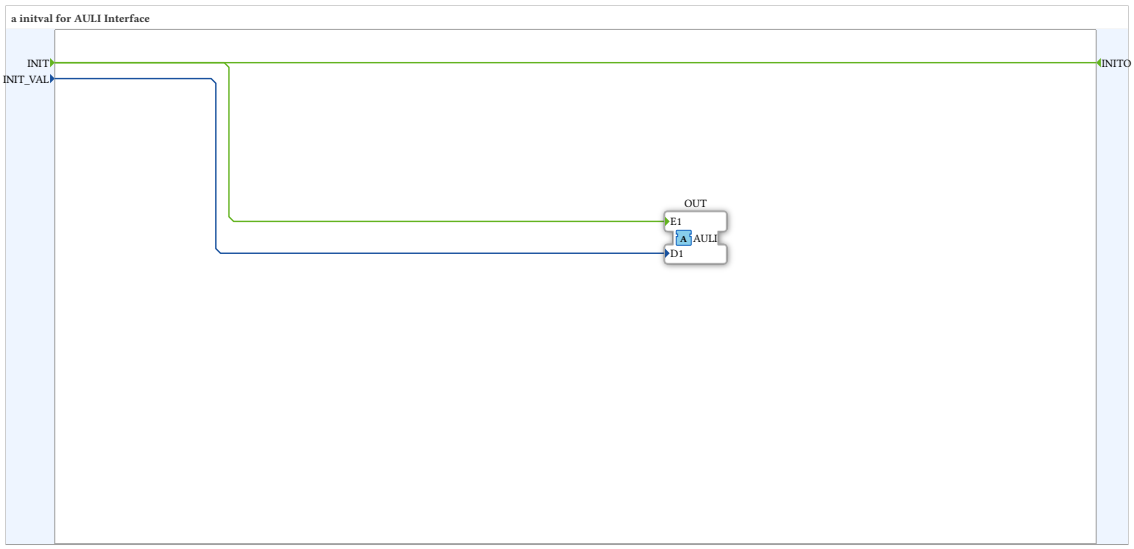
Name	Type	Comment
D1	ULINT	Indication (or Request) Data from Plug

Unidirectional ULINT Adapter

####### Initval

initval_AULI





• – ◦ • – ◦ • – ◦ •

Einleitung

`initval_AULI` ist der konfigurierbare Initialisierungsbaustein für den AULI-Adapter (Datentyp `ULINT`, 64-Bit-Ganzzahl (vorzeichenlos)): Er nimmt bei der Initialisierung einen Startwert als Dateneingang entgegen und legt diesen auf einen `AULI`-Adapter-Plug. Damit ersetzt er die früher verbreitete Praxis, einen konstanten Vergleichswert über einen gemischten Adapter/Elementar-Baustein einzuspeisen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, liefert `INIT_VAL`. Löst die Ausgabe von `INIT_VAL` über den Plug `OUT` aus.

Ereignis-Ausgänge

- **INITO**: Bestätigt den Abschluss der Initialisierung.

Daten-Eingänge

- **INIT_VAL** (ULINT): Der Startwert, der beim Eintreffen von `INIT` auf den Adapter `OUT` geschrieben wird.

Daten-Ausgänge

- Keine direkten Datenausgänge; der Wert wird ausschließlich über den Adapter `OUT` bereitgestellt.

Adapter

- **OUT** (Plug, Typ `AULI`): gibt bei `INIT` den über `INIT_VAL` übergebenen Wert über sein `E1`/`D1`-Adapterereignis aus.

Funktionsweise

Bei Eintreffen des Ereignisses `INIT` (mit `With Var="INIT_VAL"`) verknüpft `initval_AULI` intern zwei Datenpfade: `INIT_VAL` wird per Datenverbindung auf `OUT.D1` kopiert, und `INIT` wird per Ereignisverbindung an `OUT.E1` weitergereicht – der `OUT`-Plug feuert damit sein Adapterereignis mit dem gerade übergebenen Wert. Anschließend wird `INIT` zusätzlich an `INITO` weitergereicht, wodurch der Aufrufer den Abschluss der Initialisierung bestätigt bekommt.

Technische Besonderheiten

- Der ausgegebene Wert ist nicht im Baustein selbst parametriert, sondern kommt bei jedem `INIT`-Aufruf frisch über `INIT_VAL` – der Baustein kann also auch außerhalb einer reinen Initialisierungsphase zum dynamischen Setzen eines `AULI`-Werts verwendet werden.
- Reine Verdrahtungslogik ohne eigene ECC/Zustandsmaschine (Composite-FB ohne innere Funktionsblöcke); die Datenverbindung `INIT_VAL → OUT.D1` muss vor der Ereignisverbindung ausgewertet sein, damit `OUT.E1` den aktuellen Wert trägt – dies wird durch die 61499-Ausführungsreihenfolge der `FBNetwork`-Verbindungen sichergestellt und ist nicht separat zu verdrahten.

Zustandsübersicht

Der Baustein ist zustandslos: Jedes `INIT`-Ereignis führt unmittelbar zur Übernahme von `INIT_VAL` auf `OUT` und zur Bestätigung über `INITO`.

Anwendungsszenarien

- Initialisierung von `AULI`-Adapterverbindungen mit einem projekt- oder parameterabhängigen Startwert (z. B. aus einer Konfigurationsvariable)
- Ersatz für die entfernten „gemischten“ Adapter↔Elementar-Vergleichsbausteine: Ein konstanter `ULINT`-Wert wird über `initval_AULI` auf einen `AULI`-Adapter gelegt und kann dann mit dem reinen adapterbasierten `AULI`-Vergleichsbaustein weiterverwendet werden.
- Wiederverwendbare Bausteinvorlage für Testaufbauten, in denen unterschiedliche `AULI`-Startwerte durchprobiert werden sollen

⚖️ Vergleich mit ähnlichen Bausteinen

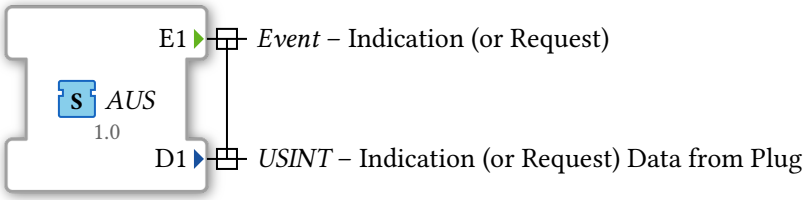
Vergleich mit initval_AX, das denselben Verdrahtungsmechanismus für den Datentyp `BOOL` nutzt. Anders als `const_AX_TRUE`/`const_AX_FALSE` (die es nur für `AX` gibt) parametriert `initval_AULI` den Ausgabewert nicht fest im Baustein, sondern nimmt ihn über den Dateneingang `INIT_VAL` entgegen.

Fazit

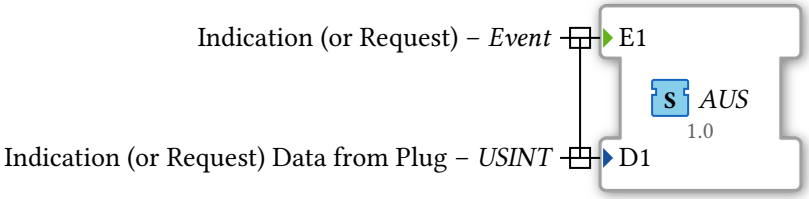
`initval_AULI` liefert eine einfache, wiederverwendbare Initialisierungslogik für den `AULI`-Adapter und macht den ausgegebenen Wert über `INIT_VAL` frei konfigurierbar. Er eignet sich überall dort, wo ein `ULINT`-Startwert nicht bereits zur Entwurfszeit feststeht oder als Ersatz für einen entfernten gemischten Vergleichsbaustein benötigt wird.

USINT

AUS (USINT)
Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



🎧 Podcast

- Der E_CTU-Baustein: Ereignisgesteuertes Hochzählen in der Industrie nach IEC 61499
- Der E_PERMIT-Baustein: Der „Türsteher“ für Ereignisse in IEC 61499-Systemen entschlüsselt
- Der E_T_FF_SR-Baustein: Herzstück der IEC 61499 – Speichern, Umschalten, Reagieren
- DIN EN 61499-1 Entmystifiziert: Funktionsbausteine, Objektorientierung und verteilte Systeme
- DIN EN 61499-1: Bauklötze für die Industrie 4.0 – So revolutionieren Funktionsbausteine die Automatisierung

Interface

Events

Name	Comment	With
E1	Indication (or Request)	D1

Data

Name	Type	Comment
D1	USINT	Indication (or Request) Data from Plug

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de

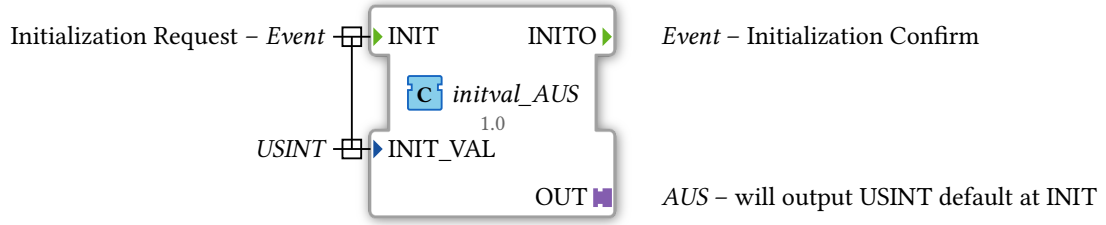
Unidirectional USINT Adapter

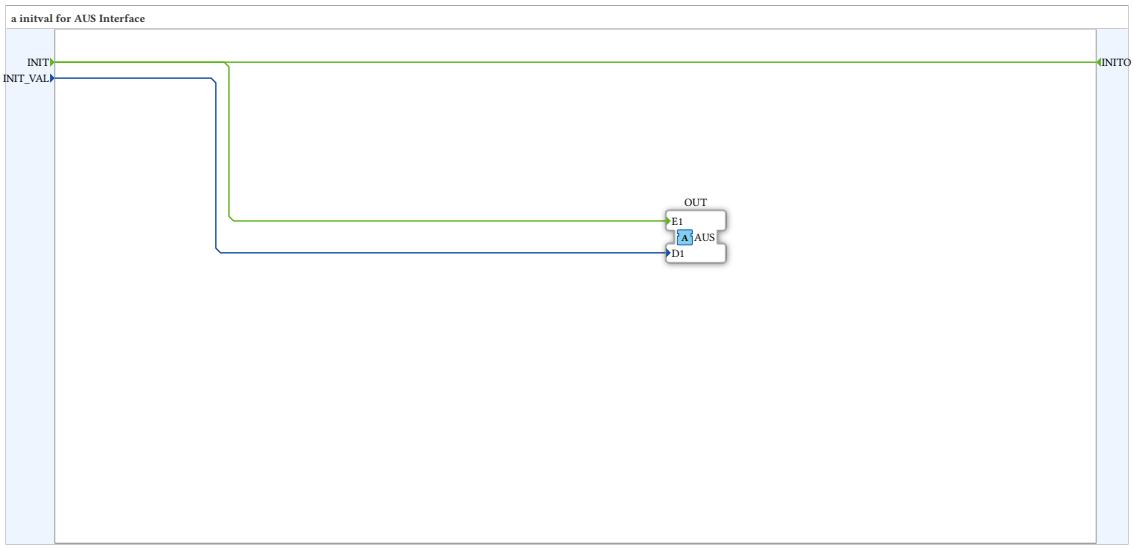
🛠️ Zugehörige Übungen

- Funk_IX_TO_DataPanel_MI_QX
- Funk_IXA_TO_DataPanel_MI_QXA
- IX_TO_Aux_QX
- IXA_TO_Aux_QXA
- IX_TO_logiBUS_LED_strip_QX
- SIN_TO_PCAN_Callback_Byte

####### Initval

initval_AUS





• – ○ • – ○ • – ○ •

Einleitung

`initval_AUS` ist der konfigurierbare Initialisierungsbaustein für den AUS-Adapter (Datentyp `USINT`, 8-Bit-Ganzzahl (vorzeichenlos)): Er nimmt bei der Initialisierung einen Startwert als Dateneingang entgegen und legt diesen auf einen `AUS`-Adapter-Plug. Damit ersetzt er die früher verbreitete Praxis, einen konstanten Vergleichswert über einen gemischten Adapter/Elementar-Baustein einzuspeisen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, liefert `INIT_VAL`. Löst die Ausgabe von `INIT_VAL` über den Plug `OUT` aus.

Ereignis-Ausgänge

- **INITO**: Bestätigt den Abschluss der Initialisierung.

Daten-Eingänge

- **INIT_VAL** (USINT): Der Startwert, der beim Eintreffen von `INIT` auf den Adapter `OUT` geschrieben wird.

Daten-Ausgänge

- Keine direkten Datenausgänge; der Wert wird ausschließlich über den Adapter `OUT` bereitgestellt.

Adapter

- **OUT** (Plug, Typ `AUS`): gibt bei `INIT` den über `INIT_VAL` übergebenen Wert über sein `E1`/`D1`-Adapterereignis aus.

Funktionsweise

Bei Eintreffen des Ereignisses `INIT` (mit `With Var="INIT_VAL"`) verknüpft `initval_AUS` intern zwei Datenpfade: `INIT_VAL` wird per Datenverbindung auf `OUT.D1` kopiert, und `INIT` wird per Ereignisverbindung an `OUT.E1` weitergereicht – der `OUT`-Plug feuert damit sein Adapterereignis mit dem gerade übergebenen

Wert. Anschließend wird `INIT` zusätzlich an `INITO` weitergereicht, wodurch der Aufrufer den Abschluss der Initialisierung bestätigt bekommt.

Technische Besonderheiten

- Der ausgegebene Wert ist nicht im Baustein selbst parametriert, sondern kommt bei jedem `INIT`-Aufruf frisch über `INIT_VAL` – der Baustein kann also auch außerhalb einer reinen Initialisierungsphase zum dynamischen Setzen eines `AUS`-Werts verwendet werden.
- Reine Verdrahtungslogik ohne eigene ECC/Zustandsmaschine (Composite-FB ohne innere Funktionsblöcke); die Datenverbindung `INIT_VAL → OUT.D1` muss vor der Ereignisverbindung ausgewertet sein, damit `OUT.E1` den aktuellen Wert trägt – dies wird durch die 61499-Ausführungsreihenfolge der `FBNetwork`-Verbindungen sichergestellt und ist nicht separat zu verdrahten.

Zustandsübersicht

Der Baustein ist zustandslos: Jedes `INIT`-Ereignis führt unmittelbar zur Übernahme von `INIT_VAL` auf `OUT` und zur Bestätigung über `INITO`.

Anwendungsszenarien

- Initialisierung von `AUS`-Adapterverbindungen mit einem projekt- oder parameterabhängigen Startwert (z. B. aus einer Konfigurationsvariable)
- Ersatz für die entfernten „gemischten“ Adapter↔Elementar-Vergleichsbausteine: Ein konstanter `USINT`-Wert wird über `initval_AUS` auf einen `AUS`-Adapter gelegt und kann dann mit dem reinen adapterbasierten `AUS`-Vergleichsbaustein weiterverwendet werden.
- Wiederverwendbare Bausteinvorlage für Testaufbauten, in denen unterschiedliche `AUS`-Startwerte durchprobiert werden sollen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit initval_AX, das denselben Verdrahtungsmechanismus für den Datentyp `BOOL` nutzt. Anders als `const_AX_TRUE`/`const_AX_FALSE` (die es nur für `AX` gibt) parametriert `initval_AUS` den Ausgabewert nicht fest im Baustein, sondern nimmt ihn über den Dateneingang `INIT_VAL` entgegen.

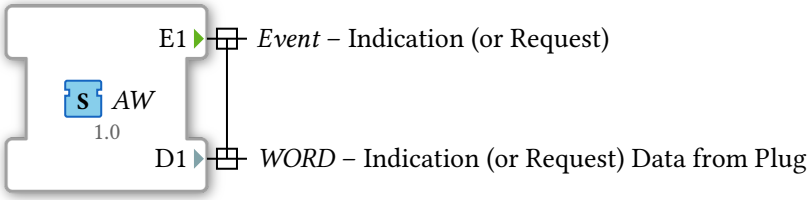
Fazit

`initval_AUS` liefert eine einfache, wiederverwendbare Initialisierungslogik für den `AUS`-Adapter und macht den ausgegebenen Wert über `INIT_VAL` frei konfigurierbar. Er eignet sich überall dort, wo ein `USINT`-Startwert nicht bereits zur Entwurfszeit feststeht oder als Ersatz für einen entfernten gemischten Vergleichsbaustein benötigt wird.

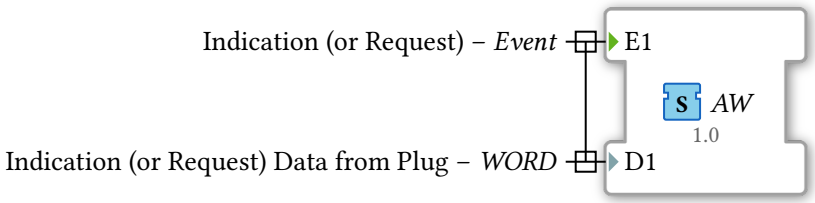
WORD

AW (WORD)

Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



🎧 Podcast

- From „Get Away with Electronics“ to Essential: How Agricultural Tech Earned Trust and Delivered Real Relief

Interface

Events

Name	Comment	With
E1	Indication (or Request)	D1

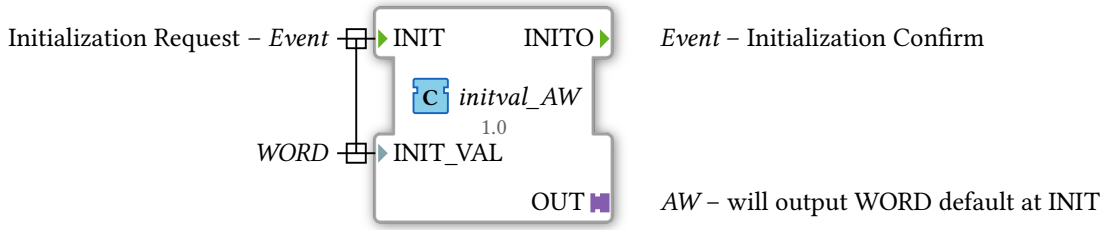
Data

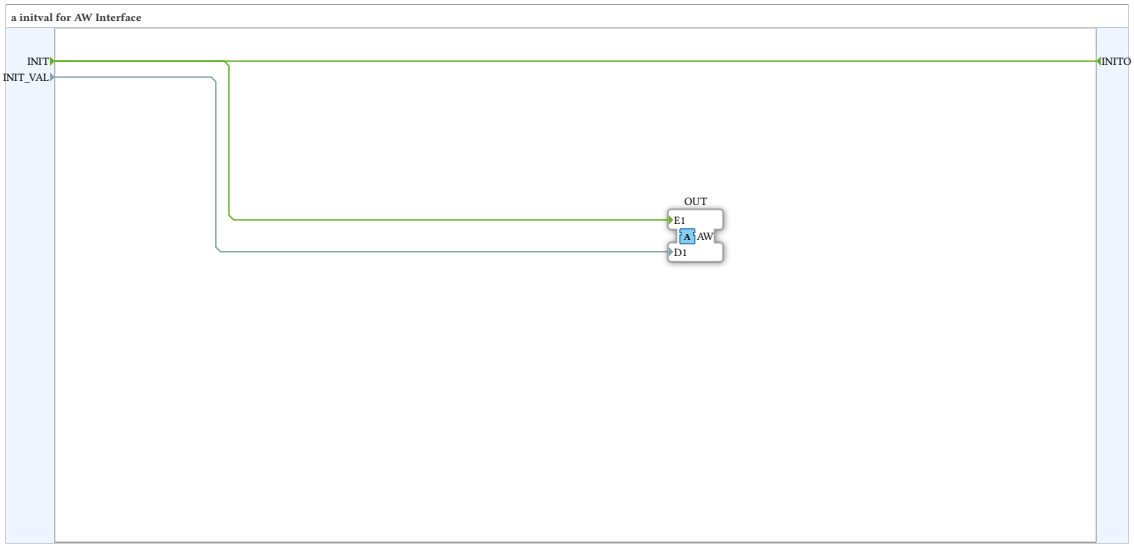
Name	Type	Comment
D1	WORD	Indication (or Request) Data from Plug

Unidirectional WORD Adapter

####### Initval

initval_AW





• – ◦ • – ◦ • – ◦ •

Einleitung

`initval_AW` ist der konfigurierbare Initialisierungsbaustein für den AW-Adapter (Datentyp `WORD`, 16-Bit-Bitmuster): Er nimmt bei der Initialisierung einen Startwert als Dateneingang entgegen und legt diesen auf einen `AW`-Adapter-Plug. Damit ersetzt er die früher verbreitete Praxis, einen konstanten Vergleichswert über einen gemischten Adapter/Elementar-Baustein einzuspeisen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, liefert `INIT_VAL`. Löst die Ausgabe von `INIT_VAL` über den Plug `OUT` aus.

Ereignis-Ausgänge

- **INITO**: Bestätigt den Abschluss der Initialisierung.

Daten-Eingänge

- **INIT_VAL** (WORD): Der Startwert, der beim Eintreffen von `INIT` auf den Adapter `OUT` geschrieben wird.

Daten-Ausgänge

- Keine direkten Datenausgänge; der Wert wird ausschließlich über den Adapter `OUT` bereitgestellt.

Adapter

- **OUT** (Plug, Typ `AW`): gibt bei `INIT` den über `INIT_VAL` übergebenen Wert über sein `E1/D1`-Adapterereignis aus.

Funktionsweise

Bei Eintreffen des Ereignisses `INIT` (mit `With Var="INIT_VAL"`) verknüpft `initval_AW` intern zwei Datenpfade: `INIT_VAL` wird per Datenverbindung auf `OUT.D1` kopiert, und `INIT` wird per Ereignisverbindung an `OUT.E1` weitergereicht – der `OUT`-Plug feuert damit sein Adapterereignis mit dem gerade übergebenen Wert. Anschließend wird `INIT` zusätzlich an `INITO` weitergereicht, wodurch der Aufrufer den Abschluss der Initialisierung bestätigt bekommt.

Technische Besonderheiten

- Der ausgegebene Wert ist nicht im Baustein selbst parametriert, sondern kommt bei jedem `INIT`-Aufruf frisch über `INIT_VAL` – der Baustein kann also auch außerhalb einer reinen Initialisierungsphase zum dynamischen Setzen eines `AW`-Werts verwendet werden.
- Reine Verdrahtungslogik ohne eigene ECC/Zustandsmaschine (Composite-FB ohne innere Funktionsblöcke); die Datenverbindung `INIT_VAL → OUT.D1` muss vor der Ereignisverbindung ausgewertet sein, damit `OUT.E1` den aktuellen Wert trägt – dies wird durch die 61499-Ausführungsreihenfolge der `FBNetwork`-Verbindungen sichergestellt und ist nicht separat zu verdrahten.

Zustandsübersicht

Der Baustein ist zustandslos: Jedes `INIT`-Ereignis führt unmittelbar zur Übernahme von `INIT_VAL` auf `OUT` und zur Bestätigung über `INITO`.

Anwendungsszenarien

- Initialisierung von `AW`-Adapterverbindungen mit einem projekt- oder parameterabhängigen Startwert (z. B. aus einer Konfigurationsvariable)
- Ersatz für die entfernten „gemischten“ Adapter↔Elementar-Vergleichsbausteine: Ein konstanter `WORD`-Wert wird über `initval_AW` auf einen `AW`-Adapter gelegt und kann dann mit dem reinen adapterbasierten `AW`-Vergleichsbaustein weiterverwendet werden.
- Wiederverwendbare Bausteinvorlage für Testaufbauten, in denen unterschiedliche `AW`-Startwerte durchprobiert werden sollen

⚖️ Vergleich mit ähnlichen Bausteinen

Vergleich mit initval_AX, das denselben Verdrahtungsmechanismus für den Datentyp `BOOL` nutzt. Anders als `const_AX_TRUE`/`const_AX_FALSE` (die es nur für `AX` gibt) parametriert `initval_AW` den Ausgabewert nicht fest im Baustein, sondern nimmt ihn über den Dateneingang `INIT_VAL` entgegen.

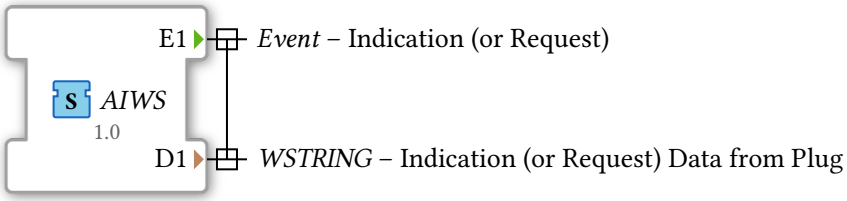
Fazit

`initval_AW` liefert eine einfache, wiederverwendbare Initialisierungslogik für den `AW`-Adapter und macht den ausgegebenen Wert über `INIT_VAL` frei konfigurierbar. Er eignet sich überall dort, wo ein `WORD`-Startwert nicht bereits zur Entwurfszeit feststeht oder als Ersatz für einen entfernten gemischten Vergleichsbaustein benötigt wird.

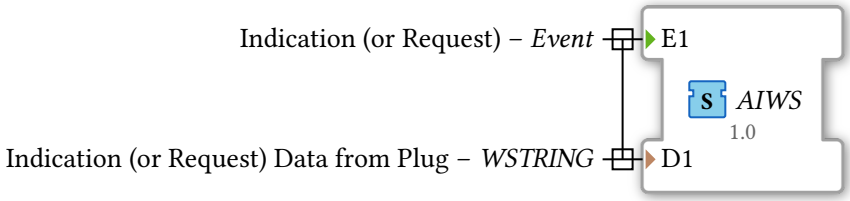
WSTRING

AIWS

Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Adapter **AIWS** stellt eine unidirektionale Schnittstelle für die Übertragung genau eines Ereignisses (Event) und eines zugehörigen WSTRING-Datums bereit. Er definiert die Kommunikationsrichtung von einer Quelle zu einem Ziel, wobei das Ereignis immer von dem Datum begleitet wird. Der Adapter dient als Grundbaustein für lose gekoppelte, ereignisgesteuerte Datenflüsse in IEC 61499-Anwendungen.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Name	Typ	Kommentar
E1	Event	Indication (or Request)

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Name	Typ	Kommentar
D1	WSTRING	Indication (or Request) Data from Plug

Adapter

Der Adapter selbst ist als Schnittstellendefinition zu verstehen. Er besitzt keine weiteren internen Adapter.

Funktionsweise

Der Adapter **AIWS** überträgt ein Ereignis **E1** zusammen mit einem WSTRING-Wert **D1** in einer festen Kopplung. Sobald die sendende Komponente das Ereignis auslöst, wird der aktuelle Wert von **D1** an die empfangende Seite übergeben. Die Richtung ist strikt von der Plug-Seite zur Socket-Seite (oder umgekehrt, je nach Verwendungszweck) – es findet keine Rückmeldung oder Bestätigung statt. Der Adapter definiert somit einen einfachen, unidirektionalen Datenkanal.

Technische Besonderheiten

- **Unidirektionalität**: Es existiert nur eine Ereignis-Ausgangsseite, keine Eingänge. Dies erzwingt eine klare Flussrichtung.
- **Kompaktheit**: Genau ein Event mit einem einfachen Datentyp (WSTRING) minimiert die Schnittstellenkomplexität.
- **Keine Fehlerbehandlung**: Der Adapter bietet keine Mechanismen für Timeouts, Wiederholungen oder Fehlererkennung. Die Datenintegrität muss von den angeschlossenen Funktionsblöcken sichergestellt werden.
- **Standardkonform**: Erfüllt die IEC 61499-1 Spezifikation für Adaptertypen.

Zustandsübersicht

Der Adapter selbst besitzt keinen internen Zustand. Er dient ausschließlich als Schnittstellendefinition und realisiert keine eigene Ablaufsteuerung. Die Zustandslogik liegt bei den verbundenen Funktionsblöcken.

Anwendungsszenarien

- **Einbindung von Sensordaten**: Übermittlung eines Ereignisses mit einem Messwert (z. B. Statusmeldung oder Identifikation) als WSTRING an eine übergeordnete Steuerung.
- **Textbasierte Befehle**: Senden von Kommandos (z. B. „START“, „STOP“) als Ereignis mit Parameter.
- **Logging und Diagnose**: Ausgabe von beschreibenden Ereignissen mit Klartextinformationen in eine Protokollkomponente.

Vergleich mit ähnlichen Bausteinen

Baustein	Ereignisse	Daten	Richtung	Besonderheit
AIWS	1 Output	1 WSTRING	Unidirektional	Einfachster Adapter für WSTRING
AI (hypothetisch)	1 Output	1 ANY (generisch)	Unidirektional	Flexibler Datentyp, aber aufwendiger
BIWS (bidirektional)	2 Events (in/out)	2 WSTRING in beide Richtungen	Bidirektional	Erfordert Handshake

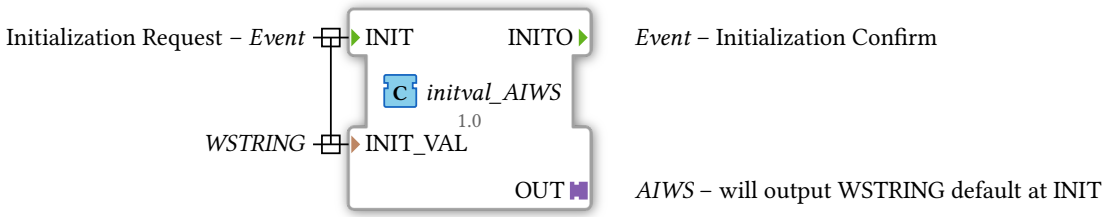
Der **AIWS** sticht durch seine minimale Schnittstelle hervor und eignet sich besonders für einfache, klar definierte Punkt-zu-Punkt-Verbindungen, bei denen kein Rückkanal benötigt wird.

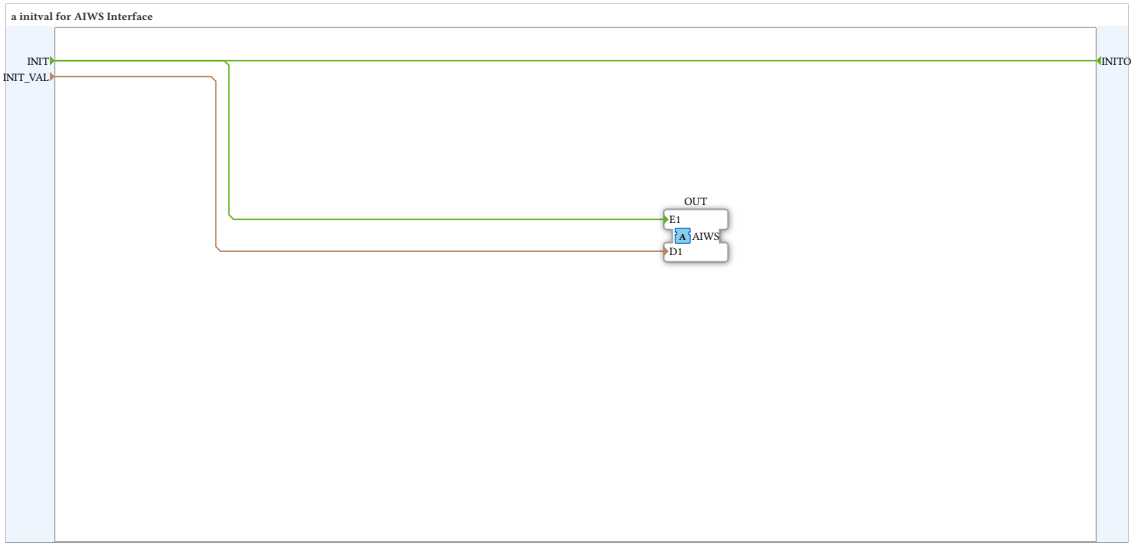
Fazit

Der Adapter **AIWS** ist ein grundlegender und effizienter Baustein für die unidirektionale Übertragung eines Ereignisses mit einem WSTRING-Datum. Durch seine Einfachheit reduziert er die Kopplung zwischen Komponenten und erleichtert die modulare Gestaltung von Automatisierungslösungen nach IEC 61499. Er eignet sich ideal für Aufgaben, die eine schnelle und unkomplizierte ereignisgesteuerte Kommunikation erfordern.

####### Initval

initval_AIWS





• – ◦ • – ◦ • – ◦ •

Einleitung

`initval_AIWS` ist der konfigurierbare Initialisierungsbaustein für den AIWS-Adapter (Datentyp `WSTRING`, Wide-Zeichenkette (Unicode)): Er nimmt bei der Initialisierung einen Startwert als Dateneingang entgegen und legt diesen auf einen `AIWS`-Adapter-Plug. Damit ersetzt er die früher verbreitete Praxis, einen konstanten Vergleichswert über einen gemischten Adapter/Elementar-Baustein einzuspeisen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, liefert `INIT_VAL`. Löst die Ausgabe von `INIT_VAL` über den Plug `OUT` aus.

Ereignis-Ausgänge

- **INITO**: Bestätigt den Abschluss der Initialisierung.

Daten-Eingänge

- **INIT_VAL** (WSTRING): Der Startwert, der beim Eintreffen von `INIT` auf den Adapter `OUT` geschrieben wird.

Daten-Ausgänge

- Keine direkten Datenausgänge; der Wert wird ausschließlich über den Adapter `OUT` bereitgestellt.

Adapter

- **OUT** (Plug, Typ `AIWS`): gibt bei `INIT` den über `INIT_VAL` übergebenen Wert über sein `E1`/`D1`-Adapterereignis aus.

Funktionsweise

Bei Eintreffen des Ereignisses `INIT` (mit `With Var="INIT_VAL"`) verknüpft `initval_AIWS` intern zwei Datenpfade: `INIT_VAL` wird per Datenverbindung auf `OUT.D1` kopiert, und `INIT` wird per Ereignisverbindung an `OUT.E1` weitergereicht – der `OUT`-Plug feuert damit sein Adapterereignis mit dem gerade übergebenen Wert. Anschließend wird `INIT` zusätzlich an `INITO` weitergereicht, wodurch der Aufrufer den Abschluss der Initialisierung bestätigt bekommt.

Technische Besonderheiten

- Der ausgegebene Wert ist nicht im Baustein selbst parametriert, sondern kommt bei jedem `INIT`-Aufruf frisch über `INIT_VAL` – der Baustein kann also auch außerhalb einer reinen Initialisierungsphase zum dynamischen Setzen eines `AIWS`-Werts verwendet werden.
- Reine Verdrahtungslogik ohne eigene ECC/Zustandsmaschine (Composite-FB ohne innere Funktionsblöcke); die Datenverbindung `INIT_VAL → OUT.D1` muss vor der Ereignisverbindung ausgewertet

sein, damit `OUT.E1` den aktuellen Wert trägt – dies wird durch die 61499-Ausführungsreihenfolge der `FBNetwork`-Verbindungen sichergestellt und ist nicht separat zu verdrahten.

Zustandsübersicht

Der Baustein ist zustandslos: Jedes `INIT`-Ereignis führt unmittelbar zur Übernahme von `INIT_VAL` auf `OUT` und zur Bestätigung über `INITO`.

Anwendungsszenarien

- Initialisierung von `AIWS`-Adapterverbindungen mit einem projekt- oder parameterabhängigen Startwert (z. B. aus einer Konfigurationsvariable)
- Ersatz für die entfernten „gemischten“ Adapter↔Elementar-Vergleichsbausteine: Ein konstanter `WSTRING`-Wert wird über `initval_AIWS` auf einen `AIWS`-Adapter gelegt und kann dann mit dem reinen adapterbasierten `AIWS`-Vergleichsbaustein weiterverwendet werden.
- Wiederverwendbare Bausteinvorlage für Testaufbauten, in denen unterschiedliche `AIWS`-Startwerte durchprobiert werden sollen

⚖️ Vergleich mit ähnlichen Bausteinen

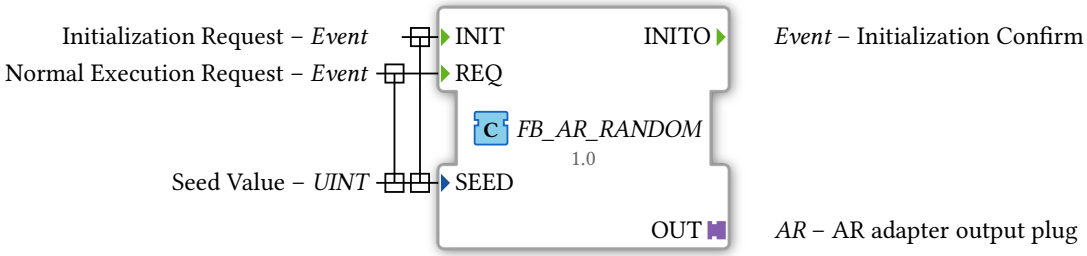
Vergleich mit initval_AX, das denselben Verdrahtungsmechanismus für den Datentyp `BOOL` nutzt. Anders als `const_AX_TRUE`/`const_AX_FALSE` (die es nur für `AX` gibt) parametriert `initval_AIWS` den Ausgabewert nicht fest im Baustein, sondern nimmt ihn über den Dateneingang `INIT_VAL` entgegen.

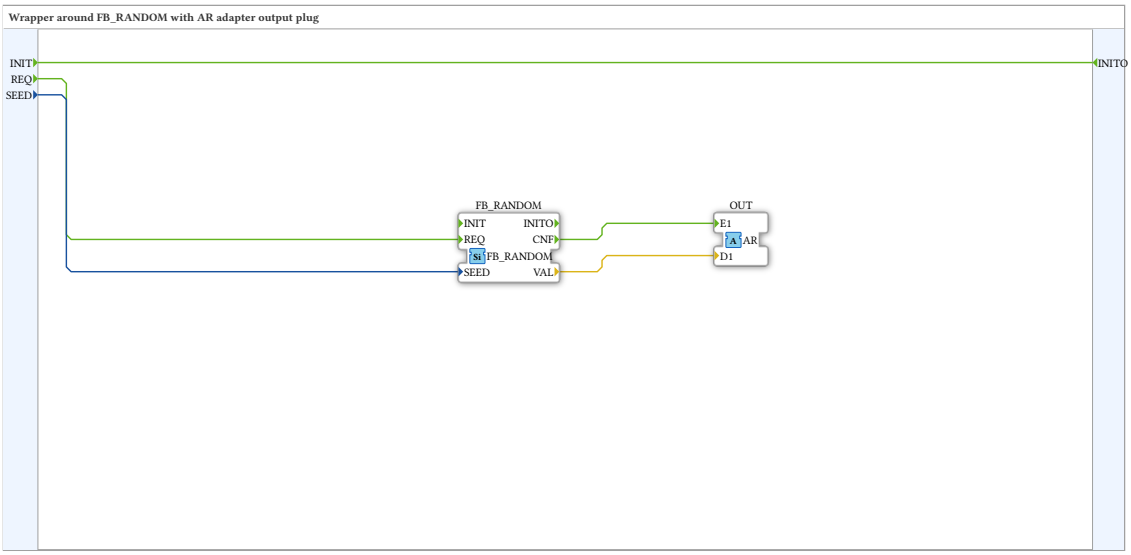
Fazit

`initval_AIWS` liefert eine einfache, wiederverwendbare Initialisierungslogik für den `AIWS`-Adapter und macht den ausgegebenen Wert über `INIT_VAL` frei konfigurierbar. Er eignet sich überall dort, wo ein `WSTRING`-Startwert nicht bereits zur Entwurfszeit feststeht oder als Ersatz für einen entfernten gemischten Vergleichsbaustein benötigt wird.

2.1.11.18 Utils

FB_AR_RANDOM





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **FB_AR_RANDOM** ist ein Wrapper um den Baustein `FB_RANDOM` aus der Bibliothek `eclipse4diac::utils`. Er erweitert dessen Funktionalität um eine unidirektionale AR-Adapter-Schnittstelle (`OUT`), sodass Zufallswerte direkt über einen Adapter an andere Applikationsteile weitergegeben werden können. Dadurch wird eine einfache Integration in adapterbasierte Kommunikationsstrukturen ermöglicht.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsanforderung – lädt den Start-Seed für den Zufallsgenerator.
- **REQ** (Event): Normaler Ausführungsauftrag – löst die Erzeugung eines neuen Zufallswerts aus.

Ereignis-Ausgänge

- **INITO** (EInit): Initialisierungsbestätigung – wird nach erfolgreicher Initialisierung gesendet.

Daten-Eingänge

- **SEED** (UINT, Initialwert `0`): Startwert für den Zufallsgenerator.

Daten-Ausgänge

- (Keine direkten Datenausgänge; der erzeugte Wert wird ausschließlich über den Adapter bereitgestellt.)

Adapter

- **OUT** (Typ `adapter::types::unidirectional::AR`): Ausgangs-Adapter, der das Ereignis `E1` und die Daten-Variable `D1` bereitstellt. Über diesen Adapter wird der erzeugte Zufallswert (als `D1`) und ein zugehöriges Ereignis (`E1`) an verbundene Bausteine übertragen.

Funktionsweise

Der Baustein enthält intern ein FBNetzwerk, in dem eine Instanz von `FB_RANDOM` eingebunden ist. Die Ereignis- und Daten-Verbindungen sind wie folgt realisiert:

- Das **INIT**-Ereignis wird direkt mit dem **INITO**-Ausgang verbunden – es erfolgt keine Weitergabe an den internen Zufallsgenerator. Somit dient INIT lediglich der Bestätigung, ohne weitere Aktionen auszulösen.
- Das **REQ**-Ereignis löst den internen `FB_RANDOM.REQ` aus.
- Das Ausgangsereignis `FB_RANDOM.CNF` (Bestätigung) wird über den Adapter als `OUT.E1` weitergegeben.
- Der **SEED**-Wert wird direkt an `FB_RANDOM.SEED` übergeben.
- Der erzeugte Zufallswert `FB_RANDOM.VAL` wird an die Adapter-Variable `OUT.D1` übertragen.

Damit erzeugt der Baustein bei jedem **REQ** einen neuen Zufallswert, der über den Adapter gesendet wird. Die Initialisierung hat keine Auswirkung auf den Zufallsgenerator, da sie nur intern mit `INITO` quittiert wird.

Technische Besonderheiten

- Der Baustein ist als reiner **Wrapper** implementiert und enthält keine eigene Logik – die eigentliche Zufallserzeugung übernimmt `FB_RANDOM`.
- Die Adapter-Schnittstelle ist **unidirektional** (`AR` = „Adapter Receive“ bzw. hier als Ausgang), d.h. sie unterstützt nur das Senden von Daten und Ereignissen.
- Der Baustein ist für die Verwendung in adapterbasierten Architekturen optimiert, wie sie typisch für IEC 61499-Applikationen sind.
- Es sind keine zusätzlichen Attribute oder Konfigurationsparameter außer dem Seed vorhanden.

Zustandsübersicht

Da der Baustein keinen eigenen Zustandsautomaten besitzt, beschränkt sich die Zustandslogik auf das ereignisgesteuerte Verhalten:

- Nach dem erstmaligen **INIT** wird `INITO` ausgegeben – der Baustein ist dann bereit für `REQ`.
- Bei jedem **REQ** wird ein Zufallswert erzeugt und über `OUT` gesendet. Es gibt keinen dauerhaften internen Zustand; der Seed wird nur beim Start gesetzt und kann über `SEED` beim Aufruf von `REQ` geändert werden.

Der interne `FB_RANDOM` kann je nach Implementierung einen Zustand (z.B. letzter Generatorwert) halten, aber nach außen hin ist der Baustein zustandslos.

Anwendungsszenarien

- **Simulationen**: Erzeugung von Zufallswerten in Simulationsmodellen, z.B. für stochastische Prozesse oder Testdaten.
- **Adapterbasierte Kommunikation**: Einbindung von Zufallswerten in komplexe Vernetzungen über Adapter, ohne zusätzliche Verdrahtung von Einzel-Datenpunkten.
- **Test- und Validierungsumgebungen**: Bereitstellung von Zufallseingaben für andere Bausteine über einen standardisierten Adapter-Kanal.

Vergleich mit ähnlichen Bausteinen

- **Direkter Einsatz von `FB_RANDOM`**: Dieser bietet die gleiche Zufallserzeugung, aber ohne Adapter-Anschluss. Der Wert muss über zusätzliche Datenverbindungen weitergegeben werden.
- **Alternative Zufallsgeneratoren**: Es gibt andere Bausteine mit unterschiedlichen Algorithmen oder Konfigurationsmöglichkeiten, aber `FB_AR_RANDOM` zeichnet sich durch seine direkte Adapter-Anbindung aus.
- **Bausteine mit mehreren Ausgängen**: Manche Zufallsgeneratoren liefern mehrere Werte oder sogar Verteilungen, während `FB_AR_RANDOM` einen einzelnen Wert pro Ereignis ausgibt.

Fazit

FB_AR_RANDOM ist ein einfacher, aber effektiver Wrapper, der die Zufallserzeugung von `FB_RANDOM` um eine unidirektionale AR-Adapter-Schnittstelle erweitert. Dadurch wird die Integration in adapterbasierte Applikationen erheblich vereinfacht. Der Baustein ist leichtgewichtig, benötigt nur einen Seed-Parameter und eignet sich hervorragend für Simulations- und Testumgebungen, in denen Zufallswerte über Adapter an andere Komponenten verteilt werden sollen.

2.1.12 iec61131-3-bool

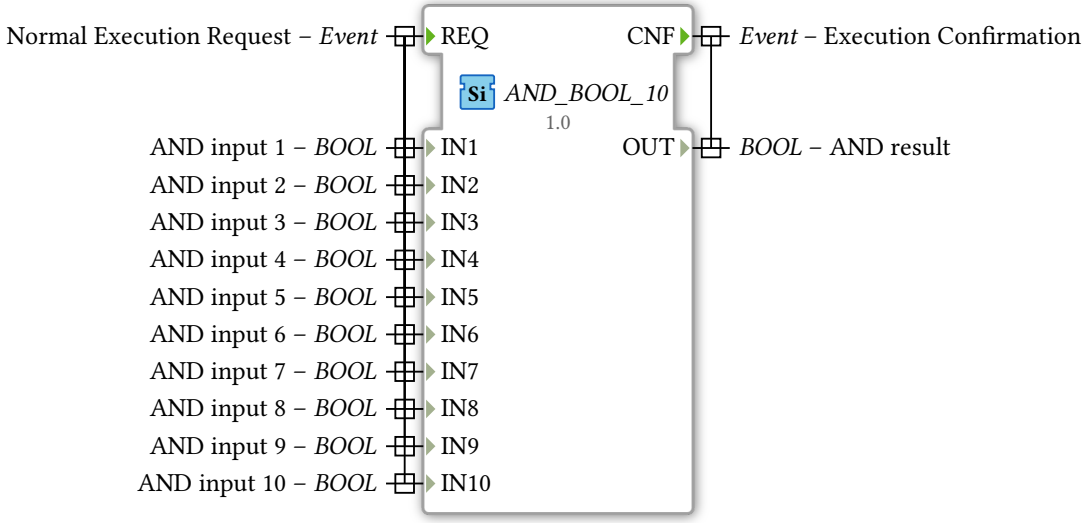
2.1.12.1 booleanOperators

AND_BOOL_10

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AND_BOOL_10` ist ein generischer Baustein zur Berechnung der logischen UND-Verknüpfung. Er führt eine bitweise UND-Operation über bis zu zehn separate BOOL-Eingänge durch. Der Baustein ist nach der Norm IEC 61131-3 klassifiziert und gehört zur Kategorie der Standard-Booleschen Funktionen. Er wird typischerweise in Steuerungslogik eingesetzt, um Bedingungen zu verknüpfen, bei denen alle Eingangssignale wahr (TRUE) sein müssen, um ein Ausgangssignal zu aktivieren.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Dieses Ereignis löst die Ausführung des Funktionsblocks aus. Bei seinem Eintreffen werden die Werte aller zehn Daten-Eingänge (`IN1` bis `IN10`) gelesen und die UND-Operation berechnet.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Dieses Ereignis signalisiert den Abschluss der Berechnung. Es wird zusammen mit dem berechneten Ergebnis am Datenausgang `OUT` ausgegeben.

Daten-Eingänge

- **IN1 (AND input 1):** Boolescher Eingang 1.
- **IN2 (AND input 2):** Boolescher Eingang 2.
- **IN3 (AND input 3):** Boolescher Eingang 3.
- **IN4 (AND input 4):** Boolescher Eingang 4.
- **IN5 (AND input 5):** Boolescher Eingang 5.
- **IN6 (AND input 6):** Boolescher Eingang 6.
- **IN7 (AND input 7):** Boolescher Eingang 7.
- **IN8 (AND input 8):** Boolescher Eingang 8.
- **IN9 (AND input 9):** Boolescher Eingang 9.
- **IN10 (AND input 10):** Boolescher Eingang 10.

Daten-Ausgänge

- **OUT (AND result):** Das Ergebnis der UND-Verknüpfung aller zehn Eingänge. Der Ausgang ist nur TRUE, wenn **alle** Eingänge `IN1` bis `IN10` den Wert TRUE haben. In allen anderen Fällen ist der Ausgang FALSE.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Die Funktionsweise ist deterministisch und ereignisgesteuert. Bei jedem Eintreffen des Ereignisses `REQ` wird die folgende logische Operation ausgeführt: `OUT = IN1 AND IN2 AND IN3 AND IN4 AND IN5 AND IN6 AND IN7 AND IN8 AND IN9 AND IN10`

Anschließend wird das Ergebnis am Datenausgang `OUT` bereitgestellt und das Bestätigungsereignis `CNF` ausgelöst, um nachfolgende Blöcke über die Verfügbarkeit des neuen Ergebnisses zu informieren.

Technische Besonderheiten

- **Generischer Baustein:** Der Block ist als generischer Baustein implementiert (Attribut `GenericClassName = 'GEN_AND'`). Dies bedeutet, dass seine Funktionalität möglicherweise auf eine andere Anzahl von Eingängen parametrisiert werden kann, wobei diese spezielle Instanz fest auf zehn Eingänge ausgelegt ist.
- **Package:** Der Baustein ist im Package `iec61131::bitwiseOperators` enthalten.
- **Vollständige Auswertung:** Alle zehn Eingänge werden bei jeder Ausführung ausgewertet.

Zustandsübersicht

Der `AND_BOOL_10`-Block besitzt keinen internen Zustand (speicherlos). Sein Ausgang hängt ausschließlich von den aktuellen Werten der Eingänge zum Zeitpunkt der Auslösung durch `REQ` ab. Es gibt keine Verzögerungen, Filter oder Hysterese.

Anwendungsszenarien

- **Sicherheitsverknüpfungen:** In Maschinensicherheitskreisen, wo mehrere Sicherheitsschalter (Not-Aus, Lichtschranken, Türkontakte) alle geschlossen sein müssen, bevor ein gefährlicher Vorgang starten darf.
- **Mehrfachbedingungen:** In Prozesssteuerungen, um zu prüfen, ob mehrere Voraussetzungen (z.B. Füllstand, Temperatur, Druck) gleichzeitig innerhalb ihrer Sollbereiche liegen.
- **Freigabelogik:** Als Teil einer Freigabekette, bei der mehrere Stationen oder Bediener ihre Zustimmung (TRUE-Signal) geben müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`AND` (mit 2 Eingängen):** Der Standard-AND-Baustein nach IEC 61131-3 hat typischerweise nur zwei Eingänge. `AND_BOOL_10` erweitert diese Funktionalität auf zehn Eingänge ohne die Notwendigkeit, mehrere Zwei-Eingangs-AND-Blöcke zu verschalten. Siehe: AND_10
- **`OR_BOOL_10`:** Führt eine logische ODER-Verknüpfung durch. Das Ergebnis ist TRUE, wenn mindestens ein Eingang TRUE ist, während `AND_BOOL_10` verlangt, dass alle Eingänge TRUE sind.
- **`XOR` / `XNOR`:** Berechnen die exklusive ODER- bzw. die Äquivalenz-Verknüpfung, die sich auf die Parität der TRUE-Signale beziehen, nicht auf eine Alles-oder-Nichts-Bedingung wie das UND.
- **Generische Blöcke (`GEN_AND`):** Der `AND_BOOL_10` ist eine spezifische Instanz eines generischen UND-Blocks. In anderen Umgebungen könnte ein konfigurierbarer `GEN_AND`-Block verwendet werden, dem die gewünschte Anzahl an Eingängen parametrisiert zugewiesen wird.

Fazit

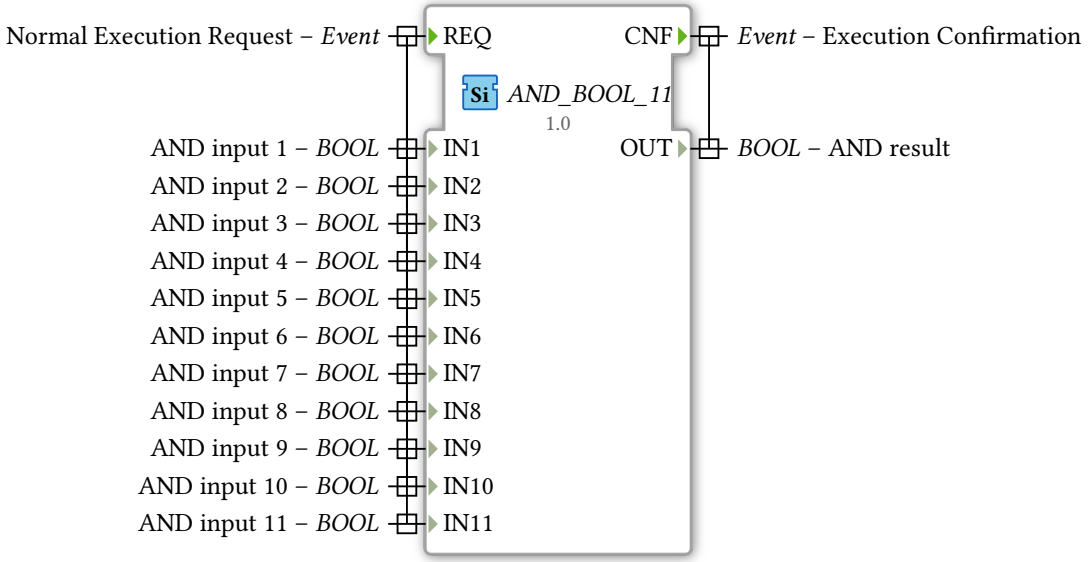
Der `AND_BOOL_10` ist ein spezialisierter, zuverlässiger und einfach zu verwendender Funktionsblock für Anwendungen, die eine logische UND-Verknüpfung über eine größere, feste Anzahl von Bedingungen erfordern. Seine ereignisgesteuerte, zustandslose Natur macht ihn vorhersehbar und einfach in bestehende Steuerungsabläufe zu integrieren. Für Szenarien mit einer variablen Anzahl an Bedingungen sollten flexible, parametrierbare Blöcke in Betracht gezogen werden.

AND_BOOL_11

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AND_BOOL_11` ist ein generischer Baustein zur Berechnung der logischen UND-Verknüpfung (Klassifikation `standard boolean function`, IEC 61131-3). Er verknüpft 11 separate BOOL-Eingänge (`IN1` bis `IN11`) und liefert das Ergebnis am Ausgang `OUT`. Er ergänzt die Standard-AND_BOOL-Familie (bisher bis `AND_BOOL_10`) um die Variante mit 11 Eingängen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Löst die Berechnung aus. Das Ereignis ist mit allen 11 Daten-Eingängen (`IN1` bis `IN11`) verknüpft.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Signalisiert den Abschluss der Berechnung. Das Ereignis ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN1 (AND input 1):** Boolescher Eingang 1.
- **IN2 (AND input 2):** Boolescher Eingang 2.
- **IN3 (AND input 3):** Boolescher Eingang 3.
- **IN4 (AND input 4):** Boolescher Eingang 4.
- **IN5 (AND input 5):** Boolescher Eingang 5.
- **IN6 (AND input 6):** Boolescher Eingang 6.
- **IN7 (AND input 7):** Boolescher Eingang 7.
- **IN8 (AND input 8):** Boolescher Eingang 8.
- **IN9 (AND input 9):** Boolescher Eingang 9.
- **IN10 (AND input 10):** Boolescher Eingang 10.
- **IN11 (AND input 11):** Boolescher Eingang 11.

Daten-Ausgänge

- **OUT (AND result):** Das Ergebnis der UND-Verknüpfung aller 11 Eingänge. Der Ausgang ist nur TRUE, wenn **alle** Eingänge `IN1` bis `IN11` den Wert TRUE haben, sonst FALSE.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei jedem Eintreffen des Ereignisses `REQ` werden alle 11 Eingänge gelesen und die Operation `OUT = IN1 AND IN2 AND ... AND IN11` berechnet. Anschließend wird das Ergebnis an `OUT` bereitgestellt und `CNF` ausgelöst.

Technische Besonderheiten

- **Generischer Baustein:** Der Typ trägt das Attribut `GenericClassName = 'GEN_AND_BOOL'`; diese Instanz ist fest auf 11 Eingänge ausgelegt.
- **Package:** `iec61131::booleanOperators`.
- **Zustandslos:** Der Baustein ist speicherlos: `OUT` hängt ausschließlich von den Werten der Eingänge zum Zeitpunkt von `REQ` ab.

Anwendungsszenarien

- **Freigabe- und Sicherheitsverknüpfungen:** Alle Bedingungen (z. B. Türkontakte, Not-Aus, Betriebsbereitschaft) müssen erfüllt sein, bevor eine Aktion starten darf.
- **Mehrfachbedingungen:** Prüfen, ob viele Voraussetzungen gleichzeitig erfüllt sind, ohne mehrere kleinere AND-Bausteine kaskadieren zu müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- `OR_BOOL_11`: Führt die ODER-Verknüpfung durch: Das Ergebnis ist TRUE, sobald mindestens ein Eingang TRUE ist, während `AND_BOOL_11` verlangt, dass alle Eingänge TRUE sind.
- `AND_BOOL_10`: Gleichartiger Baustein der Familie mit 10 Eingängen.

Fazit

`AND_BOOL_11` ist ein einfacher, ereignisgesteuerter Baustein für die UND-Verknüpfung von 11 booleschen Signalen mit fester Schnittstelle.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

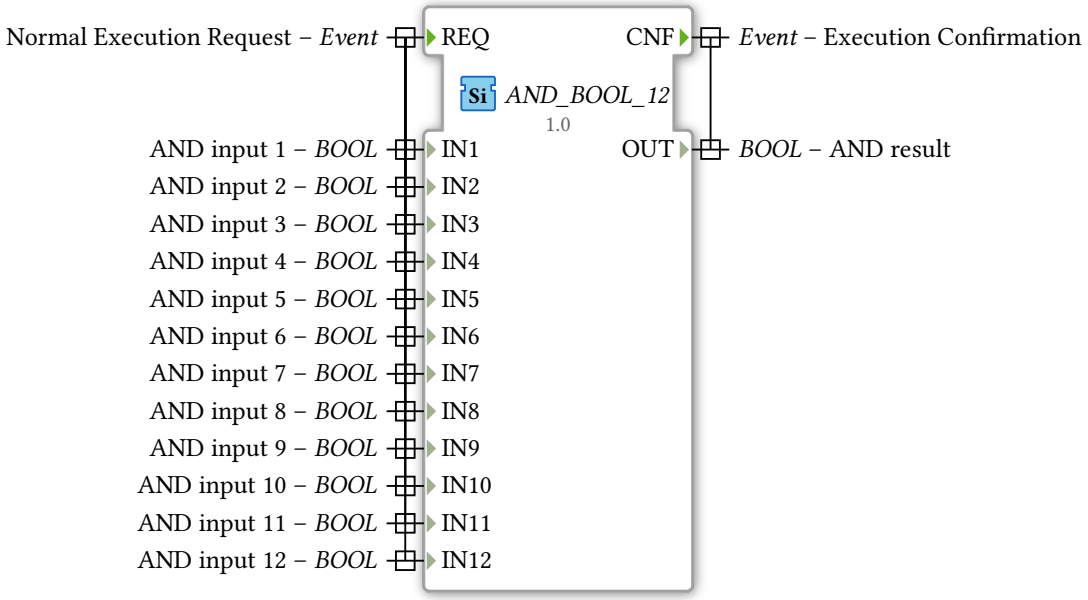
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AND_BOOL_12

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AND_BOOL_12` ist ein generischer Baustein zur Berechnung der logischen UND-Verknüpfung (Klassifikation `standard boolean function`, IEC 61131-3). Er verknüpft 12 separate BOOL-Eingänge (`IN1` bis `IN12`) und liefert das Ergebnis am Ausgang `OUT`. Er ergänzt die Standard-AND_BOOL-Familie (bisher bis `AND_BOOL_10`) um die Variante mit 12 Eingängen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Löst die Berechnung aus. Das Ereignis ist mit allen 12 Daten-Eingängen (`IN1` bis `IN12`) verknüpft.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Signalisiert den Abschluss der Berechnung. Das Ereignis ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN1 (AND input 1):** Boolescher Eingang 1.
- **IN2 (AND input 2):** Boolescher Eingang 2.
- **IN3 (AND input 3):** Boolescher Eingang 3.
- **IN4 (AND input 4):** Boolescher Eingang 4.
- **IN5 (AND input 5):** Boolescher Eingang 5.
- **IN6 (AND input 6):** Boolescher Eingang 6.
- **IN7 (AND input 7):** Boolescher Eingang 7.
- **IN8 (AND input 8):** Boolescher Eingang 8.
- **IN9 (AND input 9):** Boolescher Eingang 9.
- **IN10 (AND input 10):** Boolescher Eingang 10.
- **IN11 (AND input 11):** Boolescher Eingang 11.
- **IN12 (AND input 12):** Boolescher Eingang 12.

Daten-Ausgänge

- **OUT (AND result):** Das Ergebnis der UND-Verknüpfung aller 12 Eingänge. Der Ausgang ist nur TRUE, wenn **alle** Eingänge `IN1` bis `IN12` den Wert TRUE haben, sonst FALSE.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei jedem Eintreffen des Ereignisses `REQ` werden alle 12 Eingänge gelesen und die Operation `OUT = IN1 AND IN2 AND ... AND IN12` berechnet. Anschließend wird das Ergebnis an `OUT` bereitgestellt und `CNF` ausgelöst.

Technische Besonderheiten

- **Generischer Baustein:** Der Typ trägt das Attribut `GenericClassName = 'GEN_AND_BOOL'`; diese Instanz ist fest auf 12 Eingänge ausgelegt.
- **Package:** `iec61131::booleanOperators`.
- **Zustandslos:** Der Baustein ist speicherlos: `OUT` hängt ausschließlich von den Werten der Eingänge zum Zeitpunkt von `REQ` ab.

Anwendungsszenarien

- **Freigabe- und Sicherheitsverknüpfungen:** Alle Bedingungen (z. B. Türkontakte, Not-Aus, Betriebsbereitschaft) müssen erfüllt sein, bevor eine Aktion starten darf.
- **Mehrfachbedingungen:** Prüfen, ob viele Voraussetzungen gleichzeitig erfüllt sind, ohne mehrere kleinere AND-Bausteine kaskadieren zu müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- `OR_BOOL_12`: Führt die ODER-Verknüpfung durch: Das Ergebnis ist TRUE, sobald mindestens ein Eingang TRUE ist, während `AND_BOOL_12` verlangt, dass alle Eingänge TRUE sind.
- `AND_BOOL_10`: Gleichartiger Baustein der Familie mit 10 Eingängen.

Fazit

`AND_BOOL_12` ist ein einfacher, ereignisgesteuerter Baustein für die UND-Verknüpfung von 12 booleschen Signalen mit fester Schnittstelle.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

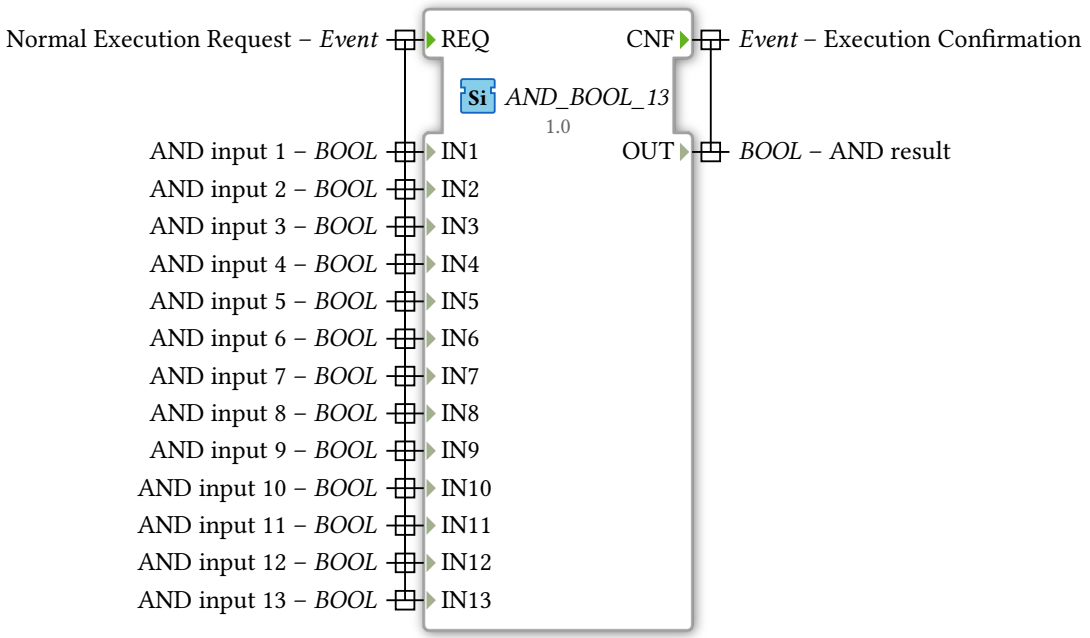
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AND_BOOL_13

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AND_BOOL_13` ist ein generischer Baustein zur Berechnung der logischen UND-Verknüpfung (Klassifikation `standard boolean function`, IEC 61131-3). Er verknüpft 13 separate BOOL-Eingänge (`IN1` bis `IN13`) und liefert das Ergebnis am Ausgang `OUT`. Er ergänzt die Standard-AND_BOOL-Familie (bisher bis `AND_BOOL_10`) um die Variante mit 13 Eingängen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Löst die Berechnung aus. Das Ereignis ist mit allen 13 Daten-Eingängen (`IN1` bis `IN13`) verknüpft.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Signalisiert den Abschluss der Berechnung. Das Ereignis ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN1 (AND input 1):** Boolescher Eingang 1.
- **IN2 (AND input 2):** Boolescher Eingang 2.
- **IN3 (AND input 3):** Boolescher Eingang 3.
- **IN4 (AND input 4):** Boolescher Eingang 4.
- **IN5 (AND input 5):** Boolescher Eingang 5.
- **IN6 (AND input 6):** Boolescher Eingang 6.
- **IN7 (AND input 7):** Boolescher Eingang 7.
- **IN8 (AND input 8):** Boolescher Eingang 8.
- **IN9 (AND input 9):** Boolescher Eingang 9.
- **IN10 (AND input 10):** Boolescher Eingang 10.
- **IN11 (AND input 11):** Boolescher Eingang 11.
- **IN12 (AND input 12):** Boolescher Eingang 12.
- **IN13 (AND input 13):** Boolescher Eingang 13.

Daten-Ausgänge

- **OUT (AND result):** Das Ergebnis der UND-Verknüpfung aller 13 Eingänge. Der Ausgang ist nur TRUE, wenn **alle** Eingänge `IN1` bis `IN13` den Wert TRUE haben, sonst FALSE.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei jedem Eintreffen des Ereignisses `REQ` werden alle 13 Eingänge gelesen und die Operation `OUT = IN1 AND IN2 AND ... AND IN13` berechnet. Anschließend wird das Ergebnis an `OUT` bereitgestellt und `CNF` ausgelöst.

Technische Besonderheiten

- **Generischer Baustein:** Der Typ trägt das Attribut `GenericClassName = 'GEN_AND_BOOL'`; diese Instanz ist fest auf 13 Eingänge ausgelegt.
- **Package:** `iec61131::booleanOperators`.
- **Zustandslos:** Der Baustein ist speicherlos: `OUT` hängt ausschließlich von den Werten der Eingänge zum Zeitpunkt von `REQ` ab.

Anwendungsszenarien

- **Freigabe- und Sicherheitsverknüpfungen:** Alle Bedingungen (z. B. Türkontakte, Not-Aus, Betriebsbereitschaft) müssen erfüllt sein, bevor eine Aktion starten darf.
- **Mehrfachbedingungen:** Prüfen, ob viele Voraussetzungen gleichzeitig erfüllt sind, ohne mehrere kleinere AND-Bausteine kaskadieren zu müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- `OR_BOOL_13`: Führt die ODER-Verknüpfung durch: Das Ergebnis ist TRUE, sobald mindestens ein Eingang TRUE ist, während `AND_BOOL_13` verlangt, dass alle Eingänge TRUE sind.
- `AND_BOOL_10`: Gleichartiger Baustein der Familie mit 10 Eingängen.

Fazit

`AND_BOOL_13` ist ein einfacher, ereignisgesteuerter Baustein für die UND-Verknüpfung von 13 booleschen Signalen mit fester Schnittstelle.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

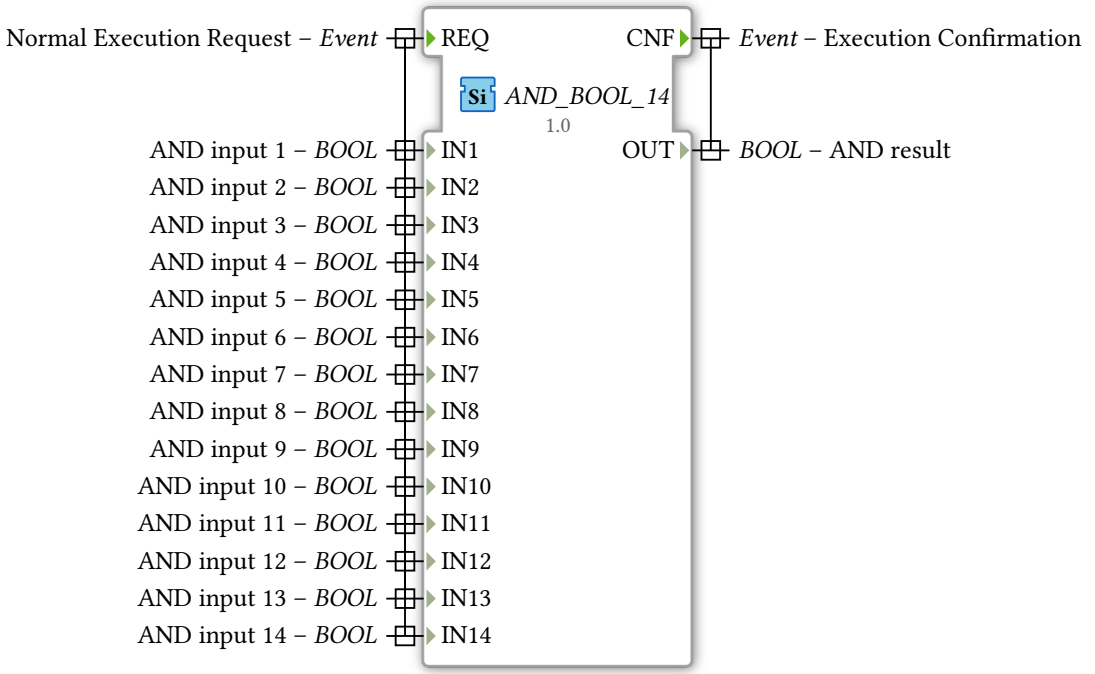
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AND_BOOL_14

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AND_BOOL_14` ist ein generischer Baustein zur Berechnung der logischen UND-Verknüpfung (Klassifikation `standard boolean function`, IEC 61131-3). Er verknüpft 14 separate BOOL-Eingänge (`IN1` bis `IN14`) und liefert das Ergebnis am Ausgang `OUT`. Er ergänzt die Standard-AND_BOOL-Familie (bisher bis `AND_BOOL_10`) um die Variante mit 14 Eingängen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Löst die Berechnung aus. Das Ereignis ist mit allen 14 Daten-Eingängen (`IN1` bis `IN14`) verknüpft.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Signalisiert den Abschluss der Berechnung. Das Ereignis ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN1 (AND input 1):** Boolescher Eingang 1.
- **IN2 (AND input 2):** Boolescher Eingang 2.
- **IN3 (AND input 3):** Boolescher Eingang 3.
- **IN4 (AND input 4):** Boolescher Eingang 4.
- **IN5 (AND input 5):** Boolescher Eingang 5.
- **IN6 (AND input 6):** Boolescher Eingang 6.
- **IN7 (AND input 7):** Boolescher Eingang 7.
- **IN8 (AND input 8):** Boolescher Eingang 8.
- **IN9 (AND input 9):** Boolescher Eingang 9.
- **IN10 (AND input 10):** Boolescher Eingang 10.
- **IN11 (AND input 11):** Boolescher Eingang 11.
- **IN12 (AND input 12):** Boolescher Eingang 12.
- **IN13 (AND input 13):** Boolescher Eingang 13.

- **IN14 (AND input 14):** Boolescher Eingang 14.

Daten-Ausgänge

- **OUT (AND result):** Das Ergebnis der UND-Verknüpfung aller 14 Eingänge. Der Ausgang ist nur TRUE, wenn **alle** Eingänge `IN1` bis `IN14` den Wert TRUE haben, sonst FALSE.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei jedem Eintreffen des Ereignisses `REQ` werden alle 14 Eingänge gelesen und die Operation `OUT = IN1 AND IN2 AND ... AND IN14` berechnet. Anschließend wird das Ergebnis an `OUT` bereitgestellt und `CNF` ausgelöst.

Technische Besonderheiten

- **Generischer Baustein:** Der Typ trägt das Attribut `GenericClassName = 'GEN_AND_BOOL'`; diese Instanz ist fest auf 14 Eingänge ausgelegt.
- **Package:** `iec61131::booleanOperators`.
- **Zustandslos:** Der Baustein ist speicherlos: `OUT` hängt ausschließlich von den Werten der Eingänge zum Zeitpunkt von `REQ` ab.

Anwendungsszenarien

- **Freigabe- und Sicherheitsverknüpfungen:** Alle Bedingungen (z. B. Türkontakte, Not-Aus, Betriebsbereitschaft) müssen erfüllt sein, bevor eine Aktion starten darf.
- **Mehrfachbedingungen:** Prüfen, ob viele Voraussetzungen gleichzeitig erfüllt sind, ohne mehrere kleinere AND-Bausteine kaskadieren zu müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- `OR_BOOL_14`: Führt die ODER-Verknüpfung durch: Das Ergebnis ist TRUE, sobald mindestens ein Eingang TRUE ist, während `AND_BOOL_14` verlangt, dass alle Eingänge TRUE sind.
- `AND_BOOL_10`: Gleichartiger Baustein der Familie mit 10 Eingängen.

Fazit

`AND_BOOL_14` ist ein einfacher, ereignisgesteuerter Baustein für die UND-Verknüpfung von 14 booleschen Signalen mit fester Schnittstelle.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

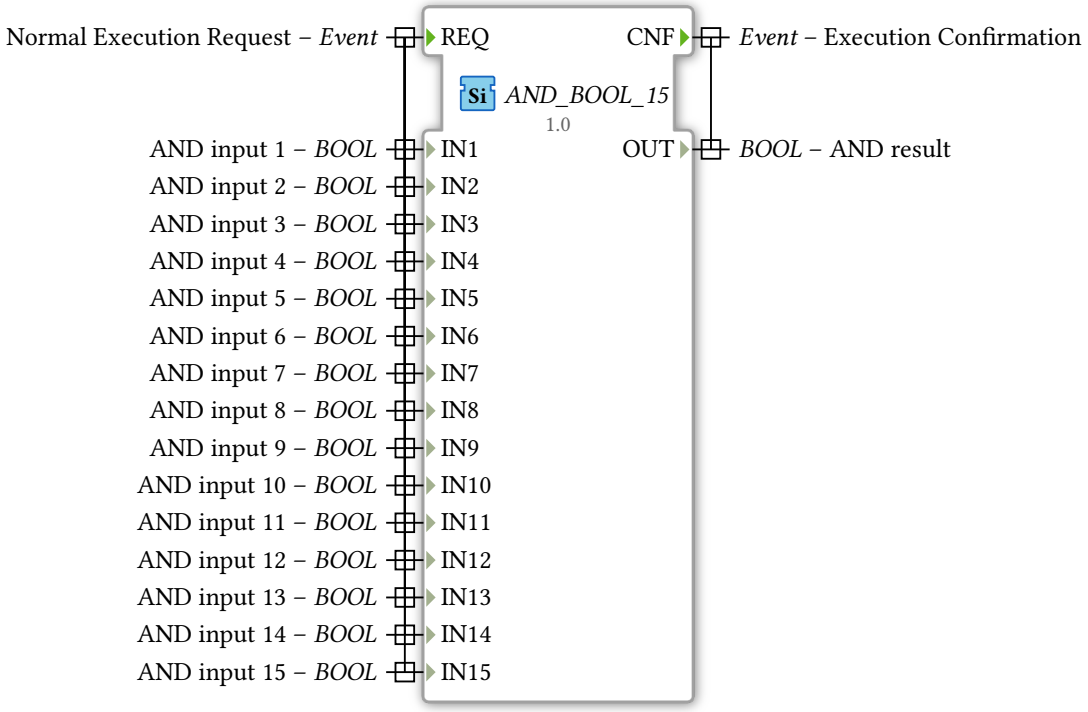
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AND_BOOL_15

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AND_BOOL_15` ist ein generischer Baustein zur Berechnung der logischen UND-Verknüpfung (Klassifikation `standard boolean function`, IEC 61131-3). Er verknüpft 15 separate BOOL-Eingänge (`IN1` bis `IN15`) und liefert das Ergebnis am Ausgang `OUT`. Er ergänzt die Standard-AND_BOOL-Familie (bisher bis `AND_BOOL_10`) um die Variante mit 15 Eingängen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Löst die Berechnung aus. Das Ereignis ist mit allen 15 Daten-Eingängen (`IN1` bis `IN15`) verknüpft.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Signalisiert den Abschluss der Berechnung. Das Ereignis ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN1 (AND input 1):** Boolescher Eingang 1.
- **IN2 (AND input 2):** Boolescher Eingang 2.
- **IN3 (AND input 3):** Boolescher Eingang 3.
- **IN4 (AND input 4):** Boolescher Eingang 4.
- **IN5 (AND input 5):** Boolescher Eingang 5.
- **IN6 (AND input 6):** Boolescher Eingang 6.
- **IN7 (AND input 7):** Boolescher Eingang 7.
- **IN8 (AND input 8):** Boolescher Eingang 8.
- **IN9 (AND input 9):** Boolescher Eingang 9.
- **IN10 (AND input 10):** Boolescher Eingang 10.
- **IN11 (AND input 11):** Boolescher Eingang 11.
- **IN12 (AND input 12):** Boolescher Eingang 12.
- **IN13 (AND input 13):** Boolescher Eingang 13.
- **IN14 (AND input 14):** Boolescher Eingang 14.
- **IN15 (AND input 15):** Boolescher Eingang 15.

Daten-Ausgänge

- **OUT (AND result):** Das Ergebnis der UND-Verknüpfung aller 15 Eingänge. Der Ausgang ist nur TRUE, wenn **alle** Eingänge `IN1` bis `IN15` den Wert TRUE haben, sonst FALSE.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei jedem Eintreffen des Ereignisses `REQ` werden alle 15 Eingänge gelesen und die Operation `OUT = IN1 AND IN2 AND ... AND IN15` berechnet. Anschließend wird das Ergebnis an `OUT` bereitgestellt und `CNF` ausgelöst.

Technische Besonderheiten

- **Generischer Baustein:** Der Typ trägt das Attribut `GenericClassName = 'GEN_AND_BOOL'`; diese Instanz ist fest auf 15 Eingänge ausgelegt.
- **Package:** `iec61131::booleanOperators`.
- **Zustandslos:** Der Baustein ist speicherlos: `OUT` hängt ausschließlich von den Werten der Eingänge zum Zeitpunkt von `REQ` ab.

Anwendungsszenarien

- **Freigabe- und Sicherheitsverknüpfungen:** Alle Bedingungen (z. B. Türkontakte, Not-Aus, Betriebsbereitschaft) müssen erfüllt sein, bevor eine Aktion starten darf.
- **Mehrfachbedingungen:** Prüfen, ob viele Voraussetzungen gleichzeitig erfüllt sind, ohne mehrere kleinere AND-Bausteine kaskadieren zu müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- `OR_BOOL_15`: Führt die ODER-Verknüpfung durch: Das Ergebnis ist TRUE, sobald mindestens ein Eingang TRUE ist, während `AND_BOOL_15` verlangt, dass alle Eingänge TRUE sind.
- `AND_BOOL_10`: Gleichartiger Baustein der Familie mit 10 Eingängen.

Fazit

`AND_BOOL_15` ist ein einfacher, ereignisgesteuerter Baustein für die UND-Verknüpfung von 15 booleschen Signalen mit fester Schnittstelle.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

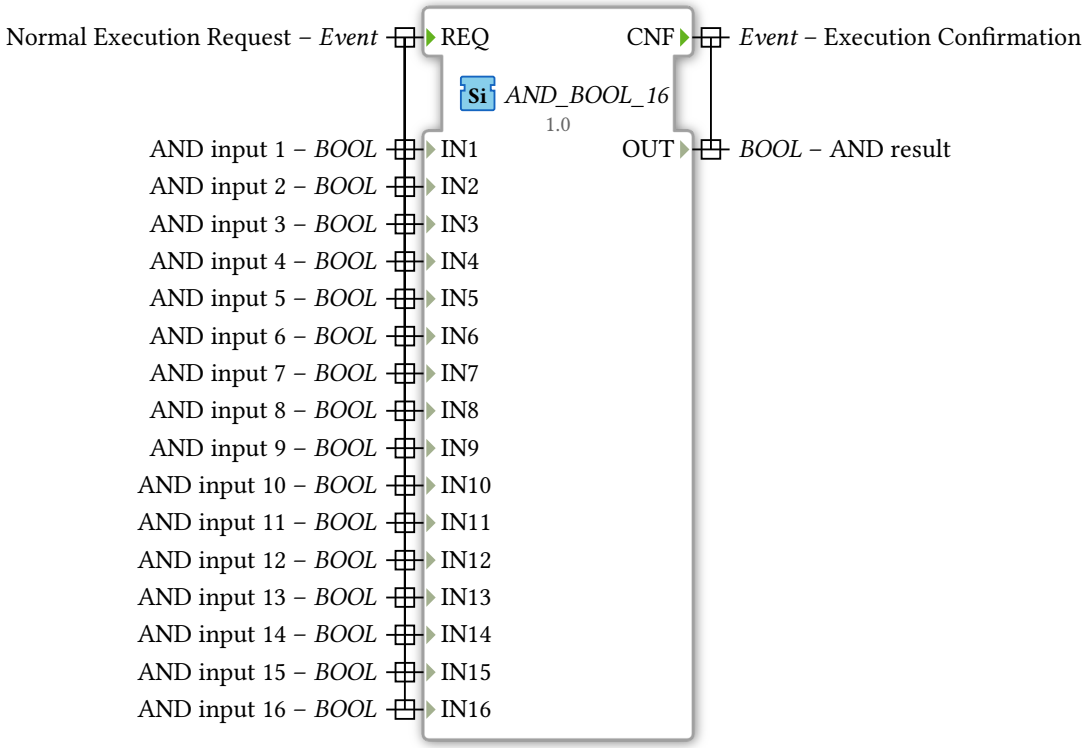
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AND_BOOL_16

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AND_BOOL_16` ist ein generischer Baustein zur Berechnung der logischen UND-Verknüpfung (Klassifikation `standard boolean function`, IEC 61131-3). Er verknüpft 16 separate BOOL-Eingänge (`IN1` bis `IN16`) und liefert das Ergebnis am Ausgang `OUT`. Er ergänzt die Standard-AND_BOOL-Familie (bisher bis `AND_BOOL_10`) um die Variante mit 16 Eingängen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Löst die Berechnung aus. Das Ereignis ist mit allen 16 Daten-Eingängen (`IN1` bis `IN16`) verknüpft.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Signalisiert den Abschluss der Berechnung. Das Ereignis ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN1 (AND input 1):** Boolescher Eingang 1.
- **IN2 (AND input 2):** Boolescher Eingang 2.
- **IN3 (AND input 3):** Boolescher Eingang 3.
- **IN4 (AND input 4):** Boolescher Eingang 4.
- **IN5 (AND input 5):** Boolescher Eingang 5.
- **IN6 (AND input 6):** Boolescher Eingang 6.
- **IN7 (AND input 7):** Boolescher Eingang 7.
- **IN8 (AND input 8):** Boolescher Eingang 8.
- **IN9 (AND input 9):** Boolescher Eingang 9.
- **IN10 (AND input 10):** Boolescher Eingang 10.
- **IN11 (AND input 11):** Boolescher Eingang 11.
- **IN12 (AND input 12):** Boolescher Eingang 12.
- **IN13 (AND input 13):** Boolescher Eingang 13.
- **IN14 (AND input 14):** Boolescher Eingang 14.
- **IN15 (AND input 15):** Boolescher Eingang 15.
- **IN16 (AND input 16):** Boolescher Eingang 16.

Daten-Ausgänge

- **OUT (AND result):** Das Ergebnis der UND-Verknüpfung aller 16 Eingänge. Der Ausgang ist nur TRUE, wenn **alle** Eingänge `IN1` bis `IN16` den Wert TRUE haben, sonst FALSE.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei jedem Eintreffen des Ereignisses `REQ` werden alle 16 Eingänge gelesen und die Operation `OUT = IN1 AND IN2 AND ... AND IN16` berechnet. Anschließend wird das Ergebnis an `OUT` bereitgestellt und `CNF` ausgelöst.

Technische Besonderheiten

- **Generischer Baustein:** Der Typ trägt das Attribut `GenericClassName = 'GEN_AND_BOOL'`; diese Instanz ist fest auf 16 Eingänge ausgelegt.
- **Package:** `iec61131::booleanOperators`.
- **Zustandslos:** Der Baustein ist speicherlos: `OUT` hängt ausschließlich von den Werten der Eingänge zum Zeitpunkt von `REQ` ab.

Anwendungsszenarien

- **Freigabe- und Sicherheitsverknüpfungen:** Alle Bedingungen (z. B. Türkontakte, Not-Aus, Betriebsbereitschaft) müssen erfüllt sein, bevor eine Aktion starten darf.
- **Mehrfachbedingungen:** Prüfen, ob viele Voraussetzungen gleichzeitig erfüllt sind, ohne mehrere kleinere AND-Bausteine kaskadieren zu müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- `OR_BOOL_16`: Führt die ODER-Verknüpfung durch: Das Ergebnis ist TRUE, sobald mindestens ein Eingang TRUE ist, während `AND_BOOL_16` verlangt, dass alle Eingänge TRUE sind.
- `AND_BOOL_10`: Gleichartiger Baustein der Familie mit 10 Eingängen.

Fazit

`AND_BOOL_16` ist ein einfacher, ereignisgesteuerter Baustein für die UND-Verknüpfung von 16 booleschen Signalen mit fester Schnittstelle.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

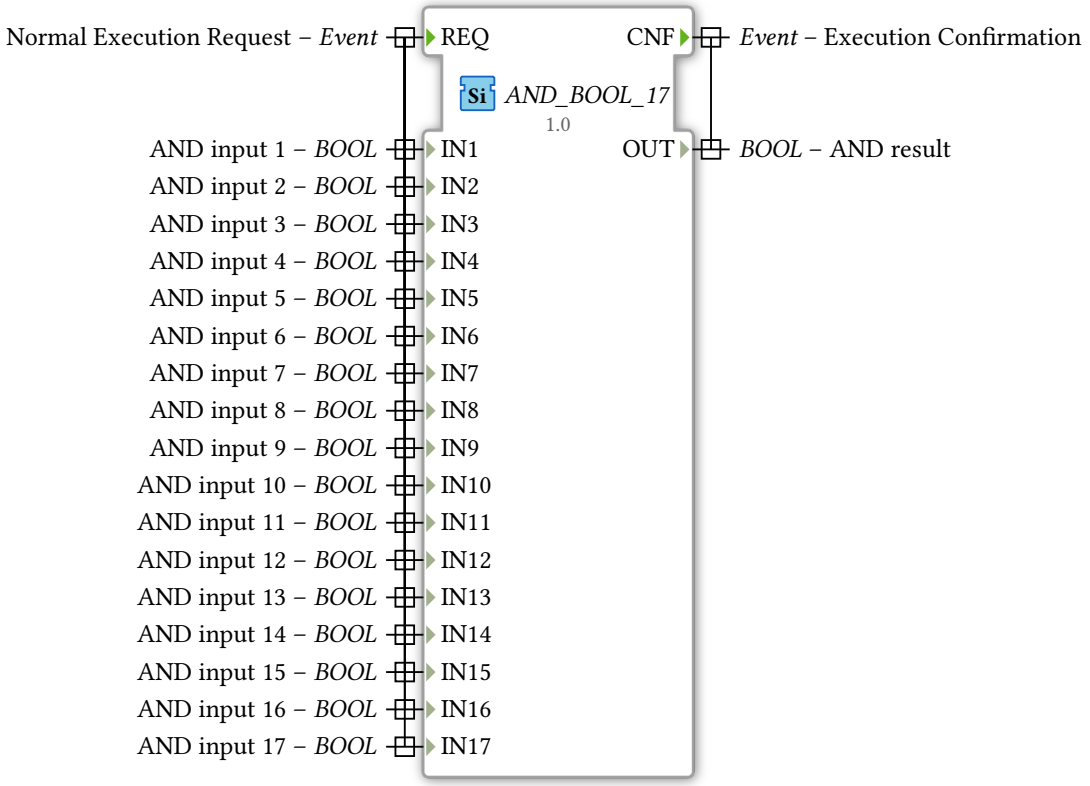
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AND_BOOL_17

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AND_BOOL_17` ist ein generischer Baustein zur Berechnung der logischen UND-Verknüpfung (Klassifikation `standard boolean function`, IEC 61131-3). Er verknüpft 17 separate BOOL-Eingänge (`IN1` bis `IN17`) und liefert das Ergebnis am Ausgang `OUT`. Er ergänzt die Standard-AND_BOOL-Familie (bisher bis `AND_BOOL_10`) um die Variante mit 17 Eingängen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Löst die Berechnung aus. Das Ereignis ist mit allen 17 Daten-Eingängen (`IN1` bis `IN17`) verknüpft.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Signalisiert den Abschluss der Berechnung. Das Ereignis ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN1 (AND input 1):** Boolescher Eingang 1.
- **IN2 (AND input 2):** Boolescher Eingang 2.
- **IN3 (AND input 3):** Boolescher Eingang 3.
- **IN4 (AND input 4):** Boolescher Eingang 4.
- **IN5 (AND input 5):** Boolescher Eingang 5.
- **IN6 (AND input 6):** Boolescher Eingang 6.
- **IN7 (AND input 7):** Boolescher Eingang 7.
- **IN8 (AND input 8):** Boolescher Eingang 8.
- **IN9 (AND input 9):** Boolescher Eingang 9.
- **IN10 (AND input 10):** Boolescher Eingang 10.
- **IN11 (AND input 11):** Boolescher Eingang 11.
- **IN12 (AND input 12):** Boolescher Eingang 12.
- **IN13 (AND input 13):** Boolescher Eingang 13.
- **IN14 (AND input 14):** Boolescher Eingang 14.
- **IN15 (AND input 15):** Boolescher Eingang 15.
- **IN16 (AND input 16):** Boolescher Eingang 16.
- **IN17 (AND input 17):** Boolescher Eingang 17.

Daten-Ausgänge

- **OUT (AND result):** Das Ergebnis der UND-Verknüpfung aller 17 Eingänge. Der Ausgang ist nur TRUE, wenn **alle** Eingänge `IN1` bis `IN17` den Wert TRUE haben, sonst FALSE.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei jedem Eintreffen des Ereignisses `REQ` werden alle 17 Eingänge gelesen und die Operation `OUT = IN1 AND IN2 AND ... AND IN17` berechnet. Anschließend wird das Ergebnis an `OUT` bereitgestellt und `CNF` ausgelöst.

Technische Besonderheiten

- **Generischer Baustein:** Der Typ trägt das Attribut `GenericClassName = 'GEN_AND_BOOL'`; diese Instanz ist fest auf 17 Eingänge ausgelegt.
- **Package:** `iec61131::booleanOperators`.
- **Zustandslos:** Der Baustein ist speicherlos: `OUT` hängt ausschließlich von den Werten der Eingänge zum Zeitpunkt von `REQ` ab.

Anwendungsszenarien

- **Freigabe- und Sicherheitsverknüpfungen:** Alle Bedingungen (z. B. Türkontakte, Not-Aus, Betriebsbereitschaft) müssen erfüllt sein, bevor eine Aktion starten darf.
- **Mehrfachbedingungen:** Prüfen, ob viele Voraussetzungen gleichzeitig erfüllt sind, ohne mehrere kleinere AND-Bausteine kaskadieren zu müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- `OR_BOOL_17`: Führt die ODER-Verknüpfung durch: Das Ergebnis ist TRUE, sobald mindestens ein Eingang TRUE ist, während `AND_BOOL_17` verlangt, dass alle Eingänge TRUE sind.
- `AND_BOOL_10`: Gleichartiger Baustein der Familie mit 10 Eingängen.

Fazit

`AND_BOOL_17` ist ein einfacher, ereignisgesteuerter Baustein für die UND-Verknüpfung von 17 booleschen Signalen mit fester Schnittstelle.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

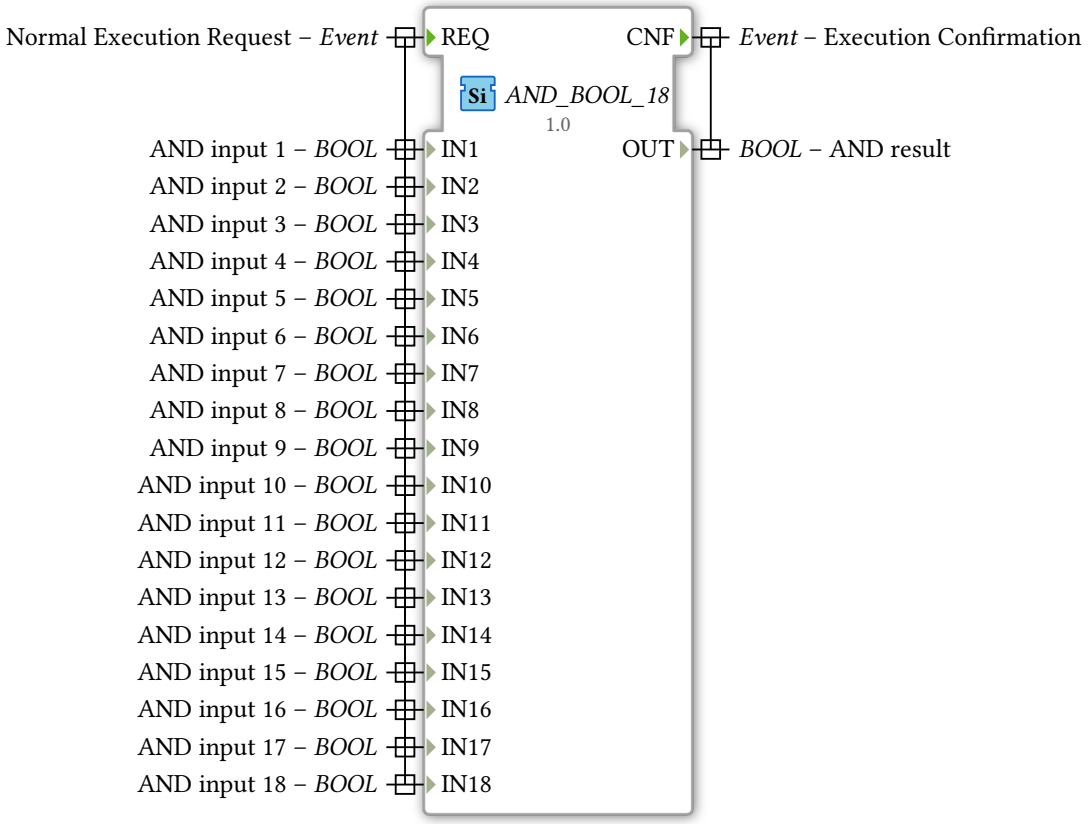
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AND_BOOL_18

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AND_BOOL_18` ist ein generischer Baustein zur Berechnung der logischen UND-Verknüpfung (Klassifikation `standard boolean function`, IEC 61131-3). Er verknüpft 18 separate BOOL-Eingänge (`IN1` bis `IN18`) und liefert das Ergebnis am Ausgang `OUT`. Er ergänzt die Standard-AND_BOOL-Familie (bisher bis `AND_BOOL_10`) um die Variante mit 18 Eingängen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Löst die Berechnung aus. Das Ereignis ist mit allen 18 Daten-Eingängen (`IN1` bis `IN18`) verknüpft.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Signalisiert den Abschluss der Berechnung. Das Ereignis ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN1 (AND input 1):** Boolescher Eingang 1.
- **IN2 (AND input 2):** Boolescher Eingang 2.
- **IN3 (AND input 3):** Boolescher Eingang 3.
- **IN4 (AND input 4):** Boolescher Eingang 4.
- **IN5 (AND input 5):** Boolescher Eingang 5.
- **IN6 (AND input 6):** Boolescher Eingang 6.
- **IN7 (AND input 7):** Boolescher Eingang 7.
- **IN8 (AND input 8):** Boolescher Eingang 8.
- **IN9 (AND input 9):** Boolescher Eingang 9.
- **IN10 (AND input 10):** Boolescher Eingang 10.
- **IN11 (AND input 11):** Boolescher Eingang 11.
- **IN12 (AND input 12):** Boolescher Eingang 12.
- **IN13 (AND input 13):** Boolescher Eingang 13.
- **IN14 (AND input 14):** Boolescher Eingang 14.
- **IN15 (AND input 15):** Boolescher Eingang 15.
- **IN16 (AND input 16):** Boolescher Eingang 16.
- **IN17 (AND input 17):** Boolescher Eingang 17.
- **IN18 (AND input 18):** Boolescher Eingang 18.

Daten-Ausgänge

- **OUT (AND result):** Das Ergebnis der UND-Verknüpfung aller 18 Eingänge. Der Ausgang ist nur TRUE, wenn **alle** Eingänge `IN1` bis `IN18` den Wert TRUE haben, sonst FALSE.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei jedem Eintreffen des Ereignisses `REQ` werden alle 18 Eingänge gelesen und die Operation `OUT = IN1 AND IN2 AND ... AND IN18` berechnet. Anschließend wird das Ergebnis an `OUT` bereitgestellt und `CNF` ausgelöst.

Technische Besonderheiten

- **Generischer Baustein:** Der Typ trägt das Attribut `GenericClassName = 'GEN_AND_BOOL'`; diese Instanz ist fest auf 18 Eingänge ausgelegt.
- **Package:** `iec61131::booleanOperators`.
- **Zustandslos:** Der Baustein ist speicherlos: `OUT` hängt ausschließlich von den Werten der Eingänge zum Zeitpunkt von `REQ` ab.

Anwendungsszenarien

- **Freigabe- und Sicherheitsverknüpfungen:** Alle Bedingungen (z. B. Türkontakte, Not-Aus, Betriebsbereitschaft) müssen erfüllt sein, bevor eine Aktion starten darf.
- **Mehrfachbedingungen:** Prüfen, ob viele Voraussetzungen gleichzeitig erfüllt sind, ohne mehrere kleinere AND-Bausteine kaskadieren zu müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- `OR_BOOL_18`: Führt die ODER-Verknüpfung durch: Das Ergebnis ist TRUE, sobald mindestens ein Eingang TRUE ist, während `AND_BOOL_18` verlangt, dass alle Eingänge TRUE sind.
- `AND_BOOL_10`: Gleichartiger Baustein der Familie mit 10 Eingängen.

Fazit

`AND_BOOL_18` ist ein einfacher, ereignisgesteuerter Baustein für die UND-Verknüpfung von 18 booleschen Signalen mit fester Schnittstelle.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

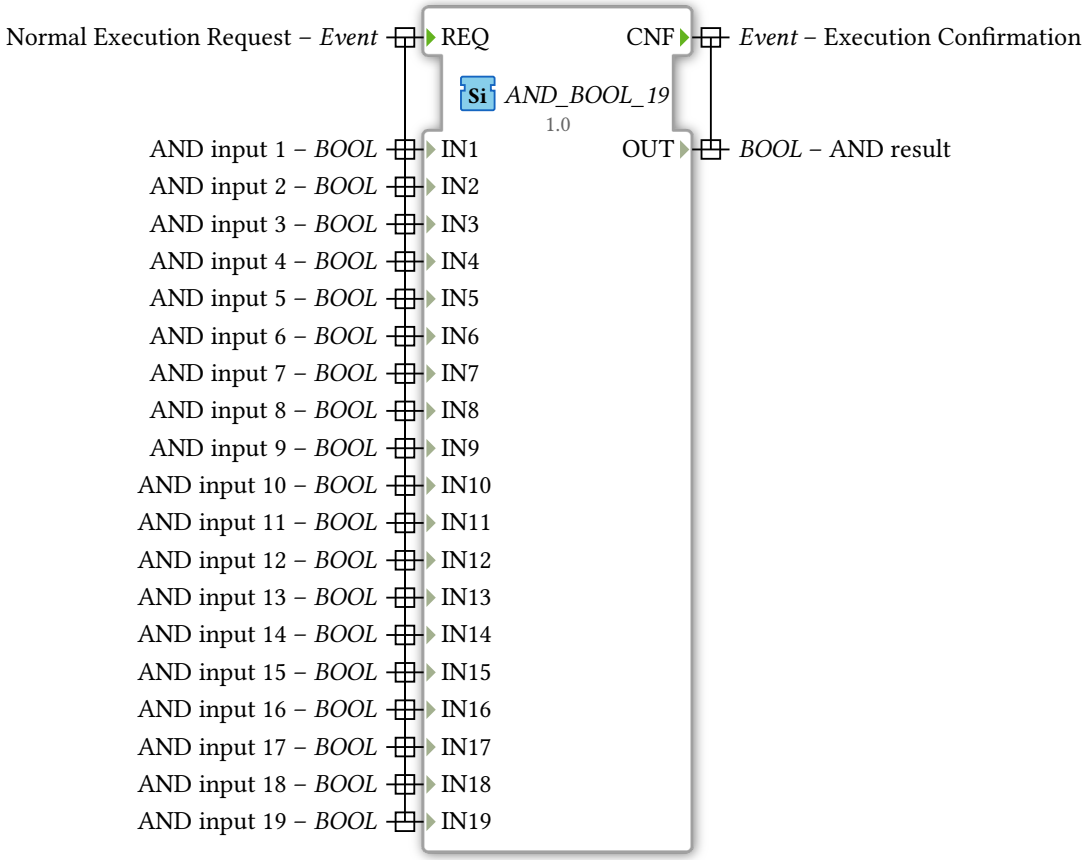
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AND_BOOL_19

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AND_BOOL_19` ist ein generischer Baustein zur Berechnung der logischen UND-Verknüpfung (Klassifikation `standard boolean function`, IEC 61131-3). Er verknüpft 19 separate BOOL-Eingänge (`IN1` bis `IN19`) und liefert das Ergebnis am Ausgang `OUT`. Er ergänzt die Standard-AND_BOOL-Familie (bisher bis `AND_BOOL_10`) um die Variante mit 19 Eingängen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Löst die Berechnung aus. Das Ereignis ist mit allen 19 Daten-Eingängen (`IN1` bis `IN19`) verknüpft.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Signalisiert den Abschluss der Berechnung. Das Ereignis ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN1 (AND input 1):** Boolescher Eingang 1.
- **IN2 (AND input 2):** Boolescher Eingang 2.
- **IN3 (AND input 3):** Boolescher Eingang 3.
- **IN4 (AND input 4):** Boolescher Eingang 4.
- **IN5 (AND input 5):** Boolescher Eingang 5.
- **IN6 (AND input 6):** Boolescher Eingang 6.
- **IN7 (AND input 7):** Boolescher Eingang 7.
- **IN8 (AND input 8):** Boolescher Eingang 8.
- **IN9 (AND input 9):** Boolescher Eingang 9.
- **IN10 (AND input 10):** Boolescher Eingang 10.
- **IN11 (AND input 11):** Boolescher Eingang 11.
- **IN12 (AND input 12):** Boolescher Eingang 12.
- **IN13 (AND input 13):** Boolescher Eingang 13.
- **IN14 (AND input 14):** Boolescher Eingang 14.
- **IN15 (AND input 15):** Boolescher Eingang 15.
- **IN16 (AND input 16):** Boolescher Eingang 16.
- **IN17 (AND input 17):** Boolescher Eingang 17.
- **IN18 (AND input 18):** Boolescher Eingang 18.

- **IN19 (AND input 19):** Boolescher Eingang 19.

Daten-Ausgänge

- **OUT (AND result):** Das Ergebnis der UND-Verknüpfung aller 19 Eingänge. Der Ausgang ist nur TRUE, wenn **alle** Eingänge `IN1` bis `IN19` den Wert TRUE haben, sonst FALSE.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei jedem Eintreffen des Ereignisses `REQ` werden alle 19 Eingänge gelesen und die Operation `OUT = IN1 AND IN2 AND ... AND IN19` berechnet. Anschließend wird das Ergebnis an `OUT` bereitgestellt und `CNF` ausgelöst.

Technische Besonderheiten

- **Generischer Baustein:** Der Typ trägt das Attribut `GenericClassName = 'GEN_AND_BOOL'`; diese Instanz ist fest auf 19 Eingänge ausgelegt.
- **Package:** `iec61131::booleanOperators`.
- **Zustandslos:** Der Baustein ist speicherlos: `OUT` hängt ausschließlich von den Werten der Eingänge zum Zeitpunkt von `REQ` ab.

Anwendungsszenarien

- **Freigabe- und Sicherheitsverknüpfungen:** Alle Bedingungen (z. B. Türkontakte, Not-Aus, Betriebsbereitschaft) müssen erfüllt sein, bevor eine Aktion starten darf.
- **Mehrfachbedingungen:** Prüfen, ob viele Voraussetzungen gleichzeitig erfüllt sind, ohne mehrere kleinere AND-Bausteine kaskadieren zu müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- `OR_BOOL_19`: Führt die ODER-Verknüpfung durch: Das Ergebnis ist TRUE, sobald mindestens ein Eingang TRUE ist, während `AND_BOOL_19` verlangt, dass alle Eingänge TRUE sind.
- `AND_BOOL_10`: Gleichartiger Baustein der Familie mit 10 Eingängen.

Fazit

`AND_BOOL_19` ist ein einfacher, ereignisgesteuerter Baustein für die UND-Verknüpfung von 19 booleschen Signalen mit fester Schnittstelle.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

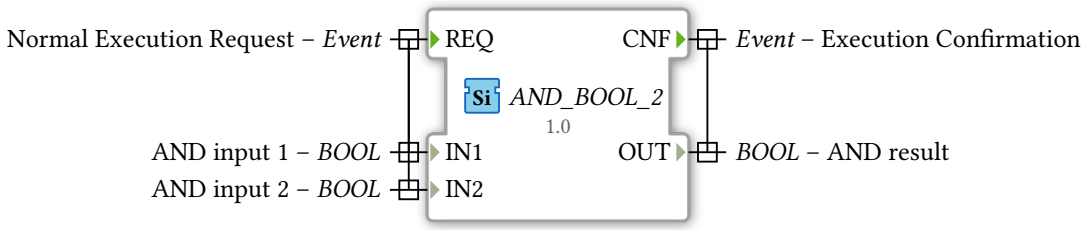
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AND_BOOL_2

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AND_BOOL_2` ist ein Standardbaustein zur Berechnung der logischen UND-Verknüpfung (Konjunktion) zweier boolescher Eingangswerte. Er folgt dem IEC 61131-3-Standard für grundlegende boolesche Funktionen und ist für den Einsatz in der 4diac-IDE konzipiert. Der Baustein wartet auf ein Auslöseereignis, führt die Operation aus und signalisiert die Beendigung der Berechnung mit einem Bestätigungsereignis.



Schnittstellenstruktur

Die Schnittstelle des `AND_BOOL_2`-Funktionsblocks ist einfach und übersichtlich aufgebaut und besteht aus einem Ereigniseingang, einem Ereignisausgang sowie den zugehörigen Datenverbindungen.

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Dieses Ereignis löst die Ausführung des Funktionsblocks aus. Bei seinem Eintreffen werden die aktuellen Werte an den Daten-Eingängen `IN1` und `IN2` gelesen und die UND-Operation berechnet.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Dieses Ereignis wird nach Abschluss der Berechnung erzeugt. Es signalisiert, dass das Ergebnis an den Daten-Ausgang `OUT` ausgegeben wurde und nun gültig ist.

Daten-Eingänge

- **IN1 (AND input 1):** Erster boolescher Operand (Typ: `BOOL`).
- **IN2 (AND input 2):** Zweiter boolescher Operand (Typ: `BOOL`).

Daten-Ausgänge

- **OUT (AND result):** Ergebnis der logischen UND-Verknüpfung von `IN1` und `IN2` (Typ: `BOOL`). Der Ausgangswert ist `TRUE` (1), wenn sowohl `IN1` als auch `IN2` den Wert `TRUE` haben. In allen anderen Fällen ist der Ausgang `FALSE` (0).

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Die Funktionsweise des `AND_BOOL_2`-Blocks ist deterministisch und folgt der klassischen booleschen Logik:

1. **Auslösung:** Das Eintreffen des Ereignisses `REQ` startet die Ausführung.
2. **Eingabe lesen:** Die aktuellen Werte der Daten-Eingänge `IN1` und `IN2` werden abgefragt.
3. **Berechnung:** Es wird die logische UND-Operation `OUT := IN1 AND IN2` durchgeführt.
4. **Ausgabe:** Das berechnete Ergebnis wird an den Daten-Ausgang `OUT` geschrieben.
5. **Bestätigung:** Unmittelbar nach der Ausgabe wird das Bestätigungsereignis `CNF` erzeugt, um nachgelagerte Blöcke über die Verfügbarkeit des neuen Ergebnisses zu informieren.

Die Operation kann durch folgende Wahrheitstabelle beschrieben werden:

IN1	IN2	OUT (IN1 AND IN2)
0	0	0
0	1	0
1	0	0
1	1	1

Technische Besonderheiten

- **Generischer Baustein:** Der Block ist als Instanz eines generischen UND-Bausteins (`GEN_AND`) implementiert, der für verschiedene Datentypen spezialisiert werden kann. In dieser konkreten Instanz ist er auf den Datentyp `BOOL` festgelegt.
- **Ereignisgesteuert:** Wie für 4diac-Funktionsblöcke typisch, ist die Ausführung strikt ereignisgesteuert. Eine Änderung der Eingangswerte allein löst keine Berechnung aus; diese erfolgt erst beim Eintreffen des `REQ`-Ereignisses.
- **Einfache Latenz:** Die Ausführungszeit ist vernachlässigbar, sodass das `CNF`-Ereignis praktisch unmittelbar auf das `REQ`-Ereignis folgt.

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand im Sinne eines Speichers für vorherige Werte. Er ist ein rein kombinatorischer Baustein, dessen Ausgang ausschließlich von den aktuellen Eingangswerten

bei Auslösung abhängt. Sein „Zustand“ ist daher immer „wartend auf REQ“ oder „in Ausführung“, wobei die Ausführungsphase sehr kurz ist.

Anwendungsszenarien

Der `AND_BOOL_2`-Block ist ein fundamentaler Baustein in der Steuerungs- und Automatisierungstechnik. Typische Anwendungen sind:

- **Sicherheitsbedingungen:** Verknüpfung mehrerer Sicherheitssignale (z.B. „Schutztür geschlossen“ UND „Not-Aus nicht gedrückt“), um eine Maschinenfreigabe zu erzeugen.
- **Zweihand-Bedienung:** Sicherstellung, dass zwei Taster gleichzeitig gedrückt werden müssen, um einen gefährlichen Vorgang zu starten.
- **Verkettung von Schaltbedingungen:** Kombination mehrerer Sensor- oder Schalterzustände, um ein Aktor-Signal zu steuern (z.B. „Material vorhanden“ UND „Position erreicht“ -> Greifer schließen).
- **Maskierung von Signalen:** Gezieltes Durchschalten oder Sperren eines Signals (`DATA`) durch ein Freigabesignal (`ENABLE`) mittels `OUT := DATA AND ENABLE`.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`AND_n` (generisch):** Der generische `GEN_AND`-Baustein kann für mehr als zwei Eingänge und andere Datentypen (z.B. `WORD`, `DWORD`) konfiguriert werden. `AND_BOOL_2` ist eine spezifische, auf zwei boolesche Eingänge festgelegte Instanz davon, die einfacher und direkter in booleschen Logikschaltungen eingesetzt werden kann. Siehe: AND_2
- **`OR_BOOL_2` / `XOR_BOOL_2`:** Diese Blöcke implementieren andere grundlegende boolesche Operationen (ODER bzw. exklusives ODER). Sie besitzen die identische ereignisgesteuerte Schnittstelle (`REQ`/`CNF`), führen aber eine andere logische Verknüpfung der Eingänge `IN1` und `IN2` durch.
- **Kombinatorische Logik in SFC/ST:** Die UND-Funktion könnte auch direkt in Structured Text (ST) mit dem `&`- oder `AND`-Operator codiert werden. Der Vorteil des `AND_BOOL_2`-FB liegt in der standardisierten, wiederverwendbaren und grafisch verknüpfbaren Form innerhalb des 4diac-Funktionsbaustein-Netzwerks (FBN).

🛠️ Zugehörige Übungen

- Uebung_002a4
- Uebung_002b3
- Uebung_006a3

Fazit

Der `AND_BOOL_2`-Funktionsblock ist ein essentieller, zuverlässiger und einfach zu verwendender Grundbaustein für die Implementierung boolescher Logik in 4diac-basierten Steuerungsanwendungen. Seine klare, ereignisgesteuerte Schnittstelle und die deterministische Funktionsweise machen ihn zum idealen Baustein für Sicherheitsverknüpfungen, Steuerungsbedingungen und allgemeine logische Operationen. Für reine boolesche Zweier-Verknüpfungen ist er die erste Wahl.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

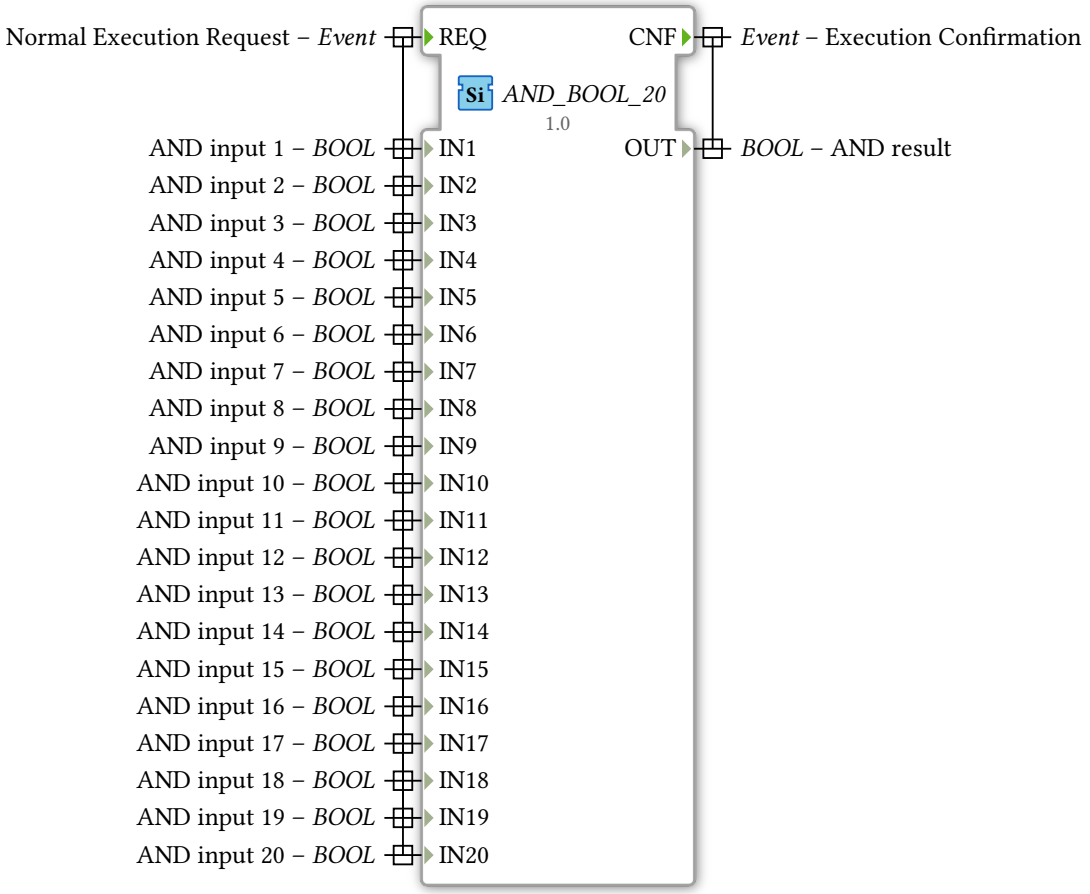
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AND_BOOL_20

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AND_BOOL_20` ist ein generischer Baustein zur Berechnung der logischen UND-Verknüpfung (Klassifikation `standard boolean function`, IEC 61131-3). Er verknüpft 20 separate BOOL-Eingänge (`IN1` bis `IN20`) und liefert das Ergebnis am Ausgang `OUT`. Er ergänzt die Standard-AND_BOOL-Familie (bisher bis `AND_BOOL_10`) um die Variante mit 20 Eingängen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Löst die Berechnung aus. Das Ereignis ist mit allen 20 Daten-Eingängen (`IN1` bis `IN20`) verknüpft.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Signalisiert den Abschluss der Berechnung. Das Ereignis ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN1 (AND input 1):** Boolescher Eingang 1.
- **IN2 (AND input 2):** Boolescher Eingang 2.
- **IN3 (AND input 3):** Boolescher Eingang 3.
- **IN4 (AND input 4):** Boolescher Eingang 4.
- **IN5 (AND input 5):** Boolescher Eingang 5.
- **IN6 (AND input 6):** Boolescher Eingang 6.
- **IN7 (AND input 7):** Boolescher Eingang 7.
- **IN8 (AND input 8):** Boolescher Eingang 8.
- **IN9 (AND input 9):** Boolescher Eingang 9.
- **IN10 (AND input 10):** Boolescher Eingang 10.
- **IN11 (AND input 11):** Boolescher Eingang 11.
- **IN12 (AND input 12):** Boolescher Eingang 12.
- **IN13 (AND input 13):** Boolescher Eingang 13.
- **IN14 (AND input 14):** Boolescher Eingang 14.
- **IN15 (AND input 15):** Boolescher Eingang 15.
- **IN16 (AND input 16):** Boolescher Eingang 16.
- **IN17 (AND input 17):** Boolescher Eingang 17.

- **IN18 (AND input 18):** Boolescher Eingang 18.
- **IN19 (AND input 19):** Boolescher Eingang 19.
- **IN20 (AND input 20):** Boolescher Eingang 20.

Daten-Ausgänge

- **OUT (AND result):** Das Ergebnis der UND-Verknüpfung aller 20 Eingänge. Der Ausgang ist nur TRUE, wenn **alle** Eingänge `IN1` bis `IN20` den Wert TRUE haben, sonst FALSE.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei jedem Eintreffen des Ereignisses `REQ` werden alle 20 Eingänge gelesen und die Operation `OUT = IN1 AND IN2 AND ... AND IN20` berechnet. Anschließend wird das Ergebnis an `OUT` bereitgestellt und `CNF` ausgelöst.

Technische Besonderheiten

- **Generischer Baustein:** Der Typ trägt das Attribut `GenericClassName = 'GEN_AND_BOOL'`; diese Instanz ist fest auf 20 Eingänge ausgelegt.
- **Package:** `iec61131::booleanOperators`.
- **Zustandslos:** Der Baustein ist speicherlos: `OUT` hängt ausschließlich von den Werten der Eingänge zum Zeitpunkt von `REQ` ab.

Anwendungsszenarien

- **Freigabe- und Sicherheitsverknüpfungen:** Alle Bedingungen (z. B. Türkontakte, Not-Aus, Betriebsbereitschaft) müssen erfüllt sein, bevor eine Aktion starten darf.
- **Mehrfachbedingungen:** Prüfen, ob viele Voraussetzungen gleichzeitig erfüllt sind, ohne mehrere kleinere AND-Bausteine kaskadieren zu müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- `OR_BOOL_20`: Führt die ODER-Verknüpfung durch: Das Ergebnis ist TRUE, sobald mindestens ein Eingang TRUE ist, während `AND_BOOL_20` verlangt, dass alle Eingänge TRUE sind.
- `AND_BOOL_10`: Gleichartiger Baustein der Familie mit 10 Eingängen.

Fazit

`AND_BOOL_20` ist ein einfacher, ereignisgesteuerter Baustein für die UND-Verknüpfung von 20 booleschen Signalen mit fester Schnittstelle.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

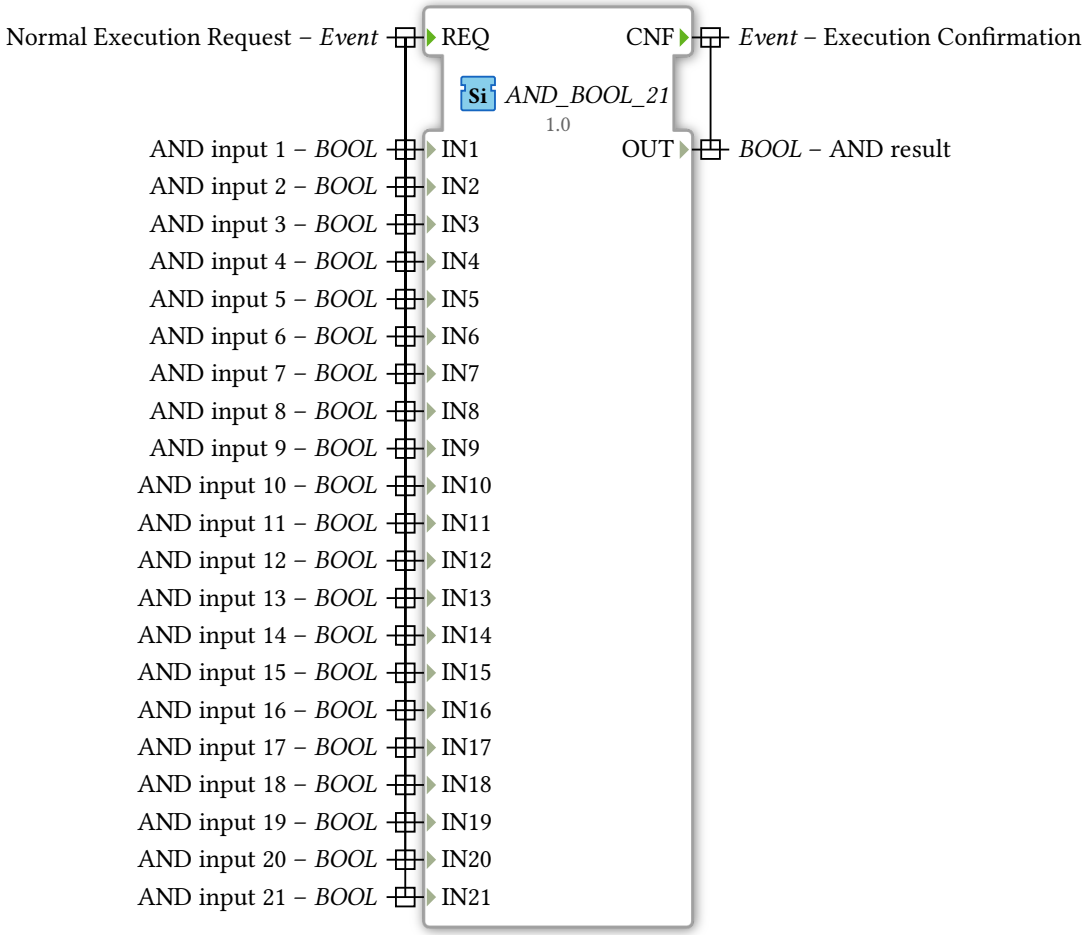
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AND_BOOL_21

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AND_BOOL_21` ist ein generischer Baustein zur Berechnung der logischen UND-Verknüpfung (Klassifikation `standard boolean function`, IEC 61131-3). Er verknüpft 21 separate BOOL-Eingänge (`IN1` bis `IN21`) und liefert das Ergebnis am Ausgang `OUT`. Er ergänzt die Standard-AND_BOOL-Familie (bisher bis `AND_BOOL_10`) um die Variante mit 21 Eingängen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Löst die Berechnung aus. Das Ereignis ist mit allen 21 Daten-Eingängen (`IN1` bis `IN21`) verknüpft.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Signalisiert den Abschluss der Berechnung. Das Ereignis ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN1 (AND input 1):** Boolescher Eingang 1.
- **IN2 (AND input 2):** Boolescher Eingang 2.
- **IN3 (AND input 3):** Boolescher Eingang 3.
- **IN4 (AND input 4):** Boolescher Eingang 4.
- **IN5 (AND input 5):** Boolescher Eingang 5.
- **IN6 (AND input 6):** Boolescher Eingang 6.
- **IN7 (AND input 7):** Boolescher Eingang 7.
- **IN8 (AND input 8):** Boolescher Eingang 8.
- **IN9 (AND input 9):** Boolescher Eingang 9.
- **IN10 (AND input 10):** Boolescher Eingang 10.
- **IN11 (AND input 11):** Boolescher Eingang 11.
- **IN12 (AND input 12):** Boolescher Eingang 12.
- **IN13 (AND input 13):** Boolescher Eingang 13.
- **IN14 (AND input 14):** Boolescher Eingang 14.
- **IN15 (AND input 15):** Boolescher Eingang 15.
- **IN16 (AND input 16):** Boolescher Eingang 16.

- **IN17 (AND input 17):** Boolescher Eingang 17.
- **IN18 (AND input 18):** Boolescher Eingang 18.
- **IN19 (AND input 19):** Boolescher Eingang 19.
- **IN20 (AND input 20):** Boolescher Eingang 20.
- **IN21 (AND input 21):** Boolescher Eingang 21.

Daten-Ausgänge

- **OUT (AND result):** Das Ergebnis der UND-Verknüpfung aller 21 Eingänge. Der Ausgang ist nur TRUE, wenn **alle** Eingänge `IN1` bis `IN21` den Wert TRUE haben, sonst FALSE.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei jedem Eintreffen des Ereignisses `REQ` werden alle 21 Eingänge gelesen und die Operation `OUT = IN1 AND IN2 AND ... AND IN21` berechnet. Anschließend wird das Ergebnis an `OUT` bereitgestellt und `CNF` ausgelöst.

Technische Besonderheiten

- **Generischer Baustein:** Der Typ trägt das Attribut `GenericClassName = 'GEN_AND_BOOL'`; diese Instanz ist fest auf 21 Eingänge ausgelegt.
- **Package:** `iec61131::booleanOperators`.
- **Zustandslos:** Der Baustein ist speicherlos: `OUT` hängt ausschließlich von den Werten der Eingänge zum Zeitpunkt von `REQ` ab.

Anwendungsszenarien

- **Freigabe- und Sicherheitsverknüpfungen:** Alle Bedingungen (z. B. Türkontakte, Not-Aus, Betriebsbereitschaft) müssen erfüllt sein, bevor eine Aktion starten darf.
- **Mehrfachbedingungen:** Prüfen, ob viele Voraussetzungen gleichzeitig erfüllt sind, ohne mehrere kleinere AND-Bausteine kaskadieren zu müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- `OR_BOOL_21`: Führt die ODER-Verknüpfung durch: Das Ergebnis ist TRUE, sobald mindestens ein Eingang TRUE ist, während `AND_BOOL_21` verlangt, dass alle Eingänge TRUE sind.
- `AND_BOOL_10`: Gleichartiger Baustein der Familie mit 10 Eingängen.

Fazit

`AND_BOOL_21` ist ein einfacher, ereignisgesteuerter Baustein für die UND-Verknüpfung von 21 booleschen Signalen mit fester Schnittstelle.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

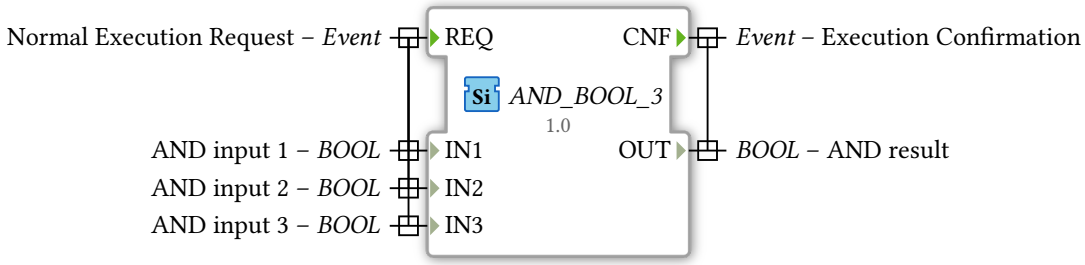
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AND_BOOL_3

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AND_BOOL_3` ist ein standardisierter Baustein zur Berechnung der logischen UND-Verknüpfung für drei boolesche Eingangssignale. Er folgt dem IEC 61131-3 Standard für grundlegende boolesche Funktionen und arbeitet nach dem Echtzeit-Execution-Control-Modell von 4diac, bei dem die Berechnung durch ein eingehendes Ereignis gestartet wird.



Schnittstellenstruktur

Die Schnittstelle des `AND_BOOL_3`-Blocks ist klar und einfach aufgebaut, bestehend aus einem Ereigniseingang, einem Ereignisausgang, drei Dateneingängen und einem Datenausgang.

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Dieses eingehende Ereignis löst die Ausführung des Funktionsblocks aus. Beim Eintreffen von `REQ` werden die aktuellen Werte an den Dateneingängen `IN1`, `IN2` und `IN3` gelesen und die logische UND-Operation durchgeführt.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Dieses Ereignis wird nach Abschluss der Berechnung erzeugt. Es signalisiert, dass das Ergebnis an den Datenausgang `OUT` ausgegeben wurde und nun gültig ist.

Daten-Eingänge

- **IN1 (BOOL):** Erstes boolesches Eingangssignal für die UND-Verknüpfung.
- **IN2 (BOOL):** Zweites boolesches Eingangssignal für die UND-Verknüpfung.
- **IN3 (BOOL):** Drittes boolesches Eingangssignal für die UND-Verknüpfung.

Daten-Ausgänge

- **OUT (BOOL):** Das Ergebnis der logischen UND-Verknüpfung aller drei Eingänge. `OUT` ist nur dann `TRUE` (1), wenn **alle** Eingänge `IN1`, `IN2` und `IN3` gleichzeitig `TRUE` sind. In allen anderen Fällen ist `OUT` `FALSE` (0).

Adapter

Dieser Funktionsblock verfügt über keine Adapter-Schnittstellen.

Funktionsweise

Die Funktionsweise ist deterministisch und folgt einer einfachen Abfolge:

1. **Auslösung:** Das eingehende Ereignis `REQ` aktiviert den Block.
2. **Lesen der Eingänge:** Die aktuellen Werte der drei booleschen Dateneingänge (`IN1`, `IN2`, `IN3`) werden eingelesen.
3. **Berechnung:** Die logische UND-Operation wird ausgeführt: `OUT := IN1 AND IN2 AND IN3`.
4. **Ausgabe:** Das berechnete Ergebnis wird an den Datenausgang `OUT` geschrieben.
5. **Bestätigung:** Das Ausgangsereignis `CNF` wird erzeugt, um den erfolgreichen Abschluss der Operation an nachfolgende Blöcke zu signalisieren.

Technische Besonderheiten

- **Generischer FB:** Der Block ist als generischer Funktionsblock (`GEN_AND`) gekennzeichnet, was bedeutet, dass er als Basis für die Erstellung ähnlicher UND-Bausteine mit einer anderen Anzahl von Eingängen dienen kann.
- **Echtzeitfähig:** Durch das Execution-Control-Modell (Ereignissteuerung) ist der Block für deterministische Echtzeitanwendungen geeignet.
- **Einfache Logik:** Enthält keine internen Zustände oder Speicher. Die Ausgabe ist eine reine Kombinatorikfunktion der aktuellen Eingänge.

Zustandsübersicht

Der `AND_BOOL_3`-Block ist ein **kombinatorischer** Funktionsblock ohne internen Zustand (zustandslos). Sein Verhalten ist bei jedem Aufruf (`REQ`) ausschließlich von den aktuellen Werten an den Dateneingängen abhängig. Es gibt keine vorherigen Berechnungen oder Zustände, die das Ergebnis beeinflussen.

Anwendungsszenarien

Typische Anwendungen sind Steuerungslogiken, bei denen eine Aktion nur dann ausgeführt werden darf, wenn mehrere Sicherheits- oder Freigabebedingungen gleichzeitig erfüllt sind:

- **Zweihand-Bedienung:** Eine Maschine startet nur, wenn drei separate Taster gleichzeitig gedrückt werden (`IN1`, `IN2`, `IN3`).
- **Sicherheitskreis:** Aktivierung eines Prozesses nur bei geschützter Schutztür (`IN1`), aktivierter Not-Aus-Umgehung (`IN2`) und vorhandener Materialzufuhr (`IN3`).
- **Verkettung von Bedingungen:** Als Teil einer komplexeren logischen Gleichung innerhalb eines Algorithmus.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`AND_BOOL_2`:** Führt die UND-Verknüpfung für nur zwei Eingänge durch. `AND_BOOL_3` ist eine direkte Erweiterung für drei Eingänge.
- **`AND_n_BOOL` (theoretisch):** Ein generischer Block für eine variable Anzahl `n` von Eingängen. `AND_BOOL_3` ist eine spezifische, fest verdrahtete Instanz für `n=3`. Siehe: AND_3
- **`OR_BOOL_3`:** Berechnet die logische ODER-Verknüpfung (Ausgang ist `TRUE`, wenn mindestens ein Eingang `TRUE` ist). Dies ist die komplementäre Funktion.
- **Kombinatorische Logik in SFC/ST:** Die gleiche Funktion könnte auch in Structured Text (ST) als `OUT := IN1 & IN2 & IN3;` oder in einer SFC-Aktion implementiert werden. Der Vorteil des FB ist die Wiederverwendbarkeit, klare Kapselung und die einfache Integration in das 4diac-Ereignissystem.

Fazit

Der `AND_BOOL_3` ist ein fundamentaler, zuverlässiger und einfach zu verwendender Baustein für die boolesche Logik in 4diac-basierten Steuerungsanwendungen. Seine strenge Ereignissteuerung gewährleistet deterministisches Verhalten, und die klare Schnittstelle ermöglicht eine intuitive Integration in Funktionsbaustein-Netzwerke. Für alle Steuerungsaufgaben, die eine dreifache UND-Verknüpfung erfordern, ist dieser Block die Standardlösung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

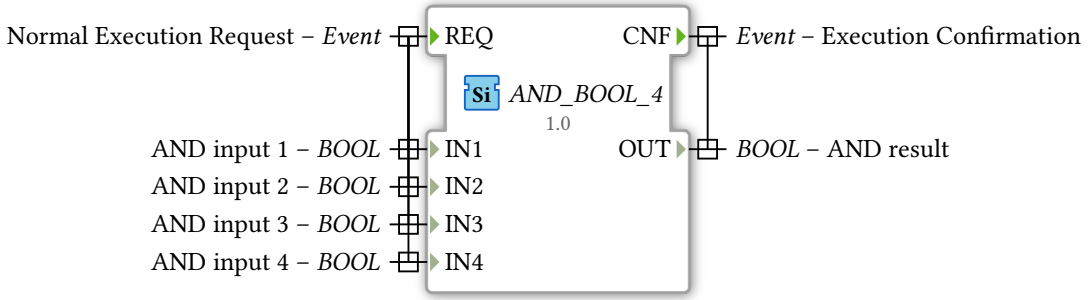
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AND_BOOL_4

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AND_BOOL_4` ist ein Standardbaustein zur Berechnung der logischen UND-Verknüpfung. Er führt eine bitweise UND-Operation über vier boolesche Eingangswerte aus. Der Baustein ist als generischer Funktionsblock implementiert und folgt dem IEC 61131-3 Standard für grundlegende boolesche Funktionen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Normal Execution Request): Dieses Ereignis löst die Ausführung des Funktionsblocks aus. Es ist mit allen vier Dateneingängen (`IN1` bis `IN4`) verknüpft.

Ereignis-Ausgänge

- **CNF** (Execution Confirmation): Dieses Ereignis signalisiert den Abschluss der Berechnung. Es wird zusammen mit dem Ergebnis am Datenausgang `OUT` ausgegeben.

Daten-Eingänge

- **IN1** (BOOL): Erster Operand für die UND-Verknüpfung.
- **IN2** (BOOL): Zweiter Operand für die UND-Verknüpfung.
- **IN3** (BOOL): Dritter Operand für die UND-Verknüpfung.
- **IN4** (BOOL): Vierter Operand für die UND-Verknüpfung.

Daten-Ausgänge

- **OUT** (BOOL): Ergebnis der logischen UND-Verknüpfung aller vier Eingänge. Der Ausgang ist nur `TRUE`, wenn **alle** Eingänge `IN1`, `IN2`, `IN3` und `IN4` den Wert `TRUE` haben.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei jedem Eintreffen eines `REQ`-Ereignisses liest der Funktionsblock die aktuellen Werte an den vier booleschen Dateneingängen. Anschließend berechnet er das logische UND dieser vier Werte gemäß der folgenden Wahrheitstabelle:

IN1	IN2	IN3	IN4	OUT
0	0	0	0	0
0	0	0	1	0
…	…	…	…	…
1	1	1	1	1

Das Ergebnis wird am Datenausgang `OUT` gesetzt und gleichzeitig wird das `CNF`-Ereignis ausgelöst, um den Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- **Generische Implementierung:** Der Baustein ist als generischer Funktionsblock (`GEN_AND`) gekennzeichnet, was bedeutet, dass er als Basis für ähnliche Bausteine mit einer anderen Anzahl von Eingängen dienen kann.
- **Ereignisgesteuerte Ausführung:** Die Operation wird ausschließlich durch ein eingehendes Ereignis (`REQ`) ausgelöst, was typisch für den Funktionsbaustein-Stil (FB) in IEC 61499 ist.
- **Standardkonformität:** Der Baustein ist gemäß IEC 61131-3 als Standard-Boolesche Funktion klassifiziert.

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand im Sinne eines Speichers. Sein Verhalten ist rein kombinatorisch und hängt nur von den aktuellen Eingangswerten zum Zeitpunkt der Auslösung ab. Der einzige „Zustand“ ist der Ruhezustand, in dem er auf das nächste `REQ`-Ereignis wartet.

Anwendungsszenarien

- **Sicherheitskette:** Verknüpfung mehrerer Sicherheitsschalter (z.B. Not-Aus, Schutzgitter, Zwei-Hand-Bedienung), die alle geschlossen sein müssen, um eine Maschine zu starten.
- **Bedingungsprüfung:** Überprüfung, ob mehrere Voraussetzungen in einem Prozessschritt gleichzeitig erfüllt sind.
- **Maskierung:** Selektives Aktivieren einer Funktion nur dann, wenn mehrere Freigabesignale vorliegen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **AND (2 Eingänge):** Ein grundlegender UND-Baustein mit nur zwei Eingängen. `AND_BOOL_4` konsolidiert vier solcher Operationen in einem einzigen Baustein, was die Lesbarkeit und Wartbarkeit des Diagramms verbessert. Siehe: AND_4
- **OR_BOOL_4:** Führt eine logische ODER-Verknüpfung durch, bei der das Ergebnis `TRUE` ist, wenn mindestens ein Eingang `TRUE` ist. Dies ist das logische Gegenstück zum UND.
- **GEN_AND:** Die generische Basisklasse, auf der `AND_BOOL_4` aufbaut. Sie kann für die Erstellung von UND-Bausteinen mit einer beliebigen Anzahl von Eingängen konfiguriert werden.

Fazit

Der `AND_BOOL_4` Funktionsblock ist ein einfacher, robuster und standardkonformer Baustein zur Verarbeitung von Logiksignalen. Seine klare ereignisgesteuerte Schnittstelle und die feste Anzahl von vier Eingängen machen ihn besonders geeignet für Steuerungsaufgaben, bei denen mehrere Bedingungen gleichzeitig erfüllt sein müssen. Als Teil der `iec61131::bitwiseOperators` Bibliothek ist er eine grundlegende Komponente für den Entwurf von Steuerungslogik in 4diac.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

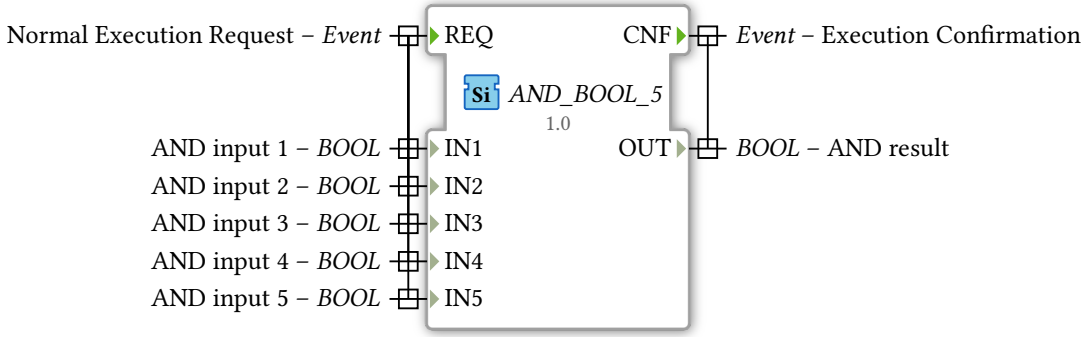
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AND_BOOL_5

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AND_BOOL_5` ist ein Standard-Baustein zur Berechnung der logischen UND-Verknüpfung. Er führt eine bitweise UND-Operation über fünf separate boolesche Eingangswerte aus. Der Block folgt dem Echtzeit- und Ereignisverarbeitungsmodell von IEC 61499, bei dem die Berechnung durch ein eingehendes Ereignis ausgelöst und mit einem ausgehenden Ereignis bestätigt wird.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Ausführung des Funktionsblocks aus. Bei Eintreffen dieses Ereignisses werden alle fünf Eingänge (`IN1` bis `IN5`) gelesen und die UND-Operation berechnet.

Ereignis-Ausgänge

- **CNF**: Bestätigt die abgeschlossene Ausführung. Dieses Ereignis wird nach der Berechnung des Ergebnisses ausgegeben und liefert den berechneten Wert `OUT` mit.

Daten-Eingänge

- **IN1**: Boolescher Eingang 1 für die UND-Operation.
- **IN2**: Boolescher Eingang 2 für die UND-Operation.
- **IN3**: Boolescher Eingang 3 für die UND-Operation.
- **IN4**: Boolescher Eingang 4 für die UND-Operation.
- **IN5**: Boolescher Eingang 5 für die UND-Operation.

Daten-Ausgänge

- **OUT**: Das boolesche Ergebnis der UND-Verknüpfung aller fünf Eingänge. Der Ausgang ist nur `TRUE`, wenn **alle** Eingänge `IN1` bis `IN5` den Wert `TRUE` haben.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Die Funktionsweise ist deterministisch und ereignisgesteuert:

1. Das eingehende Ereignis `REQ` aktiviert den Block.
2. Die aktuellen Werte aller fünf booleschen Dateneingänge (`IN1` bis `IN5`) werden eingelesen.
3. Die logische UND-Operation wird berechnet: `OUT = IN1 AND IN2 AND IN3 AND IN4 AND IN5`.
4. Das Ergebnis wird am Datenausgang `OUT` bereitgestellt.
5. Das Ausgangsereignis `CNF` wird zusammen mit dem neuen Wert von `OUT` ausgelöst, um den Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- **Generischer Baustein**: Der Block ist als Instanz eines generischen UND-Bausteins (`GEN_AND`) implementiert, der für eine variable Anzahl von Eingängen parametrisiert werden kann. Diese spezielle Instanz ist für genau fünf Eingänge festgelegt.
- **Ereigniskette**: Die Abarbeitung folgt strikt dem REQ/CNF-Pattern, was eine einfache Integration in ereignisgesteuerte Steuerungsabläufe ermöglicht.
- **Kein interner Zustand**: Der Block ist speicherfrei (kombinatorisch). Das Ausgangssignal hängt ausschließlich von den aktuellen Eingangswerten zum Zeitpunkt der `REQ`-Anforderung ab.

Zustandsübersicht

Der Funktionsblock besitzt keinen persistenten internen Zustand. Sein Verhalten ist rein kombinatorisch und wird vollständig durch die Schnittstellenereignisse und -daten beschrieben. Der „Zustand“ ist der Übergang von wartend (`REQ` abwarten) zu aktiv (Berechnung durchführen und `CNF` auslösen).

Anwendungsszenarien

Typische Anwendungen sind Sicherheits- und Freigabeketten in der Automatisierungstechnik, bei denen mehrere Bedingungen gleichzeitig erfüllt sein müssen:

- Überwachung, ob alle Schutzgitter einer Maschine geschlossen sind.
- Prüfung, ob alle erforderlichen Medien (Druck, Vakuum, Kühlung) verfügbar sind, bevor ein Prozess gestartet wird.
- Verknüpfung mehrerer Sensor- oder Schalterzustände zu einer gemeinsamen Freigabelogik.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`AND_BOOL_2` / `AND_BOOL_3` etc.**: Grundlegende UND-Bausteine mit zwei, drei oder vier Eingängen. `AND_BOOL_5` erweitert diese Logik auf fünf Eingänge, ohne dass mehrere Blöcke verkettet werden müssen. Siehe: AND_5
- **`GEN_AND`**: Der zugrundeliegende generische Baustein. `AND_BOOL_5` ist eine vorkonfigurierte, nicht veränderbare Instanz davon mit fester Eingangsanzahl.
- **OR-Bausteine**: Führen eine logische ODER-Verknüpfung durch. Während `AND_BOOL_5` nur bei allen `TRUE`-Eingängen ein `TRUE` ausgibt, geben OR-Bausteine bereits bei einem `TRUE`-Eingang ein `TRUE` aus.

Fazit

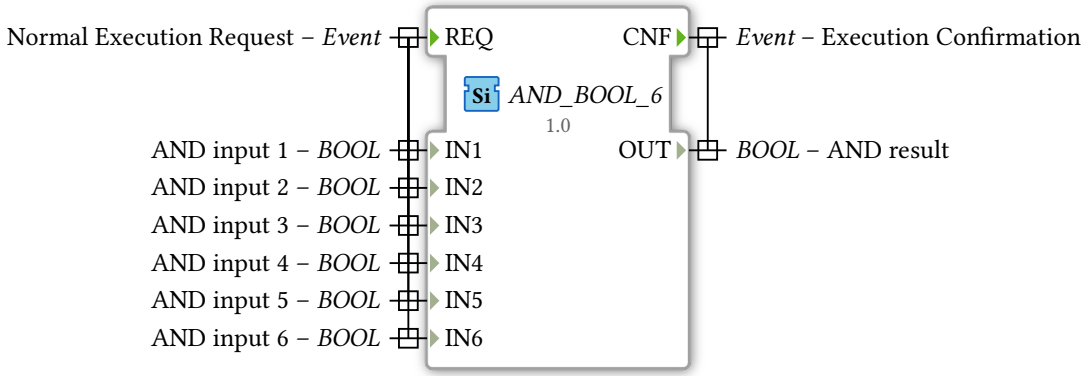
Der `AND_BOOL_5` ist ein robuster und einfach zu verwendender Standardbaustein für logische Verknüpfungen mit fünf Eingängen. Seine ereignisgesteuerte, zustandslose Architektur macht ihn gut vorhersagbar und einfach in komplexere IEC 61499-Applikationen zu integrieren. Für Anwendungen, die genau fünf Bedingungen verknüpfen müssen, bietet er eine optimale und lesbare Lösung.

AND_BOOL_6

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AND_BOOL_6` ist ein Standardbaustein zur Berechnung der logischen UND-Verknüpfung. Er führt eine bitweise UND-Operation über sechs separate boolesche Eingänge durch. Der Block gehört zur Kategorie der Standard-Booleschen Funktionen und ist für den Einsatz in Steuerungsanwendungen nach IEC 61131-3 konzipiert.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Normal Execution Request): Dieses Ereignis löst die Berechnung der UND-Verknüpfung aus. Beim Eintreffen von `REQ` werden die Werte aller sechs Daten-Eingänge (`IN1` bis `IN6`) ausgelesen und verarbeitet.

Ereignis-Ausgänge

- **CNF** (Execution Confirmation): Dieses Ereignis signalisiert den Abschluss der Berechnung. Es wird zusammen mit dem berechneten Ergebnis am Datenausgang `OUT` ausgegeben.

Daten-Eingänge

- **IN1** (BOOL): UND-Eingang 1.
- **IN2** (BOOL): UND-Eingang 2.
- **IN3** (BOOL): UND-Eingang 3.
- **IN4** (BOOL): UND-Eingang 4.
- **IN5** (BOOL): UND-Eingang 5.
- **IN6** (BOOL): UND-Eingang 6.

Daten-Ausgänge

- **OUT** (BOOL): Ergebnis der UND-Verknüpfung. Der Ausgang ist nur `TRUE`, wenn **alle** sechs Eingänge (`IN1` bis `IN6`) den Wert `TRUE` haben. In allen anderen Fällen ist der Ausgang `FALSE`.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Die Funktionsweise ist deterministisch und einfach: Bei jedem Eintreffen des Ereignisses `REQ` wird der logische Zustand aller sechs Eingänge ausgewertet. Das Ergebnis der UND-Operation wird am Datenausgang `OUT` bereitgestellt und gleichzeitig wird das Bestätigungsereignis `CNF` ausgelöst. Die logische Operation entspricht der Formel: `OUT = IN1 AND IN2 AND IN3 AND IN4 AND IN5 AND IN6`

Technische Besonderheiten

- **Generischer Baustein:** Der Block ist als generischer Baustein (`GEN_AND`) implementiert, was bedeutet, dass er eine spezifische Instanz einer allgemeineren UND-Funktionsblockfamilie ist.
- **Feste Eingangsanzahl:** Im Gegensatz zu Bausteinen mit variabler Eingangsanzahl besitzt dieser Block genau sechs fest definierte Eingänge.
- **Ereignisgesteuert:** Die Operation wird ausschließlich durch ein eingehendes Ereignis (`REQ`) ausgeführt, was typisch für die Funktionsbaustein-Architektur (FB) in 4diac ist.

Zustandsübersicht

Der Baustein besitzt keinen internen Zustand oder Speicher. Sein Verhalten ist rein kombinatorisch und hängt ausschließlich von den aktuellen Werten der Eingänge zum Zeitpunkt der Auslösung durch `REQ` ab.

Anwendungsszenarien

Typische Anwendungen sind Sicherheits- und Freigabeketten in Steuerungslogik, bei denen mehrere Bedingungen gleichzeitig erfüllt sein müssen, bevor eine Aktion ausgeführt wird.

- **Maschinenstart:** Starten einer Maschine nur, wenn alle Schutztüren geschlossen (`TRUE`) und alle Not-Aus-Taster nicht gedrückt (`TRUE`) sind.
- **Prozessfreigabe:** Freigabe eines Prozessschritts nur, wenn alle vorgelagerten Schritte erfolgreich abgeschlossen (`TRUE`) und alle erforderlichen Ressourcen verfügbar (`TRUE`) sind.
- **Mehrfachsensor-Abfrage:** Auswertung mehrerer Sensoren, die alle ein Signal geben müssen, um eine Position oder einen Zustand zu bestätigen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **AND (mit 2 Eingängen):** Der Standard-`AND`-Block hat typischerweise nur zwei Eingänge. `AND_BOOL_6` bietet eine spezialisierte, kompaktere Darstellung für Anwendungen mit genau sechs Eingängen, ohne dass mehrere Blöcke verkettet werden müssen. Siehe: AND_6
- **GEN_AND (Generischer UND-Baustein):** `AND_BOOL_6` ist eine konkrete Instanz dieses generischen Typs. Andere Instanzen könnten eine andere Anzahl von Eingängen haben (z.B. `AND_BOOL_4`).
- **OR_BOOL_6:** Ein ähnlicher Baustein, der jedoch eine logische ODER-Verknüpfung durchführt, bei der das Ergebnis `TRUE` ist, wenn mindestens ein Eingang `TRUE` ist.

Fazit

Der `AND_BOOL_6`-Funktionsblock ist ein zuverlässiger und einfach anzuwendender Baustein für logische UND-Verknüpfungen mit sechs Eingängen. Seine ereignisgesteuerte, zustandslose Natur macht ihn vorhersehbar und gut in die 4diac-Architektur integrierbar. Er ist die optimale Wahl, wenn eine feste Anzahl von sechs Bedingungen gleichzeitig erfüllt sein muss, um eine Folgeaktion freizugeben.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

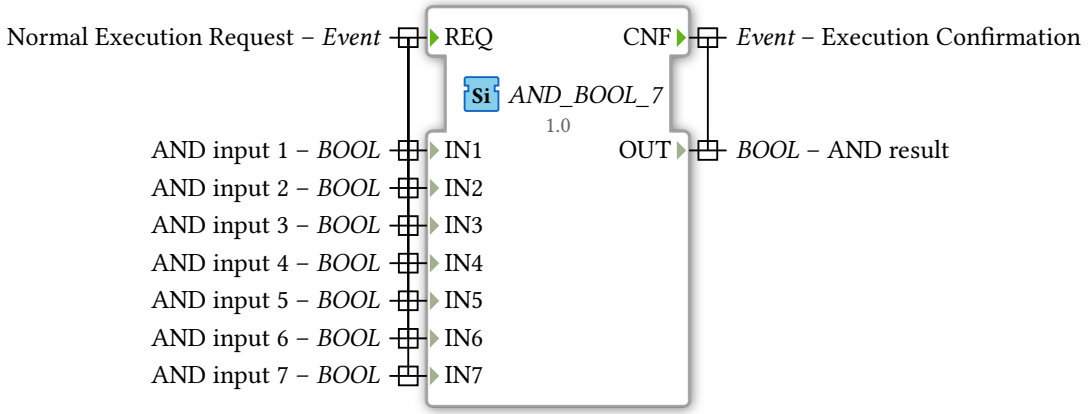
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AND_BOOL_7

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AND_BOOL_7` ist ein generischer Baustein zur Berechnung der logischen UND-Verknüpfung. Er führt eine bitweise UND-Operation über sieben separate BOOL-Eingänge durch und gibt das Ergebnis auf einem einzigen BOOL-Ausgang aus. Der Block folgt dem ECC-Ausführungsmodell, bei dem eine Berechnung durch ein ankommendes Ereignis ausgelöst und mit einem ausgehenden Ereignis bestätigt wird.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Berechnung der UND-Verknüpfung aus. Beim Eintreffen dieses Ereignisses werden alle Daten-Eingänge (`IN1` bis `IN7`) ausgelesen und verarbeitet.

Ereignis-Ausgänge

- **CNF**: Bestätigt die abgeschlossene Ausführung. Dieses Ereignis wird nach der Berechnung generiert und liefert das Ergebnis `OUT` mit.

Daten-Eingänge

- **IN1**: UND-Eingang 1 (BOOL)
- **IN2**: UND-Eingang 2 (BOOL)
- **IN3**: UND-Eingang 3 (BOOL)
- **IN4**: UND-Eingang 4 (BOOL)
- **IN5**: UND-Eingang 5 (BOOL)
- **IN6**: UND-Eingang 6 (BOOL)
- **IN7**: UND-Eingang 7 (BOOL)

Daten-Ausgänge

- **OUT**: Ergebnis der UND-Verknüpfung (BOOL). Der Ausgang ist nur `TRUE`, wenn **alle** sieben Eingänge (`IN1` bis `IN7`) den Wert `TRUE` haben. In allen anderen Fällen ist der Ausgang `FALSE`.

Adapter

Dieser Funktionsblock verfügt über keine Adapter-Schnittstellen.

Funktionsweise

Die Funktionsweise ist deterministisch und folgt der booleschen Algebra. Bei jedem Eintreffen des Ereignisses `REQ` wird die folgende Operation ausgeführt: `OUT := IN1 AND IN2 AND IN3 AND IN4 AND IN5 AND IN6 AND IN7`

Unmittelbar nach der Berechnung des Ergebnisses wird das Bestätigungsereignis `CNF` zusammen mit dem aktuellen Wert von `OUT` ausgegeben.

Technische Besonderheiten

- **Generischer Baustein**: Der Block ist als generischer Baustein gekennzeichnet (`GEN_AND`). Dies bedeutet, dass er als Vorlage für die automatische Generierung ähnlicher Blöcke mit einer anderen Anzahl von Eingängen dienen kann.
- **Ereignisgesteuert**: Die Operation wird nur bei Bedarf durch ein Ereignis ausgeführt, was Ressourcen schont und eine klare Ablaufsteuerung ermöglicht.
- **Feste Eingangsanzahl**: Dieser spezifische Block hat eine feste Anzahl von sieben Eingängen. Für eine andere Anzahl von Operanden muss ein entsprechender Block (z.B. `AND_BOOL_2`, `AND_BOOL_4`) verwendet oder ein generischer Block instanziiert werden.

Zustandsübersicht

Der Block besitzt keinen internen Zustand im Sinne eines Speichers. Sein Verhalten ist rein kombinatorisch und ereignisgesteuert:

1. **Ruhezustand**: Wartet auf das Ereignis `REQ`.
2. **Ausführungszustand**: Bei `REQ` werden alle Eingänge gelesen, die UND-Operation berechnet und das Ergebnis `OUT` gesetzt.
3. **Bestätigungszustand**: Das Ereignis `CNF` wird sofort gesendet, danach kehrt der Block in den Ruhezustand zurück.

Anwendungsszenarien

- **Sicherheitskette**: Verknüpfung mehrerer Sicherheitsschalter (z.B. Not-Aus, Schutzgitter, Lichtschranke), wobei eine Maschine nur starten darf, wenn alle Schalter geschlossen (TRUE) sind.
- **Mehrfachbedingung**: Prüfung, ob mehrere parallele Prozessschritte erfolgreich abgeschlossen wurden, bevor ein Folgeprozess gestartet wird.
- **Steuerungslogik**: Implementierung von komplexen UND-Bedingungen in Ablaufsteuerungen oder Verknüpfungssteuerungen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`AND_n_BOOL`-Familie**: Es existieren Blöcke mit identischer Funktion, aber unterschiedlicher Anzahl von Eingängen (z.B., `AND_BOOL_2`, `AND_BOOL_8`). `AND_BOOL_7` füllt die Lücke für Anwendungen, die genau sieben Bedingungen benötigen. Siehe: AND_7
- **`&` (Bit-AND) Operator**: Während `AND_BOOL_7` nur mit BOOL-Datentypen arbeitet und ereignisgesteuert ist, kann der `&`-Operator in Structured Text (ST) auch auf Byte-, Wort- oder DWORD-Datentypen angewendet werden und wird zyklisch oder anweisungsgesteuert ausgewertet.
- **Serielle UND-Verknüpfung**: Das gleiche Ergebnis könnte durch die serielle Verkettung mehrerer `AND_BOOL_2`-Blöcke erreicht werden. `AND_BOOL_7` bietet eine kompaktere und übersichtlichere Lösung.

Fazit

Der `AND_BOOL_7` ist ein spezialisierter, zuverlässiger und einfach zu verwendender Funktionsblock für die logische UND-Verknüpfung von sieben booleschen Signalen. Seine ereignisgesteuerte Architektur macht ihn ideal für die Integration in steuerungstypische Ablaufstrukturen innerhalb der 4diac-IDE. Für Anwendungen mit einer anderen Anzahl von Eingängen sollte auf die entsprechende Variante der `AND_n_BOOL`-Familie zurückgegriffen werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

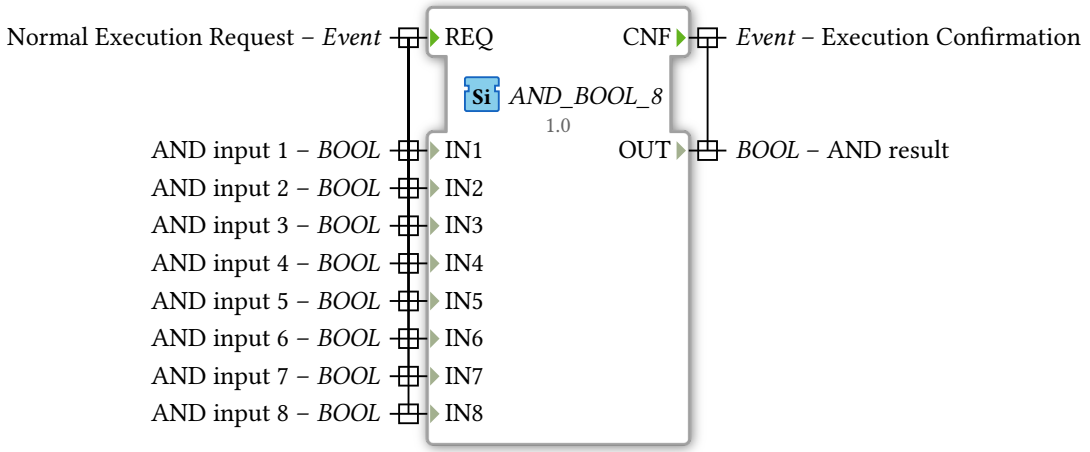
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AND_BOOL_8

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AND_BOOL_8` ist ein Standardbaustein zur Berechnung der logischen UND-Verknüpfung. Er führt eine bitweise UND-Operation auf acht boolesche Eingangswerte aus und liefert das Ergebnis auf einen einzigen booleschen Ausgang. Der Baustein folgt dem ereignisgesteuerten Ausführungsmodell von IEC 61499.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Löst die Ausführung des Funktionsblocks aus. Bei Eintreffen dieses Ereignisses werden alle acht Daten-Eingänge gelesen, die UND-Operation berechnet und das Ergebnis am Ausgang `OUT` bereitgestellt. Anschließend wird das Bestätigungsereignis `CNF` generiert.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Dieses Ereignis wird nach der erfolgreichen Berechnung und Aktualisierung des Ausgangs `OUT` ausgegeben. Es signalisiert, dass ein neues Ergebnis vorliegt.

Daten-Eingänge

- **IN1 (BOOL):** UND-Eingang 1.

- **IN2 (BOOL):** UND-Eingang 2.
- **IN3 (BOOL):** UND-Eingang 3.
- **IN4 (BOOL):** UND-Eingang 4.
- **IN5 (BOOL):** UND-Eingang 5.
- **IN6 (BOOL):** UND-Eingang 6.
- **IN7 (BOOL):** UND-Eingang 7.
- **IN8 (BOOL):** UND-Eingang 8.

Daten-Ausgänge

- **OUT (BOOL):** Ergebnis der UND-Verknüpfung aller acht Eingänge. Der Ausgang ist nur dann `TRUE` (1), wenn **alle** Eingänge `IN1` bis `IN8` den Wert `TRUE` haben. In allen anderen Fällen ist der Ausgang `FALSE` (0).

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Die Funktionsweise ist deterministisch und folgt der klassischen booleschen Logik. Bei Auslösung durch das Ereignis `REQ` wertet der Block die Zustände aller acht Eingänge (`IN1` bis `IN8`) aus. Die logische Operation wird gemäß der folgenden Wahrheitstabelle ausgeführt:

IN1	IN2	IN3	IN4	IN5	IN6	IN7	IN8	OUT
0	X	X	X	X	X	X	X	0
1	0	X	X	X	X	X	X	0
1	1	0	X	X	X	X	X	0
…	…	…	…	…	…	…	…	…
1	1	1	1	1	1	1	1	1

(X = Don't-Care / Beliebiger Wert)

Das Ergebnis `OUT = IN1 AND IN2 AND IN3 AND IN4 AND IN5 AND IN6 AND IN7 AND IN8` wird unmittelbar nach der Berechnung gesetzt. Die Ausführung wird durch das Auslösen des Bestätigungsereignisses `CNF` abgeschlossen, welches das neue Ergebnis am Datenausgang `OUT` begleitet.

Technische Besonderheiten

- **Generischer Baustein:** Der Block ist als Instanz eines generischen UND-Templates (`GEN_AND`) implementiert, was eine konsistente Architektur innerhalb der Bibliothek gewährleistet.
- **Fest verdrahtete Eingangsanzahl:** Im Gegensatz zu variablen oder konfigurierbaren Blöcken besitzt `AND_BOOL_8` genau acht fest definierte Eingänge. Für eine andere Anzahl von Operanden muss ein entsprechender anderer Baustein (z.B. `AND_BOOL_2`, `AND_BOOL_16`) verwendet werden.
- **Ereignisgesteuerte Ausführung:** Die Berechnung findet nur bei Ankunft eines `REQ`-Ereignisses statt. Kontinuierliche Abtastung der Eingänge ohne Ereignis führt zu keiner Aktualisierung des Ausgangs.

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand (speicherfrei / kombinatorisch). Sein Verhalten hängt ausschließlich von den aktuellen Werten an den Daten-Eingängen zum Zeitpunkt des `REQ`-Ereignisses ab. Vorherige Berechnungen haben keinen Einfluss auf das aktuelle Ergebnis.

Anwendungsszenarien

- **Sicherheitskritische Verknüpfungen:** Überwachung mehrerer Sicherheitsschalter (z.B. Not-Aus, Schutzgitter, Lichtschranken), bei denen alle Bedingungen erfüllt sein müssen, um einen sicheren Zustand (`OUT=TRUE`) zu signalisieren.
- **Mehrfachbedingungen in Prozessabläufen:** Prüfung, ob alle Voraussetzungen für den Start eines Maschinenzyklus erfüllt sind (z.B. „Werkstück eingelegt", „Druck erreicht", „Temperatur erreicht", „Schutzhaube geschlossen").
- **Verknüpfung von Statusmeldungen:** Kombination mehrerer Fehler- oder Statusbits zu einem zusammenfassenden Signal.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`AND_BOOL_2`, `AND_BOOL_16` etc.:** Diese Blöcke bieten die identische logische Funktion, jedoch für eine unterschiedliche Anzahl von Eingängen (2, 16, etc.). Die Wahl des Blocks richtet sich nach der benötigten Anzahl an Operanden. Siehe: AND_8
- **`OR_n_BOOL`:** Führt eine logische ODER-Verknüpfung durch. Das Ergebnis ist `TRUE`, wenn mindestens ein Eingang `TRUE` ist.
- **`XOR_n_BOOL`:** Führt eine exklusiv-ODER (XOR) Verknüpfung durch.
- **Kontinuierliche Funktionsbausteine (IEC 61131-3):** In der IEC 61131-3 sind UND-Verknüpfungen typischerweise als kontinuierlich wirkende Kontakte oder Operationen in Sprachen wie FBD oder LD realisiert. Der `AND_BOOL_8`-FB hingegen ist ereignisgesteuert und benötigt explizit ein Auslöseereignis.

Fazit

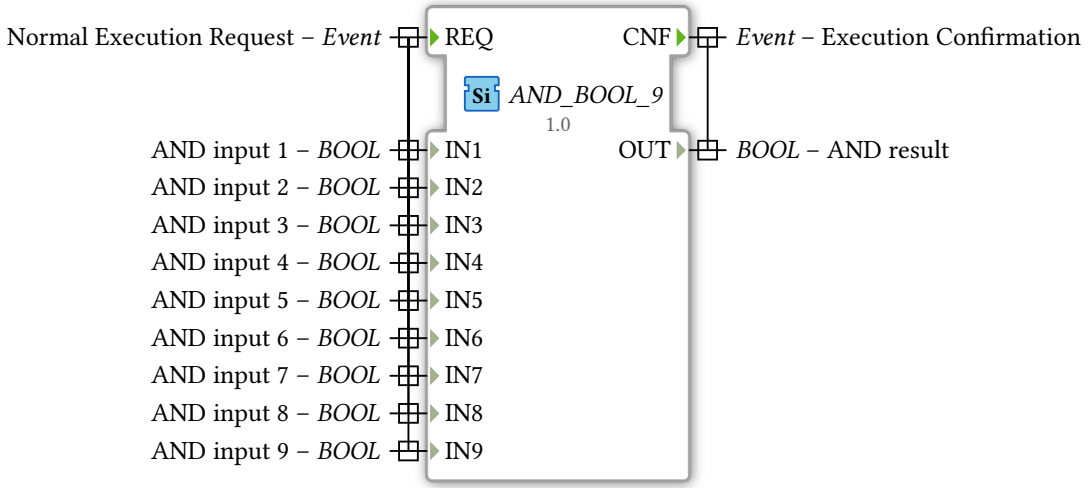
Der `AND_BOOL_8` ist ein grundlegender, zuverlässiger und einfach zu verwendender Logikbaustein für IEC 61499-Systeme. Seine Stärke liegt in der klaren, ereignisgesteuerten Semantik und der expliziten Handhabung von acht booleschen Eingangssignalen. Er eignet sich ideal für Steuerungslogik, bei der mehrere Bedingungen gleichzeitig erfüllt sein müssen, bevor eine Aktion ausgeführt wird. Für Anwendungen mit einer von acht abweichenden Anzahl an Eingängen sollte auf die entsprechende Variante der `AND_n_BOOL`-Familie zurückgegriffen werden.

AND_BOOL_9

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AND_BOOL_9` ist ein Standardbaustein zur Berechnung der logischen UND-Verknüpfung. Er führt eine bitweise UND-Operation über neun separate boolesche Eingänge durch. Der Block folgt dem IEC 61131-3 Standard und ist für den Einsatz in der 4diac-IDE konzipiert.



Schnittstellenstruktur

Der Block besitzt ein ereignisgesteuertes Ein-/Ausgangsmodell.

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Löst die Berechnung der UND-Verknüpfung aus. Bei diesem Ereignis werden alle Daten-Eingänge (`IN1` bis `IN9`) ausgelesen und das Ergebnis berechnet.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Wird nach Abschluss der Berechnung ausgelöst und signalisiert, dass das Ergebnis am Daten-Ausgang `OUT` bereitsteht.

Daten-Eingänge

- **IN1 (BOOL):** UND-Eingang 1.
- **IN2 (BOOL):** UND-Eingang 2.
- **IN3 (BOOL):** UND-Eingang 3.

- **IN4 (BOOL):** UND-Eingang 4.
- **IN5 (BOOL):** UND-Eingang 5.
- **IN6 (BOOL):** UND-Eingang 6.
- **IN7 (BOOL):** UND-Eingang 7.
- **IN8 (BOOL):** UND-Eingang 8.
- **IN9 (BOOL):** UND-Eingang 9.

Daten-Ausgänge

- **OUT (BOOL):** Ergebnis der UND-Verknüpfung. Der Ausgang ist nur `TRUE`, wenn **alle** neun Eingänge (`IN1` bis `IN9`) den Wert `TRUE` aufweisen. In allen anderen Fällen ist der Ausgang `FALSE`.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Eintreffen eines Ereignisses am Eingang `REQ` liest der Block die aktuellen Werte aller neun booleschen Eingänge (`IN1` bis `IN9`) ein. Anschließend wird die logische UND-Verknüpfung über alle neun Werte berechnet. Das Ergebnis dieser Operation wird unmittelbar am Datenausgang `OUT` ausgegeben. Gleichzeitig wird ein Bestätigungsereignis am Ausgang `CNF` generiert, um den erfolgreichen Abschluss der Operation zu signalisieren.

Die logische Funktion lässt sich wie folgt beschreiben: `OUT = IN1 AND IN2 AND IN3 AND IN4 AND IN5 AND IN6 AND IN7 AND IN8 AND IN9`

Technische Besonderheiten

- **Generischer Baustein:** Der Block ist als Instanz eines generischen UND-Bausteins (`GEN_AND`) implementiert, der für eine feste Anzahl von Eingängen (hier 9) spezialisiert wurde.
- **Ereignisgesteuert:** Die Operation wird nur bei einem eingehenden `REQ`-Ereignis ausgeführt, was eine ressourcenschonende und deterministische Abarbeitung ermöglicht.
- **Feste Eingangsanzahl:** Im Gegensatz zu variablen Funktionen arbeitet dieser Block exakt mit neun Eingängen.

Zustandsübersicht

Der Block besitzt keinen internen Zustand im Sinne eines Speichers. Sein Verhalten ist kombinatorisch und ausschließlich von den aktuellen Werten der Eingänge zum Zeitpunkt des `REQ`-Ereignisses abhängig. Der Ausgang `OUT` behält seinen Wert bis zum nächsten Ausführungszyklus.

Anwendungsszenarien

- **Sicherheitskette:** Verknüpfung mehrerer Sicherheitsschalter (z.B. Not-Aus, Schutzgitter, Zwei-Hand-Bedienung), die alle geschlossen sein müssen, um eine Maschine zu starten.
- **Mehrfachbedingung:** Prüfung, ob mehrere Voraussetzungen in einem Prozessschritt gleichzeitig erfüllt sind.
- **Verkettung von Logik:** Als Teil einer größeren logischen Schaltung, wo das Ergebnis von neun Quellen gemeinsam ausgewertet werden muss.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`AND` (2 Eingänge):** Der Standard-UND-Baustein mit nur zwei Eingängen. `AND_BOOL_9` ist eine spezielle, erweiterte Version für genau neun Eingänge. Siehe: AND_9
- **`GEN_AND`:** Der zugrunde liegende generische Baustein, der mit einer konfigurierbaren Anzahl von Eingängen instanziiert werden kann. `AND_BOOL_9` ist eine vordefinierte Instanz davon.
- **`OR_*` / `XOR_*` Bausteine:** Führen andere logische Grundoperationen (ODER, exklusives ODER) mit einer festen Anzahl von Eingängen durch.

Fazit

Der `AND_BOOL_9` ist ein robuster und einfach zu verwendender Standardbaustein für Anwendungen, die eine logische UND-Verknüpfung über genau neun boolesche Signale erfordern. Seine ereignisgesteuerte Natur und die klare Schnittstelle machen ihn gut integrierbar in 4diac-basierte Steuerungsanwendungen. Für eine andere Anzahl von Eingängen sollten die entsprechenden spezialisierten (`AND_BOOL_2`, `AND_BOOL_3`, etc.) oder der generische `GEN_AND`-Block verwendet werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

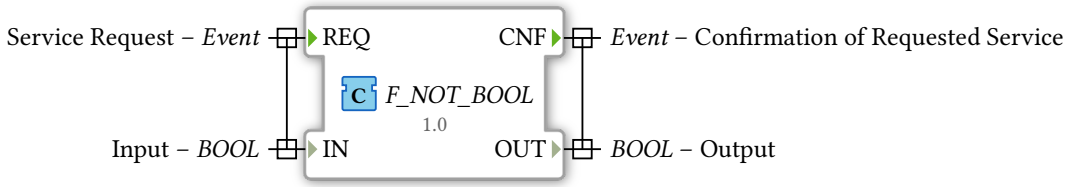
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

F_NOT_BOOL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_NOT_BOOL` implementiert die logische NICHT-Funktion (Negation) für den Datentyp `BOOL`. Er invertiert den logischen Zustand eines binären Eingangssignals. Dieser Baustein gehört zu den Standard-Bit-Operatoren gemäß IEC 61131-3 und ist für die Verwendung in Steuerungsanwendungen konzipiert, in denen logische Verknüpfungen benötigt werden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Service Request)**: Dieses Ereignis löst die Ausführung der logischen Negation aus. Bei seinem Eintreffen wird der aktuelle Wert am Daten-Eingang `IN` verarbeitet und das Ergebnis an `OUT` ausgegeben.

Ereignis-Ausgänge

- **CNF (Confirmation of Requested Service)**: Dieses Ereignis signalisiert die Beendigung der Operation. Es wird gleichzeitig mit der Ausgabe des berechneten Wertes am Daten-Ausgang `OUT` erzeugt.

Daten-Eingänge

- **IN (Input)**: Datentyp `BOOL`. Dies ist der binäre Eingangswert, der negiert werden soll. Gültige Werte sind `TRUE` (1) oder `FALSE` (0).

Daten-Ausgänge

- **OUT (Output)**: Datentyp `BOOL`. Dies ist das Ergebnis der logischen Negation. Der Ausgangswert entspricht dem logischen Gegenteil des Eingangswertes `IN`.

Adapter

Dieser Funktionsblock verwendet keine Adapter-Schnittstellen.

Funktionsweise

Die Funktionsweise ist deterministisch und einfach: Bei jedem Eintreffen eines `REQ`-Ereignisses liest der Baustein den aktuellen Wert am Daten-Eingang `IN`. Anschließend berechnet er die logische Negation und schreibt das Ergebnis auf den Daten-Ausgang `OUT`. Parallel dazu wird ein `CNF`-Ereignis generiert, um den Abschluss der Operation zu signalisieren.

Die logische Operation lässt sich durch folgende Tabelle beschreiben:

Eingang (IN)	Ausgang (OUT)
FALSE (0)	TRUE (1)
TRUE (1)	FALSE (0)

Technische Besonderheiten

- **Standardkonformität**: Der Baustein ist als Standardfunktion nach IEC 61131-3 klassifiziert.
- **E/A-Verzögerung**: Die Verarbeitung von `REQ` bis zur Ausgabe von `CNF` und `OUT` erfolgt innerhalb eines Zyklus der ausführenden Ressource. Es gibt keine inhärenten Verzögerungen.
- **Initialisierung**: Die Ausgangsvariablen werden bei der Initialisierung des Funktionsblocks auf ihren Standardwert (`FALSE` für `BOOL`) gesetzt.

Zustandsübersicht

Der Baustein besitzt keinen internen Zustand im Sinne einer Zustandsmaschine. Sein Verhalten ist rein kombinatorisch und ausschließlich von den aktuellen Eingangswerten bei Auslösung durch ein `REQ`-Ereignis abhängig. Es werden keine vorherigen Werte gespeichert.

Anwendungsszenarien

- **Invertierung von Sensorsignalen**: Umwandlung eines aktiven-Low-Sensorsignals (z.B. „Taste gedrückt“ = FALSE) in ein aktives-High-Signal für die weitere Logik.
- **Sicherheitslogik**: Erzeugung eines Sperrsignals aus einem Freigabesignal oder umgekehrt.
- **Steuerungslogik**: Als grundlegender Baustein in komplexeren logischen Ausdrücken und Bedingungen innerhalb von Ablaufsteuerungen.
- **Statusanzeige**: Umkehrung eines internen Status für die Ansteuerung einer Lampe (z.B. „Maschine bereit“ = TRUE, aber „Störungslampe leuchtet“ bei FALSE).

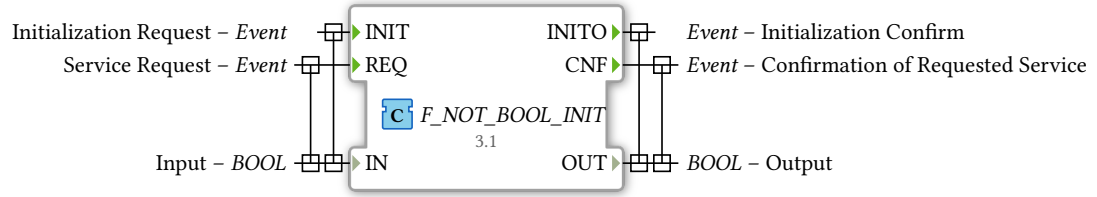
⚖️ Vergleich mit ähnlichen Bausteinen

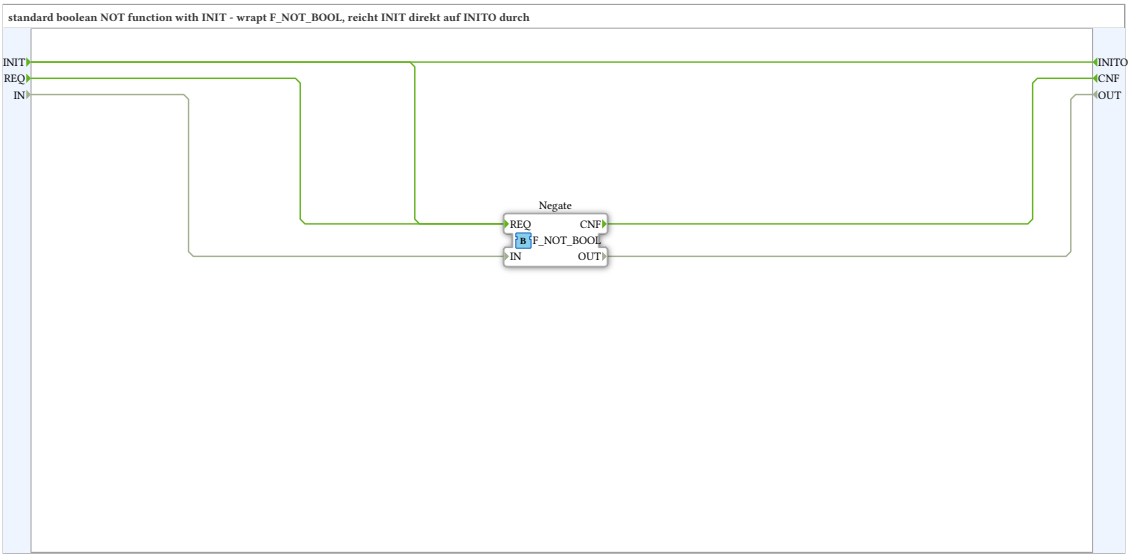
Innerhalb der `iec61131::bitwiseOperators`-Bibliothek existieren weitere logische Grundoperatoren wie `F_AND`, `F_OR` und `F_XOR`. Im Gegensatz zu diesen Bausteinen, die zwei oder mehr Eingänge verknüpfen, besitzt `F_NOT_BOOL` nur einen einzigen Dateneingang. Seine Funktion ist unverzichtbar, um logische Ausdrücke zu komplettieren und direkte Invertierungen durchzuführen. Die Implementierung als eigenständiger Baustein (anstatt als integrierter Operator in einem Ausdruck) erhöht die Übersichtlichkeit in grafischen Programmierumgebungen wie FBD. Siehe: F_NOT

Fazit

Der `F_NOT_BOOL` Funktionsblock ist ein essentieller, robuster und einfach zu verwendender Grundbaustein für die binäre Logik in IEC 61131-3-basierten Steuerungsprogrammen. Seine klare Schnittstelle und deterministische Funktionsweise machen ihn zur ersten Wahl für die Invertierung von booleschen Signalen in der Automatisierungstechnik.

F_NOT_BOOL_INIT





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein `F_NOT_BOOL_INIT` realisiert die logische Negation (NOT) eines booleschen Eingangssignals und erweitert die reine NOT-Funktion um einen Initialisierungsmechanismus. Er ist als standardisierter Baustein gemäß IEC 61131-3 für den Einsatz in speicherprogrammierbaren Steuerungen (SPS) konzipiert. Der FB ermöglicht es, bei der Initialisierung eine erste Negation auszulösen und den Ergebniswert am Ausgang bereitzustellen, bevor der normale zyklische Betrieb beginnt.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Mit Variablen	Beschreibung
`INIT`	EInit	`IN`	Initialisierungsanforderung – der FB wird einmalig initialisiert und der Eingang negiert.
`REQ`	Event	`IN`	Normale Verarbeitungsanforderung – die logische Negation wird auf den aktuellen Eingangswert angewendet.

Ereignis-Ausgänge

Ereignis	Typ	Mit Variablen	Beschreibung
`INITO`	EInit	`OUT`	Bestätigung der Initialisierung – wird nach erfolgreicher Initialisierung ausgegeben.
`CNF`	Event	`OUT`	Bestätigung der normalen Verarbeitung – wird nach jeder Ausführung auf `REQ` gesendet.

Daten-Eingänge

Variable	Typ	Beschreibung
`IN`	BOOL	Eingangswert, der negiert werden soll.

Daten-Ausgänge

Variable	Typ	Beschreibung
`OUT`	BOOL	Negierter Wert des Eingangs (`NOT IN`).

Adapter

Keine.

Funktionsweise

Der Baustein delegiert die eigentliche Negationsfunktion an den internen FB `F_NOT` (aus der Bibliothek `iec61131::bitwiseOperators`). Die Verarbeitung erfolgt ereignisgesteuert:

- **Bei `INIT`:** Der Eingangswert `IN` wird sofort über die Datenverbindung an den internen `F_NOT` übergeben. Gleichzeitig wird das Ereignis `REQ` an den internen FB gesendet, der die Negation berechnet und das Ergebnis an den Ausgang `OUT` weitergibt. Nach Abschluss dieser Operation wird das Ereignis `INITO` am Ausgang des Hauptbausteins ausgegeben. Dadurch wird der Ausgang `OUT` mit dem negierten Initialwert gesetzt.
- **Bei `REQ` (im laufenden Betrieb):** Der aktuelle Wert von `IN` wird an den internen `F_NOT` übergeben und negiert. Das Ergebnis wird auf `OUT` geschrieben, und anschließend wird das Ereignis `CNF` ausgelöst.

Somit führt sowohl `INIT` als auch `REQ` zur gleichen Kernfunktionalität (Negation), unterscheiden sich jedoch im ausgelösten Ereignis: `INIT` erzeugt `INITO`, `REQ` erzeugt `CNF`.

Technische Besonderheiten

- Der Baustein enthält keine eigene algorithmische Logik, sondern nutzt einen internen Unterbaustein (`F_NOT`). Dies fördert die Wiederverwendung und Konsistenz.

- Der Initialisierungsmechanismus erlaubt es, den Ausgang bereits vor dem ersten „normalen“ Aufruf mit dem negierten Eingang zu belegen – nützlich in SPS-Programmen, bei denen Startwerte definiert werden müssen.
- Die gleichzeitige Verwendung von `INIT` und `REQ` ermöglicht eine klare Trennung zwischen Startverhalten und zyklischem Betrieb.
- Durch die Mitnahme der Eingangsvariablen in den Ereignissen (WITH) wird sichergestellt, dass die Daten immer aktuell zur Verfügung stehen.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, da die gesamte Logik im internen Netzwerk abgebildet ist. Es liegt implizit folgendes Verhalten vor:

1. **Initialzustand:** Nach dem Start der SPS sind Ausgänge und interne Werte undefiniert.
2. **Nach `INIT`:** Der Ausgang `OUT` wird auf den negierten Wert von `IN` gesetzt; der FB ist betriebsbereit.
3. **Nach `REQ`:** Für jeden `REQ`-Impuls wird die Negation erneut ausgeführt und `OUT` aktualisiert.

Eine wiederholte Auslösung von `INIT` (z. B. nach einem Reset) führt erneut zur Initialisierung und setzt `OUT` auf den negierten aktuellen Eingangswert.

Anwendungsszenarien

- **Initialisierung eines Steuerausgangs:** Ein Ausgang soll in der Startphase den negierten Wert eines Sensorsignals annehmen, bevor das zyklische Programm startet.
- **Fehlerüberwachung:** Bei einer Startsequenz wird geprüft, ob ein Eingangssignal den negierten Zustand eines Freigabesignals aufweist.
- **Sicherheitslogik:** Verwendung in Sicherheitsfunktionen, bei denen ein initialer Negationsschritt zwingend erforderlich ist (z. B. NOT-Verknüpfung nach Systemhochlauf).

Vergleich mit ähnlichen Bausteinen

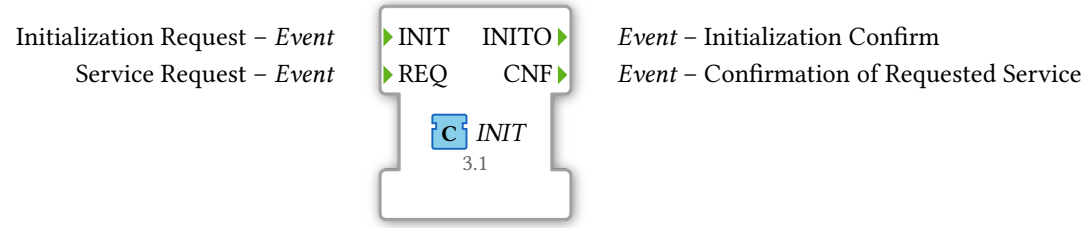
Baustein	Beschreibung	Unterschied zu `F_NOT_BOOL_INIT`
`F_NOT` (Standard)	Reine Negation ohne Initialisierungsereignis.	Fehlt das `INIT`/`INITO`-Paar; nach Systemstart ist der Ausgang nicht definiert.
`F_NOT_BOOL_INIT`	Wie `F_NOT`, aber mit explizitem Initialisierungsereignis.	Bietet definiertes Startverhalten und gibt bei Initialisierung ein separates Quittungsereignis aus.

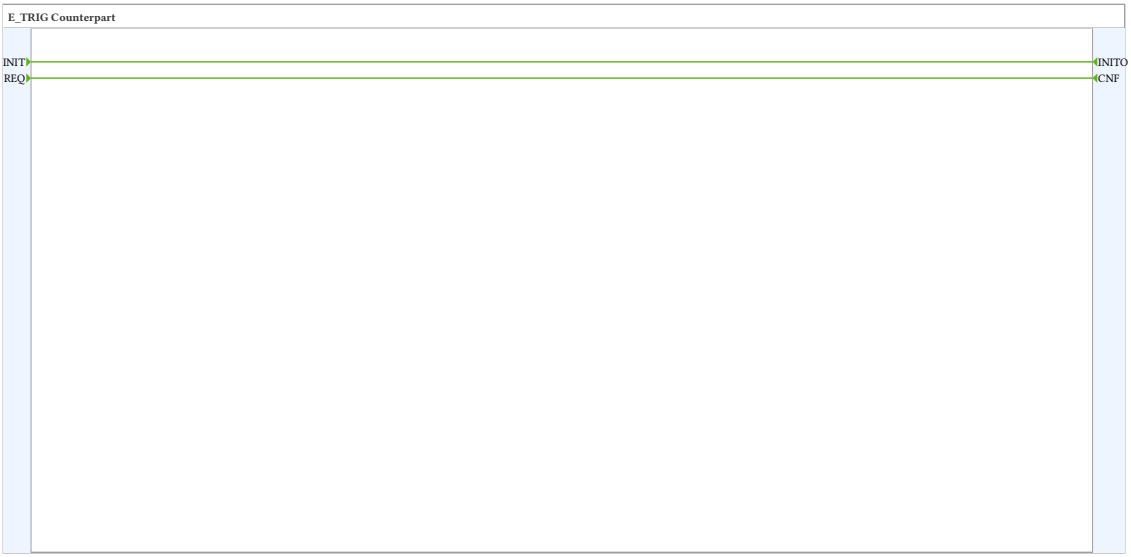
Der Standard `F_NOT` ist ausreichend, wenn das Startverhalten durch den übergeordneten Steuerungsablauf geregelt wird. Der initialisierbare Baustein ist dann vorteilhaft, wenn der FB selbst die Kontrolle über seine erste Ausführung besitzen soll.

Fazit

`F_NOT_BOOL_INIT` ist ein nützlicher Baustein, der die einfache NOT-Operation um eine Initialisierungsfazilität ergänzt. Er eignet sich besonders für SPS-Anwendungen, bei denen ein definierter Startwert des negierten Ausgangs erforderlich ist. Durch die klare Ereignissteuerung und die interne Nutzung des bewährten `F_NOT`-Bausteins bleibt die Implementierung robust und übersichtlich. Der FB stellt eine sinnvolle Erweiterung der IEC-61131-Bitoperatoren dar.

INIT





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **INIT** dient als direkte Durchleitung von Ereignissen. Er ist als Gegenstück zu einem `E_TRIG`-Baustein konzipiert und wird typischerweise in der Initialisierungsphase eines Systems eingesetzt. Der FB verfügt über zwei Ereigniseingänge und zwei korrespondierende Ereignisausgänge; Daten werden nicht verarbeitet.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Initialisierungsanforderung
REQ	Event	Dienstanforderung

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Initialisierungsbestätigung
CNF	Event	Bestätigung der Anforderung

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Keine.

Funktionsweise

Der Baustein arbeitet ereignisgesteuert und führt keine eigene Logik oder Zustandsänderung durch. Jedes ankommende Ereignis wird unverzögert und ohne Verarbeitung an den entsprechenden Ausgang weitergeleitet:

- Ein Ereignis am Eingang `INIT` erzeugt sofort ein Ereignis am Ausgang `INITO`.
- Ein Ereignis am Eingang `REQ` erzeugt sofort ein Ereignis am Ausgang `CNF`.

Dieses Verhalten wird durch eine schlichte Verbindung der Ereignispfade im inneren FBNetzwerk realisiert. Es findet keine Wertänderung oder Zustandsspeicherung statt.

Technische Besonderheiten

- **Direkte Durchleitung**: Der FB ist extrem schlank und benötigt keine internen Variablen oder zeitlichen Verzögerungen.
- **Ereignistypen**: Die Eingangs- und Ausgangsereignisse `INIT`/`INITO` sind als `EInit` typisiert, während `REQ`/`CNF` als Standard `Event` deklariert sind. Dies erlaubt eine klare semantische Trennung zwischen Initialisierungs- und Betriebsereignissen.
- **Lizenz**: Der Baustein steht unter der Eclipse Public License 2.0 (EPL-2.0).

Zustandsübersicht

Der Funktionsblock besitzt **keine internen Zustände**. Er ist ein reiner Durchleitungsbaustein, der zu jedem Zeitpunkt auf eingehende Ereignisse reagiert, ohne einen Zustand zu wechseln. Daher ist eine Zustandsmaschine nicht erforderlich.

Anwendungsszenarien

- **Platzhalter in Entwurfsphasen**: Wenn ein `E_TRIG`-Baustein später durch eine komplexere Logik ersetzt werden soll, kann `INIT` vorübergehend als einfache Durchleitung dienen.
- **Initialisierungskaskaden**: In Architekturen, die mehrere Initialisierungsschritte durchführen, können `INIT`-Bausteine verwendet werden, um Ereignisse ohne Verarbeitung an nachfolgende Blöcke weiterzugeben.
- **Prototyping und Tests**: Zum Überbrücken von nicht benötigten Initialisierungslogiken oder zum Debuggen von Ereignisflüssen.

Vergleich mit ähnlichen Bausteinen

- **E_TRIG (Eingangstrigger)**: `E_TRIG` erzeugt ein Ausgangsereignis nur bei einer steigenden oder fallenden Flanke eines angeschlossenen booleschen Eingangs. `INIT` hingegen reagiert direkt auf ein Ereignis und benötigt keinen Datenwert.
- **E_PERMIT (Freigabe)**: `E_PERMIT` lässt Ereignisse nur passieren, wenn ein boolescher Freigabeeingang gesetzt ist. `INIT` besitzt keine solche Bedingung.
- **E_DELAY (Verzögerung)**: `E_DELAY` hält Ereignisse für eine bestimmte Zeit zurück. `INIT` arbeitet ohne Verzögerung.

Fazit

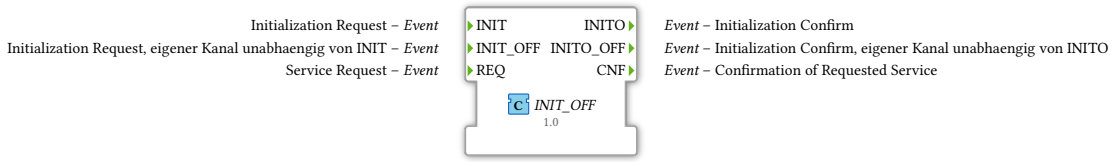
Der `INIT`-Funktionsblock ist ein minimalistischer, aber nützlicher Baustein zur direkten Ereignisdurchleitung. Aufgrund seiner Einfachheit eignet er sich besonders für Entwurfsphasen, Initialisierungsabläufe und als temporärer Platzhalter. Er erfüllt die grundlegende Anforderung, Initialisierungs- und Serviceereignisse unverändert weiterzugeben, und kann bei Bedarf durch funktional reichere Bausteine ersetzt werden.

INIT_OFF

• — ○ • — ○ • — ○ •

Einleitung

`INIT_OFF` ist das Gegenstück zu `E_TRIG`, erweitert um einen eigenständigen `INIT_OFF`/`INITO_OFF`-Pass-Through (Ereignistyp `EInitOff`). Der Baustein enthält keine Logik; jedes Eingangsereignis wird direkt an den zugehörigen Ausgang weitergereicht.



Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Initialisierungsanforderung
INIT_OFF	EInitOff	Initialisierungsanforderung, eigener Kanal unabhängig von `INIT`
REQ	Event	Dienstanforderung

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Initialisierungsbestätigung
INITO_OFF	EInitOff	Initialisierungsbestätigung, eigener Kanal unabhängig von `INITO`
CNF	Event	Bestätigung der angeforderten Dienstanforderung

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Keine.

Funktionsweise

Der Baustein besteht nur aus Ereignisverbindungen:

- `INIT` ⟶ `INITO`
- `REQ` ⟶ `CNF`
- `INIT_OFF` ⟶ `INITO_OFF`

Technische Besonderheiten

- Reine Durchleitung ohne Logik und ohne Daten
- Der `INIT_OFF`-Kanal ist unabhängig vom `INIT`-Kanal

Anwendungsszenarien

- Weiterreichen von Initialisierungs- und Abschaltereignissen (`EInitOff`) in einer Kette von Bausteinen

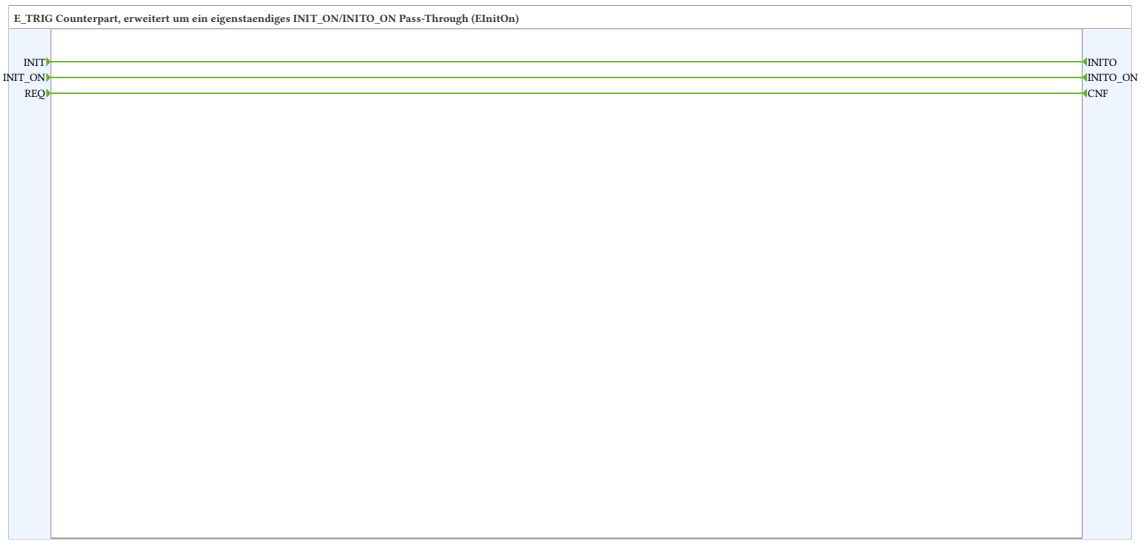
⚖ Vergleich mit ähnlichen Bausteinen

- INIT enthält nur `INIT`/`REQ` ohne den `INIT_OFF`-Kanal
- INIT_ON_OFF bietet zusätzlich einen `INIT_ON`/`INITO_ON`-Kanal

Fazit

`INIT_OFF` reicht `INIT`, `INIT_OFF` und `REQ` unverändert als `INITO`, `INITO_OFF` und `CNF` weiter.

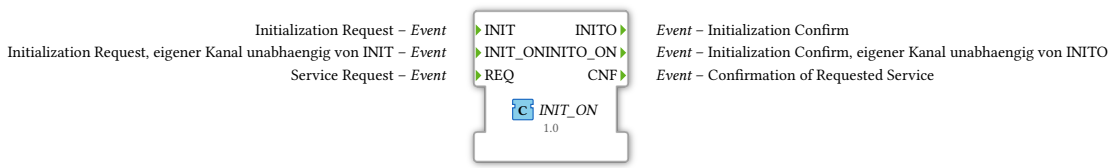
INIT_ON



• – ○ • – ○ • – ○ •

Einleitung

`INIT_ON` ist das Gegenstück zu `E_TRIG`, erweitert um einen eigenständigen `INIT_ON`/`INITO_ON`-Pass-Through (Ereignistyp `EInitOn`). Der Baustein enthält keine Logik; jedes Eingangsereignis wird direkt an den zugehörigen Ausgang weitergereicht.



Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Initialisierungsanforderung
INIT_ON	EInitOn	Initialisierungsanforderung, eigener Kanal unabhängig von `INIT`
REQ	Event	Dienstanforderung

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Initialisierungsbestätigung
INITO_ON	EInitOn	Initialisierungsbestätigung, eigener Kanal unabhängig von `INITO`
CNF	Event	Bestätigung der angeforderten Dienstanforderung

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Keine.

Funktionsweise

Der Baustein besteht nur aus Ereignisverbindungen:

- `INIT` ⟶ `INITO`
- `REQ` ⟶ `CNF`
- `INIT_ON` ⟶ `INITO_ON`

Technische Besonderheiten

- Reine Durchleitung ohne Logik und ohne Daten
- Der `INIT_ON`-Kanal ist unabhängig vom `INIT`-Kanal

Anwendungsszenarien

- Weiterreichen von Initialisierungsereignissen eines eigenen Kanals (`EInitOn`) in einer Kette von Bausteinen

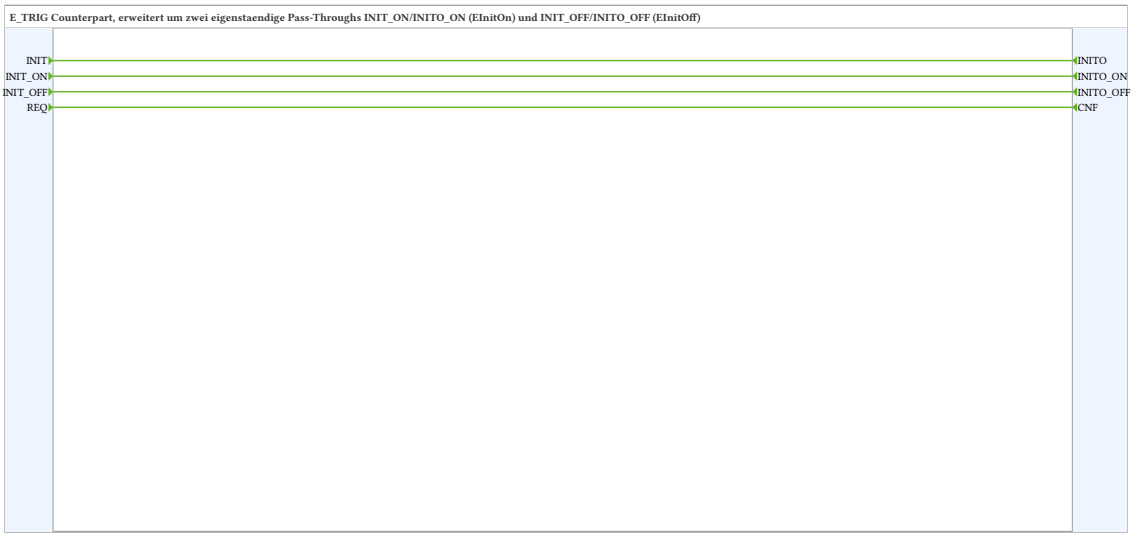
⚖️ Vergleich mit ähnlichen Bausteinen

- INIT enthält nur `INIT`/`REQ` ohne den `INIT_ON`-Kanal
- INIT_OFF bietet statt dessen einen `INIT_OFF`/`INITO_OFF`-Kanal (`EInitOff`), INIT_ON_OFF bietet beide Kanäle

Fazit

`INIT_ON` reicht `INIT`, `INIT_ON` und `REQ` unverändert als `INITO`, `INITO_ON` und `CNF` weiter.

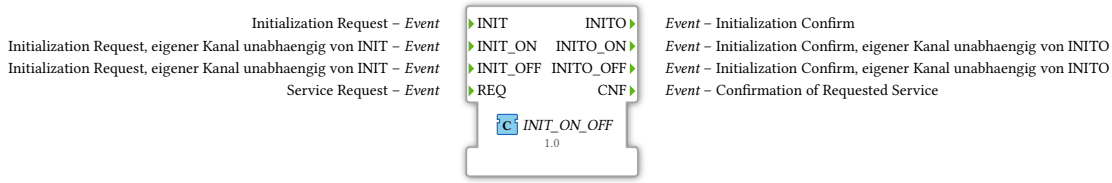
INIT_ON_OFF



• – ○ • – ○ • – ○ •

Einleitung

`INIT_ON_OFF` ist das Gegenstück zu `E_TRIG`, erweitert um zwei eigenständige Pass-Throughs: `INIT_ON`/`INITO_ON` (Ereignistyp `EInitOn`) und `INIT_OFF`/`INITO_OFF` (Ereignistyp `EInitOff`). Der Baustein enthält keine Logik; jedes Eingangsereignis wird direkt an den zugehörigen Ausgang weitergereicht.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierungsanforderung
- **INIT_ON** (`EInitOn`): Initialisierungsanforderung, eigener Kanal unabhängig von `INIT`
- **INIT_OFF** (`EInitOff`): Initialisierungsanforderung, eigener Kanal unabhängig von `INIT`
- **REQ**: Serviceanforderung

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung
- **INITO_ON** (`EInitOn`): Initialisierungsbestätigung, eigener Kanal unabhängig von `INITO`
- **INITO_OFF** (`EInitOff`): Initialisierungsbestätigung, eigener Kanal unabhängig von `INITO`
- **CNF**: Bestätigung des angeforderten Dienstes

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Reine Ereignisverbindungen:

- `INIT` ⟶ `INITO`

- `REQ` ⟶ `CNF`
- `INIT_ON` ⟶ `INITO_ON`
- `INIT_OFF` ⟶ `INITO_OFF`

Technische Besonderheiten

- Composite-Funktionsblock (`FBNetwork`) ohne Daten und ohne internen Zustand
- Die drei Initialisierungskanäle sind voneinander unabhängig, sodass ein- und ausschaltbezogene Initialisierungen getrennt verdrahtet werden können.
- Siehe auch `INIT`.

Anwendungsszenarien

- Getrennte Initialisierungs-Ereignisse für Ein- und Ausschaltpfad in einem Netzwerk

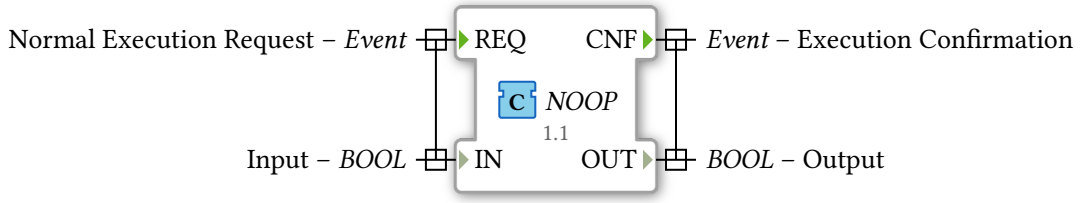
🛠️ Zugehörige Übungen

- Uebung_006g_AX

Fazit

`INIT_ON_OFF` stellt drei unabhängige Initialisierungs-Durchreichungen und eine `REQ`/`CNF`-Durchreichung bereit.

NOOP



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock NOOP („No Operation“) dient als reiner Durchlaufbaustein. Er kopiert den am Dateneingang `IN` anliegenden BOOL-Wert auf den Datenausgang `OUT`, sobald ein Ereignis am Eingang `REQ` eintritt. Anschließend wird der Ausgang `CNF` ausgelöst. Der FB führt keine logische oder arithmetische Operation aus, sondern leitet das Signal unverändert weiter.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar
REQ	Event	Normaler Ausführungsimpuls

Ereignis-Ausgänge

Ereignis	Typ	Kommentar
CNF	Event	Bestätigung der Ausführung

Daten-Eingänge

Name	Typ	Kommentar
IN	BOOL	Eingangssignal

Daten-Ausgänge

Name	Typ	Kommentar
OUT	BOOL	Ausgangssignal (identisch mit IN)

Adapter
Keine.

Funktionsweise
Intern besteht der Baustein aus einer Instanz von F_MOVE (aus der Bibliothek iec61131::selection::F_MOVE), die auf den Datentyp BOOL konfiguriert ist.

- Bei einem Ereignis auf REQ wird der Wert von IN an F_MOVE.IN übergeben und eine Ausführung angestoßen.
- Nach erfolgreicher Verarbeitung gibt F_MOVE.OUT den Wert aus, welcher auf OUT gelegt wird. Gleichzeitig wird F_MOVE.CNF ausgelöst, das den Ereignisausgang CNF aktiviert.

Somit entspricht das Verhalten genau einer Durchschaltung: OUT := IN.

Technische Besonderheiten
- Der FB verwendet intern einen vorgefertigten MOVE-Baustein, sodass keine eigene Zustandslogik notwendig ist.
- Die Datenpfade sind explizit auf BOOL ausgelegt. Eine Erweiterung auf andere Datentypen wäre durch Anpassung des F_MOVE-Attributes möglich, ist im vorliegenden Entwurf aber nicht vorgesehen.
- Es werden keine Seiteneffekte wie Verzögerungen oder Zustandsänderungen außerhalb der reinen Signalweitergabe realisiert.

Zustandsübersicht
Da der Baustein keine eigene Zustandsmaschine besitzt, kann der Ablauf als einfacher Zweischritt betrachtet werden:

1. **Warten** – auf ein Ereignis REQ.
2. **Weiterleiten** – sofortige Übernahme von IN nach OUT und Ausgabe von CNF.

Es gibt keine internen Speicherzustände oder Verzweigungen.

Anwendungsszenarien
- **Debugging und Test**: Einsatz als Platzhalter, um die Signalpfade in einem komplexen Netzwerk zu prüfen, ohne die Funktionalität zu beeinflussen.
- **Signalweiterleitung**: Wenn ein Ereignis-Signal unverändert durchgereicht werden muss, z. B. zur Synchronisation mehrerer Zweige.

- **Platzhalter für spätere Erweiterungen**: In einer frühen Entwicklungsphase kann NOOP an Stellen gesetzt werden, die später eine spezifische Verarbeitung erhalten sollen.
- **Kaskadierung von Ereignissen**: Durch die Ereignis-Kopplung können mehrere Empfänger nacheinander bedient werden.

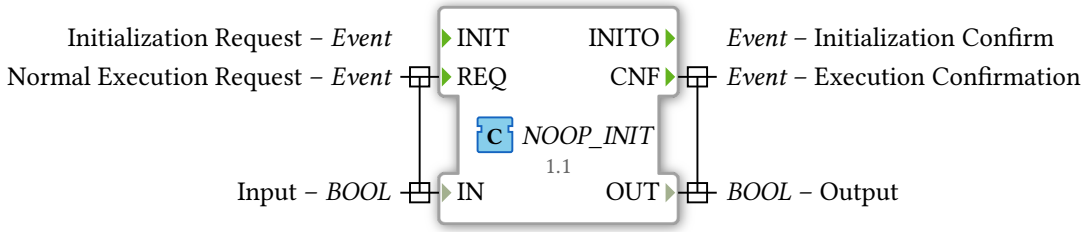
Vergleich mit ähnlichen Bausteinen

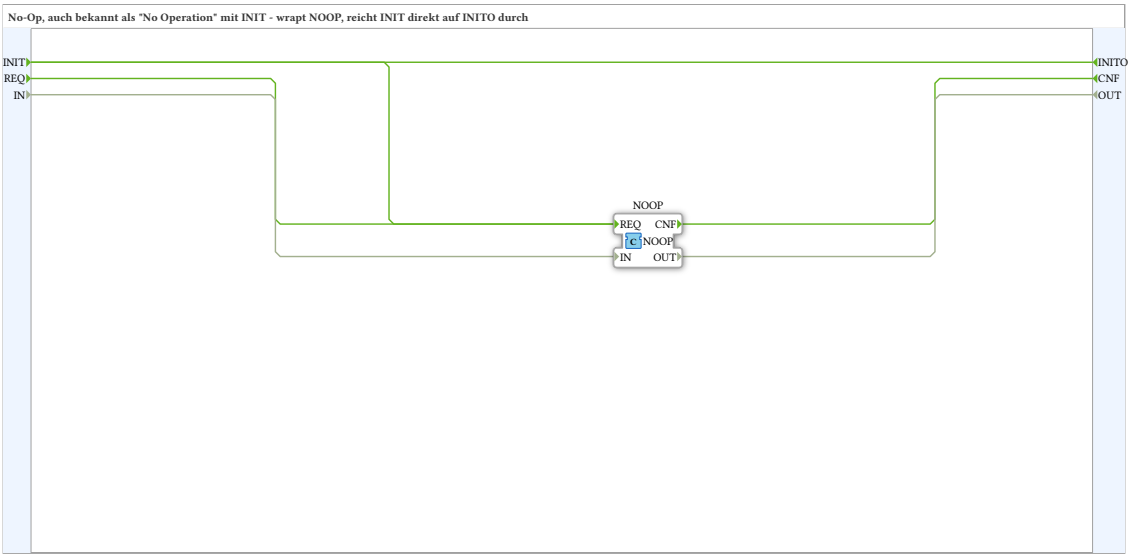
- **MOVE**: Ein MOVE-Baustein (wie der intern verwendete) führt ebenfalls eine Wertzuweisung aus. NOOP unterscheidet sich nur durch den Namen und die explizite Dokumentation als „keine Operation“. Beide sind funktional nahezu identisch, mit dem Unterschied, dass MOVE oft für beliebige Datentypen typisiert wird, während NOOP hier statisch auf BOOL festgelegt ist.
- **BISTABLE (RS / SR)**: Diese Bausteine ändern ihren Zustand basierend auf Eingängen. NOOP speichert keinen Zustand, sondern gibt nur den aktuellen Eingangswert aus.
- **CONSTANT**: Ein Konstanten-Baustein liefert immer einen festen Wert. NOOP hingegen reagiert auf wechselnde Eingänge und Ereignisse.

Fazit

Der NOOP-Funktionsblock ist ein einfacher, aber nützlicher Baustein, der ohne eigene Logik auskommt. Er kopiert den eingehenden BOOL-Wert bei jedem Ereignis auf den Ausgang und bestätigt die Ausführung. Aufgrund seiner Durchschaltungseigenschaften eignet er sich hervorragend für Testzwecke, als Platzhalter oder zur ereignisgesteuerten Signalweitergabe in IEC 61499-Netzwerken.

NOOP_INIT





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **NOOP_INIT** (No Operation with INIT) dient als einfacher Durchschalt- oder Platzhalterbaustein. Er ermöglicht die Initialisierung sowie die normale Datenweitergabe eines Booleschen Signals von einem Eingang zu einem Ausgang. Die Besonderheit liegt in der zusätzlichen Behandlung des INIT-Ereignisses, das sowohl eine Bestätigung (INITO) als auch eine Datenweitergabe auslöst.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Initialisierungsanforderung
REQ	Event	Normale Ausführungsanforderung (verbunden mit IN)

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Initialisierungsbestätigung
CNF	Event	Ausführungsbestätigung (verbunden mit OUT)

Daten-Eingänge

Name	Typ	Kommentar
IN	BOOL	Eingangssignal

Daten-Ausgänge

Name	Typ	Kommentar
OUT	BOOL	Ausgangssignal (entspricht IN nach einem Durchlauf)

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Baustein verarbeitet eingehende Ereignisse wie folgt:

1. **INIT-Ereignis**:
   - Es wird sofort ein **INITO**-Ereignis ausgegeben.
   - Gleichzeitig wird intern der `F_MOVE`-Baustein angestoßen, der den Wert von `IN` nach `OUT` kopiert. Nach Abschluss dieses Kopiervorgangs wird das **CNF**-Ereignis ausgelöst.
2. **REQ-Ereignis**:
   - Es wird der gleiche `F_MOVE`-Baustein aktiviert, der `IN` nach `OUT` überträgt. Nach Fertigstellung wird **CNF** ausgegeben.

Im Ergebnis wird sowohl bei INIT als auch bei REQ eine Datenweitergabe durchgeführt und jeweils ein Bestätigungsereignis erzeugt. Der Datenausgang `OUT` entspricht stets dem zuletzt übernommenen Wert des Eingangs `IN`.

Technische Besonderheiten

- Der Baustein nutzt intern eine Instanz des Standard-FBs `iec61131::selection::F_MOVE` für die Datenkopie.
- Die INIT-Behandlung ist zweigeteilt: Sie quittiert sofort mit `INITO` und führt parallel die Datenweitergabe aus. Dies kann für Initialisierungssequenzen nützlich sein, bei denen das System nach der ersten Übergabe eines Startwerts arbeiten soll.
- Es sind keine internen Zustände oder Timings vorhanden; die Reaktion erfolgt ereignisgesteuert.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, das Verhalten ist rein ereignisgesteuert:

- Im Ruhezustand (kein Ereignis) warten die Eingänge.
- Bei **INIT** oder **REQ** wird die Datenkopie gestartet; nach deren Abschluss wird `CNF` gesendet. Bei INIT wird zusätzlich sofort `INITO` emittiert.

Anwendungsszenarien

- **Platzhalter** in der Entwicklungsphase: Anstatt eines komplexen Bausteins wird NOOP_INIT eingefügt, um Datenflüsse zu testen.
- **Debugging**: Das explizite INIT-Ereignis kann verwendet werden, um einen initialen Wert zu setzen und gleichzeitig eine Bestätigung zu erhalten.
- **Signalkopplung**: Wenn ein Signal unverändert weitergeleitet werden muss, aber die INIT-Schnittstelle eines nachfolgenden Bausteins bedient werden soll.
- **Test von Initialisierungssequenzen**: Simulation einer Startroutine, die einen Wert setzt und quittiert.

Vergleich mit ähnlichen Bausteinen

- **NOOP (ohne INIT)**: Einfacher Durchschaltbaustein ohne Initialisierungsereignis. NOOP_INIT erweitert dies um die INIT/INITO-Paarung.
- **MOVE (direkt)**: Ein reiner Datenkopierer ohne Ereignissteuerung; NOOP_INIT bietet eine ereignisgesteuerte Version mit Bestätigungen.
- **E_CYCLE oder E_PERMIT**: Diese Bausteine steuern Ereignisse, übertragen aber keine Daten. NOOP_INIT kombiniert Datenweitergabe mit Ereignisweiterleitung.

Fazit

Der Funktionsblock **NOOP_INIT** ist ein einfacher, aber flexibler Hilfsbaustein für den ereignisgesteuerten Datentransport. Durch die Kombination von INIT- und REQ-Ereignissen eignet er sich besonders für Initialisierungsabläufe und Testumgebungen. Seine interne Struktur macht ihn leicht verständlich und gut in bestehende 4diac-Netzwerke integrierbar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

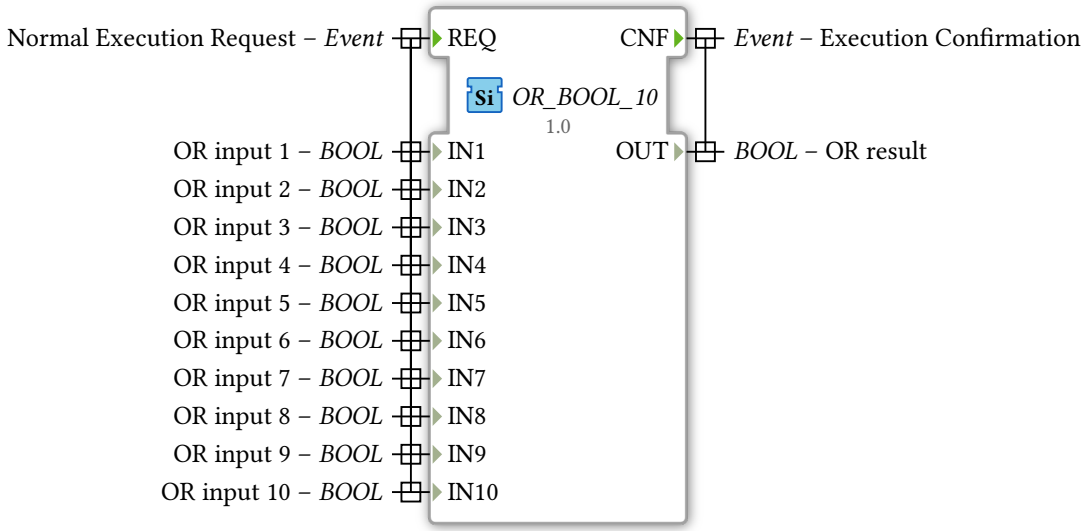
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

OR_BOOL_10

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `OR_BOOL_10` ist ein generischer Baustein zur Berechnung der logischen ODER-Verknüpfung. Er führt die ODER-Operation über bis zu zehn separate boolesche Eingangswerte aus und gibt das Ergebnis auf einem einzigen Ausgang aus. Dieser Baustein ist gemäß der Norm IEC 61131-3 klassifiziert und dient als Standardfunktion für bitweise Operationen in Steuerungsanwendungen.



Schnittstellenstruktur

Der Funktionsblock folgt dem standardmäßigen E/A- und Ereignismodell der 4diac-IDE.

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Dieses Ereignis löst die Ausführung des Funktionsblocks aus. Bei seinem Eintreffen werden alle zehn Daten-Eingänge (`IN1` bis `IN10`) ausgelesen und die ODER-Operation berechnet.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Dieses Ereignis signalisiert den Abschluss der Berechnung. Es wird zusammen mit dem berechneten Ergebnis am Datenausgang `OUT` ausgegeben.

Daten-Eingänge

- **IN1 (OR input 1):** Boolescher Eingang 1.
- **IN2 (OR input 2):** Boolescher Eingang 2.
- **IN3 (OR input 3):** Boolescher Eingang 3.
- **IN4 (OR input 4):** Boolescher Eingang 4.
- **IN5 (OR input 5):** Boolescher Eingang 5.
- **IN6 (OR input 6):** Boolescher Eingang 6.
- **IN7 (OR input 7):** Boolescher Eingang 7.
- **IN8 (OR input 8):** Boolescher Eingang 8.
- **IN9 (OR input 9):** Boolescher Eingang 9.
- **IN10 (OR input 10):** Boolescher Eingang 10.

Daten-Ausgänge

- **OUT (OR result):** Boolesches Ergebnis der ODER-Verknüpfung aller zehn Eingänge. Der Ausgang ist `TRUE` (1), wenn mindestens einer der Eingänge `TRUE` ist. Er ist nur `FALSE` (0), wenn alle zehn Eingänge `FALSE` sind.

Adapter

Dieser Funktionsblock verfügt über keine Adapter-Schnittstellen.

Funktionsweise

Die Funktionsweise ist deterministisch und ereignisgesteuert:

1. Das Eintreffen des Ereignisses `REQ` startet die Ausführung.
2. Die aktuellen Werte aller zehn booleschen Eingänge (`IN1` bis `IN10`) werden ausgelesen.
3. Es wird die logische ODER-Verknüpfung `OUT = IN1 OR IN2 OR IN3 OR ... OR IN10` berechnet.
4. Das Ergebnis wird am Datenausgang `OUT` verfügbar gemacht.
5. Das Ereignis `CNF` wird ausgelöst, um den erfolgreichen Abschluss der Operation an nachfolgende Blöcke zu signalisieren.

Technische Besonderheiten

- **Generischer Baustein:** Der Block ist als generischer Funktionsblock (`GEN_OR`) implementiert, was eine flexible Wiederverwendung in verschiedenen Kontexten ermöglicht.
- **Feste Eingangsanzahl:** Im Gegensatz zu Bausteinen mit variabler Eingangsanzahl besitzt `OR_BOOL_10` exakt zehn fest definierte Eingänge. Nicht benötigte Eingänge sollten auf `FALSE` gesetzt werden.
- **Ereignisgesteuert:** Die Berechnung erfolgt nur bei Vorliegen eines `REQ`-Ereignisses, was eine energie- und rechenzeiteffiziente Abarbeitung im Steuerungssystem ermöglicht.

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand (speicherlos). Das Ausgangssignal `OUT` ist eine reine Funktion der aktuellen Eingangswerte zum Zeitpunkt der `REQ`-Anforderung. Es gibt keine Verzögerung, Hysterese oder Speicherung vorheriger Zustände.

Anwendungsszenarien

- **Überwachungslogik:** Zusammenfassen mehrerer Fehler- oder Warnsignale (z.B. von verschiedenen Sensoren) zu einem gemeinsamen Störmeldesignal.
- **Freigabelogik:** Erzeugen einer Freigabe für einen Prozessschritt, wenn mindestens eine von mehreren Voraussetzungen erfüllt ist.
- **Verknüpfung von Tastergruppen:** In einer Bedienstation, bei der ein Prozess durch Drücken mindestens eines von mehreren „Start“-Tastern initiiert werden kann.

- **Redundante Sensorauswertung:** Auswerten mehrerer redundanter Sensoren, wobei das Signal eines beliebigen Sensors als gültig anerkannt wird.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`OR_BOOL_2` / `OR_BOOL_4`:** Diese Blöcke bieten die gleiche ODER-Funktionalität, jedoch für eine geringere Anzahl von Eingängen (2 bzw. 4). `OR_BOOL_10` ist für Anwendungen mit einer höheren Anzahl zu verknüpfender Signale vorgesehen. Siehe: OR_10
- **`AND_BOOL_10`:** Führt die logische UND-Verknüpfung durch. Das Ergebnis ist nur `TRUE`, wenn *alle* Eingänge `TRUE` sind, während es bei `OR_BOOL_10` ausreicht, wenn *mindestens ein* Eingang `TRUE` ist.
- **`XOR_BOOL_10`:** Führt die exklusive ODER-Verknüpfung durch. Das Ergebnis ist `TRUE`, wenn eine ungerade Anzahl an Eingängen `TRUE` ist, was sich grundlegend von der inklusiven ODER-Logik unterscheidet.
- **Bausteine mit variabler Eingangsanzahl:** Einige Bibliotheken bieten ODER-Bausteine an, bei denen die Anzahl der Eingänge konfigurierbar ist. `OR_BOOL_10` bietet hingegen eine feste, explizite Schnittstelle.

Fazit

Der `OR_BOOL_10` ist ein robuster und einfach zu verwendender Grundbaustein für die logische Signalverknüpfung in IEC 61131-3-basierten Steuerungsapplikationen. Seine Stärke liegt in der klaren Schnittstelle mit zehn Eingängen und der zuverlässigen, ereignisgesteuerten Berechnung der inklusiven ODER-Funktion. Für Anwendungen, die das Zusammenfassen mehrerer boolescher Quellen zu einem gemeinsamen Signal erfordern, stellt er eine optimale Lösung dar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

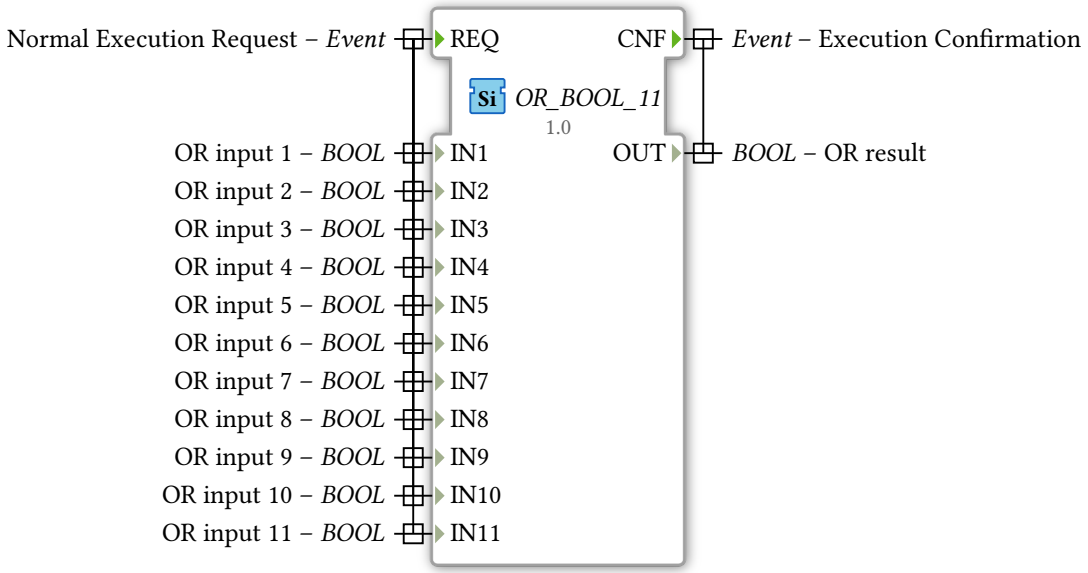
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

OR_BOOL_11

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `OR_BOOL_11` ist ein generischer Baustein zur Berechnung der logischen ODER-Verknüpfung (Klassifikation `standard boolean function`, IEC 61131-3). Er verknüpft 11 separate BOOL-Eingänge (`IN1` bis `IN11`) und liefert das Ergebnis am Ausgang `OUT`. Er ergänzt die Standard-OR_BOOL-Familie (bisher bis `OR_BOOL_16`) um die Variante mit 11 Eingängen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Löst die Berechnung aus. Das Ereignis ist mit allen 11 Daten-Eingängen (`IN1` bis `IN11`) verknüpft.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Signalisiert den Abschluss der Berechnung. Das Ereignis ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN1 (OR input 1):** Boolescher Eingang 1.
- **IN2 (OR input 2):** Boolescher Eingang 2.
- **IN3 (OR input 3):** Boolescher Eingang 3.
- **IN4 (OR input 4):** Boolescher Eingang 4.
- **IN5 (OR input 5):** Boolescher Eingang 5.
- **IN6 (OR input 6):** Boolescher Eingang 6.
- **IN7 (OR input 7):** Boolescher Eingang 7.
- **IN8 (OR input 8):** Boolescher Eingang 8.
- **IN9 (OR input 9):** Boolescher Eingang 9.
- **IN10 (OR input 10):** Boolescher Eingang 10.
- **IN11 (OR input 11):** Boolescher Eingang 11.

Daten-Ausgänge

- **OUT (OR result):** Das Ergebnis der ODER-Verknüpfung aller 11 Eingänge. Der Ausgang ist TRUE, wenn mindestens ein Eingang TRUE ist, und nur FALSE, wenn alle Eingänge `IN1` bis `IN11` FALSE sind.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei jedem Eintreffen des Ereignisses `REQ` werden alle 11 Eingänge gelesen und die Operation `OUT = IN1 OR IN2 OR ... OR IN11` berechnet. Anschließend wird das Ergebnis an `OUT` bereitgestellt und `CNF` ausgelöst.

Technische Besonderheiten

- **Generischer Baustein:** Der Typ trägt das Attribut `GenericClassName = 'GEN_OR_BOOL'`; diese Instanz ist fest auf 11 Eingänge ausgelegt.
- **Package:** `iec61131::booleanOperators`.
- **Zustandslos:** Der Baustein ist speicherlos: `OUT` hängt ausschließlich von den Werten der Eingänge zum Zeitpunkt von `REQ` ab.

Anwendungsszenarien

- **Sammelstörung:** Mehrere Fehler- oder Warnsignale zu einem gemeinsamen Signal zusammenfassen.
- **Mehrfach-Auslösung:** Eine Aktion soll starten, sobald mindestens eine von vielen Bedingungen oder Tasten aktiv ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- `AND_BOOL_11`: Führt die UND-Verknüpfung durch: Das Ergebnis ist nur TRUE, wenn alle Eingänge TRUE sind, während bei `OR_BOOL_11` mindestens ein TRUE-Eingang genügt.
- `OR_BOOL_16`: Gleichartiger Baustein der Familie mit 16 Eingängen.

Fazit

`OR_BOOL_11` ist ein einfacher, ereignisgesteuerter Baustein für die ODER-Verknüpfung von 11 booleschen Signalen mit fester Schnittstelle.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

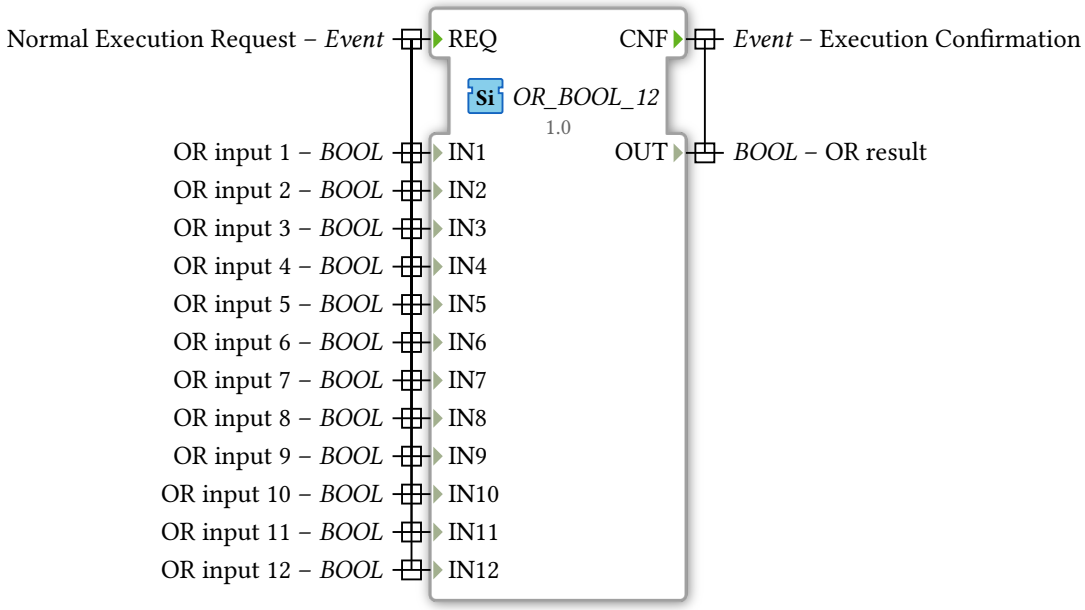
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

OR_BOOL_12

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `OR_BOOL_12` ist ein generischer Baustein zur Berechnung der logischen ODER-Verknüpfung (Klassifikation `standard boolean function`, IEC 61131-3). Er verknüpft 12 separate BOOL-Eingänge (`IN1` bis `IN12`) und liefert das Ergebnis am Ausgang `OUT`. Er ergänzt die Standard-OR_BOOL-Familie (bisher bis `OR_BOOL_16`) um die Variante mit 12 Eingängen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Löst die Berechnung aus. Das Ereignis ist mit allen 12 Daten-Eingängen (`IN1` bis `IN12`) verknüpft.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Signalisiert den Abschluss der Berechnung. Das Ereignis ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN1 (OR input 1):** Boolescher Eingang 1.
- **IN2 (OR input 2):** Boolescher Eingang 2.
- **IN3 (OR input 3):** Boolescher Eingang 3.
- **IN4 (OR input 4):** Boolescher Eingang 4.
- **IN5 (OR input 5):** Boolescher Eingang 5.
- **IN6 (OR input 6):** Boolescher Eingang 6.
- **IN7 (OR input 7):** Boolescher Eingang 7.
- **IN8 (OR input 8):** Boolescher Eingang 8.
- **IN9 (OR input 9):** Boolescher Eingang 9.
- **IN10 (OR input 10):** Boolescher Eingang 10.
- **IN11 (OR input 11):** Boolescher Eingang 11.
- **IN12 (OR input 12):** Boolescher Eingang 12.

Daten-Ausgänge

- **OUT (OR result):** Das Ergebnis der ODER-Verknüpfung aller 12 Eingänge. Der Ausgang ist TRUE, wenn mindestens ein Eingang TRUE ist, und nur FALSE, wenn alle Eingänge `IN1` bis `IN12` FALSE sind.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei jedem Eintreffen des Ereignisses `REQ` werden alle 12 Eingänge gelesen und die Operation `OUT = IN1 OR IN2 OR ... OR IN12` berechnet. Anschließend wird das Ergebnis an `OUT` bereitgestellt und `CNF` ausgelöst.

Technische Besonderheiten

- **Generischer Baustein:** Der Typ trägt das Attribut `GenericClassName = 'GEN_OR_BOOL'`; diese Instanz ist fest auf 12 Eingänge ausgelegt.
- **Package:** `iec61131::booleanOperators`.
- **Zustandslos:** Der Baustein ist speicherlos: `OUT` hängt ausschließlich von den Werten der Eingänge zum Zeitpunkt von `REQ` ab.

Anwendungsszenarien

- **Sammelstörung:** Mehrere Fehler- oder Warnsignale zu einem gemeinsamen Signal zusammenfassen.
- **Mehrfach-Auslösung:** Eine Aktion soll starten, sobald mindestens eine von vielen Bedingungen oder Tasten aktiv ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- `AND_BOOL_12`: Führt die UND-Verknüpfung durch: Das Ergebnis ist nur TRUE, wenn alle Eingänge TRUE sind, während bei `OR_BOOL_12` mindestens ein TRUE-Eingang genügt.
- `OR_BOOL_16`: Gleichartiger Baustein der Familie mit 16 Eingängen.

Fazit

`OR_BOOL_12` ist ein einfacher, ereignisgesteuerter Baustein für die ODER-Verknüpfung von 12 booleschen Signalen mit fester Schnittstelle.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

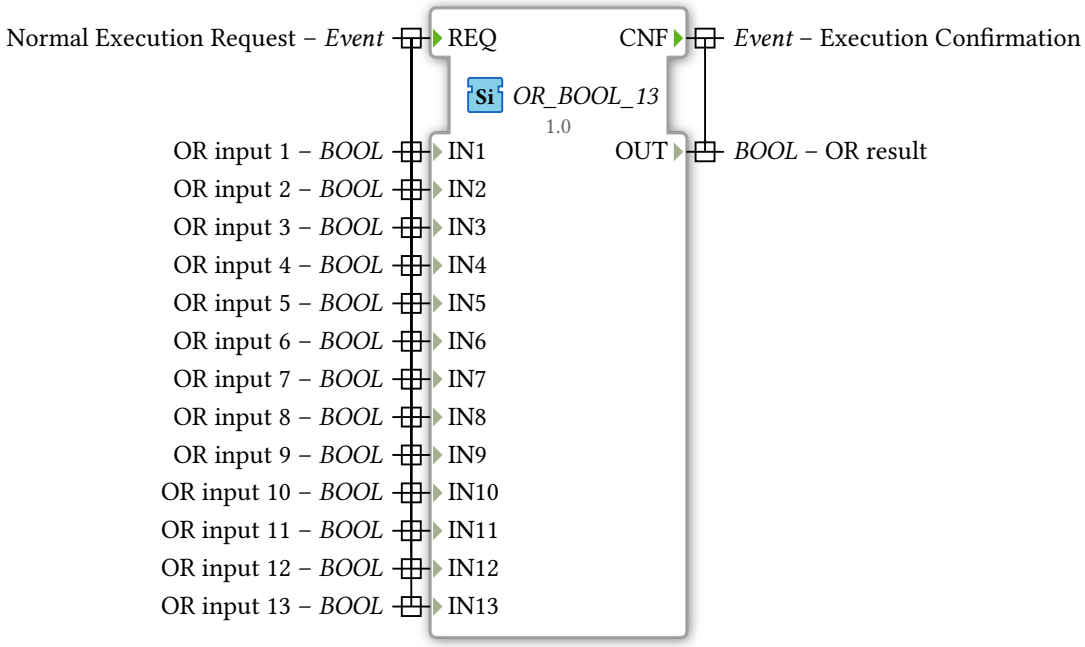
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

OR_BOOL_13

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `OR_BOOL_13` ist ein generischer Baustein zur Berechnung der logischen ODER-Verknüpfung (Klassifikation `standard boolean function`, IEC 61131-3). Er verknüpft 13 separate BOOL-Eingänge (`IN1` bis `IN13`) und liefert das Ergebnis am Ausgang `OUT`. Er ergänzt die Standard-OR_BOOL-Familie (bisher bis `OR_BOOL_16`) um die Variante mit 13 Eingängen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Löst die Berechnung aus. Das Ereignis ist mit allen 13 Daten-Eingängen (`IN1` bis `IN13`) verknüpft.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Signalisiert den Abschluss der Berechnung. Das Ereignis ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN1 (OR input 1):** Boolescher Eingang 1.
- **IN2 (OR input 2):** Boolescher Eingang 2.
- **IN3 (OR input 3):** Boolescher Eingang 3.
- **IN4 (OR input 4):** Boolescher Eingang 4.
- **IN5 (OR input 5):** Boolescher Eingang 5.
- **IN6 (OR input 6):** Boolescher Eingang 6.
- **IN7 (OR input 7):** Boolescher Eingang 7.
- **IN8 (OR input 8):** Boolescher Eingang 8.
- **IN9 (OR input 9):** Boolescher Eingang 9.
- **IN10 (OR input 10):** Boolescher Eingang 10.
- **IN11 (OR input 11):** Boolescher Eingang 11.
- **IN12 (OR input 12):** Boolescher Eingang 12.
- **IN13 (OR input 13):** Boolescher Eingang 13.

Daten-Ausgänge

- **OUT (OR result):** Das Ergebnis der ODER-Verknüpfung aller 13 Eingänge. Der Ausgang ist TRUE, wenn mindestens ein Eingang TRUE ist, und nur FALSE, wenn alle Eingänge `IN1` bis `IN13` FALSE sind.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei jedem Eintreffen des Ereignisses `REQ` werden alle 13 Eingänge gelesen und die Operation `OUT = IN1 OR IN2 OR ... OR IN13` berechnet. Anschließend wird das Ergebnis an `OUT` bereitgestellt und `CNF` ausgelöst.

Technische Besonderheiten

- **Generischer Baustein:** Der Typ trägt das Attribut `GenericClassName = 'GEN_OR_BOOL'`; diese Instanz ist fest auf 13 Eingänge ausgelegt.
- **Package:** `iec61131::booleanOperators`.
- **Zustandslos:** Der Baustein ist speicherlos: `OUT` hängt ausschließlich von den Werten der Eingänge zum Zeitpunkt von `REQ` ab.

Anwendungsszenarien

- **Sammelstörung:** Mehrere Fehler- oder Warnsignale zu einem gemeinsamen Signal zusammenfassen.
- **Mehrfach-Auslösung:** Eine Aktion soll starten, sobald mindestens eine von vielen Bedingungen oder Tasten aktiv ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- `AND_BOOL_13`: Führt die UND-Verknüpfung durch: Das Ergebnis ist nur TRUE, wenn alle Eingänge TRUE sind, während bei `OR_BOOL_13` mindestens ein TRUE-Eingang genügt.
- `OR_BOOL_16`: Gleichartiger Baustein der Familie mit 16 Eingängen.

Fazit

`OR_BOOL_13` ist ein einfacher, ereignisgesteuerter Baustein für die ODER-Verknüpfung von 13 booleschen Signalen mit fester Schnittstelle.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

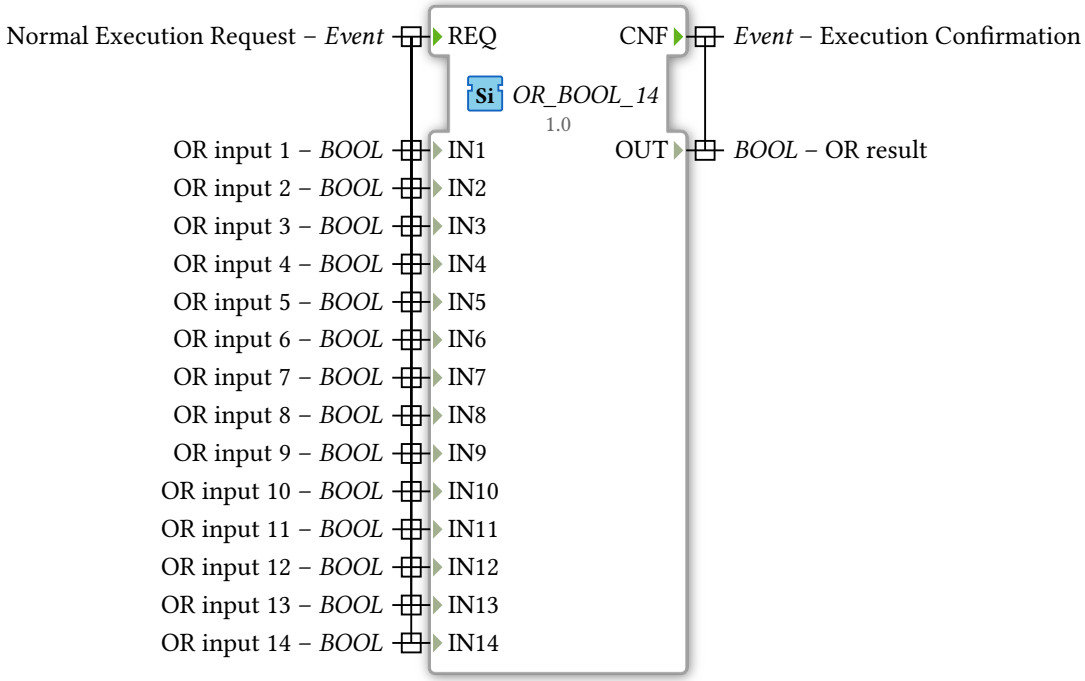
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

OR_BOOL_14

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `OR_BOOL_14` ist ein generischer Baustein zur Berechnung der logischen ODER-Verknüpfung (Klassifikation `standard boolean function`, IEC 61131-3). Er verknüpft 14 separate BOOL-Eingänge (`IN1` bis `IN14`) und liefert das Ergebnis am Ausgang `OUT`. Er ergänzt die Standard-OR_BOOL-Familie (bisher bis `OR_BOOL_16`) um die Variante mit 14 Eingängen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Löst die Berechnung aus. Das Ereignis ist mit allen 14 Daten-Eingängen (`IN1` bis `IN14`) verknüpft.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Signalisiert den Abschluss der Berechnung. Das Ereignis ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN1 (OR input 1):** Boolescher Eingang 1.
- **IN2 (OR input 2):** Boolescher Eingang 2.
- **IN3 (OR input 3):** Boolescher Eingang 3.
- **IN4 (OR input 4):** Boolescher Eingang 4.
- **IN5 (OR input 5):** Boolescher Eingang 5.
- **IN6 (OR input 6):** Boolescher Eingang 6.
- **IN7 (OR input 7):** Boolescher Eingang 7.
- **IN8 (OR input 8):** Boolescher Eingang 8.
- **IN9 (OR input 9):** Boolescher Eingang 9.
- **IN10 (OR input 10):** Boolescher Eingang 10.
- **IN11 (OR input 11):** Boolescher Eingang 11.
- **IN12 (OR input 12):** Boolescher Eingang 12.
- **IN13 (OR input 13):** Boolescher Eingang 13.
- **IN14 (OR input 14):** Boolescher Eingang 14.

Daten-Ausgänge

- **OUT (OR result):** Das Ergebnis der ODER-Verknüpfung aller 14 Eingänge. Der Ausgang ist TRUE, wenn mindestens ein Eingang TRUE ist, und nur FALSE, wenn alle Eingänge `IN1` bis `IN14` FALSE sind.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei jedem Eintreffen des Ereignisses `REQ` werden alle 14 Eingänge gelesen und die Operation `OUT = IN1 OR IN2 OR ... OR IN14` berechnet. Anschließend wird das Ergebnis an `OUT` bereitgestellt und `CNF` ausgelöst.

Technische Besonderheiten

- **Generischer Baustein:** Der Typ trägt das Attribut `GenericClassName = 'GEN_OR_BOOL'`; diese Instanz ist fest auf 14 Eingänge ausgelegt.
- **Package:** `iec61131::booleanOperators`.
- **Zustandslos:** Der Baustein ist speicherlos: `OUT` hängt ausschließlich von den Werten der Eingänge zum Zeitpunkt von `REQ` ab.

Anwendungsszenarien

- **Sammelstörung:** Mehrere Fehler- oder Warnsignale zu einem gemeinsamen Signal zusammenfassen.
- **Mehrfach-Auslösung:** Eine Aktion soll starten, sobald mindestens eine von vielen Bedingungen oder Tasten aktiv ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- `AND_BOOL_14`: Führt die UND-Verknüpfung durch: Das Ergebnis ist nur TRUE, wenn alle Eingänge TRUE sind, während bei `OR_BOOL_14` mindestens ein TRUE-Eingang genügt.
- `OR_BOOL_16`: Gleichartiger Baustein der Familie mit 16 Eingängen.

Fazit

`OR_BOOL_14` ist ein einfacher, ereignisgesteuerter Baustein für die ODER-Verknüpfung von 14 booleschen Signalen mit fester Schnittstelle.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

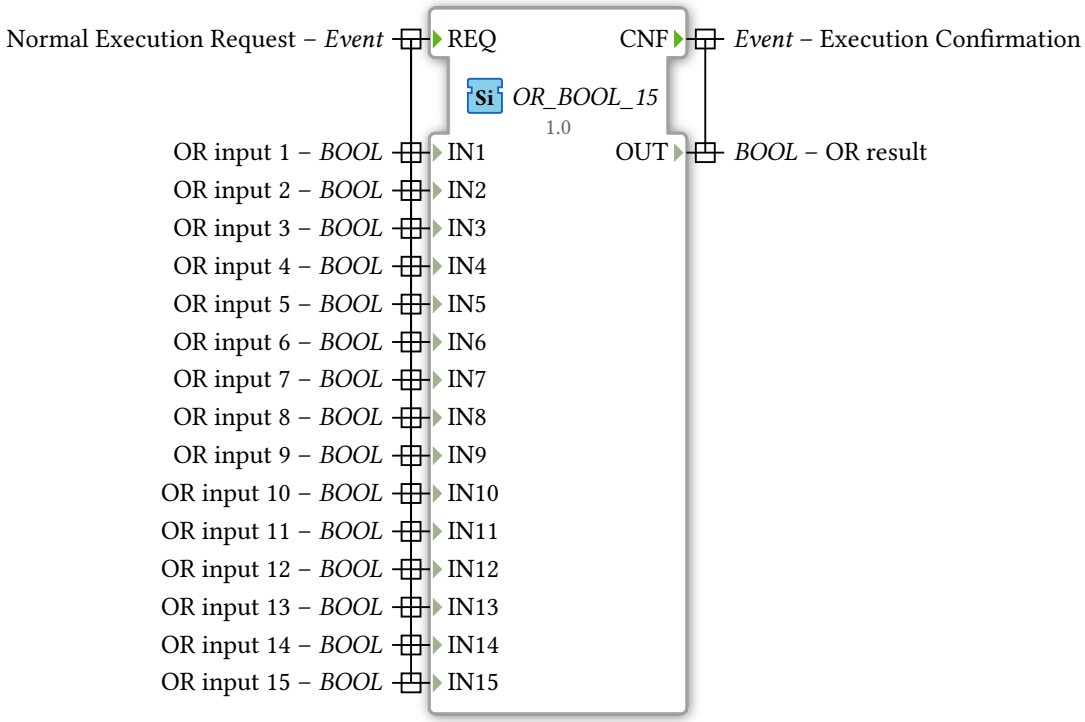
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

OR_BOOL_15

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `OR_BOOL_15` ist ein generischer Baustein zur Berechnung der logischen ODER-Verknüpfung (Klassifikation `standard boolean function`, IEC 61131-3). Er verknüpft 15 separate BOOL-Eingänge (`IN1` bis `IN15`) und liefert das Ergebnis am Ausgang `OUT`. Er ergänzt die Standard-OR_BOOL-Familie (bisher bis `OR_BOOL_16`) um die Variante mit 15 Eingängen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Löst die Berechnung aus. Das Ereignis ist mit allen 15 Daten-Eingängen (`IN1` bis `IN15`) verknüpft.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Signalisiert den Abschluss der Berechnung. Das Ereignis ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN1 (OR input 1):** Boolescher Eingang 1.
- **IN2 (OR input 2):** Boolescher Eingang 2.
- **IN3 (OR input 3):** Boolescher Eingang 3.
- **IN4 (OR input 4):** Boolescher Eingang 4.
- **IN5 (OR input 5):** Boolescher Eingang 5.
- **IN6 (OR input 6):** Boolescher Eingang 6.
- **IN7 (OR input 7):** Boolescher Eingang 7.
- **IN8 (OR input 8):** Boolescher Eingang 8.
- **IN9 (OR input 9):** Boolescher Eingang 9.
- **IN10 (OR input 10):** Boolescher Eingang 10.
- **IN11 (OR input 11):** Boolescher Eingang 11.

- **IN12 (OR input 12):** Boolescher Eingang 12.
- **IN13 (OR input 13):** Boolescher Eingang 13.
- **IN14 (OR input 14):** Boolescher Eingang 14.
- **IN15 (OR input 15):** Boolescher Eingang 15.

Daten-Ausgänge

- **OUT (OR result):** Das Ergebnis der ODER-Verknüpfung aller 15 Eingänge. Der Ausgang ist TRUE, wenn mindestens ein Eingang TRUE ist, und nur FALSE, wenn alle Eingänge `IN1` bis `IN15` FALSE sind.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei jedem Eintreffen des Ereignisses `REQ` werden alle 15 Eingänge gelesen und die Operation `OUT = IN1 OR IN2 OR ... OR IN15` berechnet. Anschließend wird das Ergebnis an `OUT` bereitgestellt und `CNF` ausgelöst.

Technische Besonderheiten

- **Generischer Baustein:** Der Typ trägt das Attribut `GenericClassName = 'GEN_OR_BOOL'`; diese Instanz ist fest auf 15 Eingänge ausgelegt.
- **Package:** `iec61131::booleanOperators`.
- **Zustandslos:** Der Baustein ist speicherlos: `OUT` hängt ausschließlich von den Werten der Eingänge zum Zeitpunkt von `REQ` ab.

Anwendungsszenarien

- **Sammelstörung:** Mehrere Fehler- oder Warnsignale zu einem gemeinsamen Signal zusammenfassen.
- **Mehrfach-Auslösung:** Eine Aktion soll starten, sobald mindestens eine von vielen Bedingungen oder Tasten aktiv ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- `AND_BOOL_15`: Führt die UND-Verknüpfung durch: Das Ergebnis ist nur TRUE, wenn alle Eingänge TRUE sind, während bei `OR_BOOL_15` mindestens ein TRUE-Eingang genügt.
- `OR_BOOL_16`: Gleichartiger Baustein der Familie mit 16 Eingängen.

Fazit

`OR_BOOL_15` ist ein einfacher, ereignisgesteuerter Baustein für die ODER-Verknüpfung von 15 booleschen Signalen mit fester Schnittstelle.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

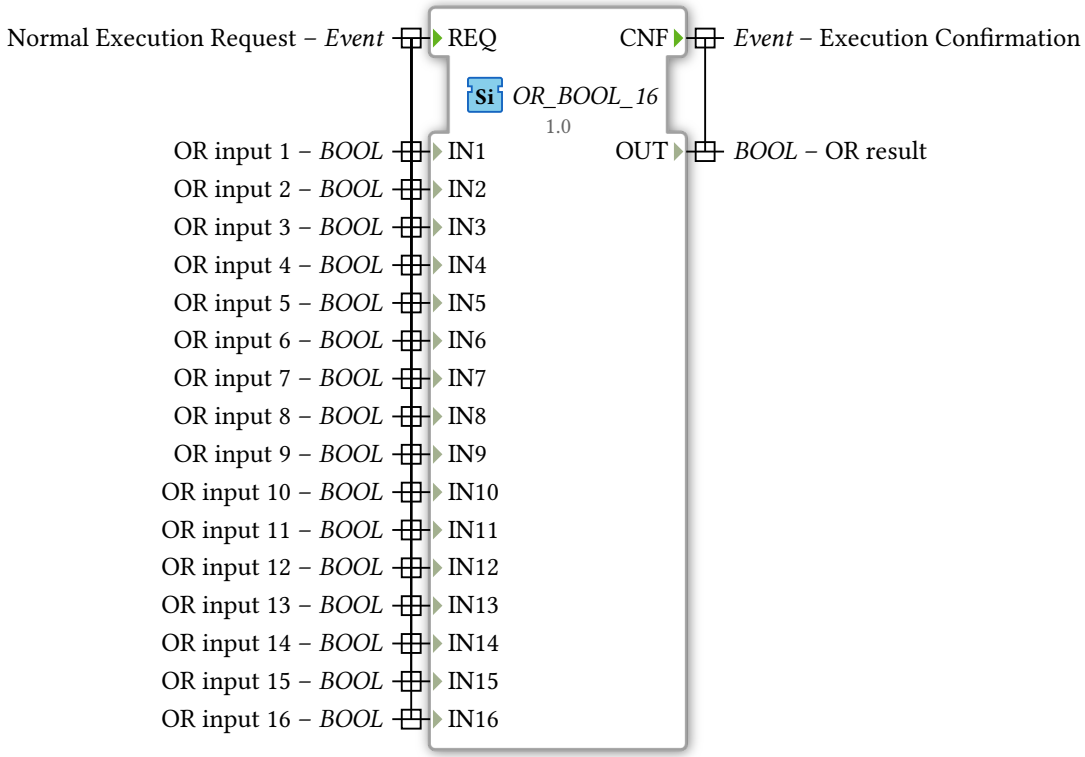
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

OR_BOOL_16

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `OR_BOOL_16` ist ein Standard-Baustein zur Berechnung der logischen ODER-Verknüpfung. Er führt die ODER-Operation über 16 separate boolesche Eingänge durch und stellt das Ergebnis an einem einzigen Ausgang zur Verfügung. Dieser Baustein ist Teil der IEC 61131-3-konformen Bibliothek für bitweise Operationen und eignet sich für Anwendungen, in denen eine logische Verknüpfung mehrerer Signale erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Dieses Ereignis löst die Berechnung der ODER-Funktion aus. Es ist mit allen 16 Dateneingängen (`IN1` bis `IN16`) verknüpft.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Dieses Ereignis signalisiert den Abschluss der Berechnung. Es wird zusammen mit dem aktualisierten Datenausgang `OUT` ausgegeben.

Daten-Eingänge

Der Baustein verfügt über 16 identische boolesche Dateneingänge:

- **IN1** bis **IN16 (BOOL):** Die zu verknüpfenden Eingangssignale. Jeder Eingang repräsentiert einen Operanden für die ODER-Operation.

Daten-Ausgänge

- **OUT (BOOL):** Das Ergebnis der logischen ODER-Verknüpfung aller 16 Eingänge. Der Ausgang ist `TRUE` (1), wenn mindestens einer der Eingänge `TRUE` ist. Er ist nur `FALSE` (0), wenn alle 16 Eingänge `FALSE` sind.

Adapter

Dieser Funktionsblock verwendet keine Adapter-Schnittstellen.

Funktionsweise

Die Funktionsweise ist deterministisch und ereignisgesteuert:

1. Bei Eintreffen eines Ereignisses am Eingang `REQ` wird der Baustein aktiviert.
2. Der Baustein liest die aktuellen Werte aller 16 Eingänge `IN1` bis `IN16`.
3. Es wird die logische ODER-Operation über alle gelesenen Werte berechnet: `OUT = IN1 OR IN2 OR ... OR IN16`.
4. Das Ergebnis wird am Datenausgang `OUT` gesetzt.
5. Unmittelbar nach der Berechnung wird das Bestätigungsereignis `CNF` zusammen mit dem neuen Wert von `OUT` ausgegeben.

Technische Besonderheiten

- **Generischer Baustein:** Der Block ist als generischer Baustein (`GEN_OR`) gekennzeichnet, was auf eine mögliche Wiederverwendung der zugrundeliegenden Logik für andere ODER-Bausteine mit unterschiedlicher Eingangsanzahl hindeutet.
- **Feste Eingangsanzahl:** Im Gegensatz zu Bausteinen mit variabler Eingangsanzahl besitzt `OR_BOOL_16` exakt 16 Eingänge. Dies bietet eine klare und fest definierte Schnittstelle.
- **Ereignisgesteuerte Ausführung:** Die Berechnung erfolgt nur bei einem eingehenden `REQ`-Ereignis, was eine energieeffiziente und bedarfsgerechte Abarbeitung im Echtzeitsystem ermöglicht.

Zustandsübersicht

Der Baustein besitzt keinen internen Zustand im Sinne eines Speichers. Er verhält sich rein kombinatorisch in Bezug auf die Eingangsdaten. Sein „Zustand“ ist ausschließlich durch das Vorhandensein oder Fehlen eines ausstehenden Berechnungsauftrags (`REQ`) definiert. Nach der Bearbeitung und dem Senden von `CNF` kehrt er in einen passiven Wartezustand zurück.

Anwendungsszenarien

- **Überwachungslogik:** Zusammenfassen mehrerer Fehler- oder Warnsignale zu einem gemeinsamen Störsignal. (z.B. „Maschine anhalten, wenn Sensor A ODER Sensor B ODER ... ODER Sensor P eine Störung meldet“).
- **Freigabelogik:** Prüfen, ob mindestens eine von mehreren Bedingungen für den Start eines Prozesses erfüllt ist.
- **Verknüpfung von Tastergruppen:** In einer Bedienstation, bei der eine Aktion durch das Betätigen mindestens eines von mehreren (bis zu 16) Tastern ausgelöst werden kann.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`OR_BOOL_2`, `OR_BOOL_8`:** Dies sind baugleiche ODER-Bausteine mit einer geringeren Anzahl an Eingängen (2 bzw. 8). `OR_BOOL_16` erweitert diese Reihe für Anwendungen mit höherer Eingangsanzahl. Siehe: OR_16
- **`AND_BOOL_16`:** Führt die logische UND-Verknüpfung durch. Das Ergebnis ist nur `TRUE`, wenn *alle* Eingänge `TRUE` sind, während es bei `OR_BOOL_16` reicht, wenn *mindestens ein* Eingang `TRUE` ist.
- **Bausteine mit variabler Eingangsanzahl:** Einige Bibliotheken bieten ODER-Bausteine an, bei denen die Anzahl der Eingänge konfigurierbar ist. `OR_BOOL_16` bietet hingegen eine feste, optimierte Schnittstelle für genau 16 Signale.

Fazit

Der `OR_BOOL_16` ist ein robuster und einfach zu verwendender Standardbaustein für die logische ODER-Verknüpfung einer größeren Gruppe von booleschen Signalen. Seine feste Struktur mit 16 Eingängen macht ihn vorhersagbar und einfach in der Anwendung, insbesondere wenn die benötigte Anzahl an Signalen bekannt und konstant ist. Die ereignisgesteuerte Ausführung integriert ihn nahtlos in die Datenfluss-orientierte Architektur von 4diac. Er ist die erste Wahl, wenn bis zu 16 digitale Signale auf ein gemeinsames ODER-Ergebnis reduziert werden sollen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

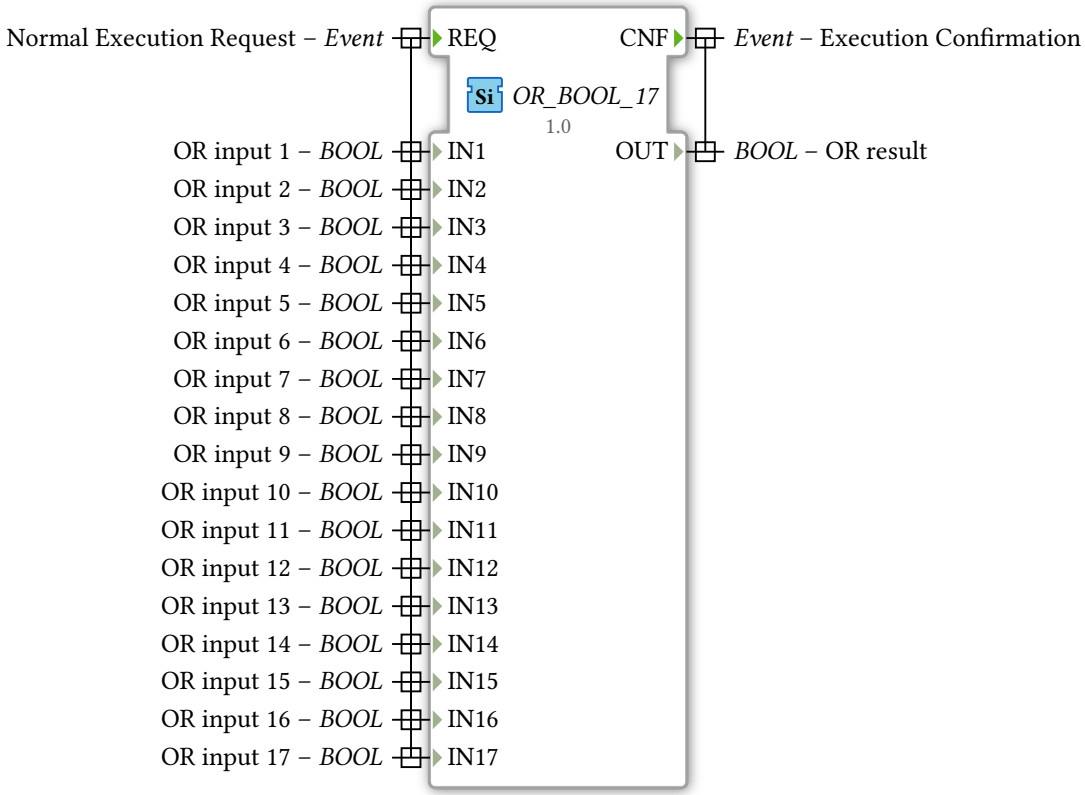
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

OR_BOOL_17

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `OR_BOOL_17` ist ein generischer Baustein zur Berechnung der logischen ODER-Verknüpfung (Klassifikation `standard boolean function`, IEC 61131-3). Er verknüpft 17 separate BOOL-Eingänge (`IN1` bis `IN17`) und liefert das Ergebnis am Ausgang `OUT`. Er ergänzt die Standard-OR_BOOL-Familie (bisher bis `OR_BOOL_16`) um die Variante mit 17 Eingängen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Löst die Berechnung aus. Das Ereignis ist mit allen 17 Daten-Eingängen (`IN1` bis `IN17`) verknüpft.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Signalisiert den Abschluss der Berechnung. Das Ereignis ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN1 (OR input 1):** Boolescher Eingang 1.
- **IN2 (OR input 2):** Boolescher Eingang 2.
- **IN3 (OR input 3):** Boolescher Eingang 3.
- **IN4 (OR input 4):** Boolescher Eingang 4.
- **IN5 (OR input 5):** Boolescher Eingang 5.
- **IN6 (OR input 6):** Boolescher Eingang 6.
- **IN7 (OR input 7):** Boolescher Eingang 7.
- **IN8 (OR input 8):** Boolescher Eingang 8.
- **IN9 (OR input 9):** Boolescher Eingang 9.
- **IN10 (OR input 10):** Boolescher Eingang 10.
- **IN11 (OR input 11):** Boolescher Eingang 11.
- **IN12 (OR input 12):** Boolescher Eingang 12.
- **IN13 (OR input 13):** Boolescher Eingang 13.
- **IN14 (OR input 14):** Boolescher Eingang 14.
- **IN15 (OR input 15):** Boolescher Eingang 15.
- **IN16 (OR input 16):** Boolescher Eingang 16.
- **IN17 (OR input 17):** Boolescher Eingang 17.

Daten-Ausgänge

- **OUT (OR result):** Das Ergebnis der ODER-Verknüpfung aller 17 Eingänge. Der Ausgang ist TRUE, wenn mindestens ein Eingang TRUE ist, und nur FALSE, wenn alle Eingänge `IN1` bis `IN17` FALSE sind.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei jedem Eintreffen des Ereignisses `REQ` werden alle 17 Eingänge gelesen und die Operation `OUT = IN1 OR IN2 OR ... OR IN17` berechnet. Anschließend wird das Ergebnis an `OUT` bereitgestellt und `CNF` ausgelöst.

Technische Besonderheiten

- **Generischer Baustein:** Der Typ trägt das Attribut `GenericClassName = 'GEN_OR_BOOL'`; diese Instanz ist fest auf 17 Eingänge ausgelegt.
- **Package:** `iec61131::booleanOperators`.
- **Zustandslos:** Der Baustein ist speicherlos: `OUT` hängt ausschließlich von den Werten der Eingänge zum Zeitpunkt von `REQ` ab.

Anwendungsszenarien

- **Sammelstörung:** Mehrere Fehler- oder Warnsignale zu einem gemeinsamen Signal zusammenfassen.
- **Mehrfach-Auslösung:** Eine Aktion soll starten, sobald mindestens eine von vielen Bedingungen oder Tasten aktiv ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- `AND_BOOL_17`: Führt die UND-Verknüpfung durch: Das Ergebnis ist nur TRUE, wenn alle Eingänge TRUE sind, während bei `OR_BOOL_17` mindestens ein TRUE-Eingang genügt.
- `OR_BOOL_16`: Gleichartiger Baustein der Familie mit 16 Eingängen.

Fazit

`OR_BOOL_17` ist ein einfacher, ereignisgesteuerter Baustein für die ODER-Verknüpfung von 17 booleschen Signalen mit fester Schnittstelle.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

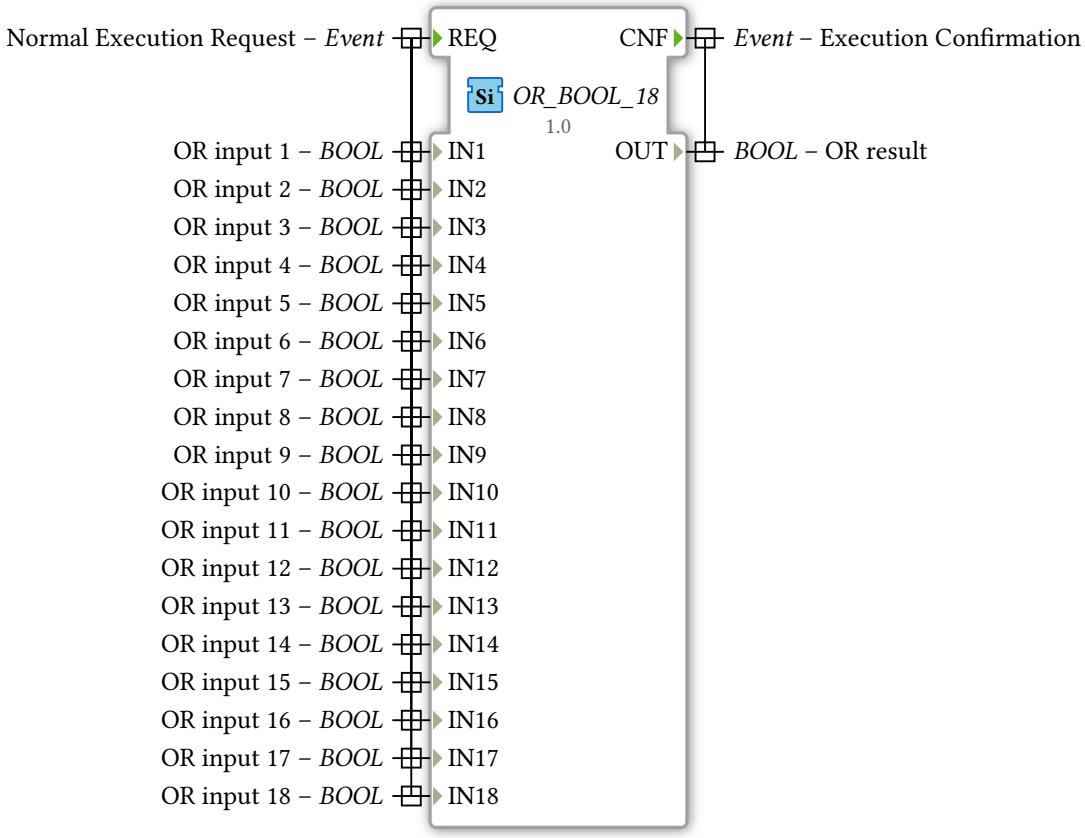
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

OR_BOOL_18

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `OR_BOOL_18` ist ein generischer Baustein zur Berechnung der logischen ODER-Verknüpfung (Klassifikation `standard boolean function`, IEC 61131-3). Er verknüpft 18 separate BOOL-Eingänge (`IN1` bis `IN18`) und liefert das Ergebnis am Ausgang `OUT`. Er ergänzt die Standard-OR_BOOL-Familie (bisher bis `OR_BOOL_16`) um die Variante mit 18 Eingängen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Löst die Berechnung aus. Das Ereignis ist mit allen 18 Daten-Eingängen (`IN1` bis `IN18`) verknüpft.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Signalisiert den Abschluss der Berechnung. Das Ereignis ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN1 (OR input 1):** Boolescher Eingang 1.
- **IN2 (OR input 2):** Boolescher Eingang 2.
- **IN3 (OR input 3):** Boolescher Eingang 3.
- **IN4 (OR input 4):** Boolescher Eingang 4.
- **IN5 (OR input 5):** Boolescher Eingang 5.
- **IN6 (OR input 6):** Boolescher Eingang 6.
- **IN7 (OR input 7):** Boolescher Eingang 7.
- **IN8 (OR input 8):** Boolescher Eingang 8.
- **IN9 (OR input 9):** Boolescher Eingang 9.
- **IN10 (OR input 10):** Boolescher Eingang 10.
- **IN11 (OR input 11):** Boolescher Eingang 11.
- **IN12 (OR input 12):** Boolescher Eingang 12.
- **IN13 (OR input 13):** Boolescher Eingang 13.
- **IN14 (OR input 14):** Boolescher Eingang 14.
- **IN15 (OR input 15):** Boolescher Eingang 15.
- **IN16 (OR input 16):** Boolescher Eingang 16.
- **IN17 (OR input 17):** Boolescher Eingang 17.
- **IN18 (OR input 18):** Boolescher Eingang 18.

Daten-Ausgänge

- **OUT (OR result):** Das Ergebnis der ODER-Verknüpfung aller 18 Eingänge. Der Ausgang ist TRUE, wenn mindestens ein Eingang TRUE ist, und nur FALSE, wenn alle Eingänge `IN1` bis `IN18` FALSE sind.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei jedem Eintreffen des Ereignisses `REQ` werden alle 18 Eingänge gelesen und die Operation `OUT = IN1 OR IN2 OR ... OR IN18` berechnet. Anschließend wird das Ergebnis an `OUT` bereitgestellt und `CNF` ausgelöst.

Technische Besonderheiten

- **Generischer Baustein:** Der Typ trägt das Attribut `GenericClassName = 'GEN_OR_BOOL'`; diese Instanz ist fest auf 18 Eingänge ausgelegt.
- **Package:** `iec61131::booleanOperators`.
- **Zustandslos:** Der Baustein ist speicherlos: `OUT` hängt ausschließlich von den Werten der Eingänge zum Zeitpunkt von `REQ` ab.

Anwendungsszenarien

- **Sammelstörung:** Mehrere Fehler- oder Warnsignale zu einem gemeinsamen Signal zusammenfassen.
- **Mehrfach-Auslösung:** Eine Aktion soll starten, sobald mindestens eine von vielen Bedingungen oder Tasten aktiv ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- `AND_BOOL_18`: Führt die UND-Verknüpfung durch: Das Ergebnis ist nur TRUE, wenn alle Eingänge TRUE sind, während bei `OR_BOOL_18` mindestens ein TRUE-Eingang genügt.
- `OR_BOOL_16`: Gleichartiger Baustein der Familie mit 16 Eingängen.

Fazit

`OR_BOOL_18` ist ein einfacher, ereignisgesteuerter Baustein für die ODER-Verknüpfung von 18 booleschen Signalen mit fester Schnittstelle.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

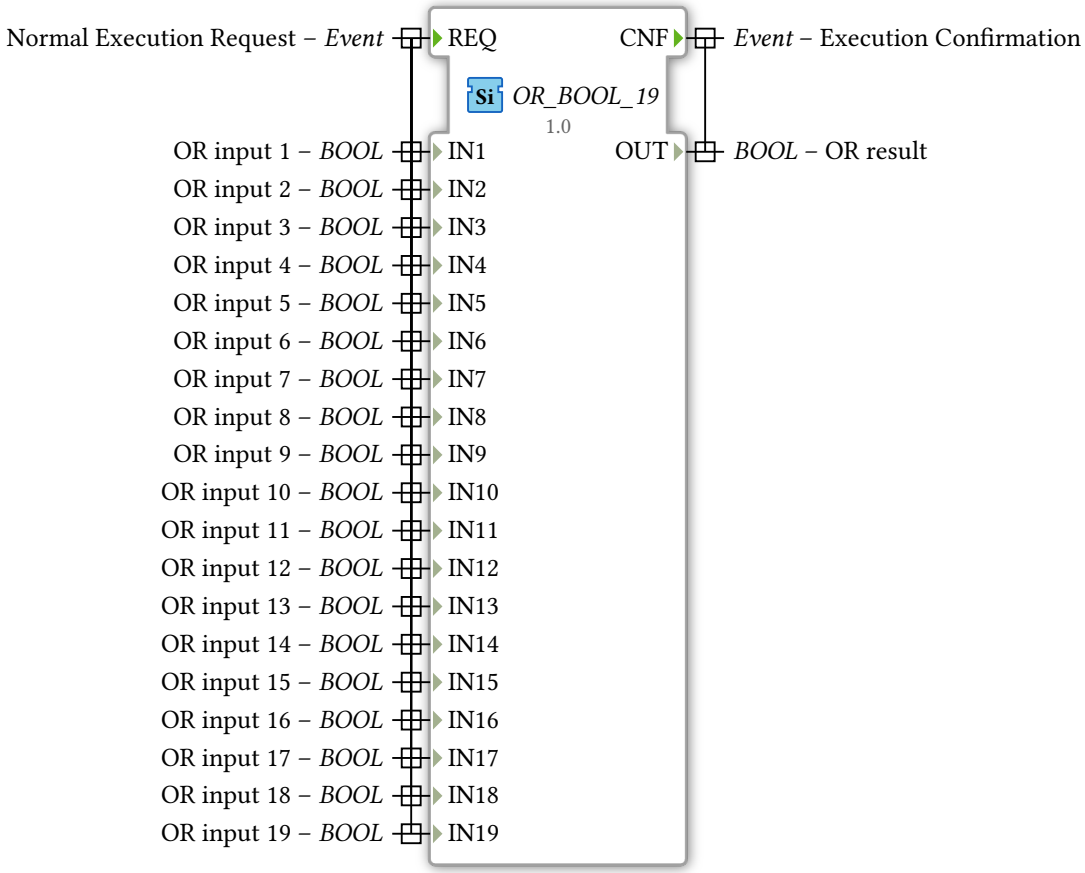
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

OR_BOOL_19

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `OR_BOOL_19` ist ein generischer Baustein zur Berechnung der logischen ODER-Verknüpfung (Klassifikation `standard boolean function`, IEC 61131-3). Er verknüpft 19 separate BOOL-Eingänge (`IN1` bis `IN19`) und liefert das Ergebnis am Ausgang `OUT`. Er ergänzt die Standard-OR_BOOL-Familie (bisher bis `OR_BOOL_16`) um die Variante mit 19 Eingängen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Löst die Berechnung aus. Das Ereignis ist mit allen 19 Daten-Eingängen (`IN1` bis `IN19`) verknüpft.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Signalisiert den Abschluss der Berechnung. Das Ereignis ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN1 (OR input 1):** Boolescher Eingang 1.
- **IN2 (OR input 2):** Boolescher Eingang 2.
- **IN3 (OR input 3):** Boolescher Eingang 3.
- **IN4 (OR input 4):** Boolescher Eingang 4.
- **IN5 (OR input 5):** Boolescher Eingang 5.
- **IN6 (OR input 6):** Boolescher Eingang 6.
- **IN7 (OR input 7):** Boolescher Eingang 7.
- **IN8 (OR input 8):** Boolescher Eingang 8.
- **IN9 (OR input 9):** Boolescher Eingang 9.
- **IN10 (OR input 10):** Boolescher Eingang 10.
- **IN11 (OR input 11):** Boolescher Eingang 11.
- **IN12 (OR input 12):** Boolescher Eingang 12.
- **IN13 (OR input 13):** Boolescher Eingang 13.
- **IN14 (OR input 14):** Boolescher Eingang 14.
- **IN15 (OR input 15):** Boolescher Eingang 15.
- **IN16 (OR input 16):** Boolescher Eingang 16.
- **IN17 (OR input 17):** Boolescher Eingang 17.
- **IN18 (OR input 18):** Boolescher Eingang 18.

- **IN19 (OR input 19):** Boolescher Eingang 19.

Daten-Ausgänge

- **OUT (OR result):** Das Ergebnis der ODER-Verknüpfung aller 19 Eingänge. Der Ausgang ist TRUE, wenn mindestens ein Eingang TRUE ist, und nur FALSE, wenn alle Eingänge `IN1` bis `IN19` FALSE sind.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei jedem Eintreffen des Ereignisses `REQ` werden alle 19 Eingänge gelesen und die Operation `OUT = IN1 OR IN2 OR ... OR IN19` berechnet. Anschließend wird das Ergebnis an `OUT` bereitgestellt und `CNF` ausgelöst.

Technische Besonderheiten

- **Generischer Baustein:** Der Typ trägt das Attribut `GenericClassName = 'GEN_OR_BOOL'`; diese Instanz ist fest auf 19 Eingänge ausgelegt.
- **Package:** `iec61131::booleanOperators`.
- **Zustandslos:** Der Baustein ist speicherlos: `OUT` hängt ausschließlich von den Werten der Eingänge zum Zeitpunkt von `REQ` ab.

Anwendungsszenarien

- **Sammelstörung:** Mehrere Fehler- oder Warnsignale zu einem gemeinsamen Signal zusammenfassen.
- **Mehrfach-Auslösung:** Eine Aktion soll starten, sobald mindestens eine von vielen Bedingungen oder Tasten aktiv ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- `AND_BOOL_19`: Führt die UND-Verknüpfung durch: Das Ergebnis ist nur TRUE, wenn alle Eingänge TRUE sind, während bei `OR_BOOL_19` mindestens ein TRUE-Eingang genügt.
- `OR_BOOL_16`: Gleichartiger Baustein der Familie mit 16 Eingängen.

Fazit

`OR_BOOL_19` ist ein einfacher, ereignisgesteuerter Baustein für die ODER-Verknüpfung von 19 booleschen Signalen mit fester Schnittstelle.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

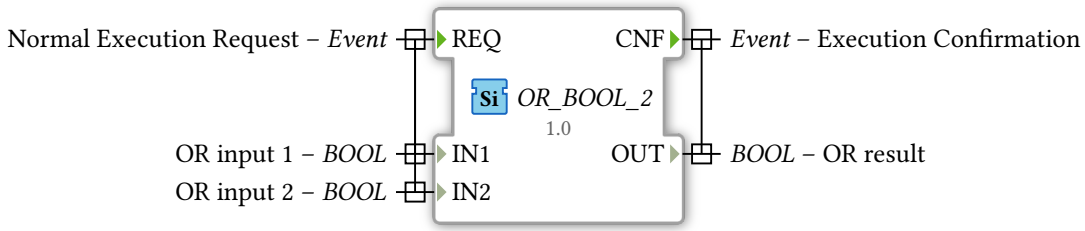
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

OR_BOOL_2

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `OR_BOOL_2` führt eine logische ODER-Verknüpfung (Disjunktion) auf zwei boolesche Eingangswerte aus. Es handelt sich um einen generischen Baustein, der gemäß der IEC 61131-3 Norm für Standard-Boolesche Funktionen klassifiziert ist. Der Block wartet auf eine Ausführungsanforderung, berechnet das Ergebnis und signalisiert dessen Verfügbarkeit.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Normal Execution Request): Dieses Ereignis löst die Ausführung des Funktionsblocks aus. Bei seinem Eintreffen werden die an den Daten-Eingängen `IN1` und `IN2` anliegenden Werte gelesen und die ODER-Operation durchgeführt.

Ereignis-Ausgänge

- **CNF** (Execution Confirmation): Dieses Ereignis wird nach Abschluss der Berechnung erzeugt. Es signalisiert, dass das Ergebnis an den Daten-Ausgang `OUT` ausgegeben wurde und gültig ist.

Daten-Eingänge

- **IN1** (BOOL): Erster Operand für die ODER-Operation.
- **IN2** (BOOL): Zweiter Operand für die ODER-Operation.

Daten-Ausgänge

- **OUT** (BOOL): Ergebnis der logischen ODER-Verknüpfung von `IN1` und `IN2`. Der Ausgangswert ist `TRUE`, wenn mindestens einer der beiden Eingänge `TRUE` ist. Andernfalls ist er `FALSE`.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Die Funktionsweise ist ereignisgesteuert und folgt einem einfachen Anforderungs-Bestätigungs-Zyklus:

1. Das Eintreffen des Ereignisses `REQ` startet die Ausführung.
2. Die aktuellen Werte der Daten-Eingänge `IN1` und `IN2` werden abgefragt.
3. Die logische ODER-Verknüpfung `OUT := IN1 OR IN2` wird berechnet.
4. Das Ergebnis wird an den Daten-Ausgang `OUT` gelegt.
5. Das Ereignis `CNF` wird erzeugt, um den Abschluss der Operation anzuzeigen.

Die Wahrheitstabelle der Operation lautet:

IN1	IN2	OUT (IN1 OR IN2)
0	0	0
0	1	1
1	0	1
1	1	1

Technische Besonderheiten

- **Generischer Baustein:** Der Block ist als generischer Baustein (`GEN_OR`) gekennzeichnet, was bedeutet, dass er als Basis für die Ableitung ähnlicher Bausteine mit anderen Datentypen dienen kann.
- **Einfache Laufzeit:** Die Ausführungszeit ist konstant und sehr kurz, da es sich um eine einfache logische Operation handelt.
- **Kein interner Zustand:** Der Block ist zustandslos (kombinatorisch). Das Ergebnis hängt ausschließlich von den aktuellen Eingangswerten zum Zeitpunkt der `REQ`-Anforderung ab.

Zustandsübersicht

Der Funktionsblock besitzt keinen persistenten internen Zustand. Sein Verhalten ist rein kombinatorisch und wird vollständig durch die Schnittstellenereignisse und -daten beschrieben.

Anwendungsszenarien

- **Logische Verknüpfungen in Steuerungslogik:** Kombination von zwei Bedingungen, bei der eine Aktion ausgeführt werden soll, wenn mindestens eine der Bedingungen erfüllt ist (z.B. „Motor starten, wenn Taster gedrückt ODER Fernbedienungssignal empfangen wird“).
- **Fehleraggregation:** Zusammenfassung mehrerer Fehlersignale zu einem gemeinsamen Fehlerstatus. Liegt mindestens ein Fehler vor (`TRUE`), wird der Gesamtfehlerstatus aktiviert.
- **Freigabelogik:** Erzeugen einer Freigabe, wenn eine von mehreren möglichen Quellen aktiv ist.

- **Grundbaustein in komplexeren logischen Schaltungen:** Als Teil von UND-ODER-Verknüpfungen (SOP - Sum of Products) zur Realisierung von Schaltfunktionen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **AND_BOOL_2:** Führt eine logische UND-Verknüpfung durch. Das Ergebnis ist nur dann `TRUE`, wenn **beide** Eingänge `TRUE` sind.
- **XOR_BOOL_2:** Führt eine exklusive ODER-Verknüpfung (Antivalenz) durch. Das Ergebnis ist `TRUE`, wenn genau **ein** Eingang `TRUE` ist.
- **NOT_BOOL:** Führt eine logische NICHT-Verknüpfung (Negation) auf einem einzelnen Eingang durch.
- **OR_n_BOOL:** Ein generischer ODER-Baustein, der eine variable Anzahl von Eingängen (n) verknüpfen kann, während `OR_BOOL_2` genau zwei feste Eingänge besitzt. Siehe: OR_2

🛠️ Zugehörige Übungen

- Uebung_002b3
- Uebung_160
- Uebung_160b
- Uebung_160b2

Fazit

Der `OR_BOOL_2` ist ein fundamentaler, zuverlässiger und einfach zu verwendender Funktionsblock zur Realisierung der grundlegenden logischen ODER-Funktion in 4diac-IDE Applikationen. Seine ereignisgesteuerte, zustandslose Architektur macht ihn vorhersehbar und gut in den Datenfluss von IEC 61499-Systemen integrierbar. Er ist ein unverzichtbarer Baustein für die Implementierung von Steuerungslogik und Bedingungsauswertungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

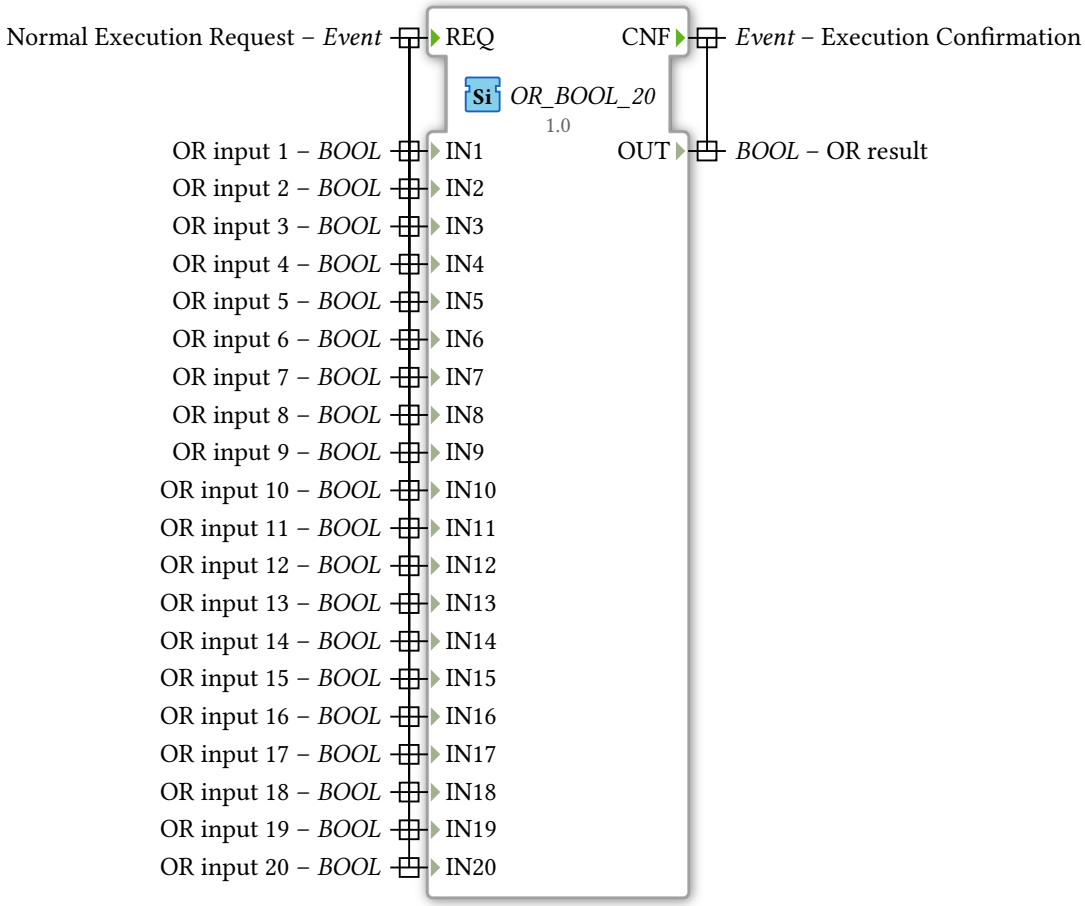
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

OR_BOOL_20

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `OR_BOOL_20` ist ein generischer Baustein zur Berechnung der logischen ODER-Verknüpfung (Klassifikation `standard boolean function`, IEC 61131-3). Er verknüpft 20 separate BOOL-Eingänge (`IN1` bis `IN20`) und liefert das Ergebnis am Ausgang `OUT`. Er ergänzt die Standard-OR_BOOL-Familie (bisher bis `OR_BOOL_16`) um die Variante mit 20 Eingängen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Löst die Berechnung aus. Das Ereignis ist mit allen 20 Daten-Eingängen (`IN1` bis `IN20`) verknüpft.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Signalisiert den Abschluss der Berechnung. Das Ereignis ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN1 (OR input 1):** Boolescher Eingang 1.
- **IN2 (OR input 2):** Boolescher Eingang 2.
- **IN3 (OR input 3):** Boolescher Eingang 3.
- **IN4 (OR input 4):** Boolescher Eingang 4.
- **IN5 (OR input 5):** Boolescher Eingang 5.
- **IN6 (OR input 6):** Boolescher Eingang 6.
- **IN7 (OR input 7):** Boolescher Eingang 7.
- **IN8 (OR input 8):** Boolescher Eingang 8.
- **IN9 (OR input 9):** Boolescher Eingang 9.
- **IN10 (OR input 10):** Boolescher Eingang 10.
- **IN11 (OR input 11):** Boolescher Eingang 11.
- **IN12 (OR input 12):** Boolescher Eingang 12.
- **IN13 (OR input 13):** Boolescher Eingang 13.
- **IN14 (OR input 14):** Boolescher Eingang 14.
- **IN15 (OR input 15):** Boolescher Eingang 15.
- **IN16 (OR input 16):** Boolescher Eingang 16.
- **IN17 (OR input 17):** Boolescher Eingang 17.

- **IN18 (OR input 18):** Boolescher Eingang 18.
- **IN19 (OR input 19):** Boolescher Eingang 19.
- **IN20 (OR input 20):** Boolescher Eingang 20.

Daten-Ausgänge

- **OUT (OR result):** Das Ergebnis der ODER-Verknüpfung aller 20 Eingänge. Der Ausgang ist TRUE, wenn mindestens ein Eingang TRUE ist, und nur FALSE, wenn alle Eingänge `IN1` bis `IN20` FALSE sind.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei jedem Eintreffen des Ereignisses `REQ` werden alle 20 Eingänge gelesen und die Operation `OUT = IN1 OR IN2 OR ... OR IN20` berechnet. Anschließend wird das Ergebnis an `OUT` bereitgestellt und `CNF` ausgelöst.

Technische Besonderheiten

- **Generischer Baustein:** Der Typ trägt das Attribut `GenericClassName = 'GEN_OR_BOOL'`; diese Instanz ist fest auf 20 Eingänge ausgelegt.
- **Package:** `iec61131::booleanOperators`.
- **Zustandslos:** Der Baustein ist speicherlos: `OUT` hängt ausschließlich von den Werten der Eingänge zum Zeitpunkt von `REQ` ab.

Anwendungsszenarien

- **Sammelstörung:** Mehrere Fehler- oder Warnsignale zu einem gemeinsamen Signal zusammenfassen.
- **Mehrfach-Auslösung:** Eine Aktion soll starten, sobald mindestens eine von vielen Bedingungen oder Tasten aktiv ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- `AND_BOOL_20`: Führt die UND-Verknüpfung durch: Das Ergebnis ist nur TRUE, wenn alle Eingänge TRUE sind, während bei `OR_BOOL_20` mindestens ein TRUE-Eingang genügt.
- `OR_BOOL_16`: Gleichartiger Baustein der Familie mit 16 Eingängen.

Fazit

`OR_BOOL_20` ist ein einfacher, ereignisgesteuerter Baustein für die ODER-Verknüpfung von 20 booleschen Signalen mit fester Schnittstelle.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

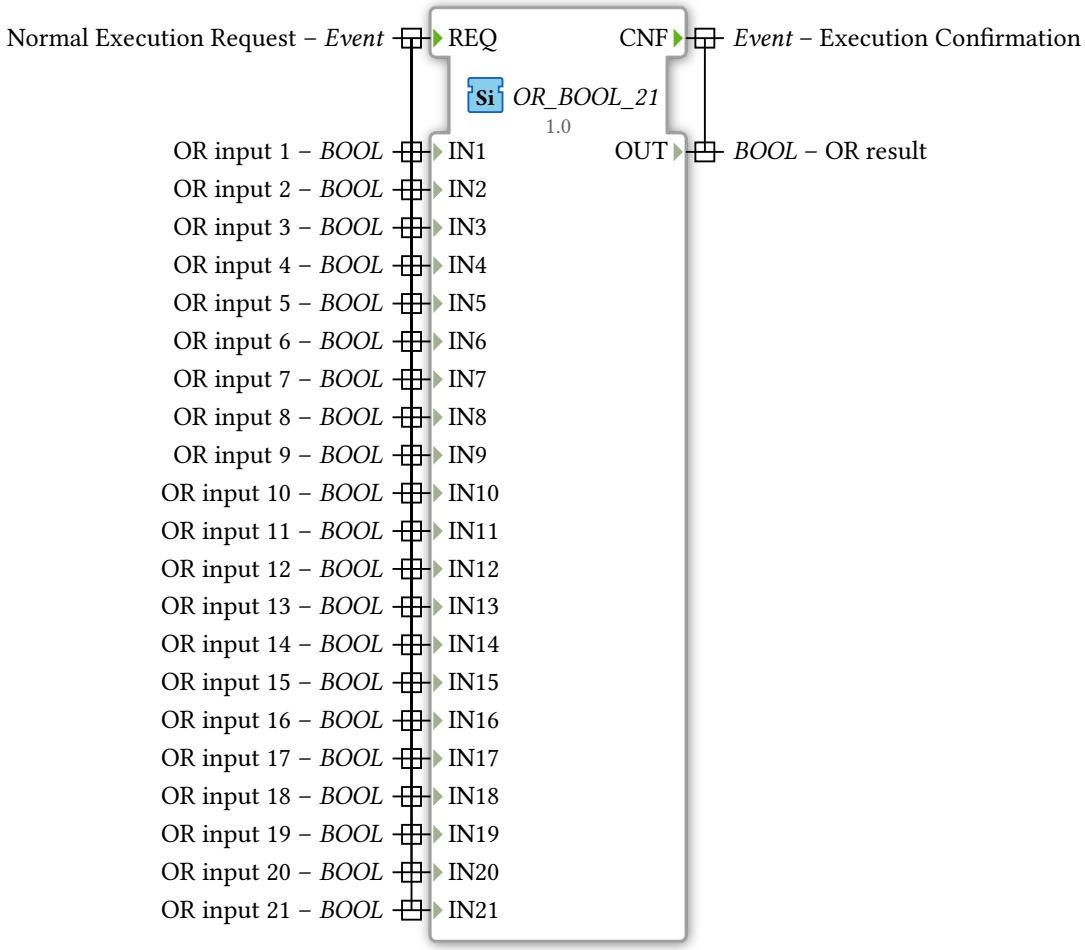
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

OR_BOOL_21

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `OR_BOOL_21` ist ein generischer Baustein zur Berechnung der logischen ODER-Verknüpfung (Klassifikation `standard boolean function`, IEC 61131-3). Er verknüpft 21 separate BOOL-Eingänge (`IN1` bis `IN21`) und liefert das Ergebnis am Ausgang `OUT`. Er ergänzt die Standard-OR_BOOL-Familie (bisher bis `OR_BOOL_16`) um die Variante mit 21 Eingängen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Löst die Berechnung aus. Das Ereignis ist mit allen 21 Daten-Eingängen (`IN1` bis `IN21`) verknüpft.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Signalisiert den Abschluss der Berechnung. Das Ereignis ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN1 (OR input 1):** Boolescher Eingang 1.
- **IN2 (OR input 2):** Boolescher Eingang 2.
- **IN3 (OR input 3):** Boolescher Eingang 3.
- **IN4 (OR input 4):** Boolescher Eingang 4.
- **IN5 (OR input 5):** Boolescher Eingang 5.
- **IN6 (OR input 6):** Boolescher Eingang 6.
- **IN7 (OR input 7):** Boolescher Eingang 7.
- **IN8 (OR input 8):** Boolescher Eingang 8.
- **IN9 (OR input 9):** Boolescher Eingang 9.
- **IN10 (OR input 10):** Boolescher Eingang 10.
- **IN11 (OR input 11):** Boolescher Eingang 11.
- **IN12 (OR input 12):** Boolescher Eingang 12.
- **IN13 (OR input 13):** Boolescher Eingang 13.
- **IN14 (OR input 14):** Boolescher Eingang 14.
- **IN15 (OR input 15):** Boolescher Eingang 15.

- **IN16 (OR input 16):** Boolescher Eingang 16.
- **IN17 (OR input 17):** Boolescher Eingang 17.
- **IN18 (OR input 18):** Boolescher Eingang 18.
- **IN19 (OR input 19):** Boolescher Eingang 19.
- **IN20 (OR input 20):** Boolescher Eingang 20.
- **IN21 (OR input 21):** Boolescher Eingang 21.

Daten-Ausgänge

- **OUT (OR result):** Das Ergebnis der ODER-Verknüpfung aller 21 Eingänge. Der Ausgang ist TRUE, wenn mindestens ein Eingang TRUE ist, und nur FALSE, wenn alle Eingänge `IN1` bis `IN21` FALSE sind.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei jedem Eintreffen des Ereignisses `REQ` werden alle 21 Eingänge gelesen und die Operation `OUT = IN1 OR IN2 OR ... OR IN21` berechnet. Anschließend wird das Ergebnis an `OUT` bereitgestellt und `CNF` ausgelöst.

Technische Besonderheiten

- **Generischer Baustein:** Der Typ trägt das Attribut `GenericClassName = 'GEN_OR_BOOL'`; diese Instanz ist fest auf 21 Eingänge ausgelegt.
- **Package:** `iec61131::booleanOperators`.
- **Zustandslos:** Der Baustein ist speicherlos: `OUT` hängt ausschließlich von den Werten der Eingänge zum Zeitpunkt von `REQ` ab.

Anwendungsszenarien

- **Sammelstörung:** Mehrere Fehler- oder Warnsignale zu einem gemeinsamen Signal zusammenfassen.
- **Mehrfach-Auslösung:** Eine Aktion soll starten, sobald mindestens eine von vielen Bedingungen oder Tasten aktiv ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- `AND_BOOL_21`: Führt die UND-Verknüpfung durch: Das Ergebnis ist nur TRUE, wenn alle Eingänge TRUE sind, während bei `OR_BOOL_21` mindestens ein TRUE-Eingang genügt.
- `OR_BOOL_16`: Gleichartiger Baustein der Familie mit 16 Eingängen.

Fazit

`OR_BOOL_21` ist ein einfacher, ereignisgesteuerter Baustein für die ODER-Verknüpfung von 21 booleschen Signalen mit fester Schnittstelle.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

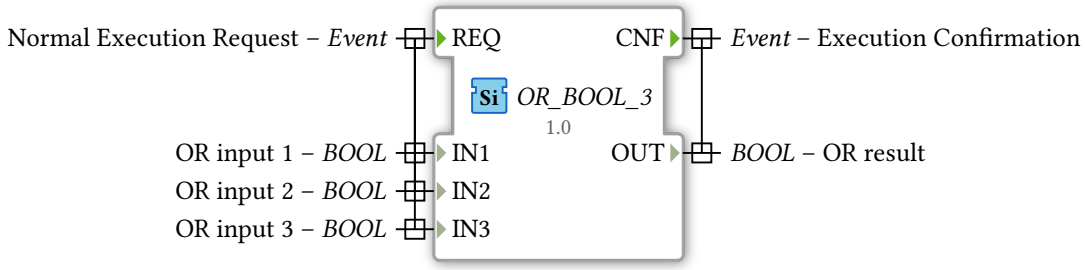
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

OR_BOOL_3

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `OR_BOOL_3` ist ein standardisierter Baustein gemäß IEC 61131-3 zur Durchführung der logischen ODER-Verknüpfung. Er berechnet das logische ODER (Disjunktion) von drei booleschen Eingangssignalen. Der Baustein ist als generische Funktion realisiert und wird bei jedem Auslöseereignis ausgeführt.



Schnittstellenstruktur

Der Funktionsblock verfügt über einen Ereigniseingang, einen Ereignisausgang, drei Dateneingänge und einen Datenausgang. Es sind keine Adapterschnittstellen vorhanden.

Ereignis-Eingänge

- **REQ** (Normal Execution Request): Dieses Ereignis löst die Berechnung der ODER-Funktion aus. Beim Eintreffen von `REQ` werden die aktuellen Werte an den Eingängen `IN1`, `IN2` und `IN3` gelesen und das Ergebnis berechnet.

Ereignis-Ausgänge

- **CNF** (Execution Confirmation): Dieses Ereignis signalisiert den Abschluss der Berechnung. Es wird zusammen mit dem berechneten Ergebnis am Ausgang `OUT` ausgegeben.

Daten-Eingänge

- **IN1** (BOOL): Erstes ODER-Eingangssignal.
- **IN2** (BOOL): Zweites ODER-Eingangssignal.
- **IN3** (BOOL): Drittes ODER-Eingangssignal.

Daten-Ausgänge

- **OUT** (BOOL): Ergebnis der ODER-Verknüpfung (`IN1` ODER `IN2` ODER `IN3`).

Adapter

Es sind keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Bei jedem Eintreffen des Ereignisses `REQ` führt der Funktionsblock eine logische ODER-Verknüpfung der drei booleschen Eingänge `IN1`, `IN2` und `IN3` durch. Das Ergebnis dieser Operation wird an den Datenausgang `OUT` gelegt. Unmittelbar nach der Berechnung wird das Bestätigungsereignis `CNF` ausgelöst, um den erfolgreichen Abschluss der Operation zu signalisieren.

Die logische Funktion lässt sich durch folgende Wahrheitstabelle beschreiben:

IN1	IN2	IN3	OUT (IN1 OR IN2 OR IN3)
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Der Ausgang `OUT` ist genau dann `TRUE` (1), wenn mindestens einer der drei Eingänge `TRUE` ist.

Technische Besonderheiten

- **Generischer Baustein:** Der Block ist als Instanz eines generischen ODER-Bausteins (`GEN_OR`) implementiert, was eine konsistente Integration in Bibliotheken ermöglicht.
- **Ereignisgesteuerte Ausführung:** Die Funktionsausführung ist strikt an das Eintreffen des `REQ`-Ereignisses gekoppelt, was eine deterministische Ablaufsteuerung im Echtzeitsystem gewährleistet.
- **Standardkonformität:** Entspricht der Klassifikation „standard boolean function“ nach IEC 61131-3.

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand (speicherfrei, kombinatorisch). Seine Ausgabe hängt ausschließlich von den aktuellen Werten der Eingänge zum Zeitpunkt des Auslöseereignisses `REQ` ab.

Anwendungsszenarien

- **Sicherheits- und Überwachungslogik:** Kombination mehrerer Fehlersensoren (z.B. Temperatur zu hoch, Druck zu niedrig, Füllstand kritisch), wobei jeder Sensor einen Alarm auslösen kann.
- **Steuerungslogik:** Aktivierung eines Aktors (z.B. einer Pumpe oder eines Motors), wenn mindestens eine von mehreren Bedingungen erfüllt ist (z.B. manueller Startbefehl ODER automatischer Sollwert erreicht ODER Notstart aktiv).
- **Verkettung von Logikoperationen:** Als Teil komplexerer logischer Ausdrücke in Verbindung mit UND- und NOT-Bausteinen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **OR_BOOL_2:** Verknüpft nur zwei Eingänge. `OR_BOOL_3` erweitert diese Funktionalität um einen dritten Eingang. Siehe auch: OR_3
- **GEN_OR:** Der zugrundeliegende generische Baustein, der für eine variable Anzahl von Eingängen konfiguriert werden kann. `OR_BOOL_3` ist eine spezifische, auf drei Eingänge festgelegte Instanz davon.
- **AND_n_BOOL:** Führt eine logische UND-Verknüpfung durch. Das Ergebnis ist nur dann `TRUE`, wenn **alle** Eingänge `TRUE` sind, während bei der ODER-Verknüpfung bereits **ein** `TRUE`-Eingang ausreicht.

🛠️ Zugehörige Übungen

- Uebung_002a5b

Fazit

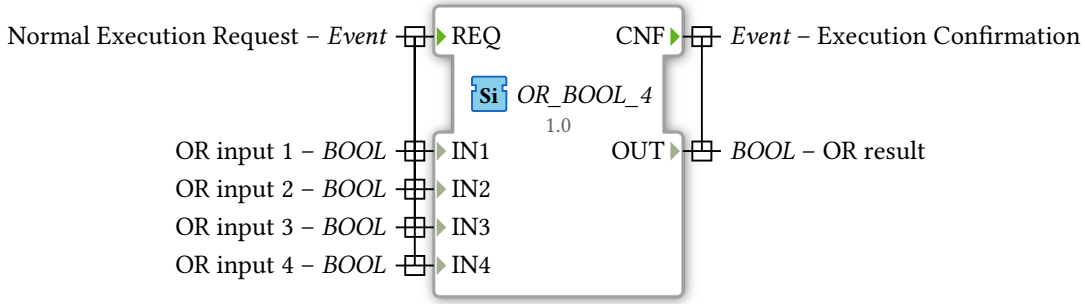
Der `OR_BOOL_3` ist ein grundlegender, zuverlässiger und standardkonformer Funktionsblock zur logischen ODER-Verknüpfung von drei Signalen. Seine ereignisgesteuerte, zustandslose Arbeitsweise macht ihn ideal für den Einsatz in klar strukturierten Steuerungsabläufen der Automatisierungstechnik. Er stellt eine essentielle Komponente für die Realisierung von Überwachungs-, Sicherheits- und Steuerungslogiken dar.

OR_BOOL_4

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `OR_BOOL_4` ist ein Standardbaustein zur Berechnung der logischen ODER-Verknüpfung. Er führt eine ODER-Operation mit bis zu vier booleschen Eingangswerten durch und gibt das Ergebnis als einen einzigen booleschen Wert aus. Der Baustein folgt dem EVA-Prinzip (Ereignisgesteuerte Verarbeitung) und ist für den Einsatz in Steuerungsanwendungen nach IEC 61131-3 konzipiert.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung des Funktionsblocks. Bei diesem Ereignis werden die Werte an den Daten-Eingängen `IN1` bis `IN4` gelesen und die ODER-Operation berechnet.

Ereignis-Ausgänge

- **CNF**: Bestätigt die abgeschlossene Ausführung. Dieses Ereignis wird nach der Berechnung des Ergebnisses ausgelöst und liefert den Wert an den Daten-Ausgang `OUT`.

Daten-Eingänge

- **IN1**: Boolescher ODER-Eingang 1.
- **IN2**: Boolescher ODER-Eingang 2.
- **IN3**: Boolescher ODER-Eingang 3.
- **IN4**: Boolescher ODER-Eingang 4.

Daten-Ausgänge

- **OUT**: Boolesches Ergebnis der ODER-Verknüpfung aller aktiven Eingänge.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei Eintreffen des Ereignisses `REQ` werden die aktuellen Werte an den vier booleschen Eingängen `IN1`, `IN2`, `IN3` und `IN4` ausgelesen. Der Funktionsblock berechnet das logische ODER dieser vier Werte nach folgender Regel: Das Ergebnis `OUT` ist `TRUE` (1), wenn mindestens einer der vier Eingänge `TRUE` ist. Das Ergebnis ist nur dann `FALSE` (0), wenn alle vier Eingänge `FALSE` sind. Nach der Berechnung wird das Bestätigungsereignis `CNF` ausgelöst, wobei der berechnete Wert am Ausgang `OUT` mitgeführt wird.

Technische Besonderheiten

- **Generischer Baustein**: Der Block ist als generischer Baustein gekennzeichnet (`GEN_OR`). Dies ermöglicht es, ihn als Basis für die Erstellung von ODER-Bausteinen mit einer anderen Anzahl von Eingängen zu verwenden.
- **Ereignisgesteuert**: Die Operation wird nur bei einem eingehenden `REQ`-Ereignis ausgeführt, was eine ressourcenschonende und deterministische Abarbeitung im Steuerungssystem gewährleistet.
- **Standardkonform**: Entspricht der Klassifikation nach IEC 61131-3 für Standard-Boolesche Funktionen.

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand (speicherfrei). Sein Verhalten ist kombinatorisch und hängt ausschließlich von den aktuellen Werten an den Eingängen zum Zeitpunkt des `REQ`-Ereignisses ab.

Anwendungsszenarien

- **Verknüpfung von Meldesignalen**: Kombination mehrerer Zustands- oder Fehlermelder, bei denen bereits eine einzige Meldung eine Aktion auslösen soll (z.B. „Maschine anhalten, wenn Sensor A ODER Sensor B ODER Not-Halt betätigt ist“).
- **Logik in Sicherheitssteuerungen**: Implementierung von ODER-Gattern in Sicherheitsschaltungen.
- **Vereinfachung von Steuerungslogik**: Als klar definierter Baustein zur Strukturierung von Programmen in der Funktionsbausteinsprache (FBD) oder im Ablaufsprache (SFC).

⚖️ Vergleich mit ähnlichen Bausteinen

- **`OR_BOOL_2` / `OR_BOOL_8`**: Diese Blöcke führen die identische logische ODER-Operation durch, jedoch mit einer unterschiedlichen Anzahl von Eingängen (2 bzw. 8). Die Wahl des Bausteins hängt von der benötigten Anzahl an zu verknüpfenden Signalen ab. Siehe: OR_4
- **`AND_BOOL_4`**: Führt eine logische UND-Verknüpfung durch. Das Ergebnis ist nur `TRUE`, wenn *alle* Eingänge `TRUE` sind.
- **`XOR_BOOL_4`**: Führt eine exklusiv-ODER (XOR) Verknüpfung durch. Das Ergebnis ist `TRUE`, wenn eine ungerade Anzahl an Eingängen `TRUE` ist.

Fazit

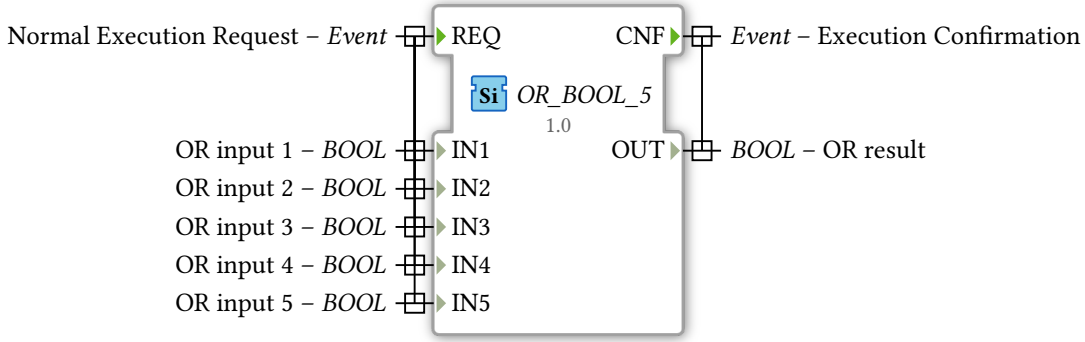
Der `OR_BOOL_4` ist ein grundlegender, zuverlässiger und standardkonformer Funktionsblock zur Realisierung einer ODER-Logik mit vier Eingängen. Seine ereignisgesteuerte, zustandslose Arbeitsweise macht ihn ideal für die Integration in größere, ereignisgesteuerte Steuerungsanwendungen nach IEC 61131-3. Durch seine generische Natur dient er zudem als Basis für die Bibliothekserweiterung.

OR_BOOL_5

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `OR_BOOL_5` ist ein Standardbaustein zur Berechnung der logischen ODER-Verknüpfung. Er führt eine ODER-Operation über fünf separate boolesche Eingangssignale aus. Der Baustein gehört zur Kategorie der Standard-Booleschen Funktionen gemäß IEC 61131-3 und ist als generischer Baustein implementiert.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Dieses Ereignis löst die Ausführung des Funktionsblocks aus. Beim Eintreffen von `REQ` werden die Werte aller fünf Eingänge (`IN1` bis `IN5`) gelesen und die ODER-Operation berechnet.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Dieses Ereignis signalisiert den Abschluss der Berechnung. Es wird zusammen mit dem berechneten Ergebnis `OUT` ausgegeben.

Daten-Eingänge

- **IN1 (BOOL):** ODER-Eingang 1.
- **IN2 (BOOL):** ODER-Eingang 2.
- **IN3 (BOOL):** ODER-Eingang 3.
- **IN4 (BOOL):** ODER-Eingang 4.
- **IN5 (BOOL):** ODER-Eingang 5.

Daten-Ausgänge

- **OUT (BOOL):** Das Ergebnis der ODER-Verknüpfung aller fünf Eingänge. Der Ausgang ist `TRUE`, wenn mindestens einer der Eingänge `IN1` bis `IN5` den Wert `TRUE` hat.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Die Funktionsweise ist deterministisch und ereignisgesteuert:

1. Das Eintreffen des Ereignisses `REQ` startet die Ausführung.
2. Der Block liest die aktuellen Werte der fünf booleschen Eingänge `IN1` bis `IN5`.
3. Es wird die logische ODER-Verknüpfung `OUT = IN1 OR IN2 OR IN3 OR IN4 OR IN5` berechnet.

4. Das Ergebnis wird am Datenausgang `OUT` bereitgestellt und gleichzeitig wird das Bestätigungsereignis `CNF` ausgelöst, um den erfolgreichen Abschluss der Operation an nachfolgende Blöcke zu signalisieren.

Technische Besonderheiten

- **Generischer Baustein:** Der Block ist mit dem Attribut `eclipse4diac::core::GenericClassName` als `'GEN_OR'` gekennzeichnet. Dies deutet darauf hin, dass er Teil einer Familie generischer ODER-Bausteine mit unterschiedlicher Anzahl an Eingängen ist.
- **Package:** Der Baustein ist im Package `iec61131::bitwiseOperators` enthalten.
- **Ereignisgesteuerte Ausführung:** Die Berechnung erfolgt nur bei Anforderung durch ein `REQ`-Ereignis, was eine energieeffiziente und deterministische Ablaufsteuerung ermöglicht.

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand im Sinne eines Speichers. Sein Verhalten ist rein kombinatorisch und hängt ausschließlich von den aktuellen Eingangswerten zum Zeitpunkt des `REQ`-Ereignisses ab. Der Block kehrt nach dem Auslösen von `CNF` in einen passiven Wartezustand zurück.

Anwendungsszenarien

Typische Anwendungen sind in Steuerungslogik zu finden, wo eine Bedingung als erfüllt gilt, wenn eine von mehreren möglichen Teilbedingungen zutrifft:

- Überwachung: Alarmauslösung, wenn mindestens eines von mehreren Überwachungssensoren einen Grenzwert überschreitet.
- Freigabelogik: Start einer Maschine, wenn mindestens eine von mehreren Sicherheitseinrichtungen aktiviert ist.
- Auswahl- oder Priorisierungslogik in verbundenen Steuerungssystemen.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu diesem Baustein mit fünf festen Eingängen existieren in 4diac oft auch:

- **OR_BOOL_2, OR_BOOL_3, etc.:** Gleiche Funktionalität, aber mit einer geringeren, festen Anzahl an Eingängen (z.B. 2, 3, 4). Siehe: OR_5
- **GEN_OR (Generisches ODER):** Ein übergeordneter, konfigurierbarer Baustein, der je nach Instanziierung eine variable Anzahl von Eingängen besitzen kann. `OR_BOOL_5` ist eine spezifische Instanziierung dieses Prinzips.
- **AND_BOOL_5:** Führt eine logische UND-Verknüpfung über alle fünf Eingänge aus (Ergebnis ist nur `TRUE`, wenn *alle* Eingänge `TRUE` sind).

Fazit

Der `OR_BOOL_5` ist ein einfacher, robuster und standardkonformer Funktionsblock zur Realisierung einer ODER-Verknüpfung mit fünf Eingängen. Seine ereignisgesteuerte Architektur integriert sich nahtlos in die Funktionsbaustein-Netzwerke von 4diac und eignet sich ideal für klare und übersichtliche Logikverknüpfungen in Automatisierungsanwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

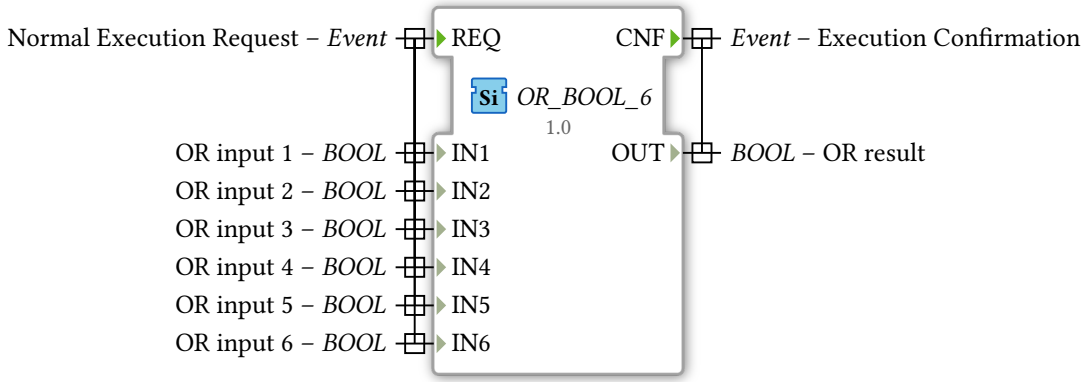
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

OR_BOOL_6

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `OR_BOOL_6` ist ein generischer Baustein zur Berechnung der logischen ODER-Verknüpfung. Er führt eine ODER-Operation über sechs separate BOOL-Eingänge durch und gibt das Ergebnis auf einem einzigen Ausgang aus. Der Baustein folgt dem IEC 61131-3 Standard und ist für den Einsatz in Steuerungsanwendungen konzipiert, in denen mehrere Bedingungen kombiniert werden müssen, um eine Aktion auszulösen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Dieses Ereignis löst die Berechnung der ODER-Funktion aus. Es ist mit allen sechs Dateneingängen (`IN1` bis `IN6`) verknüpft.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Dieses Ereignis signalisiert den Abschluss der Berechnung. Es wird zusammen mit dem berechneten Ergebnis am Datenausgang `OUT` ausgegeben.

Daten-Eingänge

- **IN1 (BOOL):** ODER-Eingang 1.
- **IN2 (BOOL):** ODER-Eingang 2.
- **IN3 (BOOL):** ODER-Eingang 3.
- **IN4 (BOOL):** ODER-Eingang 4.
- **IN5 (BOOL):** ODER-Eingang 5.
- **IN6 (BOOL):** ODER-Eingang 6.

Daten-Ausgänge

- **OUT (BOOL):** Ergebnis der ODER-Verknüpfung aller sechs Eingänge. Der Ausgang ist `TRUE`, wenn mindestens einer der Eingänge `TRUE` ist.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Die Funktionsweise ist deterministisch und ereignisgesteuert. Bei jedem Eintreffen des Eingangsereignisses `REQ` werden die aktuellen Werte aller sechs BOOL-Eingänge (`IN1` bis `IN6`) ausgelesen. Anschließend wird die logische ODER-Verknüpfung dieser Werte berechnet. Das Ergebnis dieser Operation wird unmittelbar am Datenausgang `OUT` gesetzt und gleichzeitig wird das Bestätigungsereignis `CNF` ausgelöst, um den Abschluss der Operation an nachfolgende Blöcke zu signalisieren.

Die logische Operation entspricht der Formel: `OUT = IN1 OR IN2 OR IN3 OR IN4 OR IN5 OR IN6`

Technische Besonderheiten

- **Generischer Baustein:** Der Block ist als generischer Baustein gekennzeichnet (`GEN_OR`). Dies bedeutet, dass er als Basis für die Erstellung ähnlicher Blöcke mit einer anderen Anzahl von Eingängen dienen kann.
- **Ereignisgesteuerte Ausführung:** Die Berechnung erfolgt nur bei Anforderung durch ein Ereignis, was eine effiziente und bedarfsgerechte Abarbeitung im Steuerungssystem ermöglicht.
- **Feste Eingangsanzahl:** Diese spezielle Instanz ist für genau sechs Eingänge festgelegt.

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand oder Speicher. Sein Verhalten ist rein kombinatorisch und hängt ausschließlich von den aktuellen Werten der Eingänge zum Zeitpunkt des `REQ`-Ereignisses ab. Nach der Ausgabe von `CNF` verharrt der Block in einem wartenden Zustand bis zum nächsten `REQ`.

Anwendungsszenarien

Typische Anwendungen sind Überwachungs- und Sicherheitslogiken, bei denen eine Aktion ausgelöst werden soll, wenn eine von mehreren möglichen Bedingungen erfüllt ist.

- **Not-Aus Logik:** Der Block kann genutzt werden, um mehrere Not-Aus-Taster zu überwachen. Wenn irgendeiner der Taster gedrückt wird (`TRUE`), wird ein Abschaltsignal (`OUT = TRUE`) generiert.
- **Alarmaggregation:** Mehrere Sensoren, die jeweils einen Fehlerzustand detektieren können, werden zusammengeschaltet. Ein allgemeiner Alarm wird aktiviert, sobald mindestens ein Sensor einen Fehler meldet.
- **Mehrfachbedienung:** In einer Maschine können mehrere Bedienstationen einen Prozess starten. Der Block kombiniert alle Startbefehle zu einem gemeinsamen Startsignal.

⚖️ Vergleich mit ähnlichen Bausteinen

- **AND_BOOL_6:** Führt eine logische UND-Verknüpfung durch. Das Ergebnis ist nur `TRUE`, wenn **alle** Eingänge `TRUE` sind. Im Gegensatz dazu benötigt `OR_BOOL_6` nur **einen** wahren Eingang.
- **XOR_BOOL:** Führt eine exklusiv-ODER (XOR) Verknüpfung durch. Das Ergebnis ist `TRUE`, wenn eine ungerade Anzahl an Eingängen `TRUE` ist. `OR_BOOL_6` unterscheidet nicht zwischen einer oder mehreren wahren Bedingungen. Siehe: OR_6
- **Blöcke mit variabler Eingangsanzahl:** Es existieren ODER-Blöcke mit zwei (`OR`) oder einer anderen Anzahl von Eingängen. `OR_BOOL_6` bietet eine feste, für bestimmte Anwendungen optimierte Schnittstelle mit sechs Eingängen.

Fazit

Der `OR_BOOL_6` Funktionsblock ist ein einfacher, robuster und standardkonformer Baustein zur logischen Signalverknüpfung. Seine Stärke liegt in der klaren, ereignisgesteuerten Schnittstelle und der festen Anzahl von sechs Eingängen, die ihn ideal für Anwendungen macht, die genau diese Anzahl an zu kombinierenden Bedingungen erfordern. Als generischer Baustein unterstützt er zudem die Wiederverwendung und Anpassung innerhalb der 4diac-IDE.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

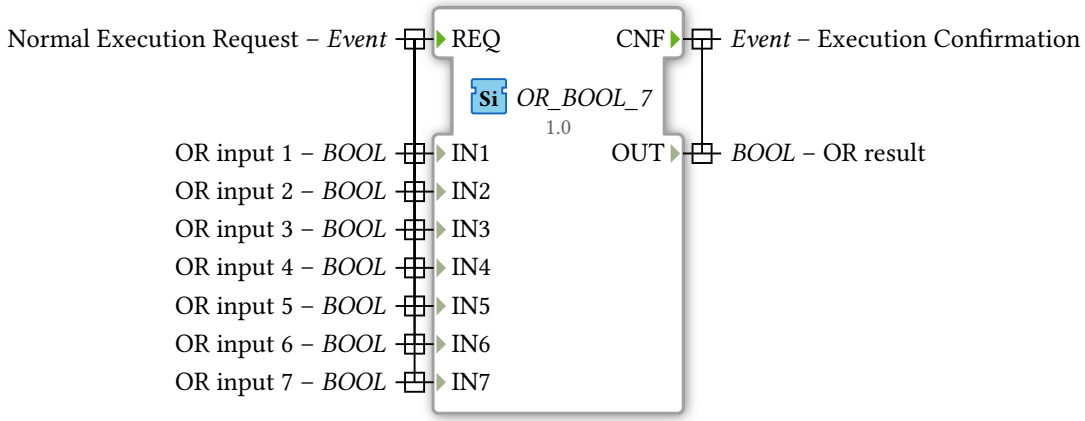
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

OR_BOOL_7

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `OR_BOOL_7` ist ein Standardbaustein zur Berechnung der logischen ODER-Verknüpfung. Er führt eine ODER-Operation über sieben separate boolesche Eingänge durch und stellt das Ergebnis an einem einzigen Ausgang zur Verfügung. Dieser Baustein ist Teil der IEC 61131-3-konformen Bibliothek für bitweise Operationen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Normal Execution Request): Dieses Ereignis löst die Berechnung der ODER-Funktion aus. Es ist mit allen sieben Dateneingängen (`IN1` bis `IN7`) verknüpft.

Ereignis-Ausgänge

- **CNF** (Execution Confirmation): Dieses Ereignis signalisiert den Abschluss der Berechnung. Es wird zusammen mit dem berechneten Ergebnis am Datenausgang `OUT` ausgegeben.

Daten-Eingänge

- **IN1** (BOOL): ODER-Eingang 1.
- **IN2** (BOOL): ODER-Eingang 2.
- **IN3** (BOOL): ODER-Eingang 3.
- **IN4** (BOOL): ODER-Eingang 4.
- **IN5** (BOOL): ODER-Eingang 5.
- **IN6** (BOOL): ODER-Eingang 6.
- **IN7** (BOOL): ODER-Eingang 7.

Daten-Ausgänge

- **OUT** (BOOL): Ergebnis der ODER-Verknüpfung aller sieben Eingänge.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei Empfang des Ereignisses `REQ` liest der Baustein die Werte aller sieben booleschen Eingänge (`IN1` bis `IN7`) ein. Anschließend wird die logische ODER-Operation auf diese Werte angewendet. Das Ergebnis dieser Operation wird am Datenausgang `OUT` gesetzt und gleichzeitig wird das Bestätigungsereignis `CNF` ausgelöst.

Die logische Funktion lässt sich wie folgt beschreiben: `OUT = IN1 OR IN2 OR IN3 OR IN4 OR IN5 OR IN6 OR IN7`

Das bedeutet, der Ausgang `OUT` ist `TRUE` (1), wenn mindestens einer der sieben Eingänge `TRUE` ist. Der Ausgang ist nur dann `FALSE` (0), wenn alle sieben Eingänge `FALSE` sind.

Technische Besonderheiten

- **Generischer Baustein:** Der FB ist als generische Implementierung (`GEN_OR`) gekennzeichnet, was auf eine mögliche Wiederverwendung des Kerns für ähnliche Bausteine hindeutet.
- **Fest verdrahtete Logik:** Die ODER-Verknüpfung ist für genau sieben Eingänge fest implementiert. Für eine andere Anzahl von Eingängen muss ein entsprechender anderer Baustein (z.B. `OR_BOOL_2`, `OR_BOOL_4`) verwendet werden.
- **Ereignisgesteuerte Ausführung:** Die Berechnung erfolgt ausschließlich bei Auftreten des Eingangsereignisses `REQ`, was eine ressourcenschonende und deterministische Abarbeitung im Echtzeitsystem ermöglicht.

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand oder Speicher. Sein Verhalten ist rein kombinatorisch und ereignisgesteuert:

1. **Ruhezustand:** Wartet auf das Ereignis `REQ`.
2. **Berechnungszustand:** Bei `REQ` werden die Eingänge gelesen und die ODER-Operation durchgeführt.
3. **Ausgabezustand:** Das Ergebnis wird an `OUT` ausgegeben und das Ereignis `CNF` wird generiert. Anschließend kehrt der Baustein in den Ruhezustand zurück.

Anwendungsszenarien

- **Überwachungslogik:** Zusammenführen mehrerer Fehler- oder Warnsignale zu einem gemeinsamen Alarmausgang. Wenn eine von sieben möglichen Störquellen aktiv wird, leuchtet eine gemeinsame Warnlampe auf.
- **Freigabelogik:** Kombinieren mehrerer Freigabesignale (z.B. aus verschiedenen Sicherheitssystemen) für einen Prozessschritt. Der Prozess startet, wenn mindestens eine Bedingung erfüllt ist.

- **Verknüpfung von Sensoren:** Auswerten mehrerer Grenzwertmelder, um zu prüfen, ob mindestens ein Sensor einen Schwellenwert überschritten hat.

⚖️ Vergleich mit ähnlichen Bausteinen

- **AND_BOOL_7:** Führt eine logische UND-Verknüpfung durch. Der Ausgang ist nur `TRUE`, wenn *alle* Eingänge `TRUE` sind.
- **XOR_BOOL_7:** Führt eine exklusiv-ODER (XOR) Verknüpfung durch. Der Ausgang ist `TRUE`, wenn eine ungerade Anzahl an Eingängen `TRUE` ist.
- **OR_n_BOOL (n=2,4,...):** Bausteine derselben Familie, die die ODER-Funktion für eine unterschiedliche, festgelegte Anzahl von Eingängen bereitstellen. Die Wahl des Bausteins hängt von der benötigten Anzahl an Signalen ab. Siehe: OR_7
- **Generic FB (GEN_OR):** Die zugrundeliegende generische Implementierung, die für die Erstellung der spezifischen `OR_n_BOOL`-Varianten verwendet wird.

Fazit

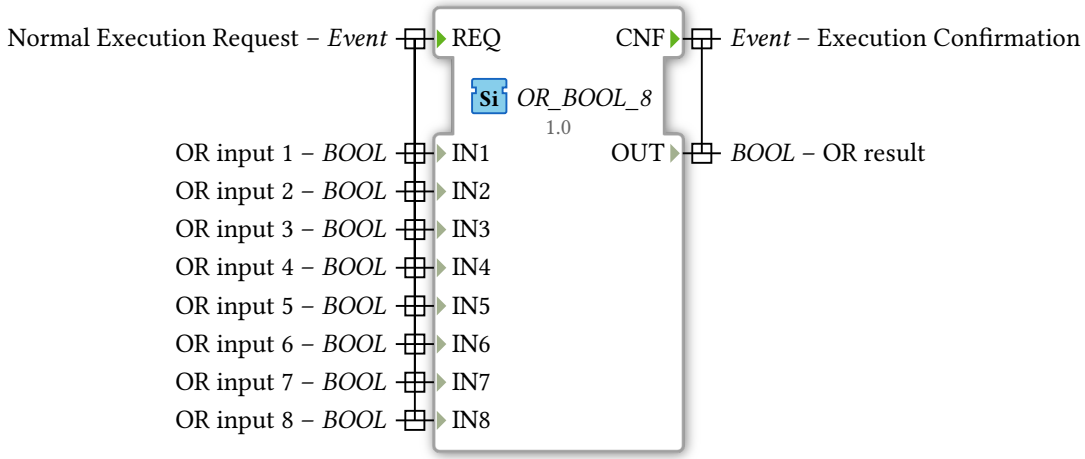
Der `OR_BOOL_7` ist ein einfacher, robuster und standardisierter Funktionsblock zur logischen ODER-Verknüpfung von sieben booleschen Signalen. Seine ereignisgesteuerte Architektur macht ihn ideal für die Integration in steuerungstechnische Anwendungen nach IEC 61131-3, wo er zuverlässig und deterministisch arbeitet. Er ist die richtige Wahl, wenn genau sieben binäre Zustände zu einem gemeinsamen Ergebnis verknüpft werden müssen.

OR_BOOL_8

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `OR_BOOL_8` ist ein standardisierter Baustein zur Berechnung der logischen ODER-Verknüpfung. Er führt die ODER-Operation über acht separate boolesche Eingangswerte aus und liefert das Ergebnis auf einem einzelnen booleschen Ausgang. Dieser Baustein ist Teil der IEC 61131-3-konformen Bibliothek für bitweise Operationen und dient der einfachen und strukturierten Logikverarbeitung in Steuerungsanwendungen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Normal Execution Request): Dieses Ereignis löst die Ausführung des Funktionsblocks aus. Bei seinem Eintreffen werden alle acht Daten-Eingänge (`IN1` bis `IN8`) ausgelesen und die ODER-Operation berechnet.

Ereignis-Ausgänge

- **CNF** (Execution Confirmation): Dieses Ereignis signalisiert den Abschluss der Berechnung. Es wird zusammen mit dem berechneten Ergebnis am Datenausgang `OUT` ausgegeben.

Daten-Eingänge

- **IN1** (BOOL): ODER-Eingang 1.
- **IN2** (BOOL): ODER-Eingang 2.
- **IN3** (BOOL): ODER-Eingang 3.
- **IN4** (BOOL): ODER-Eingang 4.
- **IN5** (BOOL): ODER-Eingang 5.
- **IN6** (BOOL): ODER-Eingang 6.
- **IN7** (BOOL): ODER-Eingang 7.
- **IN8** (BOOL): ODER-Eingang 8.

Daten-Ausgänge

- **OUT** (BOOL): Ergebnis der ODER-Verknüpfung aller acht Eingänge. Der Ausgang ist `TRUE` (1), wenn mindestens einer der Eingänge `IN1` bis `IN8` den Wert `TRUE` hat. Andernfalls ist der Ausgang `FALSE` (0).

Adapter

Dieser Funktionsblock verwendet keine Adapter-Schnittstellen.

Funktionsweise

Die Funktionsweise ist ereignisgesteuert und folgt einem einfachen Request-Confirm-Zyklus:

1. Das Eintreffen des Ereignisses `REQ` startet die Ausführung.
2. Der Block liest die aktuellen Werte aller acht booleschen Eingänge (`IN1` bis `IN8`) ein.
3. Es wird die logische ODER-Verknüpfung über alle acht Eingänge berechnet: `OUT = IN1 OR IN2 OR IN3 OR IN4 OR IN5 OR IN6 OR IN7 OR IN8`.
4. Das Ergebnis wird am Datenausgang `OUT` bereitgestellt und gleichzeitig wird das Bestätigungsereignis `CNF` ausgelöst, um den erfolgreichen Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- **Generischer Baustein:** Der Block ist als generischer Baustein (`GEN_OR`) gekennzeichnet, was bedeutet, dass er als Basis für die Erstellung ähnlicher ODER-Bausteine mit einer anderen Anzahl von Eingängen dienen kann.
- **Standardkonform:** Er ist gemäß der Norm IEC 61131-3 klassifiziert und implementiert eine standardisierte boolesche Funktion.
- **Deterministisches Verhalten:** Die Ausführung ist deterministisch und führt zu keiner internen Zustandsspeicherung zwischen den Aufrufen.

Zustandsübersicht

Der Funktionsblock `OR_BOOL_8` ist zustandslos (kombinatorische Logik). Er besitzt keinen internen Speicher. Die Ausgabe `OUT` ist zu jedem Zeitpunkt ausschließlich eine direkte Funktion der aktuellen Eingangswerte, die mit dem letzten `REQ`-Ereignis eingelesen wurden.

Anwendungsszenarien

Typische Anwendungen sind:

- **Überwachungslogik:** Zusammenfassen mehrerer Fehler- oder Warnsignale (z.B. von verschiedenen Sensoren oder Teilsystemen) zu einem gemeinsamen Alarmausgang.
- **Freigabelogik:** Prüfen, ob mindestens eine von mehreren möglichen Freigabebedingungen (z.B. „Manuell freigegeben“ ODER „Automatik freigegeben“) erfüllt ist.
- **Verknüpfung von Tastereingaben:** In Bedienpanels, um eine Aktion auszulösen, wenn einer von mehreren Tastern gedrückt wird.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`OR_BOOL_2`, `OR_BOOL_4`:** Dies sind baugleiche Funktionsblöcke, die jedoch nur zwei bzw. vier Eingänge verknüpfen. `OR_BOOL_8` bietet eine höhere Eingangsanzahl, ohne mehrere Instanzen von Blöcken mit weniger Eingängen kaskadieren zu müssen. Siehe: OR_8
- **`AND_n_BOOL`:** Führt die logische UND-Verknüpfung durch. Das Ergebnis ist nur `TRUE`, wenn *alle* Eingänge `TRUE` sind, während bei der ODER-Verknüpfung bereits ein einziger `TRUE`-Eingang ausreicht.
- **`XOR_n_BOOL`:** Führt die exklusive ODER-Verknüpfung (XOR) durch. Das Ergebnis ist `TRUE`, wenn eine ungerade Anzahl an Eingängen `TRUE` ist, was sich von der inklusiven ODER-Logik unterscheidet.

Fazit

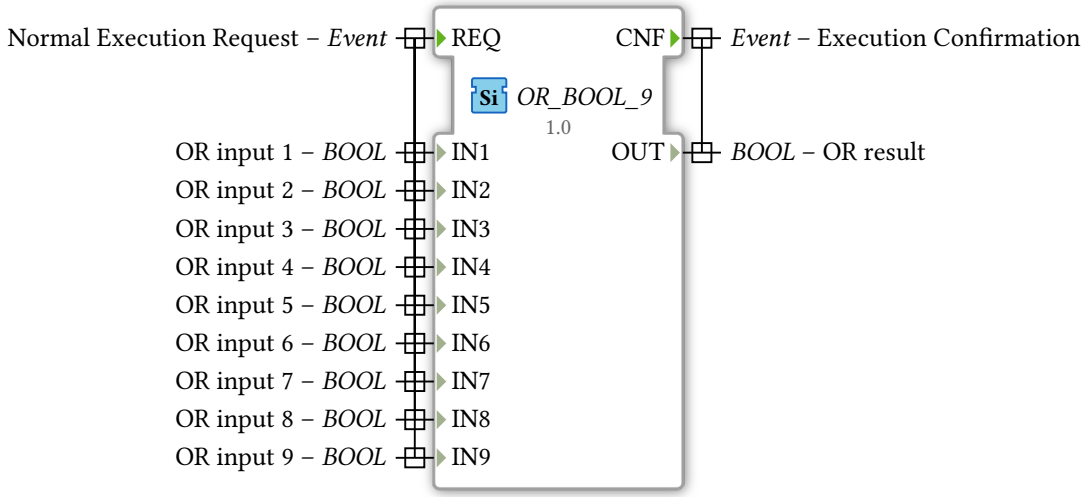
Der `OR_BOOL_8` Funktionsblock ist ein grundlegender, zuverlässiger und einfach anzuwendender Baustein für die boolesche Logikverarbeitung. Seine Stärke liegt in der klaren, normkonformen Schnittstelle und der effizienten Verarbeitung von bis zu acht Signalen zu einer gemeinsamen Aussage. Für Anwendungen, die eine inklusive ODER-Verknüpfung über mehrere Signale benötigen, ist er die erste Wahl.

OR_BOOL_9

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `OR_BOOL_9` ist ein standardisierter Baustein gemäß IEC 61131-3 zur Berechnung der logischen ODER-Verknüpfung. Er führt die ODER-Operation über neun separate boolesche Eingänge aus. Der Block folgt dem ereignisgesteuerten Ausführungsmodell der 4diac-IDE: Die Berechnung wird durch ein eingehendes Ereignis ausgelöst, und das Ergebnis wird zusammen mit einem Bestätigungsereignis ausgegeben.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Ausführung des Funktionsblocks aus. Bei Eintreffen dieses Ereignisses werden alle neun Eingangswerte (`IN1` bis `IN9`) gelesen und die ODER-Operation berechnet.

Ereignis-Ausgänge

- **CNF**: Dieses Ereignis wird nach Abschluss der Berechnung erzeugt. Es signalisiert, dass das Ergebnis am Datenausgang `OUT` gültig und verfügbar ist.

Daten-Eingänge

- **IN1** (BOOL): ODER-Eingang 1.
- **IN2** (BOOL): ODER-Eingang 2.
- **IN3** (BOOL): ODER-Eingang 3.
- **IN4** (BOOL): ODER-Eingang 4.
- **IN5** (BOOL): ODER-Eingang 5.
- **IN6** (BOOL): ODER-Eingang 6.
- **IN7** (BOOL): ODER-Eingang 7.
- **IN8** (BOOL): ODER-Eingang 8.
- **IN9** (BOOL): ODER-Eingang 9.

Daten-Ausgänge

- **OUT** (BOOL): Ergebnis der logischen ODER-Verknüpfung aller neun Eingänge. Der Ausgang ist `TRUE` (1), wenn mindestens einer der Eingänge `TRUE` ist. Er ist nur `FALSE` (0), wenn alle neun Eingänge `FALSE` sind.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Der `OR_BOOL_9`-Block arbeitet nach folgendem Algorithmus:

1. **Auslösung**: Das eingehende Ereignis `REQ` aktiviert den Block.
2. **Eingabesammlung**: Die aktuellen Werte aller neun booleschen Eingänge (`IN1` bis `IN9`) werden eingelesen.
3. **Berechnung**: Die logische ODER-Funktion wird auf die Eingangswerte angewendet. Die Operation kann als `OUT = IN1 OR IN2 OR IN3 OR IN4 OR IN5 OR IN6 OR IN7 OR IN8 OR IN9` beschrieben werden.
4. **Ausgabe**: Das berechnete Ergebnis wird am Datenausgang `OUT` gesetzt.
5. **Bestätigung**: Unmittelbar nach dem Setzen des Ausgangs wird das Bestätigungsereignis `CNF` erzeugt, um nachgelagerte Blöcke über die Verfügbarkeit des neuen Ergebnisses zu informieren.

Technische Besonderheiten

- **Generischer Funktionsblock**: Der Block ist als generischer Baustein implementiert (Attribut `eclipse4diac::core::GenericClassName` ist auf `'GEN_OR'` gesetzt). Dies bedeutet, dass er als Basis für die Ableitung ähnlicher ODER-Blöcke mit einer anderen Anzahl von Eingängen dienen kann.
- **Standardkonformität**: Er ist explizit als standardisierte boolesche Funktion nach IEC 61131-3 klassifiziert.
- **Deterministisches Verhalten**: Die Ausführungszeit ist konstant und vorhersehbar, da es sich um eine einfache logische Operation handelt.

Zustandsübersicht

Der Block besitzt keinen internen Zustand oder Speicher. Sein Verhalten ist rein kombinatorisch und ausschließlich von den aktuellen Eingangswerten zum Zeitpunkt des `REQ`-Ereignisses abhängig. Vorherige Berechnungen haben keinen Einfluss auf das aktuelle Ergebnis.

Anwendungsszenarien

- **Überwachungslogik**: Kombination mehrerer Fehler- oder Statusmeldungen (z.B. von neun Sensoren), wobei bereits eine einzige Meldung (`TRUE`) zu einer Gesamtalarmmeldung (`OUT = TRUE`) führt.
- **Freigabelogik**: In Sicherheitsschaltungen, wo eine von mehreren möglichen Freigabebedingungen ausreicht, um einen Prozessschritt zu starten.
- **Vereinfachung von Verknüpfungsnetzen**: Ersetzt mehrere hintereinandergeschaltete Zwei-Eingangs-ODER-Gatter, was die Übersichtlichkeit in einem Funktionsbaustein-Diagramm (FBD) erhöht.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`OR_BOOL_2`, `OR_BOOL_4` etc.**: Diese Blöcke bieten die gleiche ODER-Funktionalität, jedoch für eine geringere Anzahl von Eingängen (2, 4, ...). `OR_BOOL_9` ist für Anwendungen mit speziell neun zu verknüpfenden Signalen optimiert. Siehe: OR_9
- **`AND_BOOL_9`**: Führt die logische UND-Verknüpfung durch. Das Ergebnis ist nur `TRUE`, wenn *alle* Eingänge `TRUE` sind, im Gegensatz zum ODER, wo *mindestens ein* Eingang `TRUE` sein muss.
- **`XOR_BOOL_9`**: Berechnet das exklusive ODER (Antivalenz). Das Ergebnis ist `TRUE`, wenn eine ungerade Anzahl an Eingängen `TRUE` ist.

Fazit

Der `OR_BOOL_9` ist ein robuster und einfach zu verwendender Standardbaustein für die logische ODER-Verknüpfung von neun booleschen Signalen. Seine ereignisgesteuerte, zustandslose Architektur macht ihn zu einem zuverlässigen und effizienten Bestandteil für Steuerungs- und Überwachungslogik in 4diac-basierten Automatisierungssystemen. Die Verwendung dieses spezialisierten Blocks anstelle einer Kaskade von Grundgattern verbessert die Lesbarkeit und Wartbarkeit des Applikationscodes.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

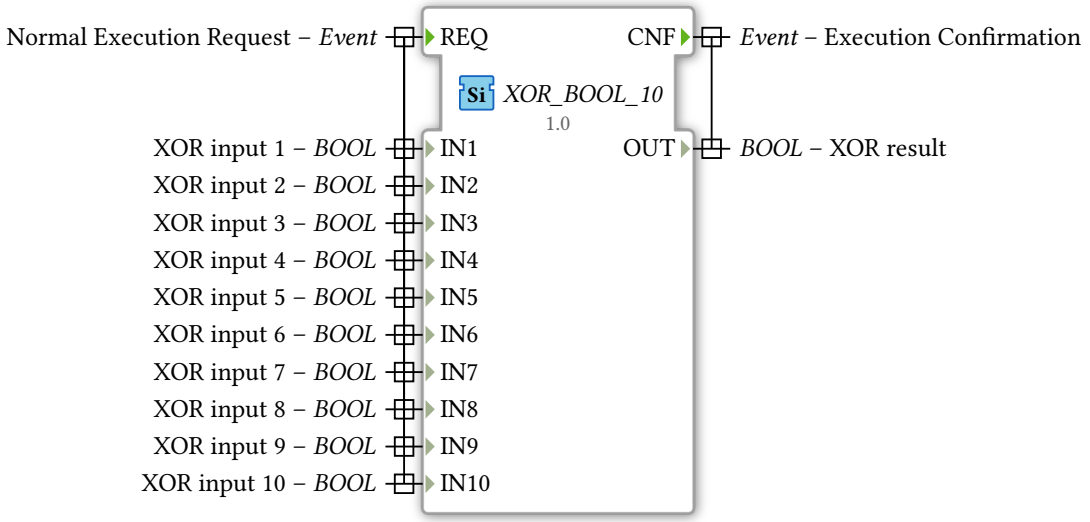
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

XOR_BOOL_10

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `XOR_BOOL_10` ist ein generischer Baustein zur Berechnung der logischen Exklusiv-ODER-Verknüpfung (XOR) über bis zu zehn boolesche Eingangssignale. Er folgt dem IEC 61131-3 Standard und ist für den Einsatz in der 4diac-IDE konzipiert. Der Baustein wertet bei jeder Anforderung alle verbundenen Eingänge aus und liefert das entsprechende Ergebnis.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Dieses Ereignis löst die Berechnung der XOR-Funktion aus. Es ist mit allen zehn Dateneingängen (`IN1` bis `IN10`) verknüpft.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Dieses Ereignis signalisiert den Abschluss der Berechnung. Es wird zusammen mit dem berechneten Ergebnis am Datenausgang `OUT` ausgegeben.

Daten-Eingänge

- **IN1 (XOR input 1):** Boolescher Eingang 1.
- **IN2 (XOR input 2):** Boolescher Eingang 2.
- **IN3 (XOR input 3):** Boolescher Eingang 3.
- **IN4 (XOR input 4):** Boolescher Eingang 4.
- **IN5 (XOR input 5):** Boolescher Eingang 5.
- **IN6 (XOR input 6):** Boolescher Eingang 6.
- **IN7 (XOR input 7):** Boolescher Eingang 7.
- **IN8 (XOR input 8):** Boolescher Eingang 8.
- **IN9 (XOR input 9):** Boolescher Eingang 9.
- **IN10 (XOR input 10):** Boolescher Eingang 10.

Daten-Ausgänge

- **OUT (XOR result):** Boolesches Ergebnis der XOR-Verknüpfung aller aktiven Eingänge.

Adapter

Dieser Funktionsblock verfügt über keine Adapter-Schnittstellen.

Funktionsweise

Bei Empfang des `REQ`-Ereignisses liest der Baustein die Werte aller zehn booleschen Dateneingänge (`IN1`…`IN10`) ein. Anschließend berechnet er das logische XOR über alle diese Eingänge. Die XOR-Funktion liefert genau dann ein `TRUE`-Signal (1) am Ausgang `OUT`, wenn eine ungerade Anzahl der Eingänge `TRUE` ist. Sind null, zwei, vier, sechs, acht oder zehn Eingänge `TRUE`, ist das Ergebnis `FALSE`. Nach der Berechnung wird das `CNF`-Ereignis zusammen mit dem aktuellen Ergebniswert ausgelöst.

Technische Besonderheiten

- **Generischer Baustein:** Der Block ist als generischer Baustein gekennzeichnet (`GEN_XOR`). Dies ermöglicht eine flexible Wiederverwendung und eventuelle Spezialisierung.
- **Feste Eingangsanzahl:** Im Gegensatz zu Bausteinen mit variabler Eingangsanzahl bietet dieser Block genau zehn fest definierte Eingänge. Nicht benötigte Eingänge sollten auf einen definierten Logikpegel (z.B. `FALSE`) gesetzt werden.
- **Ereignisgesteuerte Ausführung:** Die Berechnung erfolgt ausschließlich ereignisgesteuert durch den `REQ`-Eingang. Es gibt keine zyklische Ausführung.

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand im Sinne eines Speichers. Sein Verhalten ist rein kombinatorisch und von den aktuellen Eingangswerten abhängig. Der einzige „Zustand“ ist das Warten auf ein `REQ`-Ereignis. Nach dessen Empfang wird sofort berechnet und das `CNF`-Ereignis gesendet.

Anwendungsszenarien

- **Paritätsprüfung:** Erkennung, ob eine ungerade Anzahl von Signalen (z.B. Fehlermeldungen, Schalterpositionen) aktiv ist.
- **Steuerungslogik:** Implementierung von speziellen Verknüpfungslogiken in Ablaufsteuerungen, wo eine Zustandsänderung nur bei einem ungeraden Aktivierungsmuster erfolgen soll.
- **Codierung/Decodierung:** Einfache kryptographische oder Codierungsaufgaben, die auf der XOR-Operation basieren.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Standard XOR-Blöcke (z.B., XOR mit 2 Eingängen):** Dieser Block erweitert die klassische 2-Eingang-XOR-Funktion auf bis zu zehn Eingänge in einem einzigen Baustein, was die Übersichtlichkeit bei vielen Signalen erhöht. Siehe: XOR_10
- **Bausteine mit variabler Eingangsanzahl:** Einige Bibliotheken bieten XOR-Blöcke an, denen dynamisch Eingänge hinzugefügt werden können. `XOR_BOOL_10` hat eine feste, dafür explizit deklarierte Schnittstelle, was die Code-Analyse vereinfachen kann.
- **Kombinatorische Logik in SFC/ST:** Die gleiche Funktion könnte auch in Structured Text (ST) mit einem Ausdruck wie `OUT := IN1 XOR IN2 XOR ... XOR IN10;` implementiert werden. Der Vorteil des FBs liegt in der klaren Kapselung und der ereignisgesteuerten Schnittstelle, die besser in FBNetzwerke integrierbar ist.

Fazit

Der `XOR_BOOL_10` ist ein spezialisierter und nützlicher Funktionsblock für Anwendungen, die die XOR-Verknüpfung über mehr als zwei Signale erfordern. Seine feste, gut dokumentierte Schnittstelle mit zehn Eingängen und das ereignisgesteuerte IEC 61499-Modell machen ihn zu einem zuverlässigen Baustein für die Implementierung von Paritätsprüfungen und speziellen Steuerlogiken. Für einfachere Verknüpfungen mit weniger Eingängen oder für maximale Flexibilität sollten jedoch alternative Implementierungen in Betracht gezogen werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

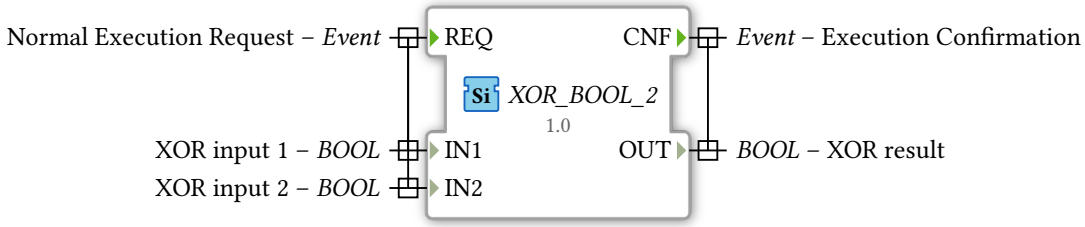
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

XOR_BOOL_2

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `XOR_BOOL_2` ist ein Standardbaustein zur Berechnung der logischen Exklusiv-ODER-Verknüpfung (XOR) zweier boolescher Eingangssignale. Er gehört zur Kategorie der bitweisen Operatoren und ist als generischer Funktionsblock implementiert, der in Steuerungsanwendungen nach IEC 61131-3 eingesetzt wird. Der Baustein wartet auf eine Ausführungsanforderung, berechnet das Ergebnis und gibt dieses zusammen mit einer Bestätigung zurück.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normale Ausführungsanforderung. Löst die Berechnung der XOR-Verknüpfung aus. Die mit diesem Ereignis verknüpften Daten (`IN1`, `IN2`) werden bei der Verarbeitung gelesen.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Dieses Ereignis wird nach der Berechnung generiert und signalisiert die Verfügbarkeit des Ergebnisses. Es ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN1**: Boolescher Eingang 1 für die XOR-Operation.
- **IN2**: Boolescher Eingang 2 für die XOR-Operation.

Daten-Ausgänge

- **OUT**: Boolesches Ergebnis der XOR-Verknüpfung von `IN1` und `IN2`.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Der `XOR_BOOL_2`-Block arbeitet ereignisgesteuert. Bei Eintreffen des Ereignisses `REQ` werden die aktuellen Werte an den Daten-Eingängen `IN1` und `IN2` gelesen. Anschließend wird die logische XOR-Operation (`OUT = (IN1 AND NOT IN2) OR (NOT IN1 AND IN2)`) durchgeführt. Das berechnete Ergebnis wird am Datenausgang `OUT` ausgegeben und gleichzeitig wird das Bestätigungsereignis `CNF` aktiviert, um den Abschluss der Operation an nachfolgende Blöcke zu signalisieren.

Die Wahrheitstabelle der Operation ist:

IN1	IN2	OUT
0	0	0
0	1	1
1	0	1
1	1	0

Technische Besonderheiten

- **Generischer Baustein**: Der Block ist mit dem generischen Klassennamen `'GEN_XOR'` gekennzeichnet, was seine Wiederverwendbarkeit und mögliche Integration in übergeordnete Strukturen erleichtert.
- **Ereignisgesteuerte Ausführung**: Die Funktionalität ist strikt an die Eingangsereignisse gekoppelt, was eine deterministische und energieeffiziente Abarbeitung in Echtzeitsystemen ermöglicht.
- **Einfache Schnittstelle**: Der Block besitzt eine minimale und intuitive Schnittstelle, die der klassischen, in der Digitaltechnik üblichen XOR-Funktion entspricht.

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand im Sinne eines Speichers. Sein Verhalten ist kombinatorisch und ausschließlich von den aktuellen Eingangswerten bei Eintreffen des `REQ`-Ereignisses abhängig. Der Block verharrt nach der Ausgabe von `CNF` in einem wartenden Zustand, bis das nächste `REQ`-Ereignis eintrifft.

Anwendungsszenarien

- **Vergleich und Änderungserkennung**: Erkennung, wenn sich genau einer von zwei Zuständen (z.B. zwei Endschalter) geändert hat.
- **Paritätsprüfung**: Einfache Prüfung auf eine ungerade Anzahl von `TRUE`-Signalen bei zwei Eingängen.
- **Steuerungslogik**: Implementierung von speziellen Verzweigungen in Ablaufsteuerungen, bei denen eine Entscheidung getroffen werden muss, wenn die Eingänge unterschiedlich sind.
- **Sicherheitskritische Schaltungen**: Erzeugung von Signalen, die nur aktiv sind, wenn sich zwei überwachte Größen widersprechen (z.B. in Not-Halt-Schaltungen).

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen logischen Grundfunktionsblöcken wie `AND_BOOL_2` oder `OR_BOOL_2` ist `XOR_BOOL_2` der einzige, der ein `TRUE`-Ergebnis liefert, wenn die Eingänge *unterschiedlich* sind. Blöcke wie `NOT` invertieren nur einen einzelnen Eingang. Der `XOR_BOOL_2` kann als Spezialfall eines Ungleichheitsvergleichs (`NE_BOOL`) für zwei boolesche Werte betrachtet werden. Siehe: XOR_2

Fazit

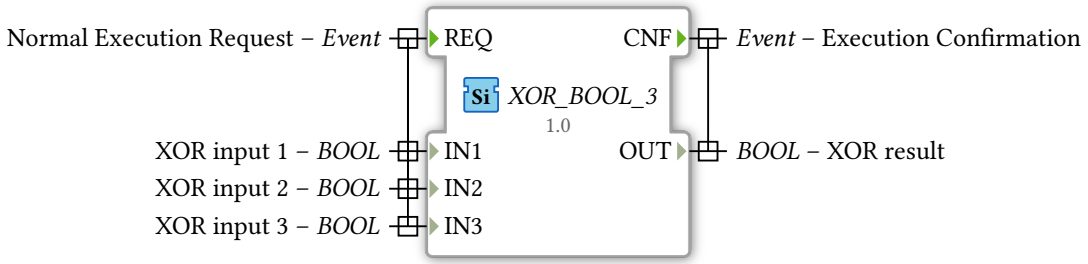
Der `XOR_BOOL_2` ist ein grundlegender, zuverlässiger und einfach zu verwendender Baustein für die logische Verknüpfung in IEC 61131-3-basierten Steuerungsapplikationen. Seine ereignisgesteuerte, zustandslose Architektur macht ihn besonders geeignet für die Integration in komplexe, deterministische Steuerungsnetzwerke. Durch seine klare Funktionsweise und minimale Schnittstelle ist er ein unverzichtbares Werkzeug für jeden Steuerungsentwickler.

XOR_BOOL_3

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `XOR_BOOL_3` ist ein standardisierter Baustein zur Berechnung der logischen Exklusiv-ODER-Verknüpfung (XOR) für drei boolesche Eingangswerte. Er ist als generischer Baustein implementiert und führt seine Operation bei jedem eingehenden Ausführungsereignis durch.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Normal Execution Request): Löst die Berechnung der XOR-Funktion aus. Die mit diesem Ereignis verbundenen Daten (`IN1`, `IN2`, `IN3`) werden gelesen und verarbeitet.

Ereignis-Ausgänge

- **CNF** (Execution Confirmation): Wird nach Abschluss der Berechnung ausgelöst. Dieses Ereignis liefert das berechnete Ergebnis über die zugehörige Datenausgabe `OUT`.

Daten-Eingänge

- **IN1** (BOOL): Erster boolescher Operand für die XOR-Operation.
- **IN2** (BOOL): Zweiter boolescher Operand für die XOR-Operation.
- **IN3** (BOOL): Dritter boolescher Operand für die XOR-Operation.

Daten-Ausgänge

- **OUT** (BOOL): Das Ergebnis der dreifachen XOR-Verknüpfung der Eingänge `IN1`, `IN2` und `IN3`.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Der Block arbeitet ereignisgesteuert. Bei jedem Eintreffen des Ereignisses `REQ` werden die aktuellen Werte der drei booleschen Eingänge `IN1`, `IN2` und `IN3` gelesen. Anschließend wird die logische XOR-Verknüpfung für alle drei Eingänge berechnet. Das Ergebnis wird am Datenausgang `OUT` ausgegeben und gleichzeitig wird das Bestätigungsereignis `CNF` aktiviert, um den Abschluss der Operation zu signalisieren.

Die logische Funktion entspricht: `OUT = (IN1 XOR IN2) XOR IN3`. Für drei Eingänge ist das Ergebnis genau dann `TRUE` (1), wenn eine ungerade Anzahl der Eingänge `TRUE` ist (also 1 oder 3 Eingänge).

Technische Besonderheiten

- **Generischer Baustein:** Der Block ist mit dem Attribut `eclipse4diac::core::GenericClassName` als generischer Baustein (`'GEN_XOR'`) gekennzeichnet. Dies ermöglicht eine potenzielle Wiederverwendung der Logik für andere Datentypen, die die XOR-Operation unterstützen.
- **Standardkonformität:** Der Block ist gemäß der Norm IEC 61131-3 als „standard boolean function“ klassifiziert.
- **Paketzuordnung:** Der Baustein ist dem Paket `iec61131::bitwiseOperators` zugeordnet.

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand im Sinne eines Speichers. Sein Verhalten ist rein kombinatorisch und ausschließlich von den aktuellen Eingangswerten bei Eintreffen des `REQ`-Ereignisses abhängig. Der einzige „Zustand“ ist der idle-Zustand, in dem der Block auf das nächste `REQ`-Ereignis wartet.

Anwendungsszenarien

- **Paritätsprüfung:** Erkennung, ob eine ungerade Anzahl von drei booleschen Signalen aktiv ist.
- **Steuerungslogik:** In speziellen Steuerungsalgorithmen, die eine „entweder-oder“-Logik über mehr als zwei Bedingungen benötigen.
- **Verschlüsselung und Codierung:** Als grundlegende Komponente in einfachen kryptografischen oder fehlererkennenden Schaltungen (z.B. für Paritätsbits).
- **Signalvergleich:** Feststellen, ob sich ein drittes Signal von der Kombination zweier anderer Signale unterscheidet.

⚖️ Vergleich mit ähnlichen Bausteinen

- **XOR (2 Eingänge):** Der Standard-XOR-Baustein mit nur zwei Eingängen. `XOR_BOOL_3` erweitert diese Funktion auf drei Operanden. Siehe: XOR_3
- **ODER / UND (3 Eingänge):** Diese Blöcke berechnen die logische ODER- bzw. UND-Verknüpfung aller Eingänge. Die XOR-Logik ist spezieller und unterscheidet sich insbesondere im Fall, wenn alle drei Eingänge `TRUE` sind (bei ODER=TRUE, bei UND=TRUE, bei XOR=TRUE).
- **GEN_XOR:** Dies ist die generische Basis, von der `XOR_BOOL_3` als spezifische Instanz für den Datentyp `BOOL` abgeleitet ist.

Fazit

Der `XOR_BOOL_3` Funktionsblock ist ein spezialisierter, normkonformer Baustein zur zuverlässigen Berechnung der dreifachen Exklusiv-ODER-Verknüpfung. Seine klare, ereignisgesteuerte Schnittstelle und die deterministische Funktionsweise machen ihn ideal für den Einsatz in Steuerungsanwendungen, die diese spezielle boolesche Logik erfordern. Als Instanz eines generischen Bausteins profitiert er zudem von den Wartungs- und Konsistenzvorteilen eines generischen Designs.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

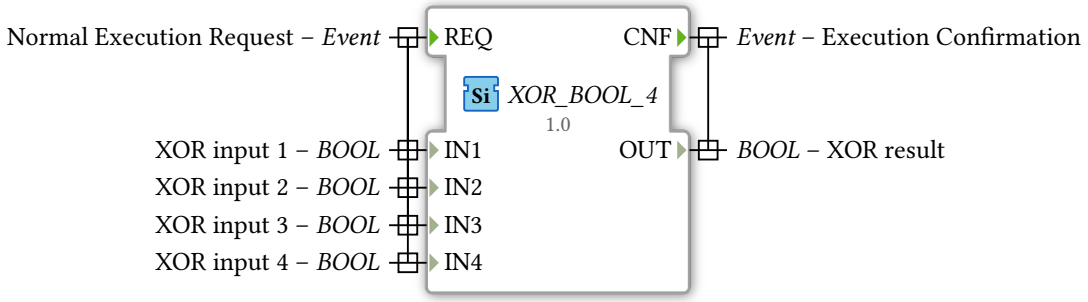
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

XOR_BOOL_4

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `XOR_BOOL_4` ist ein Standardbaustein zur Berechnung der logischen Exklusiv-ODER-Verknüpfung (XOR) für bis zu vier boolesche Eingangswerte. Er folgt dem IEC 61131-3 Standard und ist als generischer Baustein implementiert, der in Steuerungsanwendungen zur kombinatorischen Logikverarbeitung eingesetzt wird.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Normal Execution Request): Dieses Ereignis löst die Berechnung der XOR-Verknüpfung aus. Es ist mit allen vier Dateneingängen (`IN1` bis `IN4`) verknüpft.

Ereignis-Ausgänge

- **CNF** (Execution Confirmation): Dieses Ereignis signalisiert den Abschluss der Berechnung. Es wird zusammen mit dem berechneten Ergebnis am Datenausgang `OUT` ausgegeben.

Daten-Eingänge

- **IN1** (BOOL): XOR-Eingang 1.
- **IN2** (BOOL): XOR-Eingang 2.
- **IN3** (BOOL): XOR-Eingang 3.
- **IN4** (BOOL): XOR-Eingang 4.

Daten-Ausgänge

- **OUT** (BOOL): Ergebnis der XOR-Verknüpfung aller aktiven Eingänge.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei Eintreffen des Ereignisses `REQ` liest der Baustein die Werte aller vier booleschen Eingänge (`IN1`, `IN2`, `IN3`, `IN4`). Anschließend berechnet er das logische Exklusiv-ODER über diese Werte. Das Ergebnis `OUT` ist genau dann `TRUE` (1), wenn eine ungerade Anzahl der Eingänge den Wert `TRUE` aufweist. Ist die Anzahl der `TRUE`-Eingänge gerade (0, 2 oder 4), ist das Ergebnis `FALSE` (0). Nach der Berechnung wird das Bestätigungsereignis `CNF` zusammen mit dem Ergebniswert ausgegeben.

Technische Besonderheiten

- **Generischer Baustein:** Der Block ist mit dem Attribut `eclipse4diac::core::GenericClassName` als `'GEN_XOR'` gekennzeichnet. Dies ermöglicht eine flexible Instanziierung und Wiederverwendung innerhalb der 4diac-IDE.
- **Ereignisgesteuerte Ausführung:** Die Funktionalität ist streng ereignisgesteuert. Eine Berechnung findet nur bei einem eingehenden `REQ`-Ereignis statt.
- **Vier Eingänge:** Im Gegensatz zu einem einfachen 2-Eingang-XOR bietet dieser Baustein die direkte Verarbeitung von bis zu vier Signalen ohne zusätzliche Verschachtelung.

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand (speicherfrei, kombinatorisch). Sein Verhalten ist bei jedem Aufruf ausschließlich von den aktuellen Werten an den Dateneingängen abhängig.

Anwendungsszenarien

- **Paritätsprüfung:** Erkennung, ob eine ungerade Anzahl von Bedingungen (z.B. Fehlermeldungen, Grenzwertüberschreitungen) in einer Gruppe von vier Signalen aktiv ist.
- **Steuerungslogik:** Implementierung von Sicherheitsschaltungen, bei denen genau eine von mehreren möglichen Eingangskombinationen eine Aktion auslösen darf.
- **Verschachtelte Logikvereinfachung:** Ersetzung mehrerer hintereinandergeschalteter 2-Eingang-XOR-Bausteine durch einen einzigen `XOR_BOOL_4`-Block für bessere Übersichtlichkeit im Funktionsplan (FBD).

⚖️ Vergleich mit ähnlichen Bausteinen

- **Standard XOR (2 Eingänge):** Ein klassischer XOR-Baustein mit nur zwei Eingängen. Der `XOR_BOOL_4` konsolidiert die Funktion mehrerer solcher Blöcke. Siehe: XOR_4
- **OR_BOOL_4 / AND_BOOL_4:** Diese Blöcke berechnen die logische ODER- bzw. UND-Verknüpfung über vier Eingänge. Die XOR-Verknüpfung unterscheidet sich fundamental, da sie den Fall, dass alle Eingänge `TRUE` sind, als `FALSE` bewertet.
- **Parity-Check Blöcke:** Spezialisierte Blöcke zur Paritätsberechnung existieren oft für Datenworte (BYTE, WORD). `XOR_BOOL_4` bietet diese Funktion für einzelne Bits/Bool-Werte.

Fazit

Der `XOR_BOOL_4` ist ein nützlicher und effizienter Standardbaustein für die Implementierung von Exklusiv-ODER-Logik mit bis zu vier Eingängen in 4diac-basierten Steuerungsapplikationen. Seine ereignisgesteuerte, zustandslose Architektur und die direkte Unterstützung mehrerer Eingänge machen ihn zu einer praktischen Wahl für Paritätschecks und spezielle kombinatorische Steuerungsaufgaben.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

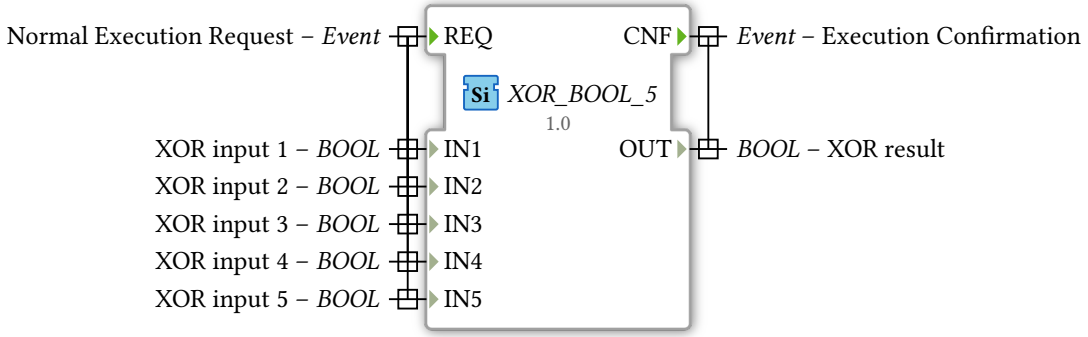
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

XOR_BOOL_5

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `XOR_BOOL_5` ist ein Standardbaustein zur Berechnung der logischen Exklusiv-ODER-Verknüpfung (XOR) für fünf boolesche Eingangswerte. Er folgt dem IEC 61131-3 Standard und ist als generischer Baustein implementiert. Die Operation wird bei jedem eingehenden Ausführungsereignis durchgeführt und das Ergebnis über ein Bestätigungsereignis ausgegeben.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Löst die Berechnung der XOR-Verknüpfung aus. Das Ereignis ist mit allen fünf Dateneingängen (`IN1` bis `IN5`) verknüpft.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Signalisiert den Abschluss der Berechnung. Dieses Ereignis ist mit dem Datenausgang `OUT` verknüpft und liefert das Ergebnis.

Daten-Eingänge

- **IN1 (XOR input 1):** Erster boolescher Operand (BOOL).
- **IN2 (XOR input 2):** Zweiter boolescher Operand (BOOL).
- **IN3 (XOR input 3):** Dritter boolescher Operand (BOOL).
- **IN4 (XOR input 4):** Vierter boolescher Operand (BOOL).
- **IN5 (XOR input 5):** Fünfter boolescher Operand (BOOL).

Daten-Ausgänge

- **OUT (XOR result):** Das boolesche Ergebnis der XOR-Verknüpfung aller fünf Eingänge (BOOL).

Adapter

Dieser Funktionsblock verfügt über keine Adapter-Schnittstellen.

Funktionsweise

Bei jedem Eintreffen des Ereignisses `REQ` führt der Baustein die logische XOR-Operation auf seinen fünf booleschen Eingängen `IN1` bis `IN5` durch. Das Ergebnis wird an den Ausgang `OUT` gelegt und gleichzeitig wird das Bestätigungsereignis `CNF` ausgelöst.

Die logische Funktion ist definiert als: `OUT = IN1 XOR IN2 XOR IN3 XOR IN4 XOR IN5`. Das Ergebnis ist genau dann `TRUE` (1), wenn eine ungerade Anzahl der Eingänge den Wert `TRUE` (1) aufweist. Bei einer geraden Anzahl von `TRUE`-Eingängen ist das Ergebnis `FALSE` (0).

Technische Besonderheiten

- **Generischer Baustein:** Der Block ist durch das Attribut `eclipse4diac::core::GenericClassName` als generische Implementierung (`'GEN_XOR'`) gekennzeichnet. Dies ermöglicht eine flexible Wiederverwendung der Kernlogik.
- **Ereignisgesteuert:** Die Berechnung erfolgt ausschließlich ereignisgesteuert durch `REQ` und nicht zyklisch.
- **Fünf Eingänge:** Im Gegensatz zu Standard-XOR-Bausteinen mit typischerweise zwei Eingängen bietet dieser Block fünf fest verdrahtete Eingänge, was die Verkettung mehrerer Blöcke für eine höhere Anzahl von Operanden erspart.

Zustandsübersicht

Der Baustein besitzt keinen internen Zustand (speicherfrei). Seine Ausgabe hängt ausschließlich von den aktuellen Werten der Eingänge zum Zeitpunkt des `REQ`-Ereignisses ab.

Anwendungsszenarien

- **Paritätsprüfung:** Überprüfung, ob eine ungerade Anzahl von Bedingungen (z.B. Fehlersignalen, Grenzwertüberschreitungen) in einer Gruppe von fünf Signalen aktiv ist.
- **Steuerungslogik:** Implementierung von speziellen Verknüpfungslogiken in SPS-Programmen, bei denen genau eine oder drei oder fünf von fünf möglichen Einzelbedingungen erfüllt sein müssen.
- **Signalvergleich:** Feststellen von Ungleichheit in einer Gruppe von fünf binären Signalen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Standard XOR (2 Eingänge):** Bibliotheksbausteine bieten oft nur die XOR-Verknüpfung für zwei Operanden. Für fünf Operanden müssten mehrere dieser Blöcke verkettet werden (z.B. `((IN1 XOR IN2) XOR (IN3 XOR IN4)) XOR IN5`). `XOR_BOOL_5` fasst diese Logik in einem einzigen, übersichtlichen Block zusammen. Siehe: XOR_5
- **ODER / UND Bausteine:** Während ODER (`OR`) `TRUE` bei mindestens einem `TRUE`-Eingang liefert und UND (`AND`) nur bei allen `TRUE`-Eingängen, reagiert XOR spezifisch auf eine ungerade Anzahl von `TRUE`-Werten.
- **Generische XOR-Blöcke:** Es könnten generische Blöcke existieren, die eine variable Anzahl von Eingängen über Arrays oder wiederholte Eingänge handhaben. `XOR_BOOL_5` ist eine fest konfigurierte, spezifische Instanz für genau fünf Eingänge.

Fazit

Der `XOR_BOOL_5` Funktionsblock ist ein spezialisierter und effizienter Baustein zur Berechnung der Exklusiv-ODER-Verknüpfung über genau fünf boolesche Signale. Seine ereignisgesteuerte, zustandslose Arbeitsweise und die klare Schnittstelle machen ihn zu einem zuverlässigen Werkzeug für Steuerungsan-

wendungen, die diese spezielle Logikfunktion benötigen, insbesondere im Bereich der Paritätskontrolle und komplexen Bedingungsverknüpfung.

Passende Themen-Unterseiten auf ms-muc-docs.de

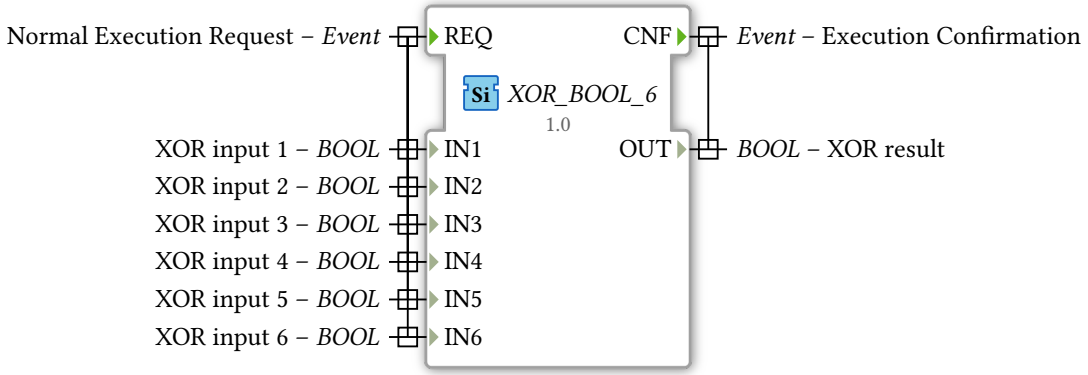
- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

XOR_BOOL_6

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `XOR_BOOL_6` ist ein generischer Baustein zur Berechnung der logischen Exklusiv-ODER-Verknüpfung (XOR) für bis zu sechs boolesche Eingangswerte. Er folgt dem IEC 61131-3 Standard und ist für den Einsatz in Steuerungs- und Automatisierungsanwendungen konzipiert. Der Baustein wartet auf ein Auslöseereignis, berechnet das Ergebnis der XOR-Operation über alle aktiven Eingänge und gibt dieses zusammen mit einem Bestätigungsereignis aus.



Schnittstellenstruktur

Der FB besitzt einen ereignisgesteuerten Ein-/Ausgangsmechanismus mit sechs booleschen Dateneingängen und einem booleschen Datenausgang.

Ereignis-Eingänge

- **REQ** (Normal Execution Request): Dieses Ereignis löst die Berechnung aus. Beim Eintreffen von `REQ` werden die Werte an den Eingängen `IN1` bis `IN6` gelesen und die XOR-Operation durchgeführt.

Ereignis-Ausgänge

- **CNF** (Execution Confirmation): Dieses Ereignis wird nach Abschluss der Berechnung ausgelöst und signalisiert, dass das Ergebnis am Ausgang `OUT` bereitsteht.

Daten-Eingänge

- **IN1** (BOOL): XOR-Eingang 1.
- **IN2** (BOOL): XOR-Eingang 2.
- **IN3** (BOOL): XOR-Eingang 3.
- **IN4** (BOOL): XOR-Eingang 4.
- **IN5** (BOOL): XOR-Eingang 5.
- **IN6** (BOOL): XOR-Eingang 6.

Daten-Ausgänge

- **OUT** (BOOL): Das Ergebnis der XOR-Verknüpfung aller sechs Eingänge.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei Auslösung durch das `REQ`-Ereignis berechnet der Baustein den logischen XOR-Wert über alle sechs Eingänge `IN1` bis `IN6`. Die XOR-Operation ergibt genau dann `TRUE` (1), wenn eine ungerade Anzahl der

Eingänge den Wert TRUE hat. Sind null, zwei, vier oder sechs Eingänge TRUE, ist das Ergebnis FALSE (0). Formal entspricht dies der Berechnung: OUT = IN1 XOR IN2 XOR IN3 XOR IN4 XOR IN5 XOR IN6. Nach der Berechnung wird das Ergebnis am Datenausgang OUT ausgegeben und gleichzeitig das Bestätigungsereignis CNF aktiviert.

Technische Besonderheiten

- **Generischer Baustein:** Der FB ist als generischer Baustein gekennzeichnet (GEN_XOR). Dies ermöglicht es, ihn in verschiedenen Kontexten wiederzuverwenden oder als Basis für spezialisierte Versionen zu nutzen.
- **Feste Eingangsanzahl:** Im Gegensatz zu Bausteinen mit variabler Eingangsanzahl besitzt XOR_BOOL_6 genau sechs fest definierte Eingänge. Nicht benötigte Eingänge sollten auf einen definierten Logikpegel (z.B. FALSE) gesetzt werden.
- **Package-Zuordnung:** Der Baustein ist dem Package iec61131::bitwiseOperators zugeordnet, was seine Einordnung als bitweise/logische Operation unterstreicht.

Zustandsübersicht

Der Funktionsblock ist zustandslos (kombinatorisch). Er besitzt keinen internen Speicher. Die Ausgabe OUT ist eine reine Funktion der aktuellen Eingangswerte zum Zeitpunkt des REQ-Ereignisses. Es gibt keine Abhängigkeit von vorherigen Berechnungen oder Zuständen.

Anwendungsszenarien

- **Paritätsprüfung:** Erkennung, ob eine ungerade Anzahl von Bedingungen in einer Gruppe von sechs Signalen aktiv ist.
- **Sicherheitskritische Logik:** In Schaltungen, wo ein Zustand nur dann aktiv sein darf, wenn exklusive Bedingungen vorliegen (z.B. Auswahl zwischen verschiedenen, sich gegenseitig ausschließenden Betriebsmodi).
- **Verknüpfung von Statusbits:** Kombination mehrerer Status- oder Fehlerflags zu einem zusammenfassenden „Ungerade-Paritäts“-Signal.

⚖️ Vergleich mit ähnlichen Bausteinen

- **XOR (2 Eingänge):** Der Standard-XOR-Baustein mit nur zwei Eingängen. XOR_BOOL_6 erweitert diese Funktionalität auf sechs Eingänge ohne die Notwendigkeit, mehrere 2-Eingangs-XOR-Blöcke zu verschalten. Siehe: XOR_6
- **AND_BOOL_6 / OR_BOOL_6:** Ähnliche Blöcke für die logischen UND- bzw. ODER-Verknüpfungen mit sechs Eingängen. Während diese Blöcke auf „Alle“ bzw. „Mindestens eines“ testen, testet XOR_BOOL_6 auf „Ungerade Anzahl“.
- **Bausteine mit variabler Eingangsanzahl:** Einige Implementierungen bieten möglicherweise einen XOR-Block an, dem eine beliebige Anzahl von Eingängen zugewiesen werden kann. XOR_BOOL_6 bietet hingegen eine explizite und feste Schnittstelle.

Fazit

Der XOR_BOOL_6 ist ein spezialisierter und einfach anzuwendender Funktionsblock für die exklusive ODER-Verknüpfung von bis zu sechs booleschen Signalen. Seine fest definierte Schnittstelle und die ereignisgesteuerte Abarbeitung machen ihn gut integrierbar in 4diac IDE-basierte Applikationen. Er ist besonders nützlich in Szenarien, die eine Paritätsprüfung oder die Auswertung exklusiver Bedingungen über eine kleine, feste Anzahl von Signalen erfordern. Für Anwendungen mit einer variablen oder deutlich höheren Anzahl an Eingängen sollten alternative Lösungen in Betracht gezogen werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

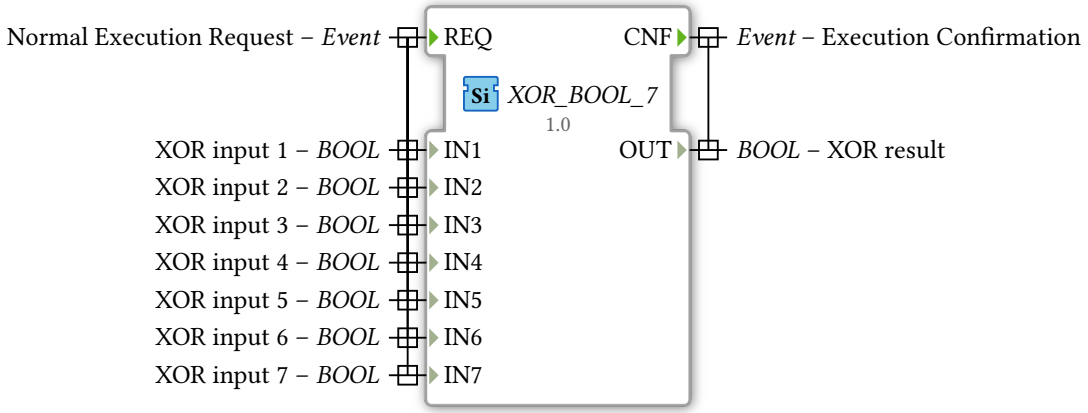
XOR_BOOL_7

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock XOR_BOOL_7 ist ein generischer Baustein zur Berechnung der logischen Exklusiv-ODER (XOR) Verknüpfung. Er verarbeitet sieben boolesche Eingangssignale und liefert das Ergebnis der XOR-Operation als ein boolesches Ausgangssignal. Der Block folgt dem IEC 61131-3 Standard und ist für

den Einsatz in Steuerungsanwendungen konzipiert, bei denen eine ungerade Paritätsprüfung oder eine logische Antivalenz über mehrere Signale erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Dieses Ereignis löst die Ausführung des Funktionsblocks aus. Bei seinem Eintreffen werden alle sieben Daten-Eingänge (`IN1` bis `IN7`) ausgelesen und die XOR-Berechnung durchgeführt.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Dieses Ereignis signalisiert den Abschluss der Berechnung. Es wird zusammen mit dem berechneten Datenausgang `OUT` ausgegeben.

Daten-Eingänge

- **IN1 (XOR input 1):** Boolescher Eingang 1.
- **IN2 (XOR input 2):** Boolescher Eingang 2.
- **IN3 (XOR input 3):** Boolescher Eingang 3.
- **IN4 (XOR input 4):** Boolescher Eingang 4.
- **IN5 (XOR input 5):** Boolescher Eingang 5.
- **IN6 (XOR input 6):** Boolescher Eingang 6.
- **IN7 (XOR input 7):** Boolescher Eingang 7.

Daten-Ausgänge

- **OUT (XOR result):** Boolesches Ergebnis der XOR-Verknüpfung aller sieben Eingänge.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei Eintreffen des Ereignisses `REQ` führt der Block die logische XOR-Operation über alle sieben booleschen Eingänge `IN1` bis `IN7` durch. Das Ergebnis `OUT` ist genau dann `TRUE` (1), wenn eine ungerade Anzahl der Eingänge den Wert `TRUE` aufweist. Ist die Anzahl der `TRUE`-Eingänge gerade (0, 2, 4, 6), ist das Ergebnis `FALSE` (0).

Mathematisch ausgedrückt: `OUT = IN1 XOR IN2 XOR IN3 XOR IN4 XOR IN5 XOR IN6 XOR IN7`

Unmittelbar nach der Berechnung wird das Abschlussereignis `CNF` zusammen mit dem gültigen Ergebnis `OUT` ausgelöst.

Technische Besonderheiten

- **Generischer Baustein:** Der Block ist als generischer Funktionsblock (`GEN_XOR`) implementiert, was eine flexible Wiederverwendung in verschiedenen Kontexten ermöglicht.
- **Feste Eingangsanzahl:** Im Gegensatz zu Bausteinen mit variabler Eingangsanzahl besitzt dieser Block exakt sieben fest definierte Eingänge.

- **Ereignisgesteuerte Ausführung:** Die Operation wird ausschließlich durch ein eingehendes Ereignis (REQ) getriggert, was eine deterministische und energieeffiziente Abarbeitung im Echtzeitsystem ermöglicht.

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand im Sinne eines Speichers. Sein Verhalten ist rein kombinatorisch und ausschließlich von den aktuellen Werten an den Eingängen zum Zeitpunkt des REQ-Ereignisses abhängig. Nach der Ausgabe von CNF verharrt der Block in einem wartenden Zustand bis zum nächsten REQ.

Anwendungsszenarien

- **Paritätsprüfung:** Überwachung, ob eine ungerade Anzahl von Sensoren (z.B. Grenztaster, Lichtschranken) in einem bestimmten Zustand (TRUE) ist.
- **Sicherheitskritische Verknüpfungen:** In Sicherheitsschaltungen, wo eine Antivalenz-Bedingung über mehrere Sicherheitseinrichtungen erforderlich ist.
- **Steuerungslogik:** Implementierung von speziellen logischen Verknüpfungen in Ablaufsteuerungen oder Verknüpfungssteuerungen, die eine XOR-Funktion mit genau sieben Operanden benötigen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **XOR (2 Eingänge):** Der Standard-XOR-Baustein verknüpft nur zwei Eingänge. Für eine Verknüpfung von sieben Signalen müsste dieser Block mehrfach verschachtelt werden, was die Lesbarkeit des Programms verringert. XOR_BOOL_7 bietet eine direkte und übersichtliche Lösung. Siehe: XOR_7
- **GEN_XOR (Generisch, n Eingänge):** Der zugrundeliegende generische Baustein GEN_XOR kann mit einer variablen Anzahl von Eingängen instanziiert werden. XOR_BOOL_7 ist eine spezifische, vorkonfigurierte Instanz mit sieben Eingängen, die direkt verwendet werden kann, ohne die Anzahl der Eingänge parametrieren zu müssen.
- **PARITY_CHECK:** Ein reiner Paritätsprüfblock könnte ähnliche Ergebnisse liefern, ist aber konzeptionell oft auf die Fehlererkennung in Datenströmen ausgelegt, während XOR_BOOL_7 als allgemeiner logischer Operator in der Steuerungslogik dient.

Fazit

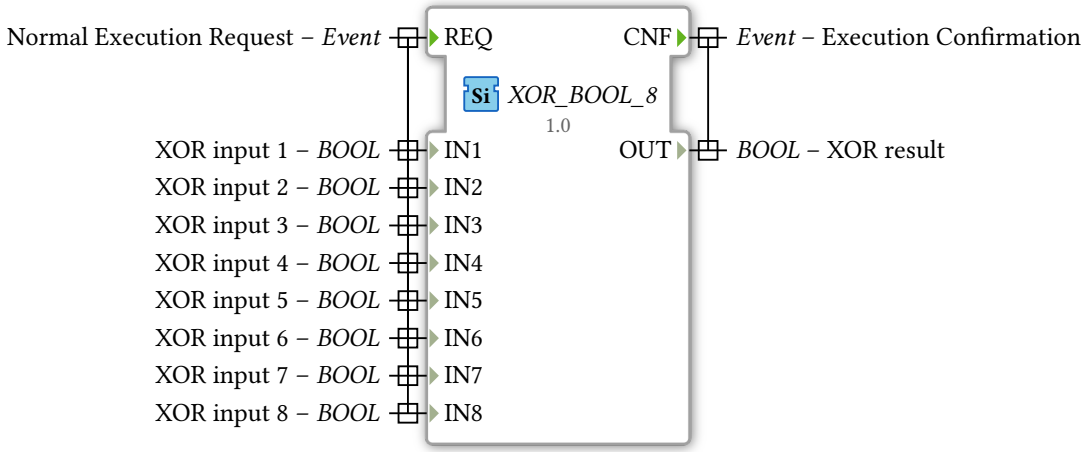
Der XOR_BOOL_7 Funktionsblock ist ein spezialisierter und effizienter Baustein für die logische Antivalenz-Verknüpfung von sieben booleschen Signalen. Durch seine ereignisgesteuerte, zustandslose Arbeitsweise und die klare Schnittstelle eignet er sich ideal für den Einsatz in strukturierten IEC 61131-3-basierten Steuerungsprogrammen, wo eine solche feste Verknüpfung benötigt wird. Er vereinfacht die Programmierung im Vergleich zur Verschachtelung mehrerer Zwei-Eingangs-XOR-Blöcke und bietet eine sofort einsatzbereite, zuverlässige Komponente.

XOR_BOOL_8

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock XOR_BOOL_8 ist ein generischer Baustein zur Berechnung der logischen Exklusiv-ODER-Verknüpfung (XOR) über bis zu acht boolesche Eingangswerte. Er folgt dem IEC 61131-3 Standard und ist für den Einsatz in der 4diac-IDE konzipiert. Der Block führt seine Operation ereignisgesteuert aus und liefert das Ergebnis über einen einzelnen booleschen Ausgang.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Normal Execution Request): Dieses Ereignis löst die Berechnung aus. Es ist mit allen acht Dateneingängen (`IN1` bis `IN8`) verknüpft.

Ereignis-Ausgänge

- **CNF** (Execution Confirmation): Dieses Ereignis signalisiert den Abschluss der Berechnung. Es ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN1** (BOOL): XOR-Eingang 1.
- **IN2** (BOOL): XOR-Eingang 2.
- **IN3** (BOOL): XOR-Eingang 3.
- **IN4** (BOOL): XOR-Eingang 4.
- **IN5** (BOOL): XOR-Eingang 5.
- **IN6** (BOOL): XOR-Eingang 6.
- **IN7** (BOOL): XOR-Eingang 7.
- **IN8** (BOOL): XOR-Eingang 8.

Daten-Ausgänge

- **OUT** (BOOL): Das Ergebnis der XOR-Verknüpfung aller aktiven Eingänge.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Eintreffen des Ereignisses `REQ` liest der Funktionsblock die Werte aller acht booleschen Eingänge (`IN1` bis `IN8`). Anschließend berechnet er das logische Exklusiv-ODER über alle diese Eingangswerte. Das Ergebnis dieser Berechnung wird am Datenausgang `OUT` ausgegeben und gleichzeitig wird das Bestätigungsereignis `CNF` gesetzt, um den Abschluss des Vorgangs zu signalisieren.

Die logische Funktion lässt sich wie folgt beschreiben: `OUT = IN1 XOR IN2 XOR IN3 XOR IN4 XOR IN5 XOR IN6 XOR IN7 XOR IN8`

Ein XOR (Exklusiv-ODER) liefert genau dann `TRUE` (1) als Ergebnis, wenn eine ungerade Anzahl der verknüpften Eingänge `TRUE` (1) ist. Ist die Anzahl der wahren Eingänge gerade, ist das Ergebnis `FALSE` (0).

Technische Besonderheiten

- **Generischer Baustein:** Der Block ist als generischer Baustein (`GEN_XOR`) gekennzeichnet, was seine Wiederverwendbarkeit und mögliche Parametrierung für andere Datentypen andeutet. In dieser konkreten Instanz ist er jedoch auf den Datentyp `BOOL` festgelegt.
- **Ereignisgesteuerte Ausführung:** Die Operation wird nur bei einem eingehenden `REQ`-Ereignis durchgeführt, was eine effiziente und deterministische Ablaufsteuerung ermöglicht.

- **Feste Eingangsanzahl:** Der Block bietet genau acht Eingänge. Nicht benötigte Eingänge sollten auf einen definierten logischen Wert (z.B. `FALSE`) gesetzt werden.

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand im Sinne eines Speichers. Sein Verhalten ist rein kombinatorisch und ereignisgesteuert:

1. **Ruhezustand:** Wartet auf das Eintreffen des `REQ`-Ereignisses.
2. **Berechnungszustand:** Bei `REQ` werden alle Eingänge gelesen und die XOR-Operation durchgeführt.
3. **Ausgabezustand:** Das Ergebnis wird an `OUT` ausgegeben und das `CNF`-Ereignis wird generiert. Anschließend kehrt der Block in den Ruhezustand zurück.

Anwendungsszenarien

- **Paritätsprüfung:** Erkennung, ob eine ungerade Anzahl von Signalen (z.B. Fehlermeldungen, Grenzwertüberschreitungen) in einer Gruppe aktiv ist.
- **Umschaltlogik:** Steuerung, bei der eine Aktion nur dann ausgeführt werden soll, wenn genau eines von mehreren möglichen Steuersignalen aktiv ist (z.B. Wahl zwischen manuellem und automatischem Modus).
- **Verschlüsselungs- und Codierungsgrundfunktionen:** Als Basisbaustein für einfache kryptografische Operationen oder Fehlererkennungscodes.
- **Vergleich von zwei oder mehr Zuständen:** Feststellung, ob sich mindestens ein Signal in einer Gruppe von seinem Sollzustand unterscheidet.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`OR_BOOL_8` / `AND_BOOL_8`:** Diese Blöcke berechnen die logische ODER- bzw. UND-Verknüpfung. Im Gegensatz zum XOR liefert ODER `TRUE`, wenn *mindestens* ein Eingang wahr ist, und UND liefert `TRUE` nur, wenn *alle* Eingänge wahr sind. XOR ist spezifischer und reagiert auf die *Parität* (gerade/ungerade) der wahren Eingänge.
- **`XOR` (mit 2 Eingängen):** Ein Standard-XOR-Block mit nur zwei Eingängen. `XOR_BOOL_8` erweitert diese Funktionalität auf bis zu acht Eingänge, ohne dass mehrere Blöcke verkettet werden müssen. Siehe: XOR_8
- **Kombinatorische Logik in SFC/ST:** Die Funktion könnte auch in Structured Text (ST) direkt programmiert werden (z.B. `OUT := IN1 XOR IN2 XOR ...;`). Der Vorteil des FBs liegt in der standardisierten, wiederverwendbaren und grafisch darstellbaren Schnittstelle innerhalb von FBD-Netzwerken.

Fazit

Der `XOR_BOOL_8` ist ein spezialisierter, zuverlässiger und einfach zu verwendender Funktionsblock für logische XOR-Operationen mit bis zu acht Eingängen. Seine ereignisgesteuerte Architektur macht ihn ideal für die Integration in steuerungstypische Ablaufstrukturen der 4diac-IDE. Er eignet sich besonders für Anwendungen, bei denen die Parität von Signalgruppen von Bedeutung ist, und bietet eine klar definierte Alternative zu den grundlegenden ODER- und UND-Verknüpfungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

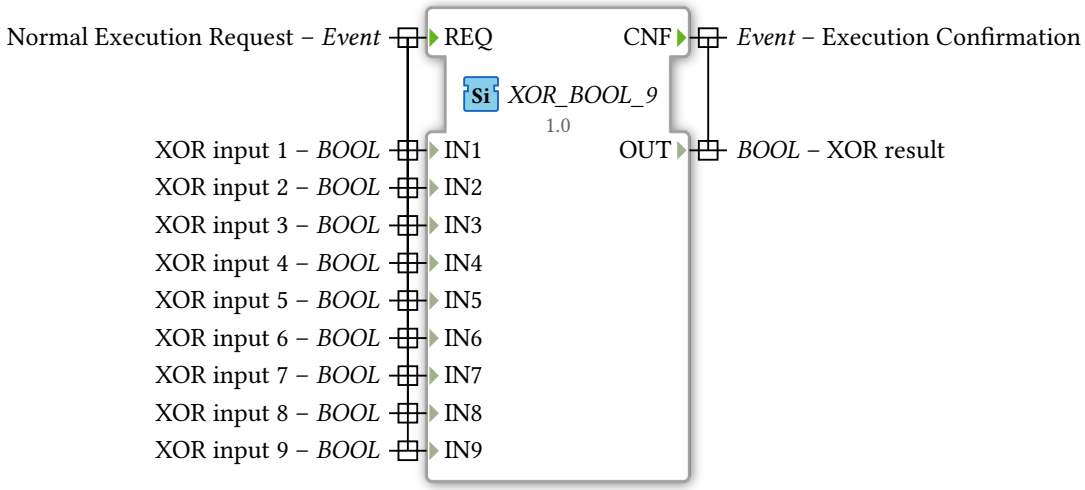
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

XOR_BOOL_9

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `XOR_BOOL_9` ist ein generischer Baustein zur Berechnung der logischen Exklusiv-ODER-Verknüpfung (XOR) für bis zu neun boolesche Eingangssignale. Er realisiert eine n-stellige XOR-Funktion, bei der der Ausgang genau dann `TRUE` ist, wenn eine ungerade Anzahl der aktiven Eingänge `TRUE` ist. Der Baustein folgt dem Standard IEC 61131-3 und ist für den Einsatz in Steuerungsanwendungen konzipiert, die eine Prüfung auf ungerade Parität oder eine spezielle Auswahl- bzw. Überwachungslogik erfordern.



Schnittstellenstruktur

Der Funktionsblock verfügt über ein einfaches ereignisgesteuertes Interface mit einem Anforderungs- und einem Bestätigungsereignis.

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Dieses Ereignis löst die Berechnung der XOR-Funktion aus. Es ist mit allen neun Dateneingängen (`IN1` bis `IN9`) verknüpft.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Dieses Ereignis signalisiert den Abschluss der Berechnung. Es wird zusammen mit dem berechneten Ergebnis am Datenausgang `OUT` ausgegeben.

Daten-Eingänge

- **IN1 (XOR input 1):** Boolescher Eingang 1.
- **IN2 (XOR input 2):** Boolescher Eingang 2.
- **IN3 (XOR input 3):** Boolescher Eingang 3.
- **IN4 (XOR input 4):** Boolescher Eingang 4.
- **IN5 (XOR input 5):** Boolescher Eingang 5.
- **IN6 (XOR input 6):** Boolescher Eingang 6.
- **IN7 (XOR input 7):** Boolescher Eingang 7.
- **IN8 (XOR input 8):** Boolescher Eingang 8.
- **IN9 (XOR input 9):** Boolescher Eingang 9.

Daten-Ausgänge

- **OUT (XOR result):** Boolesches Ergebnis der n-stelligen XOR-Verknüpfung aller aktiven Eingänge.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Eintreffen des Ereignisses `REQ` wird die logische Operation ausgeführt. Der Ausgang `OUT` wird auf `TRUE` gesetzt, wenn die Anzahl der Eingänge mit dem Wert `TRUE` ungerade ist. Ist die Anzahl der `TRUE`-Eingänge gerade (oder null), wird `OUT` auf `FALSE` gesetzt. Unmittelbar nach der Berechnung wird das Bestätigungsereignis `CNF` zusammen mit dem aktuellen Wert von `OUT` ausgegeben.

Die mathematische Beschreibung lautet: `OUT = IN1 XOR IN2 XOR IN3 XOR IN4 XOR IN5 XOR IN6 XOR IN7 XOR IN8 XOR IN9`.

Technische Besonderheiten

- **Generischer Baustein:** Der FB ist als generische Funktion (`GEN_XOR`) gekennzeichnet, was bedeutet, dass er als Basis für die Ableitung spezifischer XOR-Bausteine mit einer festen Anzahl von Eingängen dienen kann.

- **Fest verdrahtete Logik:** Die Verknüpfung erfolgt über alle neun Eingänge. Für Anwendungen mit weniger benötigten Eingängen müssen die nicht genutzten Eingänge auf einen definierten Wert (typischerweise `FALSE`) gesetzt werden.
- **Ereignisgesteuerte Ausführung:** Die Berechnung erfolgt nur bei Auftreten des `REQ`-Ereignisses, was eine ressourcenschonende und deterministische Abarbeitung ermöglicht.

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand im Sinne eines Speichers. Sein Verhalten ist rein kombinatorisch und ausschließlich von den aktuellen Werten an den Dateneingängen zum Zeitpunkt des `REQ`-Ereignisses abhängig. Der Zustand „Berechnung läuft“ ist transient und endet sofort mit der Ausgabe von `CNF`.

Anwendungsszenarien

- **Paritätsprüfung:** Überwachung, ob eine ungerade Anzahl von Sensoren (z.B. Grenzwertüberwachungen, Sicherheitsschaltern) einen Alarmzustand meldet.
- **Auswahl- oder Wechsellogik:** Steuerung, bei der eine Aktion genau dann ausgeführt werden soll, wenn sich der Zustand einer ungeraden Anzahl von Bedingungen geändert hat.
- **Fehlererkennung in redundanten Systemen:** Einfache Plausibilitätskontrolle bei Systemen mit mehreren redundanten Kanälen.
- **Verschlüsselungs- und Codierungsverfahren:** Als grundlegende Komponente in einfachen kryptografischen oder fehlerkorrigierenden Codes.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Standard XOR-Bausteine (z.B., XOR, E_XOR):** Diese haben typischerweise nur zwei Eingänge. `XOR_BOOL_9` erweitert diese Funktion auf bis zu neun Eingänge in einem einzigen Baustein.
- **ODER-Bausteine (OR) / UND-Bausteine (AND):** Liefern ein `TRUE`, wenn mindestens ein bzw. alle Eingänge `TRUE` sind. Die XOR-Logik ist spezifischer (ungerade Anzahl).
- **Paritätsbausteine:** Spezialisierte Blöcke zur Paritätsberechnung, die oft direkt für Datenworte (BYTE, WORD) arbeiten. `XOR_BOOL_9` arbeitet auf einzelnen Booleschen Bits und ist flexibler in der Eingangsanzahl.
- **Kombinatorische Logikbausteine (GEN_AND, GEN_OR):** Ähnliche generische Bausteine für andere logische Grundverknüpfungen.

Fazit

Der `XOR_BOOL_9` ist ein spezialisierter, generischer Funktionsblock, der die logische XOR-Verknüpfung für bis zu neun Signale in einer einzigen, ereignisgesteuerten Komponente bereitstellt. Seine Stärke liegt in der klaren Schnittstelle und der direkten Implementierung der n-stelligen XOR-Funktion, was ihn ideal für Paritätsprüfungen und spezielle Steuerungslogiken macht. Für Anwendungen mit variabler oder sehr großer Eingangsanzahl muss die Logik möglicherweise aus mehreren Basiskomponenten aufgebaut werden.

2.1.12.2 Numerischer Index (iec61131-3-bool)

Hier finden Sie alle Übungen des Arbeitsbereichs `iec61131-3-bool` in aufsteigender Reihenfolge.

F_MUX_32

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **F_MUX_32** ist ein generischer Multiplexer (Auswahlfunktion) gemäß IEC 61131-3. Er ermöglicht die Auswahl eines von 32 Eingangswerten (IN1 … IN32) und gibt diesen am Ausgang OUT weiter. Die Auswahl erfolgt über den ganzzahligen Selektor K. Der Baustein ist ereignisgesteuert: Bei einem Ereignis am Eingang REQ wird der aktuelle Wert von K ausgewertet und der entsprechende Eingangswert übernommen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
REQ	Event	Service Request – löst die Auswahl aus

Ereignis-Ausgänge

Name	Typ	Kommentar
CNF	Event	Confirmation – bestätigt die Durchführung der Auswahl

Daten-Eingänge

Name	Typ	Kommentar
K	ANY_INT	Selektionswert (0 … 31), wählt einen der 32 Eingänge
IN1	ANY	Eingangswert 1
IN2	ANY	Eingangswert 2
…	…	…
IN32	ANY	Eingangswert 32

Daten-Ausgänge

Name	Typ	Kommentar
OUT	ANY	Ausgangswert: IN1 , wenn K=0; IN2, wenn K=1; …; IN32, wenn K=31

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der FB `F_MUX_32` arbeitet ereignisgesteuert:

1. Ein Ereignis am Eingang **REQ** triggert die Verarbeitung.
2. Der Selektor **K** (ganzzahliger Wert) wird ausgewertet.
3. Abhängig vom Wert von K wird der entsprechende Daten-Eingang ausgewählt:
   - K = 0 ⟶ **IN1**
   - K = 1 ⟶ **IN2**
   - …
   - K = 31 ⟶ **IN32**
4. Der ausgewählte Wert wird an den Ausgang **OUT** übergeben.
5. Ein Ereignis am Ausgang **CNF** signalisiert den Abschluss der Operation.

Werte außerhalb des gültigen Bereichs (0 … 31) führen zu undefiniertem Verhalten; der Baustein bietet keine Bereichsprüfung.

Technische Besonderheiten

- **Generischer Datentyp (ANY):** Alle Daten-Eingänge und der Ausgang sind mit `ANY` deklariert. Der Baustein kann daher mit verschiedenen Datentypen (BOOL, INT, REAL, etc.) verwendet werden. Alle beteiligten Eingänge und der Ausgang müssen jedoch denselben konkreten Typ besitzen.
- **Ereignis-Assoziation:** Das Ereignis **REQ** ist mit allen Daten-Eingängen (IN1 … IN32 und K) assoziiert. Bei einem REQ-Ereignis werden alle diese Eingänge gleichzeitig gelesen.
- **Mangelnde Bereichsprüfung:** Der Baustein prüft nicht, ob K im Bereich 0 … 31 liegt. Ein ungültiger Wert kann zu unerwarteten Ergebnissen führen (z. B. Auswahl eines nicht vorhandenen Kanals). Dies muss in der Anwendung sichergestellt werden.
- **Standardkonformität:** Der FB implementiert die `SELECT`-Funktionalität aus IEC 61131-3 und ist als „standard selection function“ klassifiziert.

Zustandsübersicht

Der Baustein besitzt keine explizite Zustandsmaschine in der XML-Definition. Er verhält sich wie eine ereignisgesteuerte Funktion: Nach dem Ereignis **REQ** wird der Ausgang sofort aktualisiert und **CNF** ausgelöst. Es gibt keine internen Zustände.

Anwendungsszenarien

- **Umschaltung zwischen mehreren Sensorwerten** in einer Steuerung (z. B. Temperatur-, Druck- oder Füllstandssensoren).
- **Parametrierbare Konfiguration:** Auswahl unterschiedlicher Betriebsmodi oder Sollwerte über einen Index.
- **Test- und Diagnosefunktionen**, bei denen verschiedene Signalquellen an einen Analysepunkt geschaltet werden.
- **Erweiterung von bestehenden Multiplexer-Lösungen** mit weniger Kanälen auf 32 Eingänge.

Vergleich mit ähnlichen Bausteinen

- **F_MUX_2, F_MUX_3, … F_MUX_n:** Diese Bausteine besitzen eine geringere Anzahl an Eingängen (2, 3, …). `F_MUX_32` deckt den maximalen Bedarf für 32 Kanäle ab. Wird nur eine kleinere Anzahl benötigt, sind kleinere Varianten platzsparender und übersichtlicher.
- **CASE-Anweisung (ST):** Eine `CASE`-Struktur in Structured Text kann das gleiche Verhalten abbilden, ist aber nicht als wiederverwendbarer FB gekapselt. Der FB bietet ereignisgesteuerte Kapselung für grafische und textuelle Programmierung.
- **MUX-Funktion in IEC 61131-3:** Die integrierte `MUX`-Funktion ist typischerweise auf kleinere Anzahlen (z. B. 8) beschränkt. Der FB erweitert dies auf 32 Kanäle.

Fazit

Der **F_MUX_32** ist ein leistungsfähiger und flexibler Multiplexer-Baustein für den Einsatz in der Automatisierungstechnik. Durch seine generische Typisierung und die große Anzahl von 32 Eingängen eignet er sich für vielfältige Auswahlaufgaben. Die einfache Ereignissteuerung und die saubere Trennung von Ereignis- und Datenpfaden machen ihn in IEC 61131-3 Umgebungen gut integrierbar. Anwender sollten jedoch auf die Sicherstellung des gültigen Selektionsbereichs achten, um Fehler zu vermeiden.

2.1.13 isobus

2.1.13.1 🎧 Podcast

- Das Working Set Objekt: Das Gehirn der ISOBUS-Bedienoberfläche verstehen – Von der Norm zur Praxis im ISO-Designer
- ISOBUS Button: Mehr als nur ein Klick – Die Standardisierung der Landtechnik
- ISOBUS Object Pointer: Das Geheimnis dynamischer Displays und effizienter Fehlerdiagnose
- ISOBUS Output Meter: Dynamische Anzeigen meistern – vom Zeiger bis zur Visualisierung auf dem VT
- ISOBUS Skalierung: Wenn der Ackerschlepper-Bildschirm nicht passt – Eine Einführung in ISO 11783-6

2.1.13.2 TC

🎧 Podcast

- E_SWITCH: Die Weiche der Automatisierung – Warum Einfachheit IEC 61499 revolutioniert
- E_SWITCH: The Unsung Hero of Industrial Automation’s Modular Design

- Unpacking E_T_FF_SR: The Secret Toggle Switch of Industrial Control Systems
- ESP32-S3 Entwicklungsplatinen ESP32-S3-DevKitC-1
- ESP32-S3-DevKitC-1 Doku-Analyse: Das Speicher-Monster (32MB Flash/16MB PSRAM) und die Macht der Dual-USB-Ports

📺 Video

- Das ESP32-S3-DevKitC-1

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 ESP32 & ESP32-S3 DevKit auf ms-muc-docs.de

🔌 io

📺 Video

- Automation der Zukunft
- Die große Migration
- Digital decisions
- Ideale Dioden
- The Dual Wi Fi Solution

🎧 Podcast

- 4diac IDE: Wie der IEC 61499 Standard die Industrieautomatisierung revolutioniert
- 4diac-Präsentation: Zielgruppen, Struktur und Alleinstellungsmerkmal Schärfen
- Den Software-Drachen zähmen: Industrielle Automatisierung und die Zukunft der Produktion
- Eclipse 4diac: Innovationsmotor Forschung vs. Nutzerbedürfnisse – Was treibt die Entwicklung wirklich voran?
- Software-Drachen zähmen: Warum IEC 61499 die Industrie revolutionieren könnte

---

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

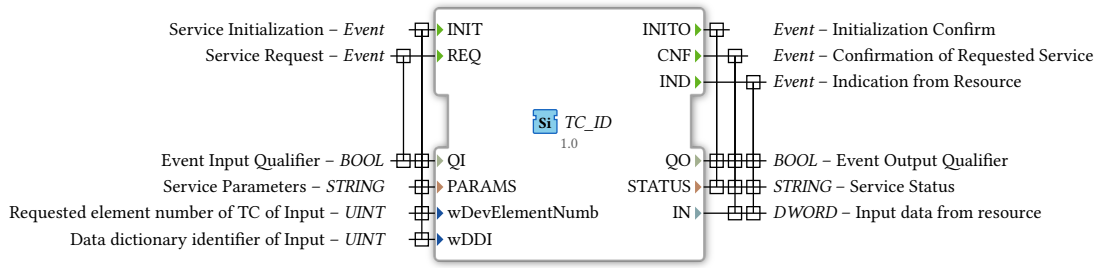
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 Diode & Halbleitergrundlagen auf ms-muc-docs.de
- 🌐 Smart Switch vs. Relais auf ms-muc-docs.de

TC_ID

• – ◦ • – ◦ • – ◦ •

Einleitung

Der TC_ID Funktionsblock ist ein Eingabe-Service-Interface-Baustein für Doppelwort-Eingabedaten (Double Word Input). Er dient als Schnittstelle zur Kommunikation mit TC-Eingabegeräten (Tractor Control) und ermöglicht den Zugriff auf spezifische Eingabedaten über definierte Datenwörter.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung

  - Verknüpft mit: QI, PARAMS, wDevElementNumb, wDDI
- **REQ**: Service-Anfrage
  - Verknüpft mit: QI

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
  - Verknüpft mit: QO, STATUS
- **CNF**: Bestätigung der angeforderten Service-Anfrage
  - Verknüpft mit: QO, STATUS, IN
- **IND**: Indikation von der Ressource
  - Verknüpft mit: QO, STATUS, IN

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangsqualifizierer
- **PARAMS** (STRING): Service-Parameter
- **wDevElementNumb** (UINT): Angeforderte Elementnummer des TC-Eingangs
- **wDDI** (UINT): Data Dictionary Identifier des Eingangs

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifizierer
- **STATUS** (STRING): Service-Status
- **IN** (DWORD): Eingabedaten von der Ressource

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der TC_ID-Block ermöglicht den Zugriff auf TC-Eingabedaten über standardisierte Service-Schnittstellen. Bei der Initialisierung (INIT) werden die Geräteparameter konfiguriert. Service-Anfragen (REQ) lösen die Datenerfassung aus, wobei die Ergebnisse über CNF- oder IND-Ereignisse zurückgemeldet werden. Der Block unterstützt sowohl angeforderte als auch spontane Datenlieferungen von der angeschlossenen Ressource.

Technische Besonderheiten

- Verarbeitet Doppelwort-Daten (DWORD, 32 Bit)
- Unterstützt Data Dictionary Identifiers (DDI) für standardisierte Datenzugriffe
- Bietet sowohl angeforderte (CNF) als auch spontane (IND) Datenlieferungen
- Enthält umfangreiche Statusrückmeldungen über STRING-Parameter

Zustandsübersicht

Der Block durchläuft typische Service-Interface-Zustände:

1. **Nicht initialisiert**: Vor INIT-Ereignis
2. **Initialisiert**: Nach erfolgreicher INITO-Rückmeldung
3. **Bereit**: Für REQ-Anfragen
4. **Aktiv**: Während Datenabfrage und -verarbeitung

Anwendungsszenarien

- Landwirtschaftliche Steuerungssysteme (ISOBUS-Umgebung)
- Traktorsteuerung und -überwachung
- Datenerfassung von TC-Eingabegeräten
- Automatisierte Agrarsysteme mit standardisierten Datenformaten

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen Eingabeblöcken bietet TC_ID:

- Standardisierte ISOBUS-Schnittstelle
- DDI-basierte Datenidentifikation
- Umfangreiches Status- und Fehlermanagement
- Sowohl angeforderte als auch spontane Datenübertragung

🛠️ Zugehörige Übungen

- Uebung_060

Fazit

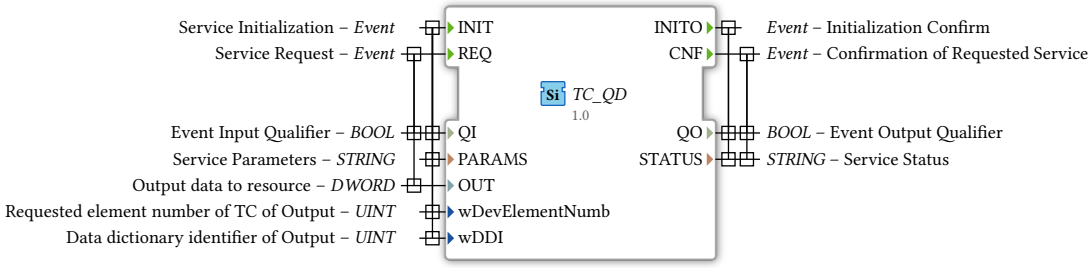
Der TC_ID Funktionsblock stellt eine robuste und standardisierte Schnittstelle für TC-Eingabedaten in ISOBUS-Umgebungen bereit. Durch seine flexible Architektur mit sowohl angefragten als auch spontanen Datenübertragungsmechanismen eignet er sich ideal für komplexe Agraranwendungen, die zuverlässigen Zugriff auf spezifische Eingabedaten erfordern.

TC_QD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der TC_QD Funktionsblock ist ein Ausgabeservice-Interface-Funktionsblock für Double-Word-Ausgabedaten. Er dient als Schnittstelle zur Kommunikation mit externen Ressourcen und ermöglicht die Ausgabe von 32-Bit-Datenwerten über spezifizierte Geräteelemente.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierungsereignis
- **REQ**: Service-Anfrageereignis

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
- **CNF**: Bestätigung der angeforderten Service-Operation

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangsqualifizierer
- **PARAMS** (STRING): Service-Parameter
- **OUT** (DWORD): Ausgabedaten für die Ressource
- **wDevElementNumb** (UINT): Angeforderte Elementnummer des TC für die Ausgabe
- **wDDI** (UINT): Data-Dictionary-Identifier der Ausgabe

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifizierer
- **STATUS** (STRING): Service-Status

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der TC_QD Funktionsblock initialisiert und verwaltet die Kommunikation mit externen Ausgabegeräten. Bei Aktivierung des INIT-Ereignisses wird der Service initialisiert, wobei die Parameter PARAMS, wDevElementNumb und wDDI konfiguriert werden. Das REQ-Ereignis löst die tatsächliche Datenausgabe aus, wobei der im OUT-Eingang bereitgestellte Double-Word-Wert an die spezifizierte Ressource gesendet wird.

Technische Besonderheiten

- Verarbeitet 32-Bit-Datenwerte (DWORD)
- Unterstützt Data-Dictionary-Identifikatoren (DDI) für standardisierte Datenreferenzierung
- Bietet Geräteelementnummern-Spezifikation für präzise Gerätesteuerung
- Liefert detaillierte Statusinformationen über den STRING-Ausgang STATUS

Zustandsübersicht

Der Funktionsblock verfügt über zwei Hauptbetriebszustände:

1. **Initialisierungszustand**: Wird durch INIT-Ereignis aktiviert und durch INITO bestätigt
2. **Service-Betriebszustand**: Wird durch REQ-Ereignis aktiviert und durch CNF abgeschlossen

Anwendungsszenarien

- Steuerung von ISOBUS-kompatiblen Landmaschinen
- Ausgabe von Steuerbefehlen an industrielle Aktoren
- Kommunikation mit TC (Tractor-Controller) Ausgabegeräten
- Anwendungen in der Agrartechnik und mobilen Arbeitsmaschinen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren Ausgabeblöcken bietet TC_QD erweiterte Funktionalität durch:

- DDI-basierte Datenidentifikation
- Parametrierbare Geräteelementnummern
- ISOBUS-Spezifikation-Kompatibilität
- Umfangreiche Statusrückmeldungen

🛠️ Zugehörige Übungen

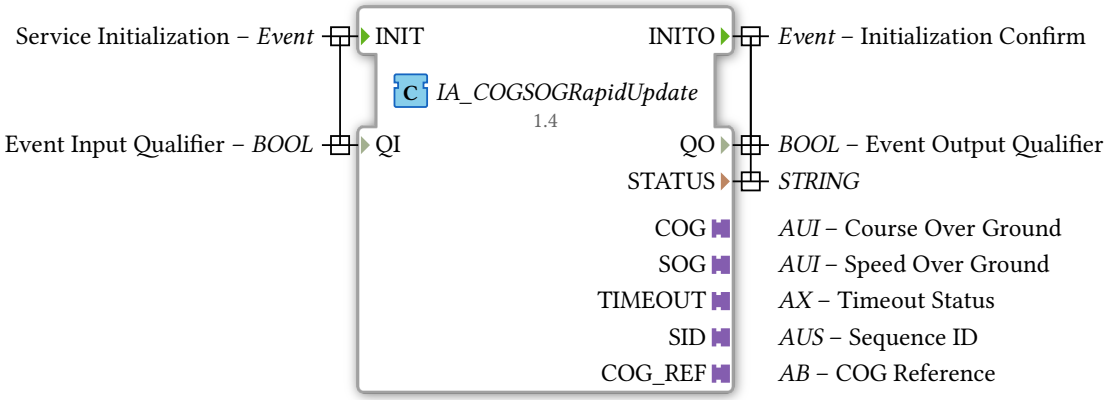
- Uebung_060

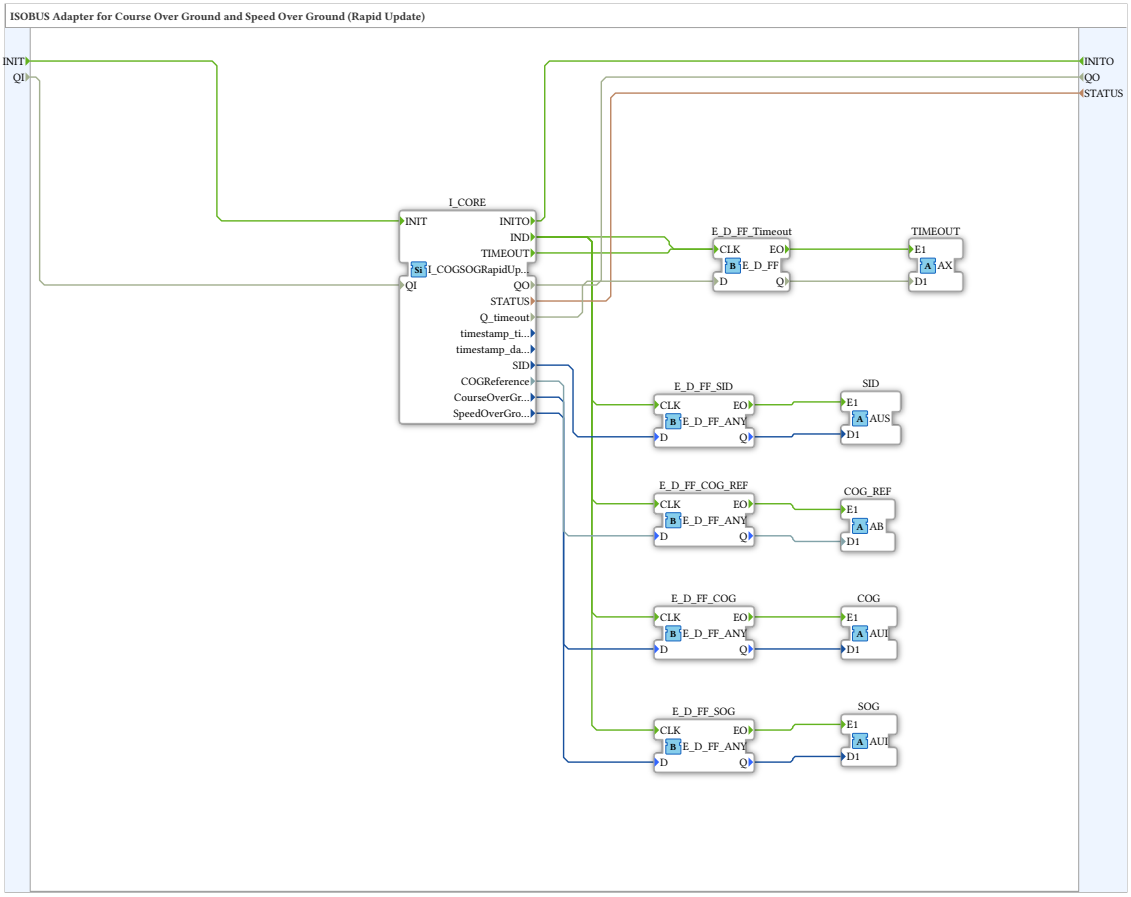
Fazit

Der TC_QD Funktionsblock stellt eine leistungsstarke und flexible Lösung für die Ausgabe von Double-Word-Daten in ISOBUS-Umgebungen dar. Seine strukturierte Schnittstelle und umfangreiche Parametrierungsmöglichkeiten machen ihn besonders geeignet für anspruchsvolle Steuerungsanwendungen in der mobilen Automation.

2.1.13.3 TECU

IA_COGSOGRapidUpdate





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **IA_COGSOGRapidUpdate** dient als ISOBUS-Adapter für die NMEA 2000 PGN 129026 Messages „Course Over Ground (COG)“ und „Speed Over Ground (SOG)“ mit Unterstützung für schnelle Aktualisierungen (Rapid Update). Er entkoppelt die ISOBUS-Kommunikation und stellt die empfangenen Navigationsdaten sowie einen Timeout-Status über standardisierte Adapter-Schnittstellen bereit. Der Baustein wurde auf Basis der Eclipse Public License 2.0 entwickelt und ist für den Einsatz in landwirtschaftlichen Steuerungssystemen (TECU) optimiert.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar
INIT	EInit	Service-Initialisierung, ausgelöst durch Eingang QI

Ereignis-Ausgänge

Ereignis	Typ	Kommentar
INITO	EInit	Initialisierungsbestätigung, liefert QO und STATUS

Daten-Eingänge

Variable	Typ	Kommentar
QI	BOOL	Eingangsqualifikator für die Initialisierung

Daten-Ausgänge

Variable	Typ	Kommentar
QO	BOOL	Ausgangsqualifikator (Initialisierungsstatus)
STATUS	STRING	Statusmeldung (z. B. Fehlertext oder „OK“)

Adapter

Adapter-Name	Typ	Kommentar
COG	adapter::types::unidirectional::AUI	Kurs über Grund (Course Over Ground)
SOG	adapter::types::unidirectional::AUI	Geschwindigkeit über Grund (Speed Over Ground)
TIMEOUT	adapter::types::unidirectional::AX	Timeout-Status (aktiv bei fehlender Aktualisierung)
SID	adapter::types::unidirectional::AUS	Sequenz-ID (lfd. Nummer des Datensatzes)
COG_REF	adapter::types::unidirectional::AB	Referenz für den Kurs (z. B. True/Magnetic)

Funktionsweise

1. **Initialisierung**
   Durch ein Ereignis an **INIT** wird der Baustein aktiviert. Der Eingang **QI** steuert den Start. Nach erfolgreicher Initialisierung wird **INITO** ausgelöst und die Ausgänge **QO** (TRUE bei Erfolg) und **STATUS** gesetzt.
2. **ISOBUS-Nachrichtenverarbeitung**
   Der interne Kern (FB `I_CORE` vom Typ `isobus::tecu::I_COGSOGRapidUpdate`) empfängt kontinuierlich die NMEA 2000 PGN 129026-Nachrichten. Sobald neue Daten vorliegen, werden folgende Werte über die entsprechenden Adapter ausgegeben:
   - **COG**: Kurs über Grund (z. B. in 0,01°-Schritten)
   - **SOG**: Geschwindigkeit über Grund (z. B. in 0,01 km/h)
   - **SID**: Sequenz-ID zur Synchronisation
   - **COG_REF**: Referenz (True oder Magnetic)
3. **Timeout-Überwachung**
   Falls innerhalb einer konfigurierten Zeitspanne keine gültige Nachricht empfangen wird, setzt der Baustein den **TIMEOUT**-Adapter auf aktiv. Dieses Ereignis kann von der Anwendung genutzt werden, um Fehlerzustände oder Plausibilitätsprüfungen auszulösen.
4. **Ausgabezyklen**
   Die Adapter **COG**, **SOG**, **SID** und **COG_REF** werden bei jedem gültigen Daten-Empfang gemeinsam getriggert (Ereignisverbindung `IND`). Der **TIMEOUT**-Adapter wird unabhängig davon bei Timeout ausgelöst.

Technische Besonderheiten

- **NMEA 2000 PGN 129026** – Der Baustein ist speziell für die Interpretation dieser PGN ausgelegt, die rapide Aktualisierungen von COG und SOG ermöglicht.
- **Adapter statt direkter Ein-/Ausgänge** – Alle Navigationsdaten werden über unidirektionale Adapter (Typ `unidirectional::AUI`, `AX` etc.) bereitgestellt, was eine flexible Weiterverarbeitung und Kapselung in übergeordneten Netzwerken erlaubt.
- **Lizenz und Herkunft** – Der Baustein ist unter der Eclipse Public License 2.0 verfügbar und wurde von **HR Agrartechnik GmbH** entwickelt (Version 1.0, 19.04.2026, Autor: Franz Höpfinger).
- **Integration in isobus::tecu** – Der FB ist Bestandteil der Bibliothek `isobus::tecu` und setzt auf einer hardwarenahen ISOBUS-Implementierung auf.

Zustandsübersicht

Der Funktionsblock besitzt keine explizit modellierten Zustände, dennoch lassen sich folgende Betriebsphasen ableiten:

Zustand	Beschreibung
Inaktiv	INIT noch nicht ausgelöst oder QI = FALSE
Initialisierend	Nach INIT, bis INITO mit QO gemeldet wird
Aktiv (Daten empfangen)	Regelmäßiger Empfang von PGN 129026, Aktualisierung der Adapter
Timeout	Keine gültige Nachricht innerhalb der Timeout-Schwelle – TIMEOUT-Adapter aktiv
Fehler	Initialisierung fehlgeschlagen (STATUS = Fehlertext)

Anwendungsszenarien

- **Landwirtschaftliche Zugmaschinen** – Bereitstellung von Kurs- und Geschwindigkeitsdaten für automatische Lenksysteme oder Applikationskarten.
- **ISOBUS-basierte Steuergeräte** – Einbindung in ein TECU-Netzwerk zur Auswertung von GPS/GNSS-Daten.
- **Schnelle Regelkreise** – Nutzung der Rapid-Update-Funktion für dynamische Führungsgrößen (z. B. bei teilflächenspezifischer Behandlung).
- **Fehlerüberwachung** – Einsatz des TIMEOUT-Adapters zur Erkennung von Datenausfällen und zur Aktivierung von Redundanzmechanismen.

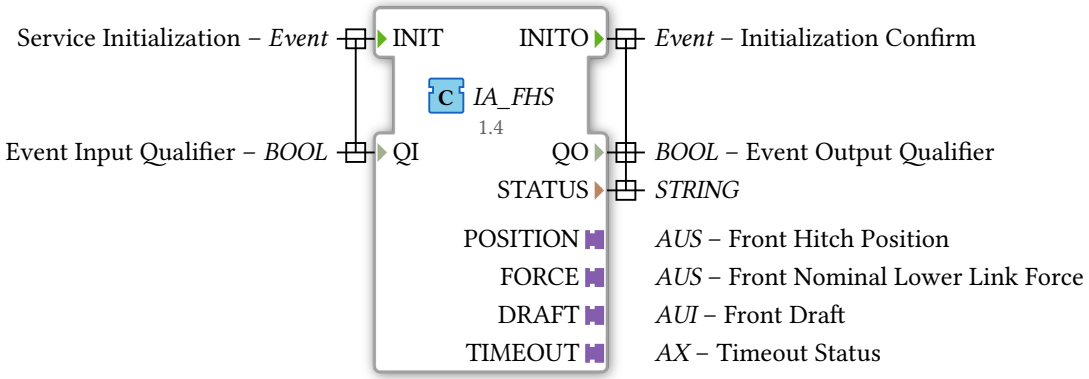
Vergleich mit ähnlichen Bausteinen

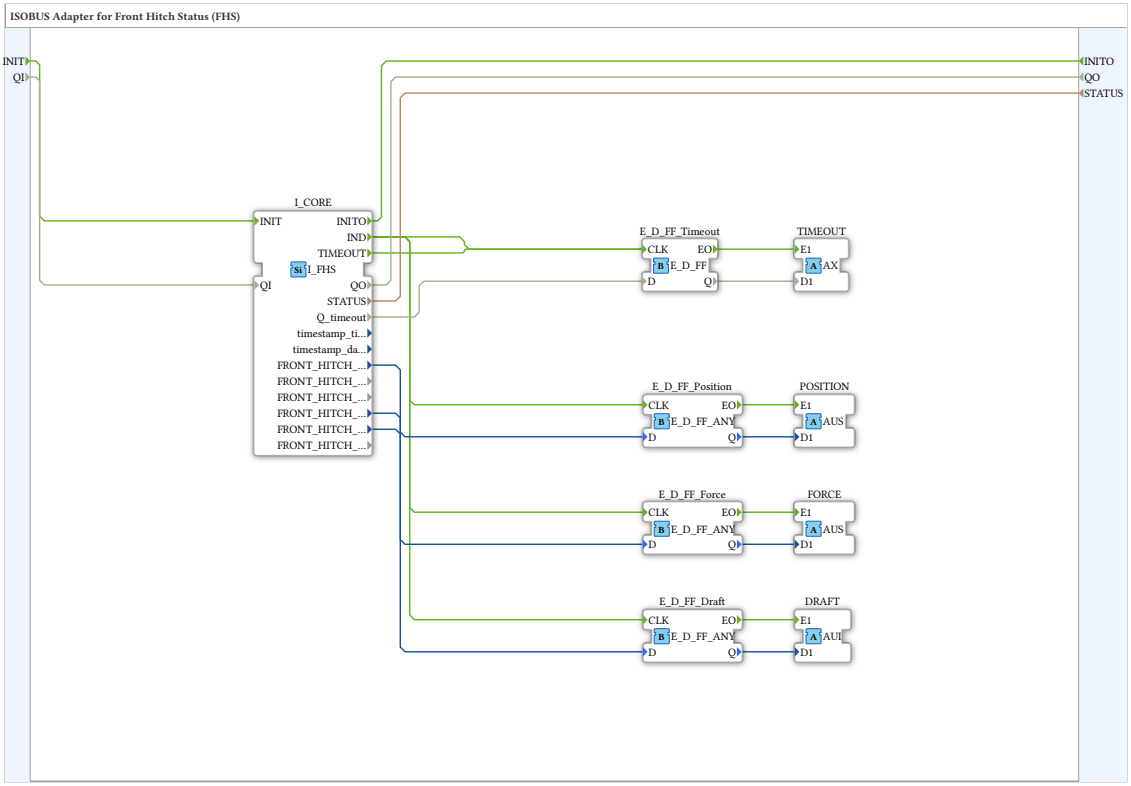
Es gibt allgemeine ISOBUS-Adapter für COG/SOG (z. B. `I_COGSOG`), die jedoch keine Rapid-Update-Unterstützung bieten. Der **IA_COGSOGRapidUpdate** ist speziell auf PGN 129026 ausgelegt und stellt zusätzlich die Sequenz-ID sowie den Timeout-Status bereit. Ähnliche Bausteine wie `I_COGSOGStd` verwenden möglicherweise eine langsamere Aktualisierungsrate oder fehlende Timeout-Erkennung. Die Verwendung von Adapter-Schnittstellen (statt fester Datenport) erhöht die Flexibilität, erfordert aber eine entsprechende Adapter-Konfiguration im übergeordneten Projekt.

Fazit

Der **IA_COGSOGRapidUpdate** ist ein spezialisierter, lizenzrechtlich freier ISOBUS-Adapter für hochfrequente Navigationsdaten. Er kapselt die komplexe NMEA-2000-Kommunikation und bietet standardisierte Adapterausgänge für Kurs, Geschwindigkeit, Sequenz-ID und Timeout. Durch seine klare Initialisierungslogik und die Möglichkeit zur Fehlerüberwachung eignet er sich ideal für Echtzeit-Anwendungen in der Agrartechnik. Die Nutzung der Adapter-Technologie ermöglicht eine saubere Modularisierung und Wiederverwendbarkeit in unterschiedlichsten Steuerungsarchitekturen.

IA_FHS





• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock **IA_FHS** dient als ISOBUS-Adapter für den Front Hitch Status (FHS) gemäß ISO 11783-7 (PGN 65094). Er kapselt die interne Verarbeitung des Bausteins `I_FHS` und stellt die gewonnenen Mess- und Statuswerte über standardisierte unidirektionale Adapterausgänge zur Verfügung. Die Initialisierung erfolgt über das Ereignis `INIT`; nach erfolgreichem Abschluss werden die Daten über die Adapterschnittstellen bereitgestellt.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	mit Variablen	Beschreibung
`INIT`	EInit	`QI`	Service-Initialisierung

Ereignis-Ausgänge

Ereignis	Typ	mit Variablen	Beschreibung
`INITO`	EInit	`QO`, `STATUS`	Initialisierungsbestätigung

Daten-Eingänge

Variable	Typ	Beschreibung
`QI`	BOOL	Ereignis-Eingangs-Qualifier

Daten-Ausgänge

Variable	Typ	Beschreibung
`QO`	BOOL	Ausgangs-Qualifier

STATUS	STRING	Statusmeldung (z. B. „OK“)

Adapter

Adapter	Typ	Beschreibung
POSITION	adapter::types::unidirectional::AUS	Front-Hitch-Position (analog, unsigniert)
FORCE	adapter::types::unidirectional::AUS	Front-Nominal-Lower-Link-Force (analog)
DRAFT	adapter::types::unidirectional::AUI	Front-Draft (analog, ganzzahlig unsigniert)
TIMEOUT	adapter::types::unidirectional::AX	Timeout-Status des ISOBUS-Empfangs

Funktionsweise

Der Baustein enthält intern eine Instanz I_CORE vom Typ isobus::tecu::I_FHS. Beim Eintreffen des Initialisierungsereignisses INIT wird der Kern mit dem Qualifier QI aktiviert. Nach erfolgreicher Initialisierung löst dieser das Ereignis INITO aus, setzt QO = TRUE und gibt den Status auf STATUS aus. Gleichzeitig generiert der Kern das Ereignis IND, das über die internen Ereignisverbindungen an die Adapter POSITION, FORCE und DRAFT weitergeleitet wird. Die zugehörigen Datenwerte (FRONT_HITCH_POSITION, FRONT_HITCH_NOMINAL_LOWER_LINK_FORCE, FRONT_HITCH_DRAFT) werden dabei über die entsprechenden Datenverbindungen an die Adapterausgänge übertragen. Ein etwaiger Timeout-Zustand des ISOBUS-Empfangs wird über das Adapter-Ereignis TIMEOUT.E1 und den Datenwert Q_timeout gemeldet.

Technische Besonderheiten

- **Standardkonformität**: Der Baustein erfüllt die Anforderungen der ISO 11783-7 für die **PGN 65094** (Front Hitch Status).
- **Adapter-Schnittstellen**: Alle Sensorwerte werden über unidirektionale Adapter ausgegeben, die eine flexible und entkoppelte Weiterverarbeitung in der 4diac-IDE ermöglichen.
- **Initialisierungssequenz**: Die Initialisierung muss explizit über das Ereignis INIT angestoßen werden; der Baustein arbeitet erst nach erfolgreichem Abschluss korrekt.
- **Timeout-Überwachung**: Der Adapter TIMEOUT signalisiert, ob der ISOBUS-Datenempfang innerhalb der erwarteten Zeit erfolgt. Dies dient der Fehlererkennung in der Feldbuskommunikation.

Zustandsübersicht

Der Baustein besitzt keinen eigenen sichtbaren Zustandsautomaten. Die interne Ablaufsteuerung erfolgt über den Kernbaustein I_FHS. Grob lassen sich zwei Phasen unterscheiden:

- **Vor Initialisierung**: Der Baustein ist inaktiv. Die Ereignis-Ausgänge und Adapter liefern keine Daten.
- **Nach Initialisierung (INIT erfolgreich)**:
  - INITO wird ausgelöst, QO = TRUE.
  - Die Adapter POSITION, FORCE, DRAFT werden zyklisch oder ereignisgesteuert mit aktuellen Messwerten versorgt.
  - Der Adapter TIMEOUT meldet den aktuellen Timeout-Status.

Tritt während des Betriebs ein Fehler auf (z. B. Verbindungsabbruch), wird dies über den Timeout-Adapter oder den Status-Ausgang signalisiert.

Anwendungsszenarien

- **Traktorsteuerung**: Erfassung der Front-Hubwerksposition und der Belastung des Unterlenkers zur Regelung von Arbeitsgeräten (z. B. Frontlader, Fronthubwerk).
- **ISOBUS-Diagnose**: Überwachung des ISOBUS-Datenstroms auf Timeouts und Plausibilität der Kraft- und Positionswerte.
- **Datenaufzeichnung**: Bereitstellung von Position, Kraft und Zugkraft für Telemetrie- oder Protokollierungsanwendungen.

Vergleich mit ähnlichen Bausteinen

- **IA_RHS** (Rear Hitch Status): Analoger Adapter für den Heckkraftheber. Die Adressierung und PGN-Nummer unterscheiden sich (z. B. PGN 65096). Während IA_FHS die Front-Daten liefert, fokussiert IA_RHS auf den hinteren Bereich.

- **Direkter I_FHS-Baustein**: Ohne Adapter-Wrapper müsste der Anwender den Kernbaustein `I_FHS` selbst einbinden und die erforderlichen Daten- und Ereignisverbindungen manuell setzen. Der Adapter-Baustein vereinfacht die Handhabung durch standardisierte, vorgefertigte Schnittstellen.

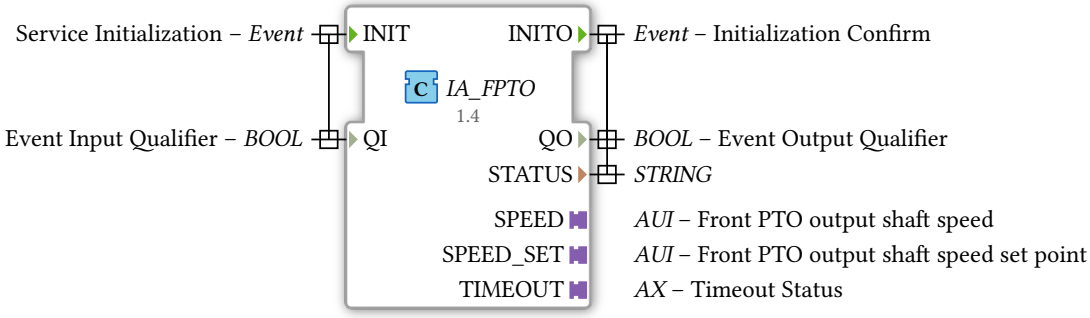
Fazit

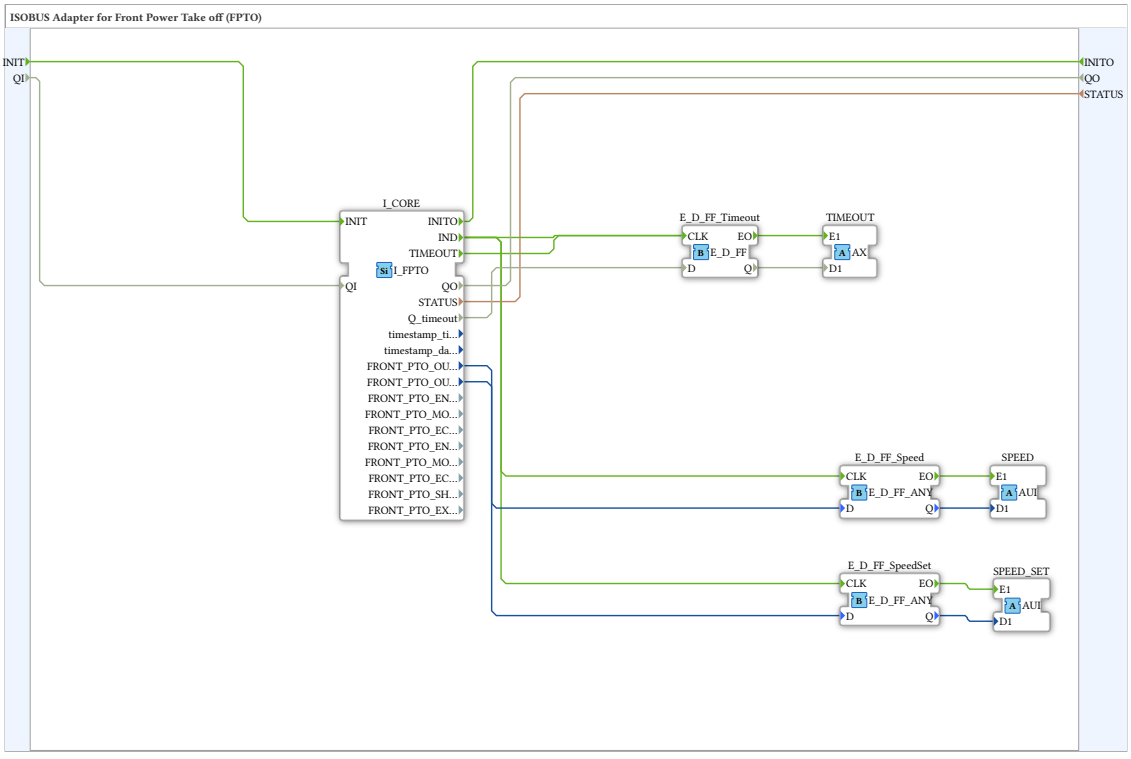
Der Funktionsblock **IA_FHS** bietet einen fertig konfektionierten ISOBUS-Adapter für den Front-Hitch-Status. Dank der internen Kapselung des `I_FHS`-Kerns und der Ausgabe über unidirektionale Adapter kann der Anwender die Werte für Position, Kraft und Zugkraft schnell und zuverlässig in eigene Steuerungs- oder Überwachungsanwendungen einbinden. Die integrierte Timeout-Überwachung erhöht die Betriebssicherheit der ISOBUS-Kommunikation.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

IA_FPTO





[Bild des Funktionsblocks, falls vorhanden]

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **IA_FPTO** dient als ISOBUS-Adapter für die Frontzapfwelle (Front Power Take Off, FPTO) gemäß ISO 11783-7 (PGN 65092). Er kapselt die interne Logik des Bausteins **I_FPTO** und stellt die

relevanten Signale über standardisierte Adapter-Schnittstellen zur Verfügung. Damit wird eine nahtlose Integration in ISOBUS-basierte Steuerungssysteme ermöglicht.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung	Mit Variablen
INIT	EInit	Service-Initialisierung	QI

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung	Mit Variablen
INITO	EInit	Bestätigung der Initialisierung	QO, STATUS

Daten-Eingänge

Variable	Typ	Beschreibung
QI	BOOL	Qualifikator für den INIT-Eingang

Daten-Ausgänge

Variable	Typ	Beschreibung
QO	BOOL	Qualifikator für den INITO-Ausgang
STATUS	STRING	Statusmeldung nach der Initialisierung

Adapter

Adapter	Typ	Beschreibung
SPEED	adapter::types::unidirectional::AUI	Ausgangsdrehzahl der Frontzapfwellenwelle (Istwert)
SPEED_SET	adapter::types::unidirectional::AUI	Sollwert der Frontzapfwellendrehzahl
TIMEOUT	adapter::types::unidirectional::AX	Timeout-Status der Kommunikation

Alle Adapter sind unidirektional ausgeführt: Die Werte werden vom FB gesendet, aber nicht empfangen.

Funktionsweise

Der FB wird über das Ereignis **INIT** aktiviert. Sofern der Eingangsqualifikator **QI** gesetzt ist, initialisiert die interne Logik die Verbindung zur ISOBUS-Kommunikationsschicht. Nach erfolgreicher Initialisierung wird das Ereignis **INITO** ausgelöst, die Ausgangsvariablen **QO** und **STATUS** geben den Status der Initialisierung zurück.

Nach der Initialisierung arbeitet der Baustein autonom: Er liefert zyklisch die aktuelle Drehzahl der Frontzapfwelle über den Adapter **SPEED** sowie den zugehörigen Sollwert über **SPEED_SET**. Der Adapter **TIMEOUT** zeigt an, ob ein Kommunikationszeitfehler aufgetreten ist (z.B. wenn das Steuergerät keine aktuellen Daten sendet). Diese Ausgaben werden kontinuierlich aktualisiert, solange die Initialisierung gültig bleibt.

Technische Besonderheiten

- Der FB basiert auf dem internen Baustein **I_FPTO** aus der Bibliothek `isobus::tecu`. Die Adapter-Schnittstellen entkoppeln die Anwendung von der konkreten Signalkodierung des ISOBUS.
- Die Adapter **SPEED** und **SPEED_SET** verwenden den unidirektionalen Typ **AUI** (Adresse unbekannt, Integer), der für analoge Werte wie Drehzahlen geeignet ist.
- Der Adapter **TIMEOUT** nutzt den Typ **AX** (Adresse unbekannt, Boolescher Wert), um den Timeout-Zustand binär darzustellen.

- Der FB enthält kein eigenes Verhalten auf Ereignisse außerhalb der Initialisierung; die Datenaktualisierung erfolgt implizit über die Netzwerkverbindungen des Bausteins **I_FPTO**.

Zustandsübersicht

Der FB besitzt keine explizite Zustandsmaschine. Das Verhalten lässt sich wie folgt beschreiben:

1. **Initialisierungsphase**: Nach dem Ereignis **INIT** wird der interne FB **I_FPTO** initialisiert. Während dieser Phase sind die Adapterausgänge ungültig oder auf einem definierten Startwert (0/FALSE).
2. **Aktive Phase**: Nach Bestätigung der Initialisierung (**INITO**) stehen die aktuellen Werte an den Adaptern **SPEED**, **SPEED_SET** und **TIMEOUT** stabil zur Verfügung.
3. **Fehlerzustand**: Tritt ein Timeout auf, wird der Adapter **TIMEOUT** auf TRUE gesetzt. Die Drehzahlwerte bleiben möglicherweise auf dem letzten gültigen Wert stehen, bis die Kommunikation wiederhergestellt ist.

Eine erneute Initialisierung (erneuter INIT-Impuls) kann den FB zurücksetzen und neu starten.

Anwendungsszenarien

- **Steuerung der Frontzapfwelle in landwirtschaftlichen Traktoren**: Der FB empfängt über den ISOBUS die Soll- und Ist-Drehzahlen der Frontzapfwelle und gibt sie an übergeordnete Steuerungen oder Anzeigen weiter.
- **Integration in ISOBUS-konforme Anbaugeräte**: Anbaugeräte wie Frontmähwerke oder Frontlader können die Drehzahl- und Timeout-Informationen nutzen, um ihre Funktionen daran anzupassen.
- **Ferndiagnose und Überwachung**: Das Timeout-Signal ermöglicht eine Überwachung der Kommunikationsverbindung zur Zapfwellensteuerung, was für Fehlererkennung und Sicherheitsfunktionen wichtig ist.

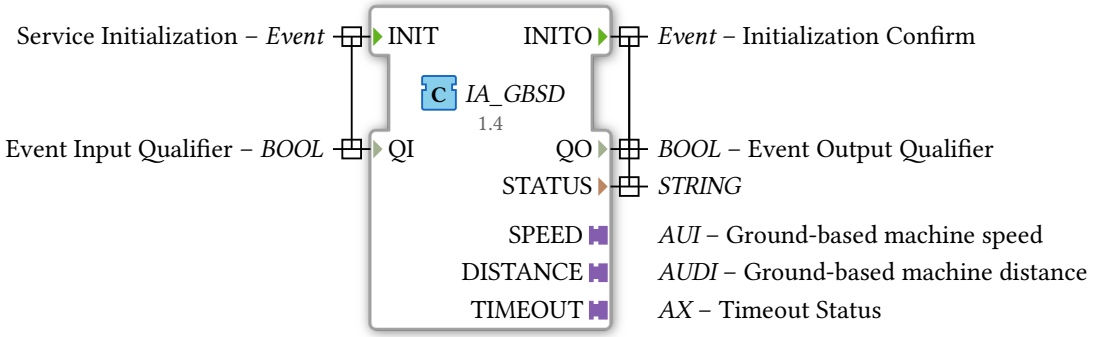
Vergleich mit ähnlichen Bausteinen

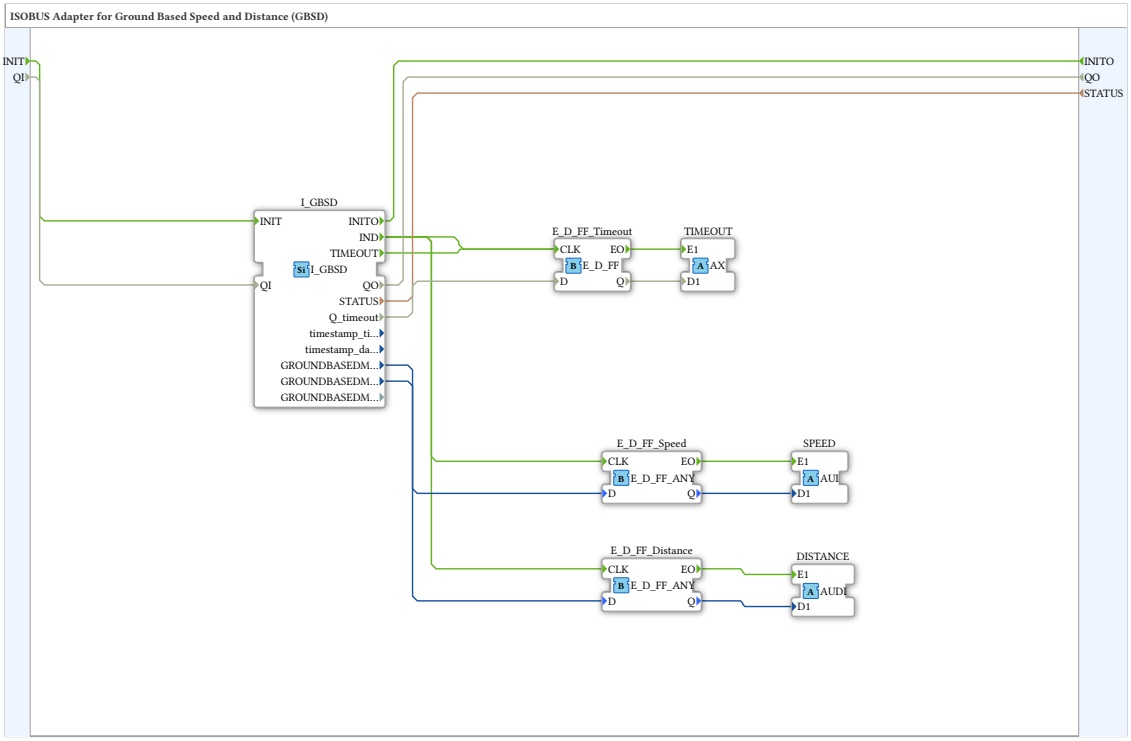
Im ISOBUS-Umfeld gibt es weitere Adapter für andere Zapfwellen (z.B. **IA_RPTO** für die Heckzapfwelle) oder für allgemeine Steuergeräte. **IA_FPTO** ist speziell auf die Frontzapfwelle zugeschnitten und enthält nur die für diese Anwendung relevanten Signale (Drehzahl Ist/Soll und Timeout). Im Vergleich zu einem generischen ISOBUS-Adapter bietet er somit eine höhere Abstraktion und einfachere Handhabung für den Anwendungsentwickler.

Fazit

Der Funktionsblock **IA_FPTO** stellt eine schlanke und spezifische Schnittstelle für die ISOBUS-konforme Nutzung der Frontzapfwelle dar. Durch die Kapselung der internen Kommunikationslogik und die Bereitstellung über unidirektionale Adapter wird die Integration in Steuerungsanwendungen erheblich vereinfacht. Der Baustein eignet sich besonders für Traktoren und landwirtschaftliche Maschinen, bei denen eine zuverlässige Drehzahlüberwachung und Timeout-Erkennung erforderlich ist.

IA_GBSD





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **IA_GBSD** dient als Adapter-Wrapper für den ISOBUS-kompatiblen Baustein **I_GBSD** („Ground Based Speed and Distance“). Er kapselt die Initialisierung und Datenkommunikation eines bodenbasierten Geschwindigkeits- und Distanzsensors gemäß ISO 11783-7 (PGN 65097) und stellt die Werte über unidirektionale Adapter-Schnittstellen bereit.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Service Initialization

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Initialization Confirm

Daten-Eingänge

Name	Typ	Kommentar
QI	BOOL	Event Input Qualifier

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Event Output Qualifier
STATUS	STRING	Statusinformation

Adapter

Name	Typ	Kommentar
SPEED	adapter::types::unidirectional::AUI	Bodenbasierte Maschinengeschwindigkeit
DISTANCE	adapter::types::unidirectional::AUDI	Bodenbasierte Maschinendistanz
TIMEOUT	adapter::types::unidirectional::AX	Timeout-Status

Funktionsweise

Der FB ist als reiner Adapter-Wrapper realisiert:

- Ein INIT-Ereignis am Eingang wird direkt an den internen Baustein **I_GBSD** weitergeleitet.
- Der Daten-Eingang **QI** aktiviert bzw. deaktiviert die Verarbeitung.
- Nach der Initialisierung wird das Ereignis **INITO** mit dem Qualifier **QO** und einem **STATUS** ausgegeben.
- Die berechneten Grössen (Geschwindigkeit und Distanz) werden über die Adapter **SPEED** und **DISTANCE** ausgegeben; bei einer Zeitüberschreitung wird der **TIMEOUT**-Adapter aktiviert.
- Alle Datenverbindungen werden innerhalb des FB-Netzwerks zwischen den Ein-/Ausgängen und dem internen Baustein hergestellt.

Technische Besonderheiten

- **Adapter-basierte Schnittstelle**: Die Ausgangsdaten werden nicht als feste Variablen, sondern über unidirektionale Adapter (AUI, AUDI, AX) bereitgestellt, was eine flexible Anbindung an andere Komponenten erlaubt.
- **ISOBUS-Konformität**: Der FB implementiert das PGN 65097 nach ISO 11783-7 – Standard für bodenbasierte Geschwindigkeit und Distanz in der Landtechnik.
- **Timeout-Überwachung**: Der separate Timeout-Adapter signalisiert, ob die Sensorwerte innerhalb einer erwarteten Zeitspanne aktualisiert wurden.
- **Kompaktes Design**: Es gibt keinen eigenen Zustandsautomaten; alle Steuerlogik liegt im internen Baustein **I_GBSD**.

Zustandsübersicht

Der FB selbst besitzt keinen expliziten Endlichen Automaten. Der interne Baustein durchläuft einen Initialisierungs-Zustand (ausgelöst durch INIT) und einen Arbeitszustand, in dem kontinuierlich Geschwindigkeit, Distanz und Timeout-Status aktualisiert werden.

Anwendungsszenarien

- **ISOBUS-Steuergeräte (TECU)**: Einbindung eines Radarsensors oder eines anderen bodenbasierten Geschwindigkeits-/Distanzsensors in ein ISOBUS-kompatibles System.
- **Landwirtschaftliche Fahrzeuge**: Bereitstellung von Fahrgeschwindigkeit und zurückgelegter Strecke für nachgeschaltete Funktionen (z. B. dosierte Applikationen, Teilbreitenschaltung).
- **Test- und Simulationsumgebungen**: Nutzung des Adapter-Wrappers, um den realen Sensor durch eine simulierte Datenquelle zu ersetzen.

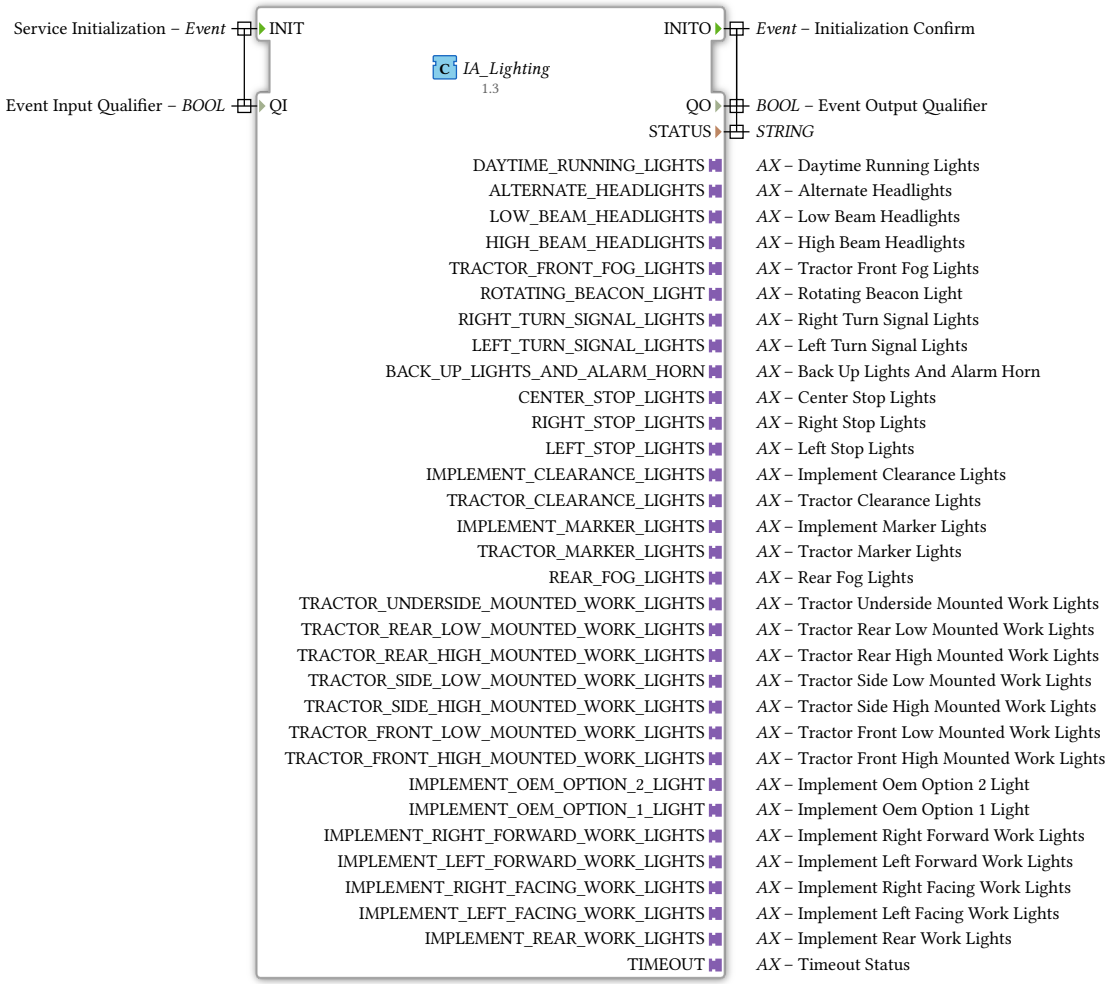
Vergleich mit ähnlichen Bausteinen

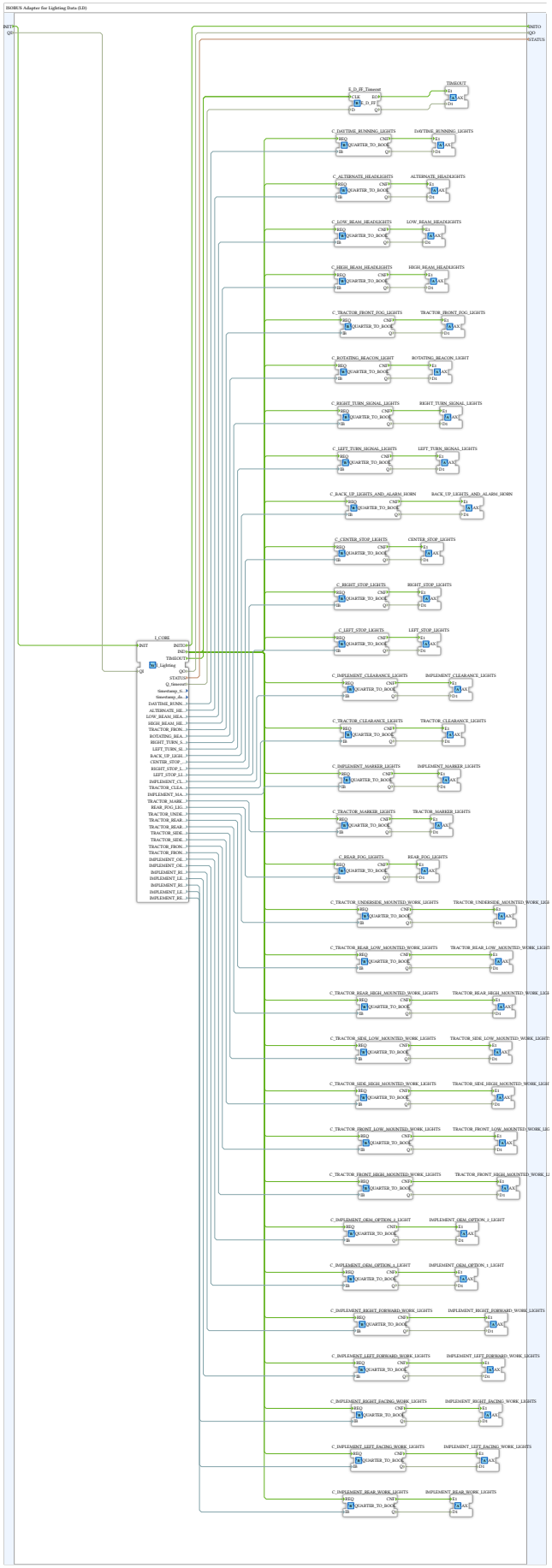
- **IA_GBSD** vs. **I_GBSD**: I_GBSD ist der nativ programmierte ISOBUS-Baustein, während IA_GBSD dessen Funktionalität ausschliesslich über Adapter zugänglich macht. Der Wrapper erleichtert die Verschaltung in grafischen Editoren und trennt die eigentliche Logik von der Schnittstellenanbindung.
- **Alternative Adapter-Wrapper**: Ähnliche Wrapper existieren für andere ISOBUS-PGNs (z. B. für Navigationsdaten oder Arbeitshydraulik), die ebenfalls Adapter für ihre Ausgangswerte bereitstellen.

Fazit

IA_GBSD ist ein effizienter Adapter-Wrapper, der die bodenbasierte Geschwindigkeits- und Distanzmessung nach ISO 11783-7 kapselt. Die Adapter-Schnittstellen ermöglichen eine saubere modulare Integration in übergeordnete Steuerungslogiken, während die eigentliche ISOBUS-Implementierung im Inneren erhalten bleibt. Dadurch wird der Baustein sowohl für direkte Anwendungen als auch für Simulations- und Testzwecke nutzbar.

IA_Lighting


Service Initialization – Event
INIT
Event Input Qualifier – BOOL
QI
C
IA_Lighting
1.3
INITO
Event – Initialization Confirm
QO
BOOL – Event Output Qualifier
STATUS
STRING
DAYTIME_RUNNING_LIGHTS
AX – Daytime Running Lights
ALTERNATE_HEADLIGHTS
AX – Alternate Headlights
LOW_BEAM_HEADLIGHTS
AX – Low Beam Headlights
HIGH_BEAM_HEADLIGHTS
AX – High Beam Headlights
TRACTOR_FRONT_FOG_LIGHTS
AX – Tractor Front Fog Lights
ROTATING_BEACON_LIGHT
AX – Rotating Beacon Light
RIGHT_TURN_SIGNAL_LIGHTS
AX – Right Turn Signal Lights
LEFT_TURN_SIGNAL_LIGHTS
AX – Left Turn Signal Lights
BACK_UP_LIGHTS_AND_ALARM_HORN
AX – Back Up Lights And Alarm Horn
CENTER_STOP_LIGHTS
AX – Center Stop Lights
RIGHT_STOP_LIGHTS
AX – Right Stop Lights
LEFT_STOP_LIGHTS
AX – Left Stop Lights
IMPLEMENT_CLEARANCE_LIGHTS
AX – Implement Clearance Lights
TRACTOR_CLEARANCE_LIGHTS
AX – Tractor Clearance Lights
IMPLEMENT_MARKER_LIGHTS
AX – Implement Marker Lights
TRACTOR_MARKER_LIGHTS
AX – Tractor Marker Lights
REAR_FOG_LIGHTS
AX – Rear Fog Lights
TRACTOR_UNDERSIDE_MOUNTED_WORK_LIGHTS
AX – Tractor Underside Mounted Work Lights
TRACTOR_REAR_LOW_MOUNTED_WORK_LIGHTS
AX – Tractor Rear Low Mounted Work Lights
TRACTOR_REAR_HIGH_MOUNTED_WORK_LIGHTS
AX – Tractor Rear High Mounted Work Lights
TRACTOR_SIDE_LOW_MOUNTED_WORK_LIGHTS
AX – Tractor Side Low Mounted Work Lights
TRACTOR_SIDE_HIGH_MOUNTED_WORK_LIGHTS
AX – Tractor Side High Mounted Work Lights
TRACTOR_FRONT_LOW_MOUNTED_WORK_LIGHTS
AX – Tractor Front Low Mounted Work Lights
TRACTOR_FRONT_HIGH_MOUNTED_WORK_LIGHTS
AX – Tractor Front High Mounted Work Lights
IMPLEMENT_OEM_OPTION_2_LIGHT
AX – Implement Oem Option 2 Light
IMPLEMENT_OEM_OPTION_1_LIGHT
AX – Implement Oem Option 1 Light
IMPLEMENT_RIGHT_FORWARD_WORK_LIGHTS
AX – Implement Right Forward Work Lights
IMPLEMENT_LEFT_FORWARD_WORK_LIGHTS
AX – Implement Left Forward Work Lights
IMPLEMENT_RIGHT_FACING_WORK_LIGHTS
AX – Implement Right Facing Work Lights
IMPLEMENT_LEFT_FACING_WORK_LIGHTS
AX – Implement Left Facing Work Lights
IMPLEMENT_REAR_WORK_LIGHTS
AX – Implement Rear Work Lights
TIMEOUT
AX – Timeout Status



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **IA_Lighting** dient als ISOBUS-Adapter für Beleuchtungsdaten (Lighting Data, LD) gemäß ISO 11783-7 (PGN 65088). Er kapselt einen internen `I_Lighting`-Kern und wandelt dessen 32-Bit-Integer-Ausgänge für jede Lichtfunktion in boolesche Einzelsignale um. Über eine Vielzahl von Adapter-Plugs werden die einzelnen Lichtfunktionen wie Tagfahrlicht, Abblendlicht, Blinker, Arbeitsleuchten usw. als getrennte logische Signale an die Applikation weitergegeben.

• – ◦ • – ◦ • – ◦ •

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung
INIT	EInit	Initialisierung des Bausteins. Wird mit `QI` ausgelöst.

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung
INITO	EInit	Bestätigung der erfolgreichen Initialisierung. Wird zusammen mit `QO` und `STATUS` ausgegeben.

Daten-Eingänge

Name	Typ	Beschreibung
QI	BOOL	Qualifizierer für die Initialisierung (Freigabe).

Daten-Ausgänge

Name	Typ	Beschreibung
QO	BOOL	Ausgangsqualifizierer – zeigt an, ob der Baustein betriebsbereit ist.
STATUS	STRING	Statusmeldung (z. B. Fehlertext oder Erfolgsmeldung).

Adapter

Der Baustein besitzt **32 unidirektionale Adapter-Plugs** (Typ `adapter::types::unidirectional::AX`). Jeder Adapter repräsentiert eine spezifische Lichtfunktion nach ISO 11783-7 und stellt einen Ereignisausgang (`E1`) sowie einen Datenausgang (`D1`) vom Typ `BOOL` bereit:

Adapter-Name	Beschreibung
DAYTIME_RUNNING_LIGHTS	Tagfahrlicht
ALTERNATE_HEADLIGHTS	Alternatives Fernlicht (z. B. Fernlichtassistent)
LOW_BEAM_HEADLIGHTS	Abblendlicht
HIGH_BEAM_HEADLIGHTS	Fernlicht
TRACTOR_FRONT_FOG_LIGHTS	Zugfahrzeug-Frontnebelscheinwerfer
ROTATING_BEACON_LIGHT	Rundumkennleuchte
RIGHT_TURN_SIGNAL_LIGHTS	Rechter Blinker
LEFT_TURN_SIGNAL_LIGHTS	Linker Blinker
BACK_UP_LIGHTS_AND_ALARM_HORN	Rückfahrleuchte und Alarmhorn
CENTER_STOP_LIGHTS	Mittleres Bremslicht

RIGHT_STOP_LIGHTS	Rechtes Bremslicht
LEFT_STOP_LIGHTS	Linkes Bremslicht
IMPLEMENT_CLEARANCE_LIGHTS	Anbaugerät-Begrenzungsleuchten
TRACTOR_CLEARANCE_LIGHTS	Zugfahrzeug-Begrenzungsleuchten
IMPLEMENT_MARKER_LIGHTS	Anbaugerät-Markierungsleuchten
TRACTOR_MARKER_LIGHTS	Zugfahrzeug-Markierungsleuchten
REAR_FOG_LIGHTS	Rücknebel-Schlussleuchten
TRACTOR_UNDERSIDE_MOUNTED_WORK_LIGHTS	Zugfahrzeug-Arbeitsscheinwerfer (unten montiert)
TRACTOR_REAR_LOW_MOUNTED_WORK_LIGHTS	Zugfahrzeug-Arbeitsscheinwerfer (hinten, niedrig)
TRACTOR_REAR_HIGH_MOUNTED_WORK_LIGHTS	Zugfahrzeug-Arbeitsscheinwerfer (hinten, hoch)
TRACTOR_SIDE_LOW_MOUNTED_WORK_LIGHTS	Zugfahrzeug-Arbeitsscheinwerfer (seitlich, niedrig)
TRACTOR_SIDE_HIGH_MOUNTED_WORK_LIGHTS	Zugfahrzeug-Arbeitsscheinwerfer (seitlich, hoch)
TRACTOR_FRONT_LOW_MOUNTED_WORK_LIGHTS	Zugfahrzeug-Arbeitsscheinwerfer (vorn, niedrig)
TRACTOR_FRONT_HIGH_MOUNTED_WORK_LIGHTS	Zugfahrzeug-Arbeitsscheinwerfer (vorn, hoch)
IMPLEMENT_OEM_OPTION_2_LIGHT	Anbaugerät-OEM-Option 2 Licht
IMPLEMENT_OEM_OPTION_1_LIGHT	Anbaugerät-OEM-Option 1 Licht
IMPLEMENT_RIGHT_FORWARD_WORK_LIGHTS	Anbaugerät-Arbeitsscheinwerfer (rechts vorne)
IMPLEMENT_LEFT_FORWARD_WORK_LIGHTS	Anbaugerät-Arbeitsscheinwerfer (links vorne)
IMPLEMENT_RIGHT_FACING_WORK_LIGHTS	Anbaugerät-Arbeitsscheinwerfer (rechts seitlich)
IMPLEMENT_LEFT_FACING_WORK_LIGHTS	Anbaugerät-Arbeitsscheinwerfer (links seitlich)
IMPLEMENT_REAR_WORK_LIGHTS	Anbaugerät-Arbeitsscheinwerfer (hinten)
TIMEOUT	Timeout-Status des internen Kerns (Bool-Signal).

• – ◦ • – ◦ • – ◦ •

Funktionsweise

Der Baustein wird über den Ereigniseingang `INIT` mit dem Datenwert `QI` initialisiert. Bei erfolgreicher Initialisierung wird das Ereignis `INITO` ausgegeben und die Daten `QO = TRUE` sowie `STATUS` mit einer Erfolgsmeldung gesetzt.

Intern enthält der Baustein einen Kern vom Typ `isobus::tecu::I_Lighting`, welcher die ISOBUS-Kommunikation für das Beleuchtungs-PGN (Parameter Group Number 65088) abwickelt. Der Kern stellt die 32 Lichtfunktionen als 32-Bit-Integer-Werte bereit. Diese Integer-Werte werden überinstanzen des Bausteins `logiBUS::utils::quarter::QUARTER_TO_BOOL` in einzelne Boolesche Signale aufgeteilt. Jede `QUARTER_TO_BOOL`-Instanz extrahiert vermutlich jeweils 4 Bits aus dem Eingangswert und gibt die entsprechenden vier Bool-Ausgänge weiter – der genaue Aufbau ist jedoch applikationsspezifisch.

Die so gewonnenen Bool-Signale werden über die Adapter-Plugs zeitgleich mit einem Ereignis (`E1`) auf den zugehörigen Datenausgängen (`D1`) angeboten. Somit liefert der Baustein bei Aktualisierung der ISOBUS-Daten einen synchronen Ereignisstrom für jede einzelne Lichtfunktion.

• – ◦ • – ◦ • – ◦ •

Technische Besonderheiten

- **ISOBUS-Konformität**: Der Baustein setzt das standardisierte PGN 65088 (Lighting Data) nach ISO 11783-7 um und kann direkt mit einem ISOBUS-Bus gekoppelt werden.

- **Bit-Aufteilung**: Die internen 32-Bit-Werte aus dem ISOBUS-Telegramm werden mit Hilfe von `QUARTER_TO_BOOL`-Bausteinen in einzelne Bool-Signale zerlegt. Der Begriff „Quarter“ deutet auf eine Aufteilung in Gruppen von je 4 Bits hin.
- **Unidirektionale Adapter**: Jeder Adapter-Plug ist unidirektional (nur Ausgang) und liefert sowohl ein Ereignis (`E1`) als auch einen Bool-Wert (`D1`). Dies ermöglicht eine einfache Weiterverarbeitung in IEC 61499-Anwendungen, z. B. zum Ansteuern von Aktoren.
- **Status-Ausgabe**: Neben dem eigentlichen Lichtstatus gibt es einen speziellen Adapter `TIMEOUT`, der den Timeout-Status des ISOBUS-Kerns signalisiert.

• – ◦ • – ◦ • – ◦ •

Zustandsübersicht

Der Baustein selbst besitzt keine explizite Zustandsmaschine, da er im Wesentlichen ein Datenkonverter ist. Sein Verhalten wird durch den internen Kern `I_Lighting` gesteuert:

- **Initialisierung**: Nach einem `INIT`-Ereignis wechselt der Kern in den Betriebszustand (sofern `QI = TRUE`). Der Abschluss wird mit `INITO` quittiert.
- **Datenbereitstellung**: Solange der Kern aktiv ist, aktualisiert er bei jedem eingehenden ISOBUS-Telegramm die Ausgangsdaten und erzeugt für jede Lichtfunktion ein Ereignis auf dem zugehörigen Adapter.
- **Timeout**: Bei Ausbleiben von ISOBUS-Nachrichten wird der Timeout-Adapter gesetzt.

• – ◦ • – ◦ • – ◦ •

Anwendungsszenarien

- **Landwirtschaftliche Steuerungen**: Einbindung der gesamten Fahrzeugbeleuchtung (Traktor und Anbaugerät) in eine IEC 61499-basierte Steuerung, z. B. für automatische Lichtsteuerung nach ISO 11783.
- **ISOBUS-Gateway-Module**: Der Baustein eignet sich als Zwischenschicht, um ISOBUS-Beleuchtungsdaten in ein einfacheres binäres Signalformat zu wandeln und damit an speicherprogrammierbare Steuerungen oder Visualisierungssysteme weiterzugeben.
- **Nachrüstung**: Alte Traktoren ohne CAN-Bus können durch diesen Adapter mit moderner ISOBUS-Lichtsteuerung ausgestattet werden.

• – ◦ • – ◦ • – ◦ •

Vergleich mit ähnlichen Bausteinen

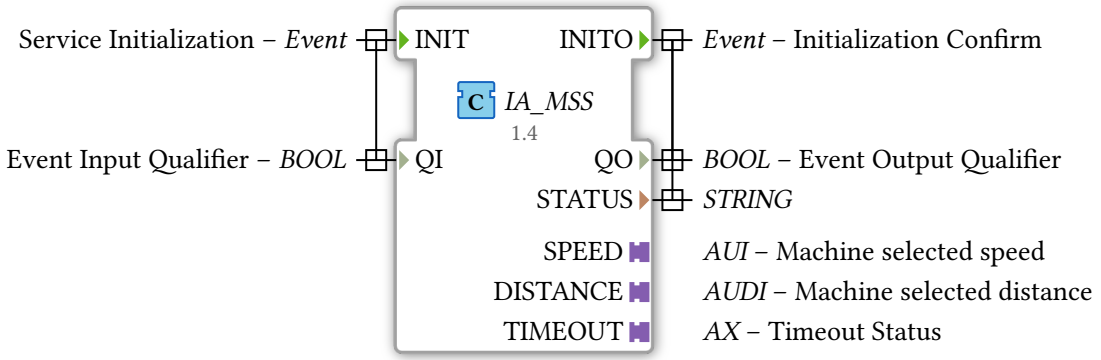
Es existieren weitere ISOBUS-Adapter-Wrapper für andere PGNs (z. B. für Arbeitshydraulik, Sitzsteuerung oder Zapfwellensteuerung). Diese Bausteine folgen dem gleichen Prinzip: Ein interner spezialisierter Kern wird über einen Adapter mit dem Applikationscode verbunden. Der wesentliche Unterschied liegt in der Anzahl und Art der Ausgangssignale – `IA_Lighting` bietet mit 32 Adaptern eine besonders hohe Anzahl von Lichtfunktionen an. Andere Adapter (z. B. `IA_ImplementSteer`) haben weniger Ausgänge, da sie nur wenige Zustände melden.

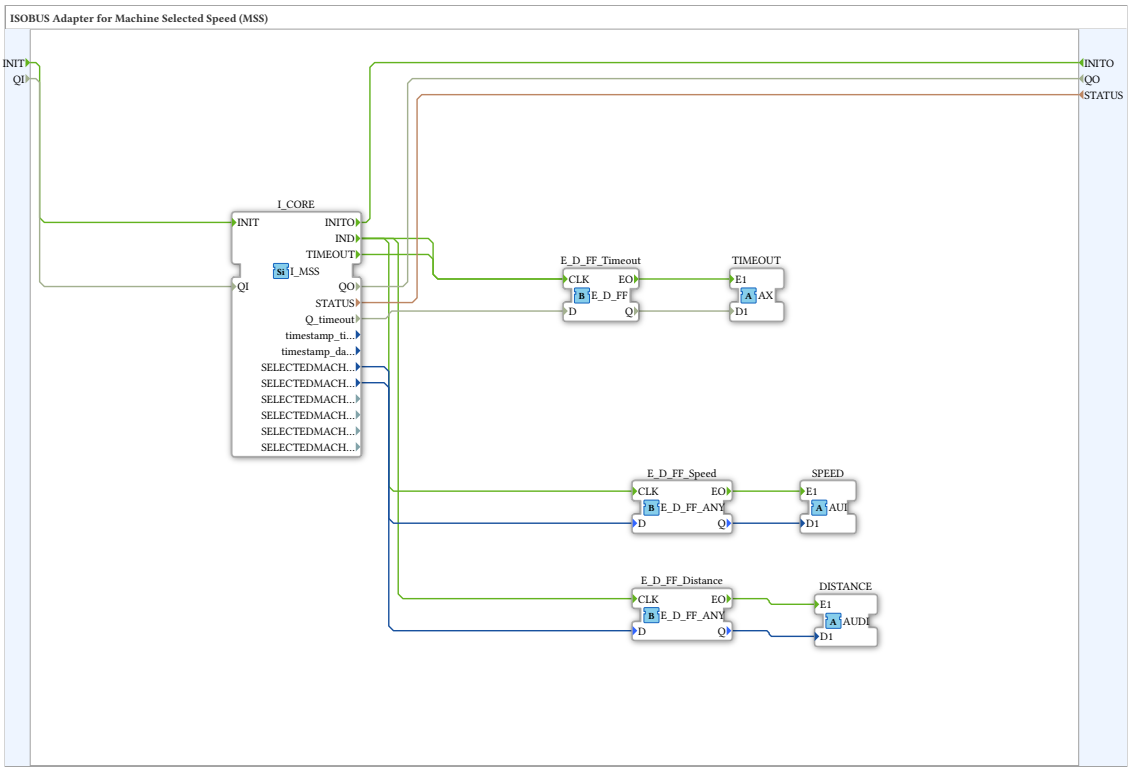
• – ◦ • – ◦ • – ◦ •

Fazit

Der Funktionsblock `IA_Lighting` ermöglicht eine komfortable und standardisierte Integration von ISOBUS-Beleuchtungsdaten in IEC 61499-Applikationen. Durch die Aufteilung der Telegramminhalte in einzelne Bool-Signale über Adapter wird eine einfache Weiterverarbeitung in der Anwendungslogik erreicht. Der Baustein ist besonders für landwirtschaftliche Steuerungssysteme geeignet, die eine vollständige Abbildung aller gängigen Lichtfunktionen nach ISO 11783-7 benötigen.

IA_MSS





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **IA_MSS** stellt einen ISOBUS-Adapter für die maschinenselektierte Geschwindigkeit (Machine Selected Speed, MSS) dar. Er kapselt die Kommunikation mit dem zugrunde liegenden ISOBUS-Protokoll (PGN 61474) und stellt die ermittelten Werte für Geschwindigkeit, zurückgelegte Distanz und Timeout-Status über standardisierte Adapter-Schnittstellen bereit. Der Baustein ist typischerweise in einem ISOBUS-fähigen Steuergerät (TECU) eingebettet und ermöglicht eine einfache Integration in IEC 61499-basierte Automatisierungssysteme.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung
INIT	EInit	Dienstinitialisierung; wird mit dem Daten-Eingang `QI` ausgelöst.

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung
INITO	EInit	Bestätigung der Initialisierung; liefert die Ausgangsdaten `QO` und `STATUS`.

Daten-Eingänge

Variable	Typ	Beschreibung
QI	BOOL	Qualifizierer für den Initialisierungseingang.

Daten-Ausgänge

Variable	Typ	Beschreibung
QO	BOOL	Qualifizierer für den Initialisierungsausgang.
STATUS	STRING	Statusmeldung der Initialisierung.

Adapter

Adapter	Typ	Beschreibung
SPEED	adapter::types::unidirectional::AUI	Ausgang für die maschinenselektierte Geschwindigkeit (Wert vom Typ `AUI`).
DISTANCE	adapter::types::unidirectional::AUDI	Ausgang für die maschinenselektierte Distanz (Wert vom Typ `AUDI`).
TIMEOUT	adapter::types::unidirectional::AX	Ausgang für den Timeout-Status (Wert vom Typ `AX`).

Funktionsweise

Der **IA_MSS**-Baustein ist als reiner Adapter-Wrapper realisiert. Er enthält intern eine Instanz des Funktionsblocks `isobus::tecu::I_MSS` (*I_CORE*), der die eigentliche ISOBUS-Logik implementiert. Über die Event- und Datenverbindungen werden die externen Schnittstellen des Adapters auf die interne Komponente abgebildet:

1. **Initialisierung** – Ein Ereignis auf `INIT` (mit gesetztem `QI`) aktiviert `I_CORE.INIT`. Nach erfolgreicher Initialisierung wird `INITO` ausgelöst, das die Qualifizierer `QO` und `STATUS` an die Ausgänge des Adapters weitergibt.
2. **Datenbereitstellung** – Der interne Baustein liefert die selektierte Geschwindigkeit (`SELECTEDMACHINESPEED`), die Distanz (`SELECTEDMACHINEDISTANCE`) sowie den Timeout-Status (`Q_timeout`). Diese Werte werden über die Adapter `SPEED`, `DISTANCE` und `TIMEOUT` ausgegeben, sobald die entsprechenden Ereignisse (`IND` bzw. `TIMEOUT`) eintreffen. Der Adapter stellt also eine saubere Trennung zwischen der ISOBUS-Kommunikation und der Applikationsebene her.

Technische Besonderheiten

- **Normkonformität** – Der Baustein basiert auf dem ISOBUS-Standard ISO 11783-7 (PGN 61474) für die maschinenselektierte Geschwindigkeit.
- **Unidirektionale Adapter** – Die Ausgänge `SPEED`, `DISTANCE` und `TIMEOUT` sind als unidirektionale Adapter (Typen `AUI`, `AUDI`, `AX`) definiert. Sie übertragen nur Daten vom Baustein zur aufrufenden Applikation; eine Rückkopplung ist nicht vorgesehen.
- **Adapter-Wrapper** – Die interne Logik ist vollständig in der Komponente `I_CORE` gekapselt. `IA_MSS` dient ausschließlich der Schnittstellenanpassung und Event-Weiterleitung.
- **Fehlerbehandlung** – Der Initialisierungsstatus wird über `STATUS` (STRING) und `QO` (BOOL) signalisiert. Ein eventueller Timeout wird über den `TIMEOUT`-Adapter gemeldet.

Zustandsübersicht

Da der Baustein keinen eigenen Ausführungszustand (ECC) besitzt, ergibt sich der Zustand aus dem Zusammenspiel von Eingangs- und Ausgangsereignissen:

Zustand	Beschreibung
Nicht initialisiert	Nach dem Systemstart oder nach einer fehlgeschlagenen Initialisierung. `INIT` kann ausgelöst werden.
Initialisiert	Nach erfolgreichem Durchlauf von `INIT` → `INITO` mit gesetztem `QO`. Die Adapter liefern gültige Werte.
Fehler/Timeout	Falls die interne Initialisierung fehlschlägt oder ein Timeout erkannt wird, wird dies über `STATUS` und den `TIMEOUT`-Adapter signalisiert.

Anwendungsszenarien

- **ISOBUS-Steuergeräte (TECU)** – Integration in landwirtschaftliche Maschinen, um die von der Zugmaschine oder dem Gerät bereitgestellte Geschwindigkeit auszulesen.
- **Fahrgeschwindigkeitsregelung** – Verwendung der selektierten Geschwindigkeit für teilflächenspezifische Applikationen (z. B. Düngung, Pflanzenschutz).
- **Distanz- und Wegmessung** – Auswertung der zurückgelegten Distanz zur Steuerung von Arbeitsbreiten oder Dokumentationszwecken.
- **Überwachung der Kommunikation** – Erkennung von Timeout-Situationen im ISOBUS-Netzwerk durch den `TIMEOUT`-Adapter.

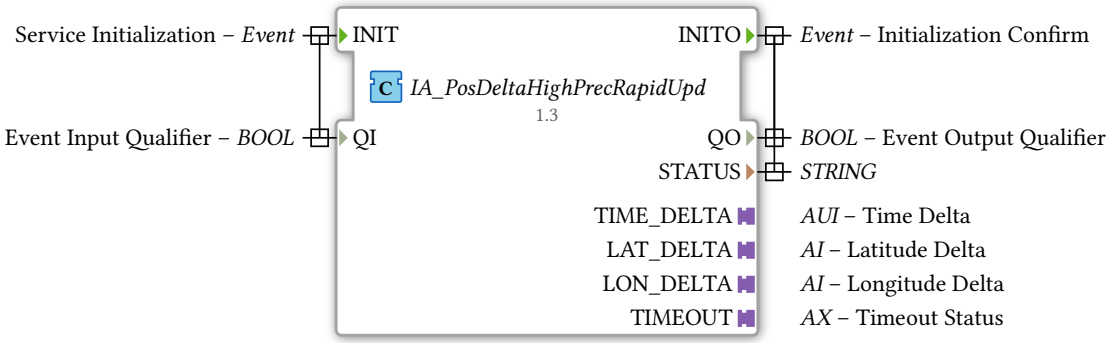
Vergleich mit ähnlichen Bausteinen

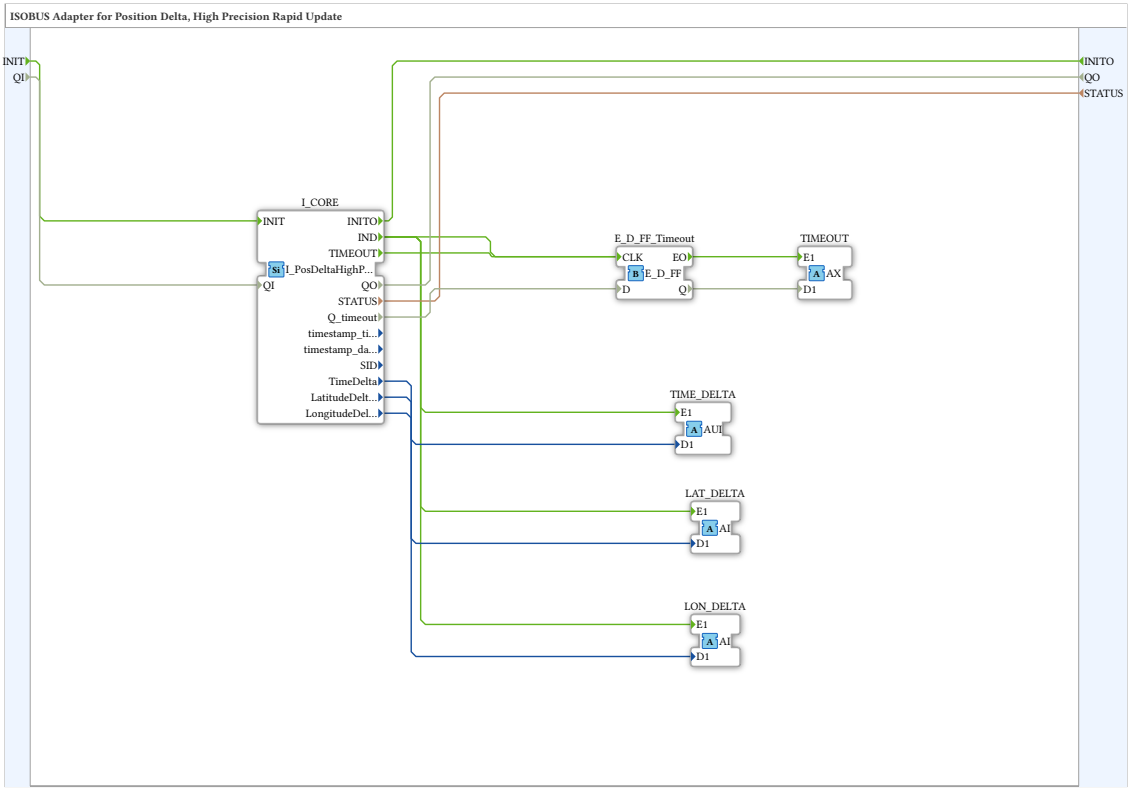
- **I_MSS (direkt)** – Der interne Baustein `I_CORE` vom Typ `isobus::tecu::I_MSS` bietet die gleiche Funktionalität, aber ohne Adapter-Schnittstellen. `IA_MSS` erleichtert die Einbindung in Systeme, die auf standardisierten Adaptern (AUI, AUDI, AX) basieren.
- **Andere ISOBUS-Adapter** – Es existieren zahlreiche Adapter für andere PGNs (z. B. für GPS, Arbeitsbreite, Motordrehzahl). `IA_MSS` ist spezifisch auf die maschinenselektierte Geschwindigkeit ausgelegt.

Fazit

Der **IA_MSS**-Baustein ist ein praktischer Adapter-Wrapper für den ISOBUS-Datensatz „Machine Selected Speed“. Er abstrahiert die Kommunikation mit dem ISOBUS-Protokoll und stellt die relevanten Größen (Geschwindigkeit, Distanz, Timeout) über standardisierte unidirektionale Adapter bereit. Dadurch wird eine einfache und wiederverwendbare Integration in IEC 61499-Anwendungen für landwirtschaftliche Maschinen ermöglicht. Der Baustein ist normkonform, robust durch Timeout-Überwachung und eignet sich besonders für den Einsatz in TECU-basierten Steuerungen.

IA_PosDeltaHighPrecRapidUpd





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `IA_PosDeltaHighPrecRapidUpd` ist ein ISOBUS-Adapterbaustein für hochpräzise, schnell aktualisierte Positionsdeltas gemäß dem NMEA 2000 Standard (PGN 129027). Er kapselt den internen Kernbaustein `I_PosDeltaHighPrecRapidUpd` und stellt dessen Ausgangsdaten über Adapterschnittstellen zur Verfügung. Dadurch können Positionsänderungen (Zeit-, Breiten- und Längengraddeltas) sowie ein Timeout-Status nahtlos in Automatisierungssysteme eingebunden werden.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung
`INIT`	EInit	Initialisierung des Bausteins; wird mit `QI` verwendet.

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung
`INITO`	EInit	Bestätigung der Initialisierung; liefert `QO` und `STATUS`.

Daten-Eingänge

Name	Typ	Beschreibung
`QI`	`BOOL`	Qualifikator für INIT-Ereignis.

Daten-Ausgänge

Name	Typ	Beschreibung
`QO`	`BOOL`	Ausgangsqualifikator (Initialisierung erfolgreich?).

STATUS	STRING	Statusmeldung zur Initialisierung.

Adapter

Adapterbezeichnung	Typ (Adapter)	Beschreibung
TIME_DELTA	adapter::types::unidirectional::AUI	Liefert das Zeitdelta (unsigned integer).
LAT_DELTA	adapter::types::unidirectional::AI	Liefert das Breitengraddelta (signed integer).
LON_DELTA	adapter::types::unidirectional::AI	Liefert das Längengraddelta (signed integer).
TIMEOUT	adapter::types::unidirectional::AX	Zeigt an, ob ein Timeout aufgetreten ist (Bool).

Funktionsweise

Der Baustein arbeitet als **Wrapper** um den internen FB `I_PosDeltaHighPrecRapidUpd`. Bei einem `INIT`-Ereignis mit gesetztem `QI` wird der Kern initialisiert. Nach erfolgreicher Initialisierung meldet sich der Kern mit `INITO` und liefert `QO = TRUE` sowie einen passenden Status. Sobald neue Positionsdaten eintreffen, erzeugt der Kern das Ereignis `IND` (intern, nicht direkt sichtbar), das die Daten an die drei Delta-Adapter weitergibt. Gleichzeitig wird bei Timeout das Ereignis `TIMEOUT` ausgelöst und der entsprechende Adapter aktualisiert.

Die internen Datenverbindungen stellen sicher, dass die Werte von `TimeDelta`, `LatitudeDelta`, `LongitudeDelta` und `Q_timeout` auf die entsprechenden Adapterdatenport `D1` übertragen werden.

Technische Besonderheiten

- **Standardkonformität**: Der Baustein realisiert den NMEA 2000 PGN 129027 – „Position Delta, High Precision Rapid Update“.
- **Adapterbasiert**: Die Ausgangsdaten werden über flexible, unidirektionale Adapter bereitgestellt, sodass sie direkt mit anderen Bausteinen eines 4diac-Systems verbunden werden können.
- **Hohe Präzision & Aktualisierungsrate**: Optimiert für Anwendungen, die genaue und schnelle Positionsänderungen benötigen (z. B. in der Landwirtschaft oder Fahrzeugnavigation).
- **Initialisierung mit Status**: Der Baustein liefert eine explizite Initialisierungsbestätigung (QO, STATUS) und kann so in sicherheitskritische Abläufe eingebunden werden.

Zustandsübersicht

Der Baustein besitzt keinen expliziten endlichen Automaten (ECC) in der XML-Definition. Sein Verhalten wird vollständig durch den internen Kern-FB `I_PosDeltaHighPrecRapidUpd` gesteuert:

1. **Initialisierung abwarten**: Nach Systemstart liegt `QI = FALSE` an; `INIT` wird nicht ausgelöst.
2. **Initialisierung starten**: Bei `INIT`-Ereignis mit `QI = TRUE` wird der Kern aktiviert.
3. **Initialisierung bestätigt**: Der Kern meldet `INITO` mit `QO = TRUE` und positivem Status; anschließend beginnt die Datenbereitstellung.
4. **Datenaktualisierung**: Bei jedem neuen Positionsdelta werden die Adapter automatisch mit den aktuellen Werten versorgt.
5. **Timeout**: Falls keine gültigen Daten mehr empfangen werden, setzt der Kern `Q_timeout = TRUE` und löst das `TIMEOUT`-Ereignis aus.

Anwendungsszenarien

- **Präzisionslandwirtschaft**: Überwachung von Positionsänderungen eines Traktors oder einer Landmaschine in Echtzeit (z. B. für Lenksysteme oder Applikationskarten).
- **Fahrzeugnavigation**: Verarbeitung von hochgenauen GNSS-Deltas in autonomen Fahrzeugen.
- **ISOBUS-Integration**: Anbindung an bestehende ISOBUS-Steuergeräte, die auf PGN 129027 basieren.
- **Sicherheitsüberwachung**: Erkennung von Datenausfällen durch den Timeout-Adapter.

Vergleich mit ähnlichen Bausteinen

Ähnliche ISOBUS-Adapter-Bausteine (z. B. `IA_PosSpeed` oder `IA_TimeDate`) kapseln ebenfalls spezifische PGNs und stellen deren Daten über Adapter bereit. Der `IA_PosDeltaHighPrecRapidUpd` unterscheidet sich durch:

- Spezialisierung auf **Positionsdeltas** (nicht absolute Positionen)
- Unterstützung von **Zeit-, Breiten- und Längengraddelta** sowie separatem **Timeout**-Signal
- Ausrichtung auf hohe **Präzision** und schnelle **Aktualisierungsraten**

Andere Bausteine liefern oft nur eine einzelne Messgröße oder haben keine Timeout-Überwachung.

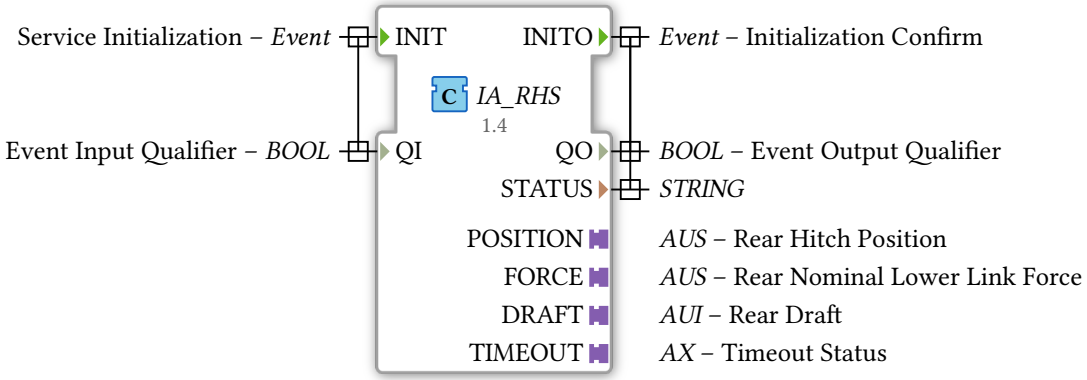
Fazit

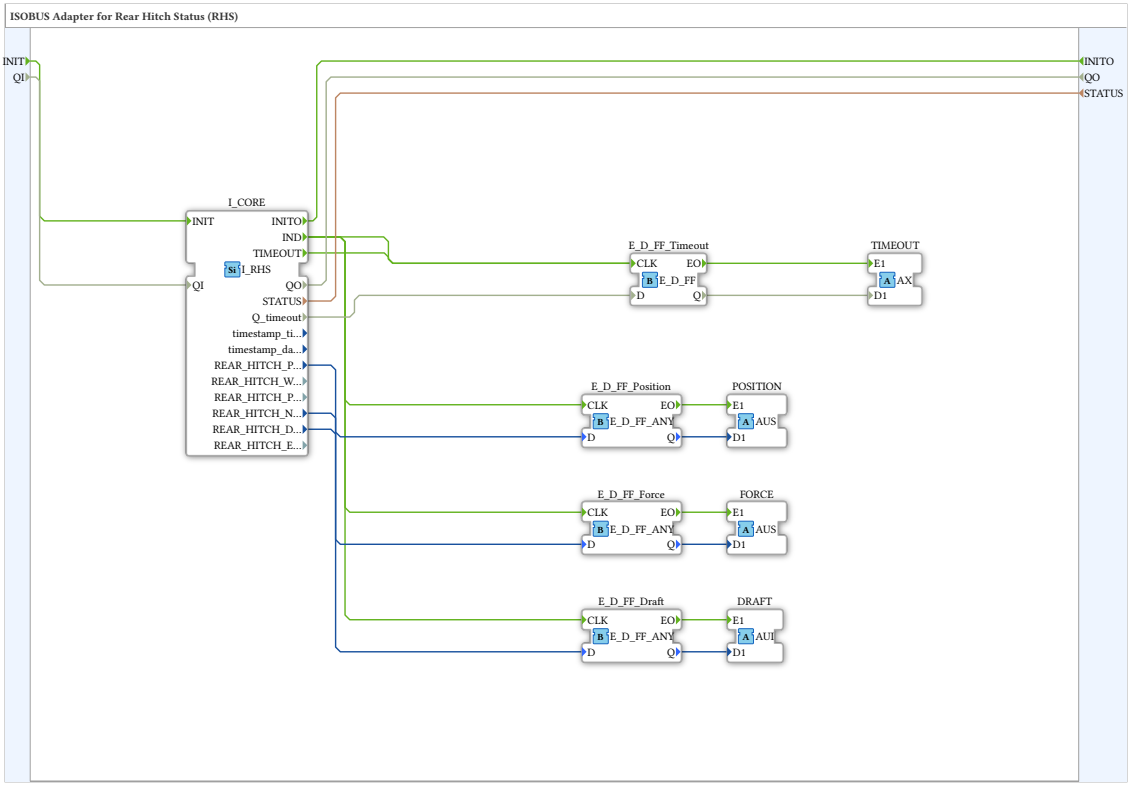
Der `IA_PosDeltaHighPrecRapidUpd` ist ein spezialisierter ISOBUS-Adapter für hochpräzise Positionsänderungen gemäß NMEA 2000 PGN 129027. Er bietet eine standardkonforme, adapterbasierte Schnittstelle, die sich nahtlos in 4diac-Systeme einfügt. Dank der integrierten Timeout-Erkennung und der expliziten Initialisierungsbestätigung eignet er sich besonders für sicherheitsrelevante Automatisierungslösungen in der Landwirtschaft und Fahrzeugtechnik.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

IA_RHS





• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock **IA_RHS** (ISOBUS Adapter for Rear Hitch Status) dient als Schnittstelle zwischen dem ISOBUS-System (gemäß ISO 11783-7, PGN 65093) und einem Anwendungsnetzwerk. Er kapselt die Kommunikation mit dem internen Baustein I_RHS und stellt die gemessenen Heckkraftheber-Daten (Position, Kraft, Zugkraft) sowie den Timeout-Status über standardisierte, unidirektionale Adapter zur Verfügung. Der Baustein wird hauptsächlich in Traktor- und Landmaschinensteuerungen eingesetzt.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar	Mit Daten
INIT	EInit	Service Initialization	QI

Ereignis-Ausgänge

Name	Typ	Kommentar	Mit Daten
INITO	EInit	Initialization Confirm	QO, STATUS

Daten-Eingänge

Name	Typ	Kommentar
QI	BOOL	Event Input Qualifier

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Event Output Qualifier

STATUS	STRING	Statusmeldung

Adapter

Name	Typ	Kommentar
POSITION	AUS	Rear Hitch Position
FORCE	AUS	Rear Nominal Lower Link Force
DRAFT	AUI	Rear Draft
TIMEOUT	AX	Timeout Status

Funktionsweise

Der Baustein wird durch das Ereignis **INIT** aktiviert. Dabei wird das Qualifier-Signal **QI** an den internen Funktionsblock **I_RHS** weitergeleitet. Nach erfolgreicher Initialisierung (oder einem Fehler) meldet sich der Baustein über das Ereignis **INITO** zurück und gibt die Ausgangssignale **QO** und **STATUS** aus. Gleichzeitig werden die vom I_RHS gelieferten Daten (Position, Unterlenkerkraft, Zugkraft) auf die entsprechenden Adapter **POSITION**, **FORCE** und **DRAFT** gelegt. Der Timeout-Status des I_RHS wird über den Adapter **TIMEOUT** ausgegeben. Die Adapter verwenden jeweils das unidirektionale Datenformat (z. B. AUS für analoge, unskalierte Werte). Die Ausgabe auf die Adapter erfolgt jeweils mit einem eigenen Ereignis (über interne Verbindungen von I_CORE.IND). Somit kann ein empfangender Baustein auf das Eintreffen neuer Daten reagieren.

Technische Besonderheiten

- Der Baustein basiert auf dem ISOBUS-Standard ISO 11783-7 und verwendet die PGN 65093.
- Die internen Berechnungen und die ISOBUS-Kommunikation werden vollständig vom Baustein **I_RHS** durchgeführt. IA_RHS dient ausschließlich als Adapter-Wrapper zur Bereitstellung der Daten an das 4diac-Netzwerk.
- Alle vier Adapter sind unidirektional (Ausgang), sodass nur Daten gesendet, aber nicht vom angeschlossenen Baustein empfangen werden.
- Die Datenausgabe auf die Adapter erfolgt zeitgleich mit dem Ereignis **IND** (intern), das vom I_RHS ausgeht.
- Timeout-Informationen werden getrennt vom Hauptdatenpfad bereitgestellt.

Zustandsübersicht

Der Baustein besitzt keine explizit modellierten Zustände. Sein Verhalten wird durch den internen Baustein I_RHS bestimmt:

- **Initialisierung (INIT)**: Der Qualifier QI steuert die Aktivierung.
- **Betrieb**: Nach erfolgreicher Initialisierung liefert I_RHS zyklisch oder ereignisgesteuert neue Daten an die Adapter.
- **Fehler/Timeout**: Ein Timeout wird über den Adapter TIMEOUT signalisiert, der Status über den Ausgang **STATUS** als Text.

Anwendungsszenarien

- **Traktorsteuerung mit ISOBUS-Anbindung**: Einlesen der Heckkraftheberdaten (Hitch Status) von einem ISOBUS-fähigen Terminal oder Steuergerät und Weiterleitung an eine übergeordnete Steuerung.
- **Landwirtschaftliche Arbeitsgeräte**: Anbindung von Anbaugeräten, die die Heckkraftheberposition oder -kraft benötigen (z. B. Pflugtiefenregelung).
- **Diagnose und Überwachung**: Erfassung von Timeout-Status und Systemzustand für Wartungs- und Fehleranalysen.

Vergleich mit ähnlichen Bausteinen

Im Unterschied zu einem allgemeinen ISOBUS-Empfänger (z. B. einem rohen PGN-Parser) abstrahiert **IA_RHS** die ISOBUS-spezifische Nachrichtenverarbeitung und stellt die Daten in normierten Adapterformaten bereit. Gegenüber einem direkten I_RHS-Baustein bietet IA_RHS den Vorteil, dass die Datenausgabe über separate Adapter erfolgt, was eine lose Kopplung im Netzwerk und einfachere

Wiederverwendung ermöglicht. Bausteine wie **IA_RHC** (Rear Hitch Control) oder **IA_PTO** folgen einem ähnlichen Adapter-Konzept, sind jedoch für andere ISOBUS-Funktionen ausgelegt.

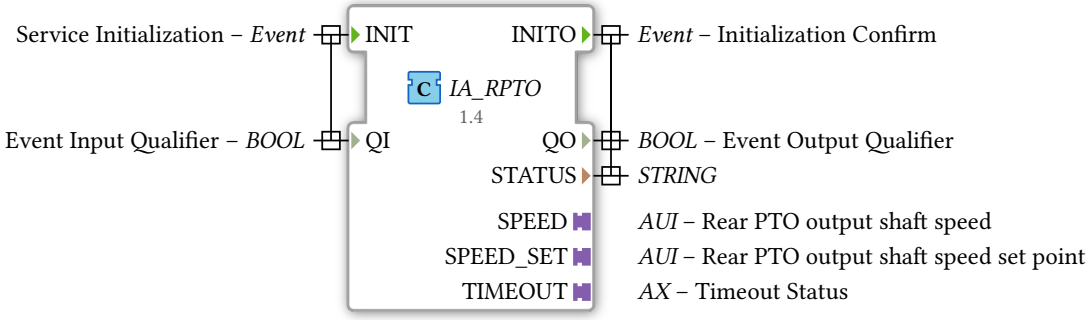
Fazit

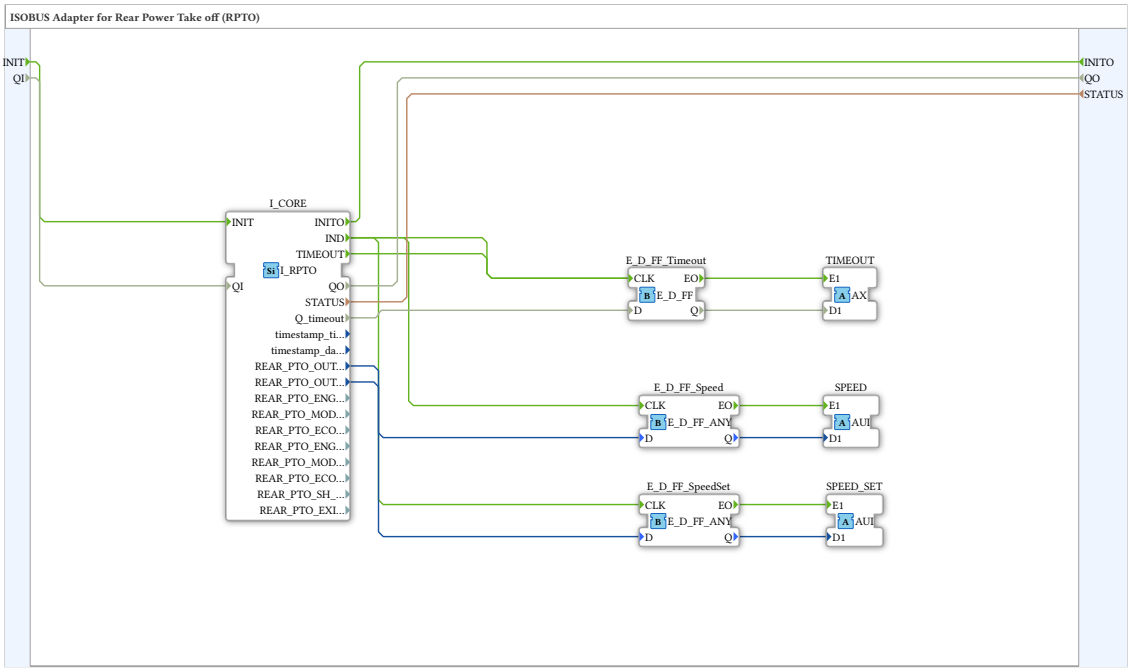
Der IA_RHS-Baustein ist eine praktische, standardkonforme Kapselung des Heckkraftheber-Status nach ISO 11783-7. Er vereinfacht die Integration von ISOBUS-Daten in ein 4diac-basiertes Steuerungssystem, indem er die komplexe Busschnittstelle hinter einfach nutzbaren Adaptern verbirgt. Die Aufteilung in mehrere Datenadapter sowie die Bereitstellung eines Timeout-Status machen ihn flexibel einsetzbar für verschiedene Anwendungen der Agrartechnik.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

IA_RPTO





Einleitung

Der Baustein IA_RPTO ist ein ISOBUS-Adapter für die hintere Zapfwelle (Rear Power Take Off, RPTO) gemäß ISO 11783-7 (PGN 65091). Er stellt eine gekapselte Schnittstelle zu einem inneren I_RPTO-Funktionsblock bereit und gibt die wichtigsten Größen (Drehzahl, Sollwert, Timeout-Status) über Adapter-Ports nach außen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar	Mit Daten
INIT	EInit	Service Initialization	QI

Ereignis-Ausgänge

Name	Typ	Kommentar	Mit Daten
INITO	EInit	Initialization Confirm	QO, STATUS

Daten-Eingänge

Name	Typ	Kommentar
QI	BOOL	Event Input Qualifier

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Event Output Qualifier
STATUS	STRING	Statusmeldung

Adapter

Name	Typ	Kommentar
SPEED	adapter::types::unidirectional::AUI	Rear PTO output shaft speed
SPEED_SET	adapter::types::unidirectional::AUI	Rear PTO output shaft speed set point
TIMEOUT	adapter::types::unidirectional::AX	Timeout Status

Funktionsweise

Der Baustein instanziiert intern einen `I_RPTO`-Block, der die eigentliche ISOBUS-Kommunikation für die hintere Zapfwelle durchführt. Über ein INIT-Ereignis wird die Initialisierung angestoßen; bei erfolgreicher Ausführung quittiert INITO mit QO = TRUE und einem passenden STATUS. Die Werte `REAR_PTO_OUTP_SHAFT_SPEED`, `REAR_PTO_OUTP_SH_SPEED_SET` und `Q_timeout` des inneren Blocks werden direkt auf die entsprechenden Adapter-Ports SPEED, SPEED_SET und TIMEOUT weitergeleitet.

Technische Besonderheiten

– Der Baustein arbeitet nach ISO 11783-7 (PGN 65091) und ist für den Einsatz in landwirtschaftlichen Steuergeräten (Tractor ECU) vorgesehen.
– Alle Adapter sind unidirektional (AUI, AX), d.h. sie liefern Daten nur in eine Richtung.
– Der innere Block stammt aus dem Paket `isobus::tecu`.
– Lizenz: Eclipse Public License 2.0.

Zustandsübersicht

Der Funktionsblock kennt zwei Hauptzustände: **nicht initialisiert** und **initialisiert**. Nach Empfang von INIT führt der innere Block die Initialisierung durch. Bei Erfolg wird QO = TRUE gesetzt und der STATUS zeigt einen positiven Wert. Tritt ein Fehler oder Timeout auf, wird dies über den TIMEOUT-Adapter und den STATUS signalisiert. Ein detailliertes Zustandsdiagramm liegt nicht als Teil der äußeren Beschreibung vor, da die komplette Logik im inneren Block implementiert ist.

Anwendungsszenarien

– Integration einer ISOBUS-kompatiblen Hinterzapfwelle in ein Traktorensteuergerät.
– Überwachung und Regelung der Zapfwellendrehzahl über das ISOBUS-Netzwerk.
– Zeitüberwachung der Datenkommunikation zur Erkennung von Verbindungsabbrüchen.

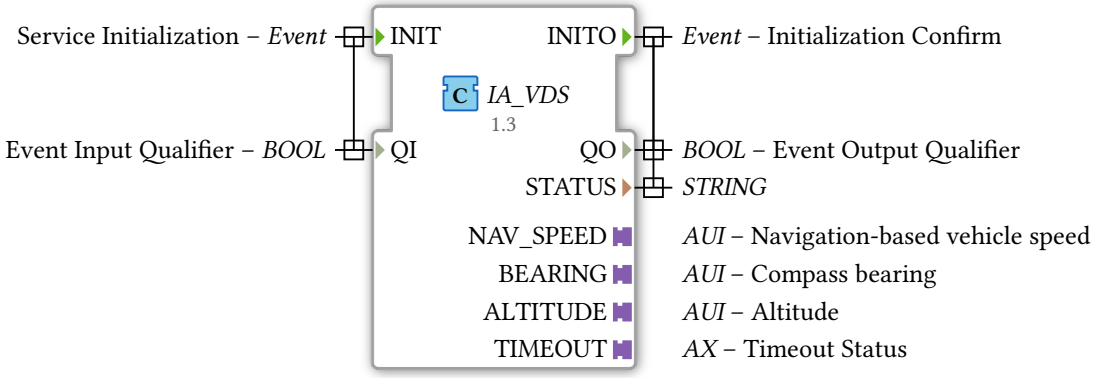
Vergleich mit ähnlichen Bausteinen

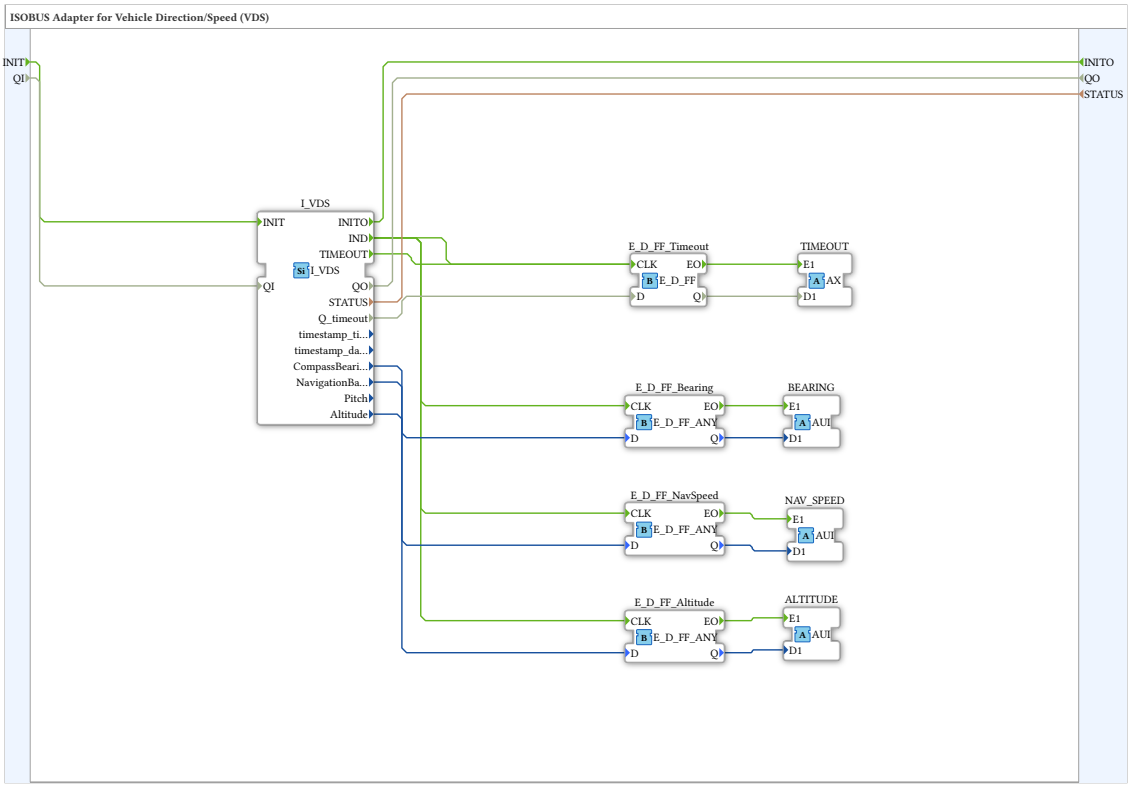
Verwandte Adapter-Bausteine existieren für andere ISOBUS-Funktionen, z.B. IA_FPTO (vordere Zapfwelle), IA_HITCH (Hubwerk) oder IA_TSC (Traktorsteuerung). Der wesentliche Unterschied liegt im verwendeten PGN (65091) und den spezifischen Datenpunkten (Drehzahl, Sollwert, Timeout). IA_RPTO bietet somit eine auf die hintere Zapfwelle zugeschnittene Schnittstelle.

Fazit

IA_RPTO kapselt die ISOBUS-Kommunikation für die hintere Zapfwelle und stellt die relevanten Informationen über klar definierte Adapter-Ports zur Verfügung. Der Baustein vereinfacht die Einbindung in IEC 61499-Applikationen und gewährleistet eine normkonforme Anbindung gemäß ISO 11783-7.

IA_VDS





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **IA_VDS** ist ein ISOBUS-Adapter für die Fahrzeugrichtung und -geschwindigkeit (Vehicle Direction/Speed) gemäß ISO 11783-7 (PGN 65256). Er kapselt den Zugriff auf die entsprechenden CAN-Bus-Daten und stellt sie als zeitgemäße 4diac-Adapter-Schnittstellen zur Verfügung. Der Baustein wird über ein INIT-Ereignis initialisiert und liefert anschließend kontinuierlich Navigationsgeschwindigkeit, Kompassrichtung, Höhe sowie einen Timeout-Status.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung
INIT	EInit	Initialisiert den Baustein. Erwartet den Qualifier QI.

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung
INITO	EInit	Bestätigt die erfolgreiche Initialisierung. Liefert QO und STATUS.

Daten-Eingänge

Name	Typ	Beschreibung
QI	BOOL	Event-Input Qualifier (TRUE = Initialisierung aktiv).

Daten-Ausgänge

Name	Typ	Beschreibung
QO	BOOL	Event-Output Qualifier (TRUE = Initialisierung ok).
STATUS	STRING	Meldungstext (z. B. Fehlerbeschreibung oder “Ready”).

Adapter

Adapter	Typ	Beschreibung
NAV_SPEED	AUI	Navigationsbasierte Fahrzeuggeschwindigkeit (unidirektional, Empfänger).
BEARING	AUI	Kompassrichtung (unidirektional, Empfänger).
ALTITUDE	AUI	Höhe über dem Meeresspiegel (unidirektional, Empfänger).
TIMEOUT	AX	Timeout-Status (unidirektional, Empfänger).

Alle Adapter sind als Plugs (out-going) realisiert – der IA_VDS sendet die Daten an die angeschlossenen Senken.

Funktionsweise

Der Baustein enthält intern einen weiteren Funktionsblock **I_VDS** (aus der Bibliothek `isobus::tecu`), der die eigentliche ISOBUS-Kommunikation abwickelt. Beim Eintreffen des INIT-Ereignisses wird die Initialisierung von I_VDS angestoßen. Nach erfolgreicher Initialisierung meldet I_VDS dies über sein INITO-Ereignis, welches an den INITO-Ausgang des IA_VDS weitergeleitet wird. Zeitgleich beginnt I_VDS, zyklisch die Werte für NavigationBasedVehicleSpeed, CompassBearing und Altitude zu aktualisieren. Diese werden über die zugehörigen Adapter (NAV_SPEED, BEARING, ALTITUDE) ausgegeben, sobald das Ereignis IND von I_VDS eintrifft. Tritt ein Timeout auf (z. B. CAN-Bus-Fehler), wird dies über den Adapter TIMEOUT gemeldet.

Technische Besonderheiten

- Der Baustein setzt auf dem **Eclipse Public License 2.0 (EPL-2.0)** auf und ist als ISOBUS-Adapter für PGN 65256 konzipiert.

- Die Daten werden asynchron über eigene Ereignisse (IND) an die Adapter-Ausgänge verteilt, was eine entkoppelte Verarbeitung in der 4diac-Umgebung ermöglicht.
- Der interne Block I_VDS ist in der Bibliothek `isobus::tecu` implementiert und für den direkten Buszugriff optimiert.
- Der Timeout-Status wird mit einem eigenen Adapter (Typ AX) bereitgestellt, um Fehlerbehandlung klar vom normalen Datenstrom zu trennen.

Zustandsübersicht

Der IA_VDS besitzt keinen expliziten Zustandsautomaten, sein Verhalten wird durch den internen I_VDS bestimmt:

1. **Initial** – Noch kein INIT empfangen.
2. **Initialisierung läuft** – Nach INIT, solange I_VDS noch nicht bereit.
3. **Aktiv** – Nach erfolgreicher Initialisierung (INITO = TRUE). Werte werden regelmäßig aktualisiert.
4. **Fehler** – Bei Timeout oder ungültigen Busdaten (STATUS enthält Fehlermeldung, QO = FALSE).

Anwendungsszenarien

- **Landmaschinensteuerung** – Integration von Fahrgeschwindigkeit, Fahrtrichtung und Höhe in Steuerungslogiken (z. B. für teilflächenspezifische Applikation).
- **ISOBUS-Netzwerkanbindung** – Als Adapter in 4diac-Applikationen, die ISOBUS-Daten von einem Traktor oder Erntefahrzeug nutzen.
- **Telemetrie-Systeme** – Weiterleitung der VDS-Daten an übergeordnete Systeme (Cloud, Display).

Vergleich mit ähnlichen Bausteinen

- **I_TC** (ISOBUS Traktorsteuerung) – Liefert allgemeine Traktordaten (Geschwindigkeit, Zapfwellendrehzahl, Hubwerk) über ähnliche Adapter, aber nicht spezifisch die VDS-PGN.
- **I_GNSS** (ISOBUS GNSS-Adapter) – Stellt GPS-basierte Positionsdaten bereit, jedoch ohne Fokus auf Richtung/Geschwindigkeit über CAN.
- **IA_VDS** ist auf die reine Fahrzeugrichtungs-/Geschwindigkeitsinformation (PGN 65256) spezialisiert und besonders schlank in der Schnittstelle.

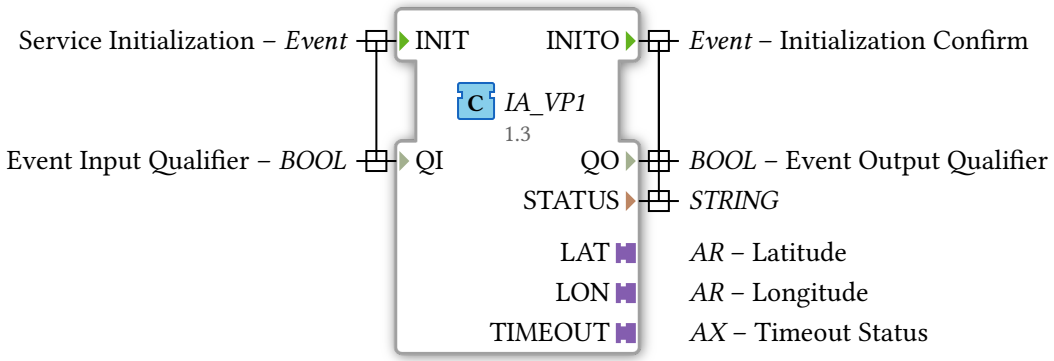
Fazit

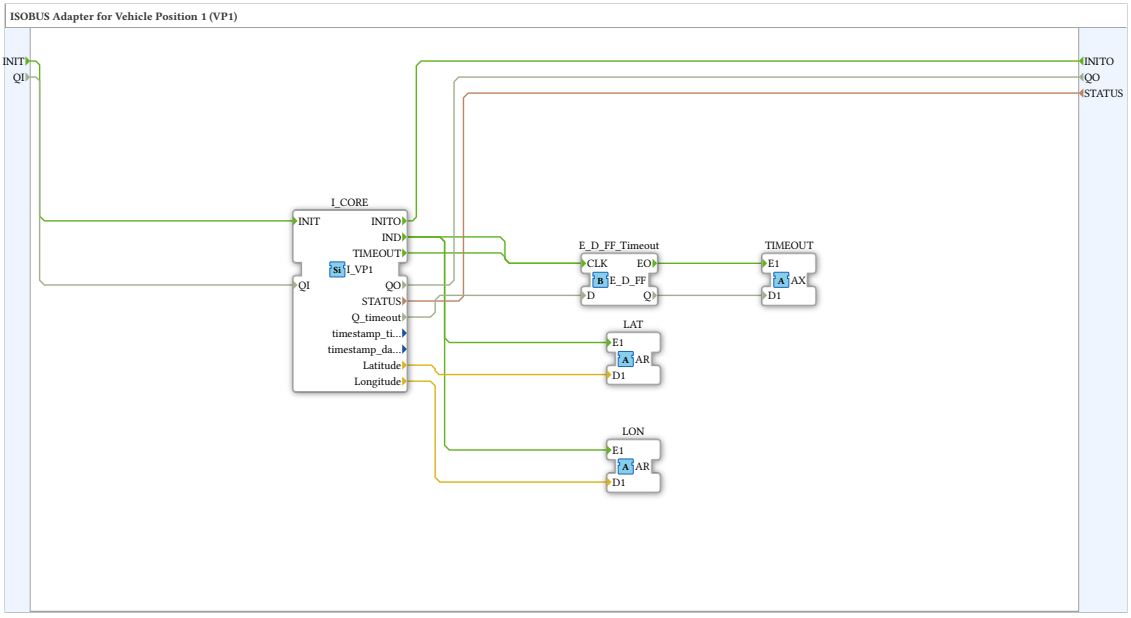
Der **IA_VDS** ist ein praktischer Baustein, um die ISO-11783-7-spezifischen Fahrzeugdaten (Richtung/Geschwindigkeit) einfach in eine 4diac-Applikation einzubinden. Durch die adapterbasierte Ausgabe bleibt die Architektur modular und erweiterbar. Die Unterteilung in Daten- und Timeout-Adapter erleichtert die robuste Fehlerbehandlung. Ideal für alle Anwendungen, die eine präzise, busbasierte Fahrzeugdynamik benötigen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

IA_VP1





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **IA_VP1** ist ein ISOBUS-Adapter für die Vehicle Position 1 (VP1) gemäß ISO 11783-7 (PGN 65267). Er kapselt die Kommunikation zur Fahrzeugposition und stellt die empfangenen Daten über unidirektionale Adapter-Schnittstellen für die weitere Verarbeitung in der 4diac-IDE bereit.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar
INIT	EInit	Service Initialization

Ereignis-Ausgänge

Ereignis	Typ	Kommentar
INITO	EInit	Initialization Confirm

Daten-Eingänge

Variable	Typ	Kommentar
QI	BOOL	Event Input Qualifier

Daten-Ausgänge

Variable	Typ	Kommentar
QO	BOOL	Event Output Qualifier
STATUS	STRING	Statusinformation

Adapter

Adapter	Typ	Kommentar
LAT	adapter::types::unidirectional::AR	Latitude

LON	adapter::types::unidirectional::AR	Longitude
TIMEOUT	adapter::types::unidirectional::AX	Timeout Status

Funktionsweise

Der Baustein enthält intern eine Instanz des Funktionsblocks `I_VP1` aus der Bibliothek `isobus::tecu`. Beim Eintreffen des INIT-Ereignisses wird die Initialisierung angestoßen – der Qualifier `QI` bestimmt, ob die Initialisierung aktiv erfolgt. Nach erfolgreicher Initialisierung wird das INITO-Ereignis ausgelöst, begleitet von den Ausgangsdaten `QO` und `STATUS`.

Die eigentlichen Positionsdaten (Breitengrad und Längengrad) sowie der Timeout-Status werden über die unidirektionalen Adapter ausgegeben. Jeder Adapter wird durch ein Ereignis (z. B. `IND` vom Kernbaustein) getriggert und stellt den zugehörigen Datenwert (`D1`) an der Adapter-Schnittstelle bereit.

Technische Besonderheiten

- **Unidirektionale Adapter:** Die Schnittstellen LAT und LON verwenden den Typ `AR`, der nur einen Ausgangs-Datenwert liefert. Der TIMEOUT-Adapter verwendet den Typ `AX` für einen booleschen Timeout-Status.
- **ISOBUS-Konformität:** Der Baustein implementiert das Vehicle Position 1 (VP1)-Nachrichtenformat gemäß ISO 11783-7.
- **Latitude-/Longitude-Bereitstellung:** Die Werte werden direkt vom internen Kernbaustein durchgereicht und sind als Gleitkommazahlen im Adapter verfügbar.
- **Timeout-Überwachung:** Der TIME OUT-Adapter signalisiert, ob die Positionsdaten innerhalb einer erwarteten Zeit eingetroffen sind.

Zustandsübersicht

Der Baustein besitzt keine explizit dargestellte Zustandsmaschine. Das Verhalten ist ereignisgesteuert:

- **Inaktiv:** Vor dem INIT-Ereignis liegt keine Initialisierung vor.
- **Initialisierung:** Sobald INIT mit `QI=TRUE` eintrifft, wird die Initialisierungssequenz gestartet.
- **Aktiv:** Nach erfolgreichem INITO stehen die Positions- und Timeout-Daten an den Adaptern zur Verfügung. Die Daten werden mit jedem neuen internen Ereignis (getriggert durch den Kern) aktualisiert.
- **Fehlerfall:** Der Status-String und `QO` geben Aufschluss über Fehler oder Timeout.

Anwendungsszenarien

- **Landwirtschaftliche Fahrzeuge:** Auslesen der aktuellen GPS-Position (Breite/Länge) über den ISOBUS-Bordcomputer.
- **Präzisionslandwirtschaft:** Verwendung der Positionsdaten zur Steuerung von Teilbreitenschaltungen, variabler Ausbringung oder Spurführung.
- **Dokumentation und Telemetrie:** Aufzeichnung von Fahrzeugbewegungen und Positionsdaten für Flottenmanagement.

Vergleich mit ähnlichen Bausteinen

- **IA_VP0, IA_GP, IA_PD:** Andere ISOBUS-Positions- und Datenadapter (z. B. für Geschwindigkeit, Maschinenposition). Im Gegensatz dazu fokussiert IA_VP1 auf die reine Vehicle Position (Breitengrad, Längengrad) und den Timeout-Status. Die Daten werden hier als unidirektionale Adapter und nicht als einfache Datenausgänge bereitgestellt.
- **I_VP1 (Kernbaustein):** Der interne Kernbaustein bietet dieselbe Funktionalität, jedoch ohne die Adapter-Kapselung. IA_VP1 macht die Schnittstellen für die modulare 4diac-Umgebung zugänglicher.

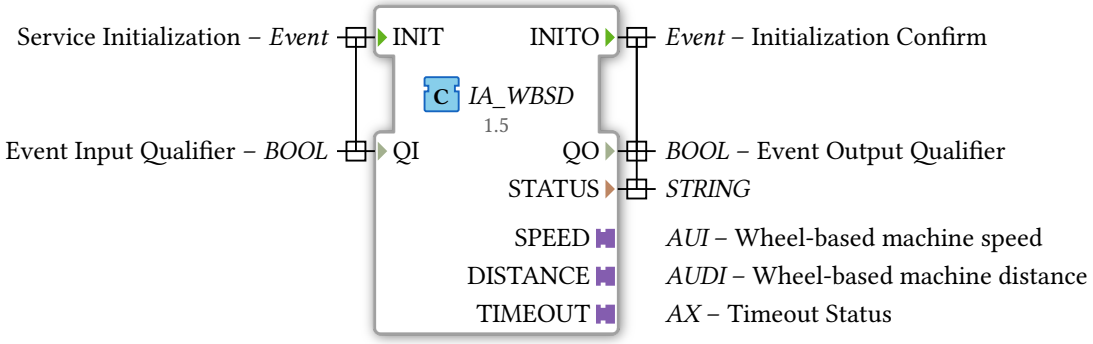
Fazit

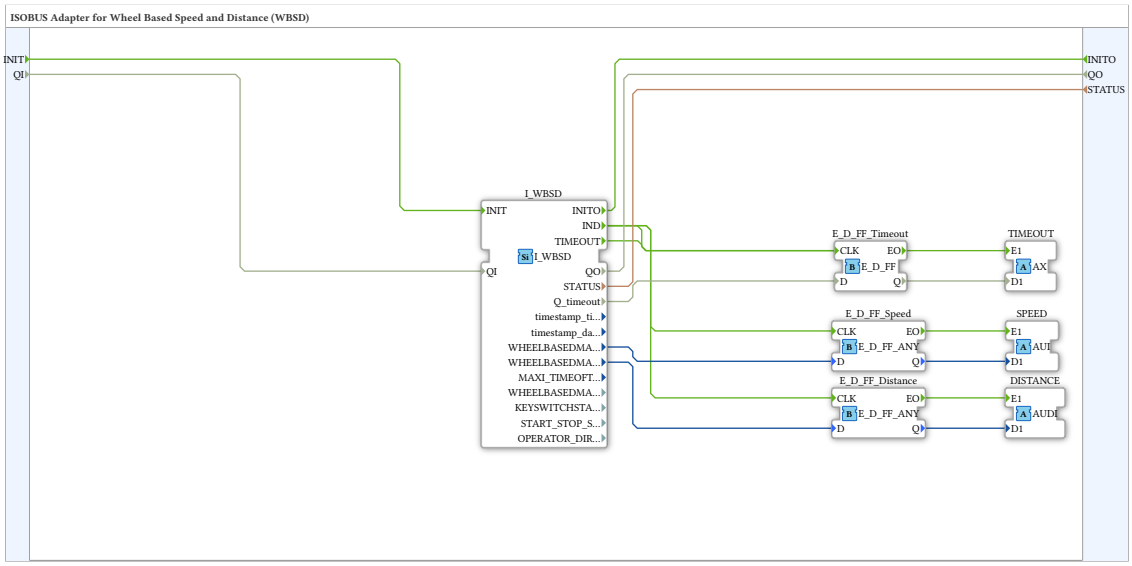
Der Funktionsblock **IA_VP1** ist ein praktischer und normgerechter Baustein, um die Fahrzeugposition aus dem ISOBUS in die 4diac-Welt zu integrieren. Die Verwendung von unidirektionalen Adaptern ermöglicht eine saubere Trennung und einfache Einbindung in größere Automatisierungsnetzwerke. Dank der Standardkonformität und der einfachen Initialisierung eignet er sich hervorragend für landwirtschaftliche Steuerungen und IoT-Anwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

IA_WBSD





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **IA_WBSD** (ISOBUS Adapter for Wheel Based Speed and Distance) dient als Schnittstellen-Wrapper für die ISOBUS-Kommunikation radbasierter Geschwindigkeits- und Distanzwerte gemäß ISO 11783-7. Er kapselt den internen Baustein `I_WBSD` und stellt dessen Funktionalität über standardisierte Adapter-Anschlüsse bereit.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar
INIT	EInit	Service Initialization

Ereignis-Ausgänge

Ereignis	Typ	Kommentar
INITO	EInit	Initialization Confirm

Daten-Eingänge

Name	Typ	Kommentar
QI	BOOL	Event Input Qualifier (Initialisierungsfreigabe)

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Event Output Qualifier (Initialisierungsergebnis)
STATUS	STRING	Statusmeldung (z. B. Fehler oder Erfolg)

Adapter

Adapter	Typ	Kommentar
SPEED	adapter::types::unidirectional::AUI	Wheel-based machine speed (unidirektional, analoger Wert)
DISTANCE	adapter::types::unidirectional::AUDI	Wheel-based machine distance (unidirektional)
TIMEOUT	adapter::types::unidirectional::AX	Timeout Status (unidirektional)

Funktionsweise

Der Baustein **IA_WBSD** ist ein reiner Adapter-Wrapper. Er verbindet die externen Ereignis- und Datenschnittstellen sowie die drei Adapter direkt mit dem internen Funktionsblock `I_WBSD` (vom Typ `isobus::tecu::I_WBSD`).

- Über den **INIT**-Eingang und das Qualifier-Signal **QI** wird die Initialisierung des ISOBUS-Dienstes angestoßen.
- Nach erfolgreicher Initialisierung wird **INITO** ausgelöst und **QO** auf `TRUE` gesetzt. **STATUS** enthält eine textuelle Rückmeldung.
- Die Adapter **SPEED** und **DISTANCE** liefern die aktuellen Messwerte für radbasierte Geschwindigkeit bzw. Distanz.
- Der Adapter **TIMEOUT** signalisiert, ob ein Timeout in der ISOBUS-Kommunikation aufgetreten ist.

Die interne Logik (Berechnung, Timeout-Überwachung, ISOBUS-Frame-Handling) wird vollständig vom inneren FB `I_WBSD` übernommen; **IA_WBSD** stellt diese nach außen als IEC-61499-kompatible Adapter-Schnittstelle zur Verfügung.

Technische Besonderheiten

- **Standard:** ISO 11783-7 (ISOBUS) – PGN 65096
- **Entwicklungsgeschichte:**
  - Version 1.0: initiale Implementierung als Adapter-Wrapper für `I_WBSD`
  - Version 1.1: Fehlerbehebung – korrekte Verbindung zwischen internen Pins und Adapter-Plugs durch zusätzliche Konvertierungsbausteine
- **Compiler-Paket:** `isobus::tecu`
- **Lizenz:** Eclipse Public License 2.0
- **Urheber:** - (nicht genannt)

Zustandsübersicht

Der Baustein besitzt keine eigene explizite Zustandsmaschine. Das Verhalten wird durch den inneren FB `I_WBSD` gesteuert:

- **Ruhezustand:** Keine Initialisierung ausstehend.
- **Initialisierungslauf:** Nach **INIT** mit **QI** = **TRUE** wird der interne Prozess gestartet. Bei **QI** = **FALSE** erfolgt keine Aktion.
- **Initialisierung abgeschlossen: INITO** wird ausgelöst, **QO** zeigt Erfolg (`TRUE`) oder Misserfolg (`FALSE`). Danach können über die Adapter laufend Messwerte und Timeout-Status empfangen werden.

Anwendungsszenarien

- **Landwirtschaftliche Maschinensteuerung:** Einbindung radbasierter Geschwindigkeits- und Distanzsensoren in ein ISOBUS-Steuerungssystem (z. B. für Traktoren, Erntemaschinen).
- **Positionierung ohne GPS:** Nutzung der Radsensoren zur relativen Positionsbestimmung in Feldarbeitsanwendungen.

- **Überwachung von Fahrzyklen:** Erfassung von Geschwindigkeits- und Distanzdaten für Diagnose oder Prozesssteuerung.

Vergleich mit ähnlichen Bausteinen

- **IA_WBSD vs. direkter I_WBSD:** Der interne FB `I_WBSD` bietet die reine ISOBUS-Implementierung, während **IA_WBSD** diese in ein IEC-61499-Adapter-Interface kapselt. Dadurch wird die Wiederverwendung in verschiedenen Tools (z. B. 4diac-IDE) und die lose Kopplung mit anderen Bausteinen erleichtert.
- **IA_WBSD vs. andere ISOBUS-Adapter:** Spezifisch für die PGN 65096 (Wheel Based Speed and Distance). Adapter für andere PGNs (z. B. für GPS-Position oder Motor-Drehzahl) haben ähnliche Struktur, unterscheiden sich aber in den Datentypen und der internen Interpretation.

Fazit

Der **IA_WBSD** ist ein praktischer Adapter-Wrapper für den ISOBUS-Baustein `I_WBSD`. Er reduziert die Einbindungskomplexität radbasierter Geschwindigkeits- und Distanzdaten in IEC-61499-Projekte und stellt die Funktionalität über standardisierte unidirektionale Adapter bereit. Dank der Lizenz unter EPL 2.0 kann er frei in eigenen Projekten verwendet und angepasst werden. Die Version 1.1 stellt sicher, dass alle internen Verbindungen korrekt auf die Adapter-Plugs geführt sind.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

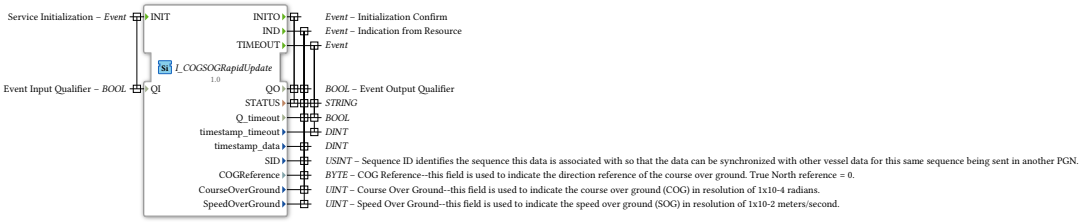
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

I_COGSOGRapidUpdate

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `I_COGSOGRapidUpdate` implementiert die Verarbeitung des NMEA 2000 Parameter Group Number (PGN) 129026 „COG & SOG, Rapid Update“. Dieser Baustein dient dem Empfang und der Bereitstellung von Navigationsdaten, speziell des momentanen Kurses über Grund (Course Over Ground, COG) und der Geschwindigkeit über Grund (Speed Over Ground, SOG) mit hoher Aktualisierungsrate. Er ist für den Einsatz in maritimen oder mobilen Arbeitsmaschinen-Steuerungssystemen konzipiert, die auf dem ISOBUS-Standard basieren.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisiert den Funktionsblock. Wird zusammen mit dem Qualifier `QI` ausgelöst.

Ereignis-Ausgänge

- **INITO**: Bestätigt die erfolgreiche Initialisierung. Löst die Ausgänge `QO` und `STATUS` aus.
- **IND**: Zeigt den erfolgreichen Empfang und die Verarbeitung neuer COG/SOG-Daten an. Löst die zugehörigen Datenausgänge aus.
- **TIMEOUT**: Wird ausgelöst, wenn ein Timeout beim Empfang der Daten auftritt.

Daten-Eingänge

- **QI** (BOOL): Qualifier für den INIT-Ereigniseingang. Steuert die Initialisierung (`TRUE` = starten).

Daten-Ausgänge

- **QO** (BOOL): Qualifier für die Ereignisausgänge INITO und IND. Zeigt den allgemeinen Betriebszustand an.
- **STATUS** (STRING): Statusmeldung, die zusätzliche Informationen (z.B. Fehlerbeschreibungen) liefert.

- **Q_timeout** (BOOL): Gibt an, ob das letzte empfangene Ereignis ein Timeout war (`TRUE`) oder gültige Daten (`FALSE`).
- **timestamp_timeout** (DINT): Zeitstempel, der dem TIMEOUT-Ereignis zugeordnet ist.
- **timestamp_data** (DINT): Zeitstempel der zuletzt empfangenen gültigen COG/SOG-Daten.
- **SID** (USINT): Sequence Identifier. Ermöglicht die Synchronisation dieser Daten mit anderen, im selben Zyklus gesendeten PGNs des Fahrzeugs.
- **COGReference** (BYTE): Referenzrichtung für den Kurs über Grund. Der Wert `0` steht für die Referenz „Wahre Nordrichtung“ (True North).
- **CourseOverGround** (UINT): Kurs über Grund (COG). Die Einheit ist 1x10-4 Radiant.
- **SpeedOverGround** (UINT): Geschwindigkeit über Grund (SOG). Die Einheit ist 1x10-2 Meter pro Sekunde.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Der Baustein agiert als passiver Empfänger für das NMEA 2000 PGN 129026. Nach der Initialisierung via `INIT` mit `QI=TRUE` wartet er auf eingehende Datenframes. Bei Empfang eines gültigen Frames werden die enthaltenen Daten (COG, SOG, Referenz, SID) dekodiert und über den `IND`-Ausgang samt den zugehörigen Datenwerten bereitgestellt. Gleichzeitig wird `Q_timeout` auf `FALSE` gesetzt. Tritt innerhalb eines konfigurierten Zeitfensters kein neuer Datenframe auf, wird ein `TIMEOUT`-Ereignis generiert und `Q_timeout` auf `TRUE` gesetzt. Der `STATUS`-Ausgang kann für Diagnosezwecke genutzt werden.

Technische Besonderheiten

- **NMEA 2000 Konformität**: Implementiert exakt die Spezifikation für PGN 129026.
- **Auflösung**: Die physikalischen Werte für Kurs und Geschwindigkeit sind in den im NMEA-Standard definierten festen Auflösungen kodiert (COG: 0.0001 rad/LSB, SOG: 0.01 m/s/LSB). Eine Umrechnung in gebräuchlichere Einheiten (Grad, Knoten) muss gegebenenfalls in nachgelagerten Blöcken erfolgen.
- **Sequenzierung**: Die `SID` unterstützt die Korrelation von Daten, die zeitgleich in verschiedenen PGNs gesendet werden.

Zustandsübersicht

1. **Inaktiv**: Vor der Initialisierung.
2. **Initialisiert/Wartend**: Nach erfolgreichem `INIT`. Der Baustein lauscht auf den CAN-Bus bzw. die entsprechende Schnittstelle auf PGN 129026.
3. **Datenempfang**: Bei Eintreffen eines Frames werden die Daten verarbeitet und ein `IND`-Ereignis generiert.
4. **Timeout**: Wird aktiv, wenn erwartete Daten nicht innerhalb der vorgegebenen Zeit eintreffen. Löst ein `TIMEOUT`-Ereignis aus.

Anwendungsszenarien

- **Maritime Navigation**: Anzeige von aktuellem Kurs und Geschwindigkeit auf einem Multifunktionsdisplay (MFD).
- **Autonome Steuerung**: Bereitstellung der grundlegenden Navigationsdaten für Autopiloten oder Routenführungsalgorithmen von Arbeitsmaschinen.
- **Datenlogging**: Protokollierung von Fahrzeugbewegungsdaten mit hoher zeitlicher Auflösung.
- **Sensorfusion**: Kombination der COG/SOG-Daten mit anderen Positions- und Bewegungssensoren (z.B. GNSS, Gyroskop) zur Verbesserung der Gesamtgenauigkeit.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Gegenüber generischen CAN-Empfangsblöcken**: `I_COGSOGRapidUpdate` ist spezialisiert auf PGN 129026. Er übernimmt die komplette Dekodierung der Rohdaten gemäß NMEA-Spezifikation und stellt die physikalischen Werte direkt bereit. Ein generischer Empfänger würde nur die Roh-Bytes liefern.
- **Gegenüber PGN 129025 (COG/SOG)**: PGN 129026 ist die „Rapid Update“-Variante, die für eine höhere Update-Rate und geringere Latenz optimiert ist, während PGN 129025 zusätzliche Felder wie Zeitstempel enthalten kann. Die Wahl des Bausteins hängt von den Anforderungen der Anwendung an Aktualität und Datenumfang ab.

🛠️ Zugehörige Übungen

- Uebung_079

Fazit

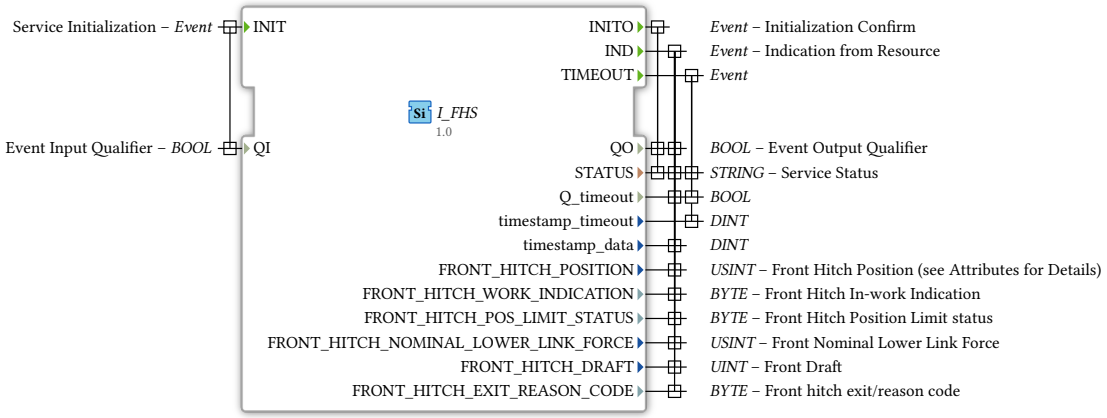
Der `I_COGSOGRapidUpdate`-Funktionsblock ist eine essentielle Komponente für jedes ISOBUS- oder NMEA 2000-basierte System, das auf präzise und schnell aktualisierte Kurs- und Geschwindigkeitsinformationen über Grund angewiesen ist. Durch seine standardkonforme Implementierung gewährleistet er zuverlässigen Datenaustausch und eine einfache Integration in übergeordnete Steuerungs- und Anzeigesysteme.

I_FHS

• – ◦ • – ◦ • – ◦ •

Einleitung

Der **I_FHS** ist ein standardkonformer Funktionsbaustein für Front-Anbaugerätestatusinformationen nach ISO 11783-7 (PGN 65094). Entwickelt unter EPL-2.0 Lizenz,ermöglicht er die Überwachung und Steuerung von Front-Anbaugeräten in modernen Agrarsystemen.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `IND`: Statusindikation
- `TIMEOUT`: Timeout-Ereignis

Daten-Eingänge

- `QI` (BOOL): Initialisierungsqualifier

Daten-Ausgänge

- `QO` (BOOL): Statusqualifier
- `STATUS` (STRING): Betriebsstatus
- `timestamp_data` (DINT): Zeitstempel der Daten (mit `IND`)
- `timestamp_timeout` (DINT): Zeitstempel des Timeouts (mit `TIMEOUT`)
- `Q_timeout` (BOOL): Timeout-Indikator

Parameter (SPN-Details)

Parameter	SPN	Typ	Länge	Skalierung	Wertebereich	Beschreibung
`FRONT_HITCH_POSITION`	1872	USINT	8 bit	0.4 %/bit	0-100%	Aktuelle Position des Front-Anbaugeräts

FRONT_HITCH_WORK_INDICATION	1876	BYTE	2 bit	4 Zustände	0-3	Arbeitsstatusindikator
FRONT_HITCH_POS_LIMIT_STATUS	1877	BYTE	3 bit	8 Zustände	0-7	Positionslimit-Status
FRONT_HITCH_NOMINAL_LOWER_LINK_FORCE	1878	SINT	8 bit	0.8 %/bit	−100% bis +100%	Normale Unterlenkerkraft
FRONT_HITCH_DRAFT	1878	UINT	16 bit	10 N/bit	−320 kN bis +320 kN	Zugkraft am Front-Anbaugerät
FRONT_HITCH_EXIT_REASON_CODE	6805	BYTE	6 bit	64 Zustände	0-63	Grundcode für Systemabbruch

Funktionsweise

1. **Initialisierung**:
   - Aktivierung durch `INIT`-Ereignis
   - Bestätigung via `INITO` mit Statusinformation
2. **Datenaktualisierung**:
   - Automatische Statusmeldungen via `IND`-Ereignis
   - Enthält alle relevanten Front-Anbaugeräteparameter
3. **Fehlerbehandlung**:
   - Timeout-Erkennung via `TIMEOUT`-Ereignis
   - Detaillierte Statuscodes im `STATUS`-Feld

Technische Besonderheiten

✓ **ISO 11783-7 konform** (PGN 65094) ✓ **Echtzeitfähige** Statusüberwachung ✓ **Umfassende Parameter** für Front-Anbaugeräte ✓ **Integrierte Fehlerdiagnose**

Anwendungsszenarien

- **Frontladersteuerung**: Präzise Positionsregelung
- **Kraftmessung**: Echtzeit-Überwachung der Zugkräfte
- **Sicherheitssysteme**: Erkennung von Grenzwertüberschreitungen
- **Diagnose**: Systematische Fehleranalyse

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	I_FHS	Standard Hitch Control	Advanced Hitch Manager
ISO-Standard	✓	✖	✖
Front-Anbau	✓	✖	✓
Kraftmessung	✓	✓	✓
Diagnosecodes	✓	✖	✓

🛠️ Zugehörige Übungen

- Uebung_079

Fazit

Der I_FHS-Baustein bietet umfassende Front-Anbaugerätesteuerung:

- **Standardkonform**: Volle ISO 11783-7 Kompatibilität
- **Präzise**: Hochauflösende Messparameter
- **Zuverlässig**: Integrierte Fehlererkennung

Essential für:

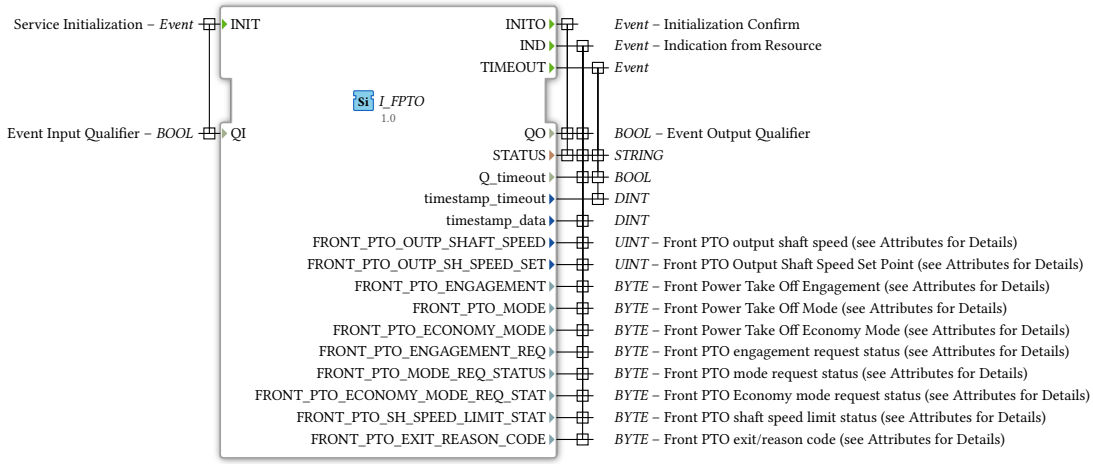
- Moderne Frontlader-Systeme
- Präzisionslandwirtschaft
- Sicherheitskritische Anwendungen

I_FPTO

• – ○ • – ○ • – ○ •

Einleitung

Der **I_FPTO** ist ein standardkonformer Funktionsbaustein zur Steuerung und Überwachung der Front Power Take-Off (FPTO)-Ausgangswelle, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-7 Spezifikation (PGN 65092) für die Messung und Steuerung von FPTO-Parametern.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Qualifier `QI`)

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung (mit Status)
- `IND`: Indikationsereignis mit allen FPTO-Parametern
- `TIMEOUT`: Timeout-Ereignis

Daten-Eingänge

- `QI` (BOOL): Qualifier für Initialisierung

Daten-Ausgänge

- `QO` (BOOL): Qualifier für Ausgangsereignisse
- `STATUS` (STRING): Betriebsstatusmeldung
- `Q_timeout` (BOOL): Timeout-Indikator
- `timestamp_timeout` (DINT): Zeitstempel für Timeout
- `timestamp_data` (DINT): Zeitstempel für Daten

FPTO-Parameter

Parameter	Typ	Beschreibung	SPN	Bit-Länge	Skalierung
`FRONT_PTO_OUTP_SHAFT_SPEED`	UINT	Aktuelle Drehzahl der FPTO-Welle	1882	16	0.125 1/min/bit
`FRONT_PTO_OUTP_SH_SPEED_SET`	UINT	Soll-Drehzahl der FPTO-Welle	1884	16	0.125 1/min/bit
`FRONT_PTO_ENGAGEMENT`	BYTE	Zustand der FPTO-Kupplung	1888	2	4 Zustände/2 bit

FRONT_PTO_MODE	BYTE	Betriebsmodus der FPTO	1889	2	4 Zustände/2 bit
FRONT_PTO_ECONOMY_MODE	BYTE	Economy-Modus der FPTO	1891	2	4 Zustände/2 bit
FRONT_PTO_ENGAGEMENT_REQ	BYTE	Kupplungsanforderungsstatus	5152	2	4 Zustände/2 bit
FRONT_PTO_MODE_REQ_STATUS	BYTE	Modusanforderungsstatus	5153	2	4 Zustände/2 bit
FRONT_PTO_ECONOMY_MODE_REQ_STAT	BYTE	Economy-Modus-Anforderungsstatus	5154	2	4 Zustände/2 bit
FRONT_PTO_SH_SPEED_LIMIT_STAT	BYTE	Drehzahlbegrenzungsstatus	5155	3	8 Zustände/3 bit
FRONT_PTO_EXIT_REASON_CODE	BYTE	Grundcode für FPTO-Ausfall	5817	6	64 Zustände/6 bit

Funktionsweise

1. **Initialisierung**:
   - `INIT` mit `QI`=TRUE startet Initialisierung
   - `INITO` bestätigt Betriebsbereitschaft mit `QO` und `STATUS`
2. **Datenbereitstellung**:
   - `IND` liefert alle FPTO-Parameter mit Zeitstempel
   - Automatische Aktualisierung bei Zustandsänderungen
3. **Fehlerbehandlung**:
   - `TIMEOUT` bei Kommunikationsproblemen
   - Detaillierte Statusmeldungen im `STATUS`-Feld

Technische Besonderheiten

✓ **ISO 11783-7 konform** (PGN 65092) ✓ **16-bit Drehzahlmessung** mit 0.125 1/min Auflösung ✓ **Zustandsautomaten** für Kupplung und Betriebsmodi ✓ **Diagnosefunktionen** mit Grundcodes

Anwendungsszenarien

- **Landmaschinen**: Steuerung von Front-PTO an Traktoren
- **Drehzahlregelung**: Präzise Geschwindigkeitskontrolle
- **Zustandsüberwachung**: Echtzeit-Diagnose des FPTO-Systems
- **Energieeffizienz**: Economy-Modus-Steuerung

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	I_FPTO	Standard_PTO	Advanced_PTO
ISO-Standard	✓ (ISO 11783-7)	✓	✖
Front-PTO	✓	✖	✓
Economy-Modus	✓	✖	✓
Diagnosecodes	✓	✖	✓

🛠️ Zugehörige Übungen

- Uebung_079

Fazit

Der I_FPTO-Baustein bietet die Standardimplementierung für Front-PTO-Systeme:

- **Präzise**: Hochauflösende Drehzahlmessung
- **Umfassend**: Komplette Zustandsüberwachung
- **Robust**: Integrierte Fehlerdiagnose

Idealer Einsatz bei:

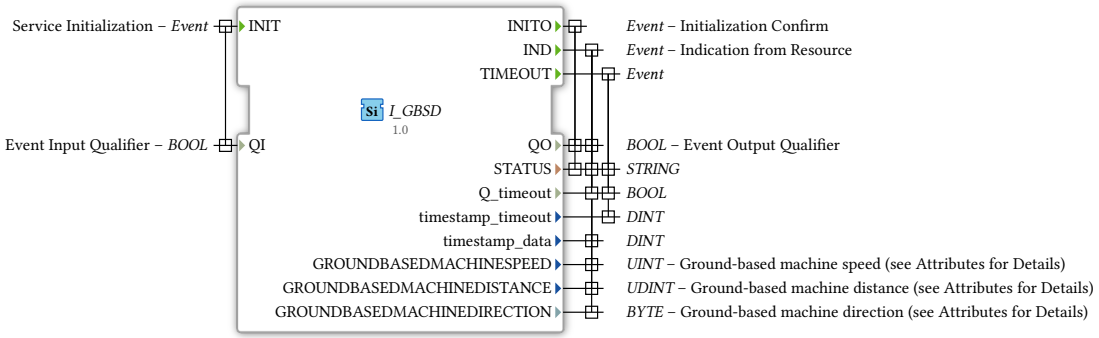
- Front-PTO-Systemen in Landmaschinen
- Anwendungen mit hohen Anforderungen an Drehzahlgenauigkeit
- Systemen mit erweiterten Diagnoseanforderungen

I_GBSD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der **I_GBSD** (Ground-based Speed and Distance) ist ein standardkonformer Funktionsbaustein zur Messung von Fahrzeuggeschwindigkeit, Distanz und Fahrtrichtung, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-7 Spezifikation (PGN 65097) für präzise Bewegungsdaten in landwirtschaftlichen und baulichen Maschinen.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Qualifier `QI`)

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung (mit Status)
- `IND`: Datenindikation mit Geschwindigkeit, Distanz und Richtung
- `TIMEOUT`: Timeout-Ereignis

Daten-Eingänge

- `QI` (BOOL): Qualifier für Initialisierung

Daten-Ausgänge

- `QO` (BOOL): Qualifier für Ausgangsereignisse
- `STATUS` (STRING): Betriebsstatusmeldung
- `Q_timeout` (BOOL): Timeout-Indikator
- `timestamp_timeout` (DINT): Zeitstempel für Timeout
- `timestamp_data` (DINT): Zeitstempel für Bewegungsdaten

Bewegungsdaten-Parameter

Parameter	Typ	Beschreibung	SPN	Bit-Länge	Skalierung	Genauigkeit
`GROUNDBASEDMACHINESPEED`	UINT	Maschinengeschwindigkeit	1859	16	0.001 m/s pro Bit	±0.1% bei >1 m/s

GROUNDBASEDMACHINEDISTANCE	UINT	Zurückgelegte Distanz	1860	32	0.001 m pro Bit	±0.5% kumulativ
GROUNDBASEDMACHINEDIRECTION	BYTE	Fahrtrichtung	1861	2	4 Zustände/2 Bit	-

Fahrtrichtungszustände

Wert	Zustand	Beschreibung
0	Stillstand	Keine Bewegung erkannt
1	Vorwärts	Bewegung in Vorwärtsrichtung
2	Rückwärts	Bewegung in Rückwärtsrichtung
3	Undefiniert	Richtung nicht bestimmbar

Funktionsweise

1. **Initialisierung**:
   - `INIT` mit `QI`=TRUE startet Sensorkalibrierung
   - `INITO` bestätigt Betriebsbereitschaft mit Systemstatus
2. **Datenaktualisierung**:
   - `IND` liefert alle Bewegungsdaten mit 100ms Update-Rate
   - Distanzmessung als freilaufender Zähler (32-bit)
3. **Besondere Betriebsmodi**:
   - Automatische Genauigkeitsanpassung bei niedrigen Geschwindigkeiten (<0.5 m/s)
   - Integrierte Plausibilitätsprüfung der Sensordaten

Technische Besonderheiten

✓ **ISO 11783-7 konform** (PGN 65097) ✓ **Hochpräzise Messung** mit 1mm Auflösung ✓ **32-bit Distanzzähler** (bis zu 4,294,967km Reichweite) ✓ **Robuste Richtungserkennung** mit 4 Zuständen

Anwendungsszenarien

- **Präzisionslandwirtschaft**: Geschwindigkeitskontrolle für Saat- und Düngemittel
- **Flächenberechnung**: Automatische Arbeitsflächenmessung
- **Fahrassistenz**: Richtungserkennung bei Rückwärtsfahrt
- **Telematik**: Betriebsdatenerfassung für Maschinenmanagement

Genauigkeitsmerkmale

Geschwindigkeitsbereich	Typische Genauigkeit	Update-Rate
> 2 m/s (7.2 km/h)	±0.1%	100 ms
0.5 - 2 m/s	±1%	200 ms
< 0.5 m/s	±5%	500 ms

⚖️ Vergleich mit ähnlichen Systemen

Feature	I_GBSD	Radbasiert	GPS-basiert
Bodenkontakt	✓ Direkt	✓ Indirekt	✖
Niedriggeschwindigkeit	✓ Gut	✖ Ungenau	✖ Ungenau
Richtungserkennung	✓ Präzise	✓	✖ Mehrdeutig
Signalausfall	Robust	Anfällig	Störanfällig

🛠️ Zugehörige Übungen

- Uebung_072
- Uebung_072b
- Uebung_072c
- Uebung_073
- Uebung_079

Fazit

Der I_GBSD-Baustein bietet zuverlässige Bewegungsdaten für mobile Arbeitsmaschinen:

- **Präzision**: Submillimeter-genaue Distanzmessung
- **Zuverlässigkeit**: Funktion auch bei schlechten GPS-Bedingungen
- **Flexibilität**: Universeller Einsatz in Landwirtschaft und Baumaschinen

Idealer Einsatz bei:

- Automatischen Lenksystemen
- Arbeitsbreitenberechnungen
- Maschinen mit hohen Genauigkeitsanforderungen
- Anwendungen mit häufiger Richtungswechsel

I_Lighting

• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock **I_Lighting** verarbeitet die Beleuchtungsdaten eines landwirtschaftlichen Fahrzeugs gemäß der ISO 11783-7 (ISOBUS). Er empfängt und decodiert die Parametergruppennummer (PGN) 65088, die den Status aller Beleuchtungsfunktionen eines Traktors und angeschlossener Anbaugeräte übermittelt. Der Baustein dient als Schnittstelle zwischen dem ISOBUS-Netzwerk und der Applikationslogik zur Überwachung und Steuerung der Beleuchtung.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung	Mitgeführte Daten
INIT	EInit	Initialisiert den Baustein und aktiviert die Verarbeitung.	QI

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung	Mitgeführte Daten
INITO	EInit	Bestätigt die erfolgreiche Initialisierung.	QO, STATUS
IND	Event	Signalisierung neuer Beleuchtungsdaten vom Bus.	QO, timestamp_data, STATUS, Q_timeout sowie alle 32 Leuchtenstatus-Ausgänge
TIMEOUT	Event	Wird ausgelöst, wenn die erwarteten Daten ausbleiben (Zeitüberschreitung).	timestamp_timeout, STATUS, Q_timeout

Daten-Eingänge

Name	Datentyp	Beschreibung
QI	BOOL	Aktivierungssignal für die Initialisierung; bei TRUE wird der Baustein gestartet.

Daten-Ausgänge

Name	Datentyp	Beschreibung
QO	BOOL	Quittierungssignal nach erfolgreicher Initialisierung oder Datenverarbeitung.
STATUS	STRING	Statusmeldung (z. B. Fehlertexte oder Betriebshinweise).
Q_timeout	BOOL	Signalisiert, ob eine Zeitüberschreitung aufgetreten ist (TRUE = Timeout aktiv).
timestamp_timeout	DINT	Zeitstempel des letzten Timeout-Ereignisses.
timestamp_data	DINT	Zeitstempel der zuletzt empfangenen Beleuchtungsdaten.
32 Leuchten-Ausgänge	BYTE	Jeweils ein 2-Bit-Wert (QUARTER) für jede Beleuchtungsfunktion. Initialwert: 0xFF. – DAYTIME_RUNNING_LIGHTS – ALTERNATE_HEADLIGHTS – LOW_BEAM_HEADLIGHTS – HIGH_BEAM_HEADLIGHTS – TRACTOR_FRONT_FOG_LIGHTS – ROTATING_BEACON_LIGHT – RIGHT_TURN_SIGNAL_LIGHTS – LEFT_TURN_SIGNAL_LIGHTS – BACK_UP_LIGHTS_AND_ALARM_HORN – CENTER_STOP_LIGHTS – RIGHT_STOP_LIGHTS – LEFT_STOP_LIGHTS – IMPLEMENT_CLEARANCE_LIGHTS – TRACTOR_CLEARANCE_LIGHTS – IMPLEMENT_MARKER_LIGHTS – TRACTOR_MARKER_LIGHTS – REAR_FOG_LIGHTS – TRACTOR_UNDERSIDE_MOUNTED_WORK_LIGHTS – TRACTOR_REAR_LOW_MOUNTED_WORK_LIGHTS – TRACTOR_REAR_HIGH_MOUNTED_WORK_LIGHTS – TRACTOR_SIDE_LOW_MOUNTED_WORK_LIGHTS –

		TRACTOR_SIDE_HIGH_MOUNTED_WORK_LIGHTS – TRACTOR_FRONT_LOW_MOUNTED_WORK_LIGHTS – TRACTOR_FRONT_HIGH_MOUNTED_WORK_LIGHTS – IMPLEMENT_OEM_OPTION_2_LIGHT – IMPLEMENT_OEM_OPTION_1_LIGHT – IMPLEMENT_RIGHT_FORWARD_WORK_LIGHTS – IMPLEMENT_LEFT_FORWARD_WORK_LIGHTS – IMPLEMENT_RIGHT_FACING_WORK_LIGHTS – IMPLEMENT_LEFT_FACING_WORK_LIGHTS – IMPLEMENT_REAR_WORK_LIGHTS

Adapter

Keine.

Funktionsweise

Der **I_Lighting**-Baustein wird durch das Ereignis `INIT` aktiviert. Dabei wird die interne Verarbeitung vorbereitet. Nach erfolgreicher Initialisierung quittiert er mit `INITO`. Anschließend wartet der Baustein auf eingehende ISOBUS-Nachrichten (PGN 65088). Trifft eine gültige Nachricht ein, wird das Ereignis `IND` ausgelöst und alle Beleuchtungsausgänge mit den dekodierten 2-Bit-Werten (QUARTER) aktualisiert. Gleichzeitig wird ein Zeitstempel in `timestamp_data` abgelegt. Falls innerhalb einer konfigurierten Zeitspanne keine neuen Daten eintreffen, wird das Ereignis `TIMEOUT` ausgelöst und `Q_timeout` auf TRUE gesetzt. Der Baustein stellt somit eine zuverlässige Überwachung des Beleuchtungsstatus sicher.

Technische Besonderheiten

- **ISO 11783-7 Konformität:** Der Baustein implementiert die Parametergruppe PGN 65088 „Lighting Data LD“ gemäß dem ISOBUS-Standard.
- **2-Bit-QUARTER-Kodierung:** Jeder Leuchtenausgang codiert vier Zustände in zwei Bits – typische Interpretation: 0 = aus, 1 = an, 2 = Fehler, 3 = nicht verfügbar. Der Initialwert `16#FF` (Dezimal 255) entspricht dem Zustand „nicht verfügbar“.
- **SPN-Attribute:** Jeder Ausgang ist mit detaillierten Metadaten (SPN, Position im Datentelegramm, Skalierung, Link zur Spezifikation) versehen, was die Rückverfolgbarkeit und Konfiguration erleichtert.
- **Timeout-Erkennung:** Der Baustein kann das Ausbleiben von Busnachrichten erkennen und darüber informieren, z. B. für Fehlerbehandlung oder redundantes Verhalten.
- **Zeitstempel:** Sowohl Datenankunft als auch Timeout werden mit einem Zeitstempel (DINT) versehen, was die Analyse im Zeitverlauf ermöglicht.

Zustandsübersicht

Der Baustein besitzt keinen expliziten, veröffentlichten Zustandsautomaten. Das logische Verhalten lässt sich jedoch wie folgt beschreiben:

- **Initialisierung:** Nach `INIT` wechselt der Baustein in den aktiven Zustand, quittiert mit `INITO`.
- **Empfangsbereit:** Er wartet auf eingehende Nachrichten. Bei Empfang → `IND` aktualisiert alle Ausgänge und setzt `Q_timeout` zurück.
- **Timeout-Überwachung:** Läuft ein interner Timer ab, ohne dass neue Daten eintreffen, wechselt der Baustein kurz in den Timeout-Zustand und signalisiert dies über `TIMEOUT`. Danach kehrt er in den Empfangsbereit-Zustand zurück.
- **Fehlerbehandlung:** Tritt ein Fehler während der Dekodierung auf, wird `STATUS` entsprechend gesetzt.

Anwendungsszenarien

- **Traktorbeleuchtungssteuerung:** Integration in ein Fahrzeugsteuergerät (ECU) zur Überwachung und Anzeige des Beleuchtungsstatus auf dem Terminal.
- **ISOBUS-konforme Anbaugeräte:** Verwendung in Anbaugeräten, die eigene Beleuchtung über das ISOBUS-Netzwerk melden (z. B. Frontlader, Heckanbaugeräte).
- **Diagnose und Fehlersuche:** Auslese der Leuchtenstatus und Timeouts zur Fehleranalyse oder Logging in der Servicewerkstatt.
- **Lichtsteuerung in automatisierten Systemen:** Kombination mit weiteren Bausteinen zur automatischen Anpassung der Beleuchtung an Umgebungsbedingungen (z. B. Tag/Nacht-Erkennung).

Vergleich mit ähnlichen Bausteinen

Ähnliche ISOBUS-Bausteine gibt es für andere Parametergruppen, z. B. **I_Engine** (Motordaten), **I_Steering** (Lenkung) oder **I_WorkState** (Arbeitszustand). Im Vergleich zu diesen:

- **Spezialisierung: I_Lighting** ist auf die Beleuchtung fokussiert und bietet eine hohe Anzahl von 32 spezifischen Leuchtenausgängen.
- **Datenbreite:** Die Ausgänge sind BYTE (2 Bit genutzt), während andere Bausteine oft WORD oder DWORD verwenden.
- **Timeout-Behandlung:** Nicht alle Bausteine implementieren eine eigene Timeout-Erkennung – hier ist sie explizit vorgesehen.
- **Initialwert:** Die Ausgänge starten mit 0xFF („nicht verfügbar“), was eine robuste Initialisierung ohne Gültigkeitskonflikte ermöglicht.

Fazit

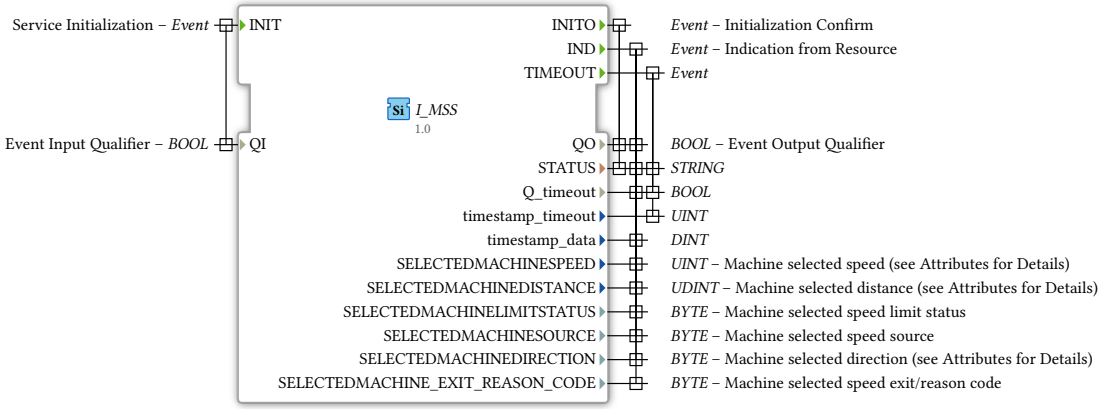
Der **I_Lighting**-Funktionsblock ist ein spezialisierter, standardkonformer Baustein zur Verarbeitung von ISOBUS-Beleuchtungsdaten (PGN 65088). Er dekodiert den Status von 32 verschiedenen Leuchtenfunktionen aus dem CAN-Bus und bietet eine zuverlässige Timeout-Überwachung. Dank detaillierter SPN-Metadaten und der einfachen Schnittstelle eignet er sich ideal für die Integration in landwirtschaftliche Steuerungssysteme, die eine präzise und normgerechte Lichtsteuerung erfordern.

I_MSS

• – ◦ • – ◦ • – ◦ •

Einleitung

Der **I_MSS** (Machine Selected Speed) ist ein standardkonformer Funktionsbaustein zur Steuerung und Überwachung der maschinengewählten Geschwindigkeit, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-7 Spezifikation (PGN 65097) für präzise Geschwindigkeitssteuerung in landwirtschaftlichen und baulichen Maschinen.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Qualifier `QI`)

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung (mit Status)
- `IND`: Indikationsereignis mit allen Geschwindigkeitsparametern
- `TIMEOUT`: Timeout-Ereignis

Daten-Eingänge

- `QI` (BOOL): Qualifier für Initialisierung

Daten-Ausgänge

- `QO` (BOOL): Qualifier für Ausgangsereignisse
- `STATUS` (STRING): Betriebsstatusmeldung
- `Q_timeout` (BOOL): Timeout-Indikator

- `timestamp_timeout` (UINT): Zeitstempel für Timeout
- `timestamp_data` (DINT): Zeitstempel für Geschwindigkeitsdaten

Geschwindigkeitsparameter

Parameter	Typ	Beschreibung	SPN	Bit-Länge	Skalierung
`SELECTEDMACHINESPEED`	UINT	Maschinengeschwindigkeit	4305	16	0.001 m/s/bit (LSB), 0.256 m/s/bit (MSB)
`SELECTEDMACHINEDISTANCE`	UDINT	Zurückgelegte Distanz	4306	32	0.001 m/bit
`SELECTEDMACHINELIMITSTATUS`	BYTE	Geschwindigkeitsbegrenzungsstatus	4307	3	8 Zustände/3 bit
`SELECTEDMACHINESOURCE`	BYTE	Geschwindigkeitsquelle	4308	3	8 Zustände/3 bit
`SELECTEDMACHINEDIRECTION`	BYTE	Fahrtrichtung	4309	2	4 Zustände/2 bit
`SELECTEDMACHINE_EXIT_REASON_CODE`	BYTE	Grundcode für Geschwindigkeitsausfall	5818	6	64 Zustände/6 bit

Fahrtrichtungszustände

Wert	Zustand	Beschreibung
0	Stillstand	Keine Bewegung erkannt
1	Vorwärts	Bewegung in Vorwärtsrichtung
2	Rückwärts	Bewegung in Rückwärtsrichtung
3	Undefiniert	Richtung nicht bestimmbar

Funktionsweise

1. **Initialisierung**:
   - `INIT` mit `QI`=TRUE startet Systeminitialisierung
   - `INITO` bestätigt Betriebsbereitschaft mit `QO` und `STATUS`
2. **Datenbereitstellung**:
   - `IND` liefert alle Geschwindigkeitsparameter mit Zeitstempel
   - Automatische Aktualisierung bei Zustandsänderungen
3. **Fehlerbehandlung**:
   - `TIMEOUT` bei Kommunikationsproblemen
   - Detaillierte Fehlercodes in `SELECTEDMACHINE_EXIT_REASON_CODE`

Technische Besonderheiten

✓ **ISO 11783-7 konform** (PGN 65097) ✓ **Duale Geschwindigkeitsskalierung** für hohe Präzision ✓ **Multiquellen-Unterstützung** (8 mögliche Geschwindigkeitsquellen) ✓ **Intelligente Limitierungserkennung** mit 3-stufigem Status

Anwendungsszenarien

- **Traktoren**: Automatische Geschwindigkeitsanpassung für Feldarbeit
- **Erntemaschinen**: Präzise Reihenlängenberechnung
- **Baumaschinen**: Geschwindigkeitsbegrenzung in Gefahrenbereichen
- **Flottenmanagement**: Betriebsdatenerfassung und -analyse

Geschwindigkeitsquellen

Code	Quelle	Typische Anwendung

0	Undefiniert	Systemstart
1	Radsensor	Standardbetrieb
2	Bodensensor	Präzisionslandwirtschaft
3	GPS	Überwachung
4-7	Reserviert	Herstellerspezifisch

⚖️ Vergleich mit ähnlichen Systemen

Feature	I_MSS	Standard	GPS-basiert
Genauigkeit	±0.2%	±1-2%	±5-10%
Reaktionszeit	<100ms	200ms	1-2s
Quellenflexibilität	8	1-2	1
Niedriggeschwindigkeit	✓ Optimal	✖ Ungenau	✖ Unbrauchbar

🛠️ Zugehörige Übungen

- Uebung_079

Fazit

Der I_MSS-Baustein bietet präzise Geschwindigkeitssteuerung für mobile Arbeitsmaschinen:

- **Flexibel**: Unterstützung mehrerer Geschwindigkeitsquellen
- **Präzise**: Duale Skalierung für hohe Genauigkeit
- **Robust**: Integrierte Fehlerdiagnose

Idealer Einsatz bei:

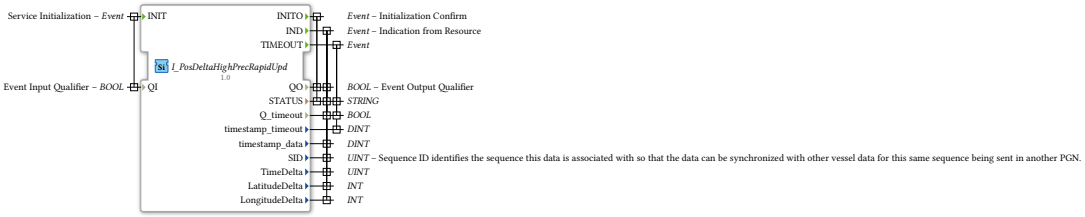
- Automatischen Lenksystemen
- Präzisionslandwirtschaft
- Maschinen mit hohen Geschwindigkeitsanforderungen
- Anwendungen mit wechselnden Betriebsbedingungen

I_PosDeltaHighPrecRapidUpd

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `I_PosDeltaHighPrecRapidUpd` implementiert die Verarbeitung des NMEA 2000 Parameter Group Numbers (PGN) 129027 „Position Delta, High Precision Rapid Update“. Dieser Baustein ist für Anwendungen konzipiert, die sehr hohe Präzision und sehr schnelle Aktualisierungsraten für Positionsdaten erfordern. Er kann Positionsänderungen (Delta) mit einer Auflösung von bis zu 1 Millimeter und einem Delta-Zeitintervall mit einer Genauigkeit von 5 Millisekunden bereitstellen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Dienstinitialisierung. Wird zusammen mit dem Qualifier `QI` ausgelöst, um den Funktionsblock zu initialisieren oder zu deaktivieren.

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung. Wird als Antwort auf das `INIT`-Ereignis ausgelöst und gibt den aktuellen Status (`STATUS`) und den Qualifier `QO` zurück.
- **IND**: Indikation vom Ressourcen-Manager. Wird ausgelöst, wenn neue Positionsdaten verfügbar sind. Überträgt alle relevanten Delta-Positionsdaten sowie Statusinformationen.
- **TIMEOUT**: Wird ausgelöst, wenn ein Timeout auftritt, z.B. wenn keine neuen Daten innerhalb eines erwarteten Zeitrahmens empfangen wurden.

Daten-Eingänge

- **QI** (BOOL): Event Input Qualifier. Steuert die Initialisierung. Bei `TRUE` wird der Baustein aktiviert, bei `FALSE` deaktiviert.

Daten-Ausgänge

- **QO** (BOOL): Event Output Qualifier. Gibt den aktuellen Betriebszustand des Bausteins wider (z.B. aktiv/inaktiv).
- **STATUS** (STRING): Enthält Status- oder Fehlermeldungen des Funktionsblocks.
- **Q_timeout** (BOOL): Gibt an, ob die letzten empfangenen Daten aufgrund eines Timeouts veraltet sind.
- **timestamp_timeout** (DINT): Zeitstempel, der dem `TIMEOUT`-Ereignis zugeordnet ist.
- **timestamp_data** (DINT): Zeitstempel der zuletzt empfangenen gültigen Positionsdaten.
- **SID** (UINT): Sequence ID. Identifiziert die Sequenz, der diese Daten zugeordnet sind, um eine Synchronisation mit anderen Fahrzeugdaten derselben Sequenz in einem anderen PGN zu ermöglichen.
- **TimeDelta** (UINT): Die Zeitdifferenz in Millisekunden, über die die Positionsänderung gemessen wurde.
- **LatitudeDelta** (INT): Die Änderung der geografischen Breite (Delta) in hochpräziser Auflösung (1/1e-7 min = 1.85 mm).
- **LongitudeDelta** (INT): Die Änderung der geografischen Länge (Delta) in hochpräziser Auflösung (1/1e-7 min = 1.85 mm).

Adapter

Dieser Funktionsblock verwendet keine Adapter-Schnittstellen.

Funktionsweise

Der Baustein wird über das `INIT`-Ereignis mit dem entsprechenden `QI`-Wert gestartet oder gestoppt. Nach erfolgreicher Initialisierung bestätigt er dies mit `INITO`. Im aktiven Zustand lauscht der Baustein auf eingehende PGN 129027 Nachrichten vom NMEA 2000-Netzwerk.

Bei Empfang neuer, gültiger Daten löst er das `IND`-Ereignis aus und stellt alle berechneten Delta-Werte (Zeit, Breite, Länge) zusammen mit der Sequenz-ID (`SID`) und einem Datenzeitstempel an seinen Ausgängen bereit. Der `STATUS`-Ausgang kann dabei zusätzliche Informationen liefern.

Sollte über einen definierten Zeitraum hinweg keine neue Nachricht empfangen werden, kann der Baustein ein `TIMEOUT`-Ereignis generieren. Dies signalisiert, dass die aktuell gehaltenen Daten möglicherweise veraltet sind (`Q_timeout` = TRUE).

Technische Besonderheiten

- **Hohe Präzision**: Verarbeitet Positionsdeltas mit einer Auflösung im Millimeterbereich (1/1e-7 Bogenminuten).
- **Schnelle Aktualisierung**: Unterstützt sehr kurze Update-Intervalle, die durch das `TimeDelta` (min. 5 ms Genauigkeit) abgebildet werden.
- **Sequenzierung**: Die `SID` (Sequence ID) ermöglicht die korrekte Zuordnung und Synchronisation von Positionsdeltas mit anderen zeitkritischen Datenströmen im System.
- **Timeout-Erkennung**: Integrierte Überwachung der Datenaktualität zur Erkennung von Kommunikationsausfällen oder verzögerten Nachrichten.

Zustandsübersicht

Der Funktionsblock besitzt im Wesentlichen zwei Hauptzustände:

1. **Inaktiv / Initialisiert**: Der Baustein ist initialisiert, aber nicht aktiv (`QI` = FALSE). Es werden keine Daten verarbeitet.
2. **Aktiv / Lauschend**: Der Baustein ist aktiviert (`QI` = TRUE) und wartet auf eingehende PGN 129027 Nachrichten. Bei Empfang werden diese verarbeitet und über `IND` ausgegeben. Ein fehlendes Signal

kann zum Übergang in einen Timeout-Zustand führen, was durch das `TIMEOUT`-Ereignis signalisiert wird.

Anwendungsszenarien

- **Präzisionslandwirtschaft**: Für hochgenaue Spurführungssysteme und automatische Lenksysteme, die Echtzeit-Positionskorrekturen benötigen.
- **Dynamische Positionierung**: In maritimen Anwendungen zur präzisen Lagehaltung von Schiffen oder Offshore-Plattformen.
- **Autonome Fahrzeuge**: Zur hochfrequenten Odometrie und Pfadkorrektur in Echtzeit.
- **Vermessung und Kartierung**: Bei der Erfassung von Geländedaten mit sehr hoher räumlicher und zeitlicher Auflösung.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-Positionsbausteinen (z.B. solchen, die PGN 129029 „Position, Rapid Update“ verarbeiten) bietet `I_PosDeltaHighPrecRapidUpd`:

- **Delta-basierte Daten**: Statt absoluter Positionen werden Änderungen (Deltas) übertragen, was bei hohen Update-Raten bandbreiteneffizienter sein kann.
- **Höhere Präzision**: Spezifiziert für Anwendungen, die eine höhere Genauigkeit als die typischen Rapid-Update-PGNs benötigen.
- **Explizite Zeitdifferenz**: Das `TimeDelta`-Feld ist integraler Bestandteil der Nachricht, was eine präzisere Geschwindigkeits- und Beschleunigungsberechnung ermöglicht.

Fazit

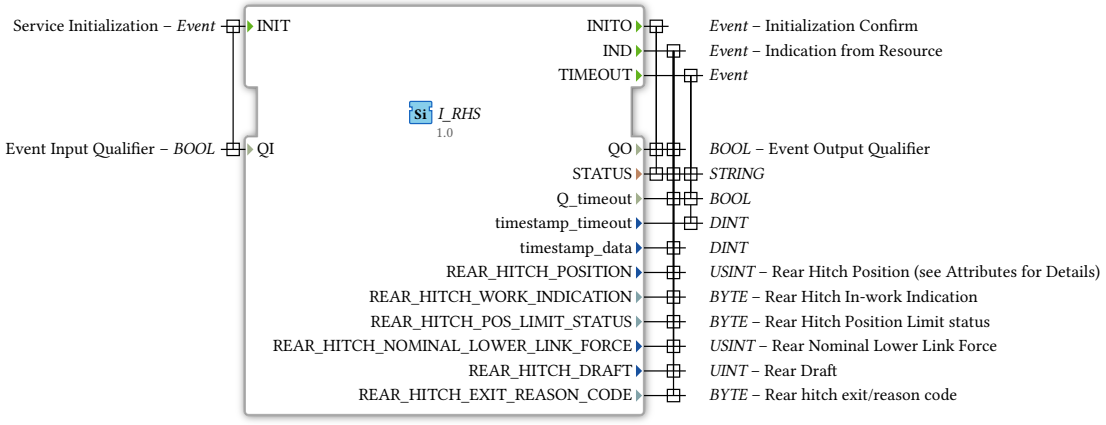
Der `I_PosDeltaHighPrecRapidUpd` Funktionsblock ist ein spezialisiertes Werkzeug für anspruchsvolle Echtzeit-Positionsbestimmung. Seine Fähigkeit, hochpräzise Positionsänderungen mit sehr hoher Frequenz zu liefern und zu synchronisieren, macht ihn ideal für präzisionskritische Steuerungs- und Regelungsaufgaben in den Bereichen Landtechnik, Maritime und autonome Systeme. Die integrierte Timeout-Erkennung erhöht zudem die Robustheit der Anwendung.

I_RHS

• – ◦ • – ◦ • – ◦ •

Einleitung

Der **I_RHS** (Rear Hitch Status) ist ein standardkonformer Funktionsbaustein zur Überwachung der Heckkupplungsparameter, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-7 Spezifikation (PGN 65093) für die präzise Erfassung von Position, Kraft und Status der Heckkupplung in Landmaschinen.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Qualifier `QI`)

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung (mit Status)
- `IND`: Indikationsereignis mit allen Kupplungsparametern
- `TIMEOUT`: Timeout-Ereignis

Daten-Eingänge

- `QI` (BOOL): Qualifier für Initialisierung

Daten-Ausgänge

- `QO` (BOOL): Qualifier für Ausgangsereignisse
- `STATUS` (STRING): Betriebsstatusmeldung
- `Q_timeout` (BOOL): Timeout-Indikator
- `timestamp_timeout` (DINT): Zeitstempel für Timeout
- `timestamp_data` (DINT): Zeitstempel für Kupplungsdaten

Kupplungsparameter

Parameter	Typ	Beschreibung	SPN	Bit-Länge	Skalierung
REAR_HITCH_POSITION	USINT	Kupplungsposition	1873	8	0.4 %/bit
REAR_HITCH_WORK_INDICATION	BYTE	Arbeitszustand	1877	2	4 Zustände/2 bit
REAR_HITCH_POS_LIMIT_STATUS	BYTE	Positionslimitierung	5151	3	8 Zustände/3 bit
REAR_HITCH_NOMINAL_LOWER_LINK_FORCE	USINT	Unterlenkerkraft	1881	8	0.8 %/bit (-100% Offset)
REAR_HITCH_DRAFT	UINT	Zugkraft	1879	16	10 N/bit (-320kN Offset)
REAR_HITCH_EXIT_REASON_CODE	BYTE	Fehlergrundcode	5819	6	64 Zustände/6 bit

Funktionsweise

1. **Initialisierung**:
   - `INIT` mit `QI`=TRUE startet Systemkalibrierung
   - `INITO` bestätigt Betriebsbereitschaft mit `QO` und `STATUS`
2. **Datenbereitstellung**:
   - `IND` liefert alle Kupplungsparameter mit Zeitstempel
   - Automatische Aktualisierung bei Zustandsänderungen
3. **Fehlerbehandlung**:
   - `TIMEOUT` bei Kommunikationsproblemen
   - Detaillierte Fehlercodes in `REAR_HITCH_EXIT_REASON_CODE`

Technische Besonderheiten

✓ **ISO 11783-7 konform** (PGN 65093) ✓ **Bidirektionale Kraftmessung** (+/- 100% Bereich) ✓ **Präzise Positionserfassung** mit 0.4% Auflösung ✓ **64-stufige Fehlerdiagnose** mit detaillierten Codes

Kupplungseigenschaften

Feature	Beschreibung
Positionsbereich	0-100% (0 = ganz unten)
Kraftmessung	±100% der Nennlast
Zugkraftbereich	−320kN bis +350kN
Update-Rate	100ms im Normalbetrieb

Rückgabecodes (REAR_HITCH_EXIT_REASON_CODE)

Codebereich	Bedeutung
0-15	Systemfehler
16-31	Positionsfehler
32-47	Kraftmessfehler
48-63	Reserviert

Anwendungsszenarien

- **Pflugsteuerung**: Automatische Tiefenregelung
- **Lastmanagement**: Anpassung der Zugkraft
- **Diagnose**: Früherkennung von Hydraulikproblemen
- **Sicherheit**: Überwachung der Kupplungsposition

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	I_RHS	Standard	Premium
Genauigkeit	±0.4%	±2%	±0.2%
Kraftmessung	Bidirektional	Nur Zug	Triaxial
Diagnosecodes	64	8	128
ISO-Konformität	Voll	Teilweise	Voll

🛠️ Zugehörige Übungen

- Uebung_079

Fazit

Der I_RHS-Baustein bietet umfassende Überwachung für Heckkupplungssysteme:

- **Präzise**: Hochauflösende Positions- und Kraftmessung
- **Zuverlässig**: Robuste Fehlererkennung
- **Flexibel**: Für verschiedene Anbaugeräte geeignet

Idealer Einsatz bei:

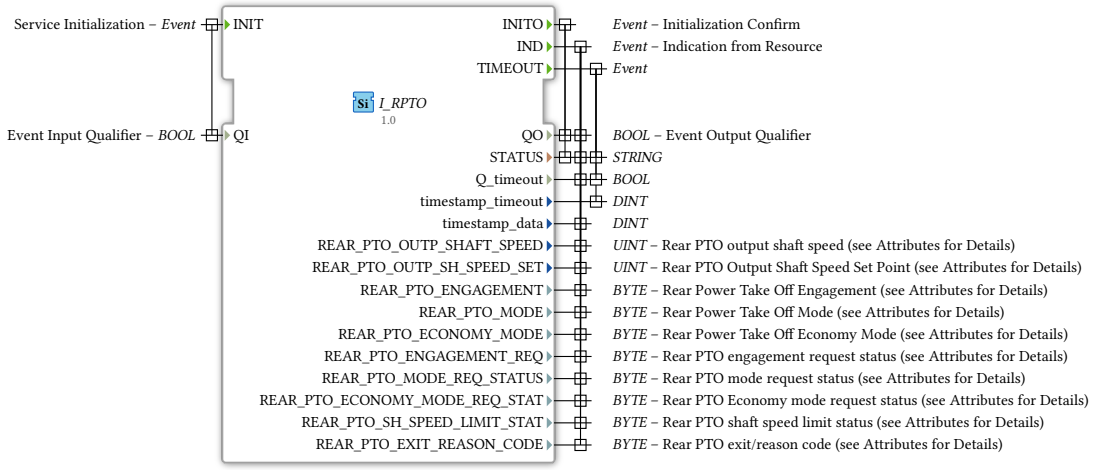
- Modernen Traktoren
- Präzisionslandwirtschaft
- Schweren Anbaugeräten
- Automatischen Lenksystemen

I_RPTO

• – ◦ • – ◦ • – ◦ •

Einleitung

Der **I_RPTO** (Rear Power Take-Off) ist ein standardkonformer Funktionsbaustein zur Steuerung und Überwachung der Heckzapfwelle, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-7 Spezifikation (PGN 65091) für die präzise Regelung von Zapfwellenparametern in Landmaschinen.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Qualifier `QI`)

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung (mit Status)
- `IND`: Indikationsereignis mit allen Zapfwellenparametern
- `TIMEOUT`: Timeout-Ereignis

Daten-Eingänge

- `QI` (BOOL): Qualifier für Initialisierung

Daten-Ausgänge

- `QO` (BOOL): Qualifier für Ausgangsereignisse
- `STATUS` (STRING): Betriebsstatusmeldung
- `Q_timeout` (BOOL): Timeout-Indikator
- `timestamp_timeout` (DINT): Zeitstempel für Timeout
- `timestamp_data` (DINT): Zeitstempel für Zapfwellendaten

Zapfwellenparameter

Parameter	Typ	Beschreibung	SPN	Bit-Länge	Skalierung
`REAR_PTO_OUTP_SHAFT_SPEED`	UINT	Aktuelle Drehzahl	1883	16	0.125 1/min/bit
`REAR_PTO_OUTP_SH_SPEED_SET`	UINT	Soll-Drehzahl	1885	16	0.125 1/min/bit
`REAR_PTO_ENGAGEMENT`	BYTE	Kupplungszustand	2408	2	4 Zustände/2 bit
`REAR_PTO_MODE`	BYTE	Betriebsmodus	1890	2	4 Zustände/2 bit
`REAR_PTO_ECONOMY_MODE`	BYTE	Economy-Modus	1892	2	4 Zustände/2 bit
`REAR_PTO_ENGAGEMENT_REQ`	BYTE	Kupplungsanforderung	5156	2	4 Zustände/2 bit
`REAR_PTO_MODE_REQ_STATUS`	BYTE	Modusanforderung	5157	2	4 Zustände/2 bit
`REAR_PTO_ECONOMY_MODE_REQ_STAT`	BYTE	Economy-Modus-Anforderung	5158	2	4 Zustände/2 bit
`REAR_PTO_SH_SPEED_LIMIT_STAT`	BYTE	Drehzahlbegrenzung	5159	3	8 Zustände/3 bit
`REAR_PTO_EXIT_REASON_CODE`	BYTE	Fehlergrundcode	5820	6	64 Zustände/6 bit

Betriebsmodi

Code	Modus	Beschreibung
0	Standby	Bereitschaftsmodus
1	Geschwindigkeitsmodus	Konstante Drehzahl
2	Lastmodus	Lastabhängige Regelung
3	Reserviert	Herstellerspezifisch

Funktionsweise

1. **Initialisierung**:
   - `INIT` mit `QI`=TRUE startet Systemkalibrierung
   - `INITO` bestätigt Betriebsbereitschaft mit `QO` und `STATUS`
2. **Drehzahlregelung**:
   - `IND` liefert Ist- und Soll-Drehzahlen
   - Automatische Anpassung an Laständerungen
3. **Fehlerbehandlung**:
   - `TIMEOUT` bei Kommunikationsproblemen
   - Detaillierte Fehlercodes in `REAR_PTO_EXIT_REASON_CODE`

Technische Besonderheiten

✓ **ISO 11783-7 konform** (PGN 65091) ✓ **Präzise Drehzahlregelung** mit 0.125 U/min Auflösung ✓ **4 Betriebsmodi** für unterschiedliche Anwendungen ✓ **64-stufige Fehlerdiagnose** mit detaillierten Codes

Anwendungsszenarien

- **Landmaschinen**: Antrieb von Mähdreschern und Ballenpressen
- **Drehzahlregelung**: Präzise Anpassung an Arbeitsgeräte
- **Energieeffizienz**: Economy-Modus für Teillastbetrieb
- **Diagnose**: Früherkennung von mechanischen Problemen

⚖️ Vergleich mit ähnlichen Systemen

Feature	I_RPTO	Standard	Premium
Drehzahlgenauigkeit	±0.2%	±1%	±0.1%
Betriebsmodi	4	2	6
Diagnoseumfang	64 Codes	8 Codes	128 Codes
Reaktionszeit	<100ms	200ms	50ms

🛠️ Zugehörige Übungen

- Uebung_074
- Uebung_079

Fazit

Der I_RPTO-Baustein bietet zuverlässige Zapfwellensteuerung für moderne Landtechnik:

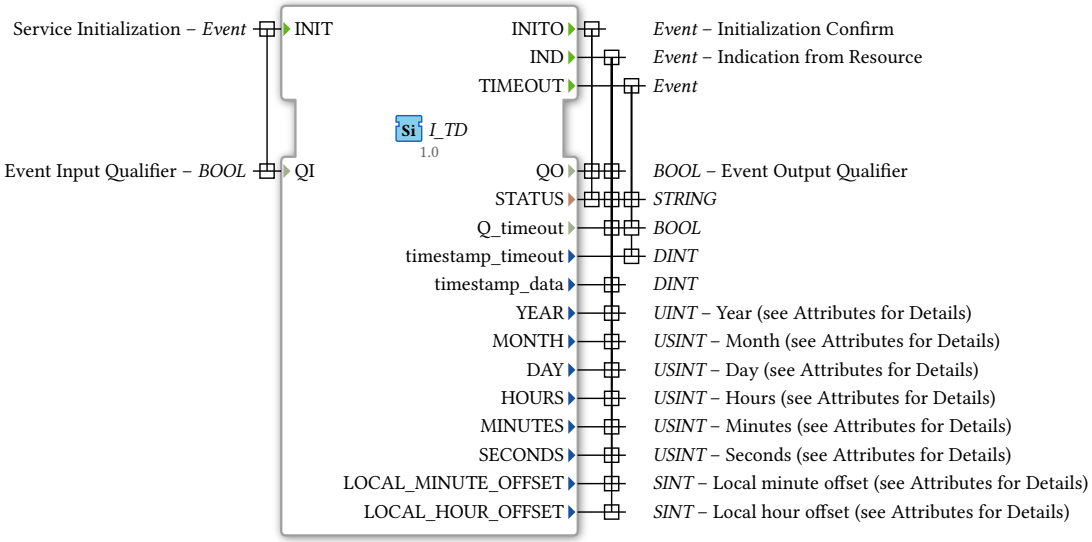
- **Präzise**: Hochauflösende Drehzahlregelung
- **Flexibel**: Mehrere Betriebsmodi für verschiedene Anwendungen
- **Robust**: Integrierte Fehlererkennung und -diagnose

Idealer Einsatz bei:

- Traktoren mit anspruchsvollen Anbaugeräten
- Präzisionslandwirtschaft
- Energieeffizienten Antriebssystemen

- Automatisierten Arbeitsprozessen

I_TD



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **I_TD** (Time/Date) ist ein spezieller Baustein für den ISOBUS, der die Verarbeitung von Zeit- und Datumsinformationen gemäß dem Standard ISO 11783-7 (PGN 65254) ermöglicht. Er dient dem Empfang und der Aufbereitung der im ISOBUS-Netzwerk übertragenen aktuellen Uhrzeit sowie des Datums. Die Ausgangsdaten entsprechen den definierten SPNs (Suspect Parameter Numbers) des Protokolls und werden inklusive der zugehörigen Skalierungen und Offsets bereitgestellt.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Mit Variable	Beschreibung
`INIT`	`QI`	Initialisierung des Bausteins.

Ereignis-Ausgänge

Ereignis	Mit Variablen	Beschreibung
`INITO`	`QO`, `STATUS`	Bestätigung der erfolgreichen Initialisierung.
`IND`	`QO`, `timestamp_data`, `STATUS`, `Q_timeout`, `SECONDS`, `MINUTES`, `HOURS`, `MONTH`, `DAY`, `YEAR`, `LOCAL_MINUTE_OFFSET`, `LOCAL_HOUR_OFFSET`	Anzeige eines empfangenen Zeit-/Datums-Telegramms.
`TIMEOUT`	`timestamp_timeout`, `STATUS`, `Q_timeout`	Zeitüberschreitungsereignis, falls kein gültiges Telegramm innerhalb einer erwarteten Frist eintrifft.

Daten-Eingänge

Variable	Typ	Beschreibung
`QI`	BOOL	Qualifier für die Initialisierung (TRUE = aktivieren).

Daten-Ausgänge

Variable	Typ	Beschreibung (mit ISOBUS-Attributen)
QO	BOOL	Ausgangsqualifier (TRUE = Betrieb bereit).
STATUS	STRING	Statusmeldung (z. B. Fehler oder Erfolg).
Q_timeout	BOOL	Signalisiert, ob ein Timeout aufgetreten ist.
timestamp_timeout	DINT	Zeitstempel des Timeouts (falls aufgetreten).
timestamp_data	DINT	Zeitstempel des empfangenen Datentelegramms.
SECONDS	USINT	Sekunden (Skalierung 0,25 s/bit, Offset 0).
MINUTES	USINT	Minuten (Skalierung 1 min/bit, Offset 0).
HOURS	USINT	Stunden (Skalierung 1 h/bit, Offset 0).
MONTH	USINT	Monat (Skalierung 1 Monat/bit, Offset 0).
DAY	USINT	Tag (Skalierung 0,25 d/bit, Offset 0).
YEAR	UINT	Jahr (Skalierung 1 y/bit, Offset 1985, Initialwert 0xFFFF = „nicht verfügbar").
LOCAL_MINUTE_OFFSET	SINT	Lokaler Minuten-Offset (Skalierung 1 min/bit, Offset −125).
LOCAL_HOUR_OFFSET	SINT	Lokaler Stunden-Offset (Skalierung 1 h/bit, Offset −125).

Adapter

Keine.

Funktionsweise

Der Baustein wird über den Ereigniseingang INIT mit gesetztem QI initialisiert. Nach erfolgreicher Initialisierung wird das Ereignis INITO ausgelöst und der Ausgang QO auf TRUE gesetzt. Anschließend wartet der FB auf eingehende ISOBUS-Telegramme (PGN 65254). Trifft ein solches Telegramm ein, wird das Ereignis IND erzeugt und alle entsprechenden Daten-Ausgänge werden mit den dekodierten Werten aktualisiert. Dabei werden die in den Attributen hinterlegten Skalierungen und Offsets automatisch angewendet.

Sollte innerhalb eines nicht näher spezifizierten Zeitrahmens kein gültiges Telegramm empfangen werden, wird das Ereignis TIMEOUT ausgelöst und der Q_timeout-Ausgang auf TRUE gesetzt. Der Zeitstempel des Timeouts wird in timestamp_timeout abgelegt.

Technische Besonderheiten

- **ISOBUS-Konformität:** Der FB implementiert exakt den PGN 65254 der Norm ISO 11783-7, Teil „Time/Date TD".
- **Initialwerte:** Viele Daten-Ausgänge sind mit vordefinierten „nicht verfügbar"-Werten initialisiert (z. B. YEAR = 0xFFFF), was eine einfache Fehlererkennung erlaubt.
- **Skalierung & Offset:** Alle Größen sind gemäß ISOBUS-Spezifikation skaliert und mit Offsets versehen. Die Rohwerte der SPNs werden in die physikalischen Einheiten umgerechnet.
- **Lokale Zeitverschiebung:** Die Ausgänge LOCAL_MINUTE_OFFSET und LOCAL_HOUR_OFFSET erlauben die Angabe der Abweichung zur UTC-Zeit im Bereich −125 min bis +130 min.
- **Keine Zustandsautomat:** Der FB ist ereignisgesteuert; es gibt keinen expliziten Zustandsautomaten im XML. Die interne Logik basiert auf den IEC-61499-Ausführungsregeln.

Zustandsübersicht

Ein expliziter Zustandsautomat ist im XML nicht definiert. Der FB arbeitet als reaktiver Baustein:

1. **Ruhezustand** – wartet auf INIT.
2. **Initialisierungsphase** – nach INIT bei TRUE QI wird INITO ausgelöst.
3. **Betriebsbereit** – Empfang von Telegrammen → IND; bei Ausbleiben → TIMEOUT.

Anwendungsszenarien

- **Landwirtschaftliche Maschinen:** Empfangen von aktuellen Zeit- und Datumsdaten aus dem ISO-BUS-Bordnetz, z. B. für Logger, Steuerungen oder Displays.
- **Zeitstempelung:** Verwendung der `timestamp_data`-Ausgänge zur Synchronisation von Ereignissen in verschiedenen Steuergeräten.
- **Zeitzonenanpassung:** Auswertung der lokalen Offsets zur Darstellung der Ortszeit in Benutzeroberflächen.

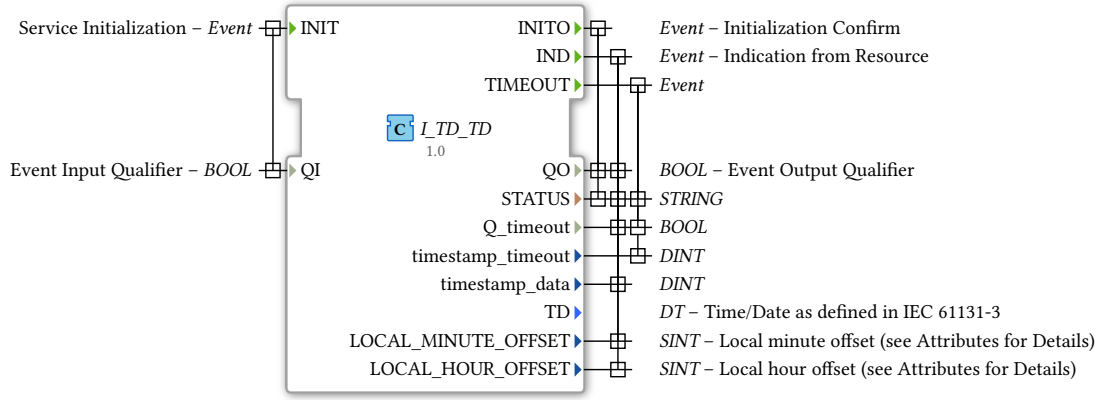
Vergleich mit ähnlichen Bausteinen

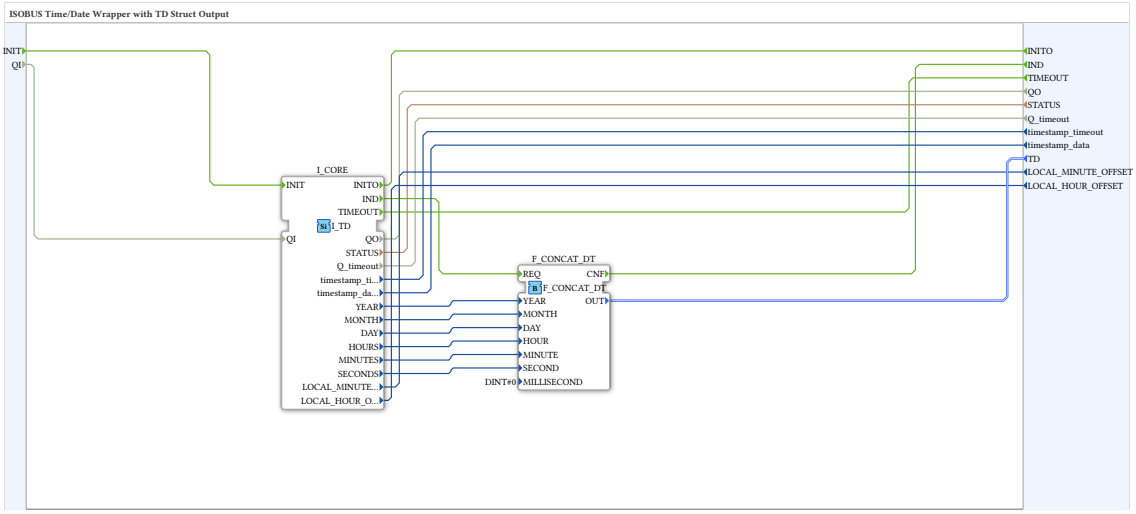
Im Vergleich zu allgemeinen IEC-61499-Zeitbausteinen (z. B. `E_CYCLE` oder `E_TIME`) ist `I_TD` speziell für die ISOBUS-Kommunikation ausgelegt. Es dekodiert keine systeminterne Uhrzeit, sondern empfängt externe Telegramme und liefert die Daten in ISOBUS-konformen Formaten. Ein universeller `I_DATE`-Baustein (falls vorhanden) wäre für proprietäre Systeme geeignet, bietet jedoch nicht die ISOBUS-SPN-Struktur.

Fazit

Der Funktionsblock `I_TD` ist ein essenzieller Baustein für ISOBUS-Anwendungen, die auf genaue Zeit- und Datumsinformationen aus dem Netzwerk angewiesen sind. Seine vollständige Abbildung der SPNs sowie die eingebaute Timeout-Behandlung machen ihn robust und normenkonform. Dank der automatischen Skalierung und Offsets können die physikalischen Werte ohne manuelle Umrechnung direkt weiterverarbeitet werden.

I_TD_TD





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **I_TD_TD** ist ein Wrapper um den Basisbaustein `I_TD` und erzeugt aus den einzelnen Zeit- und Datumskomponenten einen kombinierten IEC 61131-3 Datum/Zeit-Struct (Typ `DT`). Er dient als

Schnittstelle zu einem ISOBUS-konformen Zeit-/Datumsdienst (PGN 65254) und liefert die aufbereiteten Zeitinformationen sowie lokale Zeitverschiebungen als separate Ausgänge.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Service-Initialisierung, gesteuert durch Qualifier `QI`

Ereignis-Ausgänge

Name	Typ	Mitgeführte Daten
INITO	EInit	`QO`, `STATUS`
IND	Event	`QO`, `timestamp_data`, `STATUS`, `Q_timeout`, `LOCAL_MINUTE_OFFSET`, `LOCAL_HOUR_OFFSET`
TIMEOUT	Event	`timestamp_timeout`, `STATUS`, `Q_timeout`

Daten-Eingänge

Name	Typ	Kommentar
QI	BOOL	Eingangsqualifier (Steuerung des Initialisierungsvorgangs)

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Ausgangsqualifier (Status der Verarbeitung)
STATUS	STRING	Statusmeldung
Q_timeout	BOOL	Zeitüberschreitungs-Flag
timestamp_timeout	DINT	Zeitstempel der Zeitüberschreitung
timestamp_data	DINT	Zeitstempel der gültigen Daten
TD	DT	Kombinierte Zeit-/Datum (IEC 61131-3 `DT`)
LOCAL_MINUTE_OFFSET	SINT	Lokaler Minutenversatz (SPN 1601, Einheit: 1 min/bit, Offset −125)
LOCAL_HOUR_OFFSET	SINT	Lokaler Stundenversatz (SPN 1602, Einheit: 1 h/bit, Offset −125)

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Baustein kapselt die folgenden Schritte:

1. **Initialisierung**
   Ein Ereignis am Eingang `INIT` startet den internen Funktionsblock `I_TD` (`I_CORE`). Dieser durchläuft seinen eigenen Initialisierungsablauf und bestätigt mit `INITO`. Die Qualifier `QO` und `STATUS` werden dabei an die gleichnamigen Ausgänge weitergegeben.
2. **Zyklische Zeit-/Datumserfassung**
   Sobald der interne Baustein bereit ist, erzeugt er zyklisch ein `IND`-Ereignis. Dieses Ereignis löst den nachgeschalteten Baustein `F_CONCAT_DT` aus. `F_CONCAT_DT` setzt die vom `I_TD` bereitgestellten Einzelkomponenten (Jahr, Monat, Tag, Stunde, Minute, Sekunde) zu einem `DT`-Struct zusammen. Die

Millisekundenkomponente ist fest auf 0 gesetzt. Nach Abschluss der Konkatenation wird ein CNF-Ereignis erzeugt, das den äußeren IND-Ausgang aktiviert, zusammen mit allen weiteren Datenausgängen (Zeitstempel, Versätze, usw.).

3. **Zeitüberschreitung**
   Falls der interne Dienst eine Zeitüberschreitung meldet, wird das Ereignis TIMEOUT am Ausgang aktiviert. Die zugehörigen Daten (timestamp_timeout, STATUS, Q_timeout) werden direkt vom inneren Baustein übernommen.
4. **Weiterleitung der Rohdaten**
   Die Ausgänge timestamp_data, timestamp_timeout, LOCAL_MINUTE_OFFSET und LOCAL_HOUR_OFFSET werden unverändert vom I_TD übernommen.

Technische Besonderheiten

- **Wrapper-Prinzip**
  Der Baustein vereinfacht die Handhabung des ISOBUS-Zeitdienstes, indem er die einzelnen Zeitkomponenten zu einem plattformkompatiblen DT-Struct zusammenfasst. Der Anwender muss keine eigene Verkettung von Jahr, Monat etc. vornehmen.
- **ISO 11783-7 konform**
  Der zugrundeliegende Dienst entspricht der PGN-Nummer 65254 und stellt die nach ISO 11783-7 geforderten SPNs (1601, 1602) für lokale Zeitversätze bereit. Die Attributinformationen (Skalierung, Offset) sind in den Ausgangsvariablen vermerkt.
- **Zeitstempel-Handling**
  Sowohl für gültige Daten als auch für Timeout-Ereignisse werden separate Zeitstempel (DINT) zur Verfügung gestellt, sodass die Reihenfolge und Aktualität der Ereignisse nachverfolgt werden kann.
- **Ereignisgesteuerter Ablauf**
  Die Verarbeitung erfolgt rein ereignisgesteuert. Jeder Zyklus wird durch das IND des internen Bausteins initiiert und endet mit dem IND des Wrappers.

Zustandsübersicht

Der Baustein besitzt keinen expliziten internen Zustandsautomaten, der für den Anwender sichtbar wäre. Der Ablauf lässt sich jedoch in folgende Phasen unterteilen:

1. **Ruhezustand** – nach Initialisierung wartet der Baustein auf das erste IND des I_TD.
2. **Aktive Verarbeitung** – IND des I_TD löst F_CONCAT_DT aus; anschließend wird der äußere IND ausgegeben.
3. **Timeout-Zustand** – bei Zeitüberschreitung des I_TD wird TIMEOUT ausgegeben.
4. **Initialisierungsphase** – während INIT und INITO erfolgt die Initialisierung des internen Bausteins.

Anwendungsszenarien

- **ISOBUS-Steuergeräte**
  Einsatz in landwirtschaftlichen Steuergeräten, die regelmäßig die aktuelle UTC-Zeit und die lokale Zeitverschiebung vom ISOBUS-Bus abfragen und als einheitlichen Datentyp (DT) für Anzeige oder Logging benötigen.
- **Zeitsynchronisation**
  Verwendung in Systemen, die auf eine exakte Zeitbasis angewiesen sind, z. B. Aufzeichnungsgeräte (Logger), GPS-gestützte Anwendungen oder Prozesssteuerungen.
- **Wrapper für IEC 61131-3 Kompatibilität**
  Wenn nachfolgende Funktionsblöcke keinen Zugriff auf einzelne Zeitkomponenten, sondern nur auf einen DT-Struct erwarten, kann I_TD_TD als Bindeglied dienen.

Vergleich mit ähnlichen Bausteinen

- **I_TD (Basisbaustein)**
  Liefert die Zeitkomponenten als separate Ausgänge (YEAR, MONTH, DAY, usw.). Der Anwender müsste selbst eine Konkatenation zu einem DT vornehmen. I_TD_TD erledigt dies automatisch und fügt zusätzlich die Rohdaten-Zeitstempel hinzu.
- **Andere Zeit/Datums-Wrapper**
  Manche Implementierungen nutzen direkte Systemzeit-Funktionen. I_TD_TD ist speziell für den Einsatz

im ISOBUS-Kontext optimiert und stellt die geforderten SPN-Attribute sowie die Timeout-Behandlung bereit.

Fazit

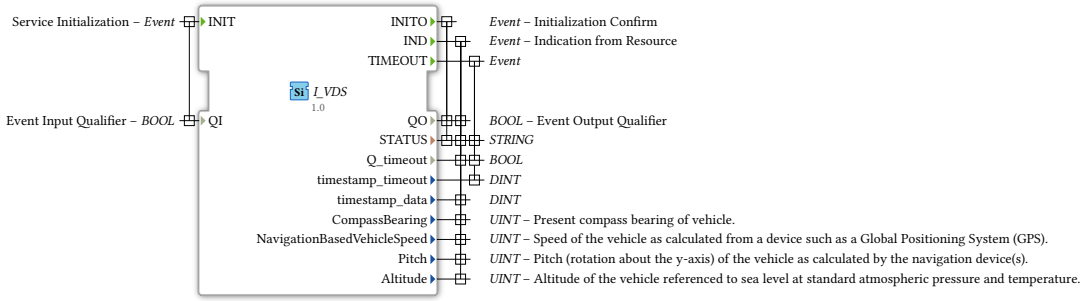
`I_TD_TD` erweitert den standardisierten ISOBUS-Zeitbaustein `I_TD` um eine direkte Ausgabe des IEC 61131-3 Datum-Zeit-Typs `DT`. Dadurch wird die Integration in Steuerungsanwendungen erleichtert, die keine separate Datumskonvertierung vornehmen möchten. Die transparente Weitergabe von Rohdaten und Timeout-Informationen sowie die durchgängige Ereignissteuerung machen den Baustein zu einer praktischen und normenkonformen Komponente für ISOBUS-basierte Systeme.

I_VDS

• – ○ • – ○ • – ○ •

Einleitung

Der **I_VDS**-Funktionsbaustein (Vehicle Direction/Speed) ist ein ISO 11783-7-konformes System zur präzisen Erfassung von Fahrzeugrichtung, Geschwindigkeit und Neigung, entwickelt unter EPL-2.0 Lizenz. Der Baustein integriert Navigationsdaten für moderne Landmaschinen und Nutzfahrzeuge.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Qualifier `QI`)

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung (mit Status)
- `IND`: Datenindikation mit Navigationsparametern (mit `QO`, `timestamp_data`, `CompassBearing`, `NavigationBasedVehicleSpeed`, `Pitch`, `STATUS`, `Q_timeout`, `Altitude`). Feuert bei jedem Zyklus ohne aktiven Timeout und publiziert dabei auch den Reset von `Q_timeout` auf FALSE
- `TIMEOUT`: Timeout-Ereignis (mit `timestamp_timeout`, `STATUS`, `Q_timeout`). Feuert ausschließlich beim tatsächlichen Eintreten eines Timeouts; das Zurücksetzen von `Q_timeout` wird nicht hier, sondern über `IND` gemeldet

Daten-Eingänge

- `QI` (BOOL): Qualifier für Initialisierung

Daten-Ausgänge

Parameter	Typ	SPN	Bit	Skalierung	Bereich	Beschreibung
`QO`	BOOL	-	1	-	TRUE/FALSE	Ereignisqualifier
`STATUS`	STRING	-	-	-	-	Systemstatusmeldung
`CompassBearing`	UINT	165	16	1/128°/bit	0-360°	Kompassrichtung
`NavigationBasedVehicleSpeed`	UINT	517	16	1/256 km/h/bit	0-250 km/h	GPS-basierte Geschwindigkeit
`Pitch`	UINT	583	16	-	0xFFFF	Nickwinkel (reserviert)

`Altitude`	UINT	580	16	-	0xFFFF	Höhe über NN (reserviert)
`Q_timeout`	BOOL	-	-	-	TRUE/FALSE	Timeout-Flag (gesetzt mit `TIMEOUT`, Reset über `IND`)
`timestamp_data`	DINT	-	-	-	-	Zeitstempel der Daten (mit `IND`)
`timestamp_timeout`	DINT	-	-	-	-	Zeitstempel des Timeouts (mit `TIMEOUT`)

Funktionsweise

1. **Initialisierung**:
   - `INIT` mit `QI`=TRUE startet GPS/Navigationssystem
   - `INITO` bestätigt Betriebsbereitschaft mit `QO` und `STATUS`
2. **Datenaktualisierung**:
   - `IND` liefert kontinuierlich aktualisierte Navigationsdaten
   - Standard-Update-Rate: 100ms für Echtzeitanwendungen
3. **Fehlerbehandlung**:
   - `TIMEOUT` bei Signalverlust oder Kommunikationsfehlern
   - Statusmeldungen im `STATUS`-Feld

Technische Besonderheiten

✓ **ISO 11783-7 konform** (PGN 65256) ✓ **Hochpräzise Richtungserfassung** mit 0.008° Auflösung ✓ **GPS-basierte Geschwindigkeit** mit 0.004 km/h Genauigkeit ✓ **Erweiterte Sensordaten** (Nickwinkel/Höhe reserviert)

Anwendungsszenarien

- **Präzisionslandwirtschaft**: Automatische Spurführung
- **Flottenmanagement**: Echtzeit-Ortung und Geschwindigkeitsüberwachung
- **Neigungserkennung**: Sicherheitssysteme für Hanglagen
- **Telematik**: Betriebsdatenerfassung und -analyse

Datenformate

Parameter	Beispielwert	Bedeutung
`CompassBearing`	16384	128° (16384 × 1/128)
`NavigationBasedVehicleSpeed`	12800	50 km/h (12800 × 1/256)
`Pitch`	0xFFFF	Nicht verwendet
`Altitude`	0xFFFF	Nicht verwendet

⚖️ **Vergleich mit ähnlichen Systemen**

Feature	I_VDS	Standard-GPS	Premium-Navi
Genauigkeit	±0.1° Richtung	±5°	±0.05°
Geschwindigkeit	±0.1%	±2%	±0.05%
Update-Rate	10Hz	1Hz	20Hz
ISO-Konformität	Voll	Teilweise	Voll

🛠️ Zugehörige Übungen

- Uebung_073
- Uebung_079

Fazit

Der I_VDS-Baustein bietet präzise Navigation für mobile Arbeitsmaschinen:

- **Präzision**: Subgrad-genaue Richtungserkennung
- **Zuverlässigkeit**: Robuste GPS-basierte Geschwindigkeit
- **Zukunftssicher**: Reservierte Parameter für Erweiterungen

Idealer Einsatz bei:

- Automatischen Lenksystemen
- Präzisionslandwirtschaft
- Flottenmanagementsystemen
- Sicherheitskritischen Anwendungen

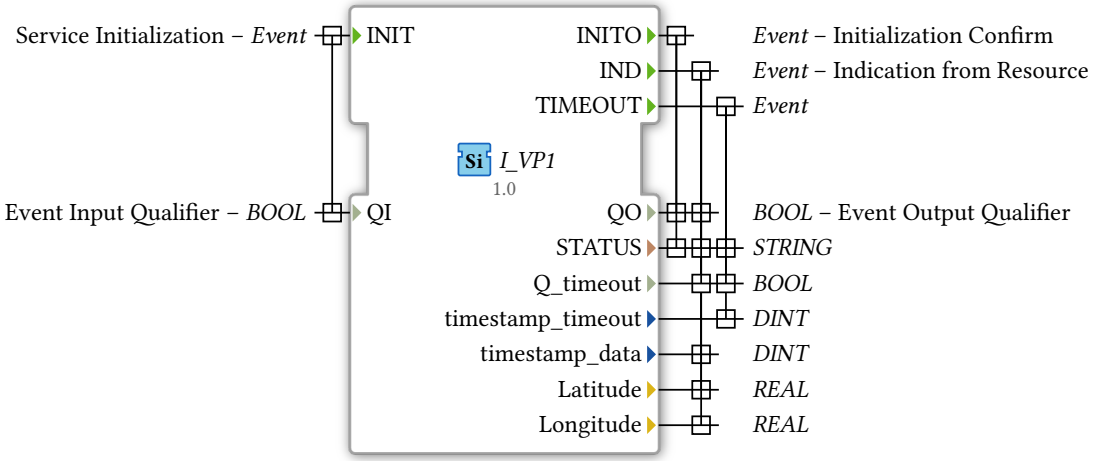
Entwickelt für hochpräzise Fahrzeugführung und -überwachung

I_VP1

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `I_VP1` dient zur Verarbeitung und Bereitstellung von Fahrzeugpositionsdaten gemäß dem ISO-11783-Standard (ISOBUS). Er implementiert speziell die Funktion „Vehicle Position 1“, die durch das Parameter Group Number (PGN) 65267 definiert ist. Der Baustein empfängt Positionsdaten von einer Ressource (z. B. einem GNSS-Empfänger) und stellt diese über standardisierte Schnittstellen für nachgelagerte Steuerungs- und Anzeigefunktionen in einem landwirtschaftlichen oder mobilen Arbeitsmaschinen-Netzwerk zur Verfügung.



Schnittstellenstruktur

Ereignis-Eingänge

- **`INIT`**: Initialisierungsereignis. Wird bei `QI=TRUE` ausgelöst, um den Baustein zu starten. Bei `QI=FALSE` wird der Baustein deaktiviert.

Ereignis-Ausgänge

- **`INITO`**: Bestätigt den Abschluss der Initialisierung. Wird als Antwort auf das `INIT`-Ereignis gesendet.
- **`IND`**: Indikationsereignis. Signalisiert den erfolgreichen Empfang neuer, gültiger Positionsdaten von der angeschlossenen Ressource.
- **`TIMEOUT`**: Zeitüberschreitungsereignis. Wird ausgelöst, wenn innerhalb eines konfigurierten Zeitrahmens keine neuen Positionsdaten empfangen wurden.

Daten-Eingänge

- **`QI` (BOOL)**: Qualifiziert das `INIT`-Ereignis. Steuert den Aktivierungszustand des Funktionsblocks (`TRUE` = aktivieren, `FALSE` = deaktivieren).

Daten-Ausgänge

- **`QO` (BOOL)**: Qualifiziert die Ereignisausgänge `INITO` und `IND`. Gibt den aktuellen Betriebszustand des Bausteins wider (`TRUE` = aktiv und betriebsbereit).
- **`STATUS` (STRING)**: Stellt Status- oder Fehlermeldungen des Bausteins in lesbarer Form bereit.
- **`Q_timeout` (BOOL)**: Signalisiert, ob ein Timeout aufgetreten ist (`TRUE` = Timeout aktiv, `FALSE` = keine Zeitüberschreitung).
- **`timestamp_timeout` (DINT)**: Zeitstempel (z. B. in Millisekunden), der dem `TIMEOUT`-Ereignis zugeordnet ist.
- **`timestamp_data` (DINT)**: Zeitstempel der zuletzt empfangenen und über `IND` ausgegebenen Positionsdaten.
- **`Latitude` (REAL)**: Der geografische Breitengrad der Fahrzeugposition in Grad. Der Wert ist gemäß PGN 65267 (SPN 584) skaliert und beinhaltet einen Offset von −210°.
- **`Longitude` (REAL)**: Der geografische Längengrad der Fahrzeugposition in Grad. Der Wert ist gemäß PGN 65267 (SPN 585) skaliert und beinhaltet einen Offset von −210°.

Adapter

Dieser Funktionsblock verwendet keine Adapter-Schnittstellen.

Funktionsweise

1. **Initialisierung**: Durch das Senden eines `INIT`-Ereignisses mit `QI=TRUE` wird der Baustein aktiviert. Er meldet den erfolgreichen Start mit dem `INITO`-Ausgangsereignis zurück, begleitet von `QO=TRUE` und einem `STATUS`-String.
2. **Datenempfang und -verarbeitung**: Im aktiven Zustand überwacht der Baustein eine angeschlossene Ressource (z. B. einen ISOBUS-Datenstrom) auf das Eintreffen von Positionsdaten gemäß PGN
3. Bei Empfang gültiger Daten werden diese dekodiert, die Skalierung und der Offset werden angewendet, und die resultierenden `Latitude`- und `Longitude`-Werte werden berechnet.
4. **Datenausgabe**: Bei erfolgreicher Verarbeitung löst der Baustein das `IND`-Ereignis aus. Gleichzeitig werden die berechneten Positionsdaten (`Latitude`, `Longitude`), ein zugehöriger Zeitstempel (`timestamp_data`), der aktive Zustand (`QO=TRUE`) und ein Status über die Ausgangsvariablen bereitgestellt. `Q_timeout` wird auf `FALSE` gesetzt.
5. **Timeout-Überwachung**: Der Baustein überwacht kontinuierlich den Datenstrom. Wenn für einen konfigurierten Zeitraum keine neuen Positionsdaten empfangen werden, löst er das `TIMEOUT`-Ereignis aus. Dabei werden `Q_timeout=TRUE`, ein Zeitstempel (`timestamp_timeout`) und ein entsprechender `STATUS` gesetzt.
6. **Deaktivierung**: Ein `INIT`-Ereignis mit `QI=FALSE` setzt den Baustein in einen inaktiven Zustand zurück, was durch `INITO` mit `QO=FALSE` quittiert wird.

Technische Besonderheiten

- **ISOBUS-Konformität**: Der Baustein ist spezifisch für den Einsatz in ISO 11783 (ISOBUS)-Netzwerken entwickelt und verarbeitet Daten gemäß der offiziellen Spezifikation für PGN 65267.
- **Datenkodierung**: Die Rohdaten für Breiten- (`Latitude`, SPN 584) und Längengrad (`Longitude`, SPN 585) werden als 4-Byte-Werte empfangen. Der Baustein wendet die in den Attributen definierte Skalierung und einen Offset von −210 Grad an, um die finalen `REAL`-Werte zu berechnen.
- **Attribut-basierte Metadaten**: Die Ausgangsvariablen für die Positionsdaten sind mit detaillierten Attributen (SPN, Name, Länge, Offset, Referenz-Link) annotiert. Dies erleichtert die Konfiguration, Dokumentation und Wartung innerhalb einer IEC 61499-Entwicklungsumgebung.

Zustandsübersicht

Der Baustein kann vereinfacht in folgende Hauptzustände übergehen:

- **Inaktiv**: Ausgangszustand. `QO=FALSE`. Keine Datenverarbeitung.
- **Aktiv (Warten auf Daten)**: Nach erfolgreicher Initialisierung. `QO=TRUE`. Der Baustein überwacht den Eingangsdatenstrom.
- **Datenverarbeitung**: Bei Empfang eines neuen PGN 65267-Datenpakets. Dekodierung, Skalierung und Berechnung der Positionswerte.

- **Datenausgabe**: Auslösen des `IND`-Ereignisses und Setzen der Ausgangsvariablen.
- **Timeout**: Auslösen des `TIMEOUT`-Ereignisses bei ausbleibenden Daten.

Anwendungsszenarien

- **Precision Farming / Teilflächenmanagement**: Bereitstellung präziser Fahrzeugpositionen für Applikationskarten, Spurführungssysteme oder Dokumentationszwecke.
- **Maschinen- und Flottenmanagement**: Ortung von landwirtschaftlichen oder kommunalen Fahrzeugen innerhalb eines Betriebsgeländes.
- **Assistenzsysteme**: Grundlage für kollisionsvermeidende Systeme oder automatische Lenksysteme, die eine genaue und standardisierte Positionsinformation benötigen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Generische Eingangsbausteinen (z. B., `E_SR`, `E_RTRIG`)**: Diese bieten grundlegende Einlese- oder Triggerfunktionen, sind aber nicht auf die Verarbeitung eines spezifischen ISOBUS-PGNs ausgelegt. `I_VP1` hingegen enthält die komplette Logik zur Dekodierung, Skalierung und Fehlerbehandlung für PGN 65267.
- **Allgemeine ISOBUS-Eingangsbausteinen**: Es könnten allgemeinere Bausteinen existieren, die verschiedene PGNs einlesen. `I_VP1` ist spezialisiert und optimiert für die effiziente und zuverlässige Verarbeitung von Fahrzeugpositionsdaten, was die Konfiguration vereinfacht und die Fehleranfälligkeit verringert.

Fazit

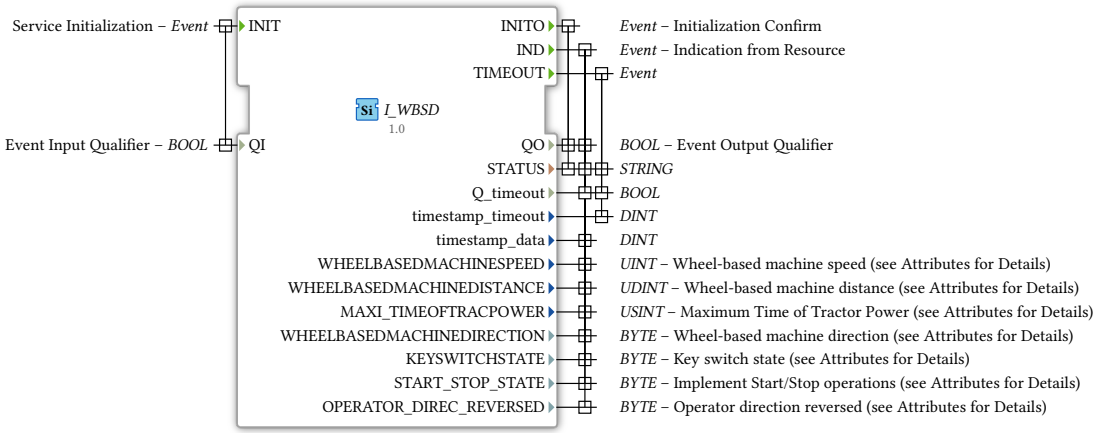
Der `I_VP1`-Funktionsblock ist ein spezialisiertes und standardkonformes Werkzeug für die Integration von Fahrzeugpositionsdaten in IEC 61499-basierte Steuerungssysteme für mobile Arbeitsmaschinen. Durch die direkte Implementierung der ISOBUS-Spezifikation (PGN 65267) und integrierte Features wie Timeout-Überwachung bietet er eine zuverlässige und einfach zu verwendende Schnittstelle für Anwendungen in den Bereichen Precision Farming und Automatisierung. Seine attributbasierte Dokumentation unterstützt Entwickler bei der korrekten Implementierung und Wartung.

I_WBSD

• – ○ • – ○ • – ○ •

Einleitung

Der **I_WBSD**-Funktionsbaustein (Wheel-based Speed and Distance) ist ein ISO 11783-7-konformes System zur Erfassung radbasierter Geschwindigkeits- und Distanzdaten, entwickelt unter EPL-2.0 Lizenz. Der Baustein ermöglicht die präzise Überwachung von Fahrzeugbewegungen und Betriebszuständen für Land- und Baumaschinen.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Qualifier `QI`)

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung (mit Status)
- `IND`: Datenindikation mit Bewegungs- und Zustandsparametern (inkl. `timestamp_data`, `Q_timeout`); feuert bei jedem Zyklus ohne aktiven Timeout und publiziert dabei auch das Zurücksetzen von `Q_timeout` auf `FALSE`
- `TIMEOUT`: Timeout-Ereignis (mit `timestamp_timeout`, `STATUS`, `Q_timeout`); feuert ausschließlich beim tatsächlichen Eintreten eines Timeouts (Entry-Trigger). Das Zurücksetzen von `Q_timeout` auf `FALSE` wird nicht hier, sondern über `IND` gemeldet

Daten-Eingänge

- `QI` (BOOL): Qualifier für Initialisierung

Daten-Ausgänge

Parameter	Typ	SPN	Bit	Skalierung	Bereich	Beschreibung
`QO`	BOOL	-	1	-	TRUE/FALSE	Ereignisqualifier
`STATUS`	STRING	-	-	-	-	Systemstatusmeldung
`Q_timeout`	BOOL	-	-	-	TRUE/FALSE	Timeout-Flag (`TRUE` bei aktivem Timeout)
`timestamp_timeout`	DINT	-	-	-	-	Zeitstempel des Timeouts (mit `TIMEOUT`)
`timestamp_data`	DINT	-	-	-	-	Zeitstempel der Daten (mit `IND`)
`WHEELBASEDMACHINESPEED`	UINT	1862	16	0.001 m/s/bit	0-65.535 m/s	Radbasierte Geschwindigkeit
`WHEELBASEDMACHINEDISTANCE`	UDINT	1863	32	0.001 m/bit	0-4,294,967 km	Zurückgelegte Distanz
`MAXI_TIMEOFTRACPOWER`	USINT	1866	8	1 min/bit	0-255 min	Maximale Betriebszeit
`WHEELBASEDMACHINEDIRECTION`	BYTE	1864	2	4 Zustände	0-3	Fahrtrichtung
`KEYSWITCHSTATE`	BYTE	1865	2	4 Zustände	0-3	Zündschalterzustand
`START_STOP_STATE`	BYTE	5203	2	4 Zustände	0-3	Start/Stop-Status
`OPERATOR_DIREC_REVERSED`	BYTE	5244	2	4 Zustände	0-3	Richtungsumkehr

Fahrtrichtungszustände

Code	Zustand	Beschreibung
0	Stillstand	Keine Bewegung
1	Vorwärts	Vorwärtsfahrt
2	Rückwärts	Rückwärtsfahrt
3	Undefiniert	Richtung nicht bestimmbar

Funktionsweise

1. **Initialisierung**:
   - `INIT` mit `QI`=TRUE startet Radsensorkalibrierung

- `INITO` bestätigt Betriebsbereitschaft mit Systemstatus

2. **Datenbereitstellung**:
   - `IND` liefert kontinuierlich aktualisierte Bewegungsdaten
   - Automatische Anpassung der Update-Rate (100ms bei >0.5 m/s)
3. **Fehlerbehandlung**:
   - `TIMEOUT` bei Signalverlust der Raddrehzahlsensoren
   - Statusmeldungen im `STATUS`-Feld

Technische Besonderheiten

✓ **ISO 11783-7 konform** (PGN 65096) ✓ **Präzise Geschwindigkeitsmessung** mit 1mm/s Auflösung ✓ **32-bit Distanzzähler** für lange Betriebszeiten ✓ **Integrierte Zustandsüberwachung** (Zündung, Start/Stop)

Anwendungsszenarien

- **Traktoren**: Geschwindigkeitsregelung für Feldarbeit
- **Erntemaschinen**: Arbeitsstreckenberechnung
- **Flottenmanagement**: Betriebsstundenerfassung
- **Sicherheitssysteme**: Richtungserkennung bei Rückwärtsfahrt

Zustandscodes

Parameter	Code	Bedeutung
`KEYSWITCHSTATE`	0	Aus
	1	Ein
	2	Start
	3	Undefiniert
`START_STOP_STATE`	0	Stopp
	1	Start
	2	Pause
	3	Reserviert

⚖️ Vergleich mit ähnlichen Systemen

Feature	I_WBSD	Standard	GPS-basiert
Genauigkeit	±0.5%	±2%	±5%
Niedriggeschwindigkeit	Gut	Ausgezeichnet	Schlecht
Signalstabilität	Hoch	Mittel	Niedrig
ISO-Konformität	Voll	Teilweise	Voll

🛠️ Zugehörige Übungen

- Uebung_070
- Uebung_071
- Uebung_071a
- Uebung_071b
- Uebung_072
- Uebung_072b
- Uebung_079

Fazit

Der I_WBSD-Baustein bietet zuverlässige Bewegungsdaten für mobile Arbeitsmaschinen:

- **Robust**: Unabhängig von GPS-Signal
- **Präzise**: Millimeter-genaue Distanzmessung
- **Umfassend**: Integrierte Zustandsüberwachung

Idealer Einsatz bei:

- Präzisionslandwirtschaft
- Automatischen Lenksystemen
- Maschinen mit häufigen Richtungswechseln
- Anwendungen mit hohen Genauigkeitsanforderungen

Entwickelt für den Einsatz unter anspruchsvollen Umgebungsbedingungen

2.1.13.4 UT

🎧 Podcast

- 4diac IDE: Dein „Hello World“ der Automatisierung – Das Blinking Tutorial Lokal
- 4diac IDE: Dein Open-Source-Werkzeugkasten für verteilte Industrieautomatisierung nach IEC 61499
- 4diac IDE: Wie der IEC 61499 Standard die Industrieautomatisierung revolutioniert
- Den Software-Drachen zähmen: Industrielle Automatisierung und die Zukunft der Produktion
- Eclipse 4diac 3.0: ST-Interpreter, FBE und 7200 Commits – Der Turbo für verteilte Automatisierung

📺 Video

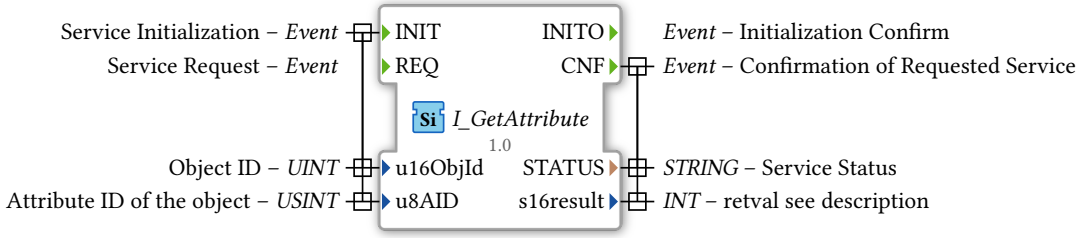
- Automation der Zukunft
- The Dual Wi Fi Solution

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

I

I_GetAttribute



• – ◦ • – ◦ • – ◦ •

Einleitung

`I_GetAttribute` implementiert das Kommando **Get Attribute Value** (ISO 11783-6, F.58). Es steht ab VT-Version 4 zur Verfügung und dient einem Working Set dazu, beim VT den aktuellen Zustand von Objekten im VT abzufragen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Service-Initialisierung
  - Verknüpft mit: `u16ObjId`, `u8AID`
- **REQ**: Service-Anfrage

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung
- **CNF**: Bestätigung der angeforderten Service-Anfrage
  - Verknüpft mit: `STATUS`, `s16result`

Daten-Eingänge

- **u16ObjId** (`UINT`, Initialwert `ID_NULL`): Objekt-ID
- **u8AID** (`USINT`): Attribut-ID des Objekts

Daten-Ausgänge

- **STATUS** (`STRING`): Service-Status
- **s16result** (`INT`): Rückgabewert (siehe unten)

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Das Kommando „Get Attribute Value“ (F.58) ist eine feste 8-Byte-Nachricht (Objekt-ID in Byte 2 und 3, Attribut-ID in Byte 4; kein Transport Protocol).

`INIT` prüft vor dem Senden des Kommandos sowohl den Objekttyp (`iso_has_readable_attribute_id`: besitzt dieser Objekttyp überhaupt eine Attribut-ID, schreibgeschützt oder beschreibbar?) als auch die konkrete Attribut-ID gegen die Attributtabelle dieses Typs (`iso_is_readable_attribute`). Anders als bei Change Attribute werden hier sowohl schreibgeschützte als auch beschreibbare Attribut-IDs akzeptiert. Die zugrunde liegende C-Implementierung (`cmd_get_attribute_value`) führt selbst keine Typprüfung durch, nur die `ID_NULL`-Behandlung; das VT validiert Objekt-ID und Attribut-ID zusätzlich auf seiner Seite und antwortet bei ungültigen Werten mit einer Fehlerantwort.

`ID_NULL` (65535) ist kein Kommandoziel, kann aber in `INIT` verwendet werden, um den Funktionsblock zu deaktivieren.

Rückgabewerte (`s16result`)

Wert	Bedeutung
`VT_E_NO_ERR` (0)	OK
`VT_E_DEACTIVATED` (-40)	FB über `ID_NULL` bei `INIT` deaktiviert
`VT_E_INVALID_OBJECT_ID` (-132)	Der Typ der Objekt-ID hat keine lesbare Attribut-ID (`iso_has_readable_attribute_id`)
`VT_E_INVALID_ATTRIBUTE_ID` (-133)	Attribut-ID ist für diesen Objekttyp keine gültige/lesbare AID (`iso_is_readable_attribute`)
`VT_E_NOT_READY` (-131)	Gepuffert: `INIT` noch nicht abgeschlossen / VT noch nicht bereit
`VT_E_OVERFLOW` (-6)	Pufferüberlauf
`VT_E_NOACT` (-8)	Kommando im aktuellen Zustand nicht möglich
`VT_E_NO_INSTANCE` (-21)	Kein VT-Client verfügbar
`VT_E_HANDLE_INVALID` (-128)	Fehlerursache: ungültiges Handle
`VT_E_ISO_INSTANCE_INVALID` (-129)	Kennung für Instanz (Verbindung) ungültig
`VT_E_NOT_ALIVE` (-130)	Kennung für Instanz gültig, aber VT tot

Technische Besonderheiten

- **Instanz-Eindeutigkeit**: Dieser Baustein erfordert Eindeutigkeit bezüglich des Paares (`u16ObjId`, `u8AID`). Im gesamten Programm darf nur eine Instanz von `I_GetAttribute` für dasselbe Objekt-Attribut-Paar existieren. `u8AID` wird bei `INIT` eingelesen; eine zweite Instanz auf dieselbe Objekt-ID und Attribut-ID wird bei `INIT` deaktiviert.
- Der Baustein liefert über `CNF` nur Status und Rückgabewert; den gelesenen Attributwert liefern die Service-Interface-Bausteine wie `Attribute_IDA`.

Anwendungsszenarien

- Abfragen des aktuellen Zustands eines Objekts im VT (ab VT-Version 4)

Fazit

`I_GetAttribute` sendet das Kommando Get Attribute Value (F.58) für ein Objekt-/Attribut-Paar und meldet Status und Rückgabewert.

Q

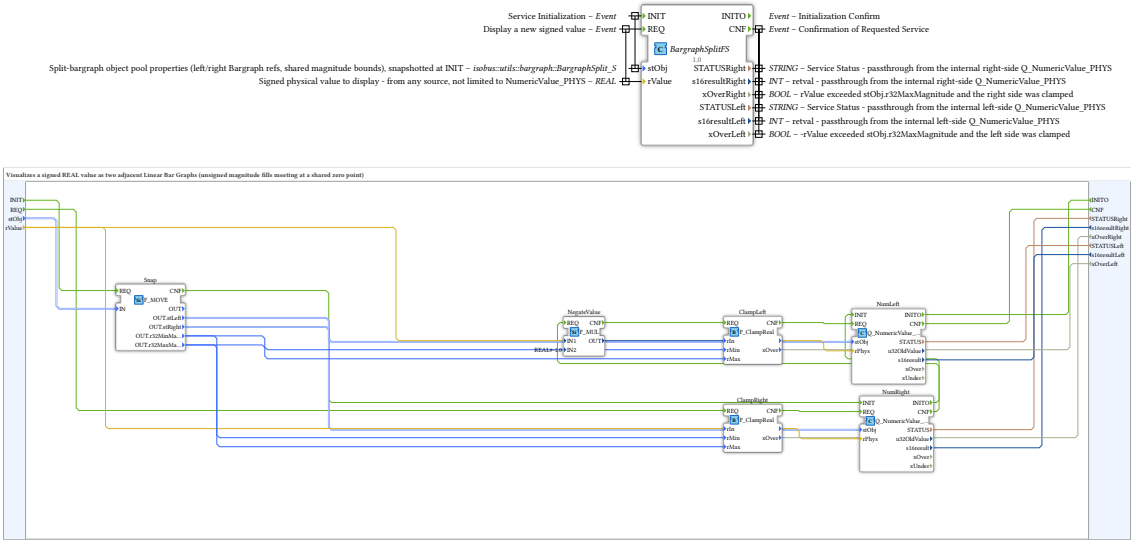
📌 Architektur & Richtlinien

- Instanz-Eindeutigkeit der Q_*-Bausteine

🎧 Podcast

- Das Geheimnis des 2-Bit-Quarter: Effizienz im CAN-Bus für Nutzfahrzeuge
- QUARTER
- 0 und 1: Warum das Binärsystem trotz Quantensprung die Grundlage unserer digitalen Welt bleibt
- Open Circuits: Die verborgene Ästhetik und faszinierende Komplexität im Querschnitt unserer Elektronik

BargraphSplitFS



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **BargraphSplitFS** visualisiert einen vorzeichenbehafteten REAL-Wert als zwei nebeneinanderliegende lineare Balkendiagramme (Linear Bar Graphs). Die beiden Balken treffen sich an einem gemeinsamen Nullpunkt, wobei positive Werte auf der rechten Seite und negative Werte auf der linken Seite dargestellt werden. Der Baustein kapselt zwei Instanzen des ISO 11783-6 Bausteins `Q_NumericValue_PHYS` und übernimmt die Aufbereitung des Eingangswertes sowie die Begrenzung auf den zulässigen Magnitudenbereich.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
`INIT`	EInit	Service Initialization; übernimmt den Objekt-Pool `stObj` per F_MOVE-Snapshot
`REQ`	Event	Anforderung zur Anzeige eines neuen vorzeichenbehafteten Wertes

Ereignis-Ausgänge

Name	Typ	Kommentar

INITO	EInit	Bestätigung der Initialisierung
CNF	Event	Bestätigung der Verarbeitung; liefert Status und Ergebnis der beiden internen Q_NumericValue_PHYS-Instanzen sowie Überlauf-Flags

Daten-Eingänge

Name	Typ	Kommentar
stObj	isobus::utils::bargraph::BargraphSplit_S	Objekt-Pool mit links/rechts Bargraph-Referenzen und gemeinsamen Magnituden-Grenzen; wird bei INIT übernommen
rValue	REAL	Vorzeichenbehafteter physikalischer Wert zur Anzeige (beliebige Quelle)

Daten-Ausgänge

Name	Typ	Kommentar
STATUSRight	STRING	Statusmeldung der rechten Bargraph-Instanz (Passthrough)
s16resultRight	INT	Rückgabewert der rechten Bargraph-Instanz (Passthrough)
xOverRight	BOOL	Wird TRUE, wenn rValue den Maximum-Magnitudenwert überschreitet und die rechte Seite begrenzt wurde
STATUSLeft	STRING	Statusmeldung der linken Bargraph-Instanz (Passthrough)
s16resultLeft	INT	Rückgabewert der linken Bargraph-Instanz (Passthrough)
xOverLeft	BOOL	Wird TRUE, wenn der negierte Wert den Maximum-Magnitudenwert überschreitet und die linke Seite begrenzt wurde

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Baustein arbeitet nach folgendem Prinzip:

1. **Initialisierung (INIT)**: Der eingehende stObj wird über einen F_MOVE-Baustein in eine interne, stabile Kopie übernommen. Dies entspricht dem in verwandten Bausteinen (z.B. ScrollFS, PositionMarkerFS) verwendeten Muster, um spätere Änderungen am Objektpool während des Betriebs zu verhindern.
2. **Wertaufbereitung (REQ)**:
   - **Positive Werte** (rValue > 0): Der Wert wird direkt an einen F_ClampReal-Baustein übergeben, der ihn auf den Bereich [0, r32MaxMagnitude] begrenzt. Das begrenzte Ergebnis wird als physikalischer Wert an die rechte Q_NumericValue_PHYS-Instanz übergeben. Die linke Instanz erhält den Wert 0 (kein Ausschlag).
   - **Negative Werte** (rValue < 0): Der Wert wird zunächst durch einen F_MUL-Baustein mit -1.0 multipliziert (Negation). Das positive Ergebnis wird dann durch einen zweiten F_ClampReal begrenzt und an die linke Q_NumericValue_PHYS-Instanz übergeben. Die rechte Seite erhält den Wert 0.
   - **Wert = 0**: Beide Seiten erhalten den Wert 0 und zeigen keine Magnitude an.
3. **Überwachung**: Tritt eine Überschreitung des Maximalwerts (r32MaxMagnitude) auf, setzt der jeweilige F_ClampReal das Flag xOver und das entsprechende xOverRight/xOverLeft wird ausgegeben. Es gibt bewusst keine xUnder-Ausgänge, da ein Unterschreiten des Minimalwerts (0) auf der inaktiven Seite den normalen Ruhezustand darstellt und keine Diagnose erfordert.
4. **Ausgabe**: Nach Verarbeitung eines REQ-Ereignisses wird CNF ausgelöst und die Status- sowie Ausgabewerte der beiden internen Q_NumericValue_PHYS-Instanzen durchgereicht.

Technische Besonderheiten

- **Snapshot-Mechanik**: Der `stObj` wird nur einmalig bei `INIT` übernommen. Dies verhindert, dass Änderungen an der übergebenen Struktur während des laufenden Betriebs die Anzeige unkontrolliert beeinflussen.
- **Zwei unabhängige Bargraph-Instanzen**: Die Verwendung von zwei separaten `Q_NumericValue_PHYS`-Bausteinen ermöglicht eine getrennte Ansteuerung der linken und rechten Balkengrafik gemäß ISO 11783-6.
- **Keine zusätzliche Skalierung**: Der Baustein akzeptiert jeden REAL-Wert, solange der physikalische Bereich der `Q_NumericValue_PHYS`-Instanzen durch die Begrenzung eingehalten wird. Eine externe Skalierung ist nicht erforderlich.
- **Kaskadierte Verarbeitung**: Die Ereigniskette verarbeitet rechts und links nacheinander, sodass eine klare Reihenfolge der Aktualisierung gewährleistet ist.

Zustandsübersicht

Der Baustein besitzt keine explizite Zustandsmaschine, sondern folgt einem ereignisgesteuerten Ablauf:

- **Initial** (nach Abschluss von `INIT`): Alle internen Instanzen sind initialisiert, die Ausgänge sind gültig.
- **Bereit** (wartend auf `REQ`): Der Baustein wartet auf neue Werte.
- **Verarbeitung** (während `REQ`): Die Eingangswerte werden verarbeitet, die Ausgänge werden aktualisiert und `CNF` wird ausgelöst.

Fehlersituationen (z.B. ungültiger `stObj`) werden nicht speziell behandelt; sie würden sich in den Status-Ausgängen der `Q_NumericValue_PHYS`-Instanzen widerspiegeln.

Anwendungsszenarien

- **Maschinensteuerung (ISO 11783)**: Anzeige von Betriebsparametern, die sowohl positive als auch negative Werte annehmen können, z.B. Differenzdrehzahl, Druckdifferenz oder Temperaturabweichung von einem Sollwert.
- **Instrumententafeln**: Darstellung von Füllständen oder Messwerten mit bilateralem Bereich, bei dem positive und negative Abweichungen visuell getrennt dargestellt werden.
- **Diagnose und Monitoring**: Visualisierung von Regelabweichungen in Steuerungssystemen, wobei die getrennte Anzeige eine schnelle Erfassung der Richtung ermöglicht.

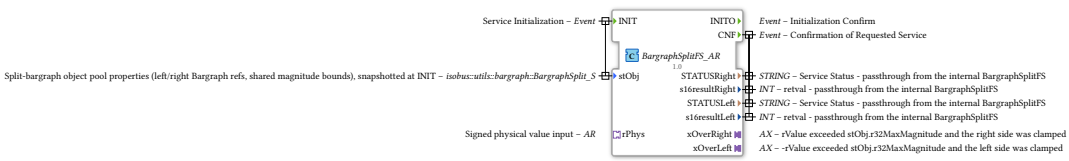
Vergleich mit ähnlichen Bausteinen

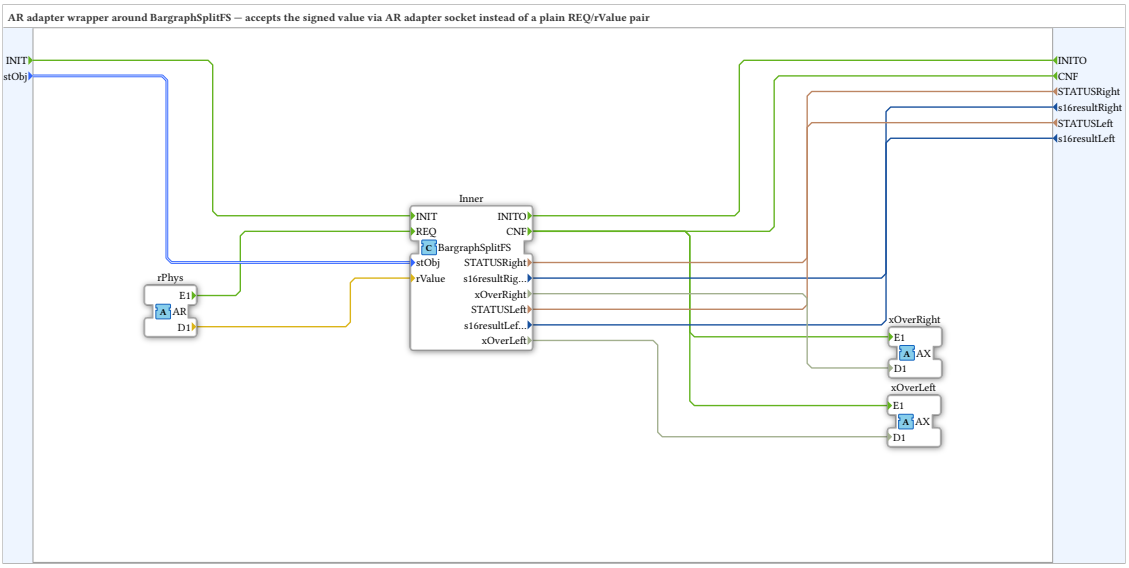
Im Gegensatz zu einem einfachen `Q_NumericValue_PHYS`, das nur einen einzelnen Bargraph ansteuert, bietet `BargraphSplitFS` eine integrierte Aufteilung in zwei Bereiche. Dies spart externe Logik und vereinfacht die Anbindung an ein Anzeigesystem mit gemeinsamer Nullpunktbasis. Ähnliche Bausteine (z.B. ein einzelner `Bargraph` mit negativem Wertebereich) würden entweder eine separate Vorzeichenbehandlung oder eine aufwändigere Konfiguration erfordern. `BargraphSplitFS` kapselt diese Komplexität und bietet eine klare Schnittstelle für den Anwendungsentwickler.

Fazit

Der Baustein `BargraphSplitFS` stellt eine robuste und effiziente Lösung zur Visualisierung vorzeichenbehafteter Messwerte auf zwei getrennten Balkendiagrammen dar. Durch die Kapselung der Aufbereitungslogik und die Verwendung bewährter ISO-11783-Bausteine wird eine hohe Wiederverwendbarkeit erreicht. Die klare Schnittstelle mit separaten Status- und Überlaufmeldungen erleichtert die Integration in übergeordnete Systeme und ermöglicht eine präzise Diagnose. Ohne unnötige Komplexität erfüllt er die Anforderungen typischer Anzeigeszenarien in der Automatisierungstechnik.

BargraphSplitFS_AR





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **BargraphSplitFS_AR** ist ein Adapter-Wrapper um den Baustein **BargraphSplitFS**. Er erweitert dessen Funktionalität, indem er die Eingabe eines vorzeichenbehafteten physikalischen Werts nicht über ein herkömmliches `REQ`/`rValue`-Paar, sondern über einen AR-Adapter-Socket (unidirektional) entgegennimmt. Dadurch lässt sich der Baustein nahtlos in eine adapterbasierte Kommunikationsstruktur einbinden, wie sie beispielsweise in ISO-11783-Netzwerken (ISOBUS) üblich ist. Die Ausgänge für Status und Ergebnisse sowie die Überlauf-Meldungen werden unverändert durchgereicht.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Mit Variablen	Kommentar
`INIT`	`EInit`	`stObj`	Service-Initialisierung; übernimmt die Objekt-Pool-Eigenschaften für das Split-Bargraph.

Ereignis-Ausgänge

Ereignis	Typ	Mit Variablen	Kommentar
`INITO`	`EInit`	–	Bestätigung der Initialisierung.
`CNF`	`Event`	`STATUSRight`, `s16resultRight`, `STATUSLeft`, `s16resultLeft`	Bestätigung des angeforderten Dienstes; enthält Status- und Ergebniswerte beider Bargraph-Seiten.

Daten-Eingänge

Variable	Typ	Kommentar
`stObj`	`isobus::utils::bargraph::BargraphSplit_S`	Objekt-Pool-Eigenschaften (Referenzen auf linken/rechten Bargraph, gemeinsame Magnituden-Grenzen), werden bei `INIT` übernommen.

Daten-Ausgänge

Variable	Typ	Kommentar

STATUSRight	STRING	Dienststatus der rechten Seite – Durchreichung vom internen Bargraph-SplitFS.
s16resultRight	INT	Ergebniswert (Rückgabewert) der rechten Seite – Durchreichung.
STATUSLeft	STRING	Dienststatus der linken Seite – Durchreichung.
s16resultLeft	INT	Ergebniswert der linken Seite – Durchreichung.

Adapter

Adapter	Richtung	Typ	Kommentar
rPhys	Socket (Eingang)	adapter::types::unidirectional::AR	Vorzeichenbehafteter physikalischer Eingangswert.
xOverRight	Plug (Ausgang)	adapter::types::unidirectional::AX	Signalisiert, dass der Wert die rechte Magnituden-Grenze überschritten hat (geklemmt).
xOverLeft	Plug (Ausgang)	adapter::types::unidirectional::AX	Signalisiert, dass der negative Wert die linke Magnituden-Grenze überschritten hat (geklemmt).

Funktionsweise

Der Baustein **BargraphSplitFS_AR** enthält intern eine Instanz des Funktionsblocks BargraphSplitFS (Name: Inner). Beim INIT-Ereignis wird das übergebene stObj direkt an den internen Baustein weitergeleitet und dort initialisiert. Der eingehende Wert am AR-Adapter rPhys wird über dessen Datenausgang D1 auf das Dateneingang rValue des internen Bausteins gespeist; das Ereignis E1 des Adapters löst dabei das REQ-Ereignis des internen Bausteins aus. Dadurch wird die Verarbeitung ohne zusätzliche manuelle Triggerung gestartet.

Die Ergebnisse und Statusmeldungen des internen Bausteins werden unverändert auf die entsprechenden Ausgänge des Adapter-Wrappers durchgeschaltet. Die Überlauf-Signale (xOverRight, xOverLeft) werden als AX-Adapter-Plugs zur Verfügung gestellt und vom internen Baustein übernommen. Das CNF-Ereignis des internen Bausteins wird sowohl auf den CNF-Ausgang des Wrappers als auch auf die Ereigniseingänge der beiden AX-Plugs gelegt, sodass eine Weiterverarbeitung synchron erfolgen kann.

Technische Besonderheiten

- **Adapterbasierte Eingabe:** Der Wert wird nicht über ein separates REQ-Ereignis und eine Datenvariable, sondern über einen AR-Adapter empfangen. Dadurch ist der Baustein besonders für modulare, adaptergesteuerte Datenflussarchitekturen geeignet.
- **Reine Durchreichung:** Der Wrapper fügt keine zusätzliche Logik hinzu, sondern leitet alle Signale transparent an die interne BargraphSplitFS-Instanz weiter.
- **Zwei getrennte Überlauf-Ausgänge:** Die Überlauf-Informationen für rechte und linke Seite werden separat als AX-Adapter-Plugs exponiert.
- **Direkte Kopplung von Ereignis und Daten:** Das Auslösen der Verarbeitung erfolgt automatisch durch das Eintreffen eines neuen Werts am AR-Adapter – es ist kein externes Triggern erforderlich.

Zustandsübersicht

Der Baustein selbst besitzt keine eigene Zustandsmaschine. Er delegiert die gesamte Zustandslogik an den internen BargraphSplitFS. Der interne Baustein durchläuft dabei typischerweise Zustände wie:

- **IDLE**: Warten auf ein REQ-Ereignis.
- **BUSY**: Verarbeitung läuft.
- **DONE**: Verarbeitung abgeschlossen, CNF wird ausgelöst.

Da der Wrapper diese Zustände nicht verändert, ist eine gesonderte Zustandsbeschreibung nicht erforderlich.

Anwendungsszenarien

- **ISOBUS-Anzeigen:** Einsatz in Traktor- oder Anbaugeräten, bei denen ein Bargraph mit zwei separaten Anzeigebereichen (links/rechts) über einen Adapter mit einem zentralen Steuergerät verbunden ist.
- **Adapterbasierte Wertübergabe:** Überall dort, wo Werte nicht über klassische Datenports, sondern über AR/AX-Adapter ausgetauscht werden sollen – z. B. in einer modularen Objekt-Pool-Architektur nach ISO 11783-6.
- **Erweiterung bestehender Systeme:** Der Baustein erlaubt es, ein bestehendes `BargraphSplitFS`-Design ohne Änderung des Kernalgorithmus in ein adapterorientiertes Umfeld zu integrieren.

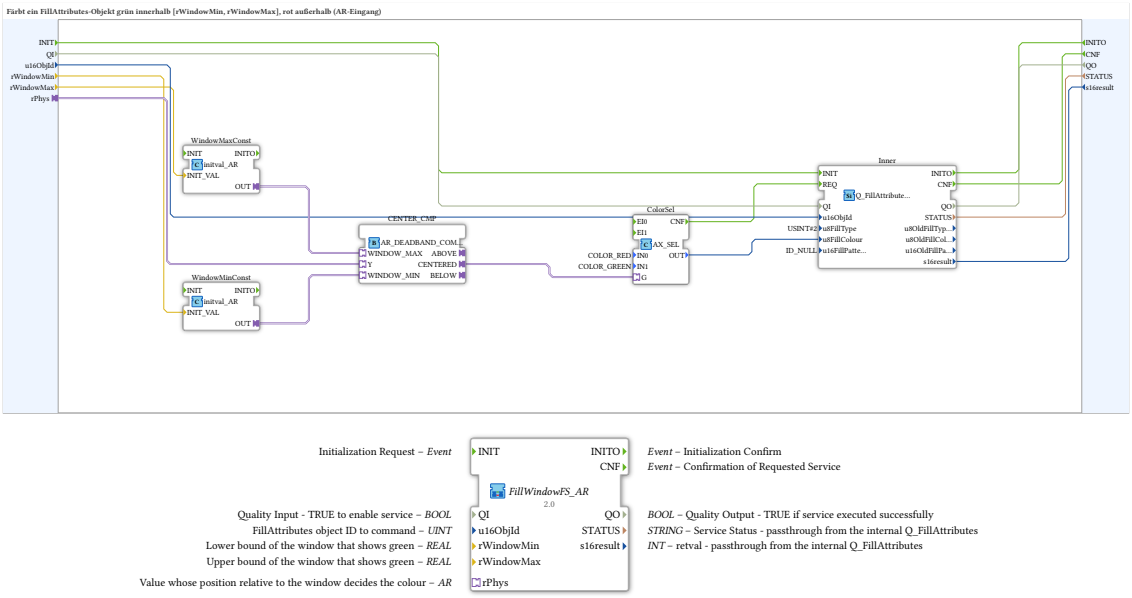
Vergleich mit ähnlichen Bausteinen

Gegenüber dem Basisbaustein `BargraphSplitFS` bietet der Wrapper eine **adapterbasierte Eingabe** statt einer direkten `REQ`/`rValue`-Schnittstelle. Dies vereinfacht die Einbindung in Adapter-Topologien und erhöht die Wiederverwendbarkeit in modularen Systemen. Ähnliche Wrapper existieren z. B. als `PositionMarkerFSA` für Positionsmarker; `BargraphSplitFS_AR` folgt dem gleichen Entwurfsmuster und stellt die Überlauf-Signale analog über AX-Plugs bereit. Im Gegensatz zu einer vollständig eigenständigen Neuimplementierung bleibt die bewährte Funktionalität des Kernbausteins unverändert erhalten.

Fazit

BargraphSplitFS_AR ist ein sauber implementierter Adapter-Wrapper, der die Funktionalität von `BargraphSplitFS` um eine AR-basierte Eingangsschnittstelle ergänzt. Er eignet sich hervorragend für den Einsatz in ISOBUS-Anwendungen und adapterorientierten Architekturen. Durch die transparente Durchreichung aller relevanten Signale bleiben Verhalten und Zuverlässigkeit des ursprünglichen Bausteins vollständig erhalten, während die Integration in moderne Kommunikationsmuster erleichtert wird.

FillWindowFS_AR



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsbaustein **FillWindowFS_AR** dient dazu, ein Objekt vom Typ `FillAttributes` (z. B. zur farblichen Hinterlegung in Bedienoberflächen) in Abhängigkeit eines über einen Adapter zugeführten Realwertes einzufärben. Liegt der Wert innerhalb eines definierten Fensters `[rWindowMin, rWindowMax]`, wird das Objekt grün dargestellt; außerhalb dieses Fensters rot. Die Fensterprüfung übernimmt ein interner `AR_DEADBAND_COMPARE`-Baustein, die eigentliche Farbänderung ein interner `Q_FillAttributes`-Baustein, dem die Farbkonstanten über eine `AX_SEL`-Auswahl zugeführt werden.

Der Baustein ist als SubAppTyp realisiert und nutzt die Adaptertechnik, um den zu vergleichenden Wert flexibel über einen unidirektionalen `AR`-Adapter zu erhalten.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Initialisierungsanforderung – wird an den internen `Q_FillAttributes` (`Inner.INIT`) durchgereicht

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Initialisierungsbestätigung (von `Inner.INITO`)
CNF	Event	Bestätigung der angeforderten Färbung (passthrough vom internen Q_FillAttributes)

Daten-Eingänge

Name	Typ	Initialwert	Kommentar
QI	BOOL	TRUE	Quality Input – TRUE, um den Dienst freizugeben (an `Inner.QI`)
u16ObjId	UINT	ID_NULL	Objekt-ID des zu färbenden FillAttributes-Objekts
rWindowMin	REAL	–	Untere Grenze des grünen Fensters
rWindowMax	REAL	–	Obere Grenze des grünen Fensters

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde (von `Inner.QO`)
STATUS	STRING	Statusmeldung des internen Q_FillAttributes
s16result	INT	Rückgabewert des internen Q_FillAttributes

Adapter

Name	Typ	Kommentar
rPhys	`adapter::types::unidirectional::AR`	Realwert, dessen Position relativ zum Fenster die Farbe bestimmt

Funktionsweise

Der über den Adapter `rPhys` ankommende Realwert wird intern einem `AR_DEADBAND_COMPARE`-Baustein (`CENTER_CMP`) als Messwert `Y` zugeführt. Die Fenstergrenzen `rWindowMin` und `rWindowMax` werden über zwei `initval_AR`-Bausteine (`WindowMinConst`, `WindowMaxConst`) in AR-Adapter-Werte gewandelt und an `WINDOW_MIN` bzw. `WINDOW_MAX` von `CENTER_CMP` gelegt.

`AR_DEADBAND_COMPARE` meldet über den AX-Plug `CENTERED`, ob `WINDOW_MIN ≤ Y ≤ WINDOW_MAX` gilt (die Plugs `ABOVE`/`BELOW` werden hier nicht verwendet). `CENTERED` steuert den Auswahlbaustein `ColorSel` (`AX_SEL`), der zwischen der roten und der grünen Farbkonstante umschaltet: `TRUE` ⟶ `COLOR_GREEN` (`IN1`), `FALSE` ⟶ `COLOR_RED` (`IN0`).

Die ausgewählte Farbe wird zusammen mit der Objekt-ID an den internen `Q_FillAttributes`-Baustein (`Inner`) übergeben: `ColorSel.CNF` ⟶ `Inner.REQ`, `ColorSel.OUT` ⟶ `Inner.u8FillColour`, `u16ObjId` ⟶ `Inner.u16ObjId`. Dieser führt die eigentliche Färbung durch und bestätigt die Ausführung über den Ereignisausgang `CNF`. Die Ausgänge `STATUS` und `s16result` übernehmen die vom `Q_FillAttributes` gemeldeten Werte.

Initialisierung und Quality-Signale werden direkt durchgereicht: `INIT` ⟶ `Inner.INIT`, `Inner.INITO` ⟶ `INITO`, `QI` ⟶ `Inner.QI`, `Inner.QO` ⟶ `QO`.

Technische Besonderheiten

- Der Baustein ist als **SubAppTyp** implementiert und nutzt ausschließlich interne Funktionsblöcke – kein eigenständiger Quellcode.
- Die Farben sind als Konstanten `COLOR_GREEN` und `COLOR_RED` vorgegeben und können nicht zur Laufzeit geändert werden.
- Außer `INIT` gibt es **keine Ereignis-Eingänge** die Färbung wird durch das Eintreffen neuer Daten am Adapter `rPhys` (bzw. an den Fenstergrenzen) ausgelöst.
- Die Fensterprüfung ist **inklusiv** (≥ und ≤), d. h. die Grenzen selbst gehören zum grünen Bereich. Sie übernimmt `AR_DEADBAND_COMPARE` (ab Version 2.0 des Bausteins, vorher eigene Split-/GE-/LE-/UND-Verdrahtung).
- Der interne `Q_FillAttributes`-Baustein wird mit festen Parametern (`u8FillType = USINT#2`, `u16FillPatternId = ID_NULL`) betrieben, die nicht nach außen geführt sind.
- Quality-Muster: `QI` (Vorgabe `TRUE`) gibt den Dienst frei, `QO` meldet den Erfolg des internen `Q_FillAttributes`.

Zustandsübersicht

Der Funktionsblock besitzt keine expliziten internen Zustände (der Vergleich in `AR_DEADBAND_COMPARE` meldet nur Änderungen seiner Ausgänge). Nach dem Anlegen aller Eingangsdaten (Adapterwert und Fenstergrenzen) wird die Färbung ausgeführt und über `CNF` bestätigt; `INIT`/`INITO` werden nur durchgereicht.

Anwendungsszenarien

- **Visualisierung von Prozesswerten** in HMI-Panels, z. B. Temperatur, Druck oder Füllstand, mit farblicher Kennzeichnung innerhalb eines Sollbereichs.
- **Grenzwertüberwachung** in Maschinensteuerungen, bei der ein zulässiger Betriebsbereich grün und Bereiche außerhalb rot dargestellt werden sollen.
- **Alarm- und Statusanzeigen**, die nur zwischen „OK“ und „Nicht OK“ unterscheiden müssen und keine stufenweisen Zustände erfordern.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu allgemeinen Vergleichs- oder Auswahlbausteinen vereint `FillWindowFS_AR` mehrere Standardfunktionen (Fenstervergleich, Auswahl) in einer spezialisierten Einheit. Gegenüber einem manuell verdrahteten Netzwerk aus einzelnen FBs bietet dieser Baustein eine kompakte, wiederverwendbare Lösung für die beschriebene Färbungsaufgabe. Andere Bausteine, die mit `FillAttributes` arbeiten, geben oft nur eine einzige Farbe vor oder besitzen zusätzliche Ereignis-Eingänge zur gezielten Steuerung; dieser Baustein reagiert rein datengetrieben über das Adaptersignal.

Fazit

`FillWindowFS_AR` ist ein praxisorientierter Funktionsbaustein, der die einfache farbliche Kennzeichnung eines `FillAttributes`-Objekts in Abhängigkeit eines Fensters realisiert. Durch die Verwendung eines AR-Adapters kann der zu bewertende Wert flexibel aus unterschiedlichen Quellen gespeist werden. Der Baustein ist sofort einsetzbar und reduziert den Verdrahtungsaufwand erheblich, da er die komplette Vergleichs- und Auswahllogik integriert.

Instanz-Eindeutigkeit der Q_*-Bausteine

Jeder `Q_*`-Baustein hier führt seine eigene, unabhängige Wert-/Pending-Pufferung (`var_qPending`, `var_*OldValue`, siehe `sendXxx()`/`resetPendingState()`). Werden zwei Instanzen DERSELBEN Klasse auf dasselbe Ziel (Objekt-ID, Masken-ID o. ä.) verdrahtet, wissen sie nichts voneinander - beide puffern/senden unabhängig, konkurrieren um denselben VT-Bus-Befehl. Deshalb hat jeder Baustein ein Eindeutigkeits-Kriterium: was darf es höchstens einmal geben.

Seit `792e35673` („Detect duplicate object-ID claims across instances of the same FB class“) gibt es dafür einen Mechanismus: `CQ_VTCommandBase::mTargetObjectId` (+ optional `mTargetSubId` für Bausteine mit einem zusammengesetzten Schlüssel, siehe `Q_Attribute` unten) + `IsObjectIdAlreadyClaimed(objId, subId = 0xFFFF)` (`CQ_VTCommandBase.h/.cpp`) - die zweite Instanz, die bei INIT ein bereits beanspruchte (Objekt-ID[, Sub-ID])-Paar vorfindet, wird mit `STATUS = "This objID is already in use"` deaktiviert. Aktuell nur für die unten als „durchgesetzt“ markierten Bausteine verdrahtet.

Kriterium: eine Instanz PRO OBJEKT-ID (aktuell durchgesetzt)

Ziel ist eine variable VT-Objekt-ID (`u16ObjId`, bei INIT per `var_u16ObjId` gesetzt und gegen eine klassenspezifische Bereichs-/Typprüfung validiert). Zwei Instanzen mit derselben ID würden unabhängig um denselben Objektwert konkurrieren (Beispiel aus der Praxis: zwei `Q_BackgroundColour` auf ObjID 4003).

Baustein	Ziel-ID =
`Q_Attribute`	**Paar** (Objekt, Attribut-ID) - `mTargetObjectId=u16ObjId`, `mTargetSubId=u8IdAttribute`. Zwei Instanzen auf demselben Objekt, aber verschiedenen Attributen, sind KEIN Konflikt - nur dasselbe Paar zweimal ist einer.
`Q_BackgroundColour`	Objekt mit Hintergrundfarbe. Darf dieselbe Objekt-ID mit EINER `Q_BackgroundColourAux`-Instanz teilen (unterschiedliche FB-Klassen), aber NIEMALS zwei `Q_BackgroundColour`-Instanzen auf dieselbe ID.
`Q_BackgroundColourAux`	AuxFunction-Objekt (AUX-VT, **durchgesetzt**). Eindeutig pro Objekt-ID innerhalb der eigenen FB-Klasse (`getFBTypeId()`). Darf dieselbe Objekt-ID mit EINER `Q_BackgroundColour`-Instanz teilen, aber NIEMALS zwei `Q_BackgroundColourAux`-Instanzen auf dieselbe ID.
`Q_ChangeObjectLabel`	beschriftbares Objekt
`Q_ChangePolygonPoint`	Polygon-Objekt
`Q_ChangePolygonScale`	Polygon-Objekt
`Q_ChildLocation`	Kind-Objekt (Child Location)
`Q_ChildPosition`	Kind-Objekt (Child Position)
`Q_EndPoint`	Linien-Objekt
`Q_FillAttributes`	Objekt mit Füllattributen
`Q_FontAttributes`	Objekt mit Schriftattributen
`Q_GraphicsContext`	Graphics-Context-Objekt
`Q_LineAttributes`	Objekt mit Linienattributen
`Q_ListItem`	**Paar** (Listen-Objekt, List-Index) - `mTargetObjectId=u16ObjId`, `mTargetSubId=u8ListIndex`. Zwei Instanzen auf derselben Liste, aber verschiedenen Indizes, sind KEIN Konflikt - nur dasselbe Paar zweimal ist einer. `u8ListIndex` wird bei `INIT` per `With Var="u8ListIndex"` eingelesen — Claim/Check erfolgt wie bei `Q_Attribute` direkt bei `INIT` in `iso_init()`.
`Q_NumericValue`	Numeric-Value-Objekt. **Ausnahme ObjectPointer:** darf dieselbe ID zusammen mit EINER `Q_NumericValueAux`-Instanz haben (unterschiedliche Klasse, kein Konflikt; relevant nur bei ObjectPointer, weil `Q_NumericValueAux` ausschließlich ObjectPointer-Objekte anspricht). Für jedes andere Zielobjekt (kein ObjectPointer) passt nur eine `Q_NumericValue`-Instanz.
`Q_ObjEnableDisable`	de-/aktivierbares Objekt
`Q_ObjHideShow`	Container (3000-3999)
`Q_ObjSelectInput`	eingabe-selektierbares Objekt
`Q_Size`	größenveränderliches Objekt
`Q_StringValue`	String-Value-Objekt

Kriterium: eine Instanz PRO OBJEKT-ID (gleiche Struktur, NOCH NICHT durchgesetzt)

Diese Bausteine haben exakt dasselbe `u16ObjId`/`var_bObjIdValid`-Muster wie oben, waren aber nicht Teil des ersten Rollouts (sie hängen nicht am Masken-Sichtbarkeits-Gate

`VtMaskVisibility_IsObjectVisible()`, das ursprünglich den Anlass gab). Das Duplikat-Risiko ist identisch - **offene Lücke**, sollte im selben Muster nachgezogen werden.

Baustein	Ziel-ID =	Bemerkung
`Q_NumericValueAux`	Objekt (AUX-VT)	eigenes Gate zusätzlich (`VtAuxAssignment`)
`Q_ExecuteMacro`	Macro-Objekt	
`Q_ExecuteExtendedMacro`	Macro-Objekt	
`Q_LockUnlockMask`	Masken-Objekt (`u16MaskId`, bei `INIT` eingelesen)	Ziel ist eine Maske (`u16MaskId`), nicht ein beliebiges Objekt
`Q_Priority`	Alarm-Mask-Objekt	

Kriterium: genau EINE Instanz im ganzen Programm (Singleton)

Diese Bausteine haben KEINE variable Ziel-Objekt-ID - ihr eigentliches Ziel (Maske, Name, Colour-Map-ID, ...) ist ein dynamischer Parameter, der bei JEDEM `REQ` neu gelesen wird (nicht bei INIT fixiert), der Baustein selbst ist als wiederverwendbarer „Dienst“ gedacht. Der Baustein „gehört“ damit konzeptionell einer festen, im Programm nur einmal vorhandenen Sache (dem WorkingSet bzw. dessen einzigem CF). Zwei Instanzen würden unabhängig um denselben Dienst konkurrieren.

Baustein	Eigentliches Ziel (Parameter bei REQ)	Status
`Q_ActiveMask`	`u16NewMaskId` (Ziel-ID = feste WorkingSet-ID 0)	**durchgesetzt**
`Q_SelectActiveWorkingSet`	`pau8Name` (Name des zu wählenden Working Sets)	offene Lücke
`Q_SelectColourMap`	Colour-Map-Objekt-ID	offene Lücke
`Q_SetAudioVolume`	Lautstärke (Working-Set-weite Einstellung)	offene Lücke
`Q_SoftKeyMask`	DataMask-/SoftKeyMask-IDs	offene Lücke
`Q_CtrlAudioSignal`	Signalparameter (Working-Set-weit)	offene Lücke

Kein Kriterium nötig (zustandslos)

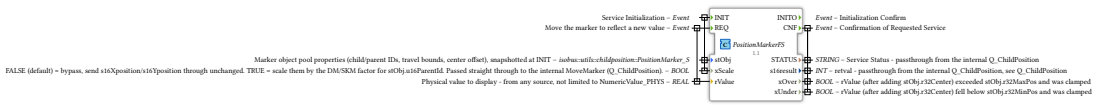
Baustein	Warum unkritisch
`Q_ESC`	Keine Objekt-ID, keine Old-Value-Pufferung überhaupt (`iso_init()` setzt nur `STATUS`) - jeder `REQ` sendet unmittelbar, nichts zum Verwechseln. Mehrere Instanzen sind harmlos.

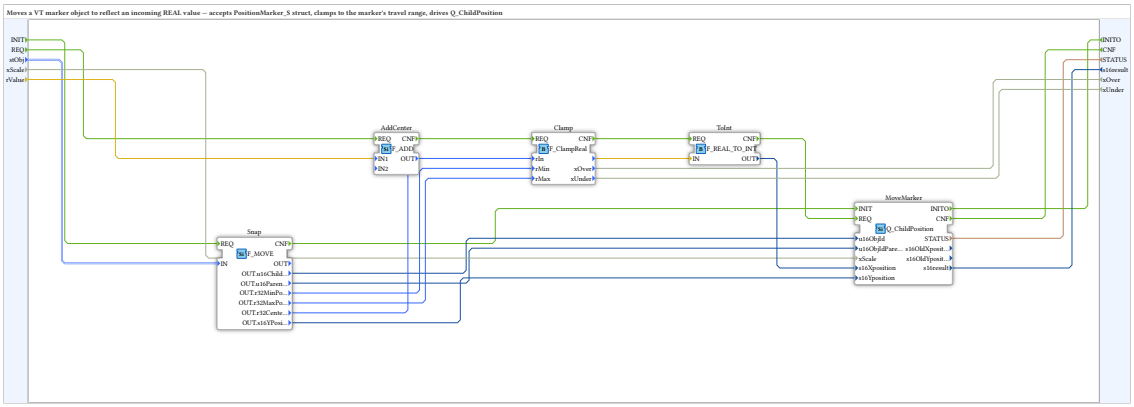
Adapter-Wrapper (`_AUI`, `_AUDI`, `_AB`, `_AX`, `_AR`)

Die Adapter-Varianten (z. B. `Q_NumericValue_AUDI`, `Q_ObjHideShow_AB`) sind Wrapper-Bausteine um die jeweiligen Basisklassen (z. B. `Q_NumericValue`, `Q_ObjHideShow`). Da der Adapter-Wrapper intern auf der Basisklasse aufbaut, **gilt die Instanz-Eindeutigkeit klassenübergreifend**: Eine Objekt-ID kann im gesamten Programm nur von **einer einzigen** Instanz (entweder der Basisklasse oder des Adapter-Wrappers) genutzt werden. Eine gleichzeitige Nutzung derselben Objekt-ID durch Basisklasse und Wrapper ist unzulässig.

`I_GetAttribute` (`isobus_UT/src/isobus/UT/I/`) hat Paar-Eindeutigkeit über (`u16ObjId`, `u8IdAttribute`).

PositionMarkerFS





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **PositionMarkerFS** dient zur Positionierung eines Markierobjekts (z. B. eines Dreiecks) innerhalb eines Virtual-Terminal-Displays gemäß ISO 11783-6. Er akzeptiert einen beliebigen physikalischen REAL-Wert (z. B. von einem Sensor oder einer Eingabe), versetzt diesen um einen vorgegebenen Mittenoffset, begrenzt ihn auf den zulässigen Bewegungsbereich und überträgt die resultierende Integer-Position an eine interne Instanz von `Q_ChildPosition`. Dadurch kann ein Markerobjekt kontinuierlich den aktuellen Messwert visualisieren, ohne dass der Anwender sich um die zugrunde liegende Skalierung oder Begrenzung kümmern muss.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Datentyp	Kommentar
`INIT`	`EInit`	Service Initialization – übernimmt die Objekteigenschaften und das Skalierungs-Flag.
`REQ`	`Event`	Fordert die Bewegung des Markers auf Basis eines neuen Wertes an.

Ereignis-Ausgänge

Ereignis	Datentyp	Kommentar
`INITO`	`EInit`	Initialisierungsbestätigung.
`CNF`	`Event`	Bestätigung des angeforderten Dienstes; liefert Status, Ergebnis und Über-/Unterbereichsflags.

Daten-Eingänge

Name	Datentyp	Kommentar
`stObj`	`isobus::utils::childposition::PositionMarker_S`	Struktur mit Markerobjekt-Eigenschaften (Kind-/Parent-ID, Bereichsgrenzen, Mittenoffset, Y-Position). Wird bei `INIT` einmalig gespeichert.
`xScale`	`BOOL`	Skalierungsflag: `FALSE` (Standard) sendet die Positionen unverändert, `TRUE` skaliert gemäß dem Faktor der Parent-ID. Wird direkt an die interne `Q_ChildPosition` durchgereicht.
`rValue`	`REAL`	Physikalischer Wert, der angezeigt werden soll – aus beliebiger Quelle (z. B. NumericValue_PHYS, Sensor).

Daten-Ausgänge

Name	Datentyp	Kommentar
STATUS	STRING	Dienststatus – direkt von der internen Q_ChildPosition übernommen.
s16result	INT	Rückgabewert der internen Q_ChildPosition (siehe dort).
xOver	BOOL	Wird TRUE, wenn der (um r32Center versetzte) Wert r32MaxPos überschritten und begrenzt wurde.
xUnder	BOOL	Wird TRUE, wenn der (um r32Center versetzte) Wert unter r32MinPos liegt und begrenzt wurde.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der FB arbeitet in zwei Phasen:

1. **Initialisierung (INIT)**: Das übergebene stObj – bestehend aus Kind-/Parent-ID, den Bewegungsgrenzen r32MinPos/r32MaxPos, dem Mittenoffset r32Center und der festen Y-Position – wird über einen internen F_MOVE-Baustein in einem Zwischenspeicher abgelegt. Anschließend wird die interne Q_ChildPosition mit diesen Eigenschaften initialisiert. Das Flag xScale wird unverändert übergeben.
2. **Positionierung (REQ)**: Beim Eintreffen eines neuen REQ wird der Eingangswert rValue zunächst um r32Center addiert (mittels F_ADD). Danach wird der resultierende Wert mithilfe des Bausteins F_ClampReal auf den Bereich [r32MinPos, r32MaxPos] begrenzt. Überschreitet der Wert den oberen Grenzwert, wird xOver gesetzt; unterschreitet er den unteren Grenzwert, wird xUnder gesetzt. Der begrenzte REAL-Wert wird anschließend über F_REAL_TO_INT in einen Integer konvertiert und als s16Xposition an die Q_ChildPosition übergeben. Die zuvor gespeicherte Y-Position (s16YPosition) wird unverändert durchgereicht. Nach Abschluss der Bewegung wird das Ereignis CNF mit den entsprechenden Ausgabedaten ausgelöst.

Technische Besonderheiten

- **Snapshot-Verhalten**: Die Objektstruktur stObj wird nur bei INIT einmalig kopiert. Änderungen an den Parametern nach der Initialisierung haben keinen Einfluss auf die laufende Positionierung.
- **Automatische Bereichsbegrenzung**: Der Eingangswert wird vor der Konvertierung immer auf den gültigen Bereich geclamped. Die Flags xOver/xUnder signalisieren, ob der Ursprungswert außerhalb des Bereichs lag.
- **Durchreichen von Skalierungs- und Statusinformationen**: Das Flag xScale sowie die Ausgaben STATUS und s16result werden direkt von der internen Q_ChildPosition übernommen, ohne weitere Verarbeitung.
- **Keine Y-Berechnung**: Die Y-Position wird nicht dynamisch verändert; sie wird als fester Wert aus der Struktur übernommen.

Zustandsübersicht

Der Funktionsblock besitzt keine expliziten Zustandsautomaten, sondern arbeitet ereignisgesteuert. Der Ablauf lässt sich jedoch wie folgt beschreiben:

- **Idle-Zustand**: Nach erfolgreicher Initialisierung (INIT0) wartet der FB auf ein REQ-Ereignis.
- **Verarbeitungsschritt**: Bei REQ werden nacheinander die Bausteine F_ADD, F_ClampReal, F_REAL_TO_INT und Q_ChildPosition durchlaufen.
- **Bestätigungsphase**: Nach Abschluss der internen Verarbeitung wird CNF mit den aktuellen Werten von STATUS, s16result, xOver und xUnder ausgegeben.

Anwendungsszenarien

- **Anzeige von Sensormesswerten** auf einem ISOBUS-VT, z. B. Füllstand, Temperatur oder Druck, wobei der Wert direkt als REAL ansteht.
- **Steuerung von Markerpositionen** in grafischen Benutzeroberflächen, bei denen der Wertebereich des Messwerts vom Anzeigebereich abweicht.

- **Einbindung in komplexere Visualisierungslogik**, bei der mehrere Markerobjekte verschiedene Werte parallel darstellen, ohne dass für jedes Objekt separate Clamping-Logik erforderlich ist.

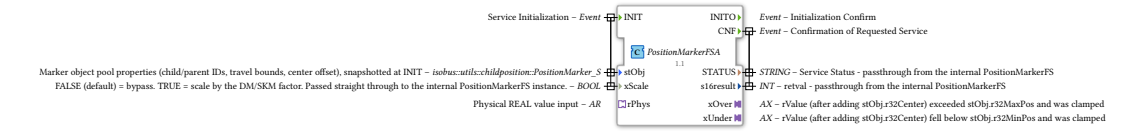
Vergleich mit ähnlichen Bausteinen

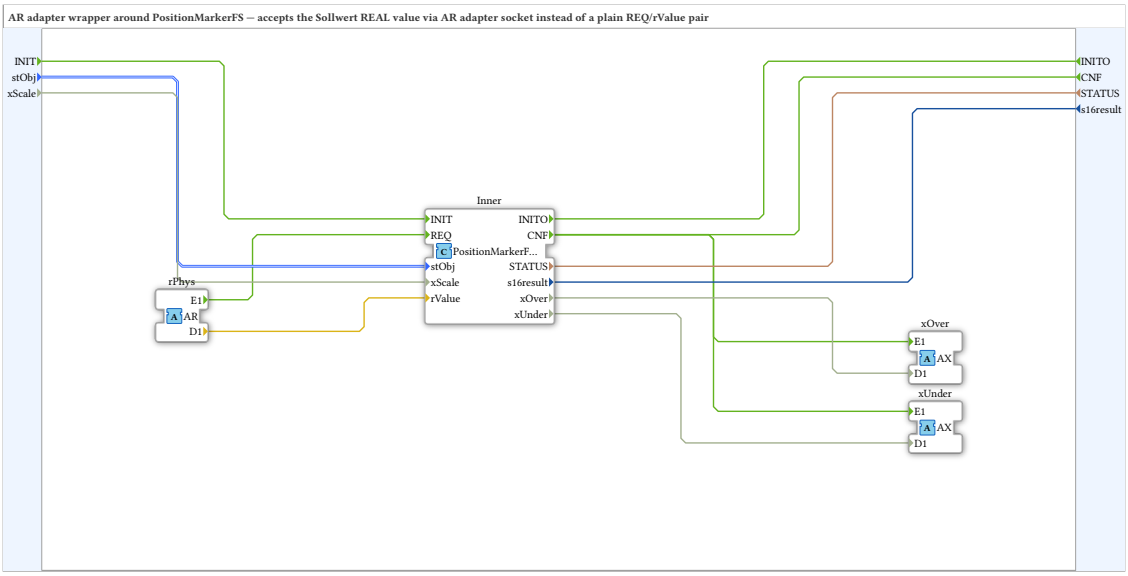
Baustein	Unterschied zu `PositionMarkerFS`
`Q_NumericValue_PHYS`	Stellt eine numerische Anzeige dar, verwendet aber typischerweise eine feste Skalierung und keinen Offset; `PositionMarkerFS` integriert Offset, Clamping und Konvertierung in einem Baustein.
`ScrollFS`	Ähnliches Muster (Snapshot bei INIT), aber fokussiert auf Scroll-Operationen; `PositionMarkerFS` spezialisiert sich auf die Positionierung eines Markers mit Wertebereichsanpassung.
Direkte Verwendung von `Q_ChildPosition`	Erfordert manuelles Umrechnen, Clamping und Integer-Konvertierung durch den Anwender; `PositionMarkerFS` kapselt diese Schritte.

Fazit

Der Funktionsblock `PositionMarkerFS` bietet eine kompakte und robuste Lösung, um einen VT-Marker basierend auf einem beliebigen REAL-Wert präzise zu positionieren. Durch die Kombination von Offsetsetzung, Bereichsbegrenzung und Integer-Konvertierung entfällt für den Anwender die Notwendigkeit, eigene Umrechnungs- und Begrenzungslogik zu implementieren. Die klare Trennung zwischen Initialisierung und Betrieb erleichtert die Integration in bestehende Steuerungsanwendungen und erhöht die Wiederverwendbarkeit.

PositionMarkerFSA





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **PositionMarkerFSA** ist ein Adapter-Wrapper um den Baustein `PositionMarkerFS`. Er ermöglicht die Übergabe eines Sollwertes (REAL) über einen AR-Adapter-Socket, anstatt über ein separates REQ-Ereignis und einen rValue-Eingang. Dadurch wird die Anbindung an ISOBUS-

oder CAN-basierte Datenströme vereinfacht, bei denen physikalische Werte bereits als Adapterwerte vorliegen. Über- und Unterschreitung der zulässigen Positionsgrenzen werden über AX-Adapter-Plugs ausgegeben.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Mit Variablen	Kommentar
`INIT`	`EInit`	`stObj`, `xScale`	Initialisiert den internen Baustein mit den Markerobjekt-Eigenschaften und der Skalierungsoption.

Ereignis-Ausgänge

Ereignis	Typ	Mit Variablen	Kommentar
`INITO`	`EInit`	–	Bestätigt die erfolgreiche Initialisierung.
`CNF`	`Event`	`STATUS`, `s16result`	Wird nach einer Verarbeitung des Sollwertes ausgegeben und liefert Status- und Ergebnisdaten.

Daten-Eingänge

Name	Typ	Kommentar
`stObj`	`isobus::utils::childposition::PositionMarkerID`	Definiert die Markerobjekt-Pooleigenschaften (Kind-/Eltern-IDs, Verfahrgrenzen, Mittenversatz). Wird bei INIT gesnappt.
`xScale`	`BOOL`	Skalierungsflag. `FALSE` (Standard) = Umgehung der Skalierung, `TRUE` = Skalierung durch den DM/SKM-Faktor. Wird direkt an den internen Baustein durchgereicht.

Daten-Ausgänge

Name	Typ	Kommentar
`STATUS`	`STRING`	Dienststatus – direkte Durchreichung vom internen `PositionMarkerFS`.
`s16result`	`INT`	Rückgabewert – direkte Durchreichung vom internen `PositionMarkerFS`.

Adapter

Typ	Richtung	Name	Kommentar
Socket	Eingang	`rPhys`	AR-Adapter, der einen physikalischen REAL-Wert als Sollwert liefert.
Plug	Ausgang	`xOver`	AX-Adapter, aktiv wenn der Sollwert (nach Addition von `stObj.r32Center`) den maximalen Verfahrbereich überschreitet und geklemmt wurde.
Plug	Ausgang	`xUnder`	AX-Adapter, aktiv wenn der Sollwert (nach Addition von `stObj.r32Center`) den minimalen Verfahrbereich unterschreitet und geklemmt wurde.

Funktionsweise

Der Baustein delegiert seine gesamte Funktionalität an eine interne Instanz von `PositionMarkerFS`. Beim Eintreffen des `INIT`-Ereignisses werden die Eingabedaten `stObj` und `xScale` an die interne Instanz übertragen und deren Initialisierung ausgelöst. Nach erfolgreicher Initialisierung quittiert der Baustein mit `INITO`.

Der eigentliche Sollwert wird über den AR-Adapter `rPhys` empfangen. Sobald der Adapter ein Ereignis `E1` liefert (typischerweise bei einer Wertänderung), wird dieses direkt als `REQ`-Ereignis an die interne Instanz weitergeleitet. Gleichzeitig wird der vom Adapter bereitgestellte Datenwert `D1` (der physikalische Sollwert) dem Eingang `rValue` des internen Bausteins zugeführt. Dadurch entfällt die Notwendigkeit, einen separaten `REQ`-Trigger mit einem Datenwert zu koppeln.

Die interne Instanz berechnet die Position unter Berücksichtigung des Mittelpunktsversatzes und der Verfahrgrenzen. Das Ergebnis wird über `STATUS` und `s16result` ausgegeben, nachdem das Ereignis `CNF` der internen Instanz empfangen wurde. Zusätzlich werden die Ereignisse für Über- und Unterschreitung (falls zutreffend) an die AX-Adapter `xOver` bzw. `xUnder` weitergegeben – jeweils ausgelöst durch das `CNF`-Ereignis der internen Instanz.

Technische Besonderheiten

- **Adapter-basiertes Design:** Der Sollwert wird über einen unidirektionalen AR-Adapter zugeführt, was die Integration in bestehende ISOBUS- oder CAN-Datenströme erleichtert.
- **Transparente Durchreichung des Skalierungsflags:** `xScale` wird unverändert an den internen Baustein weitergegeben und steuert dort die optionale Skalierung durch den DM/SKM-Faktor.
- **Rückmeldung über AX-Adapter:** Über- und Unterschreitung werden als separate unidirektionale Adapter-Plugs bereitgestellt, die Ereignisse und Daten getrennt führen.
- **Standardkonformität:** Der Baustein basiert auf dem Standard ISO 11783-6 und verwendet die in `isobus::utils` definierte Struktur `PositionMarker_S`.
- **Wrapper-Architektur:** Die interne Instanz von `PositionMarkerFS` bleibt vollständig gekapselt, sodass alle Schnittstellen über den Adapter-Rahmen laufen.

Zustandsübersicht

Der Baustein besitzt keine eigene, explizit dokumentierte Zustandsmaschine. Ablaufsteuerung ergibt sich aus den Ereignissen:

- **Vor INIT:** Baustein wartet auf Initialisierung. Es werden keine Verarbeitungen durchgeführt.
- **Nach INIT:** Die interne Instanz ist initialisiert. Der Baustein kann Sollwerte vom AR-Adapter verarbeiten und über `CNF` bestätigen.
- **Nach jeder Verarbeitung:** Die Ereignisse `CNF` (und ggf. `xOver.E1`/`xUnder.E1`) werden ausgegeben. Danach ist der Baustein bereit für den nächsten Adapterwert.

Anwendungsszenarien

- **Landwirtschaftliche Maschinensteuerung:** Einsatz in Traktoren oder Anbaugeräten, bei denen Positionsmarker gesetzt werden müssen und der Sollwert als physikalische Größe (z. B. über ein ISOBUS-Terminal) ankommt.
- **Automatische Spurführung:** Übernahme von Sollpositionen aus einem übergeordneten System, wobei eine direkte Kopplung an einen AR-Adapter gewünscht ist.
- **Flexible Wertübergabe:** Überall dort, wo bisher `PositionMarkerFS` mit separatem REQ/rValue verwendet wurde, aber eine Adapter-Schnittstelle die Netzwerk-Topologie vereinfacht.

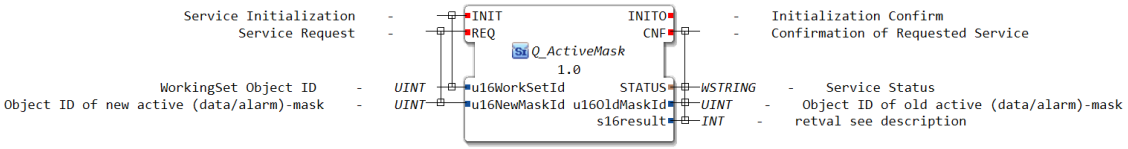
Vergleich mit ähnlichen Bausteinen

Im Gegensatz zum ursprünglichen `PositionMarkerFS`, der die Sollwertübergabe über ein separates `REQ`-Ereignis und den Datenwert `rValue` vornimmt, verwendet `PositionMarkerFSA` einen AR-Adapter–Socket. Dieser bündelt Ereignis (`E1`) und Datenwert (`D1`) in einer einzigen Schnittstelle. Der Funktionsblock `Q_NumericValue_PHYSA` verfolgt ein ähnliches Konzept für numerische Werte; `PositionMarkerFSA` ist speziell auf die Positionsmarker-Logik zugeschnitten und bietet zusätzlich die AX-Adapter für Überschreitungsmeldungen.

Fazit

`PositionMarkerFSA` stellt eine elegante, adapter-basierte Alternative zu `PositionMarkerFS` dar. Er vereinfacht die Anbindung an bestehende Datenströme und bietet eine klare, unidirektionale Kommunikation für Sollwerte und Grenzwertmeldungen. Die vollständige Kapselung der internen Logik macht den Baustein wartbar und konsistent mit anderen ISO 11783-Bausteinen.

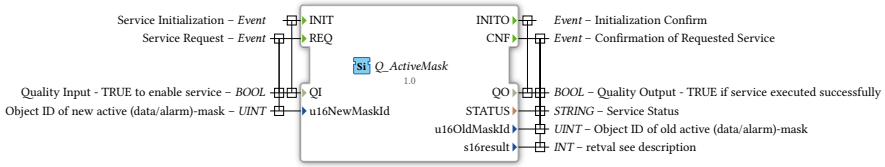
Q_ActiveMask



• — ○ • — ○ • — ○ •

Einleitung

Der **Q_ActiveMask** ist ein standardkonformer Funktionsbaustein zur Steuerung aktiver Masken in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.34) Spezifikation für landwirtschaftliche Steuersysteme.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung
- `REQ`: Maskenwechsel-Anforderung (mit Parametern)

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Maskenwechsel-Bestätigung (mit Ergebnisdaten)

Daten-Eingänge

- `QI` (BOOL, Standardwert `TRUE`): Quality Input – TRUE aktiviert den Dienst (mit `INIT` und `REQ` verknüpft)
- `u16NewMaskId` (UINT): Objekt-ID der neuen Maske

!!! note „WorkingSet Object ID fest auf 0“ Der **WorkingSet Object ID** (`u16WorkSetId`) ist **fest auf 0 gesetzt** durch die Autoren von logiBUS®. Der Benutzer muss sicherstellen, dass das WorkingSet-Objekt im Objektpool immer Object ID 0 hat. Dies ist die Standardeinstellung in den meisten Tools:
`- [ISO-Designer](https://www.bucherautomation.com/iso-designer/sw10133) von Bucher Automation AG`
`- [Isobus Studio](https://isobus-studio.com/) von [to-the-future / Tobias Tenberg] (https://www.to-the-future.de/)`

`Da das Workingset ein einzelnes Objekt ist und nur 1x existiert, stellt diese Einschränkung kein Problem dar.`

Daten-Ausgänge

- `QO` (BOOL): Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde (mit `INITO` und `CNF` verknüpft)
- `STATUS` (STRING): Betriebsstatusmeldung
- `u16OldMaskId` (UINT): Objekt-ID der vorherigen Maske
- `s16result` (INT): ISO-konformer Ergebniscode

Gültige Objekt-IDs

`u16NewMaskId` — gültige Maskenobjekttypen (Annex F.34): Data Mask (1000–1999), Alarm Mask (2000–2999).

ID_NULL (65535) ist kein gültiges Kommandoziel für die neue aktive Maske. Jede ID außerhalb dieser Bereiche ist ungültig.

Funktionsweise

`QI` (Standardwert `TRUE`) wird mit den Ereignissen `INIT` und `REQ` übernommen und aktiviert den Dienst; `QO` wird mit `INITO` und `CNF` ausgegeben und zeigt an, ob der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT`-Event startet den Baustein
   - `INITO` bestätigt erfolgreiches Setup
2. **Maskenwechsel**:
   - `REQ` mit neuen Maskenparametern auslösen
   - `CNF` liefert Ergebnis und vorherige Masken-ID
3. **Fehlerbehandlung**:
   - ISO-standardisierte Fehlercodes
   - Detaillierte Statusmeldungen

Technische Besonderheiten

✓ **ISO 11783-6 konform** (F.34) ✓ **Deterministisches** Verhalten ✓ **Multi-Client-fähige** Architektur ✓ **Echtzeitfähige** Ausführung

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreich
−6	VT_E_OVERFLOW	Pufferüberlauf
−8	VT_E_NOACT	Ungültiger Zustand
−21	VT_E_NO_INSTANCE	Keine VT-Instanz

Anwendungsszenarien

- **Traktorsteuerungen**: Arbeitsmodus-Umschaltung
- **Erntemonitoring**: Datenerfassungsmasken
- **Diagnosesysteme**: Fehleranzeigemasken
- **Multi-Terminal-Betrieb**: Synchronisierte Anzeigen

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	Q_ActiveMask	VtMaskManager	VtDynamicDisplay
ISO-Standard	✓	✖	✖
Fehlercodes	Standard	Hersteller	Teilweise
Zustandsverwaltung	Voll	Basis	Erweitert

🛠️ Zugehörige Übungen

- Uebung_019
- Uebung_019a
- Uebung_019b
- Uebung_019c

Fazit

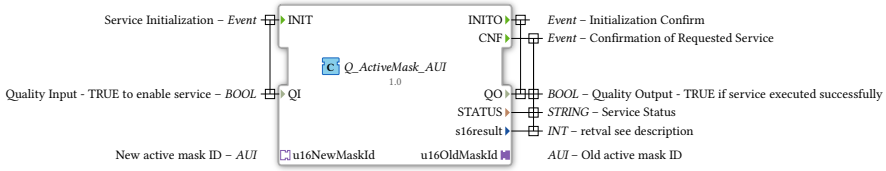
Der Q_ActiveMask-Baustein bietet die Referenzimplementierung für ISOBUS-Maskenwechsel:

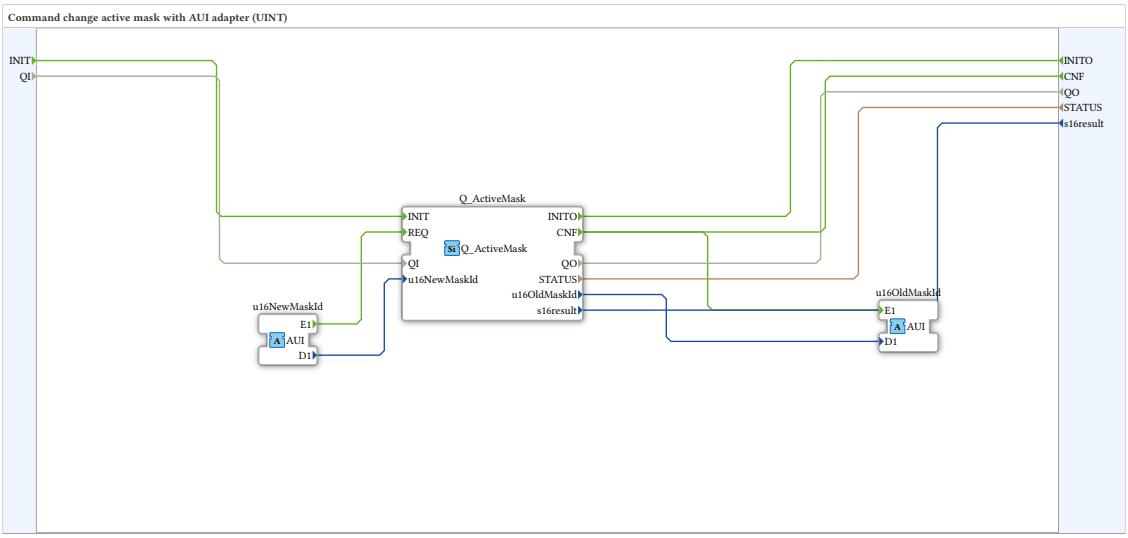
- **Standardkonform**: Volle ISO 11783-6 Kompatibilität
- **Robust**: Bewährte Technik in Serienprodukten
- **Flexibel**: Unterstützt komplexe Anzeigeszenarien

Essentiell für:

- Hersteller von ISOBUS-Terminals
- Entwickler landwirtschaftlicher Steuergeräte
- Systemintegratoren in der Agrartechnik

Q_ActiveMask_AUI





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **Q_ActiveMask_AUI** ist ein AUI-Adapter-Wrapper für den Baustein **Q_ActiveMask**. Er ermöglicht das Umschalten der aktiven Maske (Active Mask) in einem ISOBUS-System (ISO 11783-6) über eine unidirektionale AUI-Schnittstelle. Der Baustein kapselt die Kommunikation mit dem internen `Q_ActiveMask`-Kern und bietet eine einfache, adapterbasierte Anbindung für Ereignis- und Datenaustausch.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
`INIT`	`EInit`	Service-Initialisierung (mit `QI`)

Ereignis-Ausgänge

Name	Typ	Kommentar
`INITO`	`EInit`	Bestätigung der Initialisierung (mit `QO`)
`CNF`	`Event`	Bestätigung des angeforderten Dienstes (mit `QO`, `STATUS`, `s16result`)

Daten-Eingänge

Name	Typ	Kommentar
`QI`	`BOOL`	Quality-Eingang, `TRUE` aktiviert den Dienst (Standard `TRUE`)

Alle übrigen Eingangsdaten werden über die AUI-Adapter-Sockets bereitgestellt.

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Quality-Ausgang, TRUE, wenn der Dienst erfolgreich ausgeführt wurde
STATUS	STRING	Dienststatus
s16result	INT	Rückgabewert (siehe Beschreibung)

Adapter

Typ	Name	Richtung	Kommentar
adapter::types::unidirectional::AUI	u16OldMaskId	Plug (Ausgang)	Alte aktive Masken-ID
adapter::types::unidirectional::AUI	u16NewMaskId	Socket (Eingang)	Neue aktive Masken-ID

Funktionsweise

Der Baustein verbindet intern den eigentlichen Funktionsblock Q_ActiveMask (*isobus::UT::Q::Q_ActiveMask*) mit den Adapter-Schnittstellen.
Beim Eintreffen eines Ereignisses am Socket u16NewMaskId.E1 wird das Ereignis REQ des internen FBs ausgelöst, wodurch der Befehl zum Ändern der aktiven Maske initiiert wird. Der interne FB verarbeitet den Befehl und sendet anschließend über CNF eine Bestätigung zurück, die über den Plug u16OldMaskId.E1 an den verbundenen Empfänger weitergegeben wird.
Die Nutzdaten (Masken-IDs) werden über die Datenkanäle D1 der AUI-Adapter übertragen – u16NewMaskId.D1 liefert die neue ID an den internen FB, während u16OldMaskId.D1 die aktuelle (oder alte) ID ausgibt.
Zusätzlich werden INIT an Q_ActiveMask.INIT und Q_ActiveMask.INITO an INITO geführt; QI wird an Q_ActiveMask.QI und Q_ActiveMask.QO an QO durchgereicht. Die Ergebnis-Ausgänge STATUS und s16result werden direkt an die äußeren Ausgangsvariablen durchgeschliffen.

Technische Besonderheiten

- **Adapter-Wrapper:** Der Baustein besitzt neben dem Quality-Eingang QI keine eigenen Daten-Eingänge; sämtliche Nutzdaten werden ausschließlich über die AUI-Sockets empfangen.
- **Unidirektionale AUI-Kommunikation:** Die verwendeten Adapter sind unidirektional – die Daten fließen nur in eine Richtung, was die Schnittstelle einfach und robust macht.
- **ISOBUS-Konformität:** Der FB ist gemäß ISO 11783-6 (Teil 6 – Anzeige und Bedienung) als Befehl zum Wechseln der aktiven Maske spezifiziert.
- **Keine Zustandsautomaten-Logik im Wrapper:** Der Wrapper selbst enthält keine eigene Zustandsverwaltung, sondern delegiert alle Aktionen an den inneren FB Q_ActiveMask.

Zustandsübersicht

Der Baustein selbst implementiert keinen expliziten Zustandsautomaten. Das Verhalten wird durch den inneren FB Q_ActiveMask bestimmt.
Hinsichtlich der Ereignis-Steuerung lässt sich folgender Ablauf beschreiben:

1. **Initialisierung** – Nach dem Ereignis INIT wird der interne FB initialisiert; die erfolgreiche Initialisierung wird über INITO gemeldet.
2. **Befehlsausführung** – Nach Erhalt eines Ereignisses am Socket u16NewMaskId.E1 wird der Befehl zur Maskenänderung ausgelöst.
3. **Bestätigung** – Sobald die Ausführung abgeschlossen ist, wird das Ereignis CNF ausgegeben und die Ergebnisdaten (STATUS, s16result) stehen zur Verfügung.

Anwendungsszenarien

- Einsatz in ISOBUS-Terminals oder Steuergeräten, um die aktuell angezeigte Maske (Display-Seite) dynamisch zu wechseln.
- Einbindung in größere Applikationen, bei denen Maskenwechsel über standardisierte AUI-Schnittstellen erfolgen sollen, z. B. bei der Umsetzung von Bedienabläufen nach ISO 11783-6.
- Als Adapter-Wrapper für Systeme, die eine einheitliche, unidirektionale Kommunikation für Maskensteuerung verwenden.

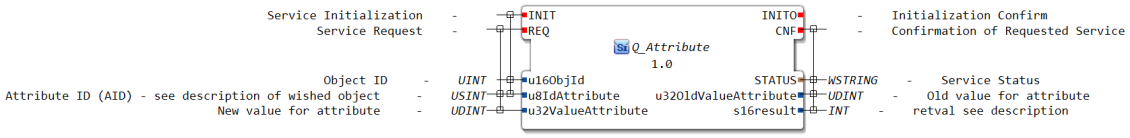
Vergleich mit ähnlichen Bausteinen

Im Gegensatz zum direkten Funktionsblock `Q_ActiveMask` bietet **Q_ActiveMask_AUI** eine zusätzliche Abstraktionsschicht über AUI-Adapter. Dadurch wird die Integration in adapterbasierte Architekturen erleichtert, während die eigentliche Befehlslogik unverändert bleibt. Der direkte FB erwartet separate Ereignis- und Datenleitungen; der Wrapper bündelt diese in einer unidirektionalen Adapter-Schnittstelle und reduziert so die Verdrahtungskomplexität.

Fazit

Der Funktionsblock **Q_ActiveMask_AUI** ist ein kompakter, adapterbasierter Wrapper zur Steuerung der aktiven Maske in ISOBUS-Anwendungen. Er abstrahiert die komplexe Ein-/Ausgabe-Schnittstelle des zugrunde liegenden FB `Q_ActiveMask` und ermöglicht eine einfache, standardisierte Anbindung über unidirektionale AUI-Adapter. Dadurch eignet er sich besonders für modulare Systeme, die eine klare Trennung von Ereignis- und Datenfluss anstreben.

Q_Attribute



• – ○ • – ○ • – ○ •

Einleitung

Der **Q_Attribute** ist ein standardkonformer Funktionsbaustein zur Änderung von VT-Objekteigenschaften, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.38) Spezifikation für landwirtschaftliche Steuersysteme.

Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Objekt-ID und Attribut-ID)
- `REQ`: Attributänderungs-Anforderung (mit neuem Wert)

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Änderungsbestätigung (mit Ergebnisdaten)

Daten-Eingänge

- `QI` (BOOL, Standardwert `TRUE`): Quality Input – TRUE aktiviert den Dienst (mit `INIT` und `REQ` verknüpft)
- `u16ObjId` (UINT): Objektkennung
- `u8IdAttribute` (USINT): Attributkennung (AID)
- `u32ValueAttribute` (UDINT): Neuer Attributwert

Daten-Ausgänge

- `QO` (BOOL): Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde (mit `INITO` und `CNF` verknüpft)
- `STATUS` (STRING): Betriebsstatusmeldung
- `u32OldValueAttribute` (UDINT): Vorheriger Attributwert
- `s16result` (INT): ISO-konformer Ergebniscode

Gültige Objekt-IDs

`u16ObjId` (das Objekt, dessen Attribut geändert wird) und `u8IdAttribute` (die Attribut-ID, AID) unterliegen **unterschiedlichen** Gültigkeitsregeln.

`u16ObjId` — gültige Objekttypen (Annex F.38): DataMask (1000–1999), AlarmMask (2000–2999), SoftKeyMask (4000–4999), Key (5000–5999), Button (6000–6999), Input Boolean Field (7000–7999), Input String Field (8000–8999), Input Number Field (9000–9999), Input List Field (10000–10999), Auxiliary Function Type 2 (31000–31999).

`u8IdAttribute` (AID): Die gültigen Attribut-IDs sind objekttypspezifisch und in Annex B in den Attributtabellen der jeweiligen Objekttypen definiert. Eine AID ohne zugewiesenes Attribut wird vom VT mit „Invalid Attribute ID“ (F.39) beantwortet.

ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei `INIT` den Baustein.

Funktionsweise

`QI` (Standardwert `TRUE`) wird mit den Ereignissen `INIT` und `REQ` übernommen und aktiviert den Dienst; `QO` wird mit `INITO` und `CNF` ausgegeben und zeigt an, ob der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` mit Objekt- und Attribut-ID
   - `INITO` bestätigt Betriebsbereitschaft
2. **Attributänderung**:
   - `REQ` mit neuem Wert auslösen
   - `CNF` liefert Ergebnis und alten Wert
3. **Fehlerbehandlung**:
   - ISO-standardisierte Fehlercodes
   - Detaillierte Statusmeldungen

Technische Besonderheiten

✓ **ISO 11783-6 konform** (F.38) ✓ **Deterministische** Ausführung ✓ **32-bit Wertunterstützung** ✓ **Echtzeitfähige** Verarbeitung

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreich
−6	VT_E_OVERFLOW	Pufferüberlauf
−8	VT_E_NOACT	Ungültiger Zustand
−21	VT_E_NO_INSTANCE	Keine VT-Instanz

Anwendungsszenarien

- **Anzeigeeinstellungen**: Sichtbarkeit/Position von VT-Objekten
- **Maschinenkonfiguration**: Arbeitsparameter anpassen
- **Diagnosesysteme**: Debug-Informationen steuern
- **Multi-Terminal-Betrieb**: Konsistente Objektattribute

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	Q_Attribute	VtPropertySet	VtDynamicConfig
ISO-Standard	✓	✖	✖
Wertbereich	32-bit	16-bit	32-bit
Zustandsverwaltung	Voll	Teilweise	Erweitert

Fazit

Der Q_Attribute-Baustein bietet die Referenzimplementierung für ISOBUS-Attributsteuerung:

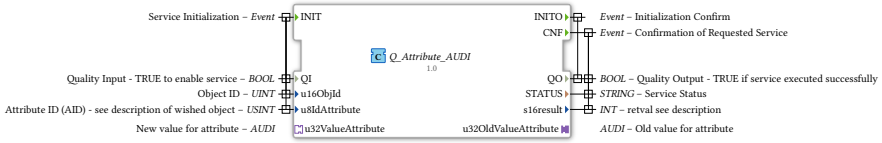
- **Standardkonform**: Volle ISO 11783-6 Kompatibilität
- **Leistungsstark**: 32-bit Wertunterstützung

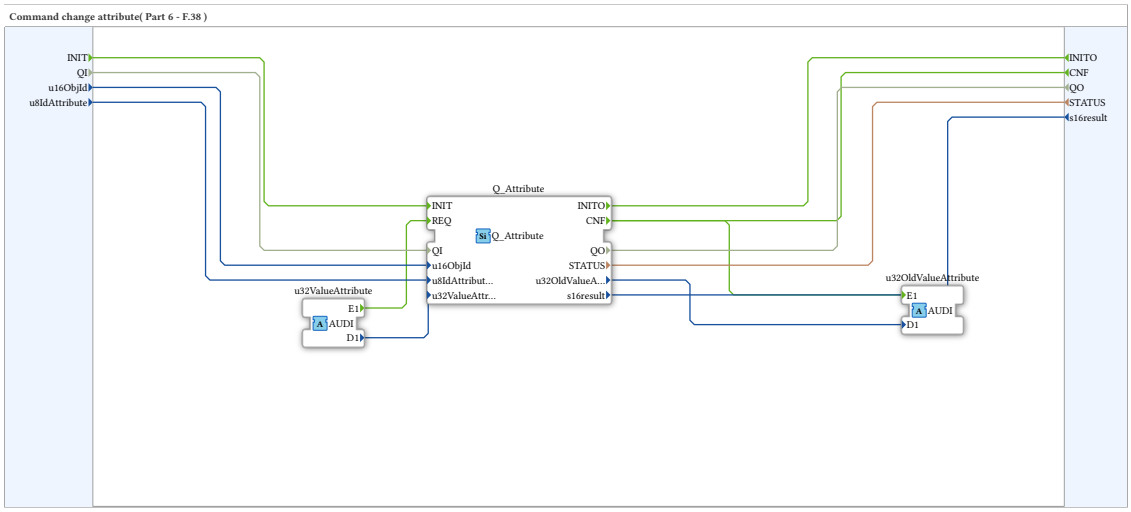
- **Zuverlässig**: Bewährte Technik in Feldgeräten

Essentiell für:

- VT-Objektsteuerungen
- Dynamische Anzeigekonfigurationen
- Maschinenparameter-Adapter

Q_Attribute_AUDI





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **Q_Attribute_AUDI** implementiert das Kommando „Change Attribute“ des ISOBUS-Standards ISO 11783-6 (Teil 6 – F.38). Er ermöglicht das Ändern eines Attributs eines Objekts, das über eine eindeutige Attribut-ID (AID) identifiziert wird. Der Baustein unterstützt die Übergabe des neuen Attributwerts über eine Adapter-Schnittstelle und gibt den alten Wert ebenfalls über einen Adapter zurück. String-Attribute können mit diesem Befehl nicht geändert werden.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung
INIT	EInit	Initialisiert den Service und startet die Kommunikation mit dem ISOBUS-VT

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung
INITO	EInit	Bestätigt den erfolgreichen Abschluss der Initialisierung
CNF	Event	Bestätigt die Durchführung des angeforderten Attributänderungs-Kommandos

Daten-Eingänge

Variable	Typ	Beschreibung
QI	BOOL	Quality Input – TRUE aktiviert den Dienst

u16ObjId	UINT	Objekt-ID des zu ändernden Attributs (Initialwert: ID_NULL)
u8IdAttribute	USINT	Attribut-ID (AID) – definiert, welches Attribut des Objekts geändert werden soll

Daten-Ausgänge

Variable	Typ	Beschreibung
QO	BOOL	Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde
STATUS	STRING	Statusmeldung des ausgeführten Dienstes
s16result	INT	Rückgabewert des Kommandos (Fehlercodes siehe Funktionsweise)

Adapter

Adapter	Richtung	Typ	Beschreibung
u32ValueAttribute	Socket	unidirectional::AUDI	Neuer Wert für das Attribut (32-Bit, wird über den Adapter in den FB eingespeist)
u32OldValueAttribute	Plug	unidirectional::AUDI	Alter Wert des Attributs (32-Bit, wird vom FB über den Adapter ausgegeben)

Qualitätssignale QI/QO: Der Dateneingang `QI` (BOOL, Initialwert `TRUE`) ist dem Ereignis `INIT` zugeordnet und wird an `QI` des internen Bausteins `Q_Attribute` durchgereicht. Der Datenausgang `QO` (BOOL) wird von `Q_Attribute.QO` übernommen und ist den Ereignissen `INITO` und `CNF` zugeordnet.

Gültige Objekt-IDs

`u16ObjId` (das Objekt, dessen Attribut geändert wird) und `u8IdAttribute` (die Attribut-ID, AID) unterliegen **unterschiedlichen** Gültigkeitsregeln.

`u16ObjId` — gültige Objekttypen (Annex F.38): DataMask (1000–1999), AlarmMask (2000–2999), SoftKeyMask (4000–4999), Key (5000–5999), Button (6000–6999), Input Boolean Field (7000–7999), Input String Field (8000–8999), Input Number Field (9000–9999), Input List Field (10000–10999), Auxiliary Function Type 2 (31000–31999).

`u8IdAttribute` (AID): Die gültigen Attribut-IDs sind objekttypspezifisch und in Annex B in den Attributtabellen der jeweiligen Objekttypen definiert. Eine AID ohne zugewiesenes Attribut wird vom VT mit „Invalid Attribute ID“ (F.39) beantwortet.

ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei `INIT` den Baustein.

Funktionsweise

1. **Initialisierung**: Nach einem positiven Flanke an **INIT** wird der Dienst initialisiert. Die dafür benötigten Parameter (Objekt-ID und Attribut-ID) werden aus den Daten-Eingängen gelesen.
2. **Ausführung des Kommandos**: Sobald am Socket **u32ValueAttribute** der Ereignisausgang **E1** ausgelöst wird (z. B. durch einen angeschlossenen FB, der einen neuen Wert bereitstellt), wird das Attributänderungs-Kommando gestartet. Der neue Attributwert wird vom Adapter übernommen.
3. **Rückmeldung**: Nach Abschluss des Kommandos wird der Ausgang **CNF** aktiviert. Der STATUS-String und der Rückgabewert **s16result** enthalten das Ergebnis. Parallel dazu wird am Plug **u32OldValueAttribute** der zuvor gespeicherte alte Attributwert über den Ereignisausgang **E1** signalisiert und über **D1** bereitgestellt.

Der Baustein delegiert die eigentliche ISOBUS-Kommunikation an den internen Funktionsblock **Q_Attribute**, der die notwendigen Protokollschritte durchführt.

Technische Besonderheiten

- **Fehlercodes** (über s16result ausgegeben):
  - `VT_E_NO_ERR (0)` – kein Fehler
  - `VT_E_OVERFLOW (-6)` – Pufferüberlauf
  - `VT_E_NOACT (-8)` – Kommando im aktuellen Zustand nicht möglich
  - `VT_E_NO_INSTANCE (-21)` – Keine VT-Client-Instanz verfügbar
  - `VT_E_ISO_INSTANCE_INVALID (-129)` – Verbindungsinstanz ungültig

  - `VT_E_NOT_ALIVE (-130)` – Instanz gültig, aber VT nicht aktiv
- **Adapterschnittstellen**: Die Verwendung von unidirektionalen Adaptern (AUDI) ermöglicht eine lose Kopplung zwischen dem Wertgeber und dem Baustein. Der alte und neue Wert können so in getrennten FBs verwaltet werden.
- **Einschränkungen**: Der Befehl unterstützt keine String-Attribute.

Zustandsübersicht

Der FB besitzt keine explizit dargestellte Zustandsmaschine. Die Ablaufsteuerung erfolgt ereignisgesteuert:

- Nach **INIT** wechselt er in einen bereiten Zustand.
- Ein weiteres Ereignis wird erst durch den angeschlossenen Adapter-Socket ausgelöst (E1 von u32ValueAttribute). Nach der Verarbeitung wird **CNF** gesendet.
- Der Baustein ist **nicht zustandsbehaftet** im Sinne einer wiederverwendbaren Sequenz; jeder Request wird einzeln abgearbeitet.

Anwendungsszenarien

- **ISOBUS-VT-Anwendungen**: Ändern von Parametern eines virtuellen Terminals (z. B. Hintergrundfarbe, Beschriftung, Sichtbarkeit) zur Laufzeit.
- **Steuerung landwirtschaftlicher Geräte**: Dynamisches Anpassen von Attributen wie Maschineneinstellungen oder Diagnosedaten.
- **Implementierung von ISO 11783-6 Workouts**: Nachbildung der Kommandostruktur „Change Attribute“ aus dem Standard.

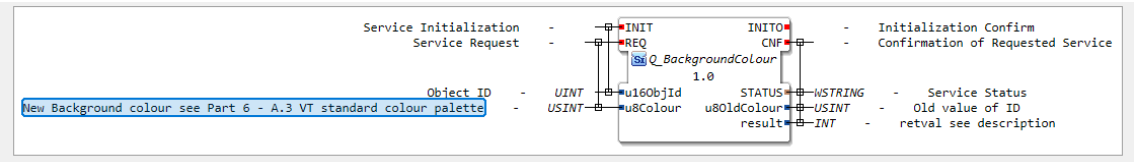
Vergleich mit ähnlichen Bausteinen

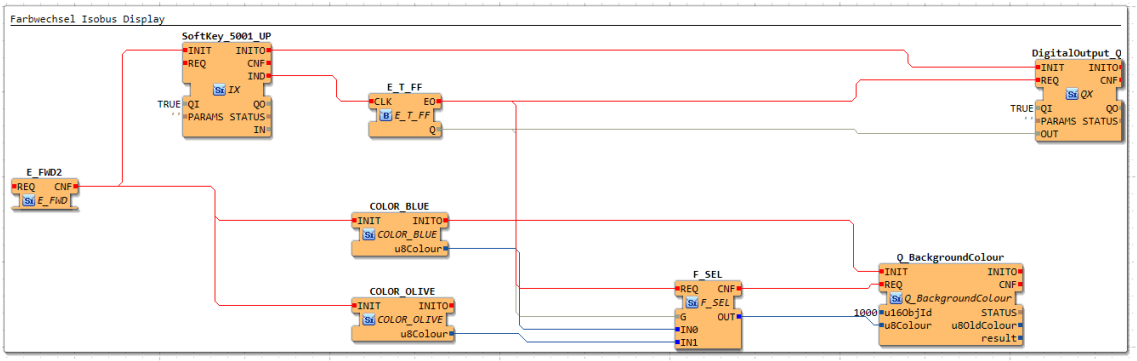
- **Q_Attribute** (ohne AUDI): Bietet die gleiche Kernfunktionalität, jedoch ohne Adapter-Schnittstellen. Der neue Wert muss hier direkt als Dateneingang übergeben werden und es erfolgt keine Rückmeldung des alten Werts. **Q_Attribute_AUDI** erweitert diesen Baustein um eine flexible, adapterbasierte Wertübergabe und die Rückgabe des alten Werts, was die Wiederverwendbarkeit und Kapselung erhöht.

Fazit

Q_Attribute_AUDI ist ein spezialisierter Funktionsblock für ISOBUS-Anwendungen, der das sichere und standardkonforme Ändern von Attributen ermöglicht. Durch die adapterbasierte Schnittstelle für Attributwerte wird eine hohe Modularität erreicht, und die Ausgabe des alten Werts erlaubt eine einfache Nachverfolgung von Änderungen. Der Baustein ist ideal für komplexe VT-Implementierungen, bei denen Werte aus unterschiedlichen Quellen stammen oder in separaten Logikeinheiten verwaltet werden.

Q_BackgroundColour

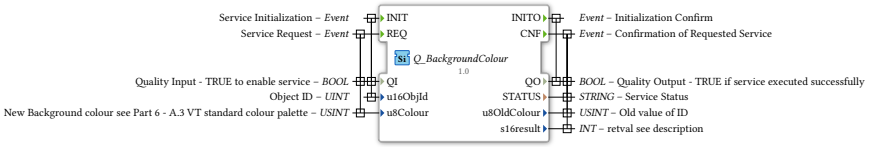




• – ○ • – ○ • – ○ •

Einleitung

Der **Q_BackgroundColour** ist ein standardkonformer Funktionsbaustein zur Steuerung von Hintergrundfarben in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.20) Spezifikation für landwirtschaftliche Steuersysteme.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Objekt-ID)
- `REQ`: Farbänderungs-Anforderung (mit Farbcode)

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Änderungsbestätigung (mit Ergebnisdaten)

Daten-Eingänge

- `QI` (BOOL, Standardwert `TRUE`): Quality Input – TRUE aktiviert den Dienst (mit `INIT` und `REQ` verknüpft)
- `u16ObjId` (UINT): Objektkennung
- `u8Colour` (USINT): Neuer Farbwert (ISO 11783-6 Palette)

Daten-Ausgänge

- `QO` (BOOL): Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde (mit `INITO` und `CNF` verknüpft)
- `STATUS` (STRING): Betriebsstatusmeldung
- `u8OldColour` (USINT): Vorheriger Farbwert
- `s16result` (INT): ISO-konformer Ergebniscode

Gültige Objekt-IDs

`u16ObjId` — gültige Objekttypen (Anhang F.20, Objekte mit Hintergrundfarben-Attribut): WorkingSet (0), Data Mask (1000–1999), Alarm Mask (2000–2999), Soft Key Mask (4000–4999), Key (5000–5999), Button (6000–6999), Input Boolean Field (7000–7999), Input String Field (8000–8999), Input Number Field (9000–9999), Output String Field (11000–11999), Output Number Field (12000–12999), Window Mask (34000–34999), Graphics Context (36000–36999), Auxiliary Function (29000–29999), Auxiliary Function Type2 (31000–31999), Auxiliary Input Type2 (32000–32999).

ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei `INIT` den Baustein.

Funktionsweise

`QI` (Standardwert `TRUE`) wird mit den Ereignissen `INIT` und `REQ` übernommen und aktiviert den Dienst; `QO` wird mit `INITO` und `CNF` ausgegeben und zeigt an, ob der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` mit Objekt-ID
   - `INITO` bestätigt Betriebsbereitschaft
2. **Farbänderung**:
   - `REQ` mit neuem Farbcode (0-255) auslösen
   - `CNF` liefert Ergebnis und alten Farbwert
3. **Fehlerbehandlung**:
   - ISO-standardisierte Fehlercodes
   - Detaillierte Statusmeldungen

Technische Besonderheiten

✓ **ISO 11783-6 konform** (F.20 + A.3 Farbpalette) ✓ **8-bit Farbunterstützung** (256 Werte) ✓ **Echtzeitfähige** Verarbeitung ✓ **Zustandserhaltung** (u8OldColour)

Standardfarbpalette (Auszug)

Code	Farbe	RGB-Äquivalent
0	Schwarz	#000000
1	Rot	#FF0000
2	Grün	#00FF00
3	Gelb	#FFFF00
4	Blau	#0000FF

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreich
−6	VT_E_OVERFLOW	Pufferüberlauf
−8	VT_E_NOACT	Ungültiger Zustand
−128	VT_E_HANDLE_INVALID	Ungültige Objekt-ID

Anwendungsszenarien

- **Maschinenstatus**: Farbliche Alarmanzeige
- **Nachtmodus**: Dunkle Farbpaletten
- **Markierungen**: Hervorhebung von Objekten
- **Branding**: Hersteller-spezifische Farben

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	Q_BackgroundColour	VtColourManager	VtDynamicStyle
ISO-Standard	✓	✖	✖
Farbpalette	Standard	Erweitert	Benutzerdef.
Objektbindung	Explizit	Implizit	Stylesheet

🛠️ Zugehörige Übungen

- Uebung_016
- Uebung_016a

Fazit

Der Q_BackgroundColour-Baustein bietet die Referenzimplementierung für ISOBUS-Farbsteuerung:

- **Standardkonform**: Volle ISO 11783-6 Kompatibilität
- **Einfache Integration**: Klare Farbpalettendefinition
- **Robust**: Bewährte Feldtauglichkeit

Essentiell für:

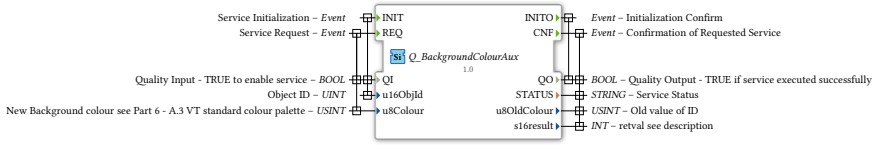
- Farbliche Statusanzeigen
- Benutzeroberflächen-Design
- Dynamische Visualisierungen

Q_BackgroundColourAux

• − ◦ • − ◦ • − ◦ •

Einleitung

Der **Q_BackgroundColourAux** ist ein standardkonformer Funktionsbaustein zur Steuerung von sekundären Hintergrundfarben in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.20) Spezifikation für Hilfsfarben in landwirtschaftlichen Steuersystemen.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Objekt-ID)
- `REQ`: Farbänderungs-Anforderung (mit Farbcode)

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Änderungsbestätigung (mit Ergebnisdaten)

Daten-Eingänge

- `QI` (BOOL, Standardwert `TRUE`): Quality Input – TRUE aktiviert den Dienst (mit `INIT` und `REQ` verknüpft)
- `u16ObjId` (UINT): Objektkennung
- `u8Colour` (USINT): Neuer Hilfsfarbwert (ISO 11783-6 Palette)

Daten-Ausgänge

- `QO` (BOOL): Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde (mit `INITO` und `CNF` verknüpft)
- `STATUS` (STRING): Betriebsstatusmeldung
- `u8OldColour` (USINT): Vorheriger Hilfsfarbwert
- `s16result` (INT): ISO-konformer Ergebniscode

Gültige Objekt-IDs

`u16ObjId` — gültige Objekttypen (Anhang F.20, Objekte mit Hintergrundfarben-Attribut): WorkingSet (0), Data Mask (1000–1999), Alarm Mask (2000–2999), Soft Key Mask (4000–4999), Key (5000–5999), Button (6000–6999), Input Boolean Field (7000–7999), Input String Field (8000–8999), Input Number Field (9000–9999), Output String Field (11000–11999), Output Number Field (12000–12999), Window Mask (34000–34999), Graphics Context (36000–36999), Auxiliary Function (29000–29999), Auxiliary Function Type2 (31000–31999), Auxiliary Input Type2 (32000–32999).

ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei `INIT` den Baustein.

Funktionsweise

`QI` (Standardwert `TRUE`) wird mit den Ereignissen `INIT` und `REQ` übernommen und aktiviert den Dienst; `QO` wird mit `INITO` und `CNF` ausgegeben und zeigt an, ob der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` mit Objekt-ID für Hilfselement
   - `INITO` bestätigt Betriebsbereitschaft
2. **Farbänderung**:
   - `REQ` mit neuem Hilfsfarbcode (0-255) auslösen
   - `CNF` liefert Ergebnis und alten Farbwert
3. **Fehlerbehandlung**:
   - ISO-standardisierte Fehlercodes
   - Detaillierte Statusmeldungen

Technische Besonderheiten

✓ **ISO 11783-6 konform** (F.20 für Hilfsfarben) ✓ **8-bit Farbunterstützung** (256 Werte) ✓ **Kompatibel mit Q_BackgroundColour** ✓ **Zustandserhaltung** (u8OldColour)

Standard-Hilfsfarben (Auszug)

Code	Anwendungsbereich	Typische Farbe
16	Statusbalken	Blau
32	Sekundärbereiche	Grau
48	Hilfslinien	Hellblau

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreich
−128	VT_E_HANDLE_INVALID	Ungültige Objekt-ID
−129	VT_E_ISO_INSTANCE_INVALID	Ungültige Instanz

Anwendungsszenarien

- **Statusanzeigen**: Sekundärfarben für Balken
- **Gruppierungen**: Farbliche Abgrenzung von Bereichen
- **Editor-Modi**: Hilfslinien in Konfiguratoren
- **Diagnose**: Zusatzinformationen

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	Q_BackgroundColourAux	Q_BackgroundColour	VtAuxColour
ISO-Standard	✓	✓	✖
Farbbereich	Hilfselemente	Hauptelemente	Alle
Verwendung	Sekundär	Primär	Universell

Fazit

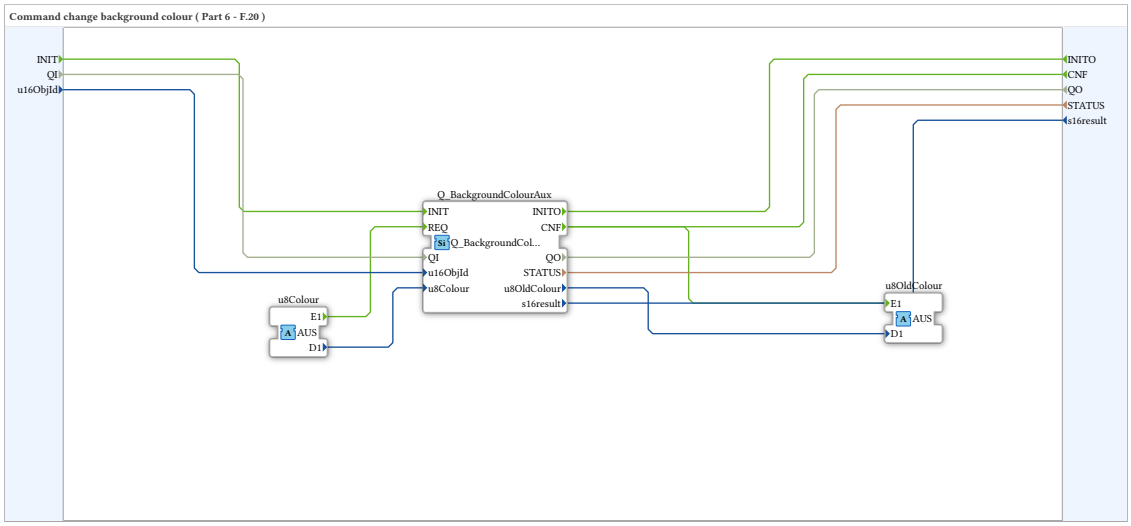
Der Q_BackgroundColourAux-Baustein ergänzt die ISOBUS-Farbsteuerung für Hilfselemente:

- **Spezialisiert**: Optimiert für sekundäre Anzeigeelemente
- **Konsistent**: Gleiche API wie Q_BackgroundColour
- **Praxisbewährt**: Eingesetzt in modernen Traktor-Displays

Idealer Einsatz bei:

- Komplexen Anzeigelayouts
- Mehrschichtigen Visualisierungen
- Systemen mit erweitertem Farbmanagement

Q_BackgroundColourAux_AUS



• – ○ • – ○ • – ○ •

Einleitung

Dieser Funktionsblock implementiert das Kommando zum Ändern der Hintergrundfarbe eines Objekts gemäß ISO 11783-6, Teil F.20.
Er ermöglicht die Interaktion mit dem Virtuellen Terminal (VT) und übernimmt die Übermittlung der neuen Farbe sowie die Rückmeldung des Ergebnisses.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Mit Parameter	Beschreibung
INIT	EInit	QI, u16ObjId	Service Initialisierung

Ereignis-Ausgänge

Ereignis	Typ	Mit Parameter	Beschreibung
INITO	EInit	QO	Initialisierung bestätigt
CNF	Event	QO, STATUS, s16result	Bestätigung des angeforderten Dienstes

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung
QI	BOOL	TRUE	Quality Input – TRUE aktiviert den Dienst
u16ObjId	UINT	ID_NULL	Objekt-ID des betroffenen Objekts

Daten-Ausgänge

Name	Typ	Beschreibung
QO	BOOL	Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde
STATUS	STRING	Dienststatus (z. B. Fehlertext)
s16result	INT	Rückgabewert (siehe unten)

Adapter

Richtung	Name	Typ	Beschreibung
Socket	u8Colour	AUS	Neue Hintergrundfarbe (gemäß ISO 11783-6 A.3)
Plug	u8OldColour	AUS	Alte Hintergrundfarbe (wird vom VT zurückgemeldet)

Qualitätssignale QI/QO: Der Dateneingang `QI` (BOOL, Initialwert `TRUE`) ist dem Ereignis `INIT` zugeordnet und wird an `QI` des internen Bausteins `Q_BackgroundColourAux` durchgereicht. Der Datenausgang `QO` (BOOL) wird von `Q_BackgroundColourAux.QO` übernommen und ist den Ereignissen `INITO` und `CNF` zugeordnet.

Gültige Objekt-IDs

`u16ObjId` ist für dieselben Objekttypen gültig wie beim umschlossenen Basisbaustein `Q_BackgroundColourAux` / `Q_BackgroundColour` (Anhang F.20, Objekte mit Hintergrundfarben-Attribut): WorkingSet (0), Data Mask (1000–1999), Alarm Mask (2000–2999), Soft Key Mask (4000–4999), Key (5000–5999), Button (6000–6999), Input Boolean Field (7000–7999), Input String Field (8000–8999), Input Number Field (9000–9999), Output String Field (11000–11999), Output Number Field (12000–12999), Window Mask (34000–34999), Graphics Context (36000–36999), Auxiliary Function (29000–29999), Auxiliary Function Type2 (31000–31999), Auxiliary Input Type2 (32000–32999).

ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei `INIT` den Baustein.

Funktionsweise

Der Baustein erwartet zuerst eine Initialisierung mit der Objekt-ID (`INIT` mit `u16ObjId`).
Sobald am Socket-Adapter `u8Colour` ein neuer Farbwert anliegt (ausgelöst durch das Ereignis `E1` des Adapters), wird intern ein `REQ`-Event an den verkapselten Funktionsblock `Q_BackgroundColourAux` gesendet. Dieser führt das VT-Kommando aus und liefert über den Ausgang `CNF` den Status (`STATUS`) und das Ergebnis (`s16result`) zurück.
Die alte Farbe wird über den Plug-Adapter `u8OldColour` ausgegeben.

Die gesamte Logik ist in einem untergeordneten FB der gleichen Familie gekapselt, sodass dieser Baustein nur als äußere Hülle für die Adapterkopplung dient.

Technische Besonderheiten

- Der Initialwert für `u16ObjId` ist `ID_NULL`, d. h. der FB muss vor der Nutzung mit einer gültigen Objekt-ID initialisiert werden.
- Die Farbwerte werden als 8-Bit-Werte über den AUS-Adapter transportiert. Die gültigen Farben sind in der VT-Standard-Farbpalette (ISO 11783-6 Anhang A.3) definiert.
- Mögliche Rückgabewerte (`s16result`):
  - `VT_E_NO_ERR` (0): Erfolg
  - `VT_E_OVERFLOW` (-6): Pufferüberlauf
  - `VT_E_NOACT` (-8): Kommando im aktuellen Zustand nicht möglich
  - `VT_E_NO_INSTANCE` (-21): Kein VT-Client verfügbar
  - `VT_E_ISO_INSTANCE_INVALID` (-129): Ungültige Verbindungskennung
  - `VT_E_NOT_ALIVE` (-130): VT-Instanz gültig, aber VT nicht erreichbar

Zustandsübersicht

Der FB besitzt keine expliziten Zustandsautomaten im XML. Die Ereignissteuerung erfolgt rein über die Ereignisverbindungen:

1. **INIT** → Initialisierung des Dienstes.
2. **Externes Ereignis über Socket-Adapter** → Start des Farbänderungs-Requests.
3. **CNF** → Abschluss der Operation mit Status und Rückgabewert.

Anwendungsszenarien

- Dynamische Anpassung der Hintergrundfarbe eines ISOBUS-VT-Objekts (z. B. Button, Textfeld) in Abhängigkeit von Betriebszuständen oder Benutzereingaben.
- Umsetzung von Farbwechseln in landwirtschaftlichen Steuerungs- und Visualisierungssystemen.

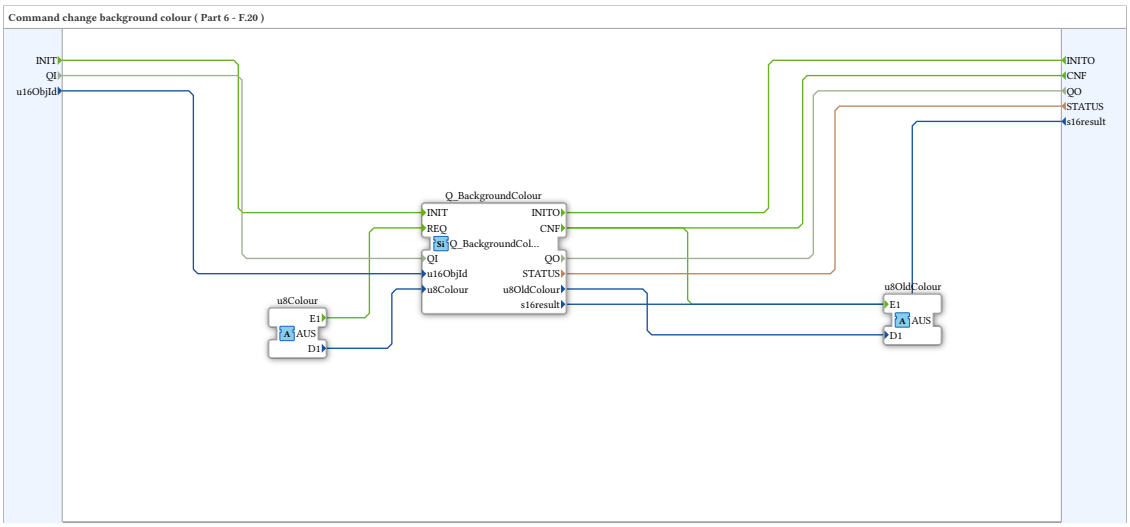
Vergleich mit ähnlichen Bausteinen

Es existieren analoge Funktionsblöcke für andere visuelle Eigenschaften (z. B. `Q_TextColourAux_AUS`, `Q_VisibilityAux_AUS`).
Gemeinsam ist allen das schema-basierte Adapter-Pattern, das die Signalisierung von Wertänderungen über AUS-Adapter realisiert.
Dieser Baustein spezialisiert sich auf die Hintergrundfarbe und verwendet die gleichen Fehler- und Rückgabewerte.

Fazit

Der Funktionsblock `Q_BackgroundColourAux_AUS` kapselt einen ISOBUS-VT-Dienst zur Hintergrundfarbänderung kompakt und bietet durch die Adapterschnittstellen eine einfache Integration in übergeordnete Steuerungen.
Er ist zuverlässig, standardkonform und erlaubt durch die Rückmeldung des Status eine robuste Fehlerbehandlung.

Q_BackgroundColour_AUS



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock **Q_BackgroundColour_AUS** dient zum Ändern der Hintergrundfarbe eines Objekts im ISOBUS Virtual Terminal (VT). Er kapselt die entsprechende Kommando-Schnittstelle gemäß ISO 11783-6 (Part 6 – F.20). Der Baustein wird über einen Socket mit der neuen Farbe versorgt und gibt über einen Plug die vorherige Farbe zurück. Die eigentliche Farbänderung wird durch den internen Baustein `Q_BackgroundColour` ausgeführt.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar	Mit Variablen
`INIT`	EInit	Service Initialisierung	`QI`, `u16ObjId`

Ereignis-Ausgänge

Name	Typ	Kommentar	Mit Variablen

INIT0	EInit	Initialisierungsbestätigung	Q0
CNF	Event	Bestätigung des angeforderten Dienstes	Q0, STATUS, s16result

Daten-Eingänge

Name	Datentyp	Initialwert	Kommentar
QI	BOOL	TRUE	Quality Input – TRUE aktiviert den Dienst
u16ObjId	UINT	ID_NULL	Objekt-ID des zu ändernden Objekts

Daten-Ausgänge

Name	Datentyp	Kommentar
Q0	BOOL	Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde
STATUS	STRING	Dienststatus (z. B. Fehlertext oder „OK“)
s16result	INT	Rückgabewert (siehe Beschreibung – Fehlercodes)

Adapter

- **Socket** u8Colour Typ: adapter::types::unidirectional::AUS Liefert die neue Hintergrundfarbe (gemäß ISO 11783-6, A.3 VT Standard-Farbpalette). Über den Adapter wird sowohl das Ereignis E1 (zur Auslösung des Farbwechsels) als auch der Datenwert D1 (Farbwert) bereitgestellt.
- **Plug** u8OldColour Typ: adapter::types::unidirectional::AUS Gibt die vorherige Hintergrundfarbe zurück, nachdem der Änderungsbefehl ausgeführt wurde. Das Ereignis E1 signalisiert, dass der Wert gültig ist, der Datenwert D1 enthält die alte Farbe.

Qualitätssignale QI/QO: Der Dateneingang QI (BOOL, Initialwert TRUE) ist dem Ereignis INIT zugeordnet und wird an QI des internen Bausteins Q_BackgroundColour durchgereicht. Der Datenausgang Q0 (BOOL) wird von Q_BackgroundColour.Q0 übernommen und ist den Ereignissen INIT0 und CNF zugeordnet.

Gültige Objekt-IDs

u16ObjId ist für dieselben Objekttypen gültig wie beim umschlossenen Basisbaustein Q_BackgroundColour (Anhang F.20, Objekte mit Hintergrundfarben-Attribut): WorkingSet (0), Data Mask (1000–1999), Alarm Mask (2000–2999), Soft Key Mask (4000–4999), Key (5000–5999), Button (6000–6999), Input Boolean Field (7000–7999), Input String Field (8000–8999), Input Number Field (9000–9999), Output String Field (11000–11999), Output Number Field (12000–12999), Window Mask (34000–34999), Graphics Context (36000–36999), Auxiliary Function (29000–29999), Auxiliary Function Type2 (31000–31999), Auxiliary Input Type2 (32000–32999).

ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei INIT den Baustein.

Funktionsweise

1. **Initialisierung** Ein Ereignis INIT startet den Baustein. Die übergebene Objekt-ID (u16ObjId) wird an den internen FB Q_BackgroundColour weitergeleitet und dort initialisiert. Nach erfolgreicher Initialisierung wird INIT0 ausgegeben.
2. **Farbänderung auslösen** Sobald am Socket u8Colour ein Ereignis eintrifft (über den Adapter-Pfad E1), wird der interne Baustein mit REQ angestoßen. Der Farbwert (u8Colour.D1) wird an Q_BackgroundColour.u8Colour übergeben.
3. **Rückmeldung** Nach Abschluss des Kommandos sendet der interne Baustein ein CNF-Ereignis.
   - Die Ausgänge STATUS und s16result werden entsprechend gesetzt und sind am Ereignisausgang CNF gültig.
   - Gleichzeitig wird über den Plug u8OldColour der alte Farbwert (Daten D1) und ein Bestätigungsereignis (E1) ausgegeben.

Technische Besonderheiten

- Der Baustein implementiert das Kommando „Change Background Colour“ nach ISO 11783-6, Abschnitt F.20.
- Die Farbwerte entsprechen der VT-Standard-Farbpalette aus Anhang A.3 der Norm.
- Mögliche Rückgabewerte (`s16result`) sind:
  - `VT_E_NO_ERR (0)` – Erfolg
  - `VT_E_OVERFLOW (-6)` – Pufferüberlauf
  - `VT_E_NOACT (-8)` – Kommando im aktuellen Zustand nicht möglich
  - `VT_E_NO_INSTANCE (-21)` – Kein VT-Client verfügbar
  - `VT_E_ISO_INSTANCE_INVALID (-129)` – Ungültige Verbindungskennung
  - `VT_E_HANDLE_INVALID (-128)` – Ungültiger Handle
  - `VT_E_NOT_ALIVE (-130)` – VT nicht erreichbar
- Der Baustein nutzt einen unidirektionalen Adapter (`AUS`), der Ereignis- und Datenkanal kombiniert – dies reduziert die Anzahl der Schnittstellen.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten auf der obersten Ebene, doch das Zusammenspiel mit `INIT`, `REQ` (via Adapter) und `CNF` ergibt folgende implizite Zustände:

1. **IDLE** – Warten auf `INIT` oder auf ein Ereignis am Socket `u8Colour`.
2. **INIT** – Nach `INIT` läuft die Initialisierung des internen Bausteins.
3. **READY** – Nach `INITO` ist der FB bereit, Farbänderungen entgegenzunehmen.
4. **BUSY** – Während der Ausführung des Farbänderungskommandos (nach `REQ`).
5. **DONE** – Nach `CNF` kann der nächste Befehl verarbeitet werden.

Anwendungsszenarien

- **ISOBUS Virtual Terminal** Ändern der Hintergrundfarbe eines grafischen Objekts (z. B. Button, Gruppe, Softkey) auf dem VT-Bildschirm in einer landwirtschaftlichen Steuerung.
- **HMI-Anpassung** Reaktion auf Benutzereingaben oder Systemzustände, um die optische Darstellung dynamisch anzupassen (z. B. Alarmfarben, Markierung aktiver Elemente).
- **Rücklesen der alten Farbe** Der Plug `u8OldColour` ermöglicht es, den vorherigen Farbwert zu speichern oder für eine spätere Wiederherstellung zu nutzen.

Vergleich mit ähnlichen Bausteinen

- **`Q_BackgroundColour`** (ohne `_AUS`): Bietet die gleiche Kernfunktion, jedoch meist mit einzelnen Ereignis-/Datenanschlüssen statt gekapselter Adapter. Der `_AUS`-Baustein vereinfacht die Verbindung zu anderen Bausteinen, die ebenfalls mit AUS-Adaptern arbeiten.
- **`Q_Colour`** oder **`Q_ForegroundColour`**: Ändern andere Farbeigenschaften (Vordergrund, Vollfarbe) und verwenden ähnliche Rückgabewerte und Statusmechanismen.
- **`Command_ChangeColour`** (allgemeiner): Könnte mehrere Farbparameter gleichzeitig ändern, während `Q_BackgroundColour_AUS` auf Hintergrundfarben spezialisiert ist.

Fazit

Der Funktionsblock `Q_BackgroundColour_AUS` bietet eine standardkonforme, kompakte Schnittstelle zum Ändern von Hintergrundfarben in einem ISOBUS-VT. Durch den Einsatz des unidirektionalen Adapters wird die Anbindung an andere Komponenten erleichtert und die Übersichtlichkeit erhöht. Die dokumentierten Fehlercodes und die saubere Trennung von Initialisierung und Dienstausführung machen ihn zu einem zuverlässigen Baustein für dynamische HMI-Anpassungen in landwirtschaftlichen Steuerungssystemen.

Q_ChangeObjectLabel

• – ○ • – ○ • – ○ •

Einleitung

Der **Q_ChangeObjectLabel** ist ein standardkonformer Funktionsbaustein zur Änderung von Objektbeschriftungen in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.50) Spezifikation für landwirtschaftliche Steuersysteme.

Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Objekt-ID)
- `REQ`: Beschriftungsänderungs-Anforderung (mit Text/Grafik-Parametern)

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Änderungsbestätigung (mit Ergebnisdaten)

Daten-Eingänge

- `QI` (BOOL, Standardwert `TRUE`): Quality Input – TRUE aktiviert den Dienst (mit `INIT` und `REQ` verknüpft)
- `u16ObjId` (UINT): Zielobjekt-Kennung
- `u16ObIdStringVar` (UINT): Textvariablen-Objekt-ID (max. 32 Zeichen)
- `u8FontType` (USINT): Schriftart (bei Textbeschriftung)
- `u16ObIdGrafic` (UINT): Grafikobjekt-ID (für Symbolbeschriftung)

Daten-Ausgänge

- `QO` (BOOL): Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde (mit `INITO` und `CNF` verknüpft)
- `STATUS` (STRING): Betriebsstatusmeldung
- `u16OldObIdStringVar` (UINT): Vorherige Textvariablen-ID
- `u8OldFontType` (USINT): Vorherige Schriftart
- `u16OldObIdGrafic` (UINT): Vorherige Grafikobjekt-ID
- `s16result` (INT): ISO-konformer Ergebniscode

Gültige Objekt-IDs

Der Befehl F.50 adressiert in Bytes 2,3 die **Objekt-ID des zu beschriftenden Objekts** — laut Annex F.51 (Fehler-Bit 0) und B.21 muss dieses Objekt in der **Object Label Reference List** gelistet sein (d. h. ein beliebiges Pool-Objekt, dessen ID ein Eintrag der Referenzliste ist).

Implementierungshinweis (VTClientHelper): `iso_is_object_label_refer_list_id` akzeptiert nur den **ObjectLabelReferList**-Bereich (die ID des Referenzlisten-Objekts selbst) — eine bekannte Abweichung von der Annex-F.50-Semantik:

- **ObjectLabelReferList**: 40000 – 40999

ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei `INIT` den Baustein.

Funktionsweise

`QI` (Standardwert `TRUE`) wird mit den Ereignissen `INIT` und `REQ` übernommen und aktiviert den Dienst; `QO` wird mit `INITO` und `CNF` ausgegeben und zeigt an, ob der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` mit Zielobjekt-ID
   - `INITO` bestätigt Betriebsbereitschaft
2. **Beschriftungsänderung**:
   - `REQ` mit neuen Beschriftungsparametern
   - Unterstützt Text (mit Schriftart) oder Grafiksymbole
   - `CNF` liefert Ergebnis und alte Werte
3. **Fehlerbehandlung**:

- ISO-standardisierte Fehlercodes
- Detaillierte Statusmeldungen

Technische Besonderheiten

✓ **ISO 11783-6 konform** (F.50) ✓ **Flexible Beschriftung**: Text oder Grafik ✓ **Schriftart-Steuerung** (8-bit) ✓ **32-Zeichen Limit** für Textbeschriftungen

Standardparameter

Parameter	NULL-Wert	Beschreibung
u16ObIdStringVar	0xFFFF	Kein Text
u16ObIdGrafic	0xFFFF	Keine Grafik

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreich
−6	VT_E_OVERFLOW	Pufferüberlauf (Text zu lang)
−128	VT_E_HANDLE_INVALID	Ungültige Objekt-ID

Anwendungsszenarien

- **Maschinensteuerung**: Dynamische Schaltflächenbeschriftung
- **Sprachumschaltung**: Mehrsprachige Oberflächen
- **Statusanzeigen**: Symbolwechsel bei Alarmen
- **Personalisierung**: Benutzerdefinierte Labels

⚖️ **Vergleich mit ähnlichen Bausteinen**

Feature	Q_ChangeObjectLabel	VtDynamicLabel	VtTextManager
ISO-Standard	✓	✖	✖
Beschriftung	Text + Grafik	Nur Text	Nur Text
Schriftart	Steuerbar	Fest	Teilweise

Fazit

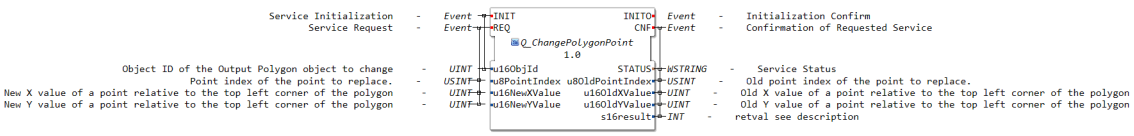
Der Q_ChangeObjectLabel-Baustein bietet die Referenzimplementierung für ISOBUS-Beschriftungen:

- **Vielseitig**: Kombination von Text und Grafiken
- **Standardkonform**: Volle ISO 11783-6 Kompatibilität
- **Praxisbewährt**: Eingesetzt in modernen Agrarmaschinen

Unverzichtbar für:

- Adaptive Benutzeroberflächen
- Mehrsprachige Systeme
- Symbolgesteuerte Bedienkonzepte

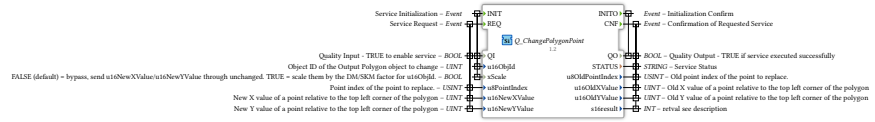
Q_ChangePolygonPoint



• — ○ • — ○ • — ○ •

Einleitung

Der **Q_ChangePolygonPoint** ist ein standardkonformer Funktionsbaustein zur Modifikation von Polygonpunkten in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.52) Spezifikation für landwirtschaftliche Steuersysteme.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Polygon-Objekt-ID und xScale)
- `REQ`: Punktänderungs-Anforderung (mit Index und Koordinaten)

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Änderungsbestätigung (mit Ergebnisdaten)

Daten-Eingänge

- `QI` (BOOL, Standardwert `TRUE`): Quality Input – TRUE aktiviert den Dienst (mit `INIT` und `REQ` verknüpft)
- `u16ObjId` (UINT): Polygon-Objekt-ID
- `xScale` (BOOL): Bei TRUE werden die Werte `u16NewXValue`/`u16NewYValue` vor dem Senden mit dem DM/SKM-Faktor für `u16ObjId` skaliert; bei FALSE (Standard) werden sie unverändert durchgereicht.
- `u8PointIndex` (USINT): Punktindex (0-basiert)
- `u16NewXValue` (UINT): Neue X-Koordinate (relativ)
- `u16NewYValue` (UINT): Neue Y-Koordinate (relativ)

Daten-Ausgänge

- `QO` (BOOL): Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde (mit `INITO` und `CNF` verknüpft)
- `STATUS` (STRING): Betriebsstatusmeldung
- `u8OldPointIndex` (USINT): Vorheriger Punktindex
- `u16OldXValue` (UINT): Vorherige X-Koordinate
- `u16OldYValue` (UINT): Vorherige Y-Koordinate
- `s16result` (INT): ISO-konformer Ergebniscode

Funktionsweise

`QI` (Standardwert `TRUE`) wird mit den Ereignissen `INIT` und `REQ` übernommen und aktiviert den Dienst; `QO` wird mit `INITO` und `CNF` ausgegeben und zeigt an, ob der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` mit Polygon-Objekt-ID
   - `INITO` bestätigt Betriebsbereitschaft
2. **Punktänderung**:
   - `REQ` mit Index und neuen Koordinaten
   - Koordinaten relativ zur linken oberen Ecke
   - `CNF` liefert Ergebnis und alte Werte
3. **Fehlerbehandlung**:
   - ISO-standardisierte Fehlercodes
   - Detaillierte Statusmeldungen

Technische Besonderheiten

✓ **ISO 11783-6 konform** (F.52) ✓ **Präzise Koordinatensteuerung** (16-bit) ✓ **Indexbasierter Zugriff** (0-255 Punkte) ✓ **Ein Punkt pro Befehl** (feste 8-Byte-Nachricht, kein Transport Protocol) ✓ **Echtzeitfähige** Verarbeitung

Koordinatenbereich

Parameter	Bereich	Beschreibung
X-Wert	0 - 65535	Horizontale Position (px)
Y-Wert	0 - 65535	Vertikale Position (px)

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreich
−6	VT_E_OVERFLOW	Ungültiger Punktindex
−128	VT_E_HANDLE_INVALID	Ungültige Polygon-ID

Anwendungsszenarien

- **Maschinenvisualisierung**: Anpassung von Konturen
- **Kartendarstellung**: Dynamische Polygonverformung
- **Diagnoseanzeigen**: Geometrische Hervorhebungen
- **Animierte Elemente**: Bewegte Polygonformen

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	Q_ChangePolygonPoint	VtGeometryEditor	VtDynamicShape
ISO-Standard	✓	✖	✖
Genauigkeit	16-bit Koordinaten	8-bit	16-bit
Punktanzahl	Bis zu 255	Unbegrenzt	Begrenzt

Fazit

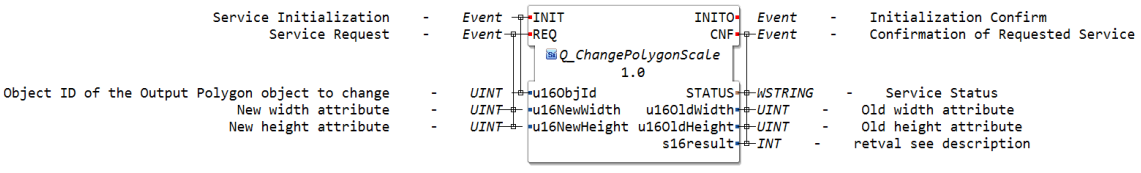
Der Q_ChangePolygonPoint-Baustein bietet die Referenzimplementierung für dynamische Polygonveränderungen:

- **Präzise**: Millimeter-genaue Positionskontrolle
- **Leistungsstark**: Echtzeitfähige Verarbeitung
- **Standardkonform**: Volle ISO 11783-6 Kompatibilität

Essentiell für:

- Adaptive Geometriedarstellungen
- Technische Zeichnungen
- Interaktive Kartenanwendungen

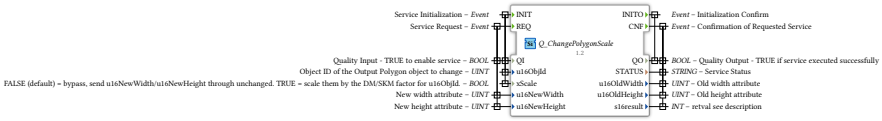
Q_ChangePolygonScale



• — ○ • — ○ • — ○ •

Einleitung

Der **Q_ChangePolygonScale** ist ein standardkonformer Funktionsbaustein zur Skalierung von Polygonobjekten in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.54) Spezifikation für landwirtschaftliche Steuersysteme.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Polygon-Objekt-ID und `xScale`)
- `REQ`: Skalierungsanforderung (mit neuen Abmessungen)

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Skalierungsbestätigung (mit Ergebnisdaten)

Daten-Eingänge

- `QI` (BOOL, Standardwert `TRUE`): Quality Input – TRUE aktiviert den Dienst (mit `INIT` und `REQ` verknüpft)
- `u16ObjId` (UINT): Polygon-Objekt-ID
- `xScale` (BOOL, Initialwert `FALSE`): `FALSE` = Bypass, `u16NewWidth`/`u16NewHeight` werden unverändert gesendet; `TRUE` = beide werden mit dem DM/SKM-Faktor für `u16ObjId` skaliert
- `u16NewWidth` (UINT): Neue Breite in Pixeln
- `u16NewHeight` (UINT): Neue Höhe in Pixeln

Daten-Ausgänge

- `QO` (BOOL): Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde (mit `INITO` und `CNF` verknüpft)
- `STATUS` (STRING): Betriebsstatusmeldung
- `u16OldWidth` (UINT): Vorherige Breite
- `u16OldHeight` (UINT): Vorherige Höhe
- `s16result` (INT): ISO-konformer Ergebniscode

Funktionsweise

`QI` (Standardwert `TRUE`) wird mit den Ereignissen `INIT` und `REQ` übernommen und aktiviert den Dienst; `QO` wird mit `INITO` und `CNF` ausgegeben und zeigt an, ob der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` mit Polygon-Objekt-ID
   - `INITO` bestätigt Betriebsbereitschaft
2. **Skalierung**:
   - `REQ` mit neuen Abmessungen auslösen
   - Proportionale Skalierung aller Polygonpunkte
   - `CNF` liefert Ergebnis und alte Werte
3. **Fehlerbehandlung**:
   - ISO-standardisierte Fehlercodes
   - Detaillierte Statusmeldungen

Technische Besonderheiten

✓ **ISO 11783-6 konform** (F.54) ✓ **Proportionale Skalierung** (Breite/Höhe) ✓ **16-bit Auflösung** (0-65535 Pixel) ✓ **Echtzeitfähige** Verarbeitung

Skalierungsverhalten

Parameter	Wirkung
Breite	Horizontale Skalierung
Höhe	Vertikale Skalierung

Die VT skaliert gemäß Annex F.54 alle Polygonpunkte mit 32-Bit-Integer-Arithmetik: `new_x = [(old_x × new_width) + (old_width/2)] / old_width` und analog für `new_y`.

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreich
−6	VT_E_OVERFLOW	Ungültige Abmessungen
−128	VT_E_HANDLE_INVALID	Ungültige Polygon-ID
−129	VT_E_ISO_INSTANCE_INVALID	Ungültige Instanz

Anwendungsszenarien

- **Maschinendarstellung**: Dynamische Größenanpassung
- **Zoomfunktionen**: Detailvergrößerungen
- **Responsive Designs**: Anpassung an Displaygrößen
- **Animierte Übergänge**: Fließende Größenänderungen

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	Q_ChangePolygonScale	VtGeometryScaler	VtDynamicResize
ISO-Standard	✓	✖	✖
Skalierung	Proportional	Frei	Nur Breite
Auflösung	16-bit	8-bit	16-bit

Fazit

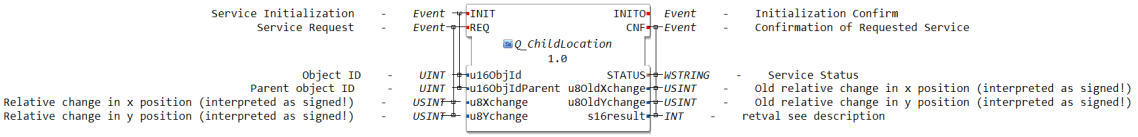
Der Q_ChangePolygonScale-Baustein bietet die Referenzimplementierung für Polygon-Skalierungen:

- **Präzise**: Pixelgenaue Größenkontrolle
- **Proportional**: Erhalt der Originalform
- **Standardkonform**: Volle ISO 11783-6 Kompatibilität

Idealer Einsatz bei:

- Adaptiven Visualisierungen
- Dynamischen Kartenansichten
- Größenveränderbaren Bedienelementen

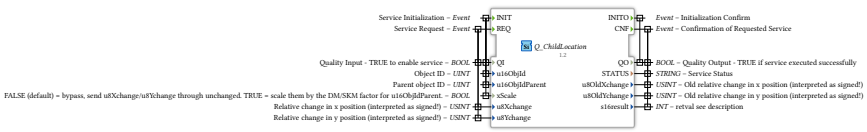
Q_ChildLocation



• — ○ • — ○ • — ○ •

Einleitung

Der **Q_ChildLocation** ist ein standardkonformer Funktionsbaustein zur Positionsänderung von Child-Objekten in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.14) Spezifikation für landwirtschaftliche Steuersysteme.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Objekt- und Parent-ID sowie `xScale`)
- `REQ`: Positionsänderungs-Anforderung (mit relativen Koordinaten)

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Änderungsbestätigung (mit Ergebnisdaten)

Daten-Eingänge

- `QI` (BOOL, Standardwert `TRUE`): Quality Input – TRUE aktiviert den Dienst (mit `INIT` und `REQ` verknüpft)
- `u16ObjId` (UINT): Child-Objekt-ID
- `u16ObjIdParent` (UINT): Parent-Objekt-ID
- `xScale` (BOOL, Standard `FALSE`): `FALSE` = Bypass, `u8Xchange`/`u8Ychange` werden unverändert gesendet; `TRUE` = Skalierung mit dem DM/SKM-Faktor für `u16ObjIdParent`
- `u8Xchange` (USINT): Relative X-Positionsänderung (-128 bis +127)
- `u8Ychange` (USINT): Relative Y-Positionsänderung (-128 bis +127)

Daten-Ausgänge

- `QO` (BOOL): Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde (mit `INITO` und `CNF` verknüpft)
- `STATUS` (STRING): Betriebsstatusmeldung
- `u8OldXchange` (USINT): Vorherige X-Änderung
- `u8OldYchange` (USINT): Vorherige Y-Änderung
- `s16result` (INT): ISO-konformer Ergebniscode

Gültige Objekt-IDs

`u16ObjId` (das zu bewegende Child-Objekt) und `u16ObjIdParent` (das Parent-Objekt) unterliegen **unterschiedlichen** Gültigkeitsregeln. Annex F der ISO 11783-6 definiert nur, welche Objekttypen als Child gültig sind — welche Objekttypen als Parent zulässig sind, steht separat in Annex B, jeweils unter „Allowed commands“ des Objekttyps.

`u16ObjId` — gültige Child-Objekttypen (Annex F.14): Container (3000–3999), Key (5000–5999), Button (6000–6999), Input Boolean Field (7000–7999), Input String Field (8000–8999), Input Number Field (9000–9999), Input List Field (10000–10999), Output String Field (11000–11999), Output Number Field (12000–12999), Output Line (13000–13999), Output Rectangle (14000–14999), Output Ellipse (15000–15999), Output Polygon (16000–16999), Meter (17000–17999), Linear Bar Graph (18000–18999), Arched Bar Graph (19000–19999), Picture Graphic (20000–20499), Object Pointer (27000–27999), Output List Object (37000–37999), External Object Pointer (43000–43999), Animation Object (44000–44999), Scaled Graphic Object (48000–48999), Auxiliary Function/Type2 (29000–29999 / 31000–31999), Auxiliary Input/Type2 (30000–30999 / 32000–32999), Auxiliary Control Designator (33000–33999).

`u16ObjIdParent` — gültige Parent-Objekttypen (Annex B, „Allowed commands“): WorkingSet (B.1), Data Mask (B.2), Alarm Mask (B.3), **Container (B.4)**, Key (B.6), Window Mask, Button, Auxiliary Function (Type 1/2), Auxiliary Input (Type 1/2).

Nicht zulässig als Parent: Soft Key Mask (B.5 listet das Kommando nicht als erlaubt) und Animation Object (dessen Kinder sind indexbasiert, nicht X/Y-positioniert).

ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei `INIT` den Baustein.

Funktionsweise

`QI` (Standardwert `TRUE`) wird mit den Ereignissen `INIT` und `REQ` übernommen und aktiviert den Dienst; `QO` wird mit `INITO` und `CNF` ausgegeben und zeigt an, ob der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` mit Child- und Parent-Objekt-ID
   - `INITO` bestätigt Betriebsbereitschaft
2. **Positionsänderung**:
   - `REQ` mit relativen Koordinatenänderungen
   - Werte werden als signed 8-bit interpretiert

- `CNF` liefert Ergebnis und alte Werte

3. **Fehlerbehandlung**:
   - ISO-standardisierte Fehlercodes
   - Detaillierte Statusmeldungen

Technische Besonderheiten

✓ **ISO 11783-6 konform** (F.14) ✓ **Relative Positionsänderung** (relativ zum Parent) ✓ **Signed 8-bit Werte** (-128 bis +127) ✓ **Echtzeitfähige** Verarbeitung

Positionsänderungsbereich

Parameter	Bereich	Beschreibung
X-Änderung	−128 - +127	Horizontale Verschiebung
Y-Änderung	−128 - +127	Vertikale Verschiebung

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreich
−6	VT_E_OVERFLOW	Ungültige Positionswerte
−128	VT_E_HANDLE_INVALID	Ungültige Objekt-ID

Anwendungsszenarien

- **Dynamische Layouts**: Objektpositionierung
- **Animierte Elemente**: Sanfte Bewegungen
- **Responsive Designs**: Anpassung an verschiedene Bildschirmgrößen
- **Gruppierte Objekte**: Gemeinsame Bewegungen

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	Q_ChildLocation	VtAbsolutePosition	VtGroupMove
ISO-Standard	✓	✖	✖
Bewegung	Relativ	Absolut	Gruppen
Wertebereich	±127 Pixel	0-65535 Pixel	±127 Pixel

Fazit

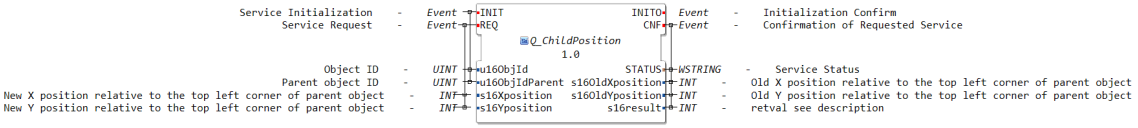
Der Q_ChildLocation-Baustein bietet die Referenzimplementierung für relative Positionsänderungen:

- **Flexibel**: Einfache relative Positionierung
- **Präzise**: Pixelgenaue Steuerung
- **Standardkonform**: Volle ISO 11783-6 Kompatibilität

Idealer Einsatz bei:

- Dynamischen Oberflächen
- Animierten Übergängen
- Parent-Child-Objektbeziehungen

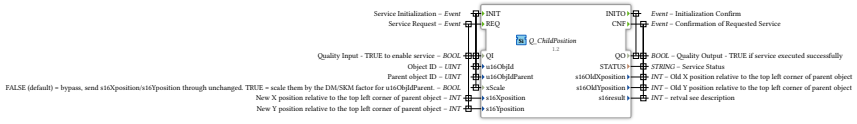
Q_ChildPosition



• — ○ • — ○ • — ○ •

Einleitung

Der **Q_ChildPosition** ist ein standardkonformer Funktionsbaustein zur absoluten Positionsänderung von Child-Objekten in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.16) Spezifikation für landwirtschaftliche Steuersysteme.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Objekt-ID, Parent-ID und xScale)
- `REQ`: Positionsänderungs-Anforderung (mit absoluten Koordinaten)

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Änderungsbestätigung (mit Ergebnisdaten)

Daten-Eingänge

- `QI` (BOOL, Standardwert `TRUE`): Quality Input – TRUE aktiviert den Dienst (mit `INIT` und `REQ` verknüpft)
- `u16ObjId` (UINT): Child-Objekt-ID
- `u16ObjIdParent` (UINT): Parent-Objekt-ID
- `xScale` (BOOL): Bei TRUE werden die Werte `s16Xposition`/`s16Yposition` vor dem Senden mit dem DM/SKM-Faktor für `u16ObjIdParent` skaliert; bei FALSE (Standard) werden sie unverändert durchgereicht.
- `s16Xposition` (INT): Absolute X-Position relativ zum Parent
- `s16Yposition` (INT): Absolute Y-Position relativ zum Parent

Daten-Ausgänge

- `QO` (BOOL): Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde (mit `INITO` und `CNF` verknüpft)
- `STATUS` (STRING): Betriebsstatusmeldung
- `s16OldXposition` (INT): Vorherige X-Position
- `s16OldYposition` (INT): Vorherige Y-Position
- `s16result` (INT): ISO-konformer Ergebniscode

Gültige Objekt-IDs

`u16ObjId` (das zu bewegende Child-Objekt) und `u16ObjIdParent` (das Parent-Objekt) unterliegen **unterschiedlichen** Gültigkeitsregeln. Annex F der ISO 11783-6 definiert nur, welche Objekttypen als Child gültig sind — welche Objekttypen als Parent zulässig sind, steht separat in Annex B, jeweils unter „Allowed commands“ des Objekttyps.

`u16ObjId` — gültige Child-Objekttypen (Annex F.16): Container (3000–3999), Key (5000–5999), Button (6000–6999), Input Boolean Field (7000–7999), Input String Field (8000–8999), Input Number Field (9000–9999), Input List Field (10000–10999), Output String Field (11000–11999), Output Number Field (12000–12999), Output Line (13000–13999), Output Rectangle (14000–14999), Output Ellipse (15000–15999), Output Polygon (16000–16999), Meter (17000–17999), Linear Bar Graph (18000–18999), Arched Bar Graph (19000–19999), Picture Graphic (20000–20499), Object Pointer (27000–27999), Output List Object (37000–37999), External Object Pointer (43000–43999), Animation Object (44000–44999), Scaled Graphic Object (48000–48999), Auxiliary Function/Type2 (29000–29999 / 31000–31999), Auxiliary Input/Type2 (30000–30999 / 32000–32999), Auxiliary Control Designator (33000–33999).

`u16ObjIdParent` — gültige Parent-Objekttypen (Annex B, „Allowed commands“): WorkingSet (B.1), Data Mask (B.2), Alarm Mask (B.3), **Container (B.4)**, Key (B.6), Window Mask, Button, Auxiliary Function (Type 1/2), Auxiliary Input (Type 1/2).

Nicht zulässig als Parent: Soft Key Mask (B.5 listet das Kommando nicht als erlaubt) und Animation Object (dessen Kinder sind indexbasiert, nicht X/Y-positioniert).

ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei `INIT` den Baustein.

Funktionsweise

`QI` (Standardwert `TRUE`) wird mit den Ereignissen `INIT` und `REQ` übernommen und aktiviert den Dienst; `QO` wird mit `INITO` und `CNF` ausgegeben und zeigt an, ob der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` mit Child- und Parent-Objekt-ID
   - `INITO` bestätigt Betriebsbereitschaft
2. **Positionsänderung**:
   - `REQ` mit absoluten Koordinaten relativ zum Parent
   - Wertebereich: −32768 bis +32767 Pixel
   - `CNF` liefert Ergebnis und alte Position
3. **Fehlerbehandlung**:
   - ISO-standardisierte Fehlercodes
   - Detaillierte Statusmeldungen

Technische Besonderheiten

✓ **ISO 11783-6 konform** (F.16) ✓ **Absolute Positionsänderung** (relativ zum Parent) ✓ **16-bit Wertebereich** (±32768 Pixel) ✓ **Echtzeitfähige** Verarbeitung

Positionsbereich

Parameter	Bereich	Beschreibung
X-Position	−32768 - +32767	Horizontale Position
Y-Position	−32768 - +32767	Vertikale Position

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreich
−6	VT_E_OVERFLOW	Position außerhalb des gültigen Bereichs
−128	VT_E_HANDLE_INVALID	Ungültige Objekt-ID

Anwendungsszenarien

- **Präzise Objektplatzierung**: Exakte Positionierung von UI-Elementen
- **Dynamische Layouts**: Neuausrichtung von Bedienelementen
- **Komplexe Anordnungen**: Mehrschichtige Oberflächen
- **Maschinenvisualisierung**: Technische Zeichnungen

⚖️ **Vergleich mit ähnlichen Bausteinen**

Feature	Q_ChildPosition	Q_ChildLocation	VtAbsoluteMove
ISO-Standard	✓	✓	✖
Positionstyp	Absolut	Relativ	Absolut
Wertebereich	±32767 Pixel	±127 Pixel	±32767 Pixel

Fazit

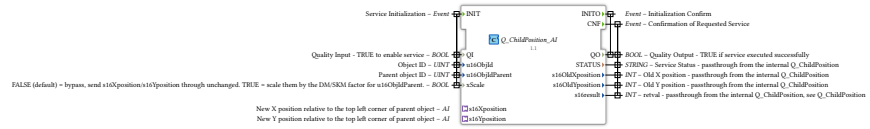
Der Q_ChildPosition-Baustein bietet die Referenzimplementierung für absolute Positionsänderungen:

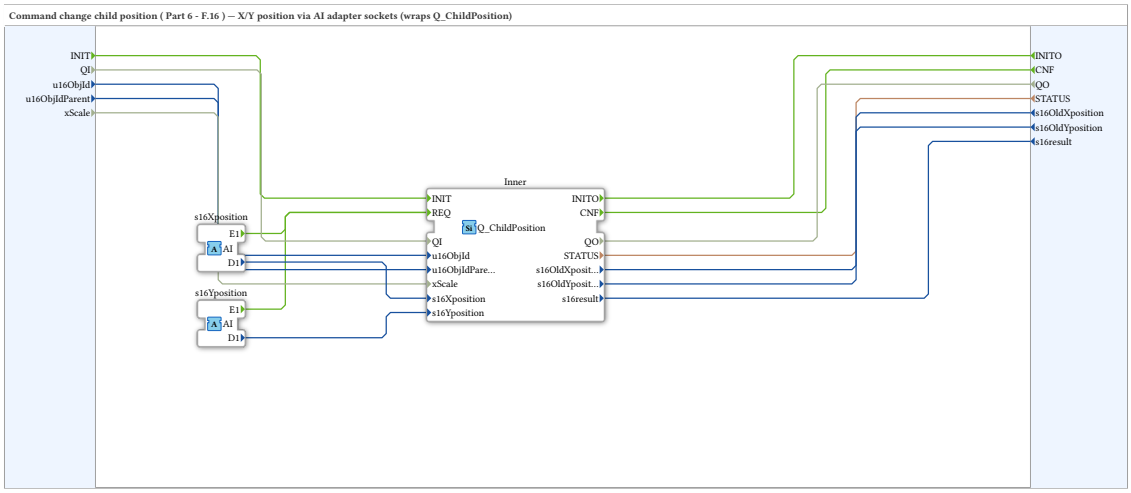
- **Präzise**: Pixelgenaue Positionierung
- **Flexibel**: Großer Wertebereich
- **Standardkonform**: Volle ISO 11783-6 Kompatibilität

Idealer Einsatz bei:

- Komplexen Benutzeroberflächen
- Technischen Zeichnungen
- Präzisen Layoutanforderungen

Q_ChildPosition_AI





• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock `Q_ChildPosition_AI` dient zur Steuerung der Position eines Kind-Objekts gemäß ISO 11783-6 (Teil 6, F.16). Er ist eine spezielle Variante des Bausteins `Q_ChildPosition`, bei dem die X- und Y-Koordinaten nicht über ein direktes REQ-Datenpaar, sondern über zwei unidirektionale AI-Adapter-Sockets empfangen werden. Diese Architektur ermöglicht eine lose Kopplung und eine flexible Anbindung an verschiedene Datenquellen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Service-Initialisierung

Das INIT-Ereignis initialisiert den internen Baustein und übernimmt die Parameter `u16ObjId`, `u16ObjIdParent` und `xScale`.

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Initialisierungsbestätigung
CNF	Event	Bestätigung des angeforderten Dienstes

Das CNF-Ereignis wird nach jeder erfolgreichen Positionsänderung ausgelöst und liefert die Ergebnisse (STATUS, alte Positionen, Rückgabewert).

Daten-Eingänge

Name	Typ	Kommentar
QI	BOOL	Quality Input – TRUE aktiviert den Dienst

u16ObjId	UINT	Objektidentifikator, Standardwert ID_NULL
u16ObjIdParent	UINT	Identifikator des Elternobjekts, Standardwert ID_NULL
xScale	BOOL	Skalierungsflag: FALSE (Standard) = unverändert durchreichen, TRUE = Skalierung mit dem DM/SKM-Faktor des Elternobjekts

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde
STATUS	STRING	Dienststatus; wird direkt vom internen Q_ChildPosition übernommen
s16OldXposition	INT	Vorherige X-Position (Passthrough)
s16OldYposition	INT	Vorherige Y-Position (Passthrough)
s16result	INT	Rückgabewert des Dienstes (Passthrough)

Adapter

Name	Typ	Kommentar
s16Xposition	adapter::types::unidirectional::AI	Neue X-Position relativ zur oberen linken Ecke des Elternobjekts
s16Yposition	adapter::types::unidirectional::AI	Neue Y-Position relativ zur oberen linken Ecke des Elternobjekts

Qualitätssignale QI/QO: Der Dateneingang QI (BOOL, Initialwert TRUE) ist dem Ereignis INIT zugeordnet und wird an QI des internen Bausteins Inner durchgereicht. Der Datenausgang QO (BOOL) wird von Inner.QO übernommen und ist den Ereignissen INITO und CNF zugeordnet.

Funktionsweise

Der Baustein wird über das INIT-Ereignis gestartet, wobei die Objekt-IDs und das Skalierungsflag an den internen Q_ChildPosition übergeben werden. Danach wartet er auf Ereignisse an den beiden AI-Adapter-Sockets. Sobald eines der beiden Sockets ein Ereignis (E1) liefert, liest der FB den aktuellen Datenwert (D1) dieses Sockets sowie den zuletzt empfangenen Wert des anderen Sockets und führt eine Positionsänderung durch. Dazu wird das REQ-Ereignis des internen Q_ChildPosition ausgelöst, das die gelesenen X- und Y-Werte als neue Position verarbeitet.

Das Ergebnis (alter und neuer Zustand) wird über das CNF-Ereignis nach außen zurückgemeldet. Die beiden AI-Sockets sind unidirektional; sie liefern nur Daten an den FB, ohne dass eine Rückantwort über den Adapter erfolgt.

Technische Besonderheiten

- Der FB ist ein reiner Wrapper um Q_ChildPosition und verwendet dessen Implementierung unverändert.
- Die Positionseingänge sind als AI-Adapter-Sockets vom Typ unidirectional ausgeführt, wodurch eine klare Trennung zwischen Ereignis- und Datenschnittstelle entsteht.
- Das optionale Skalierungsflag xScale erlaubt eine Anpassung der Positionswerte an die Skalierung des Elternobjekts, ohne zusätzliche externe Logik.
- Die beiden Adapter können unabhängig voneinander und in beliebiger Reihenfolge Ereignisse liefern; der FB verarbeitet jedes Ereignis und verwendet dabei den jeweils aktuellen Wert auf beiden Sockets.

Zustandsübersicht

Der Baustein besitzt keinen expliziten internen Zustandsautomaten, sondern arbeitet ereignisgesteuert:

- **Initialisierungszustand:** Nach dem INIT-Ereignis wird der interne FB initialisiert, danach ist er bereit für Positionsänderungen.

- **Wartezustand:** Der FB wartet auf ein Ereignis an einem der beiden AI-Sockets.
- **Verarbeitungszustand:** Bei Eingang eines Ereignisses wird die Positionsänderung durchgeführt und das Ergebnis über `CNF` ausgegeben.

Anwendungsszenarien

Dieser FB eignet sich für alle Systeme, die Positionsdaten eines Kind-Objekts (z. B. grafische Symbole) über unidirektionale Adapterschnittstellen erhalten und an ein ISOBUS-Terminal nach ISO 11783 senden möchten. Typische Einsatzbereiche sind landwirtschaftliche Maschinen und Anbaugeräte, bei denen eine flexible Signalverkabelung und eine modulare Softwarearchitektur gefordert sind.

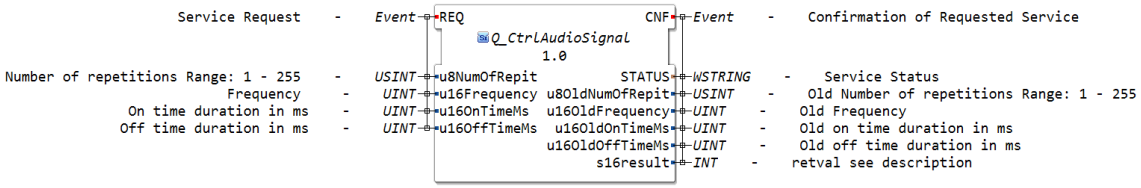
Vergleich mit ähnlichen Bausteinen

Der direkte Baustein `Q_ChildPosition` erwartet die Positionsdaten über ein separates REQ-Ereignis und direkte Eingangsvariablen. `Q_ChildPosition_AI` bietet dieselbe Funktionalität, ersetzt jedoch diese Schnittstelle durch zwei AI-Adapter-Sockets, wodurch die Integration in adapterbasierte Systeme deutlich erleichtert wird. Dies reduziert die Verdrahtung und erhöht die Wiederverwendbarkeit in übergeordneten Netzwerken.

Fazit

`Q_ChildPosition_AI` ist eine sinnvolle Erweiterung für Anwendungen, die eine lose, ereignisorientierte Datenanbindung benötigen. Die Verwendung unidirektionaler AI-Sockets macht den FB flexibel und leicht in bestehende ISOBUS-Architekturen integrierbar, ohne auf die bewährte Funktionalität des zugrunde liegenden `Q_ChildPosition` zu verzichten. Die optionale Skalierung und die klare Schnittstellenstruktur erhöhen die Einsatzmöglichkeiten in modernen Steuerungssystemen.

Q_CtrlAudioSignal

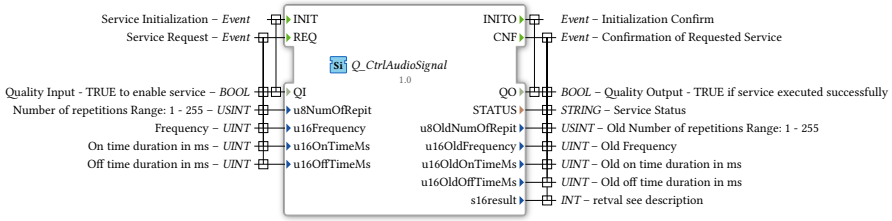


• — ○ • — ○ • — ○ •

Einleitung

Der **Q_CtrlAudioSignal** ist ein standardkonformer Funktionsbaustein zur Steuerung von Audiogeräten in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.10) Spezifikation für landwirtschaftliche Steuersysteme.

Der Befehl F.10 adressiert direkt das **Audiogerät** des VT — er referenziert **kein Objekt**, daher besitzt dieser Baustein keinen Objekt-ID-Eingang. **Aktivierungen = 0** beendet einen laufenden Ton des absendenden Arbeitsbereichs (Frequenz- und Zeitwerte werden ignoriert); 1–255 legt die Anzahl der Audio-Aktivierungen fest.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung
- `REQ`: Audiosteuerungs-Anforderung (mit Signalparametern)

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung

- `CNF`: Steuerungsbestätigung (mit Ergebnisdaten)

Daten-Eingänge

- `QI` (BOOL, Standardwert `TRUE`): Quality Input – TRUE aktiviert den Dienst (mit `INIT` und `REQ` verknüpft)
- `u8NumOfRepit` (USINT): Wiederholungen (1-255)
- `u16Frequency` (UINT): Frequenz in Hz
- `u16OnTimeMs` (UINT): Einschaltdauer in ms
- `u16OffTimeMs` (UINT): Ausschaltdauer in ms

Daten-Ausgänge

- `QO` (BOOL): Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde (mit `INITO` und `CNF` verknüpft)
- `STATUS` (STRING): Betriebsstatusmeldung
- `u8OldNumOfRepit` (USINT): Vorherige Wiederholungen
- `u16OldFrequency` (UINT): Vorherige Frequenz
- `u16OldOnTimeMs` (UINT): Vorherige Einschaltdauer
- `u16OldOffTimeMs` (UINT): Vorherige Ausschaltdauer
- `s16result` (INT): ISO-konformer Ergebniscode

Funktionsweise

`QI` (Standardwert `TRUE`) wird mit den Ereignissen `INIT` und `REQ` übernommen und aktiviert den Dienst; `QO` wird mit `INITO` und `CNF` ausgegeben und zeigt an, ob der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` ohne Parameter
   - `INITO` bestätigt Betriebsbereitschaft
2. **Audiosteuerung**:
   - `REQ` mit Signalparametern auslösen
   - Erzeugt Ton mit konfigurierbarem Muster
   - `CNF` liefert Ergebnis und alte Werte
3. **Fehlerbehandlung**:
   - ISO-standardisierte Fehlercodes
   - Detaillierte Statusmeldungen

Technische Besonderheiten

✓ **ISO 11783-6 konform** (F.10) ✓ **Flexible Signalsteuerung**: Frequenz, Dauer, Wiederholungen ✓ **16-bit Zeitauflösung** (0-65535 ms) ✓ **255 Wiederholungen** maximal

Standardparameterbereiche

Parameter	Bereich	Typischer Wert
Frequenz	0-65535 Hz	2000 Hz
Einschaltdauer	0-65535 ms	200 ms
Ausschaltdauer	0-65535 ms	100 ms
Wiederholungen	1-255	3

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreich
−6	VT_E_OVERFLOW	Ungültige Parameterwerte
−8	VT_E_NOACT	VT im falschen Zustand
−128	VT_E_HANDLE_INVALID	Ungültige Audio-Konfiguration

Anwendungsszenarien

- **Warnsignale**: Akustische Alarmmeldungen
- **Bestätigungstöne**: Bedienfeedback
- **Sprachansagen**: Voraufgezeichnete Meldungen
- **Maschinenstatus**: Akustische Statusindikatoren

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	Q_CtrlAudioSignal	VtSoundManager	VtAudioAlert
ISO-Standard	✓	✖	✖
Parametrierung	Vollständig	Teilweise	Einfach
Frequenzkontrolle	Ja	Nein	Nein

🛠️ Zugehörige Übungen

- Uebung_017
- Uebung_018
- Uebung_018a

Fazit

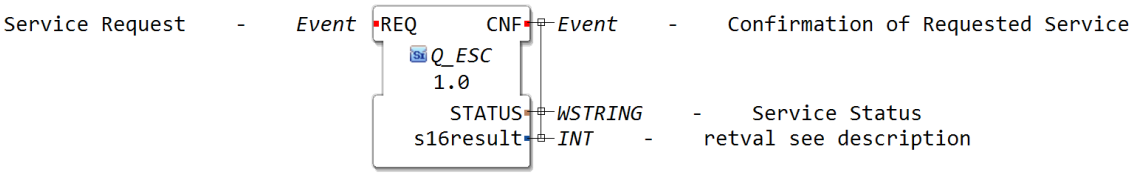
Der Q_CtrlAudioSignal-Baustein bietet die Referenzimplementierung für VT-Audiosteuerung:

- **Flexibel**: Voll parametrierbare Tonsignale
- **Standardkonform**: Volle ISO 11783-6 Kompatibilität
- **Robust**: Bewährte Technik in Feldgeräten

Unverzichtbar für:

- Sicherheitskritische Warnsignale
- Benutzerfreundliche akustische Rückmeldungen
- Maschinen mit komplexen Audiostatusmeldungen

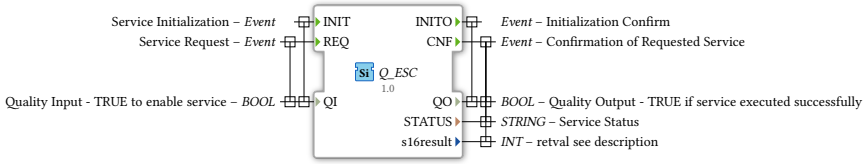
Q_ESC



• – ○ • – ○ • – ○ •

Einleitung

Der **Q_ESC** ist ein standardkonformer Funktionsbaustein zur Abbruchsteuerung in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.8) Spezifikation für landwirtschaftliche Steuersysteme.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung
- `REQ`: Abbruchanforderung

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Abbruchbestätigung (mit Status)

Daten-Eingänge

- `QI` (BOOL, Standardwert `TRUE`): Quality Input – TRUE aktiviert den Dienst (mit `INIT` und `REQ` verknüpft)

Daten-Ausgänge

- `QO` (BOOL): Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde (mit `INITO` und `CNF` verknüpft)
- `STATUS` (STRING): Betriebsstatusmeldung
- `s16result` (INT): ISO-konformer Ergebniscode

Funktionsweise

`QI` (Standardwert `TRUE`) wird mit den Ereignissen `INIT` und `REQ` übernommen und aktiviert den Dienst; `QO` wird mit `INITO` und `CNF` ausgegeben und zeigt an, ob der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` ohne Parameter
   - `INITO` bestätigt Betriebsbereitschaft
2. **Abbruchfunktion**:
   - `REQ` löst Abbruch der aktuellen Benutzereingabe aus
   - Keine zusätzlichen Parameter erforderlich
   - `CNF` liefert Ergebnisstatus
3. **Fehlerbehandlung**:
   - ISO-standardisierte Fehlercodes
   - Detaillierte Statusmeldungen

Technische Besonderheiten

✓ **ISO 11783-6 konform** (F.8) ✓ **Universeller Abbruchbefehl** ✓ **Keine Konfiguration** erforderlich ✓ **Sofortige Wirkung** bei Ausführung

Typische Anwendungsfälle

Szenario	Wirkung
Dialogabbruch	Beendet aktuelle Eingabe
Menüverlassen	Zurück zur Hauptebene
Alarmquittierung	Stoppt akustische Signale

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreich abgebrochen
−8	VT_E_NOACT	Keine aktive Eingabe zum Abbrechen
−21	VT_E_NO_INSTANCE	Keine VT-Instanz verfügbar
−130	VT_E_NOT_ALIVE	VT nicht erreichbar

Anwendungsszenarien

- **Notfallabbruch**: Schnelle Beendigung von Prozessen
- **Eingabefehler**: Korrekturmöglichkeit für Benutzer
- **Menünavigation**: Zurück-Funktionalität
- **Systemreset**: Abbruch aller aktiven Operationen

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	Q_ESC	VtAbortCmd	VtCancelOp
ISO-Standard	✓	✖	✖
Wirkung	Global	Lokal	Objektspezifisch
Parameter	Keine	Konfigurierbar	Konfigurierbar

Fazit

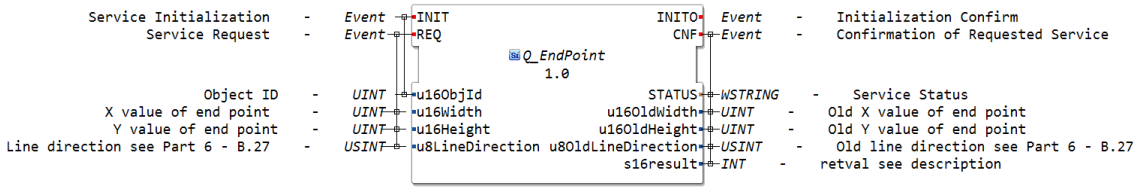
Der Q_ESC-Baustein bietet die standardisierte Abbruchfunktion für ISOBUS-Systeme:

- **Universal**: Für alle Abbruchszenarien geeignet
- **Einfach**: Keine Konfiguration notwendig
- **Zuverlässig**: Sofortige Systemreaktion

Essentiell für:

- Sicherheitskritische Anwendungen
- Benutzerfreundliche Oberflächen
- Robuste Fehlerbehandlung

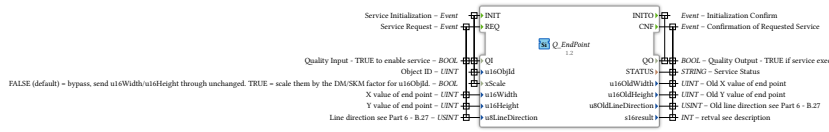
Q_EndPoint



• — ○ • — ○ • — ○ •

Einleitung

Der **Q_EndPoint** ist ein standardkonformer Funktionsbaustein zur Steuerung von Linienendpunkten in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.26) Spezifikation für landwirtschaftliche Steuersysteme.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Objekt-ID und `xScale`)
- `REQ`: Endpunktänderungs-Anforderung (mit Position/Richtung)

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Änderungsbestätigung (mit Ergebnisdaten)

Daten-Eingänge

- `QI` (BOOL, Standardwert `TRUE`): Quality Input – TRUE aktiviert den Dienst (mit `INIT` und `REQ` verknüpft)
- `u16ObjId` (UINT): Linienobjekt-ID
- `xScale` (BOOL, Initialwert `FALSE`): `FALSE` = Bypass, `u16Width`/`u16Height` werden unverändert gesendet; `TRUE` = beide werden mit dem DM/SKM-Faktor für `u16ObjId` skaliert
- `u16Width` (UINT): Neue X-Endposition (0-65535 Pixel)
- `u16Height` (UINT): Neue Y-Endposition (0-65535 Pixel)
- `u8LineDirection` (USINT): Linienrichtung (ISO 11783-6 B.27)

Daten-Ausgänge

- `QO` (BOOL): Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde (mit `INITO` und `CNF` verknüpft)
- `STATUS` (STRING): Betriebsstatusmeldung
- `u16OldWidth` (UINT): Vorherige X-Endposition
- `u16OldHeight` (UINT): Vorherige Y-Endposition
- `u8OldLineDirection` (USINT): Vorherige Linienrichtung
- `s16result` (INT): ISO-konformer Ergebniscode

Gültige Objekt-IDs

`u16ObjId` — gültiger Objekttyp (Anhang F.26, Output-Line-Objekt): Output Line (13000–13999). ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei `INIT` den Baustein.

Funktionsweise

`QI` (Standardwert `TRUE`) wird mit den Ereignissen `INIT` und `REQ` übernommen und aktiviert den Dienst; `QO` wird mit `INITO` und `CNF` ausgegeben und zeigt an, ob der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` mit Linienobjekt-ID
   - `INITO` bestätigt Betriebsbereitschaft
2. **Endpunktänderung**:
   - `REQ` mit neuen Positions- und Richtungsparametern
   - Ändert Endpunkt und Ausrichtung der Linie
   - `CNF` liefert Ergebnis und alte Werte
3. **Fehlerbehandlung**:
   - ISO-standardisierte Fehlercodes
   - Detaillierte Statusmeldungen

Technische Besonderheiten

✓ **ISO 11783-6 konform** (F.26 + B.27) ✓ **16-bit Positionsauflösung** (0-65535 Pixel) ✓ **Vollständige Liniensteuerung** (Position + Richtung) ✓ **Echtzeitfähige** Verarbeitung

Linienrichtungen (B.27)

Wert	Richtung	Beschreibung
0	Horizontal	Von links nach rechts
1	Vertikal	Von oben nach unten
2	Diagonal rechts	Links oben ⟶ rechts unten
3	Diagonal links	Rechts oben ⟶ links unten

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreich
−6	VT_E_OVERFLOW	Ungültige Positionswerte
−128	VT_E_HANDLE_INVALID	Ungültige Linienobjekt-ID
−129	VT_E_ISO_INSTANCE_INVALID	Ungültige Instanz

Anwendungsszenarien

- **Maschinenvisualisierung**: Dynamische Verbindungslinien
- **Diagramme**: Anpassbare Datenanzeigen
- **Technische Zeichnungen**: Konfigurierbare Hilfslinien

- **UI-Design**: Adaptive Trennlinien

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	Q_EndPoint	VtLineEditor	VtDynamicLine
ISO-Standard	✓	✖	✖
Steuerung	Endpunkt	Vollständig	Nur Position
Richtungen	4 Optionen	8 Optionen	Keine

Fazit

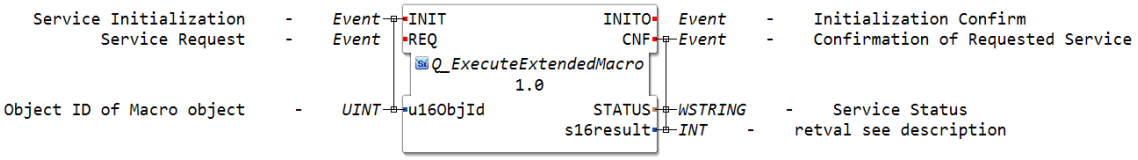
Der Q_EndPoint-Baustein bietet die Referenzimplementierung für Linienendpunkt-Steuerung:

- **Präzise**: Pixelgenaue Positionierung
- **Flexibel**: Kombinierte Positions- und Richtungssteuerung
- **Standardkonform**: Volle ISO 11783-6 Kompatibilität

Idealer Einsatz bei:

- Dynamischen technischen Zeichnungen
- Adaptiven Diagrammanzeigen
- Konfigurierbaren Benutzeroberflächen

Q_ExecuteExtendedMacro



• — ○ • — ○ • — ○ •

Einleitung

Der **Q_ExecuteExtendedMacro** ist ein standardkonformer Funktionsbaustein zur Ausführung erweiterter Makros in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.62) Spezifikation für VT ab Version 5.

Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Makro-ID)
- `REQ`: Makroausführungs-Anforderung

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Ausführungsbestätigung (mit Status)

Daten-Eingänge

- `QI` (BOOL, Standardwert `TRUE`): Quality Input – TRUE aktiviert den Dienst (mit `INIT` und `REQ` verknüpft)
- `u16ObjId` (UINT): Makro-Objekt-ID (16-bit)

Daten-Ausgänge

- `QO` (BOOL): Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde (mit `INITO` und `CNF` verknüpft)

- `STATUS` (STRING): Betriebsstatusmeldung
- `s16result` (INT): ISO-konformer Ergebniscode

Gültige Objekt-IDs

`u16ObjId` adressiert das Macro-Objekt (Bereich **1–999**, 16-Bit-Feld in Bytes 2,3 des Befehls F.62). Im Gegensatz zu Q_ExecuteMacro (F.48, 8-Bit-Feld) können hier alle Macro-IDs 1–999 angesprochen werden.
ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei `INIT` den Baustein.

Funktionsweise

`QI` (Standardwert `TRUE`) wird mit den Ereignissen `INIT` und `REQ` übernommen und aktiviert den Dienst; `QO` wird mit `INITO` und `CNF` ausgegeben und zeigt an, ob der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` mit Makro-Objekt-ID
   - `INITO` bestätigt Betriebsbereitschaft
2. **Makroausführung**:
   - `REQ` ohne zusätzliche Parameter
   - Führt gespeicherte Makrosequenz aus
   - `CNF` liefert Ergebnisstatus
3. **Fehlerbehandlung**:
   - ISO-standardisierte Fehlercodes
   - Detaillierte Statusmeldungen

Technische Besonderheiten

✓ **ISO 11783-6 konform** (F.62) ✓ **Exklusiv für VT Version 5+** ✓ **16-bit Makro-IDs** (erweiterter Adressraum) ✓ **Atomare Ausführung** (Transaktionssicher)

Makro-Eigenschaften

Feature	Beschreibung
Speicherort	Im VT dauerhaft gespeichert
Größe	Bis zu 255 Befehle
Inhalt	VT-Kommandosequenzen
ID-Bereich	0x0001 - 0xFEFF (reserviert)

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreiche Ausführung
−6	VT_E_OVERFLOW	Makro zu komplex
−8	VT_E_NOACT	VT nicht bereit
−129	VT_E_ISO_INSTANCE_INVALID	Ungültige VT-Instanz

Anwendungsszenarien

- **Automatisierte Abläufe**: Komplexe Befehlsketten
- **Systeminitialisierung**: Startprozeduren
- **Diagnoseroutinen**: Automatisierte Tests
- **Benutzeraktionen**: Makroaufzeichnungen

⚖️ **Vergleich mit ähnlichen Bausteinen**

Feature	Q_ExecuteExtendedMacro	VtMacroPlayer	VtCommandSequence

ISO-Standard	✓	✖	✖
VT-Version	5+	Alle	Alle
ID-Typ	16-bit	8-bit	16-bit

Fazit

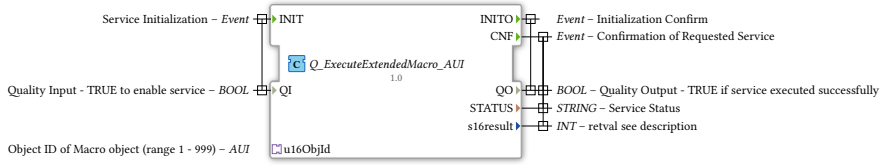
Der Q_ExecuteExtendedMacro-Baustein bietet die Standardimplementierung für erweiterte Makros:

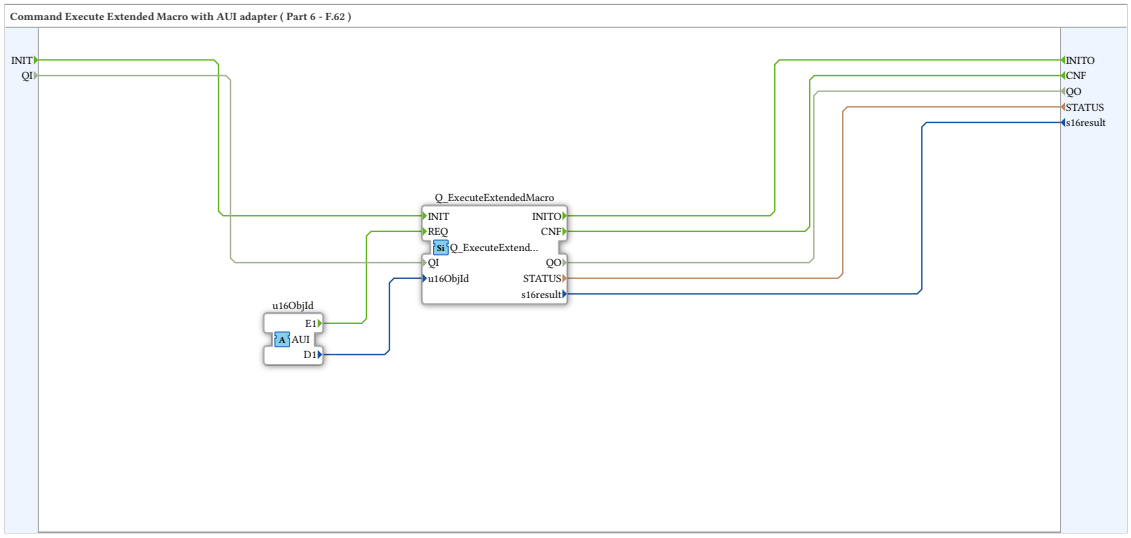
- **Leistungsstark**: Komplexe Automatisierungen möglich
- **Zukunftssicher**: Für VT Generation 5 optimiert
- **Zuverlässig**: Atomare Ausführung garantiert

Unverzichtbar für:

- Hochautomatisierte Agrarsysteme
- Komplexe Bedienprozesse
- Systeme mit erweitertem Funktionsumfang

Q_ExecuteExtendedMacro_AUI





• — ○ • — ○ • — ○ •

Einleitung

Der **Q_ExecuteExtendedMacro_AUI** ist ein AUI-Adapter-Wrapper für den Baustein **Q_ExecuteExtendedMacro** (ISO 11783-6, Teil 6 – F.62). Er ermöglicht das Ausführen von Extended-Makros (16-Bit Makro-ID 1–999) auf einem ISOBUS Terminal über eine unidirektionale AUI-Schnittstelle.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
`INIT`	`EInit`	Service-Initialisierung

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Bestätigung der Initialisierung
CNF	Event	Bestätigung des angeforderten Dienstes

Daten-Eingänge

Außer QI gibt es keine direkten Dateneingänge – die Extended-Makro-ID wird über den AUI-Adapter-Socket empfangen.

Name	Typ	Initialwert	Kommentar
QI	BOOL	TRUE	Quality Input – TRUE aktiviert den Dienst

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde
STATUS	STRING	Dienststatus
s16result	INT	Rückgabewert des Befehls

Adapter

Typ	Name	Richtung	Kommentar
adapter::types::unidirectional::AUI	u16ObjId	Socket (Eingang)	Object-ID des Extended-Makros (1–999)

Qualitätssignale QI/QO: Der Dateneingang QI (BOOL, Initialwert TRUE) ist dem Ereignis INIT zugeordnet und wird an QI des internen Bausteins Q_ExecuteExtendedMacro durchgereicht. Der Datenausgang QO (BOOL) wird von Q_ExecuteExtendedMacro.QO übernommen und ist den Ereignissen INITO und CNF zugeordnet.

Funktionsweise

Verbindet den internen FB Q_ExecuteExtendedMacro (*isobus::UT::Q::Q_ExecuteExtendedMacro*) mit der AUI-Adapter-Schnittstelle. Ein Ereignis an u16ObjId.E1 löst REQ am internen FB aus, der die 16-Bit Makro-ID aus u16ObjId.D1 an das Terminal übermittelt.

Technische Besonderheiten

- **16-Bit Extended-Makro-Bereich:** Unterstützt Makro-IDs von 1 bis 999 (ISO 11783-6 VT Version 5+).
- **Adapter-Wrapper:** Unidirektionale AUI-Anbindung.

Zustandsübersicht

Delegiert alle Aktionen an den inneren FB Q_ExecuteExtendedMacro.

Anwendungsszenarien

- Ausführen großer Extended-Makros in VT Version 5+ Umgebungen per AUI-Adapter.

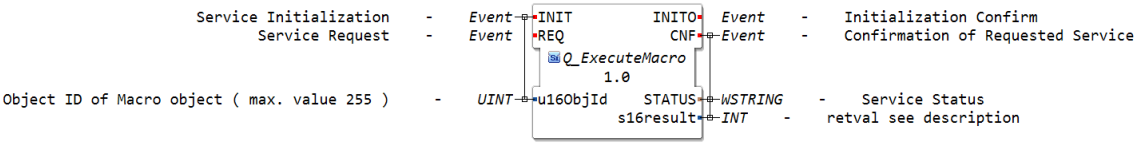
Vergleich mit ähnlichen Bausteinen

Für Standard-Makros (IDs 1–255) steht **Q_ExecuteMacro_AUI** zur Verfügung; für 16-Bit Makro-IDs (1–999) wird **Q_ExecuteExtendedMacro_AUI** eingesetzt.

Fazit

Q_ExecuteExtendedMacro_AUI ist der standardisierte AUI-Wrapper zur Ausführung von Extended-Makros in ISOBUS VT Version 5+ Systemen.

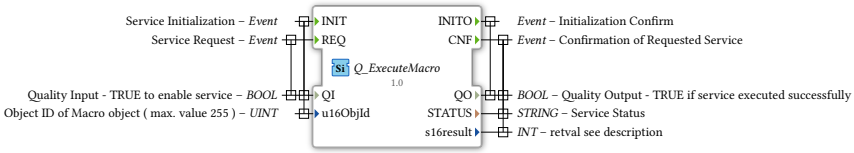
Q_ExecuteMacro



• — ○ • — ○ • — ○ •

Einleitung

Der **Q_ExecuteMacro** ist ein standardkonformer Funktionsbaustein zur Ausführung von Makros in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.48) Spezifikation für VT ab Version 4.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Makro-ID)
- `REQ`: Makroausführungs-Anforderung

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Ausführungsbestätigung (mit Status)

Daten-Eingänge

- `QI` (BOOL, Standardwert `TRUE`): Quality Input – TRUE aktiviert den Dienst (mit `INIT` und `REQ` verknüpft)
- `u16ObjId` (UINT): Makro-Objekt-ID (max. Wert 255)

Daten-Ausgänge

- `QO` (BOOL): Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde (mit `INITO` und `CNF` verknüpft)
- `STATUS` (STRING): Betriebsstatusmeldung
- `s16result` (INT): ISO-konformer Ergebniscode

Gültige Objekt-IDs

`u16ObjId` adressiert das Macro-Objekt. Annex A definiert den Bereich 1–999, aber der Befehl F.48 trägt die Macro-Objekt-ID nur in **einem einzelnen Byte** (Byte 2). Dadurch sind mit diesem Baustein nur **IDs 1–255** adressierbar (Werte > 255 werden abgeschnitten). Für Macro-IDs 256–999 ist Q_ExecuteExtendedMacro (F.62) zu verwenden.

ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei `INIT` den Baustein.

Funktionsweise

`QI` (Standardwert `TRUE`) wird mit den Ereignissen `INIT` und `REQ` übernommen und aktiviert den Dienst; `QO` wird mit `INITO` und `CNF` ausgegeben und zeigt an, ob der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` mit Makro-Objekt-ID (8-bit kompatibel)
   - `INITO` bestätigt Betriebsbereitschaft
2. **Makroausführung**:
   - `REQ` ohne zusätzliche Parameter
   - Führt gespeicherte Makrosequenz aus
   - `CNF` liefert Ergebnisstatus
3. **Fehlerbehandlung**:

- ISO-standardisierte Fehlercodes
- Detaillierte Statusmeldungen

Technische Besonderheiten

✓ **ISO 11783-6 konform** (F.48) ✓ **Kompatibel mit VT Version 4+** ✓ **8-bit Makro-IDs** (0-255) ✓ **Atomare Ausführung** (Transaktionssicher)

Makro-Eigenschaften

Feature	Beschreibung
Speicherort	Im VT dauerhaft gespeichert
Größe	Bis zu 255 Befehle
Inhalt	VT-Kommandosequenzen
ID-Bereich	0x01 - 0xFF (8-bit)

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreiche Ausführung
−6	VT_E_OVERFLOW	Makro zu komplex
−8	VT_E_NOACT	VT nicht bereit
−129	VT_E_ISO_INSTANCE_INVALID	Ungültige VT-Instanz

Anwendungsszenarien

- **Standardabläufe**: Häufig genutzte Befehlsketten
- **Systemstart**: Basisinitialisierungen
- **Bediensequenzen**: Automatisierte Menünavigation
- **Diagnose**: Vordefinierte Testroutinen

⚖️ **Vergleich mit ähnlichen Bausteinen**

Feature	Q_ExecuteMacro	Q_ExecuteExtendedMacro	VtMacroRunner
ISO-Standard	✓	✓	✖
VT-Version	4+	5+	Alle
ID-Typ	8-bit	16-bit	8-bit

Fazit

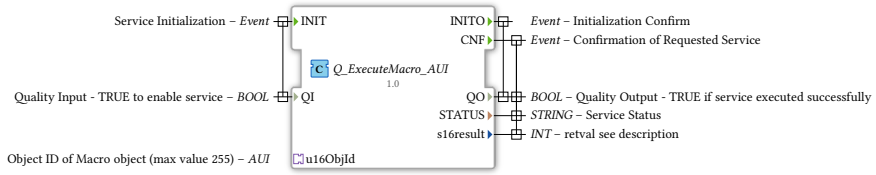
Der Q_ExecuteMacro-Baustein bietet die Standardimplementierung für Basis-Makros:

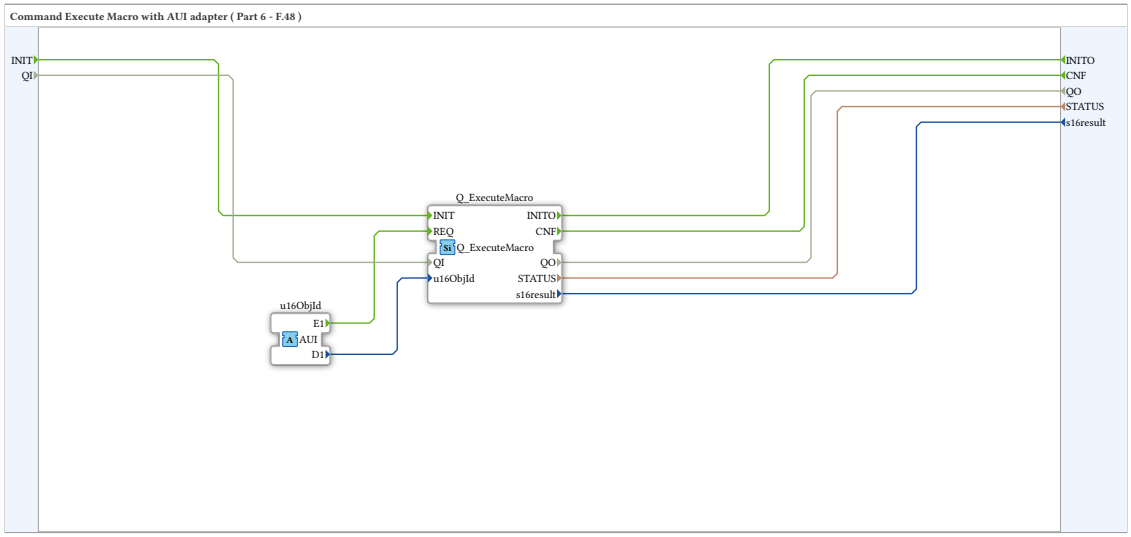
- **Einfach**: Kompakte 8-bit Adressierung
- **Kompatibel**: Funktioniert mit VT Generation 4+
- **Zuverlässig**: Bewährte Technik in Seriengeräten

Idealer Einsatz bei:

- Standard-Automatisierungen
- Kompatibilitätskritischen Systemen
- Einfachen Makroanwendungen

Q_ExecuteMacro_AUI





• — ○ • — ○ • — ○ •

Einleitung

Der **Q_ExecuteMacro_AUI** ist ein AUI-Adapter-Wrapper für den Baustein **Q_ExecuteMacro** (ISO 11783-6, Teil 6 – F.48). Er ermöglicht das Ausführen von Makros auf einem ISOBUS Terminal über eine unidirektionale AUI-Schnittstelle (UINT). Der Baustein kapselt den internen `Q_ExecuteMacro`-Kern und bietet eine einfache, adapterbasierte Anbindung.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
`INIT`	`EInit`	Service-Initialisierung

Ereignis-Ausgänge

Name	Typ	Kommentar
`INITO`	`EInit`	Bestätigung der Initialisierung
`CNF`	`Event`	Bestätigung des angeforderten Dienstes

Daten-Eingänge

Außer `QI` gibt es keine direkten Dateneingänge – die Makro-Object-ID wird über den AUI-Adapter-Socket empfangen.

Name	Typ	Initialwert	Kommentar
`QI`	`BOOL`	`TRUE`	Quality Input – TRUE aktiviert den Dienst

Daten-Ausgänge

Name	Typ	Kommentar

QO	BOOL	Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde
STATUS	STRING	Dienststatus
s16result	INT	Rückgabewert des Befehls

Adapter

Typ	Name	Richtung	Kommentar
adapter::types::unidirectional::AUI	u16ObjId	Socket (Eingang)	Object-ID des auszuführenden Makros (1–255)

Qualitätssignale QI/QO: Der Dateneingang QI (BOOL, Initialwert TRUE) ist dem Ereignis INIT zugeordnet und wird an QI des internen Bausteins Q_ExecuteMacro durchgereicht. Der Datenausgang QO (BOOL) wird von Q_ExecuteMacro.QO übernommen und ist den Ereignissen INITO und CNF zugeordnet.

Funktionsweise

Der Baustein verbindet intern den Funktionsblock Q_ExecuteMacro (*isobus::UT::Q::Q_ExecuteMacro*) mit der AUI-Adapter-Schnittstelle. Beim Eintreffen eines Ereignisses am Socket u16ObjId.E1 wird das Ereignis REQ des internen FBs ausgelöst und die Makro-ID aus u16ObjId.D1 übergeben. Nach der Ausführung sendet der interne FB eine Bestätigung über CNF zurück und aktualisiert STATUS sowie s16result.

Technische Besonderheiten

- **Adapter-Wrapper:** Sämtliche Eingabedaten werden über den AUI-Socket empfangen.
- **8-Bit-Makro-Bereich:** Q_ExecuteMacro verarbeitet Makro-IDs im Bereich 1–255.
- **ISOBUS-Konformität:** Gemäß ISO 11783-6 Annex F.48 spezifiziert.

Zustandsübersicht

Der Wrapper selbst delegiert alle Zustände an den inneren FB Q_ExecuteMacro.

1. **Initialisierung** – INIT initialisiert den Baustein und meldet INITO.
2. **Befehlsausführung** – Ereignis an u16ObjId.E1 stößt Makro-Ausführung an.
3. **Bestätigung** – CNF meldet den Ausführungsstatus.

Anwendungsszenarien

- Ausführen von VT-Makros per AUI-Adapter in ISO 11783-6 Steuerungen.
- Automatisierte Makro-Ausführung bei Ereignissen oder Sequenzschritten.

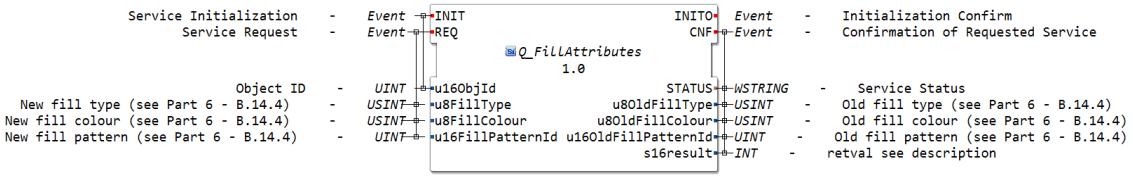
Vergleich mit ähnlichen Bausteinen

Im Vergleich zum direkten Q_ExecuteMacro bietet **Q_ExecuteMacro_AUI** eine adapterbasierte Anbindung. Für Makro-IDs > 255 wird der Baustein **Q_ExecuteExtendedMacro_AUI** verwendet.

Fazit

Q_ExecuteMacro_AUI ist ein praktischer Adapter-Wrapper zur Ausführung von ISOBUS-VT-Makros über AUI-Schnittstellen.

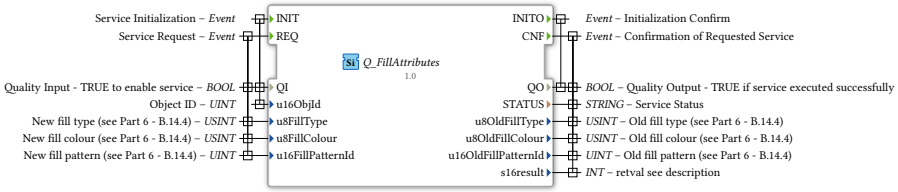
Q_FillAttributes



• – ○ • – ○ • – ○ •

Einleitung

Der **Q_FillAttributes** ist ein standardkonformer Funktionsbaustein zur Änderung von Füllattributen in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.32) Spezifikation für VT-Systeme.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Objekt-ID)
- `REQ`: Attributänderungs-Anforderung (mit neuen Füllattributen)

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Änderungsbestätigung (mit Status und alten Attributen)

Daten-Eingänge

- `QI` (BOOL, Standard `TRUE`): Quality-Eingang – `TRUE` aktiviert den Dienst
- `u16ObjId` (UINT): Objekt-ID (16-bit)
- `u8FillType` (USINT): Neuer Fülltyp
- `u8FillColour` (USINT): Neue Füllfarbe
- `u16FillPatternId` (UINT): Neues Füllmuster

Daten-Ausgänge

- `QO` (BOOL): Quality-Ausgang – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde
- `STATUS` (STRING): Betriebsstatusmeldung
- `u8OldFillType` (USINT): Alter Fülltyp
- `u8OldFillColour` (USINT): Alte Füllfarbe
- `u16OldFillPatternId` (UINT): Altes Füllmuster
- `s16result` (INT): ISO-konformer Ergebniscode

Gültige Objekt-IDs

`u16ObjId` adressiert das Zielobjekt des Kommandos **F.32** (Change Fill Attributes). Gültig ist ein **Fill-Attributes-Objekt**:

FillAttributesObject: 25000–25999
ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei `INIT` den Baustein.

Funktionsweise

Quality-Handling (QI/QO): Der Eingang `QI` (BOOL, Standard `TRUE`) ist mit den Ereignissen `INIT` und `REQ` verknüpft; `TRUE` aktiviert den Dienst. Der Ausgang `QO` (BOOL) ist mit `INITO` und `CNF` verknüpft und zeigt mit `TRUE` an, dass der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` mit Objekt-ID
   - `INITO` bestätigt Betriebsbereitschaft
2. **Attributänderung**:
   - `REQ` mit neuen Füllattributen
   - Ändert Fülleigenschaften des Objekts
   - `CNF` liefert Ergebnisstatus und vorherige Werte
3. **Fehlerbehandlung**:
   - ISO-standardisierte Fehlercodes
   - Detaillierte Statusmeldungen

Technische Besonderheiten

✓ **ISO 11783-6 konform** (F.32) ✓ **Umfassende Attributsteuerung** (Typ, Farbe, Muster) ✓ **Rückmeldung alter Werte** für Reversionsmöglichkeit ✓ **Transaktionssicher** (Atomare Ausführung)

Attribut-Eigenschaften

Feature	Beschreibung
Fülltypen	Nach ISO 11783-6 B.14.4
Farbpalette	8-bit Farbindex
Muster	16-bit Musterreferenzen
Objektbereich	Alle füllfähigen VT-Objekte

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreiche Änderung
−6	VT_E_OVERFLOW	Pufferüberlauf
−8	VT_E_NOACT	VT nicht bereit
−21	VT_E_NO_INSTANCE	Kein VT-Client verfügbar
−129	VT_E_ISO_INSTANCE_INVALID	Ungültige VT-Instanz
−130	VT_E_NOT_ALIVE	VT nicht aktiv

Anwendungsszenarien

- **Designanpassungen**: Dynamische Oberflächengestaltung
- **Statusvisualisierung**: Farbliche Hervorhebungen
- **Musterwechsel**: Kontextsensitive Darstellungen
- **Themenwechsel**: Dunkel-/Hellmodus Umschaltung

⚖️ **Vergleich mit ähnlichen Bausteinen**

Feature	Q_FillAttributes	VtSetColor	VtPatternManager
ISO-Standard	✓	✖	✖
Farbsteuerung	✓	✓	✖
Mustersteuerung	✓	✖	✓
Typsteuerung	✓	✖	✖

Fazit

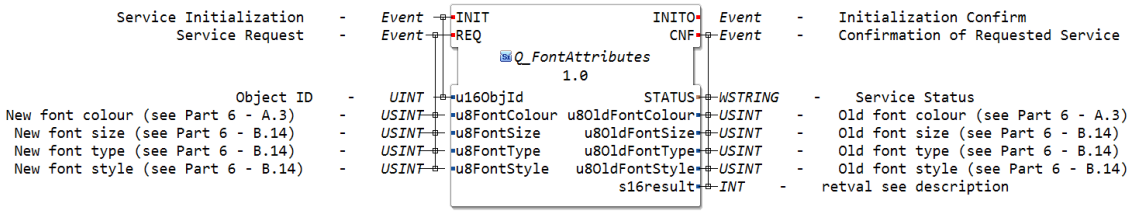
Der Q_FillAttributes-Baustein bietet die Standardimplementierung für Füllattributänderungen:

- **Vielseitig**: Komplette Kontrolle über Fülleigenschaften
- **Rückverfolgbar**: Rückmeldung vorheriger Werte
- **Robust**: ISO-konforme Fehlerbehandlung

Unverzichtbar für:

- Dynamische Benutzeroberflächen
- Zustandsabhängige Visualisierungen
- Designflexible VT-Anwendungen

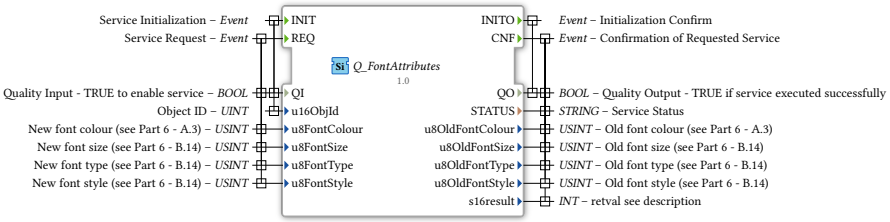
Q_FontAttributes



• — ○ • — ○ • — ○ •

Einleitung

Der **Q_FontAttributes** ist ein standardkonformer Funktionsbaustein zur Änderung von Schriftattributen in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.28) Spezifikation für VT-Systeme.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Objekt-ID)
- `REQ`: Attributänderungs-Anforderung (mit neuen Schriftattributen)

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Änderungsbestätigung (mit Status und alten Attributen)

Daten-Eingänge

- `QI` (BOOL, Standard `TRUE`): Quality-Eingang – `TRUE` aktiviert den Dienst
- `u16ObjId` (UINT): Objekt-ID (16-bit)
- `u8FontColour` (USINT): Neue Schriftfarbe
- `u8FontSize` (USINT): Neue Schriftgröße
- `u8FontType` (USINT): Neue Schriftart
- `u8FontStyle` (USINT): Neuer Schriftstil

Daten-Ausgänge

- `QO` (BOOL): Quality-Ausgang – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde
- `STATUS` (STRING): Betriebsstatusmeldung
- `u8OldFontColour` (USINT): Alte Schriftfarbe
- `u8OldFontSize` (USINT): Alte Schriftgröße
- `u8OldFontType` (USINT): Alte Schriftart
- `u8OldFontStyle` (USINT): Alter Schriftstil
- `s16result` (INT): ISO-konformer Ergebniscode

Gültige Objekt-IDs

`u16ObjId` adressiert das Zielobjekt des Kommandos **F.28** (Change Font Attributes). Gültig ist ein **Font-Attributes-Objekt**:

FontAttributesObject: 23000–23999
ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei `INIT` den Baustein.

Funktionsweise

Quality-Handling (QI/QO): Der Eingang `QI` (BOOL, Standard `TRUE`) ist mit den Ereignissen `INIT` und `REQ` verknüpft; `TRUE` aktiviert den Dienst. Der Ausgang `QO` (BOOL) ist mit `INITO` und `CNF` verknüpft und zeigt mit `TRUE` an, dass der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` mit Objekt-ID
   - `INITO` bestätigt Betriebsbereitschaft
2. **Attributänderung**:
   - `REQ` mit neuen Schriftattributen
   - Ändert Schrifteigenschaften des Objekts
   - `CNF` liefert Ergebnisstatus und vorherige Werte
3. **Fehlerbehandlung**:
   - ISO-standardisierte Fehlercodes
   - Detaillierte Statusmeldungen

Technische Besonderheiten

✓ **ISO 11783-6 konform** (F.28) ✓ **Umfassende Schriftsteuerung** (Farbe, Größe, Art, Stil) ✓ **Rückmeldung alter Werte** für Reversionsmöglichkeit ✓ **Transaktionssicher** (Atomare Ausführung)

Attribut-Eigenschaften

Feature	Beschreibung
Farbpalette	8-bit Farbindex (ISO 11783-6 A.3)
Größen	Standardisierte Schriftgrößen
Schriftarten	Vordefinierte Typen
Stile	Bitfeld für Fett/Kursiv/Unterstrichen

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreiche Änderung
−6	VT_E_OVERFLOW	Pufferüberlauf
−8	VT_E_NOACT	VT nicht bereit
−21	VT_E_NO_INSTANCE	Kein VT-Client verfügbar
−129	VT_E_ISO_INSTANCE_INVALID	Ungültige VT-Instanz
−130	VT_E_NOT_ALIVE	VT nicht aktiv

Anwendungsszenarien

- **Textdarstellung**: Dynamische Schriftanpassungen
- **Warnhinweise**: Farbliche Hervorhebungen
- **Benutzeroberflächen**: Stilwechsel
- **Barrierefreiheit**: Schriftvergrößerungen

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	Q_FontAttributes	VtTextStyle	VtFontManager
ISO-Standard	✓	✖	✖
Farbsteuerung	✓	✓	✖

Größensteuerung	✓	✖	✓
Stilsteuerung	✓	✓	✓

Fazit

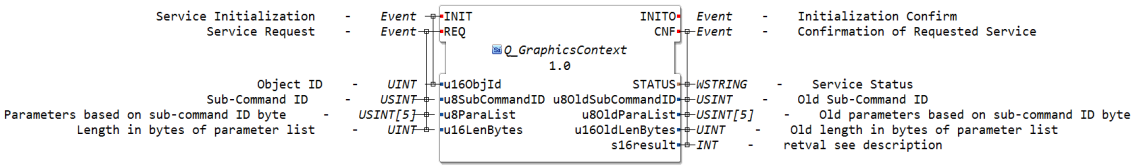
Der Q_FontAttributes-Baustein bietet die Standardimplementierung für Schriftattributänderungen:

- **Vielseitig**: Komplette Kontrolle über Schrifteigenschaften
- **Rückverfolgbar**: Rückmeldung vorheriger Werte
- **Robust**: ISO-konforme Fehlerbehandlung

Unverzichtbar für:

- Dynamische Textdarstellungen
- Barrierefreie Anwendungen
- Designflexible VT-Oberflächen

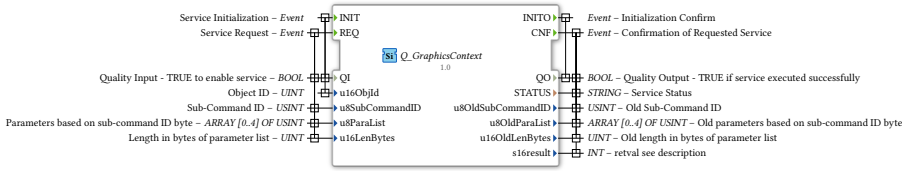
Q_GraphicsContext



• — ○ • — ○ • — ○ •

Einleitung

Der **Q_GraphicsContext** ist ein standardkonformer Funktionsbaustein zur Steuerung von Grafik-Kontextobjekten in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.56) Spezifikation für VT-Systeme ab Version 4.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Objekt-ID)
- `REQ`: Konfigurationsanforderung (mit Sub-Command und Parametern)

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Konfigurationsbestätigung (mit Status und vorherigen Werten)

Daten-Eingänge

- `QI` (BOOL, Standard `TRUE`): Quality-Eingang – `TRUE` aktiviert den Dienst
- `u16ObjId` (UINT): Grafik-Kontext Objekt-ID (16-bit)
- `u8SubCommandID` (USINT): Sub-Befehls-ID (8-bit)
- `u8ParaList` (USINT[5]): Parameterliste (5 Byte Array)
- `u16LenBytes` (UINT): Parameterlänge in Bytes

Daten-Ausgänge

- `QO` (BOOL): Quality-Ausgang – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde
- `STATUS` (STRING): Betriebsstatusmeldung
- `u8OldSubCommandID` (USINT): Vorherige Sub-Befehls-ID
- `u8OldParaList` (USINT[5]): Vorherige Parameterliste
- `u16OldLenBytes` (UINT): Vorherige Parameterlänge

- `s16result` (INT): ISO-konformer Ergebniscode

Funktionsweise

Quality-Handling (QI/QO): Der Eingang `QI` (BOOL, Standard `TRUE`) ist mit den Ereignissen `INIT` und `REQ` verknüpft; `TRUE` aktiviert den Dienst. Der Ausgang `QO` (BOOL) ist mit `INITO` und `CNF` verknüpft und zeigt mit `TRUE` an, dass der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` mit Grafik-Kontext Objekt-ID
   - `INITO` bestätigt Betriebsbereitschaft
2. **Konfiguration**:
   - `REQ` mit Sub-Befehls-ID und Parameterliste
   - Konfiguriert Grafik-Kontext Eigenschaften
   - `CNF` liefert Ergebnisstatus und vorherige Konfiguration
3. **Fehlerbehandlung**:
   - ISO-standardisierte Fehlercodes
   - Pufferüberlauf-Prüfung

Technische Besonderheiten

✓ **ISO 11783-6 konform** (F.56) ✓ **Exklusiv für VT Version 4+** ✓ **Flexible Parametrisierung** (5-Byte Parameterarray) ✓ **Rückverfolgbarkeit** (Vorherige Konfiguration)

Sub-Befehls Referenz

Die Sub-Befehls-IDs entsprechen Tabelle F.1 der ISO 11783-6 (nicht zu verwechseln mit früheren/abgeleiteten Nummerierungen):

ID	Befehl	Parameterbeschreibung
0	Set Graphics Cursor	Bytes 5-6 X, 7-8 Y (signed)
1	Move Graphics Cursor	Bytes 5-6 X-Offset, 7-8 Y-Offset (signed)
2	Set Foreground Colour	Byte 5 Farbindex
3	Set Background Colour	Byte 5 Farbindex
4	Set Line Attributes Object ID	Bytes 5-6 Objekt-ID (65535 = keine Linie)
5	Set Fill Attributes Object ID	Bytes 5-6 Objekt-ID (65535 = keine Füllung)
6	Set Font Attributes Object ID	Bytes 5-6 Objekt-ID (65535 = kein Font)

⚠️ **Variable Datenlänge / Transport Protocol:** Der Befehl F.56 hat eine **variable Datenlänge**. Nachrichten **größer als 8 Byte** werden per **Transport Protocol (TP)** übertragen; kleinere Befehle werden auf 8 Byte mit FF16 aufgefüllt. Mit der 5-Byte-Parameterliste dieses Bausteins (`u8ParaList`, ArraySize 5) entsteht eine 9-Byte-Nachricht (⟶ TP). Sub-Befehle mit mehr als 5 Parameter-Bytes (z. B. Draw Polygon mit mehreren Punkten, Pan & Zoom Viewport) lassen sich über diesen Baustein daher nicht vollständig ausdrücken.

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreiche Ausführung
−6	VT_E_OVERFLOW	Parameterpuffer zu klein
−8	VT_E_NOACT	VT nicht bereit
−21	VT_E_NO_INSTANCE	Kein VT-Client verfügbar
−129	VT_E_ISO_INSTANCE_INVALID	Ungültige VT-Instanz

−130	VT_E_NOT_ALIVE	VT nicht aktiv

Anwendungsszenarien

- **Grafikrendering**: Linien- und Füllstile
- **Benutzeroberflächen**: Transparenzeffekte
- **Diagramme**: Clip-Bereiche für Ausschnitte
- **Animationen**: Transformationsmatrizen

Sicherheitshinweise

⚠️ **Puffergröße beachten**: Parameterlisten dürfen nicht länger als `ISO_VTC_CMD_STR_MAX_LENGTH` sein
⚠️ **Sub-Command Validierung**: Unbekannte Befehls-IDs können zu unerwartetem Verhalten führen

Fazit

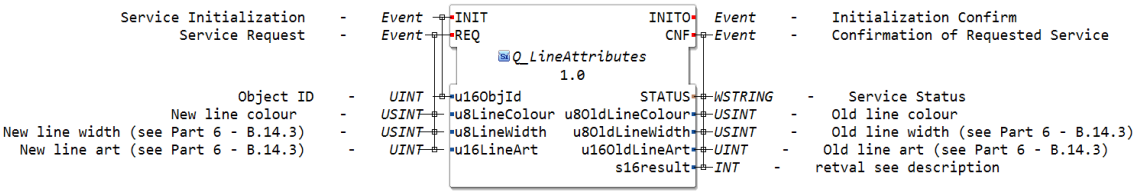
Der Q_GraphicsContext-Baustein bietet leistungsstarke Grafiksteuerung:

- **Präzise Kontrolle**: Über alle Grafikattribute
- **Rückverfolgbar**: Vorherige Zustandsicherung
- **Performant**: Optimierte Befehlslast

Essential für:

- Hochwertige Grafikdarstellungen
- Dynamische Visualisierungen
- Komplexe Benutzeroberflächen

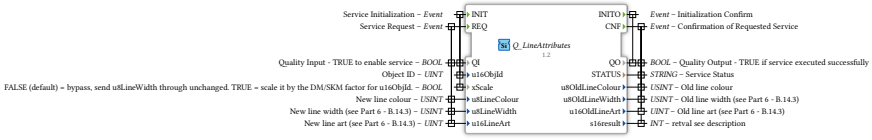
Q_LineAttributes



• — ○ • — ○ • — ○ •

Einleitung

Der **Q_LineAttributes** ist ein standardkonformer Funktionsbaustein zur Änderung von Linienattributen in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.30) Spezifikation für VT-Systeme.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Objekt-ID und `xScale`)
- `REQ`: Attributänderungs-Anforderung (mit neuen Linienattributen)

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Änderungsbestätigung (mit Status und alten Attributen)

Daten-Eingänge

- `QI` (BOOL, Standard `TRUE`): Quality-Eingang – `TRUE` aktiviert den Dienst
- `u16ObjId` (UINT): Objekt-ID (16-bit)
- `xScale` (BOOL, Standard `FALSE`): `FALSE` = Bypass, `u8LineWidth` wird unverändert gesendet; `TRUE` = Skalierung mit dem DM/SKM-Faktor für `u16ObjId`

- `u8LineColour` (USINT): Neue Linienfarbe
- `u8LineWidth` (USINT): Neue Linienbreite
- `u16LineArt` (UINT): Neuer Linienstil

Daten-Ausgänge

- `QO` (BOOL): Quality-Ausgang – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde
- `STATUS` (STRING): Betriebsstatusmeldung
- `u8OldLineColour` (USINT): Alte Linienfarbe
- `u8OldLineWidth` (USINT): Alte Linienbreite
- `u16OldLineArt` (UINT): Alter Linienstil
- `s16result` (INT): ISO-konformer Ergebniscode

Gültige Objekt-IDs

`u16ObjId` adressiert das Zielobjekt des Kommandos **F.30** (Change Line Attributes). Gültig ist ein **Line-Attributes-Objekt**:

LineAttributesObject: 24000–24999
ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei `INIT` den Baustein.

Funktionsweise

Quality-Handling (QI/QO): Der Eingang `QI` (BOOL, Standard `TRUE`) ist mit den Ereignissen `INIT` und `REQ` verknüpft; `TRUE` aktiviert den Dienst. Der Ausgang `QO` (BOOL) ist mit `INITO` und `CNF` verknüpft und zeigt mit `TRUE` an, dass der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` mit Objekt-ID
   - `INITO` bestätigt Betriebsbereitschaft
2. **Attributänderung**:
   - `REQ` mit neuen Linienattributen
   - Ändert Linieneigenschaften des Objekts
   - `CNF` liefert Ergebnisstatus und vorherige Werte
3. **Fehlerbehandlung**:
   - ISO-standardisierte Fehlercodes
   - Detaillierte Statusmeldungen

Technische Besonderheiten

✓ **ISO 11783-6 konform** (F.30) ✓ **Umfassende Liniensteuerung** (Farbe, Breite, Stil) ✓ **Rückmeldung alter Werte** für Reversionsmöglichkeit ✓ **Transaktionssicher** (Atomare Ausführung)

Attribut-Referenz

Attribut	Wertebereich	Beschreibung
Farbe	0-255 (8-bit)	Farbindex nach ISO 11783-6 A.3
Breite	1-255 Pixel	Linienstärke in Pixeln
Stil	16-bit Bitmaske	Linienart nach B.14.3

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreiche Änderung
−6	VT_E_OVERFLOW	Pufferüberlauf
−8	VT_E_NOACT	VT nicht bereit
−21	VT_E_NO_INSTANCE	Kein VT-Client verfügbar
−129	VT_E_ISO_INSTANCE_INVALID	Ungültige VT-Instanz

−130	VT_E_NOT_ALIVE	VT nicht aktiv

Anwendungsszenarien

- **Grafikdesign**: Dynamische Linienanpassungen
- **Diagramme**: Hervorhebung wichtiger Elemente
- **Benutzerführung**: Visuelle Markierungen
- **Maschinenvisualisierung**: Zustandsanzeigen

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	Q_LineAttributes	VtLineStyle	VtGraphicAttributes
ISO-Standard	✓	✖	✖
Farbsteuerung	✓	✓	✓
Breitensteuerung	✓	✓	✖
Stilsteuerung	✓	✓	✓

Fazit

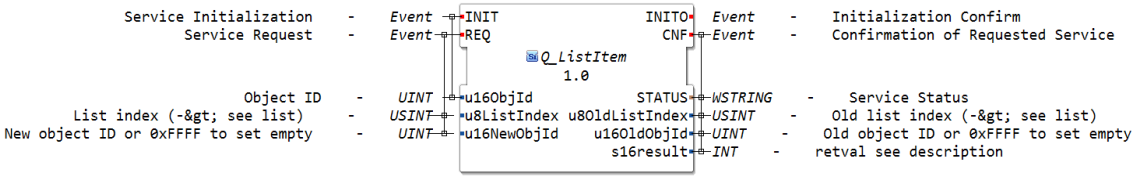
Der Q_LineAttributes-Baustein bietet präzise Kontrolle über Linienattribute:

- **Flexibel**: Vollständige Gestaltungsfreiheit
- **Zuverlässig**: ISO-konforme Implementierung
- **Rückverfolgbar**: Vorherige Zustandsspeicherung

Unverzichtbar für:

- Technische Zeichnungen
- Prozessvisualisierungen
- Interaktive Grafikanwendungen

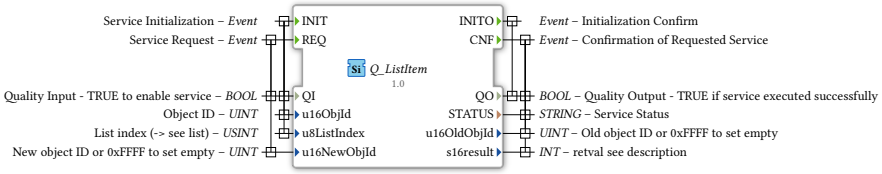
Q_ListItem



• — ○ • — ○ • — ○ •

Einleitung

Der **Q_ListItem** ist ein standardkonformer Funktionsbaustein zur Verwaltung von Listeneinträgen in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.42) Spezifikation für Listen-Input-Objekte in VT-Systemen.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Listen-Objekt-ID `u16ObjId` und Listen-Index `u8ListIndex`)
- `REQ`: Änderungsanforderung (mit neuer Objekt-ID `u16NewObjId`)

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Änderungsbestätigung (mit Status und vorheriger Objekt-ID)

Daten-Eingänge

- `QI` (BOOL, Standard `TRUE`): Quality-Eingang – `TRUE` aktiviert den Dienst
- `u16ObjId` (UINT): Listen-Objekt-ID (16-bit)
- `u8ListIndex` (USINT): Listenindex (0-basiert, bei `INIT` übergeben)
- `u16NewObjId` (UINT): Neue Objekt-ID oder 0xFFFF (leerer Eintrag)

Daten-Ausgänge

- `QO` (BOOL): Quality-Ausgang – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde
- `STATUS` (STRING): Betriebsstatusmeldung
- `u16OldObjId` (UINT): Vorherige Objekt-ID
- `s16result` (INT): ISO-konformer Ergebniscode

Instanz-Eindeutigkeit (Instance Uniqueness)

Dieser Baustein erfordert Instanz-Eindeutigkeit bezüglich des Paares (**u16ObjId**, **u8ListIndex**). Es darf im gesamten Programm nur eine Instanz von `Q_ListItem` für denselben Listeneintrag existieren. `u8ListIndex` wird bei `INIT` eingelesen — eine zweite Instanz auf dieselbe Listen-Objekt-ID und denselben Listen-Index wird bei `INIT` deaktiviert (STATUS = „This objID is already in use“). Siehe auch Instanz-Eindeutigkeit.

Gültige Objekt-IDs

`u16ObjId` (das Listenobjekt) und `u16NewObjId` (der neue Listeneintrag) unterliegen **unterschiedlichen** Regeln.

`u16ObjId` — gültige Listenobjekt-Typen (Annex F.42): Input List Field (10000–10999), Output List Object (37000–37999, VT v4+), Animation Object (44000–44999, VT v5+), External Object Definition (41000–41999, VT v5+).

`u16NewObjId` — neuer Listeneintrag: Jede beliebige Objekt-ID (der Eintrag zeigt auf das Objekt) oder 0xFFFF (ID_NULL) für einen leeren Eintrag. Die Gültigkeit der neuen Objekt-ID prüft das VT und meldet sie über F.43 zurück (Bit 2 = Invalid New List Item Object ID).

ID_NULL (65535) ist für `u16ObjId` kein gültiges Kommandoziel, deaktiviert aber bei `INIT` den Baustein.

Funktionsweise

Quality-Handling (QI/QO): Der Eingang `QI` (BOOL, Standard `TRUE`) ist mit den Ereignissen `INIT` und `REQ` verknüpft; `TRUE` aktiviert den Dienst. Der Ausgang `QO` (BOOL) ist mit `INITO` und `CNF` verknüpft und zeigt mit `TRUE` an, dass der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` mit Listen-Objekt-ID und Listen-Index
   - `INITO` bestätigt Betriebsbereitschaft
2. **Listenänderung**:
   - `REQ` mit neuer Objekt-ID
   - Aktualisiert den Listeneintrag
   - `CNF` liefert Ergebnisstatus und vorherige Objekt-ID
3. **Spezialfall**:
   - `u16NewObjId = 0xFFFF` erzeugt leeren Eintrag

Technische Besonderheiten

✓ **ISO 11783-6 konform** (F.42) ✓ **Dynamische Listenverwaltung** (Echtzeit-Änderungen) ✓ **Paar-Eindeutigkeit bei INIT** (`u16ObjId` + `u8ListIndex`) ✓ **Flexible Leerstellen** (0xFFFF-Sonderwert)

Index-Referenz

Index	Beschreibung
0	Erster Listeneintrag

...	
255	Maximaler Index (uint8_t)

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreiche Änderung
−6	VT_E_OVERFLOW	Pufferüberlauf
−8	VT_E_NOACT	VT nicht bereit
−21	VT_E_NO_INSTANCE	Kein VT-Client verfügbar
−129	VT_E_ISO_INSTANCE_INVALID	Ungültige VT-Instanz
−130	VT_E_NOT_ALIVE	VT nicht aktiv

Anwendungsszenarien

- **Dynamische Menüs**: Kontextabhängige Einträge
- **Geräteauswahl**: Plug-and-Play Gerätelisten
- **Parametereinstellungen**: Auswahloptionen
- **Diagnoselisten**: Fehlercode-Verwaltung

Fazit

Der Q_ListItem-Baustein bietet effiziente Listenverwaltung:

- **Reaktiv**: Sofortige GUI-Aktualisierungen
- **Robust**: Integrierte Fehlerbehandlung
- **Kompatibel**: Volle ISO-Konformität

Q_LockUnlockMask

• — ○ • — ○ • — ○ •

Einleitung

Der **Q_LockUnlockMask** ist ein standardkonformer Funktionsbaustein zur Steuerung des Sperrzustands von Masken in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.46) Spezifikation für VT-Systeme ab Version 4.

Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Masken-Objekt-ID `u16MaskId`)
- `REQ`: Sperr-/Entsperr-Anforderung (mit Sperrkommando und Timeout)

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Betriebsbestätigung

Daten-Eingänge

- `QI` (BOOL, Standard `TRUE`): Quality-Eingang – `TRUE` aktiviert den Dienst
- `u16MaskId` (UINT): Masken-Objekt-ID (bei `INIT` übergeben)
- `u8LockCmd` (USINT): Sperrbefehl (0=Entsperren, 1=Sperren)
- `u16LockTimeoutMs` (UINT): Timeout in ms (0=kein Timeout)

Daten-Ausgänge

- `QO` (BOOL): Quality-Ausgang – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde
- `STATUS` (STRING): Betriebsstatusmeldung
- `u8OldLockCmd` (USINT): Vorheriger Sperrzustand
- `u16OldLockTimeoutMs` (UINT): Vorheriger Timeout
- `s16result` (INT): ISO-konformer Ergebniscode

Instanz-Eindeutigkeit (Instance Uniqueness)

Dieser Baustein erfordert Instanz-Eindeutigkeit bezüglich **u16MaskId**. Es darf im gesamten Programm nur eine Instanz von `Q_LockUnlockMask` für dieselbe Masken-ID existieren. `u16MaskId` wird bei `INIT` eingelesen — eine zweite Instanz auf dieselbe Masken-ID wird bei `INIT` deaktiviert (STATUS = „This objID is already in use“). Siehe auch Instanz-Eindeutigkeit.

Gültige Objekt-IDs

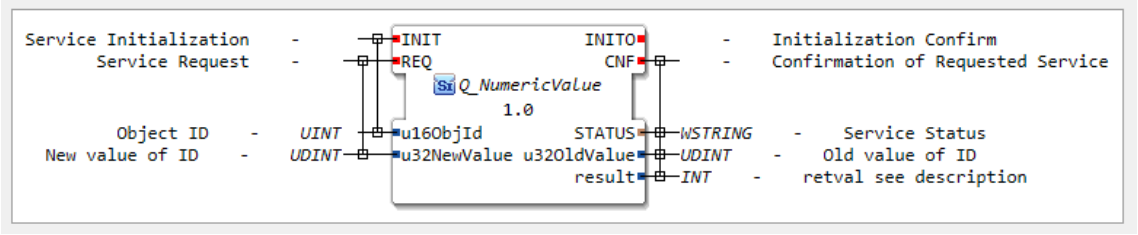
`u16MaskId` adressiert die zu sperrende/entsperrende Maske (F.46). Gültig sind:
DataMask (1000–1999) und **WindowMask / User-Layout Data Mask (34000–34999)**.
ID_NULL (65535) ist kein gültiges Kommandoziel — das Kommando wird vom VT mit einem Fehlercode beantwortet (die Maske muss der aktuell sichtbaren Maske entsprechen).

Funktionsweise

Quality-Handling (QI/QO): Der Eingang `QI` (BOOL, Standard `TRUE`) ist mit den Ereignissen `INIT` und `REQ` verknüpft; `TRUE` aktiviert den Dienst. Der Ausgang `QO` (BOOL) ist mit `INITO` und `CNF` verknüpft und zeigt mit `TRUE` an, dass der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` mit `u16MaskId`
   - `INITO` bestätigt Betriebsbereitschaft
2. **Maskensperrung**:
   - `REQ` mit Sperrkommando und Timeout
   - Steuert die Bildschirmaktualisierung der Maske
   - `CNF` liefert Betriebsstatus und vorherigen Sperrzustand/Timeout
3. **Timeout-Verhalten**:
   - Automatische Entsperrung nach Ablauf

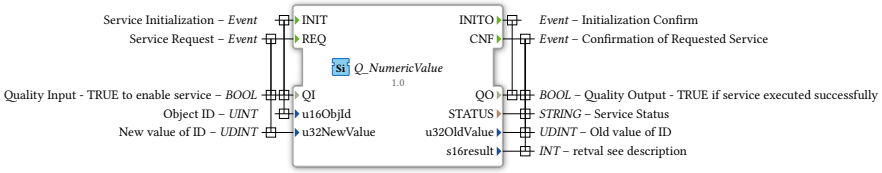
Q_NumericValue



• — ○ • — ○ • — ○ •

Einleitung

Der **Q_NumericValue** ist ein standardkonformer Funktionsbaustein zur Änderung numerischer Werte in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.22) Spezifikation für numerische VT-Objekte.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Objekt-ID)
- `REQ`: Wertänderungs-Anforderung

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Änderungsbestätigung

Daten-Eingänge

- `QI` (BOOL, Standard `TRUE`): Quality-Eingang – `TRUE` aktiviert den Dienst
- `u16ObjId` (UINT): Objekt-ID (16-bit)
- `u32NewValue` (UDINT): Neuer numerischer Wert (32-bit unsigned)

Daten-Ausgänge

- `QO` (BOOL): Quality-Ausgang – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde
- `STATUS` (STRING): Betriebsstatusmeldung
- `u32OldValue` (UDINT): Vorheriger numerischer Wert
- `s16result` (INT): ISO-konformer Ergebniscode

Gültige Objekt-IDs

`u16ObjId` — gültige Objekttypen (Anhang F.22, Objekte mit numerischem Wert-Attribut): Input Boolean Field (7000–7999), Input Number Field (9000–9999), Input List Field (10000–10999), Output Number Field (12000–12999), Meter (17000–17999), Linear Bar Graph (18000–18999), Arched Bar Graph (19000–19999), Number Variable (21000–21999), Object Pointer (27000–27999), Output List Object (37000–37999), External Object Pointer (43000–43999), Animation Object (44000–44999), Scaled Graphic Object (48000–48999).

ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei `INIT` den Baustein.

Funktionsweise

Quality-Handling (QI/QO): Der Eingang `QI` (BOOL, Standard `TRUE`) ist mit den Ereignissen `INIT` und `REQ` verknüpft; `TRUE` aktiviert den Dienst. Der Ausgang `QO` (BOOL) ist mit `INITO` und `CNF` verknüpft und zeigt mit `TRUE` an, dass der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` mit Zielobjekt-ID
   - `INITO` bestätigt Betriebsbereitschaft
2. **Wertaktualisierung**:
   - `REQ` mit neuem 32-Bit-Wert
   - Aktualisiert das numerische VT-Objekt
   - `CNF` liefert Betriebsstatus und vorherigen Wert
3. **Wertbereich**:
   - 0 bis 4.294.967.295 (32-bit unsigned)

Technische Besonderheiten

✓ **ISO 11783-6 konform** (F.22) ✓ **32-Bit-Wertebereich** (UDINT) ✓ **Sofortige Aktualisierung** ✓ **Rückverfolgbarkeit** (Vorheriger Wert) ✓ **Interne Pufferung**: Der Funktionsbaustein puffert den Wert intern. Eine Nachricht wird nur dann auf den Bus gesendet, wenn sich `u32NewValue` von `u32OldValue` unterscheidet. Dies reduziert die Buslast erheblich und verzeiht häufige REQ-Events.

Wertebereich

Parameter	Typ	Wertebereich
u32NewValue	UDINT	0 bis 4.294.967.295

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreiche Änderung
−6	VT_E_OVERFLOW	Pufferüberlauf
−8	VT_E_NOACT	VT nicht bereit
−21	VT_E_NO_INSTANCE	Kein VT-Client verfügbar
−128	VT_E_HANDLE_INVALID	Ungültige Objekt-ID
−129	VT_E_ISO_INSTANCE_INVALID	Ungültige VT-Instanz
−130	VT_E_NOT_ALIVE	VT nicht aktiv

Anwendungsszenarien

- **Prozessvisualisierung**: Echtzeit-Messwerte
- **Steuerungselemente**: Sollwertvorgaben
- **Diagnosesysteme**: Fehlercode-Anzeige
- **Produktionsdaten**: Zähler und Statistiken

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	Q_NumericValue	VtNumberUpdate	VtDataManager
ISO-Standard	✓	✖	✖
Wertebereich	32-bit	16-bit	32-bit
Rückmeldung	✓	✖	✓
Objekttyp	Numerisch	Alle	Alle

🛠️ Zugehörige Übungen

- Uebung_009
- Uebung_009a
- Uebung_011a
- Uebung_011a2
- Uebung_012
- NumericValue_ID_TO_NVS
- Uebung_012b
- Uebung_015
- Uebung_015a
- Uebung_020c2_sub
- Uebung_035
- Uebung_035b
- Uebung_035c
- Uebung_036
- Uebung_037
- Uebung_038
- Uebung_038_AX
- SINT_TO_Q_NumericValue
- Uebung_040
- Uebung_040_2

- Uebung_040_AX
- Uebung_041
- Uebung_070
- Uebung_071
- Uebung_071a
- Uebung_071b
- Uebung_072
- Uebung_072b
- Uebung_072c
- Uebung_073
- Uebung_074
- Uebung_083

Fazit

Der Q_NumericValue-Baustein bietet präzise numerische Steuerung:

- **Hochauflösend**: 32-Bit-Präzision
- **Zuverlässig**: Integrierte Fehlerprüfung
- **Flexibel**: Für alle numerischen Objekte

Essential für:

- Präzise Prozessvisualisierung
- Echtzeit-Datenmonitoring
- Industrielle Steuerungssysteme

Beispielanwendungen

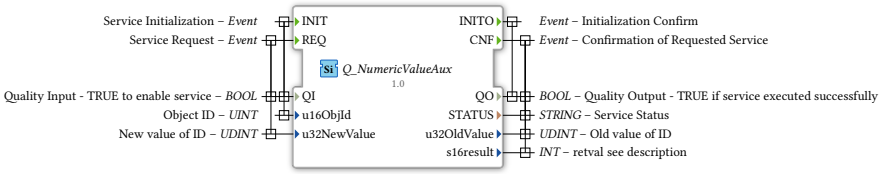
Q_NumericValue_beispiele

Q_NumericValueAux

• – ◦ • – ◦ • – ◦ •

Einleitung

Der **Q_NumericValueAux** ist ein standardkonformer Funktionsbaustein zur numerischen Wertänderung in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.22) Spezifikation für numerische VT-Objekte.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Objekt-ID)
- `REQ`: Wertänderungs-Anforderung

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Änderungsbestätigung

Daten-Eingänge

- `QI` (BOOL, Standard `TRUE`): Quality-Eingang – `TRUE` aktiviert den Dienst
- `u16ObjId` (UINT): Objekt-ID (16-bit)
- `u32NewValue` (UDINT): Neuer numerischer Wert (32-bit unsigned)

Daten-Ausgänge

- `QO` (BOOL): Quality-Ausgang – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde
- `STATUS` (STRING): Betriebsstatusmeldung

- `u32OldValue` (UDINT): Vorheriger numerischer Wert
- `s16result` (INT): ISO-konformer Ergebniscode

Gültige Objekt-IDs

`u16ObjId` — gültige Objekttypen (Annex F.22, Objekte mit numerischem Wert-Attribut): Input Boolean Field (7000–7999), Input Number Field (9000–9999), Input List Field (10000–10999), Output Number Field (12000–12999), Meter (17000–17999), Linear Bar Graph (18000–18999), Arched Bar Graph (19000–19999), Number Variable (21000–21999), Object Pointer (27000–27999), Output List Object (37000–37999), External Object Pointer (43000–43999), Animation Object (44000–44999), Scaled Graphic Object (48000–48999).

ID_NULL (65535) ist kein Befehlsziel, deaktiviert den Baustein aber bei Verwendung mit `INIT`. Jede ID außerhalb dieser Bereiche ist zum Befehlen ungültig.

Funktionsweise

Quality-Handling (QI/QO): Der Eingang `QI` (BOOL, Standard `TRUE`) ist mit den Ereignissen `INIT` und `REQ` verknüpft; `TRUE` aktiviert den Dienst. Der Ausgang `QO` (BOOL) ist mit `INITO` und `CNF` verknüpft und zeigt mit `TRUE` an, dass der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` mit Zielobjekt-ID
   - `INITO` bestätigt Betriebsbereitschaft
2. **Wertaktualisierung**:
   - `REQ` mit neuem 32-Bit-Wert
   - Aktualisiert das numerische VT-Objekt
   - `CNF` liefert Betriebsstatus und vorherigen Wert
3. **Wertbereich**:
   - 0 bis 4.294.967.295 (32-bit unsigned)

Technische Besonderheiten

✓ **ISO 11783-6 konform** (F.22) ✓ **32-Bit-Wertebereich** (UDINT) ✓ **Sofortige Aktualisierung** ✓ **Rückverfolgbarkeit** (Vorheriger Wert)

Wertebereich

Parameter	Typ	Wertebereich
u32NewValue	UDINT	0 bis 4.294.967.295

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreiche Änderung
−6	VT_E_OVERFLOW	Pufferüberlauf
−8	VT_E_NOACT	VT nicht bereit
−21	VT_E_NO_INSTANCE	Kein VT-Client verfügbar
−128	VT_E_HANDLE_INVALID	Ungültige Objekt-ID
−129	VT_E_ISO_INSTANCE_INVALID	Ungültige VT-Instanz
−130	VT_E_NOT_ALIVE	VT nicht aktiv

Anwendungsszenarien

- **Messwertanzeigen**: Echtzeit-Datenvisualisierung
- **Zählerstände**: Kilometerzähler, Produktionszähler
- **Prozesssteuerung**: Sollwertvorgaben
- **Diagnosesysteme**: Fehlercode-Anzeige

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	Q_NumericValueAux	VtNumericUpdate	VtValueManager
ISO-Standard	✓	✖	✖
Wertebereich	32-bit	16-bit	32-bit
Rückmeldung	✓	✖	✓
Objekttyp	Numerische Objekte	Alle	Alle

Fazit

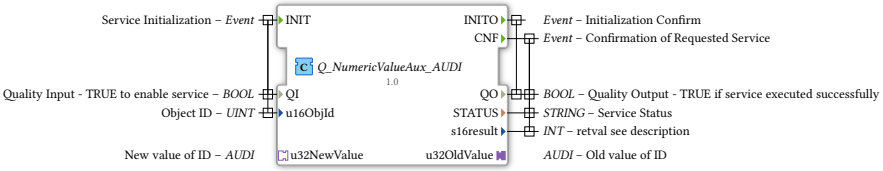
Der Q_NumericValueAux-Baustein bietet präzise numerische Steuerung:

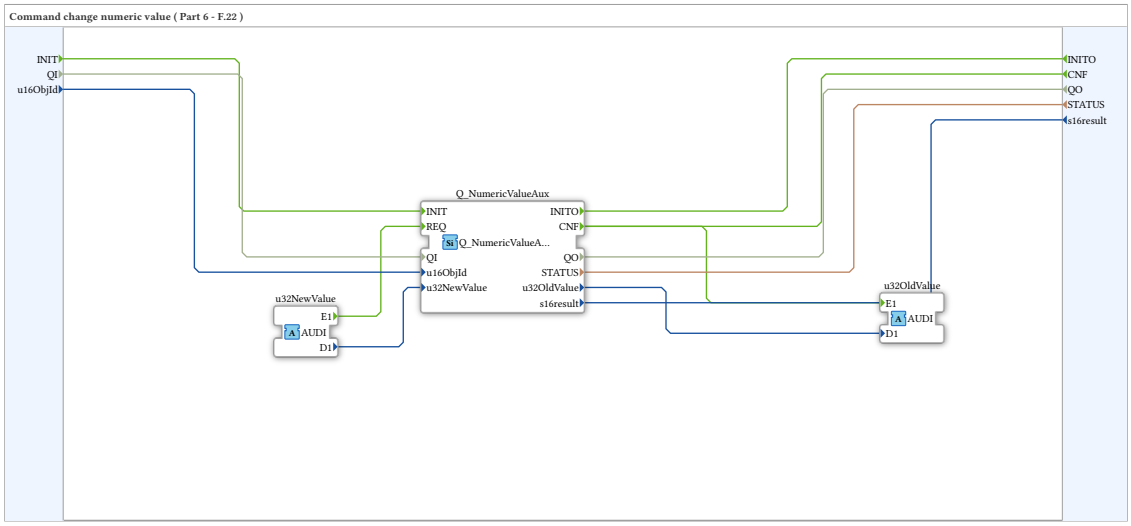
- **Hochauflösend**: 32-Bit-Wertebereich
- **Zuverlässig**: Integrierte Fehlererkennung
- **Flexibel**: Für alle numerischen Objekte

Unverzichtbar für:

- Präzise Messwertdarstellungen
- Hochauflösende Steuerungen
- Professionelle Prozessvisualisierung

Q_NumericValueAux_AUDI





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock realisiert das Kommando **Change Numeric Value** gemäß ISO 11783-6 (Part 6, F.22). Er dient dazu, den numerischen Wert eines Objekts zu ändern und eignet sich nur für Objekte, die ein numerisches Wertattribut besitzen.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar	Mitgeführte Variable
INIT	EInit	Service-Initialisierung	u16ObjId

Ereignis-Ausgänge

Ereignis	Typ	Kommentar	Mitgeführte Variable
INITO	EInit	Initialisierungsbestätigung	–
CNF	Event	Bestätigung des angeforderten Dienstes	STATUS, s16result

Daten-Eingänge

Variable	Typ	Kommentar	Initialwert
QI	BOOL	Quality Input – TRUE aktiviert den Dienst	TRUE
u16ObjId	UINT	Objekt-ID	ID_NULL

Daten-Ausgänge

Variable	Typ	Kommentar
QO	BOOL	Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde
STATUS	STRING	Dienst-Status
s16result	INT	Rückgabewert (siehe Beschreibung)

Adapter

Adapter	Richtung	Typ	Kommentar
u32OldValue	Plug	adapter::types::unidirectional::AUDI	Alter Wert der ID
u32NewValue	Socket	adapter::types::unidirectional::AUDI	Neuer Wert der ID

Qualitätssignale QI/QO: Der Dateneingang QI (BOOL, Initialwert TRUE) ist dem Ereignis INIT zugeordnet und wird an QI des internen Bausteins Q_NumericValueAux durchgereicht. Der Datenausgang QO (BOOL) wird von Q_NumericValueAux.QO übernommen und ist den Ereignissen INITO und CNF zugeordnet.

Gültige Objekt-IDs

u16ObjId ist für dieselben Objekttypen gültig wie beim umschlossenen Basisbaustein Q_NumericValueAux / Q_NumericValue (Anhang F.22, Objekte mit numerischem Wert-Attribut): Input Boolean Field (7000–7999), Input Number Field (9000–9999), Input List Field (10000–10999), Output Number Field (12000–12999), Meter (17000–17999), Linear Bar Graph (18000–18999), Arched Bar Graph (19000–19999), Number Variable (21000–21999), Object Pointer (27000–27999), Output List Object (37000–37999), External Object Pointer (43000–43999), Animation Object (44000–44999), Scaled Graphic Object (48000–48999).

ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei INIT den Baustein.

Funktionsweise

Der FB erwartet über den INIT-Eingang eine gültige Objekt-ID (u16ObjId). Nach erfolgreicher Initialisierung kann über den Socket-Adapter u32NewValue ein neuer Wert übergeben werden. Das Ereignis des Socket-Adapters (E1) löst intern eine Verarbeitung aus, bei der der alte Wert über den Plug-Adapter u32OldValue zurückgegeben wird. Das Ergebnis wird über den CNF-Ausgang mit Status und Rückgabecode (s16result) signalisiert.

Mögliche Rückgabecodes sind:

- `VT_E_NO_ERR (0)` – Erfolg
- `VT_E_OVERFLOW (-6)` – Pufferüberlauf
- `VT_E_NOACT (-8)` – Kommando im aktuellen Zustand nicht möglich
- `VT_E_NO_INSTANCE (-21)` – Kein VT-Client verfügbar
- `VT_E_ISO_INSTANCE_INVALID (-129)` – Verbindungsinstanz ungültig
- `VT_E_HANDLE_INVALID (-128)` – Ungültiger Handle
- `VT_E_NOT_ALIVE (-130)` – VT nicht erreichbar

Technische Besonderheiten

- Nutzung von unidirektionalen Adaptern für die Übermittlung alter und neuer Werte, was eine flexible, lose Kopplung in der Applikation ermöglicht.
- Initialisierung mit einer Objekt-ID (`u16ObjId`) zwingend erforderlich.
- Rückgabe von ISO-11783-spezifischen Fehlercodes, die eine detaillierte Fehlerdiagnose erlauben.
- Der Funktionsblock enthält intern einen Sub-FB (`Q_NumericValueAux`), der die eigentliche Logik kapselt.

Zustandsübersicht

Der FB besitzt keine expliziten Zustandsautomaten in der XML-Beschreibung. Die Arbeitsweise ist ereignisgesteuert:

1. **INIT** – Setzt die Objekt-ID und initialisiert die interne Logik.
2. **Warten auf Wert** – Nach INIT wird auf ein Ereignis vom Socket-Adapter (`u32NewValue.E1`) gewartet.
3. **Verarbeitung** – Beim Eintreffen wird der Wert verarbeitet, der alte Wert über den Plug-Adapter zurückgegeben und das Ergebnis über `CNF` gemeldet.

Anwendungsszenarien

- Ändern numerischer Attribute von Objekten in einem Virtual Terminal (VT) gemäß ISOBUS-Standard.
- Steuerung von Landmaschinen – z. B. Setzen von Sollwerten für Drehzahlen, Drücke oder Positionen innerhalb eines ISOBUS-Netzwerks.

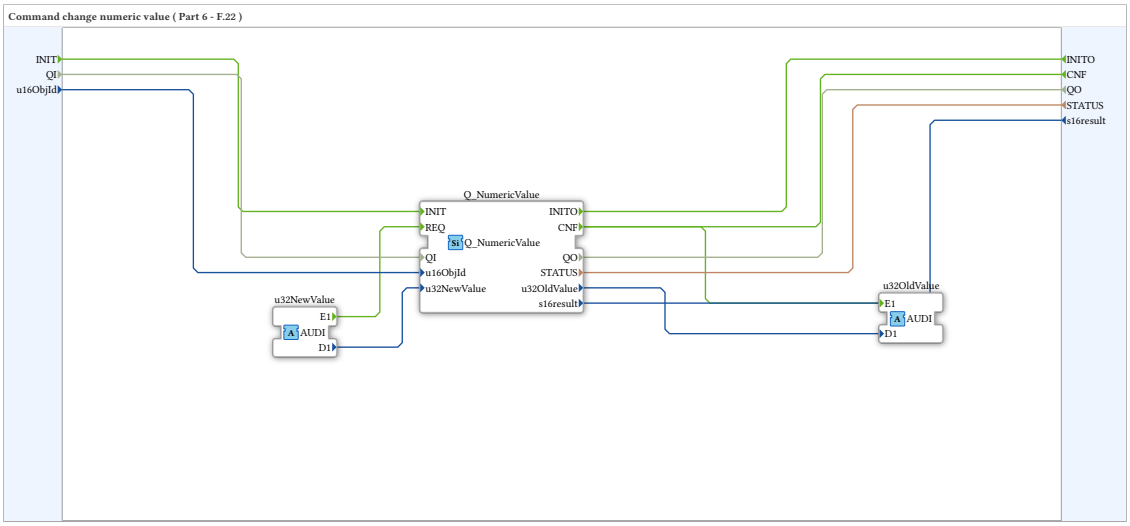
Vergleich mit ähnlichen Bausteinen

Der Basisbaustein `Q_NumericValueAux` (ohne den Zusatz `_AUDI`) bietet die gleiche Kernfunktionalität, jedoch ohne die beiden unidirektionalen Adapter. Die `_AUDI`-Variante erlaubt eine getrennte und wiederverwendbare Anbindung von Alt- und Neuwert, was den Aufbau modularer und testbarer Applikationen erleichtert.

Fazit

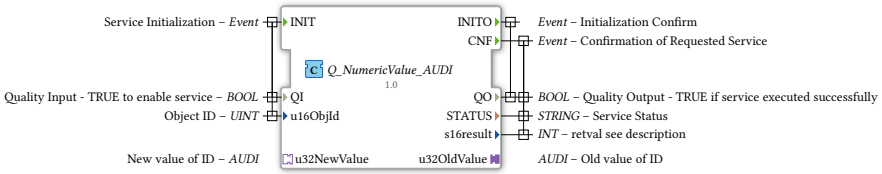
Der `Q_NumericValueAux_AUDI` ist ein spezialisierter Funktionsblock für die ISO-11783-Kommunikation. Er kapselt das Kommando „Change Numeric Value“ und stellt über Adapter eine saubere, entkoppelte Schnittstelle für die Werteübergabe bereit. Mit seiner detaillierten Fehlerrückmeldung eignet er sich besonders für sicherheitskritische Anwendungen in der Landtechnik.

Q_NumericValue_AUDI (ISOBUS Numerischer Wert - Adapter)



Einleitung

Der `Q_NumericValue_AUDI` ist ein spezieller ISOBUS-Funktionsbaustein, der es ermöglicht, numerische Werte (`UDINT`) von einer Steuerung an ein Universal Terminal (UT) zu senden, um sie dort anzuzeigen oder zu ändern. Er nutzt `AUDI`-Adapter für die Übergabe von neuen und alten Werten, was eine moderne und typsichere Anbindung gewährleistet.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: `EInit` - Initialisiert den Dienst mit der `u16ObjId`. Verknüpft mit `QI`.

Ereignis-Ausgänge

- **INITO**: `EInit` - Bestätigt die Initialisierung. Verknüpft mit `QO`.
- **CNF**: `Event` - Bestätigt die Ausführung des Dienstes. Verknüpft mit `QO`.
  - Mit Variablen `STATUS` und `s16result` verknüpft.

Input-Variablen

- **QI**: `BOOL` - Quality Input – TRUE aktiviert den Dienst. Initialwert TRUE.
- **u16ObjId**: `UINT` - Die Objekt-ID des numerischen Eingabefeldes am ISOBUS-Terminal (Standard: `ID_NULL`).

Output-Variablen

- **QO**: `BOOL` - Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde.
- **STATUS**: `STRING` - Statusmeldung des Dienstes (z.B. „OK“, „Error“).
- **s16result**: `INT` - Rückgabewert des Dienstes (Fehlercodes gemäß ISO 11783-6).

Adapter

- **u32OldValue** (Plug): `AUDI` - Der alte/aktuelle Wert des Objekts.
- **u32NewValue** (Socket): `AUDI` - Der neue Wert, der an das ISOBUS-Terminal gesendet werden soll.

Qualitätssignale QI/QO: Der Dateneingang `QI` (BOOL, Initialwert `TRUE`) ist dem Ereignis `INIT` zugeordnet und wird an `QI` des internen Bausteins `Q_NumericValue` durchgereicht. Der Datenausgang `QO` (BOOL) wird von `Q_NumericValue.QO` übernommen und ist den Ereignissen `INITO` und `CNF` zugeordnet.

Gültige Objekt-IDs

`u16ObjId` ist für dieselben Objekttypen gültig wie beim umschlossenen Basisbaustein `Q_NumericValue` (Anhang F.22, Objekte mit numerischem Wert-Attribut): Input Boolean Field (7000–7999), Input Number Field (9000–9999), Input List Field (10000–10999), Output Number Field (12000–12999), Meter (17000–17999), Linear Bar Graph (18000–18999), Arched Bar Graph (19000–19999), Number Variable (21000–21999), Object Pointer (27000–27999), Output List Object (37000–37999), External Object Pointer (43000–43999), Animation Object (44000–44999), Scaled Graphic Object (48000–48999).

ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei `INIT` den Baustein.

Funktionsweise

Der Baustein wird über das `INIT`-Ereignis mit der Objekt-ID `u16ObjId` initialisiert. Sobald ein Ereignis am `u32NewValue`-Socket eintrifft (`u32NewValue.E1`), wird der enthaltene UDINT-Wert (`u32NewValue.D1`) an das ISOBUS-Terminal gesendet, um den angezeigten numerischen Wert zu aktualisieren. Nach erfolgreicher Ausführung wird ein `CNF`-Ereignis ausgelöst, und der alte Wert des Objekts wird über den `u32OldValue`-Plug bereitgestellt.

Technische Besonderheiten

✓ **ISO 11783-6 Standard**: Konforme Implementierung für ISOBUS Universal Terminals. ✓ **Adapter-basiert**: Nutzt `AUDI`-Adapter für die numerischen Werte. ✓ **Asynchroner Betrieb**: Ereignisgesteuerte Kommunikation.

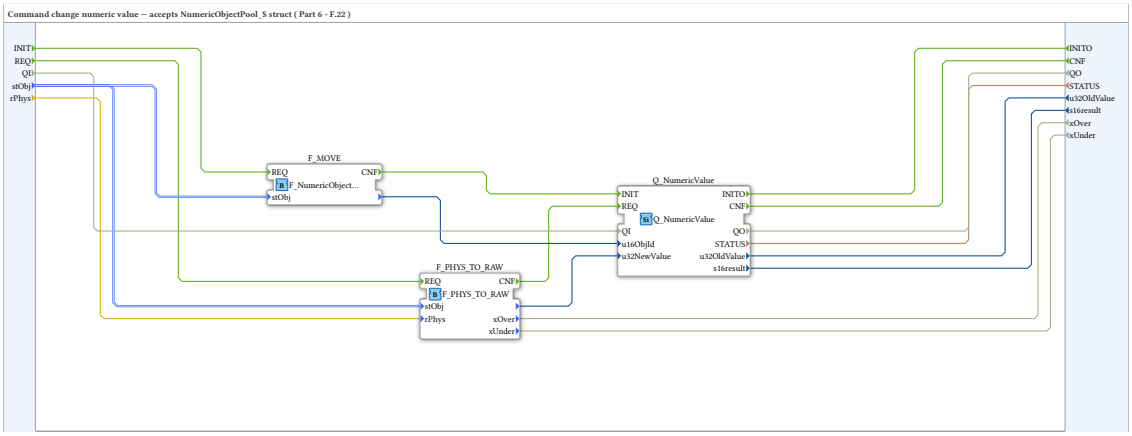
Anwendungsszenarien

- **Anzeige von Zählerständen**: Visualisierung von Zählerständen (z.B. Betriebsstunden, Stückzahlen) auf dem Terminal.
- **Sollwertvorgabe**: Übertragen von Zielwerten (z.B. Temperatur, Druck) vom ISOBUS-Terminal zur Steuerung (in Kombination mit einem NumericValue_IDA Eingangsbaustein).
- **Feedback-Systeme**: Anzeigen von Prozesswerten in Echtzeit.

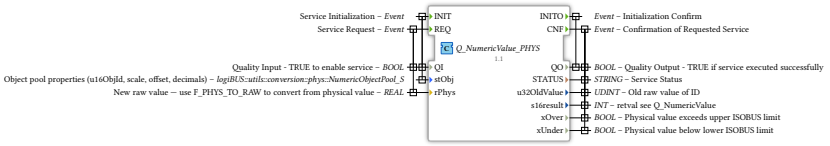
🛠️ Zugehörige Übungen

- Uebung_009_AX
- Uebung_083_AX

Q_NumericValue_PHYS



Einleitung



Der Funktionsblock `Q_NumericValue_PHYS` ist ein zusammengesetzter Baustein nach dem Standard **ISO 11783-6 (ISOBUS)**. Er dient dazu, einen numerischen Wert eines ISOBUS-Objekts durch Angabe eines physikalischen Wertes zu ändern. Die Umrechnung von physikalischen in rohe Datenwerte (Raw-Value) erfolgt automatisch anhand der in der Objektstruktur `NumericObjectPool_S` hinterlegten Skalierung, Offset und Dezimalstellen.

Der Baustein kapselt drei Unterbausteine:

- **`F_MOVE`** – Zwischenspeicherung der Objektparameter bei Initialisierung
- **`F_PHYS_TO_RAW`** – Umrechnung physikalischer Werte in rohe Ganzzahlen (UDINT)
- **`Q_NumericValue`** – eigentliche ISOBUS-Schreiboperation auf das numerische Objekt

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
`INIT`	EInit	Service-Initialisierung – lädt die Objektparameter (`stObj`) in den Baustein
`REQ`	Event	Service-Anforderung – wandelt `rPhys` um und schreibt den Wert auf das ISOBUS-Objekt

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Bestätigung der Initialisierung
CNF	Event	Bestätigung der durchgeführten Wertänderung – enthält Ergebnisdaten

Daten-Eingänge

Name	Typ	Kommentar
QI	BOOL	Quality-Eingang, TRUE aktiviert den Dienst (Standard TRUE)
stObj	ISOBUS::utils::conversion::phys::NumericObjectPtrUI_S	Struktur mit Objekt-ID, Skalierung, Offset und Dezimalstellen (Initialwert: ID_NULL, 1.0, 0, 0)
rPhys	REAL	Physikalischer Wert, der gesetzt werden soll (z.B. Temperatur, Druck)

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Quality-Ausgang, TRUE, wenn der Dienst erfolgreich ausgeführt wurde
STATUS	STRING	Statusmeldung des Dienstes (z.B. Fehler- oder Erfolgsmeldung)
u32OldValue	UDINT	Alter Rohwert des ISOBUS-Objekts vor der Änderung
s16result	INT	Rückgabewert der Schreiboperation (siehe Q_NumericValue)
xOver	BOOL	Wahr, wenn der physikalische Wert die obere Grenze des ISOBUS-Wertebereichs überschreitet
xUnder	BOOL	Wahr, wenn der physikalische Wert die untere Grenze des ISOBUS-Wertebereichs unterschreitet

Adapter

Keine Adapter vorhanden.

Gültige Objekt-IDs

stObj.u16ObjId ist für dieselben Objekttypen gültig wie beim intern verwendeten Q_NumericValue (Anhang F.22, Objekte mit numerischem Wert-Attribut): Input Boolean Field (7000–7999), Input Number Field (9000–9999), Input List Field (10000–10999), Output Number Field (12000–12999), Meter (17000–17999), Linear Bar Graph (18000–18999), Arched Bar Graph (19000–19999), Number Variable (21000–21999), Object Pointer (27000–27999), Output List Object (37000–37999), External Object Pointer (43000–43999), Animation Object (44000–44999), Scaled Graphic Object (48000–48999).

ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei INIT den Baustein.

Funktionsweise

Quality-Handling (QI/QO): Der Eingang QI (BOOL, Standard TRUE) ist mit den Ereignissen INIT und REQ verknüpft; TRUE aktiviert den Dienst. Der Ausgang QO (BOOL) ist mit INITO und CNF verknüpft und zeigt mit TRUE an, dass der Dienst erfolgreich ausgeführt wurde. QI wird an Q_NumericValue.QI weitergereicht und Q_NumericValue.QO an QO.

Die Verarbeitung erfolgt in zwei getrennten Abläufen:

1. **Initialisierung (Ereignis INIT):**
   - Der übergebene Parameter stObj wird über F_MOVE zwischengespeichert.
   - Nach Abschluss wird Q_NumericValue.INIT getriggert, wobei die Objekt-ID (u16ObjId) aus F_MOVE.OUT bereitgestellt wird.

- Die Initialisierung wird mit `INITO` quittiert.

2. **Wertänderung (Ereignis `REQ`):**
   - Der physikalische Wert `rPhys` wird zusammen mit der gespeicherten Struktur `stObj` an `F_PHYS_TO_RAW` übergeben.
   - `F_PHYS_TO_RAW` berechnet den Rohwert (`u32NewValue`) sowie die Grenzen-Flags `xOver` und `xUnder`.
   - Anschließend wird `Q_NumericValue.REQ` ausgelöst, der den berechneten Rohwert auf das ISOBUS-Objekt schreibt.
   - Die Ergebnisdaten (`STATUS`, `u32OldValue`, `s16result`) werden von `Q_NumericValue` übernommen und am Ausgang bereitgestellt.
   - Den Abschluss signalisiert das Ereignis `CNF`.

Technische Besonderheiten

- **Standardkonformität:** Der Baustein setzt die ISO 11783-6 Spezifikation (Teil 6, Anhang F.22) um – entwickelt für landwirtschaftliche ISOBUS-Anwendungen.
- **Berechnung der Grenzverletzung:** Die Flags `xOver` / `xUnder` werden bereits in der Umrechnungsstufe (`F_PHYS_TO_RAW`) ermittelt und parallel zum eigentlichen Schreibvorgang ausgegeben. So kann der Anwender frühzeitig erkennen, ob der angeforderte physikalische Wert außerhalb des zulässigen ISOBUS-Wertebereichs liegt.
- **Typisierung:** Die Objekt-ID wird als `UINT` (16-Bit) bereitgestellt, der Rohwert als `UDINT` – dies entspricht der gängigen ISOBUS-Konvention für numerische Attribute.
- **Wiederverwendung interner Bausteine:** Die Aufteilung in `F_PHYS_TO_RAW` und `Q_NumericValue` erlaubt eine modulare Testbarkeit und Wiederverwendbarkeit der Umrechnungslogik.

Zustandsübersicht

Der Funktionsblock besitzt keinen expliziten Einstiegs-Zustandsautomaten, sondern realisiert seine Funktionalität als Datenfluss-Netzwerk. Der Ablauf ist strikt sequenziell:

- **Initialisierungsphase:**
  `INIT` ⟶ `F_MOVE.REQ` ⟶ `F_MOVE.CNF` ⟶ `Q_NumericValue.INIT` ⟶ `Q_NumericValue.INITO` ⟶ `INITO`
- **Betriebsphase (Schreiben):**
  `REQ` ⟶ `F_PHYS_TO_RAW.REQ` ⟶ `F_PHYS_TO_RAW.CNF` ⟶ `Q_NumericValue.REQ` ⟶ `Q_NumericValue.CNF` ⟶ `CNF`

Während der Ausführung eines Durchlaufs ist der Baustein nicht für neue Ereignisse vorbereitet – es muss das jeweilige Bestätigungssignal (`INITO` oder `CNF`) abgewartet werden.

Anwendungsszenarien

- **Fahrzeugterminal mit ISOBUS-Anbindung:** Einstellen von Sollwerten (z.B. Arbeitshöhe, Dosierrate) über eine Benutzereingabe in physikalischen Einheiten (m, kg/h, °C).
- **Fernsteuerung landwirtschaftlicher Geräte:** Senden physikalischer Werte von einer Steuerung an ein ISOBUS-Gerät (z.B. Sämaschine) ohne manuelle Rohwert-Umrechnung.
- **Automatisierte Kalibrierung:** Nachführen von Parametern während des Betriebs, wobei die Skalierung und der Offset aus einer Konfigurations-Struktur (`NumericObjectPool_S`) stammen.

Vergleich mit ähnlichen Bausteinen

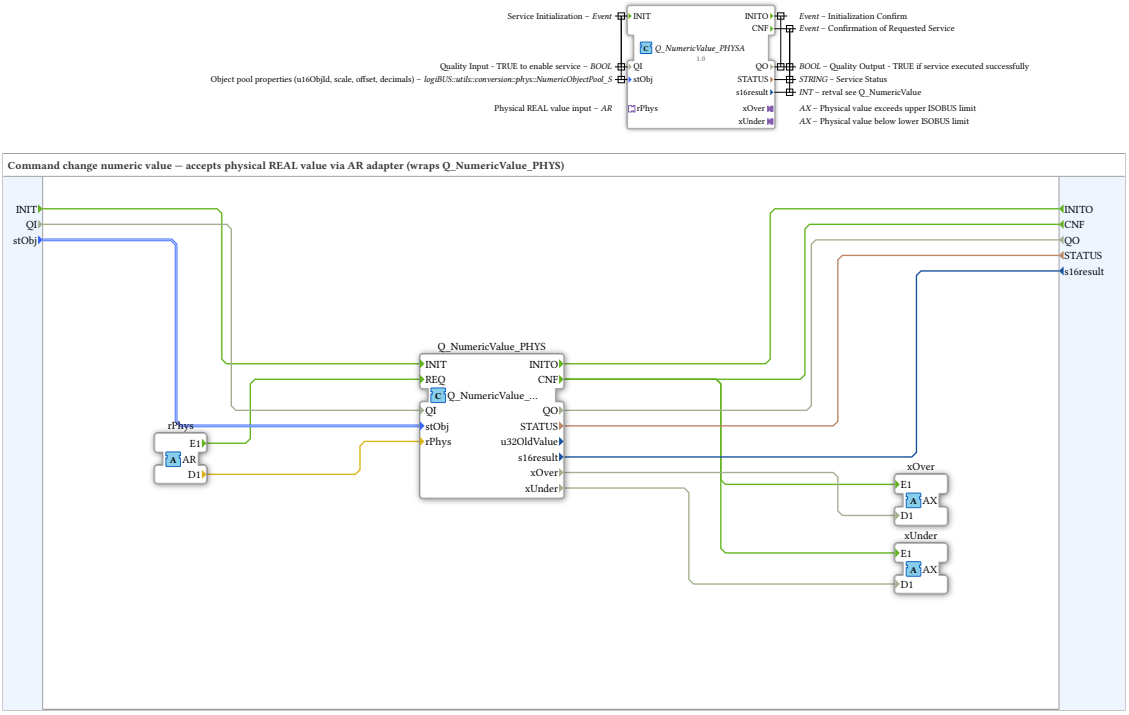
Baustein	Funktion	Unterschied zu `Q_NumericValue_PHYS`
`Q_NumericValue` (aus `isobus::UT::Q`)	Schreiben eines rohen (bereits umgerechneten) Werts	Erwartet `u32NewValue` direkt – ohne physikalische Umrechnung und ohne Grenzprüfung
`F_PHYS_TO_RAW`	Reine Umrechnung physikalisch ⟶ roh	Liefert nur `xOver`, `xUnder` und den Rohwert – keine ISOBUS-Kommunikation
`Q_NumericValue_PHYS`	Kombinierte Umrechnung + Schreibzugriff	Bietet eine vollständige Schnittstelle für physikalische Werte in einem Schritt

Der vorliegende Baustein vereinfacht die Anwendung, da der Anwender keinen separaten Umrechnungsschritt programmieren muss. Er eignet sich besonders für Steuerungen, die Werte in gebräuchlichen Einheiten verarbeiten.

Fazit

`Q_NumericValue_PHYS` ist ein praxisorientierter, standardkonformer Funktionsblock für die ISOBUS-Kommunikation. Er verbindet die Umrechnung physikalischer Werte mit dem Schreibzugriff auf numerische Objekte und reduziert so den Entwicklungsaufwand in der Landtechnik-Automation. Durch die klare Trennung der Teilaufgaben (Speicherung, Umrechnung, Buszugriff) bleibt der Baustein wartbar und testbar. Die Ausgabe der Grenzverletzungs-Flags ermöglicht eine robuste Fehlerbehandlung im Anwendungscode.

Q_NumericValue_PHYSA



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `Q_NumericValue_PHYSA` dient als **Wrapper** für den Baustein `Q_NumericValue_PHYS`. Er ermöglicht das Setzen eines numerischen Werts, der als physischer `REAL`-Wert über einen **AR-Adapter** (Socket `rPhys`) empfangen wird. Der FB ist nach dem Standard ISO 11783-6 ausgelegt und eignet sich für ISOBUS-Anwendungen, die eine physikalische Wertvorgabe benötigen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung
`INIT`	`EInit`	Service-Initialisierung; wird mit dem Daten-Eingang `stObj` ausgeführt

Ereignis-Ausgänge

Name	Typ	Beschreibung
`INITO`	`EInit`	Bestätigung der erfolgreichen Initialisierung

CNF	Event	Bestätigung der durchgeführten Wertänderung; ausgegeben zusammen mit STATUS und s16result

Daten-Eingänge

Name	Typ	Beschreibung
QI	BOOL	Quality Input – TRUE aktiviert den Dienst
stObj	isobus::utils::conversion::phys::NumericObjectPool_t	Objektpool-Eigenschaften: u16ObjId (Objekt-ID), r32Scale (Skalierung), i32Offset (Offset), u8Decimals (Dezimalstellen). Standardwert: (u16ObjId := ID_NULL, r32Scale := 1.0, i32Offset := 0, u8Decimals := 0)

Daten-Ausgänge

Name	Typ	Beschreibung
QO	BOOL	Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde
STATUS	STRING	Statusmeldung des Dienstes
s16result	INT	Rückgabewert (siehe Dokumentation von Q_NumericValue)

Adapter

Typ	Name	Richtung	Beschreibung
adapter::types::unidirectional::AR	rPhys	Socket (Eingang)	Empfängt den physikalischen REAL-Wert zur Verarbeitung
adapter::types::unidirectional::AX	xOver	Plug (Ausgang)	Signalisiert, dass der physikalische Wert den oberen ISOBUS-Grenzwert überschreitet
adapter::types::unidirectional::AX	xUnder	Plug (Ausgang)	Signalisiert, dass der physikalische Wert den unteren ISOBUS-Grenzwert unterschreitet

Qualitätssignale QI/QO: Der Dateneingang QI (BOOL, Initialwert TRUE) ist dem Ereignis INIT zugeordnet und wird an QI des internen Bausteins Q_NumericValue_PHYS durchgereicht. Der Datenausgang QO (BOOL) wird von Q_NumericValue_PHYS.QO übernommen und ist den Ereignissen INITO und CNF zugeordnet.

Gültige Objekt-IDs

stObj.u16ObjId ist für dieselben Objekttypen gültig wie beim umschlossenen Basisbaustein Q_NumericValue_PHYS / Q_NumericValue (Anhang F.22, Objekte mit numerischem Wert-Attribut): Input Boolean Field (7000–7999), Input Number Field (9000–9999), Input List Field (10000–10999), Output Number Field (12000–12999), Meter (17000–17999), Linear Bar Graph (18000–18999), Arched Bar Graph (19000–19999), Number Variable (21000–21999), Object Pointer (27000–27999), Output List Object (37000–37999), External Object Pointer (43000–43999), Animation Object (44000–44999), Scaled Graphic Object (48000–48999).

ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei INIT den Baustein.

Funktionsweise

1. Nach dem **INIT**-Ereignis wird der FB mit den in stObj definierten Objektpool-Eigenschaften initialisiert.
2. Sobald ein neuer physischer Wert über den **rPhys**-Adapter (Ereignis E1 des Sockets) eintrifft, wird dieser intern an den gebundenen **Q_NumericValue_PHYS**-Baustein weitergeleitet.

3. Der `Q_NumericValue_PHYS` verarbeitet den Wert (unter Berücksichtigung von Skalierung, Offset und Dezimalstellen) und löst das **CNF**-Ereignis aus.
4. Über das **CNF**-Ereignis werden gleichzeitig der Status (`STATUS`), der Rückgabewert (`s16result`) sowie die Grenzwertindikatoren `xOver` und `xUnder` ausgegeben.
5. Die Ausgänge `xOver` und `xUnder` werden als AX-Adapter bereitgestellt, um übergeordneten Logiken mitzuteilen, ob der eingegebene Wert außerhalb des zulässigen ISOBUS-Bereichs liegt.

Technische Besonderheiten

- Der FB ist eine **reine Adapter-Wrapper-Komponente**. Die eigentliche Logik liegt im intern verwendeten `Q_NumericValue_PHYS`.
- Die Parameter für Skalierung und Offset werden über die **Struktur `NumericObjectPool_S`** konfiguriert – dies erlaubt eine flexible Anpassung an verschiedene physikalische Einheiten.
- Die Kommunikation erfolgt **ereignisgesteuert** über die Adapter `AR` (Wert-Eingang) und `AX` (Grenzsignal-Ausgabe). Dies ermöglicht eine modulare Einbindung in bestehende ISOBUS-Kommunikationsabläufe.

Zustandsübersicht

Der FB selbst besitzt keine explizite Zustandsmaschine. Der initialisierte Zustand wird durch das erste `INIT`-Ereignis hergestellt. Nachfolgende Wertänderungen durch `rPhys.E1` führen direkt zur Verarbeitung und Ausgabe. Fehlerzustände werden über den Ausgang `STATUS` kommuniziert.

Anwendungsszenarien

- **ISOBUS-APP-Steuerung:** Setzen eines numerischen Werts (z. B. Sollwert für Maschinenparameter) aus einem physikalischen Sensorwert, der über einen Adapter angebunden ist.
- **Wandlung von REAL auf ISOBUS-Format:** Der Baustein übernimmt die Umrechnung von physikalischen Werten auf das interne Ganzzahl-Format unter Verwendung von Skalierung und Offset.
- **Grenzwertüberwachung:** Durch die Ausgänge `xOver` und `xUnder` kann die übergeordnete Steuerung auf Über- oder Unterschreitungen reagieren.

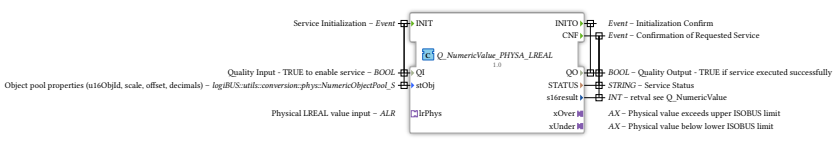
Vergleich mit ähnlichen Bausteinen

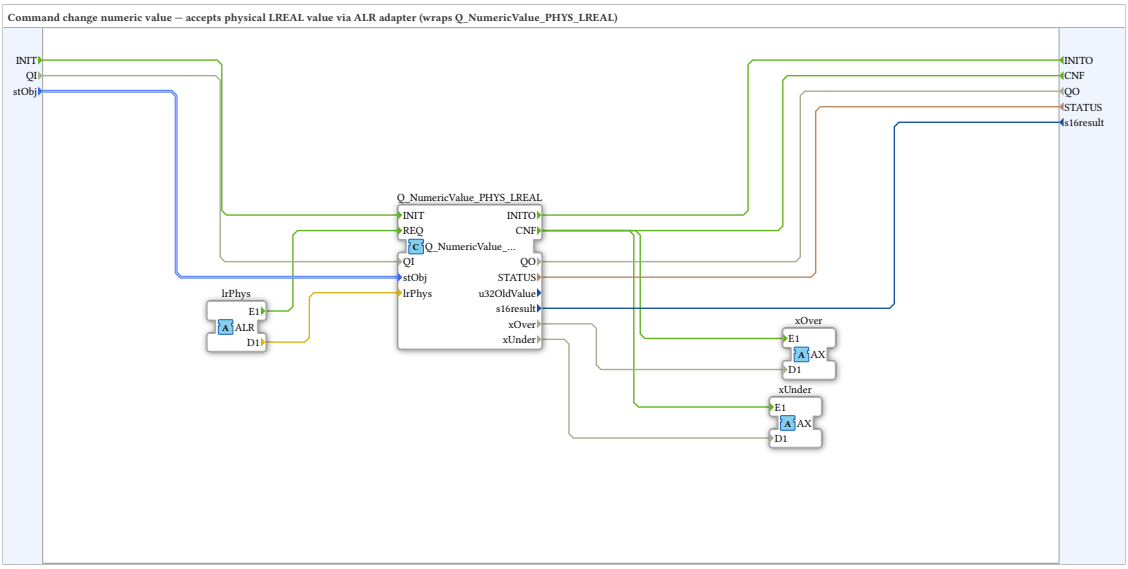
Baustein	Beschreibung	Unterschied
`Q_NumericValue_PHYS`	Direkter FB für physikalische Werte	`Q_NumericValue_PHYSA` wrappt diesen FB und fügt explizite Adapterausgänge (`xOver`, `xUnder`) für Grenzsignale hinzu
`Q_NumericValue`	Basis-FB für numerische Werte (keine physikalische Umrechnung)	`Q_NumericValue_PHYSA` ist speziell für physikalische REAL-Werte ausgelegt und enthält Skalierung/Offset

Fazit

Der `Q_NumericValue_PHYSA` vereinfacht die Integration von physikalischen Werten in ISOBUS-Systeme, indem er die Adapter-Kommunikation kapselt und Grenzwerte direkt signalisiert. Durch die Wiederverwendung des erprobten `Q_NumericValue_PHYS` bleibt die Logik robust und standardkonform.

Q_NumericValue_PHYSA_LREAL





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **Q_NumericValue_PHYSA_LREAL** dient als Kommando zum Ändern eines numerischen Werts im ISOBUS-Kontext (ISO 11783-6). Er nimmt einen physikalischen LREAL-Wert über den **ALR**-Adapter entgegen und wickelt die eigentliche Verarbeitung über den eingebetteten Baustein **Q_NumericValue_PHYS_LREAL** ab. Der FB nutzt Adapter-Schnittstellen für die physikalische Werteingabe sowie für die Signalisierung von Über- und Unterschreitungen der zulässigen Bereichsgrenzen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Initialisierung des Bausteins (mit stObj).

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Bestätigung der erfolgreichen Initialisierung.
CNF	Event	Bestätigung der angeforderten Wertänderung (enthält STATUS und s16result).

Daten-Eingänge

Name	Typ	Kommentar
QI	BOOL	Quality Input – TRUE aktiviert den Dienst
stObj	logiBUS::utils::conversion::phys::NumericObjectPool_S	Eigenschaften des Objektpools (ObjID, Skalierung, Offset, Dezimalstellen). Initial: (u16ObjId := ID_NULL, r32Scale := 1.0, i32Offset := 0, u8Decimals := 0).

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde
STATUS	STRING	Statusmeldung des Dienstes.

s16result	INT	Rückgabewert – siehe Q_NumericValue (Ergebnis der Wertänderungsanfrage).

Adapter

Sockets (Eingangsadapter)

Name	Typ	Kommentar
lrPhys	ALR (unidirectional)	Eingang für den physikalischen LREAL-Wert (wird über E1 getriggert).

Plugs (Ausgangsadapter)

Name	Typ	Kommentar
xOver	AX (unidirectional)	Signalisiert, dass der physikalische Wert die obere ISOBUS-Grenze überschreitet.
xUnder	AX (unidirectional)	Signalisiert, dass der physikalische Wert die untere ISOBUS-Grenze unterschreitet.

Qualitätssignale QI/QO: Der Dateneingang `QI` (BOOL, Initialwert `TRUE`) ist dem Ereignis `INIT` zugeordnet und wird an `QI` des internen Bausteins `Q_NumericValue_PHYS_LREAL` durchgereicht. Der Datenausgang `QO` (BOOL) wird von `Q_NumericValue_PHYS_LREAL.QO` übernommen und ist den Ereignissen `INITO` und `CNF` zugeordnet.

Gültige Objekt-IDs

`stObj.u16ObjId` ist für dieselben Objekttypen gültig wie beim umschlossenen Basisbaustein `Q_NumericValue_PHYS_LREAL` / `Q_NumericValue` (Anhang F.22, Objekte mit numerischem Wert-Attribut): Input Boolean Field (7000–7999), Input Number Field (9000–9999), Input List Field (10000–10999), Output Number Field (12000–12999), Meter (17000–17999), Linear Bar Graph (18000–18999), Arched Bar Graph (19000–19999), Number Variable (21000–21999), Object Pointer (27000–27999), Output List Object (37000–37999), External Object Pointer (43000–43999), Animation Object (44000–44999), Scaled Graphic Object (48000–48999).

ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei `INIT` den Baustein.

Funktionsweise

1. **Initialisierung** (`INIT`): Der Baustein wird mit den Objektpool-Eigenschaften (`stObj`) initialisiert. Das Ereignis wird direkt an den inneren Baustein `Q_NumericValue_PHYS_LREAL` weitergeleitet. Bei erfolgreicher Initialisierung wird `INITO` ausgelöst.
2. **Wertänderung**: Sobald am Adapter `lrPhys` das Ereignis `E1` eintrifft, wird der physikalische LREAL-Wert über die Datenverbindung `lrPhys.D1` an den inneren Baustein übergeben und das Ereignis `REQ` ausgelöst.
3. **Ergebnis**: Der innere Baustein führt die eigentliche Verarbeitung durch. Über `CNF` werden der Status (`STATUS`) und der Rückgabewert (`s16result`) ausgegeben. Gleichzeitig kann über die Plugs `xUnder` und `xOver` signalisiert werden, ob der eingehende Wert die definierten ISOBUS-Grenzen verletzt.
4. Die Ausgabeevents des inneren Bausteins (`INITO`, `CNF`, `xUnder.E1`, `xOver.E1`) werden direkt an die entsprechenden Ausgänge des `Q_NumericValue_PHYSA_LREAL` weitergeleitet.

Technische Besonderheiten

- **Wrapper-Design**: Der Baustein kapselt den Funktionsblock `Q_NumericValue_PHYS_LREAL` und stellt über Adapter eine komfortablere, physikalische Schnittstelle bereit.
- **Adapterbasierte Kommunikation**: Die Werteingabe und Grenzsignalisierung erfolgt über unidirektionale Adapter (`ALR`, `AX`), was die Kapselung und Wiederverwendung in unterschiedlichen Kontexten erleichtert.
- **Konfigurierbare Objektpool-Parameter**: Über `stObj` werden Objekt-ID, Skalierung, Offset und Dezimalstellen festgelegt, sodass der Block flexibel auf verschiedene Sensor- oder Aktorwerte angepasst werden kann.
- **Standardkonformität**: Der FB ist gemäß ISO 11783-6 (ISOBUS) entwickelt und für landwirtschaftliche Anwendungen optimiert.

Zustandsübersicht

Eine explizite Zustandsmaschine ist im XML nicht abgebildet, da der FB als reines Netzwerk aus einem inneren Baustein realisiert ist. Das typische Verhalten folgt jedoch einem einfachen Ablauf:

- **Idle** – wartet auf INIT.
- **Initialisiert** – nach INIT und vor ersten Wertänderungen.
- **Verarbeitung** – nach E1-Trigger (REQ) bis zum Empfang des CNF-Events.
- **Abschluss** – Ausgabe von STATUS/s16result über CNF, gefolgt von erneuter Bereitschaft.

Anwendungsszenarien

- **ISOBUS-Kommando**: Ändern eines Sollwerts (z.B. Drehzahl, Position, Druck) in einem landwirtschaftlichen Steuergerät, wobei der Wert als physikalische Größe (LREAL) vorliegt.
- **Grenzwertüberwachung**: Einsatz in Anwendungen, die bei Überschreitung der zulässigen Bereichsgrenzen (z.B. bei Sensordaten) zusätzliche Alarme oder Reaktionen auslösen müssen (über `xOver` / `xUnder`).
- **Skalierte Werte**: Verwendung von Skalierungs- und Offsetparametern, um rohe ISOBUS-Werte in benutzerfreundliche physikalische Einheiten umzurechnen.

Vergleich mit ähnlichen Bausteinen

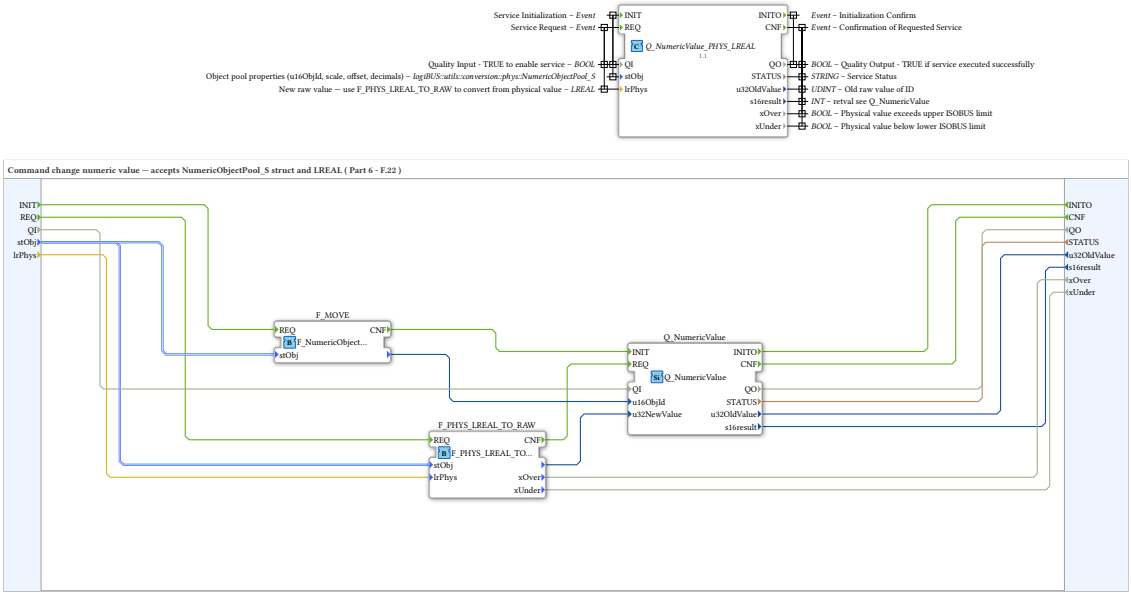
Im direkten Vergleich zum inneren Baustein **Q_NumericValue_PHYS_LREAL** bietet der **Q_NumericValue_PHYSA_LREAL** eine höhere Abstraktionsebene:

- Statt direkter Ein-/Ausgangssignale wird die Kommunikation über Adapter realisiert – dies erlaubt eine lose Kopplung in modularen Steuerungsarchitekturen.
- Der äußere FB fügt keine eigenständige Logik hinzu, sondern vereinfacht die Einbindung durch die adapterbasierte Schnittstelle (z.B. `ALR`-Eingang).
- Anders als reine Funktionsblöcke mit festen Datenports kann `Q_NumericValue_PHYSA_LREAL` flexibel an verschiedene Umgebungen angepasst werden, ohne die Signalwege im Netzwerk ändern zu müssen.

Fazit

Der **Q_NumericValue_PHYSA_LREAL** ist ein praktischer Wrapper-Baustein, der die physikalische Wertänderung im ISOBUS-Kontext über Adapter vereinfacht. Er kombiniert die bewährte Logik von `Q_NumericValue_PHYS_LREAL` mit einer komfortablen, adapterbasierten Schnittstelle für LREAL-Eingänge und Grenzsignalisierung. Durch die konfigurierbaren Objektpool-Parameter und die Einhaltung von ISO 11783-6 eignet er sich besonders für den Einsatz in modularen landwirtschaftlichen Steuerungssystemen.

Q_NumericValue_PHYS_LREAL



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **Q_NumericValue_PHYS_LREAL** dient dazu, über ISOBUS (ISO 11783-6) einen numerischen Wert als physikalische Größe zu setzen. Er nimmt einen physikalischen Wert vom Typ `LREAL` entgegen, konvertiert ihn automatisch in den erforderlichen Rohwert und sendet den entsprechenden Befehl an das angeschlossene Gerät. Dies entspricht der Spezifikation in Teil 6, Anhang F.22.

Der Baustein kapselt die notwendigen Schritte der physikalischen Umrechnung und der eigentlichen Befehlsausführung, sodass der Anwender direkt mit physikalischen Einheiten arbeiten kann.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung
`INIT`	Initialisiert den Baustein mit den Objekt-Pool-Eigenschaften (`stObj`).
`REQ`	Startet die Verarbeitung: der physikalische Wert (`lrPhys`) wird an das Zielobjekt gesendet.

Ereignis-Ausgänge

Ereignis	Beschreibung
`INITO`	Quittiert die erfolgreiche Initialisierung.
`CNF`	Quittiert die Ausführung des Befehls; die Ausgangsdaten sind gültig.

Daten-Eingänge

Name	Typ	Beschreibung
`QI`	BOOL	Quality-Eingang, `TRUE` aktiviert den Dienst (Standard `TRUE`)
`stObj`	`tc_ISOBUS::utils::conversion::phys::NumericObjectPool_S`	Objekt-Pool-Eigenschaften (Objekt-ID, Skalierung, Offset, Dezimalstellen). Standardwert: `(u16ObjId := ID_NULL, r32Scale := 1.0, i32Offset := 0, u8Decimals := 0)`.
`lrPhys`	`LREAL`	Der physikalische Wert (z. B. Druck, Temperatur), der gesendet werden soll. Hinweis: Vor der Übergabe sollte der Wert ggf. mit `F_PHYS_LREAL_TO_RAW` umgerechnet werden; der Baustein führt dies intern automatisch durch.

Daten-Ausgänge

Name	Typ	Beschreibung
`QO`	BOOL	Quality-Ausgang, `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde
`STATUS`	`STRING`	Statusmeldung des durchgeführten Dienstes.
`u32OldValue`	`UDINT`	Alter Rohwert des Objekts vor der Änderung.
`s16result`	`INT`	Rückgabewert (siehe `Q_NumericValue`).
`xOver`	`BOOL`	`TRUE`, wenn der physikalische Wert den oberen ISOBUS-Grenzwert überschreitet.
`xUnder`	`BOOL`	`TRUE`, wenn der physikalische Wert den unteren ISOBUS-Grenzwert unterschreitet.

Adapter

Keine.

Gültige Objekt-IDs

`stObj.u16ObjId` ist für dieselben Objekttypen gültig wie beim intern verwendeten `Q_NumericValue` (Anhang F.22, Objekte mit numerischem Wert-Attribut): Input Boolean Field (7000–7999), Input Number Field (9000–9999), Input List Field (10000–10999), Output Number Field (12000–12999), Meter (17000–17999), Linear Bar Graph (18000–18999), Arched Bar Graph (19000–19999), Number Variable (21000–21999), Object Pointer (27000–27999), Output List Object (37000–37999), External Object Pointer (43000–43999), Animation Object (44000–44999), Scaled Graphic Object (48000–48999).

ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei `INIT` den Baustein.

Funktionsweise

Quality-Handling (QI/QO): Der Eingang `QI` (BOOL, Standard `TRUE`) ist mit den Ereignissen `INIT` und `REQ` verknüpft; `TRUE` aktiviert den Dienst. Der Ausgang `QO` (BOOL) ist mit `INITO` und `CNF` verknüpft und zeigt mit `TRUE` an, dass der Dienst erfolgreich ausgeführt wurde. `QI` wird an `Q_NumericValue.QI` weitergereicht und `Q_NumericValue.QO` an `QO`.

Der Baustein arbeitet intern mit drei untergeordneten Funktionsblöcken zusammen:

1. **F_MOVE** – Bei einem `INIT`-Ereignis wird die übergebene Struktur `stObj` (Objekt-ID, Skalierung, Offset, Dezimalstellen) in den internen `Q_NumericValue` kopiert.
2. **F_PHYS_LREAL_TO_RAW** – Bei einem `REQ`-Ereignis wird der physikalische Wert `lrPhys` unter Verwendung der in `stObj` hinterlegten Parameter in einen ISOBUS-Rohwert umgerechnet.
3. **Q_NumericValue** – Der erzeugte Rohwert wird anschließend zusammen mit der Objekt-ID an den eigentlichen Befehlsbaustein übergeben, der den Befehl auf dem Bus ausführt.

Die Ausgänge `STATUS`, `u32OldValue` und `s16result` stammen direkt von `Q_NumericValue`. Die Über-/Unterlauf-Meldungen (`xOver`, `xUnder`) werden von der Umrechnungsfunktion bereitgestellt.

Technische Besonderheiten

- **Automatische Umrechnung**: Der Anwender muss physikalische Werte nicht manuell in Rohwerte konvertieren; dies erfolgt transparent im Inneren des Bausteins.
- **Standardwerte**: Falls keine spezifischen Objekt-Pool-Eigenschaften übergeben werden, verwendet der Baustein sinnvolle Vorgaben (Skalierung 1.0, Offset 0, keine Dezimalstellen). Die Objekt-ID ist dann `ID_NULL`.
- **Grenzwertprüfung**: Die Ausgänge `xOver` und `xUnder` signalisieren, ob der übergebene physikalische Wert außerhalb des für ISOBUS zulässigen Bereichs liegt. So können Anwendungen frühzeitig reagieren.
- **Zwischenspeicherung**: Die Initialisierung (`INIT`) kopiert die Objekt-Eigenschaften nur einmal und bewahrt sie für spätere `REQ`-Aufrufe auf.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten im Sinne einer ECC, sondern arbeitet ereignisgesteuert nach folgender Logik:

Zustand / Ablauf	Beschreibung
Initialisierung	Nach einem `INIT`-Ereignis werden die Objekt-Eigenschaften intern gespeichert. Anschließend wird `INITO` gesendet.
Befehl senden	Nach einem `REQ`-Ereignis wird der physikalische Wert umgerechnet, der Befehl abgesetzt und nach Abschluss `CNF` mit den Ergebnisdaten gesendet.
Fehlerbehandlung	Tritt bei der Umrechnung ein Über-/Unterlauf auf, werden `xOver` bzw. `xUnder` bereits vor dem Absetzen des Befehls gesetzt. Ein fehlerhafter Befehl wird durch `s16result` und die Statusmeldung signalisiert.

Anwendungsszenarien

- **Steuerung von ISOBUS-Geräten**: Setzen von Sollwerten für Aktoren (z. B. Ventile, Antriebe) in physikalischen Einheiten wie Druck (bar, kPa), Temperatur (°C) oder Füllstand.

- **Landwirtschaftliche Maschinen**: Ändern von Arbeitsparametern (z. B. Ausbringmenge, Geschwindigkeit) direkt aus einer Steuerungsapplikation.
- **Test- und Simulationsumgebungen**: Einfaches Senden von physikalischen Werten, ohne sich um die Rohwertkonvertierung kümmern zu müssen.

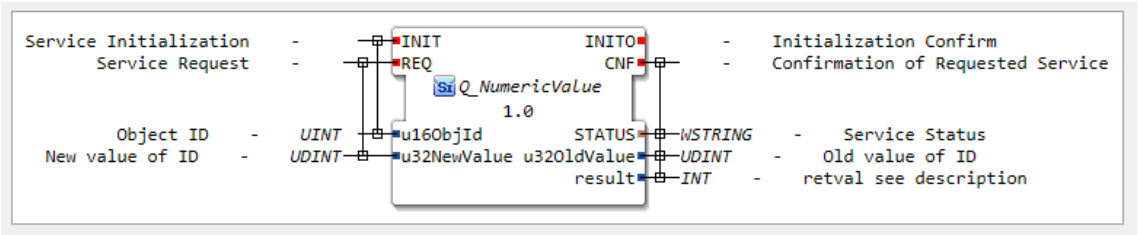
Vergleich mit ähnlichen Bausteinen

- **`Q_NumericValue` (ohne PHYS)**: Erwartet den Rohwert (UDINT) direkt. Der Anwender muss die Umrechnung selbst vornehmen. `Q_NumericValue_PHYS_LREAL` kapselt diesen Schritt und erhöht die Lesbarkeit und Wartbarkeit von Applikationen, die mit physikalischen Größen arbeiten.
- **`Q_NumericValue_PHYS_REAL`** (analog für REAL-Typ): Funktioniert identisch, jedoch mit einfacher Genauigkeit. Für Anwendungen, die höhere Genauigkeit benötigen (LREAL), ist der vorliegende Baustein die richtige Wahl.

Fazit

`Q_NumericValue_PHYS_LREAL` ist ein praxisorientierter und standardkonformer Funktionsblock, der die Anbindung an das ISOBUS-System vereinfacht. Durch die Integration der physikalischen Umrechnung entfällt manuelle Konvertierungslogik, was die Fehleranfälligkeit reduziert und die Wiederverwendbarkeit erhöht. Der Baustein eignet sich besonders für Steuerungsanwendungen, bei denen physikalische Werte in hoher Genauigkeit übertragen werden müssen.

Q_NumericValue



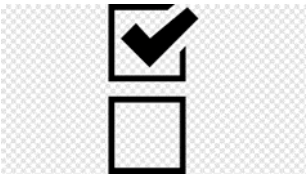
Normativen Referenz:

Dieser Funktionsbaustein bezieht sich auf den Abschnitt ISO 11783-6 F.22 Change Numeric Value command

ACHTUNG: dieser Befehl kann so oft aufgerufen werden wie vom Anwendungsprogrammierer gewünscht, jedoch kann ein zu häufiges Aufrufen den ISOBUS stark überlasten, weshalb dieser Baustein mit Bedacht eingesetzt werden sollte.

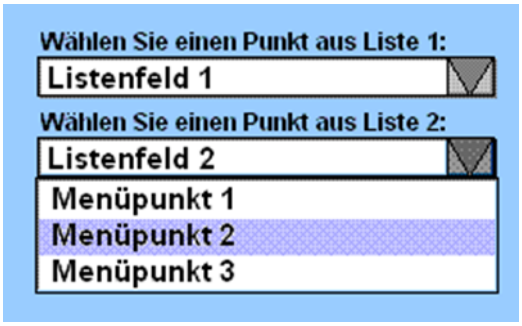
Aufgabe dieses Bausteins ist es eine der folgenden Objekte die auf einem VT dargestellt werden zu aktualisieren:

- Boolean input field: 1 byte for TRUE/FALSE

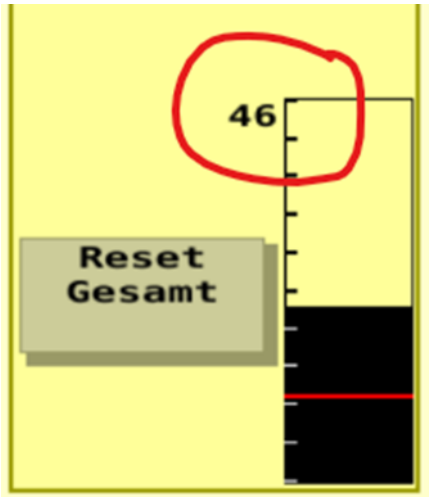


- Number input field: 4 bytes for integer input

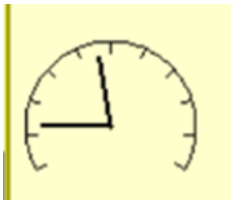
- List input field: 1 byte for list index



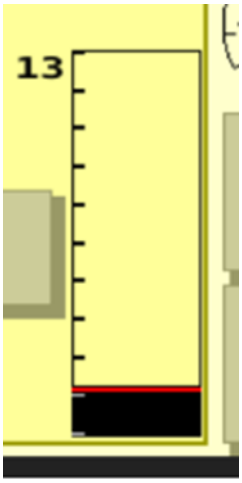
- List output field: 1 byte for list index
  - (wie list Input field, jedoch in VT3 nicht vorhanden)
- Number output field: 4 bytes for integer output



- Meter: 2 bytes for integer value



- Linear bar graph: 2 bytes for integer value



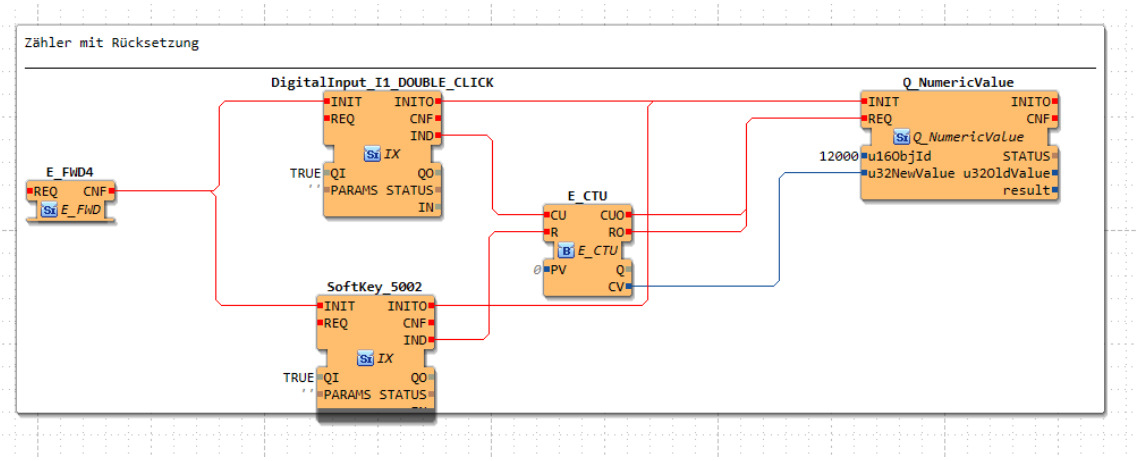
- Arched bar graph: 2 bytes for integer value

- Number variable: 4 bytes for integer value
  - Die Number Variable kann nicht direkt auf dem Display angezeigt werden, kann aber mit allen bisher genannen Objekten verknüpft werden; Vorteil: auch mehrfache Verknüpungen Möglich.
- Object pointer: 2 bytes for Object ID
  - Sinnvoll um andere Objekte an Stelle des Pointers ein& und auszublenden.

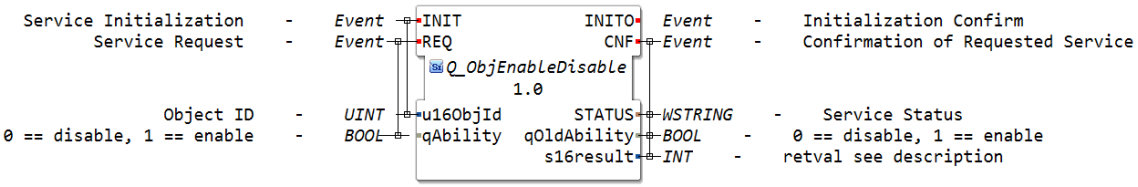
Querverweise:

BEISPIEL

Zähler mit Rücksetzung



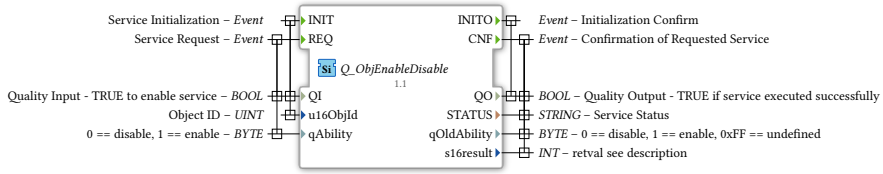
Q_ObjEnableDisable



• — ○ • — ○ • — ○ •

Einleitung

Der **Q_ObjEnableDisable** ist ein standardkonformer Funktionsbaustein zur Aktivierung/Deaktivierung von Eingabeobjekten in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.4) Spezifikation für VT-Eingabeelemente.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Objekt-ID)
- `REQ`: Aktivierungs-/Deaktivierungsanforderung

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Betriebsbestätigung

Daten-Eingänge

- `QI` (BOOL, Standard `TRUE`): Quality-Eingang – `TRUE` aktiviert den Dienst
- `u16ObjId` (UINT): Objekt-ID (16-bit)
- `qAbility` (BOOL): Aktivierungsstatus (TRUE=aktivieren, FALSE=deaktivieren)

Daten-Ausgänge

- `QO` (BOOL): Quality-Ausgang – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde
- `STATUS` (STRING): Betriebsstatusmeldung
- `qOldAbility` (BOOL): Vorheriger Aktivierungsstatus
- `s16result` (INT): ISO-konformer Ergebniscode

Gültige Objekt-IDs

`u16ObjId` — gültige Objekttypen (Annex F.4): Button (6000–6999), Input Boolean Field (7000–7999), Input String Field (8000–8999), Input Number Field (9000–9999), Input List Field (10000–10999), Animation Object (44000–44999).

ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei `INIT` den Baustein. Jede ID außerhalb dieser Bereiche ist ungültig.

Funktionsweise

Quality-Handling (QI/QO): Der Eingang `QI` (BOOL, Standard `TRUE`) ist mit den Ereignissen `INIT` und `REQ` verknüpft; `TRUE` aktiviert den Dienst. Der Ausgang `QO` (BOOL) ist mit `INITO` und `CNF` verknüpft und zeigt mit `TRUE` an, dass der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` mit Zielobjekt-ID
   - `INITO` bestätigt Betriebsbereitschaft
2. **Statusänderung**:

- `REQ` mit gewünschtem Aktivierungszustand
- Ändert die Interaktionsfähigkeit des Objekts
- `CNF` liefert Betriebsstatus und vorherigen Zustand

3. **Spezialfälle**:
   - Bei ungültigen Objekt-IDs wird VT_E_HANDLE_INVALID zurückgegeben

Technische Besonderheiten

✓ **ISO 11783-6 konform** (F.4) ✓ **Einfache Binärsteuerung** (Aktiviert/Deaktiviert) ✓ **Zustandsrückmeldung** (Vorheriger Status) ✓ **Breite Objektunterstützung** (Alle Eingabeelemente)

Statusreferenz

qAbility	Objektzustand
FALSE (0)	Deaktiviert (grau)
TRUE (1)	Aktiviert (normal)

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreiche Ausführung
−6	VT_E_OVERFLOW	Pufferüberlauf
−8	VT_E_NOACT	VT nicht bereit
−21	VT_E_NO_INSTANCE	Kein VT-Client verfügbar
−128	VT_E_HANDLE_INVALID	Ungültige Objekt-ID
−129	VT_E_ISO_INSTANCE_INVALID	Ungültige VT-Instanz
−130	VT_E_NOT_ALIVE	VT nicht aktiv

Anwendungsszenarien

- **Formularsteuerung**: Konditionelle Eingabefreigabe
- **Sicherheitsfunktionen**: Kritische Eingaben sperren
- **Workflow-Management**: Schrittweise Freischaltung
- **Diagnosemodi**: Eingabebeschränkungen

⚖️ **Vergleich mit ähnlichen Bausteinen**

Feature	Q_ObjEnableDisable	VtInputControl	VtObjectState
ISO-Standard	✓	✖	✖
Binärsteuerung	✓	✓	✖
Rückmeldung	✓	✖	✓
Objekttyp	Eingabeobjekte	Alle	Alle

Fazit

Der Q_ObjEnableDisable-Baustein bietet essentielle Eingabesteuerung:

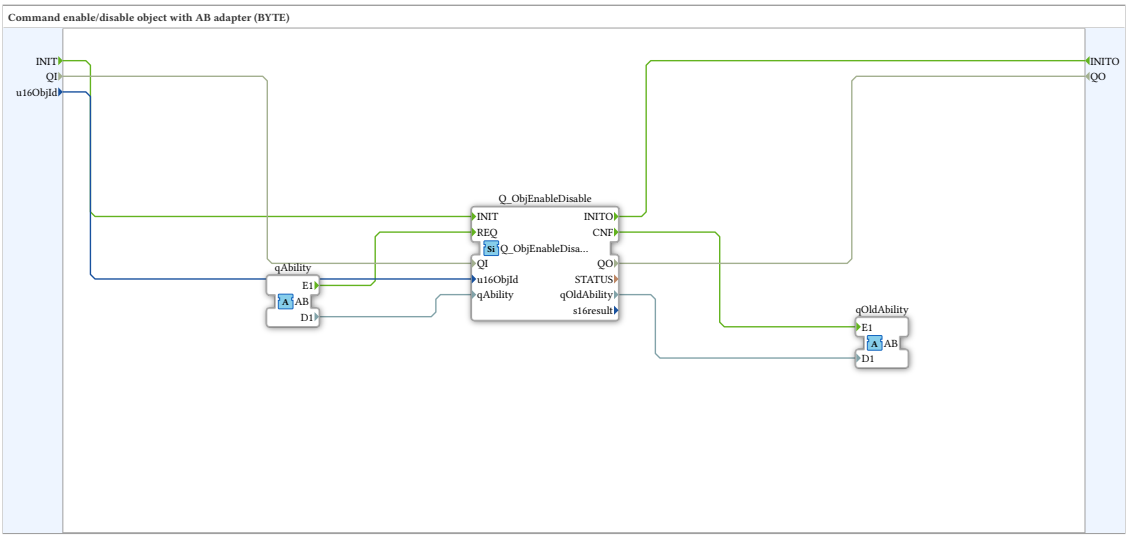
- **Einfach**: Klare Aktivierungslogik
- **Zuverlässig**: Integrierte Fehlererkennung
- **Flexibel**: Für alle Eingabeelementtypen

Unverzichtbar für:

- Interaktive Formulare

- Sichere Bedienoberflächen
- Zustandsgesteuerte Anwendungen

Q_ObjEnableDisable_AB



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock `Q_ObjEnableDisable_AB` dient als Wrapper, der die Steuerung eines binären Objekts (Ein-/Ausschalten) über einen unidirektionalen **AB-Adapter (BYTE)** ermöglicht. Er kapselt die Logik des Bausteins `isobus::UT::Q::Q_ObjEnableDisable` und stellt dessen Funktionalität über die standardisierten Adapterschnittstellen `qAbility` (Socket) und `qOldAbility` (Plug) bereit. Der FB eignet sich zur Anbindung an Adapter-basierte Systeme, z. B. für die Fernsteuerung von Aktoren.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung
`INIT`	EInit	Service-Initialisierung: startet den internen Baustein und übergibt die Objekt-ID.

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung
`INITO`	EInit	Bestätigung der erfolgreichen Initialisierung.

Daten-Eingänge

Name	Typ	Beschreibung
`QI`	BOOL	Quality Input – TRUE aktiviert den Dienst
`u16ObjId`	UINT	Eindeutige Identifikation des zu steuernden Objekts.

Daten-Ausgänge

Außer QO keine eigenen Datenausgänge – die übrigen Ausgangsdaten werden über den Adapter-Plug bereitgestellt.

Name	Typ	Kommentar
QO	BOOL	Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde

Adapter

Adaptertyp	Richtung	Beschreibung
Socket qAbility	Eingang	Unidirektionaler AB-Adapter: empfängt den Befehl (BYTE). 0 = deaktivieren, 1 = aktivieren, 0xFF = undefiniert.
Plug qOldAbility	Ausgang	Unidirektionaler AB-Adapter: gibt den vorherigen Zustand des Objekts zurück (gleiche Codierung).

Qualitätssignale QI/QO: Der Dateneingang QI (BOOL, Initialwert TRUE) ist dem Ereignis INIT zugeordnet und wird an QI des internen Bausteins Q_ObjEnableDisable durchgereicht. Der Datenausgang QO (BOOL) wird von Q_ObjEnableDisable.QO übernommen und ist dem Ereignis INITO zugeordnet.

Gültige Objekt-IDs

Es gelten die Gültigkeitsregeln des internen Basisbausteins Q_ObjEnableDisable:

u16ObjId — gültige Objekttypen (Annex F.4): Button (6000–6999), Input Boolean Field (7000–7999), Input String Field (8000–8999), Input Number Field (9000–9999), Input List Field (10000–10999), Animation Object (44000–44999).

ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei INIT den Baustein.

Funktionsweise

Der FB dient als Brücke zwischen einem AB-Adapter (BYTE) und dem internen Baustein Q_ObjEnableDisable. Die Abläufe sind:

1. **Initialisierung**
   Ein INIT-Ereignis mit einer gültigen Objekt-ID (u16ObjId) wird an den internen FB weitergeleitet. Der interne FB wird initialisiert und meldet dies über INITO zurück.
2. **Steuerung**
   Über den Socket qAbility wird ein BYTE-Wert empfangen. Das Ereignis qAbility.E1 triggert den internen FB (Ereignis REQ). Der Datenwert aus qAbility.D1 wird an den internen FB übergeben.

   Der interne FB verarbeitet den Befehl und setzt den Zustand des Objekts entsprechend. Der vorherige Zustand wird über die interne Verbindung an den Plug qOldAbility weitergegeben. Das Ereignis CNF des internen FB wird als qOldAbility.E1 ausgegeben.
3. **Rückmeldung**
   Der Plug qOldAbility liefert den alten Zustand des Objekts (BYTE) und das zugehörige Ereignis, sodass ein nachgeschalteter Baustein auf Zustandsänderungen reagieren kann.

Technische Besonderheiten

- **Adapter-Basierte Kapselung**
  Der FB verwendet den standardisierten unidirektionalen AB-Adapter, wodurch er einfach in bestehende Adapter-Vernetzungen integriert werden kann.
- **Typische BYTE-Codierung**
  0 ⟶ deaktivieren, 1 ⟶ aktivieren, 0xFF ⟶ undefiniert (Fehler oder keine Änderung). Diese Konvention wird sowohl am Eingang als auch am Ausgang verwendet.
- **Nur Initialisierung mit Objekt-ID**
  Das INIT-Ereignis muss mit einer gültigen u16ObjId erfolgen. Der FB selbst speichert keine persistente Konfiguration.
- **Ereignisgesteuerte Verarbeitung**
  Die Steuerung erfolgt ausschließlich über die Ereignisse des Adapter-Sockets. Ohne ein Ereignis auf qAbility.E1 findet keine Zustandsänderung statt.

Zustandsübersicht

Der FB besitzt keinen eigenen expliziten Zustandsautomaten – die Zustandslogik liegt vollständig im internen FB `Q_ObjEnableDisable`. Dennoch lassen sich folgende Betriebszustände ableiten:

Zustand	Beschreibung
Idle	Nach erfolgreicher Initialisierung (INITO gesendet) wartet der FB auf ein Ereignis am Socket `qAbility`.
Processing	Ein eingehendes Ereignis auf `qAbility.E1` wird verarbeitet; der interne FB führt die Umschaltung aus.
Completed	Verarbeitung abgeschlossen – der neue (bzw. alte) Zustand liegt am Plug an, Ereignis `qOldAbility.E1` wird ausgegeben.

Fehlerzustände (z. B. ungültige Objekt-ID, fehlerhafte INIT-Sequenz) werden durch den internen FB behandelt und können über Rückgabewerte oder spezielle BYTE-Codes (`0xFF`) signalisiert werden.

Anwendungsszenarien

- **Fernsteuerung eines Binärausgangs**
  Ein übergeordneter Steuerungsbaustein sendet über einen AB-Adapter den Befehl „Ein“ oder „Aus“. Der FB setzt den Befehl um und meldet den vorherigen Zustand zurück.
- **Zustandsabfrage ohne Änderung**
  Durch Senden des Codes `0xFF` kann der aktuelle Zustand abgefragt werden, ohne dass eine Änderung erfolgt.
- **Integration in Adapter-basierte Service-Interfaces**
  Der FB dient als Konverter zwischen einem AB-Adapter (z. B. aus einem HMI oder einer Fernbedienung) und einem funktionalen Baustein, der natives ISOBUS-Interface benötigt.

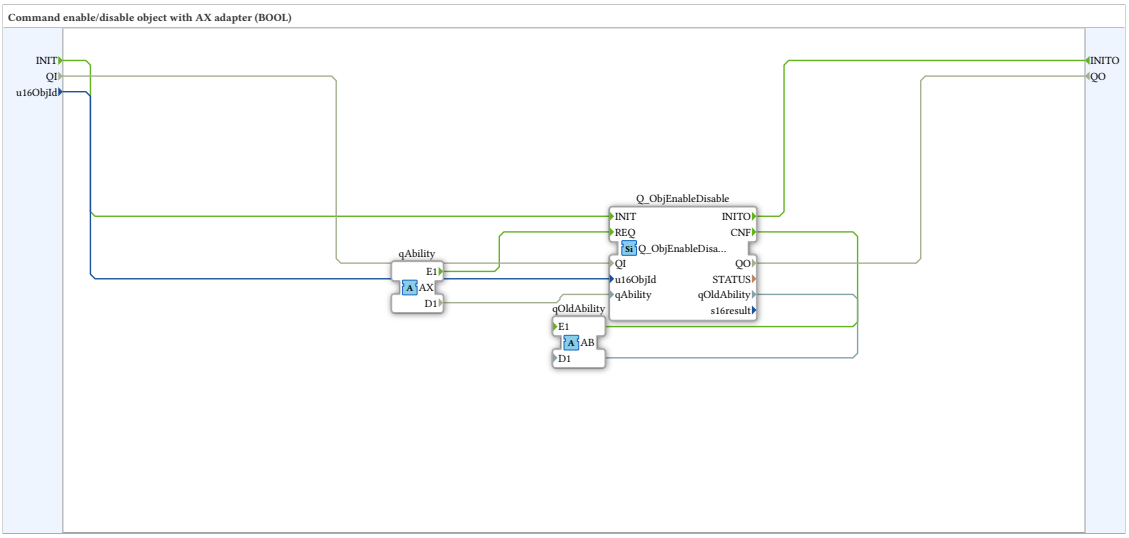
Vergleich mit ähnlichen Bausteinen

Baustein	Unterschied
`Q_ObjEnableDisable` (direkt)	Erfordert native Ereignis- und Datenschnittstellen ohne Adapter. `Q_ObjEnableDisable_AB` fügt die Adapter-Kompatibilität hinzu.
`Q_ObjEnableDisable_BB` (hypothetisch)	Würde einen bidirektionalen BB-Adapter verwenden. Der vorliegende FB verwendet den einfacheren unidirektionalen AB-Adapter.
Generischer „CommandOnOff“	Ist typischerweise nicht auf ein spezifisches Objekt (ID) ausgerichtet und besitzt keine Zustandsrückmeldung über Adapter.

Fazit

Der FB `Q_ObjEnableDisable_AB` ist ein nützlicher Wrapper, der die Funktionalität eines ISOBUS-Objektschalters über den standardisierten AB-Adapter zugänglich macht. Er erleichtert die Integration in modulare Adapter-Netzwerke und reduziert den Verdrahtungsaufwand. Die einfache Schnittstelle mit nur einem Ereigniseingang zur Steuerung und einer Rückmeldung über den Plug macht ihn besonders geeignet für wiederverwendbare Komponenten in der Automatisierungstechnik.

Q_ObjEnableDisable_AX



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **Q_ObjEnableDisable_AX** ist ein Composite-Baustein, der als Adapter-Wrapper für den internen Baustein `Q_ObjEnableDisable` dient. Er ermöglicht die Ansteuerung eines Enable/Disable-Befehls (BOOL) über einen unidirektionalen AX-Adapter (Socket) und gibt den vorherigen Zustand über einen AB-Adapter (Plug) zurück. Der Baustein wird typischerweise in ISOBUS-basierten Steuerungssystemen eingesetzt.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar
`INIT`	EInit	Initialisierung des Bausteins; übergeben wird die Objekt-ID (u16ObjId).

Ereignis-Ausgänge

Ereignis	Typ	Kommentar
`INITO`	EInit	Bestätigung der erfolgreichen Initialisierung.

Daten-Eingänge

Name	Typ	Kommentar
`QI`	BOOL	Quality Input – TRUE aktiviert den Dienst
`u16ObjId`	UINT	Objekt-ID, die beim INIT-Ereignis gesetzt wird.

Daten-Ausgänge

Außer `QO` keine direkten Datenausgänge; der alte Zustand wird über den Adapter-Plug `qOldAbility` ausgegeben.

Name	Typ	Kommentar
`QO`	`BOOL`	Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde

Adapter

Name	Typ	Richtung	Kommentar

qAbility	adapter::types::unidirectional::AX	Socket	Empfängt den Enable/Disable-Befehl (0 = disable, 1 = enable, 0xFF = undefined).
qOldAbility	adapter::types::unidirectional::AB	Plug	Gibt den vorherigen Zustand des Objekts zurück (gleiche Codierung).

Qualitätssignale QI/QO: Der Dateneingang QI (BOOL, Initialwert TRUE) ist dem Ereignis INIT zugeordnet und wird an QI des internen Bausteins Q_ObjEnableDisable durchgereicht. Der Datenausgang QO (BOOL) wird von Q_ObjEnableDisable.QO übernommen und ist dem Ereignis INITO zugeordnet.

Gültige Objekt-IDs

Es gelten die Gültigkeitsregeln des internen Basisbausteins Q_ObjEnableDisable:

u16ObjId — gültige Objekttypen (Annex F.4): Button (6000–6999), Input Boolean Field (7000–7999), Input String Field (8000–8999), Input Number Field (9000–9999), Input List Field (10000–10999), Animation Object (44000–44999).

ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei INIT den Baustein.

Funktionsweise

1. **Initialisierung**
   Ein INIT-Ereignis am Eingang INIT setzt die gewünschte Objekt-ID (u16ObjId) und initialisiert den internen Baustein Q_ObjEnableDisable. Nach erfolgreicher Initialisierung wird das Ereignis INITO ausgegeben.
2. **Signalverarbeitung**
   - Ein neuer Befehl wird über den Socket-Adapter qAbility empfangen:
     - Das Ereignis E1 des Adapters löst den REQ-Eingang des internen Bausteins aus.
     - Die Daten D1 (0/1/0xFF) werden auf den Eingang qAbility des internen Bausteins weitergeleitet.
   - Der interne Baustein verarbeitet den Befehl, aktiviert oder deaktiviert das Objekt mit der gegebenen ID und gibt den vorherigen Zustand aus.
   - Die Bestätigung CNF des internen Bausteins wird auf das Ereignis E1 des Plug-Adapters qOldAbility übertragen.
   - Der vorherige Zustand wird über den Datenausgang D1 des Plug-Adapters bereitgestellt.
3. **Ausgabe alter Zustand**
   Der Adapter-Plug qOldAbility stellt den Zustand vor der letzten Änderung als BOOL-Wert (0/1/0xFF) zur Verfügung. Die Ereignisverbindung stellt sicher, dass das Ausgangseignis zeitlich korrekt erfolgt.

Technische Besonderheiten

- Der Baustein kapselt die Logik des komplexeren Q_ObjEnableDisable und bietet eine vereinfachte Schnittstelle über zwei standardisierte Adapter (AX/AB).
- Adapter-Socket qAbility erwartet unidirektionale Kommunikation (nur Wertübergabe), Adapter-Plug qOldAbility gibt das Ergebnis zurück.
- Die Objekt-ID wird nur bei INIT gesetzt; ein Wechsel der ID während des Betriebs ist nicht vorgesehen.

Zustandsübersicht

Der FB Q_ObjEnableDisable_AX selbst besitzt keinen eigenen Zustandsautomaten. Die interne Zustandslogik wird vom enthaltenen Baustein Q_ObjEnableDisable realisiert. Dessen typische Zustände sind:

- **IDLE** – Warten auf einen Befehl (REQ).
- **Processing** – Verarbeitung des Befehls (z. B. Senden einer CAN-Nachricht).
- **Done** – Befehl abgeschlossen, Ausgabe des alten Zustands und Rücksprung nach IDLE.

Anwendungsszenarien

- **ISOBUS-Steuerung**: Aktivieren/Deaktivieren eines landwirtschaftlichen Geräts (z. B. Ventil, Beleuchtung) basierend auf einem booleschen Signal eines AX-Adapters.
- **Universaladapter**: Einbindung von Enable/Disable-Funktionen in Systeme, die standardisierte unidirektionale Adapter verwenden.
- **Zustandsrückmeldung**: Erfassung des vorherigen Zustands für Diagnose- oder Überwachungszwecke.

Vergleich mit ähnlichen Bausteinen

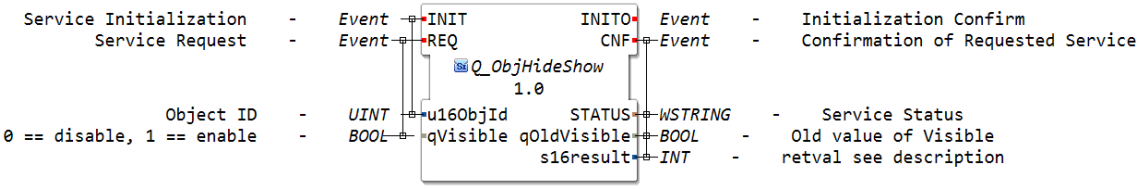
Baustein	Schnittstelle	Besonderheit
Q_ObjEnableDisable	Direkte Ereignis-/Dateneingänge	Erfordert separate Bool-Eingänge und Ereignisse.
Q_ObjEnableDisable_AX	AX/AB-Adapter	Einfache Integration in Adapter-basierte Architekturen; reduziert Verdrahtungsaufwand.

Der AX-Wrapper fügt keine neue Funktionalität hinzu, sondern optimiert die Anbindung an bestehende Adapter-Kommunikation.

Fazit

Der Funktionsblock **Q_ObjEnableDisable_AX** ist ein praktischer Adapter-Wrapper, der die Steuerung eines ISOBUS-Objekts mittels Enable/Disable-Befehlen über einen unidirektionalen AX-Adapter vereinfacht. Er kapselt die interne Logik, ermöglicht eine saubere Schnittstellentrennung und gibt den vorherigen Zustand über einen AB-Adapter zurück. Der Baustein eignet sich besonders für modulare Steuerungsarchitekturen, die auf standardisierten Adapter-Typen basieren.

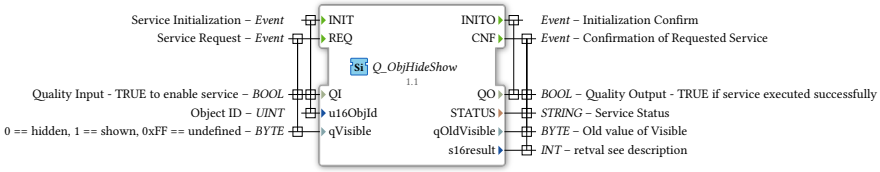
Q_ObjHideShow



• — ○ • — ○ • — ○ •

Einleitung

Der **Q_ObjHideShow** ist ein standardkonformer Funktionsbaustein zur Sichtbarkeitssteuerung von Container-Objekten in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.2) Spezifikation für VT-Container.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Container-Objekt-ID)
- `REQ`: Sichtbarkeitsänderungs-Anforderung

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Änderungsbestätigung

Daten-Eingänge

- `QI` (BOOL, Standard `TRUE`): Quality-Eingang – `TRUE` aktiviert den Dienst
- `u16ObjId` (UINT): Container-Objekt-ID (16-bit)
- `qVisible` (BOOL): Sichtbarkeitsstatus (TRUE=anzeigen, FALSE=verstecken)

Daten-Ausgänge

- `QO` (BOOL): Quality-Ausgang – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde
- `STATUS` (STRING): Betriebsstatusmeldung
- `qOldVisible` (BOOL): Vorheriger Sichtbarkeitsstatus

- `s16result` (INT): ISO-konformer Ergebniscode

Gültige Objekt-IDs

`u16ObjId` — gültige Objekttypen (Annex F.2): Container (3000–3999).
ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei `INIT` den Baustein. Jede ID außerhalb dieses Bereichs ist ungültig.

Funktionsweise

Quality-Handling (QI/QO): Der Eingang `QI` (BOOL, Standard `TRUE`) ist mit den Ereignissen `INIT` und `REQ` verknüpft; `TRUE` aktiviert den Dienst. Der Ausgang `QO` (BOOL) ist mit `INITO` und `CNF` verknüpft und zeigt mit `TRUE` an, dass der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` mit Container-Objekt-ID
   - `INITO` bestätigt Betriebsbereitschaft
2. **Sichtbarkeitsänderung**:
   - `REQ` mit gewünschtem Sichtbarkeitsstatus
   - Ändert die Darstellung des Containers und aller enthaltenen Objekte
   - `CNF` liefert Betriebsstatus und vorherigen Zustand
3. **Spezialfälle**:
   - Bei ungültigen Objekt-IDs wird VT_E_HANDLE_INVALID zurückgegeben

Technische Besonderheiten

✓ ISO 11783-6 konform (F.2) **✓ Kaskadierender Effekt** (Beeinflusst enthaltene Objekte) **✓ Sofortige GUI-Aktualisierung ✓ Zustandserhaltung** (Merkt vorherigen Status)

Statusreferenz

qVisible	Containerzustand
FALSE (0)	Unsichtbar
TRUE (1)	Sichtbar

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreiche Ausführung
−6	VT_E_OVERFLOW	Pufferüberlauf
−8	VT_E_NOACT	VT nicht bereit
−21	VT_E_NO_INSTANCE	Kein VT-Client verfügbar
−128	VT_E_HANDLE_INVALID	Ungültige Container-ID
−129	VT_E_ISO_INSTANCE_INVALID	Ungültige VT-Instanz
−130	VT_E_NOT_ALIVE	VT nicht aktiv

Anwendungsszenarien

- **Dynamische Oberflächen**: Kontextabhängige Anzeigen
- **Bildschirmmanager**: View-Wechsel
- **Alarmmanagement**: Störmeldungscontainer
- **Benutzerführung**: Schrittweise Anleitungen

⚖️ **Vergleich mit ähnlichen Bausteinen**

Feature	Q_ObjHideShow	VtContainerControl	VtVisibilityManager

ISO-Standard	✓	✖	✖
Containerfokus	✓	✓	✖
Kaskadierend	✓	✖	✓
Rückmeldung	✓	✖	✓

🛠️ Zugehörige Übungen

- Uebung_014

Fazit

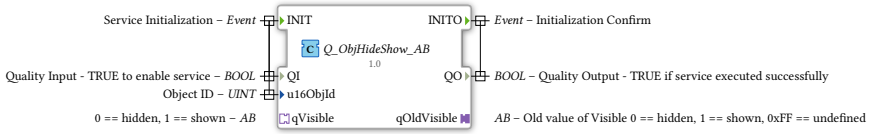
Der Q_ObjHideShow-Baustein bietet zentrale Sichtbarkeitskontrolle:

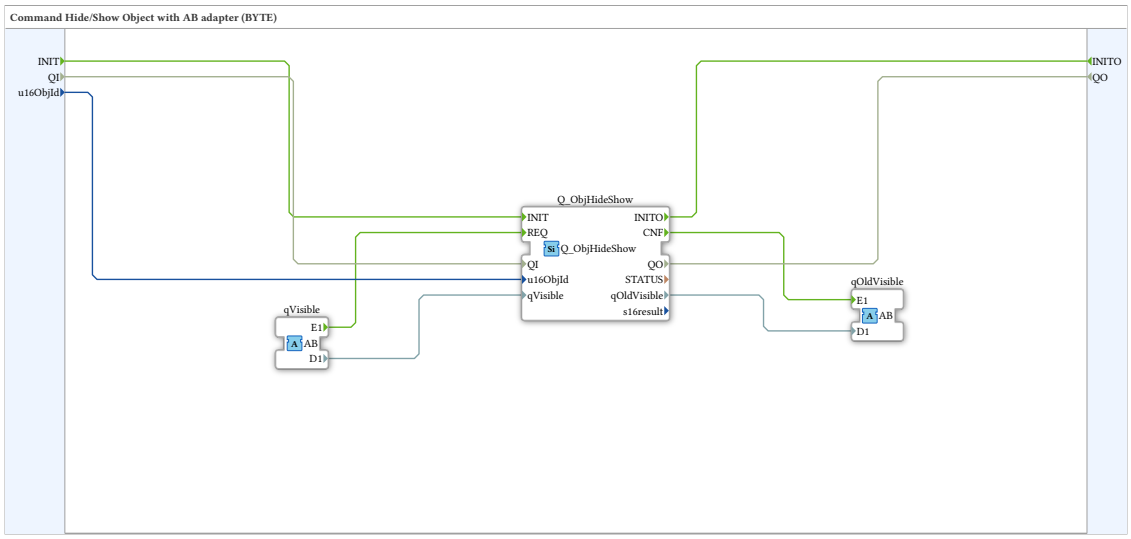
- **Effizient**: Sofortige GUI-Aktualisierung
- **Umfassend**: Beeinflusst gesamte Containerelemente
- **Zuverlässig**: ISO-konforme Implementierung

Essential für:

- Adaptive Benutzeroberflächen
- Komplexe Maskenhierarchien
- Zustandsgesteuerte Visualisierungen

Q_ObjHideShow_AB





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **Q_ObjHideShow_AB** dient als Wrapper für den Baustein `Q_ObjHideShow` und ermöglicht das Verstecken bzw. Anzeigen eines Objekts über einen **AB-Adapter** (unidirektional, BYTE). Er kapselt die direkte Ansteuerung des internen FB und stellt eine adapterbasierte Schnittstelle zur Verfügung, die eine lose Kopplung in der Anwendung fördert.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	mit Variablen	Kommentar
INIT	QI, u16ObjId	Service Initialization; Objekt-ID wird übergeben

Ereignis-Ausgänge

Ereignis	Kommentar
INITO	Initialisierung bestätigt

Daten-Eingänge

Name	Typ	Kommentar
QI	BOOL	Quality Input – TRUE aktiviert den Dienst
u16ObjId	UINT	Objekt-ID (Object ID)

Daten-Ausgänge

Auf FB-Ebene gibt es außer QO keine expliziten Datenausgänge. Die übrigen Ausgangsdaten werden über den Adapter qOldVisible bereitgestellt.

Name	Typ	Kommentar
QO	BOOL	Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde

Adapter

Richtung	Name	Typ	Kommentar
Socket (Eingang)	qVisible	adapter::types::unidirectional::A	Sollwert der Sichtbarkeit: 0 = versteckt, 1 = angezeigt
Plug (Ausgang)	qOldVisible	adapter::types::unidirectional::A	Alter Wert der Sichtbarkeit: 0 = versteckt, 1 = angezeigt, 0xFF = undefiniert

Qualitätssignale QI/QO: Der Dateneingang QI (BOOL, Initialwert TRUE) ist dem Ereignis INIT zugeordnet und wird an QI des internen Bausteins Q_ObjHideShow durchgereicht. Der Datenausgang QO (BOOL) wird von Q_ObjHideShow.QO übernommen und ist dem Ereignis INITO zugeordnet.

Gültige Objekt-IDs

Es gelten die Gültigkeitsregeln des internen Basisbausteins Q_ObjHideShow:
u16ObjId — gültige Objekttypen (Annex F.2): Container (3000–3999).
ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei INIT den Baustein.

Funktionsweise

1. **Initialisierung**
   Durch ein Ereignis am Eingang INIT wird der interne FB Q_ObjHideShow initialisiert. Die übergebene u16ObjId identifiziert das zu steuernde Objekt. Nach erfolgreichem Start wird das Ereignis INITO ausgegeben.
2. **Steuerung der Sichtbarkeit**
   Sobald am Socket-Adapter qVisible ein Ereignis (E1) eintrifft, wird der zugehörige Datenwert (D1) ausgelesen und an den internen FB weitergeleitet. Der interne FB setzt daraufhin die Sichtbarkeit des Objekts entsprechend (0 = versteckt, 1 = angezeigt).
3. **Rückmeldung**
   Nach Ausführung der Aktion sendet der interne FB das Ereignis CNF an den Plug-Adapter qOldVisible.

Der alte Sichtbarkeitszustand wird als Datenwert (D1) auf `qOldVisible` bereitgestellt. So kann die aufrufende Komponente den vorherigen Zustand abfragen.

Technische Besonderheiten

- **Wrapper-Konzept**: Der FB kapselt die direkte Verwendung von `Q_ObjHideShow` und ersetzt die lose Ereignis-/Datenverbindung durch standardisierte AB-Adapter, was eine modulare und wiederverwendbare Anbindung ermöglicht.
- **AB-Adapter (unidirektional, BYTE)**: Die Kommunikation erfolgt über einen unidirektionalen Adapter, der jeweils nur ein Ereignis und ein zugehöriges Byte überträgt. Dies vereinfacht die Einbindung in Adapter-basierte Architekturen.
- **Zustandserhaltung**: Der interne FB speichert den letzten Sichtbarkeitszustand; bei jedem neuen Befehl wird der alte Zustand am Ausgangsadapter zurückgemeldet.
- **Undefinierter Wert (0xFF)**: Der Ausgang `qOldVisible` kann den Wert 0xFF liefern, falls der vorherige Zustand nicht bekannt ist (z. B. nach Neustart ohne vorherige Initialisierung).

Zustandsübersicht

Der FB selbst besitzt keinen eigenen Zustandsautomaten. Sein Verhalten wird vollständig durch den eingebetteten Baustein `Q_ObjHideShow` bestimmt. Grundsätzlich gilt:

- **Nach INIT**: FB ist betriebsbereit und wartet auf Ereignisse vom Socket-Adapter `qVisible`.
- **Bei jedem eingehenden Socket-Ereignis**: Sichtbarkeit wird gesetzt und das Ergebnis über den Plug-Adapter zurückgemeldet.

Anwendungsszenarien

- **Visualisierungssteuerung**: Ein- und Ausblenden grafischer Objekte (z. B. in HMI-Systemen) basierend auf binären oder numerischen Steuerwerten.
- **Zustandsrückmeldung**: Überwachung von Sichtbarkeitsänderungen; die alte Sichtbarkeit kann für Logikentscheidungen oder Protokollierung genutzt werden.
- **Adapter-basierte Automatisierung**: Einbindung in ein System, das auf Adapter-Schnittstellen setzt, um Austauschbarkeit und Entkopplung zu erreichen.

Vergleich mit ähnlichen Bausteinen

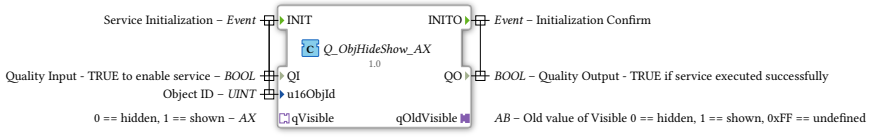
Baustein	Eigenschaften
`Q_ObjHideShow` (direkt)	Ereignis-/Daten-Schnittstelle ohne Adapter; direkte Verschaltung erforderlich
`Q_ObjHideShow_AB` (dieser FB)	Nutzt AB-Adapter für lose Kopplung; einfachere Integration in adapterbasierte Netzwerke
Andere AB-basierte Wrapper	Ähnliches Konzept für andere Befehle (z. B. `Q_ObjMove_AB`), jedoch spezifisch auf Sichtbarkeit ausgelegt

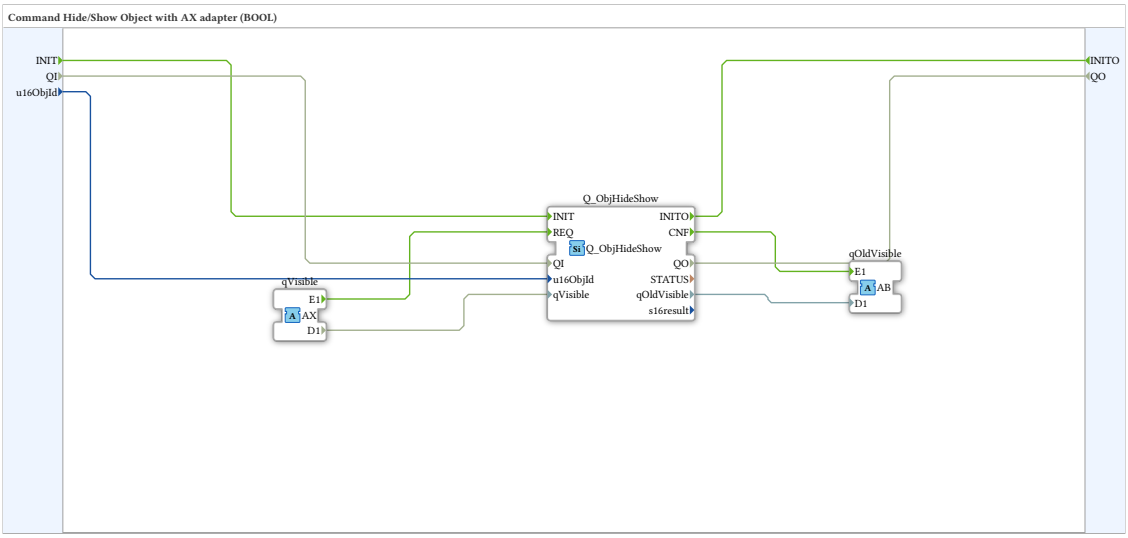
Der wesentliche Vorteil des Wrappers liegt in der Anpassung an die AB-Adapter-Schnittstelle, ohne die eigentliche Funktionalität zu verändern.

Fazit

Der Funktionsblock `Q_ObjHideShow_AB` stellt eine adapterbasierte, wiederverwendbare Komponente zur Steuerung der Sichtbarkeit von Objekten dar. Durch die Kapselung des internen FB und die Nutzung unidirektionaler AB-Adapter wird eine saubere Trennung von Steuerlogik und Ausführung erreicht. Er eignet sich besonders für modulare Automatisierungslösungen, die auf lose Kopplung und standardisierte Schnittstellen angewiesen sind.

Q_ObjHideShow_AX





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **Q_ObjHideShow_AX** dient als Adapter-Wrapper für den bestehenden Baustein `Q_ObjHideShow`. Er erlaubt das einheitliche Verbergen und Anzeigen eines Objekts (Hide/Show) über einen unidirektionalen AX-Adapter. Der Sichtbarkeitswert wird dabei als boolescher Zustand (`0` = verborgen, `1` = angezeigt) übergeben.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar	Mit Variable
`INIT`	EInit	Service Initialization	`QI`, `u16ObjId`

Ereignis-Ausgänge

Name	Typ	Kommentar
`INITO`	EInit	Bestätigung der Initialisierung

Daten-Eingänge

Name	Typ	Kommentar
`QI`	BOOL	Quality Input – TRUE aktiviert den Dienst
`u16ObjId`	UINT	Objekt-Identifikation

Daten-Ausgänge

Der Baustein besitzt außer `QO` keine eigenen Datenausgänge. Der alte Sichtbarkeitswert wird über den Plug-Adapter `qOldVisible` ausgegeben.

Name	Typ	Kommentar
`QO`	`BOOL`	Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde

Adapter

Richtung	Name	Typ	Kommentar

Socket	qVisible	unidirectional::AX	Eingabe des neuen Sichtbarkeitswerts (0 = hidden, 1 = shown)
Plug	qOldVisible	unidirectional::AB	Ausgabe des bisherigen Sichtbarkeitswerts (0/1/0xFF)

Qualitätssignale QI/QO: Der Dateneingang QI (BOOL, Initialwert TRUE) ist dem Ereignis INIT zugeordnet und wird an QI des internen Bausteins Q_ObjHideShow durchgereicht. Der Datenausgang QO (BOOL) wird von Q_ObjHideShow.QO übernommen und ist dem Ereignis INITO zugeordnet.

Gültige Objekt-IDs

Es gelten die Gültigkeitsregeln des internen Basisbausteins Q_ObjHideShow:
u16ObjId — gültige Objekttypen (Annex F.2): Container (3000–3999).
ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei INIT den Baustein.

Funktionsweise

1. Über das Ereignis INIT wird der Baustein initialisiert und die Objekt-ID (u16ObjId) an den internen FB Q_ObjHideShow übergeben.
2. Sobald am Socket qVisible ein Ereignis eintrifft (verbunden mit dem internen REQ), wird der neue Sichtbarkeitswert (qVisible.D1) an den internen FB weitergeleitet.
3. Der interne FB führt die Umschaltung durch und gibt über sein CNF-Ereignis den alten Wert aus. Dieser wird über das Plug qOldVisible (Ereignis E1 und Daten D1) nach außen gemeldet.
4. Der Ausgang INITO signalisiert den erfolgreichen Abschluss der Initialisierung.

Technische Besonderheiten

- **Adapter-Kapselung:** Der Baustein nutzt die standardisierten Adapter-Typen AX (für Eingang) und AB (für Ausgang), sodass er nahtlos in adapterbasierte ISOBUS-Netzwerke eingebunden werden kann.
- **Wiederverwendung:** Die eigentliche Logik wird durch den bewährten FB Q_ObjHideShow realisiert – der Wrapper fügt lediglich die Adapterschnittstelle hinzu.
- **Boolesche Steuerung:** Der Sichtbarkeitswert wird als einfacher Bool-Wert (über AX-Adapter) übertragen, was die Verwendung in typischen Ein-/Ausschalt-Szenarien erleichtert.

Zustandsübersicht

Der Baustein besitzt keine explizit sichtbare Zustandsmaschine. Er durchläuft einen Initialisierungsschritt (INIT → INITO) und reagiert anschließend zyklisch auf eingehende Ereignisse am Socket qVisible (REQ → CNF). Der interne FB Q_ObjHideShow kann jedoch einen impliziten Zustand (z. B. aktuell angezeigte Sichtbarkeit) halten.

Anwendungsszenarien

- Ein-/Ausblenden von Bedienelementen auf einem ISOBUS-Terminal.
- Steuerung der Anzeige von Symbolen oder Textfeldern in Abhängigkeit von Benutzereingaben oder Systemzuständen.
- Einheitliche Schnittstelle für alle Objekte, die über Adapter verbunden werden sollen.

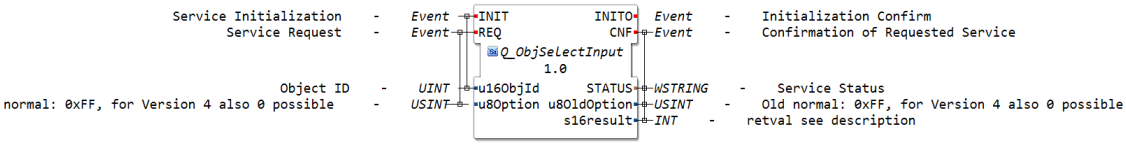
Vergleich mit ähnlichen Bausteinen

Der direkte FB Q_ObjHideShow erfordert separate Event- und Datenverbindungen für die Steuerung. Der wrapper-Baustein Q_ObjHideShow_AX kapselt diese Verbindungen in zwei standardisierten Adaptern (AX/AB) und vereinfacht so die Integration in bestehende adapterbasierte Architekturen. Zudem wird der alte Sichtbarkeitswert automatisch über den Plug-Adapter bereitgestellt.

Fazit

Der **Q_ObjHideShow_AX** ist ein praktischer Wrapper, der die bewährte Funktionalität des Q_ObjHideShow in eine adapterbasierte Schnittstelle überführt. Er ermöglicht eine saubere, modulare Anbindung von Hide/Show-Befehlen und erleichtert die Wiederverwendung in komplexen ISOBUS-Steuerungssystemen.

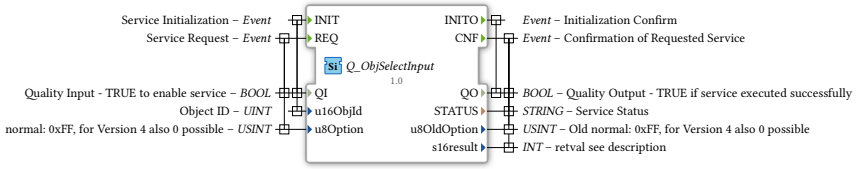
Q_ObjSelectInput



• — ○ • — ○ • — ○ •

Einleitung

Der **Q_ObjSelectInput** ist ein standardkonformer Funktionsbaustein zur programmatischen Auswahl von Eingabefeldern in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.6) Spezifikation für VT-Eingabeelemente.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Objekt-ID)
- `REQ`: Auswahlanforderung (mit Option-Parameter)

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Auswahlbestätigung

Daten-Eingänge

- `QI` (BOOL, Standard `TRUE`): Quality-Eingang – `TRUE` aktiviert den Dienst
- `u16ObjId` (UINT): Eingabeobjekt-ID (16-bit)
- `u8Option` (USINT): Auswahloption (0x00 oder 0xFF)

Daten-Ausgänge

- `QO` (BOOL): Quality-Ausgang – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde
- `STATUS` (STRING): Betriebsstatusmeldung
- `u8OldOption` (USINT): Vorherige Auswahloption
- `s16result` (INT): ISO-konformer Ergebniscode

Gültige Objekt-IDs

`u16ObjId` — gültige Objekttypen (Annex F.6): Key (5000–5999), Button (6000–6999), Input Boolean Field (7000–7999), Input String Field (8000–8999), Input Number Field (9000–9999), Input List Field (10000–10999).

ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei `INIT` den Baustein. Jede ID außerhalb dieser Bereiche ist ungültig.

Funktionsweise

Quality-Handling (QI/QO): Der Eingang `QI` (BOOL, Standard `TRUE`) ist mit den Ereignissen `INIT` und `REQ` verknüpft; `TRUE` aktiviert den Dienst. Der Ausgang `QO` (BOOL) ist mit `INITO` und `CNF` verknüpft und zeigt mit `TRUE` an, dass der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` mit Zielobjekt-ID
   - `INITO` bestätigt Betriebsbereitschaft
2. **Feldauswahl**:
   - `REQ` mit Auswahloption
   - Setzt den Fokus auf das Eingabefeld

- `CNF` liefert Betriebsstatus und vorherige Option

3. **Optionen**:
   - `0xFF`: Standardauswahl (VT Version 3+)
   - `0x00`: Erweiterte Auswahl (nur VT Version 4+)

Technische Besonderheiten

✓ **ISO 11783-6 konform** (F.6) ✓ **VT-Versionsabhängig** (Option 0x00 nur ab VT4) ✓ **Sofortige Fokusänderung** ✓ **Rückmeldung** des vorherigen Zustands

Optionsreferenz

u8Option	Beschreibung	VT-Version
0xFF	Standardauswahl	3+
0x00	Erweiterte Auswahl	4+

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreiche Ausführung
−6	VT_E_OVERFLOW	Pufferüberlauf
−8	VT_E_NOACT	VT nicht bereit
−21	VT_E_NO_INSTANCE	Kein VT-Client verfügbar
−129	VT_E_ISO_INSTANCE_INVALID	Ungültige VT-Instanz
−130	VT_E_NOT_ALIVE	VT nicht aktiv

Anwendungsszenarien

- **Formularnavigation**: Automatische Feldauswahl
- **Eingabeassistenten**: Geführte Bedienabläufe
- **Fehlermanagement**: Direktes Springen zu Fehlerfeldern
- **Touch-Optimierung**: Programmatische Fokussteuerung

⚖️ **Vergleich mit ähnlichen Bausteinen**

Feature	Q_ObjSelectInput	VtFocusControl	VtInputSelector
ISO-Standard	✓	✖	✖
VT-Version 4+	✓	✖	✓
Optionssteuerung	✓	✖	✖
Rückmeldung	✓	✖	✓

Fazit

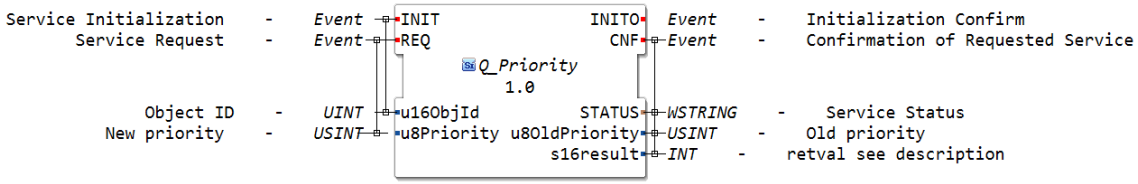
Der Q_ObjSelectInput-Baustein bietet präzise Eingabefeldsteuerung:

- **Effizient**: Direkte Fokusänderung
- **Flexibel**: Versionsspezifische Optionen
- **Zuverlässig**: ISO-konforme Implementierung

Unverzichtbar für:

- Professionelle Formularanwendungen
- Geführte Bedienprozesse
- Touchscreen-Optimierte Oberflächen

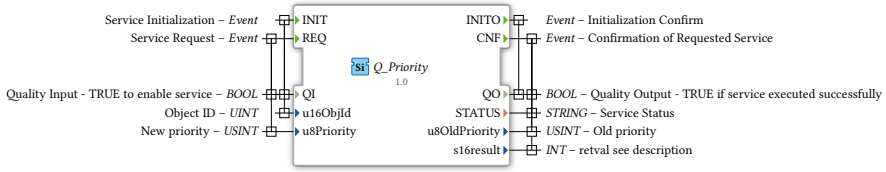
Q_Priority



• — ○ • — ○ • — ○ •

Einleitung

Der **Q_Priority** ist ein standardkonformer Funktionsbaustein zur Prioritätssteuerung von Alarmmasken in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.40) Spezifikation für VT-Alarmmanagement.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Alarmmasken-ID)
- `REQ`: Prioritätsänderungs-Anforderung

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Änderungsbestätigung

Daten-Eingänge

- `QI` (BOOL, Standard `TRUE`): Quality-Eingang – `TRUE` aktiviert den Dienst
- `u16ObjId` (UINT): Alarmmasken-Objekt-ID (16-bit)
- `u8Priority` (USINT): Neue Priorität (0-255)

Daten-Ausgänge

- `QO` (BOOL): Quality-Ausgang – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde
- `STATUS` (STRING): Betriebsstatusmeldung
- `u8OldPriority` (USINT): Vorherige Priorität
- `s16result` (INT): ISO-konformer Ergebniscode

Gültige Objekt-IDs

`u16ObjId` adressiert die **AlarmMask**, deren Priorität geändert wird (F.40):
AlarmMask: 2000–2999
ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei `INIT` den Baustein.

Funktionsweise

Quality-Handling (QI/QO): Der Eingang `QI` (BOOL, Standard `TRUE`) ist mit den Ereignissen `INIT` und `REQ` verknüpft; `TRUE` aktiviert den Dienst. Der Ausgang `QO` (BOOL) ist mit `INITO` und `CNF` verknüpft und zeigt mit `TRUE` an, dass der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` mit Alarmmasken-ID
   - `INITO` bestätigt Betriebsbereitschaft
2. **Prioritätsänderung**:
   - `REQ` mit neuem Prioritätswert
   - Aktualisiert die Alarmdarstellungsreihenfolge

- `CNF` liefert Ergebnisstatus und vorherigen Wert

3. **Prioritätslogik**:
   - Niedrigere Zahlen = Höhere Priorität
   - Beeinflusst Alarm-Anzeigereihenfolge

Technische Besonderheiten

✓ **ISO 11783-6 konform** (F.40) ✓ **Echtzeit-Priorisierung** (Sofortige Wirkung) ✓ **Rückverfolgbarkeit** (Vorherige Priorität) ✓ **Breiter Wertebereich** (0-255)

Prioritätsreferenz

Priorität	Typische Verwendung
0-31	Kritische Systemalarme
32-63	Wichtige Prozessalarme
64-127	Standardwarnungen
128-255	Informationsmeldungen

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreiche Änderung
−6	VT_E_OVERFLOW	Pufferüberlauf
−8	VT_E_NOACT	VT nicht bereit
−21	VT_E_NO_INSTANCE	Kein VT-Client verfügbar
−129	VT_E_ISO_INSTANCE_INVALID	Ungültige VT-Instanz
−130	VT_E_NOT_ALIVE	VT nicht aktiv

Anwendungsszenarien

- **Alarmmanagement**: Dynamische Prioritätsanpassung
- **Prozesssteuerung**: Wichtigkeitseinstufung
- **Diagnosesysteme**: Eskalationsmanagement
- **Wartungsmodi**: Temporäre Prioritätsänderungen

⚖️ **Vergleich mit ähnlichen Bausteinen**

Feature	Q_Priority	VtAlarmPriority	VtEventManager
ISO-Standard	✓	✖	✖
Echtzeitwirkung	✓	✓	✖
Wertebereich	0-255	0-127	0-31
Rückmeldung	✓	✖	✓

Fazit

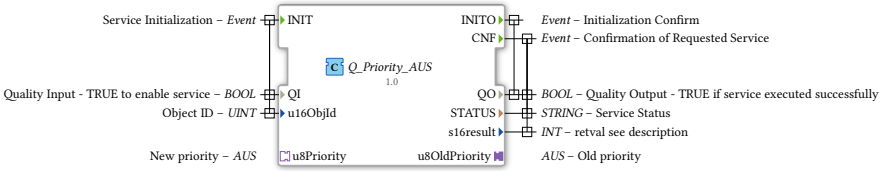
Der Q_Priority-Baustein bietet essentielle Alarmsteuerung:

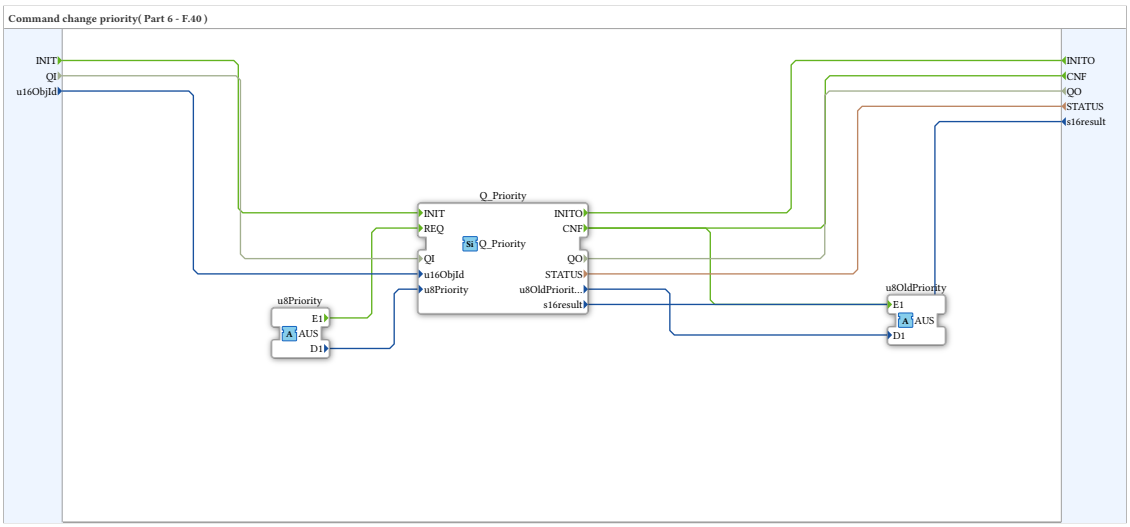
- **Dynamisch**: Sofortige Prioritätsanpassung
- **Transparent**: Rückmeldung des vorherigen Zustands
- **Robust**: Volle ISO-Konformität

Unverzichtbar für:

- Professionelle Alarmmanagementsysteme
- Prozesskritische Anwendungen
- Flexible Diagnoselösungen

Q_Priority_AUS





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock Q_Priority_AUS dient zur Änderung der Priorität einer Alarmmaske gemäß ISO 11783-6 (Part 6, F.40). Über einen Initialisierungsvorgang wird die Alarmmaske identifiziert, anschließend kann eine neue Priorität gesetzt werden. Die Bestätigung des Dienstes sowie der alte Prioritätswert werden ausgegeben.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung
INIT	EInit	Service-Initialisierung; wird mit dem Daten-Eingang `u16ObjId` ausgeführt.

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung
INITO	EInit	Bestätigung der erfolgreichen Initialisierung.
CNF	Event	Bestätigung des angeforderten Dienstes; wird mit `STATUS` und `s16result` ausgegeben.

Daten-Eingänge

Name	Typ	Beschreibung	Initialwert
QI	BOOL	Quality Input – TRUE aktiviert den Dienst	TRUE
u16ObjId	UINT	Objekt-ID der Alarmmaske	ID_NULL

Daten-Ausgänge

Name	Typ	Beschreibung
QO	BOOL	Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde
STATUS	STRING	Status des Dienstes (z. B. Fehler- oder Erfolgsmeldung)
s16result	INT	Rückgabewert gemäß Beschreibung (siehe Funktionsweise)

Adapter

Name	Typ	Richtung	Beschreibung
u8Priority	AUS (unidirectional)	Socket	Neue Priorität (wird über den Adapter gesendet)
u8OldPriority	AUS (unidirectional)	Plug	Alte Priorität (wird über den Adapter empfangen)

Qualitätssignale QI/QO: Der Dateneingang `QI` (BOOL, Initialwert `TRUE`) ist dem Ereignis `INIT` zugeordnet und wird an `QI` des internen Bausteins `Q_Priority` durchgereicht. Der Datenausgang `QO` (BOOL) wird von `Q_Priority.QO` übernommen und ist den Ereignissen `INITO` und `CNF` zugeordnet.

Gültige Objekt-IDs

Der Baustein kapselt **Q_Priority** (F.40). `u16ObjId` adressiert die **AlarmMask**, deren Priorität geändert wird:

AlarmMask: 2000–2999
ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei `INIT` den Baustein.

Funktionsweise

Nach dem Auslösen von `INIT` mit der gültigen Objekt-ID der Alarmmaske initialisiert der Baustein die interne Kommunikation. Sobald über den Adapter `u8Priority` ein Ereignis eintrifft, wird der Dienst zur Prioritätsänderung ausgeführt. Der Baustein sendet die neue Priorität an den virtuellen Terminal und gibt nach Abschluss die Bestätigung über `CNF` mit dem Status und dem Ergebniswert aus. Gleichzeitig wird über den Adapter `u8OldPriority` die bisherige Priorität (sofern verfügbar) zurückgegeben.

Rückgabewerte (s16result):

- **VT_E_NO_ERR (0)** – Erfolgreich
- **VT_E_OVERFLOW (-6)** – Pufferüberlauf
- **VT_E_NOACT (-8)** – Befehl im aktuellen Zustand nicht möglich
- **VT_E_NO_INSTANCE (-21)** – Kein VT-Client verfügbar
- **VT_E_ISO_INSTANCE_INVALID (-129)** – Verbindungsinstanz ungültig
- **VT_E_NOT_ALIVE (-130)** – Instanz gültig, aber VT inaktiv

Technische Besonderheiten

- Der Baustein setzt auf das ISO 11783-6 Protokoll auf und nutzt unidirektionale Adapter für die Datenübertragung.
- Die Initialisierung erwartet eine gültige Objekt-ID; ein ungültiger Wert (ID_NULL) führt zu keinem Dienst.
- Die Fehlercodes sind standardisiert und ermöglichen eine einfache Diagnose auf VT-Ebene.
- Die interne Logik delegiert die eigentliche Prioritätsänderung an den eingebetteten FB `Q_Priority`.

Zustandsübersicht

Der Baustein besitzt keine explizit dargestellte Zustandsmaschine. Ablauf:

1. **INIT** → Initialisierung wartet auf gültige Objekt-ID.
2. Nach erfolgreicher Initialisierung → **INITO** aktiv.
3. Über den Adapter `u8Priority` wird die Anforderung (REQ) gestartet.
4. Nach Verarbeitung → **CNF** mit Ergebnis.

Anwendungsszenarien

- Ändern der Priorität einer Alarmmaske in einem virtuellen Terminal der Landtechnik (z. B. ISOBUS-konformes Steuergerät).

- Integration in eine Steuerung, die dynamisch unterschiedliche Alarmstufen setzen muss (z. B. von „niedrig“ auf „hoch“ bei einem kritischen Ereignis).

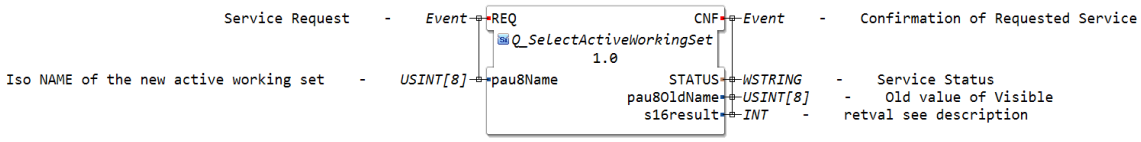
Vergleich mit ähnlichen Bausteinen

- **Q_Priority** (ohne AUS): Bietet die gleiche Kernfunktion, benötigt jedoch separate Datenleitungen für Priorität und Alter Wert. Der vorliegende Baustein kapselt die Ein- und Ausgabe über Adapter und erleichtert so die modulare Verwendung in hierarchischen Netzwerken.
- **Q_AlarmMask_Select**: Wählt eine Alarmmaske aus, ändert jedoch nicht die Priorität. Q_Priority_AUS ergänzt diese Funktionalität.

Fazit

Der Funktionsblock Q_Priority_AUS realisiert eine standardisierte, robuste Schnittstelle zur Prioritätsänderung von Alarmmasken nach ISO 11783-6. Durch die Adapter-basierte Datenübertragung und die klare Trennung von Initialisierung und Dienstausführung eignet er sich besonders für modulare, landwirtschaftliche Steuerungssysteme.

Q_SelectActiveWorkingSet



• — ○ • — ○ • — ○ •

Einleitung

Der **Q_SelectActiveWorkingSet** ist ein standardkonformer Funktionsbaustein zur Steuerung aktiver Arbeitsbereiche in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.64) Spezifikation für VT-Systeme ab Version 6.

Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung
- `REQ`: Arbeitsbereichswechsel-Anforderung

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Wechselbestätigung

Daten-Eingänge

- `QI` (BOOL, Standard `TRUE`): Quality-Eingang – `TRUE` aktiviert den Dienst
- `pau8Name` (USINT[8]): ISO-NAME des neuen aktiven Arbeitsbereichs

Daten-Ausgänge

- `QO` (BOOL): Quality-Ausgang – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde
- `STATUS` (STRING): Betriebsstatusmeldung
- `pau8OldName` (USINT[8]): ISO-NAME des vorherigen Arbeitsbereichs
- `s16result` (INT): ISO-konformer Ergebniscode

Funktionsweise

Quality-Handling (QI/QO): Der Eingang `QI` (BOOL, Standard `TRUE`) ist mit den Ereignissen `INIT` und `REQ` verknüpft; `TRUE` aktiviert den Dienst. Der Ausgang `QO` (BOOL) ist mit `INITO` und `CNF` verknüpft und zeigt mit `TRUE` an, dass der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` ohne Parameter
   - `INITO` bestätigt Betriebsbereitschaft
2. **Arbeitsbereichswechsel**:
   - `REQ` mit 8-Byte ISO-NAME des Zielarbeitsbereichs
   - Führt Kontextwechsel durch
   - `CNF` liefert Betriebsstatus und vorherigen Arbeitsbereich
3. **ISO-NAME Format**:
   - 8-Byte Array nach ISO 11783-5 Spezifikation

Technische Besonderheiten

✓ ISO 11783-6 konform (F.64) **✓ Exklusiv für VT Version 6+ ✓ Vollständiger Kontextwechsel** (Alle zugehörigen Ressourcen) **✓ Rückverfolgbarkeit** (Vorheriger Arbeitsbereich)

ISO-NAME Struktur

Byte	Inhalt
0-1	Herstellerkennung
2-3	Gerätekennung
4-5	Geräteinstanz
6-7	Arbeitsbereichskennung

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreicher Wechsel
−6	VT_E_OVERFLOW	Pufferüberlauf
−8	VT_E_NOACT	VT nicht bereit
−21	VT_E_NO_INSTANCE	Kein VT-Client verfügbar
−129	VT_E_ISO_INSTANCE_INVALID	Ungültige VT-Instanz
−130	VT_E_NOT_ALIVE	VT nicht aktiv

Anwendungsszenarien

- **Multi-Client-Systeme**: Gerätewechsel
- **Benutzerprofile**: Individuelle Arbeitsumgebungen
- **Prozessphasen**: Betriebsmodus-Wechsel
- **Diagnosemanagement**: Service-/Normalmodus

⚖️ **Vergleich mit ähnlichen Bausteinen**

Feature	Q_SelectActiveWorkingSet	VtContextSwitch	VtWorkspaceManager
ISO-Standard	✓	✖	✖
VT-Version	6+	Alle	Alle
ISO-NAME	✓	✖	✖
Rückmeldung	✓	✖	✓

Fazit

Der Q_SelectActiveWorkingSet-Baustein bietet zentrale Arbeitsbereichssteuerung:

- **Sicher**: ISO-konforme Identifikation
- **Effizient**: Kompletter Kontextwechsel
- **Transparent**: Vorheriger Zustand rückverfolgbar

Unverzichtbar für:

- Komplexe Multi-Client-Systeme
- Flexible Benutzerumgebungen
- Prozessorientierte VT-Anwendungen

Q_SelectColourMap

• — ○ • — ○ • — ○ •

Einleitung

Der **Q_SelectColourMap** ist ein standardkonformer Funktionsbaustein zur Verwaltung von Farbpaletten in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.60) Spezifikation für VT-Systeme ab Version 4.

Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung
- `REQ`: Farbpaletten-Wechselanforderung

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Wechselbestätigung

Daten-Eingänge

- `QI` (BOOL, Standard `TRUE`): Quality-Eingang – `TRUE` aktiviert den Dienst
- `u16ObjIdColourMap` (UINT): Farbpaletten-Objekt-ID oder 0xFFFF für Standardpalette

Daten-Ausgänge

- `QO` (BOOL): Quality-Ausgang – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde
- `STATUS` (STRING): Betriebsstatusmeldung
- `u16OldObjIdColourMap` (UINT): Vorherige Farbpaletten-ID
- `s16result` (INT): ISO-konformer Ergebniscode

Gültige Objekt-IDs

`u16ObjIdColourMap` adressiert (Bytes 2,3) laut Annex F.60:

- **ColourMap**: 39000 – 39999 (VT ab Version 4)
- **ColourPalette**: 45000 – 45999 (VT ab Version 6, Annex F.60)
- **FFFF16**: stellt die Standard-Farbtabelle wieder her (A.3)

Implementierungshinweis (VTClientHelper): `iso_is_colour_map_id` akzeptiert nur den Colour-Map-Bereich. Zudem wird ID_NULL (65535) von `cmd_select_colour_map_or_palette` abgefangen und mit VT_E_NO_ERR (0) beantwortet, ohne einen Befehl an das VT zu senden — der dokumentierte FFFF-Wert zum Wiederherstellen der Standardpalette funktioniert daher derzeit nicht (bekannte Abweichung).

Funktionsweise

Quality-Handling (QI/QO): Der Eingang `QI` (BOOL, Standard `TRUE`) ist mit den Ereignissen `INIT` und `REQ` verknüpft; `TRUE` aktiviert den Dienst. Der Ausgang `QO` (BOOL) ist mit `INITO` und `CNF` verknüpft und zeigt mit `TRUE` an, dass der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` ohne Parameter
   - `INITO` bestätigt Betriebsbereitschaft
2. **Palettenwechsel**:
   - `REQ` mit Ziel-Farbpaletten-ID
   - Globaler Farbmanagement-Wechsel
   - `CNF` liefert Betriebsstatus und vorherige Palette
3. **Spezialwerte**:
   - `0xFFFF`: Setzt Standard-ISO-Farbtabelle

Technische Besonderheiten

✓ **ISO 11783-6 konform** (F.60) ✓ **Globale Wirkung**: Beeinflusst alle Arbeitsbereiche ✓ **Lange Ausführungszeit**: Bei komplexen Paletten ✓ **Rückverfolgbarkeit**: Vorherige Palette speicherbar

Farbpaletten-Referenz

Wert	Bedeutung
0x0001-FFFE	Benutzerdefinierte Paletten
0xFFFF	ISO-Standardpalette (A.3)

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreicher Wechsel
−6	VT_E_OVERFLOW	Pufferüberlauf
−8	VT_E_NOACT	VT nicht bereit
−21	VT_E_NO_INSTANCE	Kein VT-Client verfügbar
−129	VT_E_ISO_INSTANCE_INVALID	Ungültige VT-Instanz
−130	VT_E_NOT_ALIVE	VT nicht aktiv

Anwendungsszenarien

- **Designwechsel**: Dynamische Farbschemata
- **Nachtmodus**: Augenschonende Paletten
- **Markenspezifika**: Corporate Design
- **Barrierefreiheit**: Hochkontrast-Modi

⚖️ **Vergleich mit ähnlichen Bausteinen**

Feature	Q_SelectColourMap	VtColorManager	VtPaletteControl
ISO-Standard	✓	✖	✖
Globalwirkung	✓	✖	✓
Standardpalette	✓	✖	✖
Rückmeldung	✓	✖	✓

Fazit

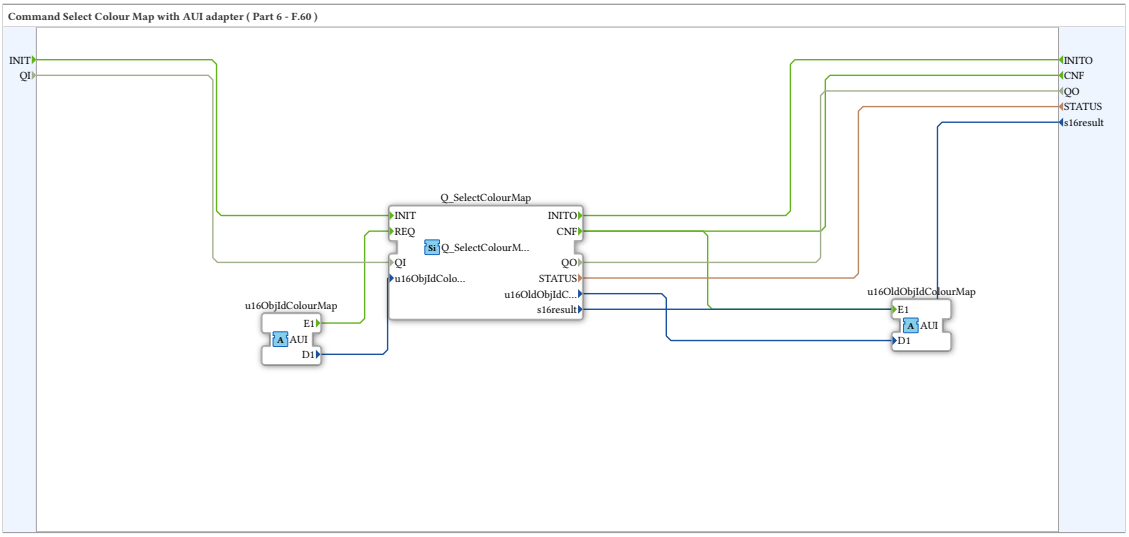
Der Q_SelectColourMap-Baustein bietet professionelle Farbverwaltung:

- **Umfassend**: Globale Farbsteuerung
- **Flexibel**: Benutzer- und Standardpaletten
- **Zuverlässig**: ISO-konforme Implementierung

Unverzichtbar für:

- Designorientierte Anwendungen
- Adaptive Visualisierungssysteme
- Markenspezifische Oberflächen

Q_SelectColourMap_AUI



• – ○ • – ○ • – ○ •

Einleitung

Der **Q_SelectColourMap_AUI** ist ein AUI-Adapter-Wrapper für den Baustein **Q_SelectColourMap** (ISO 11783-6, Teil 6 – F.60). Er ermöglicht die Auswahl der aktiven Farbpalette / Colour Map auf einem ISOBUS Terminal über AUI-Adapter-Schnittstellen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
`INIT`	`EInit`	Service-Initialisierung

Ereignis-Ausgänge

Name	Typ	Kommentar
`INITO`	`EInit`	Bestätigung der Initialisierung
`CNF`	`Event`	Bestätigung des angeforderten Dienstes

Daten-Eingänge

Außer `QI` gibt es keine direkten Dateneingänge – die Farbpaletten-IDs werden über die AUI-Adapter übertragen.

Name	Typ	Initialwert	Kommentar
`QI`	`BOOL`	`TRUE`	Quality Input – TRUE aktiviert den Dienst

Daten-Ausgänge

Name	Typ	Kommentar
`QO`	`BOOL`	Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde
`STATUS`	`STRING`	Dienststatus
`s16result`	`INT`	Rückgabewert des Befehls

Adapter

Typ	Name	Richtung	Kommentar
`adapter::types::unidirectional::AUI6ObjIdColourMap`	`u16ObjIdColourMap`	Socket (Eingang)	Neue Farbpaletten-Object-ID (39000–39999 für ColourMap, 45000–45999 für ColourPalette, ID_NULL / 0xFFFF für Standard-Farbtabelle)
`adapter::types::unidirectional::AUI6OldObjIdColourMap`	`u16OldObjIdColourMap`	Plug (Ausgang)	Bisherige Farbpaletten-Object-ID

Qualitätssignale QI/QO: Der Dateneingang `QI` (BOOL, Initialwert `TRUE`) ist dem Ereignis `INIT` zugeordnet und wird an `QI` des internen Bausteins `Q_SelectColourMap` durchgereicht. Der Datenausgang `QO` (BOOL) wird von `Q_SelectColourMap.QO` übernommen und ist den Ereignissen `INITO` und `CNF` zugeordnet.

Funktionsweise

Der Baustein kapselt den inneren FB `Q_SelectColourMap` (*isobus::UT::Q::Q_SelectColourMap*). Bei einem Ereignis an `u16ObjIdColourMap.E1` wird das `REQ`-Ereignis des internen FBs mit der neuen ID aus `u16ObjIdColourMap.D1` aufgerufen. Nach erfolgreicher Umschaltung wird über `u16OldObjIdColourMap.E1` und `CNF` die alte ID (`u16OldObjIdColourMap.D1`) ausgegeben.

Technische Besonderheiten

- **AUI-Adapter-Kopplung:** Adapterbasierte Umschaltung von Farbpaletten über den Socket `u16ObjIdColourMap`.
- **Wertebereiche gemäß ISO 11783-6 F.60:**
  - `ColourMap`: 39000–39999 (ab VT Version 4)
  - `ColourPalette`: 45000–45999 (ab VT Version 6)
  - `ID_NULL` (`0xFFFF` / 65535): Wiederherstellen der Standard-Farbtabelle.
- **Implementierungseinschränkungen (VTClientHelper):**
  - `VTClientHelper::iso_is_colour_map_id` akzeptiert aktuell nur den Bereich 39000–39999 (Farbpaletten-IDs 45000–45999 werden abgelehnt).
  - `cmd_select_colour_map_or_palette` verschluckt `ID_NULL` (`0xFFFF`), gibt `VT_E_NO_ERR` (0) zurück und sendet keinen Befehl an den VT, weshalb das Wiederherstellen der Standard-Farbtabelle derzeit nicht funktioniert.

Zustandsübersicht

Delegiert alle Aktionen an `Q_SelectColourMap`.

Anwendungsszenarien

- Dynamische Umschaltung von Farbschemata (z. B. Tag-/Nachtmodus) im ISOBUS Terminal per AUI-Adapter.

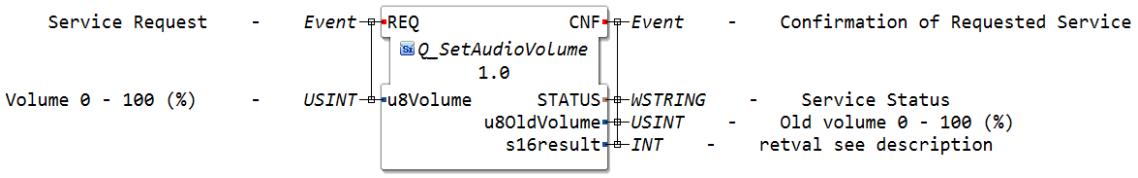
Vergleich mit ähnlichen Bausteinen

Erweitert `Q_SelectColourMap` um AUI-Adapter-Sockets und -Plugs zur einfachen Einbindung in adapterbasierte Architekturen.

Fazit

Q_SelectColourMap_AUI ist der adapterbasierte Wrapper für Farbpaletten-Umschaltungen in ISOBUS VT-Anwendungen.

Q_SetAudioVolume

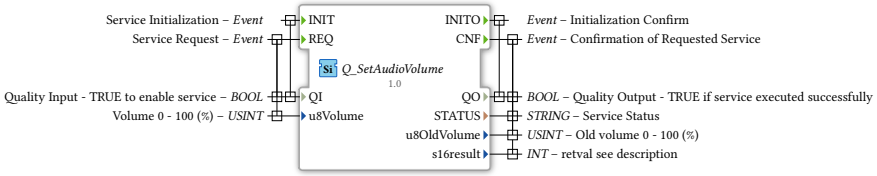


• – ○ • – ○ • – ○ •

Einleitung

Der **Q_SetAudioVolume** ist ein standardkonformer Funktionsbaustein zur Lautstärkeregelung in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.12) Spezifikation für VT-Audiosteuerung.

Der Befehl F.12 adressiert direkt das **Audiogerät** des VT — er referenziert **kein Objekt**, daher besitzt dieser Baustein keinen Objekt-ID-Eingang. Die Nachricht hat eine feste Länge von 8 Byte; Byte 2 enthält den Lautstärkeprozentsatz (0 % – 100 %).



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung
- `REQ`: Lautstärkeänderungs-Anforderung

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Änderungsbestätigung

Daten-Eingänge

- `QI` (BOOL, Standard `TRUE`): Quality-Eingang – `TRUE` aktiviert den Dienst
- `u8Volume` (USINT): Lautstärkewert (0-100%)

Daten-Ausgänge

- `QO` (BOOL): Quality-Ausgang – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde
- `STATUS` (STRING): Betriebsstatusmeldung
- `u8OldVolume` (USINT): Vorherige Lautstärkeeinstellung
- `s16result` (INT): ISO-konformer Ergebniscode

Funktionsweise

Quality-Handling (QI/QO): Der Eingang `QI` (BOOL, Standard `TRUE`) ist mit den Ereignissen `INIT` und `REQ` verknüpft; `TRUE` aktiviert den Dienst. Der Ausgang `QO` (BOOL) ist mit `INITO` und `CNF` verknüpft und zeigt mit `TRUE` an, dass der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` ohne Parameter
   - `INITO` bestätigt Betriebsbereitschaft
2. **Lautstärkeregelung**:
   - `REQ` mit gewünschtem Lautstärkewert
   - Ändert die globale VT-Lautstärke
   - `CNF` liefert Betriebsstatus und vorherigen Wert
3. **Wertebereich**:
   - 0%: Stummgeschaltet
   - 100%: Maximale Lautstärke

Technische Besonderheiten

✓ **ISO 11783-6 konform** (F.12) ✓ **Globale Audio-Steuerung** ✓ **Prozentuale Regelung** (0-100%) ✓ **Rückverfolgbarkeit** (Vorherige Einstellung)

Lautstärke-Referenz

Wert	Bedeutung
0	Stumm
50	Mittlere Lautstärke
100	Maximale Lautstärke

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreiche Änderung
−6	VT_E_OVERFLOW	Pufferüberlauf
−8	VT_E_NOACT	VT nicht bereit
−21	VT_E_NO_INSTANCE	Kein VT-Client verfügbar
−128	VT_E_HANDLE_INVALID	Ungültige Audio-Konfiguration
−129	VT_E_ISO_INSTANCE_INVALID	Ungültige VT-Instanz
−130	VT_E_NOT_ALIVE	VT nicht aktiv

Anwendungsszenarien

- **Alarmmanagement**: Regelbare Warnsignale
- **Sprachausgabe**: Lautstärkeanpassung
- **Bedienkomfort**: Benutzerdefinierte Audioeinstellungen
- **Nachtbetrieb**: Automatische Lautstärkereduzierung

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	Q_SetAudioVolume	VtAudioControl	VtSoundManager
ISO-Standard	✓	✖	✖
Wertebereich	0-100%	0-255	0-10 Stufen

Globalwirkung	✓	✓	✖
Rückmeldung	✓	✖	✓

Fazit

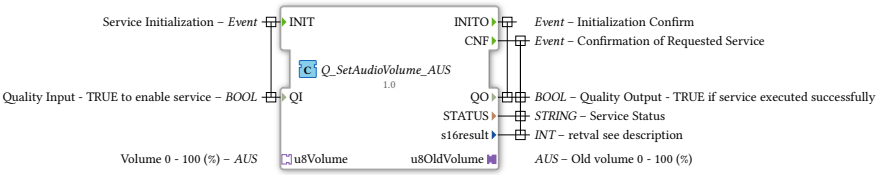
Der Q_SetAudioVolume-Baustein bietet präzise Audiosteuerung:

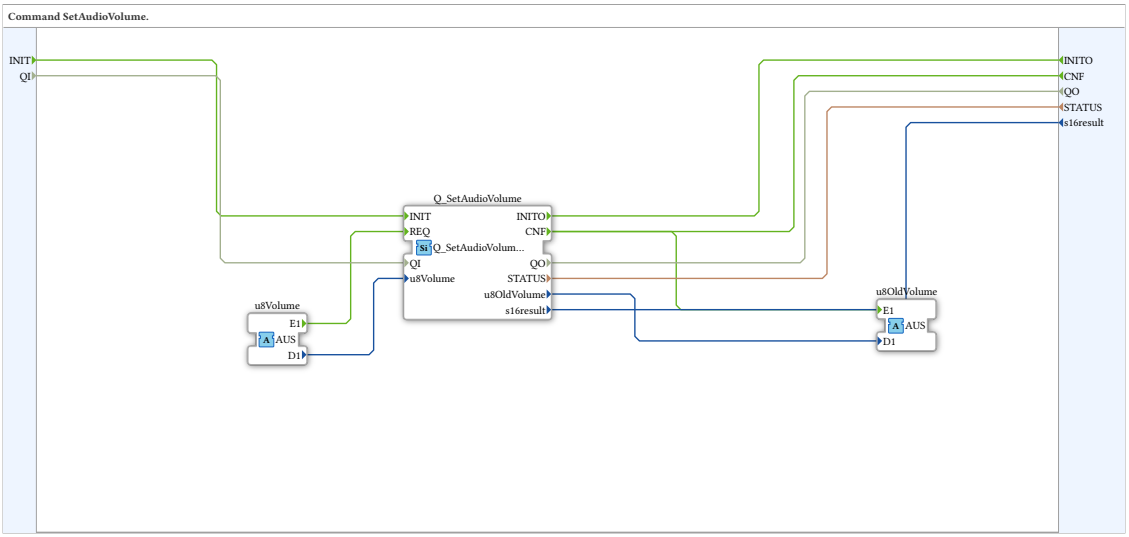
- **Intuitiv**: Prozentuale Regelung
- **Zuverlässig**: ISO-konforme Implementierung
- **Flexibel**: Für alle Audioausgaben

Unverzichtbar für:

- Professionelle Alarmysteme
- Sprachgesteuerte Anwendungen
- Komfortorientierte Bedienoberflächen

Q_SetAudioVolume_AUS





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock Q_SetAudioVolume_AUS ist ein zusammengesetzter Baustein (Composite FB), der den Befehl „SetAudioVolume“ gemäß ISO 11783-6 auf einem Virtual Terminal (VT) ausführt. Er kapselt den Basisbaustein Q_SetAudioVolume und stellt eine adapterbasierte Schnittstelle zur Verfügung, um die Lautstärke zu setzen und den alten Wert zurückzulesen. Der FB initialisiert sich über den Ereigniseingang INIT und bestätigt dies mit INITO. Anschließend kann über den u8Volume-Adapter eine neue Lautstärke übergeben werden, woraufhin der Befehl ausgelöst und das Ergebnis über CNF mit STATUS und s16result ausgegeben wird.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Dienstinitialisierung. Löst die Initialisierung des internen Bausteins aus. Verknüpft mit `QI`.

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der erfolgreichen Initialisierung. Verknüpft mit `QO`.
- **CNF** (Event): Bestätigung des angeforderten Dienstes. Wird nach dem Setzen der Lautstärke ausgegeben. Enthält die Ausgabedaten STATUS und s16result. Verknüpft mit `QO`.

Daten-Eingänge

Der FB besitzt außer `QI` keine direkten Daten-Eingänge. Die Eingabe der Lautstärke erfolgt über den Adapter u8Volume.

Name	Typ	Initialwert	Kommentar
`QI`	`BOOL`	`TRUE`	Quality Input – TRUE aktiviert den Dienst

Daten-Ausgänge

- **QO** (BOOL): Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde.
- **STATUS** (STRING): Service-Statusmeldung.
- **s16result** (INT): Rückgabewert des Befehls. Mögliche Werte:
  - 0 (VT_E_NO_ERR): Erfolg.
  - −6 (VT_E_OVERFLOW): Pufferüberlauf.
  - −8 (VT_E_NOACT): Befehl im aktuellen Zustand nicht möglich.
  - −21 (VT_E_NO_INSTANCE): Kein VT-Client verfügbar.
  - −129 (VT_E_ISO_INSTANCE_INVALID): Verbindungsinstanz ungültig.
  - −128 (VT_E_HANDLE_INVALID): Handle ungültig.
  - −130 (VT_E_NOT_ALIVE): VT nicht aktiv.

Adapter

- **u8Volume** (Socket): Unidirektionaler Adapter vom Typ AUS. Empfängt die gewünschte Lautstärke als Wert zwischen 0 und 100 %.
- **u8OldVolume** (Plug): Unidirektionaler Adapter vom Typ AUS. Gibt die vor dem Befehl gültige Lautstärke (0–100 %) aus.

Qualitätssignale QI/QO: Der Dateneingang `QI` (BOOL, Initialwert `TRUE`) ist dem Ereignis `INIT` zugeordnet und wird an `QI` des internen Bausteins `Q_SetAudioVolume` durchgereicht. Der Datenausgang `QO` (BOOL) wird von `Q_SetAudioVolume.QO` übernommen und ist den Ereignissen `INITO` und `CNF` zugeordnet.

Funktionsweise

Nach dem Start wird der FB über den INIT-Eingang initialisiert. Der interne Baustein Q_SetAudioVolume wird bereitgemacht und INITO signalisiert die Bereitschaft. Sobald ein neuer Lautstärkewert über den u8Volume-Adapter anliegt (Ereignis E1 des Adapters), wird der Befehl zum Ändern der Lautstärke an das VT gesendet. Nach Abschluss des Vorgangs wird CNF ausgelöst, und die Ausgabevariablen STATUS und s16result enthalten das Ergebnis. Parallel wird der vorherige Lautstärkewert über den Adapter u8OldVolume ausgegeben, sodass eine externe Logik den alten Wert auslesen kann.

Die möglichen Fehlercodes entsprechen den ISO 11783-6-Definitionen und decken typische VT-Fehlersituationen ab.

Technische Besonderheiten

- Der FB ist als Composite-Baustein realisiert, der den Basisfunktionsblock Q_SetAudioVolume aus der isobus-Bibliothek verwendet.
- Die Schnittstelle nutzt ausschließlich Adapter für die Datenübergabe, wodurch eine lose Kopplung und flexible Verdrahtung erreicht wird.
- Die Adapter sind vom Typ „unidirectional::AUS“ und ermöglichen eine ereignisgesteuerte Datenweitergabe.
- Der FB ist für den Einsatz in ISOBUS-Systemen nach ISO 11783-6 ausgelegt.

Zustandsübersicht

Da es sich um einen zusammengesetzten Baustein mit internem FB handelt, ist keine explizite Zustandsmaschine auf oberster Ebene vorhanden. Der FB durchläuft folgende Phasen:

1. Initialisierung (INIT empfangen → INITO senden)
2. Bereit (wartet auf Lautstärkeänderung über u8Volume-Adapter)

3. Verarbeitung (interner Befehl wird ausgeführt)
4. Ergebnisausgabe (CNF mit STATUS/s16result)

Ein detaillierter Zustandsautomat ist im internen Baustein Q_SetAudioVolume implementiert.

Anwendungsszenarien

- Landwirtschaftliche Maschinensteuerung: Anpassung der Lautstärke von Sprachausgaben oder Warntönen des Virtual Terminals.
- Integration in ISOBUS-konforme Applikationen, die über das Bedienterminal die Audioeinstellungen verändern.
- Kombination mit anderen VT-Befehlsbausteinen wie Q_SelectAudioOutput, Q_Beep etc.

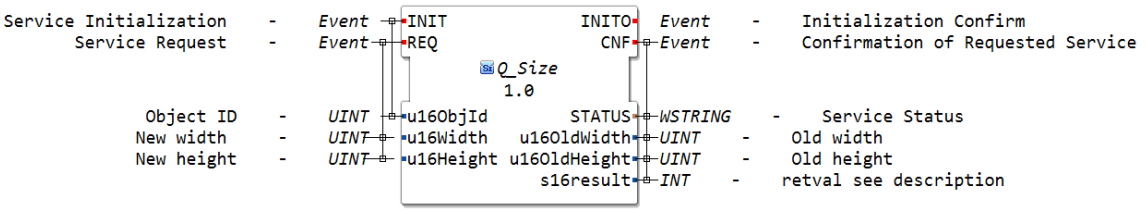
Vergleich mit ähnlichen Bausteinen

Der Baustein Q_SetAudioVolume_AUS unterscheidet sich von einem direkten Q_SetAudioVolume durch die Verwendung von Adaptern zur Datenübergabe. Während Q_SetAudioVolume möglicherweise direkte Daten-Eingänge (z. B. u8Volume als VarDeclaration) besitzt, erlaubt die Adaptervariante eine entkoppelte, ereignisgesteuerte Kommunikation. Ähnliche Adapter-Bausteine existieren für andere VT-Befehle, sodass ein einheitliches Entwurfsmuster entsteht.

Fazit

Q_SetAudioVolume_AUS ist ein spezialisierter Funktionsblock zur Lautstärkeregelung eines ISOBUS-VT. Durch die adapterbasierte Schnittstelle wird eine klare Trennung von Steuerungs- und Datenebene erreicht. Der Baustein liefert vollständige Rückmeldungen über den Erfolg des Befehls und eignet sich für den zuverlässigen Einsatz in landwirtschaftlichen Steuerungen.

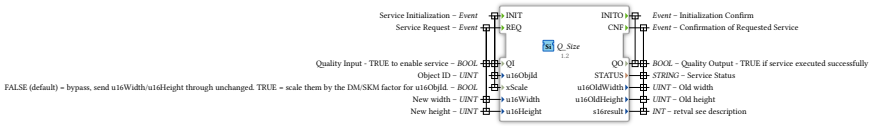
Q_Size



• — ○ • — ○ • — ○ •

Einleitung

Der **Q_Size** ist ein standardkonformer Funktionsbaustein zur Größenänderung von Objekten in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.18) Spezifikation für VT-Objekte.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit Objekt-ID und xScale)
- `REQ`: Größenänderungs-Anforderung

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Änderungsbestätigung

Daten-Eingänge

- `QI` (BOOL, Standard `TRUE`): Quality-Eingang – `TRUE` aktiviert den Dienst
- `u16ObjId` (UINT): Objekt-ID (16-bit)
- `xScale` (BOOL): Bei TRUE werden die Werte `u16Width`/`u16Height` vor dem Senden mit dem DM/SKM-Faktor für `u16ObjId` skaliert; bei FALSE (Standard) werden sie unverändert durchgereicht.

- `u16Width` (UINT): Neue Breite in Pixeln
- `u16Height` (UINT): Neue Höhe in Pixeln

Daten-Ausgänge

- `QO` (BOOL): Quality-Ausgang – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde
- `STATUS` (STRING): Betriebsstatusmeldung
- `u16OldWidth` (UINT): Vorherige Breite
- `u16OldHeight` (UINT): Vorherige Höhe
- `s16result` (INT): ISO-konformer Ergebniscode

Gültige Objekt-IDs

`u16ObjId` — gültige Objekttypen (Anhang F.18, Objekte mit Größen-Attribut): Container (3000–3999), Button (6000–6999), Input Boolean Field (7000–7999), Input String Field (8000–8999), Input Number Field (9000–9999), Input List Field (10000–10999), Output String Field (11000–11999), Output Number Field (12000–12999), Output Line (13000–13999), Output Rectangle (14000–14999), Output Ellipse (15000–15999), Output Polygon (16000–16999), Meter (17000–17999), Linear Bar Graph (18000–18999), Arched Bar Graph (19000–19999), Output List Object (37000–37999), Animation Object (44000–44999).

ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei `INIT` den Baustein.

Funktionsweise

Quality-Handling (QI/QO): Der Eingang `QI` (BOOL, Standard `TRUE`) ist mit den Ereignissen `INIT` und `REQ` verknüpft; `TRUE` aktiviert den Dienst. Der Ausgang `QO` (BOOL) ist mit `INITO` und `CNF` verknüpft und zeigt mit `TRUE` an, dass der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` mit Zielobjekt-ID
   - `INITO` bestätigt Betriebsbereitschaft
2. **Größenänderung**:
   - `REQ` mit neuen Abmessungen
   - Passt die Objektgeometrie dynamisch an
   - `CNF` liefert Betriebsstatus und vorherige Maße
3. **Maßeinheit**:
   - Alle Werte in Bildschirmpixeln

Technische Besonderheiten

✓ ISO 11783-6 konform (F.18) **✓ Dynamische Größenanpassung** (Echtzeit-Update) **✓ Umfassende Objektunterstützung ✓ Rückverfolgbarkeit** (Vorherige Abmessungen)

Größenreferenz

Parameter	Wertebereich	Beschreibung
Width	1-65535 Pixel	Horizontale Ausdehnung
Height	1-65535 Pixel	Vertikale Ausdehnung

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreiche Änderung
−6	VT_E_OVERFLOW	Pufferüberlauf
−8	VT_E_NOACT	VT nicht bereit
−21	VT_E_NO_INSTANCE	Kein VT-Client verfügbar
−129	VT_E_ISO_INSTANCE_INVALID	Ungültige VT-Instanz
−130	VT_E_NOT_ALIVE	VT nicht aktiv

Anwendungsszenarien

- **Responsive Designs**: Anpassung an Displaygrößen
- **Zoom-Funktionen**: Dynamische Skalierung
- **Layoutmanager**: Automatische Größenoptimierung
- **Animationssteuerung**: Größenänderungseffekte

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	Q_Size	VtResizeObject	VtGeometryManager
ISO-Standard	✓	✖	✖
Echtzeit-Update	✓	✓	✖
2D-Steuerung	✓	✓	✓
Rückmeldung	✓	✖	✓

Fazit

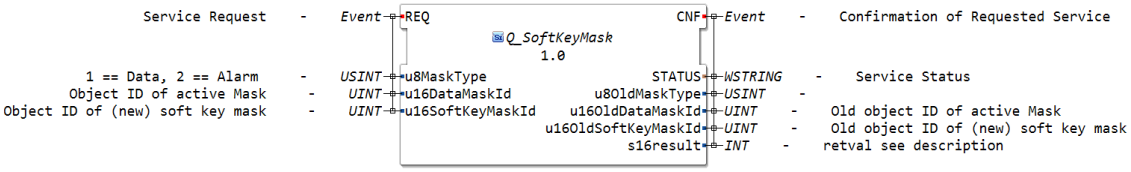
Der Q_Size-Baustein bietet präzise Größensteuerung:

- **Flexibel**: Pixelgenaue Anpassung
- **Performant**: Sofortige Darstellungsaktualisierung
- **Zuverlässig**: ISO-konforme Implementierung

Unverzichtbar für:

- Adaptive Benutzeroberflächen
- Dynamische Visualisierungen
- Professionelle VT-Anwendungen

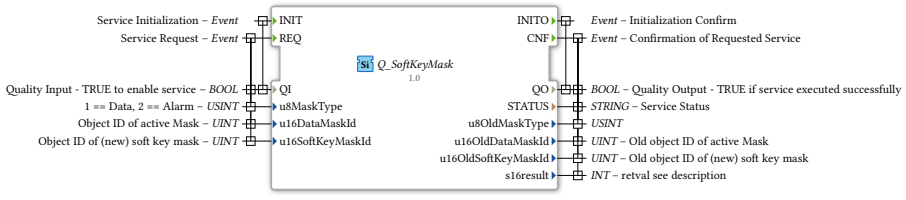
Q_SoftKeyMask



• — ○ • — ○ • — ○ •

Einleitung

Der **Q_SoftKeyMask** ist ein standardkonformer Funktionsbaustein zur Verwaltung von Softkey-Masken in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.36) Spezifikation für VT-Systeme.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung
- `REQ`: Maskenwechsel-Anforderung

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung
- `CNF`: Wechselbestätigung

Daten-Eingänge

- `QI` (BOOL, Standard `TRUE`): Quality-Eingang – `TRUE` aktiviert den Dienst
- `u8MaskType` (USINT): Maskentyp (1=Daten, 2=Alarm)
- `u16DataMaskId` (UINT): Objekt-ID der aktiven Maske
- `u16SoftKeyMaskId` (UINT): Objekt-ID der neuen Softkey-Maske

Daten-Ausgänge

- `QO` (BOOL): Quality-Ausgang – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde
- `STATUS` (STRING): Betriebsstatusmeldung
- `u8OldMaskType` (USINT): Vorheriger Maskentyp
- `u16OldDataMaskId` (UINT): Vorherige Datenmasken-ID
- `u16OldSoftKeyMaskId` (UINT): Vorherige Softkey-Masken-ID
- `s16result` (INT): ISO-konformer Ergebniscode

Funktionsweise

Quality-Handling (QI/QO): Der Eingang `QI` (BOOL, Standard `TRUE`) ist mit den Ereignissen `INIT` und `REQ` verknüpft; `TRUE` aktiviert den Dienst. Der Ausgang `QO` (BOOL) ist mit `INITO` und `CNF` verknüpft und zeigt mit `TRUE` an, dass der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` ohne Parameter
   - `INITO` bestätigt Betriebsbereitschaft
2. **Maskenwechsel**:
   - `REQ` mit Maskentyp, Datenmasken-ID und neuer Softkey-Maske
   - Verknüpft Softkeys mit der aktiven Maske
   - `CNF` liefert Betriebsstatus und vorherige Konfiguration
3. **Maskentypen**:
   - `1`: Datenmasken-Verknüpfung
   - `2`: Alarmmasken-Verknüpfung

Gültige Objekt-IDs

Der Baustein adressiert **zwei verschiedene Objekt-IDs** mit jeweils eigenen Gültigkeitsregeln:

Gültige Werte für u16DataMaskId (Daten-/Alarmmaske)

Der Change Soft Key Mask-Befehl (Annex F.36) ordnet einer Daten- oder Alarmmaske eine Softkey-Maske zu. Gültig ist `u16DataMaskId`, wenn er in einen der folgenden ISO 11783-6-ID-Bereiche fällt:

Objekttyp	ID-Bereich
DataMask	1000 – 1999
AlarmMask	2000 – 2999

Gültige Werte für u16SoftKeyMaskId (Softkey-Maske)

Gültig ist `u16SoftKeyMaskId`, wenn er in den ISO 11783-6-ID-Bereich einer Softkey-Maske fällt:

Objekttyp	ID-Bereich
SoftKeyMask	4000 – 4999

Hinweis: ID_NULL (65535) ist kein Befehlsziel, kann aber in INIT zur Deaktivierung des Bausteins verwendet werden. Jede Daten-/Alarmmasken-ID außerhalb der genannten Bereiche oder jede Softkey-Masken-ID, die keine SoftKeyMask ist, ist unabhängig voneinander ungültig.

Technische Besonderheiten

✓ **ISO 11783-6 konform** (F.36) ✓ **Duale Maskentypen** (Daten/Alarme) ✓ **Dynamische Neuzuordnung** ✓ **Rückverfolgbarkeit** (Vorherige Konfiguration)

Maskentyp-Referenz

Wert	Typ	Beschreibung
1	Data	Normale Datenmaske
2	Alarm	Alarmmasken-Verknüpfung

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreicher Wechsel
−6	VT_E_OVERFLOW	Pufferüberlauf
−8	VT_E_NOACT	VT nicht bereit
−21	VT_E_NO_INSTANCE	Kein VT-Client verfügbar
−129	VT_E_ISO_INSTANCE_INVALID	Ungültige VT-Instanz
−130	VT_E_NOT_ALIVE	VT nicht aktiv

Anwendungsszenarien

- **Kontextsensitive Softkeys**: Funktionswechsel
- **Alarmmanagement**: Spezielle Alarm-Softkeys
- **Maskenhierarchien**: Dynamische Tastenbelegungen
- **Benutzerführung**: Adaptive Bedienoberflächen

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	Q_SoftKeyMask	VtKeyMapping	VtMaskLinker
ISO-Standard	✓	✖	✖
Maskentypen	2 (Data/Alarm)	1 (Universal)	1 (Data)
Rückmeldung	✓	✖	✓
Echtzeitwechsel	✓	✓	✖

Fazit

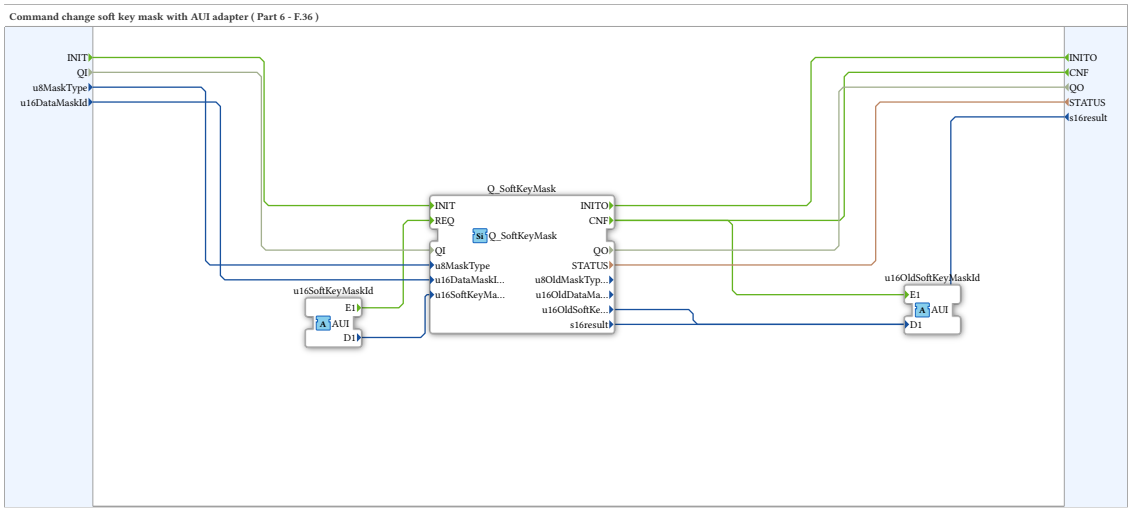
Der Q_SoftKeyMask-Baustein bietet flexible Softkey-Steuerung:

- **Kontextsensitiv**: Unterschiedliche Maskentypen
- **Dynamisch**: Sofortige Neuzuordnung
- **Zuverlässig**: ISO-konforme Implementierung

Unverzichtbar für:

- Professionelle Bedienkonzepte
- Komplexe Maskenhierarchien
- Adaptive VT-Anwendungen

Q_SoftKeyMask_AUI



• – ○ • – ○ • – ○ •

Einleitung

Der **Q_SoftKeyMask_AUI** ist ein AUI-Adapter-Wrapper für den Baustein **Q_SoftKeyMask** (ISO 11783-6, Teil 6 – F.36). Er ermöglicht das Umschalten der Softkey-Maske auf einem ISOBUS Terminal über AUI-Adapter-Schnittstellen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
`INIT`	`EInit`	Service-Initialisierung

Ereignis-Ausgänge

Name	Typ	Kommentar
`INITO`	`EInit`	Bestätigung der Initialisierung
`CNF`	`Event`	Bestätigung des angeforderten Dienstes

Daten-Eingänge

Name	Typ	Kommentar
`QI`	`BOOL`	Quality Input – TRUE aktiviert den Dienst
`u8MaskType`	`USINT`	Maskentyp (1: DataMask, 2: AlarmMask)
`u16DataMaskId`	`UINT`	Object-ID der zugehörigen Data/AlarmMask

Daten-Ausgänge

Name	Typ	Kommentar
`QO`	`BOOL`	Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde
`STATUS`	`STRING`	Dienststatus
`s16result`	`INT`	Rückgabewert des Befehls

Adapter

Typ	Name	Richtung	Kommentar
`adapter::types::unidirectional::AUI16SoftKeyMaskId`		Socket (Eingang)	Neue Softkey-Masken-ID
`adapter::types::unidirectional::AUI16OldSoftKeyMaskId`		Plug (Ausgang)	Bisherige Softkey-Masken-ID

Qualitätssignale QI/QO: Der Dateneingang `QI` (BOOL, Initialwert `TRUE`) ist dem Ereignis `INIT` zugeordnet und wird an `QI` des internen Bausteins `Q_SoftKeyMask` durchgereicht. Der Datenausgang `QO` (BOOL) wird von `Q_SoftKeyMask.QO` übernommen und ist den Ereignissen `INITO` und `CNF` zugeordnet.

Funktionsweise

Der Baustein kapselt den internen FB `Q_SoftKeyMask` (*isobus::UT::Q::Q_SoftKeyMask*). Ein Ereignis an `u16SoftKeyMaskId.E1` löst `REQ` am internen FB aus, der die Maskenumschaltung initiiert. Nach der Ausführung wird die alte Softkey-Masken-ID über `u16OldSoftKeyMaskId.D1` und `u16OldSoftKeyMaskId.E1` ausgegeben.

Technische Besonderheiten

- **Adapter-basierte Softkey-Steuerung:** Flexibles Umschalten von Softkey-Layouts per AUI-Adapter.
- **ISOBUS-Konformität:** Basiert auf ISO 11783-6 F.36.

Zustandsübersicht

Delegiert alle Aktionen an `Q_SoftKeyMask`.

Anwendungsszenarien

- Dynamisches Wechseln von Softkey-Belegungen in ISOBUS Terminals über AUI-Adapter.

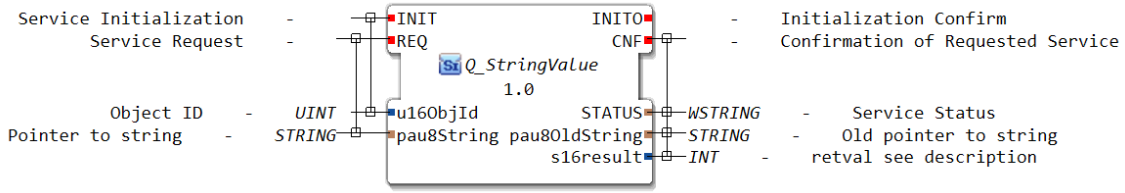
Vergleich mit ähnlichen Bausteinen

Bietet im Vergleich zu `Q_SoftKeyMask` eine unidirektionale AUI-Adapter-Schnittstelle.

Fazit

Q_SoftKeyMask_AUI ist der adapterbasierte Wrapper für Softkey-Masken-Umschaltungen in ISOBUS-Anwendungen.

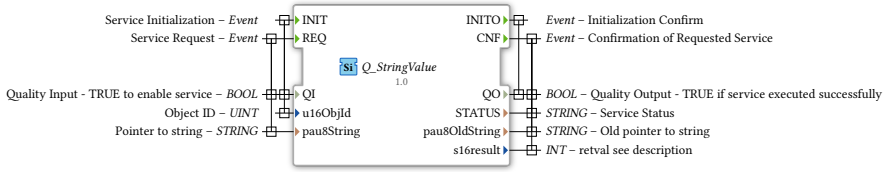
Q_StringValue



• – ○ • – ○ • – ○ •

Einleitung

Der **Q_StringValue** ist ein standardkonformer Funktionsbaustein zur dynamischen Textaktualisierung in Virtual Terminals, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 implementiert die ISO 11783-6 (Teil 6 - F.24) Spezifikation für VT-Textobjekte.



Schnittstellenstruktur

Ereignis-Eingänge
- `INIT`: Initialisierungsanforderung (mit Objekt-ID)
- `REQ`: Textänderungs-Anforderung

Ereignis-Ausgänge
- `INITO`: Initialisierungsbestätigung
- `CNF`: Änderungsbestätigung

Daten-Eingänge
- `QI` (BOOL, Standard `TRUE`): Quality-Eingang – `TRUE` aktiviert den Dienst
- `u16ObjId` (UINT): Textobjekt-ID (16-bit)
- `pau8String` (STRING): Neuer Textwert (nullterminiert)

Daten-Ausgänge
- `QO` (BOOL): Quality-Ausgang – `TRUE`, wenn der Dienst erfolgreich ausgeführt wurde
- `STATUS` (STRING): Betriebsstatusmeldung
- `pau8OldString` (STRING): Vorheriger Textwert
- `s16result` (INT): ISO-konformer Ergebniscode

Gültige Objekt-IDs

`u16ObjId` — gültige Objekttypen (Anhang F.24, Objekte mit String-Wert-Attribut): Input String Field (8000–8999), Output String Field (11000–11999), String Variable (22000–22999), Input Attributes (26000–26999).

Datenlänge / Transport-Protokoll: Das F.24-Kommando hat eine **variable Datenlänge** (Kommando-Frame: Objekt-ID, Anzahl String-Bytes, String-Wert). Strings, die in eine einzelne CAN-Nachricht passen, werden direkt gesendet; längere Strings werden über das ISO 11783 **Transport-Protokoll** übertragen. Der übertragene String darf laut Anhang F.24 kürzer sein als das Längen-Attribut des Zielobjekts (das VT füllt mit Leerzeichen auf), darf es aber nicht überschreiten.

ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei `INIT` den Baustein.

Funktionsweise

Quality-Handling (QI/QO): Der Eingang `QI` (BOOL, Standard `TRUE`) ist mit den Ereignissen `INIT` und `REQ` verknüpft; `TRUE` aktiviert den Dienst. Der Ausgang `QO` (BOOL) ist mit `INITO` und `CNF` verknüpft und zeigt mit `TRUE` an, dass der Dienst erfolgreich ausgeführt wurde.

1. **Initialisierung**:
   - `INIT` mit Zielobjekt-ID
   - `INITO` bestätigt Betriebsbereitschaft
2. **Textaktualisierung**:
   - `REQ` mit neuem Textstring
   - Aktualisiert das VT-Textobjekt sofort
   - `CNF` liefert Betriebsstatus und vorherigen Text
3. **String-Anforderungen**:
   - Maximal `ISO_VTC_CMD_STR_MAX_LENGTH` inkl. Nullterminierung
   - Muss nullterminiert sein

Technische Besonderheiten

✓ ISO 11783-6 konform (F.24) **✓ Echtzeit-Textupdate** (FIFO-basiert) **✓ Nullterminierung erforderlich ✓ Rückverfolgbarkeit** (Vorheriger Textwert)

String-Format

Parameter	Anforderung
Maximallänge	ISO_VTC_CMD_STR_MAX_LENGTH

Terminierung	‚\0‘ am Ende obligatorisch
Zeichensatz	ISO-8859-1 (Latin-1) empfohlen

Rückgabecodes (s16result)

Code	Konstante	Bedeutung
0	VT_E_NO_ERR	Erfolgreiche Aktualisierung
−6	VT_E_OVERFLOW	String zu lang oder Puffer voll
−8	VT_E_NOACT	VT nicht bereit
−21	VT_E_NO_INSTANCE	Kein VT-Client verfügbar
−129	VT_E_ISO_INSTANCE_INVALID	Ungültige VT-Instanz
−130	VT_E_NOT_ALIVE	VT nicht aktiv

Anwendungsszenarien

- **Dynamische Anzeigen**: Messwerte, Statusmeldungen
- **Sprachumschaltung**: Mehrsprachige Texte
- **Prozessvisualisierung**: Echtzeit-Daten
- **Benutzerkommunikation**: Dialoge und Hinweise

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	Q_StringValue	VtTextUpdate	VtStringManager
ISO-Standard	✓	✖	✖
FIFO-Update	✓	✖	✓
Längenprüfung	✓	✓	✖
Rückmeldung	✓	✖	✓

🛠️ Zugehörige Übungen

- Uebung_012c

Fazit

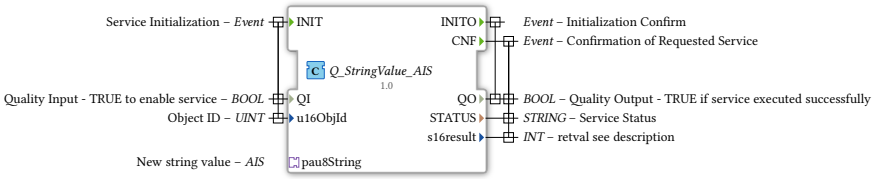
Der Q_StringValue-Baustein bietet effiziente Textverwaltung:

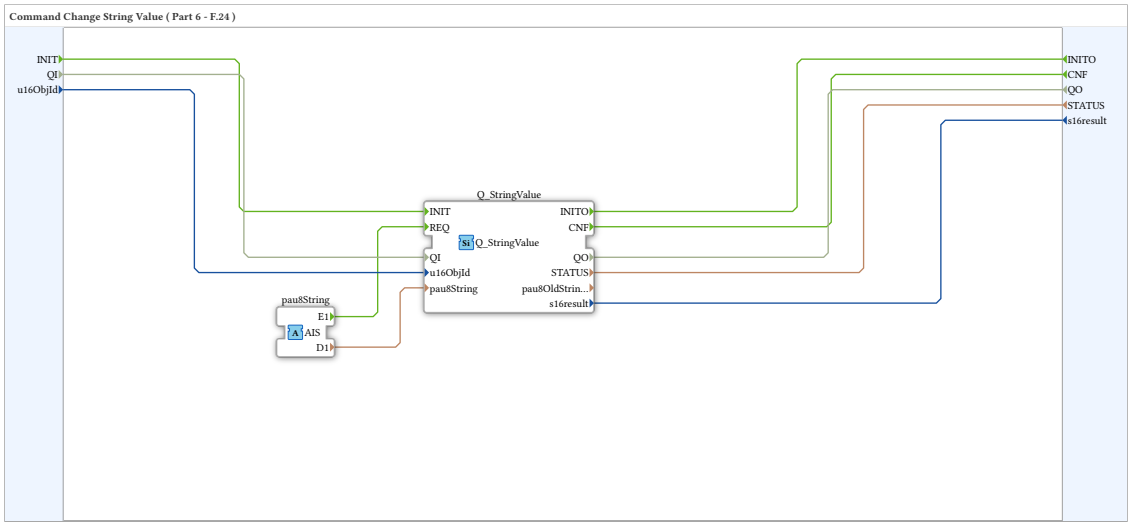
- **Performant**: Direkte FIFO-Aktualisierung
- **Sicher**: Integrierte Längenprüfung
- **Flexibel**: Für alle Textobjekttypen

Essential für:

- Dynamische Prozessvisualisierung
- Mehrsprachige Anwendungen
- Echtzeit-Datendarstellung

Q_StringValue_AIS





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **Q_StringValue_AIS** realisiert den Dienst „Command Change String Value“ gemäß ISO 11783-6 (Teil 6, Abschnitt F.24). Er dient dazu, über ein ISOBUS-Netzwerk einen neuen String-Wert an ein angeschlossenes Gerät zu senden. Der Baustein kapselt die gesamte Logik zur Initialisierung, zum Senden des Befehls und zur Rückmeldung des Ergebnisses. Die Eingabe des neuen String-Werts erfolgt über einen unidirektionalen Adapter (AIS), der die eigentlichen Daten bereitstellt.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar
INIT	EInit	Service-Initialisierung, setzt die Objekt-ID

Ereignis-Ausgänge

Ereignis	Typ	Kommentar
INITO	EInit	Bestätigung der erfolgreichen Initialisierung
CNF	Event	Bestätigung des angeforderten Dienstes (Ergebnis)

Daten-Eingänge

Name	Typ	Initialwert	Kommentar
QI	BOOL	TRUE	Quality Input – TRUE aktiviert den Dienst
u16ObjId	UINT	ID_NULL	Objekt-ID des zu ändernden Werts

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Quality Output – TRUE, wenn der Dienst erfolgreich ausgeführt wurde
STATUS	STRING	Dienst-Status (z. B. „IDLE“, „BUSY“, „ERROR“)
s16result	INT	Rückgabewert (siehe Beschreibung des Dienstes)

Adapter

Name	Typ	Kommentar
pau8String	adapter::types::unidirectional::AIS	Liefert den neuen String-Wert

Qualitätssignale QI/QO: Der Dateneingang `QI` (BOOL, Initialwert `TRUE`) ist dem Ereignis `INIT` zugeordnet und wird an `QI` des internen Bausteins `Q_StringValue` durchgereicht. Der Datenausgang `QO` (BOOL) wird von `Q_StringValue.QO` übernommen und ist den Ereignissen `INITO` und `CNF` zugeordnet.

Gültige Objekt-IDs

`u16ObjId` ist für dieselben Objekttypen gültig wie beim umschlossenen Basisbaustein `Q_StringValue` (Anhang F.24, Objekte mit String-Wert-Attribut): Input String Field (8000–8999), Output String Field (11000–11999), String Variable (22000–22999), Input Attributes (26000–26999). Auch das Verhalten der variablen F.24-Datenlänge / des ISO-Transport-Protokolls von `Q_StringValue` gilt hier.

ID_NULL (65535) ist kein gültiges Kommandoziel, deaktiviert aber bei `INIT` den Baustein.

Funktionsweise

Der Baustein enthält intern eine Instanz des Funktionsbausteins `Q_StringValue` (aus der Bibliothek `isobus::UT::Q`), die die eigentliche ISOBUS-Kommunikation durchführt. Die Verdrahtung erfolgt folgendermaßen:

1. **Initialisierung**: Ein INIT-Ereignis am Eingang aktiviert den internen `Q_StringValue`-Baustein. Die mitgegebene `u16ObjId` wird an diesen weitergeleitet.
2. **Auslösen des Dienstes**: Sobald der Adapter `pau8String` ein Ereignis (E1) empfängt, wird das REQ-Ereignis des internen Bausteins ausgelöst. Gleichzeitig wird der Stringwert über den Datenausgang D1 des Adapters an `Q_StringValue.pau8String` übergeben.
3. **Rückmeldung**: Nach Abschluss der Verarbeitung gibt der interne Baustein die Ereignisse `INITO` und `CNF` aus, deren Ausgangsdaten (`STATUS`, `s16result`) direkt an die entsprechenden Ausgänge des Gesamtbausteins durchgereicht werden.

Die gesamte Logik ist somit in dem enthaltenen `Q_StringValue`-Baustein implementiert; der `Q_StringValue_AIS` dient als spezialisierte Hülle mit einem Adaptereingang.

Technische Besonderheiten

- Der Baustein folgt der **ISO 11783-6** (ISOBUS) Spezifikation, Teil 6, Dienst „Command Change String Value“.
- Die Implementierung verwendet einen **unidirektionalen Adapter** (`adapter::types::unidirectional::AIS`), der nur Daten vom Host zum Baustein überträgt. Dies vereinfacht die Integration in Steuerungssysteme, bei denen der neue Stringwert asynchron bereitgestellt wird.
- Der initiale Wert von `u16ObjId` ist `ID_NULL` – dies muss vor der ersten Verwendung auf eine gültige Objekt-ID gesetzt werden.
- Die Ausgänge `STATUS` und `s16result` liefern den Status und den Rückgabewert des Dienstes, sobald das `CNF`-Ereignis ausgelöst wird. Die genauen Werte sind in der zugrunde liegenden Dienstbeschreibung definiert.

Zustandsübersicht

Der Baustein selbst besitzt keinen expliziten internen Automaten, da die Zustandslogik vollständig im enthaltenen `Q_StringValue` abläuft. Im Wesentlichen können folgende Phasen unterschieden werden:

- **Idle**: Nach dem Start oder nach einer erfolgreichen/fehlgeschlagenen Ausführung – der Baustein wartet auf ein INIT oder ein Ereignis am Adapter.
- **Initialisierung**: Aktiv durch INIT, bis das INITO-Ereignis gesendet wird.
- **Dienst aktiv**: Nach Auslösen durch den Adapter, bis das CNF-Ereignis eintrifft.
- **Fehler**: Bei ungültigen Parametern oder Kommunikationsfehlern wird ein entsprechender Status/Rückgabewert geliefert.

Anwendungsszenarien

- **ISOBUS-Kommando:** Ändern eines String-Parameters an einem landwirtschaftlichen Gerät (z. B. Maschinenname, Aufgabe, Variablenname) über das ISOBUS-Netzwerk.

- **Steuerungssysteme:** Anbindung an eine SPS oder ein HMI, das den neuen String-Wert über einen Adapter bereitstellt (beispielsweise aus einer Textbox, einem Datenbankfeld oder einem Kommunikationskanal).
- **Testumgebungen:** Simulieren des Dienstes zu Entwicklungs- und Testzwecken.

Vergleich mit ähnlichen Bausteinen

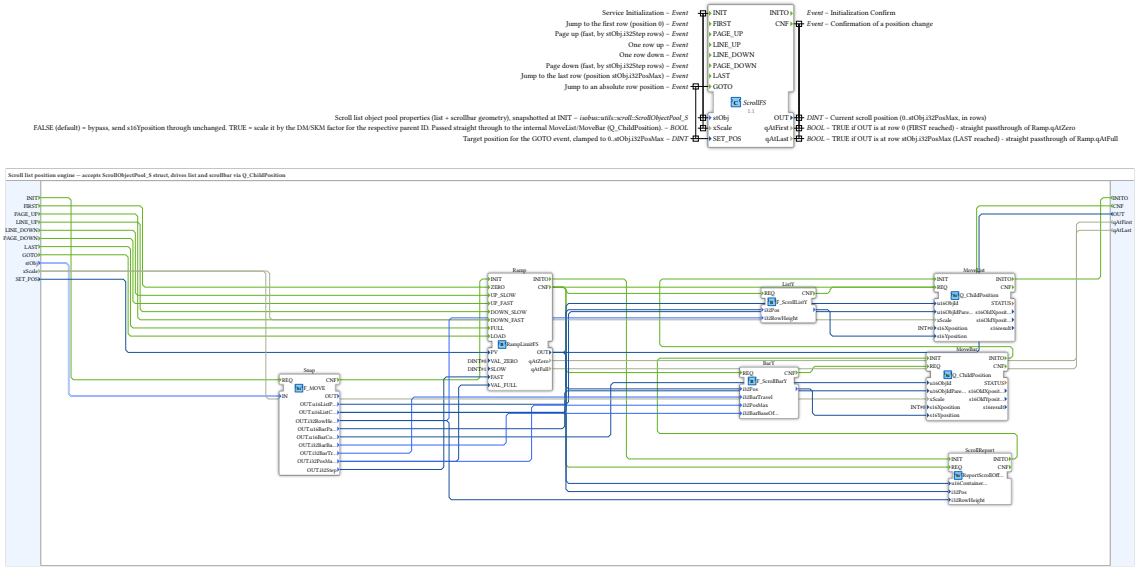
Der `Q_StringValue_AIS` ist eine spezialisierte Variante des Bausteins `Q_StringValue`, der bereits die gesamte ISOBUS-Kommunikation enthält. Der Unterschied liegt in der Eingabeschnittstelle: Während `Q_StringValue` den String-Wert über einen normalen Daten-Eingang `pau8String` (vom Typ `STRING`-Array) entgegennimmt, verwendet `Q_StringValue_AIS` einen **Adapter** (AIS), der asynchron Daten liefern kann. Der Adapter ermöglicht eine lose Kopplung und erleichtert die Wiederverwendung in unterschiedlichen Kontexten, z. B. wenn der Stringwert von einem anderen, unabhängigen Baustein bereitgestellt wird.

Weitere ähnliche Bausteine aus der ISO 11783-6-Familie behandeln andere Datentypen (z. B. `Q_NumberValue_AIS` für numerische Werte).

Fazit

Der `Q_StringValue_AIS` ist ein praktischer, standardkonformer Funktionsbaustein zur Änderung von String-Werten im ISOBUS-Kontext. Durch die Verwendung eines Adapters wird die Integration flexibel und modular. Er eignet sich besonders für Anwendungen, bei denen der neue Wert asynchron oder von externen Quellen bereitgestellt wird. Die klare Trennung von Initialisierung und Dienstausführung sowie die standardisierten Ausgänge erleichtern die Fehlersuche und Wiederverwendung.

ScrollFS



• – ○ • – ○ • – ○ •

Einleitung

`ScrollFS` ist die reine Positions-Engine einer scrollbaren VT-Liste: er hält die aktuelle Zeilen-Position (0…`i32PosMax`), reagiert auf Navigations-Ereignisse (Zeile/Seite hoch/runter, Anfang/Ende, absolute Position) und schreibt die passende Y-Verschiebung sowohl für den Listen-Inhalt als auch für den Scrollbalken-Thumb über zwei `Q_ChildPosition`-Aufrufe (ISO 11783-6 F.16, „Change Child Location“) an den VT. Er liest **keine** physischen Tasten oder Eingabefelder — dafür ist ScrollFS_PHYS_Softkey bzw. ScrollFS_PHYS_Button zuständig, die `ScrollFS` intern kapseln. Details zur Herleitung siehe `Workspace_Scroll/SCROLL_KONZEPT.md` im Projekt `4diac_training1`.

Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Übernimmt die Konfiguration (`stObj`) per Snapshot und `xScale`.
- `FIRST`: Springt zur ersten Zeile (Position 0).

- `PAGE_UP`: Eine Seite nach oben (schnell, um `stObj.i32Step` Zeilen).
- `LINE_UP`: Eine Zeile nach oben.
- `LINE_DOWN`: Eine Zeile nach unten.
- `PAGE_DOWN`: Eine Seite nach unten (schnell, um `stObj.i32Step` Zeilen).
- `LAST`: Springt zur letzten Zeile (Position `stObj.i32PosMax`).
- `GOTO`: Springt zu einer absoluten Zeilen-Position (`SET_POS`), geklemmt auf `0…stObj.i32PosMax`.

Ereignis-Ausgänge

- `CNF`: Bestätigt eine Positionsänderung, liefert `OUT`, `qAtFirst` und `qAtLast`.

Daten-Eingänge

- `stObj` (`isobus::utils::scroll::ScrollObjectPool_S`): Listen-/Balken-Geometrie (Container-IDs, Zeilenhöhe, Balken-Offsets, `i32PosMax`, `i32Step`) — wird bei `INIT` einmalig geschnappt.
- `xScale` (BOOL, Standard `FALSE`): `FALSE` = Bypass, `s16Yposition` wird unverändert gesendet; `TRUE` = Skalierung mit dem DM/SKM-Faktor der jeweiligen Parent-ID. Wird direkt an das interne MoveList/MoveBar (`Q_ChildPosition`) durchgereicht.
- `SET_POS` (DINT): Zielposition für `GOTO`.

Daten-Ausgänge

- `OUT` (DINT): Aktuelle Scroll-Position (0…`stObj.i32PosMax`, in Zeilen).
- `qAtFirst` (BOOL): `TRUE`, wenn `OUT` bei Zeile 0 steht (Listenanfang erreicht).
- `qAtLast` (BOOL): `TRUE`, wenn `OUT` bei `stObj.i32PosMax` steht (Listenende erreicht).

Adapter

Keine Adapter vorhanden.

Funktionsweise

`ScrollFS` verdrahtet fünf Bausteine zu einer fertigen Positions-Engine:

1. **`Snap`** (`F_MOVE`, `DataType := ScrollObjectPool_S`): schnappt `stObj` bei `INIT` einmalig, danach stehen alle neun Geometrie-Felder als `Snap.OUT.<Feldname>` dauerhaft zur Verfügung.
2. **`Ramp`** (`RampLimitFS`, Paket `eclipse4diac::signalprocessing`): die eigentliche Positions-Zustandsmaschine, direkt wiederverwendet (kein eigener Zustandsautomat nötig). Die acht `ScrollFS`-Events sind 1:1 auf `RampLimitFS`-Events gemappt:

`ScrollFS`-Event	`RampLimitFS`-Event	Wirkung auf `OUT`
`FIRST`	`ZERO`	`OUT := 0`
`PAGE_UP`	`DOWN_FAST`	`OUT -= stObj.i32Step` (geklemmt bei 0)
`LINE_UP`	`DOWN_SLOW`	`OUT -= 1` (geklemmt bei 0)
`LINE_DOWN`	`UP_SLOW`	`OUT += 1` (geklemmt bei `i32PosMax`)
`PAGE_DOWN`	`UP_FAST`	`OUT += stObj.i32Step` (geklemmt bei `i32PosMax`)
`LAST`	`FULL`	`OUT := stObj.i32PosMax`
`GOTO`	`LOAD` (mit `PV := SET_POS`)	`OUT := SET_POS`, geklemmt auf `0…i32PosMax`

`VAL_ZERO := 0`, `VAL_FULL := Snap.OUT.i32PosMax`, `SLOW := 1`, `FAST := Snap.OUT.i32Step`. 3. **`ListY`** (`F_ScrollListY`) und **`BarY`** (`F_ScrollBarY`): rechnen bei jedem `Ramp.CNF` aus `Ramp.OUT` die Y-Verschiebung für Liste bzw. Scrollbalken. 4. **`MoveList`** und **`MoveBar`** (je `Q_ChildPosition`): schreiben die berechneten Y-Werte an den VT — `MoveList` bewegt `Container_Scrolling_Content` innerhalb von `Container_Scrolling_Parent`, `MoveBar` bewegt `Container_Scrollbar_Content` innerhalb von `Container_Scrollbar_Parent`. Beide `s16Xposition := 0` fest (nur vertikales Scrollen). 5. **`ScrollReport`** (`ReportScrollOffset`): meldet bei jedem `Ramp.CNF` (sowie initial bei `Ramp.INIT0`) die aktuelle Scroll-Position (Offset in Pixeln = `Ramp.OUT * Snap.OUT.i32RowHeight`) für den Container `Snap.OUT.u16ListContentId` an das ECU-Sichtbarkeits-Gate (`VtMaskVisibility`), damit sichtbare Objekte für VT-Refreshs präzise gefiltert werden können.

`Ramp.qAtZero`/`Ramp.qAtFull` werden unverändert als `qAtFirst`/`qAtLast` durchgereicht.

Technische Besonderheiten

- **`F_MOVE`-Snapshot für Struct-Zugriff**: `Snap.IN`/`.OUT` sind generisch `ANY`-typisiert; eine `ANY`-Verbindung (hier: der konkret typisierte `stObj` auf den generischen `Snap.IN`) läuft in 4diac grundsätzlich nur über einen `MOVE`-Baustein, nie direkt. Jedes einzelne Struct-Feld, das per `Snap.OUT.<Feld>` angesprochen wird, braucht dafür einen eigenen `Parameter Name="OUT.<Feld>"`-Eintrag mit `Attribute Name="Visible" Value="true"` auf der `Snap`-Instanz — fehlt der, kompiliert das Netz nicht.
- **Init-Kette seriell, nicht parallel**: `Snap.CNF` löst `Ramp.INIT` aus; erst nach `Ramp.INITO` werden `MoveList.INIT` und `MoveBar.INIT` angestoßen — dadurch stehen `VAL_ZERO`/`VAL_FULL`/`SLOW`/`FAST` in `Ramp` garantiert, bevor irgendein Positionswert an den VT geschrieben wird.
- **`ListY`/`BarY` sind konkret typisiert** (`DINT`/`INT`, siehe F_ScrollListY, F_ScrollBarY) — dadurch ist zwischen ihnen und `Ramp.OUT` kein zusätzlicher `MOVE`-Anker nötig, obwohl `Ramp.OUT` selbst aus einer generischen `RampLimitFS`-Instanz stammt.

Zustandsübersicht

Die einzige Zustandshaltung steckt in `Ramp` (`RampLimitFS`): `OUT` bleibt zwischen den Ereignissen erhalten, jedes Navigations-Event ändert ihn schrittweise oder springt direkt. `qAtFirst`/`qAtLast` werden bei jeder Änderung neu berechnet, nie separat gepflegt.

Anwendungsszenarien

- Wird nicht direkt verwendet, sondern immer über ScrollFS_PHYS_Softkey (Softkeys) oder ScrollFS_PHYS_Button (Bildschirm-Buttons), die die acht Navigations-Events aus physischen Tastendrücken ableiten.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Gegenüber `RampLimitFS`**: `ScrollFS` ist im Kern ein `RampLimitFS` mit VT-Anbindung — die Positions-Mathematik selbst (Klemmen, Grenzwert-Flags) liegt vollständig in `RampLimitFS`, `ScrollFS` ergänzt nur Struct-Konfiguration, Zeilen-/Balken-Umrechnung und das Schreiben an den VT.

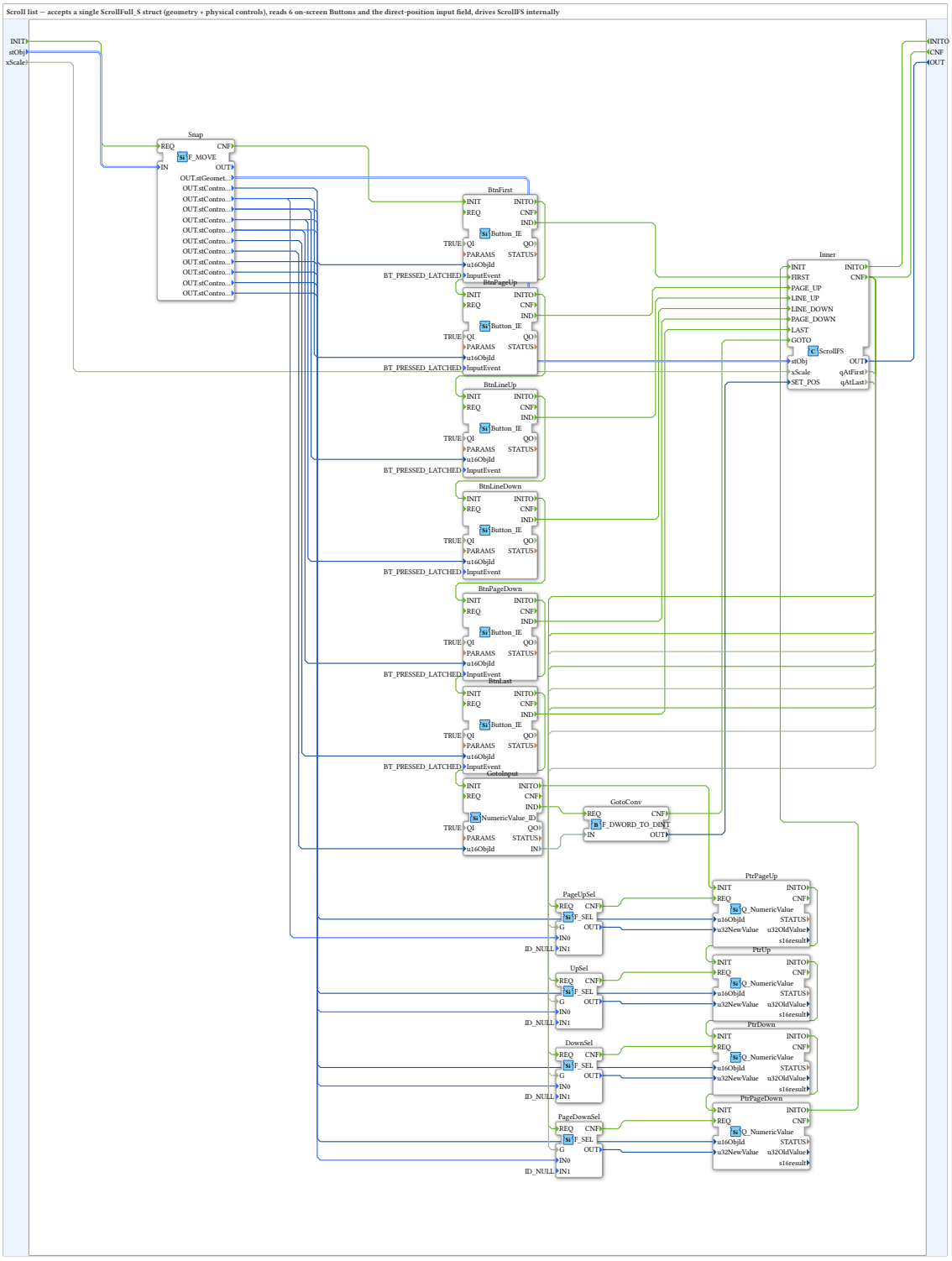
🛠️ Zugehörige Übungen

- Kein eigenständiges Übungsbeispiel — siehe `Workspace_Scroll/SCROLL_KONZEPT.md` (Projekt `4diac_training1`) für die vollständige Herleitung anhand eines realen Pools.

Fazit

`ScrollFS` bündelt Positions-Zustand, Grenzwert-Erkennung und VT-Schreibzugriff einer scrollbaren Liste in einem einzigen, wiederverwendbaren Composite-Baustein — unabhängig davon, ob die Bedienung später über Softkeys oder Bildschirm-Buttons erfolgt.

ScrollFS_PHYS_Button



• — ○ • — ○ • — ○ •

Einleitung

`ScrollFS_PHYS_Button` ist strukturell identisch zu ScrollFS_PHYS_Softkey — derselbe fertige, sofort einsetzbare Scroll-Baustein, dieselbe `ScrollFull_S`-Konstante, dieselbe Anschlag-Ausblendung —, liest die 6 Bedienelemente aber als Bildschirm-`Button`-Objekte (`Button_IE`, horcht auf `BT_PRESSED_LATCHED`) statt als physische `SoftKey`-Objekte. Gedacht für Masken, bei denen gescrollt werden soll, ohne dass eine SoftKey-Spalte zur Verfügung steht (z. B. Touch-Bedienung). Die 6 IDs in `ScrollControls_S` zeigen dann auf `Button`- statt auf `SoftKey`-Objekte, sonst ist `ScrollFull_S`/`ScrollControls_S` unverändert. Siehe `Workspace_Scroll/SCROLL_KONZEPT.md` (Projekt `4diac_training1`) für die vollständige Herleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Übernimmt die komplette Konfiguration (`stObj`) per Snapshot (mit `stObj`, `xScale`).

Ereignis-Ausgänge

- `INITO`: Bestätigung der Initialisierung.
- `CNF`: Bestätigt eine Positionsänderung, liefert `OUT`.

Daten-Eingänge

- `stObj` (`isobus::utils::scroll::ScrollFull_S`): komplette Konfiguration (Geometrie + Bedienelemente-IDs), bei `INIT` geschnappt.
- `xScale` (BOOL): FALSE (Standard) = Bypass, TRUE = Skalierung mit dem DM/SKM-Faktor. Wird unverändert an die interne `ScrollFS`-Instanz durchgereicht.

Daten-Ausgänge

- `OUT` (DINT): Aktuelle Scroll-Position (0…`stObj.stGeometry.i32PosMax`, in Zeilen).

Adapter

Keine Adapter vorhanden.

Funktionsweise

Identisch zu ScrollFS_PHYS_Softkey, mit einem Unterschied: die 6 Bedienelemente sind `Button_IE`-Instanzen (`BtnFirst`, `BtnPageUp`, `BtnLineUp`, `BtnLineDown`, `BtnPageDown`, `BtnLast`), die auf `BT_PRESSED_LATCHED` statt `SK_PRESSED` horchen. Alles Weitere — Direkteingabe über `NumericValue_ID+F_DWORD_TO_DINT`, interne ScrollFS-Positions-Engine, Anschlag-Ausblendung über `F_SEL+Q_NumericValue` auf die 4 ObjectPointer-IDs — ist 1:1 identisch zu `ScrollFS_PHYS_Softkey`.

Technische Besonderheiten

Siehe ScrollFS_PHYS_Softkey, Abschnitt „Technische Besonderheiten“ — alle dort beschriebenen Punkte (F_SEL-Richtung, ObjectPointer- vs. Key-ID, serielle Init-Kette) gelten unverändert. Der einzige Unterschied ist der Bedienelement-Typ (`Button_IE` statt `Softkey_IE`).

Zustandsübersicht

Wie ScrollFS_PHYS_Softkey — kein eigener Zustand außer dem in `Inner` (`ScrollFS`/`RampLimitFS`) gehaltenen.

Anwendungsszenarien

- Scrollbare Listen auf Masken ohne SoftKey-Spalte, insbesondere bei Touch-fähigen Terminals, wo Bildschirm-Buttons statt physischer Softkeys bevorzugt werden.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Gegenüber `ScrollFS_PHYS_Softkey`**: identisch bis auf den Bedienelement-Typ (`Button_IE`/`BT_PRESSED_LATCHED` statt `Softkey_IE`/`SK_PRESSED`). Siehe ScrollFS_PHYS_Softkey für die vollständige Beschreibung.
- **Gegenüber `ScrollFS`**: `ScrollFS_PHYS_Button` ist der praxistaugliche Wrapper — `ScrollFS` selbst kennt keine physischen Bedienelemente, nur abstrakte Navigationsereignisse.

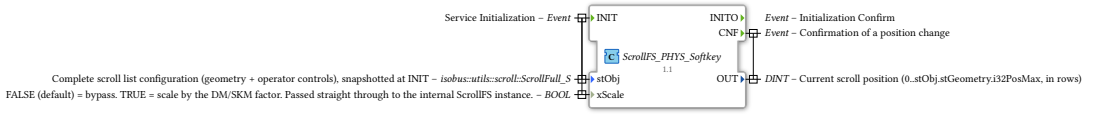
🛠️ Zugehörige Übungen

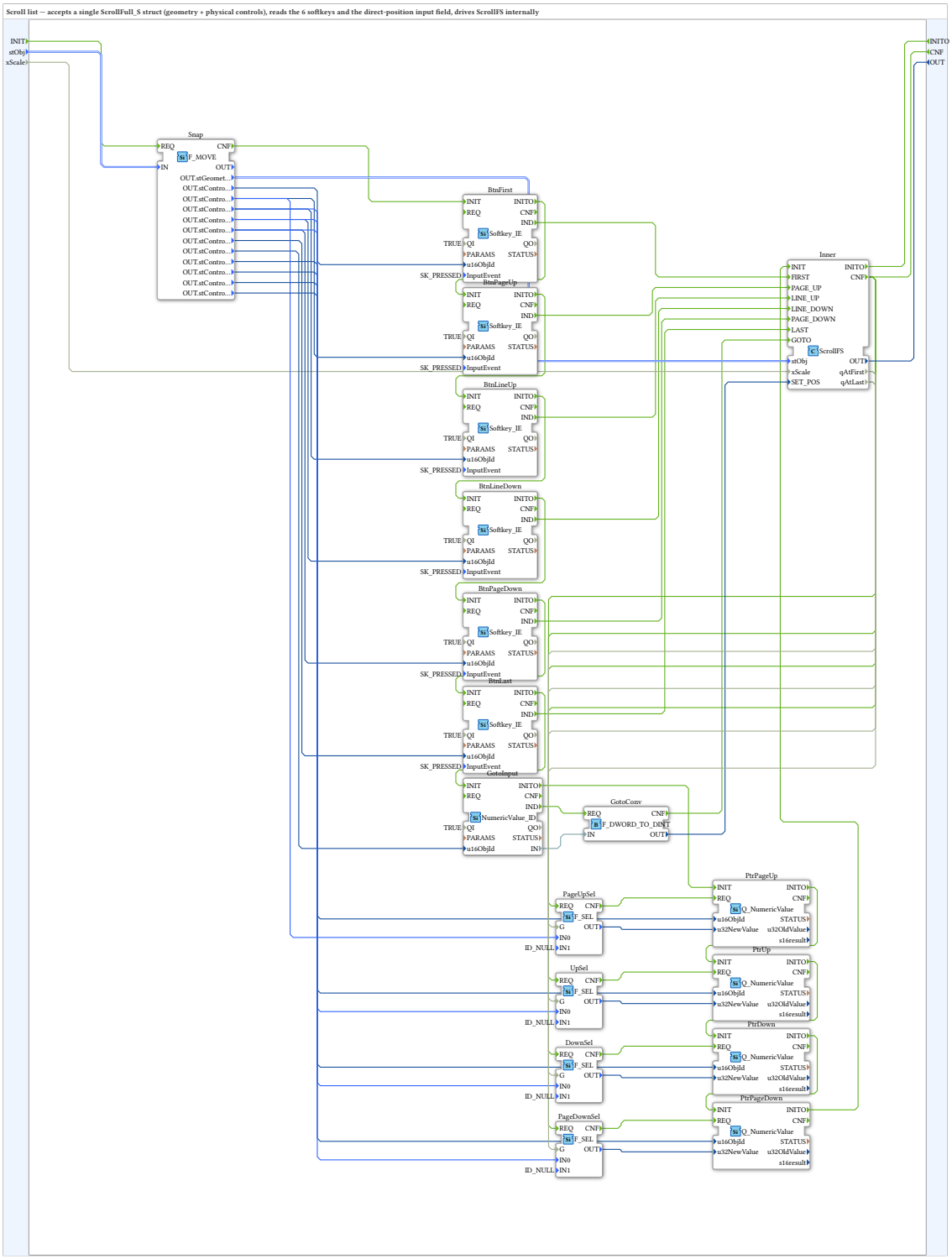
- Kein eigenständiges Übungsbeispiel — siehe `Workspace_Scroll/SCROLL_KONZEPT.md` (Projekt `4diac_training1`).

Fazit

`ScrollFS_PHYS_Button` bietet dieselbe fertige Scroll-Funktionalität wie `ScrollFS_PHYS_Softkey`, nur für Bildschirm-Buttons statt Softkeys — welche Variante passt, hängt allein davon ab, mit welchem Objekttyp die Bedienelemente im Pool angelegt sind.

ScrollFS_PHYS_Softkey





• — ○ • — ○ • — ○ •

Einleitung

`ScrollFS_PHYS_Softkey` ist der fertige, sofort einsetzbare Baustein für eine scrollbare VT-Liste mit Softkey-Bedienung: eine `ScrollFull_S`-Konstante hinein (wie sie `GcfScript.py` aus dem ISO-Designer-Pool erzeugt), fertig — er verdrahtet die 6 physischen Softkeys, das Direkteingabe-Feld und die interne Positions-Engine ScrollFS selbst, inklusive automatischer Ausblendung der Scroll-Tasten am jeweiligen Listenanschlag. Zwei Listen auf einer Maske = zwei Instanzen, jede mit eigener `ScrollFull_S`-Konstante. Siehe `Workspace_Scroll/SCROLL_KONZEPT.md` (Projekt `4diac_training1`) für die vollständige Herleitung.

Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Übernimmt die komplette Konfiguration (`stObj`) per Snapshot, mit `xScale`.

Ereignis-Ausgänge

- `CNF`: Bestätigt eine Positionsänderung, liefert `OUT`.

Daten-Eingänge

- `stObj` (`isobus::utils::scroll::ScrollFull_S`): komplette Konfiguration (Geometrie + Bedienelemente-IDs), bei `INIT` geschnappt.
- `xScale` (BOOL, Initialwert `FALSE`): `FALSE` = Bypass, `TRUE` = Skalierung mit dem DM/SKM-Faktor. Wird direkt an die interne `ScrollFS`-Instanz durchgereicht.

Daten-Ausgänge

- `OUT` (DINT): Aktuelle Scroll-Position (0…`stObj.stGeometry.i32PosMax`, in Zeilen).

Adapter

Keine Adapter vorhanden.

Funktionsweise

1. **`Snap`** (`F_MOVE`, `DataType = ScrollFull_S`): schnappt `stObj` bei `INIT`, danach stehen Geometrie und alle Bedienelemente-IDs als `Snap.OUT.stGeometry`/`Snap.OUT.stControls.<Feld>` zur Verfügung.
2. **6× `Softkey_IE`** (`BtnFirst`, `BtnPageUp`, `BtnLineUp`, `BtnLineDown`, `BtnPageDown`, `BtnLast`), je auf `SK_PRESSED` horchend, `u16ObjId` aus dem jeweiligen `Snap.OUT.stControls.u16Btn*Id` — ihr `IND`-Ereignis löst direkt das passende ScrollFS-Navigationsereignis auf `Inner` aus.
3. **`GotoInput`** (`NumericValue_ID`) + **`GotoConv`** (`F_DWORD_TO_DINT`): liest das Direkteingabe-Feld (`u16GotoInputId`) und speist den Wert als `SET_POS` in `Inner.GOTO`.
4. **`Inner`** (`ScrollFS`): die eigentliche Positions-Engine, siehe ScrollFS. `Inner.OUT` wird direkt als eigenes `OUT` durchgereicht.
5. **Anschlag-Ausblendung** — pro Richtung (Up-Seite: `BtnPageUp+BtnLineUp`, Down-Seite: `BtnPageDown+BtnLineDown`) je eine Kette aus `F_SEL` + `Q_NumericValue`, ausgelöst bei jeder Positionsänderung (`Inner.CNF`):
   - `F_SEL.G := Inner.qAtFirst` (Up-Seite) bzw. `Inner.qAtLast` (Down-Seite)
   - `F_SEL.IN0 :=` echte SoftKey-ID (`Snap.OUT.stControls.u16Btn*Id`) — Wert, wenn **kein** Anschlag
   - `F_SEL.IN1 := ID_NULL` — Wert, wenn Anschlag erreicht (`F_SEL.OUT := G ? IN1 : IN0`)
   - `Q_NumericValue.u16ObjId :=` die zugehörige ObjectPointer-ID (`Snap.OUT.stControls.u16Btn*PtrId`), `.u32NewValue := F_SEL.OUT`
   - Ergebnis: der ObjectPointer der SoftKeyMask zeigt bei Anschlag auf `ID_NULL` (Taste verschwindet), sonst auf den echten `SoftKey` (Taste sichtbar).

Technische Besonderheiten

- **`F_SEL`-Richtung ist nicht symmetrisch**: `F_SEL.OUT := G ? IN1 : IN0` — bei Verwechslung von `IN0`/`IN1` zeigt die Taste genau dann, wenn sie eigentlich verschwinden sollte (und umgekehrt). Diese Reihenfolge (`IN0` = sichtbarer Wert, `IN1` = `ID_NULL`) war in einer früheren Version vertauscht und musste korrigiert werden.
- **`Q_NumericValue` auf ObjectPointer statt „Hide/Show Object“**: Object Pointer werden nicht über das ISO-11783-6-Kommando „Hide/Show Object“ (Annex F.2, nur für Container gültig) aus-/eingeblendet, sondern über „Change Numeric Value“ (Annex F.22) direkt auf die ObjectPointer-ID selbst — das

ändert, worauf der Pointer zeigt (`ID_NULL` = zeigt auf nichts = unsichtbar). Vorbild: `Uebung_015` (Projekt `4diac_training1`).
- **ObjectPointer- vs. Key-ID nicht verwechseln**: `Snap.OUT.stControls.u16Btn*Id` ist die **echte** `SoftKey`-Objekt-ID (Ziel für `Softkey_IE.u16ObjId` **und** für `F_SEL.IN0`), `Snap.OUT.stControls.u16Btn*PtrId` ist die ID des **ObjectPointers**, der auf diese SoftKey im SoftKey-Mask-Baum zeigt (Ziel für `Q_NumericValue.u16ObjId`) — beide werden von `GcfScript.py` automatisch aus dem Pool aufgelöst (SoftKeyMask-Kind ist der ObjectPointer, dessen Ziel über eine `CProxy`-Indirektion die echte SoftKey ist).
- **Serielle Init-Kette**: `Snap.CNF → BtnFirst.INIT → … → BtnLast.INIT → GotoInput.INIT → PtrPageUp.INIT → PtrUp.INIT → PtrDown.INIT → PtrPageDown.INIT → Inner.INIT` — jeder Baustein wird erst initialisiert, wenn der vorherige fertig ist (`INITO`-Kette), analog zum entsprechenden Muster in ScrollFS.

Zustandsübersicht

Kein eigener Zustand außer dem in `Inner` (`ScrollFS`/`RampLimitFS`) gehaltenen. Die Anschlag-Ausblendung ist rein ereignisgetrieben — bei jeder Positionsänderung wird der Sichtbarkeitszustand aller vier Tasten neu berechnet, nicht nur beim tatsächlichen Anschlag-Übergang.

Anwendungsszenarien

- Jede scrollbare Liste am VT, die über physische Softkeys bedient wird (z. B. lange Auswahllisten, Diagnose-/Statusübersichten) — siehe `Workspace_Scroll` im Projekt `4diac_training1` für ein reales, getestetes Beispiel.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Gegenüber `ScrollFS`**: `ScrollFS_PHYS_Softkey` ist der praxistaugliche Wrapper — `ScrollFS` selbst kennt keine physischen Bedienelemente, nur abstrakte Navigationsereignisse.
- **Gegenüber `ScrollFS_PHYS_Button`**: identische Struktur und Anschlag-Ausblendung, liest die 6 Bedienelemente aber als Bildschirm-`Button`-Objekte (`BT_PRESSED_LATCHED`) statt als `SoftKey`-Objekte — für Masken ohne SoftKey-Spalte bzw. mit Touch-Bedienung. Siehe ScrollFS_PHYS_Button.

🛠️ Zugehörige Übungen

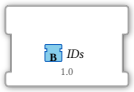
- Kein eigenständiges Übungsbeispiel — siehe `Workspace_Scroll/SCROLL_KONZEPT.md` (Projekt `4diac_training1`).

Fazit

`ScrollFS_PHYS_Softkey` ist der komplette, produktionsreife Scroll-Baustein für Softkey-bediente Listen — inklusive automatischer Anschlag-Ausblendung der Navigationstasten, ohne dass die Steuerungsanwendung selbst Grenzwerte prüfen oder Object Pointer manuell umbiegen muss.

const

IDs



• – ◦ • – ◦ • – ◦ •

Einleitung

Das globale Konstantenpaket `IDs` stellt eine Sammlung von vordefinierten, unveränderlichen Werten (Konstanten) bereit, die innerhalb von Funktionsbausteinen der IEC 61499-Laufzeitumgebung verwendet werden können. Diese Konstanten dienen primär der Verbesserung der Lesbarkeit und Wartbarkeit des Quellcodes, indem sie magische Zahlen durch aussagekräftige Bezeichner ersetzen. Das Paket ist im Namensraum `isobus::UT::Q::const` definiert.

Schnittstellenstruktur

Da es sich um ein Paket mit globalen Konstanten handelt, besitzt es keine ereignis- oder datenflussbasierten Schnittstellen im herkömmlichen Sinne eines Funktionsbausteins. Die Konstanten sind statisch und können direkt über ihren Namen referenziert werden.

Ereignis-Eingänge

Nicht vorhanden.

Ereignis-Ausgänge

Nicht vorhanden.

Daten-Eingänge

Nicht vorhanden.

Daten-Ausgänge

Nicht vorhanden.

Adapter

Nicht vorhanden.

Funktionsweise

Die Konstanten werden beim Kompilieren der Anwendung durch ihre fest definierten Werte ersetzt. Sie verhalten sich wie Literale, besitzen jedoch einen semantischen Namen. Die im Paket enthaltenen Konstanten sind:

- `ID_NULL`: Repräsentiert einen ungültigen oder nicht initialisierten Bezeichner (ID) vom Typ `UINT` mit dem Hexadezimalwert `16#FFFF` (dezimal 65535).
- `ID_OFF`: Repräsentiert einen speziellen Bezeichnerwert, der „Aus“ oder „Inaktiv“ signalisiert, vom Typ `UINT` mit dem Wert `16#0` (dezimal 0).
- `GLOBAL`: Ein Bezeichner vom Typ `USINT` mit dem Wert `16#FF` (dezimal 255), der typischerweise für globale oder broadcast-Adressierung verwendet wird.
- `OFF`: Ein einfacher Statuswert vom Typ `USINT` für den Zustand „Aus“ mit dem Wert `16#0` (dezimal 0).

Technische Besonderheiten

- **Typisierung:** Die Konstanten sind streng typisiert (`UINT`, `USINT`). Dies gewährleistet Typsicherheit bei ihrer Verwendung in Zuweisungen und Vergleichen.
- **Initialwert:** Jede Konstante wird bei ihrer Deklaration mit einem festen, unveränderlichen Initialwert versehen.
- **Namensraum:** Die Konstanten sind im Paket `isobus::UT::Q::const` gekapselt, was Namenskonflikte mit anderen Bibliotheken verhindert.

Zustandsübersicht

Da es sich um statische, konstante Werte handelt, existiert kein Zustandsautomat.

Anwendungsszenarien

- **Initialisierung:** Zuweisung eines definierten Startwerts (z.B. `ID_NULL`) an Variablen, die später eine gültige ID erhalten.
- **Vergleich:** Prüfung, ob eine empfangene oder gelesene ID einem speziellen Wert entspricht (z.B., `IF receivedID = IDs.ID_OFF THEN ...`).
- **Parameterisierung:** Verwendung der Konstanten als aussagekräftige Parameter in Funktionsbausteinen, anstelle von direkten numerischen Werten.
- **Kommando- und Statuswerte:** Verwendung von `OFF` und `GLOBAL` zur Steuerung von Prozessen oder zur Adressierung von Geräten in einem Netzwerk.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu Funktionsbausteinen (FBs) mit Algorithmen und Ereignissteuerung stellt dieses Paket lediglich Datenbereitstellung dar. Es ist vergleichbar mit:

- **ENUM-Datentypen:** Während ENUMs eine Liste von benannten Ganzzahlwerten eines Typs definieren, definiert dieses Paket einzelne, lose Konstanten verschiedener Typen.

- **#define-Präprozessor-Direktiven (in C):** Beide ersetzen einen Namen durch einen Wert. Die Konstanten in `IDs` sind jedoch typsicher und in der 4diac-IDE verwaltbar.

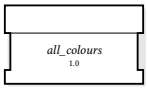
Fazit

Das globale Konstantenpaket `IDs` ist ein einfaches, aber nützliches Hilfswerkzeug für die strukturierte Programmierung in 4diac. Es erhöht die Code-Qualität, reduziert Fehler durch Tippfehler in numerischen Werten und macht die Absicht des Programmierers explizit. Seine Verwendung wird insbesondere in größeren Projekten oder bei der Arbeit mit Protokollen wie ISOBUS empfohlen, wo spezielle Bezeichnerwerte häufig vorkommen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

all_colours



• – ◦ • – ◦ • – ◦ •

Einleitung

Der GlobalConstants-Baustein **all_colours** stellt eine umfangreiche Farbpalette mit 256 definierten Farben (0 bis 255) bereit. Jede Farbe ist als Konstante vom Typ `USINT` angelegt und besitzt einen festen Zahlenwert. Die Kommentare enthalten den zugehörigen Hexadezimal-Code, den RGB-Wert sowie englische und deutsche Farbbezeichnungen. Die Farben 0 bis 231 entsprechen dem standardisierten ISOBUS-Farbraum, während die Farben 232 bis 255 für herstellerspezifische Zwecke reserviert und mit einem neutralen Grau belegt sind.

Schnittstellenstruktur

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Die Farben werden nicht über dedizierte Ausgangsvariablen bereitgestellt, sondern als globale Konstanten (`COLOR_0` … `COLOR_255`) deklariert. Diese können von anderen Bausteinen direkt referenziert werden.

Adapter

Keine.

Funktionsweise

Der Baustein deklariert die Konstanten beim Systemstart. Die Werte sind vorinitialisiert und während der gesamten Laufzeit unveränderlich. Alle Konstanten sind sofort gültig und können ohne zusätzliche Initialisierung in anderen Bausteinen verwendet werden. Die Nummerierung folgt dem ISOBUS-Standard, wobei die ersten 16 Farben (0–15) die Basispalette abdecken und die darauffolgenden Farben (16–231) feiner abgestufte Töne enthalten.

Technische Besonderheiten

- **Typ:** `USINT` (0–255)
- **Standardkonformität:** Farben 0–231 sind gemäß ISOBOBus definiert, die restlichen 24 Einträge (232–255) sind herstellerspezifisch und auf denselben Grauwert (`#808080`) gesetzt.

- **Namensgebung:** Die Bezeichnungen `COLOR_0` … `COLOR_255` ermöglichen eine einfache, numerische Referenzierung in Algorithmen oder Visualisierungen.
- **Keine Laufzeitabhängigkeit:** Der Baustein besitzt keine internen Zustände oder Algorithmen – er dient ausschließlich der Bereitstellung von Konstanten.

Zustandsübersicht

Dieser Baustein besitzt keine Zustandsautomaten, da es sich um eine rein deklarative Konstantensammlung handelt.

Anwendungsszenarien

- **ISOBUS-konforme UI-Entwicklung** in der Landtechnik (Terminals, Maschinensteuerungen)
- **Farbcodierung von Statusinformationen** (z. B. grün = aktiv, rot = Alarm, gelb = Warnung)
- **Vereinheitlichung von Farben** über mehrere Komponenten oder Projekte hinweg – alle Teilnehmer nutzen dieselbe Farbdefinition.

Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu einem typischen `FUNCTION_BLOCK`, der Ein-/Ausgangsdaten verarbeitet und über Ereignisse gesteuert wird, handelt es sich bei `all_colours` um einen **GlobalConstants**-Baustein. Er besitzt keine dynamischen Schnittstellen und keine ausführbare Logik. Ähnliche Bausteine sind z. B. `E_CTD` (Zähler) oder `F_TRIG` (Flankenerkennung), die jedoch aktives Verhalten aufweisen. Andere Konstanten-Sammlungen (z. B. physikalische Einheiten oder Schwellwerte) folgen dem gleichen deklarativen Muster.

Fazit

Der Baustein `all_colours` bietet eine standardisierte, leicht wiederzuverwendende Farbpalette für ISOBUS-Anwendungen. Durch die klare Nummerierung und die vollständigen Kommentare wird eine konsistente Farbverwendung über verschiedene Module und Entwickler hinweg gewährleistet. Für eigene Anpassungen stehen die reservierten Farben (232–255) zur Verfügung.

colours

• – ◦ • – ◦ • – ◦ •

Einleitung

Das globale Konstantenpaket `colours` definiert eine Reihe von benannten Konstanten für Farbwerte. Diese Konstanten repräsentieren einen Satz von 16 vordefinierten Farben, die durch numerische Werte des Typs `USINT` (Unsigned Short Integer) kodiert sind. Die Verwendung solcher Konstanten verbessert die Lesbarkeit und Wartbarkeit von IEC 61499-Anwendungen, da anstelle von magischen Zahlen aussagekräftige Bezeichner wie `COLOR_RED` oder `COLOR_BLUE` verwendet werden können.

Schnittstellenstruktur

Bei einem globalen Konstantenpaket handelt es sich nicht um einen Funktionsblock im klassischen Sinne, daher existieren keine Ereignis- oder Datenschnittstellen. Stattdessen stellt das Paket eine Sammlung von Konstantendeklarationen zur Verfügung, die in anderen Funktionsblöcken oder Programmen innerhalb desselben Projekts oder Namensraums importiert und genutzt werden können.

Ereignis-Eingänge

Nicht zutreffend. Globale Konstanten besitzen keine Ereigniseingänge.

Ereignis-Ausgänge

Nicht zutreffend. Globale Konstanten besitzen keine Ereignisausgänge.

Daten-Eingänge

Nicht zutreffend. Globale Konstanten besitzen keine Dateneingänge.

Daten-Ausgänge

Nicht zutreffend. Globale Konstanten besitzen keine Datenausgänge.

Adapter

Nicht zutreffend. Globale Konstanten besitzen keine Adapterschnittstellen.

Funktionsweise

Das Paket `colours` deklariert 16 Konstanten vom Datentyp `USINT`. Jeder Konstanten wird ein fester numerischer Wert zugewiesen, der eine spezifische Farbe repräsentiert. Die Werte reichen von `USINT#0` bis `USINT#15`. Sobald das Paket in eine Anwendung oder einen Funktionsblock importiert ist, können diese Konstantennamen überall dort verwendet werden, wo ein `USINT`-Wert erwartet wird. Der Compiler ersetzt den Konstantennamen bei der Übersetzung durch den zugewiesenen numerischen Wert.

Technische Besonderheiten

- **Datentyp:** Alle Konstanten sind vom Typ `USINT` (Unsigned Short Integer), einem vorzeichenlosen 8-Bit-Datentyp.
- **Wertebereich:** Die definierten Werte liegen im Bereich von 0 bis 15 und passen somit gut in den Wertebereich des `USINT`-Typs.
- **Namenskonvention:** Die Konstantennamen folgen der Konvention `COLOR_<FARBNAME>` (z.B., `COLOR_GREEN`), was sie leicht identifizierbar macht.
- **Kompilierung:** Als `CONSTANT` deklarierte Werte werden zur Kompilierzeit festgelegt und können zur Laufzeit nicht verändert werden.

Zustandsübersicht

Nicht zutreffend. Da es sich um statische, unveränderliche Werte handelt, besitzen globale Konstanten keinen Zustandsautomaten.

Anwendungsszenarien

Dieses Konstantenpaket ist besonders nützlich in Anwendungen, die eine begrenzte Palette von Farben für Benutzeroberflächen, visuelle Statusanzeigen (z.B. auf HMIs oder Bedienterminals) oder zur Kennzeichnung von Prozesszuständen benötigen. Typische Einsatzgebiete sind:

- **Visualisierung:** Zuweisung von Farben zu verschiedenen Maschinenzuständen (z.B. Grün = Betrieb, Rot = Fehler, Gelb = Warnung).
- **ISOBUS-konforme Anwendungen (UT/Universal Terminal):** Standardisierung von Farbwerten in landwirtschaftlichen Anwendungen, die das ISOBUS-Protokoll verwenden, um eine konsistente Darstellung auf verschiedenen Terminalgeräten zu gewährleisten.
- **Code-Verbesserung:** Ersetzen von undurchsichtigen Zahlenwerten in Algorithmen oder Konfigurationen durch selbsterklärende Bezeichner.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu Funktionsblöcken, die Logik ausführen oder Daten verarbeiten, stellt ein globales Konstantenpaket lediglich benannte Werte bereit. Alternativen wären:

1. **Direkte Verwendung numerischer Werte:** Führt zu schlechter lesbarem und schwerer wartbarem Code („Magic Numbers“).
2. **Definition von Konstanten innerhalb jedes Funktionsblocks:** Führt zu Redundanz und erschwert die zentrale Änderung von Werten.
3. **Verwendung von Aufzählungstypen (ENUM):** Bietet eine noch stärkere Typsicherheit, ist in der IEC 61499-Standardbibliothek jedoch nicht direkt als globales Konstrukt verfügbar. Die Verwendung von `USINT`-Konstanten ist eine praktische und weit verbreitete Alternative.

Fazit

Das globale Konstantenpaket `colours` ist ein einfaches aber effektives Werkzeug zur Strukturierung und Verbesserung von IEC 61499-Applikationen. Durch die Bereitstellung einer standardisierten Farbpalette als benannte Konstanten fördert es die Code-Qualität, erleichtert die Wartung und unterstützt die Entwicklung konsistenter und verständlicher Steuerungsanwendungen, insbesondere im Umfeld von ISOBUS und visuellen Schnittstellen.

####### AID

AID_AM

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `AID_AM` ist eine globale Konstantenliste, die die Attribut-IDs für ein **Alarm Mask**-Objekt im ISOBUS-Kontext definiert. Diese Konstanten werden verwendet, um eindeutige Kennungen für verschiedene Eigenschaften einer Alarmmaske (z. B. Hintergrundfarbe, Softkey-Maske, Alarmpriorität, akustisches Signal) zu referenzieren. Der Baustein ist als Teil des Pakets `isobus::UT::Q::const::AID` verfügbar und dient als wiederverwendbare Definitionssammlung für die Erstellung und Verwaltung von ISOBUS-basierten Alarmmasken in Steuerungssystemen.

Schnittstellenstruktur

Ereignis-Eingänge

Nicht vorhanden – es handelt sich um eine reine Konstantendefinition ohne prozessbezogene Eingaben.

Ereignis-Ausgänge

Nicht vorhanden – es werden keine Ereignisse oder Signale ausgegeben.

Daten-Eingänge

Nicht vorhanden – die Konstanten sind fest definiert und können zur Laufzeit nicht verändert werden.

Daten-Ausgänge

Nicht vorhanden – die Werte sind als globale Konstanten für andere Bausteine direkt verfügbar, ohne dass ein Datenausgang des Bausteins verwendet werden muss.

Adapter

Nicht vorhanden.

Funktionsweise

`AID_AM` stellt vier symbolische Konstanten bereit, die jeweils eine ganzzahlige Kennung (Typ `USINT`) für ein Attribut eines ISOBUS-Alarmmasken-Objekts repräsentieren. Die Zuordnung ist wie folgt:

Konstante	Wert	Beschreibung
`BACKGROUND_COLOUR`	1	Index für die Hintergrundfarbe der Alarmmaske.
`SOFT_KEY_MASK`	2	Objekt-ID einer Softkey-Maske, die mit der Alarmmaske verknüpft ist.
`ALARM_PRIORITY`	3	Prioritätsstufe des Alarms (0 = Hoch, 1 = Mittel, 2 = Niedrig).
`ACOUSTIC`	4	Akustisches Signalverhalten (0 = höchste Priorität, 1 = mittel, 2 = niedrig, 3 = still).

Diese Konstanten werden typischerweise in anderen Funktionsbausteinen verwendet, um auf die entsprechenden Attribute eines Alarmmasken-Objekts im ISOBUS-Netzwerk zuzugreifen (z. B. beim Lesen oder Setzen von Objekteigenschaften über den ISOBUS-Dienst).

Technische Besonderheiten

- **Datentyp:** Alle Konstanten sind als `USINT` (Unsigned Short Integer, 8 Bit) definiert.
- **Initialwerte:** Die Werte sind fest mit `USINT#1` bis `USINT#4` belegt und entsprechen den offiziellen ISOBUS-Attribut-IDs.
- **Paketierung:** Die Konstanten sind im Paket `isobus::UT::Q::const::AID` gekapselt, wodurch eine übersichtliche und wiederverwendbare Struktur innerhalb der 4diac-IDE ermöglicht wird.
- **Keine Laufzeitänderung:** Da es sich um `GLOBAL CONSTANT` handelt, sind die Werte während der gesamten Anwendungsdauer unveränderlich.

Zustandsübersicht

Nicht zutreffend – der Baustein besitzt keinen internen Zustandsautomaten und keine Zustandsübergänge.

Anwendungsszenarien

- **ISOBUS-Terminal-Programmierung:** Einsatz in Funktionen zur Konfiguration von Alarmmasken, z. B. beim Ändern der Hintergrundfarbe oder beim Zuordnen einer Softkey-Maske.
- **Alarmmanagement:** Verwendung zur Unterscheidung der Alarmprioritäten in der Steuerungslogik, um unterschiedliche Reaktionen oder Darstellungen auszulösen.
- **Akustische Signalisierung:** Steuerung des akustischen Verhaltens von Alarmen durch die entsprechende Attribut-ID.
- **Basis für weitere Konstanten:** Der Baustein kann als Vorlage für andere AID-Konstantenlisten (z. B. für Softkey-Masken oder Objekteigenschaften) dienen.

Vergleich mit ähnlichen Bausteinen

Es existieren in ISOBUS-Implementierungen weitere AID-Konstanten für andere Objekttypen, z. B. `AID_SKM` für Softkey-Masken oder `AID_BGM` für Background-Masken. `AID_AM` ist spezifisch auf Alarmmasken zugeschnitten und enthält nur die hierfür relevanten Attribut-IDs. Andere Bausteine könnten mehr oder andere Konstanten enthalten, je nach den geforderten Objekteigenschaften. Im Gegensatz zu Funktionsbausteinen mit Ein-/Ausgängen bietet `AID_AM` keine prozessbezogene Logik, sondern dient ausschließlich als Konstantenquelle.

Fazit

`AID_AM` ist ein einfacher, aber essenzieller Baustein für die Entwicklung von ISOBUS-konformen Anwendungen, die Alarmmasken verwenden. Durch die klare Definition der Attribut-IDs als globale Konstanten wird die Lesbarkeit und Wartbarkeit des Codes erhöht und die Gefahr von Tippfehlern minimiert. Die Konzentration auf eine spezifische Objektart macht den Baustein schlank und leicht einsetzbar. Für Entwickler, die ISOBUS-Terminals steuern, bietet `AID_AM` eine verlässliche Grundlage für die korrekte Adressierung von Alarmmasken-Eigenschaften.

AID_AO

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AID_AO** ist ein globaler Konstanten-Baustein der 4diac-IDE, der die Attribut-IDs für Animationsobjekte (Animation Objects) im ISOBUS-Kontext definiert. Er stellt neun benannte Konstanten vom Typ `USINT` bereit, die in der Kommunikation mit ISOBUS-Terminals zur Identifikation von Objektattributen verwendet werden. Dieser Baustein dient ausschließlich als Nachschlagewerk für Entwickler und wird nicht direkt in einer Applikation instanziiert, sondern global referenziert.

Schnittstellenstruktur

Da es sich um einen GlobalConstants-Baustein handelt, besitzt er keine klassischen Ein-/Ausgänge oder Ereignisse. Die **Schnittstellenstruktur** besteht ausschließlich aus den globalen Konstanten, die als Datenwerte für andere Bausteine verfügbar sind.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine – die Konstanten werden nicht als Ausgänge geführt, sondern global im Projekt bereitgestellt. Sie können über den Namen (z.B. `AID_AO.WIDTH`) direkt in Ausdrücken verwendet werden.

Adapter

Keine.

Konstante	Wert	Beschreibung

WIDTH	1	AID_AO_WIDTH – Breite in Pixeln
HEIGHT	2	AID_AO_HEIGHT – Höhe in Pixeln
REFRESHINT	3	AID_AO_REFRESHINT – Aktualisierungsintervall
VALUE	4	AID_AO_VALUE – aktueller Wert
ENABLED	5	AID_AO_ENABLED – 0 = gestoppt, 1 = animiert
FICHILDINDEX	6	AID_AO_FICHILDINDEX – Index des ersten Kindes
LACHILDINDEX	7	AID_AO_LACHILDINDEX – Index des letzten Kindes
DEFCHILDINDEX	8	AID_AO_DEFCHILDINDEX – Index des Standard-Kindes
OBJECTS	9	AID_AO_OBJECTS – Optionsbits (siehe unten)

Funktionsweise

Dieser Baustein definiert Konstanten, die die numerischen Kennungen der Attribute eines Animationsobjekts im ISOBUS-Objektmodell festlegen. Diese IDs werden typischerweise verwendet, um über den ISOBUS-Dienst „Get/Set Attribute“ gezielt Eigenschaften eines Animationsobjekts auszulesen oder zu verändern. Die Konstanten sind als `USINT` (Unsigned Short Integer) deklariert und fest mit den Werten 1 bis 9 belegt.

Die Bedeutung der `OBJECTS`-Konstante wird in den Kommentaren näher erläutert:

- **Bit 0**: Bestimmt die Animationssequenz (0 = Einzelschuss, 1 = Schleife)
- **Bits 1-2**: Legen das Verhalten bei deaktivierter Animation fest:
  - 0 = Pause
  - 1 = Zurücksetzen auf erstes Element
  - 2 = Standardobjekt anzeigen
  - 3 = Leer lassen

Technische Besonderheiten

- Der Baustein ist als **GlobalConstants**-Element gekapselt und wird im Compilerinfo als `isobus::UT::Q::const::AID` definiert.
- Alle Konstanten sind mit `VAR_GLOBAL CONSTANT` deklariert, d.h. sie sind schreibgeschützt und projektweit gültig.
- Die Werte sind als Literale mit Präfix `USINT#` versehen, um den Datentyp eindeutig festzulegen.
- Der Baustein enthält Versionsinformationen (Version 1.0, Org: HR Agrartechnik GmbH, Autor: Franz Höpfinger, Datum: 2026-06-20) und unterliegt der Eclipse Public License 2.0 (EPL-2.0).
- Es wird keine grafische Darstellung oder Zustandslogik benötigt; der Baustein dient rein der Konstantendefinition.

Zustandsübersicht

Nicht zutreffend – es existieren keine dynamischen Zustände, da der Baustein ausschließlich statische Konstanten bereitstellt.

Anwendungsszenarien

- **ISOBUS-Implementierung**: Verwendung der Konstanten beim Aufbau von Nachrichten zum Lesen oder Schreiben von Attributen eines Animationsobjekts (z.B. `OBJECTS`, `ENABLED`, `WIDTH`).
- **Objektsteuerung**: Steuerung der Animation eines ISOBUS-Objekts durch Setzen der Werte für `ENABLED` und `OBJECTS`.
- **Konfiguration**: Verwaltung von Größe (`WIDTH`, `HEIGHT`) und Aktualisierungsintervall (`REFRESHINT`) eines animierten Objekts.
- **Entwicklungshilfe**: Dient als zentrale Referenz für ID-Nummern, um Fehlzuordnungen in der Programmierung zu vermeiden.

Vergleich mit ähnlichen Bausteinen

Es existieren weitere GlobalConstants-Bausteine für andere Objekttypen, z.B. `AID_OBJ` für generische Objektattribute oder `AID_WD` für Working Set-Definitionen. Im Vergleich zu diesen definiert `AID_AO` spezifisch die Attribute für Animationsobjekte und deckt damit die besonderen Eigenschaften ab, die

bei animierten Darstellungen benötigt werden (z.B. Sequenzmodus, Verhalten bei Deaktivierung). Die Konstanten sind bewusst auf die ISOBUS-Spezifikation abgestimmt.

Fazit

Der Baustein `AID_A0` ist eine einfache, aber essenzielle Konstantensammlung für die Entwicklung ISOBUS-konformer Anwendungen mit animierten Objekten. Er füllt die Lücke zwischen abstrakten Objekt-IDs und den spezifischen Attributen eines Animationsobjekts und erleichtert die Einhaltung der ISOBUS-Standards. Durch die zentrale Definition wird eine konsistente Verwendung der ID-Werte im gesamten Projekt gewährleistet.

AID_AUXC2

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AID_AUXC2** ist ein GlobalConstants-Baustein der 4diac-IDE, der Konstanten für die Attribut-IDs eines „Auxiliary Control Designator Type 2“ (AUX-C2) Objekts im ISOBUS (ISO 11783) Protokoll bereitstellt. Er definiert die numerischen Identifikatoren für die Pointer-Typ- und Objekt-ID-Felder, die bei der Kommunikation mit landwirtschaftlichen Geräten verwendet werden. Die bereitgestellten Konstanten sind global und stehen im gesamten System zur Verfügung.

Schnittstellenstruktur

Da es sich um einen GlobalConstants-Baustein handelt, besitzt er keine herkömmlichen Ein-/Ausgangsports. Stattdessen werden die definierten Konstanten systemweit als globale Variablen bereitgestellt.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Keine.

Funktionsweise

Der Baustein definiert zwei globale Konstanten, die die Attribut-IDs für das AUX-C2-Objekt im ISOBUS-Datenmodell festlegen. Diese Konstanten werden zur Laufzeit als unveränderliche Werte genutzt und können von anderen Bausteinen referenziert werden, um die entsprechenden Attribute eindeutig zu identifizieren.

Die definierten Konstanten sind:

- **PTR_TYPE** (USINT, Initialwert = 1): Gibt an, worauf der Pointer verweist.
  - 0: Points to Aux Object
  - 1: Points to Aux Function/Input
  - 2: Points to Working Set owner of pointer
  - 3: Points to Working Set owner of assigned function/input
- **OBJ_ID** (USINT, Initialwert = 2): Object ID eines referenzierten Auxiliary Function- oder Auxiliary Input-Objekts oder NULL.

Diese Werte sind gemäß der ISOBUS-Spezifikation festgelegt und werden als Konstante deklariert, um versehentliche Änderungen zu vermeiden.

Technische Besonderheiten

- **Globaler Geltungsbereich**: Die Konstanten sind im gesamten Projekt verfügbar und können von allen Bausteinen verwendet werden.
- **Unveränderlichkeit**: Da sie als `CONSTANT` deklariert sind, können sie zur Laufzeit nicht verändert werden.
- **Kompatibilität**: Der Baustein ist für die Verwendung im ISOBUS-Kontext konzipiert und erfüllt die Anforderungen der Norm ISO 11783.
- **Wartung**: Durch die zentrale Definition wird eine einheitliche Referenz auf die Attribut-IDs gewährleistet.

Zustandsübersicht

Nicht zutreffend – der Baustein besitzt keinen internen Zustandsautomaten. Er ist ein reiner Konstantencontainer ohne dynamisches Verhalten.

Anwendungsszenarien

- **ISOBUS-Implementierungen**: Einsatz in Steuergeräten (ECUs) für landwirtschaftliche Maschinen, die das AUX-C2-Protokoll unterstützen.
- **Konfiguration von Bediengeräten**: Definition der Pointer-Typ- und Objekt-ID-Attribute für die Zuordnung von Steuerbefehlen.
- **Entwicklung von Diagnosetools**: Verwendung als Referenz, um die Bedeutung von Attribut-IDs zu dokumentieren.

Vergleich mit ähnlichen Bausteinen

Ähnliche GlobalConstants-Bausteine existieren für andere Attribute, z. B. **AID_AUXC1** (für Typ 1) oder **AID_AUXC3**. Diese unterscheiden sich in den Konstantenwerten und deren Semantik. AID_AUXC2 ist speziell auf den Typ 2 des Auxiliary Control Designators ausgelegt und definiert die dafür relevanten Attribut-IDs.

Fazit

Der Baustein **AID_AUXC2** ist eine einfache, aber wichtige Komponente für die ISOBUS-Kommunikation. Er stellt die benötigten Konstanten für die AUX-C2-Attribute bereit und sorgt durch die globale Definition für eine konsistente Verwendung im gesamten System. Obwohl er keine komplexe Funktionalität besitzt, ist er für die korrekte Implementierung von ISOBUS-Anwendungen unerlässlich.

AID_AUXF2

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AID_AUXF2** ist ein GlobalConstants-Objekt der 4diac-IDE. Er definiert die Attribut-IDs für das **Auxiliary Function Type 2 (AUX-F2)**-Objekt im ISOBUS-Kontext. Diese Konstanten werden verwendet, um auf die Attribute **Hintergrundfarbe** und **Funktionstyp** eines solchen Objekts zuzugreifen. Der Baustein stellt die Werte als globale, unveränderliche Konstanten bereit und dient als zentraler Referenzpunkt für die entsprechenden Attribut-IDs.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Dieser Baustein besitzt keine Ereignis-Eingänge, da es sich um ein reines Konstantenobjekt handelt.

Ereignis-Ausgänge

Keine. Es werden keine Ereignisse ausgelöst oder weitergegeben.

Daten-Eingänge

Keine. Die Konstanten sind fest definiert und benötigen keine Eingabedaten.

Daten-Ausgänge

Dieser Baustein stellt folgende globale Konstanten bereit, die als Ausgabewerte interpretiert werden können:

Name	Datentyp	Wert	Kommentar
BACKGROUND_COLOUR	USINT	1	AID_AUXF2_BACKGROUND_COLOUR – Index für die Hintergrundfarbe.
FUNC	USINT	2	AID_AUXF2_FUNC – Bitmaske: Bits 0-4 = Funktionstyp (0-14), Bit 5 = Critical Control, Bit 6 = Assignment Restriction, Bit 7 = Single-assignment.

Adapter

Keine. Es sind keine Adapter definiert.

Funktionsweise

Der Baustein *AID_AUXF2* ist ein reines Konstantenpaket. Er definiert über das Schlüsselwort `VAR_GLOBAL CONSTANT` zwei unveränderliche Werte vom Typ `USINT`. Diese Werte sind im gesamten Projekt verfügbar und können von anderen Funktionsbausteinen gelesen werden, um auf die entsprechenden Attribut-IDs des AUX-F2-Objekts zuzugreifen. Der Zugriff erfolgt über den Bausteinnamen, z.B. `AID_AUXF2.BACKGROUND_COLOUR`. Die Konstanten dienen der Lesbarkeit und Wartbarkeit des Codes, da sie symbolische Namen für magische Zahlen bereitstellen.

Technische Besonderheiten

- Der Baustein ist als `GlobalConstants`-Objekt der Eclipse 4diac-IDE implementiert.
- Er trägt den `CompilerInfo`-Paketnamen `isobus::UT::Q::const::AID`, was auf eine organisierte Zuordnung innerhalb eines ISOBUS-Projekts hinweist.
- Die Konstanten sind als `USINT` (Unsigned Short Integer) definiert und haben feste Initialisierungswerte.
- Es gibt keinerlei Laufzeitlogik, Zustände oder dynamische Berechnungen.

Zustandsübersicht

Da es sich um ein statisches Konstantenobjekt handelt, existiert kein Zustandsautomat. Der Zustand ist immer konstant und unveränderlich.

Anwendungsszenarien

- **ISOBUS-Kommunikation:** Bei der Implementierung von ISOBUS-Nachrichten für AUX-Funktionen können diese Konstanten verwendet werden, um die Attribut-IDs für Hintergrundfarbe und Funktionstyp zu referenzieren.
- **Konfiguration von Bedienterminals:** In landwirtschaftlichen Maschinen können AUX-F2-Objekte auf Terminals konfiguriert werden; die Konstanten erleichtern die Programmierung dieser Konfiguration.
- **Vereinheitlichung von Werten:** Durch die zentrale Definition werden Tippfehler vermieden und spätere Änderungen der IDs zentral angepasst.

Vergleich mit ähnlichen Bausteinen

Es existieren weitere ähnliche GlobalConstants-Objekte, z.B. `AID_AUXF1` oder `AID_AUXF3`, die sich auf andere AUX-Funktionszeiger beziehen. Diese Bausteine folgen demselben Muster: Sie definieren eine Liste von Attribut-IDs für das jeweilige AUX-Objekt. Der Unterschied liegt lediglich in den spezifischen Konstanten und deren Werten. `AID_AUXF2` ist speziell für den Typ 2 vorgesehen.

Fazit

Der Baustein **AID_AUXF2** erfüllt eine wichtige Aufgabe in der ISOBUS-Entwicklung, indem er die Attribut-IDs für AUX-F2-Objekte standardisiert und als globale Konstanten bereitstellt. Durch seine einfache, rein deklarative Struktur ist er wartungsfreundlich und trägt zur Codequalität bei. Er ist ein typisches Beispiel für die Nutzung von GlobalConstants in der 4diac-IDE, um feste Werte sicher und zentral zu verwalten.

AID_AUXI2

• – ◦ • – ◦ • – ◦ •

Einleitung

AID_AUXI2 ist ein GlobalConstants-Datentyp im Rahmen der 4diac-IDE, der konstante Attribut-IDs für das Auxiliary Input Type 2 (AUXI2) Objekt im ISOBUS-Protokoll (ISO 11783) definiert. Die Konstanten werden zentral verwaltet und ermöglichen eine standardisierte Referenzierung der Objektattribute in anderen Bausteinen.

Das Modul wurde von HR Agrartechnik GmbH entwickelt, Version 1.0, erstellt von Franz Höpfinger am 20.06.2026. Es steht unter der Eclipse Public License 2.0 (EPL-2.0).

Schnittstellenstruktur

Da es sich um eine GlobalConstants-Definition handelt, existieren keine typischen Funktionsblock-Schnittstellen wie Ereignis- oder Datenein-/ausgänge. Stattdessen werden zwei globale Konstanten bereitgestellt, die in beliebigen Bausteinen referenziert werden können.

Ereignis-Eingänge

Nicht vorhanden.

Ereignis-Ausgänge

Nicht vorhanden.

Daten-Eingänge

Nicht vorhanden.

Daten-Ausgänge

Nicht vorhanden (die Konstanten sind global und können überall gelesen werden).

Adapter

Nicht vorhanden.

Global Konstanten:

Name	Typ	Initialwert	Beschreibung
`BACKGROUND_COLOUR`	USINT	`USINT#1`	1: AID_AUXI2_BACKGROUND_COLOUR – Hintergrundfarb-Index.
`FUNC`	USINT	`USINT#2`	2: AID_AUXI2_FUNC – Bitmaske: Bits 0–4 = Auxiliary-Funktionstyp (0–14), Bit 5 = Critical Control, Bit 6 = reserviert (0), Bit 7 = Single-Assignment.

Funktionsweise

Die Konstanten definieren die Attribut-IDs, die für das Auxiliary Input Type 2 Objekt im ISOBUS verwendet werden. `BACKGROUND_COLOUR` codiert die ID für den Hintergrundfarb-Index (Wert 1), während `FUNC` die ID für die Funktions-Bitmaske (Wert 2) festlegt. Diese IDs sind erforderlich, um auf die entsprechenden Attributwerte im Objekt zuzugreifen oder diese zu setzen. Durch die zentrale Definition wird eine einheitliche und fehlerresistente Nutzung im gesamten Projekt ermöglicht.

Technische Besonderheiten

- **Paketstruktur:** Die Konstanten sind im Paket `isobus::UT::Q::const::AID` organisiert, um eine klare Zuordnung zum ISOBUS-Kontext zu gewährleisten.
- **Datentyp:** Beide Konstanten sind vom Typ `USINT` (Unsigned Short Integer), was eine effiziente Speicherung und Verarbeitung erlaubt.
- **Bitdefinition:** Die `FUNC`-Konstante enthält eine definierte Bitmaske mit besonderen Bedeutungen (Funktionstyp, Critical Control, reserviertes Bit, Single-Assignment). Diese Bitstruktur ist im ISOBUS-Standard festgelegt und muss exakt eingehalten werden.
- **Initialwerte:** Die Werte sind bereits als Konstanten vordefiniert und können nicht zur Laufzeit geändert werden.

Zustandsübersicht

Für GlobalConstants existieren keine dynamischen Zustände. Die Werte sind statisch und unveränderlich, daher ist eine Zustandsüberwachung nicht erforderlich.

Anwendungsszenarien

- **ISOBUS-basierte Steuerungen:** Einsatz in landwirtschaftlichen Maschinen, die das ISOBUS-Protokoll nutzen, insbesondere zur Anbindung von Auxiliary-Eingabegeräten (Typ 2).
- **Attribut-Referenzierung:** Andere Bausteine, die auf das Objekt `Auxiliary Input Type 2` zugreifen, können auf diese Konstanten zurückgreifen, um die Attribut-IDs eindeutig zu referenzieren.
- **Code-Wartung:** Durch die zentrale Definition wird vermieden, dass magische Zahlen im Code verstreut sind; Änderungen an den IDs müssen nur an einer Stelle vorgenommen werden.

Vergleich mit ähnlichen Bausteinen

Ähnliche GlobalConstants-Datentypen existieren für andere ISOBUS-Objekte, z. B. `AID_AUXI1` oder `AID_VC` (Virtual Terminal). Diese weisen analoge Strukturen auf, unterscheiden sich jedoch in den spezifischen Attribut-IDs und deren semantischer Bedeutung. Während `AID_AUXI1` möglicherweise andere Funktionsbits definiert, umfasst `AID_AUXI2` zusätzlich Konzepte wie Critical Control und Single-Assignment. Die einheitliche Konvention erleichtert die Wiederverwendung und das Verständnis über verschiedene Module hinweg.

Fazit

AID_AUXI2 stellt eine kompakte und klar definierte Sammlung von Attribut-IDs für das Auxiliary Input Type 2 Objekt bereit. Durch die Verwendung globaler Konstanten wird die Lesbarkeit und Wartbarkeit von ISOBUS-Anwendungen im 4diac-Umfeld verbessert. Die Dokumentation der Bitstruktur in der `FUNC`-Konstante erhöht die Transparenz und reduziert potenzielle Fehlkonfigurationen. Insgesamt ist der Baustein eine sinnvolle Grundlage für die Entwicklung robuster ISOBUS-Kommunikationslösungen in der Agrartechnik.

AID_BO

• – ◦ • – ◦ • – ◦ •

Einleitung

Der globale Konstantenbaustein **AID_BO** definiert die Attribut-IDs für Button-Objekte im ISOBUS-Kontext. Er stellt numerische Konstanten bereit, die zur Identifizierung und Konfiguration von Button-Eigenschaften verwendet werden.

Schnittstellenstruktur

Ereignis-Eingänge

Keine (GlobalConstants-Baustein)

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Keine

Funktionsweise

Der Baustein enthält sechs Konstanten vom Typ `USINT`, die jeweils eine eindeutige Attribut-ID für ein Button-Objekt repräsentieren. Die Werte sind fest verdrahtet und können in ISOBUS-Programmen direkt referenziert werden, um auf die entsprechenden Attribute zuzugreifen. Die IDs sind wie folgt definiert:

Name	Wert	Beschreibung
`WIDTH`	1	Breite in Pixeln
`HEIGHT`	2	Höhe in Pixeln

BACKGROUND_COLOUR	3	Index der Hintergrundfarbe
BORDER_COLOUR	4	Index der Rahmenfarbe
KEY_CODE	5	Tastencode
OPTIONS	6	Bitmaske für Optionen (Latch, Zustand, Unterdrückung, Transparenz, etc.)

Technische Besonderheiten

- Verwendung des Typs `USINT` (8-Bit unsigned integer) für kompakte Speicherung.
- Die Konstanten sind im Paket `isobus::UT::Q::const::AID` organisiert.
- Die `OPTIONS`-Konstante enthält eine Bitmaske, deren einzelne Bits spezifische Eigenschaften steuern:
  - Bit 0: latchable
  - Bit 1: Zustand (0 = losgelassen, 1 = verriegelt)
  - Bit 2: Rahmen unterdrücken
  - Bit 3: transparenter Hintergrund
  - Bit 4: deaktiviert
  - Bit 5: kein Rahmen

Zustandsübersicht

Nicht zutreffend – es handelt sich um einen Konstantenbaustein ohne Laufzeitlogik oder Zustände.

Anwendungsszenarien

- In ISOBUS-Bediengeräten zur Definition und Anpassung von Button-Eigenschaften.
- Als zentrale Referenz für Attribut-IDs in Applikationen, die auf das UT (Universal Terminal) zugreifen.
- Zur Sicherstellung konsistenter Attributwerte über verschiedene Module hinweg.

Vergleich mit ähnlichen Bausteinen

Es existieren weitere globale Konstantenblöcke für andere Objekttypen (z. B. `AID_AL` für Alarme, `AID_OP` für Operationen), die nach demselben Muster aufgebaut sind. `AID_BO` ist auf Buttons spezialisiert und liefert die entsprechenden Attribut-IDs.

Fazit

Der Baustein `AID_BO` bietet eine einfache und typsichere Möglichkeit, die standardisierten Attribut-IDs für ISOBUS-Buttons bereitzustellen. Durch die zentrale Definition wird die Wartbarkeit und Fehlervermeidung in der Anwendungsentwicklung verbessert.

AID_CO

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `AID_CO` ist eine Sammlung globaler Konstanten, die in der ISOBUS-Kommunikation (ISO 11783) für Container-Objekte verwendet werden. Er definiert numerische Kennungen (Attribute Identifiers) für Eigenschaften eines Containers innerhalb einer landwirtschaftlichen Maschine. Diese Konstanten dienen als symbolische Referenzen, um in anderen Funktionsbausteinen die entsprechenden Attribut-IDs nicht als magische Zahlen verwenden zu müssen. Der Baustein ist als reiner Konstantencontainer ohne eigene Logik oder Schnittstellen ausgelegt.

Schnittstellenstruktur

Ereignis-Eingänge

Es sind keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge

Es sind keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Es sind keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Es sind keine Daten-Ausgänge vorhanden.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Baustein stellt drei globale Konstanten bereit, die die Attribut-IDs für Container-Objekte im ISOBUS-System repräsentieren. Diese Konstanten sind direkt in allen anderen Anwendungen des Projekts zugreifbar und können beispielsweise in Daten-Kommunikationsbausteinen oder Verarbeitungslogiken verwendet werden, um eine einheitliche und verständliche Referenz auf die jeweilige Eigenschaft eines Containers zu ermöglichen.

Die definierten Konstanten:

- `WIDTH` mit dem Wert `1` – steht für die maximale Breite des Containerbereichs in Pixeln.
- `HEIGHT` mit dem Wert `2` – steht für die maximale Höhe des Containerbereichs in Pixeln.
- `HIDDEN` mit dem Wert `3` – kennzeichnet, ob der Container und seine untergeordneten Objekte verborgen sind (0 = sichtbar, 1 = verborgen).

Technische Besonderheiten

- Der Baustein ist als GlobalConstants-Element definiert, das in der gesamten 4diac-Projektstruktur verwendet werden kann.
- Die Konstanten sind vom Typ `USINT` (Unsigned Short Integer) und mit Initialwerten versehen, die den ISOBUS-Spezifikationen entsprechen.
- Der Baustein enthält keinerlei ausführbare Logik – er ist rein deklarativ.
- Die Kommentare zu den Konstanten beschreiben die zugrunde liegenden Attribut-Bedeutungen und erleichtern die Wiederverwendung.

Zustandsübersicht

Da es sich um einen Konstantencontainer handelt, existiert kein Zustandsmodell. Der Baustein besitzt keinen internen Zustand und führt keine Zustandsübergänge aus. Die Werte der Konstanten sind statisch und unveränderlich.

Anwendungsszenarien

- **ISOBUS-Datenkommunikation**: Wenn ein ISOBUS-Netzwerk (z. B. ein Terminal) Daten über Container-Objekte austauscht, können diese Konstanten verwendet werden, um die entsprechenden Attribut-IDs in Nachrichten zu referenzieren.
- **Bildschirmdarstellung**: Bei der Darstellung eines Containers auf einem virtuellen Terminal können die Konstanten `WIDTH` und `HEIGHT` zur Berechnung der Anzeigegröße herangezogen werden.
- **Sichtbarkeitssteuerung**: Die Konstante `HIDDEN` kann verwendet werden, um über einen Logikbaustein die Sichtbarkeit eines Containers dynamisch zu steuern.

Vergleich mit ähnlichen Bausteinen

Es gibt weitere GlobalConstants-Bausteine, die ähnliche Attribut-IDs für andere ISOBUS-Objekttypen definieren, z. B. `AID_OBJ` für Objekt-Attribute oder `AID_POOL`. Im Vergleich zu diesen enthält `AID_CO` spezifisch die Kennungen für Container-Objekte. Andere Bausteine besitzen möglicherweise zusätzliche Konstanten (z. B. für Position, Farbe oder Text), während `AID_CO` nur die drei Grundattribute des Containers abdeckt.

Fazit

Der Baustein `AID_CO` stellt eine einfache, aber wichtige Grundlage für die ISOBUS-basierte Entwicklung in 4diac dar. Durch die klare Definition der Attribut-IDs wird die Lesbarkeit und Wartbarkeit des Projekts erhöht und die Gefahr von Fehlern durch falsche Zahlenwerte reduziert. Obwohl er keine Funktionalität im klassischen Sinne besitzt, ist er ein unverzichtbares Hilfsmittel für die konsistente Kommunikation mit landwirtschaftlichen Terminals.

AID_CP

• – ◦ • – ◦ • – ◦ •

Einleitung

Der GlobalConstants-Baustein `AID_CP` definiert Konstanten für Attribut-IDs des Farbpaletten-Objekts im ISOBUS-Kontext. Er ist Teil des Pakets `isobus::UT::Q::const::AID`. Der Baustein stellt eine Konstante `OPTIONS` bereit, die eine bestimmte Attribut-ID identifiziert.

Schnittstellenstruktur

Da es sich um einen GlobalConstants-Baustein handelt, besitzt er keine Ereignis-Eingänge, Ereignis-Ausgänge, Daten-Eingänge, Daten-Ausgänge oder Adapter. Stattdessen werden globale Konstanten definiert, die zur Compile-Zeit verfügbar sind.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Keine.

Funktionsweise

Der Baustein stellt die globale Konstante `OPTIONS` vom Typ `USINT` mit dem Initialwert `USINT#1` bereit. Diese repräsentiert die Attribut-ID `AID_CP_OPTIONS` für das Farbpaletten-Objekt. Der Kommentar weist darauf hin, dass der Wert normalerweise 0 sein sollte – hier wird jedoch 1 verwendet, was vermutlich eine spezifische Konfiguration oder einen Platzhalter darstellt.

Technische Besonderheiten

- Der Baustein ist als `GlobalConstants` deklariert, wodurch die Konstanten zur Übersetzungszeit aufgelöst werden und keine Laufzeitressourcen belegen.
- Die Konstante ist im Paket `isobus::UT::Q::const::AID` organisiert, was eine klare Strukturierung für ISOBUS-bezogene Konstanten bietet.
- Der Datentyp `USINT` (Unsigned Short Integer) hat eine Größe von 8 Bit.

Zustandsübersicht

Nicht relevant, da der Baustein keine Zustandslogik oder Ausführungszustände besitzt.

Anwendungsszenarien

- Bereitstellung standardisierter Attribut-IDs für ISOBUS-Kommunikation, insbesondere für Farbpaletten-Objekte.
- Verwendung in ISOBUS-Implementierungen zur Identifikation von Optionen oder Konfigurationen eines Farbpaletten-Objekts.
- Unterstützung der Wartbarkeit, indem magische Zahlen durch sprechende Namen ersetzt werden.

Vergleich mit ähnlichen Bausteinen

Ähnliche GlobalConstants-Bausteine könnten andere Attribut-IDs für unterschiedliche ISOBUS-Objekte definieren (z.B. `AID_CPC` für Farbpaletten-Command, `AID_CPP` für Farbpaletten-Properties). Im Gegensatz zu Funktionsblöcken oder Adaptern besitzen diese Bausteine keine ausführbare Logik, sondern dienen ausschließlich der Konstantendefinition.

Fazit

`AID_CP` ist ein einfacher GlobalConstants-Baustein zur Definition einer spezifischen Attribut-ID für ISOBUS-Farbpaletten. Er trägt zur Standardisierung und Strukturierung des Codes bei und erleichtert die Wartung durch die Verwendung benannter Konstanten.

AID_DM

• – ◦ • – ◦ • – ◦ •

Einleitung

Der GlobalConstants-Baustein `AID_DM` definiert die Objektattribut-IDs für das ISOBUS-Objekt „Data Mask“ (Datenmaske). Diese Konstanten ermöglichen eine einheitliche und lesbare Referenzierung der Attribute in der Applikationslogik, insbesondere für die Hintergrundfarbe und die zugeordnete Soft-Key-Maske. Sie sind Teil eines umfassenden ISOBUS-Implementierungsrahmens und dienen als zentrale Definitionsquelle für Attributkennungen.

Schnittstellenstruktur

Da es sich um einen GlobalConstants-Block handelt, besitzt er keine Ereignis- oder Datenschnittstellen im herkömmlichen Sinne. Die bereitgestellten Werte sind als globale Konstanten im Projekt verfügbar und werden von anderen Bausteinen direkt referenziert.

Ereignis-Eingänge

Nicht vorhanden – es handelt sich um einen reinen Konstantencontainer.

Ereignis-Ausgänge

Nicht vorhanden – es handelt sich um einen reinen Konstantencontainer.

Daten-Eingänge

Nicht vorhanden – es handelt sich um einen reinen Konstantencontainer.

Daten-Ausgänge

Nicht vorhanden – es handelt sich um einen reinen Konstantencontainer.

Adapter

Nicht vorhanden – es handelt sich um einen GlobalConstants-Block.

Funktionsweise

Der Baustein definiert zwei globale Konstanten:

Konstante	Datentyp	Initialwert	Beschreibung
`BACKGROUND_COLOUR`	`USINT`	`USINT#1`	Attribut-ID für den Hintergrundfarbindex der Datenmaske.
`SOFT_KEY_MASK`	`USINT`	`USINT#2`	Attribut-ID für die Soft-Key-Maske, die der Datenmaske zugeordnet ist.

Diese Konstanten können in anderen Bausteinen verwendet werden, um Attributwerte eindeutig zu identifizieren, ohne magische Zahlen im Code zu verwenden. Der Zugriff erfolgt über den qualifizierten Namen, z.B. `AID_DM.BACKGROUND_COLOUR`.

Technische Besonderheiten

- Die Konstanten sind als `USINT` (Unsigned Short Integer, 8 Bit) definiert und besitzen feste Initialwerte.
- Der Baustein ist als `GlobalConstants` deklariert, wodurch die Werte zur Laufzeit unveränderlich sind.
- Die Attribut-IDs folgen dem ISOBUS-Standard (ISO 11783) und sind spezifisch für das Objekt „Data Mask“.
- Die Werte sind konsistent mit den in der ISOBUS-Spezifikation festgelegten Attributnummern.

Zustandsübersicht

Nicht zutreffend – der Baustein besitzt keinen internen Zustandsautomaten. Er stellt ausschließlich konstante Daten bereit.

Anwendungsszenarien

- **ISOBUS-Datenmasken konfigurieren:** Beim Erstellen oder Bearbeiten einer Datenmaske können die Attributwerte über diese Konstanten gesetzt werden, z.B. um die Hintergrundfarbe zu ändern.
- **Soft-Key-Zuordnung:** Die Verknüpfung einer Soft-Key-Maske mit einer Datenmaske kann über die zugehörige Attribut-ID hergestellt werden.

- **Wartung und Lesbarkeit:** Durch die Verwendung benannter Konstanten wird der Quellcode verständlicher und weniger fehleranfällig, da die numerischen Werte zentral definiert sind.

Vergleich mit ähnlichen Bausteinen

Im ISOBUS-Umfeld existieren ähnliche Konstantenblöcke für andere Objekttypen, z.B. `AID_A` für Alarmmasken, `AID_WG` für Working Sets oder `AID_OB` für Objekte. Gemeinsam ist allen, dass sie die Attribut-IDs für die jeweilige Objektklasse definieren. `AID_DM` ist speziell auf die Anforderungen von Datenmasken zugeschnitten, unterscheidet sich jedoch nicht grundlegend in der Struktur – lediglich die enthaltenen Konstanten sind objektspezifisch.

Fazit

Der GlobalConstants-Baustein `AID_DM` ist ein essentieller Bestandteil einer ISOBUS-Anwendung, die mit Datenmasken arbeitet. Er stellt klar definierte, standardkonforme Attribut-IDs bereit und trägt so zur Robustheit und Wartbarkeit der Software bei. Durch die Verwendung dieser Konstanten wird die Implementierung lesbarer und die Einhaltung des ISOBUS-Standards gewährleistet.

AID_EIA

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `AID_EIA` (Extended Input Attributes Object Attribute IDs) definiert globale Konstanten zur Identifizierung von Attribut-IDs im Kontext erweiterter Eingabeattribute. Er dient als zentrale Referenz für eindeutige Werte, die in ISOBUS-Anwendungen verwendet werden, um Validierungsarten oder ähnliche Attributkennungen festzulegen. Durch die Kapselung dieser Konstanten wird eine konsistente und wiederverwendbare Basis für weitere Bausteine geschaffen.

Schnittstellenstruktur

Da `AID_EIA` ein **GlobalConstants**-Baustein ist, besitzt er **keine** Ereignis-, Daten- oder Adapteranschlüsse. Stattdessen stellt er ausschließlich globale Konstanten bereit, die von anderen Bausteinen über ihre Namen referenziert werden können.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Keine.

Funktionsweise

`AID_EIA` aggregiert Konstanten, die als unveränderliche Werte im globalen Kontext einer 4diac-Anwendung zur Verfügung stehen. Die definierte Konstante `VALTYPE` besitzt den Datentyp `USINT` (8-Bit-unsigned Integer) und ist mit dem Wert `1` initialisiert. Dieser Wert repräsentiert laut Kommentar eine „Validation Type“, wobei `1` bedeutet, dass ungültige Zeichen aufgelistet sind (im Gegensatz zu `0`, wo gültige Zeichen aufgeführt würden). Durch die Bereitstellung als globale Konstante können andere Funktionsbausteine auf diesen Wert zugreifen, ohne ihn lokal duplizieren zu müssen.

Technische Besonderheiten

- **Struktur**: Der Baustein ist als `GlobalConstants`-Objekt implementiert, nicht als klassischer FB mit einem Zustandsautomaten. Er wird nicht instanziiert, sondern direkt über den qualifizierten Namen referenziert (z. B. `AID_EIA.VALTYPE`).

- **Compiler-Informationen**: Der Baustein ist dem Compiler-Paket `isobus::UT::Q::const::AID` zugeordnet, was auf eine Verwendung im ISOBUS-Umfeld hindeutet.
- **Typisierung**: Die Konstante ist strikt typisiert (`USINT`), wodurch Typkonflikte bei der Verwendung vermieden werden.
- **Initialwert**: Der vordefinierte Startwert (`1`) gibt eine Standardkonfiguration vor, die in der Regel nicht verändert werden sollte, da es sich um eine semantische Festlegung handelt.

Zustandsübersicht

Da `AID_EIA` keine Logik oder Ereignisbehandlung besitzt, existiert **kein Zustandsdiagramm** und keine Laufzeit-Zustände. Der Baustein ist statisch und liefert nur konstante Daten.

Anwendungsszenarien

- **Validierung von Eingaben**: Die Konstante `VALTYPE` kann in Funktionen verwendet werden, die die Gültigkeit von Zeichenfolgen prüfen. Beispielsweise kann ein anderer FB den Wert abfragen, um zu entscheiden, ob eine Validierung als „Whitelist“ (gültige Zeichen aufgelistet) oder „Blacklist“ (ungültige Zeichen aufgelistet) erfolgt.
- **Konfiguration von Attributedefinitionen**: Im ISOBUS-Kontext können erweiterte Eingabeattribute verwendet werden, um gerätespezifische Eigenschaften zu beschreiben. `AID_EIA` stellt die zugehörigen IDs bereit.
- **Vereinheitlichung von Konstanten:** Durch die zentrale Ablage werden Duplikate und Inkonsistenzen vermieden, was die Wartung größerer Steuerungsprojekte erleichtert.

Vergleich mit ähnlichen Bausteinen

Andere `GlobalConstants`-Bausteine (z. B. `AID_K`, `AID_V` oder ähnliche) definieren ebenfalls Konstanten für unterschiedliche Attributgruppen. Im Gegensatz zu solchen Bausteinen fokussiert `AID_EIA` spezifisch auf die „Extended Input Attributes“ und bietet nur eine einzelne, aber klar definierte Konstante. Während andere Bausteine möglicherweise mehrere Konstanten bündeln, hebt sich `AID_EIA` durch seine minimale und spezialisierte Ausrichtung ab.

Fazit

`AID_EIA` ist ein schlanker GlobalConstants-Baustein, der eine essenzielle Konstante für die Validierungstypen in erweiterten Eingabeattributen bereitstellt. Seine Verwendung fördert die Lesbarkeit und Wartbarkeit von 4diac-Applikationen, da semantische Werte zentral und typisiert definiert werden. Obwohl er keine prozessurale Logik besitzt, ist er ein wichtiger Baustein für die konsistente Konfiguration in ISOBUS-basierten Automatisierungslösungen.

AID_EX

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AID_EX** ist ein Global-Constants-Objekt in der 4diac-IDE, das Konstanten für Attribut-IDs externer Objekte im ISOBUS-Kontext definiert. Diese Konstanten werden typischerweise in anderen Funktionsblöcken oder Anwendungen verwendet, um auf bestimmte Attribute eines externen Objekts zu verweisen. Der Baustein stellt eine zentrale, klar benannte Sammlung von Zahlenwerten als `USINT` (Unsigned Short Integer) bereit, was die Lesbarkeit und Wartbarkeit von Code verbessert, der auf diese IDs zugreift.

Die drei definierten Konstanten sind:

- **OPTIONS** (Wert 1) – eine Bitmaske, die Steueroptionen für referenzierte Objekte beschreibt.
- **NAME_0** (Wert 2) – die ID für das erste Namen-Attribut.
- **NAME_1** (Wert 3) – die ID für das zweite Namen-Attribut.

Diese Konstanten sind als globale Konstanten deklariert und können daher projektspezifisch oder systemweit referenziert werden.

Schnittstellenstruktur

Da es sich um ein Global-Constants-Objekt und nicht um einen Funktionsblock handelt, besitzt der Baustein **keine ereignis- oder datenbasierten Ein-/Ausgänge** und auch keine Adapter. Die Komponente dient ausschließlich der Bereitstellung von Konstantenwerten.

Ereignis-Eingänge
Keine vorhanden.

Ereignis-Ausgänge
Keine vorhanden.

Daten-Eingänge
Keine vorhanden.

Daten-Ausgänge
Keine vorhanden.

Adapter
Keine vorhanden.

Funktionsweise
Der Baustein definiert drei globale Konstanten, die als feste numerische Werte (`USINT`) zur Verfügung stehen. Diese Werte werden zur Laufzeit nicht verändert und dienen als symbolische Referenzen für Attribute im ISOBUS-Protokoll. Die Konstanten werden im Paket `isobus::UT::Q::const::AID` bereitgestellt und sind über die Compiler-Info diesem Paket zugeordnet.

Jede Konstante besitzt einen Kommentar, der ihre Bedeutung beschreibt:

- `OPTIONS` (`USINT#1`): Bitmaske, wobei Bit 0 angibt, ob eine referenzierte NAME-WS auf gelistete Objekte verweisen darf.
- `NAME_0` (`USINT#2`) und `NAME_1` (`USINT#3`): IDs für unterschiedliche Namensfelder.

Da es sich um Konstanten handelt, sind sie zur Compile-Zeit festgelegt und können in anderen Bausteinen über den qualifizierten Namen `AID_EX.OPTIONS`, `AID_EX.NAME_0` oder `AID_EX.NAME_1` referenziert werden.

Technische Besonderheiten
- **Paketzuordnung**: Der Baustein ist im Paket `isobus::UT::Q::const::AID` enthalten, was eine organisierte und hierarchische Struktur für ISOBUS-bezogene Konstanten bietet.
- **CompilerInfo**: Die definierten Konstanten werden in das Paket `isobus::UT::Q::const::AID` kompiliert, sodass sie in den erzeugten C/C++-Code als Symbole eingebunden werden.
- **Lizenz**: Das Objekt wird unter der Eclipse Public License 2.0 bereitgestellt (SPDX-Lizenz-Identifier: EPL-2.0), was die Nutzung und Weiterverbreitung regelt.
- **Typ**: Alle Konstanten sind vom Typ `USINT` (8-Bit unsigned integer), was den Wertebereich 0–255 abdeckt und für Attribut-IDs ausreichend ist.
- **Initialisierung**: Die initialen Werte sind direkt als `USINT#1`, `USINT#2` und `USINT#3` gesetzt, sodass die Konstanten ohne weitere Initialisierung sofort verwendbar sind.

Zustandsübersicht
Nicht anwendbar, da es sich um ein reines Konstanten-Objekt handelt und keine Zustandsautomaten oder Laufzeitlogik vorhanden sind.

Anwendungsszenarien
- **ISOBUS-Kommunikation**: Referenzierung von Attribut-IDs in ISOBUS-Nachrichten (z. B. Objektattribute wie Namen).
- **Konfiguration von externen Objekten**: Verwenden von `AID_EX.OPTIONS`, um Berechtigungen oder Optionen für referenzierte Objekte zu setzen.
- **Code-Wartung**: Zentrale Definition von IDs verbessert die Lesbarkeit und vermeidet „magic numbers“ in der Anwendungslogik.
- **Integration in andere Funktionsblöcke**: Andere FB können diese Konstanten in ihren eigenen Berechnungen oder Konfigurationsdaten verwenden.

Vergleich mit ähnlichen Bausteinen
Es gibt keine direkt vergleichbaren Funktionsblöcke, da `AID_EX` ein Global-Constants-Objekt ist. Ähnliche Konstanten-Bausteine könnten global definierte Werte wie Schwellwerte, Protokollkennungen oder

andere Attribut-IDs enthalten. Im Gegensatz zu typischen Funktionsblöcken besitzt es keine Eingangs-/Ausgangslogik und keine Zustände. Es dient ausschließlich der Datenhaltung auf Konstanten-Ebene.

Fazit

`AID_EX` ist ein kompakter und klar strukturierter Global-Constants-Baustein, der wichtige Attribut-IDs für ISOBUS-Externe-Objekte bereitstellt. Durch die symbolische Benennung wird die Verständlichkeit des Programms erhöht und die Fehleranfälligkeit reduziert. Die Verwendung von USINT-Typen und die Einbettung in ein Paket machen den Baustein effizient und in der 4diac-Umgebung integrierbar. Er eignet sich hervorragend als Basis für gemeinsame Konstanten in ISOBUS-basierten Projekten.

AID_EXOP

• – ◦ • – ◦ • – ◦ •

Einleitung

Der globale Konstantenblock `AID_EXOP` definiert Attribut-IDs für das ISOBUS-Objektmodell zur Identifikation von Attributen des „External Object Pointer“. Diese IDs werden verwendet, um auf externe Objektverweise, deren Namen und Standarddarstellungen zuzugreifen. Der Block ist Teil des Pakets `isobus::UT::Q::const::AID` und stellt einheitliche Konstanten für die Kommunikation in landwirtschaftlichen Maschinen gemäß ISO 11783 dar.

Schnittstellenstruktur

Dieser Block besitzt weder Ereignis- noch Datenein- oder -ausgänge, da er ausschließlich als **globale Konstantenquelle** dient. Alle Werte sind zur Laufzeit unveränderlich.

Ereignis-Eingänge

Nicht vorhanden.

Ereignis-Ausgänge

Nicht vorhanden.

Daten-Eingänge

Nicht vorhanden.

Daten-Ausgänge

Nicht vorhanden.

Adapter

Nicht vorhanden.

Bereitgestellte Konstanten (wertvoll für die Verwendung in jeglicher Logik):

Konstante	Typ	Wert	Beschreibung
`DEF_OBJ_ID`	USINT	1	Standard-Objekt-ID, die angezeigt wird, wenn eine externe ID ungültig oder `NULL` ist.
`EXT_REF_NAME_ID`	USINT	2	Attribut-ID für den Namen des externen Objektverweises.
`EXT_OBJ`	USINT	3	Attribut-ID für das externe Objekt selbst.

Funktionsweise

Die drei Konstanten repräsentieren Attributkennungen (AIDs) für den „External Object Pointer“ (EXOP) im ISOBUS-Datenmodell. Sie werden typischerweise als Parameter bei der Abfrage oder Manipulation von Objektattributen über den Objekt-Pool-Zugriff verwendet. Durch die Definition als globale Konstanten wird eine einheitliche und fehlerfreie Referenzierung über mehrere Funktionsblöcke hinweg ermöglicht.

Technische Besonderheiten

- **Datentyp:** Alle Konstanten sind vom Typ `USINT` (Unsigned Short Integer, 8 Bit).
- **Wertebereich:** Die Werte 1–3 entsprechen den standardisierten Attribut-IDs laut ISOBUS-Spezifikation.

- **Eigentum:** Der Block ist unter der Eclipse Public License 2.0 (EPL-2.0) veröffentlicht und wurde von der HR Agrartechnik GmbH entwickelt.
- **Compiler-Paket:** `isobus::UT::Q::const::AID` – die Konstanten werden in den generierten C++-Code eingebettet und sind dort als `const` verfügbar.

Zustandsübersicht

Da es sich um einen Konstantenblock handelt, existiert kein dynamischer Zustand oder eine Zustandsmaschine. Die Werte sind während der gesamten Laufzeit stabil und unveränderlich.

Anwendungsszenarien

- **ISOBUS-Implementierung:** Verwendung in Funktionsblöcken, die auf externe Objektverweise innerhalb eines ISOBUS-Terminals zugreifen müssen.
- **Objekt-Pool-Steuerung:** Referenzierung der IDs bei der Definition von Objekt-Attributen in einem Virtuellen Terminal (VT).
- **Fehlerbehandlung:** Einsatz der Konstanten zur Bestimmung einer Standard-Darstellung, falls eine externe Objekt-ID nicht aufgelöst werden kann.

Vergleich mit ähnlichen Bausteinen

Innerhalb desselben Pakets (`isobus::UT::Q::const::AID`) können weitere Konstantenblöcke existieren, die andere Attribut-IDs (z.B. für das Arbeitsmengen-Management oder den Task-Controller) definieren. `AID_EXOP` ist spezifisch auf den „External Object Pointer“ zugeschnitten und stellt nur die dafür relevanten AIDs bereit. Andere Blöcke haben ähnliche Struktur, unterscheiden sich jedoch in den enthaltenen Konstanten.

Fazit

Der globale Konstantenblock `AID_EXOP` bietet eine klar definierte und zentrale Sammlung von Attribut-IDs für den ISOBUS-Mechanismus der externen Objektverweise. Er vereinfacht die Implementierung, erhöht die Lesbarkeit des Codes und reduziert Fehlerquellen durch hartcodierte Zahlen. Durch die Verwendung als globale Konstante wird die Wartbarkeit und Wiederverwendbarkeit von ISOBUS-Anwendungen in der 4diac-Umgebung verbessert.

AID_EXRN

• – ◦ • – ◦ • – ◦ •

Einleitung

Bei `AID_EXRN` handelt es sich nicht um einen klassischen Funktionsblock, sondern um eine globale Konstantendefinition (`GlobalConstants`). Sie stellt Attribut-IDs für **External Reference Name**-Objekte im ISOBUS Universal Terminal bereit. Ziel ist es, die numerischen Kennungen für Optionen und Namen von `EXRN`-Objekten an zentraler Stelle als symbolische Konstanten verfügbar zu machen.

Die Definition ist in das Paket `isobus::UT::Q::const::AID` eingeordnet und kann über den Namen `AID_EXRN` referenziert werden.

Schnittstellenstruktur

Da es sich um eine `GlobalConstants`-Definition handelt, besitzt `AID_EXRN` keine Ereignis- oder Datenschnittstellen im Sinne eines Funktionsblocks. Stattdessen werden globale Konstanten mit festen Werten bereitgestellt.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Keine.

Die bereitgestellten Konstanten sind:

Konstante	Typ	Wert	Bedeutung
`OPTIONS`	`USINT`	1	Bitmaske für Optionen; Bit 0 = Enabled, erlaubt externe Referenz per Name
`NAME_0`	`USINT`	2	Attribut-ID für den ersten Namen des externen Referenzobjekts
`NAME_1`	`USINT`	3	Attribut-ID für den zweiten Namen des externen Referenzobjekts

Funktionsweise

Die globalen Konstanten von `AID_EXRN` werden verwendet, um Attribut-IDs für External-Reference-Name-Objekte zu referenzieren. Anstatt im Programmcode magische Zahlenwerte wie `1`, `2` oder `3` direkt zu verwenden, kann auf die symbolischen Konstanten zugegriffen werden, z. B. über `AID_EXRN.OPTIONS`, `AID_EXRN.NAME_0` und `AID_EXRN.NAME_1`.

Dadurch bleibt der Quellcode lesbar, eindeutig und wartungsfreundlich. Die Werte entsprechen den ISOBUS-Attribut-IDs für die jeweiligen Eigenschaften des External Reference Name Objekts.

Technische Besonderheiten

- Die Konstanten sind vom Typ `USINT`, also 8-Bit-Ganzzahlen ohne Vorzeichen.
- Die Initialwerte sind fest als `USINT#1`, `USINT#2` und `USINT#3` deklariert.
- Die Definition ist in das Compiler-Paket `isobus::UT::Q::const::AID` eingeordnet.
- Die Werte sind zur Compilezeit festgelegt und zur Laufzeit unveränderlich.
- `OPTIONS` ist als Bitmaske definiert; Bit 0 steuert die Aktivierung der externen Referenz über einen Namen.
- Die Bereitstellung erfolgt unter der Eclipse Public License 2.0.

Zustandsübersicht

Nicht zutreffend. `AID_EXRN` besitzt kein Zustandsverhalten und keine Zustandsmaschine. Es handelt sich um rein passive, globale Konstanten.

Anwendungsszenarien

- ISOBUS-Implementierungen für Universal-Terminal-Objekte mit externer Namensreferenz.
- Einheitliche Referenzierung von Attribut-IDs in ST- oder FBS-Code.
- Erweiterung bestehender ISOBUS-Konstantensammlungen um standardisierte `EXRN`-Attribut-IDs.
- Vermeidung von Streuwerten und Tippfehlern durch symbolische Konstanten.

Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu Funktionsblöcken besitzt `AID_EXRN` keine Logik, keine Ein-/Ausgänge und keinen Zustand. Es ist vergleichbar mit anderen `GlobalConstants`-Definitionen, die Attribut-IDs für ISO-11783-6-Objekte bündeln. Der Vorteil gegenüber direkten Zahlenwerten liegt in der Lesbarkeit, Wartbarkeit und zentralen Pflege der Kennungen.

Fazit

`AID_EXRN` ist eine kompakte globale Konstantenliste zur Unterstützung der ISOBUS-Entwicklung. Sie sorgt für klare, wiederverwendbare und leicht pflegbare Referenzen auf Attribut-IDs für External Reference Name Objekte und ergänzt die vorhandenen Konstantenpakete sinnvoll.

AID_FA_FILL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AID_FA_FILL** definiert die Attribut-IDs für das Füll-Attribut eines Objekts im ISOBUS (ISO 11783). Er stellt globale Konstanten bereit, die als Referenzwerte für Fülltyp, Farbe und Muster verwendet werden. Diese Konstanten sind Teil des Pakets `isobus::UT::Q::const::AID` und werden typischerweise bei der Erstellung oder Manipulation von Objekten mit Füll-Eigenschaften in ISOBUS-Terminals eingesetzt.

Schnittstellenstruktur

Der Baustein besitzt weder Ereignis-Eingänge, Ereignis-Ausgänge, Daten-Eingänge, Daten-Ausgänge noch Adapter, da es sich um eine reine Konstantensammlung handelt. Die Werte werden direkt als globale Konstanten zur Verfügung gestellt und können an beliebiger Stelle im 4diac-Programm referenziert werden.

Konstantenname	Datentyp	Initialwert	Beschreibung
`FILL_TYPE`	`USINT`	`1`	Kennung für die Art der Füllung: 0 = keine Füllung, 1 = mit Linienfarbe, 2 = mit spezifizierter Farbe, 3 = mit Muster.
`COLOUR`	`USINT`	`2`	Index der verwendeten Farbe für die Füllung.
`PATTERN`	`USINT`	`3`	Kennung für das Füllmuster.

Funktionsweise

Die Konstanten dienen als numerische Identifikatoren (Attribut-IDs) für das Füll-Attribut eines ISOBUS-Objekts. Bei der Kommunikation zwischen Terminal und Steuergerät werden diese Werte verwendet, um die Eigenschaften eines grafischen Objekts (z. B. eines Bereichs oder einer Form) zu setzen oder abzufragen. Der Zugriff erfolgt über die globalen Konstanten, die im 4diac-IDE als Konstantenbausteine in den Netzwerken referenziert werden können.

Technische Besonderheiten

- Der Baustein ist als `GlobalConstants`-Element definiert und enthält ausschließlich Konstanten, keine ausführbare Logik.
- Die Konstanten sind im Compiler-Paket `isobus::UT::Q::const::AID` enthalten und werden in der Regel von anderen Funktionsbausteinen oder Adaptern importiert.
- Die Lizenzierung erfolgt unter der Eclipse Public License 2.0 (EPL-2.0), was die Weiterverwendung und Anpassung erlaubt.
- Die Werte sind als `USINT` (Unsigned Short Integer) deklariert und werden mit dem Präfix `USINT#` initialisiert.

Zustandsübersicht

Da es sich um einen reinen Konstanten-Baustein handelt, existiert kein Zustandsmodell. Es gibt keine internen Zustände, Übergänge oder ereignisgesteuerte Abläufe.

Anwendungsszenarien

- **ISOBUS-Objektprogrammierung:** Ein Funktionsbaustein, der ein grafisches Objekt mit Füllung erzeugt, verwendet `AID_FA_FILL.FILL_TYPE`, `AID_FA_FILL.COLOUR` und `AID_FA_FILL.PATTERN`, um die entsprechenden Attribut-IDs zu setzen.
- **Attributabfrage:** Bei der Auswertung von ISOBUS-Nachrichten können diese Konstanten verwendet werden, um zu prüfen, ob das erhaltene Attribut dem Fülltyp, der Farbe oder dem Muster entspricht.
- **Konfiguration von Terminals:** In einem übergeordneten Konfigurationswerkzeug können die Konstanten genutzt werden, um benutzerdefinierte Füllungen für Objekte festzulegen.

Vergleich mit ähnlichen Bausteinen

Ähnliche Konstanten-Bausteine existieren für andere Objekteigenschaften, z. B. `AID_FA_LINE` für Linienattribute oder `AID_FA_LINE_STYLE` für Linienstile. Diese Bausteine folgen demselben Muster: Sie definieren numerische IDs als globale Konstanten und ermöglichen eine einheitliche Referenzierung im gesamten Programm. Der Unterschied liegt lediglich in den spezifischen Attributwerten und ihrer Semantik.

Fazit

Der Baustein `AID_FA_FILL` ist eine einfache, aber wichtige Sammlung von Konstanten, die die Attribut-IDs für Füllungen in ISOBUS-konformen Anwendungen bereitstellt. Durch die klare Definition und die zentrale Bereitstellung als globale Konstanten wird die Wartbarkeit und Lesbarkeit von 4diac-Projekten verbessert. Er eignet sich ideal für die Entwicklung von Anwendungen im Bereich der landwirtschaftlichen Elektronik, die auf ISOBUS-Kommunikation basieren.

AID_FA_FONT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AID_FA_FONT`-GlobalConstants-Block definiert die Attribut-IDs (Adressen) für das **Font-Attribut-Objekt** im ISOBUS-Protokoll. Diese Konstanten werden verwendet, um Eigenschaften wie Farbe, Größe, Schrifttyp und Stil eines Textes in der virtuellen Benutzeroberfläche (VT) zu spezifizieren. Der Block stellt eine referenzierbare, zentrale Sammlung dieser IDs bereit und erleichtert so die Konsistenz in der Applikationsentwicklung.

Schnittstellenstruktur

Der `AID_FA_FONT`-Baustein besitzt **keine** Ereignis- oder Dateneingänge/-ausgänge und auch keine Adapter. Es handelt sich um einen globalen Konstantenblock, der über eine Sammlung von `USINT`-Konstanten die Attribut-IDs für das Font-Objekt definiert.

Konstantenname	Typ	Wert	Beschreibung
`COLOUR`	`USINT`	`1`	Attribut-ID für den Farbindex des Fonts.
`SIZE`	`USINT`	`2`	Attribut-ID für die Schriftgröße.
`FONT_TYPE`	`USINT`	`3`	Attribut-ID für den Schrifttyp (Font-Familie).
`STYLE`	`USINT`	`4`	Attribut-ID für den Textstil (Bitmaske).

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

Keine vorhanden.

Funktionsweise

Die Konstanten werden als ganzzahlige Werte deklariert und dienen als Schlüsselwerte, die in ISOBUS-Nachrichten verwendet werden, um auf bestimmte Felder des Font-Attribut-Objekts zuzugreifen. Jeder Wert entspricht einer fest definierten Attribut-ID der ISO-11783-Norm. Bei der Kommunikation mit einem Virtual Terminal werden diese IDs verwendet, um die jeweilige Eigenschaft des Fonts zu setzen oder abzufragen. Der Wert `STYLE` ist dabei als Bitmaske codiert:

- Bit 0: **Bold** (fett)
- Bit 1: **Crossed Out** (durchgestrichen)
- Bit 2: **Underlined** (unterstrichen)
- Bit 3: **Italic** (kursiv)
- Bit 4: **Inverted** (invertierte Darstellung)
- Bit 5: **Flashing inverted** (blinkend invertiert)
- Bit 6: **Flashing hidden** (blinkend unsichtbar)
- Bit 7: **Proportional rendering** (proportionale Darstellung)

Die übrigen Konstanten sind einfache Indizes.

Technische Besonderheiten

- Der Baustein ist als `GlobalConstants` deklariert, d.h. die Konstanten sind global verfügbar und können in mehreren Funktionsblöcken verwendet werden.
- Die Werte sind als `USINT` (8-Bit) definiert und mit dem Präfix `USINT#` initialisiert.

- Die Bitmaske für `STYLE` erlaubt die Kombination mehrerer Stile über logische ODER-Verknüpfung.
- Das Objekt gehört zur Package `isobus::UT::Q::const::AID` und wird im Compiler-Paket `isobus::UT::Q::const::AID` referenziert.

Zustandsübersicht

Entfällt – es handelt sich um einen statischen Konstantenblock ohne Zustandsmaschine. Die Werte sind während der gesamten Laufzeit konstant.

Anwendungsszenarien

- **ISOBUS-Applikationen:** Verwendung in Steuerungssystemen, die mit einem Virtual Terminal (VT) kommunizieren und dort Texte mit unterschiedlichen Formatierungen darstellen müssen.
- **Entwicklung von ISOBUS-Funktionsblöcken:** In FB-Implementierungen, die ein Font-Attribut-Objekt konfigurieren, können die Konstanten direkt referenziert werden, um die korrekte Attribut-ID zu verwenden.
- **Test und Simulation:** Zur Implementierung von Testtreibern, die die Funktionalität des VT verifizieren.

Vergleich mit ähnlichen Bausteinen

Es existieren analoge GlobalConstants-Blöcke für andere Attribut-Objekte im ISOBUS-Standard, z.B. `AID_FA_LINE`, `AID_FA_OUTLINE` oder `AID_FA_FILL`. Diese folgen dem gleichen Muster: Sie definieren Konstanten für Attribut-IDs der jeweiligen Objektkategorie. Der vorliegende Block spezifiziert speziell die Eigenschaften des Schriftattributs und ist damit ein wichtiger Baustein für alle textbasierten Elemente einer virtuellen Benutzeroberfläche.

Fazit

Der `AID_FA_FONT`-GlobalConstants-Block stellt eine essenzielle Konstantensammlung für die Arbeit mit Schriftformatierungen in ISOBUS-Systemen bereit. Durch die klare Benennung der Attribute und die Kommentierung wird die Fehleranfälligkeit in der Applikationsentwicklung reduziert und die Lesbarkeit des Codes erhöht. Als zentrale Definitionsquelle trägt er zur Standardisierung und Wartbarkeit von ISOBUS-Projekten bei.

AID_GC

• – ◦ • – ◦ • – ◦ •

Einleitung

Der globale Konstantenblock **AID_GC** definiert die Attribut-Identifikatoren für ein Grafikkontextobjekt im ISOBUS-Protokoll. Er stellt numerische Konstanten bereit, mit denen auf die verschiedenen Eigenschaften eines grafischen Kontexts (z. B. Viewport, Farben, Schriften) zugegriffen werden kann. Diese Konstanten werden im Bereich der Virtual Terminal (VT) Anwendungen verwendet, um Objektattribute eindeutig zu referenzieren.

Schnittstellenstruktur

Da es sich um einen GlobalConstants-Block handelt, besitzt dieser keine ereignis- oder datenbasierten Ein-/Ausgänge im herkömmlichen Sinne. Stattdessen stellt er eine Sammlung globaler Konstanten bereit, die von anderen Bausteinen direkt referenziert werden können.

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

Keine vorhanden.

Funktionsweise

Der Block definiert 17 ganzzahlige Konstanten (Typ `USINT`) mit festen Werten von 1 bis 17. Jede Konstante repräsentiert eine spezifische Attribut-ID eines Grafikkontextobjekts. Die Namen sind sprechend gewählt und geben die Bedeutung der jeweiligen ID an, z. B. `VP_WIDTH` für die Breite des Viewports, `FG_COLOUR` für die Vordergrundfarbe oder `FORMAT` für den Zeichensatztypen. Diese Konstanten können in Applikationen verwendet werden, um auf Attribute von Grafikkontexten zuzugreifen, ohne die numerischen Werte direkt im Code zu verwenden.

Technische Besonderheiten

- Alle Konstanten sind vom Typ `USINT` (8-Bit unsigned integer) und werden mit expliziten Initialwerten (`USINT#1` etc.) definiert.
- Die Kommentare zu den Konstanten enthalten zusätzliche Erläuterungen, z. B. für `FORMAT` die Bedeutung des Werts (0 = Monochrom, 1 = 4-Bit Farbe, 2 = 8-Bit Farbe) und für `OPTIONS` die Bitmaske zur Steuerung von Transparenz und Farbmodus.
- Der Block ist als `GlobalConstants` gekennzeichnet und kann in 4diac IDE direkt eingebunden werden, um die Konstanten projektweit verfügbar zu machen.

Zustandsübersicht

Da es sich nicht um einen Funktionsblock mit Zustandslogik handelt, entfällt eine Zustandsübersicht.

Anwendungsszenarien

- **ISOBUS-VT-Anwendungen:** Verwendung der Konstanten zur Konfiguration von Grafikobjekten (z. B. Setzen von Viewport-Parametern, Farben oder Schriftattributen).
- **Einheitliche Referenzierung:** Vermeidung von „magic numbers“ im Code durch Verwendung sprechender Konstantennamen.
- **Projektübergreifende Wiederverwendung:** Die globalen Konstanten können in mehreren Funktionsblöcken desselben Projekts referenziert werden, um Konsistenz zu gewährleisten.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem klassischen Funktionsblock (FB) besitzt der GlobalConstants-Block **keine** Ein-/Ausgangsschnittstellen und **keine** interne Verarbeitungslogik. Er dient ausschließlich der Definition von globalen Konstanten. Ähnliche Blöcke könnten z. B. `AID_OBJECT` oder `AID_VT` sein, die ebenfalls Konstanten für andere Objekttypen bereitstellen. Der Vorteil gegenüber einer direkten Definition im Code liegt in der zentralen und typisierten Verwaltung, die eine bessere Wartbarkeit und Übersichtlichkeit ermöglicht.

Fazit

Der GlobalConstants-Block `AID_GC` ist eine wichtige Grundlage für die Entwicklung von ISOBUS-Terminal-Anwendungen in der 4diac-Umgebung. Er stellt eine klar strukturierte Sammlung von Attribut-IDs für Grafikkontexte bereit und erleichtert die Implementierung von VT-basierten HMI-Lösungen durch die Verwendung definierter Konstanten.

AID_GD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `AID_GD` (Attribute ID für Grafikdaten) ist ein globaler Konstantendefinitions-Baustein gemäß IEC 61499. Er stellt konstante Werte bereit, die als Attribut-IDs für die Kommunikation mit ISO-busfähigen Geräten dienen. Insbesondere wird hier die Konstante `FORMAT` definiert, die das Grafikformat für PNG-basierte Bilddaten festlegt. Solche Konstanten werden in der Regel in übergeordneten Systemen verwendet, um einheitliche und eindeutige Kennungen für den Austausch von Grafikinformationen zu gewährleisten.

Da es sich um einen `GlobalConstants`-Baustein handelt, werden keine ausführbaren Funktionen oder Zustandsautomaten bereitgestellt. Stattdessen dient er als zentrale Definitionsquelle für wiederverwendbare Konstanten innerhalb einer Applikation.

Schnittstellenstruktur

Ereignis-Eingänge
Es sind keine Ereignis-Eingänge vorhanden. Der Baustein besitzt keine ereignisgesteuerten Schnittstellen.

Ereignis-Ausgänge
Es sind keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge
Es sind keine Daten-Eingänge vorhanden.

Daten-Ausgänge
Es sind keine Daten-Ausgänge vorhanden. Die definierten Konstanten werden nicht über Ausgänge bereitgestellt, sondern stehen global im Kontext des Bausteins zur Verfügung.

Adapter
Es sind keine Adapter vorhanden.

Globale Konstanten (Definitionen)

Name	Typ	Initialwert	Kommentar
`FORMAT`	`USINT`	`USINT#1`	1: AID_GD_FORMAT – Grafiktyp: 0 = PNG, beschränkt auf 32-Bit-RGBA Maximum.

Funktionsweise
Der Baustein `AID_GD` definiert eine globale Konstante `FORMAT` mit dem Wert 1. Dieser Wert entspricht dem Attribut-Identifier `AID_GD_FORMAT`, der in der ISOBUS-Spezifikation (ISO 11783) für Grafikdatenobjekte verwendet wird. Der Wert 1 kennzeichnet, dass das Grafikformat PNG (Portable Network Graphics) ist, wobei die Farbtiefe auf maximal 32 Bit RGBA begrenzt ist.

Da es sich um einen GlobalConstants-Baustein handelt, werden die Konstanten beim Laden der Applikation in den Speicher übernommen und können von anderen Bausteinen referenziert werden. Sie sind während der gesamten Laufzeit unveränderlich (constant).

Technische Besonderheiten
- Der Baustein ist als `GlobalConstants` definiert und verwendet das IEC-61499-Format.
- Die Konstante `FORMAT` ist vom Typ `USINT` (Unsigned Short Integer) und wird mit dem Wert 1 initialisiert.
- Die Bedeutung des Wertes ist in der ISOBUS-Norm festgelegt: 0 = PNG, eingeschränkt auf 32-Bit-RGBA-Maximum.
- Durch die Verwendung als globale Konstante wird vermieden, dass der Wert mehrfach an verschiedenen Stellen hartcodiert werden muss, was die Wartbarkeit erhöht.
- Die Definition ist in einem Paket `isobus::UT::Q::const::AID` eingebettet, was auf eine klare Strukturierung des Codebasis hinweist.

Zustandsübersicht
Der Baustein besitzt keinen Zustandsautomaten, da er keine ausführbare Logik enthält. Es gibt keine Zustände oder Übergänge. Die Funktionalität beschränkt sich auf die einmalige Definition von Konstanten.

Anwendungsszenarien
Typische Anwendungsfälle für `AID_GD` sind:

- **ISOBUS-Kommunikation**: Der Baustein wird in Steuerungssystemen für landwirtschaftliche Maschinen eingesetzt, um Grafikdaten (z.B. für Display-Anzeigen) korrekt zu identifizieren und zu übertragen.
- **Einheitliche Konstantenverwaltung**: Durch die zentrale Definition von Attribut-IDs wird sichergestellt, dass alle beteiligten Softwaremodule denselben Wert für `FORMAT` verwenden, was Konsistenz und Interoperabilität fördert.
- **Konfiguration von Anzeigen**: In Anwendungen, die PNG-Grafiken auf ISOBUS-Terminals darstellen, wird `FORMAT` als Referenz für den Bildtyp genutzt.

Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu Funktionsblöcken (FBs), die über Ein-/Ausgänge und Verarbeitungslogik verfügen, ist `AID_GD` ein reiner Konstanten-Baustein. Ähnliche Bausteine könnten `GlobalConstants`-Definitionen für andere Attribut-IDs sein (z.B. `AID_ObjectPool`, `AID_Label` etc.), die ebenfalls nur Konstanten bereitstellen. Diese Bausteine unterscheiden sich lediglich in den definierten Konstanten und deren Bedeutung.

Fazit

`AID_GD` ist ein einfacher, aber wichtiger Bestandteil einer ISOBUS-basierten Applikation. Er stellt die konstante Attribut-ID für Grafikdaten bereit und sorgt so für eine einheitliche und normkonforme Kommunikation. Durch die klare Definition und zentrale Bereitstellung wird die Entwicklung und Wartung von Steuerungssoftware für landwirtschaftliche Maschinen vereinfacht.

AID_IB

• – ◦ • – ◦ • – ◦ •

Einleitung

Der GlobalConstants-Baustein `AID_IB` definiert die Attribut-IDs für ein **Input Boolean Objekt** im ISOBUS-Standard (ISO 11783). Diese Konstanten werden verwendet, um auf einzelne Attribute eines solchen Objekts in einem ISOBUS-Terminal zuzugreifen, z. B. für Hintergrundfarbe, Breite, Wert oder Aktivierungsstatus. Der Baustein ist Teil des Pakets `isobus::UT::Q::const::AID` und stellt eine zentrale Sammlung numerischer Identifikatoren zur Verfügung, die in der Anwendungslogik und in Protokollnachrichten referenziert werden.

Schnittstellenstruktur

Ereignis-Eingänge

Keine (definiert als GlobalConstants).

Ereignis-Ausgänge

Keine (definiert als GlobalConstants).

Daten-Eingänge

Keine – der Baustein stellt ausschließlich Konstanten bereit, keine Eingangsparameter.

Daten-Ausgänge

Keine – der Baustein ist nicht als ausführbarer Funktionsblock konzipiert.

Adapter

Keine – es werden keine Adapter bereitgestellt.

Globale Konstanten

Konstante	Datentyp	Wert	Beschreibung
`BACKGROUND_COLOUR`	`USINT`	`1`	Index der Hintergrundfarbe des Objekts.
`WIDTH`	`USINT`	`2`	Breite des Objekts in Pixeln.
`FG_COLOUR`	`USINT`	`3`	Objekt-ID eines Font-Attributes-Objekts zur Bestimmung der Schriftfarbe.
`VARIABLE_REF`	`USINT`	`4`	Objekt-ID eines Number-Variable-Objekts; wenn `NULL`, wird der Wert direkt gespeichert.
`VALUE`	`USINT`	`5`	Aktueller Wert des Eingabefelds: `0` = FALSE, `>0` = TRUE.
`ENABLED`	`USINT`	`6`	Aktivierungsstatus: `0` = deaktiviert, `1` = aktiviert.

Funktionsweise

Die Konstanten von `AID_IB` kennzeichnen die numerischen Attribut-IDs eines **Input Boolean Objekts** im ISOBUS-Protokoll. Jede ID entspricht einem spezifischen Attribut, das in Nachrichten (z. B. Get/

Set-Kommandos) verwendet wird, um Werte auszulesen oder zu setzen. Die Werte sind Teil des standardisierten Objektmodells für Virtual Terminals und ermöglichen eine einheitliche Adressierung der Objekteigenschaften. Durch die Definition als globale Konstanten können Referenzen im Quellcode zentral und wartungsfreundlich verwendet werden.

Technische Besonderheiten

- **Lizenz & Copyright:** Der Baustein ist unter der **Eclipse Public License 2.0** veröffentlicht (SPDX-Identifier: EPL-2.0).
- **Versionierung:** Version `1.0`, Stand `2026-06-20`, erstellt von Franz Höpfinger (HR Agrartechnik GmbH).
- **Compilerhinweis:** Der Quellcode ist für das Zielpaket `isobus::UT::Q::const::AID` vorgesehen.
- **Konstantentyp:** Alle Werte sind vom Typ `USINT` (Unsigned Short Integer, 8 Bit).
- **Struktur:** Der Baustein enthält keine ausführbare Logik – er dient ausschließlich der Bereitstellung von Konstanten.

Zustandsübersicht

Da es sich um einen GlobalConstants-Baustein handelt, existiert **kein Zustandsmodell**. Er besitzt weder interne Zustände noch eine Zustandsmaschine.

Anwendungsszenarien

`AID_IB` wird typischerweise in Projekten eingesetzt, die **ISOBUS-Terminals** (Virtual Terminals) steuern. Anwendungsbeispiele:

- Konfiguration eines **Input-Boolean-Objekts** (z. B. Ein-/Ausschalter) auf der Benutzeroberfläche.
- Abrufen oder Setzen von Attributen wie `VALUE` oder `ENABLED` zur Anzeige und Bedienung von Schaltern.
- Kombination mit Funktionsbausteinen, die auf die Konstanten zugreifen, um die Kommunikation mit dem Terminal zu vereinheitlichen.

Vergleich mit ähnlichen Bausteinen

Für andere GUI-Objekte existieren analoge Konstantensätze, z. B. `AID_KEY` oder `AID_OUTPUT_STRING`. Der Unterschied liegt in den spezifischen Attributen:

- **Input Boolean** besitzt Attribute wie `VALUE` (boolesch) und `ENABLED`.
- **Output String** würde z. B. `TEXT`, `FONT_ATTRIBUTE` oder `TEXT_ALIGNMENT` enthalten.
  Der `AID_IB`-Baustein ist speziell auf die Bedürfnisse von booleschen Eingabefeldern zugeschnitten, während andere Bausteine andere Attributsätze abdecken.

Fazit

Der GlobalConstants-Baustein `AID_IB` ist eine essenzielle Sammlung von Attribut-IDs für ISOBUS-Eingabefelder vom Typ Boolean. Er vereinfacht die Entwicklung von Anwendungen, indem er die numerischen Kennungen zentral und klar dokumentiert bereitstellt. Die Verwendung der Konstanten erhöht die Lesbarkeit des Codes und minimiert Fehler bei der Adressierung von Objektattributen. Durch die standardkonforme Lizenzierung und Versionierung ist er für industrielle Anwendungen geeignet.

AID_IL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `AID_IL` ist ein Global-Constants-Baustein, der die Attribut-IDs für ein **Input-List-Objekt** im ISOBUS-Kontext (ISO 11783) definiert. Diese Konstanten werden benötigt, um in der Kommunikation mit einem Universal Terminal (UT) gezielt auf Eigenschaften einer Input-Liste zuzugreifen, z. B. um deren Größe, aktuelle Auswahl oder Optionen zu setzen oder zu lesen.

Schnittstellenstruktur

Ereignis-Eingänge

Nicht vorhanden.

Ereignis-Ausgänge

Nicht vorhanden.

Daten-Eingänge

Es sind keine beschreibbaren Daten-Eingänge vorhanden. Der Baustein stellt jedoch die folgenden globalen Konstanten bereit, die in der gesamten Applikation gelesen werden können:

Konstante	Typ	Wert	Beschreibung
`WIDTH`	`USINT`	1	AID_IL_WIDTH – Breite des Objekts in Pixeln.
`HEIGHT`	`USINT`	2	AID_IL_HEIGHT – Höhe des Objekts in Pixeln.
`VARIABLE_REF`	`USINT`	3	AID_IL_VARIABLE_REF – Objekt-ID einer Number-Variable, die der Input-Liste zugeordnet ist.
`VALUE`	`USINT`	4	AID_IL_VALUE – Ausgewählter Listenindex (0 bis 254); 255 bedeutet, dass kein Element ausgewählt ist.
`OPTIONS`	`USINT`	5	AID_IL_OPTIONS – Bitmaske: Bit 0 = aktiviert (Enabled), Bit 1 = Echtzeit-Bearbeitung (Real-time editing).

Daten-Ausgänge

Nicht vorhanden.

Adapter

Nicht vorhanden.

Funktionsweise

`AID_IL` stellt fünf symbolische Konstanten bereit, die als Attribut-IDs für ein Input-List-Objekt im ISOBUS-Universal-Terminal dienen. Die IDs sind als `USINT`-Werte fest definiert und unveränderlich. Sie werden typischerweise zusammen mit ISOBUS-Funktionsbausteinen verwendet, um Attributoperationen (z. B. Setzen oder Abfragen von Attributen) auf ein Input-List-Objekt zu adressieren.

Die Konstanten decken die wichtigsten Attribute einer Input-Liste ab:

- **Geometrie** (`WIDTH`, `HEIGHT`)
- **Datenanbindung** (`VARIABLE_REF`)
- **Zustand** (`VALUE`)
- **Verhalten** (`OPTIONS`)

Durch die Verwendung dieser Konstanten wird die Lesbarkeit und Wartbarkeit des Applikationscodes verbessert, da anstelle von magischen Zahlen aussagekräftige Namen verwendet werden können.

Technische Besonderheiten

- Die Konstanten sind im Paket `isobus::UT::Q::const::AID` abgelegt.
- Sie sind als **globale Konstanten** definiert und können applikationsweit ohne zusätzliche Instanziierung verwendet werden.
- Alle Werte sind vom Typ `USINT` (8-Bit unsigned integer).
- Der Baustein enthält weder Zustandslogik noch prozedurale Anteile; er ist rein deklarativer Natur.

Zustandsübersicht

Nicht zutreffend, da es sich um einen reinen Konstanten-Baustein ohne Laufzeitlogik handelt.

Anwendungsszenarien

- **ISOBUS-Applikationen mit Input-Listen:** Zugriff auf Eigenschaften einer Input-Liste über die definierten Attribut-IDs.
- **Parametrierung von UT-Objekten:** Setzen der Breite, Höhe oder Optionen einer Input-Liste zur Laufzeit.
- **Auswertung von Benutzereingaben:** Lesen des aktuell ausgewählten Listeneintrags über die Konstante `VALUE`.
- **Kommunikation mit Universal Terminal:** Verwendung der Konstanten in Kombination mit ISOBUS-Funktionsbausteinen, die Attributoperationen auf Objekte ausführen.

Vergleich mit ähnlichen Bausteinen

Im ISOBUS-Standard besitzen verschiedene Objekttypen eigene Attribut-IDs. `AID_IL` ist speziell für **Input-List-Objekte** ausgelegt. Vergleichbare Global-Constants-Bausteine existieren für andere Objekttypen, z. B. für Number-Variablen, Output-Listen oder grafische Objekte. Der grundsätzliche Aufbau und die Verwendung sind ähnlich; die Werte der Attribut-IDs unterscheiden sich jedoch je nach Objekttyp.

Fazit

`AID_IL` ist eine kompakte und klar strukturierte Konstanten-Sammlung für die Arbeit mit ISOBUS-Input-Listen. Sie ermöglicht eine saubere, lesbare und standardkonforme Programmierung, indem sie die relevanten Attribut-IDs benennt und dokumentiert. Der Baustein ist besonders für Entwickler von ISOBUS-Applikationen nützlich, die mit Universal-Terminals und Input-Listen arbeiten.

AID_IN

• – ◦ • – ◦ • – ◦ •

Einleitung

Der GlobalConstants-Baustein `AID_IN` definiert die Attribut-IDs eines **Eingabezahlenobjekts** (Input Number Object) im ISOBUS-UT-Kontext. Diese Konstanten werden verwendet, um auf spezifische Eigenschaften des Objekts zuzugreifen, wie z. B. Breite, Höhe, Farbwerte oder Formatierungsoptionen. Die Definition erfolgt gemäß dem Standard IEC 61499-1 und ist Teil des Pakets `isobus::UT::Q::const::AID`.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Der Baustein besitzt keine Ereignis-Eingänge, da er ausschließlich konstante Werte bereitstellt.

Ereignis-Ausgänge

Keine. Es existieren keine Ereignis-Ausgänge.

Daten-Eingänge

Keine. Der Baustein besitzt keine Daten-Eingänge, da die Werte als globale Konstanten fest definiert sind.

Daten-Ausgänge

Keine. Es werden keine Datenwerte über Schnittstellen ausgegeben.

Adapter

Keine. Es sind keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Baustein `AID_IN` stellt eine Sammlung von **15 Konstanten** vom Typ `USINT` (Unsigned Short Integer) bereit. Jede Konstante entspricht einer Attribut-ID, die im ISOBUS-Protokoll für ein Eingabezahlenobjekt verwendet wird. Die Werte ermöglichen den Zugriff auf folgende Attribute:

Konstante	Wert	Beschreibung
`WIDTH`	1	Breite in Pixeln
`HEIGHT`	2	Höhe in Pixeln
`BACKGROUND_COLOUR`	3	Index für die Hintergrundfarbe
`FONT_ATT`	4	Objekt-ID eines Font-Attributobjekts
`OPTIONS`	5	Bitmaske: Bit 0 = transparent, Bit 1 = führende Nullen anzeigen, Bit 2 = Null als leer anzeigen, Bit 3 = Dezimalstellen abschneiden (sonst runden)
`VARIABLE_REF`	6	Objekt-ID eines Variablenobjekts
`MIN_VALUE`	7	Minimalwert
`MAX_VALUE`	8	Maximalwert

OFFSET	9	Offset
SCALE	10	Skalierungsfaktor
NUMB_DECIMALS	11	Anzahl der Dezimalstellen
FORMAT	12	Format: 0 = feste Dezimaldarstellung (####.nn), 1 = Exponentialformat
JUSTIFICATION	13	Ausrichtung: Bits 0–1 horizontal (0=links, 1=mittig, 2=rechts), Bits 2–3 vertikal (0=oben, 1=mittig, 2=unten)
VALUE	14	Aktueller Wert des Objekts
OPTIONS_2	15	Bitmaske: Bit 0 = aktiviert, Bit 1 = Echtzeit-Bearbeitung

Diese Konstanten werden in Anwendungen genutzt, um Attribute eines Eingabezahlenobjekts zu setzen oder zu lesen, beispielsweise bei der Konfiguration eines Bedienterminals.

Technische Besonderheiten

- **Datentyp**: Alle Konstanten sind vom Typ `USINT` (8-Bit ohne Vorzeichen) und initial mit ihrem jeweiligen Attributwert belegt.
- **Paketzuordnung**: Der Baustein gehört zum Paket `isobus::UT::Q::const::AID`, was eine klare Strukturierung im Projekt ermöglicht.
- **Lizenz**: Die Definitionen stehen unter der **Eclipse Public License 2.0** (EPL-2.0). Eine Kopie der Lizenz ist unter eclipse.org/legal/epl-2.0 verfügbar.
- **Standardkonformität**: Die Konstanten folgen dem IEC 61499-1-Standard für die Modellierung verteilter Automatisierungssysteme.

Zustandsübersicht

Nicht anwendbar. Da es sich um eine reine Konstantendefinition handelt, existieren keine Zustände oder Zustandsübergänge.

Anwendungsszenarien

- **Konfiguration eines ISOBUS-Bedienterminals**: Ein Programmierer verwendet `AID_IN.WIDTH`, um die Breite eines numerischen Eingabefelds zu definieren.
- **Dynamische Anpassung von Objekteigenschaften**: Bei der Laufzeit können Werte wie `MIN_VALUE` und `MAX_VALUE` referenziert werden, um Eingabevalidierungen durchzuführen.
- **Formatierung von Zahlen**: Über `FORMAT` und `NUMB_DECIMALS` kann die Darstellung von Dezimalzahlen gesteuert werden.
- **Zugriff auf Variablenwerte**: Über `VARIABLE_REF` kann eine Verknüpfung zu einer externen Variable hergestellt werden.

Vergleich mit ähnlichen Bausteinen

Im ISOBUS-UT-Standard existieren analoge Konstantendefinitionen für andere Objekttypen, z. B. `AID_OUT` für Ausgabezahlenobjekte oder `AID_LABEL` für Beschriftungen. Gemeinsam ist allen die Struktur als `GLOBALCONSTANTS`, die die Attribut-IDs einheitlich und typsicher bereitstellen. Der Unterschied liegt in den spezifischen Attributen, die auf den jeweiligen Objekttyp zugeschnitten sind.

Fazit

Der `AID_IN`-Baustein bietet eine klar definierte und standardisierte Sammlung von Konstanten für die Arbeit mit Eingabezahlenobjekten im ISOBUS-UT-Umfeld. Durch die zentrale Bereitstellung der Attribut-IDs wird die Programmierung erleichtert, Fehler durch magische Zahlen vermieden und die Wartbarkeit von Automatisierungsprojekten verbessert. Die vollständige Integration in das Paket- und Lizenzsystem der 4diac-IDE gewährleistet eine nahtlose Verwendung in verteilten Steuerungssystemen.

AID_IP

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `AID_IP` ist ein globaler Konstanten-Baustein (GlobalConstants) zur Definition von Attribut-IDs für Eingabeobjekte im ISOBUS-Kontext. Er stellt eine einzige Konstante `VALIDATION_TYPE` bereit, die

den Validierungstyp für Eingabeattribute festlegt. Diese Konstante wird typischerweise in Client- oder Server-Komponenten verwendet, um zu bestimmen, ob bei der Validierung von Zeichenlisten gültige oder ungültige Zeichen aufgeführt werden.

Schnittstellenstruktur

Da `AID_IP` ein GlobalConstants-Baustein ist, besitzt er keine Ereignis- oder Dateneingänge/-ausgänge im Sinne eines Funktionsblocks. Es sind keine Adapter vorhanden.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Keine.

Globale Konstanten

Konstante	Typ	Initialwert	Kommentar
`VALIDATION_TYPE`	`USINT`	`USINT#1`	1: AID_IP_VALIDATION_TYPE – Validierungstyp: 0 = gültige Zeichen sind gelistet, 1 = ungültige Zeichen sind gelistet.

Funktionsweise

Der Baustein stellt eine globale Konstante bereit, die überall im Projekt referenziert werden kann. Die Konstante `VALIDATION_TYPE` wird als `USINT` (Unsigned Short Integer) mit dem Wert `1` initialisiert. Gemäß Kommentar bedeutet der Wert `1`, dass bei einer Validierung die Liste der **ungültigen** Zeichen angegeben wird. Ein Wert `0` würde bedeuten, dass gültige Zeichen aufgeführt sind.

Die Konstante ist als `VAR_GLOBAL CONSTANT` definiert und besitzt damit im gesamten Anwendungsbereich Gültigkeit. Sie kann in FB-Implementierungen oder anderen Bausteinen per Namen angesprochen werden, ohne dass eine Instanziierung erforderlich ist.

Technische Besonderheiten

- Der Typ `USINT` entspricht einer 8-Bit-Ganzzahl ohne Vorzeichen.
- Der Initialwert wird explizit als `USINT#1` gesetzt, was eine typsichere Initialisierung darstellt.
- Der Baustein enthält keine Logik oder Zustände; er dient rein der Bereitstellung von Konstanten für die ISOBUS-Objektattribut-Verarbeitung.
- Der Paketname `isobus::UT::Q::const::AID` deutet auf eine hierarchische Struktur innerhalb eines ISOBUS-Frameworks hin.

Zustandsübersicht

Nicht zutreffend – dieser Baustein besitzt keine Zustände.

Anwendungsszenarien

- **Validierung von Eingabeattributen:** In ISOBUS-Datenbanken müssen Eingabefelder (z.B. für Benutzereingaben) validiert werden. Mit `VALIDATION_TYPE` kann festgelegt werden, ob die übermittelte Zeichenliste gültige oder ungültige Zeichen enthält.
- **Parametrierung von Client-Objekten:** Wenn ein Client ein Eingabeobjekt mit Attribut-IDs konfiguriert, kann er die Konstante nutzen, um den Validierungsmodus festzulegen.
- **Einheitliche Konfiguration:** Durch die zentrale Definition wird vermieden, dass Magische Zahlen im Code auftauchen; Änderungen können an einer Stelle vorgenommen werden.

Vergleich mit ähnlichen Bausteinen

Im ISOBUS-Kontext existieren weitere GlobalConstants-Bausteine für andere Objekttypen (z.B. `AID_*` für verschiedene Attribut-IDs). Diese Bausteine folgen dem gleichen Muster: Sie definieren Konstanten für spezifische Parameter. `AID_IP` ist speziell auf Eingabeobjekte (Input) ausgerichtet und stellt den Validierungstyp bereit. Andere Bausteine können z.B. Attribut-IDs für Sichtbarkeit, Editierbarkeit oder Formatierungen enthalten, ohne dass sie sich in der grundlegenden Funktionsweise unterscheiden.

Fazit

Der Baustein `AID_IP` ist ein einfacher, aber wichtiger Bestandteil zur Vereinheitlichung von ISOBUS-Konstanten. Er definiert den Validierungstyp für Eingabeattribute und trägt so zur Klarheit und Wartbarkeit des Codes bei. Durch die Verwendung globaler Konstanten wird die Konfiguration zentralisiert und erleichtert die Anpassung an unterschiedliche Anforderungen.

AID_IS

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `AID_IS` ist ein globaler Konstanten-Baustein (GlobalConstants) im Kontext des ISO 11783 (ISOBUS) Standards. Er definiert numerische Konstanten, die als Attribut-IDs für das **Input String Object** verwendet werden. Diese IDs werden benötigt, um in der ISOBUS-Kommunikation auf bestimmte Eigenschaften von Textfeldern (z. B. Breite, Höhe, Hintergrundfarbe) zuzugreifen. Der Baustein stellt die Konstanten als `USINT`-Werte (0 bis 255) bereit und erleichtert so die lesbare Programmierung von ISOBUS-Objekten.

Schnittstellenstruktur

Da es sich um einen globalen Konstanten-Baustein handelt, besitzt er keine klassischen Ein-/Ausgänge im Sinne eines Funktionsblocks. Die folgenden Abschnitte sind daher leer oder zeigen explizit „Keine“.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Keine.

Funktionsweise

Der Baustein definiert neun Konstanten, die jeweils eine bestimmte Attribut-ID für das ISOBUS-Input-String-Objekt repräsentieren. Diese Konstanten können in anderen Bausteinen referenziert werden, um die korrekte ID numerisch zu verwenden, ohne hartkodierte Zahlen im Code zu benötigen.

Die im Baustein enthaltenen Konstanten sind:

Konstante	Wert	Beschreibung
`WIDTH`	1	Breite des Textfelds in Pixeln (AID_IS_WIDTH).
`HEIGHT`	2	Höhe des Textfelds in Pixeln (AID_IS_HEIGHT).
`BACKGROUND_COLOUR`	3	Farbindex des Hintergrunds (AID_IS_BACKGROUND_COLOUR).
`FONT_ATT`	4	Objekt-ID eines Font-Attributes-Objekts (AID_IS_FONT_ATT).
`INP_ATT`	5	Objekt-ID eines Input-Attributes- oder Extended-Input-Attributes-Objekts (AID_IS_INP_ATT).

OPTIONS	6	Bitmaske für Optionen, z. B. Umrahmung oder Sichtbarkeit (AID_IS_OPTIONS).
VARIABLE_REF	7	Objekt-ID eines String-Variable-Objekts, das den Textinhalt bereitstellt (AID_IS_VARIABLE_REF).
JUSTIFICATION	8	Ausrichtung des Textes – Bits 0–1 horizontal (0=links, 1=mittig, 2=rechts), Bits 2–3 vertikal (0=oben, 1=mittig, 2=unten).
ENABLED	9	Status des Feldes – 0 = deaktiviert, 1 = aktiviert (AID_IS_ENABLED).

Diese Konstanten werden als `USINT` mit den entsprechenden Initialwerten definiert. Sie sind über den Paketnamen `isobus::UT::Q::const::AID` referenzierbar.

Technische Besonderheiten

- **Typ**: `GLOBALCONSTANTS` – der Baustein ist kein ausführbarer FB, sondern eine Sammlung von Konstanten.
- **Datentyp**: Alle Werte sind vom Typ `USINT` (8-Bit unsigned integer), passend zu den ISOBUS-Attribut-IDs.
- **Kompatibilität**: Der Baustein ist für die Verwendung im Rahmen des Eclipse 4diac-IDE und des IEC 61499-Modells konzipiert.
- **Lizenz**: Die Definition steht unter der Eclipse Public License 2.0 (EPL-2.0).
- **Namensraum**: Die Konstanten sind im Paket `isobus::UT::Q::const::AID` organisiert, was eine klare hierarchische Einordnung ermöglicht.

Zustandsübersicht

Da es sich um einen reinen Konstanten-Baustein handelt, gibt es keine Zustände oder Zustandsübergänge. Der Baustein ist statisch und unveränderlich.

Anwendungsszenarien

- **ISOBUS-Terminal-Applikationen**: Bei der Erstellung von Bedienoberflächen für landwirtschaftliche Maschinen müssen Attribute von Input-String-Objekten gesetzt oder gelesen werden. Die Konstanten können direkt in FB-Programmen verwendet werden, um die Attribut-IDs zu referenzieren.
- **Objekt-Pool-Konfiguration**: Beim Aufbau eines ISOBUS-Objektpools (e.g. per Array) helfen diese Konstanten, eindeutige und lesbare Bezeichnungen für die Attribut-IDs zu verwenden.
- **Entwicklung und Wartung**: Durch die zentrale Definition wird die Änderbarkeit von IDs erleichtert, falls der Standard angepasst wird oder Hersteller-spezifische Erweiterungen nötig sind.

Vergleich mit ähnlichen Bausteinen

Im ISOBUS-Kontext gibt es weitere globale Konstanten-Bausteine, z. B. für andere Objekttypen (Object IDs, Arbeitsaufträge). Der Baustein `AID_IS` spezialisiert sich ausschließlich auf das Input-String-Objekt und ist daher gezielt für Textfelder einsetzbar. Andere Bausteine wie `AID_OBJ` oder `AID_FONT` decken weitere Attributbereiche ab. Der Vorteil von `AID_IS` liegt in der klaren Trennung und Benennung der IDs für Textattribute, wodurch der Programmcode verständlicher und weniger fehleranfällig wird.

Fazit

`AID_IS` ist ein nützlicher Hilfsbaustein für alle Entwickler, die mit ISOBUS-Objekten und insbesondere mit Textfeldern arbeiten. Er bietet eine definierte, zentrale Sammlung von Attribut-IDs in einem standardisierten Format. Obwohl er selbst keine Funktionalität besitzt, vereinfacht er die Implementierung und Pflege von ISOBUS-Kommunikationslogiken erheblich und trägt zur Lesbarkeit und Wartbarkeit des Codes bei.

AID_KG

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `AID_KG` (Attribute Identifier – Key Group) ist ein globaler Konstantencontainer innerhalb des ISOBUS-Anwendungskontexts. Er stellt numerische Konstanten bereit, die als Attributed-IDs für Schlüsselgruppen (Key Groups) verwendet werden. Diese Konstanten ermöglichen eine standardisierte

Identifikation von Objekteigenschaften wie Optionen und Namen, wodurch die Interoperabilität zwischen verschiedenen ISOBUS-Komponenten erleichtert wird.

Schnittstellenstruktur

Da `AID_KG` als reine Konstantensammlung definiert ist, besitzt es keine klassischen Schnittstellen eines Funktionsblocks. Es werden weder Ereignisse noch Datenports oder Adapter bereitgestellt. Die Struktur eines solchen Containers besteht ausschließlich aus den deklarierten Konstanten.

Ereignis-Eingänge

Keine vorhanden – der Baustein enthält keine Ereignis-Eingänge.

Ereignis-Ausgänge

Keine vorhanden – der Baustein enthält keine Ereignis-Ausgänge.

Daten-Eingänge

Keine vorhanden – der Baustein besitzt keine Dateneingangsports.

Daten-Ausgänge

Keine vorhanden – der Baustein besitzt keine Datenausgangsports.

Adapter

Keine vorhanden – der Baustein enthält keine Adapter.

Funktionsweise

Der Baustein definiert zwei globale Konstanten, die in anderen Bausteinen oder Systemen referenziert werden können:

- **`OPTIONS`**: Der Wert `USINT#1` (dezimal 1) repräsentiert die Attributed-ID `AID_KG_OPTIONS`. Diese ID ist als Bitmaske definiert, bei der Bit 0 die Verfügbarkeit (Available) und Bit 1 die Transparenz (Transparent) kennzeichnet. Sie wird verwendet, um Optionen einer Schlüsselgruppe zu identifizieren.
- **`NAME`**: Der Wert `USINT#2` (dezimal 2) steht für `AID_KG_NAME` und referenziert ein Objekt (z. B. eine Ausgangszeichenkette oder einen Objektzeiger), das den Namen der Schlüsselgruppe enthält.

Die Konstanten sind als `USINT` (Unsigned Short Integer) typisiert und besitzen feste Initialwerte, die während der Laufzeit nicht geändert werden können. Durch die zentrale Definition werden Doppelbelegungen und Inkonsistenzen vermieden.

Technische Besonderheiten

- **Typisierung**: Die Konstanten sind vom Typ `USINT`, also 8-Bit-Ganzzahlen ohne Vorzeichen, was eine effiziente Speichernutzung ermöglicht.
- **Bitmaskenlogik**: Die Konstante `OPTIONS` verwendet eine Bitmaske, um mehrere Eigenschaften in einem einzigen Wert zu kodieren – Bit 0 für „Verfügbar“ und Bit 1 für „Transparent“. Diese Kodierung erlaubt eine kompakte Repräsentation und einfache Abfragen.
- **Initialisierung**: Die Initialwerte sind direkt im Deklarationstext hinterlegt (`USINT#1` bzw. `USINT#2`), sodass beim Laden des Bausteins keine zusätzliche Initialisierungssequenz erforderlich ist.
- **Namensgebung**: Die Konstantenbezeichnungen folgen einem präfixierten Schema (`AID_KG_...`), was die Zuordnung zur Schlüsselgruppe erleichtert.

Zustandsübersicht

Da `AID_KG` kein funktionaler Baustein mit einem Zustandsautomaten ist, existiert kein expliziter Zustandsübergang. Die Konstanten sind statisch und verändern ihren Wert während der Laufzeit nicht. Der Baustein kann als „permanent aktiv“ betrachtet werden, ohne dass ein Zustandsmanagement erforderlich ist.

Anwendungsszenarien

- **ISOBUS-Implementierungen**: Der Baustein wird in ISOBUS-Anwendungen eingesetzt, um Schlüsselgruppen-Attribute zu identifizieren. Beispielsweise dient `NAME` dazu, den Namen einer Schlüsselgruppe an eine Anzeige oder ein Bedienfeld zu übergeben.

- **Konfigurationssysteme**: Bei der Konfiguration von Maschinensteuerungen können die Konstanten verwendet werden, um auf einheitliche Attribut-IDs zu verweisen, ohne magische Zahlen im Code zu verwenden.
- **Interoperabilität**: Durch die zentrale Definition können verschiedene Module eines Systems die gleichen IDs verwenden und so konsistente Datenstrukturen sicherstellen.

Vergleich mit ähnlichen Bausteinen

Andere Konstantencontainer wie `AID_COMM`, `AID_COM` oder `AID_OBJ` (nicht im Lieferumfang) folgen typischerweise demselben Muster: Sie definieren numerische IDs für verschiedene Objekttypen (z. B. Kommunikation, gemeinsame Objekte). Im Gegensatz dazu fokussiert `AID_KG` speziell auf Schlüsselgruppen-Attribute. Ein direkter Vergleich zeigt, dass die Struktur aller dieser Container ähnlich ist – sie bestehen aus globalen Konstanten ohne Schnittstellen –, jedoch unterschiedliche Satellitenbezeichnungen verwenden, um die jeweiligen Bereiche abzudecken.

Fazit

Der Baustein `AID_KG` ist ein schlanker, aber essenzieller Bestandteil für ISOBUS-basierte Systeme. Er definiert klar benannte und typisierte Konstanten für Schlüsselgruppen-Attribute und reduziert dadurch die Fehleranfälligkeit bei der Verwendung numerischer IDs. Obwohl er keine dynamischen Funktionen besitzt, trägt er durch seine statische Natur zur Robustheit und Wartbarkeit des Gesamtsystems bei. Die klare Struktur und die dokumentierte Bedeutung der Konstanten machen ihn zu einem verlässlichen Baustein für Entwickler und Integratoren.

AID_LA

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `AID_LA` ist ein Satz globaler Konstanten, der die Attribut-IDs für Linienobjekte (Line Attributes) im ISOBUS-Protokoll definiert. Diese Konstanten werden verwendet, um auf spezifische Eigenschaften von Linienobjekten zuzugreifen, beispielsweise Farbe, Breite und Art der Linie. Sie dienen als gemeinsame Referenz für andere Bausteine und Anwendungen, die mit Linien in der virtuellen Benutzeroberfläche arbeiten.

Schnittstellenstruktur

Da es sich bei `AID_LA` um eine reine Konstantendefinition handelt, besitzt der Baustein keine Ereignis- oder Datenein-/ausgänge. Die bereitgestellten Informationen sind als globale Konstanten verfügbar und können direkt über die Namen referenziert werden.

Ereignis-Eingänge

Nicht zutreffend – Es sind keine Ereignis-Eingänge definiert.

Ereignis-Ausgänge

Nicht zutreffend – Es sind keine Ereignis-Ausgänge definiert.

Daten-Eingänge

Nicht zutreffend – Es sind keine Daten-Eingänge definiert.

Daten-Ausgänge

Nicht zutreffend – Es sind keine Daten-Ausgänge definiert.

Adapter

Nicht zutreffend – Es sind keine Adapter definiert.

Funktionsweise

Der Baustein stellt drei Konstanten vom Typ `USINT` (Unsigned Short Integer) bereit:

Name	Wert	Kommentar
`LINE_COLOUR`	1	AID_LA_LINE_COLOUR – ID für die Linienfarbe
`LINE_WIDTH`	2	AID_LA_LINE_WIDTH – ID für die Linienbreite

LINE_ART	3	AID_LA_LINE_ART – Bitmuster für die Linienart (jedes Bit repräsentiert einen Pinselpunkt)

Diese Konstanten werden typischerweise als Schlüsselwerte verwendet, um in Objektlisten auf die jeweilige Eigenschaft einer Linie zuzugreifen. Die Werte sind fest definiert und während der Laufzeit nicht veränderbar.

Technische Besonderheiten

- **Datentyp:** Alle Konstanten sind als `USINT` deklariert, was einen Wertebereich von 0 bis 255 ermöglicht und im Speicher nur ein Byte belegt.
- **Initialwerte:** Die Werte sind mit `USINT#1`, `USINT#2` und `USINT#3` vorbelegt und werden beim Systemstart geladen.
- **Zugehörigkeit:** Der Baustein ist Teil des Pakets `isobus::UT::Q::const::AID` und wird über den Compiler als globale Konstantenquelle eingebunden.
- **Kommentar:** Der Kommentar gibt den Zweck der einzelnen Konstanten an und dient der Selbstbeschreibung im Quellcode.

Zustandsübersicht

Da es sich um reine Datenkonstanten handelt, gibt es keinen Zustandsautomaten oder dynamisches Verhalten. Die Werte sind statisch und stehen ab der Initialisierung des Systems zur Verfügung.

Anwendungsszenarien

- **ISOBUS-Implementierung:** Verwendung in Objekten, die Linienattribute darstellen, z.B. für den Zugriff auf die Linienfarbe oder -breite in einem grafischen Terminal.
- **Objektdefinition:** Als Referenz in Verbindung mit Objekt-IDs, um die Eigenschaften eines Linienobjekts eindeutig zu identifizieren.
- **Konfiguration:** Zur Parametrierung von Anzeigeelementen in landwirtschaftlichen Maschinensteuerungen, die auf dem ISOBUS-Standard basieren.

Vergleich mit ähnlichen Bausteinen

Ähnliche Konstantendefinitionen existieren für andere Objektattribute (z.B. `AID_OBJ` für allgemeine Objekte oder `AID_OP` für Bedienelemente). Diese folgen demselben Muster, unterscheiden sich jedoch in der Anzahl und Bedeutung der definierten Konstanten. `AID_LA` fokussiert ausschließlich auf Linienattribute und bietet eine kompakte, klar abgegrenzte Menge von IDs, die leicht in andere Bausteine integriert werden können.

Fazit

`AID_LA` ist eine einfache und dennoch essentielle Konstantendefinition für die Arbeit mit Linienobjekten im ISOBUS-Umfeld. Sie stellt eindeutige IDs bereit, die eine standardisierte Kommunikation und Konfiguration ermöglichen. Durch die Verwendung von `USINT`-Werten bleibt der Speicherbedarf minimal und die Kompatibilität zu anderen Systembausteinen gewährleistet. Die Struktur ist klar und erweiterbar, falls zukünftige Versionen weitere Linienattribute benötigen.

AID_NV

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `AID_NV` ist eine globale Konstantendefinition im Bereich der ISOBUS-Kommunikation (ISO 11783). Er definiert Attribut-IDs für **Number Variable Objects (NV)**. Diese Objekte dienen dazu, numerische Werte (z. B. Messwerte oder Einstellungen) zwischen Steuergeräten und Terminals auszutauschen. Die hier bereitgestellte Konstante `VALUE` referenziert das Attribut **1**, welches den aktuellen Wert (`AID_NV_VALUE`) eines NV-Objekts kennzeichnet.

Schnittstellenstruktur

Da `AID_NV` ein `GlobalConstants`-Baustein ist, besitzt er keine klassischen Ein-/Ausgangs-Schnittstellen.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge
Keine.

Daten-Eingänge
Keine.

Daten-Ausgänge
Keine (es werden keine Werte exportiert, sondern symbolische Konstanten zur Verfügung gestellt).

Adapter
Keine.

Funktionsweise
`AID_NV` definiert eine einzige globale Konstante:

- `VALUE : USINT = 1`
  Diese Konstante entspricht der Attribut-ID `1`, die laut ISO-11783 den **aktuellen Wert** eines *Number Variable Object* angibt. In der Anwendung kann diese Konstante verwendet werden, um auf das Attribut eines NV-Objekts zuzugreifen, ohne magische Zahlen hartkodieren zu müssen.

Die Definition erfolgt im Paket `isobus::UT::Q::const::AID` und ist damit für alle Module des ISOBUS-Stacks zugänglich.

Technische Besonderheiten
- Der Baustein ist als `GlobalConstants` mit **konstantem** Wert (`CONSTANT`) deklariert und kann nicht zur Laufzeit verändert werden.
- Der Quellcode ist in der Anfangssprache ST (Structured Text) verfasst und wird durch den 4diac-Compiler zu einem entsprechenden C-Code umgesetzt.
- Die Copyright-Hinweise weisen auf die Eclipse Public License 2.0 (EPL-2.0) hin.
- Die Version 1.0 wurde am 20.06.2026 von Franz Höpfinger erstellt.

Zustandsübersicht
Nicht zutreffend – der Baustein besitzt keinen Zustandsautomaten, da er nur Konstanten bereitstellt.

Anwendungsszenarien
- **ISOBUS-Anwendungen:** Zugriff auf die Attribut-ID des aktuellen Werts eines *Number Variable Object* (z. B. bei der Konfiguration eines Terminals oder bei der Übertragung von Messdaten).
- **Code-Lesbarkeit:** Vermeidung von „Magic Numbers“ durch Verwendung aussagekräftiger Konstanten wie `AID_NV.VALUE`.
- **Wartung:** Änderungen an der ISOBUS-Spezifikation können zentral in dieser Konstantendefinition angepasst werden, ohne dass jede Verwendungsstelle angepasst werden muss.

Vergleich mit ähnlichen Bausteinen
Es existieren weitere globale Konstantendefinitionen für andere ISOBUS-Objekttypen, z. B. für *Data Dictionary* (`AID_DD`) oder *Process Variable* (`AID_PV`). Während diese ähnlich aufgebaut sind, ist `AID_NV` speziell auf *Number Variable Objects* ausgerichtet und definiert nur die hier benötigte Attribut-ID. Andere Bausteine können zusätzliche Konstanten für weitere Attribute (z. B. minimaler/maximaler Wert, Einheit) enthalten.

Fazit
`AID_NV` ist eine einfache, aber wichtige Konstantendefinition für ISOBUS-Anwendungen. Sie stellt sicher, dass die Attribut-ID für den aktuellen Wert eines *Number Variable Object* einheitlich und fehlerfrei referenziert wird. Durch die Einbettung in eine globale Konstantenstruktur wird die Wartbarkeit und Lesbarkeit des Gesamtsystems verbessert.

AID_OABG
• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `AID_OABG` ist ein GlobalConstants-Baustein, der Konstanten für Attribut-IDs eines „Output arched bar graph object“ (bogenförmiges Balkendiagramm-Objekt) im ISOBUS-Kontext definiert. Diese Konstanten werden verwendet, um auf die verschiedenen Attribute eines solchen Objekts zuzugreifen, z. B. Breite, Höhe, Farbe, Optionen, Winkel usw. Der Baustein ist Teil des Pakets `isobus::UT::Q::const::AID`.

Schnittstellenstruktur

Da es sich um einen GlobalConstants-Baustein handelt, besitzt er keine Ereignis-Eingänge oder -Ausgänge und auch keine Adapter. Die Schnittstelle besteht ausschließlich aus globalen Konstanten, die als Datenwerte zur Verfügung stehen. Im Folgenden sind die Konstanten als „Daten-Ausgänge“ aufgeführt, da sie von anderen Bausteinen gelesen werden können.

Ereignis-Eingänge

Nicht vorhanden.

Ereignis-Ausgänge

Nicht vorhanden.

Daten-Eingänge

Nicht vorhanden.

Daten-Ausgänge

Name	Typ	Wert	Kommentar
`WIDTH`	USINT	1	AID_OABG_WIDTH – Breite in Pixeln
`HEIGHT`	USINT	2	AID_OABG_HEIGHT – Höhe in Pixeln
`COLOUR`	USINT	3	AID_OABG_COLOUR – Farbindex
`TARGET_LINE_COLOUR`	USINT	4	AID_OABG_TARGET_LINE_COLOUR – Farbe der Ziellinie
`OPTIONS`	USINT	5	AID_OABG_OPTIONS – Bitmaske: Bit 0=Border zeichnen, Bit 1=Ziellinie zeichnen, Bit 2=undefiniert (0), Bit 3=Typ (0=geflutet, 1=einzelne Linie), Bit 4=Abweichung (0=gegen den Uhrzeigersinn, 1=im Uhrzeigersinn)
`START_ANGLE`	USINT	6	AID_OABG_START_ANGLE – Startwinkel
`END_ANGLE`	USINT	7	AID_OABG_END_ANGLE – Endwinkel
`BAR_GRAPH_WIDTH`	USINT	8	AID_OABG_BAR_GRAPH_WIDTH – Breite des Balkendiagramms
`MIN_VALUE`	USINT	9	AID_OABG_MIN_VALUE – Minimalwert
`MAX_VALUE`	USINT	10	AID_OABG_MAX_VALUE – Maximalwert
`VARIABLE_REF`	USINT	11	AID_OABG_VARIABLE_REF – Objekt-ID eines Variable-Objekts
`TARGET_VAL_VAR_REF`	USINT	12	AID_OABG_TARGET_VAL_VAR_REF – Referenz auf die Zielwertvariable
`TARGET_VALUE`	USINT	13	AID_OABG_TARGET_VALUE – Zielwert
`VALUE`	USINT	14	AID_OABG_VALUE – aktueller Wert

Adapter

Nicht vorhanden.

Funktionsweise

Der Baustein definiert ausschließlich Konstanten, die als numerische Kennungen für die Attribute eines ISOBUS-Objekts vom Typ „Output arched bar graph“ dienen. Er besitzt keine eigene Logik oder Verarbeitungsfunktion. Er stellt die Attribut-IDs bereit, die von anderen Funktionsbausteinen verwendet werden, um auf die entsprechenden Attribute zuzugreifen, z. B. beim Setzen oder Lesen von Objekteigenschaften über die ISOBUS-Kommunikation.

Technische Besonderheiten

- Die Konstanten sind als `USINT` (Unsigned Short Integer) definiert und mit festen Werten belegt.
- Sie sind im Paket `isobus::UT::Q::const::AID` zusammengefasst.
- Die Bitmasken für `OPTIONS` sind im Kommentar dokumentiert.
- Es handelt sich um einen GlobalConstants-Baustein, d. h. die Konstanten sind global verfügbar und nicht an eine Instanz gebunden.

Zustandsübersicht

Nicht zutreffend, da es sich um einen reinen Konstanten-Baustein ohne Zustandslogik handelt.

Anwendungsszenarien

- Verwendung in ISOBUS-Applikationen, um ein bogenförmiges Balkendiagramm auf einem Terminal zu konfigurieren.
- Zugriff auf Attribute wie Breite, Höhe, Farben, Winkel, Minimal-/Maximalwert und Zielwert.
- Einbindung in Funktionsbausteine, die Objektattribute über die ISOBUS-Schnittstelle setzen oder abfragen.

Vergleich mit ähnlichen Bausteinen

Es gibt ähnliche GlobalConstants-Bausteine für andere ISOBUS-Objekttypen, z. B. `AID_OP` für Output-Objekte oder `AID_OV` für Variablen. Diese folgen dem gleichen Muster: Sie definieren Konstanten für Attribut-IDs, um eine einheitliche und typsichere Referenzierung zu ermöglichen. `AID_OABG` ist speziell auf das bogenförmige Balkendiagramm zugeschnitten.

Fazit

`AID_OABG` ist ein einfacher, aber wichtiger GlobalConstants-Baustein, der die Attribut-IDs für ein spezielles ISOBUS-Anzeigeelement bereitstellt. Er erleichtert die Entwicklung von ISOBUS-Anwendungen, indem er symbolische Namen für die numerischen Kennungen liefert und so die Lesbarkeit und Wartbarkeit des Codes verbessert.

AID_OE

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein AID_OE ist ein globaler Konstanten-Container (GlobalConstants) aus der 4diac-IDE, der die Objektattribut-IDs (Attribute IDs) für ein „Output Ellipse“-Objekt im ISOBUS-Standard (ISO 11783) definiert. Diese Konstanten werden verwendet, um Attribute wie Linienstärke, Größe, Ellipsentyp oder Füllbereich eindeutig zu identifizieren. Der Baustein stellt keine eigene Funktionalität bereit, sondern dient als zentrale Definitionsquelle für andere Bausteine im System.

Schnittstellenstruktur

Da AID_OE ausschließlich globale Konstanten bereitstellt, besitzt er keine ereignis- oder datenbasierten Ein-/Ausgänge im klassischen Sinne. Die folgenden Abschnitte sind daher leer bzw. nicht zutreffend.

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Der Baustein stellt die folgenden Konstanten als feste Werte bereit, die als „Daten-Ausgänge“ interpretiert werden können:

Konstante	Typ	Wert	Beschreibung
`LINE_ATT`	USINT	1	Objekt-ID eines Linien-Attribut-Objekts (Line Attributes)
`WIDTH`	USINT	2	Breite in Pixeln
`HEIGHT`	USINT	3	Höhe in Pixeln
`ELLIPSE_TYPE`	USINT	4	Ellipsentyp: 0 = geschlossene Ellipse, 1 = offene Ellipse, 2 = geschlossenes Segment, 3 = geschlossener Sektor
`START_ANGLE`	USINT	5	Startwinkel der Ellipse
`END_ANGLE`	USINT	6	Endwinkel der Ellipse
`FILL_ATT`	USINT	7	Objekt-ID eines Füll-Attribut-Objekts (Fill Attributes)

Adapter

Keine vorhanden.

Funktionsweise

AID_OE definiert sieben numerische Konstanten vom Typ `USINT` (Unsigned Short Integer), die jeweils eine spezifische Attribut-ID für ein Ellipsen-Objekt im ISOBUS-Kontext repräsentieren. Diese IDs werden in anderen Funktionsbausteinen verwendet, um auf die entsprechenden Eigenschaften eines grafischen Ellipsenelements zuzugreifen oder diese zu setzen. Da es sich um globale Konstanten handelt, sind die Werte während der gesamten Laufzeit unveränderlich und stehen allen beteiligten Bausteinen zur Verfügung.

Technische Besonderheiten

- Die Konstanten sind als `USINT` (8-Bit ohne Vorzeichen) definiert und besitzen feste Initialwerte.
- Der gesamte Definitionsblock ist unter dem Paketnamen `isobus::UT::Q::const::AID` abgelegt, was eine klare Namensraumzuordnung für ISOBUS-Anwendungen ermöglicht.
- Die Konstanten sind im IEC 61499-Standard als `VAR_GLOBAL CONSTANT` deklariert, wodurch sie global und unveränderlich sind.
- Der Baustein ist als `GlobalConstants`-Block modelliert und besitzt daher keine Zustandslogik oder dynamische Schnittstellen.

Zustandsübersicht

Da es sich um einen reinen Konstantencontainer handelt, existiert kein Zustandsautomat. Der Baustein hat keinen internen Zustand und führt keine Zustandsübergänge aus.

Anwendungsszenarien

- **Grafische Objektverwaltung**: In ISOBUS-Terminals werden Ellipsen (z.B. zur Darstellung von Bodenbearbeitungsspuren) gezeichnet. Die Attribut-IDs aus AID_OE werden verwendet, um die Geometrie (Breite, Höhe) und Darstellung (Linienart, Füllung) dieser Objekte zu parameterisieren.
- **Konfiguration von Arbeitsgeräten**: In landwirtschaftlichen Maschinen können über diese IDs die Attribute eines Ellipsen-Objekts gesetzt oder ausgelesen werden, z.B. für die Anzeige von Feldgrenzen oder Arbeitsbreiten.
- **Entwicklung von ISOBUS-Kommunikationsmodulen**: Zusammen mit anderen Konstantenblöcken (z.B. für Linien, Punkte) bildet AID_OE die Basis für die Implementierung von Objekt-Pools (Object Pool) gemäß ISO 11783-6.

Vergleich mit ähnlichen Bausteinen

Ähnliche Konstantenblöcke existieren für andere grafische Grundobjekte, beispielsweise:

- **AID_OL** (Line): Objektattribut-IDs für Linienobjekte.
- **AID_OT** (Text): Objektattribut-IDs für Textobjekte.
- **AID_OR** (Rectangle): Objektattribut-IDs für Rechtecke.

Diese Blöcke folgen demselben Muster: Sie definieren eine Sammlung von USINT-Konstanten, die die Attribute des jeweiligen Objekttyps identifizieren. Sie unterscheiden sich lediglich in der spezifischen Bedeutung und Anzahl der Konstanten.

Fazit

Der Baustein AID_OE stellt eine wichtige, klar strukturierte Konstantensammlung für die Attribut-IDs von Ellipsen-Objekten im ISOBUS-Bereich bereit. Durch seine globale Verfügbarkeit und die festen Werte unterstützt er eine konsistente und fehlerfreie Entwicklung von Anwendungen, die grafische Objekte nach ISO 11783 handhaben müssen. Auch wenn er selbst keine aktive Logik ausführt, ist er ein unverzichtbarer Bestandteil für die modulare und wartbare Implementierung entsprechender Steuerungssysteme.

AID_OL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der globale Konstantenblock **AID_OL** definiert die Attribut-IDs (Object Attribute IDs) für das **Output Line Object** (Ausgabe-Linienobjekt) im ISOBUS-Kontext. Diese Konstanten dienen als eindeutige Referenzen für die Identifikation spezifischer Attribute eines Linienobjekts, das in der isobus-Klasse `isobus::UT::Q` verwendet wird. Durch die Bereitstellung symbolischer Namen wird der Code lesbarer und weniger fehleranfällig, da numerische IDs nicht mehr direkt verwendet werden müssen.

Schnittstellenstruktur

Da **AID_OL** ein globaler Konstantenblock und kein Funktionsblock, Adapter oder Subapp ist, besitzt er keine klassischen Schnittstellenelemente wie Ereignis- oder Datenports. Die bereitgestellten Konstanten sind als **globale Variablen** verfügbar und können vom gesamten Projekt referenziert werden.

Ereignis-Eingänge

Keine (nicht vorhanden – reine Konstantendefinition).

Ereignis-Ausgänge

Keine (nicht vorhanden – reine Konstantendefinition).

Daten-Eingänge

Keine (nicht vorhanden – reine Konstantendefinition).

Daten-Ausgänge

Keine (nicht vorhanden – reine Konstantendefinition).

Adapter

Keine (nicht vorhanden – reine Konstantendefinition).

Funktionsweise

Der Konstantenblock stellt vier Konstanten vom Typ `USINT` (Unsigned Short Integer) bereit, die jeweils eine bestimmte Attribut-ID eines Output-Line-Objekts repräsentieren. Die Werte sind fest definiert und können während der Laufzeit nicht verändert werden. Sie dienen als Parameter für Funktionen oder Dienstleistungen, die auf die Attribute eines Linienobjekts zugreifen (z.B. beim Setzen oder Abfragen von Eigenschaften). Durch die symbolischen Bezeichnungen wird die Nutzung im Quellcode erleichtert und die Lesbarkeit verbessert.

Die einzelnen Konstanten sind:

Konstante	Wert (USINT)	Beschreibung
`LINE_ATT`	1	Objekt-ID eines Line-Attribut-Objekts (AID_OL_LINE_ATT)
`WIDTH`	2	Breite des Linienobjekts in Pixeln (AID_OL_WIDTH)
`HEIGHT`	3	Höhe des Linienobjekts in Pixeln (AID_OL_HEIGHT)
`LINE_DIR`	4	Linienrichtung: 0 = von links oben nach rechts unten, 1 = von links unten nach rechts oben (AID_OL_LINE_DIR)

Technische Besonderheiten

- **Datentyp:** Alle Konstanten sind als `USINT` (8-Bit unsigned integer) deklariert.
- **Wertebereich:** Die Werte sind von 1 bis 4 aufsteigend festgelegt.
- **Bedeutung:** Jede Zahl hat eine feste semantische Zuordnung im ISOBUS-Protokoll.
- **Initialwerte:** Die Werte sind als Konstanten initialisiert und können nicht überschrieben werden.
- **Verwendungszweck:** Sie werden typischerweise in der ISOBUS-Kommunikation verwendet, um Attribute eines Output-Line-Objekts zu identifizieren – beispielsweise bei der Konfiguration von virtuellen Terminals oder der Übertragung von Grafikdaten.

Zustandsübersicht

Es existieren keine dynamischen Zustände, da der Block ausschließlich statische Konstanten definiert. Der Zustand ist immer identisch: die Werte sind fest und unveränderlich.

Anwendungsszenarien

- **ISOBUS-Implementierung:** Verwendung in Anwendungen, die mit ISOBUS-kompatiblen Terminals oder Steuergeräten kommunizieren und Linienobjekte darstellen oder verwalten.
- **Objektbibliotheken:** Einbindung in Objektbibliotheken (Object Pool), um die Attribut-IDs des Output-Line-Objekts konsistent zu referenzieren.
- **Konfiguration:** Setzen von Linieneigenschaften (Breite, Höhe, Richtung) über die entsprechenden IDs.
- **Entwicklungsunterstützung:** Erhöhung der Code-Qualität durch sprechende Namen anstelle magischer Zahlen.

Vergleich mit ähnlichen Bausteinen

Im ISOBUS-Umfeld existieren zahlreiche weitere Global-Konstantenblöcke für unterschiedliche Objekttypen (z.B. `AID_OL` für Output Line, `AID_AL` für Alarm, `AID_NM` für Number, etc.). Diese folgen demselben Muster: Sie definieren Attribut-IDs als benannte Konstanten und ermöglichen so eine einheitliche und fehlerresistente Programmierung. `AID_OL` ist speziell auf die Attribute des Output-Line-Objekts zugeschnitten.

Fazit

Der globale Konstantenblock `AID_OL` stellt eine kompakte und klar strukturierte Sammlung von Attribut-IDs für ISOBUS-Ausgabe-Linienobjekte bereit. Durch die Verwendung sprechender Namen wird die Wartbarkeit und Lesbarkeit von ISOBUS-Anwendungen erheblich verbessert. Die konstante Definition verhindert versehentliche Änderungen und sorgt für eine stabile, protokollkonforme Implementierung.

AID_OLA

• – ◦ • – ◦ • – ◦ •

Einleitung

Der GlobalConstants-Baustein **AID_OLA** definiert die Attribut-IDs (Attribute Identifiers) für ein Output List Object (OLA) im ISOBUS-Protokoll. Diese Konstanten sind integraler Bestandteil der Kommunikation zwischen landwirtschaftlichen Maschinen und Geräten, wobei sie die spezifischen Attribute eines Ausgabelisten-Objekts identifizieren. Jede Konstante repräsentiert eine feste numerische ID, die im Rahmen der ISOBUS-Datenübertragung verwendet wird, um auf das jeweilige Attribut des Objekts zu referenzieren.

Schnittstellenstruktur

Da es sich bei **AID_OLA** um eine GlobalConstants-Definition handelt, besitzt der Baustein keine klassische Schnittstelle mit Ereignis- oder Datenein-/ausgängen. Stattdessen werden vier Konstanten bereitgestellt, die als eindeutige Identifikatoren für die Attribute des Output List Objects dienen. Diese Konstanten stehen global zur Verfügung und können in anderen Bausteinen direkt referenziert werden.

Ereignis-Eingänge

- Keine

Ereignis-Ausgänge

- Keine

Daten-Eingänge

- Keine (Die Konstanten sind implizit als globale Definitionen verfügbar.)

Daten-Ausgänge

- Keine (Die Konstanten sind als feste Werte in den Baustein integriert und werden nicht über eine Schnittstelle ausgegeben.)

Adapter

- Keine

Hinweis: Die vier definierten Konstanten sind:

Konstante	Wert	Beschreibung
`WIDTH`	1	Breite des Objekts in Pixeln (`AID_OLA_WIDTH`)
`HEIGHT`	2	Höhe des Objekts in Pixeln (`AID_OLA_HEIGHT`)
`VARIABLE_REF`	3	Objekt-ID einer Zahlenvariablen (`AID_OLA_VARIABLE_REF`)
`VALUE`	4	Gewählter Listenindex (0 bis 254) oder 255, wenn kein Element ausgewählt (`AID_OLA_VALUE`)

Funktionsweise

Der Baustein stellt die Attribut-IDs als Konstanten bereit, die in der ISOBUS-Kommunikation für das Output List Object verwendet werden. Ein Output List Object ist ein Element der virtuellen Endgeräte (VT), das eine Liste von Auswahlmöglichkeiten (z.B. Menüpunkte) darstellt. Die Konstanten dienen dazu, die Attribute dieses Objekts (Breite, Höhe, referenzierte Variable und aktueller Wert) eindeutig zu identifizieren, wenn sie über das ISOBUS-Protokoll angesprochen werden. Durch die Verwendung dieser Konstanten wird eine konsistente und fehlerfreie Zuordnung zwischen den physischen Objekten und den übertragenen Daten sichergestellt.

Technische Besonderheiten

- Die Konstanten sind als `USINT` (Unsigned Short Integer) deklariert, was eine kompakte und effiziente Darstellung im ISOBUS-Rahmen ermöglicht.
- Die Initialwerte entsprechen den definierten IDs gemäß ISO 11783-6 (ISOBUS – Virtual Terminal).
- Der Baustein enthält einen Copyright-Hinweis gemäß Eclipse Public License 2.0 (EPL-2.0).
- Er ist in der `packageName`-Struktur `isobus::UT::Q::const::AID` organisiert, was eine klare Modulstruktur fördert.
- Die Werte sind als Konstanten im globalen Kontext definiert und können ohne weitere Instanziierung in anderen Bausteinen verwendet werden.

Zustandsübersicht

Da **AID_OLA** ausschließlich Konstanten definiert und keine ausführbare Logik enthält, existiert kein Zustandsmodell. Der Baustein hat keine transitions- oder zustandsbasierten Abläufe.

Anwendungsszenarien

- **Erstellung eines VT (Virtual Terminal) – Objekts:** Bei der Definition eines neuen Output List Object in einem ISOBUS-Terminal können die Konstanten verwendet werden, um Attribute wie Breite und Höhe festzulegen.
- **Referenzierung von Variablen:** Mittels `VARIABLE_REF` kann eine Verknüpfung zu einer vorhandenen Zahlenvariablen hergestellt werden, deren Wert dann als aktueller Listenindex (`VALUE`) dargestellt wird.
- **Programmierung von Bedienelementen:** In Steuerungsanwendungen, die auf ISOBUS-Terminals zugreifen, können diese Konstanten helfen, Eingaben (z.B. Auswahl eines Menüpunkts) korrekt zu interpretieren.
- **Debugging und Tests:** Durch die klar definierten IDs wird die Fehleranalyse bei Kommunikationsproblemen mit dem VT vereinfacht.

Vergleich mit ähnlichen Bausteinen

In der ISOBUS-Umgebung gibt es für jedes Objekt (z.B. Number Variable, Output String, List Object) eigene GlobalConstants-Bausteine, die die jeweiligen Attribut-IDs definieren. Beispiele sind `AID_NVS` (Number Variable Selection) oder `AID_OS` (Output String). Diese Bausteine folgen demselben Muster: Sie stellen numerische Konstanten für die Attribute des jeweiligen Objekttyps bereit. Der Aufbau ist stets

ähnlich, unterscheidet sich jedoch in der Anzahl und Bedeutung der Konstanten. **AID_OLA** ist speziell auf die Anforderungen eines Output List Objects zugeschnitten.

Fazit

AID_OLA ist ein bedeutender Baustein für die ISOBUS-Kommunikation, der eine saubere und standardisierte Definition der Attribut-IDs für Output List Objects bereitstellt. Durch die klare Struktur und die einfache Integration in andere Bausteine erleichtert er die Entwicklung von Anwendungen für landwirtschaftliche Terminals und trägt zur Interoperabilität zwischen verschiedenen Herstellern bei. Seine Verwendung reduziert potenzielle Fehlerquellen und erhöht die Wartbarkeit des Gesamtsystems.

AID_OLBG

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AID_OLBG** (Attribute IDs for Output Linear Bar Graph) ist eine globale Konstantendefinition für den ISOBUS (ISO 11783) Standard. Er dient der Bereitstellung numerischer Kennungen (Attribute IDs), die benötigt werden, um auf die Attribute eines Objekts vom Typ *Output Linear Bar Graph* (lineares Balkendiagramm) zuzugreifen. Diese Konstanten werden typischerweise in Funktionsbausteinen verwendet, die ISOBUS-Objekte über das Netzwerk konfigurieren oder auslesen.

Schnittstellenstruktur

Da es sich um eine reine Konstantendefinition handelt, besitzt dieser Baustein keine Ereignis- oder Dateneingänge/ausgänge sowie keine Adapter. Stattdessen werden die folgenden globalen Konstanten definiert:

Konstante	Wert (USINT)	Bedeutung
`WIDTH`	1	Breite des Balkendiagramms in Pixeln.
`HEIGHT`	2	Höhe des Balkendiagramms in Pixeln.
`COLOUR`	3	Farbindex für das Diagramm.
`TARGET_LINE_COLOUR`	4	Farbe der Ziellinie (Target Line).
`OPTIONS`	5	Bitmaske, die verschiedene Optionen festlegt (z. B. Rand zeichnen, Ziellinie, Ticks, Typ, Ausrichtung, Richtung).
`NUMB_TICKS`	6	Anzahl der anzuzeigenden Ticks (Skalenteilungen).
`MIN_VALUE`	7	Minimalwert des darzustellenden Bereichs.
`MAX_VALUE`	8	Maximalwert des darzustellenden Bereichs.
`VARIABLE_REF`	9	Objekt-ID einer Variablenreferenz, die den aktuellen Wert liefert.
`TARGET_VAL_VAR_REF`	10	Objekt-ID einer Variablenreferenz für den Zielwert.
`TARGET_VALUE`	11	Der gewünschte Zielwert (statisch).
`VALUE`	12	Der aktuelle Wert des Balkendiagramms.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Keine.

Funktionsweise

Die hier definierten Konstanten sind numerische Kennungen (Attribute IDs), die im ISOBUS-Protokoll verwendet werden, um auf die Eigenschaften eines linearen Balkendiagramm-Objekts zuzugreifen. Jede Konstante entspricht einem bestimmten Attribut, das entweder gesetzt oder ausgelesen werden kann. Durch die Verwendung benannter Konstanten wird der Code lesbarer und weniger fehleranfällig, da magische Zahlen vermieden werden.

Die Konstanten sind als `USINT` (Unsigned Short Integer) mit den Werten 1 bis 12 deklariert und liegen im Namensraum `isobus::UT::Q::const::AID`. Sie sind Teil einer Gruppe von Global Constants, die in 4diac-IDE Projekten für die Entwicklung von ISOBUS-Anwendungen verwendet werden.

Technische Besonderheiten

- Alle Konstanten sind als `USINT` (8-Bit unsigned) definiert und werden mit einem initialen Wert deklariert.
- Sie sind als `GLOBAL CONSTANT` gekennzeichnet, d. h. sie sind während der gesamten Laufzeit unveränderlich.
- Die Konstanten sind in einem speziellen Paket (`isobus::UT::Q::const::AID`) organisiert, um eine klare Zuordnung zum ISOBUS-Standard zu ermöglichen.
- Die Werte 1–12 entsprechen den Attribut-IDs des Objekts *Output Linear Bar Graph* nach ISO 11783-6.

Zustandsübersicht

Dieser Baustein besitzt keine Zustandsmaschine und keinen internen Zustand. Es handelt sich um eine reine Deklarationsdatei, die zur Compilezeit ausgewertet wird.

Anwendungsszenarien

- **ISOBUS-Terminal-Bedienoberflächen**: Konfiguration eines linearen Balkendiagramms zur Anzeige von Maschinenparametern (z. B. Füllständen, Temperaturen).
- **Eingebettete Steuerungen**: Kommunikation mit ISOBUS-fähigen Steuergeräten, um Attribute wie Breite, Farbe, Min/Max-Werte zu setzen.
- **Entwicklung mit 4diac IDE**: Einbinden der Konstanten in IEC 61499-Funktionsbausteine, die ISOBUS-Objekte manipulieren.

Vergleich mit ähnlichen Bausteinen

Es gibt ähnliche globale Konstantendefinitionen für andere ISOBUS-Objekttypen, z. B. `AID_OB` (Output Bar Graph), `AID_OT` (Output Text) usw. Diese unterscheiden sich in den Namen und der Anzahl der Attribute. `AID_OLBG` ist speziell auf lineare Balkendiagramme zugeschnitten und enthält Attribute wie `TARGET_LINE_COLOUR` und `TARGET_VALUE`, die bei anderen Balkendiagrammtypen möglicherweise nicht existieren.

Fazit

Die globale Konstantendefinition `AID_OLBG` stellt eine saubere und standardisierte Möglichkeit dar, die Attribut-IDs eines linearen Balkendiagramms im ISOBUS-Umfeld zu referenzieren. Sie erhöht die Wartbarkeit und Lesbarkeit von 4diac-Projekten und unterstützt Entwickler bei der Implementierung von ISOBUS-Anwendungen. Durch die klar definierten Konstanten können Fehler durch falsche Zahlenwerte vermieden werden.

AID_OLRL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `AID_OLRL` (Object Label Reference List Object Attribute IDs) ist eine globale Konstantendeklaration aus dem Paket `isobus::UT::Q::const::AID`. Er definiert eine Konstante, die die Anzahl der beschrifteten Objekte in einer ISO-bus-basierten Objektreferenzliste angibt. Diese Konstante wird im Rahmen des ISOBUS-Protokolls verwendet, um die Struktur von Nachrichten zur Verwaltung von Objektattributen zu beschreiben.

Schnittstellenstruktur

`AID_OLRL` besitzt keine Ein- oder Ausgänge im klassischen Sinne eines Funktionsblocks, da es sich um eine globale Konstantendefinition handelt. Die Struktur beschränkt sich auf die deklarierte Konstante.

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

Keine vorhanden.

Funktionsweise

Die globale Konstante `NUMB_LABELLED_OBJ` vom Typ `USINT` (Unsigned Short Integer) wird mit dem Wert `1` initialisiert. Sie gibt die Anzahl der beschrifteten Objekte an, die in einer Objektreferenzliste folgen. Im ISOBUS-Kontext wird diese Konstante verwendet, um die Länge oder den Aufbau einer Datenstruktur zu kennzeichnen, die Referenzen auf Objekte mit Attributen (z. B. Beschriftungen) enthält.

Die Deklaration erfolgt als `VAR_GLOBAL CONSTANT` und ist damit innerhalb des gesamten Projekts unveränderlich verfügbar. Auf sie kann über den vollständigen Pfad `isobus::UT::Q::const::AID::AID_OLRL::NUMB_LABELLED_OBJ` zugegriffen werden.

Technische Besonderheiten

- **Typisierung:** Die Konstante ist explizit als `USINT` typisiert, was eine sichere Verwendung in arithmetischen Operationen gewährleistet.
- **Initialisierung:** Der vorgegebene Wert `1` entspricht der üblichen Anzahl von Beschriftungsobjekten in einer Standard-Objektreferenzliste.
- **Konstantheit:** Durch die Deklaration als `CONSTANT` ist der Wert zur Laufzeit unveränderlich, was die Robustheit des Systems erhöht.
- **Paketzuordnung:** Die Einbindung in das Paket `isobus::UT::Q::const::AID` stellt sicher, dass die Konstante im ISOBUS-Namensraum eindeutig identifizierbar ist.

Zustandsübersicht

Da es sich nicht um einen funktionalen Baustein handelt, existiert kein Zustandsdiagramm. Die Konstante hat stets den gleichen Wert und benötigt keine Zustandslogik.

Anwendungsszenarien

- **ISOBUS-Nachrichtenaufbau:** In der Kommunikation zwischen Traktor und Anbaugerät kann die Konstante verwendet werden, um die Anzahl der Objektreferenzen in einer Nachricht zu definieren.
- **Datenstrukturdefinition:** Sie dient als Grundlage für die Erstellung von Datenstrukturen, die eine variable Anzahl von Objektbeschriftungen enthalten.
- **Konfiguration:** Entwickler können sie nutzen, um die maximale oder erwartete Anzahl von beschrifteten Objekten in einem System festzulegen.

Vergleich mit ähnlichen Bausteinen

Im ISOBUS-Standard existieren weitere Konstantenbausteine, die ähnliche Attribute ID-Listen definieren (z. B. `AID_OL`, `AID_OLRL` usw.). Diese unterscheiden sich hauptsächlich in der Art der referenzierten Objekte und der spezifischen Bedeutung ihrer Konstanten. `AID_OLRL` fokussiert auf die Objekt-Referenzliste, während andere Bausteine andere Aspekte (z. B. einzelne Objekte oder deren Attribute) abdecken.

Fazit

`AID_OLRL` ist ein einfacher, aber wichtiger Konstantenbaustein für die ISOBUS-Kommunikation. Er stellt eine standardisierte Zahl bereit, die die Struktur von Objektlisten beschreibt. Durch seine klare Typisie-

rung und Konstantheit trägt er zur Wiederverwendbarkeit und Sicherheit in 4diac-Projekten bei. Für Anwender, die ISOBUS-basierten Systeme entwickeln, stellt er eine nützliche Ressource dar.

AID_OM

• – ◦ • – ◦ • – ◦ •

Einleitung

Die Globalkonstanten **AID_OM** definieren die Attribut-IDs (Attribute IDs) für das Objekt „Output Meter“ im Rahmen des ISOBUS-Protokolls (ISO 11783). Sie dienen als einheitliche Referenzwerte für die Adressierung von Eigenschaften eines Anzeige-Instruments (z. B. Rundinstrument) in der Benutzerschnittstelle eines Terminals. Die Werte sind als USINT (Unsigned Short Integer) im Bereich 1…12 festgelegt und werden typischerweise beim Zugriff auf Objektattribute über den ISOBUS-Kommunikationsstack verwendet.

Schnittstellenstruktur

AID_OM ist keine typische Funktionsblock- oder Adapter-Instanz, sondern eine Sammlung globaler Konstanten. Daher sind keine ereignis- oder datenbasierten Ein-/Ausgänge vorhanden.

Ereignis-Eingänge

Nicht vorhanden.

Ereignis-Ausgänge

Nicht vorhanden.

Daten-Eingänge

Nicht vorhanden.

Daten-Ausgänge

Nicht vorhanden.

Adapter

Nicht vorhanden.

Funktionsweise

Die Konstantenwerte repräsentieren die Attributkennungen (AIDs) des „Output Meter“-Objekts. Sie werden in der Regel als **const**-Werte in den Quellcode eingebunden, um die Lesbarkeit zu verbessern und die Gefahr von Tippfehlern bei der Verwendung numerischer Literale zu vermeiden. Jede Konstante steht für ein spezifisches Attribut:

Konstantenname	Wert	Beschreibung
WIDTH	1	Breite des Instruments in Pixeln
NEEDLE_COLOUR	2	Farbindex für die Nadel
BORDER_COLOUR	3	Farbindex für den Rahmen
ARC_THICK_COLOUR	4	Farbe/Stärke des Kreisbogens
OPTIONS	5	Bitmaske für Optionen (z. B. Bogen, Rahmen, Skalenteilung, Drehrichtung)
NUMB_TICKS	6	Anzahl der Skalenteilstriche
START_ANGLE	7	Startwinkel des Skalenbogens
END_ANGLE	8	Endwinkel des Skalenbogens
MIN_VALUE	9	Minimaler darzustellender Wert
MAX_VALUE	10	Maximaler darzustellender Wert
VARIABLE_REF	11	Objekt-ID einer Variablen, die den aktuellen Wert liefert
VALUE	12	Aktueller Wert des Instruments

Die Konstanten werden als **globale Konstanten** in einem 4diac-Projekt deklariert und können in FB-Netzwerken oder ST-Code referenziert werden.

Technische Besonderheiten

- **Typ**: `USINT` (8-Bit-Unsigned Integer)
- **Initialwerte**: Die Werte sind direkt als USINT-Literale vorgegeben (z. B. `USINT#1`).
- **Verwendung im ISOBUS**: Die AIDs sind Teil des ISO-11783-Teil 6 (Virtual Terminal) und werden zum Lesen/Schreiben von Objekteigenschaften verwendet.
- **Erweiterbarkeit**: Die Konstanten sind in einem eigenen Paket (`isobus::UT::Q::const::AID`) gekapselt, wodurch sie modular in verschiedene Module eingebunden werden können.
- **Lizenz**: Der Kommentar verweist auf die Eclipse Public License 2.0.

Zustandsübersicht

Da es sich um reine Konstanten handelt, existieren keine Zustände. Die Werte sind während der gesamten Laufzeit unveränderlich (konstant).

Anwendungsszenarien

- **Steuerung von Anzeigeinstrumenten** in landwirtschaftlichen Maschinen: Ein FB, der ein „Output Meter“ bedient, kann die Attribut-IDs aus `AID_OM` verwenden, um z. B. den aktuellen Wert (`AID_OM.VALUE`) zu setzen oder die Skalierung (`AID_OM.MIN_VALUE`, `AID_OM.MAX_VALUE`) zu konfigurieren.
- **Entwicklung von ISOBUS-Terminals**: Beim Implementieren der Virtual-Terminal-Schnittstelle (VT) erleichtern diese Konstanten die Zuordnung von Attributen zu den vom Standard geforderten Objekten.
- **Test- und Simulationsumgebungen**: Durch die zentrale Definition können Testfälle die korrekten AIDs unabhängig von konkreten Werten referenzieren.

Vergleich mit ähnlichen Bausteinen

In der ISOBUS-Welt existieren für jedes Objekt eigene Satelliten-Konstantensammlungen, z. B. `AID_SO` (Softkey-Objekt) oder `AID_WS` (Working Set). Diese folgen dem gleichen Muster: Jede Sammlung definiert numerische Konstanten für die Attribute des jeweiligen Objekttyps. `AID_OM` ist speziell für das Output-Meter-Objekt ausgelegt und unterscheidet sich durch die spezifischen Eigenschaften (Winkel, Ticks, Nadel etc.) von anderen Objekttypen.

Fazit

Die Globalkonstanten `AID_OM` stellen eine saubere und typsichere Möglichkeit dar, die Attribut-IDs für ISOBUS-Output-Meter-Objekte zu definieren. Sie erhöhen die Wartbarkeit und Lesbarkeit des Codes, indem sie numerische Literale durch sprechende Namen ersetzen. Obwohl sie keinen direkten Funktionsblock darstellen, sind sie ein essenzieller Bestandteil für die Entwicklung von 4diac-Anwendungen im Bereich der landwirtschaftlichen Automatisierung.

AID_ON

• − ◦ • − ◦ • − ◦ •

Einleitung

Der Baustein `AID_ON` ist ein **GlobalConstants**-Baustein, der die numerischen Kennungen (Attribute IDs) für die Attribute eines **Output Number Objects** im ISOBUS definiert. Diese Konstanten werden verwendet, um auf einzelne Attribute eines solchen Objekts im Rahmen der ISO 11783-6 bzw. ISO 11783-10 (ISOBUS) zuzugreifen. Der Baustein stellt eine zentrale, wiederverwendbare Sammlung von Konstanten bereit, die in verschiedenen Funktionsbausteinen und Applikationen genutzt werden können, um die Lesbarkeit und Wartbarkeit des Codes zu verbessern.

Schnittstellenstruktur

Da es sich um einen GlobalConstants-Baustein handelt, besitzt er **keine Ein-/Ausgänge** im herkömmlichen Sinne. Die Schnittstellenstruktur besteht ausschließlich aus den deklarierten Konstanten, die als globale Konstanten über `VAR_GLOBAL CONSTANT` definiert sind.

Ereignis-Eingänge

Keine – der Baustein besitzt keine Ereignis-Eingänge.

Ereignis-Ausgänge

Keine – der Baustein besitzt keine Ereignis-Ausgänge.

Daten-Eingänge

Keine – der Baustein besitzt keine Daten-Eingänge.

Daten-Ausgänge

Keine – der Baustein besitzt keine Daten-Ausgänge, da die Werte als globale Konstanten direkt im Code verfügbar sind.

Adapter

Keine – der Baustein besitzt keine Adapter-Schnittstellen.

Funktionsweise

Der Baustein definiert zwölf Konstanten vom Typ `USINT` (Unsigned Short Integer), die jeweils eine eindeutige Attribut-ID für ein **Output Number Object** im ISOBUS repräsentieren. Die Zuordnung ist wie folgt:

Konstante	Wert	Beschreibung
`WIDTH`	1	Breite in Pixeln
`HEIGHT`	2	Höhe in Pixeln
`BACKGROUND_COLOUR`	3	Index der Hintergrundfarbe
`FONT_ATT`	4	Objekt-ID eines Font-Attribut-Objekts
`OPTIONS`	5	Options-Bitmaske
`VARIABLE_REF`	6	Objekt-ID einer Number Variable (Zahlenvariable)
`OFFSET`	7	Offset zur Anzeige des Wertes
`SCALE`	8	Skalierungsfaktor zur Anzeige des Wertes
`NUMB_DECIMALS`	9	Anzahl Dezimalstellen
`FORMAT`	10	Format: 0 = Festkomma (####.nn), 1 = Exponentialformat
`JUSTIFICATION`	11	Ausrichtung: Bits 0-1 horizontal (0=links, 1=mittig, 2=rechts), Bits 2-3 vertikal (0=oben, 1=mittig, 2=unten)
`VALUE`	12	Aktueller Wert vor Skalierung

Diese Konstanten werden typischerweise bei der Kommunikation mit ISOBUS-Terminals verwendet, z.B. beim Erstellen oder Modifizieren von Objekten über die VT-Protokollschnittstelle. Durch die Verwendung dieser symbolischen Namen wird der Code lesbarer und weniger fehleranfällig, da anstelle von magischen Zahlen sprechende Konstanten verwendet werden.

Technische Besonderheiten

- **Paketzugehörigkeit:** Die Konstanten sind im Paket `isobus::UT::Q::const::AID` organisiert, wodurch eine saubere Namensraumtrennung innerhalb des ISOBUS-Moduls erreicht wird.
- **Typsicherheit:** Alle Werte sind als `USINT` deklariert, was eine Kompatibilität mit den im ISOBUS definierten Attribut-IDs gewährleistet (1-255).
- **Konstantheit:** Die Deklaration erfolgt als `VAR_GLOBAL CONSTANT`, sodass die Werte zur Laufzeit nicht verändert werden können.
- **Kommentierung:** Jede Konstante ist mit einem erklärenden Kommentar versehen, der die Bedeutung laut ISOBUS-Spezifikation beschreibt.
- **Standardkonformität:** Der Baustein basiert auf dem IEEE 61499-1 Standard und ist im 4diac-IDE-Format als XML-Datei definiert.

Zustandsübersicht

Da es sich um einen reinen Konstanten-Baustein handelt, existiert **kein Zustandsautomat** und keine dynamischen Zustände. Der Baustein ist statisch und hat keine Laufzeitlogik. Ein Einsatz im Rahmen von ereignisgesteuerten Abläufen ist nicht vorgesehen.

Anwendungsszenarien

- **ISOBUS-VT-Applikationen:** Bei der Erstellung von Bedienoberflächen für Traktoren oder Landmaschinen werden Output Number Objects benötigt, um numerische Werte (z.B. Motordrehzahl, Geschwindigkeit) anzuzeigen. Die Konstanten erlauben das eindeutige Referenzieren der zu setzenden Attribute.
- **Konfiguration von Terminalanzeigen:** Beim Senden von Kommandos zum Setzen von Attributen (z.B. `SetAttribute`) können die Konstanten verwendet werden, um die gewünschte Eigenschaft zu identifizieren.
- **Entwicklungsunterstützung:** Die Verwendung sprechender Konstanten reduziert Tippfehler und erhöht die Wartbarkeit von ISOBUS-Implementierungen.

Vergleich mit ähnlichen Bausteinen

Innerhalb des Pakets `isobus::UT::Q::const` existieren vermutlich ähnliche GlobalConstants-Bausteine für andere Objekttypen wie z.B.:

- `AID_STRING` für Output String Objects
- `AID_LIST` für Output List Objects
- `AID_ANALOG` für Analoganzeigen

Diese unterscheiden sich in der Anzahl und Semantik der Konstanten, folgen jedoch dem gleichen Muster. `AID_ON` ist speziell auf das Output Number Object zugeschnitten und stellt dessen charakteristische Attribute bereit.

Fazit

Der `AID_ON`-Baustein ist eine nützliche, statische Konstantensammlung für ISOBUS-Entwickler. Er zentralisiert die für Output Number Objects relevanten Attribut-IDs und verbessert so die Lesbarkeit und Robustheit von Anwendungen. Die klare Strukturierung durch Paket, Kommentare und die Verwendung des IEEE 61499-1-Formats machen ihn zu einem guten Beispiel für einen wiederverwendbaren globalen Konstantenbaustein in 4diac.

AID_OP_POLY

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `AID_OP_POLY` ist eine globale Konstantendefinition, die die Attribut-IDs für ein Ausgabe-Polygonobjekt im ISOBUS-Datenmodell festlegt. Er wird verwendet, um auf standardisierte Attribute wie Breite, Höhe, Linienattribute, Füllattribute und Polygontyp zuzugreifen. Diese Konstanten erleichtern die eindeutige Identifizierung und den Zugriff auf die entsprechenden Attribute in Kommunikationsprotokollen und Anwendungen der Agrartechnik.

Schnittstellenstruktur

Da es sich um eine reine Konstantendefinition handelt, besitzt `AID_OP_POLY` keine Ein- oder Ausgangsvariablen im Sinne eines Funktionsblocks. Die Struktur besteht ausschließlich aus globalen Konstanten.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Keine.

Funktionsweise

Der Baustein definiert fünf globale Konstanten vom Typ `USINT`, die jeweils eine eindeutige Attribut-Identifikationsnummer repräsentieren:

Konstante	Wert	Beschreibung
`WIDTH`	1	Attribut `AID_OP_POLY_WIDTH` – Breite des Polygons in Pixeln.
`HEIGHT`	2	Attribut `AID_OP_POLY_HEIGHT` – Höhe des Polygons in Pixeln.
`LINE_ATT`	3	Attribut `AID_OP_POLY_LINE_ATT` – Objekt-ID eines Linienattribut-Objekts.
`FILL_ATT`	4	Attribut `AID_OP_POLY_FILL_ATT` – Objekt-ID eines Füllattribut-Objekts.
`POLYGON_TYPE`	5	Attribut `AID_OP_POLY_POLYGON_TYPE` – Polygontyp (0=konvex, 1=nicht-konvex, 2=komplex, 3=offen).

Die Konstanten können an beliebiger Stelle im Programm referenziert werden, um auf die zugehörigen Attribute zuzugreifen. Sie dienen als symbolische Namen für die numerischen Positionswerte im Attributrahmen des entsprechenden ISOBUS-Objekts.

Technische Besonderheiten

- Die Konstanten sind als `GLOBALCONSTANTS` deklariert und damit innerhalb des gesamten Projekts gültig.
- Die Werte sind fest vorgegeben und können nicht zur Laufzeit verändert werden (Konstanten).
- Der Baustein ist gemäß IEC 61499-1 strukturiert und enthält Metadaten wie Version und Beschreibung (in der XML-Datei). Die Lizenzierung erfolgt unter der Eclipse Public License 2.0.
- Die Konstantenwerte entsprechen den standardisierten Attribut-IDs für Ausgabe-Polygone im ISOBUS (ISO 11783) Protokoll.

Zustandsübersicht

Nicht zutreffend – der Baustein besitzt keine Zustände oder Zustandsautomaten.

Anwendungsszenarien

- **ISOBUS-Terminals:** Verwendung bei der Darstellung von geometrischen Formen (z. B. Feldgrenzen, Fahrspuren) auf einem landwirtschaftlichen Terminal.
- **Steuerungsanwendungen:** Zugriff auf die Attribute eines Ausgabe-Polygons, um dessen Darstellung (Breite, Höhe, Linien, Füllung) dynamisch zu konfigurieren.
- **Protokollimplementierungen:** Erstellung und Interpretation von ISOBUS-Nachrichten, die Polygonattribute enthalten.

Durch die Verwendung dieser Konstanten wird der Code lesbarer und unabhängig von magischen Zahlen.

Vergleich mit ähnlichen Bausteinen

Es gibt ähnliche globale Konstantendefinitionen für andere Objekttypen, z. B. `AID_OP_LINE` oder `AID_OP_RECTANGLE`, die jeweils die Attribut-IDs für Linien- oder Rechteckobjekte bereitstellen. Der Unterschied liegt in den spezifischen Attributen: Während ein Rechteck z. B. nur Breite und Höhe hat, besitzt ein Polygon zusätzlich Linien- und Füllattribute sowie einen Polygontyp. `AID_OP_POLY` stellt die für Polygone relevanten IDs bereit.

Fazit

`AID_OP_POLY` ist eine einfache, aber essentielle Sammlung von Konstanten für die Arbeit mit ISOBUS-Ausgabe-Polygonen. Sie ermöglicht eine klare und standardisierte Referenzierung der Attribute und trägt zur Wartbarkeit und Portabilität von Anwendungen in der Agrartechnik bei. Die Verwendung symbolischer Namen reduziert Fehlerquellen und erleichtert die Einhaltung des ISOBUS-Standards.

AID_OP_PTR

• – ◦ • – ◦ • – ◦ •

Einleitung

Der globale Konstantenbaustein `AID_OP_PTR` definiert einen Konstantenwert für die Objektattribut-ID (Object Pointer Object Attribute ID) im Kontext des ISOBUS-Protokolls (ISO 11783). Er gehört zum Paket `isobus::UT::Q::const::AID` und stellt einen numerischen Identifikator bereit, der in der UT-Anwendung (Universal Terminal) verwendet wird, um auf den „aktuellen Wert“ eines Objektzeigers zu verweisen. Der Baustein ist als `GLOBALCONSTANTS` deklariert, d.h. der enthaltene Wert ist während der gesamten Laufzeit unveränderlich.

Schnittstellenstruktur

Da es sich um einen globalen Konstantenbaustein handelt, besitzt er **keine** Ein-/Ausgangs-Schnittstellen im Sinne eines Funktionsblocks. Es sind weder Ereignis- noch Datenein-/ausgänge oder Adapter vorhanden.

Ereignis-Eingänge

Nicht vorhanden.

Ereignis-Ausgänge

Nicht vorhanden.

Daten-Eingänge

Nicht vorhanden.

Daten-Ausgänge

Nicht vorhanden.

Adapter

Nicht vorhanden.

Funktionsweise

Der Baustein `AID_OP_PTR` stellt eine einzige globale Konstante bereit:

- **`VALUE`** (Typ `USINT`, initial `1`)
  Diese Konstante wird in der ISOBUS-Kommunikation als Attribut-ID für den „aktuellen Wert“ eines Objektzeigers verwendet. Sie ist über den qualifizierten Namen `AID_OP_PTR.VALUE` ansprechbar und kann in Algorithmen oder FB-Instanzen gelesen werden, um den korrekten Attributcode zu referenzieren.

Die Konstante wird zur Compile-Zeit festgelegt und bietet eine zentrale, einheitliche Referenz für alle Module, die mit Objektzeigern arbeiten.

Technische Besonderheiten

- **Unveränderlichkeit:** Der Wert ist als `CONSTANT` deklariert und kann nicht zur Laufzeit modifiziert werden.
- **Typ:** `USINT` (Unsigned Short Integer, 8 Bit) – Wertebereich 0..255.
- **Initialwert:** `1` gemäß der Kommentierung „1: AID_OP_PTR_VALUE - Current value“.
- **Paketzuordnung:** Der Baustein ist im Paket `isobus::UT::Q::const::AID` definiert, was eine klare Namensraumstruktur für ISOBUS-Attributkonstanten schafft.
- **Copyright/Lizenz:** Die Definition unterliegt der Eclipse Public License 2.0 (SPDX-License-Identifier: EPL-2.0), wie in der XML-Datei vermerkt.

Zustandsübersicht

Da es sich um eine reine Konstantendefinition handelt, existiert kein internes Zustandsmodell. Der Baustein besitzt keinen Zustand und führt keine Verarbeitungslogik aus.

Anwendungsszenarien

- In der Implementierung einer ISOBUS-UT-Anwendung, um beim Aufbau von Objektzeigern den Attributwert für „aktueller Wert“ anzugeben.
- Als Ersatz für magische Zahlen im Quellcode, wodurch Wartbarkeit und Lesbarkeit verbessert werden.
- In Verbindung mit anderen Konstantenbausteinen aus demselben Paket (`AID_*`) zur einheitlichen Verwaltung aller Attribut-IDs.

Vergleich mit ähnlichen Bausteinen

Es gibt im selben Paket vermutlich weitere Konstantenbausteine für andere Objektattribut-IDs (z.B. `AID_OP_PTR` vs. `AID_FCT_CTRL` etc.). Diese unterscheiden sich lediglich im konkreten Wert und in der semantischen Bedeutung. Der Baustein `AID_OP_PTR` ist spezifisch für den Objektzeiger und definiert den Wert 1. Ähnliche Bausteine folgen demselben Muster, sind aber je nach Attributtyp unterschiedlich benannt und belegt.

Fazit

`AID_OP_PTR` ist ein einfacher, aber wichtiger globaler Konstantenbaustein für die ISOBUS-Kommunikation. Er stellt einen standardisierten, unveränderlichen Wert für die Objektattribut-ID „aktueller Wert“ bereit und trägt so zur Robustheit und Klarheit von UT-Anwendungen bei. Die strikte Trennung von Konfiguration und Logik wird durch diese Art der Deklaration gefördert.

AID_OR

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AID_OR** ist ein globaler Konstantencontainer, der die Attribut-IDs für ein **Output Rectangle**-Objekt im ISOBUS-Objektmodell definiert. Diese IDs werden verwendet, um auf die verschiedenen Eigenschaften eines rechteckigen Objekts (z. B. Linienattribute, Breite, Höhe) zuzugreifen. Der Baustein dient als zentrale Referenz für die Verwendung dieser Konstanten in anderen Funktionsbausteinen oder Adaptern.

Schnittstellenstruktur

Da es sich um einen **GlobalConstants**-Container handelt, besitzt er keine herkömmlichen ereignis- oder datenbasierten Schnittstellen. Stattdessen werden die definierten Konstanten global bereitgestellt und können innerhalb des Projekts direkt verwendet werden.

Ereignis-Eingänge

Nicht vorhanden.

Ereignis-Ausgänge

Nicht vorhanden.

Daten-Eingänge

Nicht vorhanden.

Daten-Ausgänge

Nicht vorhanden.

Adapter

Nicht vorhanden.

Globale Konstanten

Konstantenname	Datentyp	Initialwert	Beschreibung
LINE_ATT	USINT	USINT#1	Objekt-ID eines Linienattribut-Objekts (AID_OR_LINE_ATT).
WIDTH	USINT	USINT#2	Breite in Pixeln (AID_OR_WIDTH).
HEIGHT	USINT	USINT#3	Höhe in Pixeln (AID_OR_HEIGHT).
LINE_SUPPR	USINT	USINT#4	Linienunterdrückung (AID_OR_LINE_SUPPR).
FILL_ATT	USINT	USINT#5	Objekt-ID eines Füllattribut-Objekts (AID_OR_FILL_ATT).

Funktionsweise

Der Baustein `AID_OR` stellt fünf konstante USINT-Werte bereit, die den Attribut-IDs eines **Output Rectangle**-Objekts im ISOBUS entsprechen. Diese IDs werden typischerweise in der ISOBUS-Kommunikation verwendet, um auf einzelne Eigenschaften des Rechtecks zuzugreifen, z. B. um die Linienart, Breite, Höhe oder Füllung zu referenzieren. Die Konstanten sind als **Global Constants** deklariert, sodass sie in allen Funktionsbausteinen des Projekts ohne weitere Definition verwendet werden können.

Technische Besonderheiten

- Die Werte sind als **USINT** (Unsigned Short Integer) definiert und liegen im Bereich von 1 bis 5.
- Der Baustein folgt dem **ISOBUS-Standard 61499-1** und ist in der **Eclipse Public License 2.0** lizenziert.
- Die Konstanten sind unveränderlich (`CONSTANT`) und stehen daher während der Laufzeit als feste Werte zur Verfügung.
- Die Verwendung als globaler Container ermöglicht eine zentrale und einheitliche Referenzierung von Attribut-IDs im gesamten ISOBUS-basierten Steuerungscode.

Zustandsübersicht

Der Baustein besitzt keine internen Zustände, da er ausschließlich Konstanten definiert. Es gibt weder Zustandsübergänge noch eine Ablaufsteuerung.

Anwendungsszenarien

- **ISOBUS-Objektbeschreibung**: Verwendung bei der Definition und Parametrierung von Output-Rectangle-Objekten, z. B. für die Anzeige von Rechtecken auf einem Terminal.
- **Attributreferenzierung**: Zugriff auf die Breite, Höhe oder Linienattribute eines Rechtecks über die jeweiligen Konstanten.
- **Kommunikation mit ISOBUS-Bedienterminals**: Der Baustein kann in Steuerungsfunktionen eingebunden werden, die ISOBUS-Nachrichten mit Attribut-IDs für Rechtecke erzeugen oder auswerten.

Vergleich mit ähnlichen Bausteinen

Es gibt ähnliche GlobalConstants-Container für andere Objektarten im ISOBUS, z. B. für Linienobjekte (`AID_LINE`) oder Textobjekte (`AID_TXT`). Im Gegensatz zu diesen definiert `AID_OR` spezifisch die Attribute eines Output-Rechtecks. Die grundlegende Struktur (mehrere USINT-Konstanten) ist vergleichbar, unterscheidet sich jedoch in den konkreten Attribut-IDs und deren Bedeutung.

Fazit

`AID_OR` ist ein einfacher, aber wichtiger Konstantenbaustein für die ISOBUS-Entwicklung. Er ermöglicht eine definierte und konsistente Verwendung der Attribut-IDs für Rechteck-Objekte. Durch die Bereitstellung als globale Konstanten wird die Wartbarkeit des Codes verbessert und die Gefahr von Tippfehlern oder falschen Zahlenwerten reduziert. Der Baustein erfüllt damit eine unterstützende Funktion in ISOBUS-basierten Steuerungssystemen.

AID_OS

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `AID_OS` stellt eine Sammlung globaler Konstanten zur Verfügung, die als Attribut-IDs (AID) für **Output String Objekte** im ISOBUS-Kontext (ISO 11783) dienen. Diese Konstanten werden verwendet, um auf die verschiedenen Attribute eines Output-String-Objekts in der landwirtschaftlichen Elektronik zuzugreifen. Sie sind als `GlobalConstants` in IEC 61499 definiert und erlauben eine einheitliche, symbolische Referenzierung anstelle von magischen Zahlenwerten.

Schnittstellenstruktur

Da `AID_OS` keine klassischen Ein-/Ausgänge besitzt, stellt es ausschließlich **globale Konstanten** bereit. Diese können im gesamten Projekt verwendet werden und sind in der folgenden Tabelle aufgeführt:

Konstantenname	Typ	Wert	Bedeutung
`WIDTH`	USINT	1	Breite des Objekts in Pixeln (AID_OS_WIDTH)
`HEIGHT`	USINT	2	Höhe des Objekts in Pixeln (AID_OS_HEIGHT)

BACKGROUND_COLOUR	USINT	3	Farbindex des Hintergrunds
FONT_ATT	USINT	4	Objekt-ID eines Font-Attributes-Objekts
OPTIONS	USINT	5	Bitmaske: Bit 0 = transparent, Bit 1 = automatischer Umbruch, Bit 2 = Umbruch am Bindestrich
VARIABLE_REF	USINT	6	Objekt-ID eines Variablen-Objekts
JUSTIFICATION	USINT	7	Ausrichtung: Bits 0-1 (horizontal: 0=links, 1=mittig, 2=rechts), Bits 2-3 (vertikal: 0=oben, 1=mittig, 2=unten)

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine direkten Ausgänge – die Konstanten sind global verfügbar und können über den qualifizierten Namen (z.B. `AID_OS.WIDTH`) angesprochen werden.

Adapter

Keine vorhanden.

Funktionsweise

Der Baustein definiert sieben Konstanten, die die numerischen IDs der Attribute eines Output-String-Objekts im ISOBUS-Protokoll repräsentieren. Diese IDs werden bei der Konfiguration von Terminal-Objekten in der Landmaschinen-Elektronik verwendet, um z.B. die Darstellung und das Verhalten von Textausgaben festzulegen. Durch die Verwendung symbolischer Namen wird der Code lesbarer und unabhängig von konkreten Zahlenwerten. Die Konstanten sind als `USINT` (Unsigned Short Integer) mit festen Initialwerten definiert und im globalen Namensraum verfügbar.

Technische Besonderheiten

- **Typ**: Alle Werte sind `USINT` (8-Bit, vorzeichenlos).
- **Bitmanipulation**: Die Konstanten `OPTIONS` und `JUSTIFICATION` enthalten bitweise kodierte Informationen. Die Bedeutung der einzelnen Bits ist in den Kommentaren dokumentiert.
- **ISOBUS-Konformität**: Die Attribut-IDs entsprechen den Spezifikationen des ISOBUS-Standards (ISO 11783) für Output-String-Objekte.
- **Einsatz**: Diese Konstanten werden typischerweise zusammen mit den ISOBUS-Funktionsbausteinen (z.B. für Objektmanagement) verwendet.

Zustandsübersicht

Da es sich um reine Konstanten handelt, existiert kein Verhalten oder Zustandsautomat. Der Baustein ist statisch und besitzt keine internen Zustände.

Anwendungsszenarien

Ein typisches Anwendungsbeispiel ist die Erstellung eines Output-String-Objekts in einem ISOBUS-Terminal. Hierbei müssen die Attribute des Objekts (Breite, Höhe, Hintergrundfarbe, Schriftart usw.) über deren AID-Werte gesetzt werden. Durch die Verwendung von `AID_OS.WIDTH`, `AID_OS.HEIGHT` usw. wird der Code selbstdokumentierend und leicht an Änderungen im Standard anpassbar.

Beispiel (ST-Syntax):

```
myOutputString.Width := AID_OS.WIDTH;
myOutputString.Options := AID_OS.OPTIONS; // transparent, auto-wrap
```

Vergleich mit ähnlichen Bausteinen

Im selben Kontext existieren weitere GlobalConstants-Bausteine für andere Objekttypen, z.B. `AID_OS` für Output-Strings, `AID_OT` für Objekt-Typen oder `AID_VAR` für Variablen. Diese folgen dem gleichen Muster und stellen die jeweiligen Attribut-IDs bereit. `AID_OS` ist speziell auf Output-String-Objekte zugeschnitten.

Fazit

`AID_OS` ist ein einfacher, aber nützlicher Konstantencontainer, der die Entwicklung von ISOBUS-kompatiblen Anwendungen vereinfacht. Durch die zentrale Definition der Attribut-IDs wird die Wartbarkeit erhöht und Fehler durch falsche Zahlenwerte werden vermieden. Der Baustein ist statisch und robust und eignet sich ideal für den Einsatz in allen Projekten, die mit ISOBUS-Terminals kommunizieren.

AID_PG

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `AID_PG` ist ein **GlobalConstants**-Element der 4diac-IDE, das die Attribut-IDs (Attribute IDs) für ein **Picture Graphic Objekt** im ISOBUS-Kontext definiert. Diese Konstanten werden verwendet, um auf spezifische Attribute eines Bildgrafikelements in der Steuerungskommunikation zu referenzieren (z. B. Breite, Höhe, Format). Der Baustein stellt keine typischen Ein-/Ausgänge eines Funktionsblocks bereit, sondern dient als zentrale Konstante, die global in der Applikation verfügbar ist.

Schnittstellenstruktur

Da es sich um einen **GlobalConstants**-Baustein handelt, besitzt er **keine** klassischen Ein-/Ausgangsschnittstellen. Die definierten Werte sind als globale Konstanten fest verdrahtet und können an beliebigen Stellen im Projekt verwendet werden.

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

Keine vorhanden.

Funktionsweise

Der Baustein definiert sechs globale Konstanten, die jeweils eine bestimmte Attribut-ID für ein Picture Graphic Objekt repräsentieren. Diese Konstanten werden im Compiler-Paket `isobus::UT::Q::const::AID` gebündelt und können an jedem Punkt eines 4diac-Projekts referenziert werden. Die Zuordnung der ID-Werte ist fest in der ISOBUS-Spezifikation verankert.

Konstante	Wert (USINT)	Bedeutung
`WIDTH`	`USINT#1`	AID_PG_WIDTH – Breite des Bildes in Pixeln.
`OPTIONS`	`USINT#2`	AID_PG_OPTIONS – Bitmaske: Bit 0 = transparent, Bit 1 = blinkend, Bit 2 = RLE-codiert.
`TRANSPARENCY_COLOUR`	`USINT#3`	AID_PG_TRANSPARENCY_COLOUR – Transparenzfarbe (falls transparent).
`ACTUAL_WIDTH`	`USINT#4`	AID_PG_ACTUAL_WIDTH – Tatsächliche Breite des Bildes.

ACTUAL_HEIGHT	USINT#5	AID_PG_ACTUAL_HEIGHT – Tatsächliche Höhe des Bildes.
FORMAT	USINT#6	AID_PG_FORMAT – Bildtyp: 0 = Monochrom, 1 = 4-Bit-Farbe, 2 = 8-Bit-Farbe.

Diese Konstanten werden typischerweise verwendet, um beim Zugriff auf ein Picture Graphic Objekt über den ISOBUS-Datenaustausch die entsprechenden Attribut-IDs eindeutig zu identifizieren.

Technische Besonderheiten

- Der Baustein ist als **GlobalConstants**-Element implementiert, nicht als Funktionsblock. Er definiert keine ausführbare Logik, sondern nur eine feste Sammlung von Konstanten.
- Die Werte sind als `USINT` (Unsigned Short Integer) deklariert und mit konkreten Initialwerten versehen.
- Die Konstanten sind im Paket `isobus::UT::Q::const::AID` gekapselt, was eine klare Namensraum-Trennung innerhalb der ISOBUS-Bibliothek ermöglicht.
- Das Element enthält Metadaten wie Versionsinformationen und einen Original-Quellcode-Abschnitt (`OriginalSource`), der die Definition in der 4diac-IDE dokumentiert.

Zustandsübersicht

Nicht anwendbar, da es sich um einen Konstanten-Baustein handelt und keine Zustände oder Zustandsübergänge existieren.

Anwendungsszenarien

Typische Einsatzgebiete sind:

- **ISOBUS-Terminals und -Steuerungen**, die Bildgrafiken (Picture Graphics) für die Darstellung von Symbolen oder Icon-Informationen benötigen.
- **Datenzugriff** auf Attribute eines Bildobjekts, z. B. um die Breite oder das Format eines Bildes zu ermitteln oder zu setzen.
- **Erweiterung von HMI-Anwendungen** im landwirtschaftlichen Maschinenbereich, die ISOBUS-konforme Darstellungen verwenden (z. B. gemäß ISO 11783).

Vergleich mit ähnlichen Bausteinen

In der ISOBUS-Bibliothek existieren weitere `GlobalConstants`-Elemente, die ähnliche Attribut-IDs für andere Objekttypen definieren (z. B. `AID_OBJ` für generische Objekte, `AID_BUTTON` für Tasten, `AID_LABEL` für Beschriftungen). Diese unterscheiden sich hauptsächlich durch die spezifischen Attribut-IDs und deren Bedeutung. Der vorliegende Baustein `AID_PG` ist speziell auf **Picture Graphs** ausgelegt und stellt nur die für diesen Objekttyp relevanten Konstanten bereit.

Fazit

`AID_PG` ist ein einfacher, aber wichtiger Konstanten-Baustein für die ISOBUS-Kommunikation. Er bündelt die definierten Attribut-IDs für Bildgrafiken in einer zentralen, wiederverwendbaren Form und erleichtert so die Entwicklung von 4diac-Applikationen, die mit ISOBUS-Bildobjekten interagieren. Trotz des Fehlens von Schnittstellen ist er ein unverzichtbarer Bestandteil für die korrekte Parametrierung von Bildgrafiken im landwirtschaftlichen Automatisierungsumfeld.

AID_SG

• – ◦ • – ◦ • – ◦ •

Einleitung

AID_SG ist eine Sammlung von globalen Konstanten, die die Attribut-IDs für skalierte Grafikobjekte im ISOBUS-Protokoll definieren. Diese Konstanten werden verwendet, um auf spezifische Eigenschaften wie Breite, Höhe, Skalierungsart, Optionen und Wert zuzugreifen. Die Definitionen sind als `GLOBALCONSTANTS`-Block in 4diac implementiert und dienen als Referenzwerte für die Kommunikation mit ISOBUS-fähigen Geräten.

Schnittstellenstruktur

Da es sich bei AID_SG um eine Konstantendefinition und nicht um einen Funktionsblock handelt, existieren keine Ein- oder Ausgänge im herkömmlichen Sinne. Die Struktur beschränkt sich auf die Definition von globalen Konstanten.

Ereignis-Eingänge
Nicht vorhanden.

Ereignis-Ausgänge
Nicht vorhanden.

Daten-Eingänge
Nicht vorhanden.

Daten-Ausgänge
Nicht vorhanden.

Adapter
Nicht vorhanden.

Funktionsweise
Die Konstanten in AID_SG bilden eine eindeutige Kennung (Attribut-ID) für verschiedene Attribute eines skalierten Grafikobjekts. In ISOBUS-Nachrichten werden diese IDs als Parameterwerte verwendet, um dem Empfänger mitzuteilen, welches Attribut (z. B. Breite oder Skalierungsart) gesetzt oder abgefragt werden soll. Durch die Verwendung von globalen Konstanten wird eine einheitliche und fehlerfreie Referenzierung im gesamten Projekt sichergestellt.

Technische Besonderheiten
Die einzelnen Konstanten sind wie folgt definiert:

Konstante	Wert	Bedeutung
`WIDTH`	1	AID_SG_WIDTH – Breite in Pixeln
`HEIGHT`	2	AID_SG_HEIGHT – Höhe in Pixeln
`SCALE_TYPE`	3	AID_SG_SCALE_TYPE – Bits 0-2: Skalierungswert (0 = keine Skalierung, 1 = auf Breite skalieren, 2 = auf Höhe skalieren, 3 = auf Breite und Höhe, 4 = auf passende Größe). Bits 3-4: Horizontale Ausrichtung (0 = links, 1 = mittig, 2 = rechts). Bits 5-6: Vertikale Ausrichtung (0 = oben, 1 = mittig, 2 = unten)
`OPTIONS`	4	AID_SG_OPTIONS – Bitmaske: Bit 0 = Blinken
`VALUE`	5	AID_SG_VALUE – Aktueller Wert

Diese Konstanten sind vom Typ `USINT` (Unsigned Short Integer) und tragen feste Werte, die der ISOBUS-Spezifikation entsprechen.

Zustandsübersicht
Nicht anwendbar, da AID_SG keine Zustandslogik besitzt.

Anwendungsszenarien
AID_SG wird in ISOBUS-basierten Steuerungssystemen eingesetzt, um skalierte Grafikobjekte auf Bedienterminals darzustellen. Typische Anwendungen umfassen:

- Setzen der Breite oder Höhe eines grafischen Objekts über die entsprechenden Attribut-IDs.
- Konfiguration der Skalierungsart und Ausrichtung mithilfe des `SCALE_TYPE`.
- Steuerung von Blinkeffekten über das `OPTIONS`-Flag.
- Übertragung des aktuellen Anzeigewerts über `VALUE`.

Vergleich mit ähnlichen Bausteinen
Im ISOBUS-Umfeld existieren vergleichbare Konstantendefinitionen für andere Objekttypen, z. B. für Textobjekte (`AID_Text`) oder Linien (`AID_Line`). Diese folgen dem gleichen Muster: Sie definieren numerische IDs für die jeweiligen Attribute. Im Unterschied zu einem Funktionsblock bieten sie jedoch keine Logik, sondern stellen lediglich eine Referenztabelle für die Kommunikation bereit.

Fazit

AID_SG stellt eine wichtige Grundlage für die Implementierung ISOBUS-konformer Geräte dar, indem es eine klare und standardisierte Referenz für Attribut-IDs skalierten Grafikobjekte bereitstellt. Durch die Verwendung als globale Konstanten wird die Lesbarkeit und Wartbarkeit des Codes erhöht und die Einhaltung des ISOBUS-Protokolls erleichtert.

AID_SK

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `AID_SK` (Attribute IDs für Soft Keys) definiert globale Konstanten zur Identifikation von Attributen eines Soft-Key-Objekts im ISOBUS-Virtual-Terminal (VT). Diese Konstanten werden verwendet, um auf Attribute wie Hintergrundfarbe oder Tastencode zu referenzieren. Der Baustein ist Teil des Pakets `isobus::UT::Q::const::AID` und folgt der IEC 61499-1 Norm.

Schnittstellenstruktur

Der Baustein besitzt keine Ereignis- oder Daten-Ein-/Ausgänge, da es sich um eine reine Konstantendefinition handelt. Es werden keine Adapter bereitgestellt.

Ereignis-Eingänge

Nicht vorhanden.

Ereignis-Ausgänge

Nicht vorhanden.

Daten-Eingänge

Nicht vorhanden.

Daten-Ausgänge

Nicht vorhanden.

Adapter

Nicht vorhanden.

Funktionsweise

Der Baustein stellt zwei globale Konstanten vom Typ `USINT` (Unsigned Short Integer) bereit:

- `BACKGROUND_COLOUR` mit dem Wert `1` – entspricht dem Attribut `AID_SK_BACKGROUND_COLOUR` (Hintergrundfarbindex).
- `KEY_CODE` mit dem Wert `2` – entspricht dem Attribut `AID_SK_KEY_CODE` (Tastencode, den das VT in der Soft-Key-Aktivierungsnachricht meldet).

Diese Konstanten können in anderen Bausteinen oder Programmen verwendet werden, um auf die entsprechenden Attribut-IDs zuzugreifen, ohne magische Zahlen im Code zu verwenden.

Technische Besonderheiten

- Die Konstanten sind als `VAR_GLOBAL CONSTANT` deklariert und daher zur Laufzeit unveränderlich.
- Die Werte sind als `USINT`-Datentyp definiert und initialisiert.
- Das Original stammt aus einer ISOBUS-Implementierung und ist durch die Eclipse Public License 2.0 geschützt.
- Die Konstanten sind in einem Compiler-Paket (`isobus::UT::Q::const::AID`) organisiert, was die Wiederverwendung in verschiedenen Modulen ermöglicht.

Zustandsübersicht

Da es sich um reine Konstanten handelt, besitzt der Baustein keinen Zustandsautomaten. Die Werte sind statisch und ändern sich während der Laufzeit nicht.

Anwendungsszenarien

- **Soft-Key-Objekt konfigurieren**: Bei der Erstellung eines Soft-Key-Objekts im VT können die Attribut-IDs verwendet werden, um die Hintergrundfarbe oder den Tastencode zu setzen.

- **Soft-Key-Aktivierung auswerten**: Beim Empfang einer Soft-Key-Aktivierungsnachricht wird über die Konstante `KEY_CODE` der zugehörige Tastencode identifiziert.
- **LESEN/SETZEN von Objektattributen**: In Kommunikationsprotokollen, die auf ISOBUS basieren, können diese IDs als Parameter für Objekt-Attribut-Operationen genutzt werden.

Vergleich mit ähnlichen Bausteinen

Es gibt weitere globale Konstanten-Bausteine für verschiedene Objekttypen (z.B. `AID_OBJ` für allgemeine Objekte). `AID_SK` fokussiert spezifisch auf Soft-Key-Objekte. Im Gegensatz zu Funktionsblöcken, die Verhalten kapseln, bieten diese Konstanten nur eine Wertequelle ohne Logik.

Fazit

Der Baustein `AID_SK` stellt eine saubere und standardisierte Möglichkeit dar, die Attribut-IDs für Soft-Key-Objekte im ISOBUS-Kontext zu verwenden. Durch die zentrale Definition werden Fehler vermieden und die Wartung erleichtert. Er ist ein typisches Beispiel für Konstanten-Bausteine in IEC 61499-Systemen.

AID_SKM

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AID_SKM** ist ein globaler Konstanten-Container, der Attribut-Identifikatoren für Softkey-Masken-Objekte (Soft Key Mask Object Attribute IDs) im ISOBUS-Kontext definiert. Er wird im Paket `isobus::UT::Q::const::AID` bereitgestellt und dient als zentrale Referenz für einheitliche Werte, die in Steuerungsanwendungen zur Darstellung von Benutzeroberflächen verwendet werden.

Schnittstellenstruktur

Da es sich um eine reine Konstantendeklaration handelt, besitzt der Baustein keine ein- oder ausgehenden Ereignisse, keine Daten-Ein-/Ausgänge und keine Adapter.

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

Keine vorhanden.

Funktionsweise

Der Baustein definiert eine global konstante Variable `BACKGROUND_COLOUR` vom Typ `USINT` mit dem Initialwert 1. Diese Konstante repräsentiert den Farbindex für den Hintergrund einer Softkey-Maske. Durch die Bereitstellung als globale Konstante kann dieser Wert an verschiedenen Stellen im Projekt referenziert werden, ohne dass er mehrfach deklariert werden muss. Dadurch wird die Wartbarkeit und Konsistenz in der Softwareentwicklung erhöht.

Technische Besonderheiten

- **Typ**: `GlobalConstants` (kein Funktionsblock im klassischen Sinne).
- **Initialwert**: Die Konstante ist mit dem Wert `USINT#1` vordefiniert.
- **Kommentar**: Der Kommentar `1: AID_SKM_BACKGROUND_COLOUR - Background colour index.` erläutert die Bedeutung des Wertes.
- **Herkunft**: Der Baustein gehört zum Paket `isobus::UT::Q::const::AID` und ist Teil einer ISOBUS-spezifischen Bibliothek.

Zustandsübersicht

Nicht anwendbar – es existieren keine Zustände, da es sich um reine Konstanten handelt.

Anwendungsszenarien

- **ISOBUS-Steuerterminals**: Verwendung in HMI-Anwendungen, um einheitliche Farbdefinitionen für Softkey-Masken zu referenzieren.
- **Projektentwicklung**: Als zentrale Konstante, um den Farbindex für Hintergründe an zentraler Stelle zu definieren und spätere Änderungen zentral vornehmen zu können.
- **Code-Generierung**: Der Baustein wird typischerweise in Verbindung mit von der 4diac-IDE generiertem Code verwendet, um Konstanten in die Zielsprache (z. B. C++) zu übernehmen.

Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu herkömmlichen Funktionsbausteinen, die Daten verarbeiten und Ereignisse auslösen, bietet `AID_SKM` ausschließlich Konstanten an. Vergleichbare Bausteine könnten andere globale Konstanten-Container sein (z. B. `AID_OBJ`, `AID_PROP`), die jeweils unterschiedliche Attribut-IDs für verschiedene Objekttypen bereitstellen. Der Vorteil eines solchen Containers liegt in der zentralen Verwaltung aller relevanten IDs, wodurch Redundanz vermieden wird.

Fazit

Der `AID_SKM`-Baustein ist ein schlankes Konstantenmodul, das eine spezifische Attribut-ID für Softkey-Masken bereitstellt. Er ist ein wesentlicher Bestandteil einer ISOBUS-Anwendung, um Farbwerte einheitlich zu definieren und die Softwarearchitektur übersichtlich zu halten. Durch seine einfache Struktur eignet er sich gut für die direkte Verwendung in 4diac-Projekten, in denen globale Konstanten benötigt werden.

AID_TYPE

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **AID_TYPE** ist ein **GlobalConstants**-Typ im Rahmen der 4diac-IDE. Er definiert eine globale Konstante, die als Attribut-Identifikator für Objekttypen im ISOBUS-Kontext (Universal Terminal, UT) dient. Die Konstante gehört zur Paketstruktur `isobus::UT::Q::const::AID` und stellt eine numerische Kennung für die Abfrage des Objekttyp-Attributs bereit.

Schnittstellenstruktur

Da es sich um einen globalen Konstanten-Baustein handelt, besitzt er keine klassischen Ein-/Ausgangsschnittstellen wie Funktionsblöcke (FB) oder Adapter. Die Konstante wird über den globalen Symbolnamen `AID_TYPE.OBJ_TYPE` referenziert und ist in jedem Baustein des Projekts sichtbar, sobald der Konstanten-Baustein eingebunden ist.

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine (die Konstante wird direkt über den Namen angesprochen, nicht als Ausgang)

Adapter

Keine

Funktionsweise

Der Baustein stellt eine einzige, unveränderliche Konstante bereit:

- `OBJ_TYPE` vom Typ `USINT` (8-Bit unsigned integer) mit dem initialen Wert `0`.

Diese Konstante repräsentiert die **AID_TYPE_OBJ_TYPE** – eine Kennung, die angibt, dass es sich um das Attribut „Objekttyp“ handelt. Im ISOBUS-Protokoll wird diese Kennung verwendet, um Abfragen oder Benachrichtigungen bezüglich des Objekttyps eines Elements (z. B. eines Arbeitsbildschirms) zu identifizieren.

Der Wert wird zur Compile-Zeit festgelegt und ist damit für alle weiteren Bausteine unveränderlich. Dadurch wird eine konsistente und fehlerfreie Referenzierung innerhalb des ISOBUS-Programms gewährleistet.

Technische Besonderheiten

- **Typ**: `USINT` (Unsigned Short Integer) – Bereich `0`–`255`.
- **Initialwert**: `0` – entspricht der ISO 11783-Definition für „Object Type identifier“.
- **Paketstruktur**: `isobus::UT::Q::const::AID` – deutet auf die Zugehörigkeit zu einer ISOBUS-Umsetzung hin.
- **Originalcode**: Der Quellcode ist in ST (Structured Text) verfasst und als globaler Konstantenblock definiert.
- **Eigenschaften**: Der Baustein ist als `VAR_GLOBAL CONSTANT` deklariert, d. h., er ist projektweit gültig und nicht änderbar.

Zustandsübersicht

Nicht zutreffend – ein GlobalConstants-Baustein besitzt keine internen Zustände oder Zustandsautomaten. Er dient ausschließlich der symbolischen Definition eines konstanten Wertes.

Anwendungsszenarien

- **ISOBUS Universal Terminal (UT)**: Erkennung und Verarbeitung von Objekttyp-Attributen in Steuerungssystemen.
- **Normgerechte Kommunikation**: Nutzung der vordefinierten AID-Kennung für das ISOBUS-Protokoll.
- **Referenzierung**: In anderen Funktionsblöcken oder Applikationen kann `AID_TYPE.OBJ_TYPE` direkt als Index oder Parameter verwendet werden.
- **Entwicklung landwirtschaftlicher Steuerungen**: Einsatz in Traktoren, Anbaugeräten oder Bedienterminals mit ISOBUS-Anbindung.

Vergleich mit ähnlichen Bausteinen

Es gibt zahlreiche GlobalConstants-Bausteine in ISOBUS-Umsetzungen, die jeweils spezifische Attribut-IDs (AIDs) für unterschiedliche Abfragen definieren. Im Gegensatz zu reinen Funktionsblöcken bieten Konstantenblöcke keine Logik, sondern nur Daten. Im Vergleich zu anderen Konstantenbausteinen, die z. B. Farbwerte oder Puffergrößen festlegen, ist dieser auf die Objekttyp-Identifikation spezialisiert und folgt strikt den ISOBUS-Vorgaben.

Fazit

Der Baustein `AID_TYPE` ist eine einfache, aber essenzielle Komponente für ISOBUS-basierte Anwendungen. Er stellt eine klar definierte Kennung für den Objekttyp bereit und trägt so zur Interoperabilität und Standardkonformität der Steuerungssoftware bei. Seine Verwendung als globale Konstante erleichtert die Wartung und Vermeidung von Magic Numbers im Code.

AID_WM

• – ◦ • – ◦ • – ◦ •

Einleitung

Die globale Konstantensammlung `AID_WM` definiert Attribut-IDs (Attribute Identifiers) für das **Window Mask**-Objekt im ISOBUS-Protokoll. Sie ist Teil des Pakets `isobus::UT::Q::const::AID` und stellt die numerischen Kennungen bereit, die verwendet werden, um Attribute eines Fenstermasken-Objekts (z. B. Hintergrundfarbe, Optionen, Name) in der ISO-Bus-Kommunikation anzusprechen. Die Konstanten sind als `USINT`-Werte festgelegt und dienen als Referenzwerte für die Parametrierung von Fenstermasken in Bedienterminals.

Schnittstellenstruktur

Da es sich um eine **Globalkonstanten**-Definition handelt, besitzt der Baustein keine Ein-/Ausgänge oder Adapter. Die Schnittstellenstruktur beschränkt sich auf die Bereitstellung der nachfolgenden Konstanten:

Ereignis-Eingänge
Nicht vorhanden.

Ereignis-Ausgänge
Nicht vorhanden.

Daten-Eingänge
Nicht vorhanden.

Daten-Ausgänge
Nicht vorhanden.

Adapter
Nicht vorhanden.
Stattdessen werden die folgenden **Konstanten** als Teil dieser Sammlung definiert:

Konstante	Typ	Wert	Beschreibung
`BACKGROUND_COLOUR`	`USINT`	`1`	Hintergrundfarbenindex eines Fenstermasken-Objekts.
`OPTIONS`	`USINT`	`2`	Bitmaske für Optionen: Bit 0 = Verfügbar (0 = deaktiviert/ blank), Bit 1 = Transparent (ohne Hintergrundfarbe).
`NAME`	`USINT`	`3`	Objekt-ID einer Ausgabestring- oder Objektzeiger-Referenz, die den Namen enthält.

Funktionsweise
Die Konstanten werden als **Attribut-IDs** verwendet, um im ISOBUS-Protokoll auf die Eigenschaften eines Fenstermasken-Objekts zuzugreifen. Jede ID entspricht einem spezifischen Attribut:

- **`BACKGROUND_COLOUR`** (ID 1): Referenziert den Farbindex, der für die Hintergrunddarstellung der Fenstermaske verwendet wird.
- **`OPTIONS`** (ID 2): Enthält eine Bitmaske, die das Verhalten der Maske steuert (z. B. Sichtbarkeit, Transparenz).
- **`NAME`** (ID 3): Verweist auf eine Objekt-Referenz, die den Namen der Fenstermaske bereitstellt.

Diese IDs werden typischerweise zusammen mit den Methoden des ISO-Bus-Protokolls eingesetzt, um Attribute eines Fenstermasken-Objekts zu setzen oder abzufragen. Durch die Verwendung als globale Konstanten wird eine einheitliche und fehlerfreie Referenzierung im gesamten Code gewährleistet.

Technische Besonderheiten
- **Datentyp**: Alle Konstanten sind als `USINT` (Unsigned Short Integer, 8 Bit) definiert.
- **Initialwerte**: Die Werte sind fest eingestellt (`1`, `2`, `3`) und können im Betrieb nicht verändert werden.
- **Paketzuordnung**: Die Konstanten gehören zum Paket `isobus::UT::Q::const::AID`, das speziell für Attribut-IDs im ISO-Bus-Bereich entwickelt wurde.
- **Lizenz**: Die Definition ist unter der Eclipse Public License 2.0 verfügbar (siehe Copyright-Hinweis im XML).

Zustandsübersicht
Da es sich um eine reine Konstantendefinition handelt, gibt es keine dynamischen Zustände. Die Werte bleiben während der gesamten Laufzeit konstant und benötigen keine Initialisierung oder Zustandsverwaltung.

Anwendungsszenarien
- **ISO-Bus-Kommunikation**: Verwendung in Steuergeräten oder Bedienterminals, die Fenstermasken-Objekte gemäß ISO 11783 (ISOBUS) verwalten.
- **Parametrierung**: Einfache Referenzierung der Attribut-IDs in Funktionsbausteinen oder Programmen, um z. B. die Hintergrundfarbe einer Maske zu ändern oder Optionen zu setzen.
- **Entwicklungsunterstützung**: Bereitstellung klarer, benannter Konstanten zur Vermeidung von „Magic Numbers“ im Quellcode.

Vergleich mit ähnlichen Bausteinen

Es gibt weitere Globalkonstanten-Sammlungen im Paket `isobus::UT::Q::const`, z. B. für andere Objekttypen (z. B. `AID_OBJ`, `AID_ALM`). Diese folgen demselben Muster: Sie definieren Attribut-IDs für spezifische ISO-Bus-Objekte. Im Vergleich zu diesen ist `AID_WM` auf die Fenstermasken-Objekte spezialisiert und enthält nur die zugehörigen drei Attribute.

Im Gegensatz zu Funktionsbausteinen (FBs) weist die Konstantensammlung keine Verarbeitungslogik oder Ein-/Ausgangsstruktur auf; sie dient ausschließlich als Definitionscontainer.

Fazit

Die Globalkonstanten-Sammlung `AID_WM` stellt eine unverzichtbare Grundlage für die Entwicklung von ISO-Bus-Anwendungen dar, die Fenstermasken verwenden. Durch die klar definierten, benannten Konstanten wird die Lesbarkeit und Wartbarkeit des Codes verbessert und die Einhaltung des ISO-11783-Standards unterstützt. Die einfache Struktur macht sie leicht zu integrieren und zu erweitern, falls weitere Attribut-IDs für Fenstermasken benötigt werden.

AID_WS

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `AID_WS` stellt eine Sammlung von globalen Konstanten dar, die die Attribut-IDs (Attribute IDs) für ein Working Set im ISOBUS-Kontext (ISO 11783) definieren. Diese Konstanten werden verwendet, um auf spezifische Attribute eines Working Sets zuzugreifen, wie z.B. Hintergrundfarbe, Auswählbarkeit und die aktive Maske. Der Baustein ist als `GlobalConstants`-Typ in 4diac umgesetzt und dient als zentrale Referenz für diese Attribut-IDs.

Schnittstellenstruktur

Da es sich um einen GlobalConstants-Baustein handelt, besitzt dieser weder Ereignis- noch Dateneinoder -ausgänge. Die nachfolgenden Abschnitte sind daher leer. Alle Informationen sind als Konstanten innerhalb des Bausteins definiert.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Keine.

Funktionsweise

Der Baustein definiert drei globale Konstanten vom Typ `USINT` (Unsigned Short Integer), die jeweils eine spezifische Attribut-ID für ein Working Set repräsentieren:

- `BACKGROUND_COLOUR` (Wert 1): Diese Konstante identifiziert das Attribut „Background colour index“ eines Working Sets. Der Wert entspricht der Attribut-ID `AID_WS_BACKGROUND_COLOUR`.
- `SELECTABLE` (Wert 2): Diese Konstante kennzeichnet das Attribut „Selectable“, das angibt, ob ein Working Set vom Bediener ausgewählt werden kann (0 = FALSE, 1 = TRUE). Die zugehörige Attribut-ID ist `AID_WS_SELECTABLE`.
- `ACTIVE_MASK` (Wert 3): Diese Konstante verweist auf das Attribut „Active Mask“, das die Objekt-ID der Daten- oder Alarmmaske angibt, die angezeigt werden soll, solange das Working Set aktiv ist. Die Attribut-ID lautet `AID_WS_ACTIVE_MASK`.

Diese Konstanten werden typischerweise in ISOBUS-Kommunikationsmodulen verwendet, um Attribute eines Working Sets über den entsprechenden Dienst (z.B. Get/Set Attribute) zu adressieren. Sie sind als `VAR_GLOBAL CONSTANT` deklariert und stehen daher im gesamten Projekt zur Verfügung.

Technische Besonderheiten

- **Typ**: `USINT` – 8-Bit vorzeichenloser Integer, passend zu ISOBUS-Datenformaten.
- **Initialwerte**: Die Werte 1, 2 und 3 entsprechen den offiziellen Attribut-IDs gemäß ISO 11783-6.
- **Paketname**: Der Baustein ist in das Paket `isobus::UT::Q::const::AID` eingebettet, was auf eine strukturierte Bibliothek für UT (Universal Terminal) Konstanten hinweist.
- **Lizenz**: Die enthaltenen Informationen und die Deklaration unterliegen der Eclipse Public License 2.0.

Zustandsübersicht

Als reiner Konstanten-Baustein besitzt `AID_WS` keinen internen Zustand oder Zustandsautomaten. Die Werte sind statisch und unveränderlich.

Anwendungsszenarien

Typische Anwendungsfälle sind:

- Implementierung von ISOBOBUS-UT-Funktionen, bei denen auf Working-Set-Attribute zugegriffen wird.
- Erstellung von Dienstfunktionen zur Abfrage oder Änderung von Working-Set-Eigenschaften (z.B. Hintergrundfarbe ändern, Auswählbarkeit setzen).
- Einbindung in größere 4diac-Anwendungen zur Steuerung von Terminals in landwirtschaftlichen Maschinen.

Vergleich mit ähnlichen Bausteinen

In der ISOBUS-Welt existieren ähnliche Konstanten-Bausteine für andere Objekttypen, z.B. `AID_OBJECT_POOL`, `AID_MASK`, `AID_SOFTKEY` etc. Diese folgen demselben Muster und definieren die jeweiligen Attribut-IDs. `AID_WS` ist spezifisch auf Working Sets zugeschnitten und enthält nur die drei oben genannten Konstanten. Andere Bausteine könnten weitere Attribute enthalten, abhängig vom Objekttyp.

Fazit

`AID_WS` ist ein einfacher, aber essenzieller GlobalConstants-Baustein für die ISOBUS-Kommunikation. Er bietet eine klare und zentrale Definition der Attribut-IDs für Working Sets und erleichtert die Wartung und Lesbarkeit von 4diac-Anwendungen im landwirtschaftlichen Umfeld. Durch die Verwendung als globale Konstanten wird eine hohe Konsistenz und Fehlervermeidung bei der Adressierung von Attributen gewährleistet.

AID_WSSC

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `AID_WSSC` ist eine Sammlung globaler Konstanten, die Attribut-Identifikationsnummern (Attribute IDs) für das Objekt „Working Set Special Controls“ im ISOBUS-Kontext definiert. Diese Konstanten werden verwendet, um auf bestimmte Attribute eines Working Set Special Controls-Objekts (WSSC) zu referenzieren, beispielsweise für Farbkarten oder Farbpaletten. Die Konstanten sind als `USINT`-Werte (8-Bit ohne Vorzeichen) definiert und in einem GlobalConstants-Baustein gebündelt, sodass sie projektweit genutzt werden können.

Schnittstellenstruktur

Da es sich um einen GlobalConstants-Baustein handelt, besitzt dieser keine Ereignis-, Daten- oder Adapter-Schnittstellen im klassischen Sinne eines Funktionsblocks. Die bereitgestellten Konstanten dienen als statische Werte und werden nicht über Ein-/Ausgänge ausgetauscht. Nachfolgend sind die definierten Konstanten aufgelistet:

Ereignis-Eingänge

Nicht vorhanden.

Ereignis-Ausgänge

Nicht vorhanden.

Daten-Eingänge

Nicht vorhanden.

Daten-Ausgänge

Nicht vorhanden.

Adapter

Nicht vorhanden.

Bereitgestellte Konstanten:

Konstante	Typ	Wert	Kommentar
`NUMOFBYTES`	USINT	1	Anzahl der folgenden Bytes im Objekt (AID_WSSC_NUMOFBYTES).
`COLOUR_MAP`	USINT	2	Objekt-ID einer Farbkarte (Colour Map) oder `NULL` (0).
`COLOUR_PALETTE`	USINT	3	Objekt-ID einer Farbpalette (Colour Palette) oder `NULL` (0).

Funktionsweise

Der Baustein stellt die Konstantenwerte für die Attribut-IDs bereit, die im ISOBUS-Protokoll (ISO 11783) für das Objekt „Working Set Special Controls“ verwendet werden. Diese IDs werden typischerweise bei der Erstellung oder Übertragung von Nachrichten referenziert, um auf die entsprechenden Attribute eines Geräts zuzugreifen. Durch die Definition als globale Konstanten wird deren konsistente Verwendung im gesamten Projekt sichergestellt und Fehler durch hartkodierte Werte vermieden.

Technische Besonderheiten

- **ISOBUS-Standard**: Die Konstanten entsprechen den Attribut-IDs, die im ISO 11783-Teil 6 (Virtual Terminal) für WSSC-Objekte spezifiziert sind.
- **Datentyp**: Alle Werte sind vom Typ `USINT` (Unsigned Short Integer) mit einem Wertebereich von 0–255.
- **Initialwerte**: Die Konstanten sind mit festen Werten initialisiert und können nicht zur Laufzeit verändert werden.
- **Lizenz**: Der Baustein ist unter der Eclipse Public License 2.0 (EPL-2.0) veröffentlicht, was die Nutzung in Open-Source- und proprietären Projekten ermöglicht.
- **Paketzuordnung**: Der Baustein befindet sich im Paket `isobus::UT::Q::const::AID`, was eine klare strukturelle Einordnung in einem ISOBUS-Projekt ermöglicht.

Zustandsübersicht

Der Baustein besitzt keinerlei Zustände, da es sich um eine rein deklarative Ansammlung von Konstanten handelt. Es existieren keine internen Variablen oder Ablaufsteuerungen, die einen Zustandsübergang erfordern würden.

Anwendungsszenarien

- **ISOBUS-Implementierung**: Verwendung in der Kommunikation zwischen Traktor und Anbaugerät, insbesondere bei der Übertragung von Arbeitsbildschirmen oder Maschinensteuerungen, die Working Set Special Controls verwenden.
- **Entwicklung von Virtual-Terminal-Applikationen**: Referenzierung der konstanten Werte beim Aufbau von Nachrichten, die Attribute von WSSC-Objekten ansprechen.
- **Konfiguration von Farbdarstellungen**: Nutzung von `COLOUR_MAP` und `COLOUR_PALETTE`, um Farbkarten oder Farbpaletten einem WSSC zuordnen zu können.

Vergleich mit ähnlichen Bausteinen

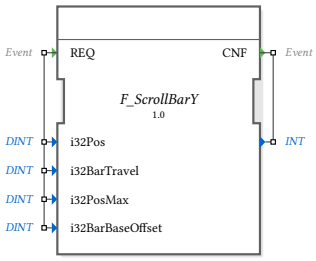
Im ISOBUS-Kontext gibt es weitere GlobalConstants-Bausteine, die Attribut-IDs für andere Objekttypen definieren, z. B. für „Object Pool“ oder „Working Set“. Im Vergleich zu diesen Bausteinen fokussiert `AID_WSSC` ausschließlich auf die Attribute, die für Working Set Special Controls relevant sind. Die Struktur ist identisch: Eine Auflistung von `USINT`-Konstanten mit Initialwerten, die die jeweiligen Attributed-IDs repräsentieren. Der Unterschied liegt lediglich in der semantischen Bedeutung der Werte für das jeweilige Objekt.

Fazit

`AID_WSSC` ist ein zweckmäßiger GlobalConstants-Baustein, der die Attribut-IDs für Working Set Special Controls im ISOBUS bereitstellt. Er trägt zur Codeklarheit und Wartbarkeit bei, indem er die Verwendung fester Zahlenwerte an zentraler Stelle bündelt. Durch seine einfache Struktur – nur drei Konstanten – ist er leicht zu integrieren und in Projekten einzusetzen, die eine ISOBUS-Kommunikation auf Basis des Virtual-Terminal-Protokolls implementieren.

helpers

F_ScrollBarY



• – ◦ • – ◦ • – ◦ •

Einleitung

Die Funktion `F_ScrollBarY` berechnet aus der aktuellen Scroll-Position (in Zeilen) die Y-Verschiebung für den Scrollbalken-Inhalts-Container (`Container_Scrollbar_Content.Top`), so dass der Balken-„Thumb“ die Listenposition proportional widerspiegelt. Sie ist das Scrollbalken-Gegenstück zu F_ScrollListY und wird intern vom Scroll-Baustein `ScrollFS` (Paket `isobus::UT::Q`) verwendet.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Berechnungsanforderung, liest `i32Pos`, `i32BarTravel`, `i32PosMax` und `i32BarBaseOffset` ein.

Ereignis-Ausgänge

- `CNF`: Bestätigt die Berechnung und liefert den Rückgabewert (namenlose `OutputVar`, angesprochen über `<Instanzname>.` mit abschließendem Punkt).

Daten-Eingänge

- `i32Pos` (DINT): Aktuelle Scroll-Position in Zeilen (0…`i32PosMax`).
- `i32BarTravel` (DINT): Verfügbarer Verschiebeweg des Thumbs (Sichtfensterhöhe – Thumbhöhe).
- `i32PosMax` (DINT): Maximale Scroll-Position (letzte gültige Zeile).
- `i32BarBaseOffset` (DINT): `Top`-Wert von `Container_Scrollbar_Content` bei Position 0 — muss betragsmäßig genau der festen internen `Top`-Position des Thumbs entsprechen, mit umgekehrtem Vorzeichen, damit der Thumb bei `i32Pos = 0` netto auf Y = 0 steht.

Daten-Ausgänge

- *(namenlos)* (INT): `Container_Scrollbar_Content.Top`-Wert (`s16Yposition` für `Q_ChildPosition`).

Adapter

Keine Adapter vorhanden.

Funktionsweise

Die Berechnung ist eine einzige Formel:
`OUT := DINT_TO_INT(i32BarBaseOffset + (i32Pos * i32BarTravel) / i32PosMax)`
Bei `i32Pos = 0` ergibt sich `OUT = i32BarBaseOffset` (Thumb am Listenanfang), bei `i32Pos = i32PosMax` ergibt sich `OUT = i32BarBaseOffset + i32BarTravel` (Thumb am Listenende). Dazwischen wird linear interpoliert.

Technische Besonderheiten

- **`DINT_TO_INT`-Cast**: wie bei `F_ScrollListY` — die Rechnung läuft in `DINT`, das Ergebnis wird erst am Ende nach `INT` konvertiert (`Q_ChildPosition.s16Yposition`).
- **Ganzzahlige Division**: `(i32Pos * i32BarTravel) / i32PosMax` rundet ab (Integer-Division) — bei kleinen `i32BarTravel`-Werten relativ zu `i32PosMax` kann der Thumb dadurch mehrere Positionsschritte lang stehen bleiben, bevor er sich sichtbar bewegt. In der Praxis unauffällig, da `i32BarTravel` üblicherweise deutlich größer als `i32PosMax` ist.
- **Konkret typisiert, kein `ANY`**: wie `F_ScrollListY` — feste `DINT`/`INT`-Typisierung, kein `MOVE`-Anker beim Verdrahten nötig.

Zustandsübersicht

Zustandslos — jeder `REQ` berechnet das Ergebnis unmittelbar aus den aktuellen Eingangswerten neu.

Anwendungsszenarien

- Ausschließlich intern in `ScrollFS` verwendet, um bei jeder Positionsänderung `Container_Scrollbar_Content.Top` neu zu berechnen — parallel zu `F_ScrollListY` für die eigentliche Liste, mit demselben `i32Pos`-Wert als gemeinsame Quelle.

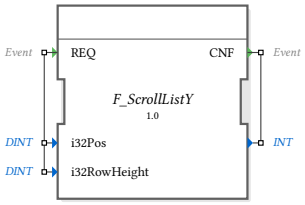
⚖️ Vergleich mit ähnlichen Bausteinen

- **Gegenüber `F_ScrollListY`**: gleiche Grundidee (Zeilenposition ⟶ Pixel-Offset), aber mit Skalierung auf die (kleinere) Scrollbalken-Spanne statt direkter Multiplikation mit der Zeilenhöhe, plus einem Basis-Offset. Siehe F_ScrollListY.

Fazit

Das Scrollbalken-Pendant zu `F_ScrollListY` — dieselbe Zeilen-Position treibt über zwei getrennte, konkret typisierte Formeln sowohl die Liste als auch den Scrollbalken an, jeweils mit ihrer eigenen Skalierung.

F_ScrollListY



• – ◦ • – ◦ • – ◦ •

Einleitung

Die Funktion `F_ScrollListY` berechnet aus der aktuellen Scroll-Position (in Zeilen) und der Zeilenhöhe (in Pixeln) die Y-Verschiebung, die `Container_Scrolling_Content.Top` bekommen muss, damit die richtige Zeile im Sichtfenster erscheint. Sie ist eine reine Rechenhilfsfunktion für den Scroll-Baustein `ScrollFS` (Paket `isobus::UT::Q`) und wird dort intern verwendet.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Berechnungsanforderung, liest `i32Pos` und `i32RowHeight` ein.

Ereignis-Ausgänge

- `CNF`: Bestätigt die Berechnung und liefert den Rückgabewert (namenlose `OutputVar`, angesprochen über `<Instanzname>.` mit abschließendem Punkt, IEC-61131-3-Konvention für den Rückgabewert einer Funktion).

Daten-Eingänge

- `i32Pos` (DINT): Aktuelle Scroll-Position in Zeilen (0…`i32PosMax`).
- `i32RowHeight` (DINT): Zeilenhöhe in Pixeln.

Daten-Ausgänge

- *(namenlos)* (INT): `Container_Scrolling_Content.Top`-Wert (`s16Yposition` für `Q_ChildPosition`).

Adapter

Keine Adapter vorhanden.

Funktionsweise

Die Berechnung ist eine einzige Formel:
`OUT := DINT_TO_INT(-(i32Pos * i32RowHeight))`
Je größer `i32Pos`, desto weiter negativ wird `Top` — der Inhalts-Container schiebt sich nach oben, spätere Zeilen rutschen ins Sichtfenster. Bei `i32Pos = 0` ist `Top = 0` (Listenanfang, keine Verschiebung).

Technische Besonderheiten

- **`DINT_TO_INT`-Cast**: Die interne Rechnung läuft in `DINT` (32 Bit), das Ergebnis wird erst am Ende nach `INT` (16 Bit) konvertiert, weil `Q_ChildPosition.s16Yposition` ein `INT` erwartet (ISO 11783-6 F.16, „Change Child Location“).
- **Konkret typisiert, kein `ANY`**: Im Gegensatz zu generischen Bausteinen wie `F_MOVE` oder `F_SEL` sind alle Ein-/Ausgänge fest typisiert (`DINT`/`INT`) — dadurch ist beim Verdrahten in `ScrollFS` kein `MOVE`-Anker nötig, obwohl der Wert direkt aus anderen generischen/Struct-Feldern stammt.

Zustandsübersicht

Zustandslos — jeder `REQ` berechnet das Ergebnis unmittelbar aus den aktuellen Eingangswerten neu.

Anwendungsszenarien

- Ausschließlich intern in `ScrollFS` verwendet, um bei jeder Positionsänderung `Container_Scrolling_Content.Top` neu zu berechnen.

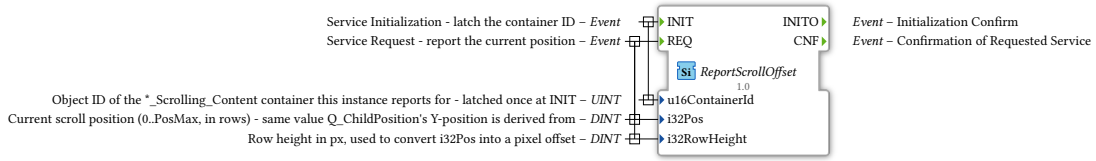
⚖️ Vergleich mit ähnlichen Bausteinen

- **Gegenüber `F_ScrollBarY`**: Beide berechnen eine Y-Verschiebung aus der Scroll-Position, `F_ScrollBarY` zusätzlich mit Skalierung auf die (kleinere) Scrollbalken-Spanne und einem Offset — siehe F_ScrollBarY.

Fazit

Eine kleine, robuste Rechenfunktion — verantwortlich für exakt eine Umrechnung (Zeilenposition → Pixel-Offset der Liste), konkret typisiert und ohne Nebenwirkungen.

ReportScrollOffset



• – ○ • – ○ • – ○ •

Einleitung

`ReportScrollOffset` ist ein Hilfs-Service-Interface-Baustein aus dem Paket `isobus::UT::Q::helpers`. Er sitzt innerhalb der ScrollFS-FBNetwork-Struktur (parallel zu `MoveList` und `ListY`) und meldet die aktuelle Scroll-Position (Zeilen-Index `i32Pos` × Zeilenhöhe `i32RowHeight` in Pixeln) für den Container `u16ContainerId` an das ECU-Sichtbarkeits-Gate (`VtMaskVisibility`).

Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierung des Service-Interfaces (`With u16ContainerId`).
- `REQ`: Übermittelt eine Positionsänderung (`With i32Pos`, `With i32RowHeight`).

Ereignis-Ausgänge

- `INITO`: Bestätigt die Initialisierung.
- `CNF`: Bestätigt die Positionsmeldung.

Daten-Eingänge

- `u16ContainerId` (UINT): Objekt-ID des `*_Scrolling_Content`-Containers.
- `i32Pos` (DINT): Aktuelle Zeilen-Position (0…i32PosMax).
- `i32RowHeight` (DINT): Zeilenhöhe in Pixeln zur Umrechnung in den Pixel-Offset.

Daten-Ausgänge

Keine Daten-Ausgänge.

Adapter

Keine Adapter vorhanden.

Funktionsweise

1. Bei `INIT` sperrt der Baustein die Container-ID `u16ContainerId`.
2. Bei jedem `REQ`-Event berechnet der Baustein `PixelOffset := i32Pos * i32RowHeight` und leitet diesen an das Sichtbarkeits-Gate der ECU weiter (`VtMaskVisibility_OnScroll`).
3. Das ECU-Gate nutzt diese Information, um VT-Aktualisierungen (z. B. `Q_*`-Kommandos) auf derzeit sichtbare Zeilen zu begrenzen.

Siehe auch

- ScrollFS
- ScrollFS_PHYS_Softkey
- ScrollFS_PHYS_Button

🔌 io

📺 Video

- Automation der Zukunft
- Die große Migration
- Digital decisions
- Ideale Dioden
- The Dual Wi Fi Solution

🎧 Podcast

- 4diac IDE: Wie der IEC 61499 Standard die Industrieautomatisierung revolutioniert
- 4diac-Präsentation: Zielgruppen, Struktur und Alleinstellungsmerkmal Schärfen
- Den Software-Drachen zähmen: Industrielle Automatisierung und die Zukunft der Produktion
- Eclipse 4diac: Innovationsmotor Forschung vs. Nutzerbedürfnisse – Was treibt die Entwicklung wirklich voran?
- Software-Drachen zähmen: Warum IEC 61499 die Industrie revolutionieren könnte

---

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 Diode & Halbleitergrundlagen auf ms-muc-docs.de
- 🌐 Smart Switch vs. Relais auf ms-muc-docs.de

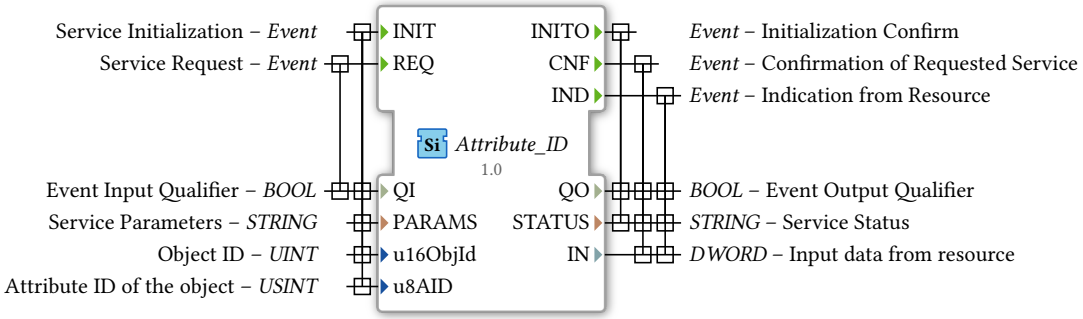
Attribute

Attribute_ID

- – ◦ • – ◦ • – ◦ •

Einleitung

`Attribute_ID` ist ein Eingabe-Service-Interface-Funktionsblock für Doppelwort-Eingabedaten (`DWORD`). Er liest ein Attribut eines VT-Objekts (Objekt-ID `u16ObjId`, Attribut-ID `u8AID`) und stellt dessen Wert als `DWORD` am Ausgang `IN` bereit.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Service-Initialisierung
  - Verknüpft mit: `QI`, `PARAMS`, `u16ObjId`, `u8AID`
- **REQ**: Service-Anfrage
  - Verknüpft mit: `QI`

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung
  - Verknüpft mit: `QO`, `STATUS`
- **CNF**: Bestätigung der angeforderten Service-Anfrage
  - Verknüpft mit: `QO`, `STATUS`, `IN`
- **IND**: Indikation von der Ressource
  - Verknüpft mit: `QO`, `STATUS`, `IN`

Daten-Eingänge

- **QI** (`BOOL`): Ereignis-Eingangsqualifizierer
- **PARAMS** (`STRING`): Service-Parameter
- **u16ObjId** (`UINT`): Objekt-ID, Initialwert `ID_NULL`
- **u8AID** (`USINT`): Attribut-ID des Objekts

Daten-Ausgänge

- **QO** (`BOOL`): Ereignis-Ausgangsqualifizierer
- **STATUS** (`STRING`): Service-Status
- **IN** (`DWORD`): Eingabedaten von der Ressource

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Funktionsblock ist ein Service-Interface-FB ohne interne Implementierung im Typ (Schnittstellen-Deklaration). `INIT` initialisiert den Dienst mit `QI`, `PARAMS`, `u16ObjId` und `u8AID` und wird mit `INITO` bestätigt. `REQ` stellt eine Anfrage, die mit `CNF` bestätigt wird; `IND` zeigt eine Indikation von der Ressource an. In allen drei Fällen werden `QO`, `STATUS` und (bei `CNF`/`IND`) der Wert `IN` ausgegeben.

Technische Besonderheiten

- Verwendet den Datentyp `DWORD` (32 Bit) für den Attributwert
- Die zu lesende Attribut-ID wird über den Daten-Eingang `u8AID` angegeben
- Die Objekt-ID ist mit `ID_NULL` (`isobus::UT::Q::const::IDs::ID_NULL`) vorbelegt
- Ursprünglich von fortiss GmbH (Monika Wenger, 2017) als Eingabe-Service-Interface angelegt

Anwendungsszenarien

- Auslesen von Attributwerten (z. B. Attribut-ID eines Objekts im VT) in ISOBUS-Anwendungen

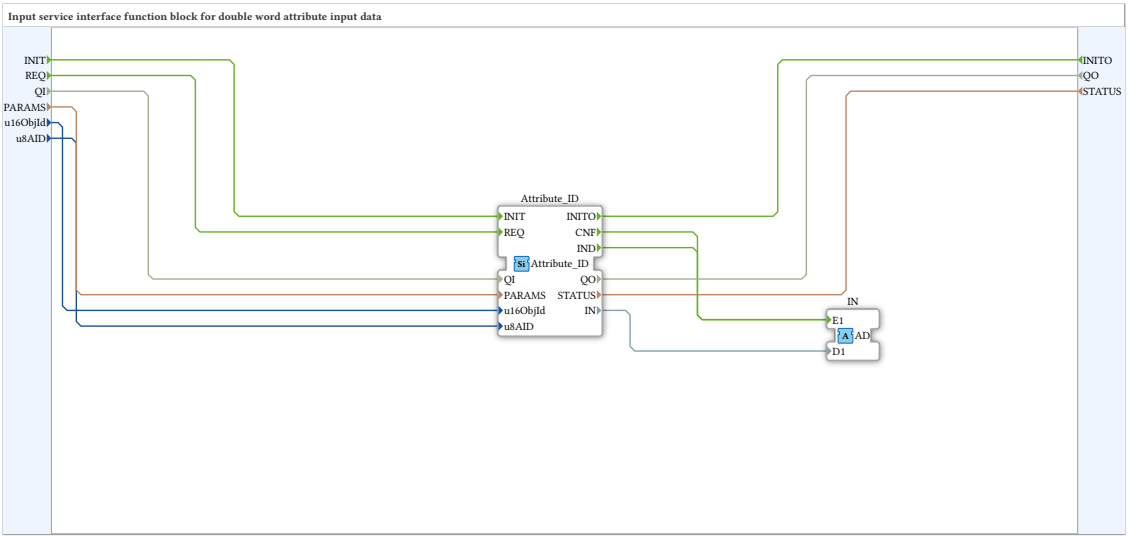
⚖️ Vergleich mit ähnlichen Bausteinen

- NumericValue_ID hat dieselbe Service-Schnittstelle, liest aber den Zahlenwert eines Objekts und besitzt keinen `u8AID`-Eingang.

Fazit

`Attribute_ID` liefert den Attributwert eines VT-Objekts als `DWORD` über eine Service-Interface-Schnittstelle.

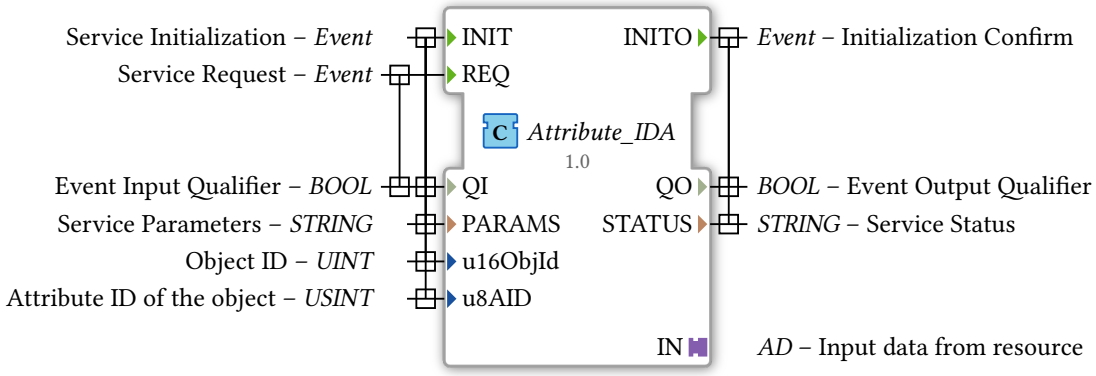
Attribute_IDA



• – ○ • – ○ • – ○ •

Einleitung

`Attribute_IDA` ist ein Eingabe-Service-Interface-Funktionsblock für Doppelwort-Attributdaten (`DWORD`). Er kapselt `Attribute_ID` und stellt den gelesenen Attributwert eines VT-Objekts (Objekt-ID `u16ObjId`, Attribut-ID `u8AID`) nicht über einen Datenausgang, sondern über den Adapter-Plug `IN` (`AD`) bereit. Ereignis und Wert werden dabei in einer Adapterverbindung gebündelt.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Attribute_IDA

- **Typ**: Composite-Funktionsblock
- **Verwendete interne FBs**:
  - **Attribute_ID**: `isobus::UT::io::Attribute::Attribute_ID` – liest das Attribut des VT-Objekts
- **Import**: `isobus::UT::Q::const::IDs::ID_NULL`

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Service-Initialisierung
  - Verknüpft mit: `QI`, `PARAMS`, `u16ObjId`, `u8AID`
- **REQ**: Service-Anfrage
  - Verknüpft mit: `QI`

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung
  - Verknüpft mit: `QO`, `STATUS`

Daten-Eingänge

- **QI** (`BOOL`): Ereignis-Eingangsqualifizierer
- **PARAMS** (`STRING`): Service-Parameter
- **u16ObjId** (`UINT`, Initialwert `ID_NULL`): Objekt-ID
- **u8AID** (`USINT`): Attribut-ID des Objekts

Daten-Ausgänge

- **QO** (`BOOL`): Ereignis-Ausgangsqualifizierer
- **STATUS** (`STRING`): Service-Status

Adapter

- **IN** (Plug): `adapter::types::unidirectional::AD` – Eingabedaten von der Ressource (Ereignis `E1`, Daten `D1` mit dem `DWORD`-Attributwert)

Programmablauf und Verbindungen

- `INIT` ⟶ `Attribute_ID.INIT`; `REQ` ⟶ `Attribute_ID.REQ`; `Attribute_ID.INITO` ⟶ `INITO`
- `Attribute_ID.IND` ⟶ `IN.E1` und `Attribute_ID.CNF` ⟶ `IN.E1`
- `u16ObjId`, `u8AID`, `PARAMS`, `QI` ⟶ gleichnamige Eingänge von `Attribute_ID`
- `Attribute_ID.IN` ⟶ `IN.D1`
- `Attribute_ID.STATUS` ⟶ `STATUS`; `Attribute_ID.QO` ⟶ `QO`

Technische Besonderheiten

- Sowohl `IND` als auch `CNF` des inneren Bausteins lösen `IN.E1` aus; der Wert liegt jeweils an `IN.D1`
- Der Adapter bündelt Ereignis- und Datenfluss in einer Verbindungslinie
- Version 1.0 (2026-09-15)

Anwendungsszenarien

- Auslesen von Attributwerten eines VT-Objekts und Weiterverarbeitung über `AD`-Adapter, z. B. über `AD_TO_AX`-Konverter

⚖️ Vergleich mit ähnlichen Bausteinen

- Attribute_ID stellt `IN` als Datenausgang und `CNF`/`IND` als Ereignisausgänge bereit, `Attribute_IDA` bündelt beides im Adapter `IN`.

Fazit

`Attribute_IDA` liefert den Attributwert eines VT-Objekts als `DWORD` über einen `AD`-Adapter.

Auxiliary

IN

🎧 Podcast

- 4diac IDE: Dein „Hello World“ der Automatisierung – Das Blinking Tutorial Lokal
- 4diac IDE: Dein Open-Source-Werkzeugkasten für verteilte Industrieautomatisierung nach IEC 61499
- 4diac IDE: Wie der IEC 61499 Standard die Industrieautomatisierung revolutioniert
- 4diac-Präsentation: Zielgruppen, Struktur und Alleinstellungsmerkmal Schärfen
- Den Software-Drachen zähmen: Industrielle Automatisierung und die Zukunft der Produktion

📺 Video

- Decoding the datasheet
- Löten wie ein Profi
- Via in Pad Plated Over

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

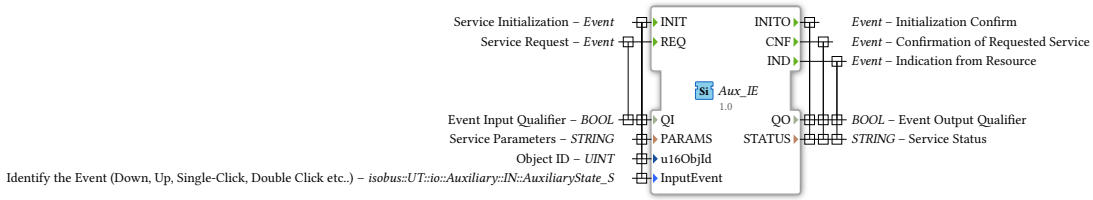
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Aux_IE

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Aux_IE Funktionsblock ist ein Eingabeservice-Interface-Funktionsblock für Ereigniseingabedaten. Er dient zur Verarbeitung von Hilfseingaben (Auxiliary Inputs) und ermöglicht die Erkennung verschiedener Eingabeereignisse wie Drücken, Loslassen, Einfachklick, Doppelklick etc.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung
  - Verknüpft mit: QI, PARAMS, u16ObjId, InputEvent
- **REQ**: Service-Anfrage
  - Verknüpft mit: QI

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
  - Verknüpft mit: QO, STATUS
- **CNF**: Bestätigung der angeforderten Service-Anfrage
  - Verknüpft mit: QO, STATUS
- **IND**: Anzeige von der Ressource
  - Verknüpft mit: QO, STATUS

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingabe-Qualifizierer
- **PARAMS** (STRING): Service-Parameter
- **u16ObjId** (UINT): Objekt-ID, Initialwert: ID_NULL
- **InputEvent** (AuxiliaryState_S): Identifiziert das Ereignis (Down, Up, Single-Click, Double-Click etc.), Initialwert: Invalid

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgabe-Qualifizierer
- **STATUS** (STRING): Service-Status

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Aux_IE Block verarbeitet Hilfseingabeereignisse und bietet eine standardisierte Schnittstelle für die Kommunikation mit Eingabegeräten. Bei der Initialisierung (INIT) werden die Service-Parameter und

Objekt-ID konfiguriert. Über REQ-Ereignisse können Service-Anfragen gestellt werden, während IND-Ereignisse eingehende Signale von der Ressource anzeigen.

Technische Besonderheiten

- Verwendet spezifische ISOBUS-Datentypen für die Landtechnik
- Unterstützt verschiedene Eingabeereignis-Typen über die AuxiliaryState-Struktur
- Initialisierung mit standardisierten NULL-Werten (ID_NULL)
- String-basierte Statusrückmeldungen

Zustandsübersicht

Der Block verfügt über einen Initialisierungszustand (INIT/INITO) und operative Zustände für Service-Anfragen (REQ/CNF) sowie Ressourcen-Anzeigen (IND).

Anwendungsszenarien

- Steuerung von landwirtschaftlichen Maschinen über Hilfseingaben
- Verarbeitung von Benutzerinteraktionen in ISOBUS-Systemen
- Erkennung komplexer Eingabemuster (Einzelklick, Doppelklick)
- Integration in Steuerungssysteme für Agrartechnik

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Eingangsblöcken bietet Aux_IE erweiterte Funktionalität für die Erkennung komplexerer Eingabeereignisse und ist speziell für ISOBUS-Umgebungen optimiert.

🛠️ Zugehörige Übungen

- Uebung_010b3
- Uebung_010b3_AX
- Uebung_010bA2
- Uebung_010bA2_AX
- Uebung_010bA3
- Uebung_010bA3_AX
- Uebung_010bA4
- Uebung_010bA4_AX

Fazit

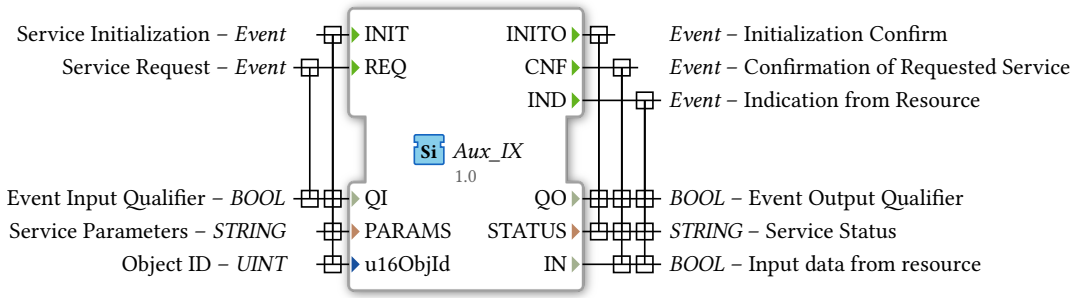
Der Aux_IE Funktionsblock stellt eine leistungsfähige Schnittstelle für die Verarbeitung von Hilfseingabeereignissen in ISOBUS-basierten Steuerungssystemen dar und ermöglicht die zuverlässige Erkennung verschiedener Benutzerinteraktionen.

Aux_IX

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Aux_IX Funktionsblock ist ein Service-Interface-Funktionsblock für boolesche Eingangsdaten. Er dient als Schnittstelle für Eingabedienste und ermöglicht die Kommunikation mit Ressourcen zur Abfrage von digitalen Eingangssignalen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung - Initialisiert den Funktionsblock mit den Parametern QI, PARAMS und u16ObjId
- **REQ**: Service-Anfrage - Löst eine Service-Anfrage mit dem Qualifier QI aus

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung - Bestätigt die Initialisierung mit QO und STATUS
- **CNF**: Bestätigung der angeforderten Service-Anfrage - Liefert QO, STATUS und IN-Daten
- **IND**: Indikation von der Ressource - Signalisiert Datenverfügbarkeit mit QO, STATUS und IN-Daten

Daten-Eingänge

- **QI**: BOOL - Event Input Qualifier (Ereignis-Eingangsqualifier)
- **PARAMS**: STRING - Service-Parameter
- **u16ObjId**: UINT - Objekt-ID mit Initialwert ID_NULL

Daten-Ausgänge

- **QO**: BOOL - Event Output Qualifier (Ereignis-Ausgangsqualifier)
- **STATUS**: STRING - Service-Status
- **IN**: BOOL - Eingangsdaten von der Ressource

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Aux_IX Funktionsblock arbeitet als Eingabeservice-Schnittstelle für boolesche Daten. Bei der Initialisierung (INIT) werden die Service-Parameter und die Objekt-ID konfiguriert. Service-Anfragen (REQ) lösen die Abfrage der Eingangsdaten von der zugehörigen Ressource aus. Die Daten werden über die CNF- und IND-Ereignisse zurückgemeldet, wobei der STATUS-Ausgang Informationen über den Erfolg oder Fehler der Operation liefert.

Technische Besonderheiten

- Verwendet spezifische TypeHash-Identifikation für 4diac
- Initialisiert die Objekt-ID mit einem Standardwert (ID_NULL)
- Unterstützt parametrierbare Service-Konfiguration über STRING-Parameter
- Bietet sowohl anforderungsbasierte (CNF) als auch ereignisbasierte (IND) Datenausgabe

Zustandsübersicht

Der Funktionsblock durchläuft folgende Zustände:

1. **Nicht initialisiert**: Vor der INIT-Verarbeitung
2. **Initialisiert**: Nach erfolgreicher INIT-Verarbeitung mit INITO-Bestätigung
3. **Bereit**: Kann REQ-Anfragen verarbeiten
4. **Aktiv**: Während der Datenabfrage von der Ressource

Anwendungsszenarien

- Abfrage von digitalen Eingangssignalen in Steuerungssystemen
- Integration von Hardware-Eingängen in IEC 61499-Applikationen
- Überwachung von Schalterzuständen und Sensoren
- Implementierung von standardisierten Eingabeschnittstellen in verteilten Systemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Eingangsblöcken bietet Aux_IX erweiterte Service-Funktionalität mit parametrierbarer Konfiguration, Statusrückmeldungen und sowohl anforderungs- als auch ereignisgesteuerter Datenbereitstellung.

🛠️ Zugehörige Übungen

- Uebung_010b1

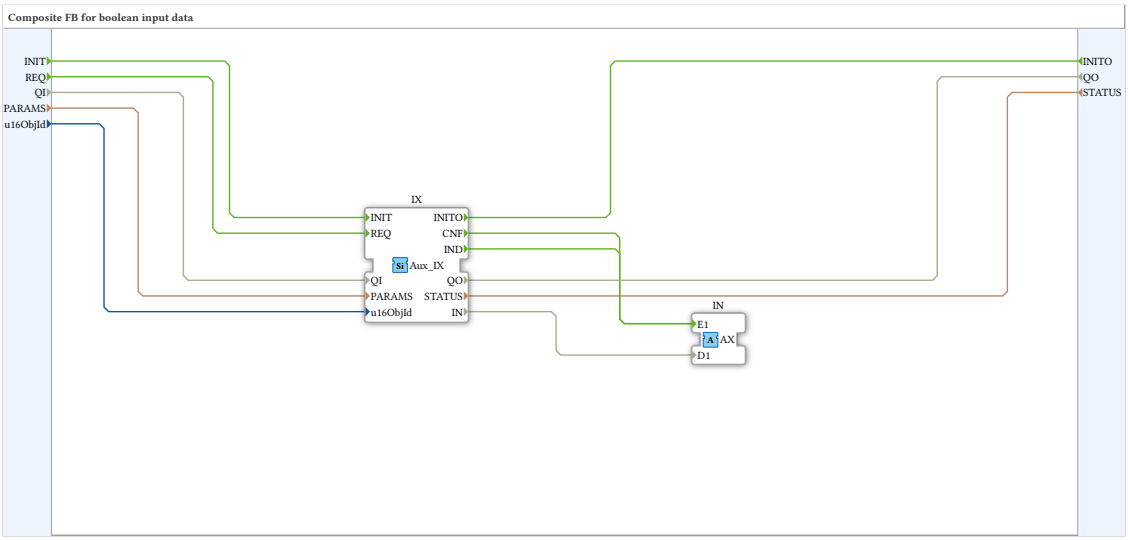
Fazit

Der Aux_IX Funktionsblock stellt eine robuste und flexible Lösung für die Integration von booleschen Eingangsdaten in IEC 61499-basierte Steuerungssysteme dar. Durch seine Service-orientierte Architektur und umfangreiche Statusrückmeldungen eignet er sich besonders für anspruchsvolle Industrieanwendungen mit hohen Anforderungen an Zuverlässigkeit und Diagnosefähigkeit.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

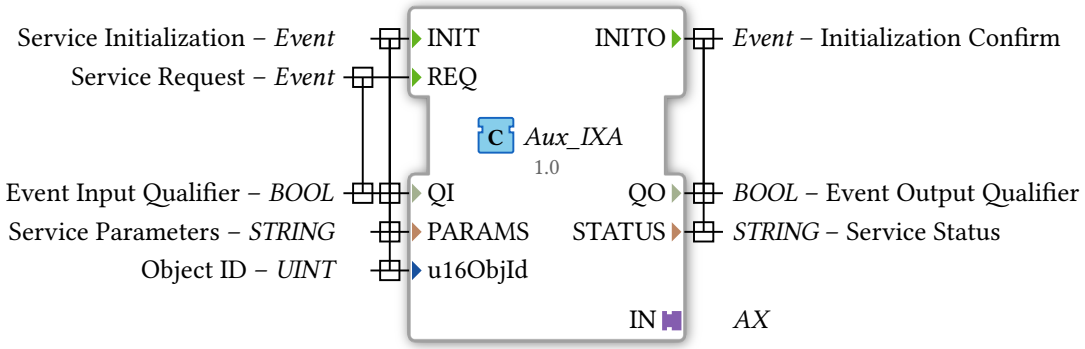
Aux_IXA



• – ○ • – ○ • – ○ •

Einleitung

Der Aux_IXA ist ein Composite-Funktionsblock zur Verarbeitung von booleschen Eingangsdaten. Er dient als Wrapper für den Basis-Funktionsblock Aux_IX und bietet eine erweiterte Schnittstelle für die Integration in ISOBUS-kompatible Systeme.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung mit den zugehörigen Daten QI, PARAMS und u16ObjId
- **REQ**: Service-Anfrage mit dem zugehörigen Daten QI

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung mit den zugehörigen Daten QO und STATUS

Daten-Eingänge

- **QI**: Event Input Qualifier (BOOL) - Qualifizierer für Ereigniseingänge
- **PARAMS**: Service Parameters (STRING) - Konfigurationsparameter für den Service
- **u16ObjId**: Object ID (UINT) - Objektkennung mit Initialwert ID_NULL

Daten-Ausgänge

- **QO**: Event Output Qualifier (BOOL) - Qualifizierer für Ereignisausgänge
- **STATUS**: Service Status (STRING) - Statusinformationen des Services

Adapter

- **IN**: Unidirektionaler Adapter vom Typ AX für die Datenkommunikation

Funktionsweise

Der Aux_IXA fungiert als Composite-Block, der den Basis-Funktionsblock Aux_IX kapselt. Bei INIT-Ereignis werden alle Parameter an den internen IX-Block weitergeleitet. REQ-Ereignisse lösen Service-Anfragen aus, während INITO die erfolgreiche Initialisierung bestätigt. Der Adapter IN ermöglicht die bidirektionale Kommunikation mit externen Komponenten.

Technische Besonderheiten

- Integration in ISOBUS-Umgebungen durch spezielle Package-Struktur
- Verwendung von TypeHash für Typidentifikation
- Unidirektionaler Adapter für standardisierte Schnittstellen
- Initialisierung mit NULL-Objekt-ID als Standardwert

Zustandsübersicht

Der Block durchläuft folgende Zustände:

1. **Nicht initialisiert**: Vor INIT-Ereignis
2. **Initialisierung**: Während INIT-Verarbeitung
3. **Bereit**: Nach erfolgreicher Initialisierung (INITO)
4. **Service-Modus**: Bei REQ-Ereignisverarbeitung

Anwendungsszenarien

- ISOBUS-kompatible Landmaschinensteuerung
- Boolesche Signalverarbeitung in Agrartechnik-Systemen
- Wrapper-Implementierung für bestehende IO-Funktionalität
- Parameterkonfiguration für Auxiliary-Inputs

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen booleschen Eingangsblöcken bietet Aux_IXA:

- Erweiterte Parametrisierungsmöglichkeiten
- ISOBUS-Spezifikation
- Adapter-basierte Schnittstelle
- Composite-Struktur für bessere Wiederverwendbarkeit

🛠️ Zugehörige Übungen

- Uebung_010b1_AX

Fazit

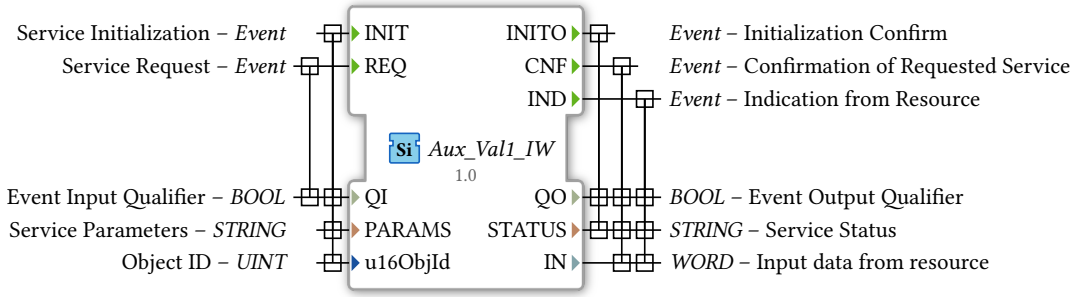
Der Aux_IXA stellt einen robusten Composite-Funktionsblock für die boolesche Eingangsdatenverarbeitung in ISOBUS-Umgebungen dar. Durch seine klare Schnittstellendefinition und Adapter-Integration ermöglicht er eine flexible Einbindung in komplexe Steuerungssysteme der Agrartechnik.

Aux_Val1_IW

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Aux_Val1_IW Funktionsblock ist ein Eingabeservice-Schnittstellen-Funktionsblock für Wort-Eingabedaten. Er dient als Schnittstelle zwischen der Steuerungslogik und externen Ressourcen zur Verarbeitung von 16-Bit-Wort-Eingabedaten im ISOBUS-Kontext.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung
  - Verknüpft mit: QI, PARAMS, u16ObjId
- **REQ**: Service-Anfrage
  - Verknüpft mit: QI

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
  - Verknüpft mit: QO, STATUS
- **CNF**: Bestätigung der angeforderten Service-Anfrage
  - Verknüpft mit: QO, STATUS, IN
- **IND**: Indikation von der Ressource
  - Verknüpft mit: QO, STATUS, IN

Daten-Eingänge

- **QI**: Event-Eingabe-Qualifier (BOOL)
- **PARAMS**: Service-Parameter (STRING)
- **u16ObjId**: Objekt-ID (UINT) - Initialwert: ID_NULL

Daten-Ausgänge

- **QO**: Event-Ausgabe-Qualifier (BOOL)
- **STATUS**: Service-Status (STRING)
- **IN**: Eingabedaten von der Ressource (WORD)

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Funktionsblock ermöglicht die Kommunikation mit externen Ressourcen zur Erfassung von Wort-Eingabedaten. Über die INIT-Initialisierung werden die Service-Parameter und die Objekt-ID konfiguriert. Die REQ-Anfrage löst die Datenerfassung aus, während IND-Ereignisse asynchrone Datenupdates von der Ressource signalisieren.

Technische Besonderheiten

- Verwendet WORD-Datentyp für 16-Bit-Datenerfassung
- ISOBUS-kompatible Schnittstelle
- Unterstützt asynchrone Datenindikationen
- Initialisierung mit konfigurierbaren Parametern
- Fehlerstatus-Rückmeldung über STATUS-Ausgabe

Zustandsübersicht

Der Funktionsblock durchläuft folgende Hauptzustände:

1. **Nicht initialisiert**: Wartet auf INIT-Ereignis
2. **Initialisiert**: Bereit für REQ-Anfragen
3. **Aktiv**: Verarbeitet Datenanfragen und -indikationen
4. **Fehler**: Statusmeldung bei Problemen

Anwendungsszenarien

- ISOBUS-basierte Landmaschinensteuerung
- Erfassung von 16-Bit-Sensordaten
- Kommunikation mit externen I/O-Modulen
- Automatisierungssysteme in der Agrartechnik

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen Eingabeblöcken bietet Aux_Val1_IW erweiterte ISOBUS-Funktionalität mit parametrierbarer Initialisierung, Statusrückmeldungen und asynchronen Indikationsmechanismen.

Fazit

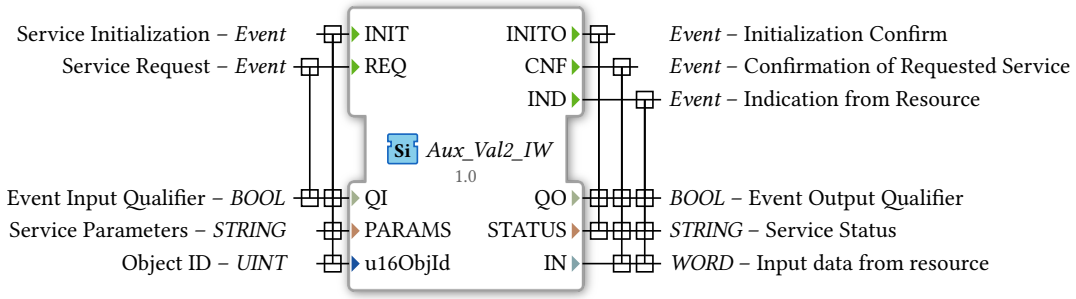
Der Aux_Val1_IW Funktionsblock stellt eine robuste und flexible Lösung für die Erfassung von Wort-Eingabedaten in ISOBUS-Umgebungen dar, mit umfassender Fehlerbehandlung und konfigurierbaren Service-Parametern.

Aux_Val2_IW

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Aux_Val2_IW ist ein Eingabeservice-Schnittstellen-Funktionsblock für Wort-Eingabedaten. Dieser Baustein dient als Schnittstelle zur Kommunikation mit Ressourcen und ermöglicht die Verarbeitung von 16-Bit-Wort-Eingabedaten in verteilten Automatisierungssystemen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung - Initialisiert den Funktionsblock mit Parametern
- **REQ**: Service-Anfrage - Löst eine Serviceanfrage aus

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung - Bestätigt die erfolgte Initialisierung
- **CNF**: Bestätigung der angeforderten Service - Bestätigt eine Serviceanfrage
- **IND**: Indikation von der Ressource - Signalisiert eingehende Daten von der Ressource

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangsqualifizierer - Aktiviert/deaktiviert die Verarbeitung
- **PARAMS** (STRING): Service-Parameter - Konfigurationsparameter für den Service
- **u16ObjId** (UINT): Objekt-ID - Identifikator für das Datenobjekt (Initialwert: ID_NULL)

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifizierer - Status der Ausgangsverarbeitung

- **STATUS** (STRING): Service-Status - Rückmeldung über den Servicezustand
- **IN** (WORD): Eingabedaten von der Ressource - Die empfangenen 16-Bit-Wortdaten

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Funktionsblock arbeitet als Service-Schnittstelle für Wort-Eingabedaten. Bei der INIT-Initialisierung werden die Service-Parameter und die Objekt-ID konfiguriert. Über REQ-Ereignisse können Serviceanfragen gestellt werden, die mit CNF-Ereignissen bestätigt werden. Bei eingehenden Daten von der Ressource wird ein IND-Ereignis mit den empfangenen Wortdaten ausgegeben.

Technische Besonderheiten

- Verarbeitet 16-Bit-Wort-Datentypen (WORD)
- Unterstützt konfigurierbare Objekt-IDs für die Datenidentifikation
- Bietet umfangreiche Statusrückmeldungen über den SERVICE-Status
- Ermöglicht parametrierbare Service-Konfiguration

Zustandsübersicht

Der Funktionsblock durchläuft folgende Zustände:

1. **Nicht initialisiert**: Vor der INIT-Initialisierung
2. **Initialisiert**: Nach erfolgreicher INIT-Verarbeitung mit INITO-Bestätigung
3. **Bereit**: Für REQ-Anfragen und IND-Indikationen
4. **Service aktiv**: Während der Bearbeitung von Serviceanfragen

Anwendungsszenarien

- Einlesen von 16-Bit-Sensordaten in verteilten Systemen
- Kommunikation mit ISOBUS-kompatiblen Geräten
- Eingabe von Wort-Daten in Automatisierungsanwendungen
- Schnittstelle zu externen Ressourcen für Datenerfassung

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren Eingabebausteinen bietet Aux_Val2_IW erweiterte Service-Funktionalität mit parametrierbarer Konfiguration, Statusrückmeldungen und Objekt-Identifikation. Er ist speziell für die Verarbeitung von Wort-Daten (16-Bit) ausgelegt.

Fazit

Der Aux_Val2_IW Funktionsblock stellt eine leistungsfähige Service-Schnittstelle für die Verarbeitung von Wort-Eingabedaten in industriellen Automatisierungssystemen bereit. Durch seine umfangreiche Parametrierbarkeit und Statusrückmeldungen eignet er sich besonders für anspruchsvolle Anwendungen mit hohen Anforderungen an Zuverlässigkeit und Diagnosefähigkeit.

AuxiliaryState

• – ◦ • – ◦ • – ◦ •

Einleitung

AuxiliaryState ist eine Sammlung von globalen Konstanten für Hilfseingänge in ISOBUS-Systemen. Diese Konstanten definieren verschiedene Zustände von Auxiliary-Eingängen, die typischerweise für Steuerfunktionen in landwirtschaftlichen und mobilen Arbeitsmaschinen verwendet werden.

Schnittstellenstruktur

Bei AuxiliaryState handelt es sich um globale Konstanten und nicht um einen Funktionsblock im klassischen Sinne. Daher verfügt diese Komponente über keine Ereignis- oder Datenschnittstellen.

Ereignis-Eingänge

- Nicht vorhanden (Globale Konstanten)

Ereignis-Ausgänge

- Nicht vorhanden (Globale Konstanten)

Daten-Eingänge

- Nicht vorhanden (Globale Konstanten)

Daten-Ausgänge

- Nicht vorhanden (Globale Konstanten)

Adapter

- Nicht vorhanden

Funktionsweise

AuxiliaryState definiert eine Reihe von vordefinierten Konstanten für den Datentyp `AuxiliaryState_S`. Jede Konstante repräsentiert einen spezifischen Zustand eines Auxiliary-Eingangs mit einem entsprechenden numerischen Wert und optionalen Zusatzparametern.

Technische Besonderheiten

- Alle Konstanten sind vom Typ `isobus::UT::io::Auxiliary::IN::AuxiliaryState_S`
- Die Wertebereiche reichen von 0 bis 255
- Einige Konstanten enthalten zusätzlich das Flag `bCyclic := FALSE`
- Die Struktur folgt ISOBUS-Standards für landwirtschaftliche Steuerungen

Zustandsübersicht

Die definierten Konstanten umfassen folgende Zustände:

Konstante	Wert	Beschreibung
`AuxDisabled`	0	Aus - rückwärts, unten, links oder nicht gedrückt
`AuxDisabled_START`	0	Aus - rückwärts, unten, links oder nicht gedrückt (mit bCyclic=FALSE)
`AuxEnabled`	1	Ein - vorwärts, oben, rechts oder gedrückt
`AuxEnabled_START`	1	Ein - vorwärts, oben, rechts oder gedrückt (mit bCyclic=FALSE)
`AuxHeld`	2	Gehalten (vorwärts, oben oder rechts)
`AuxHeld_START`	2	Gehalten (vorwärts, oben oder rechts) (mit bCyclic=FALSE)
`AuxEnabledBackwards`	4	Momentan ein - rückwärts, unten oder links
`AuxHeldBackwards`	8	Gehalten (rückwärts, unten oder links)
`AuxHeldBackwards_START`	8	Gehalten (rückwärts, unten oder links) (mit bCyclic=FALSE)
`AuxDefault`	9	Standardwert
`Invalid`	255	Ungültiger Wert

Anwendungsszenarien

- Steuerung von Hydraulikfunktionen in Traktoren
- Bedienung von Anbaugeräten über ISOBUS-Schnittstellen
- Zustandsüberwachung von Hilfsfunktionen in mobilen Arbeitsmaschinen
- Implementierung von standardisierten Eingabezuständen in landwirtschaftlichen Steuerungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu Funktionsblöcken mit aktiver Logik handelt es sich bei AuxiliaryState um passive Konstantendefinitionen, die als Referenzwerte in anderen Funktionsblöcken verwendet werden können. Sie bieten eine standardisierte Möglichkeit, Auxiliary-Zustände in ISOBUS-Umgebungen zu definieren.

Fazit

AuxiliaryState stellt eine umfassende Sammlung von standardisierten Konstanten für die Behandlung von Auxiliary-Eingängen in ISOBUS-Systemen bereit. Die klare Definition der verschiedenen Zustände ermöglicht eine konsistente Implementierung von Steuerungslogik über verschiedene Systeme hinweg und erleichtert die Wartbarkeit und Lesbarkeit des Codes.

AuxiliaryState_S

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Datentyp `AuxiliaryState_S` ist eine strukturierte Datenstruktur (Struct), die den Zustand eines Hilfseingangs (Auxiliary Input) repräsentiert. Er wird im Kontext der ISOBUS-Schnittstelle (ISO 11783) für die Kommunikation in landwirtschaftlichen und forstwirtschaftlichen Maschinen verwendet. Dieser Datentyp kapselt sowohl den eigentlichen Wert des Eingangs als auch eine Steuerinformation für dessen zyklische Übertragung.

Schnittstellenstruktur

`AuxiliaryState_S` ist ein strukturierter Datentyp (Struct) und besitzt daher keine Ereignis- oder Adapterschnittstellen im herkömmlichen Sinne eines Funktionsbausteins. Seine Struktur besteht aus den folgenden zwei Datenfeldern:

Daten-Eingänge

(Nicht anwendbar – Dies ist ein Datentyp, kein Funktionsbaustein)

Daten-Ausgänge

(Nicht anwendbar – Dies ist ein Datentyp, kein Funktionsbaustein)

Strukturierte Datenfelder

Name	Datentyp	Initialwert	Kommentar
`value`	`USINT` (0…255)	255	Repräsentiert den Wert des Hilfseingangs.
`bCyclic`	`BOOL`	`TRUE`	Steuert die zyklische Übertragung. Wenn auf `FALSE` gesetzt, wird die Nachricht nicht alle 200 ms wiederholt.

Funktionsweise

Der Datentyp dient als Container für Informationen über einen Hilfseingang. Das Feld `value` speichert den eigentlichen, gelesenen Wert. Der Initialwert 255 kann einen ungültigen oder undefinierten Zustand repräsentieren. Das Feld `bCyclic` ist ein Steuerflag, das das Übertragungsverhalten in einem ISOBUS-Netzwerk beeinflusst. Gemäß dem Kommentar wird bei `bCyclic = TRUE` die zugehörige Nachricht standardmäßig alle 200 Millisekunden zyklisch gesendet. Wird `bCyclic` auf `FALSE` gesetzt, unterbleibt diese regelmäßige Wiederholung, was den Netzwerkverkehr reduzieren kann, wenn sich der Wert nicht häufig ändert.

Technische Besonderheiten

- **ISOBUS-Kontext:** Der Datentyp ist spezifisch für die ISOBUS-Kommunikation (Package `isobus::UT::io::Auxiliary::IN`) konzipiert.
- **Zyklussteuerung:** Die integrierte Steuerung der Zyklizität (`bCyclic`) ist ein charakteristisches Merkmal, das die Effizienz der Buskommunikation berücksichtigt.
- **Initialwerte:** Beide Felder haben definierte Initialwerte (`value=255`, `bCyclic=TRUE`), was für eine deterministische Initialisierung sorgt.

Zustandsübersicht

(Nicht anwendbar – Dies ist ein passiver Datentyp ohne Zustandsautomaten.)

Anwendungsszenarien

Dieser Struct wird typischerweise als Dateneingang oder -ausgang in Funktionsbausteinen (FBs) verwendet, die ISOBUS-Hilfseingänge verarbeiten oder steuern. Beispiele sind:

- FBs zum Auslesen des Zustands von Schaltern, Potentiometern oder Sensoren, die als Hilfseingänge konfiguriert sind.
- FBs, die Hilfseingangswerte verarbeiten und an andere Systemkomponenten weiterleiten.
- FBs, die basierend auf dem `bCyclic`-Flag entscheiden, ob ein Wert aktualisiert und gesendet werden muss.

⚖️ Vergleich mit ähnlichen Bausteinen

Als reiner Datentyp ist ein direkter Vergleich mit Funktionsbausteinen nicht sinnvoll. Innerhalb der ISOBUS-Datentypen könnte er mit einfacheren Datentypen wie einem reinen `USINT` für den Wert verglichen werden. Der Vorteil von `AuxiliaryState_S` liegt in der Erweiterung um die Metainformation zur Übertragungssteuerung (`bCyclic`), was eine höhere Integration und effizientere Kommunikation im ISOBUS-Netzwerk ermöglicht.

Fazit

`AuxiliaryState_S` ist ein spezialisierter, strukturierter Datentyp für die ISOBUS-Kommunikation. Er bietet eine elegante Möglichkeit, nicht nur den Wert eines Hilfseingangs, sondern auch ein wichtiges Protokollattribut für dessen zyklische Übertragung zu kapseln. Seine Verwendung fördert die Klarheit und Wartbarkeit von Applikationen, die ISOBUS-I/O-Funktionalitäten implementieren.

OUT

🎧 Podcast

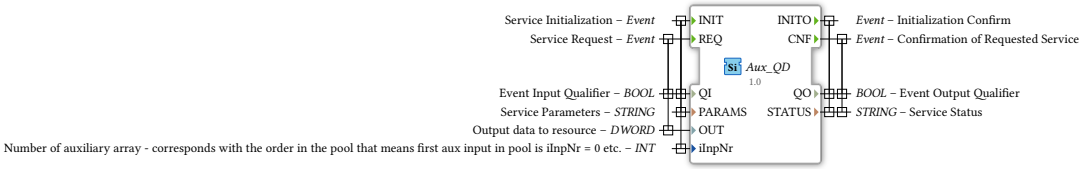
- ISOBUS Output Meter: Dynamische Anzeigen meistern – vom Zeiger bis zur Visualisierung auf dem VT
- ISOBUS-Balkendiagramm: Das Output Linear Bar Graph Objekt der ISO 11783-6 entschlüsselt
- ISOBUS-Terminals: Zahlen verstehen – NumberVariable, InputNumber & OutputNumber erklärt

Aux_QD

• – ○ • – ○ • – ○ •

Einleitung

Der Aux_QD Funktionsblock ist ein Ausgabeservice-Interface-Baustein für die Verarbeitung von Doppelwort-Ausgabedaten (DWORD). Er dient als Schnittstelle für Auxiliary-Ausgabefunktionen und unterstützt verschiedene Auxiliary-Typen zur flexiblen Datenausgabe.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierungsereignis
- **REQ**: Service-Anfrageereignis

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
- **CNF**: Bestätigung der angeforderten Service-Operation

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangsqualifizierer

- **PARAMS** (STRING): Service-Parameter
- **OUT** (DWORD): Ausgabedaten für die Ressource
- **iInpNr** (INT): Nummer des Auxiliary-Arrays (entspricht der Reihenfolge im Pool, Startwert: 0)

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifizierer
- **STATUS** (STRING): Service-Statusinformation

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Aux_QD-Block verarbeitet Doppelwort-Daten (32-Bit) und gibt diese über Auxiliary-Schnittstellen aus. Bei der Initialisierung (INIT) werden die Service-Parameter konfiguriert. Über das REQ-Ereignis werden die Ausgabedaten an die entsprechende Ressource gesendet. Der Block unterstützt mehrere Auxiliary-Typen und ermöglicht die parallele Ausgabe beider Werte in einer Operation.

Technische Besonderheiten

- Unterstützt Auxiliary-Typen: 0, 1, 2, 3, 4, 6, 11, 12, 13
- Verarbeitet 32-Bit DWORD-Daten
- Initialwert für iInpNr ist 0 (erstes Auxiliary-Array im Pool)
- Parallele Ausgabe beider Werte in einer Operation möglich

Zustandsübersicht

Der Block verfügt über zwei Hauptzustände:

1. **Initialisierungszustand**: Verarbeitung der INIT-Anfrage mit Parameterkonfiguration
2. **Betriebszustand**: Verarbeitung von REQ-Anfragen mit Datenausgabe

Anwendungsszenarien

- ISOBUS-kompatible Steuerungssysteme
- Landwirtschaftliche Maschinensteuerungen
- Industrielle Automatisierungssysteme
- Embedded Systems mit Auxiliary-Schnittstellen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen Ausgabeblöcken bietet Aux_QD erweiterte Funktionen für Auxiliary-Schnittstellen und unterstützt spezifische ISOBUS-kompatible Datentypen. Die parallele Ausgabefähigkeit unterscheidet ihn von sequenziellen Ausgabeblöcken.

Fazit

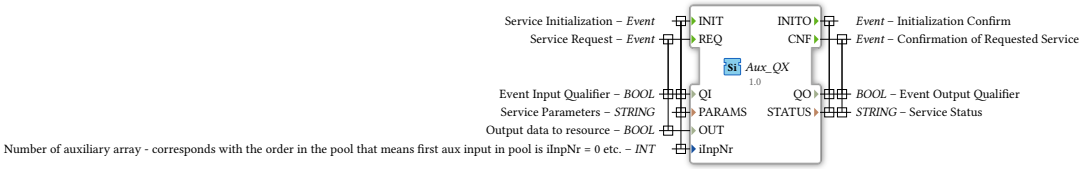
Der Aux_QD Funktionsblock stellt eine leistungsstarke Lösung für die Ausgabe von Doppelwort-Daten über Auxiliary-Schnittstellen dar. Seine ISOBUS-Kompatibilität und die Unterstützung verschiedener Auxiliary-Typen machen ihn besonders geeignet für anspruchsvolle Automatisierungsanwendungen in industriellen und mobilen Systemen.

Aux_QX

• – ○ • – ○ • – ○ •

Einleitung

Der Aux_QX Funktionsblock ist ein Ausgabeservice-Interface-Funktionsblock für boolesche Ausgabedaten. Er dient als Schnittstelle zu Auxiliary-Ausgabegeräten und ermöglicht die Steuerung von booleschen Ausgabesignalen über einen definierten Service-Mechanismus.



Schnittstellenstruktur

Ereignis-Eingänge
- **INIT**: Service-Initialisierungsereignis
- **REQ**: Service-Anfrageereignis

Ereignis-Ausgänge
- **INITO**: Initialisierungsbestätigung
- **CNF**: Bestätigung der angeforderten Service-Operation

Daten-Eingänge
- **QI** (BOOL): Ereignis-Eingangsqualifizierer
- **PARAMS** (STRING): Service-Parameter
- **OUT** (BOOL): Ausgabedaten zur Ressource
- **iInpNr** (INT): Nummer des Auxiliary-Arrays - entspricht der Reihenfolge im Pool (erster Aux-Eingang im Pool hat iInpNr = 0, etc.)

Daten-Ausgänge
- **QO** (BOOL): Ereignis-Ausgangsqualifizierer
- **STATUS** (STRING): Service-Status

Funktionsweise
Der Aux_QX Funktionsblock implementiert ein standardisiertes Service-Interface für boolesche Ausgabedaten. Bei Empfang eines INIT-Ereignisses wird der Service initialisiert, wobei die Parameter PARAMS und iInpNr zur Konfiguration verwendet werden. Das REQ-Ereignis triggert die Ausgabe des booleschen Wertes an die spezifizierte Auxiliary-Ressource. Jede Operation wird durch entsprechende Bestätigungsereignisse (INITO, CNF) quittiert.

Technische Besonderheiten
- Unterstützt die Adressierung von Auxiliary-Ausgängen über den iInpNr-Parameter
- Flexible Parameterkonfiguration über STRING-Parameter
- Vollständige Statusrückmeldung über STATUS-Ausgang
- Qualifizierte Ereignisbehandlung durch QI/QO-Signale

Zustandsübersicht
Der Funktionsblock folgt einem typischen Service-Interface-Muster mit Initialisierungs- und Betriebszuständen. Nach erfolgreicher Initialisierung (INIT → INITO) können Service-Anfragen (REQ → CNF) verarbeitet werden.

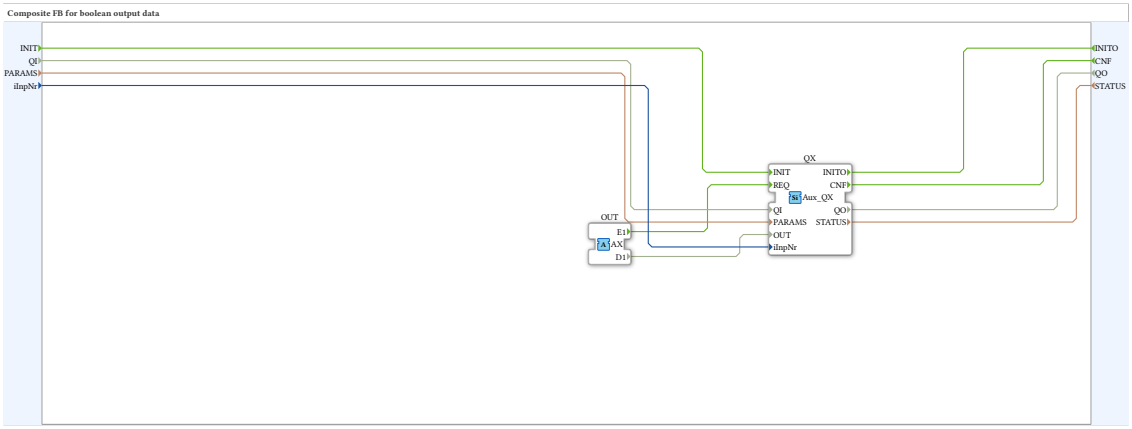
Anwendungsszenarien
- Steuerung von Auxiliary-Ausgabegeräten in ISOBUS-Umgebungen
- Anbindung von booleschen Aktoren in landwirtschaftlichen Maschinen
- Integration in Steuerungssysteme für Hilfsfunktionen
- Universelle boolesche Ausgabeschnittstelle in verteilten Automatisierungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen
Im Vergleich zu einfachen booleschen Ausgabeblöcken bietet Aux_QX erweiterte Service-Funktionalität mit Initialisierungssequenzen, Parameterkonfiguration und detailliertem Statusmanagement. Die spezifische Adressierung über iInpNr ermöglicht die Integration in komplexe Gerätepools.

🛠️ Zugehörige Übungen
- IX_TO_Aux_QX

Fazit
Der Aux_QX Funktionsblock stellt eine robuste und standardisierte Lösung für die Ansteuerung von booleschen Auxiliary-Ausgängen dar. Durch sein vollständiges Service-Interface und die flexible Konfigurierbarkeit eignet er sich besonders für den Einsatz in professionellen Automatisierungsumgebungen mit ISOBUS-Kompatibilität.

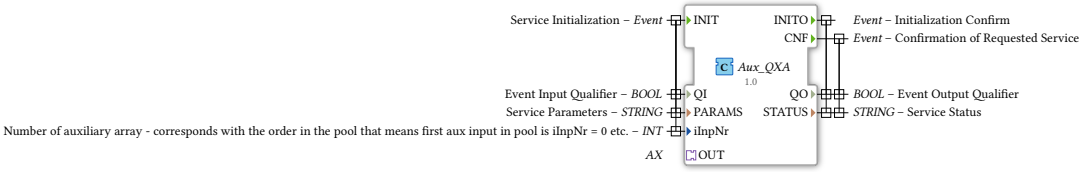
Aux_QXA



• – ○ • – ○ • – ○ •

Einleitung

Der Aux_QXA ist ein zusammengesetzter Funktionsblock für die Ausgabe von booleschen Daten. Er dient als Schnittstelle für Hilfsausgänge (Auxiliary Outputs) in ISOBUS-kompatiblen Systemen und ermöglicht die Steuerung von booleschen Ausgabesignalen über standardisierte Schnittstellen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierungsereignis
  - Verknüpft mit: QI, PARAMS, iInpNr

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
  - Verknüpft mit: QO, STATUS
- **CNF**: Bestätigung des angeforderten Services
  - Verknüpft mit: QO, STATUS

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangsqualifizierer
- **PARAMS** (STRING): Service-Parameter
- **iInpNr** (INT): Nummer des Hilfsausgangs-Arrays (entspricht der Reihenfolge im Pool, wobei der erste Hilfseingang iInpNr = 0 hat)

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifizierer
- **STATUS** (STRING): Service-Status

Adapter

- **OUT**: Unidirektionaler Adapter vom Typ AX
  - Position: x=„–400“, y=„1000“

Funktionsweise

Der Aux_QXA Funktionsblock fungiert als Wrapper für den internen Aux_QX Baustein. Bei Initialisierung (INIT-Ereignis) werden die Konfigurationsparameter übergeben und der interne Baustein initiali-

siert. Über den OUT-Adapter können Ausgabeanforderungen gesendet werden, die dann vom internen Baustein verarbeitet werden. Die Bestätigung erfolgt über die CNF- und INITO-Ausgangsereignisse.

Technische Besonderheiten

- Verwendet einen unidirektionalen Adapter für die Ausgabekommunikation
- Unterstützt parametrierbare Service-Konfiguration über STRING-Parameter
- Ermöglicht die Adressierung mehrerer Hilfsausgänge über den iInpNr-Parameter
- Implementiert standardisierte Initialisierungs- und Bestätigungsprotokolle

Zustandsübersicht

Der Funktionsblock durchläuft folgende Zustände:

1. **Nicht initialisiert**: Vor dem INIT-Ereignis
2. **Initialisierung**: Während der INIT-Verarbeitung
3. **Bereit**: Nach erfolgreicher Initialisierung (INITO)
4. **Aktiv**: Während der Ausgabeverarbeitung (CNF)

Anwendungsszenarien

- Steuerung von booleschen Aktoren in landwirtschaftlichen Maschinen
- ISOBUS-kompatible Hilfsausgänge in Traktoren und Anbaugeräten
- Automatisierte Steuerungssysteme in der Agrartechnik
- Vernetzte Steuerung von Peripheriegeräten

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen booleschen Ausgabeblöcken bietet Aux_QXA:

- Erweiterte Parametrierbarkeit über STRING-Parameter
- ISOBUS-Kompatibilität
- Standardisierte Initialisierungsprozeduren
- Adapter-basierte Schnittstelle für erweiterte Kommunikation

🛠️ Zugehörige Übungen

- IXA_TO_Aux_QXA

Fazit

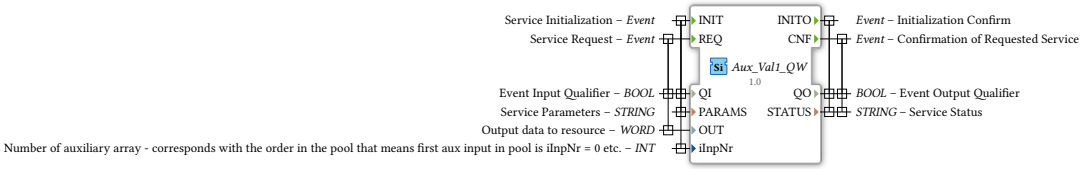
Der Aux_QXA Funktionsblock stellt eine robuste und standardkonforme Lösung für die Steuerung von booleschen Hilfsausgängen in ISOBUS-Systemen dar. Durch seine modulare Architektur und erweiterte Parametrierungsmöglichkeiten eignet er sich besonders für komplexe Steuerungsaufgaben in der modernen Agrartechnik.

Aux_Val1_QW

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Aux_Val1_QW ist ein Ausgabeservice-Schnittstellen-Funktionsblock für Wort-Ausgabedaten. Dieser Baustein dient als Schnittstelle zu Auxiliary-Output-Ressourcen und ermöglicht die Ausgabe von 16-Bit-Datenwörtern an externe Peripheriegeräte.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierungsereignis
- **REQ**: Service-Anforderungsereignis

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
- **CNF**: Bestätigung der angeforderten Serviceausführung

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangsqualifizierer
- **PARAMS** (STRING): Service-Parameter
- **OUT** (WORD): Ausgabedaten zur Ressource
- **iInpNr** (INT): Nummer des Auxiliary-Arrays - entspricht der Reihenfolge im Pool (erster Aux-Eingang im Pool hat iInpNr = 0, etc.)

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifizierer
- **STATUS** (STRING): Service-Status

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Funktionsblock initialisiert und steuert Auxiliary-Ausgabegeräte über den ISOBUS-UT-Protokollstack. Bei der Initialisierung (INIT-Event) werden die Service-Parameter konfiguriert und das Auxiliary-Gerät entsprechend eingerichtet. Bei Service-Anforderungen (REQ-Event) werden die WORD-Daten an das konfigurierte Ausgabegerät gesendet. Der Baustein unterstützt verschiedene Auxiliary-Typen und bietet eine Statusrückmeldung über den Erfolg oder Fehler der Operationen.

Technische Besonderheiten

- Unterstützt Auxiliary-Typen: 0, 1, 2, 3, 4, 6, 11, 12, 13
- Verwendet WORD-Datentyp (16-Bit) für Ausgabedaten
- Implementiert ISOBUS-UT-Protokoll für Auxiliary-IO
- Bietet detaillierte Statusinformationen über STRING-Ausgabe
- Ermöglicht Konfiguration mehrerer Auxiliary-Geräte über iInpNr-Parameter

Zustandsübersicht

Der Baustein verfügt über zwei Hauptbetriebszustände:

1. **Initialisierungszustand**: Verarbeitung der INIT-Anforderung und Konfiguration des Auxiliary-Geräts
2. **Service-Zustand**: Verarbeitung von REQ-Anforderungen und Datenausgabe an das konfigurierte Gerät

Anwendungsszenarien

- Steuerung von Auxiliary-Ausgabegeräten in mobilen Arbeitsmaschinen
- ISOBUS-konforme Peripherieansteuerung
- Landwirtschaftliche und baumaschinenspezifische Anwendungen
- Anbindung von Wort-basierten Ausgabegeräten über UT-Protokoll

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Ausgabeblöcken bietet Aux_Val1_QW erweiterte Funktionen für ISOBUS-Umgebungen, einschließlich Parametrierbarkeit, Statusrückmeldungen und Unterstützung verschiedener Auxiliary-Gerätetypen. Die WORD-Datenbreite ermöglicht die Ansteuerung komplexerer Peripheriegeräte im Vergleich zu einfachen BOOL-Ausgabeblöcken.

Fazit

Der Aux_Val1_QW Funktionsblock stellt eine robuste und standardkonforme Lösung für die Ansteuerung von Auxiliary-Wort-Ausgabegeräten in ISOBUS-Umgebungen dar. Seine flexible Konfigurierbarkeit und umfassende Statusrückmeldung machen ihn besonders geeignet für anspruchsvolle industrielle und mobile Anwendungen.

Button

🎧 Podcast

- ISOBUS Button: Mehr als nur ein Klick – Die Standardisierung der Landtechnik

ButtonActivationCode

• – ◦ • – ◦ • – ◦ •

Einleitung

ButtonActivationCode ist eine Sammlung von globalen Konstanten für die Verarbeitung von Tastenaktivierungen in ISOBUS-kompatiblen Systemen. Diese Konstanten definieren verschiedene Zustände und Ereignisse, die bei der Bedienung von Tasten auftreten können, und werden hauptsächlich in landwirtschaftlichen Anwendungen eingesetzt.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

Keine Adapter vorhanden

Funktionsweise

Die ButtonActivationCode-Konstanten definieren verschiedene Tastenaktivierungszustände als strukturierte Datentypen vom Typ `ButtonActivationCode_S`. Jede Konstante repräsentiert einen spezifischen Zustand einer Taste und enthält einen numerischen Code sowie optionale zusätzliche Parameter.

Technische Besonderheiten

- **Übertragungswiederholungsrate**: Bei Tastendruck/-freigabe und alle 200 ms, wenn die Taste gehalten wird
- **Verriegelbare Tasten**: Wiederholen sich nicht bei gedrückt gehaltener Taste
- **ISOBUS-Kompatibilität**: Speziell für ISOBUS Virtual Terminal (VT) Systeme entwickelt
- **Strukturierte Konstanten**: Verwendung eines strukturierten Datentyps mit Code-Werten und zusätzlichen Flags

Zustandsübersicht

Konstante	Code	Beschreibung
BT_RELEASED_UNLATCHED	0	Taste wurde entriegelt oder losgelassen (Zustandsänderung)
BT_PRESSED_LATCHED	1	Taste wurde „gedrückt“ oder verriegelt (Zustandsänderung)
BT_STILL_HELD	2	Taste wird weiterhin gehalten (verriegelbare Tasten wiederholen sich nicht)
BT_STILL_HELD_START	2	Taste wird weiterhin gehalten (mit bCyclic := FALSE)

BT_ABORTED	3	Tastendruck abgebrochen (gilt für VT Version 4 und höher)
Invalid	255	Ungültiger Zustand

Anwendungsszenarien
- Steuerung von landwirtschaftlichen Maschinen über ISOBUS Virtual Terminal
- Implementierung von Tastenlogik in Bedienpanels
- Verarbeitung von Tasteneingaben in Fahrzeugsteuerungssystemen
- Entwicklung von Benutzerschnittstellen für landwirtschaftliche Anwendungen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Eingängen bietet ButtonActivationCode eine fein granulierte Zustandsdarstellung für Tasten, die speziell auf die Anforderungen von ISOBUS-Systemen zugeschnitten ist. Die Konstanten unterstützen erweiterte Tastenfunktionen wie Verriegelungsmechanismen und Abbruchfunktionen.

Fazit

ButtonActivationCode bietet eine standardisierte Methode zur Behandlung von Tastenaktivierungen in ISOBUS-Umgebungen. Durch die definierten Konstanten wird eine konsistente und zuverlässige Verarbeitung von Tasteneingaben über verschiedene Systeme hinweg ermöglicht, was besonders in sicherheitskritischen landwirtschaftlichen Anwendungen von Bedeutung ist.

ButtonActivationCode_S

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Datentyp `ButtonActivationCode_S` ist eine strukturierte Datenstruktur (Struct), die zur Konfiguration der Aktivierung eines Tasters (Buttons) in einem Steuerungssystem verwendet wird. Er definiert einen Aktivierungscode und ein Steuerungsflag für die Wiederholungslogik. Dieser Datentyp ist Teil des Pakets `isobus::UT::io::Button`.

Schnittstellenstruktur

Da es sich um einen Datentyp und nicht um einen Funktionsblock handelt, gibt es keine Ereignis- oder Adapterschnittstellen. Die Struktur besteht ausschließlich aus Datenfeldern.

Daten-Eingänge

(Nicht anwendbar - Dies ist ein Datentyp, kein Funktionsblock)

Daten-Ausgänge

(Nicht anwendbar - Dies ist ein Datentyp, kein Funktionsblock)

Strukturierte Datenfelder

Die folgenden Felder bilden die interne Struktur des Datentyps `ButtonActivationCode_S`:

Name	Datentyp	Initialwert	Kommentar
`code`	`USINT`	255	Definiert den spezifischen Aktivierungscode für den Taster.
`bCyclic`	`BOOL`	TRUE	Steuert das zyklische Verhalten. Wenn auf `FALSE` gesetzt, wird die Aktivierung nicht alle 200 ms wiederholt.

Funktionsweise

Die Struktur `ButtonActivationCode_S` dient als Container für Konfigurationsdaten.

- Das Feld `code` identifiziert den Taster eindeutig innerhalb des Systems. Der Standardwert 255 kann als inaktiver oder undefinierter Zustand interpretiert werden.
- Das Feld `bCyclic` bestimmt das Zeitverhalten bei einer dauerhaften Betätigung des Tasters. Bei `TRUE` (Standard) wird das zugehörige Ereignis in einem festen Intervall (z.B. alle 200 ms) wiederholt ausgelöst.

Bei `FALSE` wird das Ereignis nur einmalig beim Drücken (und ggf. beim Loslassen) ausgelöst, auch wenn der Taster gedrückt gehalten wird.

Technische Besonderheiten

- **Initialwerte:** Beide Felder der Struktur haben vordefinierte Initialwerte, was eine sichere Initialisierung gewährleistet.
- **Kommentar:** Das Feld `bCyclic` enthält einen hilfreichen Kommentar, der die konkrete Auswirkung (`do not repeat every 200ms`) direkt in der Deklaration erklärt.
- **Typ-Hash:** Der Datentyp enthält ein Attribut (`eclipse4diac::core::TypeHash`), das für die Typprüfung und Versionierung innerhalb der 4diac-Umgebung genutzt wird.

Zustandsübersicht

(Nicht anwendbar - Dies ist ein statischer Datentyp ohne Zustandsautomaten.)

Anwendungsszenarien

Dieser Struct wird typischerweise als Eingangs- oder Konfigurationsparameter für Funktionsblöcke verwendet, die Tastereingänge verarbeiten, z.B.:

- In einer Bedienoberfläche (HMI) zur Konfiguration von Softkeys.
- In Maschinensteuerungen zur Parametrierung von physischen Steuertastern.
- Überall dort, wo zwischen einmaliger und wiederholter Aktion bei Tastendruck unterschieden werden muss.

⚖️ Vergleich mit ähnlichen Bausteinen

Als reiner Datentyp ist `ButtonActivationCode_S` mit anderen strukturierten Datentypen (STRUCTs) in 4diac vergleichbar, wie z.B. `DATE_AND_TIME`. Sein spezifischer Zweck ist die Bündelung der für die Tastersteuerung relevanten Parameter (`code` für Identifikation, `bCyclic` für Zeitverhalten) in einer einzigen, wiederverwendbaren Einheit.

Fazit

Der Datentyp `ButtonActivationCode_S` ist eine einfache, aber wesentliche Struktur zur standardisierten Konfiguration von Tasterverhalten in 4diac-basierten Steuerungsapplikationen. Durch die Kapselung des Codes und des Wiederholungsflags fördert er die Wiederverwendbarkeit und Lesbarkeit von Code. Die integrierten Initialwerte und der klare Kommentar tragen zur Robustheit und einfachen Handhabung bei.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

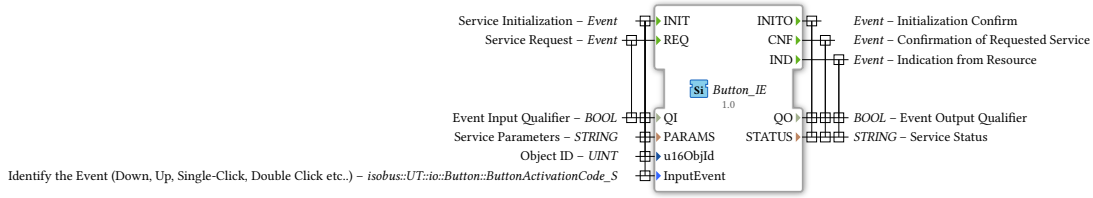
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Button_IE

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Button_IE Funktionsblock ist ein Eingabeservice-Interface-Funktionsblock für Ereigniseingabedaten. Er dient als Schnittstelle für Taster-Ereignisse in Steuerungssystemen und ermöglicht die Verarbeitung verschiedener Tasteraktivitäten wie Drücken, Loslassen oder Mehrfachklicks.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung
- **REQ**: Service-Anfrage

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
- **CNF**: Bestätigung der angeforderten Service-Anfrage
- **IND**: Indikation von der Ressource

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangsqualifizierer
- **PARAMS** (STRING): Service-Parameter
- **u16ObjId** (UINT): Objekt-ID (Initialwert: ID_NULL)
- **InputEvent** (ButtonActivationCode_S): Identifiziert das Ereignis (Down, Up, Single-Click, Double-Click etc.) mit Initialwert „Invalid“

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifizierer
- **STATUS** (STRING): Service-Status

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Button_IE Funktionsblock verarbeitet Tasterereignisse über das ISOBUS-UT-Protokoll. Bei Initialisierung (INIT) werden die Service-Parameter und Objekt-ID konfiguriert. Der Block kann verschiedene Tasteraktivitäten wie Einfachklick, Doppelklick, Drücken und Loslassen erkennen und entsprechend verarbeiten. Über die IND-Ausgabe werden eingehende Ereignisse von der Hardware-Ressource gemeldet.

Technische Besonderheiten

- Verwendet ISOBUS-UT-spezifische Datentypen für Tasteraktivierungscodes
- Unterstützt Initialisierung mit spezifischen Objekt-IDs
- Bietet umfassende Statusrückmeldungen über den STATUS-Ausgang
- Implementiert qualifizierte Ereignisverarbeitung über QI/QO-Signale

Zustandsübersicht

Der Funktionsblock verfügt über einen Initialisierungszustand (INIT/INITO) und operative Zustände für Service-Anfragen (REQ/CNF) sowie asynchrone Ereignisindikationen (IND). Die genaue Zustandsmaschine ist implementierungsabhängig.

Anwendungsszenarien

- Landwirtschaftliche Steuerungssysteme mit ISOBUS-Kompatibilität
- Bedienpanels mit Tastereingaben
- Maschinensteuerungen mit Ereignis-basierten Eingaben
- Systeme, die verschiedene Tasteraktivitäten unterscheiden müssen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Eingangsblöcken bietet Button_IE erweiterte Funktionalität für Tasterspezifische Ereignisse wie Mehrfachklicks und unterscheidet zwischen verschiedenen Aktivierungszuständen. Die ISOBUS-Integration macht ihn speziell für landwirtschaftliche Anwendungen geeignet.

🛠️ Zugehörige Übungen

- Uebung_010b7
- Uebung_010b7_AX
- Uebung_010b8
- Uebung_010b8_AX
- Uebung_010b9
- Uebung_010b9_AX
- Uebung_010bA
- Uebung_010bA_AX

Fazit

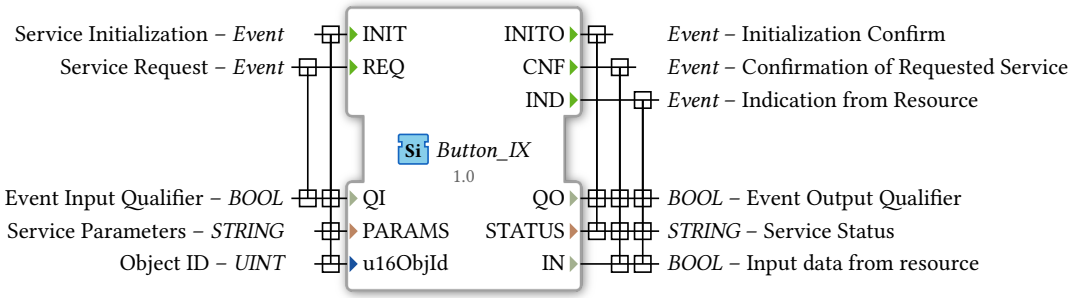
Button_IE ist ein spezialisierter Funktionsblock für die Verarbeitung von Tasterereignissen in ISOBUS-Umgebungen. Seine Fähigkeit, verschiedene Tasteraktivitäten zu unterscheiden und umfassende Statusinformationen bereitzustellen, macht ihn besonders geeignet für anspruchsvolle Steuerungsanwendungen in der Agrartechnik und verwandten Bereichen.

Button_IX

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Button_IX ist ein Service-Interface-Funktionsblock für boolesche Eingabedaten, der speziell für die Verarbeitung von Tasteneingaben in 4diac-Systemen entwickelt wurde. Dieser Baustein dient als Schnittstelle zwischen der Anwendungslogik und physikalischen Eingabegeräten wie Tastern oder Schaltern.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierungsereignis
- **REQ**: Service-Anfrageereignis

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
- **CNF**: Bestätigung der angeforderten Service-Anfrage
- **IND**: Indikation vom Ressourcen-Interface

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangsqualifizierer
- **PARAMS** (STRING): Service-Parameter
- **u16ObjId** (UINT): Objekt-ID mit Initialwert ID_NULL

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifizierer
- **STATUS** (STRING): Service-Statusinformation
- **IN** (BOOL): Eingabedaten von der Ressource

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Button_IX-Funktionsblock verwaltet den gesamten Lebenszyklus einer Tasteneingabe. Bei der Initialisierung (INIT-Ereignis) werden die Service-Parameter konfiguriert und das System für den Betrieb vorbereitet. Das REQ-Ereignis löst eine aktuelle Abfrage des Eingabezustands aus, während das IND-Ereignis asynchrone Zustandsänderungen der Tasteneingabe signalisiert.

Technische Besonderheiten

- Unterstützt ISOBUS-konforme Kommunikation
- Verwendet spezifische TypeHash-Identifikation für 4diac
- Integriert Objekt-ID-Verwaltung für Geräteidentifikation

- Bietet umfangreiche Statusrückmeldungen über den STATUS-Ausgang

Zustandsübersicht

Der Funktionsblock durchläuft folgende Hauptzustände:

1. **Nicht initialisiert**: Wartet auf INIT-Ereignis
2. **Initialisiert**: Bereit für REQ- und IND-Ereignisse
3. **Aktiv**: Verarbeitet Eingabeanfragen und -indikationen
4. **Fehler**: Bei ungültigen Parametern oder Kommunikationsproblemen

Anwendungsszenarien

- Bedienpulte in industriellen Steuerungen
- Maschinensteuerung mit Tastereingaben
- ISOBUS-kompatible Landmaschinensteuerungen
- Sicherheitskritische Eingabesysteme mit Statusüberwachung

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Eingabeblöcken bietet Button_IX erweiterte Funktionalitäten wie:

- Parametrierbare Service-Konfiguration
- Detaillierte Statusrückmeldungen
- ISOBUS-Integration
- Asynchrone Ereignisbehandlung

🛠️ Zugehörige Übungen

- Uebung_010a2
- Uebung_010a3

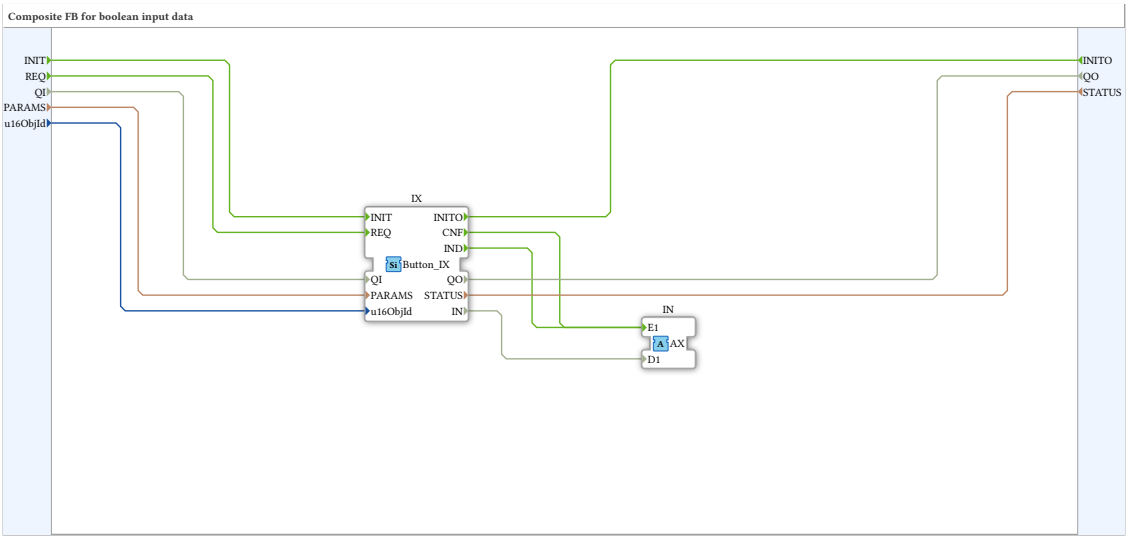
Fazit

Der Button_IX-Funktionsblock stellt eine robuste und flexible Lösung für die Integration von Tasteneingaben in 4diac-basierten Steuerungssystemen dar. Seine umfangreiche Schnittstelle und Statusverwaltung machen ihn besonders geeignet für anspruchsvolle industrielle Anwendungen, die zuverlässige und überwachbare Eingabeverarbeitung erfordern.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

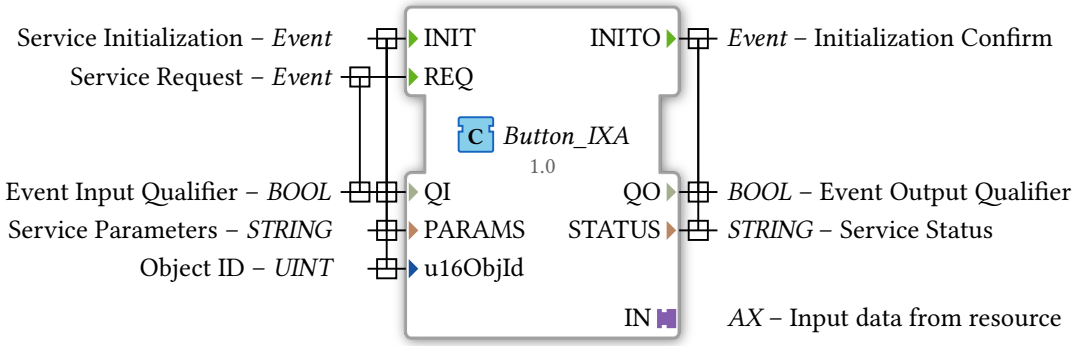
Button_IXA



• – ○ • – ○ • – ○ •

Einleitung

Button_IXA ist ein zusammengesetzter Funktionsblock zur Verarbeitung von booleschen Eingabedaten. Der Baustein dient als Schnittstelle für Taster-Eingaben und bietet eine standardisierte Verarbeitung von Schaltzuständen in Automatisierungssystemen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung - wird mit QI, PARAMS und u16ObjId ausgelöst
- **REQ**: Service-Anfrage - wird mit QI ausgelöst

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung - wird mit QO und STATUS ausgelöst

Daten-Eingänge

- **QI**: Ereignis-Eingangsqualifizierer (BOOL)
- **PARAMS**: Service-Parameter (STRING)
- **u16ObjId**: Objekt-ID (UINT) - Initialwert: ID_NULL

Daten-Ausgänge

- **QO**: Ereignis-Ausgangsqualifizierer (BOOL)
- **STATUS**: Service-Status (STRING)

Adapter

- **IN**: Unidirektionaler Adapter vom Typ adapter::types::unidirectional::AX

Funktionsweise

Der Button_IXA Funktionsblock fungiert als Wrapper um den internen Button_IX Baustein. Bei der Initialisierung (INIT) werden die Konfigurationsparameter übergeben und der interne Baustein konfiguriert. Service-Anfragen (REQ) lösen die Abfrage des Tasterzustands aus. Der Adapter IN dient als Schnittstelle für die eigentliche Taster-Eingabe.

Technische Besonderheiten

- Verwendet ISOBUS-konforme Typen und Adapter
- Unterstützt Objekt-Identifikation über u16ObjId
- Flexible Parametrierung über STRING-Parameter
- Unidirektionale Adapter-Schnittstelle für Eingabedaten

Zustandsübersicht

Der Baustein verfügt über folgende Hauptzustände:

- Nicht initialisiert: Wartet auf INIT-Ereignis
- Initialisiert: Bereit für REQ-Anfragen
- Aktiv: Verarbeitet Taster-Eingaben über den Adapter

Anwendungsszenarien

- Taster-Eingaben in landwirtschaftlichen Maschinen

- ISOBUS-kompatible Steuerungssysteme
- Boolesche Eingabeverarbeitung in verteilten Automatisierungssystemen
- Bedienfelder und Schaltpulte

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen booleschen Eingabeblöcken bietet Button_IXA:

- ISOBUS-Kompatibilität
- Erweiterte Parametrierungsmöglichkeiten
- Standardisierte Statusrückmeldungen
- Adapter-basierte Architektur für bessere Wiederverwendbarkeit

🛠️ Zugehörige Übungen

- Uebung_010a2_AX
- Uebung_010a3_AX

Fazit

Button_IXA ist ein spezialisierter Funktionsblock für die zuverlässige Verarbeitung von Taster-Eingaben in ISOBUS-Umgebungen. Durch seine modulare Architektur und standardisierte Schnittstellen eignet er sich ideal für den Einsatz in komplexen Automatisierungssystemen der Agrartechnik.

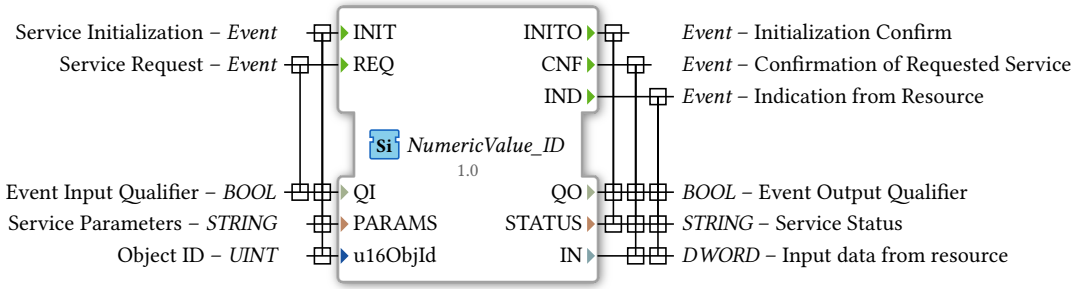
NumericValue

NumericValue_ID

• – ◦ • – ◦ • – ◦ •

Einleitung

Der NumericValue_ID Funktionsblock ist ein Eingabeservice-Interface-Funktionsblock für Doppelwort-Eingabedaten (DWORD). Er dient als Schnittstelle zur Kommunikation mit Ressourcen und ermöglicht die Verarbeitung von numerischen Eingabewerten in ISOBUS-kompatiblen Systemen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung
  - Verknüpft mit: QI, PARAMS, u16ObjId
- **REQ**: Service-Anfrage
  - Verknüpft mit: QI

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
  - Verknüpft mit: QO, STATUS
- **CNF**: Bestätigung der angeforderten Service-Anfrage
  - Verknüpft mit: QO, STATUS, IN
- **IND**: Indikation von der Ressource
  - Verknüpft mit: QO, STATUS, IN

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangsqualifizierer

- **PARAMS** (STRING): Service-Parameter
- **u16ObjId** (UINT): Objekt-ID mit Initialwert ID_NULL

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifizierer
- **STATUS** (STRING): Service-Status
- **IN** (DWORD): Eingabedaten von der Ressource

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Funktionsblock initialisiert sich über das INIT-Ereignis und kann über REQ Service-Anfragen stellen. Bei erfolgreicher Initialisierung gibt er INITO zurück, während CNF und IND die Bestätigung von Service-Anfragen bzw. Indikationen von der Ressource darstellen. Die Datenübertragung erfolgt über DWORD-Werte (32-Bit), die über den IN-Ausgang bereitgestellt werden.

Technische Besonderheiten

- Verwendet DWORD-Datentyp für numerische Eingabewerte
- ISOBUS-kompatible Implementierung
- Initialisierung mit standardisierter Objekt-ID (ID_NULL)
- String-basierte Statusrückmeldungen
- Unterstützt parametrierbare Service-Konfiguration

Zustandsübersicht

Der Funktionsblock durchläuft folgende Zustände:

1. **Nicht initialisiert**: Vor INIT-Aufruf
2. **Initialisiert**: Nach erfolgreicher INIT-Verarbeitung
3. **Service-aktiv**: Bei aktiven REQ-Anfragen
4. **Datenbereitstellung**: Bei aktiver IND-Indikation

Anwendungsszenarien

- Landwirtschaftliche Steuerungssysteme (ISOBUS)
- Industrielle Automatisierung mit numerischen Sensordaten
- Embedded Systems mit Doppelwort-Datenverarbeitung
- Systeme mit standardisierten Objekt-Identifikationen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen Eingabeblöcken bietet NumericValue_ID erweiterte Service-Funktionalität mit Initialisierungsprozeduren, Statusrückmeldungen und parametrierbaren Konfigurationsmöglichkeiten. Die spezielle Ausrichtung auf ISOBUS-Umgebungen unterscheidet ihn von generischen Eingabeblöcken.

🛠️ Zugehörige Übungen

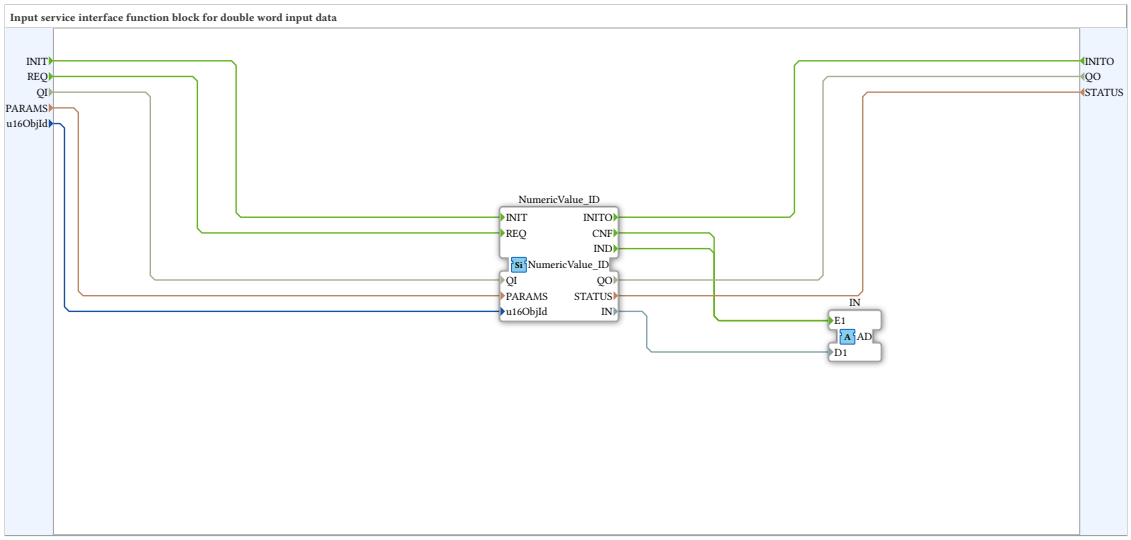
- Uebung_011
- Uebung_012
- NumericValue_ID_TO_NVS
- Uebung_012b
- Uebung_012c
- Uebung_020c2_sub
- Uebung_034a1_Q1
- Uebung_034a1_Q2
- Uebung_034a1_Q4
- Uebung_103

Fazit

Der NumericValue_ID Funktionsblock stellt eine robuste Lösung für die Verarbeitung von Doppelwort-Eingabedaten in ISOBUS-basierten Systemen dar. Durch seine strukturierte Initialisierung, klare Status-

rückmeldungen und flexible Parametrierung eignet er sich ideal für industrielle Anwendungen mit hohen Anforderungen an Zuverlässigkeit und Diagnosefähigkeit.

NumericValue_IDA

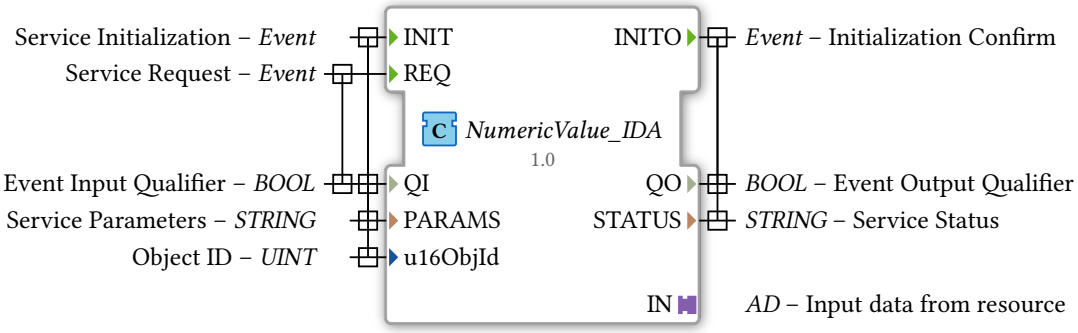


• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **NumericValue_IDA** ist ein Service-Schnittstellen-Baustein (Service Interface Function Block), der speziell für die Verarbeitung von numerischen Eingabedaten (Double Word Input Data) im Kontext von ISOBUS-Universal-Terminals (UT) entwickelt wurde. Er dient dazu, Änderungen von numerischen Werten, die auf einer Benutzeroberfläche (z. B. einem Traktor-Terminal) vorgenommen werden, in die Steuerungslogik zu übernehmen.

Im Gegensatz zu Standard-Eingabebausteinen stellt dieser Baustein die empfangenen Daten nicht über direkte Datenausgänge, sondern über einen Adapter-Anschluss (`IN`) zur Verfügung. Er kapselt die Funktionalität des Bausteins `NumericValue_ID` und leitet dessen Signale auf eine Adapter-Schnittstelle um.



Schnittstellenstruktur

Der Baustein verfügt über Standard-Initialisierungs- und Anforderungs-Ereignisse sowie spezifische Konfigurationsparameter. Die eigentliche Datenübertragung erfolgt über den Adapter.

Ereignis-Eingänge

- **INIT** (Typ: `EInit`): Service-Initialisierung. Dient zum Starten und Konfigurieren des Bausteins.
- **REQ** (Typ: `Event`): Service-Anforderung. Kann genutzt werden, um den Status zu aktualisieren oder eine spezifische Interaktion anzufordern (abhängig von der internen Implementierung des Service-Layers).

Ereignis-Ausgänge

- **INITO** (Typ: `EInit`): Bestätigung der Initialisierung. Zeigt an, ob der Baustein erfolgreich konfiguriert wurde.

Daten-Eingänge

- **QI** (Typ: `BOOL`): Event Input Qualifier. Aktiviert die Ausführung der mit den Ereignissen verknüpften Aktionen.
- **PARAMS** (Typ: `STRING`): Service-Parameter für spezifische Konfigurationen des darunterliegenden Service-Layers.
- **u16ObjId** (Typ: `UINT`): Objekt-ID. Referenziert das spezifische numerische Eingabeobjekt auf dem Universal Terminal (UT), dessen Wert überwacht werden soll. Der Standardwert ist `ID_NULL`.

Daten-Ausgänge

- **QO** (Typ: `BOOL`): Event Output Qualifier. Zeigt an, ob die letzte Operation (z. B. Initialisierung) erfolgreich war.
- **STATUS** (Typ: `STRING`): Service-Status. Enthält Fehlercodes oder Statusmeldungen zur Diagnose.

Adapter

- **IN** (Typ: `adapter::types::unidirectional::AD`):
  - Dies ist ein unidirektionaler Adapter-Plug („Input data from resource“).
  - Er bündelt das Ereignissignal (E1) und das Datensignal (D1), welches den aktuellen numerischen Wert vom Terminal enthält.
  - Der Adapter wird getriggert, wenn der interne `NumericValue_ID` Baustein ein `IND` (Indication) oder `CNF` (Confirmation) Ereignis sendet.

Funktionsweise

Der **NumericValue_IDA** fungiert als Wrapper für den internen Baustein `NumericValue_ID`.

1. **Initialisierung:** Über das Ereignis `INIT` und die Daten-Eingänge `PARAMS` und `u16ObjId` wird die Verbindung zum spezifischen Objekt auf dem ISOBUS-Terminal hergestellt.
2. **Datenempfang:** Wenn sich der Wert des referenzierten Objekts auf dem Terminal ändert oder vom System aktualisiert wird, empfängt der interne Baustein diese Informationen.
3. **Weiterleitung:** Der interne Baustein leitet den neuen Wert (`IN`) und das zugehörige Ereignis an den Adapter `IN` des **NumericValue_IDA** weiter.
   - Das Ereignis `E1` des Adapters wird ausgelöst.
   - Der Datenpunkt `D1` des Adapters stellt den numerischen Wert zur Verfügung.

Dieser Mechanismus ermöglicht eine saubere Kapselung der Kommunikation, bei der die Anwendungslogik lediglich den Adapter `IN` verbinden muss, um auf Wertänderungen zu reagieren.

Technische Besonderheiten

- **Komposition:** Der Baustein ist als zusammengesetzter Funktionsbaustein (Composite FB) implementiert, der Instanzen aus dem Paket `isobus::UT::io::NumericValue` nutzt.
- **Adapter-Nutzung:** Die Besonderheit liegt in der Verwendung des Adapters `AD` zur Datenausgabe. Dies reduziert die visuelle Komplexität im Funktionsplan, da Daten- und Ereignisfluss in einer einzigen Verbindungslinie gebündelt werden.
- **ISOBUS-Kontext:** Der Baustein ist explizit für die Verwendung im ISOBUS-Stack (ISO 11783) vorgesehen, erkennbar an der Verwendung von `u16ObjId` zur Identifikation von UT-Objekten.

Zustandsübersicht

Da es sich um einen Composite FB handelt, wird der Zustand primär durch die internen Service-Bausteine bestimmt.

- **Nicht Initialisiert:** Nach dem Start oder bei `QI=FALSE` während `INIT`.
- **Initialisiert / Bereit:** Nach erfolgreichem `INIT` (`QO=TRUE`). Der Baustein lauscht auf Updates vom Terminal.
- **Fehler:** Wenn `QO=FALSE` und `STATUS` eine Fehlermeldung enthält (z. B. ungültige Objekt-ID).

Anwendungsszenarien

- **Sollwertvorgabe:** Ein Bediener gibt auf dem Terminal eine Menge (z. B. Ausbringmenge in Litern) oder eine Zielgeschwindigkeit ein. Dieser Baustein empfängt den Wert.
- **Parametrierung:** Einstellung von Maschinenparametern über numerische Felder auf dem Display.
- **Modulare Anwendungen:** Einsatz in Anwendungen, die stark auf Adapter-Verbindungen setzen, um den „Spaghetti-Code“ von parallelen Event- und Datenleitungen zu vermeiden.

⚖️ Vergleich mit ähnlichen Bausteinen

- **NumericValue_ID:** Dies ist der interne Basisbaustein. Er bietet dieselbe Funktionalität, stellt die Daten jedoch über klassische `IND`/`CNF` Ereignisse und einen `IN` Datenausgang bereit. **NumericValue_IDA** ist die Adapter-Variante dieses Bausteins.
- **Andere `_IDA` Bausteine:** Ähnlich wie `Button_IDA` (für Schaltflächen) oder `String_IDA` (für Texte), folgt dieser Baustein dem Entwurfsmuster, Terminal-Eingaben über Adapter bereitzustellen.

🛠️ Zugehörige Übungen

- Uebung_011_AUDI

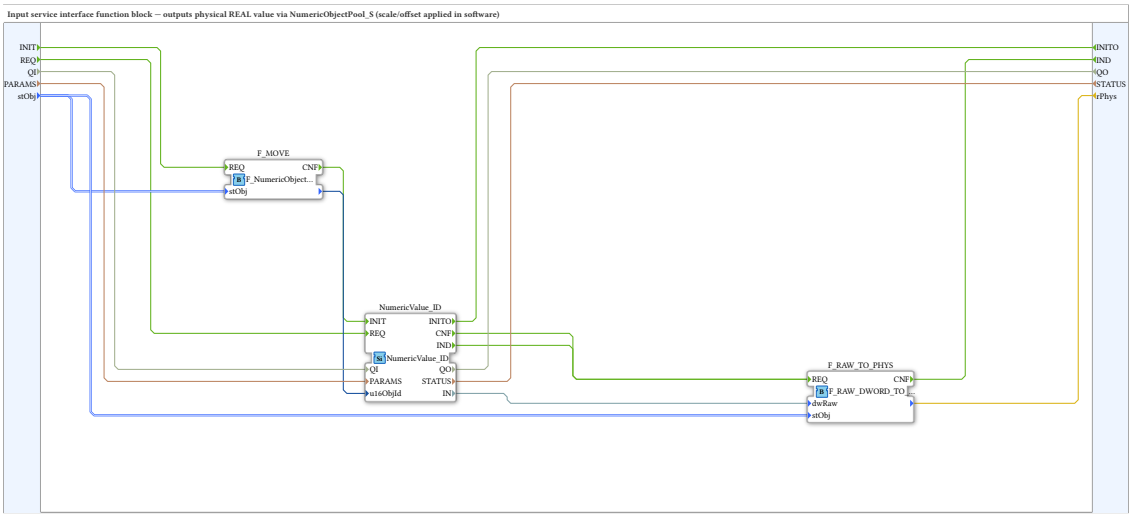
Fazit

Der **NumericValue_IDA** ist ein spezialisierter Schnittstellenbaustein für ISOBUS-Anwendungen in 4diac. Durch die Verwendung eines Adapters für die Datenausgabe ermöglicht er eine kompakte und übersichtliche Verschaltung von numerischen Eingabewerten im Funktionsplan. Er eignet sich ideal für Entwickler, die eine strukturierte und modulare Anwendungsarchitektur bevorzugen und numerische Eingaben vom Universal Terminal verarbeiten müssen.

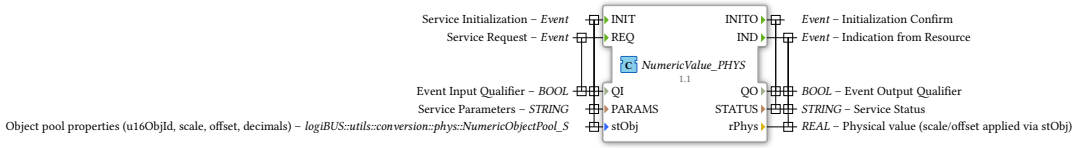
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

NumericValue_PHYS



(Bild nicht vorhanden)



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock **NumericValue_PHYS** ist ein Eingangs-Service-Interface-Baustein nach ISO 11783-6. Er liefert einen physikalischen REAL-Wert, indem er einen rohen Digitalwert (DWORD) aus dem ISOBUS-Objektpool ausliest und unter Berücksichtigung einer vorgegebenen Skalierung und eines Offsets in einen physikalischen Wert umrechnet. Dabei wird die Konvertierung vollständig in Software (innerhalb des FBs) durchgeführt.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Mit Variablen	Kommentar
INIT	EInit	QI, PARAMS, stObj	Initialisierung des Bausteins
REQ	Event	QI	Anforderung eines neuen physikalischen Wertes

Ereignis-Ausgänge

Ereignis	Typ	Mit Variablen	Kommentar
INITO	EInit	QO, STATUS	Bestätigung der Initialisierung
IND	Event	QO, STATUS, rPhys	Ausgabe des berechneten physikalischen Wertes

Daten-Eingänge

Variable	Typ	Kommentar
QI	BOOL	Eingangsqualifikator (aktiviert die Verarbeitung)
PARAMS	STRING	Service-Parameter (z. B. Konfigurationszeichenfolge)
stObj	logiBUS::utils::conversion::phys::NumericObjectPool_S	Objektpool-Eigenschaften: Objekt-ID (16 Bit), Skalierung, Offset, Dezimalstellen

Daten-Ausgänge

Variable	Typ	Kommentar
QO	BOOL	Ausgangsqualifikator (Status der Verarbeitung)
STATUS	STRING	Statusmeldung (Fehler- oder Erfolgsmeldung)
rPhys	REAL	Physikalischer Wert nach Anwendung von Skalierung/Offset

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der interne Ablauf wird über die Ereignisse INIT und REQ gesteuert und nutzt vier Sub-FBs:

1. **INIT**
   - Der übergebene Strukturparameter `stObj` wird über den Sub-Baustein `F_MOVE` (vom Typ `iec61131::selection::F_MOVE`) kopiert.
   - Der kopierte Wert (`stObj.u16ObjId`) wird an den Sub-Baustein `NumericValue_ID` weitergeleitet, der damit initialisiert wird (`NumericValue_ID.INIT`).
2. **REQ** (oder erneute Ausgabe nach INIT)
   - Der Sub-Baustein `NumericValue_ID` wird mit `REQ` ausgelöst. Er liefert über seinen Ausgang `IN` einen rohen DWORD-Wert aus dem ISOBUS-Objektpool.
   - Dieser DWORD-Wert wird über `F_DWORD_TO_UDINT` in einen vorzeichenlosen 32-Bit-Integer (`UDINT`) umgewandelt.
   - Der Sub-Baustein `F_RAW_TO_PHYS` erhält den Integer-Wert und die Original-Struktur `stObj` (Skalierung, Offset, Dezimalstellen) und berechnet daraus den physikalischen REAL-Wert (`rPhys`).
   - Schließlich wird der Ausgang `IND` aktiviert und der berechnete Wert am Datenausgang `rPhys` ausgegeben.

Die Verkettung gewährleistet, dass bei jedem REQ der aktuelle rohe Wert aus dem Objektpool gelesen und kalkuliert wird.

Technische Besonderheiten

- **Standardkonformität**: Der Baustein folgt dem ISO-11783-6-Standard (ISOBUS).
- **Software-Skalierung**: Im Gegensatz zu reinen Hardware-Interfaces wird die Umrechnung (Skalierung/Offset) innerhalb des FBs durchgeführt, was eine flexible Anpassung ohne Änderung der Peripherie erlaubt.
- **Wiederverwendbare Sub-Bausteine**: Die verwendeten Sub-FBs (`NumericValue_ID`, `F_DWORD_TO_UDINT`, `F_RAW_TO_PHYS`, `F_MOVE`) sind standardisierte logiBUS- oder IEC-61131-Bausteine und können in anderen Zusammenhängen eingesetzt werden.
- **Objektpool-Struktur**: Die Eingabestruktur `stObj` enthält alle notwendigen Parameter (ObjektID, Skalierung, Offset, Dezimalstellen) und kann zentral verwaltet werden.

Zustandsübersicht

Der Baustein besitzt keine explizite Zustandsmaschine, sondern reagiert rein ereignisgesteuert. Der Ablauf gliedert sich in zwei Hauptphasen:

- **Initialisierungsphase** (Ereignis `INIT` ⟶ `INITO`):
  Übernahme und Speicherung der Konfiguration (`stObj`), Initialisierung von `NumericValue_ID`.
- **Betriebsphase** (Ereignis `REQ` ⟶ `IND`):
  Auslesen des Rohwertes, Konvertierung und Ausgabe des physikalischen Wertes.

Nach einer erfolgreichen Initialisierung kann `REQ` beliebig oft wiederholt werden.

Anwendungsszenarien

- **ISOBUS-Fahrzeugsteuerung**: Einlesen von Sensordaten (z. B. Drehzahl, Druck, Temperatur) aus dem ISOBUS-Objektpool und Umwandlung in physikalische Einheiten.
- **Landwirtschaftliche Automatisierung**: Verarbeitung von Messwerten von ISOBUS-kompatiblen Geräten (Traktoren, Anbaugeräte) zur weiteren Steuerung oder Visualisierung.
- **Test- und Simulationsumgebungen**: Ersatz realer Hardware-Bausteine durch Software-Emulation mit definierten Skalierungen.

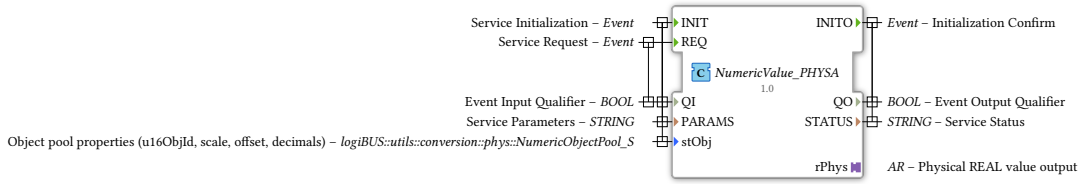
Vergleich mit ähnlichen Bausteinen

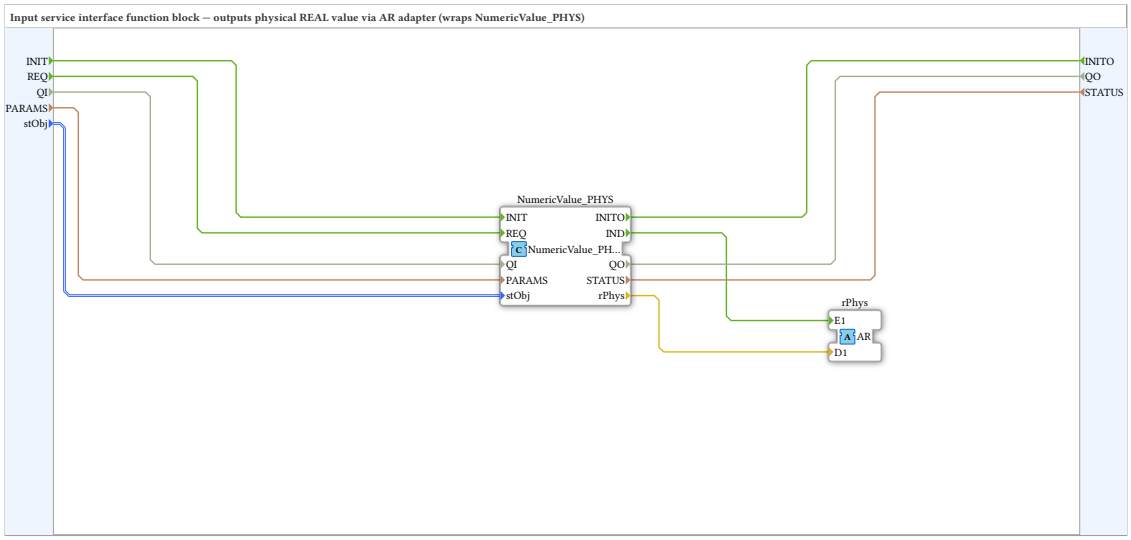
- **NumericValue_ID**: Liefert nur den rohen DWORD-Wert ohne Skalierung/Offset. `NumericValue_PHYS` erweitert diesen um die physikalische Umrechnung.
- **Analoge Eingangsbausteine** (z. B. `AI_SCALED`) in SPS-Systemen: Diese skalieren oft direkt im Hardwaretreiber. `NumericValue_PHYS` hingegen arbeitet rein softwarebasiert und ist dadurch flexibler hinsichtlich Parametrierung.
- **logiBUS-Bausteine wie `F_RAW_TO_PHYS`**: Werden von `NumericValue_PHYS` intern genutzt; dieser FB kapselt die gesamte Kette von der ID-Abfrage bis zur finalen Ausgabe.

Fazit

Der Funktionsblock `NumericValue_PHYS` bietet eine saubere und standardkonforme Möglichkeit, ISOBUS-Objekte mit integrierter Skalierung und Offset-Korrektur in physikalische Werte umzuwandeln. Durch die Nutzung modularer Sub-Bausteine und die Parametrierung über eine Strukturvariable ist er wartungsfreundlich und wiederverwendbar. Er eignet sich insbesondere für Anwendungen, bei denen softwaregesteuerte Umrechnungen gefordert sind, ohne die Hardware-Konfiguration zu beeinflussen.

NumericValue_PHYSA





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **NumericValue_PHYSA** ist ein Eingangs-Service-Interface-Funktionsblock (SIFB) nach ISO 11783-6. Seine Aufgabe besteht darin, einen physikalischen REAL-Wert über einen AR-Adapter (Adapter Typ *unidirectional::AR*) bereitzustellen. Intern kapselt er den Baustein `NumericValue_PHYS` und dessen gesamte Logik zur Umrechnung und Bereitstellung des Wertes.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung
INIT	EInit	Initialisiert den Baustein (mit Parametern und Objekt-Pool)
REQ	Event	Löst die Ausgabe des aktuellen physikalischen Wertes aus

Ereignis-Ausgänge

Name	Typ	Beschreibung
INITO	EInit	Bestätigung der erfolgreichen Initialisierung

Daten-Eingänge

Name	Typ	Beschreibung
QI	BOOL	Eingangsqualifizierer (aktiviert/steuert die Verarbeitung)
PARAMS	STRING	Parameter für den Dienst (z. B. Konfigurationsstring)
stObj	*NumericObjectPool_S*	Objekt-Pool-Eigenschaften: Objekt-ID (UINT16), Skalierung, Offset, Dezimalstellen

Daten-Ausgänge

Name	Typ	Beschreibung
QO	BOOL	Ausgangsqualifizierer (zeigt erfolgreiche Verarbeitung an)
STATUS	STRING	Statusmeldung (OK oder Fehlertext)

Adapter

Name	Typ	Beschreibung
rPhys	*adapter::types::unidirectional::AR*	Adapter-Schnittstelle zur Ausgabe des physikalischen REAL-Wertes (Ereigniskanal `E1` und Datenkanal `D1`)

Funktionsweise

Der Baustein ist als reiner Wrapper um den internen FB `NumericValue_PHYS` realisiert. Sämtliche Ein- und Ausgänge werden eins-zu-eins auf den inneren Baustein durchverbunden:

- Die Ereignisse `INIT` und `REQ` werden direkt an `NumericValue_PHYS` weitergeleitet.
- Das Initialisierungs-Bestätigungsereignis `INITO` des inneren FB wird nach außen geführt.
- Das Ausgangsereignis `IND` von `NumericValue_PHYS` wird mit dem Ereigniseingang `rPhys.E1` des Adapters verbunden, sodass der Adapter bei jeder neuen Wertberechnung ein Ereignis erhält.
- Die Daten-Eingänge (`QI`, `PARAMS`, `stObj`) werden an den inneren FB verdrahtet.
- Die Daten-Ausgänge (`QO`, `STATUS`) kommen direkt vom inneren FB.
- Der physikalische REAL-Wert aus `NumericValue_PHYS.rPhys` wird auf den Datenkanal `rPhys.D1` des Adapter-Plugs gelegt.

Damit ist der FB ein reiner Konnektor, der die Funktionalität von `NumericValue_PHYS` um eine standardisierte Adapter-Schnittstelle ergänzt, ohne selbst zusätzliche Logik zu implementieren.

Technische Besonderheiten

- **Eingesetzte Datentypen:** Die physikalischen Eigenschaften des Messwerts (Skalierung, Offset, Dezimalstellen) werden über den strukturierten Typ `logiBUS::utils::conversion::phys::NumericObjectPool_S` definiert.
- **Adapter-Kopplung:** Der AR-Adapter (unidirektional) ermöglicht eine lose Kopplung an nachfolgende Bausteine, die den physikalischen Wert konsumieren.
- **Typ-Hash:** Der Baustein trägt ein Attribut `eclipse4diac::core::TypeHash` zur Identifikation der exakten Version.
- **Keine interne Zustandsmaschine:** Der FB delegiert alle Zustandslogik an den gekapselten Baustein.

Zustandsübersicht

Da der FB keine eigene Zustandsmaschine besitzt, ergibt sich der Zustand vollständig aus dem inneren `NumericValue_PHYS`:

1. **Initialisierungsphase:** Nach dem Ereignis `INIT` wird der FB konfiguriert. Erst nach `INITO` ist er betriebsbereit.
2. **Betriebsphase:** Mit jedem `REQ` wird der aktuelle physikalische Wert berechnet und über den Adapter ausgegeben.
3. **Fehlerzustand:** Bei fehlerhafter Initialisierung oder ungültigen Parametern wird `STATUS` mit einem entsprechenden Text belegt und `QO` auf FALSE gesetzt.

Anwendungsszenarien

- **ISOBUS-Steuergeräte:** Der FB eignet sich ideal, um Sensordaten (z. B. Drehzahl, Druck, Temperatur) in ein standardisiertes Adapter-Interface zu wandeln, wie es in landwirtschaftlichen Maschinen gemäß ISO 11783-6 gefordert wird.
- **Messwerterfassung mit Skalierung:** Über die Parameter-Struktur können Rohwerte mit individuellem Offset und Skalierungsfaktor in physikalische Einheiten umgerechnet werden.
- **Modulare Systeme:** Durch die Adapter-Schnittstelle können mehrere Ausgabebausteine einfach an verschiedene Konsumenten (z. B. Visualisierung, Logik) angeschlossen werden.

Vergleich mit ähnlichen Bausteinen

- **NumericValue_PHYS** (ohne „A“-Suffix): Der reine Funktionsblock ohne Adapter-Schnittstelle. Er stellt den physikalischen Wert nur als Datenausgang (`rPhys`) bereit. Der vorliegende FB fügt den Adapter-Plug hinzu, um die Kopplung über einen standardisierten Kanal zu ermöglichen.
- **NumericValue_RAW / NumericValue_SCALED** (hypothetisch): Diese Bausteine würden Roh- bzw. skalierte Werte ohne Bezug zu einem Objekt-Pool liefern. `NumericValue_PHYSA` hingegen nutzt die Struktur `NumericObjectPool_S` mit expliziten Umrechnungsparametern.

Fazit

Der Funktionsblock `NumericValue_PHYSA` bietet eine saubere, adapter-basierte Schnittstelle zur Ausgabe physikalischer REAL-Werte. Durch die Kapselung der gesamten Umrechnungslogik in `NumericValue_PHYS` bleibt er schlank und wartbar. Sein Einsatz empfiehlt sich überall dort, wo nach ISO 11783-6 konforme, physikalische Messwerte in einem modularen Automatisierungssystem bereitgestellt werden müssen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Softkey

🎧 Podcast

- ISO 11783-6: Softkeys und das Virtual Terminal verstehen – Dein Schlüssel zur Landmaschinen-Mechatronik

SoftKeyActivationCode

• – ◦ • – ◦ • – ◦ •

Einleitung

SoftKeyActivationCode ist eine Sammlung von globalen Konstanten für die Verarbeitung von SoftKey-Ereignissen in ISOBUS-kompatiblen Systemen. Diese Konstanten definieren verschiedene Aktivierungszustände von SoftKeys auf virtuellen Terminals und werden zur Steuerung von Benutzerinteraktionen in landwirtschaftlichen Maschinen eingesetzt.

Schnittstellenstruktur

Bei SoftKeyActivationCode handelt es sich um globale Konstanten und nicht um einen Funktionsblock. Daher existieren keine Ereignis- oder Datenschnittstellen im herkömmlichen Sinne.

Konstantendefinitionen

- **SK_RELEASED**: SoftKey wurde losgelassen (Zustandsänderung) - Code: 0
- **SK_PRESSED**: SoftKey wurde gedrückt (Zustandsänderung) - Code: 1
- **SK_STILL_HELD**: SoftKey wird weiterhin gehalten - Code: 2
- **SK_STILL_HELD_START**: SoftKey wird weiterhin gehalten (Start) - Code: 2, bCyclic: FALSE
- **SK_ABORTED**: SoftKey-Druck wurde abgebrochen (gilt für VT Version 4 und höher) - Code: 3
- **Invalid**: Ungültiger Zustand - Code: 255

Funktionsweise

Die Konstanten repräsentieren verschiedene Aktivierungszustände von SoftKeys in ISOBUS-Systemen. Sie werden verwendet, um den Status von Benutzerinteraktionen mit virtuellen Terminals zu kennzeichnen. Die Übertragungswiederholrate beträgt bei SoftKey-Druck/Loslassen und alle 200 ms, wenn ein SoftKey gehalten wird.

Technische Besonderheiten

- Alle Konstanten sind vom Typ `isobus::UT::io::Softkey::SoftKeyActivationCode_S`
- Die Konstanten sind als globale Konstanten definiert und können systemweit verwendet werden
- Die Abort-Funktionalität (SK_ABORTED) ist nur in Virtual Terminal Version 4 und höher verfügbar
- Die Konstante SK_STILL_HELD_START enthält zusätzlich einen booleschen Parameter `bCyclic`

Anwendungsszenarien

- Steuerung von Benutzeroberflächen in landwirtschaftlichen Maschinen
- Verarbeitung von Touch-Eingaben auf ISOBUS-kompatiblen Displays
- Implementierung von SoftKey-Interaktionen in Virtual Terminal Applikationen

- Zustandsüberwachung von Benutzereingaben in Echtzeit-Steuerungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu Funktionsblöcken bieten globale Konstanten keine aktive Funktionalität, sondern dienen lediglich der Vereinheitlichung von Zustandswerten. Sie sind vergleichbar mit ENUM-Definitionen in anderen Programmiersprachen.

Fazit

SoftKeyActivationCode stellt eine standardisierte Sammlung von Konstanten für die SoftKey-Verarbeitung in ISOBUS-Umgebungen bereit. Durch die Verwendung dieser Konstanten wird eine einheitliche und wartbare Implementierung von Benutzerinteraktionen in landwirtschaftlichen Steuerungssystemen ermöglicht.

SoftKeyActivationCode_S

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Datentyp `SoftKeyActivationCode_S` ist eine strukturierte Datenstruktur (Struct), die zur Konfiguration der Aktivierung eines Softkeys in einem ISOBUS-konformen System dient. Er definiert den spezifischen Aktivierungscode und ein Steuerflag für die Wiederholungsfrequenz der Tasteneingabe.

Schnittstellenstruktur

Dies ist ein Datentyp und besitzt daher keine Ereignis- oder Adapterschnittstellen im herkömmlichen Sinne eines Funktionsbausteins. Die Struktur besteht aus den folgenden Datenfeldern:

Daten-Eingänge

(Nicht anwendbar – dies ist ein Datentyp)

Daten-Ausgänge

(Nicht anwendbar – dies ist ein Datentyp)

Datenfelder (Strukturkomponenten)

Name	Datentyp	Initialwert	Kommentar
`code`	`USINT`	255	Definiert den numerischen Code, der den spezifischen Softkey identifiziert.
`bCyclic`	`BOOL`	TRUE	Steuert das zyklische Senden des Tastencodes. Bei `FALSE` wird die Taste nicht alle 200ms wiederholt gesendet.

Funktionsweise

Die Struktur `SoftKeyActivationCode_S` dient als Container für Konfigurationsdaten. Der Wert im Feld `code` korrespondiert mit einer spezifischen Taste auf einem virtuellen oder physischen Terminal. Das Feld `bCyclic` ermöglicht es, das Verhalten bei gedrückt gehaltener Taste zu steuern. Ist es auf `TRUE` gesetzt (Standard), wird der Tastendruck in einem definierten Intervall (z.B. alle 200ms) wiederholt gesendet, solange die Taste gedrückt bleibt. Bei `FALSE` wird nur ein einzelnes Ereignis beim initialen Drücken der Taste ausgelöst.

Technische Besonderheiten

- **Initialwerte:** Beide Felder besitzen vordefinierte Initialwerte. Der Standardwert `255` für `code` kann als „ungültiger“ oder „nicht konfigurierter“ Code interpretiert werden.
- **ISOBUS-Kontext:** Der Datentyp ist Teil des Pakets `isobus::UT::io::Softkey`, was auf seine Verwendung in ISOBUS Virtual Terminal (VT) oder ähnlichen IO-Kontexten hindeutet.
- **Zyklussteuerung:** Die präzise Steuerung der Wiederholungsrate über `bCyclic` ist entscheidend für eine benutzerfreundliche und zuverlässige Tastatureingabe, insbesondere bei langen Datenfeldern oder Listensteuerungen.

Zustandsübersicht

(Nicht anwendbar – dies ist ein passiver Datentyp ohne Zustandsmaschine.)

Anwendungsszenarien

Diese Datenstruktur wird typischerweise als Eingabe- oder Konfigurationsparameter für Funktionsbausteine verwendet, die:

1. Die virtuelle Tastatureingabe eines ISOBUS-Terminals emulieren oder steuern.
2. Softkey-Ereignisse in einer Anwendungslogik verarbeiten (z.B. Menünavigation, Werteingabe).
3. Die Tastenbelegung oder das Tastenverhalten eines Geräts dynamisch konfigurieren.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu einem Funktionsbaustein (FB) ist `SoftKeyActivationCode_S` ein reiner **Datentyp** (Struct). Er enthält keine Algorithmen oder ereignisgesteuerte Logik.

- **Ähnlichkeit zu `E_SR` oder `E_RS`:** Diese sind Funktionsbausteine mit Speicherfunktion und besitzen eine interne Zustandslogik. `SoftKeyActivationCode_S` ist hingegen ein reiner Datenträger.
- **Vergleich mit einfachen Datentypen:** Im Gegensatz zu atomaren Typen wie `INT` oder `BOOL` bündelt dieser Struct zwei logisch zusammengehörige Informationen (`code` und Verhalten `bCyclic`) zu einer einzigen, wiederverwendbaren Einheit.

Fazit

Der Datentyp `SoftKeyActivationCode_S` ist eine essentielle Hilfsstruktur für die Steuerung von Softkey-Eingaben in ISOBUS-Umgebungen. Durch die Kapselung des Tastencodes und des Wiederholungsverhaltens in einem einzigen Struct wird die Konfiguration vereinfacht und die Wiederverwendbarkeit des Codes erhöht. Seine klare Struktur und die sinnvollen Standardwerte machen ihn zu einem robusten Baustein für die Entwicklung von Terminal-IO-Logik.

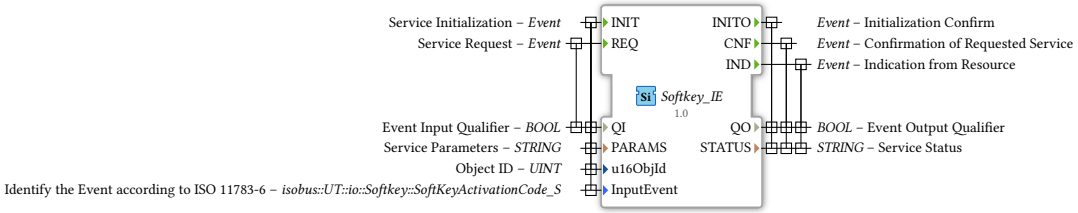
Softkey_IE

🎧 Podcast

- ISO 11783-6: Softkeys und das Virtual Terminal verstehen – Dein Schlüssel zur Landmaschinen-Mechatronik

Einleitung

Der Softkey_IE Funktionsblock ist ein Eingabeservice-Schnittstellen-Funktionsblock für Ereigniseingabedaten, der speziell für die Verarbeitung von Softkey-Ereignissen gemäß ISO 11783-6 entwickelt wurde. Er dient als Schnittstelle zwischen der Anwendungslogik und den physikalischen Softkey-Eingabegeräten in landwirtschaftlichen und mobilen Arbeitsmaschinen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung mit Parametern QI, PARAMS, u16ObjId und InputEvent
- **REQ**: Service-Anfrage mit Parameter QI

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung mit Parametern QO und STATUS
- **CNF**: Bestätigung der angeforderten Service-Anfrage mit Parametern QO und STATUS
- **IND**: Indikation von der Ressource mit Parametern QO und STATUS

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangsqualifizierer
- **PARAMS** (STRING): Service-Parameter
- **u16ObjId** (UINT): Objekt-ID mit Initialwert ID_NULL

- **InputEvent** (SoftKeyActivationCode_S): Identifiziert das Ereignis gemäß ISO 11783-6 mit Initialwert „Invalid“

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifizierer
- **STATUS** (STRING): Service-Status

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Softkey_IE Funktionsblock verwaltet die Kommunikation mit Softkey-Eingabegeräten gemäß dem ISO-Bus-Standard 11783-6. Bei der Initialisierung (INIT) werden die Service-Parameter und Objekt-ID konfiguriert. Service-Anfragen (REQ) lösen die entsprechende Funktionalität aus, während Indikationen (IND) eingehende Ereignisse von den physikalischen Softkeys signalisieren.

Technische Besonderheiten

- Unterstützt ISO 11783-6 Standard für landwirtschaftliche Fahrzeuge
- Verwendet spezifische SoftKeyActivationCode-Struktur zur Ereignisidentifikation
- Integriert Objekt-ID-Verwaltung für Geräteidentifikation
- Bietet umfassende Statusrückmeldungen über STRING-Parameter

Zustandsübersicht

Der Funktionsblock verfügt über einen initialisierten und einen Betriebszustand. Nach erfolgreicher INIT-Initialisierung wechselt der Block in den Betriebszustand, in dem REQ-Anfragen verarbeitet und IND-Ereignisse empfangen werden können.

Anwendungsszenarien

- Steuerung von Bedienpanels in landwirtschaftlichen Maschinen
- Implementierung von Softkey-Funktionalitäten in mobilen Arbeitsgeräten
- ISO-Bus-konforme Eingabeverarbeitung in Fahrzeugen
- Benutzerschnittstellen für komplexe Maschinensteuerungen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu generischen Eingabebausteinen bietet Softkey_IE spezifische ISO 11783-6-Konformität und ist optimiert für die Anforderungen landwirtschaftlicher und mobiler Arbeitsmaschinen. Die Integration von SoftKeyActivationCode ermöglicht eine standardisierte Ereignisbehandlung.

🛠️ Zugehörige Übungen

- Uebung_010b2
- Uebung_010b2_AX
- Uebung_010b6
- Uebung_010b6_AX
- Uebung_013
- Uebung_013_AX
- Uebung_014
- Uebung_015
- Uebung_015a
- Uebung_016
- Uebung_016a
- Uebung_017
- Uebung_018
- Uebung_018a
- Uebung_019a
- Uebung_019b
- Uebung_021
- Uebung_022
- Uebung_023
- Uebung_024

- Uebung_025
- Uebung_026
- Uebung_039
- Uebung_039a
- Softkey_T_FF_ILOCK_TO_QX_BG

Fazit

Der Softkey_IE Funktionsblock stellt eine spezialisierte Lösung für die Softkey-Ereignisverarbeitung in ISO 11783-6-konformen Systemen dar. Durch seine standardisierte Schnittstelle und umfassende Statusrückmeldungen eignet er sich ideal für den Einsatz in komplexen mobilen Arbeitsmaschinen-Steuerungen.

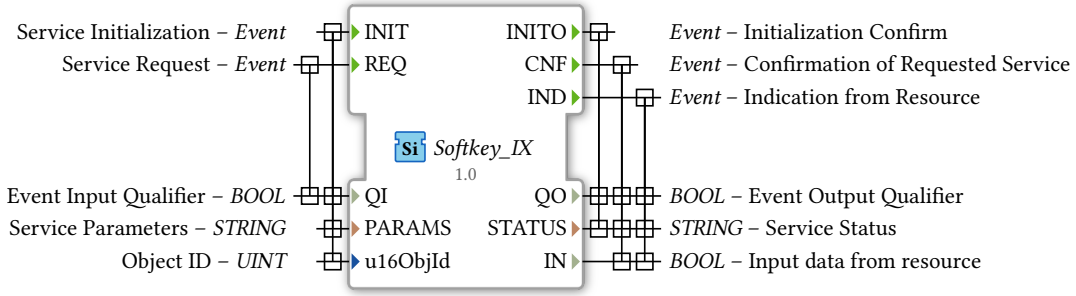
Softkey_IX

🎧 Podcast

- ISO 11783-6: Softkeys und das Virtual Terminal verstehen – Dein Schlüssel zur Landmaschinen-Mechatronik

Einleitung

Der Softkey_IX ist ein Service-Interface-Funktionsblock für boolesche Eingabedaten, der als Eingabeschnittstelle für Softkey-Funktionalitäten dient. Dieser Baustein ermöglicht die Kommunikation mit Ressourcen und die Verarbeitung von Tastatureingaben in Steuerungssystemen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung
  - Verknüpft mit: QI, PARAMS, u16ObjId
- **REQ**: Service-Anfrage
  - Verknüpft mit: QI

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
  - Verknüpft mit: QO, STATUS
- **CNF**: Bestätigung der angeforderten Service-Funktion
  - Verknüpft mit: QO, STATUS, IN
- **IND**: Indikation von der Ressource
  - Verknüpft mit: QO, STATUS, IN

Daten-Eingänge

- **QI**: BOOL - Ereignis-Eingangsqualifizierer
- **PARAMS**: STRING - Service-Parameter
- **u16ObjId**: UINT - Objekt-ID (Initialwert: ID_NULL)

Daten-Ausgänge

- **QO**: BOOL - Ereignis-Ausgangsqualifizierer
- **STATUS**: STRING - Service-Status
- **IN**: BOOL - Eingabedaten von der Ressource

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Softkey_IX-Funktionsblock dient als Vermittler zwischen der Anwendungslogik und physischen oder virtuellen Eingabegeräten. Bei der Initialisierung (INIT) werden die Service-Parameter konfiguriert. Über REQ-Ereignisse können Service-Anfragen gestellt werden, die mit CNF-Ereignissen bestätigt werden. IND-Ereignisse signalisieren asynchrone Eingaben von der Ressource.

Technische Besonderheiten

- Verwendet ISOBUS-kompatible Objekt-Identifikation
- Unterstützt parametrierbare Service-Konfiguration über STRING-Parameter
- Bietet sowohl synchrone (CNF) als auch asynchrone (IND) Betriebsmodi
- Initialisierung mit standardisierter Objekt-ID (ID_NULL)

Zustandsübersicht

Der Funktionsblock durchläuft folgende Hauptzustände:

1. **Nicht initialisiert**: Vor der INIT-Verarbeitung
2. **Initialisiert**: Nach erfolgreicher INIT-Verarbeitung
3. **Bereit**: Kann Service-Anfragen verarbeiten
4. **Aktiv**: Verarbeitet aktuelle Eingaben

Anwendungsszenarien

- Bedienpanels in mobilen Arbeitsmaschinen
- Virtuelle Tastaturen in Industrieanlagen
- Softkey-Implementierungen in Fahrzeugsystemen
- Benutzerschnittstellen für ISOBUS-kompatible Geräte

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Eingangsbausteinen bietet Softkey_IX erweiterte Service-Funktionalitäten mit parametrierbaren Konfigurationen und ISOBUS-Kompatibilität. Die IND-Funktionalität ermöglicht asynchrone Ereignisbehandlung, was bei reinen Abfrage-basierten Bausteinen nicht verfügbar ist.

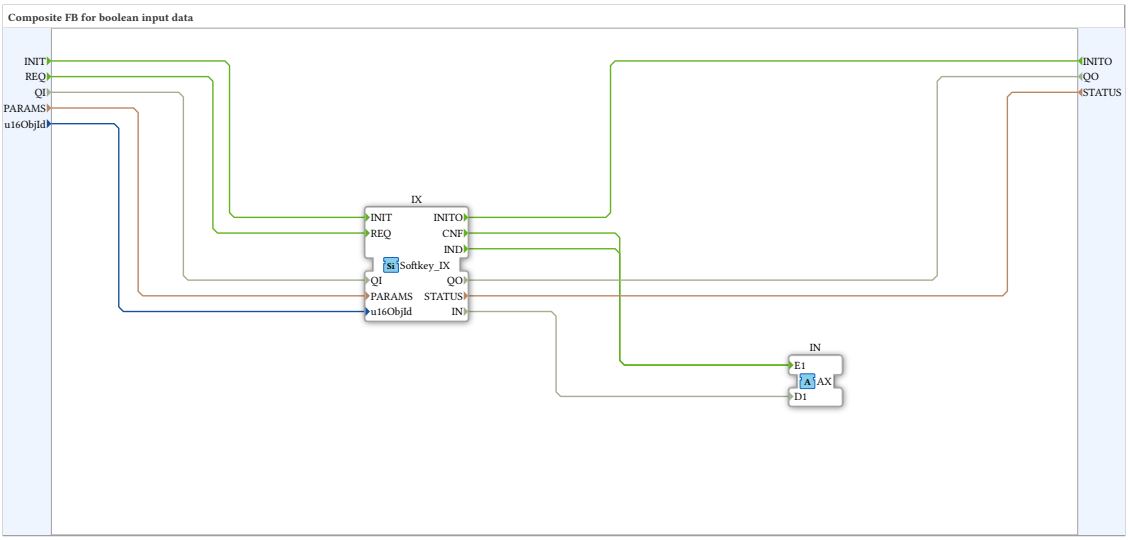
🛠️ Zugehörige Übungen

- Uebung_010
- Uebung_010a
- Uebung_010a4
- Uebung_010c
- Uebung_010c2
- SoftkeyOverride_TO_QX_BG
- Uebung_039b

Fazit

Der Softkey_IX-Funktionsblock stellt eine flexible und standardisierte Lösung für Softkey-Eingaben in industriellen Steuerungssystemen dar. Seine ISOBUS-Kompatibilität und die umfangreiche Service-Schnittstelle machen ihn besonders geeignet für anspruchsvolle Anwendungen in mobilen Arbeitsmaschinen und Industrieanlagen.

Softkey_IXA

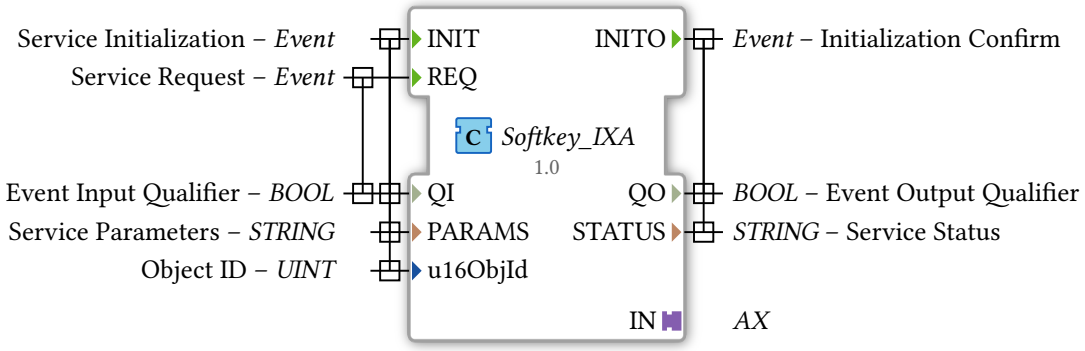


🎧 Podcast

- ISO 11783-6: Softkeys und das Virtual Terminal verstehen – Dein Schlüssel zur Landmaschinen-Mechatronik

Einleitung

Softkey_IXA ist ein Composite-Funktionsblock zur Verarbeitung von booleschen Eingabedaten. Der Baustein dient als Schnittstelle für Softkey-Operationen und ermöglicht die Initialisierung und Abfrage von Eingabezuständen über standardisierte ISOBUS-Kommunikationsprotokolle.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierungsereignis mit den zugehörigen Daten QI, PARAMS und u16ObjId
- **REQ**: Service-Anfrageereignis mit dem Qualifier QI

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung mit den Ausgabedaten QO und STATUS

Daten-Eingänge

- **QI**: BOOL - Event Input Qualifier
- **PARAMS**: STRING - Service-Parameter
- **u16ObjId**: UINT - Objekt-ID mit Initialwert ID_NULL

Daten-Ausgänge

- **QO**: BOOL - Event Output Qualifier
- **STATUS**: STRING - Service-Status

Adapter

- **IN**: Adapter vom Typ adapter::types::unidirectional::AX für unidirektionale Kommunikation

Funktionsweise

Der Composite-Funktionsblock Softkey_IXA kapselt den internen Funktionsblock Softkey_IX und dient als Vermittler zwischen externen Aufrufen und der eigentlichen Softkey-Funktionalität. Bei Initialisierung (INIT) werden die Parameter an den internen Baustein weitergeleitet, und bei Service-Anfragen (REQ) werden die entsprechenden Abfragen durchgeführt. Der Adapter IN ermöglicht die Kommunikation mit externen Komponenten.

Technische Besonderheiten

- Verwendet ISOBUS-spezifische Typen und Konstanten
- Unterstützt Objekt-Identifikation über u16ObjId
- Flexible Parameterübergabe via STRING-Parameter
- Unidirektionale Adapter-Schnittstelle für erweiterte Kommunikation

Zustandsübersicht

Der Funktionsblock verfügt über zwei Hauptzustände:

1. **Initialisierungszustand**: Verarbeitung der INIT-Ereignisse mit Parameterübergabe
2. **Service-Zustand**: Bearbeitung von REQ-Anfragen und Rückgabe der Ergebnisse

Anwendungsszenarien

- Landwirtschaftliche Bedienterminals mit Softkey-Funktionalität
- ISOBUS-kompatible Steuersysteme
- Anwendungen mit booleschen Eingabeabfragen
- Systeme mit parametrierbaren Service-Schnittstellen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen booleschen Eingabeblöcken bietet Softkey_IXA erweiterte ISOBUS-Kompatibilität, parametrierbare Service-Schnittstellen und Adapter-basierte Erweiterbarkeit. Die Composite-Struktur ermöglicht eine klare Trennung zwischen Schnittstellenlogik und Kernfunktionalität.

🛠️ Zugehörige Übungen

- Uebung_010_AX
- Uebung_010a4_AX
- Uebung_010a_AX
- Uebung_010b4_sub_AX
- Uebung_010b5_sub_AX
- Uebung_010c2_AX
- AuxIXA_TO_logiBUS_QXA_BG
- Uebung_010c_AX

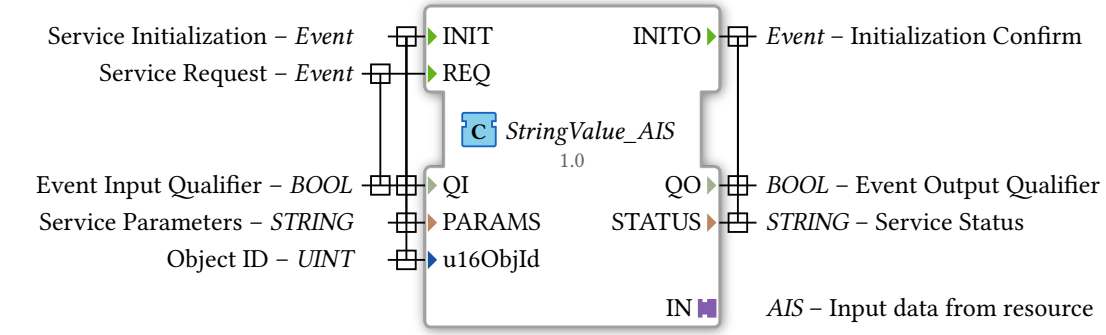
Fazit

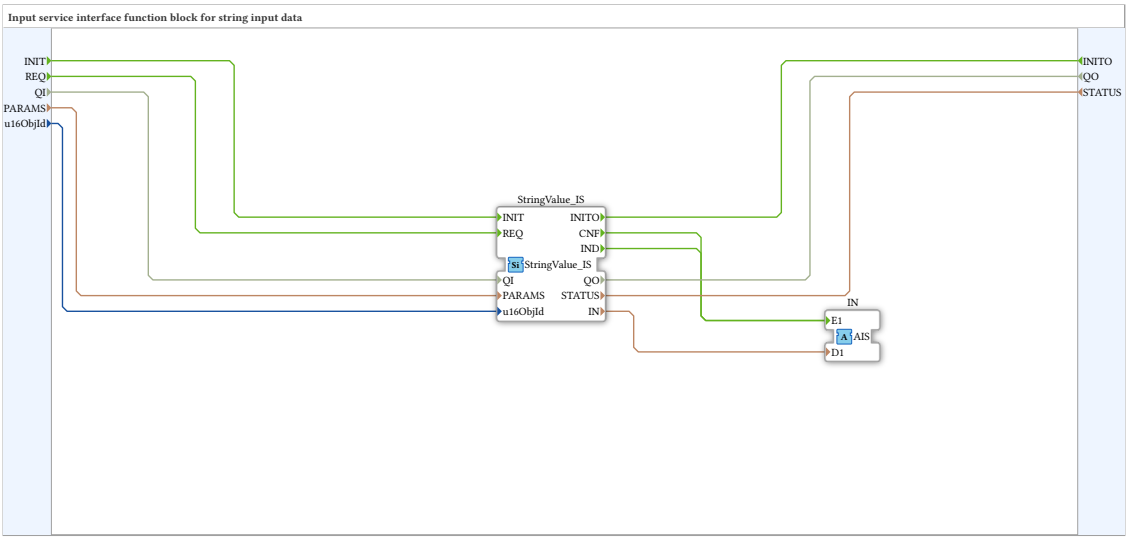
Softkey_IXA stellt einen robusten und erweiterbaren Composite-Funktionsblock für Softkey-Operationen in ISOBUS-Umgebungen dar. Durch die klare Schnittstellentrennung und Adapter-Unterstützung eignet er sich ideal für komplexe Steuerungssysteme in der Agrartechnik.

Numerischer Index (io)

Hier finden Sie alle Übungen des Arbeitsbereichs `io` in aufsteigender Reihenfolge.

StringValue_AIS





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **StringValue_AIS** ist ein Dienstschnittstellen-Baustein (Service Interface Function Block) zur Bereitstellung von Zeichenketten-Eingabedaten (String). Er dient als Schnittstelle zwischen einer externen Ressource (z. B. ISOBUS Universal Terminal) und der Steuerungslogik. Der Baustein initialisiert die Kommunikation, verarbeitet Anforderungen und gibt Statusinformationen an die Applikation weiter.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Mit Variablen	Beschreibung
INIT	EInit	QI, PARAMS, u16ObjId	Initialisiert den Dienst und übergibt Qualifier, Parameter und Objekt-ID.
REQ	Event	QI	Fordert eine Datenabfrage oder -verarbeitung an.

Ereignis-Ausgänge

Ereignis	Typ	Mit Variablen	Beschreibung
INITO	EInit	QO, STATUS	Bestätigt die Initialisierung und liefert Statusinformationen.

Daten-Eingänge

Variable	Typ	Initialwert	Beschreibung

QI	BOOL	–	Qualifier für Ereignisse (TRUE = aktiv).
PARAMS	STRING	–	Dienstparameter (z. B. Konfigurationsdaten).
u16ObjId	UINT	ID_NULL	Objekt-ID, die im Protokoll referenziert wird.

Daten-Ausgänge

Variable	Typ	Beschreibung
QO	BOOL	Ausgangsqualifier (Bestätigung des Dienstes).
STATUS	STRING	Statusmeldung des Dienstes (z. B. Fehlertext oder Erfolgsmeldung).

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::AIS	Eingang	Unidirektionaler Adapter für den Datenaustausch mit einer externen Ressource (z. B. Hardware-Schnittstelle).

Funktionsweise

Der Baustein **StringValue_AIS** kapselt einen internen Dienstbaustein **StringValue_IS** und steuert dessen Ereignis- und Datenflüsse. Bei einem INIT-Ereignis werden die Eingangsparameter (QI, PARAMS, u16ObjId) an den internen Baustein weitergeleitet, der die Initialisierung der Kommunikation durchführt. Der Ereignisausgang INITO signalisiert den Erfolg (QO, STATUS).

Ein REQ-Ereignis löst eine weitere Abfrage aus – der interne Baustein sendet die Anforderung (z. B. Lesen eines String-Werts) über den Adapter **IN** an die externe Ressource. Die Antwort wird asynchron über den Adapter empfangen (Ereignis IND oder CNF) und als STATUS-/QO-Ausgang zurückgegeben.

Die Verknüpfungen sind im FB-Netzwerk fest verdrahtet:

- **INIT** und **REQ** werden auf den internen Baustein **StringValue_IS** geführt.
- Die Antwortereignisse **IND** und **CNF** des internen Bausteins werden auf den Adapter-Ausgang **IN.E1** gelegt.
- Die Datenleitungen **QI**, **PARAMS**, **u16ObjId** speisen den internen Baustein, während **IN.D1** (vom Adapter) und **STATUS**/ **QO** an die Ausgänge des Bausteins weitergegeben werden.

Technische Besonderheiten

- Der Baustein ist als **Service Interface Function Block (SIFB)** konzipiert und eignet sich für die Einbindung in eine ISOBUS-konforme Steuerungsumgebung.
- Die Schnittstelle verwendet einen **unidirektionalen Adapter (AIS)**, der ausschließlich Daten von der Ressource zur Steuerung überträgt.
- Der Initialwert von `u16ObjId` ist auf `ID_NULL` gesetzt – ein Platzhalter, der im Initialisierungsschritt durch eine gültige Objekt-ID ersetzt werden muss.
- Die Implementierung ist unter der **Eclipse Public License 2.0** lizenziert.
- Der Baustein nutzt einen internen Type-Hash-Mechanismus zur Integritätsprüfung.

Zustandsübersicht

Eine explizite Zustandsmaschine ist im XML nicht dargestellt. Der Baustein arbeitet ereignisgesteuert:

- **IDLE**: Warten auf INIT.
- **INIT durchgeführt**: Initialisierung abgeschlossen, wartet auf REQ.
- **IN Bearbeitung**: Nach REQ wird die Anfrage an den Adapter gesendet; die Antwort erfolgt asynchron.

Die Rückmeldung über INITO signalisiert den Abschluss der Initialisierung, während der STATUS-Ausgang den aktuellen Zustand des Dienstes widerspiegelt.

Anwendungsszenarien

- **ISOBUS UT-Anbindung**: Einlesen eines Textfeldes (z. B. Maschinenname, Feldbezeichnung) von einem Universal Terminal.

- **Konfigurationsschnittstelle**: Übernahme von String-Parametern aus einem übergeordneten Steuerungssystem.
- **Datenprotokollierung**: Erfassung von externen Zeichenketten (z. B. Barcode-Scanner, Seriennummern) für die Weiterverarbeitung.

Vergleich mit ähnlichen Bausteinen

- **IntegerValue_AIS**: Analog aufgebaut, jedoch für Integer-Daten. Unterscheidet sich nur im Datentyp des Ausgangswerts.
- **CommandValue_AIS**: Bietet bidirektionale Kommunikation (Senden/Empfangen) und erwartet ein Kommando-Format.
- **StringValue_AIS** ist speziell auf den Empfang von Zeichenketten optimiert und erfordert keine Rückantwort außer Status.

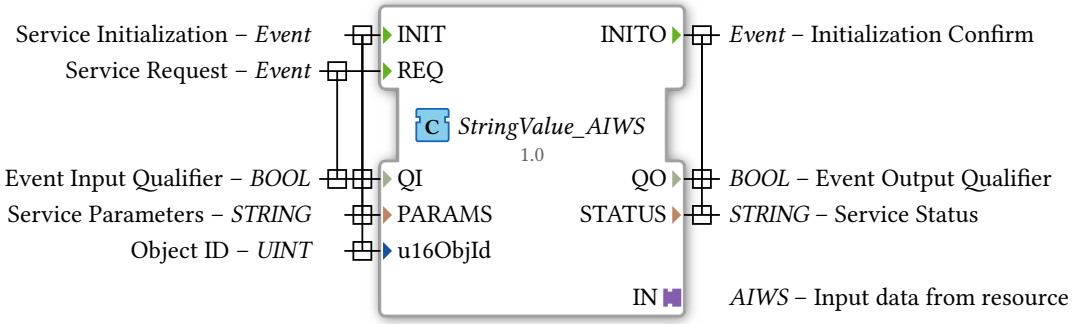
Fazit

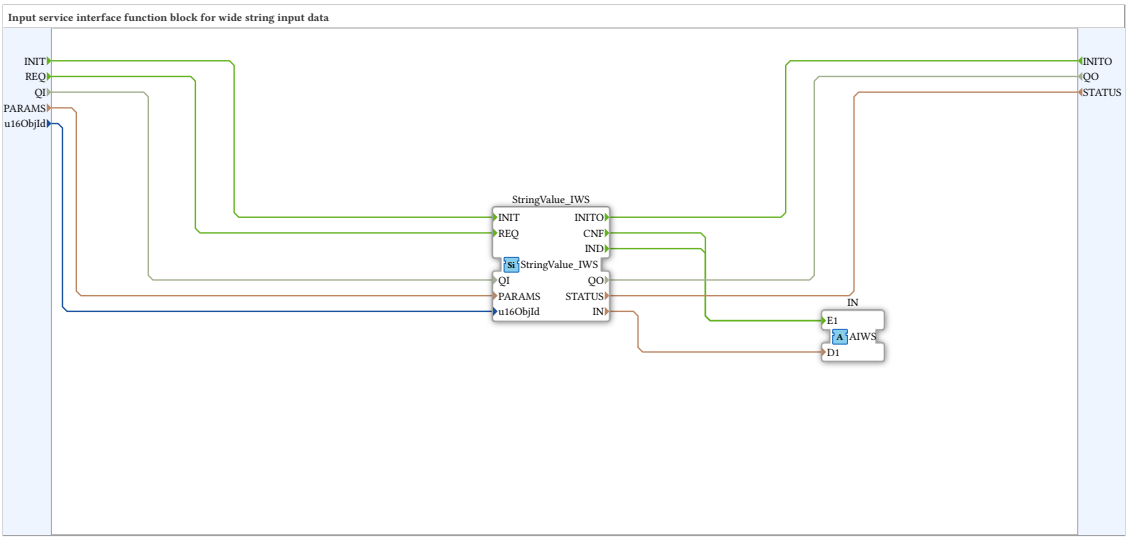
Der **StringValue_AIS** ist ein spezialisierter Dienstschnittstellen-Baustein für den Empfang von String-Daten über einen unidirektionalen Adapter. Er vereinfacht die Einbindung externer Textquellen in Automatisierungssysteme und bietet eine saubere Trennung zwischen Kommunikationsdiensten und Applikationslogik. Dank der standardisierten Schnittstelle (Ereignis-/Datenanschlüsse) lässt er sich flexibel in 4diac-IDE-Projekte integrieren.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

StringValue_AIWS





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **StringValue_AIWS** ist ein Eingabe-Service-Interface-Baustein für die Verarbeitung von Wide-String-Daten (UTF-16) in einem ISOBUS-Kontext. Er stellt eine Adapter-basierte Schnittstelle zur Verfügung, über die externe Ressourcen Zeichenketten eingeben können. Der Baustein dient als Wrapper für den internen Baustein *StringValue_IWS* und bietet eine standardisierte Initialisierungs- und Anforderungsschnittstelle.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Mit-Variablen	Kommentar
INIT	EInit	QI, PARAMS, u16ObjId	Initialisierung des Dienstes
REQ	Event	QI	Dienstanforderung (Abfrage)

Ereignis-Ausgänge

Ereignis	Typ	Mit-Variablen	Kommentar
INITO	EInit	QO, STATUS	Bestätigung der Initialisierung

Daten-Eingänge

Name	Typ	Initialwert	Kommentar
QI	BOOL	–	Ereignis-Qualifier (Freigabe)
PARAMS	STRING	–	Dienstparameter (z. B. Konfigurationsstring)
u16ObjId	UINT	ID_NULL	Objekt-ID (z. B. ISOBUS-Objektkennung)

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Ausgangs-Qualifier (Status der letzten Operation)
STATUS	STRING	Statusmeldung (z. B. Fehlertext oder Erfolgsmeldung)

Adapter

Name	Typ	Kommentar
IN	adapter::types::unidirectional::AIWS	Eingabe von Wide-String-Daten aus der Ressource (unidirektional)

Der Adapter IN empfängt die eigentlichen Zeichenketten und stellt sie dem Baustein zur Verfügung. Die Verbindung erfolgt über die Ereignisleitung E1 und den Datenkanal D1.

Funktionsweise

Der Baustein initialisiert über das Ereignis INIT den internen *StringValue_IWS*-Baustein. Dabei werden die Parameter QI, PARAMS und u16ObjId an diesen weitergegeben. Nach erfolgreicher Initialisierung wird das Ereignis INITO ausgelöst und die Ausgabevariablen QO und STATUS gesetzt.

Mit dem Ereignis REQ wird eine Anforderung zur Bereitstellung eines neuen Wide-String-Werts gestartet. Die eigentliche Zeichenkette wird asynchron über den Adapter IN geliefert: Sobald der interne Baustein eine neue Zeichenkette empfängt, sendet er ein Ereignis (IND oder CNF) an den Adapterausgang E1, und die Daten werden über den Datenkanal D1 an die angeschlossene Ressource weitergegeben.

Der Baustein kapselt die gesamte Logik zur Initialisierung und Kommunikation mit dem zugrunde liegenden Wide-String-Dienst und stellt eine einheitliche Adapter-Schnittstelle nach außen bereit.

Technische Besonderheiten

- **Wide-String-Unterstützung**: Der Baustein ist für UTF-16 (Wide Strings) ausgelegt und eignet sich daher besonders für internationale Zeichensätze (Unicode).
- **Adapterbasierte Kommunikation**: Die Verwendung eines unidirektionalen Adapters (`AIWS`) erlaubt eine saubere Trennung zwischen Dienstlogik und Ressourcenschnittstelle.
- **ISOBUS-Konformität**: Die Objekt-ID (`u16ObjId`) wird mit dem Initialwert `ID_NULL` belegt, was auf eine ISOBUS-spezifische Vergabe hinweist.
- **Ereignisgesteuerte Verarbeitung**: Sowohl Initialisierung als auch Datenanforderung werden über explizite Ereignisse ausgelöst, sodass der Baustein in zeitkritischen Umgebungen einsetzbar ist.

Zustandsübersicht

Der Baustein besitzt keine explizite Zustandsmaschine, das Verhalten wird durch die Ereignisse gesteuert:

1. **Ruhezustand** – Der Baustein wartet auf `INIT` oder `REQ`.
2. **Initialisierung** – Nach `INIT` werden die Parameter an den internen Baustein übergeben; die Ausgänge `QO` und `STATUS` aktualisiert und `INITO` gesendet.
3. **Anforderungszustand** – Nach `REQ` wird eine Datenanforderung gestartet. Die Antwort erfolgt asynchron über den Adapter `IN` (Ereignis `E1`).
4. **Fehlerbehandlung** – Bei einem Fehler während der Initialisierung wird `QO` auf `FALSE` gesetzt und eine entsprechende `STATUS`-Meldung ausgegeben.

Anwendungsszenarien

- **Eingabe von Unicode-Text** in landwirtschaftliche Bedienterminals (ISOBUS-UT) – z. B. für Fahrzeugnamen, Feldnamen oder Produktbezeichnungen.
- **Konfigurationsschnittstelle** für ISOBUS-Geräte, die Wide-String-Parameter benötigen.
- **Anbindung externer Eingabegeräte** (Tastaturen, Touchscreens) an ein ISOBUS-Steuergerät über eine standardisierte Adapter-Schnittstelle.

Vergleich mit ähnlichen Bausteinen

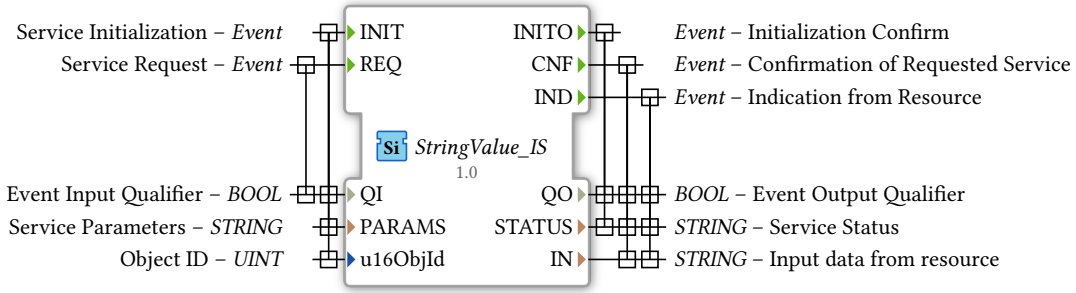
Baustein	Schnittstelle	Besonderheit
`StringValue_AIWS`	Adapter (Eingang)	Empfängt Wide-Strings über den Adapter `AIWS`
`StringValue_IWS`	Direkte Events/Daten	Interne Logik, keine Adapter-Schnittstelle
`StringValue` (Standard)	Events/Daten	Einfacher Wide-String-Eingang (ohne Adapter)

Der `StringValue_AIWS` hebt sich durch seine Adapter-basierte Schnittstelle hervor, die eine lose Kopplung zwischen Ressource und Dienst ermöglicht. Er eignet sich besonders für modulare Systeme, bei denen die Datenquelle dynamisch wechseln kann.

Fazit

Der **StringValue_AIWS** ist ein robuster Eingabe-Service-Interface-Baustein für Wide-String-Daten in ISOBUS-Systemen. Durch die Kombination von ereignisgesteuerter Initialisierung, Adapter-Kommunikation und klarer Schnittstellenstruktur bietet er eine zuverlässige Grundlage für die Verarbeitung von Unicode-Texten in industriellen Anwendungen. Die Trennung von Dienstlogik und Ressourcenschnittstelle vereinfacht die Wartbarkeit und Wiederverwendbarkeit des Bausteins.

StringValue_IS



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **StringValue_IS** ist ein Eingangs-Service-Interface-Funktionsblock (Input SIFB) für den Empfang von Zeichenketten-Daten (String) aus einer Ressource. Er dient als standardisierte Schnittstelle, um externe String-Informationen in die 4diac-Umgebung zu integrieren. Typische Anwendungen sind die Anbindung an ISOBUS-Systeme, bei denen Zeichenketten wie Benennungen, Statusmeldungen oder Kennungen empfangen werden.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung	Mitgeführte Daten
`INIT`	EInit	Initialisierung des Service-Interface-Bausteins	`QI`, `PARAMS`, `u16ObjId`
`REQ`	Event	Anforderung zur Datenabfrage oder -aktualisierung	`QI`

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung	Mitgeführte Daten
`INITO`	EInit	Bestätigung der erfolgreichen Initialisierung	`QO`, `STATUS`
`CNF`	Event	Bestätigung der ausgeführten Anforderung	`QO`, `STATUS`, `IN`
`IND`	Event	Asynchrone Indikation – neue Eingangsdaten von der Ressource	`QO`, `STATUS`, `IN`

Daten-Eingänge

Name	Typ	Anfangswert	Beschreibung
`QI`	BOOL	–	Ereignisqualifizierer: Steuert die Ausführung der zugehörigen Ereignisse
`PARAMS`	STRING	–	Service-Parameter (z. B. Konfiguration des Zugriffs auf die Ressource)
`u16ObjId`	UINT	`ID_NULL`	Objekt-ID, die den zu lesenden String in der Ressource identifiziert

Daten-Ausgänge

Name	Typ	Beschreibung
`QO`	BOOL	Ereignisqualifizierer: Zeigt an, ob der Ausgang gültige Daten liefert
`STATUS`	STRING	Statusmeldung des Service (z. B. Fehlercodes oder Erfolgsmeldung)
`IN`	STRING	Empfangene Zeichenkette von der Ressource

Adapter

Der Funktionsblock verfügt über keine Adapter-Schnittstellen.

Funktionsweise

Der **StringValue_IS** arbeitet nach dem üblichen Schema eines Service-Interface-Funktionsblocks für den Dateneingang:

1. **Initialisierung (`INIT`)** Durch Anlegen eines `INIT`-Ereignisses wird der Baustein konfiguriert. Die Parameter `QI` (Qualifizierer), `PARAMS` (Service-Parameter) und `u16ObjId` (Objekt-ID) werden übergeben. Nach erfolgreicher Initialisierung bestätigt der Baustein mit dem Ereignis `INITO` und liefert den Status über `QO` und `STATUS`.

2. **Anforderung (`REQ`)** Ein `REQ`-Ereignis löst eine aktive Datenabfrage bei der Ressource aus. Der Qualifizierer `QI` wird ausgewertet. Die Antwort wird mit dem Ereignis `CNF` zurückgegeben. Der empfangene String steht dann am Ausgang `IN` zur Verfügung.
3. **Asynchrone Indikation (`IND`)** Die Ressource kann jederzeit neue Daten senden. Der Baustein signalisiert dies durch das Ereignis `IND`. Gleichzeitig werden die aktuellen Daten am Ausgang `IN` sowie der Status über `QO` und `STATUS` bereitgestellt.

Die Ausgänge `IN`, `QO` und `STATUS` werden bei jedem der Ereignisse `CNF` und `IND` aktualisiert und können sofort von nachfolgenden Bausteinen verwendet werden.

Technische Besonderheiten

- **Objekt-ID:** Der Eingang `u16ObjId` ist mit dem Initialwert `ID_NULL` vordefiniert. Vor der ersten Nutzung muss eine gültige Objekt-ID gesetzt werden. Die tatsächliche Bedeutung der ID ist abhängig vom zugrundeliegenden Ressourcen-Protokoll (z. B. ISOBUS-Objektnummer).
- **Service-Parameter (`PARAMS`):** Dieser String-Eingang dient der Übergabe konfigurationsspezifischer Parameter (z. B. Busadresse, Kommunikationspfad). Die genaue Syntax ist vom implementierten Treiber abhängig.
- **Typische Ereignissequenz:** `INIT` → `INITO` → (ggf. wiederholte `REQ` ↔ `CNF` oder asynchrone `IND`).
- **Fehlerbehandlung:** Fehler werden über den `STATUS`-Ausgang signalisiert und können zur Fehlerdiagnose herangezogen werden.

Zustandsübersicht

Der Baustein durchläuft folgende grundlegende Zustände (nicht im XML explizit modelliert, aber typisch für SIFBs):

- **Nicht initialisiert:** Nach dem Start des Systems. Nur `INIT` wird verarbeitet.
- **Initialisiert:** Nach erfolgreichem `INITO`. Der Baustein ist bereit für `REQ` und kann asynchrone `IND` empfangen.
- **Anforderung läuft:** Nach einem `REQ` bis zum Eintreffen von `CNF`. In dieser Zeit werden keine weiteren Anforderungen verarbeitet.
- **Fehlerzustand:** Bei Fehlern in der Kommunikation. Kann durch erneutes `INIT` zurückgesetzt werden.

Anwendungsszenarien

- **ISOBUS-Steuergeräte:** Empfangen von Zeichenketten wie Gerätenamen, Variablenbezeichnungen oder Statusmeldungen von einem ISOBUS-Server.
- **Benutzerschnittstellen:** Einlesen von Textdaten, die von einem externen HMI oder einem Bedienpanel gesendet werden.
- **Konfigurationsdaten:** Abrufen von String-Parametern aus einem zentralen Konfigurationsservice.

Vergleich mit ähnlichen Bausteinen

Baustein	Datentyp	Besonderheit
`BOOL_IS`	BOOL	Einfache boolesche Eingänge (z. B. Schalter)
`INT_IS`	INT	Ganzzahlige Eingänge (z. B. Zählerstände)
`REAL_IS`	REAL	Gleitkomma-Eingänge (z. B. Messwerte)
`StringValue_IS`	**STRING**	**Zeichenketten – für Namen, Beschreibungen, irregular formatierte Daten**

Der `StringValue_IS` ergänzt die Familie der Eingangs-SIFBs um den flexiblen Datentyp `STRING`, der sich durch variable Länge und Inhalt auszeichnet.

Fazit

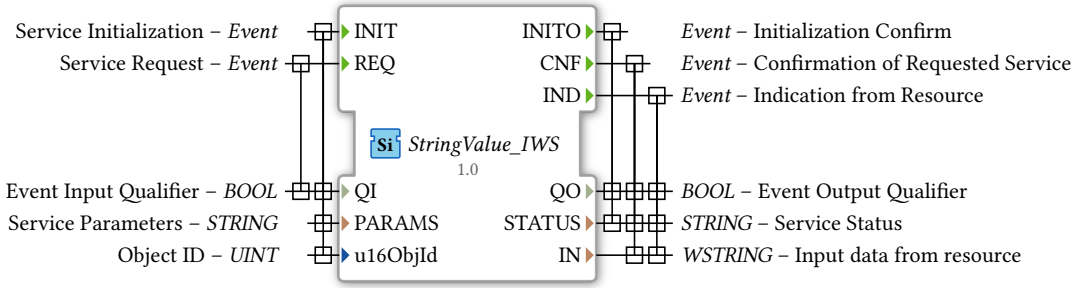
Der **StringValue_IS** ist ein wesentlicher Baustein für die Integration zeichenkettenbasierter Daten in 4diac. Er bietet eine standardisierte Schnittstelle zur Ressource, ermöglicht sowohl aktive Abfragen als auch asynchrone Indikationen und lässt sich durch die Parameter `PARAMS` und `u16ObjId` flexibel

konfigurieren. Seine Verwendung vereinfacht die Anbindung an Protokolle wie ISOBUS und erhöht die Wiederverwendbarkeit von Anwendungslogiken.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

StringValue_IWS



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **StringValue_IWS** ist ein Service-Interface-Funktionsblock (SIFB), der für den Empfang breiter Zeichenketten (*Wide String*, `WSTRING`) als Eingabedaten aus einer ressourcenseitigen Quelle (z. B. Hardware-Schnittstelle, Bussystem) zuständig ist. Er stellt eine standardisierte Schnittstelle zur Verfügung, um Initialisierung und zyklische/anforderungsgesteuerte Datenabrufe sowie asynchrone Indikationen zu verwalten. Der Baustein ist typischerweise in ISOBUS-basierten Steuerungsumgebungen oder anderen Feldbussystemen anzutreffen, die Unicode-kodierte Textdaten verarbeiten.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Kommentar	Mitgeführte Daten
`INIT`	Service-Initialisierung	`QI`, `PARAMS`, `u16ObjId`
`REQ`	Service-Anforderung	`QI`

Ereignis-Ausgänge

Ereignis	Kommentar	Mitgeführte Daten
`INITO`	Bestätigung der Initialisierung	`QO`, `STATUS`
`CNF`	Bestätigung der angeforderten Aktion	`QO`, `STATUS`, `IN`
`IND`	Asynchrone Indikation von der Ressource	`QO`, `STATUS`, `IN`

Daten-Eingänge

Name	Typ	Initialwert	Kommentar
`QI`	`BOOL`	–	Ereigniseingangs-Qualifier
`PARAMS`	`STRING`	–	Dienstparameter (z. B. Port-/Gerätekonfiguration)
`u16ObjId`	`UINT`	`ID_NULL`	Objekt-ID zur Identifikation der Datenquelle

Daten-Ausgänge

Name	Typ	Kommentar

QO	BOOL	Ereignisausgangs-Qualifier
STATUS	STRING	Status- oder Fehlermeldung
IN	WSTRING	Empfangene Eingabedaten (breite Zeichenkette)

Adapter

Keine.

Funktionsweise

Der StringValue_IWS folgt dem typischen Verhalten eines Service-Interface-Funktionsblocks:

1. **Initialisierung (INIT → INITO)** Durch Auslösen des INIT-Ereignisses wird der Dienst gestartet. Der Qualifier QI bestimmt, ob die Initialisierung aktiv (TRUE) oder deaktivierend (FALSE) ist. Mit PARAMS werden notwendige Konfigurationsparameter übergeben (z. B. Busadresse, Protokoll-Optionen). Die Objekt-ID u16ObjId legt fest, welche Datenquelle (z. B. ein bestimmtes ISOBUS-Objekt) angesprochen werden soll. Nach erfolgreicher Initialisierung wird INITO ausgegeben; QO und STATUS geben den Ausgangszustand an.
2. **Anforderung (REQ → CNF)** Mit dem REQ-Ereignis wird eine explizite Datenabfrage ausgelöst. Der Baustein holt (z. B. über einen Treiber) einen neuen Wert vom angeschlossenen Gerät. Die Antwort wird über den Ausgang IN (als WSTRING) und den Status STATUS bereitgestellt. Gleichzeitig wird das CNF-Ereignis gesendet.
3. **Asynchrone Indikation (IND)** Falls die Ressource unaufgefordert neue Daten liefert (z. B. bei Änderung einer Eingabe), wird das IND-Ereignis ausgelöst. IN enthält dann die aktuellen Daten, STATUS den zugehörigen Zustand. Dieser Ausgang kann parallel zum CNF-Ereignis genutzt werden.

Die Ereignisausgänge QO liefern immer den aktuellen Qualifier-Wert (in der Regel Übernahme von QI bei erfolgreicher Operation, sonst FALSE). Der Status STATUS enthält menschenlesbare Fehler- oder Erfolgsmeldungen.

Technische Besonderheiten

- **Breite Zeichenketten (WSTRING)** Der Baustein verwendet WSTRING (UTF-16-kodiert), um internationale Zeichensätze, Sonderzeichen und Emojis korrekt zu übertragen. Dies ist besonders in der ISOBUS-Landtechnik wichtig, wenn z. B. Bedienterminals Unicode-Texte anzeigen.
- **Objekt-ID u16ObjId** Der Initialwert ID_NULL wird aus der importierten Konstanten isobus::UT::Q::const::IDs::ID_NULL übernommen. Eine gültige Objekt-ID muss vor der INIT-Aktion gesetzt werden, um die korrekte Datenquelle zu adressieren.
- **Service-Interface-Charakter** Der Baustein ist als SIFB realisiert und erwartet eine ressourcenseitige Implementierung (Treiber). Die eigentliche Kommunikation mit der Hardware oder dem Bussystem erfolgt außerhalb des Funktionsblocks und wird über die Ereignis-/Datenschnittstellen abstrahiert.
- **Verwendung von eclipse4diac::core::TypeHash** Das Attribut dient der Laufzeit-Identifikation des Typs und wird für dynamische Funktionsblock-Instanzen verwendet.

Zustandsübersicht

Eine explizite Zustandsmaschine ist im XML nicht definiert, jedoch ergibt sich aus der Semantik des SIFB folgender impliziter Ablauf:

1. **IDLE** – Warten auf INIT oder erste REQ.
2. **INIT** – Initialisierungsphase, nach erfolgreicher Ausführung Wechsel in **READY**.
3. **READY** – Bereit für REQ oder IND. Bei REQ kurzzeitig in **BUSY**, dann zurück zu **READY**.
4. **INDICATION** – Asynchrone Datenindikation, nach Verarbeitung Rückkehr zu **READY**.
5. **ERROR** – Bei Fehlern (z. B. Kommunikationsabbruch) wird STATUS gesetzt und der Block verharrt, bis ein erneutes INIT (mit QI=FALSE ggf. als Reset) erfolgt.

Anwendungsszenarien

- **ISOBUS-Terminal (UT) – Eingabe von Unicode-Text** Empfang von Benutzereingaben wie Maschinenbezeichnungen, Adressen oder Diagnosetexten von einem Bedienterminal über den ISOBUS.

- **Universal-Interface für Zeicheneingabe** Anbindung externer Sensoren oder Eingabegeräte, die Zeichenketten im UTF-16-Format liefern (z. B. Barcode-Scanner, RFID-Leser mit Textausgabe).
- **Diagnose- und Logging-Schnittstelle** Aufnahme von Statusmeldungen, Warnungen oder Fehlertexten aus einer Steuerungseinheit zur Weiterleitung an ein HMI oder eine Cloud-Anbindung.

Vergleich mit ähnlichen Bausteinen

Funktionsblock	Typ	Unterschied
`StringValue_IWS` (vorliegend)	`WSTRING`	Empfängt UTF-16-kodierte Zeichenketten.
`StringValue_IWS` (normale Variante)	`STRING`	Empfängt ASCII-/UTF-8-Strings, geringere Zeichenabdeckung.
`AnalogInput_IWS`	`REAL`	Empfängt analoge Messwerte, keine Zeichenketten.
`EventInput_IWS`	–	Nur Ereignis-Schnittstelle, keine Daten.

Der `StringValue_IWS` ist speziell für die Verarbeitung von Unicode-Text konzipiert und eignet sich daher für internationale oder zeichensatzlastige Anwendungen.

Fazit

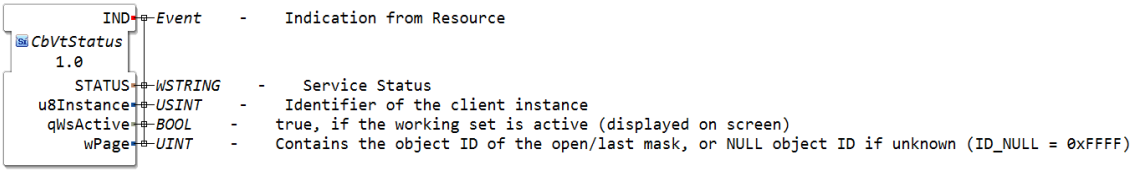
Der `StringValue_IWS` ist ein essenzieller Baustein für die Integration von Text-Eingaben in IEC 61499-basierten Steuerungssystemen, insbesondere im ISOBUS-Umfeld. Seine Unterstützung von `WSTRING`, die saubere Trennung von Initialisierung, Anforderung und asynchroner Indikation sowie die flexible Objekt-ID-Adressierung machen ihn zu einer robusten Lösung für den Empfang breiter Zeichenketten aus verteilten Ressourcen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

status

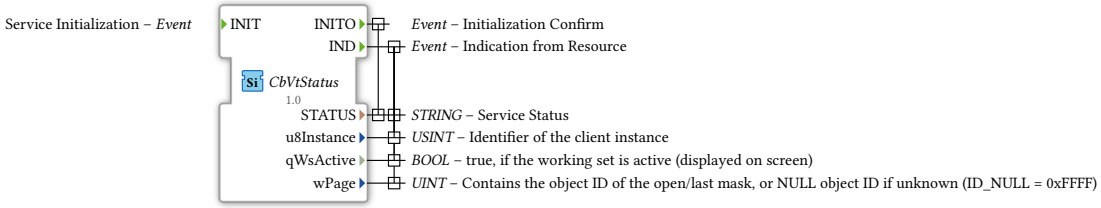
CbVtStatus



• — ○ • — ○ • — ○ •

Einleitung

Der **CbVtStatus** ist ein standardisierter Callback-Baustein für Virtual-Terminal-Seitenwechsel, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 unterstützt ISO 11783-6 (ISOBUS) in landwirtschaftlichen Steuersystemen.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung (mit STATUS)
- `IND`: Seitenwechsel-Indikation (mit Seiteninfo, `STATUS`, `wPage`, `qWsActive`, `u8Instance`, `wSKM_Scal`, `w_DMScal`, `u8SKM_Number`, `qScalingTestEnabled`)

Daten-Ausgänge

- `STATUS` (STRING): Betriebsstatus
- `u8Instance` (USINT): Client-Instanzkennung
- `qWsActive` (BOOL): Arbeitsmengen-Status
- `wPage` (UINT): Objekt-ID der geöffneten/letzten Maske, oder `ID_NULL` (0xFFFF), falls unbekannt
- `wSKM_Scal` (UINT): Skalierungsfaktor der Softkey-Maske (PoolSoftKeyMaskScalFaktor), Einheit 1/10000 (10000 = 100 %); ggf. durch SCALING-TEST überschrieben
- `w_DMScal` (UINT): Skalierungsfaktor der Datenmaske (PoolDataMaskScalFaktor), Einheit 1/10000 (10000 = 100 %); ggf. durch SCALING-TEST überschrieben
- `u8SKM_Number` (USINT): Anzahl der physischen Softkeys (VT_PHYSICALSOFTKEYS/sknu), gültiger Bereich 6..11; ggf. durch SCALING-TEST überschrieben
- `qScalingTestEnabled` (BOOL): `TRUE`, wenn der SCALING-TEST-Override (`[ScalingTest] enable=1` in der `settings.ini`) aktiv ist

Funktionsweise

1. **Initialisierung**:
   - `INIT`-Ereignis startet Setup
   - `INITO` bestätigt mit STATUS
2. **Seitenwechsel**:
   - `IND` wird bei Maskenwechsel ausgelöst
   - Liefert: Seiten-ID, Aktivstatus, Instanznummer
3. **Betrieb**:
   - Echtzeit-Überwachung des VT-Zustands
   - Multi-ECU-fähige Architektur

Technische Besonderheiten

✓ **ISO 11783-6 konform** ✓ **Deterministisches** Verhalten ✓ **Multi-ECU Unterstützung** ✓ **Plug-in-fähige** Architektur

Unterstützte Betriebsmodi

Modus	Beschreibung
Einzelgerät	Standardbetrieb mit u8Instance=0
Multi-ECU	Parallele Instanzen (1-255)
Diagnose	STATUS-Meldungen für Service

Anwendungsszenarien

- **Traktorsteuerungen**: Bedienmasken-Überwachung
- **Erntemaschinen**: Arbeitsdaten-Anzeige
- **Diagnosetools**: VT-Zustandsmonitoring
- **Multi-Display-Systeme**: Synchronisation

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	CbVtStatus	VtPageManager	VtEventLogger
Seitenwechsel	✓	✓	✖
Multi-ECU	✓	✖	✖

STATUS-Rückmeldung	✓	✖	✓

Fehlerbehandlung

- Ungültige Seiten-ID:
  - STATUS = „ERR_INVALID_PAGE“
  - wPage = 0xFFFF
- Initialisierungsfehler:
  - STATUS = „ERR_INIT_FAILED“
  - System im Safe-State

🛠️ Zugehörige Übungen

- Uebung_012
- Uebung_012a
- Uebung_012b
- Uebung_012c

Fazit

Der CbVtStatus-Baustein ist essentiell für ISOBUS-konforme Agrarsysteme:

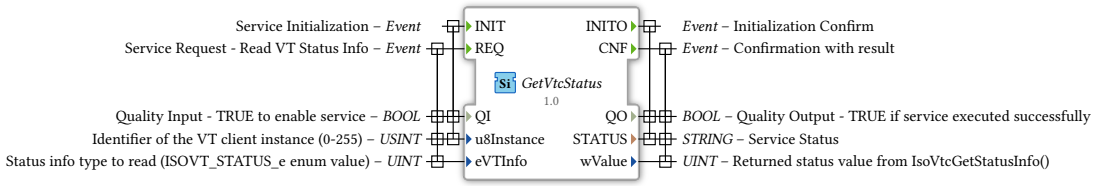
- Robustes Seitenwechsel-Management
- Volle ISO-11783-6-Kompatibilität
- Bewährte Architektur in Serieprodukten

Besonders wertvoll für Hersteller von:

- Traktorsteuergeräten
- Erntemaschinen-Kontrollsystemen
- ISOBUS-Diagnosegeräten

Die standardisierte Implementierung ermöglicht zuverlässigen Betrieb in heterogenen Systemumgebungen.

GetVtcStatus



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock **GetVtcStatus** ist ein Service Interface Block (SIFB), der die Funktion `IsoVtcGetStatusInfo()` aus dem ISOBUS-Treiber kapselt. Er ermöglicht das Auslesen verschiedener Statusinformationen eines VT-Clients sowie des angeschlossenen Virtual Terminals (VT). Der Baustein wurde gemäß dem Standard ISO 11783-6 entwickelt und eignet sich für die Integration in ISOBUS-Anwendungen.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung	Mitgeführte Daten
`INIT`	EInit	Initialisiert den Baustein.	`QI`, `u8Instance`
`REQ`	Event	Fordert das Auslesen eines Statuswertes an.	`QI`, `u8Instance`, `eVTInfo`

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung	Mitgeführte Daten
`INITO`	EInit	Bestätigung der Initialisierung.	`QO`, `STATUS`
`CNF`	Event	Bestätigung der Leseanforderung mit Ergebnis.	`QO`, `STATUS`, `wValue`

Daten-Eingänge

Name	Typ	Anfangswert	Beschreibung
`QI`	BOOL	–	Qualitätseingang: TRUE aktiviert den Dienst.
`u8Instance`	USINT	–	Identifikator der VT-Client-Instanz (0–255).
`eVTInfo`	UINT	0	Art der abzufragenden Statusinformation (Werte aus der Enumeration `ISOVT_STATUS_e`).

Daten-Ausgänge

Name	Typ	Beschreibung
`QO`	BOOL	Qualitätsausgang: TRUE bei erfolgreicher Ausführung.
`STATUS`	STRING	Dienststatus – enthält eine textuelle Rückmeldung.
`wValue`	UINT	Der von `IsoVtcGetStatusInfo()` zurückgegebene Wert (abhängig von `eVTInfo`).

Adapter

Keine Adapter vorhanden.

Funktionsweise

1. **Initialisierung**
   Durch das Ereignis `INIT` wird der Baustein aktiviert. Der Parameter `QI` muss auf TRUE gesetzt sein, damit der Dienst gestartet werden kann. Die zu verwendende VT-Client-Instanz wird über `u8Instance` festgelegt. Nach erfolgreicher Initialisierung wird das Ereignis `INITO` mit den Ausgabedaten `QO` und `STATUS` gesendet.

2. **Auslesen einer Statusinformation**
   Mit dem Ereignis `REQ` wird eine konkrete Abfrage gestartet. Dabei muss über `eVTInfo` der gewünschte Statuswert aus der Enumeration `ISOVT_STATUS_e` ausgewählt werden (siehe Liste in den technischen Besonderheiten). Der Baustein ruft intern die Funktion `IsoVtcGetStatusInfo()` auf und sendet nach Abschluss das Ereignis `CNF`. Die Ausgänge enthalten:

   - `QO` = TRUE bei erfolgreichem Lesevorgang,
   - `STATUS` = Beschreibung des Ergebnisses,
   - `wValue` = der ausgelesene Zahlenwert.

   Falls `QI` während des REQ-Aufrufs FALSE ist, wird der Dienst nicht ausgeführt und ein entsprechender Fehlerstatus zurückgegeben.

Technische Besonderheiten

- Der Baustein ist als **Service Interface Block** (SIFB) realisiert und greift auf die systemnahe ISOBUS-Treiberbibliothek zu.
- Die möglichen Werte für `eVTInfo` (ISOVT_STATUS_e) sind:

Wert	Bezeichnung	Beschreibung
0	VT_SOURCE_ADDRESS	Quelladresse des VT
2	VT_HND	CF-Handle des VT
3	CF_SOURCE_ADDRESS	Quelladresse des VT-Client

4	CF_HND	CF-Handle des VT-Client
6	ID_VISIBLE_DATA_MASK	Auf dem VT geöffnete Datenmaske
7	ID_VISIBLE_SOFTKEY_MASK	Auf dem VT geöffnete Softkey-Maske
8	VT_BUSY_CODE	Busy-Code der VT-Statusmeldung
9	AUXUNITS_TYPE1_ONBUS	TRUE, wenn Hilfsgerät Typ 1 am Bus ist
11	VT_ALIVE	VT seit mehr als 3 Sekunden nicht erreichbar
12	VT_DOWNLOAD_FINISHED	Vollständiges Announcing abgeschlossen
13	VT_POOL_ACTIVE_onVT	Pool (ausgewählt) auf dem VT aktiv
14	VT_STATEOFANNOUNCING	Zustand des Announcing
15	WS_VERSION_NR	Versionsnummer des Working Set
16	VT_NUMBOFVERSIONSTRINGS	Anzahl der gesendeten Versionsstrings des VT
17	VT_NAVSOFTKEYS	Navigations-Softkeys (Version 4)
18	VT_SOFTKEYXDOT	Softkey-Bezeichner – Pixel X
19	VT_SOFTKEYYDOT	Softkey-Bezeichner – Pixel Y
20	VT_VIRTUALSOFTKEYS	Anzahl der virtuellen Softkeys
21	VT_PHYSICALSOFTKEYS	Anzahl der physischen Softkeys
25	VT_BOOTTIME	Bootzeit des VT
26	VT_GRAPHICTYPE	Grafiktyp der VT-Hardware
29	VT_VERSIONNR	Version des Working Set VT

- Der Baustein unterstützt sowohl die Initialisierung als auch den wiederholten Aufruf von Leseoperationen, sodass in einer Schleife verschiedene Statuswerte abgefragt werden können.

Zustandsübersicht

Der Baustein implementiert eine einfache zustandsgesteuerte Logik:

1. **Startzustand** – Der Baustein wartet auf das Ereignis `INIT`.
2. **Initialisierung** – Nach Erhalt von `INIT` (mit gültigem QI) wird der Dienst initialisiert. Im Erfolgsfall wird `INITO` mit QO=TRUE gesendet. Bei Fehler wird QO=FALSE und ein entsprechender STATUS gesendet.
3. **Bereit** – Nach erfolgreicher Initialisierung können beliebig viele `REQ`-Ereignisse verarbeitet werden. Jedes `REQ` führt zu einem Aufruf von `IsoVtcGetStatusInfo()` und dem anschließenden Senden von `CNF`.
4. **Fehlerbehandlung** – Tritt während eines REQ ein interner Fehler auf, wird `CNF` mit QO=FALSE und einer Fehlerbeschreibung in `STATUS` ausgegeben.

Anwendungsszenarien

- **Diagnose und Überwachung** eines ISOBUS-Systems: Auslesen der aktuellen VT-Adresse, des Grafiktyps oder der Bootzeit.
- **Erkennung der VT-Verfügbarkeit** durch Abfrage von `VT_ALIVE` und `VT_DOWNLOAD_FINISHED`.
- **Steuerung der Softkey-Konfiguration** durch Abfrage der Anzahl physikalischer und virtueller Softkeys (`VT_PHYSICALSOFTKEYS`, `VT_VIRTUALSOFTKEYS`).
- **Abgleich der Working-Set-Version** zwischen verschiedenen Komponenten mithilfe von `VT_VERSIONNR` und `WS_VERSION_NR`.

Vergleich mit ähnlichen Bausteinen

Ähnliche Bausteine wie `GetVtPool` oder `GetVtObject` fokussieren auf das Lesen von Pool- oder Objektdaten. Der `GetVtcStatus`-Baustein hingegen ist speziell für den Zugriff auf den allgemeinen

VT-Client-Status ausgelegt. Er bietet eine direkte Schnittstelle zu den Systemstatuseigenschaften der ISOBUS-Spezifikation und ergänzt Bausteine, die auf Masken- oder Softkey-Informationen zugreifen.

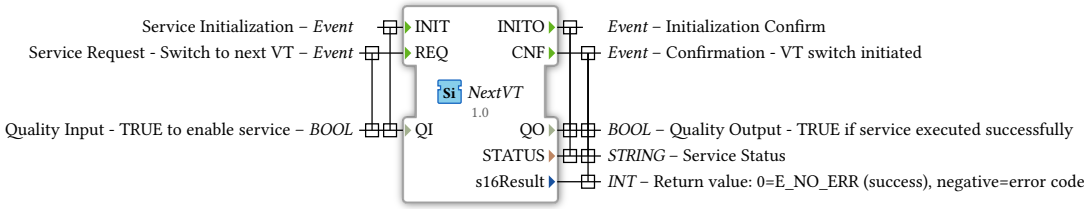
Fazit

Der Funktionsblock `GetVtcStatus` stellt eine kompakte und standardkonforme Möglichkeit dar, um auf die systemnahen Statusdaten eines ISOBUS-Virtual Terminals zuzugreifen. Seine klare Schnittstelle und die vielfältigen Abfragemöglichkeiten machen ihn zu einem wertvollen Werkzeug für die Implementierung von Diagnose-, Überwachungs- und Konfigurationsfunktionen in landwirtschaftlichen Steuerungssystemen.

Numerischer Index (UT)

Hier finden Sie alle Übungen des Arbeitsbereichs `UT` in aufsteigender Reihenfolge.

NextVT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **NextVT** ist ein Service-Interface-Block, der das Umschalten zum nächsten verfügbaren Virtual Terminal (VT) in einem ISOBUS-Netzwerk (ISO 11783-6) ermöglicht. Er kapselt die Funktionalität von `VTC_NextVTButtonPressed()` aus den ISOBUS-Treiberbeispielen und ermöglicht eine nahtlose Umschaltung zwischen mehreren VT-Geräten.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Datentyp	Mit Variablen	Beschreibung
INIT	EInit	QI	Service-Initialisierung. Wird ausgelöst, um den Baustein zu konfigurieren.
REQ	Event	QI	Service-Anforderung. Startet den Vorgang zum Umschalten zum nächsten verfügbaren VT.

Ereignis-Ausgänge

Ereignis	Datentyp	Mit Variablen	Beschreibung
INITO	EInit	QO, STATUS	Bestätigung der Initialisierung. Signalisiert, ob die Initialisierung erfolgreich war.
CNF	Event	QO, STATUS, s16Result	Bestätigung des Umschaltvorgangs. Wird ausgegeben, sobald der Wechsel zum nächsten VT initiiert wurde.

Daten-Eingänge

Variable	Typ	Beschreibung
QI	BOOL	Qualitätseingang. TRUE aktiviert den Dienst. Bei FALSE werden keine Aktionen ausgeführt.

Daten-Ausgänge

Variable	Typ	Beschreibung
QO	BOOL	Qualitätsausgang. TRUE, wenn der Dienst erfolgreich ausgeführt wurde.
STATUS	STRING	Statusmeldung zum Ergebnis des letzten Vorgangs.
s16Result	INT	Rückgabewert: 0 = E_NO_ERR (erfolgreich), negativer Wert = Fehlercode.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Block führt beim Empfang eines **REQ**-Ereignisses folgende Schritte durch:

1. **Liste aller VTs ermitteln** – Mittels `IsoClientsReadListofExtHandles()` werden alle im ISOBUS-Netzwerk verfügbaren Virtual Terminals erfasst.
2. **Aktuell verbundenes VT finden** – Über `IsoVtcGetStatusInfo(VT_HND)` wird das aktuell genutzte VT identifiziert.
3. **Nächstes VT bestimmen** – Aus der Liste wird der nächste Eintrag nach dem aktuellen VT ausgewählt (zyklische Reihenfolge).
4. **Umschaltung durchführen** – Der Wechsel zum nächsten VT wird mittels `IsoVtcMultipleNextVT()` angestoßen.

- **Fehlerfall:** Falls nur ein VT am Bus vorhanden ist oder der Wechsel fehlschlägt, wird ein negativer Rückgabewert (`s16Result`) sowie eine entsprechende Statusmeldung ausgegeben.
- **Wichtiger Hinweis:** Die Anwendung muss nach dem Aufruf dieses Blocks in einen sicheren Zustand übergehen, da die Verbindung zum aktuellen VT während des Übergangs verloren geht.

Technische Besonderheiten

- **Standardkonformität:** Der Block basiert auf ISO 11783-6 (ISOBUS) und implementiert die VT-Umschaltfunktion gemäß dem Standard.
- **Service-Interface-Block:** Die Ausführung erfolgt asynchron; das Ereignis **CNF** signalisiert den Abschluss des Umschaltvorgangs.
- **Initialisierung:** Vor der ersten Nutzung muss der Block über das **INIT**-Ereignis initialisiert werden, um interne Ressourcen zu belegen.

Zustandsübersicht

Der Baustein besitzt keine expliziten Zustände in der XML-Definition. Das Verhalten wird durch die ereignisgesteuerte Abarbeitung bestimmt:

- **Initialisierungsphase:** Nach **INIT** mit `QI=TRUE` wird die Initialisierung durchgeführt, quittiert durch **INITO**.
- **Betriebsphase:** Nach **REQ** mit `QI=TRUE` wird der Umschaltvorgang gestartet. Nach Abschluss erfolgt das Ereignis **CNF**.
- **Fehlerbehandlung:** Bei Fehlern werden `QO=FALSE` und ein entsprechender Fehlercode in `s16Result` gesetzt.

Anwendungsszenarien

- **ISOBUS-Maschinenbedienung:** Wechsel zwischen mehreren Virtual Terminals in landwirtschaftlichen Fahrzeugen, um z. B. zwischen Display und Bordcomputer umzuschalten.
- **Multimonitor-Systeme:** Umschaltung der aktiven Anzeige auf ein anderes VT-Gerät im Netzwerk.
- **Diagnose und Test:** Simulieren des Druckes der „Next VT“-Taste während der Entwicklung von ISOBUS-konformen Anwendungen.

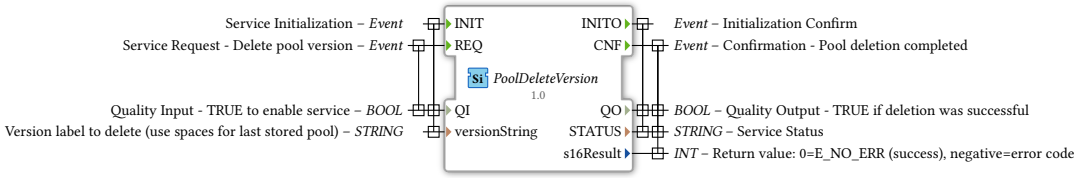
Vergleich mit ähnlichen Bausteinen

- **VTSelection** (hypothetisch): Ein Block, der ein bestimmtes VT-Objekt auswählt, anstatt einfach zum nächsten zu springen. NextVT ist einfacher und folgt der Standardtastenlogik.
- **VTConnect** (hypothetisch): Stellt eine direkte Verbindung zu einem benannten VT her. NextVT hingegen arbeitet automatisch und benötigt keine Zieladresse.

Fazit

Der **NextVT**-Funktionsblock ist ein spezialisierter Service-Interface-Block für ISOBUS-Anwendungen, der die standardkonforme Umschaltung zum nächsten verfügbaren Virtual Terminal ermöglicht. Durch die einfache Schnittstelle (INIT, REQ) und die klare Fehlerrückmeldung eignet er sich hervorragend für die Integration in Steuerungssoftware, die eine flexible VT-Auswahl benötigt. Die Anwendung muss jedoch die kurze Verbindungsunterbrechung berücksichtigen.

PoolDeleteVersion



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Service Interface Block `PoolDeleteVersion` löscht eine gespeicherte Objekt-Pool-Version aus dem nichtflüchtigen Speicher eines Virtual Terminals (VT). Er kapselt die Funktion `VTC_PoolDeleteVersion()` aus dem ISOBUS-Treiber. Der Block ruft `IsoVtcCmd_DeleteVersion()` auf, um die angegebene Pool-Version zu entfernen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Service-Initialisierung
REQ	Event	Service-Anforderung – Pool-Version löschen

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Initialisierungsbestätigung
CNF	Event	Bestätigung – Löschvorgang abgeschlossen

Daten-Eingänge

Name	Typ	Kommentar	Initialwert
QI	BOOL	Qualitätseingang – TRUE aktiviert den Dienst	
versionString	STRING	Zu löschende Versionsbezeichnung (Leerzeichen für letzte gespeicherte Version)	' '

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Qualitätsausgang – TRUE bei erfolgreichem Löschvorgang
STATUS	STRING	Dienststatus
s16Result	INT	Rückgabewert: 0 = E_NO_ERR (Erfolg), negativ = Fehlercode

Adapter

Keine.

Funktionsweise

Der Block wird über den Ereigniseingang `INIT` initialisiert, wobei die zu löschende Versionsbezeichnung (`versionString`) übergeben wird. Nach erfolgreicher Initialisierung löst ein `REQ`-Ereignis den Löschvorgang aus. Der Block ruft die systemnahe Funktion `IsoVtcCmd_DeleteVersion()` auf, die die angegebene Pool-Version aus dem Flash-Speicher des VT entfernt. Das Ergebnis wird über die Ausgänge `QO`, `STATUS` und `s16Result` signalisiert und ein entsprechendes Bestätigungsereignis (`CNF` oder `INITO`) ausgelöst.

Technische Besonderheiten

- Der `versionString` muss nullterminiert sein oder genau 32 Byte lang sein (ISO V11+).
- Wird ein String aus Leerzeichen (`'  '`) übergeben, löscht die Funktion die zuletzt gespeicherte Pool-Version.
- Maximale Länge des Versionseintrags: 32 Byte.
- Der Block ist als Service Interface Block (SIB) für die ISOBUS-Kommunikation (ISO 11783-6) ausgelegt.

Zustandsübersicht

Der Block besitzt keine explizit in der XML definierten Zustände. Die Steuerung erfolgt ereignisgesteuert über `INIT` und `REQ`:

1. **Initialisierung** (INIT) ⟶ Bestätigung (INITO)
2. **Anforderung** (REQ) ⟶ Bestätigung (CNF) mit Ergebnis

Fehlerhafte Initialisierung oder fehlgeschlagene Löschung werden durch `QO=FALSE` und entsprechende `STATUS`-Meldung signalisiert.

Anwendungsszenarien

- Löschen alter Pool-Versionen vor dem Hochladen einer neuen Version
- Freigeben von Flash-Speicherplatz auf dem VT
- Entfernen beschädigter oder nicht mehr benötigter Pool-Versionen
- Zurücksetzen auf die werkseitig gespeicherte Version durch Löschen aller benutzerdefinierten Versionen

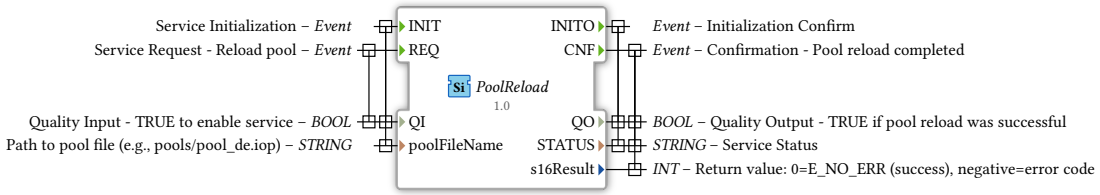
Vergleich mit ähnlichen Bausteinen

Es liegt kein direkter Vergleich zu anderen Funktionsblöcken vor. Der Block ist spezifisch für die Verwaltung von ISOBUS-Objektpools in VT-Flash-Speichern.

Fazit

`PoolDeleteVersion` bietet eine standardisierte und zuverlässige Möglichkeit, gespeicherte Objektpool-Versionen aus dem Flash-Speicher eines Virtual Terminals zu löschen. Durch die einfache ereignisgesteuerte Schnittstelle lässt er sich gut in Automatisierungs- und Steuerungsanwendungen integrieren. Die Berücksichtigung der ISO-Vorgaben (Version String-Länge, Leerzeichen für letzte Version) macht den Block besonders robust für den Einsatz in landwirtschaftlichen Maschinen nach ISO 11783.

PoolReload



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `PoolReload` ist ein Service-Interface-Baustein gemäß ISO 11783-6 (ISOBUS). Er ermöglicht das Nachladen oder Aktualisieren des Objektpools eines Virtual Terminals (VT) während der Laufzeit der Applikation. Typischerweise wird dieser Baustein eingesetzt, um z. B. zwischen verschiedenen Sprachvarianten umzuschalten oder geänderte Pool-Dateien dynamisch einzuspielen, ohne das Gesamtsystem neu starten zu müssen.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar
`INIT`	`EInit`	Initialisierung des Dienstes (mit den Parametern `QI` und `poolFileName`)
`REQ`	`Event`	Dienstaufforderung – führt das Nachladen des Pools aus (mit `QI`)

Ereignis-Ausgänge

Ereignis	Typ	Kommentar
`INITO`	`EInit`	Bestätigung der Initialisierung (gibt `QO` und `STATUS` aus)
`CNF`	`Event`	Bestätigung – Pool-Nachladen abgeschlossen (gibt `QO`, `STATUS` und `s16Result` aus)

Daten-Eingänge

Name	Typ	Kommentar
`QI`	`BOOL`	Quality Input: `TRUE` schaltet den Dienst aktiv
`poolFileName`	`STRING`	Pfad zur Pool-Datei (z. B. `pools/pool_de.iop`)

Daten-Ausgänge

Name	Typ	Kommentar
`QO`	`BOOL`	Quality Output: `TRUE`, wenn das Nachladen erfolgreich war
`STATUS`	`STRING`	Dienststatus (z. B. Fehlermeldung oder Erfolgsmeldung)
`s16Result`	`INT`	Rückgabewert: `0` = `E_NO_ERR` (Erfolg), negative Werte entsprechen Fehlercodes

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Baustein kapselt die ISOBUS-Funktion `VTC_PoolReload()`. Der Ablauf gliedert sich in folgende Schritte:

1. **Initialisierung (`INIT`)**
   - Die Pool-Daten werden aus der unter `poolFileName` angegebenen Datei geladen.
   - Der Pool wird für die konfigurierte Farbtiefe geöffnet.
   - Nach erfolgreichem Laden liefert der Baustein eine Bestätigung über `INITO`.
2. **Dienstausführung (`REQ`)**
   - Der Baustein ruft `IsoVtcPoolUpdate()` auf, um den Pool auf dem VT zu aktualisieren.
   - Optional können ID-Bereichsmodi für die Pool-Manipulation angewendet werden.
   - Nach Abschluss des Vorgangs wird ein `CNF`-Ereignis ausgelöst, das den Erfolg oder Fehler (über `s16Result`) meldet.

Technische Besonderheiten

- **Standardkonformität**: Der Baustein folgt der ISOBUS-Norm ISO 11783-6 (Landwirtschaftliche Fahrzeuge – Virtual Terminal).
- **Dateipfad**: Standardmäßig wird die Pool-Datei unter `pools/pool_de.iop` erwartet. Der Pfad kann jedoch über den Eingang `poolFileName` konfiguriert werden.
- **Laufzeitaktualisierung**: Im Gegensatz zu einem statischen Pool-Import erlaubt dieser FB ein dynamisches Update, ohne die Applikation neu starten zu müssen.
- **Fehlerbehandlung**: Der Ausgang `s16Result` gibt den detaillierten ISOBUS-Fehlercode zurück (0 = Erfolg, negative Werte = Fehler).

Zustandsübersicht

Der Baustein kann folgende grundlegende Zustände durchlaufen:

Zustand	Beschreibung
IDLE	Warten auf ein INIT-Ereignis.
INIT_PENDING	Initialisierung wird ausgeführt; nach Abschluss wird `INIT0` gesendet.
READY	Nach erfolgreicher Initialisierung bereit für `REQ`.
REQ_PENDING	Pool-Nachladen läuft; nach Abschluss wird `CNF` gesendet.
ERROR	Bei fehlgeschlagener Initialisierung oder Nachladen wird ein Fehlerstatus gemeldet und der Baustein verharrt im Fehlerzustand, bis ein erneutes INIT erfolgt.

Anwendungsszenarien

- **Sprachumschaltung**: Während der Laufzeit kann zwischen verschiedenen Sprachvarianten (z. B. Deutsch, Englisch) durch Austausch der Pool-Datei umgeschaltet werden.
- **Dynamische Pool-Updates**: Neue Masken oder Symbole können eingespielt werden, ohne das gesamte VT neu zu starten – nützlich für Firmware-Updates oder Anpassungen im Feld.
- **Last-Minute-Änderungen**: Während der Inbetriebnahme können geänderte Pool-Dateien schnell geladen werden.

Vergleich mit ähnlichen Bausteinen

Baustein	Beschreibung
`PoolLoader`	Lädt den Pool nur beim Systemstart; kein Nachladen zur Laufzeit.
`PoolActivate`	Schaltet zwischen bereits geladenen Pools um, erfordert aber vorheriges Laden.
`PoolReload`	Vereint Laden und Aktualisieren in einem Schritt und ermöglicht dynamisches Nachladen während der Laufzeit.

Fazit

Der `PoolReload`-Funktionsblock ist ein spezialisierter Service-Interface-Baustein für ISOBUS-Virtual-Terminals, der das flexible Nachladen und Aktualisieren von Objektpools zur Laufzeit ermöglicht. Er vereinfacht Sprachumschaltungen, dynamische Updates und erleichtert die Wartung von VT-Anwendungen. Durch die klare Ereignisschnittstelle und die detaillierte Fehlerrückmeldung eignet er sich gut für den Einsatz in landwirtschaftlichen Steuerungssystemen.

2.1.13.5 pgn

ADDRESS

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ADDRESS` ist ein globaler Konstantenblock, der spezielle Adresskonstanten für die Verwendung in ISOBUS-basierten Anwendungen definiert. Er stellt vordefinierte Werte für Quell- und Zieladressen (Source Address, SA) bereit, die in der ISOBUS-Kommunikation eine besondere Bedeutung haben. Dieser Block dient dazu, den Quellcode lesbarer und wartbarer zu machen, indem magische Zahlen durch aussagekräftige Namen ersetzt werden.

Schnittstellenstruktur

Dieser Block ist ein `GlobalConstants`-Block und besitzt daher keine klassischen Ereignis- oder Datenschnittstellen. Stattdessen definiert er Konstanten, die global in anderen Funktionsblöcken des Pakets `isobus::pgn` verwendet werden können.

Ereignis-Eingänge

Nicht vorhanden.

Ereignis-Ausgänge

Nicht vorhanden.

Daten-Eingänge

Nicht vorhanden.

Daten-Ausgänge

Nicht vorhanden.

Adapter

Nicht vorhanden.

Funktionsweise

Der Block `ADDRESS` deklariert zwei Konstanten:

1. `NULL_A` (Typ: `BYTE`, Wert: `254`): Diese Konstante repräsentiert die NULL-Adresse (`0xFE`). Nachrichten mit dieser Quelladresse (SA) dürfen nur von einem Sender gesendet werden, der über einen Netzwerk-Manager verfügt.
2. `GLOBAL_A` (Typ: `INT`, Wert: `255`): Diese Konstante repräsentiert die globale Adresse (`0xFF`). Diese Adresse wird ausschließlich im Zieladressfeld (Destination Address Field) verwendet, um eine Nachricht an alle Teilnehmer im Netzwerk zu broadcasten.

Diese Konstanten werden zur Compile-Zeit in den Code eingefügt und können nicht zur Laufzeit geändert werden.

Technische Besonderheiten

- **Typunterschied:** Auffällig ist, dass `NULL_A` als `BYTE` und `GLOBAL_A` als `INT` deklariert ist. Beide Werte liegen jedoch im darstellbaren Bereich beider Datentypen. Die Verwendung von `INT` für `GLOBAL_A` könnte auf eine spätere Erweiterbarkeit oder Kompatibilität mit anderen Teilen des Systems hindeuten.
- **Paketzuordnung:** Der Block ist Teil des Pakets `isobus::pgn`, was auf seine Verwendung in der Parametergruppennummer (PGN)-Kommunikation des ISOBUS-Protokolls hindeutet.

Zustandsübersicht

Da es sich um einen Konstantenblock handelt, besitzt er keinen Zustandsautomaten und keine internen Zustände.

Anwendungsszenarien

Der `ADDRESS`-Block wird in ISOBUS-Anwendungen verwendet, beispielsweise in der Landtechnik (Traktoren, Anbaugeräte). Typische Szenarien sind:

- **Konfiguration von Nachrichten:** Beim Aufbau von PGN-Nachrichten (Parameter Group Number) werden diese Konstanten verwendet, um das Quell- oder Zieladressfeld zu belegen.
- **Netzwerkmanagement:** Die `NULL_A`-Adresse wird in speziellen Netzwerkmanagement-Nachrichten verwendet.
- **Broadcast-Kommunikation:** Die `GLOBAL_A`-Adresse wird genutzt, um Befehle oder Daten an alle angeschlossenen Geräte im ISOBUS-Netz zu senden.

⚖️ Vergleich mit ähnlichen Bausteinen

In der IEC 61499-Standardbibliothek gibt es keinen direkten äquivalenten Baustein, da globale Konstanten eine sprachspezifische Erweiterung (hier abgeleitet von Structured Text) innerhalb der 4diac-IDE sind. Ein ähnlicher Zweck könnte erreicht werden durch:

- **Direkte Werteingabe (Magic Numbers):** Das direkte Schreiben der Werte `254` oder `255` im Code ist möglich, aber schlechter Stil, da es die Lesbarkeit und Wartbarkeit beeinträchtigt.
- **Konstanten-Blöcke in anderen Sprachen:** In Funktionsblock-Programmiersprachen wie CFC oder SFC werden vergleichbare Konstanten oft in separaten Header-Dateien oder globalen Variablenlisten definiert.

Der Vorteil des `ADDRESS`-Blocks liegt in der zentralen, paketgebundenen Definition und der semantischen Klarheit.

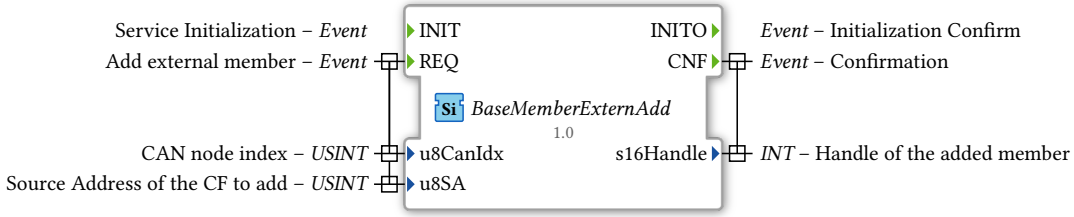
Fazit

Der `ADDRESS`-Funktionsblock ist ein einfacher, aber essentieller Hilfsbaustein für die Entwicklung von ISOBUS-konformen Applikationen in der 4diac-IDE. Er erhöht die Codequalität, indem er numerische Spezialwerte durch selbsterklärende Bezeichner ersetzt und so Fehler vermeidet sowie die Wartung erleichtert. Seine Verwendung ist insbesondere bei der Implementierung von Netzwerkmanagement und Broadcast-Funktionalitäten empfohlen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

BaseMemberExternAdd



Adds an external member to the network list manually.

The use of this function is not ISO 11783 Part 5 conform. Use this function only if you must communicate with an external CF, that does not send a address claimed message. An address claimed message of an external CF with the same source address will replace this manual added member entry. The function works independent from the configuration of the #CFG_PARAM_EXTERNAL_ADDR_CLAIM parameter. The added CF will be set to announced in the next iso_DlCyclicGlobal() run.

Interface

Event Inputs

Name	Comment	With
INIT	Service Initialization	
REQ	Add external member	u8SA, u8CanIdx

Event Outputs

Name	Comment	With
INITO	Initialization Confirm	
CNF	Confirmation	s16Handle

Input Vars

Name	Type	Initial Value	Comment
u8CanIdx	USINT	NODE1	CAN node index
u8SA	USINT		Source Address of the CF to add

Output Vars

Name	Type	Comment
s16Handle	INT	Handle of the added member

🛠️ Zugehörige Übungen

- Uebung_134

CAN_MSG

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Datentyp `CAN_MSG` ist eine strukturierte Typdefinition (Struct) zur Repräsentation eines klassischen CAN-Nachrichtenrahmens (ohne Transportprotokoll, TP). Er dient als Container für die wesentlichen Attribute einer CAN-Nachricht, einschließlich Priorität, Nutzdatenlänge und dem eigentlichen Datenpuffer. Dieser Typ ist für den Einsatz in Funktionsblöcken vorgesehen, die CAN-Kommunikation in einem IEC 61499-System implementieren.

Schnittstellenstruktur

Da es sich bei `CAN_MSG` um einen Datentyp und nicht um einen Funktionsblock handelt, existieren keine Ereignis- oder Adapterschnittstellen. Die Struktur wird ausschließlich über ihre Datenmember definiert.

Daten-Eingänge

(Nicht anwendbar – Datentyp)

Daten-Ausgänge

(Nicht anwendbar – Datentyp)

Ereignis-Eingänge

(Nicht anwendbar – Datentyp)

Ereignis-Ausgänge

(Nicht anwendbar – Datentyp)

Adapter

(Nicht anwendbar – Datentyp)

Funktionsweise

Der `CAN_MSG`-Typ ist eine passive Datenstruktur. Er kapselt die für eine CAN-Nachricht relevanten Informationen:

1. **Priorität (`u8Priority`)**: Definiert die Arbitrierungspriorität auf dem CAN-Bus. Ein niedrigerer numerischer Wert entspricht einer höheren Priorität.
2. **Datenlänge (`u16DaSize`)**: Gibt die tatsächliche Anzahl der genutzten Bytes im Datenpuffer an (0 bis 8).
3. **Datenpuffer (`data`)**: Ein Array von 8 Bytes, das die Nutzdaten der Nachricht speichert. Nicht genutzte Bytes sind mit einem Standardwert initialisiert.

Diese Struktur kann als Eingabe- oder Ausgabedatentyp für Funktionsblöcke verwendet werden, die CAN-Nachrichten senden, empfangen, verarbeiten oder loggen.

Technische Besonderheiten

- **Feste Puffergröße**: Das Datenarray ist statisch auf eine Länge von 8 Bytes dimensioniert, was der maximalen Nutzdatenlänge eines klassischen CAN-Frames entspricht.
- **Initialisierung**: Das Datenarray wird standardmäßig mit dem Hexadezimalwert `16#FF` (dezimal 255) in allen 8 Positionen vorbelegt. Die Datenlänge `u16DaSize` ist standardmäßig `0`, und die Priorität `u8Priority` ist standardmäßig `7` (niedrigste Priorität).
- **Typensicherheit**: Die Verwendung eines strukturierten Typs anstelle einzelner Variablen erhöht die Wartbarkeit und Klarheit in der Funktionsblock-Verbindungsansicht.

Zustandsübersicht

(Nicht anwendbar – Datentyp besitzt kein Verhalten oder Zustandsautomaten.)

Anwendungsszenarien

Der `CAN_MSG`-Typ ist grundlegend für:

- **CAN-Sender-FBs**: Ein FB, der einen `CAN_MSG`-Typ als Eingang erhält und dessen Inhalt als physikalischen CAN-Frame auf den Bus sendet.
- **CAN-Empfänger-FBs**: Ein FB, der empfangene CAN-Frames in eine Variable vom Typ `CAN_MSG` packt und an nachgelagerte Logik weiterreicht.
- **Nachrichten-Konstruktion**: FBs, die aus einzelnen Werten (z.B. aus Sensor-FBs) eine gültige `CAN_MSG`-Struktur für die spätere Übertragung aufbauen.
- **Nachrichten-Filterung/Verarbeitung**: FBs, die eingehende `CAN_MSG`-Strukturen basierend auf Priorität, Dateninhalt oder Länge analysieren und verarbeiten.

⚖️ Vergleich mit ähnlichen Bausteinen

Als Datentyp ist `CAN_MSG` mit anderen strukturierten Typen in 4diac vergleichbar, die Kommunikationsprotokolle abbilden (z.B. `UDP_PACKET`, `MODBUS_MSG`). Sein spezifisches Merkmal ist die enge Abbildung auf die Hardware-Eigenschaften des CAN-Bus-Protokolls (Priorität, 8-Byte-Limit). Im Gegensatz zu einem TP-basierten Nachrichtentyp (z.B. für ISO 11783 oder CANopen) fehlen hier Felder für Paketierung, Sequenznummern oder spezifische Protokollkopfzeilen.

Fazit

Der `CAN_MSG`-Datentyp stellt eine essentielle und gut strukturierte Grundlage für die Implementierung von CAN-Bus-Kommunikation in 4diac-basierten Steuerungssystemen dar. Durch die Bündelung von Priorität, Länge und Daten in einem einzigen Typ wird die Handhabung von CAN-Nachrichten in Funktionsblock-Netzwerken deutlich vereinfacht und standardisiert. Seine Verwendung ist für alle Anwendungen empfohlen, die den direkten Zugriff auf CAN-Frames erfordern.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

CF_INFO_T

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Datentyp `CF_INFO_T` (Control Function Information Type) ist eine strukturierte Datenkomponente, die innerhalb der ISO 11783 (ISOBUS)-Kommunikation verwendet wird. Er dient der standardisierten Abbildung und Verwaltung von Informationen über einen Teilnehmer (Control Function oder User) im ISOBUS-Netzwerk. Dieser Typedef fasst alle wesentlichen Merkmale und Zustände eines Netzwerkteilnehmers in einer einzigen Datenstruktur zusammen.

Schnittstellenstruktur

`CF_INFO_T` ist ein strukturierter Datentyp (StructuredType) und verfügt über keine Ereignis- oder Adapterschnittstellen im herkömmlichen Sinne eines Funktionsbausteins. Seine Struktur besteht ausschließlich aus Daten-Elementen (Variablen).

Daten-Eingänge

Nicht vorhanden, da es sich um einen reinen Datentyp handelt.

Daten-Ausgänge

Nicht vorhanden, da es sich um einen reinen Datentyp handelt.

Daten-Elemente (Struktur)

Die interne Struktur des Typs `CF_INFO_T` besteht aus den folgenden Elementen:

Name	Typ	Kommentar / Beschreibung	Initialwert

s16Handle	INT	Netzwerk-Member-Handle der Control Function (CF) / des Users. Dient als eindeutiger Identifikator innerhalb der Anwendung.	UNVALID
u8SourceAddress	BYTE	Quelladresse (Source Address, SA) der Control Function (CF) / des Users im ISOBUS-Netzwerk.	NULL_A
u8CanIdx	BYTE	CAN-Bus Node / Index der Control Function (CF) / des Users. Identifiziert den physischen Bus-Knoten.	–
au8Name	isobus::pgn::CF_NAME_T	Der ISOBUS-NAME der Control Function. Dies ist eine 64-bit eindeutige Kennung gemäß ISO 11783-5.	–
eIsoUserFunct	SINT	Intern verwendeter CF-Funktionstyp (z.B. Arbeitsgerät, Terminal, Task-Controller).	undefined
bExternalCf	BOOL	Kennzeichnet, ob es sich um eine externe CF im CAN-Netzwerk handelt (TRUE) oder nicht.	–
bIsAnnounced	BOOL	TRUE, wenn die CF aktiv und vollständig im Netzwerk angekündigt (announced) ist.	–
bIsActive	BOOL	NIU-spezifisch (Network Interface Unit): TRUE, wenn sich die CF nicht in einem Fehlerzustand befindet oder abgemeldet (logged off) ist.	–

Funktionsweise

Der Datentyp CF_INFO_T ist eine passive Datenhaltungsstruktur. Er wird verwendet, um den Zustand und die Identität eines ISOBUS-Netzwerkteilnehmers zu speichern und zwischen verschiedenen Komponenten einer Steuerungsanwendung auszutauschen. Die enthaltenen Informationen werden typischerweise durch andere Funktionsbausteine gelesen, geschrieben und interpretiert, die für die Netzwerkverwaltung (z.B. Adressvergabe, NAME-Verwaltung) oder die Applikationslogik zuständig sind.

Technische Besonderheiten

- **Typedef für ISOBUS:** Speziell für die Anwendung in ISO 11783-Umgebungen (Landtechnik) konzipiert.
- **Zustandsflags:** Enthält mehrere boolesche Flags (bIsAnnounced, bIsActive), die den Lebenszyklus und Betriebszustand einer CF abbilden.
- **Initialwerte:** Kritische Identifikatoren wie s16Handle und u8SourceAddress werden mit definierten Konstanten (UNVALID, NULL_A) initialisiert, um einen ungültigen Ausgangszustand sicherzustellen.
- **Integration:** Verwendet den benutzerdefinierten Typ isobus::pgn::CF_NAME_T für die Abbildung des 64-bit ISOBUS-NAMEN.

Zustandsübersicht

Da es sich um einen Datentyp und keinen aktiven Baustein handelt, existiert keine Zustandsmaschine. Der „Zustand“ wird durch die Kombination der enthaltenen Datenfelder beschrieben, insbesondere durch die Flags:

- **Ungültig/Inaktiv:** s16Handle == UNVALID, bIsAnnounced == FALSE
- **Gültig & Aktiv:** s16Handle != UNVALID, bIsAnnounced == TRUE, bIsActive == TRUE
- **Gültig & Fehlerhaft/Inaktiv:** s16Handle != UNVALID, bIsActive == FALSE

Anwendungsszenarien

- **Teilnehmerverwaltung:** In einem ISOBUS-Manager-Baustein, der eine Liste aller bekannten Netzwerkteilnehmer (`CF_INFO_T[]`) verwaltet.
- **Diagnose und Anzeige:** Weitergabe von Teilnehmerinformationen an ein Diagnose- oder Display-System, um den Netzwerkstatus anzuzeigen.
- **Kommando-Weiterleitung:** Als Parameter für Kommunikationsbausteine, um Nachrichten gezielt an eine bestimmte CF (identifiziert durch `u8SourceAddress` und `au8Name`) zu adressieren.
- **Filterung:** Logikbausteine können anhand der Felder `eIsoUserFunct` oder `bExternalCf` entscheiden, ob Nachrichten einer bestimmten CF verarbeitet werden sollen.

⚖️ Vergleich mit ähnlichen Bausteinen

`CF_INFO_T` ist kein Funktionsbaustein und kann daher nicht direkt mit Blöcken wie `E_SR` oder `E_CYCLE` verglichen werden. Innerhalb der ISOBUS-Kommunikation ist er vergleichbar mit anderen strukturierten Datentypen, die Netzwerk- oder Geräteinformationen bündeln (z.B. `PGN_INFO_T` für Parameter Group Numbers). Seine Besonderheit liegt in der spezifischen Zusammenstellung von Daten, die für die Identifikation und das Zustandsmanagement einer Control Function erforderlich sind.

Fazit

Der Datentyp `CF_INFO_T` ist ein grundlegender und wesentlicher Bestandteil für die Implementierung von ISOBUS-fähigen Steuerungsapplikationen in 4diac. Er bietet eine standardisierte, typsichere Methode, um alle relevanten Informationen eines Netzwerkteilnehmers zu gruppieren und zu transportieren. Durch seine Verwendung wird die Wiederverwendbarkeit von Code erhöht und die Konsistenz in der Netzwerkkommunikationslogik sichergestellt.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

CF_NAME_T

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Datentyp `CF_NAME_T` ist ein strukturierter Datentyp, der in der 4diac-IDE zur Modellierung von ISO CF (Communication Function) oder Anwenderinformationsdaten-Arrays verwendet wird. Er dient als Typedef für ein Byte-Array fester Länge und wird typischerweise im Kontext von ISOBUS PGN (Parameter Group Number) Kommunikation eingesetzt.

Schnittstellenstruktur

`CF_NAME_T` ist ein strukturierter Datentyp (STRUCT) und besitzt keine Ereignis- oder Adapterschnittstellen. Seine Struktur besteht aus einem einzelnen Datenfeld.

Daten-Eingänge

Dieser Datentyp selbst hat keine Eingänge. Er definiert die Struktur für Daten.

Daten-Ausgänge

Dieser Datentyp selbst hat keine Ausgänge. Er definiert die Struktur für Daten.

Strukturierte Elemente

- **`data`** (BYTE[0..7]): Ein Array aus genau 8 Bytes (`BYTE`). Jedes Element im Array wird bei der Initialisierung standardmäßig auf den Hexadezimalwert `16#FF` (dezimal 255) gesetzt. Dies repräsentiert einen typischen „ungültigen“ oder „leeren“ Anfangszustand in Kommunikationsprotokollen.

Funktionsweise

Der Datentyp `CF_NAME_T` kapselt ein Byte-Array mit einer festen Länge von 8 Bytes. Seine Hauptfunktion besteht darin, eine typsichere und wiederverwendbare Definition für Datenpakete bereitzustellen, die in ISOBUS-basierten Steuerungssystemen ausgetauscht werden. Die vordefinierte Initialisierung aller Bytes auf `0xFF` stellt sicher, dass Instanzen dieses Typs in einem bekannten, neutralen Zustand beginnen.

Technische Besonderheiten

- **Feste Array-Größe:** Das Array ist auf die Indizes 0 bis 7 begrenzt, was einer festen Länge von 8 Bytes entspricht. Dies ist eine gängige Größe für viele Nachrichtenblöcke in Feldbussystemen.
- **Vordefinierte Initialisierung:** Alle Array-Elemente werden bei der Deklaration einer Variable dieses Typs automatisch mit dem Wert `16#FF` initialisiert, sofern nicht anders angegeben.
- **ISOBUS-Kontext:** Der Typ ist Teil des Pakets `isobus::pgn`, was auf seine spezifische Verwendung in landwirtschaftlichen und mobilen Maschinensteuerungen (ISOBUS, ISO 11783) hindeutet.

Anwendungsszenarien

Dieser Datentyp wird primär verwendet, um die Nutzdaten von ISOBUS PGN-Nachrichten innerhalb von Funktionsbausteinen (FBs) zu repräsentieren. Typische Anwendungen sind:

- Deklaration von Eingangs- oder Ausgangsvariablen in FBs, die ISOBUS-Nachrichten senden oder empfangen.
- Strukturierung von Daten für die Kommunikation zwischen verschiedenen Steuergeräten (ECUs) in einer mobilen Maschine.
- Verwendung als Teil eines größeren, strukturierten Datentyps, der mehrere Parameter einer komplexen Nachricht kombiniert.

⚖️ Vergleich mit ähnlichen Bausteinen

`CF_NAME_T` ist ein Datentyp und kein Funktionsbaustein. Er kann mit anderen strukturierten Datentypen (STRUCT) in 4diac verglichen werden, die Arrays definieren. Sein Unterscheidungsmerkmal ist die feste 8-Byte-Länge und die ISOBUS-spezifische Semantik. Ein generisches `ARRAY [0..7] OF BYTE` wäre funktional ähnlich, bietet aber nicht die typsichere Wiederverwendbarkeit und den kontextuellen Bezug.

Fazit

Der Datentyp `CF_NAME_T` ist ein spezialisiertes, wiederverwendbares Konstrukt für die 4diac-IDE, das die Handhabung von 8-Byte-Datenblöcken in ISOBUS-Anwendungen standardisiert und vereinfacht. Durch seine feste Größe und vordefinierte Initialisierung erhöht er die Zuverlässigkeit und Lesbarkeit von Applikationscode, der auf dem ISO 11783-Standard basiert.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

CommunicationPartners

• – ◦ • – ◦ • – ◦ •

Einleitung

`CommunicationPartners` ist ein globaler Konstanten-Datensatz, der im Kontext der ISOBUS PGN-Kommunikation (Parameter Group Number) verwendet wird. Er dient dazu, einen vordefinierten Kommunikationspartner, in diesem Fall die globale Adresse, als konstante Konfiguration für andere Funktionsblöcke bereitzustellen. Dies vereinfacht die Konfiguration von ISOBUS-Netzwerkkomponenten, indem häufig benötigte Adressen zentral und wiederverwendbar definiert werden.

Schnittstellenstruktur

Da es sich um einen globalen Konstanten-Datensatz handelt, besitzt `CommunicationPartners` keine ereignisgesteuerten oder datenflussbasierten Schnittstellen im herkömmlichen Sinne eines Funktionsblocks. Stattdessen stellt er eine benannte, konstante Variable für die gesamte Applikation bereit.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Adapter

Keine.

Funktionsweise

Der Datensatz `CommunicationPartners` deklariert eine globale Konstante namens `GLOBAL` vom Datentyp `isobus::pgn::CF_INFO_T`. Diese Struktur wird mit einem Initialwert belegt, der das Feld `s16Handle` auf den vordefinierten Wert `GLOBAL_A` aus der importierten Konstante `isobus::pgn::ADDRESS::GLOBAL_A` setzt.

Bei der Kompilierung und Ausführung der Applikation wird diese Konstante im Speicher abgelegt und kann von jedem Funktionsblock innerhalb des Projekts referenziert werden, der eine ISOBUS-Adresse vom Typ `CF_INFO_T` benötigt. Die Verwendung dieser Konstante stellt sicher, dass die globale Adresse konsistent und ohne Tippfehler im gesamten Code verwendet wird.

Technische Besonderheiten

- **Datentyp:** Die Konstante ist vom Typ `isobus::pgn::CF_INFO_T`. Dies ist vermutlich eine strukturierte (STRUCT) oder Aufzählungstyp (ENUM), die speziell für die ISOBUS-Kommunikationskonfiguration definiert ist.
- **Initialisierung:** Die Initialisierung erfolgt direkt in der Deklaration mittels Strukturinitialisierung `(s16Handle := GLOBAL_A)`.
- **Import:** Der Datensatz importiert eine externe Konstante (`GLOBAL_A`) aus dem Paket `isobus::pgn::ADDRESS`. Dies zeigt eine modulare Architektur, bei der Adressdefinitionen an einer zentralen Stelle gepflegt werden.
- **Kompilierung:** Das `CompilerInfo`-Tag weist darauf hin, dass dieses Element Teil des Pakets `isobus::pgn` ist und für die Codegenerierung relevant ist.

Zustandsübersicht

Da es sich um eine statische, konstante Definition handelt, gibt es keine Zustandsmaschine oder veränderlichen Zustände.

Anwendungsszenarien

- **ISOBUS-Netzwerkkonfiguration:** Wird verwendet, wenn ein Funktionsblock eine Nachricht an alle Teilnehmer im ISOBUS-Netzwerk (Broadcast) senden soll. Die Konstante `GLOBAL` repräsentiert diese Broadcast-Adresse.
- **Vereinfachung der Parametrierung:** In komplexen Applikationen mit vielen kommunizierenden Blöcken ersetzt die Verwendung von `CommunicationPartners.GLOBAL` die wiederholte, hartkodierte Eingabe der globalen Adresse und erhöht so die Wartbarkeit.
- **Referenz für andere Konstanten:** Kann als Basis für die Definition weiterer, spezifischerer Kommunikationspartner-Konstanten dienen (z.B. `IMPLEMENT`, `TOOL`).

⚖️ Vergleich mit ähnlichen Bausteinen

- **Direkte Wertzuweisung:** Anstatt `CommunicationPartners.GLOBAL` zu verwenden, könnte ein Entwickler den Wert `GLOBAL_A` auch direkt in jedem Daten-Eingang eines Funktionsblocks eintragen. Die Verwendung der Konstanten ist jedoch sauberer, typsicherer und änderungsfreundlicher.
- **Andere Adresskonstanten:** Innerhalb des gleichen Pakets (`isobus::pgn::ADDRESS`) existieren vermutlich ähnliche Konstanten für andere Adressen (z.B. `NULL_A`, `TOOL_A`). `CommunicationPartners` kapselt eine spezifische Auswahl oder Kombination dieser Adressen in einem für die Applikation sinnvollen Kontext.
- **Parameter-FBs vs. Globale Konstanten:** Ein alternativer Ansatz wäre ein Funktionsblock, der seine Zieladresse als konfigurierbaren Parameter erhält. Globale Konstanten sind hingegen einfacher und direkter, wenn der Wert für die gesamte Laufzeit der Applikation unveränderlich ist.

Fazit

Der globale Konstanten-Datensatz `CommunicationPartners` ist ein einfaches, aber nützliches Hilfskonstrukt für die Entwicklung von ISOBUS-fähigen Steuerungen in 4diac. Er fördert den Prinzipien des „Don't Repeat Yourself" (DRY) und der zentralen Konfigurationsverwaltung, indem er die häufig benötigte globale ISOBUS-Adresse an einer einzigen, gut sichtbaren Stelle definiert. Seine Verwendung reduziert potenzielle Fehlerquellen und erleichtert zukünftige Anpassungen der Netzwerkkonfiguration.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

HANDLE

• – ◦ • – ◦ • – ◦ •

Einleitung

Die `HANDLE`-Konstante ist eine globale Konstantendefinition für die 4diac-IDE, die im Kontext der ISOBUS-Protokoll- und Parameter-Gruppen-Nummer (PGN) Verarbeitung verwendet wird. Sie definiert spezielle Werte zur Kennzeichnung ungültiger oder globaler Adressierungsziele innerhalb eines ISOBUS-Netzwerks. Diese Konstanten dienen als zentrale Referenzwerte für Funktionsblöcke, die mit Handle-basierter Kommunikation arbeiten.

Schnittstellenstruktur

Dies ist eine globale Konstantendefinition und besitzt daher keine Ereignis- oder Datenschnittstellen im herkömmlichen Sinne eines Funktionsblocks. Stattdessen stellt sie benannte Konstanten zur Verfügung, die in anderen Blöcken verwendet werden können.

Ereignis-Eingänge

Nicht vorhanden. Diese Definition enthält keine Ereignisseingänge.

Ereignis-Ausgänge

Nicht vorhanden. Diese Definition enthält keine Ereignisausgänge.

Daten-Eingänge

Nicht vorhanden. Diese Definition enthält keine Dateneingänge.

Daten-Ausgänge

Nicht vorhanden. Diese Definition enthält keine Datenausgänge.

Adapter

Nicht vorhanden. Diese Definition enthält keine Adapterschnittstellen.

Funktionsweise

Die `HANDLE`-Konstante definiert zwei symbolische Namen für spezielle Integer-Werte:

1. `UNVALID`: Repräsentiert einen ungültigen oder nicht genutzten Handle mit dem Wert `-1` (dezimal) bzw. `0xFFFF` (hexadezimal). Dieser Wert kann verwendet werden, um anzuzeigen, dass ein Handle nicht initialisiert ist, eine Operation fehlgeschlagen ist oder eine Ressource nicht verfügbar ist.
2. `GLOBAL`: Repräsentiert einen Pseudo-Handle mit dem Wert `32767` (dezimal) bzw. `0x7FFF` (hexadezimal). Dieser Wert dient dazu, eine globale Zieladresse (Broadcast) innerhalb des ISOBUS-Netzwerks anzusprechen, anstatt ein spezifisches Einzelgerät.

Diese Konstanten werden typischerweise in den Datenverbindungen oder internen Logiken anderer Funktionsblöcke (z.B. für ISOBUS-Kommunikation) verwendet, um die Code-Lesbarkeit und Wartbarkeit zu verbessern, anstatt magische Zahlen direkt im Code zu verwenden.

Technische Besonderheiten

- **Paketzuordnung:** Die Konstanten sind dem Paket `isobus::pgn` zugeordnet, was ihren Einsatzbereich in der ISOBUS-PGN-Kommunikation klar definiert.
- **Typisierung:** Beide Konstanten sind vom Typ `INT` (Integer).
- **Initialisierung:** Die Werte werden direkt bei der Deklaration zugewiesen (`InitialValue`).
- **Sichtbarkeit:** Als globale Konstanten (`GLOBALCONSTANTS`) sind sie innerhalb des definierten Pakets (`isobus::pgn`) für alle darin enthaltenen Funktionsblöcke sichtbar und verwendbar.

Zustandsübersicht

Da es sich um eine statische Konstantendefinition handelt, gibt es keine Zustandsmaschine oder veränderlichen Zustände.

Anwendungsszenarien

- **Initialisierung von Handles:** Beim Start eines Systems können Kommunikationshandles mit dem Wert `UNVALID` initialisiert werden, um ihren „leeren“ Zustand zu kennzeichnen.
- **Fehlerbehandlung:** Funktionsblöcke, die Handles zurückgeben (z.B. nach dem Öffnen einer Verbindung), können `UNVALID` zurückgeben, um einen Fehler anzuzeigen.
- **Broadcast-Kommunikation:** Ein Senderblock kann den `GLOBAL`-Handle verwenden, um eine Nachricht an alle Teilnehmer im ISOBUS-Netzwerk zu adressieren, anstatt an einen spezifischen Empfänger.
- **Vergleiche:** Die Lesbarkeit von Bedingungen wie `IF currentHandle = UNVALID THEN ...` oder `IF targetHandle = GLOBAL THEN ...` wird deutlich verbessert.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`CONSTANT`-Funktionsblock:** Ein generischer `CONSTANT`-Block kann einen festen Wert ausgeben, ist aber eine Instanz innerhalb eines Funktionsblocknetzwerks. Die `HANDLE`-Globalkonstante ist eine zentrale, paketweite Definition, die nicht instanziiert werden muss und direkt über ihren Namen referenziert wird.
- **Direkte Werteingabe:** Anstatt `UNVALID` oder `GLOBAL` zu verwenden, könnten die numerischen Werte (`-1`, `32767`) direkt in die Blöcke eingegeben werden. Die Verwendung der Konstanten ist jedoch die deutlich bessere Praxis, da sie den Code selbsterklärender und änderungsresistenter macht (Änderungen nur an einer Stelle).

Fazit

Die `HANDLE`-Globalkonstante ist ein einfaches aber wesentliches Hilfselement für die Entwicklung von ISOBUS-konformen Applikationen in 4diac. Sie erhöht die Codequalität, indem sie zwei häufig benötigte, spezielle Handle-Werte mit aussagekräftigen Namen versieht. Ihre Verwendung fördert lesbaren, wartbaren und fehlerrobusten Code in Funktionsblöcken, die mit ISOBUS-PGN-Adressierung arbeiten.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ISONETEVENT_T

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Datentyp `ISONETEVENT_T` ist eine Struktur (Structured Type), die zur Beschreibung eines Netzwerkereignisses innerhalb eines ISOBUS-Netzwerks dient. Er kapselt alle relevanten Informationen, die benötigt werden, um ein Ereignis im Zusammenhang mit der Verwaltung von Netzwerkteilnehmern (Control Functions) zu identifizieren und zu verarbeiten. Dieser Typ wird typischerweise in Callback-Funktionen verwendet, um über Zustandsänderungen im Netzwerk zu informieren.

Schnittstellenstruktur

Die Struktur `ISONETEVENT_T` besteht aus den folgenden sechs Elementen:

Daten-Elemente

- **`eNetEvent`** (Typ: `USINT`, Initialwert: `ISO_NETEVENT_e::Isonet_Last`)
  - **Beschreibung:** Gibt an, welches spezifische Ereignis in der Netzwerk-Mitgliederverwaltung aufgetreten ist (z.B. Verbindungsaufbau, -abbruch, Adressänderung).

- **eMemberRefer** (Typ: SINT, Initialwert: ISOUSERHOME_e::notdef)
  - **Beschreibung:** Identifiziert, welches Netzwerkmitglied von dem Ereignis betroffen ist. Spezifiziert, ob es sich um den Besitzer des Callbacks, eine andere interne oder eine externe Control Function (CF) handelt.
- **s16OwnCfHnd** (Typ: INT)
  - **Beschreibung:** Das CF-Handle des Eintrags, für den der Callback ausgelöst wurde. Dies identifiziert die lokale Repräsentation des betroffenen Netzwerkteilnehmers.
- **s16Handle** (Typ: INT)
  - **Beschreibung:** Das Netzwerk-Mitglieder-Handle der beteiligten Control Function bzw. des Benutzers. Dies ist das Handle der CF, die das Ereignis verursacht hat (Event Source CF Handle).
- **u8SAMember** (Typ: USINT)
  - **Beschreibung:** Die aktuelle Quelladresse (Source Address, SA) des Netzwerkmitglieds zum Zeitpunkt des Ereignisses.
- **cfName** (Typ: isobus::pgn::CF_NAME_T)
  - **Beschreibung:** Der ISOBUS-Name der beteiligten Control Function. Dieser Name ist eine eindeutige Identifikation gemäß ISOBUS-Standard.

Funktionsweise

Die Struktur ISONETEVENT_T fungiert als ein Container oder Nachrichtenobjekt. Wenn ein relevantes Netzwerkereignis eintritt (definiert durch eNetEvent), wird eine Instanz dieser Struktur mit den aktuellen Daten gefüllt und an eine registrierte Callback-Funktion übergeben. Die enthaltenen Daten ermöglichen es der empfangenden Softwarekomponente, das Ereignis genau zu klassifizieren und die betroffenen Netzwerkentitäten zu identifizieren (eMemberRefer, s16OwnCfHnd, s16Handle), deren aktuelle Adresse (u8SAMember) und ihren ISOBUS-Namen (cfName) zu kennen.

Technische Besonderheiten

- **Typisierung:** Es handelt sich um einen StructuredType innerhalb der 4diac IDE.
- **Initialwerte:** Zwei Elemente (eNetEvent, eMemberRefer) besitzen definierte Initialwerte, die auf enumerierte Typen (ISO_NETEVENT_e, ISOUSERHOME_e) verweisen. Dies stellt einen definierten Ausgangszustand sicher.
- **ISOBUS-Integration:** Die Struktur ist spezifisch für den ISOBUS-Kontext entworfen, was sich in der Verwendung des typspezifischen CF_NAME_T und der Parameterbezeichnungen (SA, CF) zeigt.
- **Compiler-Paket:** Der Datentyp ist dem Paket isobus::pgn zugeordnet, was auf seine Zugehörigkeit zur ISOBUS Parameter Group Number (PGN) Verarbeitung hindeutet.

Anwendungsszenarien

Dieser Datentyp wird primär in Szenarien verwendet, in denen eine Applikation über dynamische Änderungen im ISOBUS-Netzwerk informiert werden muss. Beispiele sind:

- Überwachung des Erscheinens oder Verschwindens von Steuergeräten (ECUs) im Netzwerk.
- Reaktion auf Adressänderungen von Netzwerkteilnehmern.
- Implementierung von benutzerdefinierter Logik beim Verbindungsaufbau oder -abbau zu einer bestimmten CF.
- Protokollierung von Netzwerkaktivitäten für Diagnosezwecke.

⚖️ Vergleich mit ähnlichen Bausteinen

Innerhalb der 4diac-Typbibliothek existieren verschiedene strukturierte Datentypen für Kommunikationsprotokolle. Im Gegensatz zu generischen Kommunikationsstrukturen (wie z.B. für TCP/IP) ist ISONETEVENT_T hochspezialisiert auf die Ereignismeldungen des ISOBUS-Protokollstapels, insbesondere den Teil der Netzwerkverwaltung (NM). Andere ISOBUS-bezogene Typen könnten sich eher auf den Austausch von Prozessdaten (PGN-Daten) konzentrieren.

Fazit

Der Datentyp ISONETEVENT_T ist ein essentielles Hilfsmittel für die Entwicklung von ISOBUS-fähigen Applikationen in 4diac, die auf Netzwerkereignisse reagieren müssen. Er bietet eine standardisierte, informationsreiche Schnittstelle, um Callback-basierte Ereignisbenachrichtigungen aus dem ISOBUS-Netzwerkstack zu empfangen und zu verarbeiten. Seine Verwendung fördert eine entkoppelte und ereignisgesteuerte Architektur in der Steuerungssoftware.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ISOUSERHOME_e

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ISOUSERHOME_e` ist eine **Konstantendefinition** (GlobalConstants) und kein ausführbarer Funktionsblock im klassischen Sinne. Er definiert eine Aufzählung (Enumeration) von ganzzahligen Konstanten, die zur Beschreibung der Netzwerkherkunft oder des Ziels eines Ereignisses oder einer Nachricht in einem ISOBUS-Kontext (ISO 11783) dienen. Diese Konstanten werden typischerweise in anderen Funktionsblöcken verwendet, um die Quelle oder den Empfänger von Daten innerhalb eines Controllernetzes zu kennzeichnen.

Schnittstellenstruktur

Da es sich um eine Konstantendefinition handelt, besitzt `ISOUSERHOME_e` keine ereignis- oder datenfluss-basierten Schnittstellen wie ein Standard-Funktionsblock. Stattdessen stellt es benannte Konstanten zur Verfügung.

Ereignis-Eingänge

Nicht vorhanden.

Ereignis-Ausgänge

Nicht vorhanden.

Daten-Eingänge

Nicht vorhanden.

Daten-Ausgänge

Nicht vorhanden.

Adapter

Nicht vorhanden.

Funktionsweise

Der Block definiert vier symbolische Konstanten vom Typ `SINT` (Short Integer):

- `intern` (Wert: 0): Kennzeichnet ein internes Control Function (CF) oder einen internen Benutzer innerhalb eines Geräts oder einer logischen Einheit.
- `network` (Wert: 1): Kennzeichnet ein externes CF oder einen externen Benutzer im Netzwerk.
- `thisMember` (Wert: 2): Spezifiziert, dass ein Ereignis oder eine Nachricht für dieses spezifische interne CF oder diesen Benutzer bestimmt ist.
- `notdef` (Wert: –1): Zeigt an, dass der Wert ungültig oder nicht definiert ist.

Diese Konstanten können in der gesamten Anwendung importiert und verwendet werden, um Code lesbarer und wartbarer zu machen, anstatt magische Zahlen (wie 0, 1, 2, –1) direkt zu verwenden.

Technische Besonderheiten

- **Typ:** `GlobalConstants` gemäß IEC 61499-1.
- **Datenbereich:** Die Konstanten sind vom Typ `SINT` (8-Bit vorzeichenbehaftete Ganzzahl).
- **Paket:** Der Block ist Teil des Pakets `isobus::pgn`, was auf seine Verwendung im Kontext von ISOBUS Parameter Group Numbers (PGNs) hindeutet.

Zustandsübersicht

Nicht anwendbar, da es sich um eine statische Konstantendefinition ohne Zustandsautomaten oder Verhalten handelt.

Anwendungsszenarien

Die Konstanten werden primär in ISOBUS-konformen Steuerungssystemen in der Landtechnik verwendet, beispielsweise in Traktoren und deren Anbaugeräten. Typische Anwendungen sind:

- **Nachrichtenfilterung:** Ein Empfängerblock kann anhand des `ISOUSERHOME_e`-Werts entscheiden, ob eine eingehende ISOBUS-Nachricht für das eigene Gerät (`thisMember`), ein anderes Gerät im Netzwerk (`network`) oder für eine interne Verarbeitung (`intern`) bestimmt ist.
- **Nachrichtenkonstruktion:** Ein Senderblock kann den Wert setzen, um den beabsichtigten Empfänger einer ausgesendeten Nachricht zu kennzeichnen.
- **Statusrückmeldung:** Ein Block kann den Status `notdef` zurückgeben, wenn die Herkunft oder das Ziel einer Nachricht nicht ermittelt werden kann.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`E_SR` oder `E_RESTART`:** Dies sind ausführbare Basis-Funktionsblöcke mit Ereignislogik. `ISOUSERHOME_e` ist hingegen eine reine Datendefinition ohne Logik.
- **Benutzerdefinierte Datentypen (DT):** Ähnlich wie ein strukturierter Datentyp (STRUCT) oder eine Aufzählung (ENUM) dient `ISOUSERHOME_e` dazu, semantische Bedeutung zu kodieren. Der Unterschied liegt in der IEC 61499-Spezifikation: `GlobalConstants` sind für unveränderliche Werte vorgesehen, die projektweit gültig sind.
- **Direkte Verwendung von Zahlenwerten:** Die Verwendung von `ISOUSERHOME_e.intern` ist im Code deutlich aussagekräftiger und weniger fehleranfällig als die direkte Verwendung der Zahl `0`.

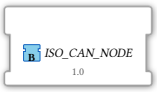
Fazit

`ISOUSERHOME_e` ist ein essentielles Hilfskonstrukt für die Entwicklung klarer und wartbarer ISOBUS-Anwendungen in 4diac. Durch die Bereitstellung aussagekräftiger Konstantennamen abstrahiert es von den zugrundeliegenden numerischen Werten und verbessert so die Lesbarkeit und Robustheit des Systementwurfs. Es ist ein Baustein auf Datenebene, der die Kommunikationslogik in höheren Funktionsblöcken unterstützt.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ISO_CAN_NODE



• – ◦ • – ◦ • – ◦ •

Einleitung

Die globale Konstantendatei `ISO_CAN_NODE` definiert eine Reihe von numerischen Konstanten zur Identifikation von CAN-Knoten (Controller Area Network) innerhalb eines 4diac-Projekts. Sie dient der Abstraktion und Vereinheitlichung von Knotenadressen, um die Lesbarkeit und Wartbarkeit von Applikationen zu erhöhen, die mit mehreren CAN-Bussen arbeiten.

Schnittstellenstruktur

Dieses Element ist eine Sammlung globaler Konstanten und besitzt keine ereignis- oder datenflussgesteuerten Schnittstellen wie ein Funktionsblock.

Daten-Konstanten

Die Datei definiert die folgenden Konstanten vom Typ `USINT` (Unsigned Short Integer):

Name	Typ	Initialwert	Kommentar

NODE1	USINT	0	Kennung für CAN 1
NODE2	USINT	1	Kennung für CAN 2
NODE3	USINT	2	Kennung für CAN 3
NODE4	USINT	3	Kennung für CAN 4
INVALID	USINT	USINT#16#FF	Kennung für einen ungültigen oder nicht konfigurierten CAN-Knoten (dezimal: 255)

Funktionsweise

Die Konstanten werden zur Kompilierzeit in den Code eingefügt. Anstatt direkte numerische Werte (wie `0` oder `1`) in Funktionsblöcken zu verwenden, die CAN-Kommunikation steuern, können Entwickler die sprechenden Namen (`NODE1`, `NODE2`, etc.) nutzen. Dies macht den Code intuitiver verständlich und erleichtert zukünftige Änderungen, da die tatsächlichen Adressen zentral an einem Ort verwaltet werden.

Der Wert `INVALID` (255) stellt einen reservierten Wert dar, der typischerweise verwendet wird, um einen nicht initialisierten, fehlerhaften oder nicht existenten Knoten zu kennzeichnen.

Technische Besonderheiten

- **Paketzuordnung:** Die Konstanten sind dem Paket `isobus::pgn` zugeordnet, was auf eine Verwendung im Kontext des ISO-Bus (ISOBUS, Landtechnik) und Parameter Group Numbers (PGN) hindeutet.
- **Datentyp:** Alle Konstanten sind vom Typ `USINT`, was einen Wertebereich von 0 bis 255 ermöglicht und für CAN-Knotenadressen üblich ist.
- **Hexadezimale Notation:** Der Wert für `INVALID` wird in hexadezimaler Schreibweise (`16#FF`) definiert, was der dezimalen Zahl 255 entspricht.

Zustandsübersicht

Da es sich um statische Konstanten handelt, existiert kein Zustandsautomat.

Anwendungsszenarien

- **Multi-CAN-Systeme:** In Steuerungen, die mit mehreren unabhängigen CAN-Bussen kommunizieren (z.B. ein Hauptbus und ein Antriebsbus), um die Busse eindeutig zu adressieren.
- **ISOBUS-Anwendungen:** Zur Konfiguration von virtuellen Terminals (VT), Task-Controllern (TC) oder anderen Geräten in einem landtechnischen ISOBUS-Netzwerk.
- **Fehlerbehandlung:** Die Konstante `INVALID` kann als Rückgabewert oder Initialwert in Funktionsblöcken dienen, um anzuzeigen, dass kein gültiger CAN-Knoten ausgewählt oder erkannt wurde.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Direkte Werte:** Der alternative Ansatz wäre die direkte Verwendung numerischer Literale (0, 1, 2, ...) im Code. Die Verwendung von `ISO_CAN_NODE` ist diesem Ansatz in puncto Codequalität und Wartbarkeit überlegen.
- **Enumerations (ENUM):** In einigen Programmiersprachen oder FB-Typen könnten ähnliche Konstrukte als Enumeration definiert werden. Globale Konstanten sind in IEC 61499 ein einfaches und weit verbreitetes Mittel, um dieses Muster umzusetzen.

Fazit

Die `ISO_CAN_NODE` Konstantendatei ist ein einfaches, aber wertvolles Hilfsmittel für die strukturierte Entwicklung von CAN-basierten Steuerungsapplikationen in 4diac. Sie fördert sauberen, selbsterklärenden Code und zentralisiert Konfigurationsdaten, was Fehler reduziert und Anpassungen erleichtert. Ihre Verwendung ist besonders in Projekten mit mehreren CAN-Knoten oder im ISOBUS-Umfeld empfehlenswert.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ISO_NETEVENT_e

• – ◦ • – ◦ • – ◦ •

Einleitung

Die globale Konstantendatei `ISO_NETEVENT_e` definiert eine Aufzählung (Enumeration) von Ereignistypen für das Netzwerkmanagement in einem ISOBUS-kompatiblen System. Sie stellt benannte, ganzzahlige Konstanten zur Verfügung, die verschiedene Zustandsänderungen und Ereignisse im Netzwerk repräsentieren, wie z.B. das Hinzufügen oder Entfernen von Teilnehmern oder Adresskonflikte. Die Verwendung dieser Konstanten erhöht die Lesbarkeit und Wartbarkeit des Quellcodes.

Schnittstellenstruktur

Bei `ISO_NETEVENT_e` handelt es sich um eine **Globale Konstantendatei** und nicht um einen Funktionsblock. Daher besitzt sie keine Ereignis- oder Datenschnittstellen im herkömmlichen Sinne. Stattdessen stellt sie eine Sammlung von Konstanten (`VAR_GLOBAL CONSTANT`) zur Verfügung, die systemweit genutzt werden können.

Daten-Eingänge

Nicht zutreffend. Diese Ressource definiert Konstanten, keine Eingänge.

Daten-Ausgänge

Nicht zutreffend. Diese Ressource definiert Konstanten, keine Ausgänge.

Ereignis-Eingänge

Nicht zutreffend.

Ereignis-Ausgänge

Nicht zutreffend.

Adapter

Nicht zutreffend.

Funktionsweise

Die Datei deklariert eine Reihe von Konstanten vom Typ `USINT` (Unsigned Short Integer, 8-Bit). Jeder Konstante ist ein eindeutiger numerischer Wert und ein sprechender Name zugeordnet. Diese Konstanten können in anderen Funktionsblöcken oder Programmen importiert und verwendet werden, um Netzwerkereignisse zu identifizieren oder Zustände abzufragen, anstatt magische Zahlen im Code zu verwenden.

Technische Besonderheiten

- **Typ:** Alle Konstanten sind vom Typ `USINT`.
- **Initialisierung:** Jede Konstante wird bei der Deklaration mit einem eindeutigen Wert initialisiert (0 bis 9).
- **Bedingte Kompilierung:** Die Konstanten `Isonet_HsiActive` und `Isonet_HsiInactive` sind als bedingte Konstanten gekennzeichnet (ursprünglich aus C-Code). Ihre Verwendung kann von der Definition eines Präprozessor-Symbols (`CCI_HSI`) abhängen.
- **Paketzuordnung:** Die Konstanten sind dem Paket `isobus::pgn` zugeordnet, was auf ihre Verwendung im Kontext der ISOBUS-Protokollbehandlung (Parameter Group Number) hindeutet.

Zustandsübersicht

Nicht zutreffend. Konstanten haben keinen Zustand.

Anwendungsszenarien

Diese Konstanten werden typischerweise in Funktionsblöcken verwendet, die:

1. Das ISOBUS-Netzwerkmanagement (Address Claiming, Network Management) implementieren.
2. Netzwerkereignisse überwachen und auf diese reagieren (z.B. bei Adresskonflikten).

3. Den Zustand anderer Netzwerkteilnehmer (Controller Applications, ECUs) verwalten.
4. Diagnose- oder Logging-Funktionen bereitstellen, die Netzwerkereignisse protokollieren.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu Funktionsblöcken (FB) oder Basis-Funktionsblöcken (BFB) enthält eine Globale Konstantendatei keine ausführbare Logik oder Algorithmen. Sie ist vergleichbar mit einer Header-Datei in C/C++ oder einem `ENUM` in höheren Programmiersprachen. Ihr Zweck ist ausschließlich die Definition von unveränderlichen Werten für die gesamte Anwendung.

Fazit

Die `ISO_NETEVENT_e` ist eine essentielle Ressource für die strukturierte Entwicklung von ISOBUS-Anwendungen in der 4diac IDE. Sie standardisiert die Kodierung von Netzwerkmanagement-Ereignissen, fördert die Code-Klarheit und reduziert Fehler, die durch die Verwendung undifferenzierter numerischer Werte entstehen können. Ihre Integration ist besonders für komplexe, netzwerkbasierte Steuerungssysteme im landwirtschaftlichen und mobilen Maschinenbereich von Vorteil.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

NAMEFIELD_T

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Datentyp `NAMEFIELD_T` repräsentiert eine strukturierte Datenkomponente zur Abbildung des ISO 11783 (ISOBUS) NAME-Feldes gemäß Teil 5 der Norm und den Konventionen der Implementierten Funktionsmodule (IFM). Dieser strukturierte Datentyp dient der eindeutigen Identifikation und Adressierung von Steuergeräten (ECUs) in einem ISOBUS-Netzwerk. Er fasst alle für die Geräteidentifikation erforderlichen Parameter in einer einzigen Datenstruktur zusammen.

Schnittstellenstruktur

`NAMEFIELD_T` ist ein strukturierter Datentyp (STRUCT), der aus zehn einzelnen Variablen besteht. Jede Variable entspricht einem definierten Teil des 64-Bit-NAME-Feldes im ISOBUS-Protokoll.

Daten-Eingänge

(Als strukturierter Datentyp besitzt `NAMEFIELD_T` keine Ereignisse, sondern ausschließlich Datenfelder. Diese werden hier als Bestandteile der Struktur aufgeführt.)

Name	Datentyp	Initialwert	Kommentar
`bSelfConf`	`BOOL`	`TRUE`	Gibt an, ob es sich um eine selbstkonfigurierbare Adress-ECU handelt.
`bIndGroup`	`BYTE`	`16#FF`	Weist den NAME einer Industriegruppe zu (Bereich 0..7).
`bDevClassInst`	`BYTE`	`16#FF`	Identifikator, wenn eine Geräteklasse mehrfach im Netzwerk vorkommt (0..15).
`bDevClass`	`BYTE`	`16#FF`	ISO-definierte Geräteklasse (0..127).
`bReserved`	`BOOL`	`TRUE`	Reserviertes Bit (nicht verwendet).
`bFunction`	`BYTE`	`16#FF`	ISO-definierte Funktion, entspricht der Source Address (SA) (0..255).
`bFunctionInst`	`BYTE`	`16#FF`	Untergruppe einer Funktion (0..31).
`bEcuInstance`	`BYTE`	`16#FF`	Gibt an, welches Steuergerät einer bestimmten Funktion zugeordnet ist (0..7).
`wManufCode`	`WORD`	`16#FFFF`	ISO-definierter Herstellercode (0..2047).

dwIdentNumb	DWORD	16#FFFFFFFF	Eindeutige Seriennummer; wird verwendet, wenn der NAME im selben Netzwerk nicht eindeutig ist (0..2.097.151).

Daten-Ausgänge

(Nicht zutreffend, da es sich um einen reinen Datentyp handelt.)

Adapter

(Nicht zutreffend, da es sich um einen reinen Datentyp handelt.)

Funktionsweise

Der Datentyp `NAMEFIELD_T` ist eine passive Datenstruktur. Seine Hauptaufgabe ist die strukturierte Speicherung und Übertragung der Parameter, die zusammen den 64-Bit-ISOBUS-NAME bilden. Jede Variable innerhalb der Struktur belegt einen spezifischen Bitbereich innerhalb dieses Gesamtfeldes. Die Initialwerte (`16#FF`, `16#FFFF`, `16#FFFFFFFF`) repräsentieren typischerweise einen undefinierten oder nicht zugewiesenen Zustand.

Technische Besonderheiten

- **Bitgenaue Abbildung:** Die Struktur bildet das binäre Layout des ISO-NAME-Feldes direkt ab, was eine effiziente Serialisierung und Deserialisierung für die Kommunikation über das CAN-Bus-Protokoll ermöglicht.
- **Initialwerte:** Die gewählten Initialwerte (maximale Werte für die jeweiligen Datentypen) signalisieren klar einen nicht initialisierten oder Platzhalter-Zustand.
- **ISOBUS-Konformität:** Die Benennung und Aufteilung der Felder folgt strikt der ISO 11783-5 Spezifikation und den erweiterten IFM-Konventionen.

Zustandsübersicht

Da es sich um einen Datentyp und nicht um einen Funktionsbaustein handelt, existiert keine Zustandsmaschine. Der „Zustand“ ist durch die konkreten Werte der zehn enthaltenen Variablen definiert.

Anwendungsszenarien

Der Datentyp `NAMEFIELD_T` wird primär in ISOBUS-konformen Steuerungsapplikationen verwendet, beispielsweise in der Land- und Forsttechnik oder im Baumaschinenbereich.

- **Geräteidentifikation:** Als Teil von Nachrichten (z.B. im Address Claim Prozess) zur Identifikation eines Teilnehmers im Netzwerk.
- **Datenhaltung:** Zur Speicherung der eigenen Geräteidentität innerhalb einer ECU-Software.
- **Filterung:** Zur Konfiguration von Nachrichtenfiltern basierend auf Geräteklasse, Funktion oder Hersteller.
- **Diagnose:** Zur eindeutigen Identifikation von Geräten in Diagnose- und Konfigurationstools.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu Funktionsbausteinen (FBs) wie `E_SR` oder `E_CYCLE` ist `NAMEFIELD_T` ein reiner **Datentyp** (Structured Data Type). Er enthält keine Algorithmen, Ereignisse oder einen ausgeführten Code.

- **Ähnlichkeit zu anderen STRUCTs:** Er ist vergleichbar mit anderen strukturierten Datentypen in 4diac (z.B. für komplexe Messwerte), jedoch spezialisiert auf das ISOBUS-Protokoll.
- **Unterschied zu Basis-Datentypen:** Während Basistypen wie `INT` oder `BOOL` einzelne Werte halten, fasst `NAMEFIELD_T` mehrere logisch zusammengehörige Werte zu einer Einheit zusammen.

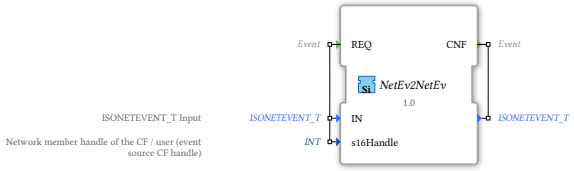
Fazit

Der Datentyp `NAMEFIELD_T` ist ein essentielles Hilfsmittel für die Entwicklung von ISOBUS-konformen Steuerungsapplikationen mit 4diac FORTE. Er abstrahiert die komplexe Bitstruktur des ISO-NAME-Feldes in eine leicht handhabbare, stark typisierte Datenstruktur. Dies fördert die Code-Wiederverwendbarkeit, verbessert die Lesbarkeit und reduziert Fehler bei der Handhabung von Geräteidentifikationsdaten. Seine Verwendung ist standardkonform und empfiehlt sich für alle Projekte, die ISOBUS-Kommunikation implementieren.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

NetEv2NetEv



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `NetEv2NetEv` dient der Manipulation von Netzwerkereignissen (NetEv) innerhalb eines ISOBUS-basierten Systems. Sein primärer Zweck ist es, einen eingehenden Netzwerkereignis-Datensatz zu übernehmen und einen spezifischen Wert – den Netzwerk-Member-Handle – zu überschreiben. Der Block fungiert somit als einfacher Konverter oder Adapter für ISONETEVENT_T-Datenstrukturen.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Dieses Ereignis löst die Verarbeitung des Funktionsblocks aus. Es ist mit den Daten-Eingängen `IN` und `s16Handle` verknüpft.

Ereignis-Ausgänge

- **CNF**: Dieses Ereignis signalisiert den Abschluss der Verarbeitung. Es wird nach der Bearbeitung des `REQ`-Ereignisses ausgelöst und ist mit dem (leeren) Daten-Ausgang verknüpft, der die manipulierte `ISONETEVENT_T`-Struktur ausgibt.

Daten-Eingänge

- **IN** (`isobus::pgn::ISONETEVENT_T`): Der Eingang für die ursprüngliche Netzwerkereignis-Datenstruktur, die manipuliert werden soll.
- **s16Handle** (`INT`): Der neue Netzwerk-Member-Handle, der in die übergebene `ISONETEVENT_T`-Struktur geschrieben werden soll. Dieser Wert identifiziert den Controller/Function (CF) oder Benutzer, der als Quelle des Ereignisses dient.

Daten-Ausgänge

- **<leer>** (`isobus::pgn::ISONETEVENT_T`): Der Ausgang liefert die manipulierte `ISONETEVENT_T`-Datenstruktur. Der Name des Ausgangs ist leer, der Datentyp ist identisch mit dem des Eingangs `IN`.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Die Funktionsweise des `NetEv2NetEv`-Blocks ist deterministisch und einfach:

1. Bei Eintreffen eines `REQ`-Ereignisses wird die Funktion ausgeführt.
2. Der gesamte Inhalt der Eingangsdatenstruktur `IN` wird in die Ausgangsdatenstruktur kopiert.
3. Anschließend wird das Feld `.s16Handle` innerhalb dieser kopierten Ausgangsstruktur mit dem Wert des Eingangs `s16Handle` überschrieben.
4. Das `CNF`-Ereignis wird ausgelöst, um die Bereitstellung des manipulierten Datensatzes am Ausgang zu signalisieren.

Der Block ändert nur das spezifizierte Handle-Feld. Alle anderen Felder der `ISONETEVENT_T`-Struktur bleiben unverändert.

Technische Besonderheiten

- **Datentyp**: Der Block arbeitet spezifisch mit dem Datentyp `isobus::pgn::ISONETEVENT_T`. Die genaue Struktur dieses Typs ist in der zugrundeliegenden `isobus::pgn`-Bibliothek definiert.
- **Direkte Zuweisung**: Die Implementierung erfolgt über eine direkte Zuweisung in Structured Text (ST). Es findet keine zusätzliche Logik, Prüfung oder Fehlerbehandlung statt.
- **Namenskonvention**: Der Daten-Ausgang besitzt keinen expliziten Namen, was in 4diac zulässig ist. Er wird über das `CNF`-Ereignis referenziert.

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand oder Speicher. Sein Verhalten ist rein kombinatorisch: Bei jedem `REQ`-Ereignis wird sofort die beschriebene Operation durchgeführt und ein `CNF`-Ereignis generiert.

Anwendungsszenarien

- **Handle-Überschreibung in Event-Streams**: In Netzwerken, wo Ereignisse zwischen verschiedenen Controllern (CFs) weitergeleitet werden, muss der Quell-Handle (`s16Handle`) eventuell angepasst werden, um den aktuellen Absender korrekt zu identifizieren.
- **Event-Proxy/Adapter**: Der Block kann als einfacher Proxy eingesetzt werden, um eingehende Netzwerkereignisse mit einer neuen Quellkennung zu versehen, bevor sie an nachgelagerte Verarbeitungslogik weitergegeben werden.
- **Test und Simulation**: Zum Simulieren von Netzwerkereignissen von verschiedenen virtuellen Quellen (CFs) innerhalb eines Test-Setups.

⚖️ Vergleich mit ähnlichen Bausteinen

- **E_SR (SR-Flip-Flop) oder E_D_FF (D-Flip-Flop)**: Diese Blöcke besitzen einen internen Zustand (Speicher). `NetEv2NetEv` ist zustandslos und manipuliert nur durchlaufende Daten.
- **E_SELECT / E_MUX**: Diese Blöchte wählen zwischen verschiedenen Datenquellen aus oder leiten sie durch. `NetEv2NetEv` verändert aktiv einen Teil der durchlaufenden Daten.
- **E_R_TRIG / E_F_TRIG**: Diese sind reine Ereignis-Trigger ohne Datenmanipulation. `NetEv2NetEv` ist ein datenverarbeitender Block.
- **Generische ASSIGN-Bausteine**: Ein einfacher `ASSIGN`-Block kopiert einen Eingangswert 1:1 auf den Ausgang. `NetEv2NetEv` ist ein spezialisierter `ASSIGN`-Block für den Typ `ISONETEVENT_T`, der eine gezielte Modifikation (Handle-Überschreibung) vornimmt.

🛠️ Zugehörige Übungen

- Uebung_128
- Uebung_128b
- Uebung_134

Fazit

Der `NetEv2NetEv`-Funktionsblock ist ein spezialisiertes, schlankes Werkzeug für ISOBUS-Anwendungen. Er erfüllt eine klare und notwendige Nischenaufgabe: das Überschreiben des Quell-Handles in Netzwerkereignis-Datensätzen. Seine einfache, zustandslose Implementierung macht ihn robust und vorhersehbar. Für Entwickler, die mit ISOBUS PGNs und Netzwerkereignissen arbeiten, stellt er ein nützliches Hilfselement dar, um Datenflüsse korrekt zu kennzeichnen und anzupassen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

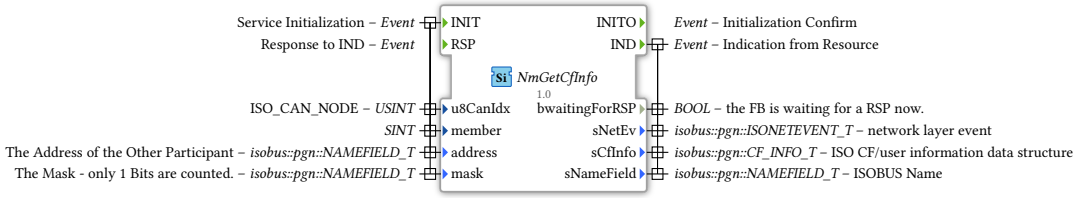
NmGetCfInfo

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `NmGetCfInfo` dient innerhalb eines ISOBUS-Netzwerks dazu, Informationen über verbundene Kommunikationspartner (Connected CFs) abzurufen. Er ermöglicht es, sowohl interne als auch externe Teilnehmer im Netzwerk zu identifizieren und deren Details zu erfassen. Der Baustein

ist Teil der ISOBUS-PGN-Kommunikation und wird typischerweise für Netzwerkmanagement-Aufgaben eingesetzt.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT (Typ: EInit)**: Initialisiert den Dienst. Bei Auslösung werden die konfigurierten Daten-Eingänge übernommen und der Initialisierungsprozess gestartet.
- **RSP (Typ: Event)**: Bestätigt den Empfang einer Indikation (`IND`). Dieses Ereignis signalisiert dem Baustein, dass eine zuvor gesendete Indikation verarbeitet wurde.

Ereignis-Ausgänge

- **INITO (Typ: EInit)**: Bestätigt den erfolgreichen Abschluss der Initialisierung.
- **IND (Typ: Event)**: Wird ausgelöst, um eine empfangene Netzwerkindikation (z.B. gefundene Kommunikationspartner) an die Anwendung weiterzugeben.

Daten-Eingänge

- **u8CanIdx (Typ: USINT, Initialwert: ISO_CAN_NODE::INVALID)**: Identifiziert den verwendeten CAN-Knoten (CAN-Controller-Index).
- **member (Typ: SINT, Initialwert: ISOUSERHOME_e::notdef)**: Definiert die Mitgliedschaft oder Rolle des anfragenden Teilnehmers im Netzwerk.
- **address (Typ: isobus::pgn::NAMEFIELD_T)**: Die Adresse des anderen Netzwerkteilnehmers, für den Informationen abgerufen werden sollen.
- **mask (Typ: isobus::pgn::NAMEFIELD_T)**: Eine Bitmaske, die angibt, welche Teile der Adresse bei der Suche berücksichtigt werden sollen. Nur gesetzte Bits (1) werden ausgewertet.

Daten-Ausgänge

- **bwaitingForRSP (Typ: BOOL)**: Zeigt an, ob der Funktionsblock aktuell auf eine Antwort (`RSP`) auf eine gesendete Indikation wartet.
- **sNetEv (Typ: isobus::pgn::ISONETEVENT_T)**: Enthält Details zum aufgetretenen Netzwerkereignis (z.B. Fehler oder Status).
- **sCfInfo (Typ: isobus::pgn::CF_INFO_T)**: Die abgerufenen Informationen über den gefundenen Kommunikationspartner (CF = Communication Function).
- **sNameField (Typ: isobus::pgn::NAMEFIELD_T)**: Der ISOBUS-Name des gefundenen Kommunikationspartners.

Adapter

Dieser Funktionsblock verwendet keine Adapter-Schnittstellen.

Funktionsweise

1. **Initialisierung**: Durch das Auslösen des `INIT`-Ereignisses wird der Baustein konfiguriert. Die übergebenen Parameter (`u8CanIdx`, `member`, `address`, `mask`) definieren den Kontext und das Ziel der Abfrage. Nach erfolgreicher Initialisierung wird `INITO` ausgegeben.
2. **Abfrage und Indikation**: Nach der Initialisierung beginnt der Baustein, basierend auf der `address` und `mask`, nach passenden Kommunikationspartnern im Netzwerk zu suchen. Wird ein Teilnehmer gefunden, löst der Baustein das `IND`-Ereignis aus und stellt die gefundenen Informationen in den Daten-Ausgängen `sCfInfo` und `sNameField` bereit. Gleichzeitig wird `bwaitingForRSP` auf `TRUE` gesetzt.
3. **Antwortverarbeitung**: Die übergeordnete Anwendung bestätigt den Empfang der Indikation durch das Senden des `RSP`-Ereignisses. Daraufhin setzt der Baustein `bwaitingForRSP` auf `FALSE` zurück. Dieser Handshake-Mechanismus stellt eine sichere Übertragung sicher.

Technische Besonderheiten

- Der Baustein implementiert ein Request-Response-Protokoll (`IND`/`RSP`) für zuverlässige Kommunikation.
- Die Verwendung einer Bit-`mask` ermöglicht flexible Suchanfragen, z.B. nach Teilnehmern einer bestimmten Geräteklasse.
- Die Datentypen (`ISONETEVENT_T`, `CF_INFO_T`, `NAMEFIELD_T`) sind spezifisch für die ISOBUS-PGN-Kommunikation und enthalten strukturierte Informationen gemäß dem ISO 11783-Standard.

Zustandsübersicht

1. **Inaktiv**: Vor der Initialisierung.
2. **Initialisiert**: Nach `INITO`. Der Baustein ist betriebsbereit und kann Anfragen bearbeiten.
3. **Wartend auf RSP**: Nach dem Auslösen von `IND`. Der Baustein wartet auf die Bestätigung (`RSP`) der Anwendung, bevor er weitere Aktionen durchführt.

Anwendungsszenarien

- **Netzwerk-Discovery**: Beim Start eines Geräts, um alle aktiven Teilnehmer im ISOBUS-Netzwerk zu identifizieren.
- **Diagnose und Monitoring**: Zur Überwachung, ob bestimmte erwartete Steuergeräte (z.B. eines Anbaugeräts) verbunden und erreichbar sind.
- **Dynamische Konfiguration**: Um Informationen über neu hinzugekommene Geräte zu erhalten und die Applikation entsprechend anzupassen.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu einfachen Lese- oder Abfrage-Bausteinen ist `NmGetCfInfo` speziell auf die Abfrage von ISOBUS-spezifischen Netzwerkinformationen ausgelegt. Er bietet mehr Kontext (durch `member`, `address`, `mask`) und strukturierte Ergebnisdaten (`CF_INFO_T`) als ein generischer Kommunikationsbaustein. Bausteine wie `E_SWITCH` oder `E_DEMUX` leiten nur Ereignisse weiter, während `NmGetCfInfo` aktive Netzwerkkommunikation und Protokollhandlung durchführt.

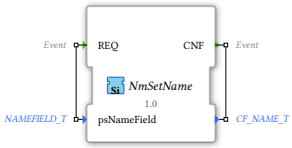
🛠️ Zugehörige Übungen

- Uebung_120
- Uebung_121
- Uebung_122
- Uebung_122b
- Uebung_123
- Uebung_124
- Uebung_125
- Uebung_126
- Uebung_126b
- Uebung_126b2
- Uebung_127
- Uebung_128
- Uebung_128b
- Uebung_130
- Uebung_131
- Uebung_132
- Uebung_133
- Uebung_134

Fazit

Der `NmGetCfInfo`-Funktionsblock ist ein essenzielles Werkzeug für das Management von ISOBUS-Netzwerken. Er abstrahiert die komplexe Protokollkommunikation zur Erkennung von Netzwerkteilnehmern in eine einfach zu verwendende, ereignisgesteuerte Schnittstelle. Seine Fähigkeit, gezielt nach Teilnehmern zu suchen und detaillierte Informationen zurückzuliefern, macht ihn wertvoll für robuste und intelligente landwirtschaftliche Steuerungssysteme.

NmSetName



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `NmSetName` dient der Konvertierung einer strukturierten Namensfeld-Definition in ein kompaktes, 8-Byte langes ISO-NAME gemäß der Spezifikation ISO 11783-5 (Abschnitt 5.1.2). Dieser Baustein ist ein wesentlicher Bestandteil für die Geräteidentifikation und -adressierung in ISOBUS-Netzwerken (Landwirtschaftliche und Forstmaschinen). Er übernimmt die Bit-genaue Zuordnung der einzelnen Felder einer `NAMEFIELD_T`-Struktur in das entsprechende Byte-Array eines `CF_NAME_T`.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Dieses Ereignis löst die Konvertierung aus. Bei seinem Eintreffen werden die anliegenden Daten am Eingang `psNameField` verarbeitet.

Ereignis-Ausgänge

- **CNF**: Dieses Ereignis signalisiert den Abschluss der Verarbeitung. Es wird nach der erfolgreichen Umwandlung der Eingangsdaten generiert.

Daten-Eingänge

- **psNameField** (`isobus::pgn::NAMEFIELD_T`): Die Eingangsstruktur, die alle Einzelkomponenten eines ISOBUS-Namens (wie Herstellercode, Geräteklasse, Funktionsinstanz, etc.) in einem für den Programmierer gut handhabbaren Format enthält.

Daten-Ausgänge

- **(Unbenannt)** (`isobus::pgn::CF_NAME_T`): Der Ausgang liefert das resultierende 8-Byte große ISO-NAME-Array (`data[0]` bis `data[7]`), das direkt in ISOBUS-Nachrichten verwendet werden kann.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Baustein arbeitet als reiner Daten-Transformer. Bei Auslösung durch das `REQ`-Ereignis liest er die Werte aus der Eingangsstruktur `psNameField` und verteilt sie gemäß der ISO-Norm bitweise auf die acht Bytes des Ausgangs-Arrays. Die Logik besteht aus einer sequentiellen Abfolge von bitweisen Operationen (AND, OR, SHL) und Typkonvertierungen. Jede Zeile im ST-Code setzt bestimmte Bits in einem spezifischen Byte des `CF_NAME_T`-Ausgangs, während die restlichen Bits unverändert bleiben (durch die `AND`-Operationen maskiert). Nachdem alle Bytes gesetzt sind, wird das `CNF`-Ereignis ausgegeben.

Technische Besonderheiten

- **Bitweise Verarbeitung:** Die Kernfunktionalität basiert auf präzisen bitweisen Operationen, um die komplexe Struktur des ISO-NAME in ein kompaktes Byte-Array zu packen.
- **Typensicherheit:** Es werden explizite Typkonvertierungen (z.B. `DWORD_TO_BYTE`, `WORD_TO_BYTE`) verwendet, um die korrekte Datenhandhabung sicherzustellen.
- **Normkonformität:** Die Implementierung folgt strikt der Bit-Zuordnung, wie sie in ISO 11783-5 definiert ist.
- **Zustandslos:** Der Block besitzt keinen internen Zustand zwischen zwei Aufrufen. Die Ausgabe hängt ausschließlich von den aktuellen Eingabedaten ab.

Zustandsübersicht

Der Funktionsblock ist zustandslos. Sein Verhalten lässt sich als einfache, deterministische Funktion beschreiben:

1. **Warten:** Block wartet auf das `REQ`-Ereignis.
2. **Verarbeiten:** Bei `REQ` werden die Eingangsdaten verarbeitet und das Ausgangs-Array berechnet.
3. **Bestätigen:** Unmittelbar nach der Berechnung wird das `CNF`-Ereignis ausgegeben. Der Block kehrt danach direkt wieder in den Wartezustand (1) zurück.

Anwendungsszenarien

- **Initialisierung eines ISOBUS-Knotens:** Beim Start eines Steuergeräts (ECU) in einem ISOBUS-Netzwerk muss dessen eindeutiger NAME aus konfigurierbaren Parametern (z.B. aus einem NVRAM) zusammengesetzt und in die Kommunikationsstack eingetragen werden.
- **Dynamische Gerätekonfiguration:** In Systemen, wo sich die Geräteeigenschaften (z.B. Funktionsinstanz) zur Laufzeit ändern können, wird dieser Block verwendet, um den neuen, gültigen NAME zu generieren.
- **Test- und Simulationstools:** Zum Erzeugen von korrekten ISO-NAMEs für die Simulation verschiedener virtueller Geräte in einem ISOBUS-Netzwerk.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Gegenstück `NmGetName`:** Während `NmSetName` eine strukturierte Beschreibung in ein Byte-Array packt, würde ein hypothetischer `NmGetName`-Block die umgekehrte Operation durchführen: Er extrahiert aus einem gegebenen ISO-NAME-Byte-Array die einzelnen Felder und stellt sie in einer `NAMEFIELD_T`-Struktur bereit. `NmSetName` ist somit der „Encoder“, sein Pendant wäre der „Decoder“.
- **Generische Byte-Packer:** Im Gegensatz zu generischen Bausteinen, die beliebige Datenstrukturen serialisieren, ist `NmSetName` spezifisch auf die ISO 11783-NAME-Struktur optimiert und garantiert so Normkonformität ohne zusätzliche Konfiguration.

🛠️ Zugehörige Übungen

- Uebung_121

Fazit

Der `NmSetName`-Funktionsblock ist ein spezialisierter, unverzichtbarer Baustein für die ISOBUS-Kommunikation. Er abstrahiert die komplexe, bitgenaue Zusammensetzung des ISO-NAME gemäß der Norm und bietet dem Anwender eine einfache, strukturierte Schnittstelle. Seine deterministische und zustandslose Funktionsweise macht ihn robust und einfach in Steuerungsabläufe zur Geräteinitialisierung und -konfiguration integrierbar.

NmSetNameField

• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock `NmSetNameField` dient der Konvertierung eines ISO-NAME gemäß der Spezifikation ISO 11783-5 (Abschnitt 5.1.2) in eine strukturierte Darstellung. Ein ISO-NAME ist ein 8-Byte langer Bezeichner, der in landwirtschaftlichen und forstwirtschaftlichen Maschinen (ISOBUS) zur eindeutigen Identifikation von Steuergeräten (ECUs) verwendet wird. Dieser Baustein extrahiert die einzelnen Felder aus dem Byte-Array und füllt eine entsprechende Datenstruktur (`NAMEFIELD_T`). Dies erleichtert die weitere Verarbeitung und Auswertung der Geräteinformationen in einer IEC 61499-Applikation.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Verarbeitung. Bei Empfang dieses Ereignisses wird der an `au8IsoName` anliegende Wert verarbeitet.

Ereignis-Ausgänge

- **CNF**: Wird ausgelöst, sobald die Konvertierung abgeschlossen ist und die strukturierten Daten am Ausgang bereitstehen.

Daten-Eingänge

- **au8IsoName** (`isobus::pgn::CF_NAME_T`): Ein 8-Byte langes Array, das den rohen ISO-NAME gemäß ISO 11783-5 enthält.

Daten-Ausgänge

- **(kein expliziter Name)** (`isobus::pgn::NAMEFIELD_T`): Die Ausgabevariable enthält die strukturierte Aufschlüsselung des ISO-NAME. Der genaue Aufbau der `NAMEFIELD_T`-Struktur wird durch die verwendete `isobus::pgn`-Bibliothek definiert.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Auslösung des `REQ`-Ereignisses verarbeitet der Baustein das anliegende Byte-Array `au8IsoName`. Die Implementierung in Structured Text (ST) extrahiert bit- und byteweise die einzelnen Komponenten des ISO-NAME und weist sie den entsprechenden Feldern der `NAMEFIELD_T`-Ausgabestruktur zu. Die Logik folgt exakt der Bit-Zuordnung, wie sie in ISO 11783-5, Abschnitt 5.1.2, definiert ist. Nach Abschluss der Berechnungen wird das `CNF`-Ereignis gesetzt, um den erfolgreichen Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- **Bitmanipulation:** Der Block verwendet logische UND-Operationen (`AND`), Schiebeoperationen (`SHL`, `SHR`) und Typkonvertierungen (`BYTE_TO_WORD`, `BYTE_TO_DWORD`), um die relevanten Bits aus dem Eingabe-Array zu isolieren.
- **Bibliotheksabhängigkeit:** Der Block ist von den Datentypen `CF_NAME_T` und `NAMEFIELD_T` der `isobus::pgn`-Bibliothek abhängig. Die genaue Definition dieser Typen (z.B. die Feldnamen in `NAMEFIELD_T`) liegt außerhalb dieses Blocks.
- **Deterministisch:** Die Operation ist rein datengetrieben und hat keinen internen Zustand. Bei gleicher Eingabe erzeugt sie stets die gleiche Ausgabe.

Zustandsübersicht

Der Funktionsblock besitzt keinen persistenten internen Zustand. Er verhält sich wie eine reine Funktion:

1. **Warte- bzw. Idle-Zustand:** Block wartet auf das `REQ`-Ereignis.
2. **Verarbeitungszustand:** Bei `REQ` werden die Eingangsdaten verarbeitet und die Ausgangsstruktur berechnet. Dieser Zustand ist instantan (keine Verzögerung).
3. **Ausgabe:** Unmittelbar nach der Berechnung wird das `CNF`-Ereignis ausgelöst und die berechnete `NAMEFIELD_T`-Struktur ausgegeben. Der Block kehrt danach in den Wartezustand zurück.

Anwendungsszenarien

- **ISOBUS-Geräteerkennung:** In einem ISOBUS-Netzwerkmanager, der eingehende NAME-Daten von angeschlossenen ECUs decodieren muss, um Gerätetyp, Hersteller und Instanz zu identifizieren.
- **Diagnose und Protokollierung:** Zur übersichtlichen Darstellung und Protokollierung von Gerätenamen in menschenlesbarer, strukturierter Form anstelle des rohen 8-Byte-Wertes.
- **Filterung und Adressierung:** Als Vorverarbeitungsschritt, um basierend auf spezifischen Feldern (wie Geräteklasse oder Herstellercode) Entscheidungen über die weitere Kommunikation zu treffen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Gegenstück `NmGetIsoName`:** Ein hypothetischer komplementärer Baustein würde die umgekehrte Operation durchführen: Er nähme eine `NAMEFIELD_T`-Struktur als Eingabe und erzeugte daraus das entsprechende 8-Byte-`CF_NAME_T`-Array. `NmSetNameField` ist der Dekodierer, sein Gegenstück wäre der Kodierer.
- **Generische Byte-Parser:** Im Gegensatz zu generischen Byte-zu-Struktur-Konvertern ist dieser Block spezifisch auf das ISO-NAME-Format zugeschnitten und enthält die exakte Bitlogik der Norm. Die Verwendung dieses spezialisierten Blocks reduziert Fehler und erhöht die Wartbarkeit im ISOBUS-Kontext.

🛠️ Zugehörige Übungen

- Uebung_120
- Uebung_121
- Uebung_122
- Uebung_122b

Fazit

Der `NmSetNameField`-Funktionsblock ist ein essenzielles Hilfsmittel für die Arbeit mit ISOBUS-Netzwerken in 4diac. Er abstrahiert die komplexe Bit-Manipulation, die zur Interpretation des ISO-NAME erforderlich ist, und stellt die Informationen in einer leicht verarbeitbaren strukturierten Form bereit. Seine Verwendung fördert korrekte, normkonforme Applikationen und verbessert die Lesbarkeit des Quellcodes erheblich.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

PgnDef

• – ○ • – ○ • – ○ •

Einleitung

`PgnDef` ist eine Sammlung globaler Konstanten für die 4diac-IDE, die Parameter Group Numbers (PGNs) für ISOBUS- und NMEA-2000-Kommunikationsprotokolle definiert. Diese Konstanten repräsentieren numerische Identifikatoren für spezifische Nachrichtentypen, die in der landwirtschaftlichen und maritimen Fahrzeugvernetzung verwendet werden. Die Sammlung dient als zentrale Referenzbibliothek, um in IEC 61499-Applikationen auf standardisierte PGNs zugreifen zu können, was die Lesbarkeit und Wartbarkeit des Codes verbessert.

Schnittstellenstruktur

`PgnDef` ist ein `GlobalConstants`-Element und besitzt daher keine ereignis- oder datenflussbasierten Schnittstellen im herkömmlichen Sinne eines Funktionsblocks. Stattdessen stellt es eine Sammlung von Konstanten zur Verfügung, die in anderen Funktionsblöcken oder Programmen deklariert und verwendet werden können.

Ereignis-Eingänge

Nicht vorhanden. Dies ist eine Konstantendefinition.

Ereignis-Ausgänge

Nicht vorhanden. Dies ist eine Konstantendefinition.

Daten-Eingänge

Nicht vorhanden. Dies ist eine Konstantendefinition.

Daten-Ausgänge

Nicht vorhanden. Dies ist eine Konstantendefinition.

Adapter

Nicht vorhanden. Dies ist eine Konstantendefinition.

Funktionsweise

`PgnDef` selbst hat keine ausführbare Logik. Es definiert lediglich eine umfangreiche Liste von Konstanten vom Datentyp `UDINT` (Unsigned Double Integer). Jede Konstante ist ein PGN-Wert, der in hexadezimaler Notation (`UDINT#16#XXXX`) initialisiert wird und einen menschenlesbaren Kommentar mit dem PGN-Namen und oft einer Referenz zum entsprechenden Standard (z.B. ISO 11783-7) enthält. Andere Funktionsblöcke können diese Konstanten über ihren vollqualifizierten Namen (z.B. `isobus::pgn::PGN_HEARTBEAT`) importieren und in ihrer Logik verwenden, um Nachrichten zu identifizieren oder zu erzeugen.

Technische Besonderheiten

- **Umfangreiche Abdeckung:** Die Sammlung enthält PGNs aus einer Vielzahl von Standards, einschließlich ISO 11783 (ISOBUS, Teile 3-7, 12, 13), SAE J1939 (71, 73, 75) und NMEA 2000.
- **Organisierte Struktur:** Die Konstanten sind in logischen Gruppen kommentiert (PDU1/Group Functions, PDU2, J1939-71, ISO 11783-7, Diagnose (J1939-73), Generator Sets (J1939-75), NMEA).
- **Datentyp:** Alle PGNs sind als `UDINT` (32-Bit, vorzeichenlos) definiert, was dem adressierbaren Bereich der Protokolle entspricht.
- **Paketzuordnung:** Die Konstanten sind dem Paket `isobus::pgn` zugeordnet, was eine klare Namensraumtrennung ermöglicht.

Zustandsübersicht

Nicht zutreffend, da es sich um eine statische Konstantendeklaration handelt.

Anwendungsszenarien

- **ISOBUS-ECU-Entwicklung:** Bei der Entwicklung von Steuergeräten (ECUs) für Traktoren oder Anbaugeräte, die über ein ISOBUS-Netzwerk kommunizieren.
- **NMEA-2000-Integration:** In maritimen Anwendungen zur Integration von Schiffselektronik wie GPS-Empfängern, Tiefenmessern oder Motorensteuerungen.
- **Protokoll-Implementierung:** Als Grundlage für die Implementierung von Protokoll-Stacks (z.B. für Transportprotokolle wie TP/CM oder ETP/CM).
- **Nachrichtenfilterung:** In Gateway- oder Bridge-Funktionsblöcken, um eingehende Nachrichten anhand ihrer PGN zu filtern und weiterzuleiten.
- **Diagnose-Tools:** In Applikationen zur Diagnose und Überwachung von Fahrzeugnetzwerken, um empfangene Nachrichten korrekt zu benennen und anzuzeigen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Einfache `VAR_GLOBAL CONSTANT`-Deklarationen:** `PgnDef` bietet gegenüber manuell im Funktionsblock deklarierten Konstanten den Vorteil der Zentralisierung, Wiederverwendbarkeit und Vermeidung von Redundanz und Tippfehlern.
- **Enumerations (ENUM):** Während ENUMs in IEC 61499 für geordnete Listen verwendet werden können, ist `PgnDef` als Sammlung diskreter, nicht notwendigerweise aufeinanderfolgender Werte besser geeignet. Konstanten bieten zudem mehr Flexibilität bei der Wertzuweisung (hexadezimal).
- **Spezifische Protokoll-FBs:** Im Gegensatz zu Funktionsblöcken, die eine bestimmte Protokollfunktion implementieren (z.B. ein `ADDRESS_CLAIM`-FB), ist `PgnDef` ein reines Hilfsmittel zur Referenzierung und enthält keine aktive Logik.

Fazit

Die `PgnDef`-Konstantensammlung ist ein wesentliches Hilfsmittel für die Entwicklung von IEC 61499-basierten Applikationen in den Domänen der mobilen Arbeitsmaschinen (ISOBUS) und der maritimen Elektronik (NMEA 2000). Sie fördert die Standardisierung, verbessert die Codequalität durch verbesserte Lesbarkeit und reduziert Fehler durch die Verwendung zentral definierter und getesteter Werte. Für jeden Entwickler, der mit diesen Protokollen in 4diac arbeitet, stellt `PgnDef` eine wertvolle und wahrscheinlich unverzichtbare Ressource dar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

USERFUNC_e

• – ◦ • – ◦ • – ◦ •

Einleitung

Die globale Konstante `USERFUNC_e` definiert eine Aufzählung (Enumeration) von numerischen Werten, die spezifische Gerätefunktionen oder Rollen innerhalb eines ISOBUS-Netzwerks repräsentieren. Sie dient der standardisierten und lesbaren Identifikation verschiedener ISOBUS-Gerätetypen und -Funktionen in Steuerungsanwendungen.

Schnittstellenstruktur

Bei `USERFUNC_e` handelt es sich um eine Sammlung globaler Konstanten und nicht um einen Funktionsblock. Daher existieren keine Ereignis- oder Datenschnittstellen im herkömmlichen Sinne. Die Struktur besteht ausschließlich aus benannten Konstanten vom Typ `SINT` (Short Integer).

Daten-Eingänge

Nicht zutreffend. Diese Ressource stellt Konstanten zur Verfügung und besitzt keine Eingänge.

Daten-Ausgänge

Nicht zutreffend. Die Konstanten werden direkt im Anwendungscode referenziert.

Adapter

Nicht zutreffend.

Funktionsweise

`USERFUNC_e` ist eine Sammlung vordefinierter, konstanter Werte. Jeder Name (z.B. `task_controller`) ist mit einem festen numerischen Wert (z.B. `1`) verknüpft. Diese Konstanten können in der gesamten Anwendung verwendet werden, um Geräterollen zu identifizieren, Vergleiche durchzuführen oder Parameter zu setzen, was die Code-Wartbarkeit und Lesbarkeit im Vergleich zur direkten Verwendung von „Magic Numbers“ erheblich verbessert.

Technische Besonderheiten

- **Datentyp:** Alle Konstanten sind vom Typ `SINT` (8-Bit vorzeichenbehaftete Ganzzahl).
- **Bereich:** Die definierten Werte liegen im Bereich von `-1` (`undefined`) bis `21`.
- **ISOBUS-Konformität:** Die Bezeichnungen und Werte orientieren sich an den im ISOBUS-Standard (ISO 11783) definierten Gerätefunktionen.
- **Spezielle Werte:**
  - `undefined` (-1): Kennzeichnet eine undefinierte oder unbekannte Funktion.
  - `all_clients` (20) & `all_server` (21): Werte für Broadcast- oder Gruppenadressierung.

Zustandsübersicht

Nicht zutreffend. Konstanten besitzen keinen Zustand.

Anwendungsszenarien

- **Geräteidentifikation:** In einem Task-Controller kann mit `IF device_function = USERFUNC_e.virtual_terminal THEN ...` geprüft werden, ob eine Verbindung zu einem Terminal besteht.
- **Nachrichtenfilterung:** Filterung von ISOBUS-Nachrichten (PGNs) basierend auf der Quell- oder Zielgerätefunktion.
- **Konfiguration:** Parametrierung von Netzwerkdiensten, die nur für bestimmte Geräterollen (z.B. `file_server`) verfügbar sein sollen.

- **Diagnose:** Lesbare Ausgabe von Geräterollen in Diagnoseprotokollen oder auf Benutzeroberflächen.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu Funktionsblöcken, die eine aktive Logik enthalten, ist `USERFUNC_e` eine passive Datentyp-/Konstantendefinition. Sie ist vergleichbar mit einem `ENUM`-Datentyp in höheren Programmiersprachen oder einer `CONSTANT`-Sektion in strukturiertem Text (ST). Ihr primärer Zweck ist die Bereitstellung semantisch bedeutsamer Namen für numerische Werte, die im ISOBUS-Kontext verwendet werden.

Fazit

Die globale Konstante `USERFUNC_e` ist ein essentielles Hilfsmittel für die Entwicklung lesbarer und wartbarer ISOBUS-Anwendungen in 4diac. Sie abstrahiert die rohen, im Standard definierten numerischen Werte für Gerätefunktionen durch aussagekräftige Bezeichner und trägt so maßgeblich zur Vermeidung von Fehlern und zur Verbesserung der Codequalität bei. Ihre Verwendung wird für alle Projekte empfohlen, die mit ISOBUS-Gerätekommunikation arbeiten.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Const

Global_NAME_Functions

• – ◦ • – ◦ • – ◦ •

Einleitung

Dieser Baustein (GlobalConstants) definiert eine umfassende Sammlung globaler Konstanten für die Funktionseinstufung von Steuergeräten (ECUs) im ISOBUS-Protokoll (ISO 11783). Die Konstanten repräsentieren die **NAME Function**-Werte, die in der ISOBUS-Nomenklatur zur Identifizierung der primären Funktion eines Geräts im Fahrzeug- oder Maschinennetzwerk verwendet werden. Sie sind als numerische BYTE-Werte definiert und decken alle standardisierten Funktionen von 0 bis 94 sowie den Platzhalter 255 für „nicht verfügbar“ ab.

Schnittstellenstruktur

Der Baustein stellt keine Ereignis- oder Dateneingänge bereit. Er definiert ausschließlich globale Konstanten, die als **Daten-Ausgänge** interpretiert werden können, da sie von anderen Bausteinen referenziert und abgefragt werden können.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Die folgenden globalen Konstanten (Typ `BYTE`) werden bereitgestellt:

Konstante	Wert	Beschreibung
F_ENGINE	0	Motor – typischerweise die mechanische Antriebsquelle der Maschine
F_AUX_POWER_UNIT	1	Hilfsaggregat (APU) – Energiequelle für Betriebssysteme ohne den Hauptantrieb
F_ELEC_PROPULSION_CONTROL	2	Elektrische Antriebssteuerung – Steuerung des elektrisch angetriebenen Antriebsmechanismus

F_TRANSMISSION	3	Getriebe – Mechanisches System zur Änderung von Drehzahl/Drehmoment des Motors
F_BATTERY_PACK_MONITOR	4	Batteriepack-Überwachung – überwacht Ladezustand, Temperatur, Restleistung usw.
F_SHIFT_CONTROL_CONSOLE	5	Schaltsteuerung/Konsole – bestimmt und sendet Gang, Bereich und Betriebsart auf das Netzwerk
F_POWER_TAKEOFF_MAIN_OR_REAR	6	Zapfwelle (Haupt oder Heck) – steuert die vom Motor abgeleitete mechanische Leistung für Zusatzgeräte
F_AXLE_STEERING	7	Achse – Lenkung – passt den Angriffswinkel in Abhängigkeit von der Lenkung an
F_AXLE_DRIVE	8	Achse – Antrieb
F_BRAKES_SYSTEMCONTROL	9	Bremssystem-Controller – elektronische Steuerung des Betriebsbremssystems
F_BRAKES_STEER_AXLE	10	Bremsen – Gelenkte Achse – Steuerung der Betriebsbremse an einer gelenkten Achse
F_BRAKES_DRIVE_AXLE	11	Bremsen – Antriebsachse – Steuerung der Betriebsbremse an einer Antriebsachse
F_RETARDER_ENGINE	12	Retarder – Motor – Steuerung der Motorbremse
F_RETARDER_DRIVELINE	13	Retarder – Antriebsstrang – Steuerung der Antriebsstrangbremse
F_CRUISE_CONTROL	14	Tempomat – Steuerungssystem zur Konstantgeschwindigkeitsregelung
F_FUEL_SYSTEM	15	Kraftstoffsystem – steuert den Kraftstofffluss vom Tank zum Filter und Motor
F_STEERING_CONTROL	16	Lenkungscontroller (Steer-by-Wire)
F_SUSPENSION_STEER_AXLE	17	Federung – Gelenkte Achse – Steuerung der Federung an einer gelenkten Achse
F_SUSPENSION_DRIVE_AXLE	18	Federung – Antriebsachse – Steuerung der Federung an einer angetriebenen Achse
F_INSTRUMENT_CLUSTER	19	Instrumententafel – Anzeige für Fahrzeugdaten, typischerweise im Armaturenbrett
F_TRIP_RECORDER	20	Fahrtenschreiber – erfasst Fahrzeugdaten (Distanz, Zeit)
F_CAB_CLIMATE_CONTROL	21	Kabinenklimasteuerung – steuert das Klima in der Kabine
F_AERODYNAMIC_CONTROL	22	Aerodynamiksteuerung – verändert z.B. Spoiler oder Luftleiteinrichtungen
F_VEHICLE_NAVIGATION	23	Fahrzeugnavigation – Position und Routenführung
F_VEHICLE_SECURITY	24	Fahrzeugsicherheit – Zugriffskontrolle und Diebstahlschutz

F_NETWORK_INTERCONNECT_UNIT	25	Netzwerk-Verbindungs-ECU – verbindet verschiedene Netzwerksegmente
F_BODY_CONTROLLER	26	Karosseriecontroller – steuert Karosseriekomponenten (nicht Chassis oder Kabine)
F_POWER_TAKEOFF_SECONDARY_OR_FRONT	27	Zapfwelle (Sekundär oder Front) – mechanische Leistungsabgabe für Zusatzgeräte
F_OFF_VEHICLE_GATEWAY	28	Off-Vehicle-Gateway – verbindet Fahrzeugnetzwerke mit externen Systemen
F_VTERMINAL	29	Virtuelles Terminal (im Fahrzeug) – intelligentes Display für Bedienung und Anzeige
F_MANAGEMENT_COMPUTER	30	Management-Computer – steuert Fahrzeugsysteme, z.B. Antriebsstrang
F_PROPULSION_BATTERY_CHARGER	31	Ladegerät für Antriebsbatterie – lädt Batterien aus externer Quelle
F_HEADWAY_CONTROLLER	32	Abstandsregelung – Kollisionsvermeidung, Warnung, Geschwindigkeitsanpassung
F_SYSTEM_MONITOR	33	Systemmonitor
F_HYDRAULIC_PUMP_CONTROLLER	34	Hydraulikpumpensteuerung – versorgt angebaute Geräte mit Hydraulikleistung
F_SUSPENSION_SYSTEM_CONTROLLER	35	Federungs-Systemcontroller – koordiniert die gesamte Federung des Fahrzeugs
F_PNEUMATIC_SYSTEM_CONTROLLER	36	Pneumatik-Systemcontroller
F_CAB_CONTROLLER	37	Kabinencontroller – verwaltet Funktionen in der Kabine
F_TIRE_PRESSURE_CONTROL	38	Reifendruckregelung – zentrale Reifenfüllung
F_IGNITION_CONTROL_MODULE	39	Zündsteuermodul – steuert die Zündung eines Motors
F_SEAT_CONTROL	40	Sitzsteuerung – bedient Sitz und Federung, Positionierung
F_LIGHTING_OPERATOR_CONTROLS	41	Beleuchtungssteuerung – Senden von Beleuchtungsbefehlen durch Bedienelemente
F_WATER_PUMP_CONTROL	42	Wasserpumpensteuerung – z.B. für Feuerwehrfahrzeuge oder Nutzfahrzeuge
F_TRANSMISSION_DISPLAY	43	Getriebe-Display – zeigt Getriebeinformationen
F_EXHAUST_EMISSION_CONTROL	44	Abgasemissionssteuerung
F_VEHICLE_DYNAMIC_STABILITY_CONTROL	45	Fahrzeugdynamik- und Stabilitätskontrolle
F_OIL_SENSOR_UNIT	46	Ölsensoreinheit
F_INFORMATION_SYSTEM_CONTROLLER	47	Informationssystem-Controller – z.B. Transit-, Fracht- oder Flottenmanagement
F_RAMP_CONTROL	48	Rampensteuerung – Ladung, Entladung, Hublifte, Heckklappen

F_CLUTCH_CONVERTER_CONTROL	49	Kupplungs-/Wandlersteuerung – steuert den Drehmomentwandler oder die Motor-Getriebe-Verbindung
F_AUXILIARY_HEATER	50	Zusatzheizung – Heizung ohne laufenden Hauptmotor
F_FORWARD_LOOKING_COLLISION_WARNING_SYSTEM	51	Vorausschauendes Kollisionswarnsystem – erkennt und warnt vor Hindernissen
F_CHASSIS_CONTROLLER	52	Chassis-Controller – steuert Chassis-Komponenten (nicht Karosserie oder Kabine)
F_ALTERNATOR_CHARGING_SYSTEM	53	Lichtmaschine/Ladegerät – primäre Ladeeinrichtung des Fahrzeugs
F_COMMUNICATIONS_UNIT_CELLULAR	54	Kommunikationseinheit, Mobilfunk – Kommunikation über Mobilfunknetz
F_COMMUNICATIONS_UNIT_SATELLITE	55	Kommunikationseinheit, Satellit – Kommunikation über Satellit
F_COMMUNICATIONS_UNIT_RADIO	56	Kommunikationseinheit, Funk – terrestrische Punkt-zu-Punkt-Kommunikation
F_AUXILIARY_DEVICE	57	Lenksäuleneinheit – sammelt Bedieneingaben von Lenkrad und Schaltern
F_FAN_DRIVE_CONTROL	58	Lüftersteuerung – steuert den Hauptkühllüfter des Motors
F_STARTER	59	Anlasser – startet den Motor, hier die Steuerung des Anlassers
F_CAB_DISPLAY	60	Kabinen-Display – erweiterte Anzeige (mehr als 30 Zeichen)
F_FILESERVER	61	Dateiserver/ Drucker – Druck- oder Speichereinheit im Netzwerk
F_ON_BOARD_DIAGNOSTIC_UNIT	62	Bordsystem-Diagnoseeinheit – Diagnosetool, ggf. dauerhaft installiert
F_ENGINE_VALVE_CONTROLLER	63	Motorventilsteuerung – steuert Einlass-/Auslassventile des Motors
F_ENDURANCE_BRAKING	64	Dauerbremsung – Gesamtheit aller verschleißfreien Bremsfunktionen
F_GAS_FLOW_MEASUREMENT	65	Gasdurchflussmessung – misst Gasdurchsatz und zugehörige Parameter
F_I_O_CONTROLLER	66	E/A-Controller – Berichts- und Steuergerät für externe Ein-/Ausgänge
F_ELECTRICAL_SYSTEM_CONTROLLER	67	Elektriksystem-Controller – z.B. Lastzentralen, Sicherungskästen
F_AFTERTREATMENT_SYSTEM_GAS_MEASUREMENT	68	Abgasnachbehandlung Gasmesstechnik – misst Gaseigenschaften vor/nach der Nachbehandlung
F_ENGINE_EMISSION_AFTERTREATMENT_SYSTEM	69	Motor-Abgasnachbehandlungssystem
F_AUXILIARY_REGENERATION_DEVICE	70	Zusätzliches Regenerationsgerät – Teil eines Nachbehandlungssystems

F_TRANSFER_CASE_CONTROL	71	Verteilergetriebesteuerung – steuert Anzahl der angetriebenen Räder (z.B. 2WD/4WD)
F_COOLANT_VALVE_CONTROLLER	72	Kühlmittelventilsteuerung – steuert Kühlmittelstrom für thermische Systeme
F_ROLLOVER_DETECTION_CONTROL	73	Überrollschutzsteuerung – erkennt drohenden Fahrzeugüberschlag
F_LUBRICATION_SYSTEM	74	Schmiersystem – pumpt Schmiermittel an Gelenke und bewegliche Teile
F_SUPPLEMENTAL_FAN	75	Zusatzlüfter – zusätzlicher Kühllüfter
F_TEMPERATURE_SENSOR	76	Temperatursensor – misst Temperatur
F_FUEL_PROPERTIES_SENSOR	77	Kraftstoffeigenschaftssensor – misst Kraftstoffeigenschaften
F_FIRE_SUPPRESSION_SYSTEM	78	Feuerlöschsystem
F_POWER_SYSTEMS_MANAGER	79	Energiesystemmanager – verwaltet Leistungsabgabe mehrerer Energiequellen
F_ELECTRIC_POWERTRAIN	80	Elektrischer Antriebsstrang – koordiniert ein elektrisches Antriebssystem
F_HYDRAULIC_POWERTRAIN	81	Hydraulischer Antriebsstrang – koordiniert ein hydraulisches Antriebssystem
F_FILE_SERVER	82	Dateiserver – Speichereinheit im Netzwerk
F_PRINTER	83	Drucker – Druckeinheit im Netzwerk
F_START_AID_DEVICE	84	Startunterstützungsgerät – z.B. Glühkerzen, Gitterheizung
F_ENGINE_INJECTION_CONTROL_MODULE	85	Motor-Einspritzsteuerung – steuert Kraftstoffeinspritzung
F_EV_COMMUNICATION_CONTROLLER	86	EV-Kommunikationscontroller – verwaltet Verbindung zu externer Ladeeinrichtung
F_DRIVER_IMPAIRMENT_DEVICE	87	Fahrerbeeinträchtigungs-Gerät – verhindert Start bei Beeinträchtigung (z.B. Alkohol-Interlock)
F_ELECTRIC_POWER_CONVERTER	88	Elektrischer Leistungswandler – Wechselrichter oder Konverter
F_SUPPLY_EQUIPMENT_COMMUNICATION_CONTROLLER	89	SECC – Kommunikationscontroller in Ladestation (EVSE)
F_VEHICLE_ADAPTER_COMMUNICATION_CONTROLLER	90	VACC – Kommunikationscontroller im Adapter zwischen EVSE und Fahrzeug
F_ACCESSORY_ELECTRIC_MOTOR_CONTROLLER	91	Elektrische Zusatzantriebssteuerung – für elektrifizierte Zubehörkomponenten
F_CURRENT_SENSOR	92	Stromsensor – misst elektrischen Strom
F_FUEL_CELL_SYSTEM	93	Brennstoffzellensystem – erzeugt Strom aus Wasserstoff
F_AUXILIARY_DISPLAY	94	Zusatzdisplay – externe oder interne Anzeige, separat zum Kabinendisplay

F_NOT_AVAILABLE	255	Nicht verfügbar – Platzhalter für unbekannte Funktion

Adapter

Keine.

Funktionsweise

Die Konstanten werden in der ISOBUS-Kommunikation verwendet, um die Funktion eines Steuergeräts gemäß dem **NAME**-Parameter zu klassifizieren. Jeder numerische Wert entspricht einem standardisierten Funktionscode, der in den entsprechenden Nachrichten (z.B. BAM, RTS/CTS oder TP.DT) gesendet wird. Durch die Verwendung dieser Konstanten wird sichergestellt, dass alle Netzwerkteilnehmer (z.B. Traktor, Anbaugerät, virtuelles Terminal) dieselbe Bedeutung für einen Funktionscode verwenden. Der Baustein selbst enthält keine ausführbare Logik, sondern dient als zentrale Definitionsquelle für die 4diac-IDE-Anwendungen.

Technische Besonderheiten

- **Typ:** Alle Konstanten sind als `BYTE` definiert.
- **Geltungsbereich:** Die Werte entsprechen den Definitionen der ISO 11783 (Teil 7) und sind branchenübergreifend anerkannt.
- **Erweiterbarkeit:** Die Liste kann bei neuen Fahrzeugfunktionen erweitert werden; der Platzhalter `F_NOT_AVAILABLE` (255) deckt unbekannte Funktionen ab.
- **Compiler-Einbindung:** Die Konstanten werden im Package `isobus::pgn::const` definiert und können in 4diac-Projekten über den GlobalConstants-Baustein referenziert werden.

Zustandsübersicht

Entfällt – da der Baustein keine Zustandsautomaten oder prozessabhängige Logik besitzt, sondern ausschließlich passive Konstanten bereitstellt.

Anwendungsszenarien

- **ISOBUS-Anwendungen:** Verwendung als Referenzwerte zur Identifizierung von Steuergerätefunktionen in der Netzwerkkommunikation.
- **Diagnose:** Analyse von Nachrichten und Zuordnung von Funktionscodes zu Geräten.
- **Visualisierung:** Anzeige von Geräteinformationen in virtuellen Terminals oder Diagnose-Tools.
- **Schnittstellendefinition:** Einheitliche Kennzeichnung von Funktionen in Softwareprojekten, die ISOBUS-basierte Steuergeräte implementieren.

Vergleich mit ähnlichen Bausteinen

Da es sich um eine globale Konstantensammlung handelt, gibt es keine direkten funktionalen Bausteine zum Vergleich. Andere GlobalConstants-Elemente könnten ähnliche Listen für andere Protokolle oder Zwecke bereitstellen, aber `Global_NAME_Functions` ist speziell auf ISOBUS-Funktionscodes zugeschnitten und bietet eine vollständige, normkonforme Auflistung.

Fazit

Der Baustein `Global_NAME_Functions` stellt eine wichtige, standardisierte Ressource für die Entwicklung von ISOBUS-Anwendungen in der 4diac-IDE dar. Er erleichtert die Implementierung, indem er alle gängigen Funktionscodes als symbolische Konstanten bereitstellt. Dadurch wird die Lesbarkeit des Codes verbessert, die Wartung vereinfacht und die Konformität mit ISO 11783 sichergestellt. Obwohl er keine aktive Steuerungslogik enthält, ist er ein unverzichtbares Werkzeug für die einheitliche Kommunikation in landwirtschaftlichen und mobilen Maschinennetzwerken.

IG0_Device_Classes

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **IG0_Device_Classes** ist eine globale Konstantendefinition im 4diac-IDE, die spezifische Geräteklassen (Device Classes) für die Fahrzeugsysteme der ISOBUS-Industriegruppe 0 (Industry Group 0) bereitstellt. Diese Konstanten werden verwendet, um Fahrzeugsysteme eindeutig zu identifizieren und zu klassifizieren, insbesondere in Anwendungen, die auf dem ISOBUS-Protokoll basieren. Die Definition

umfasst zwei Konstanten, die den Zustand „Nicht-spezifisches System“ und „Nicht verfügbar“ repräsentieren.

Schnittstellenstruktur

Da es sich um eine globale Konstantendeklaration handelt, besitzt der Baustein keine Ereignis- oder Datenschnittstellen im Sinne eines Funktionsblocks. Die Struktur ist rein deklarativ und stellt Werte für die Verwendung in anderen Bausteinen bereit.

Ereignis-Eingänge

- Keine

Ereignis-Ausgänge

- Keine

Daten-Eingänge

- Keine (globale Konstanten, nicht als Eingänge gebunden)

Daten-Ausgänge

- Keine (die Werte werden direkt über den Namen referenziert)

Adapter

- Keine

Funktionsweise

Die globalen Konstanten `DC_NON_SPECIFIC_SYSTEM` und `DC_NOT_AVAILABLE` sind vom Typ `BYTE` und tragen die Werte 0 bzw. 127. Sie werden im Anwendungscode über ihren qualifizierten Namen (z.B. `IG0_Device_Classes.DC_NON_SPECIFIC_SYSTEM`) aufgerufen und können für Vergleiche oder Zuweisungen verwendet werden. Die Konstanten sind als `VAR_GLOBAL CONSTANT` deklariert, d.h. ihre Werte sind unveränderlich und stehen im gesamten Projekt zur Verfügung. Sie dienen als Standardwerte für Geräteklassen, wenn kein spezifisches System angegeben ist oder die Information nicht verfügbar ist.

Technische Besonderheiten

- **Typ:** `BYTE` – ein 8-Bit-Unsigned-Integer, geeignet für Werte von 0 bis 255.
- **Initialwerte:**
  - `DC_NON_SPECIFIC_SYSTEM = 0` – kennzeichnet ein unspezifisches oder generisches Fahrzeugsystem.
  - `DC_NOT_AVAILABLE = 127` – kennzeichnet, dass die Geräteklasse nicht verfügbar ist.
- **Gültigkeitsbereich:** Global – die Konstanten sind im gesamten Projekt nutzbar.
- **Standardkonformität:** Die Deklaration folgt dem IEC 61499-1 Standard und spezifiziert den Paketnamen `isobus::pgn::const`.

Zustandsübersicht

Ein Zustandsdiagramm ist nicht anwendbar, da der Baustein keine Laufzeitlogik besitzt. Die Konstanten repräsentieren statische Werte und haben keinen Zustandsübergang.

Anwendungsszenarien

- **ISOBUS-Kommunikation:** Verwendung in Protokollnachrichten, um die Geräteklasse eines angeschlossenen Fahrzeugsystems zu setzen oder zu interpretieren.
- **Systemkonfiguration:** In Steuerungsanwendungen, um zwischen „nicht spezifiziertem“ und „nicht verfügbarem“ System zu unterscheiden.
- **Fallback-Logik:** Als Standardwerte, wenn keine konkreten Geräteklassen-Informationen vorliegen.

Vergleich mit ähnlichen Bausteinen

Ähnliche globale Konstantendefinitionen existieren für andere Industriegruppen (z.B. `IG1_Device_Classes` für Industriegruppe 1). Im Gegensatz zu einem Funktionsblock bietet `IG0_Device_Classes` keinerlei Verarbeitungslogik, sondern stellt lediglich Werte bereit. Es kann als Gegenstück zu einer Enumerations- oder Konstantenliste in klassischen Programmiersprachen betrachtet werden, die in 4diac als globaler Konstantenblock modelliert ist.

Fazit

Der Baustein `IG0_Device_Classes` ist eine einfache, aber wichtige Deklaration für die ISOBUS-Integration in Fahrzeugsystemen. Er stellt zwei universelle Konstanten bereit, die eine einheitliche Behandlung von unspezifischen und nicht verfügbaren Geräteklassen ermöglichen. Trotz fehlender interner Logik trägt er zur Standardisierung und Interoperabilität in der Automatisierungstechnik bei und ist ein grundlegendes Element für die Entwicklung ISOBUS-konformer Applikationen.

IG1_Device_Classes

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **IG1_Device_Classes** definiert globale Konstanten für die Fahrzeugklassen (Device Classes) der Industry Group 1 (IG1) gemäß dem ISOBUS-Standard (ISO 11783). Er dient als zentrale, wiederverwendbare Konstantendefinition, um einheitliche Werte für verschiedene Fahrzeugsysteme bereitzustellen.

Schnittstellenstruktur

Der Baustein besitzt keine dynamischen Schnittstellen (Ereignis- oder Dateneingänge/-ausgänge). Er stellt ausschließlich globale Konstanten bereit, die in anderen Bausteinen und Anwendungen referenziert werden können.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Keine.

Funktionsweise

IG1_Device_Classes enthält eine Liste von Konstanten vom Typ `BYTE`, die die standardisierten Fahrzeugklassenkennungen für die Industriegruppe 1 (Fahrzeugsysteme) abbilden. Die Konstanten werden zur Laufzeit in ISOBUS-Nachrichten (z.B. in Parameter Group Numbers) verwendet, um die Art des Fahrzeugs zu identifizieren. Jede Konstante besitzt einen symbolischen Namen und einen eindeutigen Wert.

Die definierten Konstanten sind:

- `DC_NON_SPECIFIC_SYSTEM` (Wert 0): Allgemeines, nicht spezifiziertes System.
- `DC_TRACTOR` (Wert 1): Traktor.
- `DC_TRAILER` (Wert 2): Anhänger.
- `DC_NOT_AVAILABLE` (Wert 127): Wert nicht verfügbar (für Fehlerbehandlung oder unbekannte Systeme).

Technische Besonderheiten

- **Paketzuordnung:** Die Konstanten sind dem Paket `isobus::pgn::const` zugeordnet, was die eindeutige Identifizierung innerhalb des ISOBUS-Kontexts erleichtert.
- **Datentyp:** Alle Werte sind als `BYTE` deklariert (8-Bit).
- **Organisatorische Herkunft:** Die Konfiguration stammt ursprünglich vom Hersteller „Demmler Andreas Fahrzeugbau“ und wurde von Franz Höpfinger erstellt.
- **Standard-Konformität:** Die Werte entsprechen den ISO 11783- (ISOBUS-) Vorgaben für Fahrzeugklassen der Industriegruppe 1.

Zustandsübersicht

Nicht anwendbar, da der Baustein keine Zustandsmaschine oder dynamische Prozesse besitzt.

Anwendungsszenarien

- **ISOBUS-Anwendungen:** Nutzung in Steuergeräten oder Anzeigeinheiten, um Fahrzeugtypen zu unterscheiden (z.B. zur Anzeige von Traktor- oder Anhängerdaten).
- **Datenvalidierung:** Überprüfung von eingehenden ISOBUS-Nachrichten auf gültige Fahrzeugklassenwerte.
- **Konfigurationsmodule:** Bereitstellung von Auswahlmöglichkeiten für Fahrzeugklassen in Einstellungs- oder Diagnoseoberflächen.

Vergleich mit ähnlichen Bausteinen

Da es sich um eine reine Konstantendefinition handelt, gibt es keine direkten bausteinartigen Alternativen. Andere Bausteine könnten jedoch ähnliche Konstantengruppen für andere Industriegruppen (z.B. IG0, IG2) bereitstellen. Diese werden oft ebenfalls als Global Constants implementiert, um eine konsistente Wertbasis über verschiedene Module hinweg zu gewährleisten.

Fazit

IG1_Device_Classes ist eine unverzichtbare Grundlage für ISOBUS-Anwendungen, die Fahrzeugklassen gemäß der Industriegruppe 1 unterstützen. Durch die zentrale Definition als globale Konstanten wird eine klare, wartbare und fehlerresistente Codierung ermöglicht. Die einfache Struktur ohne Schnittstellen macht den Baustein leicht verständlich und ideal für die Wiederverwendung in vielfältigen Kontexten.

IG1_NAME_Functions

• – ◦ • – ◦ • – ◦ •

Einleitung

Die GlobalConstants-Definition **IG1_NAME_Functions** stellt Konstanten für die Identifikation von Funktionen in Fahrzeugsystemen nach ISO 11783-6 (ISOBUS) bereit. Sie ist Teil der 4diac-IDE und wird als Baustein zur Bereitstellung dieser Konstanten verwendet. Die Werte sind als **BYTE**-Typ definiert und decken verschiedene Funktionsbereiche für unterschiedliche Fahrzeugsysteme (allgemeine Systeme, Traktor, Anhänger) ab.

Schnittstellenstruktur

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Die Konstanten werden als Daten-Ausgänge betrachtet, da sie einen festen, unveränderlichen Wert bereitstellen. Die folgende Tabelle listet alle definierten Konstanten auf:

Name	Typ	Wert (dezimal)	Kommentar
Non-specific System (System 0)			
`F_TACHOGRAPH`	BYTE	128	Nicht-spezifisches System, Funktion: Tachograph
`F_DOOR_CONTROLLER`	BYTE	129	Nicht-spezifisches System, Funktion: Türsteuerung
`F_ARTICULATION_TURNTABLE_CONTROL`	BYTE	130	Nicht-spezifisches System, Funktion: Gelenkdrehkranzsteuerung (für Gelenkbusse)

F_BODY_TO_VEHICLE_INTERFACE_CONTROL	BYTE	131	Nicht-spezifisches System, Funktion: Schnittstellensteuerung zwischen Fahrzeug und Aufbau
F_SLOPE_SENSOR	BYTE	132	Nicht-spezifisches System, Funktion: Hangneigungssensor
F_RETARDER_DISPLAY	BYTE	134	Nicht-spezifisches System, Funktion: Anzeige des Retarders (Antriebsstrang, Abgas, Motor)
F_DIFFERENTIAL_LOCK_CONTROLLER	BYTE	135	Nicht-spezifisches System, Funktion: Differentialsperre-Steuerung
F_LOW_VOLTAGE_DISCONNECT	BYTE	136	Nicht-spezifisches System, Funktion: Niederspannungs-Trenneinrichtung (Batteriewächter)
F_ROADWAY_INFORMATION	BYTE	137	Nicht-spezifisches System, Funktion: Straßeninformation
F_AUTOMATED_DRIVING	BYTE	138	Nicht-spezifisches System, Funktion: Automatisiertes Fahren
F_NOT_AVAILABLE	BYTE	255	Nicht-spezifisches System, Funktion: Nicht verfügbar (bis eine Zuordnung erfolgt)
Tractor (System 1)			
F_TRACTOR_FORWARD_ROAD_IMAGE_PROCESSING	BYTE	128	Traktor, Funktion: Bilderfassung der Fahrbahn (Spurmarkierungserkennung)
F_TRACTOR_FIFTH_WHEEL_SMART_SYSTEM	BYTE	129	Traktor, Funktion: Intelligentes Sattelkupplungssystem (Betrieb und Sicherheitsüberwachung)
F_TRACTOR_CATALYST_FLUID_SENSOR	BYTE	130	Traktor, Funktion: Katalysator-Flüssigkeitssensor (Temperatur, Füllstand, Qualität)
F_TRACTOR_ADAPTIVE_FRONT_LIGHTING_SYSTEM	BYTE	131	Traktor, Funktion: Adaptives Frontbeleuchtungssystem (Stadt, Land, Autobahn)
F_TRACTOR_IDLE_CONTROL_SYSTEM	BYTE	132	Traktor, Funktion: Leerlaufsteuerung (automatisches Start/Stopp im Stillstand)
F_TRACTOR_USER_INTERFACE_SYSTEM	BYTE	133	Traktor, Funktion: Benutzerschnittstelle (bidirektional, z. B. Klima, Systemparameter)
F_TRACTOR_NOT_AVAILABLE	BYTE	255	Traktor, Funktion: Nicht verfügbar (bis eine Zuordnung erfolgt)
Trailer (System 2)			
F_TRAILER_NOT_AVAILABLE	BYTE	255	Anhänger, Funktion: Nicht verfügbar (bis eine Zuordnung erfolgt)

Adapter

Keine vorhanden.

Funktionsweise

Die in `IG1_NAME_Functions` definierten Konstanten dienen dazu, eindeutige numerische Kennungen für Fahrzeugfunktionen innerhalb des ISOBUS-Kommunikationsprotokolls (ISO 11783) bereitzustellen. Jede Konstante entspricht einem bestimmten Funktionscode, der in Datenübertragungen (z. B. in PGNs – Parameter Group Numbers) verwendet wird, um Funktionen zu identifizieren. Die Aufteilung in die Gruppen (Non-specific, Tractor, Trailer) ermöglicht eine kollisionsfreie Zuordnung innerhalb der jeweiligen Systemkategorie. Der Wert 255 ist für den Fall vorgesehen, dass noch keine explizite Funktion zugewiesen wurde.

Technische Besonderheiten

- Alle Konstanten sind vom Typ `BYTE` (Wertebereich 0–255).
- Die Werte 128–138 werden für spezifische Funktionen verwendet, wobei die Zuordnung innerhalb der Systemgruppen variiert (z. B. hat `F_TACHOGRAPH` und `F_TRACTOR_FORWARD_ROAD_IMAGE_PROCESSING` denselben Dezimalwert 128, jedoch unterschiedliche Systemkontexte).
- Die Gruppen (Non-specific, Tractor, Trailer) werden über eine separate Konstantendefinition (`DC_NON_SPECIFIC_SYSTEM`, `DC_TRACTOR`, `DC_TRAILER`) adressiert, die nicht Teil dieser Datei ist.
- Die Konstanten sind **global** und unveränderlich (CONSTANT), d. h. sie können während der Laufzeit nicht geändert werden.
- Die Werte sind als Initialwerte in der XML hinterlegt und können in 4diac-Projekten direkt referenziert werden.

Zustandsübersicht

Da es sich um Konstanten handelt, existieren keine dynamischen Zustände oder Zustandsübergänge. Die Werte sind statisch und unveränderlich. Sie repräsentieren feste Funktionskennungen und unterliegen keiner Laufzeitlogik.

Anwendungsszenarien

- **Automatische Konfiguration:** Steuergeräte im Fahrzeug können anhand der Funktionscodes ihre Rolle im ISOBUS-Netzwerk erkennen und sich entsprechend konfigurieren.
- **Diagnose und Wartung:** Die Konstanten erleichtern die Interpretation von Diagnosemeldungen, da jedem Code eine klare Bedeutung zugeordnet ist.
- **Entwicklung von ISOBUS-Applikationen:** Entwickler können diese Konstanten in der 4diac-IDE verwenden, um PGNs oder andere Protokollelemente zu parametrieren.
- **Erweiterte Funktionszuordnung:** Bei der Integration neuer Fahrzeugfunktionen können die vorhandenen Codes verwendet oder neue ergänzt werden, solange die Systemkategorie beachtet wird.

Vergleich mit ähnlichen Bausteinen

Im ISOBUS-Umfeld existieren weitere GlobalConstants-Definitionen, z. B. für andere Industriegruppen (IG2, IG3 usw.) oder für allgemeine Fahrzeugparameter. Diese folgen einem ähnlichen Muster, unterscheiden sich jedoch in den konkreten Funktionscodes und Systemgruppen. Die vorliegende Definition ist speziell auf die Industriegruppe 1 (landwirtschaftliche Fahrzeuge und Anhänger) zugeschnitten. Sie bietet eine klare, wartbare Struktur, die eine einfache Erweiterung um weitere Funktionen ermöglicht.

Fazit

IG1_NAME_Functions ist ein zentraler Konstantenbaustein für die ISOBUS-Entwicklung in der 4diac-IDE. Er stellt eindeutige Funktionskennungen für verschiedene Fahrzeugsysteme bereit und unterstützt so eine effiziente und standardkonforme Kommunikation. Die klare Trennung nach Systemgruppen sowie die Verwendung von sprechenden Namen machen den Baustein gut verständlich und ermöglichen eine einfache Integration in Steuerungs- und Diagnoseanwendungen.

IG1_Source_Addresses

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **IG1_Source_Addresses** ist kein klassischer Funktionbaustein (FB) im Sinne eines prozessverarbeitenden Elements, sondern eine Sammlung **globaler Konstanten** (GlobalConstants) für die ISOBUS-Kommunikation (ISO 11783). Er definiert feste Quelladressen (Source Addresses, SA) für Geräte der **Industry Group 1** (landwirtschaftliche und mobile Maschinen). Diese Konstanten werden typischer-

weise in FBs verwendet, die auf dem ISOBUS kommunizieren, um eindeutige Adressen für Sender (z. B. Steuergeräte, Sensoren) innerhalb des Netzwerks bereitzustellen.

Die Konstanten repräsentieren die **offiziellen SA-Werte**, die von SAE (Society of Automotive Engineers) für unterschiedliche Fahrzeug- und Anhänger-Subsysteme vergeben wurden. Der Baustein selbst besitzt keine Laufzeitlogik, sondern stellt lediglich eine zentrale, wiederverwendbare Datenbasis bereit.

Schnittstellenstruktur

Da es sich um eine Gruppe von globalen Konstanten handelt, besitzt der Baustein **keine** ereignis- oder datenorientierten Schnittstellen im herkömmlichen Sinn. Stattdessen werden die Werte direkt über die **Konstantennamen** referenziert, z. B. `SA_AUTOMATED_DRIVING_CONTROLLER_1` in anderen Bausteinen.

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine Eingänge vorhanden.

Daten-Ausgänge

Die globalen Konstanten dienen als **quasi-Ausgabewerte**. Sie können in einem FB über den jeweiligen Konstantennamen gelesen werden und liefern den entsprechenden SA-Wert (Datentyp `BYTE`). Eine Auflistung aller Konstanten findet sich im Abschnitt Funktionsweise (Tabelle der wichtigsten Werte).

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Baustein definiert insgesamt **87 Konstanten** (im Wertebereich 128 bis 247), die die offiziellen Quelladressen für verschiedene ISOBUS-Geräte der Industry Group 1 festlegen. Die Konstanten sind ausschließlich lesbar und werden zur Kompilierzeit in andere Bausteine eingebunden.

Hier eine Auswahl der wichtigsten Adressen (vollständige Liste siehe `OriginalSource` der XML):

Konstante	Wert (dez)	Beschreibung
`SA_AUTOMATED_DRIVING_CONTROLLER_1`	158	Automatischer Fahrcontroller 1
`SA_AUTOMATED_DRIVING_CONTROLLER_2`	156	Automatischer Fahrcontroller 2
`SA_ELECTRIC_PROPULSION_CONTROL_UNIT_1`	239	Elektrische Antriebssteuerung 1
`SA_ADVANCED_EMERGENCY_BRAKING_SYSTEM`	160	Notbremssystem
`SA_TRAILER_1_BRIDGE`	200	Anhänger 1 – Brücke
`SA_TRAILER_2_BRAKES_ABS_EBS`	194	Anhänger 2 – Bremsen
`SA_BATTERY_PACK_MONITOR_1`	243	Batteriepack-Überwachung 1
`SA_AUXILIARY_POWER_UNIT_APU_1`	247	Hilfsaggregat (APU) 1
`SA_TACHOGRAPH`	238	Fahrtenschreiber
…	…	…

Die Werte sind als `BYTE`-Konstanten definiert und können direkt in arithmetischen oder logischen Ausdrücken sowie in Kommunikationsbausteinen (z. B. CAN-Sendern) verwendet werden.

Technische Besonderheiten

- **Konformität:** Die Konstanten entsprechen den SAE-Adressvorgaben für ISOBUS gemäß ISO 11783-7.
- **Datentyp:** Alle Konstanten sind vom Typ `BYTE` (8-Bit-Ganzzahl).
- **Wertebereich:** Die Adressen liegen im erlaubten Bereich von 128 bis 247 (für Industry Group 1).

- **Verwendung:** Da es sich um `CONSTANT`-Deklarationen handelt, können die Werte nicht zur Laufzeit verändert werden; sie sind fest eingebettet.
- **Reservierte Bereiche:** Einige Adressen (z. B. 208–227) sind für zukünftige SAE-Zuordnungen reserviert und ebenfalls als Konstanten definiert.
- **Portabilität:** Die Konstanten sind in einem separaten Paket (`isobus::pgn::const`) organisiert und können in alle FBs des Projekts importiert werden.

Zustandsübersicht

Der Baustein besitzt **keine Zustände** – es handelt sich um eine rein deklarative Definition. Es gibt keine Zustandsautomaten, Start-/Stopp-Aktionen oder zeitliche Abläufe. Die Funktion ist deterministisch und zur Laufzeit unveränderlich.

Anwendungsszenarien

- **ISOBUS-Kommunikation:** Einsatz in FBs, die Nachrichten über den CAN-Bus senden und dabei eine eigene, eindeutige Quelladresse benötigen. Die Konstanten verhindern hardcodierte Werte und erhöhen die Wartbarkeit.
- **Fahrzeugsteuerung:** Zuweisung der SA für Steuergeräte wie Motorsteuerung, Bremsen, Anhänger-Interfaces oder Fahrerassistenzsysteme.
- **Diagnose:** Nutzung des WWH-OBD-Testers (SA 241) in Diagnosefunktionen.
- **Anhänger- und Trailermanagement:** Adressierung mehrerer Anhänger (bis zu 5) mit eigenen Brücken, Licht-, Brems- und Cargo-Adressen.

Vergleich mit ähnlichen Bausteinen

Es gibt weitere Konstantenpakete in ISOBUS, z. B. für andere Industry Groups (IG0, IG2, IG3) oder für spezifische Parameter-Gruppen (PGN). Der Unterschied besteht in den zugewiesenen Adressbereichen und den verwendeten Gerätetypen. Diese Konstantengruppe ist speziell für die Landtechnik (IG1) optimiert und enthält typische landwirtschaftliche Geräte.

Gegenüber einem FB, der z. B. eine dynamische Adressvergabe implementiert, bietet dieser Baustein **statische, festgelegte Werte** – ideal für Systeme mit festen Rollen. Für dynamische Konfigurationen müssten zusätzliche Logikbausteine verwendet werden.

Fazit

Der Baustein **IG1_Source_Addresses** ist eine unverzichtbare, weil zentrale Datenquelle für die ISOBUS-Entwicklung in der 4diac-IDE. Er bündelt die offiziellen Quelladressen der Industry Group 1 in einer sauberen, wiederverwendbaren Form. Auch wenn er selbst keine aktive Steuerlogik enthält, vereinfacht er die korrekte und konsistente Adressierung erheblich und beugt Fehlern durch hardcodierte Werte vor. Durch seine klare Struktur und die umfassenden Kommentare ist er sowohl für Neulinge als auch für erfahrene Entwickler leicht nachvollziehbar und erweiterbar.

IG2_Device_Classes

• – ◦ • – ◦ • – ◦ •

Einleitung

Der GlobalConstants-Baustein **IG2_Device_Classes** definiert eine Sammlung von Konstanten für Geräteklassen (Device Classes) gemäß dem ISOBUS-Standard (ISO 11783). Diese Konstanten sind für die Industriegruppe 2 (Industry Group 2) spezifisch und kennzeichnen die Art eines Fahrzeugsystems (z. B. Traktor, Erntemaschine, Anhänger). Sie werden typischerweise in der Kommunikation zwischen Traktor und Anbaugeräten verwendet, um die Typklassifizierung des angeschlossenen Geräts zu übermitteln.

Schnittstellenstruktur

Da es sich um einen GlobalConstants-Baustein handelt, existieren keine Ein-/Ausgänge, Ereignisse oder Adapter. Der Baustein stellt ausschließlich Konstanten bereit, die als globale Variablen in der IEC 61499-Umgebung genutzt werden können.

Ereignis-Eingänge

Nicht vorhanden.

Ereignis-Ausgänge

Nicht vorhanden.

Daten-Eingänge
Nicht vorhanden.

Daten-Ausgänge
Nicht vorhanden.

Adapter
Nicht vorhanden.

Funktionsweise
Der Baustein definiert eine Reihe von benannten Konstanten vom Typ **BYTE**, die jeweils einen bestimmten numerischen Wert und eine dazugehörige Bezeichnung besitzen. Die Werte entsprechen den offiziellen Geräteklassen-Codierungen des ISOBUS-Standards. Sie werden typischerweise in Kommunikationsprotokollen verwendet, um die Art eines Fahrzeugs oder Anbaugeräts zu identifizieren. Beispielsweise steht der Wert 1 für **Tractor**, 7 für **Harvesters** und 127 für **Not Available**.

Die folgende Tabelle zeigt die definierten Konstanten:

Konstante	Wert (BYTE)	Bedeutung
DC_NON_SPECIFIC_SYSTEM	0	Nicht spezifisches System
DC_TRACTOR	1	Traktor
DC_TILLAGE	2	Bodenbearbeitung
DC_SECONDARY_TILLAGE	3	Sekundäre Bodenbearbeitung
DC_PLANTERS_SEEDERS	4	Pflanzmaschinen / Sägeräte
DC_FERTILIZERS	5	Düngerstreuer
DC_SPRAYERS	6	Spritzgeräte
DC_HARVESTERS	7	Erntemaschinen
DC_ROOT_HARVESTERS	8	Wurzelfrüchte-Erntemaschinen
DC_FORAGE	9	Futtererntemaschinen
DC_IRRIGATION	10	Bewässerung
DC_TRANSPORT_TRAILER	11	Transport / Anhänger
DC_FARM_YARD_OPERATIONS	12	Hofwirtschaft
DC_POWERED_AUXILIARY_DEVICES	13	Angetriebene Hilfsgeräte
DC_SPECIAL_CROPS	14	Spezialkulturen
DC_EARTH_WORK	15	Erdarbeiten
DC_SKIDDER	16	Skidder (Rückeschlepper)
DC_SENSOR_SYSTEMS	17	Sensorsysteme
DC_TIMBER_HARVESTERS	19	Holzerntemaschinen
DC_FORWARDERS	20	Forwarder
DC_TIMBER_LOADERS	21	Holzverlader
DC_TIMBER_PROCESSING_MACHINES	22	Holzverarbeitungsmaschinen
DC_MULCHERS	23	Mulcher
DC_UTILITY_VEHICLES	24	Nutzfahrzeuge
DC_SLURRY_MANURE_APPLICATORS	25	Gülle-/Mistausbringer

DC_FEEDERS_MIXERS	26	Fütterungs-/Mischgeräte
DC_WEEDERS	27	Unkrautbekämpfung (nicht chemisch)
DC_TURF_AND_LAWN_CARE_MOWERS	28	Rasen- und Rasenpflegemäher
DC_PRODUCT_MATERIAL_HANDLING	29	Produkt-/Materialhandhabung
DC_NOT_AVAILABLE	127	Nicht verfügbar

Technische Besonderheiten

- Die Konstanten sind als **BYTE** deklariert und besitzen feste Werte im Bereich von 0 bis 127.
- Der Wert 127 ist als DC_NOT_AVAILABLE reserviert und signalisiert, dass keine gültige Geräteklasse vorliegt.
- Einige Werte (z. B. 18, 30–126) sind im Standard nicht belegt und daher nicht definiert. Dies bietet Raum für zukünftige Erweiterungen.
- Der Baustein ist als GlobalConstants konzipiert und kann in FB-Netzwerken oder SFC-Programmen über den Namen direkt referenziert werden.
- Die Konstanten sind Teil der ISOBUS-Datenkommunikation (PGN-Konstanten) und erleichtern eine standardkonforme Implementierung.

Zustandsübersicht

Für einen GlobalConstants-Baustein existiert keine Zustandsautomaten- oder Zustandslogik. Die Konstanten sind statische Werte und ändern sich während der Laufzeit nicht.

Anwendungsszenarien

- **ISOBUS-Kommunikation:** In einer ISOBUS-Umgebung kann ein Steuergerät (ECU) die Geräteklasse seines Anbaugeräts mithilfe dieser Konstanten in den dafür vorgesehenen PGNs (Parameter Group Numbers) kodieren.
- **Typidentifikation:** Ein Traktor kann anhand dieser Konstanten automatisch erkennen, ob es sich bei einem angeschlossenen Gerät um einen Pflug, eine Sämaschine oder eine Erntemaschine handelt.
- **Diagnose und Konfiguration:** Bei der Inbetriebnahme oder Fehleranalyse kann die Geräteklasse zur schnellen Identifikation des Gerätetyps herangezogen werden.
- **Entwicklung von ISOBUS-Anwendungen:** Bei der Erstellung von 4diac-Applikationen können diese Konstanten direkt als symbolische Namen verwendet werden, anstatt magische Zahlenwerte zu verwenden.

Vergleich mit ähnlichen Bausteinen

Es existieren weitere GlobalConstants-Bausteine für andere Industriegruppen (z. B. IG1_Device_Classes für Industriegruppe 1), die ähnliche Konstanten für andere Anwendungsbereiche definieren. Der Unterschied liegt in den zugeordneten Werten und den spezifischen Geräteklassen. Während IG1 eher allgemeine industrielle Anwendungen abdeckt, fokussiert IG2 auf Fahrzeug- und Agrarsysteme.

Fazit

Der GlobalConstants-Baustein **IG2_Device_Classes** stellt eine standardkonforme und wartungsfreundliche Möglichkeit dar, die Geräteklassen im ISOBUS-Kontext zu verwenden. Durch die Bereitstellung symbolischer Namen wird die Lesbarkeit und Fehleranfälligkeit von Applikationen reduziert. Die fest definierten Werte gewährleisten Kompatibilität mit der ISOBUS-Norm und ermöglichen eine klare Kommunikation zwischen verschiedenen Geräten.

IG2_NAME_Functions

• – ◦ • – ◦ • – ◦ •

Einleitung

Die globale Konstantendefinition IG2_NAME_Functions stellt eine Sammlung von Funktionscodes für die ISO 11783-Identifikation (ISOBUS NAME) bereit. Sie definiert die numerischen Werte (als BYTE) für die unterschiedlichen Steuerungs- und Messfunktionen, die in landwirtschaftlichen Maschinen und Geräten verwendet werden. Diese Konstanten werden typischerweise in Softwareprojekten eingesetzt, die ISOBUS-kompatible Steuergeräte implementieren oder mit deren Nachrichten umgehen.

Schnittstellenstruktur

Da es sich um einen `GlobalConstants`-Baustein handelt, besitzt er keine Ereignis- oder Datenports. Die folgenden Abschnitte sind daher nicht anwendbar.

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge

Keine vorhanden.

Adapter

Keine vorhanden.

Funktionsweise

Die Konstanten dienen als symbolische Namen für Werte, die im ISO 11783-Standard (ISOBUS) für die Klassifizierung von Funktionen verwendet werden. Sie sind im Namespace `isobus::pgn::const` definiert und können direkt in IEC 61499- oder IEC 61131-Programmen referenziert werden, ohne dass magische Zahlen verwendet werden müssen. Jede Konstante ist eindeutig benannt und beinhaltet in ihrem Namen den Bezug auf das jeweilige System (z. B. Traktor, Pflanzenschutz, Erntemaschine) und die Funktion (z. B. Maschinensteuerung, Durchflussmessung).

Die Werte sind als `BYTE` (0-255) definiert. Der Wert 255 wird generell für „Nicht verfügbar“ verwendet, während andere Werte spezifischen Funktionen zugeordnet sind. Die Zuordnung folgt den Tabellen des ISO 11783-5 Standards für NAME-Datenelemente.

Technische Besonderheiten

- Die Konstanten sind als globale Konstanten deklariert und stehen damit in allen Bausteinen und Programmen zur Verfügung, die das Paket einbinden.
- Die Namen sind hierarchisch aufgebaut: `F_<SYSTEM>_<FUNKTION>`, wobei `SYSTEM` das jeweilige Maschinensystem (z. B. `TRACTOR`, `SPRAYERS`) und `FUNKTION` die spezifische Aufgabe (z. B. `MACHINE_CONTROL`, `PRODUCT_FLOW`) kennzeichnet.
- Viele Systeme haben gemeinsame Funktionscodes (z. B. Wert 132 für „Maschinensteuerung“), aber durch die Systemkomponente im Namen wird Eindeutigkeit gewährleistet.
- Die Werte entsprechen den offiziellen Definitionen der ISO 11783, Abschnitt 5 (Datenidentifikation).

Zustandsübersicht

Nicht zutreffend – ein `GlobalConstants`-Baustein besitzt keine Zustände oder Zustandsautomaten.

Anwendungsszenarien

- **ISOBUS-Steuergeräteentwicklung**: Verwendung der Konstanten zum Setzen des Funktionscodes im NAME-Eintrag eines Steuergeräts.
- **Nachrichtenanalyse**: Interpretation von NAME-Nachrichten, die von anderen Geräten empfangen werden, mithilfe der Konstanten.
- **Programmierung von ISOBUS-Anwendungen**: Einbinden der Konstantendatei in ein Projekt, um Funktionen wie „Saatmengenregelung“ oder „Produktflussmessung“ eindeutig zu identifizieren.
- **Diagnose-Tools**: Anzeige der Funktion eines entfernten Geräts basierend auf den empfangenen NAME-Informationen.

Vergleich mit ähnlichen Bausteinen

Im ISOBUS-Umfeld gibt es weitere Global-Constants-Definitionen, z. B. für Geräteklassen (Device Class), Industry Groups oder andere Datenelemente. `IG2_NAME_Functions` ist speziell auf die Funktionscodes der Industry Group 2 (landwirtschaftliche Maschinen) ausgerichtet. Andere Konstantensammlungen könnten allgemeinere oder zusätzliche kodierte Werte enthalten, aber diese bieten eine vollständige, standardkonforme Abdeckung der wichtigsten Funktionen.

Fazit

`IG2_NAME_Functions` ist eine umfassende und klar strukturierte Konstantensammlung für ISOBUS-Funktionscodes. Sie erleichtert die Implementierung und Wartung von ISOBUS-Software erheblich, indem sie sprechende Namen für feste Werte bereitstellt und so Fehler durch falsche Zahlenwerte minimiert. Die Einhaltung der ISO 11783-Standards wird durch die direkte Übernahme der definierten Werte gewährleistet.

IG2_Source_Addresses

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `IG2_Source_Addresses` ist eine Sammlung von globalen Konstanten, die die festgelegten **Source Addresses** für die **Industry Group 2 (IG2)** im ISOBUS-Kommunikationsprotokoll (ISO 11783) definiert. Diese Adressen werden für verschiedene Funktionen und Steuergeräte in landwirtschaftlichen Maschinen verwendet und sind je nach Zweck entweder fest reserviert oder für die Selbstkonfiguration vorgesehen. Die Konstanten sind als `BYTE`-Typ (8-Bit-Wert) mit initialen Werten von 128 bis 247 definiert und finden in der Automatisierungstechnik für Traktor-Implement-Kommunikation Anwendung.

Schnittstellenstruktur

Der Baustein besitzt keine dynamischen Schnittstellen wie Ereignisse oder Datenein-/ausgänge, da er ausschließlich globale Konstanten bereitstellt. Dennoch werden die Konstanten im Folgenden als **Daten-Ausgänge** aufgeführt, da sie von anderen Funktionsblöcken im gesamten Projekt ausgelesen werden können.

Ereignis-Eingänge

Nicht vorhanden. Es gibt keine erwartbaren Ereignisse, die diesen Baustein auslösen.

Ereignis-Ausgänge

Nicht vorhanden. Der Baustein erzeugt keine Ereignisse, da er nur statische Konstanten liefert.

Daten-Eingänge

Keine. Es werden keine Werte von außen übergeben.

Daten-Ausgänge

Die folgenden Konstanten sind als globale Datenwerte verfügbar:

Name	Wert	Kommentar
`SA_RESERVED_128`	128	Adresse 128 bis 235 sind von ISO für die selbstkonfigurierbare Adressierung reserviert.
`SA_DATA_LOGGER`	236	Datenlogger – Server, der die Protokollierungsfunktion gemäß ISO 11783-10 bereitstellt.
`SA_TIM_SERVER`	237	TIM-Server – Steuerfunktion für Tractor Implement Management (Operator-Assistance-System).
`SA_SEQUENCE_CONTROLLER`	238	Sequenzcontroller – Master im Sequenzsteuerungssystem nach ISO 11783-14.
`SA_POSITION_CONTROL`	239	Positionssteuerung – Mehr-Achsen-Positionssteuerung beweglicher Geräteelemente.
`SA_TRACTOR_ECU`	240	Traktor-ECU – Gateway zwischen Traktor und Implement-Bus.
`SA_TAILINGS_MONITORING`	241	Monitoring der Rücklaufmenge (unausgedroschenes Material) zur Dreschmaschine.

SA_HEADER_CONTROL	242	Steuerung der Haspeldrehzahl und Materialzufuhrrate des Schneidwerks.
SA_PRODUCT_LOSS_MONITORING	243	Überwachung des Produktverlusts während des Ernteprozesses.
SA_PRODUCT_MOISTURE_SENSING	244	Überwachung der Feuchtigkeit des Ernteguts.
SA_NON_VIRTUAL_TERMINAL_DISPLAY_IMPLEMENT_BUS	245	Nicht-Virtual-Terminal-Display am Implement-Bus – Anzeige in der Kabine.
SA_OPERATOR_CONTROLS_MACHINE_SPECIFIC	246	Maschinenspezifische Bedienerelemente – nicht-auxiliäre Bedienereingaben.
SA_TASK_CONTROL_MAPPING_COMPUTER	247	Task-Control-Rechner – sendet, empfängt und loggt Prozessdaten.

Adapter

Keine. Es existieren keine Adapter-Schnittstellen.

Funktionsweise

`IG2_Source_Addresses` ist ein reiner Konstanten-Baustein. Er definiert feste numerische Werte, die als symbolische Namen im gesamten Projekt verwendet werden können. Die Werte entsprechen den im ISOBUS-Standard vergebenen Source-Adressen für spezifische Funktionen innerhalb der Industrie- und Landwirtschaftsgruppe 2. Durch die Verwendung dieser Konstanten wird sichergestellt, dass Adressen nicht hart verdrahtet oder fehleranfällig in mehreren Bausteinen wiederholt werden müssen. Die Konstanten sind global verfügbar und können von jedem anderen Funktionsblock oder Programm gelesen werden.

Technische Besonderheiten

- Die Adressen von 128 bis 235 sind für die **selbstkonfigurierbare Adressvergabe** reserviert. Geräte der IG2, die eine bevorzugte Adresse verwenden, müssen in der Lage sein, ihre Adresse automatisch zu konfigurieren.
- Ab Adresse 236 beginnt der Bereich der **fest zugewiesenen** Source-Adressen für spezielle Funktionen (Datenlogger, TIM usw.).
- Alle Konstanten sind vom Typ `BYTE` und besitzen einen Initialwert im Bereich 0..255.
- Der Baustein stellt keine Laufzeitlogik dar; er definiert lediglich benannte Konstanten, die zur Verbesserung der Lesbarkeit und Wiederverwendbarkeit des Codes beitragen.

Zustandsübersicht

Da der Baustein keine Zustandsautomaten oder Verhaltenslogik besitzt, gibt es keine Zustände. Die Werte sind statisch und unveränderlich.

Anwendungsszenarien

- Implementierung eines ISOBUS-Kommunikationsmoduls, bei dem auf die festgelegten Source-Adressen von verschiedenen Funktionsblöcken zugegriffen werden muss.
- Entwicklung von Traktor-Implement-Gateways oder Steuergeräten, die die standardisierten Adressen benötigen, um mit anderen Komponenten zu kommunizieren.
- Einbindung in ein größeres Projekt zur Vereinfachung der Wartung, wenn sich Adressen in Zukunft ändern sollten – dann müssen nur die Konstantenwerte angepasst werden.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Konstanten-Bausteinen, die beispielsweise für Netzwerkprotokolle oder Konfigurationsparameter existieren, spezialisiert sich `IG2_Source_Addresses` auf die spezifischen Source-Adressen der ISOBUS-Industriegruppe 2. Andere Bausteine könnten z.B. `IG1_Source_Addresses` für Gruppe 1 oder allgemeine Konstanten für PGN-Nummern sein. Der vorliegende Baustein enthält jedoch nur eine begrenzte, aber klar definierte Menge an Werten, die direkt aus dem Standard übernommen wurden.

Fazit

Der Baustein `IG2_Source_Addresses` ist eine nützliche, modularisierte Sammlung von ISOBUS-Standardadressen für die Industriegruppe 2. Er erhöht die Wartbarkeit und Lesbarkeit von Automatisierungsprojekten in der Landtechnik, indem er zentrale Konstanten bereitstellt. Da er keine Dynamik besitzt, ist er einfach zu integrieren und benötigt keine besondere Laufzeitressourcen. Durch die klare Zuordnung der Werte zu den jeweiligen Funktionen trägt er zur Fehlervermeidung und Standardkonformität bei.

IG3_Device_Classes

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `IG3_Device_Classes` ist eine globale Konstantendefinition (Global Constants) gemäß IEC 61499. Er stellt eine Sammlung von Konstanten bereit, die die spezifischen Geräteklassen (Vehicle Systems) der Industry Group 3 (IG3) im ISOBUS-Protokoll (ISO 11783) definieren. Diese Konstanten werden üblicherweise verwendet, um den Fahrzeugtyp oder die Geräteklasse in Kommunikationsprotokollen (z.B. über PGNs) zu identifizieren. Der Baustein ist als Parameterliste oder Konstantenquelle für Applikationen gedacht, die mit land- oder forstwirtschaftlichen Maschinen arbeiten.

Schnittstellenstruktur

Der Baustein `IG3_Device_Classes` besitzt keine Ein-/Ausgangsschnittstellen im klassischen Sinne eines Funktionsblocks. Es handelt sich um eine Auflistung globaler Konstanten, die direkt im System verfügbar sind. Daher entfallen Ereignis- und Daten-Eingänge sowie -Ausgänge und Adapter.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Keine.

Funktionsweise

Der Baustein definiert eine Reihe von benannten Konstanten vom Typ `BYTE`. Jede Konstante repräsentiert einen spezifischen Fahrzeug- oder Gerätetyp gemäß der ISOBUS-Spezifikation (Industry Group 3). Die Werte sind fest und unveränderlich. Durch die Deklaration als globale Konstanten können sie in anderen Bausteinen oder Applikationen referenziert werden, ohne dass sie explizit übergeben werden müssen.

Die Konstantenwerte und ihre Bedeutungen sind:

Konstante	Wert	Beschreibung
`DC_NON_SPECIFIC_SYSTEM`	0	Nicht spezifisches System
`DC_SKID_STEER_LOADER`	1	Skid-Steer-Lader
`DC_ARTICULATED_DUMP_TRUCK`	2	Gelenk-Muldenkipper
`DC_BACKHOE`	3	Baggerlader
`DC_CRAWLER`	4	Raupenfahrzeug
`DC_EXCAVATOR`	5	Bagger
`DC_FORKLIFT`	6	Gabelstapler
`DC_FOUR_WHEEL_DRIVE_LOADER`	7	Allrad-Lader

DC_GRADER	8	Straßenhobel
DC_MILLING_MACHINE	9	Fräsmaschine
DC_RECYCLER_AND_SOIL_STABILIZER	10	Recycler und Bodenstabilisator
DC_BINDING_AGENT_SPREADER	11	Bindemittelstreuer
DC_PAVER	12	Straßenfertiger
DC_FEEDER	13	Beschicker
DC_SCREENING_PLANT	14	Siebanlage
DC_STACKER	15	Stapelgerät
DC_ROLLER	16	Walze
DC_CRUSHER	17	Brecher
DC_NOT_AVAILABLE	127	Nicht verfügbar

Die Konstanten können direkt in Programmen und Funktionsblöcken verwendet werden, z.B. um einen empfangenen Byte-Wert einer Geräteklasse zuzuordnen oder um eine solche Identifikation zu senden.

Technische Besonderheiten

- **Typ**: `BYTE` – Wertebereich 0–255.
- **Gültigkeitsbereich**: Global, d.h. sie sind im gesamten Projekt einsehbar und nutzbar.
- **Unveränderlichkeit**: Die Konstanten sind als `CONSTANT` deklariert und können zur Laufzeit nicht verändert werden.
- **Standardkonformität**: Die Werte entsprechen den von ISO 11783 (ISOBUS) definierten Geräteklassen für Industry Group 3 (Vehicle Systems).
- **Paket**: Ursprünglich definiert im Paket `isobus::pgn::const`, was auf die Verwendung im Kontext von PGN-Konstanten hinweist.
- **Dateiformat**: Die Definition erfolgt im XML-Format für die 4diac-IDE und wird dort als Global Constants verwaltet.

Zustandsübersicht

Da es sich um eine reine Konstantendefinition handelt, besitzt der Baustein keinen Zustandsautomaten und keine Laufzeitlogik. Die Konstanten sind permanent verfügbar und ändern ihren Wert nicht während der Programmausführung.

Anwendungsszenarien

- **Fahrzeugidentifikation**: Wenn ein ISOBUS-fähiges Steuergerät (ECU) seine Geräteklasse in einer Botschaft (z.B. PGN) senden möchte, kann der entsprechende Konstantenwert (z.B. `DC_EXCAVATOR`) verwendet werden.
- **Empfang und Validierung**: Beim Empfang einer Nachricht mit einer Geräteklassen-Identifikation kann der Wert gegen die Konstanten verglichen werden, um die Fahrzeugart zu ermitteln und entsprechende Reaktionen auszulösen.
- **Konfiguration**: In Projekten, die unterschiedliche Maschinentypen unterstützen, können diese Konstanten zur Konfiguration von Parametern oder zur Auswahl von Steuerlogiken verwendet werden.
- **Datenvisualisierung**: Anzeige des Fahrzeugtyps in einer Bedienoberfläche durch Zuordnung des numerischen Werts zu einer Beschreibung.

Vergleich mit ähnlichen Bausteinen

Im ISOBUS-Kontext gibt es weitere Global-Constants-Bausteine für andere Industry Groups (z.B. IG0, IG1, IG2). Diese unterscheiden sich in den definierten Konstantenwerten und deren Bedeutung. `IG3_Device_Classes` ist speziell für die Gruppe 3 (Fahrzeugsysteme) ausgelegt. Andere Bausteine könnten z.B. `IG0_Device_Classes` oder `IG1_Device_Classes` heißen und andere Listen enthalten. Der hier beschriebene Baustein stellt die spezifischen Geräteklassen für land- und baumaschinenähnliche Fahrzeuge bereit.

Fazit

Der Baustein `IG3_Device_Classes` ist eine einfache, aber wichtige Komponente für ISOBUS-basierte Anwendungen in der 4diac-IDE. Durch die Bereitstellung standardkonformer Konstanten für Geräteklassen wird eine einheitliche und fehlerfreie Kommunikation im Agrar- und Baubereich unterstützt. Er ist unverzichtbar für Entwickler, die Fahrzeugidentifikationen in ihren Steuerungssystemen integrieren müssen. Die Verwendung als globale Konstanten erleichtert die Wartung und erhöht die Lesbarkeit des Codes erheblich.

IG3_NAME_Functions

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **IG3_NAME_Functions** ist ein GlobalConstants-Baustein, der eine Sammlung von globalen Konstanten für die ISO-11783-Kommunikation (ISOBUS) bereitstellt. Er definiert Funktionscodes (Function Values) für Geräte in der **Industriegruppe 3** – Baumaschinen und Anbaugeräte. Die Konstanten werden typischerweise verwendet, um Steuergeräte (ECUs) oder Sensoren eindeutig einer bestimmten Funktion innerhalb eines Maschinensystems zuzuordnen.

Die Werte sind als BYTE-Konstanten definiert und decken verschiedene Maschinentypen ab, von generischen Systemen (Non-specific System) über Bagger, Lader, Grader bis hin zu Spezialmaschinen wie Recyclern oder Betonpumpen.

Schnittstellenstruktur

Da es sich um einen GlobalConstants-Baustein handelt, besitzt er **keine** Ein- oder Ausgänge im Sinne eines Funktionsbausteins. Die bereitgestellten Werte sind globale Konstanten, die direkt im System verwendet werden können.

Ereignis-Eingänge

Keine

Ereignis-Ausgänge

Keine

Daten-Eingänge

Keine

Daten-Ausgänge

Keine

Adapter

Keine

Globale Konstanten

Die folgende Tabelle listet alle im Baustein definierten Konstanten auf. Der Typ ist jeweils `BYTE`, der Wert entspricht dem Initialwert.

Name	Wert (dezimal)	System	Funktion / Beschreibung
F_SUPPLEMENTAL_ENGINE_CONTROL_SENSING	128	Non-specific System	Supplemental Engine Control Sensing
F_LASER_RECEIVER	129	Non-specific System	Laser Receiver
F_LAND_LEVELING_SYSTEM_OPERATOR_INTERFACE	130	Non-specific System	Land Leveling System Operator Interface – Bedienoberfläche für das Landplaniersystem
F_LAND_LEVELING_ELECTRIC_MAST	131	Non-specific System	Land Leveling Electric Mast

F_SINGLE_LAND_LEVELING_SYSTEM_SUPERVISOR	132	Non-specific System	Einzelüberwachung des Landplaniersystems
F_LAND_LEVELING_SYSTEM_DISPLAY	133	Non-specific System	Anzeige des Landplaniersystems
F_LASER_TRACER	134	Non-specific System	Laser Tracer
F_LOADER_CONTROL	135	Non-specific System	Steuerung für Lader
F_SLOPE_SENSOR	136	Non-specific System	Neigungssensor – misst die Neigung entlang einer Achse
F_LIFTARM_CONTROL	137	Non-specific System	Hubarmsteuerung – steuert Heben und Kippen bei Ladegeräten
F_SUPPLEMENTAL_SENSOR_PROCESSING_UNITS	138	Non-specific System	Zusätzliche Sensorverarbeitungseinheiten – I/O-Modul für Datenerfassung
F_HYDRAULIC_SYSTEM_PLANNER	139	Non-specific System	Hydrauliksystem-Planer – koordiniert mehrere Ventilsteuerungen
F_HYDRAULIC_VALVE_CONTROLLER	140	Non-specific System	Hydraulikventilsteuerung – steuert Ölfluss zu einem Zylinder
F_JOYSTICK_CONTROL	141	Non-specific System	Joystick-Steuerung
F_ROTATION_SENSOR	142	Non-specific System	Rotationssensor – misst den Drehwinkel um eine Achse
F_SONIC_SENSOR	143	Non-specific System	Ultraschallsensor – misst Entfernung durch Ultraschall-Echo
F_SURVEY_TOTAL_STATION_TARGET	144	Non-specific System	Ziel für Totalstation – für Vermessungsaufgaben
F_HEADING_SENSOR	145	Non-specific System	Kurssensor – misst den Azimut des Fahrzeugs
F_ALARM_DEVICE	146	Non-specific System	Alarmgerät – gibt akustische und/oder visuelle Warnungen aus
F_NOT_AVAILABLE	255	Non-specific System	Nicht verfügbar – Platzhalter für noch nicht zugewiesene Funktionen
F_SKID_STEER_LOADER_MAIN_CONTROLLER	128	Skid Steer Loader	Hauptcontroller für Kompaktlader
F_SKID_STEER_LOADER_NOT_AVAILABLE	255	Skid Steer Loader	Nicht verfügbar
F_ARTICULATED_DUMP_TRUCK_NOT_AVAILABLE	255	Articulated Dump Truck	Nicht verfügbar
F_BACKHOE_NOT_AVAILABLE	255	Backhoe	Nicht verfügbar
F_CRAWLER_BLADE_CONTROLLER	128	Crawler	Plattensteuerung – steuert die Plattenhöhe

F_CRAWLER_NOT_AVAILABLE	255	Crawler	Nicht verfügbar
F_EXCAVATOR_SLOPE_SENSOR	128	Excavator	Neigungssensor – misst die Neigung entlang einer Achse
F_EXCAVATOR_NOT_AVAILABLE	255	Excavator	Nicht verfügbar
F_FORKLIFT_NOT_AVAILABLE	255	Forklift	Nicht verfügbar
F_FOUR_WHEEL_DRIVE_LOADER_NOT_AVAILABLE	255	Four Wheel Drive Loader	Nicht verfügbar
F_GRADER_HFWD_CONTROLLER	128	Grader	HFWD-Controller – hydraulische Vorderrad-Antriebssteuerung
F_GRADER_NOT_AVAILABLE	255	Grader	Nicht verfügbar
F_MILLING_MACHINE_NOT_AVAILABLE	255	Milling Machine	Nicht verfügbar – Fräse zum Entfernen von Straßenbelag
F_RECYCLER_AND_SOIL_STABILIZER_NOT_AVAILABLE	255	Recycler and Soil Stabilizer	Nicht verfügbar – Recycler zur Aufbereitung von Asphalt
F_BINDING_AGENT_SPREADER_NOT_AVAILABLE	255	Binding Agent Spreader	Nicht verfügbar – Streuer für Bindemittel
F_PAVER_NOT_AVAILABLE	255	Paver	Nicht verfügbar – Straßenfertiger
F_FEEDER_NOT_AVAILABLE	255	Feeder	Nicht verfügbar – Beschicker für Fertiger
F_SCREENING_PLANT_NOT_AVAILABLE	255	Screening Plant	Nicht verfügbar – Siebanlage
F_STACKER_NOT_AVAILABLE	255	Stacker	Nicht verfügbar – Absetzer
F_ROLLER_NOT_AVAILABLE	255	Roller	Nicht verfügbar – Walze
F_CRUSHER_NOT_AVAILABLE	255	Crusher	Nicht verfügbar – Brecher

Funktionsweise

Der Baustein definiert Konstanten, die in der ISOBUS-Parametrierung zur Identifizierung der Funktion eines angeschlossenen Geräts verwendet werden. Die Konstanten füllen das Feld „Function“ im Parameter Group Number (PGN)-Kontext. Durch die Verwendung dieser Konstanten kann eine Applikation eindeutig erkennen, welche Aufgabe ein Gerät innerhalb einer Maschine übernimmt (z. B. Neigungssensor, Joystick, Motorsteuerung).

Der Wert 255 (dezimal) ist als „Not Available“ reserviert und wird verwendet, wenn einem Gerät noch keine spezifische Funktion zugewiesen wurde. Spezifische Werte im Bereich 128–146 sind für die Industriegruppe 3 definiert.

Technische Besonderheiten

- **Typ:** Alle Konstanten sind vom Typ `BYTE`.
- **Wertebereich:** Die definierten Werte liegen zwischen 128 und 255. Die obere Grenze (255) ist als „Not Available“ definiert.
- **Standardkonformität:** Die Konstanten basieren auf dem ISO-11783-Standard (ISOBUS) und sind speziell für die Industriegruppe 3 (Baumaschinen) vorgesehen.
- **Globaler Gültigkeitsbereich:** Da es sich um einen `GlobalConstants`-Baustein handelt, sind die Konstanten im gesamten Projekt verfügbar, sofern der Baustein eingebunden wird.

- **Versionierung:** Der Baustein ist auf Version 1.0 datiert und enthält einen Verweis auf den ursprünglichen Quellcode (`OriginalSource`).

Zustandsübersicht

Ein `GlobalConstants`-Baustein besitzt keinen inneren Zustand. Er stellt lediglich konstante Werte bereit, die zur Compile-Zeit feststehen. Daher existiert keine Zustandsmaschine oder Laufzeitlogik.

Anwendungsszenarien

- **Maschinenvernetzung:** In einem ISOBUS-Netzwerk können Steuergeräte anhand ihrer Funktionscodes identifiziert werden. Der Baustein liefert die Mapping-Konstanten für die Softwareentwicklung.
- **Parametrierung von ECUs:** Bei der Inbetriebnahme von Baumaschinen können diese Konstanten verwendet werden, um Geräte eindeutig zu konfigurieren (z. B. Zuweisung eines Joysticks zur Steuerung).
- **Diagnose und Fehlerbehebung:** Die Konstanten erlauben eine klare Zuordnung von Fehler- und Statusmeldungen zu bestimmten Maschinenkomponenten.
- **Plattformunabhängige Entwicklung:** Durch die Verwendung standardisierter Funktionscodes wird die Entwicklung von Applikationen für verschiedene Maschinenhersteller vereinfacht.

Vergleich mit ähnlichen Bausteinen

In der 4diac-IDE existieren ähnliche `GlobalConstants`-Bausteine für andere Industriegruppen (z. B. `IG1_NAME_Functions` für Landwirtschaft oder `IG2_NAME_Functions` für Forstmaschinen). Diese unterscheiden sich in den definierten Konstanten und den zugehörigen Maschinentypen. Der vorliegende Baustein deckt ausschließlich die Anforderungen der Baumaschinenbranche ab und enthält daher spezifische Funktionen wie Lader, Bagger, Grader, Walzen usw.

Fazit

`IG3_NAME_Functions` ist ein essenzieller Baustein für die Entwicklung von ISOBUS-Applikationen im Bereich Baumaschinen. Durch die Bereitstellung standardisierter Funktionscodes inklusive detaillierter Beschreibungen erleichtert er die Implementierung, Wartung und Diagnose von vernetzten Maschinen. Seine einfache Struktur ohne Laufzeitlogik macht ihn zu einem robusten und klar verständlichen Bestandteil jedes 4diac-Projekts, das mit ISOBUS-Geräten kommuniziert.

IG3_Source_Addresses

• – ◦ • – ◦ • – ◦ •

Einleitung

Das globale Konstantenobjekt **IG3_Source_Addresses** definiert feste Quelladressen (Source Addresses) für Geräte des ISOBUS-Netzwerks (ISO 11783) speziell für die Industriegruppe 3. Diese Adressen werden zur eindeutigen Identifizierung von Steuergeräten, Sensoren und Anzeigeeinheiten innerhalb eines landwirtschaftlichen oder bautechnischen Fahrzeugs verwendet. Der Baustein stellt keinen aktiven Funktionsblock dar, sondern liefert einen Satz von Konstanten, die in anderen Applikationen als globale Variablen referenziert werden können.

Schnittstellenstruktur

Da es sich um ein reines Konstantenobjekt handelt, besitzt **IG3_Source_Addresses keine** prozessbezogenen Schnittstellen. Es werden weder Ereignisse verarbeitet noch Datenein- oder -ausgänge verwendet. Die folgenden Abschnitte sind daher nicht zutreffend und dienen nur der vollständigen Strukturbeschreibung.

Ereignis-Eingänge

Keine vorhanden.

Ereignis-Ausgänge

Keine vorhanden.

Daten-Eingänge

Keine vorhanden.

Daten-Ausgänge
Keine vorhanden.

Adapter
Keine vorhanden.

Funktionsweise
Die Konstanten definieren die im ISOBUS-Standard festgelegten Quelladressen für Geräte der Industriegruppe 3. Jede Konstante hat einen sprechenden Namen, der das zugeordnete Gerät beschreibt, und einen festen Wert vom Typ `BYTE`. Die Werte sind in folgende Bereiche aufgeteilt:

- **128–207**: Reserviert für dynamische Adresszuweisung (selbstkonfigurierbar)
- **208–223**: Reserviert für individuell vorab zugewiesene Adressen
- **224–247**: Spezifische Geräteadressen (z. B. Sensoren, Controller, Displays)

Die folgende Tabelle listet alle definierten Konstanten:

Konstante	Wert	Gerätebeschreibung
`SA_RESERVED_128`	128	Reserviert für dynamische Adressvergabe
`SA_RESERVED_208`	208	Reserviert für individuelle Adressvergabe
`SA_ROTATION_SENSOR`	224	Drehwinkelsensor
`SA_LIFT_ARM_CONTROLLER`	225	Hubarm-Steuerung für Baumaschinen
`SA_SLOPE_SENSOR`	226	Neigungssensor
`SA_MAIN_CONTROLLER_SKID_STEER_LOADER`	227	Hauptcontroller für Skid-Steer-Lader
`SA_LOADER_CONTROL`	228	Steuerung der Laderhydraulik
`SA_LASER_TRACER`	229	Lasertracker für Positionserfassung
`SA_LAND_LEVELING_SYSTEM_DISPLAY`	230	Anzeige für Landplaniersystem
`SA_SINGLE_LAND_LEVELING_SYSTEM_SUPERVISOR`	231	Supervisor für Einzel-Landplaniersystem
`SA_LAND_LEVELING_ELECTRIC_MAST`	232	Elektrischer Mast für Landplaniersystem
`SA_SINGLE_LAND_LEVELING_SYSTEM_OPERATOR_INTERFACE`	233	Bedienoberfläche des Landplaniersystems
`SA_LASER_RECEIVER`	234	Laserempfänger
`SA_SUPPLEMENTAL_SENSOR_PROCESSING_UNIT_1`	235	Zusätzliche Sensor-Verarbeitungseinheit 1
`SA_SUPPLEMENTAL_SENSOR_PROCESSING_UNIT_2`	236	Zusätzliche Sensor-Verarbeitungseinheit 2
`SA_SUPPLEMENTAL_SENSOR_PROCESSING_UNIT_3`	237	Zusätzliche Sensor-Verarbeitungseinheit 3
`SA_SUPPLEMENTAL_SENSOR_PROCESSING_UNIT_4`	238	Zusätzliche Sensor-Verarbeitungseinheit 4
`SA_SUPPLEMENTAL_SENSOR_PROCESSING_UNIT_5`	239	Zusätzliche Sensor-Verarbeitungseinheit 5
`SA_SUPPLEMENTAL_SENSOR_PROCESSING_UNIT_6`	240	Zusätzliche Sensor-Verarbeitungseinheit 6
`SA_ENGINE_MONITOR_1`	241	Motor-Monitor 1
`SA_ENGINE_MONITOR_2`	242	Motor-Monitor 2
`SA_ENGINE_MONITOR_3`	243	Motor-Monitor 3
`SA_ENGINE_MONITOR_4`	244	Motor-Monitor 4
`SA_ENGINE_MONITOR_5`	245	Motor-Monitor 5
`SA_ENGINE_MONITOR_6`	246	Motor-Monitor 6
`SA_ENGINE_MONITOR_7`	247	Motor-Monitor 7

Diese Konstanten können in 4diac-Applikationen direkt referenziert werden, um sicherzustellen, dass ISOBUS-Nachrichten an die korrekte Quelladresse gesendet werden.

Technische Besonderheiten

- Alle Konstanten sind als `BYTE` deklariert und besitzen feste Initialwerte.
- Die Werte sind gemäß ISO 11783 (ISOBUS) standardisiert und dürfen nicht verändert werden.
- Die Konstanten sind im Compiler-Paket `isobus::pgn::const` gekapselt, was eine saubere Strukturierung des Projekts unterstützt.
- Die Datei enthält keine Logik oder Zustandsautomaten – sie ist rein deklarativ.

Zustandsübersicht

Nicht zutreffend, da das Objekt keine aktiven Zustände besitzt.

Anwendungsszenarien

- **Entwicklung von ISOBUS-Steuergeräten**: Die Konstanten können verwendet werden, um die eigenen Quelladressen zu konfigurieren oder um Adressen anderer Geräte im Netzwerk zu referenzieren.
- **Kommunikation mit spezifischen Geräten**: Beispielsweise können Nachrichten an den `SA_ROTATION_SENSOR` (224) adressiert werden, um Drehwinkeldaten zu empfangen.
- **Integration in 4diac-Projekte**: Die Konstanten werden als globale Variablen in Funktionsblöcken verwendet, um den Programmcode lesbarer und wartbarer zu gestalten.

Vergleich mit ähnlichen Bausteinen

Es existieren analoge `GlobalConstants`-Objekte für andere Industriegruppen, z. B. für die Gruppen 1 und 2. Diese definieren jeweils die passenden Quelladressen für ihre spezifischen Geräte. Der vorliegende Baustein konzentriert sich ausschließlich auf die Industriegruppe 3, die vorwiegend in der Bau- und Landtechnik relevante Geräte (wie Lader, Planiersysteme, Sensoren) umfasst.

Fazit

Das `IG3_Source_Addresses`-Objekt stellt eine zentrale, standardkonforme Sammlung von Quelladressen für ISOBUS-Anwendungen bereit. Durch die Verwendung dieser Konstanten wird die Einhaltung des ISO-Standards erleichtert und die Softwareentwicklung für Steuergeräte in der Industriegruppe 3 vereinfacht. Auch wenn es sich nicht um einen klassischen Funktionsblock handelt, erfüllt es eine wichtige Rolle in der Projektstrukturierung und Adressverwaltung.

IG4_Device_Classes

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `IG4_Device_Classes` ist ein GlobalConstants-Baustein in 4diac. Er definiert Konstanten für Geräteklassen (Device Classes) von Fahrzeugsystemen gemäß ISOBUS / Industry Group 4. Diese Konstanten sind im Compiler-Paket `isobus::pgn::const` verfügbar und können in IEC 61499-Anwendungen als globale, unveränderliche Werte referenziert werden.

Schnittstellenstruktur

Der Baustein besitzt keine Ereignis- oder Datenschnittstellen, da es sich um eine globale Konstantensammlung handelt.

Ereignis-Eingänge

Nicht vorhanden.

Ereignis-Ausgänge

Nicht vorhanden.

Daten-Eingänge

Nicht vorhanden.

Daten-Ausgänge

Nicht vorhanden. Als globaler Konstantenbaustein stellt er jedoch folgende Werte im Projektkontext bereit:

Konstante	Typ	Wert	Beschreibung
DC_NON_SPECIFIC_SYSTEM	BYTE	0	Nicht spezifisches System
DC_SYSTEM_TOOLS	BYTE	10	Systemwerkzeuge
DC_SAFETY_SYSTEMS	BYTE	20	Sicherheitssysteme
DC_GATEWAY	BYTE	25	Gateway
DC_POWER_MGMT_LIGHTING	BYTE	30	Energiemanagement und Beleuchtungssysteme
DC_STEERING_SYSTEMS	BYTE	40	Lenksysteme
DC_PROPULSION_SYSTEMS	BYTE	50	Antriebssysteme
DC_NAVIGATION_SYSTEMS	BYTE	60	Navigationssysteme
DC_COMMUNICATIONS_SYSTEMS	BYTE	70	Kommunikationssysteme
DC_INSTRUMENTATION_GENERAL	BYTE	80	Instrumentierung / allgemeine Systeme
DC_ENVIRONMENTAL_HVAC	BYTE	90	Umgebungssysteme (HLK)
DC_DECK_CARGO_FISHING	BYTE	100	Deck-, Ladungs- und Fangausrüstung
DC_NOT_AVAILABLE	BYTE	127	Nicht verfügbar

Adapter

Nicht vorhanden.

Funktionsweise

Der Baustein dient als zentrale Definitionsquelle für Geräteklassen innerhalb der ISOBUS-Gruppe 4 (Fahrzeugsysteme). Er stellt symbolische Namen für numerische Werte bereit, die an anderer Stelle in der Anwendung als Konstanten referenziert werden können. Dadurch werden magische Zahlen vermieden und die Lesbarkeit sowie Wartbarkeit des IEC-61499-Codes verbessert.

Technische Besonderheiten

- Alle Konstanten sind vom Typ `BYTE` und besitzen feste, unveränderliche Werte.
- Die Konstanten sind im Paket `isobus::pgn::const` gekapselt.
- Der Baustein wird als `GlobalConstants`-Element deklariert und besitzt keine Laufzeitsemantik; er wird nur zur Compilezeit verwendet.
- Die Werte folgen den ISOBUS-Konventionen der Industriegruppe 4.

Zustandsübersicht

Nicht anwendbar. Der Baustein besitzt keine Zustände oder Zustandsübergänge, da er keine ausführbare Logik enthält.

Anwendungsszenarien

- **ISOBUS-Anwendungen**: Zur Kennzeichnung von Geräten und Systemen in Fahrzeugnetzwerken.
- **Systemdiagnose**: Zuordnung von Fehlern oder Ereignissen zu bestimmten Geräteklassen.
- **Gateway-Konfiguration**: Auswahl der passenden Geräteklasse bei der Kommunikation zwischen verschiedenen Fahrzeugteilen.
- **Normkonforme Entwicklung**: Verwendung der offiziellen ISOBUS-Geräteklassen-Werte ohne manuelle Definition.

Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu Funktionsbausteinen oder Adaptern besitzt `IG4_Device_Classes` keine Ein-/Ausgänge, keine Algorithmen und kein internes Zustandsverhalten. Es ist eine reine Datenquelle für Konstanten. Ähnliche GlobalConstants-Bausteine existieren für andere Industriegruppen (z. B. `IG3_Device_Classes` für landwirtschaftliche Systeme außerhalb von Fahrzeugen). Der wesentliche Unterschied liegt in den abgedeckten Wertebereichen und der semantischen Zuordnung der Geräteklassen.

Fazit

`IG4_Device_Classes` ist ein einfacher, aber wichtiger GlobalConstants-Baustein zur Vereinheitlichung der Geräteklassen-Definitionen in ISOBUS-Anwendungen. Durch die symbolische Darstellung der Werte wird die Codequalität verbessert und die Konformität mit der Norm sichergestellt.

IG4_NAME_Functions

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `IG4_NAME_Functions` stellt eine Sammlung globaler Konstanten für die **Industry Group 4** des ISOBUS-Standards (ISO 11783) bereit. Diese Konstanten definieren die **Funktionscodes** einzelner elektronischer Steuergeräte (ECUs) in einem ISOBUS-Netzwerk. Die Werte sind als `BYTE`-Konstanten ausgeführt und ermöglichen eine eindeutige Identifizierung der Funktion eines Geräts innerhalb der Industriegruppe 4 (Schiffs- und Bootsanwendungen). Der Baustein dient als zentrale Definitionsquelle für die Entwicklung von Anwendungen, die auf die ISOBUS-Kommunikation zugreifen.

Schnittstellenstruktur

Der Baustein besitzt **keine** Ereignis- oder Dateneingänge und auch keine klassischen Datenausgänge. Stattdessen stellt er eine Reihe von globalen Konstanten bereit, die direkt über ihren symbolischen Namen referenziert werden können. Diese Konstanten werden im Folgenden als **globale Konstanten** aufgeführt und können als Werte für die Konfiguration oder Kommunikation in anderen Funktionsbausteinen verwendet werden.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Die folgenden globalen Konstanten werden als Datenwerte bereitgestellt (Typ: `BYTE`). Sie sind inhaltlich nach den ISOBUS-Systemen gruppiert.

Konstantenname	Wert	Beschreibung
Nicht spezifisches System (System 0)		
`F_ALARM_SYSTEM_CONTROL_FOR_MARINE_ENGINES`	128	Alarmsteuerung für Schiffsmaschinen – ECU, die die Alarmfunktionen steuert.
`F_PROTECTION_SYSTEM_FOR_MARINE_ENGINES`	129	Schutzsystem für Schiffsmaschinen – erste ECU, die Schutzfunktionen steuert.
`F_DISPLAY_FOR_PROTECTION_SYSTEM_FOR_MARINE_ENGINES`	130	Anzeige für das Schutzsystem – ECU, die Informationen/Indikatoren anzeigt.
`F_NOT_AVAILABLE`	255	Nicht verfügbar – Platzhalter bis zur expliziten Zuweisung.
Systemwerkzeuge (System 10)		
`F_SYSTEM_TOOLS_NOT_AVAILABLE`	255	Nicht verfügbar.
Sicherheitssysteme (System 20)		
`F_SAFETY_SYSTEMS_NOT_AVAILABLE`	255	Nicht verfügbar.
Gateway (System 25)		
`F_GATEWAY`	130	Gateway – Funktion nicht näher spezifiziert.

Stromversorgungs- und Beleuchtungssysteme (System 30)		
F_POWER_MGMT_LIGHTING_SWITCH	130	Schalter für Stromversorgungs-/Beleuchtungssysteme.
F_POWER_MGMT_LIGHTING_LOAD	140	Last für Stromversorgungs-/Beleuchtungssysteme.
Lenksysteme (System 40)		
F_STEERING_SYSTEMS_FOLLOW_UP_CONTROLLER	130	Folgeregler (Follow-up Controller).
F_STEERING_SYSTEMS_MODE_CONTROLLER	140	Modusregler.
F_STEERING_SYSTEMS_AUTOMATIC_STEERING_CONTROLLER	150	Automatiksteuerung.
F_STEERING_SYSTEMS_HEADING_SENSORS	160	Kurssensorik (Heading Sensors).
Antriebssysteme (System 50)		
F_PROPULSION_SYSTEMS_ENGINEROOM_MONITORING	130	Maschinenraumüberwachung.
F_PROPULSION_SYSTEMS_ENGINE_INTERFACE	140	Motor-Schnittstelle.
F_PROPULSION_SYSTEMS_ENGINE_CONTROLLER	150	Motorsteuerung.
F_PROPULSION_SYSTEMS_ENGINE_GATEWAY	160	Motor-Gateway.
F_PROPULSION_SYSTEMS_CONTROL_HEAD	170	Steuerkopf (Control Head).
F_PROPULSION_SYSTEMS_ACTUATOR	180	Aktor.
F_PROPULSION_SYSTEMS_GAUGE_INTERFACE	190	Anzeige-Schnittstelle (Gauge Interface).
F_PROPULSION_SYSTEMS_GAUGE_LARGE	200	Große Anzeige.
F_PROPULSION_SYSTEMS_GAUGE_SMALL	210	Kleine Anzeige.
F_PROPULSION_SYSTEMS_PROPULSION_SENSORS_GATEWAY	220	Antriebssensoren & Gateway.
Navigationssysteme (System 60)		
F_NAVIGATION_SYSTEMS_SOUNDER_DEPTH	130	Echolot (Tiefensensor).
F_NAVIGATION_SYSTEMS	140	Allgemeine Navigationsfunktion.
F_NAVIGATION_SYSTEMS_GLOBAL_NAVIGATION_SATELLITE_SYSTEMS_GNSS	145	Globales Navigationssatellitensystem (GNSS).
F_NAVIGATION_SYSTEMS_LORAN_C	150	LORAN-C-Navigation.
F_NAVIGATION_SYSTEMS_SPEED_SENSORS	155	Geschwindigkeitssensoren.
F_NAVIGATION_SYSTEMS_TURN_RATE_INDICATOR	160	Drehratenanzeige.
F_NAVIGATION_SYSTEMS_INTEGRATED_NAVIGATION	170	Integrierte Navigation.
F_NAVIGATION_SYSTEMS_RADAR_AND_OR_RADAR_PLOTTING	200	Radar und/oder Radar-Plotting.
F_NAVIGATION_SYSTEMS_ELECTRONIC_CHART_DISPLAY_INFORMATION_SYSTEM_ECDIS	205	Elektronisches Seekarten-Anzeige- und Informationssystem (ECDIS).
F_NAVIGATION_SYSTEMS_ELECTRONIC_CHART_SYSTEM_ECS	210	Elektronisches Kartensystem (ECS).
F_NAVIGATION_SYSTEMS_DIRECTION_FINDER	220	Funkpeiler (Direction Finder).
Kommunikationssysteme (System 70)		
F_COMMUNICATIONS_SYSTEMS_EMERGENCY_POSITION_INDICATING_BEACON_EPIRB	130	Notfallbake zur Positionsanzeige (EPIRB).

F_COMMUNICATIONS_SYSTEMS_AUTOMATIC_IDENTIFICATION_SYSTEM	140	Automatisches Identifikationssystem (AIS).
F_COMMUNICATIONS_SYSTEMS_DIGITAL_SELECTIVE_CALLING_DSC	150	Digitaler Selektivruf (DSC).
F_COMMUNICATIONS_SYSTEMS_DATA_RECEIVER	160	Datenempfänger.
F_COMMUNICATIONS_SYSTEMS_SATELLITE	170	Satellitenkommunikation.
F_COMMUNICATIONS_SYSTEMS_RADIO_TELEPHONE_MF_HF	180	Funktelefon (Mittel-/Kurzwelle).
F_COMMUNICATIONS_SYSTEMS_RADIO_TELEPHONE_VHF	190	Funktelefon (UKW).
Instrumentierung/allgemeine Systeme (System 80)		
F_INSTRUMENTATION_GENERAL_TIME_DATE_SYSTEMS	130	Zeit-/Datumsysteme.
F_INSTRUMENTATION_GENERAL_VOYAGE_DATA_RECORDER	140	Reisedatenschreiber (Voyage Data Recorder).
F_INSTRUMENTATION_GENERAL_INTEGRATED_INSTRUMENTATION	150	Integrierte Instrumentierung.
F_INSTRUMENTATION_GENERAL_GENERAL_PURPOSE_DISPLAYS	160	Allzweckanzeigen.
F_INSTRUMENTATION_GENERAL_GENERAL_SENSOR_BOX	170	Allgemeine Sensorbox.
F_INSTRUMENTATION_GENERAL_WEATHER_INSTRUMENTS	180	Wetterinstrumente.
F_INSTRUMENTATION_GENERAL_TRANSDUCER_GENERAL	190	Allgemeiner Messumformer.
F_INSTRUMENTATION_GENERAL_NMEA_0183_CONVERTER	200	NMEA-0183-Konverter.
Umweltsysteme (HVAC) (System 90)		
F_ENVIRONMENTAL_HVAC_NOT_AVAILABLE	255	Nicht verfügbar.
Deck-, Ladungs- und Fischereiausrüstung (System 100)		
F_DECK_CARGO_FISHING_NOT_AVAILABLE	255	Nicht verfügbar.

Funktionsweise

Die Konstanten in `IG4_NAME_Functions` dienen als symbolische Namen für ISOBUS-Funktionscodes. Jede Konstante hat einen festen `BYTE`-Wert, der in der ISOBUS-Kommunikation (z. B. in der Parameter Group Numbers (PGN) für die Adressierung und Funktionsidentifikation) verwendet wird. Durch die Verwendung dieser Konstanten wird der Code lesbarer und wartbarer, da anstelle von magischen Zahlen aussagekräftige Namen verwendet werden. Die Werte sind nach dem ISOBUS-Standard für die Industriegruppe 4 festgelegt und können unverändert in eigenen Anwendungen referenziert werden.

Die Konstanten sind in logische Gruppen entsprechend der ISOBUS-Systeme unterteilt – von nicht spezifischen Systemen über Antriebs-, Navigations- und Kommunikationssysteme bis hin zu Instrumentierung und Umgebungssystemen. Jede Konstante entspricht einer bestimmten Funktion, die ein Gerät (ECU) in einem Netzwerk ausführt.

Technische Besonderheiten

- **Typ:** Alle Konstanten sind vom Typ `BYTE` (8-Bit-Wert) und können Werte von 0–255 annehmen.
- **Wertebereich:** Die verwendeten Werte liegen zwischen 128 und 220 sowie 255 für „Nicht verfügbar“. Diese Werte folgen den ISOBUS-Spezifikationen für die Industriegruppe 4.
- **Namensschema:** Die Konstanten beginnen mit `F_` gefolgt vom Systemnamen (z. B. `NAVIGATION_SYSTEMS`) und einer genaueren Bezeichnung. Das Schema erleichtert die Zuordnung.
- **Globaler Zugriff:** Die Konstanten sind global definiert und können in jedem FB oder jeder Applikation innerhalb des 4diac-Projekts verwendet werden, ohne dass eine Instanzierung erforderlich ist.
- **Keine Laufzeitänderung:** Die Werte sind konstant und können zur Laufzeit nicht verändert werden.

Zustandsübersicht

Nicht zutreffend. Da es sich um einen reinen Konstanten-Baustein handelt, gibt es keinen internen Zustand oder eine Zustandsmaschine.

Anwendungsszenarien

- **ISOBUS-Kommunikation:** Verwendung der Konstanten zur Identifizierung der Funktion einer ECU in einem ISOBUS-Netzwerk (z. B. in PGN-Nachrichten für die Adressierung).
- **Konfiguration von Geräten:** In der Entwicklung von Anwendungen für Schiffs- oder Bootsautomation können diese Konstanten bei der Konfiguration von Steuergeräten verwendet werden.
- **Erweiterung bestehender Systeme:** Bei der Integration neuer Geräte in ein bestehendes Netzwerk können die passenden Funktionscodes über die Konstanten zugewiesen werden.
- **Lesbare Codebasis:** Ersatz von numerischen Magic Numbers durch aussagekräftige symbolische Namen verbessert die Wartbarkeit und reduziert Fehler.

Vergleich mit ähnlichen Bausteinen

Ähnliche Global-Konstanten-Bausteine existieren für andere Industriegruppen (z. B. Industry Group 1, 2, 3) im ISOBUS-Standard. Diese definieren jeweils eigene Funktionscodes, die auf die spezifischen Anforderungen der jeweiligen Gruppe (z. B. Landwirtschaft, Bau, Forst) zugeschnitten sind. `IG4_NAME_Functions` deckt ausschließlich die Anforderungen der Industriegruppe 4 (Marine) ab. Im Vergleich zu einem Funktionsbaustein, der Logik oder Verhalten implementiert, stellt dieser Baustein lediglich statische Daten bereit und besitzt keine Schnittstellen für Ereignisse oder Flusssteuerung. Er ist ein reiner Datencontainer, der als Unterstützung für die Entwicklung von FB-Kompositionen dient.

Fazit

`IG4_NAME_Functions` ist ein essenzieller Bestandteil für die Entwicklung ISOBUS-konformer Anwendungen im Marinebereich. Er bündelt alle relevanten Funktionscodes für die Industriegruppe 4 in einer zentralen, leicht zugänglichen und gut dokumentierten Form. Durch die Verwendung dieser Konstanten wird die Implementierung von ISOBUS-Funktionalitäten in der 4diac-IDE deutlich vereinfacht und standardisiert. Die klare Struktur und die umfangreiche Kommentierung machen den Baustein zu einer wertvollen Ressource für Entwickler, die marine Steuerungssysteme realisieren.

IG4_Source_Addresses

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **IG4_Source_Addresses** ist ein globaler Konstanten-Container für ISOBUS-spezifische Quelladressen (Source Addresses) der Industriegruppe 4 (Industry Group 4). Er stellt eine Sammlung von Konstanten mit festen BYTE-Werten bereit, die in der Automatisierungstechnik, insbesondere im Bereich der mobilen Arbeitsmaschinen und Marine-Antriebssysteme, zur eindeutigen Identifikation von ECUs (Electronic Control Units) verwendet werden. Die Werte sind nach dem Standard SAE J1939 bzw. ISO 11783 (ISOBUS) definiert und ermöglichen eine standardisierte Adressierung in CAN-basierten Netzwerken.

Schnittstellenstruktur

Der Baustein besitzt keine Ereignis- oder Datenschnittstellen im herkömmlichen Sinne, da er ausschließlich globale Konstanten bereitstellt. Die Konstanten sind als Variablen mit festen Werten (InitialValue) definiert und können in anderen Bausteinen direkt referenziert werden.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine – die Werte werden über die Konstantenbereitstellung der globalen Variablen zugänglich gemacht.

Adapter

Keine.

Funktionsweise

Der Baustein definiert eine Reihe von vordefinierten Konstanten, die jeweils eine spezifische Quelladresse (0–255) für verschiedene ECUs repräsentieren. Diese Adressen sind Teil des ISOBUS-Adressierungskonzepts und dienen der eindeutigen Identifikation von Steuergeräten in CAN-Netzwerken. Die Werte sind in der XML-Datei als `VarDeclaration` mit `Type="BYTE"` und einem `InitialValue` hinterlegt.

Die folgende Tabelle fasst alle definierten Konstanten zusammen:

Name	Wert (BYTE)	Beschreibung
`SA_RESERVED_128`	128	Reserviert für dynamische Adresszuweisung (selbstkonfigurierend)
`SA_RESERVED_208`	208	Reserviert für individuelle fest vergebene Adressen
`SA_PROPULSION_SENSOR_HUB_GATEWAY_1`	228	Propulsions-Sensor-Hub & Gateway #1
`SA_PROPULSION_SENSOR_HUB_GATEWAY_2`	229	Propulsions-Sensor-Hub & Gateway #2
`SA_PROPULSION_SENSOR_HUB_GATEWAY_3`	230	Propulsions-Sensor-Hub & Gateway #3
`SA_PROPULSION_SENSOR_HUB_GATEWAY_4`	231	Propulsions-Sensor-Hub & Gateway #4
`SA_TRANSMISSION_3`	232	Getriebe-ECU für das dritte Getriebe
`SA_TRANSMISSION_4`	233	Getriebe-ECU für das vierte Getriebe
`SA_TRANSMISSION_5`	234	Getriebe-ECU für das fünfte Getriebe
`SA_TRANSMISSION_6`	235	Getriebe-ECU für das sechste Getriebe
`SA_DISPLAY_1_FOR_PROTECTION_SYSTEM_FOR_MARINE_ENGINES`	236	Display #1 für Schutzsystem von Schiffsdieselmotoren
`SA_PROTECTION_SYSTEM_FOR_MARINE_ENGINES`	237	Schutzsystem-Steuergerät für Schiffsdieselmotoren
`SA_ALARM_SYSTEM_CONTROL_1_FOR_MARINE_ENGINES`	238	Alarmsystem-Steuerung #1 für Schiffsdieselmotoren
`SA_ENGINE_3`	239	Motor-ECU für den dritten Motor
`SA_ENGINE_4`	240	Motor-ECU für den vierten Motor
`SA_ENGINE_5`	241	Motor-ECU für den fünften Motor
`SA_MARINE_DISPLAY_1`	242	Marine-Display #1
`SA_MARINE_DISPLAY_2`	243	Marine-Display #2
`SA_MARINE_DISPLAY_3`	244	Marine-Display #3
`SA_MARINE_DISPLAY_4`	245	Marine-Display #4
`SA_MARINE_DISPLAY_5`	246	Marine-Display #5
`SA_MARINE_DISPLAY_6`	247	Marine-Display #6

Diese Konstanten können in anderen Funktionsbausteinen als Quelladressen für die Kommunikation verwendet werden, um eine konsistente und herstellerunabhängige Adressierung sicherzustellen.

Technische Besonderheiten

- Der Baustein ist als **GlobalConstants**-Element definiert und wird in der 4diac-IDE als globaler Konstanten-Container behandelt.
- Alle Konstanten sind vom Typ `BYTE` (8-Bit-Wert, Bereich 0–255).
- Die Werte entsprechen den im ISOBUS-Standard (ISO 11783) festgelegten Adressen für die Industriegruppe 4 (Marine- und Propulsionsanwendungen).
- Die Konstanten können in anderen Bausteinen direkt über ihren Namen referenziert werden, ohne dass sie als Eingänge übergeben werden müssen.

- Die XML-Datei enthält zusätzlich eine `CompilerInfo`-Angabe (`isobus::pgn::const`), die auf die Paketstruktur für die Nativerzeugung hinweist.

Zustandsübersicht

Da es sich um einen Konstanten-Container handelt, existieren keine internen Zustände oder Übergänge. Der Baustein ist statisch und wird zur Laufzeit nicht verändert.

Anwendungsszenarien

- **ISOBUS-Netzwerkkonfiguration**: Verwendung der Quelladressen zur einheitlichen Konfiguration von Steuergeräten in landwirtschaftlichen oder mobilen Maschinen.
- **Marine-Antriebssysteme**: Adressierung von Motoren, Getrieben, Displays und Schutzsystemen in Schiffsanwendungen.
- **Entwicklung von ISOBUS-konformen Anwendungen**: Bereitstellung vordefinierter Adressen für die Kommunikation zwischen ECU und Bediengeräten.
- **Prototypenerstellung**: Schnelle Verwendung standardisierter Adressen ohne manuelle Definition.

Vergleich mit ähnlichen Bausteinen

In der 4diac-Bibliothek existieren weitere globale Konstanten-Container für andere Industriegruppen (z.B. IG0 bis IG3). Diese unterscheiden sich in den spezifischen Adressbereichen und den zugeordneten Gerätetypen. Der vorliegende Baustein ist auf die Besonderheiten der Industriegruppe 4 (Marine und Propulsion) zugeschnitten und ergänzt die anderen Gruppen.

Fazit

IG4_Source_Addresses ist ein einfacher, aber essentieller Baustein, der die standardisierten ISOBUS-Quelladressen für Marine- und Propulsionsanwendungen in einer zentralen, wiederverwendbaren Form bereitstellt. Durch die Verwendung globaler Konstanten wird die Konsistenz in der Adressierung erhöht und die Entwicklung von Kommunikationsanwendungen vereinfacht. Der Baustein eignet sich hervorragend für Projekte, die eine konforme Adressierung gemäß ISO 11783 erfordern.

IG5_Device_Classes

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `IG5_Device_Classes` ist eine Sammlung globaler Konstanten, die spezifische Geräteklassen (Device Classes) für die Industriegruppe 5 (Industry Group 5) im ISOBUS-Standard (ISO 11783) definiert. Diese Gruppe umfasst Fahrzeugsysteme. Die Konstanten dienen als Referenzwerte für die Klassifizierung von Geräten in landwirtschaftlichen und mobilen Maschinen, insbesondere im Bereich von Stromerzeugern (Gen-Sets) und für den Fall, dass keine Geräteklasse verfügbar ist.

Schnittstellenstruktur

Da es sich um eine globale Konstantenliste handelt, besitzt der Baustein keine typischen Schnittstellen eines Funktionsblocks. Es werden keine Ereignisse oder Daten ausgetauscht, sondern die Konstanten werden direkt als symbolische Namen im Projekt verwendet.

Ereignis-Eingänge

Nicht zutreffend – es sind keine Ereignis-Eingänge definiert.

Ereignis-Ausgänge

Nicht zutreffend – es sind keine Ereignis-Ausgänge definiert.

Daten-Eingänge

Nicht zutreffend – es sind keine Daten-Eingänge definiert.

Daten-Ausgänge

Nicht zutreffend – es sind keine Daten-Ausgänge definiert. Stattdessen werden die Werte als globale Konstanten bereitgestellt, die an beliebigen Stellen im Projekt referenziert werden können.

Adapter

Nicht zutreffend – es sind keine Adapter vorhanden.

Funktionsweise

Der Baustein stellt zwei Konstanten vom Typ `BYTE` bereit:

- `DC_INDUSTRIAL_GEN_SET` mit dem Wert `0` – kennzeichnet ein stationäres Industrie-Prozesskontrollgerät (Gen-Set).
- `DC_NOT_AVAILABLE` mit dem Wert `127` – steht für „nicht verfügbar“ und wird verwendet, wenn keine Geräteklasse angegeben werden kann.

Diese Konstanten werden typischerweise in ISOBUS-Anwendungen verwendet, um in Nachrichten oder Parametern die Geräteklasse zu referenzieren. Durch die Verwendung symbolischer Namen wird die Lesbarkeit des Codes verbessert und Fehler durch hartkodierte Zahlenwerte vermieden.

Technische Besonderheiten

- Die Konstanten sind im Standard `61499-1` eingebettet und folgen der ISOBUS-Definition für Device Classes.
- Die Werte sind als `BYTE` deklariert und haben die Initialwerte `0` und `127`.
- Die Konstanten sind global gültig und können innerhalb des gesamten 4diac-Projekts verwendet werden.
- Die Definition basiert auf der ISOBUS-Spezifikation für Industry Group 5, die Fahrzeugsysteme betrifft.

Zustandsübersicht

Da es sich um eine reine Konstantendefinition handelt, existiert keine Zustandsmaschine oder ein dynamisches Verhalten. Der Baustein ändert seinen Zustand nicht und benötigt keine Zustandsverwaltung.

Anwendungsszenarien

- **Landwirtschaftliche Maschinen:** Verwendung zur Identifizierung eines angeschlossenen Geräts als stationäres Stromaggregat.
- **Fahrzeugkommunikation:** In ISOBUS-Nachrichten kann die Geräteklasse zur Laufzeit abgefragt und mit diesen Konstanten verglichen werden.
- **Fehlerbehandlung:** Wenn die Geräteklasse unbekannt ist, kann `DC_NOT_AVAILABLE` verwendet werden, um einen definierten Fallback zu bieten.
- **Konfiguration:** Die Konstanten können bei der Parametrierung von Steuergeräten eingesetzt werden, um feste Werte zu vermeiden.

Vergleich mit ähnlichen Bausteinen

Im 4diac-Umfeld gibt es weitere GlobalConstants-Bausteine für andere Industriegruppen (z.B. IG0 bis IG4). Diese definieren jeweils spezifische Geräteklassen für ihre Gruppe. Der vorliegende Baustein ist auf Industriegruppe 5 fokussiert und bietet dort die relevanten Werte. Im Gegensatz zu Funktionsblöcken mit Ein-/Ausgängen handelt es sich hier um eine passive Datenquelle, die keine Logik ausführt.

Fazit

`IG5_Device_Classes` ist ein einfacher, aber wichtiger Baustein für ISOBUS-Anwendungen, der die Verwendung standardisierter Geräteklassen erleichtert. Durch die klaren Konstantennamen wird die Wartbarkeit verbessert und die Einhaltung des ISOBUS-Standards unterstützt. Für Entwickler, die Fahrzeugsysteme der Industriegruppe 5 anbinden, stellt dieser Baustein eine nützliche Basis dar.

IG5_NAME_Functions

• – ◦ • – ◦ • – ◦ •

Einleitung

`IG5_NAME_Functions` ist eine Sammlung globaler Konstanten für die ISO 11783 (ISOBUS) Kommunikation, speziell für die **Industry Group 5** (Industrieprozesssteuerung, insbesondere stationäre Generatoren (Gen-Sets) und Bohrlochstimulationspumpen). Die Konstanten definieren Funktionscodes, die in bestimmten PGNs (Parameter Group Numbers) verwendet werden, um den Zweck eines angeschlossenen Geräts innerhalb des ISOBUS-Netzwerks zu identifizieren. Die Definitionen folgen dem Standard IEC 61499-1, jedoch handelt es sich nicht um einen Funktionsblock mit ausführbarer Logik, sondern um eine verteilbare Konstantenressource für die 4diac-IDE.

Schnittstellenstruktur

Da es sich um eine globale Konstantensammlung handelt, besitzt dieser Baustein keine klassischen Ein-/Ausgangs-Schnittstellen.

Ereignis-Eingänge

Nicht vorhanden.

Ereignis-Ausgänge

Nicht vorhanden.

Daten-Eingänge

Nicht vorhanden.

Daten-Ausgänge

Die definierten Konstanten können als Datenwerte angesehen werden, die in anderen FB-Bausteinen referenziert werden. Die folgende Tabelle listet alle Konstante auf:

Konstante	Typ	Wert	Beschreibung
F_INDUSTRIAL_GEN_SET_SUPPLEMENTAL_ENGINE_CONTROL_SENSING	BYTE	128	Zusätzliche Motorsteuerungssensorik (Gen-Sets)
F_INDUSTRIAL_GEN_SET_GENERATOR_SET_CONTROLLER	BYTE	129	Generator-Set-Controller (Generatoreinheit)
F_INDUSTRIAL_GEN_SET_GENERATOR_VOLTAGE_REGULATOR	BYTE	130	Generator-Spannungsregler
F_INDUSTRIAL_GEN_SET_CHOKE_ACTUATOR	BYTE	131	Drosselklappen-Stellantrieb für Gas-Motoren
F_WELL_STIMULATION_PUMP	BYTE	132	Pumpe zur Bohrlochstimulation (Öl-/Gasförderung)
F_INDUSTRIAL_GEN_SET_NOT_AVAILABLE	BYTE	255	Nicht verfügbar (vorläufige Zuordnung)
F_NOT_AVAILABLE	BYTE	255	Allgemein nicht verfügbar

Adapter

Nicht vorhanden.

Funktionsweise

Die globalen Konstanten werden in einem ISOBUS-Netzwerk verwendet, um die Funktion eines angeschlossenen Geräts (z. B. eines Generators oder einer Pumpe) zu kennzeichnen. Die Werte entsprechen den festgelegten **Function Codes** der ISOBUS-Telegramme. Jede Konstante liefert einen unveränderlichen Byte-Wert, der in anderen Funktionsblöcken direkt als Vergleichs- oder Zuordnungswert eingesetzt werden kann. Die Liste ist bewusst statisch gehalten, um eine klare und fehlerfreie Zuordnung im gesamten Steuerungsprojekt zu gewährleisten.

Technische Besonderheiten

- Die Konstanten sind als `BYTE` (8-Bit) definiert und decken den Bereich 128–132 sowie 255 ab.
- Es existieren zwei unterschiedliche Konstanten mit demselben Wert `255` – `F_INDUSTRIAL_GEN_SET_NOT_AVAILABLE` (für Gen-Sets) und `F_NOT_AVAILABLE` (allgemein). Dies erlaubt eine differenzierte Verwendung in verschiedenen Kontexten.
- Die Konstanten sind als **globale Konstanten** in der 4diac-IDE verfügbar und können projektweit referenziert werden, ohne sie erneut deklarieren zu müssen.
- Die Definition ist als `GlobalConstants`-Ressource angelegt und besitzt eine Versionierung (`Version 1.0`) sowie eine Zuordnung zum ISOBUS-PGN-Paket `isobus::pgn::const`.

Zustandsübersicht

Da keine ausführbare Logik oder Zustandsautomaten vorhanden sind, entfällt eine Zustandsübersicht.

Anwendungsszenarien

- **Steuerung von Generatorsätzen:** Verwendung der Konstanten, um die Funktion eines angeschlossenen Geräts (z. B. Motorsteuerung, Spannungsregler) im ISOBUS-Netzwerk zu identifizieren.
- **Industrieprozess-Steuerung:** Einsatz in stationären Anlagen, um verschiedene Komponenten (z. B. Drosselklappen-Aktuatoren) eindeutig zu klassifizieren.
- **Öl- und Gasindustrie:** Die Konstante `F_WELL_STIMULATION_PUMP` ermöglicht die Zuordnung von Steuerungsdaten für Pumpen, die in der Bohrlochstimulation eingesetzt werden.

Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu typischen Funktionsblöcken besitzt `IG5_NAME_Functions` keine Schnittstellen oder Verarbeitungslogik. Es handelt sich um eine **reine Datenressource**, vergleichbar mit einem Header-File in klassischen Programmierumgebungen. Andere GS-Konstanten (z. B. für andere Industry Groups) folgen demselben Muster, unterscheiden sich jedoch in den Wertebereichen oder Namenskonventionen.

Fazit

`IG5_NAME_Functions` stellt eine standardisierte und versionierte Sammlung von Funktionscodes für die ISOBUS-Industriegruppe 5 bereit. Durch die Verwendung globaler Konstanten wird eine hohe Wiederverwendbarkeit und Klarheit in 4diac-Projekten erreicht, insbesondere für Anwendungen im Bereich stationärer Generatoren und Bohrlochstimulation. Die klare Struktur erleichtert die Wartung und Erweiterung der Kommunikationsprotokolle.

IG5_Source_Addresses

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **IG5_Source_Addresses** ist ein globaler Konstantenblock (GlobalConstants) im Rahmen der 4diac-IDE und der IEC 61499-Norm. Er definiert eine Sammlung von Quelladressen (Source Addresses) für das ISOBUS-Protokoll (ISO 11783), die speziell für die *Industry Group 5* – also für Anwendungen im Bereich Fahrzeug- und Generatorsteuerung – reserviert sind. Diese Konstanten ermöglichen eine einheitliche und wartbare Adressierung von Steuergeräten innerhalb eines ISOBUS-Netzwerks.

Schnittstellenstruktur

Da es sich um einen GlobalConstants-Baustein handelt, besitzt er weder Ereignis- noch Datenschnittstellen. Stattdessen stellt er eine Reihe von benannten Konstanten bereit, die in anderen Funktionsbausteinen über ihren globalen Namen referenziert werden können.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine (nur Konstanten, die als Global Constants verfügbar sind).

Adapter

Keine.

Funktionsweise

Der Baustein definiert die unten aufgeführten Konstanten, jeweils vom Typ `BYTE`. Diese Werte sind fest vorgegeben und können zur Laufzeit nicht verändert werden. Durch die Verwendung dieser benannten Konstanten können Entwickler auf standardisierte Quelladressen in ISOBUS-Kommunikation verweisen, ohne sich die numerischen Werte merken zu müssen. Die Zuordnung entspricht den Spezifikationen der SAE J1939/ISOBUS.

Die folgenden Konstanten sind enthalten:

Konstante	Wert	Beschreibung
SA_RESERVED_128	128	Reservierter Adressbereich (128–207) für dynamische Adressvergabe
SA_RESERVED_208	208	Reservierter Adressbereich (208–229) für individuell vorbelegte Adressen
SA_GENERATOR_VOLTAGE_REGULATOR	230	Generator-Spannungsregler – steuert die Generator-Ausgangsspannung
SA_ENGINE_3	231	Motor Nr. 3 – ECU des dritten Motors in einem System
SA_ENGINE_4	232	Motor Nr. 4 – ECU des vierten Motors
SA_ENGINE_5	233	Motor Nr. 5 – ECU des fünften Motors
SA_GENERATOR_SET_CONTROLLER	234	Generator-Set-Controller – Datenerfassung und Steuerung einer Generatoranlage
SA_SUPPLEMENTAL_SENSOR_PROCESSING_UNIT_1	235	Zusätzliche Sensorverarbeitungseinheit Nr. 1
SA_SUPPLEMENTAL_SENSOR_PROCESSING_UNIT_2	236	Zusätzliche Sensorverarbeitungseinheit Nr. 2
SA_SUPPLEMENTAL_SENSOR_PROCESSING_UNIT_3	237	Zusätzliche Sensorverarbeitungseinheit Nr. 3
SA_SUPPLEMENTAL_SENSOR_PROCESSING_UNIT_4	238	Zusätzliche Sensorverarbeitungseinheit Nr. 4
SA_SUPPLEMENTAL_SENSOR_PROCESSING_UNIT_5	239	Zusätzliche Sensorverarbeitungseinheit Nr. 5
SA_SUPPLEMENTAL_SENSOR_PROCESSING_UNIT_6	240	Zusätzliche Sensorverarbeitungseinheit Nr. 6
SA_ENGINE_MONITOR_1	241	Motorüberwachung Nr. 1
SA_ENGINE_MONITOR_2	242	Motorüberwachung Nr. 2
SA_ENGINE_MONITOR_3	243	Motorüberwachung Nr. 3
SA_ENGINE_MONITOR_4	244	Motorüberwachung Nr. 4
SA_ENGINE_MONITOR_5	245	Motorüberwachung Nr. 5
SA_ENGINE_MONITOR_6	246	Motorüberwachung Nr. 6
SA_ENGINE_MONITOR_7	247	Motorüberwachung Nr. 7

Technische Besonderheiten

- Die Werte entsprechen den in der ISO 11783 (ISOBUS) und SAE J1939 festgelegten Adressbereichen für die **Industry Group 5**.
- Einige Adressen (128–229) sind reserviert und werden für spezielle Zwecke (dynamische oder individuelle Vergabe) verwendet. Die übrigen sind konkreten Funktionseinheiten zugeordnet.
- Die Konstanten sind als **globale Konstanten** deklariert, d.h. sie sind in allen Bausteinen eines 4diac-Projekts sichtbar und nutzbar.
- Der Baustein ist Teil des Pakets `isobus::pgn::const` und dient als referenzierbare Definition für die PGN-basierten Kommunikationsdienste.

Zustandsübersicht

Nicht anwendbar – dieser Baustein besitzt keinen Zustandsautomaten und keine internen Zustandslogik. Er dient ausschließlich als Konstantensammlung.

Anwendungsszenarien

- **ISOBUS-Netzwerkkonfiguration**: Beim Aufbau eines ISOBUS-Netzes für landwirtschaftliche Fahrzeuge oder Generatorsteuerungen können die definierten Quelladressen direkt als globale Konstanten verwendet werden, um die Kommunikationsparameter einheitlich festzulegen.

- **Entwicklung von Steuerungsfunktionen**: Funktionsbausteine, die mit ISOBUS-Geräten kommunizieren (z.B. Motorsteuerung, Generatorsteuerung), können diese Konstanten nutzen, um die Zieladressen zu spezifizieren, ohne hartkodierte Zahlen verwenden zu müssen.
- **Systemerweiterungen**: Bei Hinzufügen weiterer Geräte können die entsprechenden Konstanten in anderen Bausteinen referenziert werden, um eine klare und wartbare Adressverwaltung zu gewährleisten.

Vergleich mit ähnlichen Bausteinen

In der 4diac-Bibliothek für ISOBUS existieren möglicherweise weitere GlobalConstants-Bausteine für andere Industry Groups (z.B. IG1, IG2, ...). Diese folgen dem gleichen Muster und definieren die entsprechenden Quelladressen für ihre jeweilige Gruppe. Der vorliegende Baustein ist speziell auf die Anforderungen der **Industry Group 5** zugeschnitten. Andere Bausteine könnten z.B. Adressen für landwirtschaftliche Geräte (IG1) oder Baumaschinen (IG2) enthalten.

Fazit

Der **IG5_Source_Addresses**-Baustein stellt eine klar definierte und standardkonforme Sammlung von Quelladressen für ISOBUS-Anwendungen der Industry Group 5 bereit. Durch die Verwendung als globale Konstanten wird die Lesbarkeit und Wartbarkeit von ISOBUS-Kommunikationsfunktionen erheblich verbessert. Er ist ein essentielles Hilfsmittel für Entwickler, die auf die Adressierung von Generator-, Motor- und Überwachungseinheiten in einem ISOBUS-Netzwerk zugreifen müssen.

Industry_Groups

• – ◦ • – ◦ • – ◦ •

Einleitung

Dieses Dokument beschreibt die globalen Konstanten für ISOBUS Industry Groups. Die Konstanten definieren die verschiedenen Industriegruppen (Industry Groups) im ISOBUS-Protokoll zur Klassifizierung von Geräten und Anwendungen.

Schnittstellenstruktur

Da es sich um eine Sammlung globaler Konstanten handelt, gibt es keine Ereignis- oder Datenein-/ausgänge im Sinne eines Funktionsblocks.

Ereignis-Eingänge

Nicht anwendbar.

Ereignis-Ausgänge

Nicht anwendbar.

Daten-Eingänge

Nicht anwendbar.

Daten-Ausgänge

Nicht anwendbar.

Adapter

Nicht anwendbar.

Funktionsweise

Die Konstanten sind als Byte-Werte definiert und dienen zur Kennzeichnung der Industriegruppe in der ISOBUS-Kommunikation. Sie können in Anwendungen verwendet werden, um Geräteklassen zu identifizieren oder zu filtern.

Technische Besonderheiten

- Alle Konstanten sind vom Typ `BYTE`.
- Die Werte reichen von 0 bis 7, wobei `0` für global und `1-5` für spezifische Gruppen stehen, und `6-7` für zukünftige Verwendung durch SAE reserviert sind.
- Die Konstanten sind als `CONSTANT` deklariert, d.h. sie sind unveränderlich und global verfügbar.

Zustandsübersicht

Nicht zutreffend, da es sich um Konstanten handelt und keine Zustandsautomaten vorhanden sind.

Anwendungsszenarien

- In ISOBUS-Netzwerken zur Identifikation der Geräteklasse (z.B. landwirtschaftliche Maschinen, Bauausrüstung, Marineanwendungen).
- Bei der Entwicklung von Steuerungsanwendungen, die auf die Industriegruppe reagieren müssen.
- In der Diagnose und Überwachung von ISOBUS-Kommunikation.

Vergleich mit ähnlichen Bausteinen

Es gibt keine direkten ähnlichen Funktionsblöcke, jedoch bieten einige Bibliotheken vergleichbare Konstantendefinitionen für andere Protokolle (z.B. CAN-IDs). Diese Konstanten sind spezifisch für ISOBUS.

Fazit

Die GlobalConstants `Industry_Groups` stellen eine klare und standardisierte Möglichkeit dar, Industriegruppen im ISOBUS-Kontext zu referenzieren. Sie sind einfach zu verwenden und tragen zur Interoperabilität bei.

Manufacturer_IDs

• – ◦ • – ◦ • – ◦ •

Einleitung

Dieser GlobalConstants-Baustein definiert eine umfangreiche Liste von Hersteller-Identifikationsnummern (Manufacturer IDs) für den ISOBUS-Standard. Jede Konstante repräsentiert einen eindeutigen numerischen Wert, der einem bestimmten Hersteller zugeordnet ist. Diese IDs werden verwendet, um Geräte in landwirtschaftlichen und nutzfahrzeugbezogenen Netzwerken eindeutig zu identifizieren.

Schnittstellenstruktur

Da es sich um einen GlobalConstants-Baustein handelt, besitzt er keine klassischen Ein- oder Ausgänge. Alle Werte sind als konstante Daten verfügbar.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Die folgenden Konstanten sind als Daten-Ausgänge im Sinne von globalen Variablen verfügbar (Auszug):

- `M_FOR_EXPERIMENTAL_OR_DEVELOPMENTAL_USE_ONLY` (UINT, Wert 0)
- `M_BENDIX_COMMERCIAL_VEHICLE_SYSTEMS` (UINT, Wert 1)
- `M_ALLISON_TRANSMISSION` (UINT, Wert 2)
- ... (weitere Einträge bis `M_THOMAS_G_FARIA` mit Wert 1863)

Eine vollständige Liste befindet sich in der XML-Definition.

Adapter

Keine.

Funktionsweise

Die Konstanten sind als globale Konstanten definiert und stehen dem gesamten IEC-61499-Anwendungsmodell zur Verfügung. Sie können direkt referenziert werden, z.B. um die Hersteller-ID eines ISOBUS-Geräts zu setzen oder zu interpretieren. Die Werte sind hartcodiert und unveränderlich (CONSTANT).

Technische Besonderheiten

- Die Konstanten sind vom Typ `UINT` (vorzeichenloser 16-Bit-Integer).

- Der Gültigkeitsbereich umfasst historisch vergebene IDs sowie reservierte und unbenutzte Nummern.
- Einige Einträge verweisen auf frühere Herstellernamen oder sind als Duplikate gekennzeichnet.
- Die Liste basiert auf der offiziellen ISOBUS-Listenverwaltung und wird stetig erweitert.

Zustandsübersicht

Da es sich um Konstanten handelt, existiert kein Zustandsautomaten. Der Baustein ist passiv und liefert nur Werte.

Anwendungsszenarien

- Identifikation des Herstellers eines angeschlossenen ISOBUS-Geräts.
- Festlegung der eigenen Hersteller-ID bei der Entwicklung von ISOBUS-kompatiblen Steuerungen.
- Verwendung in Diagnose- und Konfigurationstools zur Anzeige von Herstellerinformationen.

Vergleich mit ähnlichen Bausteinen

Es gibt keine direkten Funktionsblöcke, die diese Konstanten ersetzen. Andere GlobalConstants-Bausteine könnten ähnliche Nummernkreise (z.B. für Protokoll-IDs oder Fehlercodes) definieren, jedoch ist dieser speziell auf Hersteller-IDs ausgerichtet.

Fazit

Der GlobalConstants-Baustein `Manufacturer_IDs` stellt eine zentrale, umfassende Referenz für ISOBUS-Hersteller-IDs bereit. Er ist ein nützliches Hilfsmittel für Entwickler und Systemintegratoren, um herstellerspezifische Zuordnungen zu implementieren und zu verwalten. Seine Werte sind fest definiert und daher zuverlässig für den Einsatz in der Praxis.

Hinweis: Die vollständige Liste der Konstanten mit über 1600 Einträgen ist in der XML enthalten und kann bei Bedarf als Tabelle dargestellt werden. In dieser Dokumentation wurde nur ein Auszug gezeigt.

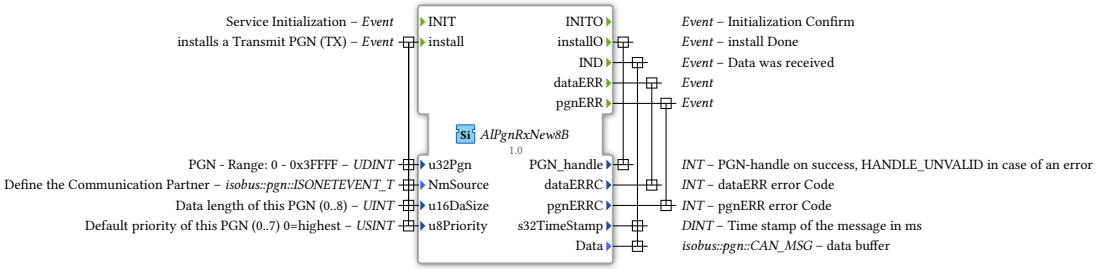
rx

AlPgnRxNew8B

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AlPgnRxNew8B` dient dem Empfang von Daten über ein CAN-Netzwerk nach dem ISOBUS-Standard (ISO 11783). Sein Hauptzweck ist die Installation und Verwaltung von Parameter Group Numbers (PGNs) für den Empfang von Nachrichten sowie die Bereitstellung empfangener Daten an die Anwendung. Er ist Teil einer spezialisierten Bibliothek für ISOBUS-Kommunikation.



Schnittstellenstruktur

Ereignis-Eingänge

- **`INIT`**: Startet die Initialisierung des Funktionsblocks. Bestätigung erfolgt über `INITO`.
- **`install`**: Löst die Installation einer neuen zu empfangenden PGN (Transmit PGN, TX) aus. Erwartet die zugehörigen Parameter `u32Pgn`, `NmSource`, `u16DaSize` und `u8Priority`. Die Bestätigung oder das Ergebnis wird über `installO` gemeldet.

Ereignis-Ausgänge

- **`INITO`**: Bestätigt den Abschluss der Initialisierung (`INIT`).
- **`installO`**: Meldet den Abschluss einer Installationsanfrage. Liefert den `PGN_handle` (positiv bei Erfolg, `HANDLE_UNVALID` im Fehlerfall).

- **IND**: Wird ausgelöst, wenn neue Daten für eine installierte PGN empfangen wurden. Liefert die empfangenen Data und einen Zeitstempel s32TimeStamp.
- **dataERR**: Wird bei einem Fehler im Datenempfang ausgelöst. Liefert einen Fehlercode dataERRC.
- **pgnERR**: Wird bei einem Fehler im Zusammenhang mit der PGN-Verwaltung (z.B. Installation) ausgelöst. Liefert einen Fehlercode pgnERRC.

Daten-Eingänge

- **u32Pgn** (UDINT): Die zu installierende oder zu überwachende Parameter Group Number (PGN). Gültiger Bereich: 0 bis 0x3FFFF.
- **NmSource** (isobus::pgn::ISONETEVENT_T): Definiert den Kommunikationspartner (z.B. eine bestimmte Node-Adresse oder einen Broadcast).
- **u16DaSize** (UINT): Die zu erwartende Datenlänge der PGN in Bytes (0..8).
- **u8Priority** (USINT): Die Standard-Priorität dieser PGN (0..7), wobei 0 die höchste Priorität darstellt. Initialwert ist 7 (niedrigste Priorität).

Daten-Ausgänge

- **PGN_handle** (INT): Ein Handle (Referenznummer) für die erfolgreich installierte PGN. Im Fehlerfall enthält es den Wert HANDLE_UNVALID.
- **dataERRC** (INT): Fehlercode, der bei Auslösen des dataERR-Ereignisses gesetzt wird.
- **pgnERRC** (INT): Fehlercode, der bei Auslösen des pgnERR-Ereignisses gesetzt wird.
- **s32TimeStamp** (DINT): Zeitstempel der empfangenen Nachricht in Millisekunden. Initialwert ist −1.
- **Data** (isobus::pgn::CAN_MSG): Der Puffer, der die empfangenen CAN-Nachrichtendaten enthält.

Adapter

Dieser Funktionsblock verwendet keine Adapter-Schnittstellen.

Funktionsweise

1. **Initialisierung**: Durch das INIT-Ereignis wird der Funktionsblock betriebsbereit gemacht. Nach Abschluss wird INITO ausgelöst.
2. **PGN-Installation**: Über das install-Ereignis können eine oder mehrere PGNs für den Empfang registriert werden. Der Block konfiguriert interne Filter im CAN-Stack basierend auf den übergebenen Parametern (PGN, Quelle, Länge, Priorität). Das Ergebnis (Handle oder Fehler) wird via installO zurückgemeldet.
3. **Datenempfang**: Sobald eine Nachricht für eine installierte PGN eintrifft, wird das IND-Ereignis ausgelöst. Die empfangenen Daten (Data) und ein Zeitstempel (s32TimeStamp) stehen dann an den Ausgängen zur Verfügung.
4. **Fehlerbehandlung**: Treten während des Empfangs (dataERR) oder bei der PGN-Verwaltung (pgnERR) Fehler auf, werden die entsprechenden Ereignisse mit einem Fehlercode ausgelöst.

Technische Besonderheiten

- Der Block ist für den Empfang von PGNs mit einer Datenlänge von bis zu 8 Bytes ausgelegt (daher „8B“ im Namen).
- Er verwendet spezifische ISOBUS-Datentypen (isobus::pgn::CAN_MSG, isobus::pgn::ISONETEVENT_T), die eine Integration in einen ISOBUS-spezifischen Software-Stack voraussetzen.
- Die Fehlerbehandlung ist in zwei separate Kanäle unterteilt: dataERR für Laufzeitfehler beim Empfang und pgnERR für Konfigurationsfehler.

Zustandsübersicht

Der Block verfügt über implizite Zustände:

- **Nicht initialisiert**: Vor dem ersten INIT-Ereignis.
- **Bereit**: Nach erfolgreicher Initialisierung (INITO). Kann install-Anfragen entgegennehmen.
- **Aktiv/Empfangend**: Nach erfolgreicher Installation mindestens einer PGN. Kann IND-Ereignisse für empfangene Daten auslösen.
- **Fehler**: Bei Auftreten eines Fehlers (pgnERR oder dataERR). Der Block kann je nach Implementierung im Fehlerzustand verbleiben oder nach Behebung der Ursache wieder funktionsfähig sein.

Anwendungsszenarien

- **Landmaschinensteuerung**: Empfang von Arbeitsgerätedaten (z.B. Drillmaschinen, Düngerstreuer) im ISOBUS-Netzwerk eines Traktors.
- **Anhängerkommunikation**: Empfang von Statusinformationen (Bremsdruck, Beleuchtung, Türkontakte) von einem intelligenten Anhänger.
- **Implementüberwachung**: Empfang von Echtzeitdaten von Sensoren an einem Gerät (z.B. Saatgutfluss, Arbeitsbreite).

⚖️ Vergleich mit ähnlichen Bausteinen

- **Generische CAN-Empfangsblöcke**: Im Gegensatz zu generischen `CAN_RX`-Blöcken, die Roh-CAN-IDs und Daten verarbeiten, abstrahiert `AlPgnRxNew8B` die ISOBUS-spezifische PGN-Adressierung und -Verwaltung.
- **`E_SR` (Service Interface Function Block)**: Während `E_SR` ein generisches Server-Request-Muster für Dienste bereitstellt, ist `AlPgnRxNew8B` speziell auf den zyklischen oder ereignisgesteuerten Empfang von ISOBUS-Nachrichten ausgelegt.
- **Blöcke für andere Protokolle**: Ähnliche Blöcke existieren für andere höhere Protokolle (z.B. J1939), unterscheiden sich aber in den spezifischen Adressierungs- und Nachrichtenformaten.

🛠️ Zugehörige Übungen

- Uebung_130
- Uebung_134

Fazit

Der `AlPgnRxNew8B` ist ein essentieller Baustein für die Entwicklung von ISOBUS-kompatiblen Empfangsanwendungen in der 4diac IDE. Er kapselt die Komplexität der PGN-Filterung und -Verwaltung und bietet der Anwendung eine saubere, ereignisgesteuerte Schnittstelle für den Empfang von CAN-Nachrichten nach dem ISO 11783-Standard. Seine klare Trennung von Konfiguration (`install`), Datentransfer (`IND`) und Fehlerbehandlung macht ihn robust und einfach in übergeordnete Steuerungslogiken zu integrieren.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

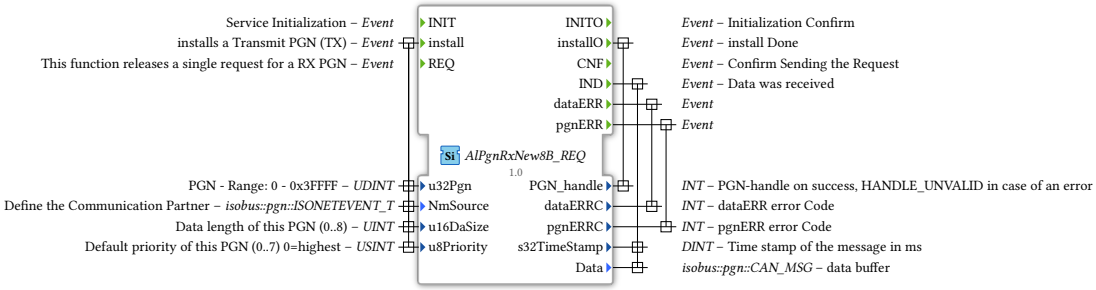
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AlPgnRxNew8B_REQ

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AlPgnRxNew8B_REQ` dient zur Anforderung (Request) von Daten über ein ISOBUS-Netzwerk. Er ermöglicht die Installation eines Empfangsparametersatzes (RX PGN) und das Auslösen einer einmaligen Anfrage für diesen Parameter. Der Baustein ist Teil einer ISOBUS-spezifischen Kommunikationsbibliothek und behandelt die Interaktion mit dem CAN-Netzwerkprotokoll nach ISO 11783.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisiert den Funktionsblock.
- **install**: Installiert einen Empfangsparametersatz (RX PGN) für die spätere Datenanforderung. Wird mit den Daten `u32Pgn`, `NmSource`, `u16DaSize` und `u8Priority` ausgelöst.
- **REQ**: Löst eine einmalige Anforderung für den zuvor installierten RX PGN aus.

Ereignis-Ausgänge

- **INITO**: Bestätigt die abgeschlossene Initialisierung.
- **installO**: Bestätigt die abgeschlossene Installation eines PGN. Liefert den zugewiesenen `PGN_handle`.
- **CNF**: Bestätigt das erfolgreiche Absenden der Anforderung (Request) an das Netzwerk.
- **IND**: Wird ausgelöst, wenn die angeforderte Daten-Nachricht empfangen wurde. Liefert die `Data` und einen `s32TimeStamp`.
- **dataERR**: Signalisiert einen Fehler bei der Datenverarbeitung. Liefert den Fehlercode `dataERRC`.
- **pgnERR**: Signalisiert einen Fehler bei der PGN-Verarbeitung (z.B. bei der Installation). Liefert den Fehlercode `pgnERRC`.

Daten-Eingänge

- **u32Pgn** (UDINT): Die Parameter Group Number (PGN) im Bereich 0 bis 0x3FFFF.
- **NmSource** (isobus::pgn::ISONETEVENT_T): Definiert den Kommunikationspartner im Netzwerk.
- **u16DaSize** (UINT): Die zu erwartende Datenlänge der PGN (0..8 Byte).
- **u8Priority** (USINT): Die Standard-Priorität dieser PGN (0..7, wobei 0 die höchste Priorität ist). Initialwert ist 7.

Daten-Ausgänge

- **PGN_handle** (INT): Ein Handle zur Identifikation des erfolgreich installierten PGN. Im Fehlerfall wird ein ungültiger Handle-Wert (`HANDLE_UNVALID`) zurückgegeben.
- **dataERRC** (INT): Fehlercode, der bei Auslösen des `dataERR`-Ereignisses gesetzt wird.
- **pgnERRC** (INT): Fehlercode, der bei Auslösen des `pgnERR`-Ereignisses gesetzt wird.
- **s32TimeStamp** (DINT): Zeitstempel der empfangenen Nachricht in Millisekunden. Initialwert ist −1.
- **Data** (isobus::pgn::CAN_MSG): Der Puffer, der die empfangenen CAN-Nachrichtendaten enthält.

Adapter

Dieser Funktionsblock verwendet keine Adapter-Schnittstellen.

Funktionsweise

Der Funktionsblock arbeitet in zwei Hauptphasen: Installation und Anforderung.

1. **Installation**: Durch das `install`-Ereignis werden die Parameter für einen zu empfangenden PGN (Nummer, Quelle, Größe, Priorität) konfiguriert. Der Block registriert diese Anforderung im zugrunde liegenden ISOBUS-Stack und gibt einen `PGN_handle` über `installO` zurück. Dieser Handle wird für spätere Referenzen benötigt.
2. **Anforderung**: Das `REQ`-Ereignis löst eine einmalige Anfrage für den zuletzt installierten PGN im Netzwerk aus. Nach erfolgreichem Versand der Anfrage wird `CNF` ausgelöst. Wenn die angeforderte Nachricht vom definierten `NmSource` eintrifft, wird sie im `Data`-Puffer abgelegt und zusammen mit einem Zeitstempel über das `IND`-Ereignis ausgegeben.

Fehler während der Installation (z.B. ungültige PGN) führen zum `pgnERR`-Ausgang. Fehler beim Empfang oder der Verarbeitung der Daten führen zum `dataERR`-Ausgang.

Technische Besonderheiten

- Der Block ist für die Verarbeitung von PGNs mit einer Datenlänge von bis zu 8 Byte ausgelegt (`u16DaSize` 0..8).
- Die Prioritätsverwaltung (`u8Priority`) folgt dem ISOBUS-Standard.
- Die Daten werden in einem typsicheren `CAN_MSG`-Puffer bereitgestellt.
- Die Fehlerbehandlung ist über dedizierte Ereignisausgänge (`pgnERR`, `dataERR`) strukturiert.

Zustandsübersicht

1. **Nicht initialisiert**: Vor dem ersten `INIT`-Ereignis.
2. **Initialisiert / Bereit**: Nach `INITO`. Der Block kann `install`-Ereignisse entgegennehmen.
3. **PGN installiert**: Nach erfolgreichem `installO`. Der Block ist bereit, `REQ`-Ereignisse zu verarbeiten.
4. **Anforderung ausstehend**: Nach `REQ` und vor `CNF`. Die Netzwerkanfrage wurde gestellt.
5. **Datenempfangsbereit**: Nach `CNF`. Der Block wartet auf die eingehende Nachricht und wird bei Empfang `IND` auslösen.

Anwendungsszenarien

Typische Anwendungen liegen in der landwirtschaftlichen oder mobilen Arbeitsmaschinensteuerung (ISO 11783 / ISOBUS):

- Abfrage von Maschinenparametern (z.B. Drehzahl, Temperatur) von einem Anbaugerät.
- Einmaliges Anfordern von Konfigurationsdaten von einem Steuergerät im Netzwerk.
- Implementierung von Diagnose- oder Service-Tools, die spezifische PGNs abfragen müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu generischen CAN-Empfangsblöcken ist `AlPgnRxNew8B_REQ` speziell auf das ISOBUS-Protokoll und dessen PGN-Konzept zugeschnitten. Er abstrahiert die niedrigleveligen CAN-Details (wie ID-Filterung) und bietet eine saubere Schnittstelle auf PGN-Ebene. Blöcke wie `E_CYCLE` oder `E_DELAY` könnten verwendet werden, um periodische Anfragen zu realisieren, während `AlPgnRxNew8B_REQ` selbst nur Einzelanfragen behandelt.

🛠️ Zugehörige Übungen

- Uebung_132

Fazit

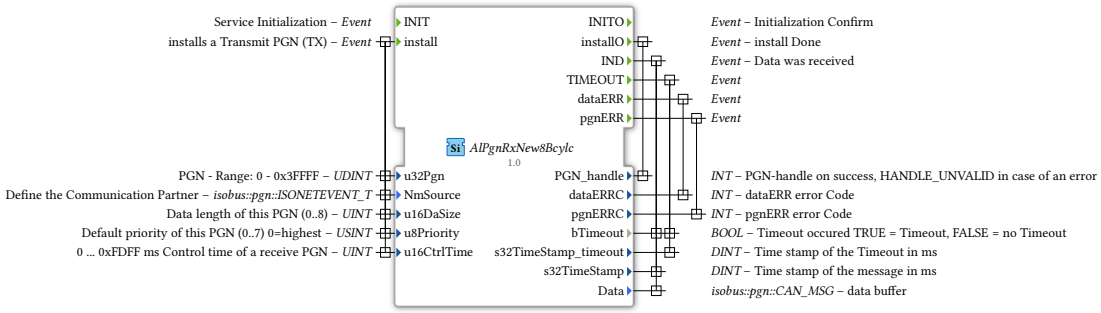
Der `AlPgnRxNew8B_REQ` ist ein spezialisierter Funktionsblock für die bedarfsgesteuerte (on-demand) Kommunikation in ISOBUS-Netzwerken. Durch die Trennung von Installation (`install`) und Ausführung (`REQ`) ermöglicht er eine flexible und kontrollierte Datenanforderung. Seine integrierte Fehlerbehandlung und typsichere Datenschnittstelle machen ihn zu einem robusten Baustein für anspruchsvolle Embedded-Steuerungen in der Agrartechnik.

AlPgnRxNew8Bcylc

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `AlPgnRxNew8Bcylc` dient dem zyklischen Empfang von Daten über ein ISOBUS-Netzwerk gemäß dem Parameter Group Number (PGN) Protokoll. Er ermöglicht die Konfiguration und Überwachung von Empfangskanälen für spezifische PGNs, inklusive der Handhabung von Timeouts und Fehlerzuständen. Der Block ist für den Einsatz in Steuerungssystemen konzipiert, die eine zuverlässige und überwachte Kommunikation mit anderen ISOBUS-Geräten erfordern.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisiert den Funktionsblock.
- **install**: Installiert einen neuen Empfangskanal für eine spezifische PGN. Löst die Konfiguration mit den zugehörigen Daten-Eingängen aus.

Ereignis-Ausgänge

- **INITO**: Bestätigt die abgeschlossene Initialisierung.
- **installO**: Signalisiert den Abschluss einer Installationsanfrage. Liefert den zugewiesenen `PGN_handle`.
- **IND**: Wird ausgelöst, wenn neue Daten für die konfigurierte PGN empfangen wurden.
- **TIMEOUT**: Signalisiert, dass für die konfigurierte PGN ein Timeout aufgetreten ist.
- **dataERR**: Zeigt einen Fehler bei den empfangenen Daten an.

- **pgnERR**: Zeigt einen Fehler im Zusammenhang mit der PGN-Konfiguration oder -Verarbeitung an.

Daten-Eingänge

- **u32Pgn** (UDINT): Die zu empfangende Parameter Group Number (PGN). Gültiger Bereich: 0 bis 0x3FFFF.
- **NmSource** (isobus::pgn::ISONETEVENT_T): Definiert den Kommunikationspartner (Netzwerk-Management-Quelle).
- **u16DaSize** (UINT): Die zu erwartende Datenlänge der PGN in Bytes (0..8).
- **u8Priority** (USINT): Die Standard-Priorität dieser PGN (0..7, wobei 0 die höchste Priorität ist). Initialwert: 7.
- **u16CtrlTime** (UINT): Die Kontrollzeit (Timeout-Zeit) für den Empfang dieser PGN in Millisekunden (0 ... 0xFDFF ms). Ein Wert von 0 deaktiviert die Timeout-Überwachung.

Daten-Ausgänge

- **PGN_handle** (INT): Ein Handle, das den erfolgreich installierten Empfangskanal identifiziert. Im Fehlerfall wird `HANDLE_UNVALID` ausgegeben.
- **dataERRC** (INT): Fehlercode, der bei Auslösung des `dataERR`-Ereignisses gesetzt wird.
- **pgnERRC** (INT): Fehlercode, der bei Auslösung des `pgnERR`-Ereignisses gesetzt wird.
- **bTimeout** (BOOL): Zeigt an, ob ein Timeout aufgetreten ist (`TRUE` = Timeout, `FALSE` = kein Timeout). Initialwert: `FALSE`.
- **s32TimeStamp_timeout** (DINT): Zeitstempel des aufgetretenen Timeouts in Millisekunden.
- **s32TimeStamp** (DINT): Zeitstempel der zuletzt empfangenen Nachricht in Millisekunden. Initialwert: −1.
- **Data** (isobus::pgn::CAN_MSG): Puffer, der die empfangenen Nutzdaten enthält.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

1. **Initialisierung**: Durch das `INIT`-Ereignis wird der Funktionsblock betriebsbereit gemacht. Die Bestätigung erfolgt über `INITO`.
2. **Kanalinstallation**: Ein Empfangskanal wird durch das `install`-Ereignis konfiguriert. Die dazugehörigen Parameter (`u32Pgn`, `NmSource`, `u16DaSize`, `u8Priority`, `u16CtrlTime`) definieren den gewünschten Datenstrom. Bei Erfolg wird ein eindeutiger `PGN_handle` zurückgegeben (`installO`). Bei Fehlern werden `pgnERR` oder `dataERR` ausgelöst.
3. **Zyklischer Empfang**: Sobald ein Kanal installiert ist, lauscht der Block auf Nachrichten der konfigurierten PGN von der angegebenen Quelle. Bei Empfang einer gültigen Nachricht werden die Daten im `Data`-Ausgang bereitgestellt und das `IND`-Ereignis ausgelöst. Der `s32TimeStamp` wird aktualisiert.
4. **Timeout-Überwachung**: Ist `u16CtrlTime` > 0, überwacht der Block die Zeit zwischen empfangenen Nachrichten. Wird innerhalb dieses Zeitfensters keine neue Nachricht empfangen, wird das `TIMEOUT`-Ereignis ausgelöst, `bTimeout` auf `TRUE` gesetzt und `s32TimeStamp_timeout` aktualisiert.
5. **Fehlerbehandlung**: Protokollfehler oder ungültige Konfigurationen führen zur Auslösung der entsprechenden Fehlerereignisse (`dataERR`, `pgnERR`) mit zugehörigem Fehlercode.

Technische Besonderheiten

- Der Block unterstützt die spezifischen Datentypen `isobus::pgn::CAN_MSG` und `isobus::pgn::ISONETEVENT_T` aus der ISOBUS-PGN-Bibliothek.
- Die Timeout-Überwachung ist optional und kann durch Setzen von `u16CtrlTime` auf 0 deaktiviert werden.
- Der `PGN_handle` dient als Referenz für den installierten Kanal und kann für spätere Verwaltungsoperationen (z.B. Deinstallation) notwendig sein.

Zustandsübersicht

1. **Nicht initialisiert**: Nach dem Start. Auf `INIT` wartend.
2. **Bereit**: Nach erfolgreicher Initialisierung (`INITO`). Kann `install`-Anfragen entgegennehmen.
3. **Empfangsbereit (Kanal aktiv)**: Nach erfolgreicher Installation eines Kanals (`installO`). Überwacht den Bus auf die konfigurierte PGN und löst `IND` bei Empfang oder `TIMEOUT` bei Überschreitung der Kontrollzeit aus.

4. **Fehlerzustand**: Wird durch die Ereignisse `pgnERR` oder `dataERR` signalisiert. Der genaue Fehler ist dem jeweiligen Fehlercode zu entnehmen. Der Block kann in der Regel nach Behebung der Ursache weiter verwendet werden.

Anwendungsszenarien

- **Landmaschinensteuerung**: Empfang von Arbeitsgerätedaten (z.B. Drillmaschinen-Status) von einem Anbaugerät am ISOBUS.
- **Fahrzeugdiagnose**: Zyklischer Empfang von Motordaten (Drehzahl, Temperatur) zur Anzeige und Überwachung im Terminal.
- **Implement Control**: Empfang von Sollwerten (z.B. Soll-Tiefe) von einer Bedienkonsole für einen hydraulischen Unterlenker.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu generischen CAN-Empfangsblöcken ist `AlPgnRxNew8Bcylc` speziell auf das ISOBUS-PGN-Protokoll zugeschnitten. Er bietet höhere Abstraktion durch direkte PGN-Adressierung, integrierte Partnererkennung (`NmSource`) und automatische Timeout-Verwaltung. Einfache Empfangsblöcke würden diese Protokollschicht und Verwaltungsfunktionen nicht bereitstellen.

🛠️ Zugehörige Übungen

- Uebung_131

Fazit

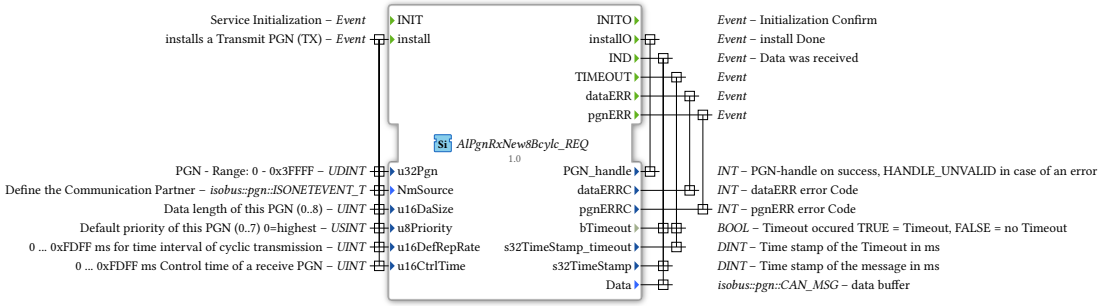
Der `AlPgnRxNew8Bcylc` ist ein essentieller Baustein für robuste ISOBUS-Anwendungen, die zyklische Daten von anderen Netzwerkteilnehmern empfangen müssen. Durch seine integrierten Funktionen zur Kanalverwaltung, Zeitüberwachung und Fehlererkennung vereinfacht er die Entwicklung zuverlässiger Kommunikationsschnittstellen erheblich und entlastet den Applikationsentwickler von niedriglegigen Protokolldetails.

AlPgnRxNew8Bcylc_REQ

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AlPgnRxNew8Bcylc_REQ` dient zur zyklischen Anforderung von Daten über ein CAN-Netzwerk gemäß dem ISOBUS-Standard (ISO 11783). Sein Hauptzweck ist die Installation und Verwaltung von Empfangsparametern für spezifische Parameter Group Numbers (PGNs). Der Baustein ermöglicht die Konfiguration eines zyklischen Empfangs und überwacht den Datenfluss, indem er bei erfolgreichem Empfang, bei Timeouts oder bei Fehlern entsprechende Ereignisse auslöst.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisiert den Funktionsblock.
- **install**: Löst die Installation einer Empfangs-PGN (RX) mit den zugehörigen Konfigurationsparametern aus. Die mitgeführten Daten sind: `u32Pgn`, `NmSource`, `u16DaSize`, `u8Priority`, `u16DefRepRate`, `u16CtrlTime`.

Ereignis-Ausgänge

- **INITO**: Bestätigt die abgeschlossene Initialisierung.

- **installO**: Signalisiert den Abschluss des Installationsvorgangs. Führt den `PGN_handle` als Ergebnis mit.
- **IND**: Wird ausgelöst, wenn neue Daten empfangen wurden. Führt `bTimeout`, `s32TimeStamp` und `Data` mit.
- **TIMEOUT**: Wird bei einem Kontrollzeit-Timeout ausgelöst. Führt `bTimeout` und `s32TimeStamp_timeout` mit.
- **dataERR**: Zeigt einen Fehler bei den empfangenen Daten an. Führt den Fehlercode `dataERRC` mit.
- **pgnERR**: Zeigt einen Fehler bei der PGN-Verarbeitung an. Führt den Fehlercode `pgnERRC` mit.

Daten-Eingänge

- **u32Pgn** (UDINT): Die zu empfangende Parameter Group Number (PGN). Gültiger Bereich: 0 bis 0x3FFFF.
- **NmSource** (isobus::pgn::ISONETEVENT_T): Definiert den Kommunikationspartner (z.B. eine bestimmte Node-Adresse).
- **u16DaSize** (UINT): Die zu erwartende Datenlänge der PGN in Bytes (0..8).
- **u8Priority** (USINT): Die Standard-Priorität für diese PGN (0..7, wobei 0 die höchste Priorität ist). Initialwert: 7.
- **u16DefRepRate** (UINT): Das erwartete zyklische Sendeintervall der Quell-PGN in Millisekunden (0 … 0xFDFF ms).
- **u16CtrlTime** (UINT): Die Kontrollzeit in Millisekunden (0 … 0xFDFF ms), nach deren Überschreitung ohne Empfang ein `TIMEOUT`-Ereignis generiert wird.

Daten-Ausgänge

- **PGN_handle** (INT): Ein Handle (Bezeichner) für die erfolgreich installierte PGN. Im Fehlerfall wird ein ungültiger Handle (HANDLE_UNVALID) zurückgegeben.
- **dataERRC** (INT): Fehlercode, der bei einem `dataERR`-Ereignis ausgegeben wird.
- **pgnERRC** (INT): Fehlercode, der bei einem `pgnERR`-Ereignis ausgegeben wird.
- **bTimeout** (BOOL): Statusflag für einen aufgetretenen Timeout. `TRUE` = Timeout ist eingetreten, `FALSE` = kein Timeout. Initialwert: `FALSE`.
- **s32TimeStamp_timeout** (DINT): Zeitstempel in Millisekunden, zu dem der Timeout erkannt wurde.
- **s32TimeStamp** (DINT): Zeitstempel in Millisekunden des zuletzt empfangenen gültigen Datensatzes. Initialwert: −1.
- **Data** (isobus::pgn::CAN_MSG): Puffer, der die empfangenen CAN-Nachrichtendaten enthält.

Adapter

Dieser Funktionsblock verwendet keine Adapter-Schnittstellen.

Funktionsweise

1. **Initialisierung**: Durch das `INIT`-Ereignis wird der Baustein in einen betriebsbereiten Grundzustand versetzt, was durch `INITO` quittiert wird.
2. **PGN-Installation**: Das `install`-Ereignis konfiguriert eine neue Empfangs-PGN. Alle notwendigen Parameter (PGN, Quelle, Datenlänge, etc.) werden übergeben. Bei erfolgreicher Konfiguration antwortet der Block mit `installO` und liefert einen gültigen `PGN_handle`. Bei Fehlern werden `pgnERR` oder `dataERR` ausgelöst.
3. **Zyklischer Empfang**: Nach erfolgreicher Installation überwacht der Baustein den CAN-Bus auf Nachrichten der konfigurierten PGN und Quelle.
4. **Datenindikation**: Bei Empfang einer gültigen Nachricht wird das `IND`-Ereignis ausgelöst. Die Daten (`Data`), ein Zeitstempel (`s32TimeStamp`) und der Timeout-Status (`bTimeout=FALSE`) werden ausgegeben.
5. **Timeout-Überwachung**: Überschreitet die Zeit seit dem letzten empfangenen Paket die konfigurierte `u16CtrlTime`, wird das `TIMEOUT`-Ereignis ausgelöst. `bTimeout` wird auf `TRUE` gesetzt und ein Zeitstempel (`s32TimeStamp_timeout`) bereitgestellt.
6. **Fehlerbehandlung**: Treten Protokoll- oder Datenfehler auf, werden die Ereignisse `pgnERR` bzw. `dataERR` mit den entsprechenden Fehlercodes generiert.

Technische Besonderheiten

- Der Baustein ist speziell für den Einsatz in ISOBUS-Umgebungen (Landtechnik) vorgesehen und nutzt typsichere Datentypen aus der `isobus::pgn`-Bibliothek (`CAN_MSG`, `ISONETEVENT_T`).
- Die Timeout-Überwachung (`u16CtrlTime`) ist unabhängig vom erwarteten Sendeintervall (`u16DefRepRate`) und dient der Robustheit, um ausgefallene Kommunikationspartner zu erkennen.

- Der `PGN_handle` ermöglicht die eindeutige Identifikation und spätere Verwaltung (z.B. Deinstallation) einer konfigurierten PGN-Instanz innerhalb einer Applikation.

Zustandsübergangsübersicht

1. **Nicht initialisiert**: Nach dem Start. Auf `INIT` wartend.
2. **Bereit**: Nach `INIT0`. Kann `install`-Anfragen entgegennehmen.
3. **Installiert**: Nach erfolgreichem `install0`. Überwacht aktiv den CAN-Bus auf die konfigurierte PGN.
   - Bei Empfang: Übergang in Zustand „Daten verfügbar“ (löst `IND` aus), danach Rückkehr zu „Installiert“.
   - Bei Timeout: Löst `TIMEOUT` aus, bleibt im Zustand „Installiert“ (weitere Überwachung).
   - Bei Fehler: Löst `pgnERR`/`dataERR` aus, bleibt im Zustand „Installiert“.

Anwendungsszenarien

- **ISOBUS-Implementierungen**: Empfang von zyklischen Daten (z.B. Drehzahl, Druck, Position) von einem Steuergerät (ECU) eines Anbaugeräts im Traktor.
- **Überwachungsfunktionen**: Kontinuierliche Überprüfung, ob eine kritische Komponente (z.B. Motorsteuerung) noch „lebendig“ ist und Daten sendet (mittels `CtrlTime`).
- **Datenlogger**: Zyklisches Sammeln von Prozessdaten aus dem CAN-Netzwerk für Analyse- oder Speicherzwecke.

⚖️ Vergleich mit ähnlichen Bausteinen

- **E_CTU vs. AlPgnRxNew8Bcylc_REQ**: Ein einfacher Zähler (`E_CTU`) hat keine Netzwerkfunktionalität. Dieser Baustein ist ein spezialisierter, anwendungsnaher Kommunikationsblock für ein bestimmtes Protokoll (ISOBUS).
- **Generische CAN-RX-Blöcke**: Im Gegensatz zu Blöcken, die rohe CAN-IDs und Daten empfangen, arbeitet dieser Block auf der höheren, standardisierten PGN-Ebene des ISOBUS und übernimmt die protokollspezifische Dekodierung und Parameterverwaltung.

🛠️ Zugehörige Übungen

- Uebung_133

Fazit

Der `AlPgnRxNew8Bcylc_REQ` ist ein essentieller Baustein für die Implementierung von ISOBUS-konformen Empfangsfunktionalitäten in 4diac. Er abstrahiert die Komplexität der CAN-Kommunikation und bietet eine saubere, ereignisgesteuerte Schnittstelle zur zuverlässigen, zyklischen Datenerfassung mit integrierter Fehler- und Timeout-Erkennung. Seine Verwendung erhöht die Wiederverwendbarkeit und Robustheit von Steuerungsapplikationen in der Landtechnik.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

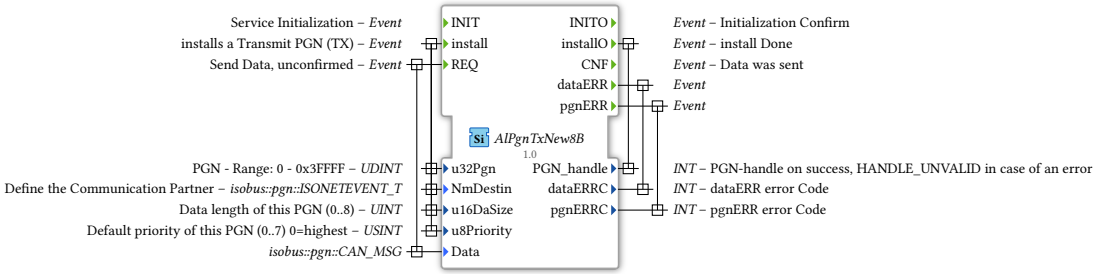
tx

AlPgnTxNew8B

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AlPgnTxNew8B` dient zum Senden von Daten über ein CAN-Netzwerk gemäß dem ISOBUS-Standard (ISO 11783). Sein Hauptzweck ist die Installation und Verwaltung von Parameter Group Numbers (PGNs) für die Übertragung (TX) und das anschließende Senden von Datenpaketen, sobald ein lokales Sendeereignis (`REQ`) eintritt. Er ist für Anwendungen konzipiert, die eine unbestätigte (unconfirmed) Datenübertragung erfordern.



Schnittstellenstruktur

Ereignis-Eingänge

- **`INIT`**: Startet die Initialisierung des Funktionsblocks.
- **`install`**: Löst die Installation einer neuen Transmit-PGN (TX-PGN) aus. Die Installation wird mit den über `With` verknüpften Daten-Eingängen (`u32Pgn`, `NmDestin`, `u16DaSize`, `u8Priority`) konfiguriert.
- **`REQ`**: Löst den Sendevorgang für die zuvor installierte PGN aus. Die zu sendenden Daten werden über den mit `With` verknüpften Daten-Eingang `Data` bereitgestellt.

Ereignis-Ausgänge

- **`INITO`**: Bestätigt den erfolgreichen Abschluss der Initialisierung (`INIT`).
- **`installO`**: Bestätigt den Abschluss der PGN-Installation. Liefert über `PGN_handle` einen Handle für die installierte PGN oder einen Fehlerwert.
- **`CNF`**: Bestätigt, dass die Daten erfolgreich gesendet wurden (Antwort auf `REQ`).
- **`dataERR`**: Zeigt einen Fehler im Zusammenhang mit den zu sendenden Daten (`Data`) an. Liefert einen Fehlercode über `dataERRC`.
- **`pgnERR`**: Zeigt einen Fehler im Zusammenhang mit der PGN-Installation oder -Verwendung an. Liefert einen Fehlercode über `pgnERRC`.

Daten-Eingänge

- **`u32Pgn`** (UDINT): Die zu installierende Parameter Group Number (PGN). Gültiger Bereich: 0 bis 0x3FFFF (dezimal 262143).
- **`NmDestin`** (isobus::pgn::ISONETEVENT_T): Definiert den Kommunikationspartner (z.B. Broadcast, spezifische Adresse).
- **`u16DaSize`** (UINT): Die Datenlänge der PGN in Bytes. Gültiger Bereich: 0 bis 8.
- **`u8Priority`** (USINT): Die Priorität der Nachricht auf dem CAN-Bus. Bereich: 0 (höchste) bis 7 (niedrigste). Standardwert: 7.
- **`Data`** (isobus::pgn::CAN_MSG): Die zu sendenden Daten, strukturiert als CAN-Nachricht.

Daten-Ausgänge

- **`PGN_handle`** (INT): Ein Handle (Bezeichner) für die erfolgreich installierte PGN. Im Fehlerfall wird der Wert `HANDLE_UNVALID` ausgegeben.
- **`dataERRC`** (INT): Numerischer Fehlercode, der bei Auslösen des `dataERR`-Ereignisses gesetzt wird.
- **`pgnERRC`** (INT): Numerischer Fehlercode, der bei Auslösen des `pgnERR`-Ereignisses gesetzt wird.

Adapter

Dieser Funktionsblock verwendet keine Adapter-Schnittstellen.

Funktionsweise

1. **Initialisierung**: Durch das Ereignis `INIT` wird der Funktionsblock in einen betriebsbereiten Zustand versetzt. Die Bestätigung erfolgt über `INITO`.
2. **PGN-Installation**: Das `install`-Ereignis konfiguriert eine spezifische PGN für die Datenübertragung. Dabei werden PGN-Nummer, Zieladresse, Datenlänge und Priorität festgelegt. Bei Erfolg wird ein `PGN_handle` über `installO` zurückgegeben. Bei einem Fehler (z.B. ungültige Parameter) wird das Ereignis `pgnERR` ausgelöst.
3. **Datenversand**: Sobald ein `REQ`-Ereignis eintritt, versucht der Block, die im `Data`-Eingang bereitgestellte CAN-Nachricht unter Verwendung der zuvor installierten PGN-Konfiguration zu senden. Bei Erfolg wird das `CNF`-Ereignis ausgelöst. Tritt ein Fehler beim Senden auf (z.B. ungültige Daten), wird `dataERR` ausgelöst.

Technische Besonderheiten

- Der Block ist für die Übertragung von Datenpaketen mit einer maximalen Länge von 8 Byte ausgelegt (typisch für klassische CAN-Frames).
- Er implementiert das unbestätigte Senden: Ein `CNF` bedeutet lediglich, dass die Nachricht zur Übertragung an die untere Protokollschicht übergeben wurde, nicht zwingend, dass sie den Empfänger erreicht hat.
- Die Fehlerbehandlung ist in zwei Kategorien unterteilt: PGN-bezogene Fehler (`pgnERR`) und Datenbezogene Fehler (`dataERR`), was die Fehlerdiagnose erleichtert.
- Der Block verwendet spezifische ISOBUS-Datentypen (`isobus::pgn::CAN_MSG`, `isobus::pgn::ISONETEVENT_T`).

Zustandsübersicht

Der Block durchläuft implizit folgende Hauptzustände:

1. **Nicht initialisiert**: Nach dem Start. Nur der `INIT`-Eingang ist aktiv.
2. **Initialisiert / Bereit**: Nach `INIT0`. Der Block kann nun PGNs installieren (`install`).
3. **PGN installiert**: Nach erfolgreichem `install0`. Der Block ist bereit, Daten zu senden (`REQ`).
4. **Sendeaktiv**: Während der Verarbeitung eines `REQ`. Übergang zu Zustand 3 nach `CNF` oder zu einem Fehlerzustand nach `dataERR`/`pgnERR`.

Anwendungsszenarien

- **Landmaschinensteuerung (ISOBUS)**: Senden von Maschinendaten (z.B. Arbeitsgeschwindigkeit, Zapfwellendrehzahl) an ein Terminal oder andere Steuergeräte im Traktor.
- **Nutzfahrzeugkommunikation**: Übertragung von Fahrzeugdaten (z.B. Drehmoment, Verbrauch) innerhalb eines Truck-Bussystems.
- **Industrielle Automatisierung**: Senden von Steuerbefehlen oder Statusinformationen über CAN-Netzwerke in Maschinen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`E_SEND` (Standard 61499)**: Ein generischer Sende-Baustein. `AlPgnTxNew8B` ist spezialisiert auf ISOBUS/CAN mit PGN-Verwaltung, Priorisierung und Zieladressierung, während `E_SEND` protokollagnostisch ist und eine Adapterverbindung benötigt.
- **Bestätigte Sende-Bausteine**: `AlPgnTxNew8B` sendet unbestätigt (`CNF`). Für bestätigte Kommunikation (Request/Response) wären andere, spezifischere TX-PGN-Blöcke oder Protokollstapel erforderlich.

🛠️ Zugehörige Übungen

- Uebung_124
- Uebung_128

Fazit

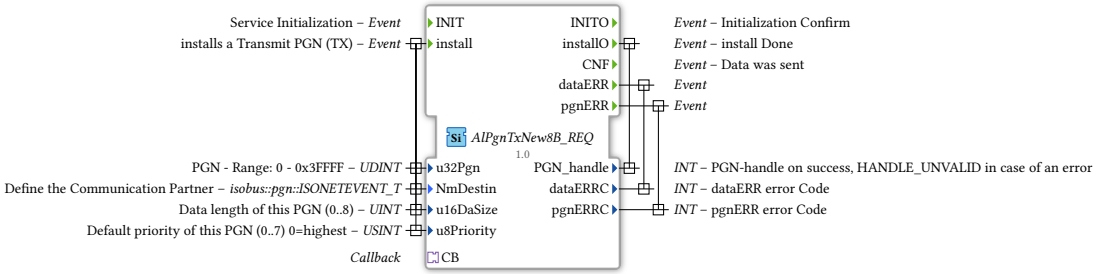
Der `AlPgnTxNew8B` ist ein spezialisierter Funktionsblock für die zuverlässige, konfigurierbare und unbestätigte Datenübertragung in ISOBUS-Umgebungen. Seine klare Trennung von Konfiguration (`install`) und Operation (`REQ`) sowie die differenzierte Fehlerrückmeldung machen ihn gut wartbar und einfach in übergeordnete Applikationslogiken zu integrieren. Er ist die erste Wahl, wenn CAN-Nachrichten nach ISOBUS-Standard mit einer festen PGN gesendet werden müssen.

AlPgnTxNew8B_REQ

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AlPgnTxNew8B_REQ` dient zur Konfiguration und Steuerung der Übertragung von Parameter-Gruppen-Nummern (PGN) in einem ISOBUS-Netzwerk (J1939). Sein Hauptzweck ist die Installation eines neuen Transmit-PGN (TX-PGN) im Netzwerk. Der Baustein reagiert auf ein REQ-Ereignis (J1939-Netzwerkereignis) und bietet eine Callback-Schnittstelle zur asynchronen Rückmeldung über den Status der Datenübertragung. Er ist Teil einer spezialisierten Bibliothek für ISOBUS-Kommunikation.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisiert den Funktionsblock. Muss vor der ersten Nutzung ausgelöst werden.
- **install**: Löst die Installation eines neuen Transmit-PGN (TX) aus. Die Installation wird mit den an diesem Ereignis gebundenen Datenparametern (`u32Pgn`, `NmDestin`, `u16DaSize`, `u8Priority`) konfiguriert.

Ereignis-Ausgänge

- **INITO**: Bestätigt die erfolgreiche Initialisierung des Bausteins.
- **installO**: Signalisiert den Abschluss der Installationsanfrage. Liefert den zugewiesenen `PGN_handle` oder im Fehlerfall einen ungültigen Handle.
- **CNF**: Bestätigt, dass Daten erfolgreich gesendet wurden. Wird typischerweise über den Callback-Adapter ausgelöst.
- **dataERR**: Zeigt einen Fehler im Zusammenhang mit den zu sendenden Daten an. Liefert einen Fehlercode über `dataERRC`.
- **pgnERR**: Zeigt einen Fehler im Zusammenhang mit der PGN-Konfiguration oder -Verwaltung an. Liefert einen Fehlercode über `pgnERRC`.

Daten-Eingänge

- **u32Pgn** (UDINT): Die zu installierende Parameter-Gruppen-Nummer (PGN). Gültiger Bereich: 0 bis 0x3FFFF (dezimal 262143).
- **NmDestin** (isobus::pgn::ISONETEVENT_T): Definiert den Kommunikationspartner (Zieladresse) für die PGN. Der genaue Typ wird durch die ISOBUS-Bibliothek definiert.
- **u16DaSize** (UINT): Die Länge der Nutzdaten für diese PGN in Bytes. Gültiger Bereich: 0 bis 8.
- **u8Priority** (USINT): Die Standard-Priorität für diese PGN. Gültiger Bereich: 0 (höchste) bis 7 (niedrigste). Initialwert: 7.

Daten-Ausgänge

- **PGN_handle** (INT): Ein Handle (Bezeichner) für die installierte PGN. Bei erfolgreicher Installation enthält er einen gültigen, positiven Wert. Im Fehlerfall wird ein ungültiger Handle (z.B. `HANDLE_UNVALID`) ausgegeben.
- **dataERRC** (INT): Der spezifische Fehlercode, wenn ein `dataERR`-Ereignis auftritt.
- **pgnERRC** (INT): Der spezifische Fehlercode, wenn ein `pgnERR`-Ereignis auftritt.

Adapter

- **CB** (Typ: `isobus::pgn::tx::Callback`): Ein Socket-Adapter, der eine Callback-Schnittstelle bereitstellt. Über diesen Adapter werden asynchrone Rückmeldungen (wie das `CNF`-Ereignis) vom zugrunde liegenden ISOBUS-Treibersystem an den Funktionsblock gemeldet. Dieser muss mit einem entsprechenden Plug-Adapter verbunden werden, um die Rückmeldungen zu empfangen.

Funktionsweise

1. **Initialisierung**: Zuerst muss das `INIT`-Ereignis ausgelöst werden, um den Baustein betriebsbereit zu machen. Dies führt zur Ausgabe von `INITO`.
2. **PGN-Installation**: Das `install`-Ereignis startet den Konfigurationsvorgang. Der Baustein übergibt die Parameter `u32Pgn`, `NmDestin`, `u16DaSize` und `u8Priority` an das ISOBUS-Protokoll-Stack, um eine neue Sende-PGN zu registrieren.

3. **Rückmeldung**: Nach der Verarbeitung der Installationsanfrage antwortet der Baustein mit dem `installO`-Ereignis und liefert den `PGN_handle`. Dieser Handle muss für spätere Sendeoperationen (z.B. mit einem separaten Sende-Baustein) verwendet werden.
4. **Asynchrone Betriebsrückmeldungen**: Während des Betriebs, wenn Daten für die konfigurierte PGN gesendet werden, erfolgt die Bestätigung (`CNF`) oder Fehlermeldung (`dataERR`, `pgnERR`) asynchron über die jeweiligen Ereignisausgänge. Die Auslösung des `CNF`-Ereignisses wird typischerweise durch den Callback-Adapter (`CB`) gesteuert.

Technische Besonderheiten

- Der Baustein ist spezifisch für ISOBUS/J1939-Netzwerke und nutzt typsichere Datenstrukturen (`isobus::pgn::ISONETEVENT_T`).
- Die Fehlerbehandlung ist aufgeteilt in `dataERR` (Datenfehler) und `pgnERR` (PGN-Konfigurationsfehler), was eine präzise Fehlerdiagnose ermöglicht.
- Die Kommunikation mit dem eigentlichen Netzwerk-Stack erfolgt über eine Callback-Schnittstelle (`CB`-Adapter), die eine lose Kopplung und asynchrone Benachrichtigung ermöglicht.
- Der `PGN_handle` dient als abstrakter Verweis auf die intern verwaltete PGN-Instanz.

Zustandsübersicht

Der Baustein durchläuft implizit folgende Hauptzustände:

1. **Nicht initialisiert**: Vor dem ersten `INIT`-Ereignis.
2. **Bereit**: Nach erfolgreicher Initialisierung (`INITO` empfangen). Der Baustein kann nun `install`-Anfragen entgegennehmen.
3. **Installation läuft**: Nach Auslösen von `install`. Der Baustein wartet auf die Antwort vom Protokoll-Stack.
4. **Betriebsbereit**: Nach erfolgreichem `installO`. Die PGN ist im Stack registriert und kann für den Versand genutzt werden. Asynchrone Ereignisse (`CNF`, `dataERR`, `pgnERR`) können nun auftreten.

Anwendungsszenarien

- **Landmaschinensteuerung**: Einarbeitung eines neuen Steuergeräts in ein ISOBUS-Netzwerk, das regelmäßig Betriebsdaten (z.B. Drehzahl, Temperatur) senden muss.
- **Anbaugeräte**: Dynamische Konfiguration der Kommunikation zwischen einem Traktor und einem spezifischen Anbaugerät, das eine individuelle PGN für seine Daten benötigt.
- **Diagnosetools**: Ein Diagnosegerät, das temporär eine PGN im Netzwerk installiert, um bestimmte Daten anzufordern oder Befehle zu senden.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`E_SEND` / `E_RCV` (Standard 61499)**: Diese generischen Kommunikationsbausteine sind protokollunabhängig. `AlPgnTxNew8B_REQ` ist hingegen spezialisiert auf ISOBUS/J1939 und übernimmt die protokollspezifische Konfiguration (PGN, Priorität, Zieladresse), die bei den Standardbausteinen manuell in den Datenparametern abgebildet werden müsste.
- **Einfache TX-Bausteine**: Andere ISOBUS-Sendebausteine erwarten oft einen bereits konfigurierten `PGN_handle`. `AlPgnTxNew8B_REQ` ist der vorgelagerte Baustein, der genau diesen Handle durch die Installation einer neuen PGN bereitstellt.

🛠️ Zugehörige Übungen

- Uebung_125

Fazit

Der `AlPgnTxNew8B_REQ` ist ein essentieller Konfigurationsbaustein für die ISOBUS-Kommunikation in 4diac. Er abstrahiert die komplexe Einrichtung einer Sende-PGN in einem J1939-Netzwerk hinter einer klaren, ereignisgesteuerten Schnittstelle. Die Trennung von Installation (`install`/`installO`) und Betrieb (`CNF`/`*ERR`) sowie die Nutzung eines Callback-Adapters machen ihn robust und gut integrierbar in größere Steuerungsanwendungen für die mobile Arbeitsmaschinenkommunikation.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

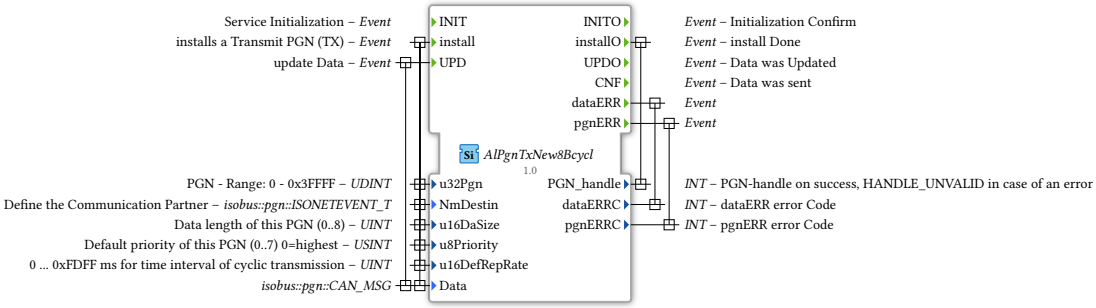
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AlPgnTxNew8Bcycl

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AlPgnTxNew8Bcycl` dient der zyklischen Übertragung von Daten über ein CAN-Netzwerk gemäß dem ISOBUS-Standard (ISO 11783). Sein Hauptzweck ist die Installation, Konfiguration und regelmäßige Aussendung von Parameter Group Numbers (PGNs). Er ermöglicht die Definition von Kommunikationseigenschaften wie Zieladresse, Priorität und Sendeintervall und stellt sicher, dass Daten zuverlässig und im konfigurierten Rhythmus versendet werden.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisiert den Funktionsblock.
- **install**: Installiert eine neue zu sendende PGN (TX) mit den zugehörigen Konfigurationsdaten. Löst die Verarbeitung der mitgeführten Daten-Eingänge (`u32Pgn`, `NmDestin`, `u16DaSize`, `u8Priority`, `u16DefRepRate`, `Data`) aus.
- **UPD**: Aktualisiert die Nutzdaten (`Data`) für die zuvor installierte PGN, um sie beim nächsten zyklischen Sendevorgang zu verwenden.

Ereignis-Ausgänge

- **INITO**: Bestätigt den Abschluss der Initialisierung.
- **installO**: Signalisiert, dass der Installationsvorgang abgeschlossen ist. Führt den `PGN_handle` als Ausgabewert mit.
- **UPDO**: Bestätigt, dass die Aktualisierung der Nutzdaten (`UPD`) erfolgreich verarbeitet wurde.
- **CNF**: Bestätigt, dass ein Datensatz erfolgreich gesendet wurde.
- **dataERR**: Zeigt einen Fehler im Zusammenhang mit den Nutzdaten (`Data`) an. Führt den Fehlercode `dataERRC` mit.
- **pgnERR**: Zeigt einen Fehler im Zusammenhang mit der PGN-Konfiguration oder -Verwaltung an. Führt den Fehlercode `pgnERRC` mit.

Daten-Eingänge

- **u32Pgn** (UDINT): Die zu sendende Parameter Group Number (PGN). Gültiger Bereich: 0 bis 0x3FFFF.
- **NmDestin** (isobus::pgn::ISONETEVENT_T): Definiert den Kommunikationspartner (Zieladresse) für die Nachricht.
- **u16DaSize** (UINT): Die Länge der zu sendenden Daten in Bytes. Gültiger Bereich: 0 bis 8.
- **u8Priority** (USINT): Die Priorität der Nachricht auf dem CAN-Bus (0=höchste, 7=niedrigste). Standardwert: 7.
- **u16DefRepRate** (UINT): Das zyklische Sendeintervall in Millisekunden (0 ... 0xFDFF ms). Ein Wert von 0 deaktiviert die zyklische Übertragung. Standardwert: 0.
- **Data** (isobus::pgn::CAN_MSG): Die Nutzdaten, die als PGN gesendet werden sollen.

Daten-Ausgänge

- **PGN_handle** (INT): Ein Handle (Bezeichner) für die erfolgreich installierte PGN. Im Fehlerfall wird ein ungültiger Handle-Wert (`HANDLE_UNVALID`) zurückgegeben.
- **dataERRC** (INT): Ein Fehlercode, der detaillierte Informationen zu einem aufgetretenen `dataERR` liefert.
- **pgnERRC** (INT): Ein Fehlercode, der detaillierte Informationen zu einem aufgetretenen `pgnERR` liefert.

Adapter

Dieser Funktionsblock verwendet keine Adapter-Schnittstellen.

Funktionsweise

1. **Initialisierung**: Nach dem Eintreffen des `INIT`-Ereignisses wird der Block betriebsbereit gemacht. Die Bestätigung erfolgt über `INITO`.
2. **Installation**: Das `install`-Ereignis löst die Konfiguration einer neuen zyklischen PGN-Übertragung aus. Alle zugehörigen Parameter (`u32Pgn`, `NmDestin`, etc.) werden ausgewertet und intern gespeichert. Bei Erfolg wird ein `PGN_handle` generiert und mit dem `installO`-Ereignis zurückgegeben. Bei Fehlern (z.B. ungültige Parameter) wird das `pgnERR`-Ereignis ausgelöst.
3. **Zyklische Übertragung**: Ist die PGN installiert und `u16DefRepRate` > 0, beginnt der Block automatisch, die in `Data` hinterlegten Nutzdaten in dem definierten Intervall zu senden. Jeder erfolgreiche Sendevorgang wird durch das `CNF`-Ereignis quittiert.
4. **Datenaktualisierung**: Das `UPD`-Ereignis erlaubt es, die Nutzdaten (`Data`) für die aktive PGN-Übertragung zu ändern. Die neuen Daten werden bei der nächsten zyklischen Aussendung verwendet. Der Empfang der neuen Daten wird mit `UPDO` bestätigt. Fehlerhafte Daten führen zum Auslösen von `dataERR`.

Technische Besonderheiten

- Der Block ist für die Verwendung in ISOBUS-Umgebungen (Landtechnik) vorgesehen und nutzt spezifische ISOBUS-Datentypen (`ISONETEVENT_T`, `CAN_MSG`).
- Die zyklische Übertragung kann durch Setzen von `u16DefRepRate` auf 0 deaktiviert werden, was einen bedarfsgesteuerten Betrieb ermöglicht.
- Die Fehlerbehandlung ist über dedizierte Ereignisse (`dataERR`, `pgnERR`) strukturiert, was eine robuste Integration in übergeordnete Steuerungen ermöglicht.
- Die Rückgabe eines `PGN_handle` erlaubt die Verwaltung mehrerer installierter PGNs innerhalb eines Systems.

Zustandsübersicht

1. **Nicht initialisiert**: Block ist nach dem Start im Ruhezustand.
2. **Initialisiert / Bereit**: Nach `INIT`/`INITO`. Block wartet auf Konfigurations- oder Steuerereignisse.
3. **PGN installiert**: Nach erfolgreichem `install`/`installO`. Die PGN ist konfiguriert und der interne Zustand für die (zyklische) Übertragung ist aktiv.
4. **Senden aktiv**: Wenn `u16DefRepRate` > 0, sendet der Block zyklisch Daten und feuert bei jedem Erfolg `CNF`.
5. **Fehlerzustand**: Bei Auftreten eines Fehlers (`pgnERR` oder `dataERR`). Der Block kann je nach Fehlerart in einem wartenden Zustand sein oder versuchen, die Operation erneut durchzuführen (soweit im Algorithmus implementiert).

Anwendungsszenarien

- **ISOBUS-konforme Maschinensteuerung**: Zyklisches Senden von Maschinendaten (z.B. Drehzahl, Druck, Position) von einem Steuergerät (ECU) an ein Terminal oder andere Teilnehmer im Netzwerk.
- **Diagnose- und Überwachungssysteme**: Regelmäßige Übertragung von Status- und Betriebsparametern zu Überwachungszwecken.
- **Implementierung von ISOBUS-„Fast-Packet“-Protokollen**: Für PGNs, die mehr als 8 Byte Daten transportieren und daher in mehreren CAN-Botschaften übertragen werden müssen (unterstützt durch den `CAN_MSG`-Datentyp).

⚖️ Vergleich mit ähnlichen Bausteinen

- **Gegenüber einfachen `E_CYC`-Blöcken**: `AlPgnTxNew8Bcycl` ist spezialisiert auf ISOBUS-PGNs und bietet integrierte Handhabung von Priorität, Zieladressierung und Fehlermanagement, während ein generischer zyklischer Event-Generator (`E_CYC`) nur Timing bereitstellt.
- **Gegenüber generischen CAN-Sendeblöcken**: Dieser Block abstrahiert die niedrigleveligen CAN-Details (Identifier-Berechnung, Datenrahmen) und arbeitet direkt auf der logischeren PGN-Ebene gemäß ISOBUS-Standard.

🛠️ Zugehörige Übungen

- Uebung_127

Fazit

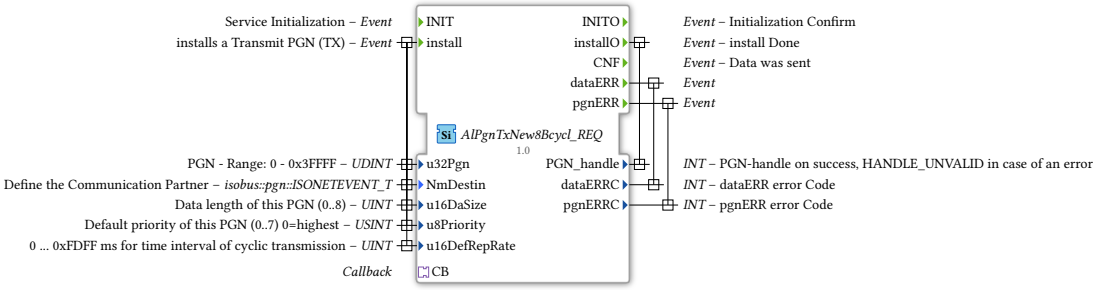
Der `AlPgnTxNew8Bcycl` ist ein spezialisierter und leistungsstarker Funktionsblock für die zyklische Datenkommunikation in ISOBUS-Netzwerken. Durch seine klare Schnittstelle, die umfassende Konfigurierbarkeit und die integrierte Fehlerrückmeldung eignet er sich ideal für die zuverlässige Implementierung von Sendediensten in komplexen, verteilten Steuerungssystemen der Landtechnik.

AlPgnTxNew8Bcycl_REQ

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `AlPgnTxNew8Bcycl_REQ` dient zur zyklischen Übertragung von Daten über ein ISOBUS-Netzwerk. Sein Hauptzweck ist die Installation und Verwaltung eines Parameter Group Number (PGN)-Transmit-Objekts (TX), das Daten in einem definierten Zeitintervall sendet. Ein zentrales Merkmal ist die Integration eines Callback-Adapters, der eine flexible Datenbereitstellung ermöglicht.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisiert den Funktionsblock.
- **install**: Startet den Installationsvorgang für ein neues zyklisches Transmit-PGN. Löst die Konfiguration mit den anliegenden Daten aus.

Ereignis-Ausgänge

- **INITO**: Bestätigt den erfolgreichen Abschluss der Initialisierung.
- **installO**: Bestätigt den Abschluss der PGN-Installation. Liefert den generierten `PGN_handle`.
- **CNF**: Wird ausgelöst, wenn Daten erfolgreich gesendet wurden.
- **dataERR**: Zeigt einen Fehler im Zusammenhang mit den zu sendenden Daten an. Liefert einen Fehlercode.
- **pgnERR**: Zeigt einen Fehler im Zusammenhang mit der PGN-Konfiguration oder -Verwaltung an. Liefert einen Fehlercode.

Daten-Eingänge

- **u32Pgn** (UDINT): Die zu verwendende Parameter Group Number (PGN). Gültiger Bereich: 0 bis 0x3FFFF.
- **NmDestin** (isobus::pgn::ISONETEVENT_T): Definiert den Kommunikationspartner (Zieladresse) für die Übertragung.
- **u16DaSize** (UINT): Die Länge der zu übertragenden Nutzdaten in Bytes. Maximal 8 Bytes (0..8).
- **u8Priority** (USINT): Die Priorität der Nachricht auf dem Bus (0..7), wobei 0 die höchste Priorität ist. Standardwert: 7 (niedrigste).
- **u16DefRepRate** (UINT): Das zyklische Sendeintervall in Millisekunden (0 ... 0xFDFF ms). Ein Wert von 0 deaktiviert die zyklische Übertragung. Standardwert: 0.

Daten-Ausgänge

- **PGN_handle** (INT): Ein eindeutiges Handle, das die installierte PGN-Instanz identifiziert. Im Fehlerfall enthält es den Wert `HANDLE_UNVALID`.
- **dataERRC** (INT): Fehlercode, der bei Auslösen des `dataERR`-Ereignisses gesetzt wird.
- **pgnERRC** (INT): Fehlercode, der bei Auslösen des `pgnERR`-Ereignisses gesetzt wird.

Adapter

- **CB** (Typ: `isobus::pgn::tx::Callback`): Ein Socket-Adapter, der eine Callback-Schnittstelle bereitstellt. Über diesen Adapter fordert der Funktionsblock bei jedem zyklischen Sendevorgang die aktuellen Nutzdaten vom übergeordneten Steuerungsalgorithmus an.

Funktionsweise

1. **Initialisierung**: Das `INIT`-Ereignis bereitet den Funktionsblock für den Betrieb vor. Nach Abschluss wird `INITO` ausgelöst.
2. **PGN-Installation**: Das `install`-Ereignis triggert die Konfiguration einer neuen zyklischen Sende-PGN. Die an den Daten-Eingängen (`u32Pgn`, `NmDestin`, etc.) anliegenden Werte werden verwendet, um die PGN im ISOBUS-Stack zu registrieren.
3. **Handle-Rückgabe**: Bei erfolgreicher Installation wird das `installO`-Ereignis ausgelöst und der generierte `PGN_handle` am Datenausgang bereitgestellt. Dieser Handle muss für spätere Operationen (z.B. Deinstallation, Änderung) gespeichert werden.
4. **Zyklischer Sendebetrieb**: Ist `u16DefRepRate` > 0, beginnt der Funktionsblock, Daten im definierten Intervall zu senden.
   - Vor jedem Sendevorgang fordert der Block über den `CB`-Adapter die aktuellen Nutzdaten an.
   - Nach erfolgreicher Übertragung wird das `CNF`-Ereignis ausgelöst.
5. **Fehlerbehandlung**: Tritt ein Fehler auf (z.B. ungültige Konfiguration, Kommunikationsproblem), wird entweder `dataERR` oder `pgnERR` ausgelöst und der entsprechende Fehlercode gesetzt.

Technische Besonderheiten

- **Datenlänge**: Unterstützt die Übertragung von maximal 8 Byte Nutzdaten pro PGN, was einer typischen ISOBUS-Datenlänge entspricht.
- **Callback-Mechanismus**: Die Nutzdaten werden nicht intern gespeichert, sondern bei Bedarf dynamisch über den Adapter angefordert. Dies ermöglicht eine effiziente und aktuelle Datenbereitstellung.
- **Fehlerhandling**: Separate Fehlerereignisse für Daten- (`dataERR`) und PGN-bezogene (`pgnERR`) Probleme erlauben eine differenzierte Fehlerdiagnose.
- **Initialwert**: Die Priorität (`u8Priority`) und das Sendeintervall (`u16DefRepRate`) haben definierte Initialwerte (7 bzw. 0).

Zustandsübersicht

Der Funktionsblock durchläuft implizit folgende Hauptzustände:

1. **Nicht initialisiert**: Der Block ist nach dem Start inaktiv.
2. **Initialisiert (Bereit)**: Nach erfolgreichem `INIT`/`INITO` wartet der Block auf eine Installationsanforderung.
3. **PGN installiert (Aktiv)**: Nach erfolgreichem `install`/`installO` ist die PGN konfiguriert. Bei `u16DefRepRate` > 0 sendet der Block zyklisch Daten und löst dabei `CNF` aus. Er reagiert weiterhin auf Fehlerbedingungen.

Anwendungsszenarien

- **Zyklische Statusmeldungen**: Regelmäßiges Senden von Maschinenstatusdaten (z.B. Drehzahl, Temperatur, Betriebsstunden) an ein Display oder ein übergeordnetes Managementsystem.
- **Implementierung von ISOBUS-„Fast-Packet"-Protokollen**: Für PGNs, die mehr als 8 Byte Daten umfassen und über mehrere CAN-Telegramme verteilt werden, kann dieser Block die zyklische Übertragung der einzelnen Pakete steuern.
- **Datenlogging**: Zyklische Übertragung von Prozessdaten zu einem Datensammler oder Gateway.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Vs. Einmal-Sende-Blöcke (z.B. `AlPgnTx_REQ`)**: Dieser Block ist für **wiederholtes, periodisches** Senden ausgelegt, während einfache TX-Blöcke typischerweise ein einmaliges Senden pro Auslöseereignis durchführen.
- **Vs. Blöcke mit interner Datenspeicherung**: Die Verwendung eines **Callback-Adapters** unterscheidet diesen Block von solchen, bei denen die Daten direkt an einem Eingang bereitgestellt werden. Dies macht ihn flexibler, wenn sich die Daten häufig ändern oder aus komplexen Quellen stammen.

🛠️ Zugehörige Übungen

- Uebung_126

- Uebung_126b
- Uebung_126b2

Fazit

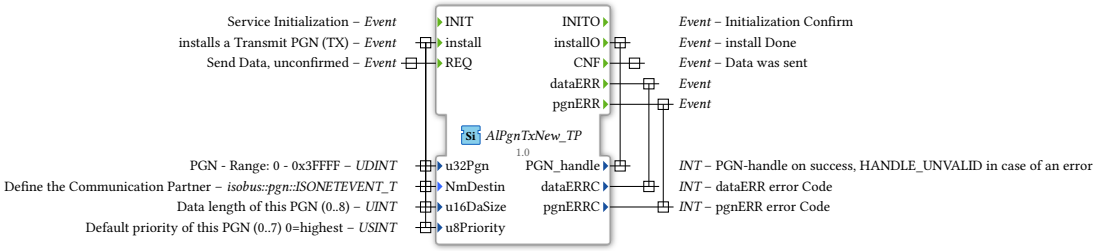
Der `AlPgnTxNew8Bcycl_REQ` ist ein spezialisierter Funktionsblock für zuverlässige, zyklische Datenkommunikation in ISOBUS-Umgebungen. Seine Stärken liegen in der klaren Trennung von Konfiguration (`install`), dynamischer Datenbeschaffung (Callback-Adapter) und robustem Fehlerfeedback. Er ist ideal geeignet für Anwendungen, die regelmäßige Statusupdates oder Prozessdatenstreams erfordern.

AlPgnTxNew_TP

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **AlPgnTxNew_TP** dient der Übertragung von Daten über den ISOBUS/CAN-Bus basierend auf Parameter Group Numbers (PGN). Er ermöglicht die Registrierung einer spezifischen PGN und das anschließende versenden von Datenpaketen, wann immer ein lokales Anforderungsereignis (REQ) auftritt. Der Baustein ist Teil des Pakets `isobus::pgn::tx` und ist speziell für die Handhabung von Transportprotokollen oder generischen PGN-Übertragungen ausgelegt.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (Type: `EInit`): Initialisierung des Dienstes. Setzt den Baustein in den Ausgangszustand.
- **install** (Type: `Event`): Installiert bzw. registriert eine Sendere-PGN (Transmit PGN). Hierbei werden Parameter wie PGN-Nummer, Ziel, Datengröße und Priorität übernommen.
- **REQ** (Type: `Event`): Fordert das Senden von Daten an (unbestätigter Sendevorgang). Nutzt den im `Data`-Eingang bereitgestellten Puffer.

Ereignis-Ausgänge

- **INITO** (Type: `EInit`): Bestätigung der Initialisierung.
- **installO** (Type: `Event`): Bestätigung, dass der Installationsvorgang abgeschlossen wurde. Liefert das `PGN_handle` zurück.
- **CNF** (Type: `Event`): Bestätigung, dass die Daten erfolgreich gesendet wurden (Confirmation).
- **dataERR** (Type: `Event`): Signalisiert einen Fehler bei der Datenübertragung oder beim Zugriff auf die Daten.
- **pgnERR** (Type: `Event`): Signalisiert einen Fehler bezüglich der PGN-Konfiguration oder Registrierung.

Daten-Eingänge

- **u32Pgn** (Type: `UDINT`): Die zu sendende Parameter Group Number (PGN). Der Bereich liegt zwischen 0 und 0x3FFFF.
- **NmDestin** (Type: `isobus::pgn::ISONETEVENT_T`): Definiert den Kommunikationspartner bzw. das Ziel im Netzwerk (Network Management Destination).
- **u16DaSize** (Type: `UINT`): Die Länge der Daten für diese PGN in Bytes (typischerweise 0 bis 8 für Standard-CAN, kann aber je nach Protokoll variieren).
- **u8Priority** (Type: `USINT`): Die Standardpriorität dieser PGN (0 bis 7), wobei 0 die höchste Priorität darstellt. Standardwert ist 7.

Daten-Ausgänge

- **PGN_handle** (Type: `INT`): Rückgabewert nach der Installation. Ein gültiges Handle bei Erfolg, oder `HANDLE_UNVALID` im Fehlerfall.

- **dataERRC** (Type: `INT`): Fehlercode, der ausgegeben wird, wenn das Ereignis `dataERR` ausgelöst wird.
- **pgnERRC** (Type: `INT`): Fehlercode, der ausgegeben wird, wenn das Ereignis `pgnERR` ausgelöst wird.

InOut-Variablen

- **Data** (Type: `BYTE`, ArraySize: *): Ein Zeiger auf den Datenpuffer (Byte-Array variabler Größe). Hier werden die zu sendenden Nutzdaten abgelegt. Durch die Deklaration als `InOut` wird eine effiziente Speicherverwaltung ohne unnötige Kopieroperationen ermöglicht.

Adapter

- Dieser Funktionsbaustein verwendet keine expliziten Adapter-Schnittstellen.

Funktionsweise

Der Baustein arbeitet in zwei wesentlichen Phasen:

1. **Konfiguration (Installation):**
   - Zunächst muss der Baustein über `INIT` initialisiert werden.
   - Anschließend wird über das Ereignis `install` die PGN konfiguriert. Dabei werden die PGN-ID (`u32Pgn`), das Ziel (`NmDestin`), die Größe (`u16DaSize`) und die Priorität (`u8Priority`) festgelegt.
   - Nach erfolgreicher Installation wird das Ereignis `installO` ausgelöst und ein `PGN_handle` bereitgestellt. Dieses Handle repräsentiert die registrierte PGN im System.
2. **Datenübertragung:**
   - Sobald Daten gesendet werden sollen, müssen diese im `Data`-Array bereitgestellt werden.
   - Das Ereignis `REQ` triggert den Sendevorgang.
   - Bei erfolgreichem Versand wird `CNF` ausgelöst.
   - Treten Probleme auf (z.B. ungültige Datenlänge oder Busfehler), wird `dataERR` mit einem entsprechenden Fehlercode `dataERRC` ausgelöst.

Technische Besonderheiten

- **Variable Array-Größe:** Der `Data`-Eingang ist als `InOut`-Variable mit `ArraySize="*"` definiert. Dies bedeutet, dass der Baustein mit Datenpuffern unterschiedlicher Größe arbeiten kann, was ihn flexibel für verschiedene PGN-Typen macht.
- **Fehlertrennung:** Der Baustein unterscheidet explizit zwischen Fehlern bei der PGN-Einrichtung (`pgnERR`) und Fehlern bei der Laufzeit-Datenübertragung (`dataERR`). Dies erleichtert das Debugging erheblich.
- **ISOBUS-Typen:** Der Baustein importiert spezifische ISOBUS-Datentypen (`isobus::pgn::ISONETEVENT_T`), was eine enge Integration in entsprechende ISOBUS-Stacks voraussetzt.

Zustandsübersicht

Der Baustein besitzt intern implizite Zustände, die durch die Verfügbarkeit eines gültigen `PGN_handle` definiert sind:

- **Nicht Initialisiert:** Vor `INIT`.
- **Initialisiert, nicht Installiert:** Nach `INIT`, aber vor erfolgreichem `install`. Sendeanforderungen (`REQ`) würden hier fehlschlagen.
- **Installiert / Betriebsbereit:** Ein gültiges Handle wurde erzeugt. Daten können via `REQ` gesendet werden.

Anwendungsszenarien

- **ISOBUS-Geräteentwicklung:** Implementierung von Virtual Terminals oder Task Controllern, die spezifische PGNs senden müssen.
- **Sensor-Datenübertragung:** Ein Sensor-Modul sendet zyklisch oder ereignisbasiert Messwerte an den CAN-Bus.
- **Steuerbefehle:** Senden von Steuerungsnachrichten an Aktoren im Netzwerk (z.B. Ventile oder Motoren) mit definierter Priorität.

⚖️ Vergleich mit ähnlichen Bausteinen

- **AlPgnTx vs. AlPgnTxNew_TP:** Während ältere oder einfachere Versionen eventuell nur statische PGNs unterstützen, deutet der Zusatz „New_TP“ auf eine überarbeitete Version hin, die möglicherweise

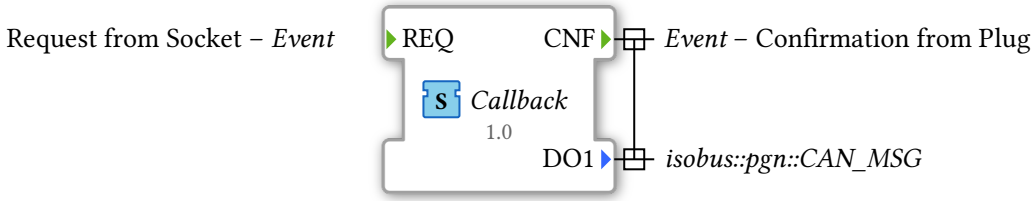
Transportprotokolle (TP) besser unterstützt oder eine modernere Speicherverwaltung (via `InOut` Variablen) nutzt.
- **Standard CAN_WRITE:** Im Vergleich zu einem generischen `CAN_WRITE`-Baustein abstrahiert `AlPgnTxNew_TP` die Komplexität der PGN-Verwaltung (Priorität, Datenlänge, Handle-Management) und ist spezifischer auf das J1939/ISOBUS-Protokoll zugeschnitten.

🛠️ Zugehörige Übungen
- Uebung_128b

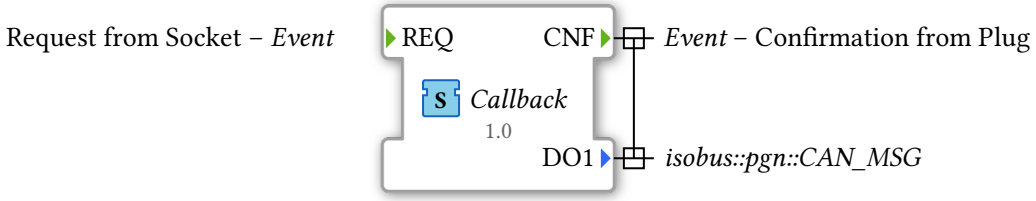
Fazit

Der **AlPgnTxNew_TP** ist ein spezialisierter und robuster Funktionsbaustein für das Senden von Nachrichten in ISOBUS-Netzwerken. Durch die Trennung von Konfiguration (`install`) und Übertragung (`REQ`) sowie die Nutzung von Referenz-Datenpuffern (`InOut`) eignet er sich hervorragend für ressourceneffiziente Steuerungsanwendungen, die eine präzise Kontrolle über PGN-Parameter erfordern.

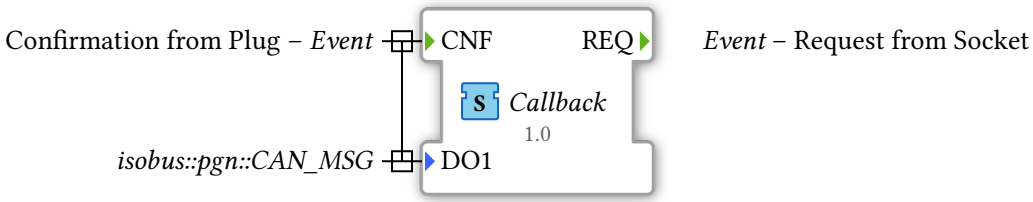
Callback



Plug (Adapter-Ausgang)



Socket (Adapter-Eingang)



Message Callback

Interface

Event Inputs

Name	Comment	With
REQ	Request from Socket	

Event Outputs

Name	Comment	With
CNF	Confirmation from Plug	DO1

Output Vars

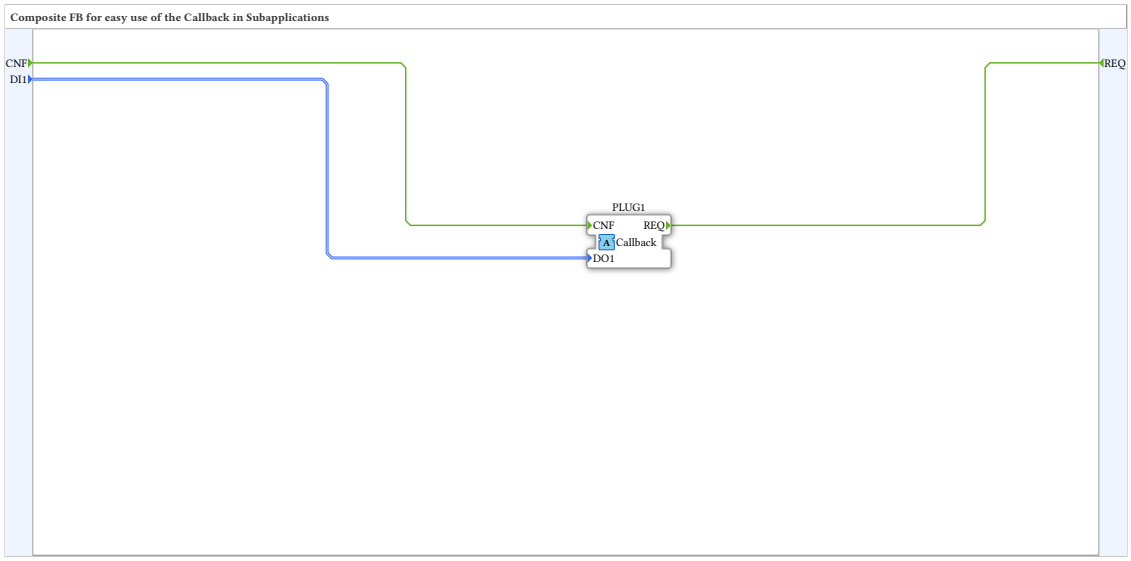
Name	Type	Comment

DO1	isobus::pgn::CAN_MSG	

🛠️ Zugehörige Übungen

- SIN_TO_PCAN_Callback_DWord
- SIN_TO_PCAN_Callback_Byte
- Uebung_12x_sub

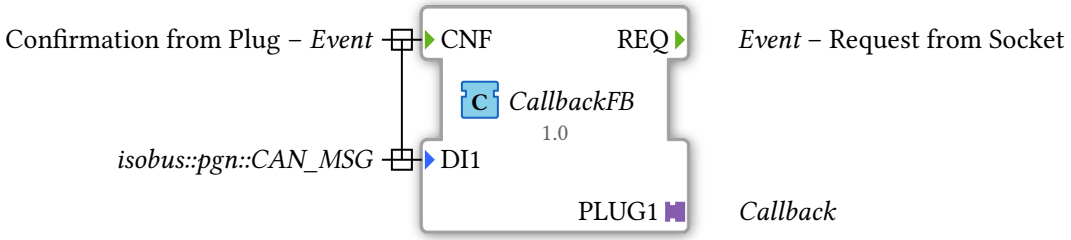
CallbackFB



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `CallbackFB` ist ein Composite-Funktionsblock, der die Verwendung des Callback-Mechanismus in Subapplikationen vereinfacht. Er dient als Wrapper-Baustein, um eine standardisierte Schnittstelle für die asynchrone Kommunikation über einen Callback-Adapter bereitzustellen. Sein primärer Zweck ist die Entkopplung und strukturierte Handhabung von Bestätigungsnachrichten (`CNF`) und Anfragen (`REQ`) innerhalb eines Funktionsblocknetzwerks.



Schnittstellenstruktur

Ereignis-Eingänge

- **`CNF`**: Bestätigungsereignis (Confirmation). Dieses Ereignis wird vom angeschlossenen Plug empfangen und signalisiert den Abschluss einer Anfrage oder das Eintreffen von Daten.

Ereignis-Ausgänge

- **`REQ`**: Anfrageereignis (Request). Dieses Ereignis wird vom internen Socket des Adapters ausgelöst und signalisiert eine ausstehende Aktion oder Datenanforderung.

Daten-Eingänge

- **`DI1`**: Dateneingang vom Typ `isobus::pgn::CAN_MSG`. Dieser Eingang transportiert die CAN-Nachrichtendaten, die mit dem eingehenden `CNF`-Ereignis assoziiert sind.

Daten-Ausgänge

- *(Keine direkten Datenausgänge am FB definiert. Die Daten werden über den Adapter weitergeleitet.)*

Adapter

- **`PLUG1`**: Ein Plug-Adapter vom Typ `isobus::pgn::tx::Callback`. Dieser Adapter stellt die Verbindung zu einem entsprechenden Socket-Adapter in einem anderen Funktionsblock her und realisiert das Callback-Muster.

Funktionsweise

Der `CallbackFB` fungiert als transparente Vermittlungsinstanz. Sein internes Netzwerk (FBNetwork) verbindet die öffentlichen Schnittstellen des Composite-FB direkt mit den Pins des integrierten `Callback`-Adapters (`PLUG1`).

1. Ein eingehendes `CNF`-Ereignis wird direkt an den `CNF`-Eingang des `PLUG1`-Adapters weitergeleitet.
2. Die mit `CNF` verbundenen Daten (`DI1` vom Typ `isobus::pgn::CAN_MSG`) werden parallel an den Dateneingang `DO1` des `PLUG1`-Adapters übergeben.
3. Wenn der interne Socket des `PLUG1`-Adapters ein `REQ`-Ereignis auslöst, wird dieses direkt an den öffentlichen `REQ`-Ereignisausgang des `CallbackFB` weitergeleitet.

Der Baustein selbst führt keine zusätzliche Logik oder Datenverarbeitung durch. Seine Funktion besteht ausschließlich in der strukturierten Bündelung und Durchreichung von Ereignissen und Daten.

Technische Besonderheiten

- **Typisierung**: Der Dateneingang `DI1` verwendet den spezifischen Datentyp `isobus::pgn::CAN_MSG`, was auf eine Anwendung im Kontext von ISOBUS- und CAN-basierten Agrarsystemen hindeutet.
- **Adapter-Pattern**: Die gesamte Funktionalität basiert auf dem 61499-Adapterkonzept. Der FB kapselt einen `Plug` und benötigt für den Betrieb einen korrespondierenden `Socket` des Typs `isobus::pgn::tx::Callback` in einem anderen Baustein.
- **Transparenz**: Als Composite-FB ohne zusätzliche Algorithmen ist sein Verhalten vollständig durch das interne Verbindungsnetzwerk definiert und somit vorhersagbar.

Zustandsübersicht

Da es sich um einen rein durchschleifenden Composite-Baustein ohne interne Zustandsmaschine (Basic FB) oder zyklisches Verhalten (Service Interface FB) handelt, besitzt der `CallbackFB` keinen eigenen Zustandsautomaten. Sein „Zustand“ wird durch den Zustand des verbundenen Adapter-Paares (`PLUG1` und sein korrespondierender Socket) bestimmt.

Anwendungsszenarien

- **Strukturierung von Subapplikationen**: In komplexen Subapplikationen kann der `CallbackFB` verwendet werden, um eine saubere und wiederverwendbare Schnittstelle für Callback-Kommunikation bereitzustellen.
- **Entkopplung von Komponenten**: Er ermöglicht die lose Kopplung zwischen einem Dienst-Anbieter (der den Socket bereitstellt) und einem Dienst-Nutzer (der diesen FB einsetzt), was die Wartbarkeit und Testbarkeit erhöht.
- **ISOBUS-Kommunikation**: Spezifische Anwendung in Steuerungssystemen für landwirtschaftliche Maschinen, um die Übertragung von CAN-Nachrichten (PGNs) asynchron zu bestätigen oder anzufordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Direkte Adapterverbindung**: Anstatt einen `Callback`-Adapter direkt in ein Netzwerk einzubinden, bietet der `CallbackFB` eine gebündelte, benannte Schnittstelle (`CNF`/`REQ`/`DI1`), die in manchen Entwürfen lesbarer sein kann.
- **Einfache E_CLASS vs. CallbackFB**: Ein einfacher E_CLASS-Baustein (Event Splitter/Merger) könnte ähnliche Ereigniswegleitungen durchführen, jedoch nicht die typsichere Datenverbindung für `isobus::pgn::CAN_MSG` zusammen mit dem standardisierten Adapter-Interface bereitstellen.
- **Generische Wrapper-FBs**: Im Gegensatz zu generischen Wrapper-Bausteinen ist der `CallbackFB` auf einen spezifischen Datentyp und Adapter spezialisiert, was Typsicherheit und klare Semantik gewährleistet.

🛠️ Zugehörige Übungen

- SIN_TO_PCAN_Callback_DWord
- SIN_TO_PCAN_Callback_Byte
- Uebung_12x_sub

Fazit

Der `CallbackFB` ist ein nützliches Hilfskonstrukt für die strukturierte Anwendung des Adapter-Patterns in IEC 61499, speziell im ISOBUS-Umfeld. Er reduziert die Komplexität im Hauptnetzwerk, indem er die Verbindungslogik für Callback-Kommunikation in einen einzigen, wiederverwendbaren Composite-Baustein auslagert. Seine Stärke liegt in der Einfachheit und Klarheit, während er selbst keine aktive Logik implementiert. Er ist die ideale Wahl, wenn eine standardisierte und typsichere Callback-Schnittstelle in einer Subapplikation benötigt wird.

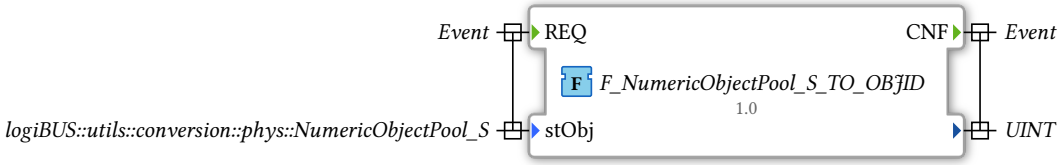
2.1.13.6 Utils

Conversion

Numerischer Index (conversion)

Hier finden Sie alle Übungen des Arbeitsbereichs `conversion` in aufsteigender Reihenfolge.

F_NumericObjectPool_S_TO_OBJID



• – ◦ • – ◦ • – ◦ •

Einleitung

`F_NumericObjectPool_S_TO_OBJID` extrahiert die ISOBUS-Object-ID (`u16ObjId`) aus einer `NumericObjectPool_S`-Struktur (`logiBUS::utils::conversion::phys::NumericObjectPool_S`).

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Ausführungsanforderung (assoziiert mit `stObj`)

Ereignis-Ausgänge

- **CNF**: Bestätigung (assoziiert mit dem Funktionsergebnis)

Daten-Eingänge

- **stObj** (`logiBUS::utils::conversion::phys::NumericObjectPool_S`): Object-Pool-Struktur einer VT-Zahlenvariable

Daten-Ausgänge

- **(Funktionsergebnis)** (UINT): die Object ID `stObj.u16ObjId`

Adapter

Es werden keine Adapter verwendet.

Funktionsweise

Die Funktion besteht aus einer einzigen Zuweisung: `F_NumericObjectPool_S_TO_OBJID := stObj.u16ObjId;`. Bei `REQ` wird die Object ID aus der Struktur gelesen und mit `CNF` ausgegeben.

Technische Besonderheiten

- Reine Funktion ohne Zustand, Version 1.0 (2026-10-11)
- Der Ausgang besitzt keinen eigenen Namen, sondern ist der Rückgabewert der Funktion

- Die Funktion liegt im Paket `logiBUS::utils::conversion::phys`

Anwendungsszenarien

- Zugriff auf die Object ID einer VT-Zahlenvariable, wenn ein Baustein nur die einfache `UINT`-ID statt der ganzen `NumericObjectPool_S`-Struktur erwartet

Fazit

`F_NumericObjectPool_S_TO_OBJID` liefert die `u16ObjId` einer `NumericObjectPool_S`-Struktur als `UINT`.

F_PHYS_LREAL_TO_RAW

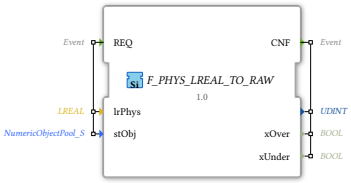


Bild nicht verfügbar

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock F_PHYS_LREAL_TO_RAW dient der Umrechnung eines physikalischen Messwerts im Datentyp LREAL in einen ISOBUS-konformen Rohwert als UDINT. Dabei werden Sättigungsgrenzen eingehalten und Über- bzw. Unterlaufzustände als boolsche Ausgänge signalisiert. Die Konvertierung folgt der ISOBUS-Formel:
Anzeigewert = (Rohwert + Offset) · Skalierung
Rohwert = UDINT(DINT(physikalischer Wert / Skalierung) – Offset)

Der Baustein eignet sich für Applikationen, die physikalische Größen in das ISOBUS-Protokoll einbinden müssen, insbesondere in der Land- und Nutzfahrzeugtechnik.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung	Mitgeführte Variablen
`REQ`	Startet die Umrechnung	`lrPhys`, `stObj`

Ereignis-Ausgänge

Ereignis	Beschreibung	Mitgeführte Variablen
`CNF`	Bestätigt die abgeschlossene Umrechnung	`xOver`, `xUnder`

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung
`lrPhys`	`LREAL`	–	Physikalischer Eingangswert (z. B. Temperatur, Druck)
`stObj`	`logiBUS::utils::conversion::phys::NumericObjectPool_S`	`(u16ObjId := 65535, r32Scale := 1.0, i32Offset := 0, u8Decimals := 0)`	Datenstruktur mit Skalier- und Offsetparametern

Daten-Ausgänge

Name	Typ	Beschreibung

(kein Name)	UDINT	Berechneter ISOBUS-Rohwert (0 … 4294967295)
xOver	BOOL	TRUE, wenn der Eingang die obere Grenze überschreitet (Überlauf)
xUnder	BOOL	TRUE, wenn der Eingang die untere Grenze unterschreitet (Unterlauf)

Adapter

Der Funktionsblock verwendet keine Adapter.

Funktionsweise

Der FB führt folgende Schritte aus:

1. **Berechnung der Grenzen:**
   - Untere Grenze: `i32Offset * r32Scale`
   - Obere Grenze: `(i32Offset + 4294967295) * r32Scale`
2. **Vergleich und Sättigung:**
   - Ist `lrPhys > obere Grenze` ⟶ Rohwert = **UDINT#4294967295**, `xOver = TRUE`.
   - Ist `lrPhys < untere Grenze` ⟶ Rohwert = **0**, `xUnder = TRUE`.
   - Andernfalls ⟶ Rohwert = `DINT_TO_UDINT( LREAL_TO_DINT( lrPhys / r32Scale ) - i32Offset )`, keine Über-/Unterlaufmeldung.
3. **Ausgabe:** Der berechnete Rohwert wird am unbenannten `UDINT`-Ausgang bereitgestellt und das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- **ISOBUS-Konformität:** Die Umrechnung folgt exakt der ISOBUS-Spezifikation (ISO 11783).
- **Sättigungsverhalten:** Kritische Bereiche werden durch die Über-/Unterlaufausgänge `xOver` und `xUnder` abgebildet, sodass Folgefunktionen (z. B. Alarmierung) reagieren können.
- **Skalierungseinheit:** Der Skalierungsfaktor `r32Scale` ist vom Typ `REAL` und wird direkt in die Berechnung einbezogen.
- **Integer-Umwandlung:** Die Konvertierung von LREAL über DINT zu UDINT erfolgt mit der in IEC 61499 üblichen Rundung (Abschneiden der Dezimalstellen).
- **Keine Zustandsautomaten:** Der FB arbeitet rein funktional ohne interne Zustandsmaschine.

Zustandsübersicht

Der Funktionsblock besitzt keine explizite Zustandsmaschine. Die Berechnung wird durch das Ereignis `REQ` angestoßen; nach einem Zyklus wird das Ergebnis am Datenausgang bereitgestellt und das Ereignis `CNF` gesendet. Die Ausführungszeit ist deterministisch und hängt nur von der Laufzeitumgebung ab.

Anwendungsszenarien

- **Landwirtschaftliche Steuergeräte:** Umwandlung von Sensorwerten (z. B. Drehmoment, Druck) in ISOBUS-Rohdaten für die Übertragung an Terminals oder Aktoren.
- **Diagnose und Überwachung:** Erkennung von Messbereichsüberschreitungen durch `xOver`/`xUnder` zur Auslösung von Warnungen.
- **Datenlogging:** Aufbereitung physikalischer Messwerte für die Archivierung im ISOBUS-Format.

Vergleich mit ähnlichen Bausteinen

Typische Alternativen sind selbstgeschriebene Rechenfunktionen oder generische Konvertierungsbausteine.
Der F_PHYS_LREAL_TO_RAW bietet folgende Vorteile:

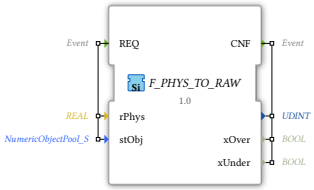
- **Vorgegebene Parametrierung** über die Struktur `NumericObjectPool_S` (ObjID, Skalierung, Offset, Dezimalstellen).
- **Integrierte Sättigung** und Grenzwertüberwachung.
- **Wiederverwendbar** in allen logiBUS-Projekten.

Nachteile gegenüber einer maßgeschneiderten Lösung sind die Bindung an die ISOBUS-Formel und die festgelegte Datenstruktur.

Fazit

Der Funktionsblock F_PHYS_LREAL_TO_RAW ist eine kompakte, normkonforme Lösung zur Umrechnung physikalischer Messwerte in das ISOBUS-Rohdatenformat. Seine Sättigungslogik und die Überwachungsausgänge machen ihn robust und sicher für den Einsatz in industriellen und landwirtschaftlichen Steuerungssystemen. Die Verwendung vordefinierter Strukturparameter erleichtert die Integration und reduziert die Fehleranfälligkeit bei der Parametrierung.

F_PHYS_TO_RAW



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **F_PHYS_TO_RAW** wandelt einen physikalischen REAL-Wert in einen ISOBUS-Rohwert (UDINT) um. Dabei werden Sättigungsgrenzen beachtet und Über- bzw. Unterlaufalarme ausgegeben. Die Umrechnung basiert auf der ISOBUS-Formel: `Anzeige = (Rohwert + Offset) * Skalierung`. Die inverse Berechnung lautet: `Rohwert = UDINT(DINT(physikalischer Wert / Skalierung) - Offset)`.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung
REQ	Anforderung zur Umrechnung des physikalischen Wertes. Wird mit den Parametern `rPhys` und `stObj` ausgeführt.

Ereignis-Ausgänge

Ereignis	Beschreibung
CNF	Bestätigung, dass die Umrechnung abgeschlossen ist. Liefert den berechneten Rohwert sowie die Alarmflags `xOver` und `xUnder`.

Daten-Eingänge

Variable	Typ	Beschreibung
rPhys	REAL	Der physikalische Wert, der umgerechnet werden soll.
stObj	NumericObjectPool_S	Struktur mit den Parametern `i32Offset` (DINT) und `r32Scale` (REAL) für die ISOBUS-Formel.

Daten-Ausgänge

Variable	Typ	Beschreibung
*	UDINT	Der berechnete ISOBUS-Rohwert (Rückgabewert der Funktion). Gesättigt auf den Bereich 0 bis 4294967295.
xOver	BOOL	TRUE, wenn der physikalische Wert den oberen Grenzwert überschreitet.
xUnder	BOOL	TRUE, wenn der physikalische Wert den unteren Grenzwert unterschreitet.

Adapter

Keine.

Funktionsweise

Der FB berechnet zunächst die untere und obere Grenze:

- Untere Grenze: `lrLower = Offset * Skalierung`
- Obere Grenze: `lrUpper = (Offset + 4294967295) * Skalierung`

Anschließend wird der physikalische Wert mit diesen Grenzen verglichen:

1. **Überlauf (xOver = TRUE):** Wenn `rPhys > lrUpper`, wird der Rohwert auf den maximalen UDINT-Wert (4294967295) gesetzt.
2. **Unterlauf (xUnder = TRUE):** Wenn `rPhys < lrLower`, wird der Rohwert auf 0 gesetzt.
3. **Normalbereich:** Im gültigen Bereich wird der Rohwert nach der Formel `UDINT(DINT(rPhys / Skalierung) - Offset)` berechnet.

Die Berechnung erfolgt innerhalb des Structured-Text-Codes bei jedem REQ-Ereignis.

Technische Besonderheiten

- Der Rückgabetyp ist UDINT, die Sättigung erfolgt auf den vollen Wertebereich von 0 bis 4294967295.
- Die Offset- und Skalierungsdaten werden aus einer externen Struktur (`NumericObjectPool_S`) gelesen, die typischerweise vom ISOBUS-Objektpool bereitgestellt wird.
- Der Baustein verwendet keine internen Zustände; die Berechnung ist rein kombinatorisch und wird bei jedem REQ komplett neu durchgeführt.
- Die Konvertierung von REAL zu DINT erfolgt mit `REAL_TO_DINT`, was bei großen Werten zu Rundungsfehlern führen kann.

Zustandsübersicht

Der FB besitzt keine explizite Zustandsmaschine. Er arbeitet zustandslos auf Basis der aktuellen Eingänge. Bei jedem REQ-Ereignis wird die Ausgabe neu berechnet.

Anwendungsszenarien

- **ISOBUS-Steuergeräte:** Umrechnung von Sensormesswerten (z. B. Temperatur, Druck) in den ISOBUS-konformen Rohdatenwert für die Anzeige in einem Terminal.
- **Landwirtschaftliche Maschinensteuerung:** Konvertierung von physikalischen Größen, die mit einer spezifischen Skalierung und Offset im ISOBUS-Diagnoseprotokoll dargestellt werden.
- **Test- und Simulationsumgebungen:** Erzeugung von ISOBUS-Rohwerten aus simulierten physikalischen Werten.

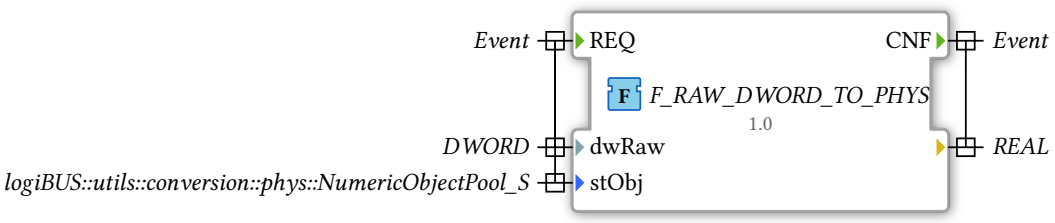
Vergleich mit ähnlichen Bausteinen

Bausteine wie `F_RAW_TO_PHYS` führen die inverse Umrechnung durch (Rohwert → physikalischer Wert). Während dieser Baustein die physikalische Größe in den Rohwert wandelt, bietet `F_PHYS_TO_RAW` zusätzlich die Sättigung und Alarmsignalisierung für Über- und Unterlauf. Im Gegensatz zu allgemeinen Arithmetikbausteinen ist er speziell auf die ISOBUS-Konvention mit Offset und Skalierung ausgelegt.

Fazit

Der Funktionsblock `F_PHYS_TO_RAW` ist ein spezialisierter Baustein für die ISOBUS-Konvertierung mit integrierter Bereichsüberwachung. Er bietet eine einfache und robuste Schnittstelle, um physikalische Messwerte in das ISOBUS-Protokoll einzubinden, und stellt sicher, dass der Rohwert immer innerhalb des gültigen Bereichs liegt. Die Alarmausgänge ermöglichen eine sofortige Fehlererkennung bei Grenzwertüberschreitungen.

F_RAW_DWORD_TO_PHYS



• – ◦ • – ◦ • – ◦ •

Einleitung

Die Funktion `F_RAW_DWORD_TO_PHYS` wandelt einen ISOBUS-Rohwert (`DWORD`) in einen physikalischen `REAL`-Wert um. Die ISOBUS-Formel lautet: `Anzeige = (Rohwert + Offset) * Skalierung`.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Umrechnung. Verbundene Daten: `dwRaw`, `stObj`.

Ereignis-Ausgänge

- **CNF**: Signalisiert die abgeschlossene Umrechnung. Das Ergebnis liegt am Daten-Ausgang an.

Daten-Eingänge

- **dwRaw** (`DWORD`): ISOBUS-Rohwert
- **stObj** (`logiBUS::utils::conversion::phys::NumericObjectPool_S`): Object-Pool-Eigenschaften der Zahlenvariable (Offset `i32Offset`, Skalierung `r32Scale`)

Daten-Ausgänge

- **(unbenannt)** (`REAL`): Physikalischer Wert (Rückgabewert der Funktion)

Adapter

Keine Adapter vorhanden.

Funktionsweise

1. `dwRaw` wird als vorzeichenlose Zahl interpretiert (`DWORD_TO_UDINT`) und zu `LINT` erweitert (`i64Raw`).
2. `stObj.i32Offset` wird ohne Umwandlung als `LINT` verwendet (`i64Offset`).
3. Das Ergebnis ist `LINT_TO_REAL(i64Raw + i64Offset) * stObj.r32Scale`.

Beispiel: `raw = 50000`, `scale = 0.01`, `offset = -100000` ⟶ `50000 + (-100000) = -50000` ⟶ `-50000 * 0.01 = -500.0`.

Technische Besonderheiten

- Implementiert als ST-Funktion (`FUNCTION ... : REAL`) im Paket `logiBUS::utils::conversion::phys`.
- Die Zwischenwerte sind `LINT`, um einen Überlauf zu vermeiden.
- Der Rohwert wird als Bitmuster ohne Vorzeichen behandelt und dann auf 64 Bit erweitert.

Anwendungsszenarien

- Umrechnung von ISOBUS-Rohwerten (z. B. aus einer VT-Zahlenvariable) in physikalische Größen

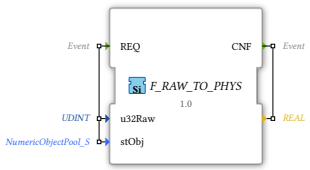
⚖️ Vergleich mit ähnlichen Bausteinen

- F_RAW_TO_PHYS und F_RAW_TO_PHYS_LREAL sind verwandte Umrechnungen; die Gegenrichtung leisten F_PHYS_TO_RAW und F_PHYS_LREAL_TO_RAW.

Fazit

`F_RAW_DWORD_TO_PHYS` rechnet einen ISOBUS-`DWORD`-Rohwert mit Offset und Skalierung aus dem Object Pool in einen `REAL`-Wert um.

F_RAW_TO_PHYS



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **F_RAW_TO_PHYS** konvertiert einen ISOBUS-Rohwert vom Typ `UDINT` in einen physikalischen `REAL`-Wert. Die Umrechnung erfolgt nach der standardisierten ISOBUS-Formel:
`display = (raw + offset) * scale`
Er wird typischerweise in landwirtschaftlichen Steuerungen eingesetzt, um Sensordaten oder Stellgliedwerte zwischen der CAN-basierten ISOBUS-Signalebene und der Applikation umzurechnen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung
REQ	Event	Startet die Umrechnung eines Rohwerts (mit `u32Raw` und `stObj`).

Ereignis-Ausgänge

Name	Typ	Beschreibung
CNF	Event	Bestätigt, dass die Berechnung abgeschlossen und der Ausgangswert gültig ist.

Daten-Eingänge

Name	Typ	Beschreibung
u32Raw	UDINT	Der rohe Integerwert (z. B. aus einem ISOBUS-Message-Objekt).
stObj	`NumericObjectPool_S`	Struktur, die den Offset (`i32Offset : DINT`) und den Skalierungsfaktor (`r32Scale : REAL`) enthält.

Daten-Ausgänge

Name	Typ	Beschreibung
(kein expliziter Name)	REAL	Der berechnete physikalische Wert, ausgegeben als Gleitkommazahl.

Adapter

Keine.

Funktionsweise

Der Baustein führt bei einem Ereignis am **REQ**-Eingang folgende Berechnung durch:

1. Der Rohwert `u32Raw` wird von `UDINT` in `LINT` konvertiert (64-Bit, um Überläufe bei der Addition zu vermeiden).
2. Der Offset `stObj.i32Offset` wird ebenfalls in `LINT` erweitert.
3. Beide Werte werden addiert: `(u32Raw + stObj.i32Offset)`.
4. Das Zwischenergebnis wird nach `REAL` konvertiert und mit dem Skalierungsfaktor `stObj.r32Scale` multipliziert.

Ergebnis: `REAL := (LINT(UDINT(u32Raw)) + LINT(stObj.i32Offset)) * stObj.r32Scale`
Beispiel (aus der Quelltext-Dokumentation):

- Rohwert = 50000
- Skalierung = 0,01
- Offset = −100000
- Rechnung: `(50000 + (-100000)) * 0,01 = (-50000) * 0,01 = -500,0`

Das Ergebnis steht nach dem Ereignis **CNF** am Datenausgang zur Verfügung.

Technische Besonderheiten

- Der Baustein verwendet **`LINT`** (64-Bit Integer) als Zwischentyp, um einen Überlauf bei der Addition von `UDINT` und `DINT` (z. B. große positive mit großen negativen Werten) sicher zu vermeiden.

- Die Ausgangsvariable hat keinen eigenen Namen – in der 4diac-IDE wird sie automatisch als `OUT` dargestellt oder kann über die Schnittstelle referenziert werden.
- Die Konfiguration von Offset und Skalierung erfolgt über eine externe Struktur (`NumericObjectPool_S`), die typischerweise aus einem Pool-Objekt stammt, das alle Parameter eines ISOBUS-Objekts bündelt.

Zustandsübersicht

Der Funktionsbaustein besitzt **keine** internen Zustände oder Speicher. Jeder Aufruf verarbeitet die aktuellen Eingabedaten und erzeugt zeitgleich den Ausgangswert. Er ist somit ein reiner **Funktionsbaustein ohne Gedächtnis** (combinatorial logic).

Anwendungsszenarien

- **ISOBUS-Steuergeräte** (Tractors, Implement) – Wandlung von CAN-Rohdaten in physikalische Einheiten (z. B. Druck, Position, Drehzahl).
- **Landwirtschaftliche Applikationen** – Umrechnung von Sensorwerten aus dem ISOBUS-Protokoll in anzeigbare Größen.
- **Test- und Simulationsumgebungen** – Nachbildung der ISOBUS-Konvertierung für Entwicklungs- und Verifikationszwecke.

Vergleich mit ähnlichen Bausteinen

- **F_RAW_TO_PHYS** ist speziell auf die ISOBUS-Formel `(raw + offset) * scale` ausgelegt und verwendet die dort üblichen Datentypen (`UDINT`, `DINT`).
- Allgemeine Konverter-Bausteine (z. B. `LREAL_TO_REAL`) bieten keine Offset-Skalierungslogik und müssen daher manuell um diese Berechnung ergänzt werden.
- Andere ISOBUS-spezifische Bausteine wie `F_PHYS_TO_RAW` kehren die Operation um (Rückrechnung von physikalischem Wert auf Rohwert).

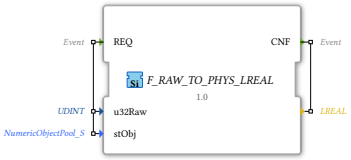
Fazit

Der **F_RAW_TO_PHYS** Baustein ist eine nützliche und sauber implementierte Komponente für die ISOBUS-Datenkonvertierung. Durch die Verwendung von 64-Bit Zwischenwerten wird Überlauf vermieden, und die Kapselung der Parameter in eine Struktur erleichtert die Wiederverwendung in Projekten, die mit dem `NumericObjectPool`-Konzept arbeiten. Er eignet sich sowohl für den produktiven Einsatz als auch für die schnelle Prototypenerstellung im Bereich landwirtschaftlicher Steuerungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

F_RAW_TO_PHYS_LREAL



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein `F_RAW_TO_PHYS_LREAL` wandelt einen rohen ISOBUS-Integerwert (UDINT) in einen physikalischen Gleitkommawert (LREAL) um.
Die Umrechnung folgt der standardisierten ISOBUS-Formel:
`display = (raw + offset) * scale`
Der Baustein ist für den Einsatz in landwirtschaftlichen Steuerungssystemen (ISOBUS) optimiert und verwendet die bereitgestellten Parameter aus einem Pool von numerischen Objekten.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Mit Variablen
REQ	Event	u32Raw, stObj

Der Baustein wird durch ein Ereignis am Eingang `REQ` gestartet.

Ereignis-Ausgänge

Name	Typ	Mit Variablen
CNF	Event	(keine)

Nach erfolgreicher Berechnung wird das Ergebnis über das Ereignis `CNF` quittiert.

Daten-Eingänge

Name	Typ	Beschreibung
u32Raw	UDINT	Rohwert aus dem ISOBUS-Bus (0 … 4.294.967.295)
stObj	NumericObjectPool_S	Struktur mit den Umrechnungsparametern: `i32Offset` (DINT) und `r32Scale` (REAL)

Daten-Ausgänge

Name	Typ	Beschreibung
(kein Name)	LREAL	Berechneter physikalischer Wert (ieee-doppelte Genauigkeit)

Das Ausgangssignal trägt keinen expliziten Namen, stellt aber den direkt berechneten LREAL-Wert dar.

Adapter

Keine.

Funktionsweise

Der Baustein wird über das Ereignis `REQ` aktiviert. Dabei werden die Eingangsdaten eingelesen und die folgende Berechnung durchgeführt:

1. Der 32-Bit-Rohwert `u32Raw` (UDINT) wird vorzeichenlos in einen 64-Bit-Integer (LINT) konvertiert.
2. Der Offset-Wert `stObj.i32Offset` (DINT) wird ebenfalls in LINT konvertiert und zum Rohwert addiert.
3. Die Summe (LINT) wird in einen LREAL umgewandelt.
4. Das Zwischenergebnis wird mit dem Skalierungsfaktor `stObj.r32Scale` (REAL) multipliziert.
5. Das Endergebnis steht als LREAL auf dem Ausgang bereit.
6. Anschließend wird das Ereignis `CNF` ausgelöst.

Die Typkonvertierung stellt sicher, dass große UDINT-Werte (z. B. 4.000.000.000) ohne Informationsverlust verarbeitet werden können.

Technische Besonderheiten

- **Erweiterte Genauigkeit**: Der Zwischenschritt über LINT vermeidet Überläufe in der Addition bei großen Rohwerten und negativen Offsets.
- **ISOBUS-Konformität**: Die Formel `display = (raw + offset) · scale` entspricht der ISOBUS-Norm ISO 11783.
- **Parametrisierung durch Struktur**: Alle Umrechnungskoeffizienten werden über die Struktur `NumericObjectPool_S` bereitgestellt, sodass der Baustein universell für verschiedene Sensoren und Geräte einsetzbar ist.
- **Single-Cycle-Verarbeitung**: Die Berechnung erfolgt ohne Zustandsspeicherung innerhalb eines einzigen Zyklus.

Zustandsübersicht

Der Baustein besitzt keinen internen Zustandsautomaten, da es sich um eine reine Transformationsfunktion handelt. Er durchläuft folgenden Ablauf:

1. Warten auf `REQ` (Ruhezustand)
2. Bei `REQ` ⟶ Berechnung ⟶ Ausgabe des Ergebnisses ⟶ `CNF` auslösen ⟶ zurück in Ruhezustand

Anwendungsszenarien

- **Landtechnik**: Umrechnung von ISOBUS-Rohwerten (z. B. Drehzahlen, Drücke, Durchflüsse) in physikalische Einheiten wie °/s, bar oder l/min.
- **Fahrzeugsteuerung**: Verarbeitung von CAN-Daten aus Traktoren, Erntemaschinen oder Applikationssystemen.
- **Simulation**: Nachbildung von ISOBUS-Sensordaten in Testumgebungen, bei denen Rohwerte aus einer Datenbank mit unterschiedlichen Skalierungen verarbeitet werden.

Vergleich mit ähnlichen Bausteinen

Baustein	Datentyp Eingang	Datentyp Ausgang	Besonderheit
`F_RAW_TO_PHYS_LREAL`	UDINT	LREAL	Doppelte Genauigkeit, LINT-Zwischenschritt
`F_RAW_TO_PHYS_REAL`	UDINT	REAL	Einfache Genauigkeit, geringere Rechenlast
`F_RAW_TO_PHYS_LINT`	UDINT	LINT	Ausgabe als Integer, ohne Skalierung
`F_RAW_TO_PHYS_INT`	UINT	REAL	Nur positive Rohwerte, geringerer Wertebereich

Der vorliegende Baustein bietet die höchste Präzision und ist für Anwendungen empfohlen, die große Wertebereiche oder feine Abstufungen erfordern.

Fazit

`F_RAW_TO_PHYS_LREAL` ist ein zuverlässiger und genauer Konverter für die ISOBUS-Rohdatenverarbeitung.
Durch die Verwendung von LINT als Zwischenspeicher und LREAL als Ausgabetyp werden Rundungsfehler minimiert und große Wertebereiche abgedeckt. Der Baustein eignet sich besonders für sicherheitskritische oder präzise Landwirtschaftsanwendungen und lässt sich über die parametrierbare Struktur flexibel an verschiedene Messgrößen anpassen.

2.1.14 logiBUS

2.1.14.1 📺 Video

- LogiBUS Projekt Update

2.1.14.2 🎧 Podcast

- Digitale Gedächtnisse: Das SR-Flip-Flop im logiBUS®-System erklärt
- ESP32 als Industrie-SPS: Revolution mit Eclipse 4diac und logiBUS®
- From Cable Chaos to Custom Control: How Logibus is Revolutionizing Agricultural Tech with Accessible ISOBUS
- Logibus Revolution: Unpacking the „No Mapping“ Future and Streamlined Development
- Logibus Unleashed: How Eliminating „Mapping“ Simplifies Complex Systems and Boosts Usability

---

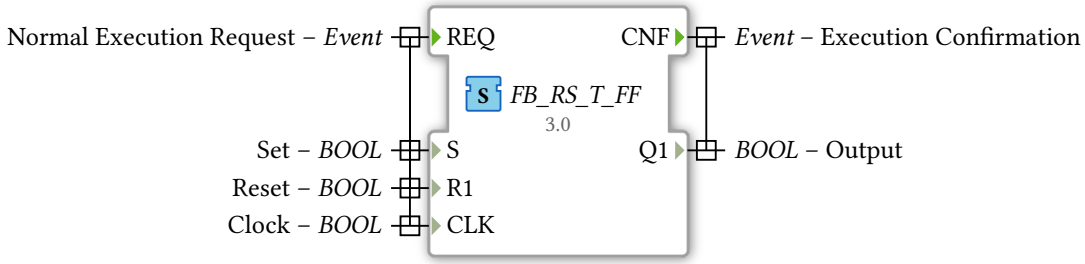
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 ESP32 & ESP32-S3 DevKit auf ms-muc-docs.de

2.1.14.3 Numerischer Index (logiBUS)

Hier finden Sie alle Übungen des Arbeitsbereichs `logiBUS` in aufsteigender Reihenfolge.

FB_RS_T_FF



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `FB_RS_T_FF` realisiert einen bistabilen, reset-dominanten Latch mit zusätzlicher Toggle-Funktion. Er kombiniert die Eigenschaften eines RS-Flipflops (Setzen und Zurücksetzen) mit der Möglichkeit, den Ausgang bei jedem positiven Flankenwechsel des Taktsignals umzuschalten. Der Reset-Eingang hat dabei Priorität, gefolgt vom Set-Eingang und dann der Toggle-Funktion.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Kommentar
`REQ`	Normaler Ausführungsanforderung – löst die Verarbeitung der Eingangssignale aus.

Ereignis-Ausgänge

Ereignis	Kommentar
`CNF`	Bestätigt die Ausführung nach erfolgter Berechnung des Ausgangs `Q1`.

Daten-Eingänge

Variable	Typ	Kommentar
`S`	BOOL	Setzen – setzt den Ausgang `Q1` auf `TRUE`, sofern `R1` nicht gleichzeitig aktiv ist.
`R1`	BOOL	Reset (dominant) – setzt den Ausgang `Q1` auf `FALSE` und hat Vorrang vor `S` und Toggle.
`CLK`	BOOL	Clock – Taktgeber für die Toggle-Funktion; bei steigender Flanke wird `Q1` invertiert, wenn weder Reset noch Set aktiv sind.

Daten-Ausgänge

Variable	Typ	Kommentar
`Q1`	BOOL	Ausgang des Latch – zeigt den aktuellen gespeicherten Zustand an.

Adapter

Der Funktionsblock besitzt keine Adapter-Schnittstellen.

Funktionsweise

Bei jedem Ereignis `REQ` wird der Algorithmus abgearbeitet:

- **Reset dominant**: Falls `R1 = TRUE`, wird `Q1` sofort auf `FALSE` gesetzt – unabhängig von allen anderen Eingängen.
- **Set**: Falls `R1 = FALSE` und `S = TRUE`, wird `Q1` auf `TRUE` gesetzt.
- **Toggle**: Falls weder Reset noch Set aktiv sind (`R1 = FALSE`, `S = FALSE`), wird der Toggle-Mechanismus ausgelöst: Wenn `CLK` den Wert `TRUE` annimmt und eine steigende Flanke erkannt wird (d.h. `CLK` war beim vorherigen Aufruf `FALSE`, gespeichert in der internen Variable `EDGE`), dann wird `Q1` invertiert (`Q1 := NOT Q1`).
- Anschließend wird die interne Variable `EDGE` auf den aktuellen Wert von `CLK` gesetzt, um Flanken für den nächsten Aufruf zu erkennen.

Der Algorithmus wird wie folgt im ST-Code umgesetzt:

```
IF R1 THEN
    Q1 := BOOL#FALSE;
ELSIF S THEN
    Q1 := BOOL#TRUE;
ELSIF CLK AND NOT EDGE THEN
    Q1 := NOT Q1;
END_IF;
EDGE := CLK;
```

Technische Besonderheiten

- **Reset-Dominanz**: Der R1-Eingang hat höchste Priorität; wird er gesetzt, überschreibt er sowohl Set- als auch Toggle-Befehle.
- **Flankenerkennung**: Der Toggle erfolgt nur bei einer steigenden Flanke von `CLK` (Übergang von `FALSE` auf `TRUE`), was durch die interne Variable `EDGE` realisiert wird.
- **Initialzustand**: Die interne `EDGE`-Variable ist mit `TRUE` initialisiert, sodass beim ersten Aufruf kein ungewollter Toggle ausgelöst wird, da die Bedingung `CLK AND NOT EDGE` sonst bei einem statischen `TRUE`-Pegel erfüllt wäre.
- **Datentyp**: Alle Ein- und Ausgänge sind vom Typ `BOOL`.

Zustandsübersicht

Der Funktionsblock besitzt keinen expliziten Zustandsautomaten; der Zustand wird implizit durch die internen Variablen `Q1` und `EDGE` abgebildet. Eine Zustandstabelle fasst das Verhalten zusammen:

Aktuelles `Q1`	`R1`	`S`	`CLK` (steigende Flanke)	Neues `Q1`
x	TRUE	x	x	FALSE
x	FALSE	TRUE	x	TRUE
x	FALSE	FALSE	TRUE (und vorher FALSE)	NOT Q1
x	FALSE	FALSE	FALSE oder keine Flanke	unverändert

Anwendungsszenarien

- **Reset-dominante Steuerung** mit zusätzlicher Umschaltmöglichkeit, z. B. für manuelle Übersteuerung in Sicherheitskreisen.
- **Toggle-Funktion** bei Taktflanken, z. B. als Frequenzteiler oder Umschalter in digitalen Logikschaltungen.
- **Kombinierte Set/Reset/Toggle-Steuerung** in Automatisierungssystemen, wo ein Ausgang sowohl durch Sensorsignale gesetzt als auch durch einen Taster umgeschaltet werden kann.

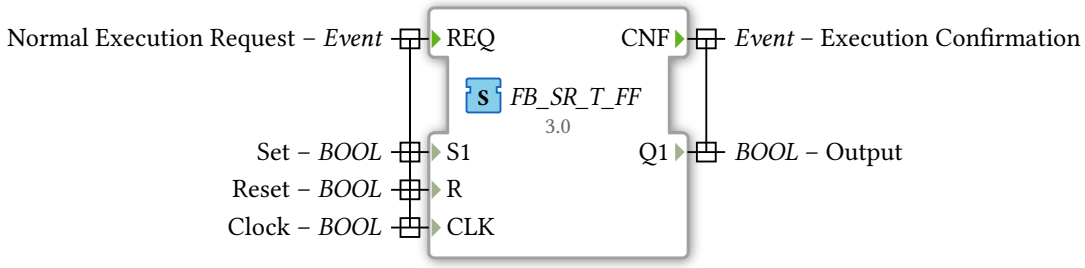
Vergleich mit ähnlichen Bausteinen

- **RS-Flipflop**: Reines RS ohne Toggle; `FB_RS_T_FF` erweitert es um eine Toggle-Funktion.
- **Toggle-Flipflop (T-FF)**: Reiner Toggle ohne Set/Reset; dieser Baustein kombiniert beide Funktionen, wobei der Toggle nur bei inaktiven Set/Reset ausgeführt wird.
- **JK-Flipflop**: Bietet ähnliche Flexibilität (Set, Reset, Toggle), benötigt aber zwei Ereigniseingänge (z. B. für J und K). `FB_RS_T_FF` vereinfacht die Schnittstelle auf einen Ereigniseingang und drei boolesche Daten.

Fazit

FB_RS_T_FF ist ein vielseitiger Funktionsblock, der einen reset-dominanten Latch mit einer flankengetriggerten Toggle-Funktion kombiniert. Er eignet sich für Anwendungen, die sowohl feste Setz- und Rücksetzsignale als auch umschaltbare Zustandswechsel erfordern. Die klare Priorisierung (Reset > Set > Toggle) sowie die eingebaute Flankenerkennung machen ihn robust und einfach in Steuerungslogiken einsetzbar.

FB_SR_T_FF



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein FB_SR_T_FF realisiert ein bistabiles, set-dominantes RS-Flipflop mit zusätzlicher Toggle-Funktion. Die Schaltung kombiniert das Verhalten eines SR-Latches (Set-Dominant) mit einem flankengesteuerten T-Flipflop, das bei jedem Taktzyklus den Ausgangszustand wechselt, wenn weder Set noch Reset aktiv sind.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
REQ	Event	Normaler Ausführungsauftrag (löst Verarbeitung aus)

Ereignis-Ausgänge

Name	Typ	Kommentar
CNF	Event	Bestätigung der Ausführung

Daten-Eingänge

Name	Typ	Initialwert	Kommentar
S1	BOOL	-	Set-Eingang (dominant)
R	BOOL	-	Reset-Eingang
CLK	BOOL	-	Taktsignal (steigende Flanke löst Toggle aus)

Daten-Ausgänge

Name	Typ	Initialwert	Kommentar
Q1	BOOL	-	Ausgangssignal

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Baustein wird durch ein Ereignis am Eingang `REQ` aktiviert. Bei jeder Ausführung werden die Eingänge `S1`, `R` und `CLK` abgetastet und der interne Zustand (`Q1`) nach folgenden Regeln aktualisiert:

1. **Set-Dominanz:** Ist `S1 = TRUE`, wird `Q1` auf `TRUE` gesetzt (unabhängig von `R` und `CLK`).
2. **Reset:** Falls `S1 = FALSE` und `R = TRUE`, wird `Q1` auf `FALSE` gesetzt.
3. **Toggle bei steigender Flanke:** Falls weder `S1` noch `R` aktiv sind (`S1 = FALSE` und `R = FALSE`) und der aktuelle Wert von `CLK` `TRUE` ist, während er im vorherigen Zyklus `FALSE` war (d.h. steigende Flanke), wird der Ausgang `Q1` invertiert (`Q1 := NOT Q1`). Andernfalls bleibt `Q1` unverändert.

Der interne Hilfsmerker `EDGE` speichert den letzten Zustand von `CLK`, um Flankenerkennung zu ermöglichen. Initial ist `EDGE = TRUE`, sodass bei der ersten Ausführung keine steigende Flanke erkannt wird, es sei denn, `CLK` ist von Anfang an `FALSE`.

Der Algorithmus wird in Structured Text (ST) implementiert und nach jeder Verarbeitung das Bestätigungsereignis `CNF` gesendet.

Technische Besonderheiten

- **Set-dominantes Verhalten:** Solange `S1 = TRUE`, wird `R` ignoriert. Dies verhindert den undefinierten Zustand eines klassischen RS-Flipflops.
- **Flankengesteuertes Toggle:** Die Toggle-Funktion wird nur bei einer steigenden Flanke von `CLK` ausgelöst, nicht bei jedem Taktzyklus. Dadurch ist das Verhalten deterministisch und ermöglicht eine Synchronisation mit dem Taktsignal.
- **Interne Speicherung der Flanke:** Der interne Merker `EDGE` speichert den vorherigen Wert von `CLK`. Initial auf `TRUE` gesetzt, damit der erste Zyklus keine Toggle-Auslösung erzwingt.
- **Zyklische Abarbeitung:** Eine Verarbeitung kann beliebig oft aufgerufen werden; die Funktion hängt nur von den aktuellen Eingangswerten und dem gespeicherten Flankenzustand ab.

Zustandsübersicht

Der Baustein besitzt einen funktionalen Zustandsautomaten mit einem Zustand `REQ`. Dieser Zustand führt den Algorithmus aus und sendet anschließend `CNF`. Es gibt keine weiteren expliziten Zustände im Automaten, da das Flipflop-Verhalten rein datengetrieben im Algorithmus abgebildet wird.

Zustand	Eingehende Ereignisse	Aktion	Ausgehende Ereignisse
REQ	REQ	Ausführen des Algorithmus, Aktualisierung von `Q1` und `EDGE`	CNF

Anwendungsszenarien

- **Steuerung mit Set/Reset-Vorrang:** Der Baustein eignet sich für Anwendungen, bei denen ein Vorrang von Set vor Reset erforderlich ist (z.B. Sicherheitsschaltungen).
- **Toggle-Funktion mit Takt:** Kombiniert mit einem Taktgeber kann der Ausgang bei jedem Taktimpuls umschalten, solange keine Set- oder Reset-Anforderung vorliegt. Dies kann z.B. für modulare Zähler oder Zustandsmaschinen genutzt werden.
- **Flankengesteuerte Schaltungen:** In Automatisierungsprozessen, bei denen Zustandsänderungen nur zu bestimmten Zeitpunkten erfolgen dürfen (z.B. Abtastungen).

Vergleich mit ähnlichen Bausteinen

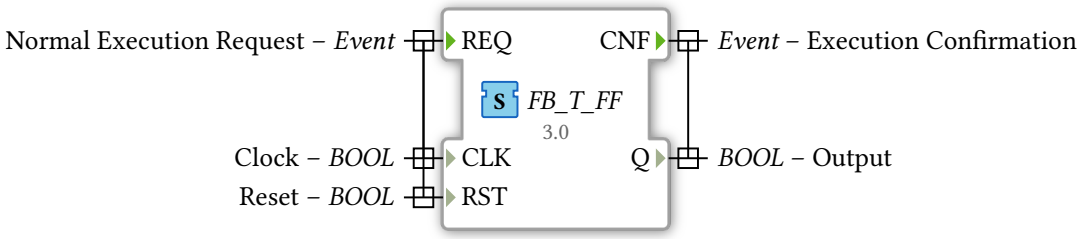
Im Vergleich zu einem reinen SR-Flipflop (z.B. `SR_FF`) bietet dieser Baustein eine eingebaute Toggle-Funktion, die den Ausgang bei jedem Taktzyklus invertieren kann. Gegenüber einem T-Flipflop (`T_FF`) hat der Baustein zusätzlich die Set- und Reset-Eingänge mit Dominanz. Das klassische JK-Flipflop besitzt zwei steuernde Eingänge (J, K) ohne Dominanz, während dieses Baustein eine Kombination aus Set-Dominant und flankengesteuertem Toggle darstellt.

Fazit

Der `FB_SR_T_FF` ist ein vielseitiger, kompakter Funktionsbaustein, der die Eigenschaften eines set-dominanten RS-Flipflops und eines flankengesteuerten T-Flipflops vereint. Dank der einfachen Schnittstelle (ein Ereigniseingang, drei boolesche Eingänge) lässt er sich leicht in bestehende Steuerungslogiken integrieren und eignet sich für eine Vielzahl von Anwendungen, bei denen ein priorisierter Set/Reset mit zusätzlicher Umschaltfunktion benötigt wird. Die Implementierung in Structured Text gewährleistet klare Lesbarkeit und einfache Anpassung.

FB_T_FF

[[Bild des Funktionsbausteins – in der 4diac-IDE als Symbol mit einem Ereigniseingang REQ, Ausgang CNF, Dateneingängen CLK (BOOL) und RST (BOOL) sowie Datenausgang Q (BOOL) dargestellt]]



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **FB_T_FF** (Toggle Flip-Flop) realisiert einen flankengesteuerten, asynchron rücksetzbaren Frequenzteiler oder Umschalter. Bei jedem ansteigenden Taktflanke am Eingang **CLK** wird der Ausgang **Q** invertiert (getoggelt). Ein hoher Pegel am **RST**-Eingang setzt **Q** sofort zurück auf `FALSE`. Der Baustein eignet sich insbesondere für binäre Zustandswechsel in Steuerungsabläufen.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	assoziierte Daten	Beschreibung
`REQ`	`CLK`, `RST`	Normaler Ausführungsauftrag; triggert den Algorithmus

Ereignis-Ausgänge

Ereignis	assoziierte Daten	Beschreibung
`CNF`	`Q`	Bestätigt die Ausführung, liefert den aktuellen Wert von `Q`

Daten-Eingänge

Name	Typ	Beschreibung
`CLK`	BOOL	Taktsignal (Flankenauswertung)
`RST`	BOOL	Asynchroner Reset (aktiv High)

Daten-Ausgänge

Name	Typ	Beschreibung
`Q`	BOOL	Ausgang – toggelt bei steigender CLK-Flanke, es sei denn, Reset ist aktiv

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Baustein wird mit dem **REQ**-Ereignis aufgerufen. Bei jedem Aufruf wird die interne Variable **EDGE** (Bool, initial `TRUE`) wie folgt aktualisiert:

1. **Reset-Vorrang:** Ist `RST = TRUE`, wird `Q` sofort auf `FALSE` gesetzt.
2. **Flankendetektion:** Falls `RST` nicht aktiv ist und die Bedingung `CLK AND NOT EDGE` erfüllt ist, liegt eine **steigende Flanke** von `CLK` vor. In diesem Fall wird `Q` invertiert: `Q := NOT Q`.
3. **Speicherung:** Nach der Auswertung wird `EDGE := CLK` gesetzt, damit bei nächsten Aufruf die Flanke erkannt werden kann.

Der Algorithmus wird im internen Zustand **REQ** ausgeführt; nach dem Algorithmus wird das **CNF**-Ereignis gesendet.

Technische Besonderheiten

- **Flankenerkennung ohne Verzögerung:** Die Erkennung einer steigenden Flanke erfolgt durch den Vergleich des aktuellen `CLK`-Werts mit dem vorherigen (gespeichert in `EDGE`). Dies ist eine typische Software-Realisation eines flankengesteuerten Flipflops.
- **Asynchroner Reset:** Der Reset wirkt sofort bei jedem Aufruf, unabhängig von der Taktflanke.
- **Initialzustand:** `Q` ist nach dem Starten des Bausteins nicht definiert; es wird erst durch den ersten REQ-Aufruf oder einen Reset gesetzt. Die interne **EDGE** ist initial `TRUE`, sodass die erste steigende Flanke (von `FALSE` auf `TRUE`) korrekt erkannt wird.

Zustandsübersicht

Der Baustein ist als **SimpleFB** mit einem einzigen EC-Zustand implementiert:

Zustand	Eingehende Ereignisse	Ausgehende Aktion	Ausgehendes Ereignis
`REQ`	`REQ`	Führe Algorithmus `REQ` aus (Toggle/Reset-Logik)	`CNF`

Der Zustand wird nach Abarbeitung sofort wieder scharf geschaltet; es gibt keine Warte- oder Folgezustände.

Anwendungsszenarien

- **Frequenzteiler:** Anlegen eines Rechtecksignals an `CLK`; an `Q` erscheint die halbe Frequenz.
- **Binärer Zustandswechsel:** Umschalten einer Leuchte oder eines Ventils bei jedem Taktimpuls.
- **Reset-gesicherter Toggle:** Durch Ansteuern von `RST` kann der Ausgang in eine definierte Grundstellung gezwungen werden.
- **Pulszähler:** Kombination mehrerer FB_T_FF hintereinander ergibt einen binären Zähler.

Vergleich mit ähnlichen Bausteinen

Baustein	Eigenschaft
SR-Flipflop	Setzt und rückgesetzt durch separate Eingänge; kein Toggle-Verhalten.
D-Flipflop	Übernimmt den Wert von `D` bei Taktflanke; kein Toggle, kein Reset integriert.
FB_T_FF	Invertiert den Ausgang bei jeder steigenden Flanke, asynchroner Reset möglich.

Der **FB_T_FF** ist damit ideal für einfache Umschaltaufgaben, bei denen kein Setz-/Rücksetz-Eingang benötigt wird.

Fazit

Der **FB_T_FF** realisiert ein robustes, flankengesteuertes Toggle-Flipflop mit asynchronem Reset. Durch die kompakte Implementierung eignet er sich für zeitkritische Steuerungen in der Automatisierungstechnik, bei denen zuverlässige Zustandswechsel erforderlich sind. Die Flankendetektion ist softwaremäßig sauber gelöst und benötigt keine zusätzliche Hardware.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

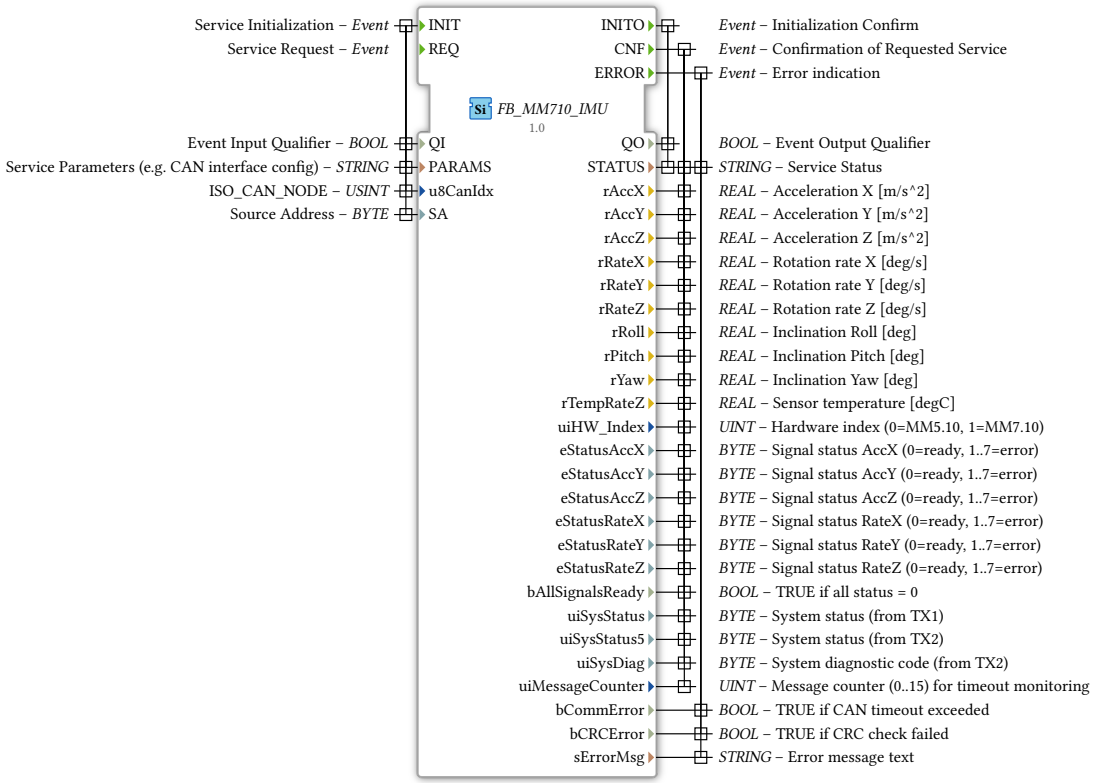
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.1.14.4 Bosch

Numerischer Index (bosch)

Hier finden Sie alle Übungen des Arbeitsbereichs `bosch` in aufsteigender Reihenfolge.

FB_MM710_IMU



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **FB_MM710_IMU** ist ein serviceorientierter Baustein (SIFB) zur Anbindung des Bosch MM7.10 IMU-Sensors über CAN/J1939. Er ermöglicht das Auslesen von Beschleunigungs-, Drehraten- und Neigungswerten sowie die Überwachung von System- und Fehlerzuständen. Der FB kapselt die gesamte CAN-Kommunikation und Signalverarbeitung und stellt die Daten standardisiert über Ereignis- und Datenausgänge zur Verfügung.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung
INIT	EInit	Initialisierung des Bausteins. Mit diesem Ereignis werden die CAN-Parameter (Index, Source-Address) sowie der Aktivierungs-Qualifier QI gesetzt.

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung
INITO	EInit	Bestätigung der erfolgreichen Initialisierung (QO = TRUE) oder Fehlermeldung.
IND	Event	Indikation von der Ressource: feuert bei jedem Zyklus ohne aktiven Fehler und publiziert dabei auch das Zurücksetzen von `bCommError`/`bCRCError` auf FALSE. Liefert QO, STATUS, alle Mess- und Statuswerte (`rAcc*`, `rRate*`, `rRoll`, `rPitch`, `rYaw`, `rTempRateZ`, `uiHW_Index`, `eStatus*`, `bAllSignalsReady`, `uiSysStatus`, `uiSysStatus5`, `uiSysDiag`, `uiMessageCounter`).
ERROR	Event	Feuert ausschließlich beim tatsächlichen Eintreten eines Fehlers (Entry-Trigger); liefert QO, STATUS, `bCommError`, `bCRCError` und `sErrorMsg`. Das Zurücksetzen von `bCommError`/`bCRCError` wird nicht hier, sondern über IND gemeldet.

Daten-Eingänge

Variable	Typ	Beschreibung
QI	BOOL	Aktivierungs-Qualifier: Nur bei QI = TRUE wird die Initialisierung (INIT) ausgeführt.
PARAMS	STRING	Service-Parameter, z. B. CAN-Interface-Konfiguration (optional).
u8CanIdx	USINT	CAN-Node-Index (Standard-Initialwert: `INVALID`).
SA	BYTE	Source-Address für J1939-Kommunikation (Initialwert: `16#DA`).

Daten-Ausgänge

Variable	Typ	Beschreibung
QO	BOOL	Quittierung der Initialisierung (TRUE = erfolgreich).
STATUS	STRING	Statusmeldung (z. B. „Initialized“, „Error“).
rAccX, rAccY, rAccZ	REAL	Beschleunigungswerte in X-, Y- und Z-Richtung [m/s²].
rRateX, rRateY, rRateZ	REAL	Drehraten um die jeweilige Achse [deg/s].
rRoll, rPitch, rYaw	REAL	Neigungswinkel (Roll, Pitch, Yaw) [deg].
rTempRateZ	REAL	Sensortemperatur [°C].
uiHW_Index	UINT	Hardware-Index (0 = MM5.10, 1 = MM7.10).
eStatusAccX … eStatusAccZ	BYTE	Signalqualität der Beschleunigung (einzeln `eStatusAccX`, `eStatusAccY`, `eStatusAccZ`; 0 = bereit, 1 .. 7 = Fehler).
eStatusRateX … eStatusRateZ	BYTE	Signalqualität der Drehraten (einzeln `eStatusRateX`, `eStatusRateY`, `eStatusRateZ`; 0 = bereit, 1 .. 7 = Fehler).
bAllSignalsReady	BOOL	TRUE, wenn alle Signal-Status 0 sind.
uiSysStatus	BYTE	Systemstatus aus TX-Nachricht 1.
uiSysStatus5	BYTE	Systemstatus aus TX-Nachricht 2.
uiSysDiag	BYTE	Systemdiagnosecode (aus TX2).
uiMessageCounter	UINT	Nachrichtenzähler (0..15) zur Timeout-Überwachung.
bCommError	BOOL	TRUE bei CAN-Timeout.
bCRCError	BOOL	TRUE bei fehlerhafter CRC-Prüfung.
sErrorMsg	STRING	Fehlertext (z. B. „CAN timeout“).

Adapter

Keine Adapter definiert.

Funktionsweise

Der FB_MM710_IMU initialisiert beim Eintreffen von **INIT** mit QI = TRUE die CAN-Kommunikation und den internen Empfangspuffer. Nach erfolgreicher Initialisierung wird **INITO** mit QO = TRUE gesetzt. Der Baustein hat keinen Anforderungseingang: Sind gültige Daten vom Sensor empfangen, wird in jedem Zyklus ohne aktiven Fehler **IND** ausgegeben und alle Datenausgänge aktualisiert. Tritt ein Kommunikations- oder CRC-Fehler auf oder überschreitet der Nachrichtenzähler einen Timeout, wird **ERROR** gesetzt.

Die Signal-Status (eStatus*) ermöglichen eine Einzelfehleranalyse für jede Achse. Der Hardware-Index unterscheidet zwischen älteren MM5.10 und aktuellen MM7.10 Sensoren.

Technische Besonderheiten

- **CAN/J1939-Protokoll** – Verwendung einer festen Source-Address (Standard: `16#DA`).
- **Time-out-Überwachung** – Der `uiMessageCounter` (0-15) wird bei jeder gültigen Nachricht hochgezählt; bleibt er aus, wird nach 16 fehlenden Nachrichten ein Kommunikationsfehler gemeldet.
- **Signal-Status-Bits** – Liefern granularere Informationen als einfache „ready/error"-Flags.
- **CRC-Prüfung** – Fehlerhafte CAN-Frames werden erkannt und über `bCRCError` und **ERROR** gemeldet.
- **Hardware-Unterscheidung** – `uiHW_Index` erlaubt adaptives Verhalten für unterschiedliche Sensor-Versionen.

Zustandsübersicht

Der Baustein durchläuft folgende Zustände (nicht explizit als ECC, aber aus dem Verhalten ableitbar):

1. **Inaktiv** – Nach dem Start, wartet auf INIT.
2. **Initialisieren** – Nach INIT-Ereignis; Aufbau der CAN-Kommunikation.
3. **Bereit** – Nach erfolgreichem INITO; auf Sensor-Nachrichten wartend.
4. **Daten empfangen** – Nach erfolgreicher CAN-Nachricht; IND wird gesendet.
5. **Fehler** – Bei Timeout oder CRC-Fehler; ERROR wird gesendet (Rückfall in Bereit nach Fehlerbehandlung).

Anwendungsszenarien

- **Mobile Arbeitsmaschinen** – Neigungs- und Beschleunigungsüberwachung von Baggern, Kränen oder Gabelstaplern.
- **Fahrzeugdynamik** – Erfassung von Roll-, Pitch- und Yaw-Winkeln für Stabilitätskontrollen.
- **Industrieroboter** – Überwachung von Vibrationen und unerwarteten Bewegungen.
- **IoT-Sensorknoten** – Einbindung in übergeordnete Steuerungen mittels CAN-Bus.

Vergleich mit ähnlichen Bausteinen

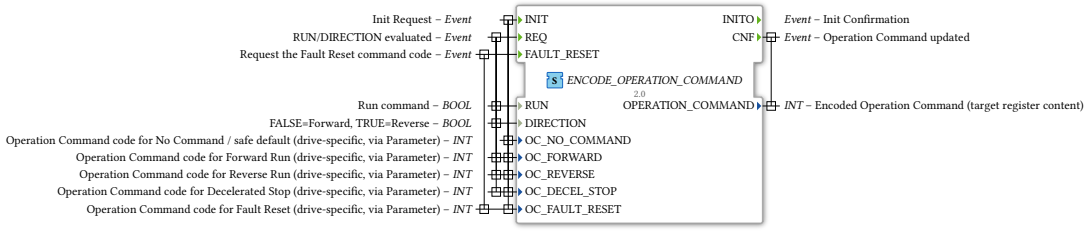
Gegenüber einfachen IMU-Treibern (z. B. per SPI/I²C) bietet dieser FB eine direkte Integration in J1939-Netzwerke. Die Signal-Status-Bits ermöglichen eine Diagnose, die bei Standard-Bausteinen oft fehlt. Der integrierte Hardware-Index (MM5.10 / MM7.10) erlaubt eine einfache Migration. Andere CAN-IMU-Bausteine verwenden ggf. proprietäre Nachrichtenformate, während dieser FB auf dem offenen J1939-Standard basiert.

Fazit

Der FB_MM710_IMU ist ein mächtiger Baustein für die zuverlässige Erfassung von IMU-Daten in CAN-basierten Automatisierungssystemen. Seine umfangreichen Status- und Fehlerinformationen unterstützen eine lückenlose Diagnose, und die einfache Parametrisierung über INIT macht ihn flexibel einsetzbar. Besonders in sicherheitskritischen Anwendungen mit J1939 ist er eine optimale Wahl.

2.1.14.5 drives

ENCODE_OPERATION_COMMAND



• – ○ • – ○ • – ○ •

Einleitung

`ENCODE_OPERATION_COMMAND` erzeugt das Operation-Command-Wort für einen Frequenzumrichter/Antrieb (VFD) aus `RUN/DIRECTION` bzw. einem expliziten Fault-Reset-Ereignis. Der Baustein ist generisch für jeden Antrieb gedacht, dessen Steuerwort dem verbreiteten Run/Direction/Stop-plus-Fault-Reset-Muster folgt (Modbus, CANopen PZD1, Profibus PPO usw.) — alle konkreten Befehlscodes werden über Parameter vom Aufrufer vorgegeben, der Baustein selbst kennt keine herstellerspezifischen Werte, nur die Kodierlogik.

Die Bezeichnung `Operation Command` (statt `Control Word`) ist bewusst gewählt: `Control Word` ist CANopen-spezifische Terminologie, während `Operation Command` der herstellerneutrale Begriff ist, wie ihn z. B. das DELIXI-H300-Handbuch verwendet.

Schnittstellenstruktur

Ereignis-Eingänge

- **`INIT`**: Initialisierung — setzt `OPERATION_COMMAND` einmalig auf `OC_NO_COMMAND`.
- **`REQ`**: `RUN`/`DIRECTION` neu auswerten und `OPERATION_COMMAND` entsprechend kodieren.
- **`FAULT_RESET`**: Fault-Reset-Befehlscode anfordern, unabhängig vom aktuellen `RUN`/`DIRECTION`-Zustand.

Ereignis-Ausgänge

- **`INITO`**: Initialisierungsbestätigung.
- **`CNF`**: `OPERATION_COMMAND` wurde aktualisiert.

Daten-Eingänge

- **`RUN`** (BOOL): Laufbefehl.
- **`DIRECTION`** (BOOL): `FALSE`=Vorwärts, `TRUE`=Rückwärts.
- **`OC_NO_COMMAND`** (INT): Befehlscode für „kein Befehl"/sicherer Grundzustand (antriebsspezifisch, per Parameter).
- **`OC_FORWARD`** (INT): Befehlscode für Vorwärtslauf (antriebsspezifisch, per Parameter).
- **`OC_REVERSE`** (INT): Befehlscode für Rückwärtslauf (antriebsspezifisch, per Parameter).
- **`OC_DECEL_STOP`** (INT): Befehlscode für kontrollierten Stopp (antriebsspezifisch, per Parameter).
- **`OC_FAULT_RESET`** (INT): Befehlscode für Fault Reset (antriebsspezifisch, per Parameter).

Daten-Ausgänge

- **`OPERATION_COMMAND`** (INT): Kodiertes Operation-Command-Wort (Inhalt des Zielregisters).

Funktionsweise

Der Baustein hat keine echte Zustandsmaschine (kein `<ECC>` mit Übergangsbedingungen) — jedes Ereignis löst unabhängig genau einen Algorithmus aus:

1. **`INIT`**: Setzt `OPERATION_COMMAND := OC_NO_COMMAND`, bevor der erste `REQ`/`FAULT_RESET` eintrifft.
2. **`REQ`**: Ist `RUN = TRUE`, wird abhängig von `DIRECTION` `OC_FORWARD` oder `OC_REVERSE` kodiert; ist `RUN = FALSE`, wird `OC_DECEL_STOP` kodiert.
3. **`FAULT_RESET`**: Überschreibt `OPERATION_COMMAND` einmalig mit `OC_FAULT_RESET`, unabhängig vom aktuellen `RUN`/`DIRECTION`-Zustand — der nächste `REQ` kodiert danach wieder normal.

Technische Besonderheiten

- **Alle Befehlscodes sind Parameter, kein festverdrahtetes Wissen.** Der Baustein selbst kennt keine antriebsspezifischen Zahlenwerte; `OC_*` werden vom Aufrufer über `Parameter` vorgegeben.
- **Umbenennung in Version 2.0**: Vormals `ENCODE_CONTROLWORD`/`CW_*`/`CONTROL_WORD`, da „Control Word" CANopen-spezifische Terminologie ist. Semantik unverändert, nur Namen generalisiert.
- **Verallgemeinerung aus `H300_ENCODE_CONTROLWORD`** (Version 1.0): Ursprünglich für einen konkreten Antriebstyp geschrieben, dann auf das generische Run/Direction/Stop-plus-Fault-Reset-Muster verallgemeinert.

Anwendungsszenarien

- Ansteuerung eines Frequenzumrichters/Antriebs über ein Feldbus-Steuerwort (Modbus-Holding-Register, CANopen PZD1, Profibus PPO), bei dem Run/Direction/Stop und ein expliziter Fault-Reset in einem einzigen Wort kodiert werden.
- Generischer Baustein für mehrere Antriebstypen: nur die `OC_*`-Parameter müssen pro Antrieb angepasst werden, die Kodierlogik bleibt gleich.

Fazit

`ENCODE_OPERATION_COMMAND` kapselt die verbreitete Run/Direction/Stop-plus-Fault-Reset-Kodierung für VFD-Steuerworte in einem einzigen, herstellerneutralen Baustein — alle konkreten Befehlscodes kommen von außen, der Baustein selbst trägt kein antriebsspezifisches Wissen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.1.14.6 esp32

📺 Video

- Das ESP32-S3-DevKitC-1

🎧 Podcast

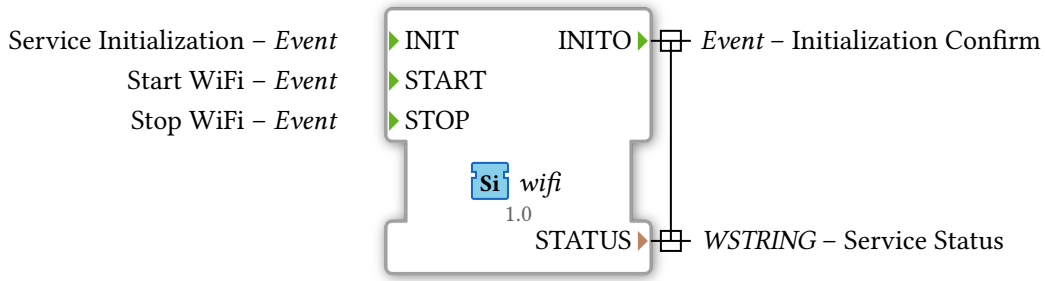
- ESP32 als Industrie-SPS: Revolution mit Eclipse 4diac und logiBUS®
- ESP32 as PLC: Democratizing Industrial Automation with Eclipse 4diac
- ESP32 wird industrielle SPS für Landmaschinen
- ESP32-S3 Entwicklungsplatinen ESP32-S3-DevKitC-1
- ESP32-S3 im Detail: Dual-Core, 32MB Power und CAN-Bus für Land- und Baumaschinen-Mechatronik

---

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 ESP32 & ESP32-S3 DevKit auf ms-muc-docs.de

wifi



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock „wifi“ ist ein Service-Interface-Funktionsblock (SIFB) zur Steuerung einer WLAN-Verbindung. Er bietet die Ereignisse INIT, START und STOP zur Initialisierung, zum Starten und Stoppen des WLANs. Der Status der Verbindung wird über den Ausgang STATUS als Zeichenkette zurückgegeben. Der Baustein ist für den Einsatz in Automatisierungssystemen konzipiert, die eine drahtlose Netzwerkverbindung benötigen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisiert den Baustein und bereitet die WLAN-Schnittstelle vor.
- **START** (EInit): Startet die WLAN-Verbindung.
- **STOP** (EInit): Stoppt die WLAN-Verbindung.

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigt die erfolgreiche Initialisierung. Wird in Verbindung mit dem Datenausgang STATUS ausgelöst.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

- **STATUS** (WSTRING): Zeigt den aktuellen Status des WLAN-Dienstes an (z. B. „OK“, „Fehler“, „Verbunden“, „Getrennt“).

Adapter

Keine Adapter verfügbar.

Funktionsweise

Der Baustein arbeitet nach dem Prinzip eines Service-Interface-Funktionsblocks. Die interne Logik wird durch die Laufzeitumgebung bereitgestellt. Typischerweise wird zuerst INIT aufgerufen, um den Baustein zu konfigurieren. Danach kann mit START die WLAN-Verbindung aktiviert werden. Mit STOP wird die Verbindung beendet. Der Ausgang STATUS gibt nach jeder Operation den aktuellen Zustand zurück.

Technische Besonderheiten

- Der Baustein ist als SIFB (Service Interface Function Block) nach IEC 61499-2 realisiert.
- Die Schnittstelle ist sowohl zur RESOURCE- als auch zur APPLICATION-Seite hin offen, was eine flexible Integration ermöglicht.
- Die genaue Implementierung ist abhängig von der zugrunde liegenden Hardware und dem Betriebssystem.

Zustandsübersicht

Der Baustein durchläuft folgende grundlegende Zustände:

- **Uninitialisiert**: Nach dem Start, bevor INIT ausgeführt wird.
- **Initialisiert**: Nach erfolgreichem INIT, bereit für START.
- **Aktiv (verbunden)**: Nach erfolgreichem START.
- **Inaktiv (getrennt)**: Nach STOP oder bei einem Fehler.

Anwendungsszenarien

- Steuerung einer WLAN-Schnittstelle in landwirtschaftlichen Maschinen oder anderen Fahrzeugen.
- Einbindung von Geräten in ein drahtloses Netzwerk zur Datenübertragung in der Automatisierungstechnik.
- Fernwartung und -diagnose über WLAN.

Vergleich mit ähnlichen Bausteinen

Ähnliche Bausteine sind beispielsweise Ethernet- oder serielle Kommunikationsbausteine. Im Gegensatz zu diesen fokussiert der wifi-Baustein speziell auf die drahtlose WLAN-Verbindung und bietet eine vereinfachte, ereignisgesteuerte Schnittstelle. Andere Bausteine erfordern oft komplexere Konfigurationen oder unterstützen mehrere Protokolle.

Fazit

Der Funktionsblock „wifi“ stellt eine einfache und standardisierte Möglichkeit zur Integration einer WLAN-Verbindung in IEC-61499-Systeme dar. Mit den drei Ereignissen INIT, START und STOP sowie dem STATUS-Ausgang ist er intuitiv bedienbar und eignet sich besonders für Anwendungen, die eine schnelle und zuverlässige kabellose Kommunikation erfordern.

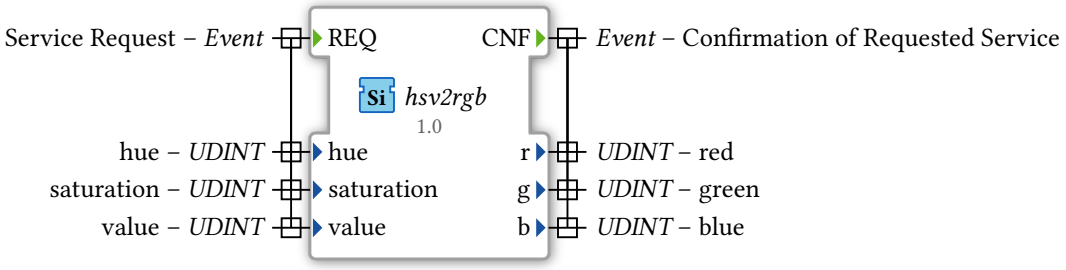
rgb

hsv2rgb

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `hsv2rgb` ist ein Hilfsbaustein zur Farbraumkonvertierung. Er wandelt eine im HSV-Farbmodell (Hue, Saturation, Value) definierte Farbe in die entsprechenden Werte des RGB-Farbmodells (Red, Green, Blue) um. Dies ist besonders nützlich für Anwendungen, die Farben auf Basis ihrer wahrnehmungsbezogenen Eigenschaften (Farbton, Sättigung, Helligkeit) steuern müssen, während die Ausgabegeräte (wie LEDs oder Displays) RGB-Werte erwarten.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Service Request):** Löst die Konvertierungsberechnung aus. Bei diesem Ereignis werden die aktuellen Werte an den Daten-Eingängen `hue`, `saturation` und `value` übernommen.

Ereignis-Ausgänge

- **CNF (Confirmation of Requested Service):** Wird ausgelöst, sobald die Konvertierung abgeschlossen ist. Dieses Ereignis liefert die berechneten RGB-Werte an den Daten-Ausgängen `r`, `g` und `b`.

Daten-Eingänge

- **hue (UDINT):** Der Farbton (Hue) im HSV-Modell. Der Wertebereich ist nicht im Baustein vordefiniert, typischerweise wird jedoch ein Bereich von 0-360° oder 0-65535 verwendet.
- **saturation (UDINT):** Die Sättigung (Saturation) im HSV-Modell. Typischer Wertebereich: 0-100% oder 0-255.
- **value (UDINT):** Die Helligkeit (Value) im HSV-Modell. Typischer Wertebereich: 0-100% oder 0-255.

Daten-Ausgänge

- **r (UDINT):** Der berechnete Rot-Anteil (Red) im RGB-Modell.
- **g (UDINT):** Der berechnete Grün-Anteil (Green) im RGB-Modell.
- **b (UDINT):** Der berechnete Blau-Anteil (Blue) im RGB-Modell.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Baustein arbeitet ereignisgesteuert. Bei Eintreffen des `REQ`-Ereignisses startet der interne Algorithmus. Er nimmt die an den Daten-Eingängen anliegenden HSV-Werte entgegen und berechnet daraus die entsprechenden RGB-Werte gemäß dem standardisierten HSV-zu-RGB-Konvertierungsalgorithmus. Nach Abschluss der Berechnung werden die Ergebnisse an den Daten-Ausgängen `r`, `g` und `b` ausgegeben und das `CNF`-Ereignis signalisiert, dass die neuen Werte gültig und abrufbereit sind.

Technische Besonderheiten

- **Datentyp:** Alle Eingangs- und Ausgangsdaten sind vom Typ `UDINT` (unsigned double integer, 32-bit). Dies ermöglicht eine hohe Auflösung und Genauigkeit bei der Farbdarstellung.
- **Wertebereich:** Der Baustein selbst definiert keine Skalierung oder Begrenzung der Wertebereiche für HSV oder RGB. Die Interpretation der Zahlenwerte (z.B. 0-360 für Hue, 0-255 für RGB) liegt in der Verantwortung der anbindenden Applikation. Der Konvertierungsalgorithmus muss entsprechend auf den erwarteten Eingabebereich ausgelegt sein.
- **Deterministisches Verhalten:** Die Ausführung ist deterministisch und liefert für identische Eingabewerte stets die gleichen RGB-Ausgabewerte.

Zustandsübersicht

Der Funktionsblock besitzt einen einfachen, zustandslosen Ablauf:

1. **Warte-Zustand:** Der Baustein wartet auf das `REQ`-Ereignis.
2. **Berechnungs-Zustand:** Bei `REQ` werden die Eingabedaten gelesen und die Konvertierung durchgeführt. Dieser Zustand ist transient.

3. **Ausgabe-Zustand:** Die Ergebnisse werden an die Ausgänge geschrieben und das `CNF`-Ereignis wird generiert. Der Baustein kehrt unmittelbar danach in den Warte-Zustand zurück.

Anwendungsszenarien

- **Farbsteuerung von RGB-LEDs:** Benutzerfreundliche Farbauswahl über HSV-Parameter (z.B. via Drehgeber oder Touch-Oberfläche) mit anschließender Umrechnung für die LED-Ansteuerung.
- **Visualisierungs-Systeme:** Konvertierung von in HSV definierten Alarm- oder Statusfarben für die Darstellung auf RGB-Monitoren oder -Panels.
- **Industrielle Beleuchtung:** Dynamische Lichtsteuerung in Produktions- oder Logistikumgebungen, bei der Farbübergänge intuitiv über Helligkeit und Sättigung definiert werden.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu universellen Rechenbausteinen (wie `MUL`, `ADD`) oder Skalierern ist `hsv2rgb` ein anwendungsspezifischer Baustein, der einen komplexen, nicht-linearen Algorithmus kapselt. Er erspart dem Anwender die aufwändige und fehleranfällige Implementierung der Konvertierung in Structured Text oder anderen Sprachen. Ein direkter Gegenpart `rgb2hsv` für die umgekehrte Konvertierung wäre ein ähnlich spezialisierter Baustein.

🛠️ Zugehörige Übungen

- Uebung_031

Fazit

Der `hsv2rgb`-Funktionsblock ist ein nützliches und spezialisiertes Werkzeug für alle 4diac-Anwendungen, die mit Farbmanipulation zu tun haben. Durch die Kapselung des Konvertierungsalgorithmus in einen standardisierten IEC 61499-Baustein vereinfacht er die Programmierung erheblich, fördert die Wiederverwendbarkeit und erhöht die Zuverlässigkeit der Software. Seine Verwendung ist dort empfehlenswert, wo eine intuitive, auf menschlicher Farbwahrnehmung basierende Steuerung auf eine geräteorientierte RGB-Ausgabe trifft.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

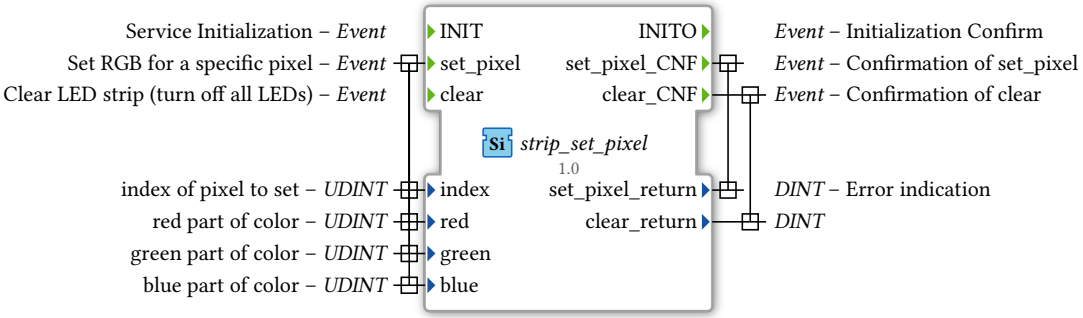
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

strip_set_pixel

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `strip_set_pixel` dient zur Steuerung einer RGB-LED-Leiste. Seine Hauptaufgabe ist es, die Farbe eines einzelnen Pixels (einer einzelnen LED) auf der Leiste zu setzen oder die gesamte Leiste auszuschalten. Er ist für den Einsatz in Steuerungssystemen konzipiert, die eine präzise, pixelweise Ansteuerung von RGB-LEDs erfordern.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisiert den Funktionsblock. Muss vor der ersten Nutzung ausgelöst werden.
- **set_pixel**: Löst den Befehl aus, die Farbe eines spezifischen Pixels zu setzen. Wird mit den Daten `index`, `red`, `green` und `blue` verknüpft.

- **clear**: Löst den Befehl aus, alle LEDs der Leiste auszuschalten (zu löschen).

Ereignis-Ausgänge

- **INITO**: Bestätigt die erfolgreiche Initialisierung des Funktionsblocks.
- **set_pixel_CNF**: Bestätigt die Ausführung des `set_pixel`-Befehls. Ist mit dem Datenausgang `set_pixel_return` verknüpft.
- **clear_CNF**: Bestätigt die Ausführung des `clear`-Befehls. Ist mit dem Datenausgang `clear_return` verknüpft.

Daten-Eingänge

- **index** (UDINT): Der Index (Position) des Pixels auf der LED-Leiste, das gesetzt werden soll. Die Zählung beginnt typischerweise bei 0.
- **red** (UDINT): Der Rotanteil der Farbe (Wertebereich abhängig von der Hardware, z.B. 0-255).
- **green** (UDINT): Der Grünanteil der Farbe (Wertebereich abhängig von der Hardware, z.B. 0-255).
- **blue** (UDINT): Der Blauanteil der Farbe (Wertebereich abhängig von der Hardware, z.B. 0-255).

Daten-Ausgänge

- **set_pixel_return** (DINT): Gibt einen Fehlerstatus nach Ausführung des `set_pixel`-Befehls zurück. Ein Wert ungleich 0 deutet auf einen Fehler hin (z.B. ungültiger Pixel-Index).
- **clear_return** (DINT): Gibt einen Fehlerstatus nach Ausführung des `clear`-Befehls zurück. Ein Wert ungleich 0 deutet auf einen Fehler hin.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Der FB arbeitet ereignisgesteuert. Bei einem eingehenden `INIT`-Ereignis wird die zugrundeliegende Hardware (LED-Leiste) initialisiert und das `INITO`-Ereignis generiert.

Das `set_pixel`-Ereignis löst den Hauptbefehl aus. Die zu diesem Zeitpunkt an den Daten-Eingängen `index`, `red`, `green` und `blue` anliegenden Werte werden ausgelesen und an die LED-Leiste gesendet, um die entsprechende LED auf die gewünschte Farbe zu setzen. Anschließend wird das `set_pixel_CNF`-Ereignis zusammen mit einem Statuscode (`set_pixel_return`) ausgegeben.

Das `clear`-Ereignis setzt alle Pixel der Leiste auf Schwarz (aus), was einem Löschen des aktuellen Musters entspricht. Danach wird das `clear_CNF`-Ereignis mit Status (`clear_return`) ausgegeben.

Technische Besonderheiten

- Die Daten-Eingänge für die Farbwerte sind vom Typ `UDINT` (unsigned double integer), was einen großen Wertebereich ermöglicht. Die tatsächlich wirksame Auflösung (z.B. 8-Bit = 0-255) hängt von der angeschlossenen Hardware ab.
- Die Daten-Ausgänge für die Rückmeldungen sind vom vorzeichenbehafteten Typ `DINT`, um positive und negative Fehlercodes darstellen zu können.
- Der FB folgt dem typischen IEC 61499 Service Interface Pattern mit separaten Bestätigungsereignissen (`CNF`) für jeden Befehl.

Zustandsübersicht

1. **Nicht initialisiert**: Nach dem Start. Nur das `INIT`-Ereignis ist gültig.
2. **Initialisiert / Bereit**: Nach erfolgreichem `INITO`. Die Befehle `set_pixel` und `clear` können empfangen und verarbeitet werden. Nach jedem Befehl kehrt der FB in diesen Zustand zurück.

Anwendungsszenarien

- **Statusanzeigen**: Einzelne LEDs können als Status- oder Warnlichter genutzt werden (z.B. Maschine in Betrieb, Fehler an Position X).
- **Einfache Lichteffekte**: Durch sequentielles Setzen einzelner Pixel können Lauflicht- oder Blinkeffekte realisiert werden.
- **Visualisierung von Prozessdaten**: Die Farbe oder Helligkeit eines Pixels kann einen Prozesswert (z.B. Temperatur, Füllstand) repräsentieren.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Bausteinen, die ganze Farbmuster oder Bilder auf eine LED-Matrix übertragen (z.B. `strip_show` oder `matrix_display`), arbeitet `strip_set_pixel` auf einer sehr granularen, pixelweisen Ebene. Er bietet maximale Kontrolle über jede einzelne LED, ist für die Übertragung komplexer Bilder jedoch weniger effizient. Bausteine wie `strip_fill` setzen typischerweise einen ganzen Bereich oder alle LEDs auf einmal auf eine Farbe, während `strip_set_pixel` individuell adressiert.

🛠️ Zugehörige Übungen

- Uebung_031

Fazit

Der `strip_set_pixel`-Funktionsblock ist ein grundlegender und essentieller Baustein für die Ansteuerung von adressierbaren RGB-LED-Leisten innerhalb von 4diac. Durch sein einfaches, ereignisbasiertes Interface ermöglicht er eine präzise und direkte Kontrolle über jedes einzelne Pixel. Seine Stärke liegt in der Einfachheit für punktuelle Steueraufgaben, während für komplexere Animationen die Kombination mit übergeordneten, sequenzierenden Bausteinen empfohlen wird.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.1.14.7 🔌 io

📺 Video

- Automation der Zukunft
- Die große Migration
- Digital decisions
- Ideale Dioden
- The Dual Wi Fi Solution

🎧 Podcast

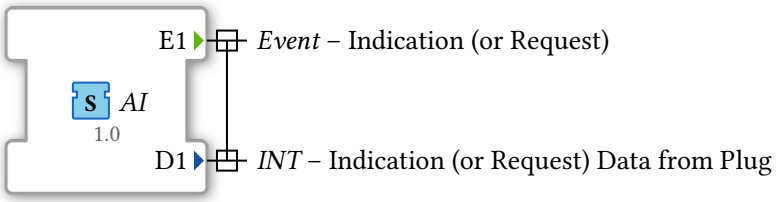
- 4diac IDE: Wie der IEC 61499 Standard die Industrieautomatisierung revolutioniert
- 4diac-Präsentation: Zielgruppen, Struktur und Alleinstellungsmerkmal Schärfen
- Den Software-Drachen zähmen: Industrielle Automatisierung und die Zukunft der Produktion
- Eclipse 4diac: Innovationsmotor Forschung vs. Nutzerbedürfnisse – Was treibt die Entwicklung wirklich voran?
- Software-Drachen zähmen: Warum IEC 61499 die Industrie revolutionieren könnte

---

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 Diode & Halbleitergrundlagen auf ms-muc-docs.de
- 🌐 Smart Switch vs. Relais auf ms-muc-docs.de

AI



📺 Video

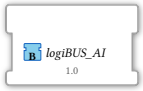
- AI: The Unknown Story
- The silent failure

🎧 Podcast

- Von klickenden Relais zum intelligenten Code: Wie Software die Industriesteuerung revolutionierte
- ISOBUS-Container: Dynamische Bedienfelder für klare Sicht und mehr Effizienz
- Analyse der Novellierung der Meisterprüfungsverordnung im Land- und Baumaschinenmechatroniker-Handwerk: Ein Detaillierter Vergleich der Verordnungen von 2024 und 2001
- Das Relais im Detail: Schaltverstärker, Schutz und die Geheimnisse von A1/A2, 85/86 und der Hysterese
- Das tödliche Dilemma des Relais: Warum Silberkontakte bei Kleinsignalen versagen und Gold bei Last schmilzt – Der Freibrenn-Effekt erklärt

---

logiBUS_AI



• – ◦ • – ◦ • – ◦ •

Einleitung

Die logiBUS_AI GlobalConstants definieren eine Sammlung von Konstanten für die Konfiguration von analogen Eingängen im logiBUS-System. Diese Konstanten werden zur einfachen und einheitlichen Referenzierung der analogen Eingangskanäle verwendet.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

Keine Adapter vorhanden

Funktionsweise

Die logiBUS_AI GlobalConstants stellen vordefinierte Konstanten vom Typ `logiBUS_AI_S` bereit, die jeweils einen spezifischen analogen Eingangskanal repräsentieren. Jede Konstante enthält eine Pin-Nummer, die den physikalischen Eingangskanal identifiziert.

Technische Besonderheiten

- **Konstantentyp**: Alle Konstanten sind vom strukturierten Datentyp `logiBUS_AI_S`
- **Pin-Zuordnung**: Die Konstanten `AnalogInput_I1` bis `AnalogInput_I9` entsprechen den analogen Eingängen 1 bis 9
- **Ungültige Referenz**: Die Konstante `Invalid` mit Pin-Wert 255 dient als Marker für ungültige oder nicht vorhandene Eingänge
- **Initialisierung**: Alle Konstanten werden bei der Deklaration mit ihren entsprechenden Pin-Werten initialisiert

Zustandsübersicht

Da es sich um Konstanten handelt, haben diese keine Zustandsänderungen. Die Werte sind während der gesamten Laufzeit unveränderlich.

Anwendungsszenarien

- Konfiguration von analogen Eingängen in logiBUS-Systemen
- Parametrierung von Funktionsblöcken, die analoge Eingänge benötigen
- Typsichere Referenzierung von analogen Kanälen in der Programmierung
- Fehlerbehandlung durch Verwendung der Invalid-Konstante

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu Funktionsblöcken mit dynamischem Verhalten bieten GlobalConstants:

- Keine Laufzeitberechnungen
- Keine Ereignisbehandlung
- Konstanten statt variabler Werte
- Direkte Initialisierung bei der Deklaration

Fazit

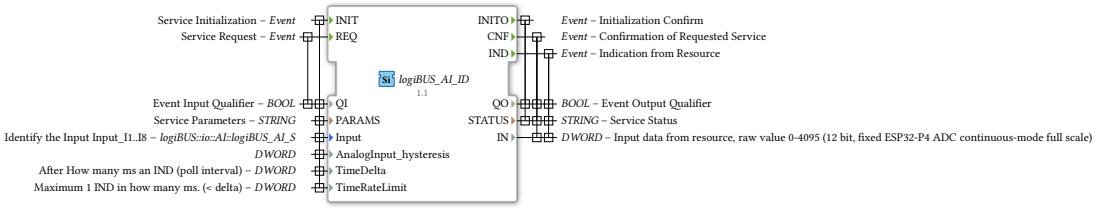
Die logiBUS_AI GlobalConstants bieten eine effiziente und typsichere Methode zur Referenzierung analoger Eingänge im logiBUS-System. Durch die vordefinierten Konstanten wird die Programmierung vereinfacht und die Fehleranfälligkeit bei der Kanalkonfiguration reduziert.

logiBUS_AI_ID

• – ○ • – ○ • – ○ •

Einleitung

Der logiBUS_AI_ID ist ein Service Interface Function Block für die Verarbeitung von Double-Word-Eingabedaten. Dieser Baustein dient als Schnittstelle für analoge Eingänge und bietet Funktionen zur Initialisierung, Abfrage und Indikation von Eingabedaten.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierungsereignis
- **REQ**: Service-Anfrageereignis

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
- **CNF**: Bestätigung der angeforderten Service-Operation
- **IND**: Indikation vom Resource

Daten-Eingänge

- **QI** (BOOL): Event Input Qualifier - Aktiviert/Deaktiviert den Service
- **PARAMS** (STRING): Service-Parameter für Konfiguration
- **Input** (logiBUS_AI_S): Identifiziert den Eingang (I1..I8) mit Initialwert „Invalid“
- **AnalogInput_hysteresis** (DWORD): Hysterese-Wert für analoge Eingänge. IND, wenn sich der Rohwert um mehr als diesen Wert ändert (0 = aus, 1 = jede Änderung)
- **TimeDelta** (DWORD): Zyklisches IND spätestens nach so vielen ms (Abfrageintervall, 0 oder 16#FFFFFFFF = aus). Initialwert 250
- **TimeRateLimit** (DWORD): Maximal ein IND pro so vielen ms (kleiner als TimeDelta). Initialwert 100

Daten-Ausgänge

- **QO** (BOOL): Event Output Qualifier - Status des Service-Ausgangs
- **STATUS** (STRING): Service-Statusinformationen

- **IN** (DWORD): Eingabedaten vom Resource — Rohwert 0-4095 (12 bit), auf dem ESP32-P4 im Continuous/DMA-ADC-Modus fix, keine andere Bit-Breite wählbar

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Function Block ermöglicht die Kommunikation mit analogen Eingabegeräten über das logiBUS-System. Bei der INIT-Initialisierung werden die Service-Parameter konfiguriert und der Eingang identifiziert. Über REQ können Daten abgefragt werden, während IND asynchrone Datenankündigungen vom Resource verarbeitet. Die Hysterese-Funktion unterstützt bei der Filterung von Signalrauschen.

Technische Besonderheiten

- **Roh-Vollausschlag von `IN`**: 0-4095 (12 bit) — der ESP32-P4-ADC-Treiber (Continuous/DMA-Modus) hat auf diesem Chip keine wählbare Bit-Breite, `SOC_ADC_DIGI_MIN_BITWIDTH` = `SOC_ADC_DIGI_MAX_BITWIDTH` = 12. Umrechnung Rohwert ⟶ Volt: `Vout = Dout × Vmax / 4096` (`Vmax` abhängig von der ADC-Dämpfung `ADC_ATTEN_DB_12`).
- Verwendet spezielle Datentypen aus dem logiBUS::io::AI-Package
- Unterstützt Hysterese-Funktionalität für analoge Signale
- Bietet sowohl synchrone (CNF) als auch asynchrone (IND) Betriebsmodi
- Initialisiert Eingänge mit einem definierten „Invalid“-Zustand

Zustandsübersicht

Der Baustein verfügt über folgende Betriebszustände:

- Nicht initialisiert (vor INIT)
- Initialisiert und bereit (nach INITO)
- Datenabfrage aktiv (bei REQ/CNF)
- Indikationsmodus (bei IND)

Anwendungsszenarien

- Industrielle Automatisierungssysteme mit analogen Sensoren
- Prozessleittechnik mit Hysterese-Anforderungen
- Embedded-Systeme mit logiBUS-Kommunikation
- Anlagen mit mehreren analogen Eingängen (I1-I8)

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen analogen Eingangsbausteinen bietet logiBUS_AI_ID erweiterte Funktionen wie Hysterese-Steuerung, detaillierte Statusrückmeldungen und eine strukturierte Initialisierungsprozedur. Die Integration in das logiBUS-System ermöglicht eine standardisierte Kommunikation.

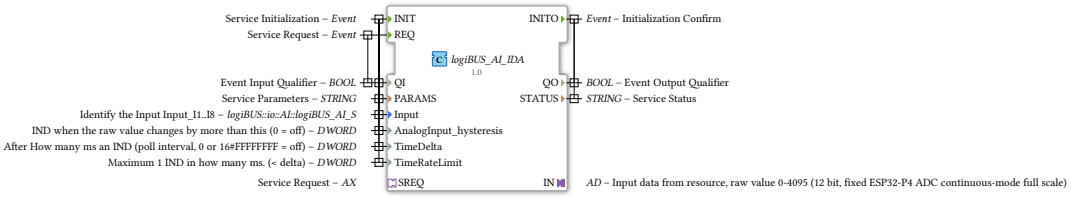
🛠️ Zugehörige Übungen

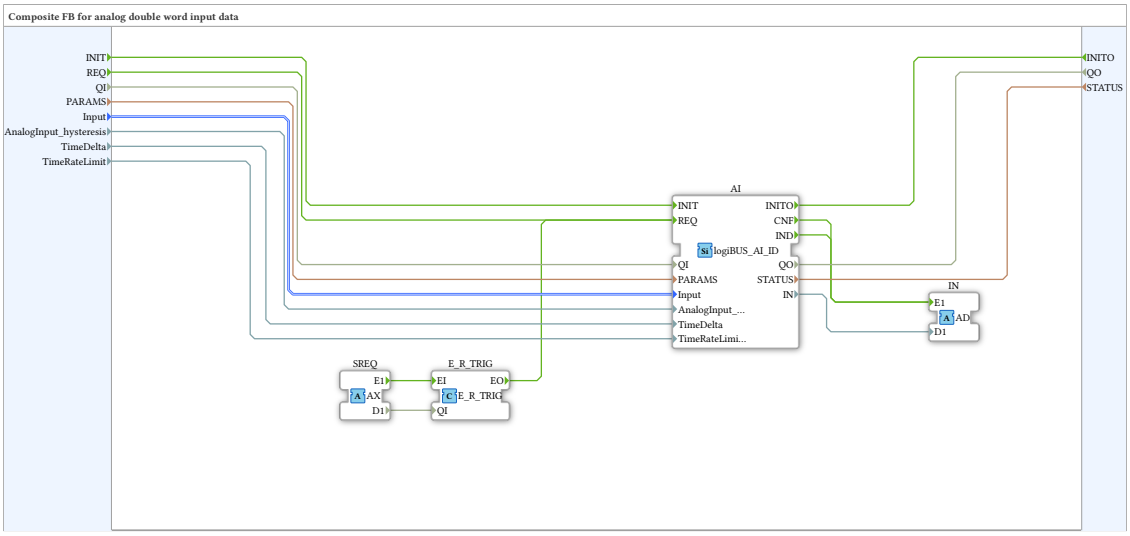
- Uebung_028
- Uebung_034

Fazit

Der logiBUS_AI_ID Function Block stellt eine robuste und flexible Lösung für die Anbindung analoger Eingabegeräte in industriellen Steuerungssystemen dar. Durch die umfangreiche Parametrierungsmöglichkeit und die integrierte Hysterese-Funktionalität eignet er sich besonders für anspruchsvolle Automatisierungsanwendungen.

logiBUS_AI_IDA





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `logiBUS_AI_IDA` ist ein zusammengesetzter Baustein (Composite FB) zur Verarbeitung analoger Doppelwort-Eingangsdaten. Er dient als Schnittstelle zwischen einer logiBUS-Ressource und der Applikation, indem er einheitliche analoge Eingangswerte über einen Adapter bereitstellt und Statusinformationen (QO, STATUS) an die aufrufende Instanz zurückmeldet. Der Baustein unterstützt sowohl initialisierungs- als auch ereignisgesteuerte Verarbeitung.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Mit	Beschreibung
INIT	EInit	QI, PARAMS, Input, AnalogInput_hysteresis, TimeDelta, TimeRateLimit	Service-Initialisierung: Konfiguration des analogen Eingangs und Start der Datenbereitstellung.
REQ	Event	QI	Service-Anforderung: Auslösen einer sofortigen Verarbeitung oder eines Status-Updates.

Ereignis-Ausgänge

Ereignis	Typ	Mit	Beschreibung
INITO	EInit	QO, STATUS	Bestätigung der Initialisierung mit Qualitäts- und Statusinformationen.

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung
QI	BOOL	–	Qualifier für Ereignisse (z. B. Freigabe der Verarbeitung).
PARAMS	STRING	–	Dienstparameter (z. B. Konfigurationsstrings).
Input	logiBUS::io::AI::logiBUS_AI_S	Invalid	Auswahl des analogen Eingangs (z. B. Input_I1…I8).

AnalogInput_hysteresis	DWORD	–	Hysterese für die Änderungserkennung. Bei Wert 0 muss TimeDelta ungleich 0 sein.
TimeDelta	DWORD	250	Zykluszeit in ms für zyklische Verarbeitung (16#FFFFFFFF = nur auf Änderung).
TimeRateLimit	DWORD	100	Mindestabstand in ms zwischen zwei Ereignissen (IND) (< TimeDelta).

Daten-Ausgänge

Name	Typ	Beschreibung
QO	BOOL	Ausgangsqualifier (z. B. gültiger Zustand nach INIT).
STATUS	STRING	Statusmeldung (z. B. Initialisierungsfehler oder OK).

Adapter

Richtung	Name	Typ	Beschreibung
Plug	IN	adapter::types::unidirectional::AD	Empfängt die analogen Eingangsdaten von der Ressource — Rohwert 0-4095 (12 bit), auf dem ESP32-P4 fix (kein Auswahlspielraum im Continuous/DMA-ADC-Modus).
Socket	SREQ	adapter::types::unidirectional::AX	Ermöglicht die externe Anforderung eines Dienstes (Service-Request).

Funktionsweise

Der Baustein kapselt den internen FB `logiBUS_AI_ID`, der die eigentliche Logik zur analogen Eingangsverarbeitung enthält. Das interne Netzwerk verbindet:

- **INIT → AI.INIT** startet die Ressourcenkonfiguration.
- **AI.INITO → INITO** gibt die Initialisierungsbestätigung zurück.
- **REQ → AI.REQ** löst eine sofortige Verarbeitung aus.
- **SREQ.E1** (externes Service-Request-Ereignis) wird über den **E_R_TRIG** (Flankenerkennung) auf **AI.REQ** geleitet – dadurch kann auch ein externer Adapter eine Verarbeitung anstoßen.
- **AI.IND** und **AI.CNF** – beide verbinden auf **IN.E1** (den Plug-Ausgang) und signalisieren nach außen, dass neue Daten am Adapter `IN` anliegen.
- Die Daten-Eingänge (QI, PARAMS, Input, Hysterese, TimeDelta, TimeRateLimit) werden direkt an den internen Baustein weitergeleitet.
- Die Ausgänge QO und STATUS kommen vom internen Baustein.

Die zyklische Verarbeitung erfolgt gemäß `TimeDelta`. Wenn `TimeDelta = 16#FFFFFFFF` gesetzt ist, wird nur bei einer Änderung des analogen Werts (unter Berücksichtigung der Hysterese) ein Ereignis erzeugt.

Technische Besonderheiten

- **Roh-Vollausschlag von `IN`**: 0-4095 (12 bit) — der ESP32-P4-ADC-Treiber (Continuous/DMA-Modus) hat auf diesem Chip keine wählbare Bit-Breite, `SOC_ADC_DIGI_MIN_BITWIDTH` = `SOC_ADC_DIGI_MAX_BITWIDTH` = 12. Umrechnung Rohwert → Volt: `Vout = Dout × Vmax / 4096` (`Vmax` abhängig von der ADC-Dämpfung `ADC_ATTEN_DB_12`).
- **Hysterese (`AnalogInput_hysteresis`)**: Ist der Wert 0, muss die Zykluszeit (`TimeDelta`) zwingend ungleich 0 sein, da sonst keine Ereignisse ausgelöst werden können.
- **Zeitsteuerung**: Mit `TimeDelta` und `TimeRateLimit` kann das Verhalten feinabgestimmt werden – z. B. zyklische Abfrage (TimeDelta > 0) oder reine Änderungsbenachrichtigung (TimeDelta = 0xFFFFFFFF).

- **Externer Service-Request**: Über den Socket `SREQ` kann eine andere Komponente (z. B. ein übergeordneter Steuerungsbaustein) ein Update anfordern.
- **Composite-Architektur**: Der Baustein ist als Composite realisiert, was eine Wiederverwendung des bewährten `logiBUS_AI_ID` ermöglicht und gleichzeitig die Schnittstelle erweitert.

Zustandsübersicht

Der Baustein selbst besitzt keinen expliziten Zustandsautomaten, da das Verhalten durch den internen FB `logiBUS_AI_ID` bestimmt wird. Folgende grundlegende Abläufe lassen sich identifizieren:

1. **Initialisierungsphase**
   - Ereignis `INIT` → interner FB wird konfiguriert → Bestätigung `INITO` mit QO/STATUS.
2. **Datenbereitstellung (zyklisch / änderungsbasiert)**
   - Nach INIT sendet der interne FB regelmäßig oder bei Wertänderung Ereignisse über `IND` (oder `CNF`) an den Plug `IN`.
3. **Manueller Request**
   - Durch `REQ` oder über `SREQ` wird die aktuelle Verarbeitung angestoßen; Ergebnisse werden ebenfalls über `IN` ausgegeben.

Anwendungsszenarien

- **Analoge Sensorerfassung**: Einloggen von analogen Sensoren (z. B. Temperatur, Druck, Füllstand) mit konfigurierbarer Hysterese und Abtastrate.
- **Überwachung mit minimaler Buslast**: Durch Setzen von `TimeDelta = 0xFFFFFFFF` werden nur bei relevanten Änderungen Ereignisse gesendet.
- **Sicherheitskritische Anwendungen**: Kombination von zyklischer Abfrage (z. B. 250 ms) mit einem schnellen Änderungsalarm (`TimeRateLimit` begrenzt die Ereignisfrequenz).
- **Steuerungsbus-Anbindung**: Der Baustein eignet sich als generische Schnittstelle für logiBUS-kompatible analoge Eingangsmodule mit einheitlichem Adapterprotokoll.

Vergleich mit ähnlichen Bausteinen

- **logiBUS_AI_ID** (interner FB): Bietet die Kernlogik, hat aber keinen zusätzlichen Socket für externe Service-Requests. `logiBUS_AI_IDA` erweitert dies durch `SREQ` und die indirekte Triggerung über `E_R_TRIG`.
- **logiBUS_AI_S** (Datenstruktur): Dient als reiner Datentyp zur Identifikation des analogen Kanals; `logiBUS_AI_IDA` verwendet ihn als Eingangsparameter.
- **Andere Composite-FBs für digitale Eingänge**: Im Unterschied zu digitalen Bausteinen liegt der Fokus auf analoger Wandlung, Hysterese und zeitgesteuerter Änderungserkennung.

Fazit

Der `logiBUS_AI_IDA` bietet eine flexible und erweiterte Schnittstelle zur Verarbeitung analoger Eingänge in logiBUS-basierten Automatisierungssystemen. Durch die Konfigurationsmöglichkeiten (Hysterese, Zeitparameter, externe Service-Requests) eignet er sich für vielfältige Anwendungen – von einfacher zyklischer Erfassung bis hin zu effizienter änderungsbasierter Kommunikation. Die Composite-Struktur ermöglicht eine klare Trennung von Kernlogik und Anpassungen der Ereignisbehandlung.

logiBUS_AI_S

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `logiBUS_AI_S` ist ein strukturierter Datentyp (Struct), der zur Konfiguration eines analogen Eingangskanals (AI - Analog Input) innerhalb der logiBUS-Bibliothek dient. Er definiert die grundlegende Parameterstruktur, die benötigt wird, um einen physischen Pin eines logiBUS-Hardwaremoduls für die Erfassung analoger Signale zu adressieren.

Schnittstellenstruktur

Dies ist ein strukturierter Datentyp (STRUCT), daher besitzt er keine Ereignis- oder Adapterschnittstellen im herkömmlichen Sinne eines ausführbaren Funktionsblocks. Seine Struktur besteht aus einem einzelnen Datenelement.

Daten-Eingänge

Die Struktur selbst enthält folgendes Datenelement, das als Eingabeparameter für einen konfigurierenden Funktionsblock (z.B. `logiBUS_AI`) dient:

- **`Pin`** (`USINT`): Dieser Wert spezifiziert den physischen Hardware-Pin oder Kanal des logiBUS-Moduls, von dem das analoge Signal gelesen werden soll. Der Standardwert ist `255`, was typischerweise als ungültiger oder nicht konfigurierter Zustand interpretiert wird.

Funktionsweise

Der Datentyp `logiBUS_AI_S` ist eine reine Datenstruktur und besitzt keine aktive Funktionsweise. Er dient als Container für Konfigurationsdaten. Seine Hauptaufgabe ist die parametrische Beschreibung eines analogen Eingangs. Ein übergeordneter IO-Manager- oder Gerätetreiber-Funktionsblock (wie `logiBUS_AI`) liest die in einer Instanz dieser Struktur gespeicherten Werte aus und verwendet sie, um die entsprechende Hardware-Schnittstelle korrekt einzurichten und anzusteuern.

Technische Besonderheiten

- **Initialwert:** Das Mitglied `Pin` ist mit dem Initialwert `255` vordefiniert. Dieser Wert liegt außerhalb des typischen Bereichs gültiger Pin-Nummern (oft 0..n) und kann daher zur Erkennung einer nicht initialisierten oder fehlerhaften Konfiguration genutzt werden.
- **Typensicherheit:** Die Verwendung eines strukturierten Datentyps anstelle einzelner, unverbundener Parameter erhöht die Typsicherheit und Wartbarkeit des Codes in der 4diac IDE.
- **Bibliothekszugehörigkeit:** Der Typ ist Teil des Pakets `logiBUS::io::AI`, was seine Zugehörigkeit zur logiBUS-IO-Bibliothek für analoge Eingänge klar kennzeichnet.

Anwendungsszenarien

Dieser Struct wird ausschließlich in Szenarien verwendet, in denen logiBUS-Hardware mit analogen Eingangsmodulen (z.B. für Spannungs- oder Strommessung) in eine 4diac FORTE-Applikation integriert wird. Typische Anwendungen sind:

- Prozessüberwachung (Temperatur, Druck, Füllstand via Sensoren)
- Erfassung von Steuerspannungen
- Konfiguration von Mehrkanal-Messsystemen, bei denen für jeden Kanal eine separate Instanz von `logiBUS_AI_S` angelegt wird.

⚖️ Vergleich mit ähnlichen Bausteinen

Innerhalb der logiBUS-Bibliothek existieren ähnliche strukturierte Datentypen für andere IO-Funktionen, wie z.B. `logiBUS_DI_S` für digitale Eingänge oder `logiBUS_AO_S` für analoge Ausgänge. Diese unterscheiden sich primär durch ihren kontextspezifischen Namen und sind für die jeweilige Art der Signalverarbeitung vorgesehen. Der grundlegende Aufbau (ein `Pin`-Feld) ist jedoch oft identisch oder sehr ähnlich.

Fazit

Der `logiBUS_AI_S`-Datentyp ist ein essentielles Konfigurationselement für die Nutzung analoger Eingänge mit logiBUS in 4diac. Als einfache, gut definierte Struktur ermöglicht er eine klare und zentrale Verwaltung der Hardware-Parameter und bildet die Grundlage für eine zuverlässige Anbindung von Analogsensoren an ein IEC 61499-Steuerungssystem.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

DI

📺 Video

- Decoding the datasheet
- Die große Migration
- Die Kunst des Lötens
- Die Welt der Normung
- Digital decisions

🎧 Podcast

- 4diac IDE: Dein „Hello World“ der Automatisierung – Das Blinking Tutorial Lokal
- 4diac IDE: Dein Open-Source-Werkzeugkasten für verteilte Industrieautomatisierung nach IEC 61499
- 4diac IDE: Wie der IEC 61499 Standard die Industrieautomatisierung revolutioniert
- 4diac-Präsentation: Zielgruppen, Struktur und Alleinstellungsmerkmal Schärfen
- Den Software-Drachen zähmen: Industrielle Automatisierung und die Zukunft der Produktion

---

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

logiBUS_DI

• – ○ • – ○ • – ○ •

Einleitung

Die logiBUS_DI GlobalConstants-Definition stellt eine Sammlung von vordefinierten Konstanten für digitale Eingänge des logiBUS-Systems bereit. Diese Konstanten definieren die Pin-Zuordnungen für verschiedene digitale Eingangskanäle und werden zur Konfiguration von logiBUS-DI-Funktionsblöcken verwendet.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereigniseingänge vorhanden - handelt sich um eine Konstantendefinition

Ereignis-Ausgänge

Keine Ereignisausgänge vorhanden - handelt sich um eine Konstantendefinition

Daten-Eingänge

Keine Dateneingänge vorhanden - handelt sich um eine Konstantendefinition

Daten-Ausgänge

Keine Datenausgänge vorhanden - handelt sich um eine Konstantendefinition

Adapter

Keine Adapter vorhanden - handelt sich um eine Konstantendefinition

Funktionsweise

Die GlobalConstants-Definition logiBUS_DI stellt strukturierte Konstanten vom Typ `logiBUS_DI_S` bereit, die jeweils einen spezifischen Pin-Wert enthalten:

- **Input_I1** bis **Input_I9**: Definiert die digitalen Eingänge 1 bis 9 mit entsprechenden Pin-Nummern
- **Invalid**: Definiert einen ungültigen Pin mit dem Wert 255

Diese Konstanten werden als Initialisierungswerte für die Konfiguration von digitalen Eingängen im logiBUS-System verwendet.

Technische Besonderheiten

- **Strukturtyp**: Alle Konstanten verwenden den strukturierten Datentyp `logiBUS_DI_S`
- **Pin-Bereich**: Gültige Pins reichen von 1-9, ungültige Konfiguration wird durch Pin 255 gekennzeichnet
- **Package-Zugehörigkeit**: Die Konstanten sind im Package `logiBUS::io::DI` organisiert
- **Initialisierung**: Alle Konstanten werden bei der Deklaration mit ihren Pin-Werten initialisiert

Zustandsübersicht

Da es sich um eine reine Konstantendefinition handelt, gibt es keine Zustandsübergänge oder Zustandsautomaten.

Anwendungsszenarien

- Konfiguration von digitalen Eingangsmodulen in logiBUS-Systemen
- Parametrierung von Funktionsblöcken, die digitale Eingänge verarbeiten
- Typsichere Pin-Zuordnungen in Steuerungsprogrammen
- Fehlerbehandlung durch Verwendung der Invalid-Konstante

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu Funktionsblöcken mit aktiver Logik handelt es sich hier um eine reine Konstantendefinition, die:

- Keine aktive Verarbeitung durchführt
- Keine Ereignisse verarbeitet
- Ausschließlich Konfigurationsdaten bereitstellt
- Zur Compile-Zeit ausgewertet wird

Fazit

Die logiBUS_DI GlobalConstants bieten eine strukturierte und typsichere Methode zur Konfiguration digitaler Eingänge im logiBUS-System. Durch die vordefinierten Konstanten wird die Fehleranfälligkeit bei der Pin-Zuordnung reduziert und die Wartbarkeit der Steuerungsprogramme verbessert.

logiBUS_DI_Events

• – ◦ • – ◦ • – ◦ •

Einleitung

Die GlobalConstants-Definition `logiBUS_DI_Events` stellt eine umfassende Sammlung von Ereigniskonstanten für digitale Eingänge (DI) im logiBUS-System bereit. Diese Konstanten definieren verschiedene Tastendruck-Ereignisse und deren eindeutige Identifikatoren, die in der Steuerungslogik für die Verarbeitung von Benutzereingaben über Taster verwendet werden.

Funktionsweise

Die GlobalConstants-Definition `logiBUS_DI_Events` enthält 14 verschiedene Ereigniskonstanten vom Typ `logiBUS_DI_Events_S`, die jeweils durch einen eindeutigen numerischen Wert (`EType`) identifiziert werden. Diese Konstanten werden verwendet, um verschiedene Zustände und Ereignisse von Tastern in der Steuerungslogik zu repräsentieren.

Technische Besonderheiten

- **Strukturtyp**: Alle Konstanten verwenden den Datentyp `logiBUS_DI_Events_S`
- **Wertebereich**: Die EType-Werte reichen von 0-12 für gültige Ereignisse, mit 255 für ungültige Ereignisse
- **Package-Zugehörigkeit**: `logiBUS::io::DI`
- **Konstantendefinition**: Alle Werte sind als globale Konstanten (CONSTANT) deklariert

Doku zu den einzelnen Events: https://components.espressif.com/components/espressif/button

Zustandsübersicht

Die Ereigniskonstanten decken folgende Zustände ab:

EType	Konstante	Beschreibung
0	BUTTON_PRESS_DOWN	Taster gedrückt
1	BUTTON_PRESS_UP	Taster losgelassen

2	BUTTON_PRESS_REPEAT	Taster wiederholt gedrückt
3	BUTTON_PRESS_REPEAT_DONE	Taster-Wiederholung beendet
4	BUTTON_SINGLE_CLICK	Einfacher Tastenklick
5	BUTTON_DOUBLE_CLICK	Doppelklick
6	BUTTON_MULTIPLE_CLICK	Mehrfachklick
7	BUTTON_LONG_PRESS_START	Langer Tastendruck beginnt
8	BUTTON_LONG_PRESS_HOLD	Langer Tastendruck wird gehalten
9	BUTTON_LONG_PRESS_UP	Langer Tastendruck beendet
10	BUTTON_PRESS_END	Tastendruck beendet
11	BUTTON_EVENT_MAX	Maximale Ereignis-ID
12	BUTTON_NONE_PRESS	Kein Tastendruck
255	Invalid	Ungültiges Ereignis

Anwendungsszenarien

- **Benutzerinterface-Steuerung**: Verarbeitung von Tasteneingaben an Bedienpanels
- **Maschinensteuerung**: Erfassung von manuellen Eingaben an industriellen Steuerungen
- **Sicherheitssysteme**: Überwachung von Not-Aus-Tastern und Sicherheitsschaltern
- **Menünavigation**: Steuerung von Display-Menüs durch Tastereingaben
- **Zustandsüberwachung**: Erkennung verschiedener Tasterbetätigungsmuster

⚖️ **Vergleich mit ähnlichen Bausteinen**

Im Gegensatz zu einfachen digitalen Eingangsbausteinen, die nur binäre Zustände (EIN/AUS) erfassen, bietet `logiBUS_DI_Events` eine erweiterte Ereigniserkennung mit:

- Zeitlich aufgelösten Ereignissen (Press/Release)
- Komplexen Interaktionsmustern (Klicks, Doppelklicks)
- Langzeit-Erkennung (Long Press)
- Wiederholungsfunktionalität

Fazit

Die `logiBUS_DI_Events` GlobalConstants-Definition stellt eine umfassende und strukturierte Basis für die Verarbeitung von Tasterereignissen in industriellen Steuerungssystemen bereit. Durch die standardisierten Ereigniscodes ermöglicht sie eine konsistente und wartbare Implementierung von Benutzerinteraktionen über digitale Eingänge.

logiBUS_DI_Events_S

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Strukturdatentyp `logiBUS_DI_Events_S` dient der Konfiguration von Ereignissen für digitale Eingänge (DI) innerhalb des logiBUS-Frameworks. Er definiert den Ereignistyp, der einem bestimmten digitalen Eingangskanal zugeordnet werden kann. Dieser Datentyp ist ein wesentlicher Bestandteil für die Parametrierung von logiBUS-E/A-Funktionsblöcken.

Schnittstellenstruktur

Dieser Baustein ist ein reiner Datentyp (Structured Type) und besitzt keine Ereignis- oder Adapterschnittstellen im herkömmlichen Sinne eines Funktionsbausteins. Seine Struktur wird als Eingangs- oder Ausgangsdatenparameter in anderen Funktionsblöcken verwendet.

Daten-Eingänge

Nicht zutreffend – Dies ist ein Datentyp.

Daten-Ausgänge

Nicht zutreffend – Dies ist ein Datentyp.

Adapter

Nicht zutreffend – Dies ist ein Datentyp.

Funktionsweise

Der Datentyp `logiBUS_DI_Events_S` kapselt eine einzelne Variable (`EType`), die einen Ereignistyp-Code repräsentiert. Der Standardwert ist auf `255` gesetzt, was typischerweise einem inaktiven oder nicht konfigurierten Zustand entspricht. Konkrete Werte (z.B. 0, 1, 2, ...) werden verwendet, um bestimmte Ereignisse (wie Flankenwechsel) zu identifizieren, die dann von übergeordneten Steuerungslogiken verarbeitet werden können.

Technische Besonderheiten

- **Datentyp:** Es handelt sich um einen strukturierten Datentyp (`STRUCT`), der in anderen Funktionsblöcken als komplexer Datenparameter genutzt wird.
- **Initialwert:** Die enthaltene Variable `EType` ist mit dem Wert `255` vorinitialisiert. Dieser Wert dient wahrscheinlich als Platzhalter oder „ungültig“-Kennung.
- **Basistyp:** Die Variable `EType` ist vom Typ `USINT` (Unsigned Short Integer, 0-255), was einen begrenzten und effizienten Wertebereich für Ereigniscodes bietet.

Zustandsübersicht

Da es sich um einen passiven Datentyp handelt, existiert keine Zustandsmaschine. Der „Zustand“ ist der gespeicherte Wert der Variable `EType`.

Anwendungsszenarien

Dieser Strukturdatentyp wird primär in Verbindung mit logiBUS-I/O-Funktionsblöcken verwendet, beispielsweise:

- Als Element in einem Array zur Konfiguration aller digitalen Eingänge eines Moduls.
- Als Parameter, um festzulegen, bei welchem physikalischen Ereignis (z.B. steigende Flanke) ein logisches Event im Steuerungsprogramm ausgelöst werden soll.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu Funktionsbausteinen (FBs) wie `E_R_TRIG` oder `F_TRIG`, die selbstständig Flankenereignisse generieren, ist `logiBUS_DI_Events_S` ein reiner Konfigurationsdatentyp. Er stellt die Information bereit, *welches* Hardware-Ereignis von einem separaten Treiber- oder Hardwareabstraktionsbaustein in ein solches logisches Flankenereignis übersetzt werden soll.

Fazit

Der `logiBUS_DI_Events_S` ist ein grundlegender Konfigurationsbaustein innerhalb der logiBUS-Architektur. Er ermöglicht eine flexible und parametrierbare Zuordnung von Hardware-Ereignissen an digitalen Eingängen zu software-seitigen Verarbeitungsroutinen. Seine einfache, auf einen `USINT` reduzierte Struktur gewährleistet Effizienz und einfache Handhabung in übergeordneten Konfigurationsstrukturen.

logiBUS_DI_S

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `logiBUS_DI_S` ist ein strukturierter Datentyp (Struct), der zur Konfiguration eines digitalen Eingangs (DI) innerhalb des logiBUS-Frameworks dient. Er definiert die grundlegenden Parameter, die notwendig sind, um einen physischen oder logischen Eingangspunkt auf einem logiBUS-kompatiblen I/O-System anzusteuern. Dieser Datentyp wird typischerweise als Eingabe für spezifische I/O-Funktionsblöcke verwendet.

Schnittstellenstruktur

Als strukturierter Datentyp besitzt `logiBUS_DI_S` keine Ereignis- oder Adapterschnittstellen im herkömmlichen Sinne. Seine Struktur besteht aus einem einzelnen Datenelement.

Daten-Eingänge

Die Struktur selbst stellt die konfigurierbaren Dateneingänge dar. Sie wird als Ganzes an einen übergeordneten Funktionsblock übergeben.

Name	Datentyp	Initialwert	Kommentar
`Pin`	`USINT` (Unsigned Short Integer)	255	Definiert die Nummer des anzusteuernden Eingangspins oder Kanals.

Daten-Ausgänge

Dieser Datentyp hat keine eigenen Datenausgänge.

Funktionsweise

Der Datentyp `logiBUS_DI_S` ist eine reine Datenhaltungsstruktur. Seine Funktionsweise beschränkt sich darauf, den Wert des Mitglieds `Pin` zu speichern. Der Initialwert `255` dient häufig als Standard- oder ungültiger Wert (z.B. „nicht konfiguriert“). Ein konkreter I/O-Funktionsblock, der diesen Struct als Eingang erwartet, liest den `Pin`-Wert aus und verwendet ihn, um die Hardware-Adressierung für den digitalen Eingang vorzunehmen.

Technische Besonderheiten

- **Datentyp `USINT`:** Die Verwendung eines `USINT` (0…255) für den Pin legt nahe, dass bis zu 256 verschiedene Eingangskanäle pro Gerät oder Modul adressierbar sind.
- **Initialwert 255:** Dieser Wert liegt außerhalb des typischen Nutzbereichs (0…n, wobei n<255) und kann von der Anwendungslogik oder dem Treiber als Indikator für eine nicht vorgenommene Konfiguration interpretiert werden.
- **StructuredType:** Als `StructuredType` ist dieser Baustein kein ausführbarer Funktionsblock, sondern ein reiner Datentyp, der in der Deklaration von Variablen und Schnittstellen anderer FBs verwendet wird.

Zustandsübersicht

Da es sich um einen passiven Datentyp handelt, besitzt `logiBUS_DI_S` keinen Zustandsautomaten.

Anwendungsszenarien

- **Konfiguration von I/O-Blöcken:** Primär dient dieser Struct als Eingabeparameter für Funktionsblöcke wie `logiBUS_DI` oder ähnliche, die den eigentlichen Lesevorgang von einem digitalen Eingang durchführen.
- **Parametrisierung über Netzwerke:** Die Struktur kann leicht in übergeordneten Steuerungen (z.B. einer SPS) instanziiert, konfiguriert (`Pin`-Nummer gesetzt) und an untergeordnete I/O-Knoten oder Treiberbausteine weitergegeben werden.
- **Strukturierte Programmierung:** Ermöglicht eine saubere Kapselung der Konfigurationsdaten für digitale Eingänge, was die Wiederverwendbarkeit und Lesbarkeit des Codes erhöht.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Einfache `USINT` Variable:** Anstelle eines Structs könnte direkt eine `USINT`-Variable verwendet werden. Der Struct bietet jedoch semantische Klarheit (der Name `logiBUS_DI_S` beschreibt den Zweck) und ist leichter erweiterbar, falls zukünftig weitere Konfigurationsparameter (wie Filterzeit, Invertierung) hinzugefügt werden müssen.
- **Gerätespezifische Structs:** Innerhalb des logiBUS-Ökosystems könnte es ähnliche Structs für andere Funktionen geben (z.B. `logiBUS_DO_S` für digitale Ausgänge, `logiBUS_AI_S` für analoge Eingänge). Diese würden eine einheitliche Konfigurationsphilosophie über verschiedene I/O-Typen hinweg ermöglichen.

🛠️ Zugehörige Übungen

- Uebung_003a_sub
- IX_TO_Aux_QX
- IXA_TO_Aux_QXA
- IX_TO_logiBUS_LED_strip_QX

Fazit

Der `logiBUS_DI_S` ist ein grundlegender und spezifischer Konfigurationsbaustein für die logiBUS-I/O-Architektur. Als strukturierter Datentyp bietet er eine standardisierte und klare Methode, um die Zieladresse (Pin) für einen digitalen Eingang zu definieren. Seine Stärke liegt in der Einfachheit und der Vorbereitung auf mögliche Erweiterungen, was ihn zu einem nützlichen Bestandteil für die strukturierte Konfiguration von Feldbussystemen in 4diac FORTE macht.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

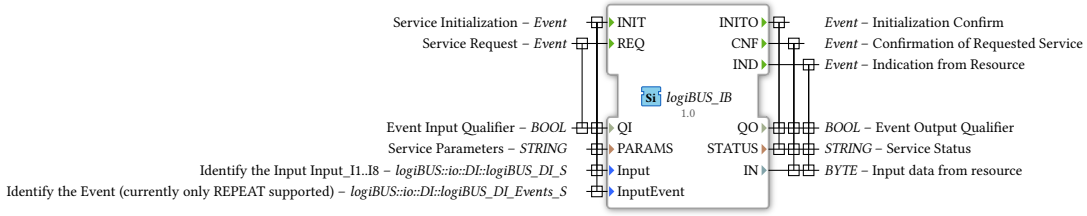
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

logiBUS_IB

• – ◦ • – ◦ • – ◦ •

Einleitung

Der logiBUS_IB ist ein Service Interface Function Block für die Verarbeitung von Byte-Eingabedaten. Dieser Baustein dient als Eingabeschnittstelle für logiBUS-Systeme und ermöglicht die Kommunikation mit digitalen Eingängen. Er unterstützt spezielle Ereignisbehandlungen wie Tastendruck-Wiederholungen und bietet eine standardisierte Schnittstelle für die Initialisierung und den Betrieb von Eingabegeräten.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierungsereignis
  - Verknüpft mit: QI, PARAMS, Input, InputEvent
- **REQ**: Service-Anfrageereignis
  - Verknüpft mit: QI

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
  - Verknüpft mit: QO, STATUS
- **CNF**: Bestätigung der angeforderten Service-Operation
  - Verknüpft mit: QO, STATUS, IN
- **IND**: Indikation vom Ressourcen-Interface
  - Verknüpft mit: QO, STATUS, IN

Daten-Eingänge

- **QI** (BOOL): Event Input Qualifier - Steuert die Ausführung der Service-Operationen
- **PARAMS** (STRING): Service-Parameter für die Konfiguration
- **Input** (logiBUS_DI_S): Identifiziert den Eingang (I1..I8), Initialwert: Invalid
- **InputEvent** (logiBUS_DI_Events_S): Identifiziert das Ereignis (aktuell nur REPEAT unterstützt), Initialwert: Invalid

Daten-Ausgänge

- **QO** (BOOL): Event Output Qualifier - Status der Service-Operation
- **STATUS** (STRING): Service-Statusinformationen
- **IN** (BYTE): Eingabedaten von der Ressource

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der logiBUS_IB Function Block arbeitet als Eingabeservice-Interface für logiBUS-Systeme. Bei der Initialisierung (INIT-Event) werden die Service-Parameter konfiguriert und die Eingänge definiert. Der Baustein kann sowohl auf explizite Anfragen (REQ-Event) als auch auf asynchrone Ereignisse von der Hardware reagieren. Besondere Unterstützung bietet er für wiederholte Tastendruck-Ereignisse (BUTTON_PRESS_REPEAT).

Technische Besonderheiten

- Unterstützt spezielle Ereignistypen wie BUTTON_PRESS_REPEAT und BUTTON_PRESS_REPEAT_DONE
- Byte-basierte Dateneingabe (8-Bit)
- Flexible Eingangskonfiguration über PARAMS-String
- Separate Ereignisausgänge für Bestätigungen (CNF) und Hardware-Indikationen (IND)
- Initialwert „Invalid“ für nicht konfigurierte Eingänge

Zustandsübersicht

Der Baustein verfügt über folgende Betriebszustände:

- **Nicht initialisiert**: Vor INIT-Event
- **Initialisiert**: Nach erfolgreicher INIT-Operation
- **Bereit**: Für REQ-Anfragen und IND-Ereignisse
- **Fehlerzustand**: Bei fehlerhaften Operationen (über STATUS meldbar)

Anwendungsszenarien

- Digitale Eingabebaugruppen mit 8 Eingängen
- Tastatur-Interface mit Wiederholfunktion
- Allgemeine digitale Eingabeschnittstellen in logiBUS-Systemen
- Hardware-nahe Eingabeoperationen mit Event-Behandlung

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Eingabebausteinen bietet logiBUS_IB erweiterte Service-Funktionalität mit:

- Parametrierbarer Konfiguration
- Ereignisbasierter Kommunikation
- Statusrückmeldungen
- Spezieller Unterstützung für Tastendruck-Wiederholungen

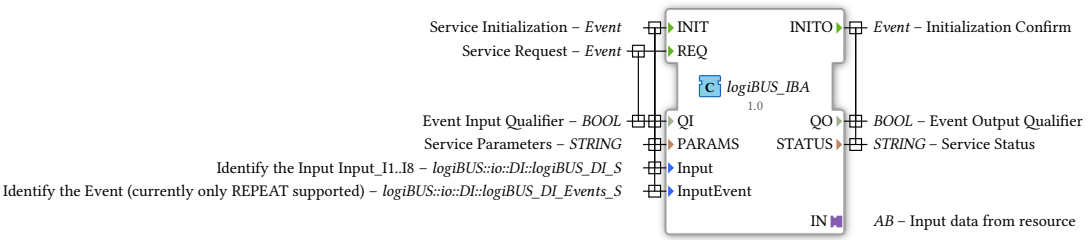
🛠️ Zugehörige Übungen

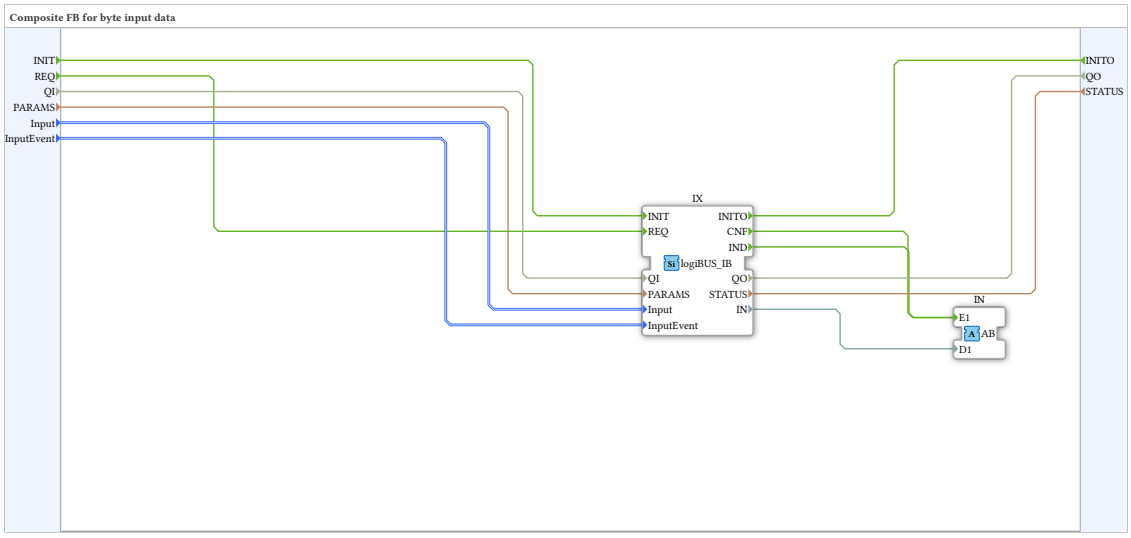
- Uebung_006c
- Uebung_011a

Fazit

Der logiBUS_IB Function Block stellt eine leistungsfähige und flexible Lösung für Byte-Eingabedaten in logiBUS-Systemen dar. Durch seine Service-orientierte Architektur und die Unterstützung spezieller Ereignistypen eignet er sich besonders für Anwendungen, die erweiterte Eingabefunktionalitäten wie Tastendruck-Wiederholungen benötigen.

logiBUS_IBA





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **logiBUS_IBA** ist ein zusammengesetzter Baustein (Composite FB) zur Verarbeitung von Byte-Eingabedaten über das logiBUS-Protokoll. Er kapselt die Initialisierung, Konfiguration und Ereignissteuerung eines logiBUS-Eingangsbausteins und stellt die Daten über einen Adapter zur Verfügung. Der Baustein ist für den Einsatz in Steuerungssystemen der Agrartechnik konzipiert.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung
INIT	EInit	Service-Initialisierung; übernimmt die Konfigurationsparameter (QI, PARAMS, Input, InputEvent)
REQ	Event	Service-Anforderung zur Ausführung einer Funktion (abhängig von QI)

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung
INITO	EInit	Initialisierungsbestätigung; signalisiert erfolgreiche oder fehlgeschlagene Initialisierung (QO, STATUS)

Daten-Eingänge

Variable	Typ	Beschreibung
QI	BOOL	Qualifizierer für Ereigniseingänge; steuert die Aktivierung der Ereignisverarbeitung
PARAMS	STRING	Service-Parameter (z. B. Konfigurationsdaten für den logiBUS-Treiber)
Input	logiBUS::io::DI::logiBUS_DI_S	Auswahl des physischen Eingangs (z. B. Input_I1..I8); Standardwert: `logiBUS_DI::Invalid`
InputEvent	logiBUS::io::DI::logiBUS_DI_Events_S	Auswahl des Ereignistyps (derzeit nur `REPEAT` unterstützt); Standardwert: `logiBUS_DI_Events::Invalid`

Daten-Ausgänge

Variable	Typ	Beschreibung
QO	BOOL	Qualifizierer für Ereignisausgänge; zeigt den Erfolg der Operation an
STATUS	STRING	Statusmeldung (z. B. Fehler- oder Erfolgsmeldung)

Adapter

Adapter	Typ	Beschreibung
IN	adapter::types::unidirectional::AB	Unidirektionaler Adapter zur Bereitstellung der Eingabedaten an die übergeordnete Ressource

Funktionsweise

Der Baustein **logiBUS_IBA** arbeitet intern mit einer Instanz des Bausteins **logiBUS_IB** (aus der Bibliothek `logiBUS::io::DI`). Die Ereignis- und Datenpfade sind wie folgt verknüpft:

- Beim **INIT**-Ereignis werden die Eingänge `QI`, `PARAMS`, `Input` und `InputEvent` an die entsprechende Initialisierung des internen Bausteins weitergeleitet. Nach Abschluss der Initialisierung wird das Ausgangsereignis **INITO** gesetzt und die Ausgänge `QO` und `STATUS` aktualisiert.
- Beim **REQ**-Ereignis wird der interne Baustein zur Verarbeitung eines Service-Requests angestoßen. Die Verarbeitung hängt vom aktuellen Zustand und den Eingangsparametern ab.
- Die vom logiBUS-Eingang gelesenen Daten werden über die Adapterverbindung (`IN.E1` und `IN.D1`) an den Ausgangsadapter **IN** übergeben. Der Adapter stellt die Daten einem übergeordneten Baustein oder der Ressource bereit.
- Fehler- und Statussignale werden über `QO` und `STATUS` rückgemeldet.

Technische Besonderheiten

- **Composite FB:** Der Baustein kapselt die komplexe Initialisierung und Datenbereitstellung eines logiBUS-Eingangs in einer einheitlichen Schnittstelle.
- **Adapterbasierte Datenausgabe:** Die Eingabedaten werden nicht über separate Datenausgänge, sondern über einen **unidirektionalen Adapter** (Typ `AB`) ausgegeben. Dies ermöglicht eine flexible Kopplung mit anderen Bausteinen.
- **Konfigurierbare Eingänge:** Über die Parameter `Input` und `InputEvent` können der physische Eingang (I1..I8) und das Ereignisverhalten (z. B. Wiederholung) ausgewählt werden.
- **Lizenz und Herkunft:** Der Baustein wird unter der **Eclipse Public License 2.0** bereitgestellt und ist für den Einsatz in der Agrartechnik optimiert.

Zustandsübersicht

Der Baustein besitzt keine explizit dargestellte Zustandsmaschine, da es sich um einen Composite FB handelt. Das interne Verhalten wird durch den enthaltenen Baustein **logiBUS_IB** bestimmt, der typischerweise folgende Zustände implementiert:

- **IDLE** – Warten auf INIT oder REQ
- **INIT** – Initialisierung mit Parametern
- **RUN** – Betriebsbereit, Datenverarbeitung aktiv
- **ERROR** – Fehlerzustand (Statusmeldung über STATUS)

Anwendungsszenarien

- **Landwirtschaftliche Steuerungen:** Erfassung digitaler Eingangssignale (z. B. Sensoren, Schalter) über logiBUS und Weiterleitung an eine Steuerung.
- **Eingabemodule in Feldbus-Systemen:** Der Baustein dient als standardisierte Schnittstelle für logiBUS-Eingänge in IEC 61499-2-Anwendungen.
- **Prototypen und Testaufbauten:** Dank der Konfigurierbarkeit kann der Baustein schnell an verschiedene Eingangskonfigurationen angepasst werden.

Vergleich mit ähnlichen Bausteinen

- **logiBUS_IB:** Der innere Baustein ist ein reiner Basic FB; **logiBUS_IBA** fügt eine Adapter-Ausgabe und eine vereinfachte Ereignissteuerung hinzu.

- **SYNC/D_SYNC:** Diese Bausteine aus der Sync-Bibliothek bieten ähnliche Eingangskonfiguration, jedoch ohne spezifische logiBUS-Protokollunterstützung.
- **Eingangsadapter generisch:** Im Vergleich zu generischen Adaptern bietet **logiBUS_IBA** eine vordefinierte logiBUS-Konfiguration, was den Entwicklungsaufwand reduziert.

Fazit

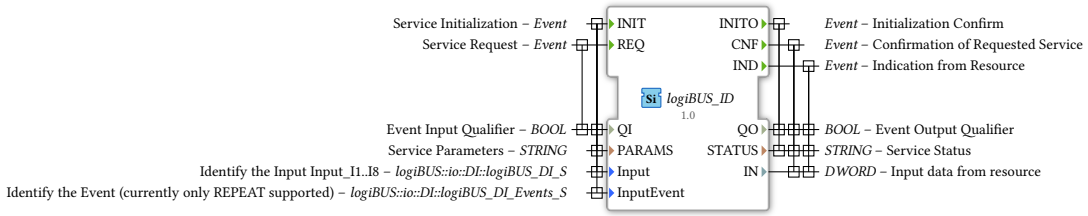
Der **logiBUS_IBA** ist ein praktischer Composite-FB für die Integration von logiBUS-Eingängen in IEC 61499-basierte Automatisierungssysteme. Durch die Kombination von Initialisierung, Parameterkonfiguration und Adapter-basierter Datenausgabe wird eine saubere und wiederverwendbare Schnittstelle geschaffen. Er eignet sich besonders für Anwendungen in der Agrartechnik, die auf das logiBUS-Protokoll setzen.

logiBUS_ID

• – ◦ • – ◦ • – ◦ •

Einleitung

Der logiBUS_ID Funktionsblock ist ein Eingabe-Service-Interface-Baustein für Doppelwort-Eingabedaten (DWORD). Er dient als Schnittstelle für die Kommunikation mit logiBUS-Eingabegeräten und ermöglicht die Verarbeitung von digitalen Eingangssignalen in 4diac-Steuerungssystemen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung - Initialisiert den Funktionsblock mit Parametern und Konfiguration
- **REQ**: Service-Anfrage - Löst eine Abfrage der aktuellen Eingabedaten aus

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung - Bestätigt die erfolgreiche Initialisierung
- **CNF**: Bestätigung der angeforderten Service - Bestätigt eine REQ-Anfrage
- **IND**: Indikation von der Ressource - Signalisiert asynchrone Ereignisse vom Eingabegerät

Daten-Eingänge

- **QI**: Event-Input-Qualifier (BOOL) - Aktiviert/Deaktiviert den Service
- **PARAMS**: Service-Parameter (STRING) - Konfigurationsparameter für den Service
- **Input**: Eingabe-Identifikation (logiBUS_DI_S) - Identifiziert den Eingang (I1..I8)
- **InputEvent**: Ereignis-Identifikation (logiBUS_DI_Events_S) - Identifiziert das Ereignis (aktuell nur REPEAT unterstützt)

Daten-Ausgänge

- **QO**: Event-Output-Qualifier (BOOL) - Status des Service-Ausgangs
- **STATUS**: Service-Status (STRING) - Statusinformationen über den Service
- **IN**: Eingabedaten von der Ressource (DWORD) - Aktuelle Eingabedaten vom Gerät

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der logiBUS_ID Funktionsblock dient als Bridge zwischen dem 4diac-System und logiBUS-Eingabegeräten. Bei der Initialisierung (INIT) werden die Geräteparameter konfiguriert. Der Block kann sowohl synchrone Abfragen (REQ/CNF) als auch asynchrone Ereignisse (IND) verarbeiten. Die Eingabedaten werden als 32-Bit-Doppelwort (DWORD) bereitgestellt.

Technische Besonderheiten

- Unterstützt Doppelwort-Eingabedaten (32-Bit)
- Verwendet spezifische logiBUS-Datentypen für Eingabe- und Ereigniskonfiguration
- Bietet sowohl synchrone als auch asynchrone Kommunikationsmodi
- Initialwerte für Eingabeparameter sind als „Invalid“ definiert

Zustandsübersicht

Der Funktionsblock verfügt über folgende Betriebszustände:

- Nicht initialisiert (vor INIT)
- Initialisiert und bereit (nach erfolgreichem INITO)
- Abfrage-Modus (bei REQ/CNF-Zyklus)
- Ereignis-Modus (bei IND-Signalisierung)

Anwendungsszenarien

- Anbindung von logiBUS-Eingabegeräten in Automatisierungssystemen
- Verarbeitung digitaler Eingangssignale in Steuerungsanwendungen
- Integration von Tasteneingaben (BUTTON_PRESS_UP, BUTTON_LONG_PRESS_HOLD, BUTTON_LONG_PRESS_UP)
- Überwachung von Zustandsänderungen an digitalen Eingängen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu standardmäßigen digitalen Eingangsblöcken bietet logiBUS_ID:

- Spezifische logiBUS-Integration
- Erweiterte Ereignisbehandlung
- Konfigurierbare Parameter über STRING-Params
- Asynchrone Ereignisindikation

🛠️ Zugehörige Übungen

- Uebung_011a2

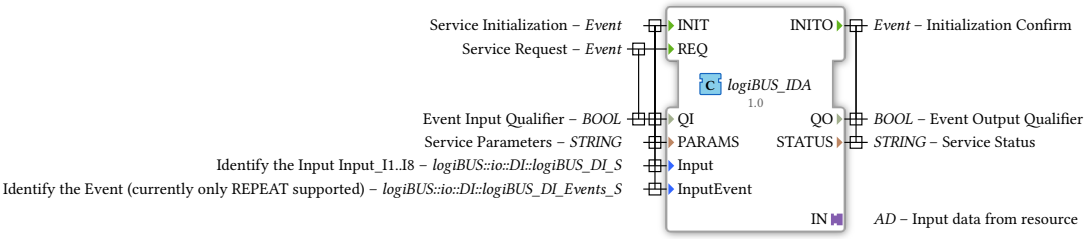
Fazit

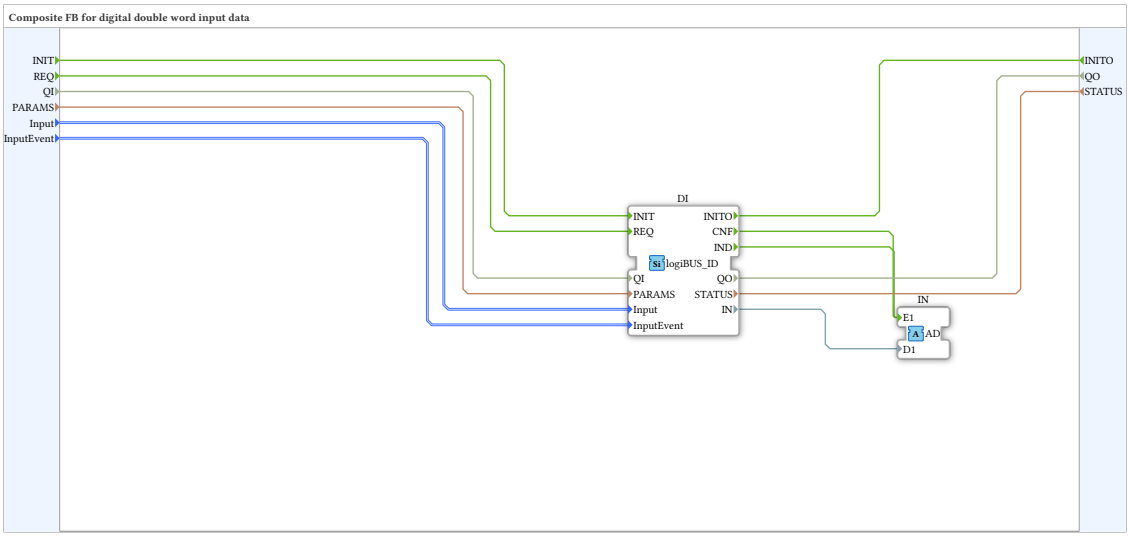
Der logiBUS_ID Funktionsblock stellt eine spezialisierte Schnittstelle für logiBUS-Eingabegeräte in 4diac-Systemen dar. Durch seine flexible Konfiguration und Unterstützung verschiedener Betriebsmodi eignet er sich ideal für Anwendungen, die eine zuverlässige Verarbeitung digitaler Eingangssignale mit erweiterter Ereignisbehandlung erfordern.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

logiBUS_IDA





• – ○ • – ○ • – ○ •

Einleitung

Der **logiBUS_IDA** ist ein Composite-Funktionsblock (CFB) zur digitalen Doppelwort-Eingangsverarbeitung. Er kapselt einen internen **logiBUS_ID**-Baustein und stellt eine einheitliche Schnittstelle zur Initialisierung, Parametrierung und ereignisgesteuerten Datenabfrage bereit. Der Block ist für den Einsatz in logiBUS-basierten Automatisierungssystemen vorgesehen und unterstützt die Auswahl eines Eingangskanals sowie die Konfiguration des Wiederholereignisses.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung
INIT	EInit	Service-Initialisierung; erwartet gültige Parameter (QI, PARAMS, Input, InputEvent)
REQ	Event	Service-Anforderung; triggert eine erneute Datenabfrage am internen Baustein

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung
INITO	EInit	Initialisierungsbestätigung; gibt den Status (QO, STATUS) zurück

Daten-Eingänge

Variable	Typ	Beschreibung
QI	BOOL	Qualifizierer für den Ereigniseingang; steuert die Ausführung
PARAMS	STRING	Service-Parameter (z. B. Konfigurationszeichenfolge)
Input	logiBUS_DI_S	Auswahl des digitalen Eingangskanals (I1 … I8); Vorgabewert: *Invalid*
InputEvent	logiBUS_DI_Events_S	Auswahl des Ereignistyps (aktuell nur *REPEAT* unterstützt); Vorgabewert: *Invalid*

Daten-Ausgänge

Variable	Typ	Beschreibung

QO	BOOL	Qualifizierer des Ausgangsereignisses
STATUS	STRING	Dienststatus (z. B. Fehlermeldungen oder Betriebszustand)

Adapter

Adapter	Typ	Beschreibung
IN	adapter::types::unidirectional::AD	Unidirektionaler Eingangsadapter zur Aufnahme der digitalen Doppelwortdaten aus der Ressource

Funktionsweise

Der **logiBUS_IDA** ist als Composite-Baustein realisiert, der einen einzelnen **logiBUS_ID**-Funktionsblock einkapselt.
Die Initialisierung erfolgt über das Ereignis **INIT**. Dabei werden die Parameter *QI*, *PARAMS*, *Input* und *InputEvent* an den inneren Baustein weitergeleitet. Der innere Baustein konfiguriert daraufhin den gewählten digitalen Eingangskanal und bereitet die Datenaufnahme vor. Nach erfolgreicher Initialisierung wird **INITO** mit den Ausgangswerten *QO* und *STATUS* ausgegeben.

Ein zyklischer oder einmaliger Lesevorgang wird über das Ereignis **REQ** ausgelöst. Der interne Baustein liest den aktuellen Wert des konfigurierten Eingangs (8-Bit / Doppelwort) über den Adapter **IN** und stellt das Ergebnis intern zur Weiterverarbeitung bereit. Die tatsächlichen Datenwerte werden nicht direkt als Ausgangsvariablen des **logiBUS_IDA** geführt; sie stehen ausschließlich über den Adapter **IN** am Ausgang des internen Bausteins zur Verfügung (siehe FBNetwork-Verbindungen: DI.IN → IN.D1).

Technische Besonderheiten

- **Composite-Architektur**: Kapselt komplexe Logik in einem wiederverwendbaren Baustein.
- **Ereignisgesteuerte Verarbeitung**: Trennung von Initialisierung (INIT) und Betriebsanforderung (REQ).
- **Adapterbasierte Ein-/Ausgabe**: Die digitale Doppelwortinformation wird über einen unidirektionalen Adapter übertragen, was die Kopplung an die Ressource flexibel hält.
- **Konfigurierbare Parameter**: Der Eingangskanal (I1 … I8) und das Ereignisverhalten (z. B. REPEAT) können zur Laufzeit festgelegt werden.
- **Standardkonformer IEC 61499-2**: Der Baustein ist gemäß dem Standard modelliert und verwendet die Typsicherheit durch vordefinierte Strukturen (logiBUS_DI_S und logiBUS_DI_Events_S).

Zustandsübersicht

Der Baustein durchläuft folgende Hauptzustände:

1. **Initialisierung (INIT)**
   - Eintritt: Ereignis **INIT** wird aktiviert.
   - Aktion: Innere Baustein wird parametriert und gestartet.
   - Ausgang: **INITO** mit QO/STATUS.
2. **Betrieb (REQ)**
   - Eintritt: Ereignis **REQ** wird aktiviert.
   - Aktion: Datenabfrage am inneren Baustein; Wert wird über Adapter **IN** bereitgestellt.
   - Zyklische Wiederholung möglich, sofern *InputEvent* auf *REPEAT* gesetzt ist.

Fehlerzustände (z. B. ungültige Parameter oder Verbindungsprobleme) werden über **STATUS** signalisiert.

Anwendungsszenarien

- **Digitaler Mehrfacheingang in der Landtechnik**: Erfassung von bis zu 8 digitalen Sensoren (z. B. Endschalter, Näherungsschalter) und Zusammenfassung als Doppelwort.
- **Feldbusanbindung**: Einsatz in logiBUS-Netzwerken zur dezentralen E/A-Erfassung.
- **Parametrierbare Überwachung**: Konfiguration des Abtastverhaltens (Ereignisart) je nach Anforderung der Steuerung.
- **Erweiterte Diagnose**: Der STATUS-Ausgang erlaubt die Rückmeldung von Fehlern oder Betriebszuständen an die übergeordnete Steuerung.

Vergleich mit ähnlichen Bausteinen

Gegenüber einem einfachen **logiBUS_DI** bietet der **logiBUS_IDA** eine erweiterte Schnittstelle mit expliziter Initialisierung (INIT) und ereignisgesteuerten Anforderungen (REQ). Während ein reiner **logiBUS_ID**-Baustein direkt die Rohdaten über Adapter liefert, integriert der **IDA** die Konfiguration von Kanal und Ereignisverhalten in einem kompakten Block. Ähnliche Composite-Bausteine aus anderen Feldbussystemen (z. B. CANopen, PROFIBUS) realisieren vergleichbare Funktionalitäten oft prozessornah, während der **logiBUS_IDA** speziell auf die logiBUS-Protokollumgebung abgestimmt ist.

Fazit

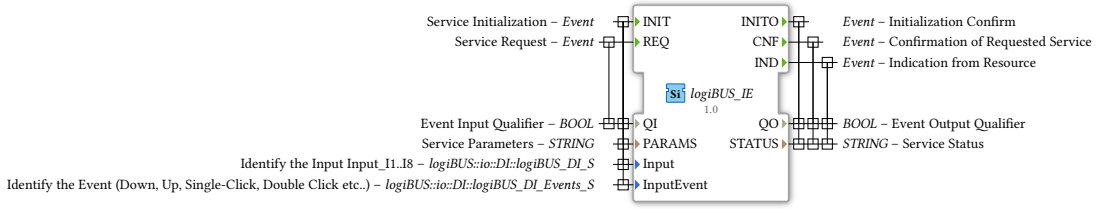
Der **logiBUS_IDA** ist ein flexibler Composite-Funktionsblock zur digitalen Doppelwort-Eingangsverarbeitung in IEC 61499-Umgebungen. Er vereinfacht die Inbetriebnahme durch seine klare Schnittstelle mit Initialisierungs- und Anforderungsereignissen und ermöglicht eine kanal- und ereigniskonfigurierte Datenaufnahme. Dank des Adapter-basierten Datentransfers lässt er sich nahtlos in vorhandene logiBUS-Ressourcen integrieren. Der Baustein eignet sich besonders für Anwendungen, die eine sichere Initialisierung und wiederholte, parametrierbare Abfragen digitaler Eingänge erfordern.

logiBUS_IE

• – ○ • – ○ • – ○ •

Einleitung

Der logiBUS_IE Funktionsblock ist ein Eingabeservice-Schnittstellenbaustein für Ereigniseingabedaten. Er dient als Interface zwischen dem logiBUS-System und der 4diac-Laufzeitumgebung zur Verarbeitung von digitalen Eingangssignalen mit erweiterten Ereigniserkennungsfunktionen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung
  - Verknüpft mit: QI, PARAMS, Input, InputEvent
- **REQ**: Service-Anfrage
  - Verknüpft mit: QI

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
  - Verknüpft mit: QO, STATUS
- **CNF**: Bestätigung der angeforderten Services
  - Verknüpft mit: QO, STATUS
- **IND**: Indikation von der Ressource
  - Verknüpft mit: QO, STATUS

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangsqualifizierer
- **PARAMS** (STRING): Service-Parameter
- **Input** (logiBUS_DI_S): Identifiziert den Eingang (I1..I8)
  - Initialwert: Invalid
- **InputEvent** (logiBUS_DI_Events_S): Identifiziert das Ereignis (Down, Up, Single-Click, Double-Click etc.)
  - Initialwert: Invalid

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifizierer
- **STATUS** (STRING): Service-Status

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Funktionsblock ermöglicht die Verarbeitung digitaler Eingangssignale mit erweiterter Ereigniserkennung. Über die INIT-Initialisierung können spezifische Parameter konfiguriert werden. Der Baustein erkennt verschiedene Eingabeereignisse wie Tastendrücke, Loslassen, Einzelklicks und Doppelklicks und signalisiert diese über die entsprechenden Ausgangsereignisse.

Technische Besonderheiten

- Unterstützt multiple Ereignistypen (Down, Up, Single-Click, Double-Click)
- Flexible Eingangskonfiguration über PARAMS-String
- Statusrückmeldung über STATUS-Ausgang
- Initialisierung mit Invalid-Werten für sicheren Start

Zustandsübersicht

Der Baustein verfügt über einen initialisierten und einen Betriebszustand. Nach erfolgreicher INIT-Initialisierung wechselt er in den Betriebszustand, in dem Ereignisverarbeitung und Service-Anfragen verarbeitet werden können.

Anwendungsszenarien

- Bedienpanel-Steuerungen
- Tastatureingabe-Verarbeitung
- Mechanische Schalter-Überwachung
- Benutzerinteraktionssysteme
- Industrielle Steuerungstaster

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Eingangsbausteinen bietet logiBUS_IE erweiterte Ereigniserkennungsfunktionen und kann komplexe Eingabemuster wie Doppelklicks erkennen. Die parametrierbare Konfiguration ermöglicht eine flexible Anpassung an verschiedene Hardware-Schnittstellen.

🛠️ Zugehörige Übungen

- Uebung_004a
- Uebung_004a2
- Uebung_004a2_2
- Uebung_004a2_3
- Uebung_004a2_AX
- Uebung_004a3
- Uebung_004a3_AX
- Uebung_004a4
- Uebung_004a4_AX
- Uebung_004a5
- Uebung_004a5_AX
- Uebung_004a6
- Uebung_004a6_AX
- Uebung_004a7
- Uebung_004a7_AX
- Uebung_004a8
- Uebung_004a8_AX
- Uebung_004a9
- Uebung_004a9_AX
- Uebung_004a_AX
- Uebung_004b
- Uebung_004b2
- Uebung_004b3
- Uebung_004b_AX
- Uebung_004b_AX_ASR

- Uebung_004b_AX_ASR_X
- Uebung_004c1
- Uebung_004c1_AX
- Uebung_004c2
- Uebung_004c2_AX
- Uebung_004c3
- Uebung_004c3_AX
- Uebung_004c4
- Uebung_004c4_AX
- Uebung_004c5
- Uebung_004c5_AX
- Uebung_006
- Uebung_006_AX
- Uebung_006a
- Uebung_006a2
- Uebung_006a2_AX
- Uebung_006a3
- Uebung_006a3_AX
- Uebung_006a4
- Uebung_006a4_AX
- Uebung_006a_AX
- Uebung_006b
- Uebung_006b_AX
- Uebung_006d
- Uebung_006d_AX
- Uebung_007a1
- Uebung_007a1_AX
- Uebung_007a2
- Uebung_007a2_AX
- Uebung_007a3
- Uebung_007a3_AX
- Uebung_009a
- Uebung_019
- Uebung_019a
- Uebung_019b
- Uebung_019c
- Uebung_020f3
- Uebung_020f3_AX
- Uebung_020h
- Uebung_020h_AX
- Uebung_020i
- Uebung_020i_AX
- Uebung_031
- Uebung_034b
- Uebung_035
- Uebung_035a
- Uebung_035a1_AX
- Uebung_035a1b_AX
- Uebung_035a2
- Uebung_035a2_AX
- Uebung_035a3
- Uebung_035a3_AX
- Uebung_035b
- Uebung_035c
- Uebung_036
- Uebung_037
- Uebung_038
- Uebung_038_AX
- Uebung_039

- Uebung_039a
- Uebung_040
- Uebung_040_2
- Uebung_040_AX
- Uebung_041
- Uebung_042
- Uebung_043
- Uebung_080
- Uebung_080b
- Uebung_080c
- Uebung_081
- Uebung_082
- Uebung_083
- Uebung_083_AX
- Uebung_084
- Uebung_085
- Uebung_087
- Uebung_087a1
- Uebung_091
- Uebung_093
- Uebung_093b
- Uebung_094
- Uebung_094a
- Uebung_094a_AX
- Uebung_095
- Uebung_095_AX
- Uebung_110
- Uebung_111
- Uebung_124
- Uebung_127
- Uebung_128
- Uebung_128b
- Uebung_132
- Uebung_150_AX
- Uebung_151_AX
- Uebung_152
- Uebung_153
- Uebung_160b
- Uebung_160b_AX
- Uebung_171_AX
- Uebung_179
- Uebung_180

Fazit

Der logiBUS_IE Funktionsblock stellt eine leistungsstarke Schnittstelle für digitale Eingabegeräte mit erweiterter Ereigniserkennung bereit. Seine flexible Konfigurierbarkeit und die Unterstützung verschiedener Ereignistypen machen ihn ideal für Anwendungen, die komplexe Benutzerinteraktionen erfordern.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

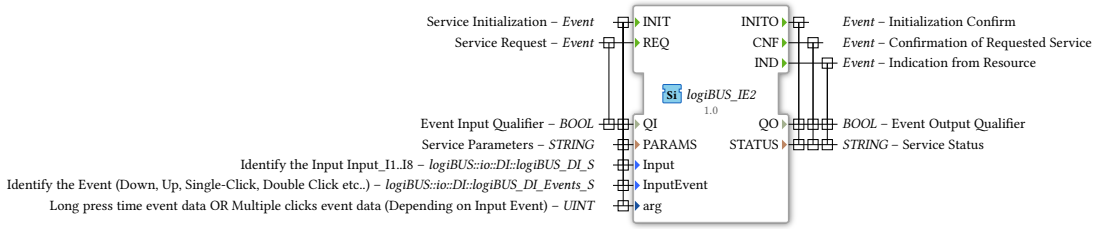
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

logiBUS_IE2

• – ◦ • – ◦ • – ◦ •

Einleitung

Der logiBUS_IE2 ist ein Eingabeservice-Schnittstellen-Funktionsblock für Ereigniseingabedaten. Er dient als Interface für die Verarbeitung von Eingabeereignissen in logiBUS-Systemen und ermöglicht die Verarbeitung verschiedener Eingabeereignistypen wie Tastendrücke, Klicks und zeitbasierte Ereignisse.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (Service Initialization) - Initialisiert den Service mit folgenden Daten:
  - QI
  - PARAMS
  - Input
  - InputEvent
  - arg
- **REQ** (Service Request) - Fordert einen Service an mit:
  - QI

Ereignis-Ausgänge

- **INITO** (Initialization Confirm) - Bestätigt die Initialisierung mit:
  - QO
  - STATUS
- **CNF** (Confirmation of Requested Service) - Bestätigt den angeforderten Service mit:
  - QO
  - STATUS
- **IND** (Indication from Resource) - Meldung von der Ressource mit:
  - QO
  - STATUS

Daten-Eingänge

- **QI** (BOOL) - Event Input Qualifier
- **PARAMS** (STRING) - Service Parameters
- **Input** (logiBUS_DI_S) - Identifiziert den Eingang Input_I1..I8 (Initialwert: Invalid)
- **InputEvent** (logiBUS_DI_Events_S) - Identifiziert das Ereignis (Down, Up, Single-Click, Double Click etc.) (Initialwert: Invalid)
- **arg** (UINT) - Langdruck-Zeitereignisdaten ODER Mehrfachklick-Ereignisdaten (abhängig vom Input Event) (Initialwert: 65535)

Daten-Ausgänge

- **QO** (BOOL) - Event Output Qualifier
- **STATUS** (STRING) - Service Status

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der logiBUS_IE2 Funktionsblock verarbeitet Eingabeereignisse von logiBUS-Eingängen und wandelt diese in standardisierte Ereignismeldungen um. Er unterstützt verschiedene Ereignistypen wie Tastendrücke (Down/Up), Einzelklicks, Doppelklicks und zeitbasierte Ereignisse wie Langdruck. Der Block kann über den INIT-Eingang konfiguriert werden und liefert über die Ausgänge Statusinformationen und Bestätigungen zurück.

Technische Besonderheiten

- Unterstützt multiple Eingänge (I1 bis I8)
- Verarbeitet verschiedene Ereignistypen über die InputEvent-Variable
- Kann zeitbasierte Ereignisse (Langdruck) und zählbasierte Ereignisse (Mehrfachklicks) verarbeiten
- Initialisierung mit spezifischen Parametern über PARAMS-Eingang
- Liefert detaillierte Statusinformationen über STATUS-Ausgang

Zustandsübersicht

Der Funktionsblock verfügt über einen Initialisierungszustand (INIT/INITO) und operative Zustände für Service-Anfragen (REQ/CNF) sowie asynchrone Ereignismeldungen (IND). Die genaue Zustandsmaschine ist implementierungsabhängig.

Anwendungsszenarien

- Bedienpanel-Steuerungen mit Tasteneingaben
- Maschinensteuerung mit Ereignis-basierten Eingaben
- Systeme mit komplexen Tasteneingabe-Sequenzen
- Anwendungen mit Langdruck- und Mehrfachklick-Erkennung

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Eingabeblöcken bietet logiBUS_IE2 erweiterte Funktionalität für komplexe Ereigniserkennung und unterstützt verschiedene Eingabeereignistypen mit konfigurierbaren Parametern.

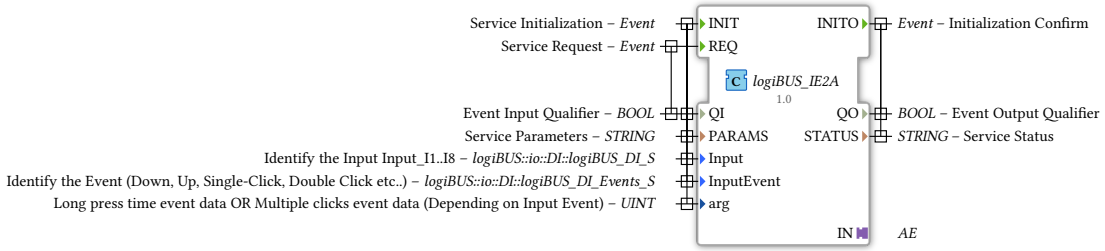
🛠️ Zugehörige Übungen

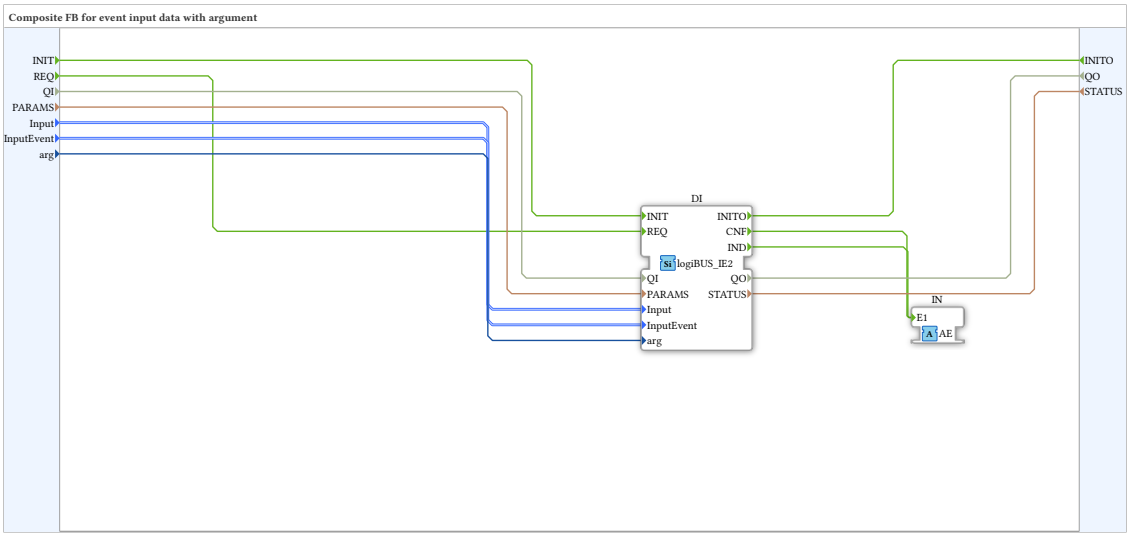
- Uebung_004c6
- Uebung_004c6_AX
- Uebung_004c7
- Uebung_004c7_AX

Fazit

Der logiBUS_IE2 ist ein leistungsstarker Funktionsblock für die Verarbeitung von Ereigniseingabedaten in logiBUS-Systemen, der durch seine Flexibilität bei der Ereigniserkennung und umfangreiche Konfigurationsmöglichkeiten für anspruchsvolle Steuerungsanwendungen geeignet ist.

logiBUS_IE2A





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `logiBUS_IE2A` ist ein Composite-Baustein (Composite FB) zur Verarbeitung von Ereignissen an Digitaleingängen des logiBUS-Systems. Er erweitert die Funktionalität des zugrundeliegenden Bausteins `logiBUS_IE2` um einen dedizierten Adapter-Ausgang für Ereignismeldungen und ermöglicht die Übergabe von Parametern wie Langdruckzeiten oder Mehrfachklick-Zählern. Der Baustein dient als standardisierte Schnittstelle zwischen der logiBUS-Hardware und einer übergeordneten Steuerungslogik.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Service-Initialisierung
REQ	Event	Service-Anforderung (Service Request)

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Initialisierungsbestätigung

Daten-Eingänge

Name	Typ	Kommentar	Initialwert
QI	BOOL	Ereignis-Eingangs-Qualifier	–
PARAMS	STRING	Service-Parameter	–
Input	`logiBUS::io::DI::logiBUS_DI_S`	Identifikation des Digitaleingangs (z. B. Input_I1..I8)	`logiBUS_DI::Invalid`
InputEvent	`logiBUS::io::DI::logiBUS_DI_Events`	Identifikation des Ereignisses (Down, Up, Single-Click, Double-Click, etc.)	`logiBUS_DI_Events::Invalid`

arg	UINT	Ereignisdaten: Langdruckzeit oder Anzahl Mehrfachklicks (abhängig vom Ereignis)	65535

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Ereignis-Ausgangs-Qualifier
STATUS	STRING	Servicestatus

Adapter

Name	Typ	Beschreibung
IN	`adapter::types::unidirectional::AE`	Unidirektionaler Adapter-Ausgang für Ereignisse (E1)

Funktionsweise

Der `logiBUS_IE2A` ist ein reiner Strukturbaustein, der die Logik des internen Bausteins `logiBUS_IE2` kapselt.

- Beim Eintreffen eines **INIT**-Ereignisses werden die Daten-Eingänge `QI`, `PARAMS`, `Input`, `InputEvent` und `arg` an den internen FB weitergeleitet und dort die Initialisierung durchgeführt. Nach erfolgreicher Verarbeitung wird über den Ausgang **INITO** eine Bestätigung mit den Ausgangsdaten `QO` und `STATUS` zurückgegeben.
- Ein **REQ**-Ereignis löst eine erneute Anforderung (z. B. erneute Messung) aus, ohne die Parameter erneut zu setzen.
- Die eigentlichen Ereignismeldungen des Digitaleingangs werden vom internen FB über dessen Ausgänge `IND` und `CNF` auf den Adapter-Ausgang **IN.E1** gelegt. Dadurch kann die übergeordnete Applikation auf eingehende Ereignisse reagieren, ohne die internen Details des Bausteins kennen zu müssen.

Die Datenflüsse sind im internen Netzwerk fest verdrahtet und entsprechen dem Protokoll des `logiBUS_IE2`.

Technische Besonderheiten

- **Composite-Struktur**: Der Baustein enthält keine eigene Zustandsmaschine, sondern delegiert die gesamte Logik an den internen FB `logiBUS_IE2`.
- **Vordefinierte Initialwerte**: Die Eingänge `Input` und `InputEvent` sind auf `Invalid` vorbesetzt, `arg` auf `65535`. Dadurch werden Fehlkonfigurationen vermieden, solange die Parameter nicht explizit gesetzt werden.
- **Adapter-Ausgang**: Die Ereignisausgabe erfolgt über einen unidirektionalen Adapter (`AE`), der eine lose Kopplung an die aufrufende Applikation ermöglicht.
- **Typ-Importe**: Der Baustein nutzt spezifische Typdefinitionen aus den Paketen `logiBUS::io::DI` und `logiBUS::io::DI::logiBUS_DI_Events`, was eine konsistente Nutzung im logiBUS-Ökosystem sicherstellt.

Zustandsübersicht

Der `logiBUS_IE2A` besitzt keine eigene Zustandsmaschine (kein ECC). Der interne FB `logiBUS_IE2` implementiert die notwendigen Zustände zur Ereigniserkennung und -ausgabe. Aus Sicht des Composite-Bausteins sind nur die zyklischen Abläufe INIT → INITO sowie REQ → (Ereignisausgabe über Adapter) relevant.

Anwendungsszenarien

- **Taster- und Schalterauswertung**: Erkennung von einfachen und mehrfachen Klicks sowie langen Druckvorgängen an digitalen Eingängen eines logiBUS-Moduls.
- **Parametrierbare Ereigniserkennung**: Durch den `arg`-Eingang können Schwellwerte für Langdruckzeiten oder die Anzahl von Mehrfachklicks dynamisch vorgegeben werden.

- **Integration in übergeordnete Steuerungen**: Der Adapter-Ausgang erlaubt eine saubere Anbindung an Automatisierungssysteme, die über Adapter-Schnittstellen kommunizieren.

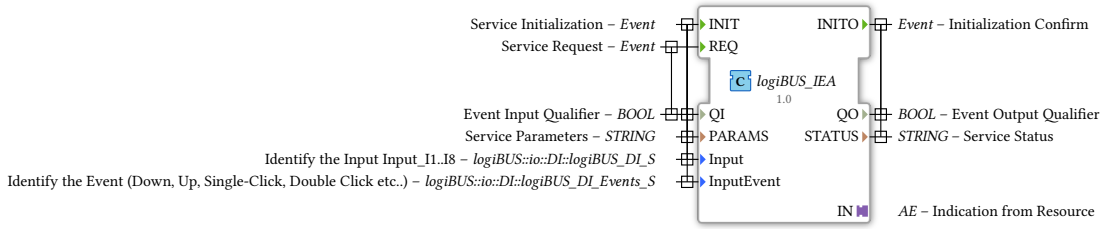
Vergleich mit ähnlichen Bausteinen

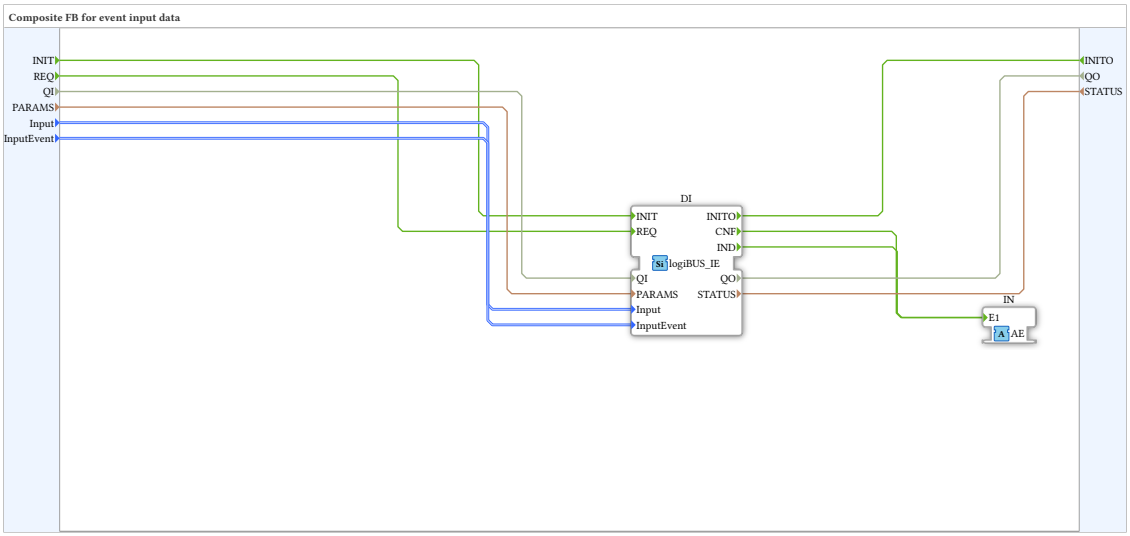
- **`logiBUS_IE2`**: Dieser Baustein wird im Inneren des `logiBUS_IE2A` verwendet. Er bietet dieselbe Funktionalität, jedoch ohne den dedizierten Adapter-Ausgang und ohne die Zusammenfassung der Parameter in einer Composite-Struktur.
- **`logiBUS_IE2S`** (falls vorhanden): Ein weiterer Baustein derselben Familie, der sich möglicherweise durch zusätzliche Statusausgaben oder andere Ereignisauswahl unterscheidet.
- Der **`logiBUS_IE2A`** zeichnet sich durch seine einfache Konfiguration und die standardisierte Adapter-Schnittstelle aus, die eine Wiederverwendung in unterschiedlichen Projekten erleichtert.

Fazit

Der `logiBUS_IE2A` ist ein fertig konfektionierter Composite-Baustein für die ereignisbasierte Verarbeitung von logiBUS-Digitaleingängen. Er kombiniert die bewährte Logik des `logiBUS_IE2` mit einem Adapter-Ausgang und parametrierbaren Ereignisdaten. Durch die klare Schnittstelle und die vorgegebenen Initialwerte eignet er sich besonders für modulare und standardisierte Automatisierungslösungen, bei denen eine schnelle Integration und einfache Handhabung im Vordergrund stehen.

logiBUS_IEA





• – ○ • – ○ • – ○ •

Einleitung

Der Composite-Funktionsblock `logiBUS_IEA` dient der ereignisgesteuerten Verarbeitung von digitalen Eingangsdaten in einem logiBUS-Feldbus-System. Er kapselt einen internen `logiBUS_IE`-Baustein und stellt dessen Funktionalität über eine einheitliche Schnittstelle zur Verfügung. Der FB ermöglicht die Initialisierung sowie die Abfrage von Ereignissen an einem digitalen Eingang und gibt die resultierenden Indikationen über einen Adapter an die Ressource weiter.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar	Mitgeführte Daten
INIT	EInit	Service Initialization	QI, PARAMS, Input, InputEvent
REQ	Event	Service Request	QI

Ereignis-Ausgänge

Name	Typ	Kommentar	Mitgeführte Daten
INITO	EInit	Initialization Confirm	QO, STATUS

Daten-Eingänge

Name	Typ	Kommentar	Initialwert
QI	BOOL	Event Input Qualifier	–
PARAMS	STRING	Service Parameters	–
Input	logiBUS::io::DI::logiBUS_DI_S	Identifiziert den Eingang (z. B. Input_I1..I8)	logiBUS_DI::Invalid
InputEvent	logiBUS::io::DI::logiBUS_DI_Events	Identifiziert das Ereignis (z. B. Down, Up, Single-Click, Double-Click)	logiBUS_DI_Events::Invalid

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Event Output Qualifier
STATUS	STRING	Service Status

Adapter

Name	Typ	Kommentar
IN	adapter::types::unidirectional::AE	Indication from Resource

Der Adapter IN nimmt die vom internen Baustein erzeugten Ereignisse (DI.IND und DI.CNF) entgegen und leitet sie als „Indication“ an die übergeordnete Ressource weiter.

Funktionsweise

Der FB logiBUS_IEA arbeitet als reiner Composite-Bautein, d. h. er besitzt keine eigene Ablaufsteuerung, sondern leitet alle Ereignisse und Daten an den eingebetteten Baustein logiBUS::io::DI::logiBUS_IE weiter.

- **INIT-Ereignis**:
  Über den Ereigniseingang INIT wird der interne Baustein initialisiert. Dabei werden die Parameter QI, PARAMS, Input und InputEvent übergeben. Nach erfolgreicher Initialisierung quittiert der interne Baustein mit dem Ereignis INITO, das an den übergeordneten FB zurückgegeben wird. Die Ausgangsdaten QO und STATUS werden dabei aktualisiert.
- **REQ-Ereignis**:
  Mit REQ (bei gesetztem QI) wird eine Anforderung an den internen Baustein gesendet. Dieser verarbeitet die Konfiguration und löst bei erkannten Ereignissen die entsprechenden Indikationen aus.
- **Indikation über Adapter**:
  Die vom internen Baustein erzeugten Ereignisse IND (Indication) und CNF (Confirm) werden auf den

Adapterausgang `IN.E1` gelegt. Auf diese Weise kann die umgebende Ressource auf Ereignisse wie Tasterdruck oder Doppelklick reagieren.

Technische Besonderheiten

- **Composite-Baustein**: Der FB enthält keinen eigenen ECC (Execution Control Chart), sondern delegiert die gesamte Logik an den internen `logiBUS_IE`. Dadurch wird die Wiederverwendung und Testbarkeit erhöht.
- **Generische Parametrierung**: Über den `PARAMS`-Eingang können systemspezifische Parameter als String übergeben werden.
- **Adapterbasierte Ereignisausgabe**: Die Nutzung eines unidirektionalen Adapters (Typ `AE`) ermöglicht eine flexible Kopplung an die Ressource ohne feste Verdrahtung auf Ereignisausgänge.
- **Typisierte Eingabeselektion**: Die Eingänge `Input` und `InputEvent` nutzen spezifische Aufzählungstypen (`logiBUS_DI_S` und `logiBUS_DI_Events_S`), die eine sichere und eindeutige Konfiguration erlauben.

Zustandsübersicht

Da `logiBUS_IEA` ein reiner Composite-Baustein ist, besitzt er keinen eigenen Zustandsautomaten. Der Zustand wird durch den internen Baustein bestimmt. Typische Zustände des inneren `logiBUS_IE` umfassen:

- **IDLE**: Warten auf INIT oder Konfiguration
- **WAIT**: Warten auf ein erkanntes Ereignis am Eingang
- **ACTIVE**: Ereignis erkannt, Indikation wird gesendet
- **ERROR**: Fehler bei Initialisierung oder Kommunikation

Die Zustandsübergänge werden durch die Ereignisse `IND` und `CNF` sichtbar und über den Adapter nach außen kommuniziert.

Anwendungsszenarien

- **Ereigniserkennung an digitalen Eingängen**: Taster, Schalter oder Sensoren mit Mehrfachklick-Funktionalität (Single/Double-Click) können über `InputEvent` konfiguriert werden.
- **Steuerung von Maschinenmodi**: Ein Digitaleingang löst bei Betätigung (z. B. Down-Ereignis) einen Moduswechsel aus.
- **Parametrierbare Initialisierung**: Über `PARAMS` können Bus-Parameter oder Filterzeiten an den logiBUS-Controller übergeben werden.
- **Integration in größere Composite-Blöcke**: Der Adapter `IN` erlaubt die Anbindung an einen übergeordneten Steuerungsbaustein (z. B. einen Ressourcen-Manager).

Vergleich mit ähnlichen Bausteinen

FB-Typ	Eigenschaft
`logiBUS_DI`	Reiner Digitaleingang ohne Ereignisdetektion; einfacher Pegelabruf
`logiBUS_IEA`	**Dieser FB:** Ereignisbasierte Eingabe mit Initialisierung und Adapter für Indikation
`logiBUS_DI_Events`	Ereignisdetektion auf einem einzelnen Eingang, aber ohne Initialisierungslogik

Gegenüber dem einfachen Digitaleingang bietet `logiBUS_IEA` eine vollständige Initialisierungsschnittstelle und eine standardisierte Ereignissignalisierung über einen Adapter. Im Vergleich zu einem reinen Ereignisbaustein enthält er zusätzlich die INIT-Funktionalität, die für die Inbetriebnahme eines logiBUS-Teilnehmers erforderlich ist.

Fazit

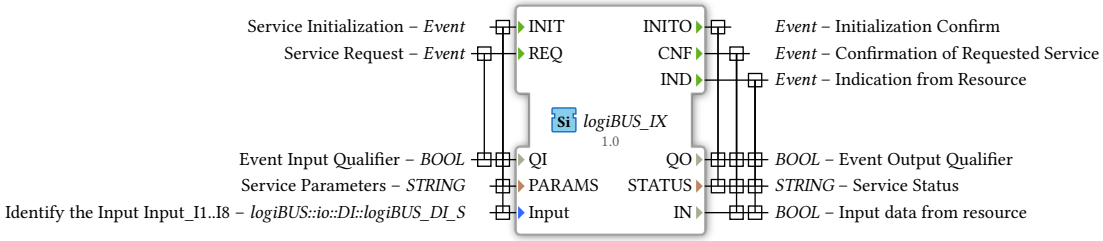
Der Composite-Baustein `logiBUS_IEA` kapselt die komplexe Initialisierung und Ereignisverarbeitung eines logiBUS-Eingangsmoduls in einer kompakten, wiederverwendbaren Einheit. Durch die klare Trennung von Initialisierung und Anforderung, die Unterstützung typisierter Eingänge sowie die adapterbasierte Ereignisausgabe eignet er sich besonders für den Einsatz in modularen Automatisierungslösungen, bei denen zuverlässige ereignisgesteuerte Eingabeerfassung gefordert ist.

logiBUS_IX

• – ◦ • – ◦ • – ◦ •

Einleitung

Der logiBUS_IX Funktionsblock ist ein Eingabeservice-Interface für boolesche Eingangsdaten, der speziell für die Kommunikation mit logiBUS-Eingabemodulen entwickelt wurde. Er dient als Schnittstelle zwischen der Steuerungslogik und physischen Eingabesignalen und ermöglicht die Abfrage von digitalen Eingangswerten.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierungsereignis
- **REQ**: Service-Anfrageereignis

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
- **CNF**: Bestätigung der angeforderten Service-Anfrage
- **IND**: Anzeige vom Ressourcen-Interface

Daten-Eingänge

- **QI**: Ereignis-Eingangsqualifizierer (BOOL)
- **PARAMS**: Service-Parameter (STRING)
- **Input**: Identifiziert den Eingang I1..I8 (logiBUS_DI_S) - Initialwert: Invalid

Daten-Ausgänge

- **QO**: Ereignis-Ausgangsqualifizierer (BOOL)
- **STATUS**: Service-Status (STRING)
- **IN**: Eingangsdaten von der Ressource (BOOL)

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Funktionsblock initialisiert sich über das INIT-Ereignis und kann anschließend über das REQ-Ereignis Eingangsdaten anfordern. Bei erfolgreicher Initialisierung gibt er das INITO-Ereignis zurück. Die tatsächlichen Eingangswerte werden über den IN-Ausgang bereitgestellt, während der STATUS-Ausgang Informationen über den aktuellen Betriebszustand liefert.

Technische Besonderheiten

- Unterstützt bis zu 8 digitale Eingänge (I1..I8)
- Verwendet spezielle logiBUS-Datentypen für die Eingangsidentifikation
- Bietet umfangreiche Statusrückmeldungen über den STATUS-Ausgang
- Initialisiert mit einem ungültigen Eingangswert (Invalid)

Zustandsübersicht

Der Funktionsblock durchläuft typischerweise folgende Zustände:

1. **Nicht initialisiert**: Vor der INIT-Anforderung
2. **Initialisiert**: Nach erfolgreicher INIT-Verarbeitung
3. **Bereit**: Kann REQ-Anfragen verarbeiten
4. **Aktiv**: Verarbeitet gerade eine Service-Anfrage

Anwendungsszenarien

- Abfrage von digitalen Eingangssignalen in Automatisierungssystemen
- Integration von logiBUS-Eingabemodulen in 4diac-basierte Steuerungen
- Überwachung von Schalterzuständen und Sensorsignalen
- Industrielle E/A-Steuerung mit Statusüberwachung

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Eingangsblöcken bietet logiBUS_IX:

- Erweiterte Statusinformationen
- Parametrierbare Service-Parameter
- Spezifische logiBUS-Hardware-Integration
- Umfangreichere Initialisierungs- und Bestätigungsmechanismen

🛠️ Zugehörige Übungen

- Uebung_001
- Uebung_001c
- Uebung_002
- Uebung_002a
- Uebung_002a2
- Uebung_002a3
- Uebung_002a4
- Uebung_002a5b
- Uebung_002b2
- Uebung_002b3
- Uebung_003
- Uebung_003a0
- Uebung_003a_sub
- IX_TO_Aux_QX
- Uebung_003d
- Uebung_005
- Uebung_006e1
- Uebung_006e2
- Uebung_019c
- Uebung_020a
- Uebung_020b
- Uebung_020c
- Uebung_020c2
- Uebung_020c3
- Uebung_020d
- Uebung_020e
- Uebung_020e2
- Uebung_020f
- Uebung_020f2
- Uebung_020g
- Uebung_020i
- Uebung_028
- Uebung_029
- Uebung_030
- Uebung_032
- IX_TO_logiBUS_LED_strip_QX
- Uebung_049
- Uebung_051
- Uebung_052
- Uebung_053
- Uebung_054
- Uebung_055
- Uebung_056
- Uebung_085

- Uebung_086
- Uebung_087
- Uebung_087a1
- Uebung_087a2
- Uebung_088
- Uebung_089
- Uebung_090a1
- Uebung_090a1_AX
- Uebung_090a2
- Uebung_090a2_AX
- Uebung_094
- Uebung_094a
- Uebung_095
- Uebung_160
- Uebung_160b2
- Uebung_177
- Uebung_178

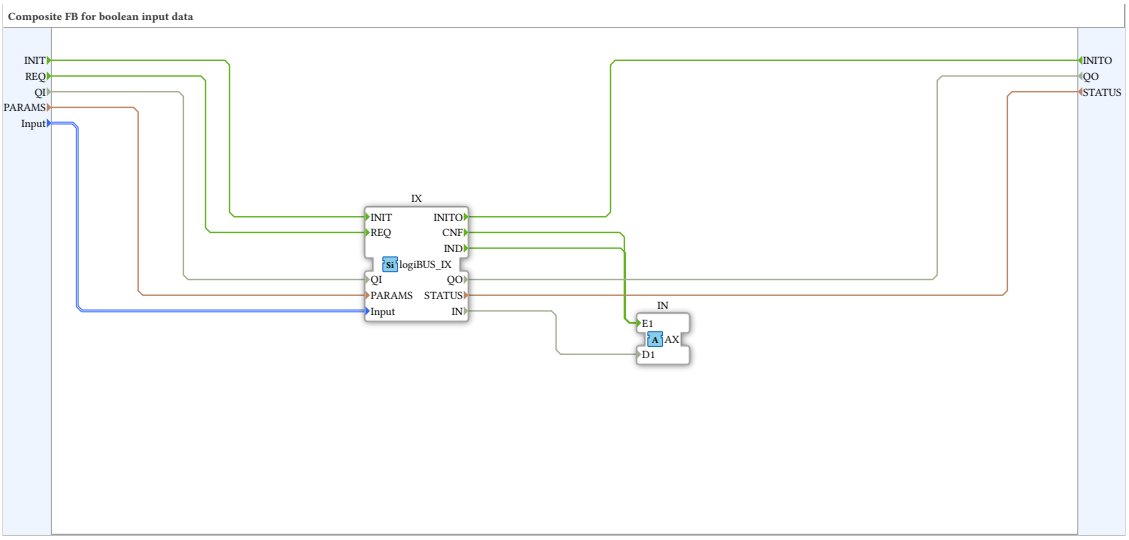
Fazit

Der logiBUS_IX Funktionsblock stellt eine robuste und flexible Lösung für die Integration von logiBUS-Eingabemodulen in 4diac-basierte Steuerungssysteme dar. Seine umfangreiche Statusrückmeldung und parametrierbare Schnittstelle machen ihn besonders geeignet für industrielle Anwendungen, bei denen zuverlässige E/A-Kommunikation erforderlich ist.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

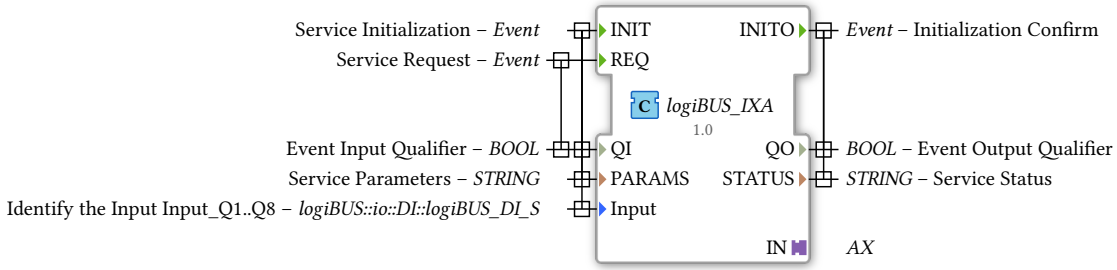
logiBUS_IXA



• – ○ • – ○ • – ○ •

Einleitung

Der logiBUS_IXA ist ein Composite-Funktionsblock zur Verarbeitung von booleschen Eingangsdaten. Er dient als Schnittstelle für digitale Eingänge und ermöglicht die Initialisierung und Abfrage von Eingangssignalen über standardisierte Service-Schnittstellen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierungsereignis mit den zugehörigen Daten QI, PARAMS und Input
- **REQ**: Service-Anfrageereignis mit dem zugehörigen Datenwert QI

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung mit den zugehörigen Datenwerten QO und STATUS

Daten-Eingänge

- **QI**: Boolescher Ereignis-Eingangsqualifizierer
- **PARAMS**: Service-Parameter als Zeichenkette
- **Input**: Identifiziert die Eingänge Q1 bis Q8 mit dem Typ logiBUS_DI_S und Initialwert „Invalid“

Daten-Ausgänge

- **QO**: Boolescher Ereignis-Ausgangsqualifizierer
- **STATUS**: Service-Status als Zeichenkette

Adapter

- **IN**: Unidirektionaler Adapter vom Typ AX für die Eingangsdatenverarbeitung

Funktionsweise

Der Composite-Funktionsblock logiBUS_IXA kapselt den Basisfunktionsblock logiBUS_IX und erweitert dessen Funktionalität durch zusätzliche Adapter-Schnittstellen. Bei INIT-Ereignissen werden die Parameter an den internen IX-Block weitergeleitet, der die Initialisierung durchführt. Die IND-Ereignisse des IX-Blocks werden an den Eingangsadapter IN weitergegeben, während die Datenverbindungen die entsprechende Signalverarbeitung sicherstellen.

Technische Besonderheiten

- Verwendet den spezifischen Datentyp logiBUS_DI_S für die Eingangsidentifikation
- Implementiert standardisierte Service-Schnittstellen gemäß 61499-2
- Unterstützt Parameterübergabe via STRING-Datentyp
- Bietet Statusrückmeldungen über STATUS-Ausgang

Zustandsübersicht

Der Funktionsblock verfügt über zwei Hauptzustände:

1. **Nicht initialisiert**: Vor dem INIT-Ereignis
2. **Initialisiert und betriebsbereit**: Nach erfolgreicher INIT-Bestätigung

Anwendungsszenarien

- Anbindung digitaler Eingänge in Automatisierungssystemen
- Integration in logiBUS-basierte Steuerungsarchitekturen
- Verwendung in SPS-Systemen mit booleschen Signalverarbeitungsanforderungen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Eingangsblöcken bietet logiBUS_IXA erweiterte Service-Funktionalitäten mit Parametrierungsmöglichkeiten und Statusrückmeldungen. Die Composite-Struktur ermöglicht eine bessere Wiederverwendbarkeit und erweiterte Diagnosefähigkeiten.

🛠️ Zugehörige Übungen

- Uebung_001_AX
- Uebung_001_AX_b
- Uebung_001c_AX
- Uebung_002_AX
- Uebung_002a2_AX
- Uebung_002a3_AX
- Uebung_002a5_AX
- Uebung_002a5b_AX
- Uebung_002a6_AX
- Uebung_002a7_AX
- Uebung_002a_AX
- Uebung_002b3_AX
- Uebung_003_AX
- Uebung_003a0_AX
- IXA_TO_Aux_QXA
- Uebung_003d_AX
- Uebung_005_AX
- Uebung_006e1_AX
- Uebung_006e2_AX
- Uebung_020a_AX
- Uebung_020b_AX
- Uebung_020c3_AX
- Uebung_020c_AX
- Uebung_020d_AX
- Uebung_020e2_AX
- Uebung_020e_AX
- Uebung_020f2_AX
- Uebung_020f_AX
- Uebung_020g_AX
- Uebung_020i_AX
- Uebung_020j2_AX
- Uebung_020j_AX
- Uebung_090a1_AX
- Uebung_090a2_AX
- Uebung_094a_AX
- Uebung_095_AX
- Uebung_103
- Uebung_103c
- Uebung_103c2
- Uebung_160_AX
- Uebung_160b2_AX
- Uebung_177_AX
- Uebung_178_AX

Fazit

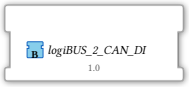
Der logiBUS_IXA Funktionsblock stellt eine robuste und flexible Lösung für die Verarbeitung digitaler Eingangssignale in 4diac-basierten Automatisierungssystemen dar. Durch seine standardisierten Schnittstellen und erweiterten Service-Funktionen eignet er sich besonders für komplexe Anwendungen mit hohen Anforderungen an Diagnose und Parametrierbarkeit.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

DI_CAN

logiBUS_2_CAN_DI



• – ◦ • – ◦ • – ◦ •

Einleitung

Die logiBUS_2_CAN_DI GlobalConstants definieren Konstanten für die Konfiguration von digitalen Eingängen (DI) im logiBUS-System über CAN-Bus. Diese Konstanten werden zur Pin-Zuweisung für digitale Eingangskanäle verwendet und stellen eine standardisierte Schnittstelle für die Hardware-Konfiguration bereit.

Schnittstellenstruktur

Konstantendefinitionen

Die GlobalConstants definieren drei Hauptkonstanten für die digitale Eingangskonfiguration:

- **Input_CI1**: Konfiguration für digitalen Eingangskanal 1
- **Input_CI2**: Konfiguration für digitalen Eingangskanal 2
- **Invalid**: Kennzeichnet einen ungültigen oder nicht konfigurierten Eingang

Datentypen

Alle Konstanten verwenden den strukturierten Datentyp `logiBUS_2_CAN_DI_S`, der folgende Komponente enthält:

- **Pin**: Ganzzahlwert zur Identifikation des physikalischen Eingangspins

Funktionsweise

Die Konstanten werden zur Laufzeit nicht verändert und dienen ausschließlich der Konfiguration der digitalen Eingänge. Sie definieren die Hardware-Zuordnung zwischen logischen Kanälen und physikalischen Pins im CAN-basierten logiBUS-System.

Technische Besonderheiten

- **Pin-Zuordnung**: Kanal 1 ist Pin 1, Kanal 2 ist Pin 2 zugeordnet
- **Ungültige Kennung**: Der Wert 255 kennzeichnet einen ungültigen oder nicht verfügbaren Eingang
- **CAN-Integration**: Speziell für CAN-basierte logiBUS-Systeme optimiert
- **Typensicherheit**: Verwendung strukturierter Datentypen für konsistente Konfiguration

Anwendungsszenarien

- Konfiguration von digitalen Eingängen in logiBUS-Steuerungen
- Hardware-Abstraktion in CAN-basierten Automatisierungssystemen
- Pin-Zuordnung in verteilten E/A-Systemen
- Fehlerbehandlung bei nicht konfigurierten Eingängen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen numerischen Konstanten bieten diese strukturierten Konstanten:

- Höhere Typsicherheit durch strukturierte Datentypen
- Bessere Wartbarkeit durch zentrale Konfiguration
- Erweiterbarkeit für zusätzliche Parameter
- Kompatibilität mit logiBUS-Systemarchitektur

Fazit

Die logiBUS_2_CAN_DI GlobalConstants bieten eine robuste und typsichere Lösung für die Konfiguration digitaler Eingänge in CAN-basierten logiBUS-Systemen. Durch die strukturierte Herangehensweise ermöglichen sie eine klare Hardware-Zuordnung und erleichtern die Wartung und Erweiterung von Automatisierungslösungen.

logiBUS_2_CAN_DI_S

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `logiBUS_2_CAN_DI_S` ist ein strukturierter Datentyp (Struct), der zur Konfiguration eines digitalen Eingangs (DI) für ein logiBUS-CAN-System dient. Er definiert die grundlegenden Parameter, die benötigt werden, um einen physischen Eingangspin auf einem logiBUS-CAN-Modul zu adressieren und zu nutzen. Dieser Datentyp dient typischerweise als Eingabe für übergeordnete Funktionsbausteine, die die eigentliche Kommunikation über das CAN-Protokoll abwickeln.

Schnittstellenstruktur

Dies ist ein strukturierter Datentyp (STRUCT) und besitzt daher keine Ereignis- oder Adapterschnittstellen. Seine Struktur besteht aus einem einzelnen Datenelement.

Daten-Eingänge

Da es sich um einen Datentyp handelt, werden diese Elemente innerhalb eines übergeordneten Funktionsbausteins als Eingangsdaten verwendet.

Name	Datentyp	Initialwert	Kommentar
`Pin`	`USINT` (Unsigned Short Integer)	255	Definiert die Nummer des physischen Eingangspins am logiBUS-CAN-Modul.

Daten-Ausgänge

(Nicht zutreffend für einen strukturierten Datentyp)

Ereignis-Eingänge

(Nicht zutreffend für einen strukturierten Datentyp)

Ereignis-Ausgänge

(Nicht zutreffend für einen strukturierten Datentyp)

Adapter

(Nicht zutreffend für einen strukturierten Datentyp)

Funktionsweise

Der Datentyp `logiBUS_2_CAN_DI_S` fungiert als Konfigurationscontainer. Der enthaltene Wert `Pin` spezifiziert, welcher konkrete digitale Eingang eines logiBUS-CAN-I/O-Moduls angesprochen werden soll. Der Initialwert `255` stellt einen typischen ungültigen oder nicht konfigurierten Zustand dar, der eine explizite Zuweisung einer gültigen Pinnummer durch den Anwender erfordert. Ein übergeordneter Funktionsbaustein (z.B. ein Treiber-FB) liest diese Struktur aus und verwendet die `Pin`-Information, um die korrekten CAN-Telegramme zu generieren oder zu interpretieren, die den Zustand (Ein/Aus) des jeweiligen Eingangs abfragen.

Technische Besonderheiten

- **Datentyp `USINT`:** Die Verwendung eines `USINT` (0 bis 255) für den Pin legt nahe, dass die logiBUS-Hardware theoretisch bis zu 256 digitale Eingänge pro Konfigurationsstruktur adressieren kann. Der Wert 255 ist als Standard-Initialwert reserviert.
- **Initialwert 255:** Dieser Wert dient als Kennzeichnung für eine nicht aktive oder nicht definierte Konfiguration. Er sollte in der Anwendung durch eine gültige Pinnummer ersetzt werden.
- **Packagename:** Der Datentyp ist im Package `logiBUS::io::DI_CAN` organisiert, was seine Zugehörigkeit zur logiBUS-CAN-Eingangskommunikation klar kennzeichnet.

Zustandsübersicht

(Nicht zutreffend für einen strukturierten Datentyp. Der „Zustand“ ist lediglich der gespeicherte Wert der `Pin`-Variable.)

Anwendungsszenarien

Dieser Strukturtyp wird in Szenarien eingesetzt, bei denen digitale Signale (z.B. Schalter, Endschalter, Sensorkontakte) über ein CAN-basiertes logiBUS-Netzwerk in eine Steuerung (z.B. eine IEC 61499-basierte Applikation in 4diac) eingelesen werden sollen. Typische Anwendungen sind:

- Maschinensteuerungen mit dezentraler, CAN-vernetzter Sensorik.
- Mobile Anwendungen (Fahrzeuge, Roboter) mit robustem Bussystem.
- Anlagen, bei denen digitale Eingangssignale über größere Entfernungen verteilt sind.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu direkten I/O-Funktionsbausteinen (wie z.B. `E_SR` oder `E_D_FF`), die logische Zustände verarbeiten, ist `logiBUS_2_CAN_DI_S` ein reiner **Konfigurationsdatentyp**. Er ist vergleichbar mit anderen strukturierten Hardware-Konfigurationstypen in 4diac (z.B. für serielle Schnittstellen oder andere Feldbusse), die Parameter für die Hardware-Anbindung kapseln. Sein Gegenstück für digitale Ausgänge wäre ein entsprechender Struct-Typ (z.B. `logiBUS_2_CAN_DO_S`).

Fazit

Der `logiBUS_2_CAN_DI_S` Datentyp ist ein essentielles und schlankes Konfigurationselement für die Anbindung digitaler Eingänge an ein CAN-basiertes logiBUS-System innerhalb der 4diac IDE. Durch die Kapselung der Pinnummer in eine strukturierte Variable ermöglicht er eine klare, typgesicherte und wartbare Konfiguration der Hardware-Schnittstelle in übergeordneten Steuerungsalgorithmen. Die Verwendung eines eindeutigen Initialwerts hilft, Fehlkonfigurationen frühzeitig zu erkennen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

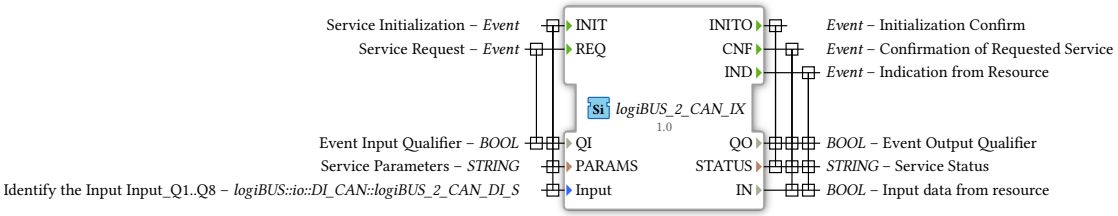
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

logiBUS_2_CAN_IX

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `logiBUS_2_CAN_IX` ist ein Service-Interface-Funktionsblock für boolesche Eingangsdaten, der speziell für die Integration von logiBUS-Systemen mit CAN-Bus-Kommunikation entwickelt wurde. Er dient als Eingabeschnittstelle für digitale Signale und ermöglicht die Kommunikation zwischen dem Steuerungssystem und angeschlossenen CAN-fähigen Geräten.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierungsereignis
  - Verknüpft mit: QI, PARAMS, Input
- **REQ**: Service-Anfrageereignis
  - Verknüpft mit: QI

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
  - Verknüpft mit: QO, STATUS
- **CNF**: Bestätigung der angeforderten Service-Anfrage
  - Verknüpft mit: QO, STATUS, IN
- **IND**: Indikation von der Ressource
  - Verknüpft mit: QO, STATUS, IN

Daten-Eingänge

- **QI**: BOOL - Ereignis-Eingangsqualifikator
- **PARAMS**: STRING - Service-Parameter
- **Input**: logiBUS_2_CAN_DI_S - Identifiziert den Eingang Input_Q1..Q8
  - Initialwert: logiBUS_2_CAN_DI::Invalid

Daten-Ausgänge

- **QO**: BOOL - Ereignis-Ausgangsqualifikator
- **STATUS**: STRING - Service-Status
- **IN**: BOOL - Eingangsdaten von der Ressource

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Funktionsblock arbeitet als Eingabeservice-Interface für digitale CAN-Signale. Bei Initialisierung (INIT-Event) werden die Service-Parameter konfiguriert und der Eingangskanal identifiziert. Über das REQ-Event können Service-Anfragen gestellt werden, während IND-Events eingehende Daten von der Ressource signalisieren. Der Block verwaltet den Status der Kommunikation und liefert die empfangenen booleschen Eingangsdaten über den IN-Ausgang.

Technische Besonderheiten

- Spezielle Datentypen für logiBUS-CAN-Integration
- Initialisierung mit Invalid-Zustand als Standard
- Separate Bestätigungs- und Indikations-Events für unterschiedliche Betriebszustände
- String-basierte Statusrückmeldung für detaillierte Fehlerdiagnose

Zustandsübersicht

Der Funktionsblock durchläuft folgende Hauptzustände:

1. **Nicht initialisiert**: Vor INIT-Event
2. **Initialisiert**: Nach erfolgreicher INITO-Bestätigung
3. **Service-Bereit**: Kann REQ- und IND-Events verarbeiten
4. **Fehlerzustand**: Bei Kommunikationsproblemen (über STATUS signalisiert)

Anwendungsszenarien

- Integration von logiBUS-I/O-Modulen in 4diac-Steuerungen
- CAN-basierte digitale Eingangssignalverarbeitung
- Industrielle Automatisierungssysteme mit verteilten I/Os
- Robotersteuerungen mit CAN-Peripherie

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu standardmäßigen digitalen Eingangsblöcken bietet `logiBUS_2_CAN_IX` spezielle CAN-Integration und erweiterte Statusrückmeldungen. Während einfache DI-Blöcke nur direkte Signalverarbeitung bieten, unterstützt dieser Block komplexere Protokollhandhabung und Fehlerbehandlung für industrielle CAN-Netzwerke.

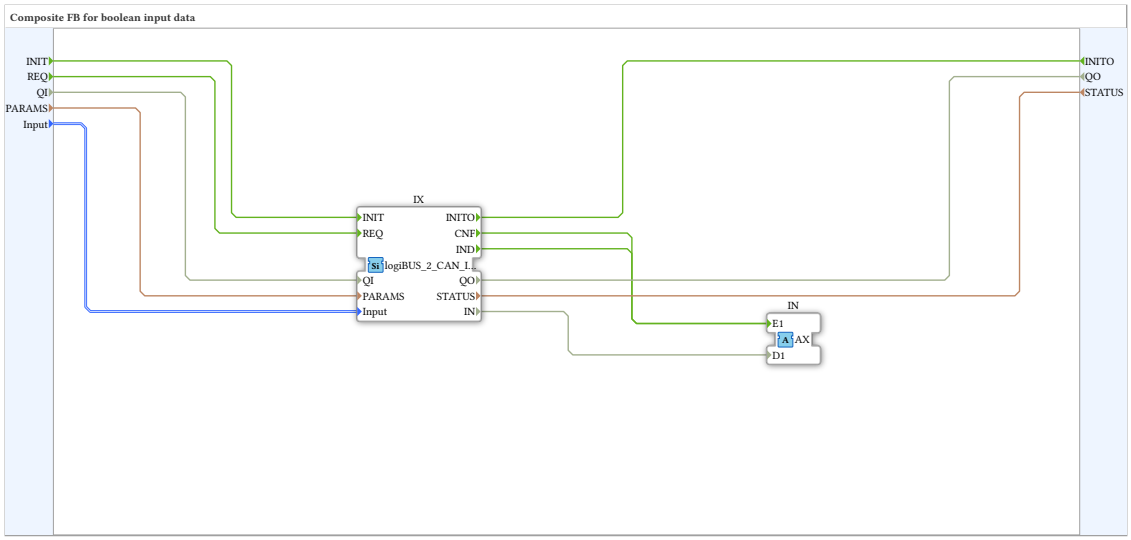
Fazit

Der `logiBUS_2_CAN_IX` Funktionsblock stellt eine robuste und zuverlässige Schnittstelle für die Integration von CAN-basierten digitalen Eingangssignalen in 4diac-Steuerungssysteme dar. Seine erweiterte Fehlerbehandlung und Statusrückmeldung machen ihn besonders geeignet für anspruchsvolle industrielle Anwendungen, wo zuverlässige Kommunikation mit verteilten I/O-Systemen erforderlich ist.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

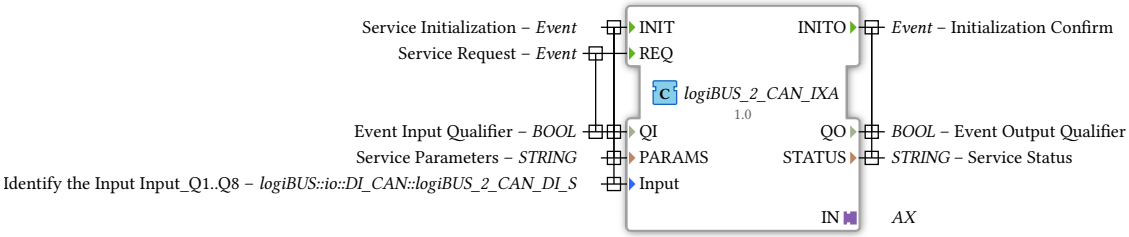
logiBUS_2_CAN_IXA



• – ○ • – ○ • – ○ •

Einleitung

Der logiBUS_2_CAN_IXA ist ein Composite-Funktionsblock zur Verarbeitung von booleschen Eingangsdaten über CAN-Bus-Schnittstellen. Der Baustein dient als Schnittstelle zwischen logiBUS-Systemen und CAN-basierten Eingabegeräten und ermöglicht die Initialisierung und Abfrage von digitalen Eingängen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung (mit QI und PARAMS)
- **REQ**: Service-Anfrage (mit QI)

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung (mit QO und STATUS)

Daten-Eingänge

- **QI**: Ereignis-Eingangsqualifizierer (BOOL)
- **PARAMS**: Service-Parameter (STRING)
- **Input**: Identifizierung des Eingangs Input_Q1..Q8 (logiBUS_2_CAN_DI_S)

Daten-Ausgänge

- **QO**: Ereignis-Ausgangsqualifizierer (BOOL)
- **STATUS**: Service-Status (STRING)

Adapter

- **IN**: Unidirektionaler Adapter vom Typ AX für die Datenkommunikation

Funktionsweise

Der Composite-Funktionsblock nutzt intern den logiBUS_2_CAN_IX-Baustein zur Verarbeitung der CAN-basierten Eingangsdaten. Bei der INIT-Initialisierung werden die Service-Parameter übergeben und

der Baustein konfiguriert. Über das REQ-Ereignis können Eingangsdaten abgefragt werden. Die Verarbeitung erfolgt über den Adapter IN, der die Kommunikation mit den CAN-Eingabegeräten ermöglicht.

Technische Besonderheiten

- Verwendet spezifische logiBUS-Datentypen für CAN-Kommunikation
- Unterstützt bis zu 8 digitale Eingänge (Input_Q1 bis Input_Q8)
- Initialwert für Input ist „Invalid“ (ungültig)
- Integriert Fehlerbehandlung über STATUS-Ausgang

Zustandsübersicht

Der Baustein durchläuft folgende Zustände:

1. **Nicht initialisiert**: Vor INIT-Aufruf
2. **Initialisierung**: Während INIT-Verarbeitung
3. **Bereit**: Nach erfolgreicher Initialisierung (INITO bestätigt)
4. **Abfrage**: Während REQ-Verarbeitung

Anwendungsszenarien

- Anbindung von CAN-basierten Sensoren und Schaltern an logiBUS-Systeme
- Industrielle Automatisierung mit digitalen Eingangssignalen
- Steuerungssysteme in der Agrartechnik
- Verteilte E/A-Systeme über CAN-Bus

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Eingangsbausteinen bietet logiBUS_2_CAN_IXA:

- CAN-Bus-Kommunikationsfähigkeit
- Erweiterte Parametrierungsmöglichkeiten
- Integrierte Statusrückmeldungen
- Spezifische logiBUS-Integration

Fazit

Der logiBUS_2_CAN_IXA ist ein spezialisierter Composite-Funktionsblock für die zuverlässige Integration von CAN-basierten digitalen Eingängen in logiBUS-Systeme. Durch seine strukturierte Initialisierung und Abfragemechanismen eignet er sich besonders für industrielle Anwendungen, die robuste und konfigurierbare E/A-Schnittstellen erfordern.

DO

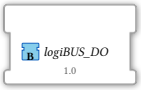
🎧 Podcast

- R_TRIG & F_TRIG: So erkennen SPS-Steuerungen Signalflanken zuverlässig – ohne Doppelbehandlung
- Adolf Webers Scharfe Analyse: Warum Technik allein nicht reicht – Lehren aus drei Industriellen Phasen
- Bulldog Legende: Wie der einfache LANZ-Traktor die Landwirtschaft revolutionierte und zum Duden-Eint
- Bulldog, Boom und Business: Heinrich Lanz – Aufstieg aus dem Handelshaus zum Landmaschinenimperium (Die unterschätzte Rolle der Familie)
- Das Kettenmonster erwacht: Lanz Bulldog Raupe – Die faszinierende Wiederbelebung des 10-Liter-Glühkopf-Arbeitstiers nach 25 Jahren Stillstand

📺 Video

- The downfall of Lanz

logiBUS_DO



• – ◦ • – ◦ • – ◦ •

Einleitung

Die logiBUS_DO GlobalConstants definieren eine Sammlung von Konstanten für die Konfiguration von digitalen Ausgängen (DO - Digital Output) im logiBUS-System. Diese Konstanten werden zur Pin-Zuweisung für digitale Ausgangskanäle verwendet und stellen eine standardisierte Schnittstelle für die Hardware-Konfiguration bereit.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

Keine Adapter vorhanden

Funktionsweise

Die logiBUS_DO GlobalConstants definieren strukturierte Konstanten vom Typ `logiBUS_DO_S`, die jeweils einen spezifischen Pin-Wert für digitale Ausgänge enthalten. Die Konstanten `Output_Q1` bis `Output_Q10` repräsentieren die verfügbaren Ausgangskanäle mit den entsprechenden Pin-Nummern 1 bis 10. Die Konstante `Invalid` mit dem Pin-Wert 255 dient als Kennzeichnung für ungültige oder nicht konfigurierte Ausgänge.

Technische Besonderheiten

- **Konstantentyp**: Alle Konstanten sind vom strukturierten Datentyp `logiBUS_DO_S`
- **Pin-Zuordnung**: Klare Zuordnung von Ausgangsnamen zu physikalischen Pin-Nummern
- **Invalid-Kennzeichnung**: Spezielle Konstante mit Pin-Wert 255 für Fehlerbehandlung
- **Package-Struktur**: Organisiert im Package `logiBUS::io::DQ`

Zustandsübersicht

Da es sich um Konstanten handelt, sind keine Zustandsübergänge vorhanden. Die Werte sind statisch und unveränderlich während der Laufzeit.

Anwendungsszenarien

- Hardware-Konfiguration von logiBUS-Systemen
- Pin-Zuweisung in Funktionsblöcken für digitale Ausgänge
- Fehlerbehandlung bei ungültigen Ausgangskonfigurationen
- Standardisierung von Ausgangsbezeichnungen in Steuerungsanwendungen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Konstantendefinitionen bietet logiBUS_DO eine spezifische Struktur für digitale Ausgänge mit klarer Pin-Zuordnung und einer definierten Invalid-Konstante für robuste Fehlerbehandlung.

Fazit

Die logiBUS_DO GlobalConstants bieten eine standardisierte und wartungsfreundliche Lösung für die Konfiguration digitaler Ausgänge in logiBUS-Systemen. Durch die strukturierte Definition der Ausgangskanäle und die bereitgestellte Invalid-Konstante wird eine zuverlässige und fehlertolerante Hardware-Konfiguration ermöglicht.

logiBUS_DO_S

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `logiBUS_DO_S` ist ein strukturierter Datentyp (Struct), der zur Konfiguration eines digitalen Ausgangs (DO) innerhalb des logiBUS-Systems dient. Er definiert die notwendigen Parameter, um einen physischen Ausgangspin eines logiBUS I/O-Moduls eindeutig zu adressieren und zu steuern. Dieser Datentyp wird typischerweise als Eingabe für Funktionsblöcke verwendet, die die eigentliche Ausgabefunktionalität bereitstellen.

Schnittstellenstruktur

Als strukturierter Datentyp besitzt `logiBUS_DO_S` keine Ereignis- oder Adapterschnittstellen im herkömmlichen Sinne eines Funktionsblocks. Stattdessen definiert er eine Datenstruktur, die als Ganzes an andere Blöcke übergeben wird.

Daten-Eingänge

Die Struktur selbst enthält folgendes Datenfeld, das bei der Instanziierung des Typs befüllt wird:

Name	Datentyp	Initialwert	Beschreibung
`Pin`	`USINT` (Unsigned Short Integer)	255	Definiert die Nummer des physischen Ausgangspins, der angesteuert werden soll. Der Wert 255 stellt einen ungültigen/undefinierten Zustand dar und dient als Standardinitialisierung.

Ereignis-Eingänge

Nicht vorhanden. Dieser Typ ist eine reine Datenstruktur.

Ereignis-Ausgänge

Nicht vorhanden. Dieser Typ ist eine reine Datenstruktur.

Daten-Ausgänge

Nicht vorhanden. Dieser Typ ist eine reine Datenstruktur.

Adapter

Nicht vorhanden. Dieser Typ ist eine reine Datenstruktur.

Funktionsweise

Der `logiBUS_DO_S`-Struct fungiert als Konfigurations-Container. Seine Hauptaufgabe ist es, die Information über die gewünschte Pin-Nummer (`Pin`) gebündelt und typisiert zu transportieren. Ein übergeordneter Funktionsblock (z.B. ein `logiBUS_DO`-Block) empfängt eine Instanz dieses Structs, liest den `Pin`-Wert aus und verwendet diese Information, um den entsprechenden physischen Ausgang des logiBUS-Hardwaremoduls zu schalten.

Der Initialwert von 255 für `Pin` signalisiert, dass der Pin noch nicht konfiguriert ist. Gültige Pin-Nummern liegen im Bereich, der vom jeweiligen logiBUS-I/O-Modul unterstützt wird (z.B. 0-7 für ein 8-Kanal-Modul).

Technische Besonderheiten

- **Typisierung:** Die Verwendung eines strukturierten Datentyps anstelle eines einfachen `USINT` erhöht die Typsicherheit und Lesbarkeit der Anwendung. Es ist klar, dass der Wert einen logiBUS-Ausgangspin repräsentiert.
- **Initialwert:** Der Initialwert 255 dient als „Safe State"-Indikator. Bevor die Struktur korrekt konfiguriert wird, zeigt dieser Wert einen inaktiven oder fehlerhaften Zustand an.
- **Paketzuordnung:** Der Struct ist Teil des Pakets `logiBUS::io::DQ`, was seine Zugehörigkeit zur digitalen Ein-/Ausgabe (Digital I/O) des logiBUS-Frameworks kennzeichnet.

Zustandsübersicht

Da es sich um einen passiven Datentyp handelt, besitzt `logiBUS_DO_S` keinen internen Zustandsautomaten. Der „Zustand" ist ausschließlich durch den gespeicherten Wert der `Pin`-Variable gegeben.

Anwendungsszenarien

1. **Konfiguration von Ausgängen:** Primär dient der Struct zur Parametrierung von Funktionsblöcken, die Lampen, Ventile, Relais oder andere binäre Aktoren über logiBUS-Module ansteuern.
2. **Strukturierte Parameterübergabe:** In komplexeren Applikationen können mehrere `logiBUS_DO_S`-Instanzen in Arrays oder weiteren Strukturen organisiert werden, um die Konfiguration aller Ausgänge einer Station zentral zu verwalten.
3. **Initialisierung:** Der Standardinitialwert ermöglicht es, Variablen dieses Typs zu deklarieren, ohne sie sofort befüllen zu müssen. Die eigentliche Konfiguration (Zuweisung einer gültigen Pin-Nummer) kann zu einem späteren Zeitpunkt im Programmablauf erfolgen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Einfacher `USINT`:** Der direkte Einsatz eines `USINT`-Werts für die Pin-Nummer ist möglich, bietet aber weniger semantische Klarheit und Typsicherheit. Der `logiBUS_DO_S` macht die Absicht des Datenelements explizit.
- **`logiBUS_DI_S`:** Dies ist das strukturelle Gegenstück für digitale Eingänge (DI). Es hat eine identische Struktur (ein `USINT`-Feld `Pin`), gehört aber konzeptionell zur Eingabeseite. Die Trennung in DO- und DI-Structs verbessert die Lesbarkeit und Wartbarkeit des Codes.

🛠️ Zugehörige Übungen

- Uebung_003a_sub
- Uebung_010b4_sub_AX
- Uebung_010b5_sub_AX
- Uebung_010c2
- Uebung_010c2_AX
- AuxIXA_TO_logiBUS_QXA_BG
- SR_TO_QX_TRIG_EO
- SoftkeyOverride_TO_QX_BG
- Softkey_T_FF_ILOCK_TO_QX_BG

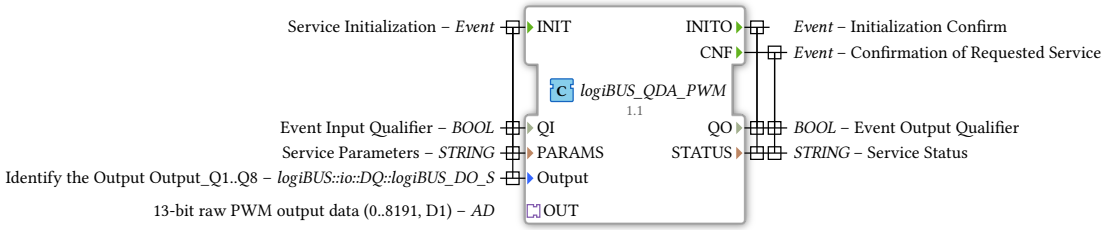
Fazit

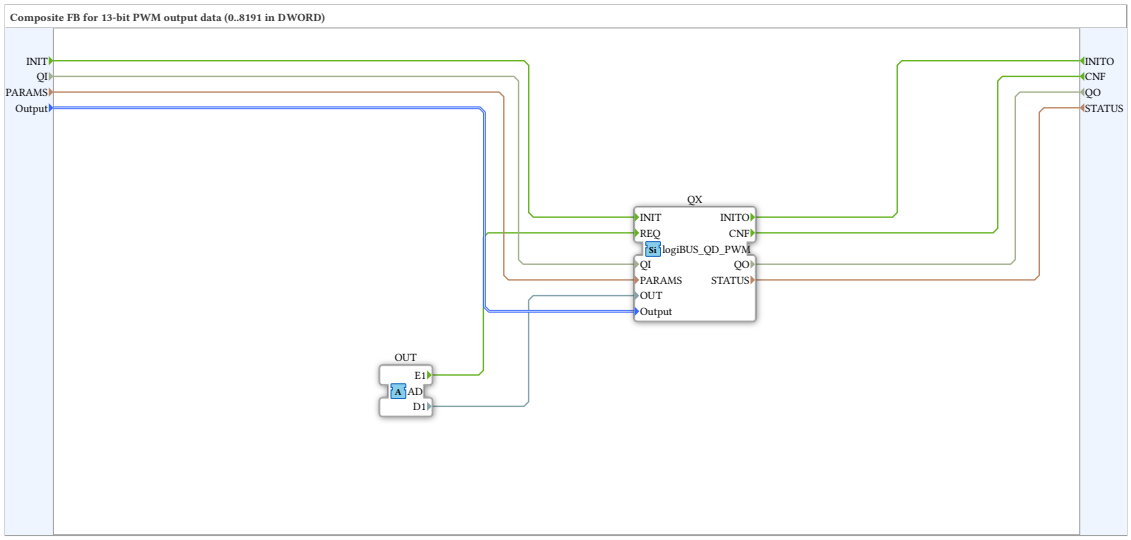
Der `logiBUS_DO_S`-Struct ist ein einfaches, aber wesentliches Hilfselement im logiBUS-Ökosystem der 4diac IDE. Er standardisiert und formalisiert die Adressierung von digitalen Ausgangspins, fördert die typsichere Programmierung und dient durch seinen definierten Initialwert als Grundlage für eine robuste Initialisierungslogik. Seine Verwendung ist empfohlen, wenn Funktionsblöcke der logiBUS-Bibliothek für digitale Ausgaben konfiguriert werden sollen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

logiBUS_QDA_PWM





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **logiBUS_QDA_PWM** ist ein Composite-Baustein zur Ansteuerung eines PWM-Ausgangs (Double Word) über ein logiBUS-System. Er kapselt die Initialisierung und die Triggerung eines internen PWM-Ausgangsbausteins und stellt eine Adapter-Schnittstelle zur Verfügung, um von außerhalb Aufträge (Ereignis und Daten) zu empfangen. Der Baustein wurde für den Einsatz in der Agrartechnik entwickelt und ist unter der EPL 2.0 lizenziert.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar
INIT	EInit	Service-Initialisierung
kein weiterer Ereignis-Eingang		*Alle Trigger erfolgen über den Adapter*

Ereignis-Ausgänge

Ereignis	Typ	Kommentar
INITO	EInit	Bestätigung der Initialisierung
CNF	Event	Bestätigung eines angeforderten Dienstes

Daten-Eingänge

Variable	Typ	Kommentar
QI	BOOL	Eingangs-Qualifier (Initialisierung freigeben)
PARAMS	STRING	Service-Parameter (z. B. Adressierung, Konfiguration)
Output	logiBUS::io::DQ::logiBUS_DO_S	Identifikation des Ausgangs (z. B. Output_Q1..Q8); Initialwert: *Invalid*

Daten-Ausgänge

Variable	Typ	Kommentar
QO	BOOL	Ausgangs-Qualifier (Initialisierungsstatus)

STATUS	STRING	Dienststatus (Fehler-/Erfolgsmeldung)

Adapter

Typ	Name	Richtung	Kommentar
adapter::types::unidirectional::AD	OUT	Socket	Empfängt Trigger-Ereignis (E1) und Ausgangsdaten (D1) von der Ressource

Funktionsweise

Der Funktionsblock ist als Composite realisiert und enthält eine interne Instanz des Bausteins `logiBUS::io::DQ::logiBUS_QD_PWM` (hier als *QX* bezeichnet). Die Logik lässt sich wie folgt beschreiben:

1. **Initialisierung:** Ein INIT-Ereignis am Eingang löst die Initialisierung des internen Bausteins aus. Dabei werden die Daten-Eingänge *QI*, *PARAMS* und *Output* an den internen Baustein weitergeleitet. Nach erfolgreicher Initialisierung wird das Ereignis *INITO* ausgegeben, zusammen mit den Ausgangsdaten *QO* und *STATUS*.
2. **Triggerung über Adapter:** Der Adapter *OUT* nimmt von außen ein Ereignis *E1* und einen Datenwert *D1* entgegen. Das Ereignis wird als *REQ* (Request) an den internen Baustein weitergegeben, die Daten *D1* als *OUT*-Wert. Der interne Baustein verarbeitet diese Anforderung und quittiert mit dem Ereignis *CNF*, das über den Ausgang *CNF* nach außen geleitet wird. Die zugehörigen Ausgangsdaten *QO* und *STATUS* werden aktualisiert.

Damit ermöglicht der Baustein eine saubere Trennung von Initialisierung und zyklischer Ausgabe: Die Konfiguration erfolgt einmalig über INIT, die eigentliche PWM-Ausgabe wird über den Adapter getriggert.

Technische Besonderheiten

- **Composite-Baustein:** Der FB kapselt die gesamte Logik eines PWM-Ausgangsbausteins und bietet eine standardisierte Adapter-Schnittstelle für den Datenaustausch mit der Ressource.
- **13-Bit-PWM-Ausgabe (`0`–`8191`):** Das Adapter-Datenelement *D1* (`DWORD`) erwartet einen 13-Bit-Rohwert (`0` = 0 % Tastgrad, `8191` = 100 % Tastgrad, $2^{13} = 8192$ Zustände).
- **Initialisierungsparameter:** Über *PARAMS* (STRING) können flexible Konfigurationsdaten übergeben werden, die für die Adressierung oder Parametrierung des logiBUS-Moduls notwendig sind.
- **Fehlerbehandlung:** Der Ausgang *STATUS* liefert eine textuelle Beschreibung des Dienstzustands (z. B. Fehlermeldungen bei ungültiger Konfiguration).

Zustandsübersicht

Da es sich um einen Composite-Baustein ohne eigenen Zustandsautomaten handelt, wird der Zustand durch den internen Baustein *logiBUS_QD_PWM* definiert. Typische Zustände sind:

- **IDLE:** Warten auf Initialisierung oder Trigger.
- **INITIALIZING:** Während der Initialisierung (INIT empfangen, INITO noch nicht gesendet).
- **OPERATIONAL:** Bereit für zyklische Trigger (über Adapter).
- **ERROR:** Fehlerzustand, erkennbar an *QO* = *FALSE* oder *STATUS* mit Fehlertext.

Der interne FB wechselt zwischen diesen Zuständen abhängig von den Ereignissen und Daten.

Anwendungsszenarien

- **Landwirtschaftliche Maschinen:** Steuerung von PWM-betriebenen Aktoren (z. B. Hydraulikventile, Motordrehzahl) über ein logiBUS-Netzwerk.
- **Automatisierungsanlagen:** Ausgabe von pulsweitenmodulierten Signalen mit 13-Bit-Auflösung (`0`–`8191`), gesteuert durch eine übergeordnete Steuerung.
- **Fernwartung und Konfiguration:** Über den Adapter können von einer übergeordneten Ressource (z. B. HMI oder SPS) neue PWM-Werte gesendet werden, ohne den Initialisierungsvorgang zu wiederholen.

Vergleich mit ähnlichen Bausteinen

Baustein	Typ	Besonderheit

logiBUS_QD_PWM	Composite/ Basic	Direkt ansteuerbar über INIT, REQ, CNF; ohne Adapter-Schnittstelle.
logiBUS_QDA_PWM	Composite	Wie logiBUS_QD_PWM, aber mit Adapter für externe Triggerung und Datenversorgung.
logiBUS_DO (einfach)	Basic	Einfacher digitaler Ausgang, keine PWM-Funktion.

Der hier beschriebene Baustein bietet eine höhere Flexibilität, da die eigentliche Ausgabe über den Adapter asynchron von der Initialisierung erfolgen kann. Nachteilig ist die Abhängigkeit von der korrekten Bereitstellung der Adapter-Signale.

Fazit

Der Funktionsblock **logiBUS_QDA_PWM** ist eine praktische Kapselung eines PWM-Ausgangs für logiBUS. Durch die Kombination von INIT-gestützter Konfiguration und adapterbasierter Triggerung eignet er sich besonders für Anwendungen, bei denen eine einmalige Parametrierung und eine anschließende zyklische oder ereignisgesteuerte Ausgabe erforderlich ist. Die Verwendung standardisierter Typen und die klare Trennung der Schnittstellen erleichtern die Integration in IEC 61499-basierte Steuerungssysteme.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

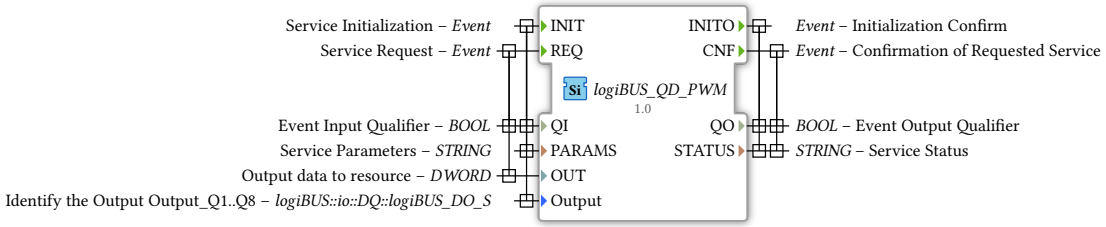
- 🌐 Das PWM-Signal & Infografik auf ms-muc-docs.de

logiBUS_QD_PWM

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `logiBUS_QD_PWM` ist ein Ausgabeservice-Interface-Funktionsblock für doppelte Wort-Ausgabedaten. Er dient als Schnittstelle zur Steuerung von PWM-Ausgaben (Pulsweitenmodulation) über das logiBUS-System und ermöglicht die Ansteuerung von Ausgängen Q1 bis Q8.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung
  - Verknüpft mit: QI, PARAMS, Output
- **REQ**: Service-Anforderung
  - Verknüpft mit: QI, OUT

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
  - Verknüpft mit: QO, STATUS
- **CNF**: Bestätigung der angeforderten Service-Operation
  - Verknüpft mit: QO, STATUS

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangsqualifizierer
- **PARAMS** (STRING): Service-Parameter
- **OUT** (DWORD): Ausgabedaten zur Ressource (13-Bit-PWM-Rohwert `0` bis `8191`, entsprechend `0 %` bis `100 %` Tastgrad, $2^{13} = 8192$ Zustände)
- **Output** (logiBUS_DO_S): Identifiziert den Ausgang Output_Q1..Q8
  - Initialwert: `logiBUS_DO::Invalid`

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifizierer
- **STATUS** (STRING): Service-Status

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Funktionsblock ermöglicht die PWM-Steuerung von Ausgängen über das logiBUS-System. Bei der Initialisierung (INIT) werden die Service-Parameter konfiguriert und der spezifische Ausgang identifiziert. Über die REQ-Anforderung können PWM-Daten (`OUT`, 13-Bit-Wert `0` bis `8191` in einer DWORD-Variable) an den konfigurierten Ausgang gesendet werden. Der Block bestätigt sowohl Initialisierung als auch Service-Anforderungen über die entsprechenden Ausgangsereignisse.

Technische Besonderheiten

- **13-Bit-PWM-Auflösung**: Verwendet eine 13-Bit-Normierung (`0` bis `8191`) im `DWORD`-Datentyp für die PWM-Ausgabe (`0` = 0 % Tastgrad, `8191` = 100 % Tastgrad, Skalierungsfaktor für ISO-Designer: `0.0122085215480405` bzw. `100 / 8191`).
- Unterstützt bis zu 8 Ausgänge (Q1-Q8) über die Output-Konfiguration
- String-basierte Parameterkonfiguration für flexible Service-Einstellungen
- Statusrückmeldung über STRING-Variable für detaillierte Fehlerinformationen

Zustandsübersicht

Der Funktionsblock verfügt über zwei Hauptzustände:

1. **Nicht initialisiert**: Block wartet auf INIT-Ereignis
2. **Initialisiert und betriebsbereit**: Block kann REQ-Anforderungen verarbeiten und PWM-Daten ausgeben

Anwendungsszenarien

- Steuerung von PWM-gesteuerten Aktoren (Motoren, Heizelemente)
- Ansteuerung von LED-Beleuchtung mit Helligkeitssteuerung
- Regelung von Ventilen mit proportionaler Steuerung
- Industrielle Automatisierungsanwendungen mit logiBUS-Hardware

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Ausgabeblöcken bietet `logiBUS_QD_PWM` eine präzise PWM-Steuerung mit 13-Bit-Auflösung (`0–8191` im DWORD). Gegenüber analogen Ausgabeblöcken ermöglicht er die direkte PWM-Steuerung ohne zusätzliche Wandlung.

🛠️ Zugehörige Übungen

- Uebung_034
- Uebung_034a1_Q1
- Uebung_034a1_Q2
- Uebung_034a1_Q4
- Uebung_034b
- Uebung_152
- Uebung_153

Fazit

Der `logiBUS_QD_PWM` Funktionsblock stellt eine leistungsstarke Schnittstelle für PWM-Ausgaben im logiBUS-System bereit. Durch die flexible Konfiguration und die Unterstützung für 13-Bit-PWM-Daten (`0–8191`) eignet er sich ideal für präzise Steuerungsanwendungen in industriellen Automatisierungssystemen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

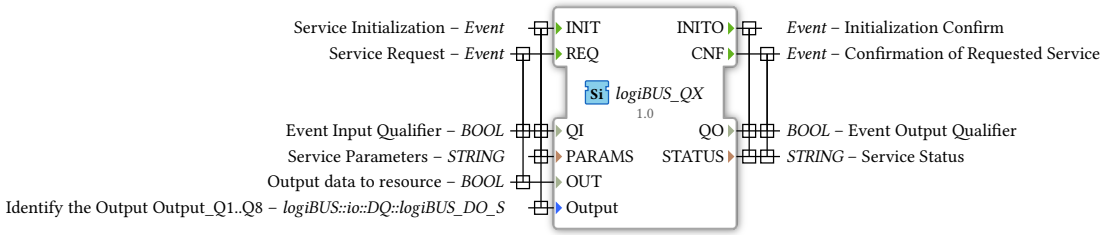
- 🌐 Das PWM-Signal & Infografik auf ms-muc-docs.de

logiBUS_QX

• – ◦ • – ◦ • – ◦ •

Einleitung

Der logiBUS_QX Funktionsblock ist ein Ausgabeservice-Interface-Baustein für boolesche Ausgabedaten. Er dient als Schnittstelle zur Steuerung von digitalen Ausgängen über das logiBUS-System und ermöglicht die Kommunikation zwischen der Steuerungsanwendung und den physischen Ausgangskanälen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierungsereignis
- **REQ**: Service-Anfrageereignis

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
- **CNF**: Bestätigung der angeforderten Service-Operation

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangsqualifizierer
- **PARAMS** (STRING): Service-Parameter
- **OUT** (BOOL): Ausgabedaten zur Ressource
- **Output** (logiBUS_DO_S): Identifiziert den Ausgang (Output_Q1..Q8), Initialwert: Invalid

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifizierer
- **STATUS** (STRING): Service-Statusinformation

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der logiBUS_QX-Block verwaltet die Kommunikation mit digitalen Ausgangskanälen des logiBUS-Systems. Bei Initialisierung (INIT-Event) werden die Service-Parameter konfiguriert und der spezifische Ausgangskanal identifiziert. Über das REQ-Event können boolesche Werte (OUT) an den konfigurierten Ausgangskanal gesendet werden. Der Block bestätigt jede Operation über die entsprechenden Ausgangsereignisse INITO und CNF.

Technische Besonderheiten

- Verwendet spezifische logiBUS-Datentypen für die Ausgangsidentifikation
- Unterstützt bis zu 8 digitale Ausgangskanäle (Q1-Q8)
- Bietet detaillierte Statusrückmeldungen über den STATUS-Ausgang
- Initialisiert Ausgänge mit einem ungültigen Zustand (Invalid)

Zustandsübersicht

Der Block durchläuft folgende Hauptzustände:

1. **Nicht initialisiert**: Vor der ersten INIT-Operation
2. **Initialisiert**: Nach erfolgreicher INIT-Operation, bereit für REQ-Operationen
3. **Aktiv**: Während der Verarbeitung von REQ-Operationen

Anwendungsszenarien

- Steuerung von digitalen Aktoren (Relais, Ventile, Lampen)
- Anbindung an logiBUS-Hardwareausgänge
- Automatisierungssysteme mit verteilten E/A-Komponenten
- Sicherheitsrelevante Schaltvorgänge mit Statusrückmeldung

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu standardmäßigen digitalen Ausgangsblöcken bietet logiBUS_QX:

- Spezifische Integration in logiBUS-Systeme
- Erweiterte Statusrückmeldungen
- Parametrierbare Ausgangskonfiguration
- Höhere Abstraktionsebene für Hardwarezugriffe

🛠️ Zugehörige Übungen

- Uebung_001
- Uebung_001c
- Uebung_002
- Uebung_002a
- Uebung_002a2
- Uebung_002a3
- Uebung_002a4
- Uebung_002a5b
- Uebung_002b2
- Uebung_002b3
- Uebung_003
- Uebung_003a0
- Uebung_003a_sub
- Uebung_003d
- Uebung_004a
- Uebung_004a2
- Uebung_004a2_2
- Uebung_004a2_3
- Uebung_004a3
- Uebung_004a4
- Uebung_004a5
- Uebung_004a6
- Uebung_004a7
- Uebung_004a8
- Uebung_004a9
- Uebung_004b
- Uebung_004b2
- Uebung_004b3
- Uebung_004c1
- Uebung_004c2
- Uebung_004c3
- Uebung_004c4
- Uebung_004c5
- Uebung_004c6
- Uebung_004c7
- Uebung_005
- Uebung_006
- Uebung_006a
- Uebung_006a2
- Uebung_006a3
- Uebung_006a4
- Uebung_006b
- Uebung_006c
- Uebung_006d

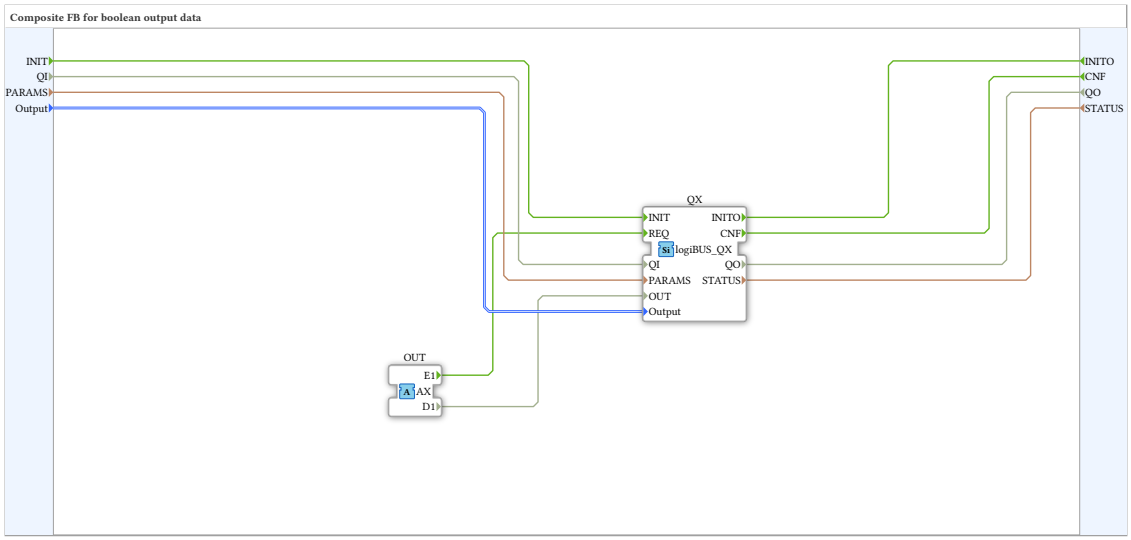
- Uebung_006e1
- Uebung_006e2
- Uebung_007
- Uebung_007a1
- Uebung_007a2
- Uebung_007a3
- Uebung_008
- Uebung_009
- Uebung_010
- Uebung_010a
- Uebung_010a2
- Uebung_010a3
- Uebung_010b1
- Uebung_010b2
- Uebung_010b3
- Uebung_010b6
- Uebung_010b7
- Uebung_010b8
- Uebung_010b9
- Uebung_010bA
- Uebung_010bA2
- Uebung_010bA3
- Uebung_010bA4
- Uebung_010c
- Uebung_010c2
- Uebung_013
- Uebung_019b
- Uebung_019c
- Uebung_020a
- Uebung_020b
- Uebung_020c
- Uebung_020c2
- Uebung_020c3
- Uebung_020d
- Uebung_020e
- Uebung_020e2
- Uebung_020f
- Uebung_020f2
- Uebung_020f3
- Uebung_020g
- Uebung_020h
- Uebung_020i
- Uebung_021
- Uebung_022
- Uebung_023
- Uebung_024
- Uebung_025
- SR_TO_QX_TRIG_EO
- Uebung_028
- Uebung_035
- Uebung_035a
- Uebung_035a2
- Uebung_035a3
- Uebung_035b
- Uebung_035c
- Uebung_036
- Uebung_037
- Uebung_038
- SoftkeyOverride_TO_QX_BG

- Softkey_T_FF_ILOCK_TO_QX_BG
- Uebung_039b
- Uebung_040
- Uebung_040_2
- Uebung_041
- Uebung_049
- Uebung_051
- Uebung_052
- Uebung_053
- Uebung_054
- Uebung_055
- Uebung_056
- Bool8_TO_logiBUS_QX8
- Uebung_071
- Uebung_071a
- Uebung_071b
- Uebung_072b
- Uebung_080
- Uebung_080b
- Uebung_080c
- Uebung_081
- Uebung_082
- Uebung_083
- Uebung_084
- Uebung_085
- Uebung_087
- Uebung_087a1
- Uebung_087a2
- Uebung_088
- Uebung_089
- Uebung_090a1
- Uebung_090a2
- Uebung_091
- Uebung_093
- Uebung_093b
- Uebung_094
- Uebung_094a
- Uebung_095
- Uebung_110
- Uebung_111
- Uebung_160
- Uebung_160b
- Uebung_160b2
- Uebung_177
- Uebung_178
- Uebung_179
- Uebung_180

Fazit

Der logiBUS_QX Funktionsblock stellt eine robuste und zuverlässige Schnittstelle für die Ansteuerung digitaler Ausgänge in logiBUS-basierten Automatisierungssystemen dar. Durch seine klare Ereignisstruktur und umfassende Statusrückmeldungen eignet er sich besonders für Anwendungen, die hohe Zuverlässigkeit und Nachverfolgbarkeit der Ausgabeoperationen erfordern.

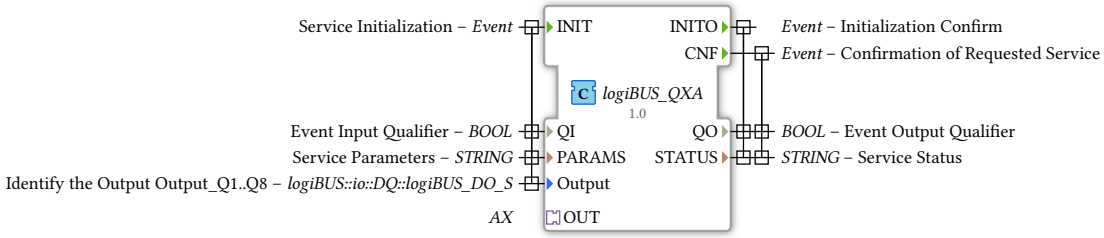
logiBUS_QXA



• – ○ • – ○ • – ○ •

Einleitung

Der logiBUS_QXA ist ein zusammengesetzter Funktionsblock für die Ausgabe von booleschen Daten. Er dient als Schnittstelle für digitale Ausgabefunktionen und ermöglicht die Steuerung von bis zu 8 digitalen Ausgängen über ein standardisiertes Protokoll.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierungsereignis mit den zugehörigen Daten QI, PARAMS und Output

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung mit den Daten QO und STATUS
- **CNF**: Bestätigung des angeforderten Services mit den Daten QO und STATUS

Daten-Eingänge

- **QI**: Boolescher Ereigniseingangsqualifizierer
- **PARAMS**: Service-Parameter als Zeichenkette
- **Output**: Identifizierung der Ausgänge Q1 bis Q8 vom Typ logiBUS_DO_S mit Initialwert „Invalid“

Daten-Ausgänge

- **QO**: Boolescher Ereignisausgangsqualifizierer
- **STATUS**: Service-Status als Zeichenkette

Adapter

- **OUT**: Unidirektionaler Adapter vom Typ AX für die Ausgabekommunikation

Funktionsweise

Der logiBUS_QXA fungiert als Wrapper für den logiBUS_QX-Funktionsblock und bietet eine vereinheitlichte Schnittstelle für digitale Ausgabefunktionen. Bei Initialisierung (INIT-Ereignis) werden die

Konfigurationsparameter übergeben und die Ausgänge entsprechend konfiguriert. Der Block ermöglicht die Steuerung von bis zu 8 digitalen Ausgängen über die Output-Datenstruktur.

Technische Besonderheiten

- Verwendet den logiBUS_QX-Kernfunktionsblock für die eigentliche Ausgabelogik
- Unterstützt bis zu 8 digitale Ausgänge (Q1 bis Q8)
- Initialisierung mit spezifischen Parametern über die PARAMS-Eingabe
- Rückmeldung des Betriebszustands über STATUS-Ausgabe

Zustandsübersicht

Der Funktionsblock durchläuft folgende Zustände:

1. **Nicht initialisiert**: Vor dem INIT-Ereignis
2. **Initialisierung**: Während der Verarbeitung des INIT-Ereignis
3. **Betriebsbereit**: Nach erfolgreicher Initialisierung (INITO-Bestätigung)
4. **Aktiver Betrieb**: Verarbeitung von Ausgabeanforderungen über den OUT-Adapter

Anwendungsszenarien

- Steuerung von digitalen Aktoren in Automatisierungssystemen
- Anbindung von Ausgabemodulen in verteilten Steuerungssystemen
- Integration in logiBUS-basierte Steuerungsarchitekturen
- Industrielle Automatisierung mit booleschen Ausgabesignalen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Ausgabeblöcken bietet logiBUS_QXA:

- Erweiterte Parametrierungsmöglichkeiten
- Statusrückmeldungen für Fehlerdiagnose
- Standardisierte Schnittstelle über Adapter
- Unterstützung für multiple Ausgänge in einer Struktur

🛠️ Zugehörige Übungen

- Uebung_001_AX
- Uebung_001_AX_b
- Uebung_001c_AX
- Uebung_002_AX
- Uebung_002a2_AX
- Uebung_002a3_AX
- Uebung_002a5_AX
- Uebung_002a5b_AX
- Uebung_002a6_AX
- Uebung_002a7_AX
- Uebung_002a_AX
- Uebung_002b3_AX
- Uebung_003_AX
- Uebung_003a0_AX
- Uebung_003d_AX
- Uebung_004a2_AX
- Uebung_004a3_AX
- Uebung_004a4_AX
- Uebung_004a5_AX
- Uebung_004a6_AX
- Uebung_004a7_AX
- Uebung_004a8_AX
- Uebung_004a9_AX
- Uebung_004a_AX
- Uebung_004b_AX
- Uebung_004b_AX_ASR
- Uebung_004b_AX_ASR_X
- Uebung_004c1_AX

- Uebung_004c2_AX
- Uebung_004c3_AX
- Uebung_004c4_AX
- Uebung_004c5_AX
- Uebung_004c6_AX
- Uebung_004c7_AX
- Uebung_005_AX
- Uebung_006_AX
- Uebung_006a2_AX
- Uebung_006a3_AX
- Uebung_006a4_AX
- Uebung_006a_AX
- Uebung_006b_AX
- Uebung_006d_AX
- Uebung_006e1_AX
- Uebung_006e2_AX
- Uebung_007_AX
- Uebung_007a1_AX
- Uebung_007a2_AX
- Uebung_007a3_AX
- Uebung_008_AX
- Uebung_009_AX
- Uebung_010_AX
- Uebung_010a2_AX
- Uebung_010a3_AX
- Uebung_010a_AX
- Uebung_010b1_AX
- Uebung_010b2_AX
- Uebung_010b3_AX
- Uebung_010b4_sub_AX
- Uebung_010b5_sub_AX
- Uebung_010b6_AX
- Uebung_010b7_AX
- Uebung_010b8_AX
- Uebung_010b9_AX
- Uebung_010bA2_AX
- Uebung_010bA3_AX
- Uebung_010bA4_AX
- Uebung_010bA_AX
- Uebung_010c2_AX
- AuxIXA_TO_logiBUS_QXA_BG
- Uebung_010c_AX
- Uebung_013_AX
- Uebung_020a_AX
- Uebung_020b_AX
- Uebung_020c3_AX
- Uebung_020c_AX
- Uebung_020d_AX
- Uebung_020e2_AX
- Uebung_020e_AX
- Uebung_020f2_AX
- Uebung_020f3_AX
- Uebung_020f_AX
- Uebung_020g_AX
- Uebung_020h_AX
- Uebung_020i_AX
- Uebung_020j2_AX
- Uebung_020j_AX
- Uebung_035a1_AX

- Uebung_035a1b_AX
- Uebung_035a2_AX
- Uebung_035a3_AX
- Uebung_038_AX
- Uebung_040_AX
- Uebung_083_AX
- Uebung_090a1_AX
- Uebung_090a2_AX
- Uebung_094a_AX
- Uebung_095_AX
- Uebung_103
- Uebung_103c
- Uebung_103c2
- Uebung_150_AX
- Uebung_151_AX
- Uebung_160_AX
- Uebung_160b2_AX
- Uebung_160b_AX
- Uebung_171_AX
- Uebung_177_AX
- Uebung_178_AX

Fazit

Der logiBUS_QXA ist ein robuster und flexibler Funktionsblock für digitale Ausgabefunktionen in industriellen Automatisierungssystemen. Durch seine standardisierte Schnittstelle und umfassende Parametrierungsmöglichkeiten eignet er sich ideal für den Einsatz in komplexen Steuerungsarchitekturen mit hohen Anforderungen an Zuverlässigkeit und Diagnosefähigkeit.

DO-LED

LED_COLOURS

• – ◦ • – ◦ • – ◦ •

Einleitung

LED_COLOURS ist eine globale Konstantendefinition, die standardisierte Farbwerte für LED-Anzeigen bereitstellt. Diese Konstanten werden in logiBUS-Systemen zur einheitlichen Steuerung von farbigen LED-Ausgängen verwendet.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereigniseingänge vorhanden

Ereignis-Ausgänge

Keine Ereignisausgänge vorhanden

Daten-Eingänge

Keine Dateneingänge vorhanden

Daten-Ausgänge

Keine Datenausgänge vorhanden

Adapter

Keine Adapter vorhanden

Funktionsweise

LED_COLOURS definiert eine Sammlung globaler Konstanten vom Typ UINT (16-Bit vorzeichenlose Ganzzahl), die verschiedene LED-Farben repräsentieren. Jede Farbe ist einem festen numerischen Wert zugeordnet:

- LED_RED: 1
- LED_YELLOW: 2
- LED_ORANGE: 3
- LED_GREEN: 4
- LED_BLUE: 5
- LED_WHITE: 6
- LED_MAGENTA: 7
- LED_CYAN: 8

Technische Besonderheiten

- Alle Konstanten sind vom Typ UINT (16-Bit unsigned integer)
- Wertebereich: 1-8 für die definierten Farben
- Konstanten sind zur Kompilierzeit festgelegt und unveränderlich
- Teil des logiBUS::io::DO_LED Packages

Zustandsübersicht

Da es sich um statische Konstanten handelt, existiert kein Zustandsmodell

Anwendungsszenarien

- Farbkodierung in Maschinensteuerungen
- Statusanzeigen mit farbigen LEDs
- Vereinheitlichung der Farbsteuerung in verteilten Systemen
- Visualisierung von Betriebszuständen in Industrieanlagen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu Funktionsblöcken mit dynamischem Verhalten handelt es sich hier um rein statische Konstantendefinitionen. Ähnliche Konstrukte wären:

- ENUM-Datentypen für Farbdefinitionen
- #define-Präprozessoranweisungen in C/C++
- Konstanten-Blöcke in anderen IEC 61131-Sprachen

Fazit

LED_COLOURS bietet eine standardisierte und wartungsfreundliche Lösung für die Farbsteuerung von LED-Anzeigen in logiBUS-Systemen. Durch die zentrale Definition werden Inkonsistenzen vermieden und die Code-Wiederverwendbarkeit erhöht.

LED_FREQ

• – ◦ • – ◦ • – ◦ •

Einleitung

LED_FREQ ist eine Sammlung globaler Konstanten für die Steuerung von LED-Blinkmustern mit Prioritätsverwaltung. Die Konstanten definieren verschiedene Blinkfrequenzen und Betriebsmodi für LEDs mit zugeordneten Prioritätswerten, die in Steuerungsanwendungen verwendet werden können.

Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

Keine Adapter vorhanden

Funktionsweise

Die globalen Konstanten stellen numerische Werte für verschiedene LED-Betriebsmodi bereit. Jeder Modus hat einen spezifischen Prioritätswert, wobei höhere Zahlen eine höhere Priorität darstellen. Die Konstanten können in Funktionsblöcken verwendet werden, um LED-Verhalten zu steuern und Prioritätskonflikte zu lösen.

Technische Besonderheiten

- **Datentyp**: Alle Konstanten sind vom Typ UINT (unsigned integer)
- **Prioritätsbereich**: Werte von 0 bis 6, wobei 6 die höchste Priorität darstellt
- **Frequenzbereiche**: 5Hz, 2Hz, 1Hz, 0.5Hz Blinken sowie Atmungseffekt
- **Package-Zugehörigkeit**: logiBUS::io::DO_LED

Zustandsübersicht

Keine Zustandsmaschine vorhanden - handelt sich um statische Konstanten

Anwendungsszenarien

- Steuerung von Status-LEDs in industriellen Steuerungssystemen
- Priorisierte Anzeigesteuerung in Maschinenbedienpanels
- Visualisierung von Betriebszuständen mit unterschiedlicher Dringlichkeit
- Mehrkanalige LED-Steuerung mit Prioritätsverwaltung

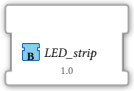
⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu Funktionsblöcken mit dynamischer Verhaltenssteuerung bietet LED_FREQ statische, vordefinierte Betriebsmodi mit festen Prioritäten. Dies ermöglicht eine einfachere Implementierung bei bekannten, standardisierten Anzeigeanforderungen.

Fazit

LED_FREQ bietet eine effiziente Methode zur Definition standardisierter LED-Betriebsmodi mit integrierter Prioritätsverwaltung. Die Konstantensammlung eignet sich besonders für Anwendungen, bei denen klare, vorhersehbare LED-Verhaltensmuster mit definierter Priorität benötigt werden.

LED_strip



• – ◦ • – ◦ • – ◦ •

Einleitung

LED_strip ist ein GlobalConstants-Baustein, der globale Konstanten für die Ansteuerung von LED-Streifen bereitstellt. Der Baustein definiert eine Ausgabevariable für die Steuerung von LED-Streifen und ist Teil des logiBUS-Systems.

Schnittstellenstruktur

Ereignis-Eingänge
Keine Ereignis-Eingänge vorhanden.

Ereignis-Ausgänge
Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge
Keine Daten-Eingänge vorhanden.

Daten-Ausgänge
Keine Daten-Ausgänge vorhanden.

Adapter
Keine Adapter vorhanden.

Funktionsweise
Der LED_strip-Baustein definiert eine globale Konstante `Output_strip` vom Typ USINT (Unsigned Short Integer) mit dem Initialwert 0. Diese Konstante dient als Referenz für die Ausgabesteuerung von LED-Streifen und kann von anderen Funktionsblöcken im System verwendet werden.

Technische Besonderheiten
- Die Konstante `Output_strip` ist vom Typ USINT (8-Bit unsigned integer)
- Initialwert ist 0
- Teil des logiBUS::io::DO_LED Packages
- Unterstützt die Eclipse Public License 2.0

Zustandsübersicht
Da es sich um einen GlobalConstants-Baustein handelt, gibt es keine Zustandsübergänge oder -verwaltung. Die definierte Konstante behält ihren festgelegten Wert während der gesamten Laufzeit.

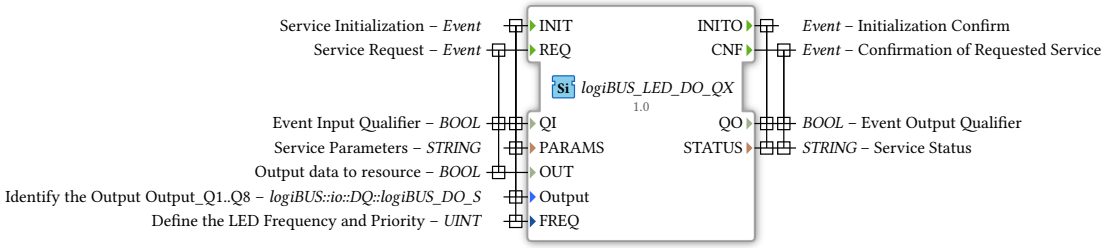
Anwendungsszenarien
- Definition von Ausgabekanälen für LED-Streifen
- Zentrale Verwaltung von Hardware-Ausgabereferenzen
- Verwendung in kombination mit digitalen Ausgabeblöcken
- Integration in Beleuchtungssteuerungssysteme

⚖️ Vergleich mit ähnlichen Bausteinen
Im Vergleich zu anderen GlobalConstants-Bausteinen bietet LED_strip eine spezifische Konstante für LED-Streifen-Steuerung. Während allgemeine Konstantenbausteine universelle Werte definieren, ist dieser Baustein auf die spezifischen Anforderungen der LED-Ansteuerung ausgelegt.

Fazit
Der LED_strip GlobalConstants-Baustein bietet eine einfache und effektive Möglichkeit, Ausgabereferenzen für LED-Streifen zentral zu verwalten. Durch die Verwendung als globale Konstante ermöglicht er eine konsistente und wartbare Integration in größere Steuerungssysteme.

logiBUS_LED_DO_QX
• – ◦ • – ◦ • – ◦ •

Einleitung
Der Funktionsblock logiBUS_LED_DO_QX ist ein Ausgabeservice-Interface-Funktionsblock für boolesche Ausgabedaten. Er dient zur Ansteuerung von LED-Ausgängen über das logiBUS-System und bietet spezielle Funktionen zur Frequenzsteuerung der LEDs.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung mit den zugehörigen Variablen QI, PARAMS, Output und FREQ
- **REQ**: Service-Anforderung mit den zugehörigen Variablen QI und OUT

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung mit den zugehörigen Variablen QO und STATUS
- **CNF**: Bestätigung der angeforderten Service-Durchführung mit den zugehörigen Variablen QO und STATUS

Daten-Eingänge

- **QI**: Ereignis-Eingangsqualifizierer (BOOL)
- **PARAMS**: Service-Parameter (STRING)
- **OUT**: Ausgabedaten an die Ressource (BOOL)
- **Output**: Identifiziert den Ausgang Output_Q1..Q8 (logiBUS::io::DQ::logiBUS_DO_S) - Initialwert: Invalid
- **FREQ**: Definiert die LED-Frequenz und Priorität (UINT) - Initialwert: LED_FREQ::LED_OFF

Daten-Ausgänge

- **QO**: Ereignis-Ausgangsqualifizierer (BOOL)
- **STATUS**: Service-Status (STRING)

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Funktionsblock ermöglicht die Ansteuerung von LED-Ausgängen mit konfigurierbaren Frequenzeinstellungen. Über den INIT-Ereigniseingang wird der Service initialisiert, wobei der spezifische Ausgang (Output_Q1 bis Q8) und die LED-Frequenz konfiguriert werden können. Der REQ-Ereigniseingang löst die eigentliche Ausgabeoperation aus, bei der der boolesche Wert an den konfigurierten Ausgang gesendet wird.

Technische Besonderheiten

- Unterstützt Frequenzsteuerung für LED-Betrieb (Blinken)
- Initialisierung mit Invalid-Wert für den Output
- Vordefinierte Frequenzkonstanten (LED_FREQ::LED_OFF als Standard)
- Spezifische Ausgangsidentifikation über den Output-Parameter

Zustandsübersicht

Der Funktionsblock durchläuft typische Service-Interface-Zustände:

1. Nicht initialisierter Zustand
2. Initialisierungsphase nach INIT-Ereignis
3. Betriebsbereiter Zustand nach erfolgreicher Initialisierung
4. Aktiver Betriebszustand während REQ-Verarbeitung

Anwendungsszenarien

- Ansteuerung von Status-LEDs in Automatisierungssystemen
- Visualisierung von Prozesszuständen mit blinkenden Signalen

- Integration in logiBUS-basierte Steuerungssysteme
- Prioritätsgesteuerte LED-Anzeigen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Ausgabeblöcken bietet logiBUS_LED_DO_QX erweiterte Funktionen zur Frequenzsteuerung und spezifische LED-Optimierungen. Während Standard-DO-Blöcke nur einfache Ein/Aus-Steuerung bieten, ermöglicht dieser Block komplexere Blinkmuster und Prioritätssteuerung.

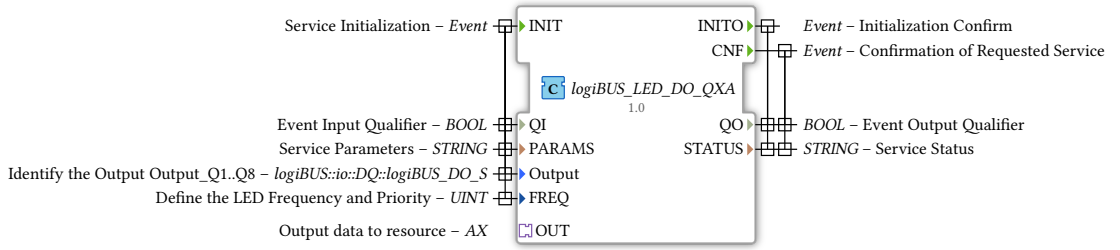
🛠️ Zugehörige Übungen

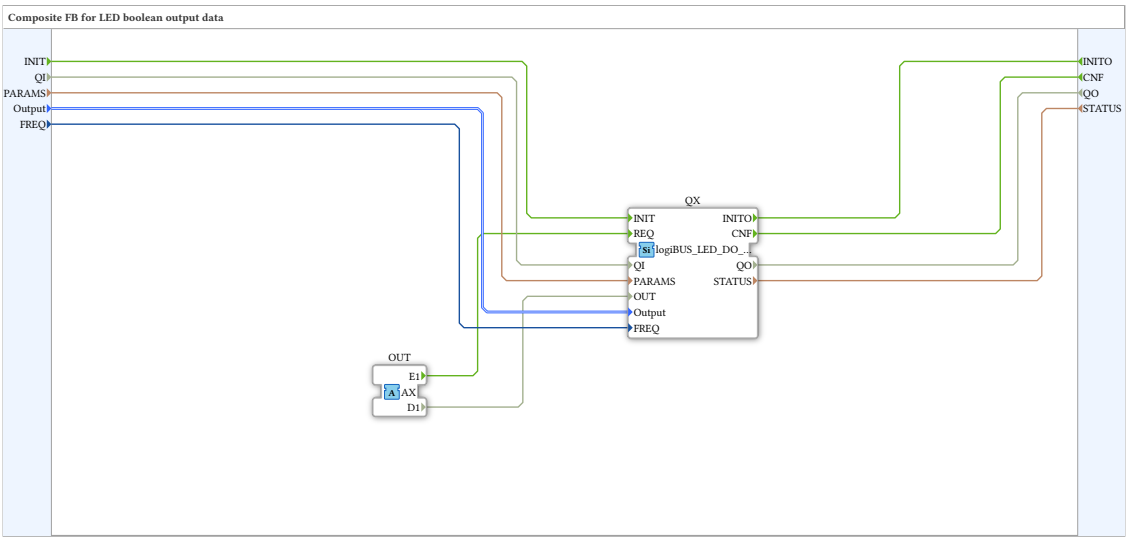
- Uebung_029

Fazit

Der logiBUS_LED_DO_QX Funktionsblock stellt eine spezialisierte Lösung für die LED-Ausgabesteuerung in logiBUS-Systemen dar. Durch die integrierte Frequenzsteuerung und die flexible Konfiguration der Ausgänge eignet er sich besonders für Anwendungen, die erweiterte Visualisierungsfunktionen erfordern.

logiBUS_LED_DO_QXA





• – ○ • – ○ • – ○ •

Einleitung

Der **logiBUS_LED_DO_QXA** ist ein zusammengesetzter Funktionsblock (Composite FB), der die Ansteuerung einer einzelnen LED über das logiBUS-System vereinfacht. Er nimmt einen booleschen Wert (z. B. von einer Steuerlogik) entgegen und konfiguriert den gewünschten LED-Ausgang (Q1 – Q8) mit einer einstellbaren Frequenz und Priorität. Der FB kapselt die Kommunikation mit dem zugrunde liegenden Hardwaretreiber und stellt eine standardisierte Schnittstelle zur Verfügung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisiert die gesamte Funktionslogik.
  Mitverkabelte Daten: `QI`, `PARAMS`, `Output`, `FREQ`

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der erfolgreichen Initialisierung.
  Mitverkabelte Daten: `QO`, `STATUS`
- **CNF** (Event): Quittierung eines abgeschlossenen Service-Auftrags (z. B. nach einer Wertänderung).
  Mitverkabelte Daten: `QO`, `STATUS`

Daten-Eingänge

- *QI* (BOOL): Ereignisqualifikator – schaltet die Verarbeitung des Eingangsereignisses frei.
- *PARAMS* (STRING): Service-Parameter, typischerweise Adress- oder Konfigurationsdaten für den logiBUS-Knoten.
- *Output* (logiBUS::io::DQ::logiBUS_DO_S): Identifikation des physischen Ausgangs (z. B. Q1 … Q8). Initialwert: `Invalid`.
- *FREQ* (UINT): Definiert die gewünschte LED-Frequenz und Priorität. Initialwert: `LED_FREQ::LED_OFF`.

Daten-Ausgänge

- *QO* (BOOL): Ausgangsqualifikator – gibt den Zustand der letzten Verarbeitung an.
- *STATUS* (STRING): Meldung über den aktuellen Dienstzustand (z. B. Fehlercode oder „OK“).

Adapter

- **OUT** (adapter::types::unidirectional::AX): Unidirektionaler Adapter zur Übergabe des booleschen Wertes an die Ressource.
  - *E1* (Ereignis): Signalisiert eine neue Datenanforderung.
  - *D1* (BOOL): Der zu übertragende Wert (LED ein/aus).

Funktionsweise

1. **Initialisierung**
   Der FB wird durch ein INIT-Ereignis gestartet. Die mitgelieferten Daten (`QI`, `PARAMS`, `Output`, `FREQ`) werden an den inneren FB `logiBUS_LED_DO_QX` weitergeleitet, der die eigentliche logiBUS-Kommunikation durchführt. Bei Erfolg wird `INITO` mit den Ausgangsdaten `QO` und `STATUS` gesendet.
2. **Betrieb**
   Nach der Initialisierung wartet der FB auf das Ereignis `OUT.E1`. Dieses wird von der angeschlossenen Ressource ausgelöst, sobald ein neuer boolescher Wert (`OUT.D1`) anliegt. Das Ereignis wird als `REQ` an den inneren Baustein weitergegeben. Dieser aktualisiert daraufhin den LED-Ausgang gemäß der konfigurierten `Output`-Identifikation und der angegebenen `FREQ`.
3. **Bestätigung**
   Nach erfolgreicher Verarbeitung sendet der innere FB ein `CNF`-Ereignis, das als `CNF` nach außen gegeben wird. Die Ausgänge `QO` und `STATUS` zeigen den aktuellen Zustand an.

Technische Besonderheiten

- **Composite-FB** – Der Baustein vereinfacht die Nutzung des komplexen Treiber-FBs `logiBUS_LED_DO_QX` durch eine reduzierte Schnittstelle.
- **Adapter-Gestaltung** – Der Adapter `OUT` ermöglicht eine lose Kopplung zwischen Steuerlogik und Ausgabekanal. Der boolesche Wert wird ereignisgesteuert übergeben.
- **LED-Frequenz** – Der Parameter `FREQ` nutzt eine vordefinierte Enumeration (`LED_FREQ`), um Blinkmuster oder Prioritäten festzulegen (z. B. `LED_OFF`, `LED_ON`, `BLINK_1HZ`, …).
- **Fehlerbehandlung** – Der Ausgang `STATUS` gibt detaillierte Meldungen (z. B. von `logiBUS::io::DQ::logiBUS_DO::Invalid`) zurück.

Zustandsübersicht

Der FB besitzt keine expliziten Zustandsautomaten im XML-Netzwerk. Die Zustandslogik wird durch das Zusammenspiel von INIT und den Ereignis-/Datenverbindungen realisiert:

- **Ruhezustand** – Nach erfolgreicher Initialisierung wartet der FB auf Ereignisse vom Adapter.
- **Verarbeitung** – Ein `OUT.E1`-Ereignis löst eine Aktualisierung des LED-Ausgangs aus.
- **Fehlerzustand** – Bei fehlgeschlagener Initialisierung oder Kommunikationsfehler wird `STATUS` entsprechend gesetzt.

Anwendungsszenarien

- **Land- und Agrartechnik** – Ansteuerung von Kontrollleuchten an Maschinen, Anzeige von Betriebszuständen über logiBUS.
- **Automatisierung** – Einfache Ein-/Aus-Signale auf verteilte I/O-Module, z. B. Schaltschrank-LEDs.
- **Kabellose Anbindung** – Nutzung des logiBUS-Protokolls zur Überbrückung größerer Distanzen zwischen Steuerung und Aktor.

Vergleich mit ähnlichen Bausteinen

Gegenüber dem direkten Treiber-FB `logiBUS_LED_DO_QX` bietet `logiBUS_LED_DO_QXA` eine schlankere Schnittstelle und integriert die Initialisierung in einem Schritt. Andere Composite-FBs der `logiBUS_DO`-Familie fokussieren auf digital-binäre Ausgänge, während dieser Baustein speziell für LED-Anwendungen (mit Frequenzunterstützung) ausgelegt ist.

Fazit

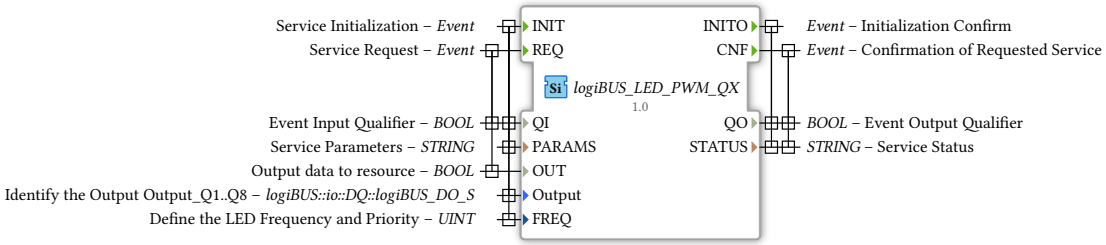
Der `logiBUS_LED_DO_QXA` ist ein komfortabler Composite-FB für die LED-Ausgabe über logiBUS. Er reduziert die Implementierungskomplexität auf wenige Parameter und einen Ereignis-Adapter. Durch seine klare Struktur eignet er sich besonders für wiederkehrende Schaltaufgaben in verteilten Steuerungen.

logiBUS_LED_PWM_QX

• – ◦ • – ◦ • – ◦ •

Einleitung

Der logiBUS_LED_PWM_QX ist ein Ausgabeservice-Interface-Funktionsblock für boolesche Ausgabedaten mit PWM-Funktionalität für LED-Ansteuerung. Der Baustein ermöglicht die Ansteuerung von LED-Ausgängen mit konfigurierbaren Frequenzen und bietet eine standardisierte Schnittstelle für die Kommunikation mit logiBUS-I/O-Ressourcen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung mit Parametern QI, PARAMS, Output und FREQ
- **REQ**: Service-Anforderung mit Parametern QI und OUT

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung mit Parametern QO und STATUS
- **CNF**: Bestätigung der angeforderten Service-Operation mit Parametern QO und STATUS

Daten-Eingänge

- **QI**: BOOL - Ereignis-Eingangsqualifizierer
- **PARAMS**: STRING - Service-Parameter
- **OUT**: BOOL - Ausgabedaten zur Ressource
- **Output**: logiBUS_DO_S - Identifiziert den Ausgang (Output_Q1..Q8), Initialwert: Invalid
- **FREQ**: UINT - Definiert die LED-Frequenz und Priorität, Initialwert: LED_FREQ::LED_OFF

Daten-Ausgänge

- **QO**: BOOL - Ereignis-Ausgangsqualifizierer

- **STATUS**: STRING - Service-Status

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Funktionsblock verwaltet die PWM-Ansteuerung von LED-Ausgängen über das logiBUS-System. Bei der Initialisierung (INIT-Event) werden die Ausgangskonfiguration, Frequenzeinstellungen und Service-Parameter übergeben. Über das REQ-Event können Ausgabedaten (OUT) gesendet werden, die dann mit der konfigurierten PWM-Frequenz auf den spezifizierten Ausgang (Output) angewendet werden.

Technische Besonderheiten

- Unterstützt verschiedene LED-Frequenzen über den FREQ-Parameter
- Integrierte Fehlerbehandlung durch STATUS-Ausgabe
- Typisierte Ausgangsidentifikation über logiBUS_DO_S-Datentyp
- Initialisierung mit Invalid-Zustand für nicht konfigurierte Ausgänge
- Frequenzprioritätssteuerung für LED-Ansteuerung

Zustandsübersicht

Der Baustein verfügt über folgende Betriebszustände:

- **Nicht initialisiert**: Vor INIT-Event, Ausgänge inaktiv
- **Initialisiert**: Nach erfolgreicher INIT-Verarbeitung, bereit für REQ-Operationen
- **Aktiv**: Während der PWM-Signalgenerierung
- **Fehler**: Bei ungültigen Parametern oder Konfigurationsfehlern

Anwendungsszenarien

- Ansteuerung von Status-LEDs in industriellen Steuerungssystemen
- PWM-gesteuerte Beleuchtungssysteme
- Visuelle Signalgebung mit variabler Helligkeitssteuerung
- Prioritätsgesteuerte LED-Anzeigen in logiBUS-basierten Anwendungen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Ausgangsbausteinen bietet logiBUS_LED_PWM_QX erweiterte PWM-Funktionalität mit Frequenzsteuerung und prioritätsbasierter Ansteuerung. Die Integration in das logiBUS-Ökosystem ermöglicht eine standardisierte Kommunikation mit I/O-Ressourcen.

🛠️ Zugehörige Übungen

- Uebung_030

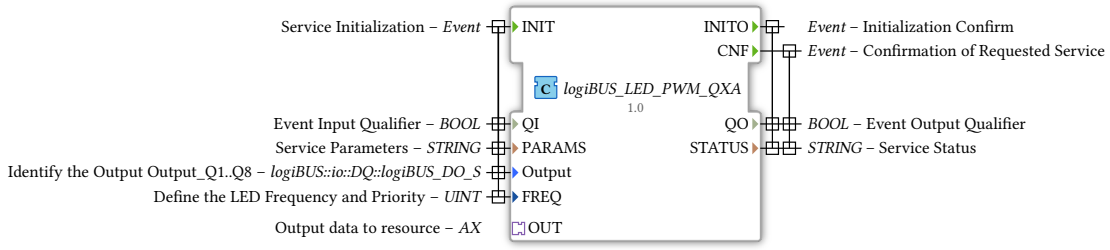
Fazit

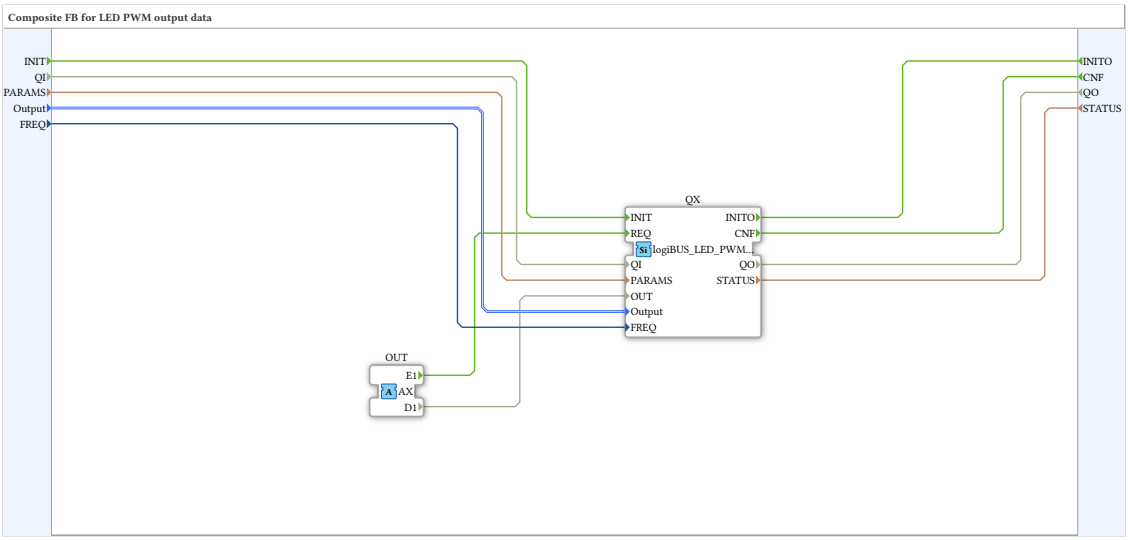
Der logiBUS_LED_PWM_QX ist ein spezialisierter Funktionsblock für anspruchsvolle LED-Ansteuerungsanwendungen in industriellen Automatisierungssystemen. Durch die PWM-Funktionalität, konfigurierbare Frequenzen und die Integration in das logiBUS-System bietet er eine leistungsstarke Lösung für visuelle Ausgabeszenarien mit hohen Anforderungen an Flexibilität und Zuverlässigkeit.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Das PWM-Signal & Infografik auf ms-muc-docs.de

logiBUS_LED_PWM_QXA





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `logiBUS_LED_PWM_QXA` ist ein Composite-Baustein zur Steuerung einer PWM-gesteuerten LED über den logiBUS. Er fasst die Konfiguration und Ausgabe eines LED-PWM-Signals zusammen und stellt eine einheitliche Schnittstelle für Initialisierung, Parametrierung und Betrieb bereit.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Mit Variablen	Beschreibung
`INIT`	EInit	`QI`, `PARAMS`, `Output`, `FREQ`	Service-Initialisierung; startet den FB mit den angegebenen Parametern

Ereignis-Ausgänge

Ereignis	Typ	Mit Variablen	Beschreibung
`INITO`	EInit	`QO`, `STATUS`	Bestätigung der erfolgreichen Initialisierung
`CNF`	Event	`QO`, `STATUS`	Bestätigung eines angeforderten Dienstes (z. B. nach Datenanforderung über den Adapter)

Daten-Eingänge

Variable	Typ	Initialwert	Beschreibung
`QI`	BOOL	–	Qualifikator für den Ereigniseingang (Aktivierung)

PARAMS	STRING	–	Service-Parameter für die Bus-Konfiguration
Output	logiBUS::io::DQ::logiBUS_DO_S	Invalid	Auswahl des physikalischen Ausgangs Q1..Q8
FREQ	UINT	LED_OFF	Frequenz und Priorität der LED-PWM (z. B. aus Enumeration LED_FREQ)

Daten-Ausgänge

Variable	Typ	Beschreibung
QO	BOOL	Qualifikator für den Ereignisausgang (Aktivierungsquittung)
STATUS	STRING	Dienststatus (Fehler-/Erfolgsmeldung)

Adapter

Adapter	Typ	Beschreibung
OUT	adapter::types::unidirectional::AX	Ausgangs-Adapter für die Datenübertragung zur logiBUS-Ressource (über Event E1 und Data D1)

Funktionsweise

Der Baustein ist als Composition realisiert und enthält intern den Sub-FB logiBUS_LED_PWM_QX, der die eigentliche PWM-Ansteuerung umsetzt.

1. **Initialisierung (INIT):** Der FB wird durch das Ereignis INIT gestartet. Dabei werden die Parameter QI, PARAMS, Output und FREQ an den Sub-Baustein weitergeleitet. Nach erfolgreicher Initialisierung gibt der Sub-Baustein INITO und die Ausgangsdaten QO und STATUS zurück.
2. **Betrieb:** Der Adapter OUT empfängt von der Ressource ein Ereignis an E1. Dieses wird als REQ an den Sub-FB durchgeschaltet. Der Sub-FB verarbeitet die Anfrage und sendet über den Adapter-Datenkanal OUT.D1 die Ausgangsdaten (den aktuellen PWM-Wert sowie das Status-Signal) an die Ressource. Die Bestätigung CNF des Sub-FBs wird als CNF des Composite-FBs ausgegeben.
3. **Fehlerbehandlung:** Treten während der Initialisierung oder im Betrieb Fehler auf, wird der Status über die Variable STATUS gemeldet. Die Ausgangs-Validierung erfolgt über QO.

Technische Besonderheiten

- **Composite-Baustein**: Der FB kapselt die Komplexität der PWM-Ausgabe und bietet eine einfache Schnittstelle für den Anwender.
- **Typisierte Ausgangsauswahl**: Der Eingang Output vom Typ logiBUS_DO_S erlaubt die explizite Zuordnung zu einem logiBUS-Digitalausgang Q1..Q8. Der Initialwert Invalid erzwingt vor der ersten Nutzung eine gültige Auswahl.
- **Frequenzvorgabe**: Über FREQ (Typ UINT) kann die PWM-Frequenz und Priorität definiert werden. Der Initialwert LED_OFF schaltet die LED standardmäßig aus.
- **Adapterbasierte Kommunikation**: Der unidirektionale Adapter AX überträgt Ereignisse und Daten zwischen dem FB und der übergeordneten logiBUS-Ressource. Der FB selbst wartet auf externe Anfragen (E1) und liefert Daten zurück.

Zustandsübersicht

Da es sich um einen Composite-Baustein handelt, besitzt er keine expliziten eigenen Zustände. Der interne Sub-FB logiBUS_LED_PWM_QX kann jedoch Zustände wie „Initialisierung“, „Bereit“ oder „Fehler“ durchlaufen. Diese werden indirekt über die Ereignisausgänge und die Statusvariable widergespiegelt:

- **Nach INIT**: Der FB befindet sich im initialisierten Zustand (sofern QO = TRUE und STATUS = „OK“).
- **Nach CNF**: Eine angeforderte Aktion (z. B. Aufruf über Adapter) wurde bestätigt. Der FB bleibt betriebsbereit.

- **Fehlerzustand**: Falls `STATUS` einen Fehlertext enthält, muss der FB erneut initialisiert werden. Eine Wiederholung des `INIT`-Ereignisses setzt alle Parameter zurück.

Anwendungsszenarien

- **PWM-gesteuerte Beleuchtung**: Einsatz in landwirtschaftlichen Maschinen oder Automatisierungssystemen zur Einstellung der Helligkeit von LED-Arbeitsleuchten.
- **Lichtsignalanlagen**: Steuerung mehrerer LED-Ausgänge mit unterschiedlichen Frequenzen (z. B. Blinklicht, Dauerlicht).
- **Dimmen über logiBUS**: Einbettung in ein logiBUS-Netzwerk, bei dem die PWM-Werte zentral oder dezentral berechnet werden.

Vergleich mit ähnlichen Bausteinen

- **logiBUS_DO_S** (direkter Digitalausgang): Dieser Baustein schaltet nur binäre Ausgänge (Ein/Aus) ohne PWM. `logiBUS_LED_PWM_QXA` bietet dagegen eine PWM-Funktion für gedimmte LEDs.
- **logiBUS_LED_PWM_QX**: Dies ist der interne Sub-Baustein, der die reine PWM-Logik ausführt. Der Composite-FB `QXA` erweitert ihn um den Adapteranschluss und die Standard-Schnittstellendefinition, was die Integration in höherwertige Steuerungen vereinfacht.
- **Generische PWM-FBs**: Im Vergleich zu plattformunabhängigen PWM-Bausteinen ist dieser FB speziell auf die logiBUS-Hardware zugeschnitten und bietet die dort üblichen Parameter und Typen.

Fazit

Der `logiBUS_LED_PWM_QXA` Funktionsblock kapselt die PWM-Ansteuerung einer LED über logiBUS in einem kompakten Composite-Baustein. Durch die klar definierte Schnittstelle mit Initialisierung, Adapterkommunikation und Statusrückmeldung ist er leicht in Automatisierungsprojekte integrierbar. Die Kombination aus Ausgangsauswahl, Frequenzvorgabe und Fehlerbehandlung macht ihn zu einem robusten Baustein für dimmbare Beleuchtungsaufgaben im industriellen Umfeld.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

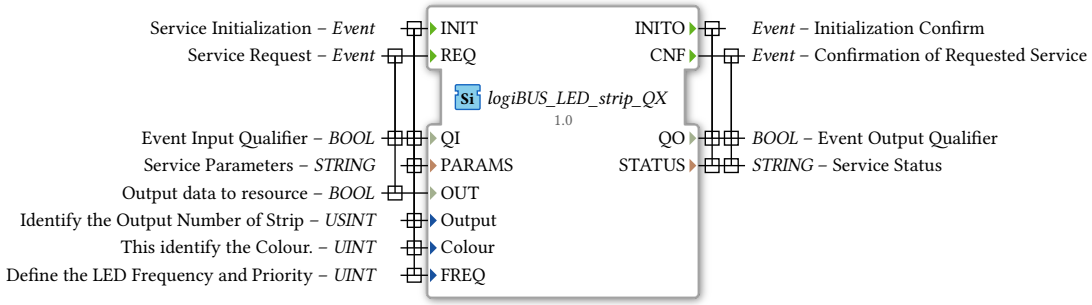
- 🌐 Das PWM-Signal & Infografik auf ms-muc-docs.de

logiBUS_LED_strip_QX

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock logiBUS_LED_strip_QX ist ein Ausgabeservice-Interface-Funktionsblock für boolesche Ausgabedaten, der speziell für die Ansteuerung von LED-Streifen entwickelt wurde. Er bietet umfangreiche Konfigurationsmöglichkeiten für verschiedene LED-Parameter wie Farbe, Frequenz und Ausgangsnummer.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierungsereignis
- **REQ**: Service-Anfrageereignis

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
- **CNF**: Bestätigung der angeforderten Service-Operation

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangsqualifizierer
- **PARAMS** (STRING): Service-Parameter
- **OUT** (BOOL): Ausgabedaten an die Ressource
- **Output** (USINT): Identifiziert die Ausgangsnummer des Streifens (Initialwert: LED_strip::Output_strip)
- **Colour** (UINT): Identifiziert die Farbe (Initialwert: LED_COLOURS::LED_GREEN)
- **FREQ** (UINT): Definiert die LED-Frequenz und Priorität (Initialwert: LED_FREQ::LED_OFF)

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifizierer
- **STATUS** (STRING): Service-Status

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Funktionsblock ermöglicht die Steuerung von LED-Streifen über eine standardisierte Schnittstelle. Bei der Initialisierung (INIT) werden die Konfigurationsparameter wie Ausgangsnummer, Farbe und Frequenz gesetzt. Über das REQ-Ereignis können Ausgabedaten (OUT) an den LED-Streifen gesendet werden. Der Block bestätigt jede Operation über die entsprechenden Ausgangsereignisse INITO und CNF.

Technische Besonderheiten

- Unterstützung verschiedener LED-Farben über den Colour-Parameter
- Konfigurierbare Frequenzeinstellungen für Blinkfunktionen
- Mehrfachausgänge über den Output-Parameter unterstützt
- Initialwerte für schnelle Inbetriebnahme vordefiniert
- Statusrückmeldung über STRING-Parameter

Zustandsübersicht

Der Funktionsblock verfügt über zwei Hauptzustände:

1. **Initialisierungszustand**: Wird durch INIT-Ereignis aktiviert
2. **Betriebszustand**: Verarbeitet REQ-Anfragen nach erfolgreicher Initialisierung

Anwendungsszenarien

- Industrielle Beleuchtungssteuerung
- Statusanzeigen in Automatisierungsanlagen
- Warn- und Signallichtsteuerung
- Visualisierung von Prozesszuständen
- Gebäudeautomation mit LED-Beleuchtung

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Ausgabeblöcken bietet logiBUS_LED_strip_QX erweiterte Funktionalität für LED-spezifische Anwendungen, insbesondere durch die integrierte Farb- und Frequenzsteuerung sowie die Möglichkeit, mehrere LED-Streifen unabhängig zu adressieren.

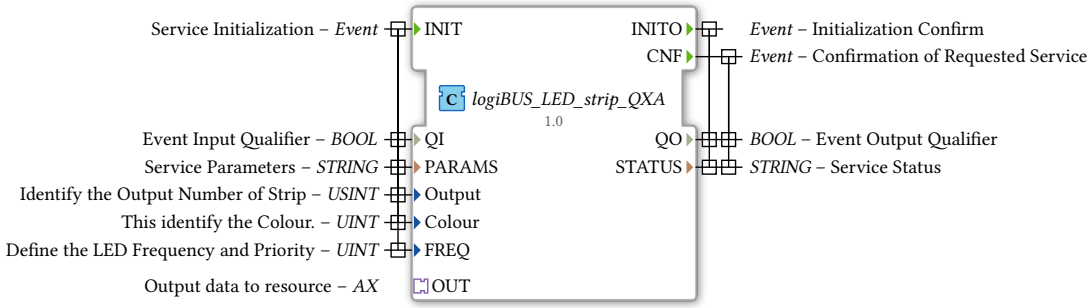
🛠️ Zugehörige Übungen

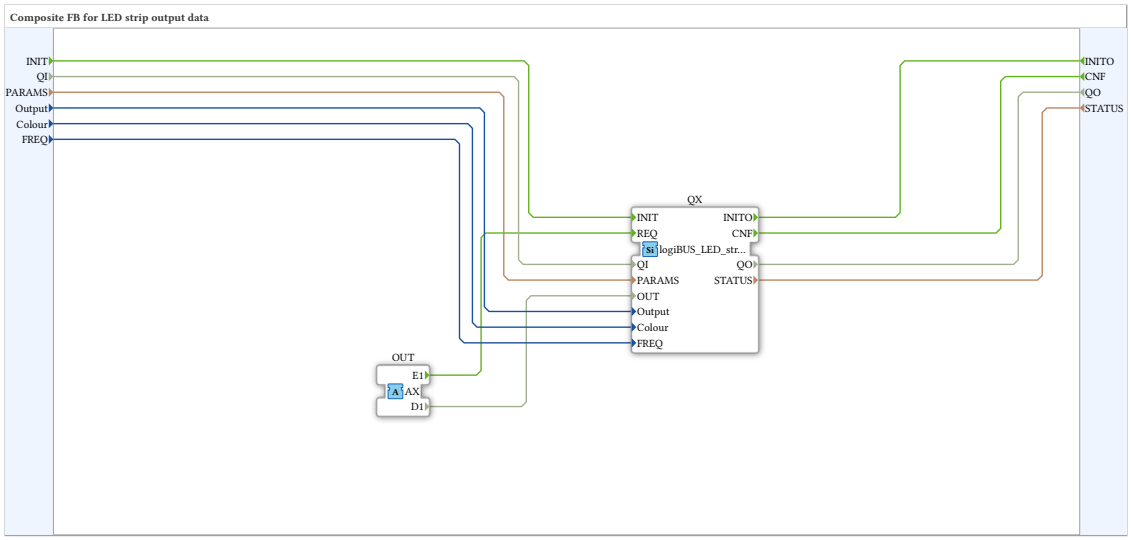
- Uebung_032
- IX_TO_logiBUS_LED_strip_QX

Fazit

Der logiBUS_LED_strip_QX ist ein leistungsstarker Funktionsblock für die professionelle LED-Streifen-Steuerung in industriellen Automatisierungslösungen. Seine flexible Parametrierung und zuverlässige Statusrückmeldung machen ihn ideal für anspruchsvolle Beleuchtungsanwendungen.

logiBUS_LED_strip_QXA





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **logiBUS_LED_strip_QXA** ist ein Composite-FB zur Ansteuerung eines LED-Stripes über das logiBUS-Protokoll. Er kapselt die Kommunikation mit der Hardware und ermöglicht die farb- und frequenzabhängige Steuerung einzelner Ausgänge. Der Baustein eignet sich insbesondere für den Einsatz in der Agrartechnik, wo flexible LED-Signalisierung gefordert ist.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung	Mit
INIT	Service-Initialisierung	QI, PARAMS, Output, Colour, FREQ

Ereignis-Ausgänge

Ereignis	Beschreibung	Mit
INITO	Bestätigung der Initialisierung	QO, STATUS
CNF	Bestätigung einer ausgeführten Anforderung	QO, STATUS

Daten-Eingänge

Name	Typ	Beschreibung	Initialwert
QI	BOOL	Freigabe der Verarbeitung (Ereignisqualifizierer)	-
PARAMS	STRING	Service-Parameter (z.B. Buskonfiguration)	-

Output	USINT	Identifikation des Ausgangs (Strip-Nummer)	`LED_strip::Output_strip`
Colour	UINT	Farbcode des LEDs	`LED_COLOURS::LED_GREEN`
FREQ	UINT	Frequenz/Priorität der LED-Anzeige	`LED_FREQ::LED_OFF`

Daten-Ausgänge

Name	Typ	Beschreibung
QO	BOOL	Ausgangsqualifizierer (Status der Verarbeitung)
STATUS	STRING	Statusmeldung (z.B. Fehlercode)

Adapter

Adapter	Typ	Beschreibung
OUT	`adapter::types::unidirectional::AX`	Unidirektionale Adapter-Schnittstelle zur Datenübergabe an die Ressource (Ausgangsdaten zum logiBUS)

Funktionsweise

Der Baustein arbeitet als Composite-FB, der intern den Unterbaustein **logiBUS_LED_strip_QX** verwendet. Nach einem **INIT**-Ereignis werden die Parameter (QI, PARAMS, Output, Colour, FREQ) an den internen FB übergeben. Die Initialisierung wird über **INITO** quittiert.

Sobald über den **OUT*-Adapter ein Ereignis (E1) eintrifft, wird im internen FB eine **REQ**-Anforderung ausgelöst. Die aktuellen Daten (Colour, FREQ) werden an den logiBUS-LED-Strip gesendet. Nach erfolgreicher Ausführung quittiert der interne FB mit **CNF**, was am äußeren Ausgang **CNF** erscheint.

Datenflüsse:

- **Out.D1** ⟶ interner FB.QX.OUT (Ausgangsdaten des Adapters)
- **Qi**, **PARAMS**, **Output**, **Colour**, **FREQ** ⟶ entsprechend an QX weitergeleitet.

Die Ausgänge **QO** und **STATUS** spiegeln den internen Zustand des Unterbausteins wider.

Technische Besonderheiten

- Composite-FB: erleichtert die Wiederverwendung und Kapselung der Hardware-Ansteuerung.
- Verwendung eines Adapters (OUT) für die unidirektionale Datenübergabe an die logiBUS-Ressource.
- Initialwerte der Parameter sind als Konstanten (z.B. `LED_strip::Output_strip`, `LED_COLOURS::LED_GREEN`) vordefiniert, können aber zur Laufzeit überschrieben werden.
- Der interne FB `logiBUS_LED_strip_QX` ist für die eigentliche Buskommunikation verantwortlich; dieser Baustein bietet nur eine vereinfachte Schnittstelle.
- Copyright und Entwickler: HR Agrartechnik GmbH (Version 1.0, 2026-02-23).

Zustandsübersicht

Der Baustein besitzt keine explizit modellierten Zustände, sein Verhalten ergibt sich aus der Ablaufsteuerung des Composite-Netzwerks:

1. **Initialisierungszustand**: Nach **INIT** werden der interne FB konfiguriert und die Busverbindung aufgebaut.
2. **Bereitschaftszustand**: Nach erfolgreicher Initialisierung kann der Baustein über den Adapter (OUT.E1) angesprochen werden.
3. **Ausführungszustand**: Die Anforderung wird an den LED-Strip gesendet, bis die Bestätigung (**CNF**) erfolgt.
4. **Fehlerzustand**: Falls QO = FALSE oder STATUS einen Fehler enthält, ist die Kommunikation gestört.

Anwendungsszenarien

- **Landwirtschaftliche Maschinen**: Farbcodierte Statusanzeigen (z.B. grün für Betriebsbereit, rot für Alarm) an LED-Strips.
- **Feldrandbeleuchtung**: Ansteuerung mehrerer LED-Strip-Ausgänge mit unterschiedlichen Farben und Blinkfrequenzen.

- **Benutzerdefinierte Signalisierung**: Einbindung in übergeordnete Steuerungen zur Visualisierung von Betriebszuständen.
- **Entwicklung und Test**: Einsatz als einfacher Baustein zur Simulation und Inbetriebnahme von logiBUS-Komponenten.

Vergleich mit ähnlichen Bausteinen

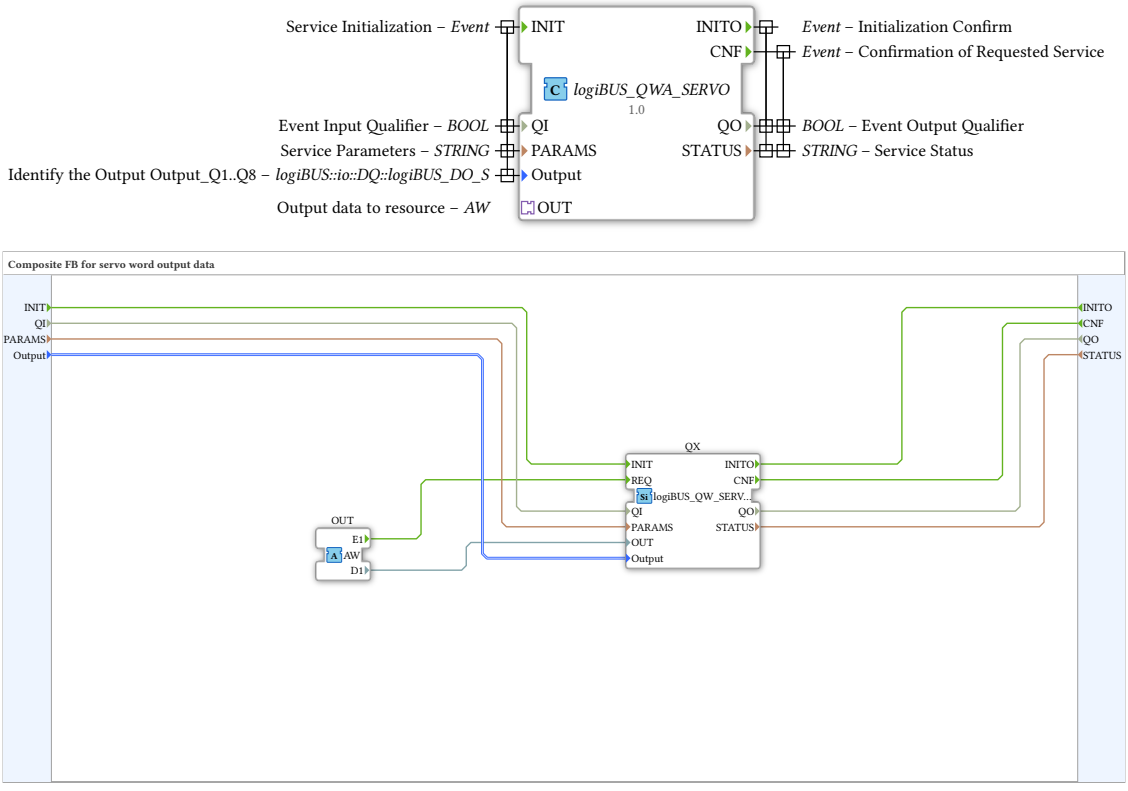
- **logiBUS_LED_strip_QX** (direkter FB): Bietet eine detailliertere Schnittstelle mit mehreren Ereignissen (REQ) und Daten. Der hier beschriebene Composite-FB **QXA** vereinfacht die Nutzung durch eine Adapter-Schnittstelle und klare Trennung von Initialisierung und Ausführung.
- **logiBUS_DO_Bit**: Steuert einzelne Digitalausgänge, keine Farb-/Frequenzsteuerung.
- **Composite-FBs im Allgemeinen**: Der QXA ist spezifisch für LED-Strip-Anwendungen konzipiert und reduziert den Verdrahtungsaufwand im übergeordneten Netzwerk.

Fazit

Der **logiBUS_LED_strip_QXA** bietet eine komfortable und standardisierte Möglichkeit, LED-Strips über logiBUS farblich und in der Blinkfrequenz zu steuern. Durch die Adapter-basierte Schnittstelle lässt er sich leicht in größere Systeme integrieren, ohne dass Details der Buskommunikation bekannt sein müssen. Der Composite-Ansatz sorgt für Übersichtlichkeit und Wiederverwendbarkeit.

DO-Servo

logiBUS_QWA_SERVO



• – ○ • – ○ • – ○ •

Einleitung

Der **logiBUS_QWA_SERVO** ist ein Composite-Funktionsblock (CFB) für die Ausgabe von Servo-Steuerworten über einen logiBUS. Er kapselt die Initialisierung, Parametrierung und Ausgabe auf Basis eines internen Bausteins (`logiBUS_QW_SERVO`) und stellt eine standardisierte Schnittstelle zur Verfügung. Entwickelt unter der Eclipse Public License 2.0 (Copyright 2026 HR Agrartechnik GmbH).
Kommentar aus der XML: „Composite FB for servo word output data“.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Mit Variablen	Beschreibung
INIT	EInit	QI, PARAMS, Output	Service Initialisierung

Ereignis-Ausgänge

Ereignis	Typ	Mit Variablen	Beschreibung
INITO	EInit	QO, STATUS	Bestätigung der Initialisierung
CNF	Event	QO, STATUS	Bestätigung einer angeforderten Aktion

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung
QI	BOOL	–	Ereignis-Eingangsqualifizierer
PARAMS	STRING	–	Service-Parameter (z. B. Bus-Konfiguration)
Output	logiBUS::io::DQ::logiBUS_DO_S	Invalid	Identifiziert den Ausgang (Output_Q1 … Output_Q8)

Daten-Ausgänge

Name	Typ	Beschreibung
QO	BOOL	Ereignis-Ausgangsqualifizierer
STATUS	STRING	Statusmeldung (Fehler, OK, etc.)

Adapter

Adapter	Typ	Richtung	Beschreibung
OUT	adapter::types::unidirectional::AW	Socket	Ausgangsdaten zur Ressource (Wortausgabe)

Funktionsweise

Der Baustein arbeitet als Composite:

1. **Initialisierung** (INIT):
   - Qualifizierer QI, Parameterstring PARAMS und der gewünschte Ausgang (Output) werden übergeben.
   - Der interne FB logiBUS_QW_SERVO wird initialisiert; nach erfolgreicher Initialisierung wird INITO gesendet.
2. **Ausführung** (über Adapter OUT):
   - Das Ereignis OUT.E1 wird vom Adapter empfangen und an den internen FB als REQ weitergeleitet.
   - Gleichzeitig werden die Daten über OUT.D1 an den internen FB (QX.OUT) übergeben.
3. **Rückmeldung**:
   - Der interne FB signalisiert Abschluss über CNF, das an den externen Ausgang CNF weitergereicht wird.
   - Status und Qualifizierer werden über QO und STATUS ausgegeben.

Die eigentliche Ansteuerung der Servo-Hardware erfolgt über den Adapter OUT, der mit der Ressource verbunden wird.

Technische Besonderheiten

- **Composite-Design**: Kapselt die komplexe Initialisierungs- und Ablaufsteuerung des Servo-Wortausgangs.

- **Adapter-Schnittstelle**: Der unidirektionale Adapter `AW` ermöglicht eine flexible Anbindung an verschiedene physische Ausgangskanäle.
- **Ausgangsauswahl**: Über den Eingang `Output` kann zwischen acht verschiedenen Ausgängen (`Q1`…`Q8`) gewählt werden; der Initialwert `Invalid` verhindert ungewollte Aktionen vor der Konfiguration.
- **Fehlerbehandlung**: Statusinformationen werden über `STATUS` bereitgestellt; Event-Sequenzen garantieren eine korrekte Ablaufsteuerung.
- **Lizenz**: Eclipse Public License 2.0 (EPL-2.0) – freie Nutzung und Weiterentwicklung unter den Lizenzbedingungen.

Zustandsübersicht

Der FB besitzt keine explizit modellierte Zustandsmaschine, jedoch ergibt sich aus den Ereignis-Abläufen folgende implizite Zustandslogik:

Zustand	Auslöser	Reaktion
Idle	–	Warten auf `INIT`
Initialize	`INIT` empfangen	Interner FB wird gestartet
Ready	`INITO` gesendet	Bereit für Anforderungen über den Adapter
Busy	`OUT.E1` empfangen	Aktion wird ausgeführt (`REQ` läuft)
Done	`CNF` gesendet	Aktion abgeschlossen, zurück zu **Ready**
Error	Fehler während Initialisierung oder Aktion	`STATUS` enthält Fehlerbeschreibung; evtl. erneute Initialisierung nötig

Anwendungsszenarien

- **Landwirtschaftliche Maschinen**: Ansteuerung von Servoantrieben (z. B. für Klappen, Ventile, Aktuatoren) über einen logiBUS-Fieldbus.
- **Modularer Ausgangsbaustein**: Einsatz in verteilten Steuerungen, bei denen mehrere Servo-Ausgänge je nach Maschinenkonfiguration parametriert werden müssen.
- **Parametrierbare Konfiguration**: Über `PARAMS` können Bus-Adresse, Protokoll-Einstellungen oder Zeitverhalten definiert werden.

Vergleich mit ähnlichen Bausteinen

- **logiBUS_QW_SERVO** (interner FB): Realisiert die direkte Wortausgabe und Bus-Kommunikation, während `logiBUS_QWA_SERVO` eine höhere Abstraktionsebene mit konsolidierten Ein-/Ausgängen bietet.
- **logiBUS_DO_S-Familie**: Andere Bausteine dieser Familie adressieren einzelne diskrete Ausgänge (z. B. logiBUS_DO) – der hier beschriebene FB ist speziell für Servo-Wortdaten und mehrere simultane Ausgänge ausgelegt.
- **Generische Wort-Ausgangs-FBs**: Im Vergleich zu standardisierten `W`-Typ-Bausteinen bietet dieser FB eine an den Servo-Anwendungsfall angepasste Schnittstelle (z. B. `logiBUS_DO_S`-Typ).

Fazit

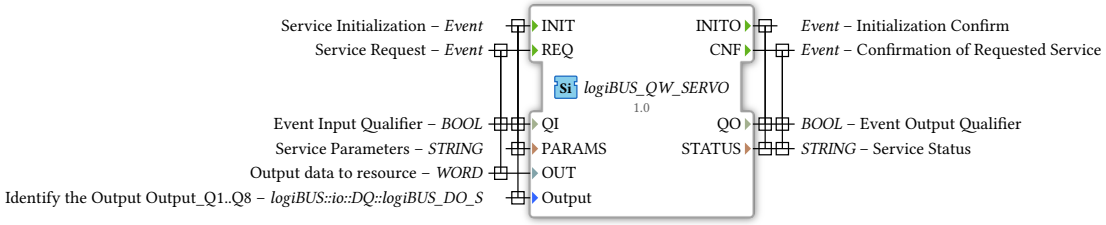
Der `logiBUS_QWA_SERVO` ist ein gut strukturierter Composite-Funktionsblock, der die Ansteuerung von Servo-Wortausgängen auf einem logiBUS vereinfacht. Durch die Trennung von Initialisierung, Ausführung und Fehlerrückmeldung sowie die Verwendung eines Adapters für die Hardwareschnittstelle eignet er sich hervorragend für modulare Automatisierungslösungen in der Landtechnik und ähnlichen Industrien. Die Lizenzierung unter EPL-2.0 ermöglicht uneingeschränkte Nutzung und Anpassung.

logiBUS_QW_SERVO

• – ◦ • – ◦ • – ◦ •

Einleitung

Der logiBUS_QW_SERVO ist ein Ausgabeservice-Schnittstellen-Funktionsblock für Wort-Ausgabedaten, der speziell für die Ansteuerung von logiBUS-Servoausgängen entwickelt wurde. Der Baustein ermöglicht die Initialisierung und Steuerung von digitalen Ausgängen über eine standardisierte Schnittstelle.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierungsereignis mit den zugehörigen Daten QI, PARAMS und Output
- **REQ**: Service-Anfrageereignis mit den zugehörigen Daten QI und OUT

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung mit den zugehörigen Daten QO und STATUS
- **CNF**: Bestätigung der angeforderten Service-Operation mit den zugehörigen Daten QO und STATUS

Daten-Eingänge

- **QI** (BOOL): Ereigniseingangs-Qualifier
- **PARAMS** (STRING): Service-Parameter für die Konfiguration
- **OUT** (WORD): Ausgabedaten zur Ressource
- **Output** (logiBUS::io::DQ::logiBUS_DO_S): Identifiziert den Ausgang Output_Q1..Q8, initial mit Wert „Invalid“

Daten-Ausgänge

- **QO** (BOOL): Ereignisausgangs-Qualifier
- **STATUS** (STRING): Service-Statusinformation

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Funktionsblock ermöglicht die Initialisierung und Steuerung von logiBUS-Servoausgängen. Bei der INIT-Initialisierung werden die Service-Parameter konfiguriert und der spezifische Ausgang identifiziert. Über das REQ-Ereignis können Ausgabedaten im WORD-Format an die Ressource gesendet werden. Der Baustein bestätigt beide Operationen durch entsprechende Ausgangsereignisse (INITO und CNF) mit Statusinformationen.

Technische Besonderheiten

- Verwendet WORD-Datentyp für Ausgabedaten (16-Bit)
- Unterstützt spezifische logiBUS-Ausgangsidentifikation über den Output-Parameter
- Initialisierung mit „Invalid“-Wert für den Output-Parameter
- String-basierte Parameter- und Statusübertragung

Zustandsübersicht

Der Baustein verfügt über zwei Hauptbetriebszustände:

1. **Nicht initialisiert**: Vor der INIT-Initialisierung
2. **Initialisiert und betriebsbereit**: Nach erfolgreicher INIT-Initialisierung, bereit für REQ-Anfragen

Anwendungsszenarien

- Ansteuerung von digitalen Ausgängen in logiBUS-Systemen
- Servosteuerungen mit Wort-Datenausgabe
- Industrielle Automatisierungsanwendungen mit präziser Ausgangsansteuerung
- Embedded-Systeme mit logiBUS-Interface

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Ausgabeblöcken bietet logiBUS_QW_SERVO erweiterte Funktionalität durch:

- Wort-basierte Datenausgabe (16-Bit statt einzelner Bits)
- Spezifische Servo-Steuerungsfunktionen
- Detaillierte Statusrückmeldungen
- Parametrierbare Initialisierung

Fazit

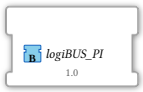
Der logiBUS_QW_SERVO ist ein spezialisierter Funktionsblock für anspruchsvolle Ausgabeanwendungen in logiBUS-Systemen. Durch seine Wort-basierte Datenverarbeitung und umfassende Statusrückmeldung eignet er sich besonders für präzise Steuerungsaufgaben in industriellen Automatisierungslösungen.

PI

🎧 Podcast

- Decoding IEC 61499: The Industrial Revolution Reshaping Factories and Careers
- ISOBUS: Wie Logos auf euer Traktor-Terminal kommen – Das Picture Graphic Objekt erklärt
- From „Mass Errors“ to Masterpiece: Streamlining Industrial Software by Eliminating Mapping
- Logibus Revolution: Unpacking the „No Mapping“ Future and Streamlined Development
- Logibus Unleashed: How Eliminating „Mapping“ Simplifies Complex Systems and Boosts Usability

logiBUS_PI



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `logiBUS_PI` ist eine Sammlung globaler Konstanten für die Konfiguration von Puls-Eingängen (Pulse Inputs) in logiBUS-Systemen. Er definiert vorkonfigurierte Strukturen, die die physischen Eingangspins eines logiBUS I/O-Moduls repräsentieren. Diese Konstanten dienen der einfachen und fehlerfreien Zuweisung von Hardware-Pins in übergeordneten Funktionsblöcken oder Anwendungen.

Schnittstellenstruktur

Bei `logiBUS_PI` handelt es sich um einen GlobalConstants-Block, nicht um einen klassischen Funktionsblock mit Ereignis- oder Datenschnittstellen. Daher besitzt er keine Ereignis-Eingänge, -Ausgänge, Daten-Eingänge, -Ausgänge oder Adapter.

Die Schnittstelle besteht ausschließlich aus den definierten globalen Konstanten. Jede Konstante ist vom Datentyp `logiBUS::io::PI::logiBUS_PI_S` und wird mit einem spezifischen Initialwert instanziiert.

Definierte Konstanten

Die folgenden Konstanten werden bereitgestellt:

- `PulseInput_I1` bis `PulseInput_I9`: Repräsentieren die physischen Puls-Eingänge 1 bis 9. Der Initialwert setzt das Strukturfeld `Pin` auf den entsprechenden Wert (1-9).
- `Invalid`: Stellt einen ungültigen oder nicht belegten Eingang dar. Der Initialwert setzt das Strukturfeld `Pin` auf den Wert 255.

Funktionsweise

Der Block `logiBUS_PI` selbst führt keine aktive Funktionalität aus. Er stellt lediglich eine Bibliothek von vordefinierten Werten bereit. Diese Werte sind vom Typ `logiBUS_PI_S`, einer Struktur, die vermutlich mindestens das Mitglied `Pin` enthält.

Die Konstanten werden zur Kompilierzeit in den Anwendungscode eingefügt. Wenn ein anderer Funktionsblock (z.B. ein Treiber- oder Messbaustein) einen Parameter vom Typ `logiBUS_PI_S` erwartet, kann eine dieser Konstanten übergeben werden, um eindeutig einen bestimmten Hardware-Eingang auszuwählen.

Technische Besonderheiten

- **Typisierung:** Alle Konstanten sind streng vom strukturierten Datentyp `logiBUS_PI_S`. Dies erzwingt eine typsichere Verwendung und verhindert die unbeabsichtigte Zuweisung einfacher Integer-Werte.
- **Initialisierung:** Die Initialwerte werden in der ST-Syntax `(Pin := X)` angegeben, was auf eine strukturierte Initialisierung hindeutet.
- **Ungültiger Zustand:** Die Konstante `Invalid` mit dem Pin-Wert 255 bietet eine standardisierte Möglichkeit, einen nicht genutzten oder fehlerhaften Kanal zu kennzeichnen.

Zustandsübersicht

Da es sich um einen Konstanten-Block handelt, besitzt `logiBUS_PI` keinen internen Zustand oder Zustandsautomaten. Die Werte der Konstanten sind unveränderlich (read-only) und während der gesamten Laufzeit der Anwendung konstant.

Anwendungsszenarien

- **Hardware-Konfiguration:** Primärer Einsatz zur Konfiguration von logiBUS I/O-Modulen, die Puls-Eingänge (z.B. für Encoder, Frequenz- oder Drehzahlmessung) bereitstellen.
- **Parametrierung von Funktionsblöcken:** Vereinfachte und lesbare Parametrierung von Funktionsblöcken, die einen spezifischen Puls-Eingang benötigen. Statt der magischen Zahl `3` wird die selbsterklärende Konstante `PulseInput_I3` verwendet.
- **Wartung und Lesbarkeit:** Erhöht die Wartbarkeit und Lesbarkeit des Anwendungscodes, da die Hardware-Zuordnung direkt im Konstantennamen ersichtlich ist.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu Funktionsblöcken wie `E_SR` (Flip-Flop) oder `E_CYCLE` (zyklischer Event-Generator) implementiert `logiBUS_PI` keine Logik oder Algorithmen. Er ist vergleichbar mit anderen Konstanten-Blöcken oder ENUM-Definitionen in der IEC 61499, die hardwarenahe Parameter kapseln (z.B. `GPIO_PIN_XX` Konstanten für Raspberry Pi). Sein Zweck ist rein deklarativ und konfigurativ.

Fazit

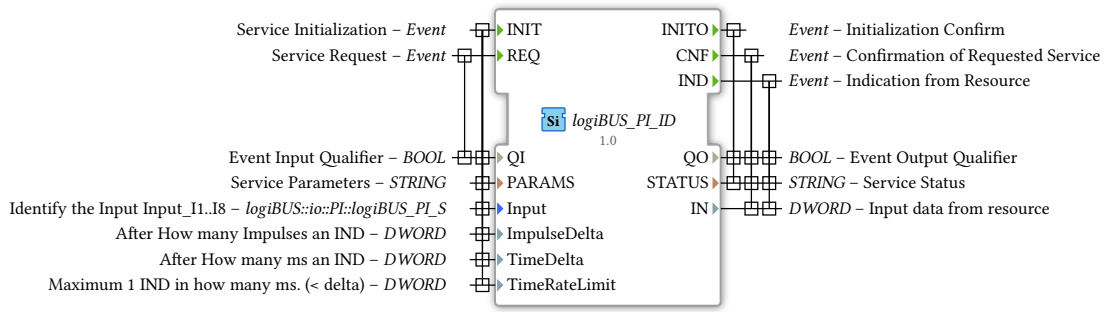
Der `logiBUS_PI` GlobalConstants-Block ist ein essentielles Hilfsmittel für die hardwarenahe Programmierung von logiBUS-Systemen. Durch die Bereitstellung typsicher, vordefinierter Konstanten für Puls-Eingänge ermöglicht er eine saubere, fehlerresistente und gut wartbare Konfiguration der Ein-/Ausgabe-Hardware. Seine Verwendung fördert Best Practices in der industriellen Automatisierungsprogrammierung.

logiBUS_PI_ID

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `logiBUS_PI_ID` ist ein Eingabeservice-Interface-Baustein für 32-Bit-DWORT-Eingabedaten. Er dient als Schnittstelle zu einem physischen Eingabemodul (vermutlich Teil des logiBUS-Systems) und ermöglicht die Initialisierung, zyklische Abfrage (Polling) und ereignisgesteuerte (Interrupt) Ausgabe von digitalen Eingangswerten. Der Baustein kann so konfiguriert werden, dass er Statusänderungen entweder nach einer bestimmten Anzahl von Impulsen oder nach einem definierten Zeitintervall meldet.



Schnittstellenstruktur

Ereignis-Eingänge

- **`INIT`**: Initialisiert den Service. Begleitet von den Daten `QI`, `PARAMS`, `Input`, `ImpulseDelta`, `TimeDelta` und `TimeRateLimit`.
- **`REQ`**: Löst eine zyklische Abfrage (Polling) des Eingangswerts aus. Begleitet von dem Daten `QI`.

Ereignis-Ausgänge

- **`INITO`**: Bestätigt die Initialisierung. Liefert die Daten `QO` und `STATUS`.
- **`CNF`**: Bestätigt eine angeforderte Abfrage (`REQ`). Liefert die Daten `QO`, `STATUS` und den aktuellen Eingangswert `IN`.
- **`IND`**: Zeigt eine ereignisgesteuerte Statusänderung (Interrupt) an. Liefert die Daten `QO`, `STATUS` und den neuen Eingangswert `IN`.

Daten-Eingänge

- **`QI` (BOOL)**: Qualifiziert den zugehörigen Ereigniseingang. Bei `TRUE` wird der Service aktiviert/ausgeführt, bei `FALSE` deaktiviert.
- **`PARAMS` (STRING)**: Enthält service-spezifische Parameter für die Initialisierung (z.B. Hardware-Adresse, Kanal-Konfiguration).
- **`Input` (logiBUS_PI_S)**: Identifiziert den konkreten physischen Eingang (z.B. I1..I8). Der Initialwert ist `logiBUS_PI::Invalid`.
- **`ImpulseDelta` (DWORD)**: Definiert, nach wie vielen aufeinanderfolgenden Zustandsänderungen (Impulsen) ein `IND`-Ereignis generiert werden soll.
- **`TimeDelta` (DWORD)**: Definiert das Zeitintervall in Millisekunden, nach dem ein `IND`-Ereignis generiert werden soll, wenn sich der Wert geändert hat.
- **`TimeRateLimit` (DWORD)**: Maximal ein `IND`-Ereignis pro angegebener Zeit in Millisekunden (muss kleiner als `TimeDelta` sein). Der Initialwert ist `100`.

Daten-Ausgänge

- **`QO` (BOOL)**: Zeigt den Status der Serviceausführung an. `TRUE` bedeutet erfolgreich, `FALSE` signalisiert einen Fehler.
- **`STATUS` (STRING)**: Liefert eine detaillierte Status- oder Fehlermeldung vom Service.
- **`IN` (DWORD)**: Enthält den aktuellen, vom physischen Eingang gelesenen 32-Bit-Wert.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Der Baustein arbeitet in zwei grundlegenden Modi, die durch die Ereignisse `REQ` und `IND` gesteuert werden:

1. **Polling-Modus**: Durch ein `REQ`-Ereignis wird der aktuelle Eingangswert abgefragt und das Ergebnis mit einem `CNF`-Ereignis zurückgemeldet.
2. **Interrupt-Modus**: Nach erfolgreicher Initialisierung (`INIT`/`INITO`) überwacht der Baustein den Eingang kontinuierlich. Bei einer Zustandsänderung werden die Parameter `ImpulseDelta` und `TimeDelta` ausgewertet. Wird eines der Kriterien erfüllt (z.B. die definierte Anzahl an Impulsen erreicht oder das Zeitintervall überschritten *und* der Wert hat sich geändert), wird automatisch ein `IND`-Ereignis mit dem neuen Wert ausgelöst.

Die Initialisierung (`INIT`) ist Voraussetzung für beide Betriebsarten. Dabei werden die Hardware-Ressource über `PARAMS` und der spezifische Eingang über `Input` konfiguriert.

Technische Besonderheiten

- **Datentyp**: Verarbeitet 32-Bit-Eingabedaten (`DWORD`).
- **Strukturierter Eingang**: Der Eingang wird nicht über einen einfachen Index, sondern über einen strukturierten Datentyp (`logiBUS_PI_S`) identifiziert, was eine typsichere und eindeutige Adressierung ermöglicht.
- **Flexible Ereignisauslösung**: Die Bedingungen für die automatische Ereignisgenerierung (`IND`) können sowohl impuls- als auch zeitbasiert konfiguriert werden.
- **Service-Interface**: Folgt dem typischen Muster eines 4diac-Service-Interface-FBs mit `QI`/`QO` und `STATUS`-Variablen für einheitliches Fehlerhandling.

Zustandsübersicht

1. **Inaktiv**: Nach dem Start oder bei `QI=FALSE`.
2. **Initialisierung**: Bei Empfang von `INIT` mit `QI=TRUE`. Konfiguriert die Hardware-Schnittstelle. Endet mit `INITO` (`QO` zeigt Erfolg/Fehler an).
3. **Bereit (Polling)**: Nach erfolgreicher Initialisierung. Auf `REQ`-Ereignisse wird mit `CNF` und dem aktuellen Wert geantwortet.
4. **Bereit (Überwachung)**: Nach erfolgreicher Initialisierung. Überwacht den Eingang kontinuierlich und löst `IND`-Ereignisse gemäß der konfigurierten `ImpulseDelta`- und `TimeDelta`-Parameter aus.

Anwendungsszenarien

- **Einlesen von Zählersignalen**: Erfassung von Impulsen eines Drehgebers oder Encoders, wobei `ImpulseDelta` für eine Vorverarbeitung (z.B. jede 10. Umdrehung melden) genutzt wird.
- **Überwachung von Statusgruppen**: Lesen eines 32-Bit-Statusworts eines angeschlossenen Geräts, wobei Änderungen nur in bestimmten Zeitabständen (`TimeDelta`) gemeldet werden müssen, um die CPU-Last zu reduzieren.
- **Zyklische Abfrage von Schalterbänken**: Polling mehrerer digitaler Eingänge, die zu einem DWORD zusammengefasst sind, über regelmäßige `REQ`-Ereignisse.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Gegenüber `E_DEMUX` oder `E_SELECT`**: Diese Bausteine leiten Ereignisse weiter oder selektieren Daten. `logiBUS_PI_ID` ist spezifisch für die Hardware-Kommunikation und beinhaltet Treiberlogik sowie Initialisierung.
- **Gegenüber generischen I/O-FBs (z.B. `WAGO_750_5xx_DI`)**: Ähnliche Funktion, aber herstellerspezifisch (hier logiBUS). Die Konfiguration erfolgt über den strukturierten `Input`-Parameter und `PARAMS` anstelle fester Kanalnummern.
- **Gegenüber einfacheren Eingabeblöcken**: Bietet erweiterte Funktionen wie filternde Ereignisgenerierung (`IND`) über `ImpulseDelta`/`TimeDelta`, die in einfachen „Read“-Blöcken typischerweise nicht vorhanden sind.

🛠️ Zugehörige Übungen

- Uebung_150
- Uebung_150_AX
- Uebung_151
- Uebung_151_AX
- Uebung_152
- Uebung_153

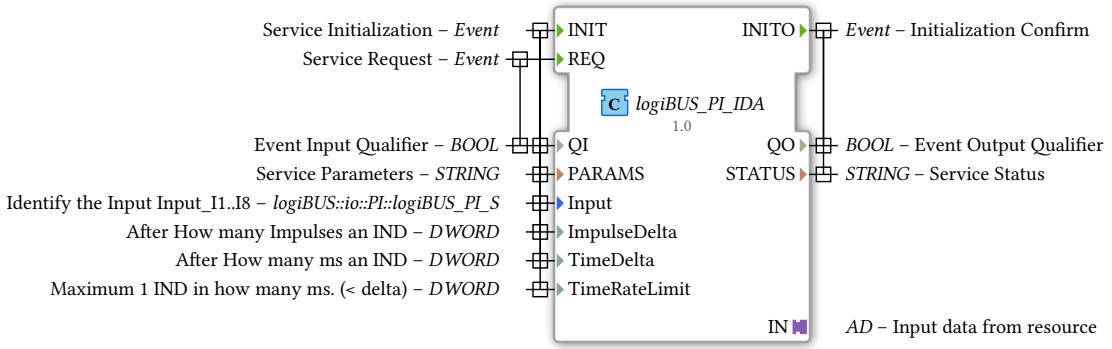
Fazit

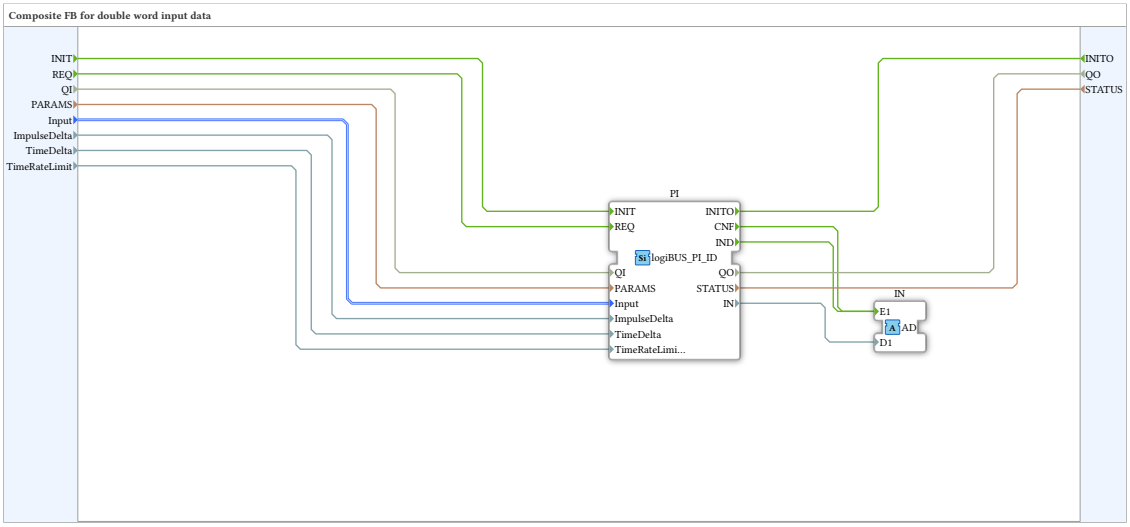
Der `logiBUS_PI_ID`-Funktionsblock ist ein leistungsfähiges und flexibles Interface für 32-Bit-Digitaleingänge innerhalb des logiBUS-Ökosystems. Durch die Kombination von Polling (`REQ`/`CNF`) und ereignisgesteuerter Abfrage (`IND`) mit konfigurierbaren Auslösekriterien eignet er sich für eine Vielzahl von Anwendungen, von der einfachen Statusabfrage bis zur komplexen Impulsauswertung. Die strikte Trennung von Initialisierungs- und Betriebslogik sowie das umfangreiche Status-Reporting machen ihn zu einem robusten Baustein für industrielle Steuerungsanwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

logiBUS_PI_IDA





• – ○ • – ○ • – ○ •

Einleitung

Der **logiBUS_PI_IDA** ist ein Composite Funktionsblock (FB) zur Verarbeitung von Doppelwort-Eingangsdaten (DWORD). Er dient als Schnittstelle zwischen einem logiBUS-Feldbus und der Steuerungslogik. Der FB ermöglicht die parametrierbare Überwachung von Impuls- und Zeitänderungen an einem digitalen Eingang. Durch die Konfiguration von Impuls-Delta, Zeit-Delta und einer Ratenbegrenzung kann das Meldeverhalten exakt auf die Anwendung abgestimmt werden. Der Baustein initialisiert den internen Unterbaustein `logiBUS_PI_ID` und stellt dessen Ergebnisse über einen unidirektionalen Adapter bereit.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Service Initialization – Parametrierung und Start
REQ	Event	Service Request – löst eine Verarbeitung aus

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Initialization Confirm – bestätigt erfolgreiche Initialisierung

Daten-Eingänge

Name	Typ	Initialwert	Kommentar
QI	BOOL	–	Event Input Qualifier (freigeben/sperren)
PARAMS	STRING	–	Service Parameters (z. B. Busadresse, Kanalkonfiguration)
Input	logiBUS::io::PI::logiBUS_PI_S	`Invalid`	Identifiziert den physikalischen Eingang (Input_I1..I8)
ImpulseDelta	DWORD	–	Anzahl der Impulse, nach denen eine Indikation (IND) ausgelöst wird
TimeDelta	DWORD	`250`	Zeit in ms, nach der eine periodische Indikation (IND) ausgelöst wird
TimeRateLimit	DWORD	`100`	Minimaler Abstand in ms zwischen zwei Indikationen (Rate Limiting)

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Event Output Qualifier – Status der letzten Verarbeitung
STATUS	STRING	Service Status – Fehler- oder Diagnosemeldung

Adapter

Name	Typ	Kommentar
IN	adapter::types::unidirectional::AD	Ausgangsdaten des FB – liefert die verarbeiteten Ereignisse und Werte an die Anwendung

Der Adapter `IN` wird mit den Ereignissen `IND` (Indikation) und `CNF` (Bestätigung) sowie dem Datenwert `D1` des internen FB `logiBUS_PI_ID` verbunden.

Funktionsweise

Der FB arbeitet als Composite, der die gesamte Initialisierungs- und Verarbeitungslogik an den internen FB `logiBUS_PI_ID` delegiert.

1. **Initialisierung (INIT):**
   Beim Eintreffen von `INIT` werden die Parameter `QI`, `PARAMS`, `Input`, `ImpulseDelta`, `TimeDelta` und `TimeRateLimit` an den internen FB weitergeleitet. Dieser konfiguriert den Hardwareeingang und startet die Überwachung. Nach erfolgreicher Initialisierung wird `INITO` ausgegeben.
2. **Verarbeitung (REQ):**
   Ein `REQ`-Ereignis löst eine Abfrage des Eingangswertes aus. Der interne FB prüft, ob die konfigurierten Schwellen (Impulszähler, Zeitschranke) erreicht oder überschritten wurden. Falls ja, werden die Ereignisse `IND` und/oder `CNF` über den Adapter `IN` an die nachgeschaltete Logik gesendet.
3. **Meldeverhalten:**
   - **Impulsabhängig:** Wenn `ImpulseDelta > 0`, wird nach jeder Änderung des Eingangssignals ein interner Impulszähler erhöht. Erreicht der Zähler den Wert von `ImpulseDelta`, wird eine Indikation (`IND`) ausgelöst und der Zähler zurückgesetzt.
   - **Zeitabhängig:** Zusätzlich oder alternativ wird nach Ablauf von `TimeDelta` Millisekunden eine periodische Indikation erzeugt.
   - **Sperrlogik:** Ist `ImpulseDelta = 0`, darf `TimeDelta` nicht 0 sein (Fehlervermeidung). Wird `TimeDelta = 0xFFFFFFFF` gesetzt, wird die zyklische Verarbeitung deaktiviert – es werden nur reine Wertänderungen gemeldet.
   - **Ratenbegrenzung:** `TimeRateLimit` verhindert zu häufige Indikationen; es wird sichergestellt, dass zwischen zwei aufeinanderfolgenden `IND`-Ereignissen mindestens die angegebene Zeit in ms liegt.

Technische Besonderheiten

- Der FB nutzt intern den Baustein `logiBUS_PI_ID`, der die eigentliche Hardwareanbindung und Zähllogik kapselt.
- Falls `ImpulseDelta = 0`, muss `TimeDelta > 0` gesetzt werden, andernfalls ist das Verhalten undefiniert (siehe Lizenzbemerkung).
- Ein `TimeDelta` von `0xFFFFFFFF` schaltet die zeitgesteuerte Indikation ab – nur impulsgesteuerte oder reine Wertänderungen werden weitergegeben.
- Der Adapter `IN` stellt die Ausgabedaten als unidirektionalen Datenstrom bereit, der in der übergeordneten Applikation weiterverarbeitet wird.
- Die Initialwerte (`Input`: `Invalid`, `TimeDelta`: `250`, `TimeRateLimit`: `100`) sind sinnvoll voreingestellt, um einen schnellen Betrieb ohne zusätzliche Konfiguration zu ermöglichen.

Zustandsübersicht

Da der FB ein Composite ist und die Zustandslogik vollständig im internen FB `logiBUS_PI_ID` gekapselt ist, wird hier eine allgemeine Zustandsbeschreibung gegeben:

- **IDLE** – Nach Reset oder fehlgeschlagener Initialisierung. Keine Verarbeitung.
- **INIT** – Initialisierung läuft (Warten auf Hardware-Rückmeldung).
- **RUN** – Normalbetrieb: Überwachung von Impulsen und Zeit, Auslösen von Indikationen.
- **ERROR** – Fehlerzustand (z. B. ungültige Parameter, Hardwarefehler). Wird über `STATUS` signalisiert.

Übergänge werden durch `INIT`, `REQ` sowie interne Fehler ausgelöst. Der interne FB verwendet vermutlich eine ECC (Execution Control Chart), die diese Zustände realisiert.

Anwendungsszenarien

- **Überwachung von digitalen Eingängen:** Anschluss von Tastern, Schaltern oder Endlagenschaltern, wobei sowohl schnelle Impulsänderungen als auch regelmäßige Statusmeldungen gefordert sind.
- **Impulszählung:** Z. B. Erfassung von Durchflusssensoren oder Drehgebern; der FB meldet nach jeder definierten Impulsanzahl einen Messwert.
- **Zeitgesteuerte Abfragen:** In Anwendungen, in denen der Eingangswert regelmäßig (alle `TimeDelta` ms) abgefragt werden muss, z. B. für Langzeitüberwachung oder sicherheitstechnische Prüfungen.
- **Ratenbegrenzung:** Verhindert eine Überlastung der Buskommunikation bei hochfrequenten Signaländerungen.

Vergleich mit ähnlichen Bausteinen

Baustein	Unterschied
`logiBUS_PI`	Einfacherer Eingangsbaustein ohne Impuls- und Zeit-Delta-Filter; nur rohe Wertänderungen.
`logiBUS_PI_ID`	Vorgänger ohne Composite-Struktur und ohne Adapter-Schnittstelle; direkte Ereignisausgänge.
`logiBUS_PI_IDA` (dieser FB)	Bietet zusätzlich einen **Adapter** für eine flexible Weiterverarbeitung und kombiniert Impuls-, Zeit- und Ratenlogik in einem Composite.

Der `logiBUS_PI_IDA` stellt eine erweiterte und modularere Variante dar, die sich besonders für komplexe Automatisierungsprojekte mit standardisierten Schnittstellen eignet.

Fazit

Der **logiBUS_PI_IDA** ist ein leistungsfähiger Composite Funktionsblock für die parametrierbare Erfassung digitaler Eingangssignale über den logiBUS. Durch die Kombination von Impuls- und Zeit-Delta sowie einer Ratenbegrenzung lässt er sich flexibel an unterschiedlichste Anforderungen anpassen. Die Kapselung der Logik im internen Baustein und die Adapter-Schnittstelle erleichtern die Wiederverwendung und Integration in übergeordnete Steuerungsprogramme. Der FB ist ideal für Anwendungen, die eine präzise und konfigurierbare Signalauswertung mit minimalem Ressourcenaufwand benötigen.

logiBUS_PI_S

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `logiBUS_PI_S` ist ein Struktur-Datentyp (Structured Type), der für die Konfiguration eines digitalen Eingangs (PI - wahrscheinlich „Peripheral Input“) innerhalb des logiBUS-Frameworks dient. Er definiert die notwendigen Parameter, um einen spezifischen physischen Pin eines I/O-Moduls oder Controllers anzusprechen. Dieser Datentyp wird typischerweise als Eingabeparameter für entsprechende IO-Funktionsblöcke verwendet.

Schnittstellenstruktur

Als Datentyp (STRUCT) besitzt `logiBUS_PI_S` keine ereignisgesteuerten Schnittstellen, sondern definiert ausschließlich Datenfelder.

Daten-Eingänge

Nicht zutreffend – dies ist ein Datentyp, kein ausführbarer Funktionsblock.

Daten-Ausgänge

Nicht zutreffend – dies ist ein Datentyp, kein ausführbarer Funktionsblock.

Interne Struktur (Variablen)

Die Struktur besteht aus einem einzigen Mitglied:

- **`Pin`** (USINT, Initialwert: 255): Diese Variable spezifiziert die Hardware-Pinnummer, die als digitaler Eingang genutzt werden soll. Der Datentyp `USINT` (Unsigned Short Integer) erlaubt Werte von 0 bis 255. Der voreingestellte Initialwert 255 dient häufig als inaktiver oder undefinierter Zustand.

Adapter

Nicht vorhanden.

Funktionsweise

Der Datentyp `logiBUS_PI_S` fungiert als reiner Konfigurations-Container. Er kapselt die Information, welcher physikalische Pin angesteuert werden soll. Die eigentliche Logik zur Abfrage des Pin-Zustands liegt in den Funktionsblöcken, die eine Variable dieses Typs als Eingang erwarten (z.B. `logiBUS_PI` oder ähnliche). Diese Blöcke lesen den Wert aus dem `Pin`-Feld aus und verwenden ihn, um die entsprechende Hardware-Schnittstelle anzusteuern.

Technische Besonderheiten

- **Initialwert:** Der Initialwert von 255 für `Pin` ist eine gängige Praxis, um einen nicht konfigurierten oder fehlerhaften Zustand zu kennzeichnen, da Pinnummern in der Praxis meist deutlich niedriger liegen.
- **Paketzuordnung:** Der Datentyp ist Teil des Pakets `logiBUS::io::PI`, was ihn der Kategorie der Peripherie-Eingänge (Input) innerhalb der logiBUS-Bibliothek zuordnet.
- **Typ-Hash:** Das Attribut `eclipse4diac::core::TypeHash` mit einem leeren Wert weist darauf hin, dass für die Typkompatibilität innerhalb des Systems ein automatisch generierter Hash verwendet wird.

Zustandsübersicht

Da es sich um einen passiven Datentyp handelt, gibt es keine Zustandsmaschine oder aktiven Zustände.

Anwendungsszenarien

Dieser Strukturdatentyp wird verwendet, um die Konfiguration von digitalen Eingangskanälen in Steuerungsapplikationen mit logiBUS-kompatibler Hardware zentral und typsicher zu halten. Typische Anwendungen sind:

- Konfiguration von Taster-, Schalter- oder Sensorsignalen (z.B. Endschalter, Lichtschranken).
- Parametrierung von Funktionsblöcken, die digitale Eingänge in einem verteilten I/O-System (wie logi.BUS) auslesen.
- Vereinfachung der Wiederverwendung und Lesbarkeit von Code, da die Pin-Konfiguration in einer benannten Struktur gebündelt ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Einfache Datentypen (z.B. USINT direkt):** Die Verwendung einer strukturierten `logiBUS_PI_S`-Variable ist einer direkten Verwendung einer `USINT`-Variable für die Pinnummer vorzuziehen, da sie

semantisch klarer ist und bei Erweiterungen der Konfiguration (z.B. um Filterzeit oder Invertierung) einfacher erweitert werden kann, ohne alle Aufrufstellen ändern zu müssen.
- **logiBUS_PO_S (Digitaler Ausgang):** Es existiert sehr wahrscheinlich ein komplementärer Datentyp `logiBUS_PO_S` für die Konfiguration digitaler Ausgänge (PO - Peripheral Output) mit identischer oder sehr ähnlicher Struktur.

Fazit

Der `logiBUS_PI_S`-Datentyp ist ein grundlegender, spezifischer Konfigurationsbaustein für die Anbindung digitaler Eingangssignale in logiBUS-basierten IEC 61499-Anwendungen. Durch seine einfache, auf einen Parameter reduzierte Struktur ermöglicht er eine klare und zentrale Hardware-Konfiguration und bildet die Basis für die entsprechende Funktionalität in den ausführenden IO-Funktionsblöcken.

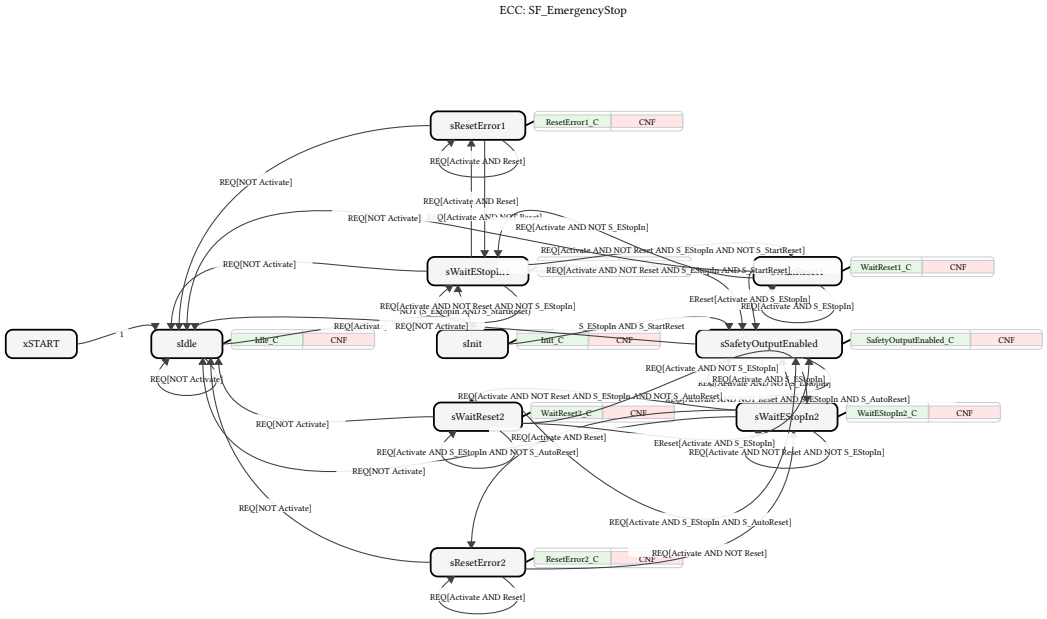
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.1.14.8 Safety

PLCopen

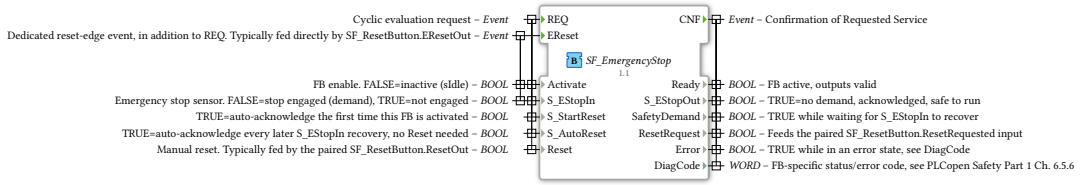
SF_EmergencyStop



• – ○ • – ○ • – ○ •

Einleitung

`SF_EmergencyStop` ist die PLCopen-Safety-Funktion `SF_EmergencyStop` (Part 1 v2.10, Kap. 6.5) zur **Überwachung eines Not-Halt-Gebers** (`S_EStopIn`). Der Ausgang `S_EStopOut` wird erst wieder `TRUE`, nachdem `S_EStopIn` wieder `TRUE` ist **und** eine Quittierung erfolgt ist (`S_StartReset`/`S_AutoReset`/`Reset`) – gemäß EN ISO 13849-1:2015 5.2.2 und IEC 60204-1:2016 9.2.3.4.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Zyklische Auswertungsanforderung. Verbundene Daten: `Activate`, `S_EStopIn`, `S_StartReset`, `S_AutoReset`, `Reset`
- **EReset**: Eigenes Reset-Flankenereignis zusätzlich zu `REQ`, typischerweise direkt von `SF_ResetButton.EResetOut` gespeist. Verbundene Daten: `Activate`, `S_EStopIn`

Ereignis-Ausgänge

- **CNF**: Bestätigung des angeforderten Dienstes. Verbundene Daten: `Ready`, `S_EStopOut`, `SafetyDemand`, `ResetRequest`, `Error`, `DiagCode`

Daten-Eingänge

- **Activate** (`BOOL`, Vorgabe `FALSE`): Baustein-Freigabe; `FALSE` = inaktiv (`sIdle`)
- **S_EStopIn** (`BOOL`, Vorgabe `FALSE`): Not-Halt-Geber; `FALSE` = Not-Halt betätigt (Anforderung), `TRUE` = nicht betätigt
- **S_StartReset** (`BOOL`, Vorgabe `FALSE`): `TRUE` = automatische Quittierung bei der ersten Aktivierung des Bausteins
- **S_AutoReset** (`BOOL`, Vorgabe `FALSE`): `TRUE` = automatische Quittierung jeder späteren Erholung von `S_EStopIn`, kein `Reset` nötig
- **Reset** (`BOOL`, Vorgabe `FALSE`): Manueller Reset, typischerweise vom zugehörigen `SF_ResetButton.ResetOut` gespeist

Daten-Ausgänge

- **Ready** (`BOOL`): Baustein aktiv, Ausgänge gültig
- **S_EStopOut** (`BOOL`): `TRUE` = keine Anforderung, quittiert, Anlauf erlaubt
- **SafetyDemand** (`BOOL`): `TRUE`, solange auf die Erholung von `S_EStopIn` gewartet wird
- **ResetRequest** (`BOOL`): Speist den zugehörigen Eingang `SF_ResetButton.ResetRequested`
- **Error** (`BOOL`): `TRUE` in einem Fehlerzustand, siehe `DiagCode`
- **DiagCode** (`WORD`, Vorgabe `16#0000`): Baustein-spezifischer Status-/Fehlercode (PLCopen Safety Part 1, Kap. 6.5.6)

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Baustein ist ein Basic-FB mit neun Zuständen. Die Ausgänge werden in jedem Zustand über einen eigenen Algorithmus gesetzt; danach wird `CNF` ausgelöst.

Zustand	DiagCode	S_EStopOut	SafetyDemand	ResetRequest	Error	Bedeutung
`sIdle`	`0000`	FALSE	FALSE	FALSE	FALSE	Baustein nicht aktiv
`sInit`	`8001`	FALSE	FALSE	FALSE	FALSE	Aktiviert, `S_StartReset` wird geprüft
`sWaitEStopIn1`	`8802`	FALSE	TRUE	FALSE	FALSE	Erste Aktivierung, Warten auf `S_EStopIn`
`sResetError1`	`C001`	FALSE	FALSE	FALSE	TRUE	`Reset` hängt auf `TRUE`, während auf `S_EStopIn` gewartet wird
`sWaitReset1`	`8402`	FALSE	FALSE	TRUE	FALSE	`S_EStopIn` ist `TRUE`, Warten

						auf steigende Flanke von Reset
sSafetyOutputEnabled	8000	TRUE	FALSE	FALSE	FALSE	Sicherheitsausgang freigegeben
sWaitEStopIn2	8804	FALSE	TRUE	NOT Reset	FALSE	Anforderung erkannt, Warten auf S_EStopIn
sResetError2	C011	FALSE	FALSE	FALSE	TRUE	Reset hängt auf TRUE, während auf S_EStopIn gewartet wird
sWaitReset2	8404	FALSE	FALSE	NOT Reset	FALSE	Warten auf S_AutoReset oder steigende Flanke von Reset

Ablauf:

1. REQ mit Activate = TRUE in sIdle führt nach sInit. Ist S_EStopIn AND S_StartReset, geht es direkt nach sSafetyOutputEnabled, sonst nach sWaitEStopIn1. S_StartReset zählt nur bei dieser allerersten Aktivierung.
2. Erste Aktivierung: In sWaitEStopIn1 führt Reset = TRUE in den Fehler sResetError1; ist S_EStopIn wieder TRUE, folgt je nach S_StartReset sSafetyOutputEnabled oder sWaitReset1. In sWaitReset1 führt das Ereignis EReset (bei S_EStopIn) nach sSafetyOutputEnabled; fällt S_EStopIn, geht es zurück nach sWaitEStopIn1.
3. Betrieb: In sSafetyOutputEnabled führt S_EStopIn = FALSE nach sWaitEStopIn2.
4. Folgezyklen: In sWaitEStopIn2 führt Reset = TRUE in den Fehler sResetError2; ist S_EStopIn wieder TRUE, folgt je nach S_AutoReset sSafetyOutputEnabled oder sWaitReset2. In sWaitReset2 führt S_AutoReset bzw. das Ereignis EReset nach sSafetyOutputEnabled; fällt S_EStopIn, geht es zurück nach sWaitEStopIn2.
5. Aus jedem Zustand führt REQ mit Activate = FALSE zurück nach sIdle.

Technische Besonderheiten

- Der Fehlerzustand sResetError1/sResetError2 verhindert, dass ein dauerhaft anstehendes Reset das Quittieren ersetzt (Reset muss eine Flanke sein).
- ResetRequest ist in den Folgezyklen (sWaitEStopIn2, sWaitReset2) mit NOT Reset verknüpft.
- Die Diagnosecodes stammen aus logiBUS::safety::PLCopen::const::DiagCodes und EmergencyStopDiagCodes.

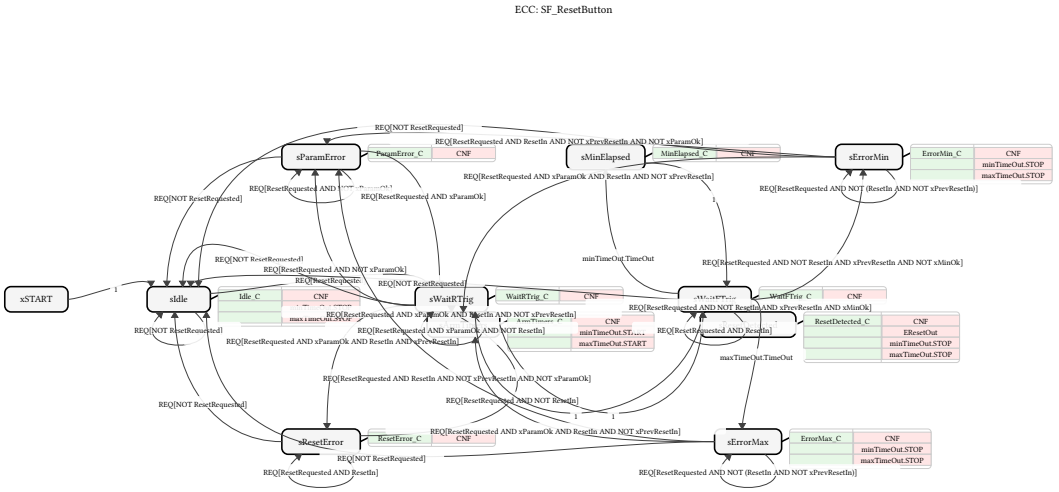
Anwendungsszenarien

- Überwachung eines Not-Halt-Tasters mit manueller Quittierung über SF_ResetButton
- Automatischer Wiederanlauf bei Bedarf über S_StartReset/S_AutoReset

Fazit

SF_EmergencyStop setzt die PLCopen-Safety-Zustandsmaschine für die Not-Halt-Überwachung um und gibt erst nach Erholung des Gebers und Quittierung den Sicherheitsausgang frei.

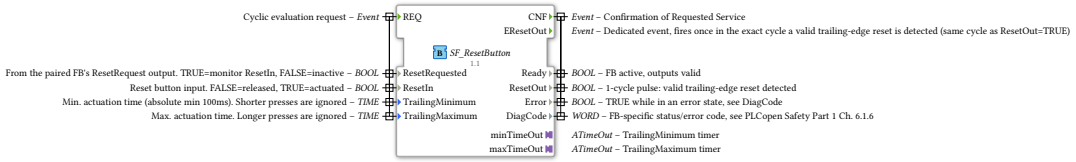
SF_ResetButton



• – ○ • – ○ • – ○ •

Einleitung

`SF_ResetButton` ist der PLCopen-Safety-Baustein `SF_ResetButton` (PLCopen TC5 Safety Part 1 v2.10, Kapitel 6.1) mit Trailing-Edge-Reset. Er überwacht einen Reset-Taster: Ein gültiger Reset liegt vor, wenn der Taster gedrückt (steigende Flanke) und nach einer Zeit zwischen `TrailingMinimum` und `TrailingMaximum` wieder losgelassen wird (fallende Flanke). Dann gibt `ResetOut` einen einzelnen Impuls aus.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: zyklische Auswertungsanforderung
  - **Verbundene Daten**: `ResetRequested`, `ResetIn`, `TrailingMinimum`, `TrailingMaximum`

Ereignis-Ausgänge

- **CNF**: Bestätigung des angeforderten Service
  - **Verbundene Daten**: `Ready`, `ResetOut`, `Error`, `DiagCode`
- **EResetOut**: eigenes Ereignis, das genau einmal im Zyklus feuert, in dem ein gültiger Trailing-Edge-Reset erkannt wird (derselbe Zyklus wie `ResetOut = TRUE`)

Daten-Eingänge

- **ResetRequested** (BOOL): vom gepaarten FB (`ResetRequest`-Ausgang). TRUE = `ResetIn` überwachen, FALSE = inaktiv (Default TRUE)
- **ResetIn** (BOOL): Reset-Taster-Eingang. FALSE = losgelassen, TRUE = betätigt (Default FALSE)
- **TrailingMinimum** (TIME): minimale Betätigungszeit (absolutes Minimum 100 ms). Kürzere Betätigungen werden nicht als Reset gewertet (Default `T#350ms`)
- **TrailingMaximum** (TIME): maximale Betätigungszeit. Längere Betätigungen werden nicht als Reset gewertet (Default `T#2s`)

Daten-Ausgänge

- **Ready** (BOOL): FB aktiv, Ausgänge gültig

- **ResetOut** (BOOL): Ein-Zyklus-Impuls, wenn ein gültiger Trailing-Edge-Reset erkannt wurde
- **Error** (BOOL): TRUE, solange ein Fehlerzustand besteht (siehe `DiagCode`)
- **DiagCode** (WORD): FB-spezifischer Status-/Fehlercode (PLCopen Safety Part 1, Kap. 6.1.6)

Adapter

- **minTimeOut** (Plug, `iec61499::events::ATimeOut`): Timer für `TrailingMinimum`
- **maxTimeOut** (Plug, `iec61499::events::ATimeOut`): Timer für `TrailingMaximum`

Funktionsweise

Der Baustein ist ein Basic-FB mit einem ECC. Interne Variablen: `xPrevResetIn` (Wert von `ResetIn` im vorigen Zyklus zur Flankenerkennung), `xMinOk` (TRUE, sobald seit der steigenden Flanke `TrailingMinimum` verstrichen ist) und `xParamOk` (`TrailingMinimum <= TrailingMaximum` und `TrailingMinimum >= T#100ms`).

- **Aktivierung**: Bei `ResetRequested = FALSE` bleibt der Baustein im Zustand `sIdle`. Bei `ResetRequested = TRUE` wechselt er nach `sWaitRTrig`.
- **`sWaitRTrig`**: Sind die Parameter ungültig, folgt `sParamError`. War `ResetIn` schon vor der Aktivierung gedrückt (`ResetIn AND xPrevResetIn`), folgt `sResetError`. Eine frische steigende Flanke (`ResetIn AND NOT xPrevResetIn`) führt nach `sArmTimers`.
- **`sArmTimers`**: setzt `minTimeOut.DT := TrailingMinimum` und `maxTimeOut.DT := TrailingMaximum`, löscht `xMinOk`, startet beide Timer (`minTimeOut.START`, `maxTimeOut.START`) und geht unbedingt nach `sWaitFTrig`.
- **`sWaitFTrig`**: Läuft `minTimeOut` ab, wird über `sMinElapsed` `xMinOk` gesetzt. Läuft `maxTimeOut` ab, folgt `sErrorMax`. Wird der Taster losgelassen (`NOT ResetIn AND xPrevResetIn`), folgt bei `xMinOk` der Zustand `sResetDetected`, sonst `sErrorMin`.
- **`sResetDetected`**: `ResetOut := TRUE`, Ausgabe von `EResetOut`, beide Timer werden gestoppt; danach unbedingt zurück nach `sWaitRTrig`.
- **Fehlerzustände** (`sParamError`, `sResetError`, `sErrorMin`, `sErrorMax`): `Error := TRUE` mit dem jeweiligen `DiagCode`. Aus `sErrorMin`/`sErrorMax` führt eine neue steigende Flanke wieder nach `sArmTimers`, aus `sResetError` das Loslassen des Tasters nach `sWaitRTrig`. In jedem Zustand führt `ResetRequested = FALSE` zurück nach `sIdle`.

Technische Besonderheiten

- `EResetOut` ist eine nicht wörtliche IEC-61499-Erweiterung der PLCopen-Schnittstelle: Ein eigenes Ereignis erlaubt es Verbrauchern, den Reset-Impuls direkt zu verdrahten, statt ihn aus `CNF` und `ResetOut` über ein externes Gatter abzuleiten (Version 1.1)
- Aufbau nach dem Muster `sequence_T_05` (ECC + `ATimeOut`-Plugs)
- `ResetOut` und `EResetOut` sind nur in genau einem Zyklus aktiv
- Die Diagnosecodes stammen aus den Konstanten `DiagCodes` und `ResetButtonDiagCodes` (`logiBUS::safety::PLCopen::const`)

Zustandsübersicht

Zustand	DiagCode	Bedeutung
xSTART	-	Anfangszustand, geht unbedingt nach `sIdle`
sIdle	0000	FB nicht aktiv
sParamError	C000	Parameterfehler: `TrailingMinimum > TrailingMaximum` oder `TrailingMinimum < 100 ms`
sResetError	C001	Reset-Fehler: `ResetIn` war bereits TRUE, als `ResetRequested` TRUE wurde
sWaitRTrig	83E2	aktiv, wartet auf Betätigung des Tasters (steigende Flanke)
sArmTimers	83F2	einmalig: beide Timer (neu) starten, danach `sWaitFTrig`
sWaitFTrig	83F2	Taster gedrückt, wartet auf Loslassen (fallende Flanke)
sMinElapsed	83F2	einmalig: `xMinOk` setzen, danach zurück nach `sWaitFTrig`

sErrorMin	C3F0	Fehler: Loslassen vor Ablauf von `TrailingMinimum`, wartet auf neue steigende Flanke
sErrorMax	C3E0	Fehler: `TrailingMaximum` vor dem Loslassen abgelaufen, wartet auf neue steigende Flanke
sResetDetected	8000	gültiger Reset, 1-Zyklus-Impuls auf `ResetOut` und `EResetOut`

Anwendungsszenarien

- Quittieren einer Sicherheitsfunktion (z. B. nach einem Not-Halt) mit bewusster, zeitlich begrenzter Tasterbetätigung
- Zusammenspiel mit einem Baustein, der über `ResetRequest` die Quittierung anfordert und mit `ResetRequested` ankoppelt

🛠️ Zugehörige Übungen

- Uebung_006_SAFE_AX

Fazit

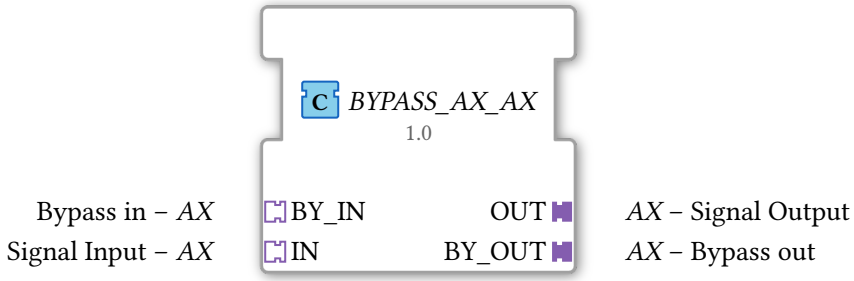
`SF_ResetButton` erkennt einen gültigen Trailing-Edge-Reset nach PLCopen Safety, überwacht Mindest- und Höchstdauer der Betätigung mit zwei Timern und meldet Fehler über `Error` und `DiagCode`.

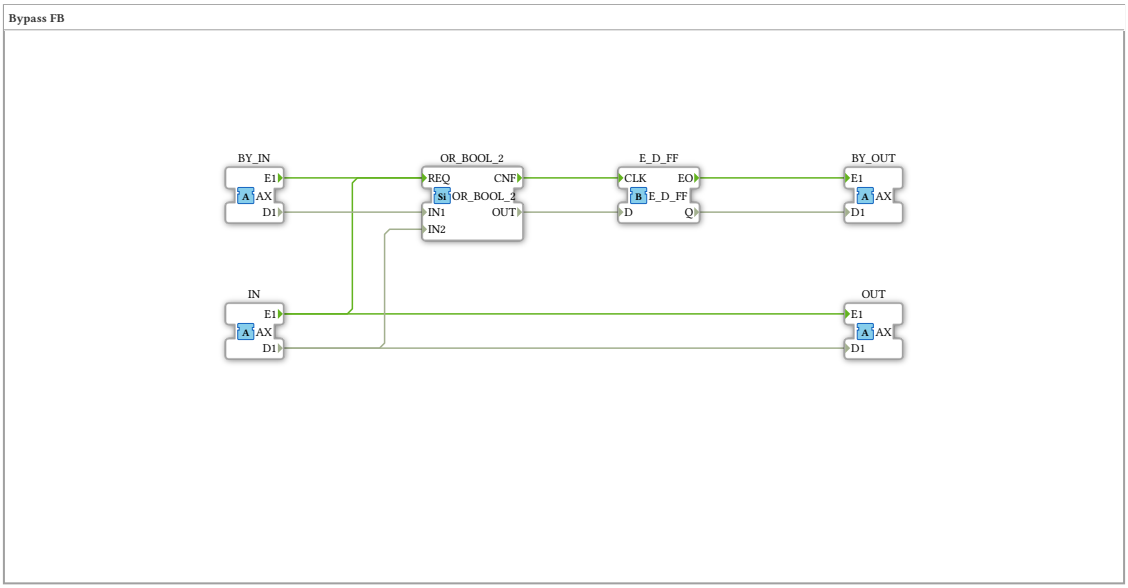
2.1.14.9 signalprocessing

Numerischer Index (signalprocessing)

Hier finden Sie alle Übungen des Arbeitsbereichs `signalprocessing` in aufsteigender Reihenfolge.

BYPASS_AX_AX





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **BYPASS_AX_AX** dient zur Weiterleitung und parallelen Bypass-Verarbeitung von AX-Signalen. Er kombiniert einen direkten Durchgang vom Eingang **IN** zum Ausgang **OUT** mit einer booleschen Oder-Verknüpfung der eingehenden Daten, die über ein flankengesteuertes D-Flipflop auf den Bypass-Ausgang **BY_OUT** gegeben wird.

Schnittstellenstruktur

Ereignis-Eingänge

- **IN.E1** – Ereignis des Signal-Eingangs
- **BY_IN.E1** – Ereignis des Bypass-Eingangs

Ereignis-Ausgänge

- **OUT.E1** – Ereignis des Signal-Ausgangs
- **BY_OUT.E1** – Ereignis des Bypass-Ausgangs

Daten-Eingänge

- **IN.D1** – Datenwert des Signal-Eingangs (Typ abhängig vom verwendeten AX-Adapter)
- **BY_IN.D1** – Datenwert des Bypass-Eingangs (gleicher Typ)

Daten-Ausgänge

- **OUT.D1** – Datenwert des Signal-Ausgangs (identisch mit IN.D1)
- **BY_OUT.D1** – Datenwert des Bypass-Ausgangs (Ergebnis der Oder-Verknüpfung, gespeichert im Flipflop)

Adapter

Der FB verwendet vier Adapter vom Typ `adapter::types::unidirectional::AX`:

- **IN** (Socket) – Signal-Eingang
- **BY_IN** (Socket) – Bypass-Eingang
- **OUT** (Plug) – Signal-Ausgang
- **BY_OUT** (Plug) – Bypass-Ausgang

Jeder AX-Adapter überträgt ein Ereignis-/Daten-Paar (E1/D1) in einer Richtung.

Funktionsweise

Der FB realisiert zwei parallele Pfade:

1. **Direkter Pfad:**
   Ein Ereignis auf **IN.E1** wird direkt an **OUT.E1** weitergegeben, ebenso der zugehörige Datenwert **IN.D1** an **OUT.D1**. Dies stellt eine 1:1-Durchschaltung des Signals dar.
2. **Bypass-Pfad:**
   Sowohl ein Ereignis auf **IN.E1** als auch auf **BY_IN.E1** lösen die Verarbeitung im Baustein `OR_2_BOOL` aus. Dieser verknüpft die Datenwerte **IN.D1** und **BY_IN.D1** boolesch – es handelt sich um eine logische Oder-Verknüpfung der beiden binären Eingänge.
   Das Ergebnis der Verknüpfung wird an den Dateneingang **D** des D-Flipflops (`E_D_FF`) gelegt. Der Takt des Flipflops wird durch das Rücksignal `OR_2_BOOL.CNF` ausgelöst, also nach Abschluss der Oder-Berechnung.
   Der Ausgang **Q** des Flipflops wird auf **BY_OUT.D1** gelegt, und das zugehörige Ereignis **BY_OUT.E1** wird durch den Flipflop-Ausgang **EO** angesteuert.

Technische Besonderheiten

- **Boolesche Oder-Verknüpfung:** Die Daten werden nicht etwa addiert oder gemischt, sondern logisch OR-verknüpft. Daher sind die Datenwerte als boolesch (TRUE/FALSE) anzusehen.
- **Flankengesteuertes Flipflop:** Der Bypass-Wert wird immer dann aktualisiert, wenn mindestens einer der beiden Eingänge (IN oder BY_IN) ein Ereignis liefert. Das Flipflop speichert den letzten berechneten OR-Wert.
- **Unabhängige Ereignisweitergabe:** Der direkte Pfad ist asynchron – jedes Ereignis auf IN wird sofort an OUT weitergegeben, ohne auf den Bypass-Pfad zu warten.

Zustandsübersicht

Der FB selbst besitzt keinen eigenen Zustandsautomaten, jedoch wird durch das interne D-Flipflop ein binärer Zustand realisiert:

- **Zustand 0:** Flipflop-Ausgang Q = FALSE ⟶ BY_OUT.D1 = FALSE
- **Zustand 1:** Flipflop-Ausgang Q = TRUE ⟶ BY_OUT.D1 = TRUE

Der Zustand wechselt bei jeder positiven Flanke des Taktes (`OR_2_BOOL.CNF`) auf den aktuellen D-Wert. Der Anfangszustand ist nicht definiert und muss ggf. extern initialisiert werden.

Anwendungsszenarien

- **Bypass-Schaltung in Steuerungen:** Ein Normalsignal (IN ⟶ OUT) soll unverändert durchgeschleift werden, gleichzeitig wird eine Oder-Verknüpfung von Normalsignal und einem zusätzlichen Bypass-Signal für Überwachungszwecke erzeugt.
- **Not-Aktivierung:** Der Bypass-Eingang kann als manuelle Übersteuerung (z. B. „Handbetrieb“) verwendet werden, während der automatische Betrieb am Hauptausgang weiterläuft.
- **Logische Signalverfolgung:** Durch die getrennten Ausgänge können sowohl das Originalsignal als auch die verknüpfte Logik parallel ausgewertet werden.

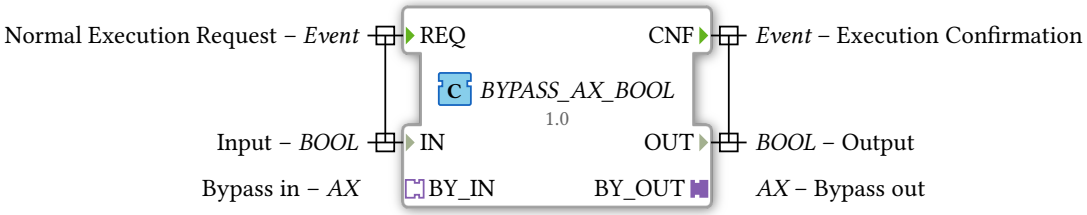
Vergleich mit ähnlichen Bausteinen

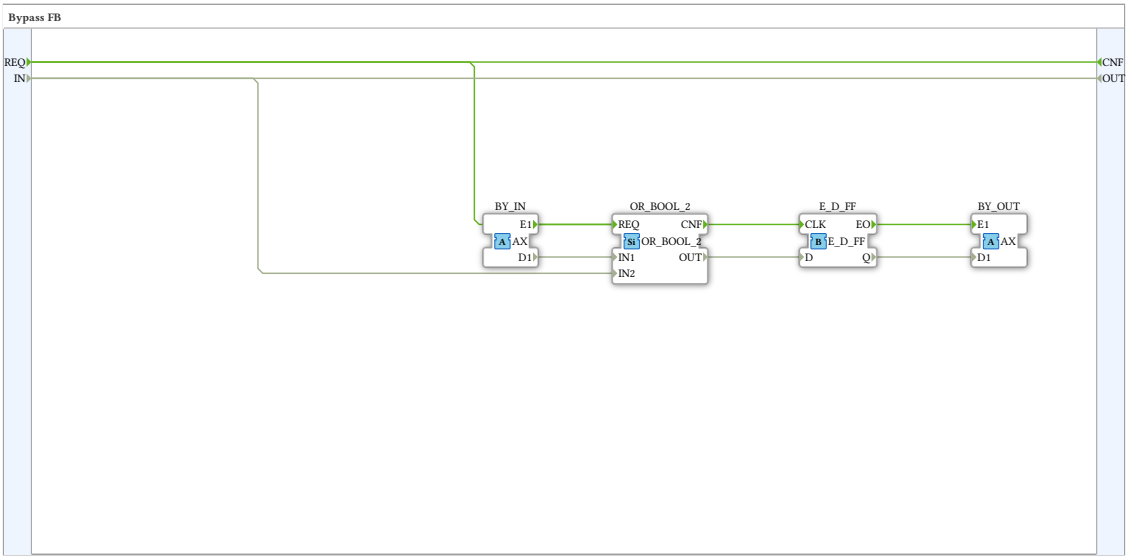
- **Einfacher Durchgangs-FB (z. B. MOVE_AX):** Leitet das Signal 1:1 weiter, erzeugt aber keinen Bypass-Ausgang. BYPASS_AX_AX bietet zusätzlich die Verknüpfungslogik und einen zweiten Ausgang.
- **OR-Baustein ohne Flipflop:** Ein reines Oder würde den Bypass-Ausgang kontinuierlich aktualisieren, ohne Zwischenspeicherung. Der integrierte Flipflop sorgt für eine flankengesteuerte Aktualisierung, die bei zyklischen Steuerungen oder zeitlich entkoppelten Ereignissen von Vorteil ist.
- **Adapter-basierte Lösung mit zwei separaten FBs:** Man könnte einen OR-FB und einen Flipflop-FB separat verdrahten – BYPASS_AX_AX kapselt diese Funktionalität in einem kompakten Baustein, reduziert die Anzahl der Verbindungen und vereinfacht die Handhabung.

Fazit

Der **BYPASS_AX_AX** ist ein nützlicher Funktionsblock für Anwendungen, die sowohl eine direkte Signalweiterleitung als auch eine logische Oder-Verknüpfung mit gespeichertem Ausgang erfordern. Durch die Kombination von Durchgangs- und Bypass-Pfad in einem Baustein wird die Netzwerkstruktur übersichtlich gehalten. Die Verwendung von unidirektionalen AX-Adaptern macht ihn kompatibel mit typischen Industriekommunikationsprotokollen auf Basis des IEC 61499-Standards.

BYPASS_AX_BOOL





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `BYPASS_AX_BOOL` realisiert eine Bypass-Funktionalität für ein boolesches Signal. Er leitet den Wert des Eingangs `IN` direkt zum Ausgang `OUT` weiter und erzeugt parallel dazu einen über einen Adapter angebundenen Bypass-Pfad. Dieser Pfad erlaubt das Einblenden eines externen Signals (`BY_IN.D1`) in die interne Logik und gibt das Ergebnis über den Ausgangsadapter `BY_OUT` aus.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung
REQ	Event	Normaler Ausführungsanstoß

Ereignis-Ausgänge

Name	Typ	Beschreibung
CNF	Event	Bestätigung der Ausführung

Daten-Eingänge

Name	Typ	Beschreibung
IN	BOOL	Eingangssignal

Daten-Ausgänge

Name	Typ	Beschreibung
OUT	BOOL	Durchgeschleiftes Eingangssignal (identisch zu IN)

Adapter

Rolle	Name	Typ	Richtung	Beschreibung
Plug	BY_OUT	`adapter::types::unidirectional::AX`	Ausgang	Gibt den gespeicherten Bypass-Zustand aus

Socket	BY_IN	adapter::types::unidirectional::AX	Eingang	Nimmt das Bypass-Eingangssignal (D1) und Ereignis (E1) entgegen

Funktionsweise

Bei jedem Ereignis `REQ` wird unverzüglich der Ausgang `CNF` gesetzt (Ereignisdurchschleifung). Gleichzeitig wird das Datenwort `IN` direkt auf `OUT` kopiert.

Intern wird eine ODER-Verknüpfung zwischen `IN` und dem Adaptersignal `BY_IN.D1` gebildet. Diese Verknüpfung wird durch `REQ` oder durch das Ereignis `BY_IN.E1` angestoßen. Das Ergebnis der ODER-Operation wird in einem flankengetakteten D-Flipflop gespeichert; die Taktflanke wird durch die Beendigung der ODER-Berechnung (Ereignis `CNF` des internen OR-Bausteins) erzeugt. Der gespeicherte Wert des Flipflops wird über den Ausgangsadapter `BY_OUT.D1` ausgegeben und der zugehörige Ereignisausgang `BY_OUT.E1` wird aktiviert.

Der Baustein arbeit somit als Kombination aus einem reinen Durchschleifer für `IN → OUT` und einem flankengesteuerten Bypass-Pfad, der das eingehende Adaptersignal mit dem aktuellen `IN` verknüpft und speichert.

Technische Besonderheiten

- Die direkte Verbindung `IN → OUT` erfolgt ohne Verzögerung durch interne Logik – der Wert liegt sofort an.
- Der Bypass-Pfad nutzt ein D-Flipflop (`E_D_FF`) zur Zustandsspeicherung. Der gespeicherte Wert wird nur bei einer steigenden Flanke des internen Takts aktualisiert, der durch die ODER-Berechnung erzeugt wird.
- Der Adapter `BY_IN` muss angeschlossen sein, um den Bypass-Pfad zu nutzen; ohne Anschluss verhält sich der Baustein wie ein einfacher Durchschleifer.
- Das Ereignis `BY_IN.E1` kann die ODER-Berechnung unabhängig von `REQ` auslösen, sodass der Bypass auch asynchron aktualisiert werden kann.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten, aber der interne D-Flipflop speichert einen booleschen Zustand (`Q = 0` oder `Q = 1`). Dieser Zustand repräsentiert den zuletzt gültigen Bypass-Wert, der über den Ausgangsadapter ausgegeben wird. Der Zustand ändert sich nur bei einer steigenden Flanke des Taktsignals (Beendigung einer ODER-Berechnung).

Anwendungsszenarien

- **Manuelles Übersteuern** eines digitalen Signals in der Automatisierungstechnik: Der normale Wert (`IN`) wird an die Steuerung weitergegeben, während über den Bypass ein Handeingriff (z. B. über eine Bedienschnittstelle) zugemischt und gespeichert werden kann.
- **Signalüberwachung und -korrektur**: Ein externes Überwachungssystem kann über den Bypass-Adapter einen Korrekturwert einblenden, der mit dem Originalsignal verknüpft wird.
- **Bypass für Testzwecke**: Während der Inbetriebnahme kann der normale Pfad (IN⟶OUT) aufrechterhalten, gleichzeitig aber ein Testsignal über den Bypass eingeschleift werden.

Vergleich mit ähnlichen Bausteinen

- **Einfacher Durchschleifer** (z. B. `MOVE`): Würde nur `IN` auf `OUT` übertragen, ohne Möglichkeit eines externen Eingriffs.
- **Set-/Reset-Flipflop** (z. B. `SR`): Erlaubt das Setzen und Rücksetzen eines Ausgangs, jedoch ohne direkte Durchschleifung des Eingangs.
- **Auswahlbaustein** (z. B. `SEL`): Wählt zwischen zwei Eingängen – hier wird dagegen der Eingang immer durchgereicht und der Bypass additiv verknüpft.

Der `BYPASS_AX_BOOL` kombiniert die Durchschleifung mit einer integrierbaren, flankengesteuerten Logik für externe Signale.

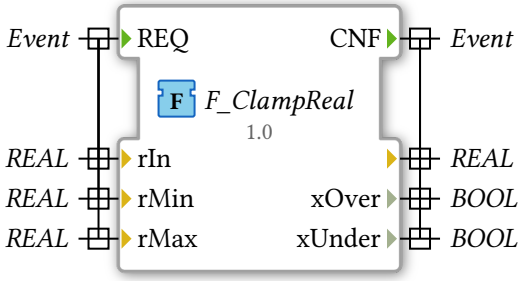
Fazit

Der Funktionsblock `BYPASS_AX_BOOL` ist ein spezialisierter Baustein für Anwendungen, bei denen ein boolesches Signal unverändert weitergeleitet, aber gleichzeitig ein externes Bypass-Signal zugemischt und gespeichert werden soll. Die Kombination aus direkter Durchschleifung und flankengesteuerter ODER-Verknüpfung mit Speicherung ermöglicht flexible Eingriffe in Signalketten, ohne die normale

Funktion zu unterbrechen. Der Baustein eignet sich besonders für Steuerungssysteme, die sowohl einen Standardbetrieb als auch temporäre oder dauerhafte Überlagerungen benötigen.

Clamp

F_ClampReal



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **F_ClampReal** begrenzt einen eingehenden reellen Wert (`rIn`) auf einen zulässigen Bereich zwischen einem unteren Grenzwert (`rMin`) und einem oberen Grenzwert (`rMax`). Zusätzlich werden zwei boolesche Ausgangssignale ausgegeben, die anzeigen, ob der Eingangswert über- oder unterhalb des Bereichs lag. Der Baustein wird typischerweise in Signalverarbeitungsketten oder Regelungslogik eingesetzt, um unzulässige Werte zu „clampen“ und gleichzeitig eine Über- oder Unterschreitung zu signalisieren.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** – Anforderungsereignis: Durch einen positiven Trigger an diesem Eingang wird die Verarbeitung gestartet. Die an den Dateneingängen anliegenden Werte (`rIn`, `rMin`, `rMax`) werden übernommen und ausgewertet.

Ereignis-Ausgänge

- **CNF** – Bestätigungsereignis: Nach Abschluss der Berechnung wird dieses Ereignis ausgelöst. Es signalisiert, dass die neuen Werte an den Datenausgängen gültig sind.

Daten-Eingänge

- **rIn** (REAL): Der zu begrenzende Eingangswert.
- **rMin** (REAL): Untere Grenze des zulässigen Bereichs.
- **rMax** (REAL): Obere Grenze des zulässigen Bereichs.

Daten-Ausgänge

- **rOut** (REAL, symbolisch – in der XML-Schnittstelle nicht benannt, aber als Ergebnis der Funktion verfügbar): Der auf den Bereich [rMin, rMax] begrenzte Wert.
- **xOver** (BOOL): Wird `TRUE`, wenn `rIn > rMax` – also der Eingangswert oberhalb des Maximalwerts liegt.
- **xUnder** (BOOL): Wird `TRUE`, wenn `rIn < rMin` – also der Eingangswert unterhalb des Minimalwerts liegt.

Adapter

Es sind keine Adapter definiert. Der Baustein agiert vollständig über diskrete Ein-/Ausgänge.

Funktionsweise

Beim Eintreffen des Ereignisses **REQ** wird der aktuelle Wert von `rIn` mit den Grenzen `rMin` und `rMax` verglichen:

1. **Fall: `rIn > rMax`**
   Der Ausgang `rOut` wird auf `rMax` gesetzt. Gleichzeitig wird `xOver` auf `TRUE` und `xUnder` auf `FALSE` gesetzt.

2. **Fall: `rIn < rMin`**
   Der Ausgang `rOut` wird auf `rMin` gesetzt. `xOver` wird `FALSE`, `xUnder` wird `TRUE`.
3. **Fall: `rMin ≤ rIn ≤ rMax`**
   Der Eingangswert liegt innerhalb des Bereichs. `rOut` wird unverändert auf `rIn` gesetzt. Beide Flags (`xOver`, `xUnder`) werden auf `FALSE` gesetzt.

Nach Abschluss der Berechnung wird das Ereignis **CNF** ausgelöst und die neuen Ausgangswerte sind gültig. Der Baustein arbeitet deterministisch und ohne internen Zustand – er führt pro Aufruf genau eine Berechnung aus.

Technische Besonderheiten

- Der Baustein ist als **Funktion** (FUNCTION) implementiert, nicht als ansonsten übliche Zustandsmaschine. Dadurch ist er **kombinatorisch** und besitzt keine internen Speichervariablen.
- Die Eingangsgrößen werden **mit dem Ereignis REQ** über die `With`-Zuweisung eingelesen; die Ausgangsgrößen werden beim Absenden von **CNF** mitgeliefert.
- Es findet **keine Echtzeitoptimierung** oder Sonderbehandlung für NaN/INF statt – dies ist eine reine Gleitkomma-Vergleichslogik.
- Die Flags `xOver` und `xUnder` sind sich gegenseitig ausschließend; es wird immer genau eines der beiden auf `TRUE` gesetzt, wenn der Grenzfall vorliegt.

Zustandsübersicht

Da keine Zustandsautomatik existiert, lässt sich die gesamte Funktionalität durch drei deterministische Bedingungen beschreiben:

Bedingung	rOut	xOver	xUnder
`rIn > rMax`	`rMax`	TRUE	FALSE
`rIn < rMin`	`rMin`	FALSE	TRUE
`rMin ≤ rIn ≤ rMax`	`rIn`	FALSE	FALSE

Es gibt keinen zusätzlichen Ruhezustand; der Baustein ist nur zwischen zwei Ereignissen aktiv.

Anwendungsszenarien

- **Schutz von Aktoren**: Begrenzen eines Stellsignals auf den zulässigen Wertebereich, um mechanische oder elektrische Überlastung zu vermeiden.
- **Sensorwertaufbereitung**: Clampen von Messwerten auf einen physikalisch sinnvollen Bereich, bevor sie an Regler oder Anzeigen weitergegeben werden.
- **Plausibilitätsprüfung**: Durch die Flags `xOver` und `xUnder` kann eine übergeordnete Steuerung erkennen, wenn ein Wert außerhalb des erwarteten Bereichs liegt.
- **Regelungstechnik**: Sicherstellen, dass das Ausgangssignal eines PID-Reglers nie die Endstufen-Grenzwerte überschreitet.

Vergleich mit ähnlichen Bausteinen

In der IEC 61131-3 gibt es die Standard-Funktionen `LIMIT` (oder `CLAMP`), die eine ähnliche Begrenzung durchführen, jedoch meist ohne zusätzliche Flag-Ausgänge. Der hier vorgestellte Baustein erweitert diese Grundfunktion um die Indikatoren `xOver` und `xUnder`. Gegenüber einer manuellen `IF-ELSE`-Programmierung bietet er eine kompakte, wiederverwendbare Lösung mit klar definierter Schnittstelle. Andere Ansätze, wie z. B. Sonderbehandlungen für ungültige Werte (NaN, INF), sind in diesem Baustein nicht implementiert, was in vielen embedded-Anwendungen ausreichend ist.

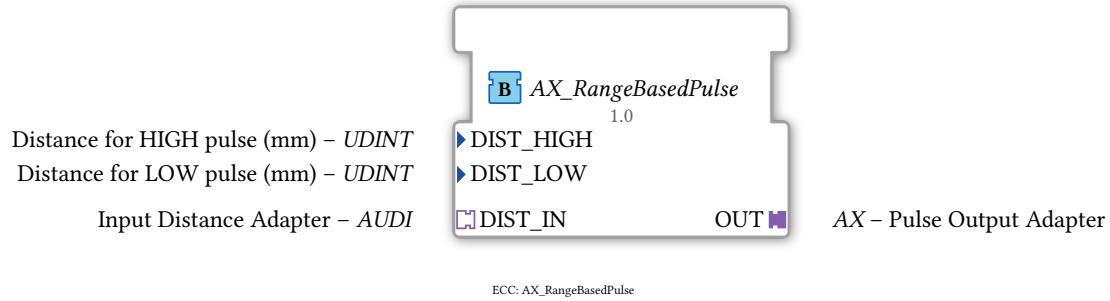
Fazit

F_ClampReal ist ein einfacher, aber nützlicher Funktionsblock zur Wertebegrenzung auf eine obere und untere Grenze. Die zusätzlichen Flags ermöglichen eine einfache Überwachung, ob ein Grenzwert verletzt wurde. Aufgrund seiner kombinatorischen Natur und der schlanken Schnittstelle eignet er sich hervorragend für den Einsatz in speicher- und rechenzeitkritischen Umgebungen, wie sie in der Automatisierungstechnik häufig vorkommen.

Numerischer Index (signalprocessing)

Hier finden Sie alle Übungen des Arbeitsbereichs `signalprocessing` in aufsteigender Reihenfolge.

AX_RangeBasedPulse



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock generiert einen Puls auf einem AX-Adapter basierend auf einer zurückgelegten Distanz (Adapter-Version). Er ist unter der Eclipse Public License 2.0 lizenziert und für den Einsatz in der Signalverarbeitung von Distanzdaten konzipiert.

Schnittstellenstruktur

Ereignis-Eingänge

- `DIST_IN.E1`: Ereignis vom Distanz-Adapter, das die Aktualisierung des Distanzwerts signalisiert.

Ereignis-Ausgänge

- `OUT.E1`: Ereignis auf dem Ausgangs-Adapter, das bei einer Zustandsänderung des Pulsausgangs gesendet wird.

Daten-Eingänge

- `DIST_HIGH` (UDINT): Distanz in mm, die zurückgelegt werden muss, um den HIGH-Puls zu erzeugen (Initialwert: 5000).
- `DIST_LOW` (UDINT): Distanz in mm, die zurückgelegt werden muss, um den LOW-Puls zu erzeugen (Initialwert: 5000).
- `DIST_IN.D1` (UDINT): Aktuelle Distanz vom Eingangs-Adapter (über das Ereignis aktualisiert).

Daten-Ausgänge

- `OUT.D1` (BOOL): Pulsausgang – wechselt zwischen TRUE (HIGH) und FALSE (LOW) abhängig von der akkumulierten Distanz.

Adapter

- **DIST_IN (Socket):** Adapter vom Typ `adapter::types::unidirectional::AUDI`. Stellt einen unidirektionalen Distanzwert (`D1`) sowie ein zugehöriges Ereignis (`E1`) bereit.
- **OUT (Plug):** Adapter vom Typ `adapter::types::unidirectional::AX`. Stellt einen unidirektionalen Bool-Wert (`D1`) und ein Ereignis (`E1`) zur Verfügung.

Funktionsweise

Der Funktionsblock arbeitet nach einem Hystereseprinzip mit zwei Schwellwerten:

1. **Initialisierung:** Beim ersten Durchlauf (`FirstRun = TRUE`) wird die aktuelle Distanz als Startwert `LastDist` gespeichert, der Ausgang `OUT.D1` auf FALSE gesetzt und `FirstRun` zurückgesetzt.
2. **Zustandsprüfung:** Bei jedem eingehenden Ereignis über `DIST_IN.E1` führt der Block den `CHECK`-Algorithmus aus:
   - Wenn der aktuelle Ausgang LOW (FALSE) ist und die neue Distanz `DIST_IN.D1` den Wert `LastDist + DIST_LOW` erreicht oder überschreitet, wird `OUT.D1` auf TRUE gesetzt und `LastDist` auf den aktuellen Distanzwert aktualisiert.
   - Wenn der Ausgang HIGH (TRUE) ist und die neue Distanz den Wert `LastDist + DIST_HIGH` erreicht oder überschreitet, wird `OUT.D1` auf FALSE gesetzt und `LastDist` aktualisiert.
3. **Ausgabe:** Bei einer Zustandsänderung von `OUT.D1` wird das Ereignis `OUT.E1` gesendet und der neue Zustand in `LastState` für die nächste Prüfung gespeichert.

Technische Besonderheiten

- **Adapter-basierte Kommunikation:** Die Verwendung von Plugs und Sockets (`AX`, `AUDI`) ermöglicht eine modulare und flexible Verschaltung in der 4diac-IDE.
- **Asymmetrische Pulsbreiten:** Die getrennten Parameter `DIST_HIGH` und `DIST_LOW` erlauben unterschiedliche Distanzen für HIGH- und LOW-Phasen.
- **Initialisierungslogik:** Ein interner `FirstRun`-Flag sorgt dafür, dass die erste Messung korrekt als Startpunkt gesetzt wird.
- **Paketkontext:** Der FB ist für das Paket `logiBUS::signalprocessing::distance` vorgesehen.

Zustandsübersicht

Das ECC (Execution Control Chart) besteht aus vier Zuständen:

- **START:** Wartezustand nach Initialisierung oder nach einer Verarbeitung.
- **INIT:** Führt den `INIT`-Algorithmus aus (Setzen der Startwerte). Wird nur beim ersten Durchlauf (`FirstRun = TRUE`) betreten.
- **CHECK:** Führt den `CHECK`-Algorithmus zur Distanzprüfung aus und ändert ggf. den Ausgang.
- **EMIT:** Führt den `SAVE`-Algorithmus aus (speichert den aktuellen Ausgangszustand) und sendet das Ereignis `OUT.E1`.

Übergänge:

- `START → INIT`: Wenn `DIST_IN.E1` eintrifft und `FirstRun = TRUE`.
- `START → CHECK`: Wenn `DIST_IN.E1` eintrifft und `FirstRun = FALSE`.
- `INIT → CHECK`: Sofort nach `INIT` (Bedingung 1).
- `CHECK → EMIT`: Wenn sich `OUT.D1` gegenüber `LastState` geändert hat.
- `CHECK → START`: Wenn keine Änderung eintritt (`OUT.D1 = LastState`).
- `EMIT → START`: Sofort nach `EMIT` (Bedingung 1).

Anwendungsszenarien

- **Wegimpulsgeber in der Landtechnik:** Erzeugen von Impulsen nach jeder definierten Wegstrecke (z. B. alle 5 m ein HIGH-Impuls, dann nach weiteren 5 m ein LOW-Impuls) für Steuerungs- oder Zählaufgaben.
- **Distanzbasierte Schaltfunktionen:** Auslösen von Aktionen in Fördersystemen, fahrerlosen Transportsystemen oder automatischen Maschinen nach bestimmten zurückgelegten Strecken.
- **Pulsweitenmodulation über Strecke:** Durch unterschiedliche Werte für `DIST_HIGH` und `DIST_LOW` lassen sich variable Pulsbreiten realisieren, z. B. 10 m HIGH und 2 m LOW.

Vergleich mit ähnlichen Bausteinen

- **Tonbasierte Pulsgeneratoren (z. B. `E_CYCLE`):** Erzeugen Pulse zeitabhängig, während `AX_RangeBasedPulse` wegabhängig arbeitet – für Anwendungen ohne feste Zeitbasis.
- **Einfache Flankendetektoren:** Erkennen Signalwechsel, jedoch ohne Hysterese und ohne Akkumulation einer Distanz.
- **Andere Distanz-Adapter-Bausteine:** Es existieren Bausteine, die Distanzwerte aggregieren, aber dieser FB kombiniert Ereignisverarbeitung mit Schwellwertlogik und asymmetrischer Konfiguration.

Fazit

Der `AX_RangeBasedPulse`-Funktionsblock bietet eine robuste und flexible Lösung zur distanzbasierten Pulsgenerierung. Durch die Adapter-Schnittstelle, die getrennt konfigurierbaren Distanzschwellen und die Hystereselogik eignet er sich hervorragend für den modularen Einsatz in industriellen Steuerungssystemen, insbesondere in der Landtechnik und Automatisierung. Die Implementierung als BasicFB gewährleistet dabei ein deterministisches und nachvollziehbares Verhalten.

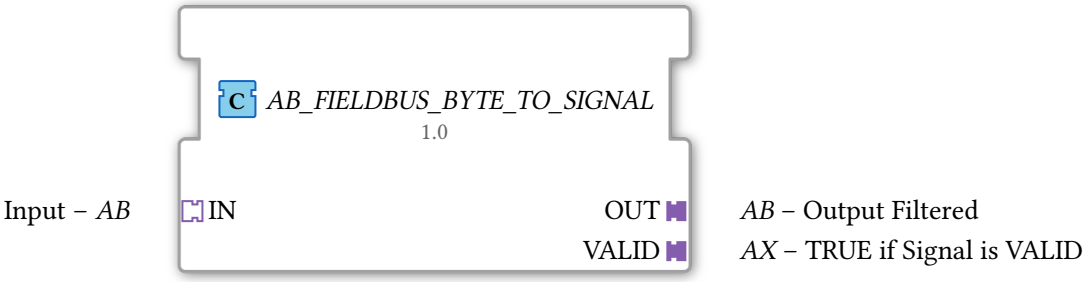
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

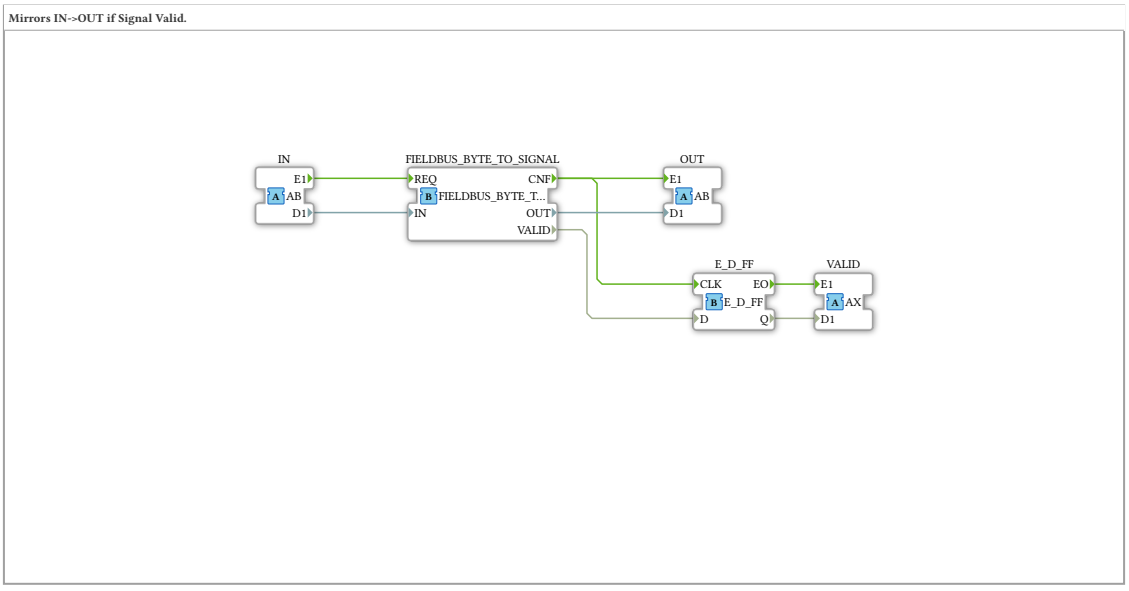
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 Das PWM-Signal & Infografik auf ms-muc-docs.de

Numerischer Index (signalprocessing)

Hier finden Sie alle Übungen des Arbeitsbereichs `signalprocessing` in aufsteigender Reihenfolge.

AB_FIELDBUS_BYTE_TO_SIGNAL





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AB_FIELDBUS_BYTE_TO_SIGNAL** dient dazu, ein eingehendes Byte-Signal (über den Adapter *IN*) auf den Ausgang (*OUT*) zu spiegeln, sofern das Signal als gültig erkannt wird. Die Gültigkeit wird über einen separaten Ausgang (*VALID*) signalisiert. Der Baustein kapselt die Verarbeitung eines Feldbus-Byte-Signals und stellt sicher, dass nur valide Daten an die nachfolgende Logik weitergegeben werden. Er basiert auf einem internen `FIELDBUS_BYTE_TO_SIGNAL`-Baustein, ergänzt durch ein D-Flipflop zur stabilen Ausgabe des Gültigkeitssignals.

Schnittstellenstruktur

Der FB besitzt **keine** direkten Ereignis- oder Daten-Ein-/Ausgänge auf der obersten Ebene. Die gesamte Kommunikation erfolgt über drei **Adapter-Schnittstellen**:

Adapter	Richtung	Typ	Beschreibung
`IN`	Socket	`adapter::types::unidirectional::AB`	Eingangsadapter für das Byte-Signal und das zugehörige Ereignis.
`OUT`	Plug	`adapter::types::unidirectional::AB`	Ausgangsadapter für das gespiegelte Byte-Signal.
`VALID`	Plug	`adapter::types::unidirectional::AX`	Ausgangsadapter, der den Gültigkeitszustand des Signals anzeigt.

Die Adapter sind vom Typ **unidirectional**, d.h. sie übertragen Daten und Ereignisse in eine Richtung. Die Typen `AB` und `AX` enthalten jeweils einen Ereignis-Eingang/-Ausgang (E1) und einen Daten-Eingang/-Ausgang (D1, vom Typ `ANY` bzw. `BOOL`).

Ereignis-Eingänge

Keine (die Ereignisse werden über den Socket-Adapter `IN` empfangen).

Ereignis-Ausgänge

Keine (die Ereignisse werden über die Plug-Adapter `OUT` und `VALID` gesendet).

Daten-Eingänge

Keine (die Daten werden über den Socket-Adapter `IN` empfangen).

Daten-Ausgänge

Keine (die Daten werden über die Plug-Adapter `OUT` und `VALID` gesendet).

Adapter

IN (Socket)

- **E1**: Ereigniseingang – löst die Verarbeitung eines neuen Byte-Wertes aus.
- **D1**: Dateneingang – das zu verarbeitende Byte (z.B. ein Feldbus-Datagramm).

OUT (Plug)

- **E1**: Ereignisausgang – wird nach erfolgreicher Spiegelung eines gültigen Signals aktiviert.
- **D1**: Datenausgang – das gespiegelte Byte-Signal (nur bei gültigem Wert).

VALID (Plug)

- **E1**: Ereignisausgang – wird bei jedem Verarbeitungsdurchlauf aktiviert, unabhängig von der Gültigkeit.
- **D1**: Datenausgang (BOOL) – `TRUE`, wenn das aktuell verarbeitete Signal gültig ist; sonst `FALSE`.

Funktionsweise

1. Ein eingehendes Ereignis am Adapter `IN.E1` triggert den internen Baustein `FIELDBUS_BYTE_TO_SIGNAL`, der das Byte an `IN.D1` verarbeitet.
2. Der interne Baustein gibt an seinen Ausgängen `OUT` (das gespiegelte Byte) und `VALID` (die Gültigkeitsinformation) weiter.
3. Das Ausgangssignal wird sofort an den Adapter `OUT` weitergeleitet und ein Ereignis (`OUT.E1`) ausgelöst.
4. Gleichzeitig wird das Gültigkeitssignal des internen Bausteins an den D-Eingang des D-Flipflops (`E_D_FF`) gelegt. Das gleiche Ereignis (`CNF` des internen Bausteins) taktet das Flipflop.

5. Das Flipflop speichert den Gültigkeitswert und gibt ihn stabil an `VALID.D1` aus. Ein Ereignis wird an `VALID.E1` gesendet.

Somit wird das Byte immer dann an `OUT` weitergegeben, wenn es gültig ist. Der Gültigkeitszustand bleibt bis zum nächsten Verarbeitungsdurchlauf erhalten.

Technische Besonderheiten

- **Adapter-basierte Kommunikation**: Der FB nutzt ausschließlich Adapter-Schnittstellen, was eine modulare und typisierte Verbindung in der 4diac-IDE ermöglicht.
- **D-Flipflop zur Entprellung**: Das Gültigkeitssignal wird durch ein Flipflop getaktet, um Stabilität zu gewährleisten und Hutschiene-Effekte zu vermeiden.
- **Lizenz**: Der Baustein ist unter der **Eclipse Public License 2.0** veröffentlicht (Copyright HR Agrartechnik GmbH).
- **Compiler-Package**: `logiBUS::signalprocessing::fieldbus` – spezifisch für Feldbus-Anwendungen.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten auf der obersten Ebene; die Zustände ergeben sich aus dem Zusammenwirken der internen Komponenten:

Zustand	Beschreibung
Idle	Warten auf ein Ereignis an `IN.E1`.
Processing	Interner `FIELDBUS_BYTE_TO_SIGNAL` verarbeitet das Byte; `OUT` und `VALID` werden aktualisiert.
Valid stable	Nachdem das Flipflop getaktet wurde, bleibt `VALID.D1` bis zum nächsten Ereignis stabil.

In jedem Zyklus wird der Zustand durchlaufen.

Anwendungsszenarien

- **Feldbus-Integration**: Empfang eines Byte-Wertes von einem Feldbus (z.B. CAN, Profibus) und dessen Weitergabe an eine Steuerungslogik, wobei nur gültige Telegramme durchgereicht werden.
- **Signalaufbereitung**: Umwandlung eines rohen Byte-Streams in ein getaktetes, gültigkeitsgeprüftes Signal für nachgeschaltete Funktionsblöcke.
- **Fehlererkennung**: Der Baustein kann mit einem externen Gültigkeitsprüfer kombiniert werden, der am `IN`-Adapter anliegt.

Vergleich mit ähnlichen Bausteinen

Einfachere **Mirror**-Bausteine leiten ein Signal ohne Gültigkeitsprüfung weiter. Dieser FB fügt eine explizite Validierung hinzu und gibt das Gültigkeitssignal separat aus. Im Gegensatz zu einem reinen Flipflop-basierten Haltebaustein verarbeitet er hier ein Byte und nicht nur boolesche Werte. Der interne `FIELDBUS_BYTE_TO_SIGNAL` übernimmt die spezifische Feldbus-Interpretation.

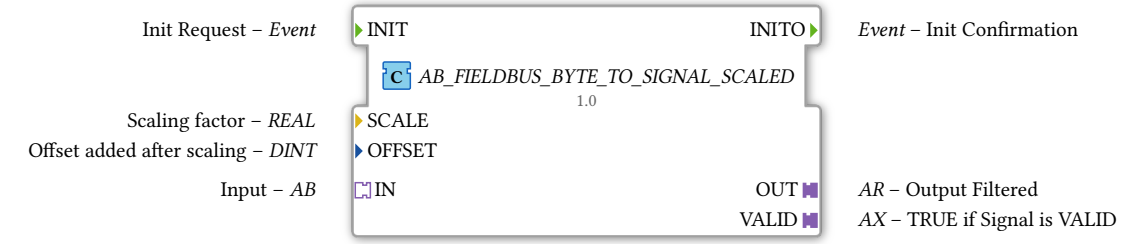
Fazit

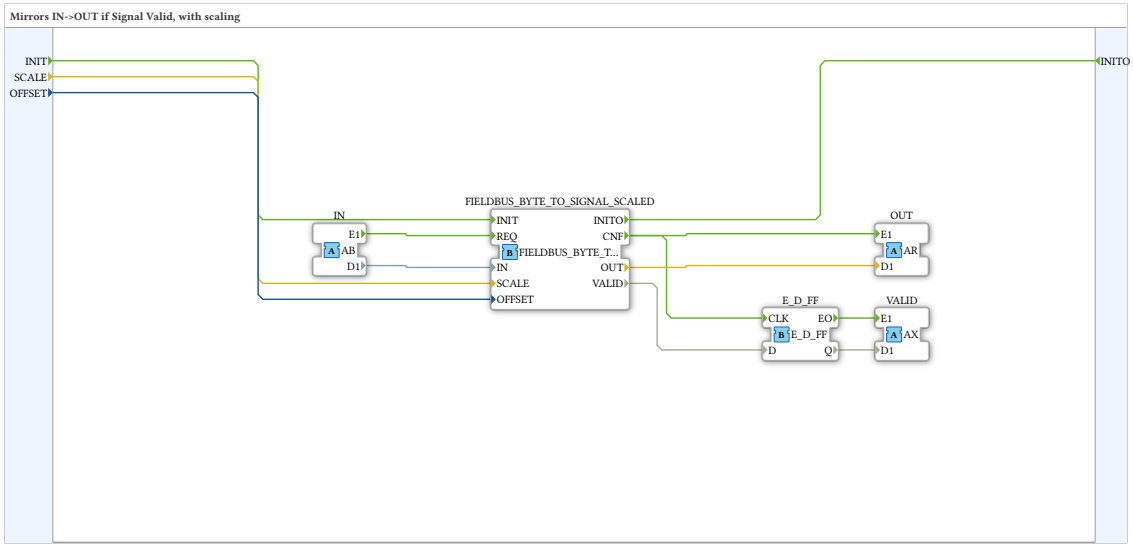
Der `AB_FIELDBUS_BYTE_TO_SIGNAL` ist ein spezialisierter Baustein für die sichere Weitergabe von Feldbus-Byte-Signalen. Durch die Kombination aus Spiegelung und Gültigkeitsprüfung eignet er sich ideal für Echtzeit-Anwendungen, bei denen nur valide Daten weiterverarbeitet werden dürfen. Die Adapter-basierte Schnittstelle erlaubt eine einfache Integration in bestehende 4diac-Projekte.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AB_FIELDBUS_BYTE_TO_SIGNAL_SCALED





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `AB_FIELDBUS_BYTE_TO_SIGNAL_SCALED` dient dazu, ein eingehendes BYTE-förmiges Signal über einen Adapter zu empfangen, es mit einem konfigurierbaren Skalierungsfaktor und Offset zu verarbeiten und das Ergebnis über einen Ausgangs-Adapter bereitzustellen. Gleichzeitig wird über einen separaten Adapter die Gültigkeit (Validität) des Signals signalisiert. Der Baustein ist speziell für den Einsatz in Feldbus-Umgebungen konzipiert, in denen Rohdaten in physikalische oder normierte Werte umgerechnet werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung
`INIT`	EInit	Initialisierungsanforderung (z. B. zum Setzen von Parametern)

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung
`INITO`	EInit	Bestätigung der erfolgreichen Initialisierung

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung
`SCALE`	REAL	1.0	Skalierungsfaktor, der auf den Rohwert angewendet wird
`OFFSET`	DINT	0	Ganzzahliger Offset, der nach der Skalierung addiert wird

Daten-Ausgänge

Der Baustein besitzt keine eigenständigen Daten-Eingänge oder -Ausgänge – die Ein-/Ausgabe erfolgt ausschließlich über Adapter.

Adapter

Adapter	Typ	Richtung	Beschreibung
`IN`	AB (Adapter Typ A⟶B)	Socket	Empfang des rohen BYTE-Signals
`OUT`	AR (Adapter Typ A⟶R)	Plug	Ausgabe des skalierten Signals
`VALID`	AX (Adapter Typ A⟶X)	Plug	Signalisiert, ob der aktuelle Ausgabewert gültig ist (TRUE = gültig)

Funktionsweise

1. Nach einer erfolgreichen Initialisierung (`INIT` ⟶ `INITO`) wird der Baustein betriebsbereit.
2. Sobald ein neues BYTE über den `IN`-Adapter eingeht (Ereignis `E1`), wird das Ereignis an den internen Funktionsblock `FIELDBUS_BYTE_TO_SIGNAL_SCALED` weitergeleitet.
3. Dieser interne Block berechnet den Ausgabewert: `OUT = IN * SCALE + OFFSET`. Gleichzeitig wird ein Gültigkeitssignal (`VALID`) erzeugt.
4. Nach Abschluss der Berechnung wird das Ergebnis über den `OUT`-Adapter ausgegeben (`OUT.E1`). Zeitgleich wird das Gültigkeitssignal in einem D-Flipflop (Baustein `E_D_FF`) zwischengespeichert und am `VALID`-Adapter ausgegeben.
5. Der Valid-Status bleibt solange erhalten, bis ein neues Signal verarbeitet wird. Ein ungültiger Zustand (z. B. bei fehlendem Eingang) kann durch entsprechende Logik extern gesetzt werden.

Der Ablauf ist rein ereignisgesteuert: Jedes eingehende Ereignis löst genau eine Verarbeitung aus.

Technische Besonderheiten

- **Skalierung und Offset**: Die Konfiguration von SCALE (REAL) und OFFSET (DINT) ermöglicht eine flexible Anpassung an verschiedene Messbereiche.
- **Adapter-Schnittstelle**: Die Verwendung von IEC 61499-Adaptern erlaubt eine lose Kopplung und einfache Wiederverwendbarkeit in unterschiedlichen Feldbus-Kontexten.
- **Valid-Speicherung**: Das D-Flipflop (E_D_FF) sorgt dafür, dass der Gültigkeitsstatus bis zum Eintreffen des nächsten Signals stabil bleibt.
- **Initialisierung**: Der Baustein besitzt einen eigenen INIT-Pfad, um interne Parameter (SCALE, OFFSET) sicher zu setzen.

Zustandsübersicht

Der Baustein definiert keinen expliziten Zustandsautomaten (ECC). Die Verarbeitung erfolgt ausschließlich ereignisgesteuert:

- **Im Ruhezustand** wartet er auf ein Ereignis.
- **Bei INIT** wird der interne Baustein initialisiert.
- **Bei IN.E1** wird die Übersetzung ausgelöst und die Ergebnisse werden asynchron ausgegeben.

Damit verhält sich der Baustein wie eine kombinatorische, aber getaktete Funktion.

Anwendungsszenarien

- **Feldbus-Signalaufbereitung**: Empfang von Rohdaten (z. B. Drucksensorwerte als BYTE) und Umrechnung in physikalische Einheiten (z. B. bar) mittels Skalierung und Offset.
- **Normierung von Messwerten**: Anpassung an einen normierten Wertebereich, z. B. 0…100 % oder -50…+50 °C.
- **Gültigkeitsprüfung**: Wenn der Sensor nur bei gültiger Messung ein Signal sendet, kann der Valid-Adapter verwendet werden, um nachgeschaltete Blöcke zu steuern.

Vergleich mit ähnlichen Bausteinen

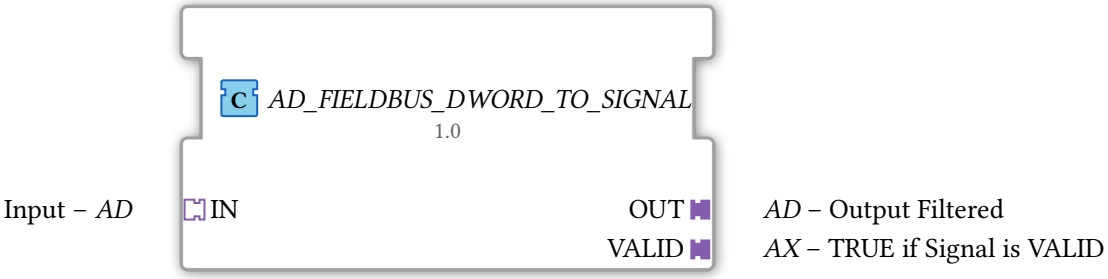
- **AB_FIELDBUS_BYTE_TO_SIGNAL** (ohne „SCALED“): Bietet keine Skalierung/Offset, sondern überträgt den Rohwert eins zu eins.
- **AB_FIELDBUS_SIGNAL_SCALED** (auf REAL-Basis): Arbeitet evtl. direkt mit REAL-Werten, benötigt aber eine andere Adapter-Konfiguration. Der vorliegende Baustein kombiniert die BYTE- zu

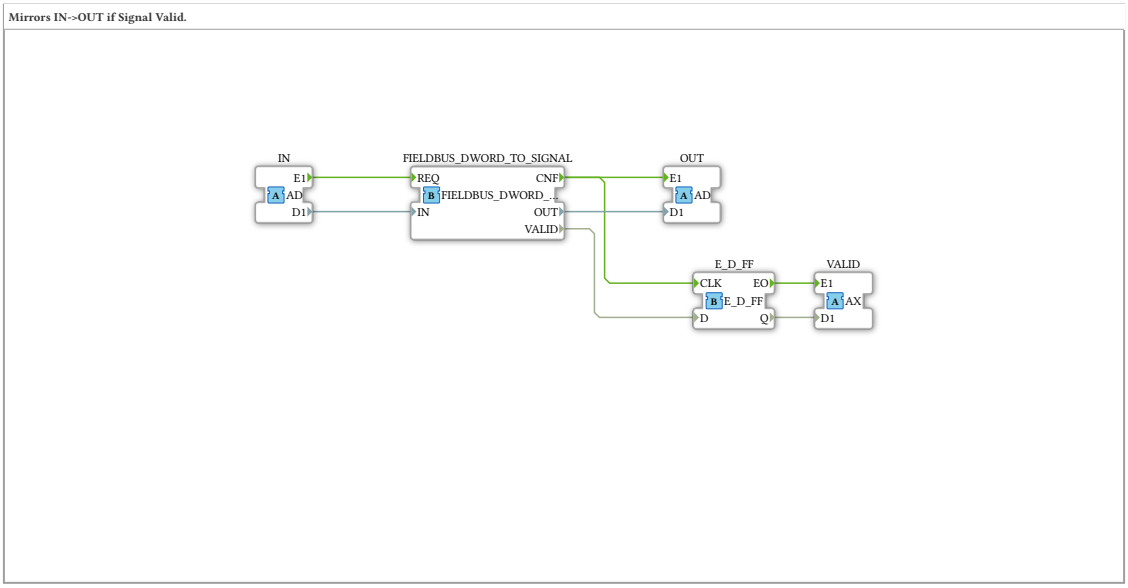
REAL-Umwandlung mit einer kompakten, parametrierbaren Skalierung und einem ausgegebenen Gültigkeitssignal.

Fazit

`AB_FIELDBUS_BYTE_TO_SIGNAL_SCALED` ist ein spezialisierter Funktionsblock zur flexiblen Umrechnung von BYTE-Rohdaten in skalierte Werte mit integrierter Gültigkeitssignalisierung. Durch seine Adapter-basierte Schnittstelle lässt er sich leicht in IEC 61499-Systeme integrieren und eignet sich besonders für Feldbusanwendungen, bei denen Rohwerte in physikalische Größen transformiert werden müssen.

AD_FIELDBUS_DWORD_TO_SIGNAL





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `AD_FIELDBUS_DWORD_TO_SIGNAL` dient dazu, ein eingehendes Datenwort (DWORD) an den Ausgang weiterzuleiten – jedoch nur, wenn das Signal als gültig (VALID) eingestuft wird. Er kombiniert einen spezialisierten Feldbus-Baustein mit einem flankengetriggerten D-Flip-Flop, um eine zuverlässige, datengetriebene Validierung und Weitergabe zu realisieren.

Schnittstellenstruktur

Der FB besitzt ausschließlich Adapter-Schnittstellen, keine separaten Ereignis- oder Datenports auf oberster Ebene. Die folgenden Adapter definieren die Ein- und Ausgänge:

Adapter	Typ	Richtung	Beschreibung
`IN`	`adapter::types::unidirectional::AD`	Socket (Eingang)	Eingangsdatenwort und zugehöriges Ereignis
`OUT`	`adapter::types::unidirectional::AD`	Plug (Ausgang)	Gefiltertes Ausgangsdatenwort (DWORD)

VALID	adapter::types::unidirectional::AX	Plug (Ausgang)	Gültigkeitssignal (BOOL) des aktuellen Datenworts

Ereignis-Eingänge (über Adapter `IN`)

Port	Beschreibung
E1	Ereignis zum Start der Verarbeitung eines neuen Datenworts

Ereignis-Ausgänge (über Adapter `OUT` und `VALID`)

Adapter	Port	Beschreibung
OUT	E1	Signalisiert, dass das gefilterte Datenwort am Ausgang anliegt
VALID	E1	Signalisiert, dass der Gültigkeitsstatus (TRUE/FALSE) aktualisiert wurde

Daten-Eingänge (über Adapter `IN`)

Port	Typ (angenommen)	Beschreibung
D1	DWORD	Das zu verarbeitende Datenwort

Daten-Ausgänge (über Adapter `OUT` und `VALID`)

Adapter	Port	Typ (angenommen)	Beschreibung
OUT	D1	DWORD	Das gefilterte – ggf. identische – Datenwort
VALID	D1	BOOL	TRUE, wenn das anliegende Datenwort als gültig bewertet wird, sonst FALSE

Adapter

Die Adapter sind vom Typ `unidirectional`, d.h. sie übertragen jeweils ein Ereignis und ein Datum in eine Richtung. Der FB verwendet zwei verschiedene Adaptertypen:

- **AD**: Überträgt ein Ereignis und ein Datenwort (DWORD)
- **AX**: Überträgt ein Ereignis und ein boolsches Signal

Funktionsweise

Das Zusammenspiel der internen Bausteine lässt sich in folgenden Schritten beschreiben:

1. Ein Ereignis an `IN.E1` triggert den internen FB `FIELDBUS_DWORD_TO_SIGNAL` über dessen `REQ`-Eingang.
2. Der interne FB verarbeitet das eingehende Datenwort (`IN.D1`) und gibt zwei Ergebnisse aus:
   - Das (ggf. identische) Datenwort an `OUT`
   - Ein boolsches Signal `VALID`, das angibt, ob das Datenwort gültig ist.
3. Nach Abschluss der Verarbeitung signalisiert der interne FB mit `CNF`:
   - Das Ereignis wird an `OUT.E1` weitergeleitet → der Ausgangsadapter gibt das neue Datenwort frei.
   - Gleichzeitig taktet das Ereignis den flankengetriggerten D-Flip-Flop `E_D_FF` über dessen `CLK`.
4. Der Flip-Flop übernimmt den aktuellen Gültigkeitsstatus (`VALID`-Signal) von `FIELDBUS_DWORD_TO_SIGNAL` an seinem `D`-Eingang und gibt ihn an seinem `Q`-Ausgang aus.
5. Der `EO`-Ausgang des Flip-Flops erzeugt ein Ereignis, das an `VALID.E1` gesendet wird – der Validitätsstatus wird somit zeitlich konsistent zum Datenwort aktualisiert.

Im Klartext: Der Baustein *spiegelt* das Eingangs-DWORD an den Ausgang, sofern der interne Validierungsmechanismus des Feldbus-Bausteins es als gültig einstuft. Die Gültigkeit wird durch einen Flip-Flop stabilisiert und als separates Signal ausgegeben.

Technische Besonderheiten

- **Composite-Baustein**: Der FB ist als Netzwerk aus zwei untergeordneten Bausteinen realisiert (`FIELDBUS_DWORD_TO_SIGNAL` und `E_D_FF`).
- **Lizenz**: Der Baustein unterliegt der Eclipse Public License 2.0 (EPL-2.0).

- **Paketstruktur**: Eingebunden in den Namespace `logiBUS::signalprocessing::fieldbus`.
- **Flankengesteuerter Gültigkeitsspeicher**: Der Einsatz eines D-Flip-Flops sorgt dafür, dass das `VALID`-Signal erst mit der nächsten Taktflanke (dem `CNF`-Ereignis) aktualisiert wird – dies verhindert asynchrone Zustandswechsel.
- **Keine eigenen Ereignis-/Datenports**: Die gesamte Kommunikation erfolgt ausschließlich über standardisierte Adapter.

Zustandsübersicht

Der FB besitzt keine explizite Zustandsmaschine, sondern arbeitet rein datenflusgesteuert. Der innere Flip-Flop `E_D_FF` kennt zwei interne Zustände:

Zustand	Beschreibung
`Q = FALSE`	Das aktuell ausgesendete `VALID`-Signal ist `FALSE` (Datenwort wird als ungültig betrachtet)
`Q = TRUE`	Das aktuell ausgesendete `VALID`-Signal ist `TRUE` (Datenwort gilt als gültig)

Der Zustand wechselt nur bei einer steigenden Flanke an `CLK` (entspricht dem `CNF`-Ereignis des internen Feldbus-Bausteins).

Anwendungsszenarien

- **Feldbus-Datenfilterung**: Einsatz in SPS-Steuerungen, bei denen nur gültige Telegramme eines Bussystems (z. B. CANopen, PROFIBUS) an die Anwendungslogik weitergegeben werden sollen.
- **Qualitätskennzeichnung**: Ein Sensor liefert einen Messwert und ein Validierungsbit – der Baustein trennt beide Informationen sauber und hält sie synchron.
- **Sichere Datenweitergabe**: In sicherheitskritischen Umgebungen kann der FB verwendet werden, um nur geprüfte Datenwörter in nachfolgende Berechnungen zu schleusen.

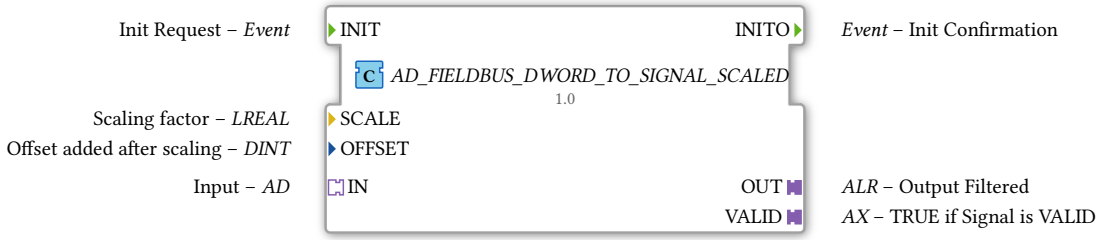
Vergleich mit ähnlichen Bausteinen

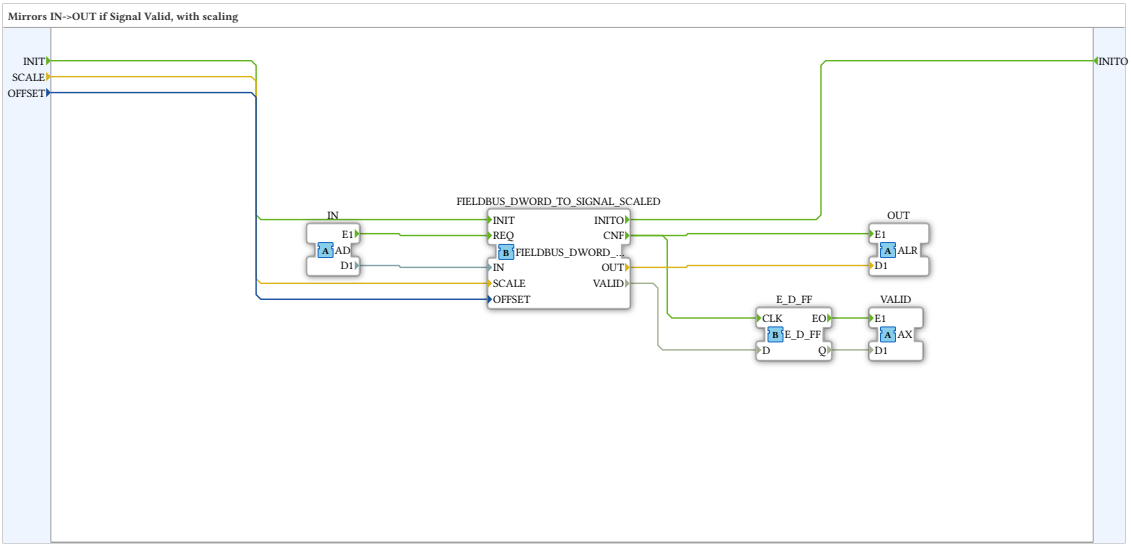
- **Einfache Buffer (z. B. `MOVE`)**: Diese geben Daten ohne Bewertung weiter. `AD_FIELDBUS_DWORD_TO_SIGNAL` fügt eine Validierungslogik hinzu und separiert das Gültigkeitssignal.
- **Typische Qualitäts-Bausteine (z. B. `QHandler`)**: Diese verarbeiten oft mehrere Qualitätsbits; dieser FB konzentriert sich auf ein einzelnes boolsches `VALID`-Signal und arbeitet mit DWORD-Daten.
- **Adapterbasierte Bausteine**: Durch die reinen Adapterschnittstellen wird eine lose Kopplung und Wiederverwendbarkeit in verschiedenen Laufzeitumgebungen gefördert.

Fazit

`AD_FIELDBUS_DWORD_TO_SIGNAL` bietet eine kompakte und standardisierte Lösung, um Feldbus-Datenwörter nur bei festgestellter Gültigkeit an die Anwendung weiterzuleiten. Die Kombination aus einem spezifischen Feldbus-Baustein und einem Flip-Flop gewährleistet eine zeitlich korrekte und stabile Ausgabe des Validierungsstatus. Aufgrund seiner Adapter-basierten Schnittstelle lässt sich der FB flexibel in bestehende IEC 61499-Systeme integrieren.

AD_FIELDBUS_DWORD_TO_SIGNAL_SCALED





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `AD_FIELDBUS_DWORD_TO_SIGNAL_SCALED` dient der Aufbereitung eines digitalen Feldbus-Signals (DWORD) zu einem skalierten analogen Ausgangswert. Er spiegelt den Eingangswert auf den Ausgang, sofern die Signalvalidität gegeben ist, und wendet dabei eine lineare Skalierung mit Offset an. Die gültige Signalzustandsinformation wird über einen flankengetriggerten Flip-Flop synchronisiert und als separates Ausgangssignal bereitgestellt.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung
INIT	EInit	Initialisierungsanforderung für den gesamten Baustein (Weiterleitung an internen Kernbaustein)

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung
INITO	EInit	Initialisierungsbestätigung nach erfolgreicher Initialisierung

Daten-Eingänge

Name	Datentyp	Initialwert	Beschreibung
SCALE	LREAL	1.0	Skalierungsfaktor, mit dem der Eingangswert multipliziert wird
OFFSET	DINT	0	Ganzzahliger Offset, der nach der Skalierung addiert wird

Daten-Ausgänge

Der Baustein besitzt keine direkten Datenausgänge. Alle Ausgangsdaten werden über die Adapter-Schnittstellen bereitgestellt.

Adapter

Name	Typ	Richtung	Beschreibung
IN	AD (unidirektional)	Socket	Eingangsadapter für den DWORD-Wert (Feldbussignal)
OUT	ALR (unidirektional)	Plug	Ausgangsadapter für den skalierten Analogwert

VALID	AX (unidirektional)	Plug	Ausgangsadapter für die Signalvalidität (TRUE = gültig)

Funktionsweise

Der Baustein realisiert die Signalkette:

1. **Initialisierung**: Das Ereignis `INIT` wird direkt an den internen Kernbaustein `FIELDBUS_DWORD_TO_SIGNAL_SCALED` weitergeleitet. Nach Abschluss wird `INITO` ausgegeben.
2. **Datenverarbeitung**: Der am Socket `IN` anliegende DWORD-Wert wird bei jedem Datenereignis (E1) übernommen. Der Kernbaustein multipliziert den Wert mit `SCALE` (LREAL) und addiert den `OFFSET` (DINT). Das Ergebnis wird als LREAL auf den Plug `OUT` gegeben.
3. **Validitätssignal**: Der Kernbaustein gibt ein internes boolsches Validitätssignal aus, das angibt, ob der momentane Datenwert gültig ist. Dieses Signal wird über einen flankengetriggerten D-Flipflop (`E_D_FF`) synchronisiert und auf den Plug `VALID` ausgegeben. Der Flipflop taktet mit dem gleichen Ereignis (`CNF` des Kernbausteins), sodass die Validität zeitgleich mit dem Ausgangswert aktualisiert wird.

Die interne Verschaltung stellt sicher, dass der Validitätsstatus immer dem letzten verarbeiteten Datenwert entspricht.

Technische Besonderheiten

- **Skalierungsreihenfolge**: Zuerst Multiplikation mit `SCALE` (LREAL), dann Addition von `OFFSET` (DINT). Dadurch kann der Offset auch in Einheiten des skalierten Bereichs angegeben werden.
- **Flankensynchronisation**: Das Validitätssignal wird über einen D-Flipflop flankengesteuert ausgegeben, sodass es keine metastabilen Zustände auf der Ausgangsseite gibt.
- **Initialwertvorgabe**: `SCALE` ist auf 1.0 vordefiniert, `OFFSET` auf 0. Im unskalierten Fall entspricht der Ausgang somit direkt dem Eingang.
- **Adapterstruktur**: Die Ein- und Ausgänge sind als Adapter realisiert, was eine typsichere Verbindung mit anderen Bausteinen aus derselben Adapterfamilie (AD, ALR, AX) ermöglicht.

Zustandsübersicht

Der Baustein besitzt kein eigenes explizites Zustandsdiagramm (ECC). Die gesamte Zustandslogik liegt im internen Kernbaustein `FIELDBUS_DWORD_TO_SIGNAL_SCALED` und im flankengetriggerten D-Flipflop. Dieser speichert den letzten gültigen Validitätszustand. Bei jedem Verarbeitungsereignis wird der Flipflop getaktet und gibt den aktuellen Validitätswert aus.

Anwendungsszenarien

- **Messwerterfassung über Feldbus**: Ein DWORD-Rohwert (z. B. von einem Analog-Digital-Wandler oder Encoder) wird mit physikalischen Faktoren (Skalierung) und Offset umgerechnet und als Analogwert an eine Steuerung übergeben.
- **Signalvalidierung mit Speicherung**: Wenn der Feldbus eine Gültigkeitsflagge (z. B. „Daten neu“) liefert, kann diese über die Validitätsadapterebene ausgewertet werden. Der Flipflop stellt sicher, dass das Gültigkeitssignal bis zur nächsten Aktualisierung stabil bleibt.
- **Parametrierbare Skalierung**: Durch externe Vorgabe von `SCALE` und `OFFSET` kann der gleiche Baustein für unterschiedliche Sensorbereiche verwendet werden, ohne den Baustein selbst zu ändern.

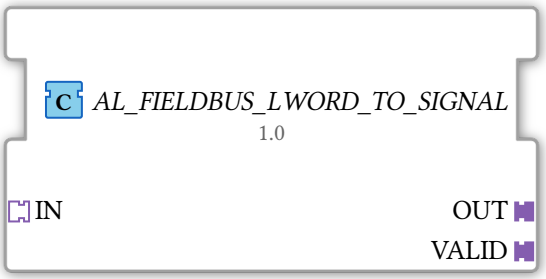
Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem einfachen `MOVE`- oder `SCALE`-Baustein bietet dieser FB eine integrierte Validitätsbehandlung und eine saubere Trennung zwischen Daten- und Ereignisadaptern. Ähnliche Bausteine (z. B. `FIELD_DWORD_TO_ANALOG`) fehlen oft die separate Ausgabe der Signalvalidität oder die flankensynchrone Speicherung. Die hier verwendete Kombination aus Kernbaustein und D-Flipflop ist eine bewährte Lösung für sicherheitskritische Anwendungen, bei denen die Gültigkeit eines Datenwerts zuverlässig mitgeführt werden muss.

Fazit

Der Funktionsblock `AD_FIELDBUS_DWORD_TO_SIGNAL_SCALED` ist ein robuster und parametrierbarer Baustein zur Umwandlung digitaler Feldbussignale in skalierte Analogwerte mit synchroner Validitätsausgabe. Dank der modularen Adapteranschlüsse und integrierten Flankensynchronisation eignet er sich besonders für industrielle Automatisierungssysteme, bei denen Datenkonsistenz und Signalqualität gleichermaßen wichtig sind.

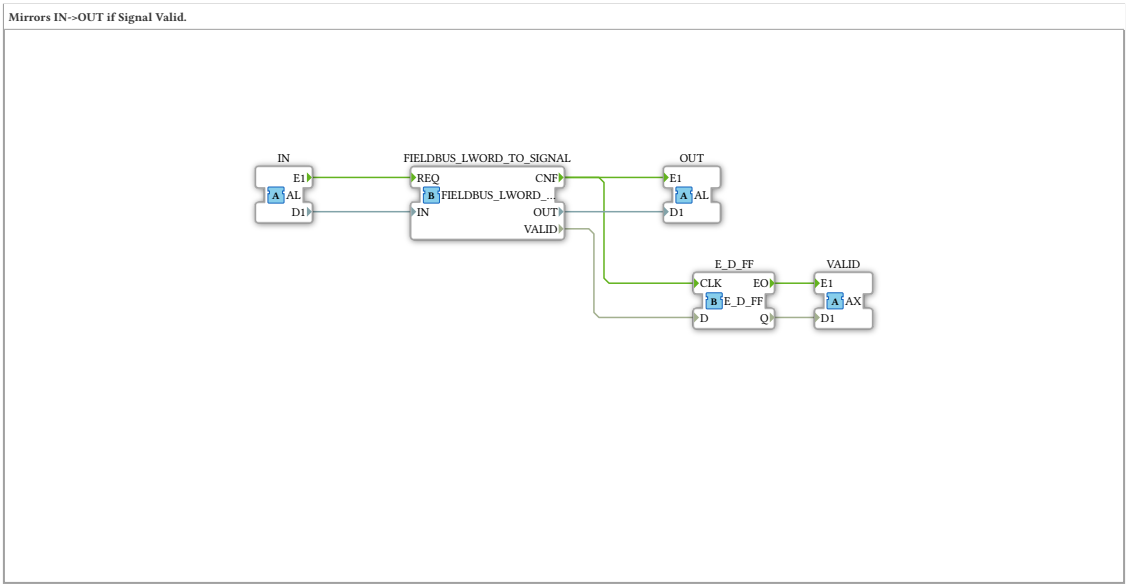
AL_FIELDBUS_LWORD_TO_SIGNAL



Input – *AL*

AL – Output Filtered

AX – TRUE if Signal is VALID



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AL_FIELDBUS_LWORD_TO_SIGNAL` dient der filternden Weitergabe eines Feldbus-Signals vom Typ `LWORD`. Er gibt den am Eingang anliegenden Wert nur dann an den Ausgang weiter, wenn das zugehörige Gültigkeitssignal (Valid) gesetzt ist. Gleichzeitig wird der Gültigkeitsstatus über einen separaten Ausgang bereitgestellt und durch einen internen Flipflop stabilisiert. Der Baustein ist als Composite-FB realisiert und kapselt die Logik zur Signalverarbeitung sowie die Zustandshaltung des Valid-Signals.

Schnittstellenstruktur

Ereignis-Eingänge

- **`IN.E1`** (über Socket `IN`): Ereignis zum Anfordern der Signalverarbeitung. Ein eingehendes Ereignis startet die Spiegelung des Eingangswerts auf den Ausgang und aktualisiert den Gültigkeitsstatus.

Ereignis-Ausgänge

- **`OUT.E1`** (über Plug `OUT`): Ereignis, das nach erfolgreicher Verarbeitung ausgegeben wird. Signalisiert, dass der aktuelle Ausgangswert (`OUT.D1`) gültig ist.
- **`VALID.E1`** (über Plug `VALID`): Ereignis, das zeitgleich mit `OUT.E1` ausgegeben wird. Bestätigt die Aktualisierung des Gültigkeitssignals (`VALID.D1`).

Daten-Eingänge

- **`IN.D1`** (über Socket `IN`, Typ: `LWORD`): Das zu verarbeitende Feldbus-Signal. Der Wert wird unverändert an den Ausgang weitergegeben, sofern das interne Gültigkeitssignal aktiv ist.

Daten-Ausgänge

- **OUT.D1** (über Plug OUT, Typ: LWORD): Das gefilterte Ausgangssignal. Enthält den Wert von IN.D1, wenn das Signal gültig ist, andernfalls bleibt der letzte gültige Wert erhalten (durch das interne Verhalten des verwendeten Bausteins FIELDBUS_LWORD_TO_SIGNAL).
- **VALID.D1** (über Plug VALID, Typ: BOOL): Zeigt an, ob das aktuell ausgegebene Signal gültig ist. Der Wert wird durch ein internes Flipflop (E_D_FF) zwischengespeichert und bei jedem Verarbeitungszyklus aktualisiert.

Adapter

Adapter	Richtung	Typ	Beschreibung
IN	Socket	adapter::types::unidirectional::AL	Empfängt das Ereignis- und Datensignal von einem vorgeschalteten Baustein.
OUT	Plug	adapter::types::unidirectional::AL	Sendet das gefilterte Signal und zugehöriges Ereignis an nachfolgende Bausteine.
VALID	Plug	adapter::types::unidirectional::AX	Sendet den Gültigkeitsstatus als booleschen Wert und ein Bestätigungsereignis.

Funktionsweise

Der Baustein arbeitet als Composite-FB mit folgender interner Logik:

1. Ein eingehendes Ereignis an IN.E1 triggert den internen FB FIELDBUS_LWORD_TO_SIGNAL über dessen REQ-Eingang.
2. Dieses interne FB spiegelt den Wert von IN.D1 auf seinen OUT-Ausgang und erzeugt gleichzeitig ein boolesches VALID-Signal, das angibt, ob der Wert gültig ist.
3. Das CNF-Ereignis des internen FB wird auf drei Ziele verteilt:
   - Es triggert den Ausgangsplug OUT.E1, sodass der gefilterte Wert (OUT.D1) an die nächste Verarbeitungsstufe weitergegeben wird.
   - Es triggert den CLK-Eingang des Flipflops E_D_FF.
4. Das VALID-Signal des internen FB wird an den D-Eingang des Flipflops gelegt. Bei jeder positiven Flanke des Takts (ausgelöst durch CNF) übernimmt das Flipflop diesen Wert und gibt ihn an Q aus.
5. Der Ausgang Q des Flipflops wird auf VALID.D1 gelegt und das gleichzeitige Ereignis VALID.E1 wird durch das Flipflop-Ereignis EO ausgelöst.

Dadurch wird sichergestellt, dass der Gültigkeitsstatus erst dann an den Valid-Ausgang weitergegeben wird, wenn die Signalverarbeitung abgeschlossen ist. Der ausgegebene Wert OUT.D1 entspricht dem Eingangswert IN.D1, sofern dieser als gültig erkannt wurde; andernfalls wird der letzte gültige Wert beibehalten (abhängig von der Implementierung des internen FB FIELDBUS_LWORD_TO_SIGNAL).

Technische Besonderheiten

- **Composite-Architektur:** Die Funktionalität wird durch ein internes Netzwerk aus einem spezialisierten FB (FIELDBUS_LWORD_TO_SIGNAL) und einem flankengetriggerten Flipflop (E_D_FF) realisiert. Dies ermöglicht eine modulare und testbare Implementierung.
- **Zustandshaltung:** Der Gültigkeitsstatus wird über ein Flipflop gespeichert. Dadurch bleibt der Valid-Ausgang stabil zwischen zwei Verarbeitungszyklen und wird nur bei neuen Ereignissen aktualisiert.
- **Unidirektionale Adapter:** Alle Schnittstellen sind als unidirektionale Adapter (AL bzw. AX) ausgeführt, was eine klare Trennung von Ereignis- und Datenfluss ermöglicht.
- **Lizenzierung:** Der Baustein ist unter der Eclipse Public License 2.0 verfügbar (Copyright HR Agrartechnik GmbH).

Zustandsübersicht

Der Baustein besitzt keinen expliziten ECC, da es sich um einen Composite-FB handelt. Die innere Zustandslogik beschränkt sich auf das Flipflop E_D_FF, das zwei Zustände kennt:

Zustand	Q (VALID.D1)	Bedeutung

RESET (Anfang)	FALSE	Signal ist ungültig (initial).
SET	TRUE	Signal ist gültig, nachdem ein gültiger Wert erkannt wurde.

Der Zustandswechsel erfolgt bei jedem eingehenden Ereignis an `CLK` (von `CNF`) in Abhängigkeit des `D`-Eingangs (vom internen `VALID`-Signal). Der Zustand bleibt solange erhalten, bis das nächste Ereignis eintrifft.

Anwendungsszenarien

- **Feldbus-Signalverarbeitung:** Übernahme von LWORD-Signalen (z.B. aus einem CANopen- oder Profibus-Netzwerk) und deren Weitergabe nur bei gültigem Status.
- **Filterung ungültiger Werte:** Wenn ein Sensor zeitweise ungültige Messwerte liefert, gibt der Baustein den letzten gültigen Wert weiter und signalisiert die Ungültigkeit über `VALID.D1`.
- **Synchronisation mehrerer Pfade:** Durch die getrennten Ausgänge für Signal und Valid können nachgelagerte Bausteine beide Informationen gleichzeitig verarbeiten.

Vergleich mit ähnlichen Bausteinen

Einfachere Bausteine wie `MOVE` oder `REPEAT` geben eingehende Werte ungefiltert weiter, ohne eine Gültigkeitsprüfung. Der `AL_FIELDBUS_LWORD_TO_SIGNAL` erweitert diese Funktionalität um eine Validierungslogik und einen gesonderten Gültigkeitsausgang. Andere Bausteine aus der `logiBUS`-Bibliothek könnten ähnliche Aufgaben für andere Datentypen (z.B. `WORD` oder `BOOL`) bieten, sind aber nicht auf `LWORD` spezialisiert.

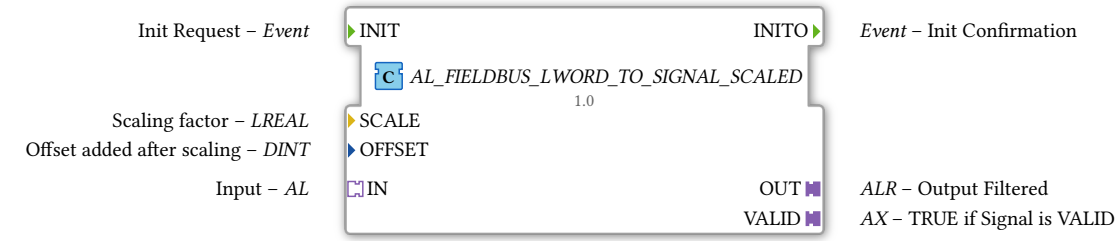
Fazit

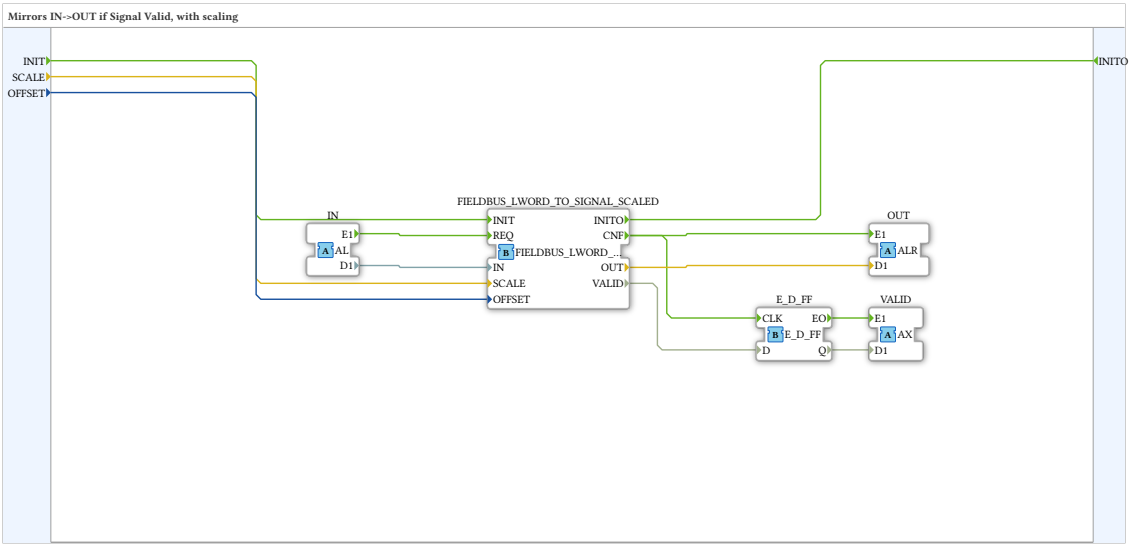
Der Funktionsblock `AL_FIELDBUS_LWORD_TO_SIGNAL` ist ein spezialisierter Composite-Baustein zur zuverlässigen Weitergabe von Feldbus-Signalen unter Berücksichtigung eines Gültigkeitskriteriums. Durch die Kombination aus Signalverarbeitung und Zustandshaltung eignet er sich ideal für sicherheitskritische oder qualitätsbewusste Anwendungen in der Automatisierungstechnik. Die klar definierten Adapter-Schnittstellen erleichtern die Integration in bestehende 4diac-IDE Projekte.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AL_FIELDBUS_LWORD_TO_SIGNAL_SCALED





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `AL_FIELDBUS_LWORD_TO_SIGNAL_SCALED` dient der Umwandlung eines Feldbus-Signals (LWORD) in einen skalierten und mit Offset versehenen Wert. Dabei wird nur dann ein gültiges Ausgangssignal erzeugt, wenn das Eingangssignal als gültig markiert ist. Die Gültigkeit wird über ein internes D-Flipflop stabil gehalten und als boolesches Signal ausgegeben.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Init-Anforderung

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Init-Bestätigung

Daten-Eingänge

Name	Typ	Anfangswert	Kommentar
SCALE	LREAL	1.0	Skalierungsfaktor
OFFSET	DINT	0	Nach der Skalierung addierter Offset

Adapter

Typ	Richtung	Name	Kommentar
AL	Socket	IN	Eingangssignal (LWORD)
ALR	Plug	OUT	Ausgangssignal (skaliert)
AX	Plug	VALID	Signal gültig (TRUE/FALSE)

Funktionsweise

Der FB enthält intern zwei Komponenten:

- **FIELDBUS_LWORD_TO_SIGNAL_SCALED**: Führt die eigentliche Umrechnung des LWORD-Eingangs mithilfe der Parameter SCALE und OFFSET durch. Bei einem Ereignis an `IN.E1` (verbunden mit `REQ`) wird der verarbeitete Wert auf `OUT` und das Gültigkeitssignal auf `VALID` gesetzt.
- **E_D_FF**: Ein flankengesteuertes D-Flipflop, das das Gültigkeitssignal aus dem vorherigen Baustein übernimmt. Der Takt (`CLK`) wird durch das Bestätigungsereignis (`CNF`) des ersten Bausteins ausgelöst. Der Ausgang `Q` wird auf den Adapter `VALID.D1` geschrieben.

Somit ergibt sich folgende Ablaufkette:

1. Ein eingehendes Ereignis an `IN.E1` startet die Verarbeitung.
2. Der interne Baustein berechnet den skalierten Wert und gibt das Gültigkeitssignal aus.
3. Das Flipflop stabilisiert dieses Signal und gibt es an den `VALID`-Adapter weiter.
4. Gleichzeitig wird der skalierte Wert über den `OUT`-Adapter ausgegeben.

Die INIT-Schnittstelle dient der Initialisierung und wird direkt durchgeschliffen.

Technische Besonderheiten

- Die INIT- und INITO-Verbindungen sind im Netzwerk als unsichtbar markiert, sie existieren jedoch für eine saubere Initialisierungskaskade.
- Skalierung und Offset werden intern an den unterlagerten FB weitergeleitet und sind direkt konfigurierbar.
- Das D-Flipflop verhindert unerwünschte Flanken oder kurzzeitige Invaliditätsänderungen am Ausgang.

Zustandsübersicht

Der FB besitzt keine explizite Zustandsmaschine (ECC). Sein Verhalten wird durch die Ereignisverarbeitung gesteuert:

- **Nach INIT**: Der FB ist bereit, Ereignisse an `IN.E1` zu empfangen.
- **Nach jedem REQ/CNF-Zyklus**: Ein neuer skalierten Wert und ein aktualisiertes Gültigkeitssignal liegen an den Adaptern an.

Anwendungsszenarien

Typische Einsatzbereiche sind:

- Umwandlung von Feldbus-Rohdaten (z. B. LWORD aus PROFIBUS, CAN etc.) in physikalische Größen (Temperatur, Druck, Drehzahl) mit benutzerdefinierter Skalierung.
- Weiterleitung von Sensordaten, bei denen nur als gültig markierte Werte an die nachfolgende Logik übergeben werden sollen.
- Integration in Anlagensteuerungen, die eine saubere Initialisierungsreihenfolge (über INIT) erfordern.

Vergleich mit ähnlichen Bausteinen

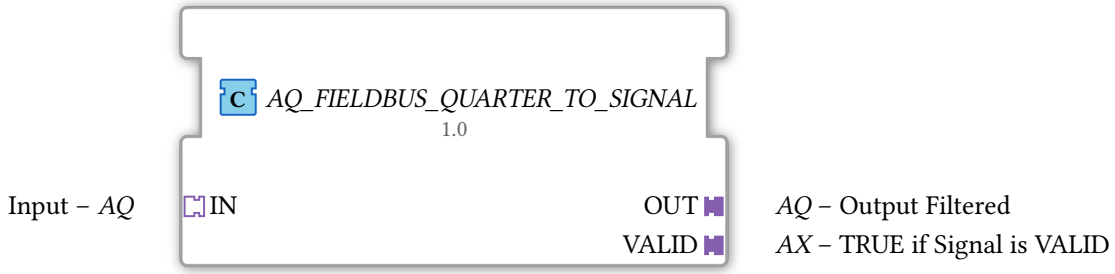
Einfachere Bausteine zur LWORD-Konvertierung bieten oft keine integrierte Gültigkeitsprüfung und geben das Signal direkt aus. `AL_FIELDBUS_LWORD_TO_SIGNAL_SCALED` ergänzt diese Funktionalität um:

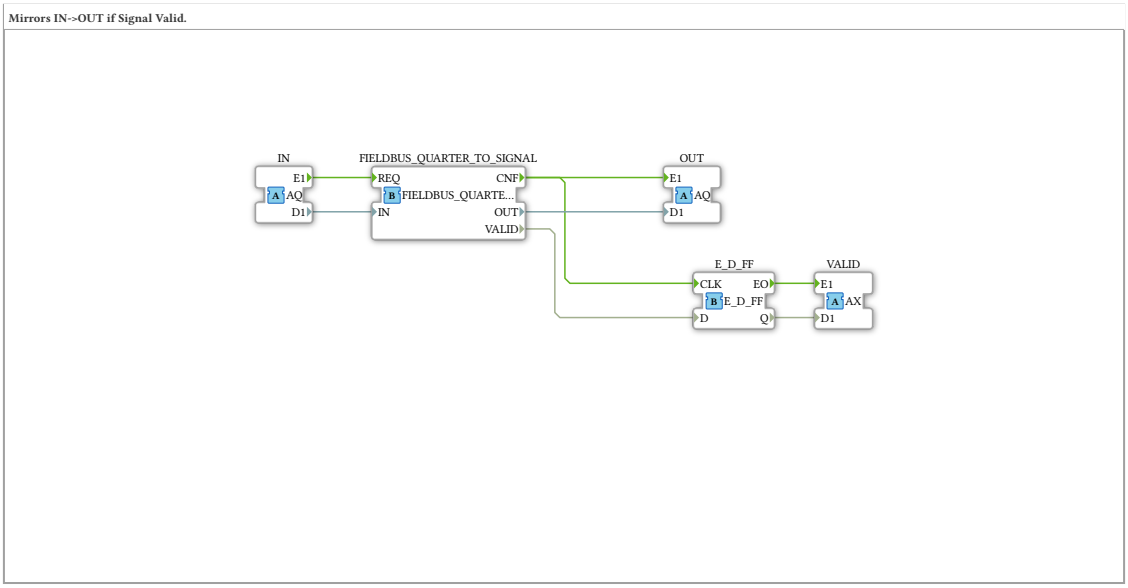
- Ein separates Gültigkeitssignal, flankengesteuert über D-Flipflop.
- Explizite Initialisierungsschnittstelle. Die Kombination von Skalierung, Offset und Gültigkeitslatch hebt ihn von reinen Umrechnungsbausteinen ab.

Fazit

Dieser Funktionsblock stellt eine robuste Lösung für die Aufbereitung von Feldbus-Signalen dar. Er kombiniert Skalierung, Offset-Verschiebung und eine zuverlässige Gültigkeitsanzeige in einem kompakten, initialisierbaren Baustein. Durch die Verwendung eines D-Flipflops wird die Signalqualität stabilisiert, was insbesondere in sicherheitskritischen oder rauschbehafteten Umgebungen von Vorteil ist.

AQ_FIELDBUS_QUARTER_TO_SIGNAL





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AQ_FIELDBUS_QUARTER_TO_SIGNAL** dient dazu, ein über einen Feldbusadapter eingehendes Signal (vom Typ `AQ`) auf einen Ausgangsadapter zu spiegeln – jedoch nur, wenn das Signal als gültig (valid) gekennzeichnet ist. Er kombiniert einen feldbusspezifischen Verarbeitungsbaustein mit einem flankengesteuerten D-Flipflop, um den Gültigkeitsstatus zu puffern und als separates Ausgangssignal bereitzustellen.

Schnittstellenstruktur

Ereignis-Eingänge

Der Baustein besitzt keine direkten Ereignis-Eingänge auf oberster Ebene. Stattdessen werden Ereignisse über den **Socket-Adapter `IN`** (Typ `AQ`) bereitgestellt. Der Ereignisausgang `IN.E1` startet die Verarbeitung des eingehenden Signals.

Ereignis-Ausgänge

Auch Ereignis-Ausgänge werden ausschließlich über die **Plug-Adapter `OUT`** und `VALID` geführt:

- **`OUT.E1`**: Wird nach erfolgreicher Spiegelung des Signals ausgelöst. (Ergebnis liegt am Datenausgang `OUT.D1` an.)
- **`VALID.E1`**: Wird ausgelöst, sobald der Gültigkeitsstatus des Signals aktualisiert wurde (steigende Flanke des internen D-Flipflops).

Daten-Eingänge

Daten werden ebenfalls über den **Socket-Adapter `IN`** übernommen:

- **`IN.D1`**: Empfängt das analoge oder digitale Eingangssignal (Typ des Adapters `AQ`).

Daten-Ausgänge

- **`OUT.D1`**: Gibt das gespiegelte Eingangssignal aus – aber nur, wenn es als gültig erkannt wurde. (Bei ungültigem Signal kann ein definierter Wert oder letzter gültiger Wert ausgegeben werden; dies hängt von der internen Implementierung von `FIELDBUS_QUARTER_TO_SIGNAL` ab.)
- **`VALID.D1`**: Gibt den aktuellen Gültigkeitsstatus aus (`TRUE` = gültig, `FALSE` = ungültig). Dieser Wert ist das Ausgangssignal `Q` des D-Flipflops.

Adapter

Name	Typ	Richtung	Beschreibung
`IN`	`adapter::types::unidirectional::AQ`	**Socket** (Eingang)	Feldbussignal (Ereignis und Daten)
`OUT`	`adapter::types::unidirectional::AQ`	**Plug** (Ausgang)	Gefiltertes / gespiegeltes Ausgangssignal
`VALID`	`adapter::types::unidirectional::AX`	**Plug** (Ausgang)	Gültigkeitsstatus des Signals

Hinweis: Die Adaptertypen `AQ` und `AX` sind eindirektionale Protokolle, die üblicherweise ein Ereignis- und ein Datenfeld kombinieren.

Funktionsweise

Der Baustein arbeitet wie folgt:

1. Ein eingehendes Ereignis auf `IN.E1` triggert den internen Funktionsbaustein **`FIELDBUS_QUARTER_TO_SIGNAL`** (Aufruf über `REQ`).
2. Dieser interne Baustein verarbeitet den Datenwert (`IN.D1`) und erzeugt zwei Ausgaben:
   - **`OUT`** – das aufbereitete Signal (z. B. gespiegelt, normiert, gefiltert).
   - **`VALID`** – ein Boolescher Wert (`TRUE`, wenn das Signal gültig ist).
3. Das verarbeitete Signal wird direkt an den Ausgangsadapter `OUT` weitergeleitet (über `OUT.D1`) und gleichzeitig das Ereignis `OUT.E1` ausgelöst.
4. Der Gültigkeitsstatus (`VALID`) wird dem D-Eingang eines flankengesteuerten D-Flipflops (`E_D_FF`) zugeführt. Die Taktflanke wird durch dasselbe Ereignis (`CNF`) erzeugt, das auch `OUT.E1` triggert. Somit wird der Gültigkeitsstatus bei jedem Zyklus in das Flipflop übernommen.
5. Der Flipflop-Ausgang `Q` wird über den Adapter `VALID.D1` ausgegeben. Das dazugehörige Ereignis `VALID.E1` wird durch den Flipflop-Ausgang `EO` ausgelöst.

Der Baustein stellt sicher, dass das Ausgangssignal stets mit dem aktuell verarbeiteten Gültigkeitsstatus synchronisiert ist.

Technische Besonderheiten

- Der Baustein ist als **Komposition** zweier interner Funktionsbausteine realisiert: `FIELDBUS_QUARTER_TO_SIGNAL` und `E_D_FF`. Dies bietet eine hohe Wiederverwendbarkeit und klare Trennung von Verarbeitungslogik und Zustandsspeicherung.
- Die Verwendung eines **D-Flipflops** puffert den Gültigkeitsstatus zwischen den Ereigniszyklen. Dadurch bleibt der Status auch dann erhalten, wenn das Eingangssignal zwischenzeitlich ausfällt.
- Die Adapter-Schnittstellen folgen dem **unidirektionalen** Muster, das für Feldbusprotokolle typisch ist. Ein Ereignis transportiert sowohl einen Datenwert als auch eine Ausführungsflanke.
- Die interne Verarbeitung (`FIELDBUS_QUARTER_TO_SIGNAL`) kann je nach Anwendung angepasst werden (z. B. Skalierung, Formatierung oder Plausibilitätsprüfung), ohne die äußere Schnittstelle zu ändern.

Zustandsübersicht

Der Baustein besitzt keinen eigenen sichtbaren Zustandsautomaten. Der innere Zustand wird maßgeblich durch das **D-Flipflop** bestimmt:

- **Zustand 0**: Flipflop-Ausgang `Q = FALSE` ⟶ Signal gilt als **ungültig**.
- **Zustand 1**: Flipflop-Ausgang `Q = TRUE` ⟶ Signal gilt als **gültig**.

Der Zustandswechsel erfolgt taktflankengesteuert bei jedem Verarbeitungszyklus (Ereignis auf `IN.E1`). Der aktuelle Zustand ist über den Adapter `VALID.D1` ablesbar.

Anwendungsszenarien

- **Feldbusanbindung in der Landtechnik**: Übernahme eines analogen Sensorsignals (z. B. Drehmoment, Druck) von einem Feldbus, das nur bei gültiger Kommunikation weitergegeben wird.
- **Signalqualitätsfilterung**: Einbindung in eine Kette von Verarbeitungsbausteinen, bei der nur als gültig markierte Werte in die Regelung oder Visualisierung einfließen.
- **Redundanzprüfung**: Kombination mit einem zweiten Signalweg, wobei der Gültigkeitsstatus als Umschaltkriterium dient.

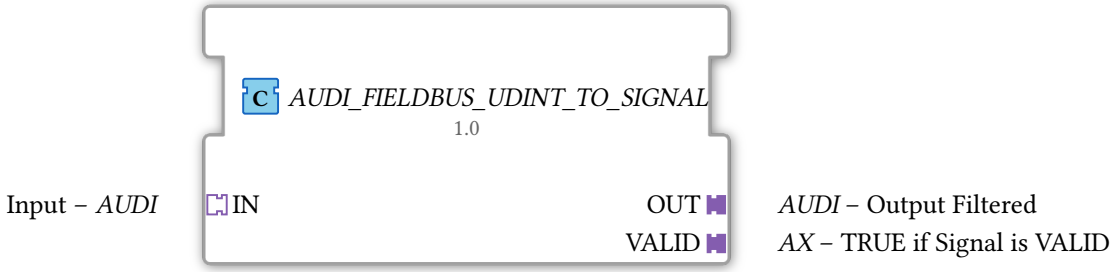
Vergleich mit ähnlichen Bausteinen

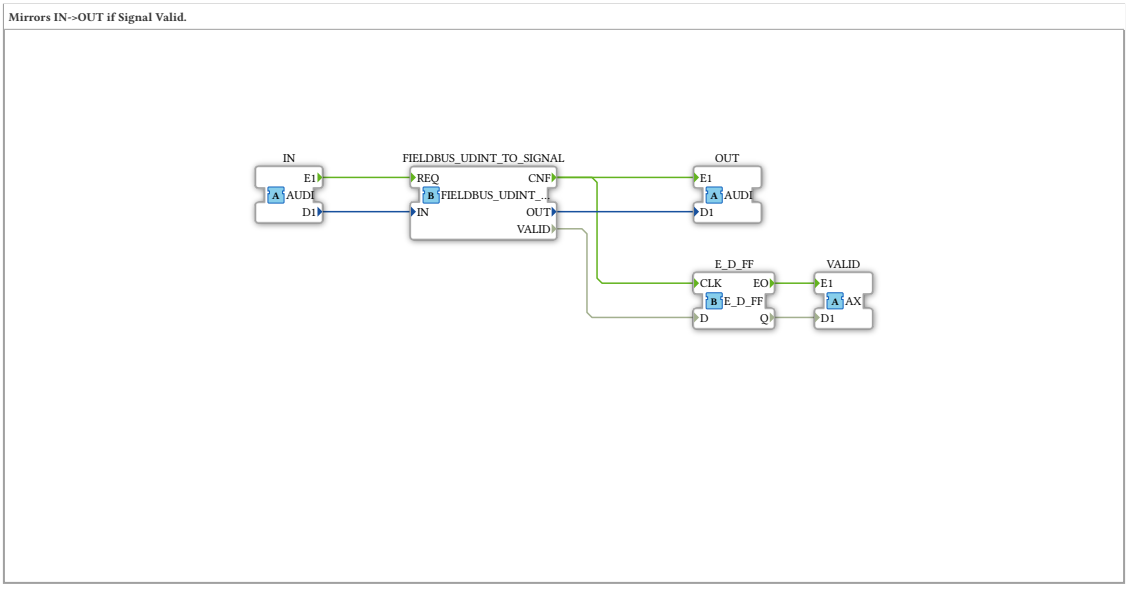
- **`AQ_FIELDBUS_SIGNAL_TO_QUARTER`** (hypothetisch): Würde den umgekehrten Weg – Umwandlung eines allgemeinen Signals in ein Feldbusformat – abbilden.
- **`AQ_BUFFER`**: Ein reiner Puffer ohne Gültigkeitsprüfung; der vorliegende Baustein erweitert diese Funktion um eine Validierung und Zustandsspeicherung.
- **`AQ_SELECT`**: Wählt zwischen zwei Eingängen aus; im vorliegenden Baustein wird stattdessen die Gültigkeit eines einzelnen Signals bewertet.

Fazit

AQ_FIELDBUS_QUARTER_TO_SIGNAL ist ein kompakter, adapterbasierter Funktionsbaustein zur zuverlässigen Spiegelung von Feldbussignalen unter Berücksichtigung der Signalvalidität. Durch die Kombination von Verarbeitungslogik und flankengesteuertem Zustandsspeicher eignet er sich besonders für sicherheitskritische oder qualitätsüberwachte Anwendungen im industriellen Umfeld. Die klare Trennung der internen Komponenten erleichtert Wartung und Anpassung.

AUDI_FIELDBUS_UDINT_TO_SIGNAL





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AUDI_FIELDBUS_UDINT_TO_SIGNAL` spiegelt einen eingehenden `UDINT`-Wert (über den `IN`-Adapter) auf den `OUT`-Adapter, sofern das anliegende Signal als gültig erkannt wurde. Zusätzlich wird

das Gültigkeitssignal über den VALID-Adapter ausgegeben. Der Baustein dient als Filter, der nur valide Datenpakete durchlässt und deren Status synchronisiert bereitstellt.

Schnittstellenstruktur

Der Baustein besitzt keine direkten Ein-/Ausgangs-Ereignisse oder Daten, sondern arbeitet ausschließlich über **Adapter**, die sowohl Ereignisse als auch Daten transportieren.

Ereignis-Eingänge

- **IN.E1** (über den IN-Adapter) – Startet die Verarbeitung des eingehenden Signals.

Ereignis-Ausgänge

- **OUT.E1** (über den OUT-Adapter) – Wird ausgelöst, nachdem der IN-Wert aufbereitet und an OUT.D1 weitergegeben wurde.
- **VALID.E1** (über den VALID-Adapter) – Wird ausgelöst, sobald die Gültigkeit des Signals aktualisiert wurde (siehe Funktionsweise).

Daten-Eingänge

- **IN.D1** (über den IN-Adapter) – Der eingehende UDINT-Wert (z. B. aus einem Feldbus).

Daten-Ausgänge

- **OUT.D1** (über den OUT-Adapter) – Der gefilterte UDINT-Wert (identisch mit IN.D1, wenn gültig).
- **VALID.D1** (über den VALID-Adapter) – Boolescher Wert, der TRUE signalisiert, sofern das anliegende Signal gültig ist.

Adapter

Adapter	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::AUDI	Socket	Signal-Eingang (liest Daten und Ereignisse)
OUT	adapter::types::unidirectional::AUDI	Plug	Gefilterter Signal-Ausgang
VALID	adapter::types::unidirectional::AX	Plug	Gültigkeitsanzeige (Ausgang für Bool & Ereignis)

Funktionsweise

Der Baustein enthält zwei interne Funktionsblöcke:

1. **FIELDBUS_UDINT_TO_SIGNAL** – Wandelt den eingehenden UDINT-Wert in ein Signal um und bestimmt dessen Gültigkeit.
2. **E_D_FF** – Ein flankengesteuertes D-Flipflop, das das Gültigkeitssignal synchronisiert.

Der Ablauf:

- Ein Ereignis auf IN.E1 triggert den Verarbeitungsblock FIELDBUS_UDINT_TO_SIGNAL (Eingang REQ).
- Dieser block liest IN.D1 und gibt nach Abschluss auf CNF ein Ereignis aus.
- Das Ereignis CNF wird an drei Stellen weitergeleitet:
  - An OUT.E1 ⟶ der Ausgangswert (FIELDBUS_UDINT_TO_SIGNAL.OUT) wird an OUT.D1 gegeben.
  - An den Takteingang CLK des Flipflops (E_D_FF).
- Gleichzeitig wird der Gültigkeitsstatus (FIELDBUS_UDINT_TO_SIGNAL.VALID) auf den Dateneingang D des Flipflops gelegt.
- Mit der steigenden Flanke von CLK übernimmt das Flipflop den Wert von D nach Q und gibt auf EO ein Ereignis aus.
- Der Flipflop-Ausgang Q speist VALID.D1, und das Ereignis EO löst VALID.E1 aus.

Somit wird das Gültigkeitssignal erst nach Abschluss der Signalverarbeitung aktualisiert und ausgegeben.

Technische Besonderheiten

- **Synchronisation der Gültigkeit:** Durch das D-Flipflop wird sichergestellt, dass der Gültigkeitstatus erst im nächsten Takt nach der Datenverarbeitung zur Verfügung steht – dies vermeidet inkonsistente Zustände.

- **Adapterbasierte Schnittstelle:** Alle Daten und Ereignisse werden über standardisierte unidirektionale Adapter ausgetauscht, was die Wiederverwendung in hierarchischen Projekten erleichtert.
- **Lizenzhinweis:** Der Baustein unterliegt der **Eclipse Public License 2.0** und enthält einen Copyright-Vermerk (HR Agrartechnik GmbH).

Zustandsübersicht

Der FB besitzt keinen eigenen Zustandsautomaten; das Verhalten wird durch das interne D-Flipflop bestimmt. Dieses kennt zwei Zustände:

Zustand	Q (Ausgang)	Bedeutung
0	`FALSE`	Signal derzeit ungültig
1	`TRUE`	Signal gültig

Der Zustandswechsel erfolgt bei jeder steigenden Taktflanke (`CLK`) auf den aktuellen Wert von `D`. Ein Zurücksetzen (Reset) ist nicht vorgesehen – bei ungültigem Signal bleibt Q auf dem letzten bekannten gültigen Wert, bis ein neuer Takt mit `D = FALSE` eintrifft.

Anwendungsszenarien

- **Feldbus-Signalaufbereitung:** Ein Sensor liefert über einen Feldbus (z. B. CANopen, PROFIBUS) einen `UDINT`-Wert samt Gültigkeitsflag. Der Baustein filtert ungültige Werte aus und stellt nur valide Daten der Steuerung zur Verfügung.
- **Datenvalidierung in sicherheitskritischen Umgebungen:** Wenn eine übergeordnete Anwendung nur verarbeitete, gültige Messwerte erhalten soll, kann dieser Baustein zwischen Bus und Logik geschaltet werden.
- **Synchronisation mehrerer paralleler Signale:** Das separate Gültigkeitssignal kann genutzt werden, um nachgeschaltete Bausteine zu takten oder Alarme auszulösen.

Vergleich mit ähnlichen Bausteinen

Baustein	Merkmal
`FIELDBUS_UDINT_TO_SIGNAL` allein	Gibt das Signal sofort ungültig weiter – ohne Synchronisation der Gültigkeit.
`AUDI_FIELDBUS_UDINT_TO_SIGNAL` (dieser FB)	**Zusätzliche Synchronisation** des Gültigkeitssignals über ein D-Flipflop, sodass `VALID` erst mit dem nächsten Takt aktualisiert wird.
Andere Validierungs-Bausteine	Oft ohne dynamische Synchronisation; dieser Baustein eignet sich besonders für zyklische Bus-Systeme, bei denen Daten und Gültigkeit zeitlich versetzt ankommen können.

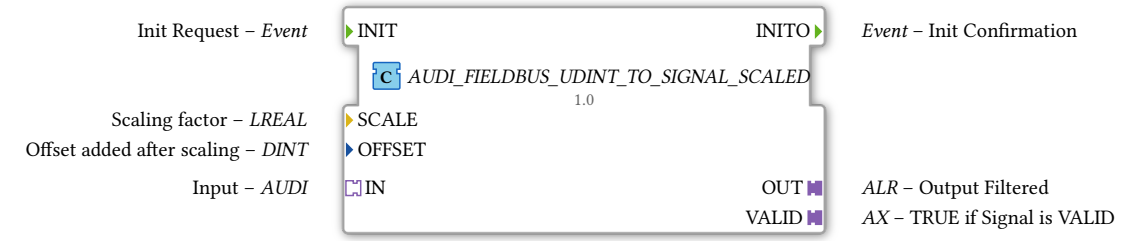
Fazit

`AUDI_FIELDBUS_UDINT_TO_SIGNAL` ist ein spezialisierter Filterbaustein für Feldbus-Signale, der eingehende `UDINT`-Werte nur bei Gültigkeit an den Ausgang weiterleitet und den Status über einen synchronisierten Pfad ausgibt. Die Verwendung eines internen Flipflops vermeidet inkonsistente Zustände und eignet sich ideal für den Einsatz in zeitkritischen Automatisierungsumgebungen. Durch die adapterbasierte Schnittstelle ist er leicht in größere 4diac-Projekte integrierbar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUDI_FIELDBUS_UDINT_TO_SIGNAL_SCALED



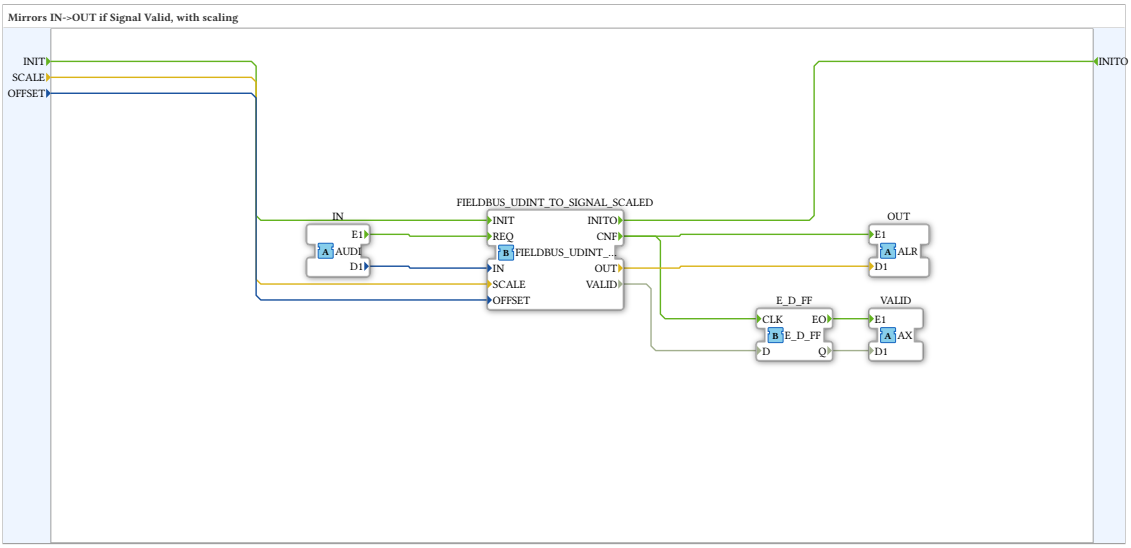


Bild des Funktionsblocks: nicht verfügbar

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUDI_FIELDBUS_UDINT_TO_SIGNAL_SCALED** dient dazu, einen empfangenen UDINT-Wert (unsigned 32-Bit Integer) aus einem Feldbusprotokoll in ein skaliertes Signal umzuwandeln und nur dann an den Ausgang weiterzuleiten, wenn das Eingangssignal als gültig markiert ist. Er stellt eine kombinierte Lösung aus Skalierung und Signalgültigkeitsprüfung dar und wird vor allem in der Automatisierungstechnik für LogiBUS-Anwendungen eingesetzt.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Initialisierungsanforderung

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Initialisierungsbestätigung

Daten-Eingänge

Name	Typ	Kommentar	Initialwert
SCALE	LREAL	Skalierungsfaktor	LREAL#1.0
OFFSET	DINT	Offset, der nach der Skalierung addiert wird	DINT#0

Daten-Ausgänge

Keine separaten Datenausgänge – die skalierten Daten werden über den Adapterausgang `OUT` bereitgestellt.

Adapter

Name	Typ	Richtung	Kommentar
IN	unidirectional::AUDI	Eingang	Feldbus-UDINT-Eingangssignal
OUT	unidirectional::ALR	Ausgang	Skaliertes Signal (falls gültig)
VALID	unidirectional::AX	Ausgang	Gültigkeitsflag (TRUE = gültig)

Funktionsweise

Der FB arbeitet intern als zusammengesetzter Baustein (Composite). Er besteht aus einem Unter-FB `FIELDBUS_UDINT_TO_SIGNAL_SCALED` (der die eigentliche Skalierung vornimmt) und einem flankengetriggerten D-Flipflop `E_D_FF`.

- Über den Adapter `IN` wird ein UDINT-Wert (über `IN.D1`) und ein Ereignis (`IN.E1`) empfangen.
- Der eingehende Wert wird an den internen FB `FIELDBUS_UDINT_TO_SIGNAL_SCALED` weitergeleitet, der den Wert mit `SCALE` multipliziert und anschließend `OFFSET` addiert. Das Ergebnis steht am Ausgang `OUT.D1` zur Verfügung.
- Gleichzeitig wird vom internen FB ein Gültigkeitssignal (`VALID`) erzeugt, das angibt, ob der empfangene Rohwert als gültig erachtet wird.
- Dieses Gültigkeitssignal wird dem D-Flipflop an seinem Dateneingang (`D`) zugeführt. Die steigende Flanke des Verarbeitungsereignisses (`CNF`) taktet das Flipflop (`CLK`).
- Am Ausgang `Q` des Flipflops liegt das verzögerungsfreie, synchronisierte Gültigkeitssignal an, das über den Adapter `VALID.D1` ausgegeben wird.
- Das Ereignis `OUT.E1` wird zeitgleich mit der Aktualisierung der Ausgabedaten ausgelöst, sodass nachgeschaltete Bausteine die neuen Werte abholen können.

Technische Besonderheiten

- Der FB ist als **Composite-FB** realisiert, d.h. seine Funktionalität wird durch ein internes Netzwerk aus anderen Bausteinen abgebildet. Dies erlaubt eine klare Trennung von Skalierung und Gültigkeitslogik.
- Die Initialisierung erfolgt über das `INIT`-Ereignis, das an den internen FB weitergeleitet wird. Nach erfolgreicher Initialisierung wird `INITO` ausgegeben.
- Der Skalierungsfaktor `SCALE` und der `OFFSET` werden bei der Initialisierung gesetzt und können zur Laufzeit nicht geändert werden, da sie nicht über separate Ereignisse aktualisiert werden.
- Die Verwendung eines D-Flipflops sorgt dafür, dass der Gültigkeitsstatus stabil bleibt, bis der nächste gültige Datenwert verarbeitet wird.
- Der Baustein ist für die Verwendung im LogiBUS-Umfeld konzipiert (Package `logiBUS::signalprocessing::fieldbus`).

Zustandsübersicht

Der FB besitzt keine expliziten Zustände im Sinne einer Zustandsmaschine. Seine interne Logik wird rein datenflussgesteuert über die Ereigniskette ausgeführt:

1. **Initialisierungsphase**: Nach Empfang von `INIT` wird der interne Baustein initialisiert und `INITO` gesendet.
2. **Datenverarbeitungsphase**: Ein eingehendes `IN.E1` löst die Skalierung und Gültigkeitsprüfung aus. Nach Abschluss werden `OUT.E1` und die Ausgabedaten aktualisiert.

Anwendungsszenarien

- **Skalierung von Feldbussignalen**: Wenn ein Feldbus einen UDINT-Wert liefert (z.B. einen Sensorzählerstand), der in eine physikalische Einheit umgerechnet werden muss, z.B. Skalierung und Offset.
- **Gültigkeitsprüfung**: Nur gültige Messwerte sollen an die Steuerung weitergegeben werden. Das Flag `VALID` kann z.B. von einem übergeordneten Sicherheitsbaustein ausgewertet werden.
- **Kopplung von LogiBUS- und IEC-61499-Systemen**: Der FB dient als Brücke zwischen dem LogiBUS-Protokoll und standardisierten Funktionsbausteinen.

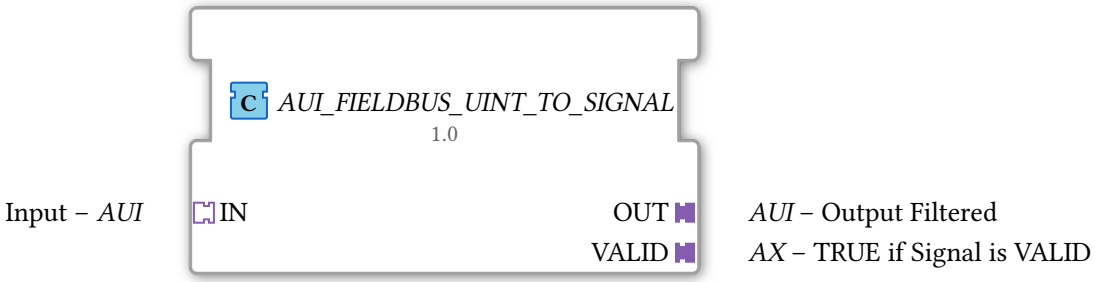
Vergleich mit ähnlichen Bausteinen

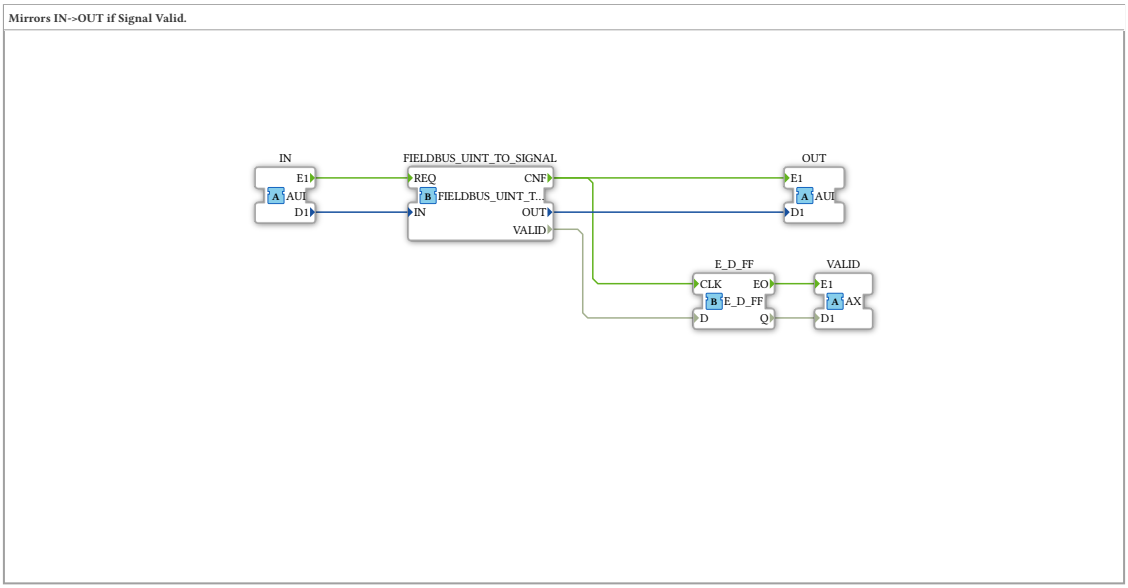
- **FIELDBUS_UDINT_TO_SIGNAL_SCALED** (nackter Skalierbaustein): Dieser Baustein führt nur die Skalierung durch, ohne die Gültigkeitsprüfung und ohne die Speicherlogik. Der vorliegende Composite-FB erweitert ihn um die Zustandsverwaltung des Gültigkeitsflags.
- **Standard IEC 61499 Skalierbausteine** (z.B. `SCALE`): Meist als einfache Arithmetik-Bausteine realisiert, fehlt oft die Integration einer Gültigkeitslogik und die Anbindung an Feldbus-Adapter.

Fazit

Der Funktionsblock `AUDI_FIELDBUS_UDINT_TO_SIGNAL_SCALED` bietet eine kompakte, wiederverwendbare Lösung zur skalierenden Signalverarbeitung mit integrierter Gültigkeitsprüfung in LogiBUS-Umgebungen. Er kombiniert Skalierung, Offset und eine flankengetriggerte Gültigkeitssperre, sodass am Ausgang nur gültige und korrekt skalierte Werte anliegen. Dank der Composite-Struktur ist er leicht erweiterbar und gut in bestehende IEC-61499-Projekte integrierbar.

AUI_FIELDBUS_UINT_TO_SIGNAL





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AUI_FIELDBUS_UINT_TO_SIGNAL** dient der Weiterleitung eines als `UINT` kodierten Feldbussignals an einen nachgeschalteten AUI-Adapter, sofern das Signal als gültig erkannt wurde. Er stellt zusätzlich einen separaten Gültigkeitsindikator (`VALID`) bereit. Der Baustein kapselt einen Datenumsetzer und ein flankengesteuertes D-Flip-Flop, das das Gültigkeitssignal bis zum nächsten Ereignis zwischenspeichert.

Schnittstellenstruktur

Ereignis-Eingänge

Der Funktionsblock besitzt keine separaten Ereignis-Eingänge. Alle ereignisgesteuerten Abläufe werden über den **Adapter-Socket `IN`** abgewickelt.

Ereignis-Ausgänge

Der Funktionsblock besitzt keine separaten Ereignis-Ausgänge. Ereignisse werden über die **Adapter-Plugs `OUT` und `VALID`** ausgegeben.

Daten-Eingänge

Der Funktionsblock besitzt keine separaten Daten-Eingänge. Der Eingangswert wird über den **Adapter-Socket `IN`** als Daten-Signal `D1` bereitgestellt.

Daten-Ausgänge

Der Funktionsblock besitzt keine separaten Daten-Ausgänge. Ausgangsdaten werden über die **Adapter-Plugs `OUT` und `VALID`** als Daten-Signale `D1` übertragen.

Adapter

Bezeichnung	Typ	Richtung	Beschreibung
`IN`	AUI (unidirectional)	Socket	Empfängt das rohe Feldbussignal als `UINT`. Stellt Ereignis `E1` und Daten `D1` bereit.
`OUT`	AUI (unidirectional)	Plug	Gibt das gefilterte Signal als `UINT` aus. Ereignis `E1` signalisiert anliegende Daten.
`VALID`	AX (unidirectional)	Plug	Liefert ein Gültigkeitssignal (`TRUE`/`FALSE`) über `D1`; Ereignis `E1` zeigt Aktualisierung an.

Funktionsweise

1. Ein externes Ereignis am **Socket `IN`** (über dessen Ereignis-Eingang `E1`) triggert den Ablauf.
2. Der eingehende Datenwert (`IN.D1`) wird durch den internen FB `FIELDBUS_UINT_TO_SIGNAL` verarbeitet. Dies erzeugt einen Ausgangswert (`OUT`) sowie ein Gültigkeitsflag (`VALID`).
3. Der verarbeitete Wert wird sofort an den **Plug `OUT`** (Daten-Signal `D1`) weitergegeben und ein Ereignis an `OUT.E1` ausgelöst.
4. Gleichzeitig wird das Gültigkeitsflag des internen FB in das **D-Flip-Flop `E_D_FF`** übernommen (getaktet durch dasselbe Ereignis).
5. Der Ausgang des Flip-Flops (`Q`) wird an den **Plug `VALID`** (Daten-Signal `D1`) gelegt; gleichzeitig wird ein Ereignis an `VALID.E1` gesendet.
6. Der Zustand des Gültigkeitssignals bleibt bis zum nächsten Verarbeitungszyklus erhalten.

Technische Besonderheiten

- Der Baustein ist als **Composite FB** realisiert; seine Funktionalität setzt sich aus zwei internen FBs zusammen:
  - `FIELDBUS_UINT_TO_SIGNAL` (Datenumsetzer)
  - `E_D_FF` (flankengesteuertes D-Flip-Flop)
- Die Gültigkeitsanzeige ist **ereignisgetaktet** und wird über ein Flip-Flop zwischengespeichert. Dadurch ist sie auch dann stabil, wenn das Eingangssignal für mehrere Zyklen ausbleibt.
- Die Signal- und Gültigkeitsausgabe erfolgen **quasi-parallel** (beide über dasselbe Ereignis des internen FB).
- Die Schnittstellen sind ausschließlich als **Adapter** definiert, was eine modulare Einbindung in Feldbussysteme erleichtert.

Zustandsübersicht

Der FB selbst besitzt keine explizite Zustandsmaschine. Sein Verhalten kann jedoch durch die interne Logik beschrieben werden:

Zustand	Beschreibung
Idle	Kein Eingangsereignis; die Ausgänge `OUT` und `VALID` behalten ihre letzten Werte.
Processing	Ein Ereignis an `IN.E1` startet die Verarbeitung.
Output	Nach Abschluss werden `OUT.D1` und `VALID.D1` aktualisiert und die Ereignisse an `OUT.E1` und `VALID.E1` gesendet.

Hold	Der Gültigkeitswert wird im Flip-Flop gehalten, bis das nächste Ereignis eintrifft.

Anwendungsszenarien

- **Feldbus-Schnittstelle**: Ein von einem Feldbus stammender `UINT`-Wert soll in ein standardisiertes AUI-Signal umgewandelt und nur bei Datenintegrität weitergereicht werden.
- **Gültigkeitsgeprüfte Weiterleitung**: Anwendungen, bei denen das Ausgangssignal erst dann als gültig betrachtet werden soll, wenn eine interne Validierung (z. B. CRC-Prüfung) erfolgreich war.
- **Einkanal-Signalaufbereitung**: Der Baustein kann in sicherheitsgerichteten Ketten eingesetzt werden, um das Ergebnis einer Plausibilitätsprüfung separat zu signalisieren.

Vergleich mit ähnlichen Bausteinen

Baustein	Unterschied / Gemeinsamkeit
`FIELDBUS_UINT_TO_SIGNAL`	Enthält nur die reine Datenumsetzung ohne Gültigkeits-Zwischenspeicher.
`AUI_SIGNAL_FILTER`	Filtert Signale, bietet aber keine explizite Gültigkeitsanzeige.
`E_D_FF`	Reines Flip-Flop ohne Datenumsetzung – hier als Hilfsbaustein genutzt.

Der vorliegende FB kombiniert die Umsetzung mit einer **ereignisgesteuerten Gültigkeitshaltung**, was ihn für sequenzielle Feldbusprotokolle besonders geeignet macht.

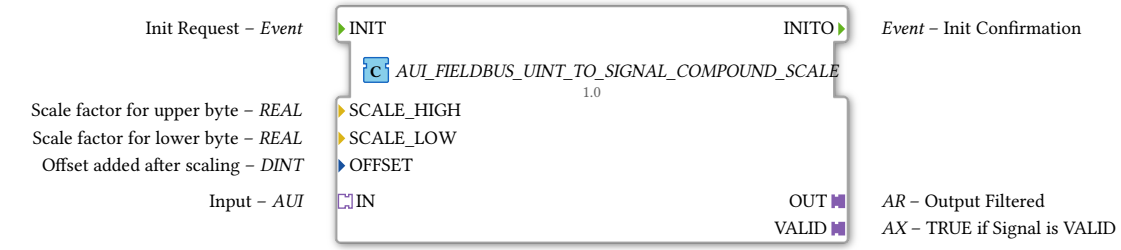
Fazit

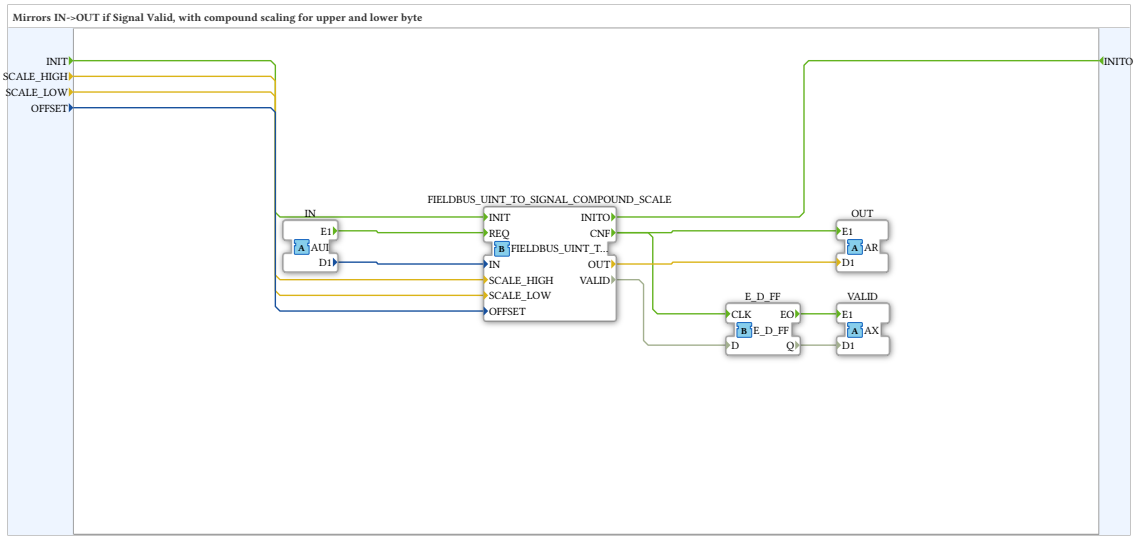
Der Funktionsblock `AUI_FIELDBUS_UINT_TO_SIGNAL` ist ein kompakter, adapterbasierter Baustein zur geprüften Signalweiterleitung in Feldbussystemen. Durch die interne Kopplung von Datenumsetzung und flankengesteuerter Gültigkeitsanzeige bietet er eine robuste und nachvollziehbare Schnittstelle für die industrielle Automatisierung. Die Verwendung von Adaptern ermöglicht eine einfache Integration in bestehende 4diac-Netzwerke.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AUI_FIELDBUS_UINT_TO_SIGNAL_COMPOUND_SCALE





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `AUI_FIELDBUS_UINT_TO_SIGNAL_COMPOUND_SCALE` dient der Umwandlung und Skalierung eines über einen AUI-Feldbusadapter eingehenden 16-Bit-Wertes in ein skalierbares Signal. Der Baustein spiegelt den Eingangswert auf den Ausgang, sofern das Gültigkeitssignal aktiv ist, und führt dabei eine zusammengesetzte Skalierung für das obere und untere Byte des Eingangswertes durch. Ein Offset kann nach der Skalierung addiert werden. Die Ausgabe erfolgt über einen unidirektionalen AR-Adapter, während die Gültigkeit über einen AX-Adapter signalisiert wird. Der Baustein ist insbesondere für industrielle Feldbusprotokolle konzipiert, bei denen Nutzdaten aus zwei getrennt skalierbaren Byte-Bereichen bestehen.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung
INIT	EInit	Initialisierungsanforderung; versetzt den Baustein in einen definierten Startzustand.

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung
INITO	EInit	Initialisierungsbestätigung; wird nach erfolgreicher Initialisierung gesendet.

Daten-Eingänge

Variable	Typ	Initialwert	Beschreibung
SCALE_HIGH	REAL	0.256	Skalierungsfaktor für das obere Byte des eingehenden Wertes.
SCALE_LOW	REAL	0.001	Skalierungsfaktor für das untere Byte des eingehenden Wertes.
OFFSET	DINT	0	Ganzzahliger Offset, der nach der Skalierung addiert wird.

Daten-Ausgänge

Der Baustein besitzt keine separaten Datenausgänge als `VarDeclaration`. Die Ausgangsdaten werden über die Adapter `OUT` und `VALID` bereitgestellt.

Adapter

Adapter	Richtung	Typ	Beschreibung
IN	Socket	adapter::types::unidirectional::AUI	Eingang für den zu verarbeitenden Feldbus-Datenwert.
OUT	Plug	adapter::types::unidirectional::AR	Ausgang des skalierten Werts (z. B. REAL oder DINT, abhängig vom internen Baustein).
VALID	Plug	adapter::types::unidirectional::AX	Ausgangssignal: TRUE, wenn der aktuelle Ausgangswert gültig ist.

Funktionsweise

Der Baustein arbeitet als Composite-Funktionsblock, der einen internen Baustein `FIELDBUS_UINT_TO_SIGNAL_COMPOUND_SCALE` in Kombination mit einem flankengesteuerten D-Flipflop (`E_D_FF`) verwendet. Die Verarbeitung läuft wie folgt ab:

1. **Eingangsereignis**: Ein Ereignis am Ereigniseingang `IN.E1` des AUI-Adapters triggert die Verarbeitung des momentanen Datenwerts `IN.D1`.
2. **Skalierung**: Der eingehende 16-Bit-Wert wird in ein oberes und ein unteres Byte aufgeteilt. Jedes Byte wird mit dem zugehörigen Skalierungsfaktor (`SCALE_HIGH`, `SCALE_LOW`) multipliziert. Das Ergebnis beider Skalierungen wird summiert und der `OFFSET` addiert. Der so berechnete Wert wird über den Datenausgang des internen Bausteins an `OUT.D1` weitergegeben.
3. **Gültigkeitssignal**: Parallel generiert der interne Baustein ein logisches Signal `VALID`, das anzeigt, ob der berechnete Wert gültig ist. Dieses Signal wird durch das D-Flipflop mit dem Abschlussereignis `CNF` getaktet und an den `VALID.D1`-Adapter weitergeleitet. Dadurch wird sichergestellt, dass das Gültigkeitssignal nur dann aktiv wird, wenn der Wert tatsächlich neu berechnet wurde.
4. **Ausgangsereignis**: Mit dem `CNF`-Ereignis des internen Bausteins wird sowohl der Ausgangsadapter `OUT` (Ereignis `E1`) als auch der `VALID`-Adapter (Ereignis `E1`) bedient.
5. **Initialisierung**: Das `INIT`-Ereignis wird direkt an den internen Baustein weitergeleitet; die Initialisierungsbestätigung `INITO` wird entsprechend zurückgegeben.

Technische Besonderheiten

- **Compound-Skalierung**: Anders als einfache Skalierungsbausteine ermöglicht dieser FB eine separate Gewichtung der beiden Byte-Hälften eines 16-Bit-Wertes. Dies ist nützlich, wenn ein Sensorwert aus zwei verschiedenen physikalischen Größen zusammengesetzt ist (z. B. ganze Zahl und Nachkommastelle).
- **Integrierte Gültigkeitslogik**: Durch das D-Flipflop wird das `VALID`-Signal flankengesteuert und nur bei tatsächlich neuen Berechnungen aktiviert. Dadurch werden unerwünschte Ausgaben bei fehlender Aktualisierung vermieden.
- **Initialisierungsdurchleitung**: Die INIT-/INITO-Ereignisse werden transparent durch den Composite-Baustein gereicht, sodass der interne Skalierungsbaustein korrekt initialisiert werden kann.

Zustandsübersicht

Der Baustein selbst besitzt keinen expliziten Zustandsautomaten auf Composite-Ebene. Die Betriebsbereitschaft wird durch das `INIT`-Ereignis hergestellt. Nach erfolgreicher Initialisierung wartet der Baustein auf eingehende Ereignisse am `IN`-Adapter und verarbeitet diese zyklisch. Solange kein INIT-Ereignis gesendet wurde, befindet sich der FB in einem undefinierten Zustand (keine Ausgabe, `VALID` inaktiv).

Anwendungsszenarien

- **Feldbusschnittstellen**: Umwandlung von analogen Sensorwerten, die als 16-Bit-Wort mit getrennten Skalierungsfaktoren für Vor- und Nachkommaanteil übertragen werden (z. B. Temperatur-/Druckwerte mit Nachkommastelle).
- **Messwertaufbereitung**: Skalierung von digitalen Eingangssignalen aus Feldbus-Controllern, die Rohdaten in zwei Byte-Bereichen liefern.
- **Signalvalidierung**: Einsatz in sicherheitsgerichteten Anwendungen, bei denen nur gültige und korrekt berechnete Werte an die nachfolgende Steuerung weitergegeben werden dürfen.

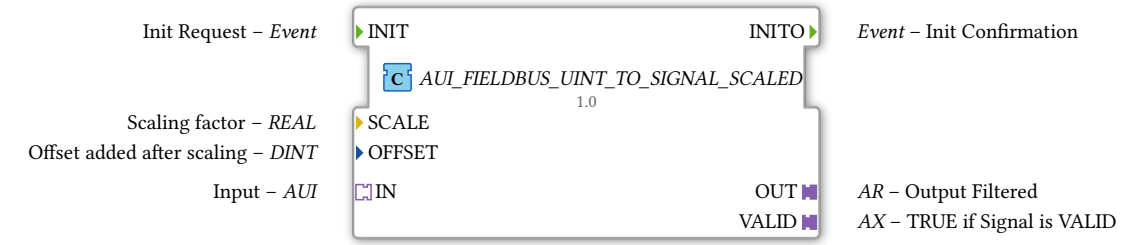
Vergleich mit ähnlichen Bausteinen

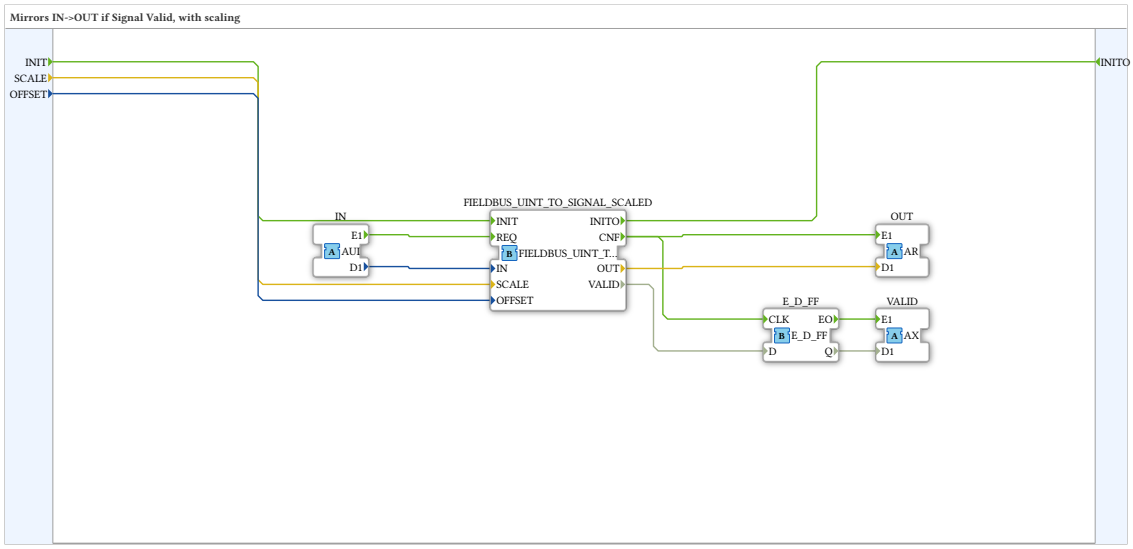
Im Gegensatz zu einfachen Skalierungsblöcken (z. B. `SCALE` oder `UINT_TO_REAL`) bietet dieser Baustein eine differenzierte Verarbeitung der beiden Byte-Hälften. Er ist speziell für Datenformate ausgelegt, bei denen das obere und untere Byte unterschiedliche physikalische Einheiten oder Skalierungsbereiche repräsentieren. Alternative Bausteine müssten diese Logik manuell durch Aufteilung, separate Skalierung und anschließende Addition abbilden, was den Implementierungsaufwand erhöht.

Fazit

Der `AUI_FIELDBUS_UINT_TO_SIGNAL_COMPOUND_SCALE` ist ein spezialisierter, aber flexibler Funktionsblock für die Aufbereitung von Feldbussignalen. Seine Fähigkeit, ein eingehendes 16-Bit-Wort mit zwei unterschiedlichen Skalierungsfaktoren zu verarbeiten und die Gültigkeit des Ausgangssignals zu steuern, macht ihn zu einem wertvollen Baustein in IEC-61499-basierten Automatisierungssystemen, insbesondere wenn es um die Integration von Geräten mit compoundierten Datenformaten geht.

AUI_FIELDBUS_UINT_TO_SIGNAL_SCALED





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock AUI_FIELDBUS_UINT_TO_SIGNAL_SCALED dient der skalieren Übertragung eines eingehenden UINT-Wertes (Feldbus-Signal) auf einen analogen REAL-Ausgang. Er arbeitet nur dann, wenn das eingehende Signal als gültig markiert ist (VALID-Signal). Eine zusätzliche Initialisierungsschnittstelle erlaubt das Zurücksetzen der internen Logik. Der Baustein ist als Composite ausgeführt und kombiniert einen skalierenden Unterbaustein mit einem D-Flipflop zur Gültigkeitsspeicherung.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Initialisierungsanforderung

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Initialisierungsbestätigung

Daten-Eingänge

Name	Typ	Initialwert	Kommentar
SCALE	REAL	REAL#1.0	Skalierungsfaktor
OFFSET	DINT	DINT#0	Offset nach der Skalierung

Daten-Ausgänge

(Keine direkten Datenausgänge – Ausgaben erfolgen über Adapter)

Adapter

Typ	Name	Richtung	Kommentar
adapter::types::unidirectional::AUI	IN	Socket	UINT-Eingang (Feldbus-Signal)
adapter::types::unidirectional::AR	OUT	Plug	Skalierter REAL-Ausgang
adapter::types::unidirectional::AX	VALID	Plug	Gültigkeitssignal (TRUE = gültig)

Funktionsweise

Der Baustein besteht intern aus zwei Sub-FBs: dem skalierenden Kernbaustein `logiBUS::signalprocessing::fieldbus::FIELDBUS_UINT_TO_SIGNAL_SCALED` und einem D-Flipflop `E_D_FF`.

Die Initialisierung erfolgt über den Ereignis-Eingang INIT, der direkt an den internen Kernbaustein weitergeleitet wird. Die Bestätigung INITO wird nach erfolgreicher Initialisierung zurückgegeben.

Die eigentliche Signalverarbeitung wird durch ein eingehendes Ereignis (E1) des IN-Adapters ausgelöst. Dieses Ereignis triggert den Kernbaustein, der den über IN.D1 anliegenden UINT-Wert wie folgt verarbeitet:

- **Skalierung**: `Wert_skaliert = (REAL)IN_D1 * SCALE`
- **Offset**: `Wert_skaliert = Wert_skaliert + OFFSET`

Das Ergebnis (REAL) wird als OUT.D1 ausgegeben und gleichzeitig ein Ereignis OUT.E1 erzeugt. Parallel wird das vom Kernbaustein berechnete VALID-Signal (TRUE = gültig, FALSE = ungültig) an den D-Eingang des D-Flipflops gelegt. Das gleiche Ereignis CNF des Kernbausteins dient als Takt für das Flipflop. Dadurch wird der Gültigkeitszustand übernommen und am Q-Ausgang dauerhaft bis zum nächsten Takt gehalten. Der finale Gültigkeitswert wird über den VALID-Adapter (D1) und dessen Ereignis E1 ausgegeben.

Damit wird das VALID-Signal erst nach Abschluss der Skalierung aktualisiert und bleibt solange stabil, bis ein neuer Wert am Eingang eintrifft.

Technische Besonderheiten

- **Composite-Architektur**: Der FB ist komplett aus Sub-FBs aufgebaut, was eine einfache Anpassung und Wiederverwendung ermöglicht.
- **Gültigkeitsspeicherung**: Ein D-Flipflop speichert das Gültigkeitssignal zwischen den Verarbeitungszyklen. So kann der Ausgang auch dann einen definierten Zustand anzeigen, wenn keine neuen Daten anliegen.
- **Skalierung und Offset**: Die Skalierung erfolgt mit einem REAL-Faktor und einem DINT-Offset. Dies erlaubt eine flexible Anpassung an physikalische Einheiten (z. B. Umrechnung von Digitalwerten in Druck oder Temperatur).
- **Adapterbasierte Ein-/Ausgabe**: Alle Signale werden über standardisierte 61499-Adapter (AUI, AR, AX) ausgetauscht, was die Integration in modulare Feldbussysteme erleichtert.

Zustandsübersicht

Der Funktionsblock selbst besitzt keinen eigenen Zustandsautomaten. Der interne Zustand wird durch das D-Flipflop repräsentiert:

- **Zustand Q = FALSE**: Letzter verarbeiteter Wert war ungültig oder noch kein Wert eingetroffen.
- **Zustand Q = TRUE**: Der letzte verarbeitete Wert war gültig und liegt skaliert am OUT-Ausgang an.

Der Zustandswechsel erfolgt mit jedem neuen Ereignis auf dem IN-Adapter (E1), wenn der Kernbaustein ein neues VALID-Signal liefert.

Anwendungsszenarien

- **Landwirtschaftliche Sensorik**: Übertragung skalierter Messwerte (z. B. Füllstand, Druck) von Feldbussensoren mit Gültigkeitsflag.
- **Datenvorverarbeitung**: Anpassung von Rohwerten aus Digitalwandlern (z. B. 0–10 V ⟶ REAL-Werte) mit automatischer Gültigkeitsprüfung.
- **Qualitätssicherung**: Nur Werte, die als gültig markiert sind, werden an nachfolgende Steuerlogiken weitergegeben.

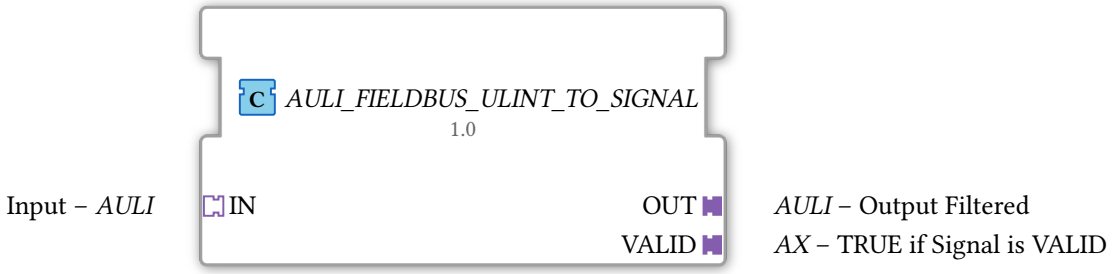
Vergleich mit ähnlichen Bausteinen

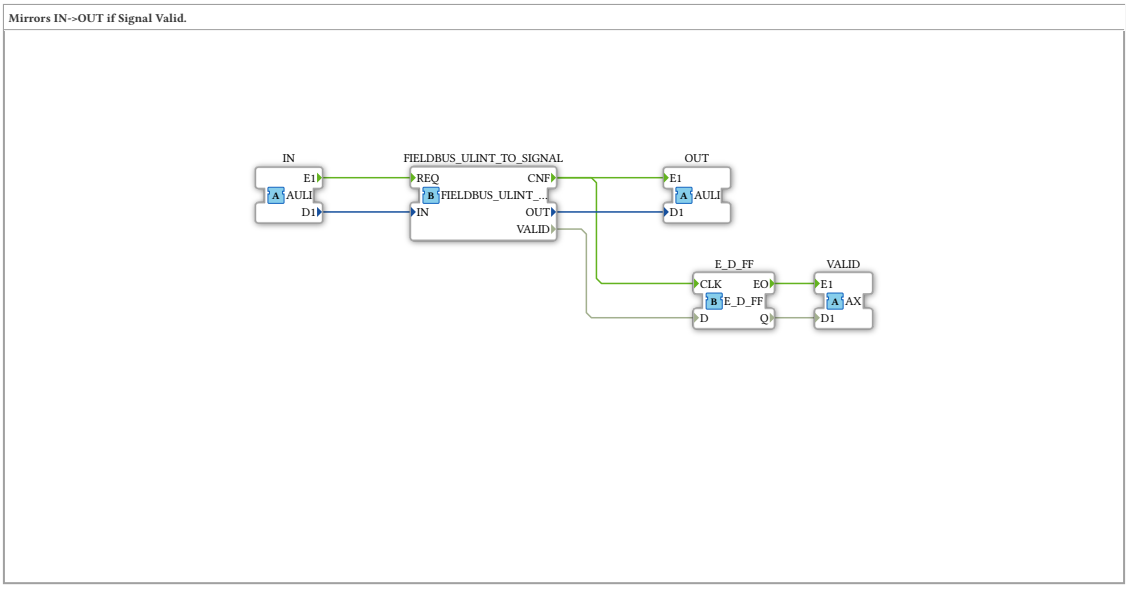
- **AUI_FIELDBUS_UINT_TO_SIGNAL** (ohne Gültigkeitsflag): Gibt immer einen skalierten Wert aus, unabhängig von der Signalqualität.
- **AUI_FIELDBUS_UINT_TO_SIGNAL_SCALED** (Core): Liefert bereits eine skalierte Ausgabe mit Gültigkeitsflag, jedoch ohne Speicherung. Der vorliegende Composite ergänzt diese Speicherung und erlaubt eine stabilere Ausgabe zwischen den Ereignissen.
- **Adapterbasierte Filter**: Andere Bausteine nutzen ähnliche Prinzipien, aber mit spezifischen Gültigkeitsbedingungen (z. B. Zeitfilter).

Fazit

Der AUI_FIELDBUS_UINT_TO_SIGNAL_SCALED bietet eine zuverlässige und flexible Lösung zur skalierten Übertragung von UINT-Feldbussignalen mit expliziter Gültigkeitskennzeichnung. Die Kombination aus Skalierung, Offset und gespeichertem Gültigkeitssignal macht ihn besonders geeignet für sicherheitskritische oder qualitätsüberwachte Anwendungen in der Automatisierungstechnik.

AULI_FIELDBUS_ULINT_TO_SIGNAL





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AULI_FIELDBUS_ULINT_TO_SIGNAL** dient als zusammengesetzter Baustein zur Verarbeitung eines ULINT-Wertes aus einem Feldbus. Er spiegelt den eingehenden Wert auf den Ausgang, sofern das Signal als gültig („Valid“) erkannt wird, und gibt zusätzlich ein synchronisiertes Gültigkeitssignal aus. Der Baustein kombiniert einen internen Feldbus-Konverter mit einem D-Flipflop, um eine getaktete, zuverlässige Signalweitergabe zu gewährleisten.

Schnittstellenstruktur

Der Baustein besitzt keine klassischen Ereignis- oder Datenports auf oberster Ebene, sondern kommuniziert ausschließlich über **Adapter**. Diese sind als **Sockets** (Eingänge) und **Plugs** (Ausgänge) realisiert.

Ereignis-Eingänge

Port	Bezeichnung	Beschreibung
`IN.E1`	Ereignis-Eingang (über Socket IN)	Löst die Verarbeitung eines neuen ULINT-Wertes aus.

Ereignis-Ausgänge

Port	Bezeichnung	Beschreibung
`OUT.E1`	Ereignis-Ausgang (über Plug OUT)	Signalisiert die Fertigstellung der Ausgabe des gespiegelten Wertes.
`VALID.E1`	Ereignis-Ausgang (über Plug VALID)	Zeigt an, dass das Gültigkeitssignal aktualisiert wurde.

Daten-Eingänge

Port	Bezeichnung	Datentyp	Beschreibung
`IN.D1`	Daten-Eingang (über Socket IN)	ULINT	Der zu verarbeitende Feldbus-Wert.

Daten-Ausgänge

Port	Bezeichnung	Datentyp	Beschreibung
`OUT.D1`	Daten-Ausgang (über Plug OUT)	ULINT	Der gespiegelte Eingangswert (wenn gültig).

VALID.D1	Daten-Ausgang (über Plug VALID)	BOOL	Gültigkeitssignal (TRUE = Wert ist gültig).

Adapter

Adapter	Richtung	Typ	Beschreibung
IN	Socket (Eingang)	adapter::types::unidirectional::AULI	Nimmt den ULINT-Wert und das zugehörige Ereignis entgegen.
OUT	Plug (Ausgang)	adapter::types::unidirectional::AULI	Gibt den verarbeiteten ULINT-Wert und ein Ereignis aus.
VALID	Plug (Ausgang)	adapter::types::unidirectional::AX	Stellt das Gültigkeitssignal (BOOL) und ein Ereignis bereit.

Funktionsweise

Der Baustein arbeitet intern mit zwei Subbausteinen:

1. **FIELDBUS_ULINT_TO_SIGNAL** – Ein Feldbus-Konverter, der einen ULINT-Wert in ein Signal umwandelt und parallel ein Gültigkeitssignal (BOOL) erzeugt.
2. **E_D_FF** – Ein flankengesteuertes D-Flipflop, das das Gültigkeitssignal synchron zum Ereignis des Konverters übernimmt.

Ablauf:

1. Ein Ereignis am Eingang `IN.E1` startet die Verarbeitung.
2. Der Eingangswert `IN.D1` wird an den internen Konverter übergeben.
3. Der Konverter berechnet den Ausgangswert `OUT` und das `VALID`-Signal.
4. Das Ergebnis `OUT` wird sofort auf den Ausgangsadapter `OUT.E1` / `OUT.D1` gespiegelt.
5. Gleichzeitig wird das `VALID`-Signal an das Flipflop weitergeleitet und durch das Ereignis `CNF` getaktet.
6. Der Flipflop-Ausgang `Q` wird auf den Adapter `VALID.D1` ausgegeben und ein Ereignis `VALID.E1` erzeugt.

Dadurch wird sichergestellt, dass der Ausgangswert und das Gültigkeitssignal stets konsistent und synchron zueinander aktualisiert werden.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Baustein verwendet standardisierte unidirektionale Adapter (`AULI`, `AX`), was eine lose Kopplung und Wiederverwendbarkeit ermöglicht.
- **Synchronisation durch Flipflop**: Das Gültigkeitssignal wird flankengesteuert übernommen, um Takt- und Glitch-Probleme zu vermeiden.
- **Komposit-Struktur**: Die innere Logik ist als Netzwerk aus zwei Standardbausteinen realisiert, was die Wartbarkeit und Transparenz erhöht.
- **Keine direkten Ereignis-/Datenports**: Die gesamte Kommunikation erfolgt über Adapter, was die Integration in Adapter-basierte Architekturen erleichtert.

Zustandsübersicht

Der Baustein selbst ist zustandslos (rein kombinatorisch mit getakteter Übernahme). Das interne Flipflop `E_D_FF` besitzt zwei Zustände:

- **Q = FALSE**: Kein gültiges Signal (initial oder nach einem ungültigen Wert).
- **Q = TRUE**: Gültiges Signal liegt an.

Ein Zustandsdiagramm für den gesamten FB ist nicht erforderlich, da das Verhalten vollständig durch die Ereignis- und Datenflüsse beschrieben wird.

Anwendungsszenarien

- **Feldbus-Signalaufbereitung**: Ein ULINT-Wert (z. B. Zählerstand, Positionswert) wird von einem Feldbus übernommen, auf Gültigkeit geprüft und an eine Steuerung oder Visualisierung weitergegeben.
- **Qualitätskennzeichnung**: Zusammen mit dem Datenwert wird ein Bool-Signal bereitgestellt, das die Gültigkeit anzeigt (z. B. „Daten gültig“/„Daten fehlerhaft“).
- **Sicherheitsgerichtete Übertragung**: Durch die getaktete Übernahme des Gültigkeitssignals können Fehler in der Datenleitung erkannt und signalisiert werden.

Vergleich mit ähnlichen Bausteinen

- **Einfacher Durchschleif-Baustein**: Ein einfacher `MOVE`- oder `REPEAT`-Baustein leitet den Wert ungeprüft weiter. `AULI_FIELDBUS_ULINT_TO_SIGNAL` fügt eine explizite Gültigkeitsprüfung und -signalisation hinzu.
- **FIELDBUS_ULINT_TO_SIGNAL (einzeln)**: Dieser Kernbaustein liefert bereits das Gültigkeitssignal, jedoch ohne flankengesteuerte Synchronisation. Der vorliegende Kompositbaustein ergänzt die Signalqualität durch das Flipflop.
- **Adapter-basierte Bausteine**: Speziell für den Einsatz in 4diac-Umgebungen, die auf Adapter-Kommunikation setzen; andere Bausteine mit direkten Ports sind weniger flexibel.

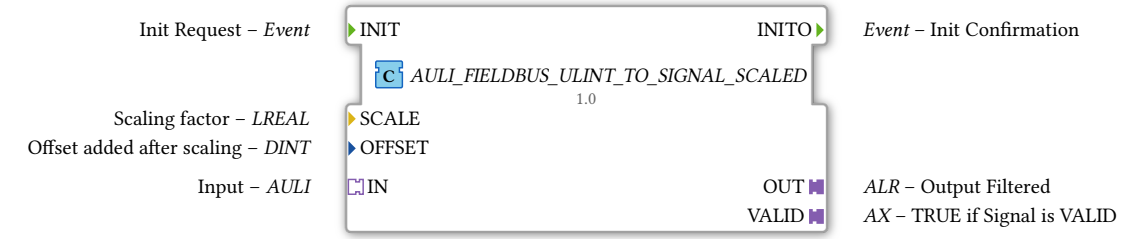
Fazit

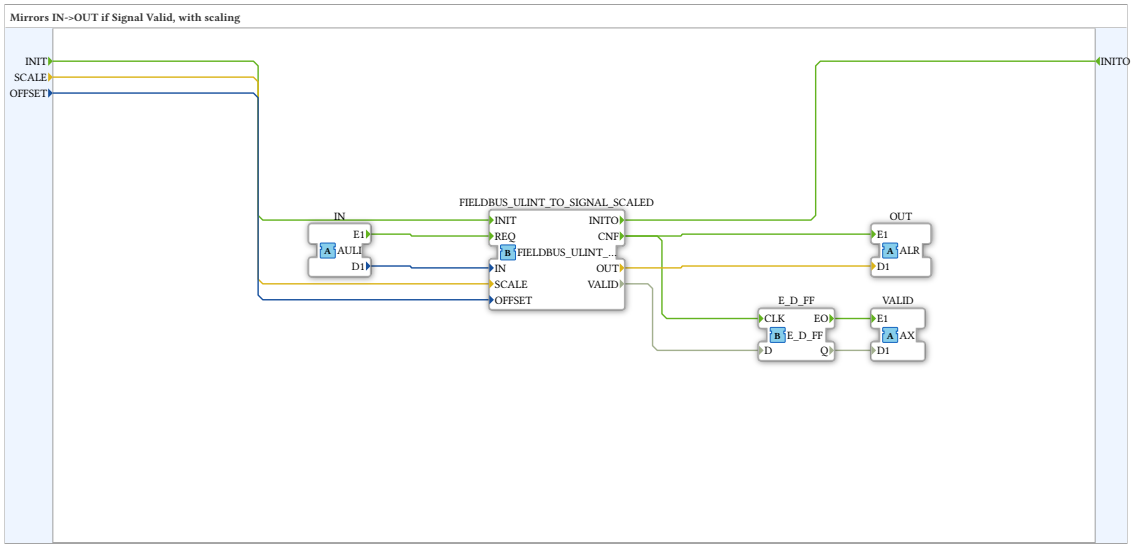
`AULI_FIELDBUS_ULINT_TO_SIGNAL` ist ein robuster, adapterbasierter Funktionsblock zur sicheren Weitergabe eines ULINT-Feldbussignals mit Gültigkeitsanzeige. Die Kombination aus Konverter und Flipflop gewährleistet eine synchronisierte und zuverlässige Signalausgabe. Durch seine klare Schnittstellenstruktur eignet er sich besonders für modulare Automatisierungslösungen, die auf lose gekoppelte Adapter setzen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AULI_FIELDBUS_ULINT_TO_SIGNAL_SCALED





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **AULI_FIELDBUS_ULINT_TO_SIGNAL_SCALED** dient der skalierenden Spiegelung eines vorzeichenlosen Ganzzahlwertes (ULINT) in einen skalierten Signalwert (LREAL) unter Berücksichtigung einer Gültigkeitsinformation. Er ist als Composite-Baustein realisiert und kombiniert die eigentliche Skalierungslogik mit einer synchronisierten Valid-Ausgabe. Der Baustein ist für den Einsatz in Feldbus-Umgebungen konzipiert, wo Rohdaten (z. B. Sensorwerte) mit einem linearen Faktor und Offset umgerechnet werden müssen und die Gültigkeit des Signals zuverlässig weitergereicht werden soll.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Init Request

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Init Confirmation

Daten-Eingänge

Name	Typ	Initialwert	Kommentar
SCALE	LREAL	1.0	Skalierungsfaktor
OFFSET	DINT	0	Offset, der nach der Skalierung addiert wird

Daten-Ausgänge

Keine expliziten Datenausgänge an der FB-Schnittstelle vorhanden – die Ausgabe erfolgt über die Adapter.

Adapter

Socket (Eingang)

Name	Typ	Kommentar
IN	adapter::types::unidirectional::AULI	Input

Plug (Ausgang)

Name	Typ	Kommentar
OUT	adapter::types::unidirectional::ALR	Output Filtered
VALID	adapter::types::unidirectional::AX	TRUE if Signal is VALID

Funktionsweise

Der Baustein arbeitet ereignisgesteuert. Nach einem INIT-Ereignis (am Eingang `INIT`) wird der interne Skalierungsbaustein initialisiert und quittiert mit `INITO`. Die eigentliche Verarbeitung erfolgt bei jedem Ereignis, das über den Adapter `IN.E1` ankommt:

1. **Datenübernahme:** Der von `IN.D1` bereitgestellte ULINT-Wert wird an den internen Funktionsblock `FIELDBUS_ULINT_TO_SIGNAL_SCALED` weitergeleitet.
2. **Skalierung:** Der interne Block wendet den Skalierungsfaktor `SCALE` (LREAL) und den Offset `OFFSET` (DINT) auf den Rohwert an und erzeugt einen skalierten LREAL-Wert sowie ein Gültigkeitssignal (`VALID`).
3. **Synchronisation:** Das Gültigkeitssignal wird über ein flankengetriggertes D-Flipflop (`E_D_FF`) synchronisiert. Die Taktflanke wird durch das `CNF`-Ereignis des internen Blocks bereitgestellt.
4. **Ausgabe:** Der skalierte Wert wird über `OUT.D1` und das zugehörige Ereignis `OUT.E1` an den Adapter `OUT` gesendet. Das synchronisierte Gültigkeitssignal wird über `VALID.D1` und das Ereignis `VALID.E1` an den Adapter `VALID` ausgegeben.

Der interne Skalierungsbaustein ist derselbe Typ wie der äußere, was bedeutet, dass die Skalierungslogik rekursiv aufgebaut sein könnte, jedoch im vorliegenden Entwurf die innere Instanz die eigentliche Rechenlogik enthält. Die äußere Hülle fügt die Synchronisation der Gültigkeitsinformation hinzu und stellt die Adapter-Schnittstellen bereit.

Technische Besonderheiten

- **Adaptierte Ein-/Ausgänge:** Der Baustein verwendet ausschließlich Adapter für die Datenübertragung. Dies ermöglicht eine lose Kopplung in Feldbus- und Komponentennetzwerken.
- **Synchronisation der Gültigkeit:** Die Valid-Information wird über ein D-Flipflop getaktet, sodass sie zeitlich mit dem ausgegebenen Signalwert übereinstimmt.
- **Skalierungsparameter:** `SCALE` und `OFFSET` sind als Eingangsvariablen direkt zugänglich und können zur Laufzeit geändert werden. Initialwerte erlauben einen sofortigen Betrieb ohne Konfiguration.

Zustandsübersicht

Der Baustein besitzt keine explizite Zustandsmaschine. Sein Verhalten wird durch die Ereignissteuerung bestimmt:

- **Initialisiert:** Nach erfolgreichem `INIT` ist der interne Skalierungsblock bereit.
- **Bereit:** Auf ein Ereignis auf `IN.E1` wird die Datenverarbeitung ausgelöst. Nach Abschluss werden `OUT.E1` und `VALID.E1` ausgelöst.
- **Fehlerfall:** Ein fehlerhafter Skalierungsvorgang (z. B. Überlauf) wird durch den internen Block behandelt; das Gültigkeitssignal würde in diesem Fall `FALSE` werden.

Anwendungsszenarien

- **Feldbus-Sensorauswertung:** Ein Sensor liefert einen ULINT-Rohwert über einen Bus (z. B. CANopen, PROFIBUS). Der Baustein skaliert den Wert in eine physikalische Einheit (z. B. Temperatur, Druck) und gibt ihn als LREAL an die Steuerung weiter, zusammen mit einem Gültigkeitsflag.
- **Signalaufbereitung in landwirtschaftlichen Maschinen** (wie im Copyright-Hinweis angedeutet): Umwandlung von Drehzahl-, Füllstands- oder Geschwindigkeitsdaten in normierte Signale für eine übergeordnete Steuerung.
- **Qualitätsgesicherte Datenweitergabe:** Wenn die Gültigkeit eines Signals (z. B. Sensorfehler, Kommunikationsabbruch) explizit mitgeführt werden muss, sorgt die synchronisierte Valid-Ausgabe für konsistente Signale.

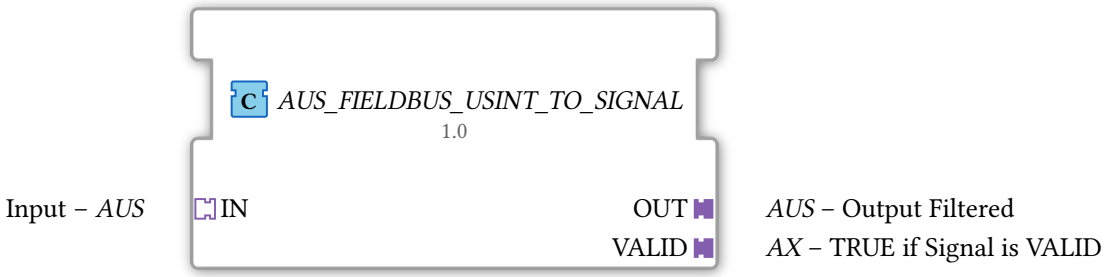
Vergleich mit ähnlichen Bausteinen

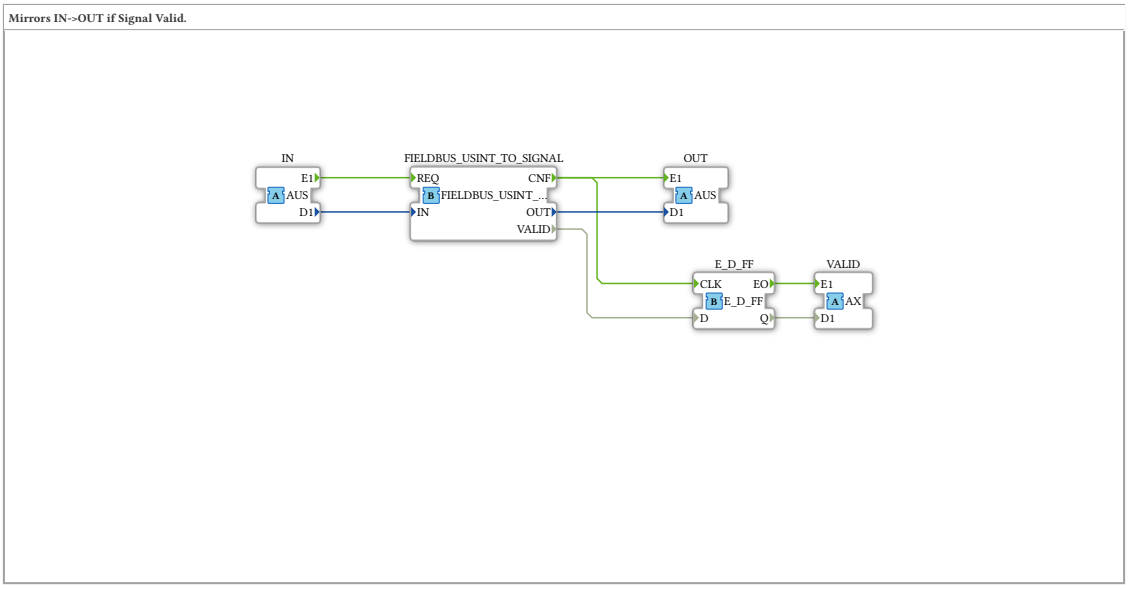
- **FIELDBUS_ULINT_TO_SIGNAL** (ohne Scaled): Ein direkter Durchgang ohne Skalierung – hier ist die Skalierung integriert.
- **Standard-Skalierungsbausteine** (z. B. aus IEC-61131-3): Diese arbeiten oft mit einfachen Ein-/Ausgängen und ohne Adapter oder Gültigkeitssynchronisation.
- **FB mit integrierter Validierung:** Viele Feldbus-FB geben nur einen Datenwert aus. Dieser Baustein hebt sich durch die separate, getaktete Gültigkeitsausgabe ab.

Fazit

Der **AULI_FIELDBUS_ULINT_TO_SIGNAL_SCALED** ist ein spezialisierter Composite-FB für die skalierte Umwandlung von ULINT-Feldbusdaten in LREAL-Signale mit synchronisierter Gültigkeitsinformation. Durch die Verwendung von Adaptern und die klare Trennung von Skalierung und Synchronisation eignet er sich besonders für modulare, sicherheitskritische Anwendungen in der Automatisierungs- und Agrartechnik. Die Parametrierung über `SCALE` und `OFFSET` macht ihn flexibel einsetzbar.

AUS_FIELDBUS_USINT_TO_SIGNAL





• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock `AUS_FIELDBUS_USINT_TO_SIGNAL` dient der Weitergabe eines USINT-Signals von einem Feldbus-Adapter unter Berücksichtigung einer Gültigkeitsprüfung. Er spiegelt den Eingangswert auf den Ausgang, sofern das Signal als gültig erkannt wird. Ein zusätzlicher Adapter gibt den Gültigkeitsstatus aus. Die interne Logik verwendet einen dedizierten Fieldbus-Umwandlungsbaustein und ein flankengesteuertes D-Flip-Flop zur Synchronisation.

Schnittstellenstruktur

Ereignis-Eingänge

Bezeichnung	Beschreibung
`IN.E1`	Über den Adapter `IN` (Socket) eingehendes Ereignis, das die Verarbeitung anstößt.

Ereignis-Ausgänge

Bezeichnung	Beschreibung
`OUT.E1`	Über den Adapter `OUT` (Plug) ausgegebenes Ereignis nach erfolgreicher Spiegelung des gültigen Signals.
`VALID.E1`	Über den Adapter `VALID` (Plug) ausgegebenes Ereignis nach Aktualisierung des Gültigkeitsstatus.

Daten-Eingänge

Bezeichnung	Typ	Beschreibung
`IN.D1`	USINT (implizit)	Eingangsdatenwert, der auf Gültigkeit geprüft und gespiegelt wird.

Daten-Ausgänge

Bezeichnung	Typ	Beschreibung
`OUT.D1`	(analog zu IN)	Ausgangsdatenwert, der bei gültigem Signal den Eingangswert widerspiegelt.

VALID.D1	BOOL	Gültigkeitsflag: TRUE wenn das Eingangssignal als gültig erkannt wurde, sonst FALSE.

Adapter

Name	Typ	Richtung	Beschreibung
IN	adapter::types::unidirectional::AUS	Socket	Empfängt das Rohsignal (Ereignis + Daten) vom Feldbus.
OUT	adapter::types::unidirectional::AUS	Plug	Gibt das validierte Signal an nachgelagerte Komponenten weiter.
VALID	adapter::types::unidirectional::AX	Plug	Stellt den Gültigkeitsstatus (Ereignis + BOOL) zur Verfügung.

Funktionsweise

Der FB reagiert auf ein eingehendes Ereignis IN.E1. Dieses triggert den internen Baustein FIELDBUS_USINT_TO_SIGNAL, der den Datenwert IN.D1 verarbeitet und zwei Ergebnisse liefert:

- Einen Ausgangswert (OUT)
- Ein Gültigkeitssignal (VALID)

Der Ausgangswert wird direkt auf OUT.D1 durchgeschaltet und über OUT.E1 quittiert. Gleichzeitig wird das Gültigkeitssignal dem Daten-Eingang D des D-Flip-Flops E_D_FF zugeführt. Das Flip-Flop wird durch dasselbe Ereignis (über CFN) getaktet, sodass der Gültigkeitsstatus stabil auf Q gehalten und auf VALID.D1 ausgegeben wird. Erst nach dem Taktimpuls wird VALID.E1 ausgelöst.

Zusammengefasst: Der Ausgang OUT spiegelt den Eingang IN, sofern das Signal im Feldbus-Baustein als gültig erkannt wird. Der Gültigkeitsstatus wird flankengetriggert gespeichert.

Technische Besonderheiten

- **Adapterbasierte Kommunikation:** Alle Ein- und Ausgänge sind über standardisierte unidirektionale Adapter (AUS, AX) realisiert, die eine klare Trennung zwischen Ereignis- und Datenpfaden ermöglichen.
- **Flankengesteuerte Gültigkeitsspeicherung:** Ein D-Flip-Flop sorgt dafür, dass der Gültigkeitszustand nur bei einem Ereignis (steigende Flanke) übernommen wird – dies verhindert Glitches und asynchrone Zustandswechsel.
- **Wiederverwendete interne FB:** Der Baustein FIELDBUS_USINT_TO_SIGNAL übernimmt die eigentliche Feldbus-spezifische Konvertierung und Gültigkeitsprüfung; der äußere FB dient als reine Hüll- und Synchronisationslogik.

Zustandsübersicht

Der interne Zustand wird durch das D-Flip-Flop E_D_FF definiert. Es besitzt zwei Zustände:

Zustand	Beschreibung
Q = FALSE (Initial)	Ausgang VALID.D1 zeigt „ungültig“ an. Nach jedem Ereignis wird der aktuelle VALID-Wert (D) übernommen.
Q = TRUE	Ausgang VALID.D1 zeigt „gültig“ an. Der Zustand bleibt bis zum nächsten Ereignis stabil.

Zustandsübergänge erfolgen ausschließlich bei jeder steigenden Flanke des Taktsignals (Ereignis CNF des Fieldbus-Bausteins).

Anwendungsszenarien

- **Feldbus-Signalaufbereitung:** Ein Sensor sendet USINT-Werte über einen Feldbus; der FB filtert ungültige Werte heraus und gibt nur valide Daten sowie einen klaren Gültigkeitszustand weiter.
- **Sicherheitsgerichtete Datenweitergabe:** In Steuerungen, die auf gültige Signale angewiesen sind (z. B. in der Agrartechnik), kann dieser FB als einfache Validierungsstufe eingesetzt werden.
- **Schnittstellenanpassung:** Der Baustein verbindet einen feldbusspezifischen Adapter (z. B. CANopen, PROFIBUS) mit einem einheitlichen Logik-Adapter, der nur bei gültigen Daten Ereignisse auslöst.

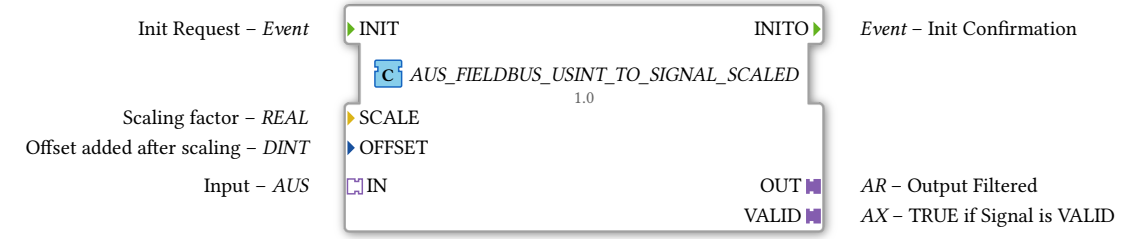
Vergleich mit ähnlichen Bausteinen

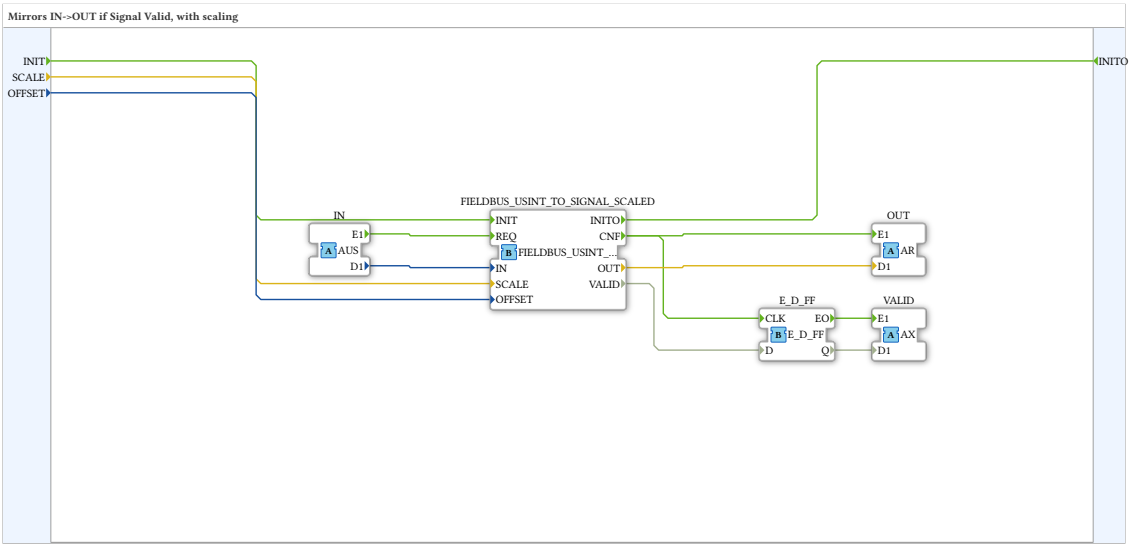
- **Einfache Mirror-FBs** (z. B. `MOVE` oder `AUS_MIRROR`): Diese geben das Signal ohne Gültigkeitsprüfung weiter. Der vorliegende FB bietet zusätzlich eine Validierungslogik und einen separaten Gültigkeitsausgang.
- **Fieldbus-Konverter ohne Speicher:** Manche Bausteine geben das Gültigkeitssignal direkt und ungetaktet aus. Der Einsatz eines Flip-Flops hier vermeidet metastabile Zustände und sorgt für deterministische Ausgaben.

Fazit

Der `AUS_FIELDBUS_USINT_TO_SIGNAL` ist ein nützlicher Funktionsblock für die saubere, validierte Weiterleitung von USINT-Signalen aus Feldbussystemen. Durch die Kombination aus einem spezialisierten Fieldbus-Baustein und einem flankengesteuerten Flip-Flop wird eine zuverlässige Trennung von gültigen und ungültigen Daten erreicht. Die adapterbasierte Schnittstelle erleichtert die Integration in modulare Steuerungsarchitekturen.

AUS_FIELDBUS_USINT_TO_SIGNAL_SCALED





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock AUS_FIELDBUS_USINT_TO_SIGNAL_SCALED spiegelt einen eingehenden Feldbussignalwert (USINT) auf den Ausgang, sofern das Signal als gültig erkannt wird. Dabei wird der Eingangswert mit einem konfigurierbaren Skalierungsfaktor multipliziert und ein Offset addiert. Der FB ist als Composite-Baustein realisiert und besteht aus einem Sub-FB für die Skalierung sowie einem flankengesteuerten D-Flipflop zur Synchronisation des Gültigkeitssignals. Er wird typischerweise in der Feldbus-Signalverarbeitung eingesetzt, um rohe USINT-Werte in physikalische Größen umzurechnen und die Datenvalidität weiterzuleiten.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsanforderung. Wird durchgereicht an das interne Sub-FB.

Ereignis-Ausgänge

- **INITO** (EInit): Initialisierungsbestätigung. Wird vom internen Sub-FB übernommen.

Daten-Eingänge

- **SCALE** (REAL): Skalierungsfaktor, der auf den Eingangswert multipliziert wird. Standardwert: 1.0.
- **OFFSET** (DINT): Offsetwert, der nach der Multiplikation zum skalierten Wert addiert wird. Standardwert: 0.

Daten-Ausgänge

Keine direkten Datenausgänge – die Ausgabe erfolgt ausschließlich über die Adapter.

Adapter

- **IN** (Socket, Typ `adapter::types::unidirectional::AUS`): Eingangsadapter für das Rohsignal. Liefert die USINT-Daten über `D1` und ein Ereignis über `E1`.
- **OUT** (Plug, Typ `adapter::types::unidirectional::AR`): Ausgangsadapter für das skalierte Signal. Stellt den skalierten Wert über `D1` und ein Ereignis über `E1` bereit.
- **VALID** (Plug, Typ `adapter::types::unidirectional::AX`): Ausgangsadapter für das Gültigkeitssignal. Gibt einen booleschen Wert (TRUE = gültig) über `D1` und ein Ereignis über `E1` aus.

Funktionsweise

1. **Initialisierung**: Das Ereignis `INIT` wird direkt an das interne Sub-FB `FIELDBUS_USINT_TO_SIGNAL_SCALED` weitergeleitet. Dessen `INITO` wird an den äußeren `INITO`-Ausgang durchgereicht.
2. **Datenverarbeitung**: Sobald ein Ereignis am `IN.E1` eintrifft, triggert es die Ausführung des internen Sub-FBs über dessen `REQ`-Eingang. Das Sub-FB verarbeitet den aktuellen Wert von `IN.D1` (USINT) unter Anwendung von `SCALE` und `OFFSET` und berechnet einen skalierten Ausgabewert (REAL) sowie ein Gültigkeitsflag (BOOL).
3. **Ergebnisausgabe**: Nach der Berechnung erzeugt das Sub-FB ein `CNF`-Ereignis. Dieses triggert:
   - Die Ausgabe des skalierten Werts über `OUT.D1` und das Ereignis `OUT.E1`.
   - Den Takteingang (`CLK`) des D-Flipflops `E_D_FF`.
4. **Gültigkeitssignal**: Das Gültigkeitsflag des Sub-FBs wird auf den Dateneingang (`D`) des Flipflops gelegt. Mit jedem Taktereignis übernimmt das Flipflop den aktuellen `D`-Wert auf seinen Ausgang `Q`. Das Ereignis `E_D_FF.EO` triggert anschließend die Ausgabe des Gültigkeitssignals über `VALID.D1` und `VALID.E1`.
5. **Zusammenspiel**: Dadurch wird sichergestellt, dass das Gültigkeitssignal erst nach Abschluss der Skalierung und streng getaktet mit dem ausgegebenen skalierten Wert aktualisiert wird.

Technische Besonderheiten

- **Composite-Struktur**: Der FB besteht aus einem spezialisierten Sub-FB (`FIELDBUS_USINT_TO_SIGNAL_SCALED`) zur Skalierung und einem standardisierten D-Flipflop (`E_D_FF`) aus der IEC 61499-Ereignisbibliothek.
- **Verborgene Verbindungen**: Einige interne Verbindungen (z. B. für `INIT`, `SCALE`, `OFFSET`) sind als unsichtbar (`Visible="false"`) markiert, um die Übersichtlichkeit in der grafischen Darstellung zu erhöhen.
- **Datentypen**: Der Eingangswert ist als USINT (0…255) angenommen, der Ausgang wird durch die Skalierung zu REAL. Der Offset ist als DINT deklariert.
- **Flankengesteuerte Synchronisation**: Das D-Flipflop sorgt für eine entprellte Weitergabe des Gültigkeitssignals, sodass es nur bei einem neuen Verarbeitungszyklus aktualisiert wird.

Zustandsübersicht

Der FB besitzt keine eigenen expliziten Zustände. Die Zustandslogik ergibt sich aus den Sub-Komponenten:

- **Sub-FB `FIELDBUS_USINT_TO_SIGNAL_SCALED`**: Führt eine ablauforientierte Berechnung ohne Zustandsspeicher durch; es reagiert auf jedes `REQ`-Ereignis mit einem `CNF`.
- **D-Flipflop `E_D_FF`**: Besitzt zwei Zustände (Q = 0 oder 1). Bei jedem positiven Takt (CLK) wird Q = D gesetzt. Der Zustand bleibt bis zum nächsten Takt erhalten.

Anwendungsszenarien

- **Feldbus-Sensorauswertung**: Ein Sensor liefert über Feldbus einen USINT-Wert (z. B. 0–100 % Füllstand oder 0–255 Druckwerte). Der FB skaliert diesen Wert in eine physikalische Größe (z. B. Meter, bar) und gibt ihn zusammen mit einem Gültigkeitsflag aus.
- **Signalaufbereitung in der Agrartechnik**: In landwirtschaftlichen Maschinen werden oft skalierte Signale benötigt, bei denen die Gültigkeit der Daten (z. B. nach einer Kommunikationsstörung) explizit ausgewiesen werden muss.
- **Initialisierungskaskaden**: Der durchgereichte `INIT`-Ausgang ermöglicht die Initialisierung nachgeschalteter Bausteine, die ebenfalls auf dieses Signal angewiesen sind.

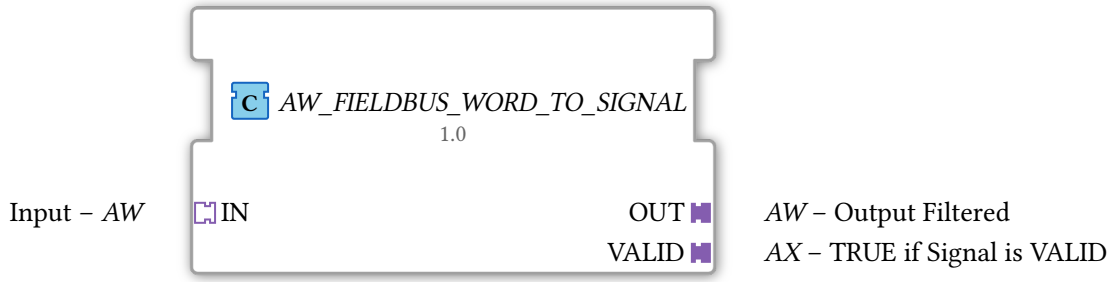
Vergleich mit ähnlichen Bausteinen

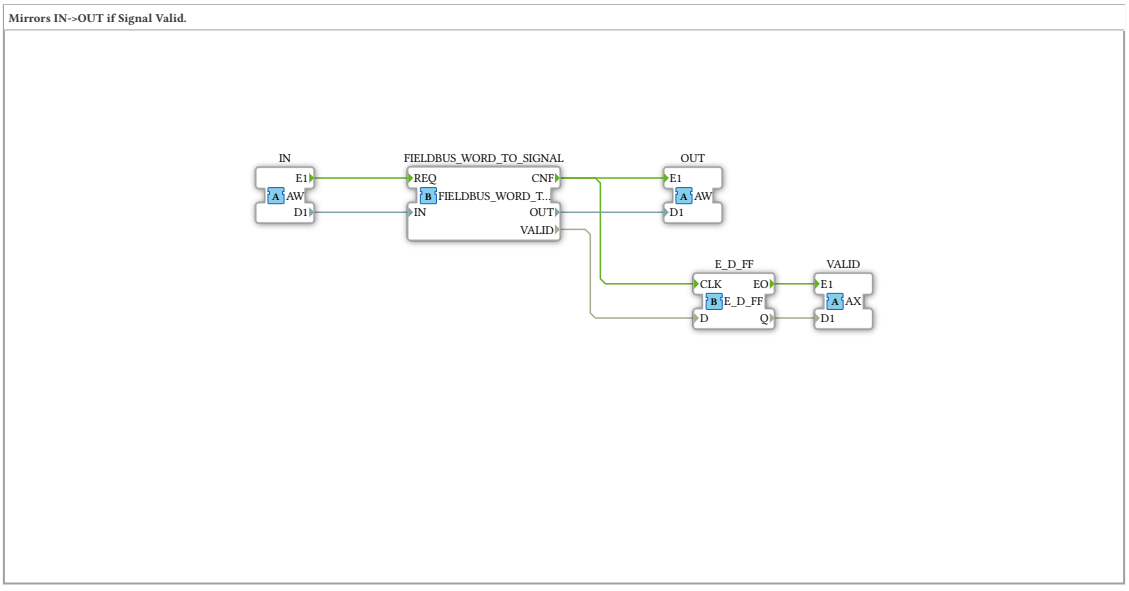
- **`AUS_FIELDBUS_USINT_TO_SIGNAL`** (ohne Skalierung): Bietet lediglich eine direkte Durchleitung des USINT-Werts ohne Skalierung und Offset.
- **`AUS_FIELDBUS_USINT_TO_SIGNAL_SCALED` mit separatem Valid-Toggle**: Ein alternativer Aufbau könnte das Gültigkeitssignal asynchron (ohne Flipflop) weiterleiten, was aber zu möglichen Zittereffekten führen kann.
- **`AUS_FIELDBUS_USINT_TO_SIGNAL_SCALED` mit integrierter Zustandsmaschine**: Statt eines Flipflops könnte eine eigene Zustandslogik das Gültigkeitssignal verwalten; der vorliegende Baustein nutzt jedoch bewährte Standardkomponenten.

Fazit

Der Funktionsblock `AUS_FIELDBUS_USINT_TO_SIGNAL_SCALED` bietet eine robuste und wartbare Lösung zur Skalierung von USINT-Feldbussignalen und zur zuverlässigen Weiterleitung des Gültigkeitsstatus. Durch die Composite-Struktur und die Verwendung eines D-Flipflops wird eine saubere zeitliche Trennung zwischen Datenberechnung und Ausgabe erreicht. Er ist besonders in Systemen mit Zyklusbasierter Datenverarbeitung und expliziter Signalgültigkeit nützlich.

AW_FIELDBUS_WORD_TO_SIGNAL





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **AW_FIELDBUS_WORD_TO_SIGNAL** dient als Filter für Feldbus-Wörter. Er spiegelt das eingehende Signal (über den Adapter **IN**) auf den Ausgang (Adapter **OUT**), sofern das Signal als gültig erkannt wurde. Die Gültigkeit wird über einen separaten Ausgangsadapter (**VALID**) signalisiert. Eingesetzt wird der Baustein typischerweise in Feldbus-Umgebungen, um nur valide Datenwerte an nachfolgende Komponenten weiterzuleiten.

Schnittstellenstruktur

Ereignis-Eingänge

Keine separaten Ereignis-Eingänge vorhanden.
Die Ereignissteuerung erfolgt implizit über die verwendeten Adapter (siehe Abschnitt **Adapter**).

Ereignis-Ausgänge

Keine separaten Ereignis-Ausgänge vorhanden.
Die Ereignisausgabe erfolgt implizit über die Adapter.

Daten-Eingänge

Keine separaten Daten-Eingänge vorhanden.
Die Datenübergabe erfolgt über die angeschlossenen Adapter.

Daten-Ausgänge

Keine separaten Daten-Ausgänge vorhanden.
Die Datenausgabe erfolgt über die angeschlossenen Adapter.

Adapter

Name	Typ	Beschreibung
IN	`adapter::types::unidirectional::AW` (Socket)	Eingang des zu filternden Wortes.
OUT	`adapter::types::unidirectional::AW` (Plug)	Ausgang des gefilterten Wortes (nur bei gültigem Signal).
VALID	`adapter::types::unidirectional::AX` (Plug)	Gibt den Gültigkeitsstatus des Ausgangssignals aus (`TRUE` = gültig).

Funktionsweise

Der Baustein besteht intern aus zwei Sub-Funktionsbausteinen: `FIELDBUS_WORD_TO_SIGNAL` und `E_D_FF` (D-Flip-Flop).

- Ein eingehendes Ereignis auf `IN.E1` triggert die Verarbeitung des aktuellen Wortes über `FIELDBUS_WORD_TO_SIGNAL.REQ`.
- Der Sub-Baustein `FIELDBUS_WORD_TO_SIGNAL` prüft die Gültigkeit des Wortes und gibt nach Abschluss ein `CNF`-Ereignis aus.
- Dieses `CNF`-Ereignis bewirkt:
  - Das gefilterte Wort wird über `OUT.D1` an den Ausgangsadapter übergeben und über `OUT.E1` ausgegeben.
  - Gleichzeitig wird das D-Flip-Flop (`E_D_FF`) getaktet: Der aktuelle Gültigkeitsstatus (`VALID`-Signal von `FIELDBUS_WORD_TO_SIGNAL`) wird übernommen.
- Der Ausgang des Flip-Flops (`Q`) wird über `VALID.D1` und das Ereignis `VALID.E1` (ausgelöst durch `E_D_FF.EO`) an den Adapter **VALID** weitergegeben.

Somit wird nur dann ein gültiges Wort ausgegeben, wenn die interne Prüfung des eingehenden Signals erfolgreich ist. Der Gültigkeitsstatus bleibt bis zur nächsten Verarbeitung erhalten.

Technische Besonderheiten

- Der Baustein basiert vollständig auf Adaptern und hat keine separate Event-/Data-Schnittstelle auf oberster Ebene.

- Die interne Realisierung verwendet einen standardisierten Feldbus-Verarbeitungsbaustein (`FIELDBUS_WORD_TO_SIGNAL`) und ein D-Flip-Flop, was eine klare Trennung von Datenpfad und Statuslogik ermöglicht.
- Die Implementierung ist unter der **Eclipse Public License 2.0** lizenziert.

Zustandsübersicht

Der Baustein selbst besitzt keinen expliziten Zustandsautomaten. Der innere Zustand wird durch das D-Flip-Flop repräsentiert:

Zustand	Bedeutung
`VALID = FALSE`	Der Ausgangswert ist ungültig (alte Daten oder Initialzustand).
`VALID = TRUE`	Der Ausgangswert ist gültig und wurde beim letzten Verarbeitungszyklus als valide erkannt.

Anwendungsszenarien

- **Feldbus-Datenfilterung**: In Automatisierungssystemen, bei denen nur gültige Messwerte oder Steuerbefehle an nachfolgende Logik weitergegeben werden sollen.
- **Signalvalidierung**: Einsatz in sicherheitskritischen Pfaden, wo ungültige Daten (z. B. aufgrund von Übertragungsfehlern) nicht ungeprüft weitergeleitet werden dürfen.
- **Adapterbasierte Schnittstellen**: Geeignet für modulare Systeme, die über standardisierte Adapter kommunizieren (z. B. logiBUS-Umgebung).

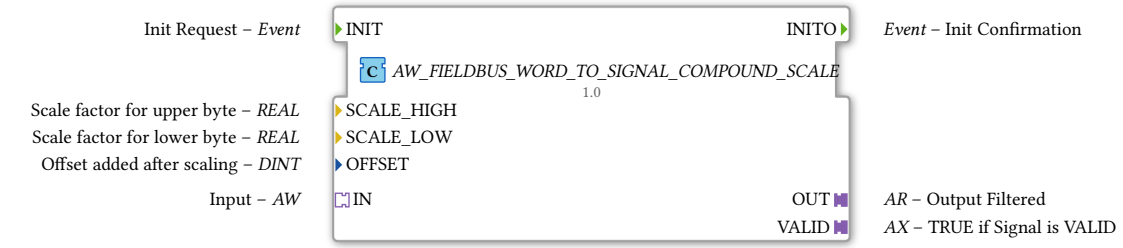
Vergleich mit ähnlichen Bausteinen

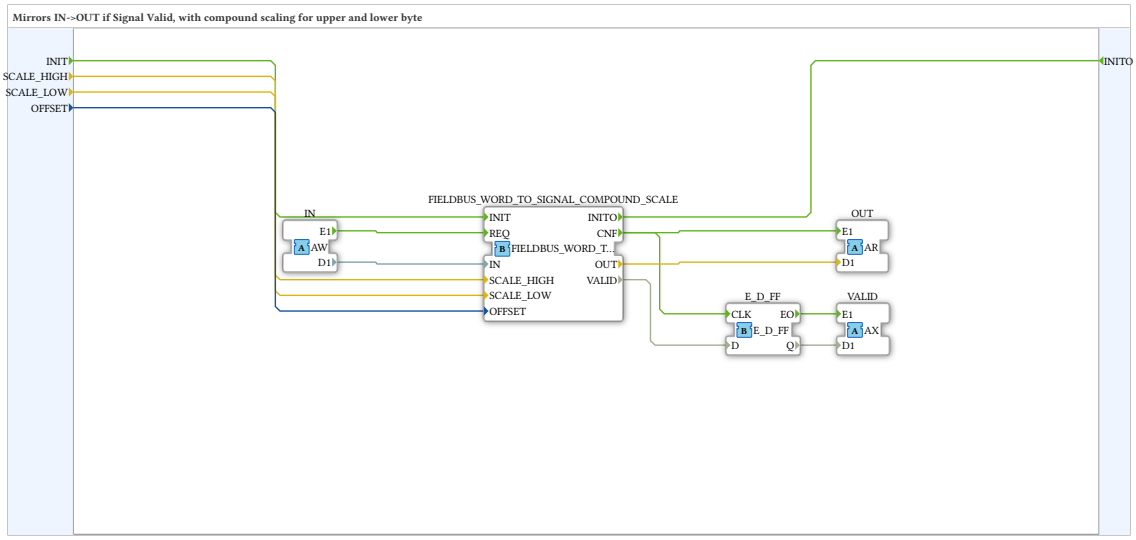
- Einfache **Word-to-Signal**-Bausteine ohne Validierung geben das Eingangssignal direkt weiter, unabhängig von dessen Gültigkeit.
- Der **AW_FIELDBUS_WORD_TO_SIGNAL** bietet im Gegensatz dazu eine explizite Gültigkeitsrückmeldung und unterdrückt ungültige Werte.
- Andere Filterbausteine könnten zusätzliche Konfigurationsmöglichkeiten (z. B. Schwellwerte, Zeitfenster) enthalten, während dieser Baustein auf eine reine „gültig / ungültig“-Entscheidung reduziert ist.

Fazit

Der AW_FIELDBUS_WORD_TO_SIGNAL ist ein spezialisierter Filterbaustein für Feldbus-Wortdaten. Er kombiniert eine einfache Durchschleifung mit einer Gültigkeitsprüfung und stellt das Ergebnis über separate Adapter bereit. Durch die Verwendung von Standard-Subbausteinen und Flip-Flop-Logik ist er robust, nachvollziehbar und gut in adapterbasierte Architekturen integrierbar.

AW_FIELDBUS_WORD_TO_SIGNAL_COMPOUND_SCALE





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AW_FIELDBUS_WORD_TO_SIGNAL_COMPOUND_SCALE** dient der Umwandlung eines über einen Feldbus eingehenden 16-Bit-Wortes in einen skalierten Signalwert. Das Wort wird dabei byteweise mit unterschiedlichen Skalierungsfaktoren multipliziert, ein Offset addiert und das Ergebnis als normiertes Signal ausgegeben. Zusätzlich wird ein Gültigkeitssignal bereitgestellt, das anzeigt, ob der eingehende Wert gültig ist.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung
INIT	EInit	Init-Anforderung

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung
INITO	EInit	Init-Bestätigung

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung
SCALE_HIGH	REAL	0.256	Skalierungsfaktor für das obere Byte
SCALE_LOW	REAL	0.001	Skalierungsfaktor für das untere Byte
OFFSET	DINT	0	Additiver Offset nach der Skalierung

Daten-Ausgänge

Der FB besitzt keine eigenständigen Datenausgänge. Die Ausgangsdaten werden über die unten genannten Adapter bereitgestellt.

Adapter

Adapter	Typ (Richtung)	Beschreibung
IN	`adapter::types::unidirectional::AW`	Eingehendes 16-Bit-Wort

OUT	`adapter::types::unidirectional::AR`	Ausgangssignal (skaliert)
VALID	`adapter::types::unidirectional::AX`	Gültigkeitssignal (TRUE = gültig)

Funktionsweise

Nach einer erfolgreichen Initialisierung (Ereignis INIT → INITO) beginnt der FB auf eingehende Daten über den **IN**-Adapter zu reagieren. Sobald das Ereignis **IN.E1** empfangen wird, wird das momentan an **IN.D1** anliegende 16-Bit-Wort verarbeitet:

1. **Byteweise Skalierung**
   Das obere Byte (Bits 15…8) wird mit `SCALE_HIGH` multipliziert, das untere Byte (Bits 7…0) mit `SCALE_LOW`.
2. **Offsetaddition**
   Die Summe beider skalierten Bytewerte wird um den konfigurierten `OFFSET` erhöht.
3. **Ausgabe**
   Das Ergebnis wird an den **OUT**-Adapter übergeben (Ereignis **OUT.E1** mit Daten **OUT.D1**).
4. **Gültigkeitshandling**
   Parallel wird ein interner Flip-Flop (`E_D_FF`) getaktet. Der Gültigkeitsstatus (Signal **VALID**) wird dabei aus dem internen **VALID**-Ausgang des Skalierungsmoduls übernommen und bleibt bis zum nächsten gültigen Signal erhalten.

Das interne Flip-Flop sorgt dafür, dass das Gültigkeitssignal erst beim nächsten Eingangsereignis aktualisiert wird. So kann ein stabiler Signalzustand am Ausgang gewährleistet werden.

Technische Besonderheiten

- **Compound Scaling**
  Die getrennte Skalierung von oberem und unterem Byte erlaubt die Verarbeitung von Feldbussignalen, die in zwei unterschiedlich skalierten Bytehälften codiert sind (z. B. Temperatur- oder Druckwerte mit Spanne und Auflösung).
- **Integrierter Flip-Flop**
  Der Gültigkeitsausgang wird über ein flankengesteuertes D-Flip-Flop realisiert. Dadurch wird das Gültigkeitssignal nur bei einem neuen Eingangsereignis aktualisiert, was eine stabile Ausgabe gewährleistet.
- **Konfigurierbare Parameter**
  Skalierung und Offset sind über die Daten-Eingänge frei einstellbar und können zur Laufzeit geändert werden.
- **Initialisierung**
  Vor der ersten Nutzung muss der FB mit dem INIT-Ereignis initialisiert werden. Dabei werden die internen Zustände zurückgesetzt.

Zustandsübersicht

Der FB besitzt einen internen Zustand, der durch das Flip-Flop `E_D_FF` abgebildet wird:

Zustand	Beschreibung
UNVALID	Kein gültiges Signal. Der **VALID**-Ausgang liefert FALSE.
VALID	Gültiges Signal erkannt. Der **VALID**-Ausgang zeigt TRUE an.

Der Zustand wechselt mit jedem neuen Eingangsereignis (also mit jeder Übergabe eines Worts) entsprechend der Gültigkeitsinformation des internen Skalierungsmoduls.

Anwendungsszenarien

- **Analogwerterfassung über Feldbus**
  Ein Feldbusgerät sendet einen 16-Bit-Rohwert, bei dem das obere Byte einen Grobwert (z. B. 0…255) und das untere Byte einen Feinwert (z. B. 0…1000) darstellt. Mit `SCALE_HIGH` und `SCALE_LOW` wird der physikalische Messwert berechnet.

- **Prozessdatenaufbereitung**
  In einer Steuerung werden mehrere skalierte Signale benötigt, deren Gültigkeit überwacht werden muss. Der FB vereint Skalierung und Validierung in einem Baustein.
- **Gateway-Funktion**
  Der FB kann als Konverter zwischen einem Feldbusprotokoll (Wort-basiert) und einem internen Signalprotokoll (mit Gültigkeitsflag) eingesetzt werden.

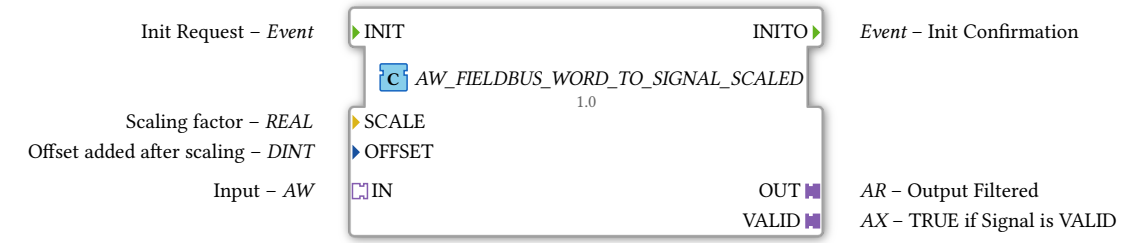
Vergleich mit ähnlichen Bausteinen

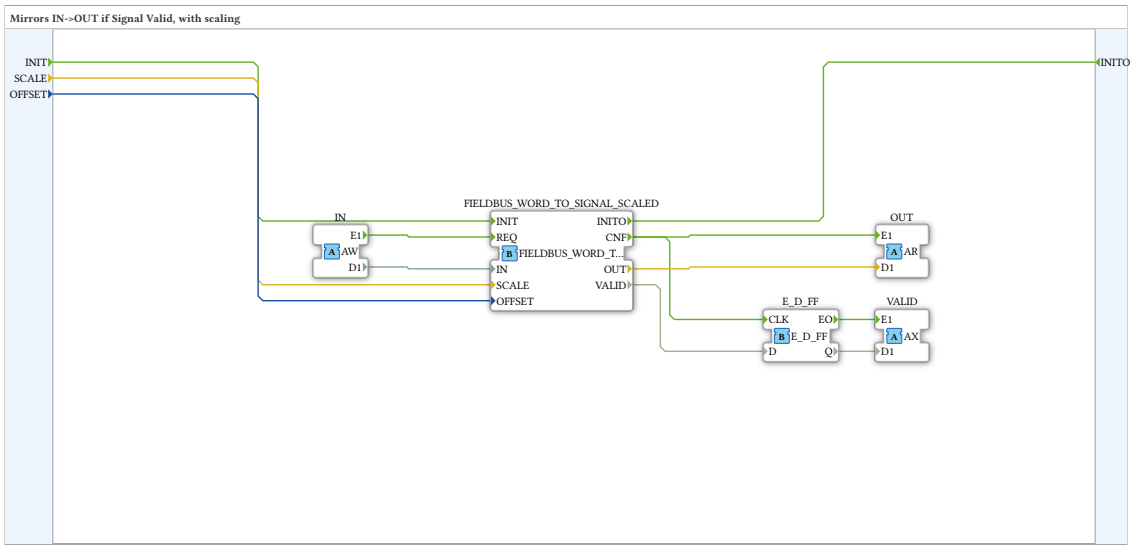
- **FIELDBUS_WORD_TO_SIGNAL** (einfach)
  Dieser Baustein skaliert das gesamte 16-Bit-Wort mit einem einzigen Faktor. Der Compound-Baustein erlaubt dagegen eine byteweise angepasste Skalierung.
- **SCALE** (allgemein)
  Ein universeller Skalierungsbaustein ohne Feldbusschnittstelle. Der vorliegende FB integriert die Feldbus-Adapter und das Gültigkeitsmanagement.
- **BADAPTER_STATUS**
  Liefert nur einen Status über die Gültigkeit, ohne Skalierung. Der Compound-Baustein kombiniert beide Funktionen.

Fazit

Der **AW_FIELDBUS_WORD_TO_SIGNAL_COMPOUND_SCALE** ist ein spezialisierter Funktionsblock für die Aufbereitung von Feldbus-Wortsignalen mit unterschiedlichen Skalierungen für oberes und unteres Byte. Seine integrierte Gültigkeitsverwaltung über ein Flip-Flop macht ihn besonders geeignet für sicherheitskritische oder anwendungsfallbezogene Signalverarbeitung in der Automatisierungstechnik. Dank der konfigurierbaren Parameter und der klar strukturierten Schnittstelle lässt er sich einfach in übergeordnete Steuerungslogiken einbinden.

AW_FIELDBUS_WORD_TO_SIGNAL_SCALED





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `AW_FIELDBUS_WORD_TO_SIGNAL_SCALED` dient dazu, ein eingehendes Feldbus-Wort (16-Bit) unter Berücksichtigung eines Skalierungsfaktors und Offsets auf einen physikalischen Ausgabewert zu übertragen. Zusätzlich wird die Gültigkeit des Signals über einen eigenen Ausgang überwacht und gespeichert. Der Baustein kapselt die typische Verarbeitungskette von der digitalen Übertragung bis zur normierten Analog- oder Signalgröße.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar
INIT	EInit	Initialisierungsanforderung (z. B. Reset der internen Logik)

Ereignis-Ausgänge

Ereignis	Typ	Kommentar
INITO	EInit	Bestätigung der erfolgreichen Initialisierung

Daten-Eingänge

Name	Typ	Default	Kommentar
SCALE	REAL	REAL#1.0	Skalierungsfaktor
OFFSET	DINT	DINT#0	Ganzzahliger Offset (nach Skalierung)

Daten-Ausgänge

Der Baustein besitzt keine separaten Datenausgänge. Die Ausgangswerte werden über die Adapter bereitgestellt.

Adapter

Adapter	Richtung	Typ (unidirektional)	Kommentar
IN	Socket	AW	Eingangssignal (Feldbus-Wort)
OUT	Plug	AR	Verarbeitetes und skaliertes Ausgangssignal
VALID	Plug	AX	Boolescher Wert: TRUE, wenn das Signal gültig ist

Funktionsweise

Der Baustein arbeitet mit einem internen Sub-FB `FIELDBUS_WORD_TO_SIGNAL_SCALED` und einem flankengesteuerten D-Flip-Flop (`E_D_FF`).

1. **Initialisierung:** Ein INIT-Ereignis wird an den Sub-FB weitergeleitet. Nach dessen Abschluss wird INITO ausgegeben.
2. **Datenverarbeitung:** Ein eingehendes Ereignis auf `IN.E1` triggert den Sub-FB (`REQ`). Dieser liest das aktuelle Wort von `IN.D1`, skaliert es mit dem Faktor `SCALE`, addiert den `OFFSET` und stellt das Ergebnis auf dem internen Ausgang `OUT` bereit. Gleichzeitig wird ein Gültigkeitsflag (`VALID`) erzeugt.
3. **Weitergabe und Speicherung:** Nach erfolgreicher Verarbeitung (`CNF`) wird das Ausgangsereignis auf `OUT.E1` gesendet und das Gültigkeitsflag in das D-Flip-Flop übernommen (Taktflanke). Der gespeicherte Wert steht am Validierungsausgang (`VALID.D1`) stabil zur Verfügung.
4. **Signalvalidität:** Wenn das eingehende Wort als ungültig erkannt wird (z. B. durch Fehlerbits oder Plausibilitätsprüfung im Sub-FB), setzt der Sub-FB das `VALID`-Signal auf FALSE. Das Flip-Flop friert diesen Zustand ein, bis der nächste gültige Zyklus eintritt.

Technische Besonderheiten

- **Skalierung mit Offset:** Die Verarbeitung erfolgt in der Reihenfolge `(Wort * SCALE) + OFFSET`. Dadurch können lineare Umrechnungen realisiert werden, z. B. Rohwerte in physikalische Einheiten.

- **Gültigkeitsspeicherung:** Das D-Flip-Flop entkoppelt die dynamische Validität vom Ausgangssignal. Ein einmal als ungültig markiertes Signal bleibt so lange ungültig, bis wieder ein gültiger Zyklus durchläuft – das verhindert kurzzeitige Fehlinterpretationen.
- **Adapterbasierte Schnittstelle:** Der Baustein verwendet uni-direktionale Adapter eines Typsystems, das auf Feldbus-Kommunikation ausgelegt ist (`AW`, `AR`, `AX`). Die genauen Datentypen der Adapter (z. B. Wort, Real, Bool) werden durch die Adapterdefinitionen bereitgestellt.

Zustandsübersicht

Der Baustein selbst hat keine explizite Zustandsmaschine in der XML. Sein Verhalten ergibt sich aus der Abfolge der internen Elemente:

- **Warten auf INIT:** Kein Betrieb bevor INIT ausgeführt.
- **Betrieb:** Nach INIT lauscht der Baustein auf Ereignisse am `IN`-Socket. Jedes eingehende Ereignis startet einen Verarbeitungsschritt.
- **Ausgabe:** Das Ausgangsereignis `OUT.E1` wird unmittelbar nach Abschluss der internen Skalierung gesendet. Das Validitätssignal `VALID` wird zeitgleich aktualisiert.

Anwendungsszenarien

- **Feldbus-Analogwertaufbereitung:** Ein von einem Feldbus übertragenes 16-Bit-Signal (z. B. 0…65535) wird mittels Skalierung in einen physikalischen Bereich wie 0 … 10 V oder 4 … 20 mA umgerechnet.
- **Sensorwert mit Gültigkeitsflag:** Sensoren, die über ein Feldbusprotokoll Daten liefern, können auch einen Gültigkeitsstatus (z. B. „Data valid“ Bit) senden. Dieser wird über den `VALID`-Ausgang geführt.
- **Sicherheitsgerichtete Übertragung:** Wenn das Feldbuswort ein Fehlerbit enthält, kann der Sub-FB `VALID` auf FALSE setzen, sodass nachgeschaltete Logiken das Signal als ungültig erkennen.

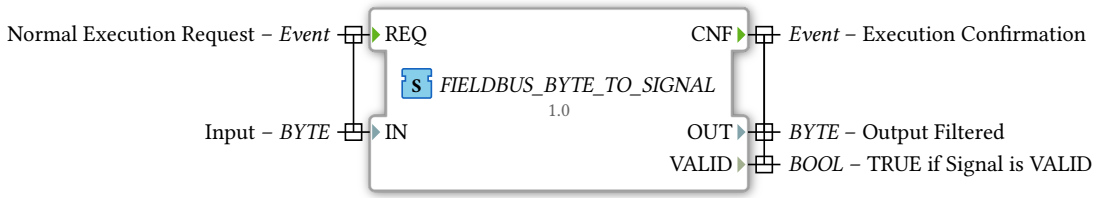
Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu einfachen Wort-zu-Signal-Konvertern (z. B. `FIELDBUS_WORD_TO_SIGNAL`) besitzt `AW_FIELDBUS_WORD_TO_SIGNAL_SCALED` eine integrierte Gültigkeitsspeicherung und getrennte Adapter für Ausgang und Validität. Dies reduziert die Verdrahtung in der Applikation und vereinfacht die Handhabung von temporären Fehlern. Bausteine ohne Offset-/Skalierungslogik müssten extern mit Rechenblöcken ergänzt werden.

Fazit

`AW_FIELDBUS_WORD_TO_SIGNAL_SCALED` ist ein kompakter, funktionaler Baustein zur Aufbereitung von Feldbussignalen. Er verbindet Skalierung, Offset und Gültigkeitsüberwachung in einem wiederverwendbaren Modul. Die adapterbasierte Schnittstelle und die gespeicherte Validität machen ihn besonders geeignet für industrielle Steuerungen, bei denen Signalqualität und einfache Parametrierung im Vordergrund stehen.

FIELDBUS_BYTE_TO_SIGNAL



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **FIELDBUS_BYTE_TO_SIGNAL** dient der einfachen Validierung eines BYTE-Signals aus einem Feldbus. Er gibt den Eingangswert nur dann unverändert an den Ausgang weiter, wenn dieser innerhalb eines definierten gültigen Bereichs liegt. Das Ergebnis der Validierung wird über einen separaten booleschen Ausgang signalisiert.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Datentyp	Kommentar
REQ	Event	Normaler Ausführungsanstoß; verarbeitet den aktuellen Wert am Eingang IN.

Ereignis-Ausgänge

Ereignis	Datentyp	Kommentar
CNF	Event	Bestätigung der Ausführung; die Ausgänge OUT und VALID wurden aktualisiert.

Daten-Eingänge

Name	Datentyp	Initialwert	Kommentar
IN	BYTE	NOT_AVAILABLE_B	Eingangssignal als Byte-Wert.

Daten-Ausgänge

Name	Datentyp	Initialwert	Kommentar
OUT	BYTE	16#00	Gefilterter Ausgangswert; 0 wenn Signal ungültig.
VALID	BOOL	FALSE	TRUE, wenn das Eingangssignal als gültig erkannt wurde.

Adapter

Keine.

Funktionsweise

Der Block prüft bei jedem REQ-Ereignis den aktuellen BYTE-Eingangswert **IN** gegen eine vorgegebene Konstante **VALID_SIGNAL_B** (aus dem Import `eclipse4diac::signalprocessing::FIELDBUS_SIGNAL`). Die Logik im Algorithmus lautet:

```
IF (BYTE_TO_USINT(IN) <= BYTE_TO_USINT(VALID_SIGNAL_B)) THEN
    OUT := IN;
    VALID := TRUE;
ELSE
    OUT := BYTE#0;
    VALID := FALSE;
END_IF;
```

- Ist der Wert von **IN** (als vorzeichenloser Integer interpretiert) **kleiner oder gleich** dem Schwellwert **VALID_SIGNAL_B**, wird der Eingang direkt auf **OUT** durchgereicht und **VALID** auf `TRUE` gesetzt.
- Überschreitet **IN** den Schwellwert, wird **OUT** auf `0` zurückgesetzt und **VALID** auf `FALSE`.

Nach der Berechnung wird das Ereignis **CNF** ausgegeben. Der Initialwert von **IN** liegt mit `NOT_AVAILABLE_B` bereits außerhalb des gültigen Bereichs, sodass der Block nach dem ersten Durchlauf standardmäßig auf „ungültig“ steht.

Technische Besonderheiten

- Der Filter basiert auf **zwei externen Konstanten** aus der Bibliothek `eclipse4diac::signalprocessing::FIELDBUS_SIGNAL`:
  - `NOT_AVAILABLE_B` – definiert den Wert für „Signal nicht verfügbar“ (Initialwert von IN).
  - `VALID_SIGNAL_B` – definiert die obere Grenze für gültige Signale.
- Die Umwandlung `BYTE_TO_USINT` stellt sicher, dass die Werte als Zahlen 0…255 verglichen werden können.
- Der Block ist als **SimpleFB** mit nur einem Zustand (**REQ**) realisiert – keine Zustandsmaschine mit mehreren Schritten.
- Lizenz: Eclipse Public License 2.0.

Zustandsübersicht

Der Funktionsblock besitzt genau einen Ausführungszustand:

Zustand	Aktion	Ausgabeereignis
REQ	Führt den Algorithmus **REQ** aus (Filterlogik)	CNF

Es gibt keine weiteren Warte- oder Initialisierungszustände.

Anwendungsszenarien

- **Feldbus-Signalvalidierung**: Ein Sensor liefert einen BYTE-Wert, der nur innerhalb eines bestimmten Bereichs (z. B. 0…100) physikalisch sinnvoll ist. Der Block verhindert, dass fehlerhafte oder unplausible Werte in die Steuerungslogik gelangen.
- **Sicherheitsbezogene Systeme**: Kann als einfacher Bereichsprüfer für analoge oder digitale Feldbussignale in der Automatisierungstechnik (z. B. Landwirtschaft, Maschinensteuerung) eingesetzt werden.
- **Signalweiche**: Trennung von gültigen und ungültigen Datenströmen – z. B. zur Ansteuerung eines UND-Filters oder zur Fehlererkennung.

Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu einfachen **Schwellwertschaltern** (z. B. `HYSTERESIS` oder `LIMIT_ALARM`) arbeitet dieser Block rein auf BYTE-Ebene und gibt den Originalwert direkt aus, solange er im Gültigkeitsbereich liegt – keine Hysterese oder Alarmfunktion.
- Gegenüber einem **Multiplexer** (`MUX`) benötigt er kein zweites Steuersignal, sondern trifft die Entscheidung automatisch anhand des Eingangswertes.
- Ähnliche Blöcke in IEC 61499 existieren oft als `RANGE_CHECK` oder `VALIDATE`, sind aber meist auf generische Datentypen ausgelegt. Dieser Block ist speziell auf das in Feldbussen übliche BYTE-Format zugeschnitten.

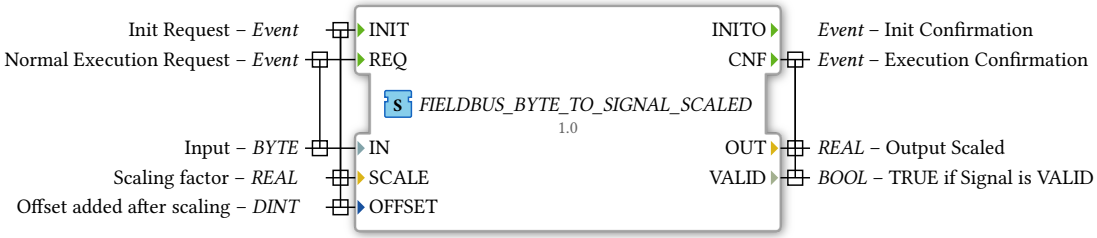
Fazit

FIELDBUS_BYTE_TO_SIGNAL ist ein kompakter, spezialisierter Filterbaustein zur Gültigkeitsprüfung von BYTE-Signalen. Er erfordert keine Konfiguration von Grenzen, da diese über die importierten Konstanten in der Bibliothek definiert sind. Durch die klare Trennung von Datenausgang und Gültigkeitssignal eignet er sich hervorragend für die Aufbereitung von Feldbusdaten in sicherheitskritischen oder fehlertoleranten Automatisierungssystemen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

FIELDBUS_BYTE_TO_SIGNAL_SCALED



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **FIELDBUS_BYTE_TO_SIGNAL_SCALED** dient der Umsetzung eines eingehenden Byte-Werts in einen skalierten Real-Wert. Dabei wird geprüft, ob das eingehende Signal gültig ist. Ist dies der Fall, wird der Eingangswert mit einem Skalierungsfaktor multipliziert und ein Offset addiert. Andernfalls wird der Ausgang auf Null gesetzt und das Gültigkeitssignal zurückgesetzt. Der Block ist für den Einsatz in Feldbussystemen konzipiert, bei denen Sensor- oder Aktordaten als Byte übertragen werden und linearisiert oder normiert werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

Event	Typ	Beschreibung	Mitgeführte Daten
`INIT`	EInit	Initialisierungsanforderung; setzt Skalierung und Offset	`SCALE`, `OFFSET`
`REQ`	Event	Normale Verarbeitungsanforderung; wertet den Eingang `IN` aus	`IN`

Ereignis-Ausgänge

Event	Typ	Beschreibung	Mitgeführte Daten
`INITO`	EInit	Bestätigung der Initialisierung	–
`CNF`	Event	Bestätigung der Verarbeitung; liefert skalierten Wert und Gültigkeit	`OUT`, `VALID`

Daten-Eingänge

Name	Datentyp	Initialwert	Beschreibung
`IN`	BYTE	`NOT_AVAILABLE_B`	Feldbus-Rohwert (Byte)
`SCALE`	REAL	`1.0`	Skalierungsfaktor (Multiplikator)
`OFFSET`	DINT	`0`	Additiver Offset (nach Skalierung)

Daten-Ausgänge

Name	Datentyp	Initialwert	Beschreibung
`OUT`	REAL	`0.0`	Skalierter Ausgangswert
`VALID`	BOOL	`FALSE`	`TRUE`, wenn der eingehende Byte-Wert als gültig bewertet wurde

Adapter

Keine.

Funktionsweise

Der Funktionsblock wird durch zwei Ereignisse gesteuert:

1. **Initialisierung (`INIT`)** Beim Eintreffen von `INIT` werden die Parameter `SCALE` und `OFFSET` übernommen. Der dazugehörige Algorithmus (`INIT`) führt keine weiteren Berechnungen durch. Nach erfolgreicher Initialisierung wird das Ereignis `INITO` gesendet.
2. **Verarbeitung (`REQ`)** Bei `REQ` wird der Wert von `IN` mit einem festen Schwellwert `VALID_SIGNAL_B` verglichen. Dieser Schwellwert wird aus dem importierten Paket `eclipse4diac::signalprocessing::FIELDBUS_SIGNAL` bezogen.
   - **Signal gültig:** Wenn `IN <= VALID_SIGNAL_B`, wird der Ausgangswert nach der Formel `OUT := IN * SCALE + OFFSET` berechnet und `VALID` auf `TRUE` gesetzt.
   - **Signal ungültig:** Andernfalls wird `OUT` auf `0.0` und `VALID` auf `FALSE` gesetzt.

   Anschließend wird das Ereignis `CNF` mit den aktuellen Werten von `OUT` und `VALID` gesendet.

Der initiale Wert von `IN` ist `NOT_AVAILABLE_B` – ein weiterer importierter Konstantwert, der ein nicht verfügbares Signal kennzeichnet. Dadurch wird bei erstmaligem Aufruf von `REQ` (ohne vorherige gültige Eingabe) stets eine ungültige Ausgabe erzeugt.

Technische Besonderheiten

- Die Konstanten `NOT_AVAILABLE_B` und `VALID_SIGNAL_B` werden aus einem separaten Bibliothekspaket importiert und nicht lokal definiert. Dadurch sind sie projektweit einheitlich.

- Die Skalierung erfolgt linear: Multiplikation mit `SCALE` und Addition von `OFFSET`. Der Offset wird vor der Umrechnung in `REAL` vom Typ `DINT` nach `REAL` konvertiert.
- Der Eingang `IN` wird zunächst von `BYTE` nach `USINT` (unsigned short integer) gewandelt, um einen ganzzahligen Wertebereich 0…255 zu erhalten.
- Der Block besitzt einen expliziten Initialisierungsmodus (`INIT`), der es erlaubt, Skalierung und Offset während der Laufzeit neu zu setzen.

Zustandsübersicht

Der Funktionsblock besitzt zwei einfache Zustände, die jeweils durch das entsprechende Ereignis aktiviert werden:

Zustand	Auslösendes Event	Ausführbarer Algorithmus	Ausgehende Events
`INIT`	`INIT`	`INIT` (leer)	`INITO`
`REQ`	`REQ`	`REQ` (Gültigkeitsprüfung + Skalierung)	`CNF`

Es gibt keine automatischen Zustandsübergänge; jeder Zustand wird ausschließlich durch das zugehörige Ereignis angesprochen.

Anwendungsszenarien

- **Austausch analoger Sensordaten über Feldbusse:** Ein 8-Bit-Sensorwert (z. B. Temperatur 0…255) soll in physikalische Einheiten (z. B. °C) umgerechnet werden. Über `SCALE` und `OFFSET` wird die Kennlinie angepasst.
- **Fehlererkennung bei der Datenübertragung:** Wenn der Feldbus bestimmte Byte-Werte als ungültig definiert (z. B. `0xFF` für „Sensorfehler“), erkennt der Block diese anhand der Schranke `VALID_SIGNAL_B` und meldet sie über `VALID = FALSE`.
- **Initialisierung von Konfigurationsparametern:** Vor der ersten Verarbeitung kann über `INIT` der gewünschte Skalierungsfaktor und Offset gesetzt werden.

Vergleich mit ähnlichen Bausteinen

- **`FIELDBUS_BYTE_TO_SIGNAL`** (ohne Skalierung): Dieser Baustein gibt den Byte-Wert direkt als REAL aus, ohne Multiplikation oder Addition. Er eignet sich für Signale, die bereits in der richtigen Einheit vorliegen.
- **`FIELDBUS_BYTE_TO_SIGNAL_SCALED`** (vorliegender Baustein): Bietet zusätzlich die lineare Skalierung und einen Offset. Der Anwender kann die Umrechnung flexibel anpassen, ohne den Algorithmus zu ändern.
- **`FIELDBUS_WORD_TO_SIGNAL_SCALED`**: Analoger Baustein für 16-Bit-Werte (WORD) mit identischer Skalierungslogik, jedoch anderem Eingangsdatentyp.

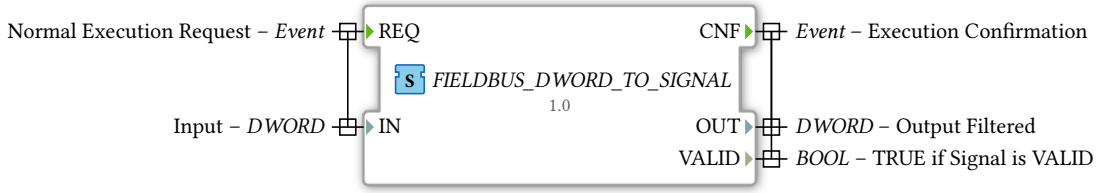
Fazit

Der Funktionsblock `FIELDBUS_BYTE_TO_SIGNAL_SCALED` ist ein praktisches Werkzeug für die Verarbeitung von Feldbus-Signalen im Byte-Format. Er kombiniert eine einfache Gültigkeitsprüfung mit einer frei konfigurierbaren linearen Skalierung. Durch die Verwendung importierter Konstanten ist er in einheitliche Signalverarbeitungspipelines integrierbar. Die klare Trennung von Initialisierung und Betrieb sowie die zwei Ereignisschnittstellen machen ihn zu einem robusten und leicht verständlichen Baustein für industrielle Automatisierungslösungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

FIELDBUS_DWORD_TO_SIGNAL



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `FIELDBUS_DWORD_TO_SIGNAL` dient der Spiegelung und Validierung eines DWORD-Eingangssignals. Ist der Eingangswert gültig, wird er unverändert an den Ausgang weitergegeben und das `VALID`-Signal gesetzt. Andernfalls liefert der Baustein einen Nullwert und signalisiert die Ungültigkeit.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Kommentar
REQ	Normaler Ausführungsanstoß; verarbeitet den aktuellen Wert von `IN`.

Ereignis-Ausgänge

Ereignis	Kommentar
CNF	Bestätigung der Ausführung; wird nach der Berechnung gesendet.

Daten-Eingänge

Variable	Typ	Initialwert	Kommentar
IN	DWORD	`NOT_AVAILABLE_DWM`	Eingangssignal

Daten-Ausgänge

Variable	Typ	Initialwert	Kommentar
OUT	DWORD	`16#00000000`	Gefilterter Ausgangswert
VALID	BOOL	`FALSE`	`TRUE`, wenn das Signal gültig ist

Adapter

Keine.

Funktionsweise

Der Baustein wird durch ein Ereignis an `REQ` aktiviert. Im zugehörigen Algorithmus wird der Eingangswert `IN` (als vorzeichenloser Integer) mit einem vordefinierten Grenzwert `VALID_SIGNAL_DW` verglichen:

- Ist `DWORD_TO_UDINT(IN) <= DWORD_TO_UDINT(VALID_SIGNAL_DW)`, so gilt das Signal als **gültig**:
  - `OUT` erhält den Wert von `IN`.
  - `VALID` wird auf `TRUE` gesetzt.
- Andernfalls (Signal ungültig):
  - `OUT` wird auf `DWORD#0` gesetzt.
  - `VALID` wird auf `FALSE` gesetzt.

Nach Abschluss der Berechnung wird das Ausgangsereignis `CNF` gesendet.

Technische Besonderheiten

- Der Grenzwert `VALID_SIGNAL_DW` stammt aus dem importierten Paket `eclipse4diac::signalprocessing::FIELDBUS_SIGNAL`. Er definiert den maximalen zulässigen Wert für ein gültiges Signal.
- Der Initialwert des Eingangs `IN` ist `NOT_AVAILABLE_DWM` (ebenfalls importiert), sodass der Baustein bei Start bereits einen ungültigen Zustand annimmt.
- Die Umwandlung von DWORD nach UDINT ermöglicht einen einfachen numerischen Vergleich (≤), ohne Vorzeichenprobleme.
- Der Baustein ist als **SimpleFB** mit einem einzigen ECA-Zustand `REQ` realisiert.

Zustandsübersicht

Zustand	Aktion	Ausgangsereignis
REQ	Ausführen `REQ`-Algorithmus	CNF

Es gibt nur einen Zustand; nach dessen Abarbeitung kehrt der Baustein in den IDLE-Zustand zurück (implizit).

Anwendungsszenarien

- **Feldbus-Signalverarbeitung**: Ein DWORD-Wert von einem Feldbus (z. B. CANopen, Profibus) wird auf Gültigkeit geprüft, bevor er in der Steuerung verwendet wird.
- **Datenvalidierung**: Der Baustein kann als Filter eingesetzt werden, um ungültige oder fehlerhafte Telegramme zu erkennen und durch einen definierten Nullwert zu ersetzen.
- **Sicherheitskritische Systeme**: Durch die klare Trennung gültiger/ungültiger Signale lässt sich die Signalqualität überwachen und weiterverarbeiten.

Vergleich mit ähnlichen Bausteinen

- Ein einfacher **MOVE**-Baustein würde den Wert ungeprüft durchreichen; `FIELDBUS_DWORD_TO_SIGNAL` fügt die Validierungslogik hinzu.
- Bausteine wie `FIELDBUS_SIGNAL_TO_DWORD` (sofern vorhanden) führen die entgegengesetzte Wandlung durch – hier wird ein DWORD-Signal in ein validiertes Signal mit Gültigkeitsflag umgesetzt.
- Der Baustein ist spezialisiert auf DWORD-Werte; für andere Datentypen (z. B. WORD, SINT) wären entsprechende Varianten denkbar.

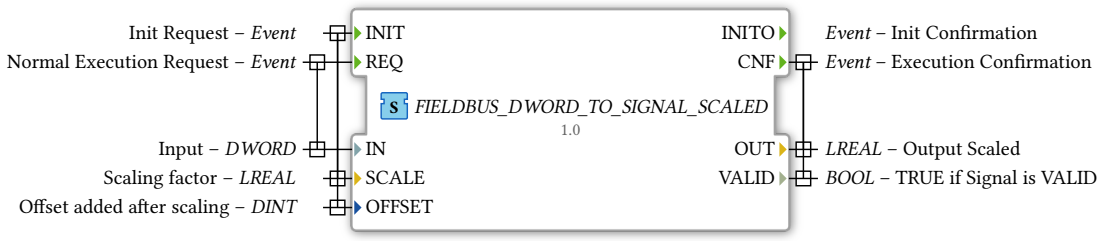
Fazit

`FIELDBUS_DWORD_TO_SIGNAL` ist ein kompakter, aber essenzieller Funktionsblock für die robuste Feldbus-Anbindung. Er kombiniert Signalweitergabe mit einer einfachen Gültigkeitsprüfung und stellt die Validität als separates Bool-Signal bereit. Dadurch können nachgeschaltete Bausteine schnell auf gültige Daten reagieren oder Fehlerbehandlungen einleiten. Der Einsatz der importierten Konstanten gewährleistet eine konsistente Definition der Gültigkeitsschwelle über mehrere Bausteine hinweg.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

FIELDBUS_DWORD_TO_SIGNAL_SCALED



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **FIELDBUS_DWORD_TO_SIGNAL_SCALED** dient der Konvertierung eines Feldbus-DWORD-Signals in einen skalierten LREAL-Wert. Er prüft die Gültigkeit des Eingangssignals anhand eines vordefinierten Bereichs und berechnet bei gültigem Signal den Ausgangswert durch Multiplikation mit einem Skalierungsfaktor und Addition eines Offsets. Bei ungültigem Signal wird der Ausgang auf 0 gesetzt und die Gültigkeitsflagge auf FALSE gesetzt.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung
INIT	EInit	Initialisierungsanforderung; übergibt Skalierungsparameter
REQ	Event	Normale Ausführungsanforderung zur Verarbeitung des Eingangs

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung
INITO	EInit	Bestätigung der Initialisierung
CNF	Event	Bestätigung der Ausführung; liefert skalierten Ausgang und Gültigkeitsflagge

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung
IN	DWORD	NOT_AVAILABLE_DWM	Eingangswert vom Feldbus
SCALE	LREAL	LREAL#1.0	Skalierungsfaktor (Multiplikator)
OFFSET	DINT	DINT#0	Offset, der nach der Skalierung addiert wird

Daten-Ausgänge

Name	Typ	Initialwert	Beschreibung
OUT	LREAL	LREAL#0.0	Skalierter Ausgangswert
VALID	BOOL	FALSE	TRUE, wenn das Eingangssignal gültig ist

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock besitzt zwei Zustände: **INIT** und **REQ**.

- **INIT-Zustand**: Wird durch das Ereignis INIT ausgelöst. Der zugehörige Algorithmus ist leer, d.h. es werden keine weiteren Aktionen durchgeführt. Der Ausgang INITO bestätigt den Abschluss der Initialisierung.
- **REQ-Zustand**: Wird durch das Ereignis REQ ausgelöst. Der Algorithmus REQ führt folgende Schritte aus:
  1. Prüfung, ob der als UDINT interpretierte Eingangswert IN kleiner oder gleich der Konstanten `VALID_SIGNAL_DW` ist.
  2. Ist diese Bedingung erfüllt, gilt das Signal als gültig:
     - `OUT` wird berechnet als: `UDINT_TO_LREAL(DWORD_TO_UDINT(IN)) * SCALE + DINT_TO_LREAL(OFFSET)`
     - `VALID` wird auf `TRUE` gesetzt.
  3. Andernfalls (Signal ungültig):
     - `OUT` wird auf `0.0` gesetzt.
     - `VALID` wird auf `FALSE` gesetzt.

4. Anschließend wird das Ereignis CNF ausgelöst, das die Ergebnisse (OUT, VALID) bereitstellt.

Die Konstanten `NOT_AVAILABLE_DWM` und `VALID_SIGNAL_DW` stammen aus importierten Bibliotheken und legen fest, welche DWORD-Werte als „nicht verfügbar“ bzw. als „gültig“ betrachtet werden.

Technische Besonderheiten

- Der Funktionsblock verwendet eine Bereichsprüfung mittels Vergleich von `DWORD_TO_UDINT(IN)` mit `VALID_SIGNAL_DW`. Dabei wird vorausgesetzt, dass gültige Feldbussignale innerhalb eines bestimmten Zahlenbereichs liegen.
- Die Konvertierung erfolgt ohne Gleitkommafehleranfälligkeit, da zuerst die Umwandlung von DWORD in UDINT und dann in LREAL erfolgt.
- Der Skalierungsfaktor `SCALE` und der Offset `OFFSET` können während der Initialisierung oder Laufzeit gesetzt werden, werden aber nur im REQ-Algorithmus verwendet.
- Der Baustein ist als SimpleFB implementiert und eignet sich für zyklische Verarbeitung.

Zustandsübersicht

Zustand	Auslöser	Aktion	Ausgang
INIT	INIT-Ereignis	Algorithmus INIT (leer)	INITO
REQ	REQ-Ereignis	Algorithmus REQ (Berechnung und Gültigkeitsprüfung)	CNF

Der Funktionsblock benötigt keine Zustandsübergänge zwischen INIT und REQ – beide Zustände werden direkt durch ihre jeweiligen Ereignisse gestartet.

Anwendungsszenarien

- **Landwirtschaftliche Feldbus-Steuerungen**: Umwandlung von rohen DWORD-Sensorwerten (z.B. Drehzahlen, Drücke) in physikalische Einheiten unter Berücksichtigung von Sensorcharakteristiken.
- **Skalierung von analogen Werten**: Wenn ein Feldbusgerät einen 32-Bit-Rohwert liefert, der erst mit einem Faktor und Offset in eine normierte Größe (z.B. 0-100%) umgerechnet werden muss.
- **Signalvalidierung**: Nur gültige Signale (innerhalb eines definierten Bereichs) werden weiterverarbeitet; bei ungültigen Werten wird ein definierter Nullwert ausgegeben.

Vergleich mit ähnlichen Bausteinen

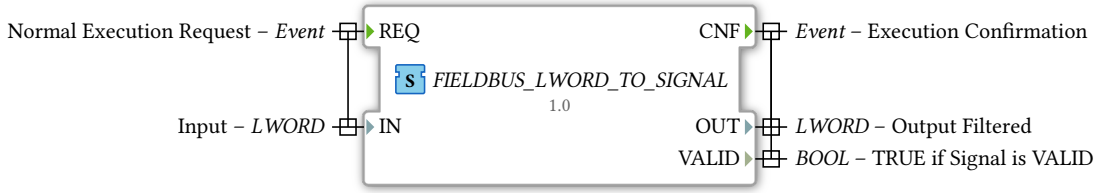
- **FIELDBUS_DWORD_TO_SIGNAL**: Ein ähnlicher Baustein ohne Skalierungsmöglichkeit, der den Rohwert direkt als LREAL ausgibt.
- **FIELDBUS_DWORD_TO_SIGNAL_SCALED** bietet zusätzlich die Parameter `SCALE` und `OFFSET` für flexible Umrechnung.
- **Ereignisgesteuerte Konverter**: Andere Bausteine nutzen möglicherweise eine andere Validierungslogik (z.B. Bitmasken) oder unterstützen unterschiedliche Feldbustypen.

Der vorliegende Baustein kombiniert Signalvalidierung mit einer einfachen linearen Transformation, was ihn für viele Anwendungen geeignet macht.

Fazit

Der `FIELDBUS_DWORD_TO_SIGNAL_SCALED` ist ein kompakter und wiederverwendbarer Funktionsblock zur Verarbeitung von Feldbussignalen. Er ermöglicht eine zuverlässige Gültigkeitsprüfung und gleichzeitige Skalierung in einem Schritt. Durch die klare Trennung von Initialisierung und Verarbeitung eignet er sich sowohl für Einmal- als auch für zyklische Aufrufe in Automatisierungssystemen, insbesondere in der Landtechnik.

FIELDBUS_LWORD_TO_SIGNAL



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **FIELDBUS_LWORD_TO_SIGNAL** dient der Umwandlung eines LWORD-Wertes aus einem Feldbussystem in ein gültiges Signal. Er übernimmt eine Eingangsgröße und gibt diese nur dann unverändert weiter, wenn ihr Wert innerhalb eines definierten gültigen Bereichs liegt. Andernfalls wird ein Nullwert ausgegeben und die Gültigkeit als `FALSE` signalisiert.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
REQ	Event	Normale Ausführungsanforderung (mit Daten-Eingang `IN`)

Ereignis-Ausgänge

Name	Typ	Kommentar
CNF	Event	Ausführungsbestätigung (mit Daten-Ausgängen `OUT` und `VALID`)

Daten-Eingänge

Name	Typ	Initialwert	Kommentar
IN	LWORD	`NOT_AVAILABLE_LWM`	Eingangswert vom Feldbus

Hinweis: `NOT_AVAILABLE_LWM` ist eine im Projekt definierte Konstante (z. B. `LWORD#16#FFFFFFFFFFFFFFFF`), die einen ungültigen oder nicht verfügbaren Wert repräsentiert.

Daten-Ausgänge

Name	Typ	Initialwert	Kommentar
OUT	LWORD	`LWORD#16#0000000000000000`	Gefilterter Ausgangswert (0 bei ungültigem Signal)
VALID	BOOL	`FALSE`	Gültigkeitsflag (`TRUE` wenn Signal gültig)

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Baustein wird durch ein Ereignis am Eingang `REQ` aktiviert. Dabei wird der aktuelle Wert von `IN` ausgelesen und gemäß folgender Logik verarbeitet:

1. Der eingehende LWORD-Wert wird in eine vorzeichenlose Ganzzahl (ULINT) konvertiert.
2. Dieser Wert wird mit einer ebenfalls als ULINT interpretierten Konstanten `VALID_SIGNAL_LW` verglichen.
   - Ist der Eingangswert **kleiner oder gleich** dieser Schwelle, gilt das Signal als gültig:
     - `OUT` erhält den Wert von `IN`.
     - `VALID` wird auf `TRUE` gesetzt.
   - Andernfalls (Wert größer als Schwelle) wird das Signal als ungültig betrachtet:
     - `OUT` wird auf Null (`LWORD#16#0000000000000000`) gesetzt.
     - `VALID` wird auf `FALSE` gesetzt.
3. Nach Abschluss der Verarbeitung wird das Ereignis `CNF` ausgegeben, um die Fertigstellung zu bestätigen.

Die Konstante `VALID_SIGNAL_LW` definiert die Obergrenze des gültigen Wertebereichs. Übliche Belegungen sind beispielsweise `LWORD#16#0000FFFFFFFFFFFF` oder ähnliche Bereiche, die vom Feldbusprotokoll vorgegeben werden.

Technische Besonderheiten

- **Typkonvertierung:** Der Vergleich erfolgt nach Umwandlung beider LWORD-Werte in ULINT mittels `LWORD_TO_ULINT`. Dadurch werden die Werte als vorzeichenlose 64-Bit-Ganzzahlen interpretiert, was einen konsistenten Vergleich unabhängig von bitmusterspezifischen Interpretationen ermöglicht.
- **Verwendung projektweiter Konstanten:** Die Werte `NOT_AVAILABLE_LWM` (Initialwert für `IN`) und `VALID_SIGNAL_LW` (Grenzwert) werden über Importe aus einem gemeinsamen Signalverarbeitungs-Package bezogen. Sie sind nicht im Baustein selbst festgelegt und können somit global angepasst werden.
- **Keine weitere Zustandsmaschine:** Der Baustein besitzt nur einen einzigen Zustand (`REQ`), der nach Ausführung des Algorithmus sofort wieder verlassen wird. Es gibt keine internen Verzögerungen oder Warteschleifen.

Zustandsübersicht

Zustand	Beschreibung	Aktion
REQ	Wartet auf ein Ereignis am Eingang `REQ`.	Führt den Algorithmus `REQ` aus (Prüfung und ggf. Filterung) und sendet anschließend das Ereignis `CNF`.

Der Baustein kehrt nach jeder Ausführung in den Zustand `REQ` zurück, sodass er nach jeder Verarbeitung sofort wieder bereit für einen neuen Auftrag ist.

Anwendungsszenarien

- **Feldbus-Signalvalidierung:** Einsatz in Automatisierungssystemen, bei denen analoge oder digitale Signale über einen Feldbus (z. B. PROFIBUS, CANopen) empfangen werden. Der Baustein filtert Werte, die außerhalb des zulässigen Bereichs liegen (z. B. Fehler- oder Initialisierungswerte), und gibt stattdessen einen definierten Nullwert aus.
- **Sicherheitsgerichtete Anwendungen:** Wenn das Feldbusprotokoll bestimmte Bitmuster für „Signal nicht verfügbar“ oder „Fehler“ vorsieht, können diese durch die Grenzwertprüfung erkannt und als ungültig markiert werden.
- **Datenvorverarbeitung vor logischen Verknüpfungen:** Der Ausgang `VALID` kann direkt als Freigabe für nachfolgende Funktionsbausteine verwendet werden, sodass diese nur bei gültigen Eingangssignalen aktiviert werden.

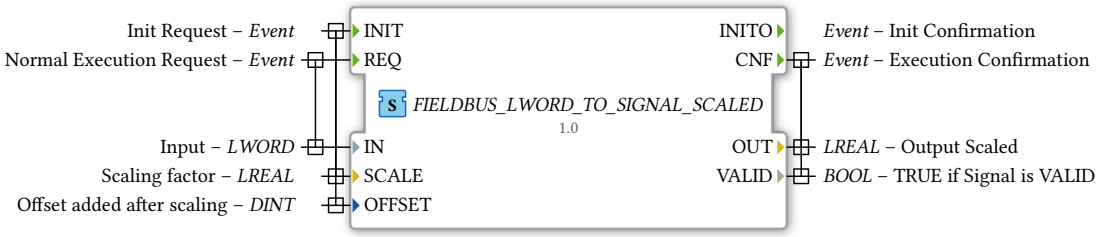
Vergleich mit ähnlichen Bausteinen

- **Einfache Filter (z. B. `FIELDBUS_LWORD_FILTER`):** Diese führen meist eine reine Grenzwertprüfung ohne Gültigkeitsflag durch. Der hier beschriebene Baustein bietet zusätzlich den booleschen Ausgang `VALID`, der die Entscheidungsfindung in der Steuerung erleichtert.
- **Signalumsetzer (z. B. `LWORD_TO_REAL`):** Diese konvertieren den Datentyp, prüfen aber nicht die Gültigkeit des Ursprungswerts. Der Baustein `FIELDBUS_LWORD_TO_SIGNAL` kombiniert Validierung und optionale Konvertierung (hier als 1:1-Durchleitung bei Gültigkeit).
- **Wächterbausteine (z. B. `WATCHDOG`):** Diese überwachen lediglich die Aktualität des Signals, nicht dessen Wertebereich. Der vorliegende Baustein ergänzt die Wertebereichsprüfung.

Fazit

Der Funktionsblock `FIELDBUS_LWORD_TO_SIGNAL` ist ein einfaches, aber effektives Werkzeug zur Aufbereitung von Feldbus-LWORD-Signalen. Durch die Kombination von Bereichsprüfung und Ausgabe eines Gültigkeitsflags wird eine saubere Trennung von gültigen und ungültigen Werten erreicht. Die Verwendung projektweiter Konstanten sorgt für Flexibilität, während die klare Zustandslogik eine zuverlässige und schnelle Verarbeitung gewährleistet. Er eignet sich besonders für Anwendungen, in denen Fehlersignale oder Initialisierungswerte sicher erkannt und weitergeleitet werden müssen.

FIELDBUS_LWORD_TO_SIGNAL_SCALED



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **FIELDBUS_LWORD_TO_SIGNAL_SCALED** dient der Umwandlung eines als **LWORD** vorliegenden Feldbus-Rohsignals in einen skalierten **LREAL**-Ausgangswert. Er spiegelt das Eingangssignal auf den Ausgang, sofern das Signal gültig ist, und wendet dabei eine lineare Skalierung an. Der Baustein ist für den Einsatz in der industriellen Automatisierung, insbesondere im Kontext von Feldbussystemen, konzipiert.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung	Mitgeführte Daten
INIT	EInit	Initialisierungsanforderung: Skalierungsfaktor (SCALE) und Offset (OFFSET) setzen	SCALE, OFFSET
REQ	Event	Normale Ausführungsanforderung: Verarbeitung des Eingangssignals (IN)	IN

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung	Mitgeführte Daten
INITO	EInit	Bestätigung der Initialisierung	–
CNF	Event	Bestätigung der Ausführung	OUT, VALID

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung
IN	LWORD	NOT_AVAILABLE_LWM	Feldbus-Rohsignal (LWORD)
SCALE	LREAL	LREAL#1.0	Skalierungsfaktor (Multiplikation)
OFFSET	DINT	DINT#0	Offset, der nach der Skalierung addiert wird

Daten-Ausgänge

Name	Typ	Initialwert	Beschreibung
OUT	LREAL	LREAL#0.0	Skalierter Ausgangswert (LREAL)
VALID	BOOL	FALSE	Gibt an, ob das Eingangssignal gültig ist (TRUE = gültig)

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Baustein besitzt zwei Algorithmen, die durch die Ereignisse **INIT** und **REQ** ausgelöst werden.

- **INIT** (bei Ereignis `INIT`): Der Algorithmus ist leer, setzt jedoch den Ausgangs-Ereignis `INITO`. Dies ermöglicht die Initialisierung der Skalierungsparameter (SCALE und OFFSET) über den INIT-Aufruf, ohne dass eine sofortige Verarbeitung stattfindet.
- **REQ** (bei Ereignis `REQ`): Führt die eigentliche Signalverarbeitung durch:
  1. Das Eingangssignal `IN` (LWORD) wird als vorzeichenloser Ganzzahlwert interpretiert (`LWORD_TO_ULINT`).
  2. Dieser Wert wird mit einem intern als gültig definierten Grenzwert **VALID_SIGNAL_LW** (ebenfalls LWORD, importierte Konstante) verglichen. Der Vergleich erfolgt auf ULINT-Ebene.
  3. **Gültigkeit**: Ist `IN` ≤ `VALID_SIGNAL_LW`, so gilt das Signal als gültig:
     - `OUT = (ULINT_TO_LREAL(IN)) * SCALE + DINT_TO_LREAL(OFFSET)`
     - `VALID = TRUE`
  4. **Ungültigkeit**: Andernfalls wird das Signal als ungültig betrachtet:
     - `OUT = 0.0`
     - `VALID = FALSE`
  5. Nach der Berechnung wird der Ausgangs-Ereignis `CNF` gesendet.

Die Skalierung erfolgt also linear: `OUT = IN_als_LREAL * SCALE + OFFSET`. Der Standardwert von `SCALE` ist 1.0, der von `OFFSET` ist 0, sodass ohne Anpassung eine identische Übertragung stattfindet.

Technische Besonderheiten

- **Gültigkeitsprüfung**: Der Baustein nutzt zwei importierte Konstanten:
  - `NOT_AVAILABLE_LWM`: Initialwert für den Eingang `IN` (kennzeichnet „nicht verfügbar“).
  - `VALID_SIGNAL_LW`: Grenzwert, bis zu dem ein Signal als gültig gilt (inklusive). Der Vergleich erfolgt durch Umwandlung beider LWORD-Werte in ULINT, sodass auch negative Werte (als Bitmuster) korrekt interpretiert werden.
- **Typkonvertierung**: Der Rohwert wird von LWORD über ULINT (64-Bit) in LREAL konvertiert, wobei potenzielle Präzisionsverluste bei sehr großen ULINT-Werten zu beachten sind.
- **Skalierungsparameter**: `SCALE` (LREAL) und `OFFSET` (DINT) werden nur während des INIT-Ereignisses gesetzt, bleiben dann aber für alle folgenden REQ-Aufrufe unverändert.
- **Ereignissteuerung**: Der Baustein hat eine einfache Zustandsmaschine mit zwei Zuständen und erfordert eine explizite Initialisierung vor der ersten Verarbeitung.

Zustandsübersicht

Zustand	Ausgelöst durch	Ausgabealgorithmus	Ereignisausgang	Beschreibung
INIT	Ereignis INIT	INIT	INITO	Initialisierung: Skalierungsparameter übernehmen
REQ	Ereignis REQ	REQ	CNF	Signalverarbeitung: Skalierung und Gültigkeitsprüfung

Der Baustein wechselt zwischen diesen beiden Zuständen nur durch die entsprechenden Ereignisse. Es gibt keinen expliziten Übergang nach der Initialisierung in einen Wartezustand; der Baustein bleibt so lange im letzten Zustand, bis ein neues Ereignis eintrifft.

Anwendungsszenarien

- **Analogwerte aus dem Feldbus**: Ein LWORD-Rohwert von einem analogen Eingangsmodul (z. B. 0…10 V, skaliert als Rohwert) wird mit Hilfe von SCALE und OFFSET in einen physikalischen Wert (z. B. Druck, Temperatur) umgerechnet.
- **Gültigkeitsfilter**: Signale, die außerhalb des zulässigen Bereichs liegen (z. B. Drahtbruch oder Überlauf), werden durch die Gültigkeitsprüfung erkannt und als ungültig markiert, während der Ausgang auf 0 gesetzt wird.
- **Initialisierung von Parametern**: Die Skalierung kann beim Start der Steuerung über das INIT-Ereignis konfiguriert werden, ohne dass die eigentliche Verarbeitung bereits läuft.

Vergleich mit ähnlichen Bausteinen

- **FIELDBUS_LREAL_TO_SIGNAL_SCALED**: Ein analoger Baustein für LREAL-Eingänge, der eine ähnliche Gültigkeitsprüfung und Skalierung bietet.

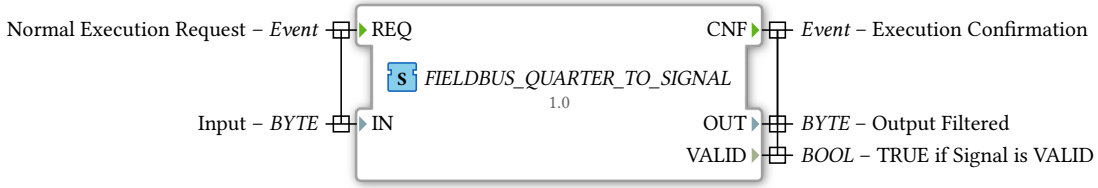
- **FIELDBUS_SINT_TO_SIGNAL_SCALED**: Für vorzeichenbehaftete 8-Bit-Werte.
- **Einfache Skalierer ohne Gültigkeitsprüfung**: Bausteine wie `SCALE` oder `MUL_ADD` bieten nur die mathematische Operation, jedoch keine Feldbus-spezifische Gültigkeitserkennung.

Der vorliegende Baustein kombiniert die Gültigkeitsprüfung (über den Grenzwert `VALID_SIGNAL_LW`) mit einer linearen Skalierung und eignet sich daher besonders für Feldbus-Anwendungen, bei denen die Gültigkeit des Signals vor der Weiterverarbeitung sichergestellt werden muss.

Fazit

FIELDBUS_LWORD_TO_SIGNAL_SCALED ist ein kompakter, aber leistungsfähiger Funktionsblock zur Verarbeitung von Feldbus-Signalen im LWORD-Format. Er bietet eine einfache Skalierung, eine integrierte Gültigkeitsprüfung und eine saubere Trennung von Initialisierung und Ausführung. Durch die Verwendung standardisierter Importe (`VALID_SIGNAL_LW`) lässt er sich leicht in bestehende Bibliotheken integrieren und an spezifische Feldbusprotokolle anpassen. Der Baustein ist unter der Eclipse Public License 2.0 verfügbar und eignet sich für den Einsatz in industriellen Steuerungssystemen mit IEC 61499-konformen Laufzeitumgebungen.

FIELDBUS_QUARTER_TO_SIGNAL



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `FIELDBUS_QUARTER_TO_SIGNAL` dient als Signalvalidierungs- und Durchleitungsbaustein. Er prüft, ob ein eingehender BYTE-Wert `IN` innerhalb eines gültigen Bereichs liegt, und gibt ihn in diesem Fall unverändert an `OUT` weiter. Zusätzlich wird ein Boolesches Signal `VALID` gesetzt, das den Gültigkeitsstatus des Ausgangs anzeigt. Der Baustein eignet sich zur Filterung von Feldbussignalen, bei denen bestimmte Werte als ungültig oder „don't care" markiert sind.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar	Mit Variablen
`REQ`	Event	Normaler Ausführungsauftrag	`IN`

Ereignis-Ausgänge

Name	Typ	Kommentar	Mit Variablen
`CNF`	Event	Ausführungsbestätigung	`OUT`, `VALID`

Daten-Eingänge

Name	Typ	Kommentar	Initialwert
`IN`	`BYTE`	Eingangssignal	`NOT_AVAILABLE_2bit` (importierte Konstante)

Daten-Ausgänge

Name	Typ	Kommentar	Initialwert
`OUT`	`BYTE`	Gefiltertes Ausgangssignal	`16#00`
`VALID`	`BOOL`	`TRUE`, wenn das Signal gültig ist	`FALSE`

Adapter

Keine Adapter vorhanden.

Funktionsweise

Bei einem Ereignis am Eingang `REQ` wird der Algorithmus `REQ` ausgeführt:

1. Der eingehende Wert `IN` wird mit der importierten Konstanten `DONT_CARE_2bit` verglichen.
2. Ist `IN` kleiner als `DONT_CARE_2bit`, wird der Wert als gültig betrachtet:
   - `OUT` erhält den Wert von `IN`.
   - `VALID` wird auf `TRUE` gesetzt.
3. Andernfalls (wenn `IN` größer oder gleich `DONT_CARE_2bit` ist) wird das Signal als ungültig eingestuft:
   - `OUT` wird auf `BYTE#0` zurückgesetzt.
   - `VALID` wird auf `FALSE` gesetzt.
4. Nach der Verarbeitung wird das Ereignis `CNF` ausgelöst.

Der Vergleich nutzt dabei die Semantik der importierten Konstanten: `DONT_CARE_2bit` definiert die Grenze, ab der Werte als „egal“ oder ungültig gelten. Der Initialwert von `IN` ist auf `NOT_AVAILABLE_2bit` gesetzt, der typischerweise oberhalb dieser Grenze liegt, sodass der Baustein beim Start einen definierten ungültigen Zustand ausgibt.

Technische Besonderheiten

- **Konstantenimport**: Die Werte `DONT_CARE_2bit` und `NOT_AVAILABLE_2bit` werden aus einem übergeordneten Signalverarbeitungs-Paket importiert. Sie müssen im Zielsystem definiert sein und sinnvolle Schwellenwerte repräsentieren (z. B. `16#FC` für don’t care).
- **Einfache Zustandslogik**: Der Baustein besitzt nur einen einzigen EC-Zustand `REQ`. Jedes Ereignis führt sofort zur Berechnung und Ausgabe.
- **Kompatibilität**: Entwickelt gemäß IEC 61499-1, einsetzbar in der 4diac-IDE und anderen konformen Laufzeitumgebungen.

Zustandsübersicht

Es existiert nur ein Zustand:

Zustand	Beschreibung	Ausgabeereignis	Ausgabeaktionen
`REQ`	Verarbeitet den Eingang und aktualisiert Ausgänge	`CNF`	`OUT` und `VALID` setzen

Anwendungsszenarien

- **Feldbus-Überwachung**: Ein Sensor liefert auf einem Feldbus (z. B. CANopen, PROFIBUS) 8-Bit-Daten, wobei bestimmte Werte als „nicht verfügbar“ oder „Fehler“ codiert sind. Der Baustein extrahiert die gültigen Messwerte und zeigt deren Gültigkeit an.
- **Signalpreprocessing**: Vor der Weiterverarbeitung in einer Steuerung werden ungültige Telegrammteile herausgefiltert, sodass nachfolgende Blöcke nur mit plausiblen Daten arbeiten.
- **Gateway zwischen Bussystemen**: Übersetzung von Don’t-Care-Konventionen eines Busses auf ein einheitliches Gültigkeitssignal für das Zielsystem.

Vergleich mit ähnlichen Bausteinen

- **`FIELDBUS_QUARTER_TO_SIGNAL`** ist spezialisiert auf die Unterscheidung zwischen gültigen und ungültigen Werten mittels eines festen Schwellwerts.
- Ein allgemeiner **Wertfilter** (z. B. `F_BAND`) könnte dagegen einen unteren und oberen Grenzwert abfragen.
- Ein **Gültigkeitsprüfer** (z. B. `VALIDITY_CHECK`) setzt oft zusätzlich Quality-of-Service-Flags. Der vorliegende Baustein besticht durch seine einfache, auf Feldbustypen zugeschnittene Logik.

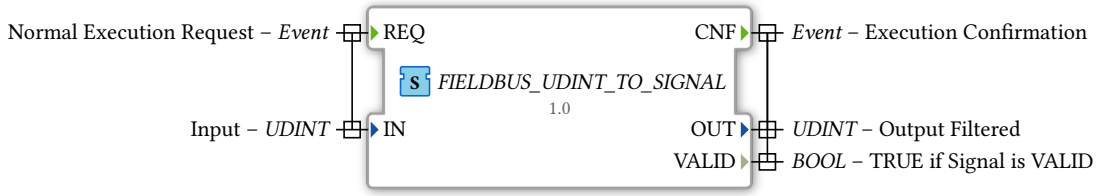
Fazit

`FIELDBUS_QUARTER_TO_SIGNAL` ist ein kompakter, effizienter Funktionsblock zur Validierung von 8-Bit-Feldbussignalen. Er trennt zuverlässig gültige Messwerte von als ungültig markierten Werten und stellt die Gültigkeit als separates Boolesches Signal bereit. Dank seiner schlanken Zustandslogik und der Verwendung importierter Konstanten lässt er sich flexibel in unterschiedliche Signalverarbeitungsketten integrieren.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

FIELDBUS_UDINT_TO_SIGNAL



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **FIELDBUS_UDINT_TO_SIGNAL** dient dazu, einen eingehenden UDINT-Wert auf Gültigkeit zu prüfen und diesen bei bestandener Prüfung unverändert an den Ausgang weiterzuleiten. Ist der Wert ungültig, wird der Ausgang auf einen definierten Fehlerwert (0) gesetzt und die Gültigkeitsanzeige deaktiviert. Der Baustein wird typischerweise in der Feldbus-Signalverarbeitung eingesetzt, um nur valide Mess- oder Steuerwerte an nachfolgende Logik weiterzugeben.

Schnittstellenstruktur

Ereignis-Eingänge

Event	Beschreibung
REQ	Normaler Ausführungsanforderung; löst die Verarbeitung des Eingangs IN aus.

Ereignis-Ausgänge

Event	Beschreibung
CNF	Bestätigt die abgeschlossene Verarbeitung. Wird nach dem Algorithmus ausgelöst.

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung
IN	UDINT	`DWORD_TO_UDINT(NOT_AVAILABLE_DWM)`	Der zu prüfende Eingangswert.

Daten-Ausgänge

Name	Typ	Initialwert	Beschreibung
OUT	UDINT	`16#00000000`	Der gefilterte Ausgangswert. Bei gültigem Signal entspricht er IN, sonst 0.
VALID	BOOL	FALSE	TRUE, wenn das Signal gültig ist, d.h. IN innerhalb des gültigen Bereichs liegt.

Adapter

Keine.

Funktionsweise

Sobald ein Ereignis am Eingang **REQ** eintrifft, wird der Algorithmus ausgeführt:

1. Der Wert von **IN** wird mit dem vordefinierten Grenzwert `VALID_SIGNAL_DW` verglichen.
2. Falls **IN** kleiner oder gleich diesem Grenzwert ist, wird **OUT** auf **IN** gesetzt und **VALID** auf `TRUE`.
3. Andernfalls wird **OUT** auf `0` (UDINT) gesetzt und **VALID** auf `FALSE`.

Der Grenzwert `VALID_SIGNAL_DW` sowie der Initialwert `NOT_AVAILABLE_DWM` sind als Konstanten aus dem Paket `eclipse4diac::signalprocessing::FIELDBUS_SIGNAL` importiert. Sie repräsentieren einen maximal zulässigen Signalwert bzw. einen „nicht verfügbar“-Marker.

Technische Besonderheiten

- **Datentyp** – Der Baustein arbeitet ausschließlich mit dem Typ `UDINT` (Unsigned Double Integer, 32 Bit).
- **Initialwert** – Der Eingang IN wird standardmäßig auf den Wert `NOT_AVAILABLE_DWM` gesetzt. Dieser wird über die Funktion `DWORD_TO_UDINT` aus einem vorher definierten DWORD konvertiert.
- **Grenzwert** – Die Gültigkeit wird durch einen einfachen Vergleich mit der importierten Konstanten `VALID_SIGNAL_DW` bestimmt. Alle Werte ≤ diesem Grenzwert gelten als gültig.
- **Einfache Zustandslogik** – Der FB besteht aus genau einem Zustand (REQ) und führt bei jeder Anforderung den Algorithmus aus.

Zustandsübersicht

Der Funktionsblock ist als einfacher FB (SimpleFB) realisiert und besitzt nur einen einzigen Zustand:

Zustand	Aktion	Ausgangsereignis
REQ	Algorithmus REQ	CNF

Es findet keine Zustandsmaschine mit mehreren Schritten oder Verzweigungen statt.

Anwendungsszenarien

- **Feldbus-Signalaufbereitung** – In Automatisierungssystemen, bei denen UDINT-Werte über einen Feldbus eingehen, wird der Baustein genutzt, um ungültige oder außerhalb des gültigen Bereichs liegende Werte zu filtern (z. B. bei Sensordaten, die einen Maximalwert nicht überschreiten dürfen).
- **Landtechnik** – Ursprünglich für den Einsatz in landwirtschaftlichen Steuerungen entwickelt (z. B. HR Agrartechnik), um analoge Messwerte wie Drehzahlen oder Drücke auf Gültigkeit zu prüfen.
- **Qualitätssicherung** – Vorverarbeitung von Messdaten, bevor sie in Regelkreise oder Protokollierungen einfließen.

Vergleich mit ähnlichen Bausteinen

- **FIELDBUS_DINT_TO_SIGNAL** – Analoger Baustein für vorzeichenbehaftete 32-Bit-Werte (DINT).
- **FIELDBUS_BOOL_TO_SIGNAL** – Vergleichbarer Baustein für boolesche Signale; dort entfällt die Schwellwertprüfung.
- **FIELDBUS_REAL_TO_SIGNAL** – Für Gleitkommazahlen, mit gegebenenfalls anderer Toleranzlogik.

Der vorliegende Baustein ist speziell auf den Datentyp UDINT zugeschnitten und nutzt einen einfachen absoluten Grenzwert zur Validierung.

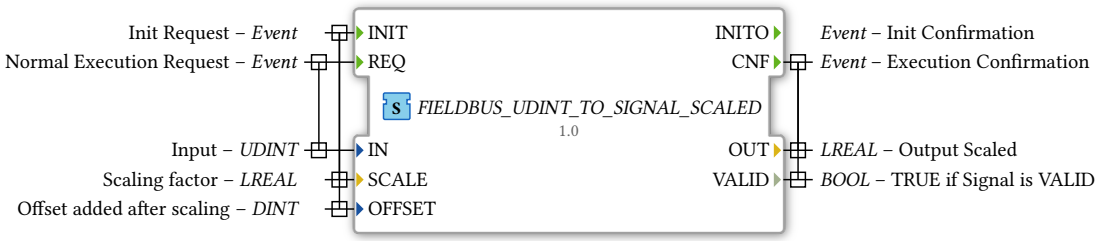
Fazit

FIELDBUS_UDINT_TO_SIGNAL ist ein kompakter und effizienter Funktionsblock zur Filterung von UDINT-Feldbussignalen. Durch die klare Trennung in gültige und ungültige Werte sowie die einfache Schnittstelle eignet er sich gut für robuste Automatisierungslösungen. Die Verwendung von importierten Konstanten erhöht die Wiederverwendbarkeit und Anpassbarkeit an verschiedene Feldbusprotokolle.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

FIELDBUS_UDINT_TO_SIGNAL_SCALED



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock FIELDBUS_UDINT_TO_SIGNAL_SCALED dient der Aufbereitung eines über den Feldbus empfangenen UDINT-Wertes. Er spiegelt den Eingang IN auf den Ausgang OUT wider, jedoch nur dann, wenn das eingehende Signal als gültig erkannt wird. Zusätzlich wird der Wert mit einem Faktor skaliert und einem Offset versehen. Dieser Baustein wird typischerweise in der Signalverarbeitungskette von Feldbusprotokollen eingesetzt, um rohe Integer-Werte in physikalische Größen (z. B. als LREAL) umzurechnen.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Mit Parametern	Beschreibung
INIT	EInit	SCALE, OFFSET	Initialisierung – setzt Skalierungsfaktor und Offset.
REQ	Event	IN	Normales Ausführungsereignis – verarbeitet den aktuellen Eingangswert.

Ereignis-Ausgänge

Ereignis	Typ	Mit Parametern	Beschreibung
INITO	EInit	–	Bestätigung der erfolgreichen Initialisierung.
CNF	Event	OUT, VALID	Bestätigung der Verarbeitung – liefert den skalierten Wert und das Gültigkeitssignal.

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung
IN	UDINT	DWORD_TO_UDINT(NOT_AVAILABLE_DWM)	Rohwert aus dem Feldbus.
SCALE	LREAL	LREAL#1.0	Multiplikativer Skalierungsfaktor.
OFFSET	DINT	DINT#0	Additiver Offset (nach Skalierung).

Daten-Ausgänge

Name	Typ	Initialwert	Beschreibung
OUT	LREAL	LREAL#0.0	Skalierter Ausgabewert.
VALID	BOOL	FALSE	TRUE, wenn der Eingangswert als gültig erkannt wurde.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Baustein besitzt zwei Zustände: INIT und REQ.

- **INIT-Zustand**: Bei einem INIT-Ereignis wird der leere Algorithmus ausgeführt (keine Aktion) und anschließend INITO ausgegeben. Die Parameter SCALE und OFFSET werden dabei an den Baustein übergeben, aber erst bei der nächsten REQ-Verarbeitung verwendet.
- **REQ-Zustand**: Bei einem REQ-Ereignis wird folgende Logik ausgeführt:
  1. Der eingehende Wert IN wird mit einem vordefinierten Schwellwert VALID_SIGNAL_DW (aus dem Import eclipse4diac::signalprocessing::FIELDBUS_SIGNAL::VALID_SIGNAL_DW) verglichen.
  2. Wenn IN <= DWORD_TO_UDINT(VALID_SIGNAL_DW) gilt, wird der Eingang als gültig betrachtet. Dann wird OUT berechnet als: OUT := UDINT_TO_LREAL(IN) * SCALE + DINT_TO_LREAL(OFFSET) und VALID auf TRUE gesetzt.

3. Andernfalls wird `OUT` auf `0.0` und `VALID` auf `FALSE` gesetzt. Dies signalisiert, dass der übergebene Feldbuswert ungültig ist (z. B. Fehlerwert oder Kommunikationsabbruch).

Der Initialwert von `IN` ist voreingestellt auf `DWORD_TO_UDINT(NOT_AVAILABLE_DWM)`, einer Konstanten aus demselben Import, die einen speziellen „nicht verfügbar"-Wert repräsentiert.

Technische Besonderheiten

- Die Gültigkeitsprüfung basiert auf zwei importierten Konstanten:
  - `VALID_SIGNAL_DW`: Obergrenze für gültige Werte.
  - `NOT_AVAILABLE_DWM`: Standardwert für nicht verfügbare Signale (wird als Initialwert für `IN` verwendet).
- Die Skalierung erfolgt in LREAL (doppelte Genauigkeit), sodass auch sehr große oder kleine Eingangswerte präzise verarbeitet werden können.
- Der Offset ist als `DINT` definiert, wird aber vor der Addition in `LREAL` konvertiert. Der gesamte Berechnungsbereich wird durch die LREAL-Darstellung abgedeckt.

Zustandsübersicht

Der Baustein verwendet ein einfaches Zustandsdiagramm mit zwei Zuständen:

- **INIT**: Wird nach dem `INIT`-Ereignis durchlaufen. Der Algorithmus `INIT` ist leer, der Ausgang `INITO` wird getriggert.
- **REQ**: Wird nach dem `REQ`-Ereignis durchlaufen. Hier wird der Algorithmus `REQ` ausgeführt, der die Skalierung und Gültigkeitsprüfung vornimmt, und anschließend `CNF` ausgegeben.

Es gibt keine weiteren Zustände oder Schleifen.

Anwendungsszenarien

- **Skalierung von Sensorwerten**: Ein Feldbusgerät liefert einen UDINT-Rohwert (z. B. 0 … 65535). Mit `SCALE` wird eine physikalische Größe (z. B. Druck in bar) umgerechnet, der Offset berücksichtigt Nullpunktverschiebungen.
- **Gültigkeitsprüfung**: Typische Feldbusprotokolle definieren bestimmte Bitmuster als „ungültig". Der Baustein filtert diese Werte automatisch heraus und setzt `VALID = FALSE`.
- **Initialisierung**: Beim Systemstart kann über `INIT` der Skalierungsfaktor gesetzt werden, während der `REQ`-Zyklus regelmäßig die aktuellen Prozesswerte verarbeitet.

Vergleich mit ähnlichen Bausteinen

Ähnliche Bausteine sind oft allgemeine `SCALE`-Funktionen, die keine Gültigkeitsprüfung enthalten. Dieser Baustein fügt eine explizite Validierung des Feldbussignals hinzu. Andere Implementierungen könnten den Gültigkeitsbereich als Parameter oder über separate Schwellwerte konfigurierbar machen. Der vorliegende Baustein setzt feste, in der Import-Bibliothek definierte Konstanten ein, was die Konfiguration vereinfacht, aber weniger flexibel ist.

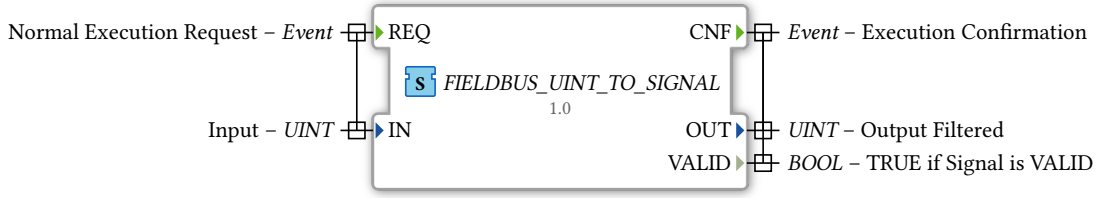
Fazit

Der `FIELDBUS_UDINT_TO_SIGNAL_SCALED`-Funktionsblock bietet eine robuste und einfache Möglichkeit, Rohdaten aus einem Feldbus zu skalieren und gleichzeitig die Signalqualität zu bewerten. Durch die Kombination von Skalierungsfaktor und Offset sowie der integrierten Gültigkeitserkennung eignet er sich besonders für industrielle Automatisierungsanwendungen, bei denen sowohl die Umrechnung als auch die Plausibilitätsprüfung der Messwerte erforderlich ist.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

FIELDBUS_UINT_TO_SIGNAL



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein FIELDBUS_UINT_TO_SIGNAL dient der Verarbeitung von Feldbussignalen. Er spiegelt einen eingehenden UINT-Wert auf den Ausgang, sofern dieser als gültig eingestuft wird. Die Gültigkeit wird anhand eines Vergleichs mit einem konfigurierten Schwellwert (VALID_SIGNAL_W) bestimmt. Liegt der Eingangswert über diesem Schwellwert, wird der Ausgang zurückgesetzt und das Signal als ungültig markiert.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Datentyp	Kommentar
REQ	Event	Normaler Ausführungsanstoß (verbunden mit IN)

Ereignis-Ausgänge

Ereignis	Datentyp	Kommentar
CNF	Event	Bestätigung der Ausführung (verbunden mit OUT und VALID)

Daten-Eingänge

Name	Datentyp	Initialwert	Kommentar
IN	UINT	WORD_TO_UINT(NOT_AVAILABLE_WM)	Eingangswert (rohes Feldbussignal)

Daten-Ausgänge

Name	Datentyp	Initialwert	Kommentar
OUT	UINT	16#0000	Gefilterter Ausgangswert
VALID	BOOL	FALSE	TRUE, wenn das Signal als gültig erkannt wurde

Adapter

Keine.

Funktionsweise

Bei jedem REQ-Ereignis wird der Algorithmus REQ ausgeführt:

1. Der eingehende Wert IN wird mit der Konstanten `VALID_SIGNAL_W` verglichen.
2. Ist `IN <= WORD_TO_UINT(VALID_SIGNAL_W)`, so gilt das Signal als gültig:
   - `OUT` erhält den Wert von `IN`.
   - `VALID` wird auf TRUE gesetzt.
3. Andernfalls (Signal ungültig):
   - `OUT` wird auf 0 zurückgesetzt.
   - `VALID` wird auf FALSE gesetzt.
4. Anschließend wird das Ereignis CNF ausgelöst.

Die Konstanten `NOT_AVAILABLE_WM` (für den Initialwert von IN) und `VALID_SIGNAL_W` (als Schwellwert) werden aus externen Importen bezogen und definieren den gültigen Bereich des Feldbussignals.

Technische Besonderheiten

- Verwendung von Typkonvertierungen (`WORD_TO_UINT`) für den Vergleich mit den importierten Konstanten.
- Der Baustein importiert zwei Konstanten aus dem Paket `eclipse4diac::signalprocessing::FIELDBUS_SIGNAL`: `NOT_AVAILABLE_WM` und `VALID_SIGNAL_W`. Diese müssen in der Anwendung entsprechend definiert oder bereitgestellt werden.
- Der Funktionsbaustein ist als einfacher FB (Simple FB) realisiert, d. h. er besitzt genau einen Ausführungszustand.

Zustandsübersicht

Der Baustein besteht aus einem einzigen Zustand **REQ**.

- **START → REQ**: Der Algorithmus wird ausgeführt, sobald das Ereignis REQ eintrifft. Nach Abarbeitung wird CNF ausgegeben und der Zustand bleibt in REQ (zyklische Verarbeitung).

Anwendungsszenarien

- Verarbeitung von analogen Sensorwerten aus einem Feldbus, bei denen ein bestimmter Wertebereich als gültig definiert ist (z. B. 0–1000 für einen Drucksensor).
- Filterung von Kommunikationsfehlern oder ungültigen Telegrammen, indem Werte außerhalb des gültigen Bereichs auf Null gesetzt und als ungültig markiert werden.
- Vorverarbeitung von Feldbussignalen vor der Weiterleitung an nachfolgende Logikbausteine.

Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu einfachen Durchleitbausteinen (z. B. `MOVE`) führt `FIELDBUS_UINT_TO_SIGNAL` eine bewusste Gültigkeitsprüfung durch und unterscheidet zwischen gültigen und ungültigen Werten. Andere Bausteine wie `LIMIT` oder `SCALE` bieten zwar Bereichsbegrenzungen, jedoch keine separate Bool-Ausgabe für die Gültigkeit. Dieser Baustein eignet sich besonders für Anwendungen, die eine explizite Signalzustandsanzeige benötigen.

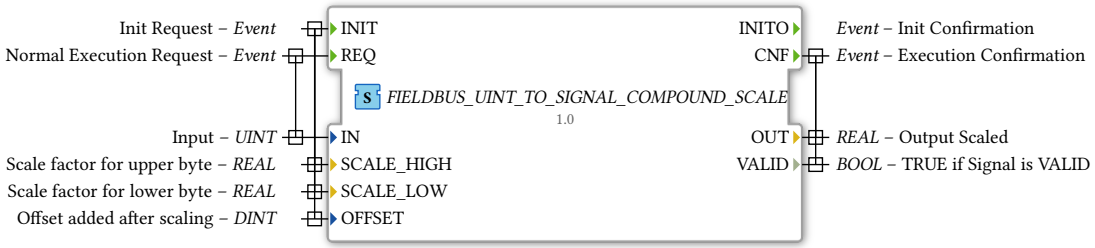
Fazit

Der Funktionsbaustein FIELDBUS_UINT_TO_SIGNAL bietet eine einfache und effiziente Möglichkeit, Feldbussignale auf Gültigkeit zu prüfen und entweder durchzureichen oder zu verwerfen. Die klare Schnittstelle und die Verwendung von konfigurierbaren Konstanten machen ihn flexibel einsetzbar in IEC-61499-Anwendungen der Automatisierungstechnik.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

FIELDBUS_UINT_TO_SIGNAL_COMPOUND_SCALE



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `FIELDBUS_UINT_TO_SIGNAL_COMPOUND_SCALE` dient zur Aufbereitung eines 16-Bit-Feldbussignals (Typ `UINT`) in einen skalierten Realwert. Dabei wird das eingehende Wort in ein oberes und ein unteres Byte zerlegt, jedes Byte mit einem eigenen Skalierungsfaktor multipliziert und ein Offset addiert. Zusätzlich wird ein Gültigkeitssignal (`VALID`) ausgegeben, das auf einem vorgegebenen

Schwellwert basiert. Der FB eignet sich besonders für Protokolle, die zwei Messgrößen (z. B. Temperatur und Druck) in einem einzigen Register kodieren.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung
`INIT`	EInit	Initialisierungsanforderung (Parameterübernahme)
`REQ`	Event	Normale Ausführungsanforderung (Verarbeitung von `IN`)

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung
`INITO`	EInit	Bestätigung der Initialisierung
`CNF`	Event	Bestätigung der Ausführung (nach `REQ`)

Daten-Eingänge

Variable	Typ	Initialwert	Beschreibung
`IN`	UINT	`WORD_TO_UINT(NOT_AVAILABLE_WM)`	Rohwert vom Feldbus
`SCALE_HIGH`	REAL	`0.256`	Skalierungsfaktor für das obere Byte
`SCALE_LOW`	REAL	`0.001`	Skalierungsfaktor für das untere Byte
`OFFSET`	DINT	`0`	Offset, der nach der Skalierung addiert wird (in REAL)

Daten-Ausgänge

Variable	Typ	Initialwert	Beschreibung
`OUT`	REAL	`0.0`	Skalierter Ausgabewert
`VALID`	BOOL	`FALSE`	`TRUE`, wenn das Eingangssignal als gültig erkannt wurde

Adapter

Keine.

Funktionsweise

1. **Initialisierung** (`INIT`): Der Algorithmus `INIT` ist leer, die Parameter `SCALE_HIGH`, `SCALE_LOW` und `OFFSET` werden jedoch über die `With`-Beziehung mit dem Ereignis verknüpft und können so beim Aufruf von `INIT` gesetzt werden.
2. **Normale Verarbeitung** (`REQ`):
   - Der Eingangswert `IN` wird mit einer Konstanten `VALID_SIGNAL_W` (aus dem importierten Namen `eclipse4diac::signalprocessing::FIELDBUS_SIGNAL::VALID_SIGNAL_W`) verglichen: `IF (IN <= WORD_TO_UINT(VALID_SIGNAL_W)) THEN`
   - Falls die Bedingung **wahr** ist (Signal gültig):
     - Das obere Byte wird durch Rechts-Shift um 8 Bit und Maskierung mit `0x00FF` extrahiert: `highByte := (IN >> 8) & 0xFF`
     - Das untere Byte wird direkt durch Maskierung gewonnen: `lowByte := IN & 0xFF`
     - Beide Bytes werden in `REAL` gewandelt, mit den jeweiligen Skalierungsfaktoren multipliziert und der Offset addiert: `OUT := highByte * SCALE_HIGH + lowByte * SCALE_LOW + OFFSET`
     - `VALID` wird auf `TRUE` gesetzt.

- Falls die Bedingung **falsch** ist (ungültiges Signal):
  - `OUT` wird auf `0.0` gesetzt.
  - `VALID` wird auf `FALSE` gesetzt.
- Anschließend wird das Ereignis `CNF` ausgegeben.

Technische Besonderheiten

- **Compound-Skalierung**: Der FB erlaubt getrennte Skalierung für das obere und untere Byte eines 16-Bit-Wortes. Dies ist nützlich, wenn zwei unterschiedliche physikalische Größen in einem Register zusammengefasst sind.
- **Gültigkeitsprüfung**: Der Vergleich mit `VALID_SIGNAL_W` definiert einen Schwellwert. Werte oberhalb dieses Schwellwerts werden als ungültig betrachtet – typisch für Feldbus-Fehlerkodierungen (z. B. „Not Available“).
- **Parametrierung über `INIT`**: Die Skalierungsfaktoren und der Offset können einmalig über das `INIT`-Ereignis gesetzt werden, sodass sie während des Betriebs konstant bleiben.
- **Initialwerte**: Die vorgegebenen Werte (`SCALE_HIGH=0.256`, `SCALE_LOW=0.001`, `OFFSET=0`) sind als Beispielwerte anzusehen; sie sollten an die konkrete Anwendung angepasst werden.

Zustandsübersicht

Der FB besitzt zwei EC-Zustände:

- **INIT**: Führt den leeren Algorithmus `INIT` aus und verlässt den Zustand über den Ausgang `INITO`.
- **REQ**: Führt den Algorithmus `REQ` aus, der die gesamte Signalverarbeitung durchführt, und verlässt den Zustand über den Ausgang `CNF`.

Der FB ist als SimpleFB implementiert; es gibt keine weiteren Zustände wie IDLE oder WAIT.

Anwendungsszenarien

- **Zwei-in-eins-Sensorwerte**: Ein Temperatur- und ein Drucksensor liefern ihre Werte als je 8-Bit-Zahlen in einem 16-Bit-Register. Der FB wandelt diese mit getrennten Skalierungen in physikalische Einheiten um.
- **Fehlererkennung im Feldbus**: Wenn das Feldbusprotokoll für ungültige Messwerte einen bestimmten Bit-Code (z. B. alle Bits 1) vorsieht, kann über den Schwellwert `VALID_SIGNAL_W` die Gültigkeit erkannt werden.
- **Nachverarbeitung von Logging-Daten**: Rohdaten aus einem Feldbus-Logger werden mit den ursprünglichen Skalierungen wieder in die physikalischen Größen zurückgerechnet.

Vergleich mit ähnlichen Bausteinen

Baustein	Eigenschaft	Unterschied
`SCALE` (z. B. nach IEC 61131)	Linearer Faktor und Offset auf den gesamten Wert	Keine Aufteilung in Bytes, keine Gültigkeitsprüfung
`LINEAR` (Feldbus-Standard)	Gleitender Mittelwert oder einfache lineare Transformation	Kein Compound-Ansatz
`UINT_TO_REAL`	Direkte Typumwandlung	Keine Skalierung, keine Fehlererkennung

Dieser FB kombiniert als einziger die Byte-trennung mit einer zweifachen Skalierung und einer integrierten Gültigkeitserkennung, was ihn für spezielle Feldbus-Protokolle mit zusammengesetzten Registern besonders geeignet macht.

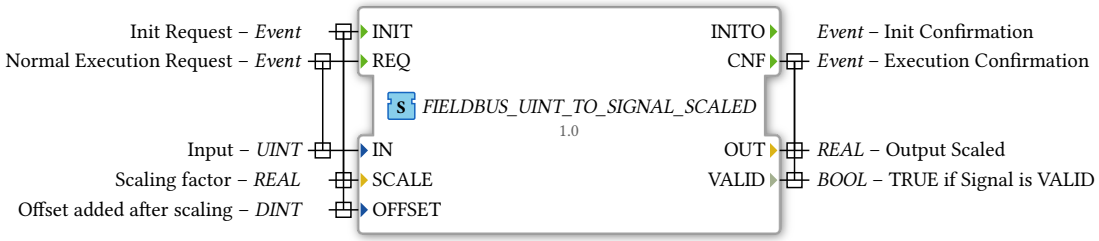
Fazit

Der `FIELDBUS_UINT_TO_SIGNAL_COMPOUND_SCALE` ist ein leistungsfähiger Funktionsblock zur Aufbereitung von 16-Bit-Feldbussignalen, die zwei unabhängige Messwerte in einem Register kodieren. Durch die getrennte Skalierung von High- und Low-Byte sowie die integrierte Gültigkeitsprüfung lassen sich komplexe Sensor-Register effizient und zuverlässig in physikalische Größen umrechnen. Die Parametrierung über das `INIT`-Ereignis und die klare Trennung von Initialisierung und Betrieb machen den FB wiederverwendbar und anpassbar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

FIELDBUS_UINT_TO_SIGNAL_SCALED



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **FIELDBUS_UINT_TO_SIGNAL_SCALED** dient der Aufbereitung eines über einen Fieldbus empfangenen UINT-Wertes. Er prüft die Gültigkeit des Eingangssignals und skaliert dieses bei gültigem Wert mittels einer Multiplikationskonstante (SCALE) und eines additiven Offsets (OFFSET) zu einem REAL-Ausgang. Bei ungültigem Signal wird der Ausgang auf Null gesetzt und ein Validitäts-Flag auf FALSE gesetzt. Typische Anwendung ist die Umrechnung roher Feldbus-Daten in physikalische Größen mit gleichzeitiger Erkennung von Ausnahmezuständen wie Drahtbruch oder Sensorfehlern.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Initialisierungsanfrage. Mit diesem Ereignis werden die Parameter `SCALE` und `OFFSET` gesetzt.
- **REQ** (Event): Normale Ausführungsanfrage. Mit diesem Ereignis wird der aktuelle Wert `IN` verarbeitet.

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung. Wird nach Verarbeitung des INIT-Ereignisses gesendet.
- **CNF** (Event): Bestätigung der Ausführung. Wird nach Verarbeitung des REQ-Ereignisses gesendet und transportiert die Ausgangsdaten `OUT` und `VALID`.

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung
IN	UINT	WORD_TO_UINT(NOT_AVAILABLE_WM)	Rohwert vom Feldbus, der skaliert werden soll.
SCALE	REAL	REAL#1.0	Skalierungsfaktor (Multiplikation).
OFFSET	DINT	DINT#0	Additiver Offset, der nach der Skalierung addiert wird.

Daten-Ausgänge

Name	Typ	Initialwert	Beschreibung
OUT	REAL	REAL#0.0	Skalierter Ausgangswert. Bei ungültigem Signal = 0.0.
VALID	BOOL	FALSE	Gültigkeitsflag: TRUE wenn das Eingangssignal gültig ist.

Adapter

Keine.

Funktionsweise

Der FB besitzt zwei Betriebszustände, die durch die entsprechenden Ereignisse gesteuert werden.

- **INIT-Zustand**: Beim Empfang von `INIT` werden die Parameter `SCALE` und `OFFSET` übernommen (der Algorithmus ist leer, die Übergabe erfolgt implizit über die `With`-Verknüpfung). Anschließend wird das Ereignis `INITO` gesendet.
- **REQ-Zustand**: Beim Empfang von `REQ` wird folgender Algorithmus ausgeführt:
  1. Prüfung, ob der Eingangswert `IN` kleiner oder gleich einer fest definierten Gültigkeitsschwelle ist. Diese Schwelle ist durch die importierte Konstante `VALID_SIGNAL_W` gegeben (aus dem Namespace `eclipse4diac::signalprocessing::FIELDBUS_SIGNAL`).
  2. **Gültig (`IN <= VALID_SIGNAL_W`)**:
     - `OUT := UINT_TO_REAL(IN) * SCALE + DINT_TO_REAL(OFFSET)`
     - `VALID := TRUE`
  3. **Ungültig (`IN > VALID_SIGNAL_W`)**:
     - `OUT := 0.0`
     - `VALID := FALSE`
  4. Anschließend wird das Ereignis `CNF` mit den aktuellen Werten von `OUT` und `VALID` gesendet.

Die Gültigkeitsschwelle wird als Teil der importierten Bibliothek bereitgestellt und muss entsprechend der verwendeten Feldbus-Norm definiert sein. Der initiale Wert von `IN` ist `WORD_TO_UINT(NOT_AVAILABLE_WM)`, wodurch bei fehlender Datenübergabe standardmäßig ein ungültiges Signal erkannt wird.

Technische Besonderheiten

- **Verwendung von importierten Konstanten**: Die Gültigkeitsprüfung erfolgt anhand von Konstanten aus dem Namespace `eclipse4diac::signalprocessing::FIELDBUS_SIGNAL`. Diese müssen im Zielsystem korrekt definiert sein (z. B. `VALID_SIGNAL_W` als maximaler gültiger UINT-Wert, `NOT_AVAILABLE_WM` als Kennzeichnung für „nicht verfügbar“).
- **Leerer INIT-Algorithmus**: Das INIT-Ereignis dient lediglich der Parameterübergabe. Eine explizite Initialisierungslogik existiert nicht – die Werte von `SCALE` und `OFFSET` werden direkt aus den Daten-Eingängen übernommen. Daher können diese Parameter auch vor dem ersten INIT gesetzt sein (siehe Initialwerte).
- **Typerhaltung**: Der Eingang ist UINT, der Ausgang REAL. Die Skalierung erfolgt mit REAL-Arithmetik nach einer expliziten Typumwandlung.
- **Keine Zustandsautomaten-Logik**: Der FB ist als SimpleFB realisiert, das nur die zwei Zustände INIT und REQ kennt. Es gibt keine zeitlichen oder wiederholten Abläufe; jede Ausführung erfolgt streng ereignisgesteuert.

Zustandsübersicht

Der FB besitzt genau zwei EC-States:

Zustand	Eingang	Algorithmus	Ausgangsereignis
INIT	INIT	– (leer)	INITO
REQ	REQ	REQ	CNF

Es existiert keine Zustandsschleife oder Verweildauer. Der FB startet nach der Initialisierung im Zustand INIT und kann danach beliebig oft durch REQ ausgeführt werden. Ein erneutes INIT setzt die Parameter zurück.

Anwendungsszenarien

- **Skalierung von Analogwerten**: Ein Feldbus liefert einen rohen 16-Bit-Messwert (z. B. 0–4095). Mit `SCALE = 0.001` und `OFFSET = -5.0` wird daraus z. B. ein Druck in bar.
- **Erkennung von Fehlerwerten**: Viele Feldbus-Protokolle definieren bestimmte Bit-Muster für „Sensorfehler“ oder „Drahtbruch“. Der FB erkennt solche Werte, indem er sie über der Gültigkeitsschwelle (z. B. 0xFFFE oder 0x8000) einstuft und den Ausgang auf Null setzt.
- **Initialisierungssequenz**: Nach Systemstart kann ein INIT-Aufruf die Skalierungsparameter aus einer Konfigurationsdatei laden. Danach werden in schneller Folge REQ-Aufrufe zur zyklischen Messwertverarbeitung gesendet.

Vergleich mit ähnlichen Bausteinen

- **FIELDBUS_UINT_TO_SIGNAL (ohne Skalierung)**: Ein einfacherer FB, der den UINT-Wert direkt als REAL ausgibt und nur die Gültigkeitsprüfung durchführt. Die Skalierung muss extern erfolgen.

- **FIELDBUS_UINT_TO_SIGNAL_FILTERED**: Zusätzlich zur Skalierung wird ein Tiefpass oder Medianfilter angewendet – geeignet für verrauschte Signale.
- **FIELDBUS_INT_TO_SIGNAL_SCALED**: Analoger FB für vorzeichenbehaftete Eingänge (INT), mit entsprechender Anpassung der Gültigkeitslogik.

Der vorliegende FB zeichnet sich durch seine Einfachheit, klare Trennung von Skalierung und Gültigkeitserkennung sowie die Verwendung von importierten Konstanten aus, die eine flexible Anpassung an verschiedene Feldbus-Normen erlaubt.

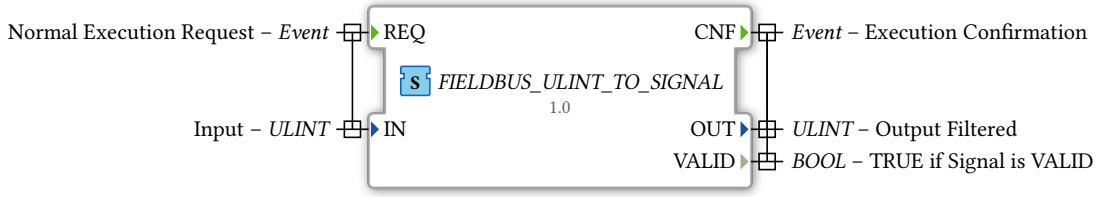
Fazit

FIELDBUS_UINT_TO_SIGNAL_SCALED ist ein kompakter, ereignisgesteuerter Funktionsblock zur Umrechnung eines UINT-Feldbussignals in einen skalierten REAL-Wert. Durch die integrierte Gültigkeitsprüfung wird eine einfache Fehlererkennung ermöglicht. Der Baustein eignet sich besonders für Anwendungen, in denen rohe Busdaten mit geringem Aufwand in physikalische Größen gewandelt werden müssen und gleichzeitig die Signalqualität überwacht werden soll. Die klare Schnittstelle und die Verwendung von importierten Konstanten erleichtern die Wiederverwendung in verschiedenen Projektkontexten.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

FIELDBUS_ULINT_TO_SIGNAL



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **FIELDBUS_ULINT_TO_SIGNAL** dient als Filterbaustein für Feldbussignale vom Typ `ULINT`. Er prüft, ob ein eingehender Wert innerhalb des gültigen Signalbereichs liegt und gibt diesen – je nach Ergebnis – entweder unverändert aus oder setzt den Ausgang auf einen definierten Ungültigkeitswert. Das VALID-Ausgangssignal zeigt den Status der Signalvalidität an.

Der Baustein ist besonders geeignet für den Einsatz in sicherheitskritischen oder qualitätsüberwachenden Applikationen, bei denen ungültige oder fehlerhafte Feldbuswerte zuverlässig erkannt und behandelt werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung
REQ	Normaler Ausführungsanforderung – löst die Verarbeitung des Eingangswerts IN aus. Der Ereignisausgang CNF wird nach Abschluss der Aktion gesendet.

Ereignis-Ausgänge

Ereignis	Beschreibung
CNF	Bestätigung der Ausführung – wird nach der Berechnung von OUT und VALID gesendet.

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung

IN	ULINT	`LWORD_TO_ULINT(NOT_AVAILABLE_LWM)`	Eingangssignal (Rohwert vom Feldbus)

Hinweis: `NOT_AVAILABLE_LWM` ist eine aus der Bibliothek `FIELDBUS_SIGNAL` importierte Konstante, die einen ungültigen oder nicht verfügbaren Signalwert repräsentiert.

Daten-Ausgänge

Name	Typ	Initialwert	Beschreibung
OUT	ULINT	`16#0000000000000000`	Gefilterter Ausgangswert – entspricht IN, wenn gültig, andernfalls 0.
VALID	BOOL	`FALSE`	Signalvalidität – `TRUE` bei gültigem Eingang, sonst `FALSE`.

Adapter

Der Baustein besitzt keine Adapter-Schnittstellen.

Funktionsweise

Nach einem Ereignis am Eingang **REQ** wird der folgende Algorithmus ausgeführt:

1. Prüfung: **IN** ≤ `LWORD_TO_ULINT(VALID_SIGNAL_LW)`
   - `VALID_SIGNAL_LW` ist eine importierte Konstante aus der `FIELDBUS_SIGNAL`-Bibliothek und definiert die obere Grenze des gültigen Wertebereichs.
2. **Fall – Eingang gültig** (Bedingung erfüllt):
   - `OUT` := `IN`
   - `VALID` := `TRUE`
3. **Fall – Eingang ungültig** (Bedingung nicht erfüllt):
   - `OUT` := `0`
   - `VALID` := `FALSE`

Anschließend wird der Ereignisausgang **CNF** ausgelöst.

Technische Besonderheiten

- Der Baustein arbeitet deterministisch und benötigt nur eine einzige Verarbeitungsstufe.
- Die Schwellwerte `NOT_AVAILABLE_LWM` und `VALID_SIGNAL_LW` sind als externe Konstanten definiert und zentral in der Bibliothek `FIELDBUS_SIGNAL` hinterlegt. Sie können ohne Änderung des Bausteins angepasst werden.
- Der Ausgangswert bei ungültigem Signal ist auf 0 gesetzt, was in vielen Feldbussystemen einen definierten „Ungültigkeitswert“ darstellt.
- Der initiale Wert von `IN` wird als „nicht verfügbar“ deklariert, sodass der Baustein auch ohne vorheriges Ereignis einen definierten Zustand liefert.

Zustandsübersicht

Der Funktionsblock ist als **SimpleFB** realisiert und besitzt nur einen einzigen Zustand:

Zustand	Beschreibung
REQ	Eingangszustand: Wartet auf Ereignis REQ. Bei Eintritt wird der Algorithmus `REQ` ausgeführt und anschließend der Ausgang `CNF` gesendet. Danach verbleibt der Baustein im selben Zustand.

Es existieren keine weiteren Zustände oder Verzweigungen – die Funktionsweise ist rein ablauforientiert.

Anwendungsszenarien

- **Feldbus-Signalaufbereitung:** Ein Sensor liefert einen ULINT-Wert, der nur innerhalb eines bestimmten Bereichs physikalisch sinnvoll ist (z. B. Drehzahl, Druck). Der Baustein filtert ungültige Messungen heraus.
- **Sicherheitskette:** In einer sicherheitsgerichteten Steuerung werden fehlerhafte oder abweichende Feldbuswerte erkannt und durch einen definierten Ersatzwert (0) ersetzt, gleichzeitig wird ein Gültigkeitssignal für die nachgeschaltete Logik bereitgestellt.

- **Qualitätsüberwachung:** Der VALID-Ausgang kann zur Überwachung der Datenintegrität verwendet werden, z. B. um Alarme auszulösen oder auf alternative Signalquellen umzuschalten.

Vergleich mit ähnlichen Bausteinen

Im 4diac-Framework existieren weitere Filterbausteine für andere Datentypen, z. B. `FIELDBUS_DINT_TO_SIGNAL` oder `FIELDBUS_REAL_TO_SIGNAL`. Gemeinsam ist allen das Prinzip der Gültigkeitsprüfung mittels einer vordefinierten Obergrenze. Der vorliegende Baustein ist speziell auf den `ULINT`-Datentyp zugeschnitten, was einen effizienten und typsicheren Einsatz in Applikationen mit 64-Bit-unsigned-Werten ermöglicht.

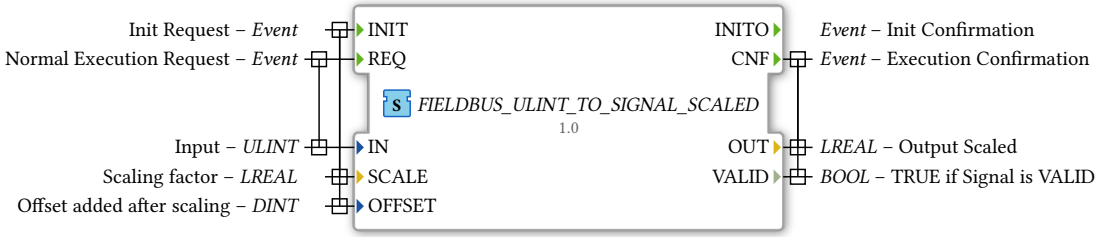
Fazit

Der **FIELDBUS_ULINT_TO_SIGNAL** ist ein einfacher, aber zuverlässiger Funktionsblock zur Signalvalidierung im Feldbusumfeld. Durch die klare Trennung in gültige und ungültige Werte sowie die Bereitstellung eines BOOL-Ausgangs für die Validität eignet er sich hervorragend für den Einsatz in modularen Automatisierungssystemen, bei denen Datenqualität und Fehlererkennung essenziell sind. Die Verwendung externer Konstanten erlaubt eine flexible Anpassung an unterschiedliche Signalbereiche ohne Änderung des Bausteins selbst.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

FIELDBUS_ULINT_TO_SIGNAL_SCALED



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **FIELDBUS_ULINT_TO_SIGNAL_SCALED** dient zur Konvertierung eines ULINT-Rohwerts (z. B. von einem Feldbus) in einen skalierten LREAL-Wert. Dabei wird die Gültigkeit des Eingangssignals anhand eines konstanten Schwellwerts (`VALID_SIGNAL_LW`) überprüft. Ist der Eingang gültig, erfolgt eine lineare Skalierung (Multiplikation mit einem Faktor und Addition eines Offsets); andernfalls wird der Ausgang auf 0,0 gesetzt und die Gültigkeit negiert.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung	Mit Variablen
INIT	EInit	Initialisierungsanforderung	SCALE, OFFSET
REQ	Event	Normale Ausführungsanforderung	IN

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung	Mit Variablen
INITO	EInit	Bestätigung der Initialisierung	–
CNF	Event	Bestätigung der Ausführung	OUT, VALID

Daten-Eingänge

Variable	Typ	Initialwert	Beschreibung
IN	ULINT	`LWORD_TO_ULINT(NOT_AVAILABLE_LWM)`	Rohwert vom Feldbus
SCALE	LREAL	`1.0`	Skalierungsfaktor
OFFSET	DINT	`0`	Offset, der nach der Skalierung addiert wird

Daten-Ausgänge

Variable	Typ	Initialwert	Beschreibung
OUT	LREAL	`0.0`	Skalierter Ausgangswert (nur gültig, wenn VALID = TRUE)
VALID	BOOL	`FALSE`	Gültigkeitssignal: TRUE, wenn IN ≤ `VALID_SIGNAL_LW`, sonst FALSE

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock besitzt zwei Betriebszustände, die durch die Ereignisse **INIT** und **REQ** aktiviert werden:

1. **INIT-Zustand** Beim Eintreffen des INIT-Ereignisses wird der Algorithmus `INIT` ausgeführt (derzeit leer). Anschließend wird das Ereignis **INITO** gesendet. Über die mit INIT verbundenen Eingänge können die Parameter **SCALE** und **OFFSET** gesetzt werden.
2. **REQ-Zustand** Bei einem REQ-Ereignis wird der Algorithmus `REQ` ausgeführt:
   - Prüfung: `IN ≤ LWORD_TO_ULINT(VALID_SIGNAL_LW)`
     - **Gültig** (Bedingung erfüllt): `OUT := ULINT_TO_LREAL(IN) * SCALE + DINT_TO_LREAL(OFFSET)` `VALID := TRUE`
     - **Ungültig** (Bedingung nicht erfüllt): `OUT := 0.0` `VALID := FALSE`
   - Anschließend wird das Ereignis **CNF** mit den neuen Werten von **OUT** und **VALID** ausgegeben.

Die Konstanten `NOT_AVAILABLE_LWM` und `VALID_SIGNAL_LW` stammen aus dem importierten Paket `eclipse4diac::signalprocessing::FIELDBUS_SIGNAL` und legen den Schwellwert für die Gültigkeitserkennung fest.

Technische Besonderheiten

- Der Baustein ist als **SimpleFB** mit zwei Zuständen implementiert, wodurch er deterministisch und ohne komplexe Zustandsautomaten arbeitet.
- Die Gültigkeitsprüfung erfolgt durch einen Vergleich mit dem importierten `VALID_SIGNAL_LW`. Dies ermöglicht eine anwendungsspezifische Konfiguration des gültigen Wertebereichs.
- Bei ungültigem Signal wird der Ausgang **OUT** auf 0,0 zurückgesetzt – eine definierte Fehlerbehandlung.
- Skalierung und Offset werden als LREAL bzw. DINT übergeben, der Offset wird intern in LREAL konvertiert, sodass auch negative Werte möglich sind.
- Der Initialwert von **IN** (`NOT_AVAILABLE_LWM`) ist so gewählt, dass er beim ersten Durchlauf als ungültig erkannt wird, falls kein gültiger Feldbuswert vorliegt.

Zustandsübersicht

Zustand	Beschreibung	Auslöser	Ausgabeereignis
INIT	Initialisierung (keine Aktion)	INIT	INITO
REQ	Verarbeitung des Eingangssignals (Skalierung oder Fehlerfall)	REQ	CNF

Anwendungsszenarien

- **Feldbus-Sensorik**: Ein 64-Bit-ULINT-Wert von einem Drucksensor wird mit einem Faktor (z. B. 0,001) in bar umgerechnet und ein Nullpunkt-Offset addiert. Die Gültigkeitsprüfung verhindert die Verarbeitung von Kommunikationsfehlern.
- **Fehlerbehandlung**: Bei Überschreitung des `VALID_SIGNAL_LW` (z. B. bei Sensorausfall) wird **VALID** auf FALSE gesetzt, sodass die übergeordnete Steuerung den Wert nicht übernimmt.
- **Initialisierung**: Beim Anlagenstart können über INIT die Skalierung und der Offset aus einem Parametersatz geladen werden.

Vergleich mit ähnlichen Bausteinen

- **FIELDBUS_SINT_TO_SIGNAL_SCALED** – gleiche Logik, jedoch für vorzeichenbehaftete 8-Bit-Werte (SINT). Der vorliegende Block verarbeitet 64-Bit-Unsigned-Werte.
- **FIELDBUS_ULINT_TO_SIGNAL (ohne Skalierung)** – einfache Typkonvertierung ohne Skalierung und ohne integrierte Gültigkeitsprüfung.
- **GENERIC_SCALER** – allgemeiner Skalierungsbaustein, bei dem die Gültigkeitsprüfung extern realisiert werden muss.

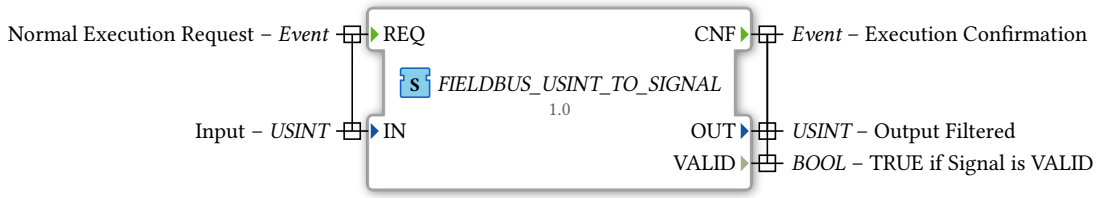
Fazit

Der Funktionsblock **FIELDBUS_ULINT_TO_SIGNAL_SCALED** bietet eine kompakte, parametrierbare Lösung zur sicheren Umwandlung von Feldbus-Rohdaten in physikalische Werte. Durch die Kombination von Gültigkeitsprüfung, Skalierung und Offset eignet er sich ideal für industrielle Automatisierungsaufgaben, bei denen Datenintegrität und einfache Kalibrierung gefordert sind.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

FIELDBUS_USINT_TO_SIGNAL



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock FIELDBUS_USINT_TO_SIGNAL dient der Aufbereitung eines USINT-Signals aus einem Feldbus. Er prüft, ob der eingehende Wert einen gültigen Signalbereich darstellt und spiegelt diesen gegebenenfalls auf den Ausgang. Andernfalls wird der Ausgang auf einen definierten Nullwert gesetzt und das Gültigkeitssignal zurückgesetzt.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung	Mit
REQ	Normaler Ausführungsauftrag	IN

Ereignis-Ausgänge

Ereignis	Beschreibung	Mit
CNF	Ausführungsbestätigung	OUT, VALID

Daten-Eingänge

Name	Typ	Kommentar	Initialwert
IN	USINT	Eingangswert vom Feldbus	BYTE_TO_USINT(NOT_AVAILABLE_B)

Daten-Ausgänge

Name	Typ	Kommentar	Initialwert
OUT	USINT	Gefilterter Ausgangswert	16#00
VALID	BOOL	TRUE, wenn Signal gültig	FALSE

Adapter

Keine.

Funktionsweise

Beim Eintreffen eines REQ-Ereignisses wird der Algorithmus ausgeführt. Der eingehende Wert IN wird mit dem konstanten Grenzwert `VALID_SIGNAL_B` (als USINT interpretiert) verglichen. Wenn `IN <= BYTE_TO_USINT(VALID_SIGNAL_B)` erfüllt ist, wird der Eingangswert direkt auf den Ausgang OUT übernommen und das Gültigkeitssignal VALID auf TRUE gesetzt. Andernfalls wird OUT auf 0 und VALID auf FALSE gesetzt. Das Ergebnis wird mit dem CNF-Ereignis quittiert.

Die beiden Konstanten `NOT_AVAILABLE_B` und `VALID_SIGNAL_B` sind Teil des Pakets `eclipse4diac::signalprocessing::FIELDBUS_SIGNAL` und definieren einen Referenzwert für „nicht verfügbar“ bzw. den oberen Grenzwert für gültige Signale.

Technische Besonderheiten

- Der Initialwert des Eingangs IN ist auf `BYTE_TO_USINT(NOT_AVAILABLE_B)` gesetzt, sodass bei erstmaligem Aufruf ohne gültigen Wert sofort ein ungültiger Zustand signalisiert wird.
- Der Vergleich verwendet die Funktion `BYTE_TO_USINT`, um den Byte-Wert der Konstanten in den Datentyp USINT zu konvertieren.
- Der Baustein ist unter der Eclipse Public License 2.0 lizenziert.

Zustandsübersicht

Der Funktionsblock besitzt einen einzigen Zustand `REQ`. Sobald ein REQ-Ereignis eintrifft, wird der Algorithmus ausgeführt und unmittelbar das CNF-Ereignis gesendet. Es gibt keine internen Zustandsänderungen oder Verzögerungen.

Anwendungsszenarien

- **Feldbus-Signalaufbereitung**: Ein Sensor liefert Werte im Bereich 0–127 als gültige Signale. Der Grenzwert VALID_SIGNAL_B wird auf 127 gesetzt. Werte darüber (z. B. 255 als Fehlercode) werden als ungültig erkannt.
- **Initialisierung**: Beim Systemstart ist der Eingang auf den „nicht verfügbar“-Wert vorbesetzt, sodass bis zum ersten gültigen Empfang der Ausgang auf 0 und VALID auf FALSE bleibt.
- **Fehlererkennung**: Durch Anpassung des Grenzwerts können beliebige zulässige Wertebereiche definiert werden.

Vergleich mit ähnlichen Bausteinen

Ähnliche Bausteine wie `FIELDBUS_BYTE_TO_SIGNAL` oder `FIELDBUS_SINT_TO_SIGNAL` arbeiten nach dem gleichen Prinzip, jedoch mit anderen Datentypen. Dieser Baustein spezialisiert sich auf USINT. Ein einfacher „Mirror“-Baustein ohne Gültigkeitsprüfung würde den Wert ungefiltert durchreichen.

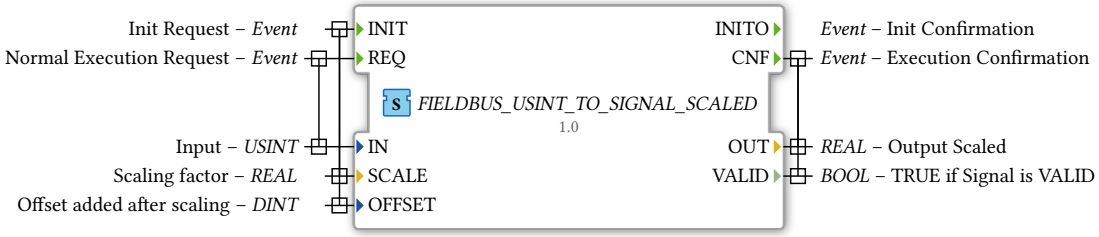
Fazit

Der `FIELDBUS_USINT_TO_SIGNAL`-Baustein bietet eine einfache und zuverlässige Möglichkeit, USINT-Feldbussignale auf ihre Gültigkeit zu prüfen und nur gültige Werte an die nachfolgende Verarbeitung weiterzugeben. Durch die Verwendung konfigurierbarer Grenzwerte (über die Paketkonstanten) ist er flexibel einsetzbar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

FIELDBUS_USINT_TO_SIGNAL_SCALED



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **FIELDBUS_USINT_TO_SIGNAL_SCALED** dient der Umwandlung eines eingehenden USINT-Signals in einen skalierten REAL-Wert unter Berücksichtigung einer Gültigkeitsprüfung. Er wird typischerweise in der Feldbus-Signalverarbeitung eingesetzt, um Rohwerte aus dem Feldbus (z. B. Sensordaten) mit einem Skalierungsfaktor und einem Offset zu versehen und parallel die Gültigkeit des Eingangssignals zu bewerten. Der Baustein ist nach dem Standard IEC 61499-1 aufgebaut.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung
`INIT`	EInit	Initialisierungsanfrage; mit den Parametern `SCALE` und `OFFSET` verknüpft.
`REQ`	Event	Normale Ausführungsanfrage; mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung
`INITO`	EInit	Bestätigung der Initialisierung.
`CNF`	Event	Bestätigung der Ausführung; mit den Ausgängen `OUT` und `VALID` verknüpft.

Daten-Eingänge

Name	Datentyp	Initialwert	Beschreibung
`IN`	USINT	`BYTE_TO_USINT(NOT_AVAILABLE_B)`	Eingangssignal als vorzeichenloser 8-Bit-Wert. Der Initialwert entspricht dem Signal „nicht verfügbar“.
`SCALE`	REAL	`REAL#1.0`	Skalierungsfaktor (Multiplikation).
`OFFSET`	DINT	`DINT#0`	Offset, der nach der Skalierung addiert wird.

Daten-Ausgänge

Name	Datentyp	Initialwert	Beschreibung
`OUT`	REAL	`REAL#0.0`	Skalierter Ausgangswert.
`VALID`	BOOL	`FALSE`	Gültigkeitsflag: `TRUE`, wenn das Eingangssignal gültig ist.

Adapter

Keine vorhanden.

Funktionsweise

Der Baustein arbeitet mit zwei Zuständen: `INIT` und `REQ`.

- Bei einem `INIT`-Ereignis werden die Parameter `SCALE` und `OFFSET` übernommen (der Algorithmus ist leer, die Werte werden jedoch durch das `With`-Binding eingelesen). Anschließend wird `INITO` ausgegeben.
- Bei einem `REQ`-Ereignis wird der Algorithmus `REQ` ausgeführt: Falls der Eingangswert `IN` kleiner oder gleich dem vordefinierten Grenzwert `BYTE_TO_USINT(VALID_SIGNAL_B)` ist, gilt das Signal als gültig. In diesem Fall wird der Wert wie folgt skaliert:

  ```
  OUT := USINT_TO_REAL(IN) * SCALE + DINT_TO_REAL(OFFSET)
  ```

  und `VALID` wird auf `TRUE` gesetzt. Überschreitet `IN` den Grenzwert (d. h. das Signal ist ungültig), werden `OUT` auf `0.0` und `VALID` auf `FALSE` gesetzt.

Das Verhalten spiegelt den Eingang 1:1 auf den Ausgang wider, sofern das Signal gültig ist – mit zusätzlicher Skalierung und Offset.

Technische Besonderheiten

- Die Konstanten `NOT_AVAILABLE_B` und `VALID_SIGNAL_B` stammen aus dem externen Paket `eclipse4diac::signalprocessing::FIELDBUS_SIGNAL`. Sie definieren die Grenzen zwischen gültigen und ungültigen USINT-Werten. `BYTE_TO_USINT(NOT_AVAILABLE_B)` ist der initiale Zustand des Eingangs und kennzeichnet ein „nicht verfügbares“ Signal.
- Der `INIT`-Algorithmus enthält keinen ST-Code – die Initialisierung beschränkt sich auf das Bereitstellen der Parameter über die Ereignisbindung. Der Baustein erwartet vor dem ersten `REQ` einen `INIT`-Aufruf, um Skalierung und Offset zu setzen.
- Die Skalierung erfolgt grundsätzlich als REAL-Arithmetik; der Offset wird deshalb von DINT nach REAL konvertiert.

Zustandsübersicht

Der Funktionsblock besitzt zwei elementare Zustände als einfacher FB:

Zustand	Aktion	Ausgangsereignis
`INIT`	Führe Algorithmus `INIT` aus (leer)	`INITO`
`REQ`	Führe Algorithmus `REQ` aus	`CNF`

Es existieren keine weiteren Wartezustände – jeder eingehende Event führt sofort zur Bearbeitung des zugehörigen Algorithmus und zur Ausgabe des zugehörigen Ausgangsereignisses.

Anwendungsszenarien

- **Feldbus-Sensorauswertung**: Ein Sensor liefert einen USINT-Rohwert (z. B. 0…200). Über `SCALE` und `OFFSET` wird daraus die physikalische Größe (z. B. Druck in bar) berechnet. Der Grenzwert `VALID_SIGNAL_B` trennt gültige Messwerte von Fehler- oder Initialisierungswerten.
- **Aktualisierung von Prozessvariablen**: In SPS- oder Steuerungsapplikationen wird der Baustein zyklisch mit `REQ` getriggert, um die neuesten Feldbusdaten zu verarbeiten und in den Steuerungskontext zu übernehmen.

Vergleich mit ähnlichen Bausteinen

- **`FIELDBUS_USINT_TO_SIGNAL`**: Ein einfacherer Baustein ohne Skalierung und Offset; er gibt den USINT-Wert direkt als REAL oder als anderen Datentyp aus.
- **`FIELDBUS_WORD_TO_SIGNAL_SCALED`**: Analoger Baustein für WORD-Eingänge – bietet eine höhere Auflösung, benötigt aber einen anderen Datentyp.
- **`FIELDBUS_USINT_TO_SIGNAL_SCALED`** zeichnet sich durch die Kombination aus Gültigkeitsprüfung (basierend auf der Konstante `VALID_SIGNAL_B`) und einfacher linearer Skalierung (Multiplikation + Addition) aus.

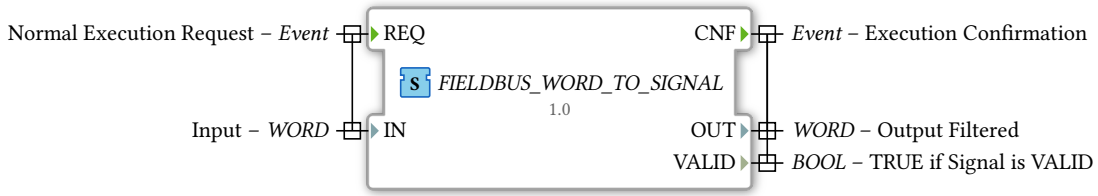
Fazit

Der Funktionsblock `FIELDBUS_USINT_TO_SIGNAL_SCALED` ist ein kompakter, aber leistungsfähiger Baustein zur Verarbeitung von 8-Bit-Feldbussignalen. Er vereint Gültigkeitserkennung, Skalierung und Offset in einer logischen Einheit und eignet sich besonders für Anwendungen, bei denen Rohwerte in physikalische Größen umgerechnet und gleichzeitig auf ihre Plausibilität geprüft werden müssen. Die einfache Zustandsmaschine und die klar definierten Schnittstellen machen ihn robust und einfach einsetzbar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

FIELDBUS_WORD_TO_SIGNAL



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock FIELDBUS_WORD_TO_SIGNAL dient der einfachen Signalfilterung in Feldbusanwendungen. Er spiegelt einen eingehenden WORD-Wert am Ausgang wider, sofern das Signal als gültig eingestuft wird. Ein boolescher Ausgang zeigt den Gültigkeitsstatus an. Die Implementierung erfolgt anhand eines Vergleichs mit einer extern definierten Grenzkonstante.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
REQ	Event	Normaler Ausführungsauftrag; getriggert durch den Eingang IN.

Ereignis-Ausgänge

Name	Typ	Kommentar
CNF	Event	Bestätigung der Ausführung; wird nach Abschluss der Verarbeitung ausgegeben.

Daten-Eingänge

Name	Typ	Initialwert	Kommentar
IN	WORD	NOT_AVAILABLE_WM	Eingangswert, der auf Gültigkeit geprüft wird.

Daten-Ausgänge

Name	Typ	Initialwert	Kommentar
OUT	WORD	16#0000	Gefilterter Ausgangswert (nur bei gültigem Signal).
VALID	BOOL	FALSE	TRUE, wenn das Eingangssignal als gültig erkannt wurde.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Baustein arbeitet mit einem einfachen Algorithmus:

1. Beim Eintreffen des Ereignisses **REQ** wird der eingehende Wert **IN** ausgelesen.
2. Der Wert wird mittels `WORD_TO_UINT` in einen vorzeichenlosen Integer konvertiert.
3. Dieser wird mit der externen Konstanten `VALID_SIGNAL_W` verglichen. – Ist der konvertierte Wert **kleiner oder gleich** `VALID_SIGNAL_W`, wird **IN** direkt an **OUT** übernommen und **VALID** auf `TRUE` gesetzt. – Andernfalls wird **OUT** auf `16#0000` zurückgesetzt und **VALID** auf `FALSE`.
4. Anschließend wird das Ereignis **CNF** ausgelöst.

Der Initialwert des Eingangs `IN` ist `NOT_AVAILABLE_WM`, sodass der Baustein im Grundzustand ein ungültiges Signal signalisiert.

Technische Besonderheiten

- **Externe Konstanten:** Die Vergleichsgrenze `VALID_SIGNAL_W` sowie der Standard-Unavailable-Wert `NOT_AVAILABLE_WM` werden aus dem Package `eclipse4diac::signalprocessing::FIELDBUS_SIGNAL` importiert. Ihre Werte sind außerhalb des Bausteins definiert.
- **Datentypkonvertierung:** Der Vergleich erfolgt nach der Umwandlung von `WORD` nach `UINT` (`WORD_TO_UINT`), um einen vorzeichenlosen numerischen Vergleich zu ermöglichen.
- **Initialwerte:** Der Ausgang `OUT` beginnt mit `16#0000` und `VALID` mit `FALSE`. Der Eingang `IN` startet mit dem als ungültig markierten Wert `NOT_AVAILABLE_WM`.
- **Einfache Zustandsmaschine:** Es existiert nur ein einziger Ausführungszustand (REQ) – der Baustein ist ereignisgesteuert und verweilt nicht in weiteren Zuständen.

Zustandsübersicht

Der Funktionsblock besitzt genau einen Zustand:

Zustand	Beschreibung
REQ	Führt den Algorithmus aus, setzt die Ausgänge entsprechend und sendet das Bestätigungsereignis **CNF**.

Es gibt keine Schleifen oder Verzweigungen in der Zustandsmaschine; jeder Aufruf von REQ wird einmalig durchlaufen.

Anwendungsszenarien

- **Feldbus-Signalfilterung:** Überprüfung, ob ein von einem Feldbus stammender WORD-Wert innerhalb eines gültigen Bereichs liegt (z. B. Sensorwerte, Steuerbefehle).
- **Plausibilitätsprüfung:** Unterdrücken von Werten, die als „nicht verfügbar“ oder „fehlerhaft“ markiert sind (dargestellt durch `NOT_AVAILABLE_WM`).
- **Datenkonsistenz:** Einsatz als einfaches Tor (Gate) in industriellen Steuerungen, das nur gültige Daten an nachfolgende Logik weitergibt.

Vergleich mit ähnlichen Bausteinen

Ähnliche Funktionsbausteine wie **FIELDBUS_SIGNAL_FILTER** oder **VALUE_CHECK** führen ebenfalls Vergleiche mit Schwellwerten durch. Der Vorteil von `FIELDBUS_WORD_TO_SIGNAL` liegt in der direkten Integration der Gültigkeitskennung (`VALID`) und der Verwendung von vordefinierten Konstanten aus der Feldbus-Bibliothek, was eine standardisierte Handhabung von „nicht verfügbar“-Zuständen ermöglicht. Im Gegensatz zu generischen Mux-Bausteinen ist die Logik hier auf den spezifischen Einsatz im Feldbuskontext optimiert.

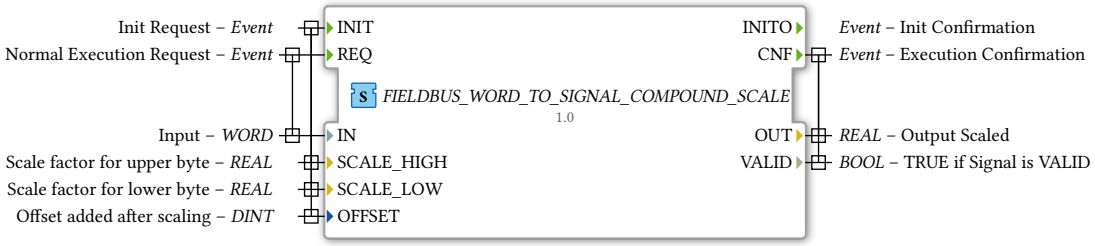
Fazit

Der Funktionsblock `FIELDBUS_WORD_TO_SIGNAL` bietet eine kompakte und zuverlässige Möglichkeit, Feldbussignale auf Gültigkeit zu prüfen und nur als gültig erkannte Werte weiterzuleiten. Durch die Verwendung von importierten Konstanten wird eine konsistente Definition von Gültigkeitsgrenzen und Unavailable-Werten über das gesamte Projekt hinweg gewährleistet. Der Baustein eignet sich besonders für sicherheitskritische Anwendungen, in denen ungültige Daten erkannt und unterdrückt werden müssen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

FIELDBUS_WORD_TO_SIGNAL_COMPOUND_SCALE



• – ◦ • – ◦ • – ◦ •

Einleitung

Dieser Funktionsbaustein bildet einen 16-Bit-Wort-Eingang auf einen skalierten Realwert ab. Dabei wird das eingehende Wort zunächst auf Gültigkeit geprüft. Ist das Signal gültig, werden das obere und das untere Byte mit jeweils eigenen Skalierungsfaktoren multipliziert und mit einem Offset addiert. Das Ergebnis wird als Skalierung des ursprünglichen Feldbussignals ausgegeben.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit) – Initialisierungsaufforderung; mit den Parametern `SCALE_HIGH`, `SCALE_LOW` und `OFFSET`.
- **REQ** (Event) – Normale Ausführungsanforderung; mit dem Parameter `IN`.

Ereignis-Ausgänge

- **INITO** (EInit) – Bestätigung der Initialisierung.
- **CNF** (Event) – Bestätigung der Ausführung; mit den Parametern `OUT` und `VALID`.

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung
`IN`	WORD	`NOT_AVAILABLE_WM`	Das zu verarbeitende 16-Bit-Feldbussignal.
`SCALE_HIGH`	REAL	0.256	Skalierungsfaktor für das obere Byte (High-Byte).
`SCALE_LOW`	REAL	0.001	Skalierungsfaktor für das untere Byte (Low-Byte).
`OFFSET`	DINT	0	Ganzzahliger Offset, der nach der Skalierung addiert wird.

Daten-Ausgänge

Name	Typ	Initialwert	Beschreibung
`OUT`	REAL	0.0	Berechneter skalarer Ausgangswert.
`VALID`	BOOL	FALSE	Zeigt an, ob das Eingangssignal gültig ist (`TRUE`) oder nicht (`FALSE`).

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Baustein arbeitet in zwei Schritten, gesteuert durch die Ereignisse:

1. **Initialisierung (INIT)**
   - Der interne Algorithmus ist leer; es wird lediglich der Ereignisausgang `INITO` gesetzt.
   - Die Skalierungsparameter (`SCALE_HIGH`, `SCALE_LOW`, `OFFSET`) werden beim Initialisierungsereignis übergeben, jedoch erst im **REQ**-Zyklus verwendet.
2. **Verarbeitung (REQ)**

- Der eingehende Wert `IN` wird mit der externen Konstanten `VALID_SIGNAL_W` verglichen.
- **Gültiges Signal** (`IN` ≤ `VALID_SIGNAL_W`):
  a. Das obere Byte wird durch Rechtsverschiebung um 8 Bit und Maskierung mit `0x00FF` extrahiert.
  b. Das untere Byte wird durch Maskierung mit `0x00FF` gewonnen.
  c. Beide Bytes werden in `REAL` konvertiert, mit den zugehörigen Skalierungsfaktoren multipliziert und zum Offset addiert: `OUT = (highByte * SCALE_HIGH) + (lowByte * SCALE_LOW) + OFFSET`
  d. `VALID` wird auf `TRUE` gesetzt.
- **Ungültiges Signal** (sonst):
  – `OUT` wird auf `0.0` gesetzt.
  – `VALID` wird auf `FALSE` gesetzt.

Der folgende ST-Code verdeutlicht die interne Logik:

```
IF (WORD_TO_UINT(IN) <= WORD_TO_UINT(VALID_SIGNAL_W)) THEN
    temp := SHR(IN, SINT#8);
    highByte := temp AND WORD#16#00FF;
    lowByte := IN AND WORD#16#00FF;
    OUT := UINT_TO_REAL(WORD_TO_UINT(highByte)) * SCALE_HIGH

         + UINT_TO_REAL(WORD_TO_UINT(lowByte)) * SCALE_LOW
         + DINT_TO_REAL(OFFSET);
    VALID := BOOL#TRUE;
ELSE
    OUT := REAL#0.0;
    VALID := BOOL#FALSE;
END_IF;
```

Technische Besonderheiten

- **Compound-Skalierung:** Die beiden Bytes eines Words werden getrennt skaliert. Dies ermöglicht die Abbildung von Messwerten, deren Information auf zwei Bytes verteilt ist (z. B. unterschiedliche Auflösungen oder Einheiten).
- **Feldbus-Konstanten:** Die beiden externen Konstanten `VALID_SIGNAL_W` und `NOT_AVAILABLE_WM` definieren den gültigen Wertebereich bzw. den „nicht verfügbar“-Zustand. Sie stammen aus dem Paket `eclipse4diac::signalprocessing::FIELDBUS_SIGNAL`.
- **Initialwerte:** Die Voreinstellungen `SCALE_HIGH = 0.256`, `SCALE_LOW = 0.001` und `OFFSET = 0` sind praxiserprobte Werte für typische Feldbussignale (z. B. Temperaturmessung).
- **Keine Zustandsabhängigkeit:** Der Baustein besitzt keine internen Zustandsautomaten, die über INIT/REQ hinausgehen – die Reaktion ist rein kombinatorisch beim REQ-Ereignis.

Zustandsübersicht

Der Baustein verfügt über zwei einfache Zustände, die direkt den Ereignissen entsprechen:

Zustand	Ausgelöst durch	Ausgeführte Aktion	Ausgangsereignis
`INIT`	Ereignis `INIT`	Algorithmus `INIT` (leer)	`INITO`
`REQ`	Ereignis `REQ`	Algorithmus `REQ` (Skalierung)	`CNF`

Weitere Zustände oder Verzweigungen sind nicht vorhanden.

Anwendungsszenarien

- **Temperaturmessung über zwei Kanäle** – High-Byte repräsentiert den ganzzahligen Anteil, Low-Byte den Nachkommateil einer Temperatur (z. B. 0.256 °C/LSB für High, 0.001 °C/LSB für Low).
- **Druck- oder Füllstandssensoren** – Liefern einen 16-Bit-Rohwert, der in zwei unterschiedlich skalierte Anteile zerlegt wird, um den physikalischen Wert zu ermitteln.
- **Feldbus-Datenvorverarbeitung** – In einer logiBUS-Signalverarbeitungskette kann dieser Baustein als erste Stufe zur Konvertierung von Wortwerten in reale physikalische Größen eingesetzt werden.

Vergleich mit ähnlichen Bausteinen

- **FIELDBUS_WORD_TO_SIGNAL** (ohne Compound) – Wendet nur einen einzigen Skalierungsfaktor auf das gesamte Word an. Geeignet, wenn High- und Low-Byte die gleiche Gewichtung haben.

- **FIELDBUS_BYTE_TO_SIGNAL** – Verarbeitet nur ein einzelnes Byte. Benötigt zwei Instanzen, um beide Bytes eines Words zu behandeln.
- **FIELDBUS_WORD_TO_SIGNAL_COMPOUND_SCALE** zeichnet sich durch die getrennte Skalierung der beiden Bytes aus, was eine flexiblere und genauere Abbildung vieler Sensorprotokolle ermöglicht.

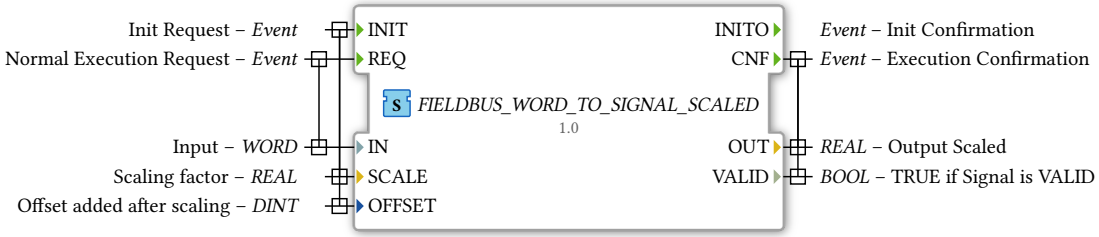
Fazit

Der Baustein **FIELDBUS_WORD_TO_SIGNAL_COMPOUND_SCALE** ist ein spezialisiertes Werkzeug zur Aufbereitung von 16-Bit-Feldbussignalen, bei denen die beiden Bytes unterschiedliche physikalische Bedeutung oder Auflösung haben. Die klare Trennung von Gültigkeitsprüfung, Byte-Extraktion und Compound-Skalierung macht ihn zu einem verlässlichen und konfigurierbaren Element in der Signalverarbeitungskette. Durch die extern definierten Feldbus-Konstanten ist er leicht an verschiedene Protokolle anpassbar.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

FIELDBUS_WORD_TO_SIGNAL_SCALED



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **FIELDBUS_WORD_TO_SIGNAL_SCALED** dient dazu, einen digitalen Feldbus-Rohwert (16-Bit WORD) in einen skalierten physikalischen Signalwert (REAL) umzurechnen. Dabei wird nur dann ein gültiger Ausgang geliefert, wenn der eingehende Wert als gültiges Signal erkannt wird. Der Baustein prüft die Gültigkeit anhand einer vordefinierten Konstanten (`VALID_SIGNAL_W`) und kann unsichere oder „nicht verfügbare“ Werte zuverlässig unterdrücken.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung
`INIT`	Initialisierungsanfrage; setzt die Skalierungsfaktoren (`SCALE`, `OFFSET`).
`REQ`	Normale Verarbeitungsanfrage; wertet den aktuellen Eingang `IN` aus.

Ereignis-Ausgänge

Ereignis	Beschreibung
`INITO`	Bestätigung der Initialisierung.
`CNF`	Bestätigung der Verarbeitung; signalisiert, dass die Ausgänge `OUT` und `VALID` aktualisiert wurden.

Daten-Eingänge

Name	Datentyp	Vorgabewert	Beschreibung

IN	WORD	NOT_AVAILABLE_WM	Rohwert vom Feldbus (z.B. 0…65535). Die Konstante NOT_AVAILABLE_WM kennzeichnet einen ungültigen Startwert.
SCALE	REAL	1.0	Skalierungsfaktor (wird mit dem eingelesenen Wert multipliziert).
OFFSET	DINT	0	Ganzzahliger Offset, der nach der Skalierung addiert wird (als REAL konvertiert).

Daten-Ausgänge

Name	Datentyp	Vorgabewert	Beschreibung
OUT	REAL	0.0	Skalierter Signalwert. Nur gültig, wenn VALID = TRUE.
VALID	BOOL	FALSE	Gültigkeitsflag – TRUE, wenn der eingehende Rohwert als gültiges Signal erkannt wurde.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Baustein arbeitet zustandsgesteuert:

1. **INIT-Zustand**: Nach dem Eintreten des INIT-Ereignisses wird der Initialisierungsalgorithmus (leer) ausgeführt und das Ereignis INITO ausgegeben. Die Skalierungsfaktoren werden dabei an den Eingängen SCALE und OFFSET übernommen (die Werte liegen bereits an den Eingängen an).
2. **REQ-Zustand** (Hauptlogik): Bei jedem REQ-Ereignis wird folgender Algorithmus durchlaufen:
   - Der eingehende IN (WORD) wird zunächst in UINT konvertiert.
   - Liegt dieser Wert **kleiner oder gleich** dem in der globalen Konstanten VALID_SIGNAL_W definierten Grenzwert (ebenfalls als WORD/UINT), gilt das Signal als **gültig**.
   - Bei Gültigkeit: `OUT = UINT_TO_REAL(IN) * SCALE + DINT_TO_REAL(OFFSET)` `VALID = TRUE`
   - Bei Ungültigkeit: `OUT = 0.0` `VALID = FALSE`
   - Anschließend wird das CNF-Ereignis ausgelöst, das die aktualisierten Ausgänge OUT und VALID begleitet.

Der Vergleichswert VALID_SIGNAL_W sowie der Standard-Eingangswert NOT_AVAILABLE_WM sind als Konstanten im Namespace `eclipse4diac::signalprocessing::FIELDBUS_SIGNAL` importiert.

Technische Besonderheiten

- **Verwendung von externen Konstanten**: Die Gültigkeitsschwelle (VALID_SIGNAL_W) und der „Nicht verfügbar"-Marker (NOT_AVAILABLE_WM) werden aus einer globalen Bibliothek bezogen. Dies ermöglicht eine konsistente Signaldefinition über mehrere FB-Instanzen hinweg.
- **Skalierung mit REAL und DINT**: Der Offset ist als DINT definiert, wird aber durch DINT_TO_REAL in eine Gleitkommazahl gewandelt. Dies kann bei großen Werten zu geringfügigen Rundungsfehlern führen, ist in der Praxis aber meist vernachlässigbar.
- **Gültigkeitslogik**: Anders als bei einfachen „Word-to-Real"-Wandlern wird der Wert nur dann ausgegeben, wenn er innerhalb des definierten gültigen Bereichs liegt. Dadurch werden Fehlinterpretationen von Störungen oder ungültigen Telegrammen vermieden.
- **Initialverhalten**: Die Eingänge IN haben als Vorgabewert NOT_AVAILABLE_WM, sodass der FB nach einem Reset oder bei fehlender Kommunikation sofort „ungültig" meldet, bis ein gültiger Wert ankommt.

Zustandsübersicht

Zustand	Inhalt	Aktion	Ausgangsereignis
INIT	Initialisierung (leerer Algorithmus)	–	INITO

REQ	Signalverarbeitung	Berechnung von OUT und VALID	CNF

Der FB kennt keine weiteren Zustände; nach jedem Ereignis kehrt er in den entsprechenden Startzustand zurück. Es gibt keine explizite Fehlerbehandlung oder Timeouts.

Anwendungsszenarien

- **Landtechnik / Agrarsteuerungen**: Ein Feldbus-Gerät (z. B. ISOBUS) liefert Rohwerte für Sensoren (Drehzahl, Temperatur, Druck). Der FB skaliert diese auf physikalische Einheiten und erkennt ungültige Messbereiche.
- **Industrielle Automatisierung**: Bei Profibus/IO-Link-Anbindung können Wort-Codes in prozessrelevante Größen umgerechnet werden – z. B. 0…1000 ⟶ 0,0…10,0 V.
- **Diagnose- und Sicherheitsfunktionen**: Durch die Gültigkeitsprüfung lassen sich Signalverluste oder Bereichsüberschreitungen einfach erkennen und eine Fehlerreaktion einleiten.

Vergleich mit ähnlichen Bausteinen

Baustein	Unterschiede
FIELDBUS_WORD_TO_SIGNAL (ohne Skalierung)	Bildet den Rohwert 1:1 ab, ohne Multiplikation/Addition.
WORD_TO_REAL (Standard-IEC-Baustein)	Reine Typumwandlung ohne Gültigkeitsprüfung und Skalierung.
SCALING_FB	Meist mit separatem Gültigkeits- und Skalierungszweig; dieser FB vereint beides in einem Schritt.

Die Stärke von `FIELDBUS_WORD_TO_SIGNAL_SCALED` liegt in der **Kombination aus Gültigkeitsprüfung, Skalierung und Offset** in einem einzigen, überschaubaren FB.

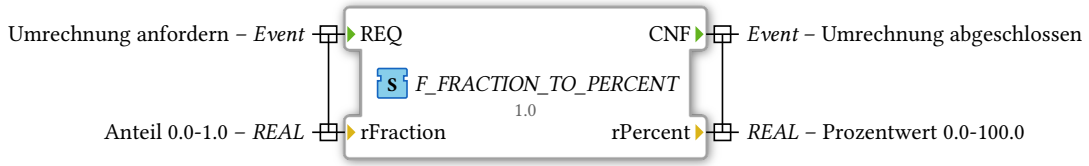
Fazit

Der Funktionsblock bietet eine kompakte und robuste Lösung zur Aufbereitung von Feldbus-Rohdaten. Er integriert die typische Aufgabenkette „Prüfen ⟶ Skalieren ⟶ Offsieten“ und sorgt durch die feste Gültigkeitsschwelle für eine klare Trennung zwischen normalen Messwerten und Fehlersignalen. Durch die Verwendung externer Konstanten bleibt er flexibel anpassbar, ohne die innere Logik zu verkomplizieren. Ideal für sicherheitskritische oder diagnostische Anwendungen in der Automatisierungstechnik.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

F_FRACTION_TO_PERCENT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **F_FRACTION_TO_PERCENT** ist das Gegenstück zu `F_PERCENT_TO_FRACTION`: Er rechnet einen Anteil (REAL 0.0–1.0), wie ihn `MyLib::sys::F_PWM_RAW_TO_PERCENT` liefert, in einen Prozentwert (REAL 0.0–100.0) um, der z. B. per OPC-UA-Publish an einen Web-Client gesendet wird.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Datentyp	Kommentar

REQ	Event	Umrechnung anfordern; verarbeitet den aktuellen Wert von `rFraction`.

Ereignis-Ausgänge

Ereignis	Datentyp	Kommentar
CNF	Event	Umrechnung abgeschlossen; `rPercent` wurde aktualisiert.

Daten-Eingänge

Name	Datentyp	Kommentar
rFraction	REAL	Anteil 0.0-1.0.

Daten-Ausgänge

Name	Datentyp	Kommentar
rPercent	REAL	Prozentwert 0.0-100.0.

Adapter

Keine.

Funktionsweise

Bei jedem `REQ`-Ereignis wird der Eingang `rFraction` unbegrenzt mit 100 multipliziert:
`rPercent := rFraction * 100.0;`
Anschließend wird das Ereignis `CNF` ausgegeben.

Technische Besonderheiten

- Realisiert als **SimpleFB** mit nur einem Ausführungszustand (`REQ`) — keine mehrstufige Zustandsmaschine.
- Im Gegensatz zum Gegenstück `F_PERCENT_TO_FRACTION` **kein Klemmen** des Eingangswerts: Da `rFraction` in der PWM-Kette bereits durch `RampLimitFS`/`F_PWM_RAW_TO_PERCENT` auf 0.0-1.0 begrenzt ist, ist eine zusätzliche Bereichsprüfung hier nicht nötig.
- Bewusst getrennt von `MyLib::sys::F_PWM_RAW_TO_PERCENT`, das den Fieldbus-Rohwert (nicht Prozent!) in den Anteil umrechnet — beide Bausteine zusammen bilden die vollständige Kette Fieldbus-Rohwert → Anteil → Prozent.
- Lizenz: Eclipse Public License 2.0.

Zustandsübersicht

Zustand	Aktion	Ausgabeereignis
REQ	Führt den Algorithmus **REQ** aus (Multiplikation mit 100)	CNF

Es gibt keine weiteren Warte- oder Initialisierungszustände.

Anwendungsszenarien

- Rückrechnung eines intern verarbeiteten Anteils (0.0-1.0) auf den Prozentwert, den ein Web-Client per OPC-UA-Publish erwartet — z. B. im PWM-Trainingsbeispiel `RampLimitFS_TO_logiBUS_QDA_PWM_OPC`.
- Allgemein: jede Schnittstelle, die intern mit einem normierten Anteil 0.0-1.0 rechnet, extern aber in Prozent kommuniziert.

Vergleich mit ähnlichen Bausteinen

- Gegenstück ist `F_PERCENT_TO_FRACTION` (Prozent → Anteil, für die Vorwärtsrichtung, inklusive Klemmen).
- Einfachste denkbare Skalenwandler ohne jede Bereichsprüfung, da der Eingang bereits garantiert normiert ankommt.

Fazit

F_FRACTION_TO_PERCENT ist eine reine Multiplikation mit 100, die den intern verwendeten Anteil 0.0-1.0 auf den extern kommunizierten Prozentwert 0.0-100.0 bringt — ohne eigene Fehlerbehandlung, da der Eingang bereits vor-normiert ist.

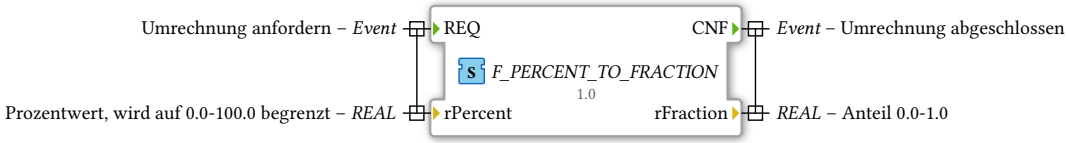
🛠️ Zugehörige Übungen

- RampLimitFS_TO_logiBUS_QDA_PWM_OPC
- InputOutputTesterButton_PWM_OPC_UA

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

F_PERCENT_TO_FRACTION



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **F_PERCENT_TO_FRACTION** rechnet einen Prozentwert (REAL 0.0–100.0), wie er z. B. über OPC-UA-Subscribe von einem Web-Client empfangen wird, in einen Anteil (REAL 0.0–1.0) um, wie ihn `MyLib::sys::F_PWM_PERCENT_TO_RAW` als Eingang erwartet. Außerhalb des gültigen Bereichs liegende Werte werden geklemmt statt einen Fehler zu erzeugen.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Datentyp	Kommentar
REQ	Event	Umrechnung anfordern; verarbeitet den aktuellen Wert von `rPercent`.

Ereignis-Ausgänge

Ereignis	Datentyp	Kommentar
CNF	Event	Umrechnung abgeschlossen; `rFraction` wurde aktualisiert.

Daten-Eingänge

Name	Datentyp	Kommentar
rPercent	REAL	Prozentwert, wird auf 0.0-100.0 begrenzt.

Daten-Ausgänge

Name	Datentyp	Kommentar
rFraction	REAL	Anteil 0.0-1.0.

Adapter

Keine.

Funktionsweise

Bei jedem `REQ`-Ereignis wird der Eingang `rPercent` geprüft und auf den gültigen Bereich geklemmt:

```
IF rPercent <= 0.0 THEN
    rFraction := 0.0;
ELSIF rPercent >= 100.0 THEN
    rFraction := 1.0;
ELSE
    rFraction := rPercent / 100.0;
END_IF;
```

- Werte `<= 0.0` ergeben `rFraction = 0.0`, Werte `>= 100.0` ergeben `rFraction = 1.0`.
- Alle anderen Werte werden linear durch 100 geteilt.
- Anschließend wird das Ereignis `CNF` ausgegeben.

Technische Besonderheiten

- Realisiert als **SimpleFB** mit nur einem Ausführungszustand (`REQ`) — keine mehrstufige Zustandsmaschine.
- Klemmt statt zu fehlerbehandeln: unplausible Prozentwerte (z. B. durch fehlerhafte Web-Eingaben) führen nicht zu einem Fehlerzustand, sondern werden stillschweigend auf 0-100 % begrenzt.
- Bewusst getrennt von `MyLib::sys::F_PWM_PERCENT_TO_RAW`, das Anteil (nicht Prozent!) in den Fieldbus-Rohwert umrechnet — beide Bausteine zusammen bilden die vollständige Kette Prozent → Anteil → Fieldbus-Rohwert.
- Lizenz: Eclipse Public License 2.0.

Zustandsübersicht

Zustand	Aktion	Ausgabeereignis
REQ	Führt den Algorithmus **REQ** aus (Klemmen + Division durch 100)	CNF

Es gibt keine weiteren Warte- oder Initialisierungszustände.

Anwendungsszenarien

- Umrechnung eines per OPC-UA-Subscribe empfangenen Web-Sollwerts (Prozent REAL) in den Anteil, den interne Fieldbus-Umrechnungsbausteine erwarten — z. B. im PWM-Trainingsbeispiel `RampLimitFS_TO_logiBUS_QDA_PWM_OPC`.
- Allgemein: jede Schnittstelle, die extern in Prozent kommuniziert, intern aber mit einem normierten Anteil 0.0-1.0 rechnet.

Vergleich mit ähnlichen Bausteinen

- Gegenstück ist `F_FRACTION_TO_PERCENT` (Anteil → Prozent, für die Rückrichtung).
- Im Gegensatz zu den `FIELDBUS_*_TO_SIGNAL`-Bausteinen (z. B. `FIELDBUS_BYTE_TO_SIGNAL`) prüft dieser Baustein keine Gültigkeit gegen eine Konstante, sondern führt eine reine, klemmende Skalenumrechnung durch.

Fazit

F_PERCENT_TO_FRACTION ist ein minimaler, klemmender Skalenwandler, der die Lücke zwischen extern kommuniziertem Prozentwert und intern verwendetem Anteil 0.0-1.0 schließt — ohne eigene Fehlerbehandlung, da außerhalb liegende Werte einfach begrenzt werden.

🛠️ Zugehörige Übungen

- RampLimitFS_TO_logiBUS_QDA_PWM_OPC
- InputOutputTesterButton_PWM_OPC_UA

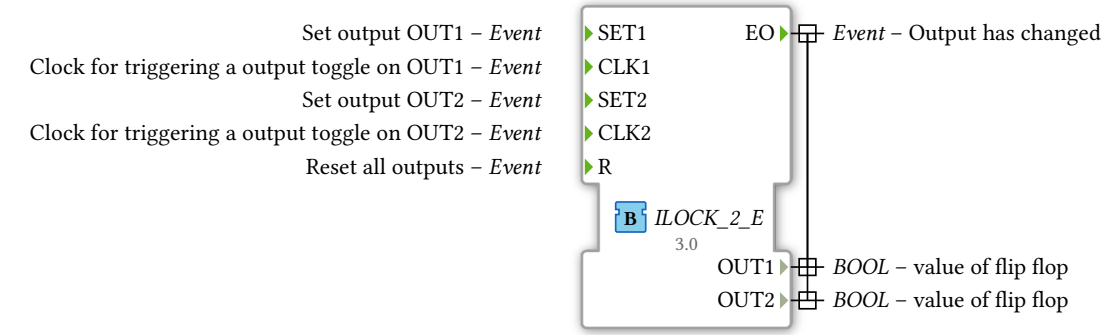
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

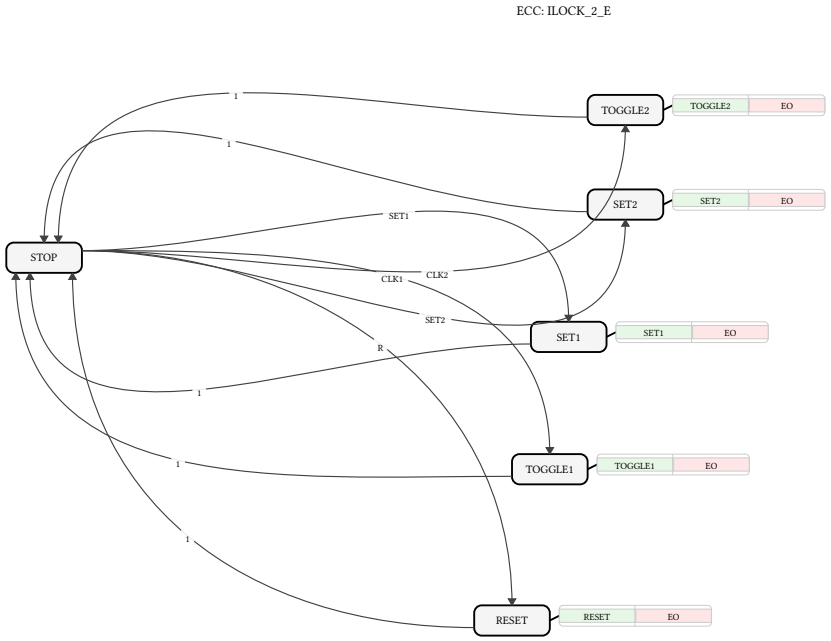
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Numerischer Index (signalprocessing)

Hier finden Sie alle Übungen des Arbeitsbereichs `signalprocessing` in aufsteigender Reihenfolge.

ILOCK_2_E





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ILOCK_2_E** ist ein ereignisgesteuerter bistabiler und Toggle-Baustein mit dualer Verriegelung (Interlock). Er dient zur Steuerung von zwei sich gegenseitig ausschließenden Ausgängen (OUT1, OUT2). Neben expliziten Setz- und Rücksetzfunktionen bietet der Block eine flankengesteuerte Toggle-Funktion, die ein einmaliges Umschalten pro Aktivierung ermöglicht. Der Baustein eignet sich besonders für Steuerungslogiken, bei denen niemals beide Ausgänge gleichzeitig TRUE sein dürfen und ein wiederholter Toggle-Befehl erst nach erneuter Freigabe (z. B. durch ein Setzen des anderen Kanals oder Reset) wirksam wird.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Kommentar
`SET1`	Setzt OUT1 auf TRUE und OUT2 auf FALSE.
`CLK1`	Toggelt OUT1 (bei erstmaliger Aktivierung nach einem Set/Reset) und setzt OUT2 auf FALSE.

SET2	Setzt OUT2 auf TRUE und OUT1 auf FALSE.
CLK2	Toggelt OUT2 und setzt OUT1 auf FALSE.
R	Rücksetzen (Reset) beider Ausgänge auf FALSE.

Ereignis-Ausgänge

Ereignis	Kommentar
EO	Wird nach jeder Ausgangsänderung (SET, TOGGLE oder RESET) ausgelöst. Trägt die aktuellen Werte von OUT1 und OUT2.

Daten-Eingänge

Keine (der Baustein ist rein ereignisgesteuert).

Daten-Ausgänge

Ausgang	Typ	Kommentar
OUT1	BOOL	Erster Ausgang (bistabiler Wert).
OUT2	BOOL	Zweiter Ausgang (bistabiler Wert).

Adapter

Keine.

Funktionsweise

Der FB besitzt sechs Zustände: den initialen Ruhezustand **STOP** sowie fünf Aktionszustände (**SET1**, **SET2**, **TOGGLE1**, **TOGGLE2**, **RESET**). Bei Eintreffen eines Ereignisses wird sofort in den entsprechenden Zustand gewechselt, der zugehörige Algorithmus ausgeführt und anschließend das Ereignis EO ausgegeben. Danach kehrt der Baustein unverzüglich in den Zustand **STOP** zurück (Bedingung 1 auf allen Transitionen).

Die interne Logik sorgt für eine **duale Verriegelung**: Beim Setzen eines Ausgangs wird der andere zwangsweise auf FALSE gesetzt. Auch beim Togglen eines Ausgangs wird der andere Ausgang zurückgesetzt.

Die **Toggle-Funktion** ist flankengesteuert: Die internen Variablen EDGE1 und EDGE2 speichern, ob bereits ein Toggle auf dem entsprechenden Kanal ausgeführt wurde. Nur bei EDGE = FALSE (d. h. nach einem Reset oder einem Set des anderen Kanals) wird der Toggle tatsächlich ausgeführt und das Flag auf TRUE gesetzt. Wiederholte CLK-Ereignisse ohne zwischenzeitliches Setzen/Rücksetzen des eigenen oder des anderen Kanals werden ignoriert.

Technische Besonderheiten

- **Kantenerkennung für Toggle** – Die internen Flags EDGE1/EDGE2 verhindern mehrfaches Togglen bei derselben Freigabe.
- **Immer exklusive Ausgänge** – Jeder Algorithmus stellt sicher, dass nie beide Ausgänge gleichzeitig TRUE sind.
- **Keine Daten-Eingänge** – Die gesamte Steuerung erfolgt ausschließlich über Ereignisse, was eine einfache Integration in ereignisbasierte Automatisierungssysteme ermöglicht.
- **Robustes Reset-Verhalten** – Ein Reset setzt sowohl die Ausgänge als auch die internen Kantenflags zurück, sodass die Toggle-Funktion wieder bereit ist.

Zustandsübersicht

Zustand	Beschreibung	Auslöser	Algorithmus-Effekt (vereinfacht)	Ausgang
STOP	Initial, wartet auf Ereignis	-	–	–
SET1	Setze Ausgang 1	SET1	OUT1=TRUE, OUT2=FALSE, EDGE1=FALSE, EDGE2=FALSE	EO

SET2	Setze Ausgang 2	SET2	OUT1=FALSE, OUT2=TRUE, EDGE1=FALSE, EDGE2=FALSE	EO
TOGGLE1	Toggle Ausgang 1	CLK1	Wenn EDGE1=FALSE: OUT1 = NOT OUT1; OUT2=FALSE; EDGE1=TRUE; EDGE2=FALSE	EO
TOGGLE2	Toggle Ausgang 2	CLK2	OUT1=FALSE; wenn EDGE2=FALSE: OUT2 = NOT OUT2; EDGE1=FALSE; EDGE2=TRUE	EO
RESET	Reset aller Ausgänge	R	OUT1=FALSE, OUT2=FALSE, EDGE1=FALSE, EDGE2=FALSE	EO

Alle Aktionszustände wechseln nach Ausführung des Algorithmus sofort zurück zu STOP.

Anwendungsszenarien

- **Richtungssteuerung** (z. B. Motor links/rechts) – nie beide Richtungen gleichzeitig aktiv.
- **Pumpensteuerung mit Priorität** – zwei Pumpen, wobei immer nur eine laufen darf.
- **Modusumschaltung mit einmaligem Toggle** – z. B. „Start/Stop"-Verhalten, bei dem ein wiederholter Toggle-Befehl erst nach einem Reset oder einer anderen Aktion wirksam wird.
- **Verriegelte Schaltzustände in Sicherheitssteuerungen** – garantierte Exklusivität der Ausgänge.

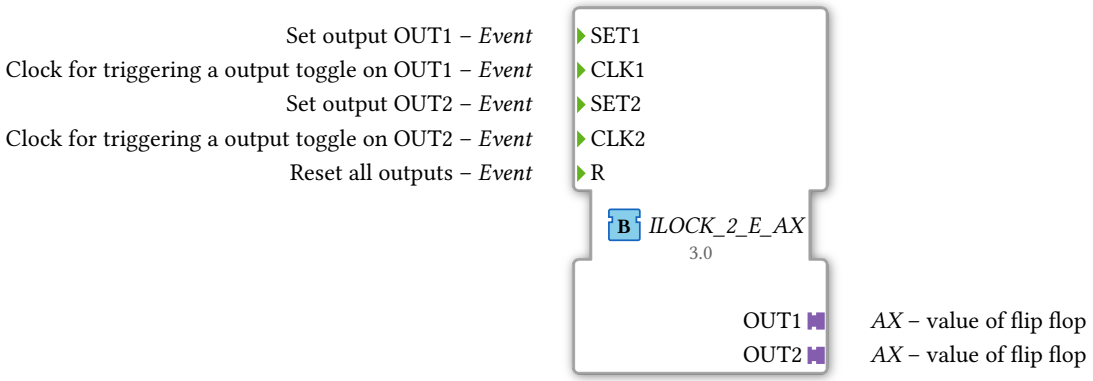
Vergleich mit ähnlichen Bausteinen

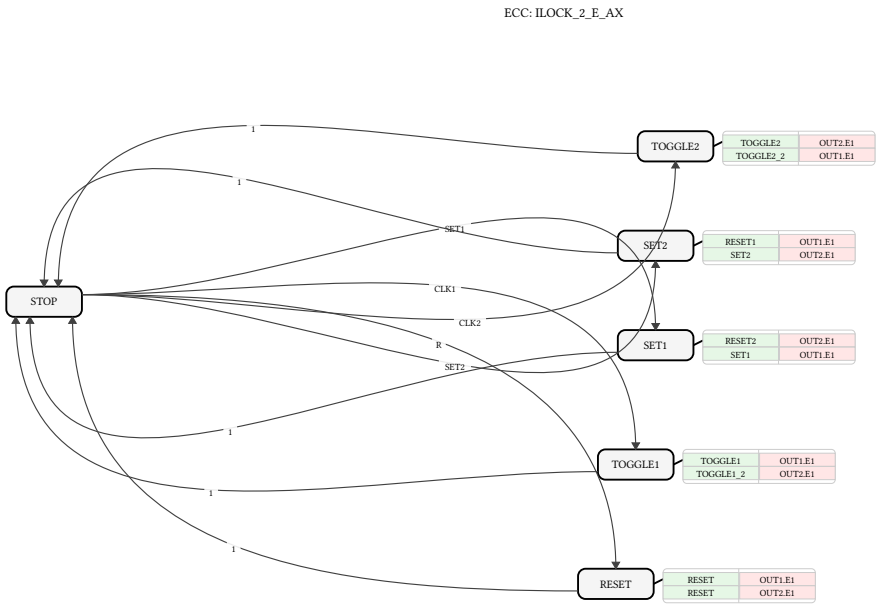
Baustein	Eigenschaften
E_SR	Einfacher Set-/Reset-Flipflop, kein Toggle, keine duale Verriegelung.
SR-Flipflop (zwei Ausgänge)	Oft nur kombinatorisch oder ohne Kantenerkennung.
ILOCK_2_E	Bietet beides: **Setzen** (exklusiv) und **flankengesteuertes Toggle** mit automatischer Verriegelung. Ideal für erweiterte Anforderungen.

Fazit

Der Funktionsblock ILOCK_2_E ist ein leistungsfähiger Baustein für Anwendungen mit zwei sich gegenseitig ausschließenden Zuständen. Die Kombination aus explizitem Setzen, flankengesteuertem Toggle und robustem Reset-Verhalten macht ihn besonders geeignet für Steuerungslogiken, die eine klare Trennung der Ausgänge und eine definierte Reaktion auf wiederholte Befehle erfordern. Die Integration in ereignisbasierte Automatisierungssysteme (z. B. IEC 61499) erfolgt durch die rein ereignisgesteuerte Schnittstelle und den einheitlichen Ausgang EO.

ILOCK_2_E_AX





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ILOCK_2_E_AX** stellt zwei bistabile Ausgänge mit gegenseitiger Verriegelung (Interlock) sowie einer Toggle-Funktion bereit. Er wird über Ereignisse gesteuert und gibt seine Zustände über zwei Adapter (Typ `AX`) aus. Der Baustein eignet sich für Anwendungen, bei denen zwei sich gegenseitig ausschließende Signale benötigt werden, z. B. für die Ansteuerung von Weichen, Ventilen oder Motoren.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Beschreibung
`SET1`	Setzt Ausgang `OUT1` auf TRUE und `OUT2` auf FALSE
`CLK1`	Toggelt den Zustand von `OUT1`, sofern nicht bereits getoggelt
`SET2`	Setzt Ausgang `OUT2` auf TRUE und `OUT1` auf FALSE
`CLK2`	Toggelt den Zustand von `OUT2`, sofern nicht bereits getoggelt
`R`	Setzt beide Ausgänge auf FALSE (Reset)

Ereignis-Ausgänge

Der Funktionsblock besitzt keine eigenen Ereignis-Ausgänge. Die Ausgangssignale werden ausschließlich über die **Adapter** (siehe unten) übertragen.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine direkten Daten-Ausgänge. Die aktuellen Zustände der Flip-Flops sind über die **Adapter** (siehe unten) abrufbar.

Adapter

Name	Typ	Beschreibung
OUT1	AX	Stellt den Zustand des ersten Flip-Flops bereit (Datenausgang D1) und quittiert Änderungen über Ereignisausgang E1
OUT2	AX	Stellt den Zustand des zweiten Flip-Flops bereit (Datenausgang D1) und quittiert Änderungen über Ereignisausgang E1

Funktionsweise

Der Baustein arbeitet ereignisgesteuert. Ein eingehendes Ereignis löst eine Zustandsänderung aus, die sofort an den entsprechenden Adaptern ausgegeben wird.

- **SET1**: Setzt `OUT1.D1 = TRUE` und `OUT2.D1 = FALSE`. Die internen Flankenspeicher (EDGE1, EDGE2) werden zurückgesetzt.
- **SET2**: Setzt `OUT1.D1 = FALSE` und `OUT2.D1 = TRUE`. Flankenspeicher werden zurückgesetzt.
- **CLK1**: Toggelt `OUT1.D1` (wenn nicht bereits durch ein vorheriges CLK1-Ereignis getoggelt) und setzt `OUT2.D1 = FALSE`. Der Flankenspeicher EDGE1 verhindert wiederholtes Toggeln beim selben Ereignis.
- **CLK2**: Toggelt `OUT2.D1` (Flankenspeicher EDGE2) und setzt `OUT1.D1 = FALSE`.
- **R**: Setzt beide Ausgänge auf FALSE und löscht die internen Flankenspeicher.

Die Verriegelung sorgt dafür, dass immer nur einer der beiden Ausgänge den Wert TRUE annehmen kann. Beim Toggeln eines Ausgangs wird der andere stets auf FALSE gesetzt.

Technische Besonderheiten

- Der Baustein verwendet **Adapter** (Typ AX) als Ausgänge. Dies ermöglicht eine flexible Verbindung mit anderen Bausteinen, die denselben Adapter unterstützen.
- Eine **interne Flankenerkennung** (Variablen EDGE1, EDGE2) verhindert mehrfaches Toggeln bei einem einzelnen Ereignis. Erst nach einem Reset oder einem expliziten Setzen wird der Flankenspeicher zurückgesetzt, sodass ein erneutes Toggeln möglich ist.
- Alle Algorithmen werden **atomar** in den Zuständen des ECC ausgeführt; der Baustein gibt nach jeder Aktion ein Ereignis über die Adapter aus (`OUT1.E1` bzw. `OUT2.E1`).
- Die Initialzustände aller Ausgänge sind **FALSE**.

Zustandsübersicht

Der Ablaufsteuerung (ECC) liegen folgende Zustände zugrunde:

- **STOP** – Wartezustand (initial)
- **SET1** – Setzt Kanal 1
- **SET2** – Setzt Kanal 2
- **TOGGLE1** – Toggelt Kanal 1, deaktiviert Kanal 2
- **TOGGLE2** – Toggelt Kanal 2, deaktiviert Kanal 1
- **RESET** – Setzt beide Kanäle zurück

Die Übergänge zwischen den Zuständen erfolgen durch die empfangenen Ereignisse (SET1, SET2, CLK1, CLK2, R) und kehren nach Ausführung der Aktionen automatisch in den Zustand STOP zurück (Übergangsbedingung 1).

Anwendungsszenarien

- **Wechselweiser Betrieb** von zwei Verbrauchern, z. B. Vorschub-/Rücklaufventile, bei denen nie beide gleichzeitig aktiv sein dürfen.
- **Taktgesteuerte Umschaltung** zwischen zwei Zuständen (z. B. Richtungswechsel eines Förderbands) mittels der Toggle-Eingänge.
- **Zentrale Reset-Funktion** für eine Gruppe von bistabilen Aktoren über den R-Eingang.

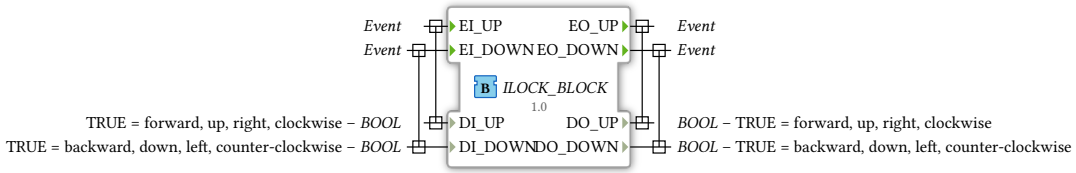
Vergleich mit ähnlichen Bausteinen

Gegenüber einem einfachen Set-Reset-Flipflop (z. B. E_SR) bietet **ILOCK_2_E_AX** zwei Ausgänge mit integrierter Verriegelung und einer Toggle-Funktion. Während E_SR nur einen einzelnen Ausgang umschaltet, kann dieser Baustein zwei gekoppelte Signale steuern. Die Verwendung von Adaptern anstelle von direkten Datenports erlaubt eine standardisierte und wiederverwendbare Kopplung mit anderen Bausteinen.

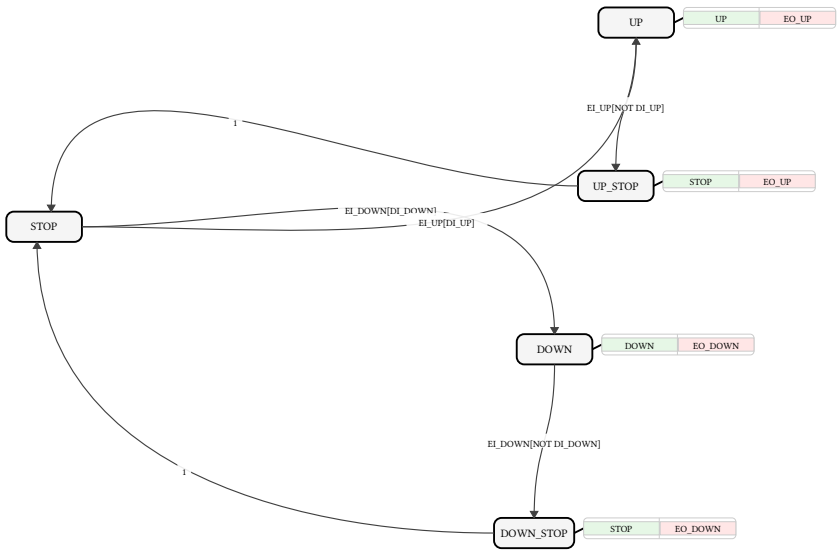
Fazit

Der Funktionsblock **ILOCK_2_E_AX** ist ein kompakter, ereignisgesteuerter Baustein zur Realisierung von verriegelten, bistabilen Ausgängen mit optionaler Toggle-Funktion. Durch die Adapter-basierte Schnittstelle und die klare Trennung von Setzen, Toggeln und Zurücksetzen eignet er sich besonders für sicherheitskritische oder logisch verknüpfte Steuerungsaufgaben in der Automatisierungstechnik.

ILOCK_BLOCK



ECC: ILOCK_BLOCK



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ILOCK_BLOCK** realisiert eine Verriegelung (Interlock) zwischen zwei gegenläufigen Signalen. Er priorisiert das zuerst eintreffende aktive Signal und ignoriert alle nachfolgenden widersprüchlichen Signale, bis das initiale Signal wieder freigegeben wird. Dadurch wird sichergestellt, dass sich zwei gegensätzliche Aktionen (z. B. Auf/Ab, Rechts/Links) niemals gleichzeitig aktivieren.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Mit Variable	Beschreibung
`EI_UP`	`DI_UP`	Ereignis zum Setzen der UP-Richtung
`EI_DOWN`	`DI_DOWN`	Ereignis zum Setzen der DOWN-Richtung

Ereignis-Ausgänge

Ereignis	Mit Variable	Beschreibung

EO_UP	DO_UP	Wird ausgelöst, wenn die UP-Richtung aktiv geschaltet wird oder wenn sie deaktiviert wird
EO_DOWN	DO_DOWN	Wird ausgelöst, wenn die DOWN-Richtung aktiv geschaltet wird oder wenn sie deaktiviert wird

Daten-Eingänge

Variable	Typ	Kommentar
DI_UP	BOOL	TRUE = vorwärts, aufwärts, rechts, im Uhrzeigersinn
DI_DOWN	BOOL	TRUE = rückwärts, abwärts, links, gegen den Uhrzeigersinn

Daten-Ausgänge

Variable	Typ	Kommentar
DO_UP	BOOL	TRUE = vorwärts, aufwärts, rechts, im Uhrzeigersinn
DO_DOWN	BOOL	TRUE = rückwärts, abwärts, links, gegen den Uhrzeigersinn

Adapter

Keine.

Funktionsweise

Der Baustein besitzt zwei Aktivierungszustände (UP, DOWN) und zwei Zwischenzustände (UP_STOP, DOWN_STOP) zum Abmelden. Die Steuerung erfolgt ausschließlich über die Ereignis-Eingänge in Verbindung mit den Daten-Eingängen.

- **In den Ruhezustand (STOP)** werden beide Ausgänge auf FALSE gesetzt.
- **UP-Aktivierung:** Trifft das Ereignis `EI_UP` mit `DI_UP = TRUE` ein, wechselt der Zustand nach **UP**. Dabei werden `DO_UP = TRUE` und `DO_DOWN = FALSE` gesetzt und `EO_UP` wird ausgegeben.
- **DOWN-Aktivierung:** Trifft das Ereignis `EI_DOWN` mit `DI_DOWN = TRUE` ein, wechselt der Zustand nach **DOWN**. Dabei werden `DO_UP = FALSE` und `DO_DOWN = TRUE` gesetzt und `EO_DOWN` wird ausgegeben.
- **Stilllegung eines aktiven Zustands:**
  - Im Zustand UP wird ein erneutes `EI_UP` mit `DI_UP = FALSE` erwartet (Freigabe). Danach wechselt der Zustand über **UP_STOP** sofort zurück zu STOP; dabei wird `EO_UP` noch einmal ausgelöst (Signalisierung des Stopps).
  - Analog wird im Zustand DOWN ein erneutes `EI_DOWN` mit `DI_DOWN = FALSE` benötigt, um über **DOWN_STOP** zurück zu STOP zu gelangen; dabei wird `EO_DOWN` ausgegeben.
- **Ignorieren widersprüchlicher Signale:** Solange der Baustein aktiv ist (UP oder DOWN), werden Ereignisse der entgegengesetzten Richtung vollständig ignoriert (keine Zustandsänderung). Dadurch wird die Priorität des ersten Signals gewahrt.

Technische Besonderheiten

- Die Zustandsübergänge erfolgen ereignisgesteuert und sofort (keine Verzögerungen).
- Im Gegensatz zu einem einfachen Set/Reset-Baustein wird die zweite Eingangsrichtung während der Verriegelung nicht akzeptiert; die Verriegelung kann nur durch das ursprüngliche Ereignis selbst aufgehoben werden.
- Alle Ausgänge werden nach einem Stopp wieder auf FALSE gesetzt.

Zustandsübersicht

Zustand	Beschreibung
STOP	Ruhezustand. Beide Ausgänge FALSE. Warte auf Aktivierung.
UP	UP-Richtung aktiv. DO_UP = TRUE, DO_DOWN = FALSE. Warte auf Freigabe durch `EI_UP` mit `DI_UP = FALSE`.

DOWN	DOWN-Richtung aktiv. DO_UP = FALSE, DO_DOWN = TRUE. Warte auf Freigabe durch `EI_DOWN` mit `DI_DOWN = FALSE`.
UP_STOP	Zwischenzustand nach Freigabe von UP. Führt sofort den STOP-Algorithmus aus, sendet `EO_UP` und wechselt zurück zu STOP.
DOWN_STOP	Zwischenzustand nach Freigabe von DOWN. Führt sofort den STOP-Algorithmus aus, sendet `EO_DOWN` und wechselt zurück zu STOP.

Übergangsmatrix (vereinfacht):

- `STOP → UP` : `EI_UP` & `DI_UP = TRUE`
- `STOP → DOWN` : `EI_DOWN` & `DI_DOWN = TRUE`
- `UP → UP_STOP` : `EI_UP` & `DI_UP = FALSE`
- `DOWN → DOWN_STOP` : `EI_DOWN` & `DI_DOWN = FALSE`
- `UP_STOP → STOP` : immer (sofort)
- `DOWN_STOP → STOP` : immer (sofort)

Anwendungsszenarien

- **Motorsteuerung (z. B. Hebebühne, Kran):** Verhindert gleichzeitiges Fahren in entgegengesetzte Richtungen.
- **Ventilsteuerung:** Schützt vor gleichzeitigem Öffnen und Schließen eines Prozessventils.
- **Richtungsverriegelung in Förderanlagen:** Sorgt dafür, dass ein Band nur eine Drehrichtung gleichzeitig aktiviert.
- **Sicherheitskritische Steuerungen:** Erzwingt eine eindeutige, priorisierte Signalfolge.

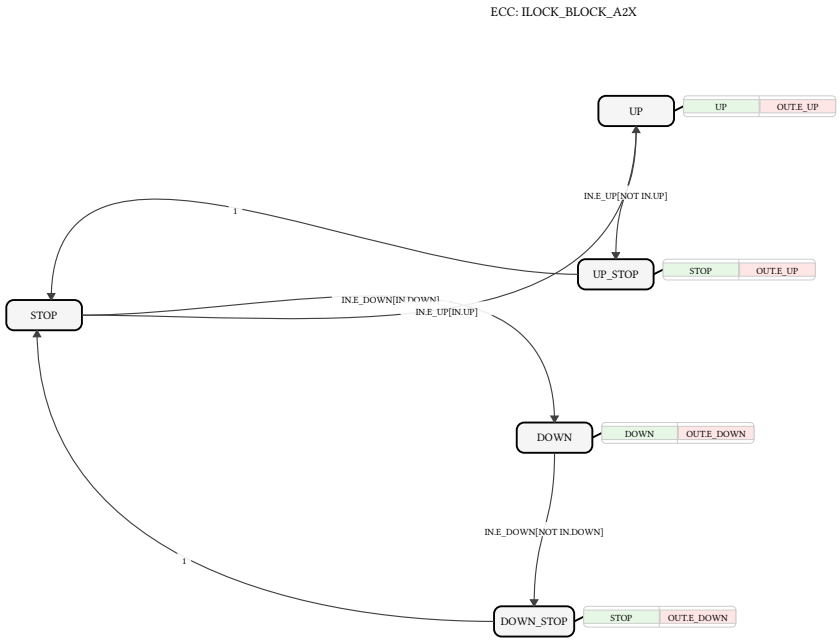
Vergleich mit ähnlichen Bausteinen

- **Set/Reset (SR/R-SR):** Erlaubt das gleichzeitige Setzen beider Richtungen, was zu undefinierten Zuständen führen kann. Der ILOCK_BLOCK verhindert dies durch strikte Verriegelung.
- **Zustandsautomat (z. B. mit mehreren Zuständen):** Bietet mehr Flexibilität, erfordert aber manuelle Implementierung der Priorisierungslogik. Der ILOCK_BLOCK kapselt diese Logik direkt.
- **Einfacher Interlock über UND-Gatter:** Reine Signalverknüpfung ignoriert die zeitliche Reihenfolge. Der ILOCK_BLOCK reagiert ereignisgesteuert auf die erste gültige Aktivierung.

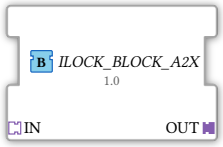
Fazit

Der **ILOCK_BLOCK** ist ein spezialisierter Funktionsbaustein für Verriegelungsanwendungen mit Priorisierung des ersten aktiven Eingangs. Durch seine klare Zustandsmaschine und die ereignisgesteuerte Verarbeitung eignet er sich besonders für zeitkritische und sicherheitsrelevante Steuerungen, bei denen widersprüchliche Signale streng ausgeschlossen werden müssen. Er bietet eine robuste Alternative zu klassischen Set/Reset-Logiken und reduziert den Implementierungsaufwand erheblich.

ILOCK_BLOCK_A2X



Input for forward/up and backward/down direction – *A2X*



A2X – Output for forward/up and backward/down direction

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ILOCK_BLOCK_A2X** realisiert eine Richtungsverriegelung für Geräte, die über eine A2X-Schnittstelle angesteuert werden. Er priorisiert die erste aktive Eingabe und stellt sicher, dass zu jedem Zeitpunkt nur eine Richtung (Aufwärts/Abwärts) aktiv ist. Die Implementierung erfolgt als Basic-Funktionsblock mit einem ECC (Execution Control Chart) und verwendet zwei A2X-Adapter: einen Eingangsadapter (IN) und einen Ausgangsadapter (OUT).

Schnittstellenstruktur

Ereignis-Eingänge

- Es sind keine direkten Ereignis-Eingänge vorhanden.
- Ereignisse werden ausschließlich über die Adapter-Ereignisse `IN.E_UP` und `IN.E_DOWN` vom angeschlossenen A2X-Adapter empfangen.

Ereignis-Ausgänge

- Es gibt keine expliziten Ereignis-Ausgänge.
- Die Ausgabe von Ereignissen erfolgt über die Adapter-Ausgänge `OUT.E_UP` und `OUT.E_DOWN` sowie die zugehörigen Datenwerte (`OUT.UP`, `OUT.DOWN`).

Daten-Eingänge

- Es sind keine direkten Dateneingänge definiert.
- Die Eingangsdaten (`UP`, `DOWN`) werden über den A2X-Adapter `IN` bereitgestellt.

Daten-Ausgänge

- Es sind keine direkten Datenausgänge definiert.

- Die Ausgangsdaten (UP, DOWN) werden über den A2X-Adapter OUT bereitgestellt.

Adapter

Der FB besitzt zwei Adapter-Schnittstellen vom Typ adapter::types::unidirectional::A2X:

- **IN** (Socket) – Eingang für die Richtungsbefehle.
  Liefert die Ereignisse E_UP und E_DOWN sowie die zugehörigen booleschen Daten UP und DOWN.
- **OUT** (Plug) – Ausgang für die geschalteten Richtungen.
  Stellt die Ereignisse E_UP und E_DOWN sowie die Datenwerte UP und DOWN bereit.

Der A2X-Adapter ist ein unidirektionaler Adaptertyp, der die beiden booleschen Variablen UP und DOWN transportiert und jeweils zwei Ereignisse (für Aufwärts- und Abwärtsbewegung) einführt.

Funktionsweise

Der FB implementiert eine Zustandsmaschine mit fünf Zuständen: STOP, UP, DOWN, UP_STOP und DOWN_STOP. Die Logik reagiert auf die Ereignisse IN.E_UP und IN.E_DOWN und prüft dabei die zugehörigen Datenwerte (IN.UP bzw. IN.DOWN).

- **Zustand STOP**: Initialer Ruhezustand.
  - Wenn IN.E_UP eintrifft und IN.UP = TRUE, wechselt der FB in den Zustand UP und setzt OUT.UP auf TRUE, OUT.DOWN auf FALSE.
  - Analog wechselt er bei IN.E_DOWN und IN.DOWN = TRUE in den Zustand DOWN und setzt OUT.DOWN auf TRUE, OUT.UP auf FALSE.
- **Zustand UP**: Richtung „Aufwärts“ ist aktiv.
  - Wenn IN.E_UP eintrifft und IN.UP = FALSE (Befehl wird zurückgenommen), wechselt der FB in den Zustand UP_STOP.
  - Im UP_STOP-Zustand wird der Ausgang OUT.E_UP ausgelöst und der Algorithmus STOP setzt OUT.UP und OUT.DOWN auf FALSE.
  - Danach kehrt der FB automatisch in den Zustand STOP zurück (Transition mit Bedingung 1).
- **Zustand DOWN**: Richtung „Abwärts“ ist aktiv.
  - Analog zum UP-Zustand wechselt er bei IN.E_DOWN und IN.DOWN = FALSE in den Zustand DOWN_STOP, setzt die Ausgänge zurück und kehrt anschließend zu STOP zurück.

Die Priorisierung ergibt sich aus der Reihenfolge der Transitionen im ECC: Wenn im Zustand STOP sowohl IN.E_UP als auch IN.E_DOWN anstehen, hat die zuerst definierte Transition (hier die nach DOWN) Vorrang. Somit gewinnt die erste aktive Eingabe.

Technische Besonderheiten

- Implementierung als **Basic FB** mit ECC und strukturierten Algorithmen in ST (Structured Text).
- Verwendung des Adaptertyps A2X (unidirektional) für die Signalein-/ausgabe.
- Keine herkömmlichen Ein-/Ausgänge (Ereignis oder Daten) – die Kommunikation erfolgt ausschließlich über Adapter.
- Der FB gehört zum Paket logiBUS::signalprocessing::interlock.
- Lizenz: Eclipse Public License 2.0 (SPDX-License-Identifier: EPL-2.0).

Zustandsübersicht

Zustand	Beschreibung
STOP	Keine Richtung aktiv. Wartet auf Befehl.
UP	Richtung „Aufwärts“ aktiv. OUT.UP = TRUE, OUT.DOWN = FALSE.
DOWN	Richtung „Abwärts“ aktiv. OUT.DOWN = TRUE, OUT.UP = FALSE.
UP_STOP	Übergang vom UP-Zustand: Richtung wird gestoppt, danach Rückkehr zu STOP.
DOWN_STOP	Übergang vom DOWN-Zustand: Richtung wird gestoppt, danach Rückkehr zu STOP.

Anwendungsszenarien

- Steuerung von Motoren oder Antrieben, die nur eine Richtung gleichzeitig zulassen (z. B. Hubwerke, Schiebetore, Förderbänder).
- Verriegelung von Bewegungsrichtungen in sicherheitsrelevanten Anwendungen.

- Übernahme von Richtungskommandos über eine serielle Schnittstelle, die über den A2X-Adapter angebunden ist.

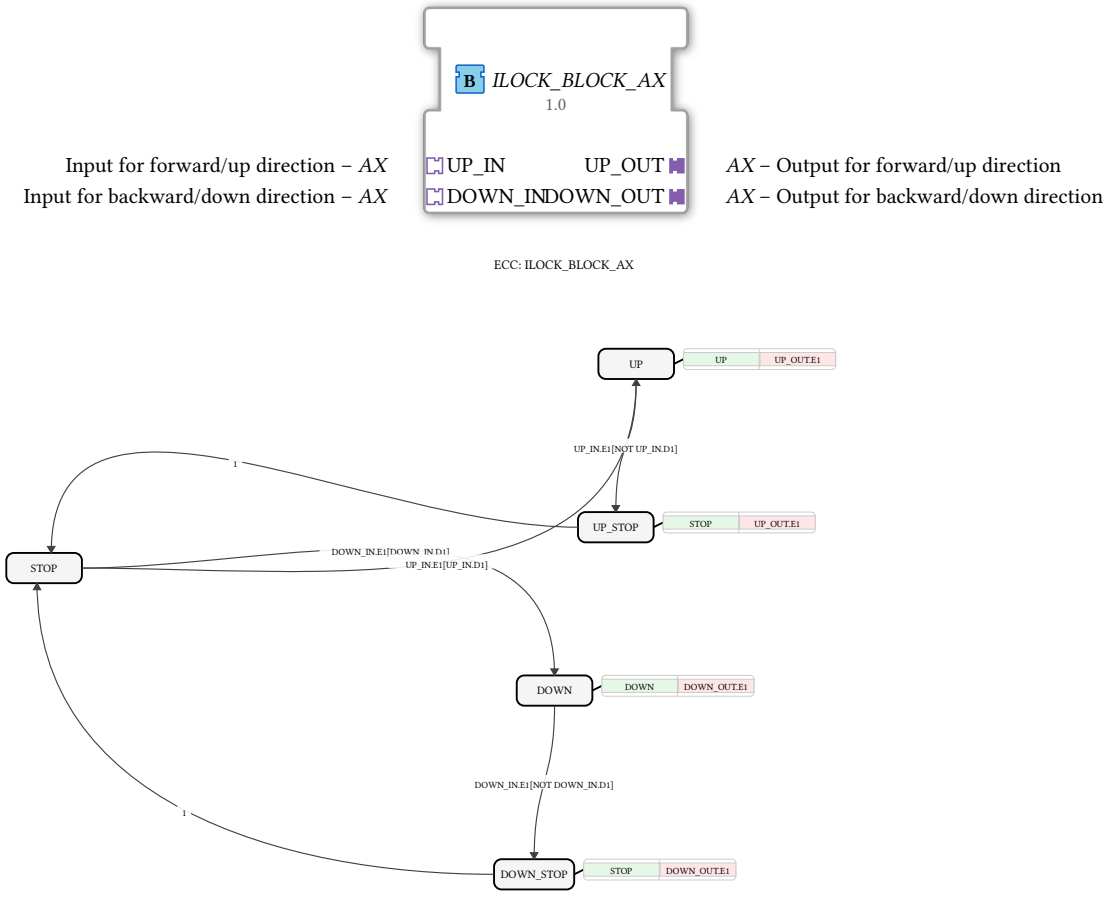
Vergleich mit ähnlichen Bausteinen

Ähnliche Bausteine wie ein SR-Latch oder ein allgemeiner Interlock-Baustein steuern ebenfalls sich gegenseitig ausschließende Ausgänge. Der ILOCK_BLOCK_A2X unterscheidet sich durch seine adapterbasierte Schnittstelle (`A2X`), die Ereignis-/Datenstruktur und die explizite Priorisierungslogik innerhalb der Zustandsmaschine. Dadurch ist er besonders für modulare Systeme geeignet, die eine einheitliche Adapterkommunikation verwenden.

Fazit

Der ILOCK_BLOCK_A2X ist ein kompakter, auf Adapterbasis arbeitender Funktionsblock für die sichere Richtungsverriegelung. Durch die klare Zustandsmaschine und die definierten Übergänge bietet er eine zuverlässige und einfache Lösung für Anwendungen, die eine gegenseitige Ausschließung von Auf-/Abwärtsbewegungen erfordern. Die Abwesenheit herkömmlicher Ein-/Ausgänge macht ihn besonders geeignet für modulare Systeme, die auf Adapter-Kommunikation aufbauen.

ILOCK_BLOCK_AX



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ILOCK_BLOCK_AX** realisiert eine Verriegelung (Interlock) für zwei gegenläufige Richtungen (aufwärts/abwärts oder vorwärts/rückwärts). Er priorisiert den ersten aktiven Eingang und stellt sicher, dass nur eine Richtung gleichzeitig aktiv sein kann. Der Block ist als Basis-Baustein (Basic FB) gemäß IEC 61499-2 implementiert.

Schnittstellenstruktur

Der Baustein verwendet Adapter-Schnittstellen vom Typ `adapter::types::unidirectional::AX`. Jeder Adapter besteht aus einem Ereignis-Eingang `E1` und einem Daten-Eingang `D1` (BOOL).

Ereignis-Eingänge

Keine diskreten Ereignis-Eingänge. Ereignisse werden über die Adapter-Schnittstellen empfangen:

- `UP_IN.E1` – Ereignis für die Aufwärts-/Vorwärts-Richtung
- `DOWN_IN.E1` – Ereignis für die Abwärts-/Rückwärts-Richtung

Ereignis-Ausgänge

Keine diskreten Ereignis-Ausgänge. Ausgaben erfolgen über die Adapterplugs:

- `UP_OUT.E1` – Ereignis bei Aktivierung/Deaktivierung der Aufwärts-Richtung
- `DOWN_OUT.E1` – Ereignis bei Aktivierung/Deaktivierung der Abwärts-Richtung

Daten-Eingänge

- `UP_IN.D1` (BOOL) – Gültigkeitssignal für die Aufwärts-Richtung
- `DOWN_IN.D1` (BOOL) – Gültigkeitssignal für die Abwärts-Richtung

Daten-Ausgänge

- `UP_OUT.D1` (BOOL) – Gültigkeit der Aufwärts-Richtung am Ausgang
- `DOWN_OUT.D1` (BOOL) – Gültigkeit der Abwärts-Richtung am Ausgang

Adapter

- **Sockets (Eingänge):**
  - `UP_IN` – Adapter für Aufwärts-Steuerung
  - `DOWN_IN` – Adapter für Abwärts-Steuerung
- **Plugs (Ausgänge):**
  - `UP_OUT` – Adapter für Aufwärts-Ausgabe
  - `DOWN_OUT` – Adapter für Abwärts-Ausgabe

Funktionsweise

Der Baustein arbeitet mit einem endlichen Automaten (ECC). Er startet im Zustand **STOP**. Sobald eines der beiden Ereignisse eintrifft und das zugehörige `D1`-Signal `TRUE` ist, wechselt er in den entsprechenden Richtungszustand. Solange `D1` des aktiven Eingangs `TRUE` bleibt, wird der Zustand gehalten. Wird das `D1`-Signal des aktiven Eingangs auf `FALSE` gesetzt (bei erneutem Ereignis), wechselt der Automat zuerst in einen **Stopp-Zwischenzustand** (UP_STOP oder DOWN_STOP) und dann automatisch zurück nach **STOP**. Der jeweils andere Eingang wird ignoriert, solange eine Richtung aktiv ist – der Baustein priorisiert den zuerst aktiven Eingang.

Die Algorithmen setzen die Ausgangssignale:

- **UP**: `UP_OUT.D1 := TRUE`, `DOWN_OUT.D1 := FALSE`
- **DOWN**: `UP_OUT.D1 := FALSE`, `DOWN_OUT.D1 := TRUE`
- **STOP**: Beide Ausgänge auf `FALSE`

Technische Besonderheiten

- **Priorisierung erster aktiver Eingang:** Nur der zuerst eintreffende gültige Befehl wird ausgeführt; der andere wird blockiert, bis der aktive deaktiviert wird.
- **Adapter-basierte Schnittstelle:** Die Verwendung von Adaptern des Typs `unidirectional::AX` erlaubt eine flexible Kopplung mit anderen Bausteinen ohne direkte Event-/Datentrennung.
- **Verriegelung auf Automatenebene:** Durch die Zustandsübergänge wird hardwarenahe Sicherheit simuliert – es kann nie gleichzeitig `UP_OUT.D1` und `DOWN_OUT.D1 = TRUE` auftreten.
- **Zwischenzustände:** Die Zustände `UP_STOP` und `DOWN_STOP` gewährleisten, dass die Deaktivierung sauber mit einem Ereignis auf dem entsprechenden Ausgangsadapter quittiert wird.

Zustandsübersicht

Zustand	Beschreibung	Ausgangssignale

STOP	Ruhezustand, keine Richtung aktiv	`UP_OUT.D1 = FALSE, DOWN_OUT.D1 = FALSE`
UP	Aufwärts-Richtung aktiv	`UP_OUT.D1 = TRUE, DOWN_OUT.D1 = FALSE`
DOWN	Abwärts-Richtung aktiv	`UP_OUT.D1 = FALSE, DOWN_OUT.D1 = TRUE`
UP_STOP	Deaktivierung von UP (wird sofort verlassen)	`UP_OUT.D1 = FALSE, DOWN_OUT.D1 = FALSE`
DOWN_STOP	Deaktivierung von DOWN (wird sofort verlassen)	`UP_OUT.D1 = FALSE, DOWN_OUT.D1 = FALSE`

Übergänge:

- `STOP` → `UP` bei Ereignis von `UP_IN` mit `UP_IN.D1 = TRUE`
- `STOP` → `DOWN` bei Ereignis von `DOWN_IN` mit `DOWN_IN.D1 = TRUE`
- `UP` → `UP_STOP` bei Ereignis von `UP_IN` mit `UP_IN.D1 = FALSE`
- `DOWN` → `DOWN_STOP` bei Ereignis von `DOWN_IN` mit `DOWN_IN.D1 = FALSE`
- `UP_STOP` → `STOP` automatisch (Transition 1)
- `DOWN_STOP` → `STOP` automatisch (Transition 1)

Anwendungsszenarien

- **Fahrsteuerung (z. B. Hubtische, Förderbänder):** Verhindert gleichzeitige Auf-/Ab-Bewegung.
- **Ventilsteuerung:** Öffnen/Schließen eines Schiebers mit gegenseitiger Verriegelung.
- **Sicherheitsgerichtete Logik:** Absicherung, dass nie beide Antriebsrichtungen gleichzeitig angesteuert werden.
- **Steuerung von Weichen oder Klappen:** Nur eine Position darf aktiv sein.

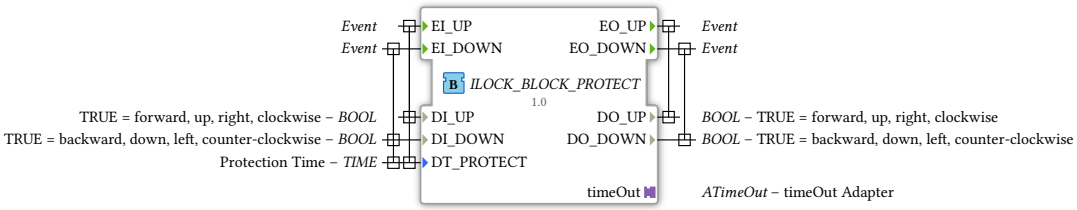
Vergleich mit ähnlichen Bausteinen

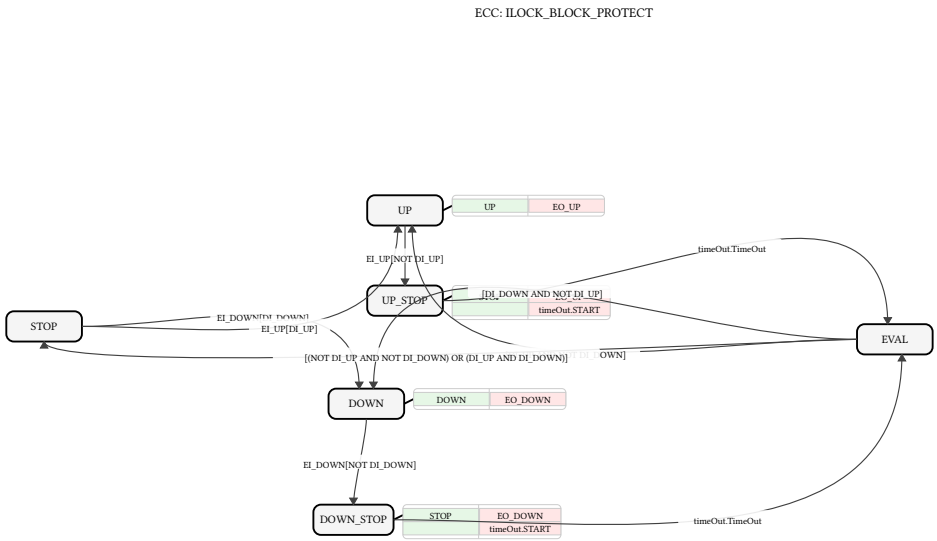
Gegenüber einem einfachen RS-Flipflop bietet `ILOCK_BLOCK_AX` eine explizite Verriegelung und die Berücksichtigung von Gültigkeitssignalen (`D1`). Ein herkömmlicher SR-Baustein würde ohne externe Logik eine gleichzeitige Aktivierung beider Ausgänge zulassen. Dieser Baustein garantiert die gegenseitige Ausschließlichkeit auf Zustandsebene.

Fazit

`ILOCK_BLOCK_AX` ist ein kompakter, sicherheitsorientierter Funktionsblock für die Verriegelung zweier gegensätzlicher Richtungen. Seine Adapter-basierte Schnittstelle und der klar definierte Automat machen ihn einfach integrierbar und zuverlässig in Anwendungen, die eine exklusive Richtungssteuerung erfordern.

ILOCK_BLOCK_PROTECT





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `ILOCK_BLOCK_PROTECT` realisiert eine interlock-geschützte Richtungssteuerung mit einstellbarer Totzeit. Sobald ein aktiver Eingang (z. B. `EI_UP` mit `DI_UP = TRUE`) erkannt wird, wird dieser priorisiert und alle gegensätzlichen Signale werden solange ignoriert, bis der aktive Eingang zurückgesetzt wird. Nach dem Rücksetzen läuft eine konfigurierbare Schutzzeit (`DT_PROTECT`) ab, bevor eine neue Richtung aktiviert werden kann. Dadurch werden unerwünschte Richtungswechsel oder Kurzschlusszustände sicher verhindert.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	mit Variablen	Beschreibung
`EI_UP`	`DI_UP`, `DT_PROTECT`	Ereignis zur Aktivierung der Vorwärts-Richtung
`EI_DOWN`	`DI_DOWN`, `DT_PROTECT`	Ereignis zur Aktivierung der Rückwärts-Richtung

Ereignis-Ausgänge

Ereignis	mit Variablen	Beschreibung
`EO_UP`	`DO_UP`	Quittierung der aktiven Vorwärts-Richtung
`EO_DOWN`	`DO_DOWN`	Quittierung der aktiven Rückwärts-Richtung

Daten-Eingänge

Variable	Typ	Initialwert	Beschreibung
`DI_UP`	BOOL	–	`TRUE` = vorwärts, aufwärts, rechts, im Uhrzeigersinn
`DI_DOWN`	BOOL	–	`TRUE` = rückwärts, abwärts, links, gegen Uhrzeigersinn
`DT_PROTECT`	TIME	`T#50ms`	Schutzzeit (Totzeit) nach Rücksetzen einer Richtung

Daten-Ausgänge

Variable	Typ	Beschreibung
DO_UP	BOOL	TRUE = Vorwärts-Richtung aktiv
DO_DOWN	BOOL	TRUE = Rückwärts-Richtung aktiv

Adapter

Adapter	Typ	Beschreibung
timeOut	iec61499::events::ATimeOut	Adapter für die zeitliche Steuerung der Totzeit

Funktionsweise

Der Baustein arbeitet nach dem Prinzip der **ersten Priorität**:

1. **Initialzustand (STOP)**
   Beide Ausgänge sind FALSE. Wird ein Ereignis mit gültiger Bedingung empfangen (z. B. EI_UP bei DI_UP = TRUE), wechselt der Zustand in die entsprechende Richtung (UP oder DOWN).
2. **Richtungszustände (UP / DOWN)**
   Der zugehörige Ausgang (DO_UP oder DO_DOWN) wird auf TRUE gesetzt, der andere auf FALSE.
   Solange der aktive Eingang bestehen bleibt, werden neue Ereignisse ignoriert (insbesondere gegensätzliche).
   Ein neues Ereignis mit demselben Eingang wird nur dann verarbeitet, wenn der Eingang zuvor auf FALSE gefallen ist (negative Flanke) – siehe UP_STOP/DOWN_STOP.
3. **Rückstellen in die Schutzphase (UP_STOP / DOWN_STOP)**
   Wird der aktive Eingang zurückgesetzt (z. B. DI_UP von TRUE auf FALSE), so wird der Ausgang sofort auf FALSE gesetzt und der Timer timeOut gestartet. Die Totzeit DT_PROTECT beginnt zu laufen.
4. **Auswertungszustand (EVAL)**
   Nach Ablauf der Totzeit verlässt der Baustein die Schutzphase und geht in den EVAL-Zustand. Hier wird anhand der aktuellen Eingänge entschieden:
   - DI_UP = TRUE und DI_DOWN = FALSE ⟶ Übergang nach UP
   - DI_DOWN = TRUE und DI_UP = FALSE ⟶ Übergang nach DOWN
   - Beide FALSE oder beide TRUE ⟶ Rückkehr in STOP

Wichtig: Solange im Zustand UP/DOWN ein neues Ereignis eintrifft, während der zugehörige Eingang noch TRUE ist, wird dieses Ereignis ignoriert (kein Zustandswechsel). Erst bei negativer Flanke wird die Stopp-Phase eingeleitet.

Technische Besonderheiten

- **Interner Zeitgeber** über den Adapter timeOut (Typ ATimeOut) – die Schutzzeit wird durch jeden Zustandsübergang, der eine Richtung deaktiviert, gestartet.
- **Keine gleichzeitigen Ausgänge** – zu keiner Zeit sind DO_UP und DO_DOWN gleichzeitig TRUE. Im EVAL-Zustand sind beide Ausgänge FALSE.
- **Konfigurierbare Totzeit** über den Eingang DT_PROTECT (Werksvorgabe 50 ms).
- **Kompakte Implementierung** als Basic Function Block mit endlicher Zustandsmaschine (6 Zustände).
- Die Verriegelung verhindert nicht nur gegensätzliche Kommandos, sondern erzwingt auch eine Mindestpause zwischen zwei Richtungswechseln.

Zustandsübersicht

Zustand	Beschreibung
STOP	Ruhezustand: beide Ausgänge FALSE, warten auf erstes gültiges Ereignis
UP	Vorwärts-Richtung aktiv: DO_UP = TRUE, DO_DOWN = FALSE
DOWN	Rückwärts-Richtung aktiv: DO_DOWN = TRUE, DO_UP = FALSE
UP_STOP	Schutzphase nach Rücksetzen der Vorwärts-Richtung: DO_UP auf FALSE, Timer läuft

DOWN_STOP	Schutzphase nach Rücksetzen der Rückwärts-Richtung: DO_DOWN auf FALSE, Timer läuft
EVAL	Auswertungszustand nach Timerablauf: Entscheidung über nächste Richtung oder Rückkehr zu STOP

Anwendungsszenarien

- **Richtungssteuerung von Motoren** (z. B. Förderbänder, Hubwerke, Drehtore) – verhindert gleichzeitige Ansteuerung in beide Richtungen und erzwungene Totzeit für den mechanischen Richtungswechsel.
- **Verriegelung von Ventilen oder Klappen** – z. B. Auf/Zu-Steuerung mit Schutz vor schnellem Wechsel, um mechanische Belastung zu vermeiden.
- **Sicherheitsgerichtete Steuerungen** – als Teil einer einfachen Interlock-Logik, wenn kein sicherheitszertifizierter Baustein erforderlich ist.
- **Steuerung von Zuführungen** in der Agrartechnik (siehe Copyright HR Agrartechnik GmbH) oder in der Fördertechnik.

Vergleich mit ähnlichen Bausteinen

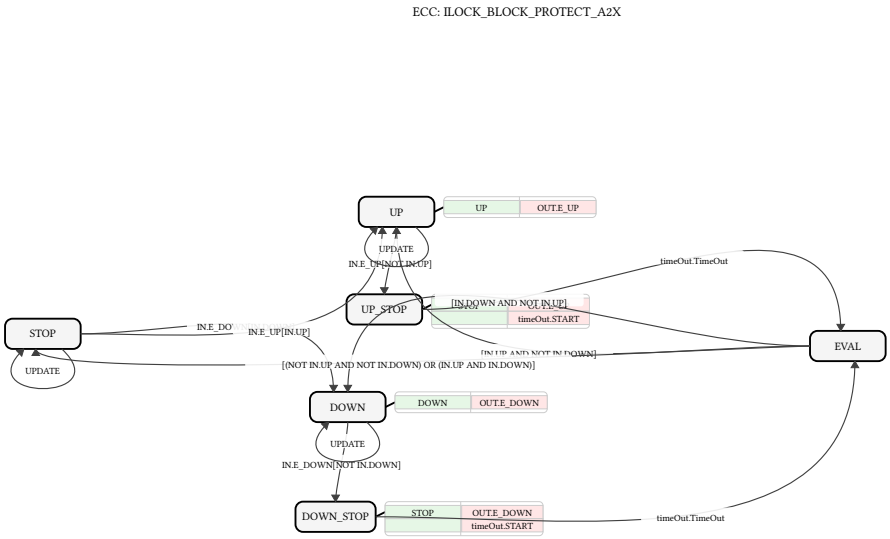
Baustein	Eigenschaften
SR-Flipflop	Einfache Set/Reset-Logik, keine Totzeit, kein Schutz gegen gleichzeitige Signale
ILOCK_BLOCK_PROTECT	Priorisierung des ersten aktiven Eingangs, Totzeit nach jedem Richtungswechsel, beide Ausgänge nie gleichzeitig TRUE
Interlock-Baustein ohne Timer	Nur Sperrlogik, sofortige Umschaltung möglich, keine Schutzzeit
RS-Sperre mit Zeitverzögerung	Ähnlich, aber oft weniger konfigurierbar und ereignisgesteuert

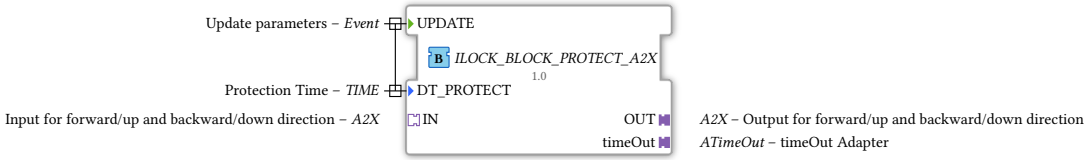
Der ILOCK_BLOCK_PROTECT bietet eine integrierte, konfigurierbare Totzeit und ist speziell für ereignisgesteuerte Systeme nach IEC 61499 optimiert.

Fazit

Der ILOCK_BLOCK_PROTECT eignet sich ideal für Anwendungen, die eine zuverlässige Richtungsverriegelung mit einstellbarer Schutzzeit erfordern. Durch die klare Zustandsmaschine und die Nutzung des Standard-ATimeOut-Adapters ist er einfach in größere Steuerungsprojekte integrierbar. Die Priorisierung des ersten aktiven Eingangs sorgt für deterministisches Verhalten, und die erzwungene Totzeit schützt sowohl Mechanik als auch Steuerungslogik vor unerwünschten Zuständen.

ILOCK_BLOCK_PROTECT_A2X





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `ILOCK_BLOCK_PROTECT_A2X` realisiert eine **Interlock-Logik mit Schutzzeit**. Er priorisiert den ersten aktiven Eingang (Vorwärts/Rückwärts bzw. Auf/Ab) und verhindert durch eine einstellbare Totzeit ein unmittelbares Umschalten zwischen den beiden Richtungen. Die Richtungssignale werden über einen bidirektionalen A2X-Adapter bereitgestellt, die Schutzzeit wird über einen Timeout-Adapter gesteuert. Dieser Baustein eignet sich besonders für sicherheitsrelevante Antriebssteuerungen, bei denen ein sofortiger Richtungswechsel unerwünscht ist.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung
`UPDATE`	Aktualisiert den Parameter `DT_PROTECT` (wird im aktuellen Zustand verarbeitet)

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge vorhanden.
Alle Ereignis-Ausgänge werden über den **Adapter `OUT`** (A2X-Typ) bereitgestellt: `OUT.E_UP` und `OUT.E_DOWN`.

Daten-Eingänge

Daten	Typ	Beschreibung

DT_PROTECT	TIME	Schutzzeit (Totzeit) für das Umschalten zwischen Auf/Ab bzw. Vorwärts/Rückwärts. Initialwert: T#50ms

Daten-Ausgänge

Keine direkten Daten-Ausgänge.
Die Ausgangsdaten OUT.UP und OUT.DOWN (BOOL) werden über den **Adapter OUT** bereitgestellt.

Adapter

Adapter	Richtung	Typ	Beschreibung
IN	Socket (Eingang)	adapter::types::unidirectional::A2X	Eingang für die Richtungssignale (Vorwärts/Up bzw. Rückwärts/Down) und deren zugehörige Ereignisse (E_UP, E_DOWN)
OUT	Plug (Ausgang)	adapter::types::unidirectional::A2X	Ausgang für die priorisierten Richtungssignale (UP, DOWN) und die zugehörigen Ereignisse (E_UP, E_DOWN)
timeOut	Plug (Ausgang)	iec61499::events::ATimeOut	Timer-Adapter zur Realisierung der Schutzzeit; erhält die Dauer DT_PROTECT und liefert das Ereignis TimeOut

Funktionsweise

Der Baustein arbeitet als endlicher Automat (ECC) und folgt einer klaren Prioritätslogik:

1. **STOP** – Warten auf ein gültiges Ereignis:
   - Bei IN.E_UP mit IN.UP = TRUE ⟶ Wechsel in Zustand **UP**.
   - Bei IN.E_DOWN mit IN.DOWN = TRUE ⟶ Wechsel in Zustand **DOWN**.
2. **UP** – Aktiver Zustand „Aufwärts“:
   - Setzt OUT.UP = TRUE und OUT.DOWN = FALSE.
   - Bei IN.E_UP mit IN.UP = FALSE (d.h. das Aufwärts-Signal fällt ab) ⟶ Wechsel in Zustand **UP_STOP**.
3. **DOWN** – Aktiver Zustand „Abwärts“:
   - Setzt OUT.UP = FALSE und OUT.DOWN = TRUE.
   - Bei IN.E_DOWN mit IN.DOWN = FALSE ⟶ Wechsel in Zustand **DOWN_STOP**.
4. **UP_STOP / DOWN_STOP** – Schutzphase:
   - Entsprechendes Ausgangssignal wird auf FALSE gesetzt (OUT.UP bzw. OUT.DOWN).
   - Der Timer timeOut wird mit DT_PROTECT gestartet.
   - Nach Ablauf des Timers (timeOut.TimeOut) erfolgt der Wechsel in den Zustand **EVAL**.
5. **EVAL** – Auswertung der aktuellen Eingänge nach Ablauf der Schutzzeit:
   - Ist IN.UP = TRUE und IN.DOWN = FALSE ⟶ Wechsel in **UP**.
   - Ist IN.DOWN = TRUE und IN.UP = FALSE ⟶ Wechsel in **DOWN**.
   - Sind beide Eingänge aktiv oder keiner aktiv ⟶ Wechsel zurück in **STOP**.

Während jedes Zustands kann das Ereignis UPDATE eintreten, das lediglich den Parameter DT_PROTECT aktualisiert und den aktuellen Zustand beibehält.

Die Zustandsübergänge und die zugehörigen Algorithmen (UP, DOWN, STOP) gewährleisten, dass immer nur eine Richtung aktiv ist und ein Richtungswechsel erst nach Ablauf der Schutzzeit möglich ist.

Technische Besonderheiten

- **Prioritätslogik mit Schutzzeit:** Das erste aktive Eingangssignal wird übernommen und erst nach Ablauf der einstellbaren Totzeit darf ein Richtungswechsel stattfinden. Dadurch werden ungewollte Schaltspitzen und mechanische Belastungen vermieden.

- **Verwendung des A2X-Adapters:** Die Richtungssignale werden über bidirektionale A2X-Adapter ein- und ausgegeben. Dies erlaubt eine konsistente Übertragung von Zustand und Ereignissen für Auf/Ab- bzw. Vorwärts/Rückwärts-Bewegungen.
- **Timer-Adapter:** Die Schutzzeit wird über den standardisierten `ATimeOut`-Adapter realisiert. Dadurch ist die Verzögerung exakt steuerbar und kann zur Laufzeit über `DT_PROTECT` angepasst werden.
- **Endlicher Automat:** Die klare ECC-Struktur erleichtert die formale Verifikation und Wartung des Bausteins.
- **Keine direkten Ausgangs-Events:** Alle Ausgangsereignisse werden über den OUT-Adapter bereitgestellt, was die Wiederverwendbarkeit erhöht.

Zustandsübersicht

Zustand	Bedeutung	Ausgang (OUT.UP / OUT.DOWN)	Timer-Status
`STOP`	Warten auf ein gültiges Eingangssignal	`FALSE` / `FALSE`	inaktiv
`UP`	Aufwärtsbewegung aktiv	`TRUE` / `FALSE`	inaktiv
`DOWN`	Abwärtsbewegung aktiv	`FALSE` / `TRUE`	inaktiv
`UP_STOP`	Schutzphase nach Ende der Aufwärtsbewegung	`FALSE` / `FALSE`	läuft
`DOWN_STOP`	Schutzphase nach Ende der Abwärtsbewegung	`FALSE` / `FALSE`	läuft
`EVAL`	Auswertung der Eingänge nach Ablauf der Schutzzeit	`FALSE` / `FALSE` (temporär)	inaktiv

Übergang	Bedingung	Aktion
`STOP → UP`	`IN.E_UP` mit `IN.UP = TRUE`	–
`STOP → DOWN`	`IN.E_DOWN` mit `IN.DOWN = TRUE`	–
`UP → UP_STOP`	`IN.E_UP` mit `IN.UP = FALSE`	`STOP`-Algorithmus, Start Timer
`DOWN → DOWN_STOP`	`IN.E_DOWN` mit `IN.DOWN = FALSE`	`STOP`-Algorithmus, Start Timer
`UP_STOP → EVAL`	`timeOut.TimeOut`	–
`DOWN_STOP → EVAL`	`timeOut.TimeOut`	–
`EVAL → UP`	`IN.UP = TRUE` und `IN.DOWN = FALSE`	–
`EVAL → DOWN`	`IN.DOWN = TRUE` und `IN.UP = FALSE`	–
`EVAL → STOP`	beide Eingänge aktiv oder beide inaktiv	–
`* → *` (Selbstschleife)	`UPDATE`	Parameter `DT_PROTECT` wird aktualisiert

Anwendungsszenarien

- **Antriebssteuerungen:** Schutz vor zu schnellem Richtungswechsel bei Förderbändern, Hubwerken oder Verfahrwerken, um mechanische Belastungen zu reduzieren.
- **Sicherheitsinterlocks:** In Maschinensteuerungen, bei denen ein sofortiges Umschalten zwischen zwei Bewegungsrichtungen verboten ist.
- **Automatisierte Prozesse:** Wenn ein definiertes Zeitfenster zwischen zwei gegenläufigen Aktionen erforderlich ist (z. B. bei Ventil- oder Klappensteuerungen).
- **Integration in bestehende 4diac-Applikationen:** Dank der A2X-Schnittstellen können die Signale direkt mit anderen A2X-fähigen Bausteinen verbunden werden.

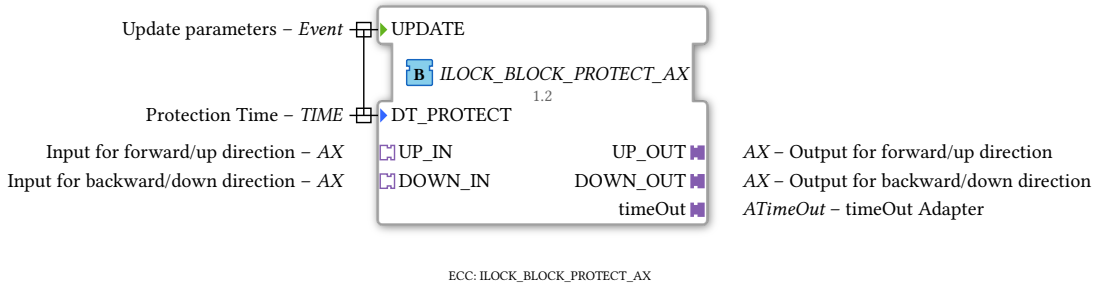
Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem einfachen Interlock-Baustein (der nur eine Prioritätslogik ohne Zeitverzögerung bietet) zeichnet sich `ILOCK_BLOCK_PROTECT_A2X` durch die integrierte Schutzzeit aus. Dadurch wird ein unkontrolliertes Oszillieren zwischen den Zuständen verhindert. Gegenüber Bausteinen, die separate Timer-Funktionen verwenden, ist hier die Zeitsteuerung direkt in die Zustandsmaschine integriert, was die Anwendung vereinfacht. Der Einsatz des standardisierten A2X-Adapters stellt sicher, dass die Daten- und Ereignisschnittstellen einheitlich und wiederverwendbar sind.

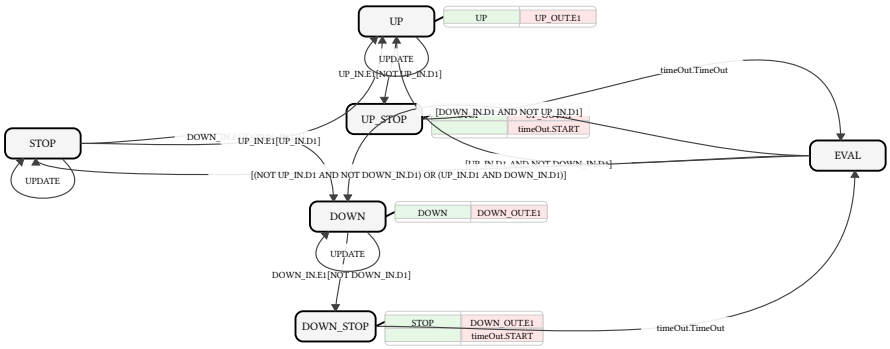
Fazit

Der FB `ILOCK_BLOCK_PROTECT_A2X` bietet eine robuste und flexible Lösung für Interlock-Anforderungen mit Schutzzeit. Die klare Zustandsmaschine und die Verwendung standardisierter Adapter machen ihn einfach in industriellen Steuerungssystemen einsetzbar. Durch die einstellbare Parameter `DT_PROTECT` kann die Schaltverzögerung an die jeweilige Anwendung angepasst werden. Somit ist dieser Baustein eine zuverlässige Komponente für sicherheitsbewusste Ansteuerungen von Antrieben und Aktoren.

ILOCK_BLOCK_PROTECT_AX



ECC: ILOCK_BLOCK_PROTECT_AX



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `ILOCK_BLOCK_PROTECT_AX` realisiert eine priorisierte Verriegelung (Interlock) für zwei gegenläufige Richtungen (UP/DOWN) unter Berücksichtigung einer konfigurierbaren Schutz-Totzeit. Der Baustein ist als Adapter-Version ausgeführt, d. h. die Ein- und Ausgänge werden über IEC 61499-Adapter bereitgestellt. Ziel ist die sichere Steuerung von Aktoren, bei denen nur eine Richtung gleichzeitig aktiv sein darf, beispielsweise bei Hebe- oder Schwenkbewegungen. Der erste aktivierte Eingang erhält den Vorrang; ein Wechsel der Richtung erfolgt erst nach Ablauf der Schutzzeit und einer erneuten Bewertung.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Mitgeführte Daten	Beschreibung
`UPDATE`	Event	`DT_PROTECT`	Ereignis zum dynamischen Aktualisieren der Schutzzeit `DT_PROTECT`.

Ereignis-Ausgänge

Der FB besitzt keine eigenständigen Ereignis-Ausgänge. Ereignisse werden über die als *Plugs* ausgeführten Adapter `UP_OUT`, `DOWN_OUT` und `timeOut` bereitgestellt.

Adapter-Ausgang	Ereignis	Beschreibung
`UP_OUT`	`E1`	Wird ausgelöst, wenn der UP-Zustand aktiv ist oder der UP_STOP-Zustand erreicht wird.
`DOWN_OUT`	`E1`	Wird ausgelöst, wenn der DOWN-Zustand aktiv ist oder der DOWN_STOP-Zustand erreicht wird.
`timeOut`	`TimeOut`	Wird ausgelöst, wenn die eingestellte Schutzzeit abgelaufen ist (vom Adapter selbst generiert).

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung
`DT_PROTECT`	TIME	`T#50ms`	Schutzzeit, die nach einer Deaktivierung eines Eingangs verstreicht, bevor eine erneute Bewertung der Eingänge erfolgt.

Daten-Ausgänge

Der FB besitzt keine eigenständigen Daten-Ausgänge. Ausgangsdaten werden über die Adapter `UP_OUT`, `DOWN_OUT` und `timeOut` bereitgestellt.

Adapter-Ausgang	Daten	Beschreibung
`UP_OUT`	`D1` (BOOL)	Signalisiert den aktiven UP-Zustand (`TRUE`) oder inaktiven UP-Zustand (`FALSE`).
`DOWN_OUT`	`D1` (BOOL)	Signalisiert den aktiven DOWN-Zustand (`TRUE`) oder inaktiven DOWN-Zustand (`FALSE`).
`timeOut`	`DT` (TIME)	Wird vom Algorithmus auf den Wert von `DT_PROTECT` gesetzt und bestimmt die Timer-Dauer.

Adapter

Typ	Name	Richtung	Beschreibung
`adapter::types::unidirectional::AXUP_IN`		Socket	Eingang für die UP-Richtung. Das Event `E1` (steigende Flanke) aktiviert die Verarbeitung; `D1` (BOOL) gibt den gewünschten Zustand vor.
`adapter::types::unidirectional::AXDOWN_IN`		Socket	Eingang für die DOWN-Richtung. Analog zu `UP_IN`.
`adapter::types::unidirectional::AXUP_OUT`		Plug	Ausgang für die UP-Richtung. Stellt Ereignis und Daten für die angeschlossene Logik bereit.
`adapter::types::unidirectional::AXDOWN_OUT`		Plug	Ausgang für die DOWN-Richtung. Stellt Ereignis und Daten bereit.

iec61499::events::ATimeOut	timeOut	Plug	Adapter für einen Timer. Der FB setzt die Zeitdauer (DT) und startet den Timer über das Ereignis START. Das Ereignis TimeOut signalisiert den Ablauf.

Funktionsweise

Der Baustein arbeitet als zustandsgesteuerte Verriegelung mit zwei Prioritätseingängen. Die grundsätzliche Funktionsweise:

1. **Initialzustand** (STOP): Keine Richtung ist aktiv. Ein eingehendes Ereignis an UP_IN.E1 oder DOWN_IN.E1 wird nur dann verarbeitet, wenn das zugehörige Datensignal D1 den Wert TRUE hat. Der erste Eingang, der diese Bedingung erfüllt, wechselt in den entsprechenden Zustand (UP oder DOWN). Sind beide gleichzeitig TRUE, wird keiner akzeptiert – der Baustein bleibt im STOP.
2. **Aktiver Zustand** (UP oder DOWN): Der Ausgangsadapter (UP_OUT oder DOWN_OUT) signalisiert die aktive Richtung über D1 = TRUE und löst das Ereignis E1 aus. Die Schutzzeit wird im Timer gespeichert, jedoch aktiv noch nicht gestartet.
3. **Deaktivierung und Schutzzeit**: Wenn der aktive Eingang sein Datensignal auf FALSE (oder das Ereignis ohne TRUE) zurücksetzt, wechselt der FB in den entsprechenden Stop-Zustand (UP_STOP oder DOWN_STOP). Dabei wird der Ausgang auf FALSE gesetzt und der Timer über das Ereignis timeOut.START gestartet. Nach Ablauf der eingestellten Schutzzeit (timeOut.TimeOut) wechselt der FB in den Evaluierungszustand (EVAL).
4. **Evaluierung** (EVAL): Im Evaluierungszustand werden die aktuellen Datensignale beider Eingänge ausgewertet:
   - Nur UP_IN.D1 = TRUE ⟶ Wechsel zu UP
   - Nur DOWN_IN.D1 = TRUE ⟶ Wechsel zu DOWN
   - Keines oder beide TRUE ⟶ Wechsel zurück zu STOP
5. **Dynamische Aktualisierung**: Das Ereignis UPDATE erlaubt es, während des Betriebs die Schutzzeit DT_PROTECT zu ändern. Die aktualisierte Zeit wird bei nächster Verwendung des Timers wirksam (der Timer wird nur in den Stop-Zuständen gestartet). Das Ereignis kann in jedem Zustand empfangen werden, ohne den aktuellen Steuerungszustand zu verlassen.

Technische Besonderheiten

- **Adapter-Schnittstelle**: Der gesamte Datenaustausch erfolgt über Adapter. Dadurch wird eine saubere Trennung zwischen Steuerungslogik und Peripherie erreicht, die Wiederverwendbarkeit erhöht und die Kapselung verbessert.
- **Dynamische Schutzzeit**: Über das UPDATE-Ereignis kann die Totzeit zur Laufzeit verändert werden, ohne den Baustein neu zu starten. Dies ist besonders in Applikationen mit variablen Schaltzeiten (z. B. temperaturabhängige Verzögerungen) nützlich.
- **Totzeit-Prinzip**: Der FB geht nach Deaktivierung eines Eingangs nicht sofort in den anderen Zustand, sondern erzwingt eine minimale Verzögerung (DT_PROTECT). Dies verhindert Prellen oder ungewollte schnelle Umschaltungen.
- **Priorisierung nach erstem Aktiv**: Die Logik reagiert nur auf das erste TRUE-Signal eines Eingangsereignisses. Gleichzeitige Aktivierung beider Eingänge führt zu keiner Richtungsänderung.
- **Selbsthaltung ohne externes Ereignis**: Solange ein aktiver Eingang sein D1 auf TRUE hält, bleibt der zugehörige Ausgangszustand erhalten, auch ohne weitere Ereignisse.

Zustandsübersicht

Zustand	Beschreibung	Aktivitäten
STOP	Initial- und Ruhezustand. Keine Richtung aktiv.	–
UP	UP-Richtung aktiv.	Algorithmus UP: setzt UP_OUT.D1 = TRUE, DOWN_OUT.D1 = FALSE, überträgt

		DT_PROTECT an timeOut.DT und sendet UP_OUT.E1.
DOWN	DOWN-Richtung aktiv.	Algorithmus DOWN: setzt DOWN_OUT.D1 = TRUE, UP_OUT.D1 = FALSE, überträgt DT_PROTECT an timeOut.DT und sendet DOWN_OUT.E1.
UP_STOP	Wartezeit nach Deaktivierung von UP.	Algorithmus STOP: setzt beide Ausgänge auf FALSE, überträgt DT_PROTECT an timeOut.DT und startet den Timer mit timeOut.START.
DOWN_STOP	Wartezeit nach Deaktivierung von DOWN.	Algorithmus STOP (analog zu UP_STOP).
EVAL	Evaluierung nach Ablauf der Schutzzeit.	Kein Algorithmus; die Transitionen entscheiden über den nächsten Zustand basierend auf den aktuellen Eingangsdaten.

Transitionen (vereinfacht):

- STOP → UP wenn UP_IN.E1 und UP_IN.D1 = TRUE
- STOP → DOWN wenn DOWN_IN.E1 und DOWN_IN.D1 = TRUE
- UP → UP_STOP wenn UP_IN.E1 und UP_IN.D1 = FALSE
- DOWN → DOWN_STOP wenn DOWN_IN.E1 und DOWN_IN.D1 = FALSE
- UP_STOP → EVAL wenn timeOut.TimeOut
- DOWN_STOP → EVAL wenn timeOut.TimeOut
- EVAL → UP wenn UP_IN.D1 = TRUE und DOWN_IN.D1 = FALSE
- EVAL → DOWN wenn DOWN_IN.D1 = TRUE und UP_IN.D1 = FALSE
- EVAL → STOP wenn beide FALSE oder beide TRUE
- In jedem Zustand kann das Ereignis UPDATE zum selben Zustand zurückführen (dient nur der Aktualisierung von DT_PROTECT).

Anwendungsszenarien

- **Hebebühnensteuerung**: Verriegelung von Heben und Senken; nach Loslassen des Hebebefehls wird eine Totzeit von z. B. 100 ms eingehalten, bevor ein Senken möglich ist.
- **Ventilsteuerung**: Umschaltung zwischen zwei Medien (z. B. Heizen/Kühlen) mit einer Schutzzeit, um Druckstöße zu vermeiden.
- **Roboterachsen**: Vermeidung von Kollisionen durch gleichzeitige Ansteuerung gegenläufiger Bewegungen.
- **Türsteuerung**: Öffnen/Schließen mit Verzögerung beim Richtungswechsel zum Schutz von Endanschlägen.

Vergleich mit ähnlichen Bausteinen

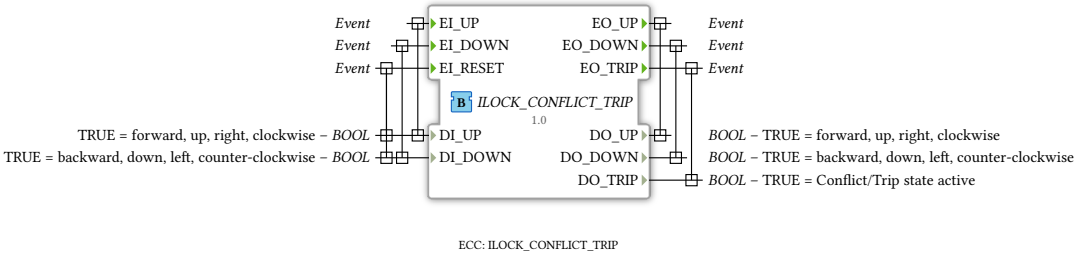
Baustein / Eigenschaft	ILOCK_BLOCK_PROTECT_AX	Einfacher Interlock (ohne Totzeit)	Interlock mit fester Totzeit
Totzeit	Dynamisch konfigurierbar über UPDATE	Keine	Fest (meist im Baustein verdrahtet)
Schnittstelle	Adapter (AX), lose Kopplung	Meist direkte Ein-/Ausgänge	Direkte Ein-/Ausgänge
Priorität	Erster aktiver Eingang	Oft festgelegte Priorität	Unterscheidet sich
Evaluierung nach Totzeit	Ja, mit klarer Zustandsmaschine	Nein	Teilweise
Flexibilität	Hoch (Laufzeitänderung)	Gering	Mittel

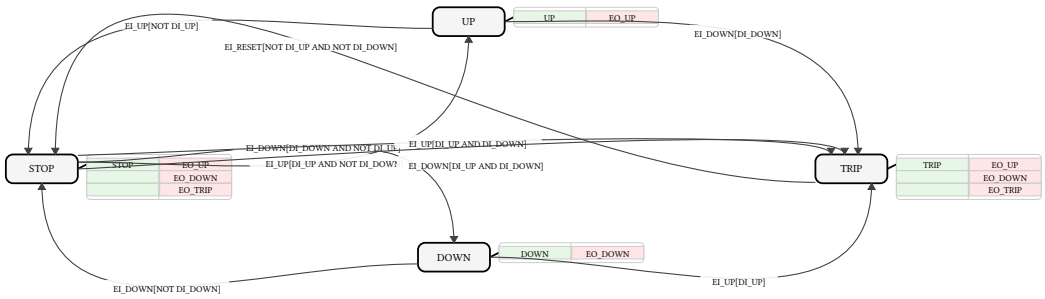
Der `ILOCK_BLOCK_PROTECT_AX` bietet eine besonders flexible Lösung durch die Adapter-Schnittstelle und die dynamisch änderbare Totzeit. Er ist ideal für modulare, wiederverwendbare Steuerungsprojekte nach IEC 61499.

Fazit

Der Funktionsblock `ILOCK_BLOCK_PROTECT_AX` ist ein robuster, zustandsgesteuerter Verriegelungsbaustein für zwei gegenläufige Richtungen. Die Kombination aus Priorisierung des ersten aktiven Eingangs, einer konfigurierbaren Schutz-Totzeit und der vollständigen Adapter-Schnittstelle macht ihn zu einer hervorragenden Wahl für sichere und flexible Steuerungsaufgaben in der Automatisierungstechnik. Die dynamische Anpassung der Totzeit über das `UPDATE`-Ereignis erhöht die Anpassungsfähigkeit an wechselnde Prozessbedingungen. Durch die klare Zustandsmaschine und die Berücksichtigung der IEC 61499-Standards ist der Baustein sowohl in einfachen als auch in komplexen verteilten Systemen einsetzbar.

ILOCK_CONFLICT_TRIP





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ILOCK_CONFLICT_TRIP** dient der **priorisierten Verriegelung** mit **Konflikterkennung**. Er wertet zwei gegensätzliche Binärsignale (z. B. „Vor“ und „Zurück“) aus und gibt nur einen der beiden Befehle aktiv weiter, solange diese nicht gleichzeitig anliegen. Bei gleichzeitiger Aktivität beider Eingänge wird ein **Trip-Zustand** (Fehler/Sperre) ausgelöst, der nur durch einen expliziten Reset (bei inaktiven Eingängen) aufgehoben werden kann. Der Baustein ist speziell für sicherheitskritische Anwendungen ausgelegt, bei denen widersprüchliche Steuerbefehle zuverlässig erkannt werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung
EI_UP	Ereignis zur Verarbeitung einer „Aufwärts“-Anforderung (mit Daten `DI_UP`)
EI_DOWN	Ereignis zur Verarbeitung einer „Abwärts“-Anforderung (mit Daten `DI_DOWN`)
EI_RESET	Ereignis zum Zurücksetzen des Trip-Zustands (liest beide Dateneingänge)

Ereignis-Ausgänge

Ereignis	Beschreibung
EO_UP	Bestätigt die Ausgabe des „Aufwärts“-Befehls (bei aktivem Zustand UP)
EO_DOWN	Bestätigt die Ausgabe des „Abwärts“-Befehls (bei aktivem Zustand DOWN)
EO_TRIP	Zeigt an, dass ein Trip-Zustand vorliegt (bei aktivem Zustand TRIP)

Daten-Eingänge

Name	Typ	Beschreibung
DI_UP	BOOL	TRUE = vorwärts, aufwärts, rechts, im Uhrzeigersinn
DI_DOWN	BOOL	TRUE = rückwärts, abwärts, links, gegen den Uhrzeigersinn

Daten-Ausgänge

Name	Typ	Beschreibung
DO_UP	BOOL	TRUE = Ausgabe „Aufwärts“ aktiv
DO_DOWN	BOOL	TRUE = Ausgabe „Abwärts“ aktiv
DO_TRIP	BOOL	TRUE = Konflikt/Trip-Zustand aktiv

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock besitzt vier Betriebszustände: **STOP**, **UP**, **DOWN** und **TRIP**.

- **STOP (Ruhezustand):** Beide Datenausgänge sind FALSE.
  - Bei `EI_UP` mit `DI_UP = TRUE` und `DI_DOWN = FALSE` wechselt der Baustein in den Zustand **UP**.
  - Bei `EI_DOWN` mit `DI_DOWN = TRUE` und `DI_UP = FALSE` wechselt er in **DOWN**.
  - Bei `EI_UP` oder `EI_DOWN`, wenn beide Dateneingänge TRUE sind, wechselt er direkt in **TRIP** (Konflikt).
- **UP (Aufwärts aktiv):** `DO_UP = TRUE`, `DO_DOWN = FALSE`, `DO_TRIP = FALSE`.
  - Bei einem erneuten `EI_UP`, wenn `DI_UP = FALSE` wird, wechselt er zurück nach **STOP** (Deaktivierung).
  - Bei `EI_DOWN`, wenn `DI_DOWN = TRUE` wird, wechselt er in **TRIP** (während der Fahrt wird eine entgegengesetzte Anforderung erkannt).
- **DOWN (Abwärts aktiv):** `DO_DOWN = TRUE`, `DO_UP = FALSE`, `DO_TRIP = FALSE`.
  - Bei erneutem `EI_DOWN`, wenn `DI_DOWN = FALSE` wird, wechselt er nach **STOP**.
  - Bei `EI_UP`, wenn `DI_UP = TRUE` wird, wechselt er in **TRIP**.
- **TRIP (Fehler/Sperre):** `DO_TRIP = TRUE`, beide Richtungsausgänge FALSE.
  - **Einzige Möglichkeit, den Trip zu verlassen:** Ein `EI_RESET`-Ereignis, bei dem `DI_UP = FALSE` und `DI_DOWN = FALSE` sind. Dann geht es zurück in **STOP**.

Priorisierungsmechanismus: Der zuerst eintreffende gültige Eingang wird bedient, bis er zurückgenommen wird oder ein Konflikt mit dem anderen Eingang auftritt. Gleichzeitige TRUE-Werte auf beiden Dateneingängen führen sofort in den Trip-Zustand.

Technische Besonderheiten

- **Reset nur im Trip erlaubt:** Der Baustein kann nur aus dem TRIP-Zustand heraus durch `EI_RESET` in den STOP-Zustand zurückgesetzt werden. Ein Reset während der Normalzustände (UP/DOWN/STOP) ist wirkungslos.
- **Bedingungen für Trip-Übergänge:**
  - Aus STOP: `(EI_UP UND DI_UP UND DI_DOWN)` ODER `(EI_DOWN UND DI_UP UND DI_DOWN)`
  - Aus UP: `(EI_DOWN UND DI_DOWN)`
  - Aus DOWN: `(EI_UP UND DI_UP)`
- **Quittierung von Ereignissen:** Die Ereignisausgänge werden **immer** gemeinsam mit den Datenausgängen ausgegeben (siehe `With`-Verknüpfungen), sodass der Aufrufer den neuen Zustand sofort abgreifen kann.
- **Verwendung von 4diac (IEC 61499):** Der FB ist als Basic Function Block mit eigenem ECC (Execution Control Chart) realisiert.

Zustandsübersicht

Zustand	DO_UP	DO_DOWN	DO_TRIP	Beschreibung
STOP	FALSE	FALSE	FALSE	Ruhezustand, keine Richtung aktiv
UP	TRUE	FALSE	FALSE	Aufwärts-Richtung aktiv
DOWN	FALSE	TRUE	FALSE	Abwärts-Richtung aktiv
TRIP	FALSE	FALSE	TRUE	Konflikt/Sperre aktiv

Anwendungsszenarien

- **Motorsteuerung für Lineareinheiten oder Drehantriebe:** Verhinderung gleichzeitiger Vor-/Rücklauf-Befehle.
- **Hydraulikventil-Steuerung:** Absicherung gegen widersprüchliche Schaltbefehle (z. B. Heben und Senken gleichzeitig).
- **Sicherheitsverriegelung in der Automatisierung:** Erkennung von Fehlbedienungen und Auslösen eines sicheren Stopps.
- **SPS-Nachbildung in verteilten Systemen:** Als Teil einer Steuerungslogik, die widersprüchliche Zustände vermeiden muss.

Vergleich mit ähnlichen Bausteinen

- **SR-Latch / Flip-Flop:** Ein einfacher SR-Latch speichert einen Zustand, erkennt aber keine Konflikte bei gleichzeitigen „Set“- und „Reset“-Signalen. ILOCK_CONFLICT_TRIP geht in den Trip-Zustand, anstatt einen undefinierten Zustand zu erzeugen.
- **F_TRIG / R_TRIG (Flankenerkennung):** Diese Bausteine erkennen nur Signalflanken, besitzen jedoch keine Verriegelungslogik.
- **Standard Interlock-Bausteine (z. B. aus IEC 61131-3):** Viele bieten eine einfache gegenseitige Verriegelung (z. B. Motorverriegelung), aber keinen dedizierten Trip-Zustand mit Reset-Erfordernis. Der ILOCK_CONFLICT_TRIP ist robuster bei Fehlersituationen.

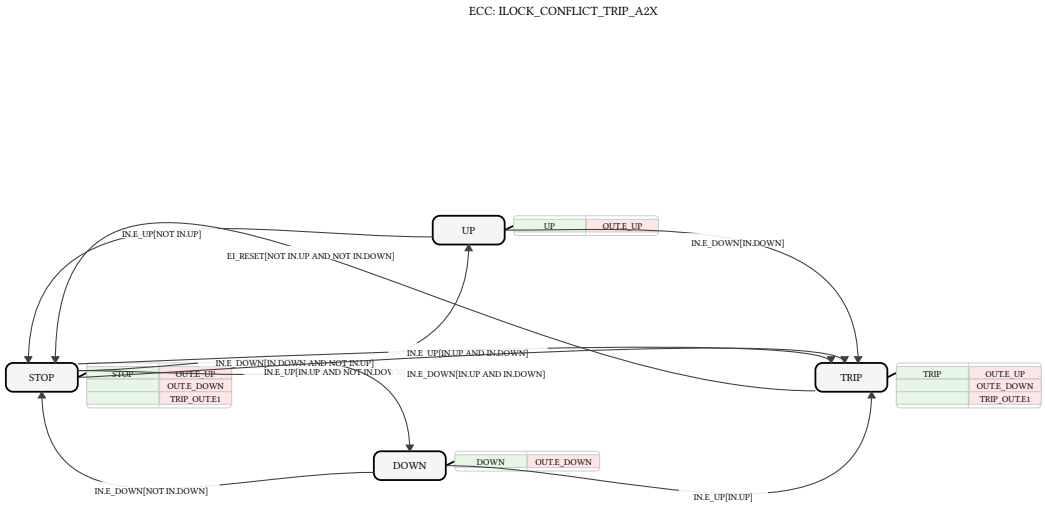
Fazit

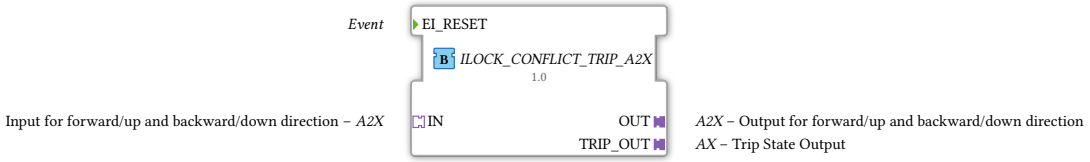
Der **ILOCK_CONFLICT_TRIP** ist ein kompakter, sicherheitsorientierter Funktionsblock für die robuste Verriegelung zweier gegensätzlicher Stellsignale. Er bietet eine klare Priorisierung des ersten aktivierten Eingangs, erkennt Konflikte durch gleichzeitige Aktivität und erzwingt einen expliziten Reset nach einem Fehlerfall. Seine Zustandsmaschine ist einfach nachvollziehbar und eignet sich hervorragend für Anwendungen, in denen widersprüchliche Steuerbefehle zuverlässig abgefangen werden müssen – z. B. in der Maschinen- oder Fahrzeugsteuerung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ILOCK_CONFLICT_TRIP_A2X





• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock **ILOCK_CONFLICT_TRIP_A2X** realisiert eine sicherheitsrelevante Interlock-Logik für bidirektionale Antriebe oder Bewegungsachsen. Er nimmt über einen A2X-Adapter (unidirektional) die Signale **UP** und **DOWN** entgegen und gibt diese über einen weiteren A2X-Adapter an die nachgelagerte Steuerung weiter. Das Besondere ist die Konfliktüberwachung: Werden beide Eingänge gleichzeitig aktiv, wird ein **Trip-Zustand** ausgelöst, der als Alarm über einen separaten AX-Adapter ausgegeben wird. Erst nach einem expliziten **Reset** und dem Deaktivieren beider Eingänge kehrt der Baustein in den Normalbetrieb zurück.

Der Baustein eignet sich für Anwendungen, bei denen eine wechselseitige Verriegelung (z. B. Auf/Ab oder Vor/Zurück) erforderlich ist und ein gleichzeitiges Ansteuern beider Richtungen als gefährlicher Fehlerzustand behandelt werden muss.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung
`EI_RESET`	Event	Setzt den Baustein nach einem Trip zurück, sofern keine der Eingangsgrößen `IN.UP` oder `IN.DOWN` mehr aktiv sind.

Ereignis-Ausgänge

Es sind keine direkten Ereignis-Ausgänge vorhanden. Die Ausgabe erfolgt ausschließlich über die Adapter `OUT` und `TRIP_OUT`, die jeweils Ereignissignale wie `OUT.E_UP`, `OUT.E_DOWN` und `TRIP_OUT.E1` auslösen.

Daten-Eingänge

Direkte Dateneingänge existieren nicht. Alle Eingabedaten werden über den **Adapter `IN`** empfangen:

- `IN.UP` (BOOL): Signal „Aufwärts/Vorwärts“
- `IN.DOWN` (BOOL): Signal „Abwärts/Rückwärts“

Daten-Ausgänge

Die Ausgabedaten werden über die Adapter `OUT` und `TRIP_OUT` bereitgestellt:

- `OUT.UP` (BOOL): Weiterleitung des Zustands „Aufwärts/Vorwärts“
- `OUT.DOWN` (BOOL): Weiterleitung des Zustands „Abwärts/Rückwärts“
- `TRIP_OUT.D1` (BOOL): Signalisiert den Trip-Zustand (TRUE = Fehler/Konflikt erkannt)

Adapter

Name	Typ	Richtung	Beschreibung
`IN`	`adapter::types::unidirectional::A2X`	Socket	Empfängt die Eingangssignale `UP` und `DOWN` sowie die zugehörigen Ereignisse `E_UP` und `E_DOWN`.
`OUT`	`adapter::types::unidirectional::A2X`	Plug	Gibt die verarbeiteten Ausgangssignale `UP` und `DOWN` sowie die Ereignisse `E_UP` und `E_DOWN` an die angeschlossene Logik weiter.
`TRIP_OUT`	`adapter::types::unidirectional::AX`	Plug	Gibt das Trip-Signal `D1` und das Ereignis `E1` bei einem erkannten Konflikt aus.

Funktionsweise

Der Funktionsblock arbeitet als **zustandsbasierter Verriegelungsbaustein**. Er besitzt vier Zustände:

- **STOP**: Beide Richtungssignale sind inaktiv. Es werden `OUT.UP = FALSE`, `OUT.DOWN = FALSE` und `TRIP_OUT.D1 = FALSE` ausgegeben.
- **UP**: Nur `IN.UP` ist aktiv. Der Baustein setzt `OUT.UP = TRUE` und `OUT.DOWN = FALSE`. Der Trip wird zurückgesetzt.
- **DOWN**: Nur `IN.DOWN` ist aktiv. Der Baustein setzt `OUT.DOWN = TRUE` und `OUT.UP = FALSE`. Der Trip wird zurückgesetzt.
- **TRIP**: Beide Eingänge sind gleichzeitig aktiv oder es wird während eines aktiven Zustands der jeweils andere Eingang aktiviert. In diesem Fall wird `TRIP_OUT.D1 = TRUE` gesetzt und beide Ausgänge `OUT.UP` und `OUT.DOWN` werden deaktiviert.

Die Zustandsübergänge werden durch die Ereignisse `IN.E_UP` und `IN.E_DOWN` ausgelöst, wobei die zugehörigen Bedingungen die Werte von `IN.UP` und `IN.DOWN` prüfen. Ein Trip kann aus jedem aktiven Zustand (STOP, UP oder DOWN) heraus ausgelöst werden, sobald beide Eingangssignale gleichzeitig TRUE sind. Der Baustein bleibt solange im TRIP-Zustand, bis ein `EI_RESET`-Ereignis eintritt und beide Eingänge wieder FALSE sind.

Technische Besonderheiten

- **Konfliktpriorisierung**: Der Baustein priorisiert nicht – er erkennt jeden Konflikt und geht in den sicheren Zustand „TRIP“. Ein gleichzeitiges Ansteuern beider Richtungen wird als Fehler behandelt.
- **Reset-Bedingung**: Ein Zurücksetzen aus dem Trip ist nur möglich, wenn keine der beiden Eingangsgrößen (`IN.UP`, `IN.DOWN`) mehr aktiv ist. Dies verhindert ein unbeabsichtigtes Quittieren während eines anhaltenden Fehlers.
- **Ereignisbasierte Verarbeitung**: Die Zustandswechsel erfolgen ausschließlich über die Ereignisse `IN.E_UP` und `IN.E_DOWN`. Die Ausgangsereignisse (`OUT.E_UP`, `OUT.E_DOWN`, `TRIP_OUT.E1`) werden bei jedem Zustandswechsel erzeugt, auch wenn sich die Datenwerte nicht ändern (z. B. beim Übergang von STOP auf TRIP).
- **Adapter-Kapselung**: Die Verwendung von standardisierten A2X/AX-Adaptern ermöglicht eine einfache Integration in bestehende 4diac- oder IEC-61499-Systeme, ohne dass zusätzliche Datentypen definiert werden müssen.
- **Typischer Einsatz**: Geeignet für sicherheitsgerichtete Steuerungen von Industrieantrieben, Verriegelungen von Schiebetüren, Hebezeugen oder anderen Positionieranlagen.

Zustandsübersicht

Zustand	Bedingung für Eintritt	Ausgangssignale	Zustand nach Ereignis
STOP	Initial oder nach Reset, wenn keine Eingänge aktiv.	`OUT.UP = F`, `OUT.DOWN = F`, `TRIP_OUT.D1 = F`	Bei `IN.E_UP` mit `IN.UP` und nicht `IN.DOWN` → UP; bei `IN.E_DOWN` mit `IN.DOWN` und nicht `IN.UP` → DOWN; bei beiden aktiv → TRIP.
UP	Aus STOP, wenn nur `IN.UP` aktiv und `IN.E_UP` eintritt.	`OUT.UP = T`, `OUT.DOWN = F`, `TRIP_OUT.D1 = F`	Bei `IN.E_UP` mit `NOT IN.UP` → STOP; bei `IN.E_DOWN` mit `IN.DOWN` → TRIP.
DOWN	Aus STOP, wenn nur `IN.DOWN` aktiv und `IN.E_DOWN` eintritt.	`OUT.UP = F`, `OUT.DOWN = T`, `TRIP_OUT.D1 = F`	Bei `IN.E_DOWN` mit `NOT IN.DOWN` → STOP; bei `IN.E_UP` mit `IN.UP` → TRIP.
TRIP	Aus jedem Zustand, sobald beide Eingänge aktiv sind (Konflikt).	`OUT.UP = F`, `OUT.DOWN = F`, `TRIP_OUT.D1 = T`	Bei `EI_RESET` mit `NOT IN.UP AND NOT IN.DOWN` → STOP.

Hinweis: F = FALSE, T = TRUE

Anwendungsszenarien

- **Antriebssteuerung mit Verriegelung**: Steuerung eines Gleichstrommotors, der nur entweder vorwärts oder rückwärts laufen darf. Die beiden Richtungssignale kommen von externen Sensoren oder einer SPS. Ein gleichzeitiges Aktivieren beider Signale (z. B. durch defekte Sensorik) führt zu einem sofortigen Stopp und einer Alarmanzeige.
- **Automatische Schiebetüren**: Binäre Signale für „Auf“ und „Zu“ werden überwacht. Ein gleichzeitiges Auftreten wird als Fehler erkannt und der Trip-Ausgang aktiviert, was ein Notstopp-Szenario einleiten kann.
- **Sicherheits-Logik für Hebezeuge**: Verriegelung von Hub- und Senkbewegungen. Bei Konflikt wird der Antrieb abgeschaltet und eine Wartungsmeldung ausgegeben.
- **Integration in 4diac-Systeme**: Dank der standardisierten Adapter kann der Baustein in verteilte Steuerungssysteme eingebunden werden, ohne dass die Signaltypen angepasst werden müssen.

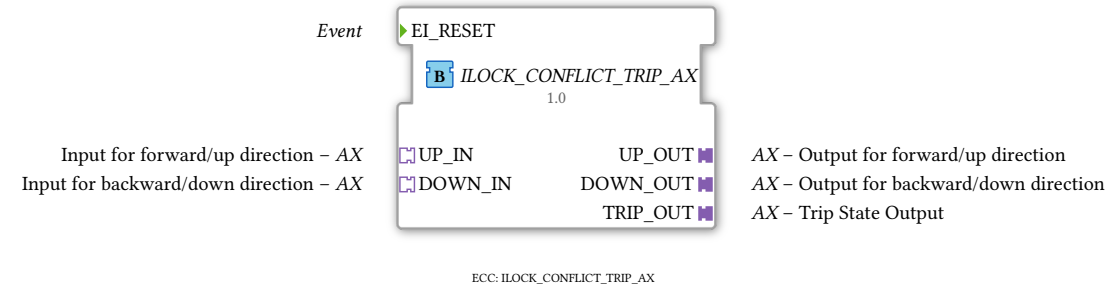
Vergleich mit ähnlichen Bausteinen

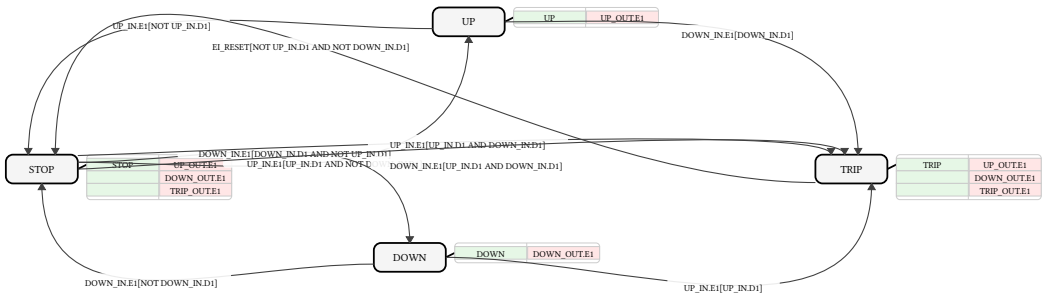
Im Gegensatz zu einfachen Verriegelungsbausteinen, die z. B. nur die Priorität festlegen (z. B. „UP hat Vorrang“), erkennt **ILOCK_CONFLICT_TRIP_A2X** jegliche Konflikte und löst einen festen Trip aus. Dies ist für sicherheitsgerichtete Anwendungen wichtiger als eine Priorisierung, da ein gleichzeitiges Aktivsein beider Richtungen immer ein potenziell gefährliches Ereignis darstellt. Andere Bausteine könnten eine mehrstufige Verriegelung mit zeitlichen Verzögerungen implementieren; dieser Baustein verzichtet bewusst darauf, um eine schnelle und deterministische Reaktion zu gewährleisten. Der Unterschied zu ähnlichen A2X-basierten Bausteinen liegt vor allem in der expliziten Konflikterkennung und der Notwendigkeit eines manuellen Resets.

Fazit

Der **ILOCK_CONFLICT_TRIP_A2X** ist ein robustes und einfach zu integrierendes Verriegelungsmodul für Anwendungen, bei denen zwei sich gegenseitig ausschließende Signale überwacht werden müssen. Seine ereignisbasierte Implementierung ermöglicht eine schnelle Reaktionszeit, während die klare Zustandsmaschine und die standardisierten Adapter eine unkomplizierte Einbindung in IEC-61499-Systeme gewährleisten. Die bewusste Entscheidung, Konflikte immer als Trip zu behandeln, macht ihn zu einer zuverlässigen Komponente für sicherheitsbewusste Steuerungslösungen.

ILOCK_CONFLICT_TRIP_AX





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `ILOCK_CONFLICT_TRIP_AX` realisiert eine Verriegelungslogik für zwei entgegengesetzte Richtungen (UP und DOWN). Er priorisiert das zuerst aktive Eingangssignal und wechselt bei gleichzeitiger Aktivierung beider Eingänge sofort in einen Trip-Zustand. Ein Zurücksetzen aus dem Trip-Zustand ist nur über das Ereignis `EI_RESET` möglich, wenn beide Eingänge inaktiv sind. Die gesamte Kommunikation erfolgt über Adapter vom Typ `unidirectional::AX`, was eine flexible und modulare Einbindung ermöglicht.

Schnittstellenstruktur

Ereignis-Eingänge

- **EI_RESET** – Ereignis zum Rücksetzen des Trip-Zustands (nur wirksam, wenn beide Daten-Eingänge der Adapter FALSE sind).

Ereignis-Ausgänge

Der FB besitzt keine direkten Ereignisausgänge. Zustandsänderungen werden über die Ereignisausgänge der Ausgangs-Adapter (Plugs) signalisiert:

- **UP_OUT.E1**
- **DOWN_OUT.E1**
- **TRIP_OUT.E1**

Daten-Eingänge

Der FB besitzt keine direkten Dateneingänge. Die Eingangsdaten werden über die Adapter-Sockets bereitgestellt:

- **UP_IN.D1** (BOOL) – Aktivierung der Aufwärts-Richtung.
- **DOWN_IN.D1** (BOOL) – Aktivierung der Abwärts-Richtung.

Daten-Ausgänge

Der FB besitzt keine direkten Datenausgänge. Die Ausgangsdaten werden über die Adapter-Plugs bereitgestellt:

- **UP_OUT.D1** (BOOL) – Signal für Aufwärts-Richtung.
- **DOWN_OUT.D1** (BOOL) – Signal für Abwärts-Richtung.
- **TRIP_OUT.D1** (BOOL) – Signal für Trip-Zustand.

Adapter

Sockets (Eingänge)

Adapter	Typ	Beschreibung
`UP_IN`	`adapter::types::unidirectional::AX`	Eingang für Aufwärts-Richtung (Ereignis + Daten)
`DOWN_IN`	`adapter::types::unidirectional::AX`	Eingang für Abwärts-Richtung (Ereignis + Daten)

Plugs (Ausgänge)

Adapter	Typ	Beschreibung
`UP_OUT`	`adapter::types::unidirectional::AX`	Ausgang für Aufwärts-Richtung
`DOWN_OUT`	`adapter::types::unidirectional::AX`	Ausgang für Abwärts-Richtung
`TRIP_OUT`	`adapter::types::unidirectional::AX`	Trip-Zustandsausgang

Hinweis: Die Schnittstelle `unidirectional::AX` wird hier als Adapter mit einem Ereignis (E1) und einem BOOL-Datenelement (D1) interpretiert, welches den Aktivierungszustand transportiert.

Funktionsweise

Der FB arbeitet als endlicher Automat mit vier Zuständen:

1. **STOP** – Ruhezustand. Alle Ausgänge auf FALSE.
   - Bei Ereignis `UP_IN.E1` mit der Bedingung `UP_IN.D1 AND NOT DOWN_IN.D1` ⟶ Übergang nach **UP**.
   - Bei Ereignis `DOWN_IN.E1` mit der Bedingung `DOWN_IN.D1 AND NOT UP_IN.D1` ⟶ Übergang nach **DOWN**.
   - Bei Ereignis an einem der beiden Eingänge mit der Bedingung `UP_IN.D1 AND DOWN_IN.D1` ⟶ Übergang nach **TRIP**.
2. **UP** – Aufwärts-Richtung aktiv. `UP_OUT.D1 = TRUE`, alle anderen FALSE.
   - Bei Ereignis `UP_IN.E1` mit `NOT UP_IN.D1` ⟶ zurück nach **STOP**.
   - Bei Ereignis `DOWN_IN.E1` mit `DOWN_IN.D1` ⟶ **TRIP** (Konflikt).
3. **DOWN** – Abwärts-Richtung aktiv. `DOWN_OUT.D1 = TRUE`, alle anderen FALSE.
   - Bei Ereignis `DOWN_IN.E1` mit `NOT DOWN_IN.D1` ⟶ zurück nach **STOP**.
   - Bei Ereignis `UP_IN.E1` mit `UP_IN.D1` ⟶ **TRIP** (Konflikt).
4. **TRIP** – Fehler-/Konfliktzustand. `TRIP_OUT.D1 = TRUE`, alle anderen FALSE.
   - Einziger Ausgang: Ereignis `EI_RESET`, wenn `NOT UP_IN.D1 AND NOT DOWN_IN.D1` ⟶ zurück nach **STOP**.

Die Priorisierung erfolgt implizit: Solange kein Konflikt vorliegt, wird die zuerst erkannte und gültige Anfrage durchgeführt. Ein Wechsel von UP zu DOWN (oder umgekehrt) ohne Zwischenstopp ist nur über STOP möglich, es sei denn, es tritt ein Konflikt auf, der sofort in den Trip führt.

Technische Besonderheiten

- **Adapter-basierte Kommunikation:** Alle Ein- und Ausgänge sind als Adapter realisiert, was eine lose Kopplung und einfache Wiederverwendung in komplexen Systemen ermöglicht.
- **Trip bei Konflikt:** Eine gleichzeitige Aktivierung beider Richtungen führt sofort in einen gesonderten Trip-Zustand, der nur durch einen expliziten Reset verlassen werden kann.
- **Reset-Bedingung:** Der Reset (`EI_RESET`) ist nur dann wirksam, wenn beide Eingangsdaten FALSE sind – das verhindert ein versehentliches Freigeben bei weiterhin bestehender Konfliktsituation.
- **Keine direkten Events/Daten:** Der FB besitzt keine klassischen Event- oder Daten-Ein-/Ausgänge, sondern nutzt ausschließlich die Adapter-Schnittstellen.

Zustandsübersicht

Zustand	UP_OUT.D1	DOWN_OUT.D1	TRIP_OUT.D1	Beschreibung
`STOP`	FALSE	FALSE	FALSE	Ruheposition, keine Richtung aktiv
`UP`	TRUE	FALSE	FALSE	Aufwärts-Richtung aktiv
`DOWN`	FALSE	TRUE	FALSE	Abwärts-Richtung aktiv
`TRIP`	FALSE	FALSE	TRUE	Konflikt / Trip, erfordert Reset

Wichtige Übergänge:

- STOP → UP: wenn UP_IN aktiv und DOWN_IN inaktiv.
- STOP → DOWN: wenn DOWN_IN aktiv und UP_IN inaktiv.
- STOP → TRIP: wenn beide Eingänge gleichzeitig aktiv (Konflikt).
- UP → STOP: wenn UP_IN deaktiviert wird.
- UP → TRIP: wenn DOWN_IN während aktivem UP kommt.
- DOWN → STOP: wenn DOWN_IN deaktiviert wird.
- DOWN → TRIP: wenn UP_IN während aktivem DOWN kommt.
- TRIP → STOP: nur durch EI_RESET, wenn beide Eingänge inaktiv sind.

Anwendungsszenarien

- **Verriegelungssteuerung an Maschinen:** Verhindert gleichzeitiges Fahren in zwei entgegengesetzte Richtungen (z. B. Heben/Senken, Vorwärts/Rückwärts).
- **Sicherheitsgerichtete Anwendungen:** Erkennt kritische Konfliktsituationen und erzwingt einen definierten Fehlerzustand, der quittiert werden muss.
- **Adapter-basierte Systeme:** Einbindung in modulare Automatisierungslösungen, bei denen Bausteine über standardisierte Adapter kommunizieren.

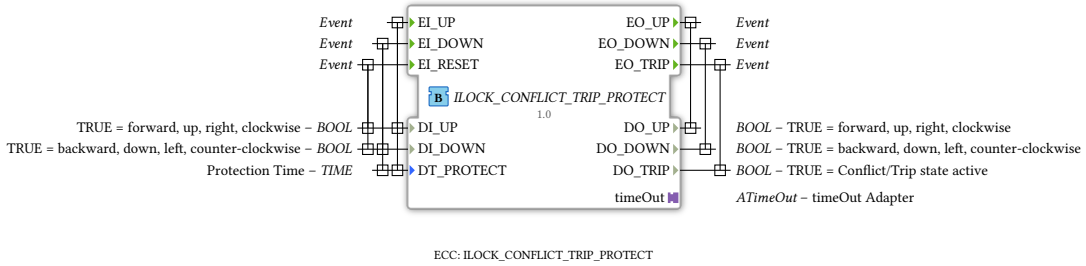
Vergleich mit ähnlichen Bausteinen

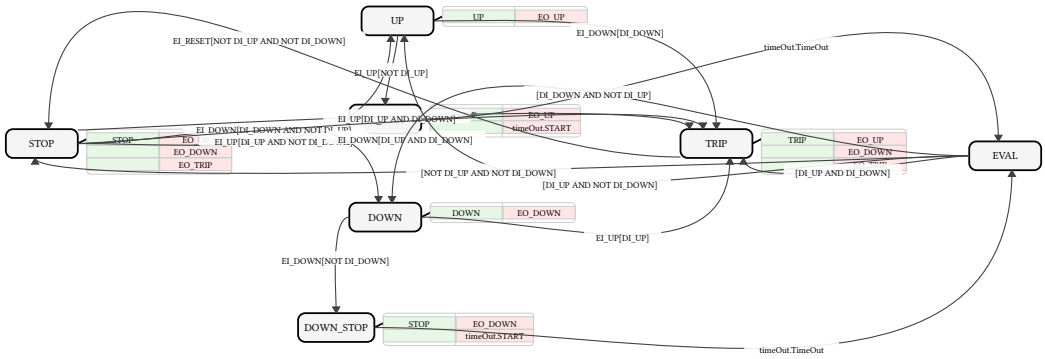
- Einfache Interlock-Bausteine (z. B. `ILOCK`) wechseln oft direkt zwischen den Richtungen, ohne einen Trip-Zustand zu implementieren.
- Bausteine mit Prioritätslogik bevorzugen eine Richtung fest, während dieser FB die zuerst eintreffende Anfrage priorisiert und bei Konflikt einen Fehler meldet.
- `ILOCK_CONFLICT_TRIP_AX` zeichnet sich durch den expliziten Trip-Zustand und die Adapter-Schnittstelle aus – dies ermöglicht eine saubere Trennung von Steuerungs- und Diagnosefunktionen.

Fazit

Der `ILOCK_CONFLICT_TRIP_AX` ist ein robuster Verriegelungsbaustein, der speziell für Konfliktsituationen ausgelegt ist. Durch den Trip-Zustand mit externem Reset wird eine sichere Handhabung von Parallelanforderungen gewährleistet. Die Adapter-basierte Struktur erleichtert die Integration in größere Automatisierungsprojekte und fördert die Wiederverwendbarkeit. Für sicherheitskritische oder fehlertolerante Systeme stellt dieser FB eine zuverlässige und flexible Lösung dar.

ILOCK_CONFLICT_TRIP_PROTECT





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `ILOCK_CONFLICT_TRIP_PROTECT` erweitert `ILOCK_CONFLICT_TRIP` um eine konfigurierbare Schutz-Totzeit (`DT_PROTECT`), analog zu der Art, wie `ILOCK_BLOCK_PROTECT` das Verhalten von `ILOCK_BLOCK` ergänzt. Er priorisiert weiterhin den zuerst aktiven Eingang, löst bei gleichzeitiger Aktivierung beider Richtungen einen Trip-Zustand aus und benötigt danach ein explizites `EI_RESET`. Neu ist, dass nach Freigabe des aktiven Eingangs zunächst die Totzeit `DT_PROTECT` verstreichen muss, bevor der Baustein die aktuellen Eingangssignale neu bewertet – erst danach wird die nächste Richtung (oder gegebenenfalls ein Trip) übernommen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Mitgeführte Daten	Beschreibung
`EI_UP`	`DI_UP`, `DT_PROTECT`	Ereignis für die Aufwärts-/Vorwärts-Richtung.
`EI_DOWN`	`DI_DOWN`, `DT_PROTECT`	Ereignis für die Abwärts-/Rückwärts-Richtung.
`EI_RESET`	`DI_UP`, `DI_DOWN`	Rücksetzen des Trip-Zustands; nur wirksam, wenn beide Daten-Eingänge FALSE sind.

Ereignis-Ausgänge

Name	Mitgeführte Daten	Beschreibung

EO_UP	DO_UP	Wird ausgelöst, wenn die UP-Richtung aktiv oder deaktiviert wird.
EO_DOWN	DO_DOWN	Wird ausgelöst, wenn die DOWN-Richtung aktiv oder deaktiviert wird.
EO_TRIP	DO_TRIP	Wird im TRIP-Zustand mitgesendet und signalisiert den Trip-Status.

Daten-Eingänge

- DI_UP (BOOL) – TRUE = vorwärts/aufwärts/rechts/im Uhrzeigersinn.
- DI_DOWN (BOOL) – TRUE = rückwärts/abwärts/links/gegen den Uhrzeigersinn.
- DT_PROTECT (TIME, Initialwert T#50ms) – Schutz-Totzeit, die nach der Freigabe des aktiven Eingangs verstreichen muss, bevor eine erneute Bewertung erfolgt.

Daten-Ausgänge

- DO_UP (BOOL) – Signalisiert die aktive UP-Richtung.
- DO_DOWN (BOOL) – Signalisiert die aktive DOWN-Richtung.
- DO_TRIP (BOOL) – Signalisiert den Trip-/Konfliktzustand.

Adapter

Adapter	Typ	Richtung	Beschreibung
timeOut	iec61499::events::ATimeOut	Plug	Timer-Adapter für die Schutzzeit. Der Baustein setzt timeOut.DT und startet ihn über timeOut.START; timeOut.TimeOut signalisiert den Ablauf.

Funktionsweise

Der Baustein arbeitet als endlicher Automat (ECC) mit sieben Zuständen:

1. **STOP** – Ruhezustand, alle Ausgänge FALSE. Bei EI_UP mit DI_UP AND NOT DI_DOWN → **UP** bei EI_DOWN mit DI_DOWN AND NOT DI_UP → **DOWN** sind bei EI_UP oder EI_DOWN beide Datensignale TRUE → sofort **TRIP**.
2. **UP** – DO_UP = TRUE. Bei EI_UP[NOT DI_UP] → **UP_STOP** bei EI_DOWN[DI_DOWN] (Konflikt während aktivem UP) → sofort **TRIP**.
3. **DOWN** – DO_DOWN = TRUE. Bei EI_DOWN[NOT DI_DOWN] → **DOWN_STOP** bei EI_UP[DI_UP] (Konflikt während aktivem DOWN) → sofort **TRIP**.
4. **UP_STOP** / **DOWN_STOP** – Übergangszustände nach Freigabe des aktiven Eingangs. Der Algorithmus STOP setzt alle Ausgänge auf FALSE, überträgt DT_PROTECT an timeOut.DT und startet den Timer (timeOut.START). Bei Ablauf (timeOut.TimeOut) → **EVAL**.
5. **EVAL** – Kein eigener Algorithmus. Die aktuellen Werte von DI_UP/DI_DOWN werden neu bewertet: nur DI_UP TRUE → **UP** nur DI_DOWN TRUE → **DOWN** beide FALSE → **STOP** beide TRUE → **TRIP**.
6. **TRIP** – DO_TRIP = TRUE, alle anderen Ausgänge FALSE. Verlassen nur über EI_RESET, wenn NOT DI_UP AND NOT DI_DOWN → **STOP**.

Die Trip-Erkennung selbst erfolgt weiterhin sofort und unabhängig von der Totzeit: Ein Konflikt während STOP, UP oder DOWN löst immer direkt TRIP aus. Die Totzeit DT_PROTECT wirkt ausschließlich zwischen der Freigabe eines aktiven Eingangs und der Übernahme der nächsten Richtung.

Technische Besonderheiten

- **Kombination zweier Muster:** Der Baustein verbindet die Trip-bei-Konflikt-Logik von ILOCK_CONFLICT_TRIP mit der Totzeit-Logik von ILOCK_BLOCK_PROTECT.
- **Sofortiger Trip, verzögerte Freigabe:** Konflikte werden ohne Verzögerung erkannt; nur die Rückkehr in einen neuen gültigen Zustand nach Freigabe wird durch DT_PROTECT verzögert.
- **Reset-Bedingung:** EI_RESET wirkt nur, wenn beide Dateneingänge inaktiv sind – ein Reset bei fortbestehendem Konflikt bleibt wirkungslos.
- **Timer-Neustart in jedem Stop-Zwischenzustand:** timeOut.DT wird bei jedem Eintritt in UP_STOP/DOWN_STOP neu aus DT_PROTECT übernommen, sodass eine Änderung des Parameters unmittelbar in der nächsten Verzögerung wirksam wird.

Zustandsübersicht

Zustand	DO_UP	DO_DOWN	DO_TRIP	Beschreibung
`STOP`	FALSE	FALSE	FALSE	Ruhezustand, keine Richtung aktiv.
`UP`	TRUE	FALSE	FALSE	Aufwärts-Richtung aktiv.
`DOWN`	FALSE	TRUE	FALSE	Abwärts-Richtung aktiv.
`UP_STOP`	FALSE	FALSE	FALSE	Wartet auf Ablauf von `DT_PROTECT` nach Freigabe von UP.
`DOWN_STOP`	FALSE	FALSE	FALSE	Wartet auf Ablauf von `DT_PROTECT` nach Freigabe von DOWN.
`EVAL`	–	–	–	Kein Algorithmus; entscheidet anhand aktueller Eingänge über den Folgezustand.
`TRIP`	FALSE	FALSE	TRUE	Konflikt/Trip, erfordert `EI_RESET`.

Anwendungsszenarien

- **Antriebe mit Nachlauf:** Wenn nach dem Loslassen eines Fahrbefehls eine mechanische oder hydraulische Nachlaufzeit eingehalten werden muss, bevor sicher in die Gegenrichtung geschaltet werden darf.
- **Sicherheitsgerichtete Verriegelung mit Quittierpflicht:** Anwendungen, in denen ein gleichzeitiger Befehl in beide Richtungen als Fehler gilt und explizit quittiert werden muss.
- **Ventil- oder Klappensteuerungen:** Schutz vor Druckstößen oder mechanischer Überlastung durch eine Mindestpause zwischen Richtungswechseln.

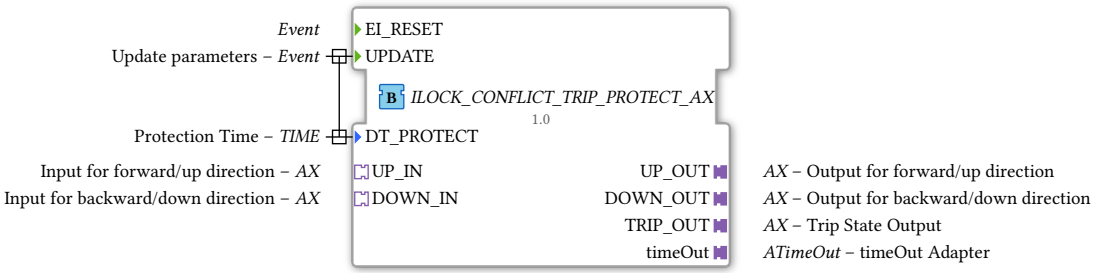
Vergleich mit ähnlichen Bausteinen

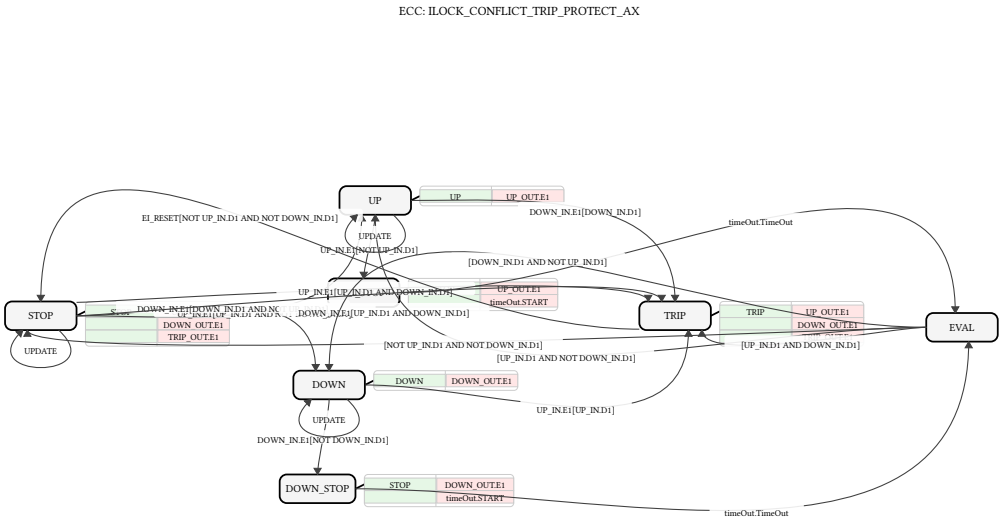
Gegenüber `ILOCK_CONFLICT_TRIP` fügt dieser Baustein die Zustände `UP_STOP`, `DOWN_STOP` und `EVAL` sowie den `timeOut`-Adapter hinzu – die Trip-Logik selbst bleibt identisch. Gegenüber `ILOCK_BLOCK_PROTECT` unterscheidet er sich dadurch, dass ein gleichzeitiger Befehl in beide Richtungen nicht ignoriert, sondern als expliziter Fehlerzustand (`TRIP`) behandelt wird, der ein `EI_RESET` erfordert.

Fazit

`ILOCK_CONFLICT_TRIP_PROTECT` verbindet die klare Fehlererkennung von `ILOCK_CONFLICT_TRIP` mit der Schutz-Totzeit von `ILOCK_BLOCK_PROTECT`. Er eignet sich für Anwendungen, die sowohl eine harte Konflikterkennung mit Quittierpflicht als auch eine Mindestpause zwischen Richtungswechseln benötigen.

ILOCK_CONFLICT_TRIP_PROTECT_AX





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `ILOCK_CONFLICT_TRIP_PROTECT_AX` ist die Adapter-Version von `ILOCK_CONFLICT_TRIP_PROTECT`: Er kombiniert die Trip-bei-Konflikt-Logik von `ILOCK_CONFLICT_TRIP_AX` mit der Schutz-Totzeit von `ILOCK_BLOCK_PROTECT_AX`. Der zuerst aktive Eingang wird priorisiert; eine gleichzeitige Aktivierung beider Richtungen löst sofort einen Trip aus, der nur über `EI_RESET` zurückgesetzt werden kann. Nach Freigabe des aktiven Eingangs wartet der Baustein zusätzlich die konfigurierbare Zeit `DT_PROTECT` ab, bevor er die Eingänge neu bewertet. Die Prozessdaten (`UP_IN`/`DOWN_IN`/`UP_OUT`/`DOWN_OUT`/`TRIP_OUT`) laufen über Adapter vom Typ `unidirectional::AX`; der Schutzzeit-Timer verwendet zusätzlich den Adapter `iec61499::events::ATimeOut` (`timeOut`).

Schnittstellenstruktur

Ereignis-Eingänge

Name	Mitgeführte Daten	Beschreibung
`EI_RESET`	–	Rücksetzen des Trip-Zustands; nur wirksam, wenn `UP_IN.D1` und `DOWN_IN.D1` beide FALSE sind.
`UPDATE`	`DT_PROTECT`	Aktualisiert die Schutzzeit `DT_PROTECT` zur Laufzeit, ohne den aktuellen Zustand zu verlassen.

Ereignis-Ausgänge

Keine direkten Ereignis-Ausgänge. Zustandsänderungen werden über die Ereignisse der Ausgangsadapter (Plugs) signalisiert:

- `UP_OUT.E1`, `DOWN_OUT.E1`, `TRIP_OUT.E1`

Daten-Eingänge

`UP_IN`/`DOWN_IN` sind keine direkten Dateneingänge, sondern werden über die Socket-Adapter bereitgestellt; `DT_PROTECT` dagegen ist ein direkter Dateneingang, mitgeführt vom Ereignis `UPDATE`:

- `UP_IN.D1` (BOOL) – Aktivierung der Aufwärts-Richtung (Socket-Adapter).
- `DOWN_IN.D1` (BOOL) – Aktivierung der Abwärts-Richtung (Socket-Adapter).

- `DT_PROTECT` (TIME, Initialwert `T#50ms`) – Schutz-Totzeit nach Freigabe des aktiven Eingangs (direkter Dateneingang, mitgeführt von `UPDATE`).

Daten-Ausgänge

Keine direkten Datenausgänge. Werden über die Plug-Adapter bereitgestellt:

- `UP_OUT.D1` (BOOL) – Signal für Aufwärts-Richtung.
- `DOWN_OUT.D1` (BOOL) – Signal für Abwärts-Richtung.
- `TRIP_OUT.D1` (BOOL) – Signal für den Trip-Zustand.

Adapter

Sockets (Eingänge)

Adapter	Typ	Beschreibung
`UP_IN`	`adapter::types::unidirectional::AX`	Eingang für die Aufwärts-Richtung.
`DOWN_IN`	`adapter::types::unidirectional::AX`	Eingang für die Abwärts-Richtung.

Plugs (Ausgänge)

Adapter	Typ	Beschreibung
`UP_OUT`	`adapter::types::unidirectional::AX`	Ausgang für die Aufwärts-Richtung.
`DOWN_OUT`	`adapter::types::unidirectional::AX`	Ausgang für die Abwärts-Richtung.
`TRIP_OUT`	`adapter::types::unidirectional::AX`	Trip-Zustandsausgang.
`timeOut`	`iec61499::events::ATimeOut`	Timer-Adapter für die Schutzzeit; der Baustein setzt `timeOut.DT` und startet ihn über `timeOut.START`.

Funktionsweise

Der Automat entspricht strukturell `ILOCK_CONFLICT_TRIP_PROTECT`, nur dass alle Signale über Adapter geführt werden:

1. **STOP** – Ruhezustand. Bei `UP_IN.E1[UP_IN.D1 AND NOT DOWN_IN.D1]` → **UP** bei `DOWN_IN.E1[DOWN_IN.D1 AND NOT UP_IN.D1]` → **DOWN** sind bei einem der beiden Ereignisse beide Datensignale TRUE → sofort **TRIP**.
2. **UP** – `UP_OUT.D1 = TRUE`. Bei `UP_IN.E1[NOT UP_IN.D1]` → **UP_STOP** bei `DOWN_IN.E1[DOWN_IN.D1]` (Konflikt) → sofort **TRIP**.
3. **DOWN** – `DOWN_OUT.D1 = TRUE`. Bei `DOWN_IN.E1[NOT DOWN_IN.D1]` → **DOWN_STOP** bei `UP_IN.E1[UP_IN.D1]` (Konflikt) → sofort **TRIP**.
4. **UP_STOP** / **DOWN_STOP** – Der Algorithmus `STOP` setzt alle Adapterausgänge auf FALSE, überträgt `DT_PROTECT` an `timeOut.DT` und startet den Timer. Bei `timeOut.TimeOut` → **EVAL**.
5. **EVAL** – Kein eigener Algorithmus. Neubewertung: nur `UP_IN.D1` TRUE → **UP** nur `DOWN_IN.D1` TRUE → **DOWN** beide FALSE → **STOP** beide TRUE → **TRIP**.
6. **TRIP** – `TRIP_OUT.D1 = TRUE`. Verlassen nur über `EI_RESET[NOT UP_IN.D1 AND NOT DOWN_IN.D1]` → **STOP**.

Zusätzlich existiert in den Zuständen `STOP`, `UP` und `DOWN` je eine Selbstschleifen-Transition auf das Ereignis `UPDATE`, die den jeweiligen Zustand nicht verlässt und ausschließlich der Übernahme eines neuen `DT_PROTECT`-Werts dient.

Technische Besonderheiten

- **Reine Adapter-Schnittstelle:** Alle Prozessdaten werden über `unidirectional::AX`-Adapter geführt (der Schutzzeit-Timer zusätzlich über einen `iec61499::events::ATimeOut`-Adapter), es gibt keine klassischen Event-/Datenports außer `EI_RESET`, `UPDATE` und `DT_PROTECT`.
- **Dynamische Totzeit:** Das Ereignis `UPDATE` erlaubt das Ändern von `DT_PROTECT` zur Laufzeit, ohne den Automaten zu verlassen – wirksam wird der neue Wert beim nächsten Eintritt in `UP_STOP`/`DOWN_STOP`.
- **Sofortiger Trip bei Konflikt:** Wie bei der Nicht-Adapter-Variante wird ein gleichzeitiger Befehl in beide Richtungen ohne Verzögerung erkannt.
- **Reset-Bedingung:** `EI_RESET` wirkt nur, wenn beide Eingangsadapter `D1 = FALSE` melden.

Zustandsübersicht

Zustand	UP_OUT.D1	DOWN_OUT.D1	TRIP_OUT.D1	Beschreibung
`STOP`	FALSE	FALSE	FALSE	Ruhezustand, keine Richtung aktiv.
`UP`	TRUE	FALSE	FALSE	Aufwärts-Richtung aktiv.
`DOWN`	FALSE	TRUE	FALSE	Abwärts-Richtung aktiv.
`UP_STOP`	FALSE	FALSE	FALSE	Wartet auf Ablauf von `DT_PROTECT` nach Freigabe von UP.
`DOWN_STOP`	FALSE	FALSE	FALSE	Wartet auf Ablauf von `DT_PROTECT` nach Freigabe von DOWN.
`EVAL`	–	–	–	Kein Algorithmus; entscheidet über den Folgezustand.
`TRIP`	FALSE	FALSE	TRUE	Konflikt/Trip, erfordert `EI_RESET`.

Anwendungsszenarien

- **Adapter-basierte Antriebssteuerungen:** Modulare Anlagen, in denen Richtungssignale bereits als `unidirectional::AX`-Adapter durch das System geführt werden.
- **Sicherheitsgerichtete Verriegelung mit Nachlaufzeit:** Anwendungen, die sowohl eine harte Konflikterkennung mit Quittierpflicht als auch eine Mindestpause zwischen Richtungswechseln benötigen.
- **Laufzeit-Parametrierung:** Systeme, in denen die Schutzzeit je nach Betriebszustand (z. B. Temperatur, Last) über `UPDATE` angepasst werden muss.

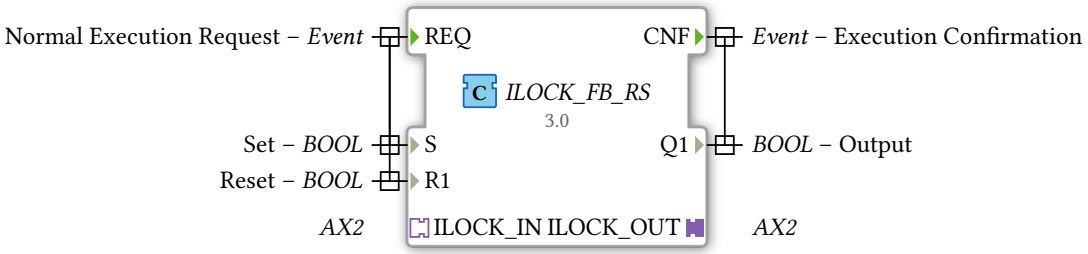
Vergleich mit ähnlichen Bausteinen

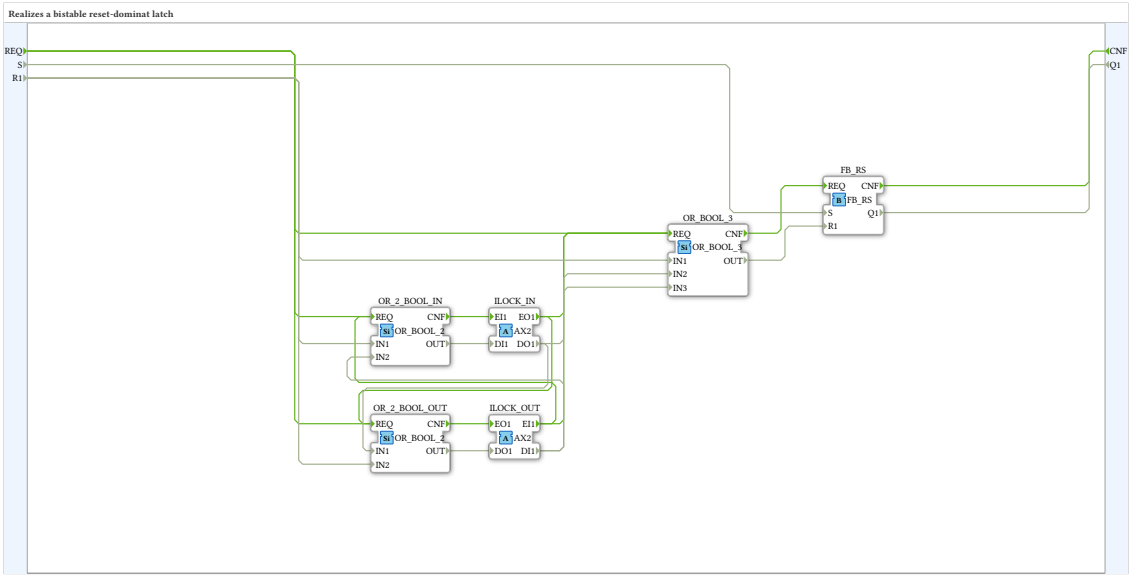
Gegenüber `ILOCK_CONFLICT_TRIP_AX` ergänzt dieser Baustein die Zustände `UP_STOP`, `DOWN_STOP`, `EVAL` sowie den `timeOut`-Adapter und das `UPDATE`-Ereignis. Gegenüber `ILOCK_BLOCK_PROTECT_AX` unterscheidet er sich dadurch, dass ein gleichzeitiger Befehl in beide Richtungen nicht stillschweigend ignoriert, sondern als expliziter `TRIP`-Zustand behandelt wird, der ein `EI_RESET` erfordert. Gegenüber der Nicht-Adapter-Variante `ILOCK_CONFLICT_TRIP_PROTECT` ist die Schnittstelle vollständig adapterbasiert, was die Einbindung in modulare, adapterorientierte Systeme erleichtert.

Fazit

`ILOCK_CONFLICT_TRIP_PROTECT_AX` vereint die Adapter-basierte Konflikterkennung mit Quittierpflicht und einer konfigurierbaren Schutz-Totzeit in einem Baustein. Er eignet sich für modulare Automatisierungslösungen, die eine strikte gegenseitige Ausschließlichkeit zweier Richtungen mit Nachlaufzeit und Laufzeit-Parametrierbarkeit kombinieren müssen.

ILOCK_FB_RS





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `ILOCK_FB_RS` realisiert ein bistabiles Reset-dominantes RS-Latch (Reset-dominant Flipflop). Er erweitert ein einfaches RS-Flipflop um eine bidirektionale Kopplungsschnittstelle, die eine Verkettung mehrerer solcher Bausteine zu einer Interlock-Kette ermöglicht. Der Ausgangszustand wird durch den Set-Eingang (S) gesetzt und durch den Reset-Eingang (R1) – auch unter Einbeziehung von Signalen benachbarter Bausteine – zurückgesetzt.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Kommentar
`REQ`	Normaler Ausführungsanforderung (mit den Daten S und R1 verknüpft)

Ereignis-Ausgänge

Ereignis	Kommentar
`CNF`	Bestätigung der Ausführung (mit dem Datenausgang Q1 verknüpft)

Daten-Eingänge

Variable	Typ	Kommentar
`S`	BOOL	Setzen (aktiver High)
`R1`	BOOL	Rücksetzen (aktiver High) – Reset-dominant

Daten-Ausgänge

Variable	Typ	Kommentar
`Q1`	BOOL	Ausgang des Flipflops

Adapter

Bezeichnung	Richtung	Typ	Beschreibung

ILOCK_IN	Socket	bidirektional::AX2	Empfängt Signale vom vorherigen Baustein der Kette (Ereignis EO1, Daten DO1)
ILOCK_OUT	Plug	bidirektional::AX2	Sendet Signale an den nächsten Baustein der Kette (Ereignis EI1, Daten DI1)

Die Adapter vom Typ AX2 stellen jeweils zwei logische Kanäle bereit:

- Ereigniskanal: EI1 (Eingang) / EO1 (Ausgang)
- Datenkanal: DI1 (Eingang) / DO1 (Ausgang)

Funktionsweise

Der Baustein besteht intern aus einem RS-Flipflop (FB_RS, Reset-dominant), zwei OR-Logiken und einem OR-3-Gatter. Die Logik verknüpft die externen Eingänge mit den Adaptersignalen:

- **Setzen:** Der Eingang S geht direkt an den Set-Eingang des internen Flipflops. Bei aktivem S und inaktivem Reset wird Q1 = TRUE.
- **Rücksetzen:** Der Reset-Eingang R1 wird mit den Daten-Ausgängen der beiden Adapter (ILOCK_IN.DO1 und ILOCK_OUT.DI1) über ein OR-3-Gatter verknüpft. Das Ergebnis steuert den Reset-Eingang des internen Flipflops. Sobald **eine** dieser drei Quellen aktiv wird, wird der Ausgang Q1 zurückgesetzt (Reset-Dominanz).
- **Kopplung:** Der eigene Reset (R1) wird zusätzlich über OR-Gatter auf die Adapterausgänge gelegt:
  - An ILOCK_OUT.DO1 wird R1 mit dem empfangenen Signal von ILOCK_IN.DO1 verodert (OR_2_BOOL_OUT), sodass ein Reset weitergegeben wird.
  - An ILOCK_IN.DI1 wird R1 mit dem empfangenen Signal von ILOCK_OUT.DI1 verodert (OR_2_BOOL_IN), sodass ein Reset von der anderen Seite ebenfalls weitergereicht wird.
- **Ereignissteuerung:** Alle Ereignisse (REQ, sowie die Adapterereignisse) werden gesammelt und über OR-Logik an das interne Flipflop weitergeleitet. Die Bestätigung (CNF) wird nach Verarbeitung des Flipflops ausgegeben.

Technische Besonderheiten

- **Reset-Dominanz:** Im Konfliktfall (S und R1 gleichzeitig aktiv) hat der Reset Vorrang ⟶ Q1 = FALSE.
- **Kaskadierung:** Mehrere ILOCK_FB_RS können über die Adapter hintereinandergeschaltet werden. Ein Reset, der an einer Stelle auftritt, pflanzt sich über die Kette fort.
- **Bidirektionale Kopplung:** Die Adapter ermöglichen sowohl Vorwärts- als auch Rückwärtskommunikation, sodass der Baustein sowohl Setz- als auch Rücksetzsignale von beiden Seiten empfangen und weiterleiten kann.
- **Kompatibilität:** Implementiert gemäß IEC 61499-1 und nutzt Subapplikationen aus der iec61131-Bibliothek.

Zustandsübersicht

Das interne RS-Flipflop hat folgende Zustandstabelle (Reset-dominant):

S	R1_eff	Q1 (neu)	Beschreibung
0	0	Q1_alt	Halten
1	0	1	Setzen
0	1	0	Rücksetzen
1	1	0	Reset dominant

Dabei ist R1_eff das Ergebnis der Oder-Verknüpfung aus R1, ILOCK_IN.DO1 und ILOCK_OUT.DI1. Der Ausgang Q1 wird nur dann gesetzt, wenn keiner dieser drei Reset-Quellen aktiv ist.

Anwendungsszenarien

- **Interlock-Ketten** in Sicherheitssteuerungen: Mehrere Bausteine werden aneinandergereiht. Ein Fehler oder Reset an einer Stelle setzt alle nachfolgenden Ausgänge zurück.
- **Prioritätslogik:** Reset-dominante Latch-Strukturen, bei denen ein bestimmtes Ereignis (z. B. Not-Aus) Vorrang vor Setzsignalen hat.
- **Signalverriegelung** in redundanten Systemen, bei denen mehrere Bedingungen für Setzen und Zurücksetzen zusammengeführt werden müssen.

Vergleich mit ähnlichen Bausteinen

Baustein	Eigenschaften
`FB_RS` (IEC 61131)	Einfaches Reset-dominantes RS-Flipflop ohne Adapter, keine Kaskadierung
`ILOCK_FB_RS`	Gleiche Kernfunktion, aber mit bidirektionalen Adaptern für Kettenschaltung und zusätzlicher Verknüpfung von Reset-Signalen

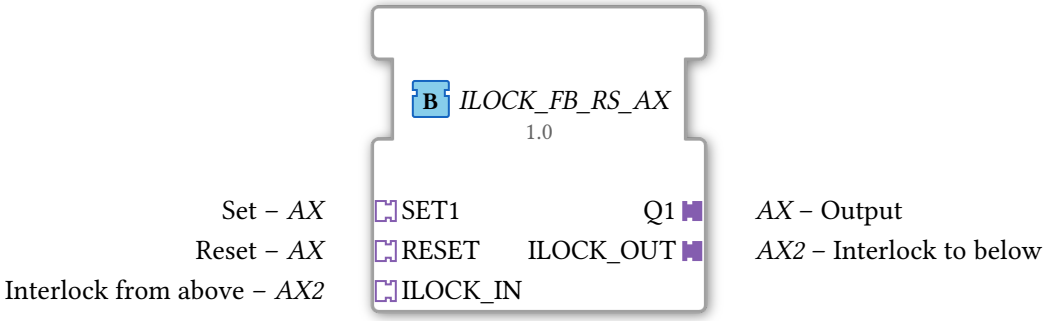
Der `ILOCK_FB_RS` fügt also die Möglichkeit hinzu, mehrere solche Flipflops in einer Kette zu betreiben, wobei ein Reset von einem Glied auf die anderen übertragen wird.

Fazit

Der `ILOCK_FB_RS` ist ein vielseitiger Baustein zur Realisierung von Reset-dominanten Interlock-Logiken. Durch die integrierte bidirektionale Kopplung eignet er sich besonders für modulare, erweiterbare Sicherheits- und Verriegelungsschaltungen, ohne dass zusätzliche externe Logik erforderlich ist.

ILOCK_FB_RS_AX

Einleitung



Der Funktionsbaustein **ILOCK_FB_RS_AX** realisiert ein bistabiles, reset-dominantes RS-Latch mit einer AX-Schnittstelle für Set- und Reset-Signale sowie einer bidirektionalen AX2-Schnittstelle zur Verkettung mehrerer Interlock-Bausteine in einer Hierarchie. Der Baustein ist für den Einsatz in sicherheitsgerichteten und verriegelungsbasierten Steuerungsumgebungen konzipiert.

Schnittstellenstruktur

Der Baustein verfügt über keine klassischen separaten Event- oder Datenports, sondern kapselt sämtliche Signalarten über **Adapter**. Die folgende Auflistung gruppiert die Ereignisse und Daten, die über die einzelnen Adapter anliegen.

Ereignis-Eingänge

- **SET1.E1** (über Socket-Adapter `SET1`): Set-Event, triggert die Latch-Funktion.
- **RESET.E1** (über Socket-Adapter `RESET`): Reset-Event, triggert die rücksetzende Logik.
- **ILOCK_IN.EO1** (über Socket-Adapter `ILOCK_IN`): Ereignis von einer übergeordneten oder benachbarten Interlock-Instanz (bidirektionale Seite).
- **ILOCK_OUT.EI1** (über Plug-Adapter `ILOCK_OUT`): Ereignis von einer untergeordneten Interlock-Instanz.

Ereignis-Ausgänge

- **Q1.E1** (über Plug-Adapter `Q1`): Ausgabe-Ereignis nach jeder Aktualisierung des Latch-Zustands.
- **ILOCK_IN.EI1** (über Socket-Adapter `ILOCK_IN`): Ereignis, das an den verbundenen Partneradapter des `ILOCK_IN` gesendet wird.
- **ILOCK_OUT.EO1** (über Plug-Adapter `ILOCK_OUT`): Ereignis, das an den verbundenen Partneradapter des `ILOCK_OUT` gesendet wird.

Daten-Eingänge

- **SET1.D1** (über Socket-Adapter `SET1`): Set-Signal (Boolescher Wert, TRUE setzt den Latch).
- **RESET.D1** (über Socket-Adapter `RESET`): Reset-Signal (Boolescher Wert, TRUE setzt den Latch zurück, dominiert über Set).
- **ILOCK_IN.DO1** (über Socket-Adapter `ILOCK_IN`): Interlock-Status von oben (Boolescher Wert, TRUE blockiert den Latch).
- **ILOCK_OUT.DI1** (über Plug-Adapter `ILOCK_OUT`): Interlock-Status von unten (Boolescher Wert, TRUE blockiert den Latch).

Daten-Ausgänge

- **Q1.D1** (über Plug-Adapter `Q1`): Ausgangszustand des Latches (Boolescher Wert).
- **ILOCK_IN.DI1** (über Socket-Adapter `ILOCK_IN`): Weiterleitung des Reset-Signals nach oben.
- **ILOCK_OUT.DO1** (über Plug-Adapter `ILOCK_OUT`): Weiterleitung des Reset-Signals nach unten.

Adapter

Adapter	Typ	Richtung	Beschreibung
`SET1`	unidirectional AX	Socket	Set-Eingang
`RESET`	unidirectional AX	Socket	Reset-Eingang
`ILOCK_IN`	bidirectional AX2	Socket	Interlock-Eingang (von oben)
`Q1`	unidirectional AX	Plug	Latch-Ausgang
`ILOCK_OUT`	bidirectional AX2	Plug	Interlock-Ausgang (nach unten)

Funktionsweise

Der Funktionsblock führt bei jedem eintreffenden Ereignis (über `SET1.E1`, `RESET.E1`, `ILOCK_IN.EO1` oder `ILOCK_OUT.EI1`) den Algorithmus `REQ` aus. Die interne Logik berechnet den neuen Ausgangswert `Q1.D1` nach der reset-dominanten Formel:

```
Q1.D1 := (SET1.D1 OR Q1.D1) AND NOT (RESET.D1 OR ILOCK_IN.DO1 OR ILOCK_OUT.DI1);
```

- Ist das Reset-Signal (`RESET.D1`) oder einer der Interlock-Eingänge (`ILOCK_IN.DO1` oder `ILOCK_OUT.DI1`) TRUE, wird der Latch sofort zurückgesetzt (Q = FALSE), unabhängig vom Set-Signal.
- Liegt kein Reset und keine Interlock-Blockade an, wird der Latch durch ein Set-Signal (`SET1.D1` = TRUE) gesetzt und bleibt selbsthaltend.
- Die Interlock-Signale der AX2-Schnittstelle werden propagiert:
  - `ILOCK_IN.DI1` (nach oben) = `RESET.D1 OR ILOCK_OUT.DI1`
  - `ILOCK_OUT.DO1` (nach unten) = `RESET.D1 OR ILOCK_IN.DO1`

Dadurch entsteht eine Kaskade, in der ein Reset-Signal in beide Richtungen (nach oben und nach unten) weitergegeben wird und alle untergeordneten sowie übergeordneten Latch-Bausteine blockiert.

Technische Besonderheiten

- **Adapterbasierte Schnittstellen**: Statt einzelner Ports verwendet der Baustein standardisierte AX/AX2-Adapter, die eine modulare und wiederverwendbare Verbindung mit anderen Komponenten ermöglichen.
- **Kaskadierbare Interlock-Kette**: Über die bidirektionalen `ILOCK_IN`/`ILOCK_OUT`-Adapter können mehrere ILOCK_FB_RS_AX-Instanzen in Reihe geschaltet werden. Ein Reset-Signal wird automatisch in beide Richtungen propagiert, sodass eine gesamte Kette blockiert wird.
- **Reset-Dominanz**: Der Reset hat stets Vorrang vor dem Set-Signal. Dies entspricht typischen Anforderungen an Sicherheitsverriegelungen, die einen definierten Vorrang des Rücksetzsignals verlangen.
- **Durchgehende Ereignisverarbeitung**: Der Baustein reagiert auf jedes eingehende Ereignis (Set, Reset, Interlock oben/unten) und aktualisiert sofort alle Ausgänge. Es gibt keine Zustandswechsel im ECC (nur ein Zustand `REQ`), die Logik ist rein kombinatorisch mit Rückkopplung über den gespeicherten Latch-Wert.

Zustandsübersicht

Der Baustein besitzt in seinem ECC nur einen Zustand `REQ`, der stets aktiv ist. Der innere Latch-Zustand wird durch den booleschen Wert `Q1.D1` repräsentiert. Die Zustandsübersicht beschreibt daher das Verhalten der internen Speicherzelle:

Aktuelles Q	SET1.D1	RESET.D1	Interlock aktiv?	Nächstes Q	Beschreibung
FALSE	FALSE	FALSE	NEIN	FALSE	Latch bleibt zurückgesetzt
FALSE	TRUE	FALSE	NEIN	TRUE	Latch wird gesetzt
FALSE	TRUE	TRUE	JA/NEIN	FALSE	Reset dominiert
TRUE	FALSE	FALSE	NEIN	TRUE	Latch bleibt gesetzt (Selbsthaltung)
TRUE	TRUE	FALSE	NEIN	TRUE	Latch bleibt gesetzt
TRUE	FALSE	TRUE	JA/NEIN	FALSE	Reset dominiert
TRUE	TRUE	TRUE	JA/NEIN	FALSE	Reset dominiert
beliebig	beliebig	beliebig	JA (ILOCK)	FALSE	Interlock-Blockade setzt oder hält Latch zurück

Anwendungsszenarien

- **Maschinensicherheit**: Verriegelung von Antrieben oder Werkzeugen, bei denen ein Reset-Signal (z. B. Not-Halt) Vorrang hat und zusätzlich benachbarte Module blockiert.
- **Prozesssteuerung mit Hierarchie**: In einer Kaskade von Steuerungsmodulen kann ein übergeordnetes Signal die untergeordneten Blöcke in den Grundzustand versetzen.
- **Redundante Verriegelung**: Mehrere ILOCK_FB_RS_AX-Instanzen können in einem Ring oder einer Linie angeordnet werden, um ein verteiltes Verriegelungssystem ohne zentrale Logik aufzubauen.
- **Latch mit externer Blockierung**: Über die Interlock-Adapter können externe Bedingungen (z. B. Sensorwerte oder Freigaben) den Latch blockieren, ohne die Set/Reset-Funktion zu beeinträchtigen.

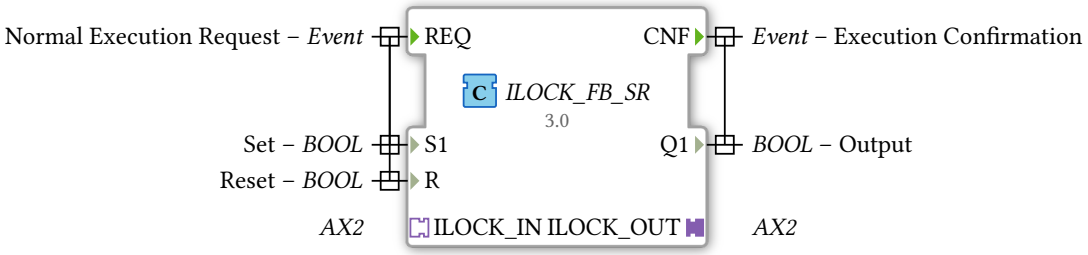
Vergleich mit ähnlichen Bausteinen

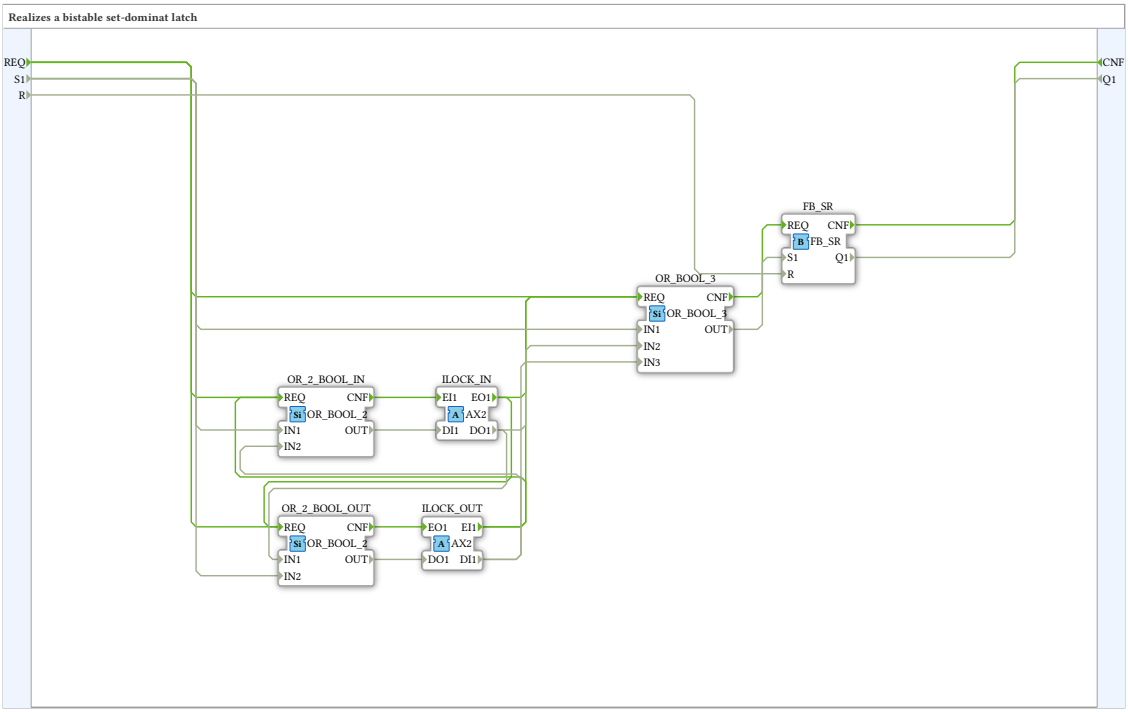
- **Standard RS-Latch (z. B. SR_FF)**: Ein typisches RS-Latch hat gleichberechtigte Set- und Reset-Eingänge. Der ILOCK_FB_RS_AX ist reset-dominant und bietet zusätzlich eine Interlock-Funktion zur Kaskadierung.
- **Reset-Dominantes Latch ohne Interlock**: Ein einfacher RS-Latch mit Reset-Dominanz hat nur Set/Reset-Ports. Der hier beschriebene Baustein erweitert dies um die bidirektionale Verriegelung über AX2-Adapter.
- **Interlock-Bausteine mit dedizierten Ports**: Manche Bibliotheken bieten Interlock-FBs mit einzelnen booleschen Eingängen für „InterlockFromAbove“ und „InterlockFromBelow“. Der ILOCK_FB_RS_AX kapselt diese Kanäle in Adaptern und vereinfacht so die Verbindung zwischen den Instanzen.

Fazit

Der **ILOCK_FB_RS_AX** ist ein spezialisierter Funktionsbaustein für verriegelungsbasierte Steuerungsaufgaben. Seine reset-dominante Logik gewährleistet eine definierte Priorität des Rücksetzsignals. Die Verwendung von AX/AX2-Adaptern ermöglicht eine saubere modulare Struktur und die einfache Kaskadierung mehrerer Bausteine. Dadurch eignet er sich besonders für hierarchische oder verteilte Sicherheits- und Verriegelungssysteme in der Automatisierungstechnik.

ILOCK_FB_SR





• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock ILOCK_FB_SR realisiert ein bistabiles set-dominantes SR-Latch (Set-Dominant Latch), das speziell für die Verwendung in Verriegelungsschaltungen (Interlocks) ausgelegt ist. Er erweitert ein einfaches SR-Latch um bidirektionale Adapter zur Kopplung mit anderen FB, sodass komplexe Verschaltungen und Prioritätslogiken abgebildet werden können.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung	Mitgeführte Daten
REQ	Ausführungsanforderung zur Verarbeitung der Set- und Reset-Signale	S1, R

Ereignis-Ausgänge

Ereignis	Beschreibung	Mitgeführte Daten
CNF	Bestätigung der Ausführung, gibt den aktuellen Ausgangswert aus	Q1

Daten-Eingänge

Name	Datentyp	Beschreibung
S1	BOOL	Set-Eingang
R	BOOL	Reset-Eingang

Daten-Ausgänge

Name	Datentyp	Beschreibung
Q1	BOOL	Ausgang des Latches (set-dominant)

Adapter

Adapter	Typ	Richtung	Beschreibung
ILOCK_IN	adapter::types::bidirectional::AX2	Socket	Empfängt Daten und Ereignisse von einem vorgelagerten Interlock-Baustein
ILOCK_OUT	adapter::types::bidirectional::AX2	Plug	Sendet Daten und Ereignisse an einen nachgelagerten Interlock-Baustein

Die Adapter ermöglichen eine bidirektionale Verkettung von Interlock-Bausteinen, indem sie sowohl den Empfang von Freigabe-/Verriegelungssignalen als auch die Weitergabe von Zuständen erlauben.

Funktionsweise

Der Kern des Bausteins ist ein internes SR-Latch (FB_SR), das **set-dominant** arbeitet: Wenn sowohl Set (S1) als auch Reset (R) gleichzeitig aktiv sind, setzt der Baustein den Ausgang Q1 auf TRUE.

Die Logik wird durch drei OR-Gatter ergänzt:

- **OR_3_BOOL** vereint die Signale des Set-Eingangs (S1) mit den Rückmeldungen der benachbarten Interlock-Bausteine (ILOCK_IN.DO1 und ILOCK_OUT.DI1). Das Ergebnis wird dem Set-Eingang des internen SR-Latches zugeführt.
- **OR_2_BOOL_IN** verknüpft das Set-Signal (S1) und das Signal des nachgeschalteten Interlock-Bausteins (ILOCK_OUT.DI1). Das Ergebnis wird über den Adapter ILOCK_IN.DI1 an den vorgelagerten Baustein gesendet.
- **OR_2_BOOL_OUT** verknüpft das Set-Signal (S1) und das Signal des vorgelagerten Interlock-Bausteins (ILOCK_IN.DO1). Das Ergebnis wird über den Adapter ILOCK_OUT.DO1 an den nachgelagerten Baustein gesendet.

Durch diese Struktur können mehrere Interlock-Bausteine kaskadiert werden: Ein Set-Signal kann durch einen vorgelagerten Baustein blockiert oder weitergeleitet werden. Gleichzeitig wird die Freigabe des eigenen Latches an die benachbarten Bausteine gemeldet.

Technische Besonderheiten

- **Set-dominantes Verhalten**: Im Konfliktfall (S1=TRUE, R=TRUE) hat das Set-Signal Vorrang.
- **Adapterbasierte Kopplung**: Die bidirektionalen Adapter ILOCK_IN und ILOCK_OUT erlauben eine modulare Verschaltung, ohne dass direkte Daten- oder Ereignisverbindungen zwischen den Bausteinen manuell gezogen werden müssen.
- **Rückkopplung**: Die interne Logik verarbeitet sowohl externe Set- und Reset-Signale als auch Zustände benachbarter Bausteine, wodurch eine dynamische Verriegelungskette entsteht.

Zustandsübersicht

Das interne SR-Latch kennt vier Zustände basierend auf den (kombinierten) Eingängen:

S1 (nach Logik)	R	Q1 (vorheriger)	Q1 (neuer)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Die effektiven Werte für S1 (Set) ergeben sich aus der ODER-Verknüpfung von `S1`, `ILOCK_IN.DO1` und `ILOCK_OUT.DI1`.

Anwendungsszenarien

- **Sicherheitsverriegelungen**: In Anlagen, bei denen mehrere Bedingungen gleichzeitig erfüllt sein müssen, bevor ein Vorgang freigegeben wird (z. B. Schutztüren, Not-Halt).
- **Kaskadierte Steuerungen**: Über- und untergeordnete Verriegelungseinheiten, die sich gegenseitig über Adapter beeinflussen (z. B. in Produktionslinien).
- **Prioritätsbasierte Setzungen**: Einsatz in Systemen, in denen ein Set-Signal Vorrang vor einem Reset haben muss, um sicherheitskritische Zustände zu erzwingen.

Vergleich mit ähnlichen Bausteinen

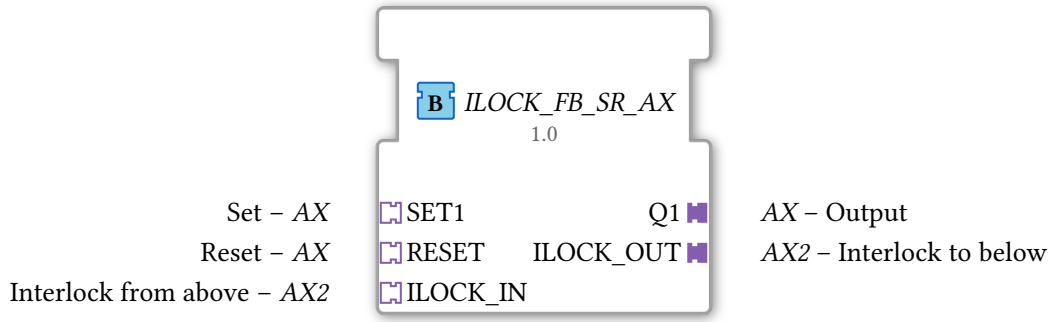
- **IEC 61131 SR (reset-dominant)**: Standard-SR-Latch, bei dem im Konfliktfall der Reset gewinnt. Der ILOCK_FB_SR kehrt diese Priorität um.
- **IEC 61131 RS (set-dominant)**: Ähnliches Verhalten im Kern, jedoch ohne die adapterbasierte Verschaltung und die Logik zur Verkettung mehrerer Bausteine.
- **D-Flipflop**: Getaktete Speicherzelle, die nur bei einer Flanke übernimmt – hier liegt ein asynchrones, taktunabhängiges Latch vor.

Der ILOCK_FB_SR ist speziell für Interlock-Ketten optimiert, während einfache SR-Latches isoliert arbeiten.

Fazit

Der ILOCK_FB_SR bietet eine flexible und robuste Lösung für set-dominante Verriegelungsschaltungen. Durch die Kombination eines SR-Latches mit bidirektionalen Adaptern und OR-Logik lassen sich beliebig komplexe Verkettungen aufbauen, ohne dass eine aufwändige Verdrahtung der Signale zwischen den Bausteinen erforderlich ist. Das set-dominante Verhalten ist insbesondere in sicherheitskritischen Anwendungen von Vorteil, wenn ein Set-Signal Vorrang haben muss.

ILOCK_FB_SR_AX



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ILOCK_FB_SR_AX** realisiert ein set-dominantes (Set-vorrangiges) bistabiles Flipflop (RS-Latch) mit Interlock-Funktion. Er ist für den Einsatz in Sicherheits- und Verriegelungsketten (Interlock) konzipiert, bei denen ein Set-Signal über mehrere Stufen propagiert und rückgekoppelt werden kann. Die Schnittstelle besteht aus zwei unidirektionalen AX-Adaptern (SET1, RESET) und einem AX2-Bidirektionaladapterpaar (ILOCK_IN / ILOCK_OUT) sowie einem Ausgangsadapter Q1. Der Baustein arbeitet ereignisgesteuert und führt bei jedem eingehenden Ereignis eine Neuberechnung der Ausgangswerte durch.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Datenherkunft	Beschreibung
`SET1.E1`	Socket SET1	Setzt den Ausgang Q1 (set-dominant)
`RESET.E1`	Socket RESET	Setzt den Ausgang Q1 zurück (nur wirksam, wenn SET1 = 0)
`ILOCK_IN.EO1`	Socket ILOCK_IN	Empfängt das Propagationsereignis von der übergeordneten Interlock-Stufe
`ILOCK_OUT.EI1`	Plug ILOCK_OUT	Empfängt ein Ereignis von der untergeordneten Interlock-Stufe (z. B. Bestätigung)

Ereignis-Ausgänge

Ereignis	Datenherkunft	Beschreibung
`Q1.E1`	Plug Q1	Ausgangsereignis nach jeder Aktualisierung von Q1
`ILOCK_IN.EI1`	Socket ILOCK_IN	Sendet Propagationsereignis an die übergeordnete Stufe
`ILOCK_OUT.EO1`	Plug ILOCK_OUT	Sendet Propagationsereignis an die untergeordnete Stufe

Daten-Eingänge

Daten	Typ	Beschreibung
SET1.D1	BOOL	Set-Eingang (dominant)
RESET.D1	BOOL	Reset-Eingang (aktiv, wenn SET1=0)
ILOCK_IN.DO1	BOOL	Daten von der übergeordneten Interlock-Stufe (Propagationssignal)
ILOCK_OUT.DI1	BOOL	Daten von der untergeordneten Interlock-Stufe (Rückkopplung)

Daten-Ausgänge

Daten	Typ	Beschreibung
Q1.D1	BOOL	Ausgang des Latches (gesetzt durch Set oder Interlock)
ILOCK_IN.DI1	BOOL	Propagiertes Set-Signal an die übergeordnete Stufe
ILOCK_OUT.DO1	BOOL	Propagiertes Set-Signal an die untergeordnete Stufe

Adapter

Adapter	Typ	Richtung	Beschreibung
Q1	AX (unidirectional)	Plug	Ausgang des Latches
ILOCK_OUT	AX2 (bidirectional)	Plug	Interlock-Schnittstelle zur untergeordneten Stufe
SET1	AX (unidirectional)	Socket	Set-Eingang
RESET	AX (unidirectional)	Socket	Reset-Eingang
ILOCK_IN	AX2 (bidirectional)	Socket	Interlock-Schnittstelle zur übergeordneten Stufe

Funktionsweise

Der FB implementiert einen Algorithmus, der bei jedem eingehenden Ereignis die folgenden Berechnungen durchführt:

1. **Q1.D1 (Latch-Ausgang):**
   `Q1.D1 := SET1.D1 OR ILOCK_IN.DO1 OR ILOCK_OUT.DI1 OR ((NOT RESET.D1) AND Q1.D1)`
   - Der Ausgang wird gesetzt, wenn **SET1.D1**, **ILOCK_IN.DO1** (von oben) oder **ILOCK_OUT.DI1** (von unten) den Wert `TRUE` haben.
   - Ist kein Set-Signal aktiv und **RESET.D1** = **FALSE**, bleibt der aktuelle Wert von Q1 erhalten (Speicherverhalten).
   - Set ist dominant: Ein aktives Set überschreibt einen Reset.
2. **ILOCK_IN.DI1 (Propagation nach oben):**
   `ILOCK_IN.DI1 := SET1.D1 OR ILOCK_OUT.DI1`
   - Das Set-Signal wird nach oben weitergegeben, wenn entweder der eigene Set-Eingang oder das Signal der unteren Stufe aktiv ist.
3. **ILOCK_OUT.DO1 (Propagation nach unten):**
   `ILOCK_OUT.DO1 := SET1.D1 OR ILOCK_IN.DO1`
   - Das Set-Signal wird nach unten weitergegeben, wenn entweder der eigene Set-Eingang oder das Signal der oberen Stufe aktiv ist.

Nach jeder Berechnung werden die Ereignisausgänge `Q1.E1`, `ILOCK_IN.EI1` und `ILOCK_OUT.EO1` aktiviert.

Technische Besonderheiten

- **Interlock-Kaskade:** Der FB kann in einer Kette von mehreren identischen Bausteinen verwendet werden. Über die AX2-Adapter (ILOCK) wird ein Set-Signal sowohl nach oben als auch nach unten propagiert. Dies ermöglicht eine wechselseitige Verriegelung (Interlock) über mehrere Stufen.

- **Bidirektionale Kommunikation:** Der AX2-Adapter erlaubt sowohl das Senden als auch das Empfangen von Ereignissen und Daten in beide Richtungen, sodass eine Rückkopplung von untergeordneten Stufen möglich ist.
- **Set-Dominanz:** Ein aktiver Set-Eingang setzt den Ausgang sofort und blockiert einen Reset. Dieser Baustein ist daher ein **SR-Flipflop mit Vorrang des Set-Signals**.
- **Reine Ereignissteuerung:** Der FB hat keinen zyklischen Ablauf; jede Änderung wird nur durch ein entsprechendes Ereignis ausgelöst.

Zustandsübersicht

Der FB besitzt einen einzigen internen Zustand **REQ**, in dem er sich immer befindet. Auf jedes eingehende Ereignis (SET1.E1, RESET.E1, ILOCK_IN.EO1, ILOCK_OUT.EI1) wird der Algorithmus ausgeführt und anschließend wieder in den Zustand REQ zurückgekehrt. Es gibt keine weiteren Zustände; der FB ist im Wesentlichen zustandslos (kombinatorisch mit Speicher durch Rückkopplung von Q1.D1).

Anwendungsszenarien

- **Sicherheitsverriegelung in der Agrartechnik:** Kaskadierung mehrerer ILOCK_FB_SR_AX, um eine durchgängige Verriegelungskette (z. B. für Maschinenachsen) zu realisieren. Ein Set-Signal von einer beliebigen Stelle blockiert alle nachfolgenden Stufen.
- **Produktionsanlagen mit Freigabeketten:** Setzen und Rücksetzen von Freigabesignalen in einer hierarchischen Steuerung, wobei höhere Stufen niedrigere dominieren.
- **Redundante Steuerungssysteme:** Nutzung der Interlock-Funktion zur gegenseitigen Überwachung und Verriegelung mehrerer Steuerungskanäle.

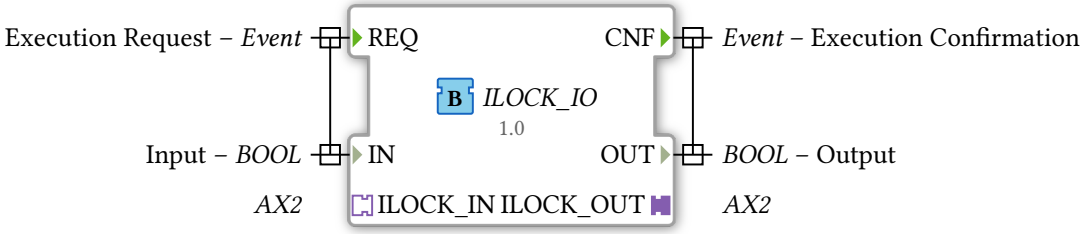
Vergleich mit ähnlichen Bausteinen

- **Standard RS-Flipflop (SR):** Ein einfacher SR-Baustein besitzt keinen Interlock-Mechanismus und keine bidirektionale Datenpropagation. Er kann nur lokal gesetzt oder zurückgesetzt werden.
- **ILOCK_FB_SR_AX** erweitert das klassische RS-Flipflop um eine Kaskadierungsfähigkeit über die AX2-Schnittstellen. Die Dominanz des Set-Signals und die Propagation in beide Richtungen machen ihn besonders für Sicherheitsketten geeignet.
- **Monostabile Bausteine:** Im Gegensatz zu monostabilen Funktionen behält dieser FB seinen Zustand bis zum nächsten Reset oder Set bei (bistabil).

Fazit

Der ILOCK_FB_SR_AX ist ein spezialisierter Funktionsblock für den Aufbau von Interlock-Ketten mit set-dominanter Latch-Funktion. Durch die Kombination von unidirektionalen Set/Reset-Eingängen und bidirektionalen Interlock-Adaptern eignet er sich hervorragend für sicherheitsgerichtete Anwendungen, bei denen eine Kaskadierung und gegenseitige Verriegelung mehrerer Stufen erforderlich ist. Seine einfache, ereignisgesteuerte Logik ermöglicht eine effiziente und deterministische Verarbeitung in der Automatisierungstechnik.

ILOCK_IO



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ILOCK_IO** realisiert eine kaskadierbare, momentane (nicht rücksetzbare) Verriegelung (Interlock) mit BOOL-Schnittstelle. Er wird verwendet, um in einer Kette mehrerer Bausteine sicherzustellen, dass immer nur ein Baustein seinen Ausgang auf TRUE setzt, solange sein Eingang aktiv ist und kein anderer Baustein in der Kette bereits aktiv ist. Der Baustein eignet sich besonders für Steuerungen, bei denen konkurrierende Anforderungen exklusiv bedient werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Datentyp	Kommentar
REQ	Event	Ausführungsanforderung (triggert die Verarbeitung des Eingangs IN)

Ereignis-Ausgänge

Ereignis	Datentyp	Kommentar
CNF	Event	Ausführungsbestätigung (signalisiert, dass der Ausgang OUT aktualisiert wurde)

Daten-Eingänge

Variable	Datentyp	Kommentar
IN	BOOL	Eingangswert (Setzanforderung)

Daten-Ausgänge

Variable	Datentyp	Kommentar
OUT	BOOL	Ausgangswert (freigegeben/gesperrt)

Adapter

Bezeichnung	Typ	Richtung	Beschreibung
ILOCK_IN	adapter::types::bidirectional::AX2	Socket	Eingangsadapter zum Empfang von Verriegelungsinformationen aus der Kette (vom Vorgänger)
ILOCK_OUT	adapter::types::bidirectional::AX2	Plug	Ausgangsadapter zum Senden von Verriegelungsinformationen an die Kette (an den Nachfolger)

Die Adapter vom Typ `AX2` sind bidirektional und enthalten die Datenpunkte `DI1` (Data Input 1) und `DO1` (Data Output 1). In der Kette werden `ILOCK_OUT.DO1` und `ILOCK_IN.DI1` sowie `ILOCK_OUT.DI1` und `ILOCK_IN.DO1` paarweise miteinander verbunden, um die Verriegelungslogik zu realisieren.

Funktionsweise

Der Baustein arbeitet als **momentaner Latch mit Interlock**. Bei jedem Aufruf des Ereignisses `REQ` wird der folgende Algorithmus ausgeführt:

1. Der Ausgang `OUT` wird auf TRUE gesetzt, wenn der Eingang `IN` TRUE ist **und** gleichzeitig kein anderer Baustein in der Kette aktiv ist. Die Aktivität anderer Bausteine wird über die Adapter abgefragt:
   - `ILOCK_IN.DO1`: Signal vom Vorgänger (ob dieser aktiv ist)
   - `ILOCK_OUT.DI1`: Signal vom Nachfolger (ob dieser aktiv ist)
   - Logik: `OUT := IN AND NOT (ILOCK_IN.DO1 OR ILOCK_OUT.DI1)`
2. Anschließend wird der eigene Zustand über die Adapter an die Nachbarn propagiert:
   - `ILOCK_IN.DI1 := OUT` – schreibt den Zustand zurück an den Vorgänger
   - `ILOCK_OUT.DO1 := OUT` – schreibt den Zustand an den Nachfolger

Die Kette ist so aufgebaut, dass immer nur ein Baustein aktiv sein kann. Wenn `IN` auf FALSE geht, wird `OUT` sofort FALSE, und die Verriegelung wird aufgehoben.

Technische Besonderheiten

- **Kaskadierbarkeit**: Durch die bidirektionalen Adapter können beliebig viele `ILOCK_IO`-Blöcke in Reihe geschaltet werden. Jeder Block prüft, ob sein Vorgänger oder Nachfolger bereits aktiv ist, bevor er selbst aktiv wird.
- **Momentanes Verhalten**: Der Ausgang ist nur solange TRUE, wie der Eingang aktiv ist. Es gibt keine Speicherfunktion – der Baustein ist kein Flipflop.
- **Exklusive Freigabe**: Auch bei mehreren gleichzeitigen Setzanforderungen kann nur genau ein Baustein aktiv werden. Die Priorität ergibt sich aus der Reihenfolge der `REQ`-Ereignisse (wer zuerst verarbeitet wird, blockiert die anderen).

Zustandsübersicht

Der Baustein besitzt einen einzigen **algorithmischen Zustand** `REQ`. Bei jedem Aufruf von `REQ` wird der Algorithmus ausgeführt, und es erfolgt sofort die Transition zurück zum selben Zustand. Es gibt keine Zustandsspeicherung – die Logik ist rein kombinatorisch mit Rückkopplung über die Adapter.

Zustand	Aktion	Ausgang
REQ	Führe Algorithmus `REQ` aus	`CNF` wird nach Berechnung gesendet

Anwendungsszenarien

- **Ampelsteuerung mit mehreren Anforderungstastern**: Jeder Taster ist mit einem `ILOCK_IO` verbunden; nur die zuerst gedrückte Taste erhält die Freigabe.
- **Exklusive Maschinensteuerung**: Mehrere Bedienpulte konkurrieren um die Steuerung einer Maschine – nur das aktive Pult erhält die Berechtigung.
- **Sicherheitsverriegelung**: In einer Förderanlage blockieren sich gegenseitig mehrere Schutztüren; nur eine Tür kann gleichzeitig geöffnet sein.

Vergleich mit ähnlichen Bausteinen

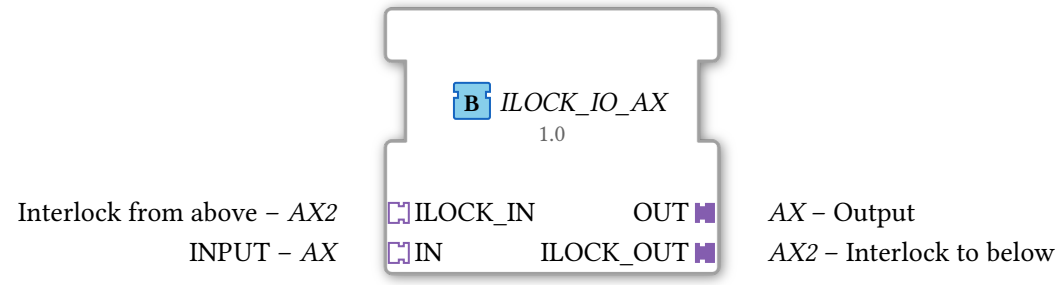
- **SR-Flipflop (`SR`)** – speichert einen Zustand, `ILOCK_IO` speichert nicht (momentan).
- **Monostabile Kippstufe (`RTRIG`, `FTRIG`)** – reagiert nur auf Flanken, `ILOCK_IO` arbeitet pegelgesteuert.

- **Einfacher `AND`-Gatter-Baustein** – nur lokale Logik, keine Vernetzung mit anderen Bausteinen. `ILOCK_IO` bietet diese Vernetzung über Adapter.

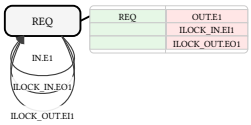
Fazit

Der Baustein `ILOCK_IO` ist ein spezialisierter Funktionsblock für die Realisierung von exklusiven Freigabeketten. Durch seine einfache Schnittstelle und die Kaskadierbarkeit eignet er sich ideal für modulare Steuerungen, bei denen mehrere konkurrierende Anforderungen verwaltet werden müssen. Die bidirektionalen Adapter ermöglichen eine flexible und erweiterbare Verdrahtung.

ILOCK_IO_AX



ECC: ILOCK_IO_AX



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ILOCK_IO_AX** realisiert einen kettbaren (chainable) Momentanschalter (Momentary Latch) mit Interlock-Funktion. Er stellt sicher, dass immer nur ein Element in einer Kette aktiv ist – sobald ein anderer FB in der Kette aktiv wird, wird dieser zurückgesetzt. Der FB ist speziell für die Verwendung mit AX/AX2-Adapterschnittstellen ausgelegt.

Schnittstellenstruktur

Der Baustein besitzt keine klassischen Ereignis- oder Datenein-/ausgänge auf FB-Ebene. Die gesamte Kommunikation erfolgt über **Adapter-Schnittstellen**. Nachfolgend sind die über die Adapter übertragenen Signale aufgeschlüsselt.

Ereignis-Eingänge

Signal	Adapter	Beschreibung
`IN.E1`	IN (AX)	Auslöser für die Verarbeitung des Eingangssignals.
`ILOCK_IN.EO1`	ILOCK_IN (AX2)	Ereignis vom übergeordneten FB in der Kette (z. B. wenn dort ein anderer Ausgang aktiv wird).
`ILOCK_OUT.EI1`	ILOCK_OUT (AX2)	Ereignis vom untergeordneten FB in der Kette (z. B. wenn dieser aktiv wird).

Ereignis-Ausgänge

Signal	Adapter	Beschreibung
`OUT.E1`	OUT (AX)	Bestätigung, dass die Verarbeitung abgeschlossen ist.
`ILOCK_IN.EI1`	ILOCK_IN (AX2)	Ereignis an den übergeordneten FB (wird bei jedem Verarbeitungszyklus gesendet).
`ILOCK_OUT.EO1`	ILOCK_OUT (AX2)	Ereignis an den untergeordneten FB (wird bei jedem Verarbeitungszyklus gesendet).

Daten-Eingänge

Signal	Adapter	Beschreibung
`IN.D1`	IN (AX)	Setzeingang (BOOL) – wenn TRUE und kein anderer FB in der Kette aktiv, wird der Ausgang aktiv.
`ILOCK_IN.DO1`	ILOCK_IN (AX2)	Zustand des übergeordneten FBs (TRUE bedeutet, dass dort der Ausgang aktiv ist).
`ILOCK_OUT.DI1`	ILOCK_OUT (AX2)	Zustand des untergeordneten FBs (TRUE bedeutet, dass dort der Ausgang aktiv ist).

Daten-Ausgänge

Signal	Adapter	Beschreibung
`OUT.D1`	OUT (AX)	Ausgangssignal (BOOL) – aktiv, wenn `IN.D1` gesetzt und kein anderer FB in der Kette aktiv ist.
`ILOCK_IN.DI1`	ILOCK_IN (AX2)	Eigenes Ausgangssignal wird an den übergeordneten FB weitergegeben.
`ILOCK_OUT.DO1`	ILOCK_OUT (AX2)	Eigenes Ausgangssignal wird an den untergeordneten FB weitergegeben.

Adapter

Adapter	Richtung	Typ	Beschreibung
`IN`	Socket (Eingang)	`adapter::types::unidirectional::AX`	Eingangssignal vom Anwender (Setzeingang und Ereignis).
`OUT`	Plug (Ausgang)	`adapter::types::unidirectional::AX`	Ausgangssignal zum Anwender (Ausgangswert und Bestätigung).
`ILOCK_IN`	Socket (Eingang)	`adapter::types::bidirectional::AX2`	Verbindung zum **übergeordneten** FB in der Kette.

ILOCK_OUT	Plug (Ausgang)	adapter::types::bidirectional::AX2	Verbindung zum **untergeordneten** FB in der Kette.

Funktionsweise

Der FB arbeitet als **Momentanschalter mit Interlock** (auch „exklusiver Latch“). Im Zustand REQ wird folgende Logik ausgeführt:

- Der Ausgang OUT.D1 wird nur dann **TRUE**, wenn der Eingang IN.D1 aktiv ist **und** kein anderer FB in der Kette gerade aktiv ist (ILOCK_IN.DO1 vom Vorgänger und ILOCK_OUT.DI1 vom Nachfolger müssen FALSE sein).
- Der eigene aktive Zustand (OUT.D1) wird über die Adapter ILOCK_IN.DI1 (an den Vorgänger) und ILOCK_OUT.DO1 (an den Nachfolger) weitergegeben.
- Jeder auslösende Ereigniseingang (egal ob von IN.E1, ILOCK_IN.EO1 oder ILOCK_OUT.EI1) führt zur erneuten Berechnung der Logik und zur Ausgabe aller Ereignisse (OUT.E1, ILOCK_IN.EI1, ILOCK_OUT.EO1).

Durch die bidirektionale Weitergabe des Zustands über die ILOCK-Adapter wissen alle FBs in der Kette voneinander, sodass nur einer aktiv sein kann.

Technische Besonderheiten

- **Kettbare Interlock-Struktur**: Mehrere ILOCK_IO_AX-Bausteine können hintereinander geschaltet werden, indem ILOCK_OUT eines FBs mit ILOCK_IN des nächsten verbunden wird. Die Kette kann beliebig lang sein.
- **Keine eigenen Ereignisse/Daten auf FB-Ebene**: Die gesamte Signalübertragung erfolgt über die standardisierten AX/AX2-Adapter. Dies ermöglicht eine einfache Integration in bestehende Adapter-basierte Systeme.
- **Immer nur eine Verarbeitung**: Da nur ein Zustand (REQ) existiert, wird bei jedem Ereignis dieselbe Logik ausgeführt. Es gibt keine internen Zustandswechsel.

Zustandsübersicht

Der FB besitzt nur einen einzigen Zustand:

```
+-------+
|  REQ  |  (Standardzustand)
+-------+
    | ^
    | | (Ereignisse: IN.E1, ILOCK_IN.EO1, ILOCK_OUT.EI1)
    +-+
```

Im Zustand REQ werden die Algorithmus REQ und die Ereignisausgaben ausgeführt. Der Zustand wird nie verlassen.

Anwendungsszenarien

- **Steuerung von Schaltgeräten** (z. B. Ventile, Motoren) in einer Linie, bei der immer nur eines aktiv sein darf.
- **Prioritätslose Auswahl** aus mehreren Quellen (z. B. Tasterfeld mit Interlock).
- **Verkettete Sicherheitssteuerungen**, bei denen eine Aktivierung einer nachfolgenden Stufe die vorherige deaktivieren muss (oder umgekehrt).

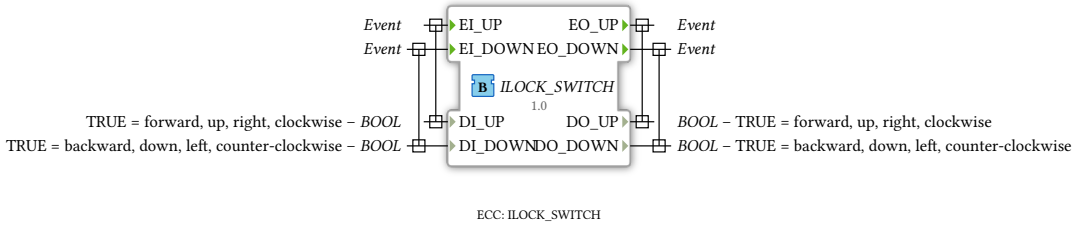
Vergleich mit ähnlichen Bausteinen

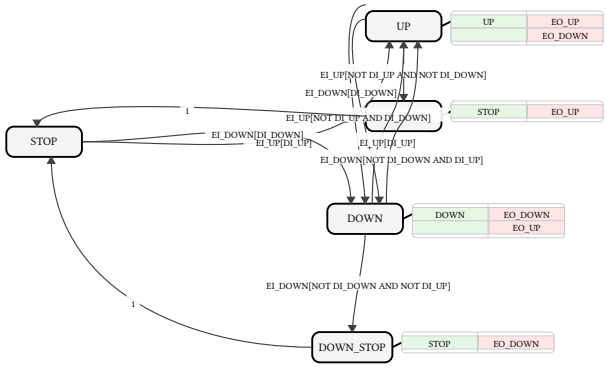
Baustein	Beschreibung	Unterschied zu ILOCK_IO_AX
SR-Flipflop	Setzt Ausgang bei SET, rückgesetzt bei RESET.	Kein Interlock – mehrere FBs können gleichzeitig aktiv sein.
Einfacher Latch (z. B. LATCH_AX)	Hält Wert solange, bis erneut getriggert.	Keine Kopplung mit anderen FBs.
ILOCK_IO_AX	Exklusive Aktivierung in einer Kette.	Erzwingt, dass nur ein FB aktiv ist – ideal für Liniensteuerungen.

Fazit

Der **ILOCK_IO_AX** ist ein spezialisierter Funktionsblock für exklusive Aktivierungen in geketteten Anordnungen. Durch die Verwendung von AX/AX2-Adaptern und die bidirektionale Zustandsrückmeldung wird eine robuste Interlock-Logik ohne separate Verdrahtung realisiert. Er eignet sich besonders für modulare Steuerungsstrukturen, bei denen mehrere Aktoren nur nacheinander oder einzeln aktiv sein dürfen.

ILOCK_SWITCH





• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsbaustein **ILOCK_SWITCH** dient als prioritätsgesteuerte Weiche mit Verriegelungsfunktion. Er wertet zwei Ereignissignale (**EI_UP** und **EI_DOWN**) in Kombination mit zugehörigen booleschen Datenwerten aus und setzt die Ausgänge **DO_UP** und **DO_DOWN** entsprechend. Dabei wird stets der zuletzt aktive Eingang priorisiert – eine gleichzeitige Aktivierung beider Ausgänge ist ausgeschlossen. Die Logik verhindert Oszillationen und sorgt für einen definierten Zustand auch bei ungültigen oder widersprüchlichen Eingangsbelegungen.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	mit Daten	Beschreibung
EI_UP	DI_UP	Ereignis zur Anforderung der Aufwärts-/Vorwärts-Richtung.
EI_DOWN	DI_DOWN	Ereignis zur Anforderung der Abwärts-/Rückwärts-Richtung.

Ereignis-Ausgänge

Ereignis	mit Daten	Beschreibung
EO_UP	DO_UP	Wird bei Aktivierung der Aufwärts-Richtung oder beim Verlassen des UP-Zustands getriggert.
EO_DOWN	DO_DOWN	Wird bei Aktivierung der Abwärts-Richtung oder beim Verlassen des DOWN-Zustands getriggert.

Daten-Eingänge

Name	Typ	Kommentar
DI_UP	BOOL	TRUE = vorwärts, aufwärts, rechts, im Uhrzeigersinn
DI_DOWN	BOOL	TRUE = rückwärts, abwärts, links, gegen Uhrzeigersinn

Daten-Ausgänge

Name	Typ	Kommentar
DO_UP	BOOL	TRUE = vorwärts, aufwärts, rechts, im Uhrzeigersinn
DO_DOWN	BOOL	TRUE = rückwärts, abwärts, links, gegen Uhrzeigersinn

Adapter

Keine.

Funktionsweise

Der Baustein realisiert einen endlichen Automaten mit sechs Zuständen. Die grundlegende Idee: **Der zuletzt eingetroffene gültige Befehl setzt den Ausgangszustand**. Dabei wird ein Befehl nur dann als gültig betrachtet, wenn der zugehörige boolesche Datenwert **TRUE** ist.

- **STOP** (Ruhezustand): Beide Ausgänge sind **FALSE**.
- **UP**: DO_UP = TRUE, DO_DOWN = FALSE.
  Wird erreicht, wenn im STOP-Zustand **EI_UP** mit **DI_UP** = **TRUE** eintritt.
- **DOWN**: DO_UP = FALSE, DO_DOWN = TRUE.
  Wird erreicht, wenn im STOP-Zustand **EI_DOWN** mit **DI_DOWN** = **TRUE** eintritt.
- **UP_STOP**: Zwischenzustand, der eingenommen wird, wenn während des UP-Zustands ein **EI_UP** eintritt, aber **DI_UP** = **FALSE** (und auch **DI_DOWN** = **FALSE**).
  In UP_STOP werden beide Ausgänge auf FALSE gesetzt und anschließend sofort nach STOP zurückgegangen.
- **DOWN_STOP**: Analog zu UP_STOP für den DOWN-Zustand bei **EI_DOWN** mit **DI_DOWN** = **FALSE**.
- Direkte Umschaltungen sind möglich:
  Von **UP** → **DOWN**, wenn **EI_DOWN** mit **DI_DOWN** = **TRUE** eintrifft oder **EI_UP** mit **DI_UP** = **FALSE** und **DI_DOWN** = **TRUE** (implizite Anforderung der Gegenrichtung).
  Von **DOWN** → **UP** analog.

Durch diese Logik wird sichergestellt, dass stets nur eine Richtung aktiv ist und bei ungültigen Signalen (FALSE-Werte) der Baustein sauber in den Ruhezustand zurückfällt.

Technische Besonderheiten

- **Verriegelung durch Zustandsautomaten**: Keine Möglichkeit, beide Ausgänge gleichzeitig auf TRUE zu setzen – selbst bei zeitgleichen Ereignissen wird sequenziell geschaltet.
- **Zwischenzustände (UP_STOP, DOWN_STOP)**: Diese vermeiden unerwünschtes Flackern der Ausgänge, indem sie bei ungültigen Signalen sofort die Ausgänge zurücksetzen und nach STOP übergehen.

- **Implizite Umschaltung**: Ein Ereignis, dessen Datenwert die Gegenrichtung anfordert (z. B. EI_UP mit DI_UP=FALSE und DI_DOWN=TRUE), bewirkt direkt den Richtungswechsel, ohne dass ein separates EI_DOWN-Ereignis nötig ist.

Zustandsübersicht

Zustand	DO_UP	DO_DOWN	Erreicht durch
STOP	FALSE	FALSE	Start / nach UP_STOP und DOWN_STOP
UP	TRUE	FALSE	EI_UP mit DI_UP=TRUE aus STOP oder DOWN
DOWN	FALSE	TRUE	EI_DOWN mit DI_DOWN=TRUE aus STOP oder UP
UP_STOP	FALSE	FALSE	EI_UP mit DI_UP=FALSE und DI_DOWN=FALSE im Zustand UP
DOWN_STOP	FALSE	FALSE	EI_DOWN mit DI_DOWN=FALSE und DI_UP=FALSE im Zustand DOWN

Die Übergänge zwischen den Zuständen erfolgen immer über ein eingehendes Ereignis und die Auswertung der aktuellen Datenwerte.

Anwendungsszenarien

- **Motorsteuerung für Lineartriebe, Schwenkarme oder Hubvorrichtungen**, bei denen eine gleichzeitige Bewegung in beide Richtungen mechanisch oder sicherheitstechnisch ausgeschlossen werden muss.
- **Verriegelung von Ventilantrieben** (Öffnen / Schließen) mit Rückmeldung über Endschalter.
- **Bedienschnittstellen mit Tastern für „Auf“ und „Ab“**, wobei der letzte Tastendruck Vorrang hat und eine dauerhafte Blockade vermieden wird.

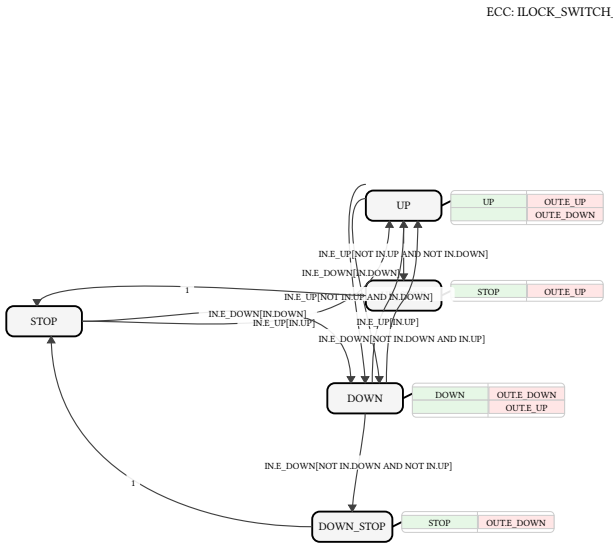
Vergleich mit ähnlichen Bausteinen

- Ein einfaches **AND- / OR-Gatter** kann keine Verriegelung realisieren und würde bei gleichzeitigen TRUE-Signalen beide Ausgänge aktivieren.
- Ein **RS-Flipflop** speichert einen Zustand, erlaubt aber unter Umständen beide Setz-/Rücksetz-Eingänge gleichzeitig (metastabil). ILOCK_SWITCH vermeidet dies durch die starre Übergangslogik.
- Ein **Mux** würde nur einen Datenwert durchschalten, kann aber keine ereignisgesteuerte Priorisierung mit Zwischenzuständen abbilden. ILOCK_SWITCH bietet eine speziell für Antriebsverriegelungen optimierte Lösung.

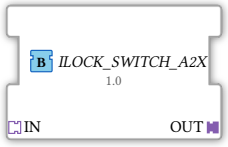
Fazit

Der **ILOCK_SWITCH** ist ein robuster, ereignisgesteuerter Funktionsbaustein zur verriegelten Richtungsansteuerung. Durch die Kombination von Zustandsautomat, Datenabhängigkeit und expliziten Zwischenzuständen werden typische Probleme wie gleichzeitige Aktivierung beider Ausgänge oder Oszillationen zuverlässig ausgeschlossen. Er eignet sich besonders für sicherheitskritische Steuerungen in der Automatisierungstechnik, bei denen eine eindeutige und rücksetzbare Vorranglogik gefordert ist.

ILOCK_SWITCH_A2X



Input for forward/up and backward/down direction – *A2X*



A2X – Output for forward/up and backward/down direction

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ILOCK_SWITCH_A2X` ist ein Verriegelungsbaustein für zwei Antriebsrichtungen. Er empfängt über einen unidirektionalen `A2X`-Adapter Steuersignale und gibt sie über einen zweiten `A2X`-Adapter ausgegeben. Dabei wird sichergestellt, dass niemals beide Ausgangsrichtungen gleichzeitig aktiv sind. Das zuletzt wirksame Eingangssignal erhält die Priorität und bestimmt die aktive Ausgangsrichtung. Zusätzlich werden definierte Stoppzustände unterstützt.

Schnittstellenstruktur

Der Baustein besitzt keine klassischen Ereignis- oder Datenein-/ausgänge. Die Kommunikation erfolgt ausschließlich über zwei A2X-Adapter:

- Der Socket `IN` dient als Eingangsschnittstelle.
- Der Plug `OUT` dient als Ausgangsschnittstelle.

Ereignis-Eingänge

Über den Socket `IN` werden folgende Ereignisse empfangen:

- `IN.E_UP` – Ereignis für die Aufwärts-/Vorwärtsrichtung.
- `IN.E_DOWN` – Ereignis für die Abwärts-/Rückwärtsrichtung.

Ereignis-Ausgänge

Über den Plug `OUT` werden folgende Ereignisse gesendet:

- `OUT.E_UP` – Wird ausgegeben, wenn sich der Datenausgang `OUT.UP` ändert.
- `OUT.E_DOWN` – Wird ausgegeben, wenn sich der Datenausgang `OUT.DOWN` ändert.

Da beim Richtungswechsel beide Datenausgänge wechseln, werden in den Zuständen `UP` und `DOWN` jeweils beide Ereignisse gesendet.

Daten-Eingänge

Über den Socket `IN` stehen folgende Datenwerte zur Verfügung:

- `IN.UP` (BOOL) – Zustand des Aufwärts-/Vorwärts-Eingangssignals.
- `IN.DOWN` (BOOL) – Zustand des Abwärts-/Rückwärts-Eingangssignals.

Daten-Ausgänge

Über den Plug `OUT` werden folgende Datenwerte bereitgestellt:

- `OUT.UP` (BOOL) – Aktives Ausgangssignal für die Aufwärts-/Vorwärtsrichtung.
- `OUT.DOWN` (BOOL) – Aktives Ausgangssignal für die Abwärts-/Rückwärtsrichtung.

Es ist immer nur einer der beiden Ausgänge `TRUE`.

Adapter

- `IN` – Socket vom Typ `adapter::types::unidirectional::A2X`; Eingangsadapter für Ansteuersignale.
- `OUT` – Plug vom Typ `adapter::types::unidirectional::A2X`; Ausgangsadapter für die weitergereichten Signale.

Der A2X-Adapter ist unidirektional. Die Informationsrichtung verläuft vom Plug zum Socket. Der Funktionsblock nutzt den Socket `IN` zum Empfangen und den Plug `OUT` zum Senden.

Funktionsweise

Die Verarbeitung erfolgt über einen ereignisgesteuerten Zustandsautomaten mit fünf Zuständen.

Im Ruhezustand `STOP` wartet der Baustein auf eine aktive Anforderung. Trifft `IN.E_UP` mit `IN.UP = TRUE` ein, wechselt er in den Zustand `UP`. Trifft `IN.E_DOWN` mit `IN.DOWN = TRUE` ein, wechselt er in den Zustand `DOWN`.

Befindet sich der Baustein im Zustand `UP`, wechselt er nach `DOWN`, sobald entweder:

- `IN.E_DOWN` mit `IN.DOWN = TRUE` eintrifft, oder
- `IN.E_UP` mit `IN.UP = FALSE` und `IN.DOWN = TRUE` eintrifft.

Im zweiten Fall wird die Freigabe der Aufwärtsrichtung erkannt, während die Abwärtsrichtung noch aktiv ist. Die Ausgangsrichtung wechselt dann auf `DOWN`. Analog gilt dies für den Wechsel von `DOWN` nach `UP`.

Stoppsignale werden über das Ereignis der zuvor aktiven Richtung übertragen. Bei aktivem `UP` und `IN.E_UP` mit `IN.UP = FALSE` und `IN.DOWN = FALSE` wird der transiente Zustand `UP_STOP` durchlaufen. Dort werden die Ausgänge zurückgesetzt und `OUT.E_UP` gesendet. Anschließend wechselt der Automat sofort in den Ruhezustand `STOP`. Entsprechendes gilt für `DOWN` und `DOWN_STOP` mit dem Ereignis `OUT.E_DOWN`.

Technische Besonderheiten

- Die komplette Signalübertragung erfolgt ausschließlich über A2X-Adapter.
- Die Interlock-Logik verhindert ein gleichzeitiges Aktivieren beider Ausgangsrichtungen.
- Die „Last-Active-Priorität“ wird durch die Auswertung der Ereignisse zusammen mit den Datenwerten umgesetzt.
- Die transienten Zustände `UP_STOP` und `DOWN_STOP` realisieren eine definierte Stopp-Behandlung mit korrespondierender Ereignisausgabe.
- Der Baustein ist für den Einsatz in Signalverarbeitungsketten vorgesehen und verwendet den Paketnamen `logiBUS::signalprocessing::interlock`.

Zustandsübersicht

Zustand	Bedeutung	`OUT.UP`	`OUT.DOWN`	Ereignisausgabe beim Betreten
`STOP`	Ruhezustand	`FALSE`	`FALSE`	–
`UP`	Aufwärtsrichtung aktiv	`TRUE`	`FALSE`	`OUT.E_UP`, `OUT.E_DOWN`
`DOWN`	Abwärtsrichtung aktiv	`FALSE`	`TRUE`	`OUT.E_DOWN`, `OUT.E_UP`
`UP_STOP`	Transienter Stopp von `UP`	`FALSE`	`FALSE`	`OUT.E_UP`

DOWN_STOP	Transienter Stopp von DOWN	FALSE	FALSE	OUT.E_DOWN

Anwendungsszenarien

- Ansteuerung von Motoren, Ventilen oder Aktoren, die nur eine Richtung gleichzeitig aktiv haben dürfen.
- Verriegelung zwischen einer Bedieneinheit und einem Antrieb, z. B. bei Hebe- und Senkmechanismen.
- Aufbereitung konkurrierender Steuersignale in Automatisierungs- und Steuerungssystemen.
- Einsatz in A2X-basierten Signalverarbeitungsketten, bei denen eine Richtungsfreigabe mit Priorität benötigt wird.

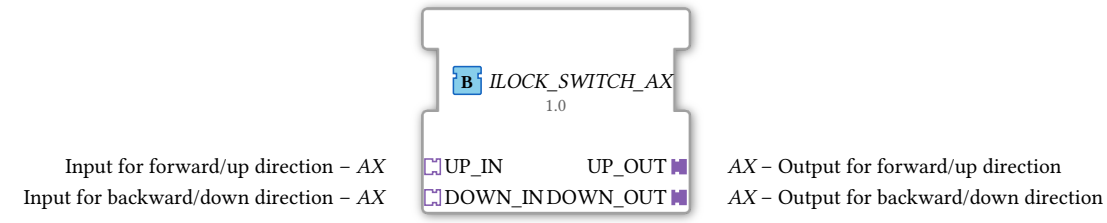
Vergleich mit ähnlichen Bausteinen

- Ein einfacher Durchschleif-Baustein würde die Signale unverändert weitergeben, ohne eine Verriegelung oder Priorisierung vorzunehmen.
- Gegenüber rein kombinatorischen Verriegelungslogiken bietet dieser Baustein eine explizite Ereignissteuerung und definierte Stoppzustände.
- Die „Last-Active-Priorität“ unterscheidet ihn von Bausteinen, die feste Vorrangregeln als Schaltpriorität verwenden.

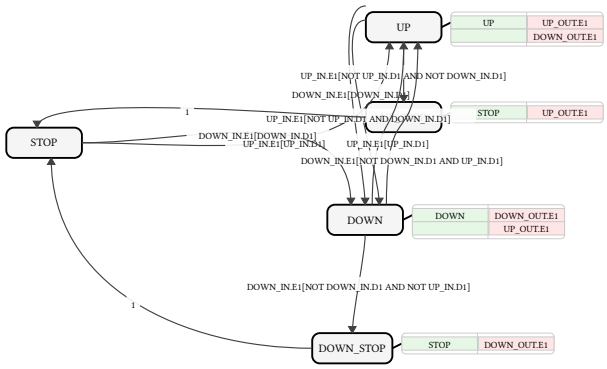
Fazit

Der ILOCK_SWITCH_A2X ist eine geeignete Lösung zur sicheren Weiterleitung von zwei Richtungsbefehlen über eine A2X-Schnittstelle. Er verhindert gleichzeitige Aktivierungen, gewährleistet eine eindeutige Richtungspriorität und stellt über die transienten Zustände eine klare Stopp-Semantik bereit. Damit eignet er sich für vielfältige Steuerungsaufgaben in der Automatisierungstechnik.

ILOCK_SWITCH_AX



ECC: ILOCK_SWITCH_AX



(Kein Bild verfügbar)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ILOCK_SWITCH_AX** realisiert eine verriegelnde Weichensteuerung für bidirektionale Achsen. Er priorisiert den zuletzt aktiven Eingang und stellt sicher, dass nie beide Richtungen gleichzeitig aktiviert werden. Die Kommunikation erfolgt ausschließlich über Adapter-Schnittstellen vom Typ `adapter::types::unidirectional::AX`. Diese Version ist speziell für den Einsatz in modularen Steuerungssystemen mit standardisierten Adaptern konzipiert.

Schnittstellenstruktur

Ereignis-Eingänge

Keine (sämtliche Ereignisse werden über die Adapter-Sockets übertragen).

Ereignis-Ausgänge

Keine (sämtliche Ereignisse werden über die Adapter-Plugs übertragen).

Daten-Eingänge

Keine (sämtliche Daten werden über die Adapter-Sockets übertragen).

Daten-Ausgänge

Keine (sämtliche Daten werden über die Adapter-Plugs übertragen).

Adapter

Name	Typ	Richtung	Kommentar
UP_IN	`adapter::types::unidirectional::AX`	Socket	Eingang für Vorwärts-/Aufwärts-Richtung
DOWN_IN	`adapter::types::unidirectional::AX`	Socket	Eingang für Rückwärts-/Abwärts-Richtung
UP_OUT	`adapter::types::unidirectional::AX`	Plug	Ausgang für Vorwärts-/Aufwärts-Richtung
DOWN_OUT	`adapter::types::unidirectional::AX`	Plug	Ausgang für Rückwärts-/Abwärts-Richtung

Jeder Adapter stellt einen Ereignisausgang `E1` und einen Daten-Eingang/Ausgang `D1` (vom Typ BOOL) bereit. Die Sockets empfangen Anforderungen, die Plugs geben die geschalteten Zustände weiter.

Funktionsweise

Die Logik wird durch einen endlichen Zustandsautomaten (ECC) realisiert. Der FB befindet sich zunächst im Zustand `STOP`. Ein Wechsel in die Zustände `UP` oder `DOWN` erfolgt, wenn der entsprechende Socket ein Ereignis `E1` mit gesetztem Datenwert `D1` liefert (d.h. `UP_IN.E1[UP_IN.D1]` oder `DOWN_IN.E1[DOWN_IN.D1]`).

- Im Zustand `UP` wird `UP_OUT.D1` auf `TRUE` und `DOWN_OUT.D1` auf `FALSE` gesetzt.
- Im Zustand `DOWN` wird `DOWN_OUT.D1` auf `TRUE` und `UP_OUT.D1` auf `FALSE` gesetzt.

Ein Wechsel vom aktuellen Zustand in den jeweils anderen ist nur möglich, wenn der andere Socket aktiv wird (Priorisierung des letzten aktiven Eingangs). Dabei werden die Bedingungen differenziert:

- **UP → DOWN**: entweder direkt durch `DOWN_IN.E1[DOWN_IN.D1]` oder wenn der eigene Eingang wegfällt und der andere aktiv ist (`UP_IN.E1[NOT UP_IN.D1 AND DOWN_IN.D1]`).
- **DOWN → UP**: analog durch `UP_IN.E1[UP_IN.D1]` oder `DOWN_IN.E1[NOT DOWN_IN.D1 AND UP_IN.D1]`.

Fällt der aktive Eingang weg, ohne dass der andere aktiv wird, wird ein Zwischenzustand (`UP_STOP` bzw. `DOWN_STOP`) erreicht. Dort werden die Ausgänge auf `FALSE` gesetzt, und anschließend wechselt der Automat sofort in den `STOP`-Zustand.

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Der Funktionsblock verwendet ausschließlich Adapter des Typs `unidirectional::AX`, was eine saubere Kapselung von Ereignis- und Datenkommunikation ermöglicht und die Wiederverwendung in verschiedenen Kontexten erleichtert.
- **Priorisierung des letzten aktiven Eingangs**: Im Gegensatz zu einem einfachen Zustandsautomaten, der nur auf steigende Flanken reagiert, wird hier das zuletzt gesendete gültige Signal berücksichtigt.
- **Sichere Verriegelung**: Durch die ECC-Struktur ist ausgeschlossen, dass beide Ausgänge gleichzeitig `TRUE` werden.
- **Explizite Stop-Zwischenzustände**: Die Zustände `UP_STOP` und `DOWN_STOP` gewährleisten eine definierte Reaktion auf das Verschwinden eines Steuersignals, bevor in den Ruhezustand zurückgekehrt wird.

Zustandsübersicht

Zustand	UP_OUT.D1	DOWN_OUT.D1	Beschreibung
`STOP`	FALSE	FALSE	Ruhezustand, keine Richtung aktiv
`UP`	TRUE	FALSE	Vorwärts-/Aufwärts-Richtung aktiv
`DOWN`	FALSE	TRUE	Rückwärts-/Abwärts-Richtung aktiv
`UP_STOP`	FALSE	FALSE	Zwischenzustand nach Verlassen von UP
`DOWN_STOP`	FALSE	FALSE	Zwischenzustand nach Verlassen von DOWN

Anwendungsszenarien

- **Steuerung von Hub-/Senkantrieben** (z.B. Hebezeug, Hydraulikzylinder) – verhindert gleichzeitiges Heben und Senken.
- **Verriegelung von Weichen** in Fördersystemen – priorisiert die jeweils letzte Anforderung.
- **Sicherheitsgerichtete Umschaltung** in modularen Automatisierungsanlagen, bei denen ein Adapterbasiertes Datenmodell verwendet wird (z.B. logiBUS).

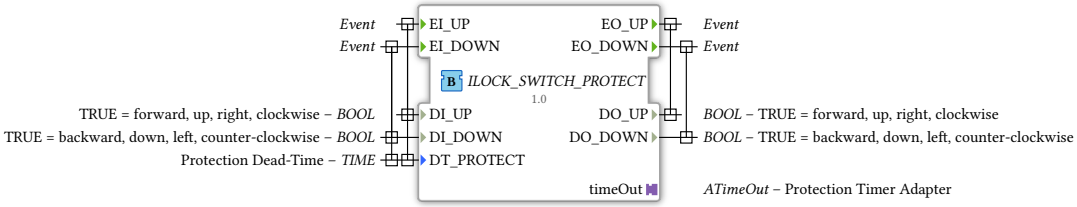
Vergleich mit ähnlichen Bausteinen

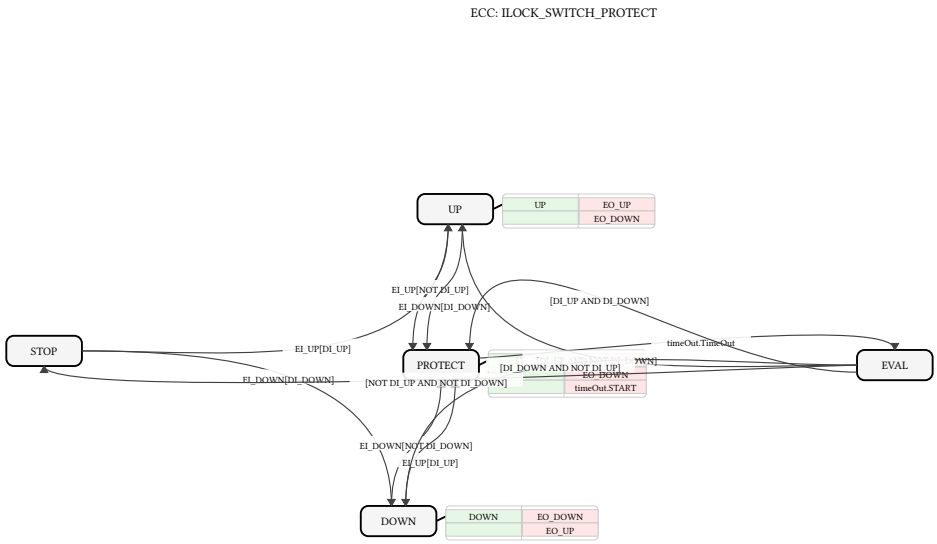
- **ILOCK_SWITCH** (ohne Adapter): Bietet ähnliche Verriegelungslogik, verwendet aber direkte Ereignis- und Datenports. Die Adapter-Version `ILOCK_SWITCH_AX` erlaubt eine einfachere Kopplung an standardisierte Schnittstellen und ist besser für hierarchische Architekturen geeignet.
- **SR-Flipflop** (Set-Reset): Kann ebenfalls zwei Zustände verriegeln, priorisiert aber immer den Set- oder Reset-Eingang (je nach Implementierung) und nicht den zuletzt aktiven.

Fazit

Der Funktionsblock `ILOCK_SWITCH_AX` bietet eine robuste, adapterbasierte Verriegelung für bidirektionale Steuerungen. Durch die Zustandsautomatik wird stets nur eine Richtung aktiv geschaltet, wobei die Priorität dem zuletzt eingehenden Befehl zufällt. Die saubere Trennung von Ereignis- und Datenkanälen sowie die definierten Stop-Zustände machen ihn zu einem zuverlässigen Baustein in sicherheitskritischen Anwendungen.

ILOCK_SWITCH_PROTECT





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ILOCK_SWITCH_PROTECT** realisiert eine priorisierte Verriegelung (Interlock) zwischen zwei Schaltrichtungen – z. B. Auf/Ab, Vor/Rück oder Rechts/Links. Er besitzt eine konfigurierbare Schutzverzögerung (Dead-Time), die ein sofortiges erneutes Umschalten nach einer Richtungsänderung verhindert. Der Baustein wertet die beiden binären Eingänge aus und gibt nur dann ein Schaltsignal aus, wenn der zuletzt aktive Eingang nach Ablauf der Schutzzeit noch immer ansteht. Dies verhindert kurzzeitige Pendelzustände und schützt angeschlossene Aktoren.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Mit Parametern	Beschreibung
EI_UP	`DI_UP`, `DT_PROTECT`	Ereignis zum Anfordern der Richtung „Auf“
EI_DOWN	`DI_DOWN`, `DT_PROTECT`	Ereignis zum Anfordern der Richtung „Ab“

Ereignis-Ausgänge

Name	Mit Parametern	Beschreibung
EO_UP	`DO_UP`	Bestätigung, dass die Richtung „Auf“ aktiviert wurde
EO_DOWN	`DO_DOWN`	Bestätigung, dass die Richtung „Ab“ aktiviert wurde

Daten-Eingänge

Name	Datentyp	Initialwert	Beschreibung
DI_UP	BOOL	–	TRUE = vorwärts, auf, rechts, im Uhrzeigersinn
DI_DOWN	BOOL	–	TRUE = rückwärts, ab, links, gegen Uhrzeigersinn
DT_PROTECT	TIME	T#50ms	Schutzverzögerung (Totzeit) vor einem Richtungswechsel

Daten-Ausgänge

Name	Datentyp	Beschreibung
DO_UP	BOOL	TRUE = Ausgang für Richtung „Auf“ aktiv
DO_DOWN	BOOL	TRUE = Ausgang für Richtung „Ab“ aktiv

Adapter

Name	Typ	Beschreibung
timeOut	`iec61499::events::ATimeOut`	Adapter zur Realisierung der Schutzverzögerung

Funktionsweise

Der Baustein arbeitet als endlicher Automat mit fünf Zuständen:

1. **STOP** – Ruhezustand. Beide Ausgänge sind FALSE.
   - Bei einem Ereignis mit gültigem Eingang (`EI_UP[DI_UP]` oder `EI_DOWN[DI_DOWN]`) wird direkt in den entsprechenden Zustand geschaltet.
2. **UP** – Richtung „Auf“ ist aktiv. `DO_UP = TRUE`, `DO_DOWN = FALSE`.
   - Ein erneutes `EI_UP` mit inaktiver Anforderung (`NOT DI_UP`) oder ein `EI_DOWN` mit aktiver Anforderung (`DI_DOWN`) führt in den Schutz-Zustand.
3. **DOWN** – Richtung „Ab“ ist aktiv. `DO_UP = FALSE`, `DO_DOWN = TRUE`.
   - Analog: `EI_DOWN[NOT DI_DOWN]` oder `EI_UP[DI_UP]` leitet den Schutz ein.
4. **PROTECT** – Schutz-Zustand. Beide Ausgänge werden sofort zurückgesetzt (FALSE) und der Timer des Adapters `timeOut` wird gestartet.
   - Erst nach Ablauf der konfigurierten Zeit (`DT_PROTECT`) wird in den Evaluierungs-Zustand gewechselt.
5. **EVAL** – Evaluierung nach der Schutzzeit.
   - Anhand der aktuellen Eingänge wird entschieden:
     - `DI_UP AND NOT DI_DOWN` ⟶ **UP**
     - `DI_DOWN AND NOT DI_UP` ⟶ **DOWN**
     - `NOT DI_UP AND NOT DI_DOWN` ⟶ **STOP**
     - `DI_UP AND DI_DOWN` (beide gleichzeitig aktiv) ⟶ erneut **PROTECT** (Ungültigkeits-Fall)

Die Auslösung der Ereignisausgänge erfolgt zusammen mit dem jeweils ausgeführten Algorithmus. In den Zuständen UP, DOWN und PROTECT werden **beide** Ereignisausgänge (EO_UP und EO_DOWN) nacheinander gesendet – dies ist ein spezifisches Verhalten dieses Bausteins.

Technische Besonderheiten

- **Adapterbasierte Timer-Implementierung**: Die Schutzverzögerung wird über den standardisierten IEC 61499-Adapter `ATimeOut` realisiert. Dadurch ist die Zeitsteuerung plattformunabhängig und kann in verschiedenen Laufzeitumgebungen genutzt werden.
- **Voreinstellung der Totzeit**: Die Zeit `DT_PROTECT` wird in den Algorithmen der Zustände UP, DOWN und STOP gesetzt, bevor der Timer ggf. gestartet wird. So ist sichergestellt, dass auch bei wiederholtem Wechsel in den Schutz-Zustand stets die aktuelle Zeitkonfiguration gilt.
- **Sperre bei gleichzeitigen Anforderungen**: Wenn beide Eingänge (`DI_UP` und `DI_DOWN`) gleichzeitig TRUE sind, verharrt der Baustein im Schutz-Zustand, bis die Anforderung aufgelöst ist.

Zustandsübersicht

Zustand	Beschreibung	DO_UP	DO_DOWN	Ausgelöste Ereignisausgänge
STOP	Warten auf gültige Anforderung; Ausgänge inaktiv	FALSE	FALSE	–

UP	Richtung Auf aktiv; Umschalten blockiert ohne Zwischenschritt	TRUE	FALSE	EO_UP, EO_DOWN
DOWN	Richtung Ab aktiv; Umschalten blockiert ohne Zwischenschritt	FALSE	TRUE	EO_DOWN, EO_UP
PROTECT	Schutzverzögerung läuft; alle Ausgänge deaktiviert	FALSE	FALSE	EO_UP, EO_DOWN, timeOut.START
EVAL	Nach Ablauf der Schutzzeit werden die Eingänge ausgewertet	–	–	– (Zustandswechsel)

Anwendungsszenarien

- **Antriebssteuerung**: Verhindert schnelle Richtungswechsel bei Motoren, die eine mechanische oder elektrische Totzeit benötigen (z. B. Ventilatoren, Förderbänder, Schiebetore).
- **Ventil- oder Klappenansteuerung**: Stellt sicher, dass vor einem Richtungswechsel (Öffnen/Schließen) eine definierte Pause eingehalten wird, um Druckschläge oder Kavitation zu vermeiden.
- **Sicherheitsgerichtete Verriegelung**: Der Baustein kann verwendet werden, um konkurrierende Steuersignale zu entprellen und nur den zuletzt anstehenden Befehl nach einer festgelegten Haltezeit zu übernehmen.

Vergleich mit ähnlichen Bausteinen

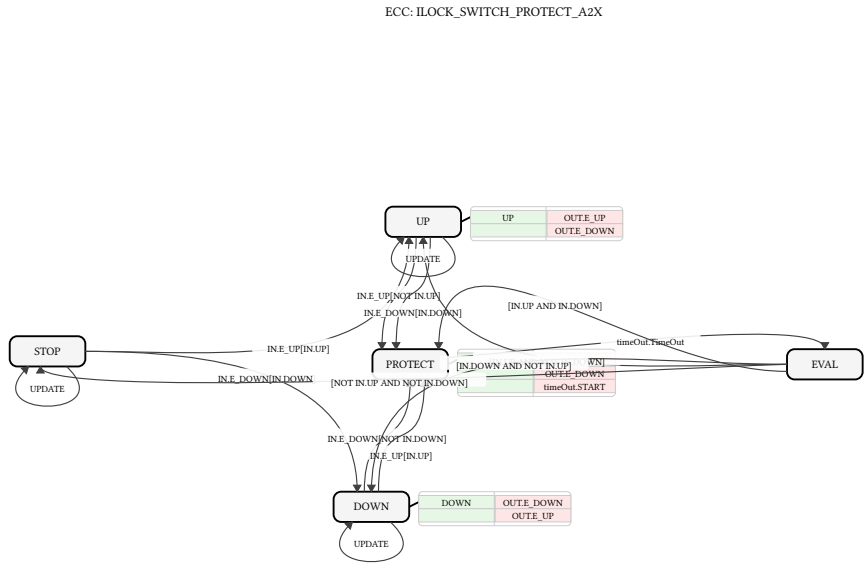
Baustein	Besonderheit
SR-Latch	Einfaches Set-Reset-Flipflop ohne Verzögerung
Interlock-Baustein (einfach)	Schaltet sofort um, keine Totzeit
ILOCK_SWITCH_PROTECT	Konfigurierbare Schutzverzögerung, Auswertung nach der Verzögerung, Behandlung gleichzeitiger Anforderungen

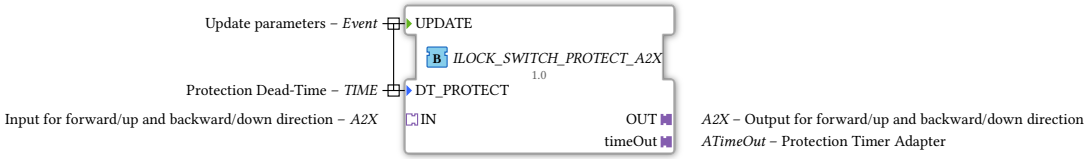
Gegenüber einfachen Verriegelungen bietet **ILOCK_SWITCH_PROTECT** eine kontrollierte Umschaltung, die insbesondere bei induktiven Lasten oder mechanisch trägen Systemen Vorteile bringt.

Fazit

Der Funktionsblock **ILOCK_SWITCH_PROTECT** ist ein robuster IEC 61499-Baustein für Anwendungen, die eine zeitverzögerte, priorisierte Umschaltung zwischen zwei Zuständen erfordern. Durch die einstellbare Totzeit, die Adapter-basierte Timer-Implementierung und die klare Zustandslogik eignet er sich besonders für industrielle Steuerungen, bei denen Zuverlässigkeit und Schutzmechanismen im Vordergrund stehen. Die gleichzeitige Aktivierung beider Ereignisausgänge in den Aktionszuständen sollte bei der Integration beachtet werden.

ILOCK_SWITCH_PROTECT_A2X





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `ILOCK_SWITCH_PROTECT_A2X` realisiert eine Interlock-Schutzfunktion für die Ansteuerung einer bidirektionalen Bewegung (z. B. Auf/Ab, Vorwärts/Rückwärts). Er priorisiert den zuletzt aktiven Eingang und schaltet erst nach Ablauf einer konfigurierbaren Schutz-Totzeit um. Dadurch werden mechanische oder elektrische Schäden durch schnelles Richtungswechseln verhindert. Die Besonderheit dieser Variante liegt in der Adapter-basierten Schnittstelle: Sowohl der Eingang als auch der Ausgang verwenden den unidirektionalen A2X-Adapter, der Ereignisse und Daten für beide Richtungen bündelt.

Schnittstellenstruktur

Ereignis-Eingänge

- **UPDATE** – Über dieses Ereignis werden die Parameter (insbesondere `DT_PROTECT`) aktualisiert.

Ereignis-Ausgänge

- Keine direkten Ereignis-Ausgänge vorhanden. Die Ausgabe erfolgt ausschließlich über den Adapter `OUT`, der die Ereignisse `E_UP` und `E_DOWN` an die verbundene Logik sendet.

Daten-Eingänge

- **DT_PROTECT** (Typ: TIME, Initialwert: `T#50ms`) – Legt die Dauer der Schutz-Totzeit fest, die vor jedem Richtungswechsel abgewartet wird.

Daten-Ausgänge

- Keine direkten Daten-Ausgänge vorhanden. Die Richtungsdaten (`UP` und `DOWN`) werden über den Adapter `OUT` als boolesche Werte bereitgestellt.

Adapter

- **IN** (Typ: `adapter::types::unidirectional::A2X`) – Eingangsadapter für Ereignisse und Daten der Richtungen (UP/DOWN).
- **OUT** (Typ: `adapter::types::unidirectional::A2X`) – Ausgangsadapter, der die aktive Richtung als Ereignis (`E_UP`/`E_DOWN`) und als Datenwert (`UP`/`DOWN`) weitergibt.
- **timeOut** (Typ: `iec61499::events::ATimeOut`) – Adapter zur Steuerung des internen Schutz-Timers. Er wird beim Eintritt in den Schutzzustand gestartet und signalisiert über das Ereignis `TimeOut` das Ende der Totzeit.

Funktionsweise

Der Funktionsblock arbeitet als endlicher Zustandsautomat (ECC). Die Eingangssignale kommen über den `IN`-Adapter: Ereignisse `E_UP` bzw. `E_DOWN` lösen Richtungsanforderungen aus, und die zugehörigen Daten `UP` bzw. `DOWN` geben den aktuellen Zustand der Eingänge an.

Im Normalbetrieb (Zustand `STOP`) wartet der FB auf eine Anforderung. Erfolgt eine Anforderung `E_UP` mit aktivem `UP`-Signal, wechselt er in den Zustand `UP`; analog geht eine `DOWN`-Anforderung in den Zustand `DOWN`. In diesen Zuständen werden die Ausgänge entsprechend gesetzt (`OUT.UP := TRUE` bzw. `OUT.DOWN := TRUE`) und die jeweiligen Ereignisse am `OUT`-Adapter ausgelöst.

Sobald während der aktiven Richtung ein Signalwechsel erkannt wird – zum Beispiel ein `E_UP`-Ereignis, obwohl `UP` nicht mehr aktiv ist, oder ein konkurrierendes `E_DOWN`-Ereignis – wechselt der FB in den Zustand `PROTECT`. Dort setzt er alle Ausgänge auf inaktiv (`HALT`) und startet den Schutz-Timer über den `timeOut`-Adapter. Nach Ablauf der Totzeit (`timeOut.TimeOut`) geht er in den Zustand `EVAL`. In `EVAL` wird anhand der aktuellen Eingangsdaten entschieden:

- Ist nur `UP` aktiv ⟶ wechselt in `UP`
- Ist nur `DOWN` aktiv ⟶ wechselt in `DOWN`
- Sind beide inaktiv ⟶ zurück in `STOP`
- Sind beide aktiv (unerlaubter Zustand) ⟶ erneut in `PROTECT`, was die Totzeit erneut startet.

Während aller Zustände kann das Ereignis `UPDATE` den Parameter `DT_PROTECT` aktualisieren; der Zustand selbst bleibt erhalten.

Technische Besonderheiten

- **Adapter-basierte Ein-/Ausgabe:** Alle Signale werden über die A2X-Adapter gebündelt, was die Verdrahtung und Wiederverwendung vereinfacht.
- **Integrierter Schutz-Timer:** Der ATimeOut-Adapter übernimmt die zeitliche Steuerung der Totzeit, ohne dass zusätzliche Timer-Bausteine erforderlich sind.
- **Priorisierung des letzten aktiven Eingangs:** Die Logik wertet nach Ablauf der Totzeit die zuletzt gültigen Eingangsdaten aus und schaltet entsprechend um.
- **Unerlaubter Eingangszustand:** Wenn sowohl `UP` als auch `DOWN` aktiv sind, wird der aktive Zustand nicht übernommen; stattdessen verharrt der FB in der Schutzschleife, bis sich die Eingangssituation klärt.
- **Selbsthaltender Schutz:** Der Zustand `PROTECT` wird erst nach Ablauf der Totzeit verlassen, wodurch kurze Signalglitches sicher ignoriert werden.

Zustandsübersicht

Die folgende Tabelle fasst die Zustände und ihre Bedeutung zusammen:

Zustand	Beschreibung
`STOP`	Ausgang inaktiv; wartet auf erste Richtungsanforderung.
`UP`	Ausgang aktiviert die Richtung „UP“ (Ereignis `E_UP` und Datenup `UP=TRUE`).
`DOWN`	Ausgang aktiviert die Richtung „DOWN“ (Ereignis `E_DOWN` und Datendown `DOWN=TRUE`).
`PROTECT`	Ausgang inaktiv; Schutz-Totzeit wird durch den Timer abgewartet.
`EVAL`	Auswertung der Eingangsdaten nach der Totzeit; führt je nach Bedingung zu `UP`, `DOWN`, `STOP` oder erneutem `PROTECT`.

Die Transitionen sind im ECC definiert und reagieren auf die Ereignisse `IN.E_UP`, `IN.E_DOWN`, `timeOut.TimeOut` und die Datenbedingungen `IN.UP`, `IN.DOWN` sowie auf das Parameterupdate `UPDATE`.

Anwendungsszenarien

Typische Einsatzbereiche sind:

- Steuerung von Motoren oder Aktuatoren, die in zwei Richtungen betrieben werden (z. B. Hub-/Senkmechanismen, Förderbänder, Verstellantriebe).
- Maschinen mit Sicherheitsanforderungen, bei denen ein sofortiges Umschalten der Bewegungsrichtung vermieden werden muss (z. B. Schutz gegen Überschwingen oder mechanische Überlastung).
- Automatisierungssysteme, die eine klare Priorisierung der letzten Bedienanforderung benötigen und gleichzeitig eine Schutzzeit einhalten.
- Adapter-Version, wenn die Ein-/Ausgänge bereits über A2X-Busse angebunden sind (z. B. in logiBUS-Umgebungen).

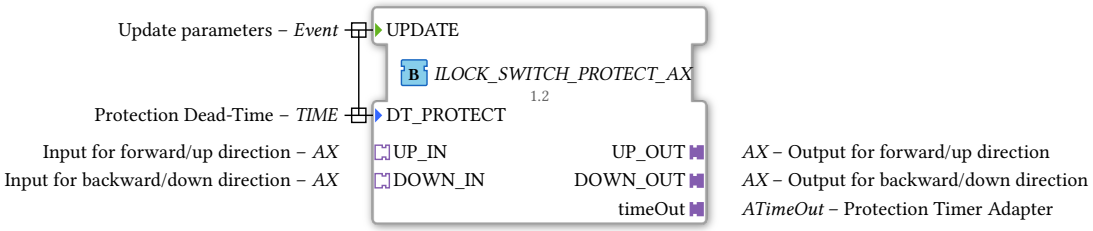
Vergleich mit ähnlichen Bausteinen

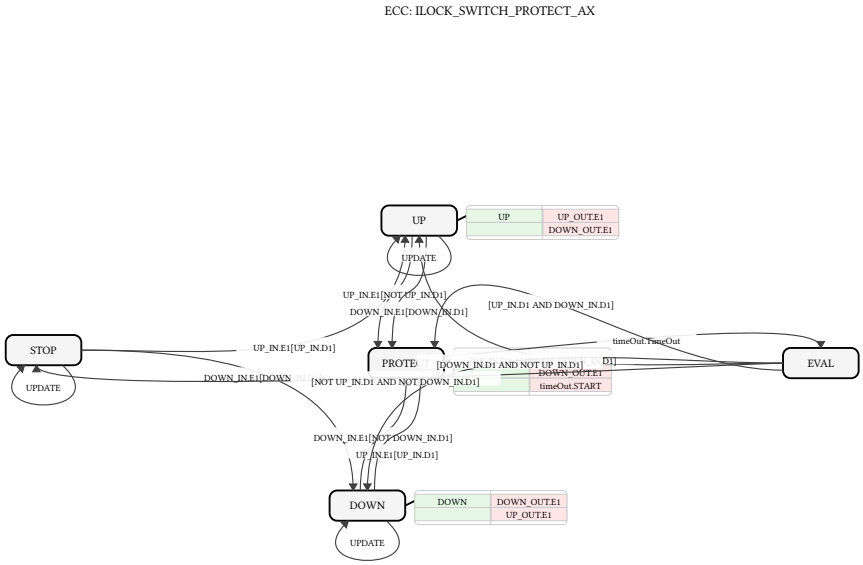
Im Gegensatz zu einfachen Interlock-Bausteinen, die ohne Totzeit arbeiten und eventbasierte Eingänge direkt übernehmen, bietet dieser FB eine explizite Schutzzeit, die unerwünschte Wechsel unterdrückt. Gegenüber einer Variante mit getrennten Ereignis-Daten-Eingängen (ohne Adapter) vereinfacht die A2X-Adapterschnittstelle die Integration in bestehende Busstrukturen und reduziert die Anzahl der Verbindungen. Andere Bausteine könnten die Totzeit extern über einen eigenen Timer realisieren, während dieser FB den Timer integriert hat und dadurch die Verdrahtung weiter minimiert.

Fazit

`ILOCK_SWITCH_PROTECT_A2X` ist ein spezialisierter Funktionsblock für sichere Richtungssteuerung mit Schutz-Totzeit. Er kombiniert eine klare Zustandslogik mit einer kompakten Adapter-Schnittstelle und integriertem Timer. Durch die Priorisierung des letzten aktiven Eingangs und die eingebaute Schutzschleife eignet er sich besonders für Anwendungen, bei denen ein schneller Richtungswechsel vermieden werden muss. Die Adapter-basierte Architektur sorgt für eine flexible und wiederverwendbare Einbindung in industrielle Automatisierungsumgebungen.

ILOCK_SWITCH_PROTECT_AX





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `ILOCK_SWITCH_PROTECT_AX` dient zur verriegelten Umschaltung zwischen zwei Eingangskanälen (UP/DOWN) mit einer einstellbaren Schutz-Totzeit. Er priorisiert den zuletzt aktiven Eingang und verhindert durch die Totzeit ein unkontrolliertes Hin- und Herschalten (Prellen) bei gleichzeitigen oder schnell wechselnden Anforderungen. Die Schnittstelle erfolgt über standardisierte unidirektionale Adapter (Typ `AX`) sowie einen Timer-Adapter.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung
`UPDATE`	Parametrierungsereignis; übernimmt den aktuellen Wert von `DT_PROTECT` in den Timer.

Ereignis-Ausgänge

Keine direkten Ereignisausgänge. Die Ausgabe erfolgt implizit über die Adapter:

- `UP_OUT.E1` – Ereignisausgang des Up-Adapters
- `DOWN_OUT.E1` – Ereignisausgang des Down-Adapters
- `timeOut.TimeOut` – Ereignis des Timer-Adapters (interne Verwendung)

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung
`DT_PROTECT`	`TIME`	`T#50ms`	Schutz-Totzeit, die nach einem Zustandswechsel aktiviert wird.

Daten-Ausgänge

Keine direkten Datenausgänge. Die Zustandsausgabe erfolgt über die Adapter:

- `UP_OUT.D1` – Boolescher Wert für den Up-Ausgang (`TRUE` wenn aktiv)
- `DOWN_OUT.D1` – Boolescher Wert für den Down-Ausgang (`TRUE` wenn aktiv)

Adapter

Richtung	Name	Typ	Beschreibung
Socket	UP_IN	adapter::types::unidirectional::AX	Eingang für die Anforderung „Aufwärts/Vorwärts“. Enthält Ereignis E1 und Bool-Daten D1.
Socket	DOWN_IN	adapter::types::unidirectional::AX	Eingang für die Anforderung „Abwärts/Rückwärts“.
Plug	UP_OUT	adapter::types::unidirectional::AX	Ausgang für die Richtung Aufwärts.
Plug	DOWN_OUT	adapter::types::unidirectional::AX	Ausgang für die Richtung Abwärts.
Plug	timeOut	iec61499::events::ATimeOut	Timer-Adapter zur Realisierung der Totzeit. Steuert das Ereignis TimeOut nach Ablauf von DT_PROTECT.

Funktionsweise

Der Baustein realisiert eine **verriegelte Zwei-Kanal-Schaltung mit Totzeit**:

- Die beiden Eingänge UP_IN und DOWN_IN können nie gleichzeitig aktiv ausgegeben werden.
- Sobald ein Eingang aktiv wird (d.h. sein Ereignis E1 tritt ein und das zugehörige Datum D1 ist TRUE), schaltet der Baustein nach einer kurzen Totzeit in den entsprechenden Zustand (UP oder DOWN).
- Erfolgt ein Richtungswechsel, wird zuerst der **Schutzzustand PROTECT** eingenommen, in dem alle Ausgänge auf FALSE gesetzt und der Timer mit DT_PROTECT gestartet wird. Nach Ablauf der Totzeit wird in den Zustand EVAL gewechselt, der anhand der aktuellen Eingangssignale entscheidet, welcher Endzustand eingenommen wird.
- Durch die Totzeit wird ein Übersprechen oder Prellen der Signale unterbunden, z. B. bei mechanischen Schaltern oder sich überlappenden Anforderungen.

Technische Besonderheiten

- **Adapter-basierte Schnittstelle**: Der FB kommuniziert ausschließlich über standardisierte Adapter (AX für boolesche Steuerung, ATimeOut für Timer). Dies ermöglicht eine einfache Wiederverwendung in verschiedenen Container-Bausteinen.
- **Integrierte Schutzlogik**: Im Zustand PROTECT werden alle Ausgänge deaktiviert, bevor eine neue Richtung aktiviert wird – dies schützt angeschlossene Hardware (z. B. Motoren, Ventile) vor Kurzschluss oder gegenseitigem Blockieren.
- **Parameteraktualisierung**: Das Ereignis UPDATE erlaubt eine dynamische Anpassung der Totzeit zur Laufzeit, ohne den Baustein resetten zu müssen.
- **Totzeit bei jedem Zustandswechsel**: Jeder Übergang (Stop⟶Up, Down⟶Up usw.) durchläuft immer die Schutzphase, unabhängig von der vorherigen Richtung.

Zustandsübersicht

Zustand	Beschreibung
STOP	Ruhezustand. Keine Ausgabe aktiv. Wartet auf Aktivierung eines Eingangs.
UP	Ausgang UP_OUT.D1 = TRUE, DOWN_OUT.D1 = FALSE. Wird aktiv, wenn UP_IN.D1 anliegt und DOWN_IN.D1 inaktiv ist.
DOWN	Ausgang DOWN_OUT.D1 = TRUE, UP_OUT.D1 = FALSE. Wird aktiv, wenn DOWN_IN.D1 anliegt und UP_IN.D1 inaktiv ist.
PROTECT	Schutzphase. Beide Ausgänge werden sofort auf FALSE gesetzt, der Timer wird gestartet. Nach Ablauf von DT_PROTECT wird in EVAL gewechselt.
EVAL	Auswertungszustand. Basierend auf den aktuellen Eingangsdaten wird entschieden, welcher Endzustand (STOP, UP, DOWN) oder erneut PROTECT (bei gleichzeitigen Anforderungen) eingenommen wird.

Die in der ECC definierten Transitionen sind:

- STOP ⟶ UP (bei UP_IN.E1 und UP_IN.D1)
- STOP ⟶ DOWN (bei DOWN_IN.E1 und DOWN_IN.D1)
- UP ⟶ PROTECT (bei UP_IN.E1 und NOT UP_IN.D1 ODER DOWN_IN.E1 und DOWN_IN.D1)
- DOWN ⟶ PROTECT (analog)
- PROTECT ⟶ EVAL (nach timeOut.TimeOut)
- EVAL ⟶ UP (bei UP_IN.D1 AND NOT DOWN_IN.D1)
- EVAL ⟶ DOWN (bei DOWN_IN.D1 AND NOT UP_IN.D1)
- EVAL ⟶ STOP (bei NOT UP_IN.D1 AND NOT DOWN_IN.D1)
- EVAL ⟶ PROTECT (bei UP_IN.D1 AND DOWN_IN.D1)
- In jedem Zustand kann ein UPDATE-Ereignis den Zustand neu bewerten und die Totzeit aktualisieren.

Anwendungsszenarien

- **Verriegelung von Antrieben**: Beispielsweise bei einem Hubtisch oder einem Förderband, das nur in eine Richtung gleichzeitig aktiv sein darf.
- **Schutz vor Kurzschluss** in H-Brücken-Schaltungen: Bevor die Polarität eines Motors umgeschaltet wird, werden beide Ausgänge kurzzeitig deaktiviert.
- **Schaltende Sicherheitslogik** in der Landtechnik oder Automatisierung, wo mechanische Endschalter oder Näherungssensoren eine zeitliche Entprellung benötigen.
- **Schnittstelle zwischen zwei Steuerungssystemen**: Wenn beide Systeme eine gemeinsame Achse ansteuern, verhindert der Baustein Kollisionen durch die Priorisierung des zuletzt aktiven Eingangs.

Vergleich mit ähnlichen Bausteinen

- **Einfaches RS-Flipflop**: Bietet keine Totzeit und schaltet sofort – kann bei sich überlappenden Signalen oszillieren.
- **SWITCH (ohne Schutz)**: Standard-Umschalter schaltet ebenfalls ohne Verzögerung und ohne Entprellung.
- **SR-Latch mit Timer**: Ähnliche Funktion, jedoch meist als Composite-Baustein realisiert. `ILOCK_SWITCH_PROTECT_AX` kapselt die gesamte Logik inklusive Timer und Auswertung in einem Basic Function Block.
- **Interlock-Bausteine in IEC 61499**: Viele Bibliotheken bieten Interlock-FBs, jedoch oft ohne Adapter-Schnittstelle oder mit festen Verzögerungen. Dieser Baustein ist speziell für den Einsatz mit unidirektionalen `AX`-Adaptern ausgelegt.

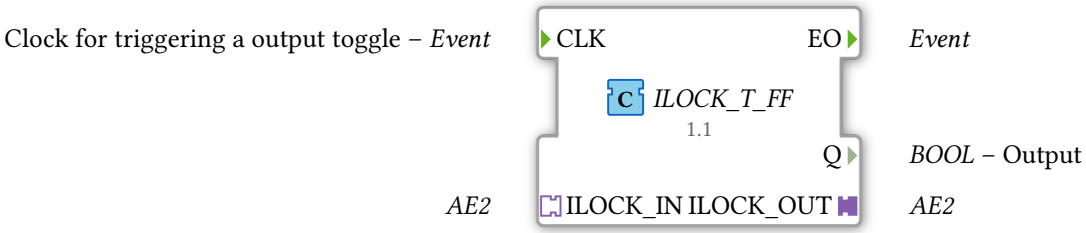
Fazit

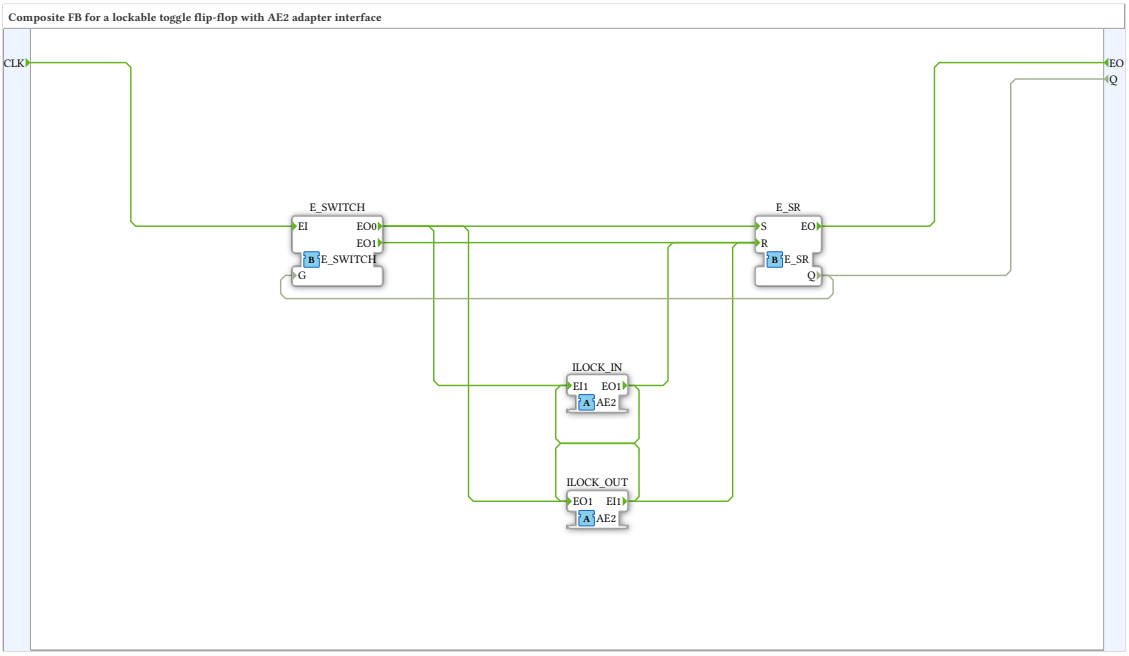
`ILOCK_SWITCH_PROTECT_AX` ist ein robuster und flexibler Baustein zur verriegelten Umschaltung mit einstellbarer Schutz-Totzeit. Die Adapter-basierte Schnittstelle ermöglicht eine saubere Integration in modulare 4diac-Architekturen. Die klare Zustandsmaschine gewährleistet deterministisches Verhalten und bietet Schutz vor Fehlschaltungen in zeitkritischen Anwendungen. Ideal für Anwendungen in der Agrartechnik, Automatisierung und Antriebstechnik, bei denen ein sicherer Richtungswechsel erforderlich ist.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ILOCK_T_FF





• – ○ • – ○ • – ○ •

Einleitung

Der **ILOCK_T_FF** ist ein zusammengesetzter Funktionsblock (Composite FB), der einen sperrbaren Toggle-Flipflop (T-Flipflop) realisiert. Er erweitert ein einfaches T-Flipflop um eine bidirektionale Interlock-Schnittstelle, die das Verketten mehrerer Bausteine zu einer Verriegelungskette erlaubt. Der Baustein eignet sich vor allem für sicherheitsgerichtete Steuerungen, bei denen einmal geschaltete Zustände blockiert und an nachfolgende Glieder weitergegeben werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung
CLK	Ereignis	Triggersignal zum Umschalten des Ausgangs (Toggle).

Ereignis-Ausgänge

Name	Typ	Beschreibung
EO	Ereignis	Wird ausgelöst, sobald der interne Set-/Reset-Latch seinen Zustand geändert hat.

Daten-Eingänge

Keine Dateneingänge (die Steuerung erfolgt rein über Ereignisse und die Adapter).

Daten-Ausgänge

Name	Typ	Beschreibung
Q	BOOL	Aktueller Zustand des Flipflops (0 oder 1).

Adapter

Name	Typ	Richtung	Beschreibung
ILOCK_IN	`adapter::types::bidirectional::AE2`	Sockel (Socket)	Empfängt und sendet Lock-Signale von bzw. an vorgeschaltete Bausteine in der Kette.

ILOCK_OUT	`Adapter::types::bidirectional::AE2`	Stecker (Plug)	Sendet und empfängt Lock-Signale von bzw. an nachgeschaltete Bausteine.

Funktionsweise

Der Baustein besteht intern aus einem **E_SWITCH** (Ereignisverzweiger) und einem **E_SR** (Set-Reset-Latch):

1. **Normalbetrieb (kein Lock)**
   - Ein Ereignis am **CLK** wird an den **E_SWITCH** weitergeleitet.
   - Der Ausgang **Q** des internen Latches dient als Steuersignal **G** für den **E_SWITCH**:
     - Ist **Q** = **FALSE**, schaltet der **E_SWITCH** das Ereignis auf **E_SR.S** (Set) – **Q** wird **TRUE**.
     - Ist **Q** = **TRUE**, schaltet der **E_SWITCH** das Ereignis auf **E_SR.R** (Reset) – **Q** wird **FALSE**.
   - Dadurch wird bei jedem **CLK** ein sauberer Toggle (Flankenwechsel) ausgeführt.
2. **Lock-Mechanismus (Interlock)**
   - Über die Adapter **ILOCK_IN** und **ILOCK_OUT** können Lock-Ereignisse in die Kette eingespeist werden.
   - Ein Lock-Ereignis (z.B. von einem vorgeschalteten Baustein über **ILOCK_IN.EO1** oder von einem nachgeschalteten über **ILOCK_OUT.EI1**) wird auf den **Reset** (**R**) des Latches gelegt.
   - Dadurch wird **Q** sofort zurückgesetzt (**FALSE**). Der Toggle-Vorgang wird unterbrochen, weil der **E_SWITCH** nun im Zustand **Q** = **FALSE** verharrt und bei nächstem **CLK** versuchen würde zu setzen – doch das Lock-Signal kann erneut wirken.
   - **Weiterleitung des Locks**: Jedes eingehende Lock-Ereignis wird gleichzeitig an den jeweils anderen Adapterausgang weitergegeben:
     - **ILOCK_IN.EO1** → **ILOCK_OUT.EO1** (an nachfolgende Bausteine)
     - **ILOCK_OUT.EI1** → **ILOCK_IN.EI1** (an vorherige Bausteine)
   - Dadurch breitet sich ein Lock in beide Richtungen aus (**Multi-Hop Chain Propagation**).
3. **Ergebnis**
   - Solange kein Lock anliegt, arbeitet der Baustein wie ein normaler T-Flipflop.
   - Sobald ein Lock aktiv wird, erzwingt er ein **Q** = **FALSE** und blockiert weitere Toggle-Versuche, bis der Lock wieder aufgehoben wird.

Technische Besonderheiten

- **Bidirektionale Lock-Propagation**: Der Baustein kann Lock-Signale sowohl aufnehmen als auch weitergeben, sodass eine Daisy-Chain mehrerer ILOCK_T_FF möglich ist.
- **Rückkopplung des eigenen Zustands**: Das Ausgangssignal **Q** wird intern als Steuersignal für den **E_SWITCH** verwendet und bestimmt die Toggle-Richtung.
- **Kein eigenes ECC**: Die gesamte Logik wird durch die beiden enthaltenen Basisbausteine (E_SR, E_SWITCH) abgebildet.

Zustandsübersicht

Da der Baustein kein eigenes Zustandsdiagramm besitzt, ergeben sich die Zustände aus dem internen **E_SR**:

Zustand **Q**	Bedeutung im Normalbetrieb	Bedeutung bei aktivem Lock
0	Beim nächsten CLK wird auf 1 getoggelt.	Lock hat Q auf 0 gesetzt und hält es dort.
1	Beim nächsten CLK wird auf 0 getoggelt.	Lock setzt Q sofort auf 0 zurück.

Ein aktiver Lock liegt vor, sobald eines der Adapter-Ereignisse (ILOCK_IN.EO1 oder ILOCK_OUT.EI1) empfangen wurde. Der Lock wird solange gehalten, bis kein Lock-Ereignis mehr anliegt (die genaue Auflösung hängt von der externen Signalsteuerung ab).

Anwendungsszenarien

- **Sicherheitsverriegelungen in der Landtechnik** (z.B. Steuerung von Anbaugeräten): Ein einmal aktivierter Sperrzustand blockiert alle folgenden Schaltvorgänge und wird über die Kette an alle weiteren Komponenten weitergegeben.

- **Verkettete Not-Aus-Systeme**: Erlaubt das Fortpflanzen eines Stopp-Signals entlang mehrerer Stationen, während jede Station selbst den Zustand verwaltet.
- **Protokolladaption für Interlock-Signale**: Der Baustein kann als universelles Glied in einer Verriegelungskette eingesetzt werden, bei der der Ausgang Q einen Schaltvorgang repräsentiert.

Vergleich mit ähnlichen Bausteinen

Baustein	Eigenschaften	Unterschied zum ILOCK_T_FF
T-Flipflop (z.B. `E_TOGGLE`)	Reine Toggle-Funktion ohne Sperrlogik.	Fehlende Lock-/Interlock-Fähigkeit, kein Adapter-Interface.
RS-Flipflop (z.B. `E_SR`)	Set-Reset mit zwei Eingängen, kein Toggle.	Kann mit externer Logik zum Toggle erweitert werden, aber ohne integrierte Lock-Propagation.
Einfacher Interlock-Baustein	Nur Lock-Weitergabe, kein Toggle.	Der ILOCK_T_FF kombiniert Toggle und Lock-Propagation in einem Baustein.

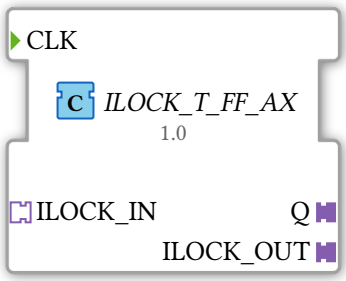
Fazit

Der **ILOCK_T_FF** vereint die Funktionalität eines getakteten T-Flipflops mit einer effizienten, bidirektionalen Interlock-Propagation. Durch die integrierte Multi-Hop-Weiterleitung eignet er sich ideal für vernetzte Sicherheitsketten, in denen ein Schaltzustand blockiert und gleichzeitig an die gesamte Kette weitergegeben werden muss. Sein kompakter Aufbau als Composite FB macht ihn leicht verständlich und wiederverwendbar.

ILOCK_T_FF_AX

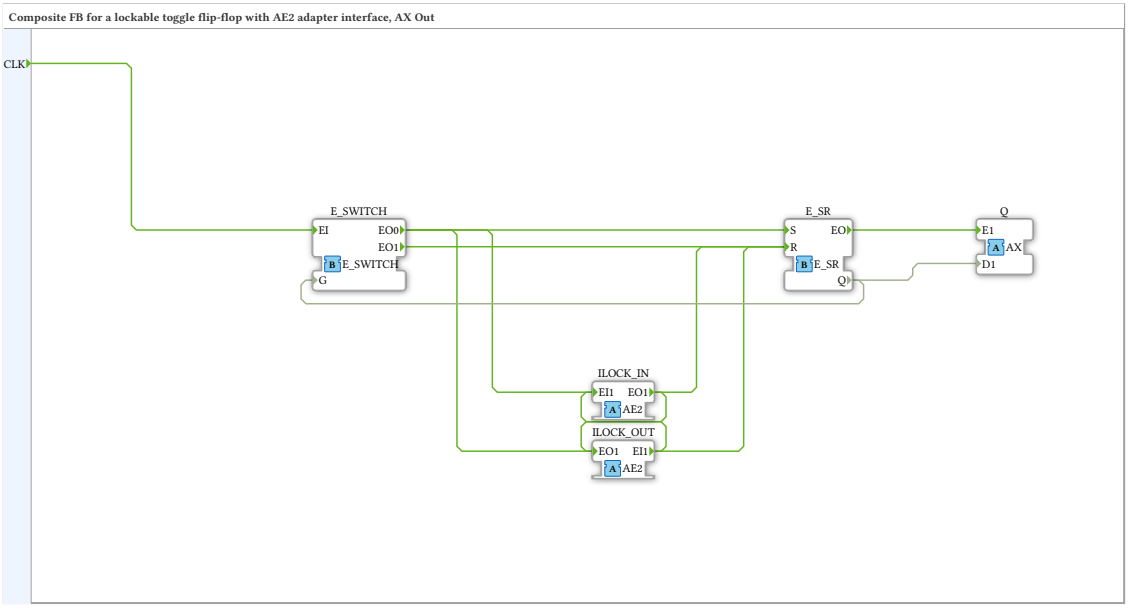
Clock for triggering a output toggle – *Event*

AE2



AX – Output

AE2



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `ILOCK_T_FF_AX` ist ein zusammengesetzter (Composite) Baustein, der einen verriegelbaren Toggle-Flip-Flop (engl. lockable toggle flip-flop) realisiert. Er verfügt über eine AE2-Adapter-Schnittstelle zur Anbindung an eine übergeordnete Verriegelungslogik und gibt den aktuellen Schaltzustand über einen unidirektionalen AX-Adapter aus. Der Baustein ist für den Einsatz in sicherheitsgerichteten oder verriegelten Steuerungsanwendungen konzipiert, bei denen das Umschalten eines Ausgangs durch externe Bedingungen beeinflusst werden muss.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung
`CLK`	Event	Taktsignal, das einen Zustandswechsel (Toggle) auslöst.

Ereignis-Ausgänge

Der Baustein besitzt keine eigenen Ereignisausgänge. Ereignisse werden über die Adapter-Schnittstellen weitergegeben.

Daten-Eingänge

Keine direkten Dateneingänge.

Daten-Ausgänge

Keine direkten Datenausgänge. Der aktuelle Zustand wird über den AX-Adapter bereitgestellt.

Adapter

Adapter	Typ	Richtung	Beschreibung
`Q`	`adapter::types::unidirectional::AX`	Plug/Output	Unidirektionaler Ausgangsadapter, der den aktuellen Zustand (boolescher Wert) als Datum liefert und ein Ereignis bei Zustandsänderung ausgibt.
`ILOCK_IN`	`adapter::types::bidirectional::AE2`	Socket	Bidirektionaler Adapter (Eingang) zur Aufnahme von Verriegelungssignalen.
`ILOCK_OUT`	`adapter::types::bidirectional::AE2`	Plug	Bidirektionaler Adapter (Ausgang) zur Weitergabe von Verriegelungsereignissen an die übergeordnete Logik.

Die AE2-Adapter besitzen je zwei Ereignispaare (`EI1/EO1`, `EI2/EO2`) und zwei zugehörige Datenports. In diesem Baustein werden nur die ersten Ports (`EI1/EO1`) genutzt.

Funktionsweise

Der interne Aufbau besteht aus zwei IEC 61499-Standardbausteinen: `E_SR` (Set-Reset-Flip-Flop) und `E_SWITCH` (Ereignisweiche). Das Toggle-Verhalten wird wie folgt realisiert:

1. Ein ankommendes `CLK`-Ereignis wird der Ereignisweiche `E_SWITCH` zugeführt.
2. Der Schalteingang `G` der Weiche ist mit dem Ausgang `Q` des SR-Flip-Flops verbunden.
   - Ist `G = false`, wird das Ereignis an den Ausgang `EO0` (Set-Pfad) weitergeleitet.
   - Ist `G = true`, wird das Ereignis an den Ausgang `EO1` (Reset-Pfad) weitergeleitet.
3. Der Set-Pfad (`EO0`) setzt den SR-Flip-Flop (`E_SR.S`) und erzeugt gleichzeitig ein Ereignis auf `ILOCK_OUT.EO1` sowie `ILOCK_IN.EI1`, um die Verriegelung über die Adapter zu informieren.
4. Der Reset-Pfad (`EO1`) setzt den SR-Flip-Flop zurück (`E_SR.R`). Dieser Reset kann auch extern über `ILOCK_IN.EO1` oder `ILOCK_OUT.EI1` ausgelöst werden, wodurch ein **verriegelter Reset** möglich ist.
5. Der Ausgang des SR-Flip-Flops (`E_SR.Q`) wird auf den AX-Adapter `Q.D1` geschrieben und bei jedem Zustandswechsel wird das Ereignis `Q.E1` ausgelöst.

6. Zusätzlich werden Ereignisse zwischen `ILOCK_IN` und `ILOCK_OUT` in beide Richtungen durchgeschleift, um eine Kommunikation mit benachbarten Bausteinen im selben Verriegelungsverbund zu ermöglichen.

Der Baustein realisiert somit einen flankengetriggerten Toggle-Flip-Flop, der durch externe Verriegelungssignale (über die AE2-Adapter) zurückgesetzt werden kann.

Technische Besonderheiten

- **Composite-Implementierung:** Der Baustein ist als Netzwerk aus Standard-FBs (`E_SR` und `E_SWITCH`) aufgebaut und kann daher leicht angepasst oder in andere Projekte eingebunden werden.
- **Bidirektionale Verriegelungsschnittstelle:** Die AE2-Adapter erlauben sowohl den Empfang als auch die Weitergabe von Verriegelungsereignissen, was eine modulare Kaskadierung mehrerer Bausteine ermöglicht.
- **Unidirektionaler AX-Ausgang:** Der Zustand wird als sauberes boolesches Signal mit Ereignisbereitstellung ausgegeben; kein zusätzlicher Datenkonvertierungsaufwand auf der Empfängerseite.
- **Verwendung nur der ersten AE2-Ports:** Die zweiten Ports der AE2-Adapter bleiben ungenutzt und können bei Bedarf in einer abgeleiteten Variante ergänzt werden.

Zustandsübersicht

Der interne Zustand des Flip-Flops ist binär:

Zustand	Beschreibung
`0` (false)	Ausgang `Q` ist inaktiv. Bei nächstem `CLK` wird der Baustein setzen.
`1` (true)	Ausgang `Q` ist aktiv. Bei nächstem `CLK` wird der Baustein zurücksetzen.

Die Zustandsübergänge erfolgen ausschließlich bei einem `CLK`-Ereignis (Toggle) oder bei einem Verriegelungs-Reset über `ILOCK_IN.EO1` bzw. `ILOCK_OUT.EI1`. Ein gleichzeitiges Setzen und Rücksetzen wird durch die Logik des SR-Flip-Flops aufgelöst (Reset hat Priorität, sofern beide Ereignisse eintreffen).

Anwendungsszenarien

- **Verriegelte Ausgangssteuerung:** In Maschinensteuerungen, bei denen ein Ausgang nur unter bestimmten Sicherheitsbedingungen umgeschaltet werden darf. Die Verriegelungssignale (z. B. von Not-Aus-Kreisen oder Lichtgittern) werden über `ILOCK_IN` eingelesen und unterdrücken den Toggle.
- **Kaskadierte Verriegelungsketten:** Mehrere `ILOCK_T_FF_AX`-Bausteine können über die AE2-Adapter miteinander verbunden werden, um eine abgestufte Verriegelungshierarchie aufzubauen.
- **Taktsynchrone Schaltzustandsverwaltung:** Der Baustein eignet sich für Anwendungen, die einen getakteten Zustandswechsel benötigen, z. B. in Schrittketten oder sequentiellen Ablaufsteuerungen.

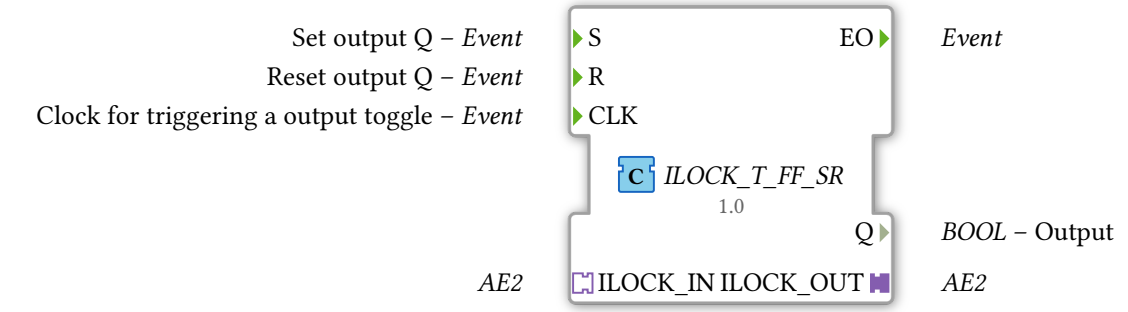
Vergleich mit ähnlichen Bausteinen

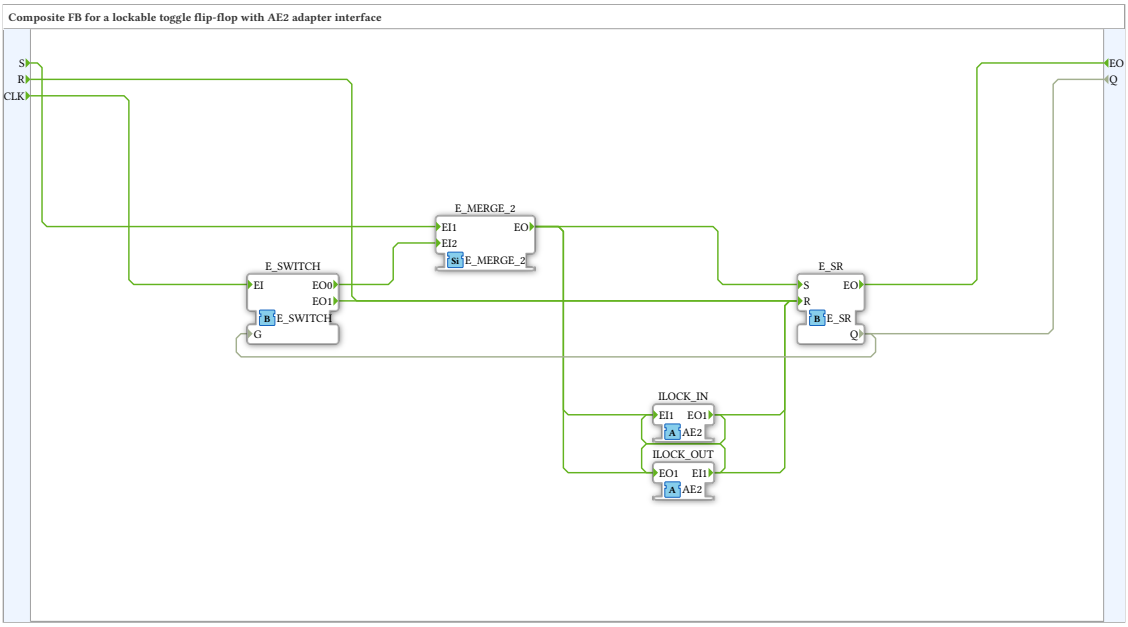
- **Standard Toggle-FF (z. B. eigener IEC-61499-Baustein `E_TOGGLE`):** Ein einfacher Toggle-Flip-Flop hat keine externe Verriegelungsschnittstelle und kann nicht durch externe Bedingungen zurückgesetzt werden. `ILOCK_T_FF_AX` erweitert dieses Grundprinzip um die bidirektionale AE2-Kommunikation.
- **Set-Reset-Flip-Flop (z. B. `E_SR`):** Ein SR-FF hat separate Set- und Reset-Eingänge, aber keinen Toggle-Mechanismus. Der vorliegende Baustein kombiniert Toggle- und SR-Verhalten mit einer Verriegelungslogik.
- **Bausteine mit AX-Ausgang:** Der AX-Adapter ist ein gebräuchlicher Standard für unidirektionale boolesche Ausgänge. Andere Bausteine verwenden oft separate Datenports, während hier eine gekapselte Adapterschnittstelle genutzt wird.

Fazit

`ILOCK_T_FF_AX` bietet eine kompakte, standardkonforme Lösung für einen verriegelbaren Toggle-Flip-Flop in der IEC-61499-Umgebung. Durch die Verwendung von AE2-Adapter-Schnittstellen lässt er sich nahtlos in moderne, modulare Automatisierungsarchitekturen integrieren. Der Composite-Aufbau erleichtert die Wartung und Anpassung, während die klare Trennung von Ereignis- und Datenflüssen die Fehlersuche vereinfacht. Der Baustein ist speziell für sicherheitsgerichtete oder verriegelungsabhängige Schaltaufgaben optimiert und bietet eine solide Grundlage für die Entwicklung komplexerer Steuerungslogiken.

ILOCK_T_FF_SR





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `ILOCK_T_FF_SR` ist ein zusammengesetzter Baustein (Composite FB), der ein verriegelbares Toggle-Flipflop mit Set- und Reset-Funktionalität realisiert. Er besitzt eine Adapter-Schnittstelle (AE2), die eine bidirektionale Kommunikation mit anderen Instanzen ermöglicht und zur Interlock-Verkettung dient. Das Flipflop schaltet seinen Ausgang `Q` bei jedem steigenden Flanke des Clock-Eingangs `CLK` um, sofern kein Reset- oder Set-Ereignis anliegt. Die Verriegelung erfolgt über die Adapter, die von externen Bausteinen gesetzt werden können.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung
`S`	Event	Setzt den Ausgang `Q` auf TRUE
`R`	Event	Setzt den Ausgang `Q` auf FALSE
`CLK`	Event	Clock-Ereignis – löst einen Toggle des Ausgangs `Q` aus

Ereignis-Ausgänge

Name	Typ	Beschreibung
`EO`	Event	Wird nach jeder Änderung des internen Zustands (Set, Reset oder Toggle) ausgelöst

Daten-Eingänge

Keine.

Daten-Ausgänge

Name	Typ	Beschreibung
Q	BOOL	Aktueller Zustand des Flipflops (TRUE = gesetzt, FALSE = zurückgesetzt)

Adapter

Name	Typ	Richtung	Beschreibung
ILOCK_IN	AE2 (bidirektional)	Socket (Eingang)	Ermöglicht den Empfang von Verriegelungssignalen von vorgeschalteten Bausteinen
ILOCK_OUT	AE2 (bidirektional)	Plug (Ausgang)	Sendet Verriegelungssignale an nachgeschaltete Bausteine

Die Adapter ILOCK_IN und ILOCK_OUT sind Typ AE2 (allgemeiner, bidirektionaler Adapter). Sie werden verwendet, um mehrere Instanzen des Bausteins zu einer Kaskade zu verbinden, sodass ein gesetzter Zustand eines Bausteins andere Bausteine zurücksetzen kann (Interlock).

Funktionsweise

Der Baustein basiert auf einem internen Netzwerk, das aus drei primitiven Funktionsblöcken besteht: einem E_SR (Set/Reset-Flipflop), einem E_SWITCH (ereignisgesteuerter Umschalter) und einem E_MERGE_2 (Ereignis-Zusammenführer).

- **Setzen (S):** Ein Ereignis an S wird über E_MERGE_2 direkt an den Set-Eingang des E_SR geleitet. Der Ausgang Q wird auf TRUE gesetzt. Gleichzeitig werden die Ereignisse über die Adapter an die verbundenen Bausteine weitergegeben.
- **Rücksetzen (R):** Ein Ereignis an R geht direkt an den Reset-Eingang des E_SR. Der Ausgang Q wird auf FALSE gesetzt. Auch von den Adaptern eingehende Reset-Signale (über ILOCK_IN.EO1 und ILOCK_OUT.EI1) führen zum Rücksetzen.
- **Toggle (CLK):** Ein Ereignis an CLK wird dem E_SWITCH zugeführt. Dessen Steuereingang G ist mit dem aktuellen Ausgang Q verbunden.
  - Ist Q = FALSE, wird das Clock-Ereignis über den Ausgang EO0 an E_MERGE_2.EI2 weitergeleitet und von dort zum Set-Eingang des E_SR. Somit wird Q auf TRUE gesetzt.
  - Ist Q = TRUE, wird das Clock-Ereignis über den Ausgang EO1 direkt an den Reset-Eingang des E_SR gelegt. Q wird auf FALSE gesetzt. Dadurch entsteht das Toggle-Verhalten: Jeder Clock-Impuls kehrt den aktuellen Zustand von Q um.
- **Verriegelung (Interlock):** Die Adapter ILOCK_IN und ILOCK_OUT sind so verschaltet, dass ein eingehendes Ereignis von ILOCK_IN (z.B. von einem vorherigen Baustein) den Reset-Eingang auslöst und gleichzeitig über ILOCK_OUT an den nächsten Baustein weitergereicht wird. So kann eine Kette von Flipflops aufgebaut werden, bei der immer nur einer aktiv sein darf.

Nach jeder Zustandsänderung wird der Ereignisausgang EO ausgelöst.

Technische Besonderheiten

- **Composite FB:** Der Baustein ist als Netzwerk aus IEC 61499-Elementarmodulen realisiert. Dadurch ist das Verhalten transparent und nachvollziehbar.
- **Bidirektionale Adapter:** Die Verwendung des Adaptertyps AE2 ermöglicht eine flexible Kopplung, sowohl als Socket (Eingang) als auch als Plug (Ausgang).
- **Konfliktbehandlung:** Wenn gleichzeitig ein Set- und ein Reset-Ereignis eintreffen (z.B. S und R), hat das Reset-Vorrang, da der Reset-Eingang des E_SR mehrere Quellen hat und priorisiert behandelt wird. Die genaue Priorität ergibt sich aus der E_SR-Implementierung (Standard: Reset hat Vorrang vor Set).
- **Keine Daten-Eingänge:** Der Baustein arbeitet rein ereignisgesteuert; der Zustand wird nur durch Ereignisse verändert.

Zustandsübersicht

Das interne Flipflop `E_SR` kennt die zwei Zustände:

- **Zustand FALSE** (Q = FALSE): Der Ausgang ist zurückgesetzt. Ein Clock-Ereignis führt zum Setzen.
- **Zustand TRUE** (Q = TRUE): Der Ausgang ist gesetzt. Ein Clock-Ereignis führt zum Rücksetzen.

Ein Set-Ereignis (`S`) führt unabhängig vom aktuellen Zustand immer in den Zustand TRUE.
Ein Reset-Ereignis (`R`) führt unabhängig vom aktuellen Zustand immer in den Zustand FALSE.

Aktueller Zustand	Ereignis	Neuer Zustand	Ausgangsaktionen
FALSE	`S`	TRUE	`EO` wird ausgelöst
FALSE	`R`	FALSE	`EO` wird ausgelöst
FALSE	`CLK`	TRUE	`EO` wird ausgelöst
TRUE	`S`	TRUE	`EO` wird ausgelöst
TRUE	`R`	FALSE	`EO` wird ausgelöst
TRUE	`CLK`	FALSE	`EO` wird ausgelöst

Die Adapter-Ereignisse wirken wie ein externer Reset und setzen den Zustand auf FALSE.

Anwendungsszenarien

- **Interlock-Steuerung in Sicherheitskreisen:** Mehrere Maschinen oder Antriebe sollen sich gegenseitig verriegeln. Nur eine darf zu einem Zeitpunkt aktiv sein. Durch Verkettung mehrerer `ILOCK_T_FF_SR` kann sichergestellt werden, dass stets nur ein Flipflop gesetzt ist.
- **Zustandsmaschinen mit wechselnden Modi:** Der Baustein eignet sich als Toggle-Flipflop, das durch externe Set/Reset-Signale übersteuert werden kann, z.B. für manuelle Eingriffe.
- **Kaskadierte Schaltungen mit Rückmeldung:** Die Adapter-Schnittstelle ermöglicht das einfache Einbinden in bestehende AE2-basierte Kommunikationsstrukturen.

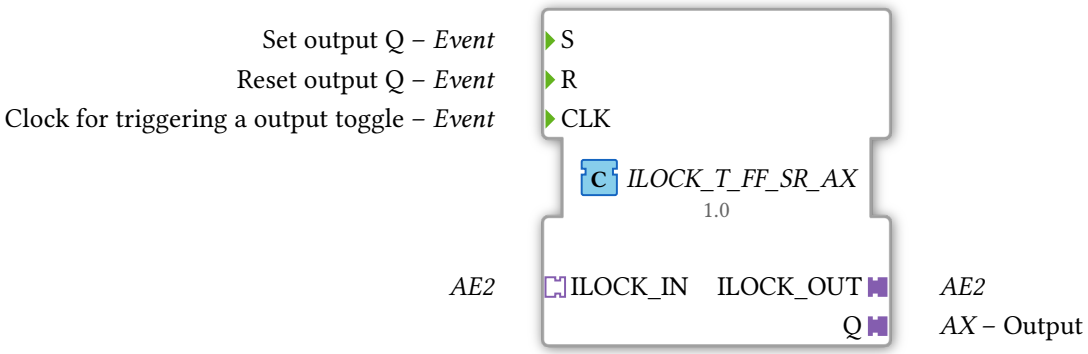
Vergleich mit ähnlichen Bausteinen

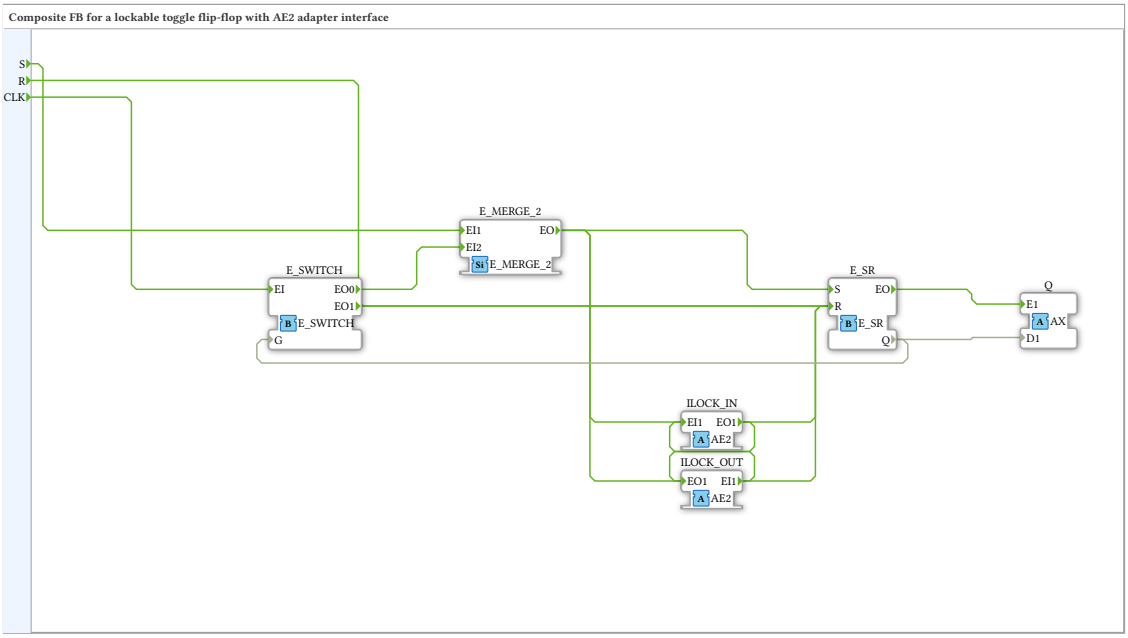
- **Standard T-FF (IEC 61499 `E_T_FF`):** Bietet nur Toggle-Funktion ohne separate Set/Reset-Eingänge. Der `ILOCK_T_FF_SR` erweitert dies um die Set- und Reset-Steuerung.
- **Standard SR-FF (z.B. `E_SR`):** Hat nur Set und Reset, kein Toggle. Der vorliegende Baustein kombiniert Toggle mit SR.
- **Baustein mit Verriegelung (Interlock):** Viele herstellerspezifische Blöcke realisieren Interlock über Busse; dieser Baustein nutzt standardisierte Adapter (AE2) und ist dadurch portabler.

Fazit

Der `ILOCK_T_FF_SR` ist ein vielseitiger Funktionsblock, der ein Toggle-Flipflop mit Set/Reset und einer Interlock-Schnittstelle vereint. Die klare Trennung zwischen Logik und Adapter-Anbindung erlaubt den Aufbau sicherer, kaskadierbarer Schaltungen. Durch die Verwendung von IEC 61499-Standardbausteinen bleibt das Verhalten nachvollziehbar und erweiterbar.

ILOCK_T_FF_SR_AX





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **ILOCK_T_FF_SR_AX** ist ein zusammengesetzter Baustein (Composite FB) für ein verriegelbares Toggle-Flipflop mit Set-Reset-Funktionalität und einer AE2-Adapter-Schnittstelle. Er ermöglicht das gezielte Setzen, Rücksetzen und Takten eines Ausgangssignals unter Berücksichtigung von Verriegelungszuständen, die über bidirektionale Adapter eingelesen und ausgegeben werden.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
S	Event	Setzt den Ausgang Q (sofern nicht verriegelt)
R	Event	Setzt den Ausgang Q zurück
CLK	Event	Takt-Eingang zum Umschalten (Toggle) des Ausgangs

Ereignis-Ausgänge

Der Baustein besitzt keine expliziten Ereignis-Ausgänge. Die Ausgabe erfolgt über Adapter.

Daten-Eingänge

Es sind keine direkten Dateneingänge vorhanden. Die Verriegelungsinformationen werden über die Adapter `ILOCK_IN` und `ILOCK_OUT` ausgetauscht.

Daten-Ausgänge

Name	Typ	Kommentar
Q	Adapter AX (unidirektional)	Ausgangssignal des internen SR-Flipflops

Adapter

Richtung	Name	Typ	Kommentar
Socket	ILOCK_IN	AE2 (bidirektional)	Empfängt Verriegelungssignale von externen Bausteinen

Plug	ILOCK_OUT	AE2 (bidirektional)	Sendet Verriegelungssignale an externe Bausteine
Plug	Q	AX (unidirektional)	Gibt den aktuellen Zustand des Flipflops aus

Funktionsweise

Der Baustein realisiert ein SR-Flipflop mit Toggle-Funktion und Verriegelungsmechanismus.

- **Setzen (S)**: Ein Ereignis an `S` bewirkt über eine Zusammenführung (E_MERGE_2) ein Setzen des internen SR-Flipflops `E_SR`. Der Ausgang Q wird auf `true` gesetzt, sofern keine verriegelnde Rücksetzung von `ILOCK_IN` oder `ILOCK_OUT` vorliegt.
- **Rücksetzen (R)**: Ein Ereignis an `R` setzt den Ausgang Q auf `false`. Zusätzlich können sowohl der Eingangsadapter `ILOCK_IN` als auch der Ausgangsadapter `ILOCK_OUT` ebenfalls ein Rücksetzen auslösen – dies dient der Verriegelung (Interlock).
- **Toggle (CLK)**: Bei jedem Takt-Ereignis an `CLK` wird der aktuelle Zustand von Q ausgewertet. Über einen Schalter (E_SWITCH) wird:
  - Bei Q = `false` ein Setz-Impuls erzeugt (Q ⟶ `true`).
  - Bei Q = `true` ein Rücksetz-Impuls erzeugt (Q ⟶ `false`). Somit wird der Ausgang bei jedem CLK-Ereignis umgeschaltet, sofern keine Verriegelung dies unterbindet.

Die Verriegelung erfolgt über das bidirektionale AE2-Protokoll: Die Signale von `ILOCK_IN` und `ILOCK_OUT` werden gegenseitig durchgeschliffen und können sowohl das interne Flipflop rücksetzen als auch untereinander ausgetauscht werden. Dadurch lässt sich eine sichere Sperre (z. B. gegenseitiger Ausschluss) realisieren.

Technische Besonderheiten

- **Composite-Architektur**: Der FB besteht aus den eingebetteten Funktionen `E_SR`, `E_SWITCH` und `E_MERGE_2`, die logisch verknüpft sind.
- **Bidirektionale Verriegelung**: Die Adapter `ILOCK_IN` und `ILOCK_OUT` sind vom Typ AE2, der einen bidirektionalen Austausch von Ereignissen und Daten ermöglicht. Dies erlaubt eine verschränkte Verriegelung mit anderen Bausteinen.
- **Toggle nur bei Zustandsänderung**: Der Toggle-Mechanismus wertet den Ausgang Q aus, sodass nur dann ein Umschalten erfolgt, wenn das Flipflop nicht bereits durch S oder R gesetzt wurde.
- **Keine direkten Dateneingänge**: Die Steuerung erfolgt ausschließlich über Ereignisse und Adapter, was die Integration in ereignisgesteuerte Systeme erleichtert.

Zustandsübersicht

Der interne Zustand des Flipflops kann die Werte `false` (0) oder `true` (1) annehmen. Die möglichen Übergänge:

Aktuell	Ereignis	Neuer Zustand	Bedingungen
0	S	1	Keine Verriegelung durch ILOCK
0	R	0	–
0	CLK	1	Keine Verriegelung
0	ILOCK_IN/ILOCK_OUT	0	–
1	S	1	–
1	R	0	Keine Verriegelung (Vorrang von R)
1	CLK	0	Keine Verriegelung
1	ILOCK_IN/ILOCK_OUT	0	–

Anwendungsszenarien

- **Maschinensteuerung**: Schalten eines Ausgangs (z. B. Ventil oder Motor) im Takt, wobei eine übergeordnete Sicherheitssteuerung den Ausgang bei Gefahr sperren kann.
- **Gegenseitige Verriegelung (Interlock)**: Mehrere Bausteine des Typs `ILOCK_T_FF_SR_AX` können über die bidirektionalen Adapter so verschaltet werden, dass immer nur einer seiner Ausgang gesetzt hat (z. B. bei Weichen- oder Antriebssteuerung).

- **Taktgesteuerte Signalgenerierung**: Erzeugen eines periodischen Rechtecksignals durch Anlegen eines Takts an `CLK`, wobei Setzen und Rücksetzen extern eingreifen können.

Vergleich mit ähnlichen Bausteinen

- **Einfaches SR-Flipflop**: Besitzt keine Toggle-Funktion und keine Adapter für Verriegelung.
- **TcFF (Toggle-Flipflop)**: Erlaubt nur das Umschalten bei einem Taktereignis, hat aber keine getrennten Set-/Reset-Eingänge und keine Verriegelungsschnittstelle.
- **Standard-Interlock-Bausteine**: Oft auf binäre Signale beschränkt, während `ILOCK_T_FF_SR_AX` die Kombination aus Toggle, SR und bidirektionaler Verriegelung in einem Baustein vereint.

Fazit

Der Funktionsblock `ILOCK_T_FF_SR_AX` bietet eine flexible und sichere Möglichkeit, einen Ausgang zu steuern, der sowohl getaktet umschalten als auch über separate Set-/Reset-Eingänge sowie über eine erweiterbare Verriegelungsschnittstelle beeinflusst werden kann. Durch die Kombination von SR-Flipflop und Toggle-Logik eignet er sich besonders für sicherheitskritische Anwendungen in der Automatisierungstechnik.

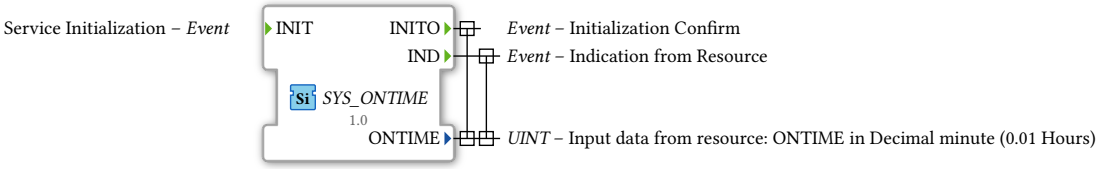
measurement

SYS_ONTIME

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `SYS_ONTIME` ist ein Service Interface Function Block (SIFB). Seine Hauptaufgabe ist es, die Betriebszeit (Ontime) der Ressource, in der er ausgeführt wird, auszulesen und bereitzustellen. Er dient als standardisierte Schnittstelle zwischen der Anwendungslogik und den systemnahen Diensten der Laufzeitumgebung.



Schnittstellenstruktur

Ereignis-Eingänge

- **`INIT`**: Dieses Ereignis initialisiert den Funktionsblock. Es muss ausgelöst werden, bevor der Block seinen Dienst aufnehmen kann.

Ereignis-Ausgänge

- **`INITO`**: Dieses Ereignis bestätigt die erfolgreiche Initialisierung des Funktionsblocks. Es wird zusammen mit dem ersten gültigen `ONTIME`-Wert ausgelöst.
- **`IND`**: Dieses Ereignis signalisiert, dass ein neuer, aktualisierter `ONTIME`-Wert vom Systemressourcen-Manager bereitgestellt wurde.

Daten-Eingänge

- Dieser Funktionsblock besitzt keine Dateneingänge.

Daten-Ausgänge

- **`ONTIME` (UINT)**: Dieser Ausgang liefert die kumulierte Betriebszeit der Ressource. Die Einheit ist „Dezimalminute“, was 0,01 Stunden entspricht (z.B. ein Wert von 150 entspricht 1,5 Stunden).

Adapter

- Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der `SYS_ONTIME`-Block agiert als Client für einen systeminternen Dienst. Nach dem Empfang des `INIT`-Ereignisses registriert er sich beim Ressourcen-Manager, um Zugriff auf die Betriebszeitdaten zu erhalten.

Sobald die Initialisierung abgeschlossen ist, wird das `INITO`-Ereignis zusammen mit dem aktuellen Betriebszeitwert ausgelöst.

Anschließend überwacht der Block kontinuierlich die Systemressource. Sobald sich die Betriebszeit ändert (typischerweise durch einen System-Tick oder einen ähnlichen Mechanismus), löst der Ressourcen-Manager eine Indikation aus. Der `SYS_ONTIME`-Block empfängt diese, aktualisiert seinen internen `ONTIME`-Wert und gibt das `IND`-Ereignis zusammen mit dem neuen Wert an die angeschlossene Anwendungslogik weiter.

Technische Besonderheiten

- **Service Interface FB**: Als SIFB ist seine Implementierung spezifisch für die jeweilige 4diac-Runtime (FORTE) und die zugrunde liegende Hardware/Plattform. Die genaue Art der Zeitmessung (z.B. basierend auf Hochlaufzeit des Prozessors oder der Laufzeitumgebung) ist runtime-definiert.
- **Einheit**: Die Betriebszeit wird in der ungewöhnlichen, aber für Steuerungsaufgaben praktischen Einheit „Dezimalminute“ (1/100 Stunde) geliefert. Dies erleichtert Berechnungen im Dezimalsystem, z.B. für Wartungsintervalle oder Energieverbrauchsschätzungen.
- **Nur Lese-Zugriff**: Der Block bietet ausschließlich Lesezugriff auf einen systeminternen Wert. Ein Zurücksetzen der Betriebszeit ist über diese Schnittstelle nicht möglich.

Zustandsübersicht

1. **Inaktiv**: Der Block ist erstellt, aber noch nicht initialisiert. Keine Kommunikation mit der Systemressource findet statt.
2. **Initialisiert**: Nach dem Empfang von `INIT`. Der Block ist beim Ressourcen-Manager registriert und wartet auf Daten. Der Zustand wechselt sofort nach erfolgreicher Initialisierung, gefolgt von der Ausgabe von `INITO`.
3. **Bereit**: Der Block hat den ersten `ONTIME`-Wert erhalten und ist betriebsbereit. Er wartet nun auf Indikations-Ereignisse (`IND`) vom System, um aktualisierte Werte auszugeben.

Anwendungsszenarien

- **Vorbeugende Wartung (Predictive Maintenance)**: Überwachung der Laufzeit von Maschinen oder Anlagenteilen, um Wartungsintervalle basierend auf tatsächlicher Nutzung auszulösen.
- **Energiemanagement**: Erfassung der Betriebsstunden von Verbrauchern (z.B. Motoren, Heizungen) zur Berechnung des Energieverbrauchs oder für Lastprofile.
- **Betriebsdatenerfassung (BDE)**: Protokollierung der Produktions- oder Maschinenlaufzeiten für Auswertungen zur Overall Equipment Effectiveness (OEE).
- **Lebensdauerüberwachung**: Überprüfung, ob kritische Komponenten ihre spezifizierte Lebensdauer (in Betriebsstunden) erreicht oder überschritten haben.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`E_CYCLE` / `E_DELAY`**: Diese Blöcke erzeugen selbst periodische Ereignisse oder Verzögerungen. `SYS_ONTIME` hingegen *liest* einen passiv vorhandenen, kumulativen Zeitwert aus dem System aus.
- **`E_SR` (Set-Reset) mit Zähler**: Ein manuell aufgebauter Zähler für Einschaltvorgänge könnte ähnliche Daten liefern, wäre aber anwendungsseitig implementiert und nicht systemseitig persistent. `SYS_ONTIME` bietet einen zuverlässigen, systemgeführten und meist auch nach einem Neustart persistenten Wert.
- **Andere `SYS_*`-Blöcke** (wie `SYS_TIME`): Während `SYS_TIME` die aktuelle Uhrzeit und das Datum liefert, stellt `SYS_ONTIME` spezifisch die reine Betriebsdauer einer Ressource bereit.

🛠️ Zugehörige Übungen

- Uebung_140

Fazit

Der `SYS_ONTIME`-Funktionsblock ist ein essentielles Werkzeug für alle Anwendungen, die auf der Erfassung und Auswertung von Betriebszeiten basieren. Als standardisierte Service-Schnittstelle abstrahiert er die plattformspezifische Implementierung und bietet der Anwendungslogik einen einfachen, zuverlässigen Zugriff auf diese wichtige Systemkenngröße. Seine Verwendung ist besonders in den Bereichen Instandhaltung, Energieeffizienz und Betriebsdatenerfassung zu empfehlen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.1.14.10 stations

Computers

Einleitung

Die Globalkonstanten `Computers` stellen eine Sammlung von vordefinierten Netzwerkadressen (IP-Adressen und Ports) als Zeichenketten (`WSTRING`) zur Verfügung. Sie dienen der zentralen Verwaltung von Verbindungsinformationen für verschiedene Rechner oder Stationen innerhalb eines logiBUS-Projekts. Durch die Verwendung dieser Konstanten wird die Wartbarkeit und Lesbarkeit des Codes verbessert, da Änderungen an Netzwerkadressen nur an einer zentralen Stelle vorgenommen werden müssen.

Schnittstellenstruktur

Bei Globalkonstanten handelt es sich nicht um einen Funktionsbaustein im klassischen Sinne, daher existieren keine Ereignis- oder Datenschnittstellen. Stattdessen werden Konstantendeklarationen bereitgestellt, die projektweit zur Verfügung stehen.

Daten-Eingänge

Nicht zutreffend. Konstanten sind feste Werte und besitzen keine Eingänge.

Daten-Ausgänge

Nicht zutreffend. Konstanten werden direkt in Ausdrücke oder Zuweisungen eingebunden.

Adapter

Nicht zutreffend.

Funktionsweise

Die Konstanten werden beim Kompilieren des Projekts definiert. Jede Konstante erhält einen festen, unveränderlichen Wert vom Typ `WSTRING` (Wide String). Sobald das Programm läuft, können diese Konstantennamen (`C_LO`, `C_ET`, etc.) anstelle der hartkodierten Zeichenketten in jedem Teil des Programms verwendet werden, das diese Konstanten importiert oder kennt.

Technische Besonderheiten

- **Typ:** Alle Konstanten sind vom Typ `WSTRING`, der Unicode-Zeichenketten unterstützt.
- **Gültigkeitsbereich:** Als globale Konstanten sind sie innerhalb des deklarierten Pakets (`logiBUS::stations`) verfügbar.
- **Initialisierung:** Die Werte werden direkt bei der Deklaration zugewiesen und sind zur Laufzeit schreibgeschützt.
- **Verwendung:** Sie sind typischerweise für die Konfiguration von Kommunikationsbausteinen (z.B. `PUBLISH`, `SUBSCRIBE`, `CLIENT`, `SERVER`) vorgesehen, wo die Adresse des Kommunikationspartners benötigt wird.

Zustandsübersicht

Nicht zutreffend. Konstanten haben keinen Zustand.

Anwendungsszenarien

1. **Vereinfachte Adresskonfiguration:** In einem verteilten Steuerungssystem mit mehreren Steuerungen (z.B. Hauptsteuerung, Etagensteuerung) können die IP-Adressen der einzelnen Geräte hier zentral hinterlegt werden.
2. **Portabilität zwischen Umgebungen:** Durch das einfache Austauschen der Werte in dieser Konstantendatei kann ein Programm schnell zwischen verschiedenen Netzwerkumgebungen (z.B. Entwicklungslokalhost, Testnetzwerk, Produktionsnetzwerk) angepasst werden.

3. **Vermeidung von Tippfehlern:** Die Verwendung des Konstantennamens `C_ET` ist weniger fehleranfällig als das wiederholte Eintippen der Zeichenkette `"192.168.5.1:61499"`.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`VAR_GLOBAL` (Globale Variablen):** Im Gegensatz zu globalen Variablen können Konstanten (`CONSTANT`) nach der Initialisierung nicht mehr verändert werden. Sie bieten somit höhere Programmsicherheit.
- **`VAR_CONFIG` (Konfigurationsvariablen):** Konfigurationsvariablen sind ebenfalls Konstanten, die jedoch pro Ressource konfiguriert werden können. Die hier definierten globalen Konstanten haben einen festen, projekteinheitlichen Wert.
- **Direkte Wertangabe:** Der direkte Einsatz von Zeichenketten-Literalen im Code ist möglich, führt aber zu schlechterer Wartbarkeit und höherer Fehleranfälligkeit im Vergleich zur Verwendung zentraler Konstanten.

Fazit

Die Globalkonstanten `Computers` sind ein einfaches, aber äußerst nützliches Werkzeug zur Strukturierung und Pflege von Netzwerkverbindungsdaten in 4diac-Projekten. Sie fördern saubere Programmierpraktiken, reduzieren Redundanz und machen den Code anpassungsfähiger gegenüber Änderungen in der Netzwerktopologie. Ihre Verwendung wird insbesondere in größeren oder verteilten Automatisierungsanlagen empfohlen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.1.14.11 storage

Const

storageStatusMessages

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `storageStatusMessages` ist ein globaler Konstantensatz der Bibliothek `logiBUS::storage::const`. Er definiert einheitliche Status- und Fehlermeldungen für Speicherzugriffe, insbesondere für die Initialisierung und die Nutzung von Non-Volatile Storage (NVS). Durch die zentrale Definition dieser Konstanten wird eine konsistente Kommunikation von Speicherzuständen innerhalb von 4diac-Anwendungen ermöglicht. Die Konstanten sind als `STRING`-Werte ausgeführt und können überall im Projekt referenziert werden.

Schnittstellenstruktur

Da es sich um einen GlobalConstants-Baustein handelt, besitzt er keine Ereignis- oder Datenein-/ausgänge und auch keine Adapter. Die definierten Konstanten dienen als globale Datenquelle und sind für alle Funktionsblöcke der Anwendung lesbar.

Ereignis-Eingänge

Keine.

Ereignis-Ausgänge

Keine.

Daten-Eingänge

Keine.

Daten-Ausgänge

Keine.

Adapter

Keine.

Funktionsweise

Der Baustein stellt eine Sammlung von statischen Konstanten bereit, die zur Laufzeit nicht verändert werden können. Sie werden verwendet, um Rückgabewerte von Speicheroperationen zu standardisieren. Die folgende Tabelle listet alle Konstanten, ihre Werte und deren Bedeutung auf:

Konstante	Wert	Bedeutung/Beschreibung
`OK`	`'OK'`	Erfolgreiche Speicherung oder allgemeiner Erfolg.
`NO_CHANGE`	`'Value was not stored. Old is new'`	Der Wert wurde nicht geändert, der alte Wert bleibt aktiv.
`INITIALISED`	`'initialized'`	Speicher wurde erfolgreich initialisiert.
`NOT_INITIALISED`	`'Not initialized'`	Speicher ist noch nicht initialisiert.
`ERR`	`'ERROR'`	Allgemeiner Fehler.
`ERR_NVS_NOT_FOUND`	`'ESP_ERR_NVS_NOT_FOUND'`	Der angeforderten NVS-Schlüssel wurde nicht gefunden.
`ERR_NVS_DEFAULT_SET`	`'Default Value was used'`	Ein Standardwert wurde verwendet, da der eigentliche Wert fehlte.
`ERR_SECTION_EMPTY`	`'SECTION empty'`	Die übergebene Sektion ist leer.
`ERR_KEY_EMPTY`	`'KEY empty'`	Der übergebene Schlüssel ist leer.
`ERR_SECTION_WHITESPACE`	`'SECTION contains whitespace'`	Die Sektion enthält Leerzeichen.
`ERR_KEY_WHITESPACE`	`'KEY contains whitespace'`	Der Schlüssel enthält Leerzeichen.
`ERR_READ_ONLY`	`'Key is read-only'`	Der Schlüssel ist schreibgeschützt und kann nicht geändert werden.

Diese Konstanten werden typischerweise als Rückgabewerte von Funktionsblöcken verwendet, die auf INI- oder NVS-Speicher zugreifen.

Technische Besonderheiten

- Die Konstanten sind im Paket `logiBUS::storage::const` definiert und können über den vollqualifizierten Namen (z. B. `logiBUS::storage::const::storageStatusMessages.OK`) angesprochen werden.
- Sie sind als `VAR_GLOBAL CONSTANT` deklariert, d. h. ihre Werte sind zur Laufzeit unveränderlich.
- Der Baustein wird in der 4diac-IDE als `GlobalConstants`-Typ geführt und kann in Projekten als Referenz auf das Konstantenset eingebunden werden.
- Die Werte sind bewusst als `STRING` gehalten, um eine flexible Ausgabe und Verarbeitung in Anwendungen zu ermöglichen.

Zustandsübersicht

Da es sich um einen Konstantenblock handelt, existieren keine internen Zustände oder Zustandsübergänge. Die Werte sind statisch und immer verfügbar.

Anwendungsszenarien

- **Fehlerbehandlung in Speicherbausteinen**: Funktionsblöcke, die auf NVS oder INI zugreifen, verwenden diese Konstanten, um Statusmeldungen an übergeordnete Logik zu übergeben.
- **Einheitliche Log-Ausgaben**: Die Meldungen können direkt für Diagnose- und Loggingzwecke genutzt werden, da sie sprechende Texte enthalten.
- **Initialisierungskontrolle**: Mit `INITIALISED` und `NOT_INITIALISED` kann der Initialisierungsstatus eines Speicherbereichs geprüft werden.
- **Schreibschutzprüfung**: Der Wert `ERR_READ_ONLY` hilft, Versuche auf schreibgeschützte Schlüssel zu erkennen.

Vergleich mit ähnlichen Bausteinen

In 4diac existieren oft separate konstante Definitionen innerhalb einzelner Funktionsblöcke. Dieser GlobalConstants-Baustein bietet jedoch eine zentrale, wiederverwendbare Sammlung, die eine konsistente Fehler- und Statusbehandlung über viele Speicherfunktionen hinweg ermöglicht. Andere Bausteine könnten eigene Statuscodes definieren, was zu Inkonsistenzen führen kann. `storageStatusMessages` standardisiert diese Codes und reduziert so den Wartungsaufwand.

Fazit

Der GlobalConstants-Baustein `storageStatusMessages` stellt eine klar strukturierte und dokumentierte Sammlung von Statusmeldungen für Speicherzugriffe bereit. Durch die zentrale Definition wird die Anwendungsentwicklung vereinfacht und die Zuverlässigkeit der Fehlerbehandlung erhöht. Er ist ein unverzichtbarer Bestandteil für Projekte, die mit NVS- oder INI-Speicher arbeiten.

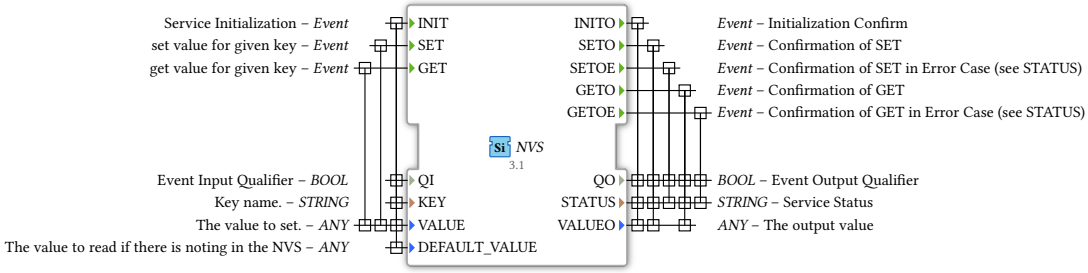
esp32_nvs

NVS

• – ◦ • – ◦ • – ◦ •

Einleitung

Der NVS-Funktionsblock (Non-Volatile Storage) ermöglicht das persistente Speichern und Laden von Daten auf einem ESP32-Mikrocontroller. Er nutzt den nichtflüchtigen Speicher (NVS) des ESP32, um Werte unter einem definierten Schlüssel (Key) abzulegen und bei Bedarf wieder abzurufen. Dieser Baustein ist besonders für Anwendungen geeignet, bei denen Konfigurationsdaten, Kalibrierwerte oder Zustandsinformationen über einen Neustart des Geräts hinweg erhalten bleiben müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisiert den Funktionsblock. Löst den Lesevorgang des mit `KEY` angegebenen Wertes aus. Wenn kein Wert gespeichert ist, wird der `DEFAULT_VALUE` zurückgegeben.
- **SET**: Speichert den an `VALUE` übergebenen Wert unter dem zuvor konfigurierten Schlüssel (`KEY`) im NVS.
- **GET**: Liest den Wert, der unter dem konfigurierten Schlüssel (`KEY`) im NVS gespeichert ist.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Ausführung der Initialisierung (INIT) und liefert das Ergebnis.
- **SETO**: Bestätigt die Ausführung des Schreibvorgangs (SET) und liefert den Status.
- **SETOE**: Bestätigung von SET im Fehlerfall (siehe `STATUS`); führt `QO` und `STATUS` mit.
- **GETO**: Bestätigt die Ausführung des Lesevorgangs (GET) und liefert den gelesenen Wert.
- **GETOE**: Bestätigung von GET im Fehlerfall (siehe `STATUS`); führt `QO` und `STATUS` mit.

Daten-Eingänge

- **QI** (BOOL): Qualifiziert den INIT-Ereigniseingang. Bei `TRUE` wird der Dienst aktiviert, bei `FALSE` deaktiviert.
- **KEY** (STRING): Der eindeutige Schlüsselname, unter dem der Wert im NVS gespeichert oder von dem er gelesen wird. Dieser Wert muss während der INIT-Phase gesetzt sein.
- **VALUE** (ANY): Der zu speichernde Datensatz. Kann einen beliebigen elementaren Datentyp (z.B. BOOL, INT, REAL, STRING) annehmen.
- **DEFAULT_VALUE** (ANY): Der Standardwert, der bei einem Lesevorgang (INIT oder GET) zurückgegeben wird, wenn unter dem angegebenen Schlüssel noch kein Wert im NVS gespeichert ist.

Daten-Ausgänge

- **QO** (BOOL): Gibt den aktuellen Betriebszustand des Funktionsblocks wider (`TRUE` = bereit/erfolgreich, `FALSE` = nicht bereit/Fehler).
- **STATUS** (STRING): Enthält eine Statusmeldung, die den Erfolg oder die Art eines Fehlers bei der letzten Operation beschreibt (z.B. „OK“, „Key not found“, „Storage full“).
- **VALUEO** (ANY): Der tatsächlich gelesene Wert aus dem NVS oder, falls kein Wert vorhanden war, der `DEFAULT_VALUE`.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der NVS-Block arbeitet zustandsbasiert. Bei der Initialisierung (INIT mit `QI=TRUE`) wird der angegebene `KEY` festgelegt. Anschließend wird automatisch versucht, den zugehörigen Wert aus dem NVS zu lesen. Das Ergebnis (gespeicherter Wert oder `DEFAULT_VALUE`) wird zusammen mit dem Status über `INITO` ausgegeben.

Danach kann der Block zwei Hauptoperationen ausführen:

1. **Schreiben (SET)**: Der an `VALUE` anliegende Wert wird unter dem festgelegten Schlüssel (`KEY`) im NVS gespeichert. Die Bestätigung erfolgt über `SETO`.
2. **Lesen (GET)**: Der aktuell unter dem Schlüssel gespeicherte Wert wird aus dem NVS gelesen und über `VALUEO` am Ausgang `GETO` bereitgestellt.

Die Datentypen der Ein- und Ausgänge `VALUE`, `DEFAULT_VALUE` und `VALUEO` müssen konsistent sein. Der Block ist für den wiederholten Zugriff auf denselben Schlüssel ausgelegt.

Technische Besonderheiten

- **ESP32-spezifisch**: Dieser Block nutzt direkt die NVS-API des ESP32-IDF und ist daher nur auf ESP32-basierten Zieltplattformen lauffähig.
- **Typflexibilität**: Die Verwendung des generischen Datentyps `ANY` erlaubt es, verschiedene elementare Datentypen zu speichern. Der konkrete Typ wird bei der ersten Verwendung (INIT) festgelegt und muss bei allen folgenden Operationen beibehalten werden.
- **Persistenz**: Daten im NVS überleben einen Neustart des Mikrocontrollers und bleiben auch nach einem Stromausfall erhalten (je nach Konfiguration des NVS-Speichers).

Zustandsübersicht

1. **Nicht initialisiert**: Der Block ist nach dem Start inaktiv. Der Schlüssel (`KEY`) ist nicht gesetzt.
2. **Initialisiert/Bereit**: Nach erfolgreichem `INIT`-Ereignis ist der Block betriebsbereit. Der Schlüssel ist konfiguriert und ein initialer Wert (aus NVS oder Standard) wurde geladen. Der Block wartet auf `SET`- oder `GET`-Ereignisse.
3. **Schreibvorgang**: Während der Verarbeitung eines `SET`-Ereignisses.
4. **Lesevorgang**: Während der Verarbeitung eines `GET`-Ereignisses oder des impliziten Lesevorgangs während `INIT`.

Anwendungsszenarien

- **Gerätekonfiguration**: Speichern von Netzwerk-Credentials (SSID, Passwort), IP-Adressen oder anderen Parametern, die nach einem Reset erhalten bleiben sollen.
- **Betriebsdatenerfassung**: Persistente Zählung von Betriebsstunden, Zykluszahlen oder produzierten Einheiten.
- **Kalibrierdaten**: Ablegen von sensor- oder actuatorspezifischen Kalibrierwerten.
- **Letzter bekannter Zustand**: Speichern des Systemzustands vor einem unerwarteten Abschalten, um diesen beim nächsten Start wiederherzustellen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **E_R_TRIG / F_TRIG**: Diese sind reine Ereignis-Trigger und bieten keine persistente Datenspeicherung.
- **BLINK / DELAY**: Zeitbasierte Blöcke ohne Speicherfunktionalität.
- **E_EEPROM**: Ein generischer EEPROM-Baustein. Der NVS-Block ist spezifisch für die ESP32-Plattform und nutzt dessen optimiertes, wear-leveling-fähiges Dateisystem (NVS), während ein generischer E_EEPROM-Block hardwareabhängig und oft auf einfacherem Speicher operieren würde.

🛠️ Zugehörige Übungen

- Uebung_012
- NumericValue_ID_TO_NVS
- Uebung_020c2_sub

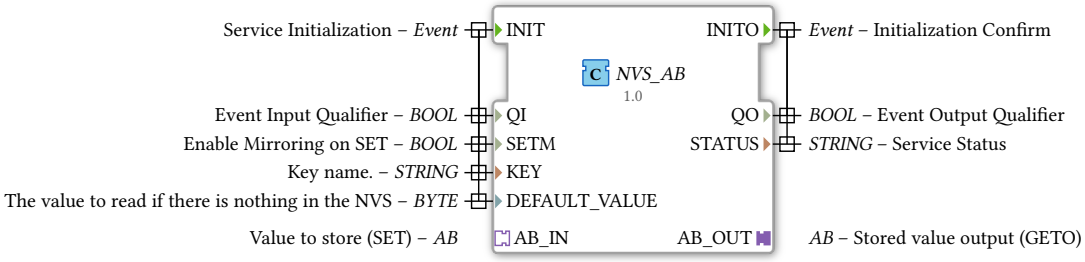
Fazit

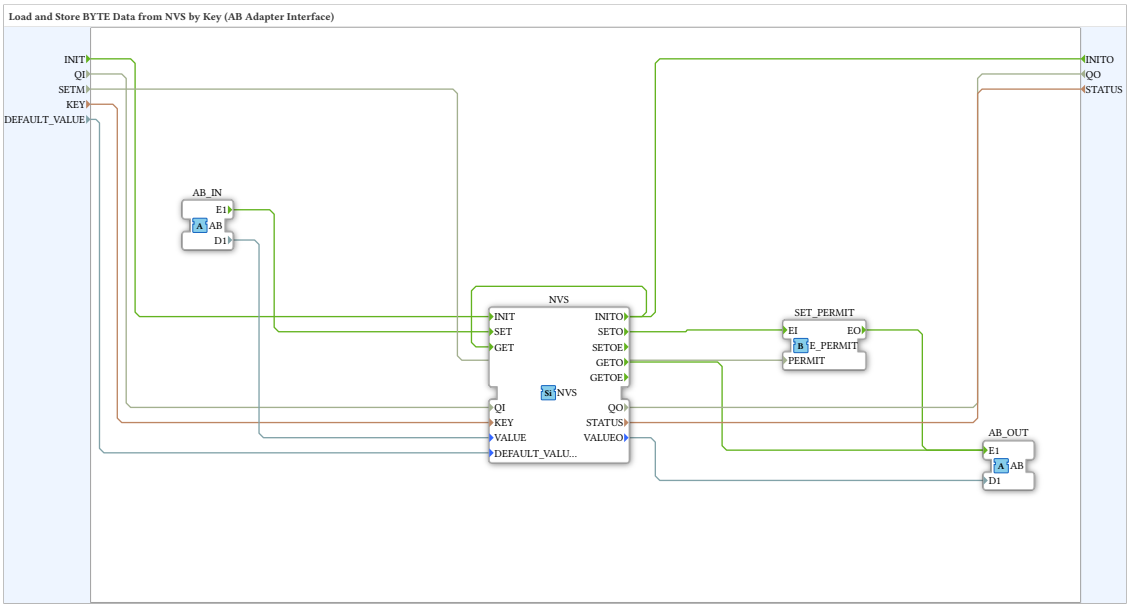
Der NVS-Funktionsblock stellt eine einfache und robuste Schnittstelle zum nichtflüchtigen Speicher des ESP32 bereit. Durch die klare Trennung von Lese- und Schreiboperationen sowie die typflexible Datenverarbeitung eignet er sich ideal für eine Vielzahl von Anwendungen, die persistente Daten auf eingebetteten ESP32-Systemen benötigen. Seine Integration in die 4diac-IDE ermöglicht eine plattformspezifische Funktionalität innerhalb des portablen IEC 61499-Modells.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 ESP32 & ESP32-S3 DevKit auf ms-muc-docs.de
- 🌐 MCU vs. MPU Vergleichsguide auf ms-muc-docs.de

NVS_AB





• – ○ • – ○ • – ○ •

Einleitung

`NVS_AB` lädt und speichert einen `BYTE`-Wert unter einem Schlüssel (`KEY`) im nichtflüchtigen Speicher (Non-Volatile Storage, NVS) des ESP32. Im Unterschied zum bidirektionalen NVS_AB2 verwendet er zwei getrennte unidirektionale `AB`-Adapter: einen Socket zum Schreiben (`AB_IN`) und einen Plug zum Lesen/Bestätigen (`AB_OUT`).

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisiert den Baustein und liest den unter `KEY` hinterlegten Wert (bzw. `DEFAULT_VALUE`, falls noch keiner existiert). Liefert `QI`, `SETM`, `KEY`, `DEFAULT_VALUE`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Initialisierung und den ersten Lesezugriff, liefert `QO` und `STATUS`.

Daten-Eingänge

- **QI** (BOOL): Qualifizierer des `INIT`-Ereignisses.
- **SETM** (BOOL): Gibt das Spiegeln (Rückmeldung) beim Schreiben (`SET`) frei.
- **KEY** (STRING): Name des Schlüssels im NVS.
- **DEFAULT_VALUE** (BYTE): Wert, der geliefert wird, wenn unter `KEY` noch nichts im NVS gespeichert ist.

Daten-Ausgänge

- **QO** (BOOL): Erfolgsqualifizierer, mit `INITO` sowie den internen Adapterbestätigungen mitgeliefert.
- **STATUS** (STRING): Statusmeldung des NVS-Zugriffs.

Adapter

- **AB_IN** (Socket, Typ `AB`): Neuer zu speichernder Wert – ein Adapterereignis auf `AB_IN.E1` löst einen `SET` im internen NVS-Baustein aus, `AB_IN.D1` liefert den zu schreibenden Wert.
- **AB_OUT** (Plug, Typ `AB`): Gibt den zuletzt gelesenen Wert aus – sowohl nach dem automatischen Erstlesen bei `INIT` als auch nach jedem bestätigten Schreibvorgang (`AB_OUT.D1`/`AB_OUT.E1`).

Funktionsweise

`NVS_AB` kapselt intern eine Instanz von `logiBUS::storage::esp32_nvs::NVS` sowie ein `E_PERMIT`-Gatter (`SET_PERMIT`). Bei `INIT` wird die interne NVS-Instanz initialisiert (`NVS.INIT`) und anschließend automatisch ein `GET` ausgelöst, das den unter `KEY` gespeicherten Wert (oder `DEFAULT_VALUE`) liefert; das Ergebnis wird über `AB_OUT` ausgegeben, `INITO` bestätigt den Abschluss.

Zum Schreiben liefert der an `AB_IN` angeschlossene Baustein ein Adapterereignis (`AB_IN.E1`) mit dem neuen Wert (`AB_IN.D1`); dies löst `NVS.SET` aus. Ob die erfolgreiche Speicherung zusätzlich über `AB_OUT` zurückgemeldet wird, steuert `SETM`: Ist `SETM` gesetzt, gibt `SET_PERMIT` das `AB_OUT.E1`-Ereignis nach erfolgreichem `SET` frei („Mirroring“).

Technische Besonderheiten

- Reines Composite-FB ohne eigene ECC – die eigentliche Flash-Logik steckt in der internen `NVS`-Instanz, `NVS_AB` verdrahtet nur Ereignisse und Daten zwischen ihr und den beiden `AB`-Adaptern.
- Zwei getrennte, unidirektionale Adapter (`AB_IN`/`AB_OUT`) statt eines einzigen bidirektionalen – dadurch lassen sich Lese- und Schreibpfad an unterschiedliche Gegenstellen anschließen, was mit NVS_AB2 (ein gemeinsamer `AB2`-Socket) nicht möglich ist.
- `SETM` steuert nur, ob ein Schreibvorgang zusätzlich per Adapterereignis über `AB_OUT` gespiegelt wird – gespeichert wird in jedem Fall.

Zustandsübersicht

Der Baustein ist zustandslos im Sinne einer eigenen ECC: Er durchläuft bei `INIT` einmalig Initialisierung + Erstlesen, danach reagiert er auf jedes `AB_IN`-Adapterereignis mit einem NVS-Schreibzugriff und optional einer Bestätigung über `AB_OUT`.

Anwendungsszenarien

- Persistieren einzelner `BYTE`-Konfigurationswerte (z. B. Betriebsmodus, Kanalauswahl) auf dem ESP32
- Aufbau eines Lese-/Schreibpaars, bei dem Lese- und Schreibseite an unterschiedliche Bausteine angeschlossen werden sollen
- Wiederherstellen eines zuletzt gespeicherten `BYTE`-Werts nach einem Neustart über `AB_OUT`

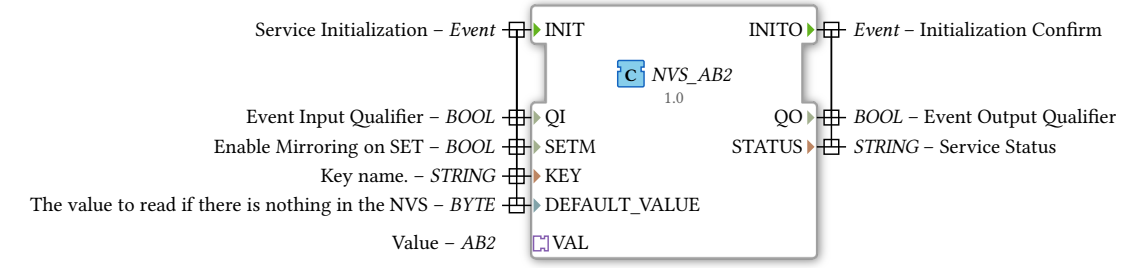
⚖️ Vergleich mit ähnlichen Bausteinen

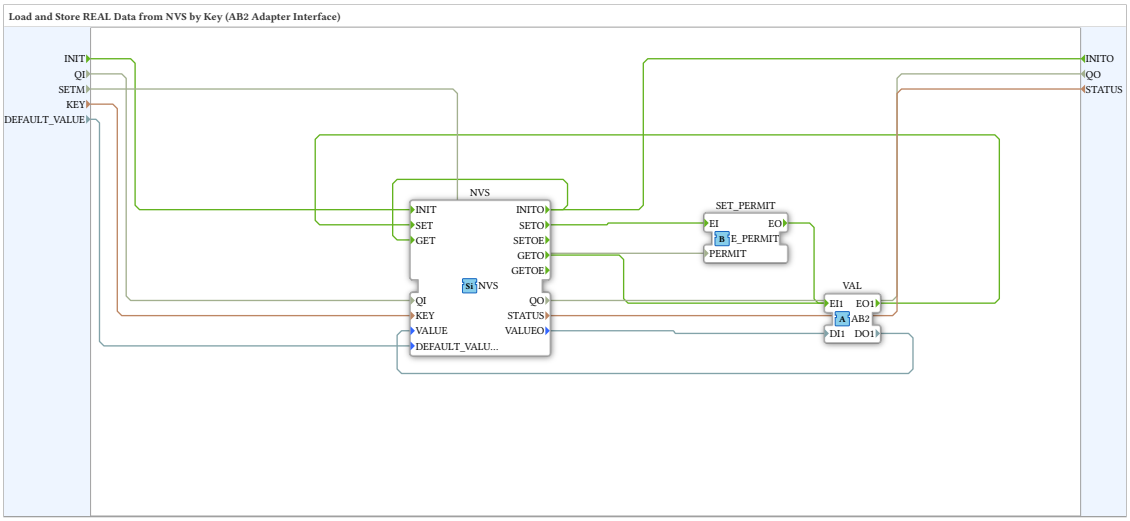
Vergleich mit NVS_AB2, das denselben Zweck über einen einzigen bidirektionalen `AB2`-Adapter statt zweier unidirektionaler `AB`-Adapter erfüllt, sowie mit INI_AB, das dieselbe Schnittstelle für die textbasierte `settings.ini` statt für den NVS-Flash-Speicher bereitstellt.

Fazit

`NVS_AB` bietet einen einfachen, unidirektional verdrahteten Zugriff auf einen persistenten `BYTE`-Wert im ESP32-NVS und eignet sich überall dort, wo Lese- und Schreibpfad getrennt geführt werden sollen.

NVS_AB2





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **NVS_AB2** ermöglicht das Laden und Speichern von `BYTE`-Daten in einem nichtflüchtigen Speicher (Non-Volatile Storage, NVS) unter Verwendung eines Zeichenketten-Schlüssels. Er kapselt den NVS-Zugriff in einer komfortablen, asynchronen Schnittstelle und kommuniziert mit der Umgebung über einen bidirektionalen **AB2**-Adapter. Dadurch eignet sich der Baustein besonders für Szenarien, in denen Konfigurationswerte oder Zustandsdaten dauerhaft auf einem ESP32 gespeichert werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung
`INIT`	EInit	Initialisiert den Baustein. Die Eingangsdaten (`QI`, `SETM`, `KEY`, `DEFAULT_VALUE`) werden beim Eintritt dieses Ereignisses übernommen.

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung

INIT0	EInit	Bestätigung der abgeschlossenen Initialisierung und der ersten Leseoperation.

Daten-Eingänge

Name	Typ	Beschreibung
QI	BOOL	Qualifizierer für das Initialisierungsereignis (z. B. Freigabe der Verarbeitung).
SETM	BOOL	Aktiviert die Spiegelung (Mirroring) bei SET.
KEY	STRING	Name des Schlüssels, unter dem der Wert im NVS abgelegt ist.
DEFAULT_VALUE	BYTE	Wert, der zurückgegeben wird, falls im NVS noch kein Wert für den angegebenen Schlüssel existiert.

Daten-Ausgänge

Name	Typ	Beschreibung
QO	BOOL	Qualifizierer für das Ausgangsereignis – signalisiert Erfolg (TRUE) oder Fehler (FALSE).
STATUS	STRING	Detailstatusmeldung (z. B. Fehlertext oder Bestätigung).

Adapter

Name	Typ (AB2-Adapter)	Beschreibung
VAL	adapter::types::bidirectional::AB2	Bidirektionale Schnittstelle für den Datenwert. Lesen und Schreiben erfolgen über die Adapterkanäle (Events: EI1, EO1; Daten: DI1, DO1).

Funktionsweise

Beim Eintreffen des **INIT**-Ereignisses wird der interne NVS-Baustein initialisiert. Unmittelbar nach erfolgreicher Initialisierung wird automatisch eine **Leseoperation** (GET) gestartet, die den zum Schlüssel KEY gehörenden Wert aus dem NVS ausliest. Falls kein Wert vorhanden ist, wird DEFAULT_VALUE verwendet. Der gelesene Wert wird über den Adapter (VAL.DI1) ausgegeben und das Ereignis INIT0 signalisiert den Abschluss.

Nach der Initialisierung kann der Baustein über den Adapter asynchrone Lese- und Schreibanforderungen bearbeiten:

- **Lesen**: Ein empfangenes Ereignis am Adapter-Port VAL.EO1 löst eine erneute Leseoperation (GET) im NVS aus. Der gelesene Wert wird wieder über VAL.DI1 bereitgestellt, und die Bestätigung (VAL.EI1) wird gesendet.
- **Schreiben**: Über den Adapter wird der zu speichernde Wert auf VAL.DO1 bereitgestellt. Ein Ereignis an VAL.EO1 triggert die **Schreiboperation** (SET) im NVS. Nach erfolgreichem Speichern wird über VAL.EI1 quittiert.

Die Ausgänge QO und STATUS werden bei jedem der Ausgangsereignisse (INIT0, sowie den internen Adapterbestätigungen) auf den aktuellen Status des NVS-Bausteins gesetzt.

Technische Besonderheiten

- **NVS-Zugriff auf dem ESP32**: Der Baustein nutzt den unter logiBUS::storage::esp32_nvs::NVS definierten Funktionsblock, der die Flash-Speicherverwaltung des ESP32 kapselt.
- **Bidirektionale Adapter-Schnittstelle**: Durch den AB2-Adapter kann der Baustein sowohl Lese- als auch Schreibanforderungen aus der Umgebung verarbeiten, ohne dass separate Auslöseereignisse am FB selbst nötig sind.
- **Automatischer Erstlesezugriff**: Nach der Initialisierung wird sofort der gespeicherte Wert gelesen, sodass nach INIT0 der aktuelle Datenwert über den Adapter verfügbar ist.
- **Fehlerbehandlung**: Der Status (STATUS) und der Qualifizierer (QO) geben Aufschluss über Erfolg oder Misserfolg jeder NVS-Operation.

Zustandsübersicht

Der Funktionsblock besitzt keine expliziten Zustandsautomaten, da er als Composite-Baustein das Verhalten des internen NVS-FBs nutzt. Im Wesentlichen lassen sich folgende Phasen unterscheiden:

1. **Initialisierung** – ausgelöst durch `INIT`
2. **Bereit** – nach erfolgreicher Initialisierung wartet der Baustein auf Adapter-Ereignisse
3. **Lese-/Schreiboperation** – kurzzeitiger Zugriff auf den NVS während der Verarbeitung eines Adapter-Ereignisses

Anwendungsszenarien

- **Konfigurationsspeicherung** für Geräteparameter (z. B. Sollwerte, Kalibrierdaten) auf ESP32-basierten Steuerungen.
- **Persistente Zustandssicherung**, z. B. letzter Betriebsmodus oder Zählerstände über einen Neustart hinweg.
- **Datenaustausch über AB2-Protokoll** mit anderen Bausteinen, die das AB2-Adapter-Interface unterstützen – einfache Integration in bestehende Automatisierungslösungen.

Vergleich mit ähnlichen Bausteinen

Gegenüber einfacheren Lese-/Schreib-Bausteinen (z. B. direkte NVS-FBs ohne Adapter) bietet **NVS_AB2**:

- **Asynchrone, ereignisgesteuerte Kommunikation** über den AB2-Adapter – entkoppelt den Datenzugriff vom Hauptsteuerungsfluss.
- **Automatische Initialisierung und Erstlese** reduziert den Programmieraufwand.
- **Einheitliche Fehlersignalisierung** über `QO` und `STATUS`.

Bausteine ohne Adapter erfordern hingegen separate Event-Eingänge für Lesen und Schreiben und bieten keine standardisierte bidirektionale Schnittstelle.

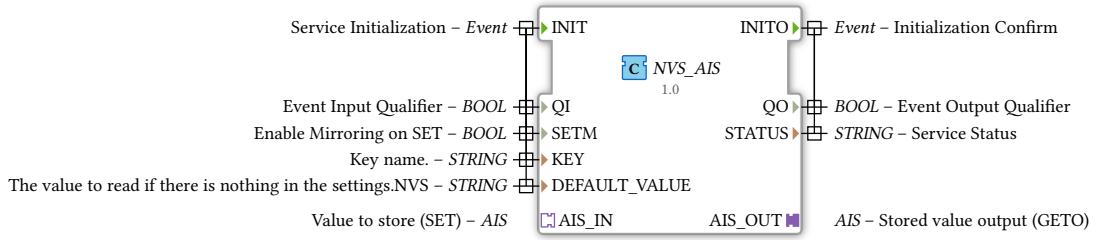
Fazit

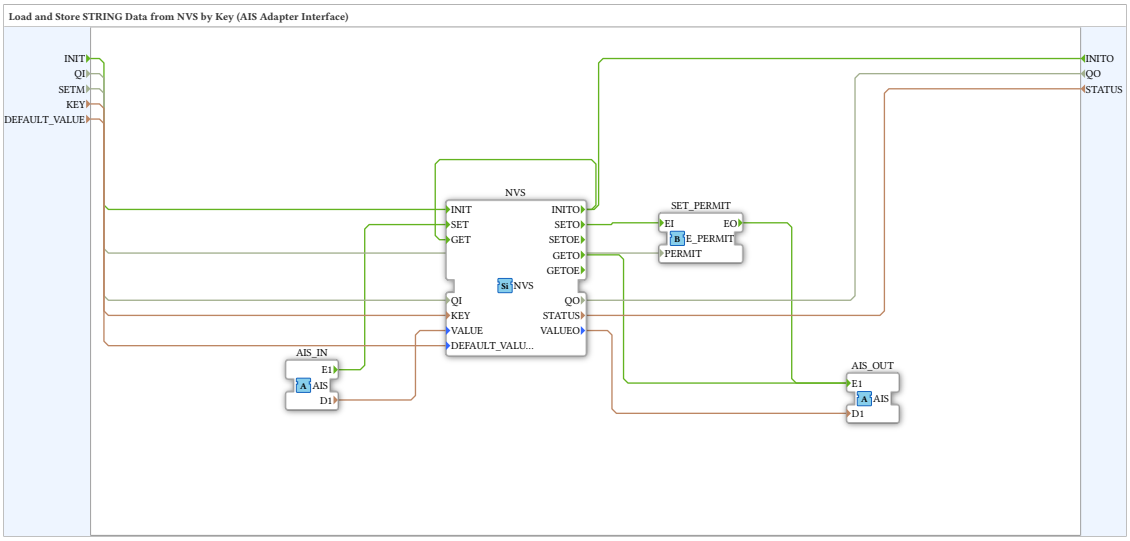
NVS_AB2 ist ein praxisorientierter Funktionsblock für den zuverlässigen, asynchronen Zugriff auf den nichtflüchtigen Speicher eines ESP32. Durch die Kapselung der NVS-Logik in einem Composite-Baustein und die bereitgestellte AB2-Adapter-Schnittstelle lässt er sich flexibel in Automatisierungsprojekte einbinden, die persistente Datenhaltung erfordern. Die Kombination aus automatischer Erstinitialisierung, Lese-/Schreibunterstützung und klarer Statusrückmeldung macht ihn zu einer effizienten Lösung für Embedded-Steuerungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 ESP32 & ESP32-S3 DevKit auf ms-muc-docs.de

NVS_AIS





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `NVS_AIS` dient zum Speichern und Laden von Zeichenketten (STRING) im nichtflüchtigen Speicher (NVS – Non-Volatile Storage) eines ESP32-Mikrocontrollers. Die Daten werden über einen Schlüssel (KEY) adressiert und über AIS-Adapter (Acyclic Information Service) ausgetauscht. Der Baustein kapselt die Initialisierung des NVS sowie die grundlegenden Lese- und Schreiboperationen und stellt eine standardisierte Schnittstelle für die Einbindung in industrielle Steuerungsanwendungen bereit.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung	Mitgeführte Daten
INIT	EInit	**Service Initialization** – Startet den Baustein und die Kommunikation mit dem NVS.	QI, SETM, KEY, DEFAULT_VALUE

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung	Mitgeführte Daten
INITO	EInit	**Initialization Confirm** – Bestätigt die erfolgreiche Initialisierung oder signalisiert einen Fehler.	QO, STATUS

Daten-Eingänge

Name	Datentyp	Beschreibung
QI	BOOL	**Event Input Qualifier** – Steuert das Verhalten während der INIT-Phase (z. B. Aktivierung).
SETM	BOOL	Enable Mirroring on SET: Bei TRUE wird die Bestätigung eines SET (SETO) zusätzlich am Ausgangsadapter ausgegeben (gespiegelt)
KEY	STRING	**Key name** – Der Schlüssel, unter dem der Wert im NVS gespeichert/ abgerufen wird.
DEFAULT_VALUE	STRING	**Default value** – Wert, der gelesen wird, falls im NVS kein Eintrag zum angegebenen KEY existiert.

Daten-Ausgänge

Name	Datentyp	Beschreibung
QO	BOOL	**Event Output Qualifier** – Gibt den Erfolg der letzten Operation an.
STATUS	STRING	**Service Status** – Rückmeldung des NVS-Treiberstatus (z. B. Fehlertexte).

Adapter

Adapter	Typ	Beschreibung
AIS_IN	`adapter::types::unidirectional::AIS` (Socket)	**Value to store (SET)** – Empfängt einen zu speichernden STRING-Wert über das AIS-Protokoll.
AIS_OUT	`adapter::types::unidirectional::AIS` (Plug)	**Stored value output (GETO)** – Sendet den gelesenen STRING-Wert aus dem NVS über das AIS-Protokoll.

Funktionsweise

1. **Initialisierung** Ein Ereignis am `INIT`-Eingang startet die interne Logik. Der Baustein ruft den eingebetteten `NVS`-Funktionsblock auf, der den nichtflüchtigen Speicher vorbereitet. Die mitgelieferten Daten (`QI`, `SETM`, `KEY`, `DEFAULT_VALUE`) werden an den internen Baustein weitergeleitet.
2. **Lesevorgang** Nach erfolgreicher INIT-Phase (Ereignis `INITO` des internen NVS) wird automatisch der `GET`-Ereigniseingang des internen NVS aktiviert. Der gelesene Wert (oder der `DEFAULT_VALUE`, falls kein Eintrag existiert) wird über den `AIS_OUT`-Adapter als `E1`-Ereignis und `D1`-Daten ausgegeben.
3. **Schreibvorgang** Über den `AIS_IN`-Adapter kann ein neuer Wert empfangen werden. Das assoziierte `SET`-Ereignis (E1) wird auf den internen NVS-Baustein geleitet, der den empfangenen Wert (D1) unter dem zuvor gesetzten `KEY` speichert. Die Bestätigung (`SETO`) wird wiederum über den `AIS_OUT`-Adapter gesendet.
4. **Rückmeldung** Sowohl nach dem Lesen als auch nach dem Schreiben gibt der Baustein den Status (`QO`, `STATUS`) sowie das `INITO`-Ereignis aus. Der Status kann für Fehlerdiagnosen genutzt werden.

Technische Besonderheiten

- **NVS für ESP32** – Der Baustein ist speziell für den nichtflüchtigen Speicher des ESP32 ausgelegt und verwendet die entsprechende Treiber-API.
- **Adapter-Schnittstelle** – Die Kommunikation mit der Außenwelt erfolgt ausschließlich über AIS-Adapter (unidirektional). Dies entkoppelt den Baustein von konkreten Bus- oder Applikationsprotokollen.
- **Automatische Leseoperation nach INIT** – Nach der Initialisierung wird unverzüglich ein Lesevorgang gestartet, sodass der gespeicherte Wert sofort am Ausgang bereitsteht.
- **Einheitlicher Status** – Alle relevanten Zustandsinformationen werden über die Ausgänge `QO` und `STATUS` zurückgemeldet.

Zustandsübersicht

Der Baustein `NVS_AIS` selbst besitzt keine explizite Zustandsmaschine. Sein Verhalten wird vollständig durch den eingebetteten `NVS`-Funktionsblock gesteuert, der die typischen Zustände eines NVS-Treibers durchläuft (z. B. Initialisierung, Bereit, Fehler). Die Ereignisse `INIT` und `INITO` bilden den Start-Stopp-Zyklus ab; die Adapter-Ereignisse `E1` leiten die Schreib-/Leseanforderungen ein.

Anwendungsszenarien

- **Konfigurationsspeicher** – Persistentes Ablegen von Geräteeinstellungen (z. B. Netzwerk-Parameter, Betriebsmodi) als Zeichenketten.
- **Datenaustausch mit anderen Bausteinen** – Verwendung der AIS-Adapter zur einfachen Einbindung in eine IEC-61499-Anwendung, ohne direkte Bus- oder Speicherzugriffe programmieren zu müssen.
- **Wiederherstellung nach Neustart** – Die Kombination aus INIT, automatischem Lesen und Default-Werten stellt sicher, dass ein defnitiert initialisierter Zustand erreicht wird.

Vergleich mit ähnlichen Bausteinen

- **`NVS` (direkt)**: Der direkte NVS-Baustein bietet detailliertere Steuerungsmöglichkeiten (eigene Ereignisse für GET/SET), erfordert aber eine manuelle Verbindung der Schnittstellen. `NVS_AIS` vereinfacht die Anwendung durch die Einbettung der Lese-/Schreiblogik und die standardisierten Adapter.
- **Weitere Speicher-Bausteine (z. B. für SD-Karte)**: Diese arbeiten meist mit anderen Protokollen (SPI, I²C) und bieten unterschiedliche Speicherschemata. `NVS_AIS` ist auf die spezifischen Eigenschaften des ESP32-NVS optimiert (Schlüssel-Wert-Paare, geringe Latenz).

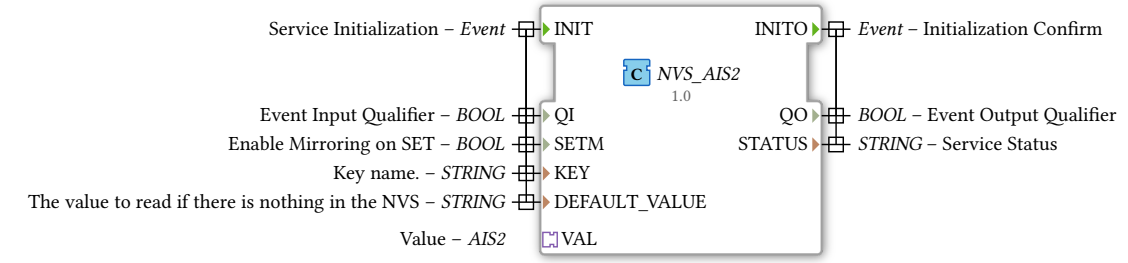
Fazit

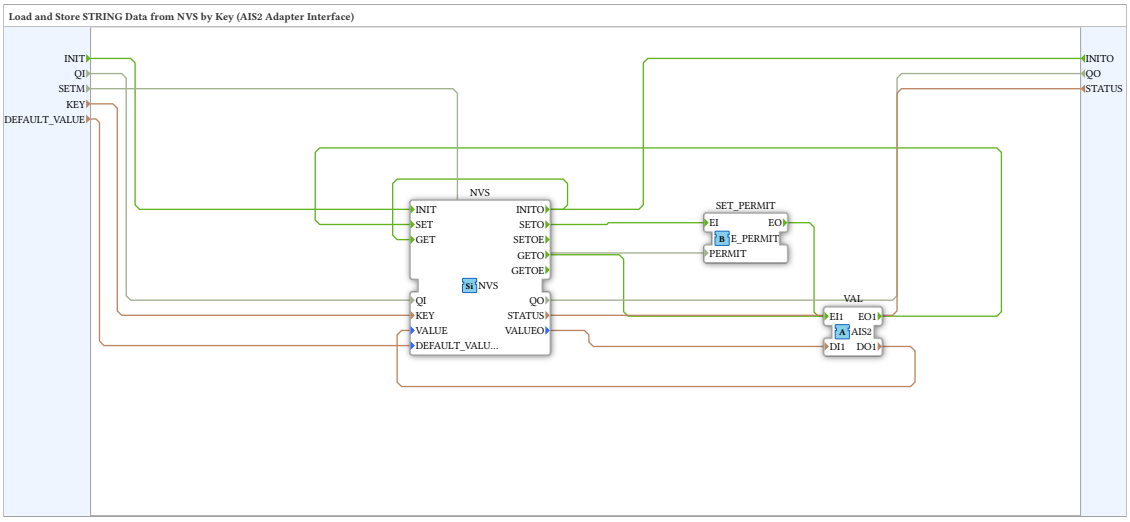
`NVS_AIS` ist ein praktischer Funktionsblock, der das Speichern und Laden von Zeichenketten im nichtflüchtigen Speicher des ESP32 kapselt und über eine saubere Adapter-Schnittstelle bereitstellt. Durch die automatisierte Initialisierung und die Integration von Lese- und Schreibvorgängen eignet er sich besonders für Anwendungen, die eine einfache, zuverlässige Persistenz von Konfigurationsdaten benötigen. Die Adapter-Anbindung ermöglicht eine lose Kopplung mit anderen Bausteinen und erhöht die Wiederverwendbarkeit in verschiedenen IEC-61499-Projekten.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 ESP32 & ESP32-S3 DevKit auf ms-muc-docs.de
- 🌐 MCU vs. MPU Vergleichsguide auf ms-muc-docs.de

NVS_AIS2





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **NVS_AIS2** dient dem Laden und Speichern von STRING-Daten im nichtflüchtigen Speicher (NVS) eines ESP32 über einen Adapter. Er kapselt die Initialisierung und den Zugriff auf einen einzelnen NVS-Eintrag, der über einen Schlüssel (KEY) identifiziert wird. Der FB bietet eine initialisierende Ereignisschnittstelle und eine bidirektionale Adapter-Schnittstelle, über die extern auf den gespeicherten Wert zugegriffen werden kann.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar	Mitgeführte Variablen
INIT	EInit	Service Initialization	QI, SETM, KEY, DEFAULT_VALUE

Ereignis-Ausgänge

Ereignis	Typ	Kommentar	Mitgeführte Variablen
INITO	EInit	Initialization Confirm	QO, STATUS

Daten-Eingänge

Name	Typ	Kommentar
QI	BOOL	Event Input Qualifier
SETM	BOOL	Spiegelung (Mirroring) bei SET aktivieren (Enable Mirroring on SET)
KEY	STRING	Schlüsselname für den NVS-Eintrag
DEFAULT_VALUE	STRING	Standardwert, falls der Schlüssel im NVS nicht existiert (Voreinstellung: '')

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Event Output Qualifier
STATUS	STRING	Dienststatus

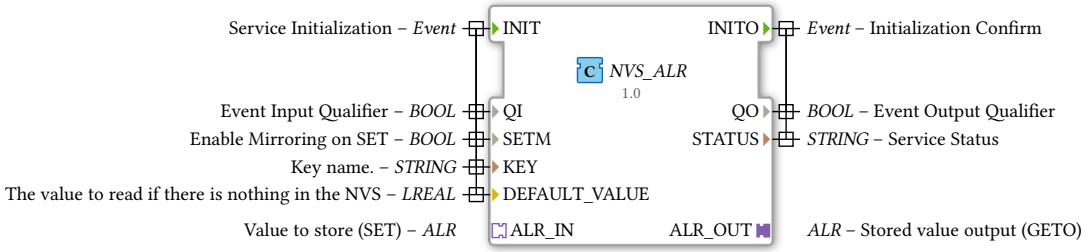
Adapter

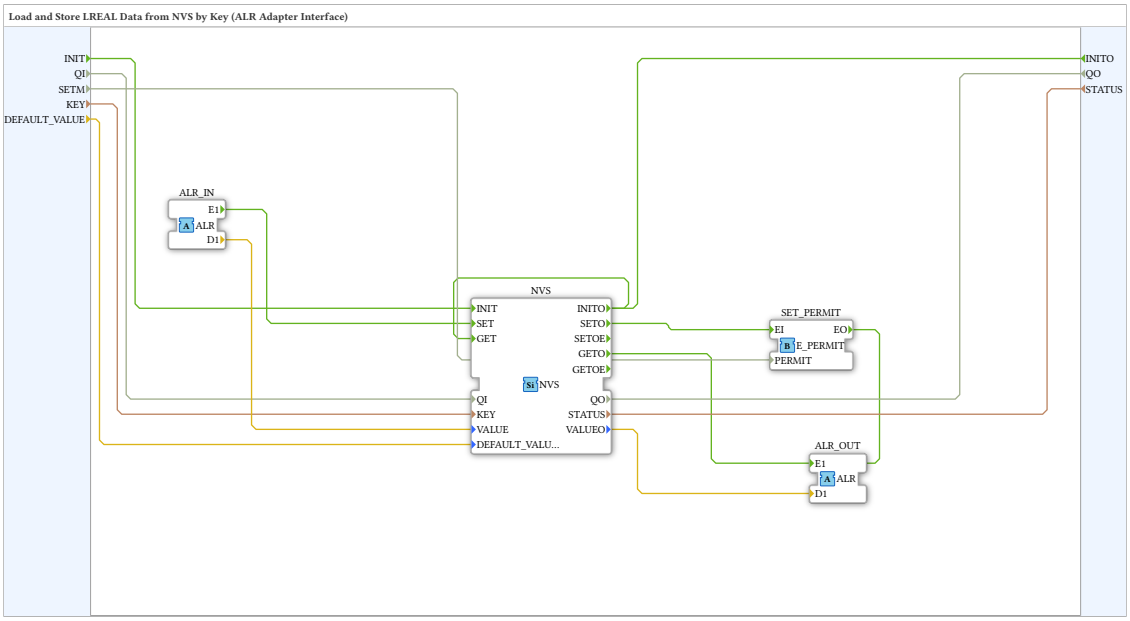
Adapter	Typ	Kommentar
VAL	adapter::types::bidirectional::AIS2 (Socket)	Wert (STRING)

Funktionsweise

1. **Initialisierung**: Ein Ereignis am Eingang INIT löst die Initialisierung des internen NVS-Bausteins aus.
2. **Wert auslesen**: Der gelesene Wert wird über den Adapter VAL als DI1 ausgegeben.
3. **Wert speichern**: Ein externer Baustein kann über den Adapter VAL ein Ereignis EO1 senden, um einen neuen Wert (DO1) in den NVS zu schreiben.

NVS_ALR





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **NVS_ALR** dient zum Laden und Speichern von Werten des Typs `LREAL` in einem nichtflüchtigen Speicher (Non-Volatile Storage, NVS) unter Verwendung eines Schlüssels (Key). Er ist speziell für die Anbindung über den unidirektionalen **ALR-Adapter** ausgelegt. Der Baustein kombiniert eine Initialisierungsphase, in der der Speicherinhalt gelesen wird, mit einer späteren Schreib- und Leseoperation über den Adapter.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung	Mitgeführte Variablen
INIT	Initialisiert den Baustein und führt einen ersten Lesevorgang aus dem NVS durch.	QI, SETM, KEY, DEFAULT_VALUE

Ereignis-Ausgänge

Ereignis	Beschreibung	Mitgeführte Variablen
INITO	Bestätigung der Initialisierung und Rückmeldung des Ergebnisses.	QO, STATUS

Daten-Eingänge

Variable	Typ	Beschreibung
QI	BOOL	Steuerung für die Initialisierung (TRUE = aktiv).
SETM	BOOL	Aktiviert die Spiegelung (Mirroring) bei SET.
KEY	STRING	Schlüsselname für den NVS-Zugriff.
DEFAULT_VALUE	LREAL	Wert, der gelesen wird, falls im NVS noch kein Eintrag unter dem angegebenen Schlüssel existiert.

Daten-Ausgänge

Variable	Typ	Beschreibung

QO	BOOL	Quittung der Initialisierung (TRUE = erfolgreich).
STATUS	STRING	Statusmeldung (z. B. Fehlertext bei misslungenem Zugriff).

Adapter

Adapter	Richtung	Typ	Beschreibung
ALR_IN	Socket	`adapter::types::unidirectional::ALR`	Empfängt Schreibaufträge (Wert speichern).
ALR_OUT	Plug	`adapter::types::unidirectional::ALR`	Sendet gelesene Werte nach einem Lesevorgang.

Funktionsweise

1. **Initialisierung** Nach einem Ereignis am **INIT**-Eingang wird der interne Baustein `NVS` mit dem angegebenen Schlüssel (`KEY`), dem Default-Wert (`DEFAULT_VALUE`) und der Aktivierung (`QI`) aufgerufen. Der Initialisierungsvorgang öffnet den NVS-Bereich und liest sofort den vorhandenen Wert aus. Dieser Wert wird über den **ALR_OUT**-Adapter als Ereignis mit dem zugehörigen Datenwert (`D1`) ausgegeben. Gleichzeitig wird der **INITO**-Ereignisausgang mit den Quittungen (`QO`, `STATUS`) ausgelöst.
2. **Schreiben über ALR_IN** Ein Ereignis am **ALR_IN.E1**-Eingang (verbunden mit dem Socket) löst einen **SET**-Vorgang aus. Der mitgelieferte Datenwert (`ALR_IN.D1`) wird in den NVS unter dem zuvor gespeicherten Schlüssel geschrieben. Nach erfolgreichem Schreiben wird das Ereignis an **ALR_OUT.E1** weitergegeben und der geschriebene Wert dort ausgegeben.
3. **Lesen über ALR_IN?** Im vorliegenden Netzwerk wird nach der Initialisierung automatisch ein **GET**-Befehl ausgelöst. Ein expliziter Leseaufruf über den Adapter ist im Entwurf nicht vorgesehen; der Baustein gibt den aktuellen Wert immer nach jeder NVS-Operation (INIT, SET) über **ALR_OUT** aus.

Technische Besonderheiten

- **Adapterinterface** Die Kommunikation mit der Außenwelt erfolgt ausschließlich über den unidirektionalen ALR-Adapter. Dadurch wird der Baustein besonders geeignet für modulare Architekturen, bei denen Datenflüsse über standardisierte Schnittstellen abgewickelt werden.
- **Interner NVS-Baustein** Im Netzwerk wird der FB `NVS` (aus der Bibliothek `logiBUS::storage::esp32_nvs`) verwendet. Dieser kapselt die eigentliche Lese- und Schreiblogik auf dem ESP32-NVS.
- **Automatischer Lesevorgang** Nach der Initialisierung wird sofort ein GET ausgeführt, sodass der Anwender unmittelbar den aktuellen Wert (Standardwert oder gespeicherten Wert) erhält.
- **Fehlerbehandlung** Die Ausgänge `QO` und `STATUS` erlauben eine einfache Prüfung des Initialisierungserfolgs. Fehler werden als String gemeldet.

Zustandsübersicht

Der Baustein durchläuft keine explizit modellierten Zustände, sondern ist ereignisgesteuert. Folgendes Verhalten tritt auf:

- **Ruhezustand** – Kein INIT erfolgt. Es werden keine Aktionen ausgeführt.
- **Initialisieren** – INIT empfangen; NVS-Operationen werden gestartet; nach Abschluss wird INITO ausgelöst und ein GET angestoßen.
- **Bereit** – Nach erfolgreicher Initialisierung wartet der Baustein auf Ereignisse über **ALR_IN.E1** (Schreiben) oder interne GETs (werden nur initial automatisch ausgelöst).
- **Schreiben** – Bei ALR_IN.E1 wird der Wert gespeichert und das Ergebnis an ALR_OUT gesendet.

Anwendungsszenarien

- **ESP32-Konfigurationsparameter** Speicherung von LREAL-Werten wie Kalibrierungsfaktoren, Schwellwerten oder PID-Parametern, die nach einem Neustart erhalten bleiben sollen. Der ALR-Adapter ermöglicht die einfache Integration mit anderen Bausteinen, die denselben Adaptertyp verwenden.

- **Gekapselter Datenzugriff** Wenn ein Baustein LREAL-Werte aus dem NVS lesen und schreiben muss, aber die direkte Verwendung des NVS-Bausteins zu komplex erscheint, bietet NVS_ALR eine abstrahierte Schnittstelle.
- **Modulare Automatisierung** Einsatz in Industrie 4.0-Anwendungen, bei denen Geräte über standardisierte Adapterdienste kommunizieren (z. B. über eine ALR-Konfigurationsschicht).

Vergleich mit ähnlichen Bausteinen

- **NVS (direkt)** Der Baustein `NVS` bietet mehr Flexibilität (z. B. wahlfreier Zugriff, unterschiedliche Datentypen), erfordert aber eine aufwändigere Verkabelung und explizite GET/SET-Aufrufe. NVS_ALR reduziert die Komplexität auf eine Adapterschnittstelle.
- **BOOL_NVS, INT_NVS (hypothetisch)** Analoge Bausteine für andere Datentypen. NVS_ALR ist speziell auf LREAL zugeschnitten und nutzt den ALR-Adapter, der typischerweise für LREAL-Daten vorgesehen ist.
- **Remanente Variablen** In manchen Systemen können remanente Variablen direkt im FB-Modell verwendet werden. NVS_ALR ist jedoch explizit für den Einsatz mit einem externen Flash-Speicher (NVS) konzipiert und daher portabler.

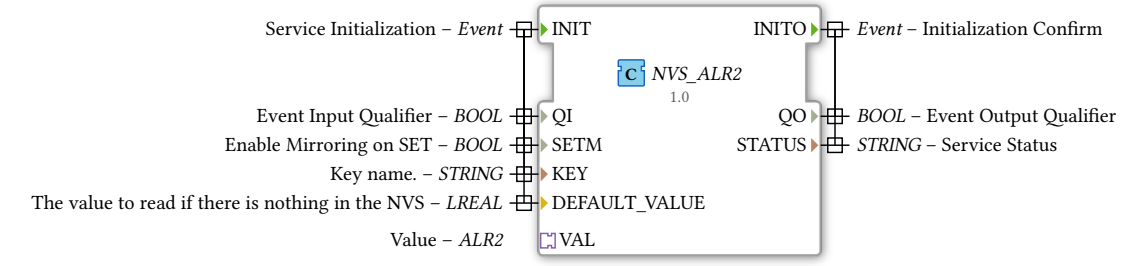
Fazit

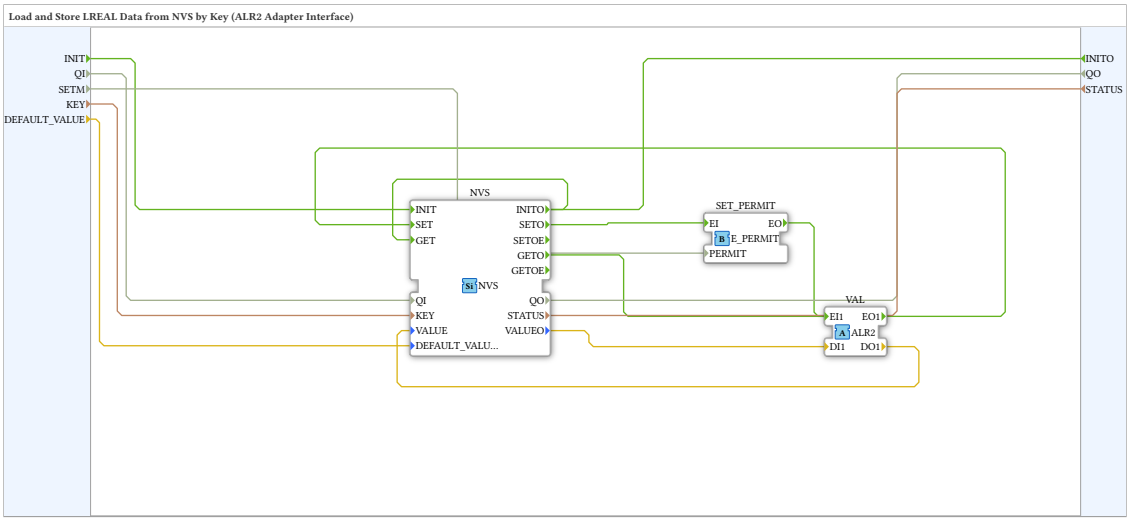
Der **NVS_ALR**-Funktionsbaustein bietet eine kompakte und benutzerfreundliche Möglichkeit, LREAL-Werte persistent in einem NVS zu speichern und über einen unidirektionalen Adapter auszutauschen. Durch die Kombination von Initialisierung, automatischem Lesen und Adapter-basiertem Schreiben eignet er sich besonders für modulare und wiederverwendbare Steuerungsanwendungen auf ESP32-Basis. Sein Design folgt dem IEC 61499-Standard und ermöglicht eine saubere Trennung von Speicherlogik und Applikation.

Passende Themen-Unterseiten auf ms-muc-docs.de

- ESP32 & ESP32-S3 DevKit auf ms-muc-docs.de

NVS_ALR2





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **NVS_ALR2** dient dem Laden und Speichern von LREAL-Daten im nichtflüchtigen Speicher (NVS) eines ESP32 über einen Adapter. Er kapselt die Initialisierung und den Zugriff auf einen einzelnen NVS-Eintrag, der über einen Schlüssel (KEY) identifiziert wird. Der FB bietet eine initialisierende Ereignisschnittstelle und eine bidirektionale Adapter-Schnittstelle, über die extern auf den gespeicherten Wert zugegriffen werden kann.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar	Mitgeführte Variablen
`INIT`	EInit	Service Initialization	QI, SETM, KEY, DEFAULT_VALUE

Ereignis-Ausgänge

Ereignis	Typ	Kommentar	Mitgeführte Variablen
`INITO`	EInit	Initialization Confirm	QO, STATUS

Daten-Eingänge

Name	Typ	Kommentar
`QI`	BOOL	Event Input Qualifier
`SETM`	BOOL	Enable Mirroring on SET: Bei TRUE wird die Bestätigung eines SET (SETO) zusätzlich am Ausgangsadapter ausgegeben (gespiegelt)
`KEY`	STRING	Schlüsselname für den NVS-Eintrag
`DEFAULT_VALUE`	LREAL	Standardwert, falls der Schlüssel im NVS nicht existiert (Voreinstellung: 0.0)

Daten-Ausgänge

Name	Typ	Kommentar
`QO`	BOOL	Event Output Qualifier
`STATUS`	STRING	Dienststatus

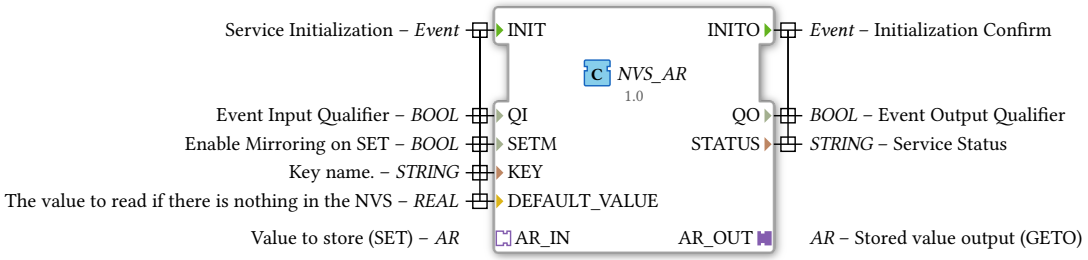
Adapter

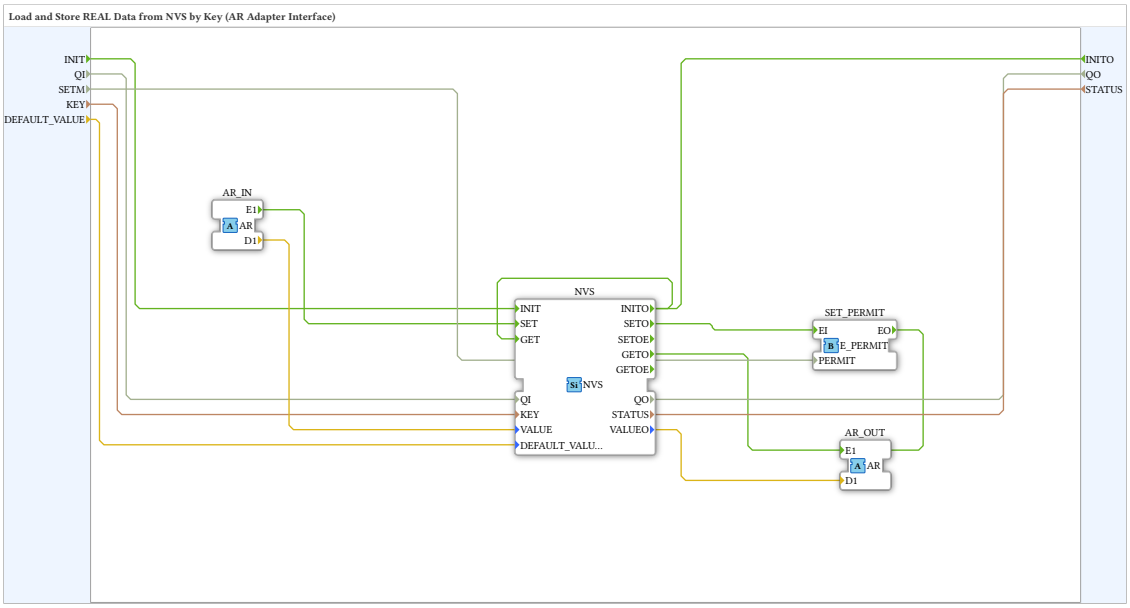
Adapter	Typ	Kommentar
`VAL`	`adapter::types::bidirectional::ALR2` (Socket)	Wert (LREAL)

Funktionsweise

1. **Initialisierung**: Ein Ereignis am Eingang `INIT` löst die Initialisierung des internen NVS-Bausteins aus.
2. **Wert auslesen**: Der gelesene Wert wird über den Adapter `VAL` als `DI1` ausgegeben.
3. **Wert speichern**: Ein externer Baustein kann über den Adapter `VAL` ein Ereignis `EO1` senden, um einen neuen Wert (`DO1`) in den NVS zu schreiben.

NVS_AR





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `NVS_AR` dient zum Laden und Speichern von REAL-Daten in einem nichtflüchtigen Speicher (Non-Volatile Storage, NVS) anhand eines Schlüssels (Key). Er stellt über zwei AR-Adapter-Schnittstellen eine unidirektionale Kommunikation mit anderen IEC-61499-Bausteinen bereit. Der Baustein kapselt den internen `NVS`-Baustein und erweitert dessen Funktionalität um eine standardisierte Adapter-Anbindung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit) – Initialisiert den Baustein. Beim Eintreten dieses Ereignisses werden die Daten-Eingänge `QI`, `SETM`, `KEY` und `DEFAULT_VALUE` übernommen.

Ereignis-Ausgänge

- **INITO** (EInit) – Bestätigt den erfolgreichen Abschluss der Initialisierung. Nach diesem Ereignis sind die Ausgänge `QO` und `STATUS` gültig.

Daten-Eingänge

- **QI** (BOOL) – Eingangsqualifizierer; steuert die Ausführung der Initialisierung.
- **SETM** (BOOL) – Spiegelung (Mirroring) bei SET aktivieren (Enable Mirroring on SET).
- **KEY** (STRING) – Schlüsselname, unter dem der Wert im NVS gespeichert/abgerufen wird.
- **DEFAULT_VALUE** (REAL) – Standardwert, der zurückgegeben wird, falls kein Eintrag mit dem angegebenen Schlüssel im NVS existiert.

Daten-Ausgänge

- **QO** (BOOL) – Ausgangsqualifizierer; zeigt den Erfolg der Initialisierung an.

- **STATUS** (STRING) – Statusmeldung zur Fehlerdiagnose oder Bestätigung.

Adapter

- **AR_IN** (Adapter vom Typ `adapter::types::unidirectional::AR`) – Empfängt einen zu speichernden Wert (SET-Ereignis und Daten).
- **AR_OUT** (Adapter vom Typ `adapter::types::unidirectional::AR`) – Stellt den gelesenen Wert für andere Bausteine bereit (GETO-Ereignis und Daten).

Funktionsweise

Der Baustein arbeitet wie folgt:

1. **Initialisierung**: Ein Ereignis an `INIT` startet die Initialisierung des internen `NVS`-Bausteins. Die Eingänge `QI`, `SETM`, `KEY` und `DEFAULT_VALUE` werden an `NVS` weitergeleitet.
2. **Nach Initialisierung**: Sobald `NVS` die Initialisierung abschließt (Ereignis `INITO`), wird automatisch ein **Lesebefehl** (`GET`) an `NVS` gesendet. Der gelesene Wert (oder der Standardwert) wird anschließend über den Adapterausgang `AR_OUT` ausgegeben.
3. **Speichern**: Ein über den Adaptereingang `AR_IN` eingehendes Ereignis (`AR_IN.E1`) löst einen **Speicherbefehl** (`SET`) im internen `NVS` aus. Der zu speichernde Wert wird aus dem Adapterdatenkanal (`AR_IN.D1`) übernommen.
4. **Rückmeldung**: Sowohl nach einem Lese- als auch nach einem Speichervorgang wird das Ergebnis (Erfolg/Fehler) über das Adapterereignis `AR_OUT.E1` und über die Datenausgänge `QO`/`STATUS` zurückgemeldet.

Technische Besonderheiten

- Der Baustein verwendet den unidirektionalen AR-Adapter, der eine lose Kopplung zwischen Funktionsblöcken ermöglicht.
- Er ist speziell für REAL-Daten ausgelegt; andere Datentypen können nicht verarbeitet werden.
- Die Implementierung basiert auf dem generischen `NVS`-Baustein der Bibliothek `logiBUS::storage::esp32_nvs`.
- Die Ereignissteuerung stellt sicher, dass nach der Initialisierung automatisch der aktuelle Wert aus dem NVS gelesen und bereitgestellt wird.

Zustandsübersicht

Der `NVS_AR`-Baustein verfügt über keine explizite Zustandsmaschine in der XML-Definition. Das funktionale Verhalten ergibt sich aus den Ereignisverkettungen:

- **Ruhezustand**: Der Baustein wartet auf ein Ereignis an `INIT` oder `AR_IN.E1`.
- **Initialisierungsphase**: Nach `INIT` wird der interne `NVS`-Baustein gestartet. In dieser Phase sind die Ausgänge noch nicht gültig.
- **Betriebsphase**: Nach `INITO` ist der Baustein bereit, Werte zu lesen (automatisch) und über `AR_IN` gespeichert zu bekommen.

Anwendungsszenarien

- **Konfigurationsspeicher**: Speichern und Abrufen von REAL-Konfigurationsparametern (z. B. Grenzwerte, Faktoren) in einem ESP32-NVS.
- **Kalibrierdaten**: Hinterlegen von Kalibrierwerten für Sensorik oder Aktorik.
- **Persistente Betriebsdaten**: Ablegen von Zählwerten oder Summen, die über Netzausfälle hinweg erhalten bleiben sollen.

Vergleich mit ähnlichen Bausteinen

Merkmal	`NVS_AR`	Direkter `NVS`-Baustein
Schnittstelle	AR-Adapter (unidirektional)	Standard-Ereignis-/Dateneingänge
Einsatzzweck	Einfache Einbindung in adapterbasierte Netzwerke	Flexibel, aber aufwändigere Verdrahtung
Datentyp	REAL	Mehrere Typen (über generischen Parameter)

Der NVS_AR vereinfacht die Integration, indem er die Adapter-Anbindung standardisiert, während der reine NVS-Baustein eine breitere Typenunterstützung bietet.

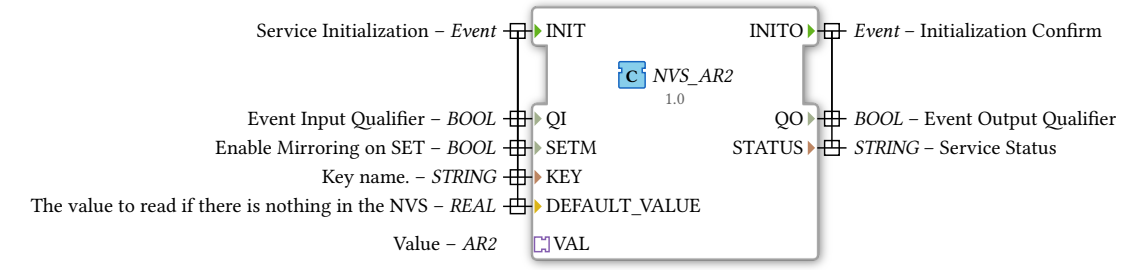
Fazit

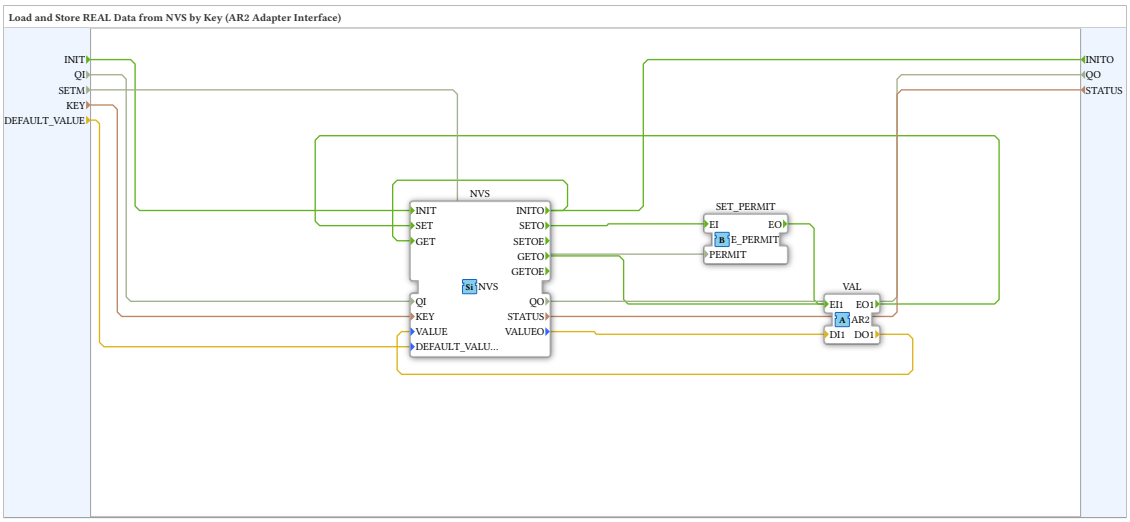
Der NVS_AR-Funktionsblock bietet eine kompakte und adapterbasierte Lösung zum Laden und Speichern von REAL-Daten im nichtflüchtigen Speicher. Durch die Verwendung des AR-Adapters lässt er sich besonders leicht in bestehende IEC-61499-Netzwerke einfügen, ohne auf detaillierte Verbindungen achten zu müssen. Die automatische initiale Auslese nach der Initialisierung erleichtert die Anwendung in typischen Szenarien der Echtzeitsteuerung.

Passende Themen-Unterseiten auf ms-muc-docs.de

- ESP32 & ESP32-S3 DevKit auf ms-muc-docs.de

NVS_AR2





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein NVS_AR2 dient zum Speichern und Laden von REAL-Werten (Gleitkommazahlen) in einen nichtflüchtigen Speicher (NVS – Non-Volatile Storage) über einen Schlüssel (Key). Er kapselt die Logik des zugrundeliegenden NVS-Bausteins und bietet eine bidirektionale Adapterschnittstelle vom Typ AR2 für eine asynchrone Kommunikation. Dadurch kann der Baustein in eine ereignisgesteuerte Umgebung integriert werden, bei der Lese- und Schreiboperationen asynchron ausgelöst werden.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisiert den Baustein. Mit diesem Ereignis werden die Parameter QI (Qualifier), SETM (Spiegelung bei SET), KEY (Schlüssel) und DEFAULT_VALUE (Standardwert) übergeben. Nach erfolgreicher Initialisierung wird der Ausgang INITO ausgelöst.

Ereignis-Ausgänge

- **INITO**: Bestätigung der Initialisierung. Dieses Ereignis signalisiert, dass der Baustein bereit ist. Es wird auch nach einer Lese- oder Schreiboperation über den Adapter ausgelöst (da die Ereignisse verkettet sind).

Daten-Eingänge

- **QI** (BOOL): Eingangsqualifizierer zur Steuerung der Ausführung. Typischerweise wird dieser auf TRUE gesetzt, um die Initialisierung zu aktivieren.
- **SETM** (BOOL): Aktiviert die Spiegelung (Mirroring) bei SET.
- **KEY** (STRING): Der Schlüsselname, unter dem der Wert im NVS gespeichert oder gelesen werden soll.
- **DEFAULT_VALUE** (REAL): Der Wert, der zurückgegeben wird, falls im NVS kein Eintrag unter dem angegebenen Schlüssel existiert.

Daten-Ausgänge

- **QO** (BOOL): Ausgangsqualifizierer. Zeigt den Status der ausgeführten Operation an.
- **STATUS** (STRING): Statusmeldung der Operation (z. B. Erfolg, Fehlercode).

Adapter

- **VAL** (vom Typ `adapter::types::bidirectional::AR2`): Bidirektionaler Adapter zum asynchronen Lesen und Schreiben von Werten. Der Adapter besitzt zwei Ereignisse (EO1 – vom Baustein zum Adapter; EI1 – vom Adapter zum Baustein) sowie zwei Datenports (DO1 – Daten vom Baustein zum Adapter; DI1 – Daten vom Adapter zum Baustein). Über diesen Adapter können externe Komponenten Lese- und Schreibanforderungen an den NVS_AR2 senden und Ergebnisse empfangen.

Funktionsweise

Der NVS_AR2 fungiert als Vermittler zwischen dem eigentlichen NVS-Speicherbaustein (`logiBUS::storage::esp32_nvs::NVS`) und einer übergeordneten Steuerung über den bidirektionalen Adapter. Beim Eintreffen des INIT-Ereignisses werden die Parameter an den internen NVS-Baustein weitergeleitet. Der NVS-Baustein führt eine Initialisierung durch und quittiert mit INITO.

Nach der Initialisierung wird automatisch ein Lesevorgang (GET) ausgelöst: Der NVS-Baustein liest den Wert unter dem angegebenen Schlüssel. Falls kein Eintrag existiert, wird DEFAULT_VALUE verwendet. Das gelesene Ergebnis wird über den Adapter-Ausgang (VAL.DI1) und das Ereignis VAL.EI1 an die angeschlossene Komponente gesendet.

Ein Schreibzugriff erfolgt, wenn die externe Komponente über den Adapter (VAL.EO1) einen Schreibbefehl sendet. Dabei wird der gewünschte Wert über VAL.DO1 an den NVS_AR2 übergeben, der ihn an den NVS-Baustein weiterleitet (SET). Nach erfolgreichem Schreiben quittiert der NVS-Baustein mit SETO, was wiederum das Ereignis VAL.EI1 auslöst, um die Bestätigung an die externe Komponente zurückzugeben.

Technische Besonderheiten

- **Asynchrone Kommunikation**: Der Baustein unterstützt asynchrone Lese- und Schreiboperationen über den AR2-Adapter, sodass er in nebenläufige Systeme integriert werden kann, ohne blockierende Aufrufe.
- **Standardwert**: Durch die Angabe eines DEFAULT_VALUE wird sichergestellt, dass auch bei erstmaliger Nutzung eines Schlüssels ein definierter Wert zurückgegeben wird.
- **Fehlerbehandlung**: STATUS und QO geben Aufschluss über den Erfolg oder Misserfolg der Operation.
- **Wiederverwendung**: Der Baustein kapselt den allgemeinen NVS-Baustein und fügt eine standardisierte Adapterschnittstelle hinzu, was die Wiederverwendung in verschiedenen Steuerungslogiken erleichtert.

Zustandsübersicht

Der Baustein hat keinen explizit definierten Zustandsautomaten im XML, da er über den internen NVS-Baustein und die Ereignisverkettung gesteuert wird. Der Zustand ergibt sich aus der Abfolge der Ereignisse:

1. Warten auf INIT → Initialisierung läuft → INITO und GET werden ausgelöst.
2. Nach GET: Warten auf Leseergebnis → Wert an Adapter gesendet.
3. Warten auf Schreibanforderung über Adapter → SET ausgeführt → Bestätigung zurück.
4. Nach jedem Schritt ist der Baustein bereit für neue Anforderungen.

Anwendungsszenarien

- **Persistente Konfigurationsspeicher**: Speichern und Abrufen von Einstellungen (z. B. Sollwerten, Parametern) in einem eingebetteten System mit NVS (wie ESP32 NVS).
- **Datenprotokollierung**: Asynchrones Ablegen von Messwerten im NVS.
- **Initiale Standardwerte laden**: Bei Systemstart einen gespeicherten Wert laden oder einen Standardwert verwenden, falls noch nie gespeichert.

Vergleich mit ähnlichen Bausteinen

- **NVS**: Der Basisbaustein NVS ohne Adapter hat direkte Ereignisschnittstellen (INIT, REQ, etc.) und erfordert eine direkte Anbindung. NVS_AR2 fügt eine bidirektionale Adapterschnittstelle hinzu, die eine flexiblere, protokollunabhängige Anbindung ermöglicht.
- **Andere Speicherbausteine (z. B. EEPROM, Dateisystem)**: Diese Bausteine haben oft andere Datentypen und Zugriffsmethoden. NVS_AR2 ist speziell auf REAL-Werte und den NVS-Zugriff optimiert.

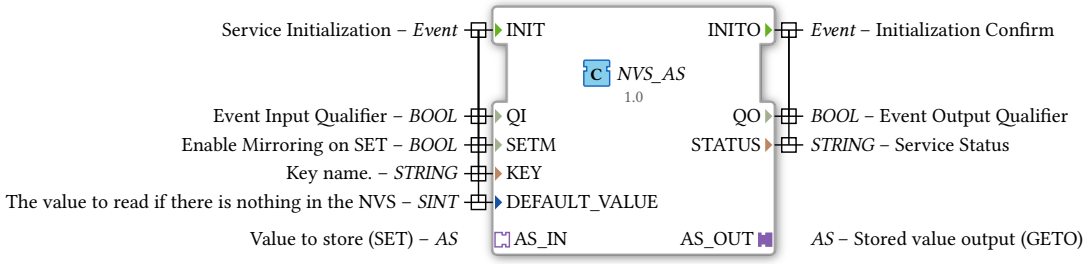
Fazit

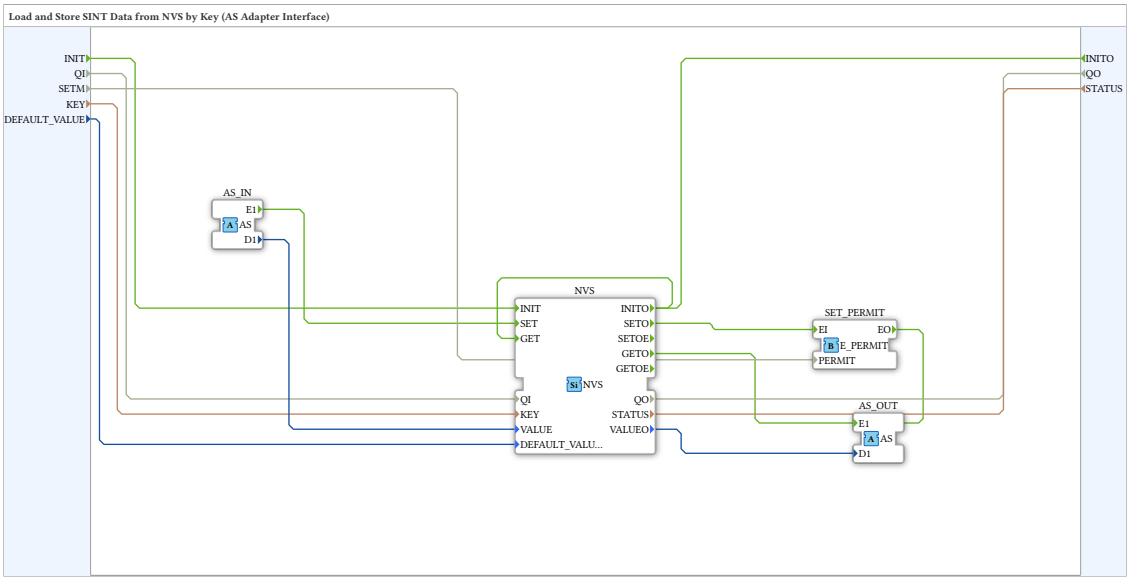
Der Funktionsbaustein NVS_AR2 bietet eine saubere, adapterbasierte Schnittstelle zum asynchronen Lesen und Schreiben von REAL-Daten im nichtflüchtigen Speicher. Durch die Kapselung des NVS-Bausteins und die Integration des AR2-Adapters eignet er sich hervorragend für modulare, ereignisgesteuerte Steuerungssysteme, die persistente Datenverwaltung benötigen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 ESP32 & ESP32-S3 DevKit auf ms-muc-docs.de

NVS_AS





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **NVS_AS** dient als Schnittstelle zum Lesen und Schreiben von SINT-Daten (signed integer) in einem nichtflüchtigen Speicher (Non-Volatile Storage, NVS). Die Speicherung erfolgt über

einen benutzerdefinierten Schlüssel (KEY). Der Baustein erweitert den Zugriff auf den NVS um eine Adapter-Schnittstelle (unidirektionaler AS-Typ), sodass Werte über standardisierte Adapterverbindungen empfangen und gesendet werden können. Dies ermöglicht eine modulare und wiederverwendbare Anbindung an Speicherfunktionen in IEC 61499-Anwendungen.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung	Mitgeführte Variablen
INIT	Initialisiert den NVS und löst das erste Laden eines gespeicherten Werts aus.	QI, SETM, KEY, DEFAULT_VALUE

Ereignis-Ausgänge

Ereignis	Beschreibung	Mitgeführte Variablen
INITO	Bestätigung der Initialisierung und des ersten Ladevorgangs.	QO, STATUS

Daten-Eingänge

Variable	Typ	Beschreibung
QI	BOOL	Freigabe der Initialisierung (TRUE = aktivieren).
SETM	BOOL	Enable Mirroring on SET: Bei TRUE wird die Bestätigung eines SET (SETO) zusätzlich am Ausgangsadapter ausgegeben (gespiegelt)
KEY	STRING	Schlüssel, unter dem der Wert im NVS gespeichert ist.
DEFAULT_VALUE	SINT	Standardwert, der gelesen wird, falls kein Wert unter dem Schlüssel existiert.

Daten-Ausgänge

Variable	Typ	Beschreibung
QO	BOOL	Bestätigt erfolgreiche Initialisierung und Betriebsbereitschaft.
STATUS	STRING	Statusmeldung (z. B. „Success“ oder Fehlertext).

Adapter

Adapter	Typ	Richtung	Beschreibung
AS_IN	AS	SOCKET	Empfängt einen zu speichernden SINT-Wert über ein unidirektionales Adapterprotokoll (Event + Data).
AS_OUT	AS	PLUG	Sendet den gelesenen SINT-Wert über ein unidirektionales Adapterprotokoll an verbundene Bausteine.

Funktionsweise

Der **NVS_AS**-Baustein kapselt eine Instanz des Bausteins **NVS** (aus der Bibliothek `logiBUS::storage::esp32_nvs`). Die Funktion gliedert sich in zwei Betriebsarten:

1. **Initialisierung und erster Lesevorgang** Nach einem INIT-Ereignis wird der interne NVS-Baustein initialisiert. Direkt im Anschluss (automatische Verkettung von `INITO` zu `GET`) wird der unter dem angegebenen KEY gespeicherte Wert gelesen. Existiert noch kein Wert, wird `DEFAULT_VALUE` zurückgegeben. Der gelesene oder vorgegebene Wert wird über den Adapter **AS_OUT** (Ereignis `E1` und Data `D1`) an nachgeschaltete Bausteine gesendet.
2. **Schreiben und Lesen über Adapter**

- Ein über den Adapter **AS_IN** eingehendes Ereignis (`E1`) mit zugehörigem Datenwert (`D1`) löst einen **SET**-Vorgang im NVS aus. Der Wert wird unter dem bei INIT angegebenen KEY gespeichert.
- Nach erfolgreichem SET wird automatisch der gespeicherte Wert über **AS_OUT** ausgegeben (durch die Verbindung von `NVS.SETO` zu `AS_OUT.E1`).
- Ebenso kann ein erneutes Lesen durch ein INIT-Ereignis oder durch den internen Ablauf nach einem erfolgreichen SET angestoßen werden. Ein separates Lese-Ereignis von außen ist nicht vorgesehen; der Wert wird stets nach einer Änderung oder bei Initialisierung aktualisiert.

Der Baustein arbeitet also als **lesender und schreibender Speicherzugriff mit automatischer Rückmeldung des aktuellen Werts**.

Technische Besonderheiten

- **Adapterbasierte Ein-/Ausgabe** Die Verwendung von unidirektionalen AS-Adaptern erlaubt eine lose Kopplung: **AS_IN** empfängt Schreibaufträge, **AS_OUT** gibt den gespeicherten Wert aus. Dies entspricht einem Publisher/Subscriber- oder Client/Server-Muster auf Adapterebene.
- **Automatische Initialisierung** Nach dem INIT-Ereignis wird sofort ein GET ausgeführt, sodass der Baustein nach der Initialisierung sofort den aktuellen oder den Standardwert bereitstellt.
- **Typbeschränkung auf SINT** Der Baustein speichert und lädt ausschließlich SINT-Werte. Für andere Datentypen (z. B. INT, REAL, STRING) sind separate Varianten erforderlich.
- **Fehlerbehandlung** Fehler während der NVS-Operationen (z. B. ungültiger KEY, Speicherfehler) werden über den STATUS-Ausgang als Fehlermeldung ausgegeben, und QO wird auf FALSE gesetzt.

Zustandsübersicht

Der interne NVS-Baustein besitzt einen eigenen Zustandsautomaten. Für den Anwender relevant sind folgende Abläufe:

Phase	Zustand
1	Warten auf INIT-Ereignis.
2	INIT ausgeführt → NVS initialisiert → GET gestartet.
3	GET abgeschlossen → Wert über AS_OUT gesendet → Bereit für Schreibaufträge über AS_IN.
4	Ereignis über AS_IN → SET ausgeführt → Wert gespeichert → Wert über AS_OUT gesendet → zurück zu Phase 3.

Ein erneutes INIT-Ereignis kann jederzeit eine Neuinitialisierung erzwingen.

Anwendungsszenarien

- **Persistente Geräteparameter** Speichern von Konfigurationswerten (z. B. Helligkeit, Verzögerungszeit) im nichtflüchtigen Speicher eines ESP32, bei Neustart automatisch laden.
- **Zustandsmerker für Automatisierung** Merken des letzten Zustands (z. B. Zählerstand, Produktionsparameter) auch nach Spannungsausfall.
- **Adapterbasierte Datenknoten** Einbindung in eine Kette von Adaptern, bei denen ein Baustein Werte setzt und ein anderer sie ausliest.

Vergleich mit ähnlichen Bausteinen

- **NVS (direkt)** Der Baustein `NVS` bietet dieselbe Funktionalität, jedoch ohne Adapterschnittstelle. Er benötigt separate Ereignis- und Datenleitungen. `NVS_AS` vereinfacht die Einbindung in adapterorientierte Architekturen.
- **NVS_AS_REAL, NVS_AS_STRING** Analoge Bausteine für andere Datentypen. Die Schnittstelle und das Verhalten sind identisch, lediglich der Datentyp variiert.
- **Retain-Werte** In manchen Systemen gibt es Retain-Variablen, die ebenfalls persistent sind. `NVS_AS` setzt auf hardwarenahen NVS (z. B. auf ESP32) und ist daher plattformspezifisch, aber performanter und kapazitätsoptimiert.

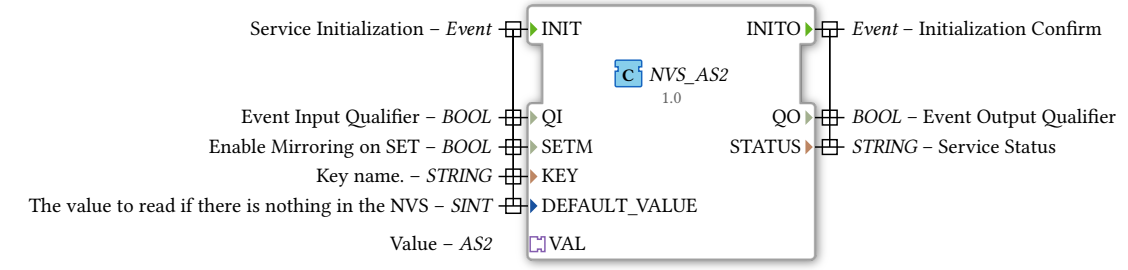
Fazit

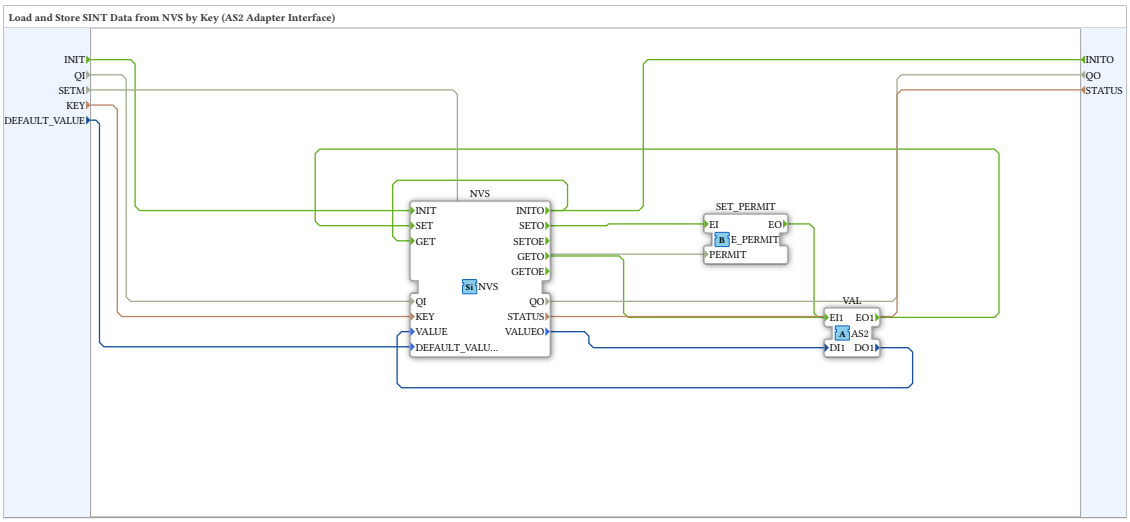
Der Funktionsblock **NVS_AS** stellt eine praktische, adapterbasierte Kapselung des nichtflüchtigen Speichers für SINT-Werte dar. Durch die Kombination von Initialisierungslogik, automatischer Wertrückmeldung und Adapterschnittstelle eignet er sich hervorragend für modulare IEC 61499-Projekte, die eine persistente Speicherung mit minimalem Verdrahtungsaufwand benötigen. Die Beschränkung auf den Typ SINT und die automatische GET-Nach-Initiierung sind zu beachten, erleichtern aber gleichzeitig die Handhabung in vielen Standardanwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 ESP32 & ESP32-S3 DevKit auf ms-muc-docs.de

NVS_AS2





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **NVS_AS2** dient dem Laden und Speichern von SINT-Daten im nichtflüchtigen Speicher (NVS) eines ESP32 über einen Adapter. Er kapselt die Initialisierung und den Zugriff auf einen einzelnen NVS-Eintrag, der über einen Schlüssel (KEY) identifiziert wird. Der FB bietet eine initialisierende Ereignisschnittstelle und eine bidirektionale Adapter-Schnittstelle, über die extern auf den gespeicherten Wert zugegriffen werden kann.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar	Mitgeführte Variablen
`INIT`	EInit	Service Initialization	QI, SETM, KEY, DEFAULT_VALUE

Ereignis-Ausgänge

Ereignis	Typ	Kommentar	Mitgeführte Variablen
`INITO`	EInit	Initialization Confirm	QO, STATUS

Daten-Eingänge

Name	Typ	Kommentar
`QI`	BOOL	Event Input Qualifier
`SETM`	BOOL	Spiegelung (Mirroring) bei SET aktivieren (Enable Mirroring on SET)
`KEY`	STRING	Schlüsselname für den NVS-Eintrag
`DEFAULT_VALUE`	SINT	Standardwert, falls der Schlüssel im NVS nicht existiert (Voreinstellung: 0)

Daten-Ausgänge

Name	Typ	Kommentar
`QO`	BOOL	Event Output Qualifier
`STATUS`	STRING	Dienststatus

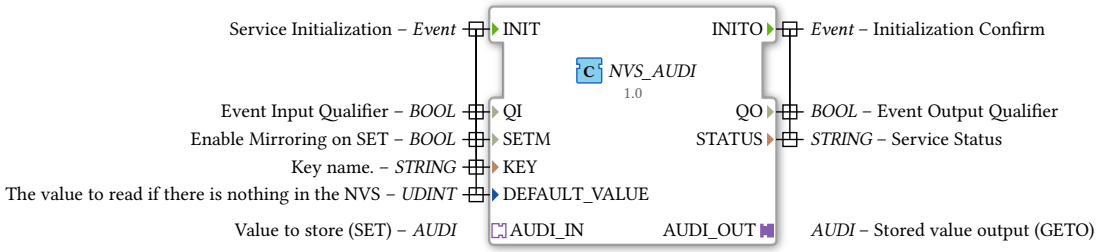
Adapter

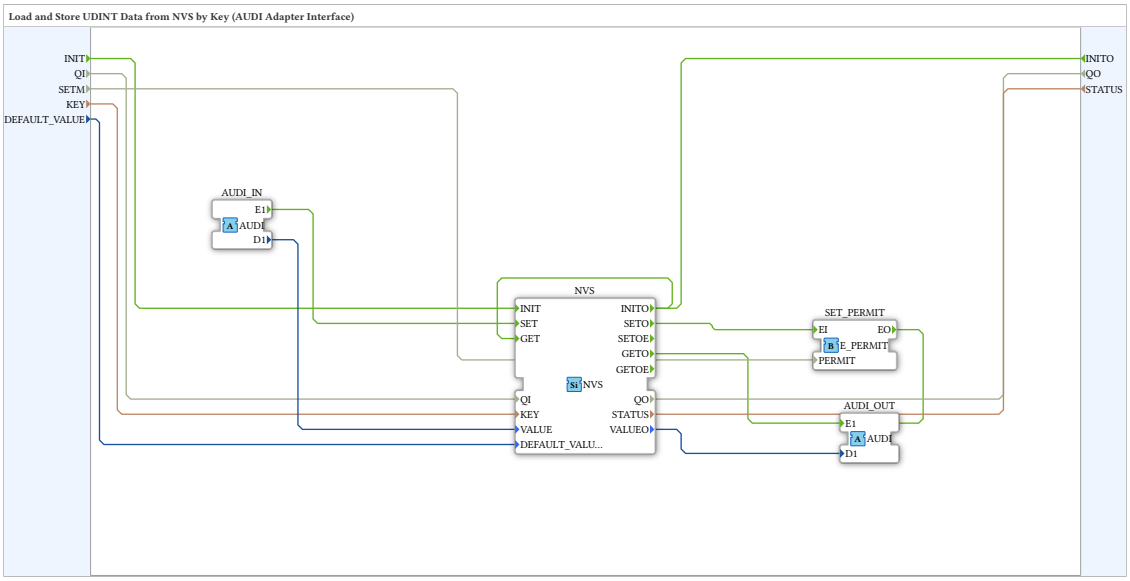
Adapter	Typ	Kommentar
`VAL`	`adapter::types::bidirectional::AS2` (Socket)	Wert (SINT)

Funktionsweise

1. **Initialisierung**: Ein Ereignis am Eingang `INIT` löst die Initialisierung des internen NVS-Bausteins aus.
2. **Wert auslesen**: Der gelesene Wert wird über den Adapter `VAL` als `DI1` ausgegeben.
3. **Wert speichern**: Ein externer Baustein kann über den Adapter `VAL` ein Ereignis `EO1` senden, um einen neuen Wert (`DO1`) in den NVS zu schreiben.

NVS_AUDI





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **NVS_AUDI** ermöglicht das Speichern und Laden von **UDINT**-Daten im **Non-Volatile Storage (NVS)** eines ESP32-Mikrocontrollers. Die Daten werden über einen eindeutigen **Schlüssel (KEY)** adressiert. Der Baustein verwendet ein **unidirektionales AUDI-Adapter-Interface** zur Kommunikation mit dem NVS: Über einen Eingangsadapter (Socket) wird der zu speichernde Wert empfangen, über einen Ausgangsadapter (Plug) wird der gelesene Wert bereitgestellt.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Mitgeführte Daten	Kommentar
INIT	EInit	QI, SETM, KEY, DEFAULT_VALUE	Initialisiert den Dienst

Ereignis-Ausgänge

Name	Typ	Mitgeführte Daten	Kommentar
INITO	EInit	QO, STATUS	Bestätigung der Initialisierung

Daten-Eingänge

Name	Typ	Kommentar
QI	BOOL	Ereignis-Eingangs-Qualifikator
SETM	BOOL	Spiegelung (Mirroring) bei SET aktivieren
KEY	STRING	Schlüsselname für den NVS-Zugriff
DEFAULT_VALUE	UDINT	Standardwert, falls kein Wert im NVS vorhanden

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Ereignis-Ausgangs-Qualifikator
STATUS	STRING	Dienststatus (Fehler/Ok)

Adapter

Name	Typ	Kommentar
AUDI_IN	adapter::types::unidirectional::AUDI	Wert zum Speichern (SET)
AUDI_OUT	adapter::types::unidirectional::AUDI	Gespeicherter/geladener Wert (GETO)

Funktionsweise

Beim Eintreffen des **INIT**-Ereignisses wird der interne NVS-Baustein initialisiert (übergeben werden QI, SETM, KEY und DEFAULT_VALUE). Nach erfolgreicher Initialisierung feuert **INITO** und gleichzeitig wird ein **GET**-Auftrag an den NVS gesendet, um den gespeicherten Wert zu laden. Das Ergebnis des GET-Vorgangs wird über **AUDI_OUT** bereitgestellt (Daten auf D1, Ereignis auf E1).

Wird über den **AUDI_IN**-Adapter ein Ereignis (E1) mit einem Wert (D1) empfangen, so wird ein **SET**-Auftrag an den NVS gesendet, um den Wert unter dem aktuellen KEY zu speichern. Nach dem Speichern feuert **SETO** und das Ergebnis wird ebenfalls über **AUDI_OUT** quittiert.

Sowohl nach dem Laden als auch nach dem Speichern wird der Status (QO, STATUS) über die Ausgänge aktualisiert.

Technische Besonderheiten

- Der Baustein nutzt den **unidirektionalen AUDI-Adapter**, der eine einfache, stream-basierte Datenübergabe erlaubt: Ein Wert wird gesendet (SET) und ein Wert wird empfangen (GETO).
- Intern wird der FB `logiBUS::storage::esp32_nvs::NVS` verwendet, der die eigentliche NVS-Operation auf dem ESP32 ausführt.
- Der Standard-Datentyp ist **UDINT** (unsigned 32-Bit).
- Es werden keine zusätzlichen Zustände oder Zeitgeber benötigt; der FB arbeitet rein ereignisgesteuert.
- Das Attribut **TypeHash** ist mit einem leeren String belegt – es kann für Integritätsprüfungen verwendet werden.

Zustandsübersicht

Da im XML keine explizite Zustandsmaschine definiert ist, ergibt sich der Ablauf aus den Ereignissen:

1. **Initialisierungsphase** – INIT empfangen → NVS-INIT → NVS-GET ausgelöst. Danach: Bereit (wartet auf SET oder erneuten GET durch erneutes INIT).
2. **Speicherphase** – Ereignis über AUDI_IN (E1) → NVS-SET → Wert wird geschrieben.
3. **Lesevorgang** – Automatisch nach INIT oder durch erneutes INIT.

Die Ausgabe über AUDI_OUT erfolgt nach jedem Lese- oder Schreibvorgang.

Anwendungsszenarien

- Speichern von **Konfigurationsparametern** (z. B. Grenzwerte, Betriebsmodi) im NVS, die auch nach Netzausfall erhalten bleiben.
- **Zählerstände** oder **Kalibrierdaten** in industriellen Steuerungen auf ESP32-Basis.
- **Persistente Zustände** von Automatisierungsprozessen, die über einen Schlüssel eindeutig identifiziert werden.
- Austausch von Daten zwischen verschiedenen FBs über den AUDI-Adapter, ohne direkt auf den NVS zuzugreifen.

Vergleich mit ähnlichen Bausteinen

- **NVS (ohne Adapter)**: Bietet direkten Zugriff auf NVS über einzelne Ereignisse und Datenports, benötigt aber mehr Verdrahtungsaufwand.
- **NVS_AUDI** kapselt die NVS-Operationen hinter einem standardisierten Adapter-Interface. Dadurch wird der Baustein leichter austauschbar und die Datenübergabe vereinheitlicht (z. B. mit anderen Speicherbausteinen, die ebenfalls AUDI-Adapter verwenden).
- Andere Speicherbausteine (z. B. **Retain**, **SD-Card**) unterscheiden sich in der Hardware-Anbindung und Latenz; NVS ist speziell für den ESP32 optimiert.

Fazit

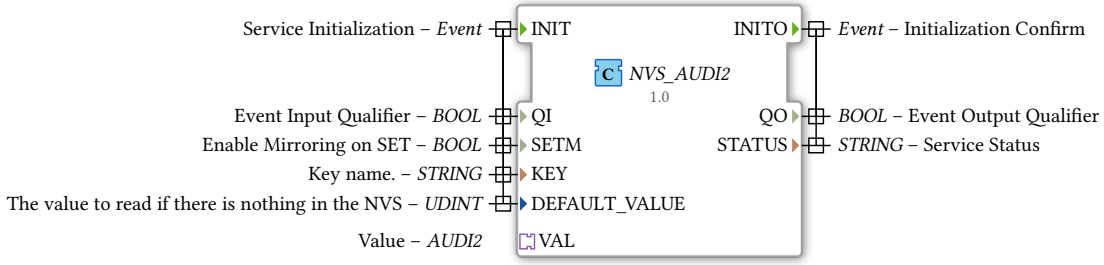
Der **NVS_AUDI**-Funktionsblock stellt eine elegante Lösung zur persistenten Speicherung von UDINT-Daten auf dem ESP32 dar. Durch die Integration des AUDI-Adapter-Interfaces wird die Anbindung

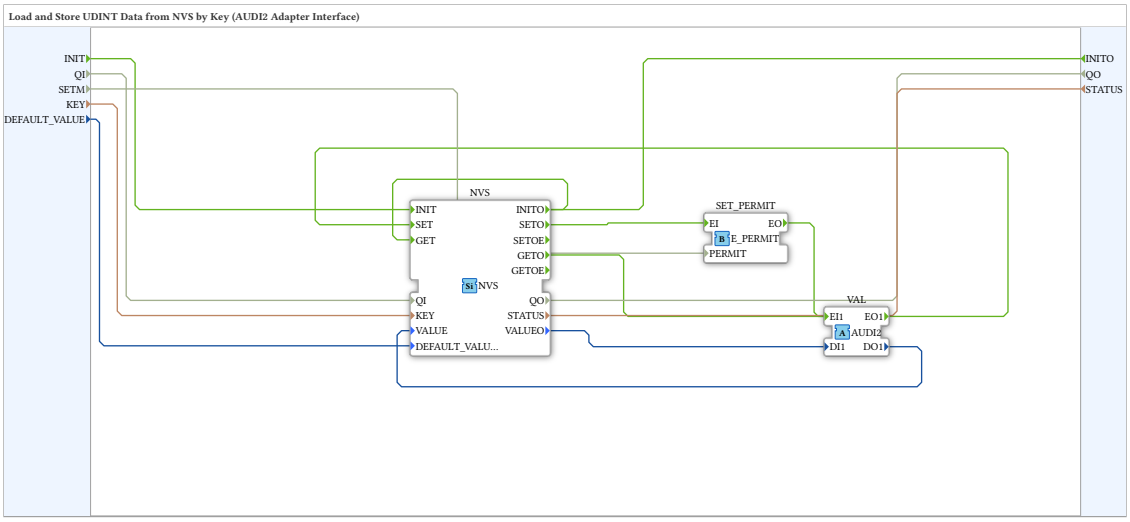
an andere Bausteine vereinfacht und der Code klarer strukturiert. Er eignet sich besonders für Anwendungen, die einen standardisierten Datenaustausch zwischen Speicher- und Verarbeitungskomponenten erfordern.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 ESP32 & ESP32-S3 DevKit auf ms-muc-docs.de
- 🌐 MCU vs. MPU Vergleichsguide auf ms-muc-docs.de

NVS_AUDI2





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **NVS_AUDI2** dient dem Laden und Speichern von UDINT-Daten im nichtflüchtigen Speicher (NVS) eines ESP32 über einen Adapter. Er kapselt die Initialisierung und den Zugriff auf einen einzelnen NVS-Eintrag, der über einen Schlüssel (KEY) identifiziert wird. Der FB bietet eine initialisierende Ereignisschnittstelle und eine bidirektionale Adapter-Schnittstelle, über die extern auf den gespeicherten Wert zugegriffen werden kann.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar	Mitgeführte Variablen
`INIT`	EInit	Service Initialization	QI, SETM, KEY, DEFAULT_VALUE

Ereignis-Ausgänge

Ereignis	Typ	Kommentar	Mitgeführte Variablen
`INITO`	EInit	Initialization Confirm	QO, STATUS

Daten-Eingänge

Name	Typ	Kommentar
QI	BOOL	Event Input Qualifier
SETM	BOOL	Enable Mirroring on SET: Bei TRUE wird die Bestätigung eines SET (SETO) zusätzlich am Ausgangsadapter ausgegeben (gespiegelt)
KEY	STRING	Schlüsselname für den NVS-Eintrag
DEFAULT_VALUE	UDINT	Standardwert, falls der Schlüssel im NVS nicht existiert (Voreinstellung: 0)

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Event Output Qualifier
STATUS	STRING	Dienststatus

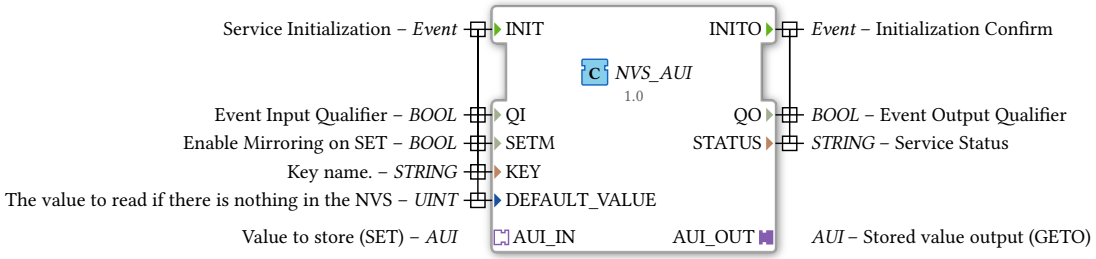
Adapter

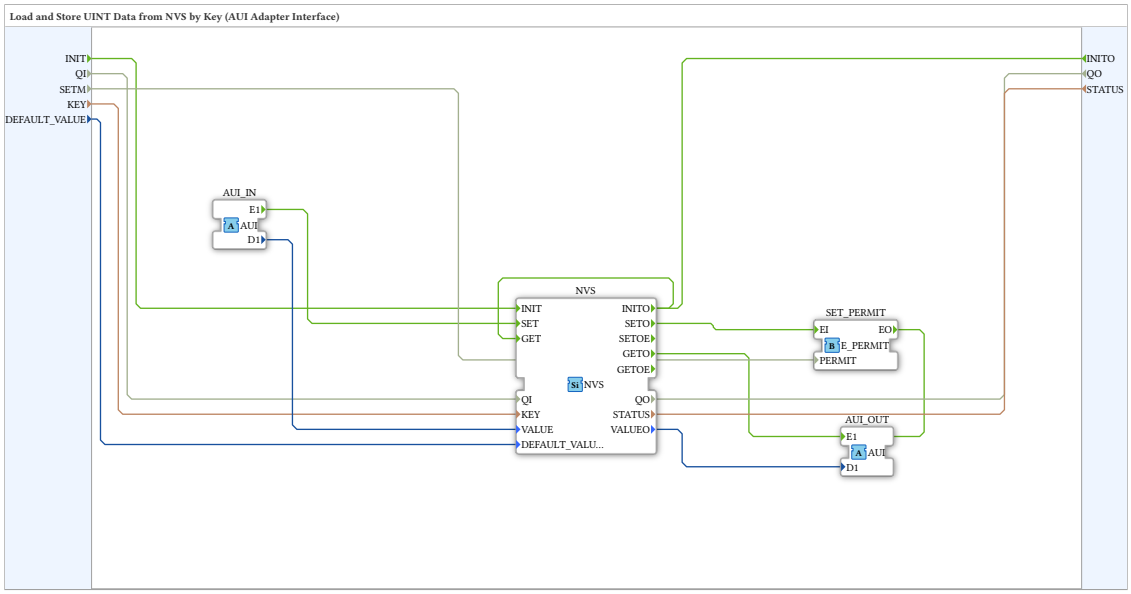
Adapter	Typ	Kommentar
VAL	adapter::types::bidirectional::AUDI2 (Socket)	Wert (UDINT)

Funktionsweise

1. **Initialisierung**: Ein Ereignis am Eingang INIT löst die Initialisierung des internen NVS-Bausteins aus.
2. **Wert auslesen**: Der gelesene Wert wird über den Adapter VAL als DI1 ausgegeben.
3. **Wert speichern**: Ein externer Baustein kann über den Adapter VAL ein Ereignis EO1 senden, um einen neuen Wert (DO1) in den NVS zu schreiben.

NVS_AUI





• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock **NVS_AUI** dient dem Laden und Speichern von UINT-Daten in einem nichtflüchtigen Speicher (NVS) mittels eines Schlüssels (Key). Er kapselt die Zugriffe auf den NVS und stellt eine standardisierte AUI-Adapter-Schnittstelle bereit, über die gelesene Werte ausgegeben und zu speichernde Werte entgegengenommen werden. Typische Einsatzorte sind eingebettete Systeme, die Konfigurations- oder Zustandsdaten persistent ablegen müssen.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Kommentar	Mitgeführte Daten
`INIT`	Service-Initialisierung	`QI` (BOOL), `SETM` (BOOL), `KEY` (STRING), `DEFAULT_VALUE` (UDINT)

Ereignis-Ausgänge

Ereignis	Kommentar	Mitgeführte Daten
`INITO`	Initialisierungsbestätigung	`QO` (BOOL), `STATUS` (STRING)

Daten-Eingänge

Name	Typ	Kommentar
`QI`	BOOL	Qualifikator für das Eingangsereignis
`SETM`	BOOL	Spiegelung (Mirroring) bei SET aktivieren (Enable Mirroring on SET)
`KEY`	STRING	Schlüsselname für den NVS-Zugriff
`DEFAULT_VALUE`	UDINT	Wert, der ausgelesen wird, falls im NVS kein Eintrag vorhanden

Daten-Ausgänge

Name	Typ	Kommentar
`QO`	BOOL	Qualifikator für das Ausgangsereignis
`STATUS`	STRING	Dienststatus-Meldung

Adapter

Adapter	Typ / Richtung	Kommentar
`AUI_IN`	**Socket** (Eingang) – Typ: `adapter::types::unidirectional::AUI`	Liefert den zu speichernden Wert (SET)
`AUI_OUT`	**Plug** (Ausgang) – Typ: `adapter::types::unidirectional::AUI`	Gibt den ausgelesenen Wert aus (GETO)

Funktionsweise

1. **Initialisierung**
   Ein `INIT`-Ereignis löst die Verbindung zum nichtflüchtigen Speicher aus. Dabei werden der Schlüssel (`KEY`) und der Standardwert (`DEFAULT_VALUE`) übergeben. Nach erfolgreicher Initialisierung wird `INITO` mit dem Ergebnis (`QO` / `STATUS`) quittiert.
2. **Daten lesen**
   Nach der Initialisierung wird intern automatisch der gespeicherte Wert zum angegebenen Schlüssel gelesen und über den Ausgangsadapter `AUI_OUT` als Ereignis (E1) und Daten (D1) bereitgestellt. Falls kein Eintrag existiert, wird der `DEFAULT_VALUE` ausgegeben.
3. **Daten speichern**
   Über den Eingangsadapter `AUI_IN` kann ein neuer Wert empfangen werden. Ein eingehendes Ereignis (E1) mit den zugehörigen Daten (D1) wird an den internen NVS-Funktionsblock weitergeleitet, der den Wert unter dem initialisierten Schlüssel persistent ablegt.

Der gesamte Daten- und Ereignisfluss wird durch den eingebetteten Funktionsblock `NVS` (aus der Bibliothek `logiBUS::storage::esp32_nvs`) realisiert.

Technische Besonderheiten

- **AUI-Adapter-Schnittstelle**
  Die unidirektionalen Adapter `AUI_IN` und `AUI_OUT` ermöglichen eine lose Kopplung zwischen dem NVS-Baustein und anderen Komponenten, die standardisierte AUI-Schnittstellen verwenden.
- **Automatischer Lesevorgang nach INIT**
  Direkt nach der Initialisierung wird der Wert gelesen und über `AUI_OUT` bereitgestellt – ohne ein separates „Get“-Ereignis.
- **Fehlerbehandlung**
  Fehler während der Initialisierung oder bei Speicherzugriffen werden über `STATUS` signalisiert.

Zustandsübersicht

Der FB befindet sich nach dem Hochlaufen im **Ruhezustand**.

- **Initialisierung** (beim Eintreffen von `INIT`): Herstellen der NVS-Verbindung, Lesen des vorhandenen Werts.
- **Bereit** (nach erfolgreichem `INITO`): Warten auf eingehende Daten über `AUI_IN` zum Speichern oder auf externe Anfragen über `AUI_OUT`.
- **Speichervorgang** (bei Ereignis an `AUI_IN`): Wert wird persistiert; danach Rückkehr in den Bereit-Zustand.
- **Fehlerzustand** (bei nicht erfolgreicher Initialisierung): `QO` = FALSE, `STATUS` enthält Fehlertext.

Anwendungsszenarien

- **Konfigurationsspeicher** in IoT-Geräten: Ablegen von Netzwerkeinstellungen, Kalibrierdaten oder Benutzereinstellungen.
- **Zustandsüberwachung**: Speichern des letzten Betriebszustands (z. B. Zählerstände) vor einem Spannungsausfall.
- **Parameterübergabe** zwischen unterschiedlichen Automatisierungskomponenten, die über AUI-Adapter kommunizieren.

Vergleich mit ähnlichen Bausteinen

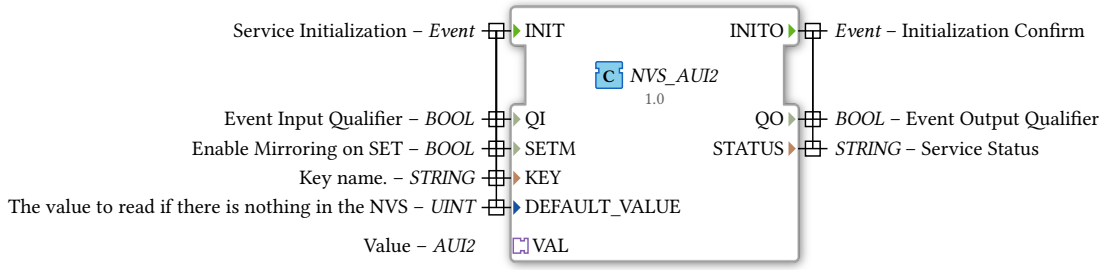
Gegenüber dem direkten **NVS**-Funktionsblock bietet `NVS_AUI` eine höhere Abstraktion durch die Adapter-Schnittstelle. Während der Basis-Baustein separate Ereignisse für Lesen (`GET`) und Schreiben

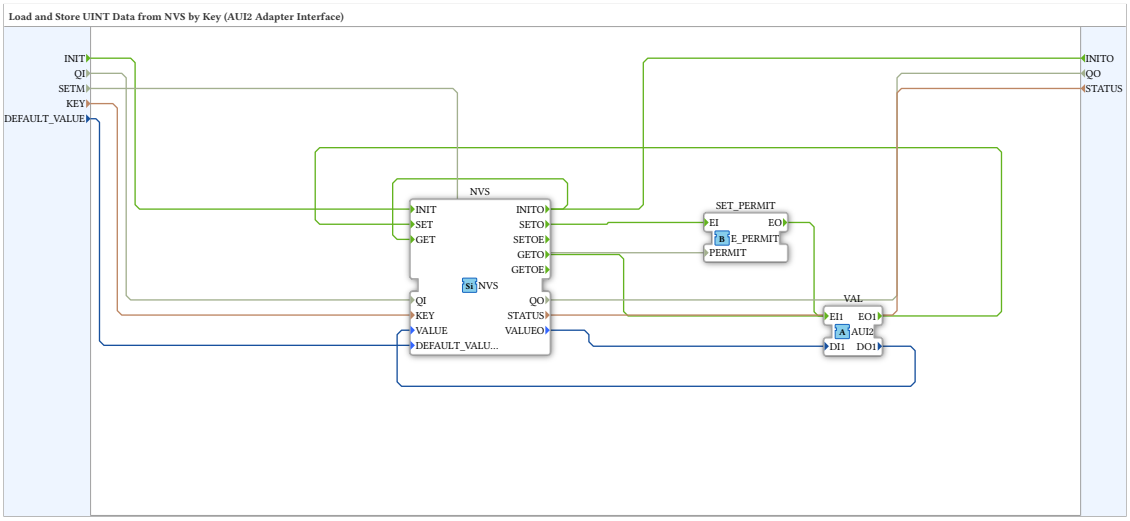
(SET) benötigt, vereinfacht NVS_AUI den Datenfluss durch die unidirektionalen Adapter und den automatischen Lesevorgang nach INIT. Andere Speicherbausteine (z. B. F_MMAP, F_FILE) verwenden meist dateibasierte oder speicherabbildende Verfahren; NVS_AUI ist speziell für nichtflüchtige Speicher mit Schlüssel-Wert-Paaren optimiert.

Fazit

NVS_AUI ist ein praktischer Funktionsblock zur persistenten Datenhaltung in Automatisierungssystemen. Die Kombination aus NVS-Speicher und AUI-Adapter erlaubt eine einfache Einbindung in bestehende IEC 61499-Applikationen, reduziert den Verdrahtungsaufwand und sorgt für eine klare Trennung zwischen Speicherlogik und Kommunikation.

NVS_AUI2





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **NVS_AUI2** dient dem Laden und Speichern von UINT-Daten im nichtflüchtigen Speicher (NVS) eines ESP32 über einen Adapter. Er kapselt die Initialisierung und den Zugriff auf einen einzelnen NVS-Eintrag, der über einen Schlüssel (KEY) identifiziert wird. Der FB bietet eine initialisierende Ereignisschnittstelle und eine bidirektionale Adapter-Schnittstelle, über die extern auf den gespeicherten Wert zugegriffen werden kann.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar	Mitgeführte Variablen
INIT	EInit	Service Initialization	QI, SETM, KEY, DEFAULT_VALUE

Ereignis-Ausgänge

Ereignis	Typ	Kommentar	Mitgeführte Variablen

INITO	EInit	Initialization Confirm	QO, STATUS

Daten-Eingänge

Name	Typ	Kommentar
QI	BOOL	Event Input Qualifier
SETM	BOOL	Aktiviert die Spiegelung (Mirroring) bei SET
KEY	STRING	Schlüsselname für den NVS-Eintrag
DEFAULT_VALUE	UINT	Standardwert, falls der Schlüssel im NVS nicht existiert (Voreinstellung: 0)

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Event Output Qualifier
STATUS	STRING	Dienststatus

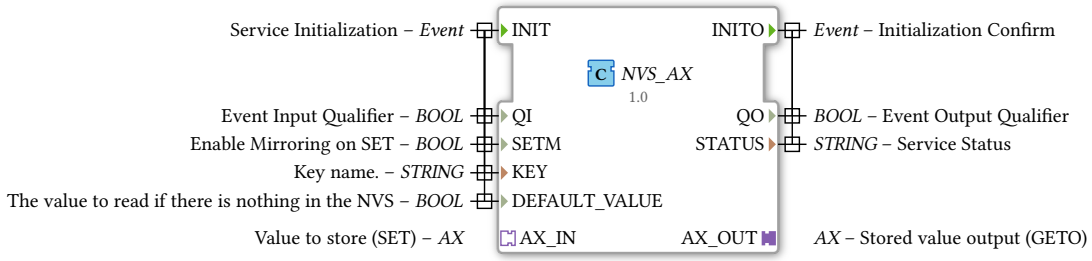
Adapter

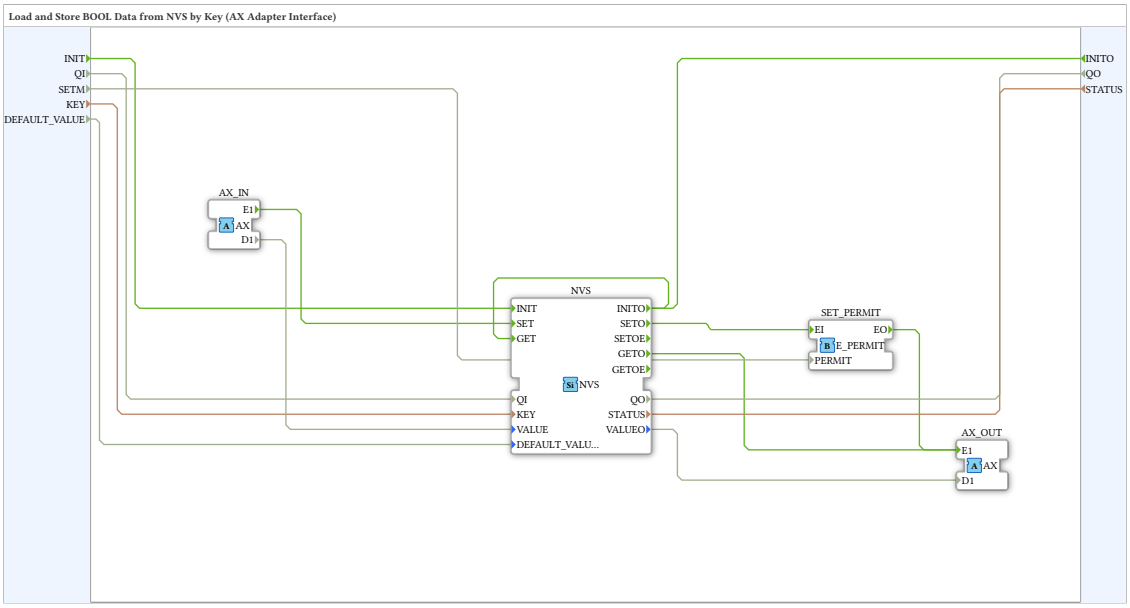
Adapter	Typ	Kommentar
VAL	adapter::types::bidirectional::AUI2 (Socket)	Wert (UINT)

Funktionsweise

1. **Initialisierung**: Ein Ereignis am Eingang INIT löst die Initialisierung des internen NVS-Bausteins aus.
2. **Wert auslesen**: Der gelesene Wert wird über den Adapter VAL als DI1 ausgegeben.
3. **Wert speichern**: Ein externer Baustein kann über den Adapter VAL ein Ereignis EO1 senden, um einen neuen Wert (DO1) in den NVS zu schreiben.

NVS_AX





• – ○ • – ○ • – ○ •

Einleitung

`NVS_AX` lädt und speichert einen `BOOL`-Wert unter einem Schlüssel (`KEY`) im nichtflüchtigen Speicher (Non-Volatile Storage, NVS) des ESP32. Er ist das boolesche Gegenstück zu NVS_AB und verwendet dieselbe Zwei-Adapter-Architektur, jedoch mit unidirektionalen `AX`-Adaptern statt `AB`.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisiert den Baustein und liest den unter `KEY` hinterlegten Wert (bzw. `DEFAULT_VALUE`, falls noch keiner existiert). Liefert `QI`, `SETM`, `KEY`, `DEFAULT_VALUE`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Initialisierung und den ersten Lesezugriff, liefert `QO` und `STATUS`.

Daten-Eingänge

- **QI** (BOOL): Qualifizierer des `INIT`-Ereignisses.
- **SETM** (BOOL): Gibt das Spiegeln (Rückmeldung) beim Schreiben (`SET`) frei.
- **KEY** (STRING): Name des Schlüssels im NVS.
- **DEFAULT_VALUE** (BOOL): Wert, der geliefert wird, wenn unter `KEY` noch nichts im NVS gespeichert ist.

Daten-Ausgänge

- **QO** (BOOL): Erfolgsqualifizierer, mit `INITO` sowie den internen Adapterbestätigungen mitgeliefert.
- **STATUS** (STRING): Statusmeldung des NVS-Zugriffs.

Adapter

- **AX_IN** (Socket, Typ `AX`): Neuer zu speichernder Wert – ein Adapterereignis auf `AX_IN.E1` löst einen `SET` im internen NVS-Baustein aus, `AX_IN.D1` liefert den zu schreibenden Wert.
- **AX_OUT** (Plug, Typ `AX`): Gibt den zuletzt gelesenen Wert aus – sowohl nach dem automatischen Erstlesen bei `INIT` als auch nach jedem bestätigten Schreibvorgang (`AX_OUT.D1`/`AX_OUT.E1`).

Funktionsweise

`NVS_AX` kapselt intern eine Instanz von `logiBUS::storage::esp32_nvs::NVS` sowie ein `E_PERMIT`-Gatter (`SET_PERMIT`). Bei `INIT` wird die interne NVS-Instanz initialisiert (`NVS.INIT`) und anschließend automatisch ein `GET` ausgelöst, das den unter `KEY` gespeicherten Wert (oder `DEFAULT_VALUE`) liefert; das Ergebnis wird über `AX_OUT` ausgegeben, `INITO` bestätigt den Abschluss.

Zum Schreiben liefert der an `AX_IN` angeschlossene Baustein ein Adapterereignis (`AX_IN.E1`) mit dem neuen Wert (`AX_IN.D1`); dies löst `NVS.SET` aus. Ob die erfolgreiche Speicherung zusätzlich über `AX_OUT` zurückgemeldet wird, steuert `SETM`: Ist `SETM` gesetzt, gibt `SET_PERMIT` das `AX_OUT.E1`-Ereignis nach erfolgreichem `SET` frei („Mirroring“).

Technische Besonderheiten

- Reines Composite-FB ohne eigene ECC – die eigentliche Flash-Logik steckt in der internen `NVS`-Instanz, `NVS_AX` verdrahtet nur Ereignisse und Daten zwischen ihr und den beiden `AX`-Adaptern.
- Zwei getrennte, unidirektionale Adapter (`AX_IN`/`AX_OUT`) – dadurch lassen sich Lese- und Schreibpfad an unterschiedliche Gegenstellen anschließen.
- `SETM` steuert nur, ob ein Schreibvorgang zusätzlich per Adapterereignis über `AX_OUT` gespiegelt wird – gespeichert wird in jedem Fall.

Zustandsübersicht

Der Baustein ist zustandslos im Sinne einer eigenen ECC: Er durchläuft bei `INIT` einmalig Initialisierung + Erstlesen, danach reagiert er auf jedes `AX_IN`-Adapterereignis mit einem NVS-Schreibzugriff und optional einer Bestätigung über `AX_OUT`.

Anwendungsszenarien

- Persistieren einzelner `BOOL`-Konfigurationswerte (z. B. Freigabe-Flags, Betriebsartenschalter) auf dem ESP32
- Aufbau eines Lese-/Schreibpaars, bei dem Lese- und Schreibseite an unterschiedliche Bausteine angeschlossen werden sollen
- Wiederherstellen eines zuletzt gespeicherten `BOOL`-Werts nach einem Neustart über `AX_OUT`

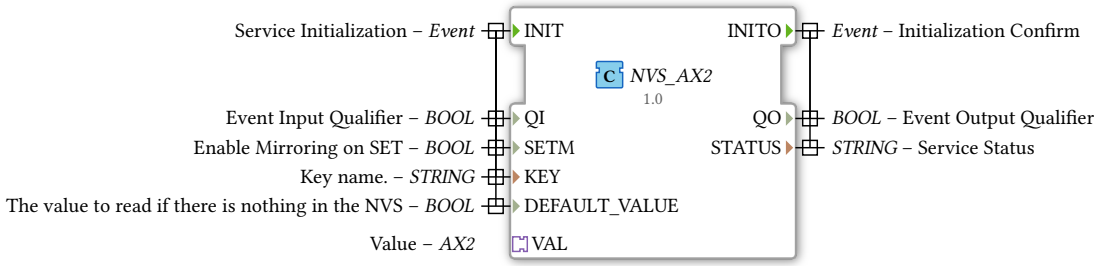
⚖️ Vergleich mit ähnlichen Bausteinen

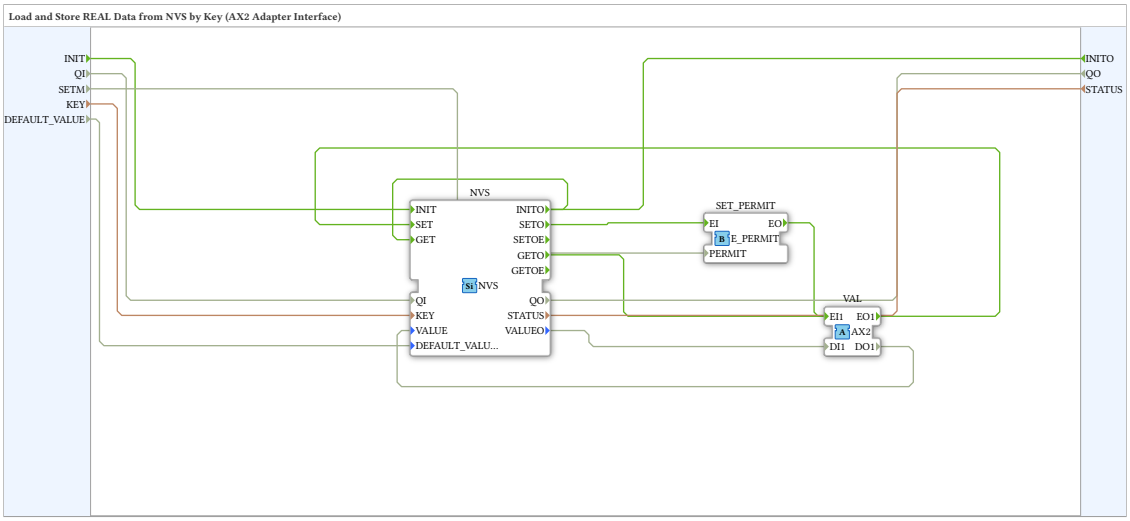
Vergleich mit NVS_AB, das dieselbe Architektur für `BYTE`- statt `BOOL`-Werte nutzt, sowie mit INI_AX, das dieselbe Schnittstelle für die textbasierte `settings.ini` statt für den NVS-Flash-Speicher bereitstellt.

Fazit

`NVS_AX` bietet einen einfachen, unidirektional verdrahteten Zugriff auf einen persistenten `BOOL`-Wert im ESP32-NVS und eignet sich überall dort, wo Lese- und Schreibpfad getrennt geführt werden sollen.

NVS_AX2





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **NVS_AX2** dient dem Laden und Speichern von REAL-Daten im nichtflüchtigen Speicher (NVS) eines ESP32 über einen Adapter. Er kapselt die Initialisierung und den Zugriff auf einen einzelnen NVS-Eintrag, der über einen Schlüssel (KEY) identifiziert wird. Der FB bietet eine initialisierende Ereignisschnittstelle und eine bidirektionale Adapter-Schnittstelle, über die extern auf den gespeicherten Wert zugegriffen werden kann. Der Baustein ist nach IEC 61499 modelliert und für den Einsatz in verteilten Automatisierungssystemen optimiert.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar	Mitgeführte Variablen
`INIT`	EInit	Service Initialization	QI, SETM, KEY, DEFAULT_VALUE

Ereignis-Ausgänge

Ereignis	Typ	Kommentar	Mitgeführte Variablen
`INITO`	EInit	Initialization Confirm	QO, STATUS

Daten-Eingänge

Name	Typ	Kommentar
`QI`	BOOL	Event Input Qualifier (Freigabe der Initialisierung)
`SETM`	BOOL	Enable Mirroring on SET: Bei TRUE wird die Bestätigung eines SET (SETO) zusätzlich am Ausgangsadapter ausgegeben (gespiegelt)
`KEY`	STRING	Schlüsselname für den NVS-Eintrag
`DEFAULT_VALUE`	BOOL	Standardwert, falls der Schlüssel im NVS nicht existiert (Voreinstellung: FALSE)

Daten-Ausgänge

Name	Typ	Kommentar
`QO`	BOOL	Event Output Qualifier

STATUS	STRING	Dienststatus

Adapter

Adapter	Typ	Kommentar
VAL	adapter::types::bidirectional::AX2 (Socket)	Wert (REAL)

Der Adapter **VAL** stellt eine bidirektionale Verbindung zu einem externen Baustein her. Über diesen Adapter kann der gespeicherte REAL-Wert gelesen und geschrieben werden. Der Adaptertyp AX2 definiert ein Ereignis- und Datenprotokoll für den Datentransfer.

Funktionsweise

1. **Initialisierung**: Ein Ereignis am Eingang INIT löst die Initialisierung des internen NVS-Bausteins aus. Dabei werden QI, SETM, KEY und DEFAULT_VALUE übergeben. Nach erfolgreicher Initialisierung wird das Ereignis INITO am Ausgang gesendet. Gleichzeitig wird automatisch ein Lesevorgang (GET) für den angegebenen Schlüssel angestoßen.
2. **Wert auslesen**: Der gelesene Wert wird über den Adapter VAL als DI1 (Datenausgang des Adapters) ausgegeben und ein Ereignis EI1 am Adapter ausgelöst. Der Ausgang QO und STATUS spiegeln das Ergebnis des Lesevorgangs wider.
3. **Wert speichern**: Ein externer Baustein kann über den Adapter VAL ein Ereignis EO1 senden, um einen neuen Wert (bereitgestellt als DO1) in den NVS zu schreiben. Der interne NVS-Baustein führt dann einen SET-Vorgang durch und bestätigt über SETO. Anschließend wird das Ereignis EI1 am Adapter ausgelöst, um dem Sender den Abschluss zu signalisieren.
4. **Statussignalisierung**: Die Ausgänge QO und STATUS werden nach jedem Lese- oder Schreibvorgang aktualisiert und mit dem Ereignis INITO ausgegeben.

Technische Besonderheiten

- **Persistenz auf ESP32**: Der FB nutzt den NVS (Non-Volatile Storage) des ESP32-Mikrocontrollers, um Daten dauerhaft zu speichern. Die Daten bleiben auch nach einem Neustart erhalten.
- **Adapter-basierte Kommunikation**: Die Schnittstelle zum Anwender erfolgt über einen bidirektionalen Adapter (AX2), der eine flexible und standardisierte Anbindung an andere Bausteine ermöglicht. Der Datentyp ist auf REAL festgelegt.
- **Fehlerbehandlung**: Über den Ausgang STATUS können Fehlermeldungen (z. B. NVS-Fehler, fehlender Speicherplatz) abgefragt werden. Der Ausgang QO zeigt den Erfolg der Operation an.
- **Standardkonformität**: Der FB ist gemäß IEC 61499 implementiert und verwendet die Bausteinklasse logiBUS::storage::esp32_nvs::NVS.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten, jedoch ergibt sich folgender logischer Ablauf:

1. **Idle**: Kein INIT-Ereignis empfangen.
2. **Initialisierend**: Nach INIT – der interne NVS-Baustein wird gestartet und ein GET ausgeführt.
3. **Bereit**: Nach INITO – der Baustein wartet auf Lese-/Schreibanforderungen über den Adapter.
4. **Schreibvorgang**: Ein EO1-Ereignis am Adapter löst einen SET-Vorgang aus.
5. **Lesen**: Ein erneutes Lesen kann nur durch erneutes INIT oder interne Logik erfolgen (der Baustein liest einmalig beim Start und danach nur bei Schreibvorgängen).

Anwendungsszenarien

- **Konfigurationsspeicherung**: Speichern von Benutzereinstellungen (z. B. Sollwerte, Betriebsmodi) auf einem ESP32-Gerät.
- **Kalibrierdaten**: Dauerhafte Ablage von Kalibrierwerten für Sensoren.
- **Zustandsspeicherung**: Speichern des letzten Betriebszustands (z. B. Zählerstände, Schaltstellungen) bei Spannungsausfall.
- **Datenprotokollierung**: Gezieltes Abspeichern einzelner Messwerte über den Adapter.

Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu einfachen Speicherbausteinen (z. B. CTUO oder generischen R_TRIG-Bausteinen) ist NVS_AX2 speziell für die persistente Speicherung auf ESP32-Hardware ausgelegt. Während andere FBs

nur flüchtige Daten verarbeiten, gewährleistet dieser Baustein einen dauerhaften Datenerhalt. Die Verwendung eines Adapters bietet mehr Flexibilität als ein fester Ein-/Ausgang, da der Datentyp und die Kommunikationsrichtung klar definiert sind. Im Vergleich zu direkten NVS-API-Aufrufen in IEC 61499 vereinfacht der FB die Integration durch die vollständig deklarative Beschreibung.

Fazit

Der Funktionsblock `NVS_AX2` ist eine effiziente und standardkonforme Lösung zum persistenten Speichern eines REAL-Werts im NVS eines ESP32. Die klare Schnittstellenstruktur mit Ereignissteuerung und bidirektionalem Adapter erleichtert die Einbindung in komplexe Automatisierungsprojekte. Dank der integrierten Initialisierung und Fehlerprotokollierung ist der Baustein robust und für den industriellen Einsatz geeignet.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 ESP32 & ESP32-S3 DevKit auf ms-muc-docs.de
- 🌐 MCU vs. MPU Vergleichsguide auf ms-muc-docs.de

INI

🎧 Podcast

- Der E_CTU in der IEC 61499: Ereignisgesteuertes Zählen und warum der Minimalist im Maschinenbau überzeugt
- From „Mass Errors“ to Masterpiece: Streamlining Industrial Software by Eliminating Mapping
- Das Kettenmonster erwacht: Lanz Bulldog Raupe – Die faszinierende Wiederbelebung des 10-Liter-Glühkopf-Arbeitstiers nach 25 Jahren Stillstand
- Meister des Wandels: Wie der Lehrplan für Land- und Baumaschinenmechatroniker lebenslanges Lernen neu definiert
- Miniware TS101: Das mobile Löt-Multitalent – Stärken, Schwächen und die USB-C Revolution

Einleitung

Der INI-Funktionsblock ermöglicht das Laden und Speichern von Daten in einer `settings.ini`-Datei. Er bietet eine einfache Schnittstelle, um Werte anhand eines Sektionsnamens (SECTION) und eines Schlüssels (KEY) persistent zu verwalten. Der Block ist besonders nützlich für die Konfigurationsverwaltung in 4diac FORTE Applikationen, z.B. für das Setzen und Abrufen von Parametern wie IP-Adressen, Schwellwerten oder Betriebsmodi.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Dienst-Initialisierung. Wird bei Start oder Neukonfiguration verwendet. Löst die Initialisierung des Blocks und optional das erste Lesen/Schreiben eines Wertes aus.
- **SET**: Setzt den Wert für den konfigurierten Schlüssel in der INI-Datei.
- **GET**: Liest den Wert für den konfigurierten Schlüssel aus der INI-Datei.

Ereignis-Ausgänge

- **INITO**: Bestätigung der Initialisierung. Wird nach Verarbeitung des INIT-Ereignisses ausgelöst.
- **SETO**: Bestätigung des Schreibvorgangs. Wird nach Verarbeitung des SET-Ereignisses ausgelöst.
- **GETO**: Bestätigung des Lesevorgangs. Wird nach Verarbeitung des GET-Ereignisses ausgelöst.

Daten-Eingänge

- **QI (BOOL)**: Ereignis-Eingangs-Qualifizierer. Steuert bei INIT, ob der Vorgang aktiviert (`TRUE`) oder deaktiviert (`FALSE`) wird.
- **SECTION (STRING)**: Der Name des Sektionsabschnitts in der `settings.ini`-Datei.
- **KEY (STRING)**: Der Name des Schlüssels innerhalb des angegebenen Sektionsabschnitts.
- **VALUE (ANY_ELEMENTARY)**: Der zu schreibende Wert. Dieser Datentyp kann jeden elementaren Typ (z.B. BOOL, INT, REAL, STRING) annehmen.
- **DEFAULT_VALUE (ANY_ELEMENTARY)**: Der Standardwert, der zurückgegeben wird, wenn der angeforderte Schlüssel in der INI-Datei nicht existiert.

Daten-Ausgänge

- **QO (BOOL)**: Ereignis-Ausgangs-Qualifizierer. Gibt den Status der Operation wieder (`TRUE` für Erfolg, `FALSE` für Fehler oder Deaktivierung).
- **STATUS (STRING)**: Detaillierter Status der ausgeführten Operation (z.B. „OK“, „FEHLER“).
- **VALUEO (ANY_ELEMENTARY)**: Der gelesene oder geschriebene Wert. Bei GET ist dies der aus der Datei gelesene oder der DEFAULT_VALUE.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der INI-Block agiert als Schnittstelle zwischen der IEC 61499-Applikation und einer persistenten `settings.ini`-Datei auf dem Dateisystem. Bei einem `INIT`-Ereignis mit `QI=TRUE` wird der Block für die angegebene `SECTION` und `KEY` konfiguriert. Optional kann dabei sofort ein Wert (`VALUE`) geschrieben oder mit `DEFAULT_VALUE` ein Standard gesetzt werden.

Ein `SET`-Ereignis schreibt den aktuellen Wert des `VALUE`-Eingangs unter dem konfigurierten Schlüssel in die Datei. Ein `GET`-Ereignis liest den Wert aus der Datei. Wenn der Schlüssel nicht existiert, wird stattdessen der `DEFAULT_VALUE` zurückgegeben. Jede Operation (`INIT`, `SET`, `GET`) wird durch ein entsprechendes Ausgangsereignis (`INITO`, `SETO`, `GETO`) quittiert, wobei `QO`, `STATUS` und `VALUEO` den Ergebnisstatus anzeigen.

Technische Besonderheiten

- **Typflexibilität**: Die Daten-Ein-/Ausgänge `VALUE`, `DEFAULT_VALUE` und `VALUEO` verwenden den generischen Datentyp `ANY_ELEMENTARY`. Dies erlaubt die Speicherung verschiedenster elementarer Datentypen (z.B. Zahlen, Boolesche Werte, Strings) mit demselben Funktionsblock.
- **Persistenz**: Die Daten werden in textbasierten INI-Dateien gespeichert (`settings.ini` für beschreibbare Werte, `settingsReadOnly.ini` für schreibgeschützte Werkseinstellungen).
- **Schreibschutz & Werkseinstellungen**: Schlüssel in `settingsReadOnly.ini` überschreiben Werte in `settings.ini` und lehnen Schreibversuche (`SET`) ab (`STATUS = "Key is read-only"`). Für Details siehe Schreibgeschützte Einstellungen (settingsReadOnly.ini).
- **Fehlerbehandlung**: Der `STATUS`-Ausgang bietet eine textuelle Rückmeldung über Erfolg oder Misserfolg der Operation, was das Debuggen erleichtert.
- **Package**: Der Block ist im Package `eclipse4diac::storage` enthalten.

Zustandsübersicht

Der Block besitzt keine explizite, komplexe Zustandsmaschine im nutzerzugänglichen Sinne. Sein Verhalten ist ereignisgesteuert: Nach der Initialisierung (`INIT`) wartet er auf `SET`- oder `GET`-Anfragen, verarbeitet diese und kehrt anschließend in einen wartenden Zustand zurück. Der interne Zustand (konfigurierter Sektion/Schlüssel) bleibt nach `INIT` erhalten.

Anwendungsszenarien

- **Konfigurationsmanagement**: Laden von Geräte-IP-Adressen, Kommunikationsports oder Netzwerkeinstellungen beim Start einer Applikation.
- **Parametrierung**: Speichern von anlagen- oder produktspezifischen Parametern (z.B. Sollwerte, Zeiten), die zwischen Neustarts erhalten bleiben sollen.
- **Betriebsdatenerfassung**: Persistentes Speichern einfacher Betriebszustände oder Zählerstände.
- **Benutzereinstellungen**: Verwalten von Spracheinstellungen oder anderen Benutzerpräferenzen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`E_SR` (SR-Flip-Flop) / `E_RS` (RS-Flip-Flop)**: Diese Blöcke speichern einen binären Zustand (`BOOL`) nur während der Laufzeit im Arbeitsspeicher. Der INI-Block speichert beliebige Typen dauerhaft auf nichtflüchtigem Speicher.
- **`E_DEMUX` / `E_MUX`**: Diese dienen der Ereignis- und Datenverteilung, nicht der persistenten Speicherung.
- **`FB_RETAIN` (aus `eclipse4diac::core`)** : Speichert Daten persistent, nutzt jedoch typischerweise ein gerätespezifisches, nicht direkt zugängliches Retain-Memory. Der INI-Block verwendet eine standardisierte, lesbare Textdatei.
- **Datenbank- oder File-IO-Blöcke**: Allgemeinere Blöcke für Dateizugriffe bieten mehr Flexibilität, sind aber auch komplexer in der Handhabung. Der INI-Block ist eine spezialisierte, einfache Lösung für den häufigen Anwendungsfall der Schlüssel-Wert-Speicherung.

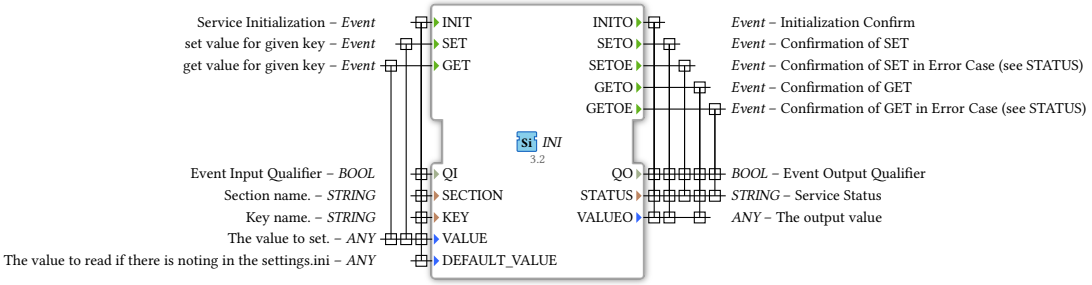
Fazit

Der INI-Funktionsblock ist ein praktisches und unkompliziertes Werkzeug zur persistenten Speicherung von Konfigurations- und Parametrierdaten in 4diac FORTE Applikationen. Seine Stärken liegen in der einfachen Bedienung über Sektion und Schlüssel, der Flexibilität durch den `ANY_ELEMENTARY`-Typ und der menschlichen Lesbarkeit der gespeicherten INI-Datei. Für Anwendungen, die eine einfache, dauerhafte Schlüssel-Wert-Datenbank benötigen, ist er eine ausgezeichnete Wahl.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

INI



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **INI** dient dem sicheren Lesen und Schreiben von Konfigurationsdaten in einer `settings.ini`-Datei. Er ermöglicht den Zugriff auf Werte über einen Abschnitts- und Schlüsselnamen und stellt sowohl Initialisierungs- als auch Lese- und Schreiboperationen als getrennte Ereignisse bereit. Durch die Verwendung generischer Datentypen (`ANY`) ist er flexibel einsetzbar.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Initialisiert den Baustein und öffnet die INI-Datei
SET	Event	Schreibt den Wert von `VALUE` für den aktuellen Schlüssel
GET	Event	Liest den Wert des aktuellen Schlüssels

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Bestätigung der Initialisierung
SETO	Event	Bestätigung des Schreibvorgangs
SETOE	Event	Bestätigung von SET im Fehlerfall (siehe `STATUS`)
GETO	Event	Bestätigung des Lesevorgangs
GETOE	Event	Bestätigung von GET im Fehlerfall (siehe `STATUS`)

Daten-Eingänge

Name	Typ	Kommentar
QI	BOOL	Event-Input-Qualifier (steuert die Ausführung)

SECTION	STRING	Name des Abschnitts in der INI-Datei
KEY	STRING	Schlüsselname innerhalb des Abschnitts
VALUE	ANY	Wert, der gesetzt werden soll (bei SET)
DEFAULT_VALUE	ANY	Standardwert, falls Schlüssel nicht existiert (bei GET)

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Event-Output-Qualifier (zeigt Erfolg/Misserfolg an)
STATUS	STRING	Statusinformationen (z. B. Fehlertext)
VALUEO	ANY	Ausgelesener Wert (bei GET) oder bestätigter gesetzter Wert

Adapter

Keine Adapter definiert.

Funktionsweise

Der Baustein arbeitet auf einer Datei `settings.ini`, die eine einfache Schlüssel-Wert-Struktur in Abschnitten (`[Section]`) enthält. Die Nutzung erfolgt in drei Schritten:

1. **INIT** – Mit dem INIT-Ereignis werden die Parameter `SECTION` und `KEY` übergeben. Der Baustein öffnet oder lädt die INI-Datei und bereitet die Lese-/Schreiboperationen vor. Eine erfolgreiche Initialisierung wird durch das INITO-Ereignis signalisiert.
2. **SET** – Nach der Initialisierung kann mit SET ein neuer Wert für den zuvor definierten Schlüssel geschrieben werden. Der übergebene `VALUE` wird in `settings.ini` gespeichert, sofern der Schlüssel nicht schreibgeschützt ist. Handelt es sich um einen schreibgeschützten Parameter aus `settingsReadOnly.ini`, wird der Schreibversuch abgelehnt, es feuert das Fehlerereignis **`SETOE`** und `STATUS` meldet `"Key is read-only"`.
3. **GET** – Mit GET wird der aktuelle Wert des Schlüssels gelesen. Bei schreibgeschützten Parametern wird stets der Wert aus `settingsReadOnly.ini` zurückgegeben. Existiert ein normaler Schlüssel nicht in `settings.ini`, wird der über `DEFAULT_VALUE` angegebene Standardwert zurückgegeben. Der gelesene Wert erscheint am Ausgang `VALUEO`.

Die Ausführung jedes Ereignisses wird durch den jeweiligen Ausgang (INITO, SETO/SETOE, GETO) quittiert. Dabei zeigt `QO` an, ob die Operation erfolgreich war, und `STATUS` liefert eine textuelle Rückmeldung.

Technische Besonderheiten

- **Generische Datentypen** – `VALUE` und `DEFAULT_VALUE` sind als `ANY` deklariert. Der Baustein kann daher mit verschiedenen Datentypen (z. B. BOOL, INT, REAL, STRING) arbeiten, sofern die Laufzeitumgebung diese unterstützt.
- **Schreibschutz (Werkseinstellungen)** – Das System unterstützt neben `settings.ini` auch die schreibgeschützte Vorgabedatei `settingsReadOnly.ini`. Parameter aus dieser Datei überschreiben veränderbare Werte und lehnen Schreibversuche mit `STATUS = "Key is read-only"` ab. Siehe auch Schreibgeschützte Einstellungen (`settingsReadOnly.ini`).
- **Event-basierte Steuerung** – Alle Aktionen werden über Ereignisse ausgelöst. Der Baustein ist somit vollständig in ein ereignisgesteuertes System (z. B. IEC 61499) integrierbar.
- **Fehlerbehandlung** – Der Ausgangsqualifier `QO` und der `STATUS`-String ermöglichen eine einfache Fehlerdiagnose (z. B. bei schreibgeschützten Schlüsseln, fehlender Datei oder ungültigem Abschnitt).

Zustandsübersicht

Der Funktionsblock durchläuft folgende Hauptzustände:

- **Idle** – Warten auf INIT.
- **Initialized** – Nach erfolgreichem INIT; bereit für SET oder GET.
- **Error** – Bei fehlgeschlagener Initialisierung oder Operation; nur ein erneutes INIT kann den Zustand verlassen.

Die Zustandsübergänge werden durch die Ereignisse INIT, SET und GET sowie die entsprechenden Bestätigungsausgänge gesteuert.

Anwendungsszenarien

- **Maschinenkonfiguration** – Speichern und Abrufen von Parametern wie Geschwindigkeit, Temperaturgrenzen oder Betriebsmodi in einer zentralen INI-Datei.
- **Anlageninitialisierung** – Einmaliges Laden von Einstellungen beim Start der Steuerung über das INIT-Ereignis.
- **Online-Parametrierung** – Dynamisches Ändern von Werten während des Betriebs mittels SET und sofortiges Auslesen mit GET.

Vergleich mit ähnlichen Bausteinen

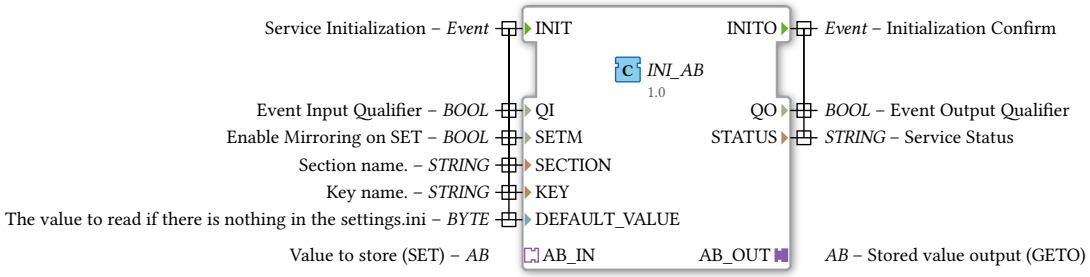
Baustein	Datenquelle	Typunterstützung	Ereignisschnittstelle
INI	settings.ini	ANY (generisch)	INIT, SET, GET
FILE_READ	Textdatei	STRING	REQ, CNF
PARAM_DB	Datenbank	typspezifisch	komplexer

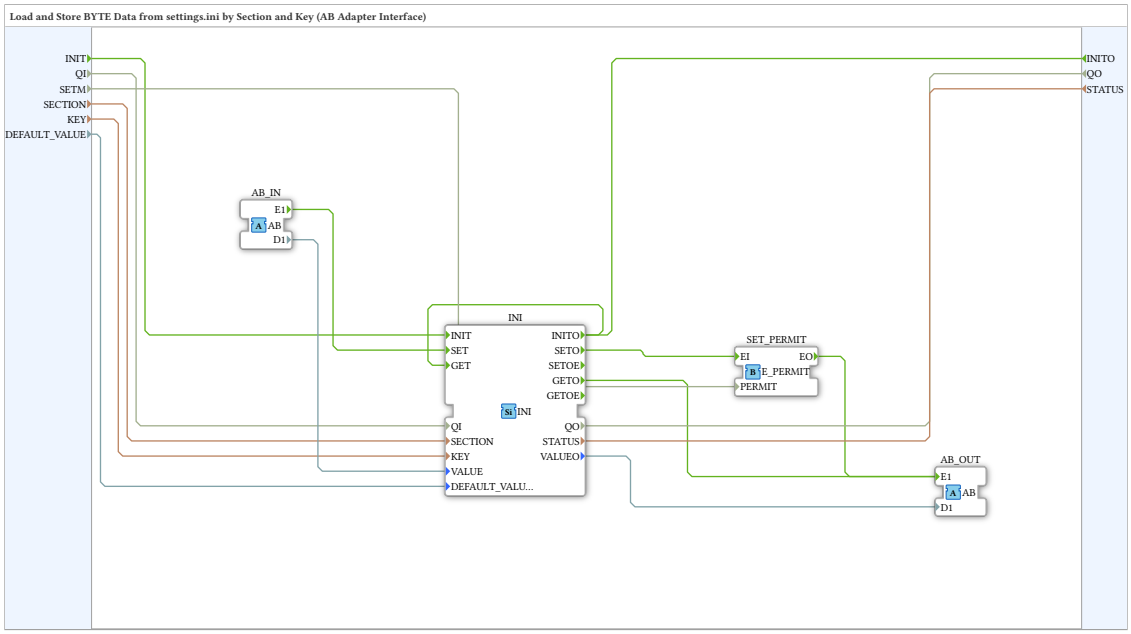
Der INI-Baustein punktet durch seine Einfachheit und die direkte Anbindung an INI-Dateien, ohne zusätzliche Treiber oder Datenbanken. Andere Bausteine bieten entweder mehr Flexibilität (Datenbank) oder sind auf reine Text-Ein-/Ausgabe beschränkt.

Fazit

Der INI-Funktionsblock stellt eine kompakte und zuverlässige Lösung für die dauerhafte Speicherung von Konfigurationsdaten in IEC-61499-Systemen dar. Seine generische Datentypdefinition und die klare Ereignissteuerung machen ihn vielseitig einsetzbar. Durch die integrierte Fehlerbehandlung eignet er sich sowohl für Entwicklungs- als auch für Produktionsumgebungen.

INI_AB





• – ○ • – ○ • – ○ •

Einleitung

`INI_AB` lädt und speichert einen `BYTE`-Wert in der `settings.ini`-Datei, adressiert über Abschnitt (`SECTION`) und Schlüssel (`KEY`). Er ist das `settings.ini`-Gegenstück zu NVS_AB und verwendet dieselbe Zwei-Adapter-Architektur mit unidirektionalen `AB`-Adaptern.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisiert den Baustein und liest den unter `SECTION`/`KEY` hinterlegten Wert (bzw. `DEFAULT_VALUE`, falls noch keiner existiert). Liefert `QI`, `SETM`, `SECTION`, `KEY`, `DEFAULT_VALUE`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Initialisierung und den ersten Lesezugriff, liefert `QO` und `STATUS`.

Daten-Eingänge

- **QI** (BOOL): Qualifizierer des `INIT`-Ereignisses.
- **SETM** (BOOL): Gibt das Spiegeln (Rückmeldung) beim Schreiben (`SET`) frei.
- **SECTION** (STRING): Name des Abschnitts in der `settings.ini`.
- **KEY** (STRING): Name des Schlüssels innerhalb des Abschnitts.
- **DEFAULT_VALUE** (BYTE): Wert, der geliefert wird, wenn unter `SECTION`/`KEY` noch nichts in der `settings.ini` gespeichert ist.

Daten-Ausgänge

- **QO** (BOOL): Erfolgsqualifizierer, mit `INITO` sowie den internen Adapterbestätigungen mitgeliefert.
- **STATUS** (STRING): Statusmeldung des Dateizugriffs.

Adapter

- **AB_IN** (Socket, Typ `AB`): Neuer zu speichernder Wert – ein Adapterereignis auf `AB_IN.E1` löst einen `SET` im internen INI-Baustein aus, `AB_IN.D1` liefert den zu schreibenden Wert.
- **AB_OUT** (Plug, Typ `AB`): Gibt den zuletzt gelesenen Wert aus – sowohl nach dem automatischen Erstlesen bei `INIT` als auch nach jedem bestätigten Schreibvorgang (`AB_OUT.D1`/`AB_OUT.E1`).

Funktionsweise

`INI_AB` kapselt intern eine Instanz von `eclipse4diac::storage::INI` sowie ein `E_PERMIT`-Gatter (`SET_PERMIT`). Bei `INIT` wird die interne INI-Instanz initialisiert (`INI.INIT`) und anschließend automatisch

ein GET ausgelöst, das den unter SECTION/KEY gespeicherten Wert (oder DEFAULT_VALUE) liefert; das Ergebnis wird über AB_OUT ausgegeben, INITO bestätigt den Abschluss.

Zum Schreiben liefert der an AB_IN angeschlossene Baustein ein Adapterereignis (AB_IN.E1) mit dem neuen Wert (AB_IN.D1); dies löst INI.SET aus. Ob die erfolgreiche Speicherung zusätzlich über AB_OUT zurückgemeldet wird, steuert SETM: Ist SETM gesetzt, gibt SET_PERMIT das AB_OUT.E1-Ereignis nach erfolgreichem SET frei („Mirroring“).

Technische Besonderheiten

- Reines Composite-FB ohne eigene ECC – die eigentliche Dateizugriffslogik steckt in der internen INI-Instanz, INI_AB verdrahtet nur Ereignisse und Daten zwischen ihr und den beiden AB-Adaptern.
- Zwei getrennte, unidirektionale Adapter (AB_IN/AB_OUT) statt eines einzigen bidirektionalen – dadurch lassen sich Lese- und Schreibpfad an unterschiedliche Gegenstellen anschließen, was mit INI_AB2 (ein gemeinsamer AB2-Socket) nicht möglich ist.
- Adressierung über SECTION **und** KEY (statt nur KEY bei den NVS_*-Bausteinen) entspricht dem klassischen INI-Dateiformat.

Zustandsübersicht

Der Baustein ist zustandslos im Sinne einer eigenen ECC: Er durchläuft bei INIT einmalig Initialisierung + Erstlesen, danach reagiert er auf jedes AB_IN-Adapterereignis mit einem INI-Schreibzugriff und optional einer Bestätigung über AB_OUT.

Anwendungsszenarien

- Persistieren einzelner BYTE-Konfigurationswerte (z. B. Betriebsmodus, Kanalauswahl) in einer menschenlesbaren settings.ini
- Aufbau eines Lese-/Schreibpaars, bei dem Lese- und Schreibseite an unterschiedliche Bausteine angeschlossen werden sollen
- Konfigurationsdateien, die auch außerhalb der Steuerung (z. B. per Texteditor) einsehbar/änderbar sein sollen

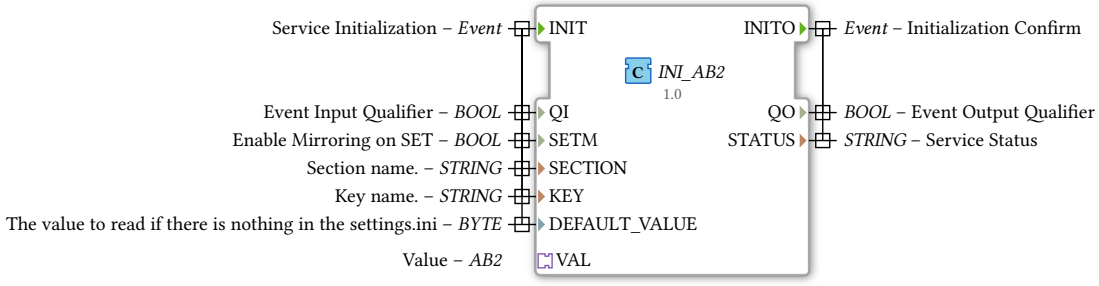
⚖️ Vergleich mit ähnlichen Bausteinen

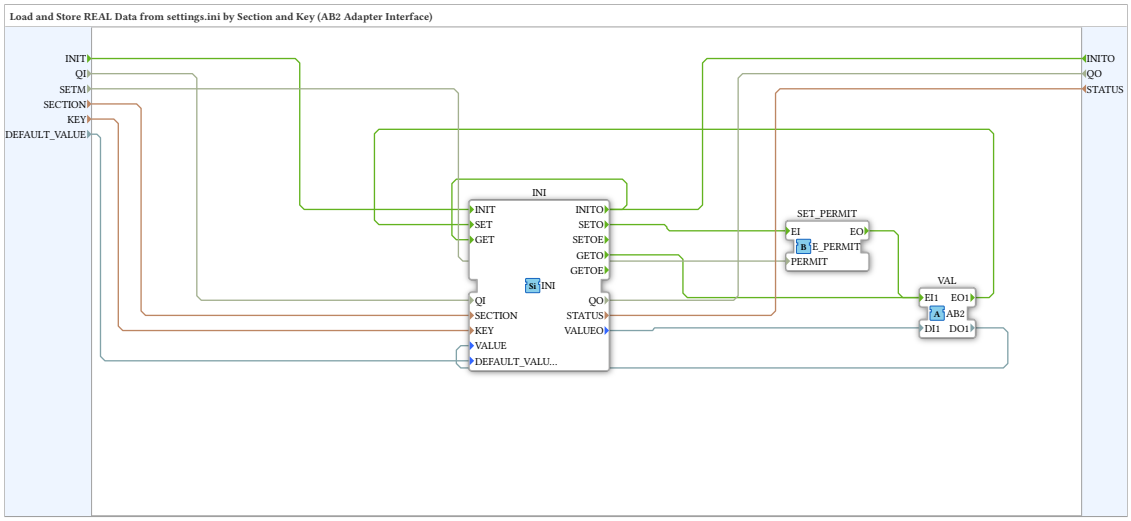
Vergleich mit INI_AB2, das denselben Zweck über einen einzigen bidirektionalen AB2-Adapter statt zweier unidirektionaler AB-Adapter erfüllt, sowie mit NVS_AB, das dieselbe Schnittstelle für den ESP32-NVS-Flash-Speicher statt für die settings.ini bereitstellt.

Fazit

INI_AB bietet einen einfachen, unidirektional verdrahteten Zugriff auf einen persistenten BYTE-Wert in der settings.ini und eignet sich überall dort, wo Lese- und Schreibpfad getrennt geführt werden sollen.

INI_AB2





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `INI_AB2` dient zum Lesen und Schreiben von BYTE-Werten aus oder in eine `settings.ini`-Datei. Der Zugriff erfolgt über eine Kombination aus Abschnittsnamen (Section) und Schlüsselnamen (Key). Die Kommunikation mit der Außenwelt erfolgt über einen bidirektionalen **AB2-Adapter**, der es ermöglicht, Werte aus der INI-Datei an andere Bausteine weiterzugeben oder Werte von anderen Bausteinen entgegenzunehmen und in der Datei zu speichern. Der Baustein wurde gemäß der Norm IEC 61499-2 entwickelt.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Beschreibung
INIT	EInit	Initialisierungsereignis; startet den Lese- oder Schreibvorgang abhängig von der angeschlossenen Adapterlogik. Die mitgegebenen Daten legen die Spiegelung bei SET (`SETM`), Section, Key und einen Standardwert fest.

Ereignis-Ausgänge

Ereignis	Typ	Beschreibung
INITO	EInit	Bestätigung der Initialisierung. Wird ausgelöst, nachdem der Lese-/Schreibzugriff abgeschlossen ist.

Daten-Eingänge

Variable	Typ	Beschreibung
QI	BOOL	Qualifizierer für den Ereigniseingang; steuert die Ausführung.
SETM	BOOL	Aktiviert die Spiegelung (Mirroring) bei SET.
SECTION	STRING	Name des Abschnitts (Section) in der `settings.ini`.
KEY	STRING	Name des Schlüssels (Key) innerhalb des Abschnitts.
DEFAULT_VALUE	BYTE	Der Wert, der zurückgegeben wird, falls der Schlüssel in der `settings.ini` nicht existiert.

Daten-Ausgänge

Variable	Typ	Beschreibung
QO	BOOL	Ausgangsqualifizierer; zeigt erfolgreiche Ausführung an.
STATUS	STRING	Statusmeldung des Dienstes (z. B. Fehlertexte oder „OK“).

Adapter

Adapter	Typ	Beschreibung
VAL	adapter::types::bidirectional::AB2	Bidirektionaler Adapter zum Austausch von BYTE-Werten. Der eingehende Wert (`DO1`) wird in die INI-Datei geschrieben; der ausgehende Wert (`DI1`) wird aus der INI-Datei gelesen und weitergereicht.

Funktionsweise

Der `INI_AB2`-Baustein enthält intern eine Instanz des Funktionsblocks `eclipse4diac::storage::INI`. Die Kommunikation mit der `settings.ini` erfolgt ausschließlich über diese interne Instanz.

1. **Initialisierung (Lesevorgang):** Ein INIT-Ereignis am Eingang setzt die internen Parameter (Section, Key, Default-Wert) und aktiviert die interne INI-Instanz (`INI.INIT`). Nach erfolgreichem Einlesen wird der gelesene Wert über den Adapterausgang (`VAL.DI1`) an den angeschlossenen Baustein gesendet. Gleichzeitig werden die Ausgänge `INITO`, `QO` und `STATUS` aktualisiert.
2. **Schreibvorgang:** Der angeschlossene Baustein kann über den Adaptereingang (`VAL.DO1`) einen neuen Wert bereitstellen. Sobald das Ereignis `VAL.EO1` eintrifft, wird der Wert in der INI-Instanz gespeichert (`INI.SET`). Die Bestätigung (`INI.SETO`) wird über `VAL.EI1` an den Adapter zurückgegeben.
3. **Zyklisches Auslesen:** Nach jedem INIT-Vorgang wird automatisch der interne GET-Befehl ausgelöst, sodass der aktuelle Wert aus der Datei gelesen und über den Adapter ausgegeben wird.

Die Verschaltung stellt sicher, dass sowohl Lese- als auch Schreibzugriffe über den Adapter bidirektional abgewickelt werden können. Der Baustein realisiert somit eine persistente Datenhaltung mit externer Steuerung.

Technische Besonderheiten

- **Bidirektionale Adapteranbindung:** Der AB2-Adapter erlaubt sowohl das Auslesen als auch das Beschreiben der INI-Datei über eine einzige Verbindung.
- **Standardwert:** Falls ein angegebener Schlüssel in der `settings.ini` nicht vorhanden ist, wird der über `DEFAULT_VALUE` übergebene Wert verwendet und als gelesener Wert ausgegeben.
- **Verwendung von `eclipse4diac::storage::INI`:** Der Baustein kapselt die eigentliche INI-Zugriffslogik und stellt eine robuste Fehlerbehandlung bereit (Statusausgabe).
- **IEC 61499-2 konform:** Der Baustein folgt dem Standard für Funktionsblöcke, was Interoperabilität und Wiederverwendbarkeit erhöht.

Zustandsübersicht

Der `INI_AB2`-Baustein selbst besitzt keine expliziten Zustände (keine ECC-Zustände), da die Logik vollständig durch die interne INI-Instanz und die Ereignis-/Datenflüsse abgebildet wird. Das Zusammenspiel kann jedoch als einfacher sequenzieller Ablauf beschrieben werden:

1. **Ruhezustand** – Warten auf INIT oder Daten vom Adapter.
2. **Initialisieren** – INIT verarbeitet, interne INI wird gestartet.
3. **Lesen** – INI liest Wert, gibt ihn über Adapter aus.
4. **Schreiben** – Wert vom Adapter wird in INI gespeichert.
5. **Rückkehr** – Bestätigung über INITO oder Adapter-Ereignis.

Anwendungsszenarien

- **Konfigurationsmanagement:** Speichern und Abrufen von Maschinenparametern (z. B. Solltemperaturen, Grenzwerte) in einer zentralen `settings.ini`.
- **Anlagenparametrierung:** Mehrere Steuerungskomponenten greifen über den AB2-Adapter auf dieselbe INI-Datei zu und können so Parameter austauschen.

- **Datenaustausch zwischen heterogenen Systemen:** Der standardisierte Adapter ermöglicht die Integration in beliebige 4diac-Architekturen.

Vergleich mit ähnlichen Bausteinen

- **INI_AB (unidirektional):** Nutzt einen unidirektionalen Adapter, der entweder nur Lesen oder nur Schreiben erlaubt. `INI_AB2` ist durch die bidirektionale Schnittstelle flexibler.
- **INI_CFB (Composite FB):** Ein ähnlicher Baustein ohne Adapter, der direkt über Ereignisse und Daten angesprochen wird. `INI_AB2` abstrahiert die Verbindung und ermöglicht modulare Kopplung.
- **eclipse4diac::storage::INI (direkt):** Der Basisbaustein bietet keine Adapterschnittstelle; `INI_AB2` kapselt diesen und fügt die bidirektionale Kommunikation hinzu.

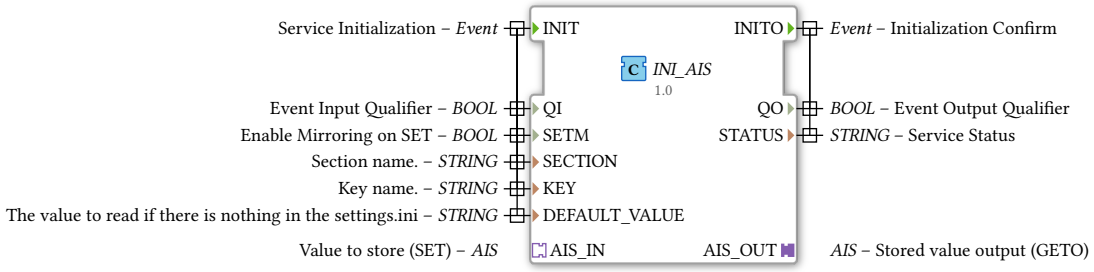
Fazit

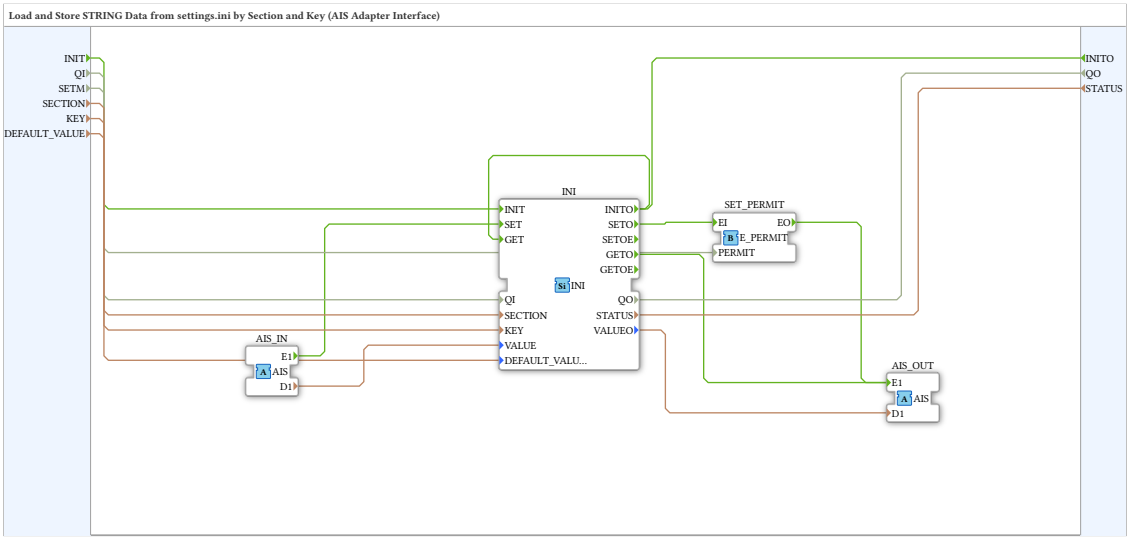
Der `INI_AB2`-Funktionsblock ist eine praktische Erweiterung für IEC 61499-basierte Automatisierungslösungen. Er vereinfacht den Zugriff auf INI-Konfigurationsdateien durch eine standardisierte Adapterschnittstelle und ermöglicht sowohl Lese- als auch Schreibvorgänge. Die einfache Handhabung und die klare Trennung von Logik und Schnittstelle machen ihn zu einem nützlichen Werkzeug für modulare und wiederverwendbare Steuerungsanwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

INI_AIS





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **INI_AIS** dient zum Laden und Speichern von Zeichenketten (STRING) aus einer `settings.ini`-Datei. Dabei wird über einen Abschnittsnamen (Section) und einen Schlüssel (Key) auf den gewünschten Wert zugegriffen. Die Schnittstelle ist über unidirektionale AIS-Adapter realisiert, sodass der Baustein sowohl Werte lesen als auch schreiben kann.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung
INIT	EInit	Service-Initialisierung

Ereignis-Ausgänge

Name	Typ	Beschreibung
INITO	EInit	Bestätigung der Initialisierung

Daten-Eingänge

Name	Typ	Beschreibung
QI	BOOL	Qualifikator für den Ereigniseingang
SETM	BOOL	Enable Mirroring on SET: Bei TRUE wird die Bestätigung eines SET (SETO) zusätzlich am Ausgangsadapter ausgegeben (gespiegelt)
SECTION	STRING	Name des Abschnitts in der Konfigurationsdatei
KEY	STRING	Name des Schlüssels im angegebenen Abschnitt
DEFAULT_VALUE	STRING	Standardwert, falls der Eintrag nicht existiert

Daten-Ausgänge

Name	Typ	Beschreibung
QO	BOOL	Qualifikator für den Ereignisausgang
STATUS	STRING	Statusmeldung (z. B. Fehlertext)

Adapter

Name	Typ	Richtung	Beschreibung
AIS_OUT	adapter::types::unidirectional::AIS	Plug	Ausgang für den gelesenen Wert (GETO)
AIS_IN	adapter::types::unidirectional::AIS	Socket	Eingang für den zu speichernden Wert (SET)

Funktionsweise

Der Baustein enthält intern eine Instanz des Funktionsblocks `INI` (aus der Bibliothek `eclipse4diac::storage`). Die Abläufe sind wie folgt:

1. **Initialisierung (INIT)**
   - Die Daten `QI`, `SETM`, `SECTION`, `KEY` und `DEFAULT_VALUE` werden an den internen INI-Baustein weitergeleitet.
   - Der INIT-Ereignisimpuls startet den INI-Baustein.
   - Nach Abschluss liefert INI das Ereignis `INITO` zurück, das nach außen als `INITO` des INI_AIS ausgegeben wird. Gleichzeitig werden die Ausgänge `QO` und `STATUS` übernommen.
   - Unmittelbar nach der Initialisierung wird im INI-Baustein automatisch der `GET`-Dienst ausgelöst, um den Wert aus der `settings.ini` zu lesen.
2. **Lesen eines Wertes (über AIS_OUT)**
   - Der gelesene Wert erscheint am Datenausgang `VALUEO` des INI-Bausteins und wird auf den Datenausgang `D1` des Adapter-Plugs `AIS_OUT` gelegt.

- Das Ereignis GET0 des INI-Bausteins wird auf den Ereigniseingang E1 von AIS_OUT übertragen, sodass der empfangende Baustein den Wert abholen kann.

3. **Schreiben eines Wertes (über AIS_IN)**
   - Ein externer Baustein kann über den Adapter-Socket AIS_IN einen Wert (Ereignis E1 mit Daten D1) an den INI_AIS senden.
   - Dieses Ereignis triggert den SET-Dienst des internen INI-Bausteins. Der ankommende Wert wird an den Dateneingang VALUE von INI übergeben.
   - Nach erfolgreichem Schreiben sendet INI das Ereignis SET0, welches an den Ereignisausgang E1 von AIS_OUT weitergeleitet wird. Der geschriebene Wert wird gleichzeitig auch über VALUE0 zurückgegeben, sodass der schreibende Baustein den Wert bestätigt bekommt.

Technische Besonderheiten

- **Interne Vernetzung:** Der Baustein nutzt den bereits existierenden INI-Baustein und kapselt dessen komplexe Schnittstelle in einer einfacheren, adapterbasierten Form.
- **Unidirektionale AIS-Adapter:** Der Datenaustausch erfolgt über AIS-Schnittstellen, die eine lose Kopplung zwischen den Bausteinen ermöglichen. Der AIS_IN-Socket ist für das Empfangen von Schreibaufträgen, der AIS_OUT-Plug für das Senden von Leseergebnissen zuständig.
- **Automatischer Lesezugriff:** Nach der Initialisierung wird sofort der gespeicherte Wert gelesen und über den Ausgang bereitgestellt – ohne separate GET-Anforderung.
- **Fehlerbehandlung:** Statusmeldungen werden über den STATUS-Ausgang kommuniziert (z.B. Datei nicht gefunden, Abschnitt/Schlüssel nicht vorhanden).

Zustandsübersicht

Der Baustein besitzt keine explizit programmierten Zustände. Der interne INI-Baustein übernimmt die Zustandsverwaltung. Aus Sicht des Benutzers ergeben sich folgende Betriebsphasen:

- **Initialisiert (nach INIT – INITO):** Der Baustein ist bereit, Werte zu lesen und zu schreiben. Der Wert des angegebenen Schlüssels wurde bereits einmal gelesen und steht am AIS_OUT-Ausgang zur Verfügung.
- **Schreibvorgang (über AIS_IN):** Ein externer Befehl löst einen Schreibzugriff aus. Nach Abschluss ist der neue Wert über AIS_OUT abrufbar.
- **Lesezugriff:** Erfolgt implizit nach jedem INIT und wird nicht explizit gesteuert.

Anwendungsszenarien

- **Persistente Konfigurationsdaten:** Speichern von Parametern wie Geräteadressen, Grenzwerten oder Betriebsmodi in einer INI-Datei.
- **Plug-and-Play-fähige Module:** Module, die über den AIS-Adapter ihre Konfiguration lesen und bei Bedarf überschreiben – ohne feste Verkabelung der Datenpfade.
- **Initialisierungswerte:** Auslesen von Standardwerten beim Systemstart, die über die settings.ini-Datei angepasst werden können.

Vergleich mit ähnlichen Bausteinen

Baustein	Beschreibung
INI	Direkter Zugriff auf INI-Dateien mit separaten Ereignissen für GET und SET.
INI_AIS	Erweitert INI um eine adapterbasierte Schnittstelle, die das Koppeln anderer Bausteine vereinfacht und die Handhabung von Lese- und Schreiboperationen integriert.

- Im Gegensatz zum bloßen INI-Baustein bietet INI_AIS eine einheitliche, ereignisorientierte Schnittstelle für lesende und schreibende Komponenten.
- Die Verwendung von Adaptern ermöglicht eine flexible Verbindung in größeren Automatisierungsnetzwerken, ohne dass die Datenleitungen einzeln verdrahtet werden müssen.

Fazit

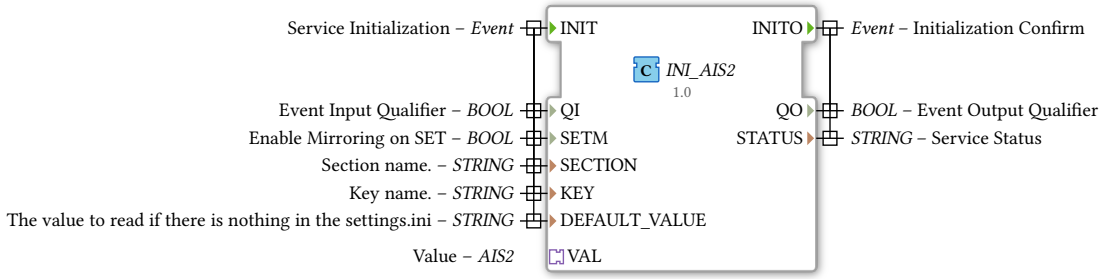
Der **INI_AIS**-Funktionsblock ist eine praktische Erweiterung des Basis-INI-Bausteins, der die Nutzung von Konfigurationsdateien in Automatisierungsprojekten deutlich vereinfacht. Durch die Integration von Lese- und Schreibvorgängen in einer einzigen Adapter-Schnittstelle wird der Systementwurf modularer

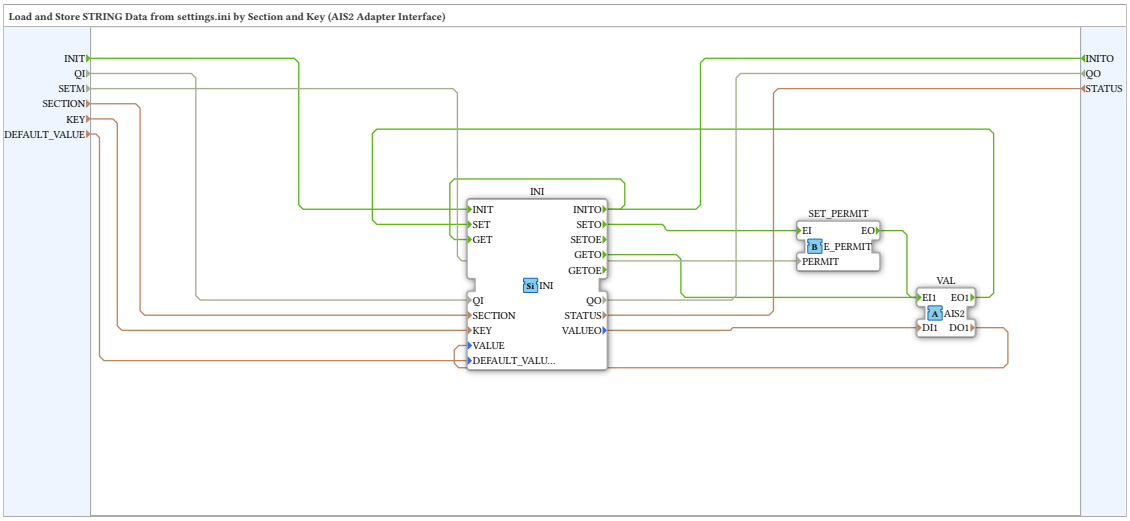
und klarer. Der Baustein eignet sich besonders für Szenarien, in denen mehrere Komponenten auf denselben Datenspeicher zugreifen müssen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

INI_AIS2





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock INI_AIS2 dient dem Lesen und Speichern von STRING-Daten aus einer settings.ini-Datei über einen AIS2-Adapter. Er kombiniert den INI-Baustein mit einem bidirektionalen AIS2-Adapter, um Werte aus einer Konfigurationsdatei auszulesen oder zurückzuschreiben. Die Initialisierung erfolgt über das INIT-Ereignis, wobei der Abschnitt (Section) und der Schlüssel (Key) angegeben werden. Der gelesene Wert wird über den Adapter angeboten, und ein Schreibvorgang wird über das Adapter-Ereignis ausgelöst.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Beschreibung
INIT	Initialisierungsereignis zum Starten des Lesevorgangs. Erwartet die Parameter QI, SECTION, KEY und DEFAULT_VALUE.

Ereignis-Ausgänge

Name	Beschreibung

INITO	Bestätigung der Initialisierung. Wird nach Abschluss des Lesevorgangs gesendet. Statusinformationen über QO und STATUS verfügbar.

Daten-Eingänge

Name	Typ	Beschreibung
QI	BOOL	Eingangsqualifikator zur Steuerung der Verarbeitung.
SETM	BOOL	Steuert die Bestätigung/Spiegelung nach erfolgreichem Schreiben. Unabhängig von SETM löst VAL.EO1 stets das Schreiben (INI.SET) aus; die Rückmeldung SETO wird jedoch nur bei SETM=TRUE an den Adapter (VAL.EI1) weitergeleitet (bei SETM=FALSE unterdrückt).
SECTION	STRING	Name des Abschnitts in der settings.ini-Datei.
KEY	STRING	Name des Schlüssels innerhalb des Abschnitts.
DEFAULT_VALUE	STRING	Standardwert, der gelesen wird, falls der Schlüssel in der Datei nicht vorhanden ist. Voreinstellung: ''.

Daten-Ausgänge

Name	Typ	Beschreibung
QO	BOOL	Ausgangsqualifikator, signalisiert erfolgreiche Verarbeitung.
STATUS	STRING	Statusmeldung des Dienstes (z. B. Fehlertexte).

Adapter

Name	Typ	Beschreibung
VAL	adapter::types::bidirectional::AIS2	Bidirektionaler Adapter für den Austausch von Werten. Über diesen Adapter wird der gelesene Wert als Ausgang (DI1) bereitgestellt und ein Schreibwert als Eingang (DO1) entgegengenommen.

Funktionsweise

Der Baustein INI_AIS2 beinhaltet einen internen INI-Funktionsblock (eclipse4diac::storage::INI), der die eigentliche Dateioperation durchführt. Die Verbindungen im Netzwerk realisieren folgende Abläufe:

- Beim Eintreffen von INIT werden QI, SECTION, KEY und DEFAULT_VALUE an den INI-Baustein weitergeleitet.
- Der INI-Baustein führt einen Lesevorgang aus und gibt das Ergebnis (über VALUEO) an den Adapter-Ausgang (VAL.DI1) weiter.
- Gleichzeitig wird nach erfolgreichem Lesen das Ereignis GET ausgelöst, das den Adapter-Eingang (VAL.EI1) aktiviert, um den Wert zu übermitteln.
- Ein Schreibvorgang wird initiiert, sobald der Adapter ein Ereignis EO1 sendet. Dieses Ereignis triggert das SET-Ereignis am INI-Baustein, wobei der über VAL.DO1 empfangene Wert als neuer VALUE gesetzt wird.
- Der INI-Baustein bestätigt den Schreibvorgang mit SETO. Wenn SETM=TRUE gesetzt ist, wird diese Bestätigung an den Adapter zurückgeleitet (VAL.EI1), während SETM=FALSE diese Spiegelung unterdrückt.
- Die Ausgänge QO und STATUS werden direkt vom INI-Baustein übernommen.

Somit können über den AIS2-Adapter Lese- und Schreibzugriffe auf die settings.ini-Datei gesteuert werden.

Technische Besonderheiten

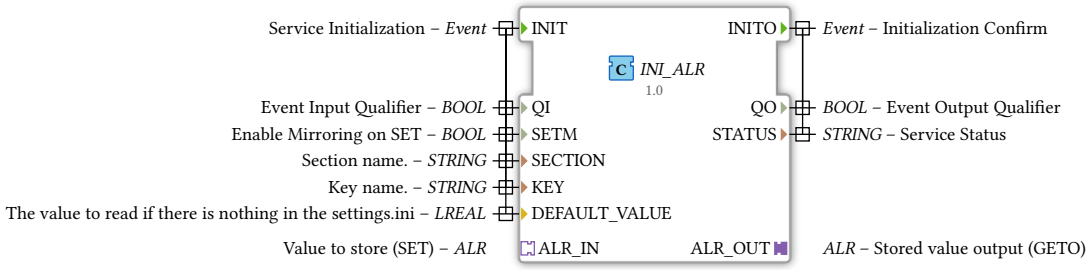
- Der Baustein arbeitet mit einem bidirektionalen AIS2-Adapter, der sowohl Werte senden als auch empfangen kann.

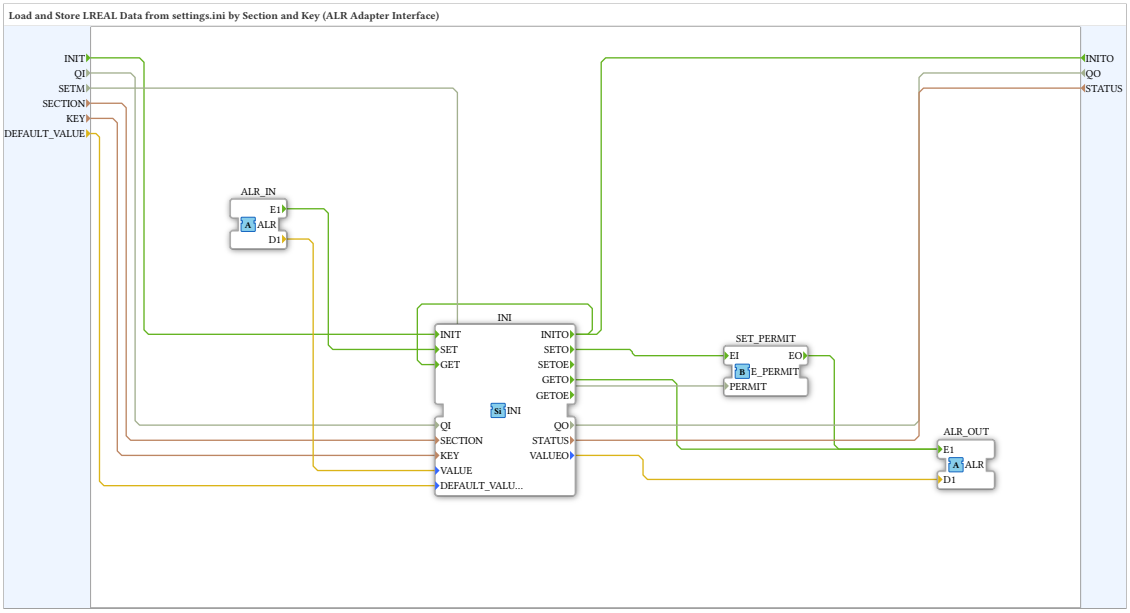
- Als Default-Wert wird ein STRING-Wert verwendet.
- Der interne INI-Baustein ist Teil der eclipse4diac-Standardbibliothek für Dateizugriffe.

Anwendungsszenarien

- Auslesen von Konfigurationsparametern aus einer INI-Datei in einer Automatisierungsanwendung, bei der die Werte über einen AIS2-Adapter an andere Komponenten weitergegeben werden.
- Schreiben von geänderten Parametern zurück in die Datei, z. B. nach einer Benutzereingabe oder einem Algorithmus.

INI_ALR





• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock **INI_ALR** dient dem Lesen und Speichern von Gleitkommawerten des Typs LREAL aus bzw. in eine `settings.ini`-Datei. Er greift dabei auf eine Konfigurationsdatei zu, die über einen Abschnittsnamen (Section) und einen Schlüssel (Key) parametriert wird. Der Baustein kapselt die Nutzung des internen `INI`-Funktionsblocks und stellt zusätzlich eine standardisierte **ALR**-Adapter-Schnittstelle (unidirektional) zur Verfügung, über die Werte sowohl gesetzt als auch gelesen werden können.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar	Mit Variablen
INIT	EInit	Initialisiert den Baustein und löst Lesen aus	QI, SETM, SECTION, KEY, DEFAULT_VALUE

Ereignis-Ausgänge

Ereignis	Typ	Kommentar	Mit Variablen
INITO	EInit	Bestätigt die Initialisierung	QO, STATUS

Daten-Eingänge

Name	Typ	Kommentar
QI	BOOL	Event-Eingangs-Qualifizierer (Freigabe)
SETM	BOOL	Spiegelung (Mirroring) bei SET aktivieren (Enable Mirroring on SET)
SECTION	STRING	Abschnittsname in der INI-Datei
KEY	STRING	Schlüsselname innerhalb des Abschnitts
DEFAULT_VALUE	LREAL	Rückfallwert, falls der Schlüssel in der INI-Datei fehlt

Daten-Ausgänge

Name	Typ	Kommentar
QO	BOOL	Event-Ausgangs-Qualifizierer
STATUS	STRING	Statusmeldung (z. B. Fehlertext)

Adapter

Adapter	Typ	Kommentar
ALR_IN	adapter::types::unidirectional::ALR	Socket – Wert zum Speichern (SET)
ALR_OUT	adapter::types::unidirectional::ALR	Plug – gelesener Wert (GET)/Bestätigung

Funktionsweise

1. **Initialisierung und erstes Lesen** Ein Ereignis am Eingang `INIT` triggert die Initialisierung des internen `INI`-Bausteins. Die Parameter `QI`, `SETM`, `SECTION`, `KEY` und `DEFAULT_VALUE` werden an den `INI`-Baustein weitergeleitet. Nach dessen Quittierung (`INITO`) wird automatisch der GET-Vorgang des `INI`-Bausteins ausgelöst. Der gelesene Wert erscheint am Datenausgang des Adapters `ALR_OUT.D1` und wird gleichzeitig über das Ereignis `ALR_OUT.E1` signalisiert. Die Ausgänge `QO` und `STATUS` des `INI`-Bausteins werden direkt an die gleichnamigen Ausgänge des `INI_ALR` durchgereicht.
2. **Schreiben über ALR-IN** Ein Set-Ereignis am Adapter-Socket `ALR_IN.E1` wird an den SET-Eingang des internen `INI`-Bausteins weitergeleitet. Der zu speichernde Wert liegt an `ALR_IN.D1` und wird über `INI.VALUE` gesetzt. Nach erfolgreichem Schreiben quittiert der `INI`-Baustein mit `SETO`, was wiederum `ALR_OUT.E1` auslöst.
3. **Lesen über ALR-IN** (implizit) Über die initiale GET-Sequenz und die Weiterleitung von `GETO` steht stets der aktuell gelesene Wert am Ausgang `ALR_OUT` zur Verfügung. Ein erneutes Lesen kann durch einen erneuten `INIT`-Impuls angestoßen werden.

Technische Besonderheiten

- **Wiederverwendung** des standardisierten `INI`-Bausteins (`eclipse4diac::storage::INI`) für den Dateizugriff.
- **ALR-Adapter-Schnittstelle** – erlaubt die unidirektionale Kopplung mit anderen Bausteinen, die denselben Adaptertyp verwenden. Dies vereinfacht die Integration in modulare Steuerungsapplikationen.
- **Asynchrone Verarbeitung**: Lese- und Schreibvorgänge werden ereignisgesteuert und asynchron durchgeführt; die Quittierung erfolgt über die entsprechenden Ereignisausgänge.
- **Konform zum Standard IEC 61499-2** – alle Ein- und Ausgänge sind mit Standardtypen definiert.
- **Fehlerbehandlung** erfolgt über den STATUS-Ausgang (z. B. bei fehlender Datei oder fehlerhaften Daten).

Zustandsübersicht

Der `INI_ALR` selbst besitzt keine eigenen Zustandsautomaten. Die Zustandsverwaltung wird vollständig durch den internen `INI`-Baustein realisiert. Typische Phasen sind:

- **Idle** – Warten auf ein INIT-Ereignis.
- **Initialize** – Verarbeitung von INIT, Datenweiterleitung an `INI`.
- **Ready/Lese-Bereit** – nach erfolgreicher INIT-Quittierung ist ein Lese-Ergebnis verfügbar.
- **Schreib-Vorgang** – ausgelöst durch `ALR_IN.E1`.
- **Fehlerzustand** – bei fehlerhaften INI-Zugriffen wird STATUS entsprechend gesetzt.

Anwendungsszenarien

- **Maschinenkonfiguration** – Laden von Parametern (z. B. Maximalgeschwindigkeit, Grenzwerte) aus einer zentralen INI-Konfigurationsdatei.
- **Kalibrierwerte** – Speichern und Auslesen von Kalibrierdaten in der Automatisierungstechnik.
- **Schnittstelle zu ALR-basierten Komponenten** – wenn ein Baustein einen LREAL-Wert über einen ALR-Adapter bereitstellt oder erwartet, kann `INI_ALR` als Bindeglied zur persistenten Speicherung eingesetzt werden.

Vergleich mit ähnlichen Bausteinen

Baustein	Beschreibung	Vorteil von INI_ALR
`INI`	Direkter Zugriff auf INI-Dateien (ohne ALR)	Fehlende Adapter-Schnittstelle, manuelle Kopplung nötig
`INI_ALR`	Kapselt `INI` und stellt ALR-Adapter-Schnittstelle bereit	Einfache Integration in ALR-basierte Architekturen

Der `INI_ALR` kombiniert die bewährte INI-Datei-Funktionalität mit einer standardisierten Adapter-Schnittstelle und reduziert dadurch den Aufwand für die Einbindung in ein diensteorientiertes Steuerungssystem.

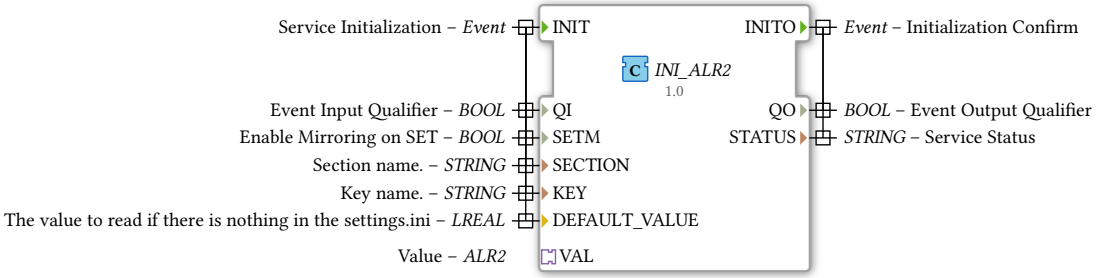
Fazit

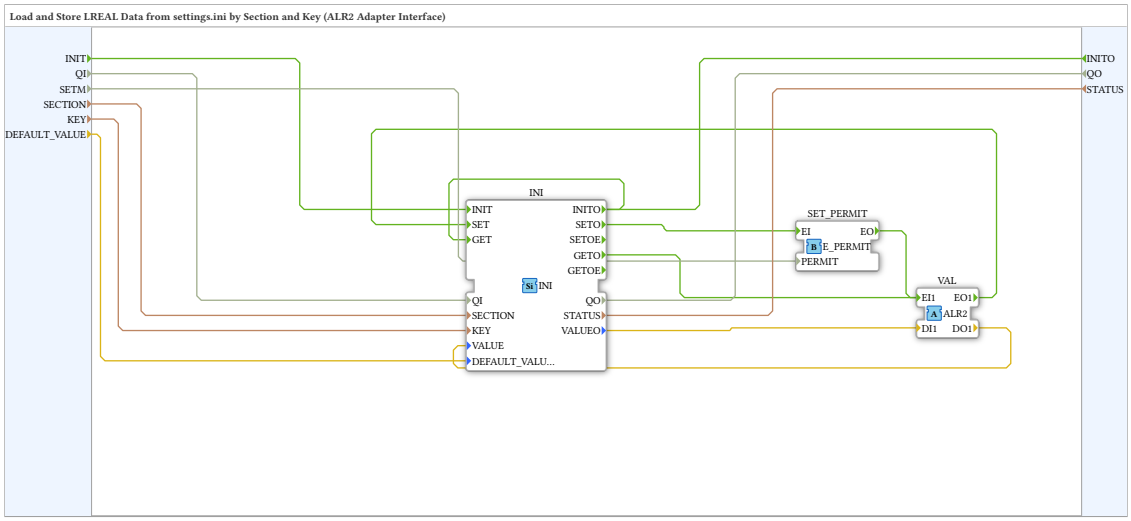
Der `INI_ALR` ist ein praxisorientierter Funktionsblock, der den Zugriff auf `settings.ini` für LREAL-Daten um eine flexible Adapter-Schnittstelle erweitert. Er eignet sich besonders für Anwendungen, die eine einheitliche ALR-basierte Kommunikation erfordern und gleichzeitig eine einfache, dateibasierte Persistenz benötigen. Durch die Kapselung des komplexen `INI`-Bausteins und die automatische Verkettung von Ereignis- und Datenflüssen wird der Entwicklungsaufwand reduziert und die Wiederverwendbarkeit erhöht.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

INI_ALR2





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock INI_ALR2 dient dem Lesen und Speichern von LREAL-Daten aus einer settings.ini-Datei über einen ALR2-Adapter. Er kombiniert den INI-Baustein mit einem bidirektionalen ALR2-Adapter, um Werte aus einer Konfigurationsdatei auszulesen oder zurückzuschreiben. Die Initialisierung erfolgt über das INIT-Ereignis, wobei der Abschnitt (Section) und der Schlüssel (Key) angegeben werden. Der gelesene Wert wird über den Adapter angeboten, und ein Schreibvorgang wird über das Adapter-Ereignis ausgelöst.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Beschreibung
INIT	Initialisierungsereignis zum Starten des Lesevorgangs. Erwartet die Parameter QI, SECTION, KEY und DEFAULT_VALUE.

Ereignis-Ausgänge

Name	Beschreibung
INITO	Bestätigung der Initialisierung. Wird nach Abschluss des Lesevorgangs gesendet. Statusinformationen über QO und STATUS verfügbar.

Daten-Eingänge

Name	Typ	Beschreibung
QI	BOOL	Eingangsqualifikator zur Steuerung der Verarbeitung.
SETM	BOOL	Steuert die Bestätigung/Spiegelung nach erfolgreichem Schreiben. Unabhängig von SETM löst VAL.EO1 stets das Schreiben (INI.SET) aus; die Rückmeldung SETO wird jedoch nur bei SETM=TRUE an den Adapter (VAL.EI1) weitergeleitet (bei SETM=FALSE unterdrückt).
SECTION	STRING	Name des Abschnitts in der settings.ini-Datei.
KEY	STRING	Name des Schlüssels innerhalb des Abschnitts.
DEFAULT_VALUE	LREAL	Standardwert, der gelesen wird, falls der Schlüssel in der Datei nicht vorhanden ist. Voreinstellung: 0.0.

Daten-Ausgänge

Name	Typ	Beschreibung
QO	BOOL	Ausgangsqualifikator, signalisiert erfolgreiche Verarbeitung.
STATUS	STRING	Statusmeldung des Dienstes (z. B. Fehlertexte).

Adapter

Name	Typ	Beschreibung
VAL	adapter::types::bidirectional::ALR2	Bidirektionaler Adapter für den Austausch von Werten. Über diesen Adapter wird der gelesene Wert als Ausgang (DI1) bereitgestellt und ein Schreibwert als Eingang (DO1) entgegengenommen.

Funktionsweise

Der Baustein INI_ALR2 beinhaltet einen internen INI-Funktionsblock (eclipse4diac::storage::INI), der die eigentliche Dateioperation durchführt. Die Verbindungen im Netzwerk realisieren folgende Abläufe:

- Beim Eintreffen von INIT werden QI, SECTION, KEY und DEFAULT_VALUE an den INI-Baustein weitergeleitet.
- Der INI-Baustein führt einen Lesevorgang aus und gibt das Ergebnis (über VALUEO) an den Adapter-Ausgang (VAL.DI1) weiter.
- Gleichzeitig wird nach erfolgreichem Lesen das Ereignis GET ausgelöst, das den Adapter-Eingang (VAL.EI1) aktiviert, um den Wert zu übermitteln.
- Ein Schreibvorgang wird initiiert, sobald der Adapter ein Ereignis EO1 sendet. Dieses Ereignis triggert das SET-Ereignis am INI-Baustein, wobei der über VAL.DO1 empfangene Wert als neuer VALUE gesetzt wird.
- Der INI-Baustein bestätigt den Schreibvorgang mit SETO. Wenn SETM=TRUE gesetzt ist, wird diese Bestätigung an den Adapter zurückgeleitet (VAL.EI1), während SETM=FALSE diese Spiegelung unterdrückt.
- Die Ausgänge QO und STATUS werden direkt vom INI-Baustein übernommen.

Somit können über den ALR2-Adapter Lese- und Schreibzugriffe auf die settings.ini-Datei gesteuert werden.

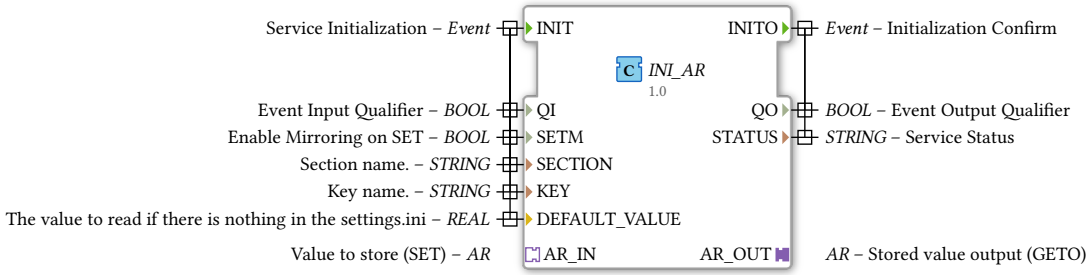
Technische Besonderheiten

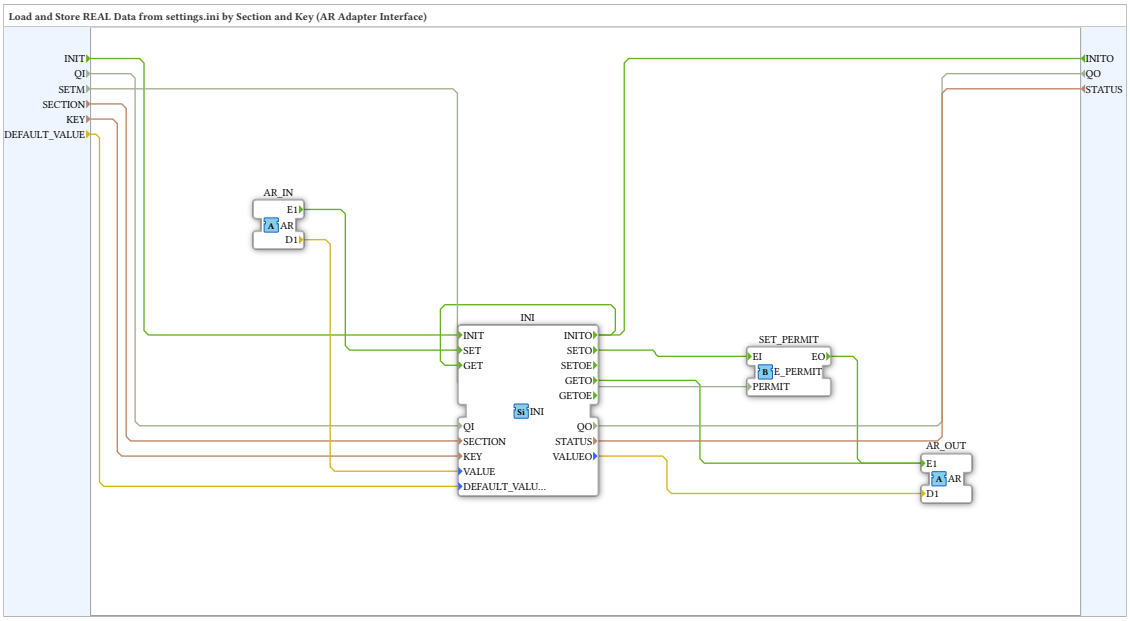
- Der Baustein arbeitet mit einem bidirektionalen ALR2-Adapter, der sowohl Werte senden als auch empfangen kann.
- Als Default-Wert wird ein LREAL-Wert verwendet.
- Der interne INI-Baustein ist Teil der eclipse4diac-Standardbibliothek für Dateizugriffe.

Anwendungsszenarien

- Auslesen von Konfigurationsparametern aus einer INI-Datei in einer Automatisierungsanwendung, bei der die Werte über einen ALR2-Adapter an andere Komponenten weitergegeben werden.
- Schreiben von geänderten Parametern zurück in die Datei, z. B. nach einer Benutzereingabe oder einem Algorithmus.

INI_AR





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock INI_AR dient zum Laden und Speichern von REAL-Daten aus einer settings.ini-Datei. Er greift über einen Sektions- und Schlüsselnamen auf die Konfigurationsdatei zu und stellt die gelesenen bzw. zu speichernden Werte über einen unidirektionalen Adapter zur Verfügung. Dies ermöglicht eine flexible Einbindung in modulare Speicher- und Steuerungsarchitekturen.

Schnittstellenstruktur

Ereignis-Eingänge

- `INIT` (EInit): Service-Initialisierung. Löst das Laden des aktuellen Wertes aus und verarbeitet die zugehörigen Daten-Eingänge.

Ereignis-Ausgänge

- `INITO` (EInit): Bestätigung der Initialisierung. Wird nach erfolgreichem Lese- oder Schreibvorgang gesendet.

Daten-Eingänge

- `QI` (BOOL): Qualifizierer für den Ereignis-Eingang.
- `SETM` (BOOL): Aktiviert die Spiegelung (Mirroring) bei SET.
- `SECTION` (STRING): Name des Abschnitts in der INI-Datei.
- `KEY` (STRING): Name des Schlüssels innerhalb des Abschnitts.
- `DEFAULT_VALUE` (REAL): Wert, der verwendet wird, falls kein Eintrag in der settings.ini gefunden wird.

Daten-Ausgänge

- `QO` (BOOL): Qualifizierer für den Ereignis-Ausgang.
- `STATUS` (STRING): Statusmeldung des Dienstes (z. B. Erfolg, Fehler).

Adapter

- **AR_OUT** (Plug, Typ `adapter::types::unidirectional::AR`): Adapter-Ausgang, über den der gelesene Wert (GETO) an andere Bausteine gesendet wird.
- **AR_IN** (Socket, Typ `adapter::types::unidirectional::AR`): Adapter-Eingang, über den ein zu speichernder Wert (SET) empfangen werden kann.

Funktionsweise

Nach einem INIT-Impuls wird der im Inneren liegende Baustein `INI` (vom Typ `eclipse4diac::storage::INI`) angestoßen. Dieser liest den aktuellen Wert aus der angegebenen Sektion

und dem Schlüssel aus der settings.ini-Datei. Falls kein Eintrag existiert, wird `DEFAULT_VALUE` verwendet. Nach Abschluss des Lesevorgangs wird das Ereignis GET ausgelöst, und der gelesene Wert wird über den Adapter-Ausgang `AR_OUT` (Ereignis GETO, Daten VALUEO) weitergegeben.

Gleichzeitig kann über den Adapter-Eingang `AR_IN` ein neuer Wert (Ereignis E1, Daten D1) empfangen werden. Dieser löst im INI-Baustein das Ereignis SET aus, wodurch der Wert in die INI-Datei geschrieben wird. Nach dem Schreiben erfolgt eine Bestätigung über `SETO`, die ebenfalls an den Adapter-Ausgang weitergeleitet wird. Die Ereignisse GETO und SETO sind auf demselben Ausgangs-Event `AR_OUT.E1` zusammengeführt. Der qualifizierte Ausgang `QO` und `STATUS` werden direkt vom internen INI-Baustein übernommen.

Technische Besonderheiten

- Der Baustein verwendet den internen INI-Funktionsblock aus der Bibliothek `eclipse4diac::storage`.
- Die Adapter `AR_IN` und `AR_OUT` müssen dem Typ `adapter::types::unidirectional::AR` entsprechen.
- Der Baustein ist unter der Eclipse Public License 2.0 (EPL-2.0) lizenziert.
- Autor: Franz Höpfinger, Version 1.0, Stand: 24.04.2026.

Zustandsübersicht

Der Baustein besitzt keinen expliziten Zustandsautomaten. Die Logik wird vollständig über die Ereignisverkettung des internen INI-Bausteins gesteuert:

- Nach `INIT` wird der Wert gelesen (GET).
- Jeder `SET`-Impuls über `AR_IN` schreibt einen neuen Wert in die INI-Datei.

Anwendungsszenarien

- Persistente Speicherung von Konfigurationsparametern (REAL) in einer INI-Datei für Automatisierungsprojekte.
- Austausch von Laufzeitparametern zwischen verschiedenen Funktionsblöcken über die Adapter-Schnittstelle.
- Initialisieren von Anlagenparametern aus einer zentralen Konfigurationsdatei.

Vergleich mit ähnlichen Bausteinen

- **Standard INI-Baustein** (`INI`): Bietet direkten Lese-/Schreibzugriff auf INI-Dateien über Daten-Eingänge und -Ausgänge ohne Adapter.
- **INI_AR** ergänzt diesen um die Adapter-Schnittstelle, sodass Werte modular über Ereignisse und Daten an andere Bausteine weitergegeben werden können, ohne dass die gesamte Logik in einem Block zusammengefasst werden muss.
- Für andere Datenformate (JSON, XML) existieren analoge Bausteine, die jedoch andere Bibliotheken nutzen.

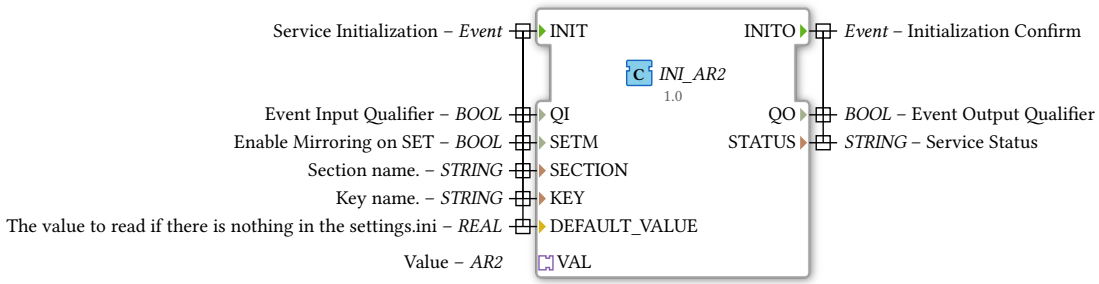
Fazit

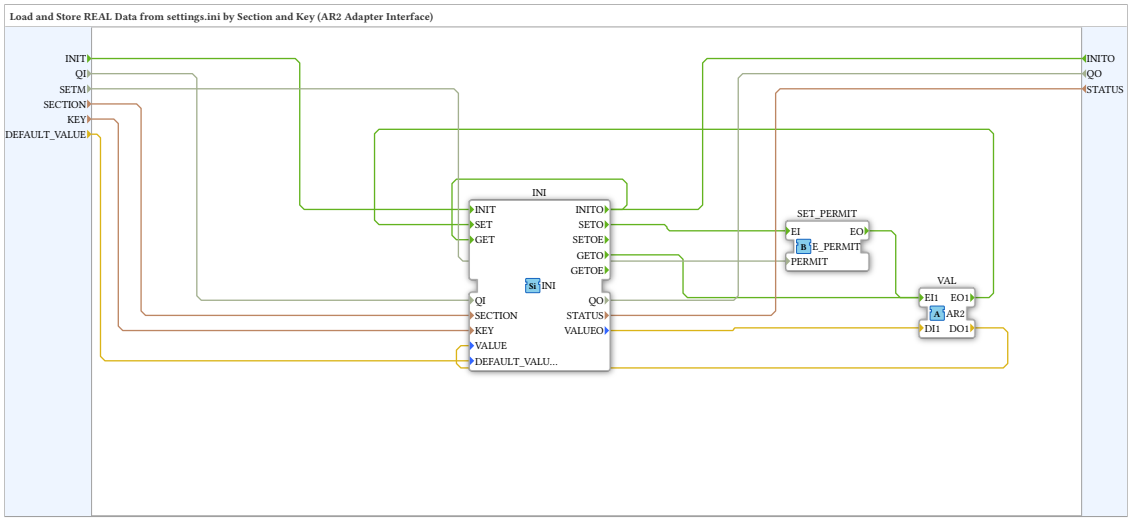
Der Funktionsblock INI_AR kapselt den Zugriff auf eine INI-Datei für REAL-Werte und bietet gleichzeitig eine standardisierte Adapter-Schnittstelle zur Integration in verteilte Steuerungsnetzwerke. Er vereinfacht das Speichern und Laden von Konfigurationsdaten und erlaubt eine klare Trennung zwischen Speicherlogik und Datenflüssen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

INI_AR2





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock INI_AR2 dient zum Auslesen und Schreiben von REAL-Werten aus bzw. in eine settings.ini-Datei über einen Sektions- und Schlüsselnamen. Er kapselt einen INI-Funktionsblock und stellt eine bidirektionale AR2-Adapter-Schnittstelle bereit. Beim Initialisierungsereignis wird der Wert für den angegebenen Sektions- und Schlüsselnamen gelesen und über den Adapter ausgegeben. Gleichzeitig können über den Adapter neue Werte empfangen und in die Datei zurückgeschrieben werden.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (EInit): Service-Initialisierung. Löst das Laden des Wertes aus der settings.ini aus und initialisiert den Baustein.

Ereignis-Ausgänge

- **INITO** (EInit): Bestätigung der Initialisierung. Wird nach erfolgreicher Verarbeitung des INIT-Ereignisses gesendet.

Daten-Eingänge

- **QI** (BOOL): Eingangsqualifizierer. Steuert die Ausführung (z.B. Aktivierung/Deaktivierung).
- **SETM** (BOOL): Enable Mirroring on SET: Bei TRUE wird die Bestätigung eines SET (SETO) zusätzlich am Ausgangsadapter ausgegeben (gespiegelt).
- **SECTION** (STRING): Sektionsname in der settings.ini (z.B. „[MySection]“).
- **KEY** (STRING): Schlüsselname in der Sektion (z.B. „MyKey“).
- **DEFAULT_VALUE** (REAL): Standardwert, der zurückgegeben wird, falls der Schlüssel in der Datei nicht vorhanden ist.

Daten-Ausgänge

- **QO** (BOOL): Ausgangsqualifizierer. Gibt den Status der Verarbeitung an.
- **STATUS** (STRING): Dienststatusmeldung (z.B. Fehlertext oder „OK“).

Adapter

- **VAL** (AR2): Bidirektionaler Adapter vom Typ `adapter::types::bidirectional::AR2`. Über diesen Adapter werden die gelesenen Werte bereitgestellt (Ausgang) und neue Werte empfangen (Eingang) zum Zurückschreiben in die INI-Datei. Der Adapter hat die Ereignisse EO1 (Ausgang) und EI1 (Eingang) sowie Daten DO1 (Ausgang) und DI1 (Eingang).

Funktionsweise

Der FB INI_AR2 ist ein zusammengesetzter (Composite) Baustein, der intern einen INI-Funktionsblock (`eclipse4diac::storage::INI`) verwendet. Der Ablauf ist wie folgt:

1. **Initialisierung**: Beim Eintreffen des INIT-Ereignisses werden die Parameter QI, SETM, SECTION, KEY, DEFAULT_VALUE an den internen INI-Block übergeben.
2. **Lesen**: Der INI-Block führt nach INIT automatisch einen GET-Vorgang aus (gesteuert durch die Eventverbindung von INI.INITO zu INI.GET). Er liest den Wert aus der settings.ini für die angegebene Sektion und den Schlüssel. Falls kein Eintrag vorhanden ist, wird der DEFAULT_VALUE verwendet.
3. **Ausgabe über Adapter**: Der gelesene Wert wird über die Datenverbindung INI.VALUEO -> VAL.DI1 und das Ereignis INI.GETO -> VAL.EI1 an den Adapter-Ausgang weitergegeben.
4. **Schreiben über Adapter**: Wenn über den Adapter ein neuer Wert an VAL.DO1 ankommt (ausgelöst durch VAL.EO1), wird das Ereignis VAL.EO1 -> INI.SET geleitet. Der INI-Block schreibt dann den neuen Wert zurück in die Datei. Nach dem Schreiben wird INI.SETO -> VAL.EI1 gesendet, was die Bestätigung an den Adapter zurückgibt.
5. **Abschluss**: Nach dem initialen Lesevorgang wird das INITO-Ereignis ausgegeben, zusammen mit den Statusinformationen QO und STATUS. Nach Schreibvorgängen über den Adapter erfolgt kein INITO, sondern nur eine Bestätigung über VAL.EI1.

Technische Besonderheiten

- Der Baustein verwendet die Eclipse 4diac-Infrastruktur und importiert Typen aus `eclipse4diac::storage` und `adapter::types::bidirectional::AR2`.
- Der AR2-Adapter ermöglicht eine klare Trennung zwischen Lese- und Schreibzugriffen über einen einzigen Adapter-Socket.
- Standardmäßig wird die Datei `settings.ini` im Arbeitsverzeichnis der Applikation erwartet. Der genaue Pfad kann über die INI-Block-Konfiguration angepasst werden (nicht direkt im INI_AR2 sichtbar).
- Der Baustein ist für REAL-Daten ausgelegt; für andere Datentypen müssten entsprechende Varianten erstellt werden.
- Die Fehlerbehandlung erfolgt über den STATUS-Ausgang und QO.
- Das INITO-Ereignis wird nur nach dem initialen Lesevorgang ausgelöst, nicht nach Schreibvorgängen über den Adapter. Schreibvorgänge werden nur über den Adapter quittiert.

Zustandsübersicht

Da es sich um einen Composite-FB handelt, gibt es keinen expliziten internen Zustandsautomaten. Der Ablauf ergibt sich aus dem Event-Netzwerk:

- INIT empfangen → INI.INIT ausführen.
- Nach INI.INITO: automatisch INI.GET ausführen.
- Nach INI.GETO: VAL.EI1 auslösen (Wertausgabe) und INITO senden.
- Bei Empfang von VAL.EO1 (neuer Wert von außen): INI.SET ausführen, danach INI.SETO → VAL.EI1 (Bestätigung an Adapter, kein INITO).

Anwendungsszenarien

- **Parameterverwaltung**: Speichern und Laden von Einstellungen (z.B. PID-Reglerparameter, Sollwerte) aus einer Konfigurationsdatei.
- **Initiale Konfiguration**: Beim Start einer Anwendung werden vorgegebene Werte aus der settings.ini geladen.
- **Online-Parametrierung**: Über eine HMI oder ein Leitsystem können zur Laufzeit neue Werte über den AR2-Adapter geschrieben werden, die der Baustein sofort in die Datei speichert und bestätigt.
- **Redundanz**: Falls keine Datei vorhanden ist, wird der DEFAULT_VALUE verwendet, sodass die Anwendung auch ohne Konfiguration starten kann.

Vergleich mit ähnlichen Bausteinen

- **INI (eclipse4diac::storage::INI)**: Der INI_AR2 erweitert den Basis-INI-Baustein um eine Adapter-Schnittstelle, sodass er leichter in eine übergeordnete Struktur eingebunden werden kann. Der INI-Baustein benötigt separate Event-Anbindung für Lesen und Schreiben; INI_AR2 vereinfacht dies durch den bidirektionalen Adapter.
- Andere Speicherbausteine (z.B. für Datenbanken oder OPC UA): INI_AR2 ist auf einfache INI-Dateien begrenzt, aber sehr leichtgewichtig und ohne zusätzliche Laufzeitabhängigkeiten.

Fazit

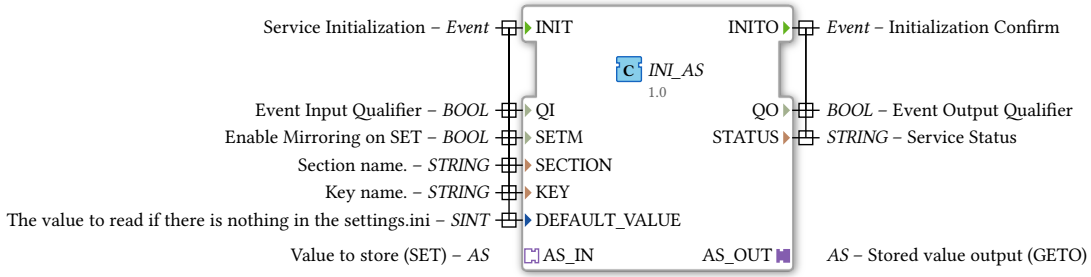
Der INI_AR2-Funktionsblock bietet eine kompakte und einfach zu integrierende Lösung zum persistenten Speichern und Laden von REAL-Konfigurationswerten. Durch die Verwendung eines AR2-Adapters wird

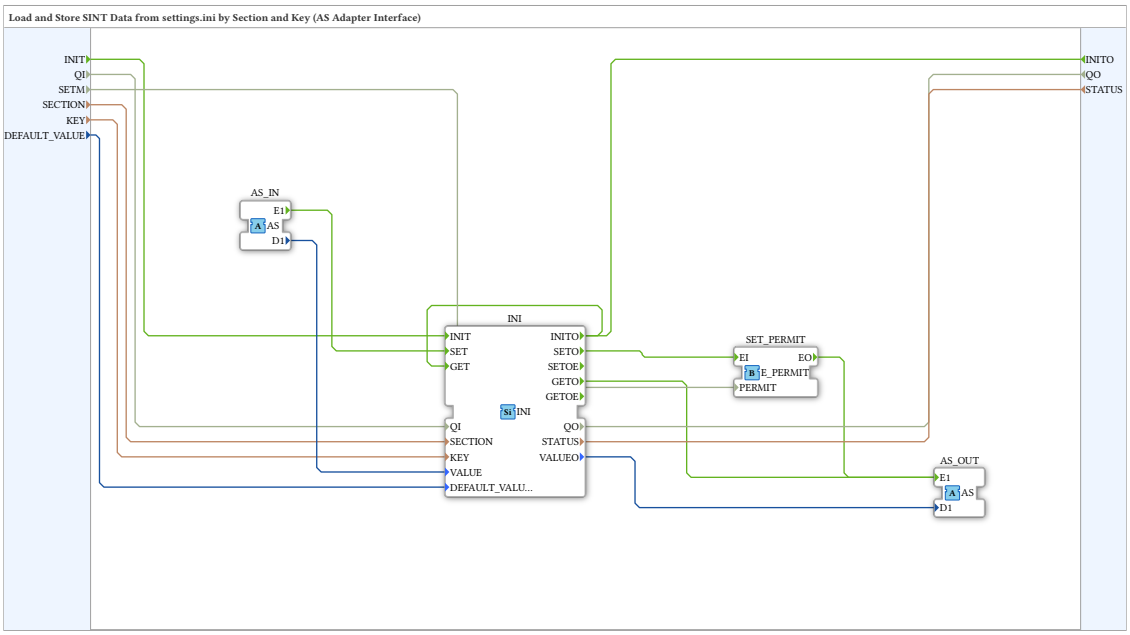
die Anbindung an andere Komponenten standardisiert. Der Baustein eignet sich besonders für Embedded-Steuerungen mit geringen Anforderungen an die Datenhaltung, bei denen eine INI-Datei ausreichend ist.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

INI_AS





• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock **INI_AS** dient dem Laden und Speichern von SINT-Daten aus einer `settings.ini`-Datei. Er greift über einen Abschnittsnamen (`SECTION`) und einen Schlüssel (`KEY`) auf einen Konfigurationswert zu. Über die Adapter-Schnittstelle `AS` kann der Wert sowohl gelesen als auch geschrieben werden. Der Baustein kapselt den internen `INI`-Funktionsblock und erweitert ihn um eine einheitliche Adapter-Schnittstelle.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Mit-Vars	Beschreibung
`INIT`	EInit	QI, SETM, SECTION, KEY, DEFAULT_VALUE	Service-Initialisierung: Liest den Wert aus der INI-Datei und gibt ihn über den Adapter aus.

Ereignis-Ausgänge

Ereignis	Typ	Mit-Vars	Beschreibung
INITO	EInit	QO, STATUS	Bestätigung der Initialisierung.

Daten-Eingänge

Name	Datentyp	Beschreibung
QI	BOOL	Eingangsqualifikator (z. B. Freigabe).
SETM	BOOL	Spiegelung (Mirroring) bei SET aktivieren (Enable Mirroring on SET)
SECTION	STRING	Name des Abschnitts in der INI-Datei.
KEY	STRING	Name des Schlüssels innerhalb des Abschnitts.
DEFAULT_VALUE	SINT	Wert, der zurückgegeben wird, wenn der Schlüssel in der INI-Datei nicht vorhanden ist.

Daten-Ausgänge

Name	Datentyp	Beschreibung
QO	BOOL	Ausgangsqualifikator (zeigt erfolgreiche Operation an).
STATUS	STRING	Statusmeldung (z. B. Fehlertexte).

Adapter

Bezeichnung	Typ	Richtung	Beschreibung
AS_IN	adapter::types::unidirectional::AS	Eingang (Socket)	Empfängt einen neuen zu speichernden Wert (SET-Operation).
AS_OUT	adapter::types::unidirectional::AS	Ausgang (Plug)	Gibt den aktuell gelesenen oder gespeicherten Wert aus (GET-Operation).

Funktionsweise

1. **Initialisierung und erstes Lesen (INIT-Ereignis)** Mit dem INIT-Ereignis werden die Parameter SECTION, KEY und DEFAULT_VALUE übergeben. Der interne INI-Baustein wird gestartet, liest den Wert aus der settings.ini und gibt ihn über den Adapter-Ausgang AS_OUT aus. Gleichzeitig wird das INITO-Ereignis ausgelöst, das den Abschluss der Initialisierung meldet. Anschließend (eventuell noch im gleichen Zyklus) wird automatisch ein GET auf dem INI-Baustein ausgeführt, sodass der gelesene Wert sofort am Adapter anliegt.
2. **Schreiben eines Wertes (über den Adapter-Eingang)** Über den Socket AS_IN kann ein neuer Wert (Ereignis E1 und Daten D1) an den Baustein gesendet werden. Dieses Ereignis wird auf den SET-Eingang des internen INI-Bausteins weitergeleitet, der den Wert in der INI-Datei speichert. Nach erfolgreichem Speichern wird das Ereignis SETO des INI-Bausteins ausgelöst, das wiederum den Adapter-Ausgang AS_OUT (Ereignis E1) bedient und den gespeicherten Wert über D1 ausgibt.
3. **Rückmeldungen** Der interne INI-Baustein liefert die Ausgangssignale QO und STATUS, die direkt an die gleichnamigen Ausgänge des INI_AS-Bausteins durchgeschliffen werden.

Technische Besonderheiten

- **Adapter-Schnittstelle AS**: Der Baustein verwendet einen unidirektionalen Adapter (adapter::types::unidirectional::AS). Dies ermöglicht eine standardisierte Anbindung an andere Bausteine, die denselben Adaptertyp unterstützen.
- **Datentyp SINT**: Der gelesene und gespeicherte Wert ist vom Typ „Short Integer“ (SINT, 8-Bit). Dadurch eignet sich der Baustein besonders für kleine ganzzahlige Konfigurationswerte.

- **Wiederverwendung des `INI`-Bausteins**: Die gesamte Logik zum Zugriff auf die INI-Datei wird vom bewährten `eclipse4diac::storage::INI`-Baustein übernommen. Der `INI_AS` kapselt diesen und ergänzt die Adapter-Schnittstelle.
- **Automatischer GET nach INIT**: Nach dem Initialisieren wird sofort ein Lesevorgang gestartet, sodass der aktuelle Wert ohne separates Ereignis am Adapterausgang bereitsteht.

Zustandsübersicht

Der Baustein besitzt keine explizite Zustandsmaschine (ECC). Das Verhalten wird rein über die Ereignisverkettung innerhalb des internen FBNetzwerks gesteuert:

- Nach `INIT` läuft die Sequenz: INIT des `INI`-Bausteins → INITO → GET (automatisch) → VALUO am Adapterausgang.
- Nach einem SET-Ereignis vom Adaptereingang: SET des `INI`-Bausteins → SETO → Adapterausgang mit dem neuen Wert.

Anwendungsszenarien

- **Persistente Konfiguration**: Speichern und Lesen von Einstellungen wie Schwellwerten, Modus-Flags oder Geräteadressen in einer `settings.ini`.
- **Parametrisierung von Steuerungsanwendungen**: Wenn eine SPS oder ein anderes Automatisierungssystem zur Laufzeit Werte aus einer Konfigurationsdatei laden oder ändern muss.
- **Adapter-basierte Kommunikation**: Einbindung in eine übergeordnete Adapter-Struktur, die mehrere solche Lese-/Schreib-Bausteine zusammenfasst.

Vergleich mit ähnlichen Bausteinen

- **INI_STRING, INI_INT, INI_BOOL**: Diese Bausteine verwenden ebenfalls den internen `INI`-Baustein, unterstützen aber andere Datentypen (STRING, INT, BOOL) und haben oft eine andere Schnittstellenstruktur (keine Adapter). Der `INI_AS` bietet durch den Adapter eine einheitliche und wiederverwendbare Schnittstelle.
- **Direkter `INI`-Baustein**: Der `INI`-Baustein selbst hat mehrere Ereigniseingänge (INIT, GET, SET) und erfordert eine manuelle Verkabelung von Lese- und Schreibvorgängen. Der `INI_AS` vereinfacht die Handhabung, indem er die typische Nutzung (Lesen beim Start, Schreiben über Adapter) standardisiert.

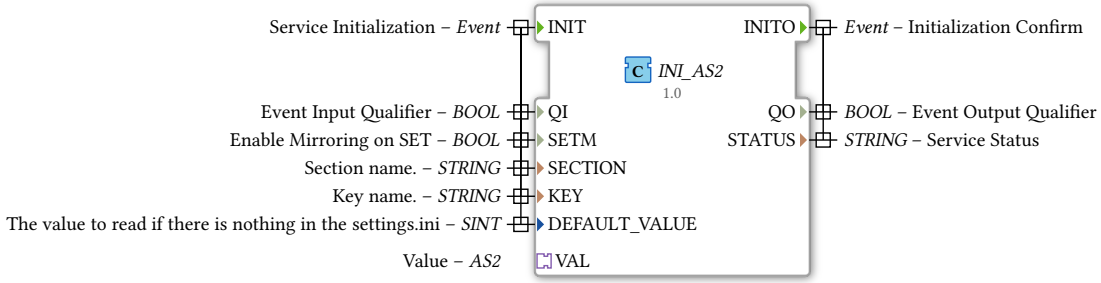
Fazit

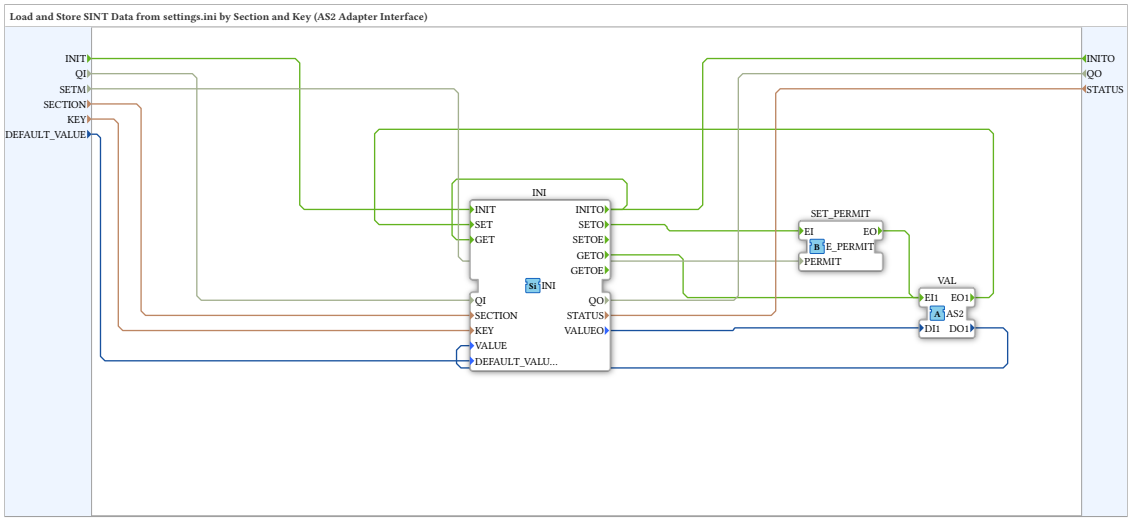
Der Funktionsblock `INI_AS` ist ein komfortabler Baustein zum persistenten Lesen und Schreiben von SINT-Werten aus einer INI-Datei. Durch die Adapter-Schnittstelle lässt er sich leicht in bestehende Architekturen integrieren und reduziert den Verdrahtungsaufwand. Er eignet sich besonders für Konfigurationsaufgaben, bei denen ein einzelner ganzzahliger Wert zwischengespeichert werden muss.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

INI_AS2





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock INI_AS2 dient dem Lesen und Speichern von SINT-Daten aus einer settings.ini-Datei über einen AS2-Adapter. Er kombiniert den INI-Baustein mit einem bidirektionalen AS2-Adapter, um Werte aus einer Konfigurationsdatei auszulesen oder zurückzuschreiben. Die Initialisierung erfolgt über das INIT-Ereignis, wobei der Abschnitt (Section) und der Schlüssel (Key) angegeben werden. Der gelesene Wert wird über den Adapter angeboten, und ein Schreibvorgang wird über das Adapter-Ereignis ausgelöst.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Beschreibung
INIT	Initialisierungsereignis zum Starten des Lesevorgangs. Erwartet die Parameter QI, SECTION, KEY und DEFAULT_VALUE.

Ereignis-Ausgänge

Name	Beschreibung
INITO	Bestätigung der Initialisierung. Wird nach Abschluss des Lesevorgangs gesendet. Statusinformationen über QO und STATUS verfügbar.

Daten-Eingänge

Name	Typ	Beschreibung
QI	BOOL	Eingangsqualifikator zur Steuerung der Verarbeitung.
SETM	BOOL	Steuert die Bestätigung/Spiegelung nach erfolgreichem Schreiben. Unabhängig von SETM löst VAL.EO1 stets das Schreiben (INI.SET) aus; die Rückmeldung SETO wird jedoch nur bei SETM=TRUE an den Adapter (VAL.EI1) weitergeleitet (bei SETM=FALSE unterdrückt).
SECTION	STRING	Name des Abschnitts in der settings.ini-Datei.
KEY	STRING	Name des Schlüssels innerhalb des Abschnitts.
DEFAULT_VALUE	SINT	Standardwert, der gelesen wird, falls der Schlüssel in der Datei nicht vorhanden ist. Voreinstellung: 0.

Daten-Ausgänge

Name	Typ	Beschreibung
QO	BOOL	Ausgangsqualifikator, signalisiert erfolgreiche Verarbeitung.
STATUS	STRING	Statusmeldung des Dienstes (z. B. Fehlertexte).

Adapter

Name	Typ	Beschreibung
VAL	adapter::types::bidirectional::AS2	Bidirektionaler Adapter für den Austausch von Werten. Über diesen Adapter wird der gelesene Wert als Ausgang (DI1) bereitgestellt und ein Schreibwert als Eingang (DO1) entgegengenommen.

Funktionsweise

Der Baustein INI_AS2 beinhaltet einen internen INI-Funktionsblock (eclipse4diac::storage::INI), der die eigentliche Dateioperation durchführt. Die Verbindungen im Netzwerk realisieren folgende Abläufe:

- Beim Eintreffen von INIT werden QI, SECTION, KEY und DEFAULT_VALUE an den INI-Baustein weitergeleitet.
- Der INI-Baustein führt einen Lesevorgang aus und gibt das Ergebnis (über VALUEO) an den Adapter-Ausgang (VAL.DI1) weiter.
- Gleichzeitig wird nach erfolgreichem Lesen das Ereignis GET ausgelöst, das den Adapter-Eingang (VAL.EI1) aktiviert, um den Wert zu übermitteln.
- Ein Schreibvorgang wird initiiert, sobald der Adapter ein Ereignis EO1 sendet. Dieses Ereignis triggert das SET-Ereignis am INI-Baustein, wobei der über VAL.DO1 empfangene Wert als neuer VALUE gesetzt wird.
- Der INI-Baustein bestätigt den Schreibvorgang mit SETO. Wenn SETM=TRUE gesetzt ist, wird diese Bestätigung an den Adapter zurückgeleitet (VAL.EI1), während SETM=FALSE diese Spiegelung unterdrückt.
- Die Ausgänge QO und STATUS werden direkt vom INI-Baustein übernommen.

Somit können über den AS2-Adapter Lese- und Schreibzugriffe auf die settings.ini-Datei gesteuert werden.

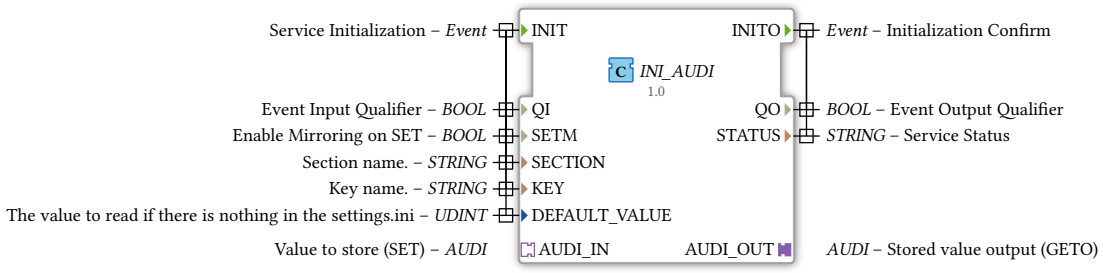
Technische Besonderheiten

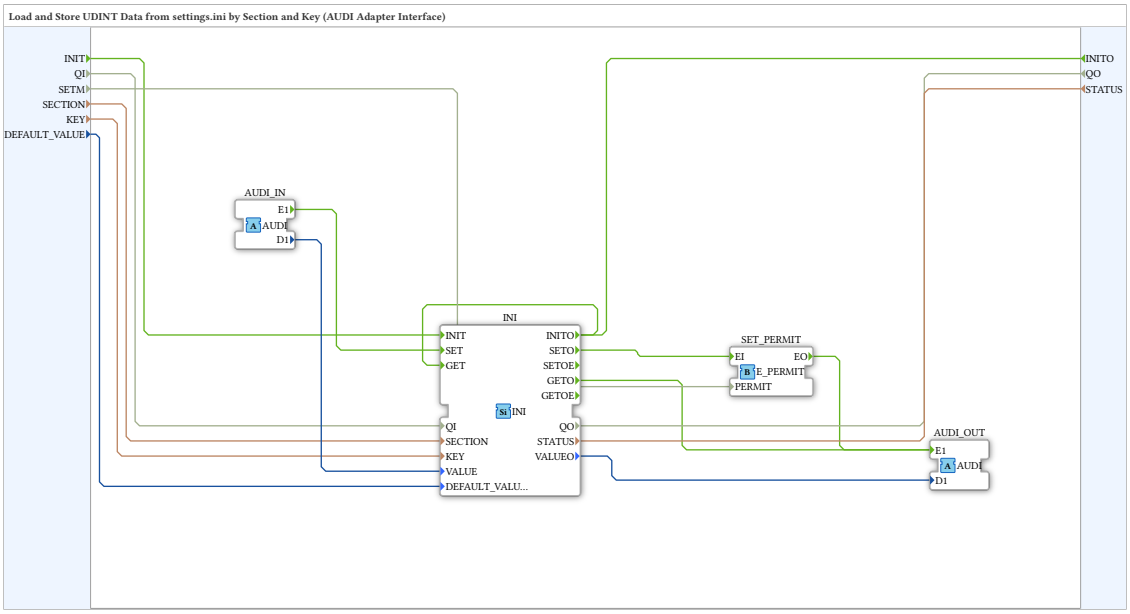
- Der Baustein arbeitet mit einem bidirektionalen AS2-Adapter, der sowohl Werte senden als auch empfangen kann.
- Als Default-Wert wird ein SINT-Wert verwendet.
- Der interne INI-Baustein ist Teil der eclipse4diac-Standardbibliothek für Dateizugriffe.

Anwendungsszenarien

- Auslesen von Konfigurationsparametern aus einer INI-Datei in einer Automatisierungsanwendung, bei der die Werte über einen AS2-Adapter an andere Komponenten weitergegeben werden.
- Schreiben von geänderten Parametern zurück in die Datei, z. B. nach einer Benutzereingabe oder einem Algorithmus.

INI_AUDI





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **INI_AUDI** ermöglicht das Lesen und Speichern von UDINT-Daten aus einer `settings.ini`-Datei. Dabei wird der Zugriff auf die Datei über einen internen `INI`-Baustein realisiert. Die Kommunikation mit der Umgebung erfolgt über zwei **AUDI-Adapter** (unidirektional), die als Plug (`AUDI_OUT`) und Socket (`AUDI_IN`) ausgeführt sind. Dies erlaubt eine modulare Anbindung an andere Bausteine, die über denselben Adaptertyp verfügen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
INIT	EInit	Service Initialization

Ereignis-Ausgänge

Name	Typ	Kommentar
INITO	EInit	Initialization Confirm

Daten-Eingänge

Name	Typ	Kommentar
QI	BOOL	Event Input Qualifier
SETM	BOOL	Spiegelung (Mirroring) bei SET aktivieren
SECTION	STRING	Section name in der settings.ini
KEY	STRING	Key name in der settings.ini
DEFAULT_VALUE	UDINT	Rückfallwert, falls der Schlüssel nicht gefunden wird

Daten-Ausgänge

Name	Typ	Kommentar

QO	BOOL	Output Qualifier
STATUS	STRING	Service Status

Adapter

Name	Typ	Richtung	Kommentar
AUDI_OUT	AUDI	Plug (Ausgang)	Ausgabe des gelesenen Werts
AUDI_IN	AUDI	Socket (Eingang)	Eingang des zu speichernden Werts

Funktionsweise

Der Baustein arbeitet mit dem internen `INI`-Funktionsblock zusammen. Die grundlegende Abfolge ist:

1. **Initialisierung (`INIT`-Ereignis)**
   - Die mit `INIT` verbundenen Eingabedaten (`QI`, `SETM`, `SECTION`, `KEY`, `DEFAULT_VALUE`) werden an den internen `INI`-Baustein weitergeleitet.
   - `INI` wird gestartet und bestätigt mit `INITO`.
2. **Lesen eines Werts**
   - Nach der INIT-Bestätigung wird automatisch der `GET`-Befehl des `INI`-Bausteins ausgelöst.
   - Der gelesene Wert (oder der `DEFAULT_VALUE`) wird über `INI.VALUEO` an den Adapter `AUDI_OUT.D1` ausgegeben.
   - Gleichzeitig wird das Ereignis `AUDI_OUT.E1` gesendet, um den angeschlossenen Baustein zu benachrichtigen.
3. **Schreiben eines Werts**
   - Wird über den Adapter `AUDI_IN` ein Ereignis `E1` mit einem Datenwert `D1` empfangen, so leitet `INI_AUDI` diesen an den `INI`-Baustein als `SET`-Befehl weiter.
   - Der interne `INI`-Baustein speichert den Wert in der INI-Datei unter der angegebenen Section und Key und quittiert mit `SETO`.
   - Das Quittungsereignis wird über `AUDI_OUT.E1` ausgegeben.
4. **Rückmeldung**
   - Die Statusausgänge `QO` und `STATUS` werden vom internen `INI`-Baustein direkt übernommen und stehen am Ausgang zur Verfügung.

Technische Besonderheiten

- Der Baustein verwendet den **Adapter `adapter::types::unidirectional::AUDI`**, der speziell für die gerichtete Kommunikation von Werten und Ereignissen ausgelegt ist.
- Die Daten werden als **UDINT (Unsigned Double Integer)** verarbeitet, was eine effiziente Speicherung und Übertragung ermöglicht.
- Der interne `INI`-Baustein stammt aus der Bibliothek `eclipse4diac::storage` und unterstützt die Standard-INI-Dateiformate.
- Der Fallback-Mechanismus über `DEFAULT_VALUE` stellt sicher, dass auch bei fehlenden Einträgen in der INI-Datei ein definierter Wert zurückgegeben wird.

Zustandsübersicht

Da der Baustein ereignisgesteuert arbeitet, ergeben sich folgende logische Zustände:

- **Idle**: Warten auf ein `INIT`-Ereignis.
- **Initialisieren**: Nach `INIT` – interne Weiterleitung an den INI-Baustein.
- **Lesen** (automatisch nach `INITO`): Der `INI`-Baustein führt einen GET-Befehl aus.
- **Schreiben**: Bei Empfang eines Ereignisses über `AUDI_IN.E1`.
- **Bestätigen**: Ausgabe des entsprechenden OUT-Ereignisses und Aktualisierung der Status-Ausgänge.

Der Baustein ist zustandslos im Sinne eines kontinuierlichen Datenflusses; jede Aktion wird durch ein Ereignis getriggert und synchron beendet.

Anwendungsszenarien

- **Konfigurationsverwaltung** in Automatisierungssystemen: Lesen von Einstellungen (z. B. Grenzwerte, Parameter) aus einer zentralen `settings.ini` und Zurückschreiben geänderter Werte.
- **Adapterbasierte Sensor-/Aktor-Kopplung**: Anbindung eines Bausteins, der über den AUDI-Adapter Werte liest oder setzt, ohne die Dateizugriffslogik selbst implementieren zu müssen.
- **Persistente Datenspeicherung**: Speichern von Zählerständen oder Maschinenzuständen in einer INI-Datei bei gleichzeitiger Kommunikation über standardisierte Adapter.

Vergleich mit ähnlichen Bausteinen

- **Gegenüber reinem `INI`-Baustein**: `INI_AUDI` kapselt die Adapter-Kommunikation und bietet eine höhere Abstraktion. Der Anwender muss nicht direkt mit Ereignissen und Datenverbindungen zwischen mehreren Komponenten hantieren.
- **Gegenüber Bausteinen mit anderen Datentypen**: Der vorliegende Baustein ist auf `UDINT` spezialisiert. Für andere Datentypen (z. B. `STRING`, `REAL`) sind separate Varianten erforderlich. Die grundlegende Struktur bleibt jedoch gleich.
- **Vorteil**: Durch die Verwendung von Adaptern (AUDI) kann der Baustein leicht gegen andere Schnittstellen ausgetauscht werden, sofern diese denselben Adaptertyp implementieren.

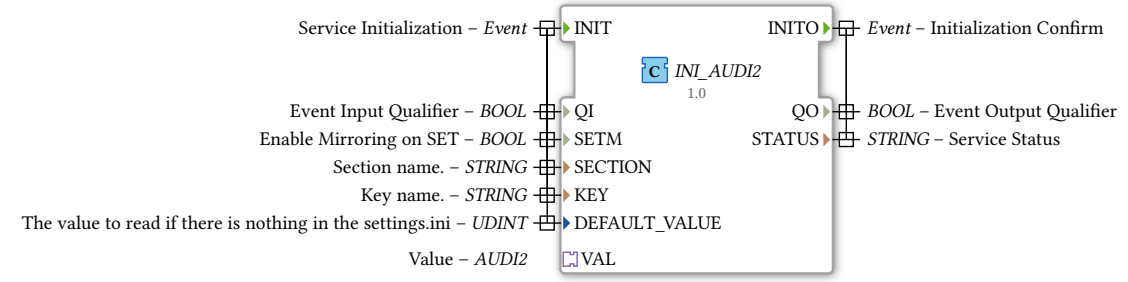
Fazit

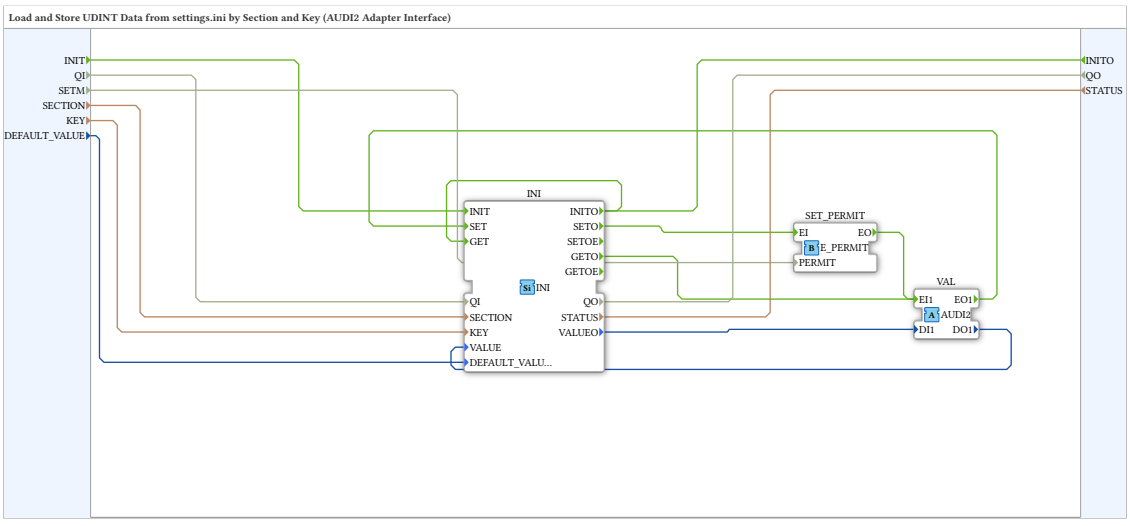
Der `INI_AUDI`-Funktionsblock stellt eine kompakte und wiederverwendbare Lösung dar, um UDINT-Werte aus einer INI-Datei zu lesen und zu schreiben, wobei die Ankopplung an die Umgebung über standardisierte AUDI-Adapter erfolgt. Durch die klare Trennung von Initialisierung, Lesen und Schreiben sowie die Einbindung eines Fallback-Werts eignet er sich besonders für robuste Konfigurations- und Datenspeicherungsaufgaben in Automatisierungsumgebungen. Die einfache Struktur erleichtert die Integration in komplexe 4diac-Netzwerke.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

INI_AUDI2





• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock INI_AUDI2 dient dem Lesen und Speichern von UDINT-Daten aus einer settings.ini-Datei über einen AUDI2-Adapter. Er kombiniert den INI-Baustein mit einem bidirektionalen AUDI2-Adapter, um Werte aus einer Konfigurationsdatei auszulesen oder zurückzuschreiben. Die Initialisierung erfolgt über das INIT-Ereignis, wobei der Abschnitt (Section) und der Schlüssel (Key) angegeben werden. Der gelesene Wert wird über den Adapter angeboten, und ein Schreibvorgang wird über das Adapter-Ereignis ausgelöst.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Beschreibung
INIT	Initialisierungsereignis zum Starten des Lesevorgangs. Erwartet die Parameter QI, SECTION, KEY und DEFAULT_VALUE.

Ereignis-Ausgänge

Name	Beschreibung
INITO	Bestätigung der Initialisierung. Wird nach Abschluss des Lesevorgangs gesendet. Statusinformationen über QO und STATUS verfügbar.

Daten-Eingänge

Name	Typ	Beschreibung
QI	BOOL	Eingangsqualifikator zur Steuerung der Verarbeitung.
SETM	BOOL	Steuert die Bestätigung/Spiegelung nach erfolgreichem Schreiben. Unabhängig von SETM löst VAL.EO1 stets das Schreiben (INI.SET) aus; die Rückmeldung SETO wird jedoch nur bei SETM=TRUE an den Adapter (VAL.EI1) weitergeleitet (bei SETM=FALSE unterdrückt).
SECTION	STRING	Name des Abschnitts in der settings.ini-Datei.
KEY	STRING	Name des Schlüssels innerhalb des Abschnitts.
DEFAULT_VALUE	UDINT	Standardwert, der gelesen wird, falls der Schlüssel in der Datei nicht vorhanden ist. Voreinstellung: 0.

Daten-Ausgänge

Name	Typ	Beschreibung
QO	BOOL	Ausgangsqualifikator, signalisiert erfolgreiche Verarbeitung.
STATUS	STRING	Statusmeldung des Dienstes (z. B. Fehlertexte).

Adapter

Name	Typ	Beschreibung
VAL	adapter::types::bidirectional::AUDI2	Bidirektionaler Adapter für den Austausch von Werten. Über diesen Adapter wird der gelesene Wert als Ausgang (DI1) bereitgestellt und ein Schreibwert als Eingang (DO1) entgegengenommen.

Funktionsweise

Der Baustein INI_AUDI2 beinhaltet einen internen INI-Funktionsblock (eclipse4diac::storage::INI), der die eigentliche Dateioperation durchführt. Die Verbindungen im Netzwerk realisieren folgende Abläufe:

- Beim Eintreffen von INIT werden QI, SECTION, KEY und DEFAULT_VALUE an den INI-Baustein weitergeleitet.
- Der INI-Baustein führt einen Lesevorgang aus und gibt das Ergebnis (über VALUEO) an den Adapter-Ausgang (VAL.DI1) weiter.
- Gleichzeitig wird nach erfolgreichem Lesen das Ereignis GET ausgelöst, das den Adapter-Eingang (VAL.EI1) aktiviert, um den Wert zu übermitteln.
- Ein Schreibvorgang wird initiiert, sobald der Adapter ein Ereignis EO1 sendet. Dieses Ereignis triggert das SET-Ereignis am INI-Baustein, wobei der über VAL.DO1 empfangene Wert als neuer VALUE gesetzt wird.
- Der INI-Baustein bestätigt den Schreibvorgang mit SETO. Wenn SETM=TRUE gesetzt ist, wird diese Bestätigung an den Adapter zurückgeleitet (VAL.EI1), während SETM=FALSE diese Spiegelung unterdrückt.
- Die Ausgänge QO und STATUS werden direkt vom INI-Baustein übernommen.

Somit können über den AUDI2-Adapter Lese- und Schreibzugriffe auf die settings.ini-Datei gesteuert werden.

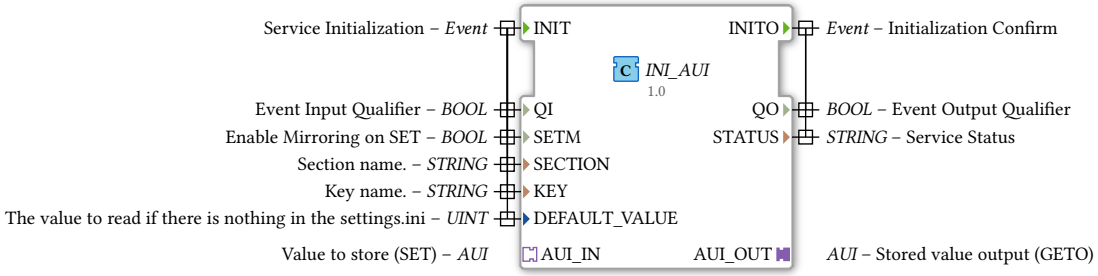
Technische Besonderheiten

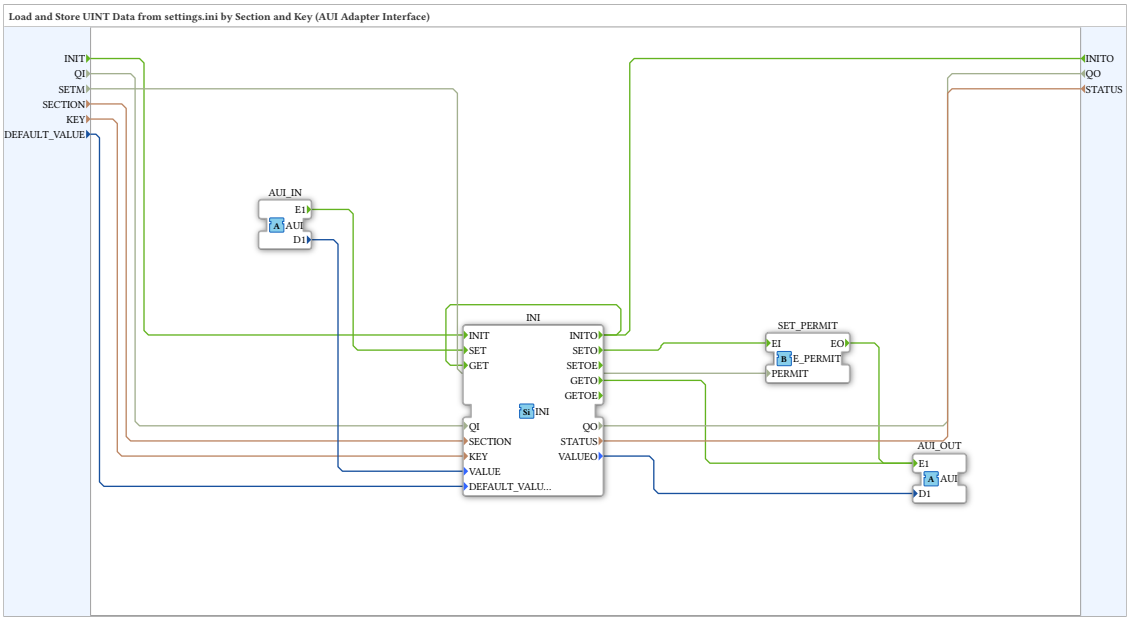
- Der Baustein arbeitet mit einem bidirektionalen AUDI2-Adapter, der sowohl Werte senden als auch empfangen kann.
- Als Default-Wert wird ein UDINT-Wert verwendet.
- Der interne INI-Baustein ist Teil der eclipse4diac-Standardbibliothek für Dateizugriffe.

Anwendungsszenarien

- Auslesen von Konfigurationsparametern aus einer INI-Datei in einer Automatisierungsanwendung, bei der die Werte über einen AUDI2-Adapter an andere Komponenten weitergegeben werden.
- Schreiben von geänderten Parametern zurück in die Datei, z. B. nach einer Benutzereingabe oder einem Algorithmus.

INI_AUI





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `INI_AUI` dient zum Lesen und Speichern von **UINT-Daten** (genauer: `UDINT`-Werte) aus einer `settings.ini`-Datei. Die Parameter **Abschnitt (SECTION)** und **Schlüssel (KEY)** bestimmen, welcher Wert ausgelesen wird. Zusätzlich kann ein **Standardwert (DEFAULT_VALUE)** angegeben werden, falls kein Eintrag in der INI-Datei existiert. Der FB bietet sowohl einen direkten Zugriff über seine Eingänge als auch eine **Adapter-Schnittstelle (AUI)** für die unidirektionale Kommunikation mit anderen Bausteinen.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kurzbeschreibung
INIT	EInit	Service-Initialisierung: Löst das Laden des Wertes aus der INI-Datei aus

Ereignis-Ausgänge

Ereignis	Typ	Kurzbeschreibung
INITO	EInit	Bestätigung der Initialisierung (wird nach Abschluss von Lese-/Schreiboperationen gesendet)

Daten-Eingänge

Name	Typ	Kurzbeschreibung
QI	BOOL	Qualifizierer für den Ereigniseingang (aktiviert die Verarbeitung)
SETM	BOOL	Enable Mirroring on SET: Bei TRUE wird die Bestätigung eines SET (SETO) zusätzlich am Ausgangsadapter ausgegeben (gespiegelt)
SECTION	STRING	Name des Abschnitts in der `settings.ini` (z. B. `[MySection]`)
KEY	STRING	Schlüsselname innerhalb des Abschnitts
DEFAULT_VALUE	UDINT	Wert, der zurückgegeben wird, wenn kein Eintrag existiert

Daten-Ausgänge

Name	Typ	Kurzbeschreibung
QO	BOOL	Qualifizierer für den Ereignisausgang (zeigt erfolgreiche Ausführung an)
STATUS	STRING	Statusmeldung (z. B. Fehler oder Erfolg)

Adapter

Richtung	Adaptertyp	Kurzbeschreibung
Plug	`adapter::types::unidirectional::AUI`	**Ausgang (OUT):** Stellt den gelesenen/geschriebenen Wert über den Adapter bereit
Socket	`adapter::types::unidirectional::AUI`	**Eingang (IN):** Nimmt den zu speichernden Wert über den Adapter entgegen

Funktionsweise

Der `INI_AUI`-FB kapselt einen internen `INI`-Funktionsblock (`eclipse4diac::storage::INI`). Die wesentlichen Abläufe sind:

1. **Lesen eines Wertes**
   - Ein **INIT**-Ereignis am Eingang löst den internen `INI.INIT` aus.
   - Die Daten `QI`, `SETM`, `SECTION`, `KEY` und `DEFAULT_VALUE` werden an den `INI`-FB weitergeleitet.
   - Nach erfolgreicher Verarbeitung sendet `INI.INITO` das Bestätigungsereignis und löst intern `INI.GET` aus, um den Wert aus der Datei zu laden.
   - Der gelesene Wert erscheint an `INI.VALUEO` und wird über den **AUI_OUT**-Adapter (Plug) als `D1` ausgegeben.
   - Gleichzeitig werden `QO` und `STATUS` vom internen FB übernommen.
2. **Schreiben eines Wertes**
   - Ein Ereignis am **AUI_IN.Socket** (über den Eingangsadapter) löst den internen `INI.SET` aus.
   - Der über den Adapter (D1) bereitgestellte Wert wird an `INI.VALUE` übergeben und in die `settings.ini` geschrieben.
   - Nach dem Schreiben sendet `INI.SETO` das Bestätigungsereignis, das über den **AUI_OUT.Plug** als `E1` ausgegeben wird.
   - Auch hier werden die Ausgänge `QO` und `STATUS` aktualisiert.

Hinweis: Der Adapter **AUI_IN** dient als Socket (entgegennehmend), **AUI_OUT** als Plug (bereitstellend). Beide verwenden den gleichen unidirektionalen AUI-Typ.

Technische Besonderheiten

- **Adapterbasierte Kommunikation:** Der FB ermöglicht den Austausch von Konfigurationsdaten über einen unidirektionalen Adapter (AUI), ohne dass direkte Datenverbindungen nötig sind. Dies vereinfacht die Modulkommunikation in verteilten Systemen.
- **Dualer Betrieb:** Der Wert kann sowohl über die klassischen Dateneingänge (`SECTION`, `KEY`, `DEFAULT_VALUE`) als auch über den Adapter (`AUI_IN.D1`) festgelegt werden.
- **Standardwert:** Fehlt ein Eintrag in der INI-Datei, wird `DEFAULT_VALUE` verwendet – dies verhindert undefinierte Zustände.
- **Statusinformation:** Über `STATUS` und `QO` kann der Anwender den Erfolg jeder Operation prüfen.

Zustandsübersicht

Der Funktionsblock `INI_AUI` besitzt **keinen eigenen Zustandsautomaten**. Die gesamte Ablaufsteuerung wird durch den integrierten `INI`-FB und die definierten Ereignisverbindungen (z. B. INIT → INI.INIT, INI.INITO → INITO, INI.INITO → INI.GET) realisiert. Der FB arbeitet daher streng ereignisgesteuert und führt die Lese-/Schreiboperationen sequenziell aus.

Anwendungsszenarien

- **Parameterverwaltung** in modularen Automatisierungssystemen, bei denen mehrere Komponenten über einen Adapter auf gemeinsame Konfigurationsdaten zugreifen.
- **Initialisierung** von Steuerungsfunktionen mit einem Standardwert, der bei Bedarf aus einer INI-Datei überschrieben wird.
- **Austausch von Einstellungen** zwischen verschiedenen Funktionsblöcken, die über den AUI-Adapter verbunden sind (z. B. ein übergeordneter Manager-Baustein und mehrere Worker-Bausteine).

Vergleich mit ähnlichen Bausteinen

- **`INI` (Basisfunktionsblock):** Der `INI_AUI` erweitert den einfachen `INI`-FB um eine standardisierte Adapterschnittstelle. Während der `INI` nur direkte Ein-/Ausgänge besitzt, erlaubt `INI_AUI` die lose Kopplung über AUI.
- **Andere Speicherbausteine (z. B. `Memory`, `Persist`):** Diese arbeiten meist mit internen Variablen oder Dateien, bieten aber oft keine Adapterschnittstelle und keine spezielle `settings.ini`-Integration.

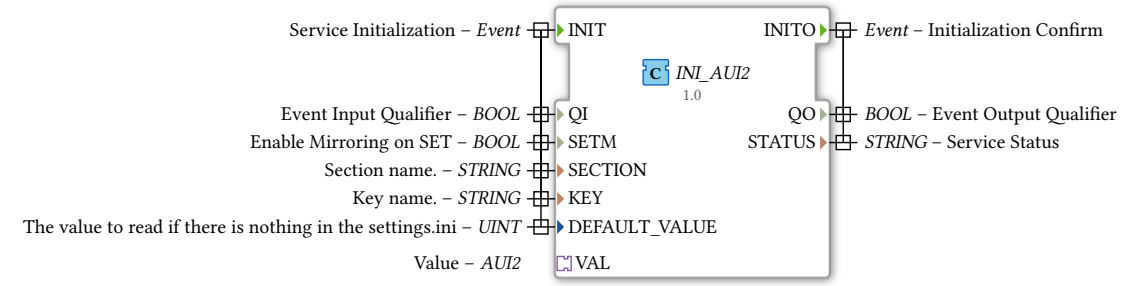
Fazit

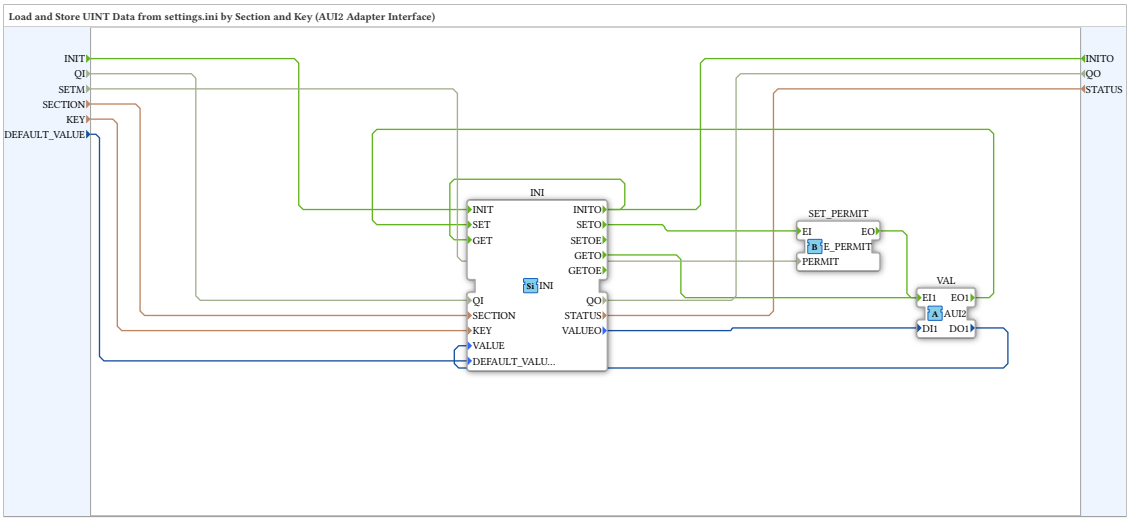
Der `INI_AUI`-Funktionsblock ist eine flexible Lösung zum Lesen und Speichern von Konfigurationswerten in einer `settings.ini`-Datei. Durch die Kombination von direkten Dateneingängen und einer unidirektionalen AUI-Adapter-Schnittstelle eignet er sich hervorragend sowohl für einfache Initialisierungsaufgaben als auch für modulare, adapterbasierte Systemarchitekturen. Die Integration eines Standardwerts und die klare Statusrückmeldung erhöhen die Robustheit und Wartbarkeit der Steuerungssoftware.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

INI_AUI2





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock INI_AUI2 dient dem Lesen und Speichern von UINT-Daten aus einer settings.ini-Datei über einen AUI2-Adapter. Er kombiniert den INI-Baustein mit einem bidirektionalen AUI2-Adapter, um Werte aus einer Konfigurationsdatei auszulesen oder zurückzuschreiben. Die Initialisierung erfolgt über das INIT-Ereignis, wobei der Abschnitt (Section) und der Schlüssel (Key) angegeben werden. Der gelesene Wert wird über den Adapter angeboten, und ein Schreibvorgang wird über das Adapter-Ereignis ausgelöst.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Beschreibung
INIT	Initialisierungsereignis zum Starten des Lesevorgangs. Erwartet die Parameter QI, SECTION, KEY und DEFAULT_VALUE.

Ereignis-Ausgänge

Name	Beschreibung
INITO	Bestätigung der Initialisierung. Wird nach Abschluss des Lesevorgangs gesendet. Statusinformationen über QO und STATUS verfügbar.

Daten-Eingänge

Name	Typ	Beschreibung
QI	BOOL	Eingangsqualifikator zur Steuerung der Verarbeitung.
SETM	BOOL	Steuert die Bestätigung/Spiegelung nach erfolgreichem Schreiben. Unabhängig von SETM löst VAL.EO1 stets das Schreiben (INI.SET) aus; die Rückmeldung SETO wird jedoch nur bei SETM=TRUE an den Adapter (VAL.EI1) weitergeleitet (bei SETM=FALSE unterdrückt).
SECTION	STRING	Name des Abschnitts in der settings.ini-Datei.
KEY	STRING	Name des Schlüssels innerhalb des Abschnitts.
DEFAULT_VALUE	UINT	Standardwert, der gelesen wird, falls der Schlüssel in der Datei nicht vorhanden ist. Voreinstellung: 0.

Daten-Ausgänge

Name	Typ	Beschreibung
QO	BOOL	Ausgangsqualifikator, signalisiert erfolgreiche Verarbeitung.
STATUS	STRING	Statusmeldung des Dienstes (z. B. Fehlertexte).

Adapter

Name	Typ	Beschreibung
VAL	adapter::types::bidirectional::AUI2	Bidirektionaler Adapter für den Austausch von Werten. Über diesen Adapter wird der gelesene Wert als Ausgang (DI1) bereitgestellt und ein Schreibwert als Eingang (DO1) entgegengenommen.

Funktionsweise

Der Baustein INI_AUI2 beinhaltet einen internen INI-Funktionsblock (eclipse4diac::storage::INI), der die eigentliche Dateioperation durchführt. Die Verbindungen im Netzwerk realisieren folgende Abläufe:

- Beim Eintreffen von INIT werden QI, SECTION, KEY und DEFAULT_VALUE an den INI-Baustein weitergeleitet.
- Der INI-Baustein führt einen Lesevorgang aus und gibt das Ergebnis (über VALUEO) an den Adapter-Ausgang (VAL.DI1) weiter.
- Gleichzeitig wird nach erfolgreichem Lesen das Ereignis GET ausgelöst, das den Adapter-Eingang (VAL.EI1) aktiviert, um den Wert zu übermitteln.
- Ein Schreibvorgang wird initiiert, sobald der Adapter ein Ereignis EO1 sendet. Dieses Ereignis triggert das SET-Ereignis am INI-Baustein, wobei der über VAL.DO1 empfangene Wert als neuer VALUE gesetzt wird.
- Der INI-Baustein bestätigt den Schreibvorgang mit SETO. Wenn SETM=TRUE gesetzt ist, wird diese Bestätigung an den Adapter zurückgeleitet (VAL.EI1), während SETM=FALSE diese Spiegelung unterdrückt.
- Die Ausgänge QO und STATUS werden direkt vom INI-Baustein übernommen.

Somit können über den AUI2-Adapter Lese- und Schreibzugriffe auf die settings.ini-Datei gesteuert werden.

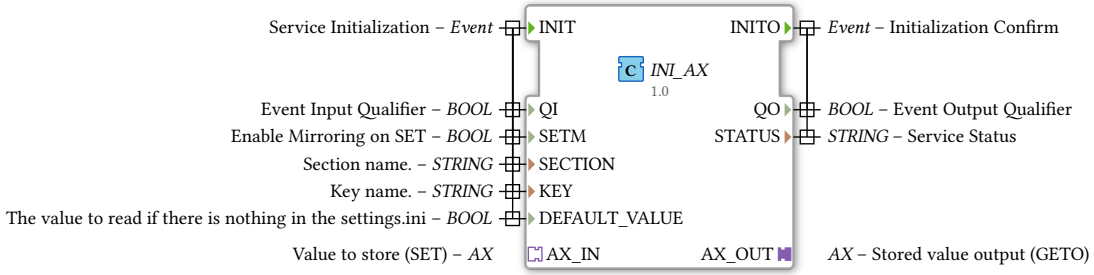
Technische Besonderheiten

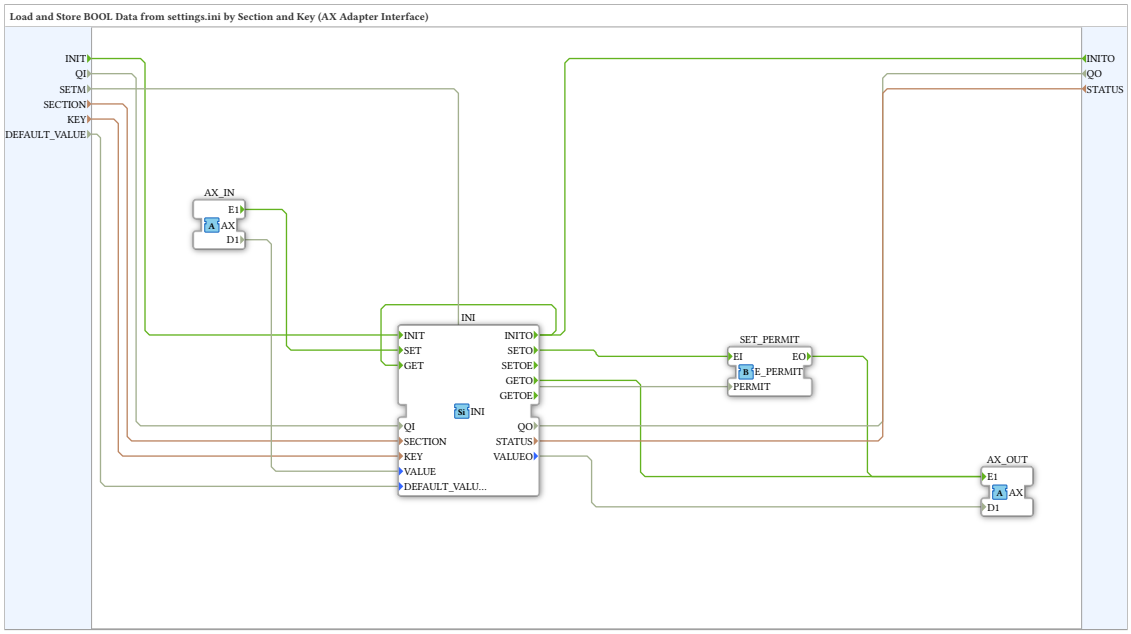
- Der Baustein arbeitet mit einem bidirektionalen AUI2-Adapter, der sowohl Werte senden als auch empfangen kann.
- Als Default-Wert wird ein UINT-Wert verwendet.
- Der interne INI-Baustein ist Teil der eclipse4diac-Standardbibliothek für Dateizugriffe.

Anwendungsszenarien

- Auslesen von Konfigurationsparametern aus einer INI-Datei in einer Automatisierungsanwendung, bei der die Werte über einen AUI2-Adapter an andere Komponenten weitergegeben werden.
- Schreiben von geänderten Parametern zurück in die Datei, z. B. nach einer Benutzereingabe oder einem Algorithmus.

INI_AX





• – ○ • – ○ • – ○ •

Einleitung

`INI_AX` lädt und speichert einen `BOOL`-Wert in der `settings.ini`-Datei, adressiert über Abschnitt (`SECTION`) und Schlüssel (`KEY`). Er ist das boolesche Gegenstück zu INI_AB und das `settings.ini`-Gegenstück zu NVS_AX.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisiert den Baustein und liest den unter `SECTION`/`KEY` hinterlegten Wert (bzw. `DEFAULT_VALUE`, falls noch keiner existiert). Liefert `QI`, `SETM`, `SECTION`, `KEY`, `DEFAULT_VALUE`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Initialisierung und den ersten Lesezugriff, liefert `QO` und `STATUS`.

Daten-Eingänge

- **QI** (BOOL): Qualifizierer des `INIT`-Ereignisses.
- **SETM** (BOOL): Gibt das Spiegeln (Rückmeldung) beim Schreiben (`SET`) frei.
- **SECTION** (STRING): Name des Abschnitts in der `settings.ini`.
- **KEY** (STRING): Name des Schlüssels innerhalb des Abschnitts.
- **DEFAULT_VALUE** (BOOL): Wert, der geliefert wird, wenn unter `SECTION`/`KEY` noch nichts in der `settings.ini` gespeichert ist.

Daten-Ausgänge

- **QO** (BOOL): Erfolgsqualifizierer, mit `INITO` sowie den internen Adapterbestätigungen mitgeliefert.
- **STATUS** (STRING): Statusmeldung des Dateizugriffs.

Adapter

- **AX_IN** (Socket, Typ `AX`): Neuer zu speichernder Wert – ein Adapterereignis auf `AX_IN.E1` löst einen `SET` im internen INI-Baustein aus, `AX_IN.D1` liefert den zu schreibenden Wert.
- **AX_OUT** (Plug, Typ `AX`): Gibt den zuletzt gelesenen Wert aus – sowohl nach dem automatischen Erstlesen bei `INIT` als auch nach jedem bestätigten Schreibvorgang (`AX_OUT.D1`/`AX_OUT.E1`).

Funktionsweise

`INI_AX` kapselt intern eine Instanz von `eclipse4diac::storage::INI` sowie ein `E_PERMIT`-Gatter (`SET_PERMIT`). Bei `INIT` wird die interne INI-Instanz initialisiert (`INI.INIT`) und anschließend automatisch

ein GET ausgelöst, das den unter SECTION/KEY gespeicherten Wert (oder DEFAULT_VALUE) liefert; das Ergebnis wird über AX_OUT ausgegeben, INITO bestätigt den Abschluss.

Zum Schreiben liefert der an AX_IN angeschlossene Baustein ein Adapterereignis (AX_IN.E1) mit dem neuen Wert (AX_IN.D1); dies löst INI.SET aus. Ob die erfolgreiche Speicherung zusätzlich über AX_OUT zurückgemeldet wird, steuert SETM: Ist SETM gesetzt, gibt SET_PERMIT das AX_OUT.E1-Ereignis nach erfolgreichem SET frei („Mirroring“).

Technische Besonderheiten

- Reines Composite-FB ohne eigene ECC – die eigentliche Dateizugriffslogik steckt in der internen INI-Instanz, INI_AX verdrahtet nur Ereignisse und Daten zwischen ihr und den beiden AX-Adaptern.
- Zwei getrennte, unidirektionale Adapter (AX_IN/AX_OUT) – dadurch lassen sich Lese- und Schreibpfad an unterschiedliche Gegenstellen anschließen.
- Adressierung über SECTION **und** KEY (statt nur KEY bei den NVS_*-Bausteinen) entspricht dem klassischen INI-Dateiformat.

Zustandsübersicht

Der Baustein ist zustandslos im Sinne einer eigenen ECC: Er durchläuft bei INIT einmalig Initialisierung + Erstlesen, danach reagiert er auf jedes AX_IN-Adapterereignis mit einem INI-Schreibzugriff und optional einer Bestätigung über AX_OUT.

Anwendungsszenarien

- Persistieren einzelner BOOL-Konfigurationswerte (z. B. Freigabe-Flags, Betriebsartenschalter) in einer menschenlesbaren settings.ini
- Aufbau eines Lese-/Schreibpaars, bei dem Lese- und Schreibseite an unterschiedliche Bausteine angeschlossen werden sollen
- Konfigurationsdateien, die auch außerhalb der Steuerung (z. B. per Texteditor) einsehbar/änderbar sein sollen

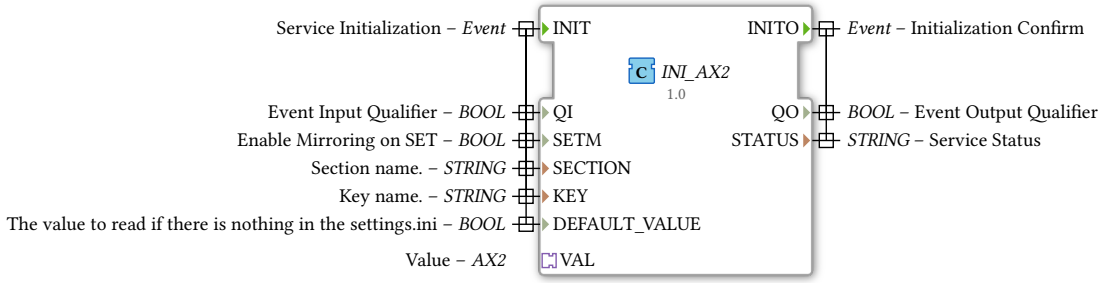
⚖ Vergleich mit ähnlichen Bausteinen

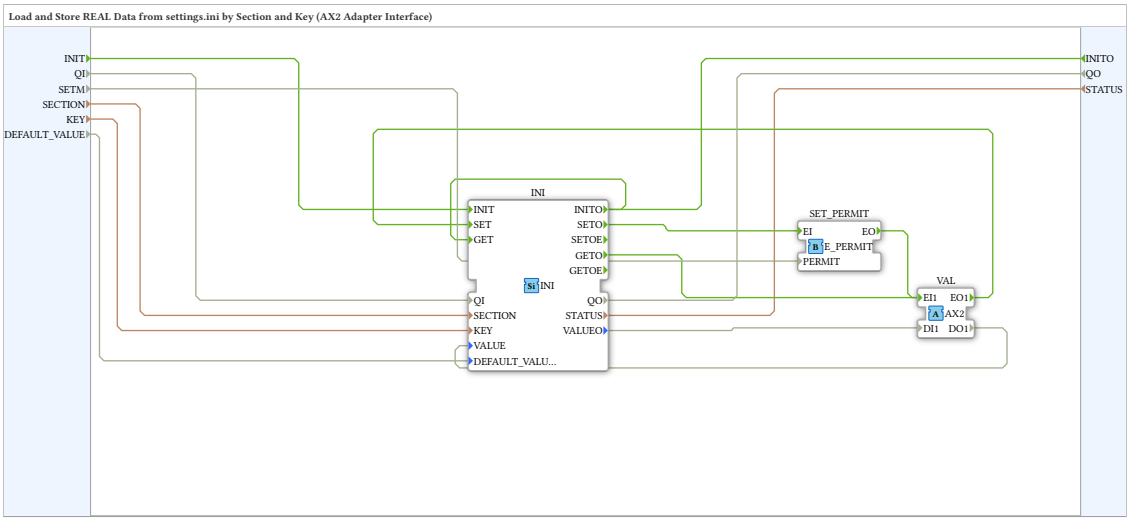
Vergleich mit INI_AB, das dieselbe Architektur für BYTE- statt BOOL-Werte nutzt, sowie mit NVS_AX, das dieselbe Schnittstelle für den ESP32-NVS-Flash-Speicher statt für die settings.ini bereitstellt.

Fazit

INI_AX bietet einen einfachen, unidirektional verdrahteten Zugriff auf einen persistenten BOOL-Wert in der settings.ini und eignet sich überall dort, wo Lese- und Schreibpfad getrennt geführt werden sollen.

INI_AX2





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock INI_AX2 dient dem Lesen und Speichern von REAL-Daten aus einer settings.ini-Datei über einen AX2-Adapter. Er kombiniert den INI-Baustein mit einem bidirektionalen AX2-Adapter, um Werte aus einer Konfigurationsdatei auszulesen oder zurückzuschreiben. Die Initialisierung erfolgt über das INIT-Ereignis, wobei der Abschnitt (Section) und der Schlüssel (Key) angegeben werden. Der gelesene Wert wird über den Adapter angeboten, und ein Schreibvorgang wird über das Adapter-Ereignis ausgelöst.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Beschreibung
INIT	Initialisierungsereignis zum Starten des Lesevorgangs. Erwartet die Parameter QI, SETM, SECTION, KEY und DEFAULT_VALUE.

Ereignis-Ausgänge

Name	Beschreibung
INITO	Bestätigung der Initialisierung. Wird nach Abschluss des Lesevorgangs gesendet. Statusinformationen über QO und STATUS verfügbar.

Daten-Eingänge

Name	Typ	Beschreibung
QI	BOOL	Eingangsqualifikator zur Steuerung der Verarbeitung.
SETM	BOOL	Spiegelung (Mirroring) bei SET aktivieren (Enable Mirroring on SET)
SECTION	STRING	Name des Abschnitts in der settings.ini-Datei.
KEY	STRING	Name des Schlüssels innerhalb des Abschnitts.
DEFAULT_VALUE	BOOL	Standardwert, der gelesen wird, falls der Schlüssel in der Datei nicht vorhanden ist. Voreinstellung: FALSE.

Daten-Ausgänge

Name	Typ	Beschreibung

QO	BOOL	Ausgangsqualifikator, signalisiert erfolgreiche Verarbeitung.
STATUS	STRING	Statusmeldung des Dienstes (z. B. Fehlertexte).

Adapter

Name	Typ	Beschreibung
VAL	adapter::types::bidirectional::AX2	Bidirektionaler Adapter für den Austausch von Werten. Über diesen Adapter wird der gelesene Wert als Ausgang (DI1) bereitgestellt und ein Schreibwert als Eingang (DO1) entgegengenommen.

Funktionsweise

Der Baustein INI_AX2 beinhaltet einen internen INI-Funktionsblock (eclipse4diac::storage::INI), der die eigentliche Dateioperation durchführt. Die Verbindungen im Netzwerk realisieren folgende Abläufe:

- Beim Eintreffen von INIT werden QI, SETM, SECTION, KEY und DEFAULT_VALUE an den INI-Baustein weitergeleitet.
- Der INI-Baustein führt einen Lesevorgang aus und gibt das Ergebnis (über VALUEO) an den Adapter-Ausgang (VAL.DI1) weiter.
- Gleichzeitig wird nach erfolgreichem Lesen das Ereignis GET ausgelöst, das den Adapter-Eingang (VAL.EI1) aktiviert, um den Wert zu übermitteln.
- Ein Schreibvorgang wird initiiert, sobald der Adapter ein Ereignis EO1 sendet. Dieses Ereignis triggert das SET-Ereignis am INI-Baustein, wobei der über VAL.DO1 empfangene Wert als neuer VALUE gesetzt wird.
- Der INI-Baustein bestätigt den Schreibvorgang mit SETO und leitet dies an den Adapter zurück (VAL.EI1).
- Die Ausgänge QO und STATUS werden direkt vom INI-Baustein übernommen.

Somit können über den AX2-Adapter Lese- und Schreibzugriffe auf die settings.ini-Datei gesteuert werden.

Technische Besonderheiten

- Der Baustein arbeitet mit einem bidirektionalen AX2-Adapter, der sowohl Werte senden als auch empfangen kann.
- Als Default-Wert wird ein BOOL-Wert verwendet, obwohl der Kommentar auf REAL-Daten hinweist. Der tatsächliche Wertetyp wird durch den Adapter bestimmt.
- Der interne INI-Baustein ist Teil der eclipse4diac-Standardbibliothek für Dateizugriffe.
- Die Verbindungen im FBNetzwerk sind so konfiguriert, dass die Ereignissteuerung zwischen Lese- und Schreibvorgängen synchronisiert wird.

Zustandsübersicht

Der Baustein selbst besitzt keine expliziten Zustände, da er als reines Netzwerk aus unterlagerten Bausteinen realisiert ist. Die Zustandslogik wird durch den internen INI-Baustein und den Adapter bestimmt. Typische Zustände sind:

- **Initialisierung**: Warten auf INIT-Ereignis.
- **Lesen**: Verarbeitung des Lesevorgangs durch INI-Baustein.
- **Schreiben**: Verarbeitung eines Schreibbefehls über Adapter.
- **Warten**: Nach Abschluss eines Vorgangs wird auf das nächste Ereignis gewartet.

Anwendungsszenarien

- Auslesen von Konfigurationsparametern aus einer INI-Datei in einer Automatisierungsanwendung, bei der die Werte über einen AX2-Adapter an andere Komponenten weitergegeben werden.
- Schreiben von geänderten Parametern zurück in die Datei, z. B. nach einer Benutzereingabe oder einem Algorithmus.
- Kombination mit anderen Bausteinen zur persistenten Speicherung von Einstellungen in einem IEC 61499-System.

Vergleich mit ähnlichen Bausteinen

- Der einfache **INI**-Baustein bietet direkten Lese-/Schreibzugriff auf INI-Dateien, jedoch ohne Adapteranbindung. INI_AX2 erweitert dies um eine standardisierte AX2-Schnittstelle für den Werteaustausch.
- Andere Speicherbausteine wie **CSV** oder **JSON**-Bausteine dienen anderen Dateiformaten; INI_AX2 ist spezifisch für das INI-Format mit bidirektionalem Adapter.

Fazit

Der Funktionsblock INI_AX2 ist eine nützliche Komponente für IEC 61499-Anwendungen, die einen einfachen Zugriff auf Konfigurationsdateien über einen AX2-Adapter benötigen. Durch die Kapselung des INI-Bausteins und die Ereignissteuerung ermöglicht er sowohl Lese- als auch Schreibvorgänge in einer settings.ini-Datei. Die Verwendung eines bidirektionalen Adapters erlaubt eine flexible Einbindung in komplexe Netzwerke.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Schreibgeschützte Einstellungen (`settingsReadOnly.ini`)

Einleitung

In logiBUS®-Systemen werden Konfigurationswerte und Parameter in INI-Dateien auf dem Dateisystem des Steuergeräts verwaltet. Neben der frei beschreibbaren Standard-Konfigurationsdatei (`settings.ini`) unterstützt das System eine schreibgeschützte Einstellungsdatei: **`settingsReadOnly.ini`**.

Dieses Feature dient dazu, Werkseinstellungen, herstellerseitige Systemparameter, feste Kommunikationsadressen oder kritische Hardware- und Sicherheitsoptionen dauerhaft einzufrieren. So wird verhindert, dass wichtige Parameter durch Bediener oder durch die Steuerungsausführung versehentlich geändert werden.

Konzept der zwei Konfigurationsdateien

Das Steuergerät unterscheidet zwei Ebenen der Konfigurationsspeicherung:

Datei	Zweck	Schreibbar?	Typische Inhalte
`settingsReadOnly.ini`	Werkseinstellungen & Systemvorgaben	❌ Nein (Schreibschutz)	ISOBUS-Quelladressen (Node-SAs), Hardware-Pinbelegungen, fest vorgegebene Boot-Zeiten
`settings.ini`	Benutzer- & Laufzeiteinstellungen	✅ Ja (Lesen & Schreiben)	Benutzerpräferenzen, dynamische Betriebsparameter, wählbare Grenzwert-Skalierungen

Funktionsweise & Priorität beim Start

1. **Vorrang von Werkseinstellungen**: Beim Booten liest das Steuergerät zuerst die Datei `settingsReadOnly.ini`. Alle darin definierten Abschnitte (`[Section]`) und Schlüssel (`KEY`) werden schreibgeschützt geladen.
2. **Automatische Bereinigung (*Purge*)**: Befindet sich ein Schlüssel sowohl in `settingsReadOnly.ini` als auch in der beschreibbaren `settings.ini`, wird der doppelte Schlüssel beim Systemstart automatisch aus `settings.ini` **entfernt**. Dadurch ist gewährleistet, dass keine widersprüchlichen Werte existieren und die Benutzerdatei übersichtlich bleibt.
3. **Ergänzendes Laden**: Alle Parameter, die *nicht* in `settingsReadOnly.ini` stehen, werden aus `settings.ini` geladen und verhalten sich normal schreibbar.

Verhalten im Betrieb (Lesen & Schreiben)

Lesezugriffe (`GET`)

Das Auslesen von Parametern erfolgt transparent. Wenn ein Schlüssel in `settingsReadOnly.ini` hinterlegt ist, wird stets dessen unveränderlicher Wert zurückgegeben.

Schreibzugriffe (`SET`)

Wird versucht, einen schreibgeschützten Schlüssel zu ändern (egal ob durch Steuerungsfunktionen oder über 4diac IEC 61499 Funktionsblöcke), passiert Folgendes:

- Der **Schreibversuch wird abgelehnt**.
- Der in `settingsReadOnly.ini` festgelegte Wert **bleibt unverändert**.
- Die Ablehnung des Schreibvorgangs wird protokolliert und an die aufrufende Baustein-Logik zurückgemeldet.

Verhalten in 4diac FORTE (IEC 61499 INI-Bausteine)

Für Anwender von 4diac FORTE Steuerungsprogrammen verhalten sich die INI-Funktionsblöcke (`INI`, `INI_AX`, `INI_AUI`, `INI_AR` usw.) bei schreibgeschützten Parametern wie folgt:

- **Lese-Ereignis (`GET`)**: Signalisiert wie gewohnt das Bestätigungsereignis `GETO`. Am Ausgang `VALUEO` liegt der geschützte Wert an, `STATUS` meldet `"OK"`.
- **Schreib-Ereignis (`SET`)**:
  - Es wird **nicht** das normale Erfolgsereignis (`SETO`) ausgelöst, sondern das Fehler-Ereignis **`SETOE`** (*Set Output Error*).
  - Der Datenausgang **`STATUS`** liefert die verständliche Meldung: **`"Key is read-only"`**.
  - Der Ausgang **`QO`** zeigt den Fehlerzustand an.

!!! note „Hinweis für Applikationsentwickler“ Durch Auswerten des Ausgangsereignisses `SETOE` oder des Statusstrings `"Key is read-only"` kann in der 4diac-Applikation gezielt auf schreibgeschützte Parameter reagiert werden (z. B. Anzeige eines Hinweises auf der Visualisierung).

Einspielen und Verwalten der Werkseinstellungen

Die Datei `settingsReadOnly.ini` liegt auf dem Speicher des Steuergeräts unter: `/data/settingsReadOnly.ini`

Übertragung auf das Steuergerät

Das Einspielen oder Aktualisieren der Werkseinstellungen erfolgt einfach über das integrierte **Web-Interface** (File Server) des Steuergeräts:

1. Verbinden Sie sich über einen Webbrowser mit dem Web-Interface des Steuergeräts.
2. Navigieren Sie zur Dateiverwaltung (`/data`).
3. Laden Sie Ihre vorbereitete `settingsReadOnly.ini` hoch.
4. Nach einem Neustart des Steuergeräts werden die schreibgeschützten Parameter wirksam.

Verhalten bei fehlender Datei

Falls keine `settingsReadOnly.ini` auf dem Steuergerät vorhanden ist, startet das System gewohnt ohne Einschränkungen. Alle Schlüssel in `settings.ini` verhalten sich wie bisher voll beschreibbar.

Praktisches Beispiel

Beispieldatei `settingsReadOnly.ini` (Werkseinstellungen)

```
[CF-A]
NODE1_SA = 128
bootTimeVT = 90

[System]
DeviceName = LOGIBUS-NODE-01
```

Beispieldatei `settings.ini` (Benutzereinstellungen)

```
[CF-A]
UserLanguage = DE
```

```
[User]
OperatorID = 42
```

Ergebnis im Betrieb:

- `NODE1_SA` (`128`), `bootTimeVT` (`90`) und `DeviceName` (`"LOGIBUS-NODE-01"`) sind geschützt und können nicht verändert werden.
- `UserLanguage` (`"DE"`) und `OperatorID` (`42`) können durch die Steuerung frei geändert und dauerhaft gespeichert werden.
- Sollte in `settings.ini` versehentlich ein alter Wert `NODE1_SA = 130` vorhanden gewesen sein, löscht das Steuergerät diesen beim Booten automatisch, sodass stets die Vorgabe `128` gilt.

Zusammenfassung & Best Practices

Ziel	Empfohlene Vorgehensweise
Werkseinstellungen sichern	Legen Sie alle unveränderlichen Parameter (z. B. Bus-Adressen, Sicherheitsgrenzen) in `settingsReadOnly.ini` an und spielen Sie diese auf das Steuergerät auf.
Benutzereinstellungen erlauben	Halten Sie Parameter, die vom Bediener angepasst werden dürfen (z. B. Sprache, Betriebsmodi, Zählerstände), ausschließlich in `settings.ini`.
Parameter freigeben	Um einen Parameter wieder beschreibbar zu machen, entfernen Sie den betreffenden Schlüssel aus `settingsReadOnly.ini` und starten Sie das Steuergerät neu.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.1.14.12 utils

conversion

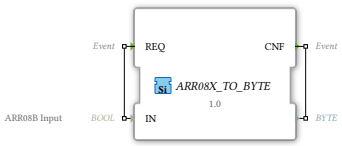
arr

🎧 Podcast

- Schalt- vs. Linearregler: Effizienz, EMV & der Kompromiss der Leistungselektronik

assembling

ARR08X_TO_BYTE



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `ARR08X_TO_BYTE` dient der Konvertierung eines Arrays von acht booleschen Werten (`BOOL`) in einen einzigen Byte-Wert (`BYTE`). Jedes Bit im resultierenden Byte entspricht dabei einem Element des Eingabe-Arrays. Diese Operation ist nützlich, um mehrere binäre Signale (z. B. Schalterzustände oder Fehlerflags) platzsparend zu bündeln und für die weitere Verarbeitung oder Kommunikation vorzubereiten.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ` (Request): Löst die Konvertierungsoperation aus. Bei einem ankommenden Ereignis an diesem Eingang wird der aktuelle Wert des Eingangsarrays `IN` gelesen und in ein Byte umgewandelt.

Ereignis-Ausgänge

- `CNF` (Confirm): Wird ausgelöst, sobald die Konvertierung abgeschlossen ist. Dieses Ereignis signalisiert, dass der Ausgangswert (das Byte) gültig und aktuell ist.

Daten-Eingänge

- `IN` (Array[0..7] of BOOL): Das Eingabe-Array, das die acht zu konvertierenden booleschen Werte enthält. Der Index `0` entspricht dem niederwertigsten Bit (LSB, Bit 0) des resultierenden Bytes, der Index `7` dem höchstwertigen Bit (MSB, Bit 7).

Daten-Ausgänge

- `OUT` (BYTE): Der Ausgangswert, der aus den acht booleschen Eingängen zusammengesetzt ist. Der Datenausgang ist nicht explizit benannt (leerer Name in der Schnittstelle), wird aber durch den Ereignisausgang `CNF` begleitet.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Empfang eines `REQ`-Ereignisses führt der Block eine bitweise Zuweisung durch. Jedes Bit des internen Ergebnisbytes (`ARR08X_TO_BYTE`) wird direkt dem entsprechenden Element des Eingabe-Arrays `IN` zugewiesen. Nach Abschluss dieser Zuweisungen wird das `CNF`-Ereignis generiert, und der berechnete Byte-Wert steht am Ausgang zur Verfügung.

Die zugrunde liegende ST-Code-Logik ist:

```
ARR08X_TO_BYTE.0 := IN[0];
ARR08X_TO_BYTE.1 := IN[1];
...
ARR08X_TO_BYTE.7 := IN[7];
```

Technische Besonderheiten

- **Fest verdrahtete Größe:** Der Block ist spezifisch für die Konvertierung von genau 8 booleschen Werten. Für andere Array-Größen sind andere Blöcke erforderlich.
- **Direkte Bit-Zuordnung:** Die Abbildung ist deterministisch und erfolgt ohne jegliche arithmetische Operationen, was eine sehr schnelle und ressourcenschonende Ausführung ermöglicht.
- **Index-Bit-Mapping:** Es ist kritisch, die Zuordnung von Array-Index zu Bit-Position im Byte zu beachten: `IN[0]` ⟶ Bit 0 (LSB), `IN[7]` ⟶ Bit 7 (MSB).

Zustandsübersicht

Der Funktionsblock ist zustandslos (stateless). Er besitzt keinen internen Speicher zwischen den Aufrufen. Seine Ausgabe ist zu jedem Zeitpunkt ausschließlich eine direkte Funktion der aktuellen Eingangswerte beim Eintreffen des `REQ`-Ereignisses.

Anwendungsszenarien

- **Kommunikationsprotokolle:** Bündeln mehrerer binärer Gerätestatusse in ein einzelnes Datenbyte für die Übertragung über serielle Schnittstellen oder Feldbusse (z. B. PROFIBUS, CANopen).
- **Speicheroptimierung:** Kompaktere Speicherung von bis zu 8 Schaltzuständen oder Alarmbedingungen in einer einzigen Byte-Variablen.
- **SPS-interne Datenhandhabung:** Vorbereitung von Daten für Blöcke oder Funktionen, die Byte- oder Wort-Datentypen erwarten, anstatt einzelner Booles.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`BOOL_TO_BYTE`-Blöcke:** Viele Bibliotheken bieten Blöcke an, die acht einzelne `BOOL`-Eingänge (z. B. `IN0`… `IN7`) zu einem `BYTE` kombinieren. `ARR08X_TO_BYTE` verwendet hingegen ein Array als Eingabe, was die

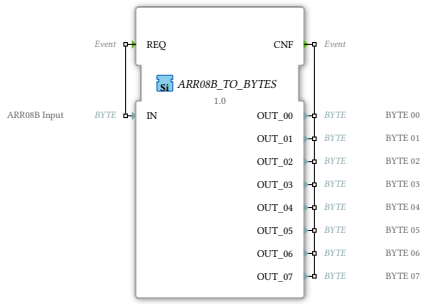
Verdrahtung bei vielen Signalen aus einer gemeinsamen Quelle (z. B. einem vorherigen Array-Ausgang) oft übersichtlicher gestaltet.
- **`MERGE`/`PACK`-Blöcke:** Allgemeinere Blöcke können verschiedene Datentypen zusammenführen. `ARR08X_TO_BYTE` ist spezialisierter, einfacher und für seinen spezifischen Zweck effizienter.
- **Bitweises Schieben (`SHL`, `OR`):** Die gleiche Funktion könnte mit einer Schleife und bitweisen Operationen in ST realisiert werden. Dieser vorgefertigte Block bietet eine standardisierte, sofort einsatzbereite und fehlerunanfälligere Lösung.

Fazit

Der `ARR08X_TO_BYTE`-Funktionsblock ist ein einfaches, aber äußerst nützliches Werkzeug für die Signalverarbeitung in IEC 61499-Systemen. Er bietet eine elegante und performante Methode, um acht boolesche Werte aus einem Array in ein kompaktes Byte zu konvertieren. Seine klare Schnittstelle und deterministische Funktionsweise machen ihn zu einer zuverlässigen Komponente für Anwendungen in der Automatisierungstechnik, insbesondere dort, wo Daten für die Kommunikation oder platzoptimierte Speicherung aufbereitet werden müssen.

forwarding

ARR08B_TO_BYTES



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `ARR08B_TO_BYTES` dient der Zerlegung eines 8-Elemente-Byte-Arrays (`ARR08B`) in acht einzelne Byte-Ausgänge. Er ist ein Hilfsbaustein für die Datenkonvertierung und -weiterleitung innerhalb von Steuerungsapplikationen, insbesondere wenn strukturierte Daten aus einem Array in diskrete Signale aufgeteilt werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Dieses eingehende Ereignis löst die Verarbeitung des Funktionsblocks aus. Bei seinem Eintreffen werden die Daten am Eingang `IN` gelesen und auf die einzelnen Ausgänge verteilt.

Ereignis-Ausgänge

- **CNF**: Dieses ausgehende Ereignis signalisiert den Abschluss der Verarbeitung. Es wird gleichzeitig mit der Ausgabe der neuen Werte an den Datenausgängen `OUT_00` bis `OUT_07` erzeugt.

Daten-Eingänge

- **IN** (`ARRAY[0..7] OF BYTE`): Der Eingang für das 8-Byte-Array, das zerlegt werden soll. Das Array hat einen festen Indexbereich von 0 bis 7.

Daten-Ausgänge

- **OUT_00** (`BYTE`): Enthält den Wert des Array-Elements mit Index 0 (`IN[0]`).
- **OUT_01** (`BYTE`): Enthält den Wert des Array-Elements mit Index 1 (`IN[1]`).
- **OUT_02** (`BYTE`): Enthält den Wert des Array-Elements mit Index 2 (`IN[2]`).
- **OUT_03** (`BYTE`): Enthält den Wert des Array-Elements mit Index 3 (`IN[3]`).
- **OUT_04** (`BYTE`): Enthält den Wert des Array-Elements mit Index 4 (`IN[4]`).
- **OUT_05** (`BYTE`): Enthält den Wert des Array-Elements mit Index 5 (`IN[5]`).

- **OUT_06** (BYTE): Enthält den Wert des Array-Elements mit Index 6 (IN[6]).
- **OUT_07** (BYTE): Enthält den Wert des Array-Elements mit Index 7 (IN[7]).

Adapter

Dieser Funktionsblock verfügt über keine Adapter-Schnittstellen.

Funktionsweise

Die Funktionsweise ist deterministisch und einfach: Bei jedem Eintreffen des REQ-Ereignisses liest der Baustein das komplette Eingangsarray IN. Anschließend wird jedes Element des Arrays einem spezifischen Byte-Ausgang zugewiesen. Die Zuordnung erfolgt direkt entsprechend dem Array-Index. Nachdem alle Zuweisungen durchgeführt wurden, wird das CNF-Ereignis ausgelöst, um den erfolgreichen Abschluss der Operation an nachfolgende Blöcke zu melden.

Der enthaltene ST-Code (Structured Text) zeigt die direkte Implementierung:

```
OUT_00 := IN[0];
OUT_01 := IN[1];
...
OUT_07 := IN[7];
```

Technische Besonderheiten

- **Feste Größe:** Der Block ist spezifisch für Arrays mit exakt 8 Elementen (ARR08B) ausgelegt. Für Arrays anderer Längen sind separate Blöcke erforderlich.
- **Synchrone Verarbeitung:** Die Verarbeitung erfolgt in einem einzigen Zyklus ohne Verzögerung. Die Latenz zwischen REQ und CNF ist minimal.
- **Kein internen Zustand:** Der Block besitzt keinen internen Speicher. Die Ausgabe ist zu jedem Zeitpunkt eine direkte Funktion der aktuellen Eingabe.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände im Sinne einer Zustandsmaschine. Sein Verhalten ist rein kombinatorisch und wird vollständig durch die Schnittstellenereignisse gesteuert:

1. **Ruhezustand:** Warten auf das REQ-Ereignis.
2. **Ausführungszustand:** Bei REQ werden die Daten transferiert und sofort das CNF-Ereignis generiert. Der Block kehrt unmittelbar danach in den Ruhezustand zurück.

Anwendungsszenarien

- **Datenverteilung:** Aufteilung eines empfangenen Datenpakets (z.B. von einer Kommunikationsschnittstelle) in seine einzelnen Byte-Komponenten für die weitere Verarbeitung in der Applikation.
- **Schnittstellen-Adapter:** Anbindung von Funktionsblöcken, die mit Array-Strukturen arbeiten, an Blöcke, die einzelne Byte-Eingänge erwarten.
- **Datenprotokollierung und -visualisierung:** Separierung von Array-Daten, um einzelne Bytes in HMIs oder Logging-Systemen gezielt anzuzeigen oder zu speichern.

⚖️ Vergleich mit ähnlichen Bausteinen

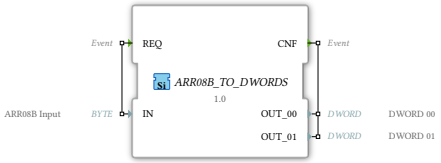
- **SEL_n / MUX-Blöcke:** Diese Blöcke wählen ein Element aus einem Array basierend auf einem Index aus. ARR08B_TO_BYTES hingegen extrahiert **alle** Elemente gleichzeitig und parallel.
- **BYTES_TO_ARR08B:** Dies ist der inverse Baustein, der acht einzelne Byte-Eingänge zu einem 8-Byte-Array zusammenfasst.
- **Generische Array-Zerlegungsblöcke:** Es könnten generischere Blöcke existieren, die die Array-Länge als Parameter erhalten. ARR08B_TO_BYTES ist durch seine feste Größe einfacher, schneller und benötigt keine Konfiguration.

Fazit

Der ARR08B_TO_BYTES Funktionsblock ist ein spezialisierter, effizienter und zuverlässiger Baustein für die grundlegende Operation der Array-Zerlegung. Seine Stärke liegt in der Einfachheit und Eindeutigkeit für den spezifischen Anwendungsfall eines 8-Byte-Arrays. Für Steuerungsanwendungen, in denen diese Datenstruktur häufig vorkommt, bietet er eine optimierte und wartungsfreundliche Lösung gegenüber der manuellen Indexierung in komplexeren Blöcken.

ARR08B_TO_DWORDS

Dieser Funktionsbaustein konvertiert ein Byte-Array der Länge 8 (8 Bytes) in zwei 32-Bit-Datenwörter (DWORDs). Er dient zur Aufteilung eines zusammenhängenden 8-Byte-Datenblocks in zwei 4-Byte große Integer-Werte.



• – ○ • – ○ • – ○ •

Einleitung

Der **ARR08B_TO_DWORDS** ist ein reiner Konvertierungsbaustein, der aus einem eingehenden 8-Byte-Array zwei separate DWORD-Werte erzeugt. Die Umwandlung erfolgt byteweise, wobei die ersten vier Bytes dem ersten Ausgang (OUT_00) und die letzten vier Bytes dem zweiten Ausgang (OUT_01) zugeordnet werden. Der Baustein wird über ein Ereignis (REQ) angestoßen und gibt nach Abschluss der Konvertierung ein Bestätigungsereignis (CNF) aus.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung	Mitgeführte Daten
REQ	Startet die Konvertierung des Eingangsarrays.	IN (8-Byte-Array)

Ereignis-Ausgänge

Ereignis	Beschreibung	Mitgeführte Daten
CNF	Bestätigt die abgeschlossene Konvertierung.	OUT_00, OUT_01

Daten-Eingänge

Name	Datentyp	Array-Dimension	Beschreibung
IN	BYTE	[0..7]	8-Byte-Eingangsarray, das in zwei DWORDs zerlegt wird.

Daten-Ausgänge

Name	Datentyp	Beschreibung
OUT_00	DWORD	Erster DWORD (Bytes 0–3 des Arrays).
OUT_01	DWORD	Zweiter DWORD (Bytes 4–7 des Arrays).

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Baustein führt eine direkte byteweise Kopie aus dem Eingangsarray in die Ausgangsvariablen durch:

1. Die Bytes **IN[0]** bis **IN[3]** werden den niederwertigsten bis höherwertigsten Bytes von **OUT_00** zugewiesen.
2. Die Bytes **IN[4]** bis **IN[7]** werden entsprechend **OUT_01** zugeordnet.

Die Zuweisung erfolgt im Structured-Text-Rumpf:

```
OUT_00.%B0 := IN[0];
OUT_00.%B1 := IN[1];
OUT_00.%B2 := IN[2];
```

```
OUT_00.%B3 := IN[3];
OUT_01.%B0 := IN[4];
OUT_01.%B1 := IN[5];
OUT_01.%B2 := IN[6];
OUT_01.%B3 := IN[7];
```

Der Baustein arbeitet ohne Verzögerung oder Zustandsspeicher; die Ausgänge werden unmittelbar nach dem Eintreffen des REQ-Ereignisses aktualisiert und das CNF-Ereignis wird sofort ausgelöst.

Technische Besonderheiten

- **Endianness**: Der Baustein interpretiert das Array im *Little-Endian*-Format – das erste Byte (Index 0) wird dem niederwertigsten Byte des DWORD zugeordnet. Diese Annahme ist typisch für viele industrielle Protokolle und Systeme.
- **Keine Prüfung der Array-Länge**: Das Array wird als genau 8 Byte vorausgesetzt. Eine abweichende Eingangsgröße würde zu undefiniertem Verhalten führen.
- **Funktionaler Baustein**: Der FB ist rein funktional und besitzt keine internen Zustände.

Zustandsübersicht

Der Baustein hat keinen internen Zustandsautomaten. Er arbeitet ereignisgesteuert und gibt nach jeder Aktivierung sofort das CNF-Signal zurück. Die Ausgänge bleiben bis zur nächsten REQ-Aktivierung erhalten.

Anwendungsszenarien

- **Datenpaket-Zerlegung**: Trennung eines 8-Byte-Telegramms in zwei 32-Bit-Werte, z. B. bei der Kommunikation mit industriellen Feldgeräten (CANopen, Modbus, etc.).
- **Sensorwertverarbeitung**: Aufteilung eines 8-Byte-Rohdatenblocks von Sensoren, die zwei physikalische Messwerte (je 4 Byte) liefern.
- **Konvertierung in der logiBUS-Umgebung**: Innerhalb des Pakets `logiBUS::utils::conversion::arr::forwarding` als Standardbaustein zur Vorwärtskonvertierung von Array-Daten.

Vergleich mit ähnlichen Bausteinen

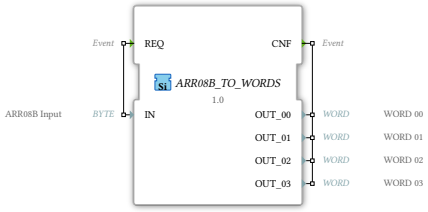
Baustein	Funktion	Unterschied
BYTE_TO_DWORD	Einzeln konvertiert ein BYTE-Array von 4 Bytes in ein DWORD.	Benötigt separate Aufrufe für jedes 4-Byte-Segment.
ARR08B_TO_DWORDS	Konvertiert ein 8-Byte-Array auf einmal in zwei DWORDs.	Integrierte Aufteilung in nur einem Schritt, reduziert Aufrufanzahl und Code.

Der vorliegende Baustein bietet somit eine kompaktere Lösung als die Kombination mehrerer elementarer Konvertierungsbausteine.

Fazit

Der **ARR08B_TO_DWORDS** ist ein einfacher, aber nützlicher Funktionsbaustein zur schnellen und direkten Zerlegung eines 8-Byte-Arrays in zwei DWORDs. Dank seiner klaren Byte-zu-Byte-Zuordnung und der Ereignissteuerung lässt er sich gut in datenflussorientierte Steuerungsanwendungen integrieren. Die Verwendung setzt ein Little-Endian-Format voraus; für andere Endianness-Anforderungen sind Anpassungen erforderlich. Insgesamt ist der Baustein eine effiziente Lösung für die Vorverarbeitung binärer Daten.

ARR08B_TO_WORDS



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **ARR08B_TO_WORDS** dient der Aufteilung eines 8-Byte-Arrays in vier 16-Bit-Wörter (WORD). Er wird typischerweise in der Datenvorverarbeitung eingesetzt, wenn rohe Byte-Daten aus Sensoren oder Kommunikationsschnittstellen in ganzzahlige Werte umgewandelt werden müssen. Die Umwandlung erfolgt byteweise unter Berücksichtigung der Little-Endian-Bytereihenfolge.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung
REQ	Auslösen der Konvertierung; der Baustein verarbeitet den aktuellen Wert von `IN` und legt die Ergebnisse an den Ausgängen an.

Ereignis-Ausgänge

Ereignis	Beschreibung
CNF	Bestätigung, dass die Konvertierung abgeschlossen ist und die Ausgangswerte gültig sind.

Daten-Eingänge

Name	Typ	Array-Größe	Beschreibung
IN	BYTE	0…7	Eingangsarray aus 8 Bytes, das in vier Words zerlegt wird.

Daten-Ausgänge

Name	Typ	Beschreibung
OUT_00	WORD	Aus den Bytes IN[0] und IN[1] gebildetes Word (niederwertigstes).
OUT_01	WORD	Aus den Bytes IN[2] und IN[3] gebildetes Word.
OUT_02	WORD	Aus den Bytes IN[4] und IN[5] gebildetes Word.
OUT_03	WORD	Aus den Bytes IN[6] und IN[7] gebildetes Word (höchstwertigstes).

Adapter

Keine Adapter vorhanden.

Funktionsweise

Bei Aktivierung des Ereignisses `REQ` wird das 8-Byte-Array `IN` sequenziell abgearbeitet. Jeweils zwei aufeinanderfolgende Bytes werden in ein 16-Bit-Wort zusammengefasst, wobei das erste Byte (`IN[0]`) das niederwertige Byte (`%B0`) und das zweite Byte (`IN[1]`) das höherwertige Byte (`%B1`) des Wortes bildet (Little-Endian-Format). Die resultierenden vier Wörter werden an den Ausgängen `OUT_00` bis `OUT_03` bereitgestellt. Anschließend wird das Ereignis `CNF` gesendet.

Technische Besonderheiten

- **Byte-Reihenfolge:** Die Konvertierung erfolgt stets im Little-Endian-Format, d.h. das zuerst im Array stehende Byte wird zum niederwertigen Teil des Wortes.
- **Keine Pufferung:** Der Baustein ist rein kombinatorisch und speichert keine Zustände. Jeder `REQ`-Aufruf verarbeitet den aktuellen Eingangswert.
- **Typisierung:** Eingänge sind als Array von `BYTE` (0-Indizierung), Ausgänge als `WORD` definiert. Der Baustein setzt keine weiteren Typumwandlungen voraus.

Zustandsübersicht

Der Baustein besitzt keinen internen Zustandsautomaten. Die Verarbeitung erfolgt streng ereignisgesteuert:

1. Warten auf `REQ`.
2. Bei `REQ`: Kopieren der Bytes in die Ausgangswörter.
3. Senden von `CNF` und Rückkehr in den Wartezustand.

Anwendungsszenarien

- **Sensorauslesung:** Ein Temperatursensor liefert 8 Rohbytes, die in vier 16-Bit-Messwerte (z.B. zwei Kanäle mit je einem Word) aufgeteilt werden müssen.
- **Protokoll-Dekodierung:** Ein serielles Protokoll überträgt Nutzdaten als Byte-Array; der Baustein extrahiert daraus vier 16-Bit-Registerwerte.
- **Datenvorverarbeitung in SPS-Systemen:** Zusammenführen mehrerer Byte-orientierter Schnittstellen (z.B. Profibus, CAN) in einheitliche Word-Signale für die weitere Verarbeitung.

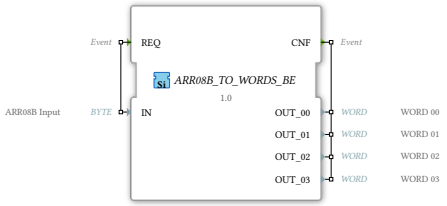
Vergleich mit ähnlichen Bausteinen

- **`BYTE_TO_WORD`:** Wandelt nur zwei Bytes in ein einzelnes Word um, benötigt bei 8 Bytes vier Aufrufe.
- **`ARRAY_TO_WORD` (allgemein):** Meist parametrierbar, z.B. mit Startindex und Anzahl – `ARR08B_TO_WORDS` ist dagegen fest auf 8 Bytes und 4 Words ausgelegt und dadurch einfacher und performanter in diesem spezifischen Fall.
- **`WORD_TO_ARR08B` (gegebenenfalls vorhanden):** Die inverse Funktion, die vier Words wieder in ein 8-Byte-Array packt.

Fazit

Der Funktionsbaustein **ARR08B_TO_WORDS** bietet eine schlanke, spezialisierte Lösung zum Zerlegen eines 8-Byte-Arrays in vier 16-Bit-Wörter. Durch die feste Verdrahtung der Byte-Zuordnung wird eine hohe Ausführungsgeschwindigkeit und einfache Handhabung erreicht, insbesondere in zeitkritischen Umgebungen oder bei wiederkehrenden Konvertierungsaufgaben. Die Little-Endian-Interpretation ist für viele industrielle und eingebettete Systeme der Standard, sodass der Baustein ohne zusätzliche Parameter auskommt.

ARR08B_TO_WORDS_BE



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `ARR08B_TO_WORDS_BE` extrahiert aus einem 8-Byte-Array (Big-Endian) vier 16-Bit-Wörter (WORD). Er wird hauptsächlich zur Konvertierung von binären Datenströmen eingesetzt, die in einem aufsteigenden Byte-Array vorliegen und als geordnete Wortfolge interpretiert werden sollen. Der Baustein gehört zur Paketgruppe `logiBUS::utils::conversion::arr::forwarding`.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung
`REQ`	Startet die Konvertierung: Das anliegende Byte-Array `IN` wird in die vier Ausgangswörter umgesetzt.

Ereignis-Ausgänge

Ereignis	Beschreibung

`CNF`	Bestätigt die erfolgreiche Umwandlung und signalisiert, dass die Ausgangswerte gültig sind.

Daten-Eingänge

Name	Typ	Array-Größe	Beschreibung
`IN`	`BYTE`	`0..7` (8 Bytes)	Eingangsarray – die Rohdaten, aus denen die Wörter extrahiert werden.

Daten-Ausgänge

Name	Typ	Beschreibung
`OUT_00`	`WORD`	Wort aus Byte 0 (High) und Byte 1 (Low)
`OUT_01`	`WORD`	Wort aus Byte 2 (High) und Byte 3 (Low)
`OUT_02`	`WORD`	Wort aus Byte 4 (High) und Byte 5 (Low)
`OUT_03`	`WORD`	Wort aus Byte 6 (High) und Byte 7 (Low)

Adapter

Keine Adapter vorhanden.

Funktionsweise

Beim Eintreffen eines `REQ`-Ereignisses werden die acht Bytes des Eingangsarrays `IN` in strenger **Big-Endian-Reihenfolge** in vier 16-Bit-Wörter verpackt:

- `OUT_00 = (IN[0] << 8) | IN[1]`
- `OUT_01 = (IN[2] << 8) | IN[3]`
- `OUT_02 = (IN[4] << 8) | IN[5]`
- `OUT_03 = (IN[6] << 8) | IN[7]`

Anschließend wird das Ausgangsereignis `CNF` ausgelöst, das die Gültigkeit aller Ausgabewerte anzeigt. Die Implementierung erfolgt direkt im Structured-Text-Teil des Funktionsblocks, ohne internen Zustandsautomaten.

Technische Besonderheiten

- **Big-Endian-Konvention:** Das erste Byte eines Wortes (`IN[0]`, `IN[2]`, …) bildet das höherwertige Byte (`%B1`), das zweite Byte (`IN[1]`, `IN[3]`, …) das niederwertige Byte (`%B0`).
- **Feste Array-Größe:** Es werden exakt 8 Bytes erwartet; das Array ist als `ARRAY[0..7] OF BYTE` deklariert.
- **Keine Nebeneffekte:** Der Baustein ist rein kombinatorisch – er benötigt keinen internen Zustand und arbeitet innerhalb eines einzigen Ereigniszyklus.
- **Typischer Einsatz:** Konvertierung von Daten aus binären Protokollen (z. B. CAN-Nachrichten, Modbus-Registern oder seriellen Streams), die als Byte-Array vorliegen.

Zustandsübersicht

Der Baustein besitzt keinen eigenen Zustandsautomaten. Er wird durch das **Ereignis `REQ`** getriggert und gibt nach der Datenverarbeitung **`CNF`** aus. Die Ausführung erfolgt innerhalb eines einzigen Zyklus.

Zustand	Beschreibung
Idle	Warten auf `REQ`; Ausgänge behalten den letzten Wert.
Processing	`REQ` eingetroffen – Daten werden umgesetzt.
Completed	`CNF` wird gesendet, Ausgänge sind gültig. Baustein kehrt sofort in den Ruhezustand zurück.

Anwendungsszenarien

1. **Modbus-Kommunikation:** Ein 8-Byte-Registerblock (z. B. Holding-Register) wird als Byte-Array empfangen und muss in vier 16-Bit-Werte aufgeteilt werden.
2. **CAN-Botschaften:** Ein 8-Byte-CAN-Frame enthält mehrere 16-Bit-Sensorwerte, die im Big-Endian-Format übertragen werden.
3. **Serielle Datenanalyse:** Zusammenfassen aufeinanderfolgender Bytes aus einer seriellen Schnittstelle zu nutzbaren WORD-Größen für die Weiterverarbeitung.
4. **Gateway-Funktionen:** Konvertierung von Binär- zu Allzweckdaten in Automatisierungsprozessen, die auf IEC 61499 basieren.

Vergleich mit ähnlichen Bausteinen

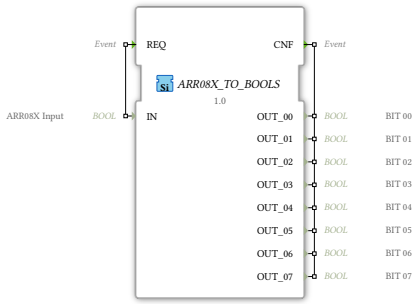
Baustein	Unterschied
`ARR08B_TO_WORDS_LE`	Verwendet Little-Endian-Reihenfolge (Byte[0] = Low-Byte).
`ARR04B_TO_WORD`	Arbeitet mit einem 4-Byte-Array und erzeugt nur ein WORD.
`BYTE_TO_WORD`	Konvertiert zwei einzelne Bytes in ein WORD; benötigt separate Zusammenstellung.

`ARR08B_TO_WORDS_BE` fokussiert auf die effiziente, vordefinierte Aufteilung eines 8-Byte-Arrays in vier Wörter unter Big-Endian-Reihenfolge – ideal für standardisierte Protokolle.

Fazit

Der Funktionsblock `ARR08B_TO_WORDS_BE` bietet eine einfache und zuverlässige Methode, ein 8-Byte-Array in vier Big-Endian-Wörter umzuwandeln. Durch die kompakte Schnittstelle (`REQ`/`CNF` + ein Eingang, vier Ausgänge) lässt er sich leicht in bestehende IEC-61499-Applikationen integrieren. Dank der direkten ST-Implementierung entstehen keine Latenzen durch Zustandsautomaten, was ihn besonders für zeitkritische Anwendungen in der Automatisierungstechnik empfiehlt.

ARR08X_TO_BOOLS



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock **ARR08X_TO_BOOLS** dient der Umwandlung eines 8-elementigen BOOL-Arrays in acht einzelne BOOL-Signale. Er extrahiert jedes Bit aus dem Eingangsarray und stellt es über einen eigenen, benannten Datenausgang zur Verfügung. Dieser Baustein ist besonders nützlich, um kompakte Array-Daten aus übergeordneten Systemen oder Kommunikationsschnittstellen in einzelne, leicht zu verarbeitende logische Signale aufzuschlüsseln.

Schnittstellenstruktur

Der Baustein besitzt eine einfache, unidirektionale Schnittstelle mit einem Anforderungs-/Bestätigungs-Ereignispaar.

Ereignis-Eingänge

- **REQ**: Dieses Ereignis löst die Verarbeitung aus. Bei seinem Eintreffen werden die Werte des Eingangsarrays `IN` gelesen und auf die einzelnen Ausgänge `OUT_00` bis `OUT_07` verteilt.

Ereignis-Ausgänge

- **CNF**: Dieses Ereignis wird ausgelöst, sobald die Extraktion der Booleschen Werte abgeschlossen ist. Es bestätigt die Ausführung des Bausteins und transportiert die neuen Werte aller acht Datenausgänge (`OUT_00` bis `OUT_07`).

Daten-Eingänge

- **IN** (ARRAY[0..7] OF BOOL): Das Eingangsarray, das die acht zu extrahierenden Booleschen Werte enthält. Der Index 0 entspricht dem niederwertigsten Bit (LSB) und wird auf `OUT_00` abgebildet.

Daten-Ausgänge

- **OUT_00** (BOOL): Extrahiertes Bit 0 (LSB) des Eingangsarrays `IN`.
- **OUT_01** (BOOL): Extrahiertes Bit 1 des Eingangsarrays `IN`.
- **OUT_02** (BOOL): Extrahiertes Bit 2 des Eingangsarrays `IN`.
- **OUT_03** (BOOL): Extrahiertes Bit 3 des Eingangsarrays `IN`.
- **OUT_04** (BOOL): Extrahiertes Bit 4 des Eingangsarrays `IN`.
- **OUT_05** (BOOL): Extrahiertes Bit 5 des Eingangsarrays `IN`.
- **OUT_06** (BOOL): Extrahiertes Bit 6 des Eingangsarrays `IN`.
- **OUT_07** (BOOL): Extrahiertes Bit 7 (MSB) des Eingangsarrays `IN`.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Die Funktionsweise ist deterministisch und nicht-zustandsbehaftet. Bei jedem Eintreffen des `REQ`-Ereignisses führt der Baustein eine direkte Zuweisung durch. Jedes Element des Eingangsarrays `IN` wird seinem entsprechenden, einzeln benannten Ausgangsvariable zugewiesen. Die Logik entspricht folgender ST-Zuweisung:

```
OUT_00 := IN[0];
OUT_01 := IN[1];
OUT_02 := IN[2];
OUT_03 := IN[3];
OUT_04 := IN[4];
OUT_05 := IN[5];
OUT_06 := IN[6];
OUT_07 := IN[7];
```

Unmittelbar nach Abschluss dieser Operation wird das `CNF`-Ereignis mit den neuen Werten generiert.

Technische Besonderheiten

- **Echtzeitverhalten:** Der Baustein führt nur Zuweisungen aus und verursacht keine nennenswerte Latenz. Die Ausführung ist in einem Zyklus abgeschlossen.
- **Speicher:** Es wird kein interner Zustand zwischen den Aufrufen gespeichert. Die Ausgabe ist eine reine Funktion der aktuellen Eingabe.
- **Array-Grenzen:** Der Baustein ist spezifisch für Arrays der festen Größe 8 (Indizes 0 bis 7) definiert. Die Verwendung eines Arrays anderer Größe führt zu einem Kompilierfehler.

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand (zustandslos). Sein Verhalten ist bei jedem Aufruf identisch und hängt ausschließlich von den aktuellen Werten am Daten-Eingang `IN` ab.

Anwendungsszenarien

- **Entpacken von Bus-Telegrammen:** Extraktion einzelner Status- oder Steuerbits aus einem Byte, das über Feldbusse (z.B. CAN, PROFIBUS) oder industrielle Netzwerke empfangen wurde.
- **Schnittstelle zu Bit-orientierten Komponenten:** Aufbereitung von Array-Daten für die Ansteuerung von einzelnen Lampen, Ventilen oder Relais, die über diskrete BOOL-Ausgänge angesprochen werden.
- **Vereinfachung der Visualisierung:** Bereitstellung von einzeln adressierbaren Signalen für HMIs oder Alarmierungsysteme, anstatt mit Array-Indizes arbeiten zu müssen.
- **Dekodierung:** Umsetzung einer einfachen 8-Bit-Binärzahl in ihre logischen Einzelwerte.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`SEL` / `MUX` Bausteine:** Diese wählen ein einzelnes Element aus einem Array basierend auf einem Index aus. `ARR08X_TO_BOOLS` hingegen extrahiert **alle** Elemente gleichzeitig und parallel.
- **`BYTE_TO_BOOLS` / `WORD_TO_BOOLS`:** Diese Bausteine zerlegen ein Byte- oder Wort-Datenelement in seine einzelnen Bits. `ARR08X_TO_BOOLS` arbeitet direkt auf einem Array von BOOLs, was semantisch klarer ist, wenn die Quelle bereits ein BOOL-Array ist. Die funktionale Wirkung auf die Bits ist jedoch äquivalent.
- **Manuelle Indexierung:** Statt dieses Bausteins zu verwenden, könnte man im umgebenden Algorithmus direkt auf `IN[0]`, `IN[1]` usw. zugreifen. Der Funktionsblock bietet jedoch eine klarere Schnittstellenbeschreibung, verbessert die Lesbarkeit des FBD-Netzwerks und kapselt die Funktionalität.

🛠️ Zugehörige Übungen

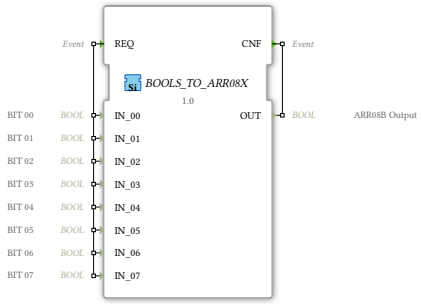
- Uebung_054

Fazit

Der `ARR08X_TO_BOOLS` ist ein spezialisierter, aber sehr nützlicher Konvertierungsbaustein. Er stellt eine einfache und effiziente Methode dar, um ein kompaktes 8-Bit-BOOL-Array in acht diskrete Signale zu entpacken. Durch seine klare, zustandslose Funktionsweise und die explizite Benennung aller Ausgänge trägt er erheblich zur Verbesserung der Lesbarkeit und Wartbarkeit von Steuerungsapplikationen bei, in denen mit array-basierten Datenstrukturen gearbeitet wird.

reversing

BOOLS_TO_ARR08X



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `BOOLS_TO_ARR08X` dient der Konvertierung von acht einzelnen booleschen (`BOOL`) Eingangssignalen in ein boolesches Array (`ARRAY[0..7] OF BOOL`). Er ist ein grundlegender Baustein zur Signalzusammenführung und Datenstrukturierung in Steuerungsanwendungen, insbesondere wenn diskrete digitale Signale für eine kompakte Weiterverarbeitung oder Übertragung gebündelt werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ` (Request): Löst die Konvertierungsoperation aus. Bei Eintreffen dieses Ereignisses werden die aktuellen Werte an den acht `IN_xx`-Eingängen gelesen und in das Ausgangsarray `OUT` geschrieben.

Ereignis-Ausgänge

- `CNF` (Confirm): Wird ausgelöst, sobald die Konvertierung abgeschlossen ist und das Ausgangsarray `OUT` mit den neuen Werten bereitsteht.

Daten-Eingänge

Acht einzelne boolesche Eingänge (`BOOL`), die jeweils einem Bit im resultierenden Array entsprechen:

- `IN_00`: Wert für Bit 00 / Array-Index 0.
- `IN_01`: Wert für Bit 01 / Array-Index 1.
- `IN_02`: Wert für Bit 02 / Array-Index 2.

- `IN_03`: Wert für Bit 03 / Array-Index 3.
- `IN_04`: Wert für Bit 04 / Array-Index 4.
- `IN_05`: Wert für Bit 05 / Array-Index 5.
- `IN_06`: Wert für Bit 06 / Array-Index 6.
- `IN_07`: Wert für Bit 07 / Array-Index 7.

Daten-Ausgänge

- `OUT`: Das Ausgangssignal vom Typ `ARRAY[0..7] OF BOOL`. Nach Auslösung des `REQ`-Ereignisses enthält es die zu diesem Zeitpunkt an den Eingängen anliegenden Werte, wobei `IN_00` auf `OUT[0]` und `IN_07` auf `OUT[7]` abgebildet wird.

Adapter

Dieser Funktionsblock besitzt keine Adapterschnittstellen.

Funktionsweise

Die Funktionsweise ist deterministisch und einfach. Bei jedem Eintreffen eines Ereignisses am `REQ`-Eingang wird der enthaltene ST-Code ausgeführt. Dieser Code weist jedem Element des Ausgangsarrays `OUT` direkt den Wert des entsprechenden einzelnen `IN_xx`-Eingangs zu. Die Abbildung ist fest verdrahtet: `OUT[0] := IN_00; OUT[1] := IN_01; ... OUT[7] := IN_07;` Unmittelbar nach dieser Zuweisung wird das `CNF`-Ereignis generiert, um den erfolgreichen Abschluss der Operation zu signalisieren. Es findet keine Pufferung, Verzögerung oder logische Verknüpfung der Eingangswerte statt.

Technische Besonderheiten

- **Echtzeitverhalten:** Die Operation ist atomar und sehr schnell, da sie nur aus direkten Zuweisungen besteht. Sie fügt keine nennenswerte Latenz in den Steuerungszyklus ein.
- **Größenfestlegung:** Das Ausgangsarray ist mit der festen Größe `[0..7]` definiert, was genau acht Elementen entspricht.
- **Synchrone Abarbeitung:** Alle Eingangswerte werden zum Zeitpunkt des `REQ`-Ereignisses synchron erfasst. Änderungen an den Eingängen während der Abarbeitung haben keinen Einfluss auf den aktuellen Zyklus.

Zustandsübersicht

Der Funktionsblock ist zustandslos (stateless). Er besitzt kein internes Gedächtnis. Seine Ausgabe ist zu jedem Zeitpunkt ausschließlich eine Funktion der aktuellen Eingangswerte beim letzten `REQ`-Ereignis.

Anwendungsszenarien

- **Bündelung von Schaltzuständen:** Zusammenfassen der Zustände mehrerer Endschalter, Taster oder Ventile in ein einziges Array für die Übertragung über ein Netzwerk oder an einen übergeordneten Steuerungsbaustein.
- **Vorverarbeitung für Array-basierte Blöcke:** Bereitstellung von Eingangsdaten für Funktionsblöcke, die ihre Operationen auf ganzen Arrays durchführen (z.B. logische Verknüpfungen, Verschiebeoperationen).
- **Strukturierung von E/A-Daten:** Ordnung von verteilten digitalen Eingangssignalen einer SPS in einer definierten Datenstruktur für verbesserte Lesbarkeit und Wartbarkeit des Programms.
- **Schnittstelle zu Visualisierungssystemen:** Einfacheres Mapping von vielen Einzelsignalen auf ein zusammengesetztes Symbol oder eine Statusanzeige in einer HMI.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`BOOL_TO_xxx` Blöcke:** Blöcke wie `BOOL_TO_INT` oder `BOOL_TO_BYTE` konvertieren mehrere Bool-Werte ebenfalls in ein kompakteres Datum. `BOOLS_TO_ARR08X` erzeugt hingegen explizit ein Array, was den indizierten Zugriff auf jedes einzelne Bit erhält und die semantische Bedeutung der Position bewahrt.
- **Manuelle Verkabelung:** Die Alternative wäre die direkte Verkabelung jedes Einzelbits auf die Indizes eines Array-Variablenbausteins. `BOOLS_TO_ARR08X` kapselt diese Operation, verbessert die Übersichtlichkeit im FBD/LD-Editor und reduziert Verdrahtungsaufwand.
- **`MERGE`/`CONCAT`-Blöcke:** Allgemeine Zusammenführungsblöcke können verschiedene Datentypen verbinden. `BOOLS_TO_ARR08X` ist spezialisiert und typsicher für genau acht BOOL-Eingänge und ein BOOL-Array als Ausgang.

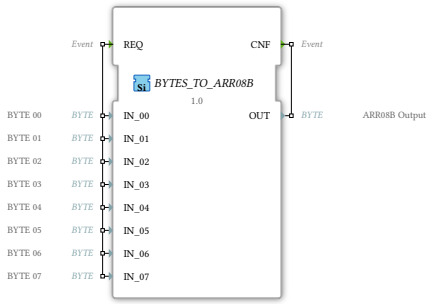
🛠️ Zugehörige Übungen

- Uebung_054

Fazit

Der `BOOLS_TO_ARR08X` ist ein spezialisierter, robuster und effizienter Funktionsblock für eine häufige Aufgabe in der industriellen Automatisierung: die Bündelung diskreter Signale. Seine Stärke liegt in der Einfachheit, der deterministischen Ablaufzeit und der klaren Schnittstellendefinition. Er ist die erste Wahl, wenn acht boolesche Signale in einer geordneten Array-Struktur für die Weiterverarbeitung bereitgestellt werden müssen.

BYTES_TO_ARR08B



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock `BYTES_TO_ARR08B` dient der Konvertierung von acht einzelnen `BYTE`-Werten in ein Array vom Typ `BYTE` mit acht Elementen. Er ist Teil einer Sammlung von Hilfsfunktionen für Datenkonvertierungen und Array-Operationen.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Dieses Ereignis löst die Verarbeitung des Funktionsblocks aus. Bei seinem Eintreffen werden die Werte an den Daten-Eingängen gelesen und in das Ausgabe-Array geschrieben.

Ereignis-Ausgänge

- **CNF**: Dieses Ereignis wird nach erfolgreicher Verarbeitung und Zuweisung der Werte zum Ausgabe-Array generiert. Es signalisiert die Verfügbarkeit des Ergebnisses.

Daten-Eingänge

- **IN_00** (`BYTE`): Wert für das Array-Element an Index 0.
- **IN_01** (`BYTE`): Wert für das Array-Element an Index 1.
- **IN_02** (`BYTE`): Wert für das Array-Element an Index 2.
- **IN_03** (`BYTE`): Wert für das Array-Element an Index 3.
- **IN_04** (`BYTE`): Wert für das Array-Element an Index 4.
- **IN_05** (`BYTE`): Wert für das Array-Element an Index 5.
- **IN_06** (`BYTE`): Wert für das Array-Element an Index 6.
- **IN_07** (`BYTE`): Wert für das Array-Element an Index 7.

Daten-Ausgänge

- **OUT** (`ARRAY[0..7] OF BYTE`): Das resultierende Byte-Array. Die Indizes 0 bis 7 enthalten nacheinander die Werte von `IN_00` bis `IN_07`.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Auslösung durch das `REQ`-Ereignis kopiert der Block die Werte der acht `BYTE`-Eingänge direkt in die entsprechenden Positionen des Ausgabe-Arrays `OUT`. Die Zuordnung ist sequenziell: `IN_00` ⟶ `OUT[0]`, `IN_01` ⟶ `OUT[1]`, usw. bis `IN_07` ⟶ `OUT[7]`. Unmittelbar nach dieser Zuweisung wird das `CNF`-Ereignis ausgelöst.

Die interne Implementierung erfolgt in Structured Text (ST) und besteht aus einer einfachen Reihe von Zuweisungen.

Technische Besonderheiten

- **Deterministische Laufzeit:** Die Operation hat eine konstante und sehr kurze Ausführungszeit, da sie nur aus direkten Zuweisungen besteht.
- **Keine Pufferung:** Es findet keine Zwischenspeicherung der Eingangswerte statt. Die Verarbeitung erfolgt ausschließlich bei Eintreffen eines `REQ`-Ereignisses.
- **Feste Größe:** Der Block ist spezifisch für die Kombination von genau acht Bytes ausgelegt.

Zustandsübersicht

Der Funktionsblock ist zustandslos (stateless). Jeder `REQ`-Aufruf ist unabhängig von vorherigen Aufrufen und verarbeitet ausschließlich die aktuell anliegenden Eingangswerte.

Anwendungsszenarien

- **Datenaggregation:** Zusammenfassen einzelner Byte-Werte (z.B. aus verschiedenen Quellen oder Sensoren) zu einem kompakten Array für die Weiterverarbeitung oder Übertragung.
- **Protokoll- und Schnittstellenanpassung:** Vorbereitung von Datenstrukturen für Kommunikationsprotokolle oder Funktionsblöcke, die ein Byte-Array anstelle einzelner Bytes erwarten.
- **Datenumstrukturierung:** Konvertierung zwischen verschiedenen Darstellungsformen von Daten innerhalb einer Steuerungsanwendung.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`CONCAT`-Blöcke:** Allgemeine Blöcke zum Verketten von Strings sind nicht direkt vergleichbar, da sie auf den Datentyp `STRING` ausgelegt sind.
- **Manuelle Zuweisung in ST:** Die Funktionalität könnte direkt in einem ST-Programm durch Zuweisungen realisiert werden. Der Funktionsblock kapselt diese Logik jedoch für eine bessere Wiederverwendbarkeit, Lesbarkeit und Wartbarkeit innerhalb von FBD-/LD-Diagrammen.
- **Individuelle Array-Bausteine:** Im Gegensatz zu universellen Array-Bausteinen, die über Indexeingänge verfügen, ist `BYTES_TO_ARR08B` für eine spezifische, feste Aufgabe optimiert und daher einfacher und schneller in der Anwendung.

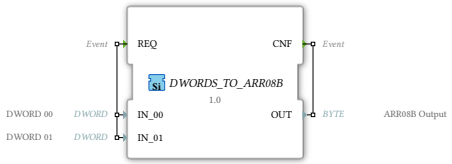
🛠️ Zugehörige Übungen

- SIN_TO_PCAN_Callback_Byte
- Uebung_12x_sub

Fazit

Der `BYTES_TO_ARR08B` ist ein spezialisierter und effizienter Hilfsbaustein für eine häufige Aufgabe in der Datenmanipulation. Er bietet eine klare, ereignisgesteuerte Schnittstelle und erspart dem Anwender die wiederholte manuelle Implementierung der Array-Zuordnung. Seine Stärke liegt in der Einfachheit und Eindeutigkeit für den konkreten Anwendungsfall der Kombination von acht Bytes.

DWORDS_TO_ARR08B



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **DWORDS_TO_ARR08B** wandelt zwei 32-Bit-DWORD-Werte in ein Array von acht Bytes (BYTE) um. Die Konvertierung erfolgt im Little-Endian-Format (Intel-Standard), d. h. das niederwertigste Byte eines DWORDs wird an die kleinste Indexposition des Ausgangsarrays geschrieben. Der Baustein ist für den Einsatz auf PC-basierten Steuerungen (z. B. Beckhoff, Wago, B&R, Raspberry Pi) optimiert.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung
REQ	Startet die Konvertierung. Die aktuellen Werte von IN_00 und IN_01 werden eingelesen.

Ereignis-Ausgänge

Ereignis	Beschreibung
CNF	Wird gesendet, sobald die Ausgabe OUT aktualisiert wurde.

Daten-Eingänge

Variable	Typ	Beschreibung
IN_00	DWORD	Erstes 32-Bit-Wort (niederwertig).
IN_01	DWORD	Zweites 32-Bit-Wort (höherwertig).

Daten-Ausgänge

Variable	Typ	Beschreibung
OUT	ARRAY[0..7] OF BYTE	Ausgabearray mit 8 Bytes (Bytes 0–3 aus IN_00, Bytes 4–7 aus IN_01).

Adapter

Keine Adapter vorhanden.

Funktionsweise

Bei einem positiven Flanke am Ereigniseingang REQ wird die folgende Logik ausgeführt:

1. Die beiden Eingangs-DWORDs IN_00 und IN_01 werden Byte für Byte ausgelesen.
2. Dabei kommen die systeminternen Byte-Zugriffe (z. B. IN_00.%B0 für das niederwertigste Byte) zum Einsatz.
3. Die Bytes werden in **Little-Endian-Reihenfolge** in das Ausgangsarray OUT kopiert:
   - OUT[0] ← niederwertigstes Byte von IN_00
   - OUT[1] ← zweites Byte von IN_00
   - OUT[2] ← drittes Byte von IN_00
   - OUT[3] ← höchstwertiges Byte von IN_00
   - OUT[4] ← niederwertigstes Byte von IN_01
   - OUT[5] ← zweites Byte von IN_01
   - OUT[6] ← drittes Byte von IN_01
   - OUT[7] ← höchstwertiges Byte von IN_01
4. Nach Abschluss der Zuweisungen wird das Ereignis CNF gesendet.

Technische Besonderheiten

- Die Byte-Reihenfolge folgt dem **Little-Endian-Format** (Intel-Standard), das auf den meisten PC-basierten Steuerungen (Windows-basierte Systeme, Raspberry Pi) nativ verwendet wird.
- Der Baustein ist als **strukturierter Text (ST)** implementiert und benötigt keine Zustandsmaschine – die Verarbeitung erfolgt linear bei jedem REQ.

- Das im ST-Code verwendete Paket `logiBUS::utils::conversion::arr::reversing` deutet auf eine wiederverwendbare Bibliothek für Konvertierungen hin.

Zustandsübersicht

Der Funktionsblock besitzt keine eigene Zustandsmaschine. Er führt bei jedem `REQ`-Ereignis eine einmalige, deterministische Berechnung durch und signalisiert die Fertigstellung mit `CNF`. Es gibt keine internen Speicherzustände, die über mehrere Aufrufe hinweg erhalten bleiben.

Anwendungsszenarien

- **Datenprotokoll-Aufbereitung**: Umwandlung von Steuerungsdaten (z. B. Positionswerte, Timer-Zähler), die als zwei DWORDs vorliegen, in ein Byte-Array für die serielle Übertragung (RS-232, CAN, Ethernet).
- **Kommunikation mit Endian-empfindlichen Geräten**: Sicherstellung der korrekten Byte-Reihenfolge beim Austausch mit Sensoren oder Aktoren, die Little-Endian erwarten.
- **Speicher-Abbildung / Union-Ersatz**: In Sprachen ohne Union-Typen dient der Baustein als portable Methode, um DWORD-Werte byteweise zu zerlegen.

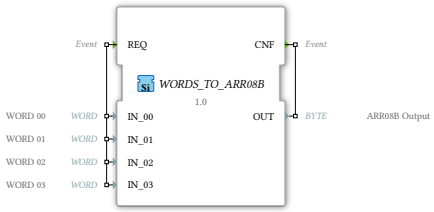
Vergleich mit ähnlichen Bausteinen

- **WORD_TO_ARR04B**: Konvertiert ein WORD in ein 4-Byte-Array – hier wird der gleiche Ansatz für zwei DWORDs erweitert.
- **DWORDS_TO_ARR08B_BIGENDIAN**: Alternativbaustein für Big-Endian-Systeme (Motorola-Standard) mit umgekehrter Byte-Reihenfolge.
- **SWAP_BYTES**: Tauscht die Bytes eines DWORDs, liefert aber kein Array. Der vorliegende Baustein erzeugt direkt das komplette 8-Byte-Array.

Fazit

Der Funktionsblock `DWORDS_TO_ARR08B` bietet eine einfache, effiziente Möglichkeit, zwei DWORDs im Little-Endian-Format in ein Byte-Array zusammenzuführen. Seine konsequente Ausrichtung auf PC-basierte Steuerungen und die klare, wartbare ST-Implementierung machen ihn zu einem zuverlässigen Werkzeug für die Datenkonvertierung in der Automatisierungstechnik.

WORDS_TO_ARR08B



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **WORDS_TO_ARR08B** dient dazu, vier 16-Bit-Wörter (Typ `WORD`) zu einem Byte-Array der Länge 8 (Typ `ARRAY[0..7] OF BYTE`) zusammenzufügen. Dabei kommt das Little-Endian-Format (Intel-Standard) zur Anwendung, das die niederwertigen Bytes zuerst ablegt. Der Baustein wird über einen Ereigniseingang gesteuert und eignet sich besonders für PC-basierte Steuerungen (z. B. Beckhoff, Wago, B&R, Raspberry Pi).

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung
`REQ`	Event	Auslösen der Konvertierung; alle Eingangsvariablen werden dabei gelesen und das Ergebnis berechnet.

Ereignis-Ausgänge

Name	Typ	Beschreibung
CNF	Event	Wird nach erfolgreicher Berechnung gesendet, signalisiert, dass am Ausgang OUT gültige Daten anliegen.

Daten-Eingänge

Name	Typ	Beschreibung
IN_00	WORD	Erstes Wort (Byte 0 und 1)
IN_01	WORD	Zweites Wort (Byte 2 und 3)
IN_02	WORD	Drittes Wort (Byte 4 und 5)
IN_03	WORD	Viertes Wort (Byte 6 und 7)

Daten-Ausgänge

Name	Typ	Beschreibung
OUT	ARRAY[0..7] OF BYTE	Ergebnis-Array, enthält die 8 Bytes in Little-Endian-Reihenfolge.

Adapter

Keine Adapter definiert.

Funktionsweise

Der FB führt bei einer steigenden Flanke am Ereigniseingang REQ die folgende Abbildung durch (beschrieben in der Strukturierten Text-Implementierung):

Index OUT	Quelle	Bedeutung
OUT[0]	IN_00.%B0	Niederwertiges Byte von Wort 0
OUT[1]	IN_00.%B1	Höherwertiges Byte von Wort 0
OUT[2]	IN_01.%B0	Niederwertiges Byte von Wort 1
OUT[3]	IN_01.%B1	Höherwertiges Byte von Wort 1
OUT[4]	IN_02.%B0	Niederwertiges Byte von Wort 2
OUT[5]	IN_02.%B1	Höherwertiges Byte von Wort 2
`OUT[6]	IN_03.%B0	Niederwertiges Byte von Wort 3
OUT[7]	IN_03.%B1	Höherwertiges Byte von Wort 3

Nach Abschluss der Zuweisungen wird der Ausgangsimpuls CNF erzeugt.

Technische Besonderheiten

- **Little-Endian-Format**: Die Bytes jedes Wortes werden in aufsteigender Speicheradresse mit niederwertigem Byte zuerst abgelegt. Dies entspricht der Intel-Byte-Reihenfolge und ist für viele PC-basierte Steuerungen und Kommunikationsprotokolle (z. B. Modbus, CANopen, serielle Schnittstellen) üblich.
- **Keine Umkehrung der Gesamtreihenfolge**: Die Wörter selbst werden in der Reihenfolge IN_00 bis IN_03 in das Array eingefügt; nur innerhalb jedes Wortes wird die Byte-Reihenfolge gedreht. Möchte man stattdessen auch die Wortreihenfolge umkehren (z. B. Big-Endian auf Wortebene), ist eine andere Implementierung notwendig.
- **Sofortige Ausführung**: Der Baustein ist rein kombinatorisch – nach dem Ereignis REQ liegt das Ergebnis am nächsten Ausführungszyklus an. Es wird kein interner Zustand gespeichert.

Zustandsübersicht

Der Funktionsblock besitzt keinen expliziten Zustandsautomaten. Das Verhalten ist rein ereignisgesteuert:

1. **Idle**: Warten auf `REQ`.
2. **Processing**: Bei `REQ` werden die Eingangswerte gelesen und die Ausgangsberechnung durchgeführt.
3. **Complete**: `CNF` wird gesendet; der FB kehrt in den Idle-Zustand zurück.

Die Implementierung ist deterministisch und benötigt nur einen Ausführungsschritt.

Anwendungsszenarien

- **Kommunikation mit Peripherie**: Ein Mikrocontroller oder IPC sendet 16-Bit-Sensordaten (z. B. Analog/Digital-Wandler, Encoder) über einen seriellen Bus, der Byte-orientiert ist. Der Baustein packt die Wörter in ein Byte-Array, das direkt versendet werden kann.
- **Datenlogging**: Zusammenführen mehrerer 16-Bit-Messwerte in einen Puffer für die Speicherung oder Übertragung (z. B. CSV, Binärformat).
- **Protokoll-Konvertierung**: Anpassung von Daten an die Byte-Anordnung, die von einem bestimmten Protokoll (z. B. OPC UA, MQTT-Binär-Payload) gefordert wird.

Vergleich mit ähnlichen Bausteinen

- **WORDS_TO_ARR08B** erwartet genau vier Wörter und erzeugt ein Array der Länge 8. Für andere Wort-/Byte-Anzahlen existieren verwandte Bausteine wie `WORDS_TO_ARR04B`, `WORDS_TO_ARR16B` oder jeweils für Big-Endian.
- Im Gegensatz zu einem universellen `MERGE`-Block, der beliebige Datentypen zusammenfügt, ist dieser FB speziell auf die 2-Byte-Pro-Wort-Konvertierung optimiert und garantiert die korrekte Byte-Reihenfolge.
- Gegenüber einer einfachen Variablenzuweisung mittels `MOVE` oder einer reinen `WORD_TO_BYTE`-Funktion vermeidet der Block manuelle Index-Berechnungen und erhöht die Lesbarkeit.

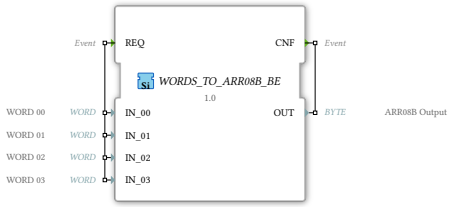
Fazit

Der Funktionsblock **WORDS_TO_ARR08B** ist ein kompakter, spezialisierter Baustein für die Umwandlung von vier 16-Bit-Wörtern in ein Little-Endian-Byte-Array. Durch die klare ereignisgesteuerte Schnittstelle und die einfache Implementierung eignet er sich ideal für den Einsatz in Steuerungsanwendungen, bei denen Daten auf Byte-Ebene für die Kommunikation oder Speicherung aufbereitet werden müssen. Er gehört zur Paketfamilie `logiBUS::utils::conversion::arr::reversing` und erweitert die 4diac-Toolbox um eine nützliche Konvertierungsmöglichkeit.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 MCU vs. MPU Vergleichsguide auf ms-muc-docs.de

WORDS_TO_ARR08B_BE



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **WORDS_TO_ARR08B_BE** fasst vier 16-Bit-Wörter (WORD) zu einem 8-Byte-Array zusammen. Die Umwandlung erfolgt im **Big-Endian-Format** (Network Byte Order), bei dem das höherwertige Byte (High-Byte) zuerst im Ausgabearray abgelegt wird. Dies entspricht der üblichen Darstellung in Netzwerkprotokollen wie Modbus TCP, Siemens S7-Kommunikation oder Ethernet-Headern.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung
REQ	Startet die Konvertierung aller vier Eingangswörter.

Ereignis-Ausgänge

Ereignis	Beschreibung
CNF	Wird gesendet, sobald das Ausgabearray vollständig berechnet wurde.

Daten-Eingänge

Name	Typ	Kommentar
IN_00	WORD	Word 00 (erstes Wort)
IN_01	WORD	Word 01 (zweites Wort)
IN_02	WORD	Word 02 (drittes Wort)
IN_03	WORD	Word 03 (viertes Wort)

Daten-Ausgänge

Name	Typ	Kommentar
OUT	ARRAY[0..7] OF BYTE	8-Byte-Array im Big-Endian-Format

Adapter

Keine Adapter vorhanden.

Funktionsweise

Nach dem Empfangen der Ereignisses REQ werden die Werte der vier Eingänge IN_00 bis IN_03 ausgelesen. Für jedes Wort wird zuerst das **High-Byte** (BYTE 1) und dann das **Low-Byte** (BYTE 0) in das Ausgabearray geschrieben. Die Reihenfolge im Array ist:

- OUT[0] = High-Byte von IN_00
- OUT[1] = Low-Byte von IN_00
- OUT[2] = High-Byte von IN_01
- OUT[3] = Low-Byte von IN_01
- OUT[4] = High-Byte von IN_02
- OUT[5] = Low-Byte von IN_02
- OUT[6] = High-Byte von IN_03
- OUT[7] = Low-Byte von IN_03

Anschließend wird das Ereignis CNF ausgelöst.

Technische Besonderheiten

- **Big-Endian-Format** (Network Byte Order) – die Bytes werden in der Reihenfolge höherwertiges Byte zuerst abgelegt. Dies ist für viele industrielle Kommunikationsprotokolle und Netzwerkstack erforderlich.
- Der Baustein arbeitet **ereignisgesteuert**: Die Konvertierung wird nur bei einem REQ-Ereignis ausgeführt, sodass keine zyklische Berechnung erfolgt.
- Die interne Implementierung erfolgt in **Strukturiertem Text (ST)** mit direkter Bit-Elementzugriff (%B1 und %B0), was eine effiziente und deterministische Umwandlung gewährleistet.

Zustandsübersicht

Der Funktionsblock besitzt keinen expliziten internen Zustand (zustandslos). Er führt bei jedem `REQ` sofort die Berechnung durch und signalisiert den Abschluss mit `CNF`. Während der Berechnung sind keine Blockierungen zu erwarten.

Anwendungsszenarien

- **Netzwerkkommunikation**: Aufbereitung von Daten für Protokolle, die Big-Endian erwarten, z. B. Modbus TCP, Siemens S7 über Industrial Ethernet oder Raw-Ethernet-Frames.
- **Datenvorverarbeitung**: Konvertierung von Sensorwerten oder Steuerbefehlen, die als WORD-Register vorliegen, in ein Byte-orientiertes Format.
- **Gateway-Funktionen**: Umsetzung zwischen Little-Endian-Steuerungen und Big-Endian-Netzwerken.

Vergleich mit ähnlichen Bausteinen

Baustein	Besonderheit
WORDS_TO_ARR08B_BE	Big-Endian, 4 WORDS ⟶ 8 Bytes
WORDS_TO_ARR08B_LE	Little-Endian (Low-Byte zuerst)
BYTES_TO_WORDS_BE / LE	Rückwärtskonvertierung
WORDS_TO_ARR16B_BE	Big-Endian, 8 WORDS ⟶ 16 Bytes
WORDS_TO_ARR08N	Ggf. unspezifische Endianness

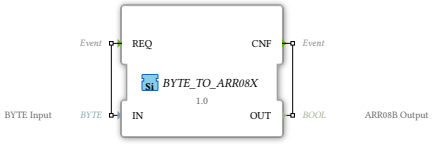
Der vorliegende Baustein ist speziell für Big-Endian-Anwendungen optimiert und kompakt gehalten.

Fazit

`WORDS_TO_ARR08B_BE` ist ein präziser, ereignisgesteuerter Funktionsblock zur Big-Endian-Konvertierung vierer WORDS in ein 8-Byte-Array. Er eignet sich hervorragend für alle Anwendungen, die eine byte-weise Darstellung im Network Byte Order erfordern, und vereinfacht die Schnittstelle zwischen Steuerungslogik und Kommunikationsprotokollen.

splitting

BYTE_TO_ARR08X



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `BYTE_TO_ARR08X` dient der Konvertierung eines einzelnen Byte-Wertes in ein Array von acht booleschen Werten (BOOL). Jedes Bit des Eingangsbytes wird einem separaten Element im Ausgangsarray zugeordnet. Dieser Baustein ist besonders nützlich, um kompakte digitale Informationen, die in einem Byte gespeichert sind, für die individuelle Verarbeitung oder Visualisierung aufzuschlüsseln.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ` (Request): Dieses Ereignis löst die Konvertierungsfunktion aus. Bei seinem Eintreffen wird der aktuelle Wert am Dateneingang `IN` verarbeitet.

Ereignis-Ausgänge

- `CNF` (Confirm): Dieses Ereignis signalisiert den Abschluss der Konvertierung. Es wird zusammen mit den berechneten Daten am Ausgang `OUT` ausgegeben.

Daten-Eingänge

- `IN` (BYTE): Der Eingang für den Byte-Wert, der in boolesche Bits zerlegt werden soll.

Daten-Ausgänge

- `OUT` (ARRAY[0..7] OF BOOL): Das Ausgangsarray mit acht BOOL-Elementen. Das Element `OUT[0]` entspricht dem niederwertigsten Bit (LSB, Bit 0) des Eingangsbytes, `OUT[7]` entspricht dem höchstwertigen Bit (MSB, Bit 7).

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei Auslösung durch das `REQ`-Ereignis liest der Baustein den Wert am Eingang `IN`. Anschließend wird jedes einzelne Bit (0 bis 7) des BYTE-Datentyps extrahiert und dem entsprechenden Index im BOOL-Array `OUT` zugewiesen. Die Zuordnung erfolgt direkt:

- `OUT[0]` erhält den Wert von `IN.0` (Bit 0)
- `OUT[1]` erhält den Wert von `IN.1` (Bit 1)
- ...
- `OUT[7]` erhält den Wert von `IN.7` (Bit 7)

Nach Abschluss dieser Zuweisungen wird das `CNF`-Ereignis zusammen mit dem gefüllten Array `OUT` ausgegeben.

Technische Besonderheiten

- **Direkte Bit-Zugriffe:** Die Implementierung verwendet die in der IEC 61131-3 ST (Structured Text) Syntax übliche direkte Bit-Adressierung (z.B. `IN.0`), was eine effiziente und lesbare Operation ermöglicht.
- **Feste Array-Größe:** Das Ausgangsarray ist mit der festen Größe `[0..7]` definiert, was exakt der Bit-Länge eines Bytes (8 Bit) entspricht.
- **Echtzeitfähig:** Der Block führt eine deterministische, zustandslose Operation durch und ist damit für Echtzeitanwendungen geeignet.

Zustandsübersicht

Der Funktionsblock ist zustandslos (stateless). Sein Verhalten ist bei jedem `REQ`-Ereignis identisch und hängt ausschließlich vom aktuellen Wert am Dateneingang `IN` ab. Es gibt keine internen Variablen, die zwischen den Aufrufen gespeichert werden.

Anwendungsszenarien

- **Entpacken von Steuerbytes:** Verarbeitung eines Status- oder Steuerbytes, das von einem Feldgerät oder einer übergeordneten Steuerung empfangen wurde, um einzelne Statusbits (z.B. „Motor ein“, „Ventil offen“, „Störung aktiv“) separat auszuwerten.
- **Visualisierung:** Aufbereitung von Daten für Visualisierungssysteme (HMIs), die einzelne Schaltzustände oft als separate boolesche Lampen oder Indikatoren darstellen.
- **Schnittstelle zu Bit-orientierter Logik:** Bereitstellung einzelner Bits für nachfolgende Funktionsblöcke, die mit boolescher Logik arbeiten (AND, OR, NOT, FlipFlops), anstatt mit ganzen Bytes.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`BYTE_TO_BOOL` (oder ähnlich benannte Blöcke):** Viele Bibliotheken bieten Blöcke an, die ein Byte in acht separate BOOL-Ausgänge konvertieren. `BYTE_TO_ARR08X` unterscheidet sich dadurch, dass es die Bits in einem strukturierten **Array** ausgibt, was die Verkabelung in der 4diac IDE vereinfacht, da nur ein Datenausgang benötigt wird. Die Handhabung in ST-Code ist ebenfalls kompakter.
- **Bitweises Auslesen mittels `AND` & Vergleichen:** Die gleiche Funktionalität könnte durch eine Reihe von Vergleichen (`(IN AND 16#01) = 1` etc.) erreicht werden. Der `BYTE_TO_ARR08X`-Block kapselt diese Logik in einem wiederverwendbaren, leicht verständlichen Baustein.
- **Gegenstück `ARR08X_TO_BYTE`:** Ein komplementärer Baustein, der ein Array von acht BOOLs zu einem Byte zusammenfasst, würde die umgekehrte Operation durchführen.

Fazit

Der `BYTE_TO_ARR08X` Funktionsblock ist ein einfacher, spezialisierter und äußerst praktischer Konvertierungsbaustein. Er löst ein häufiges Problem in der Automatisierungstechnik auf elegante Weise, indem er die Bit-Extraktion aus einem Byte abstrahiert und die Ergebnisse in einem gut handhabbaren Array bereitstellt. Seine deterministische und zustandslose Natur macht ihn zu einem zuverlässigen und performanten Bestandteil für Anwendungen, in denen digitale Bit-Informationen kompakt übertragen und dann dezentral verarbeitet werden müssen.

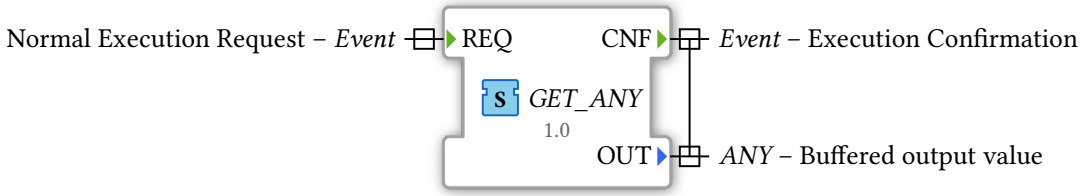
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Numerischer Index (conversion)

Hier finden Sie alle Übungen des Arbeitsbereichs `conversion` in aufsteigender Reihenfolge.

GET_ANY



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **GET_ANY** liest einen beliebigen Wert (Typ `ANY`) aus einer InOut-Variable und stellt diesen als gepufferten Ausgangswert bereit. Er dient dazu, den aktuellen Wert einer variablen Quelle zu einem bestimmten Zeitpunkt zu erfassen und für nachfolgende Verarbeitungsschritte zwischenzuspeichern.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Mit Variablen	Beschreibung
REQ	Event	IN	Normaler Ausführungsauftrag; liest den aktuellen Wert von IN.

Ereignis-Ausgänge

Name	Typ	Mit Variablen	Beschreibung
CNF	Event	IN, OUT	Bestätigung nach erfolgreicher Ausführung; die Werte von IN und OUT sind dann gültig.

Daten-Eingänge

Der FB besitzt keine separaten Dateneingänge. Die Quelldaten werden über die InOut-Variable bereitgestellt.

Daten-Ausgänge

Name	Typ	Beschreibung
OUT	ANY	Gepufferter Ausgangswert. Enthält den zum Zeitpunkt der letzten REQ-Ausführung gelesenen Wert von IN.

Adapter

Keine.

Funktionsweise

1. Ein Ereignis an **REQ** löst den internen Algorithmus `REQ` aus.
2. Der Algorithmus führt die Zuweisung `OUT := IN` aus.
   - **IN** ist eine InOut-Variable, die sowohl gelesen als auch beschrieben werden kann – hier wird sie nur als Quelle verwendet.
   - **OUT** erhält den aktuellen Wert von **IN** und speichert ihn zwischen.
3. Nach Abschluss der Zuweisung wird das Ereignis **CNF** gesendet. Zu diesem Zeitpunkt sind sowohl **IN** (unverändert) als auch **OUT** gültig.

Technische Besonderheiten

- Der FB nutzt eine **InOut-Variable** (`IN`) als Quelle. Dies ermöglicht den direkten Zugriff auf einen anderen Baustein-Ausgang oder eine globale Variable, ohne dass eine separate Verbindung über einen Dateneingang erforderlich ist.
- Der **Typ `ANY`** macht den Baustein generisch; er kann mit beliebigen Datentypen (z. B. INT, REAL, BOOL, STRING) verwendet werden.
- Es existiert nur ein einziger Zustand, daher ist die Ausführung deterministisch und erfolgt in einem Schritt.

Zustandsübersicht

Der FB ist als **SimpleFB** implementiert mit genau einem Zustand:

Zustand	Beschreibung
REQ	Führt den Algorithmus `REQ` aus und sendet CNF aus. Nach der Aktion kehrt der FB in diesen Zustand zurück (default).

Es sind keine weiteren Zustände (z. B. Warten, Fehler) definiert.

Anwendungsszenarien

- **Wertabtastung**: Zu einem bestimmten Zeitpunkt (Ereignis) soll ein aktueller Wert eines Sensors oder einer Berechnung festgehalten werden, z. B. für eine spätere Protokollierung oder Vergleiche.
- **Signalentkopplung**: Wenn ein Ausgang mehrfach gelesen werden muss, ohne dass sich der Wert während der Bearbeitung ändert, puffert GET_ANY den Wert.
- **Typunabhängige Schnittstelle**: Als generischer Zwischenspeicher in Bibliotheken, die mit verschiedenen Datentypen arbeiten.

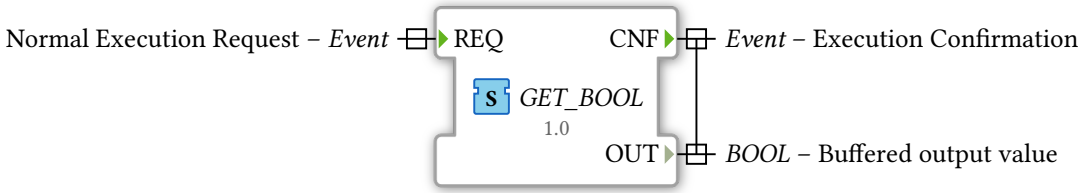
Vergleich mit ähnlichen Bausteinen

- **`GET`-Bausteine** (z. B. `GET_INT`): Diese sind typspezifisch und erfordern explizite Datentypen. `GET_ANY` bietet dagegen **Typflexibilität** auf Kosten der Typsicherheit zur Entwurfszeit.
- **`MOVE`-FB**: Ein `MOVE`-Baustein kopiert typischerweise von einem Eingang auf einen Ausgang, verwendet aber keine InOut-Variable. `GET_ANY` hingegen liest direkt aus einer bestehenden Variablen, ohne dass eine separate Eingangsverbindung nötig ist.

Fazit

`GET_ANY` ist ein einfacher, aber mächtiger Baustein zum **Zwischenspeichern eines beliebigen Werts** aus einer InOut-Quelle. Seine generische Typauslegung und die direkte Anbindung an vorhandene Variablen reduzieren den Verkabelungsaufwand und erhöhen die Flexibilität in der Steuerungslogik. Er eignet sich besonders für Anwendungen, bei denen ein Wert genau zu einem Ereigniszeitpunkt eingefroren werden muss.

GET_BOOL



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **GET_BOOL** dient dem Auslesen eines BOOL-Wertes aus einer als `InOut` deklarierten Variablen und stellt diesen als gepufferten Ausgangswert bereit. Diese einfache Kapselung ermöglicht das Weiterleiten eines booleschen Signals aus einem beliebigen Kontext, ohne dass im aufrufenden Baustein direkter Schreib-/Lesezugriff auf die Quelle notwendig ist.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung	Mit Variablen
REQ	Normale Ausführungsanforderung	IN

Ereignis-Ausgänge

Ereignis	Beschreibung	Mit Variablen
CNF	Bestätigung der Ausführung	OUT, IN

Daten-Eingänge

Es existieren keine deklarierten Dateneingänge (*InputVars*). Der Zugriff auf die Quellvariable erfolgt ausschließlich über einen InOut-Parameter.

Daten-Ausgänge

Variable	Typ	Initialwert	Beschreibung
OUT	BOOL	FALSE	Gepufferter Ausgangswert

Adapter

Keine Adapter vorhanden.

Funktionsweise

1. Am **Ereignis-Eingang REQ** wird die Abarbeitung gestartet.
2. Der interne Algorithmus **REQ** führt die einfache Zuweisung `OUT := IN` aus – der aktuelle Wert der InOut-Variablen `IN` wird in den Ausgang `OUT` kopiert.
3. Nach dem erfolgreichen Kopiervorgang wird das **Ereignis CNF** ausgelöst. Es gibt sowohl den neuen Wert `OUT` als auch den (unveränderten) Wert von `IN` an die nachfolgende Logik weiter.

Der Ausgang `OUT` bleibt solange auf dem zuletzt gelesenen Wert, bis ein erneuter `REQ`-Impuls den Wert aktualisiert.

Technische Besonderheiten

- **InOut-Parameter** `IN`: Dieser Parameter erlaubt den bidirektionalen Zugriff auf die angeschlossene Variable. Der Baustein liest den Wert, verändert ihn jedoch nicht. Dadurch kann direkt auf eine Variable aus der übergeordneten Applikation zugegriffen werden, ohne eine separate Eingangsvariable definieren zu müssen.
- **Pufferung**: Der Wert wird lokal im Ausgang `OUT` zwischengespeichert. Änderungen an der Quellvariablen zwischen zwei `REQ`-Aufrufen werden erst mit dem nächsten `REQ` übernommen.
- **Initialwerte**: Sowohl `IN` als auch `OUT` sind mit `FALSE` vorbelegt, sodass der Baustein auch ohne vorherige Initialisierung ein definiertes Verhalten zeigt.

Zustandsübersicht

Der Baustein besitzt einen einzigen Zustand:

Zustand	Aktion	Ausgabeereignis
REQ	`OUT := IN`	CNF

Nach Ausführung des Algorithmus wird sofort der Ausgangszustand wieder verlassen (kein Haltezustand). Der Baustein ist somit ereignisgesteuert und wartet nach Abarbeitung auf die nächste Anforderung.

Anwendungsszenarien

- **Signal-Erfassung**: Ein boolescher Sensorwert (z. B. Schalter oder Grenzwertmelder) soll in der Applikation genutzt und gleichzeitig an einen übergeordneten Anzeigebaustein weitergegeben werden.
- **Entkopplung**: Eine Variable aus einem anderen FB soll gelesen werden, ohne dass der Quellbaustein eine direkte Kopplung eingeht. `GET_BOOL` übernimmt die Leseverantwortung.
- **Puffer und Synchronisation**: In getakteten Systemen kann durch einen `REQ`-Impuls der aktuelle Wert abgetastet und für die Dauer eines Zyklus gehalten werden.

Vergleich mit ähnlichen Bausteinen

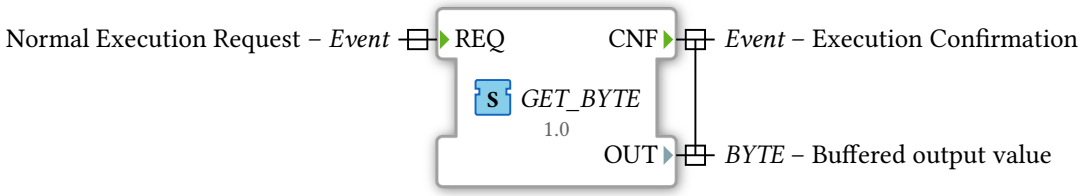
Baustein	Zweck	Datentyp
GET_BOOL	Lesen eines BOOL-Wertes über InOut	BOOL
GET_INT	Lesen eines INT-Wertes über InOut	INT
GET_REAL	Lesen eines REAL-Wertes über InOut	REAL
SET_BOOL	Schreiben eines BOOL-Wertes über InOut (schreibend)	BOOL

Gemeinsam ist allen `GET_*`-Bausteinen die Verwendung eines InOut-Parameters und die ereignisgesteuerte Werteübernahme. Der Unterschied liegt allein im verwendeten Datentyp. Im Gegensatz zu `SET_BOOL` verändert `GET_BOOL` die Quellvariable nicht.

Fazit

Der **GET_BOOL**-Baustein ist eine minimalistische, aber nützliche Komponente zum Auslesen und Puffern eines einzelnes booleschen Werts. Seine Stärke liegt in der sauberen Entkopplung der Signalquelle vom restlichen Applikationscode sowie in der einfachen Handhabung durch einen einzigen Ereignisimpuls. Er eignet sich besonders für modulare und wiederverwendbare Automatisierungslösungen, die einen klar definierten Datenfluss erfordern.

GET_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **GET_BYTE** dient dem Auslesen einer `BYTE`-Variablen, die über einen **InOut**-Parameter bereitgestellt wird. Der gelesene Wert wird am Ausgang `OUT` gepuffert und steht nach der Ausführung zur Verfügung. Der Baustein wird typischerweise eingesetzt, um einen aktuellen `BYTE`-Wert aus einer externen Quelle (z. B. einem gemeinsamen Datenbereich) abzugreifen, ohne die Quelle selbst zu verändern.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar	Mit Variablen
REQ	Event	Normaler Ausführungsanstoß	IN

Ereignis-Ausgänge

Name	Typ	Kommentar	Mit Variablen
CNF	Event	Bestätigung der Ausführung	OUT, IN

Daten-Eingänge

Der Baustein besitzt keine klassischen Dateneingänge, sondern nutzt einen **InOut-Parameter**, der gleichzeitig als Ein- und Ausgang genutzt werden kann:

Name	Typ	Initialwert	Kommentar
IN	BYTE	0	Quellvariable (lesend über InOut)

Daten-Ausgänge

Name	Typ	Initialwert	Kommentar
OUT	BYTE	0	Gepufferter Ausgabewert

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Baustein wird über ein Ereignis am Eingang `REQ` aktiviert. Daraufhin wird der aktuelle Wert der am **InOut**-Parameter `IN` anliegenden Variable gelesen und in den Ausgang `OUT` kopiert. Anschließend wird das Ereignis `CNF` ausgelöst, das die erfolgreiche Ausführung quittiert. Der Algorithmus ist denkbar einfach:

```
OUT := IN;
```

Da `IN` als InOut-Parameter deklariert ist, muss die aufrufende Applikation eine BYTE-Variable bereitstellen, die sowohl gelesen als auch (bei anderen Bausteinen) geschrieben werden kann. `GET_BYTE` führt jedoch nur eine Leseoperation aus.

Technische Besonderheiten

- **InOut-Parameter**: Der Baustein greift direkt auf eine externe Variable zu, ohne sie über einen separaten Daten-Eingang zu verbinden. Dies ermöglicht eine effiziente, direkte Datenbindung und spart Ressourcen in der Vernetzung.
- **Gepufferter Ausgang**: Der gelesene Wert wird in `OUT` zwischengespeichert und bleibt bis zur nächsten Ausführung erhalten – auch wenn sich die Quellvariable zwischenzeitlich ändert.
- **Keine Zustandsmaschine**: Der Baustein besteht aus einem einzigen Zustand, was die Implementierung schlank und deterministisch macht.

Zustandsübersicht

Der Baustein verfügt über genau einen internen Zustand:

Zustand	Beschreibung	Ausgangsereignis
REQ	Kein Warten; sofortiger Ablauf der Leseoperation	CNF

Es gibt keine Wartezustände oder Verzweigungen.

Anwendungsszenarien

- **Auslesen von Konfigurationswerten**: Ein `BYTE`-Wert aus einem zentralen Datenblock (z. B. einer globalen Konfiguration) wird in einen lokalen Puffer übernommen.
- **Synchronisation zwischen Bausteinen**: Ein anderer Baustein schreibt einen `BYTE` in eine gemeinsame InOut-Variable, und `GET_BYTE` liest diesen Wert bei Bedarf aus.
- **Sichere Datenübernahme**: Durch den gepufferten Ausgang wird verhindert, dass nachfolgende Logik auf einen sich ändernden Quellwert reagiert – sie arbeitet stets mit dem zum Zeitpunkt der Anfrage gültigen Wert.

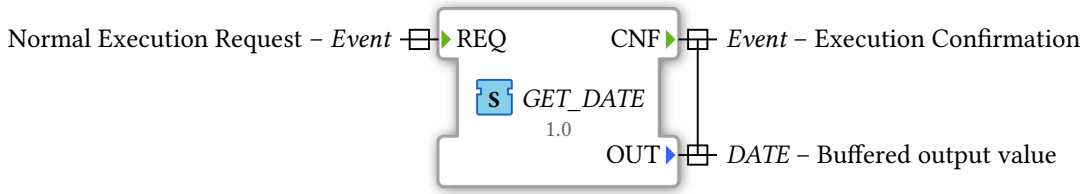
Vergleich mit ähnlichen Bausteinen

Baustein	Beschreibung	Unterschied zu GET_BYTE
GET_VALUE	Liest einen Wert aus einem InOut-Parameter aus (typ-unspezifisch)	GET_BYTE ist auf den Datentyp `BYTE` spezialisiert und daher typsicherer.
MOVE	Kopiert einen Wert von einem Eingang zu einem Ausgang	MOVE benötigt separate Eingänge und gibt keinen Puffer – GET_BYTE nutzt InOut.
READ_BYTE	Liest ein Byte über eine Hardware-Adresse	GET_BYTE arbeitet auf der Variablenebene, nicht auf Hardware-Adressen.

Fazit

GET_BYTE ist ein einfacher, aber nützlicher Baustein, um einen `BYTE`-Wert aus einer externen Quelle über einen InOut-Parameter auszulesen und gepuffert bereitzustellen. Dank des schlanken Designs (ein Zustand, minimale Schnittstelle) eignet er sich besonders für ressourcenkritische Anwendungen und für die Integration in bestehende Kommunikations- oder Speicherkonzepte, bei denen der direkte Zugriff auf eine gemeinsam genutzte Variable erwünscht ist.

GET_DATE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **GET_DATE** liest einen DATE-Wert aus einer InOut-Variable aus und stellt diesen als gepufferten Ausgang bereit. Bei Anforderung durch ein Ereignis wird der aktuelle Wert der Quelle in einen internen Puffer kopiert und anschließend am Ausgang sowie an der InOut-Variable selbst ausgegeben.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar	Mit-Variablen
REQ	Event	Normale Ausführungsanforderung	IN

Ereignis-Ausgänge

Name	Typ	Kommentar	Mit-Variablen
CNF	Event	Ausführungsbestätigung	OUT, IN

Daten-Eingänge

Name	Typ	Kommentar
IN	DATE	Quellvariable (InOut)

Daten-Ausgänge

Name	Typ	Kommentar
OUT	DATE	Gepufferter Ausgabewert

IN	DATE	Quellvariable (InOut) – wird bei CNF ebenfalls ausgegeben

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Baustein besitzt einen einfachen Ausführungszustand **REQ**:

1. Ein eingehendes Ereignis am Eingang **REQ** löst die Verarbeitung aus.
2. Der interne Algorithmus kopiert den aktuellen Wert der InOut-Variable **IN** in das Ausgangssignal **OUT**.
3. Anschließend wird das Ereignis **CNF** gesendet. Dabei werden sowohl der Wert von **OUT** als auch der (unveränderte) Wert von **IN** an den Ausgang übergeben.

Die Funktionalität entspricht einer einfachen Wertübernahme ohne weitere Verarbeitung oder Zustandsspeicherung über mehrere Aufrufe hinweg.

Technische Besonderheiten

- Die Verwendung einer **InOut-Variable** ermöglicht den direkten Zugriff auf eine externe Variable, ohne einen separaten Dateneingang zu deklarieren.
- Die Pufferung erfolgt nur während der Ausführung des Algorithmus; bei jedem neuen **REQ** wird der aktuelle Wert von **IN** erneut gelesen.
- Der Baustein ist als **SimpleFB** realisiert und besitzt keine internen Zustandsübergänge – die Verarbeitung ist rein kombinatorisch mit einem einzigen Schritt.

Zustandsübersicht

Der Funktionsblock besteht aus genau einem Zustand:

- **REQ**: Beim Eintreten wird der Algorithmus ausgeführt, der **OUT** mit **IN** beschreibt, und das Ereignis **CNF** wird gesendet.

Es gibt keine weiteren Zustände oder bedingten Übergänge.

Anwendungsszenarien

- **Lesen eines globalen Datums**: Wenn eine Systemzeit oder ein Datum aus einer zentralen Variable gelesen und lokal verwendet werden soll.
- **Pufferung für nachfolgende Verarbeitung**: Um einen DATE-Wert zu einem bestimmten Zeitpunkt festzuhalten und an andere Bausteine weiterzugeben.
- **Kaskadierung von InOut-Zugriffen**: In Kombination mit weiteren Bausteinen, die auf die gleiche InOut-Variable zugreifen, um konsistente Daten zu gewährleisten.

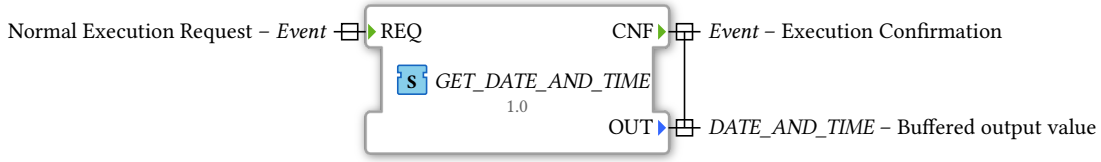
Vergleich mit ähnlichen Bausteinen

- **GET_BOOL**, **GET_INT**, **GET_STRING** etc.: Diese Bausteine arbeiten nach dem gleichen Prinzip, jedoch mit anderen Datentypen. Sie alle nutzen eine InOut-Variable als Quelle und einen gepufferten Ausgang.
- **SET_DATE**: Der Gegenpart – er schreibt einen Wert in eine InOut-Variable anstelle des Lesens.
- Alle diese Bausteine sind einfach aufgebaut und eignen sich für standardisierte Zugriffe auf gemeinsam genutzte Daten.

Fazit

GET_DATE ist ein minimalistischer, aber nützlicher Funktionsblock für den Lesezugriff auf einen DATE-Wert über eine InOut-Variable. Seine einfache Zustandslogik und die direkte Wertkopie machen ihn zuverlässig und leicht verständlich. Er eignet sich besonders für modulare Automatisierungslösungen, bei denen Daten zwischen Bausteinen über eine gemeinsame Variable ausgetauscht werden sollen.

GET_DATE_AND_TIME



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **GET_DATE_AND_TIME** dient dazu, einen Wert vom Typ `DATE_AND_TIME` aus einer als InOut deklarierten Variable auszulesen und als gepufferten (gebufferten) Ausgang bereitzustellen. Bei jeder Anforderung wird der aktuelle Wert der Quelle (InOut-Variable) auf den Ausgang kopiert.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Mit	Beschreibung
REQ	IN	Normale Ausführungsanforderung; löst das Auslesen der Quelldaten aus.

Ereignis-Ausgänge

Ereignis	Mit	Beschreibung
CNF	OUT, IN	Bestätigung der Ausführung; signalisiert, dass der Ausgang aktualisiert wurde.

Daten-Eingänge

Name	Typ	Beschreibung
IN	DATE_AND_TIME	InOut-Variable als Quelle der Daten. Sie wird gelesen, aber nicht verändert.

Daten-Ausgänge

Name	Typ	Beschreibung
OUT	DATE_AND_TIME	Gepufferter Ausgangswert, der bei jedem REQ aus der InOut-Quelle übernommen wird.

Adapter

Keine.

Funktionsweise

Der Baustein besitzt einen einfachen internen Zustand **REQ**. Beim Eintreten dieses Zustands (ausgelöst durch das Ereignis `REQ`) wird folgender Algorithmus ausgeführt:

```
OUT := IN;
```

Anschließend wird das Ereignis `CNF` gesendet. Der Wert der InOut-Variable `IN` wird dabei nicht verändert; er wird lediglich ausgelesen und in den Ausgang `OUT` kopiert. Somit arbeitet der Baustein als **Puffer** für einen beliebigen `DATE_AND_TIME`-Wert.

Technische Besonderheiten

- Die Verwendung einer **InOut-Variable** (`IN`) erlaubt den direkten Zugriff auf eine externe Variable, ohne dass diese als separater Ein- oder Ausgang deklariert werden muss. Der Baustein greift lesend darauf zu.
- Der Ausgang `OUT` speichert den zum Zeitpunkt des `REQ` vorliegenden Wert. Ändert sich die Quelle zwischen zwei `REQ`-Aufrufen, wird der neue Wert erst mit dem nächsten `REQ` übernommen (gepufferte Bereitstellung).

- Es handelt sich um einen **einfachen Funktionsblock** (SimpleFB) ohne komplexe Zustandsautomatik – nur ein einziger Ausführungszustand.

Zustandsübersicht

Zustand	Aktion	Ausgabeereignis
REQ	`OUT := IN;`	CNF

Es gibt keinen Initialzustand oder sonstige Zustände; der Baustein wartet auf das Ereignis `REQ` und führt sofort den Algorithmus aus.

Anwendungsszenarien

- **Zeitstempel-Erfassung**: Abrufen des aktuellen Zeitpunkts aus einer Systemuhr, die als InOut-Variable bereitsteht, zu einem definierten Steuerungszeitpunkt.
- **Pufferung von Datums-/Zeitwerten**: Zwischenspeichern eines momentanen Datums-/Zeitwerts für spätere Verarbeitungsschritte.
- **Synchronisation**: Übernahme eines gemeinsamen Zeitwerts aus einer zentralen Quelle in verschiedene Funktionsbausteine.

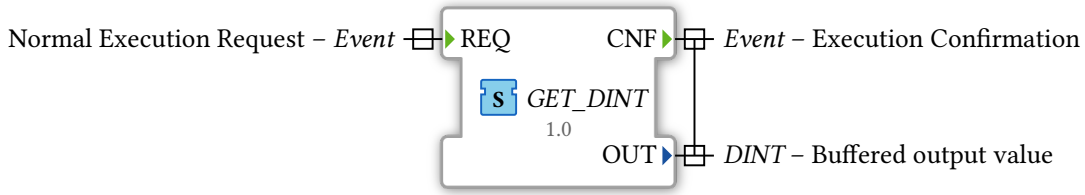
Vergleich mit ähnlichen Bausteinen

- **GET_DATE_AND_TIME** im Gegensatz zu Bausteinen wie `E_GET_TIME` (Echtzeit-Uhr-Abfrage) oder `TIME_GET` (aktuelle Systemzeit): Dieser Baustein arbeitet nicht mit einer fest eingebauten Uhr, sondern liest aus einer beliebigen InOut-Variable. Dadurch ist er flexibler einsetzbar, z.B. für den Zugriff auf applikationsspezifische Zeitwerte.
- Im Vergleich zu einem einfachen Zuweisungsbaustein (z.B. `MOVE` mit `DATE_AND_TIME`) bietet er eine ereignisgesteuerte, explizite Anforderung und explizite Bestätigung (`REQ`/`CNF`), was für zeitkritische Steuerungen vorteilhaft ist.

Fazit

GET_DATE_AND_TIME ist ein schlanker, klar strukturierter Funktionsblock, der das Auslesen und Puffern eines `DATE_AND_TIME`-Werts aus einer InOut-Variable ermöglicht. Seine einfache Handhabung und die explizite Ereignissteuerung machen ihn zu einem nützlichen Werkzeug für die zuverlässige Übernahme von Zeitdaten in Automatisierungsanwendungen.

GET_DINT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **GET_DINT** liest einen DINT-Wert aus einer InOut-Variable und stellt diesen als gepufferten Ausgangswert bereit. Die Variable wird dabei nicht verändert; der Baustein dient als reiner Lese- und Pufferbaustein.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** – Normaler Ausführungsauftrag (aktiviert das Auslesen der InOut-Variable).

Ereignis-Ausgänge

- **CNF** – Bestätigung der Ausführung (wird nach erfolgreicher Übernahme des Wertes gesendet).

Daten-Eingänge

Es sind keine klassischen Daten-Eingänge vorhanden. Die Quelle des Wertes wird über die nachfolgend beschriebene InOut-Variable bereitgestellt.

Daten-Ausgänge

- **OUT** (Typ: DINT, Initialwert: 0) – Gepufferter Ausgangswert, der den aktuellen Wert der InOut-Variable enthält.

Adapter

Keine.

Besonderheit: Der Baustein verwendet eine **InOut-Variable** mit der Bezeichnung **IN** (Typ: DINT, Initialwert: 0). Diese Variable dient als Quelle für den Lesevorgang. Sie ist weder ein reiner Daten-Eingang noch ein reiner Daten-Ausgang, sondern ermöglicht den direkten Zugriff auf eine im Aufruferkontext liegende Speicherstelle. Der Baustein greift lesend auf diese Variable zu.

Funktionsweise

Wird der Ereigniseingang **REQ** aktiviert, führt der Baustein folgende Schritte aus:

1. Der aktuelle Wert der InOut-Variable **IN** wird gelesen.
2. Dieser Wert wird in den Datenausgang **OUT** kopiert (gepuffert).
3. Anschließend wird der Ereignisausgang **CNF** gesendet, um die erfolgreiche Ausführung zu signalisieren.

Da es sich um eine einzelne Aktion handelt, ist der Baustein deterministisch und benötigt keine weiteren Zustandswechsel.

Technische Besonderheiten

- **Direkter Speicherzugriff:** Durch die Verwendung einer InOut-Variable kann der Baustein auf eine Variable zugreifen, die außerhalb seiner eigenen Instanz liegt. Dies ermöglicht das Teilen von Daten zwischen verschiedenen Bausteinen ohne zusätzliche Eingangsverbindungen.
- **Pufferung:** Der Ausgang **OUT** hält den gelesenen Wert bis zur nächsten Aktivierung von **REQ**. Somit kann der Wert mehrfach und ohne erneute Abfrage der Quelle verwendet werden.
- **Keine Seiteneffekte:** Der Baustein verändert die InOut-Variable **IN** nicht; er liest nur ihren Wert.

Zustandsübersicht

Der Baustein besitzt einen einzigen Zustand **REQ**. Bei Aktivierung dieses Zustands wird die Aktion des Algorithmus ausgeführt (Wert kopieren) und der Zustand verlassen, wobei **CNF** gesendet wird.

- **REQ** (Startzustand und einziger Zustand)
  - Aktion: `OUT := IN;`
  - Ausgangsereignis: **CNF**

Es gibt keine Verzweigungen, Schleifen oder Wartezeiten.

Anwendungsszenarien

- **Puffern eines DINT-Werts:** In einer Steuerungsapplikation soll ein Messwert (z. B. von einem Sensor) nur auf Anforderung gelesen und für die weitere Verarbeitung zwischengespeichert werden.
- **Lesen geteilter Variablen:** Wenn mehrere Bausteine Zugriff auf eine gemeinsame Variable benötigen, kann **GET_DINT** den aktuellen Wert bereitstellen, ohne dass die Variable als Daten-Eingang jedes Bausteins geführt werden muss.
- **Schnittstellenentkopplung:** Trennung des Lesezeitpunkts von der eigentlichen Verarbeitung durch eine ereignisgesteuerte Pufferung.

Vergleich mit ähnlichen Bausteinen

In der 4diac-IDE existieren analoge Bausteine für andere Datentypen (z. B. `GET_INT`, `GET_BOOL`). Der grundlegende Mechanismus ist identisch, lediglich der Datentyp unterscheidet sich.

Im Gegensatz zu einem **MOVE**-Baustein, der einen Wert von einem Eingang zu einem Ausgang kopiert, liest **GET_DINT** den Wert aus einer InOut-Variable und puffert ihn. Dies erlaubt das Lesen von Variablen, die nicht direkt als Daten-Eingang verbunden werden können.

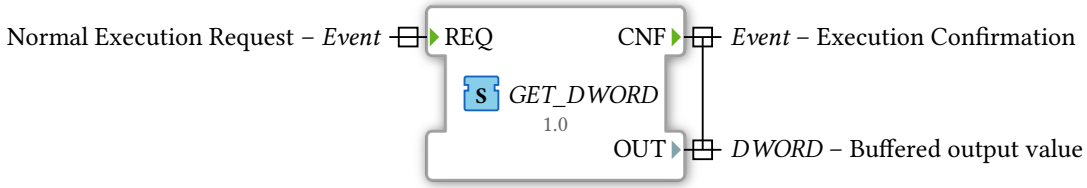
Fazit

GET_DINT ist ein einfacher, aber nützlicher Baustein zum ereignisgesteuerten Lesen und Puffern eines DINT-Werts aus einer InOut-Variable. Seine Stärke liegt in der Möglichkeit, auf gemeinsame Variablen zuzugreifen und den gelesenen Wert ohne weitere Seiteneffekte bereitzustellen. Die klare Struktur und die geringe Komplexität machen ihn zu einem zuverlässigen Werkzeug für viele Steuerungsaufgaben.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GET_DWORD



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `GET_DWORD` dient dazu, den aktuellen Wert einer als InOut-Parameter bereitgestellten Variable vom Typ `DWORD` auszulesen und diesen als gepufferten Ausgangswert bereitzustellen. Der Lesevorgang wird über ein Ereignis gesteuert.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Datentyp	Kommentar
`REQ`	`Event`	Anforderung zum Auslesen der InOut-Variable und Bereitstellen des Ausgangswerts

Ereignis-Ausgänge

Ereignis	Datentyp	Kommentar
`CNF`	`Event`	Bestätigung, dass der Lesevorgang abgeschlossen und die Ausgabe aktualisiert wurde

Daten-Eingänge

Name	Datentyp	Kommentar
`IN`	`DWORD`	Quellvariable (InOut). Der aktuelle Wert dieser Variable wird bei `REQ` gelesen. Initialwert: 0

Daten-Ausgänge

Name	Datentyp	Kommentar
`OUT`	`DWORD`	Gepufferter Ausgangswert. Enthält den bei der letzten Ausführung gelesenen Wert. Initialwert: 0

Adapter

Keine Adapter vorhanden.

Funktionsweise

Bei einem Ereignis am Eingang REQ wird der Algorithmus REQ ausgeführt. Dieser weist den aktuellen Wert der InOut-Variable IN der Ausgangsvariable OUT zu. Anschließend wird das Ereignis CNF gesendet.

Der FB puffert den gelesenen Wert, d.h. OUT behält seinen Wert, bis ein erneutes REQ ihn aktualisiert. Die InOut-Variable IN wird direkt an eine übergeordnete Variable gebunden, sodass jeder Lesevorgriff den tatsächlich aktuellen Wert dieser Variable liefert.

Technische Besonderheiten

- **InOut-Parameter:** IN ist als InOut-Variable deklariert. Dies bedeutet, dass sie als Referenz auf eine externe Variable dient. Der FB liest den Wert dieser referenzierten Variable, modifiziert sie jedoch nicht (keine Rückwirkung).
- **Initialwerte:** Sowohl IN als auch OUT sind mit 0 initialisiert. Der Initialwert von IN wird nur dann wirksam, wenn die referenzierte Variable nicht explizit gesetzt ist oder in einer Umgebung ohne externen Wert verwendet wird.
- **Keine Zustandsautomaten-Komplexität:** Der FB ist als einfacher SimpleFB mit nur einem Zustand implementiert. Es gibt keine internen Zustandsübergänge außer dem Ausführen der Aktion.

Zustandsübersicht

Der Funktionsblock besitzt einen einzigen Zustand REQ:

Zustand	Beschreibung
REQ	Wartet auf das Ereignis REQ. Bei Eintritt wird die Aktion REQ (OUT := IN) ausgeführt und danach das Ereignis CNF gesendet. Nach CNF verbleibt der FB im selben Zustand.

Anwendungsszenarien

- **Puffern eines Sensorsignals:** Ein DWORD-Sensorwert (z.B. Zählerstand) soll zu bestimmten Zeitpunkten abgetastet und für die weitere Verarbeitung zwischengespeichert werden.
- **Dekomposition von DWORD-Daten:** In Kombination mit anderen Bausteinen wie EXTRACT_BYTE oder SPLIT_DWORD kann der gelesene Wert in einzelne Bytes oder Bits zerlegt werden.
- **Prozessabbild aktualisieren:** Der Baustein eignet sich, um den aktuellen Wert einer externen globalen Variable (z.B. aus einem industriellen Feldbus) kontrolliert in einen lokalen Puffer zu übernehmen.

Vergleich mit ähnlichen Bausteinen

- **GET_DWORD mit einfachem Input:** Ein hypothetischer Baustein mit einem normalen Daten-Eingang (VarInput) würde den Wert nicht als Referenz übergeben, sondern würde bei jedem Ereignis den aktuellen Eingangswert verwenden. GET_DWORD mit InOut bietet dagegen eine feste Bindung an eine bestimmte Variable, was in Szenarien mit mehreren Zugriffen auf dieselbe Variable konsistenter ist.
- **SET_DWORD:** Der symmetrische Baustein SET_DWORD würde den Wert einer InOut-Variable *setzen*, während GET_DWORD sie nur liest. Beide verwenden das InOut-Konzept, aber mit entgegengesetzter Wirkrichtung.

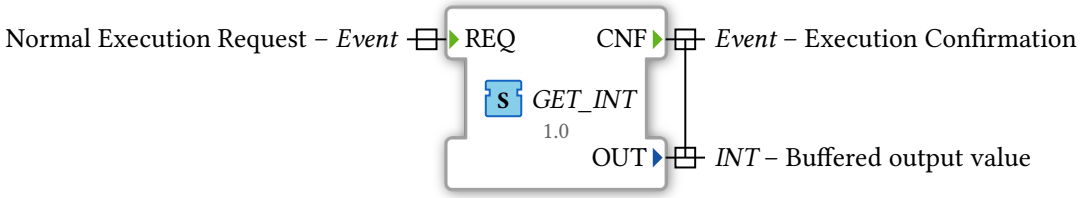
Fazit

GET_DWORD ist ein einfacher, aber nützlicher Baustein zum kontrollierten Auslesen eines DWORD-Werts aus einer extern referenzierten Variable. Seine Stärke liegt in der Pufferung des Werts und der Trennung zwischen der Quelle (InOut) und dem Ausgang (OUT). Dank des einfachen Zustandsautomaten und des standardisierten REQ/CNF-Schnittstellenmusters lässt er sich leicht in bestehende 4diac-Applikationen integrieren.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GET_INT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **GET_INT** dient dazu, den aktuellen Wert einer INT-Variablen (als InOut-Parameter definiert) auszulesen und als gepufferten Wert am Ausgang bereitzustellen. Die Pufferung erfolgt bei jedem Ausführungszyklus, sodass der Ausgangswert bis zur nächsten Anforderung stabil bleibt.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung
REQ	Normaler Ausführungsrequest; löst das Auslesen der InOut-Variablen **IN** aus.

Ereignis-Ausgänge

Ereignis	Beschreibung
CNF	Bestätigung der Ausführung; signalisiert, dass der Wert von **IN** gepuffert und am Ausgang **OUT** bereitsteht.

Daten-Eingänge

– *Keine eigenständigen Dateneingänge vorhanden.*
Der Eingabewert wird über den InOut-Parameter **IN** bereitgestellt (siehe nächster Abschnitt).

Daten-Ausgänge

Ausgang	Typ	Beschreibung
OUT	INT	Gepufferter Ausgabewert, der den letzten gelesenen Wert von **IN** enthält. Initialwert: 0.

Adapter

– *Keine Adapter vorhanden.*

InOut-Variablen (implizit)

Variable	Typ	Beschreibung
IN	INT	Quelle des auszulesenden Werts. Kann sowohl als Eingabe als auch als Ausgabe verwendet werden (hier nur lesend). Initialwert: 0.

Funktionsweise

Der FB arbeitet als einfacher Lesepuffer:

1. Beim Eintreffen des Ereignisses **REQ** wird die **ST**-Algorithmus `REQ` ausgeführt:

   ```
   OUT := IN;
   ```

2. Der aktuelle Wert der InOut-Variablen **IN** wird in den Ausgang **OUT** kopiert.
3. Anschließend wird das Ereignis **CNF** ausgegeben, um den Abschluss der Operation zu signalisieren.

Da **IN** als InOut-Parameter deklariert ist, wird innerhalb des Bausteins direkt auf die Speicherstelle der übergeordneten Applikation zugegriffen. Der Wert wird nicht verändert, sondern lediglich ausgelesen und zwischengespeichert.

Technische Besonderheiten

- **InOut-Parameter**: Die Verwendung einer InOut-Variable erlaubt den direkten Zugriff auf die Datenquelle ohne separate Daten-Eingänge und -Ausgänge. Dies reduziert die Anzahl der Verbindungen und spart Ressourcen.
- **Pufferung**: Der Ausgang **OUT** behält seinen Wert, bis der nächste **REQ**-Aufruf erfolgt. Dadurch ist der Ausgabewert unabhängig von möglichen Änderungen der Quelle zwischen zwei Lesezyklen.
- **Initialwerte**: Sowohl **IN** als auch **OUT** starten mit 0, sodass auch ohne vorherige Ausführung definierte Werte vorliegen.

Zustandsübersicht

Der FB besitzt einen einzigen Zustand **REQ**:

- **REQ**: Bei Aktivierung wird der Algorithmus `REQ` ausgeführt, danach das Ereignis **CNF** gesendet. Der Zustand bleibt bestehen (kein Übergang in einen anderen Zustand).

Zustand	Eintrittsbedingung	Aktion	Ausgangsereignis
REQ	Ereignis **REQ** empfangen	`OUT := IN;`	CNF

Anwendungsszenarien

- **Datenpufferung**: Ein von mehreren Bausteinen genutzter INT-Wert soll stabil zur Verfügung stehen, ohne dass jeder Baustein direkt auf die Quelle zugreifen muss.
- **Synchronisation**: In zeitkritischen Steuerungen kann der aktuelle Wert einer Variablen zu einem bestimmten Zeitpunkt „eingefroren“ und später verarbeitet werden.
- **Ersatz eines MOVE-FBs**: Wenn aus Performance-Gründen ein direkter Speicherzugriff über InOut bevorzugt wird, ersetzt **GET_INT** den klassischen MOVE-Baustein mit separatem Eingang und Ausgang.

Vergleich mit ähnlichen Bausteinen

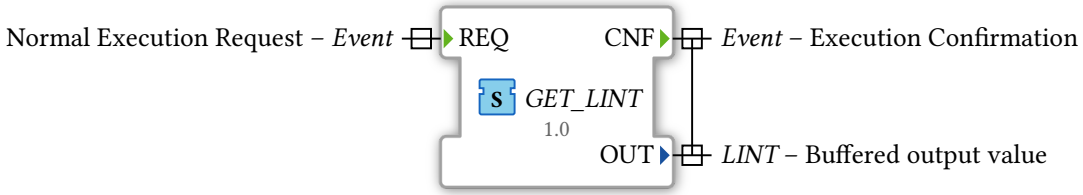
Baustein	Typ des Eingangs	Pufferung	Besonderheit
GET_INT	InOut (INT)	Ja (einmaliges Lesen bei REQ)	Zugriff auf Originalvariable, keine zusätzliche Variable erforderlich
MOVE (z.B. IEC 61131-3)	Eingang (INT)	Ja (bei jeder Ausführung)	Kopiert Wert von einem expliziten Eingang in einen Ausgang
F_TRIG / R_TRIG	Eingang (BOOL)	Nein (nur Flankenerkennung)	Arbeitet mit Booleschen Werten

Der Hauptunterschied liegt im Verzicht auf einen separaten Dateneingang durch die Nutzung eines InOut-Parameters.

Fazit

GET_INT ist ein einfacher, aber effektiver Funktionsblock zum gepufferten Auslesen einer INT-Variable. Durch die Verwendung eines InOut-Parameters reduziert er den Verbindungsaufwand und eignet sich besonders für Anwendungen, in denen ein stabiler Momentanwert benötigt wird. Seine klare, zustandsarme Struktur macht ihn zuverlässig und leicht verständlich.

GET_LINT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **GET_LINT** dient dem Auslesen eines Wertes vom Typ **LINT** aus einer InOut-Variablen und stellt diesen Wert gepuffert an seinem Ausgang zur Verfügung. Der Baustein ist Teil der Bibliothek `logiBUS::utils::conversion::provide` und realisiert eine einfache Lese-Puffer-Funktionalität.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
REQ	Event	Normaler Ausführungsauftrag

Der Ereigniseingang REQ löst die Ausführung des Bausteins aus. Dabei wird der aktuelle Wert der InOut-Variablen **IN** gelesen und im Ausgang **OUT** zwischengespeichert.

Ereignis-Ausgänge

Name	Typ	Kommentar
CNF	Event	Bestätigung der Ausführung

Das Ereignis CNF wird nach erfolgreicher Verarbeitung des REQ-Ereignisses gesendet. Es signalisiert, dass der Ausgang OUT den aktuellen Wert der InOut-Variablen IN enthält.

Daten-Eingänge

Name	Typ	Kommentar
IN	LINT	Quellvariable (InOut) – Initialwert: 0

IN ist als InOut-Variable deklariert. Sie kann sowohl gelesen als auch beschrieben werden, wird in diesem Baustein jedoch ausschließlich gelesen. Der übergebene Wert wird bei jedem REQ-Ereignis in den Ausgang OUT kopiert.

Daten-Ausgänge

Name	Typ	Kommentar
OUT	LINT	Gepufferter Ausgabewert – Initialwert: 0

Der Ausgang **OUT** hält den zuletzt von der InOut-Variablen **IN** gelesenen Wert. Er wird bei jedem REQ-Ereignis aktualisiert.

Adapter

Der Baustein besitzt keine Adapter-Schnittstellen.

Funktionsweise

Der GET_LINT-Baustein implementiert eine einfache Pufferfunktion. Er besitzt einen einzigen Eintrittszustand **REQ**, der durch ein Ereignis am gleichnamigen Eingang aktiviert wird. In diesem Zustand wird der Algorithmus `REQ` ausgeführt:

```
OUT := IN;
```

Anschließend wird das Bestätigungsereignis **CNF** gesendet. Der Ausgang **OUT** enthält danach den aktuellen Wert der InOut-Variablen **IN**. Der Baustein wartet dann auf den nächsten REQ-Impuls.

Technische Besonderheiten

- **InOut-Variable**: Der Baustein verwendet einen InOut-Parameter (`IN`). Dies erlaubt den direkten Zugriff auf eine Variable, die sowohl gelesen als auch beschrieben werden kann. Im Falle von GET_LINT wird der Wert jedoch nur gelesen und nicht verändert.
- **Datentyp LINT**: Der Baustein arbeitet ausschließlich mit dem 64-Bit-Ganzzahltyp LINT (Long Integer). Er ist für Umgebungen gedacht, in denen große Ganzzahlwerte verarbeitet werden.
- **Einfachste Zustandsmaschine**: Es existiert nur ein Zustand mit einer direkten Aktion und einer Ausgabe. Dadurch ist der Baustein deterministisch und ressourcenschonend.
- **Keine Fehlerbehandlung**: Der Baustein enthält keinerlei Plausibilitätsprüfungen oder Timeouts. Er setzt voraus, dass die InOut-Variable gültige Daten liefert.

Zustandsübersicht

Zustand	Aktion	Ausgabe
REQ	Kopieren von IN nach OUT	CNF

Der Baustein startet im Zustand **REQ** und kehrt nach Ausführung der Aktion und Senden von CNF in denselben Zustand zurück (Endlosschleife bei jedem Ereignis).

Anwendungsszenarien

- **Zwischenspeicherung eines LINT-Werts**: Wenn ein LINT-Wert aus einer globalen Variablen oder einem anderen Baustein gelesen und für eine bestimmte Zeit konstant gehalten werden soll (z. B. für eine nachfolgende Verarbeitung).
- **Synchronisation**: Der Baustein kann verwendet werden, um den aktuellen Wert einer InOut-Variablen in einem bestimmten Takt zu erfassen und an andere Bausteine weiterzugeben.
- **Entkopplung**: Durch den Puffer wird der lesende Baustein von der Quelle entkoppelt – Änderungen der Quelle wirken sich erst nach dem nächsten REQ-Impuls aus.
- **Einfache Wertübernahme**: In Steuerungssystemen kann GET_LINT genutzt werden, um einen externen Messwert oder einen Parameter von einem anderen Funktionsbaustein abzuholen.

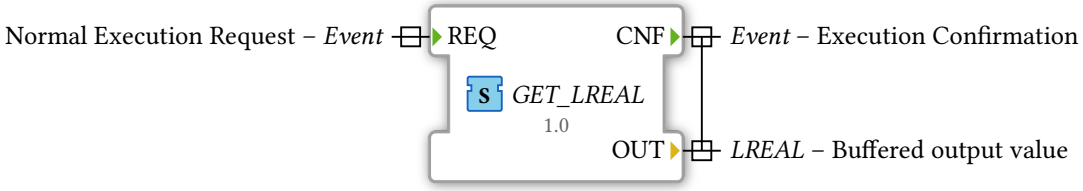
Vergleich mit ähnlichen Bausteinen

- **GET_DINT / GET_BOOL**: Analoge Bausteine für andere Datentypen (DINT, BOOL). Die Funktionsweise ist identisch; lediglich der Datentyp unterscheidet sich.
- **MOVE (z. B. MOVE_INT)**: Ein MOVE-Baustein kopiert ebenfalls einen Wert, verwendet jedoch standardmäßig Eingangs- und Ausgangsvariablen statt InOut. InOut-Variablen erlauben eine direkte Referenz auf die Quelle ohne zusätzlichen Data-Flow.
- **LINT_READER** (fiktiv): Ein hypothetischer Baustein, der zusätzliche Funktionen wie Puffer-Timeout oder Lesesperre bietet. GET_LINT ist dagegen minimalistisch und schnell.

Fazit

Der GET_LINT-Funktionsbaustein bietet eine schlanke und zuverlässige Methode, um einen LINT-Wert aus einer InOut-Variable zu lesen und gepuffert bereitzustellen. Dank seiner einfachen Zustandsmaschine und der klaren Schnittstelle eignet er sich hervorragend für Basis-Synchronisationsaufgaben in IEC 61499-basierten Systemen. Durch die Verwendung von InOut-Parametern bleibt die Implementierung flexibel, ohne dass zusätzliche Datenleitungen oder Protokolle erforderlich sind.

GET_LREAL



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `GET_LREAL` dient dem lesenden Zugriff auf eine `LREAL`-Variable (64-Bit-Gleitkommazahl), die als InOut-Parameter übergeben wird. Der gelesene Wert wird intern gepuffert und am Ausgang `OUT` zur Verfügung gestellt. Die Ausführung wird über ein Ereignis gesteuert.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Datentyp	Kommentar
REQ	Event	Normaler Ausführungsanstoß; liest den Wert von `IN`

Ereignis-Ausgänge

Ereignis	Datentyp	Kommentar
CNF	Event	Bestätigung der Ausführung; wird nach dem Lesen gesendet

Daten-Eingänge

Der Baustein besitzt keine eigenständigen Dateneingänge. Die Quelldaten werden über den InOut-Parameter `IN` bereitgestellt.

Daten-Ausgänge

Ausgang	Datentyp	Initialwert	Kommentar
OUT	LREAL	0.0	Gepufferter Ausgabewert

Adapter

Keine Adapter vorhanden.

Funktionsweise

1. Ein Ereignis am Eingang `REQ` löst die Verarbeitung aus.
2. Der Baustein führt die ST-Anweisung `OUT := IN;` aus. Dabei wird der aktuelle Wert der InOut-Variable `IN` gelesen und in der lokalen Ausgangsvariablen `OUT` zwischengespeichert.
3. Nach erfolgreichem Lesen wird das Ereignis `CNF` ausgegeben. Der Wert von `OUT` bleibt bis zur nächsten Ausführung erhalten.

Technische Besonderheiten

- Die Quelle `IN` ist als **InOut-Variable** deklariert. Sie muss von der aufrufenden Umgebung mit einer tatsächlichen Variablen (z. B. einer globalen oder einer Hardware-Ressource) verbunden werden.
- Der Baustein puffert den gelesenen Wert: solange kein neuer `REQ`-Impuls erfolgt, liefert `OUT` den zuletzt gelesenen Zustand.
- Beide Parameter (`IN` und `OUT`) sind mit dem Initialwert `0.0` belegt.

Zustandsübersicht

Da es sich um einen einfachen Funktionsbaustein (SimpleFB) handelt, ist nur ein einziger ECC-Zustand vorhanden:

Zustand	Aktion	Ausgabeereignis
REQ	`OUT := IN;`	CNF

Anwendungsszenarien

- **Gepuffertes Lesen einer hochfrequenten Prozessgröße** (z. B. Temperatur oder Druck), um den Wert im Steuerungszyklus konstant zu halten.
- **Zugriff auf eine shared Memory-Variable** in einer Multitasking-Umgebung, bei der die Lesegeschwindigkeit entkoppelt werden soll.

- **Vorbereitung von Daten für nachgelagerte Berechnungen**, die auf einen stabilen, zwischengespeicherten Gleitkommawert angewiesen sind.

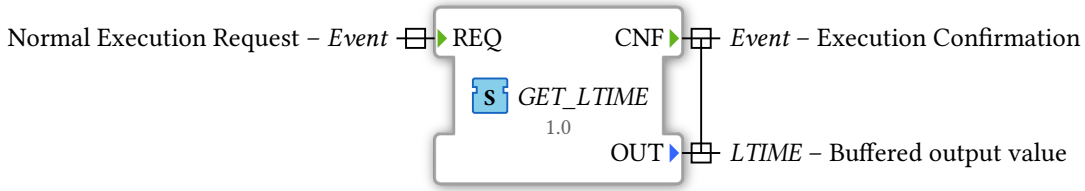
Vergleich mit ähnlichen Bausteinen

Bausteine wie `GET_REAL` oder `GET_INT` arbeiten nach dem gleichen Prinzip, jedoch mit kürzeren Datentypen. `GET_LREAL` ist speziell für Anwendungen ausgelegt, die eine hohe numerische Genauigkeit (64-Bit) erfordern. Im Unterschied zu reinen Lesebausteinen ohne Pufferung (z. B. direkte Variablenverknüpfung) garantiert der gepufferte Ausgang eine deterministische und vom Zeitpunkt des Lesens unabhängige Bereitstellung.

Fazit

`GET_LREAL` ist ein einfacher, zuverlässiger Baustein zum gepufferten Auslesen einer `LREAL`-Quelle. Er eignet sich besonders für die Entkopplung von Lese- und Verarbeitungslogik und stellt sicher, dass der bereitgestellte Wert über einen kompletten Ausführungszyklus konstant bleibt.

GET_LTIME



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **GET_LTIME** dient dazu, den aktuellen Wert einer LTIME-Variablen auszulesen und als gepufferten Ausgangswert bereitzustellen. Er greift über eine InOut-Schnittstelle auf die Quelle zu und stellt den gelesenen Wert am Ausgang zur Verfügung, sobald ein Ereignis eintrifft. Dies ermöglicht ein kontrolliertes, einmaliges Auslesen einer Zeitvariablen.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Datentyp	Kommentar
REQ	Event	Normaler Ausführungsauftrag. Verbunden mit dem Eingangswert IN.

Ereignis-Ausgänge

Ereignis	Datentyp	Kommentar
CNF	Event	Bestätigung der Ausführung. Verbunden mit den Ausgängen OUT und IN.

Daten-Eingänge

Der Baustein besitzt keine klassischen Dateneingänge. Der Zugriff auf den Quellwert erfolgt ausschließlich über die InOut-Variable.

Daten-Ausgänge

Name	Datentyp	Kommentar
OUT	LTIME	Gepufferter Ausgabewert (Initialwert LT#0s).

Adapter

Keine.

Funktionsweise

Der Funktionsbaustein implementiert eine einfache, ereignisgesteuerte Kopierlogik:

1. Ein eingehendes Ereignis **REQ** löst die Ausführung des internen Algorithmus aus.
2. Der Algorithmus weist dem Ausgang **OUT** den aktuellen Wert der InOut-Variablen **IN** zu (`OUT := IN;`).
3. Nach erfolgreicher Zuweisung wird das Ausgangsereignis **CNF** gesendet, um der aufrufenden Instanz die Fertigstellung zu signalisieren.

Somit wird der Wert der externen LTIME-Variablen nur zu dem Zeitpunkt gelesen, an dem das REQ-Ereignis eintrifft, und bis zum nächsten Lesen im Ausgang zwischengespeichert.

Technische Besonderheiten

- **InOut-Variable**: Der Zugriff auf die Quelle erfolgt über eine InOut-Schnittstelle, die eine direkte Referenz auf eine externe Variable darstellt. Änderungen an der externen Variable zwischen zwei REQ-Aufrufen werden erst beim nächsten Lesevorgang übernommen.
- **Pufferung**: Der gelesene Wert bleibt am Ausgang OUT erhalten, bis ein erneutes REQ-Ereignis eingeht. Dies schützt vor asynchronen Änderungen der Quelle während der Verarbeitung.
- **Rücklesen des Eingangs**: Das Ereignis **CNF** ist nicht nur mit OUT, sondern auch mit IN verbunden. Dies kann in bestimmten Laufzeitumgebungen genutzt werden, um die Weiterleitung des InOut-Wertes zu steuern, hat jedoch im Standardverhalten keine zusätzliche funktionale Bedeutung.

Zustandsübersicht

Der Baustein ist als SimpleFB realisiert und besitzt genau einen internen Zustand:

Zustand	Aktion	Ausgangsereignis
REQ	Algorithmus REQ (OUT := IN)	CNF

Nach dem Empfang von **REQ** wird der Zustand einmalig durchlaufen und die Ausgabe erzeugt. Es gibt keine weiteren Zustände oder Verzweigungen.

Anwendungsszenarien

- **Zeitstempel-Erfassung**: Einmaliges Einfrieren eines aktuellen Systemzeitwerts (LTIME) zu einem bestimmten Ereignis, z. B. bei einem Alarmsignal.
- **Synchronisation von Zeitmessungen**: Auslesen eines gemeinsamen Zeitzählers in einer industriellen Steuerung, um konsistente Zeitdaten an verschiedene nachfolgende Bausteine weiterzugeben.
- **Pufferung von Zeitwerten** in datengetriebenen Architekturen, bei denen eine LTIME-Variable mehrfach verwendet wird, aber exakt zum selben Zeitpunkt gelesen werden soll.

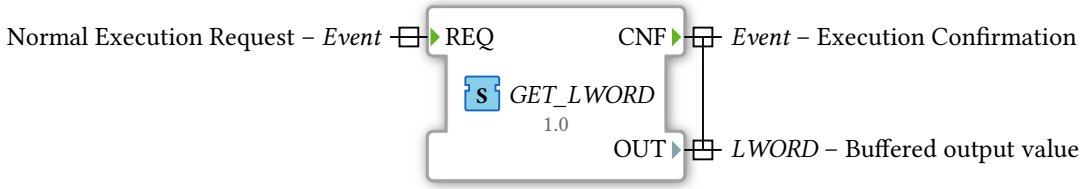
Vergleich mit ähnlichen Bausteinen

- **GET_VALUE (für andere Datentypen)**: Funktionsweise analog, jedoch für skalare Datentypen wie INT, REAL etc. GET_LTIME ist auf den Datentyp LTIME spezialisiert.
- **Latch-Funktionsbausteine**: Während Latch-Bausteine einen Wert bei einem Ereignis speichern und halten, erzeugt GET_LTIME zusätzlich ein Bestätigungsereignis und liest aus einer InOut-Variablen statt aus einem normalen Dateneingang.
- **MOVE-Baustein**: Ein MOVE kopiert ebenfalls einen Wert von einem Eingang auf einen Ausgang, jedoch ohne Ereignissteuerung und ohne InOut-Unterstützung. GET_LTIME bietet eine ereignisgesteuerte, gepufferte Variante.

Fazit

Der Baustein **GET_LTIME** ist eine einfache, aber effektive Komponente zum kontrollierten Auslesen einer LTIME-Variablen über eine InOut-Schnittstelle. Seine ereignisgesteuerte Arbeitsweise und die Pufferung des Ausgangswerts machen ihn besonders geeignet für zeitkritische Anwendungen, bei denen ein konsistenter Zeitwert zu einem definierten Zeitpunkt benötigt wird. Die klare Schnittstelle und die einfache Zustandslogik erleichtern die Integration in größere Steuerungsanwendungen.

GET_LWORD



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **GET_LWORD** dient dem konsistenten Auslesen eines LWORD-Wertes aus einer **InOut-Variable**. Er puffert den aktuellen Wert der quellvariablen und stellt ihn am Ausgang `OUT` bereit. Dies ist besonders nützlich, wenn der Inhalt einer gemeinsam genutzten Variablen zu einem definierten Zeitpunkt festgehalten werden soll, ohne von nachfolgenden Schreibzugriffen anderer Bausteine beeinflusst zu werden.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung
`REQ`	Startet die Auslese-Operation. Der aktuelle Wert von `IN` wird in `OUT` übernommen.

Ereignis-Ausgänge

Ereignis	Beschreibung
`CNF`	Bestätigt die erfolgreiche Durchführung der Operation. Wird nach dem Kopiervorgang ausgelöst.

Daten-Eingänge

Der Baustein besitzt keine klassischen Dateneingänge, sondern verwendet einen **InOut-Parameter**, der als bidirektionale Schnittstelle fungiert.

Daten-Ausgänge

Variable	Typ	Beschreibung
`OUT`	`LWORD`	Gepufferter Ausgabewert. Enthält die Kopie des Werts von `IN` zum Zeitpunkt des letzten `REQ`. Initialwert = `0`.

Adapter

Keine vorhanden.

Funktionsweise

Der Baustein implementiert einen einfachen Ablauf:

1. **Ereignis `REQ`** löst die Ausführung des Algorithmus `REQ` aus.
2. **Algorithmus `REQ`**: Die Anweisung `OUT := IN;` kopiert den aktuellen Wert der InOut-Variablen `IN` in den Ausgangspuffer `OUT`. Die InOut-Schnittstelle gewährleistet dabei einen direkten Lesezugriff auf die quellvariable.
3. Nach Abschluss der Zuweisung wird das Ausgangsereignis **`CNF`** gesendet. Die Ereignisausgänge `CNF` sind sowohl mit `OUT` als auch mit `IN` verknüpft, sodass nachfolgende Bausteine sowohl den neuen Wert als auch die Referenz auf die InOut-Variable nutzen können.

Die Pufferung stellt sicher, dass der Wert von `OUT` während eines Ablaufzyklus stabil bleibt, auch wenn die Quelle `IN` zwischenzeitlich von anderen Bausteinen geändert wird.

Technische Besonderheiten

- **InOut-Parameter**: Die Variable `IN` ist als `InOutVar` deklariert. Dies ermöglicht dem Baustein, direkt auf einen externen Speicherort zuzugreifen, ohne dass eine zusätzliche Verbindung als Dateneingang nötig ist. Der Baustein liest diesen Wert, verändert ihn jedoch nicht.
- **Einfache Zustandslogik**: Der Baustein besitzt nur einen einzigen Zustand `REQ`. Dadurch wird bei jedem `REQ`-Ereignis genau eine Aktion ausgeführt – die Ausführung ist deterministisch und ohne Verzögerung.
- **Typ der Variablen**: `LWORD` (64-Bit logarithmischer oder Bifeldwert) wird als sicherer Puffer behandelt.

Zustandsübersicht

Zustand	Beschreibung
`REQ`	Initialer und einziger Zustand. Nach Eingang von `REQ` wird der Algorithmus ausgeführt und `CNF` gesendet. Der Zustand bleibt erhalten, sodass bei erneutem `REQ` der Vorgang wiederholt wird.

Anwendungsszenarien

- **Konsistentes Abtasten**: Ein `GET_LWORD`-Baustein kann eingesetzt werden, um den Wert eines zyklisch aktualisierten Sensors zu einem definierten Zeitpunkt auszulesen und für die weitere Verarbeitung (z. B. eine Berechnungskette) festzuhalten.
- **Vermeidung von Wettlaufsituationen**: In Systemen mit mehreren parallel arbeitenden Funktionsblöcken, die auf dieselbe Speicherstelle zugreifen, verhindert die Pufferung, dass ein Baustein zwischendurch einen inkonsistenten Wert erhält.
- **Datenprotokollierung**: Vor dem Ablegen einer Messung in einem Protokollspeicher kann mit `GET_LWORD` eine Momentaufnahme erstellt werden.

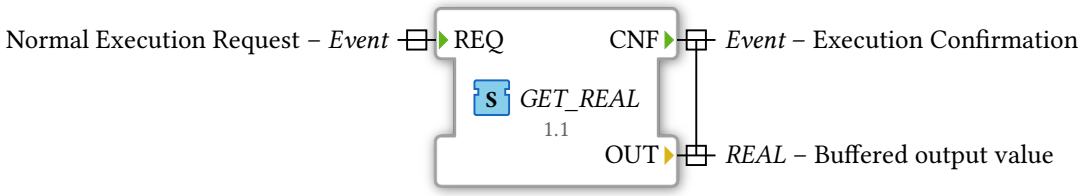
Vergleich mit ähnlichen Bausteinen

- **`GET_INT`, `GET_DWORD`, `GET_BOOL`** etc.: Diese Bausteine folgen dem gleichen Prinzip, unterscheiden sich jedoch im Datentyp. `GET_LWORD` ist speziell für 64-Bit-Werte vorgesehen.
- **Einfache Eingangsvariable**: Ein reiner Dateneingang würde erfordern, dass der verbindende Baustein den Wert selbst puffert oder die Verbindung ständig aktualisiert. Der InOut-Mechanismus bietet eine Referenz auf das Original, was in manchen Architekturen direkter und speicherschonender ist.
- **`MUX`-Bausteine**: Ein Multiplexer könnte ebenfalls einen Wert auswählen, benötigt aber zusätzliche Steuerleitungen. `GET_LWORD` ist für das einmalige, deterministische Auslesen optimiert.

Fazit

Der Funktionsblock `GET_LWORD` ist ein einfacher, aber wirkungsvoller Baustein zur sicheren und zeitlich kontrollierten Abfrage einer InOut-Variablen vom Typ `LWORD`. Seine minimale Zustandsmaschine und die direkte Kopierlogik machen ihn ideal für Echtzeitanwendungen, bei denen konsistente Datenabgriffe erforderlich sind. Durch die Nutzung des InOut-Konzepts wird eine effiziente Kopplung an bestehende Speicherstellen erreicht, ohne die Verbindungsstruktur aufzublähen.

GET_REAL



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **GET_REAL** dient dazu, einen REAL-Wert aus einer **InOut-Variable** auszulesen und diesen in einem gepufferten Ausgang bereitzustellen. Der ausgelesene Wert wird beim Empfang eines Ereignisses am Eingang `REQ` aus der InOut-Variable `IN` gelesen und am Ausgang `OUT` ausgegeben. Nach der Ausführung wird das Bestätigungsereignis `CNF` gesendet.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung	Mit Variablen
REQ	Normale Ausführungsanforderung – löst das Lesen der InOut-Variable aus.	`IN`

Ereignis-Ausgänge

Ereignis	Beschreibung	Mit Variablen
CNF	Bestätigung der Ausführung – wird nach der Bearbeitung von `REQ` gesendet.	`OUT`, `IN`

Daten-Eingänge

Variable	Typ	Initialwert	Beschreibung
IN	REAL	0.0	InOut-Variable, deren aktueller Wert gelesen wird.

Hinweis: `IN` ist als *InOut-Variable* deklariert – sie kann sowohl gelesen als auch geschrieben werden. In diesem FB wird sie jedoch nur gelesen.

Daten-Ausgänge

Variable	Typ	Initialwert	Beschreibung
OUT	REAL	0.0	Gepufferter Ausgang – hält den gelesenen Wert von `IN` bis zur nächsten Aktualisierung.

Adapter

Keine Adapter vorhanden.

Funktionsweise

1. Ein eingehendes Ereignis am Eingang `REQ` triggert die Ausführung des internen Algorithmus `REQ`.
2. Im Algorithmus wird der aktuelle Wert der InOut-Variable `IN` in die Ausgangsvariable `OUT` kopiert:

   ```
   OUT := IN;
   ```

3. Anschließend wird das Bestätigungsereignis `CNF` gesendet. Die Ausgänge `OUT` und `IN` werden mit dem Ereignis assoziiert, sodass sie gleichzeitig gültig sind.
4. Der Ausgangswert bleibt solange erhalten, bis ein weiteres `REQ`-Ereignis eintrifft und `OUT` neu geschrieben wird.

Technische Besonderheiten

- **InOut-Verwendung:** Der Baustein nutzt eine InOut-Variable (`IN`). Diese kann von außen beschrieben oder gelesen werden; hier dient sie ausschließlich als Quelle.
- **Initialisierung:** Beide REAL-Variablen sind mit `0.0` vorbelegt.
- **Minimaler Overhead:** Der FB hat nur einen Zustand (`REQ`) und einen einfachen Zuweisungsalgorithmus – er arbeitet ohne Verzögerung oder zusätzliche Logik.
- **Standardkonformität:** Der Baustein ist gemäß IEC 61499-1 definiert und für die Verwendung in Automatisierungsumgebungen wie 4diac IDE vorgesehen.

Zustandsübersicht

Der FB besitzt nur einen einzigen Zustand:

- **Zustand `REQ`:** Beim Eintritt wird der Algorithmus `REQ` ausgeführt (OUT := IN) und das Ereignis `CNF` gesendet. Der FB verbleibt in diesem Zustand und ist stets bereit für die nächste Anforderung.

Es gibt keine weiteren Zustände oder Verzweigungen – der FB ist deterministisch und reagiert unmittelbar auf jedes `REQ`-Ereignis.

Anwendungsszenarien

- **Lesen eines Prozesswertes:** In einer SPS-Steuerung kann `GET_REAL` verwendet werden, um einen aktuellen Sensorwert, der als InOut-Variable vorliegt, auszulesen und an einen nachfolgenden Funktionsblock weiterzugeben (z. B. für Berechnungen oder Visualisierung).
- **Werteabgleich zwischen Bausteinen:** Wenn ein anderer FB eine InOut-Schnittstelle mit einem REAL-Wert bereitstellt, kann `GET_REAL` diesen Wert abrufen und als gepufferten Ausgang verfügbar machen.
- **Test und Simulation:** Der Baustein eignet sich zum Debuggen, um den aktuellen Wert einer Variablen zu einem bestimmten Zeitpunkt festzuhalten und zu protokollieren.

Vergleich mit ähnlichen Bausteinen

Ähnliche Bausteine existieren für andere Datentypen, z. B. `GET_BOOL`, `GET_INT`, `GET_LREAL` etc. Sie alle teilen das gleiche Prinzip: Lesen einer InOut-Variable und Bereitstellen als gepufferten Ausgang. Der Unterschied liegt ausschließlich im Datentyp und der damit verbundenen Genauigkeit bzw. Wertebereich.

Baustein	Datentyp	Einsatzbereich
GET_BOOL	BOOL	Binäre Signale
GET_INT	INT	Ganzzahlen (16 Bit)
GET_REAL	REAL	Gleitkommazahlen (32 Bit)
GET_LREAL	LREAL	Präzise Gleitkommazahlen (64 Bit)

Alle diese Bausteine lassen sich analog verwenden und bieten eine einheitliche Schnittstelle für den Zugriff auf InOut-Variablen.

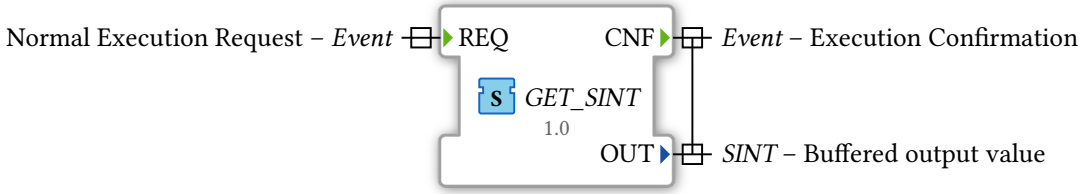
Fazit

Der `GET_REAL`-Funktionsblock ist ein schlanker und zuverlässiger Baustein zum Lesen einer REAL-InOut-Variable. Er erfüllt eine grundlegende Aufgabe in der IEC-61499-Welt: das Weiterleiten eines Wertes aus einer bidirektionalen Schnittstelle in einen gepufferten Ausgang. Durch seine Einfachheit und klare Semantik eignet er sich sowohl für den Produktiveinsatz als auch für Testzwecke. Die Kombination mit den parallelen Bausteinen anderer Datentypen macht ihn zu einem flexiblen Werkzeug in der Automatisierungstechnik.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GET_SINT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **GET_SINT** dient dazu, einen Wert des Datentyps `SINT` (signed short integer) aus einer als `InOut` deklarierten Variable auszulesen und als gepufferten Ausgang bereitzustellen. Er wird typischerweise verwendet, um auf eine gemeinsam genutzte Variable oder einen physikalischen Eingang zuzugreifen, ohne den ursprünglichen Wert zu verändern. Der Baustein arbeitet nach dem **IEC 61499** Standard.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Datentyp	Kommentar
REQ	Event	Normale Ausführungsanforderung. Löst das Auslesen des InOut-Variablen aus.

- **With-Verknüpfung**: Der Ereigniseingang REQ ist mit der Daten-Variable IN verknüpft.

Ereignis-Ausgänge

Ereignis	Datentyp	Kommentar
CNF	Event	Bestätigung der Ausführung. Wird nach erfolgreichem Kopiervorgang gesendet.

- **With-Verknüpfung**: Der Ereignisausgang CNF ist mit den Daten-Variablen OUT und IN verknüpft.

Daten-Eingänge

Der Baustein besitzt keine klassischen Daten-Eingänge. Stattdessen wird eine InOut-Variable verwendet.

Daten-Ausgänge

Variable	Datentyp	Initialwert	Kommentar
OUT	SINT	0	Gepufferter Ausgabewert. Enthält die Kopie von IN nach einer erfolgreichen REQ-Ausführung.

Adapter

Keine.

Funktionsweise

Der Baustein führt einen einfachen Kopiervorgang durch:

1. Ein Ereignis am Eingang REQ triggert die Ausführung des hinterlegten Algorithmus.
2. Der Algorithmus weist dem Ausgang OUT den aktuellen Wert der InOut-Variable IN zu: `OUT := IN;`.
3. Nach Abschluss der Zuweisung wird das Ereignis CNF gesendet, um dem aufrufenden Baustein die Fertigstellung zu signalisieren.

Die InOut-Variable IN erlaubt den direkten Zugriff auf eine externe Variable oder einen Speicherort, ohne dass diese als separater Eingang oder Ausgang geführt werden muss. Der Baustein puffert den gelesenen Wert, sodass der Ausgang OUT auch nach Änderungen der Quelle stabil bleibt, bis die nächste Anforderung erfolgt.

Technische Besonderheiten

- **Verwendung von InOut-Variablen**: IN ist als InOut deklariert, d.h. sie kann sowohl gelesen als auch geschrieben werden. In diesem Fall wird sie nur gelesen, was eine flexible Kopplung mit dem aufrufenden Netzwerk ermöglicht.
- **Pufferung**: Der Ausgang OUT speichert den gelesenen Wert bis zur nächsten Aktualisierung durch REQ. Dies verhindert unerwünschte Seiteneffekte durch Änderungen der Quelle.
- **Keine Zustandslogik**: Der Baustein besteht aus einem einzigen Zustand (REQ), der auf jedes eingehende Ereignis sofort reagiert. Keine Verzögerung, kein Warten auf Bedingungen.
- **Initialwerte**: Sowohl IN als auch OUT sind mit 0 initialisiert, was einen definierten Startzustand gewährleistet.

Zustandsübersicht

Der Baustein verwendet einen einfachen ECA-Zustandsautomaten mit genau einem Zustand:

Zustand	Aktion	Ausgehendes Ereignis
REQ	REQ-Algorithmus (OUT := IN)	CNF

- Jedes eintreffende `REQ`-Ereignis führt sofort zur Aktion und sendet `CNF`. Es gibt keine Verzweigungen, Fehlerzustände oder Zeitverhalten.

Anwendungsszenarien

- **Auslesen von Hardware-Eingängen**: Wenn ein SINT-Wert von einem analogen Eingangsbaustein oder einer speicherprogrammierbaren Steuerung (SPS) gelesen werden muss, kann `GET_SINT` als Schnittstelle verwendet werden, um den aktuellen Wert zu einem bestimmten Zeitpunkt abzugreifen.
- **Zugriff auf geteilte Variablen**: In einer verteilten Steuerung kann `GET_SINT` genutzt werden, um den Wert einer globalen oder peripheren Variable in eine lokale Verarbeitungskette zu übernehmen.
- **Datenkonsistenz**: Durch die Pufferung wird sichergestellt, dass der gelesene Wert während der gesamten Verarbeitung des nachfolgenden Netzwerks stabil bleibt, auch wenn die Quelle asynchron aktualisiert wird.

Vergleich mit ähnlichen Bausteinen

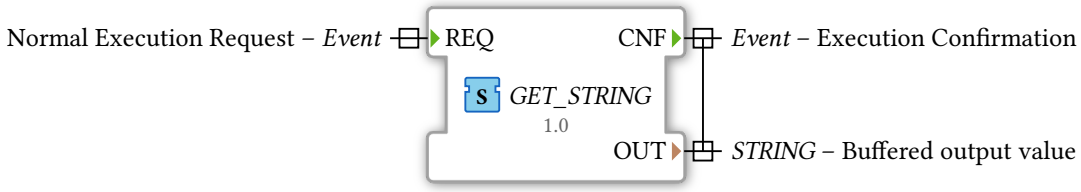
Baustein	Datentyp	Besonderheit
`GET_BOOL`	BOOL	Gleiche Funktionsweise, aber für boolesche Werte.
`GET_INT`	INT	Entspricht `GET_SINT`, jedoch mit 16-Bit-Ganzzahl (Integer).
`GET_REAL`	REAL	Für Gleitkommazahlen.
`MOVE` (generic)	Beliebig	Ein generischer MOVE-Baustein kann ebenfalls Werte kopieren, benötigt aber separate Ein- und Ausgänge. `GET_SINT` nutzt dagegen InOut-Variablen, was bei der Kopplung an periphere Adressen praktischer sein kann.

Allen `GET_*`-Bausteinen ist gemeinsam, dass sie einen Wert aus einer InOut-Quelle lesen und als gepufferten Ausgang bereitstellen. Im Gegensatz dazu überschreibt `SET_*` den Wert der Quelle.

Fazit

`GET_SINT` ist ein minimalistischer und zuverlässiger Baustein zum Auslesen eines `SINT`-Wertes aus einer InOut-Variable. Seine Einfachheit macht ihn zu einem idealen Baustein für das Lesen von peripheren oder geteilten Daten in IEC 61499-Anwendungen. Die Verwendung der InOut-Schnittstelle reduziert den Verdrahtungsaufwand und fördert eine saubere Trennung zwischen Datenzugriff und Verarbeitungslogik.

GET_STRING



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `GET_STRING` dient dazu, einen Stringwert aus einer InOut-Variable auszulesen und als gepufferten Ausgang bereitzustellen. Er wird typischerweise verwendet, um den aktuellen Wert einer externen Variablen zu einem bestimmten Zeitpunkt zu erfassen und für die Weiterverarbeitung bereitzustellen.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Datentyp	Kommentar
REQ	Event	Normaler Ausführungsanforderung. Löst das Einlesen der InOut-Variablen aus.

Ereignis-Ausgänge

Ereignis	Datentyp	Kommentar
CNF	Event	Bestätigung der Ausführung. Wird gesendet, nachdem der Wert gelesen und an den Ausgang übergeben wurde.

Daten-Eingänge

Name	Datentyp	Kommentar
IN	STRING	InOut-Variable, die als Quelle dient. Ihr aktueller Wert wird bei REQ gelesen und gepuffert.

Daten-Ausgänge

Name	Datentyp	Kommentar
OUT	STRING	Gepufferter Ausgabewert. Enthält den zum Zeitpunkt des letzten REQ gelesenen String von IN.

Adapter

Keine.

Funktionsweise

Der Funktionsblock verfügt über einen Zustand `REQ`. Ein eingehendes Ereignis an `REQ` führt den Algorithmus aus, der den Wert der InOut-Variablen `IN` direkt in den Ausgang `OUT` kopiert. Anschließend wird das Ereignis `CNF` gesendet. Die InOut-Variable `IN` wird dabei nicht verändert, sondern nur gelesen. Der gepufferte Wert bleibt solange erhalten, bis ein weiterer `REQ` einen neuen Wert einliest.

Technische Besonderheiten

- Die Variable `IN` ist als **InOut** deklariert, was bedeutet, dass sie als Referenz übergeben wird (Pass-by-Reference). Dies ermöglicht einen direkten Zugriff auf den Speicher der übergebenen Stringvariablen, ohne den Wert vorher in eine lokale Kopie zu übernehmen.
- Der Baustein ist auf den Datentyp `STRING` spezialisiert. Für andere Datentypen sind entsprechende Varianten (z. B. `GET_ANY`) erforderlich.
- Es werden keine initialen Werte gesetzt. Der Ausgang `OUT` ist nach dem ersten Zyklus undefiniert, solange kein `REQ` empfangen wurde.

Zustandsübersicht

Der Baustein besitzt einen einzigen Zustand:

Zustandsname	Beschreibung
REQ	Wartet auf ein Ereignis an `REQ`. Führt den Algorithmus aus (kopiert `IN` ⟶ `OUT`) und sendet `CNF`.

Es gibt keine Start- oder Endzustände; der Baustein verharrt nach jedem Durchlauf wieder im Zustand `REQ`.

Anwendungsszenarien

- **Zwischenspeicherung**: Ein Stringwert einer externen Quelle (z. B. eines Datenbausteins) wird abgetastet und für eine nachfolgende Verarbeitung zwischengespeichert, ohne die Quelle zu beeinflussen.
- **Protokollierung**: Momentaufnahme eines sich ändernden Strings zu bestimmten Ereigniszeitpunkten.
- **Synchronisation**: Bereitstellung eines konsistenten Wertes für mehrere nachgeschaltete Bausteine, wenn die Quelle von mehreren Stellen gleichzeitig gelesen werden soll.

Vergleich mit ähnlichen Bausteinen

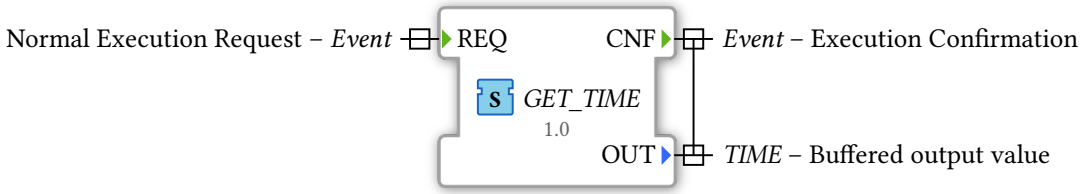
- **MOVE / ASSIGN**: Einfache Zuweisungsbausteine für beliebige Typen, oft ohne explizite Ereignissteuerung. `GET_STRING` ist dagegen speziell auf `STRING` ausgelegt und nutzt eine InOut-Schnittstelle.
- **GET_ANY**: Ein generischer Baustein für beliebige Datentypen. `GET_STRING` ist typsicher und vermeidet Parameterisierungsaufwand für das Einlesen von Strings.

- **READ_REF**: Ebenfalls ein Baustein zur Referenzübergabe, aber häufig als Funktionsbaustein ohne Ereignissteuerung realisiert. `GET_STRING` integriert die Ereignissteuerung und eignet sich daher für ablauforientierte Steuerungen.

Fazit

`GET_STRING` ist ein einfacher, aber spezialisierter Funktionsbaustein für die gepufferte Übernahme eines Stringwerts aus einer InOut-Variablen. Durch die Ereignissteuerung lässt er sich präzise in zeitkritischen Abläufen einsetzen. Die Verwendung einer InOut-Schnittstelle spart Speicher und Rechenzeit, da keine Wertkopie auf der Aufruferseite nötig ist. Für die meisten Anwendungen, in denen Strings abgetastet werden müssen, stellt dieser Baustein eine saubere und ressourcenschonende Lösung dar.

GET_TIME



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **GET_TIME** ermöglicht das Auslesen eines TIME-Werts aus einer InOut-Variable und stellt diesen als gepufferten Ausgangswert zur Verfügung. Er dient als einfache Schnittstelle, um einen externen Zeitwert in die Bausteinstruktur zu übernehmen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar	Mit Variablen
REQ	Event	Normaler Ausführungsanforderung	IN

Ereignis-Ausgänge

Name	Typ	Kommentar	Mit Variablen
CNF	Event	Bestätigung der Ausführung	OUT, IN

Daten-Eingänge

Keine dedizierten Daten-Eingänge vorhanden.

Daten-Ausgänge

Name	Typ	Kommentar	Initialwert
OUT	TIME	Gepufferter Ausgangswert	T#0s

Adapter

Keine Adapter vorhanden.

Funktionsweise

GET_TIME implementiert einen einfachen, ereignisgesteuerten Ablauf. Ein Ereignis am Eingang **REQ** löst die Ausführung des Algorithmus aus. Dabei wird der aktuelle Wert der InOut-Variable **IN** in den Datenausgang **OUT** kopiert und zwischengespeichert. Anschließend wird ein Ereignis am Ausgang **CNF** erzeugt, das die erfolgreiche Übernahme quittiert. Die InOut-Variable **IN** dient dabei ausschließlich als lesbare Quelle und behält ihren Wert bei.

Technische Besonderheiten

- Die Variable **IN** ist als **InOut** deklariert. Sie kann von außen mit einer anderen Variablen oder einem anderen Ausgang verbunden werden, ohne dass der Baustein Schreibzugriff darauf nimmt. Dies erlaubt eine flexible Anbindung an externe Zeitgeber oder Signale.
- Der Ausgang **OUT** puffert den gelesenen Wert. Nach einem **REQ**-Ereignis steht der aktuelle **IN**-Wert bis zur nächsten Anforderung am Ausgang zur Verfügung.
- Der Baustein besitzt keinen eigenen Zeitgeber oder Puffer für Verzögerungen.

Zustandsübersicht

Zustand	Beschreibung	Ausgeführte Aktion
REQ	Wartet auf ein Ereignis am Eingang **REQ**. Nach Eintritt wird der Algorithmus ausgeführt und ein **CNF**-Ereignis gesendet.	`OUT := IN;`

Anwendungsszenarien

- **Zeitmessung**: Übernahme einer aktuellen Systemzeit (z. B. `TIME()` oder eines Taktgebers) in einen Ausgang für weitere Verarbeitung.
- **Signalweiterleitung**: Gepuffertes Weiterleiten eines externen TIME-Signals, um asynchrone Änderungen der Quelle vom restlichen Ablauf zu entkoppeln.
- **Initialisierung**: Setzen eines Startzeitwerts in einer Steuerungslogik.

Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem reinen **MOVE**-Baustein für TIME-Daten verwendet **GET_TIME** ein **InOut**-Interface, das eine direkte Verbindung zur Quelle auch ohne zusätzlichen Daten-Eingang ermöglicht. Es ist speziell auf TIME-Werte zugeschnitten und bietet keine allgemeine Datenkopierfunktion. Andere Zeitbausteine wie **E_TIME** oder **TIME_IN** arbeiten oft mit eigenem Zeitgeber oder Ereigniszählung, während **GET_TIME** passiv einen externen Wert abruft.

Fazit

GET_TIME ist ein schlanker, übersichtlicher Funktionsbaustein zum Auslesen und Puffern eines TIME-Werts. Seine einfache Ereignissteuerung und die Verwendung einer InOut-Variable machen ihn besonders geeignet für Szenarien, in denen ein externer Zeitwert punktuell in eine Steuerungslogik übernommen werden soll.

GET_TIME_OF_DAY

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `GET_TIME_OF_DAY` dient dem Auslesen eines Tageszeitwertes (Typ `TIME_OF_DAY`) aus einer InOut-Variablen. Der gelesene Wert wird intern gepuffert und über einen Datenausgang bereitgestellt. Auf diese Weise wird eine konsistente, einmalig gelesene Tageszeit weitergereicht – auch wenn sich die Quellvariable zwischenzeitlich ändert.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Datentyp	Kommentar
REQ	Event	Ausführungsanforderung; liest den aktuellen Wert von IN

Ereignis-Ausgänge

Ereignis	Datentyp	Kommentar
CNF	Event	Bestätigung nach erfolgreichem Lesen und Puffern

Daten-Eingänge

Es sind keine reinen Eingänge vorhanden. Die Quelldaten werden über den InOut-Eingang `IN` bereitgestellt.

Daten-Ausgänge

Ausgang	Datentyp	Kommentar
OUT	TIME_OF_DAY	Gepufferter Tageszeitwert

Adapter

Keine Adapter vorhanden.

Funktionsweise

Ein Aufruf des Ereignisses `REQ` löst die interne assoziierte Aktion aus: Der aktuelle Wert der InOut-Variable `IN` wird gelesen und in die Ausgangsvariable `OUT` kopiert. Anschließend wird das Ereignis `CNF` gesendet. Die Tageszeit steht dann bis zum nächsten `REQ` konsistent am Ausgang `OUT` zur Verfügung – unabhängig von zwischenzeitlichen Änderungen an `IN`.

Technische Besonderheiten

- **InOut-Variable als Quelle:** Der FB nutzt einen InOut-Parameter, der im IEC-61499-Modell direkt mit einer externen Variablen verbunden ist. Dadurch kann der Baustein den Wert „von außen“ lesen, ohne selbst einen Daten-Eingang definieren zu müssen.
- **Pufferung:** Der gelesene Wert wird in der Ausgangsvariable `OUT` zwischengespeichert, sodass nachfolgende Abfragen immer denselben Wert liefern, bis erneut `REQ` ausgelöst wird.
- **Einfacher Zustandsautomat:** Der FB besteht aus nur einem Zustand, der die Aktion ausführt und sofort das Bestätigungsereignis ausgibt.

Zustandsübersicht

Der einfache Funktionsblock besitzt nur den Zustand `REQ`. Sobald das Ereignis `REQ` empfangen wird, wird die Aktion `REQ` ausgeführt (OUT := IN) und das Ereignis `CNF` wird gesendet. Es gibt keine weiteren Zustände oder Verzweigungen.

Anwendungsszenarien

- **Zeitstempel in Steuerungen:** Einmalige Erfassung eines Tageszeitwertes aus einem übergeordneten Taktgeber, um z. B. Betriebsstunden oder Schaltzeitpunkte dokumentieren zu können.
- **Pufferung für asynchrone Auswertung:** Ein externer Zeitgeber (z. B. eine Systemuhr) ändert sich laufend; der FB friert seinen Wert zum Zeitpunkt des Aufrufs ein und stellt ihn einem langsameren Verbraucher oder einer Logik bereit.
- **Kommunikation über Grenzen hinweg:** In verteilten Systemen kann die aktuelle Tageszeit von einer Komponente abgerufen und lokal zwischengespeichert werden, um sie später konsistent auszugeben.

Vergleich mit ähnlichen Bausteinen

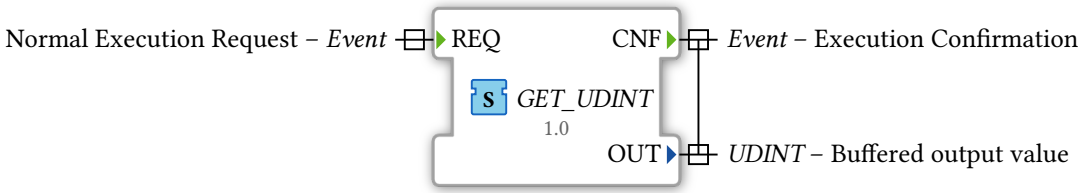
- **Im Gegensatz zu einem reinen Datenverbinder** (z. B. MOVE oder einfache Datenkopie) verwendet `GET_TIME_OF_DAY` einen InOut-Parameter, der eine direkte Kopplung an die Quellvariable erlaubt und so die Signalwege im IEC-61499-Modell sauber trennt.
- **Im Unterschied zu Zeit-Funktionsbausteinen**, die selbstständig Ticks oder Echtzeitwerte generieren, greift dieser FB nur auf einen extern bereitgestellten Wert zu – er ist reine Lese- und Pufferlogik.
- **Ähnliche Bausteine** existieren für andere Datentypen (z. B. `GET_BOOL`, `GET_REAL`) und folgen dem gleichen Entwurfsmuster.

Fazit

`GET_TIME_OF_DAY` ist ein kompakter, aber nützlicher Funktionsblock, der eine Tageszeit aus einer InOut-Variable ausliest, puffert und kontrolliert als Ausgangswert weitergibt. Seine klare, zustandsarme

Logik und die Pufferung machen ihn besonders geeignet für Aufgaben, bei denen eine einmal erfasste Zeit zu einem späteren Zeitpunkt noch in identischer Form vorliegen muss.

GET_UDINT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **GET_UDINT** dient dazu, den aktuellen Wert einer als InOut deklarierten Variable vom Typ **UDINT** zu lesen und als gepufferten Ausgang bereitzustellen. Die Pufferung stellt sicher, dass der Ausgangswert während eines Zyklus stabil bleibt, auch wenn die Quelldaten sich später ändern.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
REQ	Event	Normaler Ausführungsanforderung

Assoziierte Daten:

- **IN** (InOut-Variable) wird bei REQ gelesen.

Ereignis-Ausgänge

Name	Typ	Kommentar
CNF	Event	Bestätigung der Ausführung

Assoziierte Daten:

- **OUT** (Ausgangswert) wird nach erfolgreichem Lesen bereitgestellt.
- **IN** (InOut-Variable) wird ebenfalls assoziiert, d. h. ihr Wert kann nach dem Event am Ausgang gelesen werden.

Daten-Eingänge

Name	Typ	Vorgabewert	Kommentar
IN	UDINT (InOut)	0	Quellvariable, die gelesen wird

Daten-Ausgänge

Name	Typ	Vorgabewert	Kommentar
OUT	UDINT	0	Gepufferter Ausgangswert

Adapter

Keine.

Funktionsweise

Bei einem Ereignis am **REQ**-Eingang führt der Baustein den Algorithmus aus und sendet ein Bestätigungsereignis auf **CNF**. Der Algorithmus kopiert den aktuellen Wert der InOut-Variablen **IN** in die Ausgangsvariable **OUT**. Dadurch wird der Wert zum Zeitpunkt der Anforderung eingefroren und steht bis zur nächsten Anforderung stabil am Ausgang zur Verfügung.

Technische Besonderheiten

- **InOut-Variable:** Der Parameter **IN** ist als InOut deklariert. Dies ermöglicht einen direkten Zugriff auf die Speicheradresse der Quellvariablen, was bei der Kommunikation über verteilte Systeme (z. B. Feldbusse) nützlich sein kann. Der FB liest jedoch nur den Wert, er schreibt nicht zurück.
- **Pufferung:** Der Ausgang **OUT** wird nur bei einem **REQ**-Ereignis aktualisiert. Änderungen an **IN** zwischen zwei Anforderungen haben keinen Einfluss auf **OUT**.
- **Keine Seiteneffekte:** Der Baustein verändert die InOut-Variable nicht.

Zustandsübersicht

Der FB besitzt einen einzigen Zustand:

- **REQ:** Der Algorithmus wird ausgeführt. Nach Kopieren des Wertes wird sofort der Ausgang **CNF** gesetzt.

Es gibt keine weiteren Zustände, d. h. der FB ist immer bereit, eine Anforderung zu bearbeiten.

Anwendungsszenarien

- **Pufferung von Prozesswerten:** In Automatisierungssystemen, in denen ein Messwert oder eine Stellgröße (z. B. ein Zählerstand oder eine Geschwindigkeit) in einem bestimmten Takt eingefroren werden muss, bevor er an andere Bausteine weitergegeben wird.
- **Entkopplung von Zyklen:** Wenn eine InOut-Variable von mehreren Bausteinen gelesen wird und ein stabiler Wert innerhalb eines Zyklus benötigt wird.
- **Ersatz für direkten Durchgriff:** Statt einer InOut-Variable direkt auszulesen, bietet der FB eine klare Schnittstelle (Event-Eingang, Ausgang) und verbessert die Lesbarkeit des Systems.

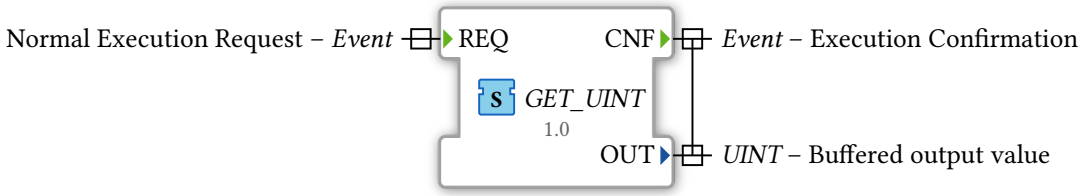
Vergleich mit ähnlichen Bausteinen

- **MUX (Multiplexer):** Ein MUX wählt zwischen mehreren Eingängen aus, während **GET_UDINT** genau einen definierten Wert von einer InOut-Variablen puffert.
- **GET_DINT / GET_REAL:** Analoge Bausteine für andere Datentypen, die das gleiche Prinzip der gepufferten Ausgabe von InOut-Werten umsetzen.
- **Latch-Bausteine:** Diese speichern Werte und geben sie bei einem Ereignis aus; **GET_UDINT** hingegen überschreibt den Ausgang bei jeder Anforderung mit dem aktuellen Wert der Quelle.

Fazit

Der Funktionsblock **GET_UDINT** ist ein einfaches, aber nützliches Hilfsmittel, um den Wert einer InOut-Variable vom Typ UDINT stabil und ereignisgesteuert zwischenzuspeichern. Er eignet sich besonders für Anwendungen, in denen ein Zugriff auf externe oder geteilte Variablen in kontrollierten Schritten erfolgen soll, und trägt so zu einer sauberen und vorhersagbaren Signalverarbeitung bei.

GET_UINT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **GET_UINT** liest den aktuellen Wert einer als InOut-Variable übergebenen `UINT`-Quelle aus und stellt diesen als gepufferten Ausgangswert bereit. Er dient dazu, einen konsistenten Schnappschuss eines externen oder internen UINT-Werts zu einem definierten Zeitpunkt zu erfassen und für die weitere Verarbeitung im Ablauf bereitzustellen.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung
REQ	Normaler Ausführungsauftrag; löst das Lesen der InOut-Variablen und die Aktualisierung des Ausgangs aus.

Ereignis-Ausgänge

Ereignis	Beschreibung
CNF	Bestätigt die erfolgreiche Ausführung von REQ und signalisiert, dass die Ausgänge gültig sind.

Daten-Eingänge

Variable	Typ	Beschreibung
IN	UINT	Quellvariable, die als InOut-Parameter bereitgestellt wird. Der Wert wird bei REQ ausgelesen und unverändert an den Ausgang übergeben. Standardwert: `0`.

Hinweis: `IN` ist als InOut-Variable deklariert, wird innerhalb des Bausteins jedoch nur lesend verwendet.

Daten-Ausgänge

Variable	Typ	Beschreibung
OUT	UINT	Gepufferter Ausgangswert, der den bei der letzten REQ-Ausführung gelesenen Wert von `IN` enthält. Standardwert: `0`.

Adapter

Keine.

Funktionsweise

Der Baustein verfügt über einen einzigen Ausführungszustand (`REQ`). Wird der Ereigniseingang `REQ` aktiviert, führt er folgende Aktion aus:

1. **Lesen** des aktuellen Werts der InOut-Variablen `IN`.
2. **Kopieren** dieses Werts in die Ausgangsvariable `OUT`.
3. **Auslösen** des Ereignisausgangs `CNF`, um die Fertigstellung zu signalisieren.

Die Verarbeitung erfolgt strikt sequenziell und ohne Verzögerung.

Technische Besonderheiten

- **InOut-Variable als Quelle:** Obwohl `IN` als InOut deklariert ist, wird sie ausschließlich lesend verwendet. Der Baustein schreibt nicht zurück, sodass die Quelle unverändert bleibt.
- **Pufferung:** Der Ausgang `OUT` hält den Wert, bis der nächste `REQ` ausgeführt wird. Dadurch wird eine Entkopplung zwischen der Quelle und der nachfolgenden Logik erreicht.
- **Einfaches Zustandsmodell:** Der Baustein besteht nur aus einem Zustand, sodass keine komplexen Ablaufsteuerungen erforderlich sind.

Zustandsübersicht

Zustand	Beschreibung
REQ	Wartet auf das Eintreffen des Ereignisses `REQ` und führt dann die Zuweisung `OUT := IN` durch. Anschließend wird `CNF` gesendet.

Es gibt keinen expliziten Start- oder Endzustand; nach der Ausführung bleibt der Baustein im Zustand `REQ` bereit für den nächsten Auftrag.

Anwendungsszenarien

- **Sichere Wertübernahme:** Ein UINT-Wert aus einer globalen Variablen oder einem anderen Baustein soll zu einem definierten Zeitpunkt eingefroren werden, um während eines Ablaufs konsistent zu bleiben.
- **Abtastung von Messwerten:** In einer zyklischen Steuerung wird der aktuelle Wert eines Zählers oder Sensors ausgelesen und zwischengespeichert.
- **Datenübergabe zwischen Tasks:** Der Baustein kann als einfache Pufferschicht zwischen asynchronen Kommunikationspfaden dienen.

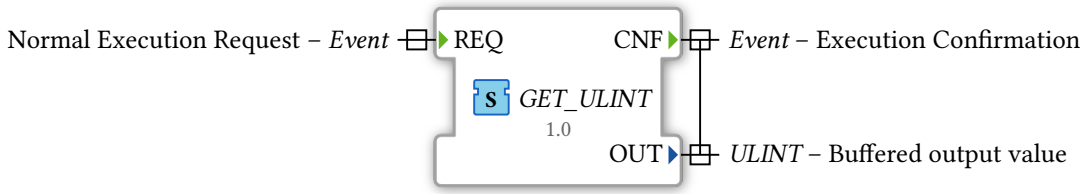
Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu einem reinen Eingangsbaustein, der einen festen Datenport verwendet, erlaubt **GET_UINT** die Nutzung einer **InOut-Variablen** als Quelle. Dadurch kann die Variable sowohl von außen beschrieben als auch gelesen werden, ohne eine separate Verbindung zu einem Eingang zu benötigen. Reine Lesebausteine (z. B. UINT-Eingang) bieten keine solcher Flexibilität, sind aber klarer in der Schnittstelle. Gegenüber einem Speicherbaustein (z. B. `R_TRIG` oder `SR`) speichert **GET_UINT** keine Zustände, sondern lediglich den letzten gelesenen Wert.

Fazit

GET_UINT ist ein einfacher, aber nützlicher Funktionsblock, der einen UINT-Wert aus einer InOut-Variablen puffert. Sein minimalistisches Design – ein Ereignis, eine Aktion – macht ihn ideal für Situationen, in denen eine saubere, zeitlich definierte Wertübernahme erforderlich ist. Er eignet sich besonders für Anwendungen, die eine konsistente Datenbasis innerhalb eines Ablaufs benötigen, ohne die Quelle zu beeinflussen.

GET_ULINT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **GET_ULINT** dient dazu, den aktuellen Wert einer als InOut-Parameter bereitgestellten Variable vom Typ `ULINT` (unsigned 64-Bit-Integer) zu lesen und an einem Datenausgang als gepufferten Wert bereitzustellen. Auf eine Anfrage (REQ) wird der Wert der Quelle eingelesen und am Ausgang `OUT` ausgegeben. Anschließend wird die Bestätigung (CNF) gesendet. Damit ist GET_ULINT ein einfacher Lese-Baustein für InOut-Variablen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar	Mit Daten
`REQ`	Event	Normal Execution Request	`IN`

Ereignis-Ausgänge

Name	Typ	Kommentar	Mit Daten
`CNF`	Event	Execution Confirmation	`OUT`, `IN`

Daten-Eingänge

Name	Typ	Kommentar

-	-	-

Daten-Ausgänge

Name	Typ	Kommentar	Initialwert
OUT	ULINT	Buffered output value	0

Adapter

Keine Adapter vorhanden.

InOut-Variablen (intern)

Zusätzlich zu den reinen Ein-/Ausgängen besitzt der Baustein eine InOut-Variable:

Name	Typ	Kommentar	Initialwert
IN	ULINT	Source variable (InOut)	0

Funktionsweise

Der Baustein verarbeitet eine einzige Aktion:

1. Ein eingehendes Ereignis am Eingang REQ löst die Ausführung des Algorithmus REQ aus.
2. Im Algorithmus wird der aktuelle Wert der InOut-Variablen IN direkt in den Ausgang OUT kopiert: `OUT := IN;`
3. Nach Abschluss der Zuweisung wird das Ereignis CNF ausgegeben. Die Ausgänge OUT und IN werden mit dem Ereignis CNF assoziiert, sodass aufrufende Bausteine den aktualisierten Wert verarbeiten können.

Durch die Verwendung einer InOut-Variablen kann der Baustein auf einen Speicherplatz außerhalb des eigenen Kontexts zugreifen, ohne diesen über einen herkömmlichen Eingang zu erhalten. Der Wert wird bei jeder Anfrage frisch gelesen, wodurch sich der Baustein als Lese-Puffer eignet.

Technische Besonderheiten

- **InOut-Variable:** Der Baustein nutzt eine IN-Variable mit der Eigenschaft InOut. Dies ermöglicht das Verknüpfen mit einer übergeordneten Variablen (z.B. aus einem Ressourcen- oder globalen Datenraum), ohne dass der Baustein die Variable über einen normalen Daten-Eingang erhalten muss. Der Zugriff erfolgt referenzbasiert.
- **Typ ULINT:** Sowohl IN als auch OUT sind als vorzeichenloser 64-Bit-Integer (ULINT) deklariert – geeignet für große Zählwerte, Zeitstempel oder andere Werte im Bereich 0 … 2^{64}-1.
- **Sofortiges Lesen:** Der Algorithmus ist extrem kurz (eine Zuweisung), wodurch die Ausführungszeit vernachlässigbar ist.

Zustandsübersicht

Der Baustein ist als SimpleFB mit einem einzigen ECC-Zustand implementiert:

Zustand	Aktion	Ausgabeereignis	Bedeutung
REQ	REQ (OUT := IN)	CNF	Auf eine Anfrage wird der Wert gelesen und bestätigt.

Es gibt keine weiteren Zustände, keine Verzweigungen und keine internen Speichervariablen (außer dem Ausgang OUT selbst, der den letzten gelesenen Wert puffert).

Anwendungsszenarien

- **Sichere Leseoperation:** Ein übergeordneter Baustein möchte einen aktuellen Wert aus einer gemeinsam genutzten Variablen (z.B. einem Sensorwert, der von einer anderen Steuerungskomponente beschrieben wird) abfragen, ohne den Wert dauerhaft zu referenzieren.
- **Wertpuffer für asynchrone Verarbeitung:** In einem zeitgesteuerten Ablauf kann GET_ULINT aufgerufen werden, um einen Schnappschuss einer globalen Zählervariablen zu nehmen und diesen zwischenzuspeichern.

- **Parametrierung über InOut-Schnittstellen:** Wenn der Baustein in eine übergeordnete Komponente eingebettet ist, kann die InOut-Variable direkt mit einer Datenverbindung des Elternbausteins verbunden werden, sodass Leseanfragen auf diese Variable zeigen.

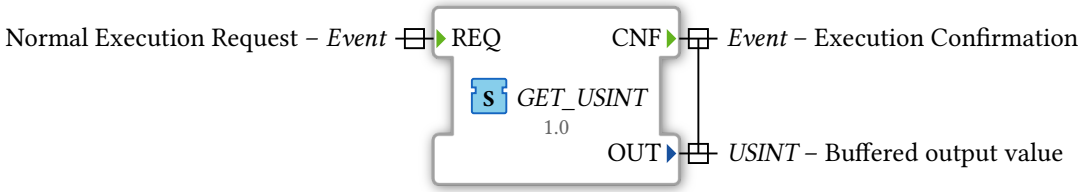
Vergleich mit ähnlichen Bausteinen

- **GET_DINT / GET_LINT / GET_UINT:** Analoge Bausteine für andere Integer-Typen (DINT, LINT, UINT etc.). Die Funktionsweise ist identisch, der einzige Unterschied ist der Datentyp.
- **READ_ULINT (hypothetisch):** Ein Baustein, der statt einer InOut-Variable einen separaten Dateneingang für den Quellwert verwendet. Das hätte den Vorteil, dass der Baustein lose gekoppelt ist, aber den Nachteil, dass der Quellwert als Datenstrom mitgeführt werden müsste. Der InOut-Ansatz von `GET_ULINT` erlaubt eine direkte Referenz auf eine gemeinsam genutzte Variable, was in manchen Architekturen bevorzugt wird.
- **SAMPLE_ULINT:** Ein Baustein, der bei jedem Ereignis den aktuellen Wert eines Eingangs abtastet und puffert – ebenfalls ähnlich, jedoch mit normalem Dateneingang statt InOut.

Fazit

Der `GET_ULINT`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, den aktuellen Wert einer InOut-Variablen vom Typ `ULINT` zu lesen und als gepufferten Ausgang bereitzustellen. Die Verwendung einer InOut-Variable erlaubt den direkten, referenzbasierten Zugriff auf externe Speicherstellen, ohne zusätzliche Datenkopien während des Aufrufs. Dank seines minimalistischen Zustandsautomaten eignet er sich hervorragend für zeitkritische Anwendungen, in denen ein schnelles Lesen eines Wertes ohne Nebenwirkungen erforderlich ist.

GET_USINT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `GET_USINT` dient dem Auslesen eines `USINT`-Wertes aus einer InOut-Variable. Er puffert den gelesenen Wert und stellt ihn über einen Ausgang zur Verfügung. Der Baustein eignet sich insbesondere dann, wenn der Wert einer Variablen aus einem übergeordneten Kontext (z. B. einem Verbund) einmalig abgefragt und im weiteren Ablauf als konstanter Zwischenwert verwendet werden soll.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Datentyp	Kommentar
REQ	Event	Normaler Ausführungsauftrag. Auslöser, um den aktuellen Wert von `IN` zu lesen und an `OUT` weiterzugeben.

Ereignis-Ausgänge

Ereignis	Datentyp	Kommentar
CNF	Event	Bestätigung der erfolgreichen Ausführung. Wird nach dem Lesen und Puffern gesendet.

Daten-Eingänge

Der Baustein besitzt keine klassischen reinen Daten-Eingänge. Stattdessen wird der zu lesende Wert über die InOut-Variable `IN` bereitgestellt.

Variable	Datentyp	Kommentar
IN	USINT	Quellvariable (InOut). Der aktuelle Wert dieser Variablen wird bei einem REQ-Ereignis gelesen und zwischengespeichert. Die Variable kann sowohl von außen beschrieben als auch ausgelesen werden; innerhalb des Bausteins erfolgt ausschließlich ein lesender Zugriff.

Daten-Ausgänge

Variable	Datentyp	Kommentar
OUT	USINT	Gepufferter Ausgangswert. Enthält den bei der letzten Ausführung von IN gelesenen Wert. Initialwert: 0.

Adapter

Keine.

Funktionsweise

1. Der Baustein wartet auf ein REQ-Ereignis.
2. Beim Eintreffen von REQ wird der aktuelle Wert der InOut-Variable IN gelesen und in die lokale Ausgangsvariable OUT kopiert (`OUT := IN;`).
3. Anschließend wird das Bestätigungsereignis CNF gesendet.
4. Der Wert von OUT bleibt so lange erhalten, bis ein weiteres REQ-Ereignis einen neuen Lesevorgang auslöst.

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Verzweigungen oder zeitgesteuerten Abläufe – die Aktion wird in einem Schritt ausgeführt.

Technische Besonderheiten

- **InOut-Variable IN** – Die Schnittstelle verwendet eine InOut-Variable, die sowohl als Eingang als auch als Ausgang fungiert. Dies ist typisch für 4diac-Funktionsblöcke, die auf Variablen eines übergeordneten Verbunds zugreifen. Der Baustein ändert den Wert von IN **nicht**, er liest ihn lediglich.
- **Pufferung** – Ohne erneutes REQ bleibt der Ausgangswert stabil, selbst wenn sich die Quellvariable später ändert. Dies entkoppelt den Wert vom Zeitpunkt des Lesens.
- **Keine Zustandsautomaten** – Der Baustein besteht aus nur einem Zustand REQ, was die Implementierung trivial und ressourcenschonend macht.

Zustandsübersicht

Der Baustein besitzt genau einen Zustand:

Zustand	Aktion	Ausgangsereignis
REQ	Führe Algorithmus REQ aus (`OUT := IN;`)	CNF

Ein Startzustand ist nicht explizit definiert; nach dem Initialisieren wird der Baustein durch das erste REQ aktiv.

Anwendungsszenarien

- **Einmaliges Abfragen einer Konfigurationsvariablen** – Ein in einer übergeordneten Ressource definierter USINT-Wert soll genau einmal zu Beginn des Ablaufs gelesen und danach als lokale Konstante verwendet werden.
- **Entkopplung von lesendem und schreibendem Zugriff** – Wenn eine Variable sowohl von mehreren Bausteinen geschrieben als auch gelesen wird, verhindert GET_USINT eine ständige Aktualisierung der lesenden Seite, da nur auf Anforderung gepuffert wird.
- **Testhilfsmittel** – Der Baustein kann genutzt werden, um den aktuellen Wert einer InOut-Variable in einem Diagnosezweig sichtbar zu machen.

Vergleich mit ähnlichen Bausteinen

Baustein	Typ	Besonderheit

`GET_USINT`	Eigener FB	Liest eine InOut-Variable und puffert den Wert. Ausgang bleibt stabil bis zum nächsten REQ.
`USINT_TO_*` (z. B. Konvertierer)	Konvertierungs-FB	Wandelt einen USINT in einen anderen Typ um, liest aber nicht zwingend aus einem In-Out.
Direkter Datenzugriff	Kein FB	Ein Wert kann über Datenverbindungen ohne Puffer gelesen werden, jedoch ohne Entkopplung und ohne Ereignissteuerung.

`GET_USINT` ist spezialisiert auf den lesenden Zugriff auf InOut-Variablen mit expliziter Pufferung und Ereignissteuerung – eine Aufgabe, die mit Standard-Konvertern oder direkten Verbindungen so nicht abgebildet werden kann.

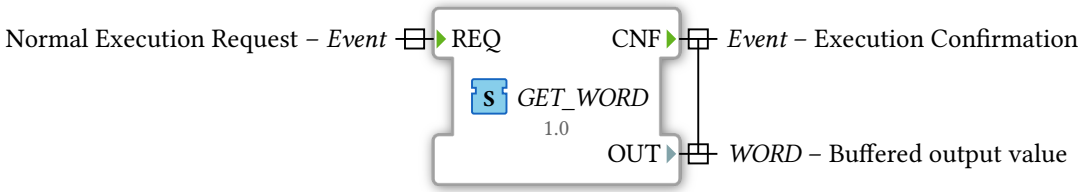
Fazit

Der `GET_USINT`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, einen `USINT`-Wert aus einer InOut-Variable auszulesen und als gepufferten Ausgang bereitzustellen. Seine klare, ereignisgesteuerte Schnittstelle und der minimale Zustandsautomat machen ihn zu einem nützlichen Werkzeug für Entkopplungs- und Initialisierungsaufgaben in 4diac-Anwendungen. Er ist ideal geeignet, wenn der Wert einer Variablen nur auf Anforderung abgerufen werden soll und zwischenzeitliche Änderungen der Quelle ignoriert werden müssen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

GET_WORD



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **GET_WORD** dient dem Auslesen eines WORD-Werts aus einer InOut-Variablen und stellt diesen als gepufferten Ausgang zur Verfügung. Er wird typischerweise eingesetzt, um einen aktuellen Wert von einer Hardware-peripherie oder einer gemeinsam genutzten Speicherstelle zu erfassen und für die weitere Verarbeitung im Steuerungsprogramm bereitzustellen.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung
REQ	Normale Ausführungsanforderung; löscht den Lesevorgang aus und gibt das Ergebnis am Ausgang `OUT` aus.

Ereignis-Ausgänge

Ereignis	Beschreibung
CNF	Bestätigung der Ausführung; signalisiert, dass der Wert von `IN` erfolgreich in `OUT` übernommen wurde.

Daten-Eingänge

Der FB besitzt keine expliziten Dateneingänge im klassischen Sinne, sondern nutzt eine **InOut-Variable**:

Variable	Typ	Beschreibung
IN	WORD	Quellvariable (InOut), aus der der gelesene Wert stammt. Der FB greift lesend darauf zu.

Daten-Ausgänge

Ausgang	Typ	Beschreibung
OUT	WORD	Gepufferter Ausgangswert – Kopie von `IN` zum Zeitpunkt der letzten `REQ`-Ausführung.

Adapter

Keine Adapter vorhanden.

Funktionsweise

1. Der FB startet mit einem **REQ**-Ereignis am Eingang.
2. Der Algorithmus **REQ** wird ausgeführt:
   - Der aktuelle Wert der InOut-Variablen `IN` wird in den Ausgang `OUT` kopiert.
3. Nach erfolgreicher Zuweisung wird das **CNF**-Ereignis am Ausgang gesendet.

Somit liegt der einmal gelesene Wert solange am Ausgang `OUT` an, bis ein erneuter **REQ**-Impuls eine Aktualisierung auslöst. Dies entspricht einer gepufferten Leseoperation.

Technische Besonderheiten

- **InOut-Verwendung** – Der Baustein greift lesend auf eine InOut-Variable zu, die typischerweise mit einem Speicher- oder Hardware-Register verbunden ist.
- **Pufferung** – Der gelesene Wert wird im Ausgang `OUT` gehalten, selbst wenn sich die Quelle `IN` später ändert. Erst ein erneuter **REQ**-Aufruf aktualisiert den Puffer.
- **Initialisierung** – `IN` und `OUT` sind standardmäßig mit 0 vorbelegt.

Zustandsübersicht

Der FB implementiert einen einfachen Zustandsautomaten mit nur einem Zustand:

Zustand	Beschreibung
REQ	Immer aktiv bei Eintreffen von **REQ** führt die Zuweisung `OUT := IN` durch und sendet **CNF**.

Es gibt keine weiteren Zustände wie IDLE oder WARTE, da die Logik streng ereignisgesteuert ohne Verzweigung ist.

Anwendungsszenarien

- **Einlesen eines Hardware-Registers** – Z. B. Auslesen eines Analog-Digital-Wandlerwerts oder eines Zählerstands, der als WORD vorliegt.
- **Sichern eines flüchtigen Werts** – Wenn die Quelle nur kurzzeitig stabil ist (z. B. nach einem Interrupt), kann der FB den Wert in einem definierten Moment einfrieren.
- **Zugriff auf gemeinsam genutzte Variablen** – In Mehrfachaufruf-Umgebungen wird der aktuelle Stand einer globalen WORD-Variablen abgeholt und lokal weiterverarbeitet.

Vergleich mit ähnlichen Bausteinen

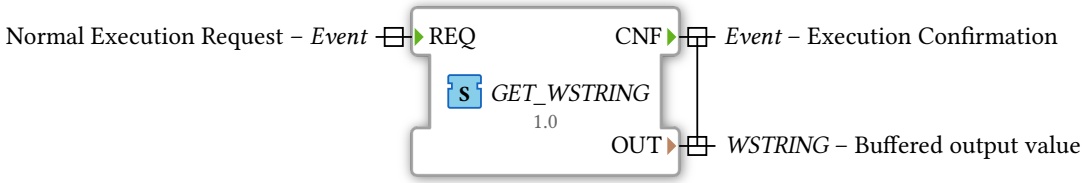
Baustein	Unterschied
MOVE	Kopiert einen Wert von einem Dateneingang zu einem Ausgang. `GET_WORD` liest dagegen aus einer InOut-Variablen, die oft eine Verbindung zu einer externen Ressource darstellt.
READ	Oft standardisiert für IEC 61499, liest aus einem Prozessabbild. `GET_WORD` ist spezifischer für eine gepufferte WORD-Übernahme aus einer InOut-Stelle.

LATCH	Hält einen Wert nach einem Setz-Impuls. `GET_WORD` aktualisiert nur bei jedem **REQ**, ähnelt also einem getakteten Latch, ist aber nicht setz-/rücksetzbar.

Fazit

GET_WORD ist ein einfacher, aber nützlicher Funktionsblock für das gezielte Auslesen und Puffern eines WORD-Werts aus einer InOut-Variablen. Er eignet sich besonders für Anwendungen, in denen zu einem bestimmten Ereigniszeitpunkt der aktuelle Stand einer Quelle festgehalten und anschließend stabil zur Verfügung gestellt werden muss. Der FB ist leicht verständlich, ressourcenschonend und lässt sich nahtlos in ereignisgesteuerte Abläufe integrieren.

GET_WSTRING



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **GET_WSTRING** dient dem Auslesen einer WSTRING-Variablen über einen InOut-Parameter und stellt den gelesenen Wert gepuffert als Datenausgang bereit. Er wird typischerweise eingesetzt, um einen externen WSTRING-Wert in die interne Verarbeitung eines Applikationsbausteins zu übernehmen, ohne den Ursprungswert zu verändern.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Datentyp	Beschreibung
REQ	Event	Normale Ausführungsanforderung – löst das Lesen der InOut-Variablen aus.

Ereignis-Ausgänge

Ereignis	Datentyp	Beschreibung
CNF	Event	Bestätigung der Ausführung – wird nach dem erfolgreichen Kopiervorgang gesendet.

Daten-Eingänge

Variable	Datentyp	Beschreibung
IN	WSTRING	InOut-Quellvariable, deren Wert ausgelesen wird. (Die Variable wird nicht verändert.)

Daten-Ausgänge

Variable	Datentyp	Beschreibung
OUT	WSTRING	Gepufferter Ausgabewert – enthält den zum Zeitpunkt des REQ-Ereignisses gelesenen Inhalt von IN.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Baustein verfügt über einen einzigen Echtzeitschritt (EC State) namens `REQ`. Sobald am Ereigniseingang **REQ** ein Impuls eintrifft, wird der hinterlegte Algorithmus ausgeführt:

1. Der aktuelle Wert der InOut-Variablen **IN** wird gelesen und direkt der Ausgangsvariablen **OUT** zugewiesen (`OUT := IN`).
2. Anschließend wird das Ereignis **CNF** ausgelöst, um dem aufrufenden Netzwerk die erfolgreiche Übernahme zu signalisieren.

Die InOut-Variable bleibt unverändert; es findet lediglich ein lesender Zugriff statt.

Technische Besonderheiten

- **InOut-Nutzung**: Die Variable `IN` wird formal als `InOut` deklariert. Dies erlaubt dem FB, auf die Speicheradresse der angeschlossenen Quelle zuzugreifen, ohne eine separate Eingangsvariable zu benötigen. Die Variable kann daher sowohl von außen beschrieben als auch innerhalb des FB gelesen werden.
- **Gepufferter Ausgang**: Der Wert von `OUT` bleibt stabil, bis der FB durch ein erneutes **REQ** aktualisiert wird. So kann der nachgeschaltete Teil der Applikation jederzeit auf einen konsistenten Wert zugreifen.
- **Typreinheit**: Der FB arbeitet ausschließlich mit dem Datentyp `WSTRING`. Für andere Datentypen sind eigene Bausteine (z. B. `GET_STRING`, `GET_DINT`) erforderlich.

Zustandsübersicht

Der FB besitzt nur einen aktiven Zustand:

Zustand	Beschreibung
REQ	Initial- und Arbeitszustand. Bei jedem **REQ**-Ereignis wird der Algorithmus ausgeführt und **CNF** gesendet. Es gibt keine weiteren Verzweigungen oder Wartezustände.

Anwendungsszenarien

- **Datenübernahme von einer externen WSTRING-Quelle** – z. B. Lesen einer globalen Konfigurationszeichenkette oder eines von einem anderen Baustein bereitgestellten WSTRING-Werts.
- **Entkopplung von Lese- und Schreibzugriffen** – Der Wert aus einer gemeinsam genutzten Variable kann sicher gepuffert werden, bevor er in der eigenen Logik weiterverarbeitet wird.
- **Realisierung eines „Get"-Funktionsblocks** in einer InOut-basierten Architektur, wie sie häufig in der **EC-61499**-Vernetzung verwendet wird.

Vergleich mit ähnlichen Bausteinen

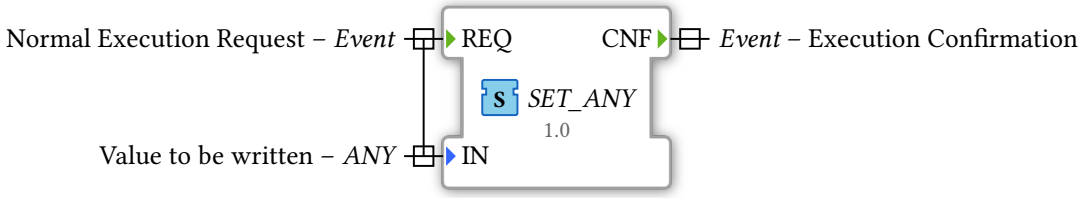
Baustein	Datentyp	Besonderheit
GET_STRING	STRING	Analoger FB für ASCII-Zeichenketten (nicht Unicode).
GET_DINT	DINT	Für 32-Bit-Ganzzahlen.
FORCE	Jeder Typ	Erzwingt einen Wert unabhängig von der Quelle, oft mit zusätzlichem Rücksetzmechanismus.

GET_WSTRING unterscheidet sich von diesen durch die spezifische Unterstützung des breiten Zeichentyps (WSTRING), der für Unicode-Zeichenfolgen (UTF-16) verwendet wird. Es handelt sich um einen einfachen, auf Lesen und Puffern beschränkten Baustein ohne Nebenwirkungen.

Fazit

Der **GET_WSTRING** ist ein schlanker, leicht verständlicher Funktionsblock, der in IEC-61499-Umgebungen eine grundlegende Aufgabe übernimmt: das sichere Auslesen einer WSTRING-InOut-Variablen und die Bereitstellung eines gepufferten Ausgangswerts. Seine einfache Struktur und der Verzicht auf Zustandsverzweigungen machen ihn ideal für den Einsatz in datenorientierten Steuerungsaufgaben, bei denen die Werte einer externen Quelle einmalig oder zyklisch übernommen werden müssen.

SET_ANY



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **SET_ANY** dient dazu, einen beliebigen Wert vom Eingang `IN` auf eine als InOut-Parameter deklarierte Variable `OUT` zu schreiben. Der Vorgang wird durch ein Ereignis am Eingang `REQ` ausgelöst und nach erfolgreicher Zuweisung mit dem Ausgangsereignis `CNF` quittiert. Der Baustein ist generisch und kann mit beliebigen Datentypen arbeiten, solange die Typen von `IN` und `OUT` kompatibel sind.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Datentyp	Kommentar
REQ	Event	Normaler Ausführungsanforderung – löst die Zuweisung aus. Verbunden mit den Variablen `IN` und `OUT`.

Ereignis-Ausgänge

Ereignis	Datentyp	Kommentar
CNF	Event	Bestätigung der erfolgreichen Ausführung. Verbunden mit der Variablen `OUT`.

Daten-Eingänge

Name	Datentyp	Kommentar
IN	ANY	Wert, der auf die Zielvariable geschrieben wird.
OUT	ANY	**InOut-Parameter** – wird gleichzeitig als Eingang und Ausgang verwendet (siehe Daten-Ausgänge).

Hinweis: `OUT` ist als InOut deklariert. Es fungiert als Daten-Eingang (die ursprüngliche Variable wird gelesen, falls nötig) und als Daten-Ausgang (überschrieben mit dem Wert von `IN`).

Daten-Ausgänge

Name	Datentyp	Kommentar
OUT	ANY	**InOut-Parameter** – enthält nach der Ausführung den Wert des Eingangs `IN`. Die Variable kann auch außerhalb des Bausteins verwendet werden.

Adapter

Keine.

Funktionsweise

Der Baustein besitzt einen einzigen verarbeitenden Zustand `REQ`. Tritt am Ereigniseingang `REQ` ein Ereignis auf, wird der Algorithmus `REQ` ausgeführt:

```
OUT := IN;
```

Nach der Zuweisung wird das Ausgangsereignis `CNF` gesendet. Die Ausführung ist atomar – es findet keine weitere Zustandslogik statt.

Technische Besonderheiten

- **Generischer Datentyp `ANY`:** Der Baustein kann mit beliebigen IEC 61499-Datentypen (z. B. BOOL, INT, REAL, STRING, Strukturen) verwendet werden. Die Typkompatibilität zwischen `IN` und `OUT` muss vom Anwender sichergestellt werden.
- **InOut-Mechanismus:** `OUT` ist als InOut-Parameter deklariert. Das bedeutet, dass die Variable sowohl als Eingabe (z. B. für Lesevorgänge) als auch als Ausgabe dient. In 4diac-IDE wird InOut typischerweise als Referenz auf eine externe Variable realisiert.
- **Einfache Zustandsmaschine:** Der FB enthält nur einen Zustand, wodurch er sehr leichtgewichtig und ressourcenschonend ist.

Zustandsübersicht

Zustand	Beschreibung
REQ	Wartet auf ein `REQ`-Ereignis. Bei Eintritt wird der Algorithmus ausgeführt, anschließend wird `CNF` gesendet und der Zustand bleibt aktiv.

Es gibt keine weiteren Zustände, keine Verzweigungen oder Zeitüberwachungen.

Anwendungsspezifische Szenarien

- **Initialisierung von Variablen:** Einmaliges Setzen eines Anfangswertes auf eine global oder instanzspezifisch verwendete Variable.
- **Parameterüberschreibung:** Zur Laufzeit wird ein bestehender Wert durch einen neuen überschrieben, z. B. zum Ändern von Sollwerten oder Konfigurationsparametern.
- **Punkt-zu-Punkt-Datentransfer:** Wenn ein Wert von einer Quelle auf ein direkt verbundenes Ziel kopiert werden soll, ohne dass eine separate Logik erforderlich ist.

Vergleich mit ähnlichen Bausteinen

Baustein	Unterschied
SET	Ist auf einen spezifischen Datentyp festgelegt (z. B. SET_BOOL, SET_INT). `SET_ANY` ist generisch.
MOVE	Kopiert ebenfalls einen Wert von einem Eingang auf einen Ausgang, verwendet jedoch normale Ausgänge. `SET_ANY` nutzt einen InOut-Parameter, der eine persistente Variable referenziert.
SELECT	Wählt zwischen zwei Eingangswerten aus und benötigt ein Steuersignal. `SET_ANY` arbeitet ohne Bedingung.

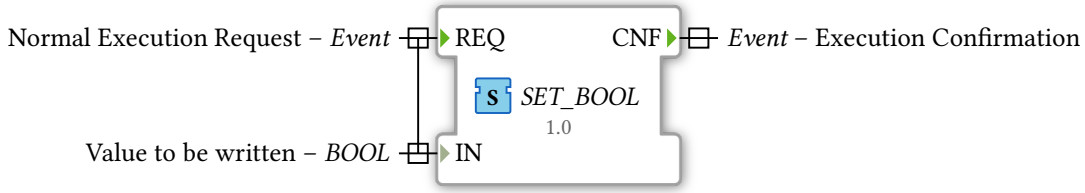
Fazit

`SET_ANY` ist ein einfacher, aber flexibler Funktionsblock zum gezielten Überschreiben von Variablen in IEC 61499. Die Verwendung des generischen Typs `ANY` und des InOut-Parameters macht ihn universell einsetzbar, insbesondere in Szenarien, in denen zur Laufzeit Werte in bestehende Speicherstellen geschrieben werden müssen. Dank seiner minimalen Zustandslogik eignet er sich für zeitkritische Anwendungen oder als Grundbaustein für komplexere Datenflussstrukturen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SET_BOOL



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `SET_BOOL` dient dazu, einen booleschen Eingangswert (`IN`) auf eine InOut-Variable (`OUT`) zu schreiben. Er stellt eine einfache Zuweisungslogik für BOOL-Daten bereit und wird über ein Ereignis gesteuert.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
REQ	Event	Normaler Ausführungsanstoß

Ereignis-Ausgänge

Name	Typ	Kommentar
CNF	Event	Bestätigung der Ausführung

Daten-Eingänge

Name	Typ	Initialwert	Kommentar
IN	BOOL	FALSE	Wert, der auf die Zielvariable geschrieben werden soll

Daten-Ausgänge

Keine separaten Datenausgänge. Der ausgegebene Wert wird über die InOut-Variable `OUT` bereitgestellt.

Adapter

Keine.

Funktionsweise

Wird das Ereignis `REQ` empfangen, führt der Baustein die Zuweisung `OUT := IN` aus. Dabei wird der aktuelle Wert des Eingangs `IN` auf die InOut-Variable `OUT` kopiert. Anschließend wird das Bestätigungsereignis `CNF` gesendet. Die InOut-Variable `OUT` kann von außen mit einer beliebigen BOOL-Variablen verbunden werden, sodass der Wert direkt an die entsprechende Stelle im System geschrieben wird.

Technische Besonderheiten

- Der Baustein verwendet eine **InOut-Variable** (`OUT`), die sowohl als Ausgang als auch als Eingang fungiert. Dadurch kann der Wert direkt in eine externe Variable geschrieben werden, ohne einen separaten Datenausgang zu benötigen.
- Die Initialwerte von `IN` und `OUT` sind jeweils `FALSE`, was bei fehlender Initialisierung einen definierten Zustand sicherstellt.
- Es gibt nur einen Algorithmus und einen Zustand, daher ist die Ausführungslogik deterministisch und nebenläufigkeitsfrei.

Zustandsübersicht

Zustand	Aktion	Ereignisausgang	Beschreibung

REQ	OUT := IN	CNF	Bei Ereignis REQ wird der Eingangswert auf die InOut-Variable geschrieben und die Ausführung bestätigt.

Der Baustein besitzt nur einen aktiven Zustand. Nach der Ausführung verharrt er wartend auf das nächste REQ-Ereignis.

Anwendungsszenarien

- **Setzen eines globalen Flags**: Über eine InOut-Variable kann ein boolesches Flag in einem anderen Baustein oder einer globalen Variable gesetzt werden.
- **Wertübernahme von einem Sensor**: Ein boolescher Sensorwert (z.B. Schalter) wird direkt in eine Steuervariable übernommen.
- **Initialisierung von BOOL-Variablen**: Bei Systemstart kann über REQ ein definierter Wert in eine Variable geschrieben werden.

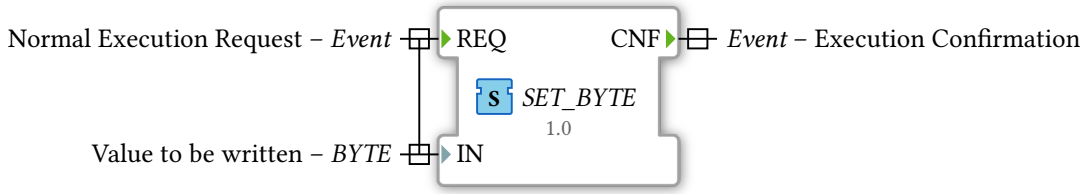
Vergleich mit ähnlichen Bausteinen

- **MOVE (z.B. ANY to ANY)**: MOVE kopiert Werte beliebiger Datentypen, benötigt aber getrennte Ein-/Ausgänge. SET_BOOL ist auf BOOL spezialisiert und nutzt eine InOut-Variable, was kompaktere Verbindungen ermöglicht.
- **SET_BOOL vs. direkte Zuweisung in ST**: Der Baustein abstrahiert die Zuweisung als ereignisgesteuerten Block und kann in grafischen Editoren (z.B. FBD) verwendet werden, wo eine direkte Zuweisung nicht möglich ist.
- **R_TRIG / F_TRIG**: Diese Bausteine detektieren Flanken und verändern den Ausgang nur bei Signalwechseln. SET_BOOL führt dagegen bei jedem REQ eine unbedingte Zuweisung durch.

Fazit

SET_BOOL ist ein einfacher, aber praktischer Funktionsblock zur direkten Zuweisung eines booleschen Werts auf eine Variable mittels einer InOut-Schnittstelle. Er eignet sich hervorragend für Steuerungsaufgaben, bei denen ein boolescher Zustand gezielt gesetzt werden muss, und fügt sich nahtlos in das ereignisgesteuerte Paradigma der IEC 61499 ein.

SET_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **SET_BYTE** dient dem einfachen Überschreiben einer BYTE-Variablen. Er liest den Wert des Dateneingangs IN und schreibt ihn in die InOut-Variable OUT, sobald ein Ereignis am Eingang REQ eintrifft. Anschließend wird das Ereignis CNF ausgegeben.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
REQ	Event	Normaler Ausführungsauftrag (Schreibvorgang starten)

Ereignis-Ausgänge

Name	Typ	Kommentar
CNF	Event	Bestätigung der erfolgten Zuweisung

Daten-Eingänge

Name	Typ	Wertebereich	Initialwert	Kommentar
IN	BYTE	0…255	0	Wert, der auf die Zielvariable geschrieben werden soll

Daten-Ausgänge

Der FB verwendet eine **InOut**-Variable. Daher gibt es keine separaten Datenausgänge – der ausgegebene Wert ist identisch mit dem Wert der InOut-Variablen OUT.

Adapter

Keine vorhanden.

Funktionsweise

Bei einem Ereignis am Eingang REQ wird der Algorithmus REQ ausgeführt. Dieser weist dem InOut-Variable OUT den Wert von IN zu:

```
OUT := IN;
```

Unmittelbar nach der Zuweisung wird das Ereignis CNF gesendet. Die InOut-Variable kann sowohl gelesen als auch von außerhalb des Bausteins (z. B. über eine Verbindung im Applikationsmodell) beschrieben werden – nach dem Schreibvorgang besitzt sie den Wert von IN.

Technische Besonderheiten

- **Einfachste Zuweisung**: Der Algorithmus besteht aus nur einer Zeile – keine Verzögerungen, keine Bedingungen oder Filter.
- **InOut-Variable**: OUT ist als InOut deklariert, d. h. sie kann von außen mit einer Variablen verbunden werden, die sowohl gelesen als auch beschrieben wird. Dies ermöglicht eine direkte Manipulation einer globalen oder verbundenen BYTE-Variablen.
- **Kein Zustandsautomat**: Der FB besitzt nur einen impliziten Zustand (REQ) und benötigt keine weiteren Zustandslogiken.

Zustandsübersicht

Der FB hat keine expliziten Zustände. Der Eintritt in die Aktion erfolgt sofort beim Ereignis REQ. Es gibt weder Fehlerzustände noch Verzögerungszustände.

Anwendungsszenarien

- **Initialisierung oder Rücksetzen**: Ein bestimmter Startwert soll in eine bestehende BYTE-Variable geschrieben werden.
- **Umschaltung/Bitmuster setzen**: Ein externes Signal (z. B. von einem Kommunikationsbaustein) wird über IN an einen Ausgang oder an eine Speicherstelle weitergegeben.
- **Test / Diagnose**: Temporäres Überschreiben eines Wertes in einer Steuerungsapplikation.

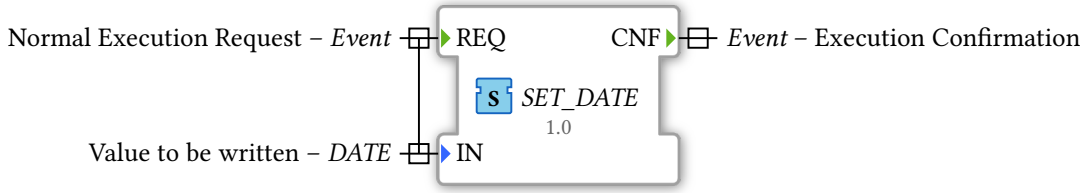
Vergleich mit ähnlichen Bausteinen

- **MOVE** (allgemeiner): Der einfache MOVE-Baustein für beliebige Datentypen arbeitet meist mit Eingang und Ausgang, nicht mit InOut. SET_BYTE ist speziell für BYTE optimiert und verwendet InOut, sodass der Wert direkt in die Zielvariable geschrieben wird (kein separater Ausgang nötig).
- **SET_BOOL** (bei Vorhandensein): Analog für BOOL – beide Bausteine folgen dem gleichen einfachen Prinzip, beschränken sich aber auf ihren jeweiligen Datentyp.

Fazit

SET_BYTE ist ein minimalistischer, klar definierter Funktionsblock zum einmaligen Beschreiben einer BYTE-Variablen. Seine Stärke liegt in der Einfachheit und der Verwendung einer InOut-Schnittstelle, die eine direkte Wertzuweisung ohne zusätzliche Verdrahtung eines Datenausgangs ermöglicht. Er eignet sich besonders für Anwendungen, in denen ein vorgegebener Wert zu einem bestimmten Zeitpunkt in eine persistente Variable geschrieben werden muss.

SET_DATE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **SET_DATE** dient dazu, einen Datumswert (Datentyp `DATE`) von einem Eingang in eine InOut-Variable zu schreiben. Die InOut-Variable wird dabei direkt beschrieben und steht nach der Ausführung mit dem neuen Wert zur Verfügung. Dies ermöglicht eine effiziente Übergabe von Datumswerten ohne zusätzliche Ausgangsvariable.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung	Mit Variablen
REQ	Normaler Ausführungsauftrag	IN, OUT

Ereignis-Ausgänge

Ereignis	Beschreibung	Mit Variablen
CNF	Bestätigung der Ausführung	OUT

Daten-Eingänge

Variable	Typ	Beschreibung
IN	DATE	Wert, der in die Zielvariable geschrieben werden soll
OUT	DATE	Zielvariable (InOut) – initialer Wert kann gelesen werden, wird aber überschrieben

Daten-Ausgänge

Variable	Typ	Beschreibung
OUT	DATE	Zielvariable (InOut) – enthält nach Ausführung den Wert von IN

Adapter

Keine.

Funktionsweise

Wird das Ereignis **REQ** ausgelöst, führt der Baustein den Algorithmus `REQ` aus:

- Der Wert des Eingangs `IN` wird der InOut-Variablen `OUT` zugewiesen (`OUT := IN;`).
- Anschließend wird das Ereignis **CNF** gesendet, um den erfolgreichen Abschluss zu signalisieren.

Die InOut-Variable `OUT` dient gleichzeitig als Eingang und Ausgang. Vor der Ausführung kann der aktuelle Wert gelesen werden, nach der Ausführung steht der neue Wert zur Verfügung.

Technische Besonderheiten

- **InOut-Variable:** Der Baustein verwendet eine InOut-Variable, die direkt in der aufrufenden Umgebung referenziert wird. Änderungen wirken unmittelbar auf die angeschlossene Variable, ohne zusätzlichen Datenausgang.
- **Datentyp:** Ausschließlich für den Datentyp `DATE` ausgelegt.

- **Einfache Zustandslogik:** Der Baustein besteht aus nur einem Zustand, der direkt die Zuweisung durchführt.

Zustandsübersicht

Zustand	Beschreibung
REQ	Führt die Wertzuweisung `OUT := IN` aus und sendet CNF

Es existiert kein weiterer Zustand; der Baustein ist sofort ausführungsbereit.

Anwendungsszenarien

- Setzen eines Datumswertes in einer SPS-Steuerung, z. B. Initialisierung eines aktuellen Datums.
- Übernahme eines berechneten oder übergeordneten Datums in eine globale oder übergeordnete Variable.
- Umschaltung zwischen verschiedenen Datumswerten durch wiederholte Aufrufe mit unterschiedlichen `IN`-Werten.

Vergleich mit ähnlichen Bausteinen

Ähnliche Funktionsbausteine existieren für andere Datentypen:

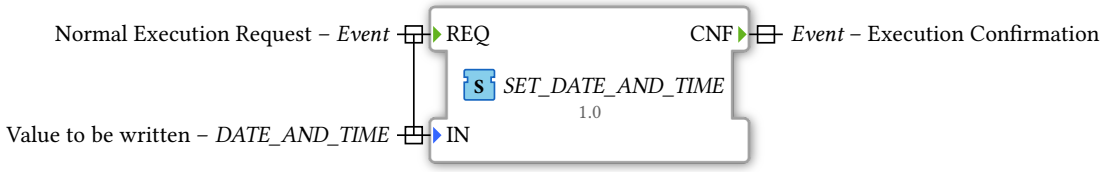
- **SET_TIME** (für `TIME`)
- **SET_DT** (für `DATE_AND_TIME`)

Der wesentliche Unterschied liegt im verwendeten Datentyp. Die Funktionsweise und die InOut-Schnittstelle sind identisch.

Fazit

SET_DATE ist ein einfacher, aber nützlicher Baustein zum direkten Setzen einer Datumsvariablen. Durch die Verwendung einer InOut-Variable wird der Speicher- und Kommunikationsaufwand minimal gehalten. Der Baustein eignet sich besonders für Standardisierungsaufgaben in der Automatisierungstechnik, bei denen Datumswerte effizient und zuverlässig übertragen werden müssen.

SET_DATE_AND_TIME



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock SET_DATE_AND_TIME dient dazu, einen eingehenden DATE_AND_TIME-Wert auf eine Zielvariable zu schreiben. Die Zielvariable wird als InOut-Parameter übergeben, sodass sie sowohl gelesen als auch beschrieben werden kann. Der Baustein realisiert eine einfache Zuweisungsoperation, die durch ein Ereignis ausgelöst wird.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung
REQ	Normaler Ausführungsanforderung; startet das Schreiben des Eingangswerts in die InOut-Variable.

Ereignis-Ausgänge

Ereignis	Beschreibung

CNF	Bestätigung der abgeschlossenen Ausführung.

Daten-Eingänge

Name	Typ	Beschreibung
IN	DATE_AND_TIME	Der Wert, der in die Zielvariable geschrieben werden soll.

Daten-Ausgänge

Der Baustein besitzt keine expliziten Daten-Ausgänge. Der aktualisierte Wert wird über den InOut-Parameter `OUT` nach außen sichtbar.

Adapter

Keine.

Funktionsweise

Sobald ein Ereignis am Eingang `REQ` anliegt, wird die ST-Anweisung `OUT := IN` ausgeführt. Der Wert des Eingangs `IN` wird direkt in die InOut-Variable `OUT` kopiert. Nach erfolgreichem Abschluss wird der Ausgang `CNF` getriggert, um die Fertigstellung zu signalisieren. Die Verwendung einer InOut-Variable erlaubt es, den Baustein in eine bestehende Datenverbindung einzufügen, ohne eine separate Ausgangsvariable zu benötigen.

Technische Besonderheiten

- **InOut-Variable:** Der Parameter `OUT` ist als InOut deklariert. Dadurch kann der Baustein den Wert der verbundenen Variable direkt überschreiben und gleichzeitig nach außen hin lesbar sein.
- **Einfache Zuweisung:** Es handelt sich um eine reine Kopieroperation ohne Typkonvertierung oder Plausibilitätsprüfung.
- **Keine Zustandsautomaten:** Der Baustein ist als SimpleFB mit nur einem EZ-Zustand (REQ) realisiert. Die Ausführung erfolgt streng in einem Zyklus.

Zustandsübersicht

Der Baustein besitzt einen einzigen Zustand:

Zustand	Aktion	Ausgabeereignis
REQ	`OUT := IN`	CNF

Der Zustandsautomat startet bei Erhalt von `REQ` in diesem Zustand, führt die Aktion aus und sendet sofort `CNF`.

Anwendungsszenarien

- **Initialisierung einer Systemzeit:** Ein übergeordneter Baustein kann eine aktuelle Zeit (z. B. von einer Schnittstelle) über `IN` bereitstellen und über `SET_DATE_AND_TIME` in eine globale oder persistent gespeicherte Zeitvariable (InOut) schreiben.
- **Rücksetzen einer Zeitvariablen:** Beim Setzen eines Referenzzeitpunkts kann der Baustein genutzt werden, um einen vorgegebenen Wert in eine übergeordnete Variable zu übernehmen.
- **Parametrierung:** In Regel-/Steuerungsanwendungen kann der Baustein dazu verwendet werden, Soll-Zeitwerte in Datenbausteine zu schreiben.

Vergleich mit ähnlichen Bausteinen

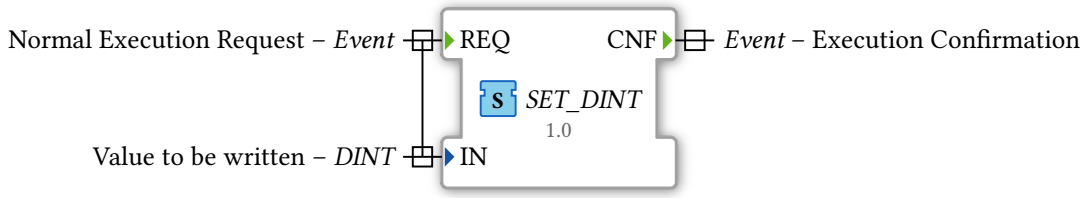
Einfache Zuweisungsbausteine für andere Datentypen (z. B. `SET_VALUE` für integer oder bool) folgen dem gleichen Muster. Der `SET_DATE_AND_TIME` ist speziell auf den Typ `DATE_AND_TIME` zugeschnitten und bietet keine erweiterten Funktionen wie Addition oder Subtraktion. Im Unterschied zu einem `MOVE`-Baustein (der Eingang auf Ausgang legt) verwendet dieser Baustein eine InOut-Variable, sodass die Quelle und das Ziel nicht getrennt werden müssen.

Fazit

Der Funktionsblock `SET_DATE_AND_TIME` ist ein einfacher, aber nützlicher Baustein zum Überschreiben einer Datums- und Zeitvariablen innerhalb eines IEC 61499-Systems. Durch die Verwendung einer InOut-Schnittstelle lässt er sich leicht in bestehende Datenflüsse integrieren. Aufgrund seiner schlichten Struktur

ist er effizient und für alle Szenarien geeignet, in denen eine gezielte Zuweisung eines Datum-Zeit-Werts erforderlich ist.

SET_DINT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `SET_DINT` dient dazu, einen DINT-Wert von einem Dateneingang auf eine InOut-Variable zu schreiben. Er realisiert damit eine einfache Zuweisungsoperation für den Datentyp `DINT`. Der Baustein wird über ein Ereignis gesteuert und quittiert die erfolgreiche Ausführung mit einem Bestätigungsereignis.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar	Mit Variablen
REQ	Event	Normaler Ausführungsanstoß	IN, OUT

Ereignis-Ausgänge

Name	Typ	Kommentar	Mit Variablen
CNF	Event	Bestätigung der Ausführung	OUT

Daten-Eingänge

Name	Typ	Kommentar	Initialwert
IN	DINT	Wert, der geschrieben werden soll	0

Daten-Ausgänge

Der Baustein besitzt keine klassischen Datenausgänge, sondern einen InOut-Parameter (siehe unten). Das Bestätigungsereignis `CNF` ist mit dem Ausgang `OUT` verknüpft, sodass nach der Ausführung der aktualisierte Wert am Ausgang anliegt.

Adapter

Keine.

Funktionsweise

Bei einem Ereignis am Eingang `REQ` wird der Algorithmus `REQ` ausgeführt. Dieser kopiert den aktuellen Wert des Dateneingangs `IN` direkt in die InOut-Variable `OUT`. Anschließend wird das Bestätigungsereignis `CNF` gesendet.

Die Logik in Structured Text lautet:

```
OUT := IN;
```

Die Ausführung ist atomar: Der Wert wird sofort und ohne Verzögerung übernommen.

Technische Besonderheiten

- Der Baustein verwendet eine **InOut-Variable** (`OUT`). Diese Variable ist gleichzeitig Eingang und Ausgang und kann von außen gelesen und beschrieben werden. Dadurch kann der Baustein den Wert einer bestehenden Variable an Ort und Stelle überschreiben, ohne einen zusätzlichen Ausgang zu belegen.

- Der Baustein ist zustandslos – es gibt nur einen einzigen Ausführungszustand. Jeder Aufruf führt die gleiche einfache Zuweisung aus.
- Alle Variablen sind initial mit `0` vordefiniert.

Zustandsübersicht

Zustand	Aktion	Ausgabeereignis
REQ	OUT := IN	CNF

Der Baustein besitzt keine weiteren Zustände (keine Warteschleifen, keine Verzweigungen). Nach der Aktion wechselt er sofort wieder in den Bereitschaftszustand für den nächsten `REQ`.

Anwendungsszenarien

- **Parameterübergabe:** Ein externer Wert (z. B. aus einer Konfigurationsschnittstelle) wird in eine lokale Variable einer Applikation geschrieben.
- **Signalweiterleitung:** Ein digitaler Integer-Wert soll von einem Signalpfad auf einen anderen umgeleitet werden, wobei der ursprüngliche Wert erhalten bleibt.
- **Initialisierung:** Beim Systemstart kann über `SET_DINT` ein definierter Startwert in eine Zustandsvariable eingetragen werden.
- **Test- und Debuggingzwecke:** Manuelles Setzen eines DINT-Wertes über eine HMI oder ein Skript.

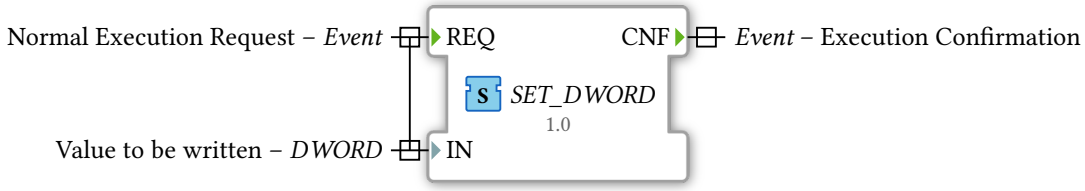
Vergleich mit ähnlichen Bausteinen

- **MOVE/DINT**: Standardisierte Bausteine wie `MOVE` aus der IEC-61131-3 führen ebenfalls Zuweisungen durch. `SET_DINT` ist auf den Typ `DINT` spezialisiert und bietet eine reduzierte Schnittstelle (nur ein Eingangsereignis, keine zusätzlichen Ausgänge).
- **SET_BOOL, SET_REAL, SET_SINT**: Analoge Bausteine für andere Datentypen existieren in der `logiBUS`-Bibliothek. Sie unterscheiden sich nur im Datentyp, die Funktionsweise ist identisch.
- **Direkte Zuweisung im Netzwerk**: In IEC-61499 könnte man eine Zuweisung auch ohne Baustein durch eine direkte Datenverbindung realisieren. Der Baustein `SET_DINT` bietet jedoch eine explizite, ereignisgesteuerte Kapselung, die zur Laufzeit überwacht und getriggert werden kann.

Fazit

`SET_DINT` ist ein einfacher, aber nützlicher Funktionsbaustein für die gezielte Zuweisung von DINT-Werten. Durch die Verwendung einer InOut-Variable bleibt die Schnittstelle schlank, und der Baustein kann direkt in bestehende Datenflüsse eingebunden werden. Seine klare, zustandslose Logik macht ihn zuverlässig und leicht verständlich – ideal für grundlegende Übergabeoperationen in Automatisierungsprojekten.

SET_DWORD



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `SET_DWORD` dient dazu, einen DWORD-Wert über einen Eingang in eine als InOut deklarierte Zielvariable zu schreiben. Er stellt eine einfache und effiziente Möglichkeit dar, Werte im laufenden Betrieb zu setzen, ohne dass eine separate Zuweisung im Programm erfolgen muss.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Comment

REQ	Event	Normal Execution Request – löst das Schreiben aus

Mit dem Ereignis `REQ` werden die Daten `IN` und `OUT` gleichzeitig bereitgestellt.

Ereignis-Ausgänge

Name	Typ	Comment
CNF	Event	Execution Confirmation – wird nach erfolgreicher Ausführung gesendet

Der Ausgang `CNF` referenziert die veränderte Variable `OUT`.

Daten-Eingänge

Name	Typ	Comment
IN	DWORD	Wert, der in die Zielvariable geschrieben werden soll (Initialwert: 0)

Daten-Ausgänge

Name	Typ	Comment
OUT	DWORD	Zielvariable (InOut) – der Wert von `IN` wird hierhin geschrieben (Initialwert: 0)

Hinweis: `OUT` ist als InOut-Variable deklariert, d.h. sie kann von außen referenziert werden und ihr Speicherort wird direkt durch den Baustein verändert. Sie erscheint im FB-Netzwerk sowohl als Ausgang als auch als Eingang.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Baustein besitzt eine einzelne Aktion, die beim Eintreffen des Ereignisses `REQ` ausgeführt wird:
`OUT := IN;`
Anschließend wird das Ereignis `CNF` gesendet. Der Baustein ist zustandslos und führt bei jedem `REQ` die Zuweisung aus, unabhängig vom aktuellen Wert.

Technische Besonderheiten

- **InOut-Variable**: Die Verwendung einer InOut-Variable erlaubt es, direkt auf den Speicherort einer externen Variable zuzugreifen. Dies reduziert den Datenkopieraufwand und ermöglicht die Veränderung von Variablen, die auch an anderen Stellen im System genutzt werden.
- **Keine Zustandslogik**: Der Baustein implementiert keinen Zustandsautomaten mit mehreren Zuständen – er reagiert immer gleich auf das Eingangsereignis.
- **Initialwerte**: Beide Variablen sind mit 0 vorbelegt, werden aber in der Regel zur Laufzeit überschrieben.

Zustandsübersicht

Der Baustein besteht aus einem einfachen Zustandsautomaten mit genau einem Zustand:

Zustand	Auslösendes Ereignis	Aktion	Ausgang
`REQ`	`REQ`	`OUT := IN;`	`CNF`

Es gibt keine Wartezustände oder Verzweigungen.

Anwendungsszenarien

- **Konfiguration von Peripherie**: Setzen eines DWORD-Registers in einem Modul (z.B. Zeitbasis, Schwellwert).
- **Parameterweitergabe**: Übergabe eines konstanten oder berechneten Werts an eine global verfügbare Variable (z.B. über einen InOut-Datentyp).
- **Test und Debugging**: Temporäres Überschreiben eines Wertes während des Betriebs, ohne den Quellcode zu ändern.
- **Migration von Klartextzuweisungen**: Ersatz für eine einfache `OUT := IN`-Anweisung in einem IEC-61499-Netzwerk durch einen wiederverwendbaren Baustein.

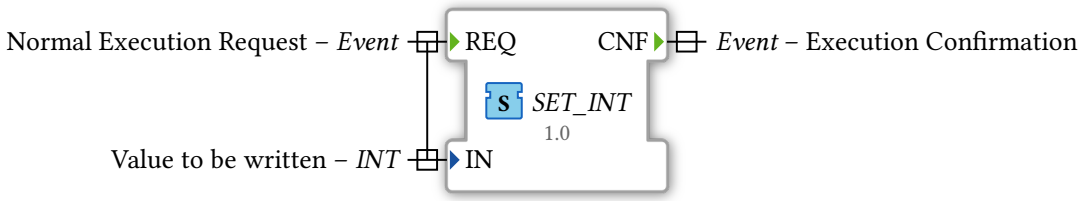
Vergleich mit ähnlichen Bausteinen

- **MOVE / MV**: In vielen Bibliotheken vorhanden, arbeiten jedoch meist mit reinen Ein-/Ausgangsvariablen und erfordern einen expliziten Zwischenspeicher. `SET_DWORD` greift dagegen direkt auf eine bestehende Variable zu (InOut).
- **SET_BOOL / SET_INT / SET_REAL**: Analoge Bausteine für andere Datentypen; das Prinzip ist identisch, lediglich der Datentyp variiert.
- **WRITE_DWORD**: Einige Bibliotheken bieten einen Baustein, der über einen Adressbus schreibt. `SET_DWORD` ist einfacher und benötigt keine Adresslogik.

Fazit

`SET_DWORD` ist ein minimaler, aber sehr nützlicher Funktionsbaustein für die direkte Wertzuweisung an eine DWORD-InOut-Variable. Seine Einfachheit macht ihn fehlerunanfällig, und der Verzicht auf Zustandslogik sorgt für vorhersagbares Echtzeitverhalten. Er eignet sich besonders für Konfigurations- und Initialisierungsaufgaben, bei denen eine schnelle, unkomplizierte Wertübernahme erforderlich ist.

SET_INT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **SET_INT** dient dazu, einen Wert vom Datentyp `INT` aus einem Eingang direkt in eine InOut-Variable zu schreiben. Der Baustein ist ein einfacher, ereignisgesteuerter FB und wird typischerweise verwendet, um globale Parameter oder Variablen anderer FBs durch einen definierten Ereignisimpuls zu setzen. Die Funktionalität ist auf die minimale Aufgabe der Wertzuweisung reduziert.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Datentyp	Beschreibung	Mit Daten
`REQ`	Event	Normaler Ausführungsanforderung. Löst das Übernehmen des Wertes von `IN` nach `OUT` aus.	`IN`, `OUT`

Ereignis-Ausgänge

Ereignis	Datentyp	Beschreibung	Mit Daten
`CNF`	Event	Ausführungsbestätigung. Wird gesendet, nachdem der Wert erfolgreich in `OUT` geschrieben wurde.	`OUT`

Daten-Eingänge

Variable	Datentyp	Beschreibung	Initialwert
`IN`	INT	Der Wert, der in die InOut-Variable kopiert werden soll.	0

Daten-Ausgänge

Der FB besitzt keinen reinen Datenausgang. Die Ausgabe erfolgt über die InOut-Variable (siehe unten).

Adapter

Keine.

InOut-Variable

Variable	Datentyp	Beschreibung	Initialwert
OUT	INT	Zielvariable, in die der Wert von IN geschrieben wird. Diese Variable ist als InOut deklariert, d. h. sie wird von außerhalb des Bausteins referenziert und kann direkt manipuliert werden.	0

Funktionsweise

Der Baustein arbeitet nach einem sehr einfachen, direkten Prinzip:

1. Ein Ereignis am Eingang REQ wird empfangen.
2. Der Algorithmus REQ führt die Zuweisung OUT := IN aus. Der aktuelle Wert des Daten-Eingangs IN wird also in die InOut-Variable OUT kopiert.
3. Nach erfolgreicher Zuweisung wird das Ereignis CNF gesendet, um der aufrufenden Umgebung die Fertigstellung zu signalisieren. Das CNF-Ereignis ist mit der InOut-Variable OUT verknüpft, sodass die Datenkonsistenz gewahrt bleibt.

Die Ausführung erfolgt ohne Verzögerung oder Bedingungen – bei jedem REQ wird der Wert sofort überschrieben.

Technische Besonderheiten

- **InOut-Variable**
  Der Baustein nutzt einen InOut-Parameter (OUT), der eine direkte Referenz auf eine externe Variable (z. B. einen globalen Datenbaustein oder eine Variable eines übergeordneten Funktionsblocks) darstellt. Dadurch entfällt die Notwendigkeit einer expliziten Rückverbindung des Ausgangs zu einem anderen Baustein. Der FB schreibt seinen Wert direkt an die referenzierte Speicherstelle.
- **Keine Pufferung**
  Der FB führt keine Zwischenspeicherung durch. Der Wert wird bei jedem REQ-Ereignis unmittelbar überschrieben.
- **Initialwerte**
  Die Daten-Eingänge und InOut-Variablen sind mit 0 initialisiert. Im Falle einer fehlenden Verbindung wird dieser Wert verwendet.

Zustandsübersicht

Der FB besitzt einen einzigen ECC-Zustand (REQ), der die Aktion ausführt und sofort das Ausgangsereignis CNF aktiviert. Es gibt keine Verzweigungen, Timer oder Fehlerzustände.

Zustand	Aktion	Auslöser
REQ	Ausführung des Algorithmus REQ (OUT := IN) und Senden von CNF	Ereignis REQ

Anwendungsszenarien

- **Setzen eines Konstantenwerts**
  Ein externer Sensor oder eine Logik liefert einen Sollwert (IN), der bei einem Takt (REQ) in eine globale Sollwertvariable (OUT) übernommen wird.
- **Initialisierung von Variablen**
  Beim Start einer Anwendung kann SET_INT verwendet werden, um mehrere InOut-Variablen auf definierte Startwerte zu setzen.
- **Parameter-Override**
  Ein übergeordneter Regler oder eine Bedienschnittstelle überschreibt bei einem speziellen Ereignis den aktuellen Wert eines anderen Bausteins, ohne den Baustein selbst ändern zu müssen (die InOut-Verbindung erlaubt den direkten Zugriff).

Vergleich mit ähnlichen Bausteinen

- **SET_BOOL / SET_REAL**
  Analog zu SET_INT, jedoch für die Datentypen BOOL bzw. REAL. Die Funktionsweise ist identisch.
- **MOVE**
  Ein allgemeiner MOVE-Baustein kopiert einen Wert von einem Daten-Eingang zu einem Daten-

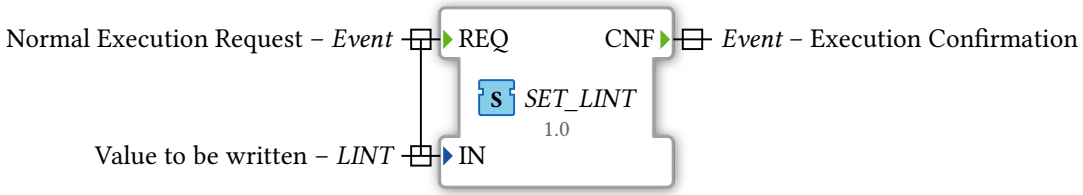
Ausgang. Im Unterschied dazu arbeitet `SET_INT` mit einer InOut-Variablen und schreibt direkt an eine externe Stelle, was die Kopplung enger macht. `MOVE` benötigt eine separate Verbindung vom Ausgang zum Ziel.

- **LATCH**
  Ein Latch-FB kann einen Wert speichern und bei Bedarf ausgeben. `SET_INT` überschreibt dagegen ohne Haltefunktion – jeder Aufruf überschreibt den vorherigen Wert.

Fazit

Der `SET_INT`-Funktionsblock ist ein minimalistischer, aber effektiver Baustein zum direkten Setzen einer INT-Variable über eine InOut-Schnittstelle. Seine Stärke liegt in der einfachen Logik und der Fähigkeit, von außen auf fremde Variablen zuzugreifen. Er eignet sich besonders für flache Steuerungsstrukturen, bei denen schnelle und direkte Wertzuweisungen benötigt werden. Aufgrund seiner Einfachheit ist er robust und fehlerarm einsetzbar, erfordert jedoch ein durchdachtes Variablenkonzept, um unerwünschte Seiteneffekte zu vermeiden.

SET_LINT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **SET_LINT** dient dazu, einen Wert vom Typ `LINT` (64-Bit-Ganzzahl) aus einem Dateneingang in eine als InOut-Parameter deklarierte Variable zu schreiben. Er wird typischerweise verwendet, um explizit den Wert einer Variablen zu setzen, die außerhalb des Bausteins definiert ist. Der Baustein folgt der Norm IEC 61499-1 und stellt einen einfachen, ereignisgesteuerten Schreibzugriff bereit.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung	Mit Variablen
REQ	Event	Auslösung des Schreibvorgangs	IN, OUT

Ereignis-Ausgänge

Name	Typ	Beschreibung	Mit Variablen
CNF	Event	Bestätigung nach erfolgreichem Schreiben	OUT

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung
IN	LINT	0	Der zu schreibende Wert

Daten-Ausgänge

Der Baustein besitzt keine klassischen Datenausgänge, sondern einen **InOut-Parameter** `OUT`, der gleichzeitig als Ein- und Ausgang fungiert. Dieser Parameter ist der Zielspeicherort für den Wert von `IN`.

Adapter

Keine.

Funktionsweise

Das Verhalten des Bausteins ist auf einen einzelnen Schritt reduziert:

1. Ein Ereignis am Eingang **REQ** löst die interne Verarbeitung aus.
2. Der Algorithmus führt die Zuweisung `OUT := IN` aus. Dadurch wird der Wert des Dateneingangs `IN` in die InOut-Variable `OUT` kopiert.
3. Nach Abschluss der Zuweisung wird das Ereignis **CNF** gesendet, um die erfolgreiche Ausführung zu signalisieren.

Die InOut-Variable `OUT` muss in der Applikation an einen Speicherort (z. B. eine globale oder gerätespezifische Variable) gebunden sein. Der Baustein überschreibt den dort gespeicherten Wert mit dem Wert von `IN`.

Technische Besonderheiten

- Der Baustein verwendet einen **InOut-Parameter** anstelle eines separaten Datenausgangs. Dies ermöglicht das direkte Schreiben auf eine externe Variable, ohne dass eine zusätzliche Verschaltung über einen Datenausgang erforderlich ist.
- Der Datentyp `LINT` (64-Bit signed integer) ist plattformabhängig; die Implementierung setzt voraus, dass das Zielsystem diesen Typ unterstützt.
- Es gibt keine Fehlerbehandlung oder Bereichsprüfung – die Zuweisung erfolgt immer, sobald `REQ` ausgelöst wird.
- Der Baustein ist ein **SimpleFB** mit nur einem ECR-Zustand, daher ist das Verhalten deterministisch und schnell.

Zustandsübersicht

Der Baustein besitzt lediglich einen Zustand `REQ`. Nach Ausführung des Algorithmus wechselt er automatisch zurück in den Startzustand (implizit). Eine Zustandsmaschine im klassischen Sinne existiert nicht – der Baustein ist stateless. Nach jedem `REQ`-Ereignis wird genau ein Schreibvorgang durchgeführt.

Anwendungsszenarien

- **Initialisierung von LINT-Variablen**: Setzen einer globalen Konfigurationsvariablen auf einen definierten Startwert.
- **Steuerungs-/Regelungsparameter ändern**: Dynamisches Überschreiben eines Parameters (z. B. Sollwert oder Grenzwert) durch einen übergeordneten Baustein.
- **Test- und Debug-Schnittstellen**: Ermöglicht das manuelle Setzen von Variablen zur Laufzeit über eine HMI oder ein Skript.
- **Nachrichtenkopf-/Payload-Setzen**: In der Kommunikation kann `IN` den Wert einer Nachricht enthalten, der in einen Puffer (`OUT`) geschrieben wird.

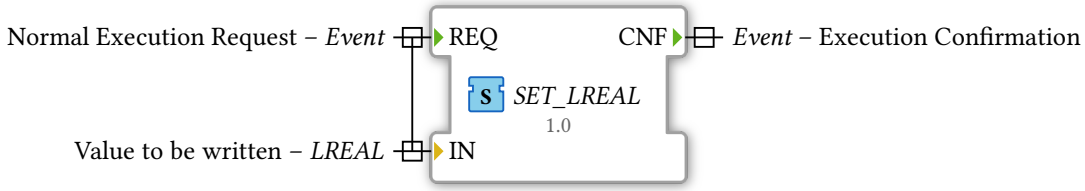
Vergleich mit ähnlichen Bausteinen

- **SET_DINT / SET_INT**: Analoge Bausteine für `DINT` (32-Bit) bzw. `INT` (16-Bit). Der Unterschied liegt ausschließlich im Datentyp.
- **MOVE**: Ein allgemeinerer Baustein zum Kopieren beliebiger Werte, oft ebenfalls mit InOut-Parametern. `SET_LINT` ist spezialisiert auf `LINT` und bietet keine weiteren Daten- oder Ereigniseingänge.
- **WRITE_LINT (hypothetisch)**: Ein Baustein, der eine Adresse explizit über einen Pointer schreibt. `SET_LINT` arbeitet ohne Adressierung und erwartet eine direkt verbundene Variable.

Fazit

`SET_LINT` ist ein minimalistischer Funktionsbaustein, der eine einzige Aufgabe zuverlässig erfüllt: das Zuweisen eines `LINT`-Werts an eine InOut-Variable. Aufgrund seiner Einfachheit eignet er sich besonders für leistungskritische oder überschaubare Steuerungsaufgaben, bei denen keine komplexe Logik erforderlich ist. Der Verzicht auf Fehlerbehandlung macht ihn transparent und vorhersagbar, erfordert aber eine korrekte Anbindung der InOut-Variable durch den Anwender.

SET_LREAL



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **SET_LREAL** dient dazu, einen Wert vom Typ LREAL an eine InOut-Variable zu übergeben. Der Eingangswert wird bei einer steigenden Flanke am REQ-Ereignis in die Zielvariable geschrieben und der Vorgang über das CNF-Ereignis quittiert.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Mit Variablen	Beschreibung
REQ	IN, OUT	Normaler Ausführungsanforderung. Löst das Schreiben des Eingangswertes in die Zielvariable aus.

Ereignis-Ausgänge

Ereignis	Mit Variablen	Beschreibung
CNF	OUT	Bestätigung der erfolgreichen Ausführung. Die Zielvariable wurde aktualisiert.

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung
IN	LREAL	0.0	Wert, der in die Zielvariable geschrieben werden soll.
OUT	LREAL	0.0	(InOut) Eingangsseite der Zielvariablen. Der aktuelle Wert kann gelesen werden.

Daten-Ausgänge

Name	Typ	Initialwert	Beschreibung
OUT	LREAL	0.0	(InOut) Ausgangsseite der Zielvariablen. Nach Ausführung wird hier der Wert von IN ausgegeben.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Bei einem Ereignis am Eingang **REQ** wird der Algorithmus ausgeführt, der den Wert der Eingangsvariablen **IN** in die InOut-Variable **OUT** kopiert. Anschließend wird das Ereignis **CNF** gesendet. Der Baustein arbeitet rein kombinatorisch und hat keine interne Zustandshaltung über die Ausführung hinaus.

Technische Besonderheiten

- Der Baustein verwendet eine InOut-Variable. Diese ermöglicht den direkten Zugriff auf eine externe Variable, ohne dass eine zusätzliche Verbindung zwischen Eingang und Ausgang erforderlich ist.
- Der Typ LREAL entspricht einer Gleitkommazahl doppelter Genauigkeit gemäß IEC 61499.
- Der Algorithmus ist in Structured Text (ST) realisiert und sehr einfach gehalten (`OUT := IN;`).

Zustandsübersicht

Der Baustein besteht aus einem einzigen Zustand **REQ**:

- **REQ**: Wartet auf das Eintreffen des REQ-Ereignisses. Führt dann den Algorithmus aus und sendet CNF. Nach dem Senden kehrt er wieder in den Zustand REQ zurück (implizit, da keine weiteren Zustände definiert sind).

Es gibt keine Verzweigungen oder Zeitsteuerungen.

Anwendungsszenarien

- **Kopieren eines LREAL-Wertes** in eine globale oder instanzübergreifende Variable, wenn eine direkte Verbindung nicht möglich oder gewünscht ist.
- **Setzen eines Sollwertes** in einer Steuerung, z. B. für einen PID-Regler, bei dem der Wert über einen analogen Eingang kommt.
- **Initialisierung** von Variablen im laufenden Betrieb, ohne die gesamte Applikation neu starten zu müssen.

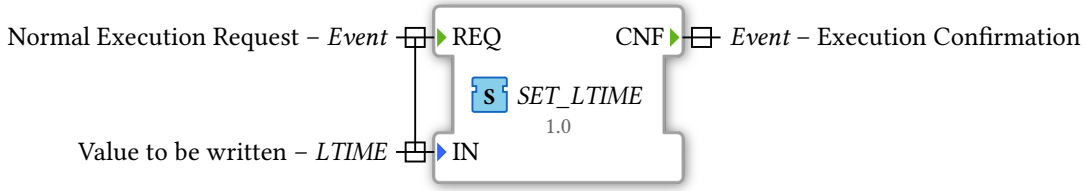
Vergleich mit ähnlichen Bausteinen

- **SET_INT** oder **SET_BOOL**: Funktionsgleiche Bausteine für andere Datentypen.
- **MOVE**: Ein allgemeiner MOVE-Baustein (z. B. in IEC 61131-3) kopiert Werte ohne Ereignissteuerung. SET_LREAL hingegen nutzt ereignisgesteuerte Ausführung und InOut-Variablen.
- **WRITE_LREAL**: Manche Bibliotheken bieten separate Bausteine zum Schreiben auf eine Adresse. SET_LREAL ist einfacher und arbeitet auf Variablenebene.

Fazit

Der **SET_LREAL**-Baustein ist ein einfaches, aber nützliches Werkzeug, um einen LREAL-Wert in eine InOut-Variable zu schreiben. Seine ereignisgesteuerte Arbeitsweise macht ihn leicht in IEC 61499-Applikationen integrierbar. Die klare Schnittstelle mit nur einem Ereignisein- und -ausgang sowie einem Dateneingang und einer InOut-Variable reduziert die Komplexität und erleichtert die Wiederverwendung.

SET_LTIME



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **SET_LTIME** dient dazu, einen Eingangswert vom Typ `LTIME` in eine als InOut deklarierte Variable zu schreiben. Dies ermöglicht es, den Wert einer externen Referenz (z. B. einer Instanzvariablen oder eines anderen Bausteins) direkt zu setzen. Die Verwendung einer InOut-Variable erlaubt eine effiziente Übergabe, ohne einen separaten Datenausgang zu benötigen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
REQ	Event	Normaler Ausführungsanstoß. Verbunden mit den Daten `IN` und `OUT`.

Ereignis-Ausgänge

Name	Typ	Kommentar
CNF	Event	Bestätigung der erfolgreichen Ausführung. Verbunden mit Daten `OUT`.

Daten-Eingänge

Name	Typ	Initialwert	Kommentar
IN	LTIME	LT#0s	Wert, der in die Zielvariable geschrieben werden soll.

Daten-Ausgänge

Name	Typ	Initialwert	Kommentar
OUT	LTIME	LT#0s	**InOut-Variable** – Zielvariable, die den übergebenen Wert erhält. Der FB schreibt den Wert von `IN` in diese Variable. Als InOut kann sie sowohl gelesen als auch von außen beschrieben werden.

Adapter

Keine.

Funktionsweise

Bei einem Ereignis am Eingang **REQ** wird der Algorithmus `REQ` ausgeführt:

- Der Wert des Dateneingangs `IN` wird direkt in die InOut-Variable `OUT` kopiert.
- Anschließend wird das Ereignis **CNF** ausgegeben, um die erfolgreiche Zuweisung zu signalisieren.

Die Logik ist denkbar einfach und besteht aus einer einzigen Zuweisung:

```
OUT := IN;
```

Technische Besonderheiten

- Der FB arbeitet ausschließlich mit dem Datentyp **LTIME** (Zeit mit hoher Auflösung, gemäß IEC 61499).
- Die Zielvariable `OUT` ist als **InOut** deklariert. Das bedeutet, dass sie von außen als Referenz übergeben wird. Änderungen, die der FB an `OUT` vornimmt, wirken sich unmittelbar auf die referenzierte Variable aus.
- Initial sind sowohl `IN` als auch `OUT` auf `LT#0s` gesetzt, um undefinierte Zustände zu vermeiden.
- Der Baustein ist ein **einfacher FB** (SimpleFB) mit nur einem Zustand, der sofort auf ein REQ-Ereignis reagiert.

Zustandsübersicht

Der FB besitzt genau einen Zustand:

Zustand	Aktion	Ausgabe
REQ	`REQ` (OUT := IN)	CNF

Es gibt keine Verzweigungen oder Zeitsteuerungen. Der FB ist im Grunde kombinatorisch, da er sofort auf das Eingangsereignis reagiert.

Anwendungsszenarien

- **Setzen einer zentralen Zeitbasis**: Wenn mehrere Bausteine auf eine gemeinsame LTIME-Variable zugreifen, kann `SET_LTIME` diese Variable aktualisieren.
- **Konfiguration von Laufzeiten**: In einer Steuerungsanwendung können Sollzeitwerte aus einer Konfigurationsschnittstelle in die entsprechenden InOut-Parameter geschrieben werden.
- **Test und Simulation**: Der Baustein eignet sich, um in Testumgebungen gezielt Werte in eine beobachtbare Variable zu schreiben.

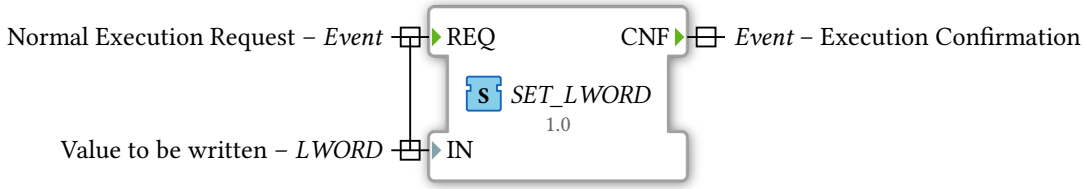
Vergleich mit ähnlichen Bausteinen

- **SET_TIME**: Arbeitet analog, jedoch mit dem Datentyp `TIME` (Sekundenauflösung). `SET_LTIME` bietet eine höhere Zeitauflösung (Nanosekunden) und ist daher für präzise Anwendungen geeignet.
- **MOVE-Bausteine**: Viele IEC-61499-Bibliotheken enthalten generische MOVE-FBs, die Werte von einem Eingang auf einen Ausgang kopieren. `SET_LTIME` spezialisiert sich auf `LTIME` und nutzt eine InOut-Referenz statt eines separaten Ausgangs, was in bestimmten Architekturen vorteilhaft sein kann.

Fazit

`SET_LTIME` ist ein minimalistischer, aber nützlicher Funktionsblock zum gezielten Setzen einer LTIME-Variablen. Seine klare Schnittstelle und die Verwendung einer InOut-Referenz machen ihn zu einem effizienten Werkzeug in Zeitsteuerungs- und Konfigurationsszenarien. Der Baustein ist einfach zu verstehen und unkompliziert einzusetzen.

SET_LWORD



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **SET_LWORD** dient dazu, einen LWORD-Wert (64-Bit-Integer) von einem Dateneingang auf eine InOut-Variable zu übertragen. Er ermöglicht das direkte Setzen eines Zielwerts, der gleichzeitig als Ein- und Ausgang fungiert. Dadurch kann der Baustein in Systemen eingesetzt werden, in denen ein bestimmter Speicherplatz oder eine Verbindungsvariable aktualisiert werden muss.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Datentyp	Kommentar
REQ	Event	Normaler Ausführungsanstoß

Ereignis-Ausgänge

Ereignis	Datentyp	Kommentar
CNF	Event	Bestätigung der Ausführung

Daten-Eingänge

Name	Datentyp	Initialwert	Kommentar
IN	LWORD	0	Wert, der geschrieben wird

Daten-Ausgänge

Name	Datentyp	Initialwert	Kommentar
OUT	LWORD	0	Zielvariable (als InOut deklariert)

> **Hinweis:** OUT wird in der XML als `<InOutVars>` deklariert. Es handelt sich um eine kombinierte Ein-/Ausgangsvariable, die sowohl als Quelle der Wertzuweisung (lesend) als auch als Ziel (schreibend) dient.

Adapter

Keine.

Funktionsweise

Wird ein Ereignis am **REQ**-Eingang empfangen, führt der Baustein den folgenden Algorithmus aus:

```
OUT := IN;
```

Der Wert des Dateneingangs **IN** wird direkt auf die InOut-Variable **OUT** übertragen. Anschließend wird das Ereignis **CNF** gesendet, um die erfolgreiche Ausführung zu bestätigen.

Der Baustein ist ein einfacher, zustandsloser Funktionsblock – es gibt keine Verzögerungen oder Nebenläufigkeiten.

Technische Besonderheiten

- **InOut-Variable:** Die Variable `OUT` ist als InOut deklariert. Das bedeutet, sie kann im Netzwerk sowohl als Eingang (zum Lesen eines bestehenden Wertes) als auch als Ausgang (zum Setzen eines neuen Wertes) verwendet werden. Der Algorithmus überschreibt den Inhalt der angeschlossenen Variable mit dem Eingangswert.
- **Typ LWORD:** Der Baustein ist auf 64-Bit-Werte spezialisiert. Bei Verwendung mit kürzeren Datentypen (z. B. DWORD) muss eine explizite Typumwandlung erfolgen.
- **Initialwert:** Sowohl `IN` als auch `OUT` sind mit 0 vorbelegt.

Zustandsübersicht

Der Baustein besitzt nur einen einzigen EC-Zustand:

- **REQ:** Führt den Algorithmus aus und sendet anschließend das CNF-Ereignis.

Es gibt keine Wartezustände oder Schleifen.

Anwendungsszenarien

- **Initialisierung von Speicherstellen:** Setzen eines 64-Bit-Registers oder einer globalen Variable auf einen definierten Anfangswert.
- **Kopieren von Werten in InOut-Verbindungen:** Wenn ein anderer FB eine LWORD-Variable als InOut bereitstellt, kann SET_LWORD diese gezielt überschreiben.
- **Test- und Debugging-Umgebungen:** Manuelles Vorgeben von Werten in Datenpfaden, um das Systemverhalten zu überprüfen.

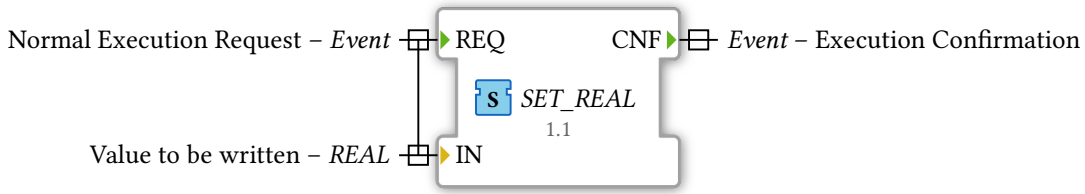
Vergleich mit ähnlichen Bausteinen

- **SET_BOOL / SET_DWORD / SET_SINT:** Analoge Bausteine für andere Basisdatentypen (BOOL, DWORD, SINT usw.). SET_LWORD deckt den 64-Bit-Integerbereich ab.
- **MOVE:** Ein allgemeiner Baustein, der Werte zwischen beliebigen Datentypen kopiert, jedoch oft ohne InOut-Unterstützung. SET_LWORD nutzt gezielt die InOut-Deklaration und ist damit für direkte Variablenzuweisungen optimiert.
- **WRITE_***: In einigen Bibliotheken existieren Bausteine zum Schreiben in Peripherieadressen. SET_LWORD ist abstrakter und auf logische Variablenebene angesiedelt.

Fazit

Der Funktionsbaustein **SET_LWORD** ist ein schlankes, aber essenzielles Werkzeug zum Setzen eines 64-Bit-Wertes auf eine InOut-Variable. Seine einfache Semantik („Schreibe IN nach OUT“) und die Unterstützung des Standarddatentyps LWORD machen ihn in IEC-61499-basierten Automatisierungssystemen vielseitig einsetzbar – insbesondere dort, wo eine direkte Manipulation von Variablen über Ereignissteuerung erforderlich ist.

SET_REAL



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **SET_REAL** dient dazu, einen REAL-Wert von einem Dateneingang auf eine InOut-Variable zu schreiben. Er ist ein einfacher, ereignisgesteuerter Baustein, der nach einer Anforderung (REQ)

den aktuellen Eingangswert IN an die referenzierte Variable OUT weitergibt und den Vorgang durch ein Bestätigungsereignis (CNF) quittiert.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung
REQ	Event	Normale Ausführungsanforderung (mit IN und OUT verbunden)

Ereignis-Ausgänge

Name	Typ	Beschreibung
CNF	Event	Bestätigung der Ausführung (mit OUT verbunden)

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung
IN	REAL	0.0	Wert, der geschrieben werden soll
OUT	REAL (InOut)	0.0	Zielvariable (InOut – lesend und schreibend)

Daten-Ausgänge

Keine separaten Datenausgänge vorhanden. Der InOut-Parameter OUT fungiert gleichzeitig als Ausgang und wird nach der Zuweisung aktualisiert.

Adapter

Keine.

Funktionsweise

Der FB besitzt einen einzigen Ausführungszustand **REQ**. Sobald ein Ereignis am Eingang **REQ** eintrifft, wird der hinterlegte Algorithmus ausgeführt:

```
OUT := IN;
```

Der aktuelle Wert von IN wird der InOut-Variable OUT zugewiesen. Anschließend wird das Ereignis **CNF** ausgegeben, um den Abschluss der Operation zu signalisieren. Der Vorgang ist atomar und benötigt keine weiteren Zustandsübergänge.

Technische Besonderheiten

- **InOut-Parameter OUT**: Die Variable OUT ist sowohl lesbar als auch beschreibbar. Sie wird direkt von der aufrufenden Umgebung referenziert, sodass die Zuweisung unmittelbar den Wert an der Ursprungsadresse ändert.
- **Initialwerte**: Sowohl IN als auch OUT sind mit 0.0 initialisiert. Dies verhindert undefinierte Zustände vor der ersten Ausführung.
- **Einfachheit**: Der FB besteht aus nur einem Zustand und einem Algorithmus, was eine sehr geringe Laufzeitkomplexität garantiert.

Zustandsübersicht

Zustand	Beschreibung	Aktion	Ausgabe
REQ	Warte auf Ausführungsanforderung	Führe Algorithmus REQ aus ⟶ OUT := IN	CNF

Es existiert kein weiterer Zustand; der FB kehrt nach dem Durchlauf sofort wieder in den wartenden Zustand REQ zurück.

Anwendungsszenarien

- **Parametereinstellung**: Übergabe eines REAL-Werts an eine globale Konfigurationsvariable, z. B. zur Skalierung oder für Sollwerte.

- **Initialisierung**: Setzen einer InOut-Variable auf einen definierten Startwert während der Inbetriebnahme.
- **Wertkopie innerhalb einer Applikation**: Einfaches Überschreiben einer bestehenden REAL-Variablen durch einen anderen Wert, gesteuert durch ein Ereignis.

Vergleich mit ähnlichen Bausteinen

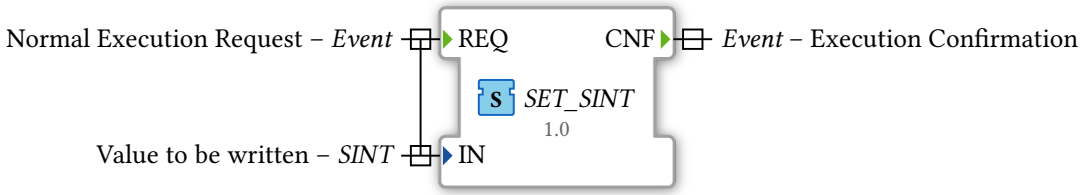
FB	Typ	Unterschied
SET_BOOL	BOOL	Schreibt boolesche Werte, analoge Funktionsweise
SET_INT	INT	Schreibt Ganzzahlen, identische Ereignissteuerung
MOVE (generisch)	ANY	Kann beliebige Datentypen kopieren, erfordert jedoch Typanpassung und hat oft mehrere Zustände
SET_REAL	REAL	Speziell für REAL optimiert, minimale Zustände und klare Semantik

Im Vergleich zu einem generischen MOVE-Baustein bietet SET_REAL eine schlankere Implementierung und ist auf den häufig benötigten REAL-Typ zugeschnitten.

Fazit

Der Funktionsblock **SET_REAL** ist ein einfacher, aber effektiver Baustein zum gezielten Setzen einer REAL-Variablen. Dank seiner minimalen Zustandsmaschine und der direkten Verwendung von InOut-Parametern eignet er sich hervorragend für ereignisgesteuerte Zuweisungen in IEC 61499-Applikationen. Er reduziert Komplexität und erhöht die Lesbarkeit des Steuerungscodes.

SET_SINT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **SET_SINT** dient dazu, einen SINT-Wert von seinem Dateneingang **IN** in eine InOut-Variable **OUT** zu schreiben. Er wird verwendet, um einen SINT-Wert an einem bestimmten Punkt der Ablaufsteuerung gezielt zu setzen. Der Baustein realisiert eine einfache, ereignisgesteuerte Zuweisung.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung
REQ	Normale Ausführungsanforderung. Löst die Zuweisung von **IN** nach **OUT** aus.

Ereignis-Ausgänge

Ereignis	Beschreibung
CNF	Bestätigung der Ausführung. Wird nach erfolgreicher Zuweisung gesendet.

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung
IN	SINT	0	Der Wert, der geschrieben werden soll.

Daten-Ausgänge

Der Funktionsblock besitzt keine separaten Datenausgänge. Der Zielwert wird über die InOut-Variable **OUT** zurückgegeben.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Ein **REQ**-Ereignis startet die Ausführung des Algorithmus. Im Algorithmus wird der aktuelle Wert von **IN** direkt der InOut-Variable **OUT** zugewiesen:
`OUT := IN;`
Nach Abschluss der Zuweisung wird das Ereignis **CNF** ausgegeben. Die InOut-Variable **OUT** dient gleichzeitig als Eingabe und Ausgabe – ihr Wert wird überschrieben und steht danach für nachfolgende Bausteine zur Verfügung.

Technische Besonderheiten

- **InOut-Variable**: Die Variable **OUT** ist als `InOut` deklariert. Sie kann sowohl gelesen als auch geschrieben werden. Der Baustein überschreibt den bisherigen Wert von **OUT** mit dem Wert von **IN**.
- **Einfachster Algorithmus**: Der Baustein führt nur eine einzige Zuweisung durch. Es gibt keine Bedingungen, Schleifen oder zusätzliche Logik.
- **Typsicherheit**: Sowohl **IN** als auch **OUT** sind vom Typ `SINT`, sodass eine direkte, typkonforme Zuweisung erfolgt.

Zustandsübersicht

Der Funktionsblock besitzt einen einzigen Zustand (ECState) mit dem Namen **REQ**:

Zustand	Aktion	Ausgabeereignis
REQ	`OUT := IN`	CNF

Es gibt nur diesen einen Zustand; nach Ausführung kehrt der Baustein in den initialen Zustand zurück. Eine explizite Zustandsmaschine mit mehreren Zuständen liegt nicht vor.

Anwendungsszenarien

- **Setzen eines SINT-Wertes zur Laufzeit**: Beispielsweise das Überschreiben eines Grenzwerts oder eines Zählers in einer Steuerung.
- **Initialisieren von SINT-Variablen** in einem bestimmten Schritt der Ablaufsteuerung.
- **Weitergabe eines konstanten oder berechneten Wertes** an einen anderen Baustein über die InOut-Variable, sofern eine Verbindung zu einer globalen Variablen oder einem anderen InOut-Port besteht.

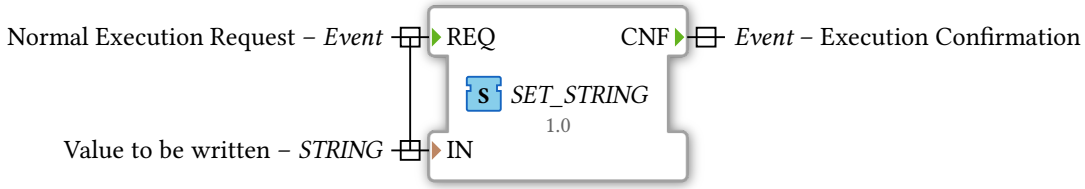
Vergleich mit ähnlichen Bausteinen

- **SET_BOOL**: Analoge Funktion für den Datentyp `BOOL`. Arbeitet identisch, jedoch mit booleschen Werten.
- **SET_INT**: Entsprechender Baustein für `INT`. Die Struktur (Ereignis REQ/CNF, Zuweisung) ist identisch.
- **MOVE (z. B. MOVE_SINT)**: Ein MOVE-Baustein kopiert einen Wert von einem Eingang zu einem Ausgang, ohne dass die Zielvariable als InOut ausgeführt ist. **SET_SINT** hingegen schreibt direkt in eine bestehende Variable, was in manchen Architekturen den Vorteil bietet, dass die Variable auch von anderen Bausteinen gelesen werden kann, ohne eine separate Verbindung zu einem Ausgang zu benötigen.

Fazit

SET_SINT ist ein minimaler und zuverlässiger Funktionsblock zum Setzen einer SINT-Variablen auf einen gewünschten Wert. Seine Einfachheit macht ihn ideal für Situationen, in denen ein Wert direkt und ohne zusätzliche Logik überschrieben werden soll. Der Einsatz einer InOut-Variable spart separate Ausgangsvariablen und erlaubt eine direkte Verkettung mit anderen Bausteinen, die dieselbe Variable lesen. Für einfache Wertzuweisungen ist er eine effiziente Lösung.

SET_STRING



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `SET_STRING` dient dazu, einen über den Dateneingang `IN` bereitgestellten Zeichenkettenwert (STRING) in eine In-Out-Variable `OUT` zu schreiben. Die Ausführung erfolgt gesteuert durch ein Ereignis am Eingang `REQ`. Nach dem Schreiben wird ein Bestätigungsereignis am Ausgang `CNF` gesendet.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar	Mit Variablen
REQ	Event	Normaler Ausführungsauftrag	IN, OUT

Ereignis-Ausgänge

Name	Typ	Kommentar	Mit Variablen
CNF	Event	Bestätigung der Ausführung	OUT

Daten-Eingänge

Name	Typ	Kommentar
IN	STRING	Der zu schreibende Wert

Daten-Ausgänge

Name	Typ	Kommentar
OUT	STRING	Zielvariable (In-Out) – nach der Ausführung identisch mit IN

Hinweis: `OUT` ist als In-Out-Variable deklariert. Sie dient gleichzeitig als Eingang (der aktuelle Wert vor der Ausführung) und als Ausgang (der neue Wert nach der Ausführung). Bei `REQ` wird `OUT := IN` gesetzt, so dass der vorherige Wert überschrieben wird.

Adapter

Keine Adapter vorhanden.

Funktionsweise

1. Ein ankommendes Ereignis am Eingang `REQ` startet die Verarbeitung.
2. Der im Algorithmus `REQ` hinterlegte Code wird ausgeführt:
   - `OUT := IN;`
     Der Wert der Eingangsvariablen `IN` wird in die In-Out-Variable `OUT` kopiert.
3. Anschließend wird das Ereignis am Ausgang `CNF` gesendet, um die erfolgreiche Ausführung zu signalisieren.

Technische Besonderheiten

- Der Baustein enthält nur einen einzigen Zustand (`REQ`), in dem die gesamte Verarbeitung stattfindet. Es handelt sich um einen **einfachen Funktionsblock** (SimpleFB) ohne sequentielle Zustandsautomaten.
- Die In-Out-Variable `OUT` erlaubt es, den Baustein direkt an eine persistent beschreibbare Variable (z. B. ein globales Datenobjekt oder eine gekoppelte Hardware-Adresse) anzubinden, ohne dass eine separate Ausgangsvariable notwendig ist.
- Der Algorithmus ist trivial und führt keinerlei Typ- oder Bereichsprüfungen durch.

Zustandsübersicht

Der Funktionsblock besitzt genau einen Zustand:

Zustand	Aktion	Ausgabeereignis	Beschreibung
`REQ`	`OUT := IN`	`CNF`	Kopiert den Eingangswert in die In-Out-Variable und bestätigt

Ein Startzustand ist nicht explizit definiert; der Baustein erwartet einen externen Ereignisimpuls auf `REQ`.

Anwendungsszenarien

- **Parametrierung von Zeichenketten in Steuerungen**: Setzen eines String-Werts in einer globalen Variable, z. B. eines Gerätenamens oder einer IP-Adresse.
- **Initialisierung von Textbausteinen**: Überschreiben einer standardmäßigen Zeichenkette in einem übergeordneten FB-Netzwerk.
- **Test- und Debugging-Umgebungen**: Gezieltes Vorgeben von Werten für nachfolgende Verarbeitungsschritte.

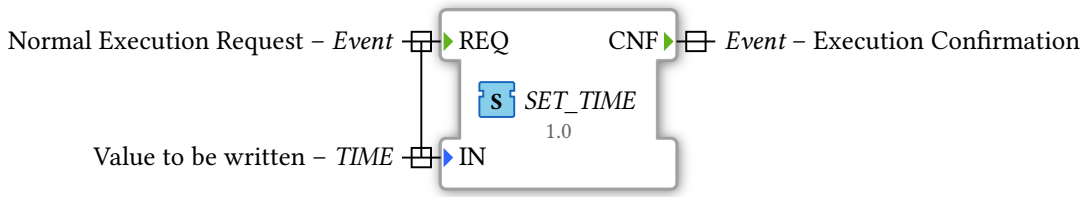
Vergleich mit ähnlichen Bausteinen

- **SET_BOOL, SET_INT, SET_REAL**: Gleichartig aufgebaute Funktionsblöcke für andere Datentypen. `SET_STRING` unterscheidet sich nur durch den Datentyp `STRING`.
- **MOVE**: Ein generischer Baustein, der Werte unterschiedlicher Datentypen kopieren kann. `SET_STRING` ist dagegen strikt auf `STRING` spezialisiert und arbeitet mit einer In-Out-Variable, während `MOVE` separate Eingänge und Ausgänge verwendet.
- **WRITE_STRING**: In einigen Bibliotheken vorhandene Blöcke mit ähnlicher Funktion, jedoch oft mit zusätzlichen Steuer- oder Fehlerausgängen ausgestattet.

Fazit

Der `SET_STRING`-Funktionsblock ist ein einfaches, aber effektives Werkzeug zum gezielten Setzen eines String-Werts in eine In-Out-Variable. Er eignet sich besonders für Anwendungen, in denen ein Wert direkt in eine bestehende Variable geschrieben werden muss, ohne dass eine Rücklese- oder Vergleichslogik erforderlich ist. Aufgrund seiner minimalen Schnittstelle und der klaren Semantik kann er leicht in komplexere Automatisierungslösungen integriert werden.

SET_TIME



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **SET_TIME** dient dazu, einen über den Dateneingang `IN` bereitgestellten Zeitwert (Typ `TIME`) in eine als InOut-Parameter übergebene Variable `OUT` zu schreiben. Der Baustein ist als einfacher, ereignisgesteuerter Block realisiert und bestätigt die Ausführung über den Ereignisausgang `CNF`.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung
REQ	Normaler Ausführungsrequest. Lösen den Algorithmus aus, der den Wert von IN nach OUT kopiert. Assoziierte Daten: IN, OUT.

Ereignis-Ausgänge

Ereignis	Beschreibung
CNF	Bestätigung der erfolgreichen Ausführung. Wird nach Abschluss des Algorithmus gesendet. Assoziierte Daten: OUT.

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung
IN	TIME	T#0s	Der Zeitwert, der in die Zielvariable geschrieben werden soll.

Daten-Ausgänge

Name	Typ	Initialwert	Beschreibung
OUT	TIME (InOut)	T#0s	Zielvariable, die beim Aufruf als InOut-Parameter übergeben wird. Der Wert von IN wird in diese Variable kopiert.

Hinweis: OUT ist ein **InOut**-Parameter, d.h. er dient gleichzeitig als Ein- und Ausgang. Der Aufrufer muss eine beschreibbare Variable bereitstellen.

Adapter

Keine vorhanden.

Funktionsweise

1. Der Funktionsblock wartet auf ein Ereignis am Eingang REQ.
2. Beim Eintreten von REQ wird der Algorithmus ausgeführt: Der aktuelle Wert des Dateneingangs IN wird der InOut-Variable OUT zugewiesen (OUT := IN).
3. Nach erfolgreicher Zuweisung wird das Ereignis CNF ausgelöst, um dem aufrufenden Baustein die Fertigstellung zu signalisieren.

Die Ausführung erfolgt in einem einzigen EC-Zustand (REQ), der den Algorithmus genau einmal durchläuft.

Technische Besonderheiten

- **InOut-Parameter:** OUT ist ein InOut-Datenpunkt. Dies ermöglicht es dem Baustein, direkt in eine vom Aufrufer bereitgestellte Variable zu schreiben, ohne einen zusätzlichen Datenausgang zu benötigen. Der Aufrufer muss sicherstellen, dass die referenzierte Variable existiert und beschreibbar ist.
- **Einfache Logik:** Der Baustein besteht aus nur einem EC-Zustand und einem ST-Algorithmus. Es gibt keine Verzweigungen, Timer oder internen Zustandsmaschinen.
- **Typisierung:** Sowohl IN als auch OUT sind vom Typ TIME, sodass nur Zeitwerte verarbeitet werden.

Zustandsübersicht

Der Funktionsblock besitzt genau einen Zustand:

Zustand	Beschreibung
REQ	Startzustand bei Initialisierung. Wird beim Eintreten von REQ aktiviert. Führt den Algorithmus aus und sendet CNF.

Es gibt keine weiteren Zustände. Der Baustein kehrt nach Abschluss des Algorithmus in den Startzustand zurück und wartet auf das nächste `REQ`-Ereignis.

Anwendungsszenarien

- **Setzen von Timer-Parametern:** Ein übergeordneter Steuerungsbaustein übergibt eine konkrete Verzögerungszeit (z.B. für einen `TON`- oder `TOF`-Baustein) über einen InOut-Parameter.
- **Initialisierung von Zeitvariablen:** In einer Startsequenz werden vordefinierte Zeitwerte in globale oder instanzspezifische Variablen geschrieben.
- **Kopieren von Zeitwerten:** Einmaliges Überschreiben einer Zeitvariablen mit einem berechneten oder konfigurierten Wert.
- **Parameter-Updates in Echtzeit:** Dynamisches Ändern einer Zeitreferenz, die von mehreren Bausteinen gemeinsam genutzt wird.

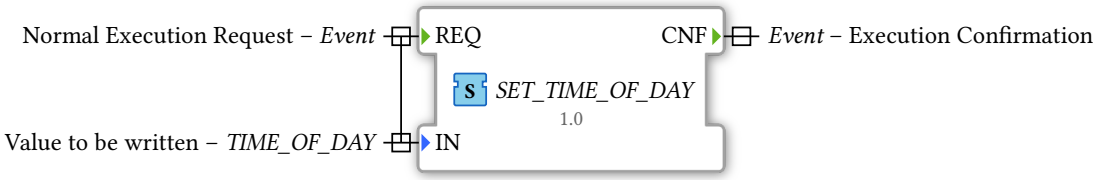
Vergleich mit ähnlichen Bausteinen

- **Einfache MOVE-Blöcke (z.B. `MOVE` für generische Typen):** SET_TIME ist speziell auf den Typ `TIME` optimiert und verwendet einen InOut-Parameter, was Speicherplatz spart und die Datenweitergabe effizienter macht.
- **Direkte Zuweisung über IEC-61499-Verbindungen:** Eine einfache Verbindung von einem `OUT`-Port eines anderen Bausteins zum `IN`-Port eines Zielbausteins könnte ebenfalls einen Wert setzen, erfordert aber oft zusätzliche Ereignis-Kopplung. SET_TIME vereinfacht dies durch die explizite `REQ`/`CNF`-Steuerung.
- **Setzen über globale Variablen:** Andere Bausteine schreiben über Dienstschnittstellen direkt in globale Variablen. SET_TIME hingegen arbeitet typischerweise auf instanzspezifischen InOut-Parametern, was die Kapselung verbessert.

Fazit

Der Funktionsblock **SET_TIME** ist ein einfacher, aber nützlicher Baustein zum Schreiben eines TIME-Wertes in eine als InOut referenzierte Variable. Seine klare Ereignissteuerung, der minimale Ressourcenverbrauch und die direkte Kopplung von Ein- und Ausgang über den InOut-Mechanismus machen ihn ideal für zeitkritische Zuweisungsaufgaben in der Automatisierungstechnik. Die Verwendung von Standard-IEC-61499-Mechanismen gewährleistet Interoperabilität und einfache Integration in bestehende Steuerungsprojekte.

SET_TIME_OF_DAY



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `SET_TIME_OF_DAY` dient dazu, einen über den Dateneingang `IN` bereitgestellten Tageszeitwert in eine als InOut-Variable deklarierte Zielvariable `OUT` zu schreiben. Der Baustein ist als einfacher FB (SimpleFB) realisiert und entspricht der Norm IEC 61499-1.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
REQ	Event	Normaler Ausführungsanforderung (gebunden an `IN` und `OUT`)

Ereignis-Ausgänge

Name	Typ	Kommentar

CNF	Event	Bestätigung der Ausführung (gebunden an OUT)

Daten-Eingänge

Name	Typ	Kommentar
IN	TIME_OF_DAY	Der zu schreibende Tageszeitwert

Daten-Ausgänge

Keine expliziten Daten-Ausgänge vorhanden. Die Zielgröße wird als InOut-Variable geführt.

Adapter

Keine.

Funktionsweise

Beim Eintreffen eines Ereignisses am Eingang REQ wird der Algorithmus REQ ausgeführt:

- Der Wert von IN wird der InOut-Variable OUT zugewiesen: OUT := IN.
- Nach erfolgreicher Zuweisung wird das Ereignis CNF ausgegeben, um den Abschluss der Operation zu quittieren.

Technische Besonderheiten

- Die Zielvariable OUT ist als **InOut-Variable** deklariert. Dies bedeutet, dass sie sowohl als Eingabe als auch als Ausgabe dient und von außen als Referenz übergeben werden muss. Der Wert von OUT wird durch die Zuweisung überschrieben.
- Der Baustein enthält nur einen Zustand und einen Algorithmus, was eine sehr einfache, deterministische Ausführung gewährleistet.
- Der Datentyp TIME_OF_DAY folgt der IEC 61499-Datentypdefinition für Tageszeiten (z. B. TOD#12:30:00).

Zustandsübersicht

Der Baustein besitzt einen einzigen Zustand:

- **REQ**: Wird beim Event REQ aktiviert. Nach Ausführung des Algorithmus wird CNF gesendet und der Zustand bleibt aktiv (kein Zustandswechsel).

Zustand	Aktion	Ausgangsereignis
REQ	Algorithmus REQ	CNF

Anwendungsspezifische Szenarien

- **Setzen einer globalen Tageszeit**: In einer Steuerung kann die aktuelle Uhrzeit von einem übergeordneten System übernommen und in eine zentrale InOut-Variable geschrieben werden.
- **Konfiguration von Zeitparametern**: Ein Benutzer gibt über eine HMI eine Tageszeit ein, die dann mit SET_TIME_OF_DAY in eine Konfigurationsvariable geschrieben wird.
- **Weitergabe von Zeitwerten**: Wenn ein Zeitwert aus einer Quelle gelesen und an mehrere Nachfolgebausteine weitergegeben werden muss, kann dieser FB als einfacher Schreibzugriff dienen.

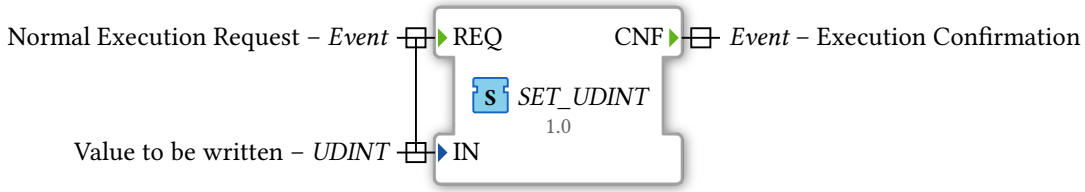
Vergleich mit ähnlichen Bausteinen

- **MOVE für elementare Typen**: Ähnliche Bausteine existieren für Basistypen wie BOOL, INT, REAL etc. SET_TIME_OF_DAY ist die spezifische Variante für TIME_OF_DAY.
- **Standard-MOVE-Bausteine**: In vielen IEC-61499-Bibliotheken gibt es generische MOVE- oder ASSIGN-Funktionsbausteine. Der vorgestellte Baustein ist eine dedizierte und typsichere Umsetzung für TIME_OF_DAY.

Fazit

SET_TIME_OF_DAY ist ein minimaler, aber nützlicher Funktionsbaustein zum Überschreiben einer TIME_OF_DAY-InOut-Variablen. Seine einfache Struktur macht ihn zuverlässig und leicht verständlich. In Kombination mit anderen Bausteinen kann er in Zeitsteuerungs- und Konfigurationsanwendungen effektiv eingesetzt werden.

SET_UDINT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **SET_UDINT** dient dazu, einen Wert vom Dateneingang `IN` auf die InOut-Variable `OUT` zu schreiben. Der Vorgang wird durch ein Ereignis am Eingang `REQ` ausgelöst und nach erfolgreicher Zuweisung wird der Ausgang `CNF` aktiviert. Der Baustein ist speziell für den UDINT-Datentyp ausgelegt und ermöglicht so eine effiziente und typsichere Datenübergabe an eine gemeinsame Variable.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Datentyp	Kommentar
`REQ`	Event	Normaler Ausführungsauftrag; triggert das Schreiben des Eingangswertes auf die Ausgangsvariable.

Mit `REQ` verknüpfte Daten: `IN`, `OUT`

Ereignis-Ausgänge

Ereignis	Datentyp	Kommentar
`CNF`	Event	Bestätigung der Ausführung; wird ausgegeben, sobald der Wert erfolgreich zugewiesen wurde.

Mit `CNF` verknüpfte Daten: `OUT`

Daten-Eingänge

Bezeichner	Datentyp	Initialwert	Kommentar
`IN`	UDINT	0	Wert, der auf die Zielvariable geschrieben werden soll.
`OUT` (InOut)	UDINT	0	Zielvariable, die sowohl als Eingang (lesbar) als auch als Ausgang (beschreibbar) fungiert.

Daten-Ausgänge

Bezeichner	Datentyp	Kommentar
`OUT` (InOut)	UDINT	Dieselbe Variable wie am Eingang; nach der Ausführung enthält sie den Wert von `IN`.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock arbeitet sehr einfach:

1. Ein Ereignis am Eingang `REQ` startet die Verarbeitung.
2. Der im Algorithmus `REQ` implementierte ST-Code führt die Zuweisung `OUT := IN;` aus.
3. Nach der Zuweisung wird das Ereignis am Ausgang `CNF` gesendet, um den erfolgreichen Abschluss zu signalisieren.

Die InOut-Variable `OUT` ist dabei sowohl lesbar als auch schreibbar. Der zugewiesene Wert bleibt so lange erhalten, bis ein erneuter `REQ` einen neuen Wert zuweist oder die Variable von außen verändert wird.

Technische Besonderheiten

- **Verwendung einer InOut-Variable**: `OUT` ist als InOut deklariert. Dadurch kann der Baustein direkt auf eine externe Variable zugreifen und sie ändern, ohne dass ein eigener Datenausgang notwendig ist. Der Wert ist nach dem Schreibvorgang sowohl innerhalb als auch außerhalb des Bausteins sichtbar.
- **Einfache Logik**: Kein Zwischenspeicher, keine Verzögerungen – der Vorgang ist atomar und erfolgt in einem einzigen Schritt.
- **Typbindung**: Der Baustein ist ausschließlich für den Typ `UDINT` (unsigned 32-Bit Integer) definiert, was typsichere Verwendung garantiert.

Zustandsübersicht

Der Baustein enthält einen einzigen Zustand `REQ` (Ereignisverarbeitung):

Zustand	Aktion	Ausgangsereignis
`REQ`	Führe Algorithmus `REQ` aus (OUT := IN)	CNF

Nach Ausführung von `CNF` kehrt der Baustein in den Grundzustand zurück und wartet auf das nächste `REQ`-Ereignis.

Anwendungsszenarien

- **Initialisierung von Variablen**: Setzen eines UDINT-Wertes auf einen vorgegebenen Startwert zur Laufzeit.
- **Steuerung von Parametern**: Überschreiben von Konfigurationswerten in einer laufenden Anwendung (z. B. Sollwerte, Zählerstände).
- **Datenweitergabe zwischen Logikbausteinen**: Nutzung einer gemeinsamen InOut-Variablen, um Daten ohne zusätzliche Zwischenspeicher zu aktualisieren.
- **Ersatz für einfache Move-Funktionen**: Ähnlich einem MOVE-Baustein, jedoch mit explizitem UDINT-Fokus und InOut-Mechanik.

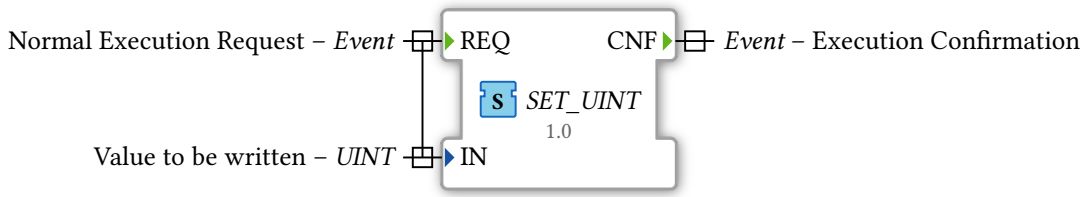
Vergleich mit ähnlichen Bausteinen

- **MOVE (Generic)**: Ein allgemeiner MOVE-Baustein kann Werte beliebigen Typs kopieren, erfordert jedoch separate Ein-/Ausgänge. `SET_UDINT` nutzt dagegen InOut, was die Verkabelung reduziert und den Zugriff auf die ursprüngliche Variable ermöglicht.
- **SET_BOOL, SET_INT, etc.**: Typvarianten desselben Konzepts für unterschiedliche Datentypen. `SET_UDINT` ist speziell auf 32-Bit-Ganzzahlen ohne Vorzeichen optimiert.
- **R_TRIG / F_TRIG**: Diese Flankenerkennungen setzen keine Werte, sondern detektieren Änderungen. `SET_UDINT` agiert direkt wertschreibend.

Fazit

Der Funktionsblock `SET_UDINT` ist ein schlanker, zuverlässiger Baustein zum Beschreiben einer InOut-Variablen mit einem UDINT-Wert. Seine einfache Struktur (ein Ereignis, eine Zuweisung) macht ihn leicht verständlich und debug-freundlich. Die Verwendung einer InOut-Schnittstelle spart Verkabelungsaufwand und ermöglicht direkte Manipulation externer Variablen. Typisch für IEC 61499 ist er ideal in Automatisierungsprojekten einsetzbar, in denen schnelle und typsichere Wertzuweisungen gefordert sind.

SET_UINT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock SET_UINT dient dazu, einen UINT-Wert, der an einem Dateneingang anliegt, in eine als InOut-Parameter deklarierte Variable zu schreiben. Diese einfache aber grundlegende Operation erlaubt es, Daten aus dem FB-Kontext in eine externe oder referenzierte Variable zu kopieren.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung
REQ	Event	Normaler Ausführungsbefehl

Ereignis-Ausgänge

Name	Typ	Beschreibung
CNF	Event	Bestätigung der Ausführung

Daten-Eingänge

Name	Typ	Initialwert	Beschreibung
IN	UINT	0	Wert, der geschrieben wird

Daten-Ausgänge

Keine separaten Ausgänge vorhanden. Der Zielwert wird über die InOut-Variable `OUT` bereitgestellt.

Adapter

Keine Adapter.

Funktionsweise

Die Verarbeitung erfolgt in einem Schritt:

1. Ein Ereignis am Eingang `REQ` triggert die Ausführung des Algorithmus.
2. Der Algorithmus kopiert den Wert des Dateneingangs `IN` in die InOut-Variable `OUT`.
3. Nach Abschluss wird das Ereignis `CNF` ausgegeben.

Die InOut-Variable `OUT` wird dabei als referenzierte Variable interpretiert – der FB schreibt direkt in die Speicherstelle, die beim Aufruf mit der `OUT`-Schnittstelle verbunden ist.

Technische Besonderheiten

- **InOut-Variable als Ziel:** Der FB verwendet eine InOut-Variable (`OUT`) anstelle eines reinen Ausgangs. Dadurch kann der Baustein in Aufrufumgebungen (z. B. innerhalb von Ressourcen oder anderen FBs) direkt auf eine bestehende Variable zugreifen und diese überschreiben.
- **Einzeiliger Algorithmus:** Die Zuweisung `OUT := IN` erfolgt in einem einzigen ST-Befehl; es werden keine weiteren Berechnungen oder Zwischenspeicherungen durchgeführt.
- **Keine Zustandsautomaten:** Der FB besitzt nur einen Einstiegszustand (`REQ`), eine Zustandsänderung oder Schleifen sind nicht vorgesehen.

Zustandsübersicht

Zustand	Aktion	Ausgabe
REQ	`OUT := IN`	CNF

Der FB bleibt nach Ausführung in einem impliziten Endzustand; er ist für jede neue `REQ`-Anforderung erneut bereit.

Anwendungsszenarien

- **Setzen eines Konfigurationswertes:** Ein Sollwert oder Parameter (z. B. ein Schwellwert in einem Regelungsbaustein) wird von außen hereingegeben und in die entsprechende interne Variable geschrieben.

- **Kopieren von UINT-Daten an eine zentrale Speicherstelle:** In einer Steuerung kann `SET_UINT` verwendet werden, um Messwerte oder Berechnungsergebnisse in einen globalen Datenbaustein zu übertragen.
- **Austausch von Daten zwischen FBs:** Über die InOut-Variable können zwei verschiedene FBs auf die gleiche Variable zugreifen – der `SET_UINT`-FB dient dabei dem gezielten Überschreiben.

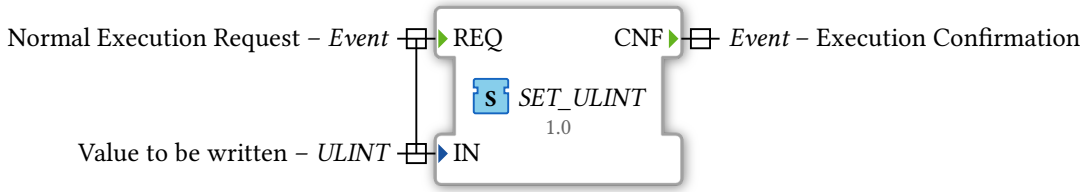
Vergleich mit ähnlichen Bausteinen

- **SET_BOOL / SET_INT / SET_*:** Diese Bausteine unterscheiden sich lediglich im verwendeten Datentyp. Die Funktionsweise – „Wert von IN nach OUT schreiben" – ist identisch.
- **MOVE (z. B. `MOVE` in IEC 61131-3):** Eine allgemeine Kopierfunktion, die jedoch meist als Standardfunktion (ohne eigenes Event-Interface) realisiert ist. Der `SET_UINT`-FB bietet die explizite Ereignissteuerung für die Ausführung.
- **Sonderfall InOut:** Im Gegensatz zu reinen Ausgangsbausteinen erlaubt `SET_UINT` den direkten Zugriff auf die Zielvariable, was bei der Datenweitergabe über Hierarchiegrenzen hinweg Vorteile bringt.

Fazit

Der FB `SET_UINT` ist ein minimaler, aber nützlicher Baustein, um einen UINT-Wert zielgerichtet in eine referenzierte Variable zu schreiben. Seine einfache Struktur – ein Ereignis löst eine Zuweisung aus – macht ihn besonders in Umgebungen nützlich, in denen ein explizites „Setzen" einer Variable über ein Ereignis gefordert wird. Erfüllt er keine weiteren Bedingungen, ist er als Musterbaustein für ähnliche Datentyp-spezifische Set-Funktionen geeignet.

SET_ULINT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **SET_ULINT** dient dazu, einen unsignierten 64-Bit-Ganzzahlwert (ULINT) von einem Dateneingang auf eine InOut-Variable zu schreiben. Er wird typischerweise verwendet, um Werte in verbundene Variablen oder globale Speicherstellen zu setzen, die auch von anderen Bausteinen gelesen werden können.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung
REQ	Normaler Ausführungsauftrag. Der Wert von `IN` wird auf die InOut-Variable `OUT` übertragen und nach Abschluss der Ereignisausgang `CNF` gesendet.

Ereignis-Ausgänge

Ereignis	Beschreibung
CNF	Bestätigung der Ausführung. Wird ausgelöst, sobald die Zuweisung von `IN` an `OUT` erfolgt ist.

Daten-Eingänge

Name	Datentyp	Beschreibung
IN	ULINT	Der zu schreibende Wert. Voreinstellung: `0`.

Daten-Ausgänge

Name	Datentyp	Beschreibung
OUT	ULINT	InOut-Variable – Ziel der Zuweisung. Sie kann sowohl gelesen als auch beschrieben werden und steht nach der Ausführung auf dem neuen Wert.

Adapter

Keine.

Funktionsweise

Der Funktionsblock ist als einfacher Zustandsautomat (SimpleFB) realisiert. Ein einziger Zustand `REQ` führt bei Aktivierung den Algorithmus `REQ` aus:

```
OUT := IN;
```

Anschließend wird der Ereignisausgang `CNF` gesendet. Der übergebene Wert `IN` wird also direkt in die InOut-Variable `OUT` kopiert. Da `OUT` als InOut deklariert ist, kann der Baustein den vorherigen Wert von `OUT` nicht lesen – es erfolgt eine reine Überschreibung.

Technische Besonderheiten

- Der Baustein verwendet eine **InOut-Variable** (`OUT`), die im IEC 61499-Modell sowohl als Eingangs- als auch als Ausgangsport fungiert. Dies ermöglicht direkte Verbindungen zu Variablen, die von mehreren Bausteinen gemeinsam genutzt werden.
- Das Paket `logiBUS::utils::conversion::provide` im CompilerInfo zeigt an, dass der FB für die Nutzung im logiBUS-Ökosystem vorgesehen ist.
- Der Algorithmus ist sehr einfach gehalten – lediglich eine ST-Zuweisung – und daher extrem schnell ausführbar.
- Der initiale Wert von `OUT` ist `0`, sodass bei erstmaliger Nutzung ohne vorherige Initialisierung ein definierter Startwert vorliegt.

Zustandsübersicht

Der FB besitzt genau einen EC-Zustand:

Zustand	Aktion	Ausgabe
REQ	`REQ` (OUT := IN)	CNF

Nach dem Start oder Reset befindet sich der FB im Zustand `REQ`. Jeder `REQ`-Ereignisimpuls löst die Aktion aus und erzeugt danach `CNF`.

Anwendungsszenarien

- **Parametriert von Echtzeitsystemen**: Setzen von Sollwerten oder Konfigurationsparametern in einer Variablen, die von mehreren Funktionsblöcken überwacht wird.
- **Globale Statusvariablen**: Schreiben von Fehlercodes oder Betriebszuständen in eine zentrale `ULINT`-Variable.
- **Test- und Diagnosezwecke**: Gezieltes Überschreiben von Werten in einer laufenden Steuerung zu Analysezwecken.

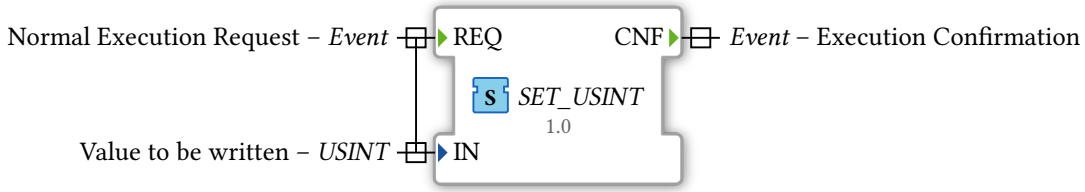
Vergleich mit ähnlichen Bausteinen

- **SET_INT, SET_DINT, SET_LINT**: Gleichartige Bausteine für andere Ganzzahltypen (INT, DINT, LINT). Die Funktionsweise ist identisch, nur der Datentyp unterschiedlich.
- **MOVE_ULINT**: Viele Bibliotheken enthalten einen MOVE-Baustein, der einen Wert von einem Eingang auf einen Ausgang kopiert. `SET_ULINT` unterscheidet sich durch die Verwendung einer **InOut-Variable**, die es erlaubt, direkt auf eine bestehende Variable zuzugreifen, ohne den Umweg über einen reinen Ausgang.

Fazit

Der **SET_ULINT**-Funktionsblock ist ein einfaches, aber effektives Werkzeug zum Setzen von ULINT-Werten in gemeinsam genutzten Variablen. Seine InOut-Schnittstelle macht ihn besonders geeignet für Anwendungen, in denen Werte überschrieben werden müssen, ohne dass der vorherige Zustand ausgewertet wird. Aufgrund seiner schlanken Implementierung eignet er sich gut für zeitkritische Steuerungen.

SET_USINT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **SET_USINT** dient dazu, einen über den Dateneingang **IN** bereitgestellten Wert vom Typ **USINT** direkt in die als InOut deklarierte Variable **OUT** zu schreiben. Der Baustein führt eine einfache Zuweisung aus und bestätigt die Ausführung durch das Ereignis **CNF**.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung
REQ	Normaler Ausführungsanforderung – übernimmt den Wert von **IN** nach **OUT** und sendet **CNF**.

Ereignis-Ausgänge

Ereignis	Beschreibung
CNF	Bestätigung der erfolgreichen Ausführung (wird nach der Zuweisung gesendet).

Daten-Eingänge

Name	Datentyp	Initialwert	Beschreibung
IN	USINT	0	Wert, der in die Zielvariable geschrieben werden soll.

Daten-Ausgänge

Name	Datentyp	Initialwert	Beschreibung
OUT	USINT (InOut)	0	Zielvariable (als InOut deklariert, d. h. sie kann sowohl gelesen als auch geschrieben werden). Nach der Ausführung enthält **OUT** den Wert von **IN**.

Adapter

Keine.

Funktionsweise

Wird das Ereignis **REQ** empfangen, führt der FB den Algorithmus **REQ** aus. Dieser weist den aktuellen Wert des Eingangs **IN** der InOut-Variable **OUT** zu (`OUT := IN;`). Anschließend wird das Ereignis **CNF** gesendet, um den Abschluss der Operation zu signalisieren. Der Baustein arbeitet ohne Verzögerung oder weitere Bedingungen.

Technische Besonderheiten

- Die Verwendung einer **InOut**-Variable erlaubt es, **OUT** sowohl als Ziel (Ausgang) als auch als Quelle (Eingang) in anderen Bausteinen zu verwenden.
- Der Baustein ist auf den Datentyp **USINT** (8-Bit vorzeichenloser Integer) festgelegt – eine direkte Konvertierung ist nicht vorgesehen.
- Es existiert nur ein einziger Zustand; der FB ist **ereignisgesteuert** und führt pro **REQ** genau eine Zuweisung aus.

Zustandsübersicht

Zustand	Aktion	Ausgangsereignis
REQ	Ausführen des Algorithmus **REQ** (`OUT := IN;`)	**CNF**

Der FB besitzt keinen expliziten Startzustand – er wartet im Zustand **REQ** auf ein Ereignis.

Anwendungsszenarien

- **Initialisierung** von Variablen oder Peripherie, die einen bestimmten USINT-Wert benötigen.
- **Parametrierung** von Geräten oder Funktionsbausteinen über eine InOut-Variable.
- **Einfaches Setzen eines Wertes** in einer Steuerungslogik, wenn keine Berechnung oder Bedingung erforderlich ist.

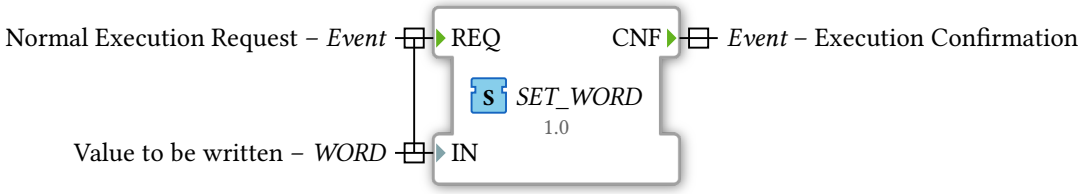
Vergleich mit ähnlichen Bausteinen

- **MOVE (ANY)**: Allgemeinerer Baustein zum Kopieren beliebiger Datentypen, oft ohne InOut-Fähigkeit.
- **SET_INT**, **SET_BYTE**, etc.: Analoge Bausteine für andere Integer-Typen – SET_USINT ist auf USINT spezialisiert.
- Im Gegensatz zu **LATCH** oder **SR-Flipflops** speichert SET_USINT keinen Zustand, sondern führt eine einmalige Zuweisung durch.

Fazit

SET_USINT ist ein minimalistischer und zuverlässiger Baustein zur direkten Wertzuweisung an eine InOut-Variable vom Typ USINT. Seine einfache Struktur eignet sich besonders für klare, sequenzielle Steuerungsaufgaben, bei denen ein Wert ohne Nebenwirkungen gesetzt werden muss.

SET_WORD



• – ◦ • – ◦ • – ◦ •

Einleitung

Der SET_WORD Funktionsbaustein schreibt einen WORD-Wert von seinem Dateneingang IN in eine InOut-Variable OUT. Er wird verwendet, um einen Ausgang oder eine global zugängliche WORD-Variable mit einem bestimmten Wert zu überschreiben.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Event): Normaler Ausführungsauftrag. Der Baustein wird ausgelöst, wenn ein Ereignis an diesem Eingang eintrifft.
  Mit den Variablen `IN` und `OUT` verbunden.

Ereignis-Ausgänge

- **CNF** (Event): Bestätigung der Ausführung. Wird nach erfolgreicher Zuweisung gesendet.
  Mit der Variable `OUT` verbunden.

Daten-Eingänge

- **IN** (WORD): Der Wert, der in die Zielvariable geschrieben werden soll. Initialwert: `0`.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden. Die Ausgabe erfolgt ausschließlich über die InOut-Variable `OUT`.

Adapter

Keine.

Funktionsweise

Bei einem Ereignis am Eingang **REQ** wird der Algorithmus ausgeführt:
`OUT := IN;`
Anschließend wird ein Bestätigungsereignis am Ausgang **CNF** gesendet. Der Baustein überschreibt den aktuellen Wert von `OUT` mit dem Wert von `IN`.

Technische Besonderheiten

- `OUT` ist als **InOut-Variable** deklariert. Sie dient gleichzeitig als Eingabe (lesend) und Ausgabe (schreibend). Der vorherige Wert von `OUT` wird durch die Zuweisung überschrieben.
- Es gibt keine eigenen Datenausgänge – die Änderung wirkt direkt auf die verbundene Variable.
- Der Baustein ist typspezifisch auf `WORD` ausgelegt. Für andere Datentypen stehen entsprechende Varianten zur Verfügung.

Zustandsübersicht

Der Baustein besitzt einen aktiven Zustand:

- **REQ**: Beim Eintritt in diesen Zustand wird die Zuweisung `OUT := IN` ausgeführt und das Ereignis **CNF** ausgelöst.

Es gibt keine weiteren Zustände oder Verzweigungen.

Anwendungsszenarien

- Setzen eines WORD-Ausgangs auf einen berechneten oder festen Wert.
- Überschreiben einer globalen WORD-Variablen in einer SPS-Steuerung.
- Einmalige Initialisierung einer WORD-Variablen durch einen externen Baustein.

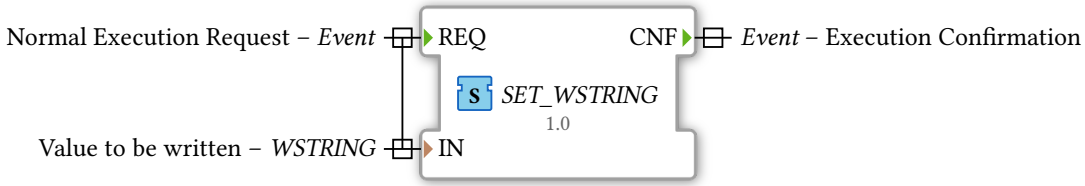
Vergleich mit ähnlichen Bausteinen

- **MOVE**: Kopiert einen Wert von einem Eingang zu einem Ausgang (separate Ein-/Ausgänge). SET_WORD verwendet stattdessen eine InOut-Variable.
- **SET_BOOL**: Analoge Funktionalität für den Datentyp BOOL.
- **F_MOVE**: Funktionsbaustein zur allgemeinen Datenübergabe, oft mit mehreren Eingängen/Ausgängen. SET_WORD ist spezifischer und direkter.

Fazit

SET_WORD ist ein einfacher und effizienter Baustein, um einen WORD-Wert direkt in eine InOut-Variable zu schreiben. Durch die Verwendung eines InOut-Parameters reduziert er die Anzahl der benötigten Variablen und ist besonders in Szenarien nützlich, in denen ein Wert überschrieben werden muss, ohne einen separaten Ausgang zu nutzen.

SET_WSTRING



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein **SET_WSTRING** dient dazu, einen Wert vom Datentyp `WSTRING` an einen InOut-Parameter zu übertragen. Er wird typischerweise eingesetzt, um einen berechneten oder extern vorgegebenen Zeichenkettenwert in eine übergeordnete Variable (z. B. einer anderen Instanz oder eines übergeordneten Bausteins) zu schreiben.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung

REQ	Normaler Ausführungsauftruf: Die Zuweisung des Eingabewerts an die InOut-Variable wird ausgelöst.

Ereignis-Ausgänge

Ereignis	Beschreibung
CNF	Bestätigung der Ausführung. Wird nach erfolgreicher Zuweisung gesendet.

Daten-Eingänge

Name	Datentyp	Beschreibung
IN	WSTRING	Quellwert, der in die Zielvariable geschrieben werden soll.

Daten-Ausgänge

Der Baustein besitzt keine klassischen Datenausgänge. Die Weitergabe des Werts erfolgt über die **InOut-Variable** OUT.

Adapter

Keine.

Funktionsweise

Wird der Ereigniseingang REQ aktiviert, führt der Baustein die folgende Zuweisung aus:
`OUT := IN;`
Der am Eingang IN anliegende WSTRING-Wert wird unverändert in die InOut-Variable OUT kopiert. Anschließend wird das Ereignis CNF gesendet, um dem aufrufenden Baustein die Beendigung der Operation zu signalisieren.

Technische Besonderheiten

- **InOut-Variable**: OUT ist als InOut-Parameter deklariert. Dies bedeutet, dass die Variable sowohl gelesen als auch beschrieben werden kann und ihre Lebensdauer außerhalb des Bausteins liegt. Der Baustein überschreibt den aktuellen Inhalt von OUT mit dem Wert von IN.
- **Datentyp WSTRING**: Es handelt sich um einen Zeichenketten-Typ, der Breitzeichen (Unicode) unterstützt. Der Baustein führt keine Längenprüfung oder Konvertierung durch – die Zuweisung erfolgt direkt.
- **Einfachster Zustandsautomat**: Der Zustandsautomat besteht aus einem einzigen Zustand (REQ), der die gesamte Funktionalität abdeckt. Es gibt keine Verzweigungen oder Zeitverzögerungen.

Zustandsübersicht

Der Baustein enthält genau einen Zustand:

Zustand	Beschreibung
REQ	Wird nach dem Eintreffen des Ereignisses REQ aktiv. Führt die Zuweisung OUT := IN aus und sendet CNF. Danach kehrt der Baustein in den initialen Zustand zurück.

Anwendungsszenarien

- **Übergabe eines formatierten Strings**: Angenommen, ein Baustein erzeugt eine Fehlermeldung als WSTRING und möchte diese in eine übergeordnete Variable schreiben. SET_WSTRING übernimmt diese Zuweisung auf einen Trigger hin.
- **Parametrierung**: In einer Steuerung kann ein externer Parameter (z. B. eine Produktbezeichnung) über SET_WSTRING in einen globalen Datenbaustein kopiert werden.
- **Debugging**: Temporäres Setzen einer WSTRING-Variablen durch eine Test- oder Initialisierungslogik.

Vergleich mit ähnlichen Bausteinen

- **SET_BOOL**: Arbeitet analog, jedoch mit dem Datentyp BOOL. Beide nutzen ein InOut, um Werte in externe Variablen zu schreiben.

- **MOVE (IEC-Baustein)**: Der Standard-MOVE-Baustein kopiert Daten von einem Eingang auf einen Ausgang. Im Unterschied dazu verwendet `SET_WSTRING` einen InOut-Parameter, was die Anbindung an bestehende Variablen ohne zusätzlichen Ausgangsport erlaubt.
- **WRITE_STRING**: Manche Bibliotheken bieten spezielle Write-Bausteine mit Längenkontrolle. `SET_WSTRING` ist bewusst einfach gehalten und verzichtet auf Prüfungen.

Fazit

Der Baustein **SET_WSTRING** ist eine minimalistische, aber effektive Lösung zum Beschreiben einer WSTRING-InOut-Variablen. Seine einfache Zustandsmaschine und die klare Schnittstelle machen ihn leicht verständlich und einsetzbar, wo eine triviale Zuweisung auf ein Ereignis hin benötigt wird. Für Aufgaben mit zusätzlichen Funktionen (Konvertierung, Längenbeschränkung) sind spezialisierte Bausteine vorzuziehen.

str

🎧 Podcast

- 4diac IDE: Dein Open-Source-Werkzeugkasten für verteilte Industrieautomatisierung nach IEC 61499
- 4diac IDE: Wie der IEC 61499 Standard die Industrieautomatisierung revolutioniert
- 4diac-Präsentation: Zielgruppen, Struktur und Alleinstellungsmerkmal Schärfen
- Den Software-Drachen zähmen: Industrielle Automatisierung und die Zukunft der Produktion
- Eclipse 4diac FORTE: IEC 61499 verstehen – Der LEGO®-Baukasten für Ihre Industrie 4.0 Steuerung

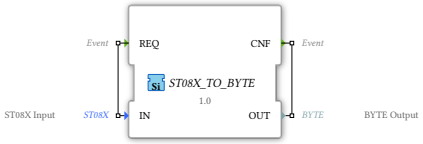
---

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

assembling

ST08X_TO_BYTE



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `ST08X_TO_BYTE` dient der Konvertierung einer speziellen Datenstruktur, die acht einzelne boolesche Werte (BOOL) enthält, in einen einzigen Byte-Wert (BYTE). Diese Operation ist nützlich, um kompakte digitale Signale für die Übertragung oder Speicherung zu bündeln oder um sie mit Systemen zu verbinden, die Daten im Byte-Format erwarten.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Dieses Ereignis löst die Konvertierungsfunktion aus. Bei seinem Eintreffen werden die aktuellen Werte am Dateneingang `IN` gelesen und in den Byte-Ausgang `OUT` umgewandelt.

Ereignis-Ausgänge

- **CNF**: Dieses Ereignis signalisiert die erfolgreiche Beendigung der Konvertierung. Es wird gleichzeitig mit der Ausgabe des berechneten Byte-Wertes am Datenausgang `OUT` erzeugt.

Daten-Eingänge

- **IN** (Typ: `ST08X`): Der Eingang für die zu konvertierende Datenstruktur. Der Typ `ST08X` ist ein benutzerdefinierter Strukturtyp (STRUCT), der acht einzelne BOOL-Elemente mit den Namen `X_00` bis `X_07` enthält.

Daten-Ausgänge

- **OUT** (Typ: `BYTE`): Der Ausgang, an dem das Ergebnis der Konvertierung bereitgestellt wird. Die acht Bits des Bytes (Bit 0 bis Bit 7) entsprechen direkt den acht BOOL-Werten der Eingangsstruktur.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei Auslösung durch das `REQ`-Ereignis führt der Block eine bitweise Zuweisung durch. Jedes Bit des Ausgangsbytes `OUT` wird auf den Wert des entsprechenden booleschen Elements in der Eingangsstruktur `IN` gesetzt. Die Zuordnung erfolgt nach folgendem, fest definiertem Schema:

- `OUT`.0 (niederwertigstes Bit, LSB) = `IN.X_00`
- `OUT`.1 = `IN.X_01`
- `OUT`.2 = `IN.X_02`
- `OUT`.3 = `IN.X_03`
- `OUT`.4 = `IN.X_04`
- `OUT`.5 = `IN.X_05`
- `OUT`.6 = `IN.X_06`
- `OUT`.7 (höchstwertiges Bit, MSB) = `IN.X_07`

Nach Abschluss dieser Zuweisungen wird das `CNF`-Ereignis zusammen mit dem neuen Wert von `OUT` ausgegeben.

Technische Besonderheiten

- **Deterministisches Mapping**: Die Zuordnung der einzelnen BOOL-Elemente zu bestimmten Bit-Positionen im Byte ist fest im Code hinterlegt und nicht konfigurierbar.
- **Einfache Logik**: Der Block enthält keine Zustandsautomaten oder komplexe Algorithmen. Die Funktionsweise beschränkt sich auf eine direkte, parallele Zuweisung.
- **Typabhängigkeit**: Der Block ist spezifisch für den benutzerdefinierten Datentyp `ST08X`. Für andere Strukturen oder eine andere Anzahl von Bits sind ggf. andere Blöcke erforderlich.

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand im Sinne eines Zustandsautomaten. Er verhält sich rein kombinatorisch in Bezug auf die Daten und sequentiell in Bezug auf die Ereignissteuerung: Auf ein eingehendes `REQ` folgt stets ein ausgehendes `CNF`.

Anwendungsszenarien

- **Kommunikation mit Byte-orientierten Schnittstellen**: Vorbereitung von Daten für die Übertragung über serielle Bussysteme (z.B. Modbus, Profibus), Netzwerke oder Kommunikationsprotokolle, die Daten in Byte-Blöcken erwarten.
- **Kompakte Speicherung**: Bündeln mehrerer digitaler Schaltzustände (z.B. von Grenzwertmeldern, Maschinenzuständen) in einem einzelnen Byte, um Speicherplatz zu optimieren.
- **Datenaufbereitung für Visualisierung oder Logging**: Umwandlung von strukturierten BOOL-Signalen in ein für Anzeigesysteme oder Datenspeicher einfacher zu handhabendes Byte-Format.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`BOOL_TO_BYTE` (oder ähnliche Blöcke aus Standardbibliotheken)**: Viele IEC 61499- oder IEC 61131-3-Bibliotheken bieten Blöcke an, die acht separate `BOOL`-Eingänge direkt in ein `BYTE` umwandeln. `ST08X_TO_BYTE` unterscheidet sich dadurch, dass die Eingangswerte bereits in einer strukturierten Variable (`ST08X`) gebündelt vorliegen, was die Verdrahtung im Anwendungsdiagramm vereinfachen kann.
- **`PACK` / `UNPACK`-Blöcke**: Allgemeinere Blöcke zum Packen und Entpacken von Daten. `ST08X_TO_BYTE` ist eine spezialisierte und typsichere Implementierung für genau diesen einen Anwendungsfall.
- **Manuelle Bit-Operationen**: Die Funktion könnte auch durch direkte Zuweisungen in Structured Text (ST) realisiert werden. Der Funktionsblock kapselt diese Logik und bietet eine standardisierte, wieder verwendbare Schnittstelle mit klarer Ereignissteuerung.

Fazit

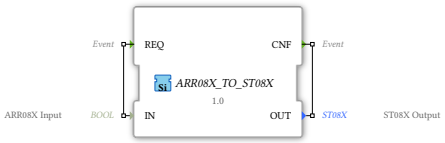
Der `ST08X_TO_BYTE`-Funktionsblock ist ein spezialisiertes und effizientes Werkzeug zur Konvertierung einer strukturierten Gruppe von acht booleschen Signalen in ein kompaktes Byte. Durch seine klare, fest definierte Schnittstelle und einfache Funktionsweise eignet er sich ideal für Anwendungen, bei denen digitale Signale für die effiziente Kommunikation oder Speicherung gebündelt werden müssen. Seine Verwendung fördert die Wiederverwendbarkeit und Lesbarkeit von Steuerungsprogrammen im Vergleich zu manuell implementierten Bit-Operationen.

converting

Numerischer Index (converting)

Hier finden Sie alle Übungen des Arbeitsbereichs `converting` in aufsteigender Reihenfolge.

ARR08X_TO_ST08X



• — ◦ • — ◦ • — ◦ •

Einleitung

Dieser Funktionsblock (FB) dient dazu, ein Array von 8 BOOL-Werten (ARR08X) in eine entsprechende Struktur von 8 BOOL-Werten (ST08X) umzuwandeln. Er ermöglicht eine einfache Konvertierung zwischen diesen beiden Datenformaten.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Ein Ereignis, das die Konvertierung auslöst. Wenn REQ empfangen wird, wird der Wert des Eingangs `IN` verarbeitet und das Ergebnis am Ausgang `OUT` bereitgestellt.

Ereignis-Ausgänge

- **CNF**: Ein Bestätigungsereignis, das ausgelöst wird, sobald die Konvertierung abgeschlossen ist und der Ausgang `OUT` aktualisiert wurde.

Daten-Eingänge

- **IN**: Ein Array von 8 BOOL-Werten (BOOL[0..7]), das die Quelldaten für die Konvertierung darstellt.

Daten-Ausgänge

- **OUT**: Eine Struktur vom Typ `logiBUS::utils::conversion::types::ST08X`, die die konvertierten 8 BOOL-Werte enthält.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Der Funktionsblock `ARR08X_TO_ST08X` führt eine direkte elementweise Zuweisung der Werte aus dem Eingangs-Array `IN` auf die entsprechenden Felder der Ausgangsstruktur `OUT` durch. Bei jedem `REQ`-Ereignis wird der Wert des `IN`-Arrays gelesen und die einzelnen Elemente des Arrays `IN[0]` bis `IN[7]` den Feldern `X_00` bis `X_07` der `ST08X`-Struktur zugewiesen. Nach der Zuweisung wird das `CNF`-Ereignis ausgelöst.

Die Zuweisungen erfolgen wie folgt:

- `OUT.X_00 := IN[0];`
- `OUT.X_01 := IN[1];`
- `OUT.X_02 := IN[2];`
- `OUT.X_03 := IN[3];`

- `OUT.X_04 := IN[4];`
- `OUT.X_05 := IN[5];`
- `OUT.X_06 := IN[6];`
- `OUT.X_07 := IN[7];`

Technische Besonderheiten

- **Datentyp-Konvertierung**: Der Baustein spezialisiert sich auf die Konvertierung eines generischen BOOL-Arrays in eine spezifische BOOL-Struktur vom Typ `ST08X`. Dieser Strukturtyp muss im System bekannt sein (mittels `IMPORT logiBUS::utils::conversion::types::ST08X;`).
- **Feste Größe**: Der Baustein ist für Arrays und Strukturen mit genau 8 BOOL-Elementen ausgelegt. Er ist nicht variabel für andere Array-Größen.
- **Determinismus**: Die Konvertierung ist direkt und deterministisch. Die Ausführung erfolgt sofort bei Empfang des `REQ`-Ereignisses.

Zustandsübersicht

Der Funktionsblock `ARR08X_TO_ST08X` ist ein zustandsloser (stateless) Baustein. Er speichert keine internen Zustände zwischen den Aufrufen. Jede Ausführung des Bausteins ist unabhängig von vorherigen Ausführungen und basiert ausschließlich auf den aktuellen Eingangswerten.

Anwendungsszenarien

- **Schnittstellenanpassung**: Ideal, um Datenformate zwischen verschiedenen Systemteilen anzupassen, insbesondere wenn ein Teil ein Array und ein anderer eine Struktur bevorzugt oder erwartet.
- **Vereinheitlichung von Datenzugriffen**: Ermöglicht den Zugriff auf einzelne Bool-Werte über benannte Felder (`X_00`, `X_01`, etc.) anstatt über Array-Indizes, was die Lesbarkeit des Codes verbessern kann.
- **Visualisierung**: Vereinfacht die Anbindung an Visualisierungssysteme, die möglicherweise eine strukturierte Darstellung von Bool-Signalen gegenüber einem Array bevorzugen.
- **Legacy-Systeme**: Kann als Brücke zwischen älteren Systemen, die eventuell Bool-Arrays verwenden, und neueren Komponenten, die mit Strukturen arbeiten, dienen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Manuelle Zuweisung**: Die Funktion könnte auch durch direkte Zuweisungen im Code des Steuerprogramms (z.B. in einem `ST`-Baustein) realisiert werden. Der `ARR08X_TO_ST08X` kapselt diese Logik jedoch in einem wiederverwendbaren Funktionsblock, was die Modularität und Übersichtlichkeit erhöht.
- **Generische Konverter**: Im Gegensatz zu einem generischen Konverter, der beliebige Array-Größen oder Typen verarbeiten könnte, ist dieser Baustein spezifisch auf 8-BOOL-Arrays und die `ST08X`-Struktur zugeschnitten. Dies vereinfacht die Implementierung, macht ihn aber weniger flexibel für andere Größen.

Fazit

Der `ARR08X_TO_ST08X` Funktionsblock ist ein nützlicher, spezialisierter Baustein zur effizienten und klaren Umwandlung von 8-elementigen BOOL-Arrays in eine 8-elementige BOOL-Struktur. Er trägt zur Verbesserung der Code-Lesbarkeit und der Modularen Architektur in 4diac-Anwendungen bei, insbesondere in Szenarien, die eine Anpassung von Datenformaten erfordern.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

LogiBUS BOOL Conversion

🎧 Podcast

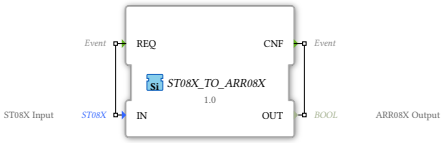
- George Boole: Wie ein Autodidakt mit 0 und 1 das Fundament unserer digitalen Welt legte

🛠️ Zugehörige Übungen

- Uebung_001_AX_b
- Uebung_002a2_AX
- Uebung_002a4
- Uebung_002a5b

- Uebung_002b3
- Uebung_004b_AX
- Uebung_006a3
- LinksRechts_T_FF_Event
- Uebung_035a2
- Uebung_035a3
- SoftkeyOverride_TO_QX_BG
- Uebung_055
- Uebung_056
- Bool8_TO_logiBUS_QX8
- Uebung_094a_AX
- Uebung_160

ST08X_TO_ARR08X



• – ○ • – ○ • – ○ •

Einleitung

Dieser Funktionsblock (FB) dient dazu, eine spezifische Struktur vom Typ `ST08X`, die acht einzelne BOOL-Werte enthält, in ein Array von ebenfalls acht BOOL-Werten umzuwandeln. Er ermöglicht eine nahtlose Anpassung der Datenrepräsentation in Automatisierungsanwendungen.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Dieses Ereignis wird verwendet, um die Konvertierungsoperation zu initiieren. Bei Empfang dieses Ereignisses werden die Daten vom Eingang `IN` verarbeitet und in den Ausgang `OUT` geschrieben.

Ereignis-Ausgänge

- **CNF**: Dieses Ereignis wird ausgelöst, nachdem die Daten erfolgreich vom `IN`-Eingang in das `OUT`-Array konvertiert und der Ausgang aktualisiert wurde.

Daten-Eingänge

- **IN** (`logiBUS::utils::conversion::types::ST08X`): Dies ist der Eingang für die Struktur vom Typ `ST08X`. Diese Struktur enthält acht benannte BOOL-Felder (X_00 bis X_07), die umgewandelt werden sollen.

Daten-Ausgänge

- **OUT** (`BOOL[0..7]`): Dies ist der Ausgang für das Array von BOOL-Werten. Das Array hat eine feste Größe von 8 Elementen, die die konvertierten Daten aus der Eingangsstruktur enthalten.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock `ST08X_TO_ARR08X` führt eine direkte und deterministische Umwandlung der Eingangsdaten durch. Wenn das `REQ`-Ereignis empfangen wird, werden die einzelnen BOOL-Felder der Eingangsstruktur `IN` dem entsprechenden Index des `OUT`-Arrays zugewiesen:

- `IN.X_00` wird `OUT[0]` zugewiesen.
- `IN.X_01` wird `OUT[1]` zugewiesen.
- ...
- `IN.X_07` wird `OUT[7]` zugewiesen.

Nach Abschluss dieser Zuweisungen wird das `CNF`-Ereignis ausgegeben, um anzuzeigen, dass der Ausgang `OUT` aktualisiert wurde und die konvertierten Daten zur Verfügung stehen.

Technische Besonderheiten

- **Spezifische Typen**: Der Funktionsblock ist speziell für die Konvertierung zwischen dem benutzerdefinierten Typ `logiBUS::utils::conversion::types::ST08X` und einem `BOOL[0..7]`-Array konzipiert.
- **Feste Zuordnung**: Die Zuordnung der Strukturfelder zu den Array-Indizes ist statisch und kann nicht zur Laufzeit konfiguriert werden.
- **Atomare Operation**: Die Konvertierung ist eine einzelne, abgeschlossene Operation, die durch ein Eingangserhöhungsereignis ausgelöst wird.

Zustandsübersicht

Der Funktionsblock `ST08X_TO_ARR08X` ist zustandslos (stateless). Er speichert keine internen Zustände oder Daten zwischen den Ausführungen. Jede Ausführung, die durch ein `REQ`-Ereignis ausgelöst wird, verarbeitet die aktuell an `IN` anliegenden Daten vollständig und unabhängig von früheren Operationen.

Anwendungsszenarien

- **Schnittstellenanpassung**: Ideal, um Daten von einer Komponente, die 8 BOOL-Werte als Struktur (`ST08X`) bereitstellt, an eine andere Komponente anzupassen, die ein `BOOL`-Array mit 8 Elementen erwartet (z.B. für Visualisierungen, externe Geräte oder generische Schleifenverarbeitung).
- **Vereinheitlichung der Datenformate**: Wenn in einem Projekt verschiedene Konventionen für die Gruppierung von 8 BOOL-Werten verwendet werden, kann dieser FB dazu beitragen, die Daten in ein konsistentes Format zu bringen.
- **Kompatibilität mit älteren Systemen**: Kann verwendet werden, um Daten aus moderneren Strukturen in array-basierte Formate zu überführen, die von älteren oder einfacheren Systemen besser verarbeitet werden können.

⚖️ Vergleich mit ähnlichen Bausteinen

Während 4diac-IDE und der IEC 61499-Standard generische Datenkonvertierungsmechanismen und Funktionen zur Manipulation von Arrays bereitstellen, ist der `ST08X_TO_ARR08X` ein hochspezialisierter Baustein. Allgemeine Konvertierungs-FBs würden möglicherweise eine flexiblere Handhabung verschiedener Datentypen oder Array-Größen bieten, müssten aber oft konfiguriert werden oder würden zusätzliche Logik erfordern. Dieser Baustein bietet eine dedizierte, einfache und effiziente Lösung für genau diesen einen spezifischen Konvertierungsfall ohne Konfigurationsaufwand.

Fazit

Der Funktionsblock `ST08X_TO_ARR08X` ist ein präzises Werkzeug zur Umwandlung einer 8-BOOL-Struktur in ein 8-BOOL-Array. Seine Einfachheit und Spezialisierung machen ihn zu einer effizienten Lösung für die Schnittstellenanpassung und Datenformatierung in 4diac-basierten Automatisierungsanwendungen, die eine solche Umwandlung erfordern.

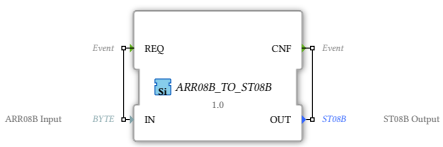
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Numerischer Index (converting)

Hier finden Sie alle Übungen des Arbeitsbereichs `converting` in aufsteigender Reihenfolge.

ARR08B_TO_ST08B



• – ○ • – ○ • – ○ •

Einleitung

Diese Funktion wandelt ein 8-Byte-Array in eine 8-Byte-Struktur um. Sie dient dazu, lineare Byte-Daten in ein strukturiertes Format zu überführen.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Ein steigendes Ereignis an diesem Eingang triggert die Konvertierung der Daten vom Eingang `IN` zum Ausgang `OUT`.
  - Verbunden mit: `IN`

Ereignis-Ausgänge

- **CNF**: Ein Ereignis an diesem Ausgang signalisiert den Abschluss der Konvertierung und dass die Ausgangsdaten an `OUT` aktualisiert wurden.
  - Verbunden mit: `OUT`

Daten-Eingänge

- **IN** (Typ: `BYTE`, Arraygröße: `[0..7]`): Der Eingangsdaten-Array, der die 8 Bytes enthält, die konvertiert werden sollen.
  - Kommentar: `ARR08B Input`

Daten-Ausgänge

- **OUT** (Typ: `logiBUS::utils::conversion::types::ST08B`): Die Ausgangsstruktur, die die konvertierten 8 Bytes enthält.
  - Kommentar: `ST08B Output`

Adapter

Es sind keine Adapter für diesen Funktionsblock definiert.

Funktionsweise

Der Funktionsblock `ARR08B_TO_ST08B` führt eine direkte Byte-für-Byte-Zuweisung durch. Wenn ein `REQ`-Ereignis empfangen wird, werden die einzelnen Elemente des Eingangs-Arrays `IN` den entsprechenden Komponenten der Ausgangsstruktur `OUT` zugewiesen:

- `OUT.B_00` wird `IN[0]` zugewiesen.
- `OUT.B_01` wird `IN[1]` zugewiesen.
- ...
- `OUT.B_07` wird `IN[7]` zugewiesen.

Nach Abschluss dieser Zuweisungen wird ein `CNF`-Ereignis ausgelöst, um anzuzeigen, dass die Konvertierung abgeschlossen ist und die neuen Daten am `OUT`-Ausgang verfügbar sind.

Technische Besonderheiten

- **Datentypen:** Der Eingangs-Array ist vom Typ `BYTE`, die Ausgangsstruktur `ST08B` ist ein benutzerdefinierter Typ, der aus 8 einzelnen `BYTE`-Komponenten besteht (z.B. `B_00` bis `B_07`).
- **Direkte Zuweisung:** Es findet keine Datenmanipulation oder Umrechnung statt; die Bytes werden direkt von der Array-Position zur Struktur-Komponente kopiert.
- **Stateless:** Der Funktionsblock hat keinen internen Zustand und produziert immer das gleiche Ergebnis für die gleiche Eingabe.

Zustandsübersicht

Der `ARR08B_TO_ST08B`-Funktionsblock ist **zustandslos**. Er führt eine rein funktionale Transformation durch: Bei jedem Trigger-Ereignis (`REQ`) wird die Umwandlung der aktuellen Eingabewerte (`IN`) vorgenommen, und das Ergebnis (`OUT`) wird ausgegeben. Es werden keine Werte zwischen den Aufrufen gespeichert.

Anwendungsszenarien

Dieser Funktionsblock ist nützlich in Szenarien, in denen:

- Rohdaten, die als Byte-Array empfangen werden (z.B. über Kommunikationsschnittstellen wie CAN, SPI, UART), in ein strukturiertes Format umgewandelt werden müssen, um die Lesbarkeit und weitere Verarbeitung zu erleichtern.
- Daten zwischen Systemen ausgetauscht werden, die unterschiedliche Repräsentationen für denselben 8-Byte-Datensatz verwenden (Array vs. Struktur).
- Eine klare und explizite Typumwandlung von einer linearen in eine benannte Struktur erforderlich ist, um die Programmlogik zu verbessern.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Generische Kopierfunktionen (z.B. `MEMCPY`):** Im Gegensatz zu einer generischen Speicher-Kopierfunktion, die Speicherbereiche byteweise kopiert, bietet `ARR08B_TO_ST08B` eine typensichere und explizite Zuweisung von Array-Elementen zu strukturierten Feldern. Dies kann helfen, Fehler bei der Größen- oder Typanpassung zu vermeiden und die Lesbarkeit des Codes zu verbessern.
- **Manuelle Zuweisungen:** Die Funktionalität könnte auch durch acht einzelne Zuweisungszeilen im ST-Code erreicht werden. Dieser Funktionsblock kapselt diese Logik jedoch in einer wiederverwendbaren und leicht integrierbaren Einheit, was die Modularität und Wartbarkeit fördert.
- **Type Casting:** Während in einigen Sprachen ein Type Cast eine ähnliche Umwandlung ermöglichen könnte, bietet dieser Funktionsblock eine explizite, ereignisgesteuerte Schnittstelle, die gut in das 4diac-Modell passt und eine klare Triggerung der Konvertierung ermöglicht.

Fazit

Der `ARR08B_TO_ST08B`-Funktionsblock ist ein spezialisierter und effizienter Baustein zur Umwandlung eines 8-Byte-Arrays in eine vordefinierte 8-Byte-Struktur. Er trägt zur Verbesserung der Code-Struktur, Lesbarkeit und Wartbarkeit bei, indem er eine häufig benötigte Konvertierungsaufgabe in einer klaren und wiederverwendbaren Einheit kapselt. Seine zustandslose Natur gewährleistet eine vorhersehbare und zuverlässige Funktion.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

LogiBUS BYTE Conversion

🛠️ Zugehörige Übungen

- SIN_TO_PCAN_Callback_Byte
- Uebung_128b
- Uebung_12x_sub

ST08B_TO_ARR08B

• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock `ST08B_TO_ARR08B` dient der Konvertierung eines 8-Byte-Strukturdatentyps (`ST08B`) in ein 8-Byte-Array. Diese Umwandlung ermöglicht die einfache Anpassung von strukturierten Daten an Schnittstellen oder Protokolle, die eine sequentielle Anordnung von Bytes erfordern.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Ein Ereignis-Eingang, der die Ausführung der Konvertierung auslöst. Dieses Ereignis ist mit dem Daten-Eingang `IN` verknüpft, um sicherzustellen, dass die Konvertierung mit den aktuellen Eingangsdaten erfolgt.

Ereignis-Ausgänge

- **CNF**: Ein Ereignis-Ausgang, der signalisiert, dass die Konvertierung erfolgreich abgeschlossen wurde und die Ausgabe `OUT` aktualisiert ist. Dieses Ereignis ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Ein Daten-Eingang vom Typ `logiBUS::utils::conversion::types::ST08B`. Dieser Eingang erwartet eine Struktur, die aus 8 Bytes besteht (z.B. `B_00` bis `B_07`).

Daten-Ausgänge

- **OUT**: Ein Daten-Ausgang vom Typ `BYTE` als Array mit der Größe `0..7`. Dieses Array enthält die 8 Bytes, die aus der Eingangsstruktur konvertiert wurden.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Die Funktionsweise des `ST08B_TO_ARR08B` ist direkt und unkompliziert. Beim Empfang eines `REQ`-Ereignisses werden die einzelnen Byte-Elemente der Eingangsstruktur `IN` direkt den entsprechenden Elementen des Ausgangsarrays `OUT` zugewiesen. Jedes Element `B_xx` der Struktur `IN` wird dem Element `OUT[xx]` des Arrays zugewiesen. Konkret erfolgt die Zuweisung wie folgt:

- `OUT[0]` wird `IN.B_00` zugewiesen.
- `OUT[1]` wird `IN.B_01` zugewiesen.
- ...
- `OUT[7]` wird `IN.B_07` zugewiesen. Nach Abschluss dieser Zuweisungen wird das `CNF`-Ereignis ausgelöst, um den Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- **Datentypen:** Der Funktionsblock arbeitet mit einem spezifischen benutzerdefinierten Strukturtyp `ST08B` und einem Standard `BYTE`-Array.
- **Direkte Zuordnung:** Es findet keine komplexe Logik, Transformation oder Berechnung statt, sondern eine direkte 1:1-Zuordnung der Bytes.
- **Transparenz:** Die Implementierung in ST (Strukturierter Text) ist klar und leicht nachvollziehbar, was eine hohe Transparenz der Funktion gewährleistet.

Zustandsübersicht

Der Funktionsblock `ST08B_TO_ARR08B` ist ein reiner Datenkonvertierungsbaustein ohne internen Zustand im Sinne einer Zustandsmaschine. Er führt seine Operation deterministisch bei jedem `REQ`-Ereignis aus und liefert ein Ergebnis basierend auf den aktuellen Eingabedaten.

Anwendungsszenarien

- **Datenvorbereitung für Kommunikation:** Um strukturierte Daten für serielle Kommunikationsprotokolle (z.B. Modbus RTU, SPI, I2C), Ethernet-Frames oder andere byteorientierte Schnittstellen vorzubereiten.
- **Anbindung externer Geräte:** Wenn externe Geräte oder Bibliotheken Daten als reines Byte-Array erwarten, kann dieser Baustein interne strukturierte Daten passend umwandeln.
- **Typumwandlung und Interoperabilität:** Verbesserung der Kompatibilität zwischen verschiedenen Teilen einer Anwendung oder zwischen verschiedenen Systemen, die unterschiedliche Datenrepräsentationen verwenden.
- **Archivierung und Logging:** Speicherung von strukturierten Daten in einem generischen Byte-Format für Archivierungs- oder Logging-Zwecke.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Konvertierungsbausteinen zeichnet sich `ST08B_TO_ARR08B` durch seine Spezialisierung auf die Umwandlung einer 8-Byte-Struktur in ein 8-Byte-Array aus. Allgemeine Typumwandlungsfunktionen könnten komplexer sein und mehr Konfigurationsmöglichkeiten bieten, während dieser Baustein eine hochoptimierte und spezifische Aufgabe erfüllt. Bausteine für die Array-zu-Struktur-Konvertierung (z.B. `ARR08B_TO_ST08B`) wären die Umkehrfunktion dieses Bausteins.

Fazit

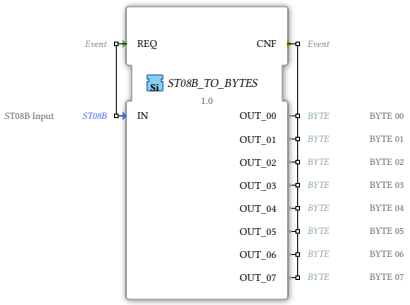
Der `ST08B_TO_ARR08B` ist ein einfacher, aber effektiver Funktionsblock zur Konvertierung einer 8-Byte-Struktur in ein 8-Byte-Array. Er bietet eine zuverlässige Lösung für Szenarien, in denen eine direkte und transparente Anpassung von strukturierten Daten an byteorientierte Schnittstellen oder Formate erforderlich ist. Seine klare Funktionalität und Implementierung machen ihn zu einem wertvollen Werkzeug in der 4diac-Entwicklungsumgebung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

forwarding

ST08B_TO_BYTES



• – ○ • – ○ • – ○ •

Einleitung

Dieser Funktionsblock dient dazu, die einzelnen 8 Bytes einer speziellen 8-Byte-Struktur (`ST08B`) zu extrahieren und als separate Byte-Ausgänge bereitzustellen. Er ermöglicht eine detaillierte Aufteilung und Weiterverarbeitung von strukturierten 8-Byte-Daten.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Funktion aus und initiiert die Extraktion der Bytes aus dem Daten-Eingang `IN`.

Ereignis-Ausgänge

- **CNF**: Wird nach erfolgreicher Ausführung und Aktualisierung aller Ausgangs-Bytes gesendet. Dieses Ereignis ist mit allen `OUT_xx` Daten-Ausgängen verknüpft.

Daten-Eingänge

- **IN** (`logiBUS::utils::conversion::types::ST08B`): Der Eingang, der die 8-Byte-Struktur enthält, deren Bytes extrahiert werden sollen.

Daten-Ausgänge

- **OUT_00** (`BYTE`): Das erste extrahierte Byte (Byte 00).
- **OUT_01** (`BYTE`): Das zweite extrahierte Byte (Byte 01).
- **OUT_02** (`BYTE`): Das dritte extrahierte Byte (Byte 02).
- **OUT_03** (`BYTE`): Das vierte extrahierte Byte (Byte 03).
- **OUT_04** (`BYTE`): Das fünfte extrahierte Byte (Byte 04).
- **OUT_05** (`BYTE`): Das sechste extrahierte Byte (Byte 05).
- **OUT_06** (`BYTE`): Das siebte extrahierte Byte (Byte 06).
- **OUT_07** (`BYTE`): Das achte extrahierte Byte (Byte 07).

Adapter

Dieser Funktionsblock verfügt über keine Adapter-Schnittstellen.

Funktionsweise

Sobald ein REQ-Ereignis empfangen wird, liest der Funktionsblock die Eingangsstruktur IN. Er extrahiert jedes einzelne Byte der Struktur (IN.B_00 bis IN.B_07) und weist es dem entsprechenden Ausgangsparameter (OUT_00 bis OUT_07) zu. Nach Abschluss dieser Zuweisungen wird das CNF-Ereignis ausgegeben, um die erfolgreiche Operation zu signalisieren. Die interne Logik ist direkt und transparent:

```
OUT_00 := IN.B_00;
OUT_01 := IN.B_01;
OUT_02 := IN.B_02;
OUT_03 := IN.B_03;
OUT_04 := IN.B_04;
OUT_05 := IN.B_05;
OUT_06 := IN.B_06;
OUT_07 := IN.B_07;
```

Technische Besonderheiten

Dieser Funktionsblock ist spezialisiert auf die Zerlegung der benutzerdefinierten Struktur ST08B. Die Typzuweisung erfolgt direkt und ohne zusätzliche Konvertierungslogik, da die Quell- und Zieltypen (Member der Struktur und Ausgänge) beide vom Typ BYTE sind. Dies gewährleistet eine effiziente und verlustfreie Aufteilung der Daten. Der Baustein ist deterministisch und stateless, da er keine internen Zustände speichert, die das Ergebnis zukünftiger Ausführungen beeinflussen könnten.

Zustandsübersicht

Der Funktionsblock ST08B_TO_BYTES ist ein reiner Konverter ohne internen Zustand. Jede Ausführung ist unabhängig von vorherigen Ausführungen und liefert stets das gleiche Ergebnis für identische Eingangsdaten.

Anwendungsszenarien

- **Datenprotokollierung und -analyse**: Wenn eine 8-Byte-Struktur empfangen wird, können die einzelnen Bytes extrahiert werden, um sie separat zu protokollieren, zu überwachen oder für spezifische Analysen zu verwenden.
- **Schnittstellenanpassung**: Anpassung von Datenformaten, wenn eine externe Schnittstelle eine 8-Byte-Struktur liefert, aber nachfolgende Bausteine einzelne Bytes erwarten.
- **Bit-Level-Manipulation**: Vorbereitung von Daten für Bit-Level-Operationen, bei denen jedes Byte einzeln adressiert werden muss.
- **Debugging**: Vereinfacht das Debugging von Daten in 8-Byte-Strukturen, indem die Inhalte der einzelnen Bytes sichtbar gemacht werden.

⚖️ Vergleich mit ähnlichen Bausteinen

Ähnliche Funktionalität könnte in anderen Systemen durch explizite Typumwandlungen oder Zeigeroperationen erreicht werden. Im 4diac-Kontext stellt dieser Baustein eine ereignisgesteuerte, klar definierte und wiederverwendbare Lösung für diese spezifische Struktur-zu-Byte-Aufteilung dar. Der Vorteil liegt in der direkten Anbindung an das 4diac-Ereignismodell und der expliziten Deklaration der Schnittstelle, was die Lesbarkeit und Wartbarkeit verbessert. Alternativ müssten die Zuweisungen direkt im ST-Code eines Composite Function Blocks erfolgen, was die Wiederverwendbarkeit einschränken würde.

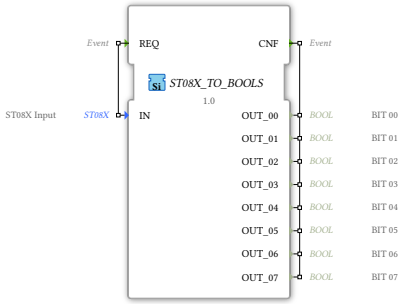
Fazit

Der ST08B_TO_BYTES-Funktionsblock ist ein nützliches Hilfsmittel zur Zerlegung einer spezifischen 8-Byte-Struktur in ihre einzelnen Byte-Bestandteile. Seine Einfachheit, klare Schnittstelle und deterministische Funktionsweise machen ihn zu einem zuverlässigen Baustein für Anwendungen, die eine präzise Kontrolle und Analyse von 8-Byte-Daten erfordern. Er fördert die Modularität und Wiederverwendbarkeit in 4diac-Anwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ST08X_TO_BOOLS



• — ○ • — ○ • — ○ •

Einleitung

Dieser Funktionsblock dient dazu, acht individuelle BOOL-Werte aus einer einzelnen Eingangsstruktur vom Typ `ST08X` zu extrahieren. Er ermöglicht die einfache Zerlegung einer gebündelten 8-BOOL-Struktur in ihre einzelnen Komponenten.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Ein Ereignis an diesem Eingang fordert die Konvertierung an. Es signalisiert dem Funktionsblock, die Daten vom Eingang `IN` zu verarbeiten.

Ereignis-Ausgänge

- **CNF**: Dieses Ereignis wird ausgegeben, sobald die Konvertierung abgeschlossen ist und alle Ausgangs-BOOL-Werte (`OUT_00` bis `OUT_07`) aktualisiert wurden.

Daten-Eingänge

- **IN** (Typ: `logiBUS::utils::conversion::types::ST08X`): Die Eingangsstruktur, die die acht zu extrahierenden BOOL-Werte enthält.

Daten-Ausgänge

- **OUT_00** (Typ: `BOOL`): Der erste extrahierte BOOL-Wert (Bit 00).
- **OUT_01** (Typ: `BOOL`): Der zweite extrahierte BOOL-Wert (Bit 01).
- **OUT_02** (Typ: `BOOL`): Der dritte extrahierte BOOL-Wert (Bit 02).
- **OUT_03** (Typ: `BOOL`): Der vierte extrahierte BOOL-Wert (Bit 03).
- **OUT_04** (Typ: `BOOL`): Der fünfte extrahierte BOOL-Wert (Bit 04).
- **OUT_05** (Typ: `BOOL`): Der sechste extrahierte BOOL-Wert (Bit 05).
- **OUT_06** (Typ: `BOOL`): Der siebte extrahierte BOOL-Wert (Bit 06).
- **OUT_07** (Typ: `BOOL`): Der achte extrahierte BOOL-Wert (Bit 07).

Adapter

Keine Adapter vorhanden.

Funktionsweise

Sobald ein Ereignis am Eingang `REQ` empfangen wird, führt der Funktionsblock eine direkte Zuweisung der einzelnen BOOL-Komponenten der Eingangsstruktur `IN` zu den entsprechenden Ausgangsvariablen durch. Konkret wird `IN.X_00` dem Ausgang `OUT_00` zugewiesen, `IN.X_01` dem Ausgang `OUT_01` und so weiter bis `IN.X_07` an `OUT_07`. Nach erfolgreicher Zuweisung wird ein Bestätigungsereignis `CNF` am Ausgang ausgegeben, um anzuzeigen, dass die Ergebnisse verfügbar sind.

Technische Besonderheiten

- **Direkte Zuordnung:** Der Funktionsblock implementiert eine einfache und direkte 1:1-Zuordnung von den Feldern der `ST08X`-Struktur zu den einzelnen BOOL-Ausgängen.
- **Typensicherheit:** Durch die Verwendung des spezifischen Typs `ST08X` wird eine klare und typensichere Schnittstelle gewährleistet.

- **Effizienz:** Die Konvertierung ist hoch effizient, da keine komplexe Logik oder Berechnungen erforderlich sind.

Zustandsübersicht

Dieser Funktionsblock ist zustandslos. Er speichert keine internen Daten zwischen Aufrufen und seine Ausgabe hängt ausschließlich von den aktuellen Eingangswerten ab. Jede Ausführung ist unabhängig von vorherigen Ausführungen.

Anwendungsszenarien

- **Zerlegung von kombinierten Signalen:** Ideal, um eine logisch gruppierte 8-Bit-Struktur in ihre einzelnen, unabhängigen BOOL-Signale aufzuteilen.
- **Anpassung der Schnittstelle:** Ermöglicht die Anbindung an andere Funktionsblöcke oder Systeme, die einzelne BOOL-Eingänge anstelle einer Struktur erwarten.
- **Visualisierung und Diagnose:** Vereinfacht die separate Anzeige oder Überwachung jedes einzelnen Bits einer `ST08X`-Struktur.

⚖️ Vergleich mit ähnlichen Bausteinen

Dieser Funktionsblock ist speziell für die `ST08X`-Struktur konzipiert. Während generische Bit-Extraktionsfunktionen (z.B. aus einem BYTE oder WORD) ähnliche Ergebnisse liefern könnten, bietet `ST08X_TO_BOOLS` eine klar definierte, typisierte Schnittstelle für die spezifische `ST08X`-Struktur. Dies erhöht die Lesbarkeit und vermeidet potenzielle Fehler bei der manuellen Bit-Indizierung. Er ist simpler und spezifischer als Bausteine, die komplexe Bitfeld-Operationen oder variable Bitanzahlen unterstützen.

Fazit

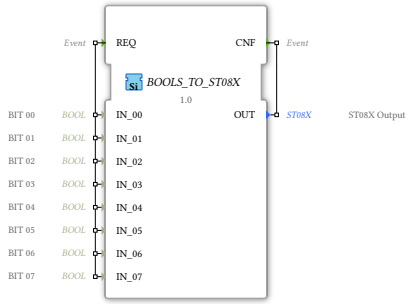
Der Funktionsblock `ST08X_TO_BOOLS` ist ein nützliches Werkzeug zur Zerlegung von `ST08X`-Strukturen in individuelle BOOL-Signale. Er besticht durch seine Einfachheit, Effizienz und die klare, typensichere Schnittstelle, was die Entwicklung und Wartung von 4diac-Anwendungen vereinfacht.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

reversing

BOOLS_TO_ST08X



• — ○ • — ○ • — ○ •

Einleitung

Dieser Funktionsblock (FB) dient dazu, acht einzelne BOOL-Werte zu einer Struktur vom Typ `ST08X` zusammenzufassen. Er ermöglicht eine übersichtliche Gruppierung und Weitergabe von acht binären Informationen als eine einzige strukturierte Variable.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Event): Dieses Ereignis löst die Konvertierung und Zuweisung der Eingangswerte an den Ausgang aus. Alle Daten-Eingänge sind mit diesem Ereignis verbunden und werden bei dessen Eintreffen gelesen.

Ereignis-Ausgänge

- **CNF** (Event): Dieses Bestätigungsereignis wird ausgelöst, nachdem die Konvertierung erfolgreich abgeschlossen und der Ausgang `OUT` aktualisiert wurde. Der Daten-Ausgang `OUT` ist mit diesem Ereignis verbunden.

Daten-Eingänge

- **IN_00** (BOOL): Der Wert für Bit 0 der Ausgangsstruktur.
- **IN_01** (BOOL): Der Wert für Bit 1 der Ausgangsstruktur.
- **IN_02** (BOOL): Der Wert für Bit 2 der Ausgangsstruktur.
- **IN_03** (BOOL): Der Wert für Bit 3 der Ausgangsstruktur.
- **IN_04** (BOOL): Der Wert für Bit 4 der Ausgangsstruktur.
- **IN_05** (BOOL): Der Wert für Bit 5 der Ausgangsstruktur.
- **IN_06** (BOOL): Der Wert für Bit 6 der Ausgangsstruktur.
- **IN_07** (BOOL): Der Wert für Bit 7 der Ausgangsstruktur.

Daten-Ausgänge

- **OUT** (logiBUS::utils::conversion::types::ST08X): Die resultierende Struktur, welche die Werte der acht binären Eingänge in ihren jeweiligen Feldern (`X_00` bis `X_07`) enthält.

Adapter

Dieser Funktionsblock besitzt keine Adapter.

Funktionsweise

Der Funktionsblock `BOOLS_TO_ST08X` wird durch ein eingehendes `REQ`-Ereignis aktiviert. Bei jeder Aktivierung werden die aktuellen Werte der acht Daten-Eingänge (`IN_00` bis `IN_07`) direkt den entsprechenden Feldern (`X_00` bis `X_07`) der Ausgangsstruktur `OUT` zugewiesen. Nach der erfolgreichen Zuweisung wird das `CNF`-Ereignis am Ausgang ausgelöst, um den Abschluss der Operation zu signalisieren und den aktualisierten `OUT`-Wert verfügbar zu machen. Die Zuweisung erfolgt 1:1, wobei `IN_00` dem Feld `X_00` entspricht, `IN_01` dem Feld `X_01` usw., bis `IN_07` dem Feld `X_07` zugewiesen wird.

Technische Besonderheiten

- **Direkte Zuweisung:** Die Logik besteht aus einer direkten 1:1-Zuweisung der Bool-Eingänge zu den Feldern der `ST08X`-Struktur.
- **Typisierung:** Der `ST08X`-Typ ist eine benutzerdefinierte Struktur, die speziell für die Aggregation von acht Bool-Werten konzipiert ist und typensicher verwendet werden kann.
- **Effizienz:** Der Baustein ist aufgrund seiner einfachen Implementierung sehr effizient und ressourcenschonend.
- **Zustandslos:** Der Funktionsblock speichert keine internen Zustände und verarbeitet jede Anforderung unabhängig von vorherigen Aufrufen.

Zustandsübersicht

Der Funktionsblock `BOOLS_TO_ST08X` ist ein zustandsloser Baustein. Dies bedeutet, dass er keine Informationen über vorherige Ausführungen speichert, die das aktuelle Verhalten beeinflussen könnten. Jede Aktivierung durch das `REQ`-Ereignis führt zu einer Neuberechnung des Ausgangs `OUT` basierend auf den zu diesem Zeitpunkt gültigen Werten der Daten-Eingänge.

Anwendungsszenarien

- **Sensordatenerfassung:** Zusammenfassen von acht einzelnen digitalen Sensorzuständen (z.B. Endschalter, Lichtschranken) in eine einzige Struktur für eine kompaktere Datenverarbeitung oder Visualisierung.
- **Statusregister:** Erstellen eines Statusbytes oder -registers aus individuellen Bool-Flags, die verschiedene Betriebszustände oder Fehler anzeigen.
- **Schnittstellenkommunikation:** Vorbereitung von binären Daten für die Übertragung über serielle Schnittstellen oder Kommunikationsprotokolle, die Daten in strukturierten Formaten erwarten.
- **Programmierlogik:** Vereinfachung der Logik in komplexeren Funktionsblöcken, indem acht separate Signale zu einer einzelnen, handlicheren Variablen gebündelt werden.

⚖️ Vergleich mit ähnlichen Bausteinen

Dieser Funktionsblock ist spezialisiert auf die Umwandlung von acht BOOL-Werten in den benutzerdefinierten `ST08X`-Typ. Im Vergleich zu:

- **`BOOLS_TO_BYTE` oder `BOOL_TO_WORD`:** Diese Bausteine wandeln einzelne BOOLs in einen primitiven Integer-Typ (Byte, Word) um, bei dem die einzelnen Bits über bitweise Operationen angesprochen werden müssen. `BOOLS_TO_ST08X` bietet hingegen eine strukturierte Variable, deren Felder direkt über Namen (`OUT.X_00`, `OUT.X_01`, etc.) zugänglich sind, was die Lesbarkeit des Codes verbessert.
- **Logische Operatoren (AND, OR, XOR):** Diese Operatoren kombinieren Bool-Werte, aber nicht zu einer Struktur, sondern zu einem einzelnen Bool-Ergebnis. `BOOLS_TO_ST08X` dient der Aggregation, nicht der logischen Verknüpfung.

Der Hauptvorteil von `BOOLS_TO_ST08X` liegt in der Verwendung eines spezifischen Strukturtyps, der eine höhere Typsicherheit und bessere Code-Organisation im Vergleich zu generischen Integer-Typen bietet.

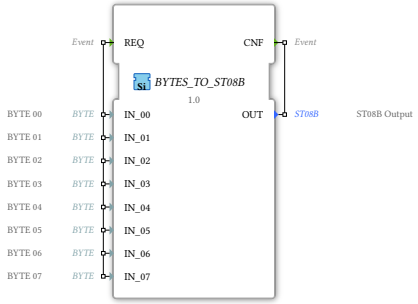
Fazit

Der `BOOLS_TO_ST08X` Funktionsblock ist ein einfacher, aber äußerst nützlicher Baustein, um acht einzelne BOOL-Signale in eine standardisierte `ST08X`-Struktur zu überführen. Er fördert die Modularität und Lesbarkeit von 4diac-Applikationen, indem er eine saubere und typensichere Methode zur Bündelung von Bit-Informationen bereitstellt. Dies vereinfacht die Datenhandhabung und -weitergabe in komplexen Steuerungssystemen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

BYTES_TO_ST08B



• – ○ • – ○ • – ○ •

Einleitung

Dieser Funktionsbaustein dient dazu, acht individuelle BYTE-Werte zu einer einzigen 8-BYTE-Struktur vom Typ `ST08B` zusammenzuführen. Er ermöglicht eine geordnete Überführung von diskreten Byte-Werten in ein strukturiertes Datenformat.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Typ: `Event`)
  - Verbindet die Aktivierung des Bausteins mit den Daten-Eingängen: `IN_00`, `IN_01`, `IN_02`, `IN_03`, `IN_04`, `IN_05`, `IN_06`, `IN_07`.
  - Löst die Konvertierung der Eingangs-BYTES in die `ST08B`-Struktur aus.

Ereignis-Ausgänge

- **CNF** (Typ: `Event`)
  - Wird nach erfolgreicher Konvertierung ausgelöst.
  - Signalisiert die Verfügbarkeit des konvertierten Werts am Daten-Ausgang `OUT`.

Daten-Eingänge

- **IN_00** (Typ: BYTE)
  - Kommentar: BYTE 00
- **IN_01** (Typ: BYTE)
  - Kommentar: BYTE 01
- **IN_02** (Typ: BYTE)
  - Kommentar: BYTE 02
- **IN_03** (Typ: BYTE)
  - Kommentar: BYTE 03
- **IN_04** (Typ: BYTE)
  - Kommentar: BYTE 04
- **IN_05** (Typ: BYTE)
  - Kommentar: BYTE 05
- **IN_06** (Typ: BYTE)
  - Kommentar: BYTE 06
- **IN_07** (Typ: BYTE)
  - Kommentar: BYTE 07

Daten-Ausgänge

- **OUT** (Typ: logiBUS::utils::conversion::types::ST08B)
  - Kommentar: ST08B Output
  - Gibt die aus den acht Eingangs-Bytes erstellte 8-Byte-Struktur aus.

Adapter

Dieser Funktionsbaustein verfügt über keine Adapter-Schnittstellen.

Funktionsweise

Bei Empfang eines REQ-Ereignisses weist der Baustein die Werte der acht Daten-Eingänge IN_00 bis IN_07 direkt den entsprechenden Feldern (B_00 bis B_07) der Ausgangsstruktur OUT zu. Nach Abschluss dieser Zuweisung wird ein CNF-Ereignis ausgelöst, welches signalisiert, dass die OUT-Struktur mit den neuen Werten aktualisiert wurde und zur Weiterverarbeitung bereitsteht.

Technische Besonderheiten

Der Funktionsbaustein verwendet den benutzerdefinierten Datentyp logiBUS::utils::conversion::types::ST08B als Ausgangsformat. Dieser Typ muss im System definiert sein, damit der Baustein korrekt funktioniert. Die Zuordnung der Bytes erfolgt positionsgenau: IN_00 wird zu OUT.B_00, IN_01 zu OUT.B_01 und so weiter.

Zustandsübersicht

Der BYTES_TO_ST08B-Baustein ist ein rein kombinatorischer Funktionsbaustein. Er besitzt keine internen Zustände, die über längere Zeit gespeichert werden. Jede Ausführung durch ein REQ-Ereignis verarbeitet die aktuellen Eingangswerte unabhängig von früheren Ausführungen und produziert ein entsprechendes Ausgangsergebnis.

Anwendungsszenarien

- **Datenpaket-Assemblierung:** Zusammenfügen von einzelnen Byte-Komponenten zu einem fest definierten Datenpaket, z.B. für die Kommunikation über serielle Schnittstellen oder Netzwerke.
- **Interfacing mit Protokollen:** Anpassung von Daten an Kommunikationsprotokolle, die spezifische 8-Byte-Strukturen erwarten.
- **Datentyp-Konvertierung:** Umwandlung von Raw-Byte-Daten in einen höherwertigen, strukturierten Datentyp zur besseren Handhabung und Lesbarkeit innerhalb der Applikation.
- **Hardware-Anbindung:** Wenn Daten von Sensoren oder Aktoren byteweise gelesen werden und zu einer festen Struktur aggregiert werden müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu allgemeinen Typumwandlungs- oder Array-Manipulationsbausteinen bietet BYTES_TO_ST08B eine spezialisierte Lösung für die Umwandlung von genau acht Bytes in eine vordefinierte ST08B-Struktur. Dies eliminiert die Notwendigkeit manueller Indizierung oder Schleifen und vereinfacht die Handhabung für diesen spezifischen Anwendungsfall. Andere Bausteine könnten flexibler

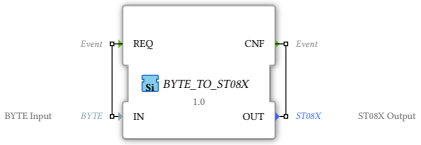
sein (z.B. eine variable Anzahl von Bytes verarbeiten), aber dieser Baustein ist für seine feste Größe optimiert.

Fazit

Der Funktionsbaustein `BYTES_TO_ST08B` ist ein nützliches Werkzeug, um acht einzelne Byte-Eingänge in eine definierte 8-Byte-Struktur `ST08B` zu konvertieren. Er trägt zur Modularisierung und Vereinfachung von Steuerungsprogrammen bei, insbesondere wenn es darum geht, diskrete Bytedaten für strukturierte Kommunikationsaufgaben oder interne Datenverarbeitung vorzubereiten. Seine direkte und ereignisgesteuerte Arbeitsweise macht ihn zuverlässig und einfach zu integrieren.

splitting

BYTE_TO_ST08X



• – ○ • – ○ • – ○ •

Einleitung

Dieser Funktionsblock (FB) dient dazu, einen einzelnen Byte-Wert in eine Struktur von acht einzelnen BOOL-Werten aufzuschlüsseln. Jedes Bit des Eingangsbytes wird dabei einem separaten Bool-Feld in der Ausgangsstruktur zugewiesen, was die Handhabung bitweiser Informationen vereinfacht.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Ein Ereigniseingang, der die Ausführung des Funktionsblocks triggert. Er ist mit dem Dateneingang `IN` assoziiert.

Ereignis-Ausgänge

- **CNF**: Ein Ereignisausgang, der signalisiert, dass die Konvertierung abgeschlossen ist. Er ist mit dem Datenausgang `OUT` assoziiert.

Daten-Eingänge

- **IN**: Typ `BYTE`. Dies ist der Byte-Wert, dessen acht Bits in einzelne Boolesche Werte aufgeteilt werden sollen.

Daten-Ausgänge

- **OUT**: Typ `logiBUS::utils::conversion::types::ST08X`. Dies ist eine Struktur, die acht BOOL-Werte (benannt X_00 bis X_07) enthält. Jedes dieser Felder repräsentiert ein Bit des Eingangsbytes.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der `BYTE_TO_ST08X` Funktionsblock konvertiert ein einzelnes 8-Bit-Byte in eine vordefinierte Struktur `ST08X`, die acht separate BOOL-Variablen enthält.

Bei einem `REQ`-Ereignis wird der Wert des `IN`-Bytes gelesen. Anschließend wird jedes Bit des `IN`-Bytes dem entsprechenden BOOL-Feld in der `OUT`-Struktur zugewiesen:

- `OUT.X_00` wird Bit 0 von `IN` zugewiesen.
- `OUT.X_01` wird Bit 1 von `IN` zugewiesen.
- ...
- `OUT.X_07` wird Bit 7 von `IN` zugewiesen.

Nach der Zuweisung wird ein `CNF`-Ereignis ausgelöst, um den Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- **Direkte Bit-Extraktion:** Die Konvertierung erfolgt durch eine direkte Zuweisung der einzelnen Bits des Bytes (z.B. `IN.0`) zu den BOOL-Feldern der Zielstruktur (z.B. `OUT.X_00`).
- **Verwendung von `ST08X`:** Der Funktionsblock nutzt die spezifische Datentyp-Struktur `logiBUS::utils::conversion::types::ST08X`, die explizit für die Aufnahme von acht Booleschen Werten vorgesehen ist. Diese Struktur muss im System bekannt sein.
- **Ereignisgesteuert:** Die Operation wird durch ein Eingangserreignis `REQ` ausgelöst und signalisiert ihren Abschluss durch ein Ausgangsereignis `CNF`. Dies gewährleistet eine kontrollierte Datenflusssteuerung im 4diac-System.

Zustandsübersicht

Dieser Funktionsblock ist ein reiner Datenkonverter ohne interne Zustände. Seine Funktion ist deterministisch und hängt ausschließlich von den Eingangswerten und dem auslösenden Ereignis ab.

Anwendungsszenarien

- **Hardware-Interfacing:** Wenn von externen Geräten (z.B. über Feldbusse wie Modbus, CANopen) ein Byte empfangen wird, das den Zustand von acht diskreten digitalen Eingängen oder Ausgängen darstellt, kann dieser FB verwendet werden, um die einzelnen Zustände für die Weiterverarbeitung im Steuerungsprogramm zu isolieren.
- **Protokoll-Parsing:** Zerlegung von Status-Bytes in Kommunikationsprotokollen, bei denen jedes Bit eine spezifische Flag oder einen Zustand anzeigt.
- **Logische Verknüpfungen:** Vorbereitung von Daten für logische Operationen, die einzelne BOOL-Werte erfordern.
- **Visualisierung:** Darstellung einzelner Bits eines Byte-Wertes auf einer Benutzeroberfläche.

⚖️ Vergleich mit ähnlichen Bausteinen

Standard-Funktionsblöcke in 4diac bieten oft allgemeine Typumwandlungen (z.B. `BYTE_TO_INT`). Dieser Baustein ist jedoch spezialisiert auf die Zerlegung eines Bytes in eine strukturierte Gruppe von Booleschen Werten (`ST08X`), was ihn von generischen Konvertern unterscheidet. Es gibt keine direkte Entsprechung in der Standardbibliothek für diese spezifische Strukturkonvertierung. Andere Ansätze könnten darin bestehen, Bitmasken und logische Operationen (AND, SHIFT) mit mehreren einzelnen Konvertierungsbausteinen zu verwenden, was jedoch zu einer komplexeren und weniger übersichtlichen Implementierung führen würde. Dieser Baustein bietet eine direkte und elegante Lösung für die genannte Aufgabe.

Fazit

Der `BYTE_TO_ST08X` Funktionsblock ist ein nützliches Werkzeug zur effizienten und übersichtlichen Zerlegung eines Byte-Wertes in acht individuelle Boolesche Zustände, die in einer `ST08X`-Struktur organisiert sind. Er vereinfacht die Handhabung bitweiser Informationen, insbesondere im Bereich der E/A-Verarbeitung und Protokollanalyse, indem er eine klare und ereignisgesteuerte Schnittstelle bietet.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

types

Numerischer Index (types)

Hier finden Sie alle Übungen des Arbeitsbereichs `types` in aufsteigender Reihenfolge.

LogiBUS BOOL Types

🎧 Podcast

- George Boole: Wie ein Autodidakt mit 0 und 1 das Fundament unserer digitalen Welt legte

🛠️ Zugehörige Übungen

- Uebung_001_AX_b
- Uebung_002a2_AX
- Uebung_002a4

- Uebung_002a5b
- Uebung_002b3
- Uebung_004b_AX
- Uebung_006a3
- LinksRechts_T_FF_Event
- Uebung_035a2
- Uebung_035a3
- SoftkeyOverride_TO_QX_BG
- Uebung_055
- Uebung_056
- Bool8_TO_logiBUS_QX8
- Uebung_094a_AX
- Uebung_160

ST01X

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Datentyp `ST01X` ist ein einfacher strukturierter Datentyp, der einen einzelnen Booleschen Wert (BOOL) kapselt. Er dient dazu, eine klar definierte Struktur für einen einzelnen Binärzustand bereitzustellen, was die Standardisierung und Lesbarkeit in komplexeren Anwendungen verbessern kann.

Schnittstellenstruktur

Die „Schnittstellenstruktur“ eines Datentyps bezieht sich auf seine internen Mitglieder, die die Definition des Datentyps bilden. Der Datentyp `ST01X` besitzt die folgende interne Struktur:

- **`X_00`**: Eine Variable vom Typ `BOOL`. Dieser Member dient als „BIT 00“ innerhalb der Struktur, wie im Kommentar angegeben.

Ereignis-Eingänge

Nicht zutreffend für einen Datentyp.

Ereignis-Ausgänge

Nicht zutreffend für einen Datentyp.

Daten-Eingänge

Nicht zutreffend für einen Datentyp im Sinne von Funktionblock-Eingängen. Der Datentyp definiert lediglich die Struktur für Daten.

Daten-Ausgänge

Nicht zutreffend für einen Datentyp im Sinne von Funktionblock-Ausgängen. Der Datentyp definiert lediglich die Struktur für Daten.

Adapter

Nicht zutreffend für einen Datentyp.

Funktionsweise

Der Datentyp `ST01X` hat keine aktive Funktionsweise im Sinne eines Algorithmus oder einer Ausführung. Seine Funktion besteht ausschließlich darin, als definierter Container für einen einzelnen Booleschen Wert zu dienen. Er ermöglicht es Entwicklern, einen spezifischen Booleschen Zustand innerhalb einer strukturierten Umgebung klar und typensicher zu referenzieren, wodurch die Modularität und die Verständlichkeit von Datenstrukturen erhöht werden können.

Technische Besonderheiten

- **Minimale Struktur:** `ST01X` ist einer der einfachsten strukturierten Datentypen, der nur ein einziges `BOOL`-Element enthält.
- **Kapselung:** Er kapselt einen einzelnen Booleschen Wert, der typischerweise einen Status, eine Flagge oder einen Zustand repräsentiert.

- **Standardisierung:** Dieser Datentyp kann zur Standardisierung der Verwendung von einzelnen Booleschen Werten in größeren Datenstrukturen oder bei der Definition von Kommunikationsprotokollen eingesetzt werden, um Konsistenz zu gewährleisten.

Zustandsübersicht

Ein Datentyp besitzt keine eigene „Zustandsübersicht“ im Sinne eines Zustandsautomaten eines Funktionsblocks. Der „Zustand“ eines Objekts vom Typ `ST01X` wird ausschließlich durch den Wert seines einzigen Members `X_00` bestimmt. Dieser kann entweder `TRUE` (wahr) oder `FALSE` (falsch) sein.

Anwendungsszenarien

- **Statusflaggen:** Repräsentation eines einfachen Ein/Aus-Status, z.B. zur Anzeige, ob ein Gerät aktiv ist, ein Prozess läuft oder eine Bedingung erfüllt ist.
- **Kommunikationsprotokolle:** Als standardisierter Block innerhalb von Datenpaketen oder Telegrammen, um einen einzelnen Bit-Zustand zwischen Systemen zu übermitteln.
- **Strukturierte Datenhaltung:** Wenn ein Boolescher Wert in einer übergeordneten Struktur mit einem spezifischen Namen und Kommentar benötigt wird, um die Lesbarkeit und Wartbarkeit zu verbessern, besonders wenn viele einzelne Booleans in einer logischen Gruppe zusammengefasst werden sollen.
- **Typensicherheit:** Erzwingt die Verwendung eines explizit definierten Typs für bestimmte Boolesche Informationen, um Fehler durch versehentliche Typinkonsistenzen zu vermeiden.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Vergleich mit dem primitiven Typ `BOOL`:** Im Gegensatz zum einfachen `BOOL`-Typ, der nur den reinen Wert darstellt, bietet `ST01X` eine benannte Kapselung (`X_00`). Dies verbessert die Selbstbeschreibung und die Semantik des Wertes innerhalb einer größeren Struktur. Während `BOOL` ein elementarer Datentyp ist, ist `ST01X` ein zusammengesetzter (strukturierter) Datentyp.
- **Vergleich mit größeren Strukturen (z.B. `ST08X`, `ST16X`):** `ST01X` ist die elementarste Form dieser Art von Struktur, spezialisiert auf nur ein Bit. Größere Strukturen wie `ST08X` oder `ST16X` würden entsprechend 8 oder 16 Boolesche Werte enthalten und dienen der Bündelung mehrerer Flags.

Fazit

Der Datentyp `ST01X` ist ein grundlegender, aber nützlicher Baustein in der 4diac-IDE für die Strukturierung von Daten. Er ermöglicht die explizite Definition und Kapselung eines einzelnen Booleschen Wertes, was zur Verbesserung der Code-Lesbarkeit, Wartbarkeit und Standardisierung in komplexen Automatisierungsprojekten beiträgt. Er dient als Basis für einheitliche Datenstrukturen, insbesondere wenn ein Bit-Flag eine spezifische Bedeutung innerhalb einer breiteren Datenlandschaft hat.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ST02X

(Kein Bild verfügbar für diesen Datentyp.)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Datentyp **ST02X** ist ein strukturierter Datentyp, der darauf ausgelegt ist, zwei boolesche Werte zu kapseln. Er dient als einfache Datenstruktur, um zwei individuelle Bits oder boolesche Zustände in einer einzigen Einheit zu organisieren und zu transportieren. Der Name „ST02X“ deutet auf eine „Struktur mit 2 [Booleschen Werten]“ hin.

Schnittstellenstruktur

Da es sich bei ST02X um einen Datentyp und nicht um einen Funktionsblock handelt, besitzt er keine externen Ereignis- oder Dateneingänge/-ausgänge im üblichen Sinne eines Funktionsblocks. Seine „Struktur“ bezieht sich auf die Definition seiner internen Komponenten, die die eigentlichen Daten darstellen.

Ereignis-Eingänge

Nicht zutreffend, da ST02X ein Datentyp und kein Funktionsblock ist und somit keine Ereignisse verarbeitet.

Ereignis-Ausgänge

Nicht zutreffend, da ST02X ein Datentyp und kein Funktionsblock ist und somit keine Ereignisse erzeugt.

Daten-Eingänge

Die internen Komponenten dieses Datentyps können als seine „Daten“ betrachtet werden, die gesetzt oder gelesen werden können, wenn eine Variable dieses Typs verwendet wird.

- **X_00** (Typ: BOOL, Kommentar: BIT 00)
  - Der erste boolesche Wert innerhalb der Struktur.
- **X_01** (Typ: BOOL, Kommentar: BIT 01)
  - Der zweite boolesche Wert innerhalb der Struktur.

Daten-Ausgänge

Die internen Komponenten können auch als „Daten-Ausgänge“ angesehen werden, da ihre Werte von externen Bausteinen oder Programmteilen ausgelesen werden können, wenn eine Variable vom Typ ST02X verwendet wird.

Adapter

Nicht zutreffend, da ST02X ein Datentyp und kein Funktionsblock ist und somit keine Adapter-Schnittstellen besitzt.

Funktionsweise

Der Datentyp ST02X ist ein passiver Container für Daten. Er selbst führt keine Operationen oder Logik aus. Seine Funktionsweise beschränkt sich darauf, die beiden booleschen Werte `X_00` und `X_01` logisch zusammenzufassen. Wenn eine Variable des Typs ST02X deklariert wird, kann auf ihre Komponenten über den Punkt-Operator (z.B. `meineVariable.X_00`) zugegriffen werden, um deren Werte zu lesen oder zu schreiben.

Technische Besonderheiten

- **Kompakte Struktur:** Der Datentyp ist sehr einfach gehalten und besteht nur aus zwei booleschen Variablen.
- **Typische Anwendung:** Er eignet sich besonders zur Gruppierung von zwei verwandten Status-Bits, Flags oder Schalterzuständen, die gemeinsam behandelt werden sollen.
- **Paketierung:** Die interne Struktur kann dazu beitragen, Daten kompakt zu halten und die Lesbarkeit des Codes zu verbessern, indem logisch zusammengehörende Informationen gebündelt werden.

Zustandsübersicht

Da ST02X selbst keine aktive Logik besitzt, hat es keinen „Zustand“ im Sinne eines Zustandsautomaten. Der „Zustand“ einer Variablen vom Typ ST02X wird durch die Kombination der Werte ihrer beiden booleschen Komponenten (`X_00` und `X_01`) bestimmt. Es gibt vier mögliche Zustände für eine Instanz von ST02X:

- `X_00 = FALSE, X_01 = FALSE`
- `X_00 = TRUE, X_01 = FALSE`
- `X_00 = FALSE, X_01 = TRUE`
- `X_00 = TRUE, X_01 = TRUE`

Anwendungsszenarien

- **Statusanzeige:** Gruppierung von zwei Status-Flags (z.B. „Alarm Aktiv“ und „Warnung Anstehend“).
- **Konfigurationsparameter:** Bündelung von zwei binären Konfigurationseinstellungen.
- **Sensordaten:** Zusammenfassung der Zustände zweier einfacher Sensoren (z.B. zwei Endlagenschalter).
- **Schnittstellen:** Als Parameter für Funktionsblöcke, die zwei boolesche Informationen auf einmal verarbeiten oder zurückgeben müssen, was die Schnittstellendefinition vereinfacht.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Einzelne BOOL-Variablen:** Anstatt ST02X zu verwenden, könnten zwei separate BOOL-Variablen deklariert werden. Der Vorteil von ST02X liegt in der logischen Bündelung und der potenziell übersichtlicheren Übergabe als einzelner Parameter an Funktionsblöcke.
- **ARRAY OF BOOL:** Ein Array wie `ARRAY[0..1] OF BOOL` könnte ebenfalls zwei boolesche Werte halten. ST02X bietet jedoch den Vorteil der benannten Zugriffe (`.X_00`, `.X_01`), was die Lesbarkeit gegenüber Indexzugriffen (`[0]`, `[1]`) verbessern kann.
- **Andere strukturierte Typen:** Abhängig von der Anwendung könnten komplexere strukturierte Datentypen mit mehr booleschen Werten (z.B. ST04X, ST08X) oder gemischten Datentypen besser geeignet sein, wenn mehr oder unterschiedliche Informationen gebündelt werden müssen. ST02X ist speziell für den Fall von genau zwei BOOLS optimiert.

Fazit

ST02X ist ein einfacher, aber nützlicher strukturierter Datentyp, der die logische Gruppierung von zwei booleschen Werten ermöglicht. Er verbessert die Code-Organisation und Lesbarkeit, insbesondere beim Umgang mit zwei verwandten binären Informationen. Seine Stärke liegt in seiner Einfachheit und klaren Definition, was ihn zu einer guten Wahl für spezifische Anwendungsfälle macht, bei denen genau zwei boolesche Zustände gebündelt werden müssen.

ST04X

• – ◦ • – ◦ • – ◦ •

Einleitung

ST04X ist ein strukturierter Datentyp, der vier boolesche Werte gruppiert. Er dient dazu, eine Sammlung von vier einzelnen Bits in einer einzigen Struktur zu organisieren und zu handhaben.

Schnittstellenstruktur

Da ST04X ein Datentyp und kein Funktionsblock ist, besitzt er keine Ereignis- oder Dateneingänge/-ausgänge im Sinne eines Funktionsblocks oder Adapter. Stattdessen definiert er eine interne Struktur von Datenfeldern.

Ereignis-Eingänge

Nicht zutreffend für einen Datentyp.

Ereignis-Ausgänge

Nicht zutreffend für einen Datentyp.

Daten-Eingänge

Nicht zutreffend für einen Datentyp.

Daten-Ausgänge

Nicht zutreffend für einen Datentyp.

Adapter

Nicht zutreffend für einen Datentyp.

Struktur-Elemente: Die Struktur ST04X besteht aus den folgenden vier booleschen Feldern:

- **X_00** (Typ: BOOL, Kommentar: BIT 00)
- **X_01** (Typ: BOOL, Kommentar: BIT 01)
- **X_02** (Typ: BOOL, Kommentar: BIT 02)
- **X_03** (Typ: BOOL, Kommentar: BIT 03)

Funktionsweise

Als Datentyp hat ST04X keine aktive „Funktionsweise“ im Sinne eines Funktionsbausteins. Seine Funktion ist passiv: Er definiert lediglich, wie vier boolesche Variablen logisch zusammengefasst und als eine Einheit referenziert werden können. Dies ermöglicht eine bessere Organisation und Lesbarkeit des Codes, indem zusammengehörige Bits in einer Struktur gebündelt werden.

Technische Besonderheiten

- **Kompaktheit**: Bündelt vier individuelle BOOL-Werte, was in manchen Systemen eine effizientere Datenhaltung oder -übertragung ermöglichen kann.
- **Strukturierung**: Verbessert die Code-Struktur und -Lesbarkeit, indem verwandte Bits unter einem gemeinsamen Namen zusammengefasst werden.
- **Verwendung in FBs**: Dieser Datentyp kann als Ein- oder Ausgangsparameter für Funktionsblöcke oder als interne Variable innerhalb eines Funktionsblocks verwendet werden, um Gruppierungen von vier Bits zu verwalten.

Zustandsübersicht

Nicht zutreffend für einen Datentyp. Ein Datentyp repräsentiert eine Datenstruktur, nicht einen Zustandsautomaten.

Anwendungsszenarien

- **Statusregister**: Abbildung eines kleinen Statusregisters oder einer Gruppe von Flags, bei denen jedes Bit einen bestimmten Zustand oder eine Eigenschaft anzeigt.
- **Binäreingänge/-ausgänge**: Gruppierung von vier digitalen Ein- oder Ausgängen einer Hardware-Schnittstelle.
- **Bitfelder**: Wenn eine kompakte Darstellung von vier unabhängigen binären Informationen erforderlich ist, z.B. bei der Verarbeitung von seriellen Datenprotokollen oder Gerätekonfigurationen.

⚖️ Vergleich mit ähnlichen Bausteinen

ST04X ist ein Datentyp und kann daher nicht direkt mit Funktionsblöcken (Bausteinen) verglichen werden. Ein Vergleich wäre eher mit anderen Datentypen möglich, z.B.:

- **BOOL**: Ein einzelner Boolescher Wert. ST04X bündelt vier davon.
- **BYTE**: Ein 8-Bit-Integer. ST04X bietet eine strukturierte Sicht auf 4 Bits, während ein BYTE in der Regel als eine unstrukturierte 8-Bit-Zahl behandelt wird. Die einzelnen Bits eines BYTE müssten über Bitmasken oder Bit-Operationen angesprochen werden, während bei ST04X jedes Bit direkt über seinen Namen zugänglich ist (z.B. `myST04X.X_00`).
- **Array von BOOL**: Ein Array von BOOLs (`ARRAY[0..3] OF BOOL`) bietet ebenfalls die Gruppierung, der Zugriff erfolgt jedoch über Indizes (`myArray[0]`), wohingegen ST04X den Zugriff über benannte Felder (`myST04X.X_00`) ermöglicht, was die Lesbarkeit verbessern kann.

Fazit

ST04X ist ein nützlicher strukturierter Datentyp in 4diac, der die Verwaltung und Organisation von vier booleschen Werten vereinfacht. Obwohl er selbst keine aktive Funktionalität bietet, ist er ein wichtiges Werkzeug zur Verbesserung der Code-Strukturierung und -Lesbarkeit in komplexeren Steuerungsprogrammen, indem er zusammengehörige Bits logisch bündelt und einen direkten, benannten Zugriff auf jedes Bit ermöglicht.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ST08X

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Datentyp **ST08X** ist eine strukturierte Datenkomponente, die für die Verwaltung und Gruppierung von acht individuellen booleschen Werten konzipiert wurde. Sein Hauptzweck ist es, eine kompakte Darstellung für eine Sammlung von acht unabhängigen Bits bereitzustellen, die oft als Status-Flags, Steuersignale oder ähnliche binäre Zustände verwendet werden.

Schnittstellenstruktur

Da **ST08X** ein Datentyp und kein Funktionsblock ist, besitzt er keine Ereignis-, Daten-Ein- oder Ausgänge im Sinne eines 4diac-Funktionsblocks. Stattdessen definiert er eine interne Struktur von Mitgliedsvariablen.

Ereignis-Eingänge

Nicht zutreffend für einen Datentyp.

Ereignis-Ausgänge

Nicht zutreffend für einen Datentyp.

Daten-Eingänge

Nicht zutreffend für einen Datentyp. Der Datentyp besteht aus folgenden internen Variablen:

- `X_00`: Typ `BOOL`, Kommentar: „BIT 00“
- `X_01`: Typ `BOOL`, Kommentar: „BIT 01“
- `X_02`: Typ `BOOL`, Kommentar: „BIT 02“
- `X_03`: Typ `BOOL`, Kommentar: „BIT 03“
- `X_04`: Typ `BOOL`, Kommentar: „BIT 04“
- `X_05`: Typ `BOOL`, Kommentar: „BIT 05“
- `X_06`: Typ `BOOL`, Kommentar: „BIT 06“
- `X_07`: Typ `BOOL`, Kommentar: „BIT 07“

Daten-Ausgänge

Nicht zutreffend für einen Datentyp.

Adapter

Nicht zutreffend für einen Datentyp.

Funktionsweise

Der Datentyp **ST08X** hat keine eigenständige „Funktionsweise“ im Sinne eines Algorithmus oder einer Verarbeitung. Er dient lediglich als Container, um acht boolesche Werte unter einem gemeinsamen Namen zu organisieren. Die einzelnen Bits können über ihre Namen (z.B. `ST08X_Variable.X_00`) in anderen Funktionsblöcken oder Programmlogiken gelesen und geschrieben werden.

Technische Besonderheiten

- **Kompakte Struktur**: Fasst acht `BOOL`-Variablen zu einer einzigen logischen Einheit zusammen.
- **Explizite Benennung**: Jedes Bit ist explizit von `X_00` bis `X_07` benannt, was die Lesbarkeit und Adressierung verbessert.
- **Keine implizite Konvertierung**: Als strukturierter Datentyp bietet er eine typsichere Kapselung der Bits.

Zustandsübersicht

Als reiner Datentyp besitzt **ST08X** keine Zustände im Sinne einer Zustandsmaschine oder eines Lebenszyklus. Sein „Zustand“ wird durch die aktuellen Werte seiner acht booleschen Mitglieder bestimmt.

Anwendungsszenarien

- **Statusregister**: Abbildung von Statusbits eines externen Geräts oder einer internen Komponente.
- **Konfigurationsparameter**: Gruppierung von 8 binären Konfigurationsoptionen.
- **Fehlermeldungen**: Sammlung von bis zu 8 verschiedenen Fehlermeldungs-Flags.
- **Binäre Ein-/Ausgänge**: Repräsentation von 8 digitalen Ein- oder Ausgängen.
- **Datenbus-Repräsentation**: In Fällen, wo eine Gruppe von 8 Bits als Einheit behandelt werden soll, ohne sie als kompletten Byte-Wert zu interpretieren, ermöglicht ST08X den direkten Zugriff auf einzelne Bits.

⚖️ Vergleich mit ähnlichen Bausteinen

Der Datentyp **ST08X** bietet eine Alternative zu:

- **Einzelne BOOL-Variablen**: Reduziert die Anzahl der Deklarationen und vereinfacht die Parameterübergabe, wenn viele boolesche Werte zusammengehören.
- **BYTE-Datentyp**: Während ein `BYTE` ebenfalls 8 Bits speichert, werden diese als numerischer Wert behandelt. **ST08X** behält die separate, namentliche Adressierbarkeit jedes einzelnen Bits, was die Semantik und den direkten Zugriff auf logische Flags verbessert, ohne Bitmaskierungen oder Bitverschiebeoperationen verwenden zu müssen. Die explizite Benennung `X_00` bis `X_07` macht den Code selbsterklärender.

Fazit

Der Datentyp **ST08X** ist ein nützliches Konstrukt, um acht unabhängige boolesche Werte in einer strukturierten und gut benannten Weise zu gruppieren. Er fördert die Code-Lesbarkeit und -Wartbarkeit, indem er eine klare semantische Kapselung für Sammlungen von binären Flags bietet, die über einfache Bit-Operationen auf einem Byte hinausgehen soll.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ST16X

• – ◦ • – ◦ • – ◦ •

Einleitung

ST16X ist ein benutzerdefinierter Datentyp, der eine Struktur von 16 einzelnen BOOL-Werten definiert. Er dient dazu, eine Sammlung von binären Zuständen oder Flags in einer einzigen, organisierten Einheit zu kapseln. Obwohl es sich um einen Datentyp und nicht um einen Funktionsblock handelt, wird seine Struktur und Verwendungsweise hier beschrieben. Der Datentyp wurde erstellt, um eine übersichtliche und typsichere Möglichkeit zur Verwaltung von 16 logischen Zuständen zu bieten.

Schnittstellenstruktur

Da ST16X ein Datentyp und kein Funktionsblock ist, besitzt er keine klassischen Schnittstellen im Sinne von Ereignis- oder Daten-Ein-/Ausgängen. Stattdessen definiert seine ‚Struktur' die internen Mitglieder, die im Folgenden detailliert werden.

Ereignis-Eingänge

Nicht zutreffend für einen Datentyp.

Ereignis-Ausgänge

Nicht zutreffend für einen Datentyp.

Daten-Eingänge

Nicht zutreffend für einen Datentyp.

Daten-Ausgänge

Nicht zutreffend für einen Datentyp.

Adapter

Nicht zutreffend für einen Datentyp.

Die Struktur des Datentyps ST16X ist wie folgt definiert:

Name	Typ	Kommentar
X_00	BOOL	BIT 00
X_01	BOOL	BIT 01
X_02	BOOL	BIT 02
X_03	BOOL	BIT 03
X_04	BOOL	BIT 04
X_05	BOOL	BIT 05
X_06	BOOL	BIT 06
X_07	BOOL	BIT 07
X_08	BOOL	BIT 08
X_09	BOOL	BIT 09

B_10	BOOL	BIT 10
B_11	BOOL	BIT 11
B_12	BOOL	BIT 12
B_13	BOOL	BIT 13
B_14	BOOL	BIT 14
B_15	BOOL	BIT 15

Funktionsweise

Ein Datentyp wie ST16X hat keine aktive ‚Funktionsweise' im Sinne eines Algorithmus, der ausgeführt wird. Er definiert lediglich eine Vorlage für Daten. Die ‚Funktionsweise' ergibt sich aus der Art und Weise, wie dieser Datentyp in Funktionsblöcken oder anderen Programmlogiken verwendet wird, um 16 individuelle boolesche Zustände zu speichern und zu verwalten. Eine Variable vom Typ ST16X kann erstellt werden, und ihre einzelnen BOOL-Elemente können gelesen und geschrieben werden (z.B. `meineST16XVariable.X_00 := TRUE;`).

Technische Besonderheiten

- **Strukturierte Anordnung:** Der Datentyp fasst 16 unabhängige BOOL-Werte zu einer logischen Einheit zusammen.
- **Eindeutige Benennung:** Jedes Bit ist über einen individuellen Namen (`X_00` bis `B_15`) anstatt über einen Index ansprechbar, was die Lesbarkeit des Codes verbessert.
- **Kommentar:** Jeder Variable ist ein Kommentar zugeordnet („BIT xx"), der die Bitposition verdeutlicht.
- **Modularität:** Fördert die Modularität und Wiederverwendbarkeit von Code, indem komplexe Statusinformationen gebündelt werden.

Zustandsübersicht

Als reiner Datentyp hat ST16X selbst keine ‚Zustandsübersicht' im Sinne eines endlichen Automaten oder einer internen Logik. Der ‚Zustand' wird durch die aktuellen Werte der 16 enthaltenen BOOL-Variablen repräsentiert, die jeweils `TRUE` oder `FALSE` sein können. Die Kombination dieser 16 Booleschen Werte ergibt 2^16 (65.536) mögliche Zustände für eine Instanz des Datentyps.

Anwendungsszenarien

- **Statusregister:** Darstellung des Zustands von bis zu 16 verschiedenen Komponenten, Sensoren oder Funktionen (z.B. „Motor EIN", „Fehler aktiv", „Grenzwert erreicht").
- **Bit-Flags:** Speicherung von Konfigurations-Flags oder Fehlerindikatoren in Steuerungsanwendungen.
- **Digitale Ein-/Ausgangssignale:** Bündelung von digitalen Ein- oder Ausgängen eines Moduls, um sie als eine einzige Datenstruktur zu übergeben.
- **Datenpaketierung:** Vereinfachte Übergabe von mehreren binären Informationen zwischen Funktionsblöcken oder über Kommunikationsschnittstellen.

⚖️ Vergleich mit ähnlichen Bausteinen

Da ST16X ein Datentyp ist, kann er nicht direkt mit einem ‚Baustein' im Sinne eines Funktionsblocks verglichen werden. Er kann jedoch mit anderen Möglichkeiten zur Speicherung von Binärdaten verglichen werden:

- **Einzelne BOOL-Variablen:** ST16X bietet den Vorteil, 16 BOOLs als eine Einheit zu behandeln, was die Datenstrukturierung und Übergabe vereinfacht und die Anzahl der deklarierten Variablen reduziert.
- **ARRAY OF BOOL:** Ein `ARRAY [0..15] OF BOOL` bietet eine ähnliche Funktionalität. ST16X hat den Vorteil, dass die einzelnen Bits über sprechende Namen (z.B. `meineST16XVariable.X_00`) adressiert werden können, was die Lesbarkeit verbessert, anstatt über Indizes (`meinARRAY[0]`).
- **WORD/INT-Typen:** Diese können ebenfalls zur Speicherung von Bits verwendet werden. ST16X bietet eine typsichere und selbstdokumentierende Möglichkeit, Bits zu verwalten, ohne manuelle Bit-Maskierungsoperationen oder typische Umwandlungsfehler.

Fazit

Der Datentyp ST16X stellt eine effiziente und übersichtliche Methode dar, um eine Gruppe von 16 booleschen Werten zu strukturieren. Er verbessert die Lesbarkeit des Codes, vereinfacht die Handhabung komplexer binärer Zustände und fördert die Typsicherheit in 4diac-Anwendungen, insbesondere in Szenarien, wo viele individuelle binäre Informationen als eine zusammengehörige Einheit verwaltet werden müssen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ST32X

(Kein Bild vorhanden, da dies ein Datentyp und kein Funktionsblock ist.)

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Datentyp `ST32X` ist ein benutzerdefinierter Strukturdatentyp, der aus 32 einzelnen BOOL-Variablen besteht. Er dient dazu, eine Sammlung von booleschen Zuständen oder Flags in einer einzigen, zusammenhängenden Datenstruktur zu bündeln, was die Handhabung und Übergabe von Statusinformationen vereinfachen kann. Der Name `ST32X` leitet sich von „Structure (ST) with 32 X/B (Boolean) bits“ ab.

Schnittstellenstruktur

Da `ST32X` ein Datentyp und kein Funktionsblock ist, besitzt er keine ereignis- oder datenbezogenen Ein- und Ausgänge im Sinne eines FB. Seine „Struktur“ definiert lediglich seine internen Elemente.

Ereignis-Eingänge

Nicht zutreffend, da `ST32X` ein benutzerdefinierter Datentyp und kein Funktionsblock ist.

Ereignis-Ausgänge

Nicht zutreffend, da `ST32X` ein benutzerdefinierter Datentyp und kein Funktionsblock ist.

Daten-Eingänge

Die Struktur besteht aus den folgenden 32 BOOL-Elementen, die intern in der Struktur als „Eingänge“ betrachtet werden können, da sie die Werte aufnehmen, aus denen die Struktur gebildet wird:

- `X_00` (Type: BOOL, Comment: BIT 00)
- `X_01` (Type: BOOL, Comment: BIT 01)
- `X_02` (Type: BOOL, Comment: BIT 02)
- `X_03` (Type: BOOL, Comment: BIT 03)
- `X_04` (Type: BOOL, Comment: BIT 04)
- `X_05` (Type: BOOL, Comment: BIT 05)
- `X_06` (Type: BOOL, Comment: BIT 06)
- `X_07` (Type: BOOL, Comment: BIT 07)
- `X_08` (Type: BOOL, Comment: BIT 08)
- `X_09` (Type: BOOL, Comment: BIT 09)
- `B_10` (Type: BOOL, Comment: BIT 10)
- `B_11` (Type: BOOL, Comment: BIT 11)
- `B_12` (Type: BOOL, Comment: BIT 12)
- `B_13` (Type: BOOL, Comment: BIT 13)
- `B_14` (Type: BOOL, Comment: BIT 14)
- `B_15` (Type: BOOL, Comment: BIT 15)
- `B_16` (Type: BOOL, Comment: BIT 16)
- `B_17` (Type: BOOL, Comment: BIT 17)
- `B_18` (Type: BOOL, Comment: BIT 18)
- `B_19` (Type: BOOL, Comment: BIT 19)
- `B_20` (Type: BOOL, Comment: BIT 20)
- `B_21` (Type: BOOL, Comment: BIT 21)
- `B_22` (Type: BOOL, Comment: BIT 22)
- `B_23` (Type: BOOL, Comment: BIT 23)

- B_24 (Type: BOOL, Comment: BIT 24)
- B_25 (Type: BOOL, Comment: BIT 25)
- B_26 (Type: BOOL, Comment: BIT 26)
- B_27 (Type: BOOL, Comment: BIT 27)
- B_28 (Type: BOOL, Comment: BIT 28)
- B_29 (Type: BOOL, Comment: BIT 29)
- B_30 (Type: BOOL, Comment: BIT 30)
- B_31 (Type: BOOL, Comment: BIT 31)

Daten-Ausgänge

Nicht zutreffend. Die Struktur ST32X selbst ist der „Ausgang“, da sie die gebündelten booleschen Werte repräsentiert. Einzelne Elemente können über den Punktoperator (.) angesprochen werden (z.B. myST32X.X_00).

Adapter

Nicht zutreffend, da ST32X ein benutzerdefinierter Datentyp und kein Funktionsblock ist.

Funktionsweise

ST32X hat keine aktive „Funktionsweise“ im Sinne von Logik oder Algorithmen. Es dient lediglich als passiver Datencontainer. Seine Funktion besteht darin, eine Gruppe von 32 einzelnen BOOL-Werten in einer einzigen Variable zu organisieren, was die Strukturierung von Programmen und die Übergabe von Daten zwischen Funktionsblöcken oder Programmteilen vereinfacht. Man kann die einzelnen Bits der Struktur ansprechen und ihren Wert setzen oder lesen.

Technische Besonderheiten

- **Anzahl der Bits:** Die Struktur fasst 32 unabhängige BOOL-Variablen zusammen.
- **Benennung:** Die Elemente sind von X_00 bis X_09 und dann von B_10 bis B_31 benannt. Die Kommentare geben jeweils die Bit-Nummer an. Dies könnte auf eine ursprüngliche Absicht hindeuten, die Bits in einem Wort (z.B. DINT) zu repräsentieren, obwohl sie hier als separate BOOLs deklariert sind.
- **Speichereffizienz:** Die genaue Speicherbelegung hängt von der Compiler-Implementierung ab. Typischerweise würde eine solche Struktur mindestens 32 Bytes (ein Byte pro BOOL) oder möglicherweise mehr belegen, je nach Padding und Ausrichtung.

Zustandsübersicht

Als passiver Datentyp hat ST32X keine eigene Zustandsmaschine. Sein „Zustand“ wird ausschließlich durch die Werte der 32 enthaltenen BOOL-Variablen definiert. Jedes Bit kann entweder TRUE oder FALSE sein.

Anwendungsszenarien

- **Bündelung von Statusflags:** Wenn ein System viele binäre Statusinformationen (z.B. Fehlercodes, Betriebszustände von Komponenten, Sensorflags) verwalten muss, kann ST32X verwendet werden, um diese in einer einzigen Variable zu übergeben oder zu speichern.
- **Schnittstellenoptimierung:** Anstatt 32 separate BOOL-Signale an einen Funktionsblock zu übergeben, kann eine Instanz von ST32X über eine einzige Datenverbindung übermittelt werden.
- **Bit-Mapping:** Obwohl die Elemente als BOOL definiert sind, könnte diese Struktur in Verbindung mit Konvertierungs-FBs (z.B. zur Umwandlung in einen DINT) verwendet werden, um auf die Bits eines 32-Bit-Wortes zuzugreifen oder diese zu manipulieren.
- **SPS-ähnliche E/A-Abbildungen:** In Umgebungen, die an SPS-Programmierungen angelehnt sind, kann eine solche Struktur zur Abbildung von digitalen Ein- oder Ausgängen dienen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Standard-Datentypen (z.B. DINT):** Ein DINT (Double Integer) ist ebenfalls 32 Bit breit, speichert jedoch die Bits als einen einzigen numerischen Wert. ST32X hingegen behandelt jedes Bit als eine separate boolesche Variable, was den direkten Zugriff per Namen ermöglicht, ohne Bitmaskierung oder Bit-Operationen.
- **Andere strukturierte Datentypen:** Es könnten andere benutzerdefinierte Strukturen existieren, die eine kleinere oder größere Anzahl von BOOLs bündeln oder unterschiedliche Datentypen mischen. ST32X ist spezifisch auf 32 BOOLs ausgelegt.

- **Array von BOOLs:** Ein `ARRAY[0..31] OF BOOL` würde ebenfalls 32 boolesche Werte speichern. Der Unterschied zu `ST32X` liegt im Zugriff: Bei `ST32X` erfolgt der Zugriff über benannte Elemente (z.B. `myST32X.X_00`), bei einem Array über den Index (z.B. `myArray[0]`). Benannte Elemente sind oft lesbarer und robuster gegenüber Änderungen in der Reihenfolge.

Fazit

Der Datentyp `ST32X` bietet eine praktische Methode, um 32 einzelne boolesche Variablen zu einem kohärenten Ganzen zu verbinden. Er fördert die Code-Organisation und vereinfacht die Datenübergabe von umfangreichen Statusinformationen oder Flags. Obwohl er selbst keine aktive Logik implementiert, ist er ein nützliches Werkzeug in der Architektur komplexer Steuerungsprogramme, die auf der 4diac-IDE basieren.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ST64X

(Kein Bild vorhanden, da dies ein Datentyp und kein Funktionsblock ist.)

• – ◦ • – ◦ • – ◦ •

Einleitung

Die bereitgestellte XML-Definition beschreibt einen **Datentyp (DataType)** namens `ST64X`, nicht einen Funktionsblock (FB). Dieser strukturierte Datentyp dient als Container für eine Sammlung von 64 einzelnen binären Werten (BOOL-Variablen). Er ermöglicht es, 64 unabhängige Schalter, Flags oder Statusbits unter einem einzigen Namen zu gruppieren und einzeln über benannte Felder anzusprechen. Obwohl der interne XML-Kommentar „A Structure with 16 BOOLS“ angibt, enthält die tatsächliche Definition 64 BOOL-Variablen.

Schnittstellenstruktur

Da `ST64X` ein Datentyp und kein Funktionsblock ist, besitzt er keine direkten Ereignis- oder Dateneinund -ausgänge im Sinne eines FBs. Stattdessen definiert er eine interne Struktur von Komponenten, die als einheitliche Variable in einem Funktionsblock verwendet werden kann (z.B. als Ein- oder Ausgangsparameter).

Ereignis-Eingänge

Nicht zutreffend für einen Datentyp.

Ereignis-Ausgänge

Nicht zutreffend für einen Datentyp.

Daten-Eingänge

Nicht zutreffend für einen Datentyp. Die hier aufgeführten Elemente sind die **internen Komponenten** der `ST64X`-Struktur, die Daten halten. Wenn `ST64X` als Dateneingang eines FBs verwendet wird, sind dies die verfügbaren Felder:

Name	Typ	Kommentar
`X_00`	`BOOL`	`BIT 00`
`X_01`	`BOOL`	`BIT 01`
`X_02`	`BOOL`	`BIT 02`
`X_03`	`BOOL`	`BIT 03`
`X_04`	`BOOL`	`BIT 04`
`X_05`	`BOOL`	`BIT 05`
`X_06`	`BOOL`	`BIT 06`
`X_07`	`BOOL`	`BIT 07`

X_08	BOOL	BIT 08
X_09	BOOL	BIT 09
B_10	BOOL	BIT 10
B_11	BOOL	BIT 11
B_12	BOOL	BIT 12
B_13	BOOL	BIT 13
B_14	BOOL	BIT 14
B_15	BOOL	BIT 15
B_16	BOOL	BIT 16
B_17	BOOL	BIT 17
B_18	BOOL	BIT 18
B_19	BOOL	BIT 19
B_20	BOOL	BIT 20
B_21	BOOL	BIT 21
B_22	BOOL	BIT 22
B_23	BOOL	BIT 23
B_24	BOOL	BIT 24
B_25	BOOL	BIT 25
B_26	BOOL	BIT 26
B_27	BOOL	BIT 27
B_28	BOOL	BIT 28
B_29	BOOL	BIT 29
B_30	BOOL	BIT 30
B_31	BOOL	BIT 31
B_32	BOOL	BIT 32
B_33	BOOL	BIT 33
B_34	BOOL	BIT 34
B_35	BOOL	BIT 35
B_36	BOOL	BIT 36
B_37	BOOL	BIT 37
B_38	BOOL	BIT 38
B_39	BOOL	BIT 39
B_40	BOOL	BIT 40
B_41	BOOL	BIT 41
B_42	BOOL	BIT 42
B_43	BOOL	BIT 43
B_44	BOOL	BIT 44

B_45	BOOL	BIT 45
B_46	BOOL	BIT 46
B_47	BOOL	BIT 47
B_48	BOOL	BIT 48
B_49	BOOL	BIT 49
B_50	BOOL	BIT 50
B_51	BOOL	BIT 51
B_52	BOOL	BIT 52
B_53	BOOL	BIT 53
B_54	BOOL	BIT 54
B_55	BOOL	BIT 55
B_56	BOOL	BIT 56
B_57	BOOL	BIT 57
B_58	BOOL	BIT 58
B_59	BOOL	BIT 59
B_60	BOOL	BIT 60
B_61	BOOL	BIT 61
B_62	BOOL	BIT 62
B_63	BOOL	BIT 63

Daten-Ausgänge

Nicht zutreffend für einen Datentyp. Die oben gelisteten Elemente sind die **internen Komponenten** der ST64X-Struktur, die Daten halten. Wenn ST64X als Datenausgang eines FBs verwendet wird, sind dies die verfügbaren Felder.

Adapter

Nicht zutreffend für einen Datentyp.

Funktionsweise

ST64X hat selbst keine „Funktionsweise“ im Sinne eines aktiven Bausteins, der Berechnungen durchführt oder Zustände ändert. Seine „Funktion“ besteht darin, eine definierte Struktur bereitzustellen, die als Vorlage für Datenobjekte dient. Einmal instanziiert, hält eine Variable vom Typ ST64X 64 individuelle BOOL-Werte, die über ihre Namen direkt gelesen oder geschrieben werden können (z.B. MyVariable.X_00, MyVariable.B_63). Es ist ein passiver Datencontainer.

Technische Besonderheiten

- **Strukturierte Daten:** Definiert eine feste Anordnung von 64 BOOL-Variablen.
- **Benannte Felder:** Jedes Bit hat einen eindeutigen Namen (X_00 bis X_09 und B_10 bis B_63), was die Lesbarkeit des Codes im Vergleich zu rein numerischen Indizes verbessert.
- **Größe:** Repräsentiert 64 Bits, was in der Regel 8 Bytes Speicherplatz belegt (abhängig von der Compiler-Implementierung und Padding).
- **Paketinformation:** Der Datentyp ist im Paket logiBUS::utils::conversion::types angesiedelt, was auf seine Rolle in Hilfsprogrammen oder Konvertierungen hindeutet.

Zustandsübersicht

Nicht zutreffend für einen Datentyp, da er keine interne Logik oder Zustandsmaschine besitzt. Der „Zustand“ einer ST64X-Variable ist einfach der aktuelle Wert der 64 enthaltenen BOOL-Variablen.

Anwendungsszenarien

- **Bit-Felder und Flags:** Ideal zur Darstellung einer großen Anzahl von binären Statusflags, Schalterstellungen oder Konfigurationsoptionen, die in einem zusammenhängenden Block organisiert werden sollen.
- **Hardware-Schnittstellen:** Kann zur Abbildung von Registern oder E/A-Ports verwendet werden, bei denen einzelne Bits spezifische Funktionen oder Zustände repräsentieren.
- **Kompakte Datenübertragung:** Wenn eine große Menge von BOOL-Werten effizient übermittelt werden muss, kann dieser Typ eine kompakte Darstellung bieten.
- **Abstrakte Zustände:** Darstellung komplexer Zustände, die sich aus der Kombination von 64 binären Merkmalen ergeben.

⚖️ Vergleich mit ähnlichen Bausteinen

Da `ST64X` ein Datentyp ist, vergleicht man ihn nicht direkt mit Funktionsblöcken, sondern mit anderen Möglichkeiten zur Strukturierung von BOOL-Werten:

- **`ARRAY OF BOOL`:** Ein `ARRAY[0..63] OF BOOL` würde ebenfalls 64 BOOL-Werte speichern. Der Hauptunterschied besteht im Zugriff: `ST64X` verwendet benannte Felder (z.B. `MyVar.X_00`), was oft lesbarer ist, wenn jedes Bit eine spezifische Bedeutung hat. Arrays verwenden Indizes (z.B. `MyArray[0]`), was flexibler für Schleifen und dynamischen Zugriff ist, aber weniger aussagekräftig, wenn die Bedeutung der einzelnen Indizes nicht offensichtlich ist.
- **Einzelne `BOOL`-Variablen:** Man könnte auch 64 separate `BOOL`-Variablen deklarieren. Dies würde jedoch zu einer unübersichtlichen Liste von Variablen führen und die Zusammengehörigkeit der Bits verdecken. `ST64X` gruppiert sie logisch.
- **Integer-Datentypen (z.B. `LWORD`):** Ein `LWORD` (64-Bit-Ganzzahl) könnte ebenfalls 64 Bits speichern. Der Zugriff auf einzelne Bits würde jedoch Bit-Operationen (AND, OR, SHIFT) erfordern, was komplexer und fehleranfälliger sein kann als der direkte Feldzugriff einer Struktur. `ST64X` bietet direkten, symbolischen Bit-Zugriff.

Fazit

Der Datentyp `ST64X` ist ein nützliches Konstrukt in 4diac-ide zur Definition einer strukturierten Variable, die 64 individuelle BOOL-Werte enthält. Er bietet eine klare, benannte Schnittstelle zu jedem einzelnen Bit, was die Lesbarkeit und Wartbarkeit des Codes verbessert, insbesondere in Anwendungen, die eine feingranulare Steuerung oder Statusanzeige mit vielen binären Flags erfordern. Er ist eine gute Alternative zu `ARRAY OF BOOL` oder Bit-Manipulationen auf Integer-Typen, wenn die individuelle Bedeutung jedes Bits im Vordergrund steht.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Numerischer Index (types)

Hier finden Sie alle Übungen des Arbeitsbereichs `types` in aufsteigender Reihenfolge.

LogiBUS BYTE Types

🛠️ Zugehörige Übungen

- SIN_TO_PCAN_Callback_Byte
- Uebung_128b
- Uebung_12x_sub

ST01B

• – ◦ • – ◦ • – ◦ •

Einleitung

Die bereitgestellte XML-Datei beschreibt den 4diac **Datentyp** `ST01B` (Structure with 1 BYTE), nicht einen Funktionsblock (FB). Dieser Datentyp definiert eine einfache Datenstruktur, die ein einzelnes Byte-Feld enthält. Die folgende Dokumentation versucht, die angeforderte Struktur für einen Funktionsblock so weit wie möglich anzuwenden, obwohl viele Abschnitte für einen reinen Datentyp nicht zutreffend sind.

Schnittstellenstruktur

Da `ST01B` ein Datentyp und kein Funktionsblock ist, besitzt er keine Ereignis- oder Daten-Schnittstellen im Sinne eines Funktionsblocks. Stattdessen definiert er eine interne Struktur für Daten.

Ereignis-Eingänge

Nicht zutreffend, da `ST01B` ein Datentyp und kein Funktionsblock ist.

Ereignis-Ausgänge

Nicht zutreffend, da `ST01B` ein Datentyp und kein Funktionsblock ist.

Daten-Eingänge

Nicht zutreffend, da `ST01B` ein Datentyp und kein Funktionsblock ist. Er definiert eine Struktur, die als Typ für Datenvariablen in Funktionsblöcken verwendet werden kann.

Daten-Ausgänge

Nicht zutreffend, da `ST01B` ein Datentyp und kein Funktionsblock ist. Er definiert eine Struktur, die als Typ für Datenvariablen in Funktionsblöcken verwendet werden kann.

Adapter

Nicht zutreffend, da `ST01B` ein Datentyp und kein Funktionsblock ist.

Funktionsweise

`ST01B` selbst hat keine Funktionsweise im Sinne eines ausführbaren Bausteins. Es handelt sich um eine statische Definition einer Datenstruktur, die ein einzelnes Byte umfasst. Diese Struktur kann von Funktionsblöcken instanziiert und zur Speicherung oder zum Austausch von Byte-Werten verwendet werden.

Die Struktur enthält die folgende Variable:

- **B_00**: Ein `BYTE`-Feld, das als „BYTE 00“ kommentiert ist.

Technische Besonderheiten

- **Einfache Struktur**: Definiert eine minimale Struktur zur Kapselung eines einzelnen Bytes.
- **Verwendung in 4diac**: Dieser Datentyp kann in 4diac-Applikationen verwendet werden, um typisierte Datenvariablen zu deklarieren oder als Teil komplexerer Datentypen.
- **Paket**: Der Datentyp ist im Paket `logiBUS::utils::conversion::types` untergebracht.

Zustandsübersicht

Nicht zutreffend, da `ST01B` ein Datentyp und kein ausführbarer Funktionsblock mit Zuständen ist.

Anwendungsszenarien

Dieser Datentyp kann in verschiedenen Szenarien nützlich sein, in denen ein einzelnes Byte mit einem spezifischen semantischen Namen in einer strukturierten Weise gehandhabt werden muss, zum Beispiel:

- Als einheitlicher Typ für bestimmte Registerwerte.
- Zur Vereinheitlichung von Schnittstellen, die einzelne Byte-Werte übergeben.
- Als Baustein für komplexere Strukturen, die aus einer Reihe von Bytes bestehen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Direkte Verwendung von `BYTE`**: `ST01B` unterscheidet sich von der direkten Verwendung des primitiven `BYTE`-Typs dadurch, dass es einen spezifischen Namen und eine Kapselung bietet. Dies kann die Lesbarkeit und Wartbarkeit des Codes verbessern, insbesondere wenn das Byte eine spezielle Bedeutung hat, die über den generischen `BYTE`-Typ hinausgeht.
- **Andere strukturierte Datentypen**: Im Vergleich zu komplexeren strukturierten Datentypen ist `ST01B` sehr einfach gehalten und dient als grundlegender Baustein.

Fazit

`ST01B` ist ein einfacher, aber nützlicher strukturierter Datentyp in 4diac, der die Kapselung eines einzelnen Bytes ermöglicht. Obwohl es sich nicht um einen Funktionsblock handelt, kann er als grundlegender Baustein die Klarheit und Struktur von 4diac-Anwendungen verbessern, indem er spezifisch benannte Byte-Variablen definiert. Die bereitgestellte XML-Definition ermöglicht eine einfache Integration in 4diac-Projekte.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ST02B

• – ◦ • – ◦ • – ◦ •

Einleitung

`ST02B` ist ein benutzerdefinierter, strukturierter Datentyp innerhalb der 4diac-IDE. Er definiert eine Datenstruktur, die zwei einzelne Bytes zu einer logischen Einheit zusammenfasst. Dies vereinfacht die Handhabung von Paaren von Byte-Werten.

Strukturdefinition

Da `ST02B` ein Datentyp und kein Funktionsblock ist, besitzt er keine Schnittstellen im Sinne von Ereignis-Ein-/Ausgängen, Daten-Ein-/Ausgängen oder Adaptern. Stattdessen werden hier die internen Komponenten des Datentyps beschrieben.

Komponenten der Struktur

Die Struktur `ST02B` besteht aus den folgenden zwei Byte-Variablen:

- **B_00** (Typ: `BYTE`): Das erste Byte der Struktur (BYTE 00).
- **B_01** (Typ: `BYTE`): Das zweite Byte der Struktur (BYTE 01).

Funktionsweise

Als passiver Datentyp hat `ST02B` keine eigene „Funktionsweise“. Seine Funktion besteht darin, eine definierte Vorlage für Daten bereitzustellen. Instanzen dieses Datentyps können verwendet werden, um Daten zu speichern, zu organisieren und als Einheit zwischen verschiedenen Funktionsblöcken auszutauschen.

Technische Besonderheiten

`ST02B` ist als `StructuredType` definiert und besteht aus zwei aufeinanderfolgenden `BYTE`-Variablen. Die Gesamtgröße einer Instanz dieses Datentyps beträgt daher 2 Bytes.

Anwendungsszenarien

- **Datenübertragung**: Bündelung von 2 Bytes, die als Einheit behandelt werden sollen.
- **Registerabbildung**: Repräsentation eines 2-Byte-Registers (ähnlich einem WORD, aber mit explizitem Byte-Zugriff).
- **Kleine Datenpakete**: Vereinfachte Handhabung von kleinen, fest definierten Datenpaketen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Array von Bytes (ARRAY OF BYTE [2])**: Ein Array von 2 Bytes erfüllt den gleichen Zweck. `ST02B` bietet jedoch den Vorteil benannter Felder (B_00, B_01).
- **`WORD`**: Repräsentiert ebenfalls 2 Bytes. `ST02B` ist flexibler, wenn die einzelnen Bytes unabhängig voneinander manipuliert werden müssen.

Fazit

`ST02B` ist ein einfacher, nützlicher strukturierter Datentyp zur Verwaltung von zwei Bytes als kohärente Einheit.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ST04B

• – ◦ • – ◦ • – ◦ •

Einleitung

`ST04B` ist ein benutzerdefinierter, strukturierter Datentyp innerhalb der 4diac-IDE. Im Gegensatz zu einem Funktionsblock (FB), der eine spezifische Funktionalität implementiert, definiert ein Datentyp eine Daten-

struktur, die von Funktionsblöcken oder anderen Datentypen verwendet werden kann. Dieser Datentyp fasst vier einzelne Bytes zu einer logischen Einheit zusammen, um deren gemeinsame Handhabung zu vereinfachen.

Strukturdefinition

Da `ST04B` ein Datentyp und kein Funktionsblock ist, besitzt er keine Schnittstellen im Sinne von Ereignis-Ein-/Ausgängen, Daten-Ein-/Ausgängen oder Adaptern. Die nachfolgenden Abschnitte adaptieren die ursprüngliche Anforderung einer Schnittstellenstruktur, um die internen Komponenten des Datentyps darzustellen.

Komponenten der Struktur

Die Struktur `ST04B` besteht aus den folgenden vier Byte-Variablen:

- **B_00** (Typ: `BYTE`): Das erste Byte der Struktur (BYTE 00).
- **B_01** (Typ: `BYTE`): Das zweite Byte der Struktur (BYTE 01).
- **B_02** (Typ: `BYTE`): Das dritte Byte der Struktur (BYTE 02).
- **B_03** (Typ: `BYTE`): Das vierte Byte der Struktur (BYTE 03).

Funktionsweise

Als passiver Datentyp hat `ST04B` keine eigene „Funktionsweise“ im Sinne einer Logik oder eines Algorithmus. Seine Funktion besteht darin, eine definierte Vorlage für Daten bereitzustellen. Instanzen dieses Datentyps können in Funktionsblöcken deklariert und verwendet werden, um Daten zu speichern, zu organisieren und als kohärente Einheit zwischen verschiedenen Funktionsblöcken auszutauschen. Die internen Komponenten (B_00 bis B_03) sind direkt adressierbar.

Technische Besonderheiten

`ST04B` ist als `StructuredType` definiert und besteht aus vier aufeinanderfolgenden `BYTE`-Variablen. Die Gesamtgröße einer Instanz dieses Datentyps beträgt daher 4 Bytes. Im Kontext der 4diac-Laufzeitumgebung wird dieser Datentyp entsprechend seiner Definition im Speicher abgelegt. Die interne Struktur ermöglicht einen direkten Zugriff auf jedes einzelne Byte über seinen Namen (z.B. `myST04B.B_00`).

Der 4diac-Compiler ordnet diesen Datentyp dem Paket `logiBUS::utils::conversion::types` zu.

Zustandsübersicht

Als passiver Datentyp, der lediglich eine Datenstruktur definiert, besitzt `ST04B` keine eigene interne Logik oder Zustandsmaschine im Sinne eines Funktionsblocks. Sein „Zustand“ ist ausschließlich durch die Werte seiner vier Komponenten-Bytes bestimmt. Eine Zustandsübersicht im Sinne von Verhaltenszuständen ist daher nicht anwendbar.

Anwendungsszenarien

- **Datenübertragung**: Bündelung von 4 Bytes, die über eine Kommunikationsschnittstelle (z.B. seriell, Modbus, CAN) empfangen oder gesendet werden, und die als Einheit behandelt werden sollen.
- **Registerabbildung**: Repräsentation eines 4-Byte-Registers oder einer Gruppe von Registern in einer speicherprogrammierbaren Steuerung (SPS) oder einem anderen Steuerungssystem.
- **Kleine Datenpakete**: Vereinfachte Handhabung von kleinen, fest definierten Datenpaketen in der industriellen Automatisierung.
- **Typkonvertierungshilfsmittel**: Interne Verwendung in Konvertierungsfunktionsblöcken, um spezifische Byte-Sequenzen zu verarbeiten und in andere Datentypen umzuwandeln.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Array von Bytes (ARRAY OF BYTE [4])**: Ein Array von 4 Bytes erfüllt den gleichen Zweck bezüglich der Speicherung. `ST04B` bietet jedoch den Vorteil, dass die einzelnen Bytes über benannte Felder (B_00, B_01, etc.) anstatt über Indizes (z.B. `myArray[0]`, `myArray[1]`) angesprochen werden können, was die Lesbarkeit und Selbstbeschreibung des Codes verbessern kann.
- **Andere strukturierte Typen**: Wenn die Bytes eine spezifische Bedeutung (z.B. Statusflag, Zählerwert) hätten, könnten komplexere strukturierte Typen mit aussagekräftigeren Variablennamen oder sogar Bitfeldern sinnvoller sein. `ST04B` ist bewusst generisch gehalten, um eine flexible Gruppierung von Bytes zu ermöglichen.
- **`WORD`, `DWORD`**: Diese Standarddatentypen repräsentieren ebenfalls eine bestimmte Anzahl von Bytes (2 bzw. 4). `ST04B` ist jedoch flexibler, wenn die einzelnen Bytes unabhängig voneinander manipuliert

werden müssen oder wenn die Byte-Reihenfolge (Endianness) eine Rolle spielt und explizit gesteuert werden soll.

Fazit

`ST04B` ist ein grundlegender, aber nützlicher strukturierter Datentyp in 4diac, der die Verwaltung von vier Bytes als kohärente Einheit vereinfacht. Er verbessert die Code-Lesbarkeit und Modularität, insbesondere in Anwendungen, die den Austausch oder die Verarbeitung von Byte-sequenzierten Daten erfordern. Seine einfache Definition macht ihn zu einem flexiblen Baustein für vielfältige Datenmanagementaufgaben in der Automatisierungstechnik.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ST08B

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Datentyp `ST08B` ist eine strukturierte Datenstruktur, die genau acht Byte (BYTE) sequenziell zusammenfasst. Er dient primär als Container für die effiziente Gruppierung und den Transport von acht einzelnen Byte-Werten innerhalb eines IEC 61499-basierten Steuerungssystems. Dieser Datentyp ist besonders nützlich für die Handhabung von Rohdaten, Protokollrahmen oder für die strukturierte Übergabe von parametrisierten Datenblöcken zwischen Funktionsblöcken.

Schnittstellenstruktur

`ST08B` ist ein Datentyp und besitzt als solcher keine ereignisgesteuerten Ein- oder Ausgänge im Sinne eines Funktionsblocks. Seine Struktur wird ausschließlich durch seine Datenmitglieder definiert.

Daten-Eingänge

Nicht zutreffend. Dieser Abschnitt ist für Funktionsblöcke vorgesehen.

Daten-Ausgänge

Nicht zutreffend. Dieser Abschnitt ist für Funktionsblöcke vorgesehen.

Ereignis-Eingänge

Nicht zutreffend. Dieser Abschnitt ist für Funktionsblöcke vorgesehen.

Ereignis-Ausgänge

Nicht zutreffend. Dieser Abschnitt ist für Funktionsblöcke vorgesehen.

Adapter

Nicht zutreffend. Dieser Abschnitt ist für Funktionsblöcke vorgesehen.

Funktionsweise

Als strukturierter Datentyp (STRUCT) fungiert `ST08B` als zusammengesetzter Behälter. Er bündelt acht unabhängige Variablen des Typs `BYTE` unter einem gemeinsamen Namen. Jedes Byte innerhalb der Struktur ist über seinen eindeutigen Member-Namen (`B_00` bis `B_07`) direkt adressierbar. Funktionsblöcke können Variablen vom Typ `ST08B` deklarieren, um alle acht Bytes als eine Einheit zu übergeben, zu speichern oder zu verarbeiten. Der Zugriff auf einzelne Bytes erfolgt durch die Punktnotation (z.B. `MyST08BInstance.B_04`).

Technische Besonderheiten

- **Feste Größe:** Die Struktur hat eine fest definierte und vorhersehbare Größe von genau 8 Bytes, was für Speicherallokation und Kommunikationsprotokolle vorteilhaft ist.
- **Sequenzielle Anordnung:** Die Bytes sind in der Reihenfolge `B_00` bis `B_07` angeordnet. Diese Reihenfolge ist für die Interpretation der Daten, z.B. bei der Konvertierung in größere Datentypen wie `WORD`, `DWORD` oder `LWORD`, entscheidend.

- **TypeHash-Attribut:** Der Datentyp enthält das Attribut `eclipse4diac::core::TypeHash` mit einem leeren Wert (`''`). Dies weist darauf hin, dass für diesen benutzerdefinierten Typ (noch) keine typsichere Prüfung via Hash-Wert konfiguriert wurde.

Zustandsübersicht

Da es sich um einen passiven Datentyp und nicht um einen aktiven Funktionsblock handelt, existiert keine Zustandsmaschine. Der „Zustand“ einer Instanz von `ST08B` ist einfach der aktuelle Wert jedes seiner acht Byte-Member.

Anwendungsszenarien

1. **Datenpaketierung:** Zusammenfassen von acht konfigurierbaren Parametern oder Statuswerten, die gemeinsam von einem Funktionsblock an einen anderen gesendet werden sollen.
2. **Protokollhandling:** Darstellung eines festen Datenrahmenheaders oder -segments in einer seriellen oder Netzwerkkommunikation.
3. **Rohdatenpuffer:** Temporäre Speicherung eines 8-Byte-blocks aus einem Sensor oder einem Kommunikationsinterface vor der weiteren Verarbeitung oder Interpretation.
4. **Konvertierungsbasis:** Nutzung als Quelle oder Ziel für Konvertierungs-Funktionsblöcke, die `ST08B` in andere Datentypen wie `LWORD` (64-Bit) umwandeln oder daraus generieren.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Einfache BYTE-Array:** Ein Array von 8 `BYTE` (`ARRAY[0..7] OF BYTE`) könnte eine ähnliche Funktion erfüllen. `ST08B` bietet jedoch den Vorteil der benannten Zugriffspfade (`B_01`), die im Gegensatz zu Index-basiertem Zugriff (`Array[1]`) die Code-Lesbarkeit und Wartbarkeit verbessern.
- **LWORD:** Der Datentyp `LWORD` repräsentiert ebenfalls 8 Bytes (64 Bit), jedoch als einen einzigen, ununterteilten numerischen Wert. `ST08B` erlaubt den gezielten Zugriff auf jedes einzelne Byte innerhalb dieser 64 Bit.
- **Andere STRUCT-Typen:** Ähnliche strukturierte Typen wie `ST04B` (4 Bytes) oder `ST16B` (16 Bytes) dienen demselben Zweck, aber für unterschiedliche Datenmengen. `ST08B` füllt die Lücke für 8-Byte-strukturierte Daten.

Fazit

Der `ST08B` ist ein einfacher, aber äußerst nützlicher strukturierter Datentyp für die IEC 61499-Programmierung. Er bietet eine klare und strukturierte Methode, um acht Byte Daten zu gruppieren, und fördert damit die Lesbarkeit, Wartbarkeit und Typsicherheit in Anwendungen, die mit festen Datenblöcken oder parametrisierten Datensätzen arbeiten. Seine feste Größe und sequenzielle Anordnung machen ihn zu einem verlässlichen Baustein für Datenkommunikation und -konvertierung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ST16B

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Datentyp `ST16B` ist eine strukturierte Datenkomponente, die im 4diac-Framework definiert ist. Im Gegensatz zu einem Funktionsblock (FB) hat `ST16B` keine eigene Logik oder ausführbaren Funktionen, sondern dient als fester Container für 16 einzelne Byte-Werte. Sein Hauptzweck ist die Bereitstellung einer klar definierten Struktur für die Handhabung von Rohdatenpaketen fester Größe, was besonders in Kommunikationsprotokollen oder bei der Interaktion mit hardwarenahen Schnittstellen von Bedeutung ist.

Schnittstellenstruktur

Da `ST16B` ein Datentyp und kein Funktionsblock ist, besitzt er keine Ereignis- oder Daten-Schnittstellen im herkömmlichen Sinne. Seine „Struktur“ bezieht sich auf die Definition seiner internen Komponenten, d.h. der einzelnen Bytes, aus denen er besteht.

Ereignis-Eingänge

Nicht zutreffend, da `ST16B` ein Datentyp und kein Funktionsblock ist und daher keine Ereignisse verarbeitet.

Ereignis-Ausgänge

Nicht zutreffend, da `ST16B` ein Datentyp und kein Funktionsblock ist und daher keine Ereignisse generiert.

Daten-Eingänge

Nicht zutreffend, da `ST16B` ein Datentyp und kein Funktionsblock ist und daher keine direkten Daten-Eingänge im Kontext von FB-Verbindungen besitzt.

Daten-Ausgänge

Nicht zutreffend, da `ST16B` ein Datentyp und kein Funktionsblock ist und daher keine direkten Daten-Ausgänge im Kontext von FB-Verbindungen besitzt.

Adapter

Nicht zutreffend, da `ST16B` ein Datentyp und kein Funktionsblock ist und daher keine Adapter-Schnittstellen implementiert.

Datenstruktur der Komponente

`ST16B` besteht aus 16 einzelnen Variablen vom Typ `BYTE`, die sequenziell angeordnet sind:

Name	Typ	Beschreibung
B_00	BYTE	BYTE 00
B_01	BYTE	BYTE 01
B_02	BYTE	BYTE 02
B_03	BYTE	BYTE 03
B_04	BYTE	BYTE 04
B_05	BYTE	BYTE 05
B_06	BYTE	BYTE 06
B_07	BYTE	BYTE 07
B_08	BYTE	BYTE 08
B_09	BYTE	BYTE 09
B_10	BYTE	BYTE 10
B_11	BYTE	BYTE 11
B_12	BYTE	BYTE 12
B_13	BYTE	BYTE 13
B_14	BYTE	BYTE 14
B_15	BYTE	BYTE 15

Funktionsweise

`ST16B` selbst besitzt keine aktive Funktionsweise. Es ist ein passiver Datencontainer. Seine Funktion besteht darin, eine definierte Speichervorlage für 16 Byte zu bieten. Jedes Byte (`B_00` bis `B_15`) kann individuell gelesen und beschrieben werden. Dieser strukturierte Zugriff auf einzelne Bytes ermöglicht eine granulare Datenmanipulation und die einfache Zuordnung zu Kommunikationsfeldern oder Speicherbereichen.

Technische Besonderheiten

- **Feste Größe:** Der Datentyp `ST16B` hat eine feste Größe von exakt 16 Bytes. Dies ist vorteilhaft für die Vorhersagbarkeit des Speicherverbrauchs und die Kompatibilität mit Protokollen, die feste Längen erwarten.
- **Direkter Byte-Zugriff:** Jedes der 16 Bytes ist über seinen spezifischen Namen (z.B. `MyVariable.B_00`) direkt zugänglich.
- **Rohdatenbehälter:** Ideal für die Speicherung und den Transport von Rohdaten, deren Inhalt erst später interpretiert werden soll.
- **Architekturunabhängigkeit:** Als Sammlung von BYTEn ist die grundlegende Struktur relativ plattformunabhängig, auch wenn die Interpretation der Bytes in verschiedenen Systemen variieren kann (z.B. Endianness bei größeren Werten, die über mehrere Bytes verteilt sind).

Zustandsübersicht

Nicht zutreffend, da `ST16B` ein Datentyp ist und keine internen Zustände im Sinne eines Funktionsblocks (z.B. Initialisierung, Ausführung, Fehler) besitzt. Die „Zustände“ von `ST16B` sind lediglich die aktuellen Werte, die in seinen 16 Byte-Komponenten gespeichert sind.

Anwendungsszenarien

- **Kommunikationsprotokolle:** Ideal zum Verpacken oder Entpacken von Nachrichten für Protokolle wie Modbus, CAN-Bus, oder proprietäre seriellen Protokollen, bei denen Datenpakete eine feste Größe von 16 Bytes haben.
- **Hardware-Schnittstellen:** Abbildung von Registern oder Datenblöcken in hardwarenahen Schnittstellen, die in festen Byte-Mustern arbeiten.
- **Puffer für binäre Daten:** Verwendung als Puffer für binäre Daten, die von externen Quellen gelesen oder an diese gesendet werden.
- **Zwischenspeicherung von Hashes oder IDs:** Für Anwendungen, die 128-Bit-Hashes (z.B. MD5) oder lange IDs speichern müssen, die genau 16 Bytes umfassen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`ARRAY[0..15] OF BYTE`:** `ST16B` ähnelt einem Array von 16 Bytes. Der Hauptunterschied liegt im Zugriff: Bei `ST16B` erfolgt der Zugriff über benannte Komponenten (`B_00`, `B_01`, etc.), während ein Array den Zugriff über Indizes (`MyArray[0]`, `MyArray[1]`) ermöglicht. Die benannten Komponenten können die Lesbarkeit verbessern, wenn jedes Byte eine spezifische semantische Bedeutung innerhalb der 16 Bytes hat.
- **Andere strukturierte Datentypen:** Im Vergleich zu benutzerdefinierten Strukturen, die verschiedene Datentypen (INT, REAL, BOOL) mischen, ist `ST16B` auf reine Byte-Sammlungen spezialisiert. Es bietet eine flache, gleichartige Struktur.

Fazit

Der Datentyp `ST16B` ist eine einfache, aber effektive Struktur zum Umgang mit Datenblöcken fester Größe von 16 Bytes. Er bietet eine klare und lesbare Möglichkeit, auf individuelle Bytes innerhalb eines solchen Blocks zuzugreifen. Obwohl er kein Funktionsblock ist und keine eigene Logik implementiert, ist er ein wertvolles Werkzeug in 4diac für Anwendungen, die eine genaue Kontrolle über die Byte-Ebene erfordern, insbesondere in der Kommunikation und Hardware-Interaktion.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ST32B

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Datentyp `ST32B` ist ein strukturierter Datentyp (Structured Type) innerhalb der 4diac-IDE, der für die Aggregation von 32 einzelnen Bytes konzipiert ist. Er dient als Vorlage für Datenstrukturen, die eine feste Länge von 32 Byte benötigen und direkten Zugriff auf einzelne Bytes ermöglichen. Dieser Typ ist in der 4diac-IDE unter dem Paket `logiBUS::utils::conversion::types` registriert.

Schnittstellenstruktur

Da `ST32B` ein strukturierter Datentyp und kein Funktionsbaustein ist, besitzt er keine externen Ereignis- oder Datenschnittstellen im herkömmlichen Sinne. Stattdessen definiert er eine interne Struktur von Elementen, auf die in einem Programm zugegriffen werden kann.

Ereignis-Eingänge

Für strukturierte Datentypen nicht zutreffend. `ST32B` besitzt keine Ereignis-Eingänge.

Ereignis-Ausgänge

Für strukturierte Datentypen nicht zutreffend. `ST32B` besitzt keine Ereignis-Ausgänge.

Daten-Eingänge

Für strukturierte Datentypen nicht zutreffend. `ST32B` besitzt keine Daten-Eingänge; seine Elemente sind direkt über den Typnamen zugänglich.

Daten-Ausgänge

Für strukturierte Datentypen nicht zutreffend. `ST32B` besitzt keine Daten-Ausgänge; seine Elemente sind direkt über den Typnamen zugänglich.

Adapter

Für strukturierte Datentypen nicht zutreffend. `ST32B` besitzt keine Adapter.

Funktionsweise

`ST32B` ist ein passiver Datentyp und implementiert keine aktive „Funktionsweise“ oder Logik. Er dient lediglich als Bauplan (Schema) für die Organisation und Speicherung von 32 Byte-Werten unter einem einzigen logischen Namen. Instanzen dieses Datentyps können in anderen Funktionsbausteinen oder Datentypen deklariert und genutzt werden, um Daten strukturiert abzulegen oder zu übergeben.

Die einzelnen Bytes sind wie folgt benannt und können direkt angesprochen werden:

- `B_00`: Byte 00
- `B_01`: Byte 01
- ...
- `B_30`: Byte 30
- `B_31`: Byte 31

Technische Besonderheiten

- **Feste Größe:** Der Datentyp `ST32B` ist auf eine exakte Größe von 32 Bytes festgelegt.
- **Direkter Byte-Zugriff:** Jedes der 32 Bytes ist individuell über seinen Namen ansprechbar (`B_00` bis `B_31`).
- **Grundlegender Typ:** Er besteht ausschließlich aus dem primitiven Datentyp `BYTE`, was ihn für Low-Level-Datenoperationen oder die Darstellung von Rohdaten prädestiniert.
- **Einfachheit:** Die Struktur ist bewusst einfach gehalten, um eine effiziente Nutzung und klare Datenorganisation zu gewährleisten.

Zustandsübersicht

Als passiver Datentyp hat `ST32B` keine internen Zustände im Sinne eines Zustandsautomaten. Der „Zustand“ einer Instanz dieses Typs wird ausschließlich durch die aktuellen Werte der 32 enthaltenen Bytes definiert.

Anwendungsszenarien

- **Rohdatentransport:** Ideal zum Übertragen von fix langen Rohdatenpaketen, beispielsweise von Sensoren oder über Kommunikationsschnittstellen wie SPI, I2C, UART oder Ethernet, wenn die Nutzlast eine feste Länge von 32 Byte hat.
- **Nachrichtenstrukturen:** Kann als Grundlage für einfache Nachrichtenformate dienen, bei denen ein Teil des Nachrichtenrahmens eine definierte Länge von 32 Bytes einnimmt.
- **Speicherabbild:** Zur Abbildung eines kleinen, festen Speicherbereichs oder Registersatzes in einer Steuerung.
- **Konfigurationsdaten:** Speicherung von 32 Byte umfassenden Konfigurationsparametern.

⚖️ Vergleich mit ähnlichen Bausteinen

Ein vergleichbarer Ansatz wäre die Verwendung eines Arrays vom Typ `ARRAY[0..31] OF BYTE`. Der Unterschied liegt hauptsächlich in der Art des Zugriffs und der Semantik:

- **`ST32B` (Struktur):** Ermöglicht den Zugriff auf Bytes über symbolische Namen (`B_00`, `B_01`, ...), was die Lesbarkeit und Wartbarkeit des Codes verbessern kann, insbesondere wenn die einzelnen Bytes eine spezifische Bedeutung haben.
- **`ARRAY[0..31] OF BYTE` (Array):** Ermöglicht den Zugriff über einen numerischen Index (`[0]`, `[1]`, ...), was für Schleifen und generische Operationen vorteilhaft ist.

Während ein Array flexibler in der Dimensionierung sein kann, bietet `ST32B` eine klar definierte, benannte Struktur, die die Absicht hinter jedem Byte expliziter macht, auch wenn die Namen hier generisch sind.

Fazit

Der Datentyp `ST32B` ist ein fundamentaler strukturierter Datentyp in 4diac, der eine einfache und effiziente Möglichkeit bietet, 32 Bytes als eine Einheit zu behandeln. Seine Stärke liegt in der Bereitstellung einer festen, gut definierten Struktur für Rohdaten, was ihn zu einem nützlichen Werkzeug in Anwendungen macht, die präzise Kontrolle über Byte-Sequenzen erfordern. Obwohl er selbst keine aktive Funktionalität besitzt, ist er eine wichtige Grundlage für datenorientierte Architekturen innerhalb von 4diac-Anwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ST64B

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Datentyp `ST64B` ist ein strukturierter Datentyp, der zur Kapselung von 64 einzelnen Byte-Werten dient. Er wurde entwickelt, um eine feste Datenstruktur von 64 Bytes bereitzustellen, die für verschiedene Anwendungsfälle genutzt werden kann, bei denen eine exakt definierte Größe und direkter Zugriff auf einzelne Bytes erforderlich ist. Die ursprüngliche XML-Definition enthielt einen Kommentar, der fälschlicherweise 16 Bytes angab; korrekt sind 64 Bytes.

Strukturdefinition

Der Datentyp `ST64B` ist eine zusammengesetzte Struktur, die ausschließlich aus einzelnen Byte-Variablen besteht. Er besitzt keine ereignis- oder adapterbasierten Schnittstellen, da es sich um eine reine Datenstruktur handelt.

Ereignis-Eingänge

Nicht zutreffend, da `ST64B` ein Datentyp und kein Funktionsblock ist.

Ereignis-Ausgänge

Nicht zutreffend, da `ST64B` ein Datentyp und kein Funktionsblock ist.

Daten-Bestandteile

Die Struktur `ST64B` setzt sich aus den folgenden 64 Byte-Variablen zusammen:

Name	Typ	Beschreibung
B_00	BYTE	BYTE 00
B_01	BYTE	BYTE 01
B_02	BYTE	BYTE 02
B_03	BYTE	BYTE 03
B_04	BYTE	BYTE 04
B_05	BYTE	BYTE 05

B_06	BYTE	BYTE 06
B_07	BYTE	BYTE 07
B_08	BYTE	BYTE 08
B_09	BYTE	BYTE 09
B_10	BYTE	BYTE 10
B_11	BYTE	BYTE 11
B_12	BYTE	BYTE 12
B_13	BYTE	BYTE 13
B_14	BYTE	BYTE 14
B_15	BYTE	BYTE 15
B_16	BYTE	BYTE 16
B_17	BYTE	BYTE 17
B_18	BYTE	BYTE 18
B_19	BYTE	BYTE 19
B_20	BYTE	BYTE 20
B_21	BYTE	BYTE 21
B_22	BYTE	BYTE 22
B_23	BYTE	BYTE 23
B_24	BYTE	BYTE 24
B_25	BYTE	BYTE 25
B_26	BYTE	BYTE 26
B_27	BYTE	BYTE 27
B_28	BYTE	BYTE 28
B_29	BYTE	BYTE 29
B_30	BYTE	BYTE 30
B_31	BYTE	BYTE 31
B_32	BYTE	BYTE 32
B_33	BYTE	BYTE 33
B_34	BYTE	BYTE 34
B_35	BYTE	BYTE 35
B_36	BYTE	BYTE 36
B_37	BYTE	BYTE 37
B_38	BYTE	BYTE 38
B_39	BYTE	BYTE 39
B_40	BYTE	BYTE 40
B_41	BYTE	BYTE 41

B_42	BYTE	BYTE 42
B_43	BYTE	BYTE 43
B_44	BYTE	BYTE 44
B_45	BYTE	BYTE 45
B_46	BYTE	BYTE 46
B_47	BYTE	BYTE 47
B_48	BYTE	BYTE 48
B_49	BYTE	BYTE 49
B_50	BYTE	BYTE 50
B_51	BYTE	BYTE 51
B_52	BYTE	BYTE 52
B_53	BYTE	BYTE 53
B_54	BYTE	BYTE 54
B_55	BYTE	BYTE 55
B_56	BYTE	BYTE 56
B_57	BYTE	BYTE 57
B_58	BYTE	BYTE 58
B_59	BYTE	BYTE 59
B_60	BYTE	BYTE 60
B_61	BYTE	BYTE 61
B_62	BYTE	BYTE 62
B_63	BYTE	BYTE 63

Daten-Ausgänge

Nicht zutreffend, da `ST64B` ein Datentyp ist und seine Bestandteile nicht als „Ausgänge“ im Sinne eines Funktionsblocks fungieren.

Adapter

Nicht zutreffend, da `ST64B` ein Datentyp und kein Funktionsblock ist.

Funktionsweise

Als strukturierter Datentyp hat `ST64B` keine eigene „Funktionsweise“ im Sinne eines aktiven Verhaltens oder einer Logik. Seine primäre Funktion ist es, einen Speicherbereich von 64 Bytes zu definieren und diesen in einer organisierten Form zugänglich zu machen. Dies ermöglicht die Adressierung einzelner Bytes innerhalb der Struktur über ihre benannten Variablen (z.B. `ST64B.B_00`, `ST64B.B_15`). Es dient als passiver Datencontainer.

Technische Besonderheiten

Die Hauptbesonderheit von `ST64B` ist seine feste Größe von exakt 64 Bytes. Jedes Byte ist einzeln über einen eindeutigen Variablennamen ansprechbar. Dies macht ihn ideal für Situationen, in denen eine strikte Datengröße und eine byteweise Manipulation des Inhalts erforderlich sind, wie z.B. bei der Verarbeitung von Kommunikationsprotokollen oder Hardware-Registern. Die sequentielle Benennung der Variablen vereinfacht die Zuordnung zu einem durchgehenden Speicherbereich.

Zustandsübersicht

Als reiner Datentyp besitzt `ST64B` keine Zustände im Sinne einer Zustandsmaschine oder eines Funktionsblocks. Der „Zustand“ eines `ST64B`-Instanz wird ausschließlich durch die aktuellen Werte der 64 enthaltenen Byte-Variablen definiert. Jede Kombination dieser Werte stellt einen gültigen Datenzustand der Struktur dar.

Anwendungsszenarien

`ST64B` eignet sich hervorragend für folgende Anwendungsszenarien:

- **Kommunikationspuffer:** Als Standardpuffer für Nachrichten oder Pakete mit einer festen Größe von 64 Bytes, z.B. in CAN-Bus-Nachrichten (wenn mehrere Frames benötigt werden), serieller Kommunikation oder anderen Protokollen.
- **Hardware-Registerabbild:** Abbildung eines 64 Byte großen Bereichs von Hardware-Registern, bei dem jeder Byte-Variable einem spezifischen Register entspricht.
- **Daten-Serialisierung:** Speicherung von Rohdaten vor der Serialisierung oder nach der Deserialisierung, insbesondere wenn Daten ohne explizite Typinformationen übertragen werden.
- **Generische Datenblöcke:** Handhabung von generischen Datenblöcken fester Größe, die flexibel interpretiert werden können, z.B. als Teil größerer Protokolle oder Datenstrukturen.

⚖️ Vergleich mit ähnlichen Bausteinen

`ST64B` ist ein grundlegender Datentyp und kann nicht direkt mit Funktionsblöcken verglichen werden. Innerhalb der Datentypen ist er vergleichbar mit:

- **Anderen strukturierten Datentypen:** `ST64B` ist eine spezifische Instanz eines `StructuredType`. Es gibt ähnliche Strukturen mit unterschiedlicher Anzahl oder Art von Elementen (z.B. Strukturen mit 8, 16, 32 Bytes oder solche, die Integers, Booleans usw. enthalten). `ST64B` zeichnet sich durch seine rein byteweise Aufteilung aus.
- **Arrays von BYTE:** Ein Array vom Typ `ARRAY[0..63] OF BYTE` würde ebenfalls 64 Bytes speichern. Der Unterschied besteht in der Zugriffsweise: Bei `ST64B` erfolgt der Zugriff über benannte Member (z.B. `myST64B.B_10`), während bei einem Array der Zugriff über einen Index erfolgt (z.B. `myArray[10]`). Die benannten Member können in bestimmten Kontexten für eine klarere Lesbarkeit des Codes oder für die direkte Adressierung spezifischer Byte-Positionen vorteilhaft sein.

Fazit

`ST64B` ist ein einfacher, aber mächtiger strukturierter Datentyp, der eine feste Datenmenge von 64 Bytes bereitstellt. Seine Stärke liegt in der klaren Definition und der direkten Zugänglichkeit jedes einzelnen Bytes, was ihn zu einer ausgezeichneten Wahl für Anwendungen macht, die eine präzise Steuerung und Interpretation von Rohdatenblöcken erfordern, insbesondere im Bereich der Kommunikation und Hardware-Interaktion.

Numerischer Index (utils)

Hier finden Sie alle Übungen des Arbeitsbereichs `utils` in aufsteigender Reihenfolge.

ARR_MAX

ARR_MAX

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsbaustein `ARR_MAX` berechnet den Maximalwert eines eindimensionalen Arrays vom Typ `INT`. Die Implementierung ist in Structured Text (ST) gemäß IEC 61131-3 gehalten und stammt aus dem Package `logiBUS::utils::dyn_arr`. Der Baustein eignet sich für dynamische Arrays beliebiger Größe, wobei die untere und obere Grenze zur Laufzeit ermittelt werden.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Mit Var	Beschreibung
REQ	Event	A	Startet die Berechnung des Maximums. Das Array A muss zum Zeitpunkt des Ereignisses gültig und mit Werten belegt sein.

Ereignis-Ausgänge

Name	Typ	Mit Var	Beschreibung
CNF	Event	(leer), A	Bestätigt das Ende der Berechnung. Das Ereignis wird ausgelöst, sobald der Maximalwert ermittelt wurde. Das Array A bleibt während des Ereignisses gültig.

Daten-Eingänge

Name	Typ	Anfangswert	Beschreibung
A (IN/OUT)	ARRAY[*] OF INT	–	Eindimensionales Integer-Array, dessen Maximum berechnet werden soll. Die Größe wird zur Laufzeit dynamisch ermittelt.

Daten-Ausgänge

Name	Typ	Anfangswert	Beschreibung
MAX	INT	–	Der ermittelte Maximalwert des Arrays A. (Der Name MAX ergibt sich aus der Funktion – in der XML-Definition ist der Ausgang ohne Namen versehen, wird aber als Funktionsrückgabewert behandelt.)

Adapter

Entfallen.

Funktionsweise

1. **Start**: Ein Signal am Ereigniseingang REQ löst die Verarbeitung aus. Das mitgegebene Array A muss zu diesem Zeitpunkt gültig sein.
2. **Initialisierung**: Die Variable i wird auf den Index des ersten Elements (LOWER_BOUND(A,1)) gesetzt. currentMax erhält den Wert dieses ersten Elements.
3. **Iteration**: Eine FOR-Schleife durchläuft alle restlichen Elemente von A (ab Index LOWER_BOUND(A,1)+1 bis UPPER_BOUND(A,1)). Dabei wird mit der Standardfunktion MAX() der aktuelle Maximalwert aktualisiert.
4. **Abschluss**: Nach der Schleife wird currentMax als Rückgabewert der Funktion ARR_MAX gesetzt. Der Ereignisausgang CNF wird gesendet, und das Ausgangssignal MAX (hier als Rückgabewert interpretiert) steht zur Verfügung.

Technische Besonderheiten

- **Dynamische Array-Grenzen**: Die Funktion nutzt LOWER_BOUND und UPPER_BOUND, um die tatsächliche Größe des Arrays zur Laufzeit zu bestimmen. Dadurch funktioniert der Baustein mit Arrays beliebiger Länge, ohne dass eine feste Größe vorgegeben werden muss.
- **Kein Zustandsspeicher**: Der Baustein ist eine reine Funktion ohne internen Zustand. Jeder Aufruf berechnet das Maximum unabhängig vom vorherigen Aufruf.
- **Typisierung**: Der Baustein ist auf den Datentyp INT spezialisiert. Für andere Datentypen (z. B. REAL, LINT) müsste eine separate Variante erstellt werden.
- **Fehlerbehandlung**: Bei einem leeren Array (z. B. untere > obere Grenze) würde der Zugriff auf A[i] undefiniert sein. Der Anwender muss sicherstellen, dass das Array mindestens ein Element enthält.

Zustandsübersicht

Da ARR_MAX als **reine Funktion** ohne Zustandsmaschine (ECA-Algorithmus) implementiert ist, existieren keine expliziten Zustände. Der Ablauf erfolgt strikt sequenziell innerhalb eines einzigen Ereigniszyklus:

- **Warten auf REQ**
- **Berechnung läuft** (kein separates Zustandssignal, da der Vorgang innerhalb eines Zyklus abgeschlossen ist)
- **CNF wird gesendet** (Ausgabe des Maximums)

Eine explizite Zustandsgrafik entfällt daher.

Anwendungsszenarien

- **Datenanalyse**: Ermittlung des Spitzenwerts in Messwertreihen (z. B. Temperatur, Druck, Geschwindigkeit).
- **Überwachung**: Erkennen von Überschreitungen eines Grenzwerts durch Vergleich mit dem Maximum eines Arrays.
- **Signalverarbeitung**: Bestimmung des Maximalpegels in Audio- oder Kommunikationssignalen.
- **Steuerungstechnik**: Auswahl des größten Werts aus mehreren Sensordaten (z. B. für Prioritätsentscheidungen).

Vergleich mit ähnlichen Bausteinen

Baustein	Funktion	Unterschied zu ARR_MAX
ARR_MIN	Minimum eines Arrays	Analog, aber Minimum statt Maximum.
ARR_SUM	Summe der Array-Elemente	Andere mathematische Operation; kein Vergleich zwischen Elementen.
ARR_MEAN	Arithmetischer Mittelwert	Erfordert zusätzlich Division durch die Anzahl; nicht nur Vergleich.
MAX (IEC 61131-3)	Maximum zweier Werte	Arbeitet nur mit Skalaren, nicht mit Arrays. ARR_MAX erweitert dies auf ganze Arrays.

ARR_MAX kombiniert die Array-Durchlauf-Fähigkeit mit der einfachen Maximums-Bestimmung und ist daher ein spezialisierter, aber wiederverwendbarer Baustein für dynamische Arrays.

Fazit

Der Funktionsbaustein ARR_MAX bietet eine kompakte und effiziente Möglichkeit, den Maximalwert eines dynamischen Integer-Arrays zu ermitteln. Die Verwendung von LOWER_BOUND und UPPER_BOUND macht ihn flexibel und robust gegenüber unterschiedlichen Arraygrößen. Durch die reine Funktionsweise ohne Zustandsspeicherung eignet er sich hervorragend für den Einsatz in datenflussorientierten Applikationen der Automatisierungstechnik.

ARR_MIN

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **ARR_MIN** ermittelt das Minimum aller Elemente eines eindimensionalen Arrays vom Typ INT. Der Baustein erwartet ein dynamisches Array (ARRAY[*]) als Ein-/Ausgangsparameter und gibt das kleinste gefundene Element über einen separaten Datenausgang aus. Die Berechnung wird durch das Ereignis REQ angestoßen; das Ergebnis wird nach Fertigstellung mit dem Ereignis CNF quittiert.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Mit Daten
REQ	A

Ereignis-Ausgänge

Ereignis	Mit Daten
CNF	A (und leerer Datenausgang)

Daten-Eingänge

Name	Typ	Kommentar
A	ARRAY[*] OF INT	Eingabearray (Ein-/Ausgang)

Daten-Ausgänge

Name	Typ	Kommentar
(unbenannt)	INT	Minimalwert des Arrays

Adapter

Keine Adapter vorhanden.

Funktionsweise

1. Der Algorithmus initialisiert eine temporäre Variable `currentMin` mit dem Wert des ersten Array-Elements (ermittelt über `LOWER_BOUND(A,1)`).
2. Anschließend wird das Array von Index `LOWER_BOUND(A,1)+1` bis `UPPER_BOUND(A,1)` durchlaufen.
3. In jeder Iteration wird der aktuelle Wert mit `currentMin` verglichen und das Minimum mit der Funktion `MIN()` aktualisiert.
4. Nach Abschluss der Schleife wird der endgültige Minimalwert über den unbenannten Datenausgang ausgegeben und das Ereignis `CNF` gesendet.

Die Berechnung erfolgt rein sequenziell innerhalb eines einzigen Ausführungsschritts, die Laufzeit steigt linear mit der Array-Größe.

Technische Besonderheiten

- **Dynamisches Array**: Der FB arbeitet mit `ARRAY[*] OF INT`, d.h. die tatsächliche Größe des Arrays wird erst zur Laufzeit über die systemeigenen Funktionen `LOWER_BOUND` und `UPPER_BOUND` bestimmt.
- **Paketzuordnung**: Der Baustein gehört zum Paket `logiBUS::utils::dyn_arr` und ist für den Einsatz in Umgebungen vorgesehen, die dynamische Arrays unterstützen.
- **Ein-/Ausgangsparameter**: Der Parameter `A` ist als `VAR_IN_OUT` deklariert – das Array wird sowohl gelesen als auch (hier nicht verändert) über den Ereignisausgang `CNF` weitergegeben.
- **Keine Zustandsmaschine**: Der Funktionsblock arbeitet rein funktional; es gibt keine internen Zustände oder Verzögerungen.

Zustandsübersicht

Der FB kennt nur zwei implizite Zustände:

- **Idle**: Warten auf ein `REQ`-Ereignis.
- **Processing**: Nach Eintreffen von `REQ` wird die Berechnung ausgeführt, unmittelbar danach wird `CNF` gesendet und der FB kehrt in den Idle-Zustand zurück.

Es gibt keine mehrfachen Zustände, Warteschleifen oder zeitgesteuerte Abläufe.

Anwendungsszenarien

- **Sensorfeldanalyse**: Ermittlung der kleinsten Temperatur in einer Reihe von Messwerten.

- **Qualitätskontrolle**: Suche nach dem minimalen Ausschussmaß in einer Produktionscharge.
- **Optimierung**: Bestimmung des günstigsten Preises aus einer Liste von Angeboten.
- **Vergleichsoperationen**: Als Grundbaustein für weitere statistische Auswertungen (z.B. zusammen mit Maximum, Summe).

Vergleich mit ähnlichen Bausteinen

- **MIN (IEC 61131-3)**: Der klassische MIN-Baustein vergleicht zwei skalare Werte. `ARR_MIN` erweitert dieses Konzept auf ein gesamtes Array und spart so eine iterative Verkettung.
- **ARR_MAX**: Ein logisches Gegenstück für das Maximum; der Algorithmus unterscheidet sich nur in der Vergleichsfunktion (`MIN` vs. `MAX`).
- **Statistische Bausteine**: `ARR_MIN` ist ein Spezialfall allgemeiner Aggregationsfunktionen (z.B. SUM, MEAN). Der Fokus liegt auf dem Minimum, was eine einfache und schnelle Implementierung ermöglicht.

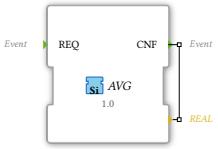
Fazit

`ARR_MIN` bietet eine kompakte und effiziente Lösung zur Bestimmung des Minimums in dynamischen Integer-Arrays. Dank des einfachen Request/Confirm-Protokolls und der sauberen Paketstruktur lässt er sich leicht in vorhandene 4diac-Industrieanwendungen integrieren. Die Verwendung von `LOWER_BOUND`/`UPPER_BOUND` macht ihn unabhängig von der tatsächlichen Array-Größe und erlaubt flexible Anpassungen zur Laufzeit.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

AVG



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **AVG** (Array Average) berechnet den arithmetischen Mittelwert (Durchschnitt) aller Elemente eines Integer-Arrays. Er ist für den Einsatz in der Steuerungstechnik konzipiert und ermöglicht eine schnelle statistische Auswertung von Messwerten oder Prozessdaten.

Schnittstellenstruktur

Ereignis-Eingänge

Event	Mit Daten
`REQ`	`A`

Ereignis-Ausgänge

Event	Mit Daten
`CNF`	*(Ergebnis-Ausgang)*, `A`

Daten-Eingänge

Name	Typ	Initialwert	Kommentar
`A`	`ARRAY[*] OF INT`	–	Array von Ganzzahlen, dessen Durchschnitt berechnet wird (InOut-Variable)

Daten-Ausgänge

Name	Typ	Initialwert	Kommentar
(kein Name)	`REAL`	–	Ergebnis des Durchschnittswerts

Adapter

Keine.

Funktionsweise

1. Der Baustein wird durch das Ereignis `REQ` aktiviert. Dabei wird das Daten-Array `A` übernommen.
2. Intern wird eine Schleife von `LOWER_BOUND(A,1)` bis `UPPER_BOUND(A,1)` durchlaufen, um alle Elemente zu summieren.
3. Die Anzahl der Elemente wird aus `UPPER_BOUND(A,1) - LOWER_BOUND(A,1) + 1` ermittelt.
4. Die Summe wird als `DINT` kumuliert und anschließend durch die Anzahl dividiert. Das Ergebnis wird in einen `REAL`-Wert umgewandelt und am Ausgang bereitgestellt.
5. Nach Abschluss der Berechnung wird das Ereignis `CNF` ausgegeben. Der Ausgangswert steht dann zur Verfügung.

Technische Besonderheiten

- **Dynamische Array-Größe**: Der Baustein akzeptiert Arrays beliebiger Länge (gekennzeichnet durch `ARRAY[*]`), was eine flexible Anpassung an unterschiedliche Datenmengen erlaubt.
- **Typumwandlungen**: Die Summe wird als `DINT` (Doppelwort) geführt, um Überläufe bei großen Arrays zu vermeiden. Die Division erfolgt nach Umwandlung in `REAL`, sodass ein Gleitkommaergebnis entsteht.
- **Keine Zustandsmaschine**: Der Baustein ist rein funktional aufgebaut – auf einen `REQ` folgt unmittelbar ein `CNF`. Es gibt keine internen Zustände oder Verzögerungen.

Zustandsübersicht

Da es sich um einen **kombinatorischen Funktionsblock** handelt, existieren keine expliziten Zustände. Der Ablauf ist:

- Initial: Warten auf `REQ`
- Nach `REQ`: Berechnung und sofortige Ausgabe von `CNF`

Anwendungsszenarien

- **Mittelwertbildung von Sensordaten**: Glättung von Messwerten (z. B. Temperatur, Druck) über einen definierten Messzyklus.
- **Durchschnittsberechnung in Chargenprozessen**: Ermittlung des mittleren Füllstands, Gewichts oder anderer Prozessgrößen.
- **Statistische Auswertung in Qualitätskontrollen**: Beispielsweise Berechnung des Durchmessers mehrerer Prüflinge als Gleitkommawert zur Genauigkeitsanalyse.

Vergleich mit ähnlichen Bausteinen

Merkmal	**AVG**	Typischer IEC-61131-3 AVG-Baustein
Parametrierung	Dynamisch via Array (kein Limit)	Oft statische Array-Größe oder Listen
Ereignisgesteuert	Ja (REQ/CNF)	Nein (reine Funktion)
Ausgabetyp	`REAL`	Meist `REAL` oder `LREAL`
Überlaufschutz	Summe als `DINT`	Abhängig von Implementierung

Der vorliegende Baustein kombiniert die IEC-61499 Ereignissteuerung mit der kompakten Logik eines IEC-61131-3 Durchschnittsbausteins und eignet sich daher besonders für verteilte Automatisierungssysteme.

Fazit

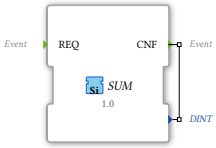
Der FB **AVG** bietet eine einfache und effiziente Möglichkeit, den Durchschnitt eines ganzzahligen Arrays in Echtzeit zu berechnen. Seine dynamische Array-Unterstützung und die klare Ereignisschnittstelle machen ihn zu einem universellen Werkzeug für Mittelwertbildungen in industriellen Steuerungsanwen-

dungen. Die saubere Integration in die 4diac-IDE und die transparente Berechnungslogik gewährleisten eine hohe Wiederverwendbarkeit und Verständlichkeit.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SUM



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock **SUM** ermöglicht die Berechnung der Summe aller Elemente eines dynamischen Arrays. Er wird durch ein Ereignis am Eingang REQ angestoßen und gibt nach erfolgreicher Berechnung das Ergebnis über den Datenausgang sowie ein Bestätigungsereignis über CNF aus.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Beschreibung
REQ	Startet die Summenberechnung des Arrays A.

Ereignis-Ausgänge

Name	Beschreibung
CNF	Signalisiert den Abschluss der Berechnung und die Gültigkeit des Ergebnisses.

Daten-Eingänge

Es sind keine reinen Dateneingänge vorhanden. Der benötigte Array wird über den **InOut**-Port `A` bereitgestellt.

Daten-Ausgänge

Name	Typ	Beschreibung
(leer)	DINT	Resultierende Summe aller Array-Elemente (Typ DINT).

Adapter

Keine.

Funktionsweise

Der FB `SUM` akzeptiert ein eindimensionales Array `A` vom Typ `INT` mit beliebiger Länge (deklariert als `ARRAY[*]`). Bei Aktivierung des Ereignisses `REQ` wird der Inhalt des Arrays elementweise aufsummiert. Die Summe wird in einer temporären Variable vom Typ `DINT` akkumuliert, um einen Überlauf zu vermeiden. Nach der vollständigen Iteration über alle gültigen Indizes (von `LOWER_BOUND(A,1)` bis `UPPER_BOUND(A,1)`) wird das Ergebnis an den Ausgang (Datentyp `DINT`) übergeben und das Ereignis `CNF` ausgelöst.

Wichtiger Hinweis: Obwohl `A` als **InOut**-Variable deklariert ist, wird ihr Wert während der Berechnung nicht verändert. Sie fungiert in diesem FB ausschließlich als Eingang.

Technische Besonderheiten

- **Dynamische Array-Größe:** Der FB unterstützt Arrays variabler Länge durch die Verwendung der Funktionen `LOWER_BOUND` und `UPPER_BOUND`, wie von IEC 61131-3 vorgesehen.

- **Typerweiterung:** Die Summe wird in einem `DINT`-Wert (32-Bit) zurückgegeben, sodass auch größere Summen von `INT`-Werten (16-Bit) ohne Überlauf abgebildet werden können.
- **Ereignisgesteuerte Ausführung:** Die Berechnung erfolgt nur bei Bedarf, was eine effiziente Ressourcennutzung ermöglicht.

Zustandsübersicht

Der FB besitzt keinen expliziten Zustandsautomaten. Das Verhalten lässt sich wie folgt beschreiben:

1. **Warten:** Kein Ereignis aktiv – Ausgangszustand.
2. **Berechnung:** Eintritt nach Empfang von `REQ`; Summierung des Arrays.
3. **Bestätigung:** Nach Abschluss der Berechnung wird `CNF` gesendet; Rücksprung in den Wartezustand.

Anwendungsszenarien

- Aufsummieren von Messwerten eines Sensors über einen bestimmten Zeitraum.
- Berechnung von Gesamtsummen in einer Produktionsanlage (z. B. Stückzahlen, Füllmengen).
- Vorverarbeitung von Datenarrays vor einer weiteren logischen oder arithmetischen Verarbeitung.

Vergleich mit ähnlichen Bausteinen

- **SUM in IEC 61131-1:** Der Baustein ist stark an die gleichnamige Funktion aus der IEC 61131-3 angelehnt, ermöglicht jedoch zusätzlich die ereignisgesteuerte Ausführung und die direkte Einbindung in das 4diac-Ereignismodell.
- **ADD / MULT:** Während `ADD` oder `MULT` einzelne Zahlenpaare verarbeiten, erlaubt `SUM` die Verarbeitung eines gesamten Arrays in einem Schritt, was die Applikationslogik vereinfacht.

Fazit

Der Funktionsblock `SUM` bietet eine einfache und performante Möglichkeit, die Summe aller Elemente eines dynamischen Arrays zu berechnen. Durch die ereignisgesteuerte Ausführung und die robuste Handhabung unterschiedlicher Array-Größen eignet er sich besonders für industrielle Steuerungsaufgaben, bei denen Datenmengen aggregiert werden müssen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

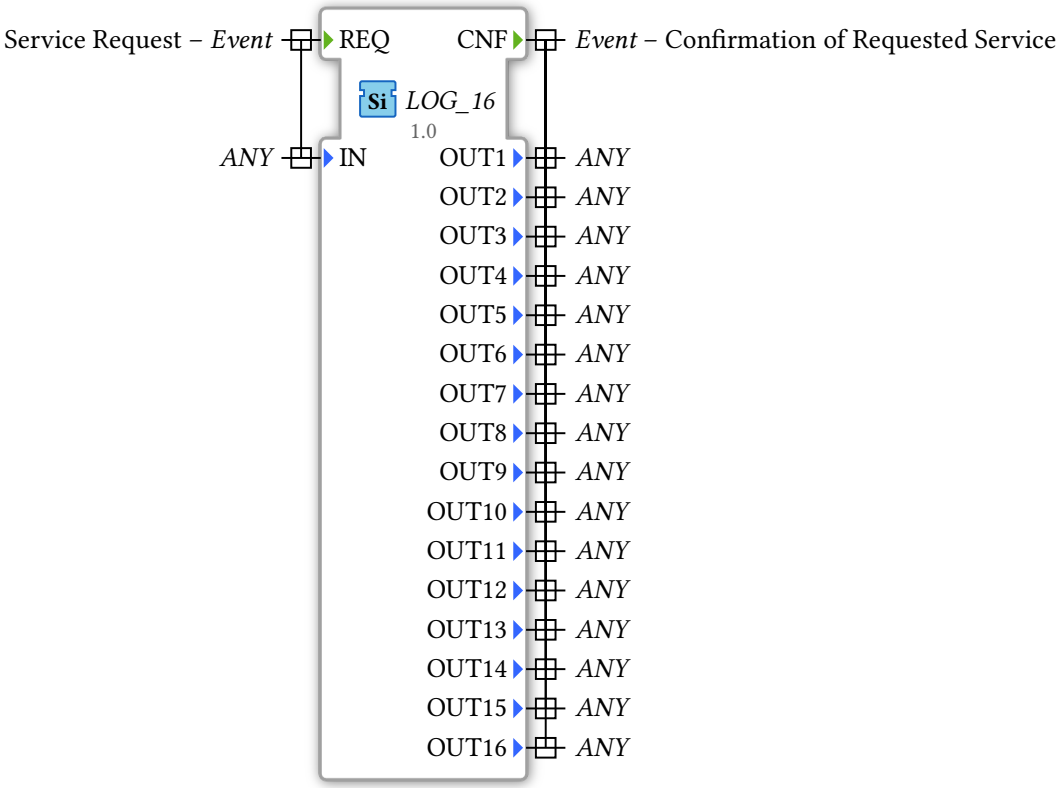
logging

LOG_16

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `LOG_16` ist ein Ring-Logger, der für den zyklischen Aufzeichnung von Daten beliebigen Typs (`ANY`) konzipiert ist. Er dient dazu, eingehende Werte in einem Puffer mit 16 Speicherplätzen zu halten, wobei ältere Einträge bei neuen Aufrufen überschrieben werden (Ringpuffer-Prinzip). Dieser Baustein ist besonders für die Protokollierung von Prozessdaten oder Zuständen in Echtzeitsteuerungen geeignet.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Service Request)**: Löst einen Log-Vorgang aus. Bei Eintreten dieses Ereignisses wird der aktuelle Wert am Daten-Eingang `IN` in den Ringpuffer übernommen.

Ereignis-Ausgänge

- **CNF (Confirmation of Requested Service)**: Wird nach erfolgreicher Verarbeitung des `REQ`-Ereignisses ausgelöst. Dieses Ereignis bestätigt den Log-Vorgang und stellt gleichzeitig alle 16 gespeicherten Werte an den Datenausgängen bereit.

Daten-Eingänge

- **IN (ANY)**: Der Datenwert, der bei einem `REQ`-Ereignis in den Ringpuffer geschrieben werden soll. Der Datentyp ist beliebig (`ANY`).

Daten-Ausgänge

- **OUT1 bis OUT16 (ANY)**: Die 16 Ausgänge, die den gesamten aktuellen Inhalt des Ringpuffers repräsentieren. `OUT1` enthält dabei den jüngsten Eintrag (das letzte geloggte `IN`), `OUT16` den ältesten. Bei jedem neuen Log-Vorgang (`REQ`) werden alle Werte im Puffer um eine Position verschoben.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Der `LOG_16` implementiert einen Ringpuffer (First-In-First-Out mit fester Größe) mit 16 Elementen. Bei jedem eintreffenden `REQ`-Ereignis wird der folgende Algorithmus ausgeführt:

1. Der aktuelle Wert am Eingang `IN` wird als neuester Eintrag gespeichert.
2. Alle bisher gespeicherten Werte werden um eine Position „nach hinten“ (in Richtung `OUT16`) verschoben.
3. Der Wert, der sich zuvor an Position 16 (`OUT16`) befand, wird verworfen.
4. Das Bestätigungsereignis `CNF` wird ausgelöst.

5. Die neuen 16 Pufferinhalte werden an den Ausgängen `OUT1` (neuester) bis `OUT16` (ältester) ausgegeben.

Technische Besonderheiten

- **Generischer Datentyp**: Die Verwendung des `ANY`-Datentyps für Ein- und Ausgänge macht den Baustein extrem flexibel. Er kann mit beliebigen Datentypen (z.B. `BOOL`, `INT`, `REAL`, `STRING` oder sogar strukturierten Typen) instanziiert und verwendet werden.
- **Feste Puffergröße**: Die Puffergröße ist auf 16 Einträge festgelegt und nicht konfigurierbar.
- **Sofortige Ausgabe**: Bei jedem Log-Vorgang wird der gesamte Pufferinhalt an den Ausgängen aktualisiert und mit dem `CNF`-Ereignis bestätigt.

Zustandsübersicht

Der Baustein besitzt keinen persistenten internen Zustand im Sinne einer Zustandsmaschine außer dem Ringpuffer selbst. Sein Verhalten ist rein reaktiv: Auf ein `REQ`-Ereignis folgt stets eine Pufferaktualisierung und die Ausgabe von `CNF` mit den aktuellen Daten.

Anwendungsszenarien

- **Protokollierung von Prozesswerten**: Kurzfristige Aufzeichnung von Sensordaten (z.B. Temperaturverlauf der letzten 16 Zyklen).
- **Fehlerhistorie**: Speicherung der letzten 16 Fehlercodes oder Alarmmeldungen.
- **Datenvorverarbeitung**: Bereitstellung eines gleitenden Fensters über die letzten 16 Werte für nachfolgende Berechnungen (z.B. in einem weiteren FB).
- **Debugging**: Einfache Überwachung von Variablenverläufen während der Entwicklung und Inbetriebnahme.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`E_DELAY` / Verzögerungsbausteine**: Diese Bausteine geben einen Eingangswert erst nach einer definierten Zeit verzögert wieder aus. Der `LOG_16` hingegen speichert eine Historie mehrerer Werte und gibt sie sofort, aber geordnet nach Aktualität, aus.
- **`FIFO`-Bausteine**: Klassische FIFO-Speicher (First-In-First-Out) haben oft variable Längen und ein separates Lese-/Schreib-Interface. Der `LOG_16` ist ein spezieller FIFO mit fester Länge (16), der bei jedem Schreibvorgang automatisch den gesamten Inhalt ausgibt und überschreibt.
- **Einfache `LOG`-Bausteine**: Einfache Logger ohne Puffer schreiben typischerweise nur einen einzelnen Wert. Die Stärke des `LOG_16` liegt in der ringförmigen Historie.

🛠️ Zugehörige Übungen

- Uebung_122
- Uebung_122b

Fazit

Der `LOG_16` ist ein nützlicher und generischer Funktionsblock für grundlegende Logging- und Pufferaufgaben in 4diac FORTE-Applikationen. Seine Stärken liegen in der Einfachheit, der generischen Typunterstützung und dem deterministischen Verhalten. Die feste Puffergröße von 16 ist für viele Überwachungsaufgaben ausreichend, für Anwendungen, die eine andere Puffertiefe oder ein selektives Lesen benötigen, sind jedoch angepasste oder erweiterte Bausteine erforderlich.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

math

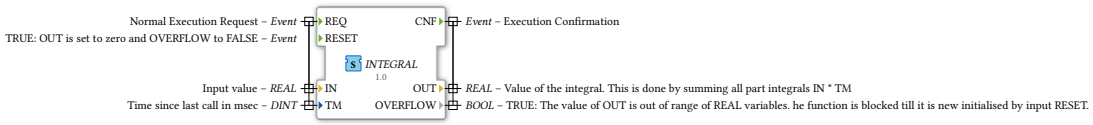
INTEGRAL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `INTEGRAL` berechnet näherungsweise das zeitliche Integral eines Eingangssignals. Die Integration erfolgt durch die Summation von Teilflächen, die sich aus dem Produkt des Eingangswertes und der seit dem letzten Aufruf verstrichenen Zeit ergeben. Der Baustein ist nützlich für Anwendungen, in

denen kumulative Werte wie verbrauchte Energie, geförderte Mengen oder zurückgelegte Wege bestimmt werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die normale Ausführung der Integration. Die mit diesem Ereignis verbundenen Daten `IN` und `TM` werden verarbeitet.
- **RESET**: Setzt den Integrator zurück. Der Ausgang `OUT` wird auf Null gesetzt und das Überlauf-Flag `OVERFLOW` auf `FALSE`.

Ereignis-Ausgänge

- **CNF**: Signalisiert die Bestätigung der Ausführung. Dieses Ereignis wird sowohl nach einem `REQ` als auch nach einem `RESET` ausgelöst und liefert die aktuellen Werte von `OUT` und `OVERFLOW`.

Daten-Eingänge

- **IN** (`REAL`, Initialwert: `0.0`): Der Eingangswert, der integriert werden soll.
- **TM** (`DINT`, Initialwert: `0`): Die seit dem letzten Aufruf verstrichene Zeit in Millisekunden (ms). Dieser Wert muss positiv sein, damit eine Integration stattfindet.

Daten-Ausgänge

- **OUT** (`REAL`, Initialwert: `0.0`): Der aktuelle Wert des Integrals. Er wird durch Aufsummieren aller Teilintegrale (`IN * TM`) berechnet. Die Einheit ist `[IN] * Sekunden`, da der Zeitwert `TM` intern von Millisekunden in Sekunden umgerechnet wird.
- **OVERFLOW** (`BOOL`, Initialwert: `FALSE`): Zeigt einen Überlauf an. Wenn dieser Ausgang `TRUE` ist, hat der Wert von `OUT` den darstellbaren Bereich des `REAL`-Datentyps verlassen. Die Integrationsfunktion ist blockiert, bis sie durch ein `RESET`-Ereignis neu initialisiert wird.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der `INTEGRAL`-FB besitzt zwei primäre Betriebszustände, die durch die eintreffenden Ereignisse gesteuert werden:

1. **Integration (`REQ`)**: Bei einem `REQ`-Ereignis wird der Algorithmus `REQ` ausgeführt. Zunächst wird geprüft, ob kein vorheriger Überlauf (`OVERFLOW = FALSE`) vorliegt. Anschließend wird eine Überlaufprüfung durchgeführt: Falls die Addition des nächsten Teilintegrals (`IN * TM`) zum aktuellen `OUT`-Wert die Grenzen des `REAL`-Datentyps (ca. ±1.0E38) überschreiten würde, wird `OVERFLOW` auf `TRUE` gesetzt. Andernfalls, und nur wenn die Zeitdifferenz `TM` größer als 0 ist, wird das Teilintegral berechnet, in Sekunden umgerechnet (`/ 1000.0`) und zum Gesamtintegral `OUT` addiert.
2. **Reset (`RESET`)**: Bei einem `RESET`-Ereignis wird der Algorithmus `RESET` ausgeführt. Dieser setzt den Integralwert `OUT` auf `0.0` und das Überlauf-Flag `OVERFLOW` auf `FALSE` zurück.

Nach der Ausführung des jeweiligen Algorithmus wird stets ein `CNF`-Ausgangsereignis mit den aktuellen Werten generiert.

Technische Besonderheiten

- **Überlaufbehandlung**: Der Baustein verfügt über eine integrierte Überlauferkennung. Bei Erkennung eines potenziellen Überlaufs wird die Integration gestoppt (`OVERFLOW = TRUE`), um undefiniertes Verhalten zu vermeiden. Ein manueller Reset ist erforderlich, um den Betrieb fortzusetzen.
- **Zeitbasis**: Die Integration basiert auf der vom Anwender bereitgestellten Zeitdifferenz `TM` in Millisekunden. Der Baustein selbst besitzt keinen internen Timer. Dies ermöglicht eine flexible Anpassung an verschiedene Zykluszeiten des umgebenden Steuerungsprogramms.
- **Initialisierung**: Bei der ersten Initialisierung des Bausteins sind `OUT = 0.0` und `OVERFLOW = FALSE`.

Zustandsübersicht

Der Funktionsblock ist ein Simple FB und besitzt explizit definierte Ausführungszustände (ECStates) für die beiden möglichen Ereignisse:

- **Zustand REQ**: Wird bei Eintreffen des `REQ`-Ereignisses aktiviert. Führt den Integrationsalgorithmus aus.
- **Zustand RESET**: Wird bei Eintreffen des `RESET`-Ereignisses aktiviert. Führt den Reset-Algorithmus aus. Beide Zustände führen nach ihrer Ausführung zur Ausgabe des `CNF`-Ereignisses.

Anwendungsszenarien

- **Energiemessung**: Berechnung des Gesamtenergieverbrauchs aus einer momentanen Leistungsmessung (`IN` = Leistung in kW, `OUT` = Energie in kWh).
- **Durchflussmessung**: Ermittlung der insgesamt geförderten Flüssigkeits- oder Gasmenge aus einem Durchflusssensor (`IN` = Durchfluss in l/min, `OUT` = Volumen in l).
- **Geschwindigkeitsintegration**: Bestimmung der zurückgelegten Strecke aus einem Geschwindigkeitssignal (`IN` = Geschwindigkeit in m/s, `OUT` = Weg in m).
- **Ladungsberechnung**: Integration eines Stromsignals zur Ermittlung der geflossenen Ladung (`IN` = Strom in A, `OUT` = Ladung in Ah).

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren Summierbausteinen (z.B. `ADD`) zeichnet sich `INTEGRAL` durch die explizite Berücksichtigung der Zeit als Integrationsvariable aus. Im Gegensatz zu komplexeren Regelbausteinen mit integrierten Timern (wie z.B. bestimmten PID-Implementierungen) ist `INTEGRAL` ein reiner, zeitdiskreter Integrator, der die Zeitdifferenz als Eingangsvariable erwartet und somit extern getaktet wird. Dies macht ihn flexibler in Systemen mit variabler oder nicht-periodischer Zykluszeit.

🛠️ Zugehörige Übungen

- Uebung_072c

Fazit

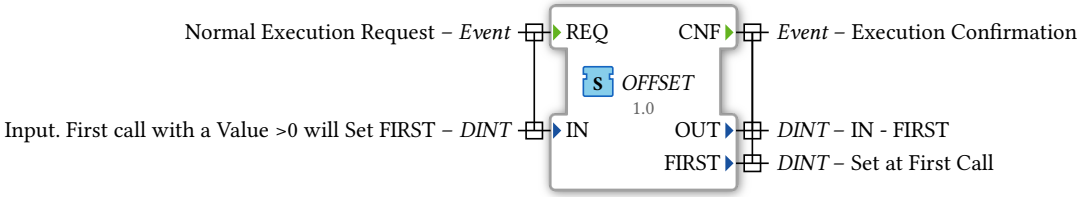
Der `INTEGRAL`-Funktionsblock ist ein grundlegendes und robustes Werkzeug zur zeitlichen Integration von Signalen in IEC 61499-Steuerungen. Seine klare Schnittstelle, die integrierte Überlaufsicherung und die flexible Handhabung der Zeitdifferenz machen ihn zu einer zuverlässigen Komponente für eine Vielzahl von Mess- und Regelaufgaben. Die Notwendigkeit, die verstrichene Zeit extern zu ermitteln und bereitzustellen, bietet maximale Flexibilität, erfordert aber auch eine korrekte Implementierung im umgebenden Anwendungsprogramm.

OFFSET

• – ◦ • – ◦ • – ◦ •

Einleitung

Der OFFSET-Funktionsblock dient dazu, einen Offset-Wert zu ermitteln und anzuwenden. Bei der ersten Verarbeitung eines Ereignisses mit einem Eingangswert größer als Null wird dieser Wert als Referenzpunkt (`FIRST`) gespeichert. Bei jedem Aufruf berechnet der Baustein die Differenz zwischen dem aktuellen Eingangswert und diesem gespeicherten Referenzwert. Dieser Baustein ist nützlich, um relative Änderungen gegenüber einem initialen Startwert zu messen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die normale Ausführung des Bausteins. Wird mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Berechnung. Wird mit den Daten-Ausgängen `OUT` und `FIRST` verknüpft.

Daten-Eingänge

- **IN** (DINT): Der Eingangswert. Beim ersten Aufruf, bei dem dieser Wert größer als 0 ist, wird er als Referenzwert (`FIRST`) gespeichert. Initialwert ist 0.

Daten-Ausgänge

- **OUT** (DINT): Das Ergebnis der Berechnung `IN - FIRST`. Initialwert ist 0.
- **FIRST** (DINT): Der beim ersten relevanten Aufruf gespeicherte Referenzwert. Initialwert ist 0.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der OFFSET-Baustein ist ein einfacher Funktionsblock mit einem einzigen Zustand (`REQ`) und einem Algorithmus gleichen Namens.

Bei jedem Eintreffen eines `REQ`-Ereignisses wird der Algorithmus `REQ` ausgeführt:

1. **Prüfung und Setzen des Referenzwerts:** Es wird geprüft, ob der Ausgang `FIRST` noch seinen Initialwert 0 hat. Falls ja **und** der aktuelle Eingangswert `IN` größer als 0 ist, wird `FIRST` auf den Wert von `IN` gesetzt. Diese Bedingung (`IN > 0`) ist im Algorithmus implizit durch die Logik enthalten, da `FIRST` nur bei der ersten Verarbeitung eines `IN > 0` von 0 auf einen anderen Wert geändert wird.
2. **Berechnung des Offsets:** Unabhängig vom vorherigen Schritt wird der Ausgang `OUT` als Differenz zwischen dem aktuellen Eingang `IN` und dem gespeicherten Referenzwert `FIRST` berechnet (`OUT := IN - FIRST`).
3. **Bestätigung:** Nach der Berechnung wird das `CNF`-Ereignis ausgelöst, wobei die aktuellen Werte von `OUT` und `FIRST` mitgegeben werden.

Technische Besonderheiten

- Der Referenzwert `FIRST` wird genau einmal gesetzt, und zwar beim ersten `REQ`-Ereignis, bei dem der Daten-Eingang `IN` einen Wert größer als 0 aufweist. Solange `IN` bei den ersten Aufrufen 0 oder kleiner ist, bleibt `FIRST` auf 0.
- Nachdem `FIRST` einmal gesetzt wurde, bleibt er für die gesamte weitere Laufzeit des Bausteins unverändert.
- Die Berechnung `OUT = IN - FIRST` findet bei *jedem* Aufruf statt, auch wenn `FIRST` noch 0 ist. Dies bedeutet, dass `OUT` solange identisch mit `IN` ist, bis der Referenzwert gesetzt wird.

Zustandsübersicht

Der Baustein besitzt einen einzigen, namensgebenden Zustand:

- **REQ:** Dieser Zustand ist immer aktiv. Jedes eintreffende `REQ`-Ereignis führt zur Ausführung des zugehörigen Algorithmus und löst sofort das `CNF`-Ereignis aus. Es gibt keine internen Zustandsübergänge.

Anwendungsszenarien

- **Inkrementelle Weg- oder Positionsmessung:** Setzen eines Nullpunkts (`FIRST`) nach einer Referenzfahrt und anschließende Messung der relativen Bewegung (`OUT`).
- **Kompensation von Grundlasten oder Nullabweichungen:** Erfassen eines initialen Basiswerts (z.B. Sensor-Grundlast) und Berechnung der reinen Änderung gegenüber dieser Basis.
- **Relative Zeitmessung:** Speichern eines Startzeitpunkts und Berechnung der vergangenen Zeit seit diesem Start.

⚖️ Vergleich mit ähnlichen Bausteinen

- **SUB / DINT_SUB:** Ein reiner Subtrahierer berechnet stets die Differenz zweier aktueller Eingangswerte. OFFSET hingegen speichert einen der Werte intern als konstanten Referenzpunkt.
- **DELAY / TON:** Zeitverzögerungsbausteine arbeiten auf der Zeitachse. OFFSET arbeitet mit Datenwerten und hat kein zeitbasiertes Verhalten.
- **SR / RS:** Speicherbausteine (Set/Reset) halten einen binären Zustand. OFFSET speichert einen ganzzahligen (`DINT`) Wert und verwendet ihn für eine arithmetische Operation.

Fazit

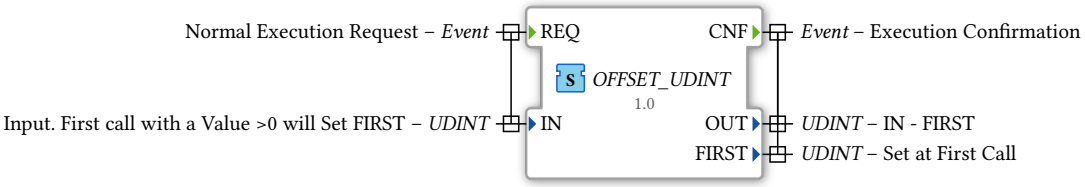
Der OFFSET-Baustein ist ein spezialisierter und einfacher Funktionsblock zur Ermittlung eines relativen Offsets. Seine Stärke liegt in der einmaligen, automatischen Erfassung eines Start- oder Referenzwerts und der fortlaufenden Berechnung der Abweichung davon. Er eignet sich besonders für Anwendungen, bei denen ein System initial „genullt“ werden muss und anschließend nur noch Änderungen relevant sind.

OFFSET_UDINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `OFFSET_UDINT` dient dazu, einen Offsetwert zu berechnen. Bei der ersten Ausführung mit einem Eingangswert größer als Null speichert der Baustein diesen Wert als Referenz (`FIRST`). Bei jedem Aufruf wird die Differenz zwischen dem aktuellen Eingangswert und diesem gespeicherten Referenzwert als Ausgang (`OUT`) berechnet. Dies ist besonders nützlich, um relative Änderungen gegenüber einem initialen Startwert zu ermitteln.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die normale Ausführung des Algorithmus. Wird mit dem Dateneingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert die Beendigung der Berechnung. Wird mit den Datenausgängen `OUT` und `FIRST` verknüpft.

Daten-Eingänge

- **IN** (UDINT): Der Eingangswert. Beim ersten Aufruf mit einem Wert >0 wird dieser Wert in `FIRST` gespeichert. Initialwert ist `0`.

Daten-Ausgänge

- **OUT** (UDINT): Das Ergebnis der Berechnung `IN - FIRST`. Initialwert ist `0`.
- **FIRST** (UDINT): Der beim ersten Aufruf (mit `IN` > 0) gespeicherte Referenzwert. Initialwert ist `0`.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der `OFFSET_UDINT` ist ein einfacher Funktionsblock (Simple FB) mit einem einzigen Zustand (`REQ`). Bei jedem Eintreffen des `REQ`-Ereignisses wird der zugehörige Algorithmus `REQ` ausgeführt.

Der Algorithmus überprüft zunächst, ob der Ausgang `FIRST` noch den Initialwert `0` hat. Falls ja, wird der aktuelle Wert von `IN` in `FIRST` gespeichert. Anschließend wird die Differenz `OUT = IN - FIRST` berechnet. Nach Abschluss der Berechnung wird das `CNF`-Ereignis ausgelöst, wobei die aktualisierten Werte von `OUT` und `FIRST` ausgegeben werden.

Wichtig: Die Prüfung `FIRST = 0` stellt sicher, dass der Wert nur beim *ersten* Aufruf gesetzt wird, bei dem `IN` einen Wert größer 0 hat. Ein späteres `IN` von `0` ändert `FIRST` nicht mehr.

Technische Besonderheiten

- **Datentyp:** Der Block arbeitet ausschließlich mit vorzeichenlosen 32-Bit-Ganzzahlen (`UDINT`).
- **Initialisierung:** Sowohl `FIRST` als auch `OUT` sind mit `0` initialisiert. Der Block ist nach dem Start sofort betriebsbereit.
- **Einmalige Setzung:** Die Variable `FIRST` wird genau einmal gesetzt (beim ersten `REQ` mit `IN` > 0) und bleibt danach für die Lebensdauer der FB-Instanz konstant, es sei denn, der FB wird neu initialisiert.

Zustandsübersicht

Der Funktionsblock besitzt einen einzigen, stabilen Zustand:

- **REQ:** Wartet auf das eingehende `REQ`-Ereignis. Bei Empfang wird der Algorithmus ausgeführt und unmittelbar danach wird das `CNF`-Ereignis ausgelöst. Der FB verbleibt im Zustand `REQ`.

Anwendungsszenarien

- **Referenzpunktbildung in Messketten:** In einem System, das absolute Positionswerte (z.B. von einem Encoder) liefert, kann dieser Block den ersten gültigen Messwert als Nullpunkt (Offset) speichern. Alle folgenden Werte werden relativ zu diesem Startpunkt ausgegeben.
- **Initialwert-Kompensation:** Zur Eliminierung eines systembedingten Grundoffsets (z.B. einer Vorspannung) bei der ersten gültigen Messung.
- **Zählbeginn-Normierung:** Wenn ein Zähler nicht bei 0 startet, kann der erste Wert gespeichert und von allen folgenden Werten subtrahiert werden, um eine Zählung ab 0 zu simulieren.

⚖️ Vergleich mit ähnlichen Bausteinen

- **SUB / DIF Bausteine:** Standard-Subtraktionsbausteine berechnen stets die Differenz zwischen zwei variablen Eingängen. `OFFSET_UDINT` hingegen friert einen der Operanden (`FIRST`) nach der ersten Operation ein und verwendet ihn fortan als festen Subtrahenden.
- **Schieberegister (FIFO) oder Verzögerungsbausteine:** Diese speichern eine Historie von Werten. `OFFSET_UDINT` speichert dagegen nur einen einzigen, initialen Wert dauerhaft.
- **Bausteine mit Merker (SR, RS):** Diese speichern einen binären Zustand. `OFFSET_UDINT` speichert einen spezifischen numerischen (`UDINT`) Startwert.

🛠️ Zugehörige Übungen

- Uebung_072c

Fazit

Der `OFFSET_UDINT` ist ein spezialisierter und effizienter Funktionsblock für Anwendungen, bei denen ein einmalig gemessener Startwert als Basis für alle folgenden relativen Berechnungen dienen soll. Seine Stärke liegt in der Einfachheit und der zuverlässigen, einmaligen Setzung des Referenzwertes, was manuelle Logik zur Erkennung des „ersten Zyklus“ überflüssig macht. Er ist ideal für Initialisierungs- und Normierungsaufgaben in Steuerungsanwendungen.

quarter

🎧 Podcast

- Das Geheimnis des 2-Bit-Quarter: Effizienz im CAN-Bus für Nutzfahrzeuge
- QUARTER

BOOLS_TO_QUARTERS

Composite FB wrapping other FBs
REQ
I_00
I_01
I_02
I_03
I_04
I_05
I_06
I_07
I_08
I_09
I_10
I_11
I_12
I_13
I_14
I_15
CNF
QB_00
QB_01
QB_02
QB_03
QB_04
QB_05
QB_06
QB_07
QB_08
QB_09
QB_10
QB_11
QB_12
QB_13
QB_14
QB_15
BOOL_TO_QUARTER_00
BOOL_TO_QUARTER_01
BOOL_TO_QUARTER_02
BOOL_TO_QUARTER_03
BOOL_TO_QUARTER_04
BOOL_TO_QUARTER_05
BOOL_TO_QUARTER_06
BOOL_TO_QUARTER_07
BOOL_TO_QUARTER_08
BOOL_TO_QUARTER_09
BOOL_TO_QUARTER_10
BOOL_TO_QUARTER_11
BOOL_TO_QUARTER_12
BOOL_TO_QUARTER_13
BOOL_TO_QUARTER_14
BOOL_TO_QUARTER_15
REQ
CNF
B
BOOL_TO_QUARTER
I

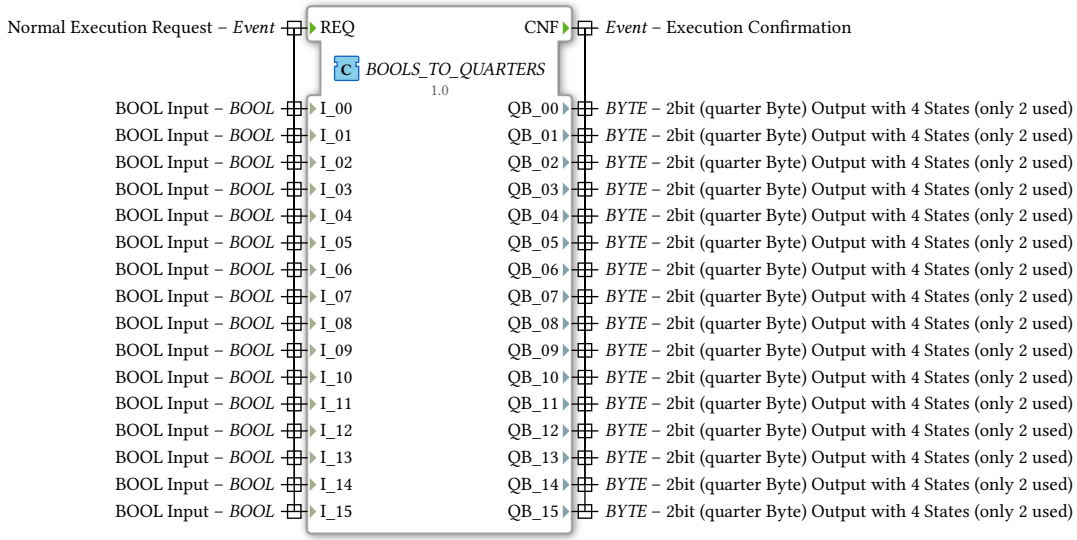
🎧 Podcast

- QUARTER

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `BOOLS_TO_QUARTERS` ist ein zusammengesetzter Baustein (Composite FB), der 16 einzelne boolesche Eingangssignale in ein spezielles 2-Bit-Format, genannt „Quarter Byte“, umwandelt. Er dient als Bündelung und serielle Abarbeitung mehrerer grundlegender Konvertierungsfunktionen und ist für Anwendungen vorgesehen, in denen viele binäre Zustände in ein kompaktes, mehrwertiges Steuerformat überführt werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ (Normal Execution Request):** Startet die Verarbeitungskette. Bei einem eingehenden REQ-Ereignis werden alle mitgeführten Daten-Eingänge (`I_00` bis `I_15`) eingelesen und die Konvertierung initiiert.

Ereignis-Ausgänge

- **CNF (Execution Confirmation):** Dieses Ereignis wird ausgegeben, nachdem alle 16 internen Konvertierungen abgeschlossen sind. Es bestätigt die Beendigung der Operation und liefert die berechneten Quarter-Byte-Werte (`QB_00` bis `QB_15`) an die nachgelagerte Applikation.

Daten-Eingänge

- **I_00 bis I_15 (BOOL Input):** 16 unabhängige boolesche Eingänge (`BOOL`). Jeder repräsentiert einen binären Schaltzustand (TRUE/FALSE). Der Initialwert aller Eingänge ist `FALSE`.

Daten-Ausgänge

- **QB_00 bis QB_15 (2bit (quarter Byte) Output):** 16 Ausgänge vom Typ `BYTE`. Jeder Ausgang kodiert das Ergebnis der Konvertierung des entsprechenden booleschen Eingangs in ein Quarter-Byte. Ein Quarter-Byte nutzt nur die unteren zwei Bits eines Bytes und kann theoretisch vier Zustände darstellen. In dieser Implementierung werden primär zwei Zustände genutzt, die durch die Konstanten `quarter::COMMAND_DISABLE` und ihr Gegenstück definiert sind. Der Initialwert aller Ausgänge ist `quarter::COMMAND_DISABLE`.

Adapter

Dieser Funktionsblock verwendet keine Adapter-Schnittstellen.

Funktionsweise

BOOLS_TO_QUARTERS ist ein Composite FB, der intern aus 16 Instanzen eines Basisfunktionsblocks BOOL_TO_QUARTER aufgebaut ist. Jede Instanz ist für die Konvertierung eines einzelnen booleschen Wertes zuständig.

Die Funktionsweise folgt einem seriellen Kettenprinzip:

1. Das eingehende REQ-Ereignis triggert die erste interne Instanz BOOL_TO_QUARTER_00.
2. Diese Instanz liest ihren zugeordneten Daten-Eingang I_00, führt die Konvertierung durch und setzt ihren Ausgang QB_00.
3. Nach Abschluss ihrer Operation generiert BOOL_TO_QUARTER_00 ein CNF-Ereignis, das direkt als REQ-Ereignis für die nächste Instanz (BOOL_TO_QUARTER_01) weitergeleitet wird.
4. Dieser Prozess setzt sich kaskadenartig durch alle 16 Instanzen fort.
5. Die letzte Instanz (BOOL_TO_QUARTER_15) leitet ihr abschließendes CNF-Ereignis an den CNF-Ausgang des umschließenden BOOLS_TO_QUARTERS-Blocks weiter. Zu diesem Zeitpunkt liegen alle 16 Quarter-Byte-Ausgänge (QB_00 bis QB_15) mit ihren neuen Werten vor.

Die Datenpfade sind parallel organisiert: Jeder boolesche Eingang I_xx ist direkt mit dem entsprechenden I-Eingang der internen Instanz verbunden, und jeder QB-Ausgang einer Instanz ist direkt mit dem entsprechenden QB_xx-Ausgang des Composite FB verbunden.

Technische Besonderheiten

- **Serielle Ausführung:** Die 16 Konvertierungen werden nacheinander, nicht parallel, ausgeführt. Dies führt zu einer definierten, aber nicht gleichzeitigen Aktualisierung der Ausgänge. Die Gesamtzykluszeit ist die Summe der Ausführungszeiten aller 16 internen Blöcke.
- **Konstanten-Nutzung:** Die Initialwerte und die konkreten Ausgabewerte der Konvertierung basieren auf vordefinierten Konstanten aus dem quarter-Namespace (z.B. quarter::COMMAND_DISABLE). Die genaue Semantik der möglichen Quarter-Byte-Zustände (z.B. COMMAND_DISABLE vs. COMMAND_ENABLE) muss der Dokumentation des zugrundeliegenden BOOL_TO_QUARTER-Blocks oder der quarter-Bibliothek entnommen werden.
- **Komposite Struktur:** Der Block dient primär der Bündelung und Vereinfachung der Verdrahtung in übergeordneten Applikationen. Die eigentliche Logik liegt in den eingebetteten BOOL_TO_QUARTER-Funktionsblöcken.

Zustandsübersicht

Als Composite FB ohne eigene, explizite Zustandsmaschine besitzt BOOLS_TO_QUARTERS keinen internen Zustand im engeren Sinne. Sein Verhalten wird vollständig durch die Kaskade der untergeordneten Blöcke und deren Zustände bestimmt. Der Block kann sich in einem von zwei makroskopischen Zuständen befinden:

1. **Idle:** Warten auf ein REQ-Ereignis. Alle Ausgänge behalten ihren letzten Wert.
2. **Verarbeitend (Processing):** Ein REQ-Ereignis läuft durch die Kaskade der 16 internen Blöcke. Während dieser Phase werden die Ausgänge nacheinander aktualisiert.

Anwendungsszenarien

- **Ansteuerung von kompakten Mehrwert-Aktoren:** Bei Aktoren oder Treibern, die Steuerbefehle nicht als einfache Ein-/Aus-Signale, sondern als 2-Bit-Kommandos (z.B. Ein/Aus/Fehlerreset/Notstop) erwarten.
- **Datenkomprimierung für Bus-Kommunikation:** Vor der Übertragung über Feldbusse, wo viele binäre Signale in ein platzsparendes Byte- oder Wort-orientiertes Protokoll gepackt werden müssen.
- **Schnittstelle zu Legacy-Systemen:** Als Adapter zwischen modernen IEC 61499-Steuerungen und älteren Systemen, die Daten in einem speziellen Quarter-Byte-Format erwarten oder liefern.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Gegenüber BOOL_TO_QUARTER:** BOOLS_TO_QUARTERS ist im Wesentlichen ein Array aus 16 BOOL_TO_QUARTER-Blöcken mit einer fest verdrahteten seriellen Ereigniskette. Während BOOL_TO_QUARTER eine einzelne Konvertierung durchführt, aggregiert BOOLS_TO_QUARTERS viele solcher Konvertierungen in einem wiederverwendbaren Baustein.
- **Gegenüber generischen Pack-Blöcken (z.B. BOOLx_TO_BYTE):** Blöcke wie BOOL8_TO_BYTE packen mehrere BOOL-Werte in die Bits eines einzelnen BYTEs. BOOLS_TO_QUARTERS hingegen erzeugt für

jeden Eingang ein eigenes (wenn auch nur teilweise genutztes) BYTE. Es findet keine Bit-Packung in ein gemeinsames Byte statt, sondern eine 1:1-Abbildung auf ein spezielles Kodierungsformat.

🛠️ Zugehörige Übungen

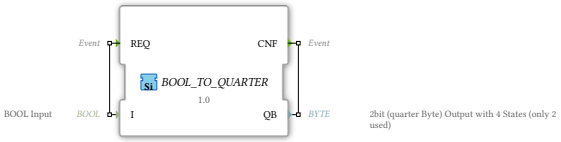
- Uebung_060

Fazit

Der `BOOLS_TO_QUARTERS`-Funktionsblock bietet eine praktische und vorkonfektionierte Lösung, um eine große Anzahl boolescher Signale seriell in das Quarter-Byte-Format zu konvertieren. Seine komposite Natur macht ihn einfach zu verstehen und zu verwenden, da er die Verdrahtung von 16 einzelnen Blöcken und ihrer Ereignislogik abstrahiert. Die serielle Abarbeitung ist ein entscheidendes Merkmal, das bei Echtzeitanforderungen berücksichtigt werden muss. Der Block ist ideal für spezifische Anwendungen, die das Quarter-Byte-Format erfordern, weniger jedoch für allgemeine Bit-Pack- oder -Unpack-Operationen.

BOOL_TO_QUARTER

🎧 Podcast



- QUARTER

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `BOOL_TO_QUARTER` dient der Konvertierung eines binären BOOL-Signals in ein spezielles, vordefiniertes Quartal-Byte (quarter byte). Er übersetzt die logischen Zustände `TRUE` und `FALSE` in entsprechende, semantisch bedeutungsvolle Byte-Konstanten, die typischerweise für Steuerbefehle (z. B. ENABLE/DISABLE) verwendet werden. Dieser Baustein ist Teil der `logiBUS::utils::quarter`-Bibliothek.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Dieses Ereignis löst die Konvertierungsfunktion aus. Bei seinem Eintreffen wird der aktuelle Wert am Daten-Eingang `I` gelesen und verarbeitet.

Ereignis-Ausgänge

- **CNF**: Dieses Ereignis signalisiert den Abschluss der Konvertierung. Es wird zusammen mit dem berechneten Ausgangswert `QB` ausgegeben.

Daten-Eingänge

- **I** (BOOL, Initialwert: `FALSE`): Der binäre Eingangswert, der konvertiert werden soll.

Daten-Ausgänge

- **QB** (BYTE, Initialwert: `quarter::COMMAND_DISABLE`): Der Ausgangswert als Byte. Der Block nutzt ein sogenanntes „Quarter Byte“ (2 Bit), das vier Zustände repräsentieren kann, von denen in dieser Implementierung nur zwei genutzt werden. Der konkrete Wert wird aus den Konstanten der importierten Bibliothek abgeleitet.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei Auslösung durch das `REQ`-Ereignis wertet der Block den Wert am Eingang `I` aus. Die Verarbeitung erfolgt über eine `CASE`-Anweisung:

- Wenn `I` den Wert `BOOL#TRUE` hat, wird der Ausgang `QB` auf die Konstante `quarter::COMMAND_ENABLE` gesetzt.
- In allen anderen Fällen (standardmäßig also bei `FALSE`) wird der Ausgang `QB` auf die Konstante `quarter::COMMAND_DISABLE` gesetzt.

Unmittelbar nach der Zuweisung wird das `CNF`-Ereignis zusammen mit dem neuen Wert von `QB` ausgegeben.

Technische Besonderheiten

- **Bibliotheksabhängigkeit**: Der Block importiert und nutzt Konstanten (`quarter::COMMAND_ENABLE`, `quarter::COMMAND_DISABLE`) aus dem Paket `logiBUS::utils::quarter::const::quarter`. Die tatsächlichen Byte-Werte sind somit dort definiert und zentral änderbar.
- **Quarter Byte Konzept**: Der Ausgang ist als `BYTE` deklariert, nutzt konzeptionell aber nur die unteren 2 Bit (ein „Quarter Byte“). Dies ermöglicht eine spätere Erweiterung um weitere Zustände, ohne die Schnittstelle ändern zu müssen.
- **Initialwerte**: Sowohl der Eingang `I` als auch der Ausgang `QB` haben definierte Initialwerte (`FALSE` bzw. `COMMAND_DISABLE`), was ein deterministisches Startverhalten gewährleistet.

Zustandsübersicht

Der Block besitzt keinen internen Zustand im Sinne eines Speichers. Er verhält sich rein kombinatorisch in Bezug auf die Eingangsdaten bei Auftreten des `REQ`-Ereignisses. Der Ausgang `QB` behält seinen zuletzt zugewiesenen Wert bis zum nächsten Ausführungszyklus.

Anwendungsszenarien

- **Steuerung von Aktoren**: Umwandlung eines einfachen Freigabe-Signals (`BOOL`) in einen gerätespezifischen Enable/Disable-Befehl, der als Byte über ein Feldbusprotokoll versendet wird.
- **Schnittstellenanpassung**: Anbindung von Logikbausteinen, die mit BOOL-Werten arbeiten, an Steuerungskomponenten, die Befehle im Quarter-Byte-Format erwarten.
- **Code-Vereinheitlichung**: Sicherstellung, dass in einem Projekt immer die gleichen, zentral definierten Konstanten für Enable-/Disable-Befehle verwendet werden.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`BOOL_TO_BYTE`**: Ein generischer Konverter, der `TRUE` typischerweise auf `1` und `FALSE` auf `0` abbildet. `BOOL_TO_QUARTER` ist spezialisierter und verwendet projektspezifische, semantische Konstanten anstelle numerischer Werte.
- **Direkte Zuweisung**: Die Funktionalität könnte auch durch direkte Zuweisung in ST-Code (`QB := I`) nachgebildet werden, dabei gingen jedoch die Vorteile der zentralen Konstantendefinition und der klaren Schnittstelle (Events) verloren.

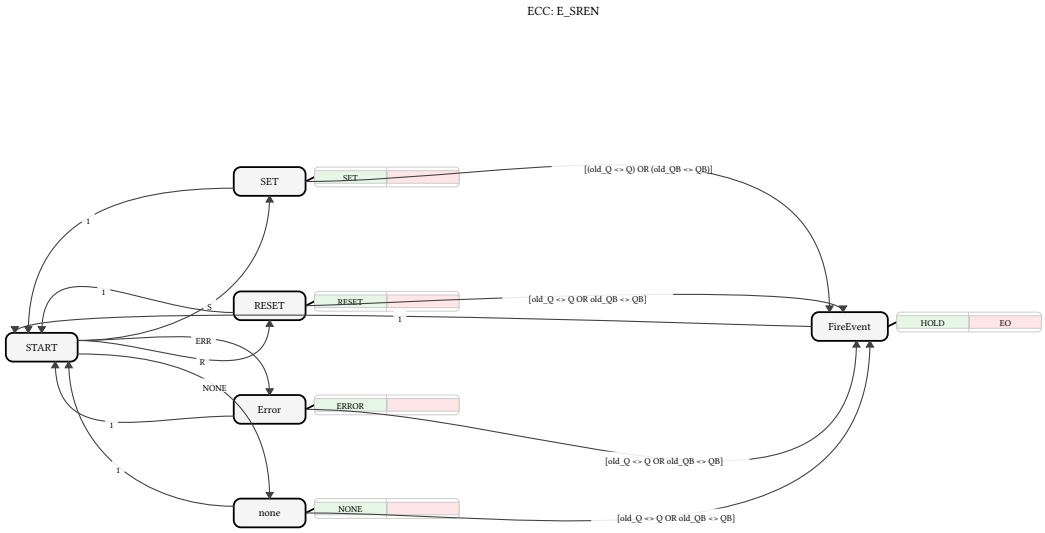
🛠️ Zugehörige Übungen

- Uebung_055
- Uebung_056

Fazit

Der `BOOL_TO_QUARTER`-Funktionsblock ist ein spezialisierter, aber nützlicher Konverter für Steueranwendungen. Er bietet eine saubere Schnittstelle mit Ereignissteuerung, verwendet zentral verwaltete Konstanten für eine einheitliche Semantik und ist durch das Quarter-Byte-Konzept potentiell erweiterbar. Seine Stärke liegt in der Verbindung von einfacher binärer Logik mit gerätespezifischen Steuerprotokollen.

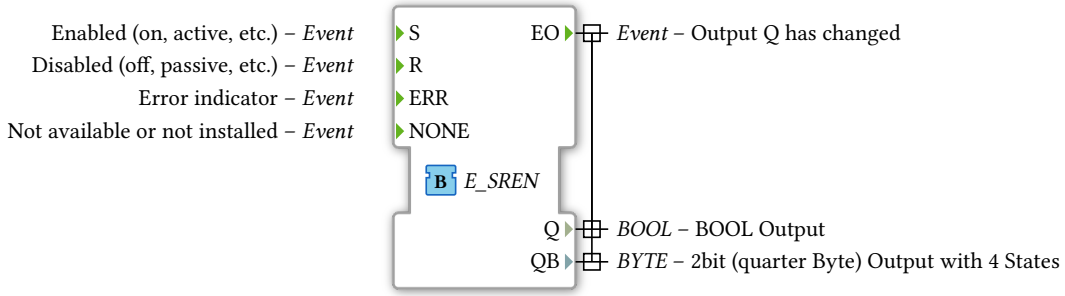
E_SREN



• – ○ • – ○ • – ○ •

Einleitung

Der E_SREN (Event-driven quad-state) ist ein ereignisgesteuerter Funktionsblock, der einen Ausgang mit vier definierten Zuständen steuert. Er reagiert auf verschiedene Eingangsereignisse und setzt seinen Ausgang entsprechend auf einen der vier Zustände: Aktiviert (`SET`), Deaktiviert (`RESET`), Fehler (`ERROR`) oder Keine Aktion (`NONE`). Der Baustein ist besonders für Anwendungen geeignet, in denen ein Signal mehrere Betriebszustände repräsentieren muss, wie z.B. in Überwachungs- oder Steuerungssystemen.



Schnittstellenstruktur

Ereignis-Eingänge

- **S (Set):** Setzt den Ausgang `Q` auf `TRUE` und `QB` auf den Zustand `COMMAND_ENABLE`.
- **R (Reset):** Setzt den Ausgang `Q` auf `FALSE` und `QB` auf den Zustand `COMMAND_DISABLE`.
- **ERR (Error):** Setzt den Ausgang `Q` auf `FALSE` und `QB` auf den Zustand `STATUS_ERROR`.
- **NONE:** Setzt den Ausgang `QB` auf den Zustand `COMMAND_NO_ACTION`. Der Ausgang `Q` bleibt unverändert.

Ereignis-Ausgänge

- **EO (Event Output):** Wird ausgelöst, wenn sich einer der Ausgangswerte (`Q` oder `QB`) geändert hat. Dieses Ereignis wird immer zusammen mit den aktuellen Werten der Datenausgänge `Q` und `QB` gesendet.

Daten-Eingänge

- Dieser Funktionsblock besitzt keine Daten-Eingänge.

Daten-Ausgänge

- **Q (BOOL):** Einfacher boolescher Ausgang. Er ist nur `TRUE`, wenn das `S`-Ereignis eintrifft. Er ist `FALSE` bei den Ereignissen `R` und `ERR` und bleibt unverändert bei `NONE`.
- **QB (BYTE):** Ein Byte-Ausgang, der speziell zur Kodierung von vier Zuständen (2 Bit) dient. Die konkreten Werte (z.B. `COMMAND_ENABLE`) werden aus der importierten Bibliothek `quarter::const::quarter` bezogen.

Adapter

- Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der E_SREN ist als Basic-FB implementiert und besitzt einen internen Zustandsautomaten (ECC). Der Ausgangszustand ist `START`. Bei Eintreffen eines Eingangsereignisses (`S`, `R`, `ERR`, `NONE`) wechselt der Automat in den entsprechenden Zustand (`SET`, `RESET`, `Error`, `none`). In diesen Zuständen wird der zugehörige Algorithmus ausgeführt, der die Werte für `Q` und `QB` berechnet.

Anschließend prüft eine Bedingung, ob sich der neue Wert von `Q` oder `QB` gegenüber dem zuletzt gespeicherten Wert (`old_Q`, `old_QB`) geändert hat. Falls ja, wechselt der Automat in den Zustand `FireEvent`. Hier wird der Algorithmus `HOLD` ausgeführt, der die aktuellen Ausgangswerte in den internen Variablen speichert, und gleichzeitig das Ausgangsereignis `EO` ausgelöst. Danach kehrt der Automat immer in den `START`-Zustand zurück. Falls sich die Ausgangswerte nicht geändert haben, erfolgt ein direkter Übergang von den Zuständen `SET`, `RESET`, `Error` oder `none` zurück in den `START`-Zustand, ohne dass `EO` ausgelöst wird.

Technische Besonderheiten

- **Zustandserkennung:** Der Baustein merkt sich den vorherigen Ausgangszustand in den internen Variablen `old_Q` und `old_QB`. Das Ausgangsereignis `EO` wird nur bei einer tatsächlichen Zustandsänderung generiert, was redundante Ereignisausgaben verhindert.
- **Bibliotheksabhängigkeit:** Die spezifischen Byte-Werte für den `QB`-Ausgang werden aus der Konstanten-Bibliothek `logiBUS::utils::quarter::const::quarter` importiert. Die korrekte Funktion setzt die Verfügbarkeit dieser Bibliothek voraus.
- **Vier-Zustands-Logik:** Die Logik des `NONE`-Ereignisses ist speziell: Während `QB` auf `COMMAND_NO_ACTION` gesetzt wird, bleibt der boolesche Ausgang `Q` explizit unverändert.

Zustandsübersicht

Der ECC (Execution Control Chart) besteht aus sechs Zuständen:

1. **START:** Initialer und Ruhezustand.
2. **SET:** Wird bei Ereignis `S` aktiviert. Setzt `Q=TRUE`, `QB=COMMAND_ENABLE`.
3. **RESET:** Wird bei Ereignis `R` aktiviert. Setzt `Q=FALSE`, `QB=COMMAND_DISABLE`.
4. **Error:** Wird bei Ereignis `ERR` aktiviert. Setzt `Q=FALSE`, `QB=STATUS_ERROR`.
5. **none:** Wird bei Ereignis `NONE` aktiviert. Setzt `QB=COMMAND_NO_ACTION`. `Q` bleibt unverändert.
6. **FireEvent:** Wird nur betreten, wenn sich `Q` oder `QB` geändert haben. Speichert die neuen Werte und löst `EO` aus.

Anwendungsszenarien

- **Aktorsteuerung:** Steuerung eines Antriebs mit den Befehlen „Einschalten“ (`S`), „Ausschalten“ (`R`), „Störung“ (`ERR`) und „Manuell/Extern gesteuert“ (`NONE`).
- **Meldesysteme:** Darstellung des Status einer Maschine: „In Betrieb“, „Stopp“, „Fehler“, „Wartung“.
- **Sicherheitsrelevante Steuerungen:** Klare Trennung von normalen Betriebs- (`S`/`R`), Fehler- (`ERR`) und Wartungs-/Override-Zuständen (`NONE`).

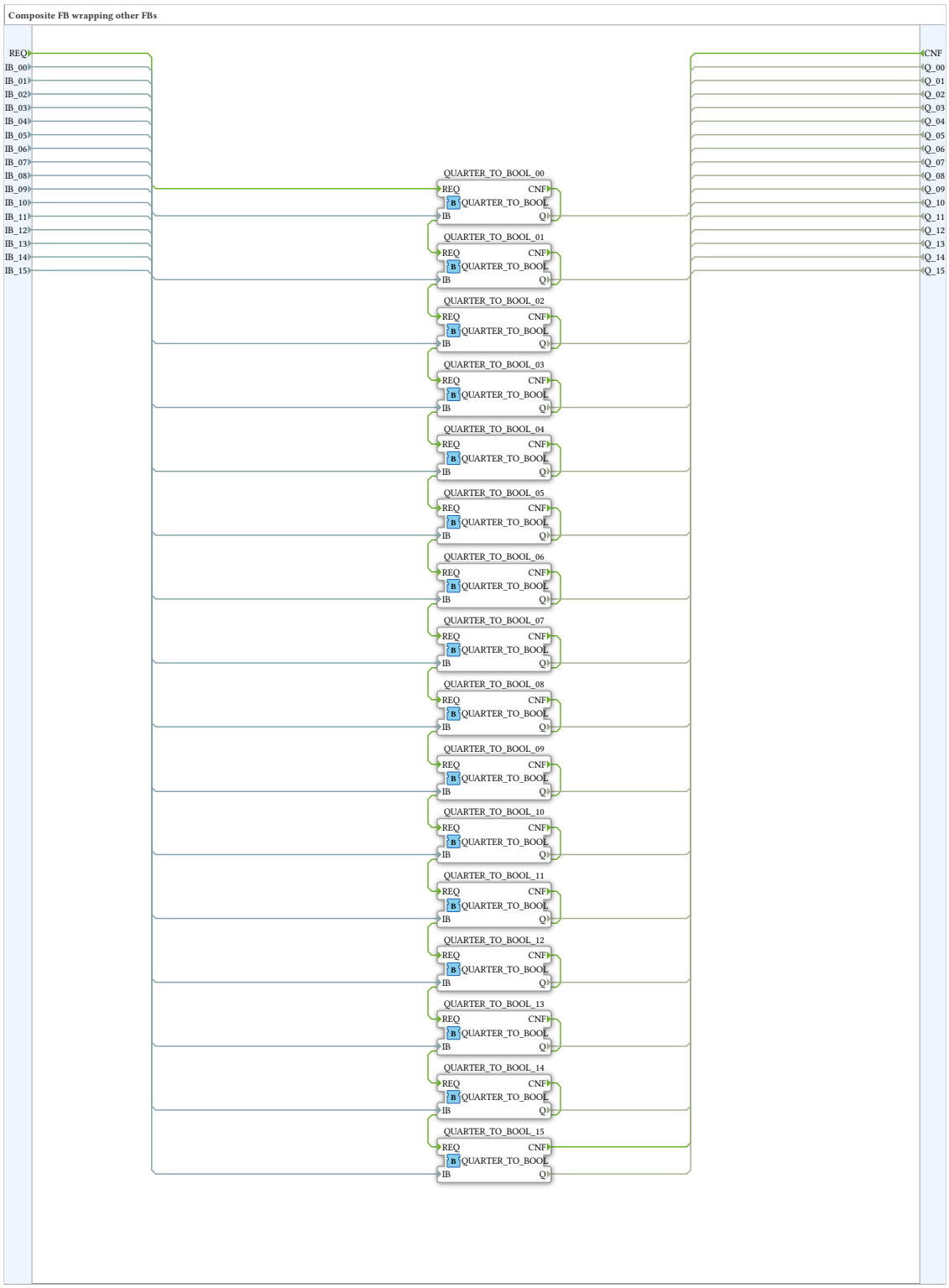
⚖️ Vergleich mit ähnlichen Bausteinen

- **E_SR (Bistabile Funktion):** Der klassische Set-Reset-Flipflop kennt nur zwei stabile Zustände (`TRUE`/`FALSE`). Der E_SREN erweitert dieses Konzept um zwei weitere Zustände (`ERROR`, `NO_ACTION`), die über ein dediziertes Byte (`QB`) kodiert werden.
- **E_D_FF (D-Flipflop):** Übernimmt einen Datenwert bei einem Takt-Ereignis. Der E_SREN ist ereignisgesteuert (vier verschiedene Ereignisse) und hat keinen separaten Dateneingang. Die „Daten“ sind in den auslösenden Ereignissen implizit enthalten.

Fazit

Der E_SREN ist ein spezialisierter, ereignisgesteuerter Funktionsblock für Anwendungen, die mehr als zwei diskrete Zustände benötigen. Durch die Kombination eines einfachen booleschen Signals (`Q`) mit einem mehrwertigen Byte-Signal (`QB`) und der intelligenten, änderungsbasierten Ereignisausgabe (`EO`) bietet er eine effiziente und übersichtliche Lösung für komplexe Zustandssteuerungen. Seine Stärke liegt in der klaren Semantik der vier Eingangsereignisse und der zuverlässigen Vermeidung von redundanten Ausgabeereignissen.

QUARTERS_TO_BOOLS



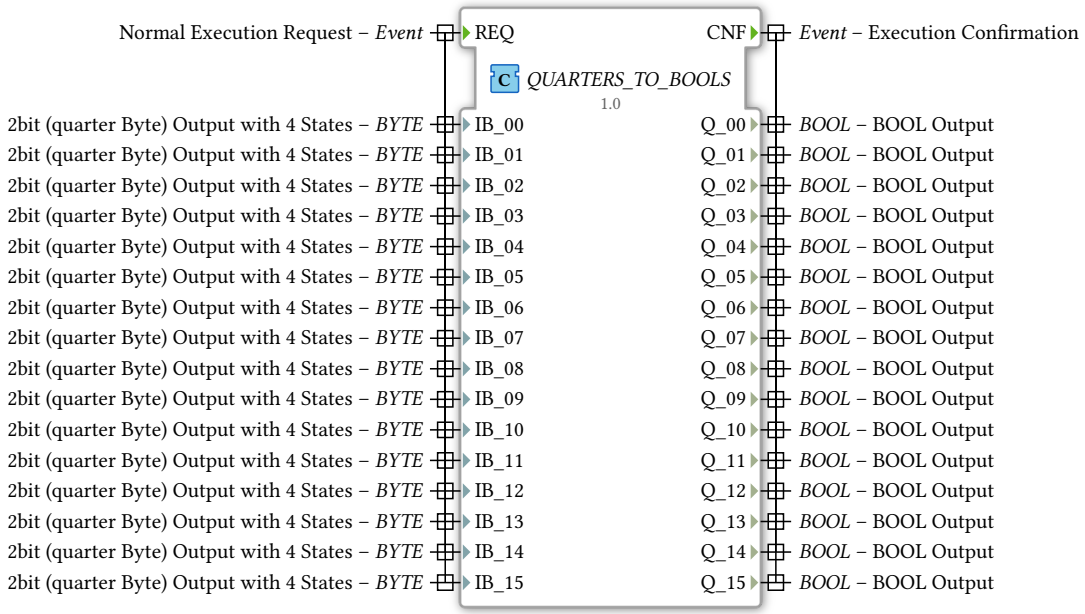
🎧 Podcast

- QUARTER

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `QUARTERS_TO_BOOLS` ist ein Composite Function Block, der 16 separate 2-Bit-Eingabewerte (sogenannte „Quarter Bytes“) parallel in entsprechende boolesche Ausgangssignale umwandelt. Er dient als Wrapper und vereinfacht die Handhabung, indem er eine Vielzahl einzelner Konvertierungsbausteine zu einem einzigen, übersichtlichen Baustein zusammenfasst. Dieser Block ist besonders nützlich in Steuerungssystemen, in denen kompakte Datenformate (wie 2-Bit-Zustände in einem Byte) in einfache binäre Steuersignale für Aktoren oder Statusanzeigen umgesetzt werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Verarbeitung. Bei Eintreffen dieses Ereignisses werden alle 16 Eingabewerte (`IB_00` bis `IB_15`) ausgelesen und verarbeitet.

Ereignis-Ausgänge

- **CNF**: Signalisiert die erfolgreiche Beendigung der Verarbeitung. Dieses Ereignis wird ausgegeben, nachdem alle 16 internen Konvertierungen abgeschlossen und die Ausgangsdaten (`Q_00` bis `Q_15`) aktualisiert sind.

Daten-Eingänge

- **IB_00** bis **IB_15** (Typ: `BYTE`): 16 Eingänge für 2-Bit-Daten (Quarter Bytes). Jeder Eingang kann einen von vier definierten Zuständen repräsentieren (z.B. `quarter::COMMAND_DISABLE`). Der Standard-Initialwert für alle Eingänge ist `quarter::COMMAND_DISABLE`.

Daten-Ausgänge

- **Q_00** bis **Q_15** (Typ: `BOOL`): 16 boolesche Ausgänge, die den konvertierten Zustand des jeweiligen Quarter-Byte-Eingangs widerspiegeln. Der Initialwert aller Ausgänge ist `FALSE`.

Adapter

Dieser Funktionsblock verwendet keine Adapter-Schnittstellen.

Funktionsweise

QUARTERS_TO_BOOLS ist ein Composite FB, der intern aus 16 Instanzen des Basisfunktionsblocks QUARTER_TO_BOOL besteht. Jede dieser Instanzen ist für die Konvertierung eines spezifischen Eingangs (IB_xx) zuständig.

1. **Auslösung**: Das Eintreffen des Ereignisses REQ löst die Verarbeitungskette aus.
2. **Parallele Datenverteilung**: Die an den Eingängen IB_00 bis IB_15 anliegenden Byte-Werte werden parallel an die entsprechenden Eingänge (IB) der 16 internen QUARTER_TO_BOOL-Blöcke weitergeleitet.
3. **Serielle Ereignisverarbeitung**: Das REQ-Ereignis wird seriell durch die Kette der internen Blöcke propagiert. Es startet bei QUARTER_TO_BOOL_00 und wird nacheinander von einem Block zum nächsten weitergereicht (CNF -> REQ), bis der letzte Block (QUARTER_TO_BOOL_15) seine Verarbeitung abschließt.
4. **Konvertierung**: Jeder interne QUARTER_TO_BOOL-Block interpretiert seinen Byte-Eingangswert gemäß einer definierten Logik (vermutlich basierend auf den zwei niederwertigsten Bits) und setzt ihn auf einen booleschen Wahrheitswert (TRUE oder FALSE) um.
5. **Ergebnisausgabe**: Die booleschen Ergebnisse (Q) der internen Blöcke werden auf die entsprechenden Ausgänge Q_00 bis Q_15 des Composite-Blocks geschaltet.
6. **Abschluss**: Sobald der letzte interne Block fertig ist, löst der Composite-Block sein CNF-Ereignis aus, um den Abschluss des gesamten Konvertierungszyklus für alle 16 Kanäle zu signalisieren.

Technische Besonderheiten

- **Initialisierung**: Alle Eingänge sind mit dem spezifischen Wert quarter::COMMAND_DISABLE vorbelegt, und alle Ausgänge starten mit FALSE. Dies gewährleistet einen definierten, inaktiven Ausgangszustand.
- **Verarbeitungsreihenfolge**: Während die Dateneingänge parallel verteilt werden, erfolgt die Ereignisverarbeitung streng sequenziell von Index 00 bis 15. Dies führt zu einer deterministischen, wenn auch nicht gleichzeitigen, Aktualisierung der Ausgänge.
- **Komposit-Bauweise**: Der Block kapselt die Komplexität von 16 einzelnen Konvertierungen und bietet eine saubere, vereinheitlichte Schnittstelle, was die Wiederverwendbarkeit und Lesbarkeit in übergeordneten Applikationen erhöht.

Zustandsübersicht

Als Composite Function Block besitzt QUARTERS_TO_BOOLS keinen eigenen komplexen Zustandsautomaten im herkömmlichen Sinne. Sein Verhalten wird durch das Netzwerk der internen Blöcke definiert. Vereinfacht kann der Gesamtzustand als **Idle** (wartet auf REQ) und **Processing** (Ereignis läuft durch die interne Kette) betrachtet werden. Der CNF-Ausgang markiert den Übergang zurück in den Idle-Zustand.

Anwendungsszenarien

- **Kompakte SPS-Anbindung**: Umwandlung von kompakten 32-Bit-Datenwörtern (die 16 2-Bit-Zustände enthalten) in 16 einzelne binäre Steuersignale für Ventile, Lampen oder Relais.
- **Statusdekodierung**: Dekodierung von Gerätestatusinformationen, die in einem „Quarter Byte"-Format übertragen werden, in einzelne, leicht verarbeitbare Fehler- oder Betriebszustandsbits.
- **Vereinfachung von Funktionsplanern**: Ersetzt 16 separate QUARTER_TO_BOOL-Blöcke und deren Verdrahtung in einem Funktionsplan durch einen einzigen, übersichtlichen Block, was die Projektwartung erleichtert.

⚖️ Vergleich mit ähnlichen Bausteinen

- **QUARTER_TO_BOOL**: Dies ist der elementare Baustein, der einen einzelnen 2-Bit-Eingang konvertiert. QUARTERS_TO_BOOLS bündelt 16 Instanzen dieses Blocks zu einer Einheit. Die Verwendung des Composite-Blocks ist effizienter für die Handhabung mehrerer Kanäle, während der Einzelblock maximale Flexibilität bei der individuellen Platzierung und Verdrahtung bietet.
- **BYTE_TO_BOOL-Blöcke**: Konventionelle Blöcke, die ein ganzes Byte in 8 einzelne Bits aufspalten. QUARTERS_TO_BOOLS ist spezialisierter, da er davon ausgeht, dass jedes Byte bereits in vier unabhängige 2-Bit-Einheiten unterteilt ist, die separat interpretiert werden müssen.

🛠️ Zugehörige Übungen

- Uebung_060

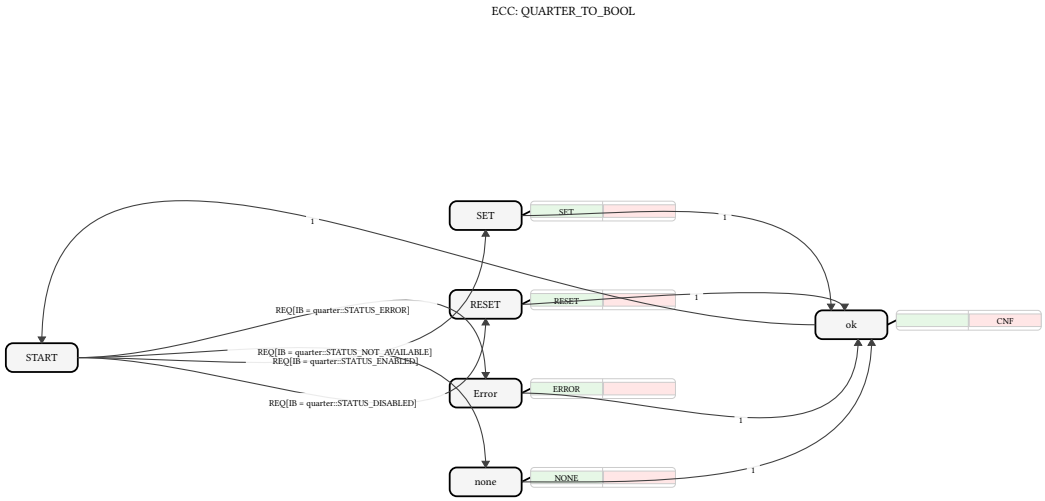
Fazit

Der QUARTERS_TO_BOOLS-Funktionsblock ist ein praktisches und zeitsparendes Hilfsmittel für die IEC 61499-Programmierung, wenn häufig mehrere 2-Bit-Datenkanäle in boolesche Signale umgewandelt

werden müssen. Durch die Kapselung von 16 Konvertierungen in einem Block reduziert er den Verdrahtungsaufwand in übergeordneten Applikationen erheblich und verbessert die Übersichtlichkeit. Seine deterministische, serielle Ereignisabarbeitung gewährleistet ein zuverlässiges Verhalten, das für Steuerungsaufgaben gut geeignet ist.

QUARTER_TO_BOOL

🎧 Podcast

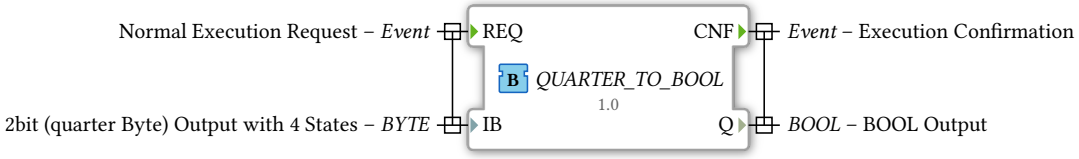


- QUARTER

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `QUARTER_TO_BOOL` dient der Konvertierung eines 4-Zustands-Signals (Quad-State), das in den unteren zwei Bits eines BYTE-Wertes kodiert ist, in ein einfaches BOOL-Signal. Er ist besonders für die Anbindung an Systeme nützlich, die Statusinformationen mit mehr als zwei Zuständen liefern, die jedoch in einer einfachen binären Logik weiterverarbeitet werden sollen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Mit diesem Ereignis wird der anliegende Eingangswert `IB` ausgewertet.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieses Ereignis wird immer nach der Bearbeitung eines `REQ`-Ereignisses ausgegeben.

Daten-Eingänge

- **IB** (BYTE): Enthält den zu konvertierenden 4-Zustands-Wert in seinen unteren zwei Bits. Der Initialwert ist `quarter::COMMAND_DISABLE`. Die erwarteten Werte sind spezifische Konstanten aus dem `quarter`-Namespace.

Daten-Ausgänge

- **Q** (BOOL): Das Ergebnis der Konvertierung. Der Initialwert ist `FALSE`.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Auslösung durch ein `REQ`-Ereignis liest der Baustein den Wert am Daten-Eingang `IB`. Anschließend wird dieser Wert mit vordefinierten Konstanten verglichen, um den entsprechenden internen Zustand zu ermitteln. Je nach Zustand wird ein spezifischer Algorithmus ausgeführt, der den BOOL-Ausgang `Q` setzt oder seinen Wert beibehält. Nach Ausführung des Algorithmus wird das `CNF`-Ereignis ausgelöst, und der Baustein kehrt in seinen Startzustand zurück.

Die konkrete Abbildung der Eingangswerte auf die Ausgangslogik ist wie folgt:

- `quarter::STATUS_ENABLED` ⟶ Algorithmus `SET` ⟶ `Q := TRUE`
- `quarter::STATUS_DISABLED` ⟶ Algorithmus `RESET` ⟶ `Q := FALSE`
- `quarter::STATUS_ERROR` ⟶ Algorithmus `ERROR` ⟶ `Q := FALSE`
- `quarter::STATUS_NOT_AVAILABLE` ⟶ Algorithmus `NONE` ⟶ `Q` bleibt unverändert

Technische Besonderheiten

- **Zustandsbehandlung**: Der Baustein ist als Basic-FB implementiert und besitzt eine explizite Zustandsmaschine (ECC). Die Zustände `SET`, `RESET`, `ERROR` und `none` sind reine Algorithmuszustände, während der Zustand `ok` für die Ausgabe des Bestätigungsereignisses verantwortlich ist.
- **Wertbeibehaltung**: Im Fall des Status `STATUS_NOT_AVAILABLE` wird der Algorithmus `NONE` ausgeführt, der den aktuellen Wert des Ausgangs `Q` explizit nicht verändert. Dies ermöglicht ein „Don't Care"- oder „Hold"-Verhalten.
- **Initialisierung**: Der Datenausgang `Q` wird beim Start auf `FALSE` initialisiert.

Zustandsübersicht

Die ECC (Execution Control Chart) besteht aus sechs Zuständen:

1. **START**: Initial- und Wartezustand. Bei eintreffendem `REQ` erfolgt je nach Wert von `IB` ein Übergang zu `SET`, `RESET`, `Error` oder `none`.
2. **SET**: Führt den Algorithmus `SET` aus (setzt `Q` auf TRUE).
3. **RESET**: Führt den Algorithmus `RESET` aus (setzt `Q` auf FALSE).
4. **Error**: Führt den Algorithmus `ERROR` aus (setzt `Q` auf FALSE).
5. **none**: Führt den Algorithmus `NONE` aus (lässt `Q` unverändert).
6. **ok**: Löst das `CNF`-Ausgangsereignis aus und wechselt anschließend zurück in den `START`-Zustand.

Anwendungsszenarien

- **Anbindung von Feldgeräten**: Viele Sensoren oder Aktoren melden Status wie „Bereit", „Gestört", „Wartung" oder „Nicht verbunden". Dieser Baustein kann solche Meldungen in ein einfaches „Ein/Aus"- oder „OK/Nicht OK"-Signal für übergeordnete Steuerlogiken umwandeln.
- **Vereinfachung von Logik**: In Steuerungen, die nur binäre Entscheidungen benötigen (z.B. „Prozess freigeben" ja/nein), kann dieser FB komplexere Statusmeldungen auf die entscheidende binäre Information reduzieren.
- **Fehlerbehandlung**: Ein einheitlicher Umgang mit Fehlerzuständen (`STATUS_ERROR`) durch Setzen des Ausgangs auf `FALSE`.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Standardkonverter (z.B., `BYTE_TO_BOOL`)**: Ein einfacher `BYTE_TO_BOOL`-Konverter würde typischerweise einen Schwellwert verwenden (z.B., alles >0 wird zu TRUE). `QUARTER_TO_BOOL` hingegen interpretiert spezifische, benannte Zustände und bietet ein definiertes Verhalten für jeden einzelnen, einschließlich der Möglichkeit, den Ausgang bei einem Zustand unverändert zu lassen.

- **`E_SELECT`- oder `E_DEMUX`-Bausteine**: Diese könnten verwendet werden, um basierend auf einem Eingangswert unterschiedliche Ereignispfade zu aktivieren. `QUARTER_TO_BOOL` kapselt diese Logik speziell für die Konvertierung von 4-Zustands-Signalen und liefert direkt das boolesche Ergebnis.

🛠️ Zugehörige Übungen

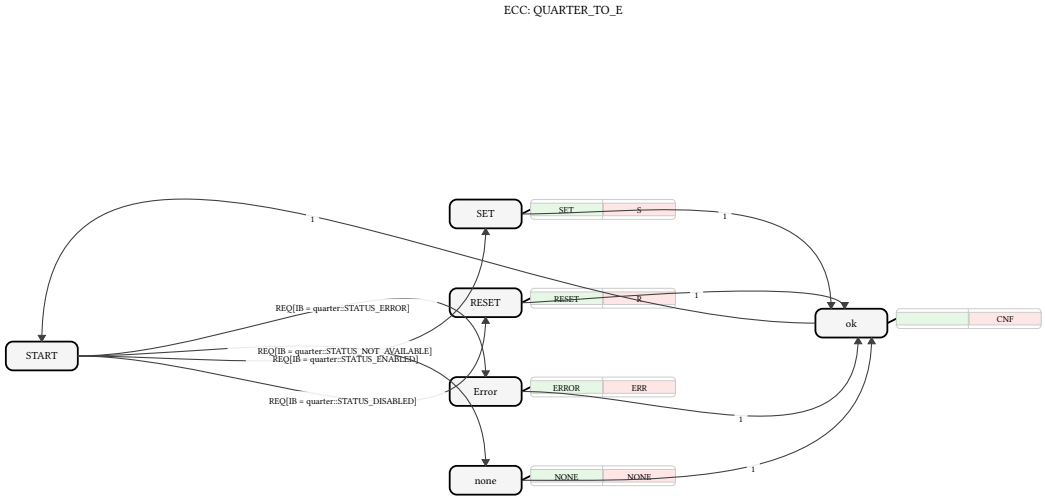
- Uebung_055
- Uebung_056

Fazit

Der `QUARTER_TO_BOOL`-Funktionsblock ist ein spezialisierter und nützlicher Konverter für Anwendungen, in denen kompakte Statusinformationen mit vier diskreten Zuständen in eine einfache binäre Logik integriert werden müssen. Seine klare Definition des Verhaltens für jeden Zustand, insbesondere die Beibehaltung des Ausgangswerts im „Nicht verfügbar“-Fall, macht ihn robust und gut vorhersagbar. Er eignet sich ideal für die Schnittstelle zwischen komplexeren Feldbussystemen und grundlegender binärer Steuerungslogik.

QUARTER_TO_E

🎧 Podcast

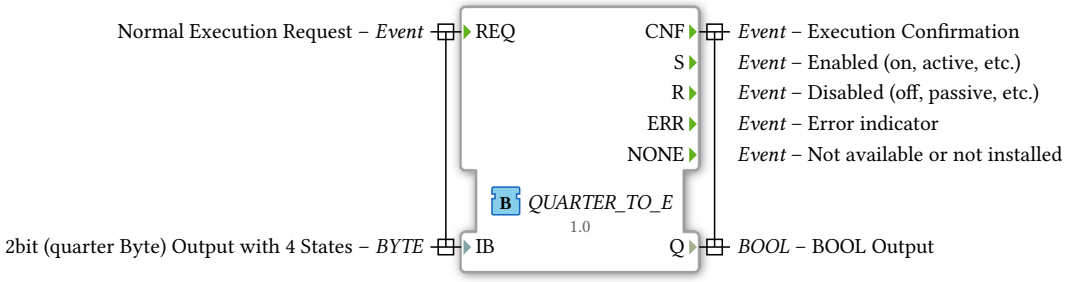


- QUARTER

• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock `QUARTER_TO_E` dient dazu, einen 2-Bit-Zustandswert (ein sogenanntes „Quarter Byte“) in vier verschiedene, eindeutige Ereignisse zu übersetzen. Er ist ein Hilfsbaustein, der häufig in Verbindung mit Bausteinen verwendet wird, die mehr als die üblichen zwei Zustände (EIN/AUS) melden, wie z.B. Statusmeldungen von Geräten, die auch Fehler- oder Nicht-Verfügbarkeitszustände anzeigen können.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Verarbeitung. Bei diesem Ereignis wird der aktuelle Wert am Daten-Eingang `IB` ausgewertet.

Ereignis-Ausgänge

- **CNF**: Signalisiert die erfolgreiche Beendigung der Verarbeitung, unabhängig vom erkannten Zustand. Wird immer nach der Ausgabe eines der spezifischen Ereignisse generiert.
- **S** (Enabled): Wird ausgelöst, wenn der Zustand „aktiviert“ oder „eingeschaltet“ erkannt wird.
- **R** (Disabled): Wird ausgelöst, wenn der Zustand „deaktiviert“ oder „ausgeschaltet“ erkannt wird.
- **ERR** (Error): Wird ausgelöst, wenn ein Fehlerzustand erkannt wird.
- **NONE** (Not available): Wird ausgelöst, wenn der Zustand „nicht verfügbar“ oder „nicht installiert“ erkannt wird.

Daten-Eingänge

- **IB** (BYTE): Enthält den zu interpretierenden 2-Bit-Zustandswert. Der Initialwert ist auf `quarter::COMMAND_DISABLE` gesetzt. Die tatsächliche Interpretation erfolgt anhand der definierten Konstanten aus dem `quarter`-Paket.

Daten-Ausgänge

- **Q** (BOOL): Ein boolescher Ausgang, dessen Wert je nach erkanntem Zustand gesetzt wird. Er dient als einfache, binäre Repräsentation des Hauptzustands (aktiviert/deaktiviert).

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der `QUARTER_TO_E` ist ein Basic Function Block mit einem internen Zustandsautomaten (ECC). Bei Eintreffen des `REQ`-Ereignisses wird der Wert am Eingang `IB` mit vordefinierten Konstanten verglichen. Abhängig vom Vergleich springt der Automat in einen der vier Zustände `SET`, `RESET`, `Error` oder `none`. In jedem dieser Zustände wird ein entsprechender Algorithmus ausgeführt, der den booleschen Ausgang `Q` setzt (oder nicht ändert) und das zugehörige spezifische Ereignis (`S`, `R`, `ERR`, `NONE`) auslöst. Anschließend wechselt der Automat in den Zustand `ok`, von wo aus das Bestätigungsereignis `CNF` ausgegeben wird, bevor der Automat in den Ausgangszustand `START` zurückkehrt und auf das nächste `REQ` wartet.

Technische Besonderheiten

- Der Block nutzt Konstanten aus dem Paket `logiBUS::utils::quarter::const::quarter` (`STATUS_ENABLED`, `STATUS_DISABLED`, `STATUS_ERROR`, `STATUS_NOT_AVAILABLE`). Diese müssen im Projekt verfügbar sein.
- Der Ausgang `Q` wird nur in den Zuständen `SET` (auf TRUE), `RESET` (auf FALSE) und `ERROR` (auf FALSE) verändert. Im Zustand `NONE` wird `Q` explizit nicht verändert („Don’t care“).
- Die Ausgabe des `CNF`-Ereignisses erfolgt immer, unabhängig davon, welcher spezifische Zustand erkannt wurde. Dies ermöglicht eine einheitliche Ablaufsteuerung in der Applikation.

Zustandsübersicht

1. **START**: Wartezustand. Bei `REQ` erfolgt die Auswertung von `IB` und der Übergang zu einem Verarbeitungszustand.

2. **SET**: Wird bei `IB = STATUS_ENABLED` erreicht. Setzt `Q:=TRUE` und löst `S` aus.
3. **RESET**: Wird bei `IB = STATUS_DISABLED` erreicht. Setzt `Q:=FALSE` und löst `R` aus.
4. **Error**: Wird bei `IB = STATUS_ERROR` erreicht. Setzt `Q:=FALSE` und löst `ERR` aus.
5. **none**: Wird bei `IB = STATUS_NOT_AVAILABLE` erreicht. Löst `NONE` aus (Q bleibt unverändert).
6. **ok**: Zwischenzustand, der das `CNF`-Ereignis auslöst und zurück zum `START`-Zustand führt.

Anwendungsszenarien

- **Interpretation von Gerätestatus**: Ein Antriebsmeldet seinen Status nicht nur als EIN/AUS, sondern auch als „Fehler“ oder „Nicht bereit“. Dieser Block wandelt den Gerätestatus in separate, leicht verarbeitbare Ereignisse um.
- **Vereinfachung der Logik**: Anstatt in nachfolgenden Blöcken ständig den Byte-Wert abfragen zu müssen, können direkt auf die spezifischen Ereignisse (`S`, `R`, `ERR`) reagiert werden.
- **Abstraktion**: Dient als Adapter zwischen Bausteinen, die einen Quarter-Byte-Status liefern, und einer Logik, die mit klassischen Ereignis-/Boolsignalen arbeitet.

⚖️ Vergleich mit ähnlichen Bausteinen

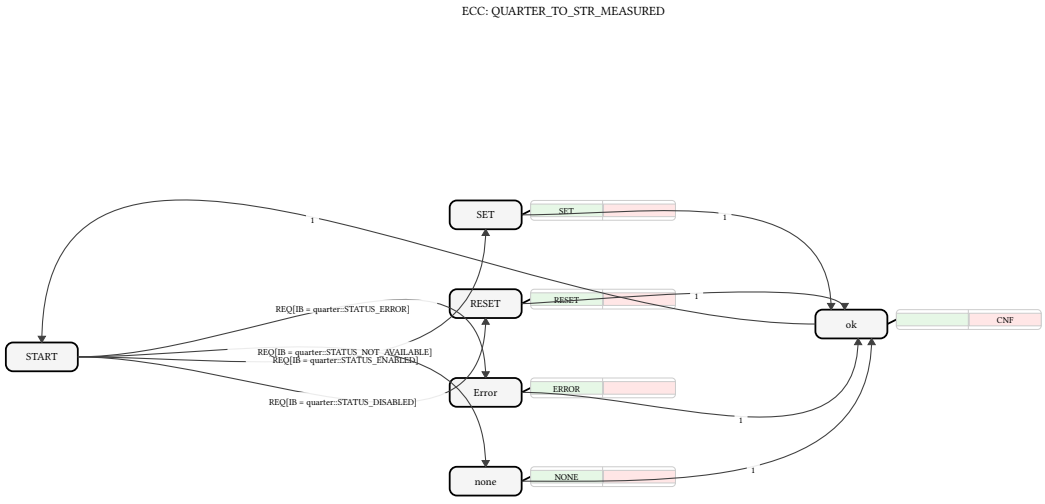
- **E_DEMUX / E_SELECT**: Diese Blöcke leiten ein Eingangsereignis basierend auf einem Steuerwert an einen von mehreren Ausgängen weiter. `QUARTER_TO_E` ist spezialisierter: Er übersetzt einen spezifischen Datenwert (`IB`) nicht nur in eine Auswahl, sondern löst auch unterschiedliche *inhaltliche* Ereignisse aus und berechnet zusätzlich einen booleschen Wert (`Q`).
- **BYTE_TO_E**: Ein generischer Block, der jedes Bit eines Bytes in ein separates Ereignis umwandeln könnte. `QUARTER_TO_E` ist semantisch reicher, da er spezifische, vordefinierte Zustände (Enabled, Disabled, Error, None) und deren Bedeutung kennt und ausgibt.

Fazit

Der `QUARTER_TO_E` ist ein nützlicher und spezialisierter Baustein für die strukturierte Verarbeitung von 4-Zustands-Signalen (Quad-State). Er entlastet die Applikationslogik, indem er die Rohdateninterpretation übernimmt und klare, semantische Ereignisse sowie einen vereinfachten booleschen Status bereitstellt. Seine Stärke liegt in der Kombination aus Datenverarbeitung und ereignisbasierter Ausgabe gemäß der IEC 61499.

QUARTER_TO_STR_MEASURED

🎧 Podcast

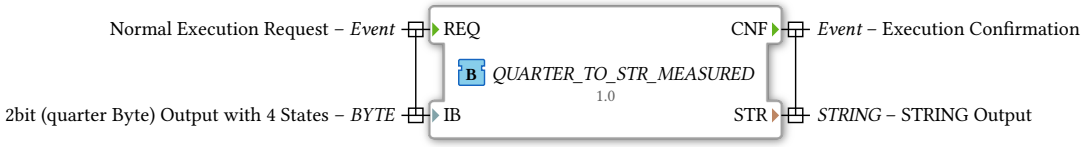


- QUARTER

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `QUARTER_TO_STR_MEASURED` dient der Konvertierung eines 4-Zustands-Signals (kodiert in den unteren zwei Bits eines BYTE-Wertes) in einen menschenlesbaren Text-String (STRING). Er ist besonders für die Anzeige oder Protokollierung von Statusinformationen in Steuerungssystemen geeignet, wo diskrete Zustände wie „Eingeschaltet“, „Ausgeschaltet“, „Fehler“ oder „Nicht verfügbar“ in eine textuelle Form überführt werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die normale Ausführung des Funktionsblocks. Bei diesem Ereignis wird der Wert am Daten-Eingang `IB` ausgewertet und die entsprechende Konvertierung durchgeführt.

Ereignis-Ausgänge

- **CNF**: Signalisiert die Beendigung der Konvertierung und die Verfügbarkeit des Ergebnisses am Daten-Ausgang `STR`.

Daten-Eingänge

- **IB** (BYTE): Der Eingang für den zu konvertierenden 4-Zustands-Wert (Quad-State). Es werden nur die unteren zwei Bits (LSB) ausgewertet. Der Block erwartet spezifische, vordefinierte Konstantenwerte (z.B. `quarter::STATUS_ENABLED`). Der Initialwert ist `quarter::STATUS_DISABLED`.

Daten-Ausgänge

- **STR** (STRING): Der Ausgang, an dem der konvertierte Text-String bereitgestellt wird. Der Initialwert ist die Textkonstante für den Status „Disabled“ (`quarter::STATUS_DISABLED_msg`).

Adapter

Dieser Funktionsblock verfügt über keine Adapter-Schnittstellen.

Funktionsweise

Der `QUARTER_TO_STR_MEASURED` ist ein Basic Function Block (BFB) mit einem internen Ausführungssteuerungsgraphen (ECC). Bei Eintreffen des `REQ`-Ereignisses wird der Wert am Eingang `IB` mit vordefinierten Konstanten aus der Bibliothek `logiBUS::utils::quarter::const::quarter` verglichen. Abhängig vom übereinstimmenden Wert verzweigt die Steuerung in einen von vier Zuständen (`SET`, `RESET`, `Error`, `none`). In jedem dieser Zustände wird ein spezifischer Algorithmus ausgeführt, der die entsprechende Textkonstante (z.B. `quarter::STATUS_ENABLED_msg`) dem Ausgang `STR` zuweist. Anschließend wechselt der Block in den Zustand `ok`, von dem aus das `CNF`-Ausgangsereignis ausgelöst wird, bevor der Block in seinen initialen Zustand `START` zurückkehrt und auf den nächsten `REQ` wartet.

Technische Besonderheiten

- **Typisierte Konstanten:** Der Block verwendet streng typisierte Konstanten für sowohl die Eingangswerte (`quarter::STATUS_...`) als auch die Ausgangsstrings (`quarter::STATUS_..._msg`). Dies erhöht die Typsicherheit und Wartbarkeit des Codes.
- **Basic FB:** Die Implementierung als Basic FB mit ECC ermöglicht eine klare und nachvollziehbare Zustandslogik.
- **Initialwerte:** Sowohl der Daten-Eingang `IB` als auch der Daten-Ausgang `STR` sind mit sinnvollen Initialwerten (deaktivierter Status) vorbesetzt.

Zustandsübersicht

Der ECC des Funktionsblocks besteht aus sechs Zuständen:

1. **START:** Der initiale Wartezustand. Bei `REQ` erfolgt eine Verzweigung basierend auf dem Wert von `IB`.
2. **SET:** Wird bei `IB = quarter::STATUS_ENABLED` aktiviert. Führt den Algorithmus `SET` aus.
3. **RESET:** Wird bei `IB = quarter::STATUS_DISABLED` aktiviert. Führt den Algorithmus `RESET` aus.
4. **Error:** Wird bei `IB = quarter::STATUS_ERROR` aktiviert. Führt den Algorithmus `ERROR` aus.
5. **none:** Wird bei `IB = quarter::STATUS_NOT_AVAILABLE` aktiviert. Führt den Algorithmus `NONE` aus.
6. **ok:** Gemeinsamer Zustand, der nach jeder erfolgreichen Konvertierung durchlaufen wird. Löst das `CNF`-Ereignis aus und führt zurück zum `START`-Zustand.

Anwendungsszenarien

- **HMI/SCADA-Anbindung:** Konvertierung von internen Steuerungsstatuswerten in lesbare Texte für die Darstellung auf Bedienpanels oder in Visualisierungssystemen.
- **Protokollierung und Diagnose:** Erzeugung von klartextbasierten Log-Einträgen für Fehler- oder Statusmeldungen, die einfacher zu analysieren sind als numerische Codes.
- **Schnittstelle zu übergeordneten Systemen:** Aufbereitung von Statusinformationen für die Übertragung an MES- oder ERP-Systeme, die String-Daten erwarten.

⚖️ Vergleich mit ähnlichen Bausteinen

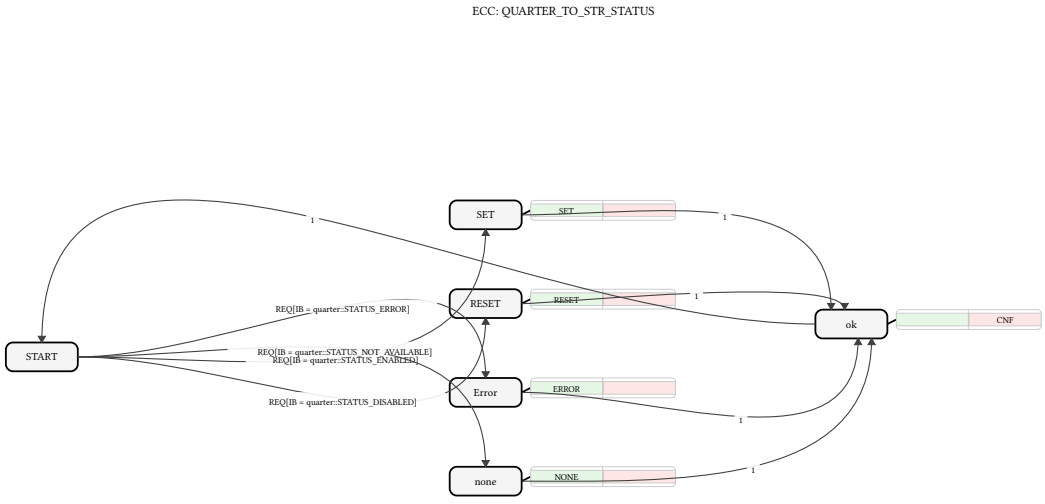
- **`BYTE_TO_STRING`:** Ein genereller Konverter, der einen beliebigen BYTE-Wert in seine dezimale String-Repräsentation umwandelt. `QUARTER_TO_STR_MEASURED` ist spezialisierter und wandelt spezifische, semantisch bedeutungsvolle Werte in vordefinierte, aussagekräftige Texte um, nicht in Zahlenstrings.
- **`E_SR` oder `E_RS` (FlipFlops):** Diese Blöcke repräsentieren binäre Zustände (SET/RESET) durch boolesche Signale. `QUARTER_TO_STR_MEASURED` erweitert dieses Konzept auf vier Zustände und bietet zusätzlich die textuelle Repräsentation.

Fazit

Der `QUARTER_TO_STR_MEASURED` ist ein spezialisierter und robuster Funktionsblock zur semantischen Konvertierung von 4-Zustands-Signalen. Durch die Verwendung von Bibliothekskonstanten für Ein- und Ausgänge gewährleistet er eine hohe Codequalität und vereinfacht die Wartung. Er ist die ideale Wahl, wenn diskrete Statusinformationen aus der Steuerungsebene für Anzeige-, Protokollierungs- oder Schnittstellenzwecke in eine textuelle Form gebracht werden müssen.

QUARTER_TO_STR_STATUS

🎧 Podcast

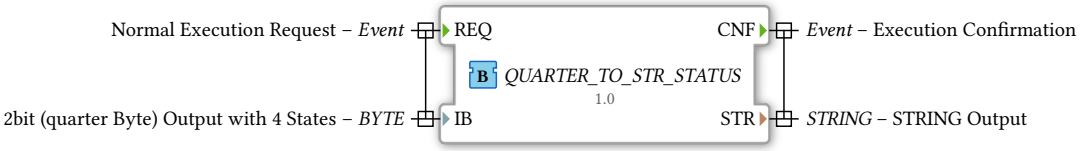


- QUARTER

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `QUARTER_TO_STR_STATUS` dient der Konvertierung eines 2-Bit-Statuswerts (auch als „Quarter Byte“ bezeichnet) in einen menschenlesbaren Text-String. Er ist Teil der `logiBUS::utils::quarter`-Bibliothek und wird typischerweise verwendet, um kompakte Statusinformationen aus Steuerungen oder Geräten für Anzeige-, Protokollierungs- oder Diagnosezwecke in eine verständliche Textform zu überführen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Verarbeitung. Bei Eintreffen dieses Ereignisses wird der Wert am Daten-Eingang `IB` ausgelesen und die entsprechende Konvertierung durchgeführt.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieses Ereignis wird zusammen mit dem konvertierten String am Daten-Ausgang `STR` ausgegeben.

Daten-Eingänge

- **IB** (BYTE): Der Eingang für den 2-Bit-Statuswert. Es werden nur die untersten zwei Bits (Bit 0 und 1) ausgewertet. Der Block erwartet spezifische, vordefinierte Konstanten aus der `quarter`-Bibliothek. Der Initialwert ist `quarter::COMMAND_DISABLE`.

Daten-Ausgänge

- **STR** (STRING): Der Ausgang, an dem der dem Statuswert entsprechende Text-String bereitgestellt wird. Der Initialwert ist `quarter::COMMAND_DISABLE_msg`.

Adapter

Dieser Funktionsblock verfügt über keine Adapter-Schnittstellen.

Funktionsweise

Der `QUARTER_TO_STR_STATUS` ist ein Basic Function Block (BFB) mit einem definierten Execution Control Chart (ECC). Bei einem eingehenden `REQ`-Ereignis wird der Wert an `IB` mit vordefinierten Konstanten verglichen. Abhängig von diesem Vergleich verzweigt die Steuerung in einen von vier möglichen Zuständen (`SET`, `RESET`, `Error`, `none`). In jedem dieser Zustände wird ein spezifischer Algorithmus ausgeführt, der den Ausgangs-`STR` auf einen entsprechenden Text-String setzt. Nach Ausführung des Algorithmus wechselt der Block in den Zustand `ok`, von dem aus das `CNF`-Ereignis ausgelöst wird. Anschließend kehrt der Block in den initialen `START`-Zustand zurück und ist bereit für die nächste Anfrage.

Die konkreten String-Werte werden aus der `quarter`-Konstanten-Bibliothek (`logiBUS::utils::quarter::const::quarter`) geladen. Die erwarteten Eingangswerte und ihre zugehörigen Ausgangsstrings sind:

- `quarter::STATUS_ENABLED` ⟶ `quarter::COMMAND_ENABLE_msg`
- `quarter::STATUS_DISABLED` ⟶ `quarter::COMMAND_DISABLE_msg`
- `quarter::STATUS_ERROR` ⟶ `quarter::COMMAND_RESERVED_msg`
- `quarter::STATUS_NOT_AVAILABLE` ⟶ `quarter::COMMAND_NO_ACTION_msg`

Technische Besonderheiten

- **Typensicherheit:** Der Block verwendet streng typisierte Konstanten aus einer speziellen Bibliothek, was die Fehleranfälligkeit gegenüber der direkten Verwendung von Rohwerten (z.B. 0, 1, 2, 3) verringert.

- **2-Bit-Verarbeitung:** Obwohl der Eingang als `BYTE` deklariert ist, wird nur ein Quarter (2 Bit) dieses Bytes effektiv genutzt. Die Semantik der vier möglichen Zustände ist durch die verwendete Konstantenbibliothek festgelegt.
- **Deterministisches Verhalten:** Die Zustandsübergänge sind ausschließlich vom Eingangswert bei `REQ` abhängig. Es gibt keine internen Speicher oder Hysterese-Effekte.

Zustandsübersicht

Der ECC besteht aus sechs Zuständen:

1. **START:** Initialer, wartender Zustand. Hier wird auf das `REQ`-Ereignis gewartet.
2. **SET:** Wird bei `IB = quarter::STATUS_ENABLED` aktiviert. Führt Algorithmus `SET` aus.
3. **RESET:** Wird bei `IB = quarter::STATUS_DISABLED` aktiviert. Führt Algorithmus `RESET` aus.
4. **Error:** Wird bei `IB = quarter::STATUS_ERROR` aktiviert. Führt Algorithmus `ERROR` aus.
5. **none:** Wird bei `IB = quarter::STATUS_NOT_AVAILABLE` aktiviert. Führt Algorithmus `NONE` aus.
6. **ok:** Gemeinsamer Zustand nach jeder erfolgreichen Konvertierung. Löst das `CNF`-Ausgangsereignis aus und führt zurück zum `START`-Zustand.

Anwendungsszenarien

- **HMI-/SCADA-Anbindung:** Konvertierung von internen Gerätestati (z.B. „aktiviert“, „fehlerhaft“) in Strings für die Anzeige auf Bedienpanels oder in Visualisierungssoftware.
- **Protokollierung und Diagnose:** Umwandlung von Statuscodes in lesbare Texte für Log-Dateien oder Diagnosetools, um die Fehleranalyse zu erleichtern.
- **Schnittstelle zu Text-basierten Systemen:** Vorbereitung von Statusinformationen für die Weiterverarbeitung in Systemen, die mit String-Nachrichten arbeiten (z.B. MQTT-Topics, CSV-Export).

⚖️ Vergleich mit ähnlichen Bausteinen

- **`E_SR` oder `E_RS` (Flip-Flops):** Diese Blöcke speichern einen binären Zustand (SET/RESET). `QUARTER_TO_STR_STATUS` hingegen konvertiert einen vorhandenen 4-Zustands-Wert nur in einen String; er besitzt keinen eigenen Speicher.
- **`E_SELECT` oder `E_MUX`:** Können ebenfalls zwischen verschiedenen Pfaden/Werten wählen, sind aber generischer und nicht speziell auf die Konvertierung in Strings mit vordefinierten Quarter-Statuswerten ausgelegt.
- **Einfache `STRING`-Zuweisung:** Eine direkte Zuweisung im ST-Code könnte ähnliches leisten, aber der `QUARTER_TO_STR_STATUS`-Block kapselt die Logik, fördert die Wiederverwendbarkeit und erzwingt die Verwendung der standardisierten Konstanten, was die Konsistenz im gesamten Projekt erhöht.

🛠️ Zugehörige Übungen

- Uebung_055
- Uebung_056

Fazit

Der `QUARTER_TO_STR_STATUS` ist ein spezialisierter und nützlicher Funktionsblock für Anwendungen, die mit dem spezifischen 4-Zustands-Statusmodell (Quarter Byte) arbeiten. Durch die Nutzung einer Konstantenbibliothek und die klare Trennung von Logik und Schnittstelle trägt er zur Robustheit und Wartbarkeit von IEC 61499-Applikationen bei. Er ist die ideale Wahl, wenn kompakte Statusinformationen zuverlässig und einheitlich in eine für den Menschen lesbare Form gebracht werden müssen.

const

quarter

🎧 Podcast

- Das Geheimnis des 2-Bit-Quarter: Effizienz im CAN-Bus für Nutzfahrzeuge
- QUARTER

• – ◦ • – ◦ • – ◦ •

Einleitung

Die `quarter`-Bibliothek ist eine Sammlung globaler Konstanten, die für die Arbeit mit SAE J1939-konformen Status- und Befehlswerten in logiBUS-Anwendungen vorgesehen ist. Sie definiert standardisierte 2-Bit-Werte (repräsentiert als `BYTE`) und zugehörige Klartextbeschreibungen (`STRING`) für Zustände und Steuerbefehle. Diese Konstanten dienen der Vereinheitlichung und verbesserten Lesbarkeit von Code, der mit binären Statusinformationen und Steuerkommandos arbeitet.

Schnittstellenstruktur

Bei `quarter` handelt es sich um eine Sammlung globaler Konstanten (`GLOBALCONSTANTS`). Es ist daher kein Funktionsblock (FB) im klassischen Sinne mit Ereignis- oder Datenschnittstellen. Die Schnittstelle besteht ausschließlich aus den definierten Konstanten, die direkt im Code referenziert werden können.

Daten-Eingänge

Nicht vorhanden (Konstantensammlung).

Daten-Ausgänge

Nicht vorhanden (Konstantensammlung). Die Konstanten selbst stellen symbolische Namen für feste Werte dar.

Ereignis-Eingänge

Nicht vorhanden (Konstantensammlung).

Ereignis-Ausgänge

Nicht vorhanden (Konstantensammlung).

Adapter

Nicht vorhanden (Konstantensammlung).

Funktionsweise

Die Bibliothek stellt lediglich benannte Konstanten zur Verfügung. Bei der Verwendung im Code wird der symbolische Name (z.B. `STATUS_ENABLED`) durch den zugewiesenen Wert (z.B. `2#01`) ersetzt. Die `_msg`-Varianten enthalten beschreibende Texte, die für Debugging, Logging oder Benutzerausgaben genutzt werden können.

Technische Besonderheiten

- **Datentypen:** Alle Status- und Befehlskonstanten sind vom Typ `BYTE`, obwohl sie nur die unteren zwei Bits nutzen (Werte 0-3). Die zugehörigen Textkonstanten sind vom Typ `STRING`.
- **Werte:** Die Werte sind als Binärliterale (`2#00`, `2#01`, etc.) definiert, was ihre Interpretation als 2-Bit-Felder klar hervorhebt.
- **Paket:** Die Konstanten sind im Paket `logiBUS::utils::quarter::const` organisiert.

Zustandsübersicht

Nicht anwendbar, da es sich um eine statische Wertesammlung handelt.

Anwendungsszenarien

- **SAE J1939-Kommunikation:** Verwendung in Funktionsblöcken, die Parameter gemäß SAE J1939 senden oder empfangen, insbesondere für Status- und Steuerinformationen, die als 2-Bit-Felder übertragen werden.
- **Statusverwaltung:** Zuweisung und Abfrage von Geräte- oder Funktionszuständen (z.B. ein/aus, fehlerhaft, nicht verfügbar) innerhalb einer Steuerungsapplikation.
- **Kommando-Schnittstellen:** Definition von Steuerbefehlen für entfernte Geräte oder Subsysteme (deaktivieren, aktivieren, keine Aktion).
- **Logging und Diagnose:** Nutzung der `_msg`-Strings zur Generierung lesbarer Log-Einträge oder Anzeigen im Bedienpanel.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`logiBUS::utils::bit::const`:** Enthält Konstanten für einzelne Bits (z.B. `BIT_FALSE`, `BIT_TRUE`). `quarter` erweitert dieses Konzept auf 2-Bit-Felder mit anwendungsspezifischer Semantik.
- **Manuelle Definition:** Anstelle der Verwendung dieser Bibliothek könnten die Werte direkt im Code als Zahlen oder Binärliterale geschrieben werden. Die Verwendung von `quarter` erhöht jedoch die Wartbarkeit, Lesbarkeit und Konsistenz über das gesamte Projekt hinweg.
- **SAE J1939-spezifische Bibliotheken:** `quarter` ist eine grundlegende Hilfsbibliothek. Umfangreichere J1939-Bibliotheken würden wahrscheinlich auf diese Konstanten zurückgreifen, um spezifischere Nachrichten (PGNs) oder Parameter (SPNs) zu definieren.

Fazit

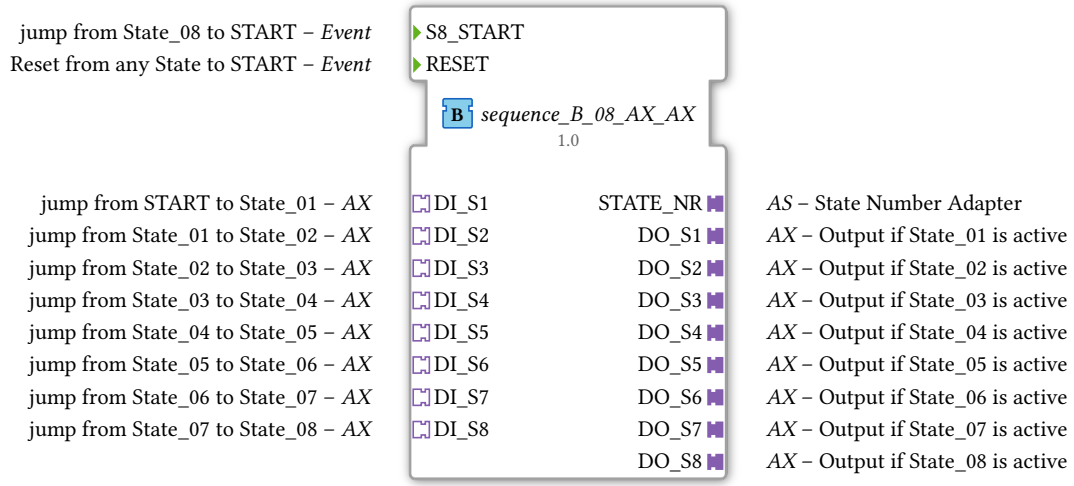
Die `quarter`-Konstantenbibliothek ist ein nützliches Hilfsmittel für die Entwicklung von Steuerungssoftware im Umfeld von SAE J1939 und anderen Systemen, die 2-Bit-Status- und Steuerfelder verwenden. Sie fördert standardisierte, selbsterklärende Programmierung und reduziert die Fehleranfälligkeit durch die Verwendung von „Magic Numbers“. Ihre Integration in das `logiBUS::utils`-Paket macht sie zu einer grundlegenden Komponente für strukturierte logiBUS-Applikationen.

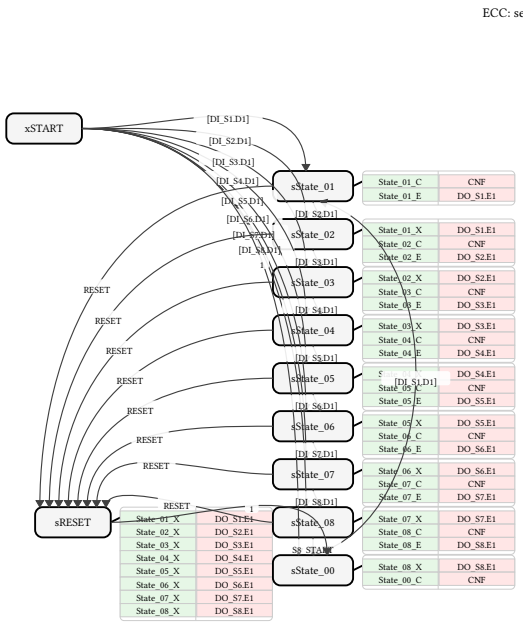
sequence

Numerischer Index (sequence)

Hier finden Sie alle Übungen des Arbeitsbereichs `sequence` in aufsteigender Reihenfolge.

sequence_B_08_AX_AX





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **sequence_B_08_AX_AX** realisiert eine sequenzielle Ablaufsteuerung mit acht Ausgängen. Die Zustandsübergänge erfolgen pegelgesteuert über BOOL-Signale, die über einen AX-Adapter bereitgestellt werden. Der Baustein ist für unterbrechungssichere Anwendungen ausgelegt und erlaubt eine Wiederherstellung des laufenden Zustands nach einem Stromausfall. Er eignet sich besonders für Ablaufsteuerungen in der Automatisierungstechnik, bei denen mehrere Schaltvorgänge nacheinander ausgeführt werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Beschreibung
S8_START	Ereignis, das einen Sprung von Zustand 8 zurück zum Startzustand auslöst.
RESET	Ereignis, das aus jedem aktiven Zustand einen sofortigen Reset auslöst.

Ereignis-Ausgänge

Name	Beschreibung
CNF	Bestätigungsereignis, das nach jedem Zustandswechsel die aktuelle Zustandsnummer ausgibt. (Mit STATE_NR)

Daten-Eingänge

Der FB besitzt keine direkten Dateneingänge. Die Übergangsbedingungen werden ausschließlich über die Socket-Adapter eingelesen.

Daten-Ausgänge

Name	Typ	Beschreibung

STATE_NR	SINT	Nummer des aktuellen Zustands: 0 = Start, 1…8 = State_01…State_08, 9 = State_00 (Ende).

Adapter

Plugs (Ausgänge – Typ `adapter::types::unidirectional::AX`)

Name	Beschreibung
DO_S1	Ausgang aktiv, wenn Zustand 1 aktiv ist.
DO_S2	Ausgang aktiv, wenn Zustand 2 aktiv ist.
DO_S3	Ausgang aktiv, wenn Zustand 3 aktiv ist.
DO_S4	Ausgang aktiv, wenn Zustand 4 aktiv ist.
DO_S5	Ausgang aktiv, wenn Zustand 5 aktiv ist.
DO_S6	Ausgang aktiv, wenn Zustand 6 aktiv ist.
DO_S7	Ausgang aktiv, wenn Zustand 7 aktiv ist.
DO_S8	Ausgang aktiv, wenn Zustand 8 aktiv ist.

Sockets (Eingänge – Typ `adapter::types::unidirectional::AX`)

Name	Beschreibung
DI_S1	Signal für Übergang vom Startzustand zu Zustand 1.
DI_S2	Signal für Übergang von Zustand 1 zu Zustand 2.
DI_S3	Signal für Übergang von Zustand 2 zu Zustand 3.
DI_S4	Signal für Übergang von Zustand 3 zu Zustand 4.
DI_S5	Signal für Übergang von Zustand 4 zu Zustand 5.
DI_S6	Signal für Übergang von Zustand 5 zu Zustand 6.
DI_S7	Signal für Übergang von Zustand 6 zu Zustand 7.
DI_S8	Signal für Übergang von Zustand 7 zu Zustand 8.

Funktionsweise

Der FB arbeitet auf Basis eines Ereignisgesteuerten Ablaufzustandsautomaten (ECC). Beim Start befindet er sich im Zustand `xSTART`. Von dort werden die Eingangssignale `DI_S1` bis `DI_S8` (jeweils das Attribut `.D1` des Adapters) auf ihren BOOL-Wert geprüft. Trifft ein Signal zu (TRUE), so wechselt der Automat in den entsprechenden Zustand (`sState_01` … `sState_08`). Ist kein Signal gesetzt, wird der Zustand `sState_00` erreicht (Ende der Sequenz).

Jeder Zustand führt beim Betreten folgende Aktionen aus:

1. **Deaktivieren des vorherigen Ausgangs**: Der Ausgangsadapter des vorherigen Zustands wird auf `FALSE` gesetzt (Algorithmus `State_n_X`).
2. **Aktualisieren der Zustandsnummer**: `STATE_NR` wird auf die Konstante des aktuellen Zustands gesetzt (z. B. `sequence::State_01`).
3. **Aktivieren des neuen Ausgangs**: Der Wert des zugehörigen Eingangs (`DI_Sn.D1`) wird auf den Ausgang (`DO_Sn.D1`) übertragen (Algorithmus `State_n_E`).
4. **Ausgabe des Bestätigungsereignisses**: `CNF` wird ausgelöst.

Von Zustand 8 führt das Ereignis `S8_START` zurück zum Startzustand (`xSTART`). Das Ereignis `RESET` kann aus jedem aktiven Zustand (1…8) eine sofortige Rückkehr zum Zustand `sState_00` erzwingen – dabei werden alle Ausgänge auf `FALSE` gesetzt.

Nach einem Reset (`RESET`) durchläuft der Automat kurz den Zustand `sRESET`, der alle Ausgänge deaktiviert, und springt dann sofort in den Zustand `sState_00`.

Technische Besonderheiten

- **Pegelgesteuerte Übergänge**: Die Transitionen werden durch den anliegenden BOOL-Wert des jeweiligen Eingangs ausgelöst (nicht flankengetriggert). Dies ermöglicht eine einfache Wiederherstellung nach einem Spannungsausfall, da der Zustand beim Wiedereinschalten direkt erkannt und fortgesetzt werden kann.
- **Verwendung von AX-Adaptern**: Die Schnittstellen sind als unidirektionale AX-Adapter ausgeführt, was eine lose Kopplung und einfache Wiederverwendung der Ein-/Ausgänge erlaubt.
- **Konfigurierbare Konstanten**: Die Zustandsnummern werden über die Konstante `sequence::State_nn` aus dem Paket `logiBUS::utils::sequence::const::sequence` bezogen – dadurch können sie zentral definiert und geändert werden.
- **Typsicherheit**: Alle Ausgänge werden explizit auf `FALSE` gesetzt, wenn ein Zustand verlassen wird, und erhalten beim Betreten den Wert des zugehörigen Eingangs. Dies verhindert Logikfehler durch hängende Werte.

Zustandsübersicht

Zustand (ECC)	Zustandsnummer	Ausgang aktiv	Übergangsbedingung (zum nächsten Zustand)
`xSTART`	0	–	`DI_S1.D1` ⟶ sState_01`DI_S2.D1` ⟶ sState_02…`DI_S8.D1` ⟶ sState_08 sonst ⟶ sState_00
`sState_01`	1	`DO_S1`	`DI_S2.D1` ⟶ sState_02`RESET` ⟶ sState_00
`sState_02`	2	`DO_S2`	`DI_S3.D1` ⟶ sState_03`RESET` ⟶ sState_00
`sState_03`	3	`DO_S3`	`DI_S4.D1` ⟶ sState_04`RESET` ⟶ sState_00
`sState_04`	4	`DO_S4`	`DI_S5.D1` ⟶ sState_05`RESET` ⟶ sState_00
`sState_05`	5	`DO_S5`	`DI_S6.D1` ⟶ sState_06`RESET` ⟶ sState_00
`sState_06`	6	`DO_S6`	`DI_S7.D1` ⟶ sState_07`RESET` ⟶ sState_00
`sState_07`	7	`DO_S7`	`DI_S8.D1` ⟶ sState_08`RESET` ⟶ sState_00
`sState_08`	8	`DO_S8`	`S8_START` ⟶ xSTARTRESET ⟶ sState_00
`sState_00`	9	–	`DI_S1.D1` ⟶ sState_01 (Neustart der Sequenz)

Hinweis: Der Zustand `sRESET` dient nur der Bereinigung aller Ausgänge und wird automatisch verlassen.

Anwendungsszenarien

- **Ablaufsteuerung in landwirtschaftlichen Maschinen**: Beispielsweise das sequenzielle Ein- und Ausschalten von acht hydraulischen Ventilen oder Beleuchtungseinheiten.
- **Förderband-Steuerung**: Aktivierung von Bandabschnitten nacheinander, wobei die Übergänge durch Füllstandssensoren (BOOL) angesteuert werden.
- **Laborautomation**: Schrittweise Freigabe von Reagenz-Dosiereinheiten, gesteuert durch Pegelsignale von Durchflusssensoren.
- **Unterbrechungssichere Abläufe**: Da der Zustand nach einem Spannungsausfall anhand der Eingangssignale wiederhergestellt werden kann, eignet sich der FB für sicherheitskritische Prozesse.

Vergleich mit ähnlichen Bausteinen

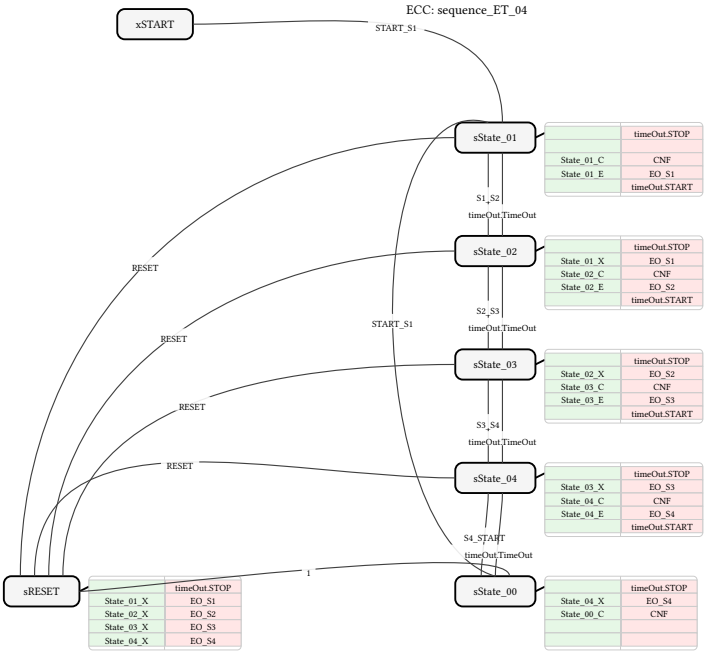
- **Timer-basierte Sequenzer**: Übergänge werden nicht durch Pegel, sondern durch Zeitabläufe ausgelöst. `sequence_B_08_AX_AX` reagiert dagegen direkt auf externe Signale und eignet sich für ereignisgesteuerte Prozesse.
- **Flankengetriggerte Sequenzer**: Würden nur auf steigende Flanken reagieren. Der hier vorgestellte FB erkennt dagegen dauerhafte Pegel, was die Wiederanlauf-Fähigkeit verbessert.
- **Adapterlose Bausteine**: Viele Standard-FBs verwenden einfache BOOL-Eingänge. Durch den Einsatz von AX-Adaptern wird eine höhere Modularität und Austauschbarkeit der Signalquellen/-senken erreicht.

Fazit

Der Funktionsblock `sequence_B_08_AX_AX` bietet eine robuste und flexible Möglichkeit zur Realisierung einer achtstufigen Ablaufsteuerung mit pegelgesteuerten Übergängen. Die Verwendung unidirektionaler AX-Adapter vereinfacht die Integration in komplexe Automatisierungssysteme, während die explizite Zustandsverwaltung und das Reset-Verhalten eine hohe Betriebssicherheit gewährleisten. Dank der Wiederherstellungsfähigkeit nach Stromausfällen eignet er sich besonders für Anwendungen in der Landtechnik und industriellen Fertigung.

combi

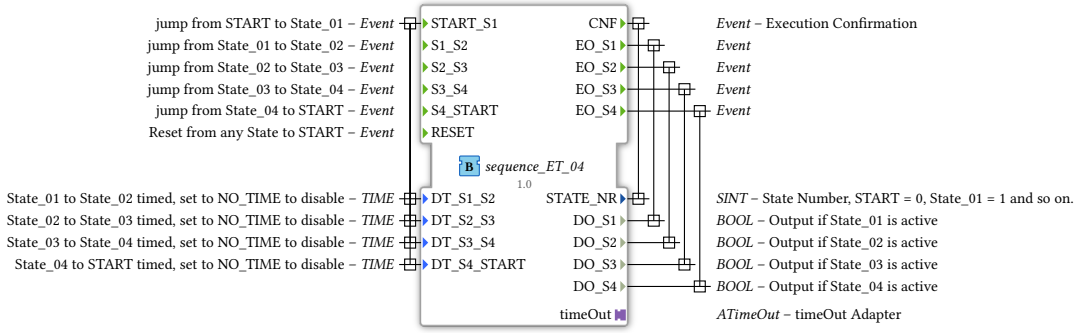
sequence_ET_04



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock `sequence_ET_04` ist ein Sequenzer mit vier Ausgabezuständen. Er ermöglicht die Steuerung einer zyklischen Abfolge von vier Schritten (State_01 bis State_04), wobei der Übergang zwischen den Zuständen entweder durch ein externes Ereignis oder nach Ablauf einer einstellbaren Zeit erfolgen kann. Der Baustein ist für Anwendungen konzipiert, bei denen Prozessschritte sequenziell und mit zeitlicher oder ereignisgesteuerter Flexibilität ausgeführt werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **START_S1**: Wechselt von START oder State_00 in den ersten Zustand State_01. Überträgt die Zeitparameter für alle Zustandsübergänge (DT_S1_S2, DT_S2_S3, DT_S3_S4, DT_S4_START).
- **S1_S2**: Manueller Übergang von State_01 zu State_02.
- **S2_S3**: Manueller Übergang von State_02 zu State_03.
- **S3_S4**: Manueller Übergang von State_03 zu State_04.
- **S4_START**: Manueller Übergang von State_04 zurück in den START-Zustand (über State_00).
- **RESET**: Setzt den Baustein aus jedem beliebigen Zustand sofort in den START-Zustand zurück.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Wird bei jedem Zustandswechsel ausgelöst und liefert die neue Zustandsnummer (STATE_NR).
- **EO_S1**: Wird bei Eintritt in State_01 ausgelöst und liefert den Ausgangswert DO_S1.
- **EO_S2**: Wird bei Eintritt in State_02 ausgelöst und liefert den Ausgangswert DO_S2.
- **EO_S3**: Wird bei Eintritt in State_03 ausgelöst und liefert den Ausgangswert DO_S3.
- **EO_S4**: Wird bei Eintritt in State_04 ausgelöst und liefert den Ausgangswert DO_S4.

Daten-Eingänge

- **DT_S1_S2** (Typ: TIME): Zeit für den automatischen Übergang von State_01 zu State_02. Der Wert NO_TIME deaktiviert den Zeitübergang für diesen Schritt.
- **DT_S2_S3** (Typ: TIME): Zeit für den automatischen Übergang von State_02 zu State_03. Der Wert NO_TIME deaktiviert den Zeitübergang für diesen Schritt.
- **DT_S3_S4** (Typ: TIME): Zeit für den automatischen Übergang von State_03 zu State_04. Der Wert NO_TIME deaktiviert den Zeitübergang für diesen Schritt.
- **DT_S4_START** (Typ: TIME): Zeit für den automatischen Übergang von State_04 zurück zum START-Zustand. Der Wert NO_TIME deaktiviert den Zeitübergang für diesen Schritt.

Daten-Ausgänge

- **STATE_NR** (Typ: SINT): Aktuelle Zustandsnummer gemäß der Konstanten sequence::State_XX (START = 0, State_01 = 1, State_02 = 2, State_03 = 3, State_04 = 4).
- **DO_S1** (Typ: BOOL): Ist TRUE, wenn State_01 aktiv ist.
- **DO_S2** (Typ: BOOL): Ist TRUE, wenn State_02 aktiv ist.
- **DO_S3** (Typ: BOOL): Ist TRUE, wenn State_03 aktiv ist.
- **DO_S4** (Typ: BOOL): Ist TRUE, wenn State_04 aktiv ist.

Adapter

- **timeOut** (Typ: ATimeOut): Ein Stecker (Plug) für einen Timeout-Adapter. Dieser Adapter wird intern verwendet, um die zeitgesteuerten Zustandsübergänge zu realisieren.

Funktionsweise

Der Baustein implementiert einen endlichen Automaten (ECC) mit den Zuständen START, State_01 bis State_04, State_00 und RESET. Der Zyklus beginnt im Zustand START. Ein START_S1-Ereignis startet die Sequenz und wechselt in State_01.

In jedem aktiven Zustand (State_01 bis State_04) werden folgende Aktionen ausgeführt:

1. Der interne Timer (timeOut) wird gestoppt.
2. Der Ausgang des vorherigen Zustands wird deaktiviert (Exit-Algorithmus State_XX_X).
3. Die Bestätigung CNF mit der neuen Zustandsnummer wird gesendet und die Zeit für den nächsten möglichen automatischen Übergang wird an den Timer übergeben (Confirmation-Algorithmus State_XX_C).
4. Der Ausgang des aktuellen Zustands wird aktiviert (Entry-Algorithmus State_XX_E).
5. Der interne Timer mit der für diesen Zustand konfigurierten Zeit (DT_...) wird gestartet.

Ein Zustandswechsel kann auf zwei Arten erfolgen:

- **Ereignisgesteuert**: Durch das entsprechende Eingangsereignis (z.B. S1_S2).
- **Zeitgesteuert**: Durch ein TimeOut-Ereignis des Adapters, sofern die Zeit DT_... nicht auf NO_TIME gesetzt ist.

Nach State_04 wechselt der Baustein in den Zustand State_00 (von wo aus die Sequenz mit START_S1 neu gestartet werden kann) oder via RESET direkt zurück in den START-Zustand. Ein RESET-Ereignis deaktiviert sofort alle aktiven Ausgänge und bringt den Baustein in den Ausgangszustand zurück.

Technische Besonderheiten

- **Hybride Übergänge**: Jeder Zustandsübergang kann individuell als rein ereignisgesteuert, rein zeitgesteuert oder als Kombination aus beidem konfiguriert werden. Ein zeitgesteuerter Übergang hat Vorrang, sofern die Zeit nicht NO_TIME ist.
- **Initialwerte**: Alle Zeit-Eingänge (DT_...) sind standardmäßig mit NO_TIME vorbelegt, was bedeutet, dass die Sequenz nach dem Start zunächst rein ereignisgesteuert abläuft.
- **Adapter-Nutzung**: Die Zeitsteuerung wird vollständig über den gekoppelten ATimeOut-Adapter abgewickelt, was eine klare Trennung der Funktionalität und potenzielle Wiederverwendbarkeit ermöglicht.
- **Zustandsrückmeldung**: Die Ausgabe STATE_NR bietet eine einfache Möglichkeit, den aktuellen Schritt extern zu überwachen oder zu visualisieren.

Zustandsübersicht

1. **START**: Initialer, inaktiver Zustand. Alle Ausgänge sind FALSE.
2. **State_01**: Erster Sequenzschritt. DO_S1 ist TRUE. Übergang zu State_02 via S1_S2-Ereignis oder nach Zeit DT_S1_S2.
3. **State_02**: Zweiter Sequenzschritt. DO_S2 ist TRUE. Übergang zu State_03 via S2_S3-Ereignis oder nach Zeit DT_S2_S3.
4. **State_03**: Dritter Sequenzschritt. DO_S3 ist TRUE. Übergang zu State_04 via S3_S4-Ereignis oder nach Zeit DT_S3_S4.
5. **State_04**: Vierter Sequenzschritt. DO_S4 ist TRUE. Übergang zu State_00 via S4_START-Ereignis oder nach Zeit DT_S4_START.
6. **State_00**: Wartezustand nach Sequenzende. Alle Ausgänge sind FALSE. Von hier aus kann mit START_S1 eine neue Sequenz gestartet werden.
7. **sRESET**: Übergangszustand für den Reset. Deaktiviert alle Ausgänge und wechselt automatisch zu State_00.

Anwendungsszenarien

- **Steuerung von Batch-Prozessen**: Sequenzielle Aktivierung von Ventilen, Pumpen oder Heizungen in einem chemischen oder verfahrenstechnischen Prozess.
- **Automatisierte Handhabungsgeräte**: Steuerung der einzelnen Schritte eines Pick-and-Place-Roboters (Greifen, Bewegen, Positionieren, Ablegen).
- **Verpackungsmaschinen**: Koordination der Abläufe wie Produktzuführung, Verpacken, Etikettieren und Ausstoßen.
- **Teststände**: Automatisierte Abfolge von Prüf- und Messschritten an einem Bauteil.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu einfachen Timer-Bausteinen (TON) oder reinen Zustandsmaschinen (E_SR) kombiniert sequence_ET_04 beides in einem spezialisierten Baustein. Er bietet eine vordefinierte, vierstufige Struktur mit dedizierten Ausgängen für jeden Schritt, was die Programmierung im Vergleich zur manuellen Verknüpfung mehrerer Einzelbausteine vereinfacht und übersichtlicher macht. Bausteine wie E_CYCLE bieten zyklische Ereignisauslösung, aber keine individuellen, zustandsabhängigen Datenausgänge oder hybride Trigger.

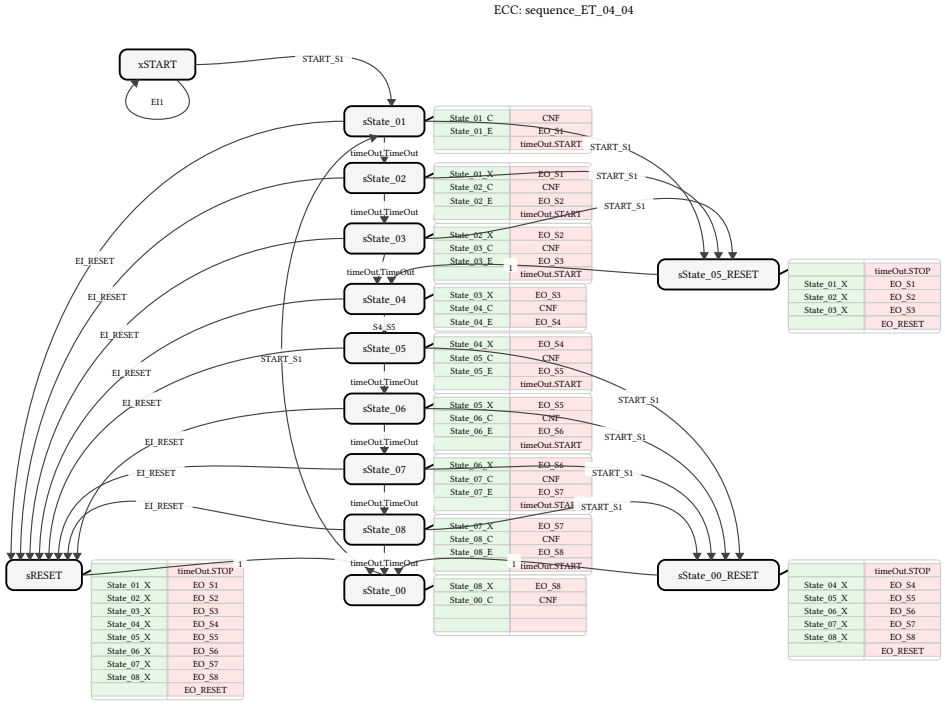
🛠️ Zugehörige Übungen

- Uebung_035
- Uebung_035b
- Uebung_035c
- Uebung_036

Fazit

Der sequence_ET_04 ist ein praktischer und flexibler Funktionsblock für alle Anwendungen, die eine klare, zyklische Schrittkette erfordern. Die Kombination aus ereignis- und zeitgesteuerten Übergängen sowie die klare Schnittstelle mit separaten Ausgängen für jeden Zustand machen ihn besonders wartungsfreundlich und einfach in übergeordnete Steuerungen zu integrieren. Die Verwendung eines Standard-Adapters für die Zeitfunktion hält den Baustein schlank und kompatibel.

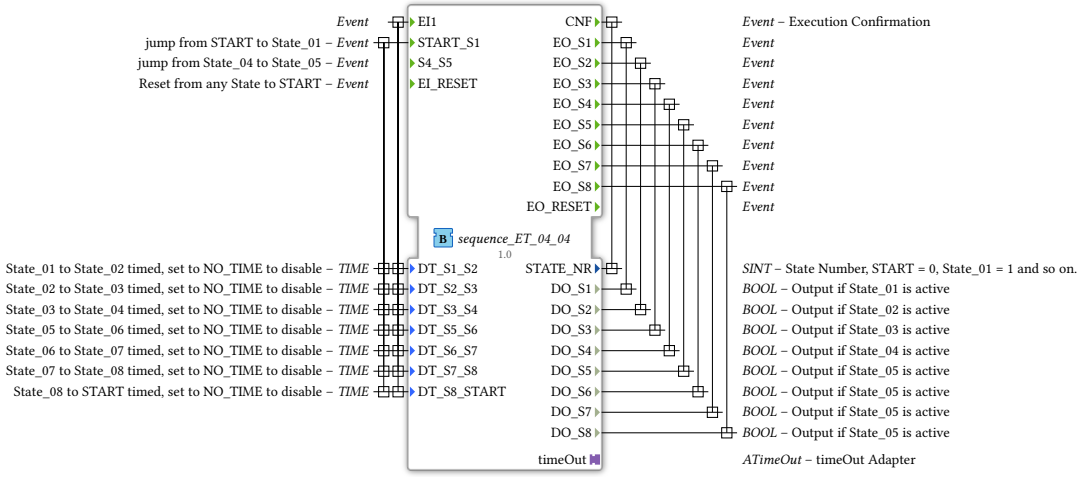
sequence_ET_04_04



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock `sequence_ET_04_04` ist ein Sequenzer mit 8 Ausgängen, der zur Steuerung von Abläufen in der Automatisierungstechnik dient. Er realisiert eine feste Zustandsfolge, wobei die Übergänge zwischen den Zuständen entweder durch ein externes Ereignis oder durch einen einstellbaren Zeitablauf ausgelöst werden können. Der Baustein ist besonders für Anwendungen geeignet, bei denen eine zyklische oder schrittweise Aktivierung von Ausgängen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **`EI1`**: Allgemeines Eingangsereignis. Führt im Startzustand (`xSTART`) zu einer Selbsttransition (kein Zustandswechsel).

- **START_S1**: Startet die Sequenz oder springt von den Zuständen 1-3 zurück in Zustand 4 bzw. von den Zuständen 5-8 zurück in den Endzustand (sState_00). Löst den Übergang vom Startzustand (xSTART) oder vom Endzustand (sState_00) in den ersten aktiven Zustand (sState_01) aus.
- **S4_S5**: Löst den Übergang vom Zustand 4 (sState_04) in den Zustand 5 (sState_05) aus. Dies ist der einzige manuelle Übergang in der Sequenz.
- **EI_RESET**: Setzt die Sequenz von jedem beliebigen aktiven Zustand in den Reset-Zustand (sRESET) und anschließend in den Endzustand (sState_00) zurück.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung (Execution Confirmation). Wird bei jedem Zustandswechsel ausgelöst und liefert die neue Zustandsnummer (STATE_NR).
- **EO_S1 bis EO_S8**: Zustandsereignisse. Werden beim Eintritt in den entsprechenden Zustand (State_01 bis State_08) ausgelöst und liefern den zugehörigen booleschen Datenausgang (DO_Sx).
- **EO_RESET**: Wird beim Durchlaufen des Reset-Zustands (sRESET) ausgelöst.

Daten-Eingänge

- **DT_S1_S2** (TIME): Zeit für den automatischen Übergang von State_01 zu State_02. Initialwert: NO_TIME (deaktiviert).
- **DT_S2_S3** (TIME): Zeit für den automatischen Übergang von State_02 zu State_03. Initialwert: NO_TIME (deaktiviert).
- **DT_S3_S4** (TIME): Zeit für den automatischen Übergang von State_03 zu State_04. Initialwert: NO_TIME (deaktiviert).
- **DT_S5_S6** (TIME): Zeit für den automatischen Übergang von State_05 zu State_06. Initialwert: NO_TIME (deaktiviert).
- **DT_S6_S7** (TIME): Zeit für den automatischen Übergang von State_06 zu State_07. Initialwert: NO_TIME (deaktiviert).
- **DT_S7_S8** (TIME): Zeit für den automatischen Übergang von State_07 zu State_08. Initialwert: NO_TIME (deaktiviert).
- **DT_S8_START** (TIME): Zeit für den automatischen Übergang von State_08 zurück in den Endzustand (sState_00). Initialwert: NO_TIME (deaktiviert).

Daten-Ausgänge

- **STATE_NR** (SINT): Aktuelle Zustandsnummer. 0 = START/State_00, 1 = State_01, ..., 8 = State_08.
- **DO_S1 bis DO_S8** (BOOL): Logische Ausgänge, die anzeigen, ob der entsprechende Zustand aktiv ist. Werden beim Zustandseintritt auf TRUE und beim Verlassen auf FALSE gesetzt.

Adapter

- **timeOut** (Plug, Typ: iec61499::events::ATimeOut): Wird für die Realisierung der zeitgesteuerten Zustandsübergänge verwendet. Der FB startet (START) und stoppt (STOP) den Timer und setzt die Verzögerungszeit (DT).

Funktionsweise

Der FB arbeitet als Basic Function Block (BFB) mit einem erweiterten endlichen Automaten (ECC). Die Sequenz durchläuft normalerweise die Zustände 1 bis 8 in fester Reihenfolge. Der Übergang von Zustand 4 zu Zustand 5 erfolgt ausschließlich durch das externe Ereignis S4_S5. Alle anderen Übergänge (1→2, 2→3, 3→4, 5→6, 6→7, 7→8, 8→Ende) können zeitgesteuert erfolgen, sofern die entsprechende Zeit (DT_...) nicht auf NO_TIME gesetzt ist.

Jeder aktive Zustand (1-8) setzt seinen zugehörigen booleschen Ausgang (DO_Sx) auf TRUE und startet ggf. den Timer für den nächsten Übergang. Beim Verlassen des Zustands wird der Ausgang auf FALSE zurückgesetzt. Bei jedem Zustandswechsel wird das Bestätigungsereignis CNF mit der neuen STATE_NR ausgelöst.

Das Ereignis START_S1 dient nicht nur zum initialen Start, sondern auch zum „Zurückspringen“ innerhalb der Sequenz: Von den Zuständen 1-3 springt es zurück in Zustand 4, von den Zuständen 5-8 zurück in den Endzustand (sState_00). Ein EI_RESET bricht die Sequenz von jedem Punkt aus ab, deaktiviert alle Ausgänge und bringt den FB in den konsistenten Endzustand sState_00.

Technische Besonderheiten

- **Hybride Triggerung**: Kombination von ereignis- und zeitgesteuerten Zustandsübergängen.

- **Flexible Zeitsteuerung**: Jeder zeitgesteuerte Übergang kann individuell durch Setzen der `DT_...`-Eingänge konfiguriert oder durch den Wert `NO_TIME` deaktiviert werden.
- **Sicherer Reset**: Der Reset-Vorgang (`EI_RESET`) stoppt sicher alle laufenden Timer und setzt alle internen Ausgänge zurück.
- **Zustandsrücksprung**: Das `START_S1`-Ereignis ermöglicht spezifische Rücksprünge in die Sequenz, was komplexe Steuerungsmuster unterstützt.

Zustandsübersicht

Der ECC umfasst folgende Zustände:

- **`xSTART`**: Initialer, inaktiver Zustand.
- **`sState_01` bis `sState_08`**: Aktive Arbeitszustände der Sequenz.
- **`sState_00`**: Inaktiver Endzustand nach Abschluss der Sequenz.
- **`sRESET`**: Reset-Zustand, der von allen aktiven Zuständen aus angesprungen werden kann.
- **`sState_05_RESET`** / **`sState_00_RESET`**: Zwischenzustände für die speziellen Rücksprungaktionen via `START_S1`.

Anwendungsszenarien

- **Steuerung von Batch-Prozessen**: Schrittweise Aktivierung von Ventilen, Pumpen oder Heizungen in einem chemischen Prozess.
- **Verkettete Maschinenabläufe**: Steuerung der einzelnen Schritte in einer Montage- oder Verpackungslinie.
- **Teststände**: Automatisierte, zyklische Prüfabläufe mit konfigurierbaren Wartezeiten zwischen den Testschritten.
- **Sicherheitssequenzen**: Geordnetes An- und Abfahren einer Anlage.

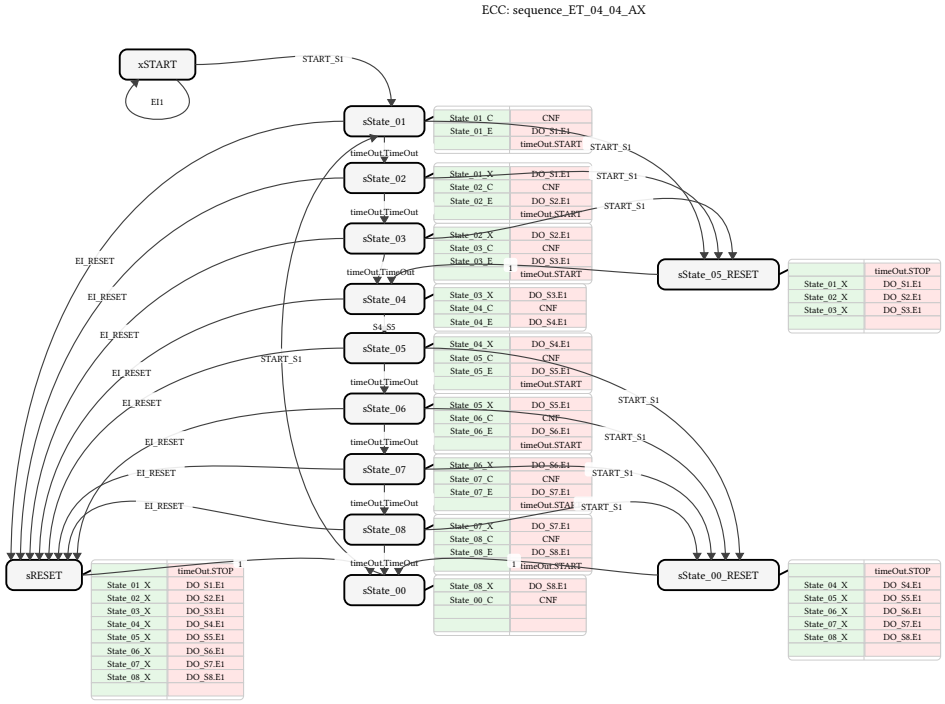
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen Timern (z.B. `TON`) oder Zählern (z.B. `CTU`) bietet dieser FB eine vordefinierte, komplexe Zustandslogik mit mehreren Ausgängen. Gegenüber generischen Schrittketten-Bausteinen (SFC) ist er weniger flexibel, aber dafür spezifischer, einfacher zu konfigurieren und bietet integrierte Reset- und Sprungfunktionen. Er stellt eine spezialisierte Lösung für 8-stufige Prozesse dar.

Fazit

Der `sequence_ET_04_04` ist ein robuster und praktischer Sequenzer für Standardautomationsaufgaben mit bis zu acht Schritten. Die Kombination aus Zeit- und Ereignissteuerung sowie die integrierten Sicherheits- und Rücksetzfunktionen machen ihn zu einer zuverlässigen Komponente für sich wiederholende Steuerungsabläufe. Seine Stärke liegt in der klaren Struktur und einfachen Parametrierung der Zustandsübergänge.

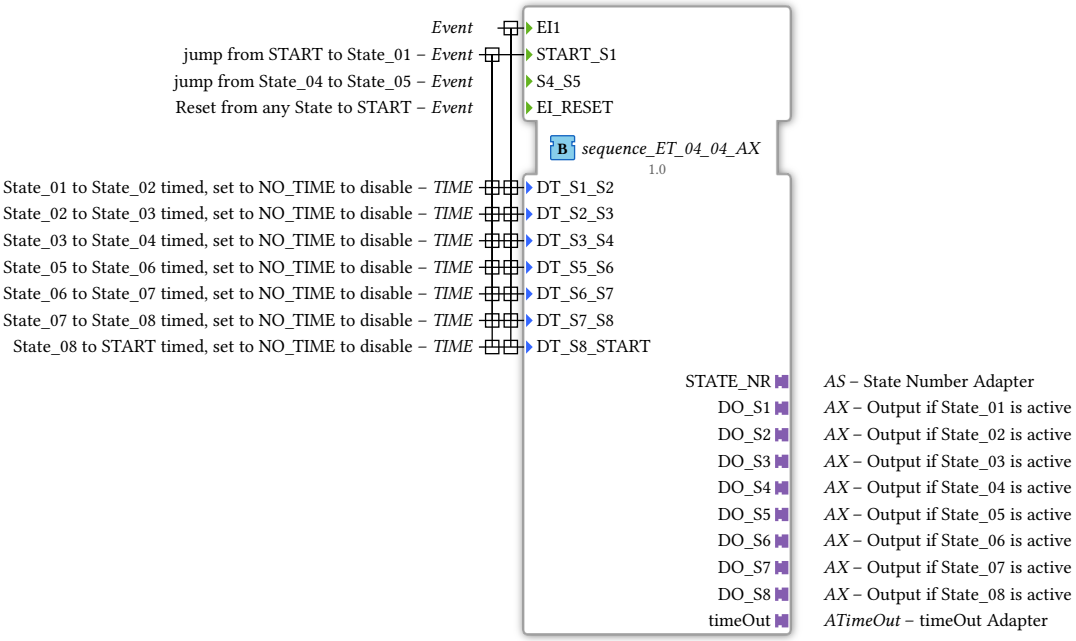
sequence_ET_04_04_AX



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock `sequence_ET_04_04_AX` ist die adapterbasierte Variante des `sequence_ET_04_04`. Er realisiert eine 8-stufige Sequenzsteuerung, bei der die Ausgänge über `AX`-Adapter (Ereignis + Daten) statt über einfache BOOL-Variablen angesteuert werden. Die Übergänge können sowohl ereignis- als auch zeitgesteuert erfolgen.



Schnittstellenstruktur

Ereignis-Eingänge

- **`EI1`**: Allgemeines Eingangsereignis.
- **`START_S1`**: Startet die Sequenz oder springt innerhalb der Phasen zurück.
- **`S4_S5`**: Manueller Übergang von Zustand 4 zu 5.
- **`EI_RESET`**: Globaler Reset der Sequenz.

Ereignis-Ausgänge

- **`CNF`**: Ausführungsbestätigung mit aktueller `STATE_NR`.
- *(Hinweis: Die zustandsspezifischen Ereignisse EO_Sx entfallen hier, da sie in den Adaptern enthalten sind)*

Daten-Eingänge

- **`DT_S1_S2`** bis **`DT_S8_START`** (TIME): Zeitverzögerungen für automatische Übergänge.

Daten-Ausgänge

- **`STATE_NR`** (SINT): Aktuelle Zustandsnummer (0-8).

Adapter

- **`DO_S1` bis `DO_S8`** (Plug, Typ: `adapter::types::unidirectional::AX`): Adapter-Ausgänge für die Zustände 1 bis 8. Jedes Betreten oder Verlassen eines Zustands löst das Ereignis `E1` am entsprechenden Adapter aus und setzt `D1` auf `TRUE` bzw. `FALSE`.
- **`timeOut`** (Plug, Typ: `iec61499::events::ATimeOut`): Schnittstelle zum externen Timer.

Funktionsweise

Die Logik entspricht exakt dem `sequence_ET_04_04`. Der entscheidende Unterschied liegt in der physikalischen Kapselung der Ausgänge. Während die Standard-Variante separate Event- und Data-Leitungen nutzt, bündelt die `_AX`-Variante diese in Adapter-Verbindungen. Dies reduziert die Komplexität im grafischen Editor bei umfangreichen Verschaltungen.

Technische Besonderheiten

✓ **Adapter-Ausgänge**: Verwendet `AX`-Adapter für alle 8 Zustände. ✓ **Kombinierte Trigger**: Ereignisse und Zeitabläufe steuern die Kette. ✓ **Zustands-Feedback**: Kontinuierliche Ausgabe der `STATE_NR`.

Zustandsübersicht

Siehe `sequence_ET_04_04`.

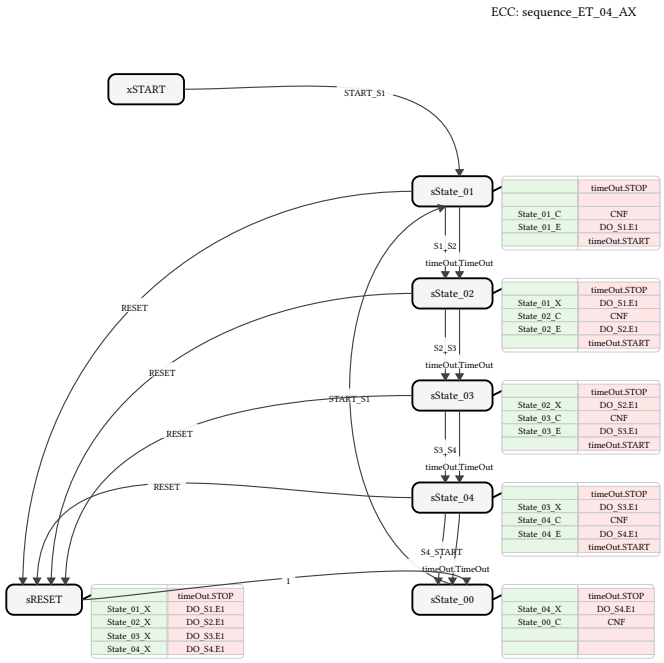
⚖️ Vergleich mit ähnlichen Bausteinen

- **sequence_ET_04_04**: Die Variante mit klassischen BOOL-Ausgängen und EO-Events.

Fazit

`sequence_ET_04_04_AX` ist die ideale Wahl für komplexe 8-Schritt-Steuerungen in Systemen, die konsequent auf adapterbasierte Kommunikation setzen.

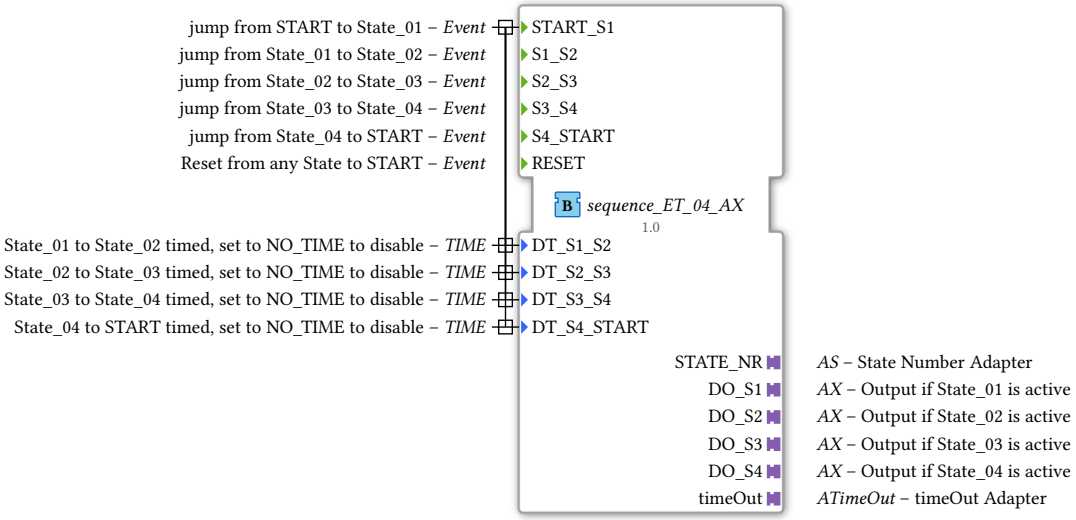
sequence_ET_04_AX



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock `sequence_ET_04_AX` ist eine Variante des `sequence_ET_04`, der zusätzlich Adapter (`AX`) für die Ausgänge verwendet. Er steuert eine Sequenz mit 4 Ausgabezuständen, wobei Übergänge ereignis- oder zeitgesteuert sein können.



Schnittstellenstruktur

Ereignis-Eingänge

- **START_S1**: Startet die Sequenz bei State_01.
- **S1_S2**: Manueller Übergang State_01 -> State_02.
- **S2_S3**: Manueller Übergang State_02 -> State_03.
- **S3_S4**: Manueller Übergang State_03 -> State_04.
- **S4_START**: Manueller Übergang State_04 -> START.

- **RESET**: Setzt die Sequenz zurück.

Ereignis-Ausgänge
- **CNF**: Bestätigung der Ausführung mit aktueller Zustandsnummer.

Daten-Eingänge
- **DT_S1_S2**: Zeit für Übergang State_01 -> State_02.
- **DT_S2_S3**: Zeit für Übergang State_02 -> State_03.
- **DT_S3_S4**: Zeit für Übergang State_03 -> State_04.
- **DT_S4_START**: Zeit für Übergang State_04 -> START.

Daten-Ausgänge
- **STATE_NR** (SINT): Aktuelle Zustandsnummer.

Adapter
- **DO_S1** (adapter::types::unidirectional::AX): Ausgangsadapter für State_01.
- **DO_S2** (adapter::types::unidirectional::AX): Ausgangsadapter für State_02.
- **DO_S3** (adapter::types::unidirectional::AX): Ausgangsadapter für State_03.
- **DO_S4** (adapter::types::unidirectional::AX): Ausgangsadapter für State_04.
- **timeOut** (iec61499::events::ATimeOut): Timer-Adapter.

Funktionsweise

Die Funktionsweise entspricht im Wesentlichen dem `sequence_ET_04`, jedoch werden die Ausgänge `DO_S1` bis `DO_S4` nicht als einfache BOOL-Variablen, sondern über Adapter gesetzt. Dies ermöglicht eine flexiblere Anbindung in komplexeren Systemen.

Technische Besonderheiten
- Verwendung von `adapter::types::unidirectional::AX` für die Ausgänge.

Zustandsübersicht

Siehe `sequence_ET_04`.

Anwendungsszenarien

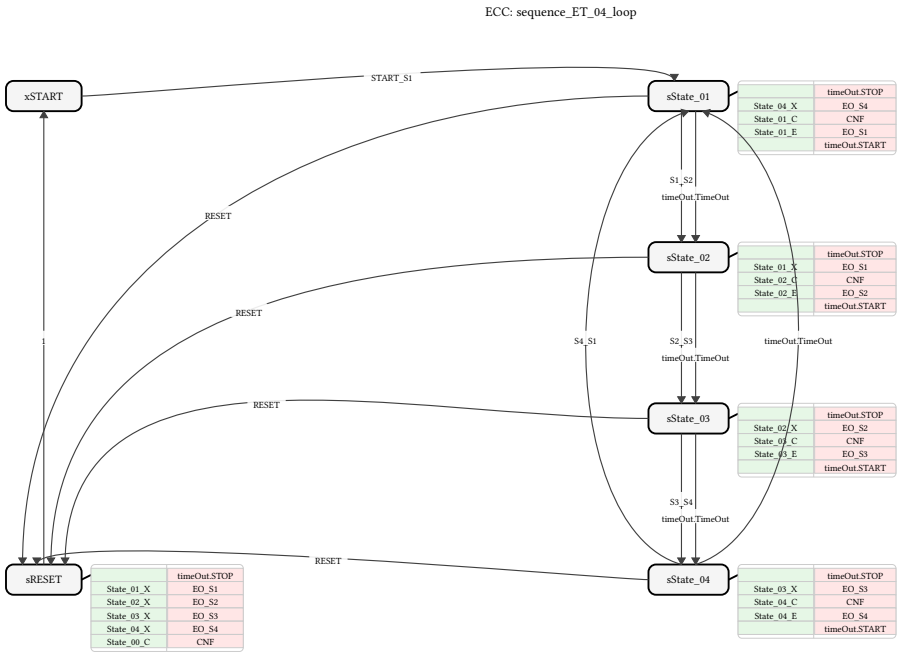
Ähnlich wie `sequence_ET_04`, aber bevorzugt, wenn Adapterverbindungen genutzt werden sollen.

⚖️ Vergleich mit ähnlichen Bausteinen
- **sequence_ET_04**: Die Standardvariante mit einfachen BOOL-Ausgängen.

Fazit

`sequence_ET_04_AX` bietet die Funktionalität des `sequence_ET_04` mit modernerer Adapter-Schnittstelle.

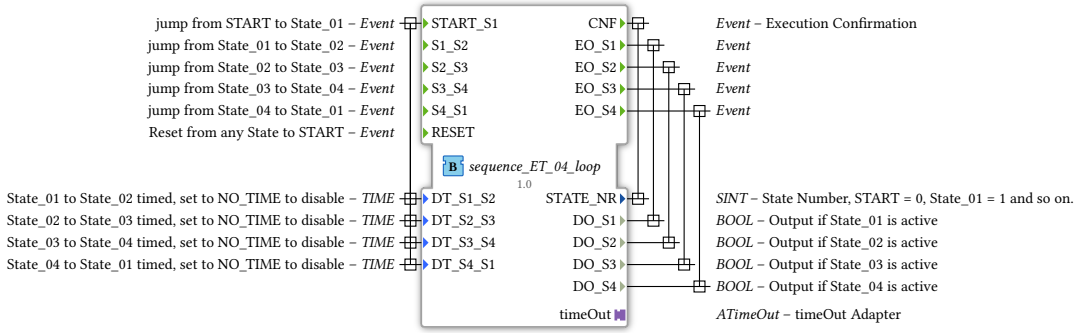
sequence_ET_04_loop



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock `sequence_ET_04_loop` implementiert eine zyklische Sequenz mit vier Zuständen. Der Übergang zwischen den Zuständen kann entweder durch ein externes Ereignis oder nach Ablauf einer einstellbaren Zeit erfolgen. Der Block ist darauf ausgelegt, wiederkehrende Abläufe in Steuerungsanwendungen zu realisieren, bei denen Aktionen nacheinander und in einer Schleife ausgeführt werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- `START_S1`: Startet die Sequenz und wechselt vom `START`-Zustand in den Zustand `State_01`. Überträgt die vier Zeitparameter `DT_S1_S2`, `DT_S2_S3`, `DT_S3_S4` und `DT_S4_S1`.
- `S1_S2`: Löst den Übergang von `State_01` zu `State_02` aus.
- `S2_S3`: Löst den Übergang von `State_02` zu `State_03` aus.
- `S3_S4`: Löst den Übergang von `State_03` zu `State_04` aus.
- `S4_S1`: Löst den Übergang von `State_04` zurück zu `State_01` aus (Schleife).
- `RESET`: Setzt die Sequenz von jedem beliebigen Zustand zurück in den `START`-Zustand.

Ereignis-Ausgänge

- `CNF`: Bestätigungsereignis, das bei jedem Zustandswechsel (einschließlich Reset) ausgelöst wird. Überträgt die aktuelle Zustandsnummer `STATE_NR`.
- `EO_S1`: Wird beim Eintritt in `State_01` ausgelöst. Überträgt den Ausgangswert `DO_S1`.
- `EO_S2`: Wird beim Eintritt in `State_02` ausgelöst. Überträgt den Ausgangswert `DO_S2`.
- `EO_S3`: Wird beim Eintritt in `State_03` ausgelöst. Überträgt den Ausgangswert `DO_S3`.
- `EO_S4`: Wird beim Eintritt in `State_04` ausgelöst. Überträgt den Ausgangswert `DO_S4`.

Daten-Eingänge

- `DT_S1_S2` (TIME): Zeit für den automatischen Übergang von `State_01` zu `State_02`. Wert `NO_TIME` deaktiviert den Zeitübergang.
- `DT_S2_S3` (TIME): Zeit für den automatischen Übergang von `State_02` zu `State_03`. Wert `NO_TIME` deaktiviert den Zeitübergang.
- `DT_S3_S4` (TIME): Zeit für den automatischen Übergang von `State_03` zu `State_04`. Wert `NO_TIME` deaktiviert den Zeitübergang.
- `DT_S4_S1` (TIME): Zeit für den automatischen Übergang von `State_04` zurück zu `State_01`. Wert `NO_TIME` deaktiviert den Zeitübergang.

Daten-Ausgänge

- `STATE_NR` (SINT): Aktuelle Zustandsnummer (`0`=START, `1`=State_01, `2`=State_02, `3`=State_03, `4`=State_04).
- `DO_S1` (BOOL): Ist `TRUE`, wenn `State_01` aktiv ist.
- `DO_S2` (BOOL): Ist `TRUE`, wenn `State_02` aktiv ist.
- `DO_S3` (BOOL): Ist `TRUE`, wenn `State_03` aktiv ist.
- `DO_S4` (BOOL): Ist `TRUE`, wenn `State_04` aktiv ist.

Adapter

- `timeOut` (Plug, Typ: `iec61499::events::ATimeOut`): Wird intern für die Realisierung der zeitgesteuerten Zustandsübergänge verwendet.

Funktionsweise

Der FB arbeitet als Basic-FB mit einer endlichen Zustandsmaschine (ECC). Die Sequenz durchläuft die Zustände `State_01` -> `State_02` -> `State_03` -> `State_04` und springt dann wieder zurück zu `State_01`. Jeder Zustand hat drei Hauptaktionen:

1. **Exit-Aktion (X)**: Setzt den zugehörigen booleschen Ausgang (`DO_Sx`) auf `FALSE`.
2. **Confirmation-Aktion (C)**: Setzt die Zustandsnummer (`STATE_NR`) und konfiguriert den `timeOut`-Adapter mit der für den nächsten Übergang vorgesehenen Zeit (`DT_...`). Löst das `CNF`-Ereignis aus.
3. **Entry-Aktion (E)**: Setzt den zugehörigen booleschen Ausgang (`DO_Sx`) auf `TRUE` und löst das entsprechende Ausgangsereignis (`EO_Sx`) aus.

Ein Zustandswechsel kann auf zwei Arten erfolgen:

1. **Durch Ereignis**: Durch das entsprechende Eingangsereignis (z.B. `S1_S2`).
2. **Durch Zeit**: Nach Ablauf der im aktuellen Zustand im `timeOut`-Adapter eingestellten Zeit, sofern diese nicht `NO_TIME` ist.

Der `RESET`-Eingang führt immer in den speziellen `sRESET`-Zustand, der alle aktiven Ausgänge ausschaltet, die Zustandsnummer auf 0 setzt und dann in den `START`-Zustand zurückkehrt.

Technische Besonderheiten

- **Hybride Triggerung**: Jeder Zustandsübergang kann individuell entweder ereignis- oder zeitgesteuert konfiguriert werden. Dies ermöglicht maximale Flexibilität innerhalb einer Sequenz.
- **Initialwerte**: Die Zeitparameter sind standardmäßig auf `NO_TIME` initialisiert, was bedeutet, dass alle zeitgesteuerten Übergänge zunächst deaktiviert sind und auf ein externes Ereignis warten.
- **Adapter-Nutzung**: Die Zeitsteuerung wird konsequent über den standardisierten `ATimeOut`-Adapter abgewickelt, was die Wiederverwendbarkeit und Klarheit erhöht.
- **Zustandsrückmeldung**: Die aktuelle Position in der Sequenz ist über den `STATE_NR`-Ausgang stets nach außen sichtbar.

Zustandsübersicht

Die ECC besteht aus sechs Zuständen:

1. **xSTART**: Initialer, inaktiver Zustand. Wartet auf `START_S1`.
2. **sState_01**: Aktiver Zustand
3. Setzt `DO_S1`. Kann via `S1_S2`-Ereignis oder Timeout zu `sState_02` wechseln.
4. **sState_02**: Aktiver Zustand
5. Setzt `DO_S2`. Kann via `S2_S3`-Ereignis oder Timeout zu `sState_03` wechseln.
6. **sState_03**: Aktiver Zustand
7. Setzt `DO_S3`. Kann via `S3_S4`-Ereignis oder Timeout zu `sState_04` wechseln.
8. **sState_04**: Aktiver Zustand
9. Setzt `DO_S4`. Kann via `S4_S1`-Ereignis oder Timeout zurück zu `sState_01` wechseln (Schleife).
10. **sRESET**: Reset-Zustand. Schaltet alle Ausgänge aus, setzt `STATE_NR` auf 0 und wechselt automatisch zurück zu `xSTART`.

Anwendungsszenarien

- **Steuerung zyklischer Prozesse**: Steuerung von Maschinen, die einen sich wiederholenden Arbeitszyklus mit mehreren Schritten durchführen (z.B. Befüllen, Erhitzen, Mischen, Entleeren).
- **Ampelschaltungen**: Modellierung einer einfachen Lichtsignalanlage mit mehreren Phasen, wobei jede Phase eine feste Zeit dauern oder vorzeitig beendet werden kann.
- **Batch-Prozesse**: Abarbeitung von Chargenprozessen, bei denen einzelne Schritte entweder durch Sensoren (Ereignisse) oder nach einer Mindestzeit beendet werden.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu einfachen Timer-Blöcken oder Flip-Flops orchestriert dieser FB eine komplette, zustandsbasierte Sequenz. Im Vergleich zu einem `E_CYCLE`-Block bietet er eine explizite Zustandslogik mit klaren Übergangsbedingungen und der Möglichkeit, jeden Schritt individuell zu triggern. Er ist spezialisierter und strukturierter als eine selbst programmierte Sequenz aus `E_SR`- und `E_DELAY`-Blöcken.

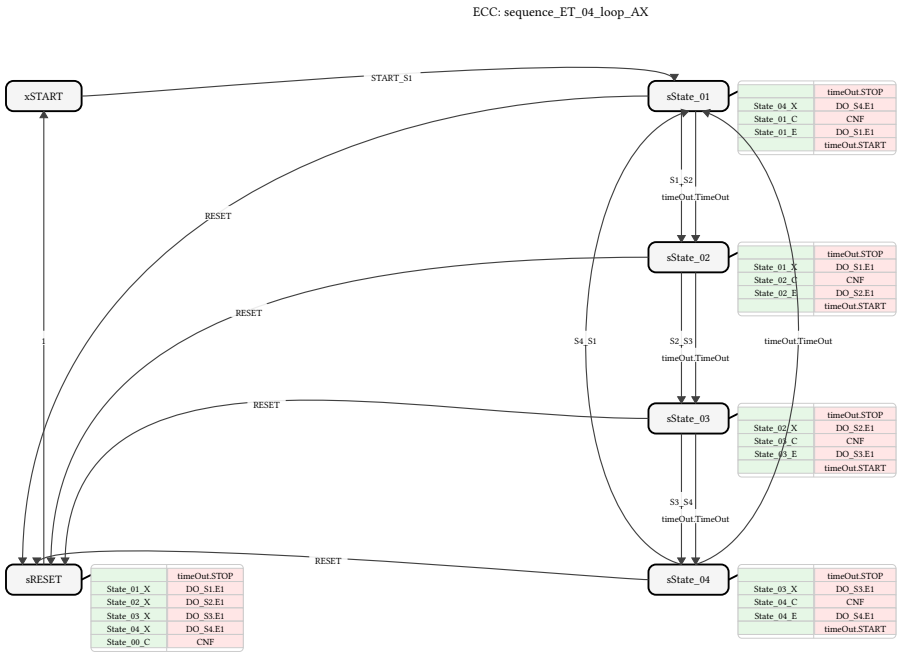
🛠️ Zugehörige Übungen

- Uebung_037

Fazit

Der `sequence_ET_04_loop` ist ein robuster und flexibler Baustein zur Implementierung von zyklischen 4-Schritt-Sequenzen. Seine Stärke liegt in der hybriden Triggerung, die für jeden Schritt zwischen Ereignis- und Zeitsteuerung wählen lässt. Die klare Zustandsmaschine, die Rückmeldung des aktuellen Zustands und der integrierte Reset machen ihn zu einer zuverlässigen Komponente für wiederkehrende Steuerungsaufgaben in der Automatisierungstechnik.

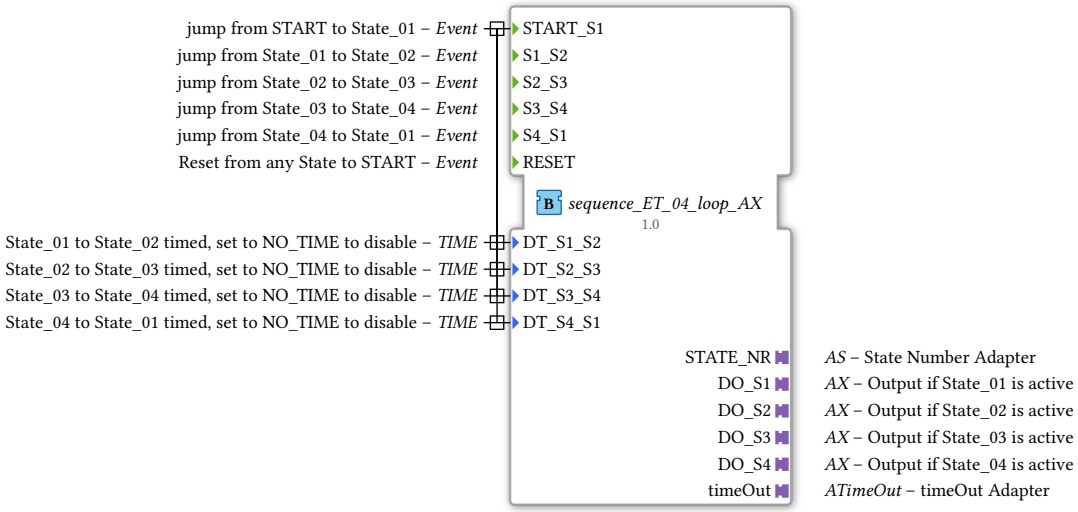
sequence_ET_04_loop_AX



• — ○ • — ○ • — ○ •

Einleitung

`sequence_ET_04_loop_AX` ist eine Variante des `sequence_ET_04_loop`, die zusätzlich Adapter (`AX`) für die Ausgänge verwendet. Er steuert eine zyklische Sequenz mit 4 Ausgabezuständen.



Schnittstellenstruktur

Ereignis-Eingänge

- **START_S1**: Startet die Sequenz bei State_01.
- **S1_S2**: Manueller Übergang State_01 -> State_02.
- **S2_S3**: Manueller Übergang State_02 -> State_03.
- **S3_S4**: Manueller Übergang State_03 -> State_04.
- **S4_S1**: Manueller Übergang State_04 -> State_01 (Loop).
- **RESET**: Setzt die Sequenz zurück.

Ereignis-Ausgänge

- **CNF**: Bestätigung der Ausführung.

Daten-Eingänge

- **DT_S1_S2**: Zeit für Übergang State_01 -> State_02.
- **DT_S2_S3**: Zeit für Übergang State_02 -> State_03.
- **DT_S3_S4**: Zeit für Übergang State_03 -> State_04.
- **DT_S4_S1**: Zeit für Übergang State_04 -> State_01.

Daten-Ausgänge

- **STATE_NR** (SINT): Aktuelle Zustandsnummer.

Adapter

- **DO_S1** (adapter::types::unidirectional::AX): Ausgangsadapter für State_01.
- **DO_S2** (adapter::types::unidirectional::AX): Ausgangsadapter für State_02.
- **DO_S3** (adapter::types::unidirectional::AX): Ausgangsadapter für State_03.
- **DO_S4** (adapter::types::unidirectional::AX): Ausgangsadapter für State_04.
- **timeOut** (iec61499::events::ATimeOut): Timer-Adapter.

Funktionsweise

Entspricht `sequence_ET_04_loop`, verwendet jedoch Adapter für die Ausgänge.

Technische Besonderheiten

- Verwendung von `adapter::types::unidirectional::AX`.

Zustandsübersicht

Siehe `sequence_ET_04_loop`.

Anwendungsszenarien

Für zyklische Prozesse, die über Adapter gesteuert werden sollen.

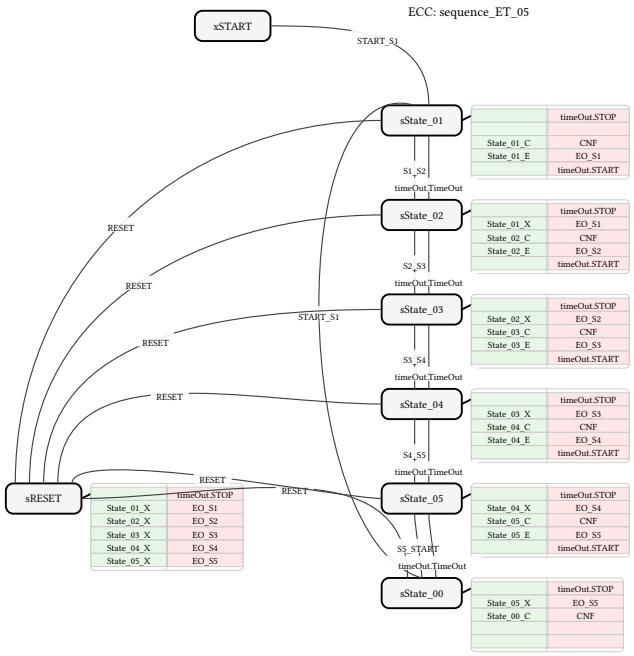
⚖️ Vergleich mit ähnlichen Bausteinen

- **sequence_ET_04_loop**: Standardvariante ohne Adapter.

Fazit

Adapter-Variante des 4-Schritt-Loop-Sequenzers.

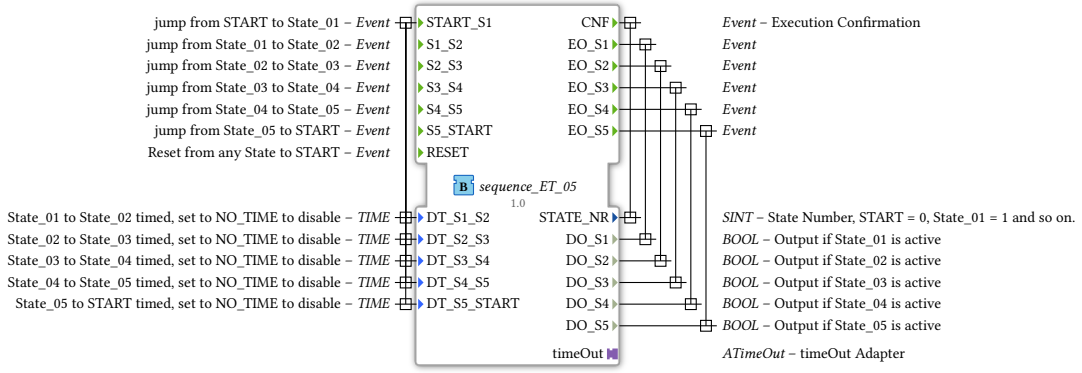
sequence_ET_05



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `sequence_ET_05` implementiert eine sequenzielle Steuerung mit fünf definierten Zuständen (State_01 bis State_05) und einem Startzustand (START). Der Übergang zwischen den Zuständen kann entweder durch ein externes Ereignis oder nach Ablauf einer einstellbaren Zeit erfolgen. Der Baustein ist für Anwendungen konzipiert, die eine schrittweise Abfolge von Aktionen mit flexiblen Übergangsbedingungen erfordern.



Schnittstellenstruktur

Ereignis-Eingänge

- **`START_S1`**: Wechselt vom Zustand START oder State_00 in den Zustand State_01. Überträgt die Zeitparameter für alle Zustandsübergänge.
- **`S1_S2`**: Manueller Übergang von State_01 zu State_02.
- **`S2_S3`**: Manueller Übergang von State_02 zu State_03.
- **`S3_S4`**: Manueller Übergang von State_03 zu State_04.
- **`S4_S5`**: Manueller Übergang von State_04 zu State_05.
- **`S5_START`**: Manueller Übergang von State_05 zurück in den Zustand State_00 (repräsentiert START).

- **RESET**: Führt von jedem beliebigen Zustand sofort zum Zustand State_00 (START) zurück und deaktiviert alle Ausgänge.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Wird bei jedem Zustandswechsel ausgelöst und liefert die neue Zustandsnummer (STATE_NR).
- **EO_S1**: Wird beim Eintritt in State_01 ausgelöst und liefert den aktiven Ausgang DO_S1.
- **EO_S2**: Wird beim Eintritt in State_02 ausgelöst und liefert den aktiven Ausgang DO_S2.
- **EO_S3**: Wird beim Eintritt in State_03 ausgelöst und liefert den aktiven Ausgang DO_S3.
- **EO_S4**: Wird beim Eintritt in State_04 ausgelöst und liefert den aktiven Ausgang DO_S4.
- **EO_S5**: Wird beim Eintritt in State_05 ausgelöst und liefert den aktiven Ausgang DO_S5.

Daten-Eingänge

- **DT_S1_S2** (TIME): Verweilzeit in State_01 vor automatischem Übergang zu State_02. Initialwert: NO_TIME (deaktiviert).
- **DT_S2_S3** (TIME): Verweilzeit in State_02 vor automatischem Übergang zu State_03. Initialwert: NO_TIME (deaktiviert).
- **DT_S3_S4** (TIME): Verweilzeit in State_03 vor automatischem Übergang zu State_04. Initialwert: NO_TIME (deaktiviert).
- **DT_S4_S5** (TIME): Verweilzeit in State_04 vor automatischem Übergang zu State_05. Initialwert: NO_TIME (deaktiviert).
- **DT_S5_START** (TIME): Verweilzeit in State_05 vor automatischem Übergang zurück zu State_00 (START). Initialwert: NO_TIME (deaktiviert).

Daten-Ausgänge

- **STATE_NR** (SINT): Aktuelle Zustandsnummer (START = 0, State_01 = 1, ..., State_05 = 5).
- **DO_S1** (BOOL): Ist TRUE, wenn State_01 aktiv ist.
- **DO_S2** (BOOL): Ist TRUE, wenn State_02 aktiv ist.
- **DO_S3** (BOOL): Ist TRUE, wenn State_03 aktiv ist.
- **DO_S4** (BOOL): Ist TRUE, wenn State_04 aktiv ist.
- **DO_S5** (BOOL): Ist TRUE, wenn State_05 aktiv ist.

Adapter

- **timeOut** (Plug, Typ: iec61499::events::ATimeOut): Wird intern für die zeitgesteuerten Zustandsübergänge verwendet. Der FB startet und stoppt den Timer und setzt dessen Laufzeit (DT).

Funktionsweise

Der FB arbeitet als Basic-FB mit einer ECC (Execution Control Chart). Der initiale Zustand ist xSTART. Bei einem START_S1-Ereignis wechselt der FB in den Zustand sState_01. In jedem aktiven Zustand (sState_01 bis sState_05) werden folgende Aktionen ausgeführt:

1. Der timeOut-Adapter wird gestoppt.
2. Der Ausgang des vorherigen Zustands wird deaktiviert (Exit-Algorithmus X).
3. Die Zustandsnummer STATE_NR wird aktualisiert und die Verweilzeit für den *nächsten* Übergang an den Timer übergeben (Confirmation-Algorithmus C). Das CNF-Ereignis wird ausgelöst.
4. Der Ausgang des aktuellen Zustands wird aktiviert (Entry-Algorithmus E). Das entsprechende EO_Sx-Ereignis wird ausgelöst.
5. Der timeOut-Adapter mit der zuvor gesetzten Zeit wird gestartet.

Ein Zustandswechsel kann nun auf zwei Arten erfolgen:

1. **Ereignisgesteuert:** Durch das entsprechende manuelle Ereignis (z.B. S1_S2).
2. **Zeitgesteuert:** Durch das timeOut.TimeOut-Ereignis des Adapters, sofern die Zeit (DT_Sx_Sy) nicht auf NO_TIME gesetzt ist.

Das RESET-Ereignis führt immer in den speziellen Zustand sRESET, in dem alle Ausgänge deaktiviert werden, und von dort sofort in den Zustand sState_00 (repräsentiert den logischen START-Zustand mit STATE_NR = 0).

Technische Besonderheiten

- **Flexible Übergänge:** Jeder Zustandsübergang kann unabhängig als ereignis- oder zeitgesteuert konfiguriert werden. Die Zeitsteuerung wird deaktiviert, indem der entsprechende `DT_`-Eingang auf die Konstante `NO_TIME` gesetzt wird.
- **Sofortiger Reset:** Der `RESET`-Eingang hat jederzeit Priorität und unterbricht die aktuelle Sequenz sofort.
- **Klar definierte Schnittstelle:** Die Zustandsnummer und die aktiven Ausgänge sind stets als Datenausgänge verfügbar und werden bei jedem Schritt per Ereignis bestätigt.
- **Adapter-basierte Zeitsteuerung:** Die Nutzung eines standardisierten TimeOut-Adapters macht die interne Zeitverwaltung robust und wiederverwendbar.

Zustandsübersicht

Die ECC umfasst folgende Zustände:

- **xSTART:** Initialer Idle-Zustand.
- **sState_01 … sState_05:** Die fünf sequenziellen Arbeitszustände.
- **sState_00:** Repräsentiert den logischen START-Zustand nach Abschluss der Sequenz oder nach einem Reset. Hier ist kein Ausgang aktiv.
- **sRESET:** Zwischenzustand zur Deaktivierung aller Ausgänge während einer Reset-Operation.

Anwendungsszenarien

- **Steuerung von Batch-Prozessen:** Schrittweise Abarbeitung von Rezepten in Misch- oder Füllanlagen, wobei ein Schritt manuell quittiert oder nach einer definierten Zeit automatisch weiter geschaltet wird.
- **Verkettete Maschinenabläufe:** Steuerung einer Maschine, deren Zyklus aus mehreren nacheinander geschalteten Aktionen besteht (z.B. Einlegen, Bearbeiten, Prüfen, Auswerfen).
- **Testsequenzen:** Automatisierte Prüfstände, die eine Reihe von Tests nacheinander durchführen, wobei jeder Test eine feste Dauer haben oder manuell bestätigt werden kann.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen Timer-Ketten oder Zählern bietet `sequence_ET_05` eine vollständig gekapselte Zustandsmaschine mit klaren Eingangs-/Ausgangsereignissen und der flexiblen Kombination aus Zeit- und Ereignissteuerung. Gegenüber einem generischen `E_SR`- oder `E_CTUD`-Baustein in einer Ad-hoc-Verknüpfung stellt er eine vordefinierte, getestete und leicht zu parametrierende Lösung für eine häufige Steuerungsaufgabe dar.

🛠️ Zugehörige Übungen

- Uebung_039
- Uebung_039a

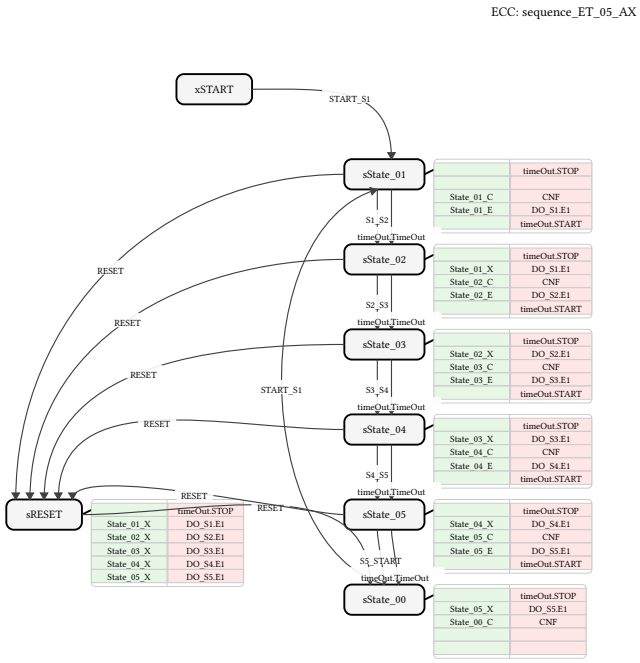
Fazit

Der `sequence_ET_05` ist ein gut strukturierter und flexibler Funktionsblock zur Implementierung von 5-stufigen Sequenzen. Die klare Trennung von Steuerfluss (Ereignisse) und Parametern (Zeiten) sowie die Möglichkeit, jeden Übergang wahlweise manuell oder automatisch zu gestalten, machen ihn für eine Vielzahl von Steuerungsaufgaben in der Automatisierungstechnik geeignet. Die Nutzung standardisierter Adapter und die ausführliche Dokumentation der Schnittstellen erleichtern die Integration in größere Anwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de

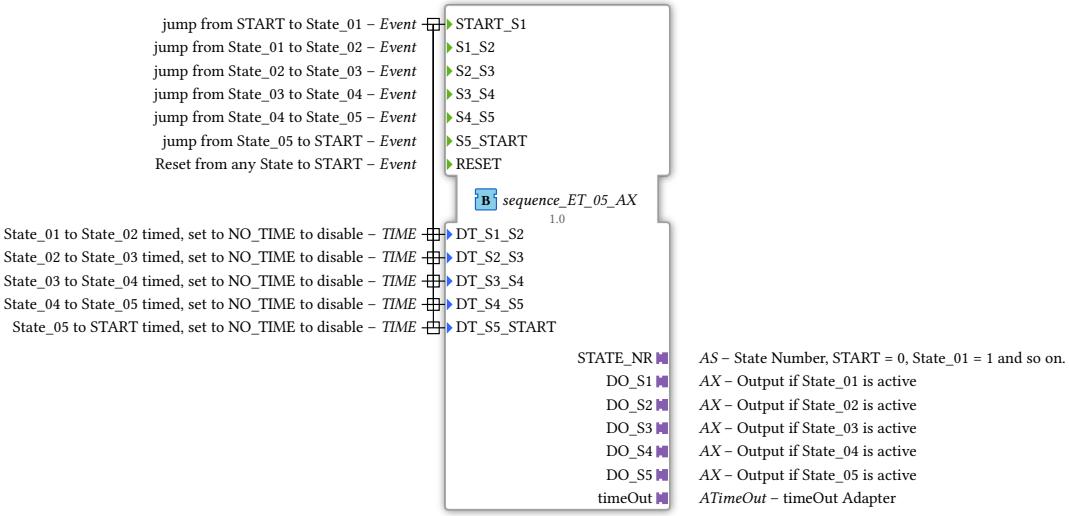
sequence_ET_05_AX



• — ○ • — ○ • — ○ •

Einleitung

`sequence_ET_05_AX` ist eine Variante des `sequence_ET_05`, die zusätzlich Adapter (`AX`) für die Ausgänge verwendet. Er steuert eine Sequenz mit 5 Ausgabezuständen.



Schnittstellenstruktur

Ereignis-Eingänge

- **START_S1**: Startet die Sequenz bei State_01.
- **S1_S2**: Manueller Übergang State_01 -> State_02.
- **S2_S3**: Manueller Übergang State_02 -> State_03.
- **S3_S4**: Manueller Übergang State_03 -> State_04.
- **S4_S5**: Manueller Übergang State_04 -> State_05.
- **S5_START**: Manueller Übergang State_05 -> START.

- **RESET**: Setzt die Sequenz zurück.

Ereignis-Ausgänge
- **CNF**: Bestätigung der Ausführung.

Daten-Eingänge
- **DT_S1_S2**: Zeit für Übergang State_01 -> State_02.
- **DT_S2_S3**: Zeit für Übergang State_02 -> State_03.
- **DT_S3_S4**: Zeit für Übergang State_03 -> State_04.
- **DT_S4_S5**: Zeit für Übergang State_04 -> State_05.
- **DT_S5_START**: Zeit für Übergang State_05 -> START.

Daten-Ausgänge
- **STATE_NR** (SINT): Aktuelle Zustandsnummer.

Adapter
- **DO_S1** (adapter::types::unidirectional::AX): Ausgangsadapter für State_01.
- **DO_S2** (adapter::types::unidirectional::AX): Ausgangsadapter für State_02.
- **DO_S3** (adapter::types::unidirectional::AX): Ausgangsadapter für State_03.
- **DO_S4** (adapter::types::unidirectional::AX): Ausgangsadapter für State_04.
- **DO_S5** (adapter::types::unidirectional::AX): Ausgangsadapter für State_05.
- **timeOut** (iec61499::events::ATimeOut): Timer-Adapter.

Funktionsweise

Entspricht `sequence_ET_05`, verwendet jedoch Adapter für die Ausgänge.

Technische Besonderheiten
- Verwendung von `adapter::types::unidirectional::AX`.

Zustandsübersicht

Siehe `sequence_ET_05`.

Anwendungsszenarien

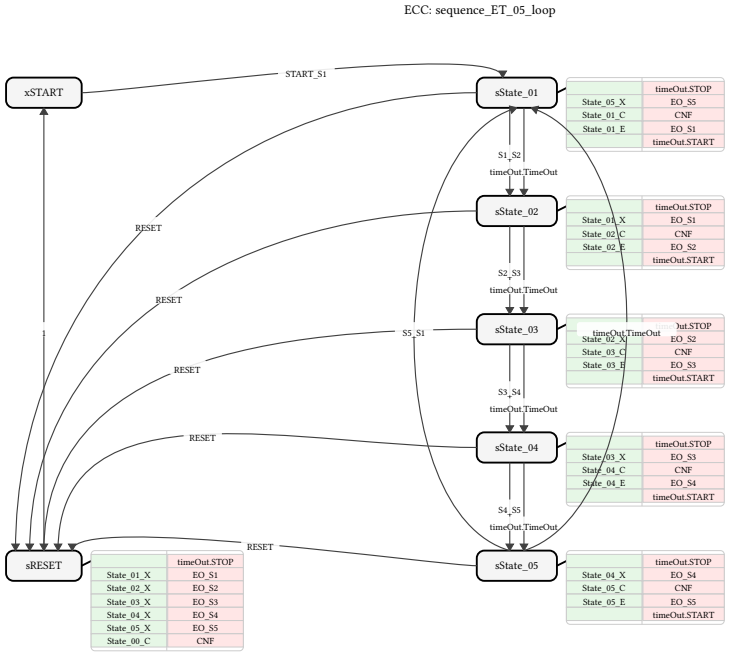
Für 5-stufige Sequenzen mit Adapteranbindung.

⚖️ Vergleich mit ähnlichen Bausteinen
- **sequence_ET_05**: Standardvariante ohne Adapter.

Fazit

Adapter-Variante des 5-Schritt-Sequenzers.

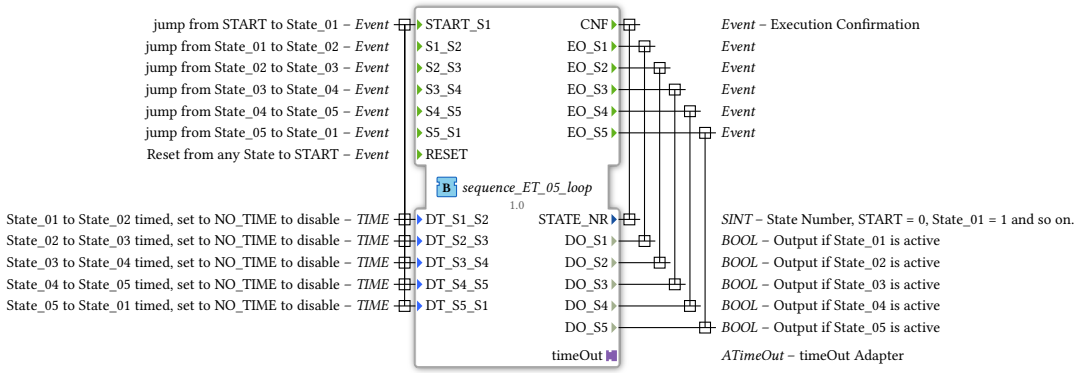
sequence_ET_05_loop



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `sequence_ET_05_loop` implementiert eine zyklische Sequenz mit fünf Zuständen. Der Übergang zwischen den einzelnen Zuständen kann entweder durch ein externes Ereignis oder nach Ablauf einer einstellbaren Zeit erfolgen. Der Baustein ist für Anwendungen konzipiert, bei denen ein Prozess schrittweise durchlaufen und dabei verschiedene Aktionen in einer festen Reihenfolge ausgelöst werden müssen, wie beispielsweise in Verpackungsmaschinen, Montageprozessen oder Waschprogrammen.



Schnittstellenstruktur

Ereignis-Eingänge

- **`START_S1`**: Startet die Sequenz und führt den Übergang vom Startzustand (`START`) in den ersten Zustand (`State_01`). Überträgt die Zeitwerte für alle zeitgesteuerten Übergänge.
- **`S1_S2`**: Löst den Übergang von `State_01` nach `State_02` aus.
- **`S2_S3`**: Löst den Übergang von `State_02` nach `State_03` aus.
- **`S3_S4`**: Löst den Übergang von `State_03` nach `State_04` aus.
- **`S4_S5`**: Löst den Übergang von `State_04` nach `State_05` aus.
- **`S5_S1`**: Löst den Übergang von `State_05` zurück in `State_01` aus (Zyklus).

- **RESET**: Setzt die Sequenz von jedem beliebigen Zustand zurück in den Startzustand (START).

Ereignis-Ausgänge

- **CNF**: Bestätigungsereignis, das bei jedem Zustandswechsel ausgelöst wird. Überträgt die aktuelle Zustandsnummer (STATE_NR).
- **EO_S1**: Wird beim Eintritt in State_01 ausgelöst. Überträgt den Ausgangswert DO_S1.
- **EO_S2**: Wird beim Eintritt in State_02 ausgelöst. Überträgt den Ausgangswert DO_S2.
- **EO_S3**: Wird beim Eintritt in State_03 ausgelöst. Überträgt den Ausgangswert DO_S3.
- **EO_S4**: Wird beim Eintritt in State_04 ausgelöst. Überträgt den Ausgangswert DO_S4.
- **EO_S5**: Wird beim Eintritt in State_05 ausgelöst. Überträgt den Ausgangswert DO_S5.

Daten-Eingänge

- **DT_S1_S2** (Typ: TIME, Initialwert: NO_TIME): Zeit für den automatischen Übergang von State_01 nach State_02. Bei NO_TIME ist der zeitgesteuerte Übergang deaktiviert.
- **DT_S2_S3** (Typ: TIME, Initialwert: NO_TIME): Zeit für den automatischen Übergang von State_02 nach State_03.
- **DT_S3_S4** (Typ: TIME, Initialwert: NO_TIME): Zeit für den automatischen Übergang von State_03 nach State_04.
- **DT_S4_S5** (Typ: TIME, Initialwert: NO_TIME): Zeit für den automatischen Übergang von State_04 nach State_05.
- **DT_S5_S1** (Typ: TIME, Initialwert: NO_TIME): Zeit für den automatischen Übergang von State_05 nach State_01.

Daten-Ausgänge

- **STATE_NR** (Typ: SINT): Gibt die Nummer des aktuellen Zustands aus (START = 0, State_01 = 1, ..., State_05 = 5).
- **DO_S1** (Typ: BOOL): Ist TRUE, wenn State_01 aktiv ist.
- **DO_S2** (Typ: BOOL): Ist TRUE, wenn State_02 aktiv ist.
- **DO_S3** (Typ: BOOL): Ist TRUE, wenn State_03 aktiv ist.
- **DO_S4** (Typ: BOOL): Ist TRUE, wenn State_04 aktiv ist.
- **DO_S5** (Typ: BOOL): Ist TRUE, wenn State_05 aktiv ist.

Adapter

- **timeOut** (Typ: ATimeOut): Ein gesteckter Adapter zur Realisierung der zeitgesteuerten Zustandsübergänge. Der FB nutzt dessen Schnittstellen START, STOP und DT (Zeitwert) sowie das Ereignis TimeOut.

Funktionsweise

Der FB ist als Basic-FB mit einer Execution Control Chart (ECC) implementiert. Die Sequenz beginnt im Zustand xSTART. Mit dem Ereignis START_S1 wird der erste Zustand sState_01 aktiviert. Bei jedem Zustandseintritt werden folgende Aktionen ausgeführt:

1. Der Timer-Adapter wird gestoppt (timeOut.STOP).
2. Der Ausgang des vorherigen Zustands wird zurückgesetzt (z.B. State_05_X für DO_S5).
3. Die Zustandsnummer STATE_NR wird aktualisiert und der Timer für den nächsten Übergang konfiguriert (z.B. State_01_C).
4. Der Ausgang des neuen Zustands wird gesetzt (z.B. State_01_E für DO_S1).
5. Der Timer-Adapter wird mit der für diesen Übergang konfigurierten Zeit gestartet (timeOut.START).

Ein Zustandswechsel kann nun auf zwei Arten erfolgen:

- **Ereignisgesteuert:** Durch das entsprechende Eingangsereignis (z.B. S1_S2).
- **Zeitgesteuert:** Durch das TimeOut-Ereignis des Adapters, sofern die Zeit (DT_...) nicht auf NO_TIME gesetzt ist.

Die Sequenz durchläuft die Zustände 1 bis 5 und springt dann von State_05 wieder zurück zu State_01, wodurch ein endloser Zyklus entsteht. Das Ereignis RESET führt unabhängig vom aktuellen Zustand in den Reset-Zustand sRESET, von wo aus die Sequenz automatisch in den xSTART-Zustand zurückkehrt.

Technische Besonderheiten

- **Hybride Triggerung:** Jeder Zustandsübergang kann individuell entweder ereignis- oder zeitgesteuert konfiguriert werden. Dies ermöglicht maximale Flexibilität innerhalb einer festen Sequenz.

- **Sichere Zustandsausgänge:** Die booleschen Ausgänge (`DO_Sx`) werden beim Verlassen eines Zustands explizit über einen Exit-Algorithmus (`State_XX_X`) zurückgesetzt. Dies verhindert ein unbeabsichtigtes Beibehalten des `TRUE`-Signals.
- **Explizite Timer-Steuerung:** Der Timer wird bei jedem Zustandswechsel neu gestartet und gestoppt, was eine präzise und deterministische Zeitsteuerung gewährleistet.
- **Konstanten:** Der FB verwendet definierte Konstanten (`sequence::State_XX`, `NO_TIME`) aus importierten Bibliotheken, was die Wartbarkeit und Lesbarkeit des Codes erhöht.

Zustandsübersicht

Die ECC besteht aus sieben Zuständen:

1. **`xSTART`**: Inaktiver Anfangszustand.
2. **`sState_01`** bis **`sState_05`**: Die fünf aktiven Sequenzzustände.
3. **`sRESET`**: Zwischenzustand, der beim Reset alle Ausgänge zurücksetzt und die Bestätigung für den `xSTART`-Zustand sendet.

Anwendungsszenarien

- **Steuerung von Zyklischen Prozessen:** Automatische Bearbeitungsmaschinen (z.B. Drehmaschinen mit mehreren Werkzeugen), bei denen jeder Schritt eine bestimmte Zeit dauert oder durch ein Sensorereignis abgeschlossen wird.
- **Batch-Prozesse:** Chargenweise Abarbeitung in der Lebensmittel- oder Pharmaindustrie, wo Mischen, Erhitzen und Abfüllen nacheinander erfolgen.
- **Sicherheitssequenzen:** Überwachte Start- oder Stoppprozeduren, bei denen jeder Schritt manuell freigegeben (`EVENT`) oder nach einer Wartezeit (`TIME`) automatisch weiterläuft.
- **Teststände:** Automatisierte Prüfabläufe, bei denen nacheinander verschiedene Messungen oder Funktionstests durchgeführt werden.

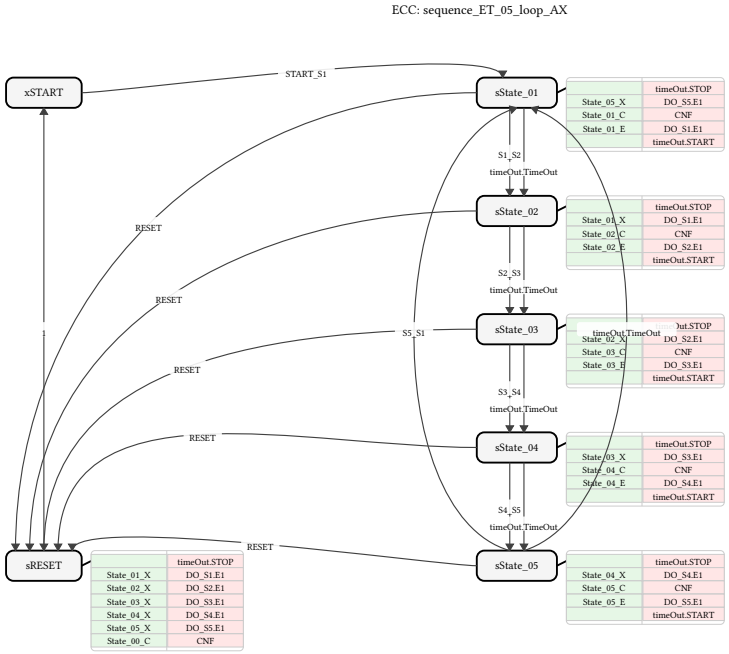
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen Timer-Bausteinen oder selbsthaltenden Relaiskaskaden bietet `sequence_ET_05_loop` eine vordefinierte, zyklische Struktur mit integrierter Zeit- und Ereignissteuerung. Gegenüber einer Programmierung mit einzelnen Blöcken (z.B. `E_SR`, `E_DELAY`) reduziert er den Verdrahtungsaufwand und erhöht die Übersichtlichkeit. Andere Sequenzer-FBs haben möglicherweise eine variable Anzahl von Schritten oder andere Übergangslogiken (nur Zeit ODER nur Ereignis). Dieser FB zeichnet sich durch die kombinierte `EVENT`/`TIME`-Triggerung pro Schritt aus.

Fazit

Der `sequence_ET_05_loop` ist ein robust und klar strukturierter Funktionsblock zur Implementierung von zyklischen 5-Schritt-Sequenzen. Die Kombination aus ereignis- und zeitgesteuerten Übergängen macht ihn äußerst flexibel für eine Vielzahl von Automatisierungsaufgaben. Die explizite Steuerung der Ausgänge und des Timers sorgt für ein deterministisches und zuverlässiges Verhalten. Er ist ideal geeignet, um wiederkehrende Prozessabläufe mit fester Schrittanzahl effizient und übersichtlich in einer IEC 61499-basierten Steuerung abzubilden.

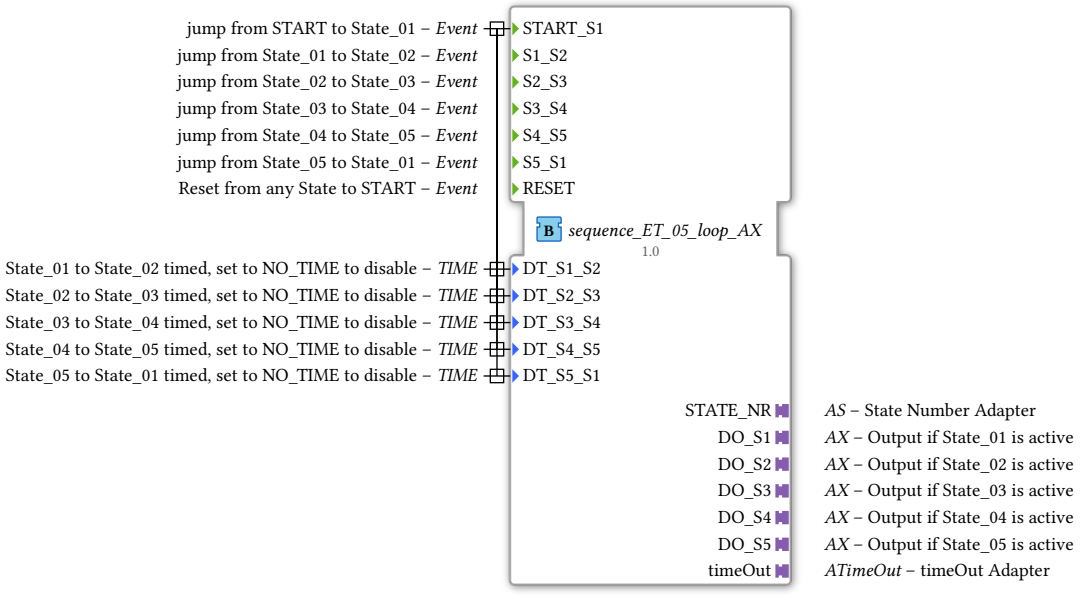
sequence_ET_05_loop_AX



• – ○ • – ○ • – ○ •

Einleitung

`sequence_ET_05_loop_AX` ist eine Variante des `sequence_ET_05_loop`, die zusätzlich Adapter (`AX`) für die Ausgänge verwendet. Er steuert eine zyklische Sequenz mit 5 Ausgabezuständen.



Schnittstellenstruktur

Ereignis-Eingänge

- **START_S1**: Startet die Sequenz bei State_01.
- **S1_S2**: Manueller Übergang State_01 -> State_02.
- **S2_S3**: Manueller Übergang State_02 -> State_03.
- **S3_S4**: Manueller Übergang State_03 -> State_04.

- **S4_S5**: Manueller Übergang State_04 -> State_05.
- **S5_S1**: Manueller Übergang State_05 -> State_01 (Loop).
- **RESET**: Setzt die Sequenz zurück.

Ereignis-Ausgänge

- **CNF**: Bestätigung der Ausführung.

Daten-Eingänge

- **DT_S1_S2**: Zeit für Übergang State_01 -> State_02.
- **DT_S2_S3**: Zeit für Übergang State_02 -> State_03.
- **DT_S3_S4**: Zeit für Übergang State_03 -> State_04.
- **DT_S4_S5**: Zeit für Übergang State_04 -> State_05.
- **DT_S5_S1**: Zeit für Übergang State_05 -> State_01.

Daten-Ausgänge

- **STATE_NR** (SINT): Aktuelle Zustandsnummer.

Adapter

- **DO_S1** (adapter::types::unidirectional::AX): Ausgangsadapter für State_01.
- **DO_S2** (adapter::types::unidirectional::AX): Ausgangsadapter für State_02.
- **DO_S3** (adapter::types::unidirectional::AX): Ausgangsadapter für State_03.
- **DO_S4** (adapter::types::unidirectional::AX): Ausgangsadapter für State_04.
- **DO_S5** (adapter::types::unidirectional::AX): Ausgangsadapter für State_05.
- **timeOut** (iec61499::events::ATimeOut): Timer-Adapter.

Funktionsweise

Entspricht `sequence_ET_05_loop`, verwendet jedoch Adapter für die Ausgänge.

Technische Besonderheiten

- Verwendung von `adapter::types::unidirectional::AX`.

Zustandsübersicht

Siehe `sequence_ET_05_loop`.

Anwendungsszenarien

Für zyklische 5-stufige Prozesse, die über Adapter gesteuert werden sollen.

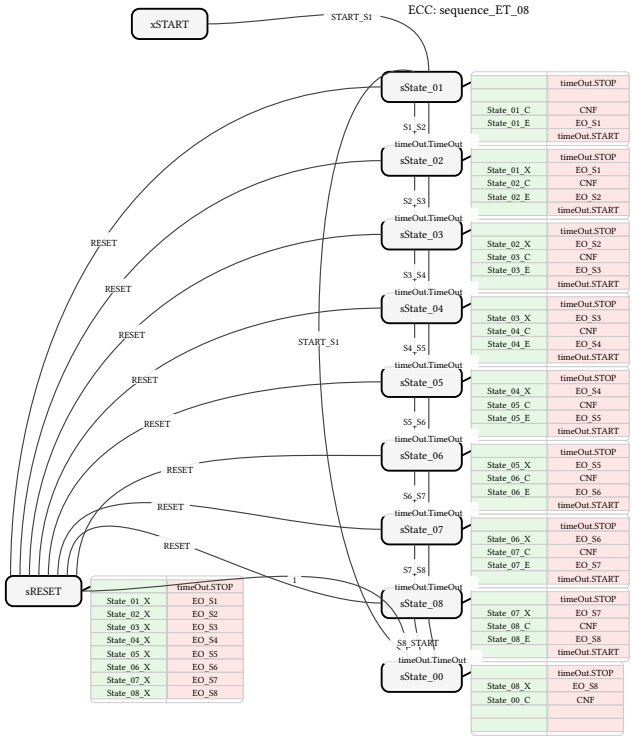
⚖️ Vergleich mit ähnlichen Bausteinen

- **sequence_ET_05_loop**: Standardvariante ohne Adapter.

Fazit

Adapter-Variante des 5-Schritt-Loop-Sequenzers.

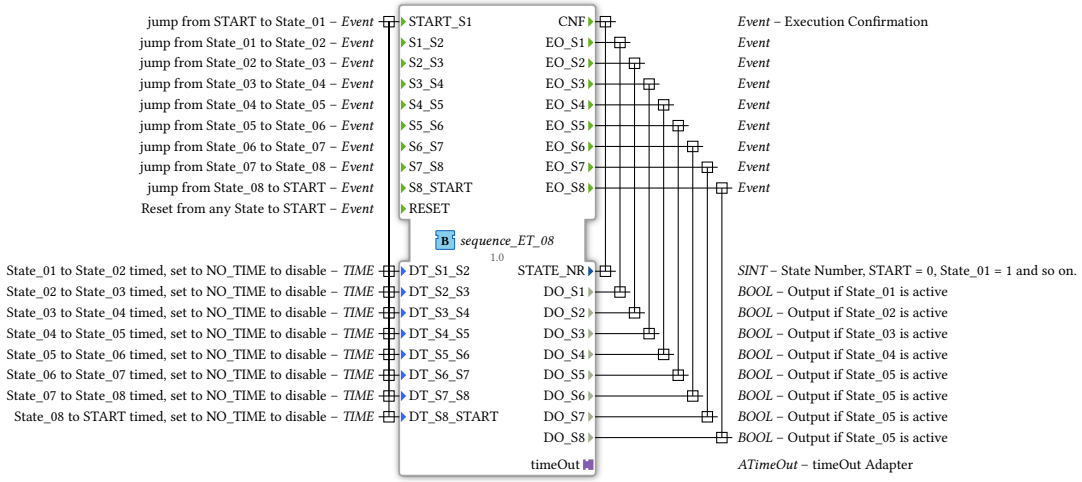
sequence_ET_08



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock `sequence_ET_08` ist ein Sequenzer mit acht Ausgängen. Er ermöglicht die Abarbeitung einer festen Schrittkette, wobei der Übergang von einem Schritt zum nächsten entweder durch ein externes Ereignis oder durch einen einstellbaren Zeitablauf erfolgen kann. Der Baustein ist besonders für Steuerungsaufgaben geeignet, bei denen eine zyklische oder zeitgesteuerte Abfolge von Aktionen benötigt wird.



Schnittstellenstruktur

Ereignis-Eingänge

- `START_S1`: Springt vom `START`-Zustand in den Zustand `State_01`. Überträgt alle Zeitdaten (`DT_S1_S2` bis `DT_S8_START`).
- `S1_S2`: Springt von `State_01` zu `State_02`.
- `S2_S3`: Springt von `State_02` zu `State_03`.

- S3_S4: Springt von State_03 zu State_04.
- S4_S5: Springt von State_04 zu State_05.
- S5_S6: Springt von State_05 zu State_06.
- S6_S7: Springt von State_06 zu State_07.
- S7_S8: Springt von State_07 zu State_08.
- S8_START: Springt von State_08 zurück in den START-Zustand.
- RESET: Setzt den Baustein aus jedem beliebigen Zustand zurück in den START-Zustand.

Ereignis-Ausgänge

- CNF: Ausführungsbestätigung. Wird bei jedem Zustandswechsel gesetzt und überträgt die aktuelle Zustandsnummer (STATE_NR).
- EO_S1 bis EO_S8: Zustands-spezifische Ereignisausgänge. Werden beim Eintritt in den entsprechenden Zustand (State_01 bis State_08) ausgelöst und übertragen den jeweiligen booleschen Datenausgang (DO_S1 bis DO_S8).

Daten-Eingänge

- DT_S1_S2 bis DT_S8_START (Typ TIME): Definieren die Zeitdauer für den automatischen Übergang vom aktuellen zum nächsten Zustand. Wenn der Wert auf NO_TIME gesetzt ist, ist der zeitgesteuerte Übergang für diesen Schritt deaktiviert und ein Ereignis ist erforderlich.

Daten-Ausgänge

- STATE_NR (Typ SINT): Gibt die aktuelle Zustandsnummer aus (START = 0, State_01 = 1, ..., State_08 = 8).
- DO_S1 bis DO_S8 (Typ BOOL): Logische Ausgänge, die TRUE sind, solange sich der Baustein im entsprechenden Zustand (State_01 bis State_08) befindet.

Adapter

- timeOut (Typ iec61499::events::ATimeOut): Ein Zeitgeber-Adapter, der für die Realisierung der zeitgesteuerten Zustandsübergänge verwendet wird.

Funktionsweise

Der Baustein ist als Basic-FB mit einer Execution Control Chart (ECC) implementiert. Er startet im initialen Zustand xSTART. Ein START_S1-Ereignis führt in den ersten aktiven Zustand sState_01. Jeder aktive Zustand (sState_01 bis sState_08) führt beim Eintritt folgende Aktionen aus:

1. Stoppt den laufenden Zeitgeber.
2. Führt den Exit-Algorithmus des vorherigen Zustands aus (schaltet den vorherigen Ausgang aus).
3. Führt den Confirmation-Algorithmus (*_C) aus, der die STATE_NR setzt und die Zeit für den nächsten möglichen automatischen Übergang im timeOut-Adapter konfiguriert.
4. Führt den Entry-Algorithmus (*_E) aus, der den zugehörigen booleschen Ausgang (DO_Sx) aktiviert und das entsprechende Ereignis (EO_Sx) auslöst.
5. Startet den Zeitgeber mit der für diesen Zustand eingestellten Dauer (DT_*).

Der Übergang zum nächsten Zustand kann auf zwei Arten erfolgen:

1. **Ereignisgesteuert:** Durch das entsprechende Sprung-Ereignis (z.B. S1_S2).
2. **Zeitgesteuert:** Durch das TimeOut-Ereignis des Adapters, sofern die Zeitdauer (DT_*) nicht NO_TIME ist.

Das RESET-Ereignis führt in einen dedizierten Reset-Zustand (sRESET), der alle aktiven Ausgänge ausschaltet, bevor er in den inaktiven START-Zustand (sState_00) übergeht.

Technische Besonderheiten

- **Hybride Transitionen:** Jeder Schritt bietet zwei parallele Übergangsbedingungen (Ereignis ODER Zeit), was maximale Flexibilität bietet.
- **Sichere Zustandsbehandlung:** Beim Zustandswechsel wird der Zeitgeber immer gestoppt, und die Ausgänge werden durch definierte Exit-Algorithmen sauber deaktiviert.
- **Konfigurierbare Zeiten:** Die Zeit für jeden Schritt kann individuell eingestellt oder durch NO_TIME deaktiviert werden.
- **Explizite Zustandsrückmeldung:** Der Ausgang STATE_NR erlaubt eine einfache externe Überwachung der aktuellen Schrittposition.

Zustandsübersicht

1. **xSTART:** Initialer, inaktiver Zustand (beim Start des FBs).
2. **sState_00:** Inaktiver `START`-Zustand (nach Reset oder Zyklusende). `STATE_NR = 0`.
3. **sState_01 bis sState_08:** Aktive Zustände der Sequenz. `STATE_NR = 1` bis `8`. Die entsprechenden Ausgänge `DO_S1` bis `DO_S8` sind aktiv.
4. **sRESET:** Temporärer Zustand, der bei einem `RESET`-Ereignis alle aktiven Ausgänge ausschaltet.

Anwendungsszenarien

- Steuerung von zyklischen Prozessen in Verpackungs- oder Fertigungsmaschinen.
- Schrittkette für einen automatisierten Test- oder Kalibrierablauf.
- Steuerung einer Belichtungs- oder Spülsequenz in der Halbleiterfertigung.
- Allgemeine Zustandsautomaten, bei denen Schritte sowohl durch Sensorik (Ereignis) als auch durch feste Zeiten vorangetrieben werden können.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren Sequenzern (z.B. `E_SR` oder `E_CTU` in Reihe) bietet `sequence_ET_08` eine vollständig vordefinierte Schrittkette mit integrierter Zeitsteuerung und dedizierten Ausgängen pro Schritt. Im Gegensatz zu einer individuell programmierten SFC (Sequential Function Chart) ist die Logik fest verdrahtet, was die Anwendung einfacher, aber auch weniger flexibel macht. Bausteine wie `E_DELAY` müssten extern hinzugefügt werden, während hier die Zeitfunktionalität integriert ist.

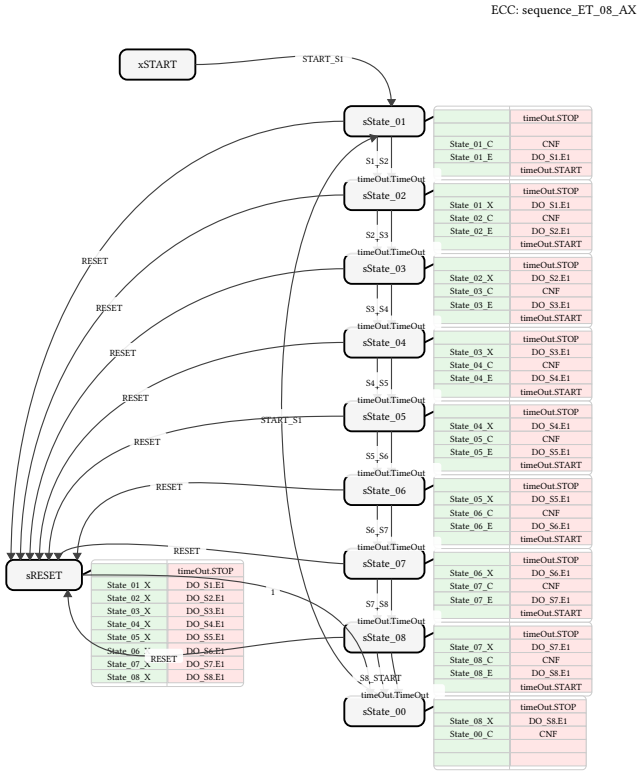
Fazit

Der `sequence_ET_08` ist ein robuster und einfach zu konfigurierender Sequenzer-Baustein für IEC 61499. Seine Stärke liegt in der kombinierten Ereignis- und Zeitsteuerung sowie der klaren, pro Schritt definierten Schnittstelle. Er eignet sich ideal für standardisierte Steuerungsabläufe mit bis zu acht Schritten, bei denen ein hohes Maß an Vorhersagbarkeit und einfache Parametrierung gewünscht ist. Für Prozesse mit variabler Schrittanzahl oder komplexeren Verzweigungen sind flexiblere Lösungen wie Composite-FBs oder eigene SFCs vorzuziehen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de

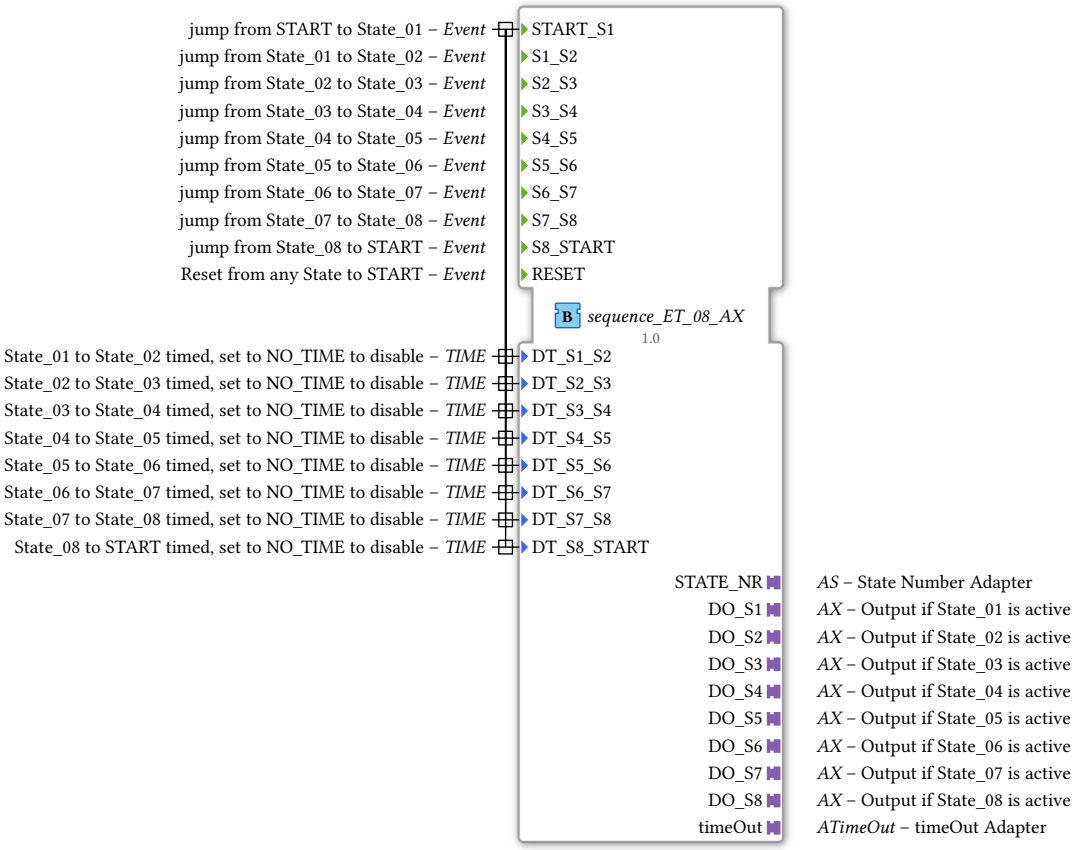
sequence_ET_08_AX



• — ○ • — ○ • — ○ •

Einleitung

`sequence_ET_08_AX` ist eine Variante des `sequence_ET_08`, die zusätzlich Adapter (`AX`) für die Ausgänge verwendet. Er steuert eine Sequenz mit 8 Ausgabezuständen.



Schnittstellenstruktur

Ereignis-Eingänge

- **START_S1**: Startet die Sequenz bei State_01.
- **S1_S2** ... **S7_S8**: Manuelle Übergänge zwischen den Zuständen.
- **S8_START**: Manueller Übergang State_08 -> START.
- **RESET**: Setzt die Sequenz zurück.

Ereignis-Ausgänge

- **CNF**: Bestätigung der Ausführung.

Daten-Eingänge

- **DT_S1_S2** ... **DT_S8_START**: Zeiten für die automatischen Übergänge zwischen den Zuständen.

Daten-Ausgänge

- **STATE_NR** (SINT): Aktuelle Zustandsnummer.

Adapter

- **DO_S1** ... **DO_S8** (adapter::types::unidirectional::AX): Ausgangsadapter für State_01 bis State_08.
- **timeOut** (iec61499::events::ATimeOut): Timer-Adapter.

Funktionsweise

Entspricht `sequence_ET_08`, verwendet jedoch Adapter für die Ausgänge.

Technische Besonderheiten

- Verwendung von `adapter::types::unidirectional::AX`.

Zustandsübersicht

Siehe `sequence_ET_08`.

Anwendungsszenarien

Für 8-stufige Sequenzen mit Adapteranbindung.

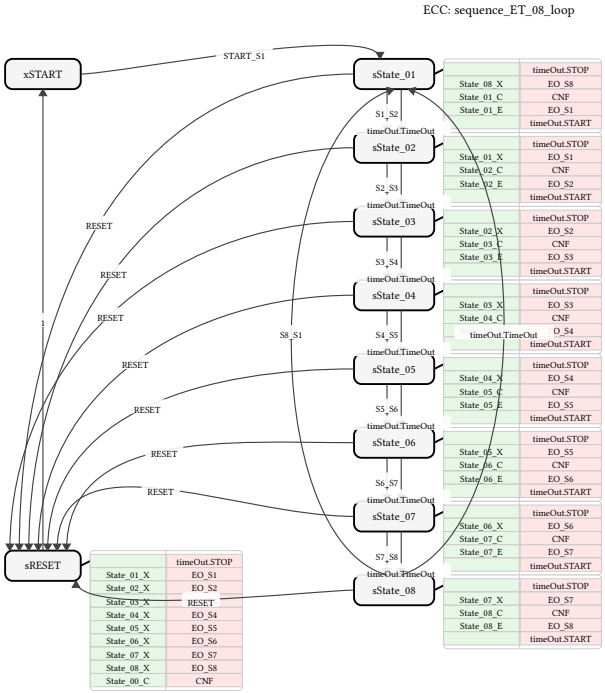
⚖️ Vergleich mit ähnlichen Bausteinen

- **sequence_ET_08**: Standardvariante ohne Adapter.

Fazit

Adapter-Variante des 8-Schritt-Sequenzers.

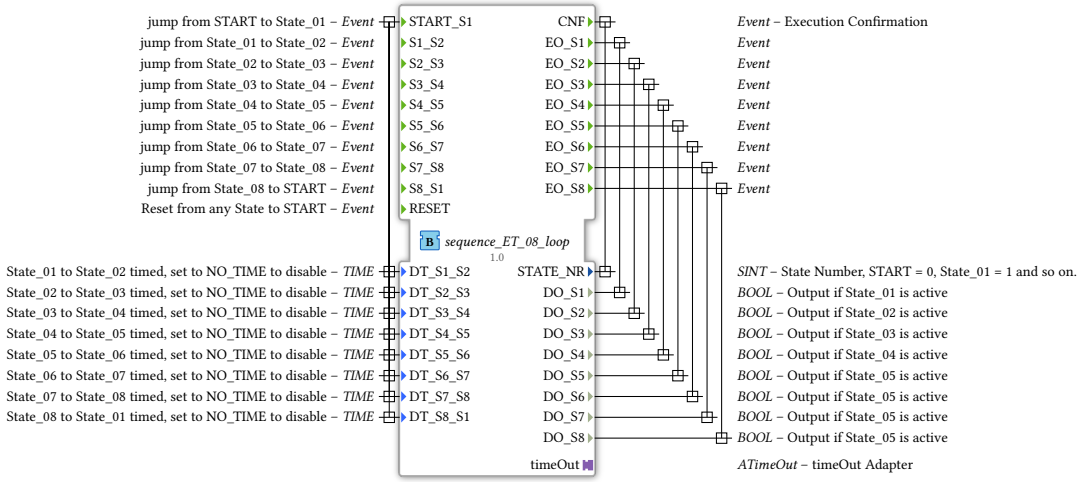
sequence_ET_08_loop



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock `sequence_ET_08_loop` ist ein Sequenzer mit acht Ausgangszuständen, der in einer Schleife arbeitet. Er ermöglicht den Übergang zwischen den Zuständen entweder durch ein externes Ereignis oder zeitgesteuert nach einem einstellbaren Zeitintervall. Der Baustein ist für Steuerungsaufgaben konzipiert, bei denen eine festgelegte Abfolge von Aktionen (repräsentiert durch die Ausgänge `DO_S1` bis `DO_S8`) durchlaufen werden muss. Ein zentrales Merkmal ist die Möglichkeit, jeden Zustandsübergang individuell als ereignis- oder zeitgesteuert zu konfigurieren.



Schnittstellenstruktur

Ereignis-Eingänge

- **`START_S1`**: Startet die Sequenz und führt vom `START`-Zustand in den Zustand `State_01`. Überträgt alle Zeitdaten (`DT_S1_S2` bis `DT_S8_S1`).
- **`S1_S2` bis `S8_S1`**: Führen den Übergang vom entsprechenden Quell- zum nächsten Zielzustand durch (z.B. `S1_S2` von `State_01` nach `State_02`). Diese Übergänge sind nur wirksam, wenn die Zeitsteuerung für diesen Schritt deaktiviert ist.
- **`RESET`**: Setzt die Sequenz aus jedem beliebigen Zustand zurück in den initialen `START`-Zustand.

Ereignis-Ausgänge

- **`CNF`**: Allgemeines Bestätigungsereignis (Execution Confirmation), das bei jedem Zustandswechsel ausgelöst wird. Überträgt die aktuelle Zustandsnummer `STATE_NR`.
- **`EO_S1` bis `EO_S8`**: Zustandsspezifische Ausgangsereignisse, die beim Eintritt in den entsprechenden Zustand (`State_01` bis `State_08`) ausgelöst werden. Jedes Ereignis überträgt den zugehörigen booleschen Datenausgang (`DO_Sx`).

Daten-Eingänge

- **`DT_S1_S2` bis `DT_S8_S1`** (Typ: `TIME`): Definieren die Zeitdauer, für die der jeweilige Zustand aktiv bleibt, bevor ein automatischer, zeitgesteuerter Übergang zum nächsten Zustand erfolgt. Um die Zeitsteuerung für einen bestimmten Übergang zu deaktivieren und auf rein ereignisgesteuerte Übergänge umzuschalten, muss der Wert auf die Konstante `NO_TIME` gesetzt werden. Initial sind alle Zeiteingänge auf `NO_TIME` voreingestellt.

Daten-Ausgänge

- **`STATE_NR`** (Typ: `SINT`): Gibt die Nummer des aktuell aktiven Zustands aus (START = 0, State_01 = 1, ..., State_08 = 8).
- **`DO_S1` bis `DO_S8`** (Typ: `BOOL`): Die physischen Ausgänge der Sequenz. Jeder Ausgang wird auf `TRUE` gesetzt, wenn sich der FB im entsprechenden Zustand befindet (`DO_S1` in `State_01`, usw.), andernfalls ist er `FALSE`.

Adapter

- **`timeOut`** (Typ: `ATimeOut`): Ein eingesteckter (Plug) Timeout-Adapter, der für die Realisierung der zeitgesteuerten Zustandsübergänge genutzt wird. Der FB startet und stoppt den Timer und übergibt ihm die jeweilige Zeiteinstellung (`DT_Sx_Sy`).

Funktionsweise

Der FB ist als Basic Function Block (BFB) mit einer Execution Control Chart (ECC) implementiert. Die ECC besteht aus den Zuständen `xSTART` (Initialzustand), `sState_01` bis `sState_08` (aktive Sequenzzustände) und `sRESET` (Reset-Zustand).

Beim Eintritt in einen aktiven Zustand (z.B. `sState_01`) werden nacheinander folgende Aktionen ausgeführt:

1. Der `timeOut`-Timer wird gestoppt.
2. Der Exit-Algorithmus (`X`) des vorherigen Zustands wird ausgeführt (setzt den vorherigen Ausgang `DO_Sx` auf `FALSE`).
3. Der Confirmation-Algorithmus (`C`) des neuen Zustands wird ausgeführt (setzt `STATE_NR` und konfiguriert den `timeOut`-Adapter mit der für diesen Zustand eingestellten Zeit `DT`).
4. Der Entry-Algorithmus (`E`) des neuen Zustands wird ausgeführt (setzt den zugehörigen Ausgang `DO_Sx` auf `TRUE` und löst das Ereignis `EO_Sx` aus).
5. Der `timeOut`-Timer wird mit der eingestellten Zeit gestartet.

Ein Zustandswechsel kann auf zwei Arten ausgelöst werden:

1. **Ereignisgesteuert:** Durch das entsprechende Ereignis `Sx_Sy`, sofern der Timer für diesen Übergang deaktiviert (`DT = NO_TIME`) ist.
2. **Zeitgesteuert:** Durch das `TimeOut`-Ereignis des Adapters, sofern eine gültige Zeit (`DT != NO_TIME`) eingestellt ist.

Das `RESET`-Ereignis führt in den `sRESET`-Zustand, deaktiviert alle Ausgänge und bestätigt den Zustand `0` (`START`), bevor der FB zurück in `xSTART` wechselt.

Technische Besonderheiten

- **Hybride Triggerung:** Jeder Zustandsübergang kann individuell konfiguriert werden. Dies bietet maximale Flexibilität für Sequenzen, die teils sensor-, teils zeitgesteuert ablaufen.
- **Initiale Konfiguration:** Standardmäßig sind alle Übergänge ereignisgesteuert (`NO_TIME`), was eine explizite Konfiguration der Zeitwerte erforderlich macht, um Zeitsteuerung zu nutzen.
- **Zustandsbestätigung:** Das `CNF`-Ereignis mit `STATE_NR` ermöglicht eine einfache Überwachung und Visualisierung der aktuellen Sequenzposition.
- **Geschlossene Schleife:** Die Sequenz läuft von `State_08` automatisch zurück zu `State_01`, wodurch zyklische Prozesse realisiert werden können.

Zustandsübersicht

Der FB durchläuft folgende Zustände in der ECC:

- **`xSTART`:** Inaktiver Initialzustand. Wartet auf `START_S1`.
- **`sState_01` bis `sState_08`:** Aktive Arbeitszustände. Jeder Zustand aktiviert seinen spezifischen Ausgang (`DO_Sx`) und wartet auf Trigger für den Übergang zum nächsten Zustand.
- **`sRESET`:** Reset-Zustand. Wird bei `RESET`-Ereignis aus jedem Zustand angesprungen, deaktiviert alle Ausgänge und führt zurück zu `xSTART`.

Die Übergangsbedingungen sind in der ECC definiert und kombinieren die Ereignisse `Sx_Sy`, `timeOut.TimeOut` und `RESET`.

Anwendungsszenarien

- **Steuerung von Batch-Prozessen:** Ablaufsteuerung für Misch-, Heiz- oder Abfüllvorgänge, bei denen einzelne Schritte unterschiedlich lange dauern.
- **Verkettete Maschinenabläufe:** Steuerung einer Maschine, deren Arbeitszyklus aus mehreren nacheinander geschalteten Positionen oder Funktionen besteht (z.B. Drehen, Bohren, Fräsen).
- **Teststände:** Automatisierte Durchführung von Testsequenzen, bei denen Prüfungen (ereignisgesteuert) mit Wartezeiten (zeitgesteuert) kombiniert werden.
- **Verpackungsmaschinen:** Steuerung des Zyklus „Produkt zuführen - Verpackung schließen - Etikettieren - Austragen“.

⚖️ Vergleich mit ähnlichen Bausteinen

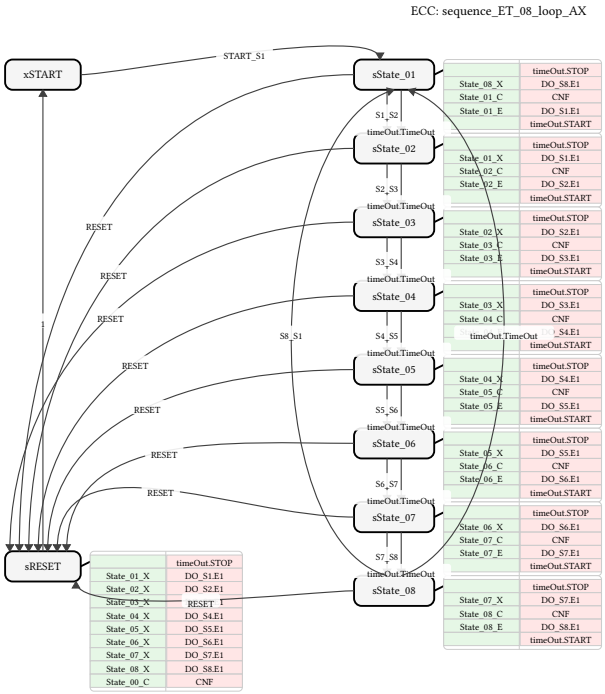
Im Vergleich zu einfacheren Timern oder Flip-Flops bietet dieser FB eine vordefinierte, robuste Zustandsmaschine für 8-Schritt-Sequenzen. Gegenüber einer individuell programmierten SFC (Sequential Function Chart) in einem Service- oder Composite-FB stellt `sequence_ET_08_loop` eine sofort einsatzbereite, getestete und wiederverwendbare Komponente dar, die die Entwicklung beschleunigt und die Fehleranfälligkeit verringert. Bausteine mit fester Zustandsanzahl sind oft performanter und einfacher zu konfigurieren als vollständig frei programmierbare Sequenzer.

Fazit

Der `sequence_ET_08_loop` ist ein leistungsstarker und flexibler Funktionsblock für die Steuerung zyklischer 8-Stufen-Sequenzen. Seine Stärke liegt in der hybriden Triggerung, die für jede Stufe frei wählbar

ist, und der klaren, ereignisbasierten Schnittstelle nach IEC 61499. Durch die integrierte Zeitsteuerung und die direkten booleschen Ausgänge eignet er sich ideal für die direkte Ansteuerung von Aktoren in einem übergeordneten Steuerungsnetzwerk.

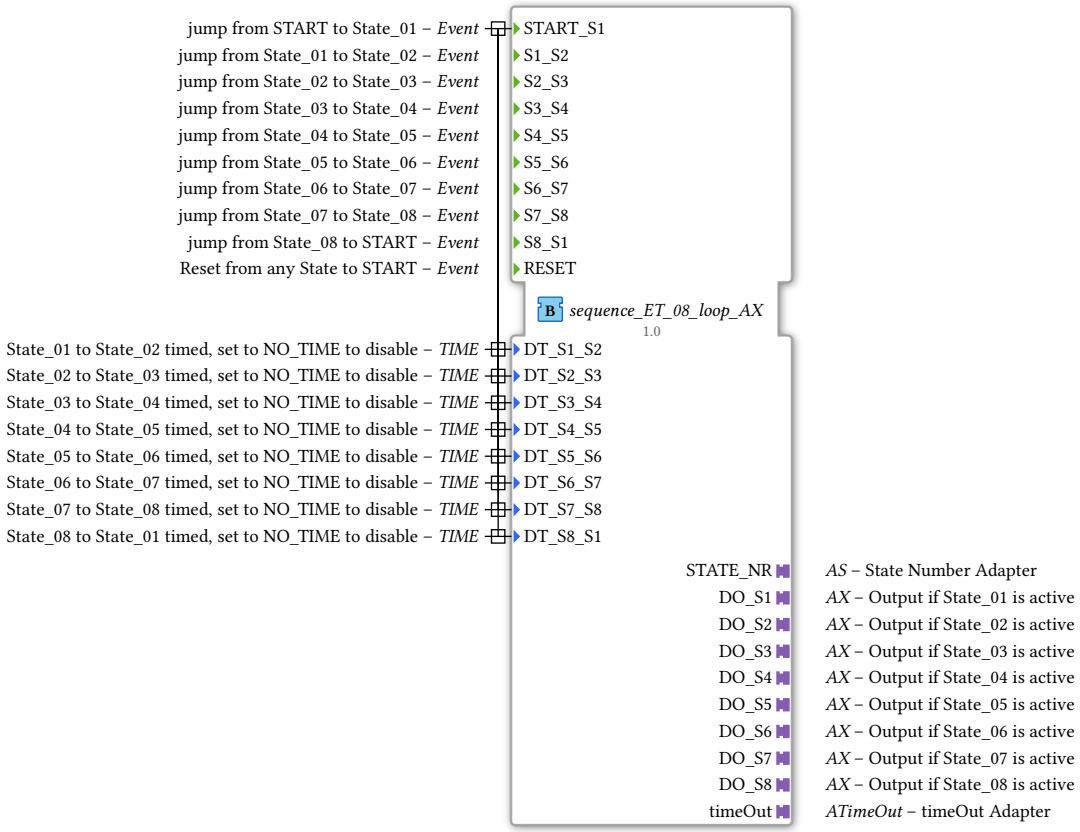
sequence_ET_08_loop_AX



• — ○ • — ○ • — ○ •

Einleitung

`sequence_ET_08_loop_AX` ist eine Variante des `sequence_ET_08_loop`, die zusätzlich Adapter (`AX`) für die Ausgänge verwendet. Er steuert eine zyklische Sequenz mit 8 Ausgabezuständen.



Schnittstellenstruktur

Ereignis-Eingänge

- **START_S1**: Startet die Sequenz bei State_01.
- **S1_S2** ... **S7_S8**: Manuelle Übergänge zwischen den Zuständen.
- **S8_S1**: Manueller Übergang State_08 -> State_01 (Loop).
- **RESET**: Setzt die Sequenz zurück.

Ereignis-Ausgänge

- **CNF**: Bestätigung der Ausführung.

Daten-Eingänge

- **DT_S1_S2** ... **DT_S8_S1**: Zeiten für die automatischen Übergänge zwischen den Zuständen.

Daten-Ausgänge

- **STATE_NR** (SINT): Aktuelle Zustandsnummer.

Adapter

- **DO_S1** ... **DO_S8** (adapter::types::unidirectional::AX): Ausgangsadapter für State_01 bis State_08.
- **timeOut** (iec61499::events::ATimeOut): Timer-Adapter.

Funktionsweise

Entspricht `sequence_ET_08_loop`, verwendet jedoch Adapter für die Ausgänge.

Technische Besonderheiten

- Verwendung von `adapter::types::unidirectional::AX`.

Zustandsübersicht

Siehe `sequence_ET_08_loop`.

Anwendungsszenarien

Für zyklische 8-stufige Prozesse mit Adapteranbindung.

⚖️ **Vergleich mit ähnlichen Bausteinen**

- **sequence_ET_08_loop**: Standardvariante ohne Adapter.

Fazit

Adapter-Variante des 8-Schritt-Loop-Sequenzers.

const

anlagenSequenz

• – ◦ • – ◦ • – ◦ •

Einleitung

Dieses Element der 4diac-IDE definiert eine Sammlung globaler Konstanten für die Steuerung einer Anlagen-Sequenz („AnlagenSequenz_06“). Die Konstanten beschreiben den aktuellen Betriebs- und Störungsstatus sowie die Art von Übergängen (Vorlauf/Nachlauf) für eine Anlage mit sechs Motoren. Sie werden im Paket `logiBUS::utils::sequence::const` bereitgestellt und können von verschiedenen Funktionsbausteinen der Anlagensequenz referenziert werden, um eine einheitliche und lesbare Codierung zu gewährleisten.

Schnittstellenstruktur

Ereignis-Eingänge

Keine (dies ist ein Global-Constants-Baustein, keine Ereignisse vorhanden).

Ereignis-Ausgänge

Keine (dies ist ein Global-Constants-Baustein, keine Ereignisse vorhanden).

Daten-Eingänge

Keine (die Werte sind statisch definiert und können nicht zur Laufzeit geändert werden).

Daten-Ausgänge

Die folgenden globalen Konstanten sind als Daten-Ausgänge verfügbar (sie können in anderen Bausteinen gelesen werden):

Konstante	Datentyp	Wert	Beschreibung
`STATUS_BETRIEB_AUS`	SINT	SINT#0	Anlage steht
`STATUS_BETRIEB_HOCHFAHREN`	SINT	SINT#1	Vorlauf-Kette aktiv
`STATUS_BETRIEB_LAEUFT`	SINT	SINT#2	alle 6 Motoren laufen
`STATUS_BETRIEB_HERUNTERFAHREN`	SINT	SINT#3	Nachlauf-Kette aktiv
`STATUS_STOERUNG_KEINE`	SINT	SINT#0	keine aktive Störung
`STATUS_STOERUNG_AKTIV`	SINT	SINT#4	Störung aktiv, verriegelt bis EIN
`TRANSIT_ART_KEINE`	SINT	SINT#0	kein Motor aktuell im Vor-/Nachlauf (S_AUS, S_LAEUFT)
`TRANSIT_ART_VORLAUF`	SINT	SINT#1	TRANSIT_MOTOR startet nach Ablauf der aktuellen Zeit
`TRANSIT_ART_NACHLAUF`	SINT	SINT#2	TRANSIT_MOTOR stoppt nach Ablauf der aktuellen Zeit

Adapter

Keine (es werden keine Adapter verwendet).

Funktionsweise

Die globalen Konstanten dienen als zentrale Definitionsquelle für Zustandswerte in der Anlagensteuerung. Sie werden in der 4diac-IDE als `GlobalConstants` definiert und können in allen FB-Typen der Anwendung über ihren symbolischen Namen verwendet werden. Dadurch wird die Verwendung von „magic numbers“ vermieden und die Wartbarkeit des Steuerungscodes erhöht.

Die Konstanten sind in zwei Gruppen aufgeteilt:

- **Betriebszustände** (`STATUS_BETRIEB_*`): beschreiben die Phasen der Anlage (aus, hochfahren, laufen, herunterfahren).
- **Störungszustände** (`STATUS_STOERUNG_*`): geben an, ob eine Störung vorliegt.
- **Transitionsarten** (`TRANSIT_ART_*`): kennzeichnen die Art eines Motor-Übergangs (kein Übergang, Vorlauf, Nachlauf).

Durch die getrennten Konstanten für Betrieb und Störung wird eine klare Trennung der Zustandslogik ermöglicht.

Technische Besonderheiten

- Die Konstanten sind als `SINT` (Small Integer) deklariert und mit dem Präfix `SINT#` initialisiert.
- Sie sind als `CONSTANT` in `VAR_GLOBAL CONSTANT` definiert, d.h. ihre Werte können während der Laufzeit nicht verändert werden.
- Der Baustein ist im Paket `logiBUS::utils::sequence::const` enthalten und hat die Versionsinformation `1.0` vom 12.08.2026.
- Die Konstanten sind direkt über den qualifizierten Namen ansprechbar, z.B. `anlagenSequenz.STATUS_BETRIEB_LAEUFT`.
- Eine Besonderheit ist, dass `STATUS_STOERUNG_AKTIV` den Wert `4` hat, was nicht in die numerische Reihenfolge der Betriebszustände passt. Dies ermöglicht eine unabhängige Behandlung von Störungen.

Zustandsübersicht

Da es sich um eine Konstantendefinition handelt, gibt es keine internen Zustände. Die definierten Werte repräsentieren jedoch die möglichen Zustände der Anlage:

- **Betriebszustände**: `STATUS_BETRIEB_AUS` (0), `STATUS_BETRIEB_HOCHFAHREN` (1), `STATUS_BETRIEB_LAEUFT` (2), `STATUS_BETRIEB_HERUNTERFAHREN` (3).
- **Störungszustand**: `STATUS_STOERUNG_KEINE` (0), `STATUS_STOERUNG_AKTIV` (4).
- **Transitionsarten**: `TRANSIT_ART_KEINE` (0), `TRANSIT_ART_VORLAUF` (1), `TRANSIT_ART_NACHLAUF` (2).

Diese Zustände werden typischerweise in einer übergeordneten Anlagen-Sequenz-Steuerung verwendet, um den aktuellen Betriebszustand zu kennzeichnen und Übergänge zu steuern.

Anwendungsszenarien

Die Konstanten werden in Funktionsbausteinen verwendet, die die Anlagen-Sequenz realisieren, z.B.:

- **Hochfahren**: Der Baustein setzt den Betriebszustand auf `STATUS_BETRIEB_HOCHFAHREN` und steuert die Motoren nacheinander an. Während dieser Phase ist die Transitionsart `TRANSIT_ART_VORLAUF`.
- **Nachlauf**: Beim Herunterfahren wird `STATUS_BETRIEB_HERUNTERFAHREN` und `TRANSIT_ART_NACHLAUF` verwendet.
- **Störungsbehandlung**: Tritt eine Störung auf, wird `STATUS_STOERUNG_AKTIV` gesetzt, und die Anlage wird bis zum nächsten EIN-Signal verriegelt.
- **Zustandsanzeige**: Über die Konstanten können Statuswerte einfach an HMI- oder Diagnosesysteme weitergegeben werden.

Vergleich mit ähnlichen Bausteinen

Da es sich um eine reine Konstantendefinition handelt, gibt es keinen direkten Funktionsblock mit gleicher Aufgabe. Andere Bausteine könnten ähnliche Konstantenbereiche über `LocalConstants` oder `Directly Represented`-Variablen definieren, aber die Verwendung eines globalen Konstantenblocks wie `anlagenSequenz` bietet den Vorteil, dass alle zugehörigen Werte an einem zentralen Ort gepflegt werden können und für das gesamte Projekt einheitlich verfügbar sind.

Fazit

Das `GlobalConstants`-Element `anlagenSequenz` stellt eine klar strukturierte Sammlung von Konstanten für die Zustands- und Übergangssteuerung einer Anlagen-Sequenz bereit. Es verbessert die Lesbarkeit und Wartbarkeit des IEC-61499-Codes, indem es symbolische Namen anstelle von Zahlenwerten

verwendet. Die klare Trennung zwischen Betriebs- und Störungszuständen sowie die Definition von Transitionsarten ermöglicht eine robuste Implementierung der Steuerungslogik. Dieses Modul ist ein sinnvoller Bestandteil einer größeren 4diac-Lösung für die industrielle Anlagensteuerung.

motorNummer

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein `motorNummer` ist eine globale Konstanten-Definition (GlobalConstants) im 4diac-IDE-Umfeld. Er stellt symbolische Namen für die Identifikation verschiedener Motoren in einer Anlagensteuerung bereit. Die Werte sind vom Typ `SINT` (signed short integer) und decken den Bereich von 0 bis 6 ab, wobei 0 für „kein Motor“ steht und 1 bis 6 den Motoren M1 bis M6 zugeordnet sind. Diese Konstanten werden typischerweise in Sequenzsteuerungen oder anderen Anwendungen verwendet, um Motor-IDs lesbar und wartbar zu definieren.

Die Definition ist unter dem Paket `logiBUS::utils::sequence::const` abgelegt und wird im Rahmen des Projekts „AnlagenSequenz“ genutzt, wie aus dem Kommentar hervorgeht.

Schnittstellenstruktur

Da es sich um eine reine Konstantendefinition handelt, besitzt der Baustein **keine** Ein-/Ausgangsschnittstellen im Sinne von Ereignissen, Daten oder Adaptern. Die Schnittstellenstruktur entfällt vollständig.

Ereignis-Eingänge

Nicht vorhanden.

Ereignis-Ausgänge

Nicht vorhanden.

Daten-Eingänge

Nicht vorhanden.

Daten-Ausgänge

Nicht vorhanden.

Adapter

Nicht vorhanden.

Funktionsweise

Der Baustein `motorNummer` definiert sieben globale Konstanten, die innerhalb des 4diac-Projekts (z. B. in ST- oder FBD-Programmen) referenziert werden können. Durch die Verwendung symbolischer Namen anstelle von magischen Zahlen wird die Lesbarkeit und Wartbarkeit des Codes erhöht. Beispielsweise kann in einer Sequenz ein Übergang `TRANSIT_MOTOR` auf die Konstante `MOTOR_M3` verweisen, um den dritten Motor anzusteuern.

Die Konstanten sind als `CONSTANT` deklariert und können zur Laufzeit nicht verändert werden. Sie sind global gültig, sobald der Baustein in das Projekt eingebunden ist.

Technische Besonderheiten

- **Typ:** `SINT` – vorzeichenbehafteter 8-Bit-Integer (Wertebereich −128 bis 127).
- **Initialwerte:** Die Konstanten sind mit expliziten Werten initialisiert (z. B. `SINT#0`, `SINT#1`, …).
- **Namensgebung:** Die Bezeichnungen folgen einem einheitlichen Muster: `MOTOR_KEINER`, `MOTOR_M1` bis `MOTOR_M6`.
- **Lizenz:** Die Software ist unter der Eclipse Public License 2.0 (EPL-2.0) veröffentlicht, wie im Identifikationsblock angegeben.
- **Autor:** Franz Höpfinger, Firma HR Agrartechnik GmbH, Datum: 2026-08-25.
- **Paket:** `logiBUS::utils::sequence::const` – gruppiert die Konstanten thematisch.

Zustandsübersicht

Ein Zustandsübersicht ist für einen Konstantendefinitions-Baustein nicht anwendbar, da keine endlichen Zustandsautomaten oder dynamischen Verhalten existieren.

Anwendungsszenarien

- **Steuerung einer Anlagensequenz:** Die Konstanten werden verwendet, um in Übergangsbedingungen (z. B. `TRANSIT_MOTOR`) die aktive Motor-ID zu vergleichen.
- **Parametrierung:** Anstelle von Zahlenwerten in Funktionsblöcken kann auf die symbolischen Namen zugegriffen werden, was Fehler bei der Eingabe vermeidet.
- **Projektweite Wiederverwendung:** Da die Konstanten global sind, können sie in verschiedenen FB-Instanzen und Programmteilen ohne erneute Definition verwendet werden.

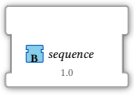
Vergleich mit ähnlichen Bausteinen

In IEC 61499 gibt es keine standardisierten GlobalConstants-Bausteine; die Definition ist eine Erweiterung der 4diac-IDE. Alternativ könnten Konstanten als lokale Variablen in jedem Baustein deklariert werden, was jedoch die Konsistenz erschwert. Die zentrale Definition in einem eigenen Baustein wie `motorNummer` bietet eine klare und zentrale Wartungsmöglichkeit. Andere Bausteine könnten ähnliche Konstanten für andere Geräte (z. B. Pumpen, Ventile) definieren, sodass das Muster wiederverwendbar ist.

Fazit

`motorNummer` ist ein einfacher, aber effektiver Baustein zur zentralen Verwaltung von Motor-IDs als globale Konstanten. Er trägt zur Codequalität bei, indem er magische Zahlen durch sprechende Namen ersetzt und so die Lesbarkeit und Wartbarkeit der Steuerungslogik verbessert. Die klare Struktur und die Verwendung von `SINT` machen ihn für die vorgesehene Anwendung in der Anlagensteuerung geeignet. Aufgrund fehlender Schnittstellen ist er prototypisch für Konstantendefinitionen in 4diac und kann als Vorlage für ähnliche Konstantenblöcke dienen.

sequence



• – ◦ • – ◦ • – ◦ •

Einleitung

Die globale Konstantendatei `sequence` definiert eine Reihe von vordefinierten Konstanten, die in Verbindung mit Sequenz-Funktionsblöcken verwendet werden können. Sie stellt symbolische Namen für Zustände und einen speziellen Zeitwert bereit, um die Lesbarkeit und Wartbarkeit von IEC 61499-Applikationen zu erhöhen. Die Konstanten sind im Paket `logiBUS::utils::sequence::const` organisiert.

Schnittstellenstruktur

Globale Konstanten besitzen keine Ereignis- oder Datenschnittstellen im herkömmlichen Sinne eines Funktionsblocks. Stattdessen werden sie als benannte Werte im gesamten Projekt zur Verfügung gestellt. Ihre Struktur ist eine Sammlung von Variablendeklarationen mit festen Initialwerten.

Daten-Eingänge

Nicht zutreffend. Globale Konstanten sind keine Eingänge, sondern fest definierte Werte.

Daten-Ausgänge

Nicht zutreffend. Globale Konstanten sind keine Ausgänge, sondern fest definierte Werte.

Adapter

Nicht zutreffend.

Funktionsweise

Die Datei definiert Konstanten, die zur Kompilierzeit durch ihre Werte ersetzt werden. Sie können in jedem Funktionsblock innerhalb des Projekts referenziert werden, sofern der entsprechende Namensraum (`logiBUS::utils::sequence::const`) importiert oder der vollqualifizierte Name verwendet wird. Die Verwendung dieser Konstanten anstelle von direkten numerischen Werten („Magic Numbers“) im Code macht die Logik klarer und änderungsresistenter.

Technische Besonderheiten

- **Datentypen:** Die Konstanten verwenden die elementaren IEC 61131-3-Datentypen `TIME` und `SINT` (Short Integer).
- **Besonderer Wert `NO_TIME`:** Die Konstante `NO_TIME` ist mit dem sehr großen Wert `TIME#49d17h2m47s295ms` initialisiert. Dieser Wert dient typischerweise als Platzhalter oder Initialwert, um einen „undefinierten“ oder „nicht gesetzten“ Zeitpunkt zu kennzeichnen, da er weit außerhalb des Bereichs normaler Steuerungszeiten liegt.
- **Zustandskonstanten:** Die Konstanten `State_00` bis `State_08` repräsentieren ganzzahlige Zustände (0-8) und sind vom Typ `SINT`. Sie sind für die Verwendung in Zustandsautomaten oder Schrittketten vorgesehen.

Zustandsübersicht

Nicht zutreffend. Dies ist eine Konstantensammlung und enthält keinen Zustandsautomaten.

Anwendungsszenarien

1. **Zustandsautomaten / Sequenzer:** Die `State_XX`-Konstanten können als Rückgabewerte oder für Vergleiche in Schrittketten-Funktionsblöcken (z.B. `E_SR`, `E_CTU` in Verbindung mit Logik) verwendet werden, um den aktuellen Schritt lesbar zu identifizieren.
2. **Zeitgeber-Initialisierung:** Die Konstante `NO_TIME` kann verwendet werden, um Zeitgeber-Funktionsblöcken einen initialen „ungültigen“ Wert zuzuweisen, bevor eine echte Zeitdauer gesetzt wird.
3. **Code-Verständlichkeit:** Ersetzen von Zahlen wie `SINT#5` im Code durch `State_05` verbessert die Lesbarkeit und zeigt die semantische Bedeutung des Wertes an.

⚖️ Vergleich mit ähnlichen Bausteinen

- **ENUM-Datentyp:** Ein benutzerdefinierter Aufzählungstyp (ENUM) könnte eine ähnliche Rolle wie die `State_XX`-Konstanten übernehmen und bietet oft eine strengere Typprüfung. Globale Konstanten sind jedoch einfacher zu definieren und in 4diac zu verwenden.
- **Direkte Werteingabe:** Der Hauptvorteil gegenüber der direkten Eingabe von Werten (z.B. `3`) ist die zentrale Definition. Eine Änderung des Wertes muss nur an einer Stelle (hier in der Konstantendatei) vorgenommen werden und wirkt sich auf alle Verwendungen aus.

Fazit

Die globale Konstantendatei `sequence` ist ein nützliches Hilfsmittel zur Strukturierung von Steuerungsanwendungen. Durch die Bereitstellung von sprechenden Namen für Zustände und einen speziellen Zeitwert trägt sie wesentlich zur Erzeugung von klarem, wartbarem und robustem Code bei. Ihre Verwendung wird insbesondere bei der Implementierung von Schrittketten und zustandsbasierten Steuerungen empfohlen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

zaehlstand

• – ◦ • – ◦ • – ◦ •

Einleitung

Der globale Konstantenblock `zaehlstand` definiert sieben symbolische SINT-Konstanten, die den aktuellen Zählstand laufender Motoren in einer Anlagensequenz repräsentieren. Die Werte reichen von 0 bis 6 und entsprechen den Zuständen von „kein Motor aktiv“ bis „alle Motoren aktiv“. Diese Konstanten dienen als einheitliche Referenz für Zustandsprüfungen und Vergleiche in der Steuerungslogik und erhöhen die Lesbarkeit sowie Wartbarkeit des IEC-61499-Codes.

Schnittstellenstruktur

Der Baustein stellt keine klassischen Schnittstellen im Sinne von Funktionsbausteinen (Ereignis-Eingänge, Ereignis-Ausgänge, Daten-Eingänge, Daten-Ausgänge oder Adapter) bereit. Stattdessen definiert er globale Konstanten, die im gesamten Projekt als unveränderliche Werte verwendet werden können. Die folgende Tabelle listet die verfügbaren Konstanten auf:

Konstantenbezeichner	Typ	Wert	Bedeutung
ZAEHLSTAND_0	SINT	SINT#0	0 Motoren laufen (S_AUS)
ZAEHLSTAND_1	SINT	SINT#1	1 Motor läuft
ZAEHLSTAND_2	SINT	SINT#2	2 Motoren laufen
ZAEHLSTAND_3	SINT	SINT#3	3 Motoren laufen
ZAEHLSTAND_4	SINT	SINT#4	4 Motoren laufen
ZAEHLSTAND_5	SINT	SINT#5	5 Motoren laufen
ZAEHLSTAND_6	SINT	SINT#6	6 Motoren laufen (S_LAEUFT, alle)

Funktionsweise

Die Konstanten werden als globale Definitionen bereitgestellt und können in beliebigen Algorithmen oder Zustandsautomaten referenziert werden. Sie stellen feste, unveränderliche Werte dar, die den jeweiligen Motorzählstand eindeutig kennzeichnen. Die Verwendung symbolischer Namen anstelle magischer Zahlen verbessert die Codequalität und reduziert Fehler bei der Programmierung.

Technische Besonderheiten

- **Gültigkeitsbereich**: Global, d.h. im gesamten Projekt verfügbar.
- **Datentyp**: Alle Konstanten sind vom Typ `SINT` (signed short integer) und belegen 8 Bit.
- **Initialwerte**: Die Werte sind als `SINT#0` bis `SINT#6` festgelegt und können nicht verändert werden.
- **Paketstruktur**: Die Konstanten sind im Paket `logiBUS::utils::sequence::const` definiert und folgen der IEC-61499-Spezifikation (Standard 61499-1).
- **Lizenz**: Die Definitionen stehen unter der Eclipse Public License 2.0 (EPL-2.0).

Zustandsübersicht

Obwohl der Block selbst keine Zustandsmaschine implementiert, entsprechen die sieben definierten Konstanten den möglichen Zuständen der Motorenanzahl:

- `ZAEHLSTAND_0` – Zustand „alle Motoren aus“ (S_AUS)
- `ZAEHLSTAND_1` bis `ZAEHLSTAND_5` – Zwischenzustände mit 1 bis 5 aktiven Motoren
- `ZAEHLSTAND_6` – Zustand „alle Motoren laufen“ (S_LAEUFT)

Diese Zustände können in einer übergeordneten Steuerungslogik verwendet werden, um den aktuellen Betriebszustand der Anlage zu bestimmen.

Anwendungsszenarien

- **Überwachung der Motoranzahl**: Ein Funktionsbaustein, der die Anzahl laufender Motoren zählt, kann die ermittelten Werte mit den Konstanten vergleichen und entsprechende Aktionen auslösen.
- **Zustandssteuerung**: In einer Anlagensteuerung dienen die Konstanten als Referenz für Zustandsübergänge, z.B. für die Aktivierung von Sicherheitsfunktionen oder Alarmen bei bestimmten Motorzahlen.
- **Vereinheitlichung**: Durch die zentrale Definition werden konsistente Bezeichnungen über verschiedene Module hinweg sichergestellt und Änderungen an Werten können zentral vorgenommen werden.

Vergleich mit ähnlichen Bausteinen

Anders als ein Funktionsbaustein, der dynamische Daten verarbeitet, bietet dieser Block keinerlei Verarbeitungslogik. Er stellt ausschließlich statische Konstanten bereit. Vergleicht man ihn mit einem Aufzählungstyp (ENUM), so geht er über diesen hinaus, da er explizite SINT-Werte definiert und so auch als Eingabe für systemnahe Funktionen verwendet werden kann. Im Gegensatz zu einer Variablen, die zur Laufzeit geändert werden kann, sind diese Werte final und unveränderlich.

Fazit

Der globale Konstantenblock `zaehlstand` ist ein einfaches, aber nützliches Hilfsmittel zur Definition von Motoranzahlzuständen in Anlagensequenzen. Er verbessert die Lesbarkeit des Steuercodes, vermeidet magische Zahlen und bietet eine zentrale und konsistente Basis für Zustandsprüfungen. Die klare Struktur und die IEC-61499-Konformität machen ihn zu einem sinnvollen Bestandteil der `logiBUS::utils`-Bibliothek.

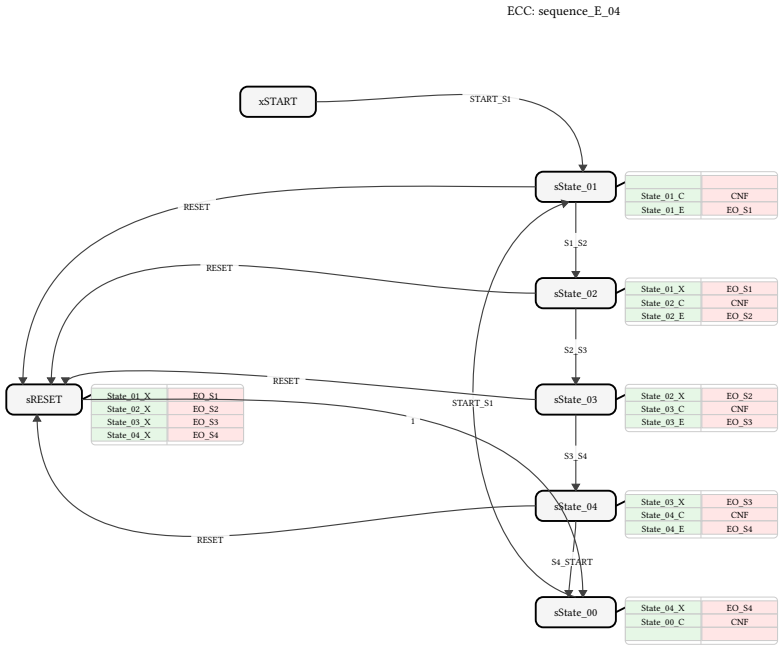
event

🎧 Podcast

- E_TOF vs. FB_TOF: Der Event-Timer, der nicht zyklisch tickt – Revolution für Automatisierungssysteme?
- EventFBs nach IEC 61499: Legosteine der Automatisierung – So funktionieren Ereignis-Funktionsbausteine
- IEC 61499: Revolution der Automatisierung – Event-gesteuerte FBs und verteilte Systeme erklärt
- E_REND: Event Synchronization in IEC 61499
- The Unstoppable Counter: Why IEC 61499's ECTU Guarantees Safe, Event-Driven Control (and Never Overflows)

---

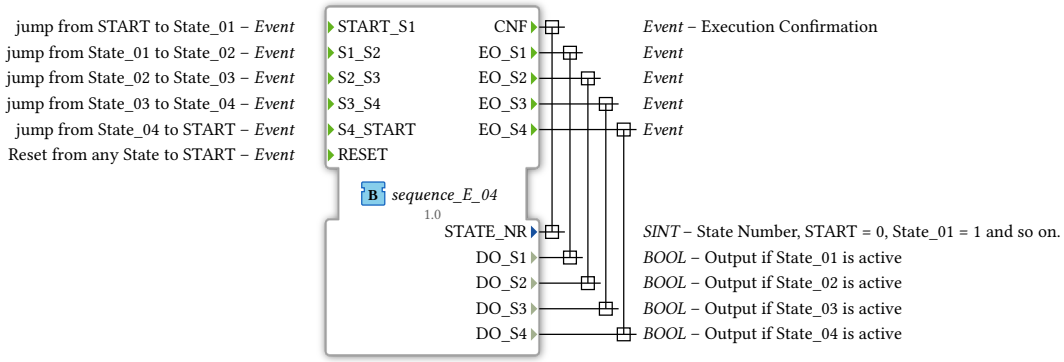
sequence_E_04



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock `sequence_E_04` ist ein Sequenzer, der eine lineare Abfolge von vier Zuständen (State_01 bis State_04) mit einem definierten Startzustand (START) und einem Endzustand (State_00) realisiert. Der Übergang zwischen den einzelnen Zuständen erfolgt ausschließlich durch externe Ereignisse. Der Baustein ist für Steuerungsaufgaben konzipiert, bei denen eine schrittweise, ereignisgesteuerte Abarbeitung erforderlich ist, wie z.B. in einfachen Prozess- oder Montageabläufen.



Schnittstellenstruktur

Ereignis-Eingänge

- **START_S1**: Wechselt vom Zustand START oder State_00 in den Zustand State_01.
- **S1_S2**: Wechselt vom Zustand State_01 in den Zustand State_02.
- **S2_S3**: Wechselt vom Zustand State_02 in den Zustand State_03.
- **S3_S4**: Wechselt vom Zustand State_03 in den Zustand State_04.
- **S4_START**: Wechselt vom Zustand State_04 in den Zustand State_00.
- **RESET**: Setzt den Baustein von jedem beliebigen Zustand (State_01 bis State_04) in den Zustand State_00 zurück.

Ereignis-Ausgänge

- **CNF**: Bestätigungsereignis, das bei jedem Zustandswechsel ausgelöst wird. Es liefert die aktuelle Zustandsnummer über `STATE_NR`.
- **EO_S1**: Wird beim Eintritt in den Zustand State_01 ausgelöst und liefert den Wert von `DO_S1` (TRUE).
- **EO_S2**: Wird beim Eintritt in den Zustand State_02 ausgelöst und liefert den Wert von `DO_S2` (TRUE).
- **EO_S3**: Wird beim Eintritt in den Zustand State_03 ausgelöst und liefert den Wert von `DO_S3` (TRUE).
- **EO_S4**: Wird beim Eintritt in den Zustand State_04 ausgelöst und liefert den Wert von `DO_S4` (TRUE).

Daten-Eingänge

- Keine vorhanden.

Daten-Ausgänge

- **STATE_NR** (SINT): Gibt die Nummer des aktuellen Zustands aus. Die Kodierung lautet: START = 0, State_01 = 1, State_02 = 2, State_03 = 3, State_04 = 4.
- **DO_S1** (BOOL): Ist TRUE, wenn der Zustand State_01 aktiv ist.
- **DO_S2** (BOOL): Ist TRUE, wenn der Zustand State_02 aktiv ist.
- **DO_S3** (BOOL): Ist TRUE, wenn der Zustand State_03 aktiv ist.
- **DO_S4** (BOOL): Ist TRUE, wenn der Zustand State_04 aktiv ist.

Adapter

- Keine vorhanden.

Funktionsweise

Der `sequence_E_04` ist als Basic Function Block (BFB) implementiert und besitzt eine Execution Control Chart (ECC). Die ECC definiert die Zustände und die ereignisgesteuerten Übergänge zwischen ihnen. Bei jedem Zustandswechsel werden spezifische Algorithmen ausgeführt:

1. **Exit-Algorithmus (X)**: Wird beim Verlassen eines Zustands ausgeführt, um den zugehörigen Datenausgang (`DO_Sx`) auf FALSE zu setzen.
2. **Entry-Algorithmus (E)**: Wird beim Betreten eines Zustands ausgeführt, um den zugehörigen Datenausgang (`DO_Sx`) auf TRUE zu setzen und das entsprechende Ereignis (`EO_Sx`) auszulösen.
3. **Confirmation-Algorithmus (C)**: Wird in jedem Zustand (außer RESET) ausgeführt, um die Zustandsnummer `STATE_NR` zu aktualisieren und das Bestätigungsereignis `CNF` auszulösen.

Ein `RESET`-Ereignis führt alle notwendigen Exit-Algorithmen der aktiven Zustände aus und wechselt über einen Zwischenzustand (`sRESET`) in den Endzustand `State_00`.

Technische Besonderheiten

- **Ereignisbasierte Transition**: Zustandswechsel sind ausschließlich durch externe Ereignisse möglich. Es gibt keine zeit- oder datengesteuerten Übergänge.
- **Explizite Zustandskodierung**: Die Zustandsnummern sind als Konstanten aus der Bibliothek `sequence` definiert, was die Wiederverwendung und Lesbarkeit des Codes verbessert.
- **Sauberer Reset**: Der Reset-Vorgang deaktiviert alle aktiven Ausgänge, bevor der Endzustand erreicht wird, um ein klares und definiertes Systemverhalten zu gewährleisten.
- **Initialzustand**: Der FB startet im Zustand `xSTART`. Der erste Übergang in die operative Sequenz erfolgt durch das Ereignis `START_S1`.

Zustandsübersicht

1. **xSTART**: Initialer Ruhezustand.
2. **sState_01**: Erster aktiver Schritt. `DO_S1` = TRUE.
3. **sState_02**: Zweiter aktiver Schritt. `DO_S2` = TRUE.
4. **sState_03**: Dritter aktiver Schritt. `DO_S3` = TRUE.
5. **sState_04**: Vierter aktiver Schritt. `DO_S4` = TRUE.
6. **sState_00**: Finaler Ruhezustand nach Abschluss der Sequenz. Alle `DO_Sx` = FALSE.
7. **sRESET**: Zwischenzustand während eines Reset-Vorgangs. Führt die Deaktivierung aller Ausgänge durch.

Anwendungsszenarien

- **Schrittkettensteuerungen**: Steuerung von Maschinen oder Anlagen, die einen festen, schrittweisen Arbeitsablauf haben (z.B. Pick-and-Place, Füllanlagen).
- **Taktung von Prozessen**: Synchronisation von Teilprozessen, bei denen jeder Schritt manuell oder durch ein Sensorsignal freigegeben wird.
- **Manuelle Bedienabläufe**: Umsetzung von geführten Bediensequenzen, bei denen der Operator jeden Schritt bestätigen muss.

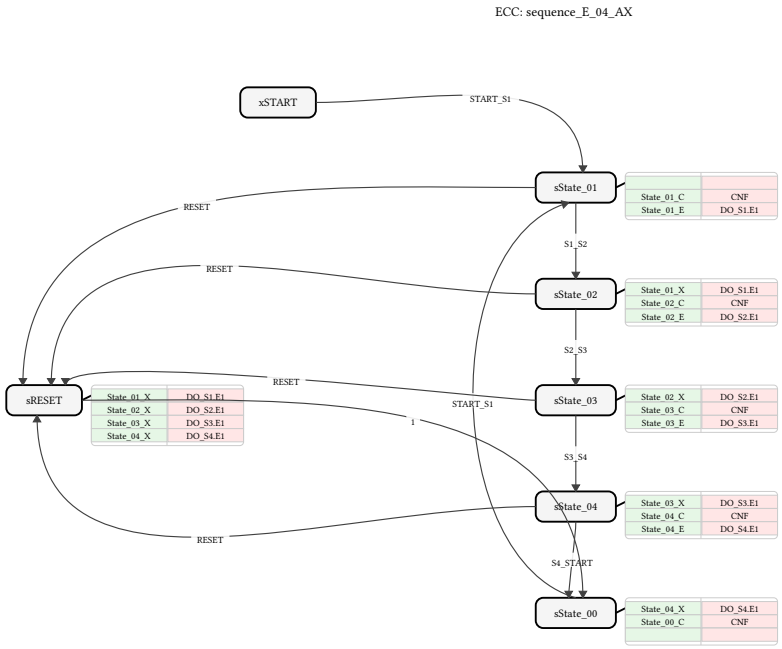
⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu einem `E_CYCLE`- oder `E_DELAY`-Baustein bietet `sequence_E_04` keine zeitgesteuerten, sondern rein ereignisgesteuerte Übergänge. Im Vergleich zu einem generischen `E_SR` (Flip-Flop) oder einer Kombination aus `E_D_FF` (D-Flip-Flop) realisiert dieser FB eine vordefinierte Zustandsmaschine mit mehreren Schritten und einer klaren Reset-Logik. Er ist spezialisierter und strukturierter als eine ad-hoc Implementierung mit mehreren Einzelbausteinen.

Fazit

Der `sequence_E_04` ist ein robuster und einfach zu verwendender Sequenzer für IEC 61499. Seine klare, ereignisgesteuerte Schnittstelle und die explizite Zustandsverwaltung machen ihn ideal für Anwendungen, bei denen ein deterministischer, schrittweiser Ablauf erforderlich ist. Die integrierte Reset-Funktionalität und die Zustandsrückmeldung über `CNF` und `STATE_NR` unterstützen eine sichere und gut überwachbare Integration in übergeordnete Steuerungen.

sequence_E_04_AX

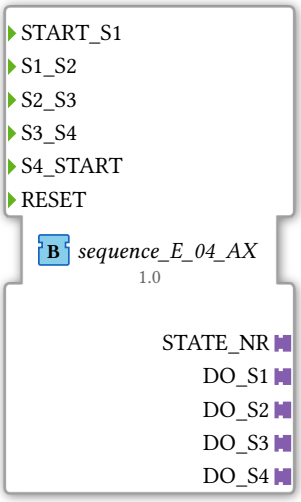


• – ○ • – ○ • – ○ •

Einleitung

Der `sequence_E_04_AX` ist eine Variante des `sequence_E_04`, die zusätzlich Adapter (`AX`) für die Ausgänge verwendet. Er steuert eine rein ereignisgesteuerte Sequenz mit 4 Ausgabezuständen.

jump from START to State_01 – *Event*
jump from State_01 to State_02 – *Event*
jump from State_02 to State_03 – *Event*
jump from State_03 to State_04 – *Event*
jump from State_04 to START – *Event*
Reset from any State to START – *Event*



AS – State Number Adapter
AX – Output if State_01 is active
AX – Output if State_02 is active
AX – Output if State_03 is active
AX – Output if State_04 is active

Schnittstellenstruktur

Ereignis-Eingänge

- **START_S1**: Startet die Sequenz bei State_01.
- **S1_S2**: Übergang State_01 -> State_02.
- **S2_S3**: Übergang State_02 -> State_03.
- **S3_S4**: Übergang State_03 -> State_04.
- **S4_START**: Übergang State_04 -> START.
- **RESET**: Setzt die Sequenz zurück.

Ereignis-Ausgänge

- **CNF**: Bestätigung der Ausführung.

Daten-Eingänge

- Keine.

Daten-Ausgänge

- **STATE_NR** (SINT): Aktuelle Zustandsnummer.

Adapter

- **DO_S1** (adapter::types::unidirectional::AX): Ausgangsadapter für State_01.
- **DO_S2** (adapter::types::unidirectional::AX): Ausgangsadapter für State_02.
- **DO_S3** (adapter::types::unidirectional::AX): Ausgangsadapter für State_03.
- **DO_S4** (adapter::types::unidirectional::AX): Ausgangsadapter für State_04.

Funktionsweise

Entspricht `sequence_E_04`, verwendet jedoch Adapter für die Ausgänge.

Technische Besonderheiten

- Verwendung von `adapter::types::unidirectional::AX`.

Zustandsübersicht

Siehe `sequence_E_04`.

Anwendungsszenarien

Für ereignisgesteuerte Sequenzen, die über Adapter gesteuert werden.

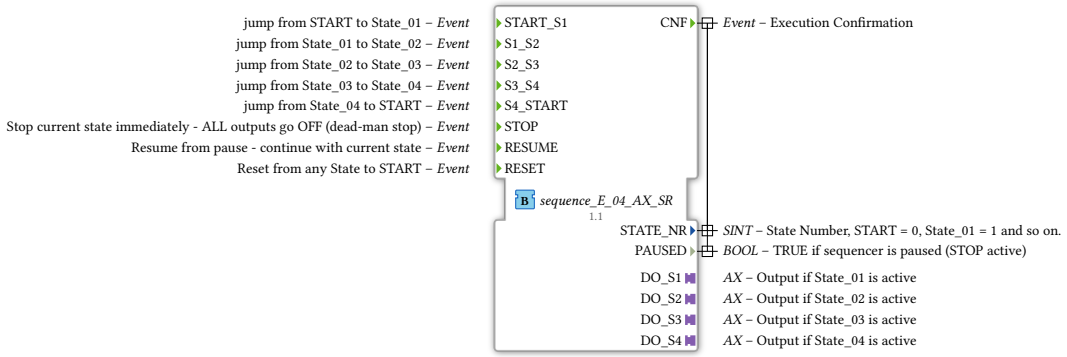
⚖️ Vergleich mit ähnlichen Bausteinen

- **sequence_E_04**: Standardvariante ohne Adapter.

Fazit

Adapter-Variante des 4-Schritt-Ereignis-Sequenzers.

sequence_E_04_AX_SR



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `sequence_E_04_AX_SR` realisiert einen ereignisgesteuerten Sequenzer mit vier Ausgängen über AX-Adapter. Er bietet zusätzlich einen Sicherheitsstopp (STOP), eine Wiederaufnahme (RESUME) und einen Reset (RESET). Die Sequenz durchläuft die Zustände State_00, State_01, State_02, State_03, State_04 und kann zyklisch betrieben werden.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung
`START_S1`	Übergang von START/State_00 nach State_01
`S1_S2`	Übergang von State_01 nach State_02
`S2_S3`	Übergang von State_02 nach State_03
`S3_S4`	Übergang von State_03 nach State_04
`S4_START`	Übergang von State_04 zurück zu State_00
`STOP`	Unterbricht den aktuellen Zustand sofort – alle Ausgänge werden ausgeschaltet (Dead-Man-Stop)
`RESUME`	Setzt die Sequenz aus dem pausierten Zustand fort (Ausgänge werden wieder aktiviert)
`RESET`	Setzt die Sequenz aus jedem Zustand zurück in den START-Zustand (State_00)

Ereignis-Ausgänge

Ereignis	Beschreibung
`CNF`	Bestätigung der Ausführung. Mitgeführte Ausgabedaten: `STATE_NR`, `PAUSED`

Daten-Eingänge

Keine externen Dateneingänge.

Daten-Ausgänge

Variable	Typ	Beschreibung
STATE_NR	SINT	Aktuelle Zustandsnummer: START = 0, State_01 = 1, ..., State_04 = 4
PAUSED	BOOL	TRUE, wenn der Sequenzer pausiert ist (STOP aktiv)

Adapter

Adapter	Typ	Beschreibung
DO_S1	adapter::types::unidirectional::AX	Ausgang, der in State_01 aktiv ist (D1 = TRUE)
DO_S2	adapter::types::unidirectional::AX	Ausgang, der in State_02 aktiv ist
DO_S3	adapter::types::unidirectional::AX	Ausgang, der in State_03 aktiv ist
DO_S4	adapter::types::unidirectional::AX	Ausgang, der in State_04 aktiv ist

Funktionsweise

Der Sequenzer arbeitet nach einem endlichen Automaten (ECC). Die fünf sequenziellen Zustände sind:

- State_00 (START)
- State_01
- State_02
- State_03
- State_04

Der normale Ablauf beginnt mit START_S1 (entweder aus xSTART oder aus State_00), wodurch in State_01 gewechselt wird. Bei jedem Schritt schaltet der FB den entsprechenden Adapterausgang ein (z. B. DO_S1.D1 = TRUE in State_01) und über den Ereignisausgang CNF werden die aktuellen Werte von STATE_NR und PAUSED mitgegeben. Der Übergang zum nächsten Zustand erfolgt durch die entsprechenden Ereignisse (S1_S2, S2_S3, S3_S4) und von State_04 zurück zu State_00 mit S4_START.

Wird STOP empfangen, deaktiviert der FB sofort den aktuellen Ausgang (Exit-Step) und speichert den aktuellen Zustand in der internen Variable savedState. Der Zustand wechselt in einen der pausierten Zustände (sPAUSED_S1 bis sPAUSED_S4 oder sPAUSED_S0). Der Ausgang PAUSED wird auf TRUE gesetzt. Ein RESUME-Ereignis stellt den gespeicherten Zustand wieder her und aktiviert den entsprechenden Ausgang erneut.

Der RESET-Befehl bringt den Sequenzer unabhängig vom aktuellen Zustand in einen Reset-Zustand, in dem alle vier Ausgänge explizit ausgeschaltet werden, und wechselt dann sofort nach State_00.

Technische Besonderheiten

- **AX-Adapter**: Die vier Ausgänge werden über unidirektionale AX-Adapter realisiert, die je einen booleschen Wert D1 besitzen. Die Ausgänge werden nur in den aktiven Zuständen gesetzt, bei Verlassen oder beim Stopp sofort zurückgesetzt.
- **Pause/Resume-Mechanismus**: Die interne Variable savedState speichert den Zustand zum Zeitpunkt des Stopps, sodass nach RESUME exakt der gleiche Zustand fortgesetzt werden kann.
- **Sicherheitsstopp**: Der STOP-Ereignis bewirkt ein sofortiges Abschalten aller Ausgänge (auch ohne Eintreten in einen pausierten Zustand), was als Dead-Man-Stop ausgelegt ist.
- **Verwendung von Konstanten aus dem Paket sequence**: Die Zustandsnummern (State_00, State_01, ...) sind als benannte Konstanten im Package logiBUS::utils::sequence::const::sequence definiert.
- **Reset-Verhalten**: Der RESET schaltet vor dem Rücksprung nach State_00 alle vier Ausgänge aus, wodurch ein definierter Ausgangszustand gewährleistet wird.

Zustandsübersicht

Zustand	Aktiver Ausgang	STATE_NR	PAUSED

xSTART	keiner	0	FALSE
sState_00	keiner	0	FALSE
sState_01	DO_S1	1	FALSE
sState_02	DO_S2	2	FALSE
sState_03	DO_S3	3	FALSE
sState_04	DO_S4	4	FALSE
sPAUSED_S0	keiner	gespeichert	TRUE
sPAUSED_S1	keiner	gespeichert	TRUE
sPAUSED_S2	keiner	gespeichert	TRUE
sPAUSED_S3	keiner	gespeichert	TRUE
sPAUSED_S4	keiner	gespeichert	TRUE
sRESET	keiner	–	–

Anwendungsszenarien

- **Steuerung sequenzieller Prozesse**: z. B. eine Maschine, die nacheinander vier Arbeitsschritte ausführt, wobei jeder Schritt einen eigenen Aktor ansteuert.
- **Sicherheitskritische Anwendungen**: Wenn ein `STOP`-Signal sofort alle Aktoren abschalten muss (z. B. Not-Halt), und nach Freigabe der genaue Zustand wiederhergestellt wird.
- **Zyklische Abläufe**: von Schritt 1 bis 4 und zurück zu Schritt 0 (z. B. in Verpackungsmaschinen).
- **Manuelles Eingreifen**: über `RESET` kann der Ablauf jederzeit auf den Anfang zurückgesetzt werden.

Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu einfacheren Sequenzern (z. B. ohne STOP/RESUME) bietet `sequence_E_04_AX_SR`:

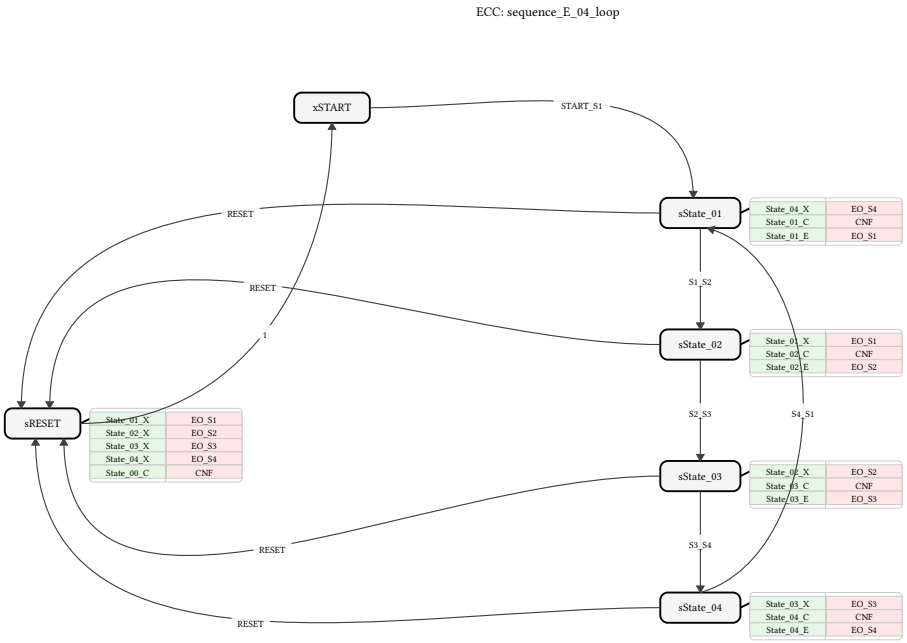
- **Sicherheitsunterbrechung mit definiertem Ausschalten** aller Ausgänge.
- **Pause- und Resume-Funktion**, die den aktuellen Zustand speichert.
- **Zyklische Rücksprungmöglichkeit** von Schritt 4 zu Schritt 0.
- **Ausgabe des aktuellen Zustands** als Zahl und Pausenstatus.

Andere Bausteine mit gleicher Ausgangsanzahl, aber ohne STOP/RESUME, sind einfacher in der Handhabung, bieten jedoch keine Sicherheitsfunktion.

Fazit

Der Funktionsblock `sequence_E_04_AX_SR` ist ein vielseitiger, sicherheitsbewusster Sequenzer für vier Schritte. Er eignet sich besonders für Steuerungen, bei denen ein unterbrechbarer Ablauf mit definierten Ausgangszuständen und Wiederaufnahmeoption erforderlich ist. Die Schnittstelle ist klar strukturiert, die Implementierung robust und durch den Einsatz von AX-Adaptern gut in Automatisierungsumgebungen integrierbar.

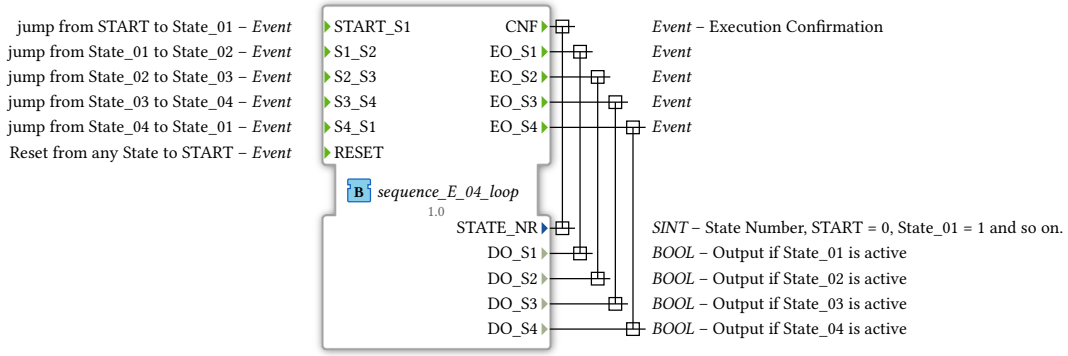
sequence_E_04_loop



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `sequence_E_04_loop` implementiert eine zyklische Sequenz mit vier Zuständen. Die Zustandsübergänge werden ausschließlich durch externe Ereignisse ausgelöst. Der Baustein ist für Steuerungsaufgaben konzipiert, bei denen eine feste Abfolge von Aktionen (repräsentiert durch die Ausgänge `DO_S1` bis `DO_S4`) durchlaufen werden muss und ein Rücksetzen von jedem Zustand aus möglich sein soll.



Schnittstellenstruktur

Ereignis-Eingänge

- **`START_S1`**: Wechselt vom initialen `START`-Zustand in den ersten aktiven Zustand `State_01`.
- **`S1_S2`**: Wechselt von `State_01` zu `State_02`.
- **`S2_S3`**: Wechselt von `State_02` zu `State_03`.
- **`S3_S4`**: Wechselt von `State_03` zu `State_04`.
- **`S4_S1`**: Wechselt von `State_04` zurück zu `State_01` und schließt damit den Zyklus.
- **`RESET`**: Setzt den Baustein von jedem beliebigen aktiven Zustand (`State_01` bis `State_04`) zurück in den initialen `START`-Zustand.

Ereignis-Ausgänge

- **`CNF`**: Wird bei jedem Zustandswechsel ausgelöst und bestätigt die Ausführung. Transportiert den aktuellen Zustandscode über `STATE_NR`.
- **`EO_S1`**: Wird beim Eintritt in `State_01` ausgelöst. Transportiert den Wert `TRUE` für `DO_S1`.
- **`EO_S2`**: Wird beim Eintritt in `State_02` ausgelöst. Transportiert den Wert `TRUE` für `DO_S2`.
- **`EO_S3`**: Wird beim Eintritt in `State_03` ausgelöst. Transportiert den Wert `TRUE` für `DO_S3`.
- **`EO_S4`**: Wird beim Eintritt in `State_04` ausgelöst. Transportiert den Wert `TRUE` für `DO_S4`.

Daten-Eingänge

- Dieser Funktionsblock besitzt keine Dateneingänge.

Daten-Ausgänge

- **`STATE_NR`** (SINT): Gibt die Nummer des aktuellen Zustands aus. Die Kodierung lautet: START = 0, State_01 = 1, State_02 = 2, State_03 = 3, State_04 = 4.
- **`DO_S1`** (BOOL): Ist `TRUE`, wenn der Zustand `State_01` aktiv ist.
- **`DO_S2`** (BOOL): Ist `TRUE`, wenn der Zustand `State_02` aktiv ist.
- **`DO_S3`** (BOOL): Ist `TRUE`, wenn der Zustand `State_03` aktiv ist.
- **`DO_S4`** (BOOL): Ist `TRUE`, wenn der Zustand `State_04` aktiv ist.

Adapter

- Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Baustein ist als Basic Function Block (BFB) mit einer Execution Control Chart (ECC) implementiert. Die interne Logik basiert auf sechs Zuständen: einem initialen Zustand (`xSTART`), vier aktiven Arbeitszuständen (`sState_01` bis `sState_04`) und einem speziellen Reset-Zustand (`sRESET`).

Bei einem Zustandswechsel werden drei Aktionen nacheinander ausgeführt:

1. **Exit-Algorithmus (X)**: Der Ausgang des vorherigen Zustands wird auf `FALSE` gesetzt.
2. **Confirmation-Algorithmus (C)**: Die Zustandsnummer `STATE_NR` wird aktualisiert und das `CNF`-Ereignis wird ausgelöst.
3. **Entry-Algorithmus (E)**: Der Ausgang des neuen Zustands wird auf `TRUE` gesetzt und das entsprechende Ereignis (`EO_Sx`) wird ausgelöst.

Ein `RESET`-Ereignis führt immer in den `sRESET`-Zustand, wo alle Ausgänge (`DO_S1` bis `DO_S4`) deaktiviert werden, die Zustandsnummer auf 0 (`START`) gesetzt und ein `CNF` gesendet wird. Anschließend wechselt der Baustein automatisch zurück in den initialen `xSTART`-Zustand.

Technische Besonderheiten

- **Ereignisgesteuerte Transitionen**: Im Gegensatz zu zeit- oder bedingungsgesteuerten Sequenzern erfolgen alle Zustandsübergänge nur bei Eintreffen des spezifischen Ereignisses. Es gibt keine automatischen oder zeitgesteuerten Weiter schaltungen.
- **Explizite Reset-Logik**: Der Reset-Vorgang ist als eigener Zustand (`sRESET`) modelliert, der alle Ausgänge sauber zurücksetzt, bevor der Startzustand wieder erreicht wird.
- **Zustandskodierung**: Die Verwendung der Konstanten aus `sequence::State_xx` für die Zuweisung an `STATE_NR` erhöht die Wartbarkeit und Lesbarkeit des Codes.

Zustandsübersicht

1. **xSTART**: Initialer, inaktiver Zustand. Alle Ausgänge sind `FALSE`, `STATE_NR` ist 0.
2. **sState_01**: Erster aktiver Zustand. `DO_S1 = TRUE`, `STATE_NR = 1`.
3. **sState_02**: Zweiter aktiver Zustand. `DO_S2 = TRUE`, `STATE_NR = 2`.
4. **sState_03**: Dritter aktiver Zustand. `DO_S3 = TRUE`, `STATE_NR = 3`.
5. **sState_04**: Vierter aktiver Zustand. `DO_S4 = TRUE`, `STATE_NR = 4`.
6. **sRESET**: Temporärer Reset-Zustand. Setzt alle Aktor-Ausgänge (`DO_Sx`) auf `FALSE` und `STATE_NR` auf 0.

Die erlaubten Übergänge sind: `START -> S1 -> S2 -> S3 -> S4 -> (S1)` sowie von jedem Zustand `S1`-`S4` via `RESET` zurück zu `START`.

Anwendungsszenarien

- **Schrittkettensteuerungen**: Steuerung von Maschinenabläufen, bei denen jeder Schritt manuell oder durch ein Sensorsignal freigegeben werden muss (z.B. manuelle Bestückungsstationen).

- **Taktgesteuerte Prozesse**: In Produktionslinien, wo ein zentraler Taktgeber (`Sx_Sy`-Ereignisse) den Fortschritt der Baugruppe von Station zu Station signalisiert.
- **Test- und Inbetriebnahmesequenzen**: Strukturierte Abfolge von Tests, die vom Bediener bestätigt werden müssen.

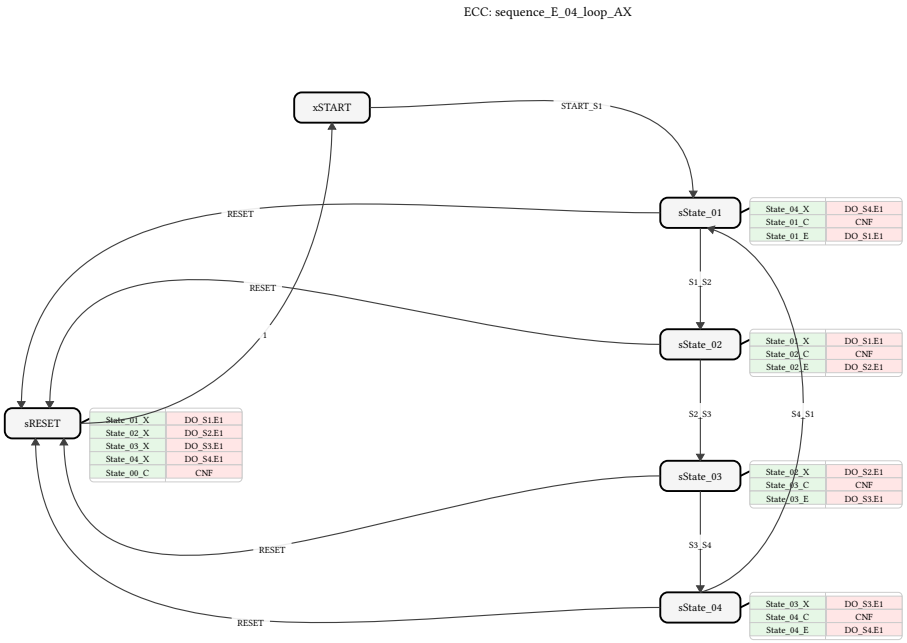
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem **Zyklischen Sequenzer mit Timer** (z.B. `E_CYCLE`) fehlt hier die integrierte Zeitsteuerung; die Weiter schaltung erfolgt rein ereignisbasiert. Gegenüber einem **Binären Schieberegister** oder Zähler bietet dieser Baustein eine explizite, leicht nachvollziehbare Zustandsmaschine mit klaren Entry/Exit-Aktionen und einem dedizierten Reset-Pfad, was die Fehlersuche vereinfacht.

Fazit

Der `sequence_E_04_loop` ist ein robuster und klar strukturierter Funktionsblock zur Implementierung einer ereignisgesteuerten 4-Zustands-Sequenz. Seine Stärken liegen in der expliziten Modellierung jedes Zustandsübergangs, der sauberen Reset-Logik und der klaren Trennung von Entry-, Exit- und Bestätigungsaktionen. Er eignet sich ideal für Anwendungen, bei denen die Ablaufsteuerung durch externe Signale (Taster, Sensoren, übergeordnete Steuerungen) erfolgen soll.

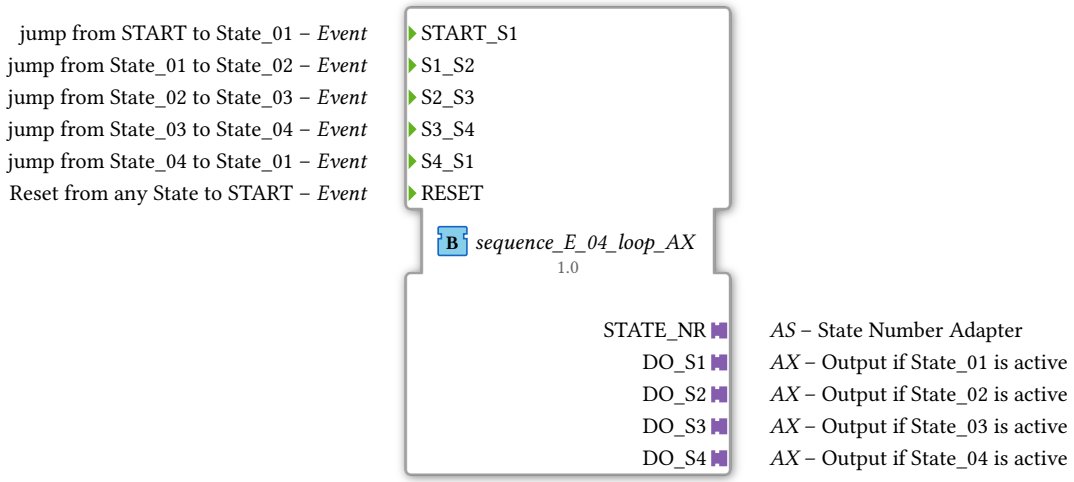
sequence_E_04_loop_AX



• — ○ • — ○ • — ○ •

Einleitung

`sequence_E_04_loop_AX` ist eine Variante des `sequence_E_04_loop`, die zusätzlich Adapter (`AX`) für die Ausgänge verwendet. Er steuert eine rein ereignisgesteuerte, zyklische Sequenz mit 4 Ausgabezuständen.



Schnittstellenstruktur

Ereignis-Eingänge

- **START_S1**: Startet die Sequenz bei State_01.
- **S1_S2**: Übergang State_01 -> State_02.
- **S2_S3**: Übergang State_02 -> State_03.
- **S3_S4**: Übergang State_03 -> State_04.
- **S4_S1**: Übergang State_04 -> State_01 (Loop).
- **RESET**: Setzt die Sequenz zurück.

Ereignis-Ausgänge

- **CNF**: Bestätigung der Ausführung.

Daten-Eingänge

- Keine.

Daten-Ausgänge

- **STATE_NR** (SINT): Aktuelle Zustandsnummer.

Adapter

- **DO_S1** (adapter::types::unidirectional::AX): Ausgangsadapter für State_01.
- **DO_S2** (adapter::types::unidirectional::AX): Ausgangsadapter für State_02.
- **DO_S3** (adapter::types::unidirectional::AX): Ausgangsadapter für State_03.
- **DO_S4** (adapter::types::unidirectional::AX): Ausgangsadapter für State_04.

Funktionsweise

Entspricht `sequence_E_04_loop`, verwendet jedoch Adapter für die Ausgänge.

Technische Besonderheiten

- Verwendung von `adapter::types::unidirectional::AX`.

Zustandsübersicht

Siehe `sequence_E_04_loop`.

Anwendungsszenarien

Für zyklische ereignisgesteuerte Sequenzen mit Adapteranbindung.

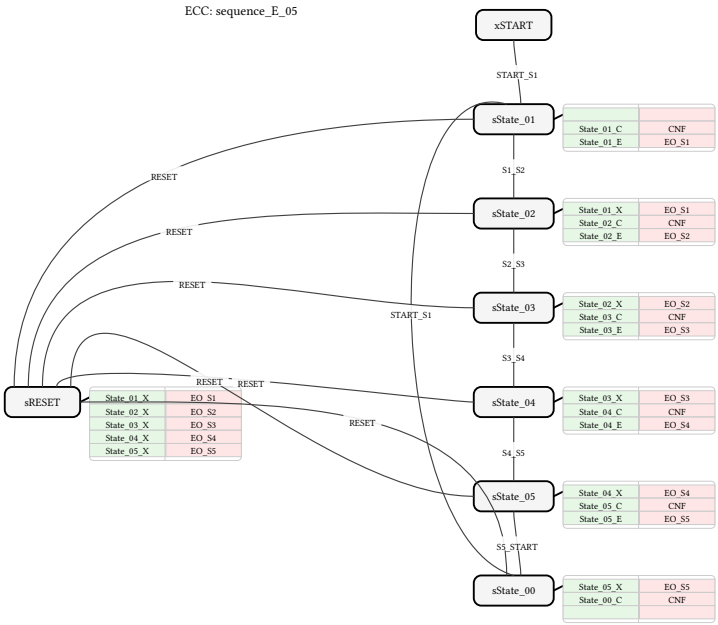
⚖️ Vergleich mit ähnlichen Bausteinen

- **sequence_E_04_loop**: Standardvariante ohne Adapter.

Fazit

Adapter-Variante des 4-Schritt-Loop-Ereignis-Sequenzers.

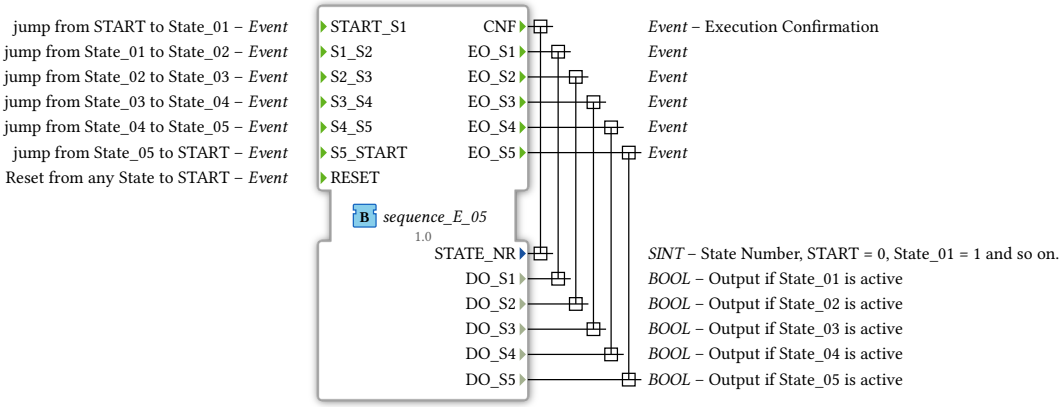
sequence_E_05



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `sequence_E_05` ist ein sequenzieller Zustandsautomat, der eine feste Abfolge von fünf Zuständen durchläuft. Der Übergang zwischen den einzelnen Zuständen erfolgt ausschließlich durch externe Ereignisse. Der Baustein ist für Steuerungsaufgaben konzipiert, bei denen eine schrittweise Abarbeitung einer Prozesssequenz erforderlich ist, wie z.B. in Handhabungs- oder Verpackungsanlagen. Jeder aktive Zustand setzt einen eigenen binären Ausgang und bestätigt die Ausführung.



Schnittstellenstruktur

Der Funktionsblock besitzt eine reine Ereignisschnittstelle. Zustandsübergänge werden durch Eingangsereignisse ausgelöst, und der aktive Zustand wird über Ausgangsereignisse und Datenausgänge signalisiert.

Ereignis-Eingänge

- `START_S1`: Wechselt vom initialen START-Zustand oder vom Zustand `sState_00` in den ersten Zustand `State_01`.
- `S1_S2`: Wechselt von `State_01` zu `State_02`.
- `S2_S3`: Wechselt von `State_02` zu `State_03`.
- `S3_S4`: Wechselt von `State_03` zu `State_04`.
- `S4_S5`: Wechselt von `State_04` zu `State_05`.
- `S5_START`: Wechselt von `State_05` zurück in den inaktiven Zustand `sState_00`.
- `RESET`: Setzt den Automaten von jedem beliebigen Zustand sofort in den inaktiven Zustand `sState_00` zurück.

Ereignis-Ausgänge

- `CNF`: Allgemeines Bestätigungsereignis. Wird bei jedem Zustandswechsel ausgelöst und liefert die aktuelle Zustandsnummer `STATE_NR`.
- `EO_S1`: Wird beim Eintritt in `State_01` ausgelöst und liefert den Wert `DO_S1`.
- `EO_S2`: Wird beim Eintritt in `State_02` ausgelöst und liefert den Wert `DO_S2`.
- `EO_S3`: Wird beim Eintritt in `State_03` ausgelöst und liefert den Wert `DO_S3`.
- `EO_S4`: Wird beim Eintritt in `State_04` ausgelöst und liefert den Wert `DO_S4`.
- `EO_S5`: Wird beim Eintritt in `State_05` ausgelöst und liefert den Wert `DO_S5`.

Daten-Eingänge

Es sind keine Dateneingänge vorhanden.

Daten-Ausgänge

- `STATE_NR` (SINT): Numerische Kennung des aktuellen Zustands. START = 0, State_01 = 1, State_02 = 2, ..., State_05 = 5.
- `DO_S1` (BOOL): Ist `TRUE`, wenn Zustand `State_01` aktiv ist.
- `DO_S2` (BOOL): Ist `TRUE`, wenn Zustand `State_02` aktiv ist.
- `DO_S3` (BOOL): Ist `TRUE`, wenn Zustand `State_03` aktiv ist.
- `DO_S4` (BOOL): Ist `TRUE`, wenn Zustand `State_04` aktiv ist.
- `DO_S5` (BOOL): Ist `TRUE`, wenn Zustand `State_05` aktiv ist.

Adapter

Es sind keine Adapterschnittstellen vorhanden.

Funktionsweise

Der FB ist als Basic Function Block (BFB) mit einem Execution Control Chart (ECC) implementiert. Das ECC besteht aus sieben Zuständen: einem initialen `xSTART`, fünf aktiven Zuständen (`sState_01` bis `sState_05`), einem inaktiven Endzustand (`sState_00`) und einem speziellen Reset-Zustand (`sRESET`).

Bei Eintritt in einen aktiven Zustand (z.B. `sState_01`) werden zwei Algorithmen nacheinander ausgeführt:

1. Ein *Confirmation Step* (z.B. `State_01_C`), der die `STATE_NR` auf die entsprechende Konstante setzt und das `CNF`-Ereignis auslöst.
2. Ein *Entry Step* (z.B. `State_01_E`), der den zugehörigen binären Ausgang (z.B. `DO_S1`) auf `TRUE` setzt und das entsprechende Ereignis (z.B. `EO_S1`) auslöst.

Beim Verlassen eines aktiven Zustands wird ein *Exit Step* (z.B. `State_01_X`) ausgeführt, der den zugehörigen binären Ausgang auf `FALSE` zurücksetzt. Die Zustandsübergänge werden strikt durch die anliegenden Ereignisseingänge gesteuert. Ein `RESET`-Ereignis führt über den Zwischenzustand `sRESET` dazu, dass alle aktiven Ausgänge (`DO_S1` bis `DO_S5`) deaktiviert werden, bevor der Automat in den inaktiven Zustand `sState_00` wechselt.

Technische Besonderheiten

- **Ereignisgesteuerte Transitionen:** Im Gegensatz zu zeit- oder bedingungsgesteuerten Sequenzern erfolgen die Zustandswechsel hier ausschließlich durch externe Ereignisse. Dies ermöglicht eine enge Kopplung an andere Prozessschritte oder Bedieneraktionen.
- **Explizite Reset-Logik:** Der Reset-Vorgang ist als eigener ECC-Zustand (`sRESET`) modelliert, der sicherstellt, dass beim Zurücksetzen alle fünf Ausgänge (`DO_S1` bis `DO_S5`) ordnungsgemäß deaktiviert werden, unabhängig vom aktuellen Zustand.

- **Separate Entry/Exit-Aktionen:** Die Logik zum Setzen und Rücksetzen der Ausgänge ist in separate Algorithmen aufgeteilt (E für Entry, X für Exit). Dies fördert eine klare und wartbare Struktur.
- **Konstanten für Zustandsnummern:** Die Werte für `STATE_NR` werden aus einer importierten Bibliothek (`sequence::State_01`, etc.) bezogen, was die Wiederverwendbarkeit und zentrale Pflege erleichtert.

Zustandsübersicht

1. **xSTART:** Initialer Zustand nach dem Start. Wartet auf `START_S1`.
2. **sState_01:** Erster aktiver Schritt. Setzt `DO_S1=TRUE` und `STATE_NR=1`. Wartet auf `S1_S2` oder `RESET`.
3. **sState_02:** Zweiter aktiver Schritt. Setzt `DO_S2=TRUE` und `STATE_NR=2`. Wartet auf `S2_S3` oder `RESET`.
4. **sState_03:** Dritter aktiver Schritt. Setzt `DO_S3=TRUE` und `STATE_NR=3`. Wartet auf `S3_S4` oder `RESET`.
5. **sState_04:** Vierter aktiver Schritt. Setzt `DO_S4=TRUE` und `STATE_NR=4`. Wartet auf `S4_S5` oder `RESET`.
6. **sState_05:** Fünfter aktiver Schritt. Setzt `DO_S5=TRUE` und `STATE_NR=5`. Wartet auf `S5_START` oder `RESET`.
7. **sState_00:** Inaktiver Endzustand. Alle Ausgänge sind `FALSE`, `STATE_NR=0`. Wartet auf `START_S1` für einen neuen Zyklus.
8. **sRESET:** Temporärer Reset-Zustand. Deaktiviert alle aktiven Ausgänge und wechselt dann zu `sState_00`.

Anwendungsszenarien

- **Schrittkette in Handhabungsgeräten:** Steuerung eines Pick-and-Place Roboters (Greifen → Anheben → Bewegen → Absenken → Loslassen), wobei jeder Schritt durch ein Sensorereignis (z.B. „Teil erkannt“, „Position erreicht“) ausgelöst wird.
- **Manuelle Bediensequenzen:** Abarbeitung einer vom Bediener vorgegebenen Reihenfolge, z.B. in einer Maschineneinrichtung („Schritt 1 freigeben“ → „Schritt 2 freigeben“).
- **Synchronisation mit übergeordneten Steuerungen:** Die Sequenz dient als Unterprogramm einer Hauptsteuerung, die den Fortschritt über die Ereignisse vorgibt.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu einem zyklisch ablaufenden Sequenzer (z.B. `CYCLE_5`), der nach einer festen Zeit automatisch weiter schaltet, verharrt `sequence_E_05` in jedem Zustand, bis das entsprechende Weitereignis eintrifft. Dies macht ihn deterministischer in Bezug auf externe Bedingungen, erfordert aber eine zuverlässige Ereignisgenerierung durch die Peripherie oder übergeordnete Logik. Er ist einfacher und transparenter als ein kompletter Schrittkettenbaustein (SFC) für feste, lineare Abfolgen.

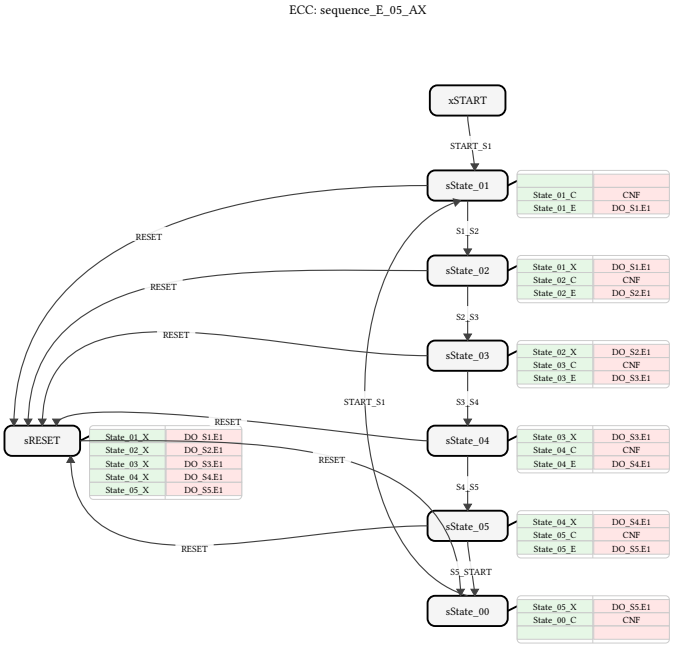
Fazit

Der `sequence_E_05` ist ein robuster und einfach zu konfigurierender Funktionsblock für ereignisgesteuerte Schrittketten mit fester Länge. Seine klare Trennung von Zustandslogik und Ausgangsaktionen sowie der integrierte Reset-Mechanismus machen ihn zu einer zuverlässigen Komponente für sequentielle Steuerungsaufgaben in 4diac-basierten Automatisierungssystemen. Die fehlende Flexibilität in der Anzahl der Schritte oder der Übergangsbedingungen ist für definierte, lineare Prozesse ein Vorteil in Bezug auf Übersichtlichkeit und Vorhersagbarkeit.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

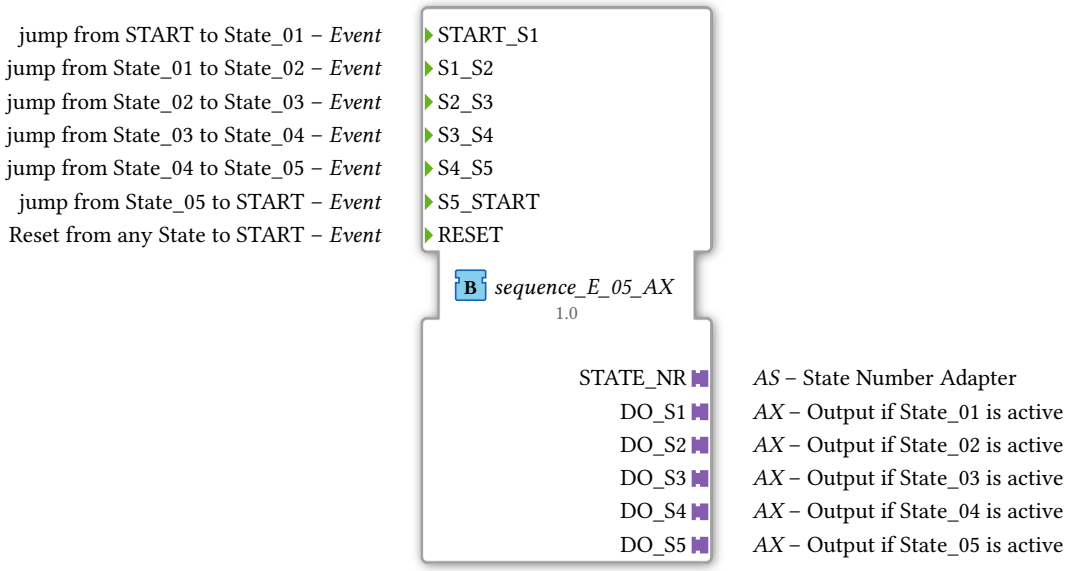
sequence_E_05_AX



• – ○ • – ○ • – ○ •

Einleitung

`sequence_E_05_AX` ist eine Variante des `sequence_E_05`, die zusätzlich Adapter (`AX`) für die Ausgänge verwendet. Er steuert eine rein ereignisgesteuerte Sequenz mit 5 Ausgabezuständen.



Schnittstellenstruktur

Ereignis-Eingänge

- **START_S1**: Startet die Sequenz bei State_01.
- **S1_S2**: Übergang State_01 -> State_02.
- **S2_S3**: Übergang State_02 -> State_03.
- **S3_S4**: Übergang State_03 -> State_04.

- **S4_S5**: Übergang State_04 -> State_05.
- **S5_START**: Übergang State_05 -> START.
- **RESET**: Setzt die Sequenz zurück.

Ereignis-Ausgänge
- **CNF**: Bestätigung der Ausführung.

Daten-Eingänge
- Keine.

Daten-Ausgänge
- **STATE_NR** (SINT): Aktuelle Zustandsnummer.

Adapter
- **DO_S1** (adapter::types::unidirectional::AX): Ausgangsadapter für State_01.
- **DO_S2** (adapter::types::unidirectional::AX): Ausgangsadapter für State_02.
- **DO_S3** (adapter::types::unidirectional::AX): Ausgangsadapter für State_03.
- **DO_S4** (adapter::types::unidirectional::AX): Ausgangsadapter für State_04.
- **DO_S5** (adapter::types::unidirectional::AX): Ausgangsadapter für State_05.

Funktionsweise
Entspricht `sequence_E_05`, verwendet jedoch Adapter für die Ausgänge.

Technische Besonderheiten
- Verwendung von `adapter::types::unidirectional::AX`.

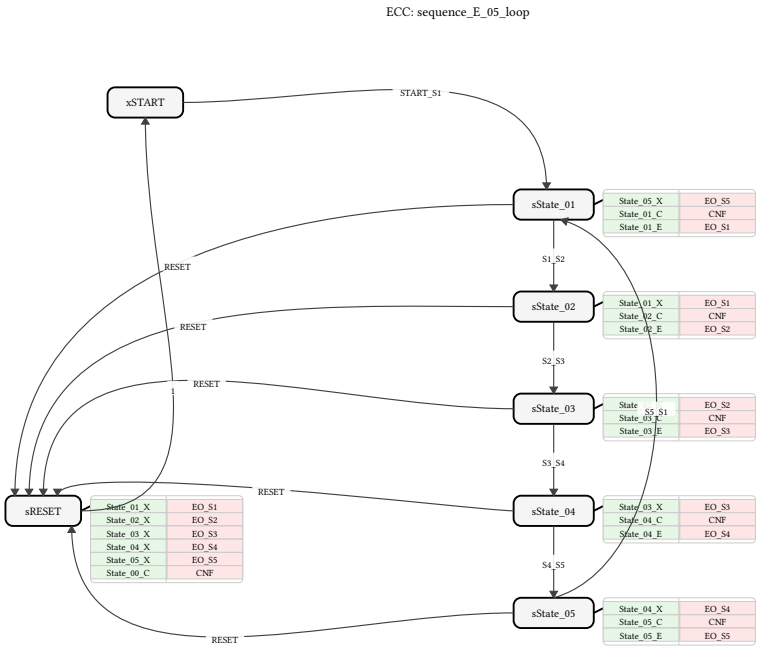
Zustandsübersicht
Siehe `sequence_E_05`.

Anwendungsszenarien
Für ereignisgesteuerte 5-stufige Sequenzen mit Adapteranbindung.

⚖️ Vergleich mit ähnlichen Bausteinen
- **sequence_E_05**: Standardvariante ohne Adapter.

Fazit
Adapter-Variante des 5-Schritt-Ereignis-Sequenzers.

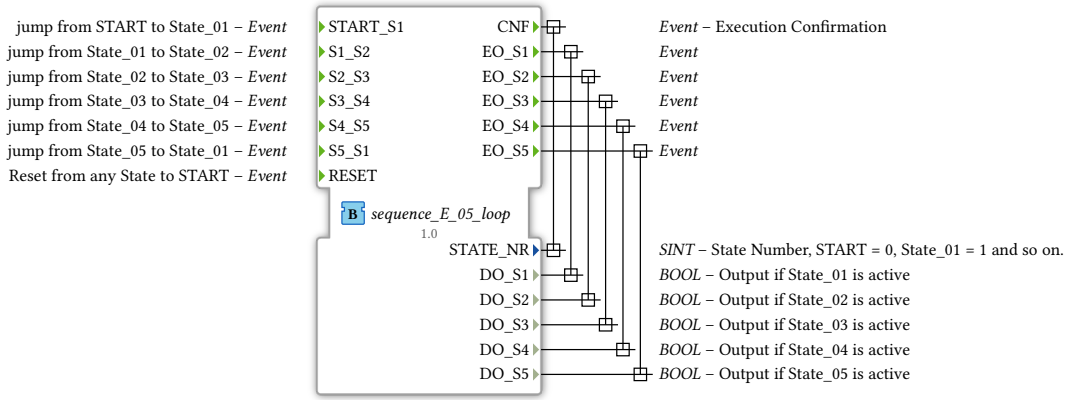
sequence_E_05_loop



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `sequence_E_05_loop` implementiert eine zyklische Sequenz mit fünf Zuständen. Der Übergang zwischen den einzelnen Zuständen erfolgt ausschließlich über externe Ereignisse. Der Baustein ist für Anwendungen konzipiert, in denen ein Prozessschritt erst nach Abschluss eines vorherigen Schritts und dem Eintreffen eines spezifischen Freigabesignals beginnen darf. Die Sequenz kann von jedem Zustand aus in den initialen Startzustand zurückgesetzt werden.



Schnittstellenstruktur

Ereignis-Eingänge

- `START_S1`: Wechselt vom Startzustand (`START`) in den ersten aktiven Zustand (`State_01`).
- `S1_S2`: Wechselt von `State_01` zu `State_02`.
- `S2_S3`: Wechselt von `State_02` zu `State_03`.
- `S3_S4`: Wechselt von `State_03` zu `State_04`.
- `S4_S5`: Wechselt von `State_04` zu `State_05`.
- `S5_S1`: Wechselt von `State_05` zurück zu `State_01` (Zyklus).
- `RESET`: Setzt die Sequenz von jedem beliebigen Zustand in den initialen Startzustand (`START`) zurück.

Ereignis-Ausgänge

- `CNF`: Allgemeines Bestätigungsereignis, das bei jedem Zustandswechsel ausgelöst wird. Es liefert die aktuelle Zustandsnummer (`STATE_NR`) mit.
- `EO_S1`: Wird beim Eintritt in `State_01` ausgelöst und liefert den zugehörigen Datenausgang `DO_S1` mit.
- `EO_S2`: Wird beim Eintritt in `State_02` ausgelöst und liefert den zugehörigen Datenausgang `DO_S2` mit.
- `EO_S3`: Wird beim Eintritt in `State_03` ausgelöst und liefert den zugehörigen Datenausgang `DO_S3` mit.
- `EO_S4`: Wird beim Eintritt in `State_04` ausgelöst und liefert den zugehörigen Datenausgang `DO_S4` mit.
- `EO_S5`: Wird beim Eintritt in `State_05` ausgelöst und liefert den zugehörigen Datenausgang `DO_S5` mit.

Daten-Eingänge

- Dieser Funktionsblock besitzt keine Dateneingänge.

Daten-Ausgänge

- `STATE_NR` (SINT): Gibt die Nummer des aktuell aktiven Zustands aus. Die Kodierung lautet: START = 0, State_01 = 1, State_02 = 2, ..., State_05 = 5.
- `DO_S1` (BOOL): Ist `TRUE`, wenn `State_01` aktiv ist.
- `DO_S2` (BOOL): Ist `TRUE`, wenn `State_02` aktiv ist.
- `DO_S3` (BOOL): Ist `TRUE`, wenn `State_03` aktiv ist.
- `DO_S4` (BOOL): Ist `TRUE`, wenn `State_04` aktiv ist.
- `DO_S5` (BOOL): Ist `TRUE`, wenn `State_05` aktiv ist.

Adapter

- Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der FB ist als Basic Function Block (BFB) mit einer Execution Control Chart (ECC) implementiert. Die interne Logik basiert auf sieben Zuständen: einem initialen Startzustand (`xSTART`), fünf aktiven Zuständen (`sState_01` bis `sState_05`) und einem separaten Reset-Zustand (`sRESET`).

Bei Eintritt in einen aktiven Zustand werden drei Aktionen nacheinander ausgeführt:

1. **Exit-Aktion des vorherigen Zustands:** Der entsprechende `State_XX_X`-Algorithmus setzt den zugehörigen Datenausgang (`DO_Sx`) auf `FALSE`.
2. **Confirmation-Aktion:** Der `State_XX_C`-Algorithmus setzt die Zustandsnummer `STATE_NR` und löst das `CNF`-Ereignis aus.
3. **Entry-Aktion des neuen Zustands:** Der `State_XX_E`-Algorithmus setzt den zugehörigen Datenausgang (`DO_Sx`) auf `TRUE` und löst das entsprechende Ereignis `EO_Sx` aus.

Ein `RESET`-Ereignis führt in den `sRESET`-Zustand. Hier werden alle potenziell aktiven Ausgänge (`DO_S1` bis `DO_S5`) über die jeweiligen Exit-Algorithmen deaktiviert, die Zustandsnummer auf 0 (`START`) gesetzt und ein `CNF`-Ereignis ausgelöst. Anschließend wechselt der FB automatisch (Bedingung = `1`, also immer wahr) zurück in den initialen `xSTART`-Zustand.

Technische Besonderheiten

- **Ereignisgesteuerte Transitionen:** Jeder Zustandsübergang muss explizit durch das entsprechende Eingangsereignis ausgelöst werden. Es gibt keine zeit- oder datengesteuerten automatischen Übergänge.
- **Entkoppelte Signale:** Die booleschen Zustandsausgänge (`DO_Sx`) und die zugehörigen Ereignisausgänge (`EO_Sx`) werden synchron gesetzt und ausgelöst. Dies erlaubt eine flexible Anbindung an nachfolgende Logik.
- **Explizite Reset-Logik:** Der Reset-Vorgang deaktiviert systematisch alle Ausgänge, bevor der Startzustand wiederhergestellt wird, was ein sauberes und definiertes Verhalten gewährleistet.
- **Konstanten für Zustandsnummern:** Die Zustandsnummern werden über den importierten Namensraum `sequence::` (z.B. `sequence::State_01`) zugewiesen, was die Wartbarkeit und Lesbarkeit des Codes erhöht.

Zustandsübersicht

Die ECC besteht aus folgenden Zuständen und möglichen Übergängen:

- **xSTART:** Initialer, inaktiver Zustand. Übergang zu `sState_01` via `START_S1`.
- **sState_01:** Erster aktiver Zustand. Übergänge zu `sState_02` via `S1_S2` oder zu `sRESET` via `RESET`.
- **sState_02:** Zweiter aktiver Zustand. Übergänge zu `sState_03` via `S2_S3` oder zu `sRESET` via `RESET`.

- **sState_03:** Dritter aktiver Zustand. Übergänge zu `sState_04` via `S3_S4` oder zu `sRESET` via `RESET`.
- **sState_04:** Vierter aktiver Zustand. Übergänge zu `sState_05` via `S4_S5` oder zu `sRESET` via `RESET`.
- **sState_05:** Fünfter aktiver Zustand. Übergänge zu `sState_01` via `S5_S1` (Zyklus) oder zu `sRESET` via `RESET`.
- **sRESET:** Reset-Zustand. Führt Reset-Aktionen aus und wechselt anschließend bedingungslos zurück zu `xSTART`.

Anwendungsszenarien

- **Schrittkettensteuerungen:** Steuerung von Maschinenabläufen, bei denen jeder Schritt manuell oder durch ein Sensorsignal freigegeben werden muss (z.B. Handarbeitsplatz mit Freigabetaster).
- **Batch-Prozesse:** Durchführung von Rezeptschritten, bei denen der Operator oder ein übergeordnetes System den nächsten Schritt bestätigen muss.
- **Sicherheitskritische Sequenzen:** Prozesse, bei denen ein unkontrollierter automatischer Wechsel vermieden werden soll.
- **Test- und Inbetriebnahmesequenzen:** Manuelles Durchschalten von einzelnen Funktionen eines Systems.

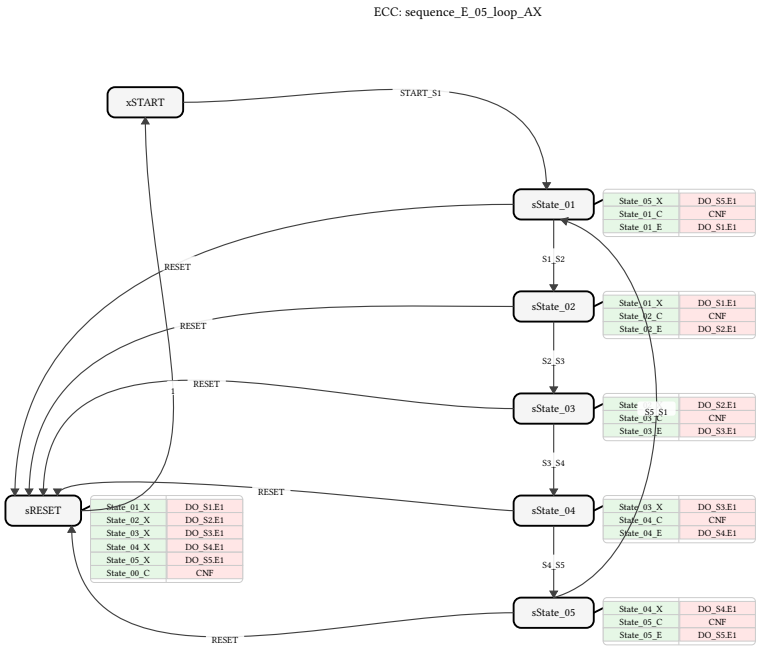
⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu **zyklisch ablaufenden Sequenzern** (z.B. `E_CYCLE`) oder **zeitgesteuerten Sequenzern** (mit `DELAY`-Blöcken) verharrt dieser Baustein in jedem Zustand, bis das spezifische Wechselereignis eintrifft. Er bietet damit maximale externe Kontrolle. Im Vergleich zu einem generischen **Zähler** (`CTU`) mit nachgelagerter Dekodierlogik bietet dieser FB eine vollständig gekapselte, zustandsbasierte Lösung mit klaren Entry/Exit-Aktionen und einem integrierten Reset-Mechanismus.

Fazit

Der `sequence_E_05_loop` ist ein robuster und einfach zu konfigurierender Funktionsblock für ereignisgesteuerte Schrittketten mit fünf Zuständen. Seine klare Schnittstelle, die saubere Trennung von Entry-/Exit-Logik und der zuverlässige Reset-Mechanismus machen ihn ideal für Anwendungen, bei denen jeder Prozessschritt explizit freigegeben werden muss. Die mitgeführten Datenausgänge und Bestätigungsereignisse ermöglichen eine einfache Integration in übergeordnete Steuerungs- und Visualisierungssysteme.

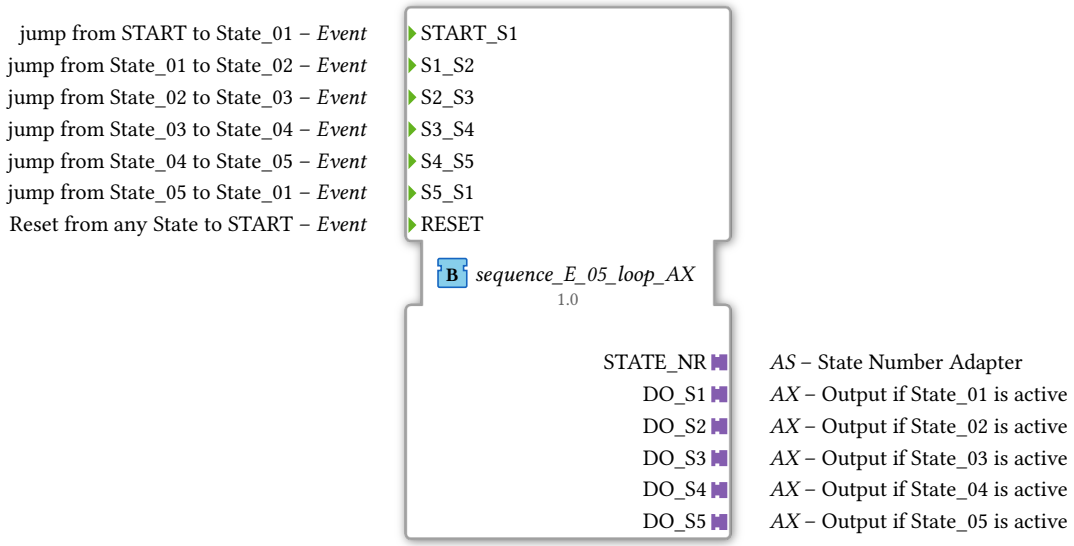
sequence_E_05_loop_AX



• – ○ • – ○ • – ○ •

Einleitung

`sequence_E_05_loop_AX` ist eine Variante des `sequence_E_05_loop`, die zusätzlich Adapter (`AX`) für die Ausgänge verwendet. Er steuert eine rein ereignisgesteuerte, zyklische Sequenz mit 5 Ausgabezuständen.



Schnittstellenstruktur

Ereignis-Eingänge

- **START_S1**: Startet die Sequenz bei State_01.
- **S1_S2**: Übergang State_01 -> State_02.
- **S2_S3**: Übergang State_02 -> State_03.
- **S3_S4**: Übergang State_03 -> State_04.
- **S4_S5**: Übergang State_04 -> State_05.
- **S5_S1**: Übergang State_05 -> State_01 (Loop).
- **RESET**: Setzt die Sequenz zurück.

Ereignis-Ausgänge

- **CNF**: Bestätigung der Ausführung.

Daten-Eingänge

- Keine.

Daten-Ausgänge

- **STATE_NR** (SINT): Aktuelle Zustandsnummer.

Adapter

- **DO_S1** (adapter::types::unidirectional::AX): Ausgangsadapter für State_01.
- **DO_S2** (adapter::types::unidirectional::AX): Ausgangsadapter für State_02.
- **DO_S3** (adapter::types::unidirectional::AX): Ausgangsadapter für State_03.
- **DO_S4** (adapter::types::unidirectional::AX): Ausgangsadapter für State_04.
- **DO_S5** (adapter::types::unidirectional::AX): Ausgangsadapter für State_05.

Funktionsweise

Entspricht `sequence_E_05_loop`, verwendet jedoch Adapter für die Ausgänge.

Technische Besonderheiten

- Verwendung von `adapter::types::unidirectional::AX`.

Zustandsübersicht

Siehe `sequence_E_05_loop`.

Anwendungsszenarien

Für zyklische ereignisgesteuerte 5-stufige Sequenzen mit Adapteranbindung.

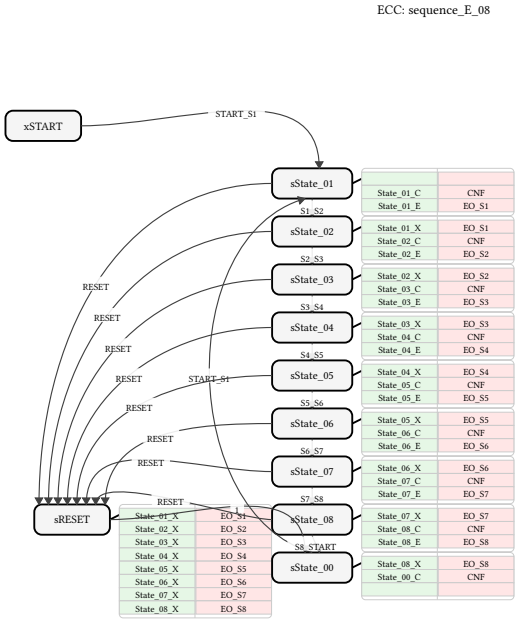
⚖️ Vergleich mit ähnlichen Bausteinen

- **sequence_E_05_loop**: Standardvariante ohne Adapter.

Fazit

Adapter-Variante des 5-Schritt-Loop-Ereignis-Sequenzers.

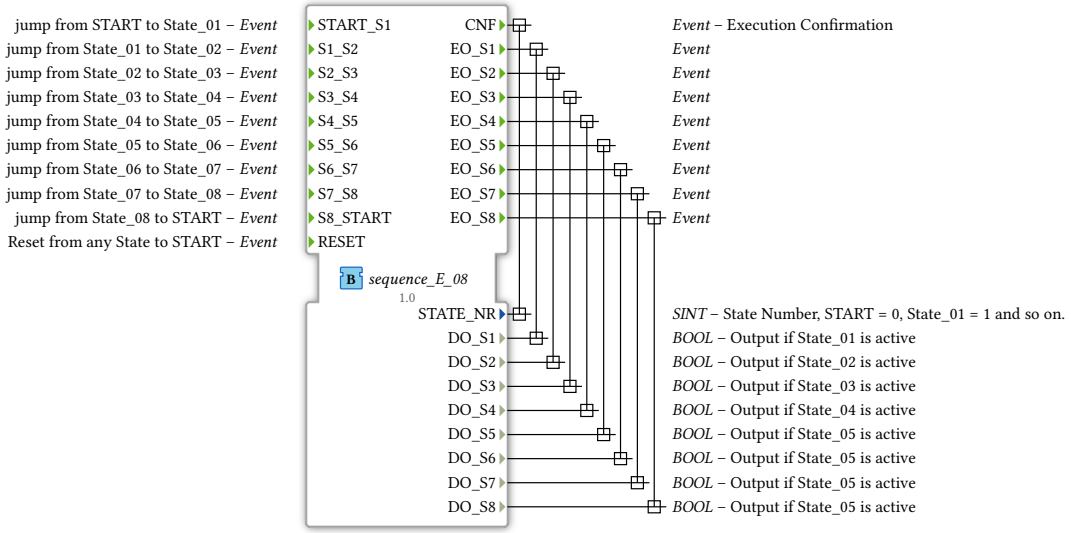
sequence_E_08



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock `sequence_E_08` ist ein Sequenzer mit acht Ausgängen, der durch Ereignisse gesteuert wird. Er realisiert eine lineare Zustandsmaschine mit einem definierten Startzustand und acht aktiven Zuständen. Der Übergang von einem Zustand zum nächsten erfolgt ausschließlich durch das Eintreffen eines spezifischen Ereignisses. Der Baustein eignet sich für Steuerungsaufgaben, bei denen Prozessschritte nacheinander und ereignisgesteuert durchlaufen werden müssen, wie z.B. in Handhabungs- oder Montageabläufen.



Schnittstellenstruktur

Ereignis-Eingänge

- **START_S1**: Wechselt vom Startzustand (`START`) in den Zustand `State_01`.
- **S1_S2**: Wechselt von `State_01` nach `State_02`.
- **S2_S3**: Wechselt von `State_02` nach `State_03`.
- **S3_S4**: Wechselt von `State_03` nach `State_04`.
- **S4_S5**: Wechselt von `State_04` nach `State_05`.
- **S5_S6**: Wechselt von `State_05` nach `State_06`.
- **S6_S7**: Wechselt von `State_06` nach `State_07`.
- **S7_S8**: Wechselt von `State_07` nach `State_08`.
- **S8_START**: Wechselt von `State_08` zurück in den `START`-Zustand (repräsentiert durch `sState_00`).
- **RESET**: Setzt den Baustein aus *jedem beliebigen Zustand* sofort in den `START`-Zustand zurück.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Wird bei jedem Zustandswechsel ausgelöst und liefert die neue Zustandsnummer (`STATE_NR`).
- **EO_S1**: Wird beim Eintritt in `State_01` ausgelöst und liefert den Wert von `DO_S1` (TRUE).
- **EO_S2**: Wird beim Eintritt in `State_02` ausgelöst und liefert den Wert von `DO_S2` (TRUE).
- **EO_S3**: Wird beim Eintritt in `State_03` ausgelöst und liefert den Wert von `DO_S3` (TRUE).
- **EO_S4**: Wird beim Eintritt in `State_04` ausgelöst und liefert den Wert von `DO_S4` (TRUE).
- **EO_S5**: Wird beim Eintritt in `State_05` ausgelöst und liefert den Wert von `DO_S5` (TRUE).
- **EO_S6**: Wird beim Eintritt in `State_06` ausgelöst und liefert den Wert von `DO_S6` (TRUE).
- **EO_S7**: Wird beim Eintritt in `State_07` ausgelöst und liefert den Wert von `DO_S7` (TRUE).
- **EO_S8**: Wird beim Eintritt in `State_08` ausgelöst und liefert den Wert von `DO_S8` (TRUE).

Daten-Eingänge

- Dieser Funktionsblock besitzt keine Daten-Eingänge.

Daten-Ausgänge

- **STATE_NR** (SINT): Aktuelle Zustandsnummer. `START` = 0, `State_01` = 1, `State_02` = 2, ..., `State_08` = 8. Die Werte werden aus der Konstanten-Bibliothek `sequence` geladen.
- **DO_S1** (BOOL): Ist `TRUE`, wenn Zustand `State_01` aktiv ist.
- **DO_S2** (BOOL): Ist `TRUE`, wenn Zustand `State_02` aktiv ist.
- **DO_S3** (BOOL): Ist `TRUE`, wenn Zustand `State_03` aktiv ist.
- **DO_S4** (BOOL): Ist `TRUE`, wenn Zustand `State_04` aktiv ist.
- **DO_S5** (BOOL): Ist `TRUE`, wenn Zustand `State_05` aktiv ist.
- **DO_S6** (BOOL): Ist `TRUE`, wenn Zustand `State_06` aktiv ist.
- **DO_S7** (BOOL): Ist `TRUE`, wenn Zustand `State_07` aktiv ist.
- **DO_S8** (BOOL): Ist `TRUE`, wenn Zustand `State_08` aktiv ist.

Adapter

- Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der `sequence_E_08` ist als Basic-FB mit einer ausführlichen Execution Control Chart (ECC) implementiert. Die Logik folgt einer linearen Kette von Zuständen (`sState_01` bis `sState_08`), einem Start-/Ruhezustand (`sState_00`) und einem expliziten Reset-Zustand (`sRESET`).

Bei jedem Zustandswechsel werden drei Aktionen ausgeführt:

1. **Exit-Aktion des vorherigen Zustands**: Ein `State_XX_X`-Algorithmus setzt den zugehörigen Datenausgang (`DO_Sx`) auf `FALSE`.
2. **Confirmation-Aktion des neuen Zustands**: Ein `State_XX_C`-Algorithmus setzt die `STATE_NR` und löst das `CNF`-Ereignis aus.
3. **Entry-Aktion des neuen Zustands**: Ein `State_XX_E`-Algorithmus setzt den zugehörigen Datenausgang (`DO_Sx`) auf `TRUE` und löst das entsprechende `EO_Sx`-Ereignis aus.

Ein `RESET`-Ereignis führt unmittelbar in den `sRESET`-Zustand. Dort werden über Exit-Algorithmen *alle* potenziell aktiven Ausgänge (`DO_S1` bis `DO_S8`) auf `FALSE` gesetzt. Anschließend erfolgt ein automatischer Übergang (Condition=1, also immer wahr) in den `sState_00` (START).

Technische Besonderheiten

- **Ereignisbasierte Transitionen**: Zustandsübergänge sind strikt an das Eintreffen der definierten Ereignisse gebunden. Es gibt keine zeit- oder datengetriggerten Übergänge.
- **Expliziter Reset-Pfad**: Der Reset-Vorgang läuft über einen eigenen Zustand (`sRESET`), der sicherstellt, dass alle Ausgänge deaktiviert werden, bevor der Startzustand (`sState_00`) erreicht wird. Dies gewährleistet einen sauberen und definierten Neustart.
- **Konsistente Schnittstelle**: Die Namensgebung der Ereignisse (z.B. `S1_S2`) macht die erwartete Transition intuitiv verständlich.
- **Verwendung von Konstanten**: Die Zustandsnummern (`STATE_NR`) werden aus einer zentralen Konstanten-Bibliothek (`sequence`) bezogen, was die Wartbarkeit und Konsistenz verbessert.

Zustandsübersicht

- **sState_00**: Start- bzw. Ruhezustand. `STATE_NR` = 0. Alle Ausgänge sind `FALSE`.
- **sState_01** bis **sState_08**: Aktive Prozesszustände 1 bis 8. Der jeweilige Ausgang `DO_Sx` ist `TRUE`, `STATE_NR` entspricht der Zustandsnummer.
- **sRESET**: Zwischenzustand während eines Reset-Vorgangs. Setzt alle Ausgänge zurück.
- **xSTART**: Initialer ECC-Zustand (wird nur einmal beim Start der Anwendung durchlaufen).

Anwendungsszenarien

- **Schrittkettensteuerungen**: Steuerung von Maschinen mit klar sequentiellen Arbeitsabläufen, wie Pressen, Schweißanlagen oder Verpackungsmaschinen.
- **Handhabungsgeräte**: Steuerung eines Roboterarms, der Teile nacheinander greift, transportiert und ablegt.
- **Teststände**: Automatisierte Abfolge von Test- und Prüfschritten an einem Bauteil.
- **Batch-Prozesse**: Steuerung von Reaktoren oder Mischbehältern, in denen nacheinander verschiedene Zutaten zugegeben oder Prozessparameter geändert werden müssen.

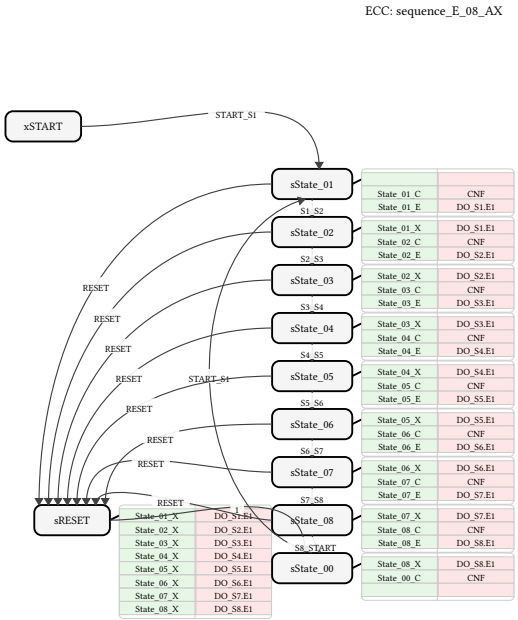
⚖️ Vergleich mit ähnlichen Bausteinen

- **VS. `E_R_TRIG` / `E_F_TRIG`**: Diese Bausteine detektieren Flanken. `sequence_E_08` nutzt solche Flankendetektoren typischerweise als Quelle für seine Transitionsevents, ist selbst aber eine komplexe Zustandsmaschine.
- **VS. `E_DELAY`**: Ein reiner Verzögerungsbaustein. `sequence_E_08` kann durch externe Timer und entsprechende Verknüpfung zeitgesteuerte Abläufe realisieren, bietet aber von Haus aus keine integrierte Zeitsteuerung.
- **VS. `E_SR` (Flip-Flop)**: Ein elementarer Speicherbaustein mit Set/Reset. `sequence_E_08` kann als eine Kette solcher Speicher aufgefasst werden, bei der das Setzen eines Zustands das Zurücksetzen des vorherigen impliziert.
- **VS. SFC (Sequential Function Chart)**: Die Funktionsweise entspricht im Prinzip einer einfachen SFC-Sequenz. `sequence_E_08` kapselt diese SFC-ähnliche Logik in einem wiederverwendbaren FB.

Fazit

Der `sequence_E_08` ist ein robustes und gut strukturiertes Werkzeug zur Implementierung ereignisgesteuerter Schrittketten in IEC 61499. Seine klare Schnittstelle und der explizite Reset-Mechanismus machen ihn zuverlässig und einfach in übergeordnete Steuerungen zu integrieren. Er ist ideal für Anwendungen geeignet, bei denen eine feste, lineare Abfolge von Aktionen durch externe Signale (z.B. Endschalter, Benutzerbefehle) vorangetrieben wird. Für zyklische oder komplex verzweigte Abläufe müsste der Baustein erweitert oder ein anderer Ansatz gewählt werden.

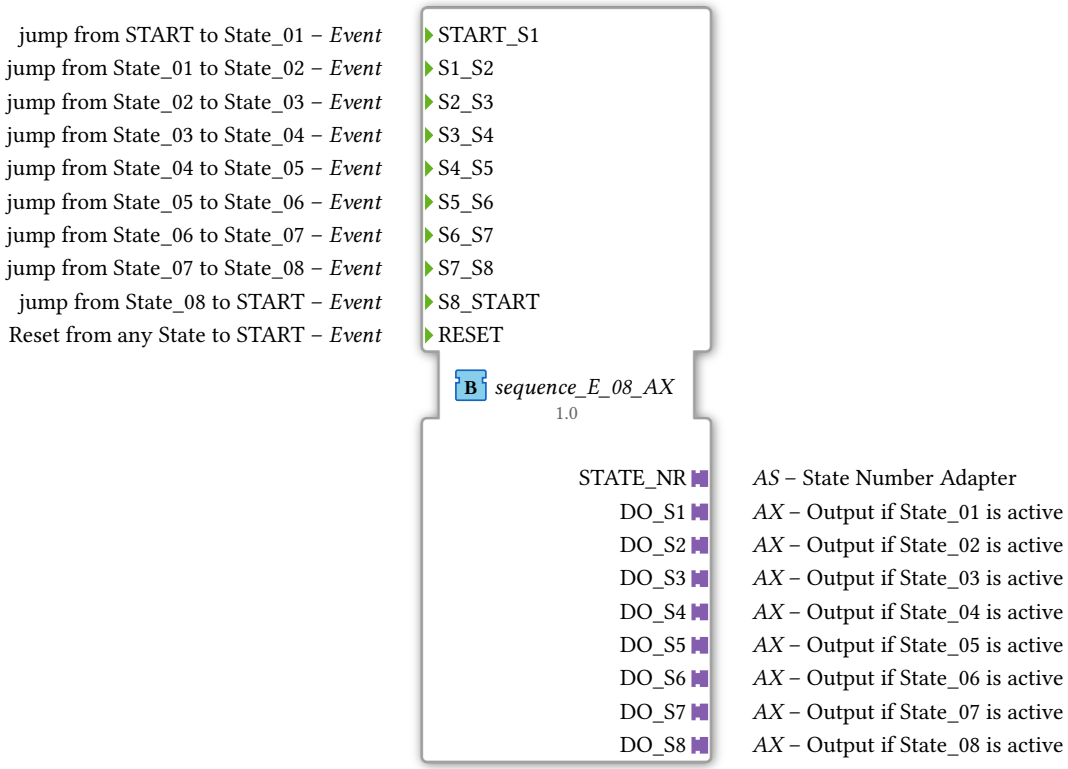
sequence_E_08_AX



• — ○ • — ○ • — ○ •

Einleitung

`sequence_E_08_AX` ist eine Variante des `sequence_E_08`, die zusätzlich Adapter (`AX`) für die Ausgänge verwendet. Er steuert eine rein ereignisgesteuerte Sequenz mit 8 Ausgabezuständen.



Schnittstellenstruktur

Ereignis-Eingänge

- **START_S1**: Startet die Sequenz bei State_01.
- **S1_S2** ... **S7_S8**: Manuelle Übergänge zwischen den Zuständen.
- **S8_START**: Manueller Übergang State_08 -> START.
- **RESET**: Setzt die Sequenz zurück.

Ereignis-Ausgänge

- **CNF**: Bestätigung der Ausführung.

Daten-Eingänge

- Keine.

Daten-Ausgänge

- **STATE_NR** (SINT): Aktuelle Zustandsnummer.

Adapter

- **DO_S1** ... **DO_S8** (adapter::types::unidirectional::AX): Ausgangsadapter für State_01 bis State_08.

Funktionsweise

Entspricht `sequence_E_08`, verwendet jedoch Adapter für die Ausgänge.

Technische Besonderheiten

- Verwendung von `adapter::types::unidirectional::AX`.

Zustandsübersicht

Siehe `sequence_E_08`.

Anwendungsszenarien

Für ereignisgesteuerte 8-stufige Sequenzen mit Adapteranbindung.

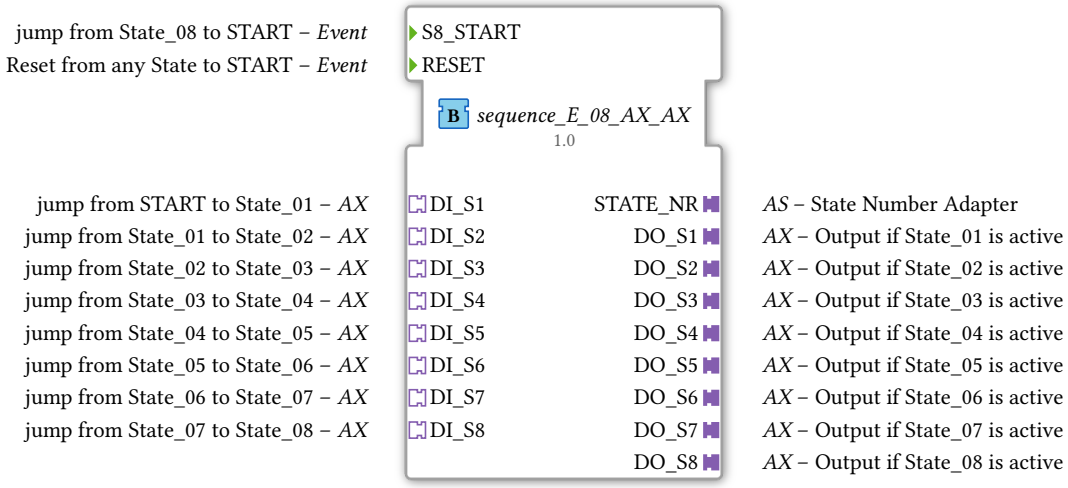
⚖️ Vergleich mit ähnlichen Bausteinen

- **sequence_E_08**: Standardvariante ohne Adapter.

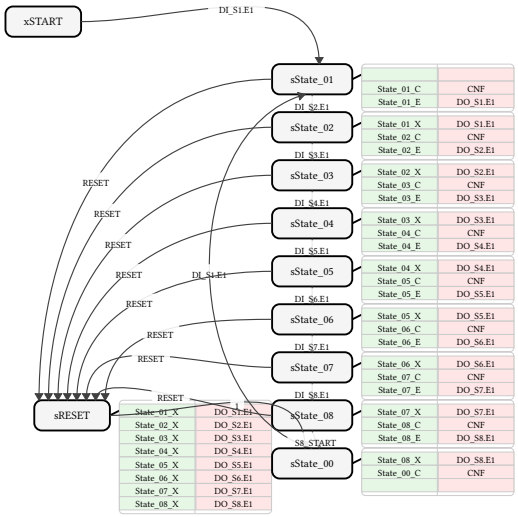
Fazit

Adapter-Variante des 8-Schritt-Ereignis-Sequenzers.

sequence_E_08_AX_AX



ECC: sequence_E_08_AX_AX



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock **sequence_E_08_AX_AX** realisiert eine sequenzielle Ablaufsteuerung mit acht Ausgangsstufen. Er ermöglicht das schrittweise Durchschalten von Zuständen, wobei jeder Zustand durch ein Ereignis über einen AX-Adapter-Eingang verlassen wird. Ein AX-Adapter stellt dabei eine unidirektionale Schnittstelle mit einem Datenwert (`D1`) zur Verfügung, der beim Eintritt in einen Zustand vom Eingangsadapter auf den zugehörigen Ausgangsadapter übertragen wird. Der Block ist für den Einsatz in Automatisierungssystemen konzipiert, die eine klare, ereignisgesteuerte Schrittkette erfordern.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Kommentar
S8_START	Event	Springt von State_08 zurück in den Startzustand START
RESET	Event	Setzt aus jedem beliebigen Zustand zurück in den Startzustand START

Ereignis-Ausgänge

Name	Typ	Kommentar
CNF	Event	Bestätigung der Ausführung (gekoppelt mit STATE_NR)

Daten-Eingänge

Keine (die Zustandsübergänge werden ausschließlich über Ereignisse gesteuert).

Daten-Ausgänge

Name	Typ	Kommentar
STATE_NR	SINT	Aktuelle Zustandsnummer: START = 0, State_01 = 1, ..., State_08 = 8

Adapter

Plugs (Ausgänge – unidirektionaler AX-Adapter)

Name	Type	Kommentar
DO_S1	adapter::types::unidirectional::AX	Ausgang aktiv, wenn State_01 aktiv ist
DO_S2	adapter::types::unidirectional::AX	Ausgang aktiv, wenn State_02 aktiv ist
DO_S3	adapter::types::unidirectional::AX	Ausgang aktiv, wenn State_03 aktiv ist
DO_S4	adapter::types::unidirectional::AX	Ausgang aktiv, wenn State_04 aktiv ist
DO_S5	adapter::types::unidirectional::AX	Ausgang aktiv, wenn State_05 aktiv ist
DO_S6	adapter::types::unidirectional::AX	Ausgang aktiv, wenn State_06 aktiv ist
DO_S7	adapter::types::unidirectional::AX	Ausgang aktiv, wenn State_07 aktiv ist
DO_S8	adapter::types::unidirectional::AX	Ausgang aktiv, wenn State_08 aktiv ist

Sockets (Eingänge – unidirektionaler AX-Adapter)

Name	Type	Kommentar
DI_S1	adapter::types::unidirectional::AX	Springt von START nach State_01
DI_S2	adapter::types::unidirectional::AX	Springt von State_01 nach State_02
DI_S3	adapter::types::unidirectional::AX	Springt von State_02 nach State_03
DI_S4	adapter::types::unidirectional::AX	Springt von State_03 nach State_04
DI_S5	adapter::types::unidirectional::AX	Springt von State_04 nach State_05
DI_S6	adapter::types::unidirectional::AX	Springt von State_05 nach State_06
DI_S7	adapter::types::unidirectional::AX	Springt von State_06 nach State_07
DI_S8	adapter::types::unidirectional::AX	Springt von State_07 nach State_08

Funktionsweise

Der Baustein arbeitet nach dem Prinzip einer ereignisgesteuerten Schrittkette. Nach dem Start befindet er sich im Zustand `xSTART`. Sobald am Socket `DI_S1` ein Ereignis eintrifft, wechselt er in den Zustand `sState_01`. Beim Eintritt in einen Zustand wird der aktuelle Datenwert des zugehörigen Eingangsadapters (`DI_Sx.D1`) auf den entsprechenden Ausgangsadapter (`DO_Sx.D1`) übertragen und gleichzeitig das Ereignis `CNF` mit der aktuellen Zustandsnummer (`STATE_NR`) ausgelöst. Verlässt der Baustein einen Zustand (z.B. durch ein Ereignis am nächsten `DI_Sx`), wird der zugehörige Ausgangsadapter auf `FALSE` gesetzt (Exit-Algorithmus). Der Übergang erfolgt schrittweise: `State_01` → `State_02` → ... → `State_08`. Nach `State_08` wird durch das Ereignis `S8_START` wieder in den Ruhezustand `sState_00` zurückgesprungen. Von dort kann mit `DI_S1` eine neue Sequenz gestartet werden. Das Ereignis `RESET` unterbricht jederzeit die Sequenz, setzt alle Ausgangsadapter auf `FALSE` und kehrt in den Ruhezustand `sState_00` zurück.

Technische Besonderheiten

- **Einsatz von AX-Adaptern**: Alle Ein- und Ausgänge sind als unidirektionale AX-Adapter realisiert, die neben dem Ereigniskanal auch einen Datenwert (`D1`) übertragen. Dadurch kann beim Zustandswechsel nicht nur das Ereignis, sondern auch ein zugehöriger Wert (z.B. ein Sollwert für einen Aktor) weitergegeben werden.
- **Entry-/Exit-Logik**: Durch die strikte Trennung von Eintritts- (`State_xx_E`) und Austrittsaktionen (`State_xx_X`) werden deterministische Zustandswechsel gewährleistet. Der Ausgang eines Zustands wird immer beim Verlassen deaktiviert, um Überschleifen zu vermeiden.
- **Konfigurierbare Startbedingung**: Der Eingang `DI_S1` dient sowohl beim ersten Start als auch nach einem Sequenzdurchlauf als Startbedingung. Der separate Ereigniseingang `S8_START` ermöglicht einen manuellen oder extern getriggerten Rücksprung aus dem letzten Zustand.
- **Zustandsnummerierung**: Der Ausgang `STATE_NR` gibt jederzeit die laufende Zustandsnummer aus, wobei `0` dem Ruhezustand entspricht.

Zustandsübersicht

Zustand (ECC)	Bedeutung	Aktionen
`xSTART`	Initialer Ruhezustand nach Aktivierung	Keine Ausgabe, erwartet `DI_S1`
`sState_01`	Erster Schritt der Sequenz	Setzt `DO_S1.D1` auf `DI_S1.D1`; Ausgabe `STATE_NR=1`
`sState_02`	Zweiter Schritt	Setzt `DO_S2.D1` auf `DI_S2.D1`; `STATE_NR=2`
`sState_03`	Dritter Schritt	Setzt `DO_S3.D1` auf `DI_S3.D1`; `STATE_NR=3`
`sState_04`	Vierter Schritt	Setzt `DO_S4.D1` auf `DI_S4.D1`; `STATE_NR=4`
`sState_05`	Fünfter Schritt	Setzt `DO_S5.D1` auf `DI_S5.D1`; `STATE_NR=5`
`sState_06`	Sechster Schritt	Setzt `DO_S6.D1` auf `DI_S6.D1`; `STATE_NR=6`
`sState_07`	Siebter Schritt	Setzt `DO_S7.D1` auf `DI_S7.D1`; `STATE_NR=7`
`sState_08`	Achter Schritt	Setzt `DO_S8.D1` auf `DI_S8.D1`; `STATE_NR=8`
`sState_00`	Ruhezustand nach Sequenzdurchlauf oder Reset	Keine Ausgabe; `STATE_NR=0`
`sRESET`	Zwischenzustand bei Reset	Setzt **alle** `DO_Sx.D1` auf `FALSE`; danach Übergang zu `sState_00`

Anwendungsszenarien

- **Mehrstufige Förderbandsteuerung**: Jeder Schritt aktiviert einen anderen Förderabschnitt oder eine andere Weiche. Der Datenwert `D1` kann die Geschwindigkeit oder die Fahrtrichtung enthalten.
- **Programmablauf in Bearbeitungsmaschinen**: Z.B. sequentielles Anfahren von Werkzeugen oder Stationen, wobei jeder Schritt einen konfigurierbaren Parameter (z.B. Druck, Temperatur) über den AX-Adapter erhält.
- **Lichtsteuerung mit Szenen**: Acht Schritte können unterschiedliche Beleuchtungsszenen aktivieren, wobei der Datenwert einer Szene (z.B. Helligkeitswerte) vom Eingangsadapter an den Ausgang übergeben wird.
- **Test- und Prüfabläufe**: Automatisierte Testsequenzen mit acht aufeinanderfolgenden Prüfschritten, bei denen die Messwerte des vorherigen Schritts als Sollwert für den nächsten dienen.

Vergleich mit ähnlichen Bausteinen

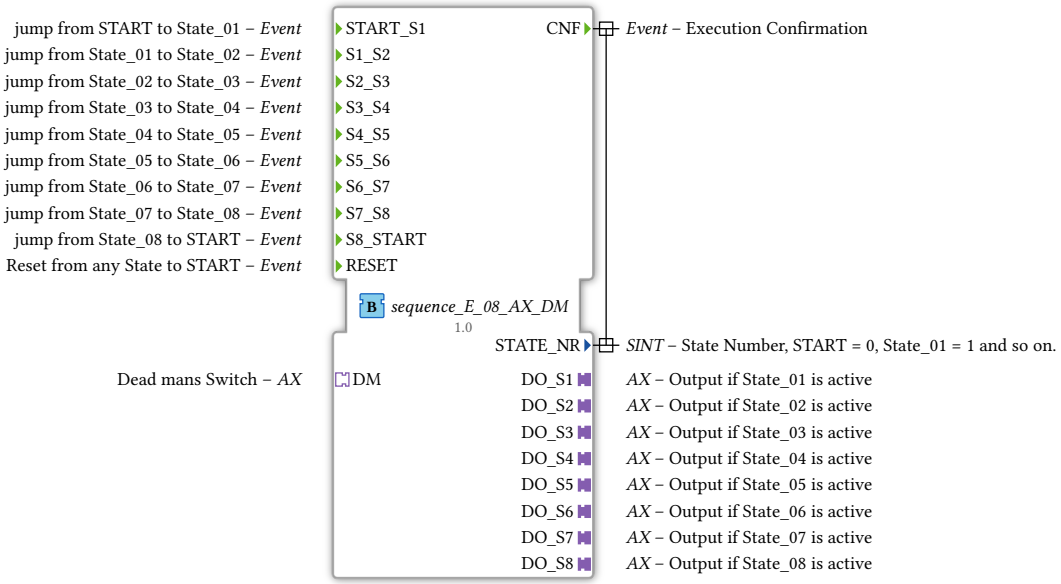
- **sequence_E_08 (ohne AX)**: Ein einfacher sequenzieller Baustein, der nur Boolesche Signale steuert. Der hier beschriebene Baustein erweitert dies um AX-Adapter, wodurch zusätzlich Datenwerte transportiert werden können.
- **sequence_E_08_AX (mit weniger Adaptern)**: Versionen mit weniger als acht Stufen bieten eine kleinere Schrittanzahl, was je nach Anwendung entweder ausreichend oder zu einschränkend sein kann.
- **SFC-Bausteine (Step-Function-Chart)**: Hochsprachliche Bausteine wie `SFC` erlauben parallele Verzweigungen, die dieser einfache Sequenzer nicht abbildet. Dafür ist er deutlich ressourcenschonender und deterministischer in der Ausführung.

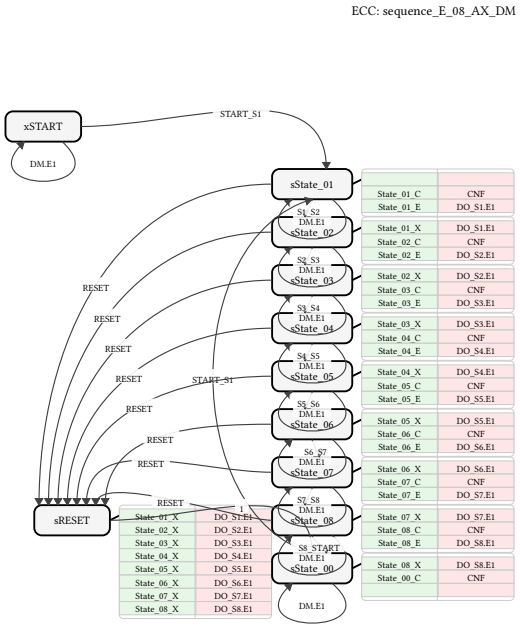
Der vorliegende Baustein stellt einen optimierten Kompromiss zwischen Flexibilität (durch AX-Adapter) und Übersichtlichkeit dar – ideal für Standard-Schrittketten mit Datenweitergabe.

Fazit

Der Funktionsblock `sequence_E_08_AX_AX` ist ein modularer, ereignisgesteuerter Sequenzer mit acht Stufen, der durch den Einsatz von AX-Adaptern eine kompakte Übergabe von Datenwerten zwischen den Schritten erlaubt. Die klare Entry-/Exit-Logik, das separate Reset-Verhalten und die Zustandsnummerierung machen ihn zu einem zuverlässigen Werkzeug für einfache bis mittelschwere Ablaufsteuerungen in der Automatisierungstechnik. Er ist besonders dann vorteilhaft, wenn jeder Schritt nicht nur ein Signal, sondern auch einen parametrierbaren Wert an den Aktor weitergeben muss.

sequence_E_08_AX_DM





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `sequence_E_08_AX_DM` realisiert eine ereignisgesteuerte Ablaufsteuerung mit acht sequenziell zu schaltenden Ausgängen. Er basiert auf einem endlichen Automaten mit neun Zuständen und ermöglicht den Wechsel zwischen den Zuständen durch explizite Ereignisse. Ein integrierter Totmannschalter (Deadman, DM) erlaubt die Überwachung und das kontrollierte Zurücksetzen der Ausgänge. Der FB ist speziell für den Einsatz in sicherheitskritischen oder überwachten Steuerungsabläufen in der Agrartechnik konzipiert.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Beschreibung
`START_S1`	Wechsel vom Startzustand zu Zustand 1 (State_01)
`S1_S2`	Wechsel von Zustand 1 zu Zustand 2 (State_02)
`S2_S3`	Wechsel von Zustand 2 zu Zustand 3 (State_03)
`S3_S4`	Wechsel von Zustand 3 zu Zustand 4 (State_04)
`S4_S5`	Wechsel von Zustand 4 zu Zustand 5 (State_05)
`S5_S6`	Wechsel von Zustand 5 zu Zustand 6 (State_06)
`S6_S7`	Wechsel von Zustand 6 zu Zustand 7 (State_07)
`S7_S8`	Wechsel von Zustand 7 zu Zustand 8 (State_08)
`S8_START`	Wechsel von Zustand 8 zum Startzustand (State_00)
`RESET`	Rücksetzen aus jedem Zustand in den Startzustand (State_00)

Ereignis-Ausgänge

Ereignis	Mit Variable	Beschreibung
CNF	STATE_NR	Bestätigung der Zustandsänderung; liefert gleichzeitig die aktuelle Zustandsnummer

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Variable	Typ	Beschreibung
STATE_NR	SINT	Aktuelle Zustandsnummer (0 = State_00, 1 = State_01, ..., 8 = State_08)

Adapter

Richtung	Name	Typ	Beschreibung
Plug	DO_S1	unidirectional::AX	Ausgang für Zustand 1 (State_01 aktiv)
Plug	DO_S2	unidirectional::AX	Ausgang für Zustand 2 (State_02 aktiv)
Plug	DO_S3	unidirectional::AX	Ausgang für Zustand 3 (State_03 aktiv)
Plug	DO_S4	unidirectional::AX	Ausgang für Zustand 4 (State_04 aktiv)
Plug	DO_S5	unidirectional::AX	Ausgang für Zustand 5 (State_05 aktiv)
Plug	DO_S6	unidirectional::AX	Ausgang für Zustand 6 (State_06 aktiv)
Plug	DO_S7	unidirectional::AX	Ausgang für Zustand 7 (State_07 aktiv)
Plug	DO_S8	unidirectional::AX	Ausgang für Zustand 8 (State_08 aktiv)
Socket	DM	unidirectional::AX	Totmannschalter (Deadman); liefert das Ereignis DM.E1 und den Datenwert DM.D1

Funktionsweise

Der Funktionsblock arbeitet als endlicher Automat mit folgenden Zuständen: xSTART (Initial), sState_01 bis sState_08 (aktive Sequenzzustände), sState_00 (Start-/Wartezustand nach Sequenzende) und sRESET (Zwischenzustand für Rücksetzung).

- **Start und Ablauf**: Nach dem Start befindet sich der Automat im Zustand xSTART. Durch das Ereignis START_S1 wird in den Zustand sState_01 gewechselt. Von dort aus führen die Ereignisse S1_S2, S2_S3, ... bis S7_S8 sequenziell durch die acht Zustände. Der letzte Zustand sState_08 wird durch S8_START in den Zustand sState_00 überführt, der als Ruhepunkt nach der Sequenz dient und wiederholt über START_S1 gestartet werden kann.
- **Ausgänge**: Jeder Zustand aktiviert den zugehörigen Plug-Adapter (DO_Sx). Beim Eintritt in einen Zustand wird der Datenwert des Deadman-Adapters (DM.D1) an den Ausgangsadapter weitergegeben (DO_Sx.D1 := DM.D1). Beim Verlassen eines Zustands (Exit) wird der Ausgang deaktiviert (DO_Sx.D1 := FALSE).
- **Totmannschalter (Deadman)**: Das Ereignis DM.E1 löst eine Selbsttransition im aktuellen Zustand aus (z. B. sState_01 → sState_01). Dies führt zur erneuten Ausführung der Exit- und Entry-Aktionen, sodass der aktuelle Wert von DM.D1 aktualisiert auf den Ausgangsadapter übertragen wird. Solange der Deadman aktiv ist (d. h. DM.E1 wiederholt eintrifft), bleibt der Ausgangswert auf dem aktuellen DM.D1-Stand. Falls der Deadman nicht mehr getriggert wird, verbleibt der Zustand, bis ein normales Sequenzereignis oder RESET eintritt.
- **Rücksetzen**: Das Ereignis RESET führt aus jedem Zustand (außer xSTART und sRESET) in den Zwischenzustand sRESET. Dort werden alle acht Ausgänge durch Aufruf der Exit-Algorithmen ausgeschaltet. Anschließend wird automatisch (Transition mit 1) in den Zustand sState_00 gewechselt. Von dort kann die Sequenz erneut mit START_S1 gestartet werden.

- **Bestätigung**: Bei jedem Zustandswechsel (außer innerhalb von sRESET) wird das Ereignis CNF mit der aktuellen Zustandsnummer STATE_NR ausgegeben. Die Zustandsnummer wird über die Konstanten sequence::State_00 bis sequence::State_08 gesetzt.

Technische Besonderheiten

- **Adapterbasierte Ausgänge**: Alle acht Ausgänge sowie der Totmannschalter sind als unidirektionale AX-Adapter implementiert. Dies ermöglicht eine flexible Anbindung an externe Hardware (z. B. analoge oder digitale Aktoren) über den Adapterrahmen.
- **Wiederverwendbare Sequenzkonstanten**: Die Zustandsnummern werden aus einer separaten Bibliothek (logiBUS::utils::sequence::const::sequence) bezogen, was eine konsistente Nummerierung über verschiedene Sequenzbausteine hinweg erlaubt.
- **Deadman-Integration**: Der Totmannschalter wirkt nicht als Sperre, sondern als dynamischer Wertegeber für die Ausgänge. Jeder Zustand übernimmt bei Eintritt den aktuellen Wert von DM.D1 und kann durch wiederholtes DM.E1-Ereignis aktualisiert werden.
- **Explizites Rücksetzen**: Das RESET-Ereignis deaktiviert sofort alle Ausgänge und kehrt in einen definierten Startzustand zurück – eine wichtige Sicherheitsfunktion.

Zustandsübersicht

Zustand	Bezeichnung	Ausgang aktiv	Transitionen
xSTART	Initialzustand	keiner	→ sState_01 bei START_S1; Selbsttransition bei DM.E1
sState_01	Sequenzschritt 1	DO_S1	→ sState_02 bei S1_S2; Selbsttrans. bei DM.E1; → sRESET bei RESET
sState_02	Sequenzschritt 2	DO_S2	→ sState_03 bei S2_S3; Selbsttrans. bei DM.E1; → sRESET bei RESET
sState_03	Sequenzschritt 3	DO_S3	→ sState_04 bei S3_S4; Selbsttrans. bei DM.E1; → sRESET bei RESET
sState_04	Sequenzschritt 4	DO_S4	→ sState_05 bei S4_S5; Selbsttrans. bei DM.E1; → sRESET bei RESET
sState_05	Sequenzschritt 5	DO_S5	→ sState_06 bei S5_S6; Selbsttrans. bei DM.E1; → sRESET bei RESET
sState_06	Sequenzschritt 6	DO_S6	→ sState_07 bei S6_S7; Selbsttrans. bei DM.E1; → sRESET bei RESET
sState_07	Sequenzschritt 7	DO_S7	→ sState_08 bei S7_S8; Selbsttrans. bei DM.E1; → sRESET bei RESET
sState_08	Sequenzschritt 8	DO_S8	→ sState_00 bei S8_START; Selbsttrans. bei DM.E1; → sRESET bei RESET
sState_00	Ruhezustand (nach Sequenz)	keiner	→ sState_01 bei START_S1; Selbsttrans. bei DM.E1
sRESET	Rückstellzustand	alle deaktiviert	Automatisch → sState_00

Anwendungsszenarien

- **Landwirtschaftliche Steuerungen**: Schrittweises Ansteuern von acht Ventilen, Antrieben oder Beleuchtungseinheiten, z. B. für Bewässerungssequenzen oder Erntemaschinen.
- **Sicherheitsüberwachte Prozesse**: Einsatz in Anlagen, die einen Totmannschalter erfordern – der Bediener muss durch wiederholtes Betätigen des Deadman die Ausgänge aktiv halten.
- **Test- und Prüfstände**: Sequenzielle Aktivierung von Prüfschritten mit manueller Freigabe durch den Bediener (über die Ereignisse).
- **Modulare Ablaufsteuerung**: Kombination mehrerer sequence_E_08_AX_DM-Bausteine für umfangreichere Sequenzen mit mehr als acht Schritten.

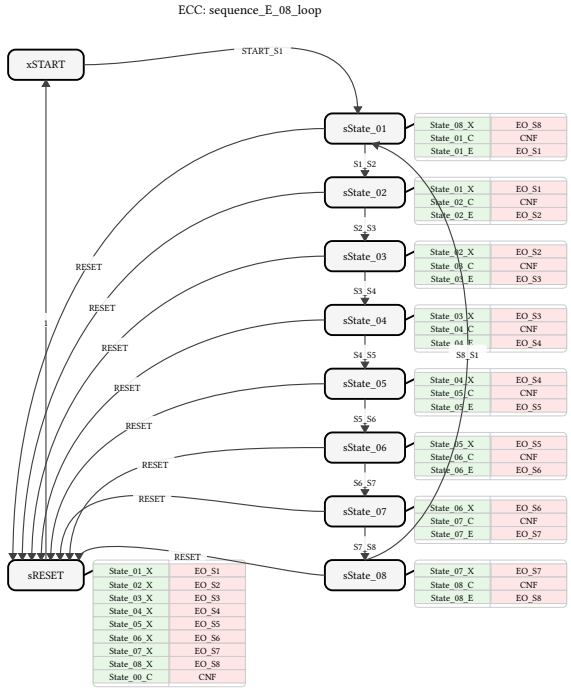
Vergleich mit ähnlichen Bausteinen

- **`sequence_E_08_AX` (ohne Deadman)**: Besitzt keinen Totmannschalter – die Ausgänge werden beim Eintritt dauerhaft aktiviert (z. B. mit `TRUE`) und nur beim Verlassen deaktiviert. Geeignet für unkritische Steuerungen.
- **`sequence_E_04_AX_DM`**: Vierstufige Variante mit entsprechend weniger Ausgängen und Ereignissen. Bietet identische Deadman-Funktionalität.
- **`sequence_T_08_AX_DM`**: Zeitgesteuerte Variante – die Zustandsübergänge erfolgen über Timer anstatt über externe Ereignisse. Vergleichbare Deadman-Integration.

Fazit

Der Funktionsblock `sequence_E_08_AX_DM` ist eine leistungsfähige und sichere Komponente für ereignisgesteuerte Abläufe mit bis zu acht Schritten. Die Integration eines Totmannschalters erhöht die Betriebssicherheit, indem sie erzwungene Bedienerinteraktion verlangt. Die adapterbasierte Ausgangsschnittstelle erlaubt eine einfache Anbindung an unterschiedliche Aktorik. Dank der klar strukturierten Zustandsmaschine und der expliziten Rücksetzmöglichkeit eignet sich dieser Baustein besonders für sicherheitsrelevante Steuerungsaufgaben in der Agrartechnik und darüber hinaus.

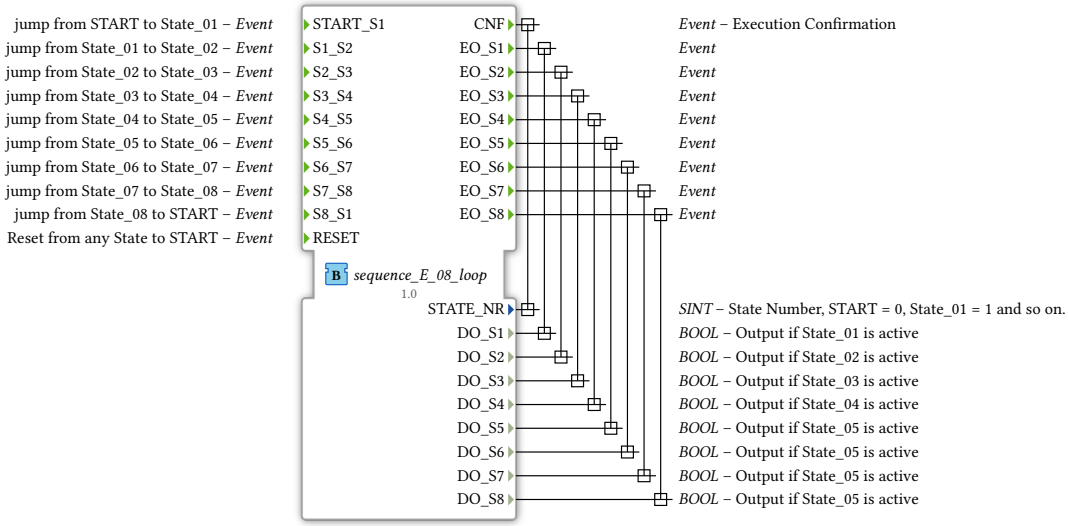
sequence_E_08_loop



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `sequence_E_08_loop` ist ein Sequenzer mit acht Ausgangszuständen, der in einer Schleife arbeitet. Er dient zur Steuerung sequentieller Abläufe, bei denen jeder Schritt durch ein externes Ereignis ausgelöst wird. Der Baustein ist als Basic Function Block (BasicFB) nach IEC 61499 implementiert und eignet sich für Anwendungen, die eine klare, ereignisgesteuerte Zustandsmaschine erfordern.



Schnittstellenstruktur

Ereignis-Eingänge

- `START_S1`: Wechselt vom Startzustand (`START`) in den Zustand `State_01`.
- `S1_S2`: Wechselt von `State_01` zu `State_02`.
- `S2_S3`: Wechselt von `State_02` zu `State_03`.
- `S3_S4`: Wechselt von `State_03` zu `State_04`.
- `S4_S5`: Wechselt von `State_04` zu `State_05`.
- `S5_S6`: Wechselt von `State_05` zu `State_06`.
- `S6_S7`: Wechselt von `State_06` zu `State_07`.
- `S7_S8`: Wechselt von `State_07` zu `State_08`.
- `S8_S1`: Wechselt von `State_08` zurück zum Zustand `State_01` (Schleife).
- `RESET`: Setzt von jedem beliebigen Zustand zurück in den Startzustand (`START`).

Ereignis-Ausgänge

- `CNF` (Execution Confirmation): Wird bei jedem Zustandswechsel ausgelöst und liefert die aktuelle Zustandsnummer (`STATE_NR`).
- `EO_S1` ... `EO_S8`: Werden beim Eintritt in den jeweiligen Zustand (`State_01` bis `State_08`) ausgelöst und liefern den zugehörigen booleschen Datenausgang (`DO_S1` ... `DO_S8`).

Daten-Eingänge

- Keine vorhanden.

Daten-Ausgänge

- `STATE_NR` (SINT): Die Nummer des aktiven Zustands. `START` = 0, `State_01` = 1, `State_02` = 2, usw.
- `DO_S1` ... `DO_S8` (BOOL): Logische Ausgänge, die `TRUE` sind, wenn der entsprechende Zustand aktiv ist.

Adapter

- Keine Adapterschnittstellen vorhanden.

Funktionsweise

Der FB arbeitet als ereignisgesteuerte Zustandsmaschine (ECC). Der initiale Zustand ist `xSTART`. Ein eingehendes Ereignis an einem der benannten Ereigniseingänge (z.B. `START_S1`) bewirkt einen Übergang in den nächsten Zustand (z.B. `sState_01`).

Bei jedem Zustandswechsel werden drei Aktionen ausgeführt:

1. **Exit-Aktion (X) des vorherigen Zustands**: Setzt den zugehörigen Datenausgang (`DO_Sx`) auf `FALSE`.
2. **Confirmation-Aktion (C) des neuen Zustands**: Setzt die Zustandsnummer `STATE_NR` und löst das `CNF`-Ereignis aus.
3. **Entry-Aktion (E) des neuen Zustands**: Setzt den zugehörigen Datenausgang (`DO_Sx`) auf `TRUE` und löst das entsprechende Ereignis `EO_Sx` aus.

Die Sequenz durchläuft die Zustände 1 bis 8 linear. Von `State_08` aus führt das Ereignis `S8_S1` zurück zu `State_01`, wodurch eine Endlosschleife entsteht. Das `RESET`-Ereignis führt über einen dedizierten Reset-Zustand (`sRESET`) immer zurück in den initialen Zustand `xSTART` und setzt dabei alle aktiven Ausgänge zurück.

Technische Besonderheiten

- **Reiner Event-Trigger**: Zustandsübergänge erfolgen ausschließlich durch externe Ereignisse, nicht durch Bedingungen oder Zeitgeber.
- **Explizite Reset-Logik**: Der Reset-Vorgang nutzt einen eigenen Zustand (`sRESET`), in dem alle möglichen Exit-Aktionen aufgerufen werden, um sicherzustellen, dass alle Ausgänge deaktiviert werden, bevor der Startzustand erreicht wird.
- **Zustandsbestätigung**: Das `CNF`-Ereignis und der `STATE_NR`-Ausgang bieten eine klare Rückmeldung über den aktuellen Systemstatus.
- **Konsistente Namensgebung**: Die Namen der Ereignisse, Zustände und Algorithmen folgen einem durchgängigen Schema (z.B. `S1_S2`, `sState_01`, `State_01_C`), was die Lesbarkeit und Wartung erleichtert.

Zustandsübersicht

Die ECC (Execution Control Chart) umfasst folgende Zustände:

- `xSTART`: Initialer, inaktiver Wartezustand (Zustandsnummer 0).
- `sState_01` bis `sState_08`: Die acht aktiven Sequenzzustände (Zustandsnummern 1-8). Jeder aktiviert seinen spezifischen Ausgang.
- `sRESET`: Dedizierter Reset-Zustand. Beim Eintreten werden alle potenziell aktiven Ausgänge (`DO_S1` bis `DO_S8`) deaktiviert, die Zustandsnummer auf 0 gesetzt und ein Übergang zu `xSTART` ausgeführt.

Anwendungsszenarien

- Steuerung von zyklischen Fertigungs- oder Montageprozessen mit manueller oder sensorischer Freigabe für den nächsten Schritt.
- Steuerung von Testsequenzen, bei denen jeder Testschritt manuell gestartet wird.
- Überwachung und Steuerung von Batch-Prozessen, bei denen ein Operator die nächste Phase einleitet.
- Als zentraler Steuerbaustein in Maschinen, die einen klar definierten, schrittweisen Arbeitsablauf haben.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Sequenzern mit zeitgesteuerten Übergängen (z.B. `sequence_T_08_loop`) bietet dieser FB maximale Flexibilität, da die Verweildauer in jedem Zustand extern vorgegeben wird. Er ist einfacher aufgebaut als Bausteine mit integrierter Fehlerbehandlung oder komplexen Verzweigungen, stellt aber eine robuste Grundlage für ereignisgesteuerte Abläufe dar. Alternative Implementierungen mit `E_SR`- oder `E_CTUD`-Blöcken wären deutlich komplexer und weniger übersichtlich.

🛠️ Zugehörige Übungen

- Uebung_040
- Uebung_040_2
- Uebung_041

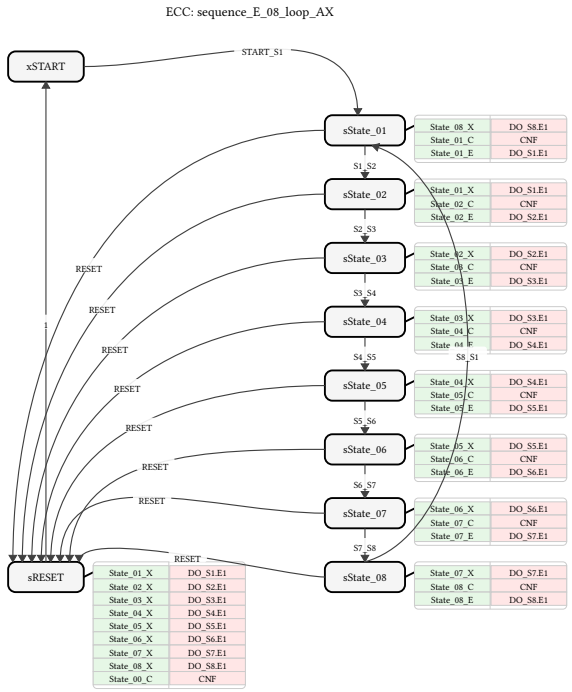
Fazit

Der `sequence_E_08_loop` ist ein spezialisierter, zuverlässiger und einfach zu konfigurierender Funktionsblock für ereignisgesteuerte Sequenzabläufe. Seine klare Struktur, die konsistente Schnittstelle und die integrierte Reset-Funktionalität machen ihn besonders geeignet für Anwendungen, bei denen ein schrittweiser Prozess durch externe Signale (z.B. Taster, Sensoren, übergeordnete Steuerungen) vorangetrieben werden muss. Die Ausgabe der Zustandsnummer und die konfirmierenden Ereignisse ermöglichen eine einfache Anbindung an Visualisierungs- und Überwachungssysteme.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de

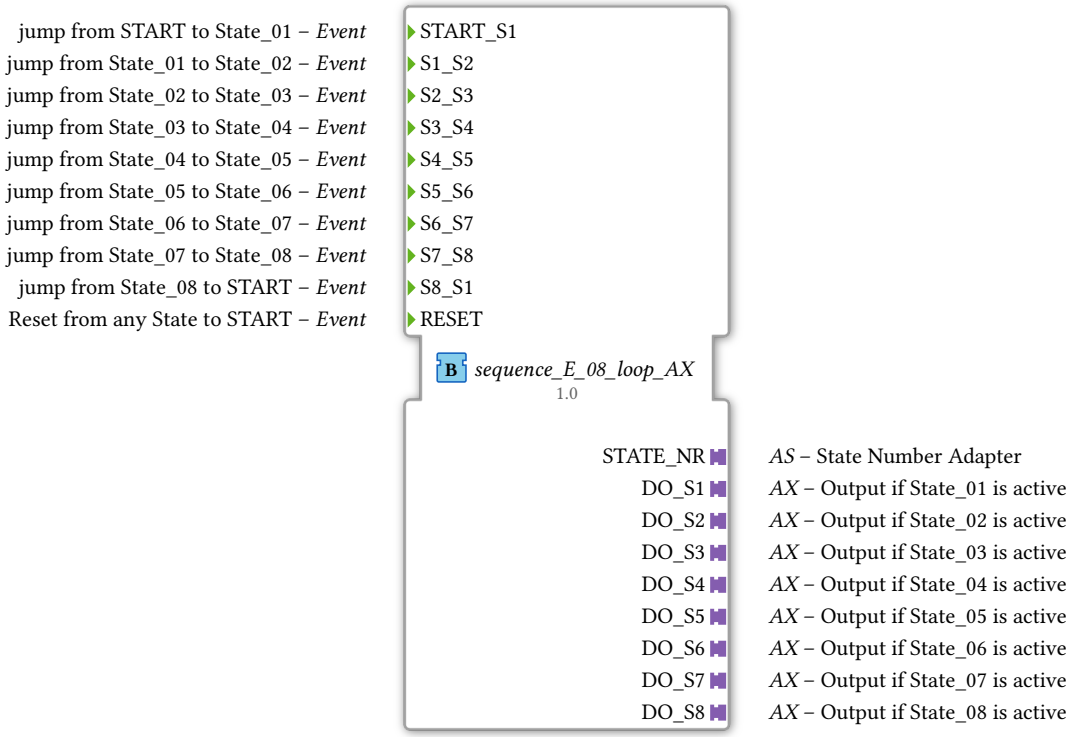
sequence_E_08_loop_AX



• — ○ • — ○ • — ○ •

Einleitung

`sequence_E_08_loop_AX` ist eine Variante des `sequence_E_08_loop`, die zusätzlich Adapter (`AX`) für die Ausgänge verwendet. Er steuert eine rein ereignisgesteuerte, zyklische Sequenz mit 8 Ausgabezuständen.



Schnittstellenstruktur

Ereignis-Eingänge
- **START_S1**: Startet die Sequenz bei State_01.
- **S1_S2** ... **S7_S8**: Manuelle Übergänge zwischen den Zuständen.
- **S8_S1**: Übergang State_08 -> State_01 (Loop).
- **RESET**: Setzt die Sequenz zurück.

Ereignis-Ausgänge
- **CNF**: Bestätigung der Ausführung.

Daten-Eingänge
- Keine.

Daten-Ausgänge
- **STATE_NR** (SINT): Aktuelle Zustandsnummer.

Adapter
- **DO_S1** ... **DO_S8** (adapter::types::unidirectional::AX): Ausgangsadapter für State_01 bis State_08.

Funktionsweise
Entspricht `sequence_E_08_loop`, verwendet jedoch Adapter für die Ausgänge.

Technische Besonderheiten
- Verwendung von `adapter::types::unidirectional::AX`.

Zustandsübersicht
Siehe `sequence_E_08_loop`.

Anwendungsszenarien
Für zyklische ereignisgesteuerte 8-stufige Sequenzen mit Adapteranbindung.

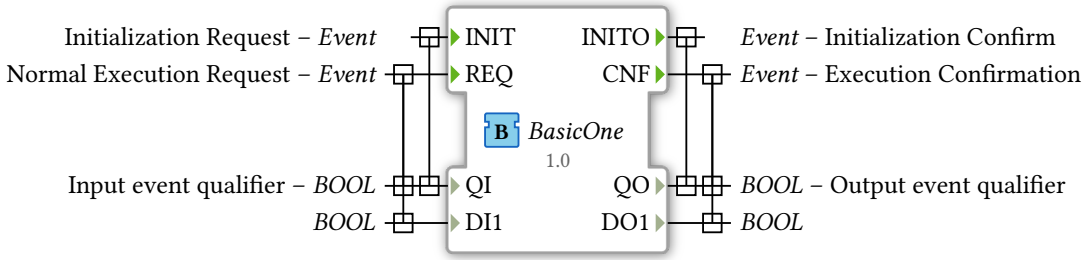
⚖️ Vergleich mit ähnlichen Bausteinen
- **sequence_E_08_loop**: Standardvariante ohne Adapter.

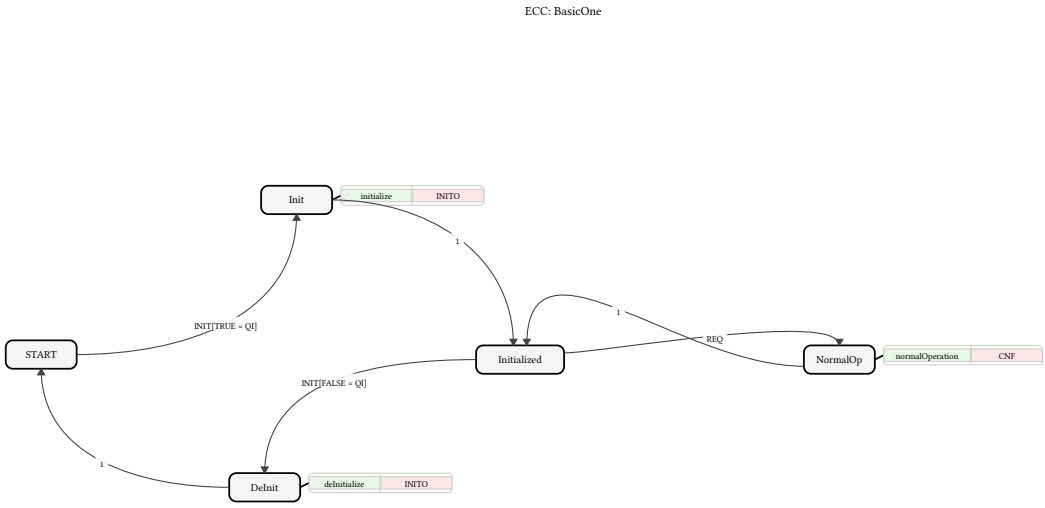
🛠️ Zugehörige Übungen
- Uebung_040_AX

Fazit
Adapter-Variante des 8-Schritt-Loop-Ereignis-Sequenzers.

Numerischer Index (sequence)
Hier finden Sie alle Übungen des Arbeitsbereichs `sequence` in aufsteigender Reihenfolge.

BasicOne





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **BasicOne** ist ein grundlegender, ereignisgesteuerter Baustein nach IEC 61499. Er dient als einfaches Sequenzelement zur Initialisierung, Ausführung einer Hauptoperation und ordnungsgemäßen Deinitialisierung. Der Baustein eignet sich besonders für Steuerungsabläufe, bei denen ein Ressourcenzustand (z. B. Gerät bereit) überwacht und zurückgesetzt werden muss. Der Baustein ist im Paket `logiBUS::utils::sequence::one` enthalten.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Mitgeführte Daten	Beschreibung
`INIT`	EInit	`QI`	Initialisierungsanforderung; die Transition hängt vom Wert des Qualifiers `QI` ab.
`REQ`	Event	`QI`, `DI1`	Normale Ausführungsanforderung; startet die Hauptoperation.

Ereignis-Ausgänge

Ereignis	Typ	Mitgeführte Daten	Beschreibung
`INITO`	EInit	`QO`	Bestätigung einer erfolgreichen Initialisierung oder Deinitialisierung.
`CNF`	Event	`QO`, `DO1`	Bestätigung der normalen Ausführung; gibt den aktuellen Zustand aus.

Daten-Eingänge

Name	Typ	Beschreibung
`QI`	BOOL	Eingangsqualifier – steuert, ob eine Initialisierung aktiv (`TRUE`) oder deinitialisierend (`FALSE`) ist. Beeinflusst auch die Ausführung der Hauptoperation.

DI1	BOOL	Erster Dateneingang; wird nur bei aktiver Initialisierung (QI = TRUE) während der normalen Operation auf den Ausgang DO1 übertragen.

Daten-Ausgänge

Name	Typ	Beschreibung
QO	BOOL	Ausgangsqualifier – spiegelt nach einer Aktion den Wert von QI wider (außer nach Deinitialisierung: dann FALSE).
DO1	BOOL	Erster Datenausgang; enthält bei einer normalen Operation den Wert von DI1, sofern QI = TRUE, ansonsten FALSE.

Adapter

Keine.

Funktionsweise

Der Baustein durchläuft einen klar definierten Lebenszyklus:

1. **Startzustand (START)**: Nach dem Hochlauf wartet der Baustein auf ein INIT-Ereignis.
2. **Initialisierung (Init)**: Wird INIT mit QI = TRUE empfangen, führt der Baustein den Algorithmus *initialize* aus. Dabei wird QO := QI gesetzt (also TRUE). Anschließend wird INITO gesendet und der Baustein wechselt in den Zustand **Initialized**.
3. **Normalbetrieb (NormalOp)**: Im Zustand **Initialized** kann durch ein REQ-Ereignis die normale Operation gestartet. Der Algorithmus *normalOperation* setzt QO := QI (immer noch TRUE) und überträgt – wenn QI = TRUE ist – den Wert von DI1 nach DO1. Danach wird CNF gesendet und der Baustein kehrt automatisch nach **Initialized** zurück.
4. **Deinitialisierung (DeInit)**: Wird im Zustand **Initialized** ein INIT-Ereignis mit QI = FALSE empfangen, führt der Baustein den Algorithmus *deInitialize* aus. Dieser setzt QO := FALSE und DO1 := FALSE. Anschließend wird INITO gesendet und der Baustein springt zurück in den Startzustand **START**.

Wichtig: Die normale Operation wird nur dann ausgeführt, wenn QI = TRUE ist. Bei QI = FALSE im REQ-Ereignis wird zwar der Algorithmus durchlaufen, aber DO1 bleibt unverändert (er wird nicht auf DI1 gesetzt; stattdessen bleibt der letzte Wert oder der Standardwert FALSE gemäß Algorithmus – bei QI = FALSE wird die IF-Bedingung nicht erfüllt, daher ändert sich DO1 nicht).

Technische Besonderheiten

- **Ereignisabhängigkeit von QI**: Die Transition INIT[TRUE = QI] bzw. INIT[FALSE = QI] zeigt, dass das gleiche Ereignis INIT je nach Wert von QI unterschiedliche Zustandsübergänge auslöst. Dies ermöglicht eine kompakte Steuerung von Initialisierung und Deinitialisierung über ein einziges Ereignis.
- **Automatischer Rückfall**: Nach einem REQ-Ereignis kehrt der Baustein ohne externes Ereignis in den Zustand **Initialized** zurück (Transition mit Bedingung 1). Dies vereinfacht die Anbindung an zyklische Abläufe.
- **Keine Selbsthemmung**: Der Baustein blockiert nicht; nach jeder erfolgreichen Ausführung ist er bereit für das nächste Ereignis.

Zustandsübersicht

Zustand	Beschreibung	Mögliche Aktionen
START	Initialer Ruhezustand nach Systemstart oder nach einer Deinitialisierung.	Wartet auf INIT mit QI = TRUE.
Init	Initialisierungsphase; setzt QO = QI.	Führt Algorithmus *initialize* aus, sendet INITO.
Initialized	Wartezustand nach erfolgreicher Initialisierung.	Kann auf REQ (startet NormalOp) oder auf INIT mit QI = FALSE (startet DeInit) reagieren.

`NormalOp`	Ausführung der Hauptoperation; überträgt `DI1` nach `DO1` (nur wenn `QI = TRUE`).	Führt Algorithmus *normalOperation* aus, sendet `CNF`.
`DeInit`	Deinitialisierungsphase; setzt `QO = FALSE` und `DO1 = FALSE`.	Führt Algorithmus *deInitialize* aus, sendet `INITO`.

Anwendungsszenarien

- **Initialisierung einer Komponente**: Ein Gerät muss beim Einschalten konfiguriert werden. Mit `INIT` (QI=TRUE) wird der Gerätezustand auf „bereit“ gesetzt, `REQ` führt dann zyklisch die eigentliche Logik aus.
- **Ressourcenverwaltung**: Der Baustein kann als einfacher Sequenzer für eine einmalige Initialisierung und spätere Deinitialisierung verwendet werden (z. B. für eine Datenbankverbindung).
- **Sicherheitsorientierte Steuerung**: Durch den Qualifier `QI` kann die Ausgabe `DO1` nur dann den Eingangswert annehmen, wenn der Baustein initialisiert ist. Bei Deinitialisierung werden alle Ausgänge zurückgesetzt (`FALSE`).

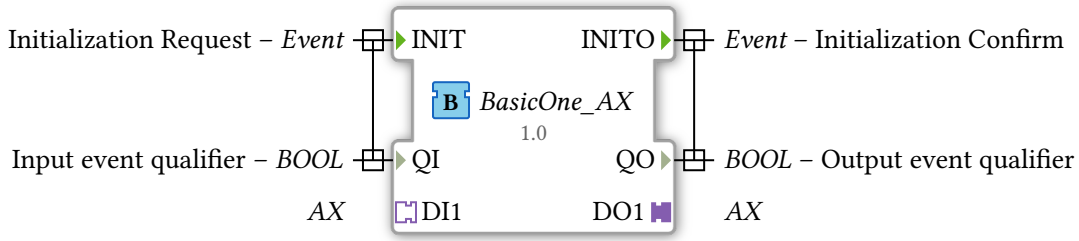
Vergleich mit ähnlichen Bausteinen

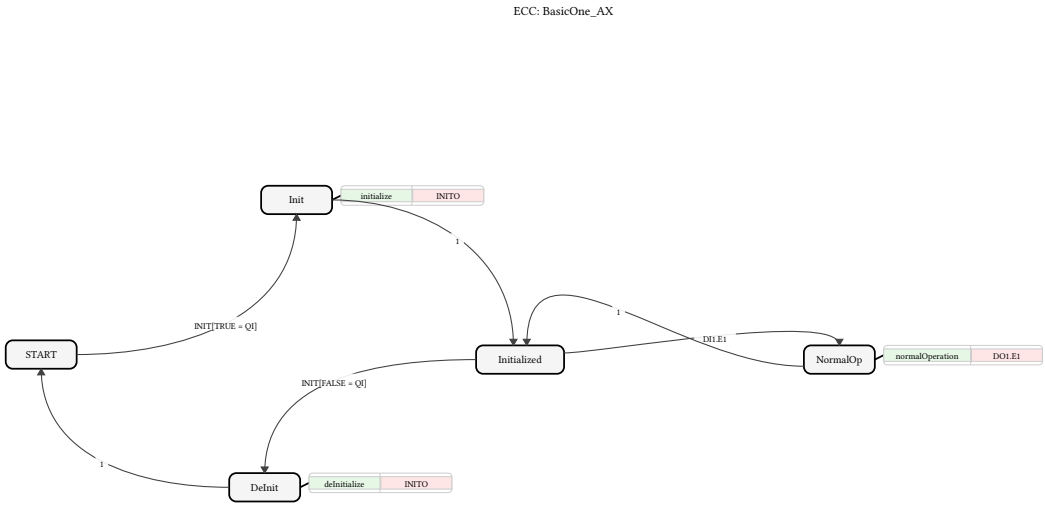
- **BasicBOOLEAN**: Ein einfacher Boolescher Baustein ohne Zustandsautomat, der nur auf ein Ereignis reagiert und einen Wert weitergibt. `BasicOne` bietet dagegen eine explizite Initialisierungs- und Deinitialisierungssequenz.
- **SR-Flipflop**: Ein Speicherbaustein mit Setzen und Rücksetzen. `BasicOne` ist eher ein Zustandsautomat, der einen einmaligen Start und Stopp abbildet, jedoch keine Haltefunktion besitzt.
- **SimpleCycle**: Ein zyklischer Baustein, der immer wieder die gleiche Operation ausführt. `BasicOne` unterscheidet zwischen Initialisierung und zyklischer Ausführung und erlaubt eine saubere Trennung.

Fazit

Der Funktionsbaustein `BasicOne` ist ein hervorragendes Basiselement für Steuerungsprojekte, die eine kontrollierte Initialisierung, einen gesicherten Betrieb und eine definierte Deinitialisierung erfordern. Seine einfache Zustandsmaschine und die Abhängigkeit vom Qualifier `QI` machen ihn flexibel einsetzbar, ohne unnötige Komplexität. Er eignet sich sowohl für Einsteiger in die IEC-61499-Welt als auch für erfahrene Entwickler, die klare, wiederverwendbare Ablaufbausteine benötigen.

BasicOne_AX





• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `BasicOne_AX` ist ein grundlegender IEC 61499-Baustein des Typs *Basic Function Block*, der als Vorlage für einfache Steuerungsaufgaben dient. Er realisiert eine initialisierbare Durchschaltung eines booleschen Signals unter Verwendung unidirektionaler Adapter-Schnittstellen vom Typ `AX`. Der Baustein eignet sich für Szenarien, in denen ein Eingangssignal auf einen Ausgang kopiert und der Initialisierungszustand überwacht werden soll.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Mit Variablen	Kommentar
`INIT`	`EInit`	`QI`	Initialisierungsanforderung

Ereignis-Ausgänge

Ereignis	Typ	Mit Variablen	Kommentar
`INIT0`	`EInit`	`QO`	Initialisierungsbestätigung

Daten-Eingänge

Variable	Typ	Kommentar
`QI`	`BOOL`	Eingangsqualifizierer (Schaltet die Initialisierung ein/aus)

Daten-Ausgänge

Variable	Typ	Kommentar
`QO`	`BOOL`	Ausgangsqualifizierer (Spiegelt den Initialisierungszustand)

Adapter

Richtung	Name	Typ	Kommentar
Plug (Ausgang)	DO1	adapter::types::unidirectional::AX	Unidirektionaler Adapter zur Datenweitergabe (AX-Schnittstelle)
Socket (Eingang)	DI1	adapter::types::unidirectional::AX	Unidirektionaler Adapter zum Empfang von Daten (AX-Schnittstelle)

Funktionsweise

Der Funktionsblock durchläuft einen endlichen Automaten (ECC) mit den Zuständen `START`, `Init`, `Initialized`, `NormalOp` und `DeInit`.

- **START → Init**: Wird beim Eintreffen des Ereignisses `INIT` mit `QI = TRUE` durchlaufen. Der Algorithmus `initialize` setzt `QO` auf den Wert von `QI` (also `TRUE`). Der Ausgang `INITO` signalisiert die erfolgreiche Initialisierung.
- **Init → Initialized**: Nach der Initialisierung wechselt der Automat sofort in den Zustand `Initialized` (Transition mit Bedingung `1`). Hier verharrt der Baustein, bis entweder ein Ereignis am Socket `DI1.E1` eintrifft oder ein erneutes `INIT` mit `QI = FALSE` die Deinitialisierung auslöst.
- **Initialized → NormalOp**: Beim Eintreffen des Adapter-Ereignisses `DI1.E1` wechselt der Baustein in den Zustand `NormalOp`. Der Algorithmus `normalOperation` führt die eigentliche Durchschaltung durch: Sofern `QI = TRUE` ist, wird der Wert des Adapter-Eingangs `DI1.D1` auf den Adapter-Ausgang `DO1.D1` kopiert. Zudem wird `QO` auf den aktuellen Wert von `QI` gesetzt (bleibt so lange `TRUE`, bis `QI` auf `FALSE` gesetzt wird).
- **NormalOp → Initialized**: Nach der Ausführung kehrt der Baustein sofort wieder in den Zustand `Initialized` zurück (Transition `1`), bereit für das nächste `DI1.E1`-Ereignis.
- **Initialized → DeInit**: Wird das Ereignis `INIT` mit `QI = FALSE` empfangen, erfolgt die Deinitialisierung. Der Algorithmus `deInitialize` setzt `QO` auf `FALSE` und den Adapter-Ausgang `DO1.D1` auf `FALSE`. Anschließend wird `INITO` ausgegeben.
- **DeInit → START**: Nach der Deinitialisierung kehrt der Automat in den Startzustand zurück und erwartet eine erneute Initialisierung.

Zusammenfassend arbeitet der Baustein als *getriggerte Durchschaltung*: Der Adapter-Eingang wird bei jedem Ereignis am Socket auf den Adapter-Ausgang kopiert, jedoch nur, wenn der Baustein zuvor korrekt initialisiert wurde (`QI=TRUE` beim `INIT`).

Technische Besonderheiten

- **Adapterbasierte Schnittstelle**: Die Ein-/Ausgabedaten werden nicht über direkte Variablen, sondern über die unidirektionalen Adapter `DI1` und `DO1` (Typ `AX`) ausgetauscht. Dies ermöglicht flexible Kopplung mit anderen Bausteinen desselben Adaptertyps.
- **Initialisierungslogik**: Der Baustein unterscheidet zwischen initialisiertem (`QI=TRUE`) und deinitialisiertem (`QI=FALSE`) Zustand. Nur nach erfolgreicher Initialisierung werden eingehende Adapter-Ereignisse verarbeitet.
- **Paketname**: Der Baustein ist im Paket `logiBUS::utils::sequence::one` organisiert, was auf eine Verwendung in einer logiBUS-Umgebung hindeutet.
- **Automatische Zustandswechsel**: Die Transitionen von `Init→Initialized`, `NormalOp→Initialized` und `DeInit→START` erfolgen ohne weitere Bedingung (Transition `1`), d.h. die Ausführung der Algorithmen ist jeweils abgeschlossen.

Zustandsübersicht

Der ECC (Execution Control Chart) des Bausteins umfasst fünf Zustände:

Zustand	Beschreibung
`START`	Wartet auf erstes `INIT`-Ereignis mit `QI=TRUE`.
`Init`	Führt Initialisierungsalgorithmus aus, setzt `QO=QI` und sendet `INITO`.
`Initialized`	Ruhezustand nach erfolgreicher Initialisierung; wartet auf `DI1.E1` oder erneutes `INIT` (mit `QI=FALSE`).

`NormalOp`	Führt Durchschaltung durch (kopiert `DI1.D1` nach `DO1.D1`, falls `QI=TRUE`).
`DeInit`	Führt Deinitialisierung durch (setzt `QO` und `DO1.D1` auf `FALSE`), sendet `INITO`.

Anwendungsszenarien

- **Signaldurchschaltung mit Sicherung**: Ein boolesches Signal (z.B. Freigabe eines Prozesses) soll nur dann von einer Quelle zu einer Senke weitergegeben werden, wenn der Baustein zuvor initialisiert wurde. Ein erneutes `INIT` mit `QI=FALSE` unterbricht die Durchschaltung.
- **Initialisierungssequenz in Fertigungsanlagen**: Vor dem Start einer Produktion wird der Baustein initialisiert (z.B. Prüfung von Sicherheitsbedingungen). Erst danach dürfen Aktoren über `DO1` angesteuert werden.
- **Datenweitergabe in modularen Systemen**: Der unidirektionale Adapter `AX` erlaubt die lose Kopplung zwischen Modulen – der Baustein dient als einfacher „Durchgang“ in einer Datenkette.

Vergleich mit ähnlichen Bausteinen

- **BasicOne (ohne Adapter)**: Ein typischer `BasicOne`-Baustein nutzt direkte Ein-/Ausgangsvariablen. `BasicOne_AX` erweitert dies um eine adapterbasierte Schnittstelle, was die Wiederverwendbarkeit und Strukturierung in größeren Systemen verbessert.
- **Standard `E_SR` (Set/Reset)**: Ein Set/Reset-Flipflop speichert einen Zustand, während `BasicOne_AX` keine Speicherfunktion besitzt – der Ausgang folgt dem Eingang nur bei aktiver Initialisierung und bei jedem Ereignis.
- **Move-Funktionsbaustein**: Ein `MOVE`-Baustein kopiert kontinuierlich Werte. `BasicOne_AX` kopiert hingegen nur auf Ereignis (getriggert) und nur nach erfolgter Initialisierung.

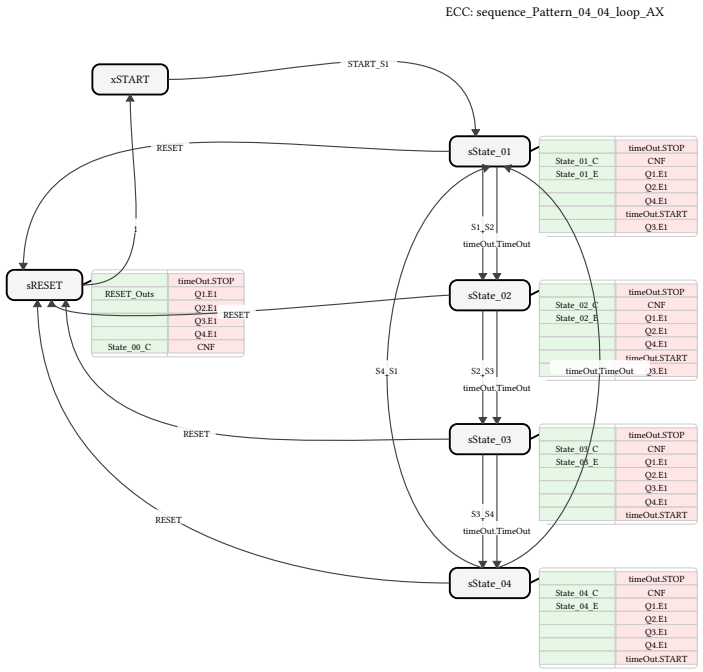
Fazit

Der `BasicOne_AX` ist ein minimalistischer, aber robuster IEC 61499-Baustein für initialisierungsgesteuerte Signaldurchschaltungen mit Adapter-Schnittstelle. Seine klare Zustandslogik und die Trennung von Initialisierung und Betrieb machen ihn besonders geeignet für sicherheitskritische oder sequentielle Anwendungen, in denen ein definierter Startzustand vor der Datenweitergabe erforderlich ist. Die Verwendung unidirektionaler Adapter erhöht die Modularität und erleichtert die Integration in komplexe Automatisierungssysteme.

Numerischer Index (sequence)

Hier finden Sie alle Übungen des Arbeitsbereichs `sequence` in aufsteigender Reihenfolge.

sequence_Pattern_04_04_loop_AX

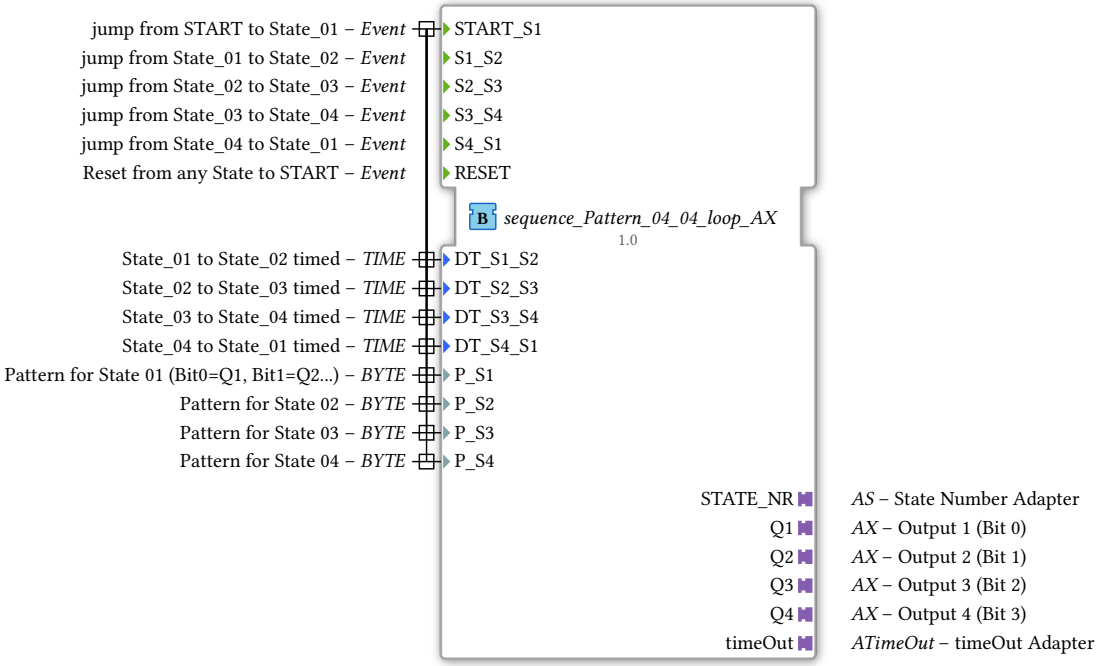


• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **sequence_Pattern_04_04_loop_AX** ist ein Sequenzer (Schrittschaltwerk), der eine Abfolge von 4 Zuständen in einer Endlosschleife steuert. Er fungiert ähnlich wie ein elektronisches Nockenschaltwerk. Für jeden der 4 Schritte kann ein spezifisches Ausgabemuster (Pattern) definiert werden, das 4 Ausgänge (Q1 bis Q4) steuert.

Der Baustein nutzt **AX-Adapter** für die Ausgänge und unterstützt sowohl zeitgesteuerte Übergänge (Timeouts) als auch ereignisgesteuerte manuelle Weiterschaltung. Nach dem vierten Schritt springt die Sequenz automatisch oder manuell zurück zum ersten Schritt (Loop-Verhalten).



Schnittstellenstruktur

Ereignis-Eingänge

- **START_S1**: Startet die Sequenz und springt direkt in den Zustand 1 (`State_01`). Dabei werden die Konfigurationsdaten (Zeiten und Muster) eingelesen.
- **S1_S2**: Manueller Auslöser für den Übergang von Schritt 1 zu Schritt 2.
- **S2_S3**: Manueller Auslöser für den Übergang von Schritt 2 zu Schritt 3.
- **S3_S4**: Manueller Auslöser für den Übergang von Schritt 3 zu Schritt 4.
- **S4_S1**: Manueller Auslöser für den Übergang von Schritt 4 zurück zu Schritt 1 (Schleifenschluss).
- **RESET**: Setzt den Baustein aus jedem Zustand zurück in den Initialzustand (`START`). Alle Ausgänge werden deaktiviert.

Ereignis-Ausgänge

- **CNF**: Bestätigungsereignis (Execution Confirmation). Wird gefeuert, wenn sich der Zustand ändert oder der Baustein initialisiert wurde.

Daten-Eingänge

- **DT_S1_S2** (TIME): Verweilzeit in Schritt 1, bevor automatisch zu Schritt 2 gewechselt wird. Standardwert: `NO_TIME` (kein automatischer Wechsel).
- **DT_S2_S3** (TIME): Verweilzeit in Schritt 2.
- **DT_S3_S4** (TIME): Verweilzeit in Schritt 3.
- **DT_S4_S1** (TIME): Verweilzeit in Schritt 4, bevor automatisch zurück zu Schritt 1 gewechselt wird.
- **P_S1** (BYTE): Ausgabemuster für Schritt 1.
  - Bit 0 steuert Q1
  - Bit 1 steuert Q2
  - Bit 2 steuert Q3
  - Bit 3 steuert Q4
- **P_S2** (BYTE): Ausgabemuster für Schritt 2.
- **P_S3** (BYTE): Ausgabemuster für Schritt 3.
- **P_S4** (BYTE): Ausgabemuster für Schritt 4.

Daten-Ausgänge

- **STATE_NR** (SINT): Aktuelle Zustandsnummer (0 = START, 1 = State_01, ..., 4 = State_04).

Adapter

- **Q1** (`adapter::types::unidirectional::AX`): Ausgang 1 (gesteuert durch Bit 0 der Pattern).
- **Q2** (`adapter::types::unidirectional::AX`): Ausgang 2 (gesteuert durch Bit 1 der Pattern).
- **Q3** (`adapter::types::unidirectional::AX`): Ausgang 3 (gesteuert durch Bit 2 der Pattern).
- **Q4** (`adapter::types::unidirectional::AX`): Ausgang 4 (gesteuert durch Bit 3 der Pattern).
- **timeOut** (`iec61499::events::ATimeOut`): Interner Adapter zur Verwaltung der Zeitverzögerungen (Timer).

Funktionsweise

Der Baustein implementiert eine State-Machine (ECC), die zyklisch durch vier aktive Zustände läuft.

1. **Initialisierung**: Der Baustein startet im Zustand `xSTART`. Durch das Event `START_S1` wechselt er in `sState_01`.
2. **Zustandsverarbeitung**: In jedem Zustand (`sState_01` bis `sState_04`):
   - Wird der Timer (`timeOut`) gestoppt und mit der entsprechenden Zeit (`DT_xx`) neu gestartet.
   - Wird die aktuelle Zustandsnummer (`STATE_NR`) aktualisiert und `CNF` gesendet.
   - Werden die Ausgänge `Q1` bis `Q4` basierend auf dem korrespondierenden Eingangs-Byte (`P_Sx`) gesetzt. Das Mapping erfolgt bitweise (Bit 0 -> Q1, Bit 1 -> Q2, etc.).
   - Werden die Events an den Adaptern `Qx` gefeuert (`Qx.E1`), um die Datenänderung zu signalisieren.
3. **Übergänge (Transitionen)**: Ein Wechsel zum nächsten Schritt erfolgt entweder:
   - **Automatisch**: Wenn die konfigurierte Zeit (`DT_...`) abgelaufen ist (`timeOut.TimeOut`).
   - **Manuell**: Wenn das explizite Weiterschalt-Event (z.B. `S1_S2`) eintrifft.
4. **Schleife**: Nach `sState_04` erfolgt der Übergang zurück zu `sState_01` (Loop), sofern kein Reset ausgelöst wird.
5. **Reset**: Das `RESET` Event führt jederzeit dazu, dass alle Ausgänge auf `FALSE` gesetzt werden, `STATE_NR` auf 0 geht und der Baustein im `xSTART` Zustand auf einen neuen Startbefehl wartet.

Technische Besonderheiten

- **Bit-Mapping**: Die Steuerung der Ausgänge erfolgt effizient über Byte-Muster. Dies ermöglicht es, beliebige Kombinationen der 4 Ausgänge pro Schritt mit einem einzigen Parameter zu definieren.
  - Beispiel: `P_S1 = 5` (Binär `0000 0101`) aktiviert Q1 und Q3.
- **AX-Adapter**: Die Verwendung von AX-Adaptern deutet darauf hin, dass dieser Baustein für modulare Systeme konzipiert ist, bei denen die Aktoren über standardisierte Schnittstellen angebunden sind.
- **Zeit & Event Dualität**: Die Flexibilität, Schritte sowohl zeitgesteuert als auch ereignisgesteuert zu verlassen, erlaubt gemischte Betriebsarten (z.B. Schritt 1 nach Zeit, Schritt 2 wartet auf manuelle Bestätigung).

Zustandsübersicht

Zustand	Beschreibung	Ausgangslogik (Q1-Q4)	Nächster Zustand (Auto/Manuell)
xSTART	Ruhezustand	Inaktiv	sState_01 (via `START_S1`)
sState_01	Schritt 1	P_S1	sState_02
sState_02	Schritt 2	P_S2	sState_03
sState_03	Schritt 3	P_S3	sState_04
sState_04	Schritt 4	P_S4	sState_01 (Loop)
sRESET	Reset-Logik	Alle FALSE	xSTART

Anwendungsszenarien

- **Lauflicht / Blinker**: Einfache zyklische Lichtsteuerungen.
- **Nockenschaltwerk-Simulation**: Steuerung von wiederkehrenden Maschinenzyklen (z.B. Greifen, Heben, Verschieben, Senken).
- **Ampelsteuerungen**: Durch Anpassung der `DT`-Zeiten können einfache Ampelphasen realisiert werden (wobei hier meist 4 Schritte ausreichend sind, z.B. Rot, Rot-Gelb, Grün, Gelb).
- **Test-Muster-Generator**: Erzeugung von definierten Signalabfolgen zu Testzwecken.

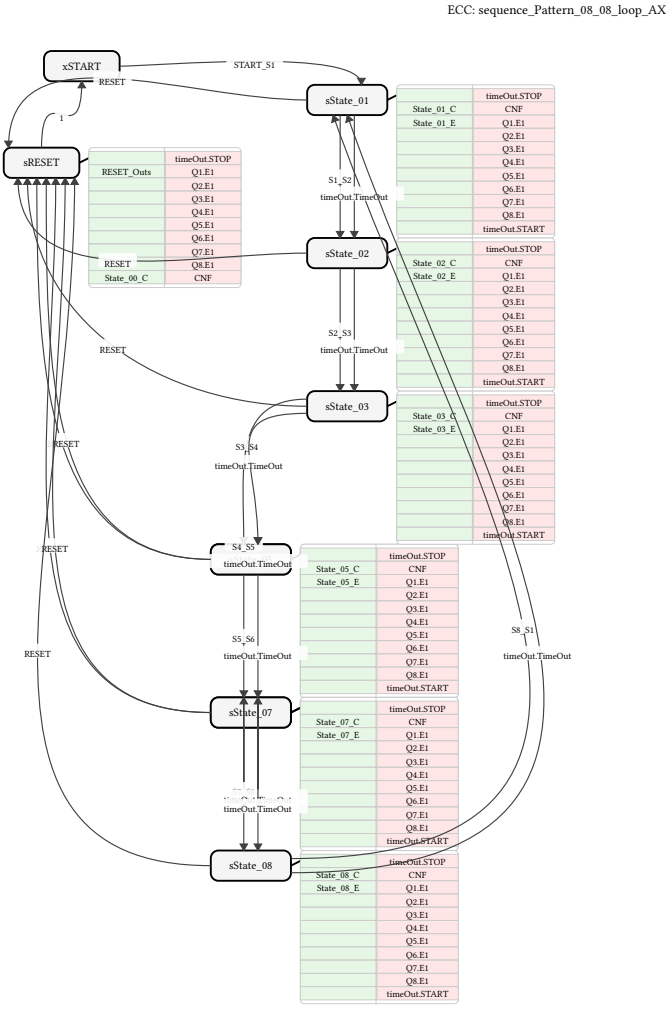
⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu einfachen Zählern bietet dieser Baustein integrierte Zeitsteuerungen pro Schritt und eine direkte Musterzuweisung. Gegenüber komplexeren Sequenzern (z.B. SFC / Ablaufsprache) ist er kompakter, aber auf genau 4 Schritte und 4 Ausgänge limitiert. Die „Loop"-Variante unterscheidet sich von „Single-Shot"-Sequenzern dadurch, dass sie nicht am Ende stehen bleibt, sondern endlos wiederholt.

Fazit

Der `sequence_Pattern_04_04_loop_AX` ist ein robuster und flexibler Baustein für kleine, zyklische Steuerungsaufgaben in IEC 61499 Anwendungen. Durch die Kombination aus Zeit- und Ereignissteuerung sowie der direkten Bit-zu-Adapter-Zuweisung eignet er sich hervorragend für repetitive Automatisierungssequenzen.

sequence_Pattern_08_08_loop_AX

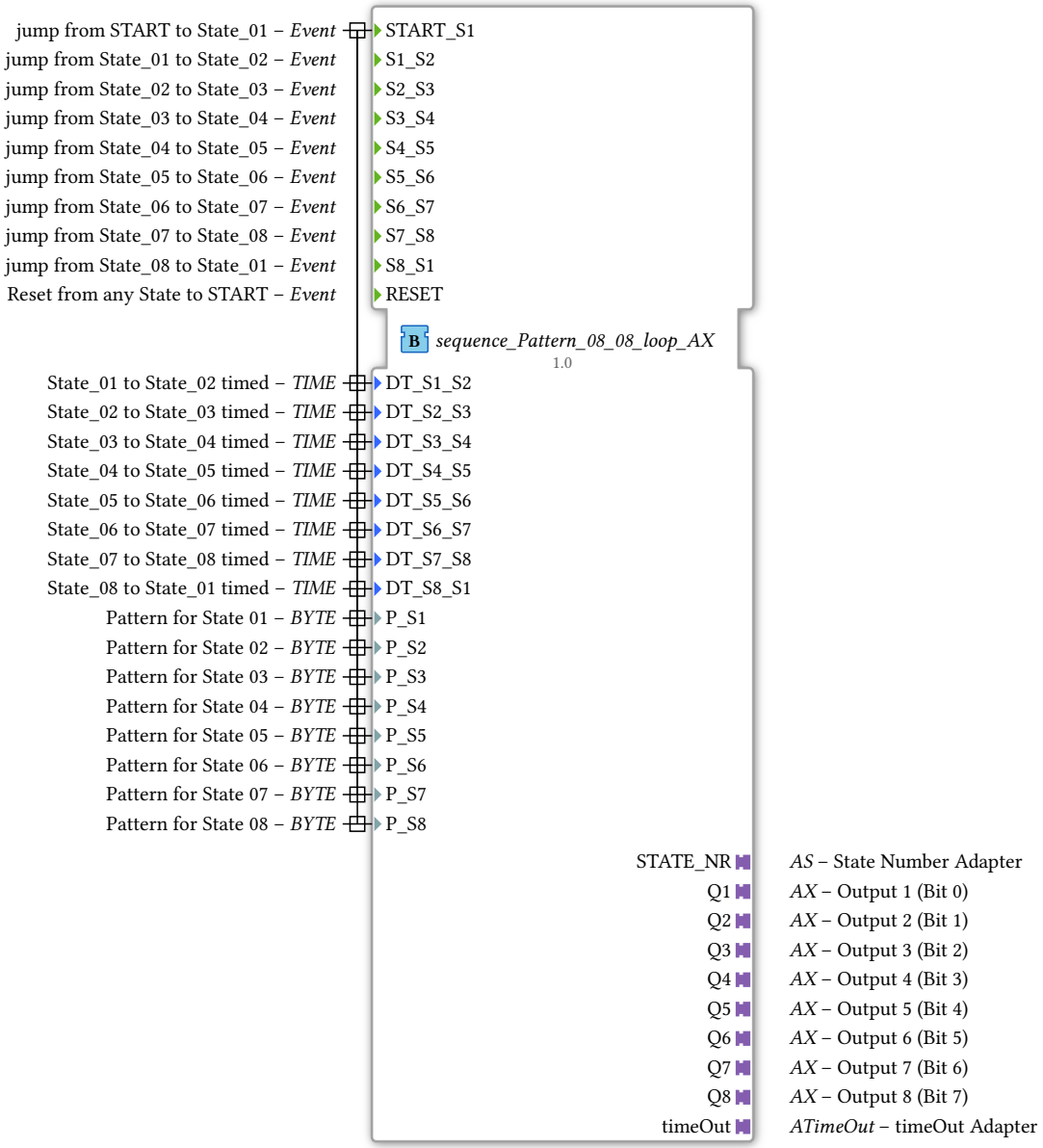


• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **sequence_Pattern_08_08_loop_AX** ist ein Sequenzer (Schrittkette), der ein konfigurierbares „Nockenschaltwerk“ mit 8 Zuständen (Schritten) und 8 Ausgängen implementiert. Er ist darauf ausgelegt, zyklisch zu arbeiten (Loop-Verhalten), wobei der Übergang von Schritt 8 zurück zu Schritt 1 erfolgt.

Dieser Baustein ermöglicht die Definition von Bitmustern für jeden der 8 Schritte, die über spezielle Adapter (AX-Schnittstelle) ausgegeben werden. Die Weiterschaltung zwischen den Zuständen kann entweder ereignisgesteuert oder zeitgesteuert erfolgen.



Schnittstellenstruktur

Ereignis-Eingänge

- **START_S1**: Initialisiert die Sequenz und springt direkt in den Zustand 1 (`State_01`). Dieses Ereignis übernimmt auch die parametrisierten Zeitwerte (`DT_...`) und Bitmuster (`P_...`).
- **S1_S2** bis **S8_S1**: Manuelle Trigger-Ereignisse, um sofort vom aktuellen Zustand in den nächsten zu wechseln (z. B. springt `S1_S2` von Zustand 1 nach 2).
- **RESET**: Setzt den Baustein aus jedem beliebigen Zustand in den Initialzustand (`START`) zurück und deaktiviert alle Ausgänge.

Ereignis-Ausgänge

- **CNF**: Bestätigungs-Ereignis (Execution Confirmation). Es wird gefeuert, sobald ein neuer Zustand erreicht wurde und die Ausgänge aktualisiert sind.

Daten-Eingänge

- **DT_S1_S2** bis **DT_S8_S1** (Typ: `TIME`): Definieren die Verweildauer im jeweiligen Zustand, bevor automatisch in den nächsten geschaltet wird. Standardwert ist `NO_TIME` (kein automatisches Umschalten per Zeit).
- **P_S1** bis **P_S8** (Typ: `BYTE`): Definieren das Ausgabemuster für den jeweiligen Zustand.

  - Bit 0 steuert Adapter Q1
  - Bit 1 steuert Adapter Q2
  - ...
  - Bit 7 steuert Adapter Q8

Daten-Ausgänge

- **STATE_NR** (Typ: `SINT`): Gibt die aktuelle Zustandsnummer aus (0 = START, 1 = State_01, ..., 8 = State_08).

Adapter

- **Q1** bis **Q8** (Typ: `adapter::types::unidirectional::AX`): Die 8 physikalischen oder logischen Ausgänge. Jeder Adapter erhält ein Daten-Signal (`D1`) und ein Ereignis-Signal (`E1`).
- **timeOut** (Typ: `iec61499::events::ATimeOut`): Ein interner Adapter zur Handhabung der zeitgesteuerten Übergänge zwischen den Schritten.

Funktionsweise

Der Baustein arbeitet als endlicher Automat (State Machine) mit einer zyklischen Struktur:

1. **Initialisierung**: Durch das Ereignis `START_S1` wird die Sequenz gestartet. Dabei werden die konfigurierten Zeitwerte und Bitmuster eingelesen. Der Automat wechselt in **State_01**.
2. **Zustandslogik**: In jedem Zustand (`State_01` bis `State_08`) wird das entsprechende Byte-Muster (`P_Sn`) zerlegt.
   - Ist Bit 0 von `P_S1` gesetzt, wird `Q1.D1` auf TRUE gesetzt.
   - Ist Bit 1 von `P_S1` gesetzt, wird `Q2.D1` auf TRUE gesetzt, usw.
   - Gleichzeitig wird das Ereignis `E1` an allen Adaptern `Q1` bis `Q8` ausgelöst, um die Änderung zu signalisieren.
3. **Transitionen**: Der Wechsel zum nächsten Zustand erfolgt, wenn:
   - Das explizite Ereignis empfangen wird (z. B. `S1_S2`).
   - ODER die konfigurierte Zeit (`DT_S1_S2`) abgelaufen ist (via `timeOut` Adapter).
4. **Zyklus**: Nach **State_08** folgt der Übergang zurück zu **State_01** (`S8_S1` oder Zeitablauf), wodurch eine Endlosschleife entsteht.
5. **Reset**: Das Ereignis `RESET` unterbricht den Ablauf sofort, setzt alle Ausgänge (`Q1` bis `Q8`) auf `FALSE` und setzt die Zustandsnummer auf 0.

Technische Besonderheiten

- **AX-Adapter-Nutzung**: Anstelle von klassischen `BOOL`-Ausgängen nutzt dieser FB Adapter vom Typ `AX`. Dies ermöglicht eine flexiblere Anbindung, z.B. an Hardware-Abstraktionsschichten oder komplexe Aktoren, die ein Event-Signal zur Datenübernahme benötigen.
- **Bit-Mapping**: Die Steuerung der 8 Ausgänge erfolgt effizient über `BYTE`-Variablen. Dies reduziert die Anzahl der notwendigen Eingangspins am Baustein drastisch im Vergleich zu einzelnen Boolschen Eingängen pro Schritt und Ausgang.
- **Hybride Steuerung**: Der Baustein unterstützt gleichzeitig zeitgesteuerte Abläufe (Timer) und ereignisgesteuerte Abläufe (externe Trigger), was ihn sehr flexibel macht.

Zustandsübersicht

Zustands-ID	Name	Beschreibung	Ausgangslogik	Übergang nach
0	xSTART	Wartezustand / Reset	Keine	State_01 (bei START_S1)
1	sState_01	Schritt 1	Q1-Q8 gemäß P_S1	State_02
2	sState_02	Schritt 2	Q1-Q8 gemäß P_S2	State_03
3	sState_03	Schritt 3	Q1-Q8 gemäß P_S3	State_04
4	sState_04	Schritt 4	Q1-Q8 gemäß P_S4	State_05
5	sState_05	Schritt 5	Q1-Q8 gemäß P_S5	State_06
6	sState_06	Schritt 6	Q1-Q8 gemäß P_S6	State_07
7	sState_07	Schritt 7	Q1-Q8 gemäß P_S7	State_08

8	sState_08	Schritt 8	Q1-Q8 gemäß P_S8	State_01 (Loop)
-	sRESET	Reset-Zustand	Q1-Q8 = FALSE	xSTART

Anwendungsszenarien

- **Lauflicht-Steuerungen**: Durch Setzen von `P_S1=1`, `P_S2=2`, `P_S3=4` usw. kann ein einfaches Lauflicht realisiert werden.
- **Nockenschaltwerke**: Steuerung von mechanischen Abläufen, bei denen mehrere Aktoren (Zylinder, Ventile) in einer festen zeitlichen Reihenfolge aktiviert werden müssen.
- **Ampelsteuerungen**: Zyklische Abfolge von Rot-Gelb-Grün Phasen.
- **Reinigungszyklen**: Spülen, Waschen, Trocknen in wiederkehrenden Intervallen.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu linearen Sequenzern (ohne Loop) ist dieser Baustein explizit für wiederkehrende Prozesse konzipiert. Er unterscheidet sich von einfachen Zählern dadurch, dass für jeden Schritt ein individuelles Ausgangsmuster und eine individuelle Zeit definiert werden kann. Die Verwendung von AX-Adaptern unterscheidet ihn von Standard-IEC-61499-Bausteinen, die meist direkte Boolesche Ausgänge verwenden, und prädestiniert ihn für strukturierte, objektorientierte Steuerungsdesigns.

Fazit

Der **sequence_Pattern_08_08_loop_AX** ist ein leistungsfähiger und kompakter Baustein zur Realisierung komplexer, zyklischer Schrittketten. Durch die Parametrierung über Byte-Muster und Zeitwerte lassen sich vielfältige Steuerungsaufgaben ohne Änderung der internen Logik realisieren. Die integrierte Reset-Funktionalität und die Adapter-Schnittstelle sorgen für eine saubere Integration in moderne 4diac-Anwendungen.

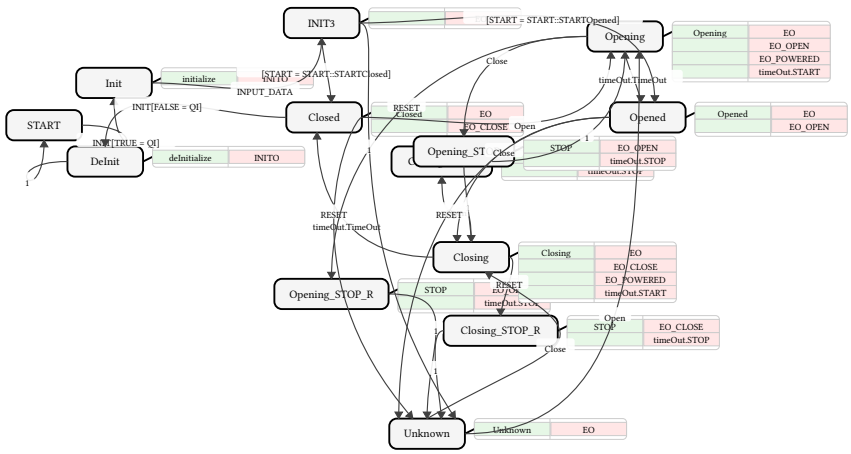
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

schieber

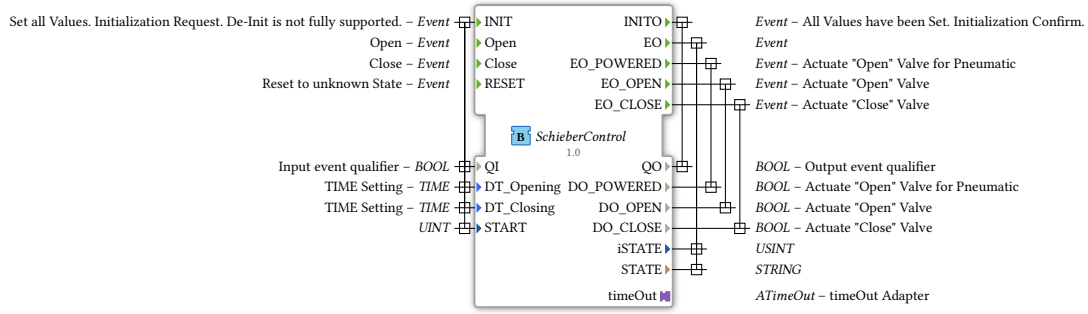
SchieberControl

ECC: SchieberControl



Einleitung

Der Funktionsblock `SchieberControl` ist ein Basis-Funktionsblock (Basic FB) gemäß IEC 61499 zur Steuerung eines Schieber-Antriebs (z. B. pneumatisch). Er realisiert eine vollständige Zustandsmaschine (ECC), die die Bewegung (Öffnen/Schließen), Haltezustände und Fehlerzustände eines Schiebers verwaltet. Der Baustein bietet eine Schnittstelle zur Parametrierung, zur Bedienung über die Ereignisse `Open`, `Close` und `RESET` sowie zur Ausgabe von Zustand und Steuersignalen für die Aktorik.



Schnittstellenstruktur

Ereignis-Eingänge

- **`INIT`**: Initialisierungsanforderung. Setzt alle Werte. Ein vollständiges De-Initialisieren wird nicht vollständig unterstützt. Wird mit den Daten `QI`, `DT_Opening`, `DT_Closing` und `START` verknüpft.
- **`Open`**: Startet den Öffnungsvorgang des Schiebers.
- **`Close`**: Startet den Schließvorgang des Schiebers.
- **`RESET`**: Setzt den internen Zustand auf „Unknown“ (Unbekannt) zurück.

Ereignis-Ausgänge

- **`INITO`**: Initialisierungsbestätigung. Wird mit dem Datenausgang `QO` verknüpft.
- **`EO`**: Allgemeines Ausgangsereignis, das den aktuellen Systemzustand und die konfigurierten Ausgangswerte liefert. Wird mit `iSTATE` und `STATE` verknüpft.
- **`EO_POWERED`**: Steuersignal zum Betätigen des „Öffnen“-Ventils für die Pneumatik. Wird mit `DO_POWERED` verknüpft.
- **`EO_OPEN`**: Steuersignal zum Betätigen des „Öffnen“-Ventils. Wird mit `DO_OPEN` verknüpft.
- **`EO_CLOSE`**: Steuersignal zum Betätigen des „Schließen“-Ventils. Wird mit `DO_CLOSE` verknüpft.

Daten-Eingänge

- **`QI` (BOOL)**: Qualifizierer für das INIT-Ereignis.
- **`DT_Opening` (TIME)**: Zeitdauer für den Öffnungsvorgang.
- **`DT_Closing` (TIME)**: Zeitdauer für den Schließvorgang.
- **`START` (UINT)**: Definiert den gewünschten Startzustand nach der Initialisierung (z. B. `START::STARTClosed`, `START::STARTOpened`, `START::STARTUnknown`).

Daten-Ausgänge

- **`QO` (BOOL)**: Qualifizierer für das INITO-Ereignis.
- **`DO_POWERED` (BOOL)**: Binäres Signal zum Betätigen des „Öffnen“-Ventils für die Pneumatik.
- **`DO_OPEN` (BOOL)**: Binäres Signal zum Betätigen des „Öffnen“-Ventils.
- **`DO_CLOSE` (BOOL)**: Binäres Signal zum Betätigen des „Schließen“-Ventils.
- **`iSTATE` (USINT)**: Numerischer Wert des aktuellen internen Zustands des FB (wird mit `EO` ausgegeben).
- **`STATE` (STRING)**: Zeichenkette, die den aktuellen internen Zustand des FB beschreibt (z. B. „Closed“, „Opening“).

Adapter

- **`timeOut` (ATimeOut)**: Ein Stecker-Adapter vom Typ `ATimeOut`. Wird zur Realisierung der zeitgesteuerten Zustandsübergänge (Öffnen/Schließen) verwendet. Der FB startet (`START`) und stoppt (`STOP`) den Timer und reagiert auf dessen `TimeOut`-Ereignis.

Funktionsweise

Der SchieberControl-FB arbeitet als zustandsgesteuerte Sequenz. Der interne Ablaufsteuerungsgraph (ECC) definiert die Zustände Closed (Geschlossen), Opened (Geöffnet), Opening (Öffnend), Closing (Schließend), verschiedene STOP-Zustände und Unknown (Unbekannt). Bei jedem Zustandswechsel wird ein entsprechender Algorithmus ausgeführt, der die Datenausgänge (wie DO_OPEN, iSTATE, STATE) setzt und bei Bedarf den Timer-Adapter startet. Die Bewegung zwischen Closed und Opened erfolgt zeitgesteuert über die Zustände Opening und Closing. Während einer Bewegung kann durch das gegenteilige Ereignis (Open während Closing oder Close während Opening) ein Stopp eingeleitet werden, der in einen entsprechenden STOP-Zustand überführt. Von jedem Zustand aus kann ein RESET den FB in den Unknown-Zustand versetzen.

Technische Besonderheiten

- **Zustandsinitialisierung**: Über den START-Eingang kann nach INIT der gewünschte Anfangszustand (Closed, Opened oder Unknown) vorgegeben werden.- **Adapter-Nutzung**: Die Zeitsteuerung ist vollständig in den Adapter timeOut ausgelagert, was eine klare Trennung der Funktionalität und potenzielle Wiederverwendbarkeit ermöglicht.
- **Zustandsausgabe**: Der aktuelle Zustand wird sowohl numerisch (iSTATE, USINT) als auch als Text (STATE, STRING) mit dem Ereignis EO ausgegeben und kann so von Bedienoberflächen ausgewertet werden.
- **Power-Signal**: Das Signal DO_POWERED wird nur in den Zuständen Opening und Opened aktiviert, was auf eine spezifische pneumatische Ansteuerlogik hindeutet (z. B. Halten der Druckluft im geöffneten Zustand).

Zustandsübersicht

Die ECC umfasst folgende Hauptzustände:

1. **START**: Inaktiver Ausgangszustand.
2. **Init/INIT3**: Initialisierungssequenz.
3. **Closed**: Endzustand „geschlossen“. Aktiviert DO_CLOSE.
4. **Opened**: Endzustand „geöffnet“. Aktiviert DO_OPEN und DO_POWERED.
5. **Opening/Closing**: Bewegungszustände. Starten den Timer mit DT_Opening/DT_Closing und aktivieren die entsprechenden Aktorsignale sowie DO_POWERED (nur bei Opening).
6. **Opening_STOP/Closing_STOP**: Stopp-Zustände während einer Bewegung, ausgelöst durch den gegenteiligen Befehl. Deaktivieren DO_OPEN/DO_CLOSE und stoppen den Timer.
7. **Opening_STOP_R/Closing_STOP_R**: Stopp-Zustände während einer Bewegung, ausgelöst durch RESET. Führen zum Unknown-Zustand.
8. **Unknown**: Fehler- oder unbekannter Zustand. Setzt alle Aktorsignale zurück.

Anwendungsszenarien

- **Steuerung pneumatischer Schieber** in Verpackungs-, Förder- oder Sortieranlagen.
- **Integration in übergeordnete Sequenzer** für Materialfluss-Steuerungen.
- **Anbindung an Bedienpanels (HMI)**, da der Zustand als Zahl (iSTATE) und Text (STATE) für Anzeigen bereitgestellt wird.
- **Protokollierung und Visualisierung** des Schieberzustands über den STATE-String-Ausgang.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren binären Aktor-Bausteinen (z. B. einem einfachen Zylinder-FB) bietet SchieberControl eine deutlich höhere Funktionalität:

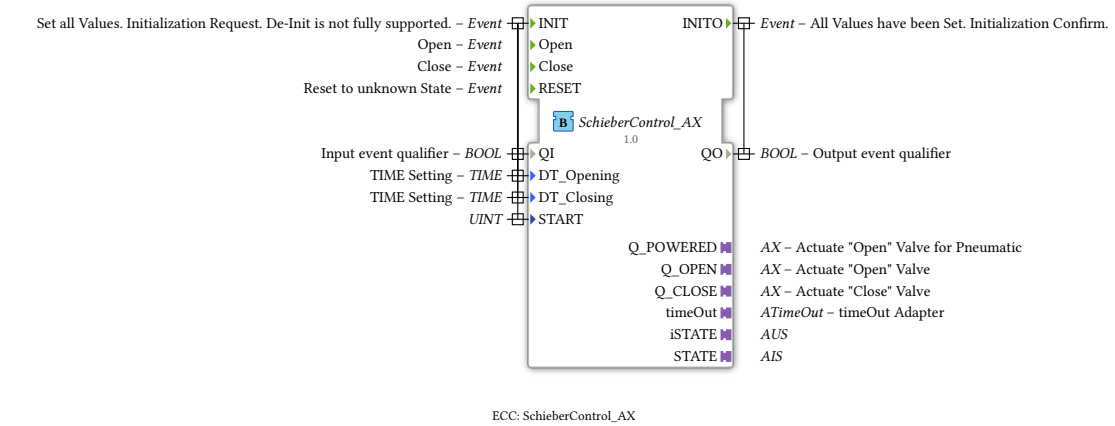
- **Vollständige Zustandsmaschine** mit Bewegungs-, End- und Stopp-Zuständen.
- **Integrierte Zeitsteuerung** für die Bewegung.
- **Umfangreiche HMI-Schnittstelle** zur Konfiguration von Bedienelementen.
- **Explizite Behandlung eines „Unbekannt“-Zustands** (RESET-Funktionalität). Er ist spezialisierter und komplexer als ein generischer E_SR (Flip-Flop) oder ein einfacher Timer-Baustein, da er diese Funktionalitäten kombiniert und auf die Steuerung eines Schieberantriebs zuschneidet.

Fazit

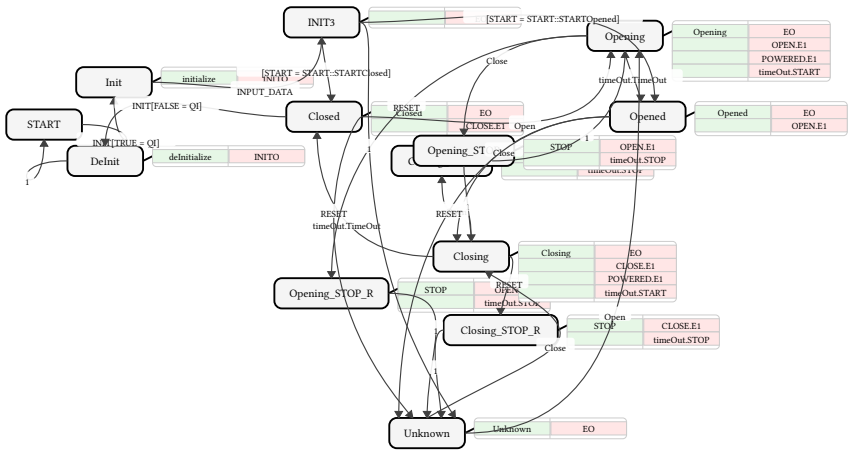
Der SchieberControl ist ein robust ausgelegter und funktional umfassender Steuerbaustein für Schieber-Antriebe. Seine Stärken liegen in der klaren Zustandslogik, der flexiblen Konfigurierbarkeit der Benutzerschnittstelle und der sauberen Trennung von Steuerlogik und Zeitgeber-Funktionalität durch

Adapter. Er eignet sich gut für den Einsatz in mittleren bis komplexen Steuerungsanwendungen, bei denen eine zuverlässige und überwachbare Ansteuerung eines Schiebers erforderlich ist.

SchieberControl_AX



ECC: SchieberControl_AX



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsbaustein **SchieberControl_AX** dient zur Steuerung eines Schiebers (Ventil, Klappe oder ähnliche Aktorik) innerhalb eines 61499-basierten Steuerungssystems. Er implementiert eine Zustandsmaschine, die nicht nur die logischen Zustände (Offen, Geschlossen, Öffnend, Schließend) verwaltet, sondern auch die Zeitsteuerung für die Bewegungen übernimmt und den aktuellen Zustand über die Adapter `STATE` und `iSTATE` bereitstellt.

Diese Version des Bausteins („AX Adapter Version“) nutzt spezielle Adapter (`adapter::types::unidirectional::AX`) zur Ansteuerung der physikalischen Ventile.

Schnittstellenstruktur

Ereignis-Eingänge

Ereignis	Typ	Kommentar

INIT	EInit	Initialisierungsanfrage (mit QI, DT_Opening, DT_Closing, START). Setzt alle Werte und Parameter.
Open	Event	Befehl zum Öffnen des Schiebers.
Close	Event	Befehl zum Schließen des Schiebers.
RESET	Event	Setzt den Baustein in einen unbekannten Zustand („Unknown“) zurück.

Ereignis-Ausgänge

Ereignis	Typ	Kommentar
INITO	EInit	Bestätigung der Initialisierung (mit QO).

Daten-Eingänge

Variable	Datentyp	Kommentar
QI	BOOL	Input Event Qualifier (True = Normalbetrieb, False = De-Init).
DT_Opening	TIME	Zeitdauer für den Öffnungsvorgang.
DT_Closing	TIME	Zeitdauer für den Schließvorgang.
START	UINT	Startkonfiguration (Definiert, in welchem Zustand der Baustein startet). Standard: `STARTUnknown`.

Daten-Ausgänge

Variable	Datentyp	Kommentar
QO	BOOL	Output Event Qualifier.

Adapter

Name	Typ	Kommentar
Q_POWERED	AX	Adapter zur Ansteuerung des „Öffnen“-Ventils (Pneumatik/Hauptversorgung). Aktiv während des Öffnens und im offenen Zustand.
Q_OPEN	AX	Adapter zur Ansteuerung des „Öffnen“-Signals.
Q_CLOSE	AX	Adapter zur Ansteuerung des „Schließen“-Signals.
timeOut	ATimeOut	Adapter für Timer-Funktionalität (überwacht die Öffnungs-/Schließzeiten).
iSTATE	AUS	Adapter (`adapter::types::unidirectional::AUS`), gibt den aktuellen Zustand als `iSTATES`-Konstante über `iSTATE.D1` aus.
STATE	AIS	Adapter (`adapter::types::unidirectional::AIS`), gibt den aktuellen Zustand als `STATES`-Konstante über `STATE.D1` aus.

Funktionsweise

Der Baustein arbeitet als Zustandsautomat (ECC), der Übergänge basierend auf den Ereignissen `Open` und `Close` sowie dem Timer-Adapter `timeOut` steuert.

1. **Initialisierung:** Beim Start (`INIT`) wird geprüft, in welchem Zustand der Schieber beginnen soll (definiert durch `START`). Möglich sind z.B. direkt „Closed“, „Opened“ oder „Unknown“.
2. **Bewegungsablauf:**
   - Erfolgt der Befehl `Open`, wechselt der Baustein in den Zustand **Opening**. Dabei werden die Adapter `Q_POWERED` und `Q_OPEN` aktiviert und der Timer mit `DT_Opening` gestartet.
   - Nach Ablauf der Zeit (`timeOut.TimeOut`) wechselt der Zustand automatisch zu **Opened**.

- Erfolgt der Befehl `Close`, wechselt der Baustein zu **Closing**. Der Adapter `Q_CLOSE` wird aktiviert (während `Q_POWERED` und `Q_OPEN` deaktiviert werden) und der Timer mit `DT_Closing` gestartet.
- Nach Ablauf der Zeit wechselt der Zustand zu **Closed**.

3. **Unterbrechung:** Wird während des Öffnens der Befehl `Close` (oder umgekehrt) gegeben, wird der Vorgang gestoppt (`STOP`-Zustände), die Ausgänge zurückgesetzt und der entgegengesetzte Vorgang eingeleitet.
4. **Visualisierung:** In jedem Zustand (Closed, Opening, Opened, Closing, Unknown) werden die Adapter `STATE` (Konstante aus `STATES`) und `iSTATE` (Konstante aus `iSTATES`) mit dem aktuellen Zustand beschrieben. Dies ermöglicht eine dynamische Anpassung der Benutzeroberfläche durch nachgeschaltete Bausteine.

Technische Besonderheiten

- **AX-Adapter Integration:** Die direkte Nutzung von `adapter::types::unidirectional::AX` deutet auf eine standardisierte Schnittstelle zur Hardware-Abstraktion hin, was den Baustein wiederverwendbar für verschiedene Ventiltypen macht, solange der Adapter passt.
- **Zustandsausgabe:** Der Baustein gibt den aktuellen Zustand über die Adapter `STATE` (AIS) und `iSTATE` (AUS) als Konstanten aus (`STATES`, `iSTATES`). Das reduziert die Logik im HMI.
- **Stop-Logik:** Es sind explizite `STOP`-Zustände implementiert, die sicherstellen, dass bei einem Richtungswechsel die Ausgänge kurzzeitig definiert abgeschaltet werden (`timeOut.STOP`, Ventile aus), bevor die neue Richtung eingeschlagen wird.

Zustandsübersicht

Die wichtigsten Zustände im ECC (Execution Control Chart) sind:

- **START / Init / DeInit:** Verwaltungszustände für den Lebenszyklus des Bausteins.
- **Unknown:** Fehler- oder Initialzustand, wenn die Position nicht bekannt ist.
- **Closed:** Der Schieber ist vollständig geschlossen. (`Q_CLOSE`=False, `Q_OPEN`=False).
- **Opening:** Der Schieber öffnet sich gerade. (`Q_POWERED`=True, `Q_OPEN`=True, Timer läuft).
- **Opened:** Der Schieber ist vollständig offen. (`Q_POWERED`=True, `Q_OPEN`=False).
- **Closing:** Der Schieber schließt sich gerade. (`Q_CLOSE`=True, `Q_POWERED`=False, Timer läuft).
- **..._STOP:** Zwischenzustände zum sauberen Abbrechen von Bewegungen.

Anwendungsszenarien

- **Landwirtschaftliche Maschinen:** Steuerung von Gülle-Schiebern, Dosierklappen oder hydraulischen Auslegern.
- **Prozessautomatisierung:** Einfache Ventilsteuerungen, bei denen keine Endlagensensoren vorhanden sind, sondern über Zeit (`DT_Opening`/`DT_Closing`) gefahren wird.
- **HMI-Integration:** Systeme, bei denen sich das Symbol oder die Tastenfarbe auf dem Display ändern muss, je nachdem, ob das Ventil gerade fährt, offen oder geschlossen ist.

Vergleich mit ähnlichen Bausteinen

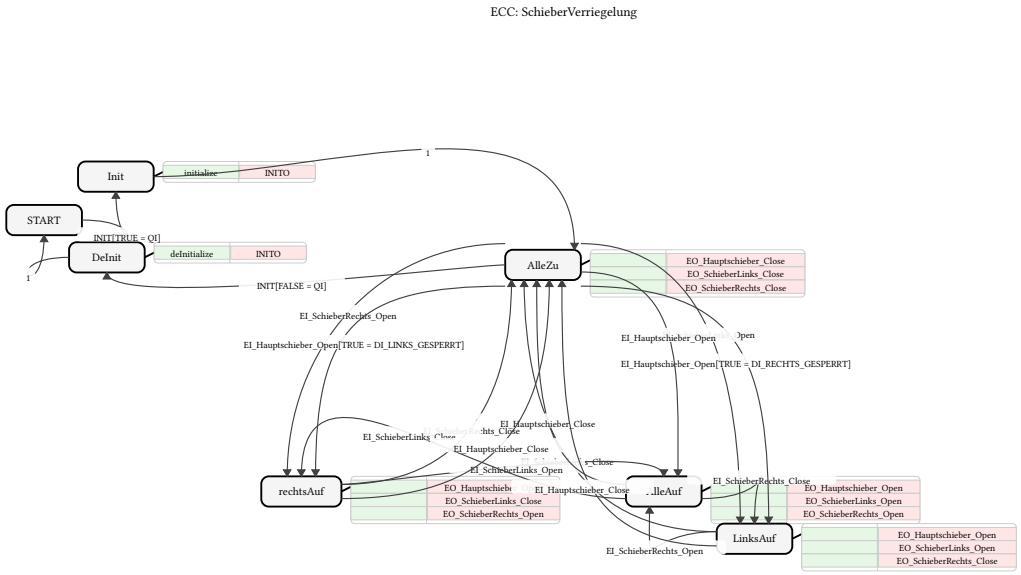
Im Gegensatz zu einem einfachen `SR`-Flipflop zur Ventilsteuerung bietet der **SchieberControl_AX**:

1. **Zeitüberwachung:** Integrierte Timer simulieren die Laufzeit.
2. **Zustandsausgabe:** Eingebaute Ausgabe des aktuellen Zustands für UI-Informationen.
3. **Hardware-Abstraktion:** Nutzung von Adaptern statt direkter boolescher Ausgänge für die Ventile.
4. **Zwischenzustände:** Explizite Darstellung von „Öffnet“ und „Schließt“, nicht nur „Auf“ und „Zu“.

Fazit

Der `SchieberControl_AX` ist ein spezialisierter, robuster Funktionsbaustein für die zeitgesteuerte Aktorik. Er kapselt die Komplexität von Zustandsübergängen, Timer-Handling und UI-Statusverwaltung in einer einzigen Komponente und erleichtert so die Entwicklung von Steuerungssoftware für Maschinen mit hydraulischen oder pneumatischen Schiebern erheblich.

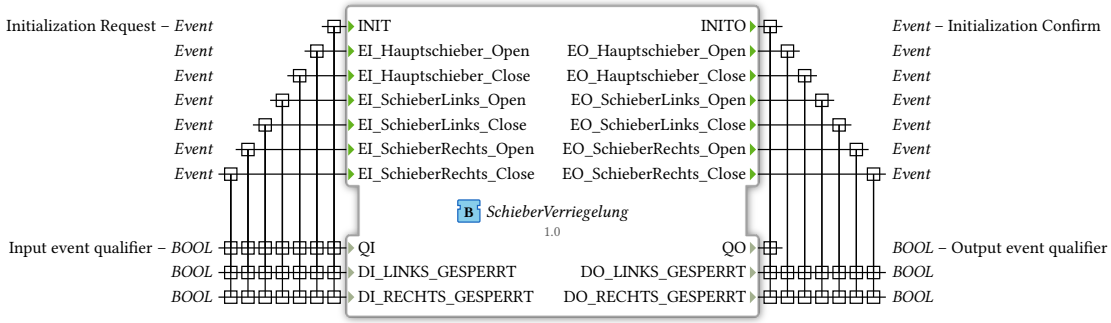
SchieberVerriegelung



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `SchieberVerriegelung` dient der koordinierten Steuerung und Verriegelung von drei Schiebern: einem Hauptschieber sowie einem linken und einem rechten Seitenschieber. Sein primärer Zweck ist es, basierend auf externen Anforderungen (Ereignisse) und Sperrzuständen, gültige und kollisionsfreie Schieber-Kombinationen zu gewährleisten. Der Baustein ist besonders für Anwendungen geeignet, bei denen die Bewegung eines Schiebers die Bewegung eines anderen verhindern oder erzwingen muss.



Schnittstellenstruktur

Ereignis-Eingänge

- **`INIT`**: Initialisierungsanforderung. Löst den Übergang in den initialisierten Zustand aus. Wird mit den Daten `QI`, `DI_LINKS_GESPERRT` und `DI_RECHTS_GESPERRT` verknüpft.
- **`EI_Hauptschieber_Open`**: Fordert das Öffnen des Hauptschiebers an.
- **`EI_Hauptschieber_Close`**: Fordert das Schließen des Hauptschiebers an.
- **`EI_SchieberLinks_Open`**: Fordert das Öffnen des linken Seitenschiebers an.
- **`EI_SchieberLinks_Close`**: Fordert das Schließen des linken Seitenschiebers an.
- **`EI_SchieberRechts_Open`**: Fordert das Öffnen des rechten Seitenschiebers an.
- **`EI_SchieberRechts_Close`**: Fordert das Schließen des rechten Seitenschiebers an.

Ereignis-Ausgänge

- **`INITO`**: Initialisierungsbestätigung. Wird nach Abschluss der Initialisierung (`INIT`) oder Deinitialisierung ausgelöst. Ist mit den Daten `QO`, `DO_LINKS_GESPERRT` und `DO_RECHTS_GESPERRT` verknüpft.
- **`EO_Hauptschieber_Open`**: Signalisiert den Befehl zum Öffnen des Hauptschiebers.
- **`EO_Hauptschieber_Close`**: Signalisiert den Befehl zum Schließen des Hauptschiebers.
- **`EO_SchieberLinks_Open`**: Signalisiert den Befehl zum Öffnen des linken Seitenschiebers.
- **`EO_SchieberLinks_Close`**: Signalisiert den Befehl zum Schließen des linken Seitenschiebers.
- **`EO_SchieberRechts_Open`**: Signalisiert den Befehl zum Öffnen des rechten Seitenschiebers.
- **`EO_SchieberRechts_Close`**: Signalisiert den Befehl zum Schließen des rechten Seitenschiebers.

Daten-Eingänge

- **`QI` (BOOL)**: Qualifiziert das INIT-Ereignis. `TRUE` startet die Initialisierung, `FALSE` die Deinitialisierung.
- **`DI_LINKS_GESPERRT` (BOOL)**: Signalisiert den Sperrzustand des linken Seitenschiebers (`TRUE` = gesperrt).
- **`DI_RECHTS_GESPERRT` (BOOL)**: Signalisiert den Sperrzustand des rechten Seitenschiebers (`TRUE` = gesperrt).

Daten-Ausgänge

- **`QO` (BOOL)**: Statusausgabe, die den Erfolg der Initialisierung/Deinitialisierung widerspiegelt.
- **`DO_LINKS_GESPERRT` (BOOL)**: Gibt den intern verarbeiteten oder weitergeleiteten Sperrzustand für den linken Schieber aus.
- **`DO_RECHTS_GESPERRT` (BOOL)**: Gibt den intern verarbeiteten oder weitergeleiteten Sperrzustand für den rechten Schieber aus.

Adapter

Dieser Funktionsblock verwendet keine Adapter-Schnittstellen.

Funktionsweise

Der `SchieberVerriegelung`-FB ist als BasicFB implementiert und folgt einer definierten Zustandsmaschine (ECC). Nach der Initialisierung startet der Baustein im Zustand `AlleZu` (alle Schieber geschlossen). Von hier aus können je nach eintreffendem Ereignis und den aktuellen Sperrzuständen (`DI_LINKS_GESPERRT`, `DI_RECHTS_GESPERRT`) Übergänge in andere Zustände erfolgen.

Die zentrale Logik liegt in der Interpretation des `EI_Hauptschieber_Open`-Ereignisses im Zustand `AlleZu`:

1. Wenn **kein** Seitenschieber gesperrt ist (`DI_LINKS_GESPERRT = FALSE` und `DI_RECHTS_GESPERRT = FALSE`), geht der FB in den Zustand `AlleAuf` über (alle Schieber öffnen).
2. Wenn **nur der rechte** Schieber gesperrt ist (`DI_RECHTS_GESPERRT = TRUE`), geht der FB in den Zustand `LinksAuf` über (Haupt- und linker Schieber öffnen, rechter bleibt geschlossen).
3. Wenn **nur der linke** Schieber gesperrt ist (`DI_LINKS_GESPERRT = TRUE`), geht der FB in den Zustand `rechtsAuf` über (Haupt- und rechter Schieber öffnen, linker bleibt geschlossen).

Die Zustände `LinksAuf` und `rechtsAuf` repräsentieren die verriegelten Betriebszustände. Von ihnen aus kann durch gezieltes Öffnen/Schließen der Seitenschieber in den Zustand `AlleAuf` gewechselt oder durch Schließen des Hauptschiebers zurück zu `AlleZu` gegangen werden.

Der Algorithmus `normalOperation` kopiert bei aktiviertem Betrieb (`QI=TRUE`) die Sperrzustände von den Eingängen (`DI_*_GESPERRT`) auf die Ausgänge (`DO_*_GESPERRT`).

Technische Besonderheiten

- **Zustandsgesteuerte Ausgabe**: Jeder operative Zustand (`AlleZu`, `AlleAuf`, `LinksAuf`, `rechtsAuf`) löst sofort eine feste Kombination von Ausgabeereignissen (`EO_*`) aus, die die gewünschte Schieberposition definieren.
- **Bedingte Transitionen**: Die Transition von `AlleZu` zu `LinksAuf`/`rechtsAuf` ist an den `EI_Hauptschieber_Open`-Event **und** den entsprechenden Sperrzustand (`DI_*_GESPERRT`) geknüpft. Dies stellt eine hardwarenahe Verriegelungslogik dar.
- **Explizite Deinitialisierung**: Ein `INIT`-Event mit `QI=FALSE` führt von jedem Zustand aus in den `DeInit`-Zustand und setzt den Ausgang `QO` auf `FALSE`.

Zustandsübersicht

1. **START**: Inaktiver Ausgangszustand vor der Initialisierung.

2. **Init**: Wird bei `INIT` mit `QI=TRUE` angesprungen. Führt den `initialize`-Algorithmus aus und bestätigt mit `INITO`.
3. **AlleZu** (Standard-Betriebszustand): Alle drei Schieber sind geschlossen. Sendet `Close`-Befehle für alle Schieber.
4. **AlleAuf**: Alle drei Schieber sind geöffnet. Sendet `Open`-Befehle für alle Schieber.
5. **LinksAuf**: Hauptschieber und linker Seitenschieber sind geöffnet, der rechte ist geschlossen (verriegelt). Sendet entsprechende `Open`/`Close`-Befehle.
6. **rechtsAuf**: Hauptschieber und rechter Seitenschieber sind geöffnet, der linke ist geschlossen (verriegelt). Sendet entsprechende `Open`/`Close`-Befehle.
7. **DeInit**: Deinitialisierungszustand. Setzt `QO` auf `FALSE` und geht dann zurück zu `START`.

Anwendungsszenarien

Typische Anwendungen finden sich in Verteil- und Förderanlagen, beispielsweise in der Landtechnik oder Materiallogistik:

- **Korn- oder Schüttgutförderer**: Der Hauptschieber leitet den Strom. Die Seitenschieber können je nach Bedarf geöffnet werden, um z.B. in verschiedene Silos zu verteilen. Die Verriegelung verhindert, dass beide Seiten gleichzeitig geöffnet sind, wenn dies mechanisch oder prozessbedingt unzulässig ist.
- **Weichensteuerung**: Analog zu einer mechanischen Weiche, bei der die Stellung eines Weichenschuhs die andere Position blockiert.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen Schieber-Steuerungsbausteinen (z.B. einzelnen `E_SR`-FlipFlops pro Schieber) bietet der `SchieberVerriegelung`-FB:

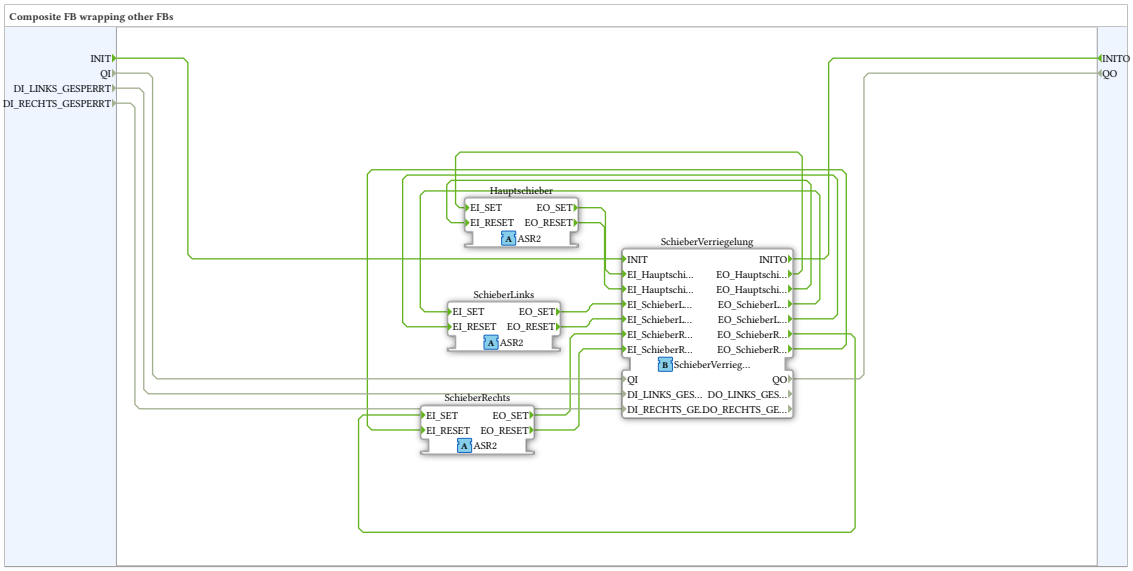
- **Integrierte Kollisionsvermeidung**: Die Verriegelungslogik ist fest im Zustandsautomaten hinterlegt und muss nicht extern verdrahtet werden.
- **Zustandsbasierte Koordination**: Die Ausgangsbefehle sind immer konsistente Sets (`Open`/`Close`-Kombinationen für alle drei Schieber).
- **Explizite Sperreingänge**: Die Berücksichtigung externer Sperrsignale (`DI_*_GESPERRT`) ist integraler Bestandteil der Steuerlogik.

Ein Nachteil ist die geringere Flexibilität. Die Logik ist spezifisch für genau drei Schieber mit dieser speziellen Verriegelungsbeziehung. Für andere Anzahlen oder Abhängigkeiten muss ein neuer Baustein erstellt werden.

Fazit

Der `SchieberVerriegelung`-Funktionsblock ist ein spezialisierter Steuerbaustein für die koordinierte Ansteuerung von drei mechanisch oder prozesslich gekoppelten Schiebern. Seine Stärke liegt in der robusten, zustandsbasierten Implementierung der Verriegelungslogik, die sichere und konsistente Schieberstellungen garantiert. Er eignet sich ideal für fest definierte Anlagenabschnitte mit klaren Abhängigkeiten zwischen den Aktoren und entlastet den übergeordneten Steuerungsentwurf von der Verdrahtung dieser Sicherheitslogik.

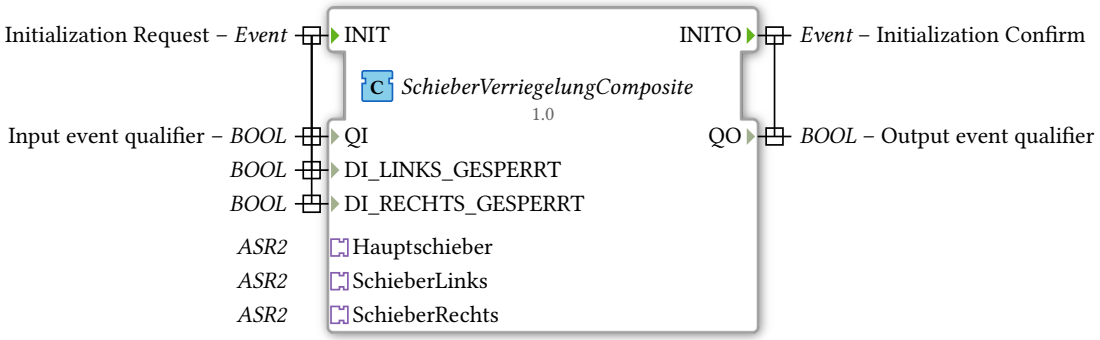
SchieberVerriegelungComposite



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `SchieberVerriegelungComposite` ist ein Composite-FB, der als Wrapper für andere Funktionsblöcke dient. Sein Hauptzweck ist die Verwaltung und Koordination der Verriegelungslogik für mehrere Schieber (Hauptschieber, Schieber links, Schieber rechts). Er kapselt die interne Logik und bietet eine vereinheitlichte Schnittstelle für die Initialisierung und den Datenaustausch mit den angeschlossenen Aktoren und Sensoren.



Schnittstellenstruktur

Ereignis-Eingänge

- **`INIT`**: Initialisierungsanforderung. Löst den Initialisierungsvorgang des internen `SchieberVerriegelung`-FB und damit des gesamten Composite-Blocks aus. Wird mit den Daten `QI`, `DI_LINKS_GESPERRT` und `DI_RECHTS_GESPERRT` verknüpft.

Ereignis-Ausgänge

- **`INITO`**: Initialisierungsbestätigung. Wird vom internen `SchieberVerriegelung`-FB ausgelöst und signalisiert den erfolgreichen Abschluss der Initialisierung. Ist mit dem Datenausgang `QO` verknüpft.

Daten-Eingänge

- **`QI` (BOOL)**: Qualifizierer für das INIT-Ereignis. Steuert die Initialisierung.
- **`DI_LINKS_GESPERRT` (BOOL)**: Status-Eingang, der anzeigt, ob der linke Schieber verriegelt (gesperrt) ist.
- **`DI_RECHTS_GESPERRT` (BOOL)**: Status-Eingang, der anzeigt, ob der rechte Schieber verriegelt (gesperrt) ist.

Daten-Ausgänge

- **`QO` (BOOL)**: Qualifizierer für das INITO-Ereignis. Gibt den Status der Initialisierung zurück.

Adapter

Der Composite-FB stellt drei bidirektionale Adapter-Schnittstellen zur Verfügung, um mit den physikalischen oder logischen Schieber-Aktoren zu kommunizieren. Jeder Adapter folgt dem Typ `adapter::types::bidirectional::ASR2`, der typischerweise SET- und RESET-Kommandos sowie entsprechende Rückmeldungen unterstützt.

1. **`Hauptschieber`**: Adapter für die Ansteuerung und Statusrückmeldung des Hauptschiebers.
2. **`SchieberLinks`**: Adapter für die Ansteuerung und Statusrückmeldung des linken Schiebers.
3. **`SchieberRechts`**: Adapter für die Ansteuerung und Statusrückmeldung des rechten Schiebers.

Funktionsweise

Der `SchieberVerriegelungComposite` fungiert primär als Vermittler. Er leitet alle externen Ereignisse und Daten an den internen, gekapselten `SchieberVerriegelung`-Funktionsblock weiter. Ebenso werden die Ausgangsereignisse und -daten dieses internen FB an die Composite-Schnittstellen durchgereicht.

Die zentrale Logik zur Steuerung der Schieber und zur Verriegelung liegt vollständig im internen `SchieberVerriegelung`-FB. Der Composite-Block übersetzt die generischen Adapter-Ereignisse (`EI_SET`/`EI_RESET`, `EO_SET`/`EO_RESET`) in die spezifischen, benannten Ereignisse des internen FB (z.B. `EI_Hauptschieber_Open`, `EO_SchieberLinks_Close`) und umgekehrt. Dies ermöglicht eine saubere Trennung zwischen der anwendungsspezifischen Steuerlogik und der generischen Adapter-Kommunikation.

Technische Besonderheiten

- **Kapselung**: Die komplexe Verriegelungslogik ist in einem separaten FB (`SchieberVerriegelung`) implementiert, was Wiederverwendbarkeit und Testbarkeit fördert.
- **Adapter-basierte Kommunikation**: Die Anbindung an die Aktoren erfolgt ausschließlich über standardisierte Adapter-Schnittstellen (`ASR2`). Dies abstrahiert die konkrete Aktorik und erhöht die Flexibilität.
- **Pass-Through-Design**: Bis auf die Übersetzung der Adapter-Ereignisnamen besitzt dieser Composite-FB keine eigene Logik. Er dient der strukturierten Organisation des FBNetworks.

Zustandsübersicht

Als Composite-FB besitzt `SchieberVerriegelungComposite` selbst keinen expliziten Zustandsautomaten. Der Systemzustand wird vollständig durch den internen `SchieberVerriegelung`-FB verwaltet. Der Composite-Block kann sich in einem der beiden folgenden Makrozustände befinden:

- **Nicht initialisiert**: Vor dem Eintreffen eines gültigen `INIT`-Ereignisses.
- **Initialisiert und betriebsbereit**: Nachdem der interne FB die Initialisierung via `INITO` quittiert hat. In diesem Zustand werden alle Adapter-Ereignisse durchgeleitet.

Anwendungsszenarien

Dieser Funktionsblock ist für Steuerungsanwendungen konzipiert, in denen mehrere mechanisch oder logisch verknüpfte Schieber oder Verschlüsse koordiniert werden müssen, z.B.:

- Verriegelungssysteme in Förderanlagen oder Silos.
- Sicherheitssteuerungen für Tore oder Klappen, die nur unter bestimmten Bedingungen geöffnet werden dürfen.
- Prozesssteuerungen, in denen die Reihenfolge von Schaltvorgängen eingehalten werden muss (z.B. Hauptschieber nur öffnen, wenn Seitenverschlüsse geschlossen sind).

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu einem einfachen, monolithischen Funktionsblock, der Logik und Schnittstellen vermischt, bietet dieser Composite-Block eine klare Trennung der Belange (Separation of Concerns). Ein direkter Vergleich wäre ein `SchieberVerriegelung`-FB, der die Adapter-Schnittstellen direkt integriert hätte. Der Composite-Ansatz ist modularer und erlaubt es, die Kernlogik (`SchieberVerriegelung`) unverändert in verschiedenen Netzwerkumgebungen wiederzuverwenden, indem nur der umschließende Composite-Block angepasst wird.

Fazit

Der `SchieberVerriegelungComposite` ist ein gut strukturierter Wrapper-Baustein, der die Integration einer komplexen Schieber-Verriegelungslogik in ein größeres 4diac-Steuerungssystem vereinfacht. Durch die Verwendung standardisierter Adapter und die Kapselung der Kernfunktionalität fördert er Wiederverwendbarkeit, Wartbarkeit und klare Schnittstellendefinitionen. Er ist ideal für Anwendungen, bei denen eine zuverlässige und übersichtliche Koordination mehrerer Aktoren erforderlich ist.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

const

START

🎧 Podcast

- Ereignisse und IEC 61499: Der Startschuss für intelligente Systeme

Einleitung

Die `START`-Konstantendeklaration definiert eine Reihe von ganzzahligen Konstanten (vom Typ `UINT`) für die Verwendung in Steuerungsprogrammen. Diese Konstanten repräsentieren spezifische Zustände, die typischerweise in Sequenzsteuerungen, beispielsweise für Schieber, verwendet werden. Sie dienen der Verbesserung der Code-Lesbarkeit und Wartbarkeit, indem sie numerische Werte durch aussagekräftige Namen ersetzen.

Schnittstellenstruktur

Bei `START` handelt es sich um eine **GlobalConstants**-Deklaration und nicht um einen Funktionsblock. Daher besitzt sie keine ereignis- oder datenflussbasierten Schnittstellen wie Eingänge oder Ausgänge. Stattdessen definiert sie globale, konstante Werte, die innerhalb anderer Funktionsblöcke oder Programme referenziert werden können.

Ereignis-Eingänge

Nicht zutreffend. Diese Deklaration enthält keine Ereigniseingänge.

Ereignis-Ausgänge

Nicht zutreffend. Diese Deklaration enthält keine Ereignisausgänge.

Daten-Eingänge

Nicht zutreffend. Diese Deklaration enthält keine Dateneingänge.

Daten-Ausgänge

Nicht zutreffend. Diese Deklaration enthält keine Datenausgänge.

Adapter

Nicht zutreffend. Diese Deklaration verwendet keine Adapter.

Funktionsweise

Die `START`-Konstantendeklaration weist den symbolischen Namen `STARTClosed`, `STARTOpened` und `STARTUnknown` feste, unveränderliche numerische Werte zu. Sobald diese Deklaration in einem Projekt enthalten ist, können die definierten Namen anstelle der rohen Zahlen (5, 6, 7) im gesamten Anwendungscode verwendet werden. Der Compiler ersetzt bei der Übersetzung jeden Verweis auf z.B. `STARTClosed` durch den Wert `UINT#5`.

Technische Besonderheiten

- **Typ:** Alle Konstanten sind vom Typ `UINT` (vorzeichenlose Ganzzahl).
- **Gültigkeitsbereich:** Die Konstanten sind global definiert (`VAR_GLOBAL CONSTANT`) und innerhalb des deklarierten Pakets (`logiBUS::utils::sequence::schieber::const`) verfügbar.
- **Unveränderlichkeit:** Als Konstanten (`CONSTANT`) können ihre Werte zur Laufzeit nicht geändert werden.
- **Paketstruktur:** Die Konstanten sind in einer spezifischen Pakethierarchie organisiert, die auf ihre Anwendungsdomäne (Hilfsfunktionen für Sequenzsteuerungen von Schiebern) hinweist.

Zustandsübersicht

Die Konstanten selbst repräsentieren abstrakte Zustände:

- `STARTClosed` (Wert 5): Steht für den Zustand „geschlossen“ oder „in Ausgangsposition“.
- `STARTOpened` (Wert 6): Steht für den Zustand „geöffnet“ oder „in Endposition“.
- `STARTUnknown` (Wert 7): Steht für einen undefinierten, unbekannten oder Fehlerzustand.

Anwendungsszenarien

Primär werden diese Konstanten in Zustandsautomaten (State Machines) oder Sequenzer-Funktionsblöcken eingesetzt, um die aktuelle Position oder den Status eines Aktors (wie einen linearen Schieber) eindeutig und lesbar zu kodieren. Beispielsweise könnte ein FB seinen internen Zustand oder einen Ausgangswert auf `STARTOpened` setzen, um anzuzeigen, dass der Schieber vollständig geöffnet ist.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu Funktionsblöcken wie `E_SR` (Flip-Flop) oder `E_CYCLE` (Zyklusgeber) besitzt `START` keinerlei Algorithmus oder Verhaltenslogik. Es ist eine reine Datendeklaration. Ähnliche Konstrukte wären andere `GlobalConstants-` oder `Type`-Deklarationen, die ebenfalls symbolische Namen für Werte bereitstellen (z.B. für Fehlercodes oder Betriebsmodi).

Fazit

Die `START`-Konstantendeklaration ist ein einfaches, aber wesentliches Werkzeug zur Strukturierung und Professionalisierung von IEC 61499-basierten Steuerungsanwendungen. Durch die Verwendung bedeutungsvoller Namen anstelle von „Magic Numbers“ wird der Code deutlich verständlicher, weniger fehleranfällig und einfacher zu warten. Ihre Integration ist besonders in größeren Projekten oder wiederverwendbaren Bibliothekskomponenten zu empfehlen.

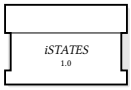
STATES

States (Closed, Opened, Transitions)

Global Constants

Name	Type	Initial Value
Closed	STRING	‚Closed‘
Opening	STRING	‚Opening‘
Opened	STRING	‚Opened‘
Closing	STRING	‚Closing‘
Unknown	STRING	‚Unknown‘

iSTATES



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Baustein **iSTATES** ist ein spezieller GlobalConstants-Baustein der 4diac-IDE. Er definiert einen Satz benannter Konstanten, die die möglichen Zustände eines (Schieber-)Aktors abbilden. Die Konstanten sind als `USINT`-Werte hinterlegt und ermöglichen eine einheitliche, menschenlesbare Referenz auf die Zustände in der Steuerungslogik.

Schnittstellenstruktur

Ereignis-Eingänge

Keine. Der Baustein besitzt keine Ereignis-Eingänge.

Ereignis-Ausgänge

Keine. Der Baustein besitzt keine Ereignis-Ausgänge.

Daten-Eingänge

Keine. Der Baustein besitzt keine Daten-Eingänge.

Daten-Ausgänge

Keine. Der Baustein besitzt keine Daten-Ausgänge. Die definierten Konstanten stehen als global verfügbare Symbole im Projektkontext zur Verfügung.

Adapter

Keine.

Funktionsweise

Der Baustein **iSTATES** stellt fünf symbolische Konstanten bereit, die in der gesamten Applikation verwendet werden können. Jede Konstante repräsentiert einen diskreten Zustand eines Schiebers oder ähnlichen Aktors. Die Werte sind als `USINT` (unsigned short integer) definiert und wie folgt belegt:

Konstante	Wert (USINT)	Bedeutung
Unknown	0	Unbekannter Zustand
Closed	1	Geschlossen
Opening	2	Öffnet
Opened	3	Geöffnet
Closing	4	Schließt

Diese Werte können in FB-Netzwerken direkt zur Initialisierung von Variablen oder zum Vergleich mit Zustandsrückmeldungen verwendet werden.

Technische Besonderheiten

- **Typ: GlobalConstants** – Der Baustein wird nicht instanziiert, sondern als globale Konstantendefinition in das Projekt eingebunden.
- **Wertebereich**: Alle Zustände sind als `USINT` deklariert, was einen geringen Speicherbedarf garantiert und mit vielen Feldbus-Protokollen kompatibel ist.
- **Erweiterbarkeit**: Das Schema kann leicht um weitere Zustände ergänzt werden, sofern die Applikation dies erfordert.
- **Paket**: Der Baustein ist im Paket `logiBUS::utils::sequence::schieber::const` organisiert, was eine klare Strukturierung innerhalb des Projekts erlaubt.

Zustandsübersicht

Die fünf Zustände decken einen vollständigen Lebenszyklus eines Schiebers ab:

1. **Unknown** (0) – Initialzustand oder Fehlerzustand, wenn die aktuelle Position nicht bekannt ist.
2. **Closed** (1) – Schieber ist geschlossen (Endlage zu).
3. **Opening** (2) – Schieber befindet sich im Öffnungsvorgang.
4. **Opened** (3) – Schieber ist vollständig geöffnet (Endlage auf).
5. **Closing** (4) – Schieber befindet sich im Schließvorgang.

Anwendungsszenarien

- **Zustandsautomaten für Schieber, Ventile oder Tore** – Der Baustein liefert die Basis für die Zustandsverwaltung in sequentiellen Steuerungen.
- **Visualisierung** – Die symbolischen Namen können in HMI- oder SCADA-Systemen zur Anzeige des aktuellen Zustands verwendet werden.
- **Fehlererkennung** – Durch den `Unknown`-Zustand können ungültige oder initiale Zustände eindeutig identifiziert werden.

Vergleich mit ähnlichen Bausteinen

Ähnliche GlobalConstants-Bausteine existieren oft in anderen Paketen (z.B. für Motorsteuerungen, Fördereinrichtungen). Der Unterschied liegt in der spezifischen Definition der Zustände – während andere Bausteine `Idle`, `Running`, `Error` etc. verwenden, ist **iSTATES** speziell auf die fünf typischen Schieberzustände zugeschnitten. Die Verwendung von `USINT` anstelle von `BOOL` oder `INT` ist plattformeffizient und standardkonform.

Fazit

Der Baustein **iSTATES** ist ein einfaches, aber nützliches Hilfsmittel zur konsistenten Definition von Schieber-Zuständen. Durch die Verwendung globaler Konstanten wird der Code lesbarer, fehlerresistenter und wartbarer. Er eignet sich hervorragend für alle Applikationen, in denen ein Aktor die Zustandsfolge „Geschlossen → Öffnet → Geöffnet → Schließt → Geschlossen“ durchläuft.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

str

🎧 Podcast

- 4diac IDE: Dein Open-Source-Werkzeugkasten für verteilte Industrieautomatisierung nach IEC 61499
- 4diac IDE: Wie der IEC 61499 Standard die Industrieautomatisierung revolutioniert
- 4diac-Präsentation: Zielgruppen, Struktur und Alleinstellungsmerkmal Schärfen
- Den Software-Drachen zähmen: Industrielle Automatisierung und die Zukunft der Produktion
- Eclipse 4diac FORTE: IEC 61499 verstehen – Der LEGO®-Baukasten für Ihre Industrie 4.0 Steuerung

---

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SchieberAuxInStruct

• – ◦ • – ◦ • – ◦ •

Einleitung

Der strukturierte Datentyp `SchieberAuxInStruct` dient der Bündelung von Informationen zur visuellen Darstellung eines Hilfselements (AUX) in einer Schieber-Steuerung. Er kombiniert die Angabe eines Bildes mit einer Hintergrundfarbe, um die grafische Repräsentation eines Hilfseingangs oder -elements zu definieren.

Schnittstellenstruktur

Dieser Baustein ist ein strukturierter Datentyp (Struct) und besitzt daher keine Ereignis- oder Adapterschnittstellen. Seine Struktur besteht aus den folgenden zwei Datenelementen:

Daten-Eingänge

Da es sich um einen Datentyp handelt, existieren keine direkten Eingänge. Die Struktur wird als Ganzes an Daten-Eingängen anderer Funktionsblöcke verwendet.

Daten-Ausgänge

Da es sich um einen Datentyp handelt, existieren keine direkten Ausgänge. Die Struktur wird als Ganzes an Daten-Ausgängen anderer Funktionsblöcke verwendet.

Strukturierte Elemente

- **`Picture`** (`logiBUS::utils::sequence::schieber::SchieberStruct`):
  - **Kommentar:** `u16ObjId for Picture`
  - **Beschreibung:** Enthält die Kennung (Object ID) für das anzuzeigende Bild des Hilfselements. Der zugrundeliegende Typ `SchieberStruct` stellt vermutlich die notwendigen Informationen zur Referenzierung und Handhabung des Bildobjekts bereit.
- **`u8Colour`** (`logiBUS::utils::sequence::schieber::SchieberColorStruct`):
  - **Kommentar:** `Background Color of the AUX`
  - **Beschreibung:** Definiert die Hintergrundfarbe des Hilfselements. Der Typ `SchieberColorStruct` spezifiziert die verwendete Farbe, wahrscheinlich in einem für das Zielsystem (z.B. ein HMI) geeigneten Format.

Funktionsweise

Der Datentyp `SchieberAuxInStruct` ist eine passive Datenstruktur. Seine Funktion besteht ausschließlich darin, die beiden Eigenschaften `Picture` (Bild) und `u8Colour` (Hintergrundfarbe) logisch zu einem einzigen, wiederverwendbaren Datensatz zusammenzufassen. Er wird verwendet, um diese visuellen Attribute konsistent zwischen verschiedenen Funktionsblöcken innerhalb einer Anwendung zu transportieren.

Technische Besonderheiten

- **Namespaces:** Der Datentyp und seine Komponenten sind im Paket `logiBUS::utils::sequence::schieber` definiert, was auf eine spezifische Anwendung in der Steuerungstechnik für Landmaschinen (Agrartechnik) hindeutet.
- **Typisierung:** Die Verwendung spezifischer, benutzerdefinierter Typen (`SchieberStruct`, `SchieberColorStruct`) für die Mitglieder fördert die Typsicherheit und Wiederverwendbarkeit im gesamten Projekt.

Zustandsübersicht

Als reiner Datentyp besitzt `SchieberAuxInStruct` keinen internen Zustand oder Zustandsautomaten.

Anwendungsszenarien

Dieser Datentyp wird typischerweise in SPS-Steuerungen (speicherprogrammierbare Steuerungen) für mobile Arbeitsmaschinen verwendet, insbesondere in:

- **HMI-Konfiguration:** Zur Definition des Erscheinungsbildes von Bedien- oder Anzeigeelementen (z.B. Schieber, Taster, Statusleuchten) auf einem Bedienpanel.
- **Sequenzsteuerungen:** Innerhalb von Ablaufsteuerungen, bei denen sich die Darstellung eines Elements (Farbe, Bild) in Abhängigkeit vom Prozesszustand ändern soll.
- **Datenkapselung:** Vereinfachung der Parametrisierung von Funktionsblöcken, die das Aussehen eines „AUX“-Elements steuern, indem alle visuellen Parameter in einer einzigen Variable übergeben werden können.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu Funktionsblöcken (FBs) besitzt dieser Baustein keinerlei Algorithmus oder Verhaltenslogik. Er ist mit einem `STRUCT` in der IEC 61131-3 oder einer `class`/`struct` in höheren Programmiersprachen vergleichbar. Andere ähnliche Elemente wären einfache Datentypen (z.B. `ARRAY` von Einzelwerten), die jedoch die semantische Gruppierung und Typsicherheit dieses strukturierten Typs nicht bieten.

Fazit

Der `SchieberAuxInStruct` ist ein grundlegender, wiederverwendbarer Datentyp zur Modellierung der visuellen Eigenschaften eines Hilfselements in einer spezifischen Steuerungsumgebung. Durch die Kap-

selung von Bild- und Farbinformationen trägt er zur besseren Strukturierung, Lesbarkeit und Wartbarkeit des Anwendungscodes bei. Seine Verwendung ist vor allem in Projekten mit komplexen Benutzerschnittstellen oder statusabhängigen Visualisierungen vorteilhaft.

SchieberColorStruct

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Datentyp `SchieberColorStruct` ist eine strukturierte Datenstruktur, die zur Verwaltung von Farbwerten für verschiedene Zustände eines Schiebers (z. B. eines Ventils, einer Klappe oder eines Verschlusses) dient. Er definiert eine konsistente Farbkodierung für die visuelle Darstellung der Zustände `Unbekannt`, `Geschlossen`, `Öffnend`, `Geöffnet` und `Schließend` in einer Benutzeroberfläche oder Anzeige.

Schnittstellenstruktur

Der Datentyp ist eine Struktur (`STRUCT`) und enthält keine Ereignisse oder Adapter. Seine Schnittstelle besteht ausschließlich aus Daten-Elementen.

Daten-Eingänge

Diese Struktur besitzt keine Ereignis-Eingänge im herkömmlichen Sinne, da es sich um einen Datentyp handelt. Die folgenden Elemente sind die Mitgliedsvariablen der Struktur, die bei der Instanziierung beschrieben oder mit Werten belegt werden können.

Daten-Ausgänge

Diese Struktur besitzt keine Ereignis-Ausgänge. Die Mitgliedsvariablen stellen die gespeicherten Daten dar.

Daten-Elemente (Struktur-Mitglieder)

Name	Datentyp	Kommentar	Initialwert
`Unknown`	`USINT`	u8Colour for Unknown State	`COLOR_WHITE`
`Closed`	`USINT`	u8Colour for Closed State	`COLOR_WHITE`
`Opening`	`USINT`	u8Colour for Opening State	`COLOR_WHITE`
`Opened`	`USINT`	u8Colour for Opened State	`COLOR_WHITE`
`Closing`	`USINT`	u8Colour for Closing State	`COLOR_WHITE`

Adapter

Diese Datenstruktur verwendet keine Adapter.

Funktionsweise

Die `SchieberColorStruct` ist ein reiner Datentyp. Sie dient als Container, um fünf verschiedene Farbwerte (jeweils als 8-Bit-unsigned integer, `USINT`) für die fünf definierten Schieber-Zustände zu gruppieren. Durch die Verwendung dieser Struktur kann ein Funktionsblock oder eine Anwendung eine einzige, konsistente Variable übergeben oder speichern, die alle für die Darstellung benötigten Farben enthält. Die Initialwerte aller Mitglieder sind auf die Konstante `COLOR_WHITE` gesetzt, die aus dem importierten Paket `isobus::UT::Q::const::colours` stammt.

Technische Besonderheiten

- **Importierte Konstante:** Die Struktur ist von der extern definierten Konstante `COLOR_WHITE` abhängig, die über den Compiler-Import `isobus::UT::Q::const::colours::COLOR_WHITE` bereitgestellt wird. Dies stellt sicher, dass ein projektweiter, einheitlicher Weißton verwendet wird.
- **Typensicherheit:** Die Verwendung eines strukturierten Typs anstelle von fünf einzelnen Variablen erhöht die Lesbarkeit und Wartbarkeit des Codes und verringert die Fehleranfälligkeit.
- **Hash-Attribut:** Die Struktur enthält das Attribut `eclipse4diac::core::TypeHash` mit einem leeren Wert. Dies wird vom 4diac-System für die Typidentifikation und -verwaltung genutzt.

Zustandsübersicht

Da es sich um einen statischen Datentyp handelt, besitzt die `SchieberColorStruct` selbst keinen Zustandsautomaten. Sie kodiert vielmehr die Darstellungsattribute für die Zustände eines anderen, übergeordneten Elements (z. B. eines `Schieber`-Funktionsblocks).

Anwendungsszenarien

- **Visualisierung in HMIs:** Übergabe der Farbkonfiguration an Visualisierungskomponenten, um den Zustand eines Schiebers farblich korrekt anzuzeigen (z. B. Rot für „Schließend“, Grün für „Geöffnet“).
- **Konfiguration von Funktionsblöcken:** Ein Funktionsblock, der einen physischen Schieber steuert, kann eine Instanz dieses Typs als konfigurierbaren Eingang besitzen, um benutzerdefinierte Farben für die Zustandsrückmeldung zu ermöglichen.
- **Datenkapselung:** Bündelung aller zustandsbezogenen Farbinformationen in einem einzigen Parameter, der zwischen verschiedenen Softwaremodulen (z. B. Steuerung und Visualisierung) ausgetauscht werden kann.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu Funktionsblöcken (FBs) wie `E_SR` oder `E_D_FF`, die eine Logik implementieren, ist `SchieberColorStruct` ein reiner **Datentyp** (STRUCT). Ähnliche strukturierte Typen könnten z. B. `MotorConfigStruct` (für Konfigurationsdaten) oder `AxisPositionStruct` (für mehrdimensionale Positionen) sein. Seine spezifische Aufgabe ist die Definition eines Farbschemas für eine vordefinierte Menge von Zuständen.

Fazit

Die `SchieberColorStruct` ist ein einfacher, aber nützlicher strukturierter Datentyp, der die Verwaltung und den Austausch von Farbwerten für die Zustandsdarstellung eines Schiebers standardisiert. Sie fördert saubere Schnittstellen, erleichtert die Wiederverwendung und trägt durch die Verwendung initialer Standardwerte und importierter Konstanten zur Konsistenz innerhalb eines Projekts bei.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SchieberStruct

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `SchieberStruct` ist ein strukturierter Datentyp, der zur Abbildung aller möglichen Zustände eines Schiebers (z. B. eines Ventils, einer Klappe oder eines Verschlusses) innerhalb eines Steuerungssystems dient. Er dient als zentraler Container für die Objekt-Identifikatoren (IDs), die den verschiedenen Zuständen des Schiebers zugeordnet sind. Dieser Datentyp wird typischerweise in Verbindung mit Funktionsblöcken verwendet, die Zustandsmaschinen oder Sequenzen für Schiebersteuerungen implementieren.

Schnittstellenstruktur

Der `SchieberStruct` ist ein Datentyp und besitzt daher keine Ereignis- oder Adapterschnittstellen im herkömmlichen Sinne. Seine Struktur besteht ausschließlich aus Datenfeldern.

Daten-Eingänge

Nicht zutreffend. Die Struktur definiert einen Datentyp, keine Funktionsblock-Schnittstellen.

Daten-Ausgänge

Nicht zutreffend. Die Struktur definiert einen Datentyp, keine Funktionsblock-Schnittstellen.

Adapter

Nicht zutreffend.

Funktionsweise

Die `SchieberStruct` ist eine Datenstruktur, die fünf Felder vom Typ `UINT` (16-Bit unsigned integer) enthält. Jedes Feld repräsentiert einen spezifischen Zustand eines Schiebers und speichert die entspre-

chende Objekt-ID (`u16ObjId`). Diese IDs werden verwendet, um in einem übergeordneten System (z. B. einer Visualisierung oder einem Logikbaustein) auf die grafischen oder logischen Repräsentationen der Zustände zuzugreifen.

Die Initialwerte aller Felder sind auf die Konstante `ID_NULL` gesetzt, was einen undefinierten oder nicht zugewiesenen Zustand kennzeichnet. Diese müssen vor der Nutzung der Struktur mit gültigen Objekt-IDs belegt werden.

Technische Besonderheiten

- **Initialisierung:** Alle Mitgliedsvariablen werden standardmäßig mit dem Wert `ID_NULL` initialisiert. Dies stellt sicher, dass die Struktur in einem definierten, neutralen Zustand beginnt, bevor spezifische IDs zugewiesen werden.
- **Typensicherheit:** Durch die Verwendung eines strukturierten Datentyps anstelle einzelner, lose gekoppelter Variablen wird die Code-Wartbarkeit und Lesbarkeit verbessert.
- **Kompilierungshinweis:** Der Datentyp ist im Package `logiBUS::utils::sequence::schieber` definiert und importiert die Konstante `ID_NULL` aus dem `isobus::UT::Q::const::IDs`-Namespace.

Zustandsübersicht

Die Struktur bildet explizit folgende fünf Zustände eines Schiebers ab:

1. **`Unknown`:** Der Zustand des Schiebers ist unbekannt oder konnte nicht ermittelt werden.
2. **`Closed`:** Der Schieber befindet sich in der geschlossenen Endlage.
3. **`Opening`:** Der Schieber ist gerade dabei, sich zu öffnen (Bewegungszustand).
4. **`Opened`:** Der Schieber befindet sich in der geöffneten Endlage.
5. **`Closing`:** Der Schieber ist gerade dabei, sich zu schließen (Bewegungszustand).

Anwendungsszenarien

- **Zustandsverwaltung:** Ein übergeordneter Steuerungsbaustein kann eine Instanz dieser Struktur halten, um die aktuell gültigen Objekt-IDs für alle Schieberzustände zentral zu verwalten.
- **Parametrierung:** Die Struktur kann als konfigurierbarer Parameter an einen Funktionsblock übergeben werden, der dann basierend auf seinem internen Zustand die entsprechende ID aus der Struktur auswählt und z. B. an ein Visualisierungssystem weiterleitet.
- **Visualisierung:** Ein HMI- oder SCADA-System kann die in der Struktur hinterlegten IDs nutzen, um das grafische Symbol des Schiebers entsprechend seines aktuellen Zustands (z. B. Farbe, Symbol) zu aktualisieren.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu Funktionsblöcken wie `E_SR` (Speicherglied) oder `E_CTUD` (Zähler), die aktive Logik implementieren, ist `SchieberStruct` ein passiver Datentyp. Ähnliche strukturierte Datentypen könnten für andere Geräte mit Zustandsmaschinen definiert werden (z. B. `PumpeStruct`, `MotorStruct`). Die Besonderheit liegt in der spezifischen Abbildung der für einen linearen Schieber charakteristischen Zustände `Opening`, `Closing`, `Opened` und `Closed` inklusive eines `Unknown`-Zustands.

Fazit

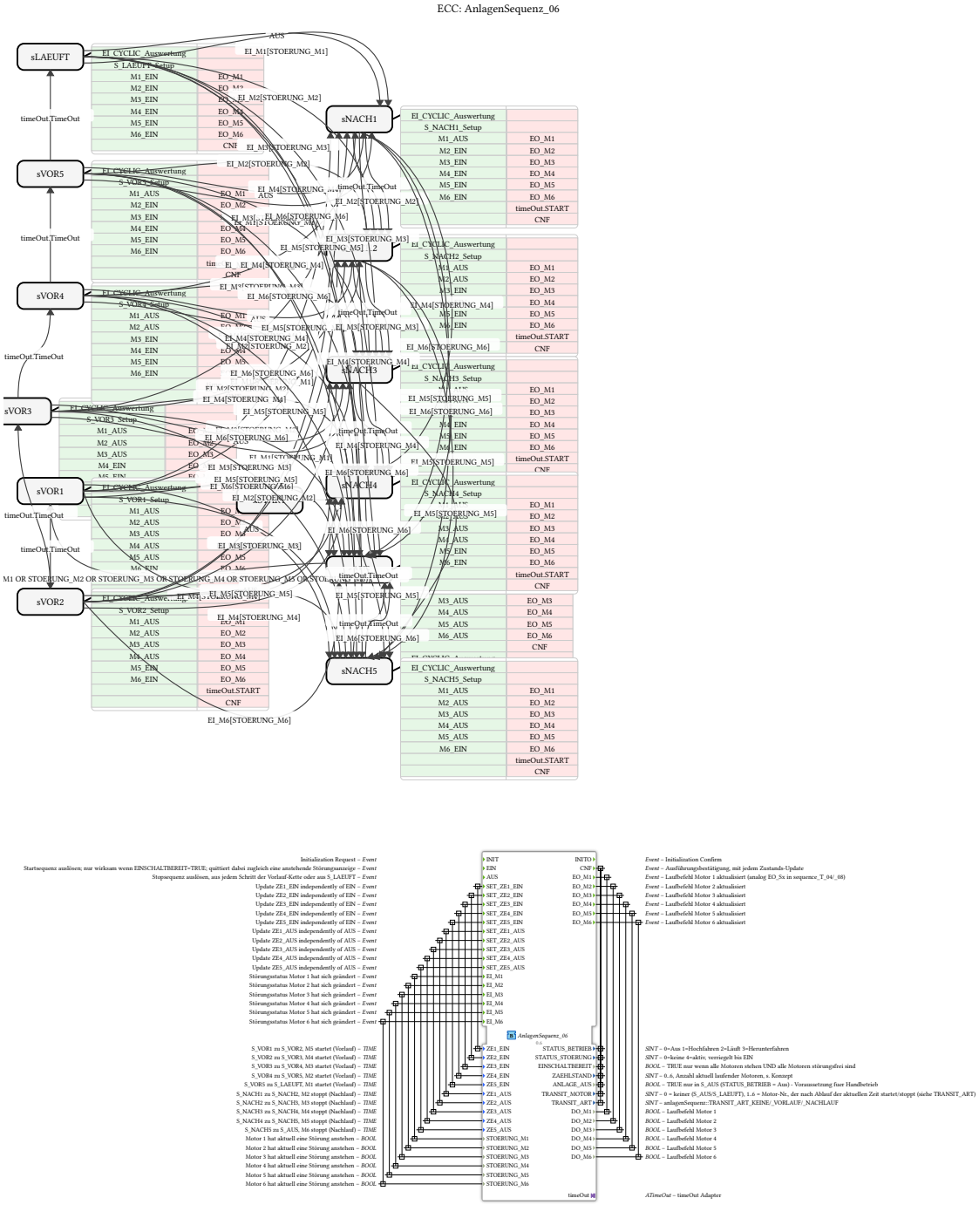
Der `SchieberStruct` Datentyp ist ein nützliches Hilfskonstrukt für die strukturierte und typsichere Handhabung von Zustandsinformationen für Schieberaktoren. Durch die Kapselung der relevanten Objekt-IDs in einer einzigen Struktur wird die Konfiguration und Wartung von Anwendungen vereinfacht, die solche Aktoren steuern und visualisieren. Seine Verwendung fördert einen klaren und konsistenten Programmierstil.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de

timed

AnlagenSequenz_06



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock `AnlagenSequenz_06` ist ein zeitgesteuerter Ring-Sequenzer für die geordnete Start- und Stopp-Reihenfolge von sechs Motoren (M1..M6), wie sie z. B. bei einer Getreideannahme-Förderkette benötigt wird. Anders als die generischen Bausteine sequence_T_04/sequence_T_08 ist er kein Allzweck-Sequenzer, sondern hart auf dieses eine Anwendungsmuster zugeschnitten: Vorlauf (Motoren schalten von hinten nach vorne der Reihe nach ein) und Nachlauf (Motoren schalten von vorne nach hinten der Reihe nach aus) sind zwei getrennte, rein lineare Ketten, die über die beiden Ring-Pole `sAUS` (unten, alle Motoren aus) und `sLAEUFT` (oben, alle Motoren an) zu einem geschlossenen Ring verbunden sind.

Zusätzlich besitzt der Baustein eine vollständige Störungs-Kaskade: Meldet einer der sechs Motoren während des Hoch- oder Herunterfahrens oder im Dauerbetrieb eine Störung, springt der FB in einem einzigen Schritt (kein Nachfassen über mehrere Zyklen) in den passenden Nachlauf-Zustand, der genau die zu diesem Zeitpunkt noch laufenden vorgeordneten Motoren berücksichtigt.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung (Typ EInit).
- **EIN**: Startet die Vorlauf-Kette (Übergang sAUS → sVOR1). Nur wirksam, wenn EINSCHALTBEREIT = TRUE (Bedingung EIN[NOT (STOERUNG_M1 OR ... OR STOERUNG_M6)]); quittiert dabei zugleich eine anstehende, verriegelte STATUS_STOERUNG-Anzeige.
- **AUS**: Löst die Stopp-Sequenz aus jedem Schritt der Vorlauf-Kette (sVOR1..sVOR5) oder aus sLAEUFT aus. Springt auf den jeweiligen Spiegelpunkt der Nachlauf-Kette (sVOR_k → sNACH_(6-k)), da hier kein Motor-Fehler, sondern nur ein Bedienerwunsch vorliegt.
- **SET_ZE1_EIN, SET_ZE2_EIN, SET_ZE3_EIN, SET_ZE4_EIN, SET_ZE5_EIN**: Aktualisieren ZE1_EIN .. ZE5_EIN unabhängig von EIN; jedes Ereignis trägt (With) die gleichnamige Zeitvariable (SET_ZEk_EIN → ZEk_EIN).
- **SET_ZE1_AUS, SET_ZE2_AUS, SET_ZE3_AUS, SET_ZE4_AUS, SET_ZE5_AUS**: Aktualisieren ZE1_AUS .. ZE5_AUS unabhängig von AUS; jedes Ereignis trägt (With) die gleichnamige Zeitvariable (SET_ZEk_AUS → ZEk_AUS).
- **EI_M1**: Störungsstatus Motor 1 hat sich geändert. Führt (With STOERUNG_M1) aus jedem Zustand vor sNACH1 direkt dorthin.
- **EI_M2**: Störungsstatus Motor 2 hat sich geändert. Führt (With STOERUNG_M2) direkt nach sNACH2.
- **EI_M3**: Störungsstatus Motor 3 hat sich geändert. Führt (With STOERUNG_M3) direkt nach sNACH3.
- **EI_M4**: Störungsstatus Motor 4 hat sich geändert. Führt (With STOERUNG_M4) direkt nach sNACH4.
- **EI_M5**: Störungsstatus Motor 5 hat sich geändert. Führt (With STOERUNG_M5) direkt nach sNACH5.
- **EI_M6**: Störungsstatus Motor 6 hat sich geändert. Führt (With STOERUNG_M6) direkt nach sAUS (M6 ist der letzte/äußerste Motor der Kette, es gibt keinen kürzeren Nachlauf-Schritt).

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung (Typ EInit).
- **CNF**: Ausführungsbestätigung, ausgelöst bei jedem Zustandswechsel; liefert STATUS_BETRIEB, STATUS_STOERUNG, ZAEHLSTAND, EINSCHALTBEREIT, ANLAGE_AUS, TRANSIT_MOTOR und TRANSIT_ART.
- **EO_M1**: Laufbefehl Motor 1 aktualisiert; liefert DO_M1 (analog EO_Sx in sequence_T_04/_08).
- **EO_M2**: Laufbefehl Motor 2 aktualisiert; liefert DO_M2.
- **EO_M3**: Laufbefehl Motor 3 aktualisiert; liefert DO_M3.
- **EO_M4**: Laufbefehl Motor 4 aktualisiert; liefert DO_M4.
- **EO_M5**: Laufbefehl Motor 5 aktualisiert; liefert DO_M5.
- **EO_M6**: Laufbefehl Motor 6 aktualisiert; liefert DO_M6.

Daten-Eingänge

- **ZE1_EIN .. ZE5_EIN** (TIME; einzeln ZE1_EIN, ZE2_EIN, ZE3_EIN, ZE4_EIN, ZE5_EIN): Verweildauer je Vorlauf-Schritt (sVOR1→sVOR2 bis sVOR5→sLAEUFT), bevor automatisch der nächste Motor dazugeschaltet wird. Initialwert: NO_TIME.
- **ZE1_AUS .. ZE5_AUS** (TIME; einzeln ZE1_AUS, ZE2_AUS, ZE3_AUS, ZE4_AUS, ZE5_AUS): Verweildauer je Nachlauf-Schritt (sNACH1→sNACH2 bis sNACH5→sAUS), bevor automatisch der nächste Motor abgeschaltet wird. Initialwert: NO_TIME.
- **STOERUNG_M1 .. STOERUNG_M6** (BOOL): Live-Störungssignal je Motor, dauerhaft (kein Puls). Initialwert: FALSE.

Daten-Ausgänge

- **STATUS_BETRIEB** (SINT): 0=Aus, 1=Hochfahren, 2=Läuft, 3=Herunterfahren.
- **STATUS_STOERUNG** (SINT): 0=keine, 4=aktiv — bleibt bis zur nächsten erfolgreichen EIN-Quittierung verriegelt (siehe Technische Besonderheiten).
- **EINSCHALTBEREIT** (BOOL): TRUE nur wenn ZAEHLSTAND = 0 (alle Motoren stehen) UND alle sechs STOERUNG_Mx aktuell FALSE sind.
- **ZAEHLSTAND** (SINT): 0..6, Anzahl der aktuell laufenden Motoren.
- **ANLAGE_AUS** (BOOL): TRUE nur in sAUS (STATUS_BETRIEB = Aus) — Voraussetzung für Handbetrieb.

- **`TRANSIT_MOTOR`** (SINT): 0 = keiner (`sAUS`/`sLAEUFT`), 1..6 = Nummer des Motors, der nach Ablauf der aktuellen Zeit startet/stoppt (siehe `TRANSIT_ART`).
- **`TRANSIT_ART`** (SINT): Art des Übergangs: `anlagenSequenz::TRANSIT_ART_KEINE`, `_VORLAUF` oder `_NACHLAUF`.
- **`DO_M1 .. DO_M6`** (BOOL): Laufbefehl je Motor.

Adapter-Anschlüsse

- **`timeOut`** (Plug, Typ: `iec61499::events::ATimeOut`): Standardisierter TimeOut-Adapter für die Zeitsteuerung der Vor- und Nachlauf-Schritte, analog zur Nutzung in `sequence_T_04`.

Funktionsweise

Der FB ist ein Basic Function Block (BFB) mit 12 realen Zuständen plus Startzustand, angeordnet als Ring aus zwei linearen Ketten:

1. **Vorlauf-Kette** (`sVOR1` ⟶ `sVOR2` ⟶ `sVOR3` ⟶ `sVOR4` ⟶ `sVOR5` ⟶ `sLAEUFT`): Jeder Schritt schaltet genau einen weiteren Motor dazu, in der Reihenfolge M6, M5, M4, M3, M2, M1 (die Motoren mit der höchsten Nummer starten zuerst). Der Übergang zum jeweils nächsten Schritt erfolgt automatisch nach Ablauf der zugehörigen `ZEk_EIN`-Zeit (`timeOut.TimeOut`).
2. **Nachlauf-Kette** (`sNACH1` ⟶ `sNACH2` ⟶ `sNACH3` ⟶ `sNACH4` ⟶ `sNACH5` ⟶ `sAUS`): Spiegelbild des Vorlaufs — jeder Schritt schaltet genau einen Motor ab, in der Reihenfolge M1, M2, M3, M4, M5, M6.
3. **Start**: `EIN` (nur aus `sAUS`, nur wenn `EINSCHALTBEREIT`) quittiert eine anstehende `STATUS_STOERUNG`-Anzeige und aktiviert `sVOR1`.
4. **Geplanter Stopp**: `AUS` aus jedem Vorlauf-Schritt oder aus `sLAEUFT` springt auf den Spiegelpunkt der Nachlauf-Kette (`sVOR_k` ⟶ `sNACH_(6-k)`, `sLAEUFT` ⟶ `sNACH1`) — es werden nur die zu diesem Zeitpunkt tatsächlich laufenden Motoren nachträglich der Reihe nach abgeschaltet, keine bereits stehenden erneut angefasst.
5. **Störungs-Kaskade**: Jedes `EI_Mx`-Ereignis (`With STOERUNG_Mx`) hat aus jedem Zustand vor seinem Zielschritt eine direkte Ein-Schritt-Transition zum passenden `sNACH_x` (bzw. zu `sAUS` bei `x=6`). Dadurch reagiert der FB auf jede Störung sofort, unabhängig davon, ob sich der Ring gerade im Vor- oder Nachlauf oder im Dauerbetrieb (`sLAEUFT`) befindet — ohne Umweg über einen Spiegelpunkt und ohne mehrzyklisches Nachfassen.
6. **Dauerbetrieb**: `sLAEUFT` ist der obere Ring-Pol — alle sechs Motoren laufen, keine Zeitsteuerung aktiv, der FB verbleibt hier bis `AUS` oder einer Störungsmeldung.
7. **Zyklusende**: Nach `sNACH5` (oder direkt aus einer Störung) erreicht der Ring wieder `sAUS` — den Ruhezustand, aus dem heraus ein neuer `EIN`-Zyklus beginnen kann.

Technische Besonderheiten

- **`EINSCHALTBEREIT` ist bewusst NICHT an `STATUS_STOERUNG` gekoppelt.** `STATUS_STOERUNG` bleibt nach einer Störung bis zur nächsten erfolgreichen `EIN`-Quittierung verriegelt (dient der Anzeige in der Visu). Würde `EINSCHALTBEREIT` davon abhängen, könnte der `EIN`-Taster nach der allerersten jemals aufgetretenen Störung nie wieder freigegeben werden — echter Deadlock. `EINSCHALTBEREIT` prüft stattdessen die *live* `STOERUNG_Mx`-Signale direkt (`((ZAEHLSTAND = 0) AND NOT (STOERUNG_M1 OR ... OR STOERUNG_M6))`).
- **Der `sAUS → sVOR1`-Übergang trägt zusätzlich die volle Interlock-Bedingung im Event-Bracket** (`EIN[NOT (STOERUNG_M1 OR ... OR STOERUNG_M6)]`), nicht nur eine Prüfung auf den berechneten Status. Ein reiner Statuswert wie `EINSCHALTBEREIT` erzwingt für sich genommen nichts — nur eine tatsächliche `ECTransition Condition` verhindert den Start bei anstehender Störung.
- **Events sind polar und werden nicht mit `AND`/`OR` verknüpft.** Alle Bedingungen, die ein Ereignis mit einer Datenbedingung kombinieren, nutzen ausschließlich die Klammer-Syntax `EventName[boolean_ausdruck]` (z. B. `EI_M3[STOERUNG_M3]`) — die einzige gültige Form dafür in IEC 61499.
- **Keine Selbstschleifen mehr im ECC.** Eine frühere Fassung nutzte ein zyklisches `EI_CYCLIC` -Ereignis zum Nachfassen über mehrere Zyklen; das ist entfernt. `EI_CYCLIC_Auswertung` läuft stattdessen als Entry-Algorithmus in jedem aktiven Zustand mit und wertet `STOERUNG_M1..M6` bei jedem Zustandswechsel neu aus.
- **Direkte Ein-Schritt-Störungsdispatch statt Spiegelpunkt-Umweg.** Jeder Zustand vor einem `sNACH_x` hat eine eigene, direkte `EI_Mx`-Transition dorthin (51 Kanten insgesamt für die Störungs-Kaskade) — kein Zwischenschritt über einen Spiegelpunkt wie beim geplanten `AUS`.
- **`MaxBetriebRest`/Nachlauf-Reste (Demo-Server-Schema) sind noch nicht abgebildet** — bewusste, dokumentierte Vereinfachung der aktuellen Fassung (s. Kommentar im FB-Header).

Zustandsübersicht

Zustandsname	Beschreibung	Aktive Motoren	Übergangsbedingung zum nächsten Zustand
xSTART	Initialer Idle-Zustand.	—	1 (sofort nach sAUS)
sAUS	Ring-Grundzustand, unten.	keine	EIN[NOT (STOERUNG_M1..M6)] → sVOR1
sVOR1	Vorlauf-Schritt 1.	M6	timeOut.TimeOut → sVOR2; AUS → sNACH5; EI_Mx → passendes sNACHx/sAUS
sVOR2	Vorlauf-Schritt 2.	M5, M6	timeOut.TimeOut → sVOR3; AUS → sNACH4; EI_Mx → passendes sNACHx/sAUS
sVOR3	Vorlauf-Schritt 3.	M4, M5, M6	timeOut.TimeOut → sVOR4; AUS → sNACH3; EI_Mx → passendes sNACHx/sAUS
sVOR4	Vorlauf-Schritt 4.	M3, M4, M5, M6	timeOut.TimeOut → sVOR5; AUS → sNACH2; EI_Mx → passendes sNACHx/sAUS
sVOR5	Vorlauf-Schritt 5.	M2, M3, M4, M5, M6	timeOut.TimeOut → sLAEUFT; AUS → sNACH1; EI_Mx → passendes sNACHx/sAUS
sLAEUFT	Ring-Pol, oben — Dauerbetrieb.	M1..M6 (alle)	AUS → sNACH1; EI_Mx → passendes sNACHx/sAUS
sNACH1	Nachlauf-Schritt 1 (M1 gestoppt).	M2..M6	timeOut.TimeOut → sNACH2; EI_Mx → passendes sNACHx/sAUS
sNACH2	Nachlauf-Schritt 2 (M1, M2 gestoppt).	M3..M6	timeOut.TimeOut → sNACH3; EI_Mx → passendes sNACHx/sAUS
sNACH3	Nachlauf-Schritt 3 (M1..M3 gestoppt).	M4, M5, M6	timeOut.TimeOut → sNACH4; EI_Mx → passendes sNACHx/sAUS
sNACH4	Nachlauf-Schritt 4 (M1..M4 gestoppt).	M5, M6	timeOut.TimeOut → sNACH5; EI_Mx → passendes sNACHx/sAUS
sNACH5	Nachlauf-Schritt 5 (M1..M5 gestoppt).	M6	timeOut.TimeOut → sAUS; EI_M6 → sAUS

Globale Störungs-Kaskade: Aus jedem Zustand vor einem sNACH_x führt EI_Mx[STOERUNG_Mx] direkt und ohne Umweg in genau diesen Zustand (bzw. nach sAUS bei x=6) — insgesamt 51 solcher Direkt-Kanten, siehe Netzwerkdiagramm.

Anwendungsszenarien

- **Getreideannahme-Förderkette**: Sechs in Reihe geschaltete Förderelemente (z. B. Elevator, Querförderer, Trog- und Schneckenförderer), die nur in einer festen Reihenfolge sicher an- und abgeschaltet werden dürfen, damit kein Element gegen einen stehenden Nachfolger fördert.
- **Mehrstufige Förderanlagen allgemein**: Jede Anlage mit einer festen Kaskade aus Motoren, bei der eine Störung an einer Stelle einen kontrollierten, aber sofortigen Rückzug der vorgeordneten Stufen erfordert.

- **Sicherheitskritische Ablaufketten**: Prozesse, bei denen ein einfacher zeitgesteuerter Sequenzer nicht ausreicht, weil zusätzlich eine verriegelte Störungsanzeige und ein Wiederanlauf-Interlock (`EINSCHALTBEREIT`) benötigt werden.

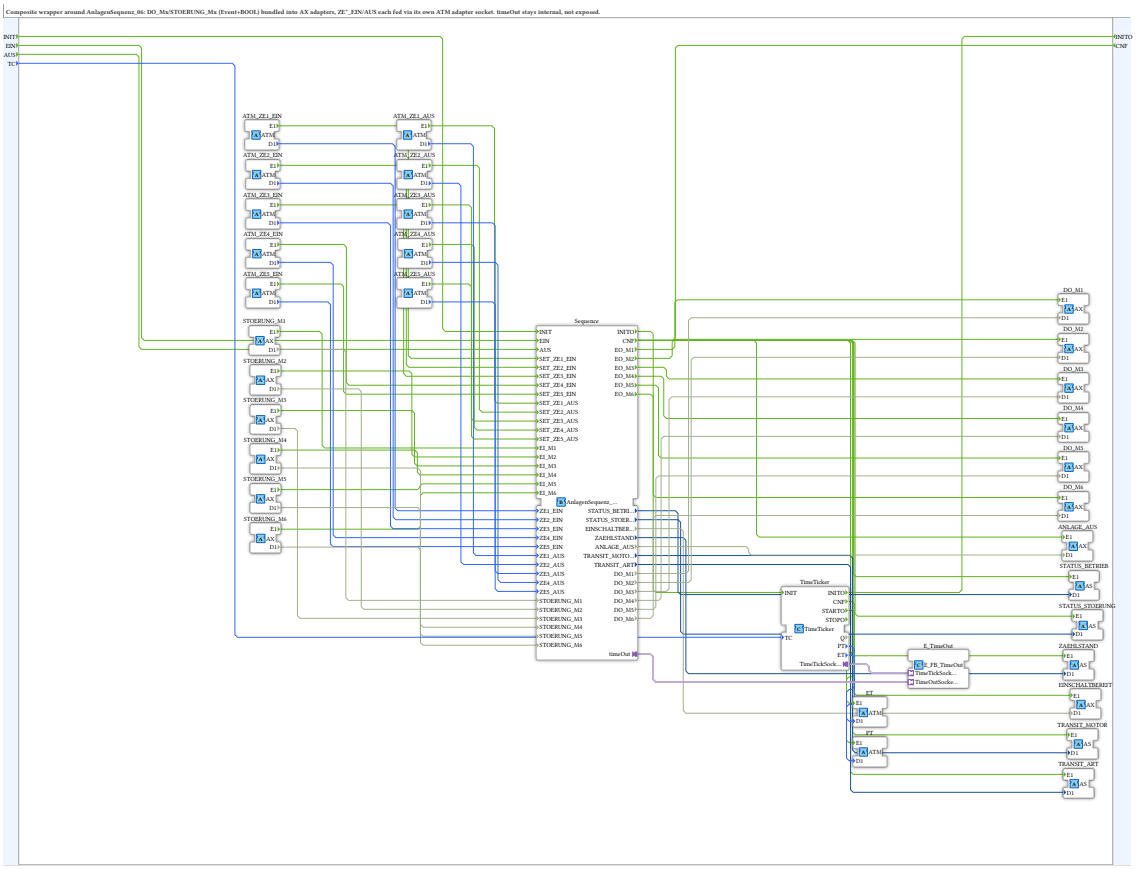
⚖️ Vergleich mit ähnlichen Bausteinen

- **sequence_T_04** / **sequence_T_08**: Generische, lineare Zeitsequenzer mit 4 bzw. 8 Schritten und einem einzigen Reset-Pfad. `AnlagenSequenz_06` verwendet denselben `ATimeOut`-Adapter-Mechanismus, ist aber kein generischer Baustein — die Ring-Topologie (zwei gekoppelte lineare Ketten), das feste 6-Motoren-Muster je Schritt und die direkte Ein-Schritt-Störungs-Kaskade sind fest einprogrammiert, nicht konfigurierbar.
- **Einfache Timer-Ketten (TON-Verkettung)**: Müssten Zustandslogik, Motor-Muster je Schritt und Störungsbehandlung von Hand nachbilden. `AnlagenSequenz_06` kapselt das vollständig inklusive Interlock-Logik.
- **Zähler-/ereignisbasierte Sequenzer**: Schalten auf externe Trigger statt auf Zeit weiter. `AnlagenSequenz_06` ist speziell für den Fall gedacht, dass die Verweildauer je Schritt fest vorgegeben ist (Anlaufzeit der Motoren), mit Störungen als einzigem asynchronen Unterbrechungspfad.

Fazit

`AnlagenSequenz_06` bündelt eine komplette, sicherheitsrelevante Start-/Stopp-Logik für sechs in Reihe arbeitende Motoren in einem einzigen Baustein: geordneter Vor- und Nachlauf über zwei gekoppelte lineare Ketten, ein verriegelter Störungsstatus mit explizitem Wiederanlauf-Interlock, und eine direkte, verzögerungsfreie Störungs-Kaskade aus jedem Betriebszustand heraus. Durch die klare Trennung von Ring-Topologie (ECC), Motor-Schaltlogik (Algorithmen) und Zeitsteuerung (`timeOut`-Adapter) bleibt der Baustein trotz seiner Komplexität wartungsfreundlich und reduziert den Programmieraufwand im übergeordneten Application-Netzwerk erheblich.

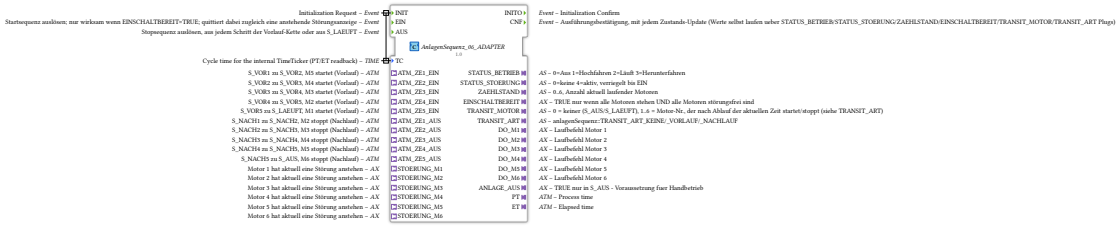
AnlagenSequenz_06_ADAPTER



• — ○ • — ○ • — ○ •

Einleitung

`AnlagenSequenz_06_ADAPTER` ist ein Composite-Wrapper um AnlagenSequenz_06: dieselbe Ring-Sequenzer-Logik für sechs Motoren, aber mit allen datentragenden Ein-/Ausgängen auf Adapterverbindungen umgestellt, damit der Baustein in einer Anwendung ausschließlich über Adapter (`AX`, `AS`, `ATM`) verdrahtet werden kann, ohne einzelne Event/BOOL/TIME-Leitungen ziehen zu müssen. Die Zeitsteuerung (`timeOut`) bleibt intern und wird nicht nach außen geführt.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: AnlagenSequenz_06_ADAPTER

- **Typ**: Composite (FBNetwork)
- **Verwendete interne FBs**:
  - **Sequence**: `logiBUS::utils::sequence::timed::AnlagenSequenz_06` — der eigentliche Ring-Sequenzer, siehe AnlagenSequenz_06 für die vollständige Zustandsübersicht.
  - **TimeTicker**: `adapter::events::TimeOut::TimeTicker` — erzeugt den zyklischen Zeittakt (`TC`, Default `T#200ms`) für die interne Zeitsteuerung.
  - **E_TimeOut**: `adapter::events::TimeOut::E_FB_TimeOut` — koppelt `Sequence.timeOut` an `TimeTicker`, damit die Vor-/Nachlauf-Zeiten ohne eigenen Timer-FB in der Anwendung ablaufen.
- **Funktionsweise**: Bündelt die sechs `DO_Mx`-Laufbefehle und sechs `STOERUNG_Mx`-Störungseingänge sowie alle Statuswerte (`STATUS_BETRIEB`, `STATUS_STOERUNG`, `ZAEHLSTAND`, `EINSCHALTBEREIT`, `TRANSIT_MOTOR`, `TRANSIT_ART`, `ANLAGE_AUS`) in `AX`/`AS`-Adapter, und jede der zehn Verweildauern (`ZE1..5_EIN`/`_AUS`) über einen eigenen `ATM`-Adapter-Socket, statt sie als einzelne `TIME`-Dateneingänge freizulegen. `PT`/`ET` (Process-/Elapsed-Time) werden als `ATM`-Plugs für ein Readback der internen Zeitsteuerung nach außen geführt.
- **Verweildauer-Sockets (`ATM`)**: Die zehn Sockets `ATM_ZE1_EIN`..`ATM_ZE5_EIN` und `ATM_ZE1_AUS`..`ATM_ZE5_AUS` (jeweils Typ `adapter::types::unidirectional::ATM`) liefern die Verweildauern der Übergänge:
  - `ATM_ZE1_EIN`: S_VOR1 zu S_VOR2, M5 startet (Vorlauf)
  - `ATM_ZE2_EIN`: S_VOR2 zu S_VOR3, M4 startet (Vorlauf)
  - `ATM_ZE3_EIN`: S_VOR3 zu S_VOR4, M3 startet (Vorlauf)
  - `ATM_ZE4_EIN`: S_VOR4 zu S_VOR5, M2 startet (Vorlauf)
  - `ATM_ZE5_EIN`: S_VOR5 zu S_LAEUFT, M1 startet (Vorlauf)
  - `ATM_ZE1_AUS`: S_NACH1 zu S_NACH2, M2 stoppt (Nachlauf)
  - `ATM_ZE2_AUS`: S_NACH2 zu S_NACH3, M3 stoppt (Nachlauf)
  - `ATM_ZE3_AUS`: S_NACH3 zu S_NACH4, M4 stoppt (Nachlauf)
  - `ATM_ZE4_AUS`: S_NACH4 zu S_NACH5, M5 stoppt (Nachlauf)
  - `ATM_ZE5_AUS`: S_NACH5 zu S_AUS, M6 stoppt (Nachlauf)

Programmablauf und Verbindungen

1. `EIN`/`AUS` werden unverändert an `Sequence.EIN`/`Sequence.AUS` durchgereicht.
2. Jeder `ATM_ZEk_EIN`/`ATM_ZEk_AUS`-Adapter liefert seine Zeit (`.D1`) an `Sequence.ZEk_EIN`/`Sequence.ZEk_AUS` und löst gleichzeitig (`.E1`) das entsprechende `Sequence.SET_ZEk_EIN`/`SET_ZEk_AUS`-Setzereignis aus.
3. Jeder `STOERUNG_Mx`-Adapter liefert sein Störungssignal (`.D1`) an `Sequence.STOERUNG_Mx` und löst (`.E1`) `Sequence.EI_Mx` aus.
4. `Sequence.CNF` löst gleichzeitig alle Status-Adapter aus (`STATUS_BETRIEB`, `STATUS_STOERUNG`, `ZAEHLSTAND`, `EINSCHALTBEREIT`, `TRANSIT_MOTOR`, `TRANSIT_ART`, `ANLAGE_AUS`); die zugehörigen Datenwerte laufen parallel über deren `.D1`.
5. Jedes `Sequence.EO_Mx` löst `DO_Mx.E1`, mit dem Laufbefehl über `DO_Mx.D1`.
6. `Sequence.INITO` startet den `TimeTicker`; dessen `CNF`/`STARTO` speisen `ET`/`PT` für ein reines Zeit-Readback nach außen.

7. `Sequence.timeOut` (Adapter-Socket) ist intern mit `E_TimeOut.TimeOutSocket` verbunden, `TimeTicker.TimeTickSocket` mit `E_TimeOut.TimeTickSocket` — die komplette Zeitsteuerung bleibt innerhalb des Composite gekapselt.

Anwendungsszenarien

- Anwendungen, die konsequent auf Adapterverbindungen setzen (z. B. für VT-Anbindung oder OPC-UA-Exposition über bereits vorhandene Adapter-Konvertierungsketten) und keine einzelnen Event-/Daten-Leitungen zur Motorsequenz ziehen wollen.
- Wiederverwendung derselben Ring-Sequenzer-Logik in mehreren Anlagenteilen, wobei die Verdrahtung über standardisierte Adaptertypen (`AX`, `AS`, `ATM`) einheitlich bleibt.

⚖️ Vergleich mit ähnlichen Bausteinen

- **AnlagenSequenz_06**: Die zugrundeliegende Basic-FB-Logik mit klassischen Event-/Daten-Anschlüssen. `AnlagenSequenz_06_ADAPTER` verändert nur die Schnittstelle, nicht das Zustandsverhalten.
- **sequence_T_08_ADAPTER**: Gleiches Adapter-Wrapper-Muster, angewendet auf den generischen 8-Schritt-Sequenzer statt auf die feste 6-Motoren-Ringtopologie.

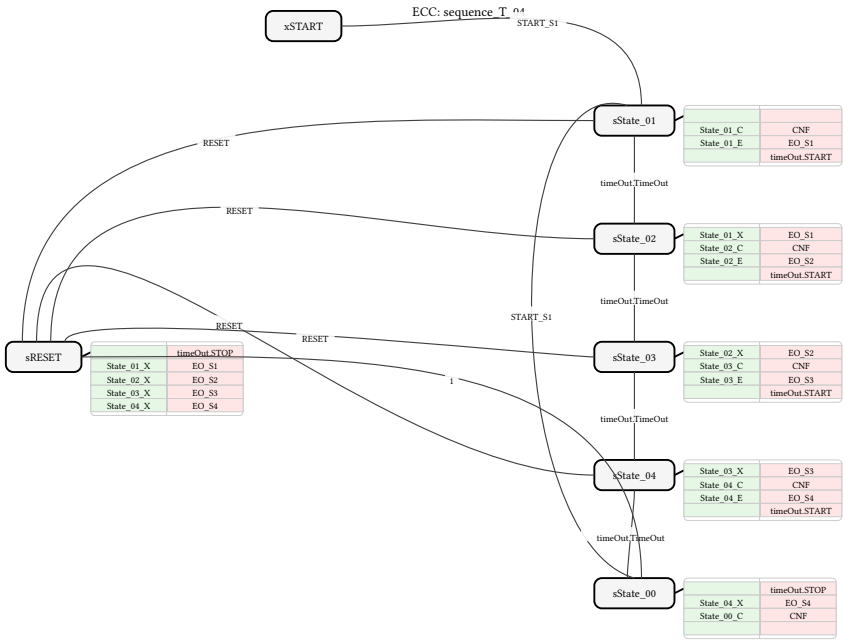
Fazit

`AnlagenSequenz_06_ADAPTER` macht die komplette Ring-Sequenzer-Logik für sechs Motoren über reine Adapterverbindungen nutzbar, ohne die zugrundeliegende Zustandslogik zu verändern — die Zeitsteuerung bleibt vollständig intern gekapselt.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

sequence_T_04

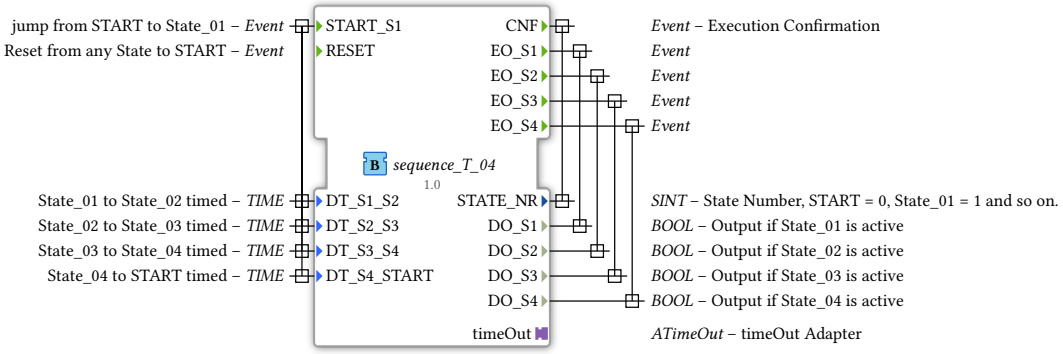


• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `sequence_T_04` ist ein zeitgesteuerter Sequenzer mit vier definierten Zuständen (State_01 bis State_04) und einem Startzustand (START). Er ermöglicht die zyklische oder einmalige Abarbeitung einer festen Reihenfolge, wobei die Verweildauer in jedem Zustand über konfigurierbare Zeitwerte gesteuert wird. Der Übergang zwischen den Zuständen erfolgt automatisch nach Ablauf der

eingestellten Zeit. Der Baustein eignet sich ideal für zeitlich getaktete Ablaufsteuerungen, wie sie z.B. in Förderanlagen, Verpackungsmaschinen oder Waschprozessen vorkommen.



Schnittstellenstruktur

Ereignis-Eingänge

- **`START_S1`**: Startet die Sequenz. Ein Ereignis an diesem Eingang bewirkt den Übergang vom Zustand START oder sState_00 in den ersten aktiven Zustand State_01. Es werden die Zeitdaten `DT_S1_S2`, `DT_S2_S3`, `DT_S3_S4` und `DT_S4_START` mitgelesen.
- **`RESET`**: Setzt die Sequenz von jedem beliebigen Zustand sofort in den Ruhezustand (sState_00) zurück. Alle Ausgänge (`DO_S1` bis `DO_S4`) werden dabei deaktiviert.

Ereignis-Ausgänge

- **`CNF`**: Bestätigungsereignis (Execution Confirmation). Wird bei jedem Zustandswechsel ausgelöst und liefert die aktuelle Zustandsnummer `STATE_NR`.
- **`EO_S1`**: Wird bei Eintritt in den Zustand State_01 ausgelöst und liefert den zugehörigen Datenausgang `DO_S1`.
- **`EO_S2`**: Wird bei Eintritt in den Zustand State_02 ausgelöst und liefert den zugehörigen Datenausgang `DO_S2`.
- **`EO_S3`**: Wird bei Eintritt in den Zustand State_03 ausgelöst und liefert den zugehörigen Datenausgang `DO_S3`.
- **`EO_S4`**: Wird bei Eintritt in den Zustand State_04 ausgelöst und liefert den zugehörigen Datenausgang `DO_S4`.

Daten-Eingänge

- **`DT_S1_S2`** (TIME): Verweildauer im Zustand State_01, bevor der automatische Übergang zu State_02 erfolgt. Initialwert: `NO_TIME`.
- **`DT_S2_S3`** (TIME): Verweildauer im Zustand State_02, bevor der automatische Übergang zu State_03 erfolgt. Initialwert: `NO_TIME`.
- **`DT_S3_S4`** (TIME): Verweildauer im Zustand State_03, bevor der automatische Übergang zu State_04 erfolgt. Initialwert: `NO_TIME`.
- **`DT_S4_START`** (TIME): Verweildauer im Zustand State_04, bevor der automatische Übergang zum Ruhezustand sState_00 erfolgt. Initialwert: `NO_TIME`.

Daten-Ausgänge

- **`STATE_NR`** (SINT): Gibt die Nummer des aktuell aktiven Zustands aus. START = 0, State_01 = 1, State_02 = 2, State_03 = 3, State_04 = 4.
- **`DO_S1`** (BOOL): Logischer Ausgang, der TRUE ist, solange sich der FB im Zustand State_01 befindet.
- **`DO_S2`** (BOOL): Logischer Ausgang, der TRUE ist, solange sich der FB im Zustand State_02 befindet.
- **`DO_S3`** (BOOL): Logischer Ausgang, der TRUE ist, solange sich der FB im Zustand State_03 befindet.
- **`DO_S4`** (BOOL): Logischer Ausgang, der TRUE ist, solange sich der FB im Zustand State_04 befindet.

Adapter

- **`timeOut`** (Plug, Typ: `iec61499::events::ATimeOut`): Ein standardisierter TimeOut-Adapter, der für die zeitgesteuerten Zustandsübergänge verwendet wird. Der FB startet (`timeOut.START`) den Timer beim Eintritt in einen aktiven Zustand und reagiert auf dessen `TimeOut`-Ereignis.

Funktionsweise

Der FB arbeitet als Basic Function Block (BFB) mit einer definierten Execution Control Chart (ECC). Die Sequenz durchläuft die Zustände in der festen Reihenfolge: START -> State_01 -> State_02 -> State_03 -> State_04 -> sState_00.

1. **Start**: Ein `START_S1`-Ereignis (aus den Zuständen `xSTART` oder `sState_00`) aktiviert State_01.
2. **Zustandsaktivierung**: Beim Eintritt in einen Zustand (State_01-04) werden folgende Aktionen ausgeführt:
   - Der entsprechende Datenausgang (`DO_Sx`) wird auf TRUE gesetzt (Entry-Algorithmus `State_x_E`).
   - Das zugehörige Ereignis (`EO_Sx`) wird ausgelöst.
   - Die Zustandsnummer `STATE_NR` wird aktualisiert (Confirmation-Algorithmus `State_x_C`).
   - Die für den nächsten Übergang konfigurierte Zeit (`DT_...`) wird an den `timeOut`-Adapter übergeben und der Timer gestartet.
   - Das allgemeine Bestätigungsereignis `CNF` wird ausgelöst.
3. **Zeitgesteuerter Übergang**: Nach Ablauf der im Timer eingestellten Zeit löst der Adapter das `TimeOut`-Ereignis aus. Dies ist die Bedingung für den Übergang zum nächsten Zustand in der Sequenz.
4. **Zustandsverlassen**: Beim Verlassen eines Zustands wird der zugehörige Datenausgang (`DO_Sx`) durch den Exit-Algorithmus (`State_x_X`) auf FALSE zurückgesetzt.
5. **Zyklusende und Reset**: Nach State_04 wechselt der FB in den Zustand `sState_00` (Ruhezustand). Von hier aus kann die Sequenz durch ein erneutes `START_S1`-Ereignis neu gestartet werden. Ein `RESET`-Ereignis von jedem Zustand aus führt sofort in den `sRESET`-Zustand, deaktiviert alle Ausgänge und geht dann in `sState_00` über.

Technische Besonderheiten

- **Initialwerte**: Die Zeitdaten-Eingänge sind standardmäßig mit der Konstante `NO_TIME` vorbelegt. Ein Wert von `NO_TIME` führt dazu, dass der FB nach dem Eintritt in den Zustand unendlich lange wartet, bis ein `RESET` erfolgt.
- **Zustand sState_00**: Dieser Zustand ist der stabile Ruhezustand nach Abschluss eines Zyklus. Im Gegensatz zum initialen `xSTART` ist er Teil der ECC und kann Algorithmen ausführen (hier die Zustandsnummer 0 setzen).
- **Adapter-Nutzung**: Die Zeitsteuerung ist komplett in den standardisierten `ATimeOut`-Adapter ausgelagert, was Wiederverwendbarkeit und Klarheit erhöht.

Zustandsübersicht

Zustandsname	Beschreibung	Aktive Ausgänge	Übergangsbedingung zum nächsten Zustand
xSTART	Initialer Idle-Zustand.	Keine	`START_S1`
sState_01	Erster aktiver Schritt.	`DO_S1=1`, `STATE_NR=1`	`timeOut.TimeOut` (nach DT_S1_S2)
sState_02	Zweiter aktiver Schritt.	`DO_S2=1`, `STATE_NR=2`	`timeOut.TimeOut` (nach DT_S2_S3)
sState_03	Dritter aktiver Schritt.	`DO_S3=1`, `STATE_NR=3`	`timeOut.TimeOut` (nach DT_S3_S4)
sState_04	Vierter aktiver Schritt.	`DO_S4=1`, `STATE_NR=4`	`timeOut.TimeOut` (nach DT_S4_START)
sState_00	Ruhezustand nach Sequenzende.	`STATE_NR=0`	`START_S1` (für neuen Zyklus)
sRESET	Zwischenzustand für Reset-Vorgang.	Keine	Immer (`Condition=1`)

Globale Übergangsbedingung: Von den Zuständen sState_01 bis sState_04 führt ein `RESET`-Ereignis immer in den sRESET-Zustand.

Anwendungsszenarien

- **Steuerung von Zyklusmaschinen**: Automatische Abfolge von Bearbeitungsschritten wie Bohren, Fräsen, Entgraten mit einstellbaren Bearbeitungszeiten.
- **Verpackungsanlagen**: Steuerung des zeitlichen Ablaufs: Produkt zuführen -> Verpackung schließen -> Etikett aufbringen -> Palettieren.
- **Reinigungsprozesse**: Steuerung einer Sprühkabine: Vorspülen (State_01) -> Hauptreinigung (State_02) -> Nachspülen (State_03) -> Trocknen (State_04).
- **Teststände**: Automatisierte Prüfsequenz, bei der jeder Testschritt eine feste Dauer hat.

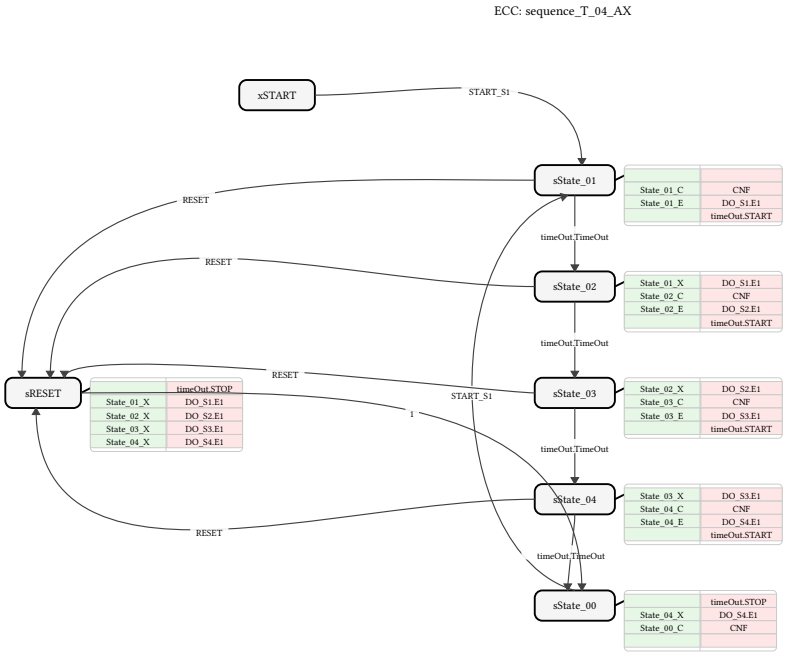
⚖️ Vergleich mit ähnlichen Bausteinen

- **Einfache Timer (TON)**: Einzelne Timerbausteine müssten für eine Sequenz verkettet und die Logik für die Zustandsübergänge und Ausgänge separat programmiert werden. `sequence_T_04` kapselt diese komplette Logik.
- **Zähler-basierte Sequenzer**: Sequenzer, die mit Ereignissen (nicht Zeit) weiterschalten, bieten eine andere Art der Steuerung. `sequence_T_04` ist speziell für zeitkritische Abläufe ohne externe Trigger gedacht.
- **SPS-spezifische Ablaufsprachen (S7-GRAFCET)**: Dieser FB implementiert ein ähnliches Prinzip wie eine Grafcet-Step-Kette, jedoch innerhalb des portablen IEC 61499-Standards.

Fazit

Der `sequence_T_04` ist ein robustes und einfach zu konfigurierendes Werkzeug für zeitgesteuerte Ablaufsteuerungen mit bis zu vier Schritten. Durch die klare Trennung von Zustandslogik (ECC) und Zeitsteuerung (Adapter) sowie die vollständige Kapselung des Verhaltens ist er wartungsfreundlich und reduziert den Programmieraufwand im übergeordneten Application-Netzwerk erheblich. Die Möglichkeit eines sofortigen Resets von jedem Zustand aus gewährleistet zudem eine hohe Betriebssicherheit.

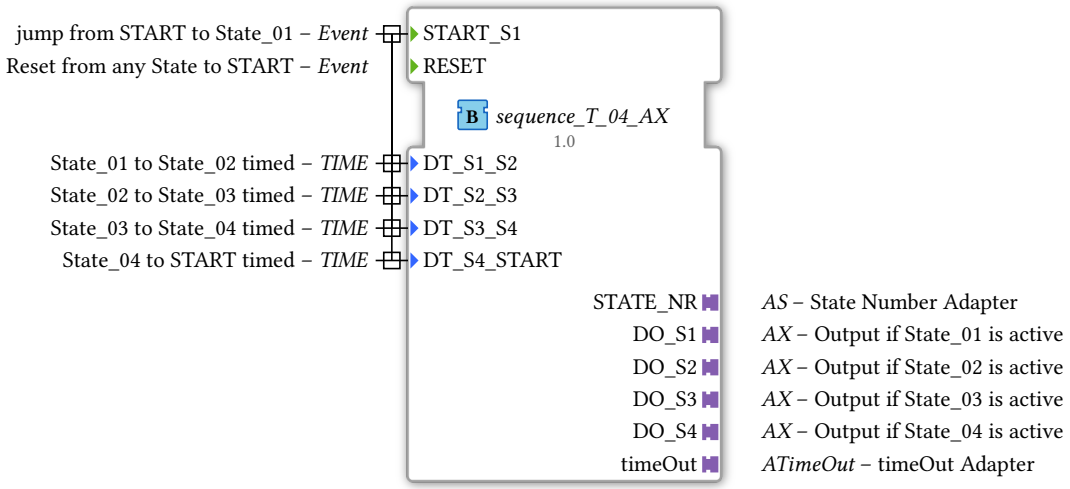
sequence_T_04_AX



• – ○ • – ○ • – ○ •

Einleitung

`sequence_T_04_AX` ist eine Variante des `sequence_T_04`, die zusätzlich Adapter (`AX`) für die Ausgänge verwendet. Er steuert eine rein zeitgesteuerte Sequenz mit 4 Ausgabezuständen.



Schnittstellenstruktur

Ereignis-Eingänge

- **START_S1**: Startet die Sequenz bei State_01.
- **RESET**: Setzt die Sequenz zurück.

Ereignis-Ausgänge

- **CNF**: Bestätigung der Ausführung.

Daten-Eingänge

- **DT_S1_S2**: Zeit für Übergang State_01 -> State_02.
- **DT_S2_S3**: Zeit für Übergang State_02 -> State_03.
- **DT_S3_S4**: Zeit für Übergang State_03 -> State_04.
- **DT_S4_START**: Zeit für Übergang State_04 -> START.

Daten-Ausgänge

- **STATE_NR** (SINT): Aktuelle Zustandsnummer.

Adapter

- **DO_S1** (adapter::types::unidirectional::AX): Ausgangsadapter für State_01.
- **DO_S2** (adapter::types::unidirectional::AX): Ausgangsadapter für State_02.
- **DO_S3** (adapter::types::unidirectional::AX): Ausgangsadapter für State_03.
- **DO_S4** (adapter::types::unidirectional::AX): Ausgangsadapter für State_04.
- **timeOut** (iec61499::events::ATimeOut): Timer-Adapter.

Funktionsweise

Entspricht `sequence_T_04`, verwendet jedoch Adapter für die Ausgänge.

Technische Besonderheiten

- Verwendung von `adapter::types::unidirectional::AX`.

Zustandsübersicht

Siehe `sequence_T_04`.

Anwendungsszenarien

Für zeitgesteuerte 4-stufige Sequenzen mit Adapteranbindung.

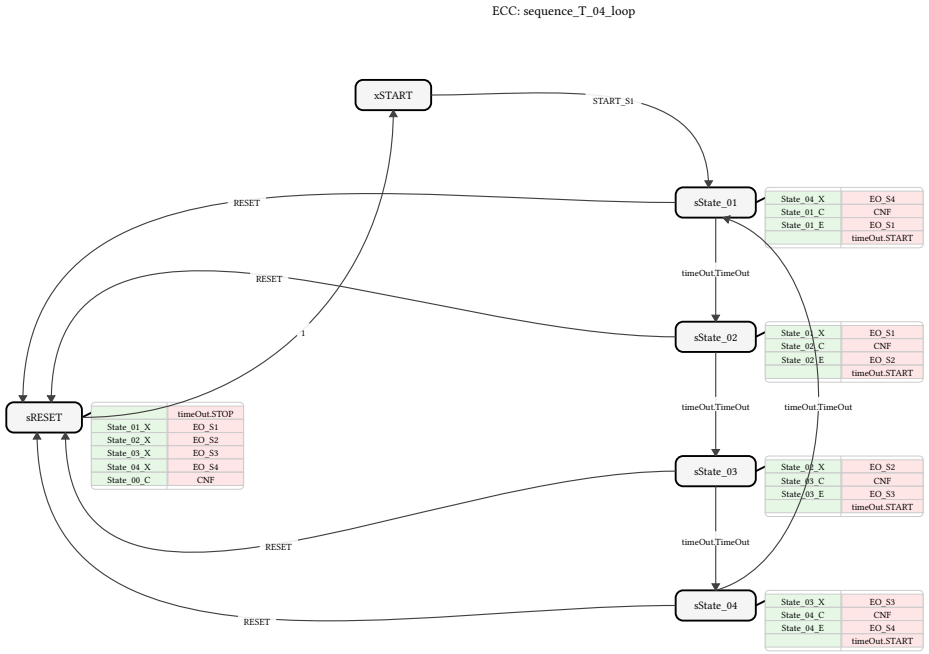
⚖️ Vergleich mit ähnlichen Bausteinen

- **sequence_T_04**: Standardvariante ohne Adapter.

Fazit

Adapter-Variante des 4-Schritt-Zeit-Sequenzers.

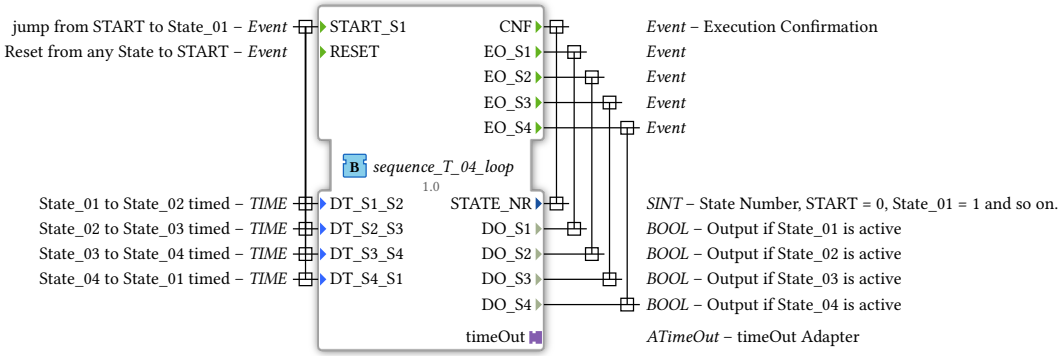
sequence_T_04_loop



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `sequence_T_04_loop` ist ein zeitgesteuerter Sequenzer mit vier Ausgängen, der in einer Schleife arbeitet. Er durchläuft nacheinander vier Zustände (State_01 bis State_04). Der Übergang von einem Zustand zum nächsten erfolgt automatisch nach Ablauf einer einstellbaren Zeitverzögerung. Der Baustein kann von jedem Zustand aus zurückgesetzt werden und startet dann die Sequenz erneut.



Schnittstellenstruktur

Ereignis-Eingänge

- **`START_S1`**: Startet die Sequenz. Der Übergang erfolgt vom initialen `START`-Zustand in `State_01`. Das Ereignis ist mit den vier Zeitdaten-Eingängen verknüpft.
- **`RESET`**: Setzt die Sequenz von jedem beliebigen aktiven Zustand zurück in den initialen `START`-Zustand.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung (Confirmation). Wird bei jedem Zustandswechsel ausgelöst und liefert die aktuelle Zustandsnummer.
- **EO_S1**: Wird beim Eintritt in State_01 ausgelöst und liefert den zugehörigen Datenausgang DO_S1.
- **EO_S2**: Wird beim Eintritt in State_02 ausgelöst und liefert den zugehörigen Datenausgang DO_S2.
- **EO_S3**: Wird beim Eintritt in State_03 ausgelöst und liefert den zugehörigen Datenausgang DO_S3.
- **EO_S4**: Wird beim Eintritt in State_04 ausgelöst und liefert den zugehörigen Datenausgang DO_S4.

Daten-Eingänge

- **DT_S1_S2** (TIME): Zeitverzögerung für den Übergang von State_01 zu State_02. Initialwert: NO_TIME.
- **DT_S2_S3** (TIME): Zeitverzögerung für den Übergang von State_02 zu State_03. Initialwert: NO_TIME.
- **DT_S3_S4** (TIME): Zeitverzögerung für den Übergang von State_03 zu State_04. Initialwert: NO_TIME.
- **DT_S4_S1** (TIME): Zeitverzögerung für den Übergang von State_04 zurück zu State_01 (Schleife). Initialwert: NO_TIME.

Daten-Ausgänge

- **STATE_NR** (SINT): Aktuelle Zustandsnummer. 0 = START, 1 = State_01, 2 = State_02, 3 = State_03, 4 = State_04.
- **DO_S1** (BOOL): Ist TRUE, wenn State_01 aktiv ist.
- **DO_S2** (BOOL): Ist TRUE, wenn State_02 aktiv ist.
- **DO_S3** (BOOL): Ist TRUE, wenn State_03 aktiv ist.
- **DO_S4** (BOOL): Ist TRUE, wenn State_04 aktiv ist.

Adapter

- **timeOut** (Plug, Typ: iec61499::events::ATimeOut): Ein Zeitgeber-Adapter, der für die zeitgesteuerten Zustandsübergänge verwendet wird. Der FB startet den Timer beim Eintritt in einen Zustand und reagiert auf dessen TimeOut-Ereignis.

Funktionsweise

Der FB ist als BasicFB mit einer ECC (Execution Control Chart) implementiert. Die Sequenz beginnt im initialen Zustand xSTART. Ein START_S1-Ereignis führt in den ersten aktiven Zustand sState_01.

In jedem aktiven Zustand (sState_01 bis sState_04) werden nacheinander folgende Aktionen ausgeführt:

1. **Exit-Step des vorherigen Zustands**: Der zugehörige Datenausgang (DO_Sx) wird auf FALSE gesetzt (außer beim ersten Eintritt von xSTART).
2. **Confirmation-Step**: Die Zustandsnummer STATE_NR wird aktualisiert und die Verzögerungszeit für den *nächsten* Übergang wird an den timeOut-Adapter übergeben (timeOut.DT).
3. **Entry-Step des neuen Zustands**: Der zugehörige Datenausgang (DO_Sx) wird auf TRUE gesetzt und das entsprechende Ereignis (EO_Sx) wird ausgelöst.
4. **Timer-Start**: Der timeOut-Adapter wird gestartet (timeOut.START).

Der Übergang zum nächsten Zustand erfolgt ausschließlich durch das TimeOut-Ereignis des Adapters. Nach State_04 springt die Sequenz gemäß Schleifenlogik zurück zu State_01.

Ein RESET-Ereignis von jedem Zustand aus führt in den sRESET-Zustand. Dort werden alle Ausgänge (DO_S1 bis DO_S4) deaktiviert, der Timer gestoppt, die Zustandsnummer auf 0 (START) gesetzt und eine Bestätigung (CNF) ausgegeben. Anschließend kehrt der FB automatisch in den xSTART-Zustand zurück.

Technische Besonderheiten

- **Zeitsteuerung**: Die Übergänge sind rein zeitgesteuert. Es gibt keine ereignisgesteuerten Übergänge zwischen den Hauptzuständen.
- **Initialwerte**: Die Zeitdaten-Eingänge sind standardmäßig mit NO_TIME vorbelegt. Dies muss für einen funktionierenden Ablauf angepasst werden.
- **Konstanten**: Der FB verwendet Konstanten aus der Bibliothek logiBUS::utils::sequence::const::sequence (z.B. für Zustandsnummern) und logiBUS::utils::sequence::const::sequence::NO_TIME.

Zustandsübersicht

1. **xSTART**: Initialer, inaktiver Zustand. Wartet auf START_S1.
2. **sState_01**: Erster aktiver Zustand. DO_S1 = TRUE. Timer für Übergang zu State_02 läuft.

3. **`sState_02`**: Zweiter aktiver Zustand. `DO_S2 = TRUE`. Timer für Übergang zu `State_03` läuft.
4. **`sState_03`**: Dritter aktiver Zustand. `DO_S3 = TRUE`. Timer für Übergang zu `State_04` läuft.
5. **`sState_04`**: Vierter aktiver Zustand. `DO_S4 = TRUE`. Timer für Übergang (zurück) zu `State_01` läuft.
6. **`sRESET`**: Reset-Zustand. Deaktiviert alle Ausgänge, stoppt den Timer und bestätigt den Reset.

Anwendungsszenarien

- Steuerung zyklischer Prozesse mit festen Zeitschritten, z.B. in Verpackungsmaschinen, Waschanlagen oder automatisierten Montagelinien.
- Ansteuerung von Aktoren in einer festen, zeitlich getakteten Reihenfolge.
- Als zentraler Taktgeber für übergeordnete Steuerungsabläufe.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Einfache Timer (TON)**: Einzelne Timer bieten keine integrierte Sequenzlogik. `sequence_T_04_loop` kapselt die komplette Zustandsmaschine mit vier Schritten.
- **Ereignisgesteuerte Sequenzer**: Im Gegensatz zu ereignisgesteuerten Sequenzern (z.B. `sequence_E_04_loop`) erfolgen die Übergänge hier ausschließlich zeitbasiert, nicht durch externe Ereignisse.
- **SPS-Zyklus**: Die zeitgesteuerte Ablaufkontrolle ist präziser und unabhängig vom SPS-Zyklus, da sie auf dem `ATimeOut`-Adapter basiert.

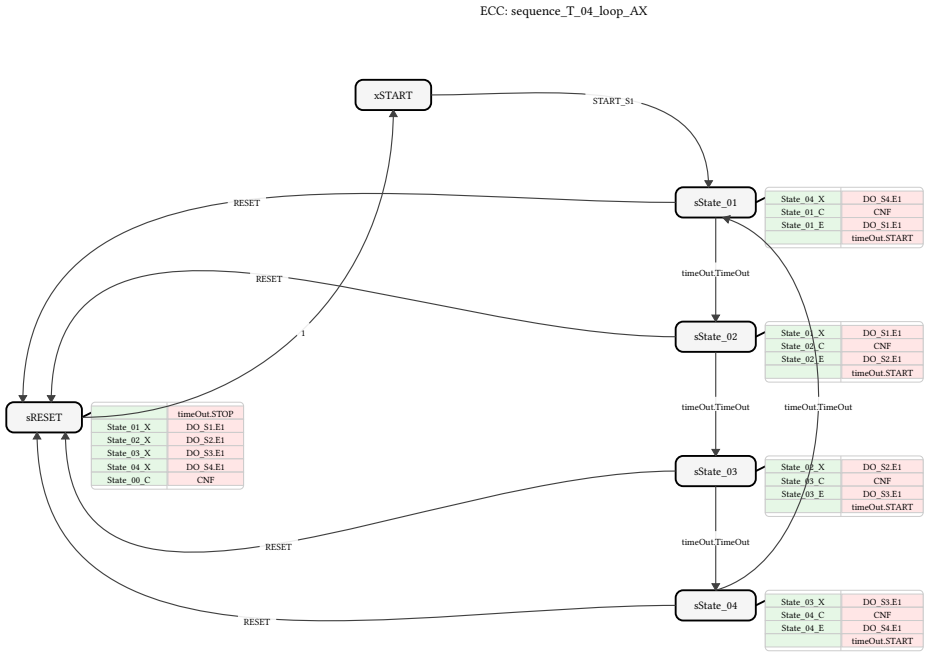
🛠️ Zugehörige Übungen

- Uebung_035a

Fazit

Der `sequence_T_04_loop` ist ein robuster und einfach zu konfigurierender Funktionsblock für zeitgesteuerte Ablaufsteuerungen mit vier Schritten. Durch die klare Trennung von Zustandslogik und Zeitparametern sowie die integrierte Reset-Funktionalität eignet er sich gut für standardisierte zyklische Prozesse in der Automatisierungstechnik. Die Verwendung eines Standard-Adapters für die Zeitmessung gewährleistet Portabilität und Zuverlässigkeit.

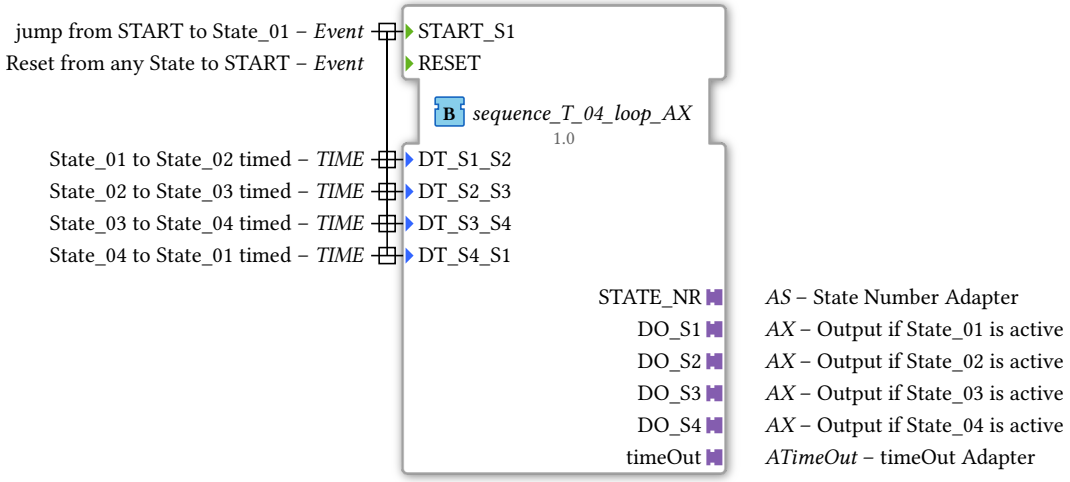
sequence_T_04_loop_AX



• – ○ • – ○ • – ○ •

Einleitung

`sequence_T_04_loop_AX` ist eine Variante des `sequence_T_04_loop`, die zusätzlich Adapter (`AX`) für die Ausgänge verwendet. Er steuert eine rein zeitgesteuerte, zyklische Sequenz mit 4 Ausgabezuständen.



Schnittstellenstruktur

Ereignis-Eingänge

- **START_S1**: Startet die Sequenz bei State_01.
- **RESET**: Setzt die Sequenz zurück.

Ereignis-Ausgänge

- **CNF**: Bestätigung der Ausführung.

Daten-Eingänge

- **DT_S1_S2**: Zeit für Übergang State_01 -> State_02.
- **DT_S2_S3**: Zeit für Übergang State_02 -> State_03.
- **DT_S3_S4**: Zeit für Übergang State_03 -> State_04.
- **DT_S4_S1**: Zeit für Übergang State_04 -> State_01.

Daten-Ausgänge

- **STATE_NR** (SINT): Aktuelle Zustandsnummer.

Adapter

- **DO_S1** (adapter::types::unidirectional::AX): Ausgangsadapter für State_01.
- **DO_S2** (adapter::types::unidirectional::AX): Ausgangsadapter für State_02.
- **DO_S3** (adapter::types::unidirectional::AX): Ausgangsadapter für State_03.
- **DO_S4** (adapter::types::unidirectional::AX): Ausgangsadapter für State_04.
- **timeOut** (iec61499::events::ATimeOut): Timer-Adapter.

Funktionsweise

Entspricht `sequence_T_04_loop`, verwendet jedoch Adapter für die Ausgänge.

Technische Besonderheiten

- Verwendung von `adapter::types::unidirectional::AX`.

Zustandsübersicht

Siehe `sequence_T_04_loop`.

Anwendungsszenarien

Für zeitgesteuerte zyklische 4-stufige Sequenzen mit Adapteranbindung.

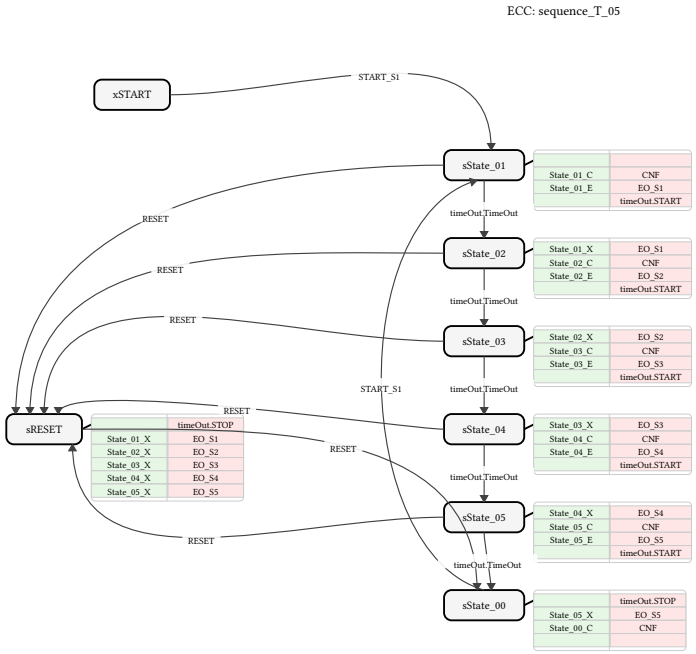
⚖️ Vergleich mit ähnlichen Bausteinen

- **sequence_T_04_loop**: Standardvariante ohne Adapter.

Fazit

Adapter-Variante des 4-Schritt-Loop-Zeit-Sequenzers.

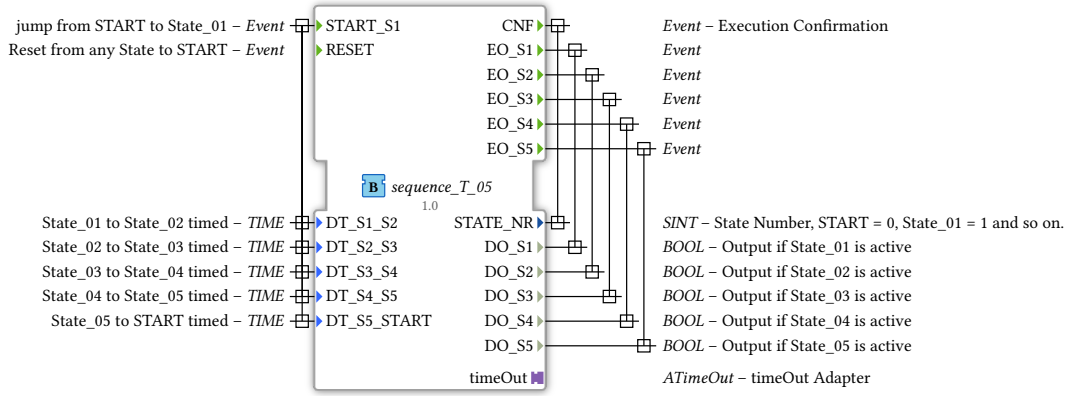
sequence_T_05



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock `sequence_T_05` ist ein zeitgesteuerter Sequenzer mit fünf Ausgangszuständen. Er durchläuft eine feste Abfolge von Zuständen (State_01 bis State_05), wobei der Übergang zwischen den einzelnen Zuständen durch einstellbare Zeitverzögerungen gesteuert wird. Der Block ist für Anwendungen konzipiert, in denen Prozessschritte oder Aktionen nacheinander und für eine definierte Dauer aktiviert werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **START_S1**: Startet die Sequenz. Der Block wechselt vom Startzustand (START oder State_00) in den ersten aktiven Zustand State_01. Dieses Ereignis ist mit den fünf Zeitdaten-Eingängen verknüpft.
- **RESET**: Setzt die Sequenz von jedem beliebigen Zustand zurück in den inaktiven Startzustand (State_00). Alle Ausgänge werden deaktiviert.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung (Confirmation). Wird bei jedem Zustandswechsel ausgelöst und liefert die neue Zustandsnummer (STATE_NR).
- **EO_S1**: Wird beim Eintritt in den Zustand State_01 ausgelöst und liefert den zugehörigen Datenausgang DO_S1.
- **EO_S2**: Wird beim Eintritt in den Zustand State_02 ausgelöst und liefert den zugehörigen Datenausgang DO_S2.
- **EO_S3**: Wird beim Eintritt in den Zustand State_03 ausgelöst und liefert den zugehörigen Datenausgang DO_S3.
- **EO_S4**: Wird beim Eintritt in den Zustand State_04 ausgelöst und liefert den zugehörigen Datenausgang DO_S4.
- **EO_S5**: Wird beim Eintritt in den Zustand State_05 ausgelöst und liefert den zugehörigen Datenausgang DO_S5.

Daten-Eingänge

Alle Zeitdaten-Eingänge sind vom Typ TIME und haben den Initialwert NO_TIME. Sie definieren die Verweildauer im jeweiligen Zustand, bevor der automatische Übergang zum nächsten Zustand erfolgt.

- **DT_S1_S2**: Verweildauer in State_01 vor dem Übergang zu State_02.
- **DT_S2_S3**: Verweildauer in State_02 vor dem Übergang zu State_03.
- **DT_S3_S4**: Verweildauer in State_03 vor dem Übergang zu State_04.
- **DT_S4_S5**: Verweildauer in State_04 vor dem Übergang zu State_05.
- **DT_S5_START**: Verweildauer in State_05 vor dem Übergang zurück zum Startzustand State_00.

Daten-Ausgänge

- **STATE_NR** (SINT): Gibt die aktuelle Zustandsnummer aus. 0 = START/State_00, 1 = State_01, ..., 5 = State_05.
- **DO_S1** (BOOL): Ist TRUE, wenn Zustand State_01 aktiv ist.
- **DO_S2** (BOOL): Ist TRUE, wenn Zustand State_02 aktiv ist.
- **DO_S3** (BOOL): Ist TRUE, wenn Zustand State_03 aktiv ist.
- **DO_S4** (BOOL): Ist TRUE, wenn Zustand State_04 aktiv ist.
- **DO_S5** (BOOL): Ist TRUE, wenn Zustand State_05 aktiv ist.

Adapter

- **timeOut** (Plug, Typ: iec61499::events::ATimeOut): Ein Timeout-Adapter, der für die zeitgesteuerten Zustandsübergänge verwendet wird. Der Block startet (START) den Timer beim Eintritt in einen aktiven Zustand und stoppt (STOP) ihn beim Verlassen.

Funktionsweise

Der Block arbeitet als Basic-FB mit einem definierten ECC (Execution Control Chart). Die Sequenz wird durch das Ereignis START_S1 initiiert. Daraufhin durchläuft der Block nacheinander die Zustände State_01 bis State_05. In jedem aktiven Zustand wird der entsprechende Datenausgang (DO_Sx) auf TRUE gesetzt und ein Timer mit der für diesen Zustand konfigurierten Dauer (DT_Sx_Sy) gestartet. Sobald der Timer abläuft (timeOut.TimeOut), erfolgt der automatische Übergang zum nächsten Zustand. Beim Zustandswechsel wird der vorherige Ausgang deaktiviert und der neue aktiviert. Nach State_05 kehrt der Block in den inaktiven Zustand State_00 zurück. Ein RESET-Ereignis unterbricht die Sequenz sofort, deaktiviert alle Ausgänge und bringt den Block in den Zustand State_00.

Technische Besonderheiten

- **Zustandsbehandlung**: Jeder aktive Zustand (State_01 bis State_05) verfügt über separate Algorithmen für Eintritt (_E), Bestätigung (_C) und Austritt (_X). Dies ermöglicht eine klare Trennung der Logik.

- **Timer-Integration**: Die Zeitsteuerung ist vollständig in den Adapter `ATimeOut` ausgelagert, was die Wiederverwendbarkeit und Wartbarkeit erhöht.
- **Konstanten**: Der Block importiert Konstanten aus `logiBUS::utils::sequence::const::sequence` (für Zustandsnummern) und `::NO_TIME` für den Initialwert der Zeitdauern.
- **Initialzustand**: Der eigentliche inaktive Ruhezustand nach einem Reset oder nach Sequenzende ist `sState_00`. Der `xSTART` ist der initiale ECC-Zustand beim ersten Hochfahren.

Zustandsübersicht

1. **`xSTART`**: Initialer ECC-Zustand beim Systemstart.
2. **`sState_00`**: Inaktiver Ruhezustand. Alle Ausgänge sind `FALSE`. Von hier aus kann die Sequenz mit `START_S1` gestartet werden.
3. **`sState_01` bis `sState_05`**: Aktive Sequenzzustände. Der jeweilige Ausgang `DO_Sx` ist `TRUE`. Der Übergang zum nächsten Zustand erfolgt zeitgesteuert.
4. **`sRESET`**: Zwischenzustand, der bei einem `RESET`-Ereignis von jedem aktiven Zustand aus angesprungen wird. Deaktiviert alle Ausgänge und wechselt dann zu `sState_00`.

Anwendungsszenarien

- **Steuerung von Batch-Prozessen**: Nacheinander aktivierte Schritte wie Füllen, Erhitzen, Rühren, Kühlen und Entleeren mit einstellbaren Schrittzeiten.
- **Ablaufsteuerungen in Maschinen**: Zeitgesteuerte Abfolge von Zylinderbewegungen oder Werkzeugwechseln in einer automatisierten Anlage.
- **Testsequenzen**: Automatisierte Prüfabläufe, bei denen verschiedene Signale nacheinander für eine bestimmte Zeit angelegt und das Ergebnis ausgewertet werden muss.
- **Beleuchtungssteuerung**: Zeitgesteuerte Choreografien für Werbe- oder Dekorationsbeleuchtung.

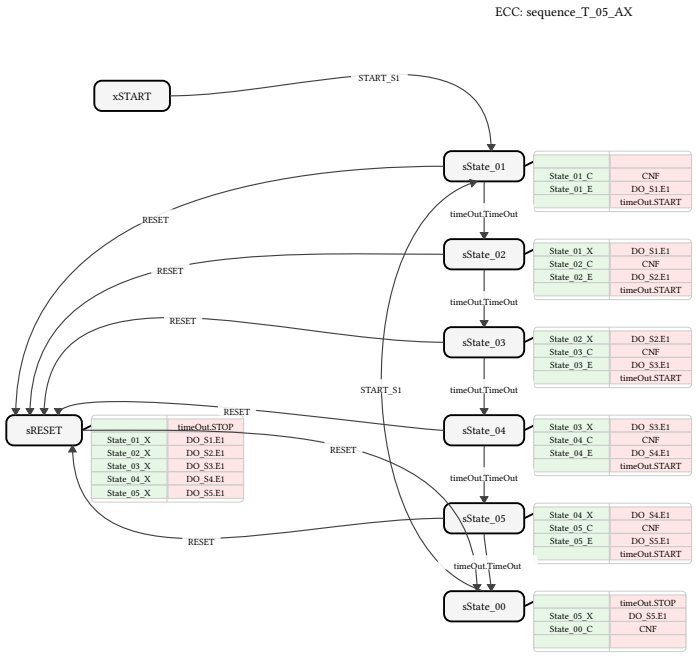
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem einfachen `E_DELAY`- oder `E_SR`-Baustein bietet `sequence_T_05` eine vordefinierte, mehrstufige Sequenzlogik in einem einzigen, konfigurierbaren Block. Gegenüber einer selbst programmierten Sequenz mit mehreren verbundenen Bausteinen vereinfacht er die Applikation erheblich und reduziert die Fehleranfälligkeit. Andere Sequenzer-Bausteine könnten eventuell auf Ereignisse (anstelle von Zeit) für den Übergang reagieren oder eine variable Anzahl von Schritten erlauben.

Fazit

Der `sequence_T_05` ist ein robustes und einfach zu konfigurierendes Werkzeug für zeitgesteuerte Ablaufsteuerungen mit fester Schrittanzahl. Durch die klare Trennung von Zustandslogik und Zeitsteuerung sowie die umfassenden Bestätigungs- und Rücksetzmechanismen eignet er sich ideal für zuverlässige Automatisierungsaufgaben in industriellen Umgebungen. Die Parametrierung der Schrittzeiten zur Laufzeit ermöglicht eine hohe Flexibilität.

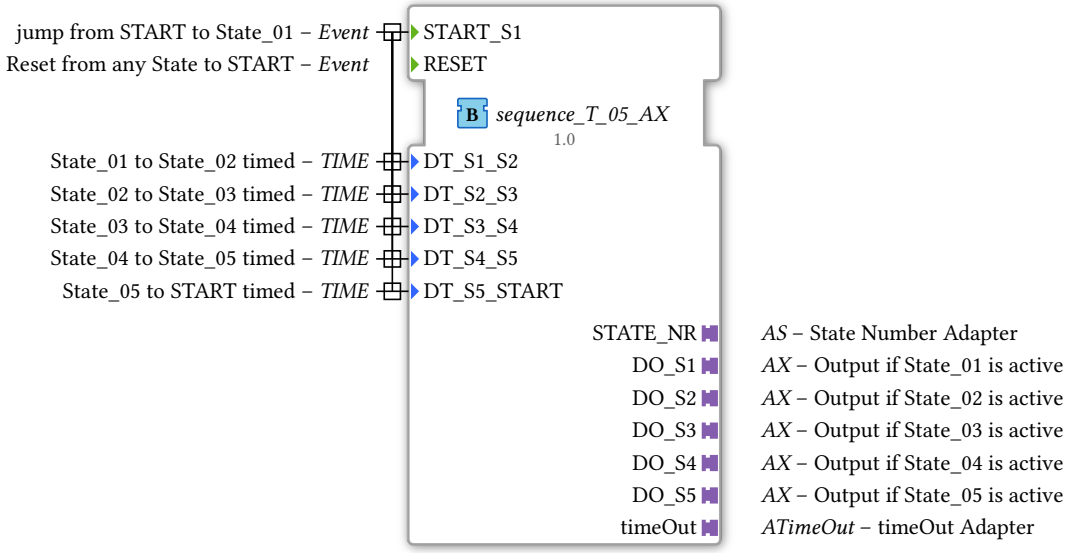
sequence_T_05_AX



• – ○ • – ○ • – ○ •

Einleitung

`sequence_T_05_AX` ist eine Variante des `sequence_T_05`, die zusätzlich Adapter (`AX`) für die Ausgänge verwendet. Er steuert eine rein zeitgesteuerte Sequenz mit 5 Ausgabezuständen.



Schnittstellenstruktur

Ereignis-Eingänge

- **START_S1**: Startet die Sequenz bei State_01.
- **RESET**: Setzt die Sequenz zurück.

Ereignis-Ausgänge

- **CNF**: Bestätigung der Ausführung.

Daten-Eingänge

- **DT_S1_S2**: Zeit für Übergang State_01 -> State_02.
- **DT_S2_S3**: Zeit für Übergang State_02 -> State_03.
- **DT_S3_S4**: Zeit für Übergang State_03 -> State_04.
- **DT_S4_S5**: Zeit für Übergang State_04 -> State_05.
- **DT_S5_START**: Zeit für Übergang State_05 -> START.

Daten-Ausgänge

- **STATE_NR** (SINT): Aktuelle Zustandsnummer.

Adapter

- **DO_S1** (adapter::types::unidirectional::AX): Ausgangsadapter für State_01.
- **DO_S2** (adapter::types::unidirectional::AX): Ausgangsadapter für State_02.
- **DO_S3** (adapter::types::unidirectional::AX): Ausgangsadapter für State_03.
- **DO_S4** (adapter::types::unidirectional::AX): Ausgangsadapter für State_04.
- **DO_S5** (adapter::types::unidirectional::AX): Ausgangsadapter für State_05.
- **timeOut** (iec61499::events::ATimeOut): Timer-Adapter.

Funktionsweise

Entspricht `sequence_T_05`, verwendet jedoch Adapter für die Ausgänge.

Technische Besonderheiten

- Verwendung von `adapter::types::unidirectional::AX`.

Zustandsübersicht

Siehe `sequence_T_05`.

Anwendungsszenarien

Für zeitgesteuerte 5-stufige Sequenzen mit Adapteranbindung.

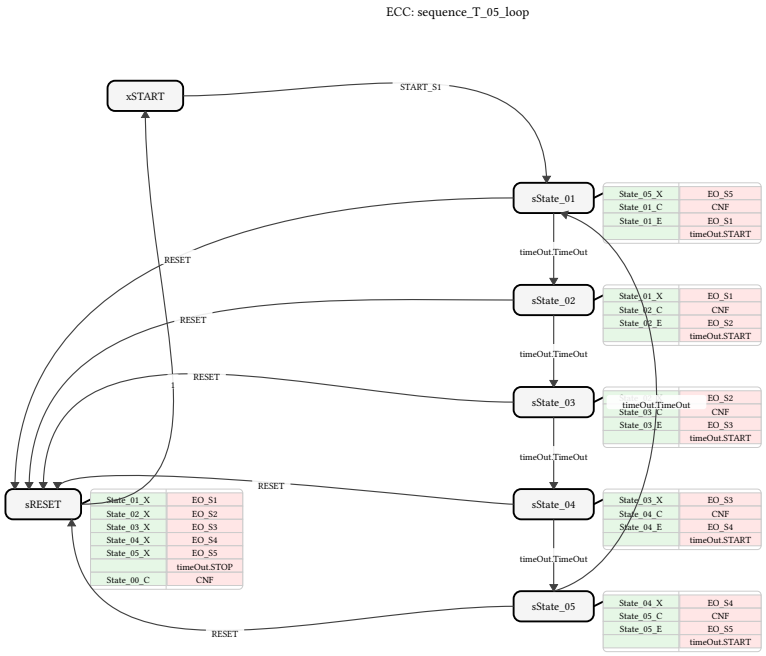
⚖️ Vergleich mit ähnlichen Bausteinen

- **sequence_T_05**: Standardvariante ohne Adapter.

Fazit

Adapter-Variante des 5-Schritt-Zeit-Sequenzers.

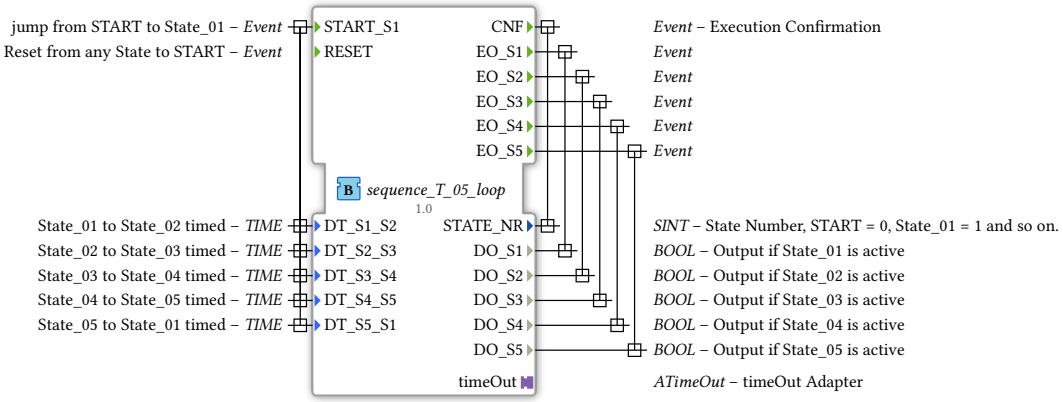
sequence_T_05_loop



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `sequence_T_05_loop` ist ein zeitgesteuerter Sequenzer, der einen zyklischen Ablauf von fünf Zuständen (State_01 bis State_05) realisiert. Der Übergang von einem Zustand zum nächsten erfolgt nach einer einstellbaren Zeitverzögerung. Der Baustein ist für Anwendungen konzipiert, in denen Aktionen oder Prozessschritte nacheinander und für eine definierte Dauer aktiviert werden müssen, beispielsweise in automatisierten Handhabungs- oder Fertigungsprozessen.



Schnittstellenstruktur

Ereignis-Eingänge

- **`START_S1`**: Startet die Sequenz und führt einen Übergang vom Initialzustand (`START`) in den ersten aktiven Zustand (`State_01`). Das Ereignis wird mit den fünf Zeitdaten-Eingängen verknüpft.
- **`RESET`**: Bricht die aktuelle Sequenz ab und führt in den Reset-Zustand (`sRESET`), von wo aus automatisch in den Initialzustand (`xSTART`) zurückgekehrt wird. Setzt alle Ausgänge zurück.

Ereignis-Ausgänge

- **CNF**: Bestätigungsereignis (Confirmation). Wird bei jedem Zustandswechsel ausgelöst und liefert die aktuelle Zustandsnummer (STATE_NR).
- **EO_S1** bis **EO_S5**: Zustandsereignisse. Werden beim Eintritt in den jeweiligen Zustand (State_01 bis State_05) ausgelöst und liefern den zugehörigen booleschen Datenausgang (DO_S1 bis DO_S5).

Daten-Eingänge

- **DT_S1_S2** (Typ: TIME, Initialwert: NO_TIME): Zeitdauer für den Übergang von State_01 zu State_02.
- **DT_S2_S3** (Typ: TIME, Initialwert: NO_TIME): Zeitdauer für den Übergang von State_02 zu State_03.
- **DT_S3_S4** (Typ: TIME, Initialwert: NO_TIME): Zeitdauer für den Übergang von State_03 zu State_04.
- **DT_S4_S5** (Typ: TIME, Initialwert: NO_TIME): Zeitdauer für den Übergang von State_04 zu State_05.
- **DT_S5_S1** (Typ: TIME, Initialwert: NO_TIME): Zeitdauer für den Übergang von State_05 zurück zu State_01, wodurch die Schleife geschlossen wird.

Daten-Ausgänge

- **STATE_NR** (Typ: SINT): Gibt die Nummer des aktuell aktiven Zustands aus. 0 = START, 1 = State_01, ..., 5 = State_05.
- **DO_S1** bis **DO_S5** (Typ: BOOL): Logische Ausgänge, die TRUE sind, solange sich der FB im zugehörigen Zustand (State_01 bis State_05) befindet.

Adapter

- **timeOut** (Typ: iec61499::events::ATimeOut, Plug): Ein Zeitgeber-Adapter, der für die zeitgesteuerten Zustandsübergänge verwendet wird. Der FB startet den Timer beim Eintritt in einen Zustand und wechselt beim Eingang des TimeOut-Ereignisses zum nächsten Zustand.

Funktionsweise

Der FB ist als Basic Function Block (BFB) mit einer Execution Control Chart (ECC) implementiert. Nach dem Start (START_S1) durchläuft er die Zustände State_01 bis State_05 in einer festen Reihenfolge. In jedem aktiven Zustand werden folgende Aktionen ausgeführt:

1. **Exit-Aktion des vorherigen Zustands**: Setzt den zugehörigen booleschen Ausgang (DO_Sx) auf FALSE.
2. **Confirmation-Aktion**: Setzt die aktuelle STATE_NR und konfiguriert die Verweildauer für den *aktuellen* Zustand im timeOut-Adapter (z.B. wird in State_01 der Wert von DT_S1_S2 geladen).
3. **Entry-Aktion des neuen Zustands**: Setzt den zugehörigen booleschen Ausgang (DO_Sx) auf TRUE.
4. **Timer-Start**: Startet den timeOut-Adapter mit der in Schritt 2 geladenen Zeit.

Der Übergang zum nächsten Zustand erfolgt ausschließlich, wenn der timeOut-Adapter das TimeOut-Ereignis liefert. Nach State_05 springt der FB gemäß DT_S5_S1 zurück zu State_01, wodurch eine Endlosschleife entsteht. Ein RESET-Ereignis von jedem Zustand aus deaktiviert alle Ausgänge, stoppt den Timer und bringt den FB in den Initialzustand xSTART zurück.

Technische Besonderheiten

- **Initialwerte**: Die Zeitdaten-Eingänge sind standardmäßig mit der Konstante NO_TIME initialisiert. Ein Wert von NO_TIME oder T#0s führt zu einem sofortigen Zustandsübergang.
- **Zustandsbestätigung**: Das CNF-Ereignis wird in jedem Zustand (inklusive Reset) ausgelöst, was eine zuverlässige Überwachung der FB-Aktivität von außen ermöglicht.
- **Adapter-Nutzung**: Die Zeitsteuerung ist komplett in den standardisierten ATimeOut-Adapter ausgelagert, was Wiederverwendbarkeit und klare Schnittstellen fördert.
- **Konstanten**: Der FB importiert Konstanten aus logiBUS::utils::sequence::const::sequence, z.B. für die Zustandsnummern (State_00, State_01, ...).

Zustandsübersicht

Die ECC umfasst sieben Zustände:

- **xSTART**: Inaktiver Initialzustand. Wartet auf START_S1.
- **sState_01** bis **sState_05**: Aktive Arbeitszustände. Jeder aktiviert seinen spezifischen Ausgang (DO_Sx) und startet den Timer für seinen eigenen Verbleib.
- **sRESET**: Reset-Zustand. Deaktiviert alle Ausgänge, stoppt den Timer und sendet eine Bestätigung (CNF) mit STATE_NR=0. Geht automatisch mit Condition 1 (immer wahr) zurück zu xSTART.

Anwendungsszenarien

- **Steuerung von Zyklusabläufen**: Automatisierte Abfolgen in Verpackungsmaschinen, wo verschiedene Aktoren (Greifer, Stempel, Förderer) nacheinander für bestimmte Zeiten aktiviert werden müssen.
- **Prozesssteuerung**: Schrittweise Durchführung von chemischen oder thermischen Prozessen, bei denen jeder Schritt eine definierte Haltezeit hat.
- **Anzeige- oder Blinksequenzen**: Steuerung von visuellen oder akustischen Signalen in einer festgelegten, zeitlich getakteten Reihenfolge.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu einem einfachen TON-Timer (Verzögerungseinschaltverzögerung) realisiert dieser FB eine vollständige Zustandsmaschine mit mehreren Schritten. Im Vergleich zu generischen Sequencer-FBs, die oft mit Schrittabfragen (z.B. über Steigungsflanken) arbeiten, ist dieser Baustein rein zeitgesteuert. Er ähnelt einer Kette von TON-Bausteinen, ist aber in einem einzigen, zustandsbasierten FB gekapselt, was die Anordnung und Parametrierung vereinfacht.

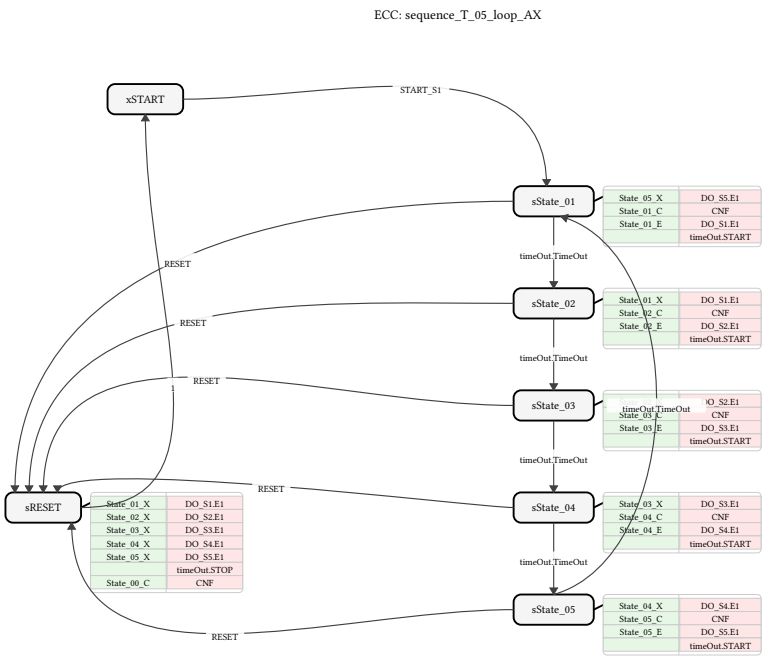
🛠️ Zugehörige Übungen

- Uebung_035a2
- Uebung_035a3

Fazit

Der `sequence_T_05_loop` ist ein spezialisierter, robuster und einfach zu konfigurierender Sequenzer für zyklische, zeitgesteuerte Abläufe mit fünf Schritten. Seine klare Struktur, die Nutzung standardisierter Adapter und die umfassende Bestätigung aller Zustandswechsel machen ihn zu einer zuverlässigen Komponente für zeitkritische Automatisierungsaufgaben. Die Schleifenfunktion und der zentrale Reset sind für Dauerbetriebsanwendungen besonders geeignet.

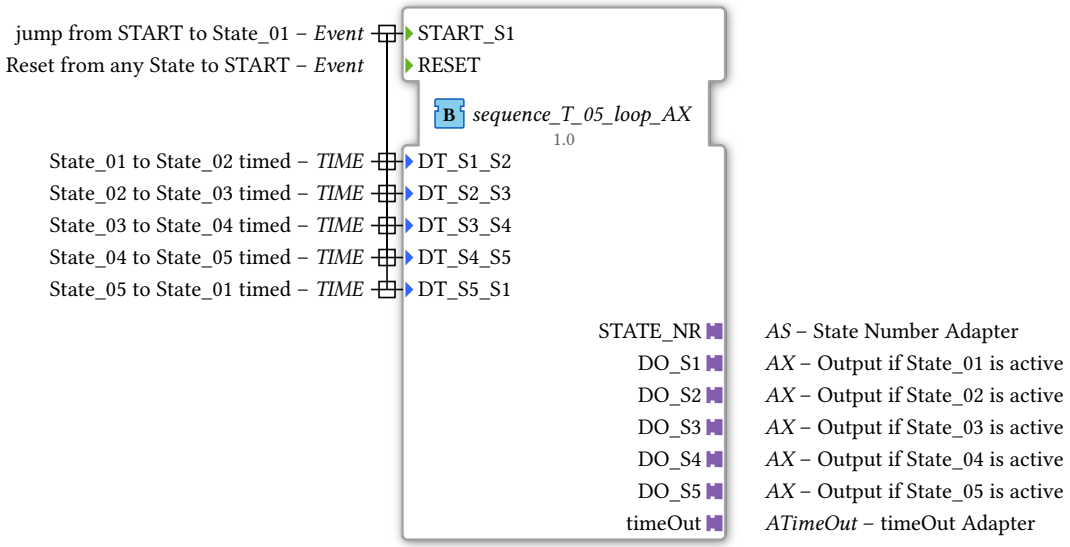
sequence_T_05_loop_AX



• – ○ • – ○ • – ○ •

Einleitung

`sequence_T_05_loop_AX` ist eine Variante des `sequence_T_05_loop`, die zusätzlich Adapter (`AX`) für die Ausgänge verwendet. Er steuert eine rein zeitgesteuerte, zyklische Sequenz mit 5 Ausgabezuständen.



Schnittstellenstruktur

Ereignis-Eingänge

- **START_S1**: Startet die Sequenz bei State_01.
- **RESET**: Setzt die Sequenz zurück.

Ereignis-Ausgänge

- **CNF**: Bestätigung der Ausführung.

Daten-Eingänge

- **DT_S1_S2**: Zeit für Übergang State_01 -> State_02.
- **DT_S2_S3**: Zeit für Übergang State_02 -> State_03.
- **DT_S3_S4**: Zeit für Übergang State_03 -> State_04.
- **DT_S4_S5**: Zeit für Übergang State_04 -> State_05.
- **DT_S5_S1**: Zeit für Übergang State_05 -> State_01.

Daten-Ausgänge

- **STATE_NR** (SINT): Aktuelle Zustandsnummer.

Adapter

- **DO_S1** (adapter::types::unidirectional::AX): Ausgangsadapter für State_01.
- **DO_S2** (adapter::types::unidirectional::AX): Ausgangsadapter für State_02.
- **DO_S3** (adapter::types::unidirectional::AX): Ausgangsadapter für State_03.
- **DO_S4** (adapter::types::unidirectional::AX): Ausgangsadapter für State_04.
- **DO_S5** (adapter::types::unidirectional::AX): Ausgangsadapter für State_05.
- **timeOut** (iec61499::events::ATimeOut): Timer-Adapter.

Funktionsweise

Entspricht `sequence_T_05_loop`, verwendet jedoch Adapter für die Ausgänge.

Technische Besonderheiten

- Verwendung von `adapter::types::unidirectional::AX`.

Zustandsübersicht

Siehe `sequence_T_05_loop`.

Anwendungsszenarien

Für zeitgesteuerte zyklische 5-stufige Sequenzen mit Adapteranbindung.

⚖️ Vergleich mit ähnlichen Bausteinen

- **sequence_T_05_loop**: Standardvariante ohne Adapter.

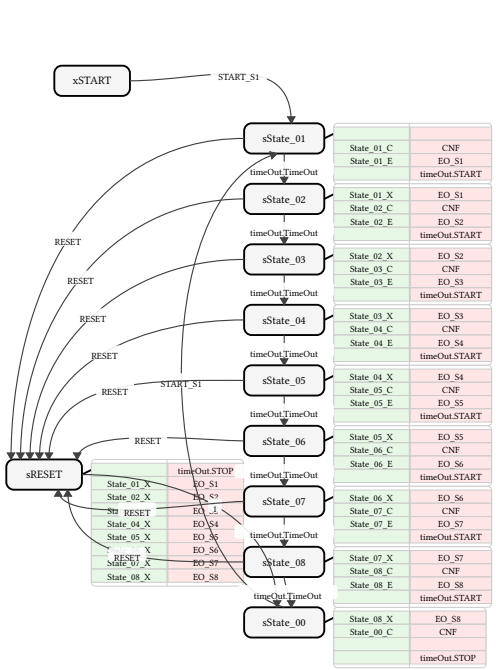
🛠️ Zugehörige Übungen

- Uebung_035a2_AX
- Uebung_035a3_AX

Fazit

Adapter-Variante des 5-Schritt-Loop-Zeit-Sequenzers.

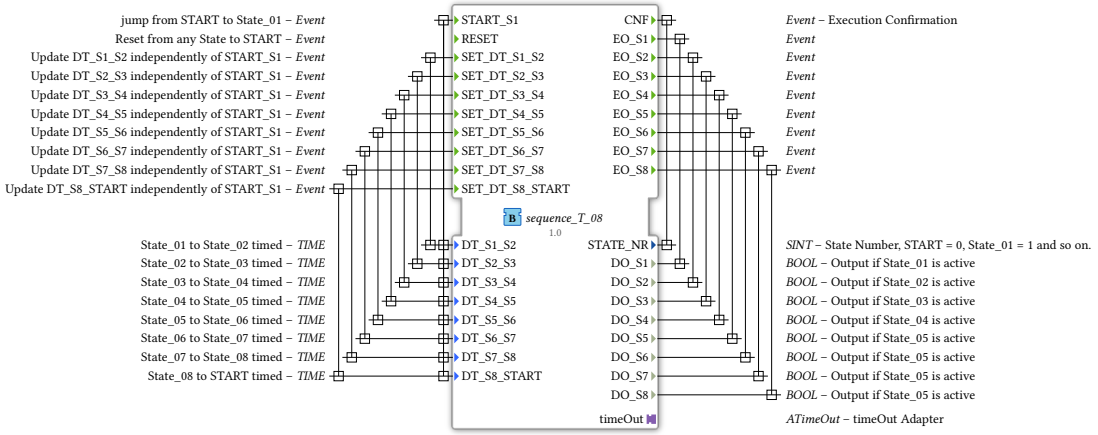
sequence_T_08



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `sequence_T_08` ist ein zeitgesteuerter Sequenzer mit acht Ausgängen. Er realisiert eine feste Abfolge von Zuständen, wobei der Übergang zwischen den einzelnen Zuständen durch einstellbare Zeitverzögerungen gesteuert wird. Der Baustein ist für Anwendungen konzipiert, in denen Prozessschritte oder Maschinenzustände nacheinander für eine definierte Dauer aktiviert werden müssen, wie z.B. in Verpackungsmaschinen, Förderstrecken oder Waschanlagen.



Schnittstellenstruktur

Ereignis-Eingänge

- `START_S1`: Startet die Sequenz. Ein Ereignis an diesem Eingang bewirkt den Übergang vom `START`-Zustand in den ersten aktiven Zustand `State_01`. Es werden alle acht Zeitdaten-Eingänge mitgelesen.
- `RESET`: Setzt die Sequenz sofort zurück. Ein Ereignis an diesem Eingang führt von jedem beliebigen Zustand zum `START`-Zustand und deaktiviert alle Ausgänge.
- `SET_DT_S1_S2`, `SET_DT_S2_S3`, `SET_DT_S3_S4`, `SET_DT_S4_S5`, `SET_DT_S5_S6`, `SET_DT_S6_S7`, `SET_DT_S7_S8`, `SET_DT_S8_START`: Aktualisieren die jeweils gleichnamige Zeit (`DT_S1_S2` bis `DT_S8_START`) unabhängig von `START_S1`. Befindet sich der FB gerade im zugehörigen Zustand, wird dessen Zeitvorgabe neu ausgeführt (Live-Update des laufenden Timers).

Ereignis-Ausgänge

- `CNF`: Bestätigungsereignis (Execution Confirmation). Wird bei jedem Zustandswechsel ausgelöst und liefert die neue Zustandsnummer `STATE_NR`.
- `EO_S1` bis `EO_S8`: Zustandsereignisse. Jedes dieser Ereignisse wird beim Eintritt in den entsprechenden Zustand (`State_01` bis `State_08`) ausgelöst und liefert den zugehörigen booleschen Datenausgangswert (`DO_S1` bis `DO_S8`).

Daten-Eingänge

- `DT_S1_S2` bis `DT_S8_START` (Typ `TIME`): Definieren die Verweilzeit für jeden Zustand. Der Wert legt fest, wie lange der FB im jeweiligen Zustand verbleibt, bevor der automatische Übergang zum nächsten Zustand erfolgt. Der Standardwert `NO_TIME` deaktiviert den zeitgesteuerten Übergang, sodass der FB im Zustand verharrt, bis ein `RESET` erfolgt.

Daten-Ausgänge

- `STATE_NR` (Typ `SINT`): Gibt die aktuelle Zustandsnummer aus. `0` entspricht dem `START`-Zustand, `1` bis `8` entsprechen den aktiven Zuständen `State_01` bis `State_08`.
- `DO_S1` bis `DO_S8` (Typ `BOOL`): Die physischen Ausgangssignale der Sequenz. Jeder Ausgang wird auf `TRUE` gesetzt, wenn der entsprechende Zustand aktiv ist, andernfalls ist er `FALSE`.

Adapter-Anschlüsse

- `timeOut` (Typ `iec61499::events::ATimeOut`): Ein Steckadapter, der die Zeitsteuerung bereitstellt. Der FB nutzt die Schnittstelle, um einen Timer zu starten (`timeOut.START`) und auf dessen Ablauf zu warten (`timeOut.TimeOut`). Die jeweilige Zeitdauer wird über `timeOut.DT` übergeben.

Funktionsweise

Der FB arbeitet als Basic Function Block (BFB) mit einer definierten Execution Control Chart (ECC). Die Sequenz durchläuft die Zustände `State_01` bis `State_08` in fester Reihenfolge. Jeder aktive Zustand führt drei wesentliche Aktionen aus:

1. **Exit-Aktion des vorherigen Zustands**: Deaktiviert den zugehörigen Ausgang (Algorithmus `State_XX_X`).

2. **Confirmation-Aktion**: Setzt die Zustandsnummer `STATE_NR` und übergibt die für diesen Zustand konfigurierte Zeitdauer an den Timer-Adapter (Algorithmus `State_XX_C`). Löst das `CNF`-Ereignis aus.
3. **Entry-Aktion**: Aktiviert den zugehörigen Ausgang (Algorithmus `State_XX_E`). Löst das entsprechende `EO_Sx`-Ereignis aus. Startet den Timer.

Der Übergang zum nächsten Zustand erfolgt ausschließlich, wenn der Timer abgelaufen ist (`timeOut.TimeOut`). Nach `State_08` springt der FB in den inaktiven `State_00` (auch als `START` dargestellt), wo der Timer gestoppt wird. Von hier aus kann die Sequenz nur durch ein neues `START_S1`-Ereignis neu gestartet werden. Ein `RESET`-Ereignis von jedem Zustand aus führt über einen dedizierten Reset-Zustand (`sRESET`) zurück in `State_00`, wobei dabei alle aktiven Ausgänge deaktiviert werden.

Technische Besonderheiten

- **Flexible Zeitsteuerung**: Jeder Zustandsübergang kann individuell und zur Laufzeit über die `DT_`-Eingänge konfiguriert werden. Der Wert `NO_TIME` erlaubt es, die Sequenz an einer bestimmten Stelle anzuhalten.
- **Sofortiger Reset**: Der `RESET`-Eingang hat jederzeit Priorität und unterbricht die laufende Zeitsteuerung sofort.
- **Zustandsrückmeldung**: Die aktuelle Position in der Sequenz ist über den `STATE_NR`-Ausgang stets nachvollziehbar.
- **Ereignisgesteuerte Ausgänge**: Neben den kontinuierlichen Datenausgängen (`DO_Sx`) bietet der FB für jeden Zustand ein separates Ereignis (`EO_Sx`), was die Ansteuerung nachgelagerter, ereignisgesteuerter FB erleichtert.

Zustandsübersicht

Die ECC umfasst folgende Zustände:

- **xSTART / sState_00**: Inaktiver Start- und Endzustand. `STATE_NR = 0`, alle Ausgänge sind `FALSE`.
- **sState_01 bis sState_08**: Die acht aktiven Sequenzzustände. `STATE_NR = 1` bis `8`. Der entsprechende Ausgang `DO_Sx` ist `TRUE`.
- **sRESET**: Interner Reset-Zustand. Deaktiviert alle Ausgänge und führt zurück zu `sState_00`.

Anwendungsszenarien

- **Steuerung von Zyklusabläufen**: In Maschinen, bei denen verschiedene Aktoren (Ventile, Motoren, Heizungen) nacheinander für eine bestimmte Zeit ein- und ausgeschaltet werden müssen.
- **Batch-Prozesse**: Für die Reihensteuerung von Prozessschritten in der Lebensmittel- oder Chemieindustrie, z.B. Füllen, Erhitzen, Rühren, Abkühlen.
- **Teststände**: Automatisierte Abfolge von Prüf- und Messschritten an einem Bauteil.
- **Sicherheitssequenzen**: Geordnetes An- und Abfahren einer Anlage, bei dem Schritte erst nach einer Wartezeit ausgelöst werden dürfen.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu einem einfachen TON (Timer On-Delay) oder TOF (Timer Off-Delay), die nur einen einzigen zeitverzögerten Schaltvorgang steuern, orchestriert `sequence_T_08` eine komplette Kette von zeitgesteuerten Aktionen. Im Vergleich zu einem Zählerbaustein (CTU) bietet er eine vordefinierte, zustandsbasierte Logik mit dedizierten Ausgängen und ist daher einfacher zu konfigurieren und abzusichern. Er ist eine spezialisierte Form einer Schrittkette (Sequential Function Chart - SFC) mit fester Schrittanzahl und reiner Zeitsteuerung.

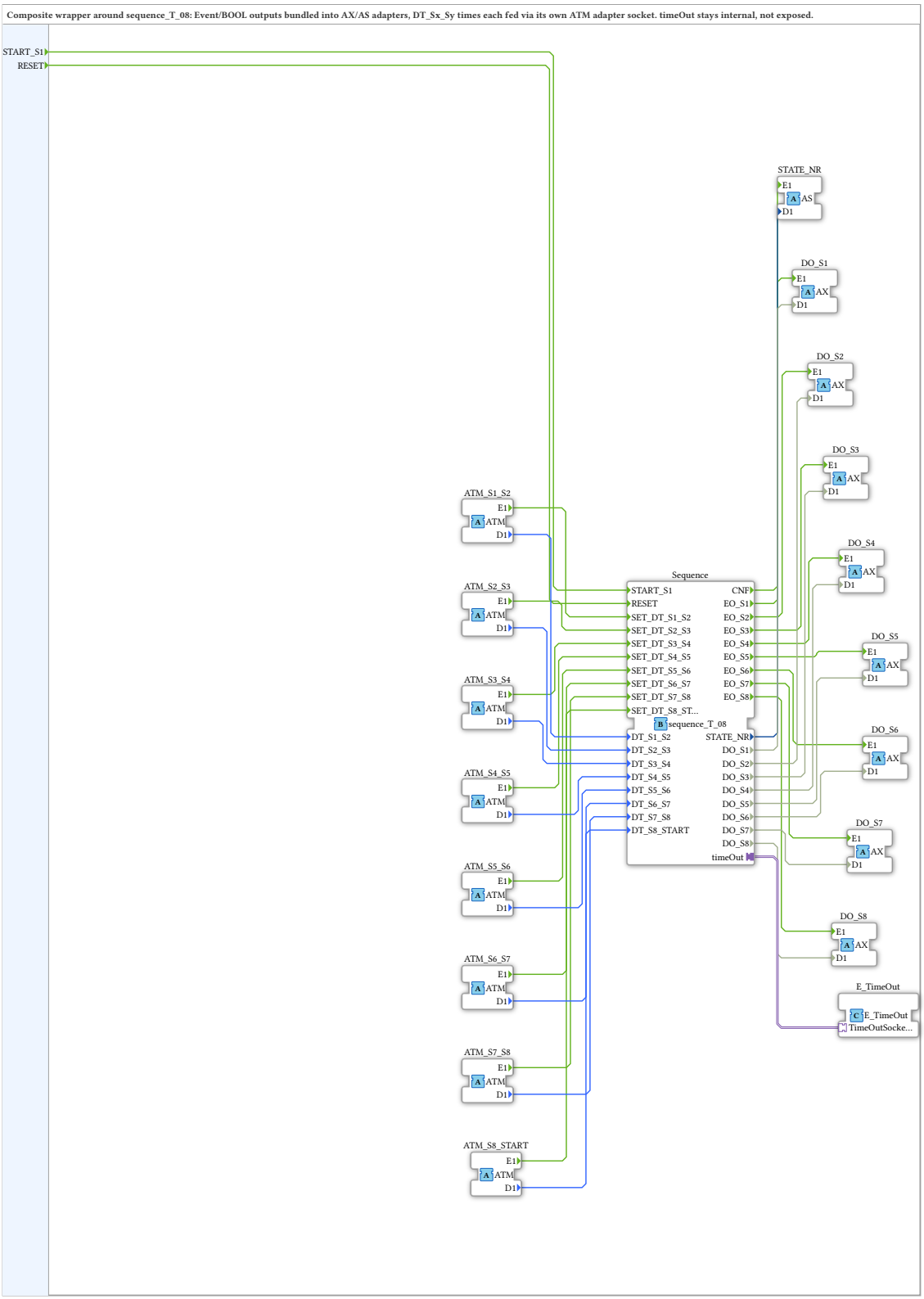
Fazit

Der `sequence_T_08` ist ein robuster und einfach zu konfigurierender Funktionsblock für alle Anwendungen, die eine feste, zeitgesteuerte Abfolge von Schritten erfordern. Durch die klare Trennung von Zustandslogik, Zeitparametern und Ausgangssignalen sowie die sofortige Reset-Funktionalität eignet er sich besonders für übersichtliche und wartungsfreundliche Steuerungsprogramme. Die Integration eines Standard-Timer-Adapters macht ihn zudem portabel und wiederverwendbar in verschiedenen 4diac-Projekten.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

sequence_T_08_ADAPTER

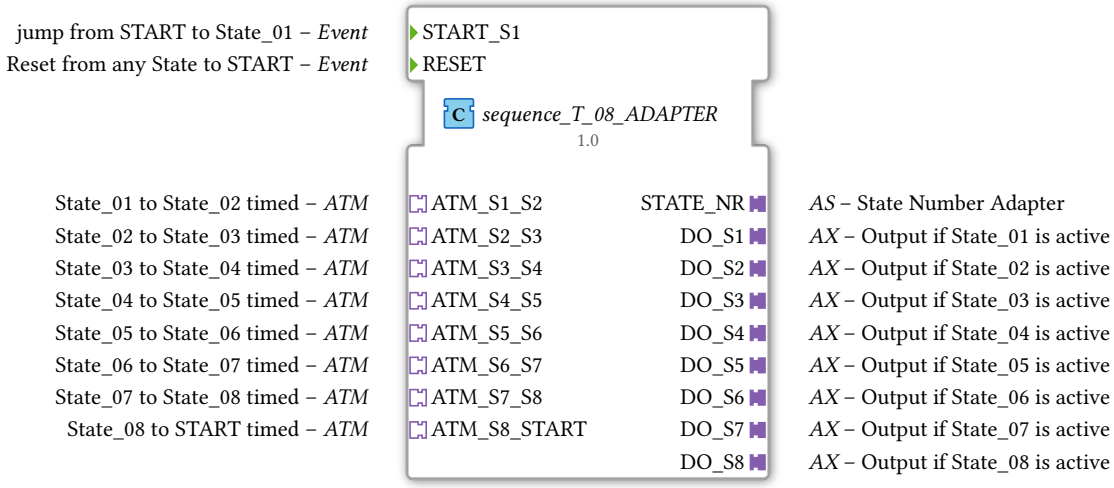


• — ○ • — ○ • — ○ •

Einleitung

`sequence_T_08_ADAPTER` ist ein Composite-Wrapper um sequence_T_08: derselbe generische zeitgesteuerte 8-Schritt-Sequenzer, aber mit allen Ausgängen und Übergangszeiten auf Adapterverbindungen umgestellt. Anders als sequence_T_08_AX, das nur die acht Ausgänge auf `AX`-Adapter umstellt, bündelt

sequence_T_08_ADAPTER zusätzlich die Zustandsnummer (AS) und führt jede der acht Übergangszeiten über einen eigenen ATM-Adapter-Socket ein.



Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: sequence_T_08_ADAPTER

- **Typ**: Composite (FBNetwork)
- **Verwendete interne FBs**:
  - **Sequence**: `logiBUS::utils::sequence::timed::sequence_T_08` — der generische 8-Schritt-Zeitsequenzer, siehe sequence_T_08.
  - **E_TimeOut**: `iec61499::events::E_TimeOut` — Standard-TimeOut-Adapter für die interne Zeitsteuerung; anders als bei `AnlagenSequenz_06_ADAPTER` genügt hier ein einzelner Standard-FB ohne zusätzlichen `TimeTicker`.
- **Funktionsweise**: `DO_S1`..`DO_S8` (Ausgänge je Zustand) und `STATE_NR` (aktuelle Zustandsnummer) laufen über `AX`- bzw. `AS`-Adapter; jede der acht Übergangszeiten `DT_S1_S2`..`DT_S8_START` wird über einen eigenen `ATM`-Adapter-Socket (`ATM_S1_S2`..`ATM_S8_START`) eingespeist statt als einzelner `TIME`-Dateneingang.

Programmablauf und Verbindungen

1. `START_S1`/`RESET` werden unverändert an `Sequence.START_S1`/`Sequence.RESET` durchgereicht.
2. Jeder `ATM_Sx_Sy`-Adapter liefert seine Zeit (`.D1`) an `Sequence.DT_Sx_Sy` und löst gleichzeitig (`.E1`) das zugehörige `Sequence.SET_DT_Sx_Sy`-Setzereignis aus.
3. `Sequence.CNF` löst `STATE_NR.E1`, mit der aktuellen Zustandsnummer über `STATE_NR.D1`.
4. Jedes `Sequence.EO_Sx` löst `DO_Sx.E1`, mit dem Ausgangswert über `DO_Sx.D1`.
5. `Sequence.timeOut` (Adapter-Socket) ist intern mit `E_TimeOut.TimeOutSocket` verbunden — die Zeitsteuerung bleibt vollständig innerhalb des Composite gekapselt, im Gegensatz zu `sequence_T_08`/`sequence_T_08_AX`, die ihren `timeOut`-Adapter selbst nach außen führen.

Anwendungsszenarien

- Anwendungen, die konsequent auf Adapterverbindungen setzen und weder einzelne `TIME`-Leitungen für die Übergangszeiten noch eine eigene `timeOut`-Adapterverbindung in der Anwendung ziehen wollen.
- Wiederverwendung des generischen 8-Schritt-Sequenzers in mehreren Anlagenteilen mit einheitlicher, adapterbasierter Verdrahtung.

⚖️ Vergleich mit ähnlichen Bausteinen

- **sequence_T_08**: Die Basisvariante mit klassischen Event-/Daten-/Adapter-Anschlüssen (nur `timeOut` als Adapter).
- **sequence_T_08_AX**: Stellt nur die acht Ausgänge auf `AX`-Adapter um, behält aber `TIME`-Dateneingänge für die Übergangszeiten und führt `timeOut` weiterhin selbst nach außen.
- **AnlagenSequenz_06_ADAPTER**: Gleiches Adapter-Wrapper-Muster, angewendet auf die feste 6-Motoren-Ringtopologie statt auf den generischen 8-Schritt-Sequenzer.

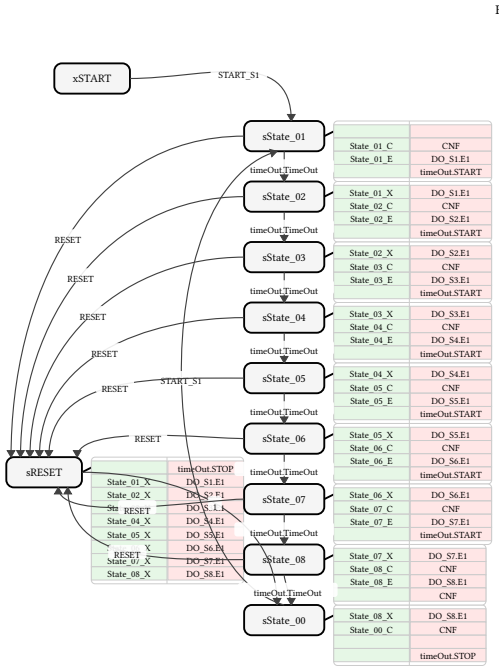
Fazit

`sequence_T_08_ADAPTER` macht den generischen 8-Schritt-Zeitsequenzer vollständig über Adapterverbindungen nutzbar — Ausgänge, Zustandsnummer, Übergangszeiten und Zeitsteuerung laufen ausschließlich über Adapter, keine einzelne Event-/Daten-Leitung verlässt den Baustein.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

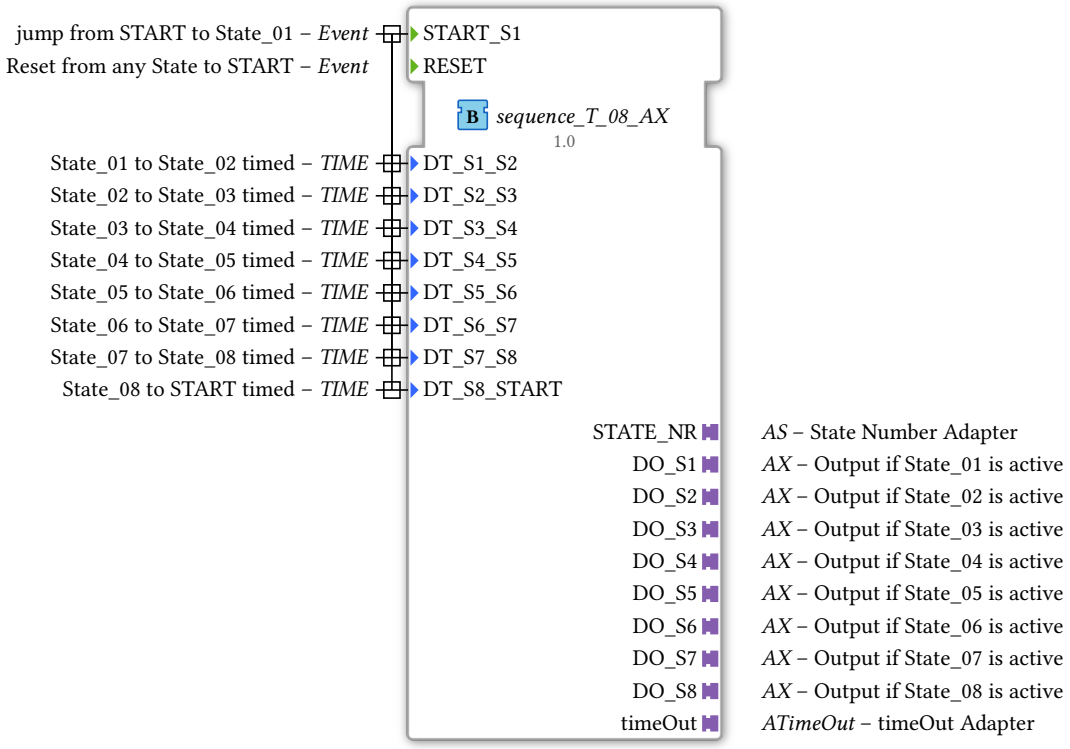
sequence_T_08_AX



• — ○ • — ○ • — ○ •

Einleitung

`sequence_T_08_AX` ist eine Variante des `sequence_T_08`, die zusätzlich Adapter (`AX`) für die Ausgänge verwendet. Er steuert eine rein zeitgesteuerte Sequenz mit 8 Ausgabezuständen.



Schnittstellenstruktur

Ereignis-Eingänge

- **START_S1**: Startet die Sequenz bei State_01.
- **RESET**: Setzt die Sequenz zurück.

Ereignis-Ausgänge

- **CNF**: Bestätigung der Ausführung.

Daten-Eingänge

- **DT_S1_S2**: Zeit für Übergang State_01 -> State_02.
- **DT_S2_S3**: Zeit für Übergang State_02 -> State_03.
- **DT_S3_S4**: Zeit für Übergang State_03 -> State_04.
- **DT_S4_S5**: Zeit für Übergang State_04 -> State_05.
- **DT_S5_S6**: Zeit für Übergang State_05 -> State_06.
- **DT_S6_S7**: Zeit für Übergang State_06 -> State_07.
- **DT_S7_S8**: Zeit für Übergang State_07 -> State_08.
- **DT_S8_START**: Zeit für Übergang State_08 -> START.

Daten-Ausgänge

- **STATE_NR** (SINT): Aktuelle Zustandsnummer.

Adapter

- **DO_S1** ... **DO_S8** (adapter::types::unidirectional::AX): Ausgangsadapter für State_01 bis State_08.
- **timeOut** (iec61499::events::ATimeOut): Timer-Adapter.

Funktionsweise

Entspricht `sequence_T_08`, verwendet jedoch Adapter für die Ausgänge.

Technische Besonderheiten

- Verwendung von `adapter::types::unidirectional::AX`.

Zustandsübersicht

Siehe `sequence_T_08`.

Anwendungsszenarien

Für zeitgesteuerte 8-stufige Sequenzen mit Adapteranbindung.

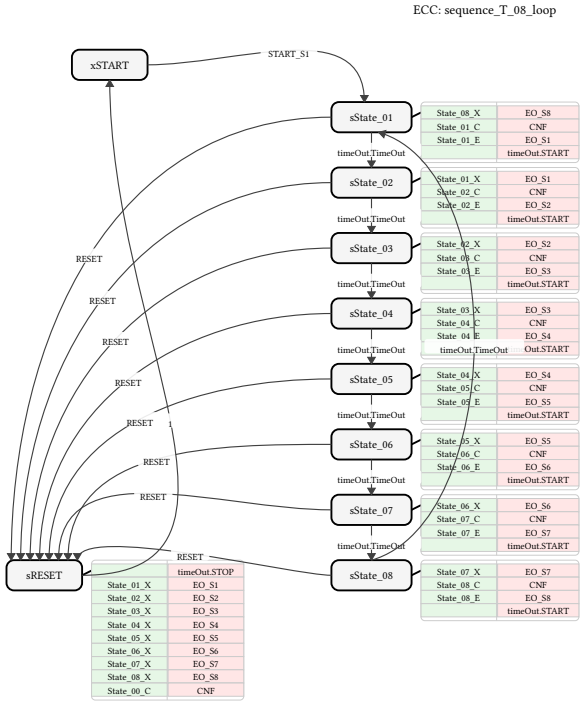
⚖️ Vergleich mit ähnlichen Bausteinen

- **sequence_T_08**: Standardvariante ohne Adapter.

Fazit

Adapter-Variante des 8-Schritt-Zeit-Sequenzers.

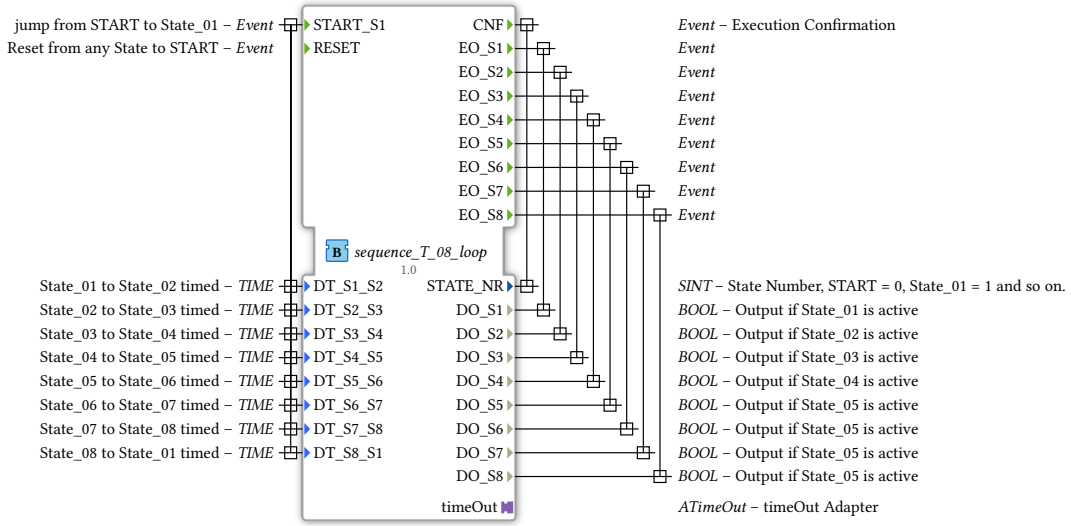
sequence_T_08_loop



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock `sequence_T_08_loop` ist ein zeitgesteuerter Sequenzer mit acht Ausgangszuständen. Er realisiert eine zyklische Abfolge von Zuständen, wobei der Übergang zwischen den einzelnen Zuständen durch einstellbare Zeitverzögerungen gesteuert wird. Der Baustein ist für Anwendungen konzipiert, in denen Prozessschritte oder Maschinenzustände nacheinander für eine definierte Dauer aktiviert werden müssen, beispielsweise in Förderanlagen, Verpackungsmaschinen oder automatisierten Montageprozessen.



Schnittstellenstruktur

Ereignis-Eingänge

- **`START_S1`**: Startet die Sequenz durch einen Sprung vom initialen `START`-Zustand in den ersten aktiven Zustand `State_01`. Löst die Ausführung der zugehörigen Algorithmen aus.
- **`RESET`**: Setzt die Sequenz unabhängig vom aktuellen Zustand sofort in den initialen `START`-Zustand zurück. Alle aktiven Ausgänge werden deaktiviert.

Ereignis-Ausgänge

- **`CNF`** (Execution Confirmation): Wird bei jedem Zustandswechsel ausgelöst und bestätigt die Ausführung. Überträgt die aktuelle Zustandsnummer `STATE_NR`.
- **`EO_S1` bis `EO_S8`**: Ereignisausgänge, die beim Eintritt in den jeweiligen Zustand `State_01` bis `State_08` ausgelöst werden. Sie übertragen den zugehörigen booleschen Datenwert (`DO_S1` bis `DO_S8`), der auf `TRUE` gesetzt wird.

Daten-Eingänge

Acht Zeitdaten-Eingänge vom Typ `TIME`, die die Verweildauer in jedem Zustand definieren. Der Standardwert ist `NO_TIME` (keine Zeit, sofortiger Übergang).

- `DT_S1_S2`: Verweildauer in `State_01` vor Übergang zu `State_02`.
- `DT_S2_S3`: Verweildauer in `State_02` vor Übergang zu `State_03`.
- `DT_S3_S4`: Verweildauer in `State_03` vor Übergang zu `State_04`.
- `DT_S4_S5`: Verweildauer in `State_04` vor Übergang zu `State_05`.
- `DT_S5_S6`: Verweildauer in `State_05` vor Übergang zu `State_06`.
- `DT_S6_S7`: Verweildauer in `State_06` vor Übergang zu `State_07`.
- `DT_S7_S8`: Verweildauer in `State_07` vor Übergang zu `State_08`.
- `DT_S8_S1`: Verweildauer in `State_08` vor dem zyklischen Übergang zurück zu `State_01`.

Daten-Ausgänge

- **`STATE_NR`** (SINT): Gibt die Nummer des aktuell aktiven Zustands aus. `START` = 0, `State_01` = 1, ..., `State_08` = 8.
- **`DO_S1` bis `DO_S8`** (BOOL): Die physischen Ausgangssignale der Sequenz. Jeder Ausgang wird auf `TRUE` gesetzt, wenn der entsprechende Zustand aktiv ist, ansonsten ist er `FALSE`.

Adapter

- **`timeOut`** (Plug, Typ: `iec61499::events::ATimeOut`): Ein Zeitgeber-Adapter, der für die Realisierung der zeitgesteuerten Zustandsübergänge verwendet wird. Der FB startet den Timer beim Eintritt in einen Zustand und wechselt bei Empfang des `TimeOut`-Ereignisses zum nächsten Zustand.

Funktionsweise

Der FB arbeitet als Basic Function Block mit einem internen Execution Control Chart (ECC). Die Sequenz beginnt im initialen Zustand `xSTART`. Ein `START_S1`-Ereignis führt zum Zustand `sState_01`. Beim Eintritt in einen Zustand (z.B. `sState_01`) werden drei Aktionen ausgeführt:

1. Der Exit-Algorithmus des vorherigen Zustands (z.B. `State_08_X`) deaktiviert dessen Ausgang.
2. Der Confirmation-Algorithmus (z.B. `State_01_C`) setzt die `STATE_NR` und lädt die für diesen Zustand konfigurierte Zeit (`DT_S1_S2`) in den `timeOut`-Adapter.
3. Der Entry-Algorithmus (z.B. `State_01_E`) setzt den zugehörigen Datenausgang (`DO_S1`) auf `TRUE` und löst das entsprechende Ereignis (`EO_S1`) aus.
4. Der Timer-Adapter wird mit `timeOut.START` gestartet.

Nach Ablauf der eingestellten Zeit löst der Adapter das `timeOut.TimeOut`-Ereignis aus, was im ECC den Übergang zum nächsten Zustand (z.B. von `sState_01` zu `sState_02`) bewirkt. Nach dem letzten Zustand (`sState_08`) erfolgt der Übergang zurück zum ersten Zustand (`sState_01`), wodurch eine Endlosschleife entsteht.

Ein `RESET`-Ereignis von jedem Zustand aus führt zum Zustand `sRESET`. Hier werden alle Ausgänge deaktiviert, der Timer gestoppt und eine Bestätigung mit `STATE_NR=0` gesendet, bevor automatisch in den `xSTART`-Zustand zurückgekehrt wird.

Technische Besonderheiten

- **Zyklus ohne Stopp**: Die Sequenz läuft nach dem Start endlos im Kreis, bis ein `RESET`-Signal empfangen wird. Es gibt keinen integrierten Stopp-Befehl.
- **Sofortige Zustandsübergänge**: Durch Setzen der Zeitwerte auf `NO_TIME` (Standard) kann der FB so konfiguriert werden, dass er sofort zum nächsten Zustand wechselt, sobald der Eintrittsalgorithmus des aktuellen Zustands abgearbeitet ist.
- **Deterministische Abarbeitung**: Die Algorithmen für Exit, Confirmation und Entry werden in genau dieser Reihenfolge und atomar beim Zustandseintritt ausgeführt.
- **Konstanten für Zustandsnummern**: Die `STATE_NR` wird über Konstanten aus dem Import `sequence` (z.B. `sequence::State_01`) gesetzt, was die Wartbarkeit und Lesbarkeit verbessert.

Zustandsübersicht

Der ECC besteht aus 10 Zuständen:

- **`xSTART`**: Initialer, inaktiver Zustand. Wartet auf `START_S1`.
- **`sState_01` bis `sState_08`**: Die acht aktiven Sequenzzustände. Jeder verwaltet seinen eigenen Ausgang und die Zeit bis zum nächsten Zustand.
- **`sRESET`**: Reset-Zustand. Wird bei `RESET`-Ereignis von jedem Zustand aus angesprungen, deaktiviert alle Ausgänge und kehrt dann zu `xSTART` zurück.

Die Übergangsbedingungen sind:

- `START_S1`: Von `xSTART` nach `sState_01`.
- `timeOut.TimeOut`: Von jedem Zustand `sState_XX` zum nächsten `sState_YY` (zyklisch von `sState_08` zu `sState_01`).
- `RESET`: Von jedem aktiven Zustand (`sState_01`-`sState_08`) nach `sRESET`.
- `1` (immer wahr): Von `sRESET` zurück zu `xSTART`.

Anwendungsszenarien

- **Steuerung von Rundtaktmaschinen**: Aktivierung verschiedener Werkzeuge oder Stationen in einer rotierenden Maschine für eine genau definierte Dauer.
- **Batch-Prozesse in der Verfahrenstechnik**: Schrittweise Steuerung von Ventilen, Pumpen und Heizungen in einem chemischen Prozess mit festen Zykluszeiten.
- **Automatisierte Prüfsequenzen**: Nacheinander Durchführung verschiedener Messungen oder Tests an einem Bauteil.
- **Licht- oder Signalanlagen**: Erzeugung von festen Blink- oder Lauflichtmustern.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu einem `E_CYCLE`- oder `E_DELAY`-FB, die einfache periodische oder verzögerte Ereignisse erzeugen, bietet `sequence_T_08_loop` eine strukturierte Zustandsmaschine mit mehreren unabhängigen Ausgängen. Im Vergleich zu einem frei programmierbaren `E_CTU` (Zähler) in Kombination mit `SEL`-

Blöcken ist dieser FB vorkonfiguriert und damit einfacher und schneller für Standard-Sequenzen mit bis zu acht Schritten einsetzbar. Für komplexere oder zustandsabhängige Sequenzen müsste auf einen Service Sequence Function Block (SFC) oder eine individuelle Basic FB-Programmierung zurückgegriffen werden.

🛠️ Zugehörige Übungen

- Uebung_038

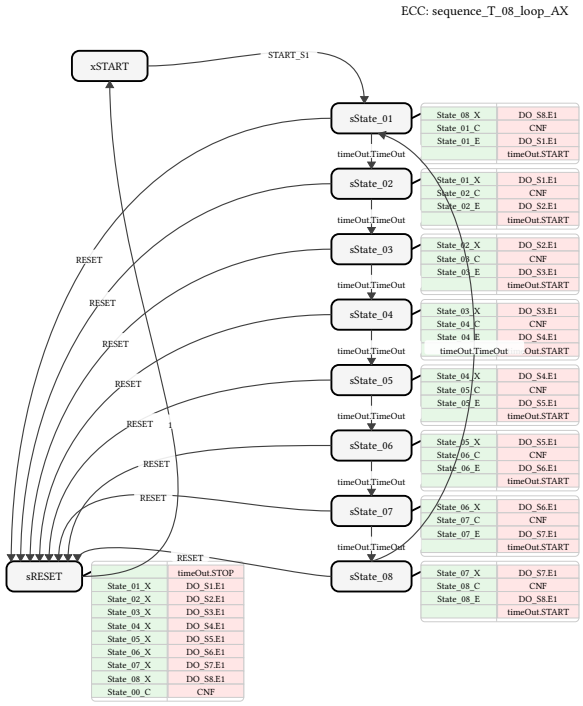
Fazit

Der `sequence_T_08_loop` ist ein robuster und einfach zu konfigurierender Funktionsblock für zeitgesteuerte Sequenzen mit fester Schrittanzahl. Seine klare Struktur aus Zuständen, konfigurierbaren Verweilzeiten und dedizierten Ausgängen macht ihn besonders geeignet für standardisierte Steuerungsaufgaben in der Automatisierungstechnik, bei denen Zuverlässigkeit und leichte Parametrierung im Vordergrund stehen. Die Integration eines Timer-Adapters entkoppelt die Zeitmessung von der FB-Logik und fördert die Wiederverwendbarkeit.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de

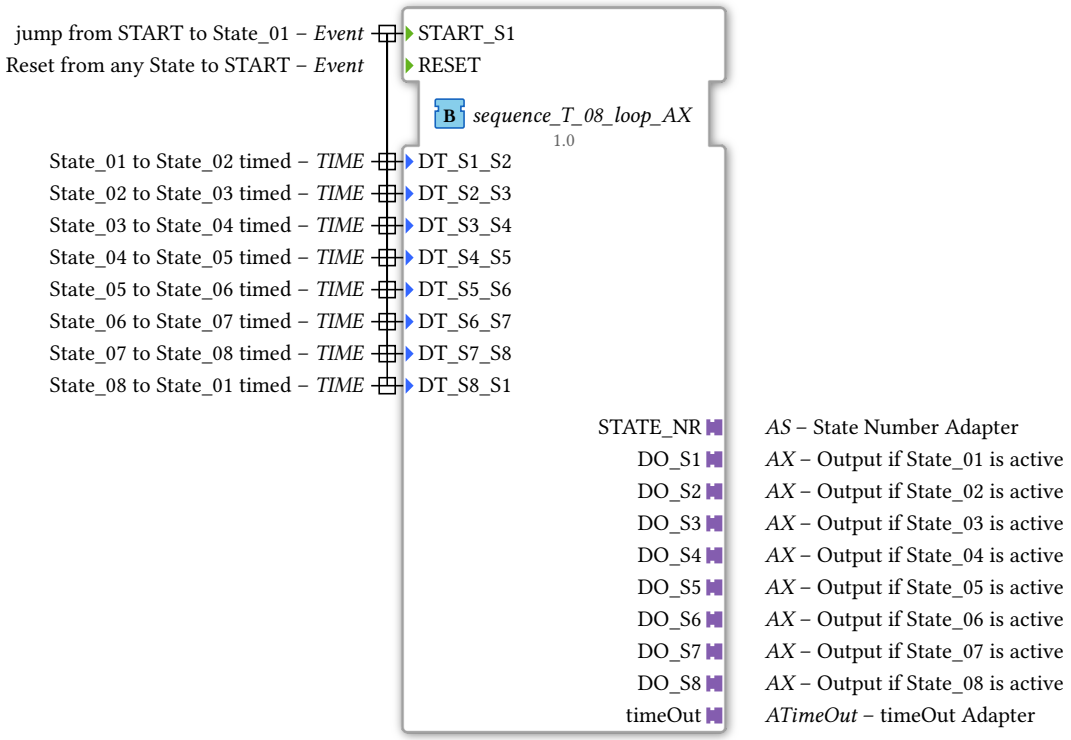
sequence_T_08_loop_AX



• — ○ • — ○ • — ○ •

Einleitung

`sequence_T_08_loop_AX` ist eine Variante des `sequence_T_08_loop`, die zusätzlich Adapter (`AX`) für die Ausgänge verwendet. Er steuert eine rein zeitgesteuerte, zyklische Sequenz mit 8 Ausgabezuständen.



Schnittstellenstruktur

Ereignis-Eingänge

- **START_S1**: Startet die Sequenz bei State_01.
- **RESET**: Setzt die Sequenz zurück.

Ereignis-Ausgänge

- **CNF**: Bestätigung der Ausführung.

Daten-Eingänge

- **DT_S1_S2** ... **DT_S8_S1**: Zeiten für die automatischen Übergänge zwischen den Zuständen.

Daten-Ausgänge

- **STATE_NR** (SINT): Aktuelle Zustandsnummer.

Adapter

- **DO_S1** ... **DO_S8** (adapter::types::unidirectional::AX): Ausgangsadapter für State_01 bis State_08.
- **timeOut** (iec61499::events::ATimeOut): Timer-Adapter.

Funktionsweise

Entspricht `sequence_T_08_loop`, verwendet jedoch Adapter für die Ausgänge.

Technische Besonderheiten

- Verwendung von `adapter::types::unidirectional::AX`.

Zustandsübersicht

Siehe `sequence_T_08_loop`.

Anwendungsszenarien

Für zeitgesteuerte zyklische 8-stufige Sequenzen mit Adapteranbindung.

⚖️ Vergleich mit ähnlichen Bausteinen

- **sequence_T_08_loop**: Standardvariante ohne Adapter.

🛠️ Zugehörige Übungen

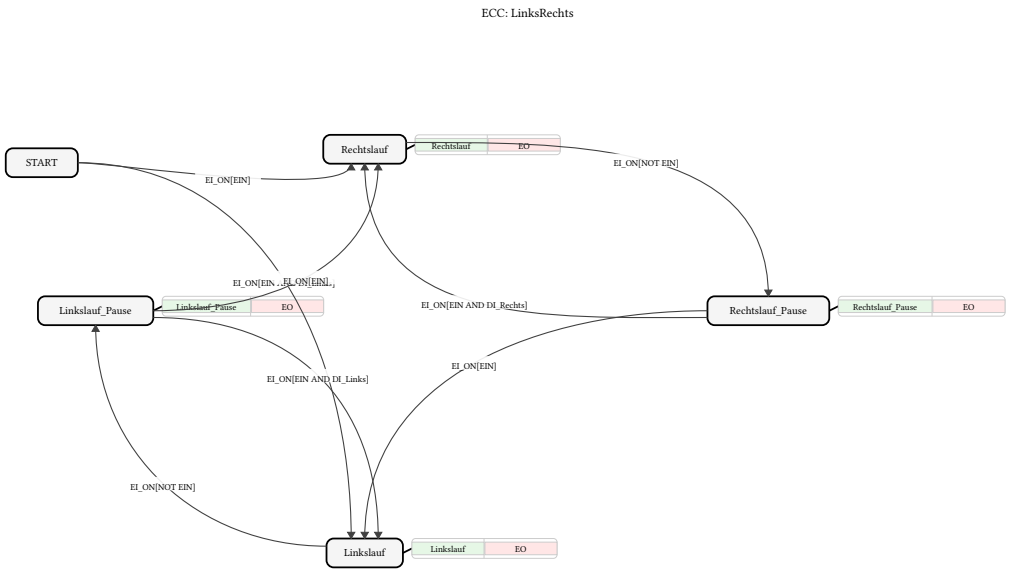
- Uebung_038_AX

Fazit

Adapter-Variante des 8-Schritt-Loop-Zeit-Sequenzers.

verteiler

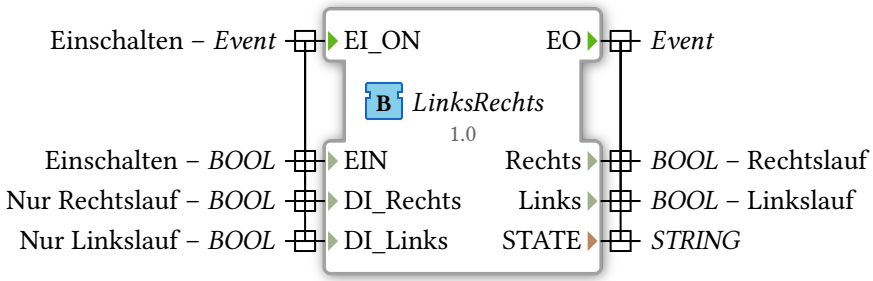
LinksRechts



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `LinksRechts` ist ein grundlegender Baustein zur Steuerung einer bidirektionalen Bewegung, beispielsweise eines Antriebs, der sowohl im Rechts- als auch im Linkslauf betrieben werden kann. Er realisiert eine einfache Priorisierungslogik, bei der der Rechtslauf Vorrang vor dem Linkslauf hat, es sei denn, ein spezieller Linkslauf-Befehl ist aktiv. Der Block ermöglicht zudem das pausieren der Bewegung.



Schnittstellenstruktur

Ereignis-Eingänge

- **`EI_ON`**: Das zentrale Steuerereignis. Es löst bei jedem Eintreffen eine Auswertung der aktuellen Eingangsdaten und einen potenziellen Zustandsübergang aus.

Ereignis-Ausgänge

- **EO**: Dieses Ereignis wird bei jedem Zustandswechsel ausgelöst. Es liefert die aktualisierten Ausgangsdaten Rechts, Links und STATE.

Daten-Eingänge

- **EIN** (BOOL): Allgemeiner Freigabe-/Einschaltbefehl. Bei TRUE ist der Betrieb erlaubt, bei FALSE geht der Block in einen Pause-Zustand.
- **DI_Rechts** (BOOL): Befehl für „Nur Rechtslauf“. Setzt den Rechtslauf durch, sofern EIN aktiv ist.
- **DI_Links** (BOOL): Befehl für „Nur Linkslauf“. Setzt den Linkslauf durch, sofern EIN aktiv ist und kein DI_Rechts-Befehl vorliegt.

Daten-Ausgänge

- **Rechts** (BOOL): Steuersignal für den Rechtslauf. Ist TRUE, wenn der Block im Zustand Rechtslauf ist.
- **Links** (BOOL): Steuersignal für den Linkslauf. Ist TRUE, wenn der Block im Zustand Linkslauf ist.
- **STATE** (STRING): Zeigt den aktuellen internen Zustand des Funktionsblocks als lesbaren Text an (z.B. „Rechtslauf“, „Linkslauf_Pause“).

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der LinksRechts-Block ist als Basic-FB mit einer Ereignissteuerung (ECC) implementiert. Das Eintreffen des Ereignisses EI_ON bewirkt eine Auswertung der mitgeführten Daten EIN, DI_Rechts und DI_Links. Basierend auf der aktuellen Kombination dieser Werte und dem aktuellen Zustand findet ein Übergang in einen neuen Zustand statt.

In jedem Zustand wird ein spezifischer Algorithmus ausgeführt, der die Ausgangssignale Rechts und Links setzt und den Zustandsnamen in STATE schreibt. Anschließend wird das Ausgangsereignis EO erzeugt, um nachgelagerte Blöcke über die Änderung zu informieren.

Die Prioritätslogik ist wie folgt definiert: Wenn EIN aktiv ist (TRUE), wird zuerst DI_Rechts geprüft. Ist dieser TRUE, wird der Rechtslauf aktiviert. Ist DI_Rechts FALSE, wird DI_Links geprüft. Ist dieser TRUE, wird der Linkslauf aktiviert. Ist EIN FALSE, wechselt der Block unabhängig von den Laufbefehlen in einen Pause-Zustand.

Technische Besonderheiten

- **Priorität**: Die Spezifikation betont, dass „Nur Rechtslauf“ (DI_Rechts) Vorrang vor „Nur Linkslauf“ (DI_Links) hat. Dies ist in der ECC-Transition von START nach Rechtslauf umgesetzt, die nur EIN benötigt, während der Übergang nach Linkslauf zusätzlich DI_Links erfordert.
- **Zustandsausgabe**: Die Ausgabe STATE ist vom Typ STRING und wird aus einer importierten Aufzählung STATES gespeist, was die Diagnose und Visualisierung erleichtert.
- **Pause-Zustände**: Es existieren zwei separate Pause-Zustände (Rechtslauf_Pause und Linkslauf_Pause). Diese merken sich die letzte aktive Laufrichtung. Bei erneuter Freigabe (EIN=TRUE) wird, sofern kein spezifischer Laufbefehl (DI_Rechts/DI_Links) anliegt, die zuletzt aktive Richtung fortgesetzt.

Zustandsübersicht

Die ECC (Execution Control Chart) des Bausteins umfasst fünf Zustände:

1. **START**: Initialzustand. Verlassen wird er mit dem ersten EI_ON-Ereignis.
2. **Rechtslauf**: Aktiver Zustand, in dem das Ausgangssignal Rechts auf TRUE gesetzt wird.
3. **Linkslauf**: Aktiver Zustand, in dem das Ausgangssignal Links auf TRUE gesetzt wird.
4. **Rechtslauf_Pause**: Pause-Zustand, der vom Rechtslauf aus erreicht wird, wenn EIN FALSE wird. Beide Ausgänge sind FALSE.
5. **Linkslauf_Pause**: Pause-Zustand, der vom Linkslauf aus erreicht wird, wenn EIN FALSE wird. Beide Ausgänge sind FALSE.

Die Übergänge zwischen den Zuständen werden ausschließlich durch das Ereignis EI_ON in Kombination mit den Datenbedingungen ausgelöst.

Anwendungsszenarien

Typische Anwendungen sind:

- Steuerung eines Wechselstrommotors mit zwei Drehrichtungen.
- Kontrolle einer horizontal verfahrbaren Einheit (z.B. Schlitten, Tor).
- Jede Applikation, bei der eine Vorwärts-/Rückwärtsbewegung mit einer allgemeinen Freigabe und individuellen Richtungsbefehlen gesteuert werden muss.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einem einfachen `SR`- oder `RS`-Flipflop bietet `LinksRechts` eine höhere Abstraktionsebene, da er die Priorisierungslogik und die Pause-Funktionalität bereits kapselt. Gegenüber einem reinen `E_SWITCH`-Block, der nur zwischen zwei Ausgängen umschaltet, verwaltet `LinksRechts` zusätzlich interne Zustände (Pause) und bietet eine definierte Priorisierung der Eingänge.

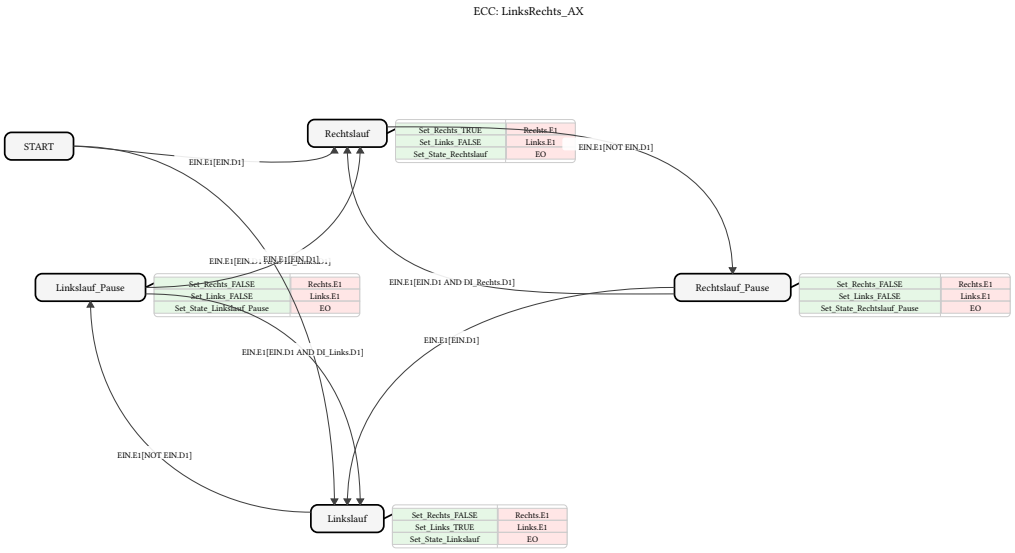
🛠️ Zugehörige Übungen

- Uebung_006a4

Fazit

Der `LinksRechts`-Funktionsblock ist ein nützlicher und robuster Grundbaustein für die Steuerung bidirektionaler Bewegungen. Durch die integrierte Priorisierungslogik (Rechtslauf vor Linkslauf) und die zustandsbasierte Pausefunktion vereinfacht er die Applikationsprogrammierung und erhöht die Übersichtlichkeit von Steuerungsprogrammen. Seine klare Schnittstelle und der ausgegebene Zustand unterstützen die Inbetriebnahme und Fehlersuche.

LinksRechts_AX



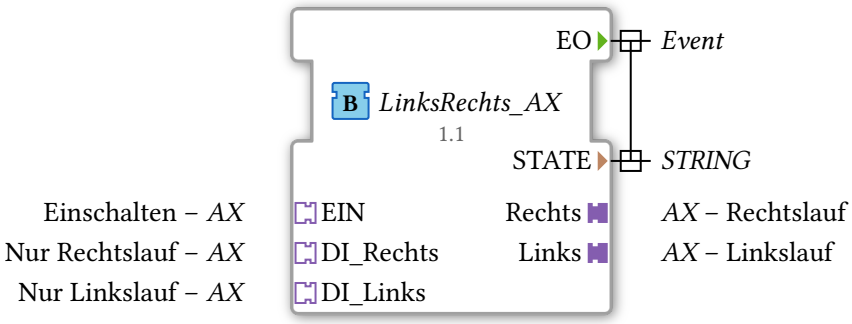
Hinweis: Ein Bild des Funktionsbausteins ist hier nicht verfügbar.

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsbaustein **LinksRechts_AX** (aus dem Paket `logiBUS::utils::sequence::verteiler`) dient der Steuerung eines alternierenden Prozesses mit zwei Richtungen (Rechtslauf und Linkslauf). Er wurde konzipiert, um zwischen zwei Ausgängen hin- und herzuschalten, wobei Pausenzustände berücksichtigt werden.

Besonders hervorzuheben ist die Möglichkeit, den automatischen Wechsel durch digitale Eingänge (`DI_Rechts`, `DI_Links`) zu unterbinden, um einen Lauf in nur einer Richtung zu erzwingen. Der Baustein nutzt hierfür die `AX` Adapter-Schnittstelle.



Schnittstellenstruktur

Der Baustein verwendet primär Adapter für die Kommunikation, stellt jedoch auch einen Status-String als direkten Ausgang zur Verfügung.

Ereignis-Eingänge

Es sind keine direkten Ereignis-Eingänge vorhanden. Die Steuerung erfolgt über die Ereignisse der Adapter-Sockets.

Ereignis-Ausgänge

Name	Typ	Beschreibung
EO	Event	Ereignis, das ausgelöst wird, wenn sich der interne Zustand (`STATE`) ändert.

Daten-Eingänge

Es sind keine direkten Daten-Eingänge vorhanden. Die Daten werden über die Adapter-Sockets eingelesen.

Daten-Ausgänge

Name	Typ	Beschreibung
STATE	STRING	Aktueller Zustand des Bausteins (z.B. „Rechtslauf“, „Linkslauf_Pause“).

Adapter

Sockets (Eingangs-Schnittstellen):

Name	Typ	Kommentar
EIN	adapter::types::unidirectional::AX	**Einschalten**: Das Hauptsignal zum Starten und Stoppen der Bewegung.
DI_Rechts	adapter::types::unidirectional::AX	**Nur Rechtslauf**: Wenn aktiv, wird ein Wechsel auf Linkslauf verhindert und der Rechtslauf erzwungen.
DI_Links	adapter::types::unidirectional::AX	**Nur Linkslauf**: Wenn aktiv, wird ein Wechsel auf Rechtslauf verhindert und der Linkslauf erzwungen.

Plugs (Ausgangs-Schnittstellen):

Name	Typ	Kommentar
Rechts	adapter::types::unidirectional::AX	**Rechtslauf**: Ausgangssignal für die Rechtsbewegung.
Links	adapter::types::unidirectional::AX	**Linkslauf**: Ausgangssignal für die Linksbewegung.

Funktionsweise

Der **LinksRechts_AX** Baustein implementiert eine Zustandsmaschine, die grundsätzlich zwischen einer Rechts- und einer Linksbewegung alterniert, getrennt durch Pausenphasen.

1. **Start:** Aus dem START-Zustand heraus entscheidet der Baustein basierend auf den Eingängen EIN und DI_Links, ob zuerst in den Linkslauf oder Rechtslauf geschaltet wird.
2. **Aktivierung (Lauf):** Solange das Signal EIN.D1 (Daten) zusammen mit einem Event EIN.E1 als TRUE anliegt, geht der Baustein in einen aktiven Zustand (Rechtslauf oder Linkslauf). Dabei wird der entsprechende Ausgangsadapter (Rechts oder Links) auf TRUE gesetzt.
3. **Deaktivierung (Pause):** Wird EIN.D1 zu FALSE (Ausschalten), wechselt der Baustein in den entsprechenden Pausenzustand (Rechtslauf_Pause oder Linkslauf_Pause). Die Ausgänge werden deaktiviert (FALSE).
4. **Wechsel-Logik (Alternieren):**
   - Befindet sich der Baustein in Rechtslauf_Pause und wird wieder eingeschaltet (EIN = TRUE), wechselt er standardmäßig in den **Linkslauf**.
   - Befindet sich der Baustein in Linkslauf_Pause und wird wieder eingeschaltet, wechselt er standardmäßig in den **Rechtslauf**.
5. **Override-Logik (Erzwingen):**
   - Ist im Zustand Rechtslauf_Pause der Eingang DI_Rechts aktiv, wird der Wechsel zum Linkslauf unterbunden und erneut der **Rechtslauf** gestartet.
   - Ist im Zustand Linkslauf_Pause der Eingang DI_Links aktiv, wird der Wechsel zum Rechtslauf unterbunden und erneut der **Linkslauf** gestartet.

Technische Besonderheiten

- **AX-Adapter:** Der Baustein nutzt den generischen unidirectional::AX Typ. Dieser kombiniert typischerweise ein boolesches Datensignal (D1) mit einem Ereignis (E1).
- **Priorisierung:** Laut der internen Beschreibung ist „Nur Rechtslauf vorrangig vor Nur Linkslauf“, was sich in den Startbedingungen widerspiegelt, jedoch wird die Sequenzlogik primär durch den vorherigen Zustand (History) bestimmt.
- **Status-Reporting:** Jede Zustandsänderung aktualisiert die STATE-Variable und feuert das EO-Event. Die Zustandsnamen werden über eine externe Enumeration (STATES::...) bezogen.

Zustandsübersicht

Die ECC (Execution Control Chart) definiert folgende Zustände:

Zustand Name	Aktion	Beschreibung
START	-	Initialzustand. Wartet auf das EIN-Signal.
Rechtslauf	Set_Rechts_TRUE, Set_Links_FALSE, Status update	Aktiviert den Adapter Rechts, deaktiviert Links.
Rechtslauf_Pause	Set_Rechts_FALSE, Set_Links_FALSE, Status update	Beide Ausgänge aus. System merkt sich, dass es zuletzt rechts war.
Linkslauf	Set_Rechts_FALSE, Set_Links_TRUE, Status update	Aktiviert den Adapter Links, deaktiviert Rechts.
Linkslauf_Pause	Set_Rechts_FALSE, Set_Links_FALSE, Status update	Beide Ausgänge aus. System merkt sich, dass es zuletzt links war.

Anwendungsszenarien

- **Pendelbetrieb:** Automatische Steuerung von Mechanismen, die sich hin- und herbewegen müssen (z.B. ein Scheibenwischer-Modus oder ein Reinigungskopf), gesteuert durch einen einzigen Taster (EIN).
- **Beregnungsanlagen:** Sequentielle Ansteuerung von zwei Sektoren (Sektor Rechts -> Pause -> Sektor Links -> Pause), wobei bei Bedarf ein Sektor mehrfach hintereinander aktiviert werden kann (mittels DI_Rechts/DI_Links).
- **Reversiermotor:** Steuerung eines Motors, der bei jedem Neustart die Drehrichtung ändern soll, sofern nicht anders vorgegeben.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Einfaches Toggle (FlipFlop):** Ein Standard-Toggle schaltet nur einen Ausgang Ein/Aus. LinksRechts_AX schaltet zwischen *zwei* Ausgängen um.

- **RS-Glied:** Ein RS-Glied speichert nur einen Zustand basierend auf Set/Reset. Dieser Baustein beinhaltet eine Sequenzlogik (History-Memory), da er weiß, welcher Zustand *vor* der Pause aktiv war.
- **E_SELECT:** Ähnelt einem Selektor, aber `LinksRechts_AX` beinhaltet die zeitliche Komponente des „Pausierens“ und des automatischen Wechsels beim nächsten Startsignal.

Fazit

Der Funktionsbaustein **LinksRechts_AX** ist eine spezialisierte Komponente für Ablaufsteuerungen, die einen alternierenden Betrieb zwischen zwei Ausgängen erfordern. Durch die Integration der Adapter-Technologie (`AX`) und die Möglichkeit, die Sequenz durch digitale Eingänge (`DI`) zu beeinflussen, bietet er eine flexible Lösung für sequentielle Steuerungsaufgaben mit Richtungspriorisierung.

STATES

• – ◦ • – ◦ • – ◦ •

Einleitung

Die globale Konstanten-Deklaration `STATES` definiert eine Sammlung von vordefinierten Zustandsbezeichnungen als Zeichenketten (STRING). Diese Konstanten werden typischerweise verwendet, um Betriebszustände in einer Steuerungssequenz zu benennen, insbesondere für Prozesse mit wechselnder Richtung (Linkslauf/Rechtslauf) und integrierten Pausen. Sie dient als zentrale und konsistente Quelle für diese Zustandsnamen innerhalb eines Projekts.

Schnittstellenstruktur

Bei globalen Konstanten handelt es sich nicht um einen Funktionsbaustein mit Ereignis- oder Datenschnittstellen. Stattdessen werden Konstantenwerte deklariert, die projektweit zur Verfügung stehen.

Daten-Deklarationen

Die folgenden Konstanten werden definiert:

- `Rechtslauf`: Zeichenkette mit dem Wert `'Rechtslauf'`
- `Linkslauf`: Zeichenkette mit dem Wert `'Linkslauf'`
- `Linkslauf_Pause`: Zeichenkette mit dem Wert `'Linkslauf_Pause'`
- `Rechtslauf_Pause`: Zeichenkette mit dem Wert `'Rechtslauf_Pause'`

Funktionsweise

Die Konstanten selbst besitzen keine aktive Funktionalität. Sie stellen lediglich benannte Werte bereit. Ihr Hauptzweck ist die Vermeidung von „Magic Numbers“ oder hartkodierten Zeichenketten im Programmcode. Durch die Verwendung dieser Konstanten wird der Code lesbarer, wartbarer und weniger fehleranfällig, da Änderungen an den Zustandsnamen nur an einer zentralen Stelle vorgenommen werden müssen.

Technische Besonderheiten

- **Typ:** Alle deklarierten Konstanten sind vom Datentyp `STRING`.
- **Gültigkeitsbereich:** Als globale Konstanten sind sie innerhalb des gesamten Namensraums `logiBUS::utils::sequence::verteiler::LinksRechts` und in davon abhängigen Komponenten verfügbar.
- **Initialisierung:** Die Werte werden direkt bei der Deklaration mit dem `InitialValue`-Attribut festgelegt.

Zustandsübersicht

Nicht zutreffend, da es sich um eine statische Datendeklaration und nicht um einen zustandsbehafteten Baustein handelt.

Anwendungsszenarien

Typische Anwendungen für diese Konstanten sind:

- **Zustandsautomaten (State Machines):** Die Konstanten können als eindeutige Bezeichner für die verschiedenen Zustände eines Automaten dienen, der einen Prozess mit Richtungswechsel und Pausen steuert (z. B. einen Verteilertisch).
- **Dokumentation und Logging:** Sie können verwendet werden, um den aktuellen Betriebszustand in Meldungen, Logdateien oder auf Bedienpanels anzuzeigen.
- **Vergleiche und Schaltlogik:** In der Programmlogik können Variablen oder Ausgänge mit diesen Konstanten verglichen werden, um Entscheidungen basierend auf dem aktuellen Zustand zu treffen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Globale Variablen (VAR_GLOBAL):** Im Gegensatz zu globalen Variablen können die Werte von Konstanten (`CONSTANT`) zur Laufzeit nicht verändert werden.
- **Enumerations (ENUM):** Während ENUM-Datentypen in einigen Programmiersprachen ähnliche Zwecke erfüllen, bietet die IEC 61499-Standardbibliothek keinen eingebauten ENUM-Typ. Die Verwendung von STRING-Konstanten ist eine gängige Methode, um semantisch benannte Werte in dieser Umgebung zu definieren.
- **Funktionsbaustein-internen Konstanten:** Konstanten können auch innerhalb von Funktionsbausteinen definiert werden. Globale Konstanten sind jedoch über Bausteingrenzen hinweg sichtbar und fördern so die Wiederverwendbarkeit und Konsistenz über ein gesamtes Projekt hinweg.

Fazit

Die globale Konstanten-Deklaration `STATES` ist ein einfaches aber nützliches Konstrukt zur Strukturierung von Steuerungssoftware. Sie erhöht die Codequalität, indem sie zentrale und unveränderliche Zustandsbezeichnungen bereitstellt, die speziell für Sequenzen mit Links-/Rechtslauf und Pausenzuständen zugeschnitten sind. Ihre Verwendung macht Programme verständlicher und erleichtert zukünftige Anpassungen.

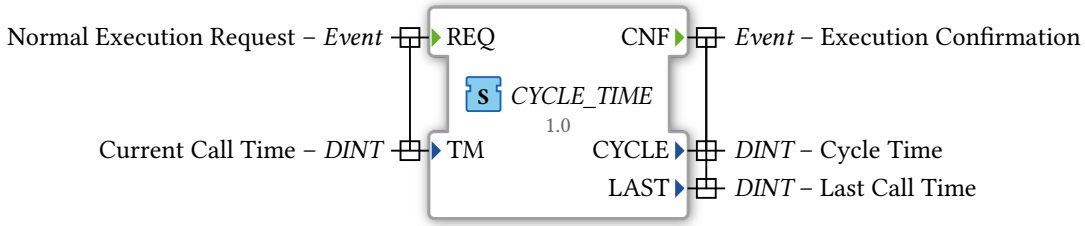
timing

CYCLE_TIME

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `CYCLE_TIME` dient zur Berechnung der verstrichenen Zeit zwischen zwei aufeinanderfolgenden Aufrufen. Er ist ein nützliches Werkzeug zur Überwachung und Diagnose von Zykluszeiten in Echtzeitanwendungen, beispielsweise zur Leistungsanalyse oder zur Erkennung von unerwarteten Verzögerungen in einem Steuerungssystem.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Berechnung der Zykluszeit. Bei diesem Ereignis wird der Algorithmus ausgeführt.

Ereignis-Ausgänge

- **CNF**: Wird ausgelöst, sobald die Berechnung abgeschlossen ist. Dieses Ereignis bestätigt die Ausführung und liefert die berechneten Werte an die Datenausgänge.

Daten-Eingänge

- **TM** (DINT): Stellt den aktuellen Zeitstempel zum Zeitpunkt des Aufrufs bereit. Dieser Wert wird typischerweise von einer Systemuhr oder einem Timer-Baustein geliefert. Initialwert ist `0`.

Daten-Ausgänge

- **CYCLE** (DINT): Enthält die berechnete Zykluszeit, also die Differenz zwischen dem aktuellen (`TM`) und dem vorherigen Zeitstempel (`LAST`). Initialwert ist `0`.
- **LAST** (DINT): Speichert den Zeitstempel des letzten Aufrufs (`TM`). Dieser Wert wird für die nächste Berechnung verwendet. Initialwert ist `0`.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Bei jedem Eintreffen des `REQ`-Ereignisses wird der zugehörige Algorithmus `REQ` ausgeführt. Dieser Algorithmus führt zwei wesentliche Operationen durch:

1. **Berechnung der Zykluszeit**: Die Differenz zwischen dem neu eingehenden Zeitwert `TM` und dem gespeicherten Wert `LAST` (vom vorherigen Aufruf) wird berechnet und an den Ausgang `CYCLE` ausgegeben.
2. **Aktualisierung des Zeitstempels**: Der aktuelle Eingangswert `TM` wird im internen Speicher `LAST` für den nächsten Berechnungszyklus gesichert.

Unmittelbar nach der Ausführung des Algorithmus wird das `CNF`-Ereignis mit den aktualisierten Werten an `CYCLE` und `LAST` ausgelöst.

Technische Besonderheiten

- Der Baustein ist ein **Simple Function Block (Basic FB)** gemäß IEC 61499.
- Die verwendeten Datentypen (`DINT`) sind vorzeichenbehaftete 32-Bit-Integer, die für die Darstellung großer Zeitwerte (z.B. in Millisekunden oder Mikrosekunden) geeignet sind.
- Die Initialwerte von `TM`, `CYCLE` und `LAST` sind auf `0` gesetzt. Der erste gültige `CYCLE`-Wert nach dem Start ergibt sich daher aus der Differenz `TM - 0`. Dies sollte bei der Initialisierungslogik der Anwendung berücksichtigt werden.

Zustandsübersicht

Der Baustein besitzt einen einzigen Execution Control State (EC-State) mit dem Namen `REQ`. Bei Eintritt in diesen Zustand wird stets der Algorithmus `REQ` ausgeführt und anschließend das `CNF`-Ereignis produziert.

Anwendungsszenarien

- **Zykluszeitüberwachung**: Kontinuierliche Messung der Ausführungszeit eines Programms oder eines bestimmten Programmteils.
- **Performance-Monitoring**: Identifikation von Engpässen oder ungewöhnlich langen Zykluszeiten in einer Steuerungsapplikation.
- **Diagnose**: Protokollierung von Zykluszeiten zu Analysezwecken oder zur Erstellung von Systemlaufzeitstatistiken.
- **Adaptive Steuerung**: Bereitstellung der aktuellen Zykluszeit für Algorithmen, deren Verhalten von der verfügbaren Rechenzeit abhängen soll.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu generischen Zeitmessbausteinen (wie `E_DELAY` oder `E_CYCLE`) ist `CYCLE_TIME` spezialisiert auf die **Differenzbildung zwischen aufeinanderfolgenden Zeitpunkten**. Er führt keinen eigenen Timer, sondern wertet extern bereitgestellte Zeitstempel aus. Bausteine wie `E_SR` oder `E_R_TRIG` dienen grundlegend anderen Zwecken (Speichern, Flankenerkennung) und bieten keine Zeitdifferenzberechnung.

🛠️ Zugehörige Übungen

- Uebung_072c

Fazit

Der `CYCLE_TIME`-Funktionsblock ist ein einfaches, aber effektives Werkzeug zur Laufzeitmessung in IEC 61499-basierten Steuerungssystemen. Seine klare Schnittstelle und einfache Funktionsweise machen ihn zu einer robusten Komponente für Diagnose- und Monitoring-Aufgaben, wo die Genauigkeit und Zuverlässigkeit der Zykluszeiterfassung im Vordergrund steht.

2.1.14.13 version

🎧 Podcast

- „Store Version“ – Dein Schlüssel zur Verwaltung von Objektdatenpools im nichtflüchtigen VT-Speicher (ISO 11783-6)
- VT-Versionen verstehen: Dein Weg zum Meister der Land- und Baumaschinenmechatronik

fboot

version

🎧 Podcast

- „Store Version“ – Dein Schlüssel zur Verwaltung von Objektdatenpools im nichtflüchtigen VT-Speicher (ISO 11783-6)
- VT-Versionen verstehen: Dein Weg zum Meister der Land- und Baumaschinenmechatronik

Einleitung

Der Funktionsblock `version` dient der einfachen Weitergabe einer Versionsinformation. Er empfängt eine Zeichenkette (String) als Eingabe und gibt diese unverändert an seinem Ausgang wieder. Dieser Baustein ist nützlich, um Versionsnummern oder andere Kennzeichnungen innerhalb einer Anwendungskette zu transportieren oder für Diagnosezwecke bereitzustellen.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Dieses Ereignis löst die Verarbeitung des Funktionsblocks aus. Bei seinem Eintreffen wird der Wert des zugehörigen Dateneingangs `VERSION` an den Datenausgang `VERSIONO` kopiert.

Ereignis-Ausgänge

- **CNF**: Dieses Ereignis signalisiert die erfolgreiche Beendigung der Operation. Es wird gleichzeitig mit der Datenausgabe `VERSIONO` erzeugt.

Daten-Eingänge

- **VERSION** (STRING): Dieser Eingang erwartet die Zeichenkette, die weitergegeben werden soll (z.B. eine Versionsnummer wie „1.2.3“).

Daten-Ausgänge

- **VERSIONO** (STRING): Dieser Ausgang gibt den empfangenen Wert von `VERSION` unverändert aus.

Adapter

Dieser Funktionsblock verfügt über keine Adapterschnittstellen.

Funktionsweise

Die Funktionsweise des `version`-Blocks ist äußerst einfach und deterministisch. Bei jedem Eintreffen des Ereignisses `REQ` wird der aktuelle Wert der Eingangsvariablen `VERSION` in die Ausgangsvariable `VERSIONO` kopiert. Unmittelbar nach dieser Zuweisung wird das Bestätigungsereignis `CNF` zusammen mit dem neuen Ausgabewert erzeugt. Es findet keine Formatprüfung, Manipulation oder Speicherung des Wertes statt.

Technische Besonderheiten

- **Einfache Durchreichung**: Der Block führt keine Transformation oder Logik auf den Daten durch.
- **Direkte Kopplung**: Die Ereignisse `REQ` und `CNF` sind direkt gekoppelt; die Ausführungszeit ist vernachlässigbar.

- **Kein interner Zustand**: Der Block besitzt keinen eigenen Speicher. Jede Ausführung ist ausschließlich von den aktuellen Eingängen abhängig.

Zustandsübersicht

Der Funktionsblock `version` ist zustandslos. Er verharrt in einem einzigen, passiven Zustand, bis ein `REQ`-Ereignis eintrifft. Die Reaktion darauf (Kopieren der Daten und Auslösen von `CNF`) ist instantan und führt nicht zu einem anderen Zustand.

Anwendungsszenarien

1. **Versionsweitergabe**: In einer Anwendungskette kann ein übergeordneter Baustein (z.B. ein Master- oder Verwaltungs-FB) seine eigene Version an diesen Block übergeben, um sie an nachgelagerte Diagnose- oder Logging-Funktionen zu senden.
2. **Diagnose und Tracing**: Der Block kann verwendet werden, um bestimmte Software-Stände oder Konfigurationsversionen in einem zentralen Log- oder Anzeigesystem zu sammeln.
3. **Datenkonsolidierung**: In Verbindung mit anderen Funktionsblöcken kann er dazu dienen, Versionsinformationen aus verschiedenen Quellen an einer zentralen Stelle (z.B. einem HMI oder einem Netzwerkdienst) bereitzustellen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`E_SR` oder `E_RS` (Flip-Flops)**: Diese Blöcke besitzen einen internen Zustand (gesetzt/rückgesetzt), der durch Ereignisse geändert wird. Der `version`-Block hat keinen solchen Zustand.
- **`E_MERGE`**: Dieser Block fusioniert Ereignisse von mehreren Eingängen. Der `version`-Block hat nur einen einzigen Ereigniseingang und verarbeitet keine Ereignisströme.
- **Einfache Daten-Blöcke (z.B. `E_R_TRIG`)**: Diese reagieren auf Signalflanken oder führen Timing-Operationen durch. Der `version`-Block führt ausschließlich eine Datenkopie durch.
- **Generische Blöcke wie `COPY`**: Ein `COPY`-Block würde eine ähnliche Funktion erfüllen (Daten von A nach B kopieren), ist aber typischerweise ein reiner Datenbaustein ohne Ereignissteuerung. Der `version`-Block folgt dem standardmäßigen IEC 61499-Ereignis-/Datenfluss-Paradigma.

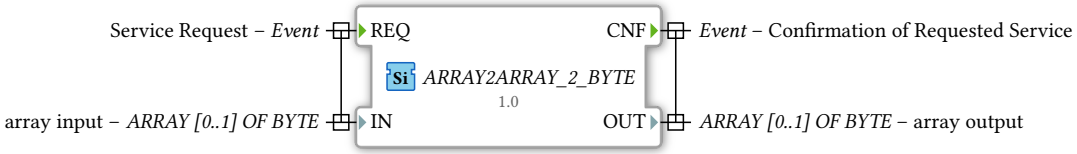
Fazit

Der `version`-Funktionsblock ist ein minimalistisches und spezialisiertes Werkzeug zur Weitergabe von String-Informationen innerhalb eines ereignisgesteuerten Steuerungssystems. Seine Stärke liegt in der Einfachheit und Klarheit seines Zwecks. Für reine Datenkopien ohne Ereignislogik könnten alternative Implementierungen in Frage kommen, für die Integration in eine standardkonforme IEC 61499-Applikation mit klar definiertem Ereignisfluss ist er jedoch gut geeignet.

2.2 Standard Libraries

2.2.1 convert

2.2.1.1 ARRAY2ARRAY_2_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ARRAY2ARRAY_2_BYTE` ist ein Service-Interface-Funktionsblock, der ein `BYTE`-Array der Größe 2 unverändert von `IN` nach `OUT` kopiert. Er gehört zur generischen `GEN_ARRAY2ARRAY`-Familie (vgl. ARRAY2ARRAY_2_LREAL) und dient als reine Typ-/Schnittstellenbrücke zwischen zwei `BYTE`-Arrays gleicher Größe.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst den Kopiervorgang aus, trägt `IN`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT`.

Daten-Eingänge

- **IN** (`BYTE`, Array-Größe 2): Das Quell-Array (8-Bit-Bitmuster).

Daten-Ausgänge

- **OUT** (`BYTE`, Array-Größe 2): Das Ziel-Array, identische Werte wie `IN`.

Funktionsweise

Bei Eintreffen von `REQ` werden die 2 Elemente von `IN` unverändert nach `OUT` kopiert, anschließend wird `CNF` ausgelöst. Es findet keine Typumwandlung oder Wertänderung statt — der Baustein dient ausschließlich der Entkopplung zweier Array-Variablen mit identischem Typ und identischer Größe.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY'`, dieselbe C++-Basis wie alle anderen `ARRAY2ARRAY_*`-Varianten (z. B. ARRAY2ARRAY_2_LREAL), lediglich Array-Größe und Datentyp unterscheiden sich pro Instanziierung.
- **Feste Größe 2**: Für andere Array-Größen desselben Typs siehe `ARRAY2ARRAY_8_BYTE`.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur vollständigen Kopie und zu `CNF`.

Anwendungsszenarien

- **Schnittstellenentkopplung**: Ein Baustein liefert ein `BYTE`-Array der Größe 2 als Ausgabe, ein anderer erwartet ein separates Eingangs-Array — `ARRAY2ARRAY_2_BYTE` dient als Brücke ohne Seiteneffekte.
- **Datenrouting** von `BYTE`-Arrays fester Größe zwischen Teilen einer 4diac-Applikation.

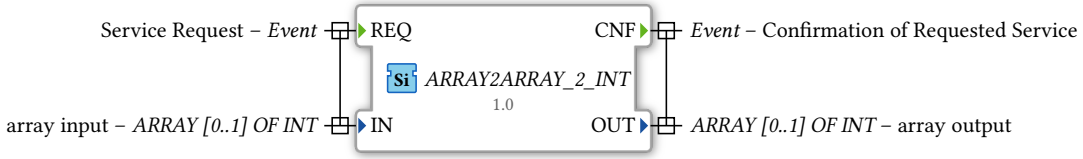
⚖️ Vergleich mit ähnlichen Bausteinen

- **ARRAY2ARRAY_2_LREAL**: dieselbe generische Implementierung für den Datentyp `LREAL`.
- **`ARRAY2ARRAY_8_BYTE`**: dieselbe Implementierung mit Array-Größe 8 statt 2.
- **`ARRAY2VALUES_2_BYTE`**: zerlegt das Array stattdessen in 2 einzelne Skalarvariablen.

Fazit

`ARRAY2ARRAY_2_BYTE` liefert eine einfache, generisch implementierte Eins-zu-eins-Kopie eines `BYTE`-Arrays der Größe 2 und dient als unkomplizierte Schnittstellenbrücke zwischen gleichartigen Array-Variablen.

2.2.1.2 ARRAY2ARRAY_2_INT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ARRAY2ARRAY_2_INT` ist ein Service-Interface-Funktionsblock, der ein `INT`-Array der Größe 2 unverändert von `IN` nach `OUT` kopiert. Er gehört zur generischen `GEN_ARRAY2ARRAY`-Familie (vgl. ARRAY2ARRAY_2_LREAL) und dient als reine Typ-/Schnittstellenbrücke zwischen zwei `INT`-Arrays gleicher Größe.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst den Kopiervorgang aus, trägt `IN`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT`.

Daten-Eingänge

- **IN** (`INT`, Array-Größe 2): Das Quell-Array (16-Bit-Ganzzahl (vorzeichenbehaftet)).

Daten-Ausgänge

- **OUT** (`INT`, Array-Größe 2): Das Ziel-Array, identische Werte wie `IN`.

Funktionsweise

Bei Eintreffen von `REQ` werden die 2 Elemente von `IN` unverändert nach `OUT` kopiert, anschließend wird `CNF` ausgelöst. Es findet keine Typumwandlung oder Wertänderung statt — der Baustein dient ausschließlich der Entkopplung zweier Array-Variablen mit identischem Typ und identischer Größe.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY'`, dieselbe C++-Basis wie alle anderen `ARRAY2ARRAY_*`-Varianten (z. B. ARRAY2ARRAY_2_LREAL), lediglich Array-Größe und Datentyp unterscheiden sich pro Instanziierung.
- **Feste Größe 2**: Für andere Array-Größen desselben Typs siehe `ARRAY2ARRAY_8_INT`.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur vollständigen Kopie und zu `CNF`.

Anwendungsszenarien

- **Schnittstellenentkopplung**: Ein Baustein liefert ein `INT`-Array der Größe 2 als Ausgabe, ein anderer erwartet ein separates Eingangs-Array — `ARRAY2ARRAY_2_INT` dient als Brücke ohne Seiteneffekte.
- **Datenrouting** von `INT`-Arrays fester Größe zwischen Teilen einer 4diac-Applikation.

⚖️ Vergleich mit ähnlichen Bausteinen

- **ARRAY2ARRAY_2_LREAL**: dieselbe generische Implementierung für den Datentyp `LREAL`.
- **`ARRAY2ARRAY_8_INT`**: dieselbe Implementierung mit Array-Größe 8 statt 2.
- **`ARRAY2VALUES_2_INT`**: zerlegt das Array stattdessen in 2 einzelne Skalarvariablen.

Fazit

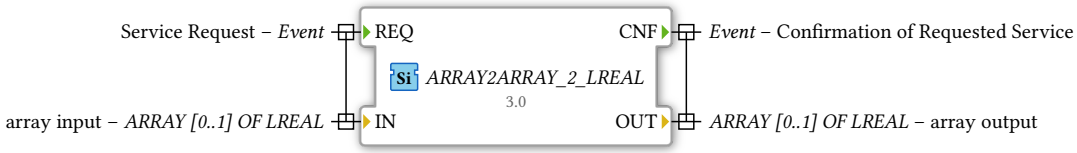
`ARRAY2ARRAY_2_INT` liefert eine einfache, generisch implementierte Eins-zu-eins-Kopie eines `INT`-Arrays der Größe 2 und dient als unkomplizierte Schnittstellenbrücke zwischen gleichartigen Array-Variablen.

2.2.1.3 ARRAY2ARRAY_2_LREAL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock (FB) `ARRAY2ARRAY_2_LREAL` ist ein Service-Interface-Funktionsblock, der darauf ausgelegt ist, Werte von einem Eingangs-Array auf ein Ausgangs-Array zu kopieren. Er dient als einfache und ereignisgesteuerte Schnittstelle für die Übertragung von Daten zwischen Arrays innerhalb einer 4diac-Anwendung.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Event): Löst den Kopierprozess aus. Dieses Ereignis signalisiert dem Funktionsblock, die Datenübertragung zu starten.
  - Verbunden mit Datenvariable: `IN`

Ereignis-Ausgänge

- **CNF** (Event): Bestätigt den Abschluss des Kopierprozesses. Dieses Ereignis wird nach erfolgreicher Datenübertragung ausgelöst.
  - Verbunden mit Datenvariable: `OUT`

Daten-Eingänge

- **IN** (Typ: `LREAL`, Array-Größe: `2`): Das Quell-Array, dessen Werte kopiert werden sollen. Es handelt sich um ein Array von Langen Reellen Zahlen (Double-Precision Floating Point).

Daten-Ausgänge

- **OUT** (Typ: `LREAL`, Array-Größe: `2`): Das Ziel-Array, in das die kopierten Werte geschrieben werden. Ebenfalls ein Array von Langen Reellen Zahlen.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock `ARRAY2ARRAY_2_LREAL` arbeitet ereignisgesteuert. Sobald das Eingangsereignis `REQ` empfangen wird, liest der Funktionsblock die aktuellen Werte aus dem Daten-Eingangs-Array `IN`. Diese Werte werden dann direkt in das Daten-Ausgangs-Array `OUT` kopiert. Nach Abschluss des Kopiervorgangs wird das Ausgangsereignis `CNF` ausgelöst, um den erfolgreichen Abschluss der Operation zu signalisieren. Der Block verarbeitet somit eine Eins-zu-Eins-Kopie der Array-Elemente.

Technische Besonderheiten

- **Service-Interface:** Der Baustein ist als „Service Interface Function Block Type“ konzipiert, was auf eine Standard-Schnittstelle für spezifische Dienste hinweist.
- **Generische Implementierung:** Intern verwendet dieser Funktionsblock wahrscheinlich eine generische Implementierung, wie durch den Attributwert `'GEN_ARRAY2ARRAY'` angedeutet. Dies ermöglicht eine flexible Erstellung ähnlicher Bausteine für verschiedene Array-Größen und Datentypen.
- **Feste Array-Größe:** Die Array-Größe für `IN` und `OUT` ist fest auf `2` definiert, was den Anwendungsbereich auf spezifische Szenarien mit dieser Array-Dimension begrenzt.
- **Datentyp `LREAL`:** Der Baustein ist auf den Datentyp `LREAL` (Lange Reelle Zahl) spezialisiert, was ihn ideal für die Verarbeitung von hochpräzisen Gleitkommazahlen macht.

Zustandsübersicht

Der Funktionsblock befindet sich typischerweise in zwei Hauptzuständen:

1. **Warten auf Anfrage:** Der Block wartet auf das `REQ`-Ereignis. In diesem Zustand ist er passiv und führt keine Operationen aus.
2. **Verarbeitung der Anfrage:** Nach Empfang von `REQ` wechselt der Block in diesen Zustand, führt den Kopierprozess von `IN` nach `OUT` aus und emittiert anschließend `CNF`, bevor er in den Zustand „Warten auf Anfrage“ zurückkehrt.

Anwendungsszenarien

- **Datenrouting:** Weiterleitung von `LREAL`-Datenpaketen der Größe 2 zwischen verschiedenen Teilen eines Steuerungssystems.
- **Datenpufferung:** Kurzfristiges Speichern oder Übertragen von Messwerten oder Sollwerten.
- **Schnittstellenanpassung:** Wenn ein Baustein ein `LREAL`-Array der Größe 2 als Ausgabe liefert und ein anderer Baustein dasselbe Array als Eingabe benötigt, kann dieser FB als Brücke dienen.
- **Konstante Datenweitergabe:** Übertragen von statischen oder selten aktualisierten `LREAL`-Arrays von einem Punkt zu einem anderen.

⚖️ Vergleich mit ähnlichen Bausteinen

Dieser Baustein ist sehr spezifisch in seiner Funktion. Im Vergleich zu allgemeineren Datenmanipulationsbausteinen, die oft flexible Datentypen oder Array-Größen über Konfigurationsparameter ermöglichen, ist `ARRAY2ARRAY_2_LREAL` fest auf `LREAL`-Arrays der Größe 2 spezialisiert. Existierten Bausteine für generische Array-Kopien (z.B. `ARRAY_COPY` mit konfigurierbarer Größe und Typ), wäre dieser Baustein eine spezialisierte Instanz davon. Er ist einfacher als solche generischen Bausteine, aber weniger flexibel. Für seine spezifische Aufgabe bietet er jedoch eine klare und effiziente Lösung.

Metadaten

Attribut	Wert
Copyright	(c) 2014 Profactor GmbH
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner), 1.0 (2014-08-05, Matthias Plasch)
4diac-Paket	eclipse4diac::convert

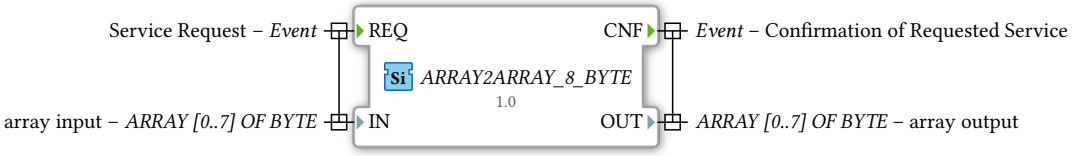
Fazit

Der `ARRAY2ARRAY_2_LREAL` Funktionsblock bietet eine unkomplizierte und ereignisgesteuerte Methode, um `LREAL`-Arrays der Größe 2 zu kopieren. Seine Spezialisierung auf diesen Datentyp und diese Array-Größe macht ihn zu einem nützlichen Werkzeug für definierte Anwendungsfälle, bei denen diese genaue Datenstruktur erforderlich ist. Er trägt zur Modularität und Lesbarkeit von 4diac-Applikationen bei, indem er eine explizite Funktion für diese Art der Datenübertragung bereitstellt.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.2.1.4 ARRAY2ARRAY_8_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ARRAY2ARRAY_8_BYTE` ist ein Service-Interface-Funktionsblock, der ein `BYTE`-Array der Größe 8 unverändert von `IN` nach `OUT` kopiert. Er gehört zur generischen `GEN_ARRAY2ARRAY`-Familie (vgl. ARRAY2ARRAY_2_LREAL) und dient als reine Typ-/Schnittstellenbrücke zwischen zwei `BYTE`-Arrays gleicher Größe.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst den Kopiervorgang aus, trägt `IN`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT`.

Daten-Eingänge

- **IN** (`BYTE`, Array-Größe 8): Das Quell-Array (8-Bit-Bitmuster).

Daten-Ausgänge

- **OUT** (`BYTE`, Array-Größe 8): Das Ziel-Array, identische Werte wie `IN`.

Funktionsweise

Bei Eintreffen von REQ werden die 8 Elemente von IN unverändert nach OUT kopiert, anschließend wird CNF ausgelöst. Es findet keine Typumwandlung oder Wertänderung statt — der Baustein dient ausschließlich der Entkopplung zweier Array-Variablen mit identischem Typ und identischer Größe.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY'`, dieselbe C++-Basis wie alle anderen ARRAY2ARRAY_*-Varianten (z. B. ARRAY2ARRAY_2_LREAL), lediglich Array-Größe und Datentyp unterscheiden sich pro Instanziierung.
- **Feste Größe 8**: Für andere Array-Größen desselben Typs siehe ARRAY2ARRAY_2_BYTE.

Zustandsübersicht

Zustandslos: jedes REQ führt unmittelbar zur vollständigen Kopie und zu CNF.

Anwendungsszenarien

- **Schnittstellenentkopplung**: Ein Baustein liefert ein BYTE-Array der Größe 8 als Ausgabe, ein anderer erwartet ein separates Eingangs-Array — ARRAY2ARRAY_8_BYTE dient als Brücke ohne Seiteneffekte.
- **Datenrouting** von BYTE-Arrays fester Größe zwischen Teilen einer 4diac-Applikation.

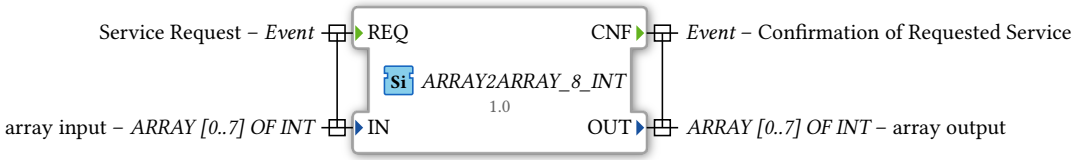
⚖️ Vergleich mit ähnlichen Bausteinen

- **ARRAY2ARRAY_2_LREAL**: dieselbe generische Implementierung für den Datentyp LREAL.
- **ARRAY2ARRAY_2_BYTE**: dieselbe Implementierung mit Array-Größe 2 statt 8.
- **ARRAY2VALUES_8_BYTE**: zerlegt das Array stattdessen in 8 einzelne Skalarvariablen.

Fazit

ARRAY2ARRAY_8_BYTE liefert eine einfache, generisch implementierte Eins-zu-eins-Kopie eines BYTE-Arrays der Größe 8 und dient als unkomplizierte Schnittstellenbrücke zwischen gleichartigen Array-Variablen.

2.2.1.5 ARRAY2ARRAY_8_INT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der ARRAY2ARRAY_8_INT ist ein Service-Interface-Funktionsblock, der ein INT-Array der Größe 8 unverändert von IN nach OUT kopiert. Er gehört zur generischen GEN_ARRAY2ARRAY-Familie (vgl. ARRAY2ARRAY_2_LREAL) und dient als reine Typ-/Schnittstellenbrücke zwischen zwei INT-Arrays gleicher Größe.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst den Kopiervorgang aus, trägt IN.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt OUT.

Daten-Eingänge

- **IN** (INT, Array-Größe 8): Das Quell-Array (16-Bit-Ganzzahl (vorzeichenbehaftet)).

Daten-Ausgänge

- **OUT** (INT, Array-Größe 8): Das Ziel-Array, identische Werte wie IN.

Funktionsweise

Bei Eintreffen von `REQ` werden die 8 Elemente von `IN` unverändert nach `OUT` kopiert, anschließend wird `CNF` ausgelöst. Es findet keine Typumwandlung oder Wertänderung statt — der Baustein dient ausschließlich der Entkopplung zweier Array-Variablen mit identischem Typ und identischer Größe.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY'`, dieselbe C++-Basis wie alle anderen `ARRAY2ARRAY_*`-Varianten (z. B. ARRAY2ARRAY_2_LREAL), lediglich Array-Größe und Datentyp unterscheiden sich pro Instanziierung.
- **Feste Größe 8**: Für andere Array-Größen desselben Typs siehe `ARRAY2ARRAY_2_INT`.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur vollständigen Kopie und zu `CNF`.

Anwendungsszenarien

- **Schnittstellenentkopplung**: Ein Baustein liefert ein `INT`-Array der Größe 8 als Ausgabe, ein anderer erwartet ein separates Eingangs-Array — `ARRAY2ARRAY_8_INT` dient als Brücke ohne Seiteneffekte.
- **Datenrouting** von `INT`-Arrays fester Größe zwischen Teilen einer 4diac-Applikation.

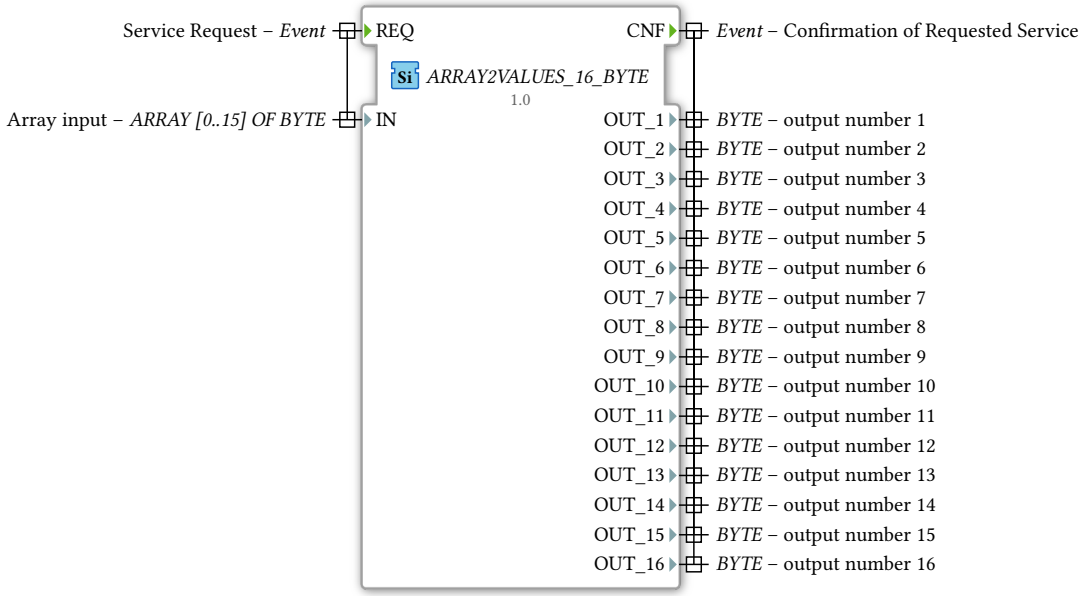
⚖️ Vergleich mit ähnlichen Bausteinen

- **ARRAY2ARRAY_2_LREAL**: dieselbe generische Implementierung für den Datentyp `LREAL`.
- **`ARRAY2ARRAY_2_INT`**: dieselbe Implementierung mit Array-Größe 2 statt 8.
- **`ARRAY2VALUES_8_INT`**: zerlegt das Array stattdessen in 8 einzelne Skalarvariablen.

Fazit

`ARRAY2ARRAY_8_INT` liefert eine einfache, generisch implementierte Eins-zu-eins-Kopie eines `INT`-Arrays der Größe 8 und dient als unkomplizierte Schnittstellenbrücke zwischen gleichartigen Array-Variablen.

2.2.1.6 ARRAY2VALUES_16_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ARRAY2VALUES_16_BYTE` zerlegt ein `BYTE`-Array der Größe 16 in 16 einzelne Skalarvariablen `OUT_1`…`OUT_16`. Er gehört zur generischen `GEN_ARRAY2ARRAY`-Familie (vgl. ARRAY2VALUES_2_LREAL) und dient der Umwandlung von Array-Daten in diskrete Einzelwerte.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Zerlegung aus, trägt `IN`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT_1`, `OUT_2`, `OUT_3`, `OUT_4`, `OUT_5`, `OUT_6`, `OUT_7`, `OUT_8`, `OUT_9`, `OUT_10`, `OUT_11`, `OUT_12`, `OUT_13`, `OUT_14`, `OUT_15`, `OUT_16`.

Daten-Eingänge

- **IN** (`BYTE`, Array-Größe 16): Das zu zerlegende Array (8-Bit-Bitmuster).

Daten-Ausgänge

- `OUT_1`, `OUT_2`, `OUT_3`, `OUT_4`, `OUT_5`, `OUT_6`, `OUT_7`, `OUT_8`, `OUT_9`, `OUT_10`, `OUT_11`, `OUT_12`, `OUT_13`, `OUT_14`, `OUT_15`, `OUT_16` (`BYTE`): `OUT_i` entspricht `IN[i-1]`, also dem i-ten Element des Arrays.

Funktionsweise

Bei Eintreffen von `REQ` wird jedes Element von `IN` auf den entsprechenden Skalarausgang `OUT_i` kopiert (`IN[0]` ⟶ `OUT_1`, ..., `IN[15]` ⟶ `OUT_16`), anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY'`, dieselbe C++-Basis wie ARRAY2VALUES_2_LREAL und die übrigen `ARRAY2VALUES_*`-Varianten.
- **Feste Größe 16**: Für andere Array-Größen desselben Typs siehe `ARRAY2VALUES_2_BYTE`, `ARRAY2VALUES_4_BYTE`, `ARRAY2VALUES_8_BYTE`, `ARRAY2VALUES_32_BYTE`.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur vollständigen Zerlegung und zu `CNF`.

Anwendungsszenarien

- **Datenaufbereitung**: Ein vorgeschalteter Baustein liefert ein `BYTE`-Array, ein nachgeschalteter Baustein benötigt jedoch diskrete Einzelvariablen.
- **Schnittstellenanpassung** zwischen array-basierten und variablenbasierten Bausteinschnittstellen.

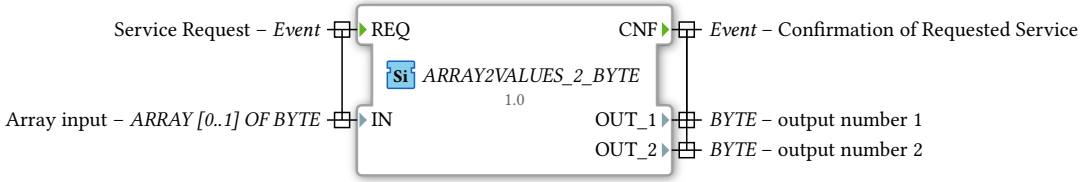
⚖️ Vergleich mit ähnlichen Bausteinen

- **ARRAY2VALUES_2_LREAL**: dieselbe generische Implementierung für den Datentyp `LREAL`.
- **`VALUES2ARRAY_16_BYTE`**: die Umkehrrichtung — fasst 16 Einzelwerte zu einem Array zusammen.
- **`ARRAY2ARRAY_2_BYTE`**: kopiert das Array unverändert, statt es zu zerlegen.

Fazit

`ARRAY2VALUES_16_BYTE` liefert eine einfache, generisch implementierte Zerlegung eines `BYTE`-Arrays der Größe 16 in 16 diskrete Einzelvariablen und eignet sich zur Anpassung array-basierter an variablenbasierte Schnittstellen.

2.2.1.7 ARRAY2VALUES_2_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ARRAY2VALUES_2_BYTE` zerlegt ein `BYTE`-Array der Größe 2 in 2 einzelne Skalarvariablen `OUT_1`... `OUT_2`. Er gehört zur generischen `GEN_ARRAY2ARRAY`-Familie (vgl. ARRAY2VALUES_2_LREAL) und dient der Umwandlung von Array-Daten in diskrete Einzelwerte.

Schnittstellenstruktur

Ereignis-Eingänge
- **REQ**: Löst die Zerlegung aus, trägt `IN`.

Ereignis-Ausgänge
- **CNF**: Bestätigt den Abschluss, trägt `OUT_1`, `OUT_2`.

Daten-Eingänge
- **IN** (`BYTE`, Array-Größe 2): Das zu zerlegende Array (8-Bit-Bitmuster).

Daten-Ausgänge
- `OUT_1`, `OUT_2` (`BYTE`): `OUT_i` entspricht `IN[i-1]`, also dem i-ten Element des Arrays.

Funktionsweise
Bei Eintreffen von `REQ` wird jedes Element von `IN` auf den entsprechenden Skalarausgang `OUT_i` kopiert (`IN[0]` ⟶ `OUT_1`, ..., `IN[1]` ⟶ `OUT_2`), anschließend wird `CNF` ausgelöst.

Technische Besonderheiten
- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY'`, dieselbe C++-Basis wie ARRAY2VALUES_2_LREAL und die übrigen `ARRAY2VALUES_*`-Varianten.
- **Feste Größe 2**: Für andere Array-Größen desselben Typs siehe `ARRAY2VALUES_4_BYTE`, `ARRAY2VALUES_8_BYTE`, `ARRAY2VALUES_16_BYTE`, `ARRAY2VALUES_32_BYTE`.

Zustandsübersicht
Zustandslos: jedes `REQ` führt unmittelbar zur vollständigen Zerlegung und zu `CNF`.

Anwendungsszenarien
- **Datenaufbereitung**: Ein vorgeschalteter Baustein liefert ein `BYTE`-Array, ein nachgeschalteter Baustein benötigt jedoch diskrete Einzelvariablen.
- **Schnittstellenanpassung** zwischen array-basierten und variablenbasierten Bausteinschnittstellen.

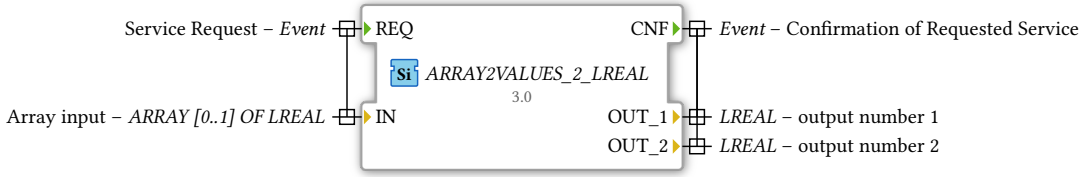
⚖️ Vergleich mit ähnlichen Bausteinen
- ARRAY2VALUES_2_LREAL: dieselbe generische Implementierung für den Datentyp `LREAL`.
- **`VALUES2ARRAY_2_BYTE`**: die Umkehrrichtung — fasst 2 Einzelwerte zu einem Array zusammen.
- **`ARRAY2ARRAY_2_BYTE`**: kopiert das Array unverändert, statt es zu zerlegen.

Fazit
`ARRAY2VALUES_2_BYTE` liefert eine einfache, generisch implementierte Zerlegung eines `BYTE`-Arrays der Größe 2 in 2 diskrete Einzelvariablen und eignet sich zur Anpassung array-basierter an variablenbasierte Schnittstellen.

2.2.1.8 ARRAY2VALUES_2_LREAL

• – ◦ • – ◦ • – ◦ •

Einleitung
Der Funktionsblock `ARRAY2VALUES_2_LREAL` ist ein Dienstschnittstellen-Baustein, der darauf ausgelegt ist, ein Array von zwei `LREAL`-Werten in zwei separate `LREAL`-Ausgangsvariablen aufzuteilen. Er dient der einfachen Umwandlung von einem strukturierten Array-Format in individuelle Datenpunkte.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Ein Service-Anforderungsereignis, das die Konvertierung initiiert. Es ist mit dem Daten-Eingang `IN` assoziiert.

Ereignis-Ausgänge

- **CNF**: Ein Bestätigungsereignis, das signalisiert, dass der angeforderte Dienst abgeschlossen wurde. Es ist mit den Daten-Ausgängen `OUT_1` und `OUT_2` assoziiert.

Daten-Eingänge

- **IN**: Ein Array-Eingang vom Typ `LREAL` mit einer Größe von 2 Elementen. Dieses Array enthält die zwei Werte, die aufgeteilt werden sollen.

Daten-Ausgänge

- **OUT_1**: Der erste Ausgangswert vom Typ `LREAL`, der dem ersten Element des `IN`-Arrays entspricht.
- **OUT_2**: Der zweite Ausgangswert vom Typ `LREAL`, der dem zweiten Element des `IN`-Arrays entspricht.

Adapter

Nicht vorhanden.

Funktionsweise

Wenn das `REQ`-Ereignis empfangen wird, liest der `ARRAY2VALUES_2_LREAL`-Funktionsblock die beiden `LREAL`-Werte aus dem `IN`-Array. Das erste Element des `IN`-Arrays (`IN[0]`) wird dem Ausgang `OUT_1` zugewiesen, und das zweite Element (`IN[1]`) wird dem Ausgang `OUT_2` zugewiesen. Nach erfolgreicher Zuweisung wird das `CNF`-Ereignis ausgelöst, um den Abschluss der Operation zu signalisieren.

Technische Besonderheiten

Der Funktionsblock ist darauf spezialisiert, genau ein Array von zwei `LREAL`-Werten zu verarbeiten und in zwei separate `LREAL`-Variablen aufzuteilen. Seine Implementierung ist direkt und auf diesen spezifischen Umwandlungszweck optimiert.

- **Generische Implementierung:** Interessanterweise verweist das Attribut `eclipse4diac::core::GenericClassName` auf `'GEN_ARRAY2ARRAY'`. Dies deutet darauf hin, dass der Baustein möglicherweise von einer generischen Array-zu-Array-Klasse abgeleitet ist, auch wenn seine Funktion die Aufteilung in einzelne Werte ist. Dies könnte ein Detail der Implementierungsgeschichte sein.

Zustandsübersicht

Dieser Funktionsblock ist zustandslos. Jede Auslösung des `REQ`-Ereignisses führt zu einer direkten Verarbeitung der aktuellen `IN`-Werte und einer Ausgabe über `OUT_1` und `OUT_2` ohne interne Zustandsänderungen, die über die aktuelle Transaktion hinausgehen würden.

Anwendungsszenarien

- **Datenaufbereitung**: Wenn Sensoren oder andere Bausteine Daten als Array von zwei `LREAL`-Werten liefern (z.B. X/Y-Koordinaten, Minimum/Maximum-Werte), aber nachfolgende Bausteine einzelne Variablen benötigen.
- **Schnittstellenanpassung**: Anpassen von Schnittstellen, bei denen ein Quellbaustein Array-Daten sendet, der Zielbaustein jedoch diskrete Eingangsvariablen erwartet.
- **Vereinfachung von Logik**: Erhöhung der Lesbarkeit und Vereinfachung der Logik, indem Array-Zugriffe in einzelne Variablen umgewandelt werden, die in weiteren Schritten leichter verarbeitet werden können.

⚖️ Vergleich mit ähnlichen Bausteinen

Ähnliche Bausteine könnten generische Array-Splitter sein, die eine variable Anzahl von Array-Elementen in eine entsprechende Anzahl von Ausgängen aufteilen können. `ARRAY2VALUES_2_LREAL` ist jedoch für den spezifischen Fall eines `LREAL`-Arrays der Größe 2 optimiert, was ihn einfacher in der Konfiguration und möglicherweise effizienter für diesen konkreten Anwendungsfall macht als einen flexibleren, aber potenziell komplexeren generischen Baustein.

Metadaten

Attribut	Wert
Copyright	(c) 2014 Profactor GmbH
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner), 1.0 (2014-07-09, Matthias Plasch)
4diac-Paket	eclipse4diac::convert

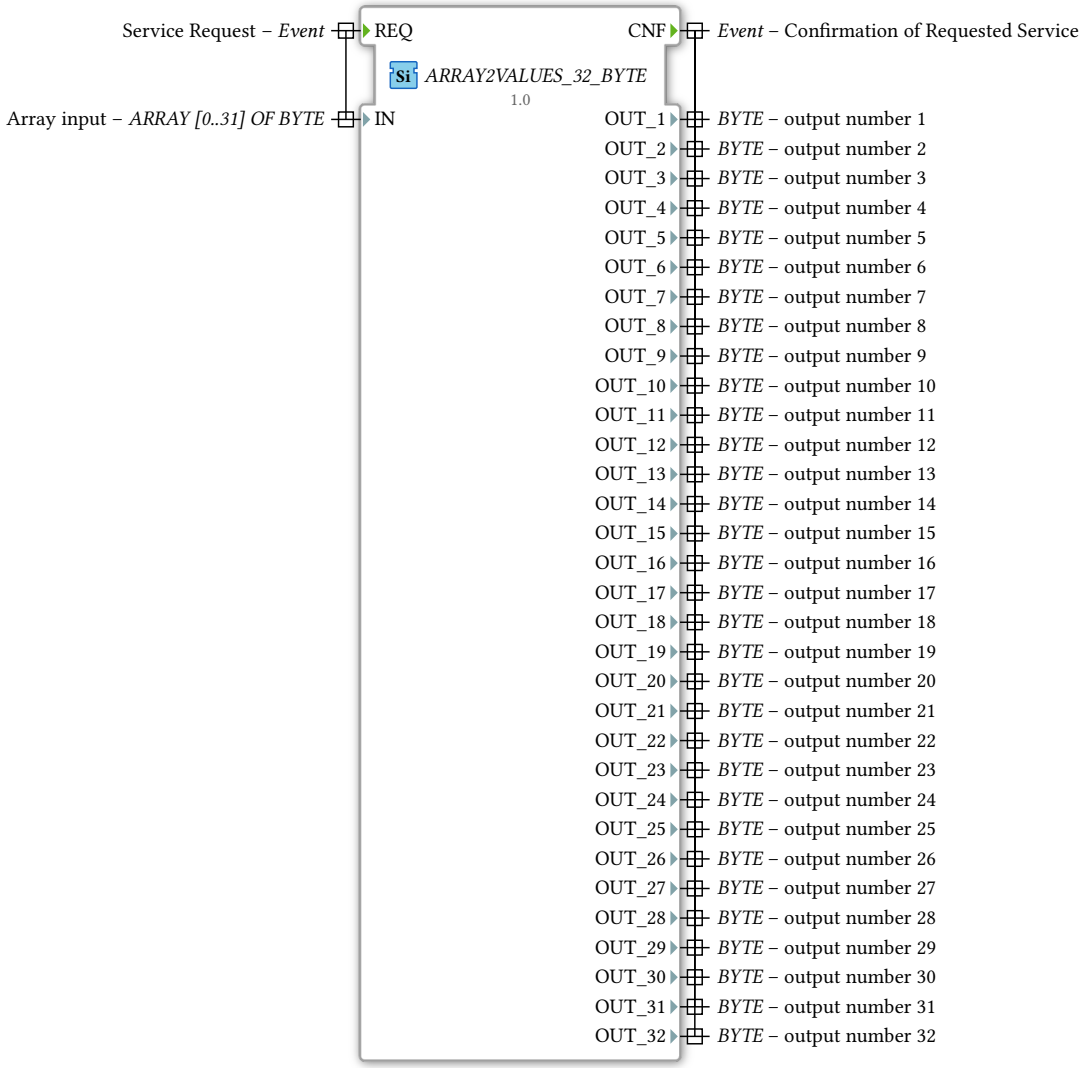
Fazit

Der `ARRAY2VALUES_2_LREAL`-Funktionsblock ist ein nützliches Werkzeug für die gezielte Umwandlung eines `LREAL`-Arrays der Größe 2 in zwei separate `LREAL`-Ausgänge. Er bietet eine einfache und effiziente Lösung für die Datenaufbereitung und Schnittstellenanpassung in 4diac-Anwendungen, bei denen diese spezifische Konvertierung erforderlich ist.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.2.1.9 ARRAY2VALUES_32_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der ARRAY2VALUES_32_BYTE zerlegt ein BYTE-Array der Größe 32 in 32 einzelne Skalarvariablen OUT_1… OUT_32. Er gehört zur generischen GEN_ARRAY2ARRAY-Familie (vgl. ARRAY2VALUES_2_LREAL) und dient der Umwandlung von Array-Daten in diskrete Einzelwerte.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Zerlegung aus, trägt IN.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt OUT_1, OUT_2, OUT_3, OUT_4, OUT_5, OUT_6, OUT_7, OUT_8, OUT_9, OUT_10, OUT_11, OUT_12, OUT_13, OUT_14, OUT_15, OUT_16, OUT_17, OUT_18, OUT_19, OUT_20, OUT_21, OUT_22, OUT_23, OUT_24, OUT_25, OUT_26, OUT_27, OUT_28, OUT_29, OUT_30, OUT_31, OUT_32.

Daten-Eingänge

- **IN** (BYTE, Array-Größe 32): Das zu zerlegende Array (8-Bit-Bitmuster).

Daten-Ausgänge

- OUT_1, OUT_2, OUT_3, OUT_4, OUT_5, OUT_6, OUT_7, OUT_8, OUT_9, OUT_10, OUT_11, OUT_12, OUT_13, OUT_14, OUT_15, OUT_16, OUT_17, OUT_18, OUT_19, OUT_20, OUT_21, OUT_22, OUT_23, OUT_24, OUT_25, OUT_26, OUT_27, OUT_28, OUT_29, OUT_30, OUT_31, OUT_32 (BYTE): OUT_i entspricht IN[i-1], also dem i-ten Element des Arrays.

Funktionsweise

Bei Eintreffen von REQ wird jedes Element von IN auf den entsprechenden Skalarausgang OUT_i kopiert (IN[0] ⟶ OUT_1, …, IN[31] ⟶ OUT_32), anschließend wird CNF ausgelöst.

Technische Besonderheiten

- **Generische Implementierung**: eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY', dieselbe C++-Basis wie ARRAY2VALUES_2_LREAL und die übrigen ARRAY2VALUES_*-Varianten.
- **Feste Größe 32**: Für andere Array-Größen desselben Typs siehe ARRAY2VALUES_2_BYTE, ARRAY2VALUES_4_BYTE, ARRAY2VALUES_8_BYTE, ARRAY2VALUES_16_BYTE.

Zustandsübersicht

Zustandslos: jedes REQ führt unmittelbar zur vollständigen Zerlegung und zu CNF.

Anwendungsszenarien

- **Datenaufbereitung**: Ein vorgeschalteter Baustein liefert ein BYTE-Array, ein nachgeschalteter Baustein benötigt jedoch diskrete Einzelvariablen.
- **Schnittstellenanpassung** zwischen array-basierten und variablenbasierten Bausteinschnittstellen.

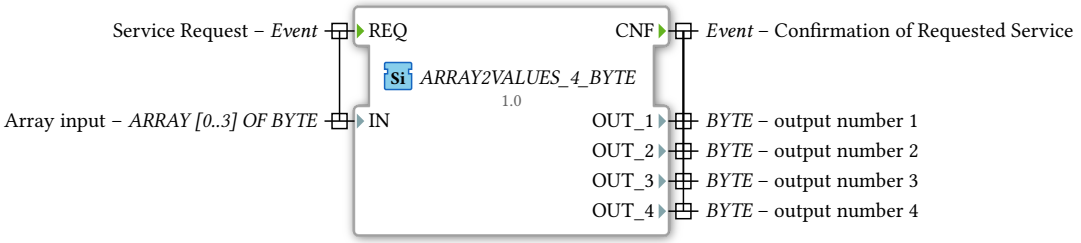
⚖️ Vergleich mit ähnlichen Bausteinen

- **ARRAY2VALUES_2_LREAL**: dieselbe generische Implementierung für den Datentyp LREAL.
- **VALUES2ARRAY_32_BYTE**: die Umkehrrichtung — fasst 32 Einzelwerte zu einem Array zusammen.
- **ARRAY2ARRAY_2_BYTE**: kopiert das Array unverändert, statt es zu zerlegen.

Fazit

ARRAY2VALUES_32_BYTE liefert eine einfache, generisch implementierte Zerlegung eines BYTE-Arrays der Größe 32 in 32 diskrete Einzelvariablen und eignet sich zur Anpassung array-basierter an variablenbasierte Schnittstellen.

2.2.1.10 ARRAY2VALUES_4_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ARRAY2VALUES_4_BYTE` zerlegt ein `BYTE`-Array der Größe 4 in 4 einzelne Skalarvariablen `OUT_1`…`OUT_4`. Er gehört zur generischen `GEN_ARRAY2ARRAY`-Familie (vgl. ARRAY2VALUES_2_LREAL) und dient der Umwandlung von Array-Daten in diskrete Einzelwerte.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Zerlegung aus, trägt `IN`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT_1`, `OUT_2`, `OUT_3`, `OUT_4`.

Daten-Eingänge

- **IN** (`BYTE`, Array-Größe 4): Das zu zerlegende Array (8-Bit-Bitmuster).

Daten-Ausgänge

- `OUT_1`, `OUT_2`, `OUT_3`, `OUT_4` (`BYTE`): `OUT_i` entspricht `IN[i-1]`, also dem i-ten Element des Arrays.

Funktionsweise

Bei Eintreffen von `REQ` wird jedes Element von `IN` auf den entsprechenden Skalarausgang `OUT_i` kopiert (`IN[0]` ⟶ `OUT_1`, …, `IN[3]` ⟶ `OUT_4`), anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY'`, dieselbe C++-Basis wie ARRAY2VALUES_2_LREAL und die übrigen `ARRAY2VALUES_*`-Varianten.
- **Feste Größe 4**: Für andere Array-Größen desselben Typs siehe `ARRAY2VALUES_2_BYTE`, `ARRAY2VALUES_8_BYTE`, `ARRAY2VALUES_16_BYTE`, `ARRAY2VALUES_32_BYTE`.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur vollständigen Zerlegung und zu `CNF`.

Anwendungsszenarien

- **Datenaufbereitung**: Ein vorgeschalteter Baustein liefert ein `BYTE`-Array, ein nachgeschalteter Baustein benötigt jedoch diskrete Einzelvariablen.
- **Schnittstellenanpassung** zwischen array-basierten und variablenbasierten Bausteinschnittstellen.

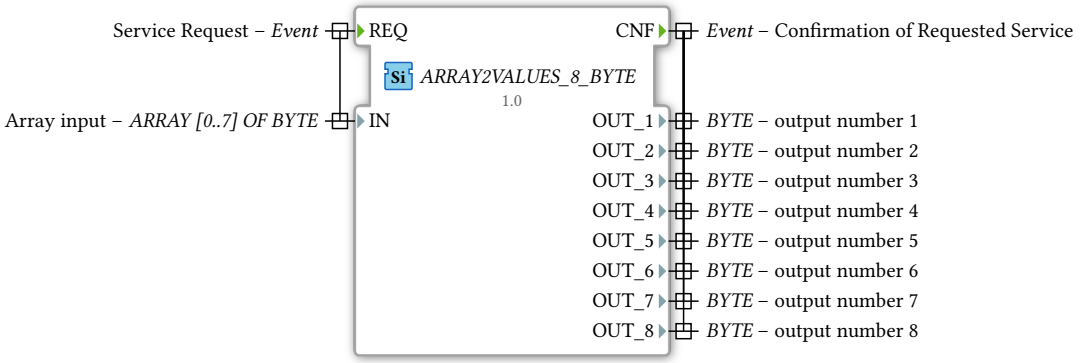
⚖️ Vergleich mit ähnlichen Bausteinen

- ARRAY2VALUES_2_LREAL: dieselbe generische Implementierung für den Datentyp `LREAL`.
- **`VALUES2ARRAY_4_BYTE`**: die Umkehrrichtung — fasst 4 Einzelwerte zu einem Array zusammen.
- **`ARRAY2ARRAY_2_BYTE`**: kopiert das Array unverändert, statt es zu zerlegen.

Fazit

`ARRAY2VALUES_4_BYTE` liefert eine einfache, generisch implementierte Zerlegung eines `BYTE`-Arrays der Größe 4 in 4 diskrete Einzelvariablen und eignet sich zur Anpassung array-basierter an variablenbasierte Schnittstellen.

2.2.1.11 ARRAY2VALUES_8_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ARRAY2VALUES_8_BYTE` zerlegt ein `BYTE`-Array der Größe 8 in 8 einzelne Skalarvariablen `OUT_1`… `OUT_8`. Er gehört zur generischen `GEN_ARRAY2ARRAY`-Familie (vgl. ARRAY2VALUES_2_LREAL) und dient der Umwandlung von Array-Daten in diskrete Einzelwerte.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Zerlegung aus, trägt `IN`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT_1`, `OUT_2`, `OUT_3`, `OUT_4`, `OUT_5`, `OUT_6`, `OUT_7`, `OUT_8`.

Daten-Eingänge

- **IN** (`BYTE`, Array-Größe 8): Das zu zerlegende Array (8-Bit-Bitmuster).

Daten-Ausgänge

- `OUT_1`, `OUT_2`, `OUT_3`, `OUT_4`, `OUT_5`, `OUT_6`, `OUT_7`, `OUT_8` (`BYTE`): `OUT_i` entspricht `IN[i-1]`, also dem i-ten Element des Arrays.

Funktionsweise

Bei Eintreffen von `REQ` wird jedes Element von `IN` auf den entsprechenden Skalarausgang `OUT_i` kopiert (`IN[0]` ⟶ `OUT_1`, …, `IN[7]` ⟶ `OUT_8`), anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY'`, dieselbe C++-Basis wie ARRAY2VALUES_2_LREAL und die übrigen `ARRAY2VALUES_*`-Varianten.
- **Feste Größe 8**: Für andere Array-Größen desselben Typs siehe `ARRAY2VALUES_2_BYTE`, `ARRAY2VALUES_4_BYTE`, `ARRAY2VALUES_16_BYTE`, `ARRAY2VALUES_32_BYTE`.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur vollständigen Zerlegung und zu `CNF`.

Anwendungsszenarien

- **Datenaufbereitung**: Ein vorgeschalteter Baustein liefert ein `BYTE`-Array, ein nachgeschalteter Baustein benötigt jedoch diskrete Einzelvariablen.
- **Schnittstellenanpassung** zwischen array-basierten und variablenbasierten Bausteinschnittstellen.

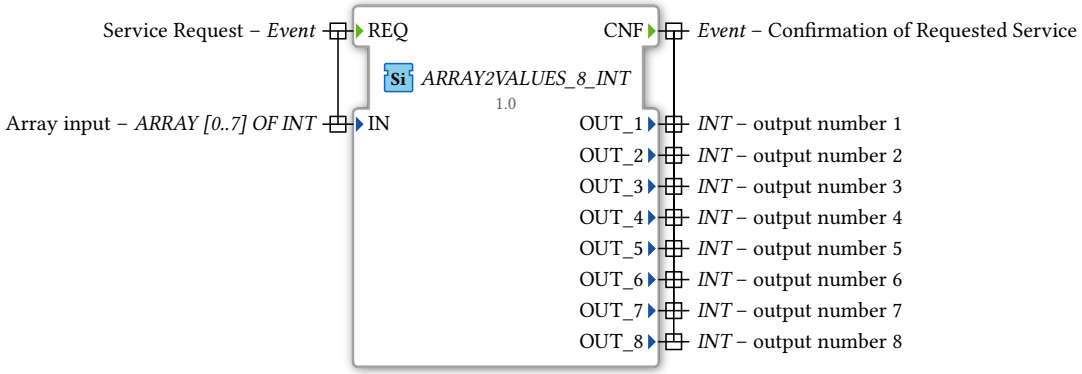
⚖️ Vergleich mit ähnlichen Bausteinen

- ARRAY2VALUES_2_LREAL: dieselbe generische Implementierung für den Datentyp `LREAL`.
- **`VALUES2ARRAY_8_BYTE`**: die Umkehrrichtung — fasst 8 Einzelwerte zu einem Array zusammen.
- **`ARRAY2ARRAY_8_BYTE`**: kopiert das Array unverändert, statt es zu zerlegen.

Fazit

`ARRAY2VALUES_8_BYTE` liefert eine einfache, generisch implementierte Zerlegung eines `BYTE`-Arrays der Größe 8 in 8 diskrete Einzelvariablen und eignet sich zur Anpassung array-basierter an variablenbasierte Schnittstellen.

2.2.1.12 ARRAY2VALUES_8_INT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ARRAY2VALUES_8_INT` zerlegt ein `INT`-Array der Größe 8 in 8 einzelne Skalarvariablen `OUT_1`…`OUT_8`. Er gehört zur generischen `GEN_ARRAY2ARRAY`-Familie (vgl. ARRAY2VALUES_2_LREAL) und dient der Umwandlung von Array-Daten in diskrete Einzelwerte.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Zerlegung aus, trägt `IN`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT_1`, `OUT_2`, `OUT_3`, `OUT_4`, `OUT_5`, `OUT_6`, `OUT_7`, `OUT_8`.

Daten-Eingänge

- **IN** (`INT`, Array-Größe 8): Das zu zerlegende Array (16-Bit-Ganzzahl (vorzeichenbehaftet)).

Daten-Ausgänge

- `OUT_1`, `OUT_2`, `OUT_3`, `OUT_4`, `OUT_5`, `OUT_6`, `OUT_7`, `OUT_8` (`INT`): `OUT_i` entspricht `IN[i-1]`, also dem i-ten Element des Arrays.

Funktionsweise

Bei Eintreffen von `REQ` wird jedes Element von `IN` auf den entsprechenden Skalarausgang `OUT_i` kopiert (`IN[0]` ⟶ `OUT_1`, …, `IN[7]` ⟶ `OUT_8`), anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY'`, dieselbe C++-Basis wie ARRAY2VALUES_2_LREAL und die übrigen `ARRAY2VALUES_*`-Varianten.
- **Feste Größe 8**: Für andere Array-Größen desselben Typs siehe .

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur vollständigen Zerlegung und zu `CNF`.

Anwendungsszenarien

- **Datenaufbereitung**: Ein vorgeschalteter Baustein liefert ein `INT`-Array, ein nachgeschalteter Baustein benötigt jedoch diskrete Einzelvariablen.
- **Schnittstellenanpassung** zwischen array-basierten und variablenbasierten Bausteinschnittstellen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **ARRAY2VALUES_2_LREAL**: dieselbe generische Implementierung für den Datentyp `LREAL`.
- **`VALUES2ARRAY_8_INT`**: die Umkehrrichtung — fasst 8 Einzelwerte zu einem Array zusammen.
- **`ARRAY2ARRAY_8_INT`**: kopiert das Array unverändert, statt es zu zerlegen.

Fazit

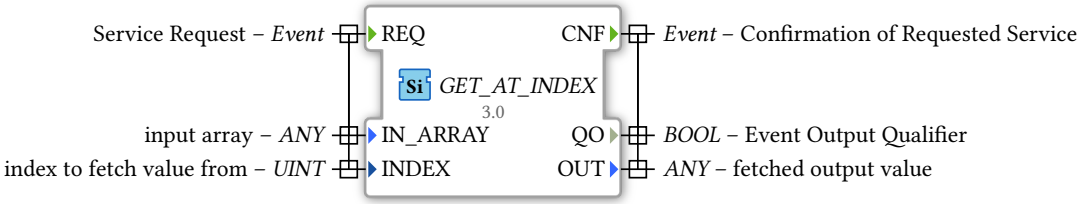
`ARRAY2VALUES_8_INT` liefert eine einfache, generisch implementierte Zerlegung eines `INT`-Arrays der Größe 8 in 8 diskrete Einzelvariablen und eignet sich zur Anpassung array-basierter an variablenbasierte Schnittstellen.

2.2.1.13 GET_AT_INDEX

• – ◦ • – ◦ • – ◦ •

Einleitung

Dieser Funktionsblock (FB) dient dazu, einen Wert aus einem Array an einer spezifischen Indexposition abzurufen. Eine wichtige Voraussetzung für die korrekte Funktionsweise ist, dass der Datentyp des Eingabe-Arrays mit dem Datentyp des Ausgabewerts übereinstimmt. Es handelt sich um einen grundlegenden Baustein für den Zugriff auf Array-Elemente.



Schnittstellenstruktur

Der `GET_AT_INDEX`-Funktionsblock verfügt über eine klar definierte Schnittstellenstruktur mit Ereignis- und Dateneingängen sowie -ausgängen.

Ereignis-Eingänge

- **REQ** (Typ: `Event`)
  - Dienstanfrage zur Ausführung des Funktionsblocks.
  - Ist verknüpft mit den Daten-Eingängen `IN_ARRAY` und `INDEX`, die bei der Auslösung des Ereignisses gelesen werden.

Ereignis-Ausgänge

- **CNF** (Typ: `Event`)
  - Bestätigung des angefragten Dienstes nach erfolgreicher Verarbeitung.
  - Ist verknüpft mit den Daten-Ausgängen `QO` und `OUT`, die nach der Verarbeitung bereitgestellt werden.

Daten-Eingänge

- **IN_ARRAY** (Typ: `ANY`)
  - Das Eingabe-Array, aus dem ein Wert abgerufen werden soll. Der Datentyp ist generisch (`ANY`), muss aber zur Ausgabe passen.
- **INDEX** (Typ: `UINT`)
  - Der Index (Position) innerhalb des `IN_ARRAY`, von dem der Wert abgerufen werden soll.

Daten-Ausgänge

- **QO** (Typ: `BOOL`)
  - Ein Qualifizierer für den Ereignis-Ausgang, der typischerweise den Erfolg oder Misserfolg des Vorgangs anzeigt (z.B. `TRUE` bei Erfolg, `FALSE` bei Fehler).
- **OUT** (Typ: `ANY`)
  - Der abgerufene Wert aus dem `IN_ARRAY` an der angegebenen `INDEX`-Position. Der Datentyp ist generisch (`ANY`).

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Wenn ein REQ-Ereignis am Eingang des Funktionsblocks empfangen wird, liest der GET_AT_INDEX-FB die aktuellen Werte der Daten-Eingänge IN_ARRAY und INDEX. Er greift dann auf das IN_ARRAY zu und extrahiert den Wert an der durch INDEX angegebenen Position. Dieser abgerufene Wert wird an den Daten-Ausgang OUT weitergeleitet. Gleichzeitig wird der QO-Ausgang entsprechend dem Ergebnis des Vorgangs gesetzt (z.B. TRUE bei erfolgreichem Zugriff). Abschließend wird das CNF-Ereignis ausgelöst, um die Beendigung des Dienstes zu signalisieren.

Technische Besonderheiten

- **Datentyp-Übereinstimmung**: Es ist entscheidend, dass der Datentyp des Eingabe-Arrays (IN_ARRAY) mit dem erwarteten Datentyp des abgerufenen Werts (OUT) kompatibel ist, um Typfehler zu vermeiden.
- **Indexbereich**: Der Wert von INDEX muss innerhalb der gültigen Grenzen des IN_ARRAY liegen, um Laufzeitfehler oder undefiniertes Verhalten zu verhindern. Eine Überprüfung des Indexbereichs muss ggf. durch vor- oder nachgeschaltete Logik sichergestellt werden.
- **ANY-Typ**: Die Verwendung des ANY-Typs für IN_ARRAY und OUT macht den Baustein sehr flexibel, erfordert aber eine sorgfältige Typisierung in der Anwendung.

Zustandsübersicht

Der GET_AT_INDEX-Funktionsblock ist im Wesentlichen zustandslos in Bezug auf die Datenhaltung. Seine Zustände beschränken sich auf den Ausführungszyklus:

1. **Bereit**: Wartet auf ein REQ-Ereignis.
2. **Ausführend**: Liest Eingänge, führt den Array-Zugriff durch und setzt Ausgänge.
3. **Abgeschlossen**: Löst das CNF-Ereignis aus und kehrt in den Zustand „Bereit“ zurück.

Anwendungsszenarien

- **Datenprotokollierung und -analyse**: Abrufen spezifischer Messwerte aus einem Zeitreihen-Array.
- **Konfigurationsverwaltung**: Lesen einzelner Parameter aus einem Array von Konfigurationswerten.
- **Abarbeitung von Listen**: Sequenzielles oder zufälliges Abrufen von Elementen aus einer Liste von Aufgaben oder Objekten.
- **Robotersteuerung**: Zugriff auf Positionen oder Bewegungsparameter, die in Arrays gespeichert sind.
- **Visualisierung**: Abrufen von Datenpunkten für die Darstellung auf einer Benutzeroberfläche.

⚖️ Vergleich mit ähnlichen Bausteinen

Der GET_AT_INDEX-Baustein ist ein spezialisierter Baustein für den Array-Zugriff. Er unterscheidet sich von:

- **SET_AT_INDEX**: Einem komplementären Baustein, der Werte an einer bestimmten Array-Position setzt, anstatt sie abzurufen.
- **Array-Iteratoren**: Bausteinen, die typischerweise das gesamte Array durchlaufen, anstatt nur ein einzelnes Element über einen direkten Index abzurufen.
- **Allgemeinen Datenmanipulationsbausteinen**: Die nicht speziell für Array-Zugriffe konzipiert sind, sondern allgemeinere Operationen (z.B. Addition, Vergleich) durchführen.

GET_AT_INDEX füllt somit eine grundlegende Lücke für den direkten, indizierten Lesezugriff auf Array-Datenstrukturen.

Metadaten

Attribut	Wert
Copyright	(c) 2014 Profactor GmbH
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner), 1.0 (2014-07-11, Matthias Plasch)
4diac-Paket	eclipse4diac::convert

Fazit

Der `GET_AT_INDEX`-Funktionsblock ist ein elementarer und flexibler Baustein, der den direkten und indizierten Zugriff auf Array-Elemente ermöglicht. Seine Einfachheit und die Unterstützung generischer Datentypen (`ANY`) machen ihn zu einem vielseitigen Werkzeug in zahlreichen Automatisierungs- und Steuerungsanwendungen, bei denen Daten in geordneten Listen oder Tabellen vorliegen. Die korrekte Anwendung erfordert lediglich die Beachtung der Datentyp-Kompatibilität und des gültigen Indexbereichs.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.2.1.14 GET_STRUCT_VALUE

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock (FB) `GET_STRUCT_VALUE` ist ein Service-Schnittstellen-Baustein, der dafür entwickelt wurde, spezifische Werte aus einer gegebenen Datenstruktur dynamisch auszulesen. Er bietet eine flexible Möglichkeit, auf Strukturmember zuzugreifen, deren Namen oder Pfade zur Laufzeit bestimmt werden.

Schnittstellenstruktur

Ereignis-Eingänge

Name	Typ	Beschreibung	Daten mit
`REQ`	`Event`	Fordert das Auslesen eines Wertes aus der Struktur an.	`member`, `in_struct`

Ereignis-Ausgänge

Name	Typ	Beschreibung	Daten mit
`CNF`	`Event`	Bestätigt den Abschluss des `REQ`-Ereignisses.	`QO`, `output`

Daten-Eingänge

Name	Typ	Beschreibung
`in_struct`	`ANY`	Die Datenstruktur, aus der ein Wert ausgelesen werden soll.
`member`	`STRING`	Der Name des Members der Struktur, der abgerufen werden soll. Innere Member werden über den Punkt-Operator (`.`) angesprochen.

Daten-Ausgänge

Name	Typ	Beschreibung
`QO`	`BOOL`	`TRUE`, wenn keine Fehler aufgetreten sind, andernfalls `FALSE`.
`output`	`ANY`	Der Wert des ausgewählten Members, wenn `QO = TRUE`. Andernfalls bleibt dieser Wert unverändert.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der GET_STRUCT_VALUE Funktionsblock wird durch ein REQ-Ereignis aktiviert. Bei Aktivierung versucht der Baustein, den Wert eines in member angegebenen Strukturmembers aus der am Eingang in_struct anliegenden Datenstruktur zu extrahieren. Der Pfad zum Member kann dabei auch verschachtelte Member (z.B. „config.port“) über den Punkt-Operator (.) spezifizieren. Nach dem Auslesen wird ein CNF-Ereignis ausgelöst, welches den Status (QO) und den extrahierten Wert (output) bereitstellt. Tritt ein Fehler auf (z.B. Member nicht gefunden oder ungültiger Pfad), wird QO auf FALSE gesetzt.

Technische Besonderheiten

- **Dynamischer Member-Zugriff:** Der Baustein ermöglicht den Zugriff auf Strukturmember über einen STRING-Eingang, was eine hohe Flexibilität bei der Datenverarbeitung bietet.
- **Verschachtelte Strukturen:** Über den Punkt-Operator (.) in der member-Zeichenkette können auch Member in verschachtelten Strukturen effizient adressiert werden.
- **Typunabhängigkeit:** Die Ein- und Ausgänge in_struct und output sind vom Typ ANY, was bedeutet, dass der Baustein mit beliebigen Datenstrukturen und Member-Datentypen umgehen kann.

Zustandsübersicht

Der GET_STRUCT_VALUE ist ein nicht-zustandsbehafteter Service-Baustein, der eine einfache Anforderungs-Bestätigungs-Logik implementiert (REQ/CNF). Er verarbeitet jede Anforderung individuell und behält keine komplexen internen Zustände bei.

Anwendungsszenarien

- **Dynamische Konfiguration:** Auslesen von Konfigurationsparametern aus einer zentralen Struktur, wobei die benötigten Parameter zur Laufzeit ermittelt werden.
- **Generische Datenverarbeitung:** Entwicklung von generischen Algorithmen oder Logiken, die basierend auf externen Eingaben oder Regeln auf verschiedene Teile einer Datenstruktur zugreifen müssen.
- **Protokollierung und Diagnose:** Extrahieren spezifischer Statusinformationen oder Messwerte aus einer komplexen Systemzustandsstruktur für Überwachungs- oder Diagnosezwecke.
- **Schnittstellen zu externen Systemen:** Anpassung an Datenformate externer Systeme, bei denen die Feldnamen dynamisch übergeben werden.

⚖️ Vergleich mit ähnlichen Bausteinen

Es gibt in der Standardbibliothek von 4diac keine direkt vergleichbaren Funktionsblöcke, die einen dynamischen Zugriff auf Strukturmember über einen STRING-Pfad ermöglichen. Dieser Baustein füllt eine Nische für Anwendungsfälle, die eine flexible und zur Laufzeit konfigurierbare Strukturmanipulation erfordern, welche über den statischen Zugriff herkömmlicher FBs hinausgeht.

Metadaten

Attribut	Wert
Copyright	(c) 2018 fortiss GmbH
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner), 1.0 (2018-10-16, Jose Cabral)
4diac-Paket	eclipse4diac::convert

🛠️ Zugehörige Übungen

- Uebung_052

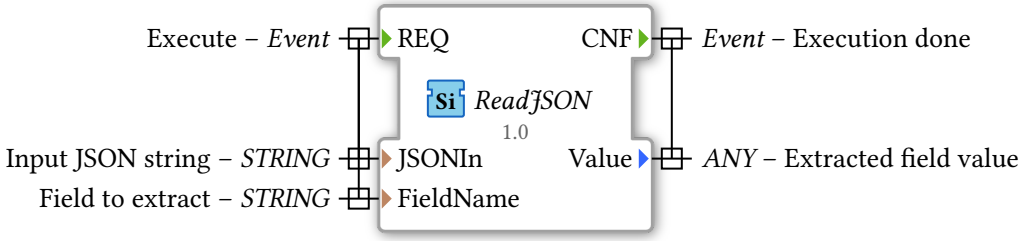
Fazit

Der GET_STRUCT_VALUE Funktionsblock ist ein wertvolles Werkzeug für Entwickler, die eine flexible und dynamische Methode zum Auslesen von Werten aus Datenstrukturen in 4diac benötigen. Seine Fähigkeit, auf Member über String-Pfade zuzugreifen, einschließlich verschachtelter Strukturen, macht ihn besonders nützlich in Szenarien, in denen die Struktur der benötigten Daten zur Kompilierzeit nicht vollständig bekannt oder variabel ist.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.2.1.15 ReadJSON



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ReadJSON`-Funktionsblock extrahiert den Wert eines benannten Felds aus einem JSON-String. Er dient als einfacher, ereignisgesteuerter JSON-Decoder für Anwendungsfälle, in denen ein einzelnes Feld aus einer JSON-Struktur ausgelesen werden soll, ohne die gesamte Struktur in dedizierte IEC-61131-3-Datentypen abbilden zu müssen.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst das Auslesen aus, trägt `JSONIn` und `FieldName`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `Value`.

Daten-Eingänge

- **JSONIn** (STRING): Der zu durchsuchende JSON-String.
- **FieldName** (STRING): Der Name des zu extrahierenden Felds.

Daten-Ausgänge

- **Value** (ANY): Der extrahierte Feldwert, generisch typisiert.

Funktionsweise

Bei Eintreffen von `REQ` durchsucht der Baustein `JSONIn` nach dem Feld `FieldName` und gibt dessen Wert über `Value` aus. Da `Value` vom Typ `ANY` ist, kann der extrahierte Wert je nach JSON-Inhalt als Zahl, Zeichenkette oder boolescher Wert interpretiert werden. Anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **ANY-Rückgabetyp**: `Value` passt sich generisch an den tatsächlichen JSON-Feldtyp an, muss aber beim Anschluss an eine konkret typisierte Variable mit deren Typ kompatibel sein.
- **Einzelfeld-Extraktion**: Der Baustein liest genau ein benanntes Feld pro Aufruf; verschachtelte oder mehrere Felder erfordern mehrere `ReadJSON`-Aufrufe oder -Instanzen.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur Extraktion und zu `CNF`.

Anwendungsszenarien

- **Konfigurationsauslesen**: Ein per Netzwerk oder Datei empfangener JSON-Konfigurationsstring soll ausgewertet werden, ohne eine vollständige STRUCT-Definition anzulegen.
- **Kommunikation mit REST-/Web-Schnittstellen**: Antworten von HTTP-APIs, die JSON liefern, können feldweise ausgelesen werden.

⚖️ Vergleich mit ähnlichen Bausteinen

- **ToJSON**: die Gegenrichtung — fügt ein Feld zu einem JSON-String hinzu bzw. erstellt einen neuen.
- **`GET_STRUCT_VALUE`**: liest ein Feld aus einer IEC-61131-3-STRUCT statt aus einem JSON-String.

Fazit

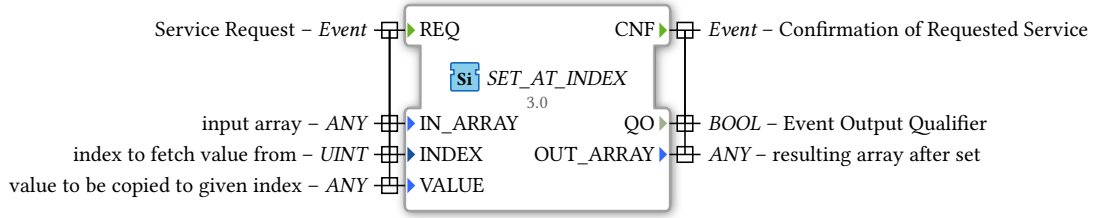
`ReadJSON` liefert einen einfachen, feldweisen JSON-Decoder für 4diac-Anwendungen, die mit JSON-basierten Konfigurationen oder Web-Schnittstellen kommunizieren, ohne dafür eine vollständige Struktur-Typdefinition zu benötigen.

2.2.1.16 SET_AT_INDEX

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `SET_AT_INDEX` dient dazu, den Wert eines Elements in einem Array an einem bestimmten Index zu setzen. Dabei muss sichergestellt werden, dass die Datentypen des zu setzenden Wertes dem Datentyp des Arrays entsprechen. Das resultierende Ausgabe-Array wird ebenfalls den Datentyp des Eingabe-Arrays aufweisen. Dieser Baustein ist nützlich für die dynamische Manipulation von Array-Daten innerhalb eines Steuerungsprogramms.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Event): Ein Anforderungseingang, der die Ausführung des Funktionsblocks initiiert. Mit diesem Ereignis werden die Daten-Eingänge `IN_ARRAY`, `INDEX` und `VALUE` verarbeitet.

Ereignis-Ausgänge

- **CNF** (Event): Ein Bestätigungsausgang, der nach erfolgreicher Ausführung des Dienstes ausgelöst wird. Mit diesem Ereignis werden die Daten-Ausgänge `QO` und `OUT_ARRAY` aktualisiert.

Daten-Eingänge

- **IN_ARRAY** (ANY): Das Eingabearray, dessen Wert an einem bestimmten Index geändert werden soll. Der Datentyp `ANY` ermöglicht die Verwendung mit verschiedenen Array-Typen.
- **INDEX** (UINT): Der nullbasierte Index des Elements im Array, dessen Wert gesetzt werden soll.
- **VALUE** (ANY): Der Wert, der an der angegebenen Position im Array gesetzt werden soll. Der Datentyp muss mit dem Datentyp des `IN_ARRAY` übereinstimmen.

Daten-Ausgänge

- **QO** (BOOL): Ein Event-Ausgabe-Qualifizierer, der den Erfolg der Operation anzeigt. `TRUE` bei Erfolg, `FALSE` bei einem Fehler (z.B. ungültiger Index).
- **OUT_ARRAY** (ANY): Das resultierende Array nach dem Setzen des Wertes am angegebenen Index.

Adapter

Nicht vorhanden.

Funktionsweise

Der `SET_AT_INDEX` Funktionsblock wartet auf ein `REQ`-Ereignis. Sobald dieses Ereignis eintrifft, überprüft der Baustein die übergebenen `IN_ARRAY`, `INDEX` und `VALUE`. Er versucht, den `VALUE` am `INDEX` des `IN_ARRAY` zu speichern. Bei erfolgreicher Operation wird ein neues Array erzeugt und an `OUT_ARRAY` ausgegeben, wobei `QO` auf `TRUE` gesetzt wird. Ist der angegebene `INDEX` außerhalb des gültigen Bereichs des `IN_ARRAY`

oder stimmen die Datentypen von VALUE und den Array-Elementen nicht überein, wird die Operation möglicherweise nicht durchgeführt, und QO könnte FALSE anzeigen.

Technische Besonderheiten

- **Datentyp-Flexibilität:** Durch die Verwendung von ANY für die Array- und Wert-Parameter ist der Baustein flexibel in Bezug auf die zu verarbeitenden Datentypen. Es ist jedoch entscheidend, dass der Datentyp von VALUE exakt mit dem Elementdatentyp des IN_ARRAY übereinstimmt, um Laufzeitfehler zu vermeiden.
- **Index-Typ:** Der INDEX ist vom Typ UINT (Unsigned Integer), was bedeutet, dass nur positive Indizes zulässig sind.
- **Nullbasierte Indizierung:** Der Index beginnt bei 0 für das erste Element des Arrays.
- **Fehlerbehandlung:** Die QO-Ausgabe dient als einfacher Indikator für den Erfolg der Operation. Eine erweiterte Fehlerbehandlung (z.B. für Index-Out-Of-Bounds) muss möglicherweise auf Anwendungsebene implementiert oder durch zusätzliche Prüfungen ergänzt werden.

Zustandsübersicht

Der SET_AT_INDEX Baustein ist im Wesentlichen ein ereignisgesteuerter Baustein ohne komplexen internen Zustand. Er wartet auf ein REQ-Ereignis, führt die Aktion aus und gibt ein CNF-Ereignis aus. Seine Funktion ist direkt und reaktiv.

Anwendungsszenarien

- **Dynamische Konfiguration:** Ändern von Konfigurationsparametern, die in Arrays gespeichert sind, zur Laufzeit.
- **Datenverarbeitung:** Aktualisieren von Sensor- oder Aktuatordaten, die in Arrays gesammelt werden.
- **Algorithmische Anpassung:** Modifizieren von Look-Up-Tabellen oder Reglerparametern in Echtzeit.
- **Benutzerinteraktion:** Setzen von Array-Werten basierend auf Benutzereingaben.

⚖️ Vergleich mit ähnlichen Bausteinen

Dieser Baustein ist komplementär zum GET_AT_INDEX-Baustein, der Werte aus einem Array an einem bestimmten Index auslesen würde. Er ist eine grundlegende Operation für die Array-Manipulation und bildet die Basis für komplexere Datenstruktur-Algorithmen. Im Gegensatz zu Bausteinen, die ganze Arrays kopieren oder verschieben, konzentriert sich SET_AT_INDEX auf die präzise Manipulation eines einzelnen Elements.

Metadaten

Attribut	Wert
Copyright	(c) 2014 Profactor GmbH
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner), 1.0 (2014-08-05, Matthias Plasch)
4diac-Paket	eclipse4diac::convert

Fazit

Der SET_AT_INDEX Funktionsblock ist ein grundlegendes, aber mächtiges Werkzeug zur Manipulation von Array-Daten in 4diac-Applikationen. Seine Flexibilität durch die ANY-Datentypen und die klare ereignisgesteuerte Schnittstelle machen ihn zu einem wertvollen Baustein für viele Anwendungen, die dynamische Datenstrukturen erfordern. Die korrekte Handhabung von Datentypen und Indizes ist entscheidend für seine fehlerfreie Nutzung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

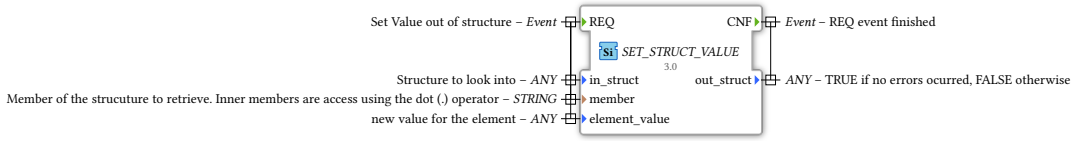
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.2.1.17 SET_STRUCT_VALUE

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `SET_STRUCT_VALUE` ist ein Service-Schnittstellen-Funktionsblock, der dazu dient, einen bestimmten Wert innerhalb einer Datenstruktur zu setzen. Er ermöglicht die dynamische Manipulation von Strukturelementen, indem er einen neuen Wert für ein benanntes Mitglied der Struktur zuweist.



Schnittstellenstruktur

Der Funktionsblock verfügt über Ereignis- und Datenschnittstellen, um seine Funktionalität zu ermöglichen.

Ereignis-Eingänge

- **REQ**: Dieses Ereignis fordert den Funktionsblock auf, den Wert in der angegebenen Struktur zu setzen. Es ist mit den Datenvariablen `member`, `in_struct` und `element_value` assoziiert.

Ereignis-Ausgänge

- **CNF**: Dieses Ereignis wird ausgelöst, wenn die `REQ`-Anforderung abgeschlossen wurde. Es ist mit der Datenvariable `out_struct` assoziiert.

Daten-Eingänge

- **in_struct** (Typ: ANY): Die Eingabestruktur, in der ein Wert geändert werden soll. Der Typ `ANY` ermöglicht die Verarbeitung beliebiger Strukturtypen.
- **member** (Typ: STRING): Der Name des Strukturmitglieds, dessen Wert gesetzt werden soll. Innere Mitglieder können über den Punktoperator (`.`) angesprochen werden (z.B. „config.port“).
- **element_value** (Typ: ANY): Der neue Wert, der dem angegebenen Strukturmitglied zugewiesen werden soll.

Daten-Ausgänge

- **out_struct** (Typ: ANY): Die modifizierte Struktur nach dem Setzen des Elements. Der Kommentar im XML besagt „TRUE if no errors occurred, FALSE otherwise“, was ein Überbleibsel eines `STATUS`-Ausgangs ist, aber hier bezieht es sich auf die ausgegebene, geänderte Struktur.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Nach Empfang eines `REQ`-Ereignisses greift der Funktionsblock auf die über `in_struct` bereitgestellte Datenstruktur zu. Basierend auf dem als `member` übergebenen String wird das entsprechende Element innerhalb der Struktur identifiziert. Der Funktionsblock weist dann den über `element_value` bereitgestellten Wert diesem Element zu. Anschließend gibt der Funktionsblock die modifizierte Struktur über `out_struct` aus und signalisiert den Abschluss des Vorgangs durch das `CNF`-Ereignis.

Technische Besonderheiten

- **Generische Typen (ANY)**: Durch die Verwendung des Datentyps `ANY` für `in_struct`, `element_value` und `out_struct` bietet dieser Funktionsblock hohe Flexibilität. Er kann mit beliebigen benutzerdefinierten oder vordefinierten Strukturtypen und Werten arbeiten, ohne dass spezifische Typen zur Kompilierzeit bekannt sein müssen.
- **Zugriff auf verschachtelte Mitglieder**: Die `member`-Variable unterstützt den Zugriff auf tiefere Ebenen einer verschachtelten Struktur mithilfe der Punktnotation (z.B. „outer.inner.value“). Dies vereinfacht die Handhabung komplexer Datenmodelle.

Zustandsübersicht

Der Funktionsblock operiert hauptsächlich in einem anforderungsbasierten (`REQ`) und bestätigungsbasierten (`CNF`) Schema. Er verarbeitet eine Anforderung, ändert die Daten und signalisiert den Abschluss. Es gibt keine komplexen internen Zustände, die über diesen einfachen Request/Confirm-Zyklus hinausgehen.

Anwendungsszenarien

- **Konfigurationsmanagement zur Laufzeit**: Dynamisches Ändern von Konfigurationsparametern, die in Strukturen gespeichert sind, ohne den Funktionsblock neu kompilieren zu müssen.
- **Datenmanipulation in SCADA/HMI-Systemen**: Aktualisieren spezifischer Werte in komplexen Datenobjekten, die von übergeordneten Systemen empfangen oder an diese gesendet werden.
- **Schnittstellen zu externen Systemen**: Anpassen von Datenstrukturen, die für die Kommunikation mit externen APIs oder Datenbanken verwendet werden.
- **Prozesssteuerung**: Modifizieren von Prozessparametern, die in Strukturen organisiert sind, basierend auf dynamischen Bedingungen oder Benutzereingaben.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Bausteinen, die fest verdrahtete Strukturen oder nur das Auslesen von Werten unterstützen, bietet `SET_STRUCT_VALUE` eine einzigartige Flexibilität durch:

- **Schreibzugriff**: Ermöglicht das Setzen von Werten, nicht nur das Lesen.
- **Generischer Zugriff**: Arbeitet mit beliebigen `ANY`-Typen, was ihn universeller macht als Bausteine, die an spezifische Strukturtypen gebunden sind.
- **Dynamische Member-Auswahl**: Der Member-Name wird zur Laufzeit als String übergeben, was eine dynamischere Feldselektion als bei fest definierten Ein-/Ausgängen erlaubt. Er ergänzt oft Bausteine, die Werte aus Strukturen auslesen (`GET_STRUCT_VALUE`), um eine vollständige Lese-/Schreibfunktionalität für generische Strukturen zu bieten.

Metadaten

Attribut	Wert
Copyright	(c) 2023 Primetals Technologies Austria GmbH
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner), 1.0 (2023-03-06, Alois Zoitl)
4diac-Paket	eclipse4diac::convert

Fazit

Der `SET_STRUCT_VALUE`-Funktionsblock ist ein mächtiges Werkzeug für die dynamische Manipulation von Datenstrukturen innerhalb einer 4diac-Anwendung. Seine generische Natur und die Unterstützung für den Zugriff auf verschachtelte Mitglieder machen ihn äußerst vielseitig für Anwendungen, die eine flexible Konfiguration oder Datenverarbeitung zur Laufzeit erfordern.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

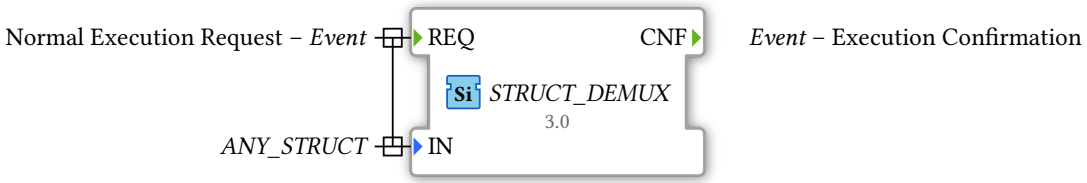
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.2.1.18 STRUCT_DEMUX

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock (FB) `STRUCT_DEMUX` ist ein generischer Demultiplexer für strukturierte Datentypen. Seine Hauptaufgabe ist es, eine am Eingang anliegende Datenstruktur in ihre einzelnen Bestandteile (Member) aufzuspalten. Diese einzelnen Member werden dann über separate Daten-Ausgänge zur Verfügung gestellt, die dynamisch erzeugt werden.



Schnittstellenstruktur

Die Schnittstelle des `STRUCT_DEMUX`-Funktionsblocks ist generisch definiert. Die tatsächlichen Daten-Ausgänge werden erst bei der Instanziierung des Blocks festgelegt.

Ereignis-Eingänge

- **REQ**: Löst die Ausführung des Bausteins aus und bewirkt das Auslesen der Eingangsstruktur.
  - **Mit Datenvariable**: `IN`

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss der Operation, nachdem die Werte der Struktur an die Ausgänge weitergeleitet wurden.
  - **Mit Datenvariablen**: Alle dynamisch erzeugten Daten-Ausgänge.

Daten-Eingänge

- **IN** (Typ: `ANY_STRUCT`): Die Eingangs-Datenstruktur, die aufgeteilt werden soll.

Daten-Ausgänge

Die Daten-Ausgänge dieses Funktionsblocks sind nicht fest vordefiniert. Stattdessen werden sie **dynamisch** basierend auf dem Datentyp erstellt, der mit dem `IN`-Eingang verbunden ist. Für jeden Member der Eingangsstruktur wird ein entsprechender Daten-Ausgang mit demselben Namen und Datentyp am Baustein erzeugt.

Beispiel: Wenn eine Struktur vom Typ `MyStruct` mit den Membern `a` (Typ `INT`) and `b` (Typ `BOOL`) an den `IN`-Eingang angeschlossen wird, erzeugt der `STRUCT_DEMUX`-Baustein automatisch zwei Daten-Ausgänge:

- `a` (Typ `INT`)
- `b` (Typ `BOOL`)

Das Bild oben illustriert genau diesen Fall.

Funktionsweise

Sobald ein `REQ`-Ereignis am Eingang des `STRUCT_DEMUX`-Funktionsblocks empfangen wird, liest der Baustein die am `IN`-Eingang anliegende Datenstruktur aus. Er extrahiert die Werte jedes einzelnen Members der Struktur und leitet sie an die entsprechenden, dynamisch erzeugten Daten-Ausgänge weiter. Nachdem alle Ausgangswerte aktualisiert wurden, wird das `CNF`-Ereignis ausgelöst, um den Abschluss des Vorgangs zu signalisieren.

Technische Besonderheiten

- **Generischer Baustein**: Dank des Attributs `GEN_STRUCT_DEMUX` ist der Baustein in der Lage, sich an jeden beliebigen strukturierten Datentyp (`ANY_STRUCT`) anzupassen.
- **Dynamische Schnittstelle**: Die Fähigkeit, seine Ausgänge basierend auf dem Eingangs-Datentyp zu generieren, macht ihn extrem flexibel und wiederverwendbar.
- **Service Interface Function Block Type**: Der Baustein ist als standardisierte Schnittstelle für diesen Dienst konzipiert.

Zustandsübersicht

Der `STRUCT_DEMUX` ist ein zustandsloser Baustein, der nach einem einfachen Anforderungs-Bestätigungs-Zyklus arbeitet:

1. **Bereit**: Wartet auf ein `REQ`-Ereignis.
2. **Ausführend**: Liest die Eingangsstruktur, extrahiert die Member-Werte und setzt die entsprechenden Ausgänge.
3. **Abgeschlossen**: Löst das `CNF`-Ereignis aus und kehrt in den Bereitschaftszustand zurück.

Anwendungsszenarien

- **Aufspaltung komplexer Daten**: Zerlegen von komplexen Datenstrukturen (z.B. Sensordaten, Statusinformationen) in einzelne Signale für die weitere Verarbeitung.
- **Verbesserung der Lesbarkeit**: Anstatt auf Strukturmember über `GET_STRUCT_VALUE` zuzugreifen, können die Member direkt als separate Datenleitungen in der Logik verwendet werden.
- **Schnittstellenanpassung**: Anpassen von Daten, die als eine einzelne Struktur von einem Baustein kommen, an mehrere Bausteine, die einzelne Eingänge erwarten.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`GET_STRUCT_VALUE`**: Während `GET_STRUCT_VALUE` einen einzelnen Member dynamisch über einen `STRING`-Namen extrahiert, legt `STRUCT_DEMUX` alle Member statisch als separate Ausgänge frei. `STRUCT_DEMUX` ist oft einfacher zu verwenden, wenn alle Member benötigt werden, während `GET_STRUCT_VALUE` flexibler ist, wenn nur bestimmte Member zur Laufzeit adressiert werden müssen.
- **`STRUCT_MUX`**: Der komplementäre Baustein, der einzelne Daten-Eingänge zu einer einzigen Datenstruktur zusammenfügt.

Metadaten

Attribut	Wert
Copyright	(c) 2020 Johannes Kepler University Linz
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	eclipse4diac::convert

🛠️ Zugehörige Übungen

- Uebung_051
- Uebung_120
- Uebung_121
- Uebung_122
- Uebung_122b
- Uebung_123
- Uebung_124
- Uebung_125
- Uebung_126
- Uebung_127
- Uebung_128
- Uebung_128b
- Uebung_130
- Uebung_131
- Uebung_132
- Uebung_133
- Uebung_134

Fazit

`STRUCT_DEMUX` ist ein fundamentaler und äußerst nützlicher Baustein für die Arbeit mit Datenstrukturen in 4diac. Seine Fähigkeit, jede beliebige Struktur automatisch in ihre Bestandteile zu zerlegen, vereinfacht die Anwendungslogik erheblich und fördert eine klare, lesbare Verdrahtung. Er ist das Standardwerkzeug, um auf die Inhalte von Strukturen zuzugreifen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

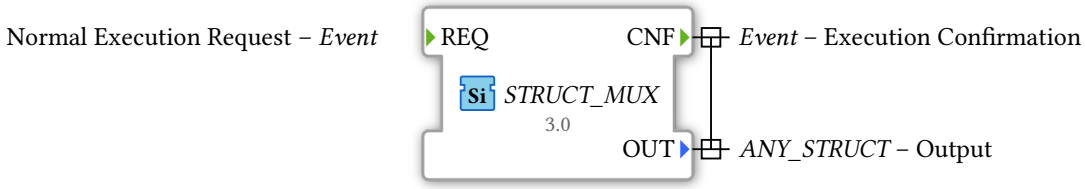
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.2.1.19 STRUCT_MUX

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock (FB) `STRUCT_MUX` ist ein generischer Multiplexer für strukturierte Datentypen. Er fungiert als Gegenstück zum `STRUCT_DEMUX` und hat die Aufgabe, einzelne Datenwerte von mehreren Eingängen zu einer einzigen Datenstruktur am Ausgang zusammenzufügen.



Schnittstellenstruktur

Die Schnittstelle des `STRUCT_MUX`-Funktionsblocks ist generisch definiert. Die tatsächlichen Daten-Eingänge werden erst bei der Instanziierung des Blocks festgelegt.

Ereignis-Eingänge

- **REQ**: Löst die Ausführung des Bausteins aus. Dies bewirkt das Einlesen der Werte an den Eingängen und das Zusammenbauen der Ausgangsstruktur.
  - **Mit Datenvariablen**: Alle dynamisch erzeugten Daten-Eingänge.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss der Operation, nachdem die Ausgangsstruktur erfolgreich erstellt wurde.
  - **Mit Datenvariable**: `OUT`

Daten-Eingänge

Die Daten-Eingänge dieses Funktionsblocks sind nicht fest vordefiniert. Stattdessen werden sie **dynamisch** basierend auf dem Datentyp erstellt, der für den `OUT`-Ausgang festgelegt wird. Für jeden Member der Ausgangsstruktur wird ein entsprechender Daten-Eingang mit demselben Namen und Datentyp am Baustein erzeugt.

Beispiel: Wenn der `OUT`-Ausgang auf den Datentyp `MyStruct` mit den Membern `a` (Typ `INT`) and `b` (Typ `BOOL`) gesetzt wird, erzeugt der `STRUCT_MUX`-Baustein automatisch zwei Daten-Eingänge:

- `a` (Typ `INT`)
- `b` (Typ `BOOL`)

Das Bild oben illustriert genau diesen Fall.

Daten-Ausgänge

- **OUT** (Typ: `ANY_STRUCT`): Die Ausgangs-Datenstruktur, die aus den Werten der Eingänge zusammengebaut wird.

Funktionsweise

Sobald ein `REQ`-Ereignis am Eingang des `STRUCT_MUX`-Funktionsblocks empfangen wird, liest der Baustein die Werte von all seinen dynamisch erzeugten Daten-Eingängen ein. Diese Werte werden dann zu einer einzigen Datenstruktur zusammengefügt. Die resultierende Struktur wird am `OUT`-Ausgang bereitgestellt. Unmittelbar danach wird das `CNF`-Ereignis ausgelöst, um den Abschluss des Vorgangs zu signalisieren.

Technische Besonderheiten

- **Generischer Baustein**: Dank des Attributs `GEN_STRUCT_MUX` ist der Baustein in der Lage, sich an jeden beliebigen strukturierten Datentyp (`ANY_STRUCT`) anzupassen.
- **Dynamische Schnittstelle**: Die Fähigkeit, seine Eingänge basierend auf dem Ausgangs-Datentyp zu generieren, macht ihn extrem flexibel und wiederverwendbar.
- **Service Interface Function Block Type**: Der Baustein ist als standardisierte Schnittstelle für diesen Dienst konzipiert.

Zustandsübersicht

Der `STRUCT_MUX` ist ein zustandsloser Baustein, der nach einem einfachen Anforderungs-Bestätigungs-Zyklus arbeitet:

1. **Bereit**: Wartet auf ein `REQ`-Ereignis.
2. **Ausführend**: Liest die Eingangswerte ein und baut die Ausgangsstruktur zusammen.
3. **Abgeschlossen**: Stellt die Struktur am `OUT`-Ausgang bereit, löst das `CNF`-Ereignis aus und kehrt in den Bereitschaftszustand zurück.

Anwendungsszenarien

- **Zusammenfügen von Daten**: Bündeln einzelner Signale (z.B. Konfigurationsparameter, Steuerwerte) in einer einzigen Struktur für die weitere Verarbeitung oder Kommunikation.
- **Verbesserung der Übersichtlichkeit**: Zusammenfassen vieler einzelner Datenleitungen zu einer einzigen strukturierten Leitung, um die Komplexität der grafischen Darstellung zu reduzieren.
- **Schnittstellenanpassung**: Anpassen einzelner Datenwerte an einen Baustein, der eine einzelne Datenstruktur als Eingang erwartet.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`STRUCT_DEMUX`**: Der komplementäre Baustein, der eine einzelne Datenstruktur in ihre einzelnen Member aufspaltet.
- **`SET_STRUCT_VALUE`**: Während `SET_STRUCT_VALUE` einen einzelnen Wert innerhalb einer bestehenden Struktur dynamisch ändert, erzeugt `STRUCT_MUX` eine komplett neue Struktur aus einzelnen Werten.

Metadaten

Attribut	Wert
Copyright	(c) 2020 Johannes Kepler University Linz
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	eclipse4diac::convert

🛠️ Zugehörige Übungen

- Uebung_051
- Uebung_052
- Uebung_121
- SIN_TO_PCAN_Callback_DWord
- SIN_TO_PCAN_Callback_Byte
- Uebung_12x_sub

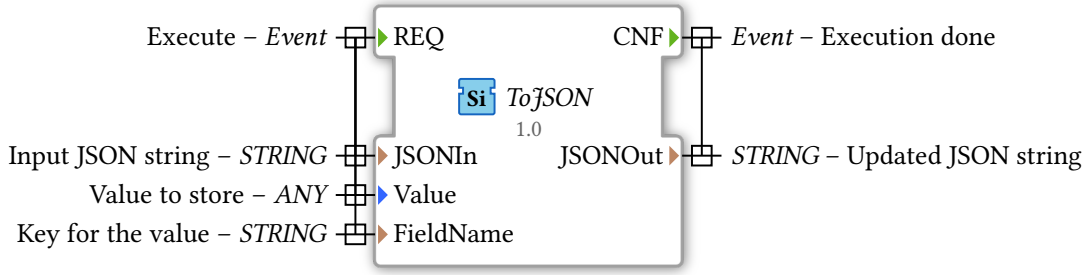
Fazit

`STRUCT_MUX` ist ein fundamentaler und sehr praktischer Baustein für die Arbeit mit Datenstrukturen in 4diac. Seine Fähigkeit, aus einzelnen Werten eine beliebige Struktur zu generieren, vereinfacht die Anwendungslogik und fördert eine saubere, strukturierte Datenhaltung. Er ist das Standardwerkzeug, um Daten zu bündeln und strukturierte Informationen zu erzeugen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.2.1.20 ToJSON



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `ToJSON`-Funktionsblock fügt ein benanntes Feld mit einem Wert zu einem JSON-String hinzu. Ist `JSONIn` leer, wird ein neues JSON-Objekt mit genau diesem einen Feld erstellt — durch mehrfaches Verketten mehrerer `ToJSON`-Aufrufe (`JSONOut` des einen als `JSONIn` des nächsten) lässt sich so schrittweise ein vollständiges JSON-Objekt aufbauen.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst das Hinzufügen aus, trägt `JSONIn`, `Value` und `FieldName`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `JSONOut`.

Daten-Eingänge

- **JSONIn** (STRING): Der bestehende JSON-String, dem ein Feld hinzugefügt wird (bei leerem String wird ein neues Objekt erstellt).
- **Value** (ANY): Der zu speichernde Wert, generisch typisiert.
- **FieldName** (STRING): Der Schlüssel (Feldname), unter dem `Value` gespeichert wird.

Daten-Ausgänge

- **JSONOut** (STRING): Der aktualisierte JSON-String mit dem neu hinzugefügten Feld.

Funktionsweise

Bei Eintreffen von `REQ` prüft der Baustein `JSONIn`: Ist der String leer, wird ein neues JSON-Objekt `{"FieldName": Value}` erzeugt; andernfalls wird das Feld `FieldName` mit `Value` in das bestehende Objekt eingefügt (bzw. ein vorhandenes Feld gleichen Namens überschrieben). Das Ergebnis wird über `JSONOut` ausgegeben, anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **ANY-Eingabetyp**: `Value` akzeptiert generisch Zahlen, Zeichenketten und boolesche Werte, die entsprechend in den JSON-Datentyp übersetzt werden.
- **Verkettbar**: Da `JSONOut` denselben Typ wie `JSONIn` hat, lassen sich mehrere `ToJSON`-Instanzen hintereinanderschalten, um ein JSON-Objekt mit mehreren Feldern Schritt für Schritt aufzubauen.
- **Leerer `JSONIn` erzeugt neues Objekt**: Kein separater „Create"-Baustein nötig — der erste Aufruf einer Verkettung kann direkt mit leerem `JSONIn` beginnen.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur Feld-Einfügung und zu `CNF`.

Anwendungsszenarien

- **Aufbau von JSON-Nutzlasten**: Für den Versand an eine REST-API oder ein MQTT-Topic wird schrittweise ein JSON-Objekt aus mehreren Prozesswerten zusammengesetzt.
- **Protokollierung**: Strukturierte Log-Einträge im JSON-Format, bei denen einzelne Felder aus verschiedenen Programmteilen ergänzt werden.

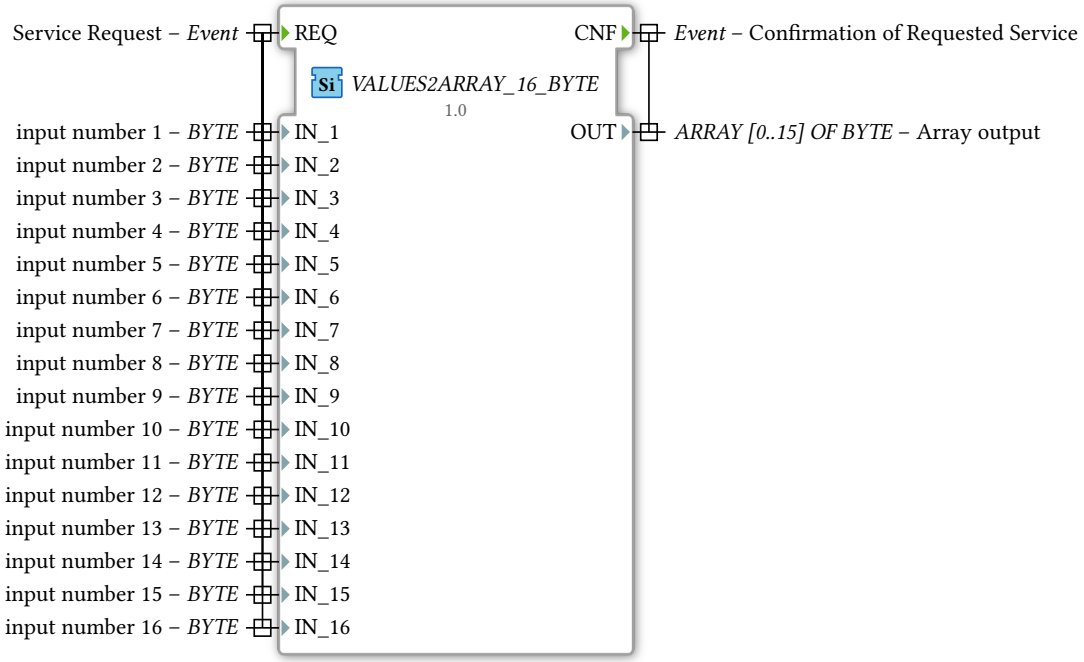
⚖️ Vergleich mit ähnlichen Bausteinen

- **ReadJSON**: die Gegenrichtung — extrahiert ein Feld aus einem JSON-String.
- **`SET_STRUCT_VALUE`**: setzt ein Feld in einer IEC-61131-3-STRUCT statt in einem JSON-String.

Fazit

`ToJSON` ermöglicht den schrittweisen, verkettbaren Aufbau von JSON-Strings aus einzelnen Feld/Wert-Paaren und eignet sich für die Erstellung von JSON-Nutzlasten in 4diac-Anwendungen, ohne eine vollständige Struktur-Typdefinition zu benötigen.

2.2.1.21 VALUES2ARRAY_16_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `VALUES2ARRAY_16_BYTE` fasst 16 einzelne `BYTE`-Skalarvariablen `IN_1`…`IN_16` zu einem `BYTE`-Array der Größe 16 zusammen. Er ist die Umkehrung von `ARRAY2VALUES_16_BYTE` und gehört zur generischen `GEN_ARRAY2ARRAY`-Familie (vgl. VALUES2ARRAY_2_LREAL).

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Zusammenführung aus, trägt `IN_1`, `IN_2`, `IN_3`, `IN_4`, `IN_5`, `IN_6`, `IN_7`, `IN_8`, `IN_9`, `IN_10`, `IN_11`, `IN_12`, `IN_13`, `IN_14`, `IN_15`, `IN_16`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT`.

Daten-Eingänge

- `IN_1`, `IN_2`, `IN_3`, `IN_4`, `IN_5`, `IN_6`, `IN_7`, `IN_8`, `IN_9`, `IN_10`, `IN_11`, `IN_12`, `IN_13`, `IN_14`, `IN_15`, `IN_16` (`BYTE`): Die 16 einzelnen Werte (8-Bit-Bitmuster), die zum Array zusammengefasst werden.

Daten-Ausgänge

- **OUT** (`BYTE`, Array-Größe 16): `OUT[i-1]` entspricht `IN_i`.

Funktionsweise

Bei Eintreffen von `REQ` wird jeder Eingangswert `IN_i` auf das entsprechende Element `OUT[i-1]` geschrieben (`IN_1` ⟶ `OUT[0]`, …, `IN_16` ⟶ `OUT[15]`), anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY'`, dieselbe C++-Basis wie VALUES2ARRAY_2_LREAL und die übrigen `VALUES2ARRAY_*`-Varianten.
- **Feste Größe 16**: Für andere Array-Größen desselben Typs siehe `VALUES2ARRAY_2_BYTE`, `VALUES2ARRAY_4_BYTE`, `VALUES2ARRAY_8_BYTE`, `VALUES2ARRAY_32_BYTE`.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur vollständigen Zusammenführung und zu `CNF`.

Anwendungsszenarien

- **Array-Aufbau aus Einzelwerten**: Mehrere diskrete BYTE-Variablen sollen als Array an einen nachgeschalteten Baustein übergeben werden, der ein Array-Interface erwartet.
- **Schnittstellenanpassung** zwischen variablenbasierten und array-basierten Bausteinschnittstellen.

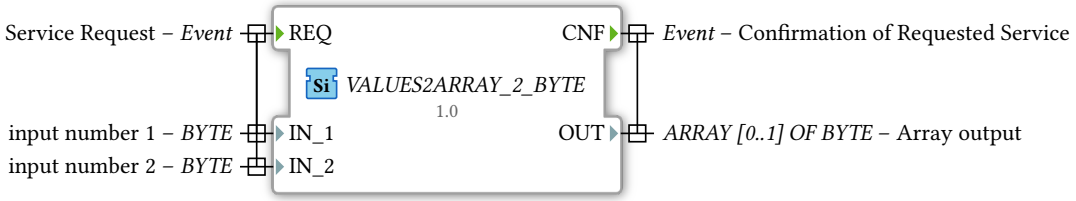
⚖️ Vergleich mit ähnlichen Bausteinen

- VALUES2ARRAY_2_LREAL: dieselbe generische Implementierung für den Datentyp LREAL.
- **ARRAY2VALUES_16_BYTE**: die Umkehrrichtung — zerlegt ein Array in 16 Einzelwerte.

Fazit

VALUES2ARRAY_16_BYTE liefert eine einfache, generisch implementierte Zusammenführung von 16 BYTE-Einzelwerten zu einem Array und eignet sich zur Anpassung variablenbasierter an array-basierte Schnittstellen.

2.2.1.22 VALUES2ARRAY_2_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der VALUES2ARRAY_2_BYTE fasst 2 einzelne BYTE-Skalarvariablen IN_1…IN_2 zu einem BYTE-Array der Größe 2 zusammen. Er ist die Umkehrung von ARRAY2VALUES_2_BYTE und gehört zur generischen GEN_ARRAY2ARRAY-Familie (vgl. VALUES2ARRAY_2_LREAL).

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Zusammenführung aus, trägt IN_1, IN_2.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt OUT.

Daten-Eingänge

- IN_1, IN_2 (BYTE): Die 2 einzelnen Werte (8-Bit-Bitmuster), die zum Array zusammengefasst werden.

Daten-Ausgänge

- **OUT** (BYTE, Array-Größe 2): OUT[i-1] entspricht IN_i.

Funktionsweise

Bei Eintreffen von REQ wird jeder Eingangswert IN_i auf das entsprechende Element OUT[i-1] geschrieben (IN_1 ⟶ OUT[0], …, IN_2 ⟶ OUT[1]), anschließend wird CNF ausgelöst.

Technische Besonderheiten

- **Generische Implementierung**: eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY', dieselbe C++-Basis wie VALUES2ARRAY_2_LREAL und die übrigen VALUES2ARRAY_*-Varianten.
- **Feste Größe 2**: Für andere Array-Größen desselben Typs siehe VALUES2ARRAY_4_BYTE, VALUES2ARRAY_8_BYTE, VALUES2ARRAY_16_BYTE, VALUES2ARRAY_32_BYTE.

Zustandsübersicht

Zustandslos: jedes REQ führt unmittelbar zur vollständigen Zusammenführung und zu CNF.

Anwendungsszenarien

- **Array-Aufbau aus Einzelwerten**: Mehrere diskrete `BYTE`-Variablen sollen als Array an einen nachgeschalteten Baustein übergeben werden, der ein Array-Interface erwartet.
- **Schnittstellenanpassung** zwischen variablenbasierten und array-basierten Bausteinschnittstellen.

⚖️ Vergleich mit ähnlichen Bausteinen

- VALUES2ARRAY_2_LREAL: dieselbe generische Implementierung für den Datentyp `LREAL`.
- **`ARRAY2VALUES_2_BYTE`**: die Umkehrrichtung — zerlegt ein Array in 2 Einzelwerte.

Fazit

`VALUES2ARRAY_2_BYTE` liefert eine einfache, generisch implementierte Zusammenführung von 2 `BYTE`-Einzelwerten zu einem Array und eignet sich zur Anpassung variablenbasierter an array-basierte Schnittstellen.

2.2.1.23 VALUES2ARRAY_2_LREAL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `VALUES2ARRAY_2_LREAL` dient dazu, zwei einzelne Gleitkommazahlen vom Typ LREAL in ein Array desselben Typs zu konvertieren. Er ist als Service-Interface-Funktionsblock konzipiert, der eine einfache Schnittstelle für diese Umwandlungsaufgabe bietet.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Typ: `Event`)
  - Kommentar: Service Request
  - Verbundene Daten: `IN_1`, `IN_2`
  - Beschreibung: Dieses Ereignis löst die Konvertierung der beiden Eingangswerte `IN_1` und `IN_2` in das Array `OUT` aus.

Ereignis-Ausgänge

- **CNF** (Typ: `Event`)
  - Kommentar: Confirmation of Requested Service
  - Verbundene Daten: `OUT`
  - Beschreibung: Dieses Ereignis wird ausgelöst, nachdem die Konvertierung erfolgreich abgeschlossen wurde und das Ausgangsarray `OUT` aktualisiert ist.

Daten-Eingänge

- **IN_1** (Typ: `LREAL`)
  - Kommentar: input number 1
  - Beschreibung: Die erste Gleitkommazahl, die in das Array aufgenommen werden soll.
- **IN_2** (Typ: `LREAL`)
  - Kommentar: input number 2
  - Beschreibung: Die zweite Gleitkommazahl, die in das Array aufgenommen werden soll.

Daten-Ausgänge

- **OUT** (Typ: `LREAL`, Größe: 2)
  - Kommentar: Array output
  - Beschreibung: Ein Array vom Typ LREAL, das die Werte von `IN_1` und `IN_2` enthält. `IN_1` belegt den ersten, `IN_2` den zweiten Index des Arrays.

Adapter

Dieser Funktionsblock verfügt über keine Adapter-Schnittstellen.

Funktionsweise

Wenn ein `REQ`-Ereignis am Eingang des Funktionsblocks empfangen wird, liest der `VALUES2ARRAY_2_LREAL` die aktuellen Werte der Daten-Eingänge `IN_1` und `IN_2` ein. Diese beiden Werte werden anschließend in das Ausgangsarray `OUT` übertragen, wobei `IN_1` dem Element `OUT[0]` und `IN_2` dem Element `OUT[1]` zugewiesen wird. Nach erfolgreicher Durchführung dieser Operation wird ein `CNF`-Ereignis ausgegeben, um den Abschluss der Anforderung zu signalisieren.

Technische Besonderheiten

- Der Funktionsblock ist auf die Konvertierung von genau zwei LREAL-Werten spezialisiert.
- Das Ausgangsarray `OUT` ist fest auf die Größe 2 definiert.
- Die Verarbeitung ist ereignisgesteuert, was eine explizite Anforderung des Konvertierungsprozesses ermöglicht.

Zustandsübersicht

Der `VALUES2ARRAY_2_LREAL` ist ein zustandsloser Funktionsblock. Jede `REQ`-Anforderung wird unabhängig voneinander verarbeitet, ohne dass interne Zustände über Anfragen hinweg beibehalten werden. Die Funktion hängt ausschließlich von den Eingangswerten zum Zeitpunkt des `REQ`-Ereignisses ab.

Anwendungsszenarien

- **Datenbündelung**: Wenn zwei separate LREAL-Werte als ein zusammenhängendes Datenpaket (Array) an andere Funktionsblöcke übergeben werden müssen, die Arrays als Eingabe erwarten.
- **Kompatibilität mit Array-Schnittstellen**: Anpassung von skalaren Daten an Schnittstellen, die explizit Arrays erfordern.
- **Vorverarbeitung für mathematische Operationen**: Vorbereitung von Datenpaaren für Operationen, die auf Array-Basis arbeiten, wie z.B. Vektoraddition oder Skalierung.

⚖️ Vergleich mit ähnlichen Bausteinen

Dieser Funktionsblock ist spezifisch für die Konvertierung von zwei LREAL-Werten in ein LREAL-Array. Allgemeine „Werte-zu-Array“-Bausteine könnten flexibler in Bezug auf die Anzahl und den Datentyp der Eingänge sein, aber dieser Baustein bietet eine einfache, spezialisierte Lösung für den angegebenen Anwendungsfall. Für eine größere Anzahl von Werten oder unterschiedliche Datentypen wären entsprechende, speziell angepasste Bausteine erforderlich.

Metadaten

Attribut	Wert
Copyright	(c) 2014 Profactor GmbH
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner), 1.0 (2014-07-09, Matthias Plasch)
4diac-Paket	eclipse4diac::convert

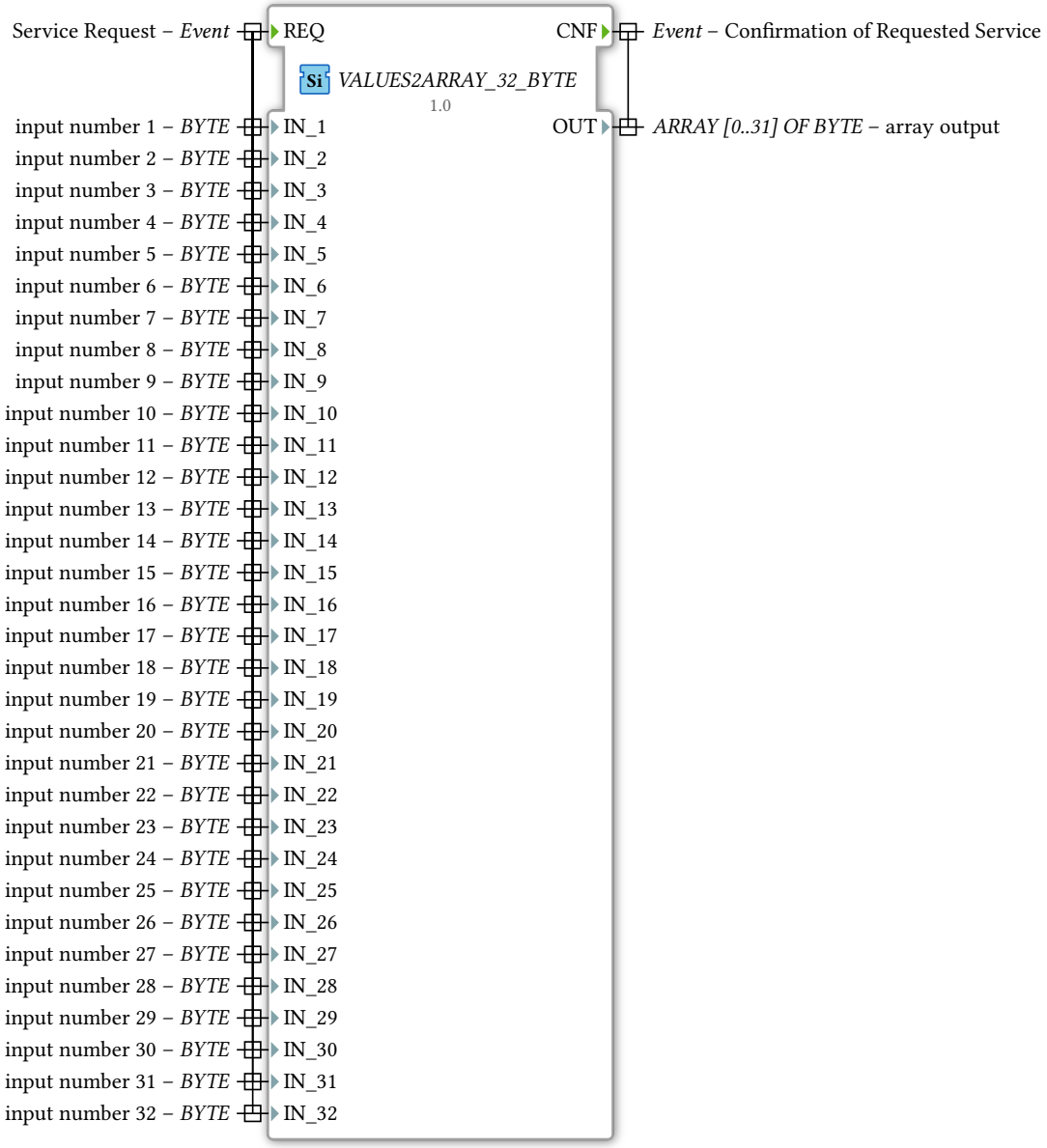
Fazit

Der `VALUES2ARRAY_2_LREAL` ist ein nützlicher und effizienter Funktionsblock für die spezifische Aufgabe, zwei LREAL-Werte in ein fest dimensioniertes LREAL-Array umzuwandeln. Seine Einfachheit und ereignisgesteuerte Arbeitsweise machen ihn zu einem klaren Bestandteil in Szenarien, wo eine solche Datenbündelung erforderlich ist, und tragen zur Modularität und Lesbarkeit der Applikation bei.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.2.1.24 VALUES2ARRAY_32_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `VALUES2ARRAY_32_BYTE` fasst 32 einzelne `BYTE`-Skalarvariablen `IN_1`…`IN_32` zu einem `BYTE`-Array der Größe 32 zusammen. Er ist die Umkehrung von `ARRAY2VALUES_32_BYTE` und gehört zur generischen `GEN_ARRAY2ARRAY`-Familie (vgl. VALUES2ARRAY_2_LREAL).

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Zusammenführung aus, trägt `IN_1`, `IN_2`, `IN_3`, `IN_4`, `IN_5`, `IN_6`, `IN_7`, `IN_8`, `IN_9`, `IN_10`, `IN_11`, `IN_12`, `IN_13`, `IN_14`, `IN_15`, `IN_16`, `IN_17`, `IN_18`, `IN_19`, `IN_20`, `IN_21`, `IN_22`, `IN_23`, `IN_24`, `IN_25`, `IN_26`, `IN_27`, `IN_28`, `IN_29`, `IN_30`, `IN_31`, `IN_32`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT`.

Daten-Eingänge

- `IN_1`, `IN_2`, `IN_3`, `IN_4`, `IN_5`, `IN_6`, `IN_7`, `IN_8`, `IN_9`, `IN_10`, `IN_11`, `IN_12`, `IN_13`, `IN_14`, `IN_15`, `IN_16`, `IN_17`, `IN_18`, `IN_19`, `IN_20`, `IN_21`, `IN_22`, `IN_23`, `IN_24`, `IN_25`, `IN_26`, `IN_27`, `IN_28`, `IN_29`, `IN_30`, `IN_31`, `IN_32` (`BYTE`): Die 32 einzelnen Werte (8-Bit-Bitmuster), die zum Array zusammengefasst werden.

Daten-Ausgänge

- **OUT** (`BYTE`, Array-Größe 32): `OUT[i-1]` entspricht `IN_i`.

Funktionsweise

Bei Eintreffen von `REQ` wird jeder Eingangswert `IN_i` auf das entsprechende Element `OUT[i-1]` geschrieben (`IN_1` ⟶ `OUT[0]`, ..., `IN_32` ⟶ `OUT[31]`), anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY'`, dieselbe C++-Basis wie VALUES2ARRAY_2_LREAL und die übrigen `VALUES2ARRAY_*`-Varianten.
- **Feste Größe 32**: Für andere Array-Größen desselben Typs siehe `VALUES2ARRAY_2_BYTE`, `VALUES2ARRAY_4_BYTE`, `VALUES2ARRAY_8_BYTE`, `VALUES2ARRAY_16_BYTE`.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur vollständigen Zusammenführung und zu `CNF`.

Anwendungsszenarien

- **Array-Aufbau aus Einzelwerten**: Mehrere diskrete `BYTE`-Variablen sollen als Array an einen nachgeschalteten Baustein übergeben werden, der ein Array-Interface erwartet.
- **Schnittstellenanpassung** zwischen variablenbasierten und array-basierten Bausteinschnittstellen.

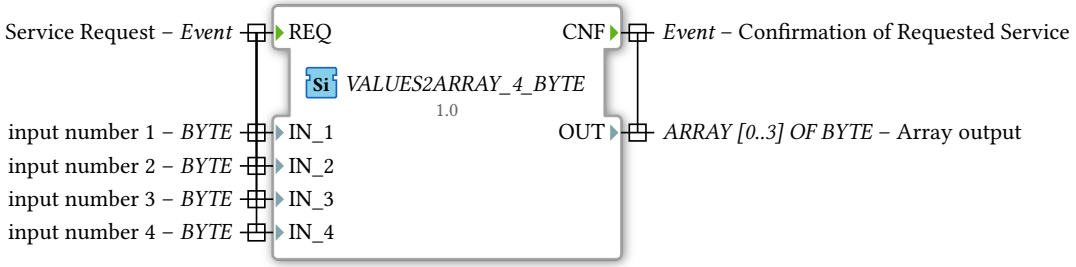
⚖️ Vergleich mit ähnlichen Bausteinen

- **VALUES2ARRAY_2_LREAL**: dieselbe generische Implementierung für den Datentyp `LREAL`.
- **`ARRAY2VALUES_32_BYTE`**: die Umkehrrichtung — zerlegt ein Array in 32 Einzelwerte.

Fazit

`VALUES2ARRAY_32_BYTE` liefert eine einfache, generisch implementierte Zusammenführung von 32 `BYTE`-Einzelwerten zu einem Array und eignet sich zur Anpassung variablenbasierter an array-basierte Schnittstellen.

2.2.1.25 VALUES2ARRAY_4_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `VALUES2ARRAY_4_BYTE` fasst 4 einzelne `BYTE`-Skalarvariablen `IN_1`...`IN_4` zu einem `BYTE`-Array der Größe 4 zusammen. Er ist die Umkehrung von `ARRAY2VALUES_4_BYTE` und gehört zur generischen `GEN_ARRAY2ARRAY`-Familie (vgl. VALUES2ARRAY_2_LREAL).

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Zusammenführung aus, trägt `IN_1`, `IN_2`, `IN_3`, `IN_4`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT`.

Daten-Eingänge

- `IN_1`, `IN_2`, `IN_3`, `IN_4` (`BYTE`): Die 4 einzelnen Werte (8-Bit-Bitmuster), die zum Array zusammengefasst werden.

Daten-Ausgänge

- **OUT** (`BYTE`, Array-Größe 4): `OUT[i-1]` entspricht `IN_i`.

Funktionsweise

Bei Eintreffen von `REQ` wird jeder Eingangswert `IN_i` auf das entsprechende Element `OUT[i-1]` geschrieben (`IN_1` ⟶ `OUT[0]`, ..., `IN_4` ⟶ `OUT[3]`), anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY'`, dieselbe C++-Basis wie VALUES2ARRAY_2_LREAL und die übrigen `VALUES2ARRAY_*`-Varianten.
- **Feste Größe 4**: Für andere Array-Größen desselben Typs siehe `VALUES2ARRAY_2_BYTE`, `VALUES2ARRAY_8_BYTE`, `VALUES2ARRAY_16_BYTE`, `VALUES2ARRAY_32_BYTE`.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur vollständigen Zusammenführung und zu `CNF`.

Anwendungsszenarien

- **Array-Aufbau aus Einzelwerten**: Mehrere diskrete `BYTE`-Variablen sollen als Array an einen nachgeschalteten Baustein übergeben werden, der ein Array-Interface erwartet.
- **Schnittstellenanpassung** zwischen variablenbasierten und array-basierten Bausteinschnittstellen.

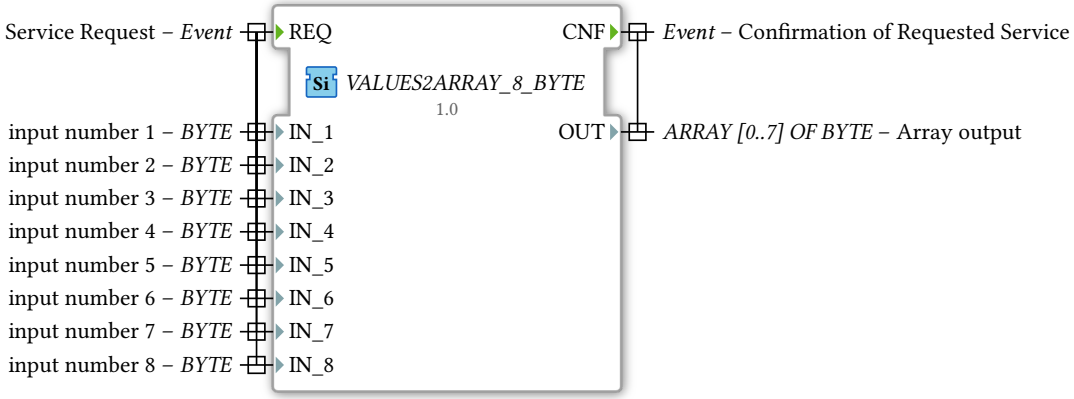
⚖️ Vergleich mit ähnlichen Bausteinen

- VALUES2ARRAY_2_LREAL: dieselbe generische Implementierung für den Datentyp `LREAL`.
- **`ARRAY2VALUES_4_BYTE`**: die Umkehrrichtung — zerlegt ein Array in 4 Einzelwerte.

Fazit

`VALUES2ARRAY_4_BYTE` liefert eine einfache, generisch implementierte Zusammenführung von 4 `BYTE`-Einzelwerten zu einem Array und eignet sich zur Anpassung variablenbasierter an array-basierte Schnittstellen.

2.2.1.26 VALUES2ARRAY_8_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `VALUES2ARRAY_8_BYTE` fasst 8 einzelne `BYTE`-Skalarvariablen `IN_1`...`IN_8` zu einem `BYTE`-Array der Größe 8 zusammen. Er ist die Umkehrung von `ARRAY2VALUES_8_BYTE` und gehört zur generischen `GEN_ARRAY2ARRAY`-Familie (vgl. VALUES2ARRAY_2_LREAL).

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Zusammenführung aus, trägt `IN_1`, `IN_2`, `IN_3`, `IN_4`, `IN_5`, `IN_6`, `IN_7`, `IN_8`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT`.

Daten-Eingänge

- `IN_1`, `IN_2`, `IN_3`, `IN_4`, `IN_5`, `IN_6`, `IN_7`, `IN_8` (`BYTE`): Die 8 einzelnen Werte (8-Bit-Bitmuster), die zum Array zusammengefasst werden.

Daten-Ausgänge

- **OUT** (`BYTE`, Array-Größe 8): `OUT[i-1]` entspricht `IN_i`.

Funktionsweise

Bei Eintreffen von `REQ` wird jeder Eingangswert `IN_i` auf das entsprechende Element `OUT[i-1]` geschrieben (`IN_1` ⟶ `OUT[0]`, ..., `IN_8` ⟶ `OUT[7]`), anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY'`, dieselbe C++-Basis wie VALUES2ARRAY_2_LREAL und die übrigen `VALUES2ARRAY_*`-Varianten.
- **Feste Größe 8**: Für andere Array-Größen desselben Typs siehe `VALUES2ARRAY_2_BYTE`, `VALUES2ARRAY_4_BYTE`, `VALUES2ARRAY_16_BYTE`, `VALUES2ARRAY_32_BYTE`.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur vollständigen Zusammenführung und zu `CNF`.

Anwendungsszenarien

- **Array-Aufbau aus Einzelwerten**: Mehrere diskrete `BYTE`-Variablen sollen als Array an einen nachgeschalteten Baustein übergeben werden, der ein Array-Interface erwartet.
- **Schnittstellenanpassung** zwischen variablenbasierten und array-basierten Bausteinschnittstellen.

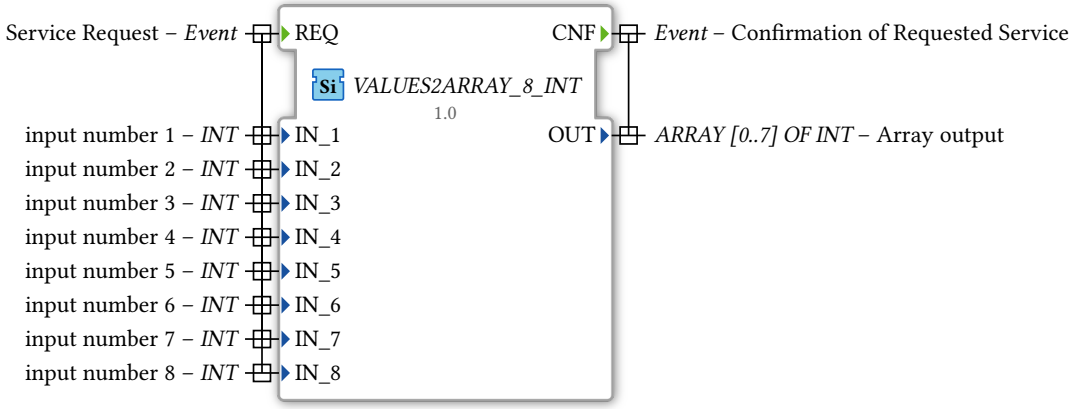
⚖️ Vergleich mit ähnlichen Bausteinen

- VALUES2ARRAY_2_LREAL: dieselbe generische Implementierung für den Datentyp `LREAL`.
- **`ARRAY2VALUES_8_BYTE`**: die Umkehrrichtung — zerlegt ein Array in 8 Einzelwerte.

Fazit

`VALUES2ARRAY_8_BYTE` liefert eine einfache, generisch implementierte Zusammenführung von 8 `BYTE`-Einzelwerten zu einem Array und eignet sich zur Anpassung variablenbasierter an array-basierte Schnittstellen.

2.2.1.27 VALUES2ARRAY_8_INT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der VALUES2ARRAY_8_INT fasst 8 einzelne INT-Skalarvariablen IN_1...IN_8 zu einem INT-Array der Größe 8 zusammen. Er ist die Umkehrung von ARRAY2VALUES_8_INT und gehört zur generischen GEN_ARRAY2ARRAY-Familie (vgl. VALUES2ARRAY_2_LREAL).

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Zusammenführung aus, trägt IN_1, IN_2, IN_3, IN_4, IN_5, IN_6, IN_7, IN_8.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt OUT.

Daten-Eingänge

- IN_1, IN_2, IN_3, IN_4, IN_5, IN_6, IN_7, IN_8 (INT): Die 8 einzelnen Werte (16-Bit-Ganzzahl (vorzeichenbehaftet)), die zum Array zusammengefasst werden.

Daten-Ausgänge

- **OUT** (INT, Array-Größe 8): OUT[i-1] entspricht IN_i.

Funktionsweise

Bei Eintreffen von REQ wird jeder Eingangswert IN_i auf das entsprechende Element OUT[i-1] geschrieben (IN_1 ⟶ OUT[0], ..., IN_8 ⟶ OUT[7]), anschließend wird CNF ausgelöst.

Technische Besonderheiten

- **Generische Implementierung**: eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY', dieselbe C++-Basis wie VALUES2ARRAY_2_LREAL und die übrigen VALUES2ARRAY_*-Varianten.
- **Feste Größe 8**: Für andere Array-Größen desselben Typs siehe .

Zustandsübersicht

Zustandslos: jedes REQ führt unmittelbar zur vollständigen Zusammenführung und zu CNF.

Anwendungsszenarien

- **Array-Aufbau aus Einzelwerten**: Mehrere diskrete INT-Variablen sollen als Array an einen nachgeschalteten Baustein übergeben werden, der ein Array-Interface erwartet.
- **Schnittstellenanpassung** zwischen variablenbasierten und array-basierten Bausteinschnittstellen.

⚖️ Vergleich mit ähnlichen Bausteinen

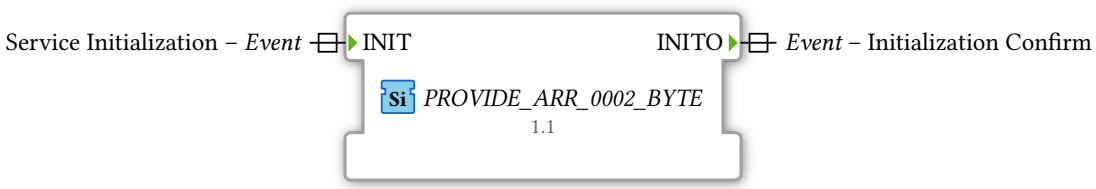
- **VALUES2ARRAY_2_LREAL**: dieselbe generische Implementierung für den Datentyp LREAL.
- **ARRAY2VALUES_8_INT**: die Umkehrrichtung — zerlegt ein Array in 8 Einzelwerte.

Fazit

VALUES2ARRAY_8_INT liefert eine einfache, generisch implementierte Zusammenführung von 8 INT-Einzelwerten zu einem Array und eignet sich zur Anpassung variablenbasierter an array-basierte Schnittstellen.

2.2.1.28 Providers

PROVIDE_ARR_0002_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der PROVIDE_ARR_0002_BYTE stellt einen BYTE-Array-Container fester Größe 2 als InOut-Variable D1 bereit. Er gehört zur generischen GEN_INIT_ARR-Familie: eine Reihe baugleicher Bausteine, die sich nur in Array-Größe und Datentyp unterscheiden, und dient dazu, ein zur Entwurfszeit mit Startwerten vorbelegbares Array in einer 4diac-Anwendung verfügbar zu machen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, trägt D1.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung, trägt D1.

Ein-/Ausgangsvariablen (InOut)

- **D1** (BYTE, Array-Größe 2, Index 0…1): Der Array-Container (8-Bit-Bitmuster je Element). Kann im FB-Instanzeditor mit Startwerten vorbelegt werden (z. B. [16#01, 16#00, ...] für BYTE).

Funktionsweise

Bei Eintreffen von INIT liest der Baustein das InOut-Array D1 (das bereits zur Entwurfszeit mit Startwerten vorbelegt sein kann) und bestätigt dies unmittelbar über INITO, wobei D1 unverändert weitergereicht wird. Der Baustein führt keine eigene Berechnung durch — er dient als deklarativer Container, dessen Inhalt über die FB-Instanzparametrierung vorgegeben wird.

Technische Besonderheiten

- **Generische Implementierung**: eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR', dieselbe C++-Basis wie alle anderen PROVIDE_ARR_*-Varianten; Array-Größe und Datentyp werden pro Instanziierung über die Typdefinition festgelegt.
- **InOut statt getrennter Input/Output**: D1 ist eine einzige InOut-Variable — Vorbelegung und Bereitstellung erfolgen über dieselbe Variable, es gibt keine separate Kopie.
- **Feste Größe 2**: Weitere Größen derselben oder anderer Datentypen: PROVIDE_ARR_0007_BYTE, PROVIDE_ARR_0008_BYTE, PROVIDE_ARR_0014_BYTE, PROVIDE_ARR_0016_BYTE.

Zustandsübersicht

Zustandslos: INIT liest D1 und bestätigt sofort über INITO, ohne den Inhalt zu verändern.

Anwendungsszenarien

- **Konfigurationsarrays**: Bereitstellung fest vorbelegter Tabellen (z. B. Kalibrierwerte, Lookup-Tabellen, Protokoll-Header-Bytes) als wiederverwendbarer, zur Entwurfszeit parametrierbarer Baustein.
- **Testdatengenerierung**: Bereitstellung eines vordefinierten BYTE-Arrays für Testaufbauten, ohne das Array manuell in jeder Applikation neu zu verdrahten.

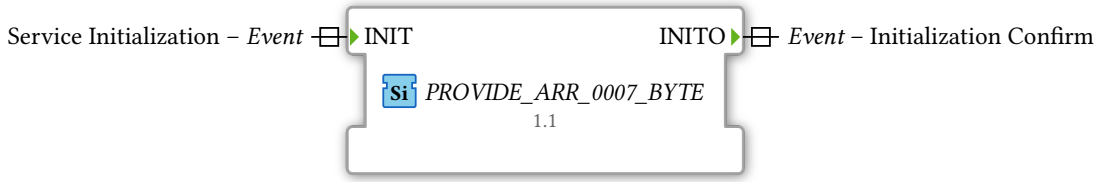
⚖️ Vergleich mit ähnlichen Bausteinen

- PROVIDE_ARR_0007_BYTE, PROVIDE_ARR_0008_BYTE, PROVIDE_ARR_0014_BYTE, PROVIDE_ARR_0016_BYTE: dieselbe generische Implementierung mit anderer Array-Größe bzw. anderem Datentyp.
- **ARRAY2ARRAY_2_BYTE**: kopiert ein bereits vorhandenes Array, ohne es bereitzustellen/vorzubelegen.

Fazit

PROVIDE_ARR_0002_BYTE liefert einen einfachen, zur Entwurfszeit vorbelegbaren BYTE-Array-Container fester Größe 2 und eignet sich für Konfigurations- und Testdatenarrays, die wiederverwendbar in 4diac-Applikationen eingebunden werden sollen.

PROVIDE_ARR_0007_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `PROVIDE_ARR_0007_BYTE` stellt einen `BYTE`-Array-Container fester Größe 7 als `InOut`-Variable `D1` bereit. Er gehört zur generischen `GEN_INIT_ARR`-Familie: eine Reihe baugleicher Bausteine, die sich nur in Array-Größe und Datentyp unterscheiden, und dient dazu, ein zur Entwurfszeit mit Startwerten vorbelegbares Array in einer 4diac-Anwendung verfügbar zu machen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, trägt `D1`.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung, trägt `D1`.

Ein-/Ausgangsvariablen (InOut)

- **D1** (`BYTE`, Array-Größe 7, Index 0…6): Der Array-Container (8-Bit-Bitmuster je Element). Kann im FB-Instanzeditor mit Startwerten vorbelegt werden (z. B. `[16#01, 16#00, ...]` für `BYTE`).

Funktionsweise

Bei Eintreffen von `INIT` liest der Baustein das `InOut`-Array `D1` (das bereits zur Entwurfszeit mit Startwerten vorbelegt sein kann) und bestätigt dies unmittelbar über `INITO`, wobei `D1` unverändert weitergereicht wird. Der Baustein führt keine eigene Berechnung durch — er dient als deklarativer Container, dessen Inhalt über die FB-Instanzparametrierung vorgegeben wird.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, dieselbe C++-Basis wie alle anderen `PROVIDE_ARR_*`-Varianten; Array-Größe und Datentyp werden pro Instanziierung über die Typdefinition festgelegt.
- **InOut statt getrennter Input/Output**: `D1` ist eine einzige `InOut`-Variable — Vorbelegung und Bereitstellung erfolgen über dieselbe Variable, es gibt keine separate Kopie.
- **Feste Größe 7**: Weitere Größen derselben oder anderer Datentypen: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`, `PROVIDE_ARR_0016_BYTE`.

Zustandsübersicht

Zustandslos: `INIT` liest `D1` und bestätigt sofort über `INITO`, ohne den Inhalt zu verändern.

Anwendungsszenarien

- **Konfigurationsarrays**: Bereitstellung fest vorbelegter Tabellen (z. B. Kalibrierwerte, Lookup-Tabellen, Protokoll-Header-Bytes) als wiederverwendbarer, zur Entwurfszeit parametrierbarer Baustein.
- **Testdatengenerierung**: Bereitstellung eines vordefinierten `BYTE`-Arrays für Testaufbauten, ohne das Array manuell in jeder Applikation neu zu verdrahten.

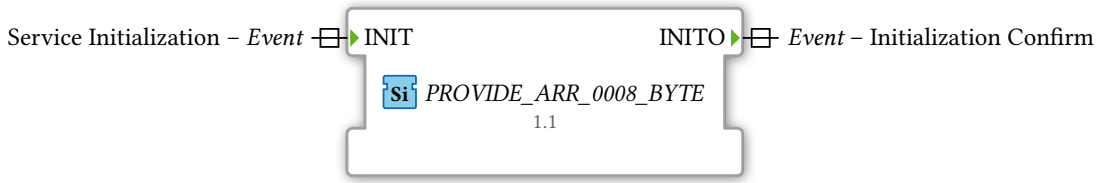
⚖️ Vergleich mit ähnlichen Bausteinen

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`, `PROVIDE_ARR_0016_BYTE`: dieselbe generische Implementierung mit anderer Array-Größe bzw. anderem Datentyp.
- **`ARRAY2ARRAY_2_BYTE`**: kopiert ein bereits vorhandenes Array, ohne es bereitzustellen/vorzubelegen.

Fazit

`PROVIDE_ARR_0007_BYTE` liefert einen einfachen, zur Entwurfszeit vorbelegbaren `BYTE`-Array-Container fester Größe 7 und eignet sich für Konfigurations- und Testdatenarrays, die wiederverwendbar in 4diac-Applikationen eingebunden werden sollen.

PROVIDE_ARR_0008_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `PROVIDE_ARR_0008_BYTE` stellt einen `BYTE`-Array-Container fester Größe 8 als `InOut`-Variable `D1` bereit. Er gehört zur generischen `GEN_INIT_ARR`-Familie: eine Reihe baugleicher Bausteine, die sich nur in Array-Größe und Datentyp unterscheiden, und dient dazu, ein zur Entwurfszeit mit Startwerten vorbelegbares Array in einer 4diac-Anwendung verfügbar zu machen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, trägt `D1`.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung, trägt `D1`.

Ein-/Ausgangsvariablen (InOut)

- **D1** (`BYTE`, Array-Größe 8, Index 0…7): Der Array-Container (8-Bit-Bitmuster je Element). Kann im FB-Instanzeditor mit Startwerten vorbelegt werden (z. B. `[16#01, 16#00, ...]` für `BYTE`).

Funktionsweise

Bei Eintreffen von `INIT` liest der Baustein das `InOut`-Array `D1` (das bereits zur Entwurfszeit mit Startwerten vorbelegt sein kann) und bestätigt dies unmittelbar über `INITO`, wobei `D1` unverändert weitergereicht wird. Der Baustein führt keine eigene Berechnung durch — er dient als deklarativer Container, dessen Inhalt über die FB-Instanzparametrierung vorgegeben wird.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, dieselbe C++-Basis wie alle anderen `PROVIDE_ARR_*`-Varianten; Array-Größe und Datentyp werden pro Instanziierung über die Typdefinition festgelegt.
- **InOut statt getrennter Input/Output**: `D1` ist eine einzige `InOut`-Variable — Vorbelegung und Bereitstellung erfolgen über dieselbe Variable, es gibt keine separate Kopie.
- **Feste Größe 8**: Weitere Größen derselben oder anderer Datentypen: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0014_BYTE`, `PROVIDE_ARR_0016_BYTE`.

Zustandsübersicht

Zustandslos: `INIT` liest `D1` und bestätigt sofort über `INITO`, ohne den Inhalt zu verändern.

Anwendungsszenarien

- **Konfigurationsarrays**: Bereitstellung fest vorbelegter Tabellen (z. B. Kalibrierwerte, Lookup-Tabellen, Protokoll-Header-Bytes) als wiederverwendbarer, zur Entwurfszeit parametrierbarer Baustein.
- **Testdatengenerierung**: Bereitstellung eines vordefinierten `BYTE`-Arrays für Testaufbauten, ohne das Array manuell in jeder Applikation neu zu verdrahten.

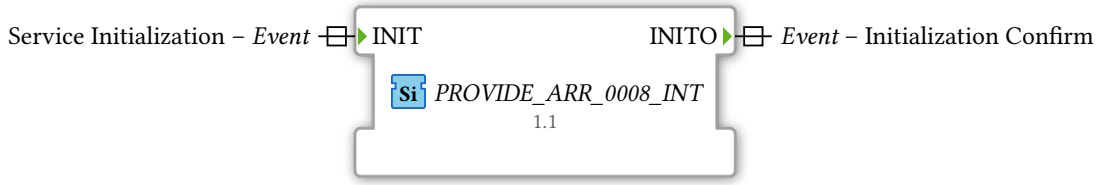
⚖️ Vergleich mit ähnlichen Bausteinen

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0014_BYTE`, `PROVIDE_ARR_0016_BYTE`: dieselbe generische Implementierung mit anderer Array-Größe bzw. anderem Datentyp.
- **`ARRAY2ARRAY_8_BYTE`**: kopiert ein bereits vorhandenes Array, ohne es bereitzustellen/vorzubelegen.

Fazit

`PROVIDE_ARR_0008_BYTE` liefert einen einfachen, zur Entwurfszeit vorbelegbaren `BYTE`-Array-Container fester Größe 8 und eignet sich für Konfigurations- und Testdatenarrays, die wiederverwendbar in 4diac-Applikationen eingebunden werden sollen.

PROVIDE_ARR_0008_INT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `PROVIDE_ARR_0008_INT` stellt einen `INT`-Array-Container fester Größe 8 als `InOut`-Variable `D1` bereit. Er gehört zur generischen `GEN_INIT_ARR`-Familie: eine Reihe baugleicher Bausteine, die sich nur in Array-Größe und Datentyp unterscheiden, und dient dazu, ein zur Entwurfszeit mit Startwerten vorbelegbares Array in einer 4diac-Anwendung verfügbar zu machen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, trägt `D1`.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung, trägt `D1`.

Ein-/Ausgangsvariablen (InOut)

- **D1** (`INT`, Array-Größe 8, Index 0…7): Der Array-Container (16-Bit-Ganzzahl (vorzeichenbehaftet) je Element). Kann im FB-Instanzeditor mit Startwerten vorbelegt werden (z. B. `[16#01, 16#00, ...]` für `BYTE`).

Funktionsweise

Bei Eintreffen von `INIT` liest der Baustein das `InOut`-Array `D1` (das bereits zur Entwurfszeit mit Startwerten vorbelegt sein kann) und bestätigt dies unmittelbar über `INITO`, wobei `D1` unverändert weitergereicht wird. Der Baustein führt keine eigene Berechnung durch — er dient als deklarativer Container, dessen Inhalt über die FB-Instanzparametrierung vorgegeben wird.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, dieselbe C++-Basis wie alle anderen `PROVIDE_ARR_*`-Varianten; Array-Größe und Datentyp werden pro Instanziierung über die Typdefinition festgelegt.
- **InOut statt getrennter Input/Output**: `D1` ist eine einzige `InOut`-Variable — Vorbelegung und Bereitstellung erfolgen über dieselbe Variable, es gibt keine separate Kopie.
- **Feste Größe 8**: Weitere Größen derselben oder anderer Datentypen: keine weiteren.

Zustandsübersicht

Zustandslos: `INIT` liest `D1` und bestätigt sofort über `INITO`, ohne den Inhalt zu verändern.

Anwendungsszenarien

- **Konfigurationsarrays**: Bereitstellung fest vorbelegter Tabellen (z. B. Kalibrierwerte, Lookup-Tabellen, Protokoll-Header-Bytes) als wiederverwendbarer, zur Entwurfszeit parametrierbarer Baustein.
- **Testdatengenerierung**: Bereitstellung eines vordefinierten `INT`-Arrays für Testaufbauten, ohne das Array manuell in jeder Applikation neu zu verdrahten.

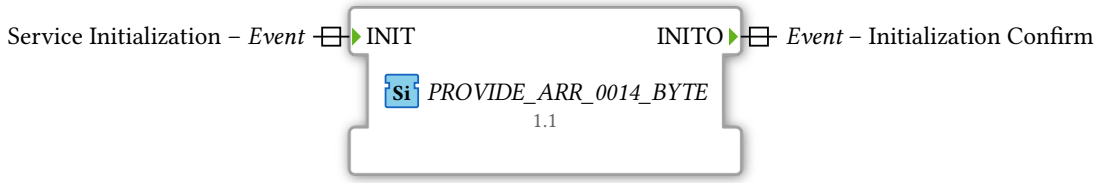
⚖️ Vergleich mit ähnlichen Bausteinen

- keine weiteren: dieselbe generische Implementierung mit anderer Array-Größe bzw. anderem Datentyp.
- **`ARRAY2ARRAY_8_INT`**: kopiert ein bereits vorhandenes Array, ohne es bereitzustellen/vorzubelegen.

Fazit

`PROVIDE_ARR_0008_INT` liefert einen einfachen, zur Entwurfszeit vorbelegbaren `INT`-Array-Container fester Größe 8 und eignet sich für Konfigurations- und Testdatenarrays, die wiederverwendbar in 4diac-Applikationen eingebunden werden sollen.

PROVIDE_ARR_0014_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `PROVIDE_ARR_0014_BYTE` stellt einen `BYTE`-Array-Container fester Größe 14 als `InOut`-Variable `D1` bereit. Er gehört zur generischen `GEN_INIT_ARR`-Familie: eine Reihe baugleicher Bausteine, die sich nur in Array-Größe und Datentyp unterscheiden, und dient dazu, ein zur Entwurfszeit mit Startwerten vorbelegbares Array in einer 4diac-Anwendung verfügbar zu machen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, trägt `D1`.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung, trägt `D1`.

Ein-/Ausgangsvariablen (InOut)

- **D1** (`BYTE`, Array-Größe 14, Index 0…13): Der Array-Container (8-Bit-Bitmuster je Element). Kann im FB-Instanzeditor mit Startwerten vorbelegt werden (z. B. `[16#01, 16#00, ...]` für `BYTE`).

Funktionsweise

Bei Eintreffen von `INIT` liest der Baustein das `InOut`-Array `D1` (das bereits zur Entwurfszeit mit Startwerten vorbelegt sein kann) und bestätigt dies unmittelbar über `INITO`, wobei `D1` unverändert weitergereicht wird. Der Baustein führt keine eigene Berechnung durch — er dient als deklarativer Container, dessen Inhalt über die FB-Instanzparametrierung vorgegeben wird.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, dieselbe C++-Basis wie alle anderen `PROVIDE_ARR_*`-Varianten; Array-Größe und Datentyp werden pro Instanziierung über die Typdefinition festgelegt.
- **InOut statt getrennter Input/Output**: `D1` ist eine einzige `InOut`-Variable — Vorbelegung und Bereitstellung erfolgen über dieselbe Variable, es gibt keine separate Kopie.
- **Feste Größe 14**: Weitere Größen derselben oder anderer Datentypen: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0016_BYTE`.

Zustandsübersicht

Zustandslos: `INIT` liest `D1` und bestätigt sofort über `INITO`, ohne den Inhalt zu verändern.

Anwendungsszenarien

- **Konfigurationsarrays**: Bereitstellung fest vorbelegter Tabellen (z. B. Kalibrierwerte, Lookup-Tabellen, Protokoll-Header-Bytes) als wiederverwendbarer, zur Entwurfszeit parametrierbarer Baustein.
- **Testdatengenerierung**: Bereitstellung eines vordefinierten `BYTE`-Arrays für Testaufbauten, ohne das Array manuell in jeder Applikation neu zu verdrahten.

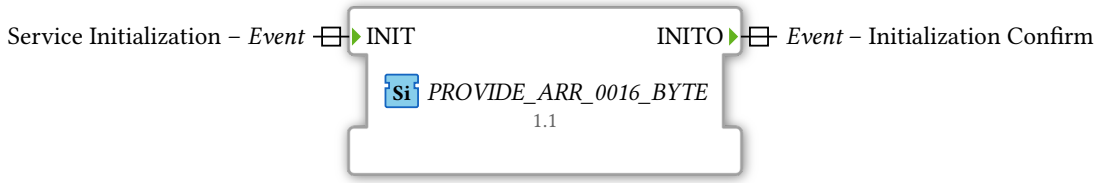
⚖️ Vergleich mit ähnlichen Bausteinen

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0016_BYTE`: dieselbe generische Implementierung mit anderer Array-Größe bzw. anderem Datentyp.
- **`ARRAY2ARRAY_2_BYTE`**: kopiert ein bereits vorhandenes Array, ohne es bereitzustellen/vorzubelegen.

Fazit

`PROVIDE_ARR_0014_BYTE` liefert einen einfachen, zur Entwurfszeit vorbelegbaren `BYTE`-Array-Container fester Größe 14 und eignet sich für Konfigurations- und Testdatenarrays, die wiederverwendbar in 4diac-Applikationen eingebunden werden sollen.

PROVIDE_ARR_0016_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `PROVIDE_ARR_0016_BYTE` stellt einen `BYTE`-Array-Container fester Größe 16 als `InOut`-Variable `D1` bereit. Er gehört zur generischen `GEN_INIT_ARR`-Familie: eine Reihe baugleicher Bausteine, die sich nur in Array-Größe und Datentyp unterscheiden, und dient dazu, ein zur Entwurfszeit mit Startwerten vorbelegbares Array in einer 4diac-Anwendung verfügbar zu machen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, trägt `D1`.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung, trägt `D1`.

Ein-/Ausgangsvariablen (InOut)

- **D1** (`BYTE`, Array-Größe 16, Index 0…15): Der Array-Container (8-Bit-Bitmuster je Element). Kann im FB-Instanzeditor mit Startwerten vorbelegt werden (z. B. `[16#01, 16#00, ...]` für `BYTE`).

Funktionsweise

Bei Eintreffen von `INIT` liest der Baustein das `InOut`-Array `D1` (das bereits zur Entwurfszeit mit Startwerten vorbelegt sein kann) und bestätigt dies unmittelbar über `INITO`, wobei `D1` unverändert weitergereicht wird. Der Baustein führt keine eigene Berechnung durch — er dient als deklarativer Container, dessen Inhalt über die FB-Instanzparametrierung vorgegeben wird.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, dieselbe C++-Basis wie alle anderen `PROVIDE_ARR_*`-Varianten; Array-Größe und Datentyp werden pro Instanziierung über die Typdefinition festgelegt.

- **InOut statt getrennter Input/Output**: `D1` ist eine einzige `InOut`-Variable — Vorbelegung und Bereitstellung erfolgen über dieselbe Variable, es gibt keine separate Kopie.
- **Feste Größe 16**: Weitere Größen derselben oder anderer Datentypen: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`.

Zustandsübersicht

Zustandslos: `INIT` liest `D1` und bestätigt sofort über `INITO`, ohne den Inhalt zu verändern.

Anwendungsszenarien

- **Konfigurationsarrays**: Bereitstellung fest vorbelegter Tabellen (z. B. Kalibrierwerte, Lookup-Tabellen, Protokoll-Header-Bytes) als wiederverwendbarer, zur Entwurfszeit parametrierbarer Baustein.
- **Testdatengenerierung**: Bereitstellung eines vordefinierten `BYTE`-Arrays für Testaufbauten, ohne das Array manuell in jeder Applikation neu zu verdrahten.

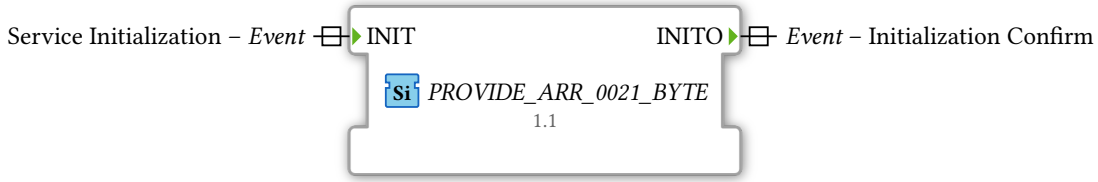
⚖️ Vergleich mit ähnlichen Bausteinen

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`: dieselbe generische Implementierung mit anderer Array-Größe bzw. anderem Datentyp.
- **`ARRAY2ARRAY_2_BYTE`**: kopiert ein bereits vorhandenes Array, ohne es bereitzustellen/vorzubelegen.

Fazit

`PROVIDE_ARR_0016_BYTE` liefert einen einfachen, zur Entwurfszeit vorbelegbaren `BYTE`-Array-Container fester Größe 16 und eignet sich für Konfigurations- und Testdatenarrays, die wiederverwendbar in 4diac-Applikationen eingebunden werden sollen.

PROVIDE_ARR_0021_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `PROVIDE_ARR_0021_BYTE` stellt einen `BYTE`-Array-Container fester Größe 21 als `InOut`-Variable `D1` bereit. Er gehört zur generischen `GEN_INIT_ARR`-Familie: eine Reihe baugleicher Bausteine, die sich nur in Array-Größe und Datentyp unterscheiden, und dient dazu, ein zur Entwurfszeit mit Startwerten vorbelegbares Array in einer 4diac-Anwendung verfügbar zu machen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, trägt `D1`.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung, trägt `D1`.

Ein-/Ausgangsvariablen (InOut)

- **D1** (`BYTE`, Array-Größe 21, Index 0…20): Der Array-Container (8-Bit-Bitmuster je Element). Kann im FB-Instanzeditor mit Startwerten vorbelegt werden (z. B. `[16#01, 16#00, ...]` für `BYTE`).

Funktionsweise

Bei Eintreffen von `INIT` liest der Baustein das `InOut`-Array `D1` (das bereits zur Entwurfszeit mit Startwerten vorbelegt sein kann) und bestätigt dies unmittelbar über `INITO`, wobei `D1` unverändert weitergereicht wird. Der Baustein führt keine eigene Berechnung durch — er dient als deklarativer Container, dessen Inhalt über die FB-Instanzparametrierung vorgegeben wird.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, dieselbe C++-Basis wie alle anderen `PROVIDE_ARR_*`-Varianten; Array-Größe und Datentyp werden pro Instanziierung über die Typdefinition festgelegt.
- **InOut statt getrennter Input/Output**: `D1` ist eine einzige `InOut`-Variable — Vorbelegung und Bereitstellung erfolgen über dieselbe Variable, es gibt keine separate Kopie.
- **Feste Größe 21**: Weitere Größen derselben oder anderer Datentypen: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`.

Zustandsübersicht

Zustandslos: `INIT` liest `D1` und bestätigt sofort über `INITO`, ohne den Inhalt zu verändern.

Anwendungsszenarien

- **Konfigurationsarrays**: Bereitstellung fest vorbelegter Tabellen (z. B. Kalibrierwerte, Lookup-Tabellen, Protokoll-Header-Bytes) als wiederverwendbarer, zur Entwurfszeit parametrierbarer Baustein.
- **Testdatengenerierung**: Bereitstellung eines vordefinierten `BYTE`-Arrays für Testaufbauten, ohne das Array manuell in jeder Applikation neu zu verdrahten.

⚖️ Vergleich mit ähnlichen Bausteinen

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`: dieselbe generische Implementierung mit anderer Array-Größe bzw. anderem Datentyp.
- **`ARRAY2ARRAY_2_BYTE`**: kopiert ein bereits vorhandenes Array, ohne es bereitzustellen/vorzubelegen.

Fazit

`PROVIDE_ARR_0021_BYTE` liefert einen einfachen, zur Entwurfszeit vorbelegbaren `BYTE`-Array-Container fester Größe 21 und eignet sich für Konfigurations- und Testdatenarrays, die wiederverwendbar in 4diac-Applikationen eingebunden werden sollen.

PROVIDE_ARR_0028_BYTE

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `PROVIDE_ARR_0028_BYTE` stellt einen `BYTE`-Array-Container fester Größe 28 als `InOut`-Variable `D1` bereit. Er gehört zur generischen `GEN_INIT_ARR`-Familie: eine Reihe baugleicher Bausteine, die sich nur in Array-Größe und Datentyp unterscheiden, und dient dazu, ein zur Entwurfszeit mit Startwerten vorbelegbares Array in einer 4diac-Anwendung verfügbar zu machen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, trägt `D1`.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung, trägt `D1`.

Ein-/Ausgangsvariablen (InOut)

- **D1** (`BYTE`, Array-Größe 28, Index 0…27): Der Array-Container (8-Bit-Bitmuster je Element). Kann im FB-Instanzeditor mit Startwerten vorbelegt werden (z. B. `[16#01, 16#00, ...]` für `BYTE`).

Funktionsweise

Bei Eintreffen von `INIT` liest der Baustein das `InOut`-Array `D1` (das bereits zur Entwurfszeit mit Startwerten vorbelegt sein kann) und bestätigt dies unmittelbar über `INITO`, wobei `D1` unverändert weitergereicht wird. Der Baustein führt keine eigene Berechnung durch — er dient als deklarativer Container, dessen Inhalt über die FB-Instanzparametrierung vorgegeben wird.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, dieselbe C++-Basis wie alle anderen `PROVIDE_ARR_*`-Varianten; Array-Größe und Datentyp werden pro Instanziierung über die Typdefinition festgelegt.
- **InOut statt getrennter Input/Output**: `D1` ist eine einzige `InOut`-Variable — Vorbelegung und Bereitstellung erfolgen über dieselbe Variable, es gibt keine separate Kopie.
- **Feste Größe 28**: Weitere Größen derselben oder anderer Datentypen: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`.

Zustandsübersicht

Zustandslos: `INIT` liest `D1` und bestätigt sofort über `INITO`, ohne den Inhalt zu verändern.

Anwendungsszenarien

- **Konfigurationsarrays**: Bereitstellung fest vorbelegter Tabellen (z. B. Kalibrierwerte, Lookup-Tabellen, Protokoll-Header-Bytes) als wiederverwendbarer, zur Entwurfszeit parametrierbarer Baustein.
- **Testdatengenerierung**: Bereitstellung eines vordefinierten `BYTE`-Arrays für Testaufbauten, ohne das Array manuell in jeder Applikation neu zu verdrahten.

⚖️ Vergleich mit ähnlichen Bausteinen

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`: dieselbe generische Implementierung mit anderer Array-Größe bzw. anderem Datentyp.
- **`ARRAY2ARRAY_2_BYTE`**: kopiert ein bereits vorhandenes Array, ohne es bereitzustellen/vorzubelegen.

Fazit

`PROVIDE_ARR_0028_BYTE` liefert einen einfachen, zur Entwurfszeit vorbelegbaren `BYTE`-Array-Container fester Größe 28 und eignet sich für Konfigurations- und Testdatenarrays, die wiederverwendbar in 4diac-Applikationen eingebunden werden sollen.

PROVIDE_ARR_0032_BYTE

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `PROVIDE_ARR_0032_BYTE` stellt einen `BYTE`-Array-Container fester Größe 32 als `InOut`-Variable `D1` bereit. Er gehört zur generischen `GEN_INIT_ARR`-Familie: eine Reihe baugleicher Bausteine, die sich nur in Array-Größe und Datentyp unterscheiden, und dient dazu, ein zur Entwurfszeit mit Startwerten vorbelegbares Array in einer 4diac-Anwendung verfügbar zu machen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, trägt `D1`.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung, trägt `D1`.

Ein-/Ausgangsvariablen (InOut)

- **D1** (`BYTE`, Array-Größe 32, Index 0…31): Der Array-Container (8-Bit-Bitmuster je Element). Kann im FB-Instanzeditor mit Startwerten vorbelegt werden (z. B. `[16#01, 16#00, ...]` für `BYTE`).

Funktionsweise

Bei Eintreffen von `INIT` liest der Baustein das `InOut`-Array `D1` (das bereits zur Entwurfszeit mit Startwerten vorbelegt sein kann) und bestätigt dies unmittelbar über `INITO`, wobei `D1` unverändert weitergereicht wird. Der Baustein führt keine eigene Berechnung durch — er dient als deklarativer Container, dessen Inhalt über die FB-Instanzparametrierung vorgegeben wird.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, dieselbe C++-Basis wie alle anderen `PROVIDE_ARR_*`-Varianten; Array-Größe und Datentyp werden pro Instanziierung über die Typdefinition festgelegt.
- **InOut statt getrennter Input/Output**: `D1` ist eine einzige `InOut`-Variable — Vorbelegung und Bereitstellung erfolgen über dieselbe Variable, es gibt keine separate Kopie.
- **Feste Größe 32**: Weitere Größen derselben oder anderer Datentypen: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`.

Zustandsübersicht

Zustandslos: `INIT` liest `D1` und bestätigt sofort über `INITO`, ohne den Inhalt zu verändern.

Anwendungsszenarien

- **Konfigurationsarrays**: Bereitstellung fest vorbelegter Tabellen (z. B. Kalibrierwerte, Lookup-Tabellen, Protokoll-Header-Bytes) als wiederverwendbarer, zur Entwurfszeit parametrierbarer Baustein.
- **Testdatengenerierung**: Bereitstellung eines vordefinierten `BYTE`-Arrays für Testaufbauten, ohne das Array manuell in jeder Applikation neu zu verdrahten.

⚖️ Vergleich mit ähnlichen Bausteinen

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`: dieselbe generische Implementierung mit anderer Array-Größe bzw. anderem Datentyp.
- **`ARRAY2ARRAY_2_BYTE`**: kopiert ein bereits vorhandenes Array, ohne es bereitzustellen/vorzubelegen.

Fazit

`PROVIDE_ARR_0032_BYTE` liefert einen einfachen, zur Entwurfszeit vorbelegbaren `BYTE`-Array-Container fester Größe 32 und eignet sich für Konfigurations- und Testdatenarrays, die wiederverwendbar in 4diac-Applikationen eingebunden werden sollen.

PROVIDE_ARR_0035_BYTE

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `PROVIDE_ARR_0035_BYTE` stellt einen `BYTE`-Array-Container fester Größe 35 als `InOut`-Variable `D1` bereit. Er gehört zur generischen `GEN_INIT_ARR`-Familie: eine Reihe baugleicher Bausteine, die sich nur in Array-Größe und Datentyp unterscheiden, und dient dazu, ein zur Entwurfszeit mit Startwerten vorbelegbares Array in einer 4diac-Anwendung verfügbar zu machen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, trägt `D1`.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung, trägt `D1`.

Ein-/Ausgangsvariablen (InOut)

- **D1** (`BYTE`, Array-Größe 35, Index 0…34): Der Array-Container (8-Bit-Bitmuster je Element). Kann im FB-Instanzeditor mit Startwerten vorbelegt werden (z. B. `[16#01, 16#00, ...]` für `BYTE`).

Funktionsweise

Bei Eintreffen von `INIT` liest der Baustein das `InOut`-Array `D1` (das bereits zur Entwurfszeit mit Startwerten vorbelegt sein kann) und bestätigt dies unmittelbar über `INITO`, wobei `D1` unverändert weitergereicht wird. Der Baustein führt keine eigene Berechnung durch — er dient als deklarativer Container, dessen Inhalt über die FB-Instanzparametrierung vorgegeben wird.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, dieselbe C++-Basis wie alle anderen `PROVIDE_ARR_*`-Varianten; Array-Größe und Datentyp werden pro Instanziierung über die Typdefinition festgelegt.
- **InOut statt getrennter Input/Output**: `D1` ist eine einzige `InOut`-Variable — Vorbelegung und Bereitstellung erfolgen über dieselbe Variable, es gibt keine separate Kopie.
- **Feste Größe 35**: Weitere Größen derselben oder anderer Datentypen: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`.

Zustandsübersicht

Zustandslos: `INIT` liest `D1` und bestätigt sofort über `INITO`, ohne den Inhalt zu verändern.

Anwendungsszenarien

- **Konfigurationsarrays**: Bereitstellung fest vorbelegter Tabellen (z. B. Kalibrierwerte, Lookup-Tabellen, Protokoll-Header-Bytes) als wiederverwendbarer, zur Entwurfszeit parametrierbarer Baustein.
- **Testdatengenerierung**: Bereitstellung eines vordefinierten `BYTE`-Arrays für Testaufbauten, ohne das Array manuell in jeder Applikation neu zu verdrahten.

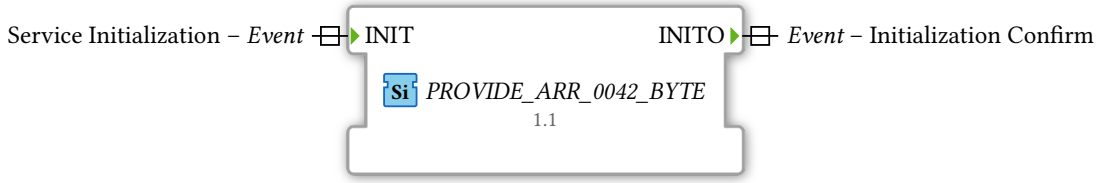
⚖️ Vergleich mit ähnlichen Bausteinen

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`: dieselbe generische Implementierung mit anderer Array-Größe bzw. anderem Datentyp.
- **`ARRAY2ARRAY_2_BYTE`**: kopiert ein bereits vorhandenes Array, ohne es bereitzustellen/vorzubelegen.

Fazit

`PROVIDE_ARR_0035_BYTE` liefert einen einfachen, zur Entwurfszeit vorbelegbaren `BYTE`-Array-Container fester Größe 35 und eignet sich für Konfigurations- und Testdatenarrays, die wiederverwendbar in 4diac-Applikationen eingebunden werden sollen.

PROVIDE_ARR_0042_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `PROVIDE_ARR_0042_BYTE` stellt einen `BYTE`-Array-Container fester Größe 42 als `InOut`-Variable `D1` bereit. Er gehört zur generischen `GEN_INIT_ARR`-Familie: eine Reihe baugleicher Bausteine, die sich nur in Array-Größe und Datentyp unterscheiden, und dient dazu, ein zur Entwurfszeit mit Startwerten vorbelegbares Array in einer 4diac-Anwendung verfügbar zu machen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, trägt `D1`.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung, trägt `D1`.

Ein-/Ausgangsvariablen (InOut)

- **D1** (`BYTE`, Array-Größe 42, Index 0…41): Der Array-Container (8-Bit-Bitmuster je Element). Kann im FB-Instanzeditor mit Startwerten vorbelegt werden (z. B. `[16#01, 16#00, ...]` für `BYTE`).

Funktionsweise

Bei Eintreffen von `INIT` liest der Baustein das `InOut`-Array `D1` (das bereits zur Entwurfszeit mit Startwerten vorbelegt sein kann) und bestätigt dies unmittelbar über `INITO`, wobei `D1` unverändert weitergereicht wird. Der Baustein führt keine eigene Berechnung durch — er dient als deklarativer Container, dessen Inhalt über die FB-Instanzparametrierung vorgegeben wird.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, dieselbe C++-Basis wie alle anderen `PROVIDE_ARR_*`-Varianten; Array-Größe und Datentyp werden pro Instanziierung über die Typdefinition festgelegt.
- **InOut statt getrennter Input/Output**: `D1` ist eine einzige `InOut`-Variable — Vorbelegung und Bereitstellung erfolgen über dieselbe Variable, es gibt keine separate Kopie.
- **Feste Größe 42**: Weitere Größen derselben oder anderer Datentypen: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`.

Zustandsübersicht

Zustandslos: `INIT` liest `D1` und bestätigt sofort über `INITO`, ohne den Inhalt zu verändern.

Anwendungsszenarien

- **Konfigurationsarrays**: Bereitstellung fest vorbelegter Tabellen (z. B. Kalibrierwerte, Lookup-Tabellen, Protokoll-Header-Bytes) als wiederverwendbarer, zur Entwurfszeit parametrierbarer Baustein.
- **Testdatengenerierung**: Bereitstellung eines vordefinierten `BYTE`-Arrays für Testaufbauten, ohne das Array manuell in jeder Applikation neu zu verdrahten.

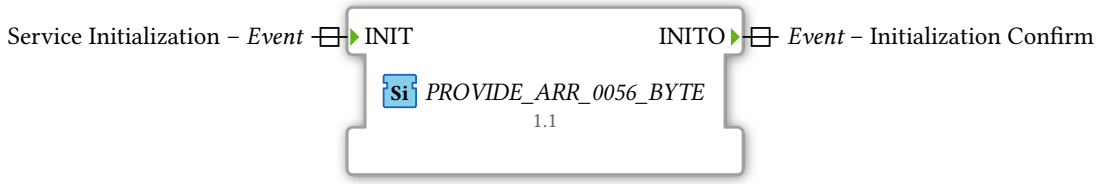
⚖️ Vergleich mit ähnlichen Bausteinen

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`: dieselbe generische Implementierung mit anderer Array-Größe bzw. anderem Datentyp.
- **`ARRAY2ARRAY_2_BYTE`**: kopiert ein bereits vorhandenes Array, ohne es bereitzustellen/vorzubelegen.

Fazit

`PROVIDE_ARR_0042_BYTE` liefert einen einfachen, zur Entwurfszeit vorbelegbaren `BYTE`-Array-Container fester Größe 42 und eignet sich für Konfigurations- und Testdatenarrays, die wiederverwendbar in 4diac-Applikationen eingebunden werden sollen.

PROVIDE_ARR_0056_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `PROVIDE_ARR_0056_BYTE` stellt einen `BYTE`-Array-Container fester Größe 56 als `InOut`-Variable `D1` bereit. Er gehört zur generischen `GEN_INIT_ARR`-Familie: eine Reihe baugleicher Bausteine, die sich nur in Array-Größe und Datentyp unterscheiden, und dient dazu, ein zur Entwurfszeit mit Startwerten vorbelegbares Array in einer 4diac-Anwendung verfügbar zu machen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, trägt `D1`.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung, trägt `D1`.

Ein-/Ausgangsvariablen (InOut)

- **D1** (`BYTE`, Array-Größe 56, Index 0…55): Der Array-Container (8-Bit-Bitmuster je Element). Kann im FB-Instanzeditor mit Startwerten vorbelegt werden (z. B. `[16#01, 16#00, ...]` für `BYTE`).

Funktionsweise

Bei Eintreffen von `INIT` liest der Baustein das `InOut`-Array `D1` (das bereits zur Entwurfszeit mit Startwerten vorbelegt sein kann) und bestätigt dies unmittelbar über `INITO`, wobei `D1` unverändert weitergereicht wird. Der Baustein führt keine eigene Berechnung durch — er dient als deklarativer Container, dessen Inhalt über die FB-Instanzparametrierung vorgegeben wird.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, dieselbe C++-Basis wie alle anderen `PROVIDE_ARR_*`-Varianten; Array-Größe und Datentyp werden pro Instanziierung über die Typdefinition festgelegt.
- **InOut statt getrennter Input/Output**: `D1` ist eine einzige `InOut`-Variable — Vorbelegung und Bereitstellung erfolgen über dieselbe Variable, es gibt keine separate Kopie.
- **Feste Größe 56**: Weitere Größen derselben oder anderer Datentypen: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`.

Zustandsübersicht

Zustandslos: `INIT` liest `D1` und bestätigt sofort über `INITO`, ohne den Inhalt zu verändern.

Anwendungsszenarien

- **Konfigurationsarrays**: Bereitstellung fest vorbelegter Tabellen (z. B. Kalibrierwerte, Lookup-Tabellen, Protokoll-Header-Bytes) als wiederverwendbarer, zur Entwurfszeit parametrierbarer Baustein.
- **Testdatengenerierung**: Bereitstellung eines vordefinierten `BYTE`-Arrays für Testaufbauten, ohne das Array manuell in jeder Applikation neu zu verdrahten.

⚖️ Vergleich mit ähnlichen Bausteinen

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`: dieselbe generische Implementierung mit anderer Array-Größe bzw. anderem Datentyp.
- **`ARRAY2ARRAY_2_BYTE`**: kopiert ein bereits vorhandenes Array, ohne es bereitzustellen/vorzubelegen.

Fazit

`PROVIDE_ARR_0056_BYTE` liefert einen einfachen, zur Entwurfszeit vorbelegbaren `BYTE`-Array-Container fester Größe 56 und eignet sich für Konfigurations- und Testdatenarrays, die wiederverwendbar in 4diac-Applikationen eingebunden werden sollen.

PROVIDE_ARR_0064_BYTE

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `PROVIDE_ARR_0064_BYTE` stellt einen `BYTE`-Array-Container fester Größe 64 als `InOut`-Variable `D1` bereit. Er gehört zur generischen `GEN_INIT_ARR`-Familie: eine Reihe baugleicher Bausteine, die sich nur in Array-Größe und Datentyp unterscheiden, und dient dazu, ein zur Entwurfszeit mit Startwerten vorbelegbares Array in einer 4diac-Anwendung verfügbar zu machen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, trägt `D1`.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung, trägt `D1`.

Ein-/Ausgangsvariablen (InOut)

- **D1** (`BYTE`, Array-Größe 64, Index 0…63): Der Array-Container (8-Bit-Bitmuster je Element). Kann im FB-Instanzeditor mit Startwerten vorbelegt werden (z. B. `[16#01, 16#00, ...]` für `BYTE`).

Funktionsweise

Bei Eintreffen von `INIT` liest der Baustein das `InOut`-Array `D1` (das bereits zur Entwurfszeit mit Startwerten vorbelegt sein kann) und bestätigt dies unmittelbar über `INITO`, wobei `D1` unverändert weitergereicht wird. Der Baustein führt keine eigene Berechnung durch — er dient als deklarativer Container, dessen Inhalt über die FB-Instanzparametrierung vorgegeben wird.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, dieselbe C++-Basis wie alle anderen `PROVIDE_ARR_*`-Varianten; Array-Größe und Datentyp werden pro Instanziierung über die Typdefinition festgelegt.
- **InOut statt getrennter Input/Output**: `D1` ist eine einzige `InOut`-Variable — Vorbelegung und Bereitstellung erfolgen über dieselbe Variable, es gibt keine separate Kopie.
- **Feste Größe 64**: Weitere Größen derselben oder anderer Datentypen: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`.

Zustandsübersicht

Zustandslos: `INIT` liest `D1` und bestätigt sofort über `INITO`, ohne den Inhalt zu verändern.

Anwendungsszenarien

- **Konfigurationsarrays**: Bereitstellung fest vorbelegter Tabellen (z. B. Kalibrierwerte, Lookup-Tabellen, Protokoll-Header-Bytes) als wiederverwendbarer, zur Entwurfszeit parametrierbarer Baustein.
- **Testdatengenerierung**: Bereitstellung eines vordefinierten `BYTE`-Arrays für Testaufbauten, ohne das Array manuell in jeder Applikation neu zu verdrahten.

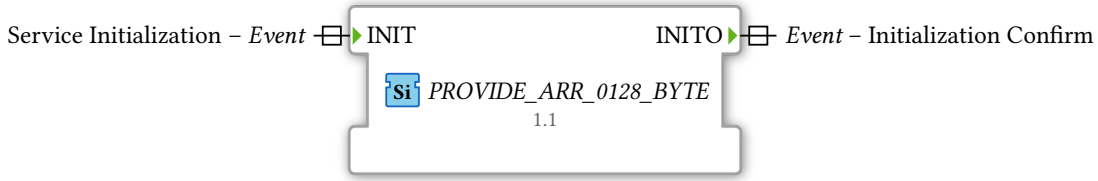
⚖️ Vergleich mit ähnlichen Bausteinen

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`: dieselbe generische Implementierung mit anderer Array-Größe bzw. anderem Datentyp.
- **`ARRAY2ARRAY_2_BYTE`**: kopiert ein bereits vorhandenes Array, ohne es bereitzustellen/vorzubelegen.

Fazit

`PROVIDE_ARR_0064_BYTE` liefert einen einfachen, zur Entwurfszeit vorbelegbaren `BYTE`-Array-Container fester Größe 64 und eignet sich für Konfigurations- und Testdatenarrays, die wiederverwendbar in 4diac-Applikationen eingebunden werden sollen.

PROVIDE_ARR_0128_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `PROVIDE_ARR_0128_BYTE` stellt einen `BYTE`-Array-Container fester Größe 128 als `InOut`-Variable `D1` bereit. Er gehört zur generischen `GEN_INIT_ARR`-Familie: eine Reihe baugleicher Bausteine, die sich nur in Array-Größe und Datentyp unterscheiden, und dient dazu, ein zur Entwurfszeit mit Startwerten vorbelegbares Array in einer 4diac-Anwendung verfügbar zu machen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, trägt `D1`.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung, trägt `D1`.

Ein-/Ausgangsvariablen (InOut)

- **D1** (`BYTE`, Array-Größe 128, Index 0…127): Der Array-Container (8-Bit-Bitmuster je Element). Kann im FB-Instanzeditor mit Startwerten vorbelegt werden (z. B. `[16#01, 16#00, ...]` für `BYTE`).

Funktionsweise

Bei Eintreffen von `INIT` liest der Baustein das `InOut`-Array `D1` (das bereits zur Entwurfszeit mit Startwerten vorbelegt sein kann) und bestätigt dies unmittelbar über `INITO`, wobei `D1` unverändert weitergereicht wird. Der Baustein führt keine eigene Berechnung durch — er dient als deklarativer Container, dessen Inhalt über die FB-Instanzparametrierung vorgegeben wird.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, dieselbe C++-Basis wie alle anderen `PROVIDE_ARR_*`-Varianten; Array-Größe und Datentyp werden pro Instanziierung über die Typdefinition festgelegt.
- **InOut statt getrennter Input/Output**: `D1` ist eine einzige `InOut`-Variable — Vorbelegung und Bereitstellung erfolgen über dieselbe Variable, es gibt keine separate Kopie.
- **Feste Größe 128**: Weitere Größen derselben oder anderer Datentypen: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`.

Zustandsübersicht

Zustandslos: `INIT` liest `D1` und bestätigt sofort über `INITO`, ohne den Inhalt zu verändern.

Anwendungsszenarien

- **Konfigurationsarrays**: Bereitstellung fest vorbelegter Tabellen (z. B. Kalibrierwerte, Lookup-Tabellen, Protokoll-Header-Bytes) als wiederverwendbarer, zur Entwurfszeit parametrierbarer Baustein.
- **Testdatengenerierung**: Bereitstellung eines vordefinierten `BYTE`-Arrays für Testaufbauten, ohne das Array manuell in jeder Applikation neu zu verdrahten.

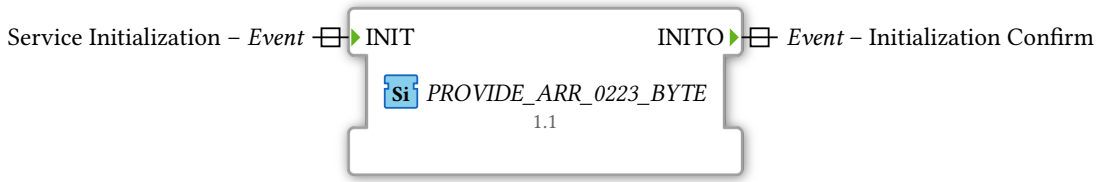
⚖️ Vergleich mit ähnlichen Bausteinen

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`: dieselbe generische Implementierung mit anderer Array-Größe bzw. anderem Datentyp.
- **`ARRAY2ARRAY_2_BYTE`**: kopiert ein bereits vorhandenes Array, ohne es bereitzustellen/vorzubelegen.

Fazit

`PROVIDE_ARR_0128_BYTE` liefert einen einfachen, zur Entwurfszeit vorbelegbaren `BYTE`-Array-Container fester Größe 128 und eignet sich für Konfigurations- und Testdatenarrays, die wiederverwendbar in 4diac-Applikationen eingebunden werden sollen.

PROVIDE_ARR_0223_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `PROVIDE_ARR_0223_BYTE` stellt einen `BYTE`-Array-Container fester Größe 223 als `InOut`-Variable `D1` bereit. Er gehört zur generischen `GEN_INIT_ARR`-Familie: eine Reihe baugleicher Bausteine, die sich nur in Array-Größe und Datentyp unterscheiden, und dient dazu, ein zur Entwurfszeit mit Startwerten vorbelegbares Array in einer 4diac-Anwendung verfügbar zu machen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, trägt `D1`.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung, trägt `D1`.

Ein-/Ausgangsvariablen (InOut)

- **D1** (`BYTE`, Array-Größe 223, Index 0…222): Der Array-Container (8-Bit-Bitmuster je Element). Kann im FB-Instanzeditor mit Startwerten vorbelegt werden (z. B. `[16#01, 16#00, ...]` für `BYTE`).

Funktionsweise

Bei Eintreffen von `INIT` liest der Baustein das `InOut`-Array `D1` (das bereits zur Entwurfszeit mit Startwerten vorbelegt sein kann) und bestätigt dies unmittelbar über `INITO`, wobei `D1` unverändert weitergereicht wird. Der Baustein führt keine eigene Berechnung durch — er dient als deklarativer Container, dessen Inhalt über die FB-Instanzparametrierung vorgegeben wird.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, dieselbe C++-Basis wie alle anderen `PROVIDE_ARR_*`-Varianten; Array-Größe und Datentyp werden pro Instanziierung über die Typdefinition festgelegt.
- **InOut statt getrennter Input/Output**: `D1` ist eine einzige `InOut`-Variable — Vorbelegung und Bereitstellung erfolgen über dieselbe Variable, es gibt keine separate Kopie.

- **Feste Größe 223**: Weitere Größen derselben oder anderer Datentypen: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`.

Zustandsübersicht

Zustandslos: `INIT` liest `D1` und bestätigt sofort über `INITO`, ohne den Inhalt zu verändern.

Anwendungsszenarien

- **Konfigurationsarrays**: Bereitstellung fest vorbelegter Tabellen (z. B. Kalibrierwerte, Lookup-Tabellen, Protokoll-Header-Bytes) als wiederverwendbarer, zur Entwurfszeit parametrierbarer Baustein.
- **Testdatengenerierung**: Bereitstellung eines vordefinierten `BYTE`-Arrays für Testaufbauten, ohne das Array manuell in jeder Applikation neu zu verdrahten.

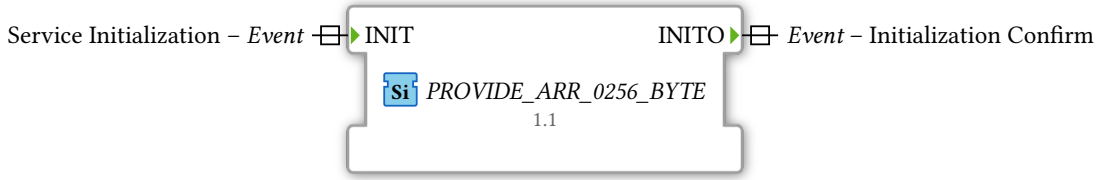
⚖️ Vergleich mit ähnlichen Bausteinen

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`: dieselbe generische Implementierung mit anderer Array-Größe bzw. anderem Datentyp.
- **`ARRAY2ARRAY_2_BYTE`**: kopiert ein bereits vorhandenes Array, ohne es bereitzustellen/vorzubelegen.

Fazit

`PROVIDE_ARR_0223_BYTE` liefert einen einfachen, zur Entwurfszeit vorbelegbaren `BYTE`-Array-Container fester Größe 223 und eignet sich für Konfigurations- und Testdatenarrays, die wiederverwendbar in 4diac-Applikationen eingebunden werden sollen.

PROVIDE_ARR_0256_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `PROVIDE_ARR_0256_BYTE` stellt einen `BYTE`-Array-Container fester Größe 256 als `InOut`-Variable `D1` bereit. Er gehört zur generischen `GEN_INIT_ARR`-Familie: eine Reihe baugleicher Bausteine, die sich nur in Array-Größe und Datentyp unterscheiden, und dient dazu, ein zur Entwurfszeit mit Startwerten vorbelegbares Array in einer 4diac-Anwendung verfügbar zu machen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, trägt `D1`.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung, trägt `D1`.

Ein-/Ausgangsvariablen (InOut)

- **D1** (`BYTE`, Array-Größe 256, Index 0…255): Der Array-Container (8-Bit-Bitmuster je Element). Kann im FB-Instanzeditor mit Startwerten vorbelegt werden (z. B. `[16#01, 16#00, ...]` für `BYTE`).

Funktionsweise

Bei Eintreffen von `INIT` liest der Baustein das `InOut`-Array `D1` (das bereits zur Entwurfszeit mit Startwerten vorbelegt sein kann) und bestätigt dies unmittelbar über `INITO`, wobei `D1` unverändert weitergereicht wird. Der Baustein führt keine eigene Berechnung durch — er dient als deklarativer Container, dessen Inhalt über die FB-Instanzparametrierung vorgegeben wird.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, dieselbe C++-Basis wie alle anderen `PROVIDE_ARR_*`-Varianten; Array-Größe und Datentyp werden pro Instanziierung über die Typdefinition festgelegt.
- **InOut statt getrennter Input/Output**: `D1` ist eine einzige `InOut`-Variable — Vorbelegung und Bereitstellung erfolgen über dieselbe Variable, es gibt keine separate Kopie.
- **Feste Größe 256**: Weitere Größen derselben oder anderer Datentypen: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`.

Zustandsübersicht

Zustandslos: `INIT` liest `D1` und bestätigt sofort über `INITO`, ohne den Inhalt zu verändern.

Anwendungsszenarien

- **Konfigurationsarrays**: Bereitstellung fest vorbelegter Tabellen (z. B. Kalibrierwerte, Lookup-Tabellen, Protokoll-Header-Bytes) als wiederverwendbarer, zur Entwurfszeit parametrierbarer Baustein.
- **Testdatengenerierung**: Bereitstellung eines vordefinierten `BYTE`-Arrays für Testaufbauten, ohne das Array manuell in jeder Applikation neu zu verdrahten.

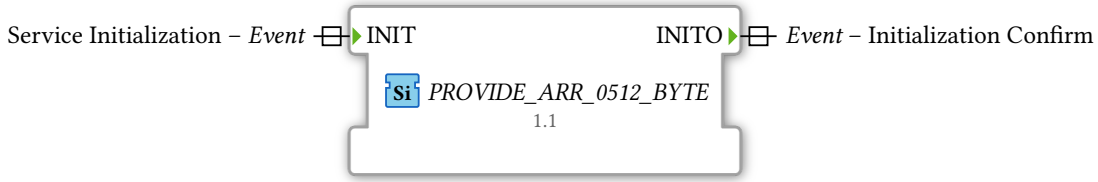
⚖️ Vergleich mit ähnlichen Bausteinen

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`: dieselbe generische Implementierung mit anderer Array-Größe bzw. anderem Datentyp.
- **`ARRAY2ARRAY_2_BYTE`**: kopiert ein bereits vorhandenes Array, ohne es bereitzustellen/vorzubelegen.

Fazit

`PROVIDE_ARR_0256_BYTE` liefert einen einfachen, zur Entwurfszeit vorbelegbaren `BYTE`-Array-Container fester Größe 256 und eignet sich für Konfigurations- und Testdatenarrays, die wiederverwendbar in 4diac-Applikationen eingebunden werden sollen.

PROVIDE_ARR_0512_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `PROVIDE_ARR_0512_BYTE` stellt einen `BYTE`-Array-Container fester Größe 512 als `InOut`-Variable `D1` bereit. Er gehört zur generischen `GEN_INIT_ARR`-Familie: eine Reihe baugleicher Bausteine, die sich nur in Array-Größe und Datentyp unterscheiden, und dient dazu, ein zur Entwurfszeit mit Startwerten vorbelegbares Array in einer 4diac-Anwendung verfügbar zu machen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, trägt `D1`.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung, trägt `D1`.

Ein-/Ausgangsvariablen (InOut)

- **D1** (`BYTE`, Array-Größe 512, Index 0…511): Der Array-Container (8-Bit-Bitmuster je Element). Kann im FB-Instanzeditor mit Startwerten vorbelegt werden (z. B. `[16#01, 16#00, ...]` für `BYTE`).

Funktionsweise

Bei Eintreffen von `INIT` liest der Baustein das `InOut`-Array `D1` (das bereits zur Entwurfszeit mit Startwerten vorbelegt sein kann) und bestätigt dies unmittelbar über `INITO`, wobei `D1` unverändert weitergereicht wird. Der Baustein führt keine eigene Berechnung durch — er dient als deklarativer Container, dessen Inhalt über die FB-Instanzparametrierung vorgegeben wird.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, dieselbe C++-Basis wie alle anderen `PROVIDE_ARR_*`-Varianten; Array-Größe und Datentyp werden pro Instanziierung über die Typdefinition festgelegt.
- **InOut statt getrennter Input/Output**: `D1` ist eine einzige `InOut`-Variable — Vorbelegung und Bereitstellung erfolgen über dieselbe Variable, es gibt keine separate Kopie.
- **Feste Größe 512**: Weitere Größen derselben oder anderer Datentypen: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`.

Zustandsübersicht

Zustandslos: `INIT` liest `D1` und bestätigt sofort über `INITO`, ohne den Inhalt zu verändern.

Anwendungsszenarien

- **Konfigurationsarrays**: Bereitstellung fest vorbelegter Tabellen (z. B. Kalibrierwerte, Lookup-Tabellen, Protokoll-Header-Bytes) als wiederverwendbarer, zur Entwurfszeit parametrierbarer Baustein.
- **Testdatengenerierung**: Bereitstellung eines vordefinierten `BYTE`-Arrays für Testaufbauten, ohne das Array manuell in jeder Applikation neu zu verdrahten.

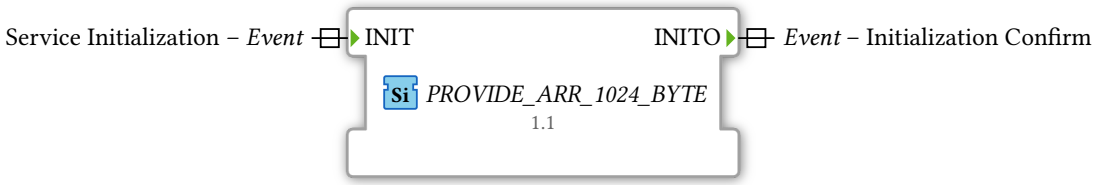
⚖️ Vergleich mit ähnlichen Bausteinen

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`: dieselbe generische Implementierung mit anderer Array-Größe bzw. anderem Datentyp.
- **`ARRAY2ARRAY_2_BYTE`**: kopiert ein bereits vorhandenes Array, ohne es bereitzustellen/vorzubelegen.

Fazit

`PROVIDE_ARR_0512_BYTE` liefert einen einfachen, zur Entwurfszeit vorbelegbaren `BYTE`-Array-Container fester Größe 512 und eignet sich für Konfigurations- und Testdatenarrays, die wiederverwendbar in 4diac-Applikationen eingebunden werden sollen.

PROVIDE_ARR_1024_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `PROVIDE_ARR_1024_BYTE` stellt einen `BYTE`-Array-Container fester Größe 1024 als `InOut`-Variable `D1` bereit. Er gehört zur generischen `GEN_INIT_ARR`-Familie: eine Reihe baugleicher Bausteine, die sich nur in Array-Größe und Datentyp unterscheiden, und dient dazu, ein zur Entwurfszeit mit Startwerten vorbelegbares Array in einer 4diac-Anwendung verfügbar zu machen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, trägt `D1`.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung, trägt `D1`.

Ein-/Ausgangsvariablen (InOut)

- **D1** (`BYTE`, Array-Größe 1024, Index 0…1023): Der Array-Container (8-Bit-Bitmuster je Element). Kann im FB-Instanzeditor mit Startwerten vorbelegt werden (z. B. `[16#01, 16#00, ...]` für `BYTE`).

Funktionsweise

Bei Eintreffen von `INIT` liest der Baustein das `InOut`-Array `D1` (das bereits zur Entwurfszeit mit Startwerten vorbelegt sein kann) und bestätigt dies unmittelbar über `INITO`, wobei `D1` unverändert weitergereicht wird. Der Baustein führt keine eigene Berechnung durch — er dient als deklarativer Container, dessen Inhalt über die FB-Instanzparametrierung vorgegeben wird.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, dieselbe C++-Basis wie alle anderen `PROVIDE_ARR_*`-Varianten; Array-Größe und Datentyp werden pro Instanziierung über die Typdefinition festgelegt.
- **InOut statt getrennter Input/Output**: `D1` ist eine einzige `InOut`-Variable — Vorbelegung und Bereitstellung erfolgen über dieselbe Variable, es gibt keine separate Kopie.
- **Feste Größe 1024**: Weitere Größen derselben oder anderer Datentypen: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`.

Zustandsübersicht

Zustandslos: `INIT` liest `D1` und bestätigt sofort über `INITO`, ohne den Inhalt zu verändern.

Anwendungsszenarien

- **Konfigurationsarrays**: Bereitstellung fest vorbelegter Tabellen (z. B. Kalibrierwerte, Lookup-Tabellen, Protokoll-Header-Bytes) als wiederverwendbarer, zur Entwurfszeit parametrierbarer Baustein.
- **Testdatengenerierung**: Bereitstellung eines vordefinierten `BYTE`-Arrays für Testaufbauten, ohne das Array manuell in jeder Applikation neu zu verdrahten.

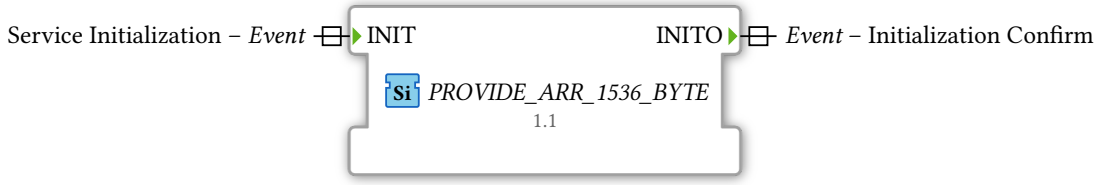
⚖️ Vergleich mit ähnlichen Bausteinen

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`: dieselbe generische Implementierung mit anderer Array-Größe bzw. anderem Datentyp.
- **`ARRAY2ARRAY_2_BYTE`**: kopiert ein bereits vorhandenes Array, ohne es bereitzustellen/vorzubelegen.

Fazit

`PROVIDE_ARR_1024_BYTE` liefert einen einfachen, zur Entwurfszeit vorbelegbaren `BYTE`-Array-Container fester Größe 1024 und eignet sich für Konfigurations- und Testdatenarrays, die wiederverwendbar in 4diac-Applikationen eingebunden werden sollen.

PROVIDE_ARR_1536_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `PROVIDE_ARR_1536_BYTE` stellt einen `BYTE`-Array-Container fester Größe 1536 als `InOut`-Variable `D1` bereit. Er gehört zur generischen `GEN_INIT_ARR`-Familie: eine Reihe baugleicher Bausteine, die sich nur in Array-Größe und Datentyp unterscheiden, und dient dazu, ein zur Entwurfszeit mit Startwerten vorbelegbares Array in einer 4diac-Anwendung verfügbar zu machen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, trägt `D1`.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung, trägt `D1`.

Ein-/Ausgangsvariablen (InOut)

- **D1** (`BYTE`, Array-Größe 1536, Index 0…1535): Der Array-Container (8-Bit-Bitmuster je Element). Kann im FB-Instanzeditor mit Startwerten vorbelegt werden (z. B. `[16#01, 16#00, ...]` für `BYTE`).

Funktionsweise

Bei Eintreffen von `INIT` liest der Baustein das `InOut`-Array `D1` (das bereits zur Entwurfszeit mit Startwerten vorbelegt sein kann) und bestätigt dies unmittelbar über `INITO`, wobei `D1` unverändert weitergereicht wird. Der Baustein führt keine eigene Berechnung durch — er dient als deklarativer Container, dessen Inhalt über die FB-Instanzparametrierung vorgegeben wird.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, dieselbe C++-Basis wie alle anderen `PROVIDE_ARR_*`-Varianten; Array-Größe und Datentyp werden pro Instanziierung über die Typdefinition festgelegt.
- **InOut statt getrennter Input/Output**: `D1` ist eine einzige `InOut`-Variable — Vorbelegung und Bereitstellung erfolgen über dieselbe Variable, es gibt keine separate Kopie.
- **Feste Größe 1536**: Weitere Größen derselben oder anderer Datentypen: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`.

Zustandsübersicht

Zustandslos: `INIT` liest `D1` und bestätigt sofort über `INITO`, ohne den Inhalt zu verändern.

Anwendungsszenarien

- **Konfigurationsarrays**: Bereitstellung fest vorbelegter Tabellen (z. B. Kalibrierwerte, Lookup-Tabellen, Protokoll-Header-Bytes) als wiederverwendbarer, zur Entwurfszeit parametrierbarer Baustein.
- **Testdatengenerierung**: Bereitstellung eines vordefinierten `BYTE`-Arrays für Testaufbauten, ohne das Array manuell in jeder Applikation neu zu verdrahten.

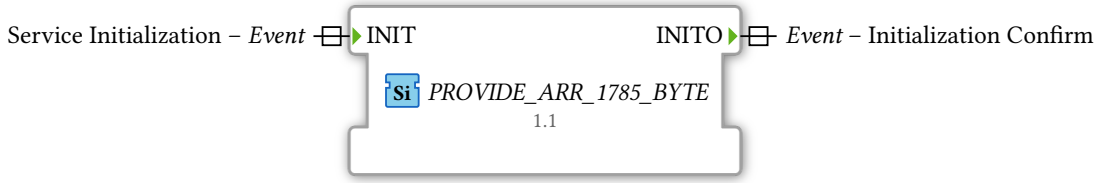
⚖️ Vergleich mit ähnlichen Bausteinen

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`: dieselbe generische Implementierung mit anderer Array-Größe bzw. anderem Datentyp.
- **`ARRAY2ARRAY_2_BYTE`**: kopiert ein bereits vorhandenes Array, ohne es bereitzustellen/vorzubelegen.

Fazit

`PROVIDE_ARR_1536_BYTE` liefert einen einfachen, zur Entwurfszeit vorbelegbaren `BYTE`-Array-Container fester Größe 1536 und eignet sich für Konfigurations- und Testdatenarrays, die wiederverwendbar in 4diac-Applikationen eingebunden werden sollen.

PROVIDE_ARR_1785_BYTE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `PROVIDE_ARR_1785_BYTE` stellt einen `BYTE`-Array-Container fester Größe 1785 als `InOut`-Variable `D1` bereit. Er gehört zur generischen `GEN_INIT_ARR`-Familie: eine Reihe baugleicher Bausteine, die sich nur in Array-Größe und Datentyp unterscheiden, und dient dazu, ein zur Entwurfszeit mit Startwerten vorbelegbares Array in einer 4diac-Anwendung verfügbar zu machen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, trägt `D1`.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung, trägt `D1`.

Ein-/Ausgangsvariablen (InOut)

- **D1** (`BYTE`, Array-Größe 1785, Index 0…1784): Der Array-Container (8-Bit-Bitmuster je Element). Kann im FB-Instanzeditor mit Startwerten vorbelegt werden (z. B. `[16#01, 16#00, ...]` für `BYTE`).

Funktionsweise

Bei Eintreffen von `INIT` liest der Baustein das `InOut`-Array `D1` (das bereits zur Entwurfszeit mit Startwerten vorbelegt sein kann) und bestätigt dies unmittelbar über `INITO`, wobei `D1` unverändert weitergereicht wird. Der Baustein führt keine eigene Berechnung durch — er dient als deklarativer Container, dessen Inhalt über die FB-Instanzparametrierung vorgegeben wird.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, dieselbe C++-Basis wie alle anderen `PROVIDE_ARR_*`-Varianten; Array-Größe und Datentyp werden pro Instanziierung über die Typdefinition festgelegt.
- **InOut statt getrennter Input/Output**: `D1` ist eine einzige `InOut`-Variable — Vorbelegung und Bereitstellung erfolgen über dieselbe Variable, es gibt keine separate Kopie.
- **Feste Größe 1785**: Weitere Größen derselben oder anderer Datentypen: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`.

Zustandsübersicht

Zustandslos: `INIT` liest `D1` und bestätigt sofort über `INITO`, ohne den Inhalt zu verändern.

Anwendungsszenarien

- **Konfigurationsarrays**: Bereitstellung fest vorbelegter Tabellen (z. B. Kalibrierwerte, Lookup-Tabellen, Protokoll-Header-Bytes) als wiederverwendbarer, zur Entwurfszeit parametrierbarer Baustein.
- **Testdatengenerierung**: Bereitstellung eines vordefinierten `BYTE`-Arrays für Testaufbauten, ohne das Array manuell in jeder Applikation neu zu verdrahten.

⚖️ Vergleich mit ähnlichen Bausteinen

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`: dieselbe generische Implementierung mit anderer Array-Größe bzw. anderem Datentyp.
- **`ARRAY2ARRAY_2_BYTE`**: kopiert ein bereits vorhandenes Array, ohne es bereitzustellen/vorzubelegen.

Fazit

`PROVIDE_ARR_1785_BYTE` liefert einen einfachen, zur Entwurfszeit vorbelegbaren `BYTE`-Array-Container fester Größe 1785 und eignet sich für Konfigurations- und Testdatenarrays, die wiederverwendbar in 4diac-Applikationen eingebunden werden sollen.

2.2.2 core

2.2.2.1 🎧 Podcast

- ESP32-S3 im Detail: Dual-Core, 32MB Power und CAN-Bus für Land- und Baumaschinen-Mechatronik
- ESP32-S3 im Detail: Dual-Core, KI-Turbo und das Schweizer Taschenmesser des IoT

🌐 **Passende Themen-Unterseiten auf ms-muc-docs.de**

- 🌐 ESP32 & ESP32-S3 DevKit auf ms-muc-docs.de

2.2.2.2 GenericClassName

• – ◦ • – ◦ • – ◦ •

Einleitung

Die Attributdeklaration `GenericClassName` definiert den Klassennamen für generische Typen in 4diac.

Technische Details

- **Basis-Datentyp:** `STRING`
- **Initialwert:** `'GEN_'`
- **Verwendung:** Wird auf `FBType`-Ebene angewendet, um generische Implementierungen zu kennzeichnen.

Metadaten

Attribut	Wert
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	eclipse4diac::core

🌐 **Passende Themen-Unterseiten auf ms-muc-docs.de**

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.2.2.3 TypeHash

• – ◦ • – ◦ • – ◦ •

Einleitung

Die Attributdeklaration `TypeHash` definiert einen festen Typ-Hash für Funktionsbausteine.

Technische Details

- **Basis-Datentyp:** `STRING`
- **Initialwert:** `''` (leer)
- **Verwendung:** Dient zur Identifizierung und Validierung von Typ-Versionen.

Metadaten

Attribut	Wert
Lizenz	EPL-2.0
Version	1.0 (2025-06-05, Martin Erich Jobst)
4diac-Paket	eclipse4diac::core

🌐 **Passende Themen-Unterseiten auf ms-muc-docs.de**

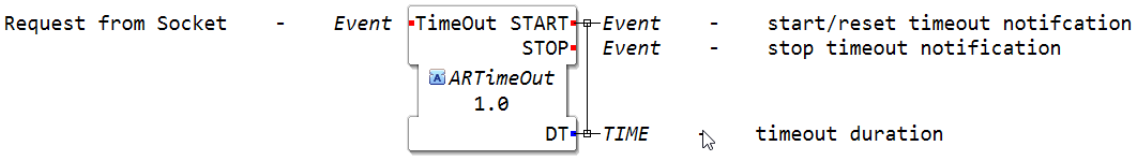
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.2.3 events

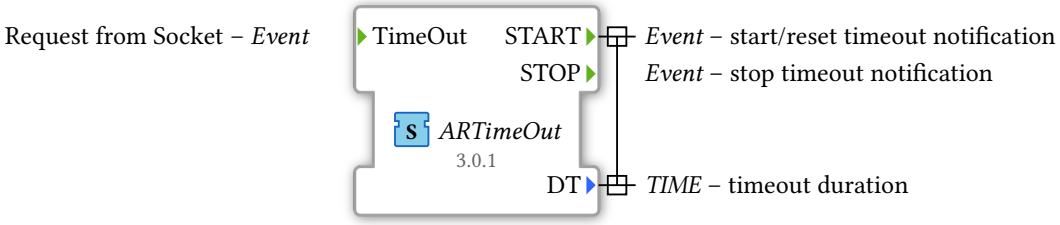
2.2.3.1 🎧 Podcast

- Taster & Events: GPIO-Grundlagen für Land- und Baumaschinen Mechatroniker

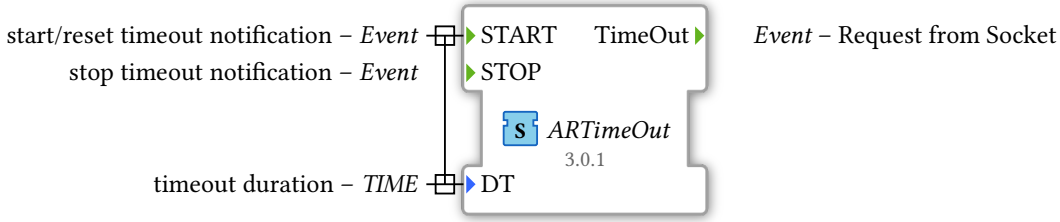
2.2.3.2 ARTimeOut



Plug (Adapter-Ausgang)



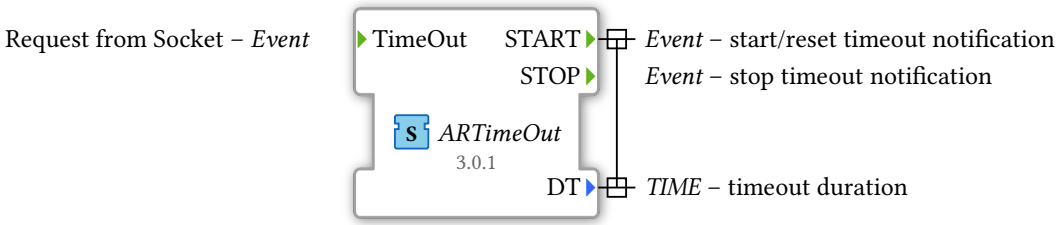
Socket (Adapter-Eingang)



• – ◦ • – ◦ • – ◦ •

Einleitung

Die IEC 61499 ist ein internationaler Standard für die Modellierung verteilter industrieller Steuerungssysteme. Innerhalb dieses Standards sind Adaptertypen eine wichtige Komponente, um wiederverwendbare Schnittstellen zwischen Funktionsbausteinen zu definieren. Der **ARTimeOut**-Adapter (Adapter for Resettable Timeout) ist ein Beispiel für einen solchen Adapter, der eine Schnittstelle für einen rücksetzbaren Timeout-Service bereitstellt.



Struktur des ARTimeOut-Adapters

Der ARTimeOut-Adapter ist als **AdapterType** in der IEC 61499 spezifiziert und definiert eine Schnittstelle für die Kommunikation zwischen einem SOCKET und einem PLUG.

Schnittstelle (Interface)

Die Schnittstelle des ARTimeOut-Adapters besteht aus:

- **Eingangsereignisse (Event Inputs)**:
  - **TimeOut**: Dieses Ereignis wird vom SOCKET ausgelöst, um anzuzeigen, dass der Timeout abgelaufen ist.
- **Ausgangsereignisse (Event Outputs)**:
  - **START**: Dieses Ereignis wird vom PLUG ausgelöst, um den Timeout zu starten oder zurückzusetzen. Es ist mit der Ausgangsvariablen **DT** verknüpft, die die Timeout-Dauer definiert.

  - **STOP**: Dieses Ereignis wird vom PLUG ausgelöst, um den Timeout anzuhalten.
- **Ausgangsvariable (Output Variable)**:
  - **DT (Duration Time)**: Eine Zeitvariable (TIME), die die Dauer des Timeouts angibt.

Service-Verhalten

Der ARTimeOut-Adapter bietet zwei Service-Sequenzen:

1. **Timeout**:
   - Der PLUG sendet ein **START**-Ereignis mit der Zeitdauer **DT**, um den Timeout zu starten.
   - Wenn der Timeout abläuft, sendet der SOCKET ein **TimeOut**-Ereignis zurück an den PLUG.
2. **NormalOperation**:
   - Der PLUG sendet ein **START**-Ereignis mit der Zeitdauer **DT**, um den Timeout zu starten.
   - Der PLUG kann ein **STOP**-Ereignis senden, um den Timeout vorzeitig zu beenden.

Verhalten des ARTimeOut-Adapters

Der ARTimeOut-Adapter ermöglicht die Steuerung eines Timeout-Mechanismus mit folgenden Funktionen:

1. **Start/Reset des Timeouts**:
   - Durch das **START**-Ereignis wird der Timeout mit der angegebenen Dauer **DT** gestartet oder zurückgesetzt.
2. **Stop des Timeouts**:
   - Das **STOP**-Ereignis beendet den Timeout vorzeitig.
3. **Timeout-Ablauf**:
   - Wenn der Timeout abläuft, wird das **TimeOut**-Ereignis ausgelöst, um den PLUG zu benachrichtigen.

Anwendungsbeispiele

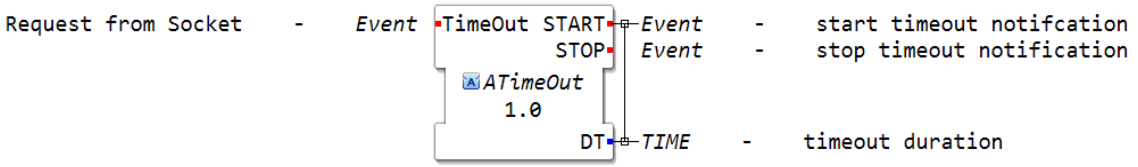
Der ARTimeOut-Adapter kann in verschiedenen industriellen Steuerungsanwendungen eingesetzt werden, insbesondere in Szenarien, in denen ein zuverlässiger und rücksetzbarer Timeout-Mechanismus erforderlich ist. Beispiele sind:

- **Netzwerkkommunikation**: Überwachung von Antwortzeitüberschreitungen in verteilten Systemen.
- **Maschinensteuerung**: Timeout für die Ausführung von Befehlen, um hängende Prozesse zu erkennen.
- **Sicherheitssysteme**: Überwachung von Sensoren, um Fehlfunktionen zu erkennen, wenn keine Aktualisierungen erfolgen.

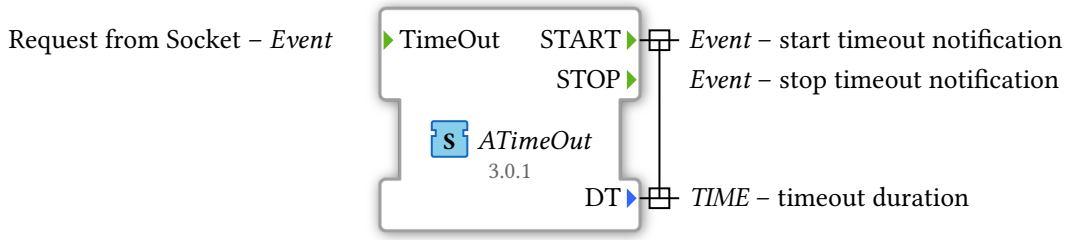
Fazit

Der ARTimeOut-Adapter ist ein nützliches Werkzeug in der IEC 61499, um eine standardisierte Schnittstelle für rücksetzbare Timeout-Services bereitzustellen. Durch seine klare Trennung zwischen SOCKET und PLUG sowie die definierten Service-Sequenzen eignet er sich ideal für die Integration in verteilte Steuerungssysteme. Die Fähigkeit, den Timeout zu starten, zu stoppen und bei Ablauf zu benachrichtigen, macht ihn zu einer flexiblen Lösung für eine Vielzahl von industriellen Anwendungen.

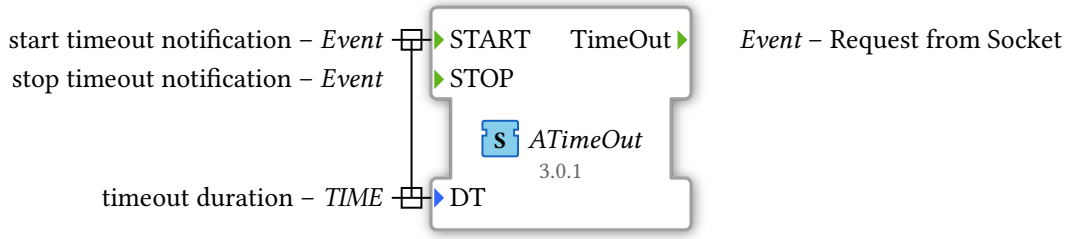
2.2.3.3 ATimeOut



Plug (Adapter-Ausgang)



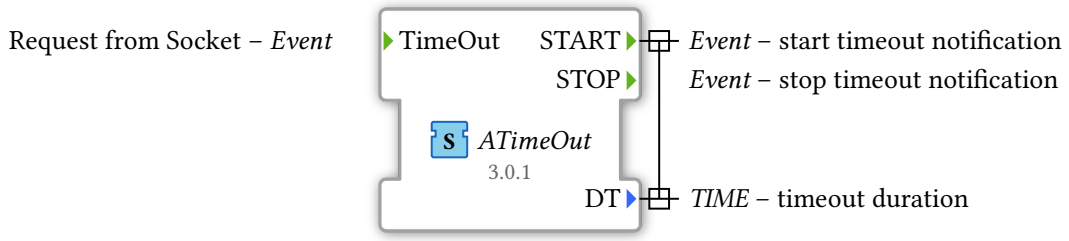
Socket (Adapter-Eingang)



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **ATimeOut**-Adapter ist eine standardisierte Schnittstelle (AdapterType) gemäß IEC 61499 zur Implementierung von Timeout-Services. Er definiert die Kommunikation zwischen einem Dienstnutzer (PLUG) und einem Zeitdienst-Anbieter (SOCKET). Im Gegensatz zum `ARTimeOut` ist dieser Adapter für einfache, nicht-nachtriggerbare Timeouts vorgesehen.



Struktur des ATimeOut-Adapters

Der ATimeOut-Adapter definiert eine klare Trennung der Zuständigkeiten zwischen der Steuerlogik (Plug) und dem Zeitgeber (Socket).

Schnittstelle (Interface)

Die Schnittstelle ist aus der Perspektive des **Plugs** definiert:

- **Eingangsereignisse (Event Inputs - vom Socket empfangen)**:
  - **TimeOut**: Signalisiert, dass die vorgegebene Zeit abgelaufen ist.
- **Ausgangsereignisse (Event Outputs - an den Socket gesendet)**:
  - **START**: Initiiert den Timeout-Vorgang. Dieses Ereignis ist mit der Variablen **DT** verknüpft.
  - **STOP**: Bricht einen laufenden Timeout-Vorgang ab.
- **Ausgangsvariablen (Output Variables - an den Socket gesendet)**:
  - **DT (Duration Time)**: Definiert die Zeitdauer für den Timeout (Datentyp: TIME).

Verhalten und Service-Sequenzen

Der Adapter unterstützt zwei grundlegende Abläufe:

1. **Timeout-Ablauf**:
   - Der Plug sendet `START` mit einem Wert für `DT`.
   - Der Socket verarbeitet die Zeit und sendet nach Ablauf `TimeOut` zurück.
2. **Vorzeitiger Abbruch**:

- Der Plug sendet `START`.
- Bevor die Zeit abläuft, sendet der Plug `STOP`. Der Socket bricht die Zeitmessung ab; es erfolgt kein `TimeOut`-Ereignis.

Technische Besonderheiten

- **Nicht nachtriggerbar**: Ein erneutes `START` während eines laufenden Timeouts wird bei der Standard-Implementierung (`E_TimeOut`) ignoriert.
- **Adapter-Konzept**: Ermöglicht eine saubere Kapselung der Zeitlogik und vereinfacht das Baustein-Netzwerk durch Reduzierung der Verbindungslinien.

Anwendungsbeispiele

- **Überwachung von Antwortzeiten**: Warten auf eine Bestätigung (z.B. von einem Kommunikationspartner) innerhalb eines festen Zeitfensters.
- **Prozessüberwachung**: Sicherstellen, dass ein mechanischer Vorgang innerhalb der erwarteten Zeit abgeschlossen wird.

⚖️ Vergleich mit ARTimeOut

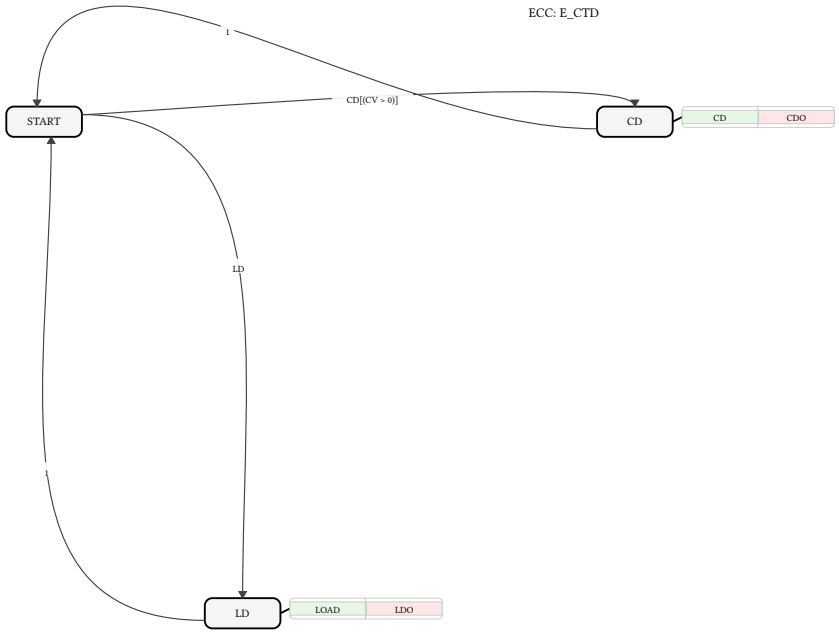
Feature	ATimeOut (dieser)	ARTimeOut
Typ	Adapter	Adapter
Nachtriggerbar	Nein	Ja
Ereignis START	Startet Timer	Startet/Resetet Timer

Fazit

Der ATimeOut-Adapter stellt eine robuste und einfache Schnittstelle für zeitkritische Überwachungsaufgaben in verteilten Steuerungssystemen dar. Er ist die Basis für den Funktionsbaustein `E_TimeOut`.

2.2.3.4 E_CTD

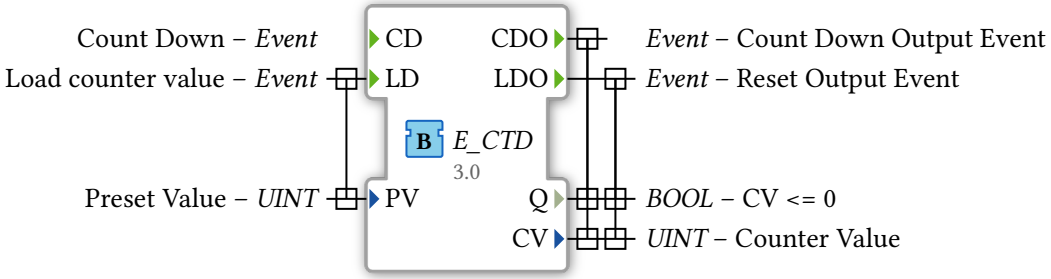
🎧 Podcast



- E_CTD: Ereignisgesteuerter Abwärtszähler nach IEC 61499

Einleitung

Der **E_CTD** (Event-Driven Down Counter) ist ein ereignisgesteuerter Abwärtszähler nach dem IEC 61499-Standard. Dieser Funktionsbaustein wird in industriellen Steuerungssystemen eingesetzt, um Zählvorgänge zu realisieren, die durch Ereignisse gesteuert werden.



Struktur des E_CTD-Bausteins

Schnittstelle (Interface)

Eingangsereignisse (Event Inputs):

- **CD (Count Down):** Löst einen Zählschritt aus, der den Zählerstand dekrementiert.
- **LD (Load):** Lädt den Startwert `PV` in den Zähler.

Ausgangsereignisse (Event Outputs):

- **CDO (Count Down Output):** Bestätigt einen Zählschritt. Wird nach jedem `CD`-Ereignis ausgelöst, solange der Zählerstand größer als 0 war.
  - **Verbundene Daten**: `Q`, `CV`
- **LDO (Load Output):** Bestätigt das erfolgreiche Laden eines neuen Zählerwerts.
  - **Verbundene Daten**: `Q`, `CV`

Eingangsvariablen (Input Variables):

- **PV (Preset Value):** Der Startwert, der bei einem LD-Ereignis geladen wird (Datentyp: `UINT`).

Ausgangsvariablen (Output Variables):

- **Q (Status):** Ausgangs-Flag, das `TRUE` wird, wenn der Zählerstand `CV` den Wert `0` erreicht (Datentyp: `BOOL`).
- **CV (Counter Value):** Der aktuelle Zählerstand (Datentyp: `UINT`).

Verhalten des E_CTD-Bausteins

1. **Initialisierung/Laden:**
   - Wenn ein **LD**-Ereignis eintritt, wird der Zählerstand `CV` auf den Wert von **PV** gesetzt.
   - Das Ausgangs-Flag `Q` wird basierend auf der Bedingung `CV = 0` aktualisiert.
   - Das **LDO**-Ereignis wird ausgelöst und gibt den neuen Zählerstand `CV` und das Flag `Q` aus.
2. **Abwärtszählen:**
   - Bei jedem **CD**-Ereignis wird der Zählerstand **CV**, sofern er größer als 0 ist, um 1 verringert.
   - Danach wird das Ausgangs-Flag `Q` basierend auf der neuen Bedingung `CV = 0` aktualisiert.
   - Das **CDO**-Ereignis wird ausgelöst und gibt den aktuellen Zählerstand `CV` und das Flag `Q` aus.
3. **Neuladen des Zählers:**
   - Ein erneutes **LD**-Ereignis setzt **CV** jederzeit zurück auf **PV** und löst **LDO** aus.

Technische Besonderheiten

- **Ereignisgesteuert:** Der Baustein arbeitet ausschließlich auf Basis von Ereignissen und benötigt keinen zyklischen Aufruf.
- **Flexible Initialisierung:** Der Startwert **PV** kann jederzeit durch ein **LD**-Ereignis geändert werden.

Anwendungsbeispiele

- **Produktionslinien:** Zählen von produzierten Einheiten.
- **Verpackungsmaschinen:** Steuerung von Füllvorgängen.

- **Energiemanagement:** Überwachung von Verbrauchszyklen.

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	E_CTD	E_CTU (Up Counter)	E_CTUD (Up/Down Counter)
Zählrichtung	Abwärts	Aufwärts	Beides
Ereignisgesteuert	Ja	Ja	Ja
Reset-Funktion	LD (Neuladen)	R (Reset)	R (Reset)

🛠️ Zugehörige Übungen

- Uebung_081

Fazit

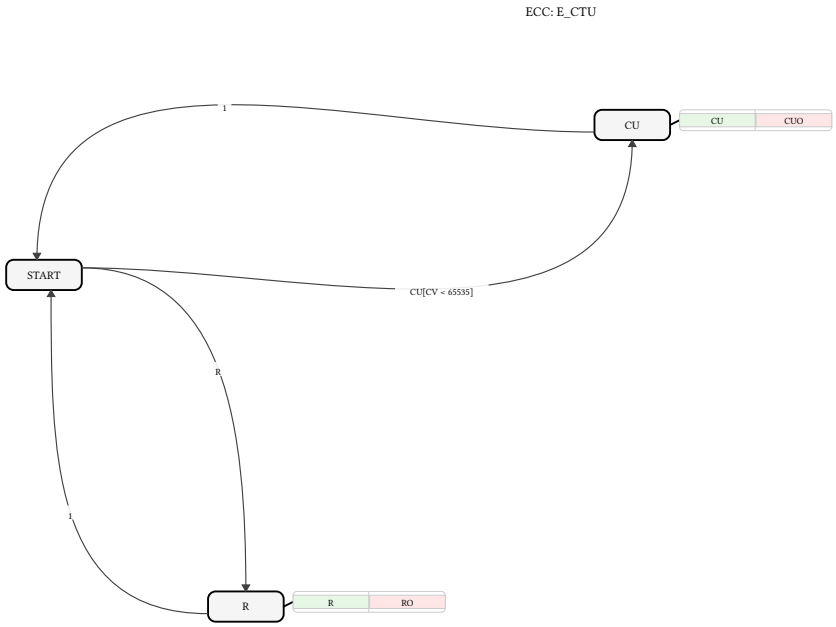
Der **E_CTD**-Baustein ist ein wesentliches Element in der IEC 61499, das eine zuverlässige und flexible Zählfunktion für industrielle Steuerungen bietet. Seine ereignisgesteuerte Natur macht ihn besonders geeignet für verteilte Systeme, wo zyklische Abfragen nicht praktikabel sind. Durch die klare Schnittstelle und das intuitive Verhalten ist er einfach in bestehende Steuerungskonzepte zu integrieren.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de

2.2.3.5 E_CTU

📺 Video

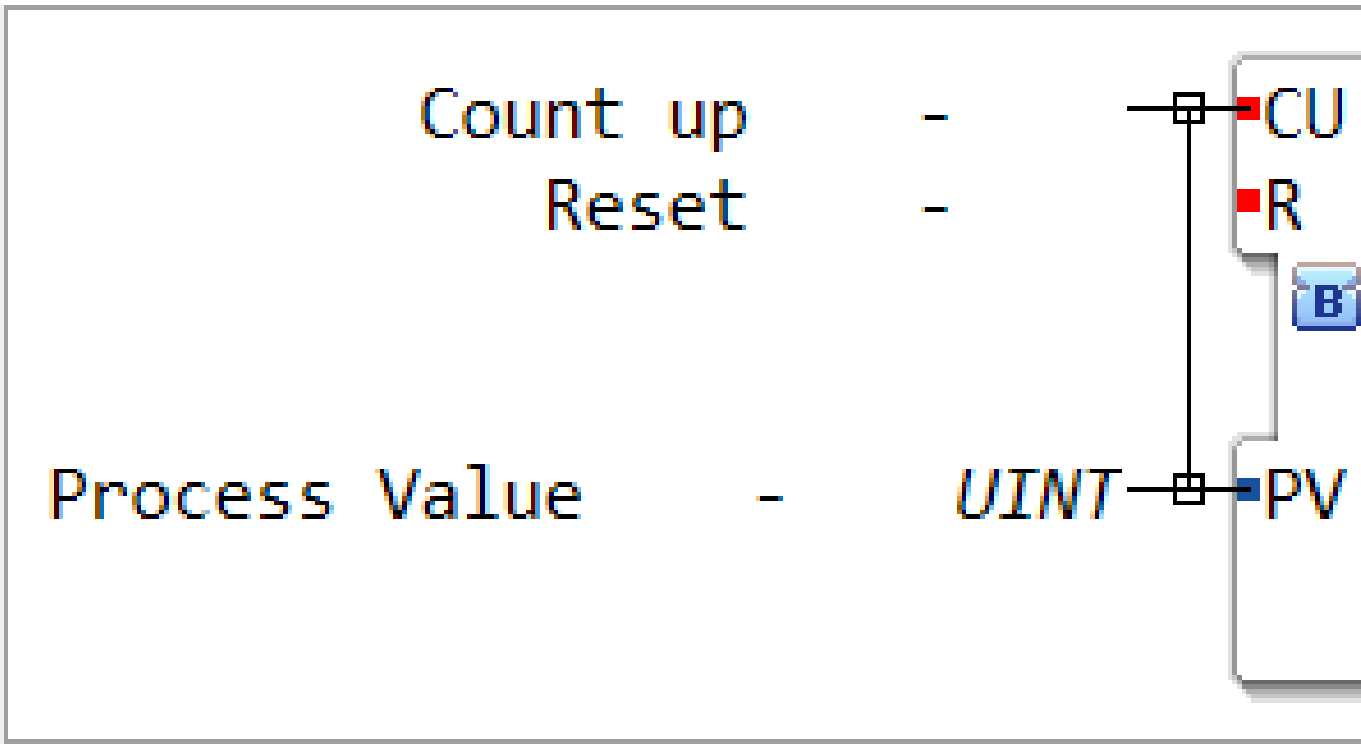


- The E_CTU upcounter

🎧 Podcast

- Der E_CTU in der IEC 61499: Ereignisgesteuertes Zählen und warum der Minimalist im Maschinenbau überzeugt
- Der E_CTU-Baustein: Ereignisgesteuertes Hochzählen in der Industrie nach IEC 61499

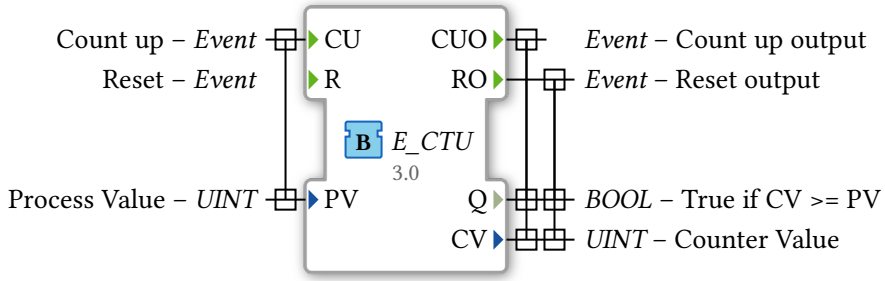
- E_CTUD: Bidirektionaler Zähler in IEC 61499 Systemen
- Meisterwissen 61499: Der Ereignisgesteuerte Aufwärtszähler (E_CTU) – Robustes Zählen in Landmaschinen-Steuerungen



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `E_CTU` (Event-Driven Up Counter) ist ein ereignisgesteuerter Aufwärtszähler gemäß dem IEC 61499-Standard. Seine Funktion ist es, bei jedem ankommenden Zählereignis einen internen Zählerstand zu erhöhen und diesen mit einem vorgegebenen Grenzwert zu vergleichen. Der Baustein kann jederzeit zurückgesetzt werden.



Schnittstellenstruktur

Ereignis-Eingänge

- **CU (Count Up)**: Löst einen Zählschritt aus, der den Zählerstand `CV` um 1 erhöht.
  - **Verbundene Daten**: `PV`
- **R (Reset)**: Setzt den Zählerstand `CV` auf 0 zurück.

Ereignis-Ausgänge

- **CUO (Count Up Output)**: Bestätigt einen Zählschritt. Wird nach jedem `CU`-Ereignis ausgelöst.
  - **Verbundene Daten**: `Q`, `CV`
- **RO (Reset Output)**: Bestätigt das Zurücksetzen des Zählers.
  - **Verbundene Daten**: `Q`, `CV`

Daten-Eingänge

- **PV (Preset Value)**: Der Grenzwert (Datentyp: `UINT`). Dieser Wert wird bei jedem `CU`-Ereignis mit dem Zählerstand verglichen.

Daten-Ausgänge

- **Q (Status)**: Ausgangs-Flag, das `TRUE` wird, wenn der Zählerstand `CV` den Grenzwert `PV` erreicht oder überschreitet (Datentyp: `BOOL`).
- **CV (Counter Value)**: Der aktuelle Zählerstand (Datentyp: `UINT`).

Funktionsweise

Der `E_CTU`-Baustein hat zwei Hauptfunktionen: Zählen und Zurücksetzen.

1. **Zählen (CU)**: Wenn ein `CU`-Ereignis eintritt und der interne Zählerstand `CV` den Maximalwert für `UINT` (65535) noch nicht erreicht hat, wird `CV` um 1 erhöht. Anschließend wird `CV` mit dem am `PV`-Eingang anliegenden Grenzwert verglichen. Wenn `CV >= PV` ist, wird der Ausgang `Q` auf `TRUE` gesetzt, andernfalls auf `FALSE`. Nach dem Zählvorgang wird das `CUO`-Ereignis ausgelöst, das den aktuellen Zählerstand `CV` und das Status-Flag `Q` ausgibt.
2. **Zurücksetzen (R)**: Wenn ein `R`-Ereignis eintritt, wird der Zählerstand `CV` sofort auf `0` und das Status-Flag `Q` auf `FALSE` gesetzt. Anschließend wird das `RO`-Ereignis ausgelöst, das die zurückgesetzten Werte `CV` und `Q` ausgibt.

Technische Besonderheiten

- **Ereignisgesteuert**: Der Baustein arbeitet ausschließlich auf Basis von Ereignissen (`CU`, `R`).
- **Überlaufschutz**: Der Zähler stoppt, wenn der maximale Wert für `UINT` (65535) erreicht ist, um einen Überlauf zu verhindern.
- **PV bei jedem Zählschritt**: Der Grenzwert `PV` wird mit dem `CU`-Ereignis verknüpft, was bedeutet, dass er potenziell bei jedem Zählschritt geändert werden kann.

Anwendungsbeispiele

- **Stückzähler**: Zählen von produzierten Teilen auf einem Förderband. Wenn eine Zielmenge (`PV`) erreicht ist, wird `Q` `TRUE`.
- **Ereigniszählung**: Erfassen der Häufigkeit von Ereignissen, wie z.B. das Betätigen eines Schalters.
- **Taktzähler**: Zählen von Taktzyklen in einer Maschine, um Wartungsintervalle zu signalisieren.

⚖️ Vergleich mit ähnlichen Bausteinen

Merkmal	E_CTU (Up Counter)	E_CTD (Down Counter)	E_CTUD (Up/Down Counter)
Zählrichtung	Aufwärts	Abwärts	Beides
Ereignisgesteuert	Ja	Ja	Ja
Reset-Funktion	R (Reset auf 0)	LD (Setzen auf PV)	R (Reset auf 0)

🛠️ Zugehörige Übungen

- Uebung_040
- Uebung_040_2
- Uebung_040_AX
- Uebung_041
- Uebung_080
- Uebung_080b
- Uebung_080c
- Uebung_084
- Uebung_12x_sub

Fazit

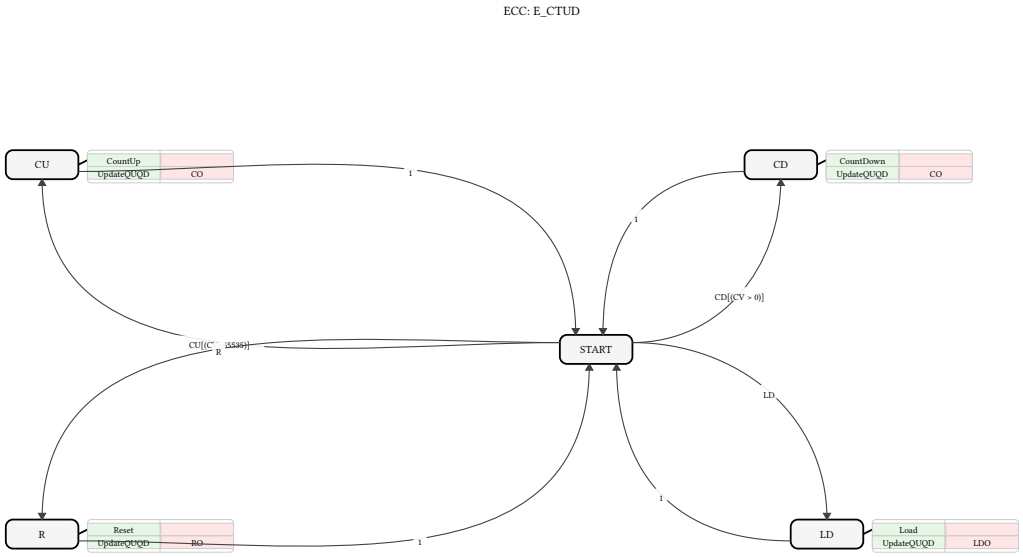
Der `E_CTU` ist ein grundlegender und vielseitiger Zählerbaustein für ereignisgesteuerte Systeme nach IEC 61499. Seine einfache Schnittstelle und sein vorhersehbares Verhalten machen ihn zu einem robusten Werkzeug für eine Vielzahl von Zähl- und Überwachungsaufgaben in der industriellen Automatisierung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

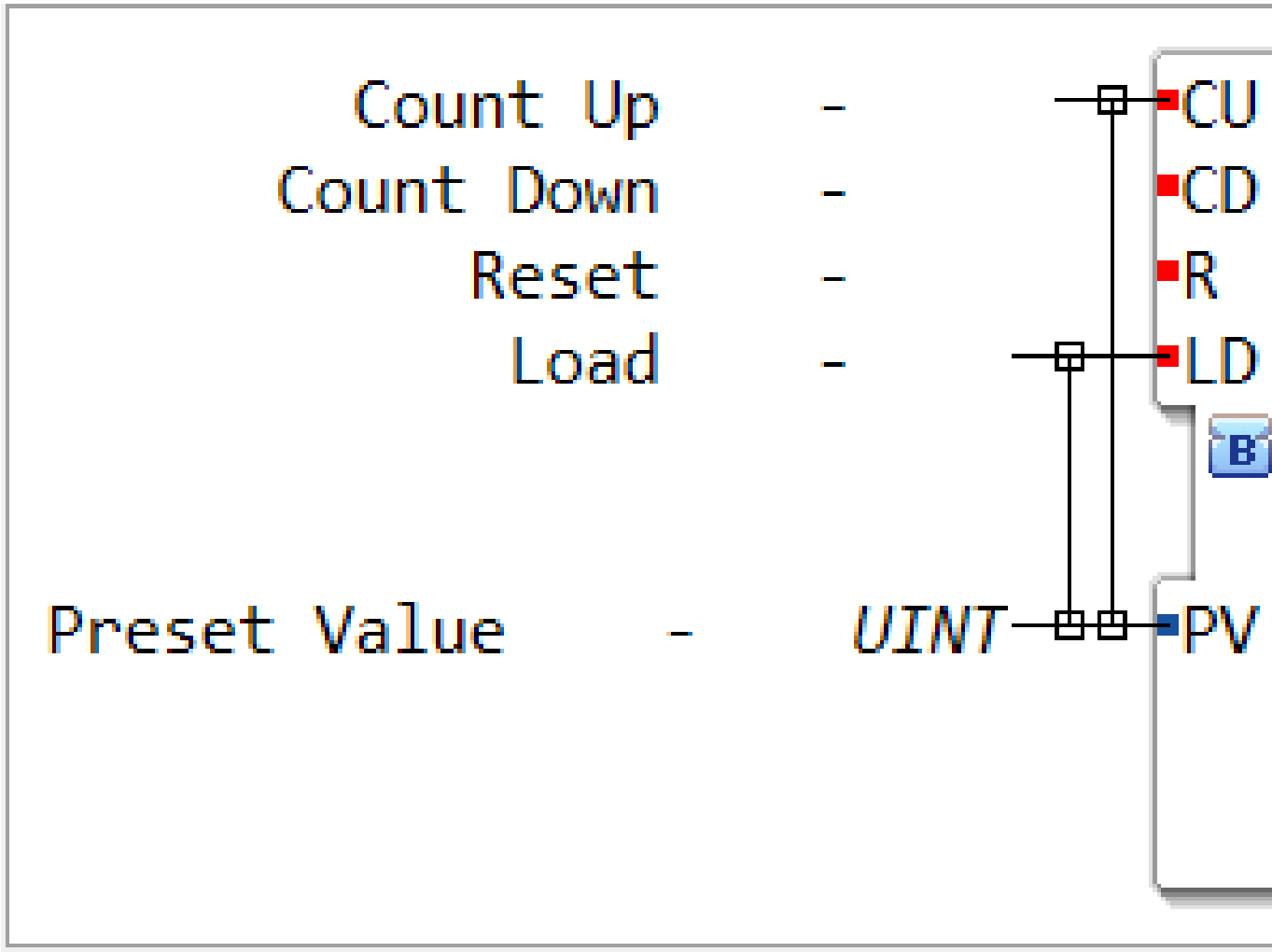
- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de

2.2.3.6 E_CTUD

🎧 Podcast



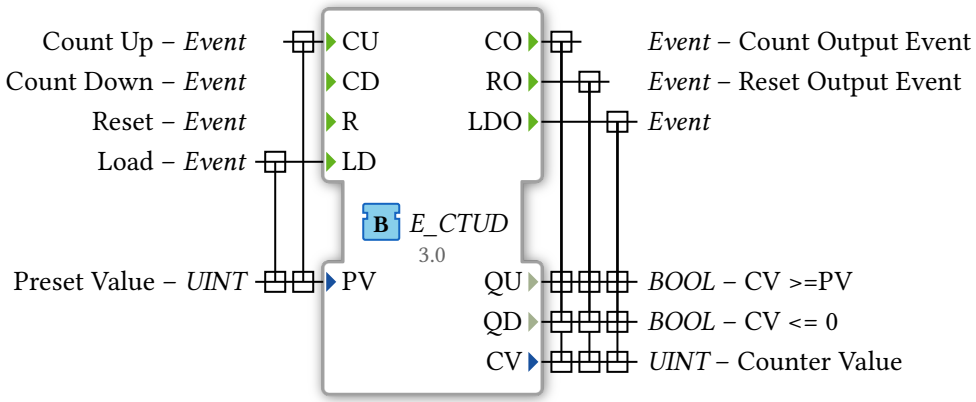
- E_CTUD: Bidirektionaler Zähler in IEC 61499 Systemen



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `E_CTUD` (Event-Driven Up-Down Counter) ist ein ereignisgesteuerter Vor- und Rückwärtszähler gemäß dem IEC 61499-Standard. Er kann einen Zählerwert basierend auf separaten Ereignissen inkrementieren, dekrementieren, zurücksetzen oder mit einem vordefinierten Wert laden. Dies macht ihn zu einem flexiblen und leistungsstarken Baustein für eine Vielzahl von Zählanwendungen.



Schnittstellenstruktur

Ereignis-Eingänge

- **CU (Count Up)**: Löst ein Aufwärtszählen aus.
  - **Verbundene Daten**: `PV`
- **CD (Count Down)**: Löst ein Abwärtszählen aus.
- **R (Reset)**: Setzt den Zähler auf 0 zurück.
- **LD (Load)**: Lädt einen neuen Wert in den Zähler.
  - **Verbundene Daten**: `PV`

Ereignis-Ausgänge

- **CO (Count Output)**: Bestätigt eine Zähloperation (`CU` oder `CD`).
  - **Verbundene Daten**: `QU`, `CV`, `QD`
- **RO (Reset Output)**: Bestätigt das Zurücksetzen des Zählers.
  - **Verbundene Daten**: `QU`, `CV`, `QD`
- **LDO (Load Output)**: Bestätigt das Laden eines neuen Zählerwertes.
  - **Verbundene Daten**: `QU`, `CV`, `QD`

Daten-Eingänge

- **PV (Preset Value)**: Der Grenzwert für `QU` bzw. der zu ladende Wert für `LD` (Datentyp: `UINT`).

Daten-Ausgänge

- **QU (Status Up)**: Ausgangs-Flag, das `TRUE` wird, wenn `CV >= PV` (Datentyp: `BOOL`).
- **QD (Status Down)**: Ausgangs-Flag, das `TRUE` wird, wenn `CV = 0` (Datentyp: `BOOL`).
- **CV (Counter Value)**: Der aktuelle Zählerstand (Datentyp: `UINT`).

Funktionsweise

Der `E_CTUD` reagiert auf vier verschiedene Ereignisse:

1. **Aufwärtszählen (CU)**: Wenn ein `CU`-Ereignis eintritt und `CV` kleiner als der Maximalwert (65535) ist, wird `CV` um 1 erhöht. Anschließend wird das `CO`-Ereignis ausgelöst.
2. **Abwärtszählen (CD)**: Wenn ein `CD`-Ereignis eintritt und `CV` größer als 0 ist, wird `CV` um 1 verringert. Anschließend wird das `CO`-Ereignis ausgelöst.
3. **Zurücksetzen (R)**: Wenn ein `R`-Ereignis eintritt, wird `CV` auf 0 gesetzt. Anschließend wird das `RO`-Ereignis ausgelöst.
4. **Laden (LD)**: Wenn ein `LD`-Ereignis eintritt, wird `CV` auf den Wert von `PV` gesetzt. Anschließend wird das `LDO`-Ereignis ausgelöst.

Nach jeder dieser Aktionen werden die Status-Flags `QU` und `QD` basierend auf dem neuen Wert von `CV` aktualisiert (`QU = (CV >= PV)` und `QD = (CV == 0)`). Die jeweiligen Ausgangsereignisse (`CO`, `RO`, `LDO`) geben dann den aktuellen Zählerstand `CV` und die beiden Status-Flags aus.

Technische Besonderheiten

- **Bidirektionale Zählung**: Der Baustein beherrscht das Auf- und Abwärtszählen in einem Block.
- **Umfassende Steuerung**: Bietet neben dem Zählen auch Funktionen zum expliziten Laden und Zurücksetzen.
- **Zwei Statusausgänge**: `QU` signalisiert das Erreichen des oberen Grenzwertes, `QD` das Erreichen des unteren Grenzwertes (0).
- **Über- und Unterlaufschutz**: Zähloperationen werden nur innerhalb der gültigen Grenzen (0 bis 65535) ausgeführt.

Anwendungsszenarien

- **Positionserfassung**: Zählen von Inkrementalgeber-Schritten in beide Richtungen.
- **Füllstandsregelung**: Erfassen von Zu- und Abflüssen in einem Tank.
- **Lagerplatzverwaltung**: Zählen von ein- und ausgelagerten Paletten.

⚖️ Vergleich mit ähnlichen Bausteinen

Merkmal	E_CTUD (Up/Down)	E_CTU (Up)	E_CTD (Down)

Zählrichtung	Auf & Ab	Nur Auf	Nur Ab
Reset (auf 0)	Ja (`R`)	Ja (`R`)	Nein
Laden (auf PV)	Ja (`LD`)	Nein	Ja (`LD`)
Status Oben	`QU` (`CV >= PV`)	`Q` (`CV >= PV`)	Nein
Status Unten	`QD` (`CV = 0`)	Nein	`Q` (`CV = 0`)

🛠️ Zugehörige Übungen

- Uebung_082

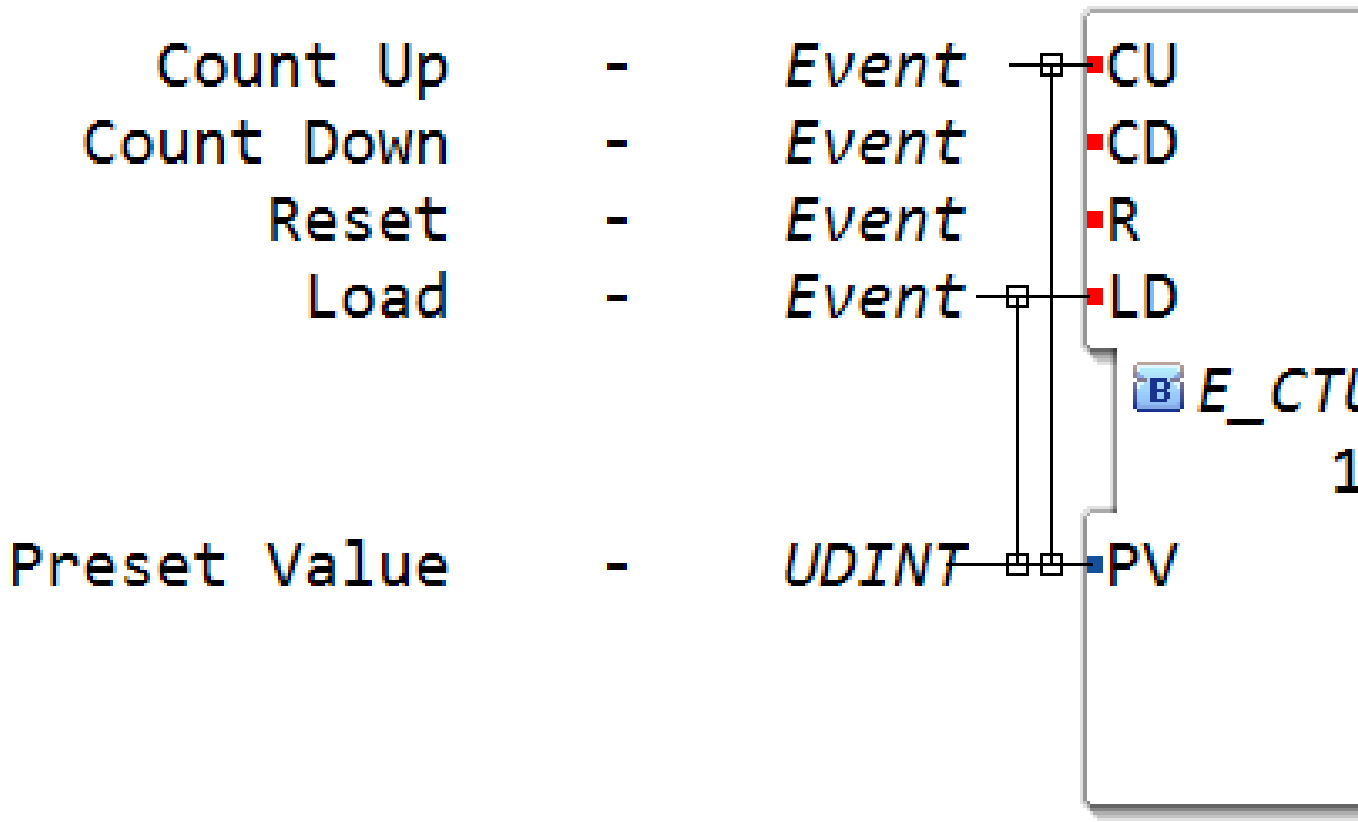
Fazit

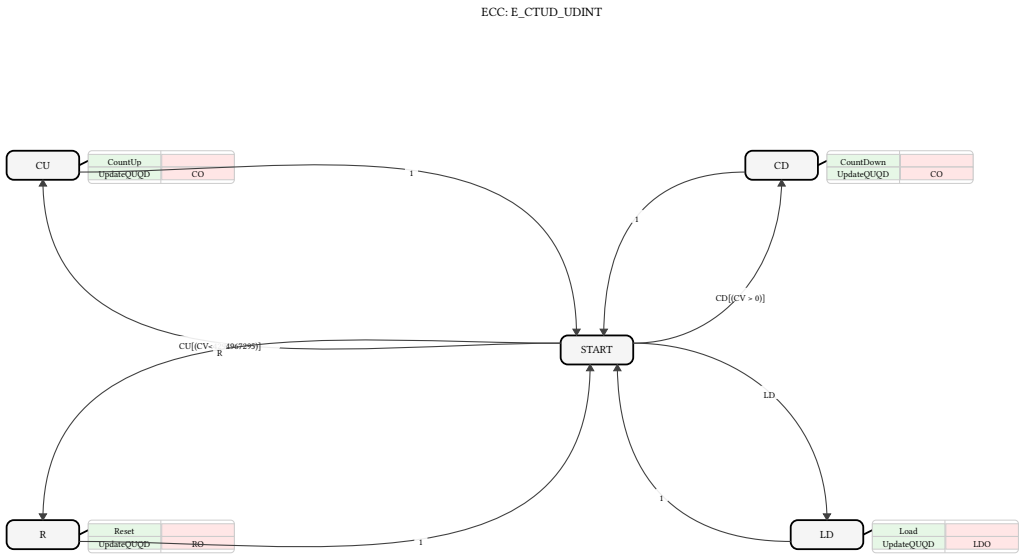
Der `E_CTUD` ist ein universeller Zählerbaustein, der die Funktionalität eines reinen Aufwärts- und Abwärtszählers kombiniert und erweitert. Durch seine vier Steuerereignisse (`CU`, `CD`, `R`, `LD`) und die beiden Statusausgänge (`QU`, `QD`) bietet er maximale Flexibilität für komplexe Zähl- und Überwachungsaufgaben in der industriellen Automatisierung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de

2.2.3.7 E_CTUD_UDINT

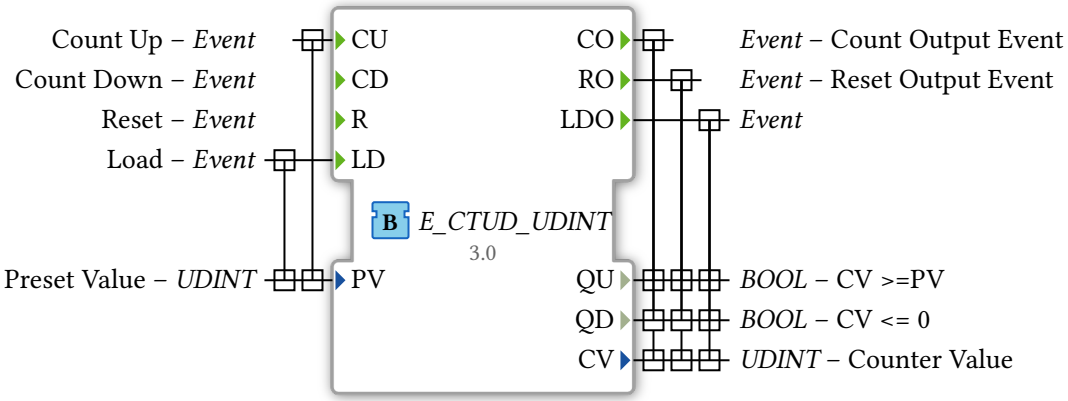




• – ○ • – ○ • – ○ •

Einleitung

Der `E_CTUD_UDINT` ist eine Variante des `E_CTUD`-Zählers, die den Datentyp `UDINT` (Unsigned Double Integer, 32-Bit) verwendet. Dieser ereignisgesteuerte Vor- und Rückwärtszähler kann einen sehr großen Zählerbereich abdecken. Er kann einen Zählerwert basierend auf separaten Ereignissen inkrementieren, dekrementieren, zurücksetzen oder mit einem vordefinierten Wert laden.



Schnittstellenstruktur

Ereignis-Eingänge

- **CU (Count Up)**: Löst ein Aufwärtszählen aus.
  - **Verbundene Daten**: `PV`
- **CD (Count Down)**: Löst ein Abwärtszählen aus.
- **R (Reset)**: Setzt den Zähler auf 0 zurück.
- **LD (Load)**: Lädt einen neuen Wert in den Zähler.
  - **Verbundene Daten**: `PV`

Ereignis-Ausgänge

- **CO (Count Output)**: Bestätigt eine Zähloperation (`CU` oder `CD`).
  - **Verbundene Daten**: `QU`, `CV`, `QD`

- **RO (Reset Output)**: Bestätigt das Zurücksetzen des Zählers.
  - **Verbundene Daten**: `QU`, `CV`, `QD`
- **LDO (Load Output)**: Bestätigt das Laden eines neuen Zählerwertes.
  - **Verbundene Daten**: `QU`, `CV`, `QD`

Daten-Eingänge

- **PV (Preset Value)**: Der Grenzwert für `QU` bzw. der zu ladende Wert für `LD` (Datentyp: `UDINT`).

Daten-Ausgänge

- **QU (Status Up)**: Ausgangs-Flag, das `TRUE` wird, wenn `CV >= PV` (Datentyp: `BOOL`).
- **QD (Status Down)**: Ausgangs-Flag, das `TRUE` wird, wenn `CV = 0` (Datentyp: `BOOL`).
- **CV (Counter Value)**: Der aktuelle Zählerstand (Datentyp: `UDINT`).

Funktionsweise

Der `E_CTUD_UDINT` reagiert auf vier verschiedene Ereignisse:

1. **Aufwärtszählen (CU)**: Wenn ein `CU`-Ereignis eintritt und `CV` kleiner als der Maximalwert (4.294.967.295) ist, wird `CV` um 1 erhöht. Anschließend wird das `CO`-Ereignis ausgelöst.
2. **Abwärtszählen (CD)**: Wenn ein `CD`-Ereignis eintritt und `CV` größer als 0 ist, wird `CV` um 1 verringert. Anschließend wird das `CO`-Ereignis ausgelöst.
3. **Zurücksetzen (R)**: Wenn ein `R`-Ereignis eintritt, wird `CV` auf 0 gesetzt. Anschließend wird das `RO`-Ereignis ausgelöst.
4. **Laden (LD)**: Wenn ein `LD`-Ereignis eintritt, wird `CV` auf den Wert von `PV` gesetzt. Anschließend wird das `LDO`-Ereignis ausgelöst.

Nach jeder dieser Aktionen werden die Status-Flags `QU` und `QD` basierend auf dem neuen Wert von `CV` aktualisiert (`QU = (CV >= PV)` und `QD = (CV == 0)`). Die jeweiligen Ausgangsereignisse (`CO`, `RO`, `LDO`) geben dann den aktuellen Zählerstand `CV` und die beiden Status-Flags aus.

Technische Besonderheiten

- **Großer Zählbereich**: Durch die Verwendung von `UDINT` kann der Zähler Werte von 0 bis 4.294.967.295 annehmen.
- **Bidirektionale Zählung**: Der Baustein beherrscht das Auf- und Abwärtszählen in einem Block.
- **Umfassende Steuerung**: Bietet neben dem Zählen auch Funktionen zum expliziten Laden und Zurücksetzen.
- **Über- und Unterlaufschutz**: Zähloperationen werden nur innerhalb der gültigen `UDINT`-Grenzen (0 bis 4.294.967.295) ausgeführt.

Anwendungsszenarien

- **Gesamtzähler**: Erfassung von Gesamtstückzahlen oder Betriebsstunden über die gesamte Lebensdauer einer Maschine, wo ein 16-Bit-Zähler nicht ausreicht.
- **Energiemessung**: Zählen von Impulsen eines Energiezählers (z.B. Wh oder kWh) über lange Zeiträume.
- **Hochauflösende Positionserfassung**: Zählen von sehr vielen Inkrementen eines hochauflösenden Encoders.

🛠️ Zugehörige Übungen

- Uebung_009
- Uebung_034b
- Uebung_083

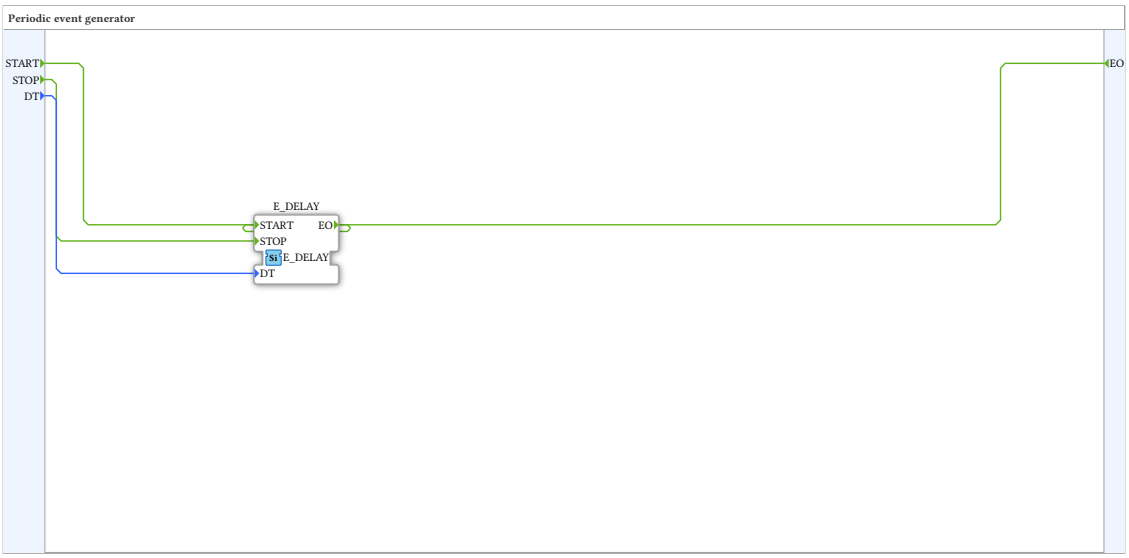
Fazit

Der `E_CTUD_UDINT` ist die `UDINT`-Variante des universellen `E_CTUD`-Zählers. Er bietet dieselbe Funktionalität, aber mit einem wesentlich größeren Zählbereich (32-Bit). Dies macht ihn zur idealen Wahl für Anwendungen, bei denen der Zählerstand über die Grenze eines 16-Bit-`UINT`-Zählers hinauswachsen kann. Seine robuste, ereignisgesteuerte Natur und die umfassenden Steuerungs- und Statusfunktionen bleiben dabei erhalten.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

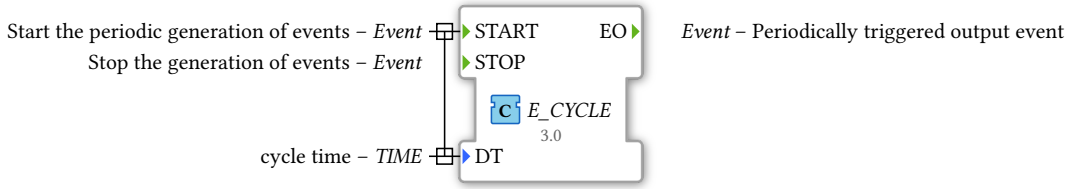
- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de

2.2.3.8 E_CYCLE



Einleitung

Der `E_CYCLE` ist ein Funktionsbaustein nach IEC 61499, der als periodischer Ereignisgenerator dient. Nach dem Starten feuert der Baustein in einem festgelegten Zeitintervall (`DT`) wiederholt ein Ausgangsereignis (`EO`). Er ist das grundlegende Werkzeug zur Erstellung von Taktgebern und zyklischen Abläufen.



Schnittstellenstruktur

Ereignis-Eingänge

- **START**: Startet die periodische Generierung von `EO`-Ereignissen.
  - **Verbundene Daten**: `DT`
- **STOP**: Stoppt die Generierung von `EO`-Ereignissen.

Ereignis-Ausgänge

- **EO (Event Output)**: Das zyklisch ausgelöste Ausgangsereignis.

Daten-Eingänge

- **DT (Delay Time)**: Die Zeitdauer zwischen den einzelnen `EO`-Ereignissen (Datentyp: `TIME`).

Funktionsweise

Die Funktionalität des `E_CYCLE` basiert intern auf einer Rückkopplungsschleife mit einem `E_DELAY`-Baustein.

1. **Starten des Zyklus**:
   - Ein `START`-Ereignis am Eingang löst den internen Timer mit der am `DT`-Eingang angegebenen Zeitdauer aus.
   - Nach Ablauf dieser Zeit `DT` wird das erste `EO`-Ereignis am Ausgang ausgelöst.
2. **Zyklischer Betrieb**:

- Das ausgelöste `EO`-Ereignis wird intern sofort wieder auf den Starteingang des Timers zurückgekoppelt.
- Dadurch wird der Timer umgehend neu gestartet, und nach einer weiteren Zeitspanne `DT` wird das nächste `EO`-Ereignis ausgelöst.
- Dieser Prozess wiederholt sich kontinuierlich und erzeugt so eine periodische Folge von `EO`-Ereignissen im Abstand von `DT`.

3. **Stoppen des Zyklus**:
   - Ein `STOP`-Ereignis am Eingang unterbricht die interne Rückkopplungsschleife.
   - Der Timer wird gestoppt und es werden keine weiteren `EO`-Ereignisse mehr generiert, bis ein erneutes `START`-Ereignis eintrifft.

Technische Besonderheiten

- **Ereignisgenerator**: Der Baustein erzeugt eine Kette von Ereignissen, keinen kontinuierlichen Zustand (wie z.B. eine Rechteckwelle). Um ein Blinken zu realisieren, müsste das `EO`-Ereignis beispielsweise einen `E_T_FF` (Toggle Flip-Flop) ansteuern.
- **Präzise Zeitsteuerung**: Ermöglicht eine genaue Definition der Zykluszeit über den `TIME`-Datentyp.
- **Ereignisgesteuerte Architektur**: Die gesamte Steuerung (Start/Stopp) erfolgt rein ereignisbasiert.

Anwendungsszenarien

- **Taktgeber**: Periodisches Anstoßen von anderen Funktionsblöcken in einem festen Takt (z.B. für Abtastungen oder Berechnungen).
- **Blinker/Warnleuchten**: Als Impulsgeber für einen nachgeschalteten Flip-Flop, um ein visuelles Signal zu erzeugen.
- **Watchdog**: Periodisches Senden eines „Lebenszeichens“. Wenn das Signal ausbleibt, kann ein Fehler erkannt werden.
- **Testautomatisierung**: Generierung von periodischen Stimuli für zu testende Systeme.

⚖️ Vergleich mit ähnlichen Bausteinen

Merkmal	E_CYCLE	E_DELAY	E_PULSE (hypothetisch)
Funktion	Periodischer Taktgeber	Einmalige Verzögerung	Einmaliger Impuls fester Dauer
Wiederholung	Kontinuierlich	Einmalig pro `START`-Ereignis	Einmalig pro `REQ`-Ereignis
Steuerung	START/STOP	START/STOP	REQ

🛠️ Zugehörige Übungen

- Uebung_007
- Uebung_007_AX
- Uebung_007a1
- Uebung_007a1_AX
- Uebung_007a2
- Uebung_007a2_AX
- Uebung_007a3
- Uebung_008
- Uebung_009
- Uebung_020c3
- Uebung_020c3_AX
- Uebung_020e2
- Uebung_020e2_AX
- Uebung_020f2
- Uebung_020f2_AX
- Uebung_084
- Uebung_152
- Uebung_153

Fazit

Der `E_CYCLE`-Baustein ist ein essenzielles Werkzeug für alle Anwendungen, die eine periodische oder zyklische Logik erfordern. Er implementiert auf einfache Weise einen Taktgeber, dessen Frequenz über den `DT`-Parameter präzise eingestellt werden kann. Sein rein ereignisbasiertes Verhalten macht ihn zu einer effizienten und fundamentalen Komponente in der IEC 61499-Systemarchitektur.

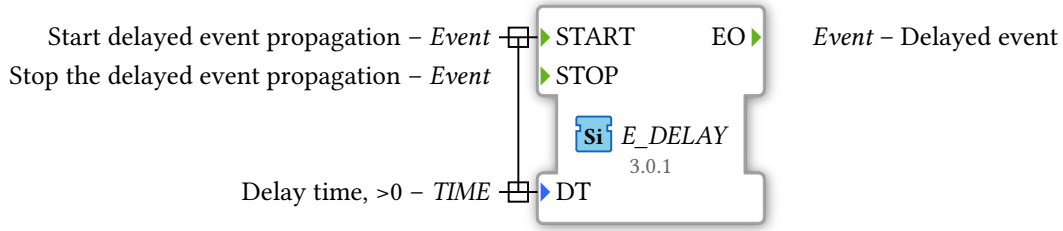
2.2.3.9 E_DELAY

🎧 Podcast

- E_DELAY in IEC 61499: Präzise, Abbrechbare Zeitverzögerung in Steuerungssystemen

Einleitung

Der **E_DELAY** ist ein zentraler Funktionsbaustein der IEC 61499 Norm für industrielle Steuerungssysteme. Als ereignisgesteuerter Timer ermöglicht er die verzögerte Auslösung von Ereignissen mit konfigurierbarer Zeitspanne und Abbruchmöglichkeit. Dieser Baustein findet breite Anwendung in zeitgesteuerten Automatisierungsprozessen.



Struktur des E_DELAY-Bausteins

Schnittstelle (Interface)

Ereignis-Eingänge:

- `START`: Initialisiert die Verzögerung
- `STOP`: Bricht die aktive Verzögerung ab

Ereignis-Ausgänge:

- `EO` (Event Output): Wird nach Ablauf der Verzögerung ausgelöst

Daten-Eingänge:

- `DT` (Delay Time): TIME-Wert für die Verzögerungsdauer

Funktionsweise

1. **Verzögerungsstart:**
   - Bei `START`-Ereignis beginnt der Timer mit der eingestellten Zeit `DT`
   - Während der Laufzeit werden weitere `START`-Ereignisse ignoriert
2. **Verzögerungsabbruch:**
   - `STOP`-Ereignis bricht die aktive Verzögerung sofort ab
   - Kein `EO`-Ereignis wird generiert
3. **Verzögerungsabschluss:**
   - Nach exakt `DT` wird das `EO`-Ereignis ausgelöst
   - Bei DT ≤ T#0s erfolgt sofortige Auslösung ohne Verzögerung

Technische Besonderheiten

✓ **Präzise Zeitsteuerung** (millisekundengenaue Verzögerung) ✓ **Abbruchfunktion** für flexible Prozesssteuerung ✓ **Ereignisgesteuert** (kein zyklischer Aufruf erforderlich) ✓ **Echtzeitfähig** für industrielle Anwendungen

Anwendungsszenarien

- **Maschinensicherheit**: Verzögerte Abschaltung von Anlagen

- **Prozesssteuerung**: Zeitgesteuerte Prozessschritte
- **Alarmmanagement**: Verzögerte Störmeldungen
- **Testautomation**: Zeitgesteuerte Testsequenzen

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	E_DELAY	E_CYCLE	E_PULSE
Funktionsprinzip	Einmalverzögerung	Dauerzyklus	Einmalimpuls
Abbruchmöglichkeit	Ja	Ja	Nein
Zeitparameter	DT (Verzögerung)	DT (Zykluszeit)	PT (Impulsdauer)

🛠️ Zugehörige Übungen

- Uebung_018
- Uebung_020b
- Uebung_020b_AX
- Uebung_020d
- Uebung_020d_AX
- Uebung_024
- Uebung_025
- Uebung_026
- Uebung_160b
- Uebung_160b_AX

Fazit

Der E_DELAY-Baustein ist ein unverzichtbares Werkzeug für zeitgesteuerte Automatisierungsprozesse. Seine Hauptvorteile sind:

- Präzise und flexible Zeitsteuerung
- Sofortige Reaktion auf Steuerereignisse
- Robuste Integration in verteilte Steuerungsarchitekturen

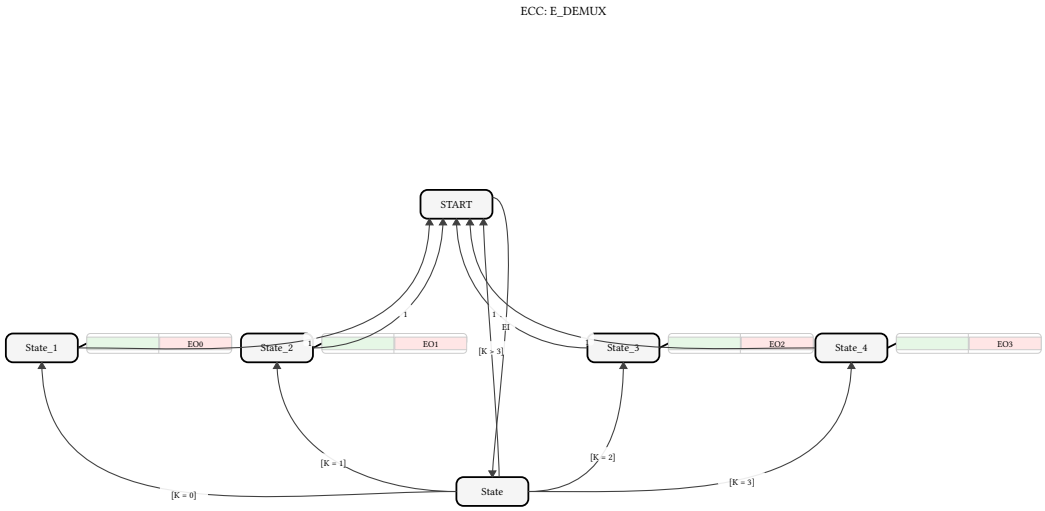
Durch seine zuverlässige Funktionsweise und Standardkonformität nach IEC 61499 eignet er sich ideal für Sicherheitsanwendungen und komplexe Prozesssteuerungen in industriellen Umgebungen. Die Abbruchfunktion macht ihn besonders wertvoll für flexible Produktionsszenarien.

2.2.3.10 E_DEMUX

Einleitung

Der `E_DEMUX` (Event Demultiplexer) ist ein Funktionsbaustein nach IEC 61499, der ein einzelnes Eingangereignis (`EI`) an einen von mehreren Ausgängen weiterleitet. Die Auswahl des spezifischen Ausgangs wird durch den Wert einer Eingangsvariable (`K`) bestimmt. Diese Version des Bausteins ist ein 1-zu-4 Demultiplexer.

Schnittstellenstruktur



Ereignis-Eingänge

- **EI (Event Input)**: Das Eingangereignis, das verteilt werden soll.
  - **Verbundene Daten**: `K`

Ereignis-Ausgänge

- **EO0**: Wird ausgelöst, wenn `EI` ankommt und `K = 0`.
- **EO1**: Wird ausgelöst, wenn `EI` ankommt und `K = 1`.
- **EO2**: Wird ausgelöst, wenn `EI` ankommt und `K = 2`.
- **EO3**: Wird ausgelöst, wenn `EI` ankommt und `K = 3`.

Daten-Eingänge

- **K**: Der Auswahlindex, der bestimmt, welcher Ausgang ausgelöst wird (Datentyp: `UINT`).

Funktionsweise

1. **Ereignisempfang**: Der Baustein wartet auf ein Ereignis am Eingang `EI`.
2. **Auswahl**: Wenn das `EI`-Ereignis eintrifft, wird der Wert der Datenvariable `K` ausgewertet.
3. **Weiterleitung**:
   - Ist `K` = 0, wird das Ereignis an `EO0` weitergeleitet.
   - Ist `K` = 1, wird das Ereignis an `EO1` weitergeleitet.
   - Ist `K` = 2, wird das Ereignis an `EO2` weitergeleitet.
   - Ist `K` = 3, wird das Ereignis an `EO3` weitergeleitet.
4. **Ungültiger Index**: Wenn der Wert von `K` außerhalb des gültigen Bereichs [0, 3] liegt, wird kein Ausgangsereignis ausgelöst, und das `EI`-Ereignis wird verworfen.

Das Eingangsereignis wird somit immer exklusiv an genau einen Ausgang weitergeleitet, sofern der Index `K` gültig ist.

Technische Besonderheiten

- **1-zu-4 Verteilung**: Dieser Baustein verteilt ein Ereignis auf vier mögliche Ausgänge.
- **Index-gesteuert**: Die Logik basiert auf einem numerischen Index.
- **Deterministisches Verhalten**: Die Weiterleitung ist klar und eindeutig definiert, was für zuverlässige Steuerungen sorgt.

Anwendungsszenarien

- **Zustandsmaschinen**: Auswahl des nächsten Zustandsübergangs basierend auf einem berechneten Index.
- **Modus-Umschaltung**: Aktivierung unterschiedlicher Anlagenteile je nach gewähltem Betriebsmodus (`K` = Modus-Nummer).
- **Sequenzer/Schrittketten**: Aktivierung des nächsten Schritts in einer Kette.
- **Fehler-Routing**: Weiterleiten eines allgemeinen Fehlerereignisses an einen spezifischen Handler basierend auf einem Fehlercode (`K` = Fehlercode).

⚖️ **Vergleich mit ähnlichen Bausteinen**

Merkmal	E_DEMUX (dieser)	E_MUX	E_SWITCH
Funktionsprinzip	1:4 Verteilung	n:1 Zusammenführung	1:2 Verteilung
Steuerung	Index `K` [0-3]	Index `K`	`BOOL`-Bedingung `G`
Ereignisfluss	Aufteilung	Zusammenführung	Bedingte Weiche

Hinweis: Es existieren auch andere Varianten wie `E_DEMUX_2` und `E_DEMUX_8` für 2 bzw. 8 Ausgänge.

🛠️ **Zugehörige Übungen**

- Uebung_040
- Uebung_040_AX
- Uebung_087

Fazit

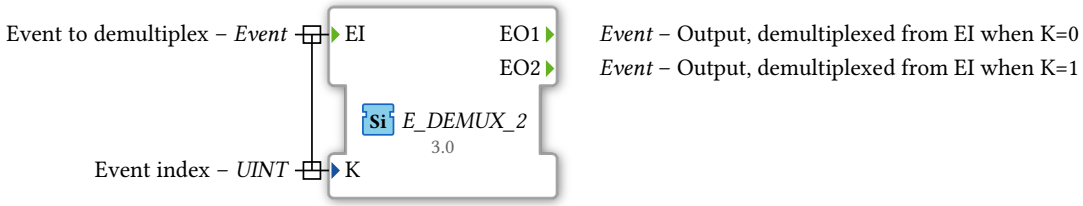
Der `E_DEMUX` ist ein fundamentaler Baustein für die Steuerung des Ereignisflusses in IEC 61499-Anwendungen. Er ermöglicht eine klare, indexbasierte Aufteilung von Ereignisströmen und ist damit ein Schlüsselwerkzeug für die Implementierung von Zustandslogiken, Modus-Umschaltungen und Sequenzsteuerungen.

2.2.3.11 E_DEMUX_2

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `E_DEMUX_2` (Event Demultiplexer) ist ein Funktionsbaustein nach IEC 61499, der ein einzelnes Eingangereignis (`EI`) an einen von zwei Ausgängen weiterleitet. Die Auswahl des Ausgangs wird durch den Wert der Eingangsvariable `K` bestimmt.



Schnittstellenstruktur

Ereignis-Eingänge

- **EI (Event Input)**: Das Eingangereignis, das verteilt werden soll.
  - **Verbundene Daten**: `K`

Ereignis-Ausgänge

- **EO1**: Wird ausgelöst, wenn `EI` ankommt und der Auswahlindex `K = 0` ist.
- **EO2**: Wird ausgelöst, wenn `EI` ankommt und der Auswahlindex `K = 1` ist.

Daten-Eingänge

- **K**: Der 0-basierte Auswahlindex, der bestimmt, welcher Ausgang ausgelöst wird (Datentyp: `UINT`).

Funktionsweise

1. **Ereignisempfang**: Der Baustein wartet auf ein Ereignis am Eingang `EI`.
2. **Auswahl**: Wenn das `EI`-Ereignis eintrifft, wird der Wert der Datenvariable `K` ausgewertet.
3. **Weiterleitung**:
   - Ist `K` = 0, wird das Ereignis an `EO1` weitergeleitet.
   - Ist `K` = 1, wird das Ereignis an `EO2` weitergeleitet.
4. **Ungültiger Index**: Wenn der Wert von `K` außerhalb des gültigen Bereichs [0, 1] liegt, wird kein Ausgangsereignis ausgelöst.

Das Eingangsereignis wird somit immer exklusiv an genau einen Ausgang weitergeleitet, sofern der Index `K` gültig ist.

Technische Besonderheiten

- **1-zu-2 Verteilung**: Dieser Baustein verteilt ein Ereignis auf zwei mögliche Ausgänge.
- **Index-gesteuert**: Die Logik basiert auf einem numerischen Index (`K`).
- **Verwirrende Namensgebung**: Man beachte, dass die Ausgänge 1-basiert (`EO1`, `EO2`) benannt sind, der Auswahlindex `K` aber 0-basiert ist (`K=0` für `EO1`, `K=1` for `EO2`).
- **Generischer Baustein**: Die Funktionalität wird durch die generische Klasse `GEN_E_DEMUX` zur Verfügung gestellt.

Anwendungsszenarien

- **Bedingte Verzweigung**: Aufteilen eines Ereignisflusses basierend auf einer einfachen Bedingung, die zu einem Ergebnis von 0 oder 1 führt.
- **Zustandsmaschinen**: Auswahl zwischen zwei möglichen nächsten Zuständen.
- **Aktivierung/Deaktivierung**: Weiterleiten eines Ereignisses an einen „Ein“- oder „Aus“-Pfad.

Fazit

Der `E_DEMUX_2` ist eine spezialisierte Version des Demultiplexers für Fälle, in denen ein Ereignis auf genau zwei Pfade aufgeteilt werden muss. Er ist nützlich für einfache binäre Entscheidungen im Ereignisfluss. Die inkonsistente Benennung der Ausgänge im Verhältnis zum Indexwert erfordert besondere Aufmerksamkeit bei der Implementierung.

2.2.3.12 E_DEMUX_4

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `E_DEMUX_4` (Event Demultiplexer) ist ein Funktionsbaustein nach IEC 61499, der ein einzelnes Eingangereignis (`EI`) an einen von vier Ausgängen weiterleitet. Die Auswahl des Ausgangs wird durch den Wert der Eingangsvariable `K` bestimmt.

Schnittstellenstruktur

Ereignis-Eingänge

- **EI (Event Input)**: Das Eingangereignis, das verteilt werden soll.
  - **Verbundene Daten**: `K`

Ereignis-Ausgänge

- **EO1**: Wird ausgelöst, wenn `EI` ankommt und der Auswahlindex `K = 0` ist.
- **EO2**: Wird ausgelöst, wenn `EI` ankommt und der Auswahlindex `K = 1` ist.

- **EO3**: Wird ausgelöst, wenn `EI` ankommt und der Auswahlindex `K = 2` ist.
- **EO4**: Wird ausgelöst, wenn `EI` ankommt und der Auswahlindex `K = 3` ist.

Daten-Eingänge

- **K**: Der 0-basierte Auswahlindex, der bestimmt, welcher Ausgang ausgelöst wird (Datentyp: `UINT`).

Funktionsweise

1. **Ereignisempfang**: Der Baustein wartet auf ein Ereignis am Eingang `EI`.
2. **Auswahl**: Wenn das `EI`-Ereignis eintrifft, wird der Wert der Datenvariable `K` ausgewertet.
3. **Weiterleitung**:
   - Ist `K` = 0, wird das Ereignis an `EO1` weitergeleitet.
   - Ist `K` = 1, wird das Ereignis an `EO2` weitergeleitet.
   - Ist `K` = 2, wird das Ereignis an `EO3` weitergeleitet.
   - Ist `K` = 3, wird das Ereignis an `EO4` weitergeleitet.
4. **Ungültiger Index**: Wenn der Wert von `K` außerhalb des gültigen Bereichs [0, 3] liegt, wird kein Ausgangsereignis ausgelöst.

Technische Besonderheiten

- **1-zu-4 Verteilung**: Dieser Baustein verteilt ein Ereignis auf vier mögliche Ausgänge.
- **Index-gesteuert**: Die Logik basiert auf einem numerischen Index (`K`).
- **Verwirrende Namensgebung**: Man beachte, dass die Ausgänge 1-basiert (`EO1` bis `EO4`) benannt sind, der Auswahlindex `K` aber 0-basiert ist (`K=0` für `EO1`, `K=1` for `EO2`, usw.).
- **Generischer Baustein**: Die Funktionalität wird durch die generische Klasse `GEN_E_DEMUX` zur Verfügung gestellt.

Anwendungsszenarien

- **Zustandsmaschinen**: Auswahl des nächsten Zustandsübergangs basierend auf einem berechneten Index.
- **Modus-Umschaltung**: Aktivierung unterschiedlicher Anlagenteile je nach gewähltem Betriebsmodus (`K` = Modus-Nummer).
- **Sequenzer/Schrittketten**: Aktivierung eines von vier möglichen nächsten Schritten.

🛠️ Zugehörige Übungen

- Uebung_040_2
- Uebung_087a1
- Uebung_087a2

Fazit

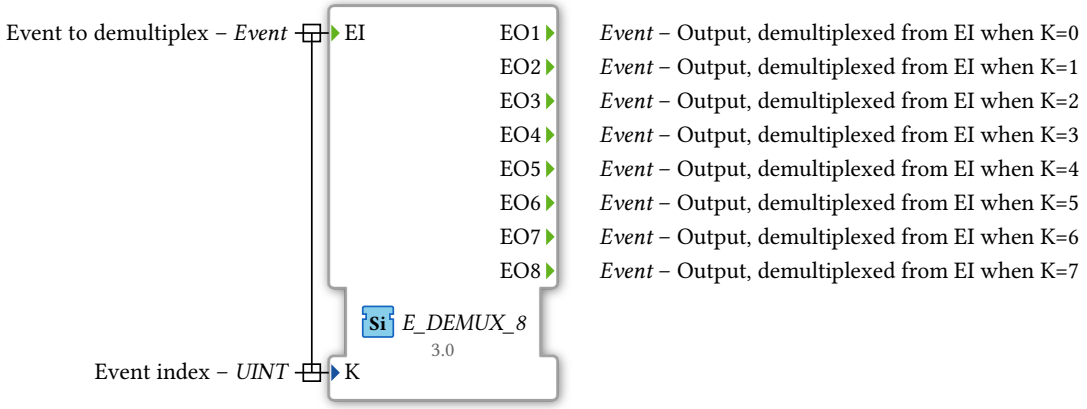
Der `E_DEMUX_4` ist eine Standard-Implementierung des Demultiplexer-Prinzips für vier Ausgänge. Er ist nützlich für die Aufteilung eines Ereignisflusses in bis zu vier Pfade. Die inkonsistente Benennung der Ausgänge im Verhältnis zum Indexwert erfordert besondere Aufmerksamkeit bei der Implementierung.

2.2.3.13 E_DEMUX_8

• − ◦ • − ◦ • − ◦ •

Einleitung

Der `E_DEMUX_8` (Event Demultiplexer) ist ein Funktionsbaustein nach IEC 61499, der ein einzelnes Eingangereignis (`EI`) an einen von acht Ausgängen weiterleitet. Die Auswahl des Ausgangs wird durch den Wert der Eingangsvariable `K` bestimmt.



Schnittstellenstruktur

Ereignis-Eingänge

- **EI (Event Input)**: Das Eingangereignis, das verteilt werden soll.
  - **Verbundene Daten**: `K`

Ereignis-Ausgänge

- **EO1**: Wird ausgelöst, wenn `EI` ankommt und `K = 0` ist.
- **EO2**: Wird ausgelöst, wenn `EI` ankommt und `K = 1` ist.
- **EO3**: Wird ausgelöst, wenn `EI` ankommt und `K = 2` ist.
- **EO4**: Wird ausgelöst, wenn `EI` ankommt und `K = 3` ist.
- **EO5**: Wird ausgelöst, wenn `EI` ankommt und `K = 4` ist.
- **EO6**: Wird ausgelöst, wenn `EI` ankommt und `K = 5` ist.
- **EO7**: Wird ausgelöst, wenn `EI` ankommt und `K = 6` ist.
- **EO8**: Wird ausgelöst, wenn `EI` ankommt und `K = 7` ist.

Daten-Eingänge

- **K**: Der 0-basierte Auswahlindex, der bestimmt, welcher Ausgang ausgelöst wird (Datentyp: `UINT`).

Funktionsweise

1. **Ereignisempfang**: Der Baustein wartet auf ein Ereignis am Eingang `EI`.
2. **Auswahl**: Wenn das `EI`-Ereignis eintrifft, wird der Wert der Datenvariable `K` ausgewertet.
3. **Weiterleitung**: Das Ereignis wird an den Ausgang `EO(K+1)` weitergeleitet. Zum Beispiel:
   - Ist `K` = 0, wird das Ereignis an `EO1` weitergeleitet.
   - Ist `K` = 7, wird das Ereignis an `EO8` weitergeleitet.
4. **Ungültiger Index**: Wenn der Wert von `K` außerhalb des gültigen Bereichs [0, 7] liegt, wird kein Ausgangsereignis ausgelöst.

Technische Besonderheiten

- **1-zu-8 Verteilung**: Dieser Baustein verteilt ein Ereignis auf acht mögliche Ausgänge.
- **Index-gesteuert**: Die Logik basiert auf einem numerischen Index (`K`).
- **Verwirrende Namensgebung**: Man beachte, dass die Ausgänge 1-basiert (`EO1` bis `EO8`) benannt sind, der Auswahlindex `K` aber 0-basiert ist (`K=0` für `EO1`, `K=7` for `EO8`, usw.).
- **Generischer Baustein**: Die Funktionalität wird durch die generische Klasse `GEN_E_DEMUX` zur Verfügung gestellt.

Anwendungsszenarien

- **Zustandsmaschinen**: Auswahl des nächsten Zustandsübergangs aus bis zu acht Möglichkeiten.
- **Modus-Umschaltung**: Aktivierung unterschiedlicher Anlagenteile je nach gewähltem Betriebsmodus.
- **Kommunikations-Router**: Weiterleiten von Nachrichten oder Befehlen an einen von acht Kanälen.

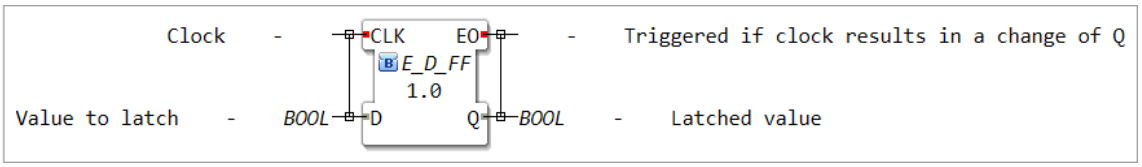
🛠️ Zugehörige Übungen

- Uebung_006c
- Uebung_041

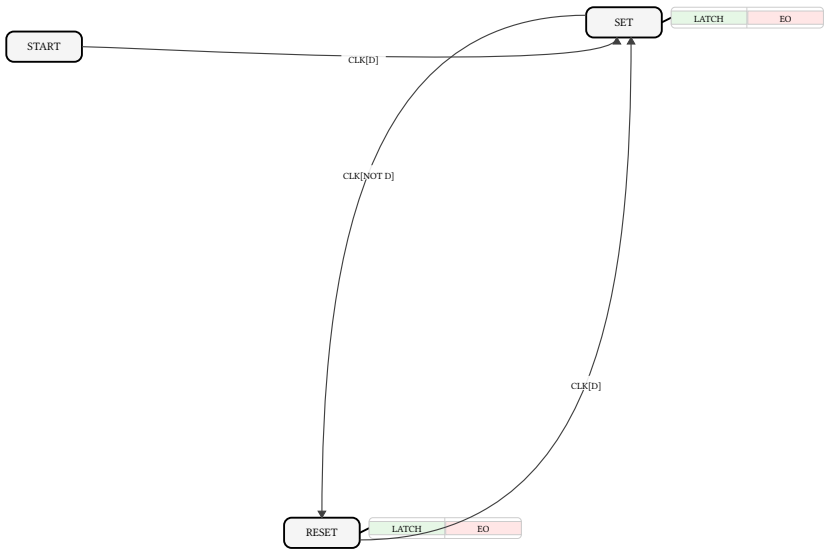
Fazit

Der `E_DEMUX_8` ist eine nützliche Implementierung des Demultiplexer-Prinzips für Anwendungsfälle mit bis zu acht verschiedenen Ereignispfaden. Wie bei den anderen `E_DEMUX`-Bausteinen erfordert die inkonsistente Benennung der Ausgänge im Verhältnis zum Indexwert besondere Aufmerksamkeit bei der Implementierung.

2.2.3.14 E_D_FF



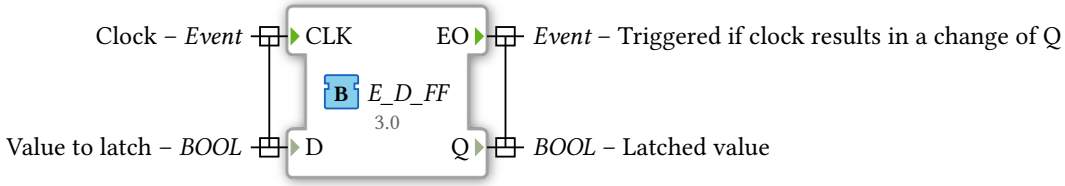
ECC: E_D_FF



• – ○ • – ○ • – ○ •

Einleitung

Die IEC 61499 ist ein internationaler Standard für die Modellierung verteilter industrieller Steuerungssysteme. Der **E_D_FF**-Baustein (Data Flip-Flop) ist ein grundlegendes Speicherelement in dieser Norm, das digitale Signale synchron zu einem Taktsignal speichert. Dieser Funktionsbaustein spielt eine zentrale Rolle in Zustandsmaschinen und Speicheranwendungen industrieller Automatisierungslösungen.



Struktur des E_D_FF-Bausteins

Schnittstelle (Interface)

Ereignis-Eingänge:

- `CLK` (Clock): Taktereignis, das die Übernahme des Daten-Eingangs `D` auslöst.

  - **Verbundene Daten**: `D`

Ereignis-Ausgänge:

- `EO` (Event Output): Wird ausgelöst, wenn der `CLK`-Impuls zu einer Änderung des Ausgangs `Q` führt.
  - **Verbundene Daten**: `Q`

Daten-Eingänge:

- `D`: Der zu speichernde Datenwert (Datentyp: `BOOL`).

Daten-Ausgänge:

- `Q`: Der aktuell gespeicherte Wert (Datentyp: `BOOL`).

Funktionsweise

1. **Datenspeicherung:**
   - Bei jeder positiven Flanke des `CLK`-Signals wird der Wert von `D` in `Q` übernommen
   - Bei Wertänderung wird das `EO`-Ereignis ausgelöst
2. **Datenhaltung:**
   - Zwischen den Taktflanken bleibt der gespeicherte Wert `Q` stabil
   - Änderungen am Eingang `D` haben ohne Taktflanke keine Auswirkung

Technische Besonderheiten

✓ **Taktgesteuerte Speicherung** für synchrone Systeme ✓ **Ereignisausgang** für Änderungsdetektion ✓ **Deterministisches Verhalten** in Echtzeitsystemen ✓ **Einfache Integration** in IEC 61499-Applikationen

Anwendungsszenarien

- **Zustandsspeicherung** in Automatisierungsprozessen
- **Eingabepufferung** für Bedienereingaben
- **Flankenerkennung** in Signalverarbeitungsketten
- **Synchronisation** zwischen asynchronen Systemteilen

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	E_D_FF	E_SR	E_R_TRIG
Speichertyp	D-Flip-Flop	SR-Latch	Flankendetektor
Taktung	Erforderlich	Keine	Keine
Datenhaltung	Ja	Ja	Nein
Ereignisausgang	Bei Änderung	Bei Set/Reset	Bei Flanke

🛠️ Zugehörige Übungen

- Uebung_071a
- Uebung_071b
- Uebung_072b
- Uebung_085

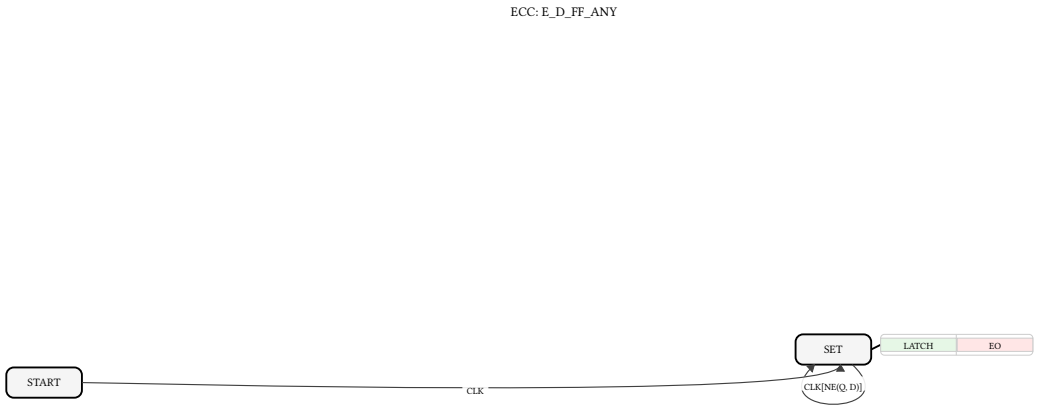
Fazit

Der E_D_FF-Baustein stellt ein essentielles Speicherelement für IEC 61499-basierte Steuerungssysteme dar. Seine Hauptvorteile sind:

- Zuverlässige taktsynchrone Datenspeicherung
- Sofortige Rückmeldung von Zustandsänderungen
- Robuste Integration in verteilte Steuerungsarchitekturen

Durch seine einfache aber wirkungsvolle Funktionalität bildet er die Grundlage für komplexere Speicher- und Zustandssteuerungen in industriellen Automatisierungslösungen. Die strikte Einhaltung der IEC 61499-Standards gewährleistet dabei die problemlose Interoperabilität mit anderen Funktionsbausteinen.

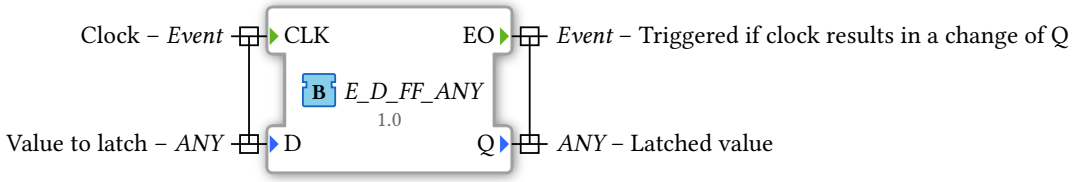
2.2.3.15 E_D_FF_ANY



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `E_D_FF_ANY` ist die generisch typisierte Variante von `E_D_FF`: Während `E_D_FF` nur `BOOL`-Werte zwischenspeichert, akzeptiert `E_D_FF_ANY` einen Eingang `D` vom Datentyp `ANY` und kann so für beliebige Datentypen (z. B. `TIME`, `DINT`, `REAL`, `STRING`) als taktgesteuertes Latch mit integrierter Änderungserkennung verwendet werden.



Schnittstellenstruktur

Ereignis-Eingänge

- **CLK**: Taktereignis, übernimmt den aktuellen Wert von `D`.
  - **Verbundene Daten**: `D`

Ereignis-Ausgänge

- **EO**: Wird nur ausgelöst, wenn sich `Q` durch den `CLK`-Impuls tatsächlich geändert hat.
  - **Verbundene Daten**: `Q`

Daten-Eingänge

- **D** (ANY): Der zu speichernde Wert, beliebiger Datentyp.

Daten-Ausgänge

- **Q** (ANY): Der zuletzt übernommene Wert, vom selben Datentyp wie `D`.

Funktionsweise

Bei jedem CLK-Ereignis vergleicht die ECC den neuen Wert D mit dem aktuell gespeicherten Q (NE(Q, D)). Nur wenn sich der Wert tatsächlich geändert hat, wird Q := D ausgeführt und EO ausgelöst — bei unverändertem Wert bleibt der Baustein im Zustand SET, ohne ein Ereignis zu erzeugen. Der Anfangszustand START übernimmt beim ersten CLK immer den Wert, unabhängig von einer Änderungserkennung.

Technische Besonderheiten

- **ANY-Typisierung**: Im Gegensatz zu E_D_FF (fest BOOL) kann E_D_FF_ANY mit jedem IEC-61131-3-Datentyp instanziiert werden, den die Zielumgebung für generische ANY-Adaption unterstützt.
- **Eingebaute Änderungserkennung**: Anders als ein reines Latch löst E_D_FF_ANY EO nur bei tatsächlicher Wertänderung aus — dies reduziert die Ereignislast in nachgeschalteten FB-Netzwerken erheblich.
- **Basis für _TMIN-Varianten**: E_D_FF_ANY_TMIN kombiniert diesen Baustein zusätzlich mit einer Mindestabstandszeit zwischen zwei EO-Ereignissen.

Zustandsübersicht

Zustand	Bedeutung
START	Anfangszustand, erster CLK übernimmt D unbedingt
SET	Q enthält den zuletzt übernommenen Wert; weitere CLK-Ereignisse lösen nur bei NE(Q, D) erneut EO aus

Anwendungsszenarien

- **Änderungsgesteuerte Weiterverarbeitung beliebiger Datentypen**: Ein TIME- oder REAL-Wert soll nur dann ein Folgeereignis auslösen, wenn er sich tatsächlich geändert hat — z. B. als Baustein innerhalb von Composite-FBs wie ATM_SUB (siehe adapter-Bibliothek), die F_MOVE + E_D_FF_ANY zur expliziten, FBNetwork-sichtbaren Änderungserkennung kombinieren.
- **Generisches Caching**: Zwischenspeichern eines beliebigen Werts mit Benachrichtigung nur bei echter Änderung, unabhängig vom konkreten Datentyp.

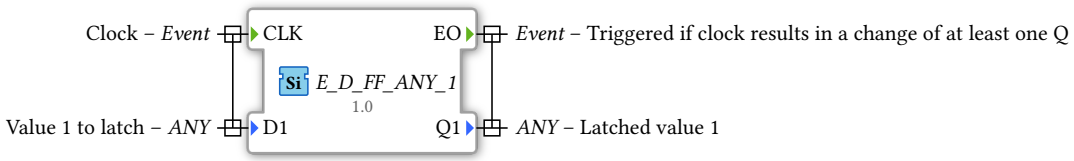
⚖️ Vergleich mit ähnlichen Bausteinen

- **E_D_FF**: funktional identisch, aber fest auf BOOL beschränkt.
- E_D_FF_ANY_TMIN: dieselbe Funktion, zusätzlich mit Mindestabstandszeit zwischen EO-Ereignissen.
- E_D_FF_TMIN: die BOOL-Variante mit Mindestabstandszeit.

Fazit

E_D_FF_ANY liefert ein generisch typisiertes, änderungserkennendes Latch und ist ein zentraler Baustein überall dort, wo Composite-FBs für beliebige Datentypen eine Ereignisauslösung nur bei tatsächlicher Wertänderung benötigen.

2.2.3.16 E_D_FF_ANY_1



• – ◦ • – ◦ • – ◦ •

Einleitung

Der E_D_FF_ANY_1 ist die Variante von E_D_FF_ANY für **einen** zwischengespeicherten Wert beliebigen Datentyps (ANY) und ist Teil der Familie E_D_FF_ANY_1 bis E_D_FF_ANY_5 mit durchnummerierten Ports (D1/Q1). Er ist ein taktgesteuertes Daten-Latch (D-Flipflop) mit integrierter Änderungserkennung: Das Ausgangsereignis EO wird nur ausgelöst, wenn sich Q1 durch den Takt ändert.

Schnittstellenstruktur

Ereignis-Eingänge

- **CLK**: Takt, übernimmt den aktuellen Wert von `D1`
  - **Verbundene Daten**: `D1`

Ereignis-Ausgänge

- **EO**: wird ausgelöst, wenn der Takt eine Änderung von `Q1` bewirkt
  - **Verbundene Daten**: `Q1`

Daten-Eingänge

- **D1** (ANY): zu speichernder Wert 1

Daten-Ausgänge

- **Q1** (ANY): zwischengespeicherter Wert 1

Adapter

Es werden keine Adapter verwendet.

Funktionsweise

Bei jedem `CLK`-Ereignis wird `D1` übernommen und mit dem bisher gespeicherten Wert `Q1` verglichen. Unterscheidet sich `D1` von `Q1`, wird der Wert gespeichert und `EO` wird ausgelöst; ist der Wert unverändert, entsteht kein Ausgangsereignis. Der Baustein ist ein generischer FB (`eclipse4diac::core::GenericClassName = GEN_E_D_FF_ANY`); die native Implementierung `GEN_E_D_FF_ANY` (FORTE) wird benötigt.

Technische Besonderheiten

- **Generischer Baustein**: Die Typen von `D1` und `Q1` sind `ANY` und werden bei der Instanziierung festgelegt
- Port-Namen `D1`/`Q1` (statt `D`/`Q` bei `E_D_FF_ANY`) entsprechen der durchnummerierten Familie
- Version 1.0 (2026-08-26): Erweiterung von `E_D_FF_ANY` auf eine nummerierte Variable

Anwendungsszenarien

- Adapter-Wrapper um Funktionen mit einem Ergebnis, mit Ausgabe nur bei Wertänderung (z. B. `CEIL_AR`)
- Änderungsgesteuerte Weiterverarbeitung eines Wertes beliebigen Datentyps

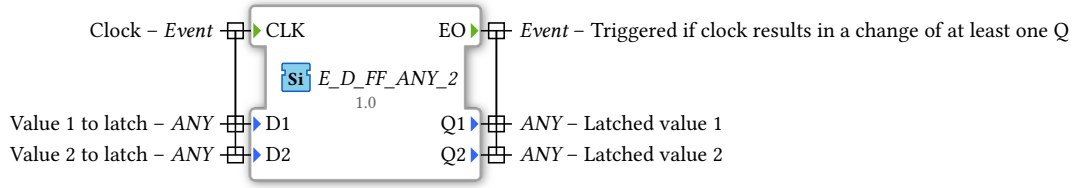
⚖️ Vergleich mit ähnlichen Bausteinen

`E_D_FF_ANY` speichert ebenfalls einen Wert (Ports `D`/`Q`), `E_D_FF_ANY_2` zwei, `E_D_FF_ANY_3` drei, `E_D_FF_ANY_4` vier und `E_D_FF_ANY_5` fünf Werte.

Fazit

`E_D_FF_ANY_1` ist ein generisches, änderungserkennendes Ein-Wert-Latch, das nur dann `EO` ausgibt, wenn sich der zwischengespeicherte Wert ändert.

2.2.3.17 E_D_FF_ANY_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `E_D_FF_ANY_2` ist ein Daten-Latch (D-Flipflop) für **zwei `ANY`-Variablen** (generischer Baustein). Der Ausgang `EO` wird nur ausgelöst, wenn sich mindestens ein `Qi` durch den Takt ändert. Er ist die Erweiterung von `E_D_FF_ANY` auf zwei Variablen.

Schnittstellenstruktur

Ereignis-Eingänge

- **CLK**: Takt; übernimmt `D1` und `D2` in die Latches. Verbundene Daten: `D1`, `D2`

Ereignis-Ausgänge

- **EO**: Wird ausgelöst, wenn der Takt zu einer Änderung von mindestens einem `Q` führt. Verbundene Daten: `Q1`, `Q2`

Daten-Eingänge

- **D1** (`ANY`): Zu speichernder Wert 1
- **D2** (`ANY`): Zu speichernder Wert 2

Daten-Ausgänge

- **Q1** (`ANY`): Gespeicherter Wert 1
- **Q2** (`ANY`): Gespeicherter Wert 2

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei jedem `CLK`-Ereignis werden `D1` und `D2` mit den gespeicherten Werten `Q1` und `Q2` verglichen. Weicht mindestens ein `Di` vom gespeicherten `Qi` ab, werden die Werte übernommen und `EO` ausgelöst. Sind beide Werte unverändert, wird kein `EO` ausgelöst.

Technische Besonderheiten

- Service-Interface-FB (generisch, `ANY`-Typen); er benötigt die native Implementierung `GEN_E_D_FF_ANY` (FORTE).
- Im Gegensatz zum einfachen D-Flipflop `E_D_FF` wird `EO` nur bei einer tatsächlichen Änderung ausgelöst.
- Die Variante für drei Variablen ist `E_D_FF_ANY_3`.

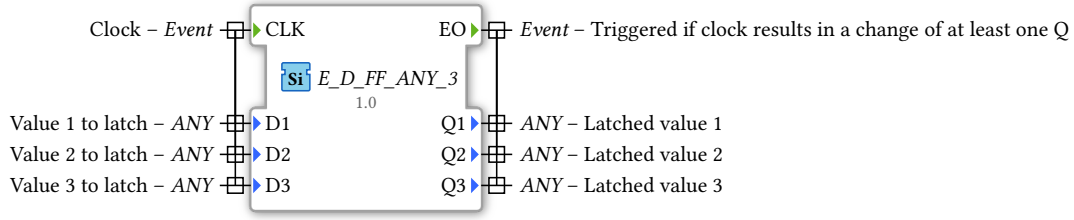
Anwendungsszenarien

- Gemeinsames Latchen zusammengehörender Werte (z. B. Stellwert und Regelabweichung) und Weitergabe nur bei Änderung, um unnötige Folgeereignisse zu vermeiden

Fazit

`E_D_FF_ANY_2` speichert zwei beliebige Werte und meldet nur dann ein Ereignis, wenn sich mindestens einer davon geändert hat.

2.2.3.18 E_D_FF_ANY_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `E_D_FF_ANY_3` ist die Variante von `E_D_FF_ANY` für **drei** gleichzeitig zwischengespeicherte Werte beliebigen Datentyps (`ANY`). Er ist ein taktgesteuertes Daten-Latch mit integrierter Änderungserkennung: Das Ausgangsereignis `EO` wird nur ausgelöst, wenn sich mindestens einer der drei Ausgänge `Q1`..`Q3` durch den Takt tatsächlich ändert.

Schnittstellenstruktur

Ereignis-Eingänge

- **CLK**: Takt, übernimmt die aktuellen Werte von `D1`, `D2` und `D3`
  - **Verbundene Daten**: `D1`, `D2`, `D3`

Ereignis-Ausgänge

- **EO**: wird ausgelöst, wenn der Takt eine Änderung mindestens eines `Qi` bewirkt
  - **Verbundene Daten**: `Q1`, `Q2`, `Q3`

Daten-Eingänge

- **D1** (ANY): zu speichernder Wert 1
- **D2** (ANY): zu speichernder Wert 2
- **D3** (ANY): zu speichernder Wert 3

Daten-Ausgänge

- **Q1** (ANY): zwischengespeicherter Wert 1
- **Q2** (ANY): zwischengespeicherter Wert 2
- **Q3** (ANY): zwischengespeicherter Wert 3

Adapter

Es werden keine Adapter verwendet.

Funktionsweise

Bei jedem `CLK`-Ereignis werden `D1`, `D2` und `D3` übernommen und mit den bisher gespeicherten Werten `Q1`, `Q2` und `Q3` verglichen. Unterscheidet sich mindestens ein `Di` vom zugehörigen `Qi`, werden die Werte gespeichert und `EO` wird ausgelöst; sind alle Werte unverändert, entsteht kein Ausgangsereignis. Der Baustein ist ein generischer Service-Interface-FB, dessen native Implementierung `GEN_E_D_FF_ANY` (FORTE) ist.

Technische Besonderheiten

- **Generischer Baustein**: Die Typen von `D1`..`D3` und `Q1`..`Q3` sind `ANY` und werden bei der Instanziierung festgelegt
- **Gemeinsame Änderungserkennung**: Ein einziges `EO` meldet die Änderung irgendeines der drei Werte; welcher sich geändert hat, ist am Ereignis nicht erkennbar
- Dient dazu, mehrere zusammengehörige Werte (z. B. Stellwert, Regelabweichung und Begrenzungs-Flag eines Reglers) mit nur einem Ereignis weiterzugeben und Folgeereignisse bei unveränderten Werten zu vermeiden

Anwendungsszenarien

- Adapter-Wrapper um Funktionen, die drei Ergebnisse liefern, mit Ausgabe nur bei Wertänderung (z. B. `CTRL_PI_AR`)
- Änderungsgesteuerte Weiterverarbeitung mehrerer Werte beliebigen Datentyps

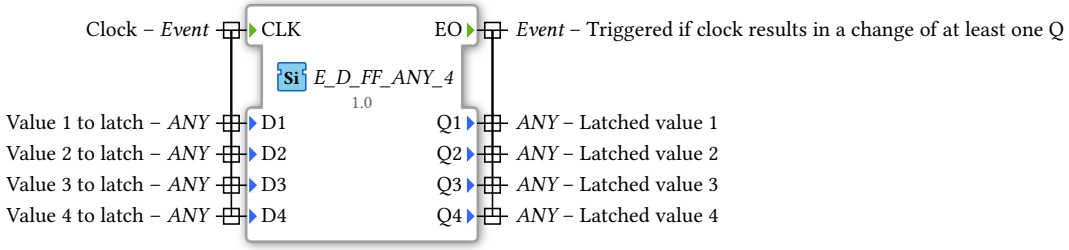
⚖️ Vergleich mit ähnlichen Bausteinen

`E_D_FF_ANY` speichert einen Wert, `E_D_FF_ANY_3` drei. Entsprechende Varianten mit anderer Anzahl gibt es als `E_D_FF_ANY_1`, `E_D_FF_ANY_2`, `E_D_FF_ANY_4` und `E_D_FF_ANY_5`.

Fazit

`E_D_FF_ANY_3` ist ein generisches, änderungserkennendes Drei-Wert-Latch, das nur dann `EO` ausgibt, wenn sich mindestens einer der drei zwischengespeicherten Werte ändert.

2.2.3.19 E_D_FF_ANY_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `E_D_FF_ANY_4` ist die Variante von `E_D_FF_ANY` für **vier** gleichzeitig zwischengespeicherte Werte beliebigen Datentyps (`ANY`). Er ist ein taktgesteuertes Daten-Latch (D-Flipflop) mit integrierter Änderungserkennung: Das Ausgangsereignis `EO` wird nur ausgelöst, wenn sich mindestens einer der vier Ausgänge `Q1`..`Q4` durch den Takt ändert.

Schnittstellenstruktur

Ereignis-Eingänge

- **CLK**: Takt, übernimmt die aktuellen Werte von `D1`..`D4`
  - **Verbundene Daten**: `D1`, `D2`, `D3`, `D4`

Ereignis-Ausgänge

- **EO**: wird ausgelöst, wenn der Takt eine Änderung mindestens eines `Qi` bewirkt
  - **Verbundene Daten**: `Q1`, `Q2`, `Q3`, `Q4`

Daten-Eingänge

- **D1** (ANY): zu speichernder Wert 1
- **D2** (ANY): zu speichernder Wert 2
- **D3** (ANY): zu speichernder Wert 3
- **D4** (ANY): zu speichernder Wert 4

Daten-Ausgänge

- **Q1** (ANY): zwischengespeicherter Wert 1
- **Q2** (ANY): zwischengespeicherter Wert 2
- **Q3** (ANY): zwischengespeicherter Wert 3
- **Q4** (ANY): zwischengespeicherter Wert 4

Adapter

Es werden keine Adapter verwendet.

Funktionsweise

Bei jedem `CLK`-Ereignis werden `D1`..`D4` übernommen und mit den bisher gespeicherten Werten `Q1`..`Q4` verglichen. Unterscheidet sich mindestens ein `Di` vom zugehörigen `Qi`, werden die Werte gespeichert und `EO` wird ausgelöst; sind alle Werte unverändert, entsteht kein Ausgangsereignis. Der Baustein ist ein generischer FB (`eclipse4diac::core::GenericClassName = GEN_E_D_FF_ANY`).

Technische Besonderheiten

- **Generischer Baustein**: Die Typen von `D1`..`D4` und `Q1`..`Q4` sind `ANY` und werden bei der Instanziierung festgelegt
- **Gemeinsame Änderungserkennung**: Ein einziges `EO` meldet die Änderung irgendeines der vier Werte; welcher sich geändert hat, ist am Ereignis nicht erkennbar

Anwendungsszenarien

- Adapter-Wrapper um Funktionen, die vier Ergebnisse liefern, mit Ausgabe nur bei Wertänderung
- Änderungsgesteuerte Weiterverarbeitung mehrerer Werte beliebigen Datentyps

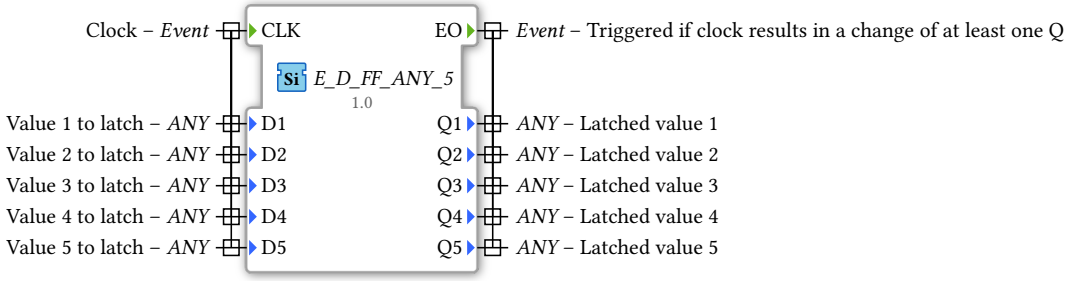
⚖️ Vergleich mit ähnlichen Bausteinen

E_D_FF_ANY speichert einen Wert, E_D_FF_ANY_2 zwei, E_D_FF_ANY_3 drei und E_D_FF_ANY_4 vier Werte.

Fazit

E_D_FF_ANY_4 ist ein generisches, änderungserkennendes Vier-Wert-Latch, das nur dann EO ausgibt, wenn sich mindestens einer der vier zwischengespeicherten Werte ändert.

2.2.3.20 E_D_FF_ANY_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der E_D_FF_ANY_5 ist die Variante von E_D_FF_ANY für **fünf** gleichzeitig zwischengespeicherte Werte beliebigen Datentyps (ANY). Er ist ein taktgesteuertes Daten-Latch (D-Flipflop) mit integrierter Änderungserkennung: Das Ausgangsereignis EO wird nur ausgelöst, wenn sich mindestens einer der fünf Ausgänge Q1..Q5 durch den Takt ändert.

Schnittstellenstruktur

Ereignis-Eingänge

- **CLK**: Takt, übernimmt die aktuellen Werte von D1..D5
  - **Verbundene Daten**: D1, D2, D3, D4, D5

Ereignis-Ausgänge

- **EO**: wird ausgelöst, wenn der Takt eine Änderung mindestens eines Qi bewirkt
  - **Verbundene Daten**: Q1, Q2, Q3, Q4, Q5

Daten-Eingänge

- **D1** (ANY): zu speichernder Wert 1
- **D2** (ANY): zu speichernder Wert 2
- **D3** (ANY): zu speichernder Wert 3
- **D4** (ANY): zu speichernder Wert 4
- **D5** (ANY): zu speichernder Wert 5

Daten-Ausgänge

- **Q1** (ANY): zwischengespeicherter Wert 1
- **Q2** (ANY): zwischengespeicherter Wert 2
- **Q3** (ANY): zwischengespeicherter Wert 3
- **Q4** (ANY): zwischengespeicherter Wert 4
- **Q5** (ANY): zwischengespeicherter Wert 5

Adapter

Es werden keine Adapter verwendet.

Funktionsweise

Bei jedem CLK-Ereignis werden D1..D5 übernommen und mit den bisher gespeicherten Werten Q1..Q5 verglichen. Unterscheidet sich mindestens ein Di vom zugehörigen Qi, werden die Werte gespeichert und EO wird ausgelöst; sind alle Werte unverändert, entsteht kein Ausgangsereignis. Der Baustein ist ein

generischer FB (`eclipse4diac::core::GenericClassName = GEN_E_D_FF_ANY`); die native Implementierung `GEN_E_D_FF_ANY` (FORTE) wird benötigt.

Technische Besonderheiten

- **Generischer Baustein**: Die Typen von `D1`..`D5` und `Q1`..`Q5` sind `ANY` und werden bei der Instanziierung festgelegt
- **Gemeinsame Änderungserkennung**: Ein einziges `EO` meldet die Änderung irgendeines der fünf Werte; welcher sich geändert hat, ist am Ereignis nicht erkennbar
- Version 1.0 (2026-08-26): Erweiterung von `E_D_FF_ANY` auf 5 Variablen

Anwendungsszenarien

- Adapter-Wrapper um Funktionen, die fünf Ergebnisse liefern, mit Ausgabe nur bei Wertänderung
- Änderungsgesteuerte Weiterverarbeitung mehrerer Werte beliebigen Datentyps

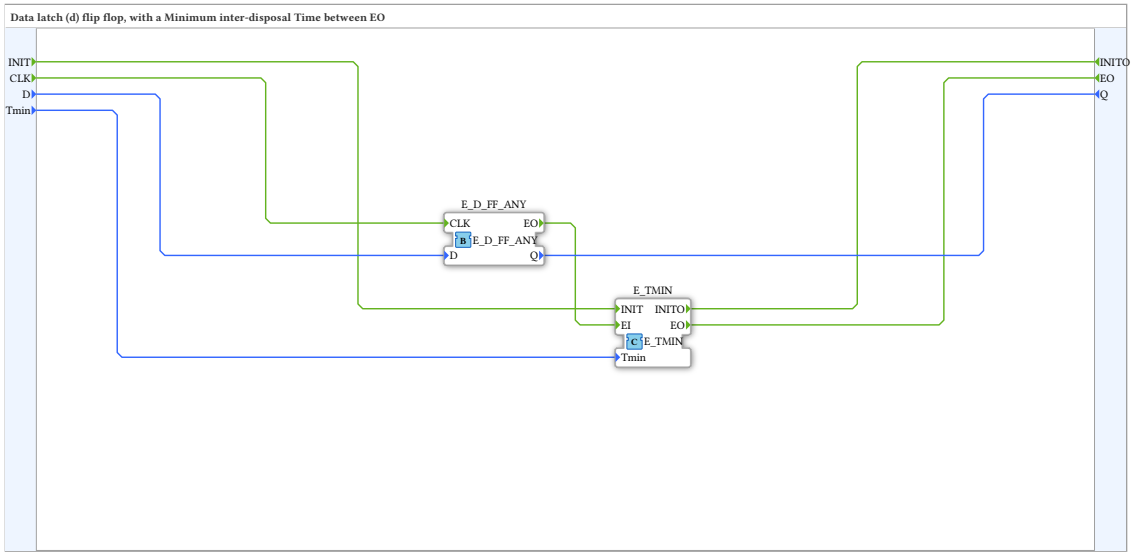
⚖️ Vergleich mit ähnlichen Bausteinen

`E_D_FF_ANY` speichert einen Wert, `E_D_FF_ANY_2` zwei, `E_D_FF_ANY_3` drei, `E_D_FF_ANY_4` vier und `E_D_FF_ANY_5` fünf Werte.

Fazit

`E_D_FF_ANY_5` ist ein generisches, änderungserkennendes Fünf-Wert-Latch, das nur dann `EO` ausgibt, wenn sich mindestens einer der fünf zwischengespeicherten Werte ändert.

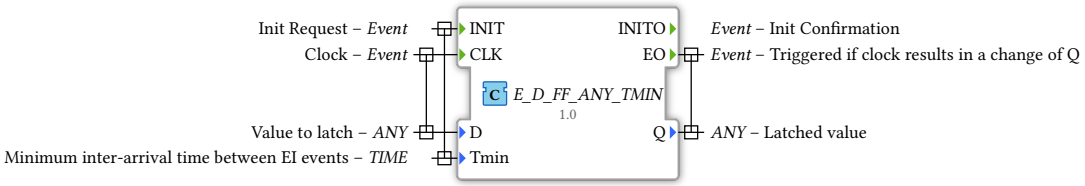
2.2.3.21 E_D_FF_ANY_TMIN



• – ○ • – ○ • – ○ •

Einleitung

Der `E_D_FF_ANY_TMIN` ist die generisch typisierte Variante von E_D_FF_TMIN: Er kombiniert E_D_FF_ANY (änderungserkennendes Latch für beliebige Datentypen) mit E_TMIN (Mindestabstandszeit) und eignet sich damit für alle Datentypen, nicht nur `BOOL`.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierung, trägt `Tmin`.
- **CLK**: Taktereignis, übernimmt `D` in `Q`.
  - **Verbundene Daten**: `D`

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung.
- **EO**: Auf `Tmin` gedrosseltes Änderungsereignis.
  - **Verbundene Daten**: `Q`

Daten-Eingänge

- **D** (ANY): Der zu speichernde Wert, beliebiger Datentyp.
- **Tmin** (TIME): Mindestabstand zwischen zwei `EO`-Ereignissen.

Daten-Ausgänge

- **Q** (ANY): Der zuletzt übernommene Wert.

Funktionsweise

Identisch zu E_D_FF_TMIN, jedoch mit E_D_FF_ANY statt `E_D_FF`: `CLK` triggert die interne Änderungserkennung (`NE(Q, D)`), deren `EO` an das interne E_TMIN weitergeleitet wird, das die zeitliche Drosselung auf mindestens `Tmin` zwischen zwei Auslösungen von `EO` übernimmt.

Technische Besonderheiten

- **ANY-Typisierung**: kann mit jedem von der Zielumgebung unterstützten Datentyp instanziiert werden (z. B. `TIME`, `REAL`, `DINT`).
- **Zweistufige Filterung**: inhaltliche Änderungserkennung (`E_D_FF_ANY`) plus zeitliche Drosselung (`E_TMIN`), `Q` bleibt davon unbeeinflusst und stets aktuell.

Zustandsübersicht

Zustandslose Verdrahtungslogik, siehe E_D_FF_ANY (Änderungserkennung) und E_TMIN (Zeitdrosselung).

Anwendungsszenarien

- **Rauscharme Weiterleitung beliebiger Messwerte**: Ein `REAL`- oder `TIME`-Wert schwankt geringfügig, `EO` soll aber höchstens alle `Tmin` ausgelöst werden — z. B. zur Begrenzung der Sendefrequenz an ein übergeordnetes System.

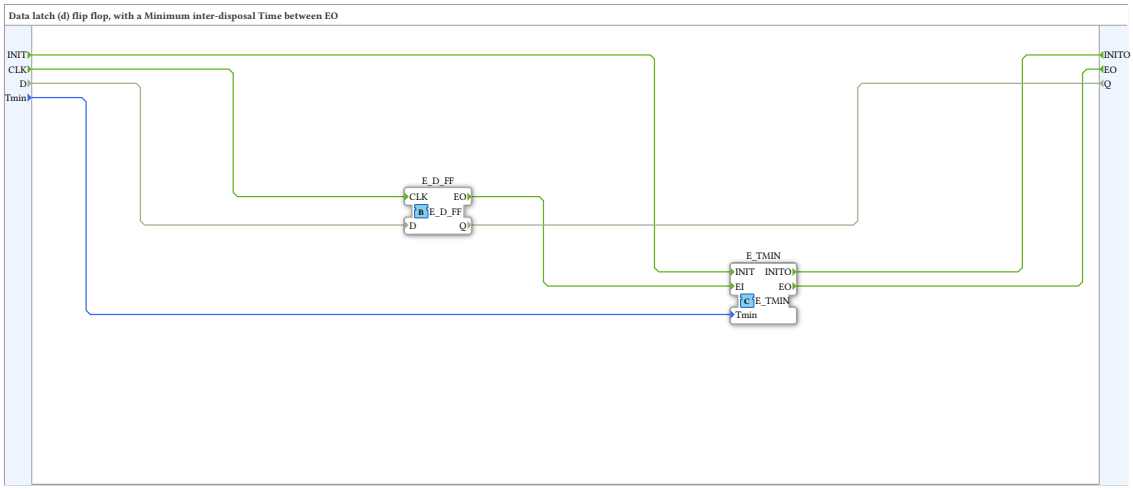
⚖️ Vergleich mit ähnlichen Bausteinen

- **E_D_FF_ANY**: dieselbe Änderungserkennung, aber ohne Zeitdrosselung.
- **E_D_FF_TMIN**: dieselbe Funktion, aber fest auf `BOOL` beschränkt.
- **E_TMIN**: die intern verwendete Drosselungslogik.

Fazit

`E_D_FF_ANY_TMIN` liefert ein generisch typisiertes, änderungserkennendes Latch mit garantiertem Mindestabstand zwischen Änderungsereignissen für beliebige Datentypen.

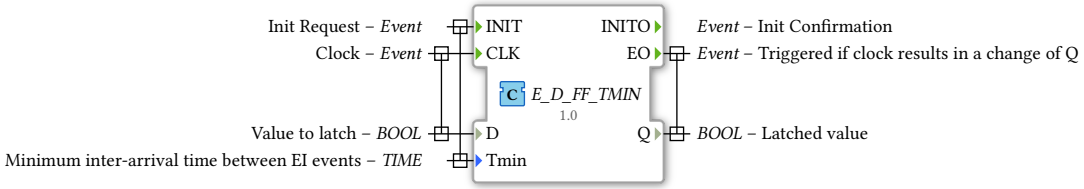
2.2.3.22 E_D_FF_TMIN



• – ○ • – ○ • – ○ •

Einleitung

Der `E_D_FF_TMIN` kombiniert `E_D_FF` (taktgesteuertes `BOOL`-Latch mit Änderungserkennung) mit E_TMIN (Mindestabstandszeit): Änderungen von `Q` werden weiterhin sofort in `Q` übernommen, das Änderungsereignis `EO` wird jedoch zusätzlich auf höchstens eine Auslösung pro `Tmin`-Intervall begrenzt.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierung, trägt `Tmin`.
- **CLK**: Taktereignis, übernimmt `D` in `Q`.
  - **Verbundene Daten**: `D`

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung.
- **EO**: Auf `Tmin` gedrosseltes Änderungsereignis.
  - **Verbundene Daten**: `Q`

Daten-Eingänge

- **D** (BOOL): Der zu speichernde Wert.
- **Tmin** (TIME): Mindestabstand zwischen zwei `EO`-Ereignissen.

Daten-Ausgänge

- **Q** (BOOL): Der zuletzt übernommene Wert.

Funktionsweise

`CLK` triggert intern das gekapselte `E_D_FF`, dessen `D`/`Q`-Datenverbindung direkt auf die äußeren Anschlüsse `D`/`Q` durchgereicht wird. Nur wenn `E_D_FF` eine echte Wertänderung erkennt (`Q := D`), löst es sein eigenes `EO` aus — dieses wird an den `EI`-Eingang des internen E_TMIN weitergeleitet, welches das Ereignis nur dann an den äußeren `EO`-Ausgang durchlässt, wenn seit dem letzten Durchlass mindestens `Tmin` vergangen ist.

Technische Besonderheiten

- **Zweistufige Filterung**: Erst inhaltliche Änderungserkennung (`E_D_FF`), dann zeitliche Drosselung (`E_TMIN`) — `Q` selbst wird davon nicht beeinflusst und ist stets aktuell, auch wenn `EO` gerade gedrosselt wird.
- **Reine Verdrahtung aus zwei Standardbausteinen**, kein eigenes ECC.

Zustandsübersicht

Zustandslose Verdrahtungslogik, siehe `E_D_FF` (Änderungserkennung) und E_TMIN (Zeitdrosselung) für die jeweiligen internen Zustände.

Anwendungsszenarien

- **Rauscharme Statusmeldung eines `BOOL`-Signals**: Ein digitaler Zustand ändert sich häufig kurzfristig (Prellen, Rauschen); `EO` soll aber höchstens alle `Tmin` ausgelöst werden, während `Q` jederzeit den aktuellen Wert zeigt.

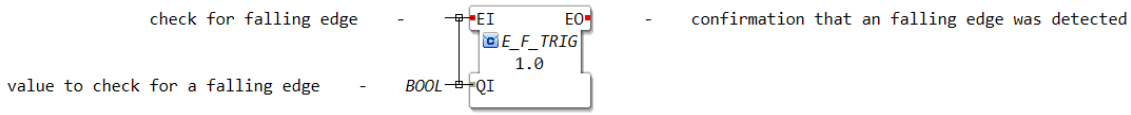
⚖️ Vergleich mit ähnlichen Bausteinen

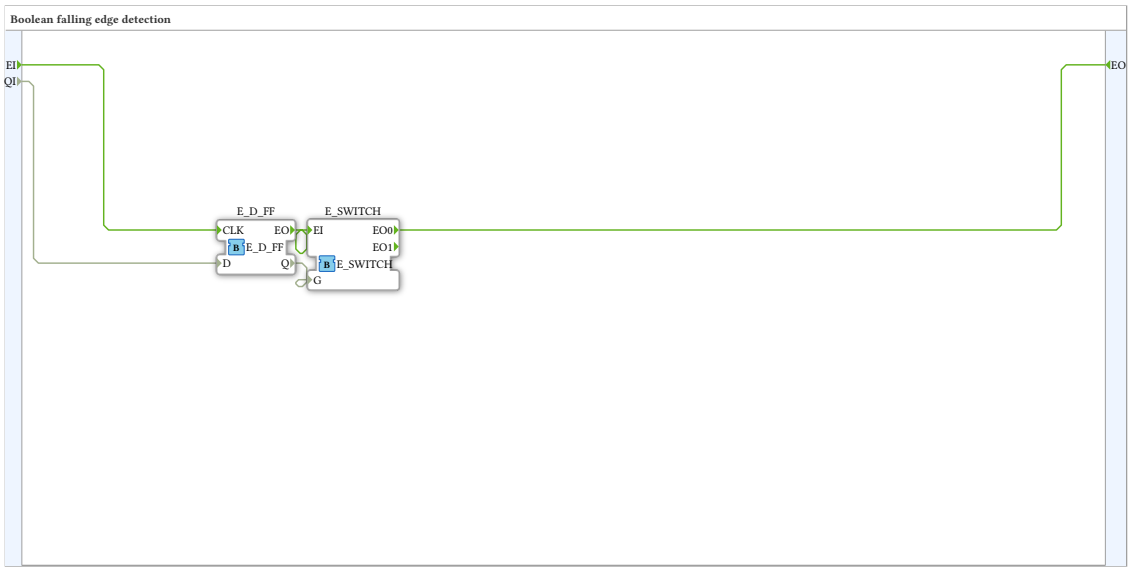
- **`E_D_FF`**: dieselbe Änderungserkennung, aber ohne Zeitdrosselung.
- **E_D_FF_ANY_TMIN**: dieselbe Funktion für beliebige Datentypen statt nur `BOOL`.
- **E_TMIN**: die intern verwendete Drosselungslogik.

Fazit

`E_D_FF_TMIN` liefert ein änderungserkennendes `BOOL`-Latch mit garantiertem Mindestabstand zwischen den Änderungsereignissen und eignet sich für rauscharme Ereignisweiterleitung digitaler Signale.

2.2.3.23 E_F_TRIG





• – ○ • – ○ • – ○ •

Einleitung

Der **E_F_TRIG** (Falling Edge Trigger) ist ein grundlegender Funktionsbaustein der IEC 61499 Norm zur Erkennung von fallenden Signalflanken. Dieser Baustein spielt eine wichtige Rolle in der Ereignisdetektion und Signalverarbeitung industrieller Steuerungssysteme.

Struktur des E_F_TRIG-Bausteins

Schnittstelle (Interface)

Ereignis-Eingänge:

- **EI (Event Input)**: Löst die Prüfung auf eine Flanke aus.
  - **Verbundene Daten**: `QI`

Ereignis-Ausgänge:

- **EO (Event Output)**: Wird ausgelöst, wenn eine Flanke erkannt wurde.

Daten-Eingänge:

- **QI (Qualifier Input)**: Das zu überwachende boolesche Signal (Datentyp: `BOOL`).

Funktionsweise

Der `E_F_TRIG` soll eine fallende Flanke am `QI`-Eingang erkennen. Eine fallende Flanke tritt auf, wenn `QI` beim letzten `EI`-Ereignis `TRUE` war und beim aktuellen `EI`-Ereignis `FALSE` ist.

Bei jedem `EI`-Ereignis wird der aktuelle Zustand von `QI` mit dem Zustand aus dem vorherigen Zyklus verglichen. Wenn eine fallende Flanke erkannt wird, sollte das `EO`-Ereignis ausgelöst werden.

Achtung: Fehlerhafte Implementierung Die interne Logik des Bausteins (Stand: `events-3.0.0`) ist fehlerhaft und implementiert tatsächlich einen **steigenden Flankendetektor (`E_R_TRIG`)** anstatt eines fallenden. Das `EO`-Ereignis wird ausgelöst, wenn `QI` von `FALSE` auf `TRUE` wechselt.

Technische Besonderheiten

✓ **Ereignisgesteuert**: Die Prüfung erfolgt nur, wenn ein `EI`-Ereignis eintritt. ✓ **Fehlerhafte Logik**: Der Baustein ist als `E_F_TRIG` benannt, seine interne Verschaltung eines `E_D_FF` und `E_SWITCH` bewirkt jedoch, dass er auf eine **steigende Flanke** reagiert. Nutzer, die eine fallende Flankenerkennung benötigen, müssen stattdessen den `E_R_TRIG`-Baustein verwenden und dessen Eingang negieren oder auf eine Korrektur des Bausteins warten.

Anwendungsszenarien (für eine fallende Flanke)

- **Sensordatenauswertung**: Erkennen, wenn ein Sensor von „aktiv“ auf „inaktiv“ wechselt.
- **Maschinensicherheit**: Detektion, wenn ein Schutzkontakt von „geschlossen“ (TRUE) auf „offen“ (FALSE) geht.
- **Prozessüberwachung**: Erkennen des Endes eines Signals oder Prozesses.

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	E_F_TRIG (dieser)	E_R_TRIG	E_SWITCH
Erkannte Flanke	Fallend (benannt), Steigend (implementiert)	Steigend	-
Ereignissteuerung	Ja	Ja	Ja
Prinzip	Flankendetektor	Flankendetektor	Ereignisweiche

🛠️ Zugehörige Übungen

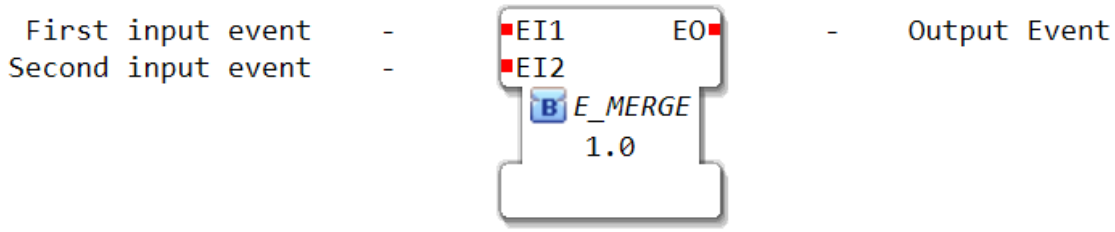
- Uebung_088

Fazit

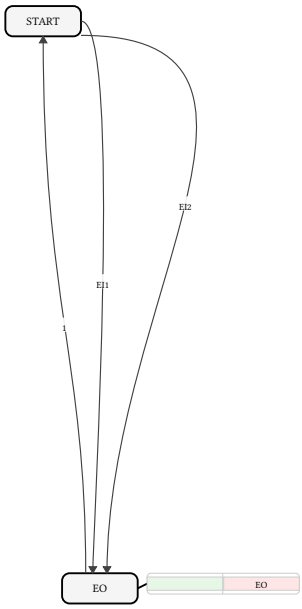
Der E_F_TRIG-Baustein ist dazu gedacht, fallende Flanken zu erkennen, was ein häufiger Anwendungsfall in der Steuerungstechnik ist. **Aufgrund eines Implementierungsfehlers in der `events-3.0.0`-Bibliothek funktioniert dieser Baustein jedoch als Detektor für steigende Flanken.** Diese Diskrepanz

zwischen Name und Funktion ist kritisch und muss bei der Verwendung unbedingt beachtet werden. Für eine zuverlässige Erkennung fallender Flanken sollte auf alternative Logiken zurückgegriffen werden.

2.2.3.24 E_MERGE



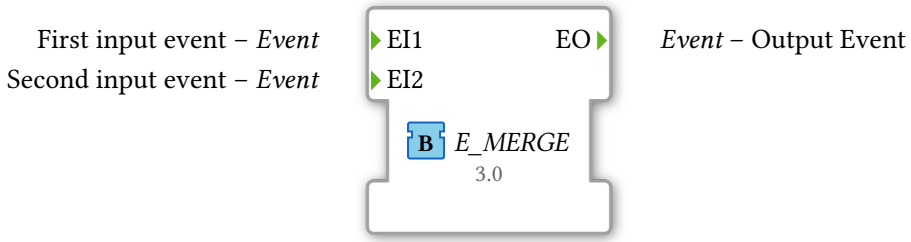
ECC: E_MERGE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **E_MERGE** ist ein grundlegender Funktionsbaustein der IEC 61499 Norm, der mehrere Ereignisströme zu einem einzigen Ausgang zusammenführt. Diese logische ODER-Verknüpfung von Ereignissen ist essentiell für die Steuerungslogik in industriellen Automatisierungssystemen.



Struktur des E_MERGE-Bausteins

Schnittstelle (Interface)
Ereignis-Eingänge:

- `EI1` (Event Input 1): Erster Ereigniseingang
- `EI2` (Event Input 2): Zweiter Ereigniseingang

Ereignis-Ausgänge:

- `EO` (Event Output): Zusammengeführter Ereignisausgang

Funktionsweise

1. **Ereigniszusammenführung:**
   - Jedes Ereignis an `EI1` oder `EI2` löst ein Ausgangsereignis an `EO` aus
   - Die Eingänge werden logisch ODER-verknüpft
2. **Unabhängige Verarbeitung:**
   - Ereignisse an beiden Eingängen werden gleichberechtigt behandelt
   - Keine Priorisierung bestimmter Eingänge
3. **Sofortige Weiterleitung:**
   - Keine Verzögerung zwischen Eingangs- und Ausgangsereignis
   - Kein Speicherverhalten oder Zustandshaltung

Technische Besonderheiten

✓ **Einfache und schnelle** Ereignisverknüpfung ✓ **Verlustfreie** Ereignisweitergabe ✓ **Echtzeitfähig** für industrielle Anwendungen ✓ **Erweiterbar** auf mehrere Eingänge

Anwendungsszenarien

- **Bedienkonzepte**: Zusammenführung von Steuersignalen mehrerer Taster
- **Sensordaten**: Kombination von Ereignissen verschiedener Sensoren
- **Fehlermanagement**: Sammelstelle für verschiedene Störmeldungen
- **Prozesssteuerung**: Verknüpfung von Prozessereignissen

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	E_MERGE	E_DEMUX	E_SWITCH
Funktionsprinzip	ODER-Verknüpfung	Verteilung	Bedingte Weiterleitung
Richtung	n:1	1:n	1:1
Ereignisfluss	Kombination	Aufteilung	Selektion

Ähnliche Bausteine

Für Anwendungsfälle, die mehr als zwei Ereigniseingänge benötigen, stellt die Bibliothek weitere Varianten zur Verfügung:

- **E_MERGE**: Dieser Baustein (2 Eingänge)
- **E_MERGE_2**: Funktional identisch zu `E_MERGE` (2 Eingänge)
- **E_MERGE_3**: Eine Variante mit 3 Eingängen (`EI1`, `EI2`, `EI3`)
- **E_MERGE_4**: Eine Variante mit 4 Eingängen

Diese Bausteine erlauben die einfache Zusammenführung von bis zu vier Ereignisquellen auf einen einzigen Ausgang.

🛠️ Zugehörige Übungen

- Uebung_004a2
- Uebung_004a2_AX

Fazit

Der E_MERGE-Baustein ist ein fundamentaler Baustein für die Ereignisverarbeitung in IEC 61499-Systemen. Seine Hauptvorteile sind:

- Einfache und effiziente Ereigniskombination
- Sofortige Reaktion auf Eingangsereignisse

- Flexible Einsatzmöglichkeiten in verschiedenen Steuerungsszenarien

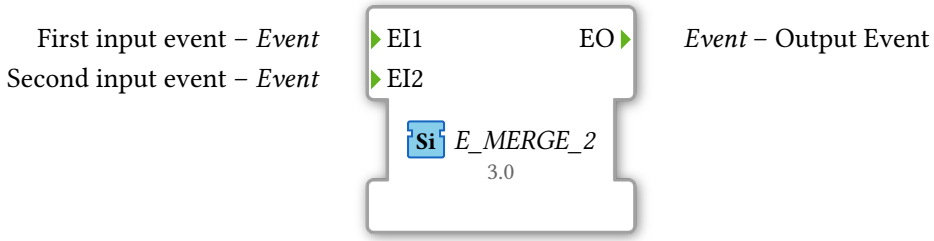
Durch seine klare Funktionsweise und Standardkonformität eignet er sich ideal für grundlegende Verknüpfungsaufgaben in industriellen Automatisierungslösungen. Die deterministische Arbeitsweise macht ihn besonders wertvoll für sicherheitskritische Anwendungen.

2.2.3.25 E_MERGE_2

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `E_MERGE_2` ist ein grundlegender Funktionsbaustein der IEC 61499 Norm, der zwei Ereignisströme (`EI1`, `EI2`) zu einem einzigen Ausgang (`EO`) zusammenführt. Diese logische ODER-Verknüpfung von Ereignissen ist essentiell für die Steuerungslogik in industriellen Automatisierungssystemen. Dieser Baustein ist funktional identisch mit dem `E_MERGE`-Baustein.



Schnittstellenstruktur

Ereignis-Eingänge:

- `EI1` (Event Input 1): Erster Ereigniseingang
- `EI2` (Event Input 2): Zweiter Ereigniseingang

Ereignis-Ausgänge:

- `EO` (Event Output): Zusammengeführter Ereignisausgang

Funktionsweise

Die Funktion ist einfach und direkt:

- Ein Ereignis am Eingang `EI1` löst sofort ein Ereignis am Ausgang `EO` aus.
- Ein Ereignis am Eingang `EI2` löst ebenfalls sofort ein Ereignis am Ausgang `EO` aus.

Die Eingänge werden logisch ODER-verknüpft und ohne Priorisierung oder Verzögerung weitergeleitet.

Technische Besonderheiten

- **2-zu-1 Verknüpfung**: Führt zwei Ereignisströme zu einem zusammen.
- **Zustandslos**: Der Baustein hat keinen internen Speicher.
- **Generischer Baustein**: Die Funktionalität wird durch die generische Klasse `GEN_E_MERGE` zur Verfügung gestellt, was eine optimierte Ausführung durch die 4diac-Laufzeitumgebung ermöglicht.

Anwendungsszenarien

- **Bedienkonzepte**: Zusammenführung von Steuersignalen von zwei Tastern auf eine gemeinsame Funktion.
- **Sensorik**: Kombination der Trigger-Ereignisse von zwei verschiedenen Sensoren.
- **Fehlermanagement**: Erstellen einer Sammel-Störmeldung aus zwei potenziellen Fehlerquellen.

🛠️ Zugehörige Übungen

- Uebung_004a2_2

Fazit

Der `E_MERGE_2` ist ein fundamentaler und häufig verwendeter Baustein für die einfachste Form der Ereignis-Verknüpfung. Seine klare Funktionsweise und die effiziente Implementierung als generischer Baustein machen ihn zu einer zuverlässigen Komponente für grundlegende Steuerungslogik.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

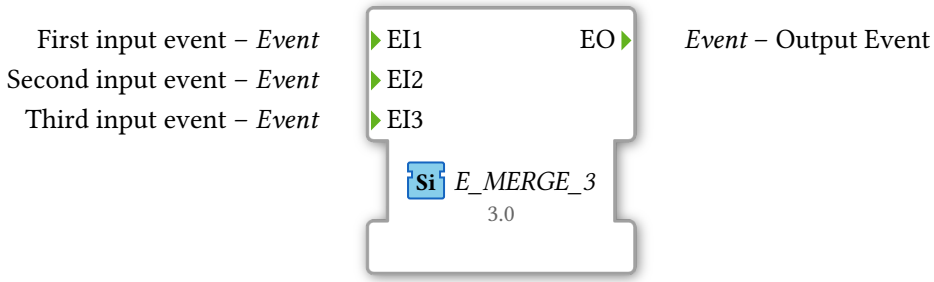
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.2.3.26 E_MERGE_3

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `E_MERGE_3` ist ein grundlegender Funktionsbaustein der IEC 61499 Norm, der drei Ereignisströme (`EI1`, `EI2`, `EI3`) zu einem einzigen Ausgang (`EO`) zusammenführt. Diese logische ODER-Verknüpfung von Ereignissen ist essentiell für die Steuerungslogik in industriellen Automatisierungssystemen.



Schnittstellenstruktur

Ereignis-Eingänge:

- `EI1` (Event Input 1): Erster Ereigniseingang
- `EI2` (Event Input 2): Zweiter Ereigniseingang
- `EI3` (Event Input 3): Dritter Ereigniseingang

Ereignis-Ausgänge:

- `EO` (Event Output): Zusammengeführter Ereignisausgang

Funktionsweise

Die Funktion ist einfach und direkt:

- Ein Ereignis am Eingang `EI1`, `EI2` oder `EI3` löst sofort und ohne Priorisierung ein Ereignis am Ausgang `EO` aus.

Die Eingänge werden logisch ODER-verknüpft und ohne Verzögerung weitergeleitet.

Technische Besonderheiten

- **3-zu-1 Verknüpfung**: Führt drei Ereignisströme zu einem zusammen.
- **Zustandslos**: Der Baustein hat keinen internen Speicher.
- **Generischer Baustein**: Die Funktionalität wird durch die generische Klasse `GEN_E_MERGE` zur Verfügung gestellt, was eine optimierte Ausführung durch die 4diac-Laufzeitumgebung ermöglicht.

Anwendungsszenarien

- **Bedienkonzepte**: Zusammenführung von Steuersignalen von drei Tastern auf eine gemeinsame Funktion (z.B. „Hand“, „Automatik“, „Service“ starten denselben Prozess).
- **Sensorik**: Kombination der Trigger-Ereignisse von drei verschiedenen Sensoren zu einem Gesamtereignis.
- **Fehlermanagement**: Erstellen einer Sammel-Störmeldung aus drei potenziellen Fehlerquellen.

🛠️ Zugehörige Übungen

- Uebung_004a2_3

Fazit

Der `E_MERGE_3` ist ein einfacher, aber nützlicher Baustein zur Verknüpfung von bis zu drei Ereignisquellen. Seine klare Funktionsweise und die effiziente Implementierung als generischer Baustein machen ihn zu einer zuverlässigen Komponente für grundlegende Steuerungslogik.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

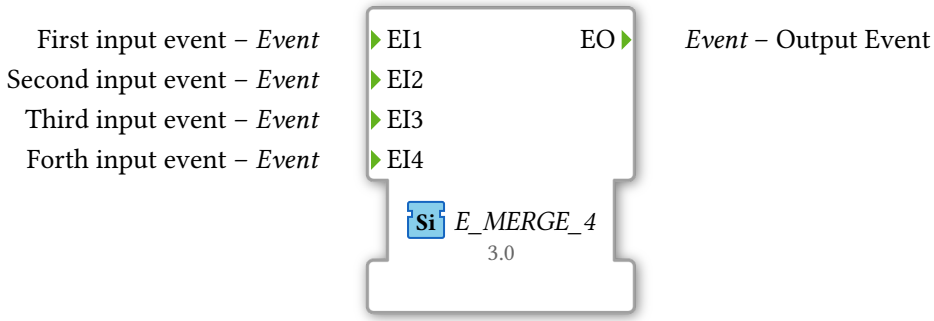
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.2.3.27 E_MERGE_4

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `E_MERGE_4` ist ein grundlegender Funktionsbaustein der IEC 61499 Norm, der vier Ereignisströme (`EI1` bis `EI4`) zu einem einzigen Ausgang (`EO`) zusammenführt. Diese logische ODER-Verknüpfung von Ereignissen ist essentiell für die Steuerungslogik in industriellen Automatisierungssystemen.



Schnittstellenstruktur

Ereignis-Eingänge:

- `EI1` (Event Input 1): Erster Ereigniseingang
- `EI2` (Event Input 2): Zweiter Ereigniseingang
- `EI3` (Event Input 3): Dritter Ereigniseingang
- `EI4` (Event Input 4): Vierter Ereigniseingang

Ereignis-Ausgänge:

- `EO` (Event Output): Zusammengeführter Ereignisausgang

Funktionsweise

Die Funktion ist einfach und direkt:

- Ein Ereignis an einem der vier Eingänge (`EI1` bis `EI4`) löst sofort und ohne Priorisierung ein Ereignis am Ausgang `EO` aus.

Die Eingänge werden logisch ODER-verknüpft und ohne Verzögerung weitergeleitet.

Technische Besonderheiten

- **4-zu-1 Verknüpfung**: Führt vier Ereignisströme zu einem zusammen.
- **Zustandslos**: Der Baustein hat keinen internen Speicher.
- **Generischer Baustein**: Die Funktionalität wird durch die generische Klasse `GEN_E_MERGE` zur Verfügung gestellt, was eine optimierte Ausführung durch die 4diac-Laufzeitumgebung ermöglicht.

Anwendungsszenarien

- **Sammelstörmeldung**: Zusammenführen von vier unterschiedlichen Fehlerereignissen zu einer einzigen Alarmmeldung.
- **Bedienung**: Ein Prozessschritt kann von vier verschiedenen Tastern oder Quellen ausgelöst werden.
- **Sensor-Array**: Ein einzelnes „Objekt erkannt“-Ereignis wird ausgelöst, wenn einer von vier Sensoren ein Objekt detektiert.

Fazit

Der `E_MERGE_4` ist ein einfacher, aber nützlicher Baustein zur Verknüpfung von bis zu vier Ereignisquellen. Seine klare Funktionsweise und die effiziente Implementierung als generischer Baustein machen ihn zu einer zuverlässigen Komponente für grundlegende Steuerungslogik.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

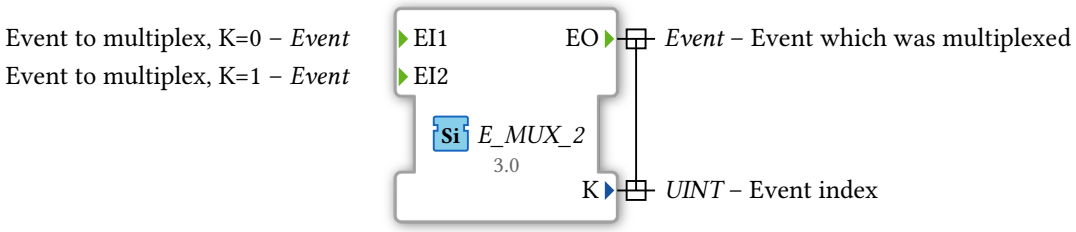
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.2.3.28 E_MUX_2

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `E_MUX_2` (Event Multiplexer) ist ein Funktionsbaustein nach IEC 61499, der zwei Ereigniseingänge (`EI1`, `EI2`) auf einen einzigen Ereignisausgang (`EO`) zusammenführt. Zusätzlich zur Zusammenführung gibt der Baustein an einem Datenausgang `K` aus, welcher der Eingänge das Ereignis ausgelöst hat. Er ist das Gegenstück zum `E_DEMUX_2`.



Schnittstellenstruktur

Ereignis-Eingänge:

- **EI1**: Erster Eingangskanal.
- **EI2**: Zweiter Eingangskanal.

Ereignis-Ausgänge:

- **EO (Event Output)**: Der gemeinsame Ereignisausgang.
  - **Verbundene Daten**: `K`

Daten-Ausgänge:

- **K**: Der Index des auslösenden Eingangskanals (Datentyp: `UINT`).
  - `K = 0` wenn `EI1` ausgelöst wurde.
  - `K = 1` wenn `EI2` ausgelöst wurde.

Funktionsweise

1. **Ereignisempfang**: Der Baustein wartet auf ein Ereignis an einem seiner Eingänge (`EI1` oder `EI2`).
2. **Weiterleitung und Identifizierung**:
   - Wenn ein Ereignis an `EI1` eintrifft, wird der Daten-Ausgang `K` auf `0` gesetzt und sofort das `EO`-Ereignis ausgelöst.
   - Wenn ein Ereignis an `EI2` eintrifft, wird der Daten-Ausgang `K` auf `1` gesetzt und sofort das `EO`-Ereignis ausgelöst.

Auf diese Weise wird der Ereignisfluss zusammengeführt, wobei die Information über die Herkunft des Ereignisses erhalten bleibt.

Technische Besonderheiten

- **2-zu-1 Multiplexer**: Führt zwei Ereignisströme zu einem zusammen.
- **Herkunfts-Index**: Gibt an, welcher Eingang das Ereignis ausgelöst hat.
- **Zustandslos**: Der Baustein hat keinen internen Speicher.
- **Generischer Baustein**: Die Funktionalität wird durch die generische Klasse `GEN_E_MUX` zur Verfügung gestellt.

Anwendungsszenarien

- **Zusammenfassen von Befehlen**: Zwei verschiedene Taster (z.B. „Start Vorwärts“ und „Start Rückwärts“) lösen denselben Prozess aus, aber der Prozess muss wissen, welcher Taster gedrückt wurde.
- **Kombinierte Alarme**: Zwei verschiedene Alarme werden zu einer zentralen Alarmroutine geleitet, die dann anhand von `K` den spezifischen Alarmtext anzeigt.
- **Vereinfachung von Logik**: Statt zwei separate Pfade für ähnliche Logik zu haben, werden die auslösenden Ereignisse zusammengeführt und der Index `K` wird als Parameter für die nachfolgende Logik verwendet.

🛠️ Zugehörige Übungen

- Uebung_172

Fazit

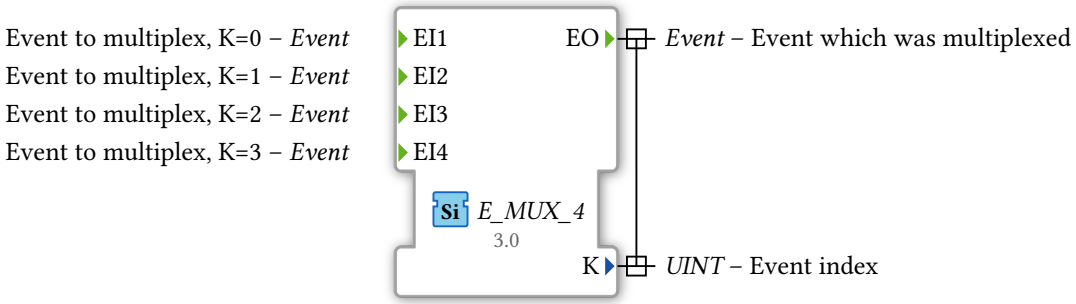
Der `E_MUX_2` ist ein nützlicher Baustein, um Ereignisströme zu bündeln und gleichzeitig die Quelle des Ereignisses zu identifizieren. Er ist das Standard-Gegenstück zum `E_DEMUX_2` und wird häufig verwendet, um die Komplexität von Verdrahtungen zu reduzieren und Logik zu zentralisieren.

2.2.3.29 E_MUX_4

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `E_MUX_4` (Event Multiplexer) ist ein Funktionsbaustein nach IEC 61499, der vier Ereigniseingänge (`EI1` bis `EI4`) auf einen einzigen Ereignisausgang (`EO`) zusammenführt. Zusätzlich zur Zusammenführung gibt der Baustein an einem Datenausgang `K` aus, welcher der Eingänge das Ereignis ausgelöst hat. Er ist das Gegenstück zum `E_DEMUX_4`.



Schnittstellenstruktur

Ereignis-Eingänge:

- **EI1**: Erster Eingangskanal.
- **EI2**: Zweiter Eingangskanal.
- **EI3**: Dritter Eingangskanal.
- **EI4**: Vierter Eingangskanal.

Ereignis-Ausgänge:

- **EO (Event Output)**: Der gemeinsame Ereignisausgang.
  - **Verbundene Daten**: `K`

Daten-Ausgänge:

- **K**: Der Index des auslösenden Eingangskanals (Datentyp: `UINT`).
  - `K = 0` wenn `EI1` ausgelöst wurde.
  - `K = 1` wenn `EI2` ausgelöst wurde.
  - `K = 2` wenn `EI3` ausgelöst wurde.
  - `K = 3` wenn `EI4` ausgelöst wurde.

Funktionsweise

1. **Ereignisempfang**: Der Baustein wartet auf ein Ereignis an einem seiner vier Eingänge.

2. **Weiterleitung und Identifizierung**: Wenn ein Ereignis an `EIn` eintrifft (wobei `n` für 1 bis 4 steht), wird der Daten-Ausgang `K` auf `n-1` gesetzt und sofort das `EO`-Ereignis ausgelöst.

Auf diese Weise wird der Ereignisfluss zusammengeführt, wobei die Information über die Herkunft des Ereignisses erhalten bleibt.

Technische Besonderheiten

- **4-zu-1 Multiplexer**: Führt vier Ereignisströme zu einem zusammen.
- **Herkunfts-Index**: Gibt an, welcher Eingang das Ereignis ausgelöst hat.
- **Zustandslos**: Der Baustein hat keinen internen Speicher.
- **Generischer Baustein**: Die Funktionalität wird durch die generische Klasse `GEN_E_MUX` zur Verfügung gestellt.

Anwendungsszenarien

- **Zusammenfassen von Befehlen**: Vier verschiedene Taster lösen denselben Prozess aus, aber der Prozess muss wissen, welcher der vier Taster gedrückt wurde.
- **Kombinierte Alarme**: Vier verschiedene Alarme werden zu einer zentralen Alarmroutine geleitet, die dann anhand von `K` den spezifischen Alarmtext anzeigt.
- **Modus-Auswahl**: Zusammenführen der Start-Ereignisse von vier verschiedenen Betriebsmodi.

🛠️ Zugehörige Übungen

- Uebung_087a2

Fazit

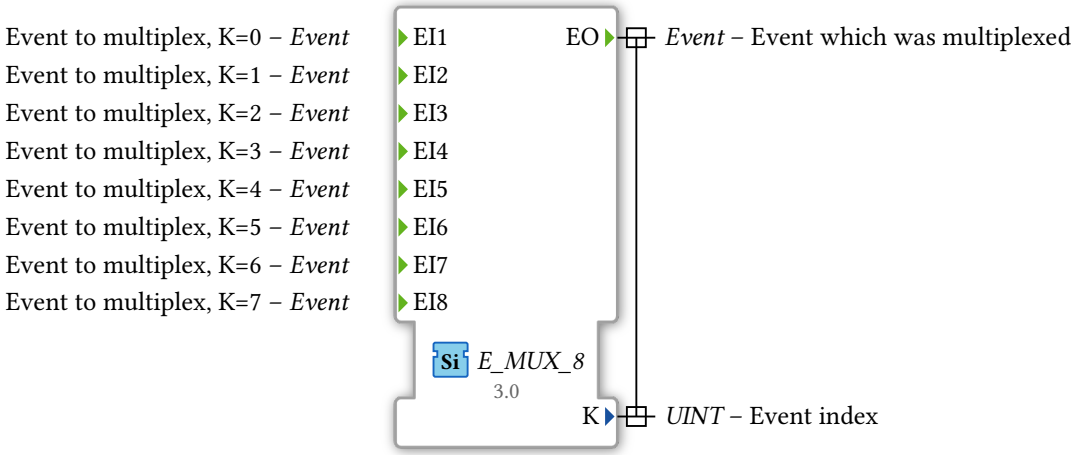
Der `E_MUX_4` ist ein nützlicher Baustein, um Ereignisströme von bis zu vier Quellen zu bündeln und gleichzeitig die Quelle des Ereignisses zu identifizieren. Er ist das Standard-Gegenstück zum `E_DEMUX_4` und wird häufig verwendet, um die Komplexität von Verdrahtungen zu reduzieren und Logik zu zentralisieren.

2.2.3.30 E_MUX_8

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `E_MUX_8` (Event Multiplexer) ist ein Funktionsbaustein nach IEC 61499, der acht Ereigniseingänge (`EI1` bis `EI8`) auf einen einzigen Ereignisausgang (`EO`) zusammenführt. Zusätzlich zur Zusammenführung gibt der Baustein an einem Datenausgang `K` aus, welcher der Eingänge das Ereignis ausgelöst hat. Er ist das Gegenstück zum `E_DEMUX_8`.



Schnittstellenstruktur

Ereignis-Eingänge:

- **EI1 ... EI8**: Die acht separaten Eingangskanäle.

Ereignis-Ausgänge:

- **EO (Event Output)**: Der gemeinsame Ereignisausgang.
  - **Verbundene Daten**: `K`

Daten-Ausgänge:

- **K**: Der Index des auslösenden Eingangskanals (Datentyp: `UINT`).
  - `K = 0` wenn `EI1` ausgelöst wurde.
  - `K = 1` wenn `EI2` ausgelöst wurde.
  - ...
  - `K = 7` wenn `EI8` ausgelöst wurde.

Funktionsweise

1. **Ereignisempfang**: Der Baustein wartet auf ein Ereignis an einem seiner acht Eingänge.
2. **Weiterleitung und Identifizierung**: Wenn ein Ereignis an `EIn` eintrifft (wobei `n` für `1` bis `8` steht), wird der Daten-Ausgang `K` auf `n-1` gesetzt und sofort das `EO`-Ereignis ausgelöst.

Auf diese Weise wird der Ereignisfluss zusammengeführt, wobei die Information über die Herkunft des Ereignisses erhalten bleibt.

Technische Besonderheiten

- **8-zu-1 Multiplexer**: Führt acht Ereignisströme zu einem zusammen.
- **Herkunfts-Index**: Gibt an, welcher Eingang das Ereignis ausgelöst hat.
- **Zustandslos**: Der Baustein hat keinen internen Speicher.
- **Generischer Baustein**: Die Funktionalität wird durch die generische Klasse `GEN_E_MUX` zur Verfügung gestellt.

Anwendungsszenarien

- **Tastatur-Matrix**: Zusammenfassen der Signale von acht Tasten auf eine zentrale Auswertungslogik.
- **Umfassende Alarmierung**: Bündeln von acht verschiedenen Alarmen zu einer zentralen Routine, die dann anhand von `K` die spezifische Alarmmeldung verarbeitet.
- **Priorisierte Befehlsauswahl**: Acht Befehlsquellen werden zusammengeführt, und eine nachgeschaltete Logik entscheidet basierend auf dem Index `K` über die Priorität.

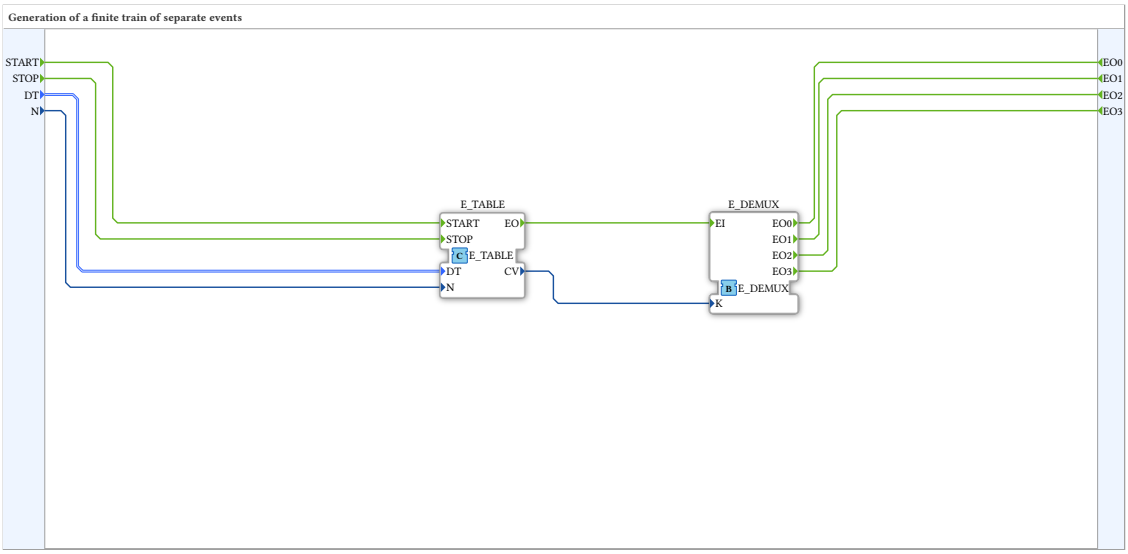
🛠️ Zugehörige Übungen

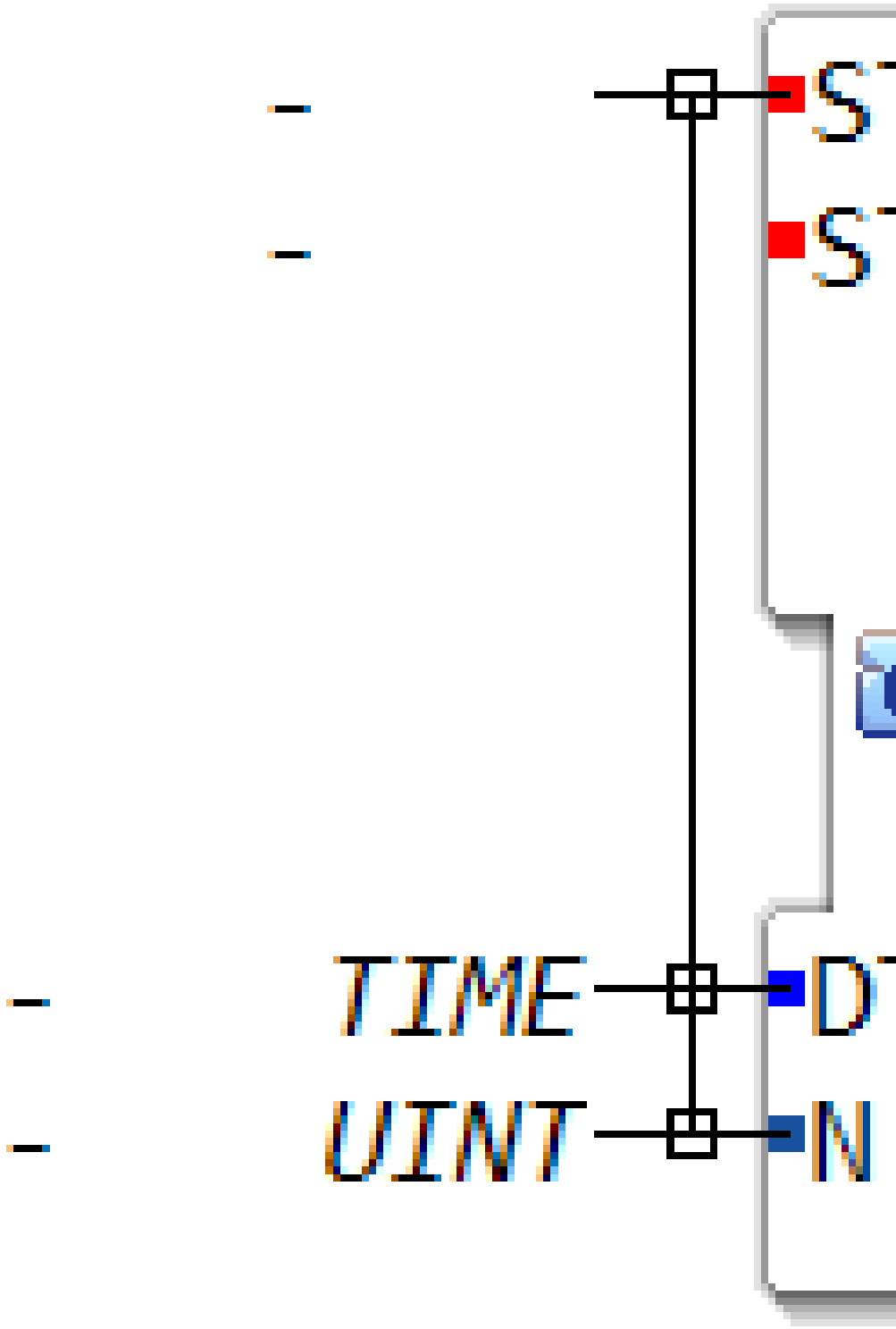
- Uebung_173

Fazit

Der `E_MUX_8` ist ein nützlicher Baustein, um Ereignisströme von bis zu acht Quellen zu bündeln und gleichzeitig die Quelle des Ereignisses zu identifizieren. Er ist das Standard-Gegenstück zum `E_DEMUX_8` und wird häufig verwendet, um die Komplexität von Verdrahtungen zu reduzieren und Logik zu zentralisieren.

2.2.3.31 E_N_TABLE

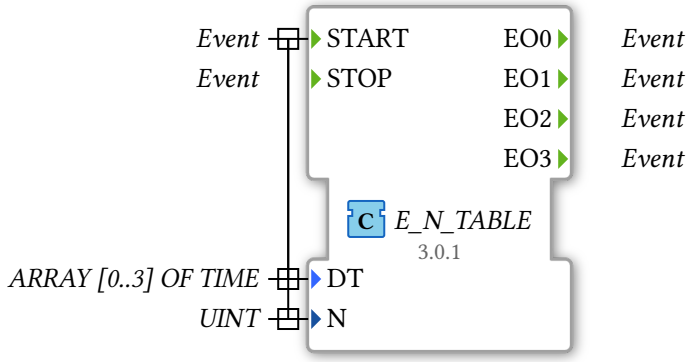




• – ◦ • – ◦ • – ◦ •

Einleitung

Der `E_N_TABLE` (Event N-Table) ist ein Funktionsbaustein nach IEC 61499, der eine endliche Sequenz von zeitlich versetzten Ereignissen auf separaten Ausgängen generiert. Er kombiniert intern einen tabellengesteuerten Zeitgeber (`E_TABLE`) mit einem Demultiplexer (`E_DEMUX`), um eine Kette von Ereignissen zu erzeugen, bei der jedes Ereignis auf einem eigenen Kanal ausgegeben wird.



Schnittstellenstruktur

Ereignis-Eingänge

- **START**: Startet die Generierung der Ereignissequenz.
  - **Verbundene Daten**: `DT`, `N`
- **STOP**: Stoppt die Sequenz vorzeitig.

Ereignis-Ausgänge

- **EO0**: Gibt das erste Ereignis der Sequenz aus (Index 0).
- **EO1**: Gibt das zweite Ereignis der Sequenz aus (Index 1).
- **EO2**: Gibt das dritte Ereignis der Sequenz aus (Index 2).
- **EO3**: Gibt das vierte Ereignis der Sequenz aus (Index 3).

Daten-Eingänge

- **DT**: Ein Array von Zeitdauern (Datentyp: `TIME`, Größe: 4). `DT[i]` definiert die Verzögerungszeit, die *vor* dem `i`-ten Ereignis gewartet wird.
- **N**: Die Gesamtzahl der zu generierenden Ereignisse (Datentyp: `UINT`, max. 4 für diesen Baustein).

Funktionsweise

1. **Start der Sequenz**: Ein `START`-Ereignis löst den internen `E_TABLE`-Baustein aus. Die Anzahl der zu generierenden Ereignisse wird durch `N` festgelegt.
2. **Erstes Ereignis**: Der Baustein wartet die in `DT[0]` definierte Zeitspanne ab. Danach wird das erste Ereignis am Ausgang `EO0` ausgelöst.
3. **Folgeereignisse**: Der Baustein wartet die in `DT[1]` definierte Zeitspanne ab und löst dann das zweite Ereignis an `EO1` aus. Dieser Prozess wiederholt sich für die nächsten Ereignisse gemäß den Zeitdauern in `DT[2]`, `DT[3]` usw., bis `N` Ereignisse generiert wurden.
4. **Ende der Sequenz**: Die Sequenz endet automatisch, nachdem `N` Ereignisse ausgelöst wurden.
5. **Stopp**: Ein `STOP`-Ereignis bricht die Sequenz an jeder Stelle sofort ab.

Beispiel:

- `N = 3`
- `DT = [T#2s, T#5s, T#1s]`
- Nach einem `START`-Ereignis:
  1. Nach 2 Sekunden wird `EO0` ausgelöst.
  2. Weitere 5 Sekunden später wird `EO1` ausgelöst.
  3. Weitere 1 Sekunde später wird `EO2` ausgelöst.
  4. Die Sequenz ist beendet. `EO3` wird nicht ausgelöst.

Technische Besonderheiten

- **Sequenzgenerator**: Erzeugt eine Kette von einzelnen, zeitlich versetzten Ereignissen.
- **Demultiplexing**: Jedes Ereignis in der Kette wird auf einen eigenen, dedizierten Ausgang geleitet.
- **Tabellengesteuert**: Die Zeitabstände zwischen den Ereignissen sind nicht fix, sondern werden aus dem `DT`-Array gelesen.

Anwendungsszenarien

- **Steuerung von Schrittketten**: Jeder Ausgang (`EO0`, `EO1`, ...) kann einen nachfolgenden Schritt in einer Prozesskette anstoßen.
- **Komplexe Ansteuerung**: Ansteuern verschiedener Aktoren in einer genau definierten zeitlichen Abfolge (z.B. Ventile in einem Spülprozess).
- **Testautomatisierung**: Generieren einer komplexen, zeitlich definierten Stimulus-Sequenz für ein Testobjekt.

🛠️ Zugehörige Übungen

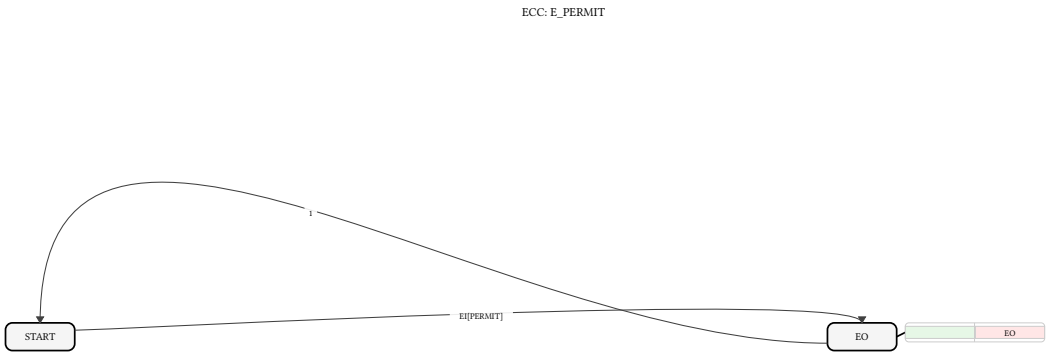
- Uebung_093b

Fazit

Der `E_N_TABLE` ist ein mächtiger Baustein zur Erzeugung von komplexen, zeitlich definierten Ereignisketten. Im Gegensatz zu einem einfachen `E_CYCLE`, der nur einen periodischen Takt erzeugt, ermöglicht `E_N_TABLE` die Definition von variablen Zeitabständen und die Verteilung der einzelnen Sequenz-Ereignisse auf separate Kanäle. Dies macht ihn ideal für die Steuerung von sequenziellen Prozessen.

2.2.3.32 E_PERMIT

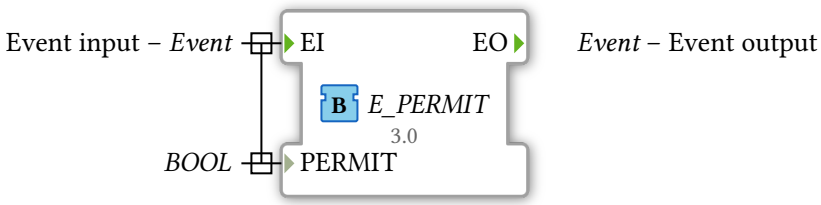
🎧 Podcast



- Der E_PERMIT-Baustein: Der „Türsteher“ für Ereignisse in IEC 61499-Systemen entschlüsselt
- Decoding E_PERMIT: The Unsung Hero of Industrial Automation’s Safety and Reliability

Einleitung

Der `E_PERMIT` (Event Permit) ist ein grundlegender Funktionsbaustein nach IEC 61499, der als steuerbares „Tor“ für Ereignisse fungiert. Er lässt ein Eingangereignis nur dann zum Ausgang passieren, wenn eine explizite Freigabebedingung erfüllt ist.



Schnittstellenstruktur

Ereignis-Eingänge:

- **EI (Event Input)**: Das zu prüfende Eingangereignis.
  - **Verbundene Daten**: `PERMIT`

Ereignis-Ausgänge:

- **EO (Event Output)**: Der Ereignisausgang, der nur ausgelöst wird, wenn die Freigabe erteilt wurde.

Daten-Eingänge:

- **PERMIT**: Die Freigabebedingung (Datentyp: `BOOL`).

Funktionsweise

1. **Ereignisempfang**: Der Baustein wartet auf ein Ereignis am Eingang `EI`.
2. **Prüfung der Freigabe**: Wenn das `EI`-Ereignis eintrifft, wird der Wert der Datenvariable `PERMIT` ausgewertet.
3. **Bedingte Weiterleitung**:
   - **Wenn `PERMIT` = `TRUE` ist**: Das Ereignis wird durchgelassen und am Ausgang `EO` ausgegeben.
   - **Wenn `PERMIT` = `FALSE` ist**: Das Ereignis wird blockiert und es geschieht nichts. Der Ausgang `EO` wird nicht ausgelöst.

Der Baustein agiert somit als einfacher Wächter für den Ereignisfluss.

Technische Besonderheiten

- **Ereignis-Tor**: Dient als grundlegendes „Gate“ zur Steuerung von Ereignisströmen.
- **Zustandslos**: Der Baustein selbst hat keinen internen Speicher; seine Entscheidung basiert ausschließlich auf dem Wert von `PERMIT` im Moment des `EI`-Ereignisses.

Anwendungsszenarien

- **Freigaben/Interlocks**: Ein Prozessschritt (`EI`) darf nur dann gestartet werden, wenn eine Sicherheitsfreigabe (`PERMIT`) vorliegt (z.B. Schutztür geschlossen).
- **Betriebsarten-Umschaltung**: Befehle von einer manuellen Steuerung (`EI`) werden nur dann weitergeleitet, wenn die Anlage im „Hand“-Modus ist (`PERMIT` = true).
- **Daten-Validierung**: Ein Ereignis, das die Weiterverarbeitung von Daten anstößt, wird nur ausgelöst, wenn eine vorherige Prüfung der Daten erfolgreich war (`PERMIT` = true).

⚖️ Vergleich mit ähnlichen Bausteinen

- **`E_SWITCH`**: Während `E_PERMIT` ein Ereignis entweder durchlässt oder blockiert (1-zu-1 oder 1-zu-0), leitet `E_SWITCH` ein Ereignis an einen von zwei verschiedenen Ausgängen weiter (1-zu-2). `E_PERMIT` ist ein Tor, `E_SWITCH` ist eine Weiche.

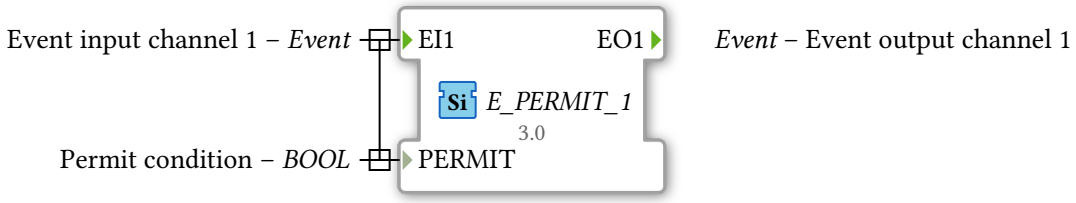
🛠️ Zugehörige Übungen

- Uebung_009
- Uebung_080c
- Uebung_094

Fazit

Der `E_PERMIT`-Baustein ist ein fundamentaler und weit verbreiteter Baustein zur Implementierung von Bedingungen und Freigaben in der ereignisgesteuerten Logik von IEC 61499. Seine Einfachheit und klare Funktion machen ihn zu einem unverzichtbaren Werkzeug für die Erstellung von sicheren und robusten Steuerungsabläufen.

2.2.3.33 E_PERMIT_1



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `E_PERMIT_1` ist ein Funktionsbaustein zur **bedingten Weiterleitung eines Ereigniskanals** (generischer Baustein). Das Ereignis `EI1` wird nur dann an `EO1` weitergegeben, wenn die Freigabebedingung `PERMIT` erfüllt ist. Er ist die Variante mit einem Kanal der generischen `E_PERMIT`-Familie.

Schnittstellenstruktur

Ereignis-Eingänge

- **EI1**: Ereigniseingang, Kanal 1. Verbundene Daten: `PERMIT`

Ereignis-Ausgänge

- **EO1**: Ereignisausgang, Kanal 1

Daten-Eingänge

- **PERMIT** (`BOOL`): Freigabebedingung

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei jedem `EI1`-Ereignis wird `PERMIT` ausgewertet. Ist `PERMIT = TRUE`, wird `EO1` ausgelöst; andernfalls wird das Ereignis unterdrückt.

Technische Besonderheiten

- Generischer Baustein, der native Implementierung `GEN_E_PERMIT` (FORTE) benötigt.
- Reiner Ereignisfilter: Es werden keine Daten gespeichert oder ausgegeben.
- `PERMIT` ist mit `EI1` verbunden und wird daher bei jedem Ereignis neu gelesen.
- Siehe auch `E_PERMIT` (Standardvariante).

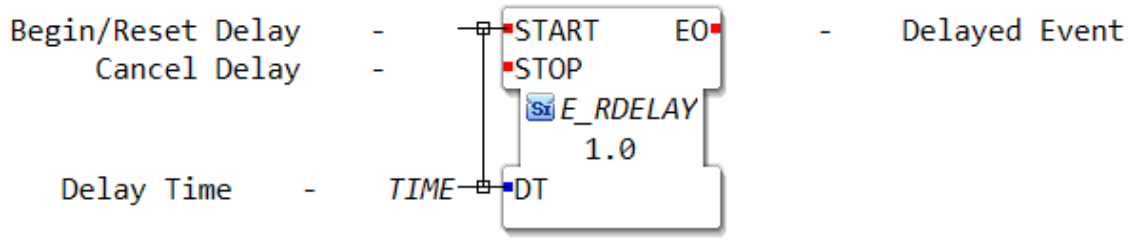
Anwendungsszenarien

- Ereignisse nur bei aktiver Freigabe weiterleiten, z. B. Tastenereignisse nur im Automatikbetrieb
- Sperren von Ereignispfaden durch ein Statussignal

Fazit

`E_PERMIT_1` leitet ein einzelnes Ereignis nur bei erfüllter `PERMIT`-Bedingung weiter.

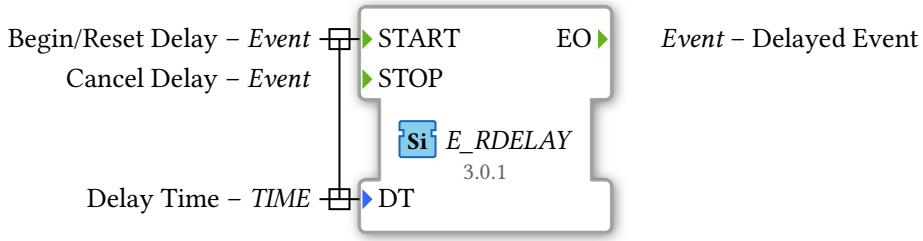
2.2.3.34 E_RDELAY



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **E_RDELAY** (Reloadable Delay) ist ein erweiterter Verzögerungsbaustein nach IEC 61499, der im Gegensatz zum einfachen E_DELAY eine nachladbare und abbrechbare Ereignisverzögerung ermöglicht. Entwickelt unter EPL-2.0 Lizenz.



Schnittstellenstruktur

Ereignis-Eingänge

- `START`: Beginnt/Resetet die Verzögerung (mit DT-Parameter)
- `STOP`: Bricht die aktive Verzögerung ab

Ereignis-Ausgänge

- `EO`: Verzögertes Ausgangsereignis

Daten-Eingänge

- `DT` (Delay Time): Verzögerungsdauer (TIME-Datentyp)

Funktionsprinzip

1. **Verzögerungsstart**:
   - Bei `START`-Ereignis beginnt Timer mit konfigurierter `DT`-Zeit
   - Neues `START` während aktiver Verzögerung resetet den Timer
2. **Verzögerungsabbruch**:
   - `STOP` bricht aktive Verzögerung sofort ab
   - Kein `EO`-Ereignis wird generiert
3. **Verzögerungsabschluss**:
   - Nach exakt `DT` wird `EO` einmalig ausgelöst
   - Bei DT ≤ T#0s sofortige Auslösung

Service-Sequenzen (laut XML-Spezifikation)

1. **event_delay**:
   - Normale Verzögerung mit START ⟶ EO
2. **delay_canceled**:
   - START gefolgt von STOP (kein EO)
3. **reload_delay**:

- Mehrfaches START lädt die Verzögerung neu (reset), es löst nur ein EO aus

Technische Besonderheiten

✓ **Nachladbare Verzögerung** (Reset-Funktion) ✓ **Abbruchfähig** während der Laufzeit ✓ **Deterministisches Zeitverhalten** ✓ **Echtzeitfähige** Implementierung

Anwendungsszenarien

- **Maschinensicherheit**: Verzögerte Abschaltsequenzen
- **Prozesssteuerung**: Zeitgesteuerte Zustandsübergänge
- **Alarmsysteme**: Störmeldungsverzögerung
- **Robotersteuerung**: Bewegungsablauf-Timing

⚖️ Vergleich mit E_DELAY

Feature	E_RDELAY	E_DELAY
Reset-Funktion	✓ (durch START)	❌
Mehrfach-Trigger	Nur 1 EO	Nur 1 EO
Service-Sequenzen	3 definiert (`event_delay`, `delay_canceled`, `reload_delay`)	3 definiert (`event_delay`, `delay_canceled`, `no_multiple_delay`)

🛠️ Zugehörige Übungen

- Uebung_018a

Fazit

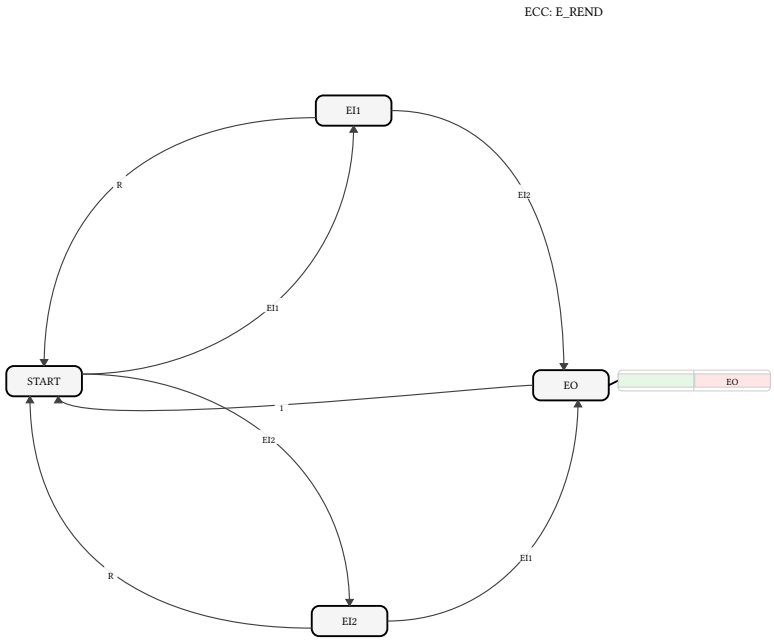
Der E_RDELAY-Baustein erweitert die klassische Verzögerungsfunktion um wesentliche industrietaugliche Features:

- Flexible Neukonfiguration während des Betriebs
- Sichere Abbruchmöglichkeit von Zeitabläufen
- Standardisierte Service-Schnittstellen

Durch die XML-basierte Spezifikation ist der Baustein besonders gut für den Einsatz in verteilten Steuerungssystemen geeignet. Die implementierte Version bietet zuverlässige Zeitsteuerung für kritische Automatisierungsprozesse.

2.2.3.35 E_REND

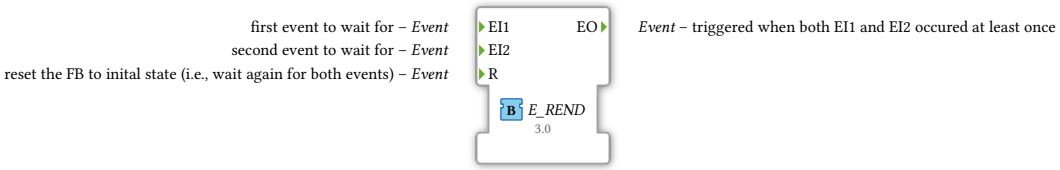
🎧 Podcast



- E_REND: Ereignissynchronisation in IEC 61499
- E_REND: Event Synchronization in IEC 61499

Einleitung

Der `E_REND` (Event Rendezvous) ist ein Funktionsbaustein nach IEC 61499, der als Synchronisationspunkt für zwei verschiedene Ereignisströme dient. Er feuert ein Ausgangsereignis erst dann, wenn er jeweils mindestens ein Ereignis von beiden seiner Eingänge empfangen hat. Dies entspricht einer logischen UND-Verknüpfung über die Zeit.



Schnittstellenstruktur

Ereignis-Eingänge:

- **EI1**: Der erste Ereigniseingang.
- **EI2**: Der zweite Ereigniseingang.
- **R (Reset)**: Setzt den Baustein in seinen Anfangszustand zurück.

Ereignis-Ausgänge:

- **EO (Event Output)**: Wird ausgelöst, nachdem sowohl `EI1` als auch `EI2` mindestens einmal seit dem letzten Reset empfangen wurden.

Funktionsweise

1. **Warten auf Ereignisse**: Im Anfangszustand wartet der Baustein auf Ereignisse an `EI1` und `EI2`.
2. **Speichern des ersten Ereignisses**: Trifft ein Ereignis am ersten Eingang ein (z.B. `EI1`), merkt sich der Baustein dies intern und wartet nun auf das Ereignis am zweiten Eingang (`EI2`).

3. **Rendezvous und Auslösung**: Sobald das zweite Ereignis (`EI2`) eintrifft, ist die „Rendezvous“-Bedingung erfüllt. Der Baustein löst sofort das Ausgangsereignis `EO` aus.
4. **Automatischer Reset**: Unmittelbar nach dem Auslösen von `EO` setzt sich der Baustein automatisch in seinen Anfangszustand zurück und wartet auf das nächste Paar von `EI1`- und `EI2`-Ereignissen.
5. **Externer Reset**: Ein Ereignis am `R`-Eingang setzt den Baustein jederzeit in den Anfangszustand zurück, wodurch alle bisher gemerkten, aber noch nicht vollständigen Ereignispaare verworfen werden.

Wichtig: Die Eingangsereignisse `EI1` und `EI2` müssen **nicht gleichzeitig** eintreffen. Die Reihenfolge ihres Eintreffens ist ebenfalls unerheblich.

Technische Besonderheiten

- **Zustandsbehaftet**: Im Gegensatz zu einem `E_MERGE` (OR) hat der `E_REND` einen internen Zustand, um sich das Eintreffen des ersten Ereignisses zu merken.
- **Synchronisation**: Dient der Synchronisation von zwei asynchronen Ereignisflüssen.

Anwendungsszenarien

- **Prozess-Synchronisation**: Ein nächster Prozessschritt (`EO`) darf erst starten, wenn zwei voneinander unabhängige Vorbedingungen erfüllt sind (z.B. „Bauteil eingespannt“ (`EI1`) und „Schutztür geschlossen“ (`EI2`)).
- **Quittierung**: Eine Aktion wird erst ausgeführt, wenn sowohl ein Befehl (`EI1`) als auch eine manuelle Quittierung durch einen Bediener (`EI2`) vorliegen.
- **Materialfluss**: Ein Förderband stoppt (`EO`), wenn sowohl der vordere als auch der hintere Sensor ein langes Bauteil detektiert haben.

🛠️ Zugehörige Übungen

- Uebung_004a6
- Uebung_004a6_AX
- Uebung_004a7
- Uebung_004a7_AX
- Uebung_025
- Uebung_026

Fazit

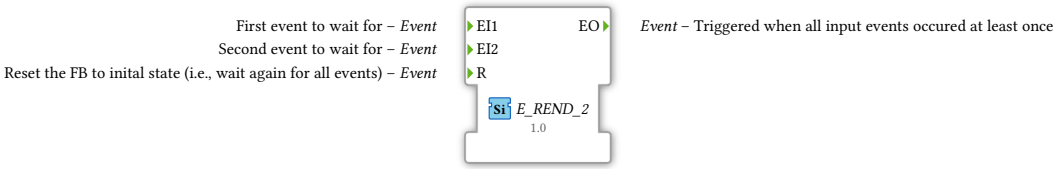
Der `E_REND`-Baustein ist ein kritisches Werkzeug für die Synchronisation von Prozessen in IEC 61499. Er bietet eine robuste Methode, um sicherzustellen, dass mehrere Bedingungen erfüllt sind, bevor eine nachfolgende Aktion ausgelöst wird. Seine Fähigkeit, sich das Eintreffen von Ereignissen zu „merken“, macht ihn wesentlich leistungsfähiger als einfache logische Verknüpfungen und ist fundamental für die Steuerung von asynchronen und parallelen Abläufen.

2.2.3.36 E_REND_2

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `E_REND_2` ist die generische 2-Eingangs-Variante des Rendezvous-Bausteins `E_REND` (2 Eingänge). Er löst `EO` erst aus, wenn er von **allen** 2 Eingängen (`EI1`, `EI2`) mindestens ein Ereignis seit dem letzten Reset empfangen hat — eine logische UND-Verknüpfung über 2 asynchrone Ereignisströme.



Schnittstellenstruktur

Ereignis-Eingänge

- **EI1**: Ereigniseingang 1.
- **EI2**: Ereigniseingang 2.

- **R (Reset)**: Setzt den Baustein in seinen Anfangszustand zurück, verwirft alle bisher gemerkten Ereignisse.

Ereignis-Ausgänge

- **EO**: Wird ausgelöst, sobald alle 2 Eingänge seit dem letzten Reset mindestens einmal ausgelöst wurden.

Funktionsweise

`E_REND_2` ist als generischer Funktionsbaustein implementiert (`eclipse4diac::core::GenericClassName = 'GEN_E_REND'`) — dieselbe C++-Implementierung liegt allen `E_REND_n`-Varianten zugrunde, nur die Anzahl der Eingänge unterscheidet sich. Intern merkt sich der Baustein, welche der `EI1`…`EI2`-Eingänge bereits mindestens ein Ereignis erhalten haben. Sobald alle 2 Eingänge markiert sind, wird `EO` ausgelöst und der interne Merker zurückgesetzt. Ein `R`-Ereignis setzt den Merker jederzeit zurück, ohne `EO` auszulösen.

Technische Besonderheiten

- **Generische Implementierung**: Wie `E_REND` (2 Eingänge) selbst basiert `E_REND_2` auf derselben generischen `GEN_E_REND`-C++-Klasse; die Eingangsanzahl wird zur Übersetzungszeit über die Typdefinition festgelegt.
- **Reihenfolgeunabhängig**: Die Eingänge `EI1`, `EI2` müssen nicht in einer bestimmten Reihenfolge eintreffen.
- **Automatischer Reset nach Auslösung**: Nach jedem `EO` beginnt der Baustein wieder von vorn.

Zustandsübersicht

Der Baustein merkt sich intern eine Bitmaske der bereits eingetroffenen Eingänge; bei vollständiger Maske wird `EO` ausgelöst und die Maske geleert.

Anwendungsszenarien

- **Synchronisation von 2 unabhängigen Vorbedingungen**, bevor ein Folgeschritt ausgelöst wird (z. B. 2 Sensoren müssen alle ausgelöst haben, bevor eine Anlage weiterläuft).
- **Erweiterung von `E_REND`** auf mehr als zwei Ereignisquellen, ohne mehrere `E_REND`-Bausteine kaskadieren zu müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`E_REND`**: die 2-Eingangs-Basisvariante.
- `E_REND_3`, `E_REND_4`: dieselbe generische Implementierung mit einer anderen Eingangsanzahl.
- **`E_MERGE_2`**: die logische ODER-Verknüpfung mehrerer Ereigniseingänge (feuert bei jedem einzelnen Ereignis statt erst nach allen).

Fazit

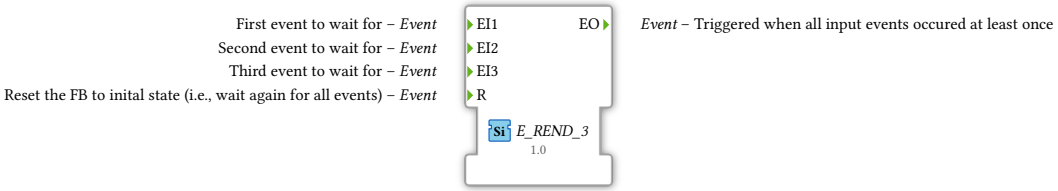
`E_REND_2` erweitert das Rendezvous-Prinzip von `E_REND` auf 2 gleichzeitig zu synchronisierende Ereignisströme und ist die generische, für 2 Eingänge instanziierte Variante derselben zugrunde liegenden Implementierung.

2.2.3.37 E_REND_3

• – ○ • – ○ • – ○ •

Einleitung

Der `E_REND_3` ist die generische 3-Eingangs-Variante des Rendezvous-Bausteins `E_REND` (2 Eingänge). Er löst `EO` erst aus, wenn er von **allen** 3 Eingängen (`EI1`, `EI2`, `EI3`) mindestens ein Ereignis seit dem letzten Reset empfangen hat — eine logische UND-Verknüpfung über 3 asynchrone Ereignisströme.



Schnittstellenstruktur

Ereignis-Eingänge

- **EI1**: Ereigniseingang 1.
- **EI2**: Ereigniseingang 2.
- **EI3**: Ereigniseingang 3.
- **R (Reset)**: Setzt den Baustein in seinen Anfangszustand zurück, verwirft alle bisher gemerkten Ereignisse.

Ereignis-Ausgänge

- **EO**: Wird ausgelöst, sobald alle 3 Eingänge seit dem letzten Reset mindestens einmal ausgelöst wurden.

Funktionsweise

`E_REND_3` ist als generischer Funktionsbaustein implementiert (`eclipse4diac::core::GenericClassName = 'GEN_E_REND'`) — dieselbe C++-Implementierung liegt allen `E_REND_n`-Varianten zugrunde, nur die Anzahl der Eingänge unterscheidet sich. Intern merkt sich der Baustein, welche der `EI1`…`EI3`-Eingänge bereits mindestens ein Ereignis erhalten haben. Sobald alle 3 Eingänge markiert sind, wird `EO` ausgelöst und der interne Merker zurückgesetzt. Ein `R`-Ereignis setzt den Merker jederzeit zurück, ohne `EO` auszulösen.

Technische Besonderheiten

- **Generische Implementierung**: Wie `E_REND` (2 Eingänge) selbst basiert `E_REND_3` auf derselben generischen `GEN_E_REND`-C++-Klasse; die Eingangsanzahl wird zur Übersetzungszeit über die Typdefinition festgelegt.
- **Reihenfolgeunabhängig**: Die Eingänge `EI1`, `EI2`, `EI3` müssen nicht in einer bestimmten Reihenfolge eintreffen.
- **Automatischer Reset nach Auslösung**: Nach jedem `EO` beginnt der Baustein wieder von vorn.

Zustandsübersicht

Der Baustein merkt sich intern eine Bitmaske der bereits eingetroffenen Eingänge; bei vollständiger Maske wird `EO` ausgelöst und die Maske geleert.

Anwendungsszenarien

- **Synchronisation von 3 unabhängigen Vorbedingungen**, bevor ein Folgeschritt ausgelöst wird (z. B. 3 Sensoren müssen alle ausgelöst haben, bevor eine Anlage weiterläuft).
- **Erweiterung von `E_REND`** auf mehr als zwei Ereignisquellen, ohne mehrere `E_REND`-Bausteine kaskadieren zu müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`E_REND`**: die 2-Eingangs-Basisvariante.
- `E_REND_2`, `E_REND_4`: dieselbe generische Implementierung mit einer anderen Eingangsanzahl.
- **`E_MERGE_3`**: die logische ODER-Verknüpfung mehrerer Ereigniseingänge (feuert bei jedem einzelnen Ereignis statt erst nach allen).

Fazit

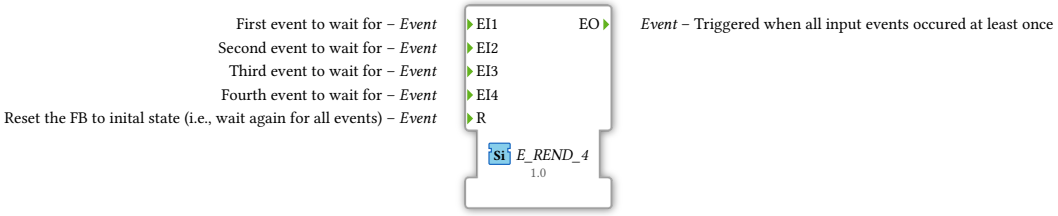
`E_REND_3` erweitert das Rendezvous-Prinzip von `E_REND` auf 3 gleichzeitig zu synchronisierende Ereignisströme und ist die generische, für 3 Eingänge instanziierte Variante derselben zugrunde liegenden Implementierung.

2.2.3.38 E_REND_4

• – ○ • – ○ • – ○ •

Einleitung

Der `E_REND_4` ist die generische 4-Eingangs-Variante des Rendezvous-Bausteins `E_REND` (2 Eingänge). Er löst `EO` erst aus, wenn er von **allen** 4 Eingängen (`EI1`, `EI2`, `EI3`, `EI4`) mindestens ein Ereignis seit dem letzten Reset empfangen hat — eine logische UND-Verknüpfung über 4 asynchrone Ereignisströme.



Schnittstellenstruktur

Ereignis-Eingänge

- **EI1**: Ereigniseingang 1.
- **EI2**: Ereigniseingang 2.
- **EI3**: Ereigniseingang 3.
- **EI4**: Ereigniseingang 4.
- **R (Reset)**: Setzt den Baustein in seinen Anfangszustand zurück, verwirft alle bisher gemerkten Ereignisse.

Ereignis-Ausgänge

- **EO**: Wird ausgelöst, sobald alle 4 Eingänge seit dem letzten Reset mindestens einmal ausgelöst wurden.

Funktionsweise

`E_REND_4` ist als generischer Funktionsbaustein implementiert (`eclipse4diac::core::GenericClassName = 'GEN_E_REND'`) — dieselbe C++-Implementierung liegt allen `E_REND_n`-Varianten zugrunde, nur die Anzahl der Eingänge unterscheidet sich. Intern merkt sich der Baustein, welche der `EI1`…`EI4`-Eingänge bereits mindestens ein Ereignis erhalten haben. Sobald alle 4 Eingänge markiert sind, wird `EO` ausgelöst und der interne Merker zurückgesetzt. Ein `R`-Ereignis setzt den Merker jederzeit zurück, ohne `EO` auszulösen.

Technische Besonderheiten

- **Generische Implementierung**: Wie `E_REND` (2 Eingänge) selbst basiert `E_REND_4` auf derselben generischen `GEN_E_REND`-C++-Klasse; die Eingangsanzahl wird zur Übersetzungszeit über die Typdefinition festgelegt.
- **Reihenfolgeunabhängig**: Die Eingänge `EI1`, `EI2`, `EI3`, `EI4` müssen nicht in einer bestimmten Reihenfolge eintreffen.
- **Automatischer Reset nach Auslösung**: Nach jedem `EO` beginnt der Baustein wieder von vorn.

Zustandsübersicht

Der Baustein merkt sich intern eine Bitmaske der bereits eingetroffenen Eingänge; bei vollständiger Maske wird `EO` ausgelöst und die Maske geleert.

Anwendungsszenarien

- **Synchronisation von 4 unabhängigen Vorbedingungen**, bevor ein Folgeschritt ausgelöst wird (z. B. 4 Sensoren müssen alle ausgelöst haben, bevor eine Anlage weiterläuft).
- **Erweiterung von `E_REND`** auf mehr als zwei Ereignisquellen, ohne mehrere `E_REND`-Bausteine kaskadieren zu müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`E_REND`**: die 2-Eingangs-Basisvariante.
- `E_REND_2`, `E_REND_3`: dieselbe generische Implementierung mit einer anderen Eingangsanzahl.
- **`E_MERGE_4`**: die logische ODER-Verknüpfung mehrerer Ereigniseingänge (feuert bei jedem einzelnen Ereignis statt erst nach allen).

Fazit

`E_REND_4` erweitert das Rendezvous-Prinzip von `E_REND` auf 4 gleichzeitig zu synchronisierende Ereignisströme und ist die generische, für 4 Eingänge instanziierte Variante derselben zugrunde liegenden Implementierung.

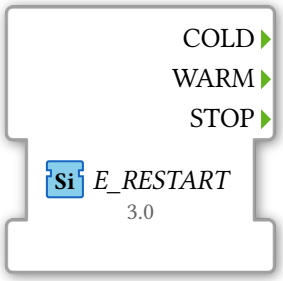
2.2.3.39 E_RESTART

- Information on cold restart
- Information on warm restart
- information that the resource is to be stopped

• – ◦ • – ◦ • – ◦ •

Einleitung

Der **E_RESTART** ist ein spezieller Service-Interface-Funktionsbaustein nach IEC 61499 (Annex A), zur Steuerung von Neustartsequenzen in verteilten Automatisierungssystemen. Der Baustein dient als Schnittstelle zwischen Ressourcen und übergeordneten Steuerungen.



Event – Information on cold restart
Event – Information on warm restart
Event – information that the resource is to be stopped

Schnittstellenstruktur

Ereignis-Ausgänge

- `COLD`: Signalisiert einen Kaltstart (vollständiger Reset)
- `WARM`: Kennzeichnet einen Warmstart (partieller Reset)
- `STOP`: Informiert über anstehenden Stopp-Befehl

Service-Schnittstellen

- Links: E_RESTART (FB-Schnittstelle)
- Rechts: RESOURCE (Geräteschnittstelle)

Funktionsweise

1. **Kaltstart-Sequenz**:
   - Bei externem `start`-Kommando an die Ressource
   - Auslösung des `COLD`-Ereignisses
2. **Warmstart-Sequenz**:
   - Bei `restart`-Befehl der Ressource
   - Generierung des `WARM`-Ereignisses
3. **Stopp-Sequenz**:
   - Bei Empfang des `stop`-Signals
   - Ausgabe des `STOP`-Ereignisses

Service-Sequenzen (XML-Spezifikation)

1. **cold_restart**:
   - `start` ⟶ `COLD` (Vollständige Reinitialisierung)
2. **warm_restart**:
   - `restart` ⟶ `WARM` (Zustandserhaltender Neustart)
3. **stopping**:

- `stop` ⟶ `STOP` (Geordnetes Herunterfahren)

Technische Besonderheiten

✓ **Standardisierte Neustart-Steuerung** nach IEC 61499 Annex A ✓ **Drei Betriebsmodi** (Cold/Warm/Stop) ✓ **Ressourcenübergreifende** Konsistenz ✓ **EPL-2.0 Open-Source** Implementierung

Anwendungsszenarien

- **Feldgeräte-Steuerung**: Geordnete Reinitialisierung
- **Systemrecovery**: Automatische Neustartroutinen
- **Energiemanagement**: Geplantes Herunterfahren
- **Sicherheitskritische Systeme**: Zustandsgesicherte Restarts

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	E_RESTART	E_CYCLE	E_DELAY
Zweck	Systemsteuerung	Zeitsteuerung	Ereignisverzögerung
Ereignistyp	Steuerbefehle	Periodisch	Einmalig
Standard	Annex A	Core	Core

🛠️ Zugehörige Übungen

- Uebung_174

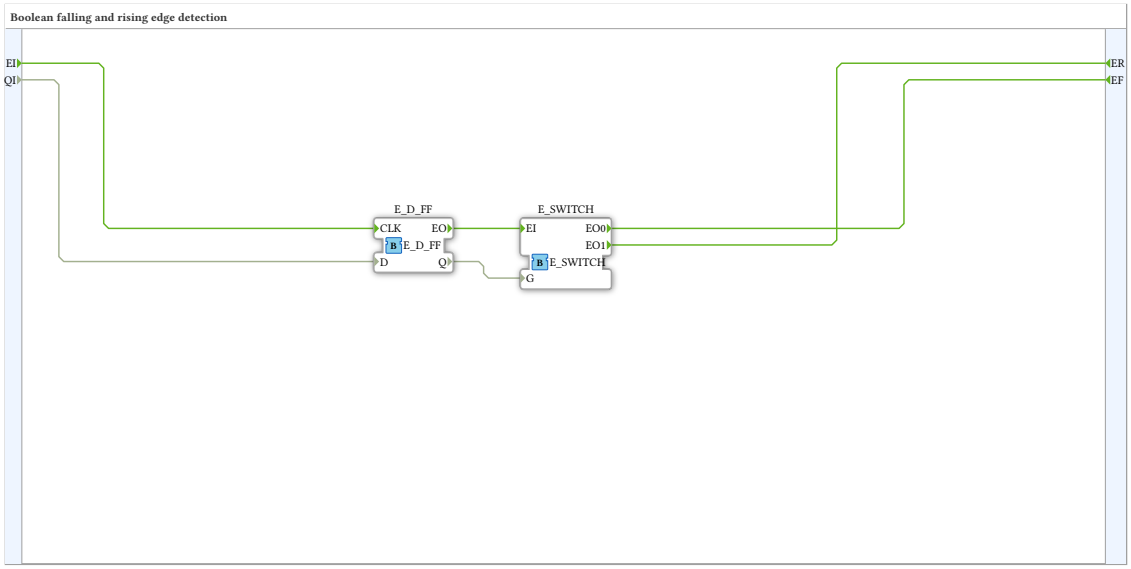
Fazit

Der E_RESTART-Baustein bietet eine normkonforme Lösung für kritische Systemsteuerungsaufgaben:

- Standardisiertes Neustart-Management
- Trennung von Kalt-/Warmstart-Logik
- Zuverlässige Signalweiterleitung

Ermöglicht besonders die Entwicklung von robusten, fehlertoleranten Steuerungssystemen. Die klare Service-Schnittstelle nach IEC 61499 Annex A gewährleistet Interoperabilität über verschiedene Geräteplattformen hinweg.

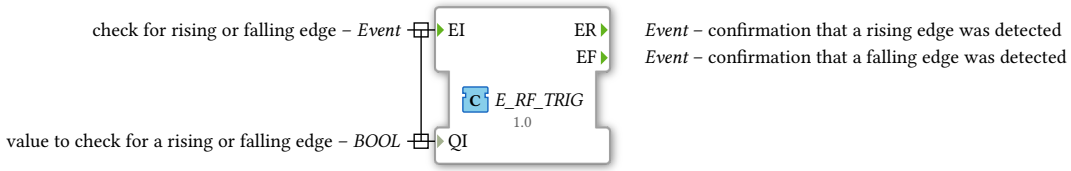
2.2.3.40 E_RF_TRIG



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `E_RF_TRIG` (Rising/Falling Trigger) erkennt sowohl steigende als auch fallende Flanken eines `BOOL`-Werts und meldet sie über zwei getrennte Ausgangsereignisse. Er kombiniert die Funktionalität von `E_R_TRIG` (steigende Flanke) und `E_F_TRIG` (fallende Flanke) in einem einzigen Baustein.



Schnittstellenstruktur

Ereignis-Eingänge

- **EI**: Prüft `QI` auf eine steigende oder fallende Flanke.
  - **Verbundene Daten**: `QI`

Ereignis-Ausgänge

- **ER**: Bestätigt eine erkannte steigende Flanke (`QI` von `FALSE` nach `TRUE`).
- **EF**: Bestätigt eine erkannte fallende Flanke (`QI` von `TRUE` nach `FALSE`).

Daten-Eingänge

- **QI** (BOOL): Der auf Flankenwechsel zu prüfende Wert.

Funktionsweise

Intern speichert ein `E_D_FF` den zuletzt bekannten Wert von `QI` und liefert bei jedem `EI`-Ereignis über sein `EO` den neuen Wert an einen `E_SWITCH`. Dieser leitet das Ereignis abhängig vom aktuellen `Q`-Wert (`G`-Eingang) entweder an `EO1` (⟶ `ER`, steigende Flanke) oder `EO0` (⟶ `EF`, fallende Flanke) weiter. Da `E_D_FF` `EO` nur bei tatsächlicher Änderung von `Q` auslöst, wird bei unverändertem `QI` weder `ER` noch `EF` ausgelöst.

Technische Besonderheiten

- **Zusammengesetzt aus zwei Standardbausteinen**: `E_D_FF` (Änderungserkennung) + `E_SWITCH` (Verzweigung nach Wert), keine eigene ECC.
- **Nur bei tatsächlicher Änderung**: Wiederholte `EI`-Ereignisse mit unverändertem `QI` erzeugen kein Ausgangsereignis, da `E_D_FF` intern bereits filtert.
- **Basis für E_TLIM**: Dort wird `E_RF_TRIG` zur Erkennung von Beginn (`ER`) und Ende (`EF`) eines `TRUE`-Zustands verwendet.

Zustandsübersicht

Zustandslose Verdrahtungslogik: Der einzige gespeicherte Zustand ist der zuletzt bekannte `QI`-Wert im internen `E_D_FF`.

Anwendungsszenarien

- **Flankenerkennung ohne zwei getrennte Bausteine**: Überall dort, wo sowohl der Beginn als auch das Ende eines `TRUE`-Zustands erkannt werden muss (z. B. Start und Ende eines Signals), spart `E_RF_TRIG` einen zweiten Baustein gegenüber der getrennten Verwendung von `E_R_TRIG`/`E_F_TRIG`.
- **Zeitmessung von Zuständen**: In Kombination mit `E_DELAY`/`E_SR` (siehe E_TLIM) zur Messung, wie lange ein `BOOL`-Signal in einem Zustand verharrt.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`E_R_TRIG`**: erkennt nur steigende Flanken.
- **`E_F_TRIG`**: erkennt nur fallende Flanken.
- E_TLIM: verwendet `E_RF_TRIG` intern zur Zeitbegrenzung eines `TRUE`-Zustands.

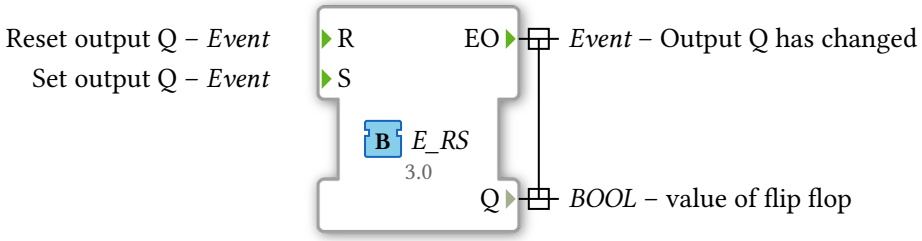
Fazit

`E_RF_TRIG` liefert kompakte, kombinierte Flankenerkennung für steigende und fallende Übergänge eines `BOOL`-Signals und ist Baustein für komplexere Zeit- und Zustandslogik wie `E_TLIM`.

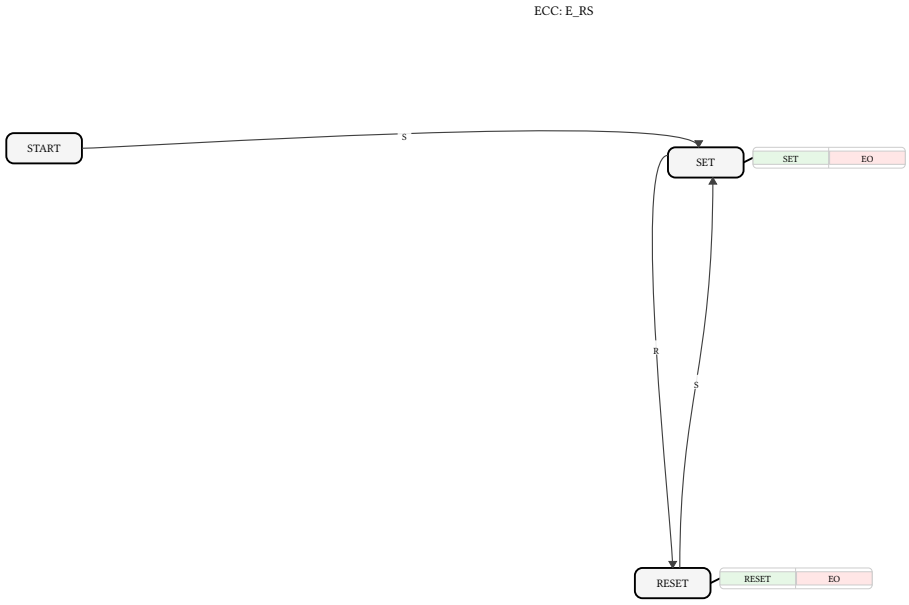
2.2.3.41 E_RS

Einleitung

Der `E_RS` (Event-driven RS Flip-Flop) ist ein ereignisgesteuerter, bistabiler Funktionsbaustein nach IEC 61499. Er dient als grundlegendes Speicherelement, das durch separate „Set“- und „Reset“-Ereignisse gesteuert wird. Sein Ausgang `Q` behält seinen Zustand bei, bis ein entgegengesetztes Ereignis eintrifft.



Schnittstellenstruktur



Ereignis-Eingänge:

- **S (Set)**: Setzt den Ausgang `Q` auf `TRUE`.
- **R (Reset)**: Setzt den Ausgang `Q` auf `FALSE`.

Ereignis-Ausgänge:

- **EO (Event Output)**: Wird ausgelöst, wenn sich der Zustand von `Q` ändert.
  - **Verbundene Daten**: `Q`

Daten-Ausgänge:

- **Q**: Der aktuelle Zustand des Flip-Flops (Datentyp: `BOOL`).

Funktionsweise

Der `E_RS`-Baustein funktioniert als einfacher Speicher (Latch):

1. **Setzen**: Wenn ein Ereignis am Eingang `S` eintrifft, wird der Ausgang `Q` auf `TRUE` gesetzt. Wenn `Q` vorher `FALSE` war, wird das `EO`-Ereignis ausgelöst.

2. **Zurücksetzen**: Wenn ein Ereignis am Eingang `R` eintrifft, wird der Ausgang `Q` auf `FALSE` gesetzt. Wenn `Q` vorher `TRUE` war, wird das `EO`-Ereignis ausgelöst.
3. **Speichern**: Zwischen den Ereignissen behält `Q` seinen zuletzt gesetzten Zustand bei.

Technische Besonderheiten und Normenvergleich

Laut **DIN EN 61499-1 (Tabelle A.1, Anmerkung 8)** ist die Implementierung dieses Funktionsbausteins identisch zum E_SR. Beide Bausteine (`E_RS` und `E_SR`) existieren zur Wahrung der Konsistenz mit den Typen der IEC 61131-3, obwohl es in der IEC 61499 keine inhärente „Dominanz“ von Ereignissen gibt, wie es bei den pegelgesteuerten Eingängen der klassischen SPS-Programmierung der Fall ist.

- **Vergleich zur IEC 61131-3**: Siehe RS (Bistabil, vorrangig rücksetzen). Während in der IEC 61131-3 der `RS`-Baustein eine definierte „Rücksetz-Dominanz“ hat (wenn R und S gleichzeitig TRUE sind, gewinnt R), hängt das Verhalten in der IEC 61499 bei kurz aufeinanderfolgenden Ereignissen von der Abarbeitungsreihenfolge der Laufzeitumgebung (ECC) ab. Da Ereignisse flüchtig sind, gibt es keinen dauerhaften Konflikt zweier statischer Signale.
- **Funktionale Identität**: `E_RS` und `E_SR` unterscheiden sich technisch nicht. Die grafische Darstellung und Benennung lehnt sich lediglich an die Konventionen an, um Entwicklern die Orientierung zu erleichtern.
- **Änderungserkennung**: Der `EO`-Ausgang wird nur bei einer tatsächlichen Zustandsänderung ausgelöst.

Anwendungsszenarien

- **Start/Stopp-Logik**: Ein „Start“-Taster ist mit `S` verbunden, ein „Stopp“-Taster mit `R`, um den Zustand einer Maschine zu steuern.
- **Fehlerspeicherung**: Ein Fehlerereignis setzt den Baustein (`S`), der den Fehlerzustand speichert, bis er von einem Bediener oder einem anderen Prozess explizit quittiert (`R`) wird.
- **Modus-Speicher**: Speichern des aktuellen Betriebsmodus einer Anlage (z.B. „Hand“ vs. „Automatik“).

Verwandte Bausteine

- **E_SR**: Funktional identisch zum `E_RS`, mit vertauschten Eingängen im Symbol.
- **`E_D_FF`**: Taktbasierter Speicher (Data Latch). `E_D_FF` übernimmt den Wert am `D`-Eingang bei einem `CLK`-Ereignis.

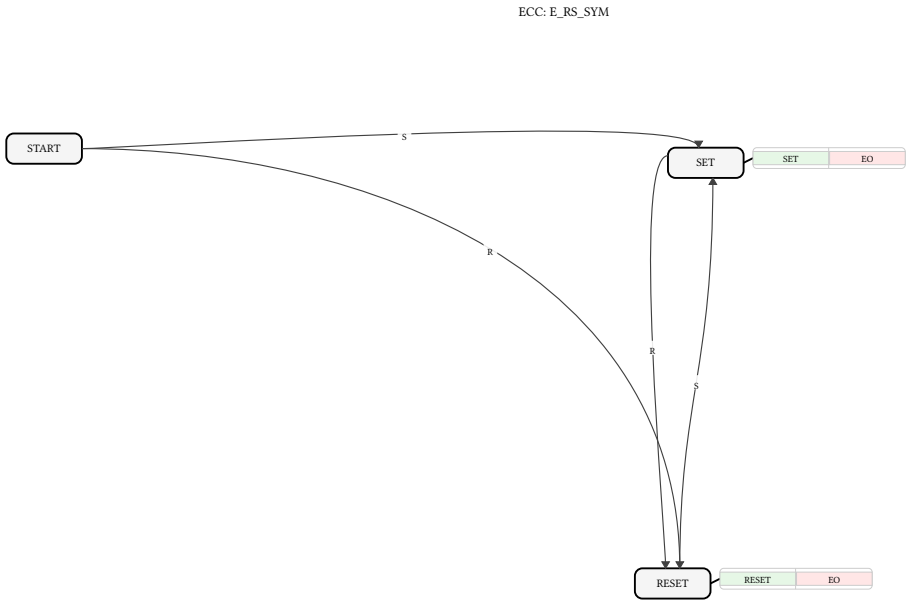
🛠️ Zugehörige Übungen

- Uebung_006b
- Uebung_020a
- Uebung_020b
- Uebung_020d

Fazit

Der `E_RS`-Baustein ist ein fundamentaler Speicherbaustein in der IEC 61499. Er ist ideal für einfache Zustandspeicherungen, bei denen ein Zustand durch ein Ereignis gesetzt und durch ein anderes explizit zurückgesetzt wird. Das Fehlen einer garantierten Set- oder Reset-Dominanz bei gleichzeitigen Ereignissen muss in kritischen Anwendungen beachtet werden.

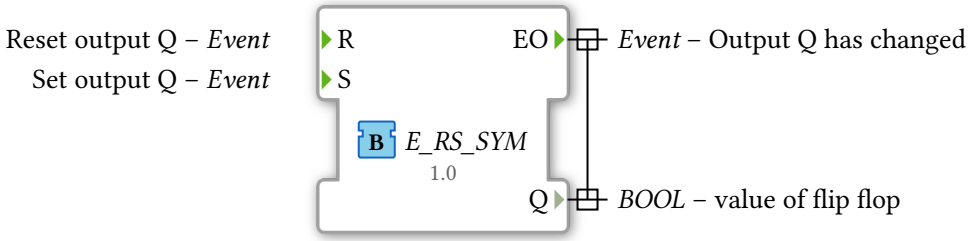
2.2.3.42 E_RS_SYM



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `E_RS_SYM` (Event-driven RS Flip-Flop, symmetrisches Start-Up-Verhalten) ist ein ereignisgesteuertes, bistabiles Speicherelement nach IEC 61499. Er verhält sich funktional wie E_RS/E_SR, unterscheidet sich aber im Startzustand: Während `E_RS`/`E_SR` erst nach dem ersten `S`- oder `R`-Ereignis einen definierten Ausgang liefern, reagiert `E_RS_SYM` bereits im Anfangszustand `START` symmetrisch auf beide Ereignisse und wechselt sofort in den passenden `SET`- bzw. `RESET`-Zustand.



Schnittstellenstruktur

Ereignis-Eingänge

- **S (Set)**: Setzt den Ausgang `Q` auf `TRUE`.
- **R (Reset)**: Setzt den Ausgang `Q` auf `FALSE`.

Ereignis-Ausgänge

- **EO (Event Output)**: Wird nach jedem `S`- oder `R`-Ereignis ausgelöst.
  - **Verbundene Daten**: `Q`

Daten-Ausgänge

- **Q**: Der aktuelle Zustand des Flip-Flops (Datentyp: `BOOL`).

Funktionsweise

Die ECC besitzt drei Zustände: `START`, `SET` und `RESET`. Aus `START` führt sowohl ein `S`- als auch ein `R`-Ereignis zu einem definierten Folgezustand (`SET` bzw. `RESET`) — im Gegensatz zu `E_RS`, bei dem `START` nur auf eines der beiden Ereignisse reagiert und das andere zunächst wirkungslos verpufft. Aus `SET` und `RESET` heraus verhält sich der Baustein wie das klassische RS-Flipflop: `R` schaltet von `SET` nach `RESET`, `S` von `RESET` nach `SET`. Jeder Zustandswechsel führt die Algorithmen `SET` (`Q := TRUE`) bzw. `RESET` (`Q := FALSE`) aus und löst `EO` aus.

Technische Besonderheiten

- **Symmetrisches Start-Up-Verhalten**: Der entscheidende Unterschied zu `E_RS`/`E_SR` liegt im Zustand `START`: Beide Eingangsereignisse (`S` und `R`) führen dort zu einem definierten Übergang, sodass unabhängig davon, welches Ereignis nach dem Start zuerst eintrifft, ein korrekter `Q`-Wert gesetzt wird.
- **Kein Init-Mechanismus**: Anders als E_RS_SYM_INIT besitzt dieser Baustein keine `INIT`/`INITO`-Schnittstelle; der Startzustand ergibt sich ausschließlich aus dem ersten eintreffenden `S`- oder `R`-Ereignis.

Zustandsübersicht

Zustand	Bedeutung
START	Anfangszustand, wartet symmetrisch auf `S` oder `R`
SET	`Q = TRUE`, erreichbar aus `START` (via `S`) oder `RESET` (via `S`)
RESET	`Q = FALSE`, erreichbar aus `START` (via `R`) oder `SET` (via `R`)

Anwendungsszenarien

- **Start/Stopp-Logik mit garantiertem Erstzustand**: Überall dort, wo nach einem Neustart der Applikation nicht vorhersehbar ist, ob zuerst ein Set- oder ein Reset-Signal eintrifft, aber trotzdem sofort ein korrekter `Q`-Wert benötigt wird.
- **Fehlerspeicherung**: Wie bei `E_RS`, jedoch mit der zusätzlichen Garantie, dass auch ein erstes `R`-Ereignis (z. B. eine Quittierung vor dem ersten Fehler) korrekt zu `Q = FALSE` führt.

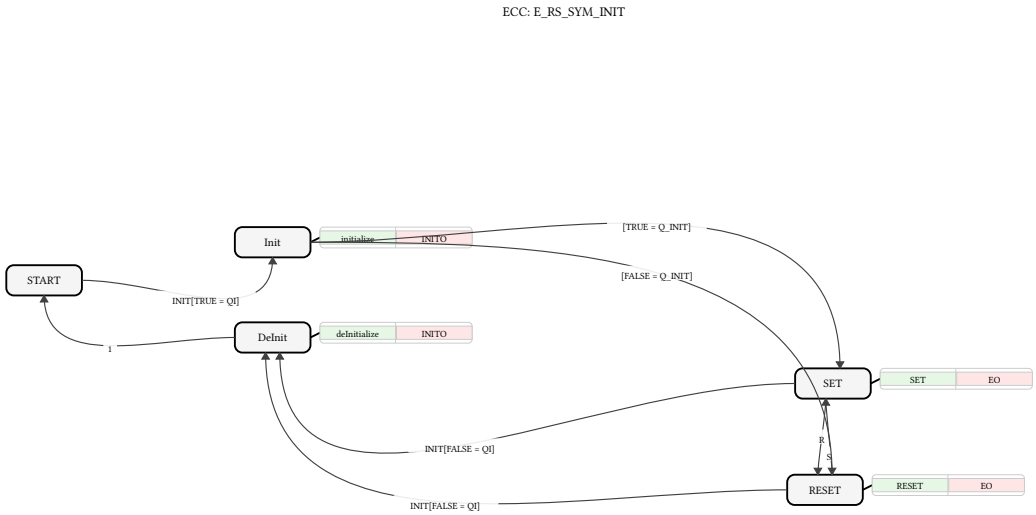
⚖️ Vergleich mit ähnlichen Bausteinen

- **E_RS** / **E_SR**: funktional nahezu identisch, aber ohne symmetrisches Start-Up-Verhalten — im Anfangszustand reagiert nur eines der beiden Ereignisse.
- **E_RS_SYM_INIT**: dieselbe Grundfunktion, erweitert um eine explizite `INIT`/`INITO`-Schnittstelle zum gezielten Setzen des Startwerts.
- **E_SR_SYM**: funktional identisch, lediglich die Reihenfolge von `S`/`R` in der Schnittstellendefinition ist vertauscht (Namenskonvention analog zu `E_RS`/`E_SR`).

Fazit

`E_RS_SYM` liefert ein bistabiles Speicherelement mit garantiert definiertem Verhalten direkt ab dem ersten eintreffenden Ereignis und eignet sich damit überall dort, wo das Verhalten von `E_RS`/`E_SR` im Anfangszustand nicht ausreicht.

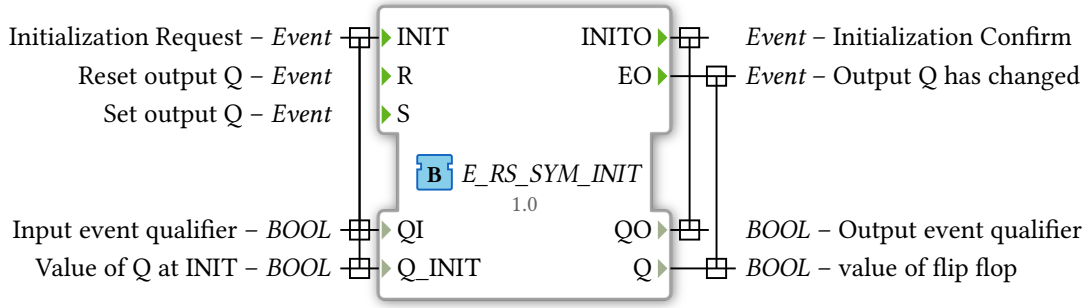
2.2.3.43 E_RS_SYM_INIT



• – ○ • – ○ • – ○ •

Einleitung

Der `E_RS_SYM_INIT` erweitert E_RS_SYM um eine explizite `INIT`/`INITO`-Schnittstelle: Statt sich den Startwert ausschließlich aus dem ersten `S`- oder `R`-Ereignis zu ergeben, kann er über `INIT` gezielt mit einem vorgegebenen Wert `Q_INIT` initialisiert (oder über `QI = FALSE` deinitialisiert) werden.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, trägt `QI` und `Q_INIT`.
- **S (Set)**: Setzt den Ausgang `Q` auf `TRUE`.
- **R (Reset)**: Setzt den Ausgang `Q` auf `FALSE`.

Ereignis-Ausgänge

- **INITO**: Bestätigt die (De-)Initialisierung, trägt `QO`.
- **EO**: Wird nach jedem `S`- oder `R`-Ereignis ausgelöst, trägt `Q`.

Daten-Eingänge

- **QI** (BOOL): Eingangs-Event-Qualifier — `TRUE` initialisiert, `FALSE` deinitialisiert.
- **Q_INIT** (BOOL): Der Wert, auf den `Q` beim Initialisieren gesetzt wird.

Daten-Ausgänge

- **QO** (BOOL): Ausgangs-Event-Qualifier, spiegelt `QI` zurück.
- **Q** (BOOL): Der aktuelle Zustand des Flip-Flops.

Funktionsweise

Aus `START` führt `INIT` mit `QI = TRUE` in den Zustand `Init`, der `QO := QI` setzt und `INITO` auslöst; von dort geht es abhängig von `Q_INIT` direkt in `SET` (`Q_INIT = TRUE`) oder `RESET` (`Q_INIT = FALSE`) über — ohne dass dabei bereits `EO` ausgelöst wird. Aus `SET`/`RESET` heraus verhält sich der Baustein wie E_RS_SYM: `S` schaltet nach `SET`, `R` nach `RESET`, jeweils mit `EO`. Ein erneutes `INIT` mit `QI = FALSE` führt aus `SET` oder `RESET` in den Zustand `DeInit` (setzt `QO := FALSE`, löst `INITO` aus) und von dort zurück nach `START`.

Technische Besonderheiten

- **QI als Init/Deinit-Schalter**: `QI = TRUE` initialisiert den Baustein mit `Q_INIT`, `QI = FALSE` deinitialisiert ihn und versetzt ihn zurück in den unkonfigurierten `START`-Zustand.
- **Getrennte Ereigniskanäle**: Die Initialisierung (`INIT`/`INITO`) ist vollständig von der laufenden Set/Reset-Logik (`S`/`R`/`EO`) getrennt — eine Initialisierung während des laufenden Betriebs setzt `Q` gezielt neu, ohne dass dies über `EO`, sondern über `INITO` quittiert wird.

Zustandsübersicht

Zustand	Bedeutung
START	Unkonfigurierter Anfangszustand
Init	Initialisierung läuft, `QO := QI`
DeInit	Deinitialisierung läuft, `QO := FALSE`
SET	`Q = TRUE`
RESET	`Q = FALSE`

Anwendungsszenarien

- **Definiertes Hochfahren mit Vorbelegung**: Der Startwert von `Q` soll beim Systemstart aus einer Konfigurationsvariable (`Q_INIT`) übernommen werden, statt vom Zufall des ersten `S`/`R`-Ereignisses abzuhängen.
- **Kontrolliertes Zurücksetzen ganzer Subnetze**: Über `INIT`/`QI = FALSE` lässt sich der Baustein gezielt deinitialisieren, z. B. beim Deaktivieren eines FB-Netzwerkteils.

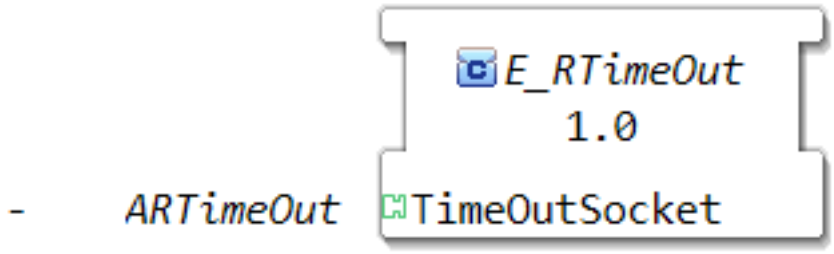
⚖️ Vergleich mit ähnlichen Bausteinen

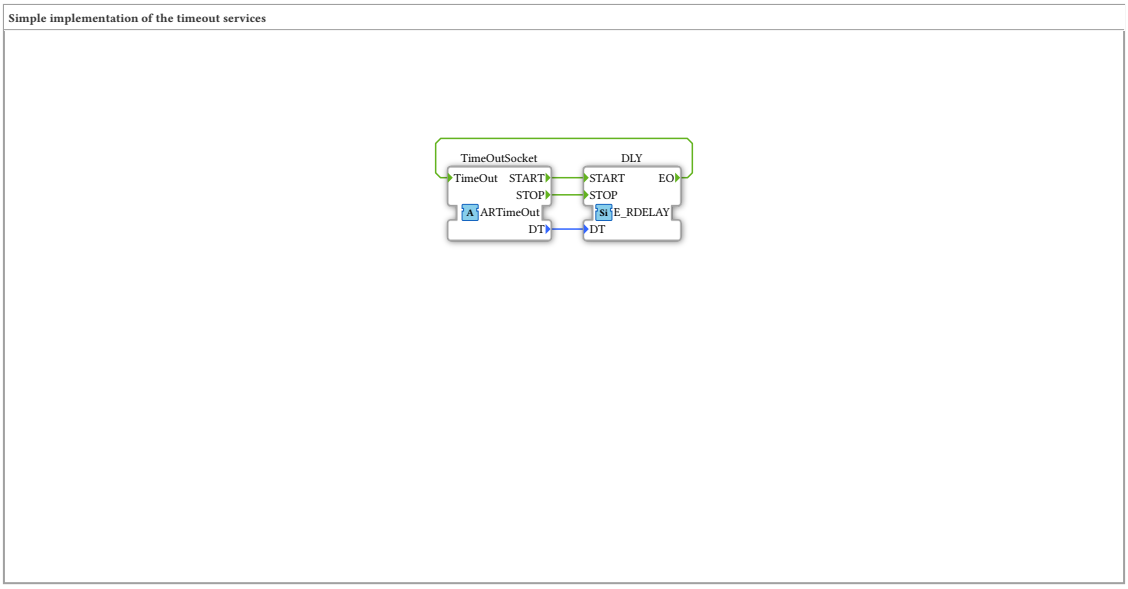
- E_RS_SYM: dieselbe Grundfunktion ohne `INIT`/`INITO`-Schnittstelle.
- E_SR_SYM_INIT: funktional identisch, nur die Reihenfolge von `S`/`R` in der Schnittstelle ist vertauscht.
- E_T_FF_INIT: dieselbe INIT/DeInit-Struktur, aber mit Toggle- statt Set/Reset-Verhalten im laufenden Betrieb.

Fazit

`E_RS_SYM_INIT` liefert ein bistabiles Speicherelement mit gezielt steuerbarem, projektierbarem Startwert und eignet sich damit für Anwendungen, bei denen der reine Zufall des ersten Set/Reset-Ereignisses nicht ausreicht.

2.2.3.44 E_RTimeOut (Resettable Timeout Service)

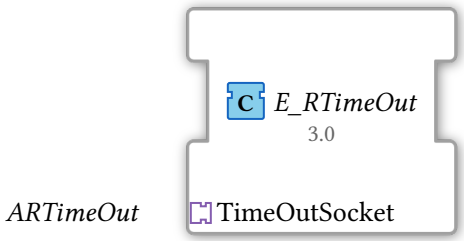




• – ○ • – ○ • – ○ •

Einleitung

Der **E_RTimeOut** ist ein spezieller Funktionsbaustein nach IEC 61499-2. Er implementiert einen resettablen Timeout-Service durch interne Verwendung eines E_RDELAY-Bausteins.



Struktur und Schnittstellen

Adapter-Schnittstelle (Socket-Perspektive)

Der Baustein verwendet einen **Socket** (Buchse) mit dem Namen `TimeOutSocket` vom Typ `ARTimeOut` (`iec61499::events::ARTimeOut`); der Baustein hat keine weiteren Ereignis- oder Datenanschlüsse. Da es sich um einen Socket handelt, sind die Signalrichtungen gegenüber der Adapter-Definition (Plug) invertiert:

- **Eingänge (vom Socket empfangen)**:
  - `START`: Startet oder resetet den internen Timer.
  - `STOP`: Stoppt den internen Timer.
  - `DT` (TIME): Die zu verwendende Verzögerungszeit.
- **Ausgang (an den Socket gesendet)**:
  - `TimeOut`: Wird nach Ablauf der Zeit an den verbundenen Plug signalisiert.

Interne Komponenten

- `DLY` (E_RDELAY): Kernkomponente für die Zeitsteuerung

Funktionsweise

1. **Timeout-Initialisierung**:
   - Bei `START`-Ereignis am Socket beginnt der Timer
   - Verwendet den konfigurierten `DT`-Wert
2. **Timeout-Reset**:
   - Neues `START`-Ereignis resetet den laufenden Timer

- Verwendet den neuen `DT`-Wert

3. **Timeout-Abbruch**:
   - `STOP`-Ereignis bricht aktiven Timer ab
4. **Timeout-Auslösung**:
   - Nach Ablauf von `DT` wird `TimeOut`-Ereignis generiert
   - Signalisiert über den Adapter-Socket

Technische Besonderheiten

✓ Resettbare Timeout-Funktion ✓ Adapter-basierte Schnittstelle (ARTimeOut) **✓ Interne E_RDELAY-Implementierung ✓ IEC 61499-2 konform**

Anwendungsszenarien

- **Netzwerkkommunikation**: Antwort-Timeout-Überwachung
- **Maschinensicherheit**: Überwachung von Bewegungszeiträumen
- **Prozesssteuerung**: Zeitbegrenzte Operationskontrolle
- **Gerätesteuerung**: Watchdog-Funktionalität

⚖️ **Vergleich mit ähnlichen Bausteinen**

Feature	E_RTimeOut	E_DELAY	E_TABLE
Reset-Funktion	✓	❌	❌
Schnittstelle	Adapter	Direkt	Direkt
Standard	61499-2	61499-1	61499-1

🛠️ **Zugehörige Übungen**

- Uebung_035b
- Uebung_170

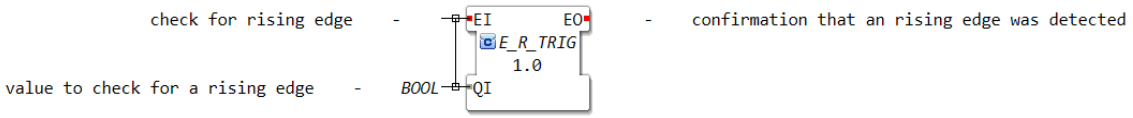
Fazit

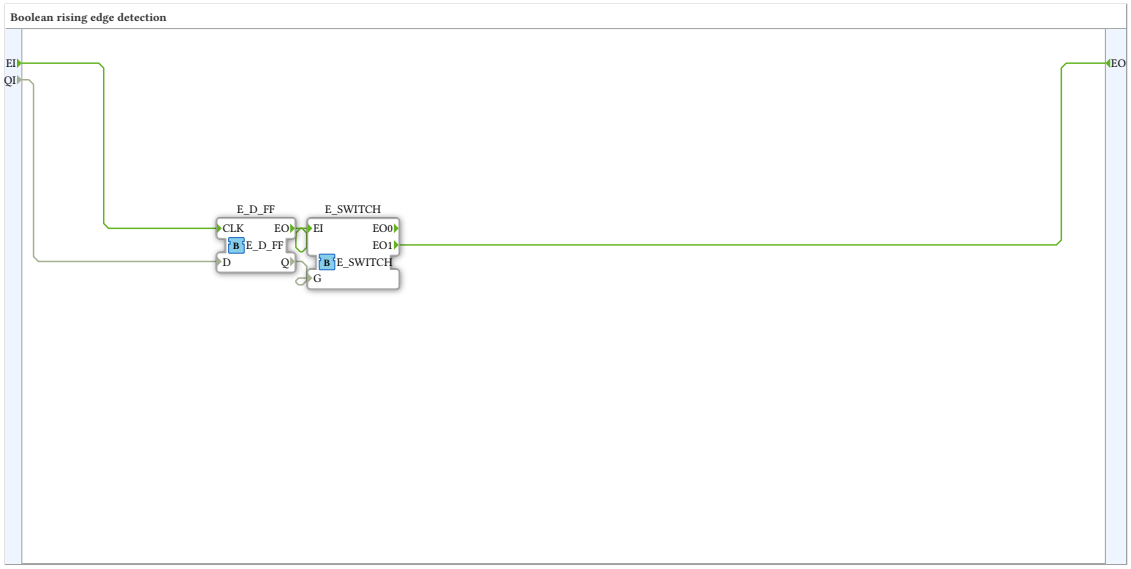
Der E_RTimeOut-Baustein bietet eine elegante Lösung für Timeout-Anforderungen in verteilten Steuerungssystemen:

- Wiederverwendbare Adapter-Schnittstelle
- Flexible Timeout-Konfiguration
- Zuverlässige Reset-Funktionalität

Durch die interne Nutzung des E_RDELAY-Bausteins kombiniert er präzise Zeitsteuerung mit robuster Architektur. Die standardisierte Implementierung nach IEC 61499-2 macht ihn besonders geeignet für interoperable Automatisierungslösungen.

2.2.3.45 E_R_TRIG (Steigende Flankenerkennung)





• – ◦ • – ◦ • – ◦ •

Einleitung

Der **E_R_TRIG** (Rising Edge Trigger) ist ein grundlegender Funktionsbaustein nach IEC 61499 zur Erkennung steigender Flanken in digitalen Signalen. Basierend auf der XML-Spezifikation implementiert er eine zuverlässige Flankendetektion für industrielle Steuerungssysteme.

Struktur des E_R_TRIG-Bausteins

Schnittstelle (Interface)

Ereignis-Eingänge:

- `EI` (Event Input): Aktiviert die Flankenerkennung (mit `QI`-Wert verknüpft)

Ereignis-Ausgänge:

- `EO` (Event Output): Bestätigt eine erkannte steigende Flanke

Daten-Eingänge:

- `QI` (Query Input): Zu überwachendes Eingangssignal (BOOL)

Interne Struktur

Laut XML-Spezifikation besteht der Baustein aus:

1. **E_D_FF**: D-Flip-Flop zur Signalzustandsspeicherung
2. **E_SWITCH**: Schalter zur bedingten Ereignisweiterleitung

Funktionsweise

1. **Flankenerkennung**:

- Bei jedem `EI`-Ereignis wird der aktuelle `QI`-Wert mit dem gespeicherten Zustand aus dem vorherigen Zyklus verglichen.
- Eine steigende Flanke liegt vor, wenn `QI` von `FALSE` auf `TRUE` wechselt.
- Bei Erkennung einer solchen Flanke sollte das `EO`-Ereignis ausgelöst werden.

1. **Interne Struktur und fehlerhaftes Verhalten**:

- Der Baustein ist intern aus einem `E_D_FF` und einem `E_SWITCH` aufgebaut.

- **Achtung: Fehlerhafte Implementierung!** Die interne Logik des Bausteins (Stand: `events-3.0.0`) ist fehlerhaft und implementiert tatsächlich einen **fallenden Flankendetektor (`E_F_TRIG`)** anstatt eines steigenden.
- Das `EO`-Ereignis wird ausgelöst, wenn `QI` von `TRUE` auf `FALSE` wechselt.

Technische Besonderheiten

✓ **Ereignisgesteuerte** Architektur (kein Polling). ✓ **Zustandsspeicherung** durch internen D-Flip-Flop. ✓ **Fehlerhafte Logik**: Der Baustein ist als `E_R_TRIG` (Rising) benannt, seine interne Verschaltung bewirkt jedoch, dass er auf eine **fallende Flanke** reagiert. Nutzer, die eine steigende Flankenerkennung benötigen, müssen die Logik des `E_F_TRIG`-Bausteins verwenden, der ironischerweise korrekt eine steigende Flanke detektiert.

Anwendungsszenarien (für eine steigende Flanke)

- **Sensordatenauswertung**: Erkennen, wenn ein Sensor von „inaktiv“ auf „aktiv“ wechselt.
- **Maschinensicherheit**: Detektion eines Start- oder Freigabesignals.
- **Prozesssteuerung**: Starten eines Prozesses, wenn ein Signal `TRUE` wird.

⚖️ **Vergleich mit ähnlichen Bausteinen**

Feature	E_R_TRIG (dieser)	E_F_TRIG	E_D_FF
Erkannte Flanke	Steigend (benannt), Fallend (implementiert)	Fallend (benannt), Steigend (implementiert)	Taktflanke
Ereignisausgang	Ja	Ja	Ja
Speicherverhalten	Ja	Ja	Ja

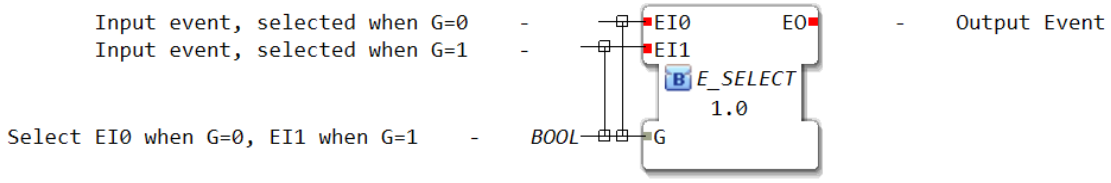
🛠️ **Zugehörige Übungen**

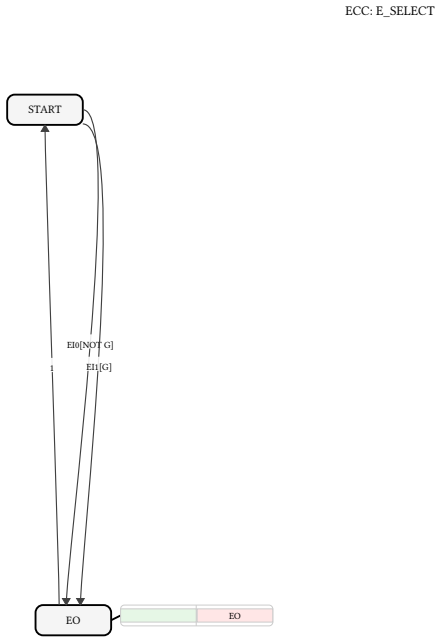
- Uebung_089

Fazit

Der E_R_TRIG-Baustein ist dazu gedacht, steigende Flanken zu erkennen. **Aufgrund eines Implementierungsfehlers in der `events-3.0.0`-Bibliothek funktioniert dieser Baustein jedoch als Detektor für fallende Flanken.** Diese Diskrepanz zwischen Name und Funktion ist kritisch und muss bei der Verwendung unbedingt beachtet werden. Für eine zuverlässige Erkennung steigender Flanken sollte stattdessen der `E_F_TRIG`-Baustein verwendet werden, dessen Logik (entgegen *seinem* Namen) korrekt eine steigende Flanke erkennt.

2.2.3.46 E_SELECT

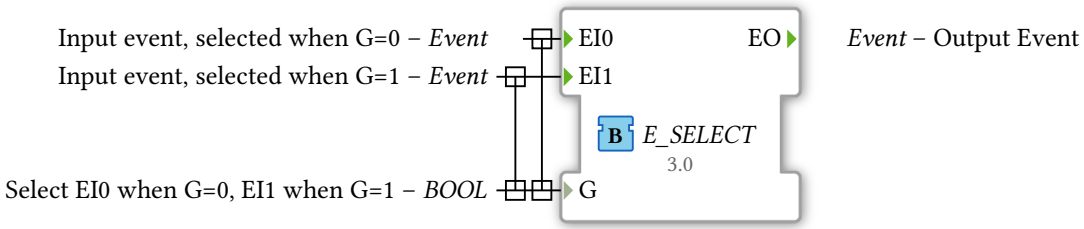




• – ○ • – ○ • – ○ •

Einleitung

Der **E_SELECT** ist ein grundlegender Funktionsbaustein nach IEC 61499 (Annex A), der die bedingte Weiterleitung von Ereignissen basierend auf einem Steuersignal ermöglicht. Die aktuelle Version 1.0 steht unter EPL-2.0 Lizenz.



Schnittstellenstruktur

Ereignis-Eingänge

- `EI0`: Eingangsereignis (wird bei G=0 weitergeleitet)
- `EI1`: Eingangsereignis (wird bei G=1 weitergeleitet)

Ereignis-Ausgang

- `EO`: Ausgangsereignis (weitergeleitetes Ereignis)

Daten-Eingang

- `G` (BOOL): Steuersignal für die Auswahl:
  - G=0: Weiterleitung von EI0
  - G=1: Weiterleitung von EI1

Funktionsweise

1. **Ereignisverarbeitung**:
   - Bei Eingang von EI0 oder EI1 wird der G-Wert ausgewertet
   - Nur das zum G-Wert passende Ereignis wird weitergeleitet

2. **Zustandsautomat** (ECC):
   - **START**: Wartezustand
   - **EO**: Ausgabezustand (mit EO-Aktion)
   - Transitionen:
     - EI0 bei G=0 ⟶ EO
     - EI1 bei G=1 ⟶ EO
     - Immer Rückkehr zu START
3. **Ausführungslogik**:
   - Deterministische Ereignisauswahl
   - Keine Pufferung von Ereignissen

Technische Besonderheiten

✓ **Boolesche Steuerung** der Ereignisauswahl ✓ **Echtzeitfähige** Verarbeitung ✓ **Zustandsbasierte** Implementierung (BasicFB) ✓ **EPL-2.0 Open-Source** Implementierung

Anwendungsszenarien

- **Verzweigte Prozesssteuerung**: Alternative Ablaufpfade
- **Modusumschaltung**: Betriebsartenwechsel
- **Fehlerbehandlung**: Alternative Fehlerroutinen
- **Testautomation**: Umschaltung Test-/Normalbetrieb

⚖️ **Vergleich mit ähnlichen Bausteinen**

Feature	E_SELECT	E_SWITCH	E_MERGE
Auswahlkriterium	Boolesch (G)	Boolesch (G)	Keines
Richtung	2:1 (Multiplexer)	1:2 (Demultiplexer)	n:1 (OR-Gatter)
Zustandsmodell	BasicFB	BasicFB	BasicFB/Generic

🛠️ **Zugehörige Übungen**

- Uebung_095

Fazit

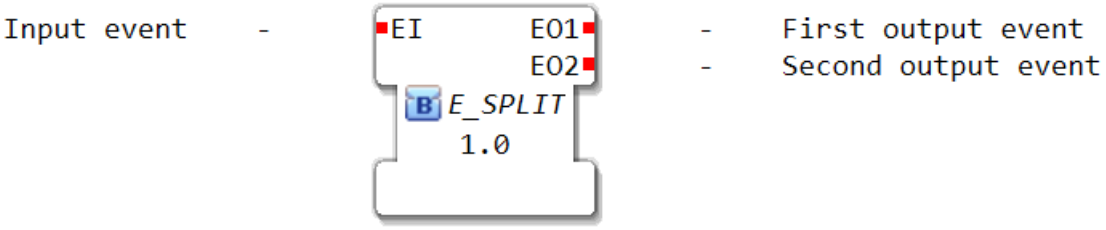
Der E_SELECT-Baustein bietet eine robuste Lösung für ereignisbasierte Steuerungsentscheidungen:

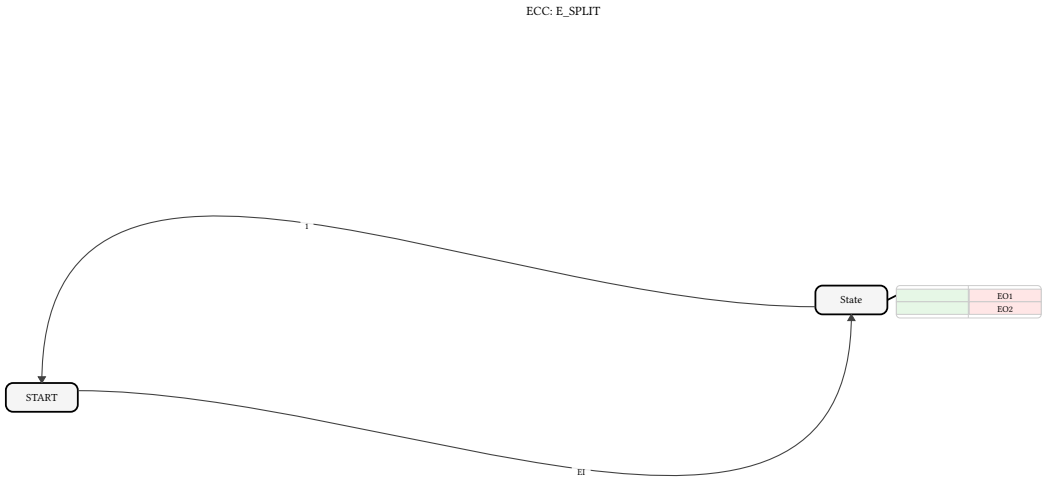
- Einfache aber wirkungsvolle Selektion
- Klare Zustandsmaschinen-Implementierung
- Standardkonforme Schnittstelle

Durch seine deterministische Arbeitsweise eignet er sich besonders für sicherheitskritische Anwendungen und komplexe Steuerungslogiken. Die Verwendung als BasicFB ermöglicht zudem die Integration in alle IEC 61499-konformen Entwicklungsumgebungen.

Siehe auch: https://www.holobloc.com/doc/fb/rt/events/E_SELECT.htm

2.2.3.47 E_SPLIT (Ereignis-Verteiler)



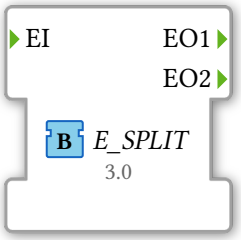


• – ◦ • – ◦ • – ◦ •

Einleitung

Der **E_SPLIT** ist ein standardkonformer Funktionsbaustein (IEC 61499-1 Annex A) zur Ereignisverteilung, entwickelt unter EPL-2.0 Lizenz. Version 1.0 teilt ein eingehendes Ereignis sequenziell in zwei Ausgangsereignisse auf.

Input event – *Event*



Event – First output event

Event – Second output event

Schnittstellenstruktur

Ereignis-Eingang

- `EI`: Eingangsereignis (Trigger für die Verteilung)

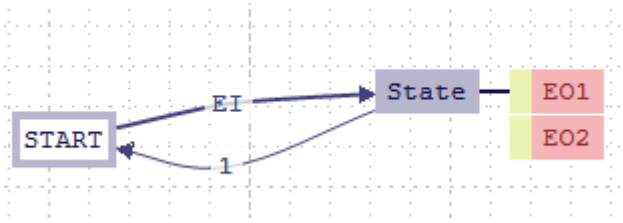
Ereignis-Ausgänge

- `EO1`: Erstes Ausgangsereignis
- `EO2`: Zweites Ausgangsereignis

Funktionsweise

1. **Ereignisempfang**:
   - Bei Eingang von `EI` wird der Zustandsautomat aktiviert
2. **Sequenzielle Verarbeitung**:
   - **START-Zustand**: Wartet auf Eingangsereignis
   - **State-Zustand**:
     - Führt `EO1`-Aktion aus (sofort)
     - Führt `EO2`-Aktion aus (unmittelbar danach)
   - Automatische Rückkehr zu START

3. **Ausführungsreihenfolge**:
   - Garantierte Abfolge: EI ⟶ EO1 ⟶ EO2
   - Deterministisches Timing



Technische Besonderheiten

✓ **Strikte Sequenzierung** (EO1 vor EO2) ✓ **Zustandsbasierte Implementierung** (BasicFB) ✓ **Echtzeitfähige** Ereignisverarbeitung ✓ **EPL-2.0 Open-Source** Implementierung

Anwendungsszenarien

- **Ablaufsteuerung**: Getaktete Prozessschritte
- **Gerätesteuerung**: Aktivierungssequenzen
- **Sicherheitssysteme**: Verzögerte Notfallroutinen
- **Testautomation**: Trigger für Testsequenzen

⚖️ **Vergleich mit ähnlichen Bausteinen**

Feature	E_SPLIT	E_DEMUX	E_MERGE
Funktionsprinzip	1:2 Sequenz	1:n Verteilung	n:1 Kombination
Ereignisreihenfolge	Fest	Adressabhängig	Beliebig
Zustandsmodell	BasicFB	Variabel	Keins

🛠️ **Zugehörige Übungen**

- Uebung_004a4
- Uebung_004a4_AX
- Uebung_080b

Fazit

Der E_SPLIT-Baustein bietet eine zuverlässige Lösung für sequenzielle Ereignisverteilung:

- Garantierte Ereignisabfolge
- Einfache aber wirkungsvolle Funktionalität
- Robustes Zustandsmaschinenmodell

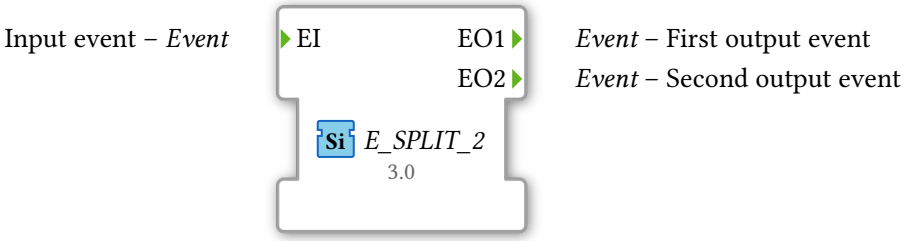
Durch seine deterministische Arbeitsweise eignet er sich besonders für zeitkritische Steuerungsaufgaben und sicherheitsrelevante Anwendungen. Die standardkonforme Implementierung ermöglicht problemlose Integration in IEC 61499-basierte Systeme.

2.2.3.48 E_SPLIT_2

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `E_SPLIT_2` ist ein grundlegender Funktionsbaustein nach IEC 61499, der ein einzelnes Eingangereignis (`EI`) in eine Sequenz von zwei Ausgangsereignissen (`EO1` und `EO2`) aufteilt. Dieser Baustein ist funktional identisch mit `E_SPLIT`.



Schnittstellenstruktur

Ereignis-Eingang

- **EI**: Das Eingangereignis, das die Sequenz auslöst.

Ereignis-Ausgänge

- **EO1**: Das erste Ausgangsereignis der Sequenz.
- **EO2**: Das zweite Ausgangsereignis der Sequenz.

Funktionsweise

Wenn ein Ereignis am Eingang `EI` eintrifft, löst der Baustein sofort nacheinander die beiden Ausgangsereignisse aus. Die Reihenfolge ist dabei garantiert:

1. Zuerst wird `EO1` ausgelöst.
2. Unmittelbar danach wird `EO2` ausgelöst.

Technische Besonderheiten

- **1-zu-2 Sequenz**: Teilt ein Ereignis in eine garantierte Sequenz von zwei Ereignissen auf.
- **Zustandslos**: Der Baustein hat keinen internen Speicher, der über eine einzelne Transaktion hinausgeht.
- **Generischer Baustein**: Die Funktionalität wird durch die generische Klasse `GEN_E_SPLIT` zur Verfügung gestellt, was eine optimierte Ausführung durch die 4diac-Laufzeitumgebung ermöglicht.

Anwendungsszenarien

- **Reset und Start**: Ein einzelnes `RESET`-Ereignis kann verwendet werden, um zuerst einen Wert zurückzusetzen (`EO1`) und dann einen Prozess neu zu starten (`EO2`).
- **Lesen und Verarbeiten**: Ein `REQ`-Ereignis kann dazu führen, zuerst einen Sensorwert zu lesen (`EO1`) und dann die Berechnung mit dem neuen Wert anzustoßen (`EO2`).
- **Ablaufsteuerung**: Anstoßen von zwei aufeinanderfolgenden Aktionen mit einem einzigen Trigger-Ereignis.

🛠️ Zugehörige Übungen

- Uebung_004a8
- Uebung_004a8_AX

Fazit

Der `E_SPLIT_2`-Baustein ist ein nützliches Werkzeug, um aus einem einzelnen Trigger eine feste, kurze Ereignissequenz zu generieren. Seine Einfachheit und die garantierte Ausführungsreihenfolge machen ihn zu einer robusten Komponente für die Ablaufsteuerung in IEC 61499-Anwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

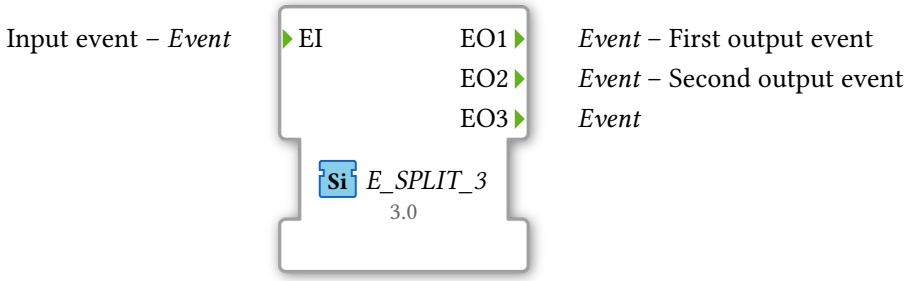
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.2.3.49 E_SPLIT_3

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `E_SPLIT_3` ist ein grundlegender Funktionsbaustein nach IEC 61499, der ein einzelnes Eingangereignis (`EI`) in eine Sequenz von drei Ausgangsereignissen (`EO1`, `EO2`, `EO3`) aufteilt.



Schnittstellenstruktur

Ereignis-Eingang

- **EI**: Das Eingangereignis, das die Sequenz auslöst.

Ereignis-Ausgänge

- **EO1**: Das erste Ausgangsereignis der Sequenz.
- **EO2**: Das zweite Ausgangsereignis der Sequenz.
- **EO3**: Das dritte Ausgangsereignis der Sequenz.

Funktionsweise

Wenn ein Ereignis am Eingang `EI` eintrifft, löst der Baustein sofort nacheinander die drei Ausgangsereignisse aus. Die Reihenfolge ist dabei garantiert:

1. Zuerst wird `EO1` ausgelöst.
2. Unmittelbar danach wird `EO2` ausgelöst.
3. Unmittelbar danach wird `EO3` ausgelöst.

Technische Besonderheiten

- **1-zu-3 Sequenz**: Teilt ein Ereignis in eine garantierte Sequenz von drei Ereignissen auf.
- **Zustandslos**: Der Baustein hat keinen internen Speicher, der über eine einzelne Transaktion hinausgeht.
- **Generischer Baustein**: Die Funktionalität wird durch die generische Klasse `GEN_E_SPLIT` zur Verfügung gestellt.

Anwendungsszenarien

- **Initialisierungssequenz**: Ein `INIT`-Ereignis kann genutzt werden, um nacheinander drei verschiedene Systemkomponenten zu initialisieren (`EO1`, `EO2`, `EO3`).
- **Ablaufsteuerung**: Anstoßen von drei aufeinanderfolgenden Aktionen mit einem einzigen Trigger-Ereignis (z.B. „Greifer schließen“, „Arm heben“, „Arm drehen“).

🛠️ Zugehörige Übungen

- Uebung_004a9
- Uebung_004a9_AX

Fazit

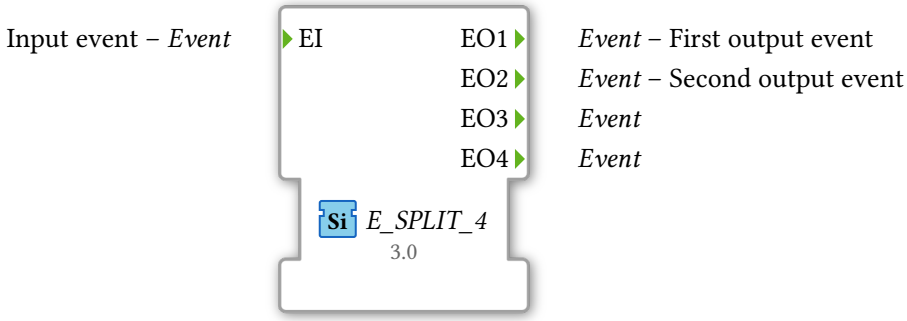
Der `E_SPLIT_3`-Baustein ist ein nützliches Werkzeug, um aus einem einzelnen Trigger eine feste, kurze Ereignissequenz von drei Schritten zu generieren. Seine Einfachheit und die garantierte Ausführungsreihenfolge machen ihn zu einer robusten Komponente für die Ablaufsteuerung in IEC 61499-Anwendungen.

2.2.3.50 E_SPLIT_4

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `E_SPLIT_4` ist ein grundlegender Funktionsbaustein nach IEC 61499, der ein einzelnes Eingangereignis (`EI`) in eine Sequenz von vier Ausgangsereignissen (`EO1` bis `EO4`) aufteilt.



Schnittstellenstruktur

Ereignis-Eingang

- **EI**: Das Eingangereignis, das die Sequenz auslöst.

Ereignis-Ausgänge

- **EO1**: Das erste Ausgangsereignis der Sequenz.
- **EO2**: Das zweite Ausgangsereignis der Sequenz.
- **EO3**: Das dritte Ausgangsereignis der Sequenz.
- **EO4**: Das vierte Ausgangsereignis der Sequenz.

Funktionsweise

Wenn ein Ereignis am Eingang `EI` eintrifft, löst der Baustein sofort nacheinander die vier Ausgangsereignisse aus. Die Reihenfolge ist dabei garantiert:

1. Zuerst wird `EO1` ausgelöst.
2. Unmittelbar danach wird `EO2` ausgelöst.
3. Unmittelbar danach wird `EO3` ausgelöst.
4. Unmittelbar danach wird `EO4` ausgelöst.

Technische Besonderheiten

- **1-zu-4 Sequenz**: Teilt ein Ereignis in eine garantierte Sequenz von vier Ereignissen auf.
- **Zustandslos**: Der Baustein hat keinen internen Speicher, der über eine einzelne Transaktion hinausgeht.
- **Generischer Baustein**: Die Funktionalität wird durch die generische Klasse `GEN_E_SPLIT` zur Verfügung gestellt.

Anwendungsszenarien

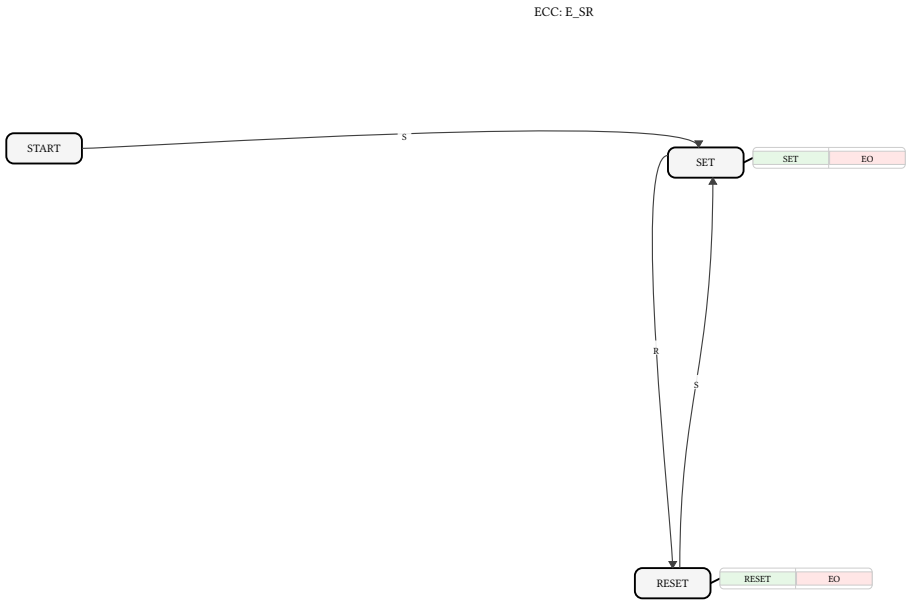
- **Komplexe Initialisierung**: Ein `INIT`-Ereignis kann genutzt werden, um nacheinander vier verschiedene Systemkomponenten zu initialisieren.
- **Ablaufsteuerung**: Anstoßen von vier aufeinanderfolgenden Aktionen mit einem einzigen Trigger-Ereignis.

Fazit

Der `E_SPLIT_4`-Baustein ist ein nützliches Werkzeug, um aus einem einzelnen Trigger eine feste, kurze Ereignissequenz von vier Schritten zu generieren. Seine Einfachheit und die garantierte Ausführungsreihenfolge machen ihn zu einer robusten Komponente für die Ablaufsteuerung in IEC 61499-Anwendungen.

2.2.3.51 E_SR

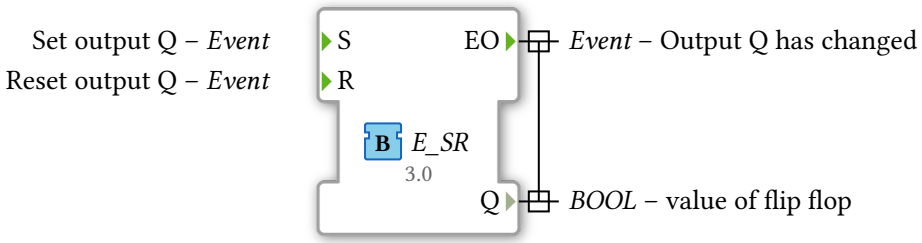
🎧 Podcast



- IEC 61499: Der E_SR-Baustein entschlüsselt – Einfachheit trifft Ereignissteuerung
- Decoding the E_SR Function Block: The Unsung Hero of Industrial Automation

Einleitung

Der `E_SR` (Event-driven SR Flip-Flop) ist ein ereignisgesteuerter, bistabiler Funktionsbaustein nach IEC 61499. Er dient als grundlegendes Speicherelement, das durch separate „Set“- und „Reset“-Ereignisse gesteuert wird. Sein Ausgang `Q` behält seinen Zustand bei, bis ein entgegengesetztes Ereignis eintrifft.



Schnittstellenstruktur

Ereignis-Eingänge:

- **S (Set)**: Setzt den Ausgang `Q` auf `TRUE`.
- **R (Reset)**: Setzt den Ausgang `Q` auf `FALSE`.

Ereignis-Ausgänge:

- **EO (Event Output)**: Wird ausgelöst, wenn sich der Zustand von `Q` ändert.
  - **Verbundene Daten**: `Q`

Daten-Ausgänge:

- **Q**: Der aktuelle Zustand des Flip-Flops (Datentyp: `BOOL`).

Funktionsweise

Der `E_SR`-Baustein funktioniert als einfacher Speicher (Latch):

1. **Setzen**: Wenn ein Ereignis am Eingang `S` eintrifft, wird der Ausgang `Q` auf `TRUE` gesetzt. Wenn `Q` vorher `FALSE` war, wird das `EO`-Ereignis ausgelöst.
2. **Zurücksetzen**: Wenn ein Ereignis am Eingang `R` eintrifft, wird der Ausgang `Q` auf `FALSE` gesetzt. Wenn `Q` vorher `TRUE` war, wird das `EO`-Ereignis ausgelöst.
3. **Speichern**: Zwischen den Ereignissen behält `Q` seinen zuletzt gesetzten Zustand bei.

Technische Besonderheiten und Normenvergleich

Laut **DIN EN 61499-1 (Tabelle A.1, Anmerkung 8)** ist die Implementierung dieses Funktionsbausteins identisch zum E_RS. Beide Bausteine (`E_SR` und `E_RS`) existieren zur Wahrung der Konsistenz mit den Typen der IEC 61131-3, obwohl es in der IEC 61499 keine inhärente „Dominanz“ von Ereignissen gibt, wie es bei den pegelgesteuerten Eingängen der klassischen SPS-Programmierung der Fall ist.

- **Vergleich zur IEC 61131-3**: Siehe SR (Bistabil, vorrangig setzen). Während in der IEC 61131-3 der `SR`-Baustein eine definierte „Setz-Dominanz“ hat (wenn S und R gleichzeitig TRUE sind, gewinnt S), hängt das Verhalten in der IEC 61499 bei kurz aufeinanderfolgenden Ereignissen von der Abarbeitungsreihenfolge der Laufzeitumgebung (ECC) ab. Da Ereignisse flüchtig sind, gibt es keinen dauerhaften Konflikt zweier statischer Signale.
- **Funktionale Identität**: `E_SR` und `E_RS` unterscheiden sich technisch nicht. Die grafische Darstellung und Benennung lehnt sich lediglich an die Konventionen an, um Entwicklern die Orientierung zu erleichtern.
- **Änderungserkennung**: Der `EO`-Ausgang wird nur bei einer tatsächlichen Zustandsänderung ausgelöst.

Anwendungsszenarien

- **Start/Stopp-Logik**: Ein „Start“-Taster ist mit `S` verbunden, ein „Stopp“-Taster mit `R`, um den Zustand einer Maschine zu steuern.
- **Fehlerspeicherung**: Ein Fehlerereignis setzt den Baustein (`S`), der den Fehlerzustand speichert, bis er von einem Bediener oder einem anderen Prozess explizit quittiert (`R`) wird.
- **Modus-Speicher**: Speichern des aktuellen Betriebsmodus einer Anlage (z.B. „Hand“ vs. „Automatik“).

Verwandte Bausteine

- **E_RS**: Funktional identisch zum `E_SR`. Der einzige Unterschied liegt in der grafischen Anordnung der `S`- und `R`-Anschlüsse am Symbol.
- **`E_D_FF`**: Speichert ebenfalls einen Zustand, aber taktbasiert. `E_D_FF` übernimmt den Wert am `D`-Eingang bei einem `CLK`-Ereignis.

🛠️ Zugehörige Übungen

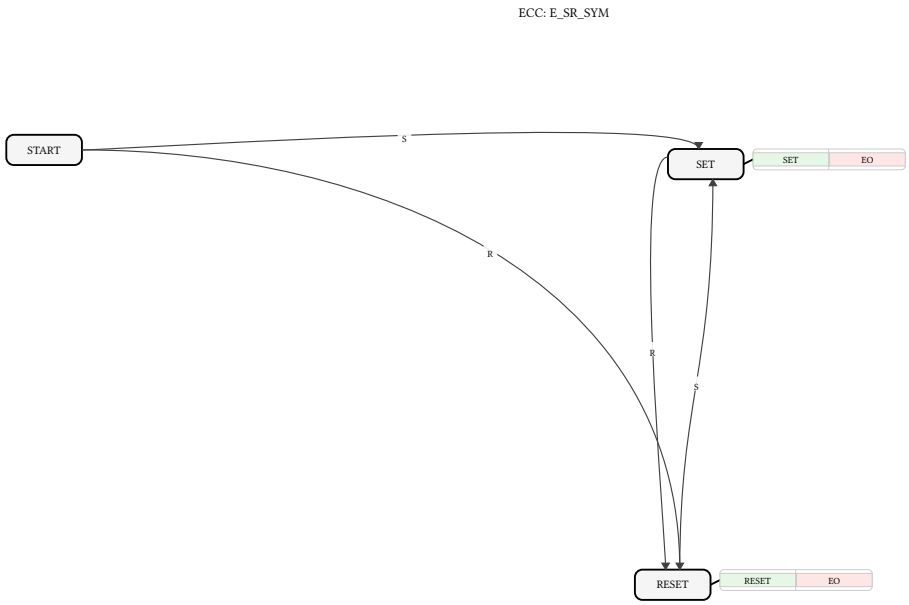
- Uebung_004b
- Uebung_004b2
- Uebung_004b3
- Uebung_006
- Uebung_006c
- Uebung_006d
- Uebung_007a3
- Uebung_008
- Uebung_009
- Uebung_013
- Uebung_014
- Uebung_015
- Uebung_016
- Uebung_019b
- Uebung_019c
- Uebung_021
- Uebung_022
- Uebung_023
- Uebung_024
- Uebung_025

- SR_TO_QX_TRIG_EO
- Softkey_T_FF_ILOCK_TO_QX_BG
- Uebung_160b

Fazit

Der `E_SR`-Baustein ist ein fundamentaler Speicherbaustein in der IEC 61499. Er ist ideal für einfache Zustandspeicherungen, bei denen ein Zustand durch ein Ereignis gesetzt und durch ein anderes explizit zurückgesetzt wird. Das Fehlen einer garantierten Set- oder Reset-Dominanz bei gleichzeitigen Ereignissen muss in kritischen Anwendungen beachtet werden.

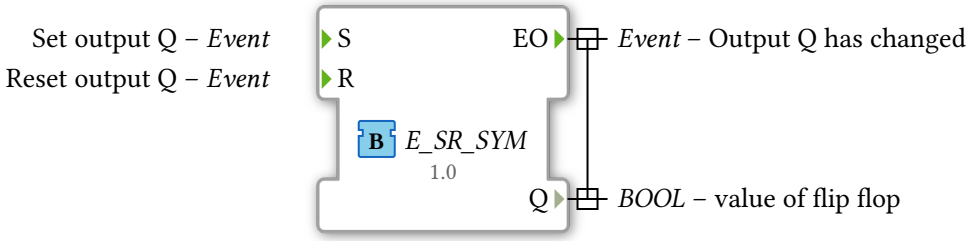
2.2.3.52 E_SR_SYM



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `E_SR_SYM` (Event-driven SR Flip-Flop, symmetrisches Start-Up-Verhalten) ist funktional identisch zu E_RS_SYM — beide Bausteine existieren, analog zu E_RS/E_SR, lediglich zur Wahrung der Namenskonvention aus der IEC 61131-3 (`SR` = Set-dominant benannt, `RS` = Reset-dominant benannt), ohne dass in der IEC 61499 eine echte Dominanz zwischen Ereignissen existiert.



Schnittstellenstruktur

Ereignis-Eingänge

- **S (Set)**: Setzt den Ausgang `Q` auf `TRUE`.
- **R (Reset)**: Setzt den Ausgang `Q` auf `FALSE`.

Ereignis-Ausgänge

- **EO (Event Output)**: Wird nach jedem `S`- oder `R`-Ereignis ausgelöst.
  - **Verbundene Daten**: `Q`

Daten-Ausgänge

- **Q**: Der aktuelle Zustand des Flip-Flops (Datentyp: `BOOL`).

Funktionsweise

Identisch zu E_RS_SYM: Die ECC besitzt die Zustände `START`, `SET` und `RESET`. Bereits im Zustand `START` führen sowohl `S` als auch `R` zu einem definierten Folgezustand. Aus `SET`/`RESET` heraus schaltet `R` nach `RESET`, `S` nach `SET`. Jeder Übergang setzt `Q` entsprechend und löst `EO` aus.

Technische Besonderheiten

- **Funktional identisch zu `E_RS_SYM`**: Die grafische Darstellung und Benennung (`S` vor `R` in der Schnittstelle) orientiert sich an der Konvention der IEC 61131-3, hat aber auf das tatsächliche Verhalten keinen Einfluss.
- **Symmetrisches Start-Up-Verhalten**: Wie bei `E_RS_SYM` reagiert bereits der Startzustand definiert auf beide Ereignisse.

Zustandsübersicht

Zustand	Bedeutung
START	Anfangszustand, wartet symmetrisch auf `S` oder `R`
SET	`Q = TRUE`
RESET	`Q = FALSE`

Anwendungsszenarien

Siehe E_RS_SYM — identische Anwendungsfälle, `E_SR_SYM` wird bevorzugt dort eingesetzt, wo im Projekt konsequent die `SR`-Namenskonvention (Set zuerst) verwendet wird.

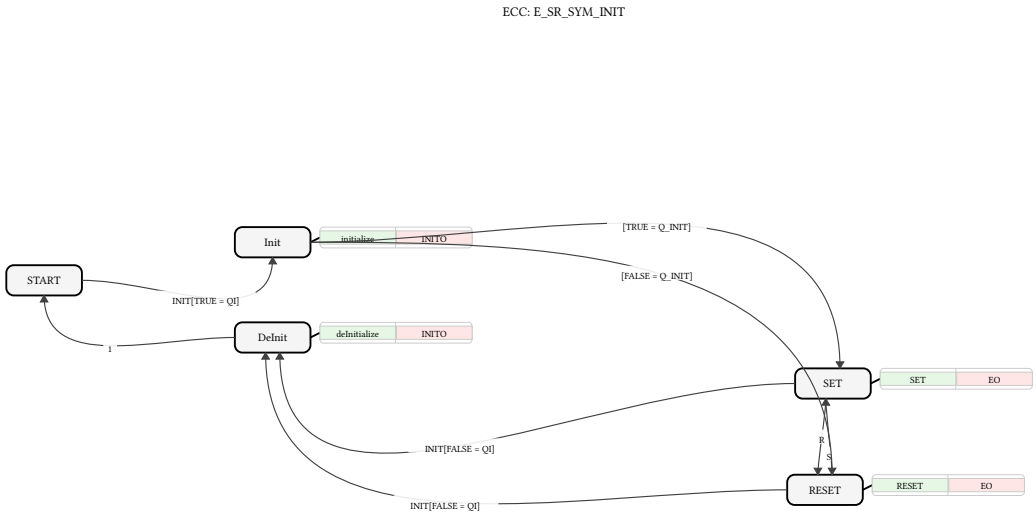
⚖️ Vergleich mit ähnlichen Bausteinen

- **E_RS_SYM**: funktional identisch, lediglich vertauschte Reihenfolge von `S`/`R` im Symbol.
- **E_SR_SYM_INIT**: dieselbe Grundfunktion, erweitert um eine `INIT`/`INITO`-Schnittstelle.
- **E_SR**: ohne symmetrisches Start-Up-Verhalten.

Fazit

`E_SR_SYM` ist die zu `E_RS_SYM` namenskonventionsgleiche, funktional identische Variante des symmetrischen Set-Reset-Flipflops.

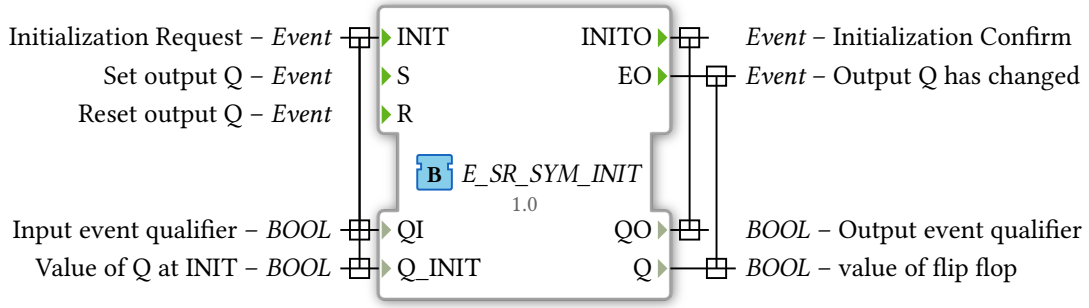
2.2.3.53 E_SR_SYM_INIT



• – ○ • – ○ • – ○ •

Einleitung

Der `E_SR_SYM_INIT` ist funktional identisch zu E_RS_SYM_INIT — analog zu E_SR_SYM/E_RS_SYM existiert er lediglich zur Wahrung der `SR`-Namenskonvention (Set zuerst) und erweitert E_SR_SYM um eine `INIT`/`INITO`-Schnittstelle.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, trägt `QI` und `Q_INIT`.
- **S (Set)**: Setzt den Ausgang `Q` auf `TRUE`.
- **R (Reset)**: Setzt den Ausgang `Q` auf `FALSE`.

Ereignis-Ausgänge

- **INITO**: Bestätigt die (De-)Initialisierung, trägt `QO`.
- **EO**: Wird nach jedem `S`- oder `R`-Ereignis ausgelöst, trägt `Q`.

Daten-Eingänge

- **QI** (BOOL): Eingangs-Event-Qualifier — `TRUE` initialisiert, `FALSE` deinitialisiert.
- **Q_INIT** (BOOL): Der Wert, auf den `Q` beim Initialisieren gesetzt wird.

Daten-Ausgänge

- **QO** (BOOL): Ausgangs-Event-Qualifier, spiegelt `QI` zurück.
- **Q** (BOOL): Der aktuelle Zustand des Flip-Flops.

Funktionsweise

Identisch zu E_RS_SYM_INIT: `INIT` mit `QI = TRUE` initialisiert `Q` über `Q_INIT`, `INIT` mit `QI = FALSE` deinitialisiert den Baustein zurück in den `START`-Zustand. Im laufenden Betrieb schaltet `S/R` zwischen `SET` und `RESET`, jeweils mit `EO`-Quittierung.

Technische Besonderheiten

Siehe E_RS_SYM_INIT — identisches Verhalten, lediglich `S/R`-Reihenfolge im Symbol vertauscht.

Zustandsübersicht

Zustand	Bedeutung
START	Unkonfigurierter Anfangszustand
Init	Initialisierung läuft, `QO := QI`
DeInit	Deinitialisierung läuft, `QO := FALSE`
SET	`Q = TRUE`
RESET	`Q = FALSE`

Anwendungsszenarien

Siehe E_RS_SYM_INIT.

⚖️ Vergleich mit ähnlichen Bausteinen

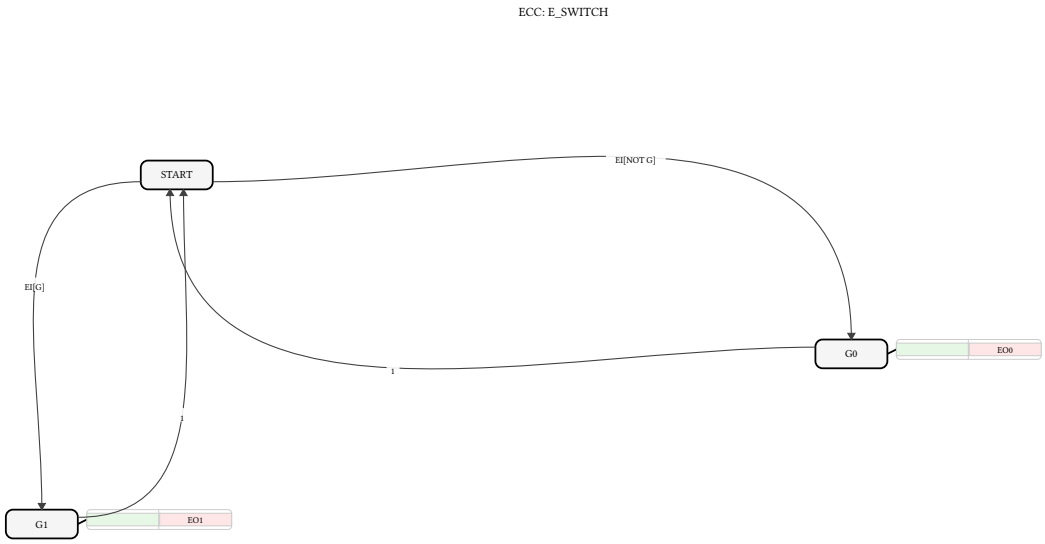
- **E_RS_SYM_INIT**: funktional identisch, vertauschte `S/R`-Reihenfolge.
- **E_SR_SYM**: dieselbe Grundfunktion ohne `INIT`/`INITO`.

Fazit

`E_SR_SYM_INIT` ist die zu `E_RS_SYM_INIT` namenskonventionsgleiche, funktional identische Variante mit projektierbarem Startwert.

2.2.3.54 E_SWITCH

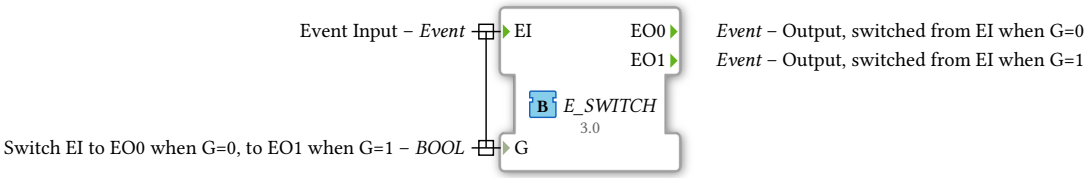
🎧 Podcast



- E_SWITCH: Die Weiche der Automatisierung – Warum Einfachheit IEC 61499 revolutioniert
- E_SWITCH: The Unsung Hero of Industrial Automation's Modular Design

Einleitung

Der `E_SWITCH` (Event Switch) ist ein grundlegender Funktionsbaustein nach IEC 61499, der als bedingte Ereignisweiche oder 1-zu-2-Demultiplexer fungiert. Er leitet ein einzelnes Eingangereignis (`EI`) basierend auf dem Zustand einer booleschen Steuervariable (`G`) an einen von zwei Ausgängen weiter.



Schnittstellenstruktur

Ereignis-Eingänge:

- **EI (Event Input)**: Das Eingangereignis, das weitergeleitet werden soll.
  - **Verbundene Daten**: `G`

Ereignis-Ausgänge:

- **EO0**: Der erste Ereignisausgang.
- **EO1**: Der zweite Ereignisausgang.

Daten-Eingänge:

- **G (Guard)**: Die boolesche Bedingung, die die Weiche steuert (Datentyp: `BOOL`).

Funktionsweise

1. **Ereignisempfang**: Der Baustein wartet auf ein Ereignis am Eingang `EI`.
2. **Prüfung der Bedingung**: Wenn das `EI`-Ereignis eintrifft, wird der Wert der Datenvariable `G` ausgewertet.
3. **Bedingte Weiterleitung**:

- **Wenn `G` = `FALSE` ist**: Das Ereignis wird an den Ausgang `EO0` weitergeleitet.
- **Wenn `G` = `TRUE` ist**: Das Ereignis wird an den Ausgang `EO1` weitergeleitet.

Das Ereignis am Eingang `EI` wird also immer an genau einen der beiden Ausgänge geleitet, abhängig von der Steuerbedingung `G`.

Technische Besonderheiten

- **Ereignisweiche**: Dient als grundlegendes „Switch“ oder „If-Else“-Konstrukt für den Ereignisfluss.
- **Zustandslos**: Der Baustein selbst hat keinen internen Speicher; seine Entscheidung basiert ausschließlich auf dem Wert von `G` im Moment des `EI`-Ereignisses.

Anwendungsszenarien

- **Bedingte Logik**: Ausführen von alternativen Aktionen basierend auf einer Bedingung. Wenn `G` wahr ist, führe Aktion A (`EO1`) aus, ansonsten führe Aktion B (`EO0`) aus.
- **Modusabhängige Verarbeitung**: Weiterleiten von Befehlen an unterschiedliche Verarbeitungspfade, je nachdem, ob sich das System in einem „Automatik“-Modus (`G=TRUE`) oder „Hand“-Modus (`G=FALSE`) befindet.
- **Fehlerbehandlung**: Weiterleiten eines Sensor-Ereignisses entweder an die normale Verarbeitungslogik (`G=FALSE` für „kein Fehler“) oder an eine Fehlerbehandlungsroutine (`G=TRUE` für „Fehler erkannt“).

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	E_SWITCH	E_SELECT	E_DEMUX
Auswahlkriterium	Boolesch (`G`)	Boolesch (`G`)	Index (`K`)
Richtung	1:2 (Demultiplexer)	2:1 (Multiplexer)	1:n (Demultiplexer)
Zustandsmodell	BasicFB	BasicFB	BasicFB/Generic

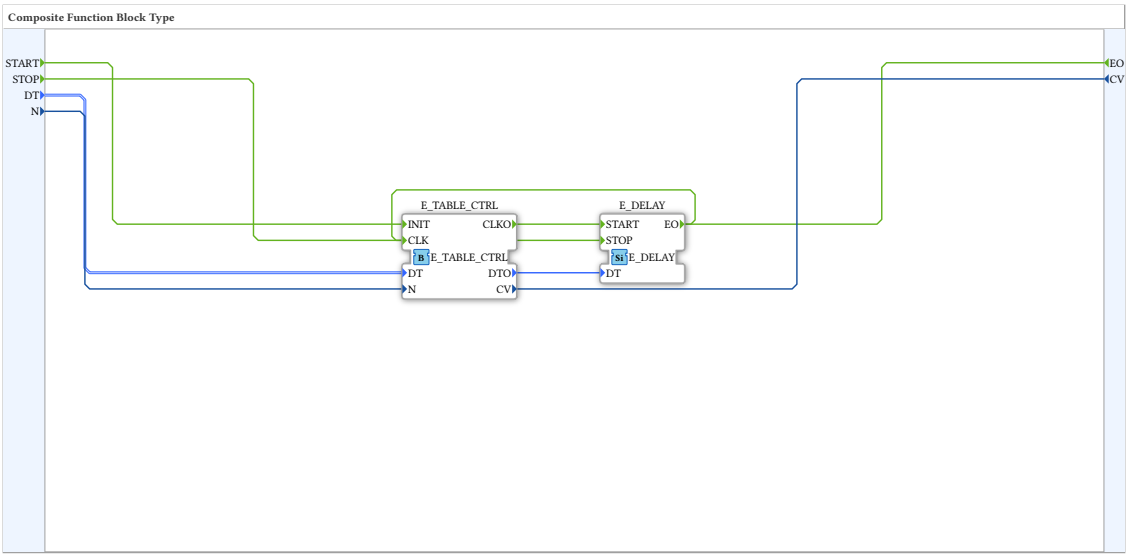
🛠️ Zugehörige Übungen

- Uebung_004b
- Uebung_004b2
- Uebung_004b3
- Uebung_005
- LinksRechts_T_FF_Event
- Uebung_007a3
- Uebung_008
- Uebung_009
- Uebung_019c
- Uebung_020a
- Uebung_020b
- Uebung_020c3
- Uebung_020d
- Uebung_020e2
- Uebung_020f2
- Uebung_020i
- Uebung_025
- SR_TO_QX_TRIG_EO
- Softkey_T_FF_ILOCK_TO_QX_BG
- Uebung_086
- Uebung_088
- Uebung_089

Fazit

Der `E_SWITCH`-Baustein ist ein fundamentaler und weit verbreiteter Baustein zur Implementierung von bedingten Verzweigungen in der ereignisgesteuerten Logik von IEC 61499. Seine Einfachheit und klare Funktion als „If-Else“-Weiche für Ereignisse machen ihn zu einem unverzichtbaren Werkzeug für die Erstellung von flexibler und robuster Steuerungssoftware.

2.2.3.55 E_TABLE

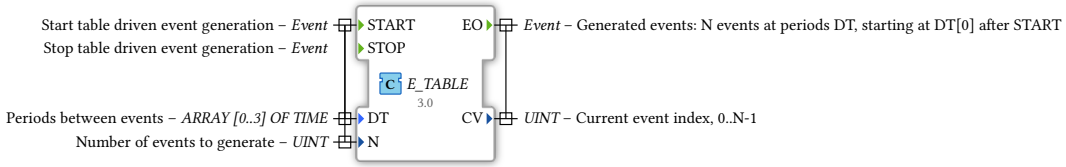


Start table driven event generation
Stop table driven event generation

• – ○ • – ○ • – ○ •

Einleitung

Der `E_TABLE` (Event Table) ist ein Funktionsbaustein nach IEC 61499, der eine endliche Sequenz von Ereignissen mit individuell definierbaren Zeitabständen generiert. Er liest die Zeitabstände aus einer Tabelle (einem Array) und feuert eine bestimmte Anzahl von Ereignissen nacheinander.



Schnittstellenstruktur

Ereignis-Eingänge

- **START**: Startet die Generierung der Ereignissequenz.
  - **Verbundene Daten**: `DT`, `N`
- **STOP**: Stoppt die Sequenz vorzeitig.

Ereignis-Ausgänge

- **EO (Event Output)**: Das Ausgangsereignis, das `N`-mal ausgelöst wird.
  - **Verbundene Daten**: `CV`

Daten-Eingänge

- **DT**: Ein Array von Zeitdauern (Datentyp: `TIME`, Größe: 4). `DT[i]` definiert die Verzögerungszeit, die *vor* dem `i`-ten Ereignis gewartet wird.
- **N**: Die Gesamtzahl der zu generierenden Ereignisse (Datentyp: `UINT`, max. 4 für diesen Baustein).

Daten-Ausgänge

- **CV (Current Value)**: Der Index des gerade ausgelösten Ereignisses (0 bis N-1) (Datentyp: `UINT`).

Funktionsweise

1. **Start der Sequenz**: Ein `START`-Ereignis löst den Baustein aus. Die Anzahl der zu generierenden Ereignisse wird durch `N` festgelegt.
2. **Ereignis-Generierung**: Der Baustein arbeitet die Tabelle ab:
   - Er wartet die in `DT[0]` definierte Zeitspanne ab. Danach wird das erste `EO`-Ereignis ausgelöst, und `CV` wird auf `0` gesetzt.
   - Er wartet die in `DT[1]` definierte Zeitspanne ab. Danach wird das zweite `EO`-Ereignis ausgelöst, und `CV` wird auf `1` gesetzt.
   - Dieser Prozess wiederholt sich, bis `N` Ereignisse generiert wurden.
3. **Ende der Sequenz**: Die Sequenz endet automatisch, nachdem `N` Ereignisse ausgelöst wurden.
4. **Stopp**: Ein `STOP`-Ereignis bricht die Sequenz an jeder Stelle sofort ab.

Beispiel:

- `N` = 3
- `DT` = `[T#2s, T#5s, T#1s]`
- Nach einem `START`-Ereignis:
  1. Nach 2 Sekunden wird `EO` mit `CV=0` ausgelöst.
  2. Weitere 5 Sekunden später wird `EO` mit `CV=1` ausgelöst.
  3. Weitere 1 Sekunde später wird `EO` mit `CV=2` ausgelöst.
  4. Die Sequenz ist beendet.

Technische Besonderheiten

- **Tabellengesteuerte Zeitabstände**: Die Stärke des Bausteins liegt in der Fähigkeit, variable Zeitabstände zwischen den Ereignissen zu definieren, anstatt eines festen Intervalls.
- **Sequenz-Zähler**: Der `CV`-Ausgang liefert wertvolle Information darüber, an welcher Stelle der Sequenz man sich gerade befindet.

Anwendungsszenarien

- **Steuerung von Schrittketten**: Anstoßen von Schritten mit jeweils unterschiedlichen Warte- oder Bearbeitungszeiten.
- **Komplexe Ansteuerung**: Ansteuern eines Aktors mit einer Serie von Impulsen mit variabler Pause.
- **Testautomatisierung**: Generieren einer komplexen, zeitlich definierten Stimulus-Sequenz für ein Testobjekt, bei der der `CV`-Ausgang zur Synchronisation mit der Auswertung genutzt werden kann.

⚖️ Vergleich mit ähnlichen Bausteinen

Merkmal	E_TABLE	E_CYCLE	E_N_TABLE
Ereignisabstand	Variabel (Array `DT`)	Fix (`DT`)	Variabel (Array `DT`)
Ausgänge	Ein `EO`-Kanal	Ein `EO`-Kanal	Mehrere (`EO0`..`EO3`)
Zähler-Ausgang (`CV`)	Ja	Nein	Nein (intern)

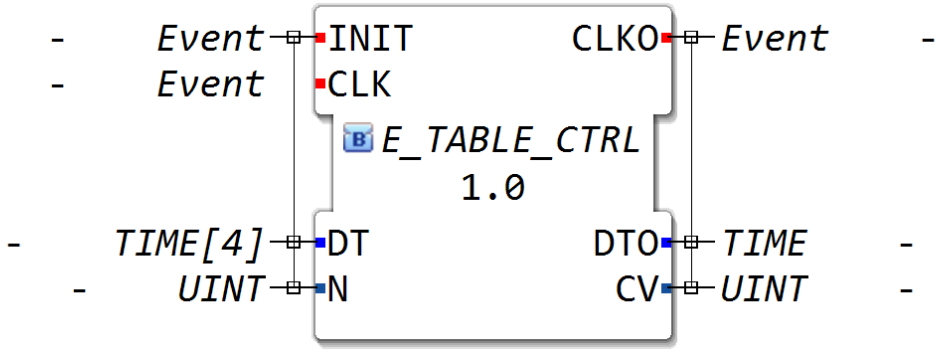
🛠️ Zugehörige Übungen

- Uebung_093

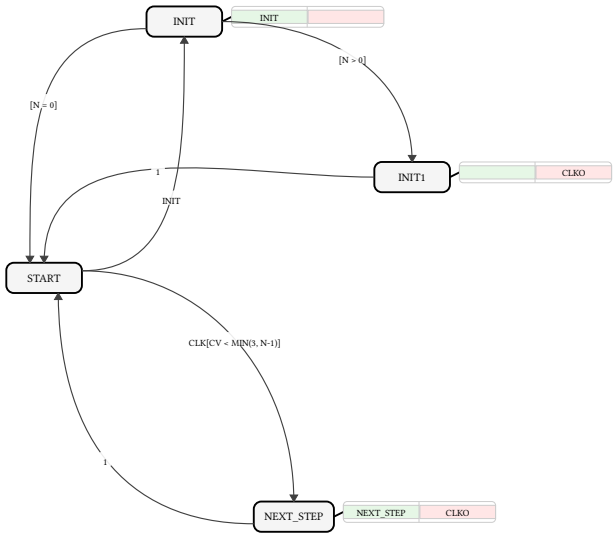
Fazit

Der `E_TABLE`-Baustein ist ein sehr flexibler Zeitgeber zur Erzeugung von komplexen, endlichen Ereignisketten. Seine Fähigkeit, variable Zeitabstände aus einer Tabelle zu lesen, macht ihn ideal für nichtperiodische, sequenzielle Steuerungsaufgaben. Er ist die Basis für andere Bausteine wie `E_N_TABLE`, liefert aber im Gegensatz zu diesem den aktuellen Zählerstand als nützliche Ausgangsinformation.

2.2.3.56 E_TABLE_CTRL



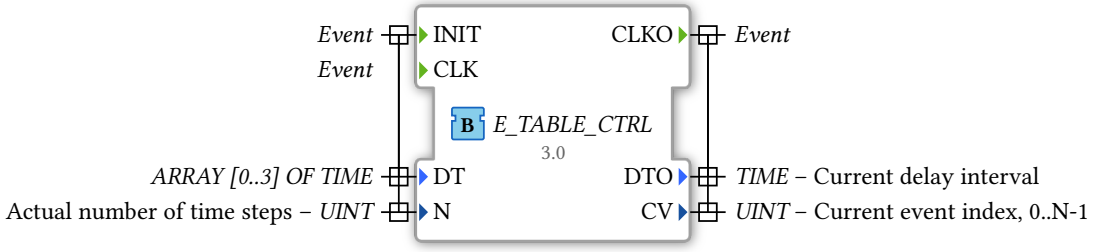
ECC: E_TABLE_CTRL



• – ○ • – ○ • – ○ •

Einleitung

Der **E_TABLE_CTRL** ist ein Unterstützungsbaustein für E_TABLE nach IEC 61499-1 (Annex A), unter EPL-2.0 Lizenz. Version 1.0 ermöglicht die präzise Steuerung von Ereignissequenzen basierend auf einer konfigurierbaren Zeit-Tabelle.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisiert die Ereignistabelle (mit DT- und N-Parametern)
- `CLK`: Taktsignal für den Tabellenfortschritt

Ereignis-Ausgang

- `CLKO`: Generiertes Taktereignis (mit DTO- und CV-Daten)

Daten-Eingänge

- `DT` (TIME-Array): Zeitintervalle für die Ereignisgenerierung
- `N` (UINT): Anzahl der aktiven Zeitschritte

Daten-Ausgänge

- `DTO` (TIME): Aktuelles Zeitintervall
- `CV` (UINT): Aktueller Ereignisindex (0..N-1)

Funktionsweise

1. **Initialisierung**:
   - Bei `INIT`-Ereignis wird der Index (CV) auf 0 gesetzt
   - Erstes Zeitintervall (DTO) aus DT-Array übernommen
2. **Tabellensteuerung**:
   - Jedes `CLK`-Ereignis erhöht CV um 1
   - Nächstes Zeitintervall aus DT-Array wird geladen
   - `CLKO` wird bei jedem Schritt generiert
3. **Zustandsautomat** (ECC):
   - **START**: Wartezustand
   - **INIT**: Initialisierungsphase
   - **INIT1**: Erste Ereignisgenerierung
   - **NEXT_STEP**: Tabellenfortschritt

Technische Besonderheiten

✓ **Tabellengesteuerte** Zeitplanung ✓ **Array-basierte** Konfiguration (bis zu 4 Zeitschritte) ✓ **Zustands-basierte** Implementierung (BasicFB) ✓ **Echtzeitfähige** Ereignisgenerierung

Anwendungsszenarien

- **Prozesssteuerung**: Komplexe Zeitabläufe
- **Testautomation**: Programmierbare Testsequenzen
- **Maschinensteuerung**: Bewegungsabläufe
- **Produktionslinien**: Taktgesteuerte Prozesse

Beziehung zu E_TABLE

Der `E_TABLE_CTRL`-Baustein ist nicht als eigenständiger Baustein für die direkte Anwendung gedacht, sondern als die **interne Steuerungslogik** des Composite-Funktionsbausteins `E_TABLE`.

Innerhalb von `E_TABLE` arbeitet `E_TABLE_CTRL` mit einem `E_DELAY`-Baustein zusammen:

1. `E_TABLE_CTRL` empfängt den `START`-Befehl und berechnet die erste Verzögerungszeit `DTO`.

2. Er sendet `DT0` über `CLK0` an den `E_DELAY`-Baustein.
3. Nachdem `E_DELAY` abgelaufen ist, meldet er dies über seinen `EO`-Ausgang zurück an den `CLK`-Eingang von `E_TABLE_CTRL`.
4. `E_TABLE_CTRL` berechnet daraufhin die nächste Verzögerungszeit und der Zyklus wiederholt sich.

Dieser Baustein kapselt also die reine Zustandslogik (welcher Schritt ist der nächste, wie lange dauert er), während der `E_DELAY` die eigentliche Zeitverzögerung ausführt.

🛠️ Zugehörige Übungen

- Uebung_175

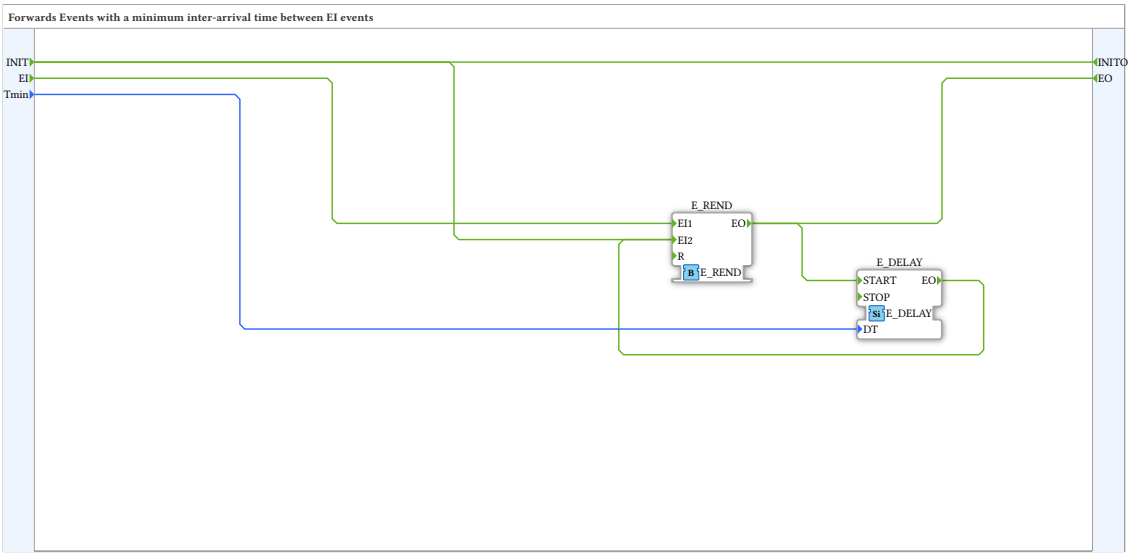
Fazit

Der E_TABLE_CTRL-Baustein erweitert die Möglichkeiten der tabellengesteuerten Ereignisgenerierung:

- Flexible Konfiguration mehrerer Zeitintervalle
- Präzise Steuerung komplexer Abläufe
- Robuste Zustandsmaschinen-Implementierung

Durch seine Array-basierte Zeitsteuerung eignet er sich ideal für Anwendungen mit variablen Prozessschritten. Die Integration als BasicFB gewährleistet zuverlässige Operation in IEC 61499-basierten Steuerungssystemen.

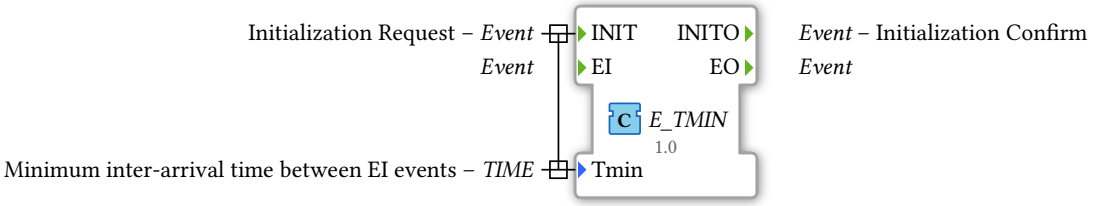
2.2.3.57 E_TMIN



• – ○ • – ○ • – ○ •

Einleitung

Der `E_TMIN` (Minimum inter-arrival time) ist ein Composite-Funktionsbaustein, der eingehende Ereignisse an `EI` nur dann an `EO` weiterleitet, wenn seit dem letzten weitergeleiteten Ereignis mindestens die Zeitspanne `Tmin` vergangen ist. Er dient als Ereignis-Drossel (Rate-Limiter) für beliebige Ereignisströme.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierung, trägt `Tmin`.

- **EI**: Eingehendes, zu drosselndes Ereignis.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung.
- **EO**: Das gedrosselte Ausgangsereignis.

Daten-Eingänge

- **Tmin** (TIME): Mindestabstand zwischen zwei aufeinanderfolgenden `EI`-Ereignissen, die tatsächlich zu `EO` führen.

Funktionsweise

Intern verwendet `E_TMIN` einen `E_REND` (Rendezvous zweier Ereignisse) und einen `E_DELAY`: Trifft `EI` ein, wird es an `E_REND.EI1` weitergegeben. `E_REND` feuert seinerseits `EO`, sobald es sowohl an `EI1` als auch an `EI2` mindestens ein Ereignis erhalten hat — zu Beginn ist `EI2` bereits durch `INIT` „vorbelegt“, sodass das erste `EI` sofort durchgereicht wird. Jedes von `E_REND` durchgelassene Ereignis startet gleichzeitig den `E_DELAY` mit der Zeit `Tmin`; erst wenn dieser abläuft, liefert er das nächste `EI2`-Ereignis an `E_REND`, wodurch der nächste `EI`-Durchlass frühestens nach `Tmin` möglich wird.

Technische Besonderheiten

- **Baustein aus zwei Standardelementen**: `E_TMIN` ist eine reine Verdrahtung aus `E_REND` und `E_DELAY`, ohne eigene ECC.
- **Erstes Ereignis unverzögert**: Das erste `EI` nach `INIT` wird sofort durchgelassen; die Drosselung setzt erst danach ein.
- **Baustein für die `_TMIN`-Familie**: Wird intern von E_D_FF_TMIN und E_D_FF_ANY_TMIN verwendet, um deren `EO`-Ausgänge zusätzlich zu drosseln.

Zustandsübersicht

Zustandslose Verdrahtungslogik: `E_REND` speichert intern, ob bereits ein `EI2` (vom `E_DELAY`) eingetroffen ist, `E_DELAY` läuft nach jedem durchgelassenen Ereignis erneut an.

Anwendungsszenarien

- **Drosselung schnell wechselnder Sensorereignisse**, um nachgeschaltete Logik (z. B. Logging, Netzwerkversand) nicht mit einer zu hohen Ereignisrate zu überlasten.
- **Baustein innerhalb von E_D_FF_TMIN/E_D_FF_ANY_TMIN**, um Änderungsereignisse zusätzlich zeitlich zu begrenzen.

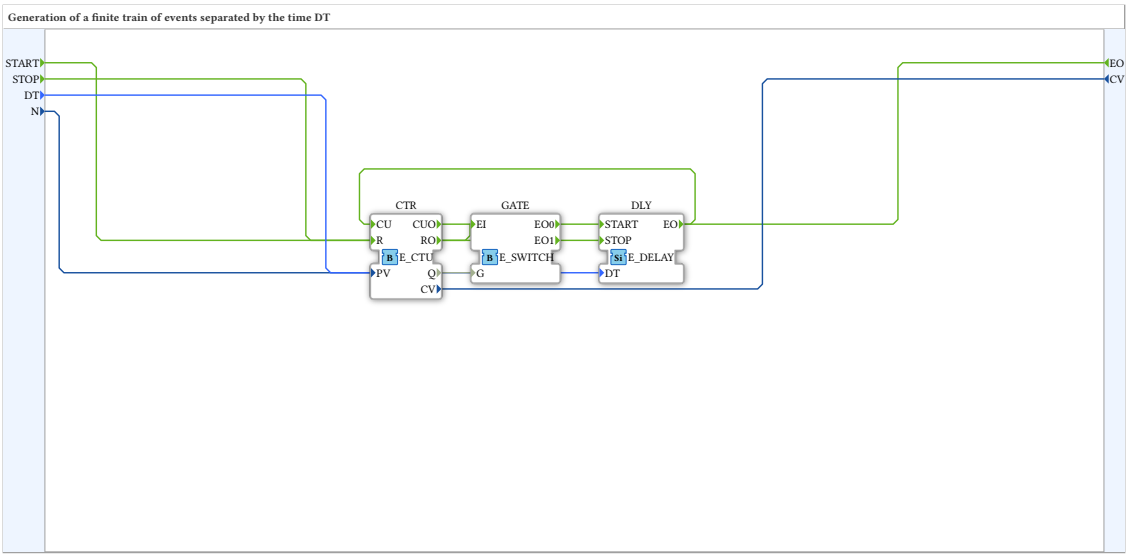
⚖️ Vergleich mit ähnlichen Bausteinen

- **`E_REND`**: die intern verwendete Synchronisationsbasis, ohne Zeitkomponente.
- **`E_DELAY`**: der intern verwendete Timer.
- **E_D_FF_TMIN** / **E_D_FF_ANY_TMIN**: kombinieren `E_TMIN` zusätzlich mit einem Daten-Latch (`E_D_FF`/`E_D_FF_ANY`).

Fazit

`E_TMIN` ist ein einfacher, aber wirkungsvoller Ereignis-Drosselbaustein, der aus zwei Standardelementen (`E_REND`, `E_DELAY`) zusammengesetzt ist und eine Mindestabstandszeit zwischen durchgelassenen Ereignissen garantiert.

2.2.3.58 E_TRAIN



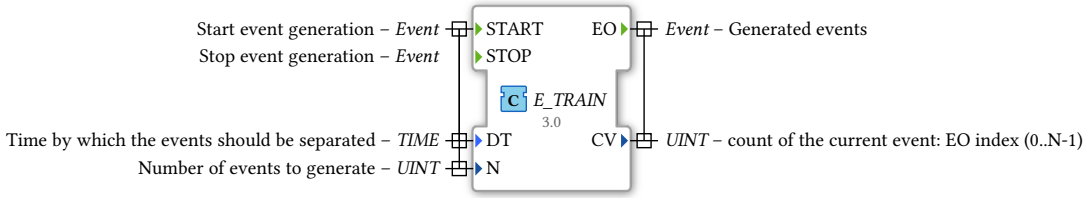
Start event generation -
Stop event generation -

Time by which the events should be seperated -
Number of events to generate -

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `E_TRAIN` (Event Train) ist ein Funktionsbaustein nach IEC 61499, der eine endliche Folge von Ereignissen („Ereigniszug“) mit einem festen Zeitabstand generiert. Nach dem Start werden `N` Ereignisse im Abstand von `DT` erzeugt.



Schnittstellenstruktur

Ereignis-Eingänge:

- **START**: Startet die Generierung der Ereignisfolge.
  - **Verbundene Daten**: `DT`, `N`
- **STOP**: Stoppt die laufende Generierung der Ereignisfolge vorzeitig.

Ereignis-Ausgänge:

- **EO (Event Output)**: Das Ausgangsereignis, das periodisch ausgelöst wird.
  - **Verbundene Daten**: `CV`

Daten-Eingänge:

- **DT (Delay Time)**: Die Zeitdauer, die zwischen den einzelnen EO-Ereignissen vergeht (Datentyp: TIME).
- **N**: Die Gesamtzahl der zu generierenden Ereignisse (Datentyp: UINT).

Daten-Ausgänge:

- **CV (Current Value)**: Der Zählerstand, der den Index des gerade ausgelösten Ereignisses angibt (0 bis N-1) (Datentyp: UINT).

Funktionsweise

1. **Start der Sequenz**: Ein START-Ereignis löst den Baustein aus. Ein interner Zähler wird zurückgesetzt, und die Parameter N (Anzahl) und DT (Zeitabstand) werden übernommen. Der Timer für das erste Ereignis wird gestartet.
2. **Ereignis-Generierung**:
   - Nach Ablauf der Zeit DT wird das erste EO-Ereignis ausgelöst. Der CV-Ausgang hat dabei den Wert 0.
   - Unmittelbar danach wird der Timer für das nächste Ereignis neu gestartet.
   - Nach einer weiteren Zeitspanne DT wird das nächste EO-Ereignis mit CV=1 ausgelöst.
3. **Ende der Sequenz**: Dieser Zyklus wiederholt sich, bis N Ereignisse generiert wurden. Nachdem das letzte Ereignis (mit CV = N-1) ausgelöst wurde, stoppt der Baustein automatisch.
4. **Stopp**: Ein STOP-Ereignis kann die Sequenz jederzeit vorzeitig abbrechen.

Technische Besonderheiten

- **Puls-Generator**: Erzeugt eine feste Anzahl von Impulsen mit konstantem Abstand.
- **Interner Zähler**: Der Baustein verwendet intern einen E_CTU-Zähler, um die Anzahl der bereits generierten Ereignisse zu verfolgen.
- **Sequenz-Zähler**: Der CV-Ausgang liefert wertvolle Information darüber, an welcher Stelle der Sequenz man sich gerade befindet.

Anwendungsszenarien

- **Schrittmotor-Ansteuerung**: Generieren einer exakten Anzahl von Schritten (N) mit einer definierten Geschwindigkeit (bestimmt durch DT).
- **Dosierprozesse**: Auslösen von N Dosiervorgängen im Abstand von DT.
- **Testsequenzen**: Senden einer definierten Anzahl von Test-Triggern an ein zu prüfendes Gerät.

⚖️ Vergleich mit ähnlichen Bausteinen

Merkmal	E_TRAIN	E_CYCLE	E_TABLE
Ereignisabstand	Fix (DT)	Fix (DT)	Variabel (Array DT)
Ereignisanzahl	Endlich (N)	Unendlich	Endlich (N)
Zähler-Ausgang (CV)	Ja	Nein	Ja

🛠️ Zugehörige Übungen

- Uebung_035a2
- Uebung_035a2_AX
- Uebung_091

Fazit

Der E_TRAIN-Baustein ist ideal für Anwendungen, die eine feste Anzahl von Ereignissen in einem konstanten Zeitintervall benötigen. Er kombiniert die Funktionalität eines Taktgebers (E_CYCLE) mit einem Zähler, um eine kontrollierte, endliche Impulsfolge zu erzeugen. Der CV-Ausgang bietet dabei eine nützliche Rückmeldung über den Fortschritt der Sequenz.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

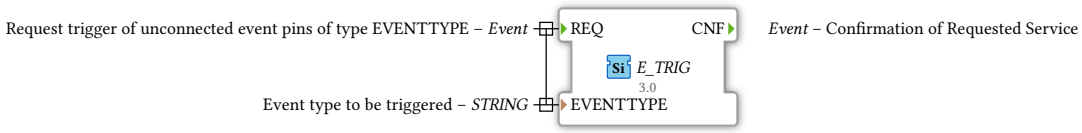
- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de

2.2.3.59 E_TRIG

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `E_TRIG` ist ein spezieller Service-Funktionsbaustein nach IEC 61499. Seine Aufgabe ist es, *unverbundene* Ereignis-Eingänge eines bestimmten Typs (Namens) innerhalb einer Ressource programmatisch auszulösen. Dies ermöglicht fortgeschrittene Steuerungs- und Initialisierungs-Szenarien.



Schnittstellenstruktur

Ereignis-Eingänge:

- **REQ (Request)**: Löst den Trigger-Mechanismus aus.
  - **Verbundene Daten**: `EVENTTYPE`

Ereignis-Ausgänge:

- **CNF (Confirmation)**: Bestätigt, dass der Trigger-Befehl abgesetzt wurde.

Daten-Eingänge:

- **EVENTTYPE**: Der Name der Ereignis-Eingänge, die ausgelöst werden sollen (Datentyp: `STRING`).
  - **Standardwert**: ‚EInit'

Funktionsweise

1. **Trigger-Anforderung**: Bei einem `REQ`-Ereignis liest der Baustein den am `EVENTTYPE`-Eingang anliegenden String.
2. **Interne Auslösung**: Der Baustein weist die 4diac-Laufzeitumgebung (die Ressource) an, alle *unverbundenen* Ereignis-Eingänge von Funktionsbausteinen innerhalb derselben Ressource zu finden, deren Name mit dem `EVENTTYPE`-String übereinstimmt.
3. **Ereignis-Generierung**: Die Laufzeitumgebung löst diese gefundenen Ereignis-Eingänge aus.
4. **Bestätigung**: Unmittelbar nachdem der Befehl abgesetzt wurde, löst der `E_TRIG`-Baustein sein `CNF`-Ereignis aus.

Beispiel: Angenommen, mehrere Funktionsbausteine in einer Ressource haben einen unverbundenen Eingang namens `INIT`. Ein `E_TRIG`-Baustein mit `EVENTTYPE = 'INIT'` würde bei einem `REQ` all diese `INIT`-Eingänge gleichzeitig auslösen, um eine erneute Initialisierung dieser Bausteine zu erzwingen.

Technische Besonderheiten

- **Service-Baustein**: Dies ist kein einfacher Logik-Baustein, sondern eine Schnittstelle zu einem speziellen Dienst der 4diac-Laufzeitumgebung.
- **Wirkt auf unverbundene Eingänge**: Die Funktionalität zielt explizit auf Eingänge, die nicht grafisch im Netzwerk verdrahtet sind. Dies ist ein mächtiges, aber mit Vorsicht zu verwendendes Feature.
- **Standard-Anwendungsfall `EInit`**: Der Standardwert ‚EInit' deutet auf den primären Anwendungsfall hin: das manuelle Auslösen von Initialisierungs-Ereignissen, die normalerweise nur beim Start einer Ressource einmalig ausgeführt werden.

Anwendungsszenarien

- **Manuelle Re-Initialisierung**: Zurücksetzen von Teilen einer Anwendung in einen definierten Anfangszustand, ohne die gesamte Ressource neu starten zu müssen.
- **Dynamische Konfiguration**: Nach dem Laden einer neuen Konfiguration können relevante Bausteine über einen gezielten `E_TRIG`-Aufruf dazu gebracht werden, die neuen Parameter zu übernehmen.
- **Test und Debugging**: Manuelles Auslösen von bestimmten Ereignissen in einer laufenden Anwendung zu Testzwecken.

🛠️ Zugehörige Übungen

- Uebung_176

Fazit

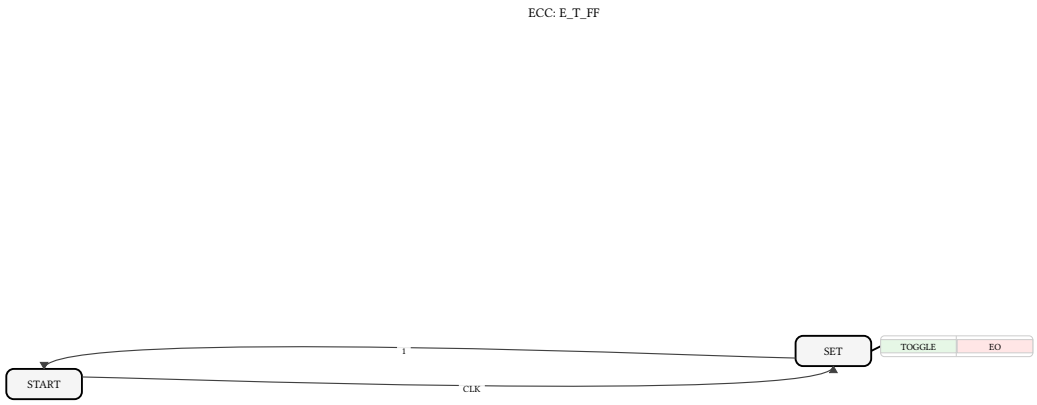
Der `E_TRIG`-Baustein ist ein Werkzeug für fortgeschrittene Anwendungsfälle und für Entwickler, die tiefere Kenntnisse der 4diac-Ausführungsumgebung haben. Er bietet eine leistungsstarke Methode, um die normale, grafisch definierte Ereignisfluss-Logik zu umgehen und direkt mit der Laufzeitumgebung zu interagieren. Er sollte mit Bedacht eingesetzt werden, da seine Wirkung nicht unmittelbar aus dem grafischen Netzwerk ersichtlich ist.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.2.3.60 E_T_FF

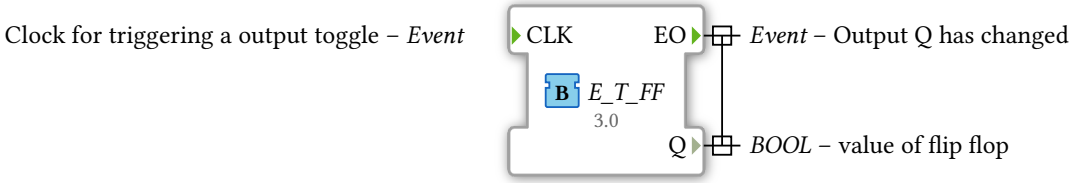
🎧 Podcast



- Der E_T_FF in IEC 61499: Modulares Kippen für die Industrie 4.0
- Der E_T_FF_SR-Baustein: Herzstück der IEC 61499 – Speichern, Umschalten, Reagieren
- Unpacking E_T_FF_SR: The Secret Toggle Switch of Industrial Control Systems

Einleitung

Der `E_T_FF` (Event-driven Toggle Flip-Flop) ist ein ereignisgesteuerter Kippschalter, der seinen Zustand (`Q`) bei jedem eingehenden Taktereignis (`CLK`) wechselt. Er ist das digitale Äquivalent eines „Stromstoß-schalters“ (Stromstoßrelais), bei dem ein kurzer Impuls den Zustand dauerhaft ändert.



Schnittstellenstruktur

Ereignis-Eingänge:

- **CLK (Clock)**: Das Taktereignis, das den Zustand von `Q` umschaltet.

Ereignis-Ausgänge:

- **EO (Event Output)**: Wird ausgelöst, wenn sich der Zustand von `Q` ändert.
  - **Verbundene Daten**: `Q`

Daten-Ausgänge:

- **Q**: Der aktuelle Zustand des Flip-Flops (Datentyp: `BOOL`).

Funktionsweise

Der `E_T_FF`-Baustein ist ein einfacher Toggle Flip-Flop:

1. **Zustandswechsel**: Bei jedem eingehenden `CLK`-Ereignis ändert der Ausgang `Q` seinen Zustand: War `Q` `TRUE`, wird es `FALSE`, und war `Q` `FALSE`, wird es `TRUE`.
2. **Ereignisauslösung**: Jede Zustandsänderung von `Q` löst das `EO`-Ereignis aus.

Technische Besonderheiten

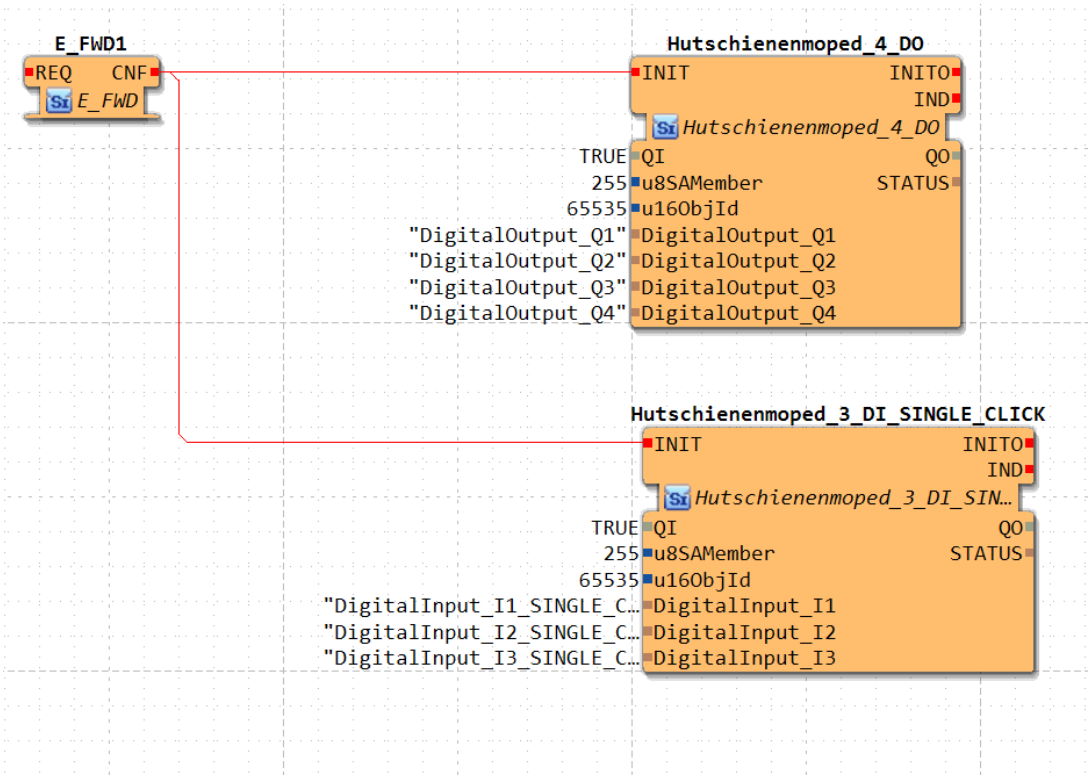
- **Stromstoßschalter-Analogie**: Der Baustein verhält sich wie ein Stromstoßschalter: Ein kurzer Impuls (`CLK`) schaltet das Licht (`Q`) ein, der nächste Impuls schaltet es aus.
- **Speicherfunktion**: `Q` speichert den letzten Zustand des Flip-Flops.
- **Zustandslos zwischen Takten**: Änderungen am `CLK`-Eingang beeinflussen `Q` nur zum Zeitpunkt des Ereignisses.

Anwendungsbeispiele

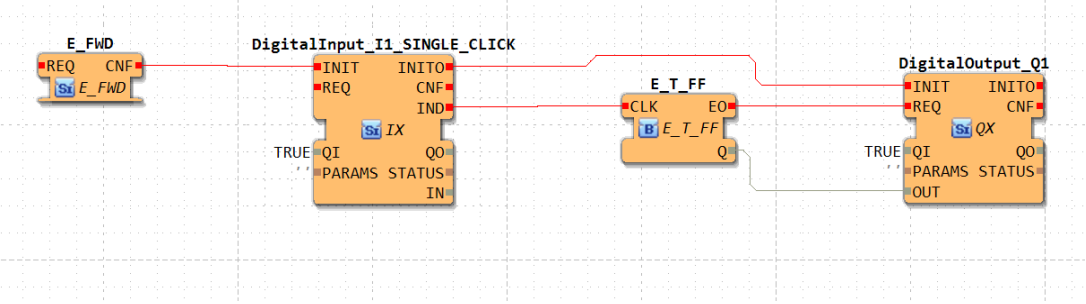
Taster für eine Lampe

Mit einem `E_T_FF` lässt sich eine Taster-Logik für eine Lampe realisieren:

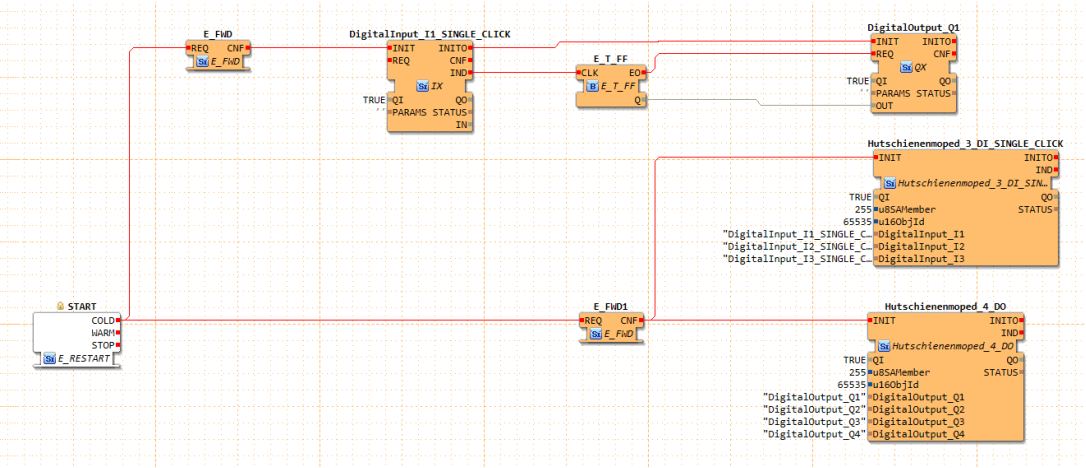
- **Konzept**: Ein Taster erzeugt ein `CLK`-Ereignis. Jedes Drücken schaltet die Lampe (`Q`) ein oder aus.
- **Grafische Darstellung**:
  - Mapping:



  - Applikation:



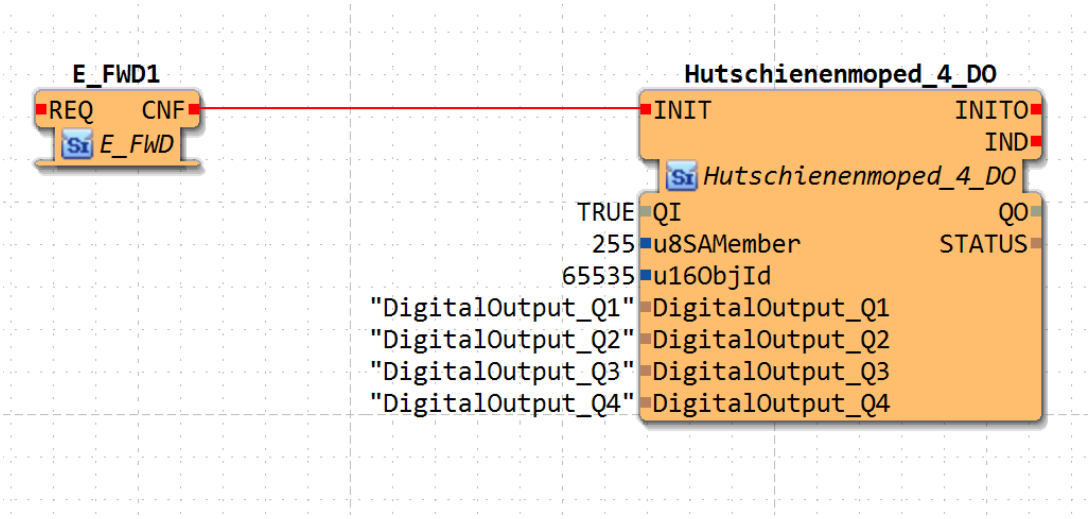
– Embedded Ressource:



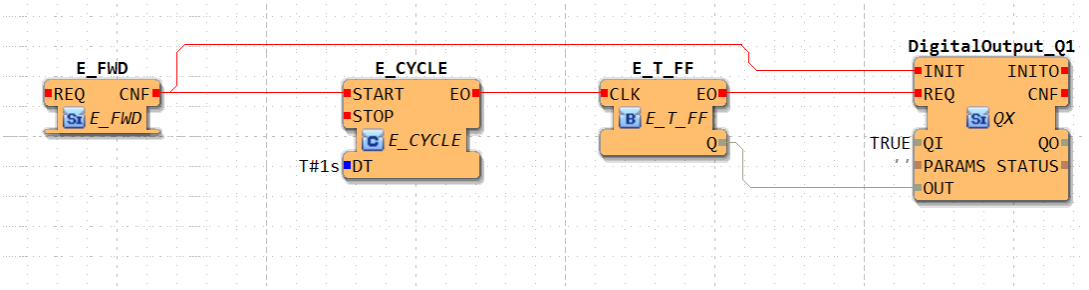
Blinker

Durch Rückkopplung mit einem Zeitgeber lässt sich ein Blinker realisieren:

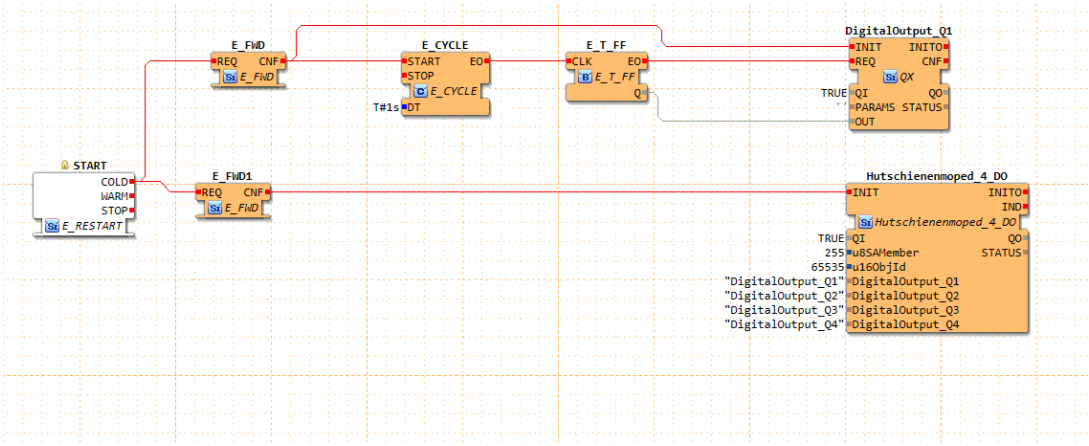
- **Konzept**: Das `EO`-Ereignis des `E_T_FF` startet einen `E_DELAY`, dessen `EO` wiederum als `CLK` für den `E_T_FF` dient. Dies erzeugt einen periodischen Zustandswechsel.
- **Grafische Darstellung**:
  – Mapping:



– Applikation:



– Embedded Ressource:



🛠️ Zugehörige Übungen

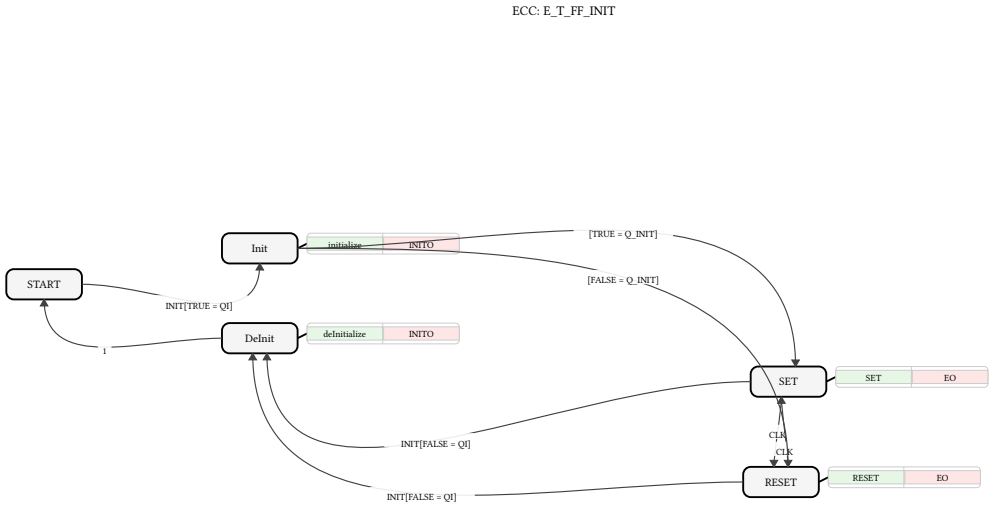
- Uebung_004a
- Uebung_004a2
- Uebung_004a2_2
- Uebung_004a2_3
- Uebung_004a3
- Uebung_004a4
- Uebung_004a5
- Uebung_004a6
- Uebung_004a8
- Uebung_004a9
- Uebung_004c1
- Uebung_004c2
- Uebung_004c3
- Uebung_004c4
- Uebung_004c5
- Uebung_004c6
- Uebung_004c7
- Uebung_005
- LinksRechts_T_FF_Event
- Uebung_007
- Uebung_007a1
- Uebung_007a2
- Uebung_010b2
- Uebung_010b3
- Uebung_010b6
- Uebung_010b7
- Uebung_010b8
- Uebung_010b9
- Uebung_010bA
- Uebung_010bA2
- Uebung_010bA3

- Uebung_010bA4
- Uebung_035a2
- Uebung_080c
- Uebung_087
- Uebung_087a1
- Uebung_088
- Uebung_089
- Uebung_091
- Uebung_093
- Uebung_093b
- Uebung_094
- Uebung_094a
- Uebung_095
- Uebung_152
- Uebung_153

Fazit

Der `E_T_FF`-Baustein ist ein einfacher, aber äußerst vielseitiger Speicherbaustein. Seine „Stromstoßschalter“-Funktion ist fundamental für die Implementierung von Taster-Logiken, Blinkern und anderen bistabilen Steuerungen in IEC 61499-Anwendungen.

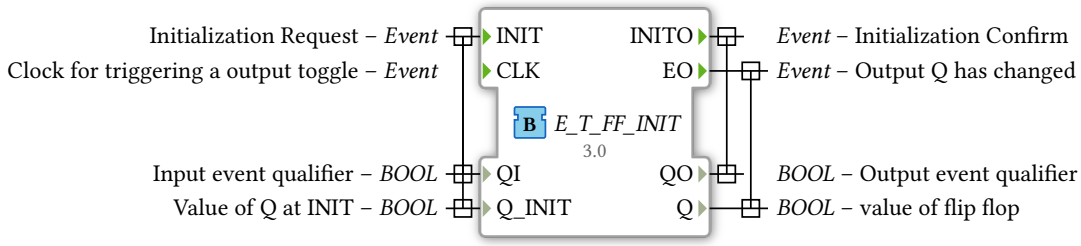
2.2.3.61 E_T_FF_INIT



• – ○ • – ○ • – ○ •

Einleitung

Der `E_T_FF_INIT` (Toggle-Flipflop mit Initialisierung) kombiniert das Umschaltverhalten eines Toggle-Flipflops (Ausgang `Q` wechselt bei jedem `CLK`-Ereignis den Zustand) mit einer expliziten `INIT`/`INITO`-Schnittstelle zum gezielten Setzen eines definierten Startwerts.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, trägt `QI` und `Q_INIT`.
- **CLK**: Taktereignis, löst einen Zustandswechsel von `Q` aus.

Ereignis-Ausgänge

- **INITO**: Bestätigt die (De-)Initialisierung, trägt `QO`.
- **EO**: Wird bei jedem `CLK`-Ereignis ausgelöst, trägt `Q`.

Daten-Eingänge

- **QI** (BOOL): Eingangs-Event-Qualifier — `TRUE` initialisiert, `FALSE` deinitialisiert.
- **Q_INIT** (BOOL): Der Wert, auf den `Q` beim Initialisieren gesetzt wird.

Daten-Ausgänge

- **QO** (BOOL): Ausgangs-Event-Qualifier, spiegelt `QI` zurück.
- **Q** (BOOL): Der aktuelle Zustand des Flipflops.

Funktionsweise

`INIT` mit `QI = TRUE` initialisiert `Q` über `Q_INIT` (Zustände `Init` → `SET`/`RESET`, je nach `Q_INIT`). Im laufenden Betrieb wechselt jedes `CLK`-Ereignis den Zustand: aus `SET` nach `RESET` und umgekehrt, jeweils mit `EO`-Quittierung und dem entsprechend gesetzten `Q`. `INIT` mit `QI = FALSE` deinitialisiert den Baustein (`DeInit` → `START`).

Technische Besonderheiten

- **Toggle statt Set/Reset**: Im Gegensatz zu E_RS_SYM_INIT/E_SR_SYM_INIT, die getrennte `S`/`R`-Eingänge besitzen, hat `E_T_FF_INIT` nur einen einzigen Takteingang `CLK`, der bei jedem Ereignis zwischen `SET` und `RESET` umschaltet.
- **Gleiche INIT/DeInit-Struktur** wie die `_SYM_INIT`-Bausteine: `QI` schaltet zwischen Initialisierung und Deinitialisierung, `Q_INIT` bestimmt den Startwert.

Zustandsübersicht

Zustand	Bedeutung
START	Unkonfigurierter Anfangszustand
Init	Initialisierung läuft, `QO := QI`
DeInit	Deinitialisierung läuft, `QO := FALSE`
SET	`Q = TRUE`, wechselt bei `CLK` nach `RESET`
RESET	`Q = FALSE`, wechselt bei `CLK` nach `SET`

Anwendungsszenarien

- **Blinklogik mit definiertem Startzustand**: Ein Signal soll bei jedem Takt umschalten, aber nach dem Systemstart mit einem bekannten Anfangswert (`Q_INIT`) beginnen, statt mit einem zufälligen Zustand.
- **Frequenzteilung**: `E_T_FF_INIT` halbiert die Ereignisfrequenz von `CLK` auf `EO` (Zustandswechsel bei jedem zweiten `CLK` wieder in denselben Zustand), mit kontrolliertem Startwert.

⚖️ Vergleich mit ähnlichen Bausteinen

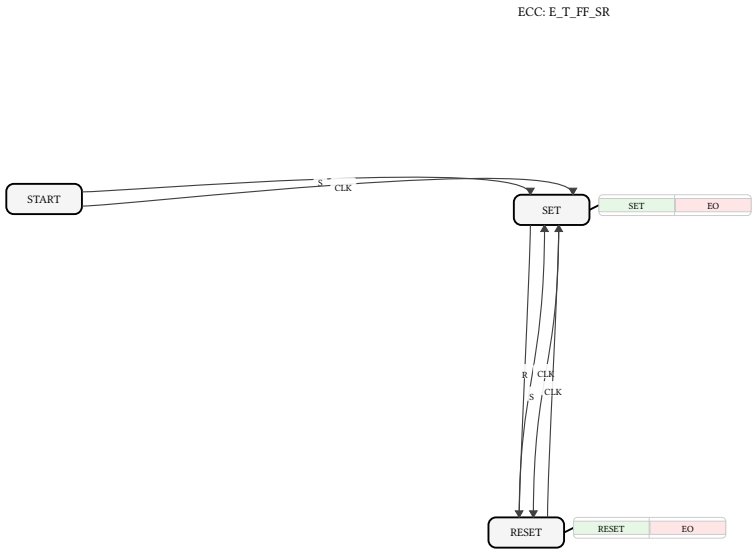
- **`E_T_FF`**: dieselbe Toggle-Grundfunktion ohne `INIT`/`INITO`-Schnittstelle.
- **E_RS_SYM_INIT** / **E_SR_SYM_INIT**: dieselbe INIT/DeInit-Struktur, aber mit getrenntem Set-/Reset-Eingang statt eines einzelnen Takteingangs.
- **E_T_FF_SR_SYM** / **E_T_FF_SR_SYM_INIT**: kombinieren zusätzlich `S`/`R`-Eingänge mit dem Toggle-Verhalten.

Fazit

`E_T_FF_INIT` liefert ein Toggle-Flipflop mit projektierbarem, garantiert definiertem Startwert und eignet sich für Blink- und Frequenzteilerlogik, bei der der Zustand nach dem Systemstart nicht dem Zufall überlassen werden soll.

2.2.3.62 E_T_FF_SR

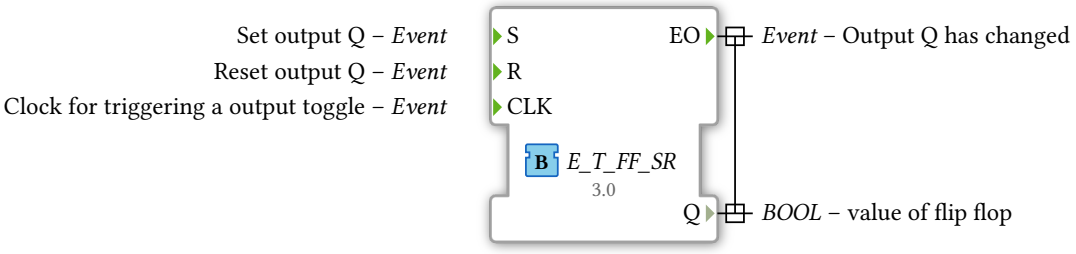
🎧 Podcast



- Der E_T_FF_SR-Baustein: Herzstück der IEC 61499 – Speichern, Umschalten, Reagieren
- Unpacking E_T_FF_SR: The Secret Toggle Switch of Industrial Control Systems

Einleitung

Der `E_T_FF_SR` (Event-driven Toggle Flip-Flop mit Set/Reset) ist ein erweiterter bistabiler Funktionsbaustein nach IEC 61499. Er kombiniert die Funktionalität eines `E_T_FF` (Toggeln) mit zusätzlichen `S`- (Setzen) und `R`- (Rücksetzen) Eingängen.



Schnittstellenstruktur

Ereignis-Eingänge:

- **S (Set)**: Setzt den Ausgang `Q` auf `TRUE`.
- **R (Reset)**: Setzt den Ausgang `Q` auf `FALSE`.
- **CLK (Clock)**: Löst eine Umschaltung (Toggeln) des Ausgangs `Q` aus.

Ereignis-Ausgänge:

- **EO (Event Output)**: Wird ausgelöst, wenn sich der Zustand von `Q` ändert.
  - **Verbundene Daten**: `Q`

Daten-Ausgänge:

- **Q**: Der aktuelle Zustand des Flip-Flops (Datentyp: `BOOL`).

Funktionsweise

Der `E_T_FF_SR` ist ein zustandsbehafteter Baustein, dessen Ausgang `Q` durch drei Ereigniseingänge beeinflusst wird:

1. **Setzen (S)**: Wenn ein `S`-Ereignis eintrifft, wird `Q` auf `TRUE` gesetzt. Falls `Q` zuvor `FALSE` war, wird `EO` ausgelöst.
2. **Zurücksetzen (R)**: Wenn ein `R`-Ereignis eintrifft, wird `Q` auf `FALSE` gesetzt. Falls `Q` zuvor `TRUE` war, wird `EO` ausgelöst.
3. **Toggeln (CLK)**: Wenn ein `CLK`-Ereignis eintrifft, wird `Q` umgeschaltet (getoggelt). Falls `Q` seinen Zustand ändert, wird `EO` ausgelöst.

Besonderheiten des Verhaltens aus dem `START`-Zustand

Der Baustein startet im `START`-Zustand (impliziert `Q` ist undefiniert/`FALSE`).

- Wenn das erste Ereignis ein `S` ist, wird `Q` auf `TRUE` gesetzt.
- Wenn das erste Ereignis ein `R` ist, wird `Q` auf `FALSE` gesetzt.
- **Wenn das erste Ereignis ein `CLK` ist, wird `Q` auf `TRUE` gesetzt** (nicht getoggelt von `FALSE` nach `TRUE`). Subsequent `CLK` events werden dann normal toggeln.

Technische Besonderheiten

- **Asynchrone Set/Reset**: Die `S`- und `R`-Eingänge können den Zustand von `Q` jederzeit überschreiben.
- **Toggel-Funktion**: Der `CLK`-Eingang ermöglicht einen einfachen Zustandswechsel.
- **Keine Priorisierung (Dominanz)**: Wie bei `E_RS` und `E_SR` gibt es keine fest definierte Priorität, wenn `S`, `R` oder `CLK` gleichzeitig eintreffen. Die Verarbeitungsreihenfolge der 4diac-Laufzeitumgebung bestimmt den Endzustand.
- **Initiales Verhalten mit `CLK`**: Bei einem `CLK`-Ereignis aus dem `START`-Zustand wird der Baustein initial gesetzt (`Q=TRUE`), anstatt zu toggeln. Dies sollte bei der Initialisierung des Systems beachtet werden.

Anwendungsszenarien

- **Steuerung mit manueller Korrektur**: Ein Toggler (`CLK`) für eine Lampe, die aber bei Bedarf (z.B. Sicherheitsgründen) direkt Ein (`S`) oder Aus (`R`) geschaltet werden kann.
- **Modus-Umschaltung**: Zwischen verschiedenen Modi hin- und herwechseln (`CLK`), mit der Möglichkeit, einen Grundmodus (`R`) oder einen Sondermodus (`S`) direkt anzusteuern.
- **Fehler-Reset und Toggle**: Ein Fehlerzustand kann durch `S` gesetzt werden, durch `R` quittiert, und der Zustand des Fehler-Handlings kann durch `CLK` getoggelt werden.

🛠️ Zugehörige Übungen

- Uebung_004a7
- Uebung_006a
- Uebung_006a2
- Uebung_006a3
- Uebung_006a4
- Uebung_179
- Uebung_180

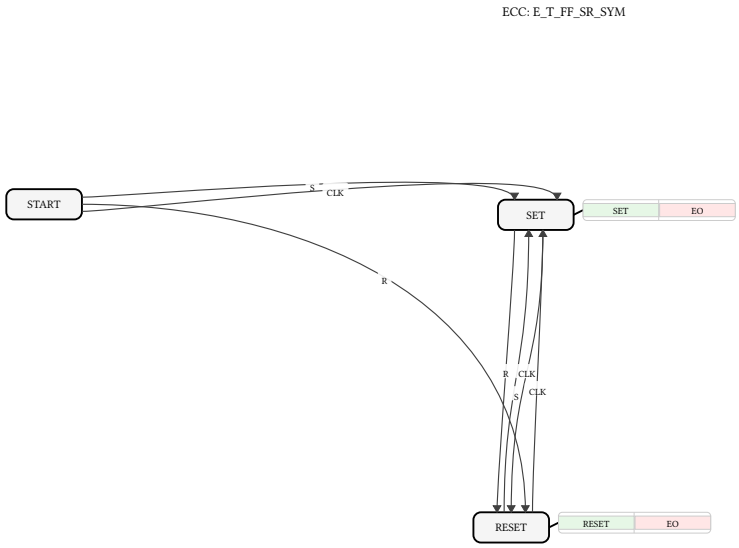
Fazit

Der `E_T_FF_SR`-Baustein bietet maximale Flexibilität für Speicher- und Steuerungsaufgaben, indem er die Toggel-Funktion mit direkten Set- und Reset-Möglichkeiten kombiniert. Die spezifischen Verhaltensweisen aus dem `START`-Zustand und die fehlende Priorisierungsgarantie bei simultanen Ereignissen müssen bei der Implementierung sorgfältig berücksichtigt werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

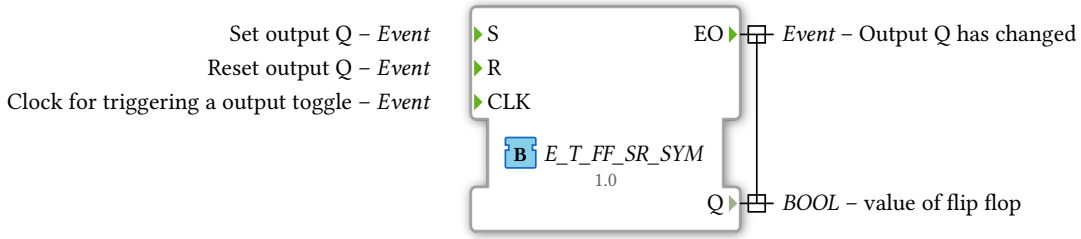
2.2.3.63 E_T_FF_SR_SYM



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `E_T_FF_SR_SYM` vereint in einem Baustein die Funktionalität von E_RS_SYM (bistabiles Set/Reset mit symmetrischem Start-Up) und eines Toggle-Flipflops: Zusätzlich zu den Eingängen `S` und `R` besitzt er einen Takteingang `CLK`, der den Ausgang `Q` unabhängig von `S`/`R` umschaltet.



Schnittstellenstruktur

Ereignis-Eingänge

- **S (Set)**: Setzt `Q` auf `TRUE`.
- **R (Reset)**: Setzt `Q` auf `FALSE`.
- **CLK**: Kehrt den aktuellen Zustand von `Q` um (Toggle).

Ereignis-Ausgänge

- **EO**: Wird nach jedem S-, R- oder CLK-Ereignis ausgelöst, trägt Q.

Daten-Ausgänge

- **Q** (BOOL): Der aktuelle Zustand.

Funktionsweise

Aus dem Anfangszustand START führen S, R **und** CLK zu einem definierten Folgezustand (CLK führt dabei nach SET) — das symmetrische Start-Up-Verhalten von E_RS_SYM gilt hier für alle drei Ereignisse. Im laufenden Betrieb (Zustände SET/RESET) schaltet S nach SET, R nach RESET, und CLK schaltet jeweils in den *entgegengesetzten* Zustand (Toggle) — unabhängig vom vorherigen S/R-Aufruf. Jeder Übergang setzt Q entsprechend und löst EO aus.

Technische Besonderheiten

- **Kombiniertes Set/Reset/Toggle-Verhalten**: Anders als E_T_FF, das ausschließlich toggelt, kann Q hier sowohl gezielt über S/R gesetzt als auch über CLK umgeschaltet werden.
- **Symmetrisches Start-Up für alle drei Eingänge**: START reagiert auf S, R und CLK gleichermaßen definiert.
- **Kein Init-Mechanismus**: Für eine projektierbare Startwert-Initialisierung siehe E_T_FF_SR_SYM_INIT.

Zustandsübersicht

Zustand	Bedeutung
START	Anfangszustand, reagiert symmetrisch auf S, R, CLK
SET	Q = TRUE; R⟶RESET, CLK⟶RESET
RESET	Q = FALSE; S⟶SET, CLK⟶SET

Anwendungsszenarien

- **Handbetrieb mit Taktumschaltung**: Ein Zustand kann sowohl gezielt gesetzt/rückgesetzt (S/R, z. B. durch einen Bediener) als auch getaktet umgeschaltet werden (CLK, z. B. durch einen Timer), ohne zwei getrennte Bausteine zu benötigen.
- **Blinklogik mit Override**: Ein Blinksignal (CLK) kann jederzeit durch ein explizites S/R überschrieben werden.

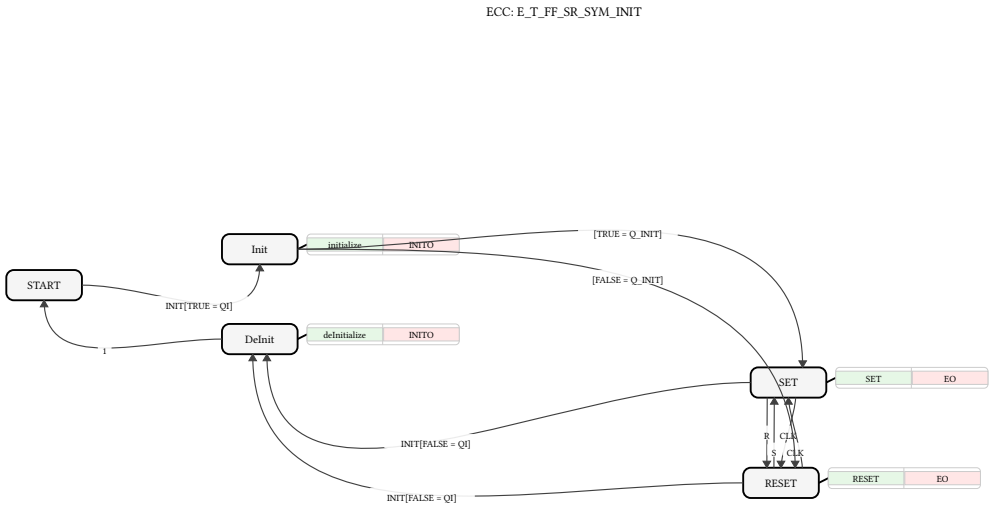
⚖️ Vergleich mit ähnlichen Bausteinen

- E_RS_SYM: dieselbe Set/Reset-Logik, aber ohne CLK-Toggle-Eingang.
- **E_T_FF**: reiner Toggle-Baustein ohne S/R.
- E_T_FF_SR_SYM_INIT: dieselbe Funktion, erweitert um INIT/INITO.

Fazit

E_T_FF_SR_SYM kombiniert gezieltes Set/Reset mit Toggle-Funktionalität in einem Baustein mit garantiert definiertem Startverhalten und eignet sich überall dort, wo beide Bedienarten gleichzeitig benötigt werden.

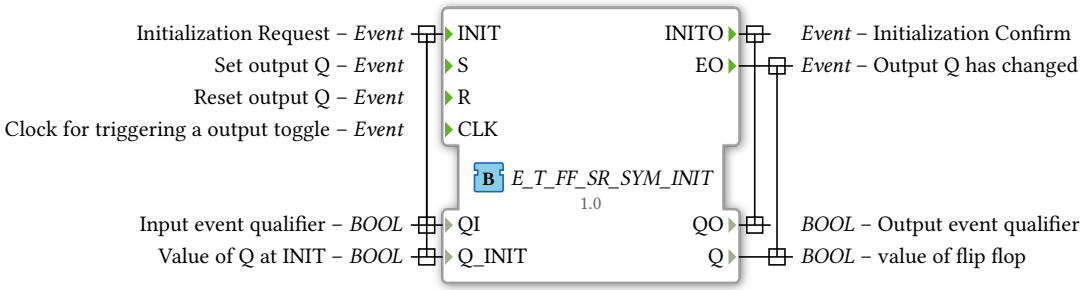
2.2.3.64 E_T_FF_SR_SYM_INIT



• – ○ • – ○ • – ○ •

Einleitung

Der `E_T_FF_SR_SYM_INIT` erweitert E_T_FF_SR_SYM um eine explizite `INIT`/`INITO`-Schnittstelle: Der Startwert von `Q` wird nicht durch das erste `S`-, `R`- oder `CLK`-Ereignis bestimmt, sondern gezielt über `Q_INIT` vorgegeben.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsanforderung, trägt `QI` und `Q_INIT`.
- **S (Set)**: Setzt `Q` auf `TRUE`.
- **R (Reset)**: Setzt `Q` auf `FALSE`.
- **CLK**: Kehrt den aktuellen Zustand von `Q` um.

Ereignis-Ausgänge

- **INITO**: Bestätigt die (De-)Initialisierung, trägt `QO`.
- **EO**: Wird nach jedem `S`-, `R`- oder `CLK`-Ereignis ausgelöst, trägt `Q`.

Daten-Eingänge

- **QI** (BOOL): Eingangs-Event-Qualifier — `TRUE` initialisiert, `FALSE` deinitialisiert.
- **Q_INIT** (BOOL): Der Wert, auf den `Q` beim Initialisieren gesetzt wird.

Daten-Ausgänge

- **QO** (BOOL): Ausgangs-Event-Qualifier.
- **Q** (BOOL): Der aktuelle Zustand.

Funktionsweise

`INIT` mit `QI = TRUE` initialisiert `Q` über `Q_INIT` (`Init` ⟶ `SET`/`RESET`). Im laufenden Betrieb schalten `S`/`R` gezielt, `CLK` toggelt zwischen `SET` und `RESET` — identisch zu E_T_FF_SR_SYM, jedoch ohne dessen symmetrisches Start-Up-Verhalten aus `START` heraus, da der Startwert stattdessen über `INIT` kommt. `INIT` mit `QI = FALSE` deinitialisiert den Baustein zurück in den `START`-Zustand.

Technische Besonderheiten

- **Kombiniert alle drei Mechanismen**: gezieltes Set/Reset (`S`/`R`), Toggle (`CLK`) und projektierbare Initialisierung (`INIT`/`Q_INIT`) in einem einzigen Baustein.
- **Gleiche INIT/DeInit-Struktur** wie E_RS_SYM_INIT, E_SR_SYM_INIT und E_T_FF_INIT.

Zustandsübersicht

Zustand	Bedeutung
START	Unkonfigurierter Anfangszustand
Init	Initialisierung läuft, `QO := QI`
DeInit	Deinitialisierung läuft, `QO := FALSE`
SET	`Q = TRUE`; `R`/`CLK`⟶RESET
RESET	`Q = FALSE`; `S`/`CLK`⟶SET

Anwendungsszenarien

- **Vollausgestattetes Bedien- und Blinkelement** mit definiertem Startwert: kombiniert manuelles Setzen/Rücksetzen, taktgesteuertes Umschalten und einen beim Hochfahren projektierbaren Anfangszustand.

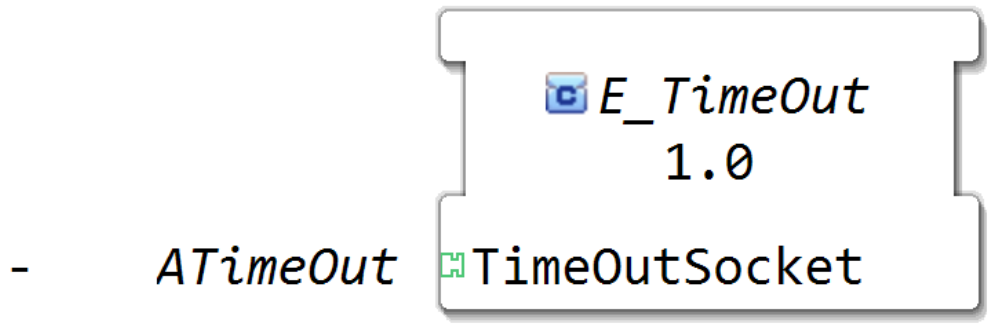
⚖️ Vergleich mit ähnlichen Bausteinen

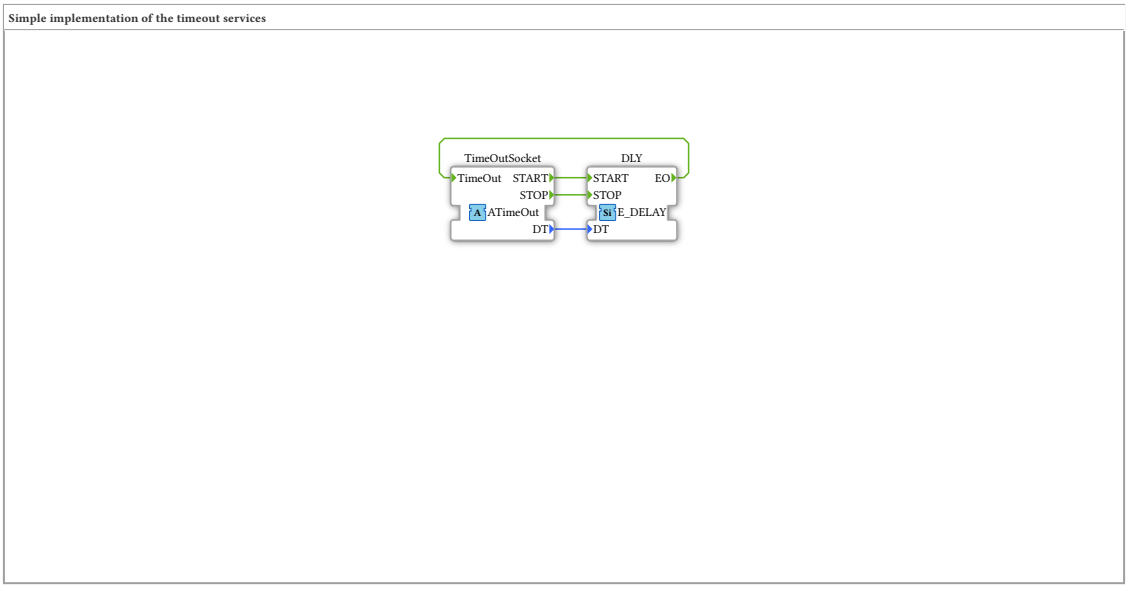
- E_T_FF_SR_SYM: dieselbe Funktion ohne `INIT`/`INITO`.
- E_T_FF_INIT: nur Toggle + Init, ohne `S`/`R`.
- E_RS_SYM_INIT / E_SR_SYM_INIT: nur Set/Reset + Init, ohne `CLK`.

Fazit

`E_T_FF_SR_SYM_INIT` ist der funktional umfangreichste Baustein der `E_*_SYM`-Familie und vereint Set/Reset, Toggle und projektierbare Initialisierung in einem Funktionsbaustein.

2.2.3.65 E_TimeOut

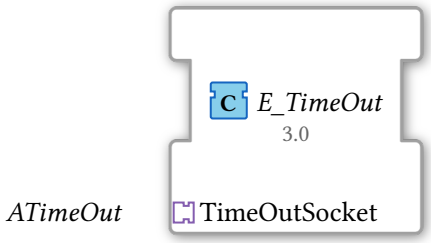




• – ○ • – ○ • – ○ •

Einleitung

Der **E_TimeOut** ist ein standardkonformer Funktionsbaustein (IEC 61499-1) zur Implementierung von Timeout-Services. Version 1.0 bietet eine einfache, aber effektive Timeout-Funktionalität durch interne Nutzung eines E_DELAY-Bausteins. Der **E_TimeOut** ist ein zusammengesetzter Funktionsblock. Innerhalb des Netzwerks eines zusammengesetzten Funktionsblocks wird jeder Adapter, der zu seiner Schnittstelle hinzugefügt wird, durch einen Adapterblock repräsentiert, der wie ein Funktionsblock aussieht. Die Schnittstellenelemente dieses Adapterblocks sind wie ein Funktionsblock verbunden.



Schnittstellenstruktur

Adapter-Schnittstelle (Socket-Perspektive)

Der Baustein verwendet einen **Socket** (Buchse) `TimeOutSocket` vom Typ `ATimeOut`. Da es sich um einen Socket handelt, sind die Signalrichtungen gegenüber der Adapter-Definition (Plug) invertiert:

- **Eingänge (vom Socket empfangen)**:
  - `START`: Startet den internen Timer.
  - `STOP`: Stoppt den internen Timer.
  - `DT` (TIME): Die zu verwendende Verzögerungszeit.
- **Ausgang (an den Socket gesendet)**:
  - `TimeOut`: Wird nach Ablauf der Zeit an den verbundenen Plug signalisiert.

Interne Komponenten

- `DLY` (E_DELAY): Kernkomponente für die Zeitsteuerung

Funktionsweise

1. **Timeout-Initialisierung**:
   - Bei `START`-Ereignis am Socket beginnt der Timer mit dem konfigurierten `DT`-Wert.
   - Ein weiteres `START`-Ereignis während der Timer läuft, wird ignoriert.

2. **Timeout-Abbruch**:
   - Ein `STOP`-Ereignis bricht den aktiven Timer sofort ab. Es wird kein `TimeOut`-Ereignis generiert.
3. **Timeout-Auslösung**:
   - Nach Ablauf von `DT` wird das `TimeOut`-Ereignis einmalig generiert.

Technische Besonderheiten

✓ **Adapter-basierte** Schnittstelle (`ATimeOut`). ✓ **Einfache, nicht-nachtriggerbare Timeout-Logik**. ✓ **Deterministisches** Zeitverhalten.

Anwendungsszenarien

- **Netzwerkkommunikation**: Überwachung auf eine Antwort innerhalb einer festen Frist. Wenn die Antwort kommt, wird der Timer per `STOP` abgebrochen.
- **Gerätesteuerung**: Einfache Watchdog-Funktionen, die nicht zurückgesetzt werden müssen.
- **Prozessüberwachung**: Sicherstellen, dass ein Prozessschritt eine maximale Dauer nicht überschreitet.

⚖️ **Vergleich mit E_RTimeOut**

Feature	E_TimeOut (dieser)	E_RTimeOut
Interner Baustein	E_DELAY	E_RDELAY
`START` bei lfd. Timer	Ignoriert	Startet Timer neu
Adaptertyp	ATimeOut	ARTimeOut

🛠️ **Zugehörige Übungen**

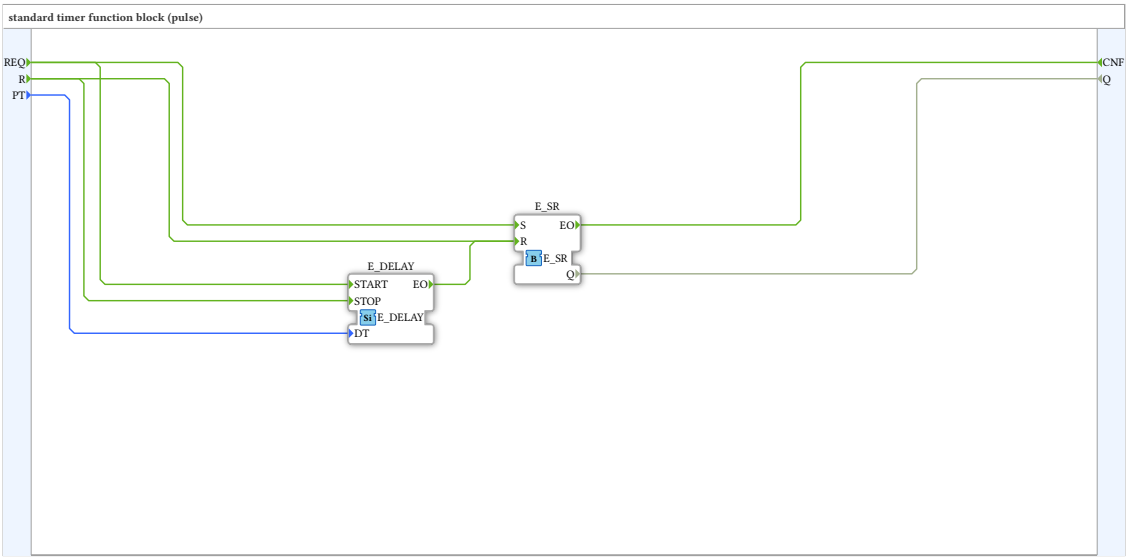
- Uebung_035
- Uebung_035a
- Uebung_035a1_AX
- Uebung_035a2
- Uebung_035a2_AX
- Uebung_035a3
- Uebung_035a3_AX
- Uebung_035c
- Uebung_036
- Uebung_037
- Uebung_038
- Uebung_038_AX
- Uebung_039
- Uebung_039a

Fazit

Der E_TimeOut-Baustein bietet eine robuste Grundimplementierung für nicht-nachtriggerbare Timeout-Anforderungen. Er ist ideal für Fälle, in denen ein Zeitablauf gestartet und entweder bis zum Ende durchlaufen oder explizit abgebrochen werden soll. Für Szenarien, die ein „Nachtriggern“ oder Zurücksetzen des Timers erfordern (wie bei einem Watchdog, der periodisch „getreten“ wird), ist der `E_RTimeOut`-Baustein die bessere Wahl.

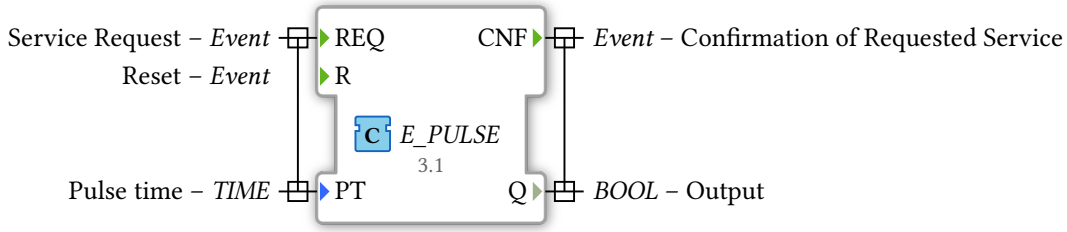
2.2.3.66 timers

E_PULSE



Wichtiger Hinweis: Dieser Baustein benötigt nur ein Ereignis (Event) und keine zyklischen Aufrufe. Er besitzt keinen Ausgang ET und zeigt die verstrichene Zeit nicht an.

• – ◦ • – ◦ • – ◦ •



Folgender Text wurde erzeugt von https://chat.deepseek.com/ am 05.02.2025 und noch nicht korrekturgelesen (HF)

Der Baustein E_PULSE in der IEC 61499: Ein Überblick

Einleitung

Die IEC 61499 ist ein internationaler Standard, der die Modellierung von verteilten industriellen Steuerungssystemen ermöglicht. Ein zentrales Konzept in diesem Standard ist der Funktionsbaustein (FB), der als grundlegende Komponente für die Steuerungslogik dient. Der Baustein **E_PULSE** (Event-driven Pulse) ist ein Beispiel für einen solchen Funktionsbaustein, der in der IEC 61499 verwendet wird, um einen Impuls mit einer bestimmten Dauer zu erzeugen. Dieser Aufsatz beschreibt die Struktur, das Verhalten und die Anwendung des E_PULSE-Bausteins anhand des bereitgestellten Quelltextes. Ein besonderer Aspekt, der hervorgehoben werden soll, ist, dass dieser Baustein **auch dann funktioniert, wenn er nicht zyklisch aufgerufen wird**, da er intern auf dem **E_DELAY**-Baustein aus der IEC 61499 basiert.

Struktur des E_PULSE-Bausteins

Der E_PULSE-Baustein ist ein **Composite Function Block (CFB)**, der gemäß der IEC 61499 spezifiziert ist. Ein CFB besteht aus einer Schnittstelle (Interface) und einem internen Netzwerk von Funktionsbausteinen, die zusammenarbeiten, um das gewünschte Verhalten zu realisieren.

Schnittstelle (Interface)

Die Schnittstelle des E_PULSE-Bausteins besteht aus:

- **Eingangsereignisse (Event Inputs)**:
  - **REQ (Request)**: Dieses Ereignis löst die Ausführung des Bausteins aus. Es ist mit der Eingangsvariablen **PT** (Pulse Time) verknüpft.

  - **R (Reset)**: Dieses Ereignis setzt den Timer zurück.
- **Ausgangsereignis (Event Output)**:
  - **CNF (Confirmation)**: Dieses Ereignis wird ausgelöst, wenn die Ausführung des Bausteins abgeschlossen ist. Es ist mit der Ausgangsvariablen **Q** verknüpft.
- **Eingangsvariable (Input Variable)**:
  - **PT (Pulse Time)**: Dies ist eine Zeitvariable (TIME), die die Dauer des Impulses definiert.
- **Ausgangsvariable (Output Variable)**:
  - **Q (Output)**: Dies ist eine boolesche Variable, die den Zustand des Impulses anzeigt. Sie ist **TRUE**, solange der Impuls aktiv ist.

Internes Netzwerk (FBNetwork)

Der E_PULSE-Baustein verwendet intern zwei Funktionsbausteine, um das gewünschte Verhalten zu realisieren:

1. **E_DELAY**: Dieser Baustein realisiert die Verzögerungszeit **PT**. Wenn das **START**-Ereignis eintritt, beginnt der Timer zu laufen. Wenn die Verzögerungszeit abgelaufen ist, wird das **EO**-Ereignis ausgelöst.
2. **E_SR**: Dieser Baustein ist ein bistabiles Element, das den Zustand des Impulses speichert. Wenn das **S**-Ereignis eintritt, wird der Ausgang **Q** auf **TRUE** gesetzt. Wenn das **R**-Ereignis eintritt, wird der Ausgang **Q** auf **FALSE** gesetzt.

Verhalten des E_PULSE-Bausteins

Der E_PULSE-Baustein verhält sich wie ein Impulsgeber, der einen Impuls mit einer bestimmten Dauer **PT** erzeugt. Das Verhalten des Bausteins kann wie folgt beschrieben werden:

1. **Start des Impulses**: Wenn das **REQ**-Ereignis eintritt, wird der Timer gestartet, und der Ausgang **Q** wird auf **TRUE** gesetzt.
2. **Impulsdauer**: Der Ausgang **Q** bleibt solange **TRUE**, bis die Verzögerungszeit **PT** abgelaufen ist.
3. **Ende des Impulses**: Sobald die Verzögerungszeit **PT** abgelaufen ist, wird der Ausgang **Q** auf **FALSE** gesetzt, und das **CNF**-Ereignis wird ausgelöst, um den Abschluss des Impulses zu signalisieren.
4. **Reset**: Wenn das **R**-Ereignis eintritt, wird der Timer zurückgesetzt, und der Ausgang **Q** wird auf **FALSE** gesetzt.

Unabhängigkeit vom zyklischen Aufruf

Ein entscheidender Vorteil des E_PULSE-Bausteins ist, dass er **nicht zyklisch aufgerufen werden muss**, um korrekt zu funktionieren. Dies liegt daran, dass der Baustein intern auf dem **E_DELAY**-Baustein basiert, der in der IEC 61499 spezifiziert ist. Der **E_DELAY**-Baustein ist ein ereignisgesteuerter Timer, der unabhängig vom zyklischen Aufruf des übergeordneten Bausteins arbeitet. Das bedeutet, dass der E_PULSE-Baustein auch in Systemen eingesetzt werden kann, die nicht zyklisch arbeiten, z.B. in ereignisgesteuerten oder verteilten Steuerungssystemen.

Anwendungsbeispiele

Der E_PULSE-Baustein kann in verschiedenen industriellen Steuerungsanwendungen eingesetzt werden, insbesondere in Szenarien, in denen ein Impuls mit einer bestimmten Dauer erforderlich ist. Einige Beispiele sind:

- **Maschinensteuerung**: Der Baustein kann verwendet werden, um einen kurzen Impuls zu erzeugen, der eine Maschine für eine bestimmte Zeit aktiviert, z.B. um einen Prozess zu starten oder zu stoppen.
- **Beleuchtungssteuerung**: Der Baustein kann in Beleuchtungssystemen eingesetzt werden, um das Licht für eine bestimmte Zeit einzuschalten, z.B. für eine Sicherheitsbeleuchtung.
- **Ventilsteuerung**: Der Baustein kann verwendet werden, um ein Ventil für eine bestimmte Zeit zu öffnen, z.B. um eine bestimmte Menge an Flüssigkeit oder Gas freizugeben.

2.3

🛠️ Zugehörige Übungen

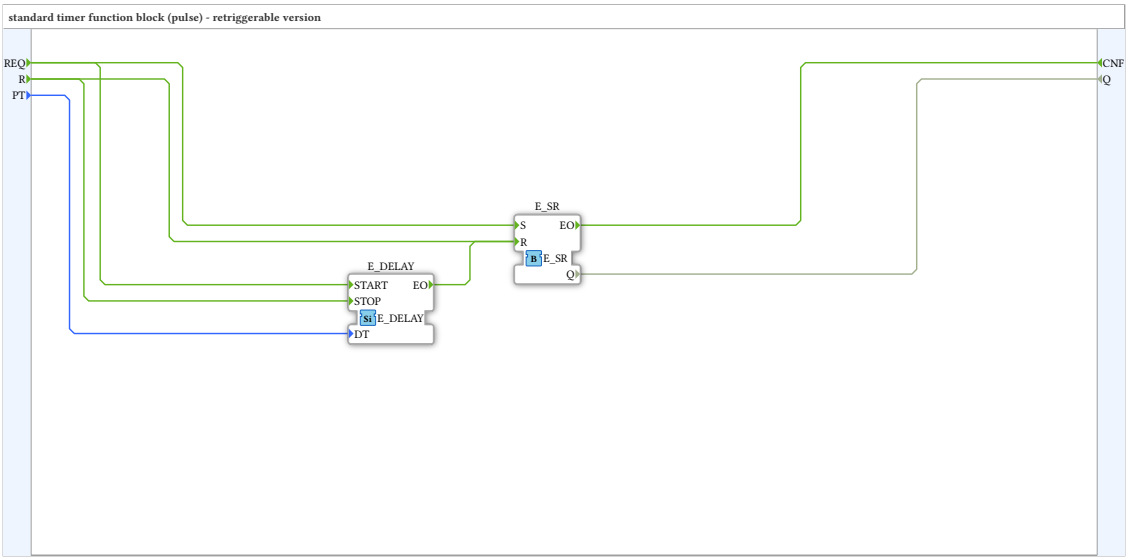
- Uebung_020h
- Uebung_020i

Fazit

Der E_PULSE-Baustein ist ein vielseitiger Funktionsbaustein in der IEC 61499, der einen Impuls mit einer bestimmten Dauer realisiert. Durch seine interne Verwendung des **E_DELAY**-Bausteins kann er **auch dann korrekt funktionieren, wenn er nicht zyklisch aufgerufen wird**. Dies macht ihn besonders geeignet für ereignisgesteuerte und verteilte Steuerungssysteme, in denen zyklische Aufrufe nicht immer gewährleistet sind.

Der bereitgestellte Quelltext zeigt, wie der E_PULSE-Baustein in der Praxis implementiert werden kann, und bietet eine solide Grundlage für die Entwicklung ähnlicher Bausteine in industriellen Steuerungssystemen. Die Unabhängigkeit vom zyklischen Aufruf ist dabei ein entscheidender Vorteil, der den Baustein für eine Vielzahl von Anwendungen geeignet macht.

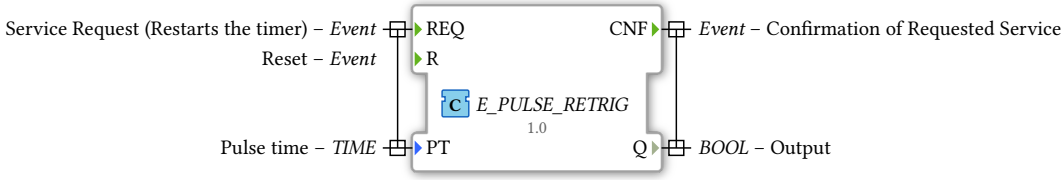
E_PULSE_RETRIG



• — ○ • — ○ • — ○ •

Einleitung

Der `E_PULSE_RETRIG` ist die retriggerbare Variante von `E_PULSE`: Er erzeugt einen `TRUE`-Impuls der Dauer `PT` an `Q`, startet den Impuls jedoch bei jedem erneuten `REQ`-Ereignis von vorne, statt ein laufendes Intervall zu ignorieren.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet (bzw. re-triggert) den Impuls, trägt `PT`.
- **R**: Bricht den laufenden Impuls sofort ab.

Ereignis-Ausgänge

- **CNF**: Bestätigung, trägt `Q`.

Daten-Eingänge

- **PT** (TIME): Dauer des Impulses.

Daten-Ausgänge

- **Q** (BOOL): `TRUE` während der Impulsdauer, sonst `FALSE`.

Funktionsweise

Jedes REQ-Ereignis setzt gleichzeitig E_SR.S (⟶ Q := TRUE) und startet E_DELAY neu mit der Zeit PT — ein erneutes REQ während eines laufenden Impulses startet den E_DELAY-Timer also von vorne, statt zu warten, bis der ursprüngliche Impuls abgelaufen ist. Läuft E_DELAY ab, wird E_SR.R ausgelöst (Q := FALSE). Ein R-Ereignis stoppt den Timer sofort und setzt Q direkt zurück auf FALSE.

Technische Besonderheiten

- **Retriggerbar**: Der entscheidende Unterschied zu E_PULSE (nicht retriggerbar) liegt darin, dass jedes REQ den E_DELAY-Timer neu startet — ein Impuls kann so durch wiederholte REQ-Ereignisse beliebig verlängert werden.
- **Zusammengesetzt aus E_DELAY + E_SR**: reine Verdrahtung, kein eigenes ECC.

Zustandsübersicht

Zustandslose Verdrahtungslogik: E_SR hält den aktuellen Q-Zustand, E_DELAY läuft bei jedem REQ neu an.

Anwendungsszenarien

- **Totzeit-/Watchdog-Logik**: Q soll TRUE bleiben, solange regelmäßig REQ-Ereignisse eintreffen (z. B. Lebenszeichen eines Sensors), und erst FALSE werden, wenn PT seit dem letzten REQ ohne weiteres REQ verstrichen ist.
- **Verlängerbare Impulse**, z. B. eine Beleuchtung, die bei jeder erneuten Bewegungserkennung (REQ) ihre Leuchtdauer neu beginnt, statt abzulaufen.

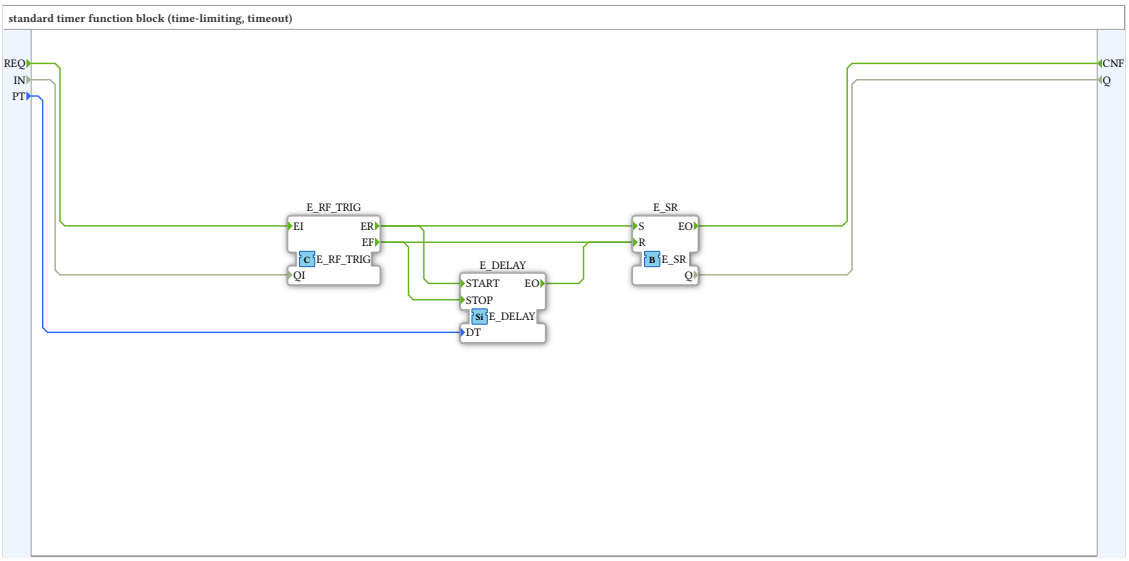
⚖️ Vergleich mit ähnlichen Bausteinen

- **E_PULSE**: dieselbe Grundfunktion, aber nicht retriggerbar (REQ während eines laufenden Impulses hat keine Wirkung).
- E_TP_RETRIG: retriggerbarer Impuls, ausgelöst über eine steigende Flanke von IN statt über ein separates REQ-Ereignis.

Fazit

E_PULSE_RETRIG liefert einen verlängerbaren TRUE-Impuls, dessen Dauer bei jedem REQ-Ereignis neu zu laufen beginnt, und eignet sich für Watchdog- und Totzeit-Anwendungen, bei denen ein laufender Impuls durch wiederholte Anfragen am Leben gehalten werden soll.

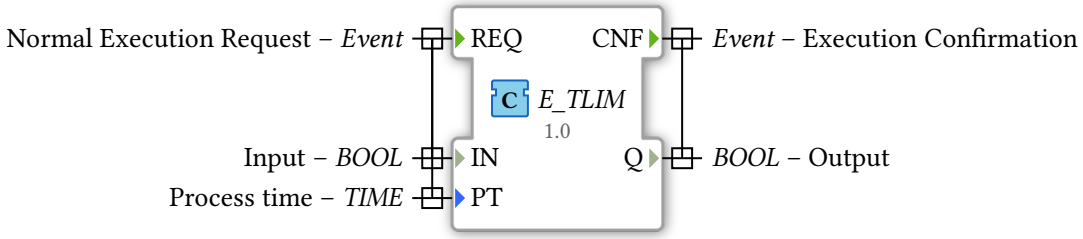
E_TLIM



• – ○ • – ○ • – ○ •

Einleitung

Der `E_TLIM` (Time-Limiting) begrenzt, wie lange `Q` maximal `TRUE` bleiben kann: Sobald `IN` auf `TRUE` wechselt, wird `Q` ebenfalls `TRUE`; bleibt `IN` länger als `PT` auf `TRUE`, schaltet `Q` nach Ablauf von `PT` automatisch auf `FALSE` (Timeout). Fällt `IN` vor Ablauf von `PT` bereits zurück auf `FALSE`, folgt `Q` sofort.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Auslösung, trägt `IN` und `PT`.

Ereignis-Ausgänge

- **CNF**: Bestätigung, trägt `Q`.

Daten-Eingänge

- **IN** (BOOL): Der zu überwachende Eingangszustand.
- **PT** (TIME): Maximale Zeit, die `Q` nach einer steigenden Flanke von `IN` `TRUE` bleiben darf.

Daten-Ausgänge

- **Q** (BOOL): `TRUE`, solange `IN` `TRUE` ist und `PT` noch nicht abgelaufen ist.

Funktionsweise

`E_RF_TRIG` erkennt Flankenwechsel von `IN`: Eine steigende Flanke (`ER`) startet `E_DELAY` (mit `PT`) und setzt `E_SR.S` (⟶ `Q := TRUE`). Eine fallende Flanke (`EF`) stoppt `E_DELAY` sofort und setzt `E_SR.R` (⟶ `Q := FALSE`) — `IN` gewinnt also immer gegenüber dem Timeout. Läuft stattdessen `E_DELAY` vollständig ab (weil `IN` länger als `PT` `TRUE` blieb), löst dies ebenfalls `E_SR.R` aus, wodurch `Q` auch ohne fallende Flanke von `IN` auf `FALSE` zurückfällt (Timeout-Fall).

Technische Besonderheiten

- **Zwei Wege zu `Q := FALSE`**: entweder durch eine fallende Flanke von `IN` (regulärer Fall) oder durch Ablauf von `PT` (Timeout-Fall) — beide führen über `E_SR.R`.
- **Zusammengesetzt aus drei Standardbausteinen**: `E_RF_TRIG` (Flankenerkennung) + `E_DELAY` (Timeout-Timer) + `E_SR` (Zustandsspeicher), keine eigene ECC.

Zustandsübersicht

Zustand (implizit über `E_SR`/`E_DELAY`)	Bedeutung
Q = FALSE, Timer inaktiv	Ruhezustand, `IN = FALSE`
Q = TRUE, Timer läuft	`IN` ist `TRUE`, `PT` noch nicht abgelaufen
Q = FALSE (Timeout)	`IN` war länger als `PT` `TRUE`, automatisch zurückgesetzt

Anwendungsszenarien

- **Überwachung von Maximaldauern**: Ein Signal darf höchstens `PT` lang aktiv sein (z. B. maximale Einschaltdauer eines Aktors als Sicherheitsbegrenzung); bleibt `IN` länger aktiv, wird `Q` automatisch zurückgesetzt.
- **Timeout-Überwachung von Prozessschritten**: `IN` markiert den laufenden Schritt, `Q` zeigt an, ob der Schritt noch innerhalb der zulässigen Zeit `PT` liegt.

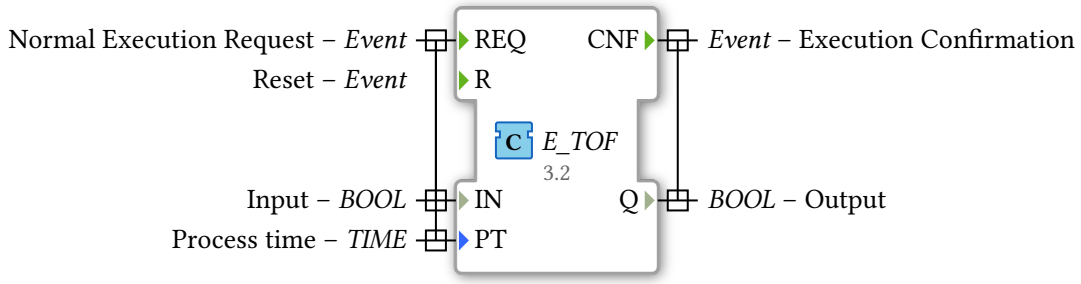
⚖️ Vergleich mit ähnlichen Bausteinen

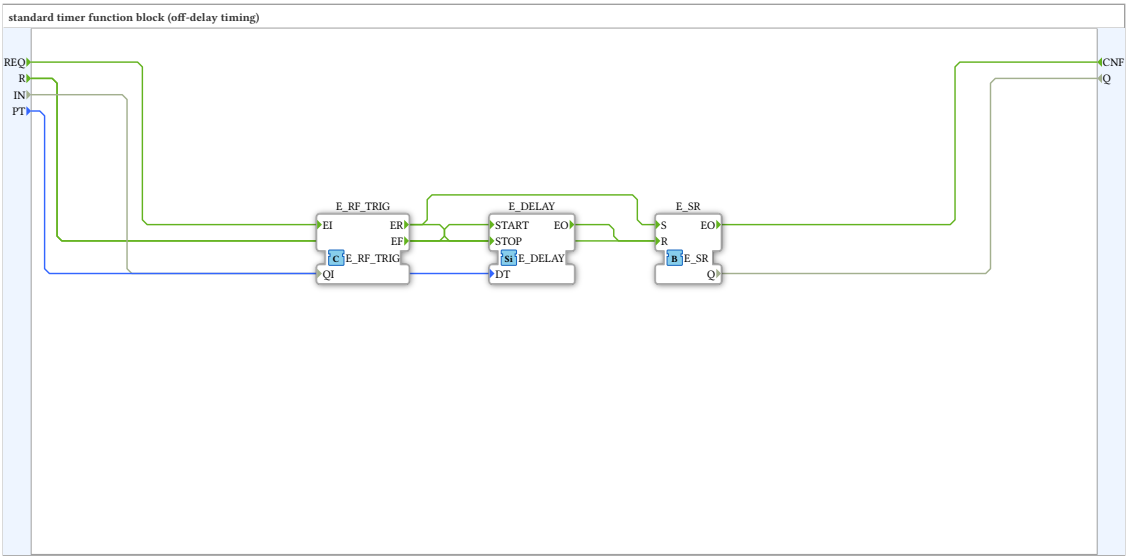
- **`E_TON`/`E_TOF`** (Standard-IEC-61131-3-Timer): verzögern das *Einschalten* bzw. *Ausschalten* von `Q` relativ zu `IN`, begrenzen aber nicht dessen maximale Ein-Dauer.
- **E_PULSE_RETRIG** / **E_TP_RETRIG**: erzeugen einen Impuls fester Dauer, unabhängig davon, wie lange `IN`/`REQ` tatsächlich aktiv bleibt.
- **`E_RF_TRIG`**: die intern verwendete Flankenerkennung.

Fazit

`E_TLIM` begrenzt die maximale Ein-Dauer eines `BOOL`-Signals und eignet sich als einfache Zeitüberwachung für Aktoren oder Prozessschritte, bei denen ein zu langes Verweilen im `TRUE`-Zustand automatisch beendet werden soll.

E_TOF





🎧 Podcast

- E_TOF vs. FB_TOF: Der Event-Timer, der nicht zyklisch tickt – Revolution für Automatisierungssysteme?
- FB_TOF und E_TOF: Verzögerungstimer in IEC 61131-3 und 61499

Beschreibung

Der Baustein E_TOF in der IEC 61499: Ein Überblick

Einleitung

Die IEC 61499 ist ein internationaler Standard, der die Modellierung von verteilten industriellen Steuerungssystemen ermöglicht. Ein zentrales Konzept in diesem Standard ist der Funktionsbaustein (FB), der als grundlegende Komponente für die Steuerungslogik dient. Der Baustein **E_TOF** (Event-driven Off-Delay Timer) ist ein Beispiel für einen solchen Funktionsbaustein, der in der IEC 61499 verwendet wird, um eine verzögerte Abschaltfunktion zu modellieren. Dieser Aufsatz beschreibt die Struktur, das Verhalten und die Anwendung des E_TOF-Bausteins anhand des bereitgestellten Quelltextes. Ein besonderer

Aspekt, der hervorgehoben werden soll, ist, dass dieser Baustein **auch dann funktioniert, wenn er nicht zyklisch aufgerufen wird**, da er intern auf dem **E_DELAY**-Baustein aus der IEC 61499 basiert.

Struktur des E_TOF-Bausteins

Der E_TOF-Baustein ist ein **Composite Function Block (CFB)**, der gemäß der IEC 61499 spezifiziert ist. Ein CFB besteht aus einer Schnittstelle (Interface) und einem internen Netzwerk von Funktionsbausteinen, die zusammenarbeiten, um das gewünschte Verhalten zu realisieren.

Schnittstelle (Interface)

Die Schnittstelle des E_TOF-Bausteins besteht aus:

- **Eingangsereignisse (Event Inputs)**:
  - **REQ (Request)**: Dieses Ereignis löst die Ausführung des Bausteins aus. Es ist mit den Eingangsvariablen **IN** und **PT** verknüpft.
  - **R (Reset)**: Dieses Ereignis setzt den Timer zurück.
- **Ausgangsereignis (Event Output)**:
  - **CNF (Confirmation)**: Dieses Ereignis wird ausgelöst, wenn die Ausführung des Bausteins abgeschlossen ist. Es ist mit der Ausgangsvariable **Q** verknüpft.
- **Eingangsvariablen (Input Variables)**:
  - **IN (Input)**: Dies ist eine boolesche Variable, die den Zustand des Timers steuert. Wenn **IN** auf **TRUE** gesetzt ist, startet der Timer. Wenn **IN** auf **FALSE** gesetzt wird, beginnt die Verzögerungszeit.
  - **PT (Process Time)**: Dies ist eine Zeitvariable (TIME), die die Verzögerungszeit definiert, nach der der Ausgang **Q** auf **FALSE** gesetzt wird.
- **Ausgangsvariable (Output Variable)**:
  - **Q (Output)**: Dies ist eine boolesche Variable, die den Zustand des Timers anzeigt. Sie bleibt solange **TRUE**, bis die Verzögerungszeit **PT** abgelaufen ist.

Internes Netzwerk (FBNetwork)

Der E_TOF-Baustein verwendet intern drei Funktionsbausteine, um das gewünschte Verhalten zu realisieren:

1. **E_SWITCH**: Dieser Baustein steuert den Fluss der Ereignisse basierend auf dem Eingang **IN**. Wenn **IN** auf **TRUE** gesetzt ist, wird das Ereignis an **E_RS.S** weitergeleitet. Wenn **IN** auf **FALSE** gesetzt ist, wird das Ereignis an **E_DELAY.START** weitergeleitet.
2. **E_DELAY**: Dieser Baustein realisiert die Verzögerungszeit **PT**. Wenn das **START**-Ereignis eintritt, beginnt der Timer zu laufen. Wenn die Verzögerungszeit abgelaufen ist, wird das **EO**-Ereignis ausgelöst.
3. **E_RS**: Dieser Baustein ist ein bistabiles Element, das den Zustand des Timers speichert. Wenn das **S**-Ereignis eintritt, wird der Ausgang **Q** auf **TRUE** gesetzt. Wenn das **R**-Ereignis eintritt, wird der Ausgang **Q** auf **FALSE** gesetzt.

Verhalten des E_TOF-Bausteins

Der E_TOF-Baustein verhält sich wie ein Off-Delay-Timer, der eine verzögerte Abschaltfunktion realisiert. Das Verhalten des Bausteins kann wie folgt beschrieben werden:

1. **Start des Timers**: Wenn das **REQ**-Ereignis eintritt und die Eingangsvariable **IN** auf **TRUE** gesetzt ist, wird der Timer gestartet. Der Ausgang **Q** wird auf **TRUE** gesetzt.
2. **Verzögerungszeit**: Wenn die Eingangsvariable **IN** auf **FALSE** gesetzt wird, beginnt die Verzögerungszeit **PT**. Der Ausgang **Q** bleibt solange **TRUE**, bis die Verzögerungszeit abgelaufen ist.
3. **Ablauf der Verzögerungszeit**: Sobald die Verzögerungszeit **PT** abgelaufen ist, wird der Ausgang **Q** auf **FALSE** gesetzt, und das **CNF**-Ereignis wird ausgelöst, um den Abschluss der Verzögerung zu signalisieren.
4. **Reset**: Wenn das **R**-Ereignis eintritt, wird der Timer zurückgesetzt, und der Ausgang **Q** wird auf **FALSE** gesetzt.

Unabhängigkeit vom zyklischen Aufruf

Ein entscheidender Vorteil des E_TOF-Bausteins ist, dass er **nicht zyklisch aufgerufen werden muss**, um korrekt zu funktionieren. Dies liegt daran, dass der Baustein intern auf dem **E_DELAY**-Baustein

basiert, der in der IEC 61499 spezifiziert ist. Der **E_DELAY**-Baustein ist ein ereignisgesteuerter Timer, der unabhängig vom zyklischen Aufruf des übergeordneten Bausteins arbeitet. Das bedeutet, dass der E_TOF-Baustein auch in Systemen eingesetzt werden kann, die nicht zyklisch arbeiten, z.B. in ereignisgesteuerten oder verteilten Steuerungssystemen.

Anwendungsbeispiele

Der E_TOF-Baustein kann in verschiedenen industriellen Steuerungsanwendungen eingesetzt werden, insbesondere in Szenarien, in denen eine verzögerte Abschaltfunktion erforderlich ist. Einige Beispiele sind:

- **Maschinensteuerung**: Der Baustein kann verwendet werden, um sicherzustellen, dass eine Maschine nach dem Ausschalten noch für eine bestimmte Zeit weiterläuft, z.B. um Prozesse abzuschließen oder Sicherheitsvorkehrungen zu treffen.
- **Beleuchtungssteuerung**: Der Baustein kann in Beleuchtungssystemen eingesetzt werden, um das Licht nach dem Ausschalten des Schalters noch für eine bestimmte Zeit brennen zu lassen.
- **Ventilsteuerung**: Der Baustein kann verwendet werden, um ein Ventil nach dem Schließen noch für eine bestimmte Zeit offen zu halten, um z.B. Druck abzulassen.

2.4

🛠️ Zugehörige Übungen

- Uebung_020e

Fazit

Der E_TOF-Baustein ist ein vielseitiger Funktionsbaustein in der IEC 61499, der eine verzögerte Abschaltfunktion realisiert. Durch seine interne Verwendung des **E_DELAY**-Bausteins kann er **auch dann korrekt funktionieren, wenn er nicht zyklisch aufgerufen wird**. Dies macht ihn besonders geeignet für ereignisgesteuerte und verteilte Steuerungssysteme, in denen zyklische Aufrufe nicht immer gewährleistet sind.

Der bereitgestellte Quelltext zeigt, wie der E_TOF-Baustein in der Praxis implementiert werden kann, und bietet eine solide Grundlage für die Entwicklung ähnlicher Bausteine in industriellen Steuerungssystemen. Die Unabhängigkeit vom zyklischen Aufruf ist dabei ein entscheidender Vorteil, der den Baustein für eine Vielzahl von Anwendungen geeignet macht.

⚖️ Vergleich E_TOF zu FB_TOF

Kapitel 1: Der FB_TOF-Baustein

Der **FB_TOF** (Off-Delay Timer) ist ein standardisierter Funktionsbaustein gemäß der IEC 61131-3, der eine verzögerte Abschaltfunktion realisiert. Ein wesentliches Merkmal dieses Bausteins ist der Ausgang **ET** (Elapsed Time), der die verstrichene Zeit seit dem Start der Verzögerung anzeigt. Dieser Ausgang ist besonders nützlich, um den Fortschritt der Verzögerungszeit zu überwachen und in Steuerungslogiken zu verwenden.

Ein kritischer Aspekt des FB_TOF-Bausteins ist jedoch, dass er **zyklisch aufgerufen werden muss**, um korrekt zu funktionieren. Das bedeutet, dass der Baustein in jedem Zyklus des SPS-Programms ausgeführt werden muss, um die verstrichene Zeit **ET** korrekt zu berechnen und die Verzögerungszeit **PT** einzuhalten. Wenn der Baustein nicht regelmäßig aufgerufen wird, kann der Timer nicht korrekt arbeiten, und die Verzögerungszeit wird möglicherweise nicht eingehalten. Diese Abhängigkeit vom zyklischen Aufruf macht den FB_TOF-Baustein ideal für traditionelle SPS-Systeme, die in festen Zyklen arbeiten.

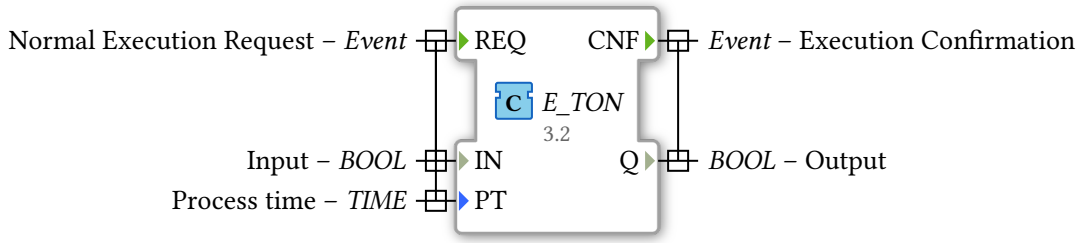
Kapitel 2: Der E_TOF-Baustein

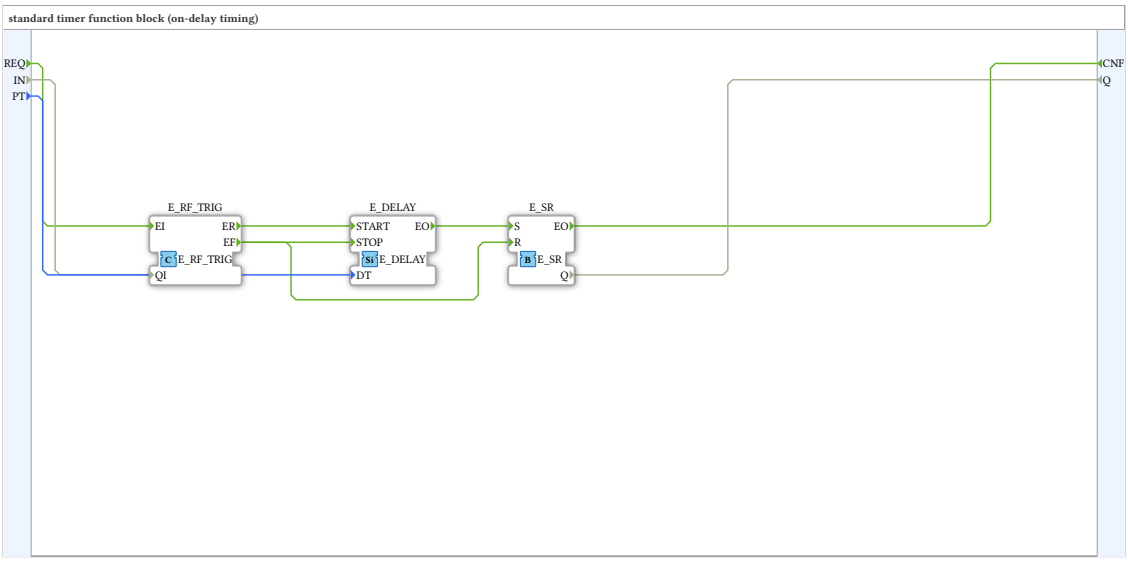
Der **E_TOF** (Event-driven Off-Delay Timer) ist ein Funktionsbaustein gemäß der IEC 61499, der ebenfalls eine verzögerte Abschaltfunktion realisiert. Im Gegensatz zum FB_TOF besitzt der E_TOF **keinen Ausgang ET**, der die verstrichene Zeit anzeigt. Stattdessen basiert der E_TOF intern auf dem **E_DELAY**-Baustein, der in der IEC 61499 spezifiziert ist und eine ereignisgesteuerte Verzögerungsfunktion bietet.

Ein entscheidender Vorteil des E_TOF-Bausteins ist, dass er **nicht zyklisch aufgerufen werden muss**, um korrekt zu funktionieren. Da der interne **E_DELAY**-Baustein unabhängig vom zyklischen Aufruf des übergeordneten Bausteins arbeitet, kann der E_TOF auch in Systemen eingesetzt werden, die nicht zyklisch arbeiten, z.B. in ereignisgesteuerten oder verteilten Steuerungssystemen. Dies macht den E_TOF-Baustein besonders flexibel und geeignet für moderne, verteilte Steuerungsarchitekturen, bei denen zyklische Aufrufe nicht immer gewährleistet sind.

Zusammenfassend lässt sich sagen, dass der FB_TOF durch seinen Ausgang **ET** eine detaillierte Überwachung der verstrichenen Zeit ermöglicht, jedoch einen zyklischen Aufruf erfordert. Der E_TOF hingegen verzichtet auf den Ausgang **ET**, bietet dafür aber die Flexibilität, unabhängig von zyklischen Aufrufen zu arbeiten, was ihn ideal für ereignisgesteuerte Systeme macht.

E_TON





🎧 Podcast

- E_TON in der Industrieautomation: Wie ein simpler Timer Sicherheit und Stabilität schafft

Text

Der Baustein **E_TON** ist ein standardisierter Funktionsbaustein-Typ (FBType) gemäß der Norm **IEC 61499**, der für **On-Delay-Timer** (Einschaltverzögerung) in industriellen Automatisierungssystemen verwendet wird. Dieser Baustein ermöglicht es, ein Signal zeitlich verzögert einzuschalten, was in vielen Steuerungsanwendungen erforderlich ist, um beispielsweise Maschinen oder Prozesse sicher zu steuern.

Allgemeine Beschreibung

- **Name**: E_TON
- **Typ**: Standard Timer Function Block (On-Delay Timing)
- **Zweck**: Der Baustein wird verwendet, um eine zeitverzögerte Einschaltfunktion zu realisieren. Er ist besonders nützlich in Anwendungen, bei denen eine Verzögerung beim Einschalten eines Signals erforderlich ist.
- **Lizenz**: Der Baustein ist unter der Eclipse Public License 2.0 (EPL-2.0) lizenziert.

Interface-Beschreibung

Eingangsereignisse (Event Inputs):

1. **REQ** (Normal Execution Request):
   - **Typ**: Event
   - **Beschreibung**: Dieses Ereignis triggert die normale Ausführung des Bausteins. Es wird verwendet, um den Timer zu starten oder zu aktualisieren.

- **Zugehörige Variablen**:
  - **IN**: Der Eingangswert, der den Timer startet.
  - **PT**: Die vorgegebene Zeitdauer für die Einschaltverzögerung (On-Delay).

Ausgangsereignisse (Event Outputs):

1. **CNF** (Execution Confirmation):
   - **Typ**: Event
   - **Beschreibung**: Dieses Ereignis bestätigt die Ausführung des Bausteins. Es wird ausgelöst, wenn der Timer abgelaufen ist oder ein Zustandswechsel stattgefunden hat.
   - **Zugehörige Variable**:
     - **Q**: Der Ausgangswert, der den Zustand des Timers anzeigt.

Eingangsvariablen (Input Variables):

1. **IN**:
   - **Typ**: BOOL
   - **Beschreibung**: Der Eingangswert, der den Timer startet. Wenn **IN** auf `TRUE` gesetzt wird, startet die Einschaltverzögerung (**PT**). Der Ausgang **Q** wird erst nach Ablauf der Zeit **PT** auf `TRUE` gesetzt.
2. **PT**:
   - **Typ**: TIME
   - **Beschreibung**: Die vorgegebene Zeitdauer für die Einschaltverzögerung (On-Delay). Diese Zeit wird ab dem Moment gemessen, in dem **IN** auf `TRUE` gesetzt wird.

Ausgangsvariablen (Output Variables):

1. **Q**:
   - **Typ**: BOOL
   - **Beschreibung**: Der Ausgangswert, der den Zustand des Timers anzeigt. **Q** wird auf `TRUE` gesetzt, nachdem die Einschaltverzögerung (**PT**) abgelaufen ist.

Funktionsweise

1. **Einschaltverzögerung (On-Delay)**:
   - Wenn **IN** auf `TRUE` gesetzt wird, startet der Timer für die Einschaltverzögerung (**PT**).
   - Nach Ablauf von **PT** wird **Q** auf `TRUE` gesetzt.
   - Wenn **IN** während des Timers auf `FALSE` gesetzt wird, wird der Timer gestoppt, und **Q** bleibt auf `FALSE`.
2. **Bestätigung der Ausführung (CNF)**:
   - Das Ereignis **CNF** wird ausgelöst, wenn der Timer abgelaufen ist und **Q** auf `TRUE` gesetzt wird.

Anwendungsbeispiele

- **Maschinensteuerung**: Verzögerte Reaktion auf Sensoren oder Schaltbefehle, z. B. um ein Überlasten von Motoren zu vermeiden.
- **Prozessautomatisierung**: Zeitgesteuerte Schaltvorgänge in industriellen Prozessen, z. B. in Förderbändern oder Heizungssteuerungen.

Zusammenfassung

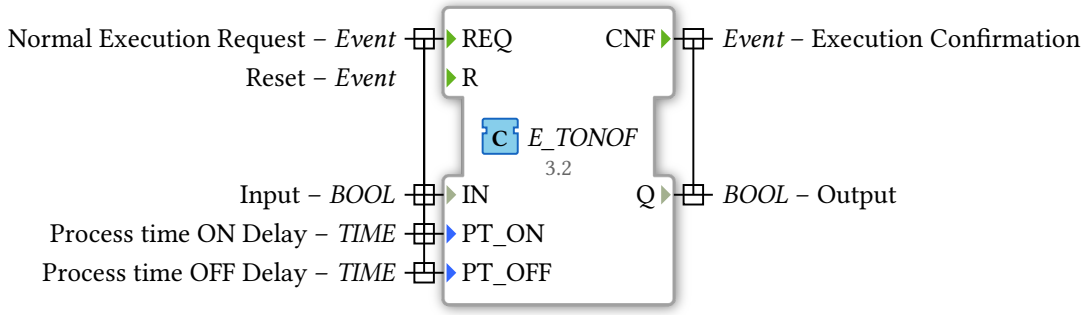
Der **E_TON**-Baustein ist ein einfaches, aber leistungsstarkes Werkzeug in der IEC 61499, um zeitverzögerte Einschaltvorgänge in industriellen Automatisierungssystemen zu realisieren. Er bietet eine zuverlässige Möglichkeit, Signale mit einer definierten Verzögerung einzuschalten, was in vielen Steuerungsanwendungen unerlässlich ist. Der Baustein ist einfach zu konfigurieren und kann in einer Vielzahl von Szenarien eingesetzt werden, bei denen eine zeitliche Verzögerung erforderlich ist.

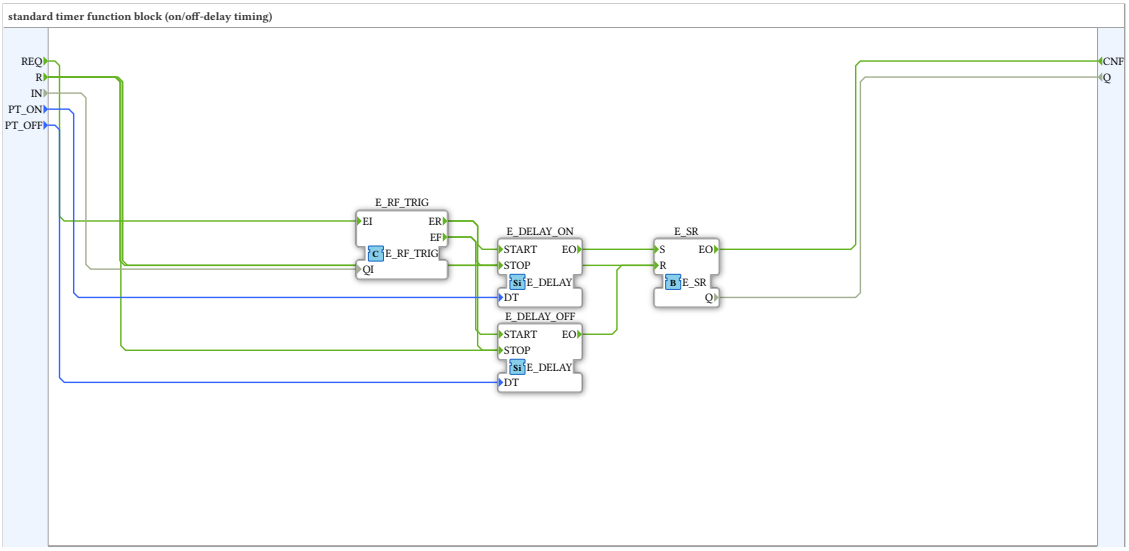
🛠️ Zugehörige Übungen

- Uebung_020c
- Uebung_020c2
- Uebung_039b

- Uebung_160b2

E_TONOF





Wichtiger Hinweis: Dieser Baustein benötigt nur ein Ereignis (Event) und keine zyklischen Aufrufe. Er besitzt keinen Ausgang ET und zeigt die verstrichene Zeit nicht an.

Beschreibung

Der Baustein **E_TONOF** ist ein standardisiertes Funktionsbaustein-Typ (FBType) gemäß der Norm **IEC 61499**, der für zeitverzögerte Schaltvorgänge in industriellen Automatisierungssystemen verwendet wird. Dieser Baustein kombiniert die Funktionen eines **On-Delay-Timers** (Einschaltverzögerung) und eines **Off-Delay-Timers** (Ausschaltverzögerung) in einem einzigen Baustein. Er ermöglicht es, sowohl das Einschalten als auch das Ausschalten eines Signals zeitlich zu verzögern.

Allgemeine Beschreibung

- **Name**: E_TONOF
- **Typ**: Standard Timer Function Block (On/Off-Delay Timing)
- **Zweck**: Der Baustein wird verwendet, um zeitverzögerte Schaltvorgänge in Steuerungssystemen zu realisieren. Er ist besonders nützlich in Anwendungen, bei denen eine Verzögerung beim Ein- oder Ausschalten von Signalen erforderlich ist.
- **Lizenz**: Der Baustein ist unter der Eclipse Public License 2.0 (EPL-2.0) lizenziert.

Interface-Beschreibung

Eingangsereignisse (Event Inputs):

1. **REQ** (Normal Execution Request):
   - **Typ**: Event
   - **Beschreibung**: Dieses Ereignis triggert die normale Ausführung des Bausteins. Es wird verwendet, um den Timer zu starten oder zu aktualisieren.

- **Zugehörige Variablen**:
  - **IN**: Der Eingangswert, der den Timer startet.
  - **PT_ON**: Die vorgegebene Zeitdauer für die Einschaltverzögerung (On-Delay).
  - **PT_OFF**: Die vorgegebene Zeitdauer für die Ausschaltverzögerung (Off-Delay).

2. **R** (Reset):

- **Typ**: Event
- **Beschreibung**: Dieses Ereignis setzt den Timer zurück. Wenn dieses Ereignis ausgelöst wird, wird der Timer gestoppt, und alle internen Zustände werden zurückgesetzt.

Ausgangsereignisse (Event Outputs):

1. **CNF** (Execution Confirmation):
   - **Typ**: Event
   - **Beschreibung**: Dieses Ereignis bestätigt die Ausführung des Bausteins. Es wird ausgelöst, wenn der Timer abgelaufen ist oder ein Zustandswechsel stattgefunden hat.
   - **Zugehörige Variable**:
     - **Q**: Der Ausgangswert, der den Zustand des Timers anzeigt.

Eingangsvariablen (Input Variables):

1. **IN**:

   - **Typ**: BOOL
   - **Beschreibung**: Der Eingangswert, der den Timer startet. Wenn **IN** auf `TRUE` gesetzt wird, startet die Einschaltverzögerung (On-Delay). Wenn **IN** auf `FALSE` gesetzt wird, startet die Ausschaltverzögerung (Off-Delay).

2. **PT_ON**:

   - **Typ**: TIME
   - **Beschreibung**: Die vorgegebene Zeitdauer für die Einschaltverzögerung (On-Delay). Diese Zeit wird ab dem Moment gemessen, in dem **IN** auf `TRUE` gesetzt wird.

3. **PT_OFF**:

   - **Typ**: TIME
   - **Beschreibung**: Die vorgegebene Zeitdauer für die Ausschaltverzögerung (Off-Delay). Diese Zeit wird ab dem Moment gemessen, in dem **IN** auf `FALSE` gesetzt wird.

Ausgangsvariablen (Output Variables):

1. **Q**:
   - **Typ**: BOOL
   - **Beschreibung**: Der Ausgangswert, der den Zustand des Timers anzeigt. **Q** wird auf `TRUE` gesetzt, nachdem die Einschaltverzögerung (On-Delay) abgelaufen ist, und bleibt so lange auf `TRUE`, bis die Ausschaltverzögerung (Off-Delay) abgelaufen ist.

Funktionsweise

1. **Einschaltverzögerung (On-Delay)**:

   - Wenn **IN** auf `TRUE` gesetzt wird, startet der Timer für die Einschaltverzögerung (**PT_ON**).
   - Nach Ablauf von **PT_ON** wird **Q** auf `TRUE` gesetzt.

2. **Ausschaltverzögerung (Off-Delay)**:

   - Wenn **IN** auf `FALSE` gesetzt wird, startet der Timer für die Ausschaltverzögerung (**PT_OFF**).
   - **Q** bleibt so lange auf `TRUE`, bis **PT_OFF** abgelaufen ist, und wird dann auf `FALSE` zurückgesetzt.

3. **Reset**:

   - Wenn das **R**-Ereignis ausgelöst wird, wird der Timer zurückgesetzt, und **Q** wird sofort auf `FALSE` gesetzt.

Anwendungsbeispiele

- **Maschinensteuerung**: Verzögerte Reaktion auf Sensoren oder Schaltbefehle, z. B. um ein Überlasten von Motoren zu vermeiden.
- **Prozessautomatisierung**: Zeitgesteuerte Schaltvorgänge in industriellen Prozessen, z. B. in Förderbändern oder Heizungssteuerungen.

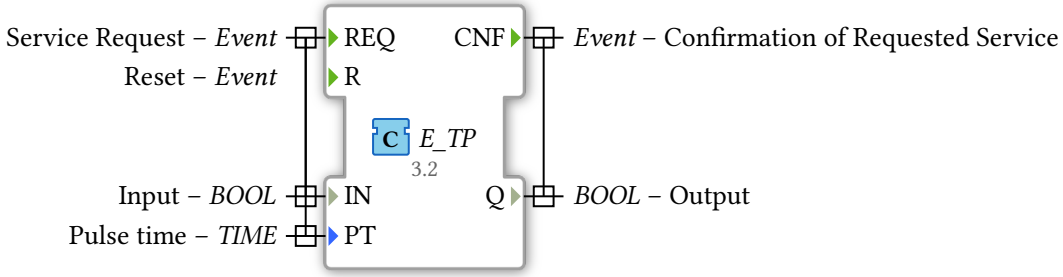
Zusammenfassung

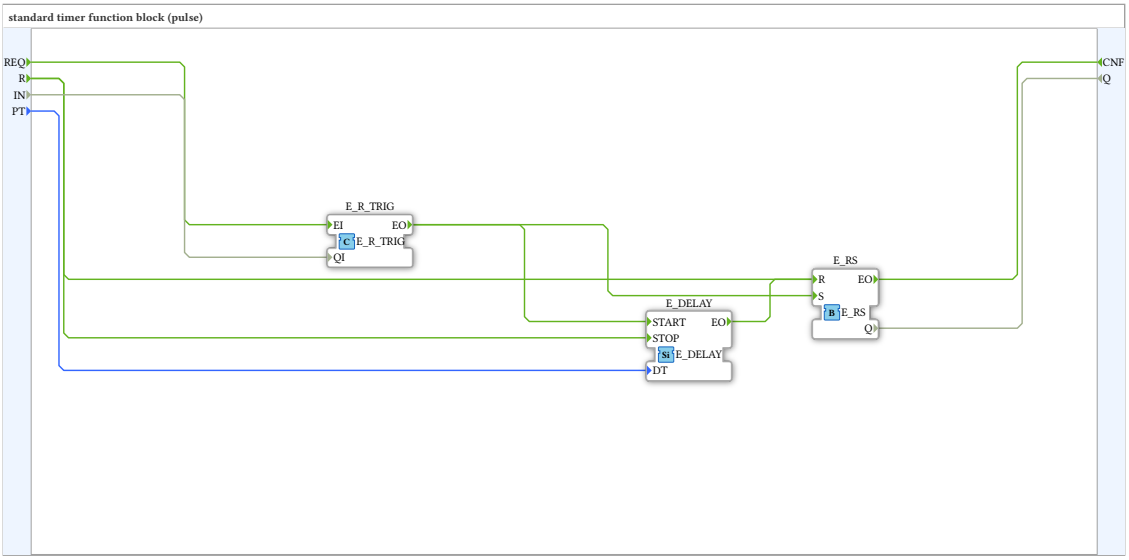
Der **E_TONOF**-Baustein ist ein leistungsstarkes Werkzeug in der IEC 61499, um zeitverzögerte Schaltvorgänge in industriellen Automatisierungssystemen zu realisieren. Er kombiniert die Funktionen eines Ein- und Ausschaltverzögerungstimers in einem einzigen Baustein und bietet zusätzlich die Möglichkeit, den Timer über ein Reset-Ereignis zurückzusetzen. Dies macht ihn besonders flexibel und vielseitig einsetzbar in komplexen Steuerungsaufgaben.

🛠️ Zugehörige Übungen

- Uebung_020g

E_TP





Wichtiger Hinweis: Dieser Baustein benötigt nur ein Ereignis (Event) und keine zyklischen Aufrufe. Er besitzt keinen Ausgang ET und zeigt die verstrichene Zeit nicht an.

Beschreibung

Der Baustein **E_TP** (auch bekannt als **Pulsgeber** oder **Timer-Puls**) ist ein standardisiertes Funktionsbaustein (FB) nach der Norm **IEC 61499** (DIN EN 61499). Dieser Baustein wird verwendet, um einen pulsierenden Ausgang zu erzeugen, der für eine bestimmte Zeit aktiviert wird, wenn ein Eingangssignal empfangen wird. Er ist besonders nützlich in Steuerungsanwendungen, bei denen zeitgesteuerte Aktionen erforderlich sind.

Beschreibung des Bausteins E_TP

Allgemeine Funktion:

- Der Baustein **E_TP** generiert einen pulsierenden Ausgang (**Q**), der für eine vorgegebene Zeit (**PT**) aktiviert wird, sobald ein Eingangssignal (**IN**) empfangen wird.
- Wenn das Eingangssignal **IN** den Wert **TRUE** annimmt, wird der Ausgang **Q** für die Zeitdauer **PT** auf **TRUE** gesetzt. Nach Ablauf der Zeit **PT** wird **Q** wieder auf **FALSE** zurückgesetzt, unabhängig vom Zustand von **IN**.
- Der Baustein verfügt über eine Reset-Funktion (**R**), die den Timer zurücksetzt und den Ausgang **Q** sofort auf **FALSE** setzt, unabhängig davon, ob die Zeit **PT** abgelaufen ist oder nicht.

Schnittstellen (Interface):

1. **Eingangsereignisse (Event Inputs):**
   - **REQ (Service Request):** Dieses Ereignis wird ausgelöst, wenn der Timer gestartet werden soll. Es ist mit den Eingangsvariablen **IN** und **PT** verknüpft.
   - **R (Reset):** Dieses Ereignis wird ausgelöst, um den Timer zurückzusetzen. Es ist mit der Eingangsvariablen **IN** verknüpft.
2. **Ausgangsereignisse (Event Outputs):**
   - **CNF (Confirmation of Requested Service):** Dieses Ereignis wird ausgelöst, wenn der Timer abgelaufen ist und der Ausgang **Q** zurückgesetzt wird. Es ist mit der Ausgangsvariablen **Q** verknüpft.
3. **Eingangsvariablen (Input Variables):**
   - **IN (BOOL):** Der Eingang, der den Timer startet. Wenn **IN** den Wert **TRUE** annimmt, wird der Timer gestartet.
   - **PT (TIME):** Die vorgegebene Zeitdauer, für die der Ausgang **Q** aktiviert wird.
4. **Ausgangsvariablen (Output Variables):**
   - **Q (BOOL):** Der Ausgang, der für die Zeitdauer **PT** auf **TRUE** gesetzt wird, nachdem **IN** aktiviert wurde.

Verhalten:

- Wenn **IN** auf **TRUE** gesetzt wird, wird der Ausgang **Q** sofort auf **TRUE** gesetzt und bleibt für die Zeitdauer **PT** aktiv.
- Nach Ablauf der Zeit **PT** wird **Q** automatisch auf **FALSE** zurückgesetzt.
- Wenn das Reset-Ereignis **R** ausgelöst wird, wird der Timer sofort zurückgesetzt, und **Q** wird auf **FALSE** gesetzt, unabhängig davon, ob die Zeit **PT** abgelaufen ist oder nicht.

Anwendungsbeispiele:

- **Steuerung von Maschinen:** Der Baustein kann verwendet werden, um eine Maschine für eine bestimmte Zeit zu aktivieren, z. B. das Einschalten eines Motors für eine festgelegte Dauer.
- **Blinklichtsteuerung:** Der Baustein kann verwendet werden, um ein Blinklicht für eine bestimmte Zeit zu aktivieren.
- **Verzögerte Aktionen:** In Szenarien, in denen eine Aktion nach einer bestimmten Verzögerung ausgeführt werden soll, kann der Baustein verwendet werden, um die Verzögerung zu realisieren.

Versionierung:

- **Version 1.0:** Ursprüngliche Version des Bausteins, erstellt von Franz Hoepfinger am 04.03.2024.
- **Version 1.1:** Eine Reset-Funktion wurde hinzugefügt, um den Timer manuell zurücksetzen zu können. Diese Version wurde am 23.04.2024 veröffentlicht.

Zusammenfassung

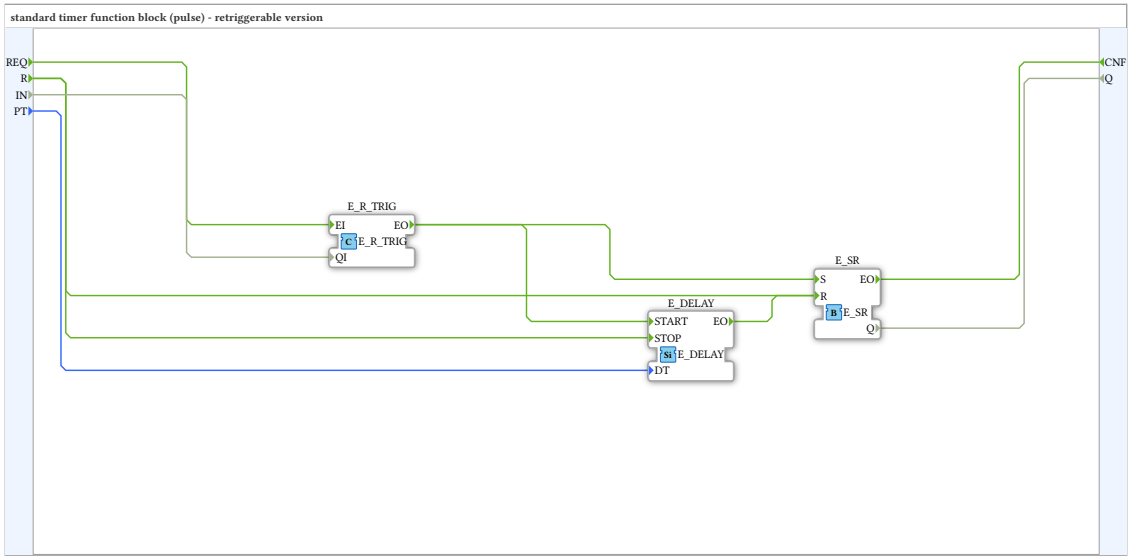
Der **E_TP**-Baustein ist ein nützliches Werkzeug in der IEC 61499-basierten Steuerungstechnik, um zeitgesteuerte Aktionen zu realisieren. Durch seine einfache Schnittstelle und das klare Verhalten ist er leicht in verschiedene Steuerungsanwendungen integrierbar. Die Reset-Funktion in Version 1.1 erhöht die Flexibilität des Bausteins, da der Timer nun manuell zurückgesetzt werden kann.

🛠️ Zugehörige Übungen

- Uebung_020f

- Uebung_039b
- Uebung_177
- Uebung_178

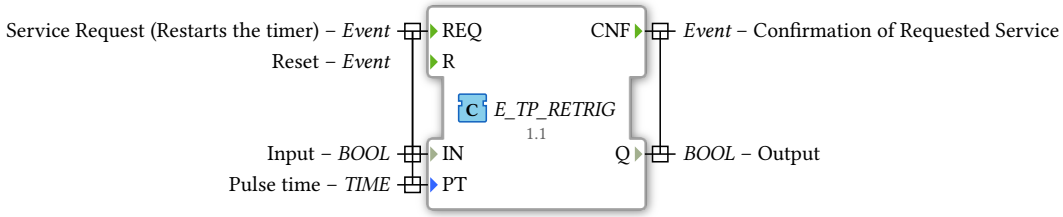
E_TP_RETRIG



• – ○ • – ○ • – ○ •

Einleitung

Der `E_TP_RETRIG` ist die retriggerbare Variante des Standard-Timers `E_TP`: Er erzeugt bei jeder steigenden Flanke von `IN` einen `TRUE`-Impuls der Dauer `PT` an `Q` — anders als `E_TP` startet eine erneute steigende Flanke während eines laufenden Impulses diesen jedoch neu, statt ignoriert zu werden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Prüft `IN` auf eine steigende Flanke, trägt `IN` und `PT`.
- **R**: Bricht den laufenden Impuls sofort ab.

Ereignis-Ausgänge

- **CNF**: Bestätigung, trägt `Q`.

Daten-Eingänge

- **IN** (BOOL): Der auf steigende Flanken zu prüfende Eingang.
- **PT** (TIME): Dauer des Impulses.

Daten-Ausgänge

- **Q** (BOOL): `TRUE` während der Impulsdauer.

Funktionsweise

`E_R_TRIG` erkennt steigende Flanken von `IN` bei jedem `REQ`. Jede erkannte Flanke setzt `E_SR.S` (⟶ `Q := TRUE`) und startet gleichzeitig `E_DELAY` (mit `PT`) neu — eine erneute steigende Flanke während eines

laufenden Impulses verlängert diesen also, statt ignoriert zu werden. Läuft `E_DELAY` ab, setzt es `E_SR.R` (⟶ `Q := FALSE`). Ein R-Ereignis stoppt den Timer sofort und setzt `Q` direkt zurück.

Technische Besonderheiten

- **Retriggerbar über `IN`-Flanke statt `REQ` allein**: Im Unterschied zu E_PULSE_RETRIG, das bei *jedem* `REQ`-Ereignis retriggert, reagiert `E_TP_RETRIG` nur auf tatsächliche steigende Flanken von `IN` — wiederholte `REQ`-Ereignisse bei bereits `TRUE` gehaltenem `IN` ohne neue Flanke lösen keinen neuen Impuls aus.
- **Zusammengesetzt aus drei Standardbausteinen**: `E_R_TRIG` + `E_DELAY` + `E_SR`, keine eigene ECC.

Zustandsübersicht

Zustandslose Verdrahtungslogik: `E_SR` hält den `Q`-Zustand, `E_DELAY` läuft bei jeder erkannten steigenden Flanke von `IN` neu an.

Anwendungsszenarien

- **Verlängerbare Impulse ausgelöst durch Flankenwechsel**: z. B. eine Bewegungsmelder-gesteuerte Beleuchtung, bei der jede neue Bewegung (steigende Flanke von `IN`) die Leuchtdauer neu beginnen lässt.
- **Entprellte, aber verlängerbare Reaktion auf einen digitalen Eingang**, wenn nur echte Flankenwechsel und kein bloßes Wiederholen von `REQ` einen neuen Impulslauf auslösen soll.

⚖ Vergleich mit ähnlichen Bausteinen

- **`E_TP`**: dieselbe Grundfunktion, aber nicht retriggerbar.
- E_PULSE_RETRIG: retriggerbarer Impuls, ausgelöst über *jedes* `REQ`-Ereignis statt über eine `IN`-Flanke.
- E_TLIM: begrenzt die Ein-Dauer von `IN` selbst, statt einen festen Impuls zu erzeugen.

Fazit

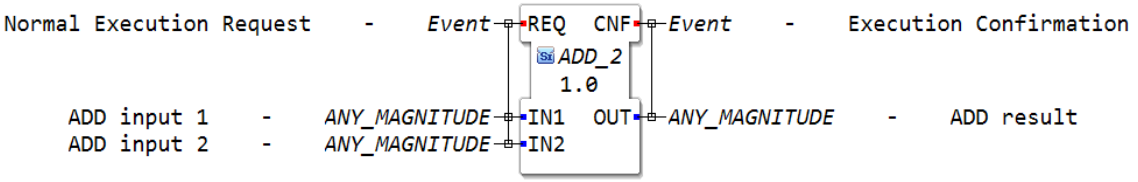
`E_TP_RETRIG` liefert einen flankengetriggerten, verlängerbaren Impuls fester Dauer und eignet sich für Anwendungen, bei denen wiederholte echte Flankenwechsel (nicht bloße `REQ`-Wiederholungen) den Impuls am Leben halten sollen.

2.4.1 iec61131-3

2.4.1.1 Arithmetische Funktionen

Hier finden Sie Standardfunktionen für arithmetische Operationen (Addition, Subtraktion, Multiplikation, Division etc.).

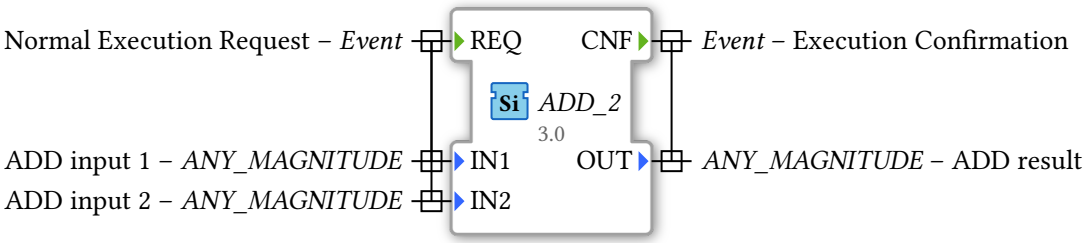
ADD_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **ADD_2** ist ein generischer Funktionsbaustein zur Addition zweier Werte, entwickelt unter EPL-2.0 Lizenz. Die Version 1.0 unterstützt verschiedene numerische Datentypen gemäß IEC 61131-3 Standard.



Schnittstellenstruktur

Ereignis-Eingänge
- `REQ`: Berechnungsanforderung (mit Eingabewerten)

Ereignis-Ausgänge
- `CNF`: Berechnungsbestätigung (mit Ergebnis)

Daten-Eingänge
- `IN1` (ANY_MAGNITUDE): Erster Summand
- `IN2` (ANY_MAGNITUDE): Zweiter Summand

Daten-Ausgänge
- `OUT` (ANY_MAGNITUDE): Summen-Ergebnis

Funktionsweise
1. **Berechnungsanforderung**:
   - `REQ`-Ereignis mit Werten für IN1 und IN2
2. **Addition**:
   - Typenerhaltende Berechnung (IN1 + IN2)
   - Unterstützte Typen: INT, DINT, REAL, LREAL etc.
3. **Ergebnisausgabe**:
   - `CNF`-Ereignis mit berechnetem OUT-Wert
   - Automatische Typanpassung

Technische Besonderheiten
✓ **Generische Implementierung** (GEN_ADD) ✓ **Typflexible** Eingänge/Ausgänge ✓ **IEC 61131-3 konform** ✓ **Deterministische** Berechnung

Unterstützte Datentypen

Typ	Beispiel
INT	123 + 456 = 579
REAL	1.23 + 4.56 = 5.79
TIME	T#1s + T#2s = T#3s
LREAL	1.23e10 + 4.56e10

Anwendungsszenarien
- **Prozessregelung**: Signalverarbeitung
- **Datenanalyse**: Messwertaggregation
- **Maschinensteuerung**: Positionsberechnung
- **Energiemanagement**: Verbrauchssummierung

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	ADD_2	ADD_4	MUL_2
Operation	Addition	Addition	Multiplikation
Eingänge	2	4	2
Typunterstützung	ANY_MAG	ANY_MAG	ANY_MAG

Fehlerbehandlung
- Typinkompatibilität führt zu:

- OUT = 0 (für numerische Typen)
- Leerer Wert (für andere Typen)
- Keine Exception-Generierung

🛠️ Zugehörige Übungen

- Uebung_000
- Uebung_000_AX
- Uebung_110
- Uebung_111

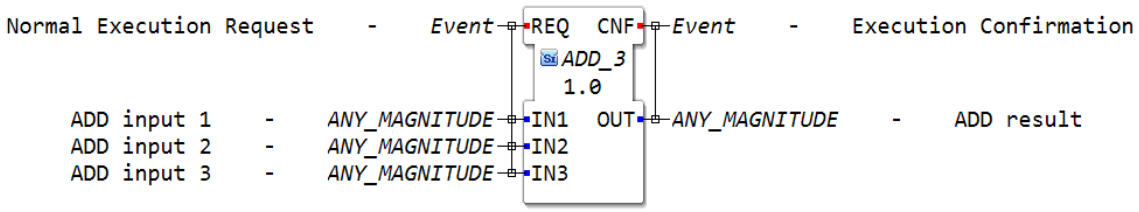
Fazit

Der ADD_2-Baustein bietet eine flexible Lösung für arithmetische Operationen:

- Einfache aber mächtige Addition
- Breite Typunterstützung
- Standardkonforme Implementierung

Besonders wertvoll für Anwendungen, die typsichere Berechnungen mit variablen Eingangsgrößen erfordern. Die generische Implementierung ermöglicht Wiederverwendung in unterschiedlichsten Steuerungskontexten.

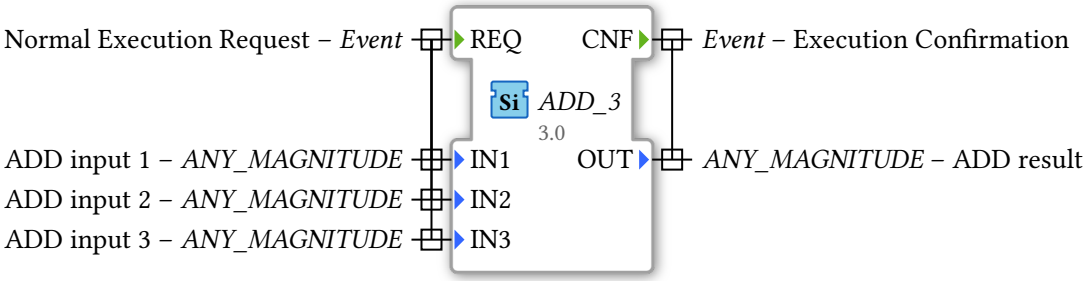
ADD_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **ADD_3** ist eine erweiterte Version des Addierers zur Berechnung der Summe dreier Werte, entwickelt unter EPL-2.0 Lizenz. Version 1.0 bietet typsichere Addition für verschiedene numerische Datentypen gemäß IEC 61131-3 Standard.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Berechnungsanforderung (mit drei Eingabewerten)

Ereignis-Ausgänge

- `CNF`: Berechnungsbestätigung (mit Summenergebnis)

Daten-Eingänge

- `IN1` (ANY_MAGNITUDE): Erster Summand
- `IN2` (ANY_MAGNITUDE): Zweiter Summand
- `IN3` (ANY_MAGNITUDE): Dritter Summand

Daten-Ausgänge

- `OUT` (ANY_MAGNITUDE): Ergebnis der Addition (IN1 + IN2 + IN3)

Funktionsprinzip

1. **Anforderung**:
   - `REQ`-Ereignis triggert die Berechnung
   - Alle drei Eingangswerte werden gleichzeitig ausgewertet
2. **Berechnung**:
   - Typenerhaltende Summenbildung
   - Unterstützt alle ANY_MAGNITUDE-Typen:
     - Ganzzahlen (INT, DINT)
     - Fließkommazahlen (REAL, LREAL)
     - Zeitwerte (TIME, LTIME)
3. **Ergebnisausgabe**:
   - `CNF`-Ereignis mit berechnetem OUT-Wert
   - Automatische Typkonvertierung bei gemischten Eingängen

Technische Besonderheiten

✓ **Drei-Eingang-Addition** in einer Operation ✓ **Generische Implementierung** (GEN_ADD-Klasse) ✓ **Breite Typunterstützung** (ANY_MAGNITUDE) ✓ **Deterministische Ausführung**

Typische Anwendungsfälle

- **Messwertaggregation**: Summierung mehrerer Sensoren
- **Energiebilanzierung**: Gesamtverbrauchsberechnung
- **Positionstechnik**: Mehrdimensionale Koordinaten
- **Produktionsstatistik**: Mengenerfassung

⚖️ Vergleich mit ADD_2

Feature	ADD_3	ADD_2
Eingänge	3	2
Berechnung	IN1+IN2+IN3	IN1+IN2
Performance	Höhere Effizienz bei 3 Werten	Leichtgewichtiger

Fehlerbehandlung

- Bei Typinkompatibilität:
  - OUT = 0 (numerische Typen)
  - Leerer Wert (nicht-numerische Typen)
  - Kein Abbruch der Ausführung

🛠️ Zugehörige Übungen

- Uebung_087
- Uebung_087a1

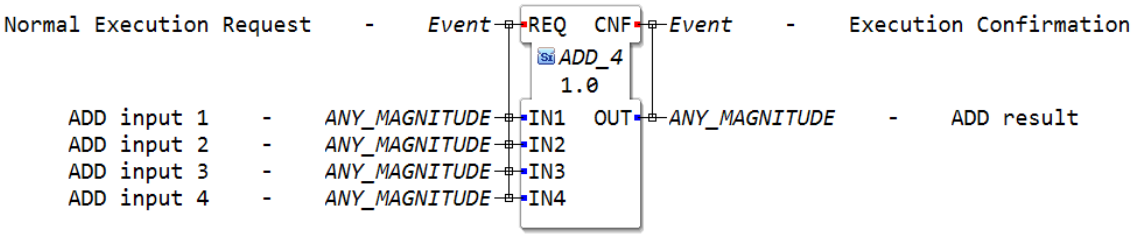
Fazit

Der ADD_3-Baustein erweitert die Möglichkeiten der typsicheren Addition:

- Effiziente Dreifach-Summation
- Konsistente Typbehandlung
- Optimierte Performance gegenüber serieller ADD_2-Nutzung

Besonders vorteilhaft für Anwendungen mit mehreren zu aggregierenden Werten. Die generische Implementierung ermöglicht maximale Flexibilität in heterogenen Steuerungsumgebungen.

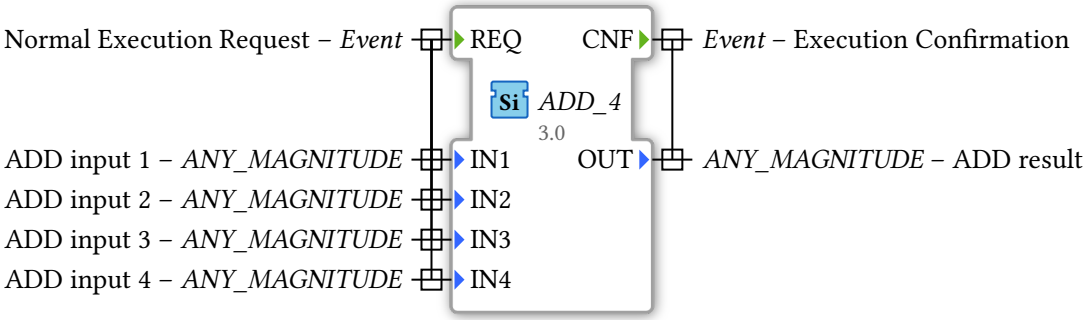
ADD_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **ADD_4** ist ein leistungsfähiger Funktionsbaustein zur Summation von vier Werten, entwickelt unter EPL-2.0 Lizenz. Als Teil der GEN_ADD-Klasse bietet Version 1.0 eine typsichere Implementierung gemäß IEC 61131-3 Standard.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Berechnungsanforderung (mit vier Eingabewerten)

Ereignis-Ausgänge

- `CNF`: Berechnungsbestätigung (mit Summenergebnis)

Daten-Eingänge

- `IN1` (ANY_MAGNITUDE): Erster Summand
- `IN2` (ANY_MAGNITUDE): Zweiter Summand
- `IN3` (ANY_MAGNITUDE): Dritter Summand
- `IN4` (ANY_MAGNITUDE): Vierter Summand

Daten-Ausgänge

- `OUT` (ANY_MAGNITUDE): Ergebnis der Addition (IN1+IN2+IN3+IN4)

Funktionale Merkmale

1. **Berechnungsauslösung**:
   - Einzelnes `REQ`-Ereignis verarbeitet alle vier Eingänge
   - Parallele Werterfassung
2. **Typhandling**:
   - Automatische Typkonvertierung bei gemischten Eingängen
   - Ergebnis im „größten“ Eingangstyp
3. **Berechnungsreihenfolge**:
   - Mathematisch assoziativ (IN1+(IN2+(IN3+IN4)))
   - Garantierte reproduzierbare Ergebnisse

Technische Spezifikationen

✓ **Vier-Eingang-Addition** in einer Operation ✓ **Volle ANY_MAGNITUDE-Unterstützung** ✓ **Optimierte Berechnungsperformance** ✓ **Deterministisches Verhalten**

Unterstützte Datentypen

Kategorie	Beispiele
Ganzzahlen	INT, DINT, UDINT
Fließkomma	REAL, LREAL
Zeitwerte	TIME, LTIME
Spezialtypen	DATE_AND_TIME

Anwendungsszenarien

- **Energiemonitoring**: Summierung von Verbrauchern
- **Produktionskontrolle**: Mengenerfassung mehrerer Linien
- **Messdatenanalyse**: Aggregation von Sensorwerten
- **Robotersteuerung**: Mehrdimensionale Positionsberechnung

⚖️ **Vergleich mit anderen Addierern**

Baustein	Eingänge	Vorteile
ADD_2	2	Einfachste Variante
ADD_3	3	Kompromiss
ADD_4	4	Höchste Integration

Fehlermanagement

- **Typkonflikte**: Ergebnis 0 bei Inkompatibilität
- **Überlauf**: Wrap-around je nach Datentyp
- **Statusrückmeldung**: Über begleitende Logik

Fazit

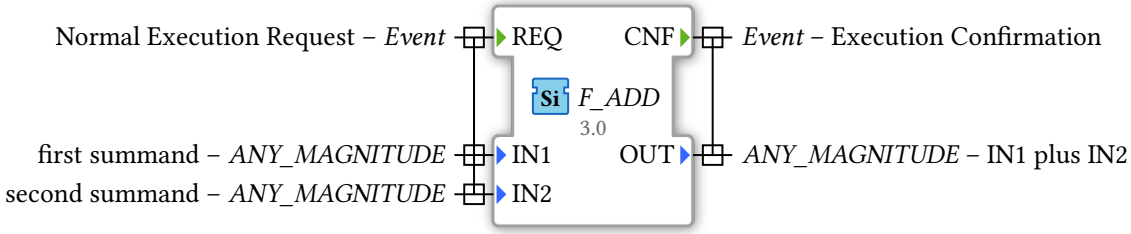
Der ADD_4-Baustein bietet maximale Effizienz für Summationsaufgaben:

- Kompakte Vierfach-Berechnung
- Breiteste Typunterstützung
- Optimierte Performance

Ideal für komplexe Steuerungsanwendungen mit multiplen zu aggregierenden Werten. Die generische Implementierung als GEN_ADD ermöglicht höchste Flexibilität bei typsicherer Ausführung.

F_ADD

• – ◦ • – ◦ • – ◦ •



Einführung

Der **F_ADD** ist ein standardkonformer arithmetischer Funktionsbaustein nach IEC 61131-3, entwickelt von der TU Wien ACIN unter EPL-2.0 Lizenz. Version 1.0 ermöglicht eine typsichere Addition von Werten (ANY_MAGNITUDE) für industrielle Steuerungsanwendungen.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Berechnungsanforderung (mit Eingabewerten)

Ereignis-Ausgänge

- `CNF`: Berechnungsbestätigung (mit Ergebnis)

Daten-Eingänge

- `IN1` (ANY_MAGNITUDE): Erster Operand
- `IN2` (ANY_MAGNITUDE): Zweiter Operand

Daten-Ausgänge

- `OUT` (ANY_MAGNITUDE): Summe von IN1 und IN2

Funktionsweise

1. **Aktivierung**:
   - Durch `REQ`-Ereignis ausgelöst
   - Verarbeitet beide Eingabewerte gleichzeitig
2. **Typenbehandlung**:
   - Unterstützt alle ANY_MAGNITUDE-Typen:
     - Numerische Typen (INT, DINT, REAL, LREAL)
     - Zeittypen (TIME, LTIME)
   - Automatische Typkonvertierung
3. **Berechnung**:
   - Führt IN1 + IN2 aus
   - Behält höchste Eingabegenauigkeit bei
4. **Ergebnisausgabe**:
   - `CNF`-Ereignis mit Berechnungsergebnis
   - Latenz: Ein Ausführungszyklus

Technische Spezifikationen

✓ Generische Typunterstützung (ANY_MAGNITUDE) **✓ Deterministische Ausführung ✓ Überlaufschutz ✓ IEC 61131-3 konform**

Unterstützte Datentypen

Kategorie	Beispiele
Ganzzahlen	INT, DINT, UINT, UDINT
Fließkomma	REAL, LREAL
Zeit	TIME, LTIME
Gemischt	INT + REAL → REAL

Anwendungsbeispiele

- **Prozessregelung**: Signalaufbereitung
- **Messsysteme**: Sensorwertaggregation
- **Bewegungssteuerung**: Positionsberechnungen
- **Energiemanagement**: Verbrauchssummierung

⚖️ Vergleich mit ähnlichen Bausteinen

Merkmal	F_ADD	F_SUB	F_MUL
Operation	Addition	Subtraktion	Multiplikation
Eingangstypen	ANY_MAGNITUDE	ANY_MAGNITUDE	ANY_NUM
Ausgangstyp	Entspricht Eingangstyp	Gleich	Gleich

Fehlerbehandlung

- **Überlauf**: Wrap-around bei Ganzzahlen, ±INF bei Fließkomma
- **Typenkonflikt**: Automatische Typanpassung
- **Ungültige Eingaben**: Gibt 0/Standardwert zurück

Besondere Hinweise

- Für Zeitberechnungen: TIME + TIME ⟶ TIME
- Gemischte Operationen verwenden „höheren“ Typ
- Keine implizite Skalierung (z.B. 100ms + 1s = 1100ms)

Fazit

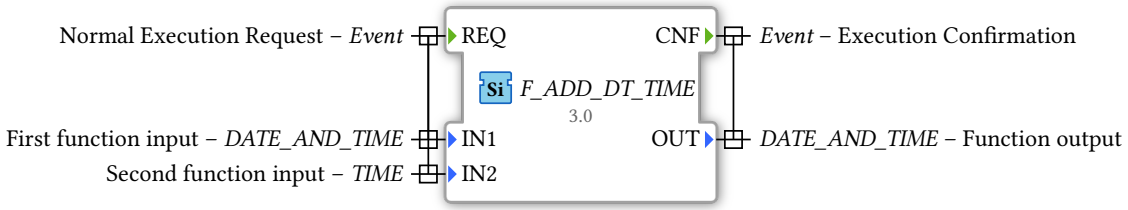
Der F_ADD-Baustein bietet wesentliche Rechenfunktionen:

- Robuste typsichere Addition
- Konsistentes Verhalten über alle Datentypen
- Nahtlose Integration in Steuerungsalgorithmen

Die standardisierte Implementierung macht ihn zu einer zuverlässigen Komponente für Industrieautomatisierungssysteme, die präzise Rechenoperationen mit verschiedenen Datentypen erfordern.

F_ADD_DT_TIME

• − ◦ • − ◦ • − ◦ •



Einführung

Der **F_ADD_DT_TIME** ist ein spezialisierter Funktionsbaustein zur Addition von Zeitwerten zu Datums-/Zeitangaben, entwickelt unter EPL-2.0 Lizenz. Version 1.0 ermöglicht präzise Zeitberechnungen mit DATE_AND_TIME und TIME-Datentypen.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Berechnungsanforderung (mit Eingabewerten)

Ereignis-Ausgänge

- `CNF`: Berechnungsbestätigung (mit Ergebnis)

Daten-Eingänge

- `IN1` (DATE_AND_TIME): Ausgangsdatum/-zeit
- `IN2` (TIME): Zeitdauer für Addition

Daten-Ausgänge

- `OUT` (DATE_AND_TIME): Ergebnis der Addition

Funktionsweise

1. **Aktivierung**:
   - Durch `REQ`-Ereignis ausgelöst
   - Verarbeitet DATE_AND_TIME und TIME-Werte
2. **Berechnung**:
   - Addiert Zeitdauer (IN2) zum Zeitstempel (IN1)
   - Berücksichtigt automatisch:
     - Schaltjahre
     - Monatslängen
     - Zeitzonen (falls konfiguriert)
3. **Ergebnisausgabe**:
   - `CNF`-Ereignis mit neuem DATE_AND_TIME
   - Format: DT#2023-12-31-23:59:59

Technische Spezifikationen

✓ Präzise Zeit-/Datumsarithmetik ✓ Automatische Kalenderberechnung ✓ IEC 61131-3 konform ✓ Deterministische Ausführung

Anwendungsbeispiele

- **Prozesssteuerung**: Fälligkeitsberechnungen
- **Produktionsplanung**: Terminverschiebungen
- **Wartungssysteme**: Wartungsintervallberechnung
- **Datenprotokollierung**: Zeitstempelkorrekturen

Besondere Hinweise

- **Gültigkeitsbereich**:
  - Unterstützt Datumsbereich 1970-2099
  - Zeitauflösung: 1 Millisekunde
- **Fehlerfälle**:
  - Bei Überlauf: Wrap-around im gültigen Bereich
  - Bei ungültigen Eingaben: OUT = IN1

⚖️ **Vergleich mit Standard-F_ADD**

Merkmal	F_ADD_DT_TIME	F_ADD
Eingangstyp 1	DATE_AND_TIME	ANY_MAGNITUDE
Eingangstyp 2	TIME	ANY_MAGNITUDE
Operation	Spezialisierte Zeitaddition	Generische Addition

Fazit

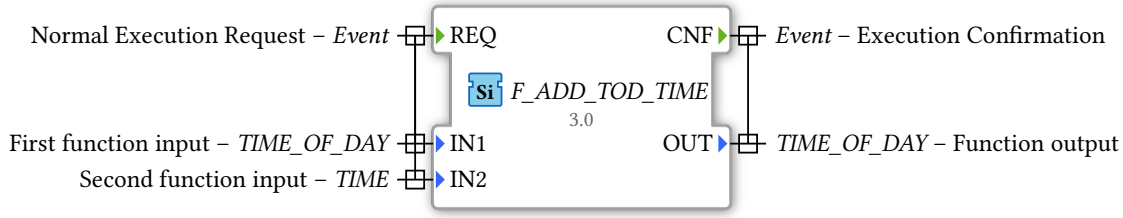
Der F_ADD_DT_TIME-Baustein bietet essentielle Funktionen für Zeitberechnungen:

- Präzise Handhabung von Datum/Zeit-Operationen
- Automatische Kalenderkorrekturen
- Zuverlässige Integration in Steuerungssysteme

Besonders wertvoll für Anwendungen, die komplexe Zeitberechnungen mit automatischer Berücksichtigung von Kalenderregeln erfordern. Die spezialisierte Implementierung garantiert korrekte Ergebnisse für alle gültigen Eingabewerte.

F_ADD_TOD_TIME

• – ◦ • – ◦ • – ◦ •



Einführung

Der **F_ADD_TOD_TIME** ist ein spezialisierter Funktionsbaustein zur Addition von Zeitwerten zu Tageszeiten, entwickelt unter EPL-2.0 Lizenz. Version 1.0 ermöglicht präzise Zeitberechnungen mit TIME_OF_DAY und TIME-Datentypen.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Berechnungsanforderung (mit Eingabewerten)

Ereignis-Ausgänge

- `CNF`: Berechnungsbestätigung (mit Ergebnis)

Daten-Eingänge

- `IN1` (TIME_OF_DAY): Ausgangszeit (Tageszeit)
- `IN2` (TIME): Zeitdauer für Addition

Daten-Ausgänge

- `OUT` (TIME_OF_DAY): Ergebnis der Addition

Funktionsweise

1. **Aktivierung**:
   - Durch `REQ`-Ereignis ausgelöst
   - Verarbeitet TIME_OF_DAY und TIME-Werte
2. **Berechnung**:
   - Addiert Zeitdauer (IN2) zur Tageszeit (IN1)
   - Berücksichtigt automatisch:
     - Tagesübergänge (24h-Format)
     - Millisekundengenauigkeit
3. **Ergebnisausgabe**:
   - `CNF`-Ereignis mit neuer TIME_OF_DAY
   - Format: TOD#23:59:59.999

Technische Spezifikationen

✓ Präzise Tageszeitberechnung ✓ Automatischer Tagesübergang ✓ IEC 61131-3 konform ✓ Millisekundengenaue Berechnung

Anwendungsbeispiele

- **Schichtplanung**: Arbeitszeitberechnungen
- **Lichtsteuerung**: Tageslichtsimulation
- **Produktionsprozesse**: Zeitgesteuerte Abläufe
- **Fahrplansysteme**: Ankunftszeitberechnungen

Besondere Hinweise

- **Gültigkeitsbereich**:
  - 00:00:00.000 bis 23:59:59.999
  - Bei Überschreitung: Wrap-around (z.B. 25:00:00 ⟶ 01:00:00)

- **Fehlerfälle**:
  - Bei Überlauf: Zyklische Weiterzählung
  - Bei ungültigen Eingaben: OUT = IN1

⚖️ Vergleich mit ähnlichen Bausteinen

Merkmal	F_ADD_TOD_TIME	F_ADD_DT_TIME
Eingangstyp 1	TIME_OF_DAY	DATE_AND_TIME
Ausgangstyp	TIME_OF_DAY	DATE_AND_TIME
Bereich	Tageszyklus	Absolutes Datum

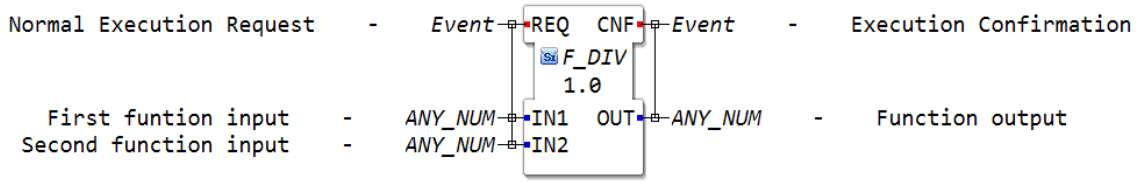
Fazit

Der F_ADD_TOD_TIME-Baustein bietet essentielle Funktionen für Tageszeitberechnungen:

- Präzise Handhabung von Tageszeit-Operationen
- Automatische Behandlung von Tagesübergängen
- Optimierte Performance für Echtzeitanwendungen

Ideal für Steuerungsanwendungen, die wiederkehrende tageszeitbasierte Operationen erfordern. Die spezialisierte Implementierung garantiert korrekte Ergebnisse im 24-Stunden-Format.

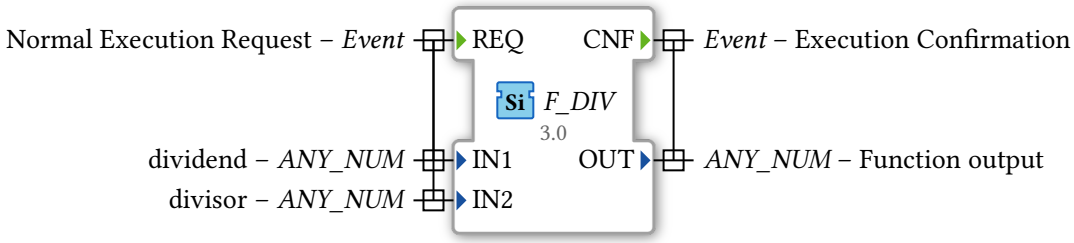
F_DIV



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **F_DIV** ist ein standardkonformer Funktionsbaustein zur Division numerischer Werte, entwickelt unter EPL-2.0 Lizenz. Version 1.0 ermöglicht die typsichere Division verschiedener numerischer Datentypen gemäß IEC 61131-3 Standard.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Berechnungsanforderung (mit Dividend und Divisor)

Ereignis-Ausgänge

- `CNF`: Berechnungsbestätigung (mit Divisionsergebnis)

Daten-Eingänge

- `IN1` (ANY_NUM): Dividend (Zähler)
- `IN2` (ANY_NUM): Divisor (Nenner)

Daten-Ausgänge

- `OUT` (ANY_NUM): Ergebnis der Division (IN1 / IN2)

Funktionsprinzip

1. **Berechnungsauslösung**:
   - `REQ`-Ereignis mit Werten für IN1 und IN2
   - Beide Eingänge müssen vom Typ ANY_NUM sein
2. **Division**:
   - Typenerhaltende Berechnung (IN1 / IN2)
   - Unterstützte Typen:
     - Ganzzahlen (INT, DINT)
     - Fließkommazahlen (REAL, LREAL)
     - Gemischte Typen (automatische Konvertierung)
3. **Ergebnisausgabe**:
   - `CNF`-Ereignis mit berechnetem OUT-Wert
   - Fließkommaergebnis bei Ganzzahldivision

Technische Besonderheiten

✓ Typsichere Division (ANY_NUM-Unterstützung) **✓ Automatische Typkonvertierung ✓ IEC 61131-3 konform ✓ Deterministische Ausführung**

Fehlerbehandlung

- **Division durch Null**:
  - OUT = 0 (für Ganzzahlen)
  - OUT = INF/NAN (für Fließkommazahlen)
  - Kein Abbruch der Ausführung
- **Typinkompatibilität**:
  - OUT = 0
  - Keine Exception-Generierung

Anwendungsszenarien

- **Prozessregelung**: Regelabweichungsberechnung
- **Datenanalyse**: Normalisierung von Messwerten
- **Maschinensteuerung**: Geschwindigkeitsberechnungen
- **Energiemanagement**: Wirkungsgradberechnungen

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	F_DIV	F_MUL	F_ADD
Operation	Division	Multiplikation	Addition
Eingänge	2	2	2
Typunterstützung	ANY_NUM	ANY_NUM	ANY_NUM

Unterstützte Datentypen

Kategorie	Beispiele
Ganzzahlen	INT, DINT, UDINT
Fließkomma	REAL, LREAL
Typkombination	INT/REAL, DINT/LREAL

🛠️ Zugehörige Übungen
- Uebung_111

Fazit

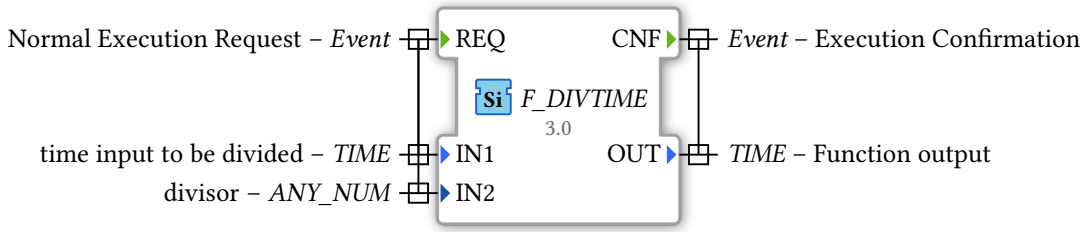
Der F_DIV-Baustein bietet eine robuste Lösung für Divisionsoperationen:

- Typsichere Berechnung
- Flexible Eingangskombinationen
- Standardkonforme Implementierung

Besonders wertvoll für Anwendungen, die präzise Berechnungen mit variablen Eingangsgrößen erfordern. Die automatische Typkonvertierung und Fehlerbehandlung machen ihn zu einem zuverlässigen Baustein in industriellen Steuerungssystemen.

F_DIVTIME

• – ◦ • – ◦ • – ◦ •



Einführung

Der **F_DIVTIME** ist ein spezialisierter Funktionsbaustein zur Division von Zeitwerten durch numerische Faktoren, entwickelt unter EPL-2.0 Lizenz. Version 1.0 ermöglicht präzise Zeitberechnungen mit TIME- und ANY_NUM-Datentypen.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Berechnungsanforderung (mit Eingabewerten)

Ereignis-Ausgänge

- `CNF`: Berechnungsbestätigung (mit Ergebnis)

Daten-Eingänge

- `IN1` (TIME): Zu dividierende Zeitdauer
- `IN2` (ANY_NUM): Divisor (numerischer Wert)

Daten-Ausgänge

- `OUT` (TIME): Ergebnis der Division (IN1 / IN2)

Funktionsweise

1. **Aktivierung**:
   - Durch `REQ`-Ereignis ausgelöst
   - Verarbeitet TIME-Werte und numerische Divisoren
2. **Berechnung**:
   - Dividiert Zeitwert (IN1) durch numerischen Wert (IN2)
   - Führt automatische Typkonvertierung durch
   - Millisekundengenaue Berechnung
3. **Ergebnisausgabe**:
   - `CNF`-Ereignis mit neuem TIME-Wert
   - Format: T#1d5h30m15s250ms

Technische Spezifikationen

✓ Präzise Zeitdivision ✓ Unterstützung für ANY_NUM-Divisoren ✓ Millisekundengenaue Berechnung ✓ Automatische Typkonvertierung

Anwendungsbeispiele

- **Prozessoptimierung**: Mittlere Zykluszeitberechnung
- **Produktionsplanung**: Zeitaufteilung auf Stationen
- **Robotersteuerung**: Geschwindigkeitsanpassung
- **Energiemanagement**: Verbrauchszeitberechnungen

Fehlerbehandlung

- **Division durch Null**:
  - OUT = T#0s
  - Kein Abbruch der Ausführung
- **Typinkompatibilität**:
  - OUT = IN1 (Originalwert bleibt erhalten)
  - Keine Exception-Generierung

Besondere Hinweise

- **Berechnungsgenauigkeit**:
  - Minimum 1 Millisekunde
  - Rundung zur nächsten Millisekunde
- **Gültigkeitsbereich**:
  - Positive Zeitwerte (T#0s bis T#106751d23h47m16s854ms)
  - Divisor muss positiv sein

⚖️ **Vergleich mit Standard-F_DIV**

Merkmal	F_DIVTIME	F_DIV
Eingangstyp 1	TIME	ANY_NUM
Eingangstyp 2	ANY_NUM	ANY_NUM
Ausgangstyp	TIME	ANY_NUM

Fazit

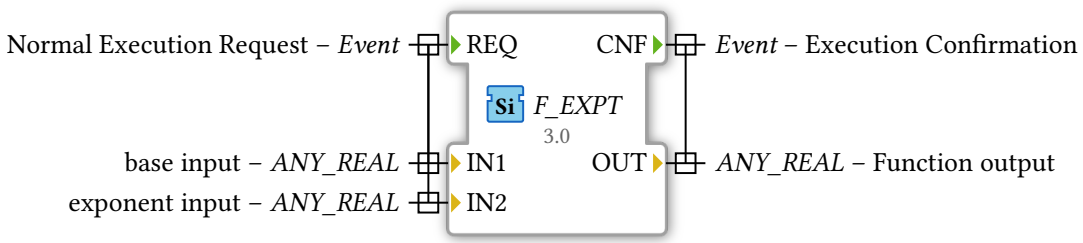
Der F_DIVTIME-Baustein bietet essentielle Funktionen für zeitbasierte Berechnungen:

- Präzise Division von Zeitwerten
- Flexible Eingabeparameter
- Zuverlässige Fehlerbehandlung

Besonders wertvoll für Anwendungen, die eine Aufteilung von Zeitintervallen erfordern. Die spezialisierte Implementierung garantiert korrekte Ergebnisse für alle gültigen Eingabekombinationen.

F_EXPT

• – ◦ • – ◦ • – ◦ •



Einführung

Der **F_EXPT** ist ein mathematischer Funktionsbaustein zur Berechnung von Potenzwerten, entwickelt unter EPL-2.0 Lizenz. Version 1.0 ermöglicht die Berechnung von Basis-Exponent-Kombinationen mit ANY_REAL-Datentypen.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Berechnungsanforderung (mit Basis und Exponent)

Ereignis-Ausgänge

- `CNF`: Berechnungsbestätigung (mit Ergebnis)

Daten-Eingänge

- `IN1` (ANY_REAL): Basiswert
- `IN2` (ANY_REAL): Exponent

Daten-Ausgänge

- `OUT` (ANY_REAL): Ergebnis der Potenzierung (IN1^IN2)

Funktionsweise

1. **Aktivierung**:
   - Durch `REQ`-Ereignis ausgelöst
   - Verarbeitet zwei ANY_REAL-Werte
2. **Berechnung**:
   - Berechnet IN1 hoch IN2
   - Unterstützt alle REAL- und LREAL-Kombinationen
   - Automatische Typanpassung (höhere Genauigkeit)
3. **Ergebnisausgabe**:
   - `CNF`-Ereignis mit berechnetem Potenzwert
   - Ergebnis im genaueren Eingabetyp

Technische Spezifikationen

✓ Präzise Potenzberechnung ✓ Unterstützung für REAL und LREAL ✓ Automatische Typkonvertierung ✓ IEC 61131-3 konform

Anwendungsbeispiele

- **Physikalische Berechnungen**: Beschleunigungsformeln
- **Finanzmathematik**: Zinseszinsberechnung
- **Regelungstechnik**: Nichtlineare Kennlinien
- **Signalverarbeitung**: Exponentialfilter

Fehlerbehandlung

- **Ungültige Eingaben**:
  - Negative Basis mit nicht-ganzzahligem Exponent: OUT = 0
  - Null hoch negativem Exponent: OUT = +INF
  - Unendlichkeitsberechnungen gemäß IEEE 754

Besondere Hinweise

- **Genauigkeit**:
  - LREAL-Eingaben liefern LREAL-Ergebnisse
  - REAL-Eingaben liefern REAL-Ergebnisse
  - Gemischte Eingaben: LREAL-Ergebnis
- **Performance**:
  - LREAL-Berechnungen benötigen mehr Zyklen
  - Für Echtzeitanwendungen REAL bevorzugen

⚖️ Vergleich mit ähnlichen Bausteinen

Merkmal	F_EXPT	F_LN	F_POWER
Funktion	a^b	ln(a)	a^b
Eingangstypen	ANY_REAL	ANY_REAL	ANY_NUM
Genauigkeit	Hoch	Hoch	Mittel

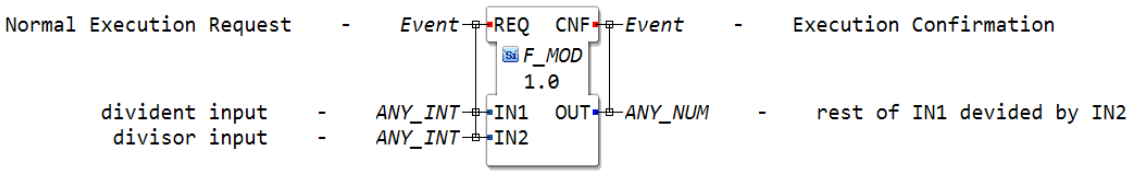
Fazit

Der F_EXPT-Baustein bietet leistungsstarke mathematische Funktionen:

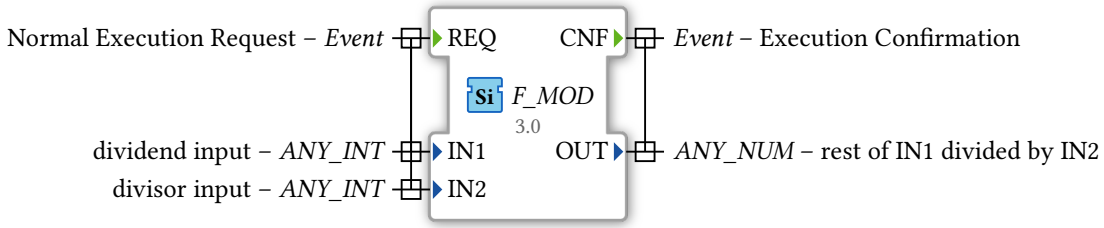
- Präzise Exponentialberechnungen
- Flexible Eingabekombinationen
- Zuverlässige Fehlerbehandlung

Unverzichtbar für fortgeschrittene Steuerungsalgorithmen, die nichtlineare Berechnungen erfordern. Die standardkonforme Implementierung ermöglicht den Einsatz in anspruchsvollen industriellen Anwendungen.

F_MOD



• – ◦ • – ◦ • – ◦ •



Einführung

Der **F_MOD** ist ein arithmetischer Funktionsbaustein zur Berechnung des Divisionsrests, entwickelt unter EPL-2.0 Lizenz. Version 1.0 ermöglicht die Modulo-Operation für ganzzahlige Datentypen gemäß IEC 61131-3 Standard.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Berechnungsanforderung (mit Dividend und Divisor)

Ereignis-Ausgänge

- `CNF`: Berechnungsbestätigung (mit Restwert)

Daten-Eingänge

- `IN1` (ANY_INT): Dividend (ganzzahlig)
- `IN2` (ANY_INT): Divisor (ganzzahlig)

Daten-Ausgänge

- `OUT` (ANY_NUM): Rest der Division (IN1 mod IN2)

Funktionsweise

1. **Aktivierung**:

- Durch `REQ`-Ereignis ausgelöst
- Verarbeitet zwei ganzzahlige Werte

2. **Modulo-Operation**:
   - Berechnet IN1 modulo IN2
   - Formel: OUT = IN1 - (IN1 / IN2) * IN2
   - Unterstützte Typen: INT, DINT, UINT, UDINT
3. **Ergebnisausgabe**:
   - `CNF`-Ereignis mit Restwert
   - Vorzeichen entspricht Dividend (IN1)

Technische Spezifikationen

✓ Ganzzahlige Modulo-Operation ✓ Unterstützung für vorzeichenbehaftete und -lose Typen ✓ IEC 61131-3 konform ✓ Deterministische Ausführung

Anwendungsbeispiele

- **Zyklische Zählung**: Ringpuffer-Indizes
- **Zeitberechnungen**: Wochentagsbestimmung
- **Nachrichtenverarbeitung**: Prüfsummenberechnung
- **Positionierung**: Schrittmotorsteuerung

Fehlerbehandlung

- **Division durch Null**:
  - OUT = IN1 (Originalwert bleibt erhalten)
  - Kein Abbruch der Ausführung
- **Typinkompatibilität**:
  - Automatische Typanpassung
  - Keine Exception-Generierung

Besondere Hinweise

- **Vorzeichenbehandlung**:
  - Ergebnis hat Vorzeichen des Dividenden
  - Beispiel: (-7 mod 3) = −1
- **Performance**:
  - Optimierte Berechnung für alle ganzzahligen Typen
  - Ein Zyklus Latenzzeit

⚖️ **Vergleich mit ähnlichen Bausteinen**

Merkmal	F_MOD	F_DIV	F_REM
Operation	Modulo	Division	Remainder
Eingangstypen	ANY_INT	ANY_NUM	ANY_NUM
Ausgangstyp	ANY_NUM	ANY_NUM	ANY_NUM

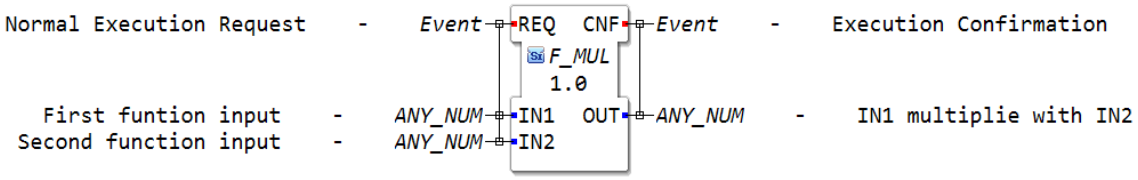
Fazit

Der F_MOD-Baustein bietet essentielle Funktionen für zyklische Berechnungen:

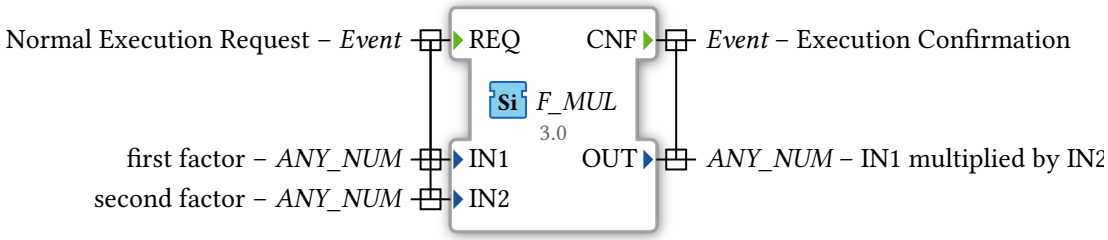
- Effiziente Restwertberechnung
- Flexible Typunterstützung
- Zuverlässige Fehlerbehandlung

Unverzichtbar für Steuerungsanwendungen mit periodischen Mustern oder Indexberechnungen. Die standardkonforme Implementierung ermöglicht den Einsatz in Echtzeitsystemen mit deterministischen Anforderungen.

F_MUL



• – ◦ • – ◦ • – ◦ •



Einführung

Der **F_MUL** ist ein standardkonformer Funktionsbaustein zur Multiplikation numerischer Werte, entwickelt unter EPL-2.0 Lizenz. Version 1.0 ermöglicht die typsichere Multiplikation verschiedener numerischer Datentypen gemäß IEC 61131-3 Standard.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Berechnungsanforderung (mit zwei Faktoren)

Ereignis-Ausgänge

- `CNF`: Berechnungsbestätigung (mit Ergebnis)

Daten-Eingänge

- `IN1` (ANY_NUM): Erster Faktor
- `IN2` (ANY_NUM): Zweiter Faktor

Daten-Ausgänge

- `OUT` (ANY_NUM): Produkt der Multiplikation (IN1 × IN2)

Funktionsprinzip

1. **Berechnungsauslösung**:
   - `REQ`-Ereignis mit Werten für IN1 und IN2
   - Beide Eingänge müssen vom Typ ANY_NUM sein
2. **Multiplikation**:
   - Typenerhaltende Berechnung (IN1 × IN2)
   - Unterstützte Typen:
     - Ganzzahlen (INT, DINT)
     - Fließkommazahlen (REAL, LREAL)
     - Gemischte Typen (automatische Konvertierung)
3. **Ergebnisausgabe**:
   - `CNF`-Ereignis mit berechnetem Produkt
   - Ergebnis im „größten“ Eingangstyp

Technische Besonderheiten

✓ **Typsichere Multiplikation** (ANY_NUM-Unterstützung) ✓ **Automatische Typkonvertierung** ✓ **Überlaufschutz** ✓ **Deterministische Ausführung**

Anwendungsszenarien

- **Prozessregelung**: Skalierungsberechnungen
- **Datenanalyse**: Messwertverarbeitung
- **Maschinensteuerung**: Positionsberechnungen
- **Energiemanagement**: Leistungsberechnungen

Fehlerbehandlung

- **Überlaufbehandlung**:
  - Wrap-around bei Ganzzahlen
  - ±INF (± unendlich) bei Fließkommaüberlauf
- **Typinkompatibilität**:
  - OUT = 0
  - Keine Exception-Generierung

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	F_MUL	F_ADD	F_DIV
Operation	Multiplikation	Addition	Division
Eingänge	2	2	2
Typunterstützung	ANY_NUM	ANY_NUM	ANY_NUM

🛠️ Zugehörige Übungen

- Uebung_034b

Fazit

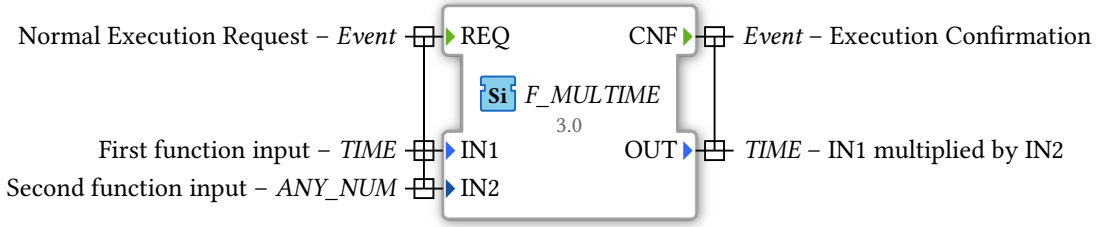
Der F_MUL-Baustein bietet eine robuste Lösung für Multiplikationsoperationen:

- Präzise typerhaltende Berechnungen
- Flexible Eingabekombinationen
- Standardkonforme Implementierung

Besonders wertvoll für Anwendungen, die skalierte Berechnungen oder physikalische Formeln implementieren müssen. Die automatische Typkonvertierung macht ihn zu einem vielseitigen Werkzeug in industriellen Steuerungssystemen.

F_MULTIME

- – ◦ • – ◦ • – ◦ •



Einführung

Der **F_MULTIME** ist ein standardkonformer Funktionsbaustein zur Multiplikation von Zeitwerten mit numerischen Faktoren, entwickelt unter EPL-2.0 Lizenz. Version 1.0 ermöglicht die typsichere Skalierung von Zeitintervallen gemäß IEC 61131-3 Standard.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Berechnungsanforderung (mit Zeitwert und Multiplikator)

Ereignis-Ausgänge

- `CNF`: Berechnungsbestätigung (mit skaliertem Zeitwert)

Daten-Eingänge

- `IN1` (TIME): Zeitwert (Basiswert)
- `IN2` (ANY_NUM): Skalierungsfaktor

Daten-Ausgänge

- `OUT` (TIME): Skalierter Zeitwert (IN1 × IN2)

Funktionsprinzip

1. **Berechnungsauslösung**:
   - `REQ`-Ereignis mit Zeitwert (IN1) und Faktor (IN2)
   - IN1 muss TIME-Typ, IN2 muss ANY_NUM-Typ sein
2. **Multiplikation**:
   - Zeitwert wird in Nanosekunden konvertiert
   - Multiplikation mit numerischem Faktor
   - Rückkonvertierung in TIME-Datentyp
3. **Ergebnisausgabe**:
   - `CNF`-Ereignis mit skaliertem Zeitwert
   - Ergebnis immer vom Typ TIME

Technische Besonderheiten

✓ **Typsichere Zeitwertverarbeitung** ✓ **Präzise Nanosekundenberechnung** ✓ **Automatische Typkonvertierung** (ANY_NUM ⟶ REAL) ✓ **Überlaufschutz** (Begrenzung auf TIME-Maximalwert)

Anwendungsszenarien

- **Zykluszeit-Anpassung**: Skalierung von Zeitintervallen
- **Prozesssteuerung**: Verlängerung/Verkürzung von Timeouts
- **Bewegungssteuerung**: Geschwindigkeitsanpassungen
- **Zeitplanung**: Dynamische Zeitplan-Skalierung

Fehlerbehandlung

- **Überlaufbehandlung**:
  - Begrenzung auf T#24d20h31m23s647ms (Maximum)
  - Keine Wrap-around bei Zeitwerten
- **Typinkompatibilität**:
  - OUT = T#0s
  - Keine Exception-Generierung

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	F_MULTIME	F_MUL	F_DIVTIME
Operation	Zeitmultiplikation	Numerische Multiplikation	Zeitdivision
Eingänge	TIME × ANY_NUM	ANY_NUM × ANY_NUM	TIME ÷ ANY_NUM
Ausgangstyp	TIME	ANY_NUM	TIME

🛠️ Zugehörige Übungen

- Uebung_020c2

Fazit

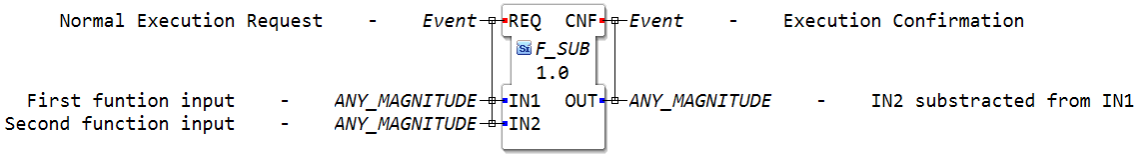
Der F_MULTIME-Baustein bietet eine spezialisierte Lösung für die Skalierung von Zeitwerten:

- Industrietaugliche Präzision (Nanosekundenbasis)

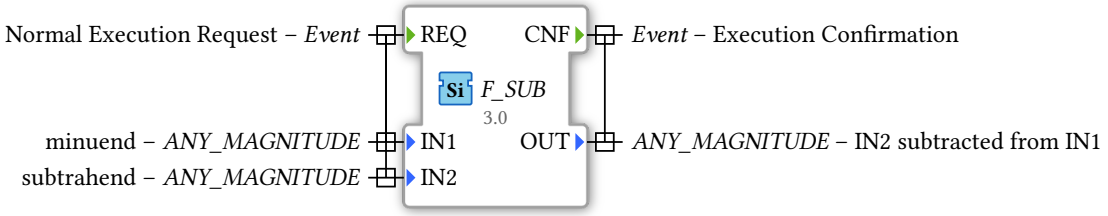
- Sichere Typkonvertierung
- Robuste Überlaufbehandlung

Unverzichtbar für Anwendungen mit dynamisch anpassbaren Zeitparametern und zyklischen Steuerungsprozessen.

F_SUB



• – ◦ • – ◦ • – ◦ •



Einführung

Der **F_SUB** ist ein standardkonformer Funktionsbaustein zur Subtraktion numerischer Werte, entwickelt unter EPL-2.0 Lizenz. Version 1.0 ermöglicht die typsichere Differenzbildung verschiedener numerischer Datentypen gemäß IEC 61131-3 Standard.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Berechnungsanforderung (mit Minuend und Subtrahend)

Ereignis-Ausgänge

- `CNF`: Berechnungsbestätigung (mit Ergebnis)

Daten-Eingänge

- `IN1` (ANY_MAGNITUDE): Minuend (Wert, von dem subtrahiert wird)
- `IN2` (ANY_MAGNITUDE): Subtrahend (Wert, der subtrahiert wird)

Daten-Ausgänge

- `OUT` (ANY_MAGNITUDE): Differenz der Subtraktion (IN1 - IN2)

Funktionsprinzip

1. **Berechnungsauslösung**:
   - `REQ`-Ereignis mit Werten für IN1 und IN2
   - Beide Eingänge müssen vom Typ ANY_MAGNITUDE sein
2. **Subtraktion**:
   - Typenerhaltende Berechnung (IN1 - IN2)
   - Unterstützte Typen:
     - Ganzzahlen (INT, DINT)
     - Fließkommazahlen (REAL, LREAL)
     - Zeitwerte (TIME, DATE)
     - Gemischte Typen (automatische Konvertierung)
3. **Ergebnisausgabe**:
   - `CNF`-Ereignis mit berechneter Differenz
   - Ergebnis im „größten“ Eingangstyp

Technische Besonderheiten

✓ **Typsichere Subtraktion** (ANY_MAGNITUDE-Unterstützung) ✓ **Automatische Typkonvertierung** ✓ **Unterlaufschutz** ✓ **Deterministische Ausführung**

Anwendungsszenarien

- **Prozessregelung**: Regelabweichungsberechnung
- **Datenanalyse**: Differenzbildung von Messwerten
- **Maschinensteuerung**: Positionsdifferenzen
- **Zeitsteuerung**: Zeitintervallberechnungen

Fehlerbehandlung

- **Unterlaufbehandlung**:
  - Wrap-around bei Ganzzahlen
  - -INF bei Fließkommaunterlauf
- **Typinkompatibilität**:
  - OUT = 0
  - Keine Exception-Generierung

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	F_SUB	F_ADD	F_MUL
Operation	Subtraktion	Addition	Multiplikation
Eingänge	2	2	2
Typunterstützung	ANY_MAGNITUDE	ANY_NUM	ANY_NUM

🛠️ Zugehörige Übungen

- Uebung_040
- Uebung_040_2
- Uebung_040_AX
- Uebung_041
- Uebung_072c

Fazit

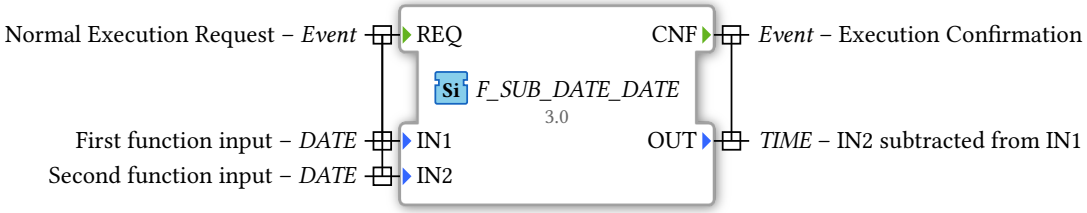
Der F_SUB-Baustein bietet eine robuste Lösung für Subtraktionsoperationen:

- Präzise typerhaltende Berechnungen
- Flexible Eingabekombinationen
- Standardkonforme Implementierung

Besonders wertvoll für Anwendungen, die Differenzberechnungen oder relative Änderungen implementieren müssen. Die automatische Typkonvertierung macht ihn zu einem unverzichtbaren Werkzeug in industriellen Steuerungssystemen.

F_SUB_DATE_DATE

• – ◦ • – ◦ • – ◦ •



Einführung

Der **F_SUB_DATE_DATE** ist ein standardkonformer Funktionsbaustein zur Berechnung von Zeitdifferenzen zwischen zwei Datumswerten, entwickelt unter EPL-2.0 Lizenz. Version 1.0 ermöglicht die präzise Berechnung von Zeitintervallen gemäß IEC 61131-3 Standard.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Berechnungsanforderung (mit Start- und Enddatum)

Ereignis-Ausgänge

- `CNF`: Berechnungsbestätigung (mit Zeitdifferenz)

Daten-Eingänge

- `IN1` (DATE): Startdatum (Minuend)
- `IN2` (DATE): Enddatum (Subtrahend)

Daten-Ausgänge

- `OUT` (TIME): Berechnetes Zeitintervall (IN1 - IN2)

Funktionsprinzip

1. **Berechnungsauslösung**:
   - `REQ`-Ereignis mit zwei DATE-Werten
   - Beide Eingänge müssen vom Typ DATE sein
2. **Differenzberechnung**:
   - Konvertierung beider Datumswerte in Sekunden seit 1970-01-01
   - Subtraktion der Zeitstempel (IN1 - IN2)
   - Umwandlung in TIME-Datentyp
3. **Ergebnisausgabe**:
   - `CNF`-Ereignis mit berechnetem Zeitintervall
   - Positive Werte wenn IN1 > IN2, negative bei IN1 < IN2

Technische Besonderheiten

✓ **Präzise Zeitdifferenz** (Tagesgenau) ✓ **Berücksichtigung von Schaltjahren** ✓ **Negative Zeitintervalle** für rückwirkende Berechnungen ✓ **Plattformunabhängige Zeitberechnung**

Anwendungsszenarien

- **Prozessdauerberechnung**: Produktionszeit zwischen zwei Daten
- **Wartungsintervalle**: Restlaufzeit bis nächste Wartung
- **Projektplanung**: Verbleibende Zeit bis Deadline
- **Lagerverwaltung**: Haltbarkeitsdauer-Berechnung

Fehlerbehandlung

- **Datumsgrenzen**:
  - Unterstützt den Bereich 1970-01-01 bis 2106-02-07
  - Bei Überschreitung: OUT = T#0s
- **Typinkompatibilität**:
  - OUT = T#0s
  - Keine Exception-Generierung

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	F_SUB_DATE_DATE	F_SUB_TIME	F_SUB_DT_DT
Eingangstypen	DATE - DATE	TIME - TIME	DT - DT

Ausgangstyp	TIME	TIME	TIME
Genauigkeit	Tagesgenau	Nanosekunde	Mikrosekunde

Fazit

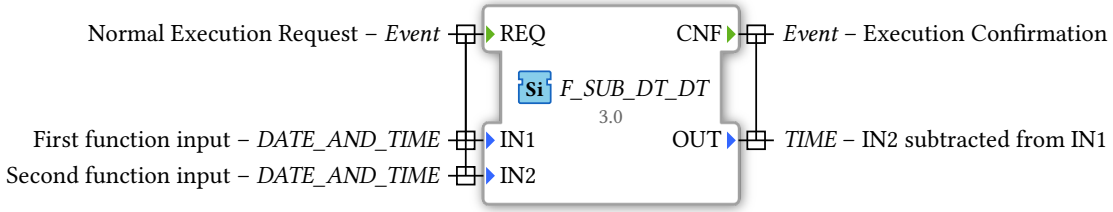
Der F_SUB_DATE_DATE-Baustein bietet eine optimierte Lösung für Datumsdifferenzen:

- Kalendergenaue Berechnungen
- Einfache Handhabung von Zeitintervallen
- Standardkonforme Implementierung

Besonders wertvoll für Anwendungen mit terminbezogenen Berechnungen und langfristigen Zeitplanungen.

F_SUB_DT_DT

• – ◦ • – ◦ • – ◦ •



Einführung

Der **F_SUB_DT_DT** ist ein standardkonformer Funktionsbaustein zur präzisen Berechnung von Zeitdifferenzen zwischen zwei Zeitstempeln (DATE_AND_TIME), entwickelt unter EPL-2.0 Lizenz. Version 1.0 ermöglicht mikrosekundengenaue Zeitintervallberechnungen gemäß IEC 61131-3 Standard.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Berechnungsanforderung (mit Start- und Endzeitstempel)

Ereignis-Ausgänge

- `CNF`: Berechnungsbestätigung (mit Zeitdifferenz)

Daten-Eingänge

- `IN1` (DATE_AND_TIME): Startzeitpunkt (Minuend)
- `IN2` (DATE_AND_TIME): Endzeitpunkt (Subtrahend)

Daten-Ausgänge

- `OUT` (TIME): Berechnetes Zeitintervall (IN1 - IN2)

Funktionsprinzip

1. **Berechnungsauslösung**:
   - `REQ`-Ereignis mit zwei DATE_AND_TIME-Werten
   - Format: DT#1970-01-01-00:00:00 bis DT#2106-02-07-06:28:15
2. **Differenzberechnung**:
   - Konvertierung in Epochen-Mikrosekunden
   - Subtraktion der Zeitstempel (IN1 - IN2)
   - Umwandlung in TIME-Datentyp
3. **Ergebnisausgabe**:
   - `CNF`-Ereignis mit berechnetem Intervall
   - Positive Werte für IN1 > IN2, negative für IN1 < IN2

Technische Besonderheiten

✓ **Mikrosekundengenauigkeit** ✓ **Zeitzonen-unabhängige Berechnung** (UTC-Basis) ✓ **Volle Kalenderarithmetik** (Schaltsekunden, Monatslängen) ✓ **Plattformübergreifende Konsistenz**

Anwendungsszenarien

- **Prozessanalysen**: Exakte Maschinenlaufzeitmessung
- **Laborautomation**: Präzise Reaktionszeitberechnung
- **Energiemonitoring**: Verbrauchszeitraum-Differenzen
- **SPS-Diagnose**: Ereigniszeitstempel-Auswertung

Fehlerbehandlung

- **Zeitbereichsüberschreitung**:
  - Unterstützt DT#1970-01-01-00:00:00 bis DT#2106-02-07-06:28:15
  - Bei Überlauf: OUT = T#0s
- **Ungültige Eingaben**:
  - OUT = T#0s bei fehlerhaften Zeitstempeln
  - Keine Exception-Generierung

⚖️ **Vergleich mit ähnlichen Bausteinen**

Feature	F_SUB_DT_DT	F_SUB_DATE_DATE	F_SUB_TIME
Genauigkeit	Mikrosekunden	Tage	Nanosekunden
Eingangstyp	DATE_AND_TIME	DATE	TIME
Typische Anwendung	Prozessdiagnose	Terminplanung	Zykluszeit

Fazit

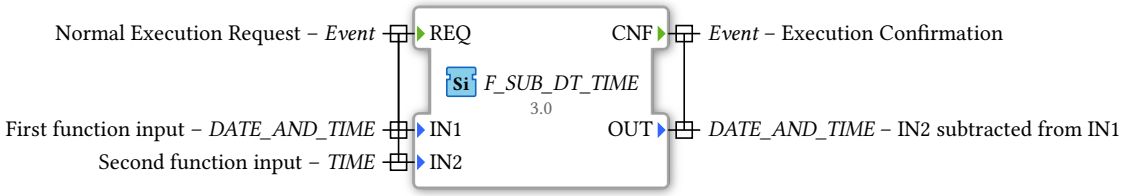
Der F_SUB_DT_DT-Baustein bietet die präziseste Lösung für Zeitstempel-Differenzen:

- Industrietaugliche Mikrosekundengenauigkeit
- Kalendergenaue UTC-Berechnungen
- Robuste Handhabung großer Zeitintervalle

Essential für hochgenaue Prozessanalysen und technische Diagnosesysteme.

F_SUB_DT_TIME

• – ◦ • – ◦ • – ◦ •



Einführung

Der **F_SUB_DT_TIME** ist ein standardkonformer Funktionsbaustein zur Subtraktion von Zeitintervallen von Zeitstempeln, entwickelt unter EPL-2.0 Lizenz. Version 1.0 ermöglicht präzise Zeitpunktberechnungen gemäß IEC 61131-3 Standard.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Berechnungsanforderung (mit Zeitstempel und Zeitintervall)

Ereignis-Ausgänge

- `CNF`: Berechnungsbestätigung (mit korrigiertem Zeitstempel)

Daten-Eingänge

- `IN1` (DATE_AND_TIME): Basiszeitpunkt
- `IN2` (TIME): Abzuziehendes Zeitintervall

Daten-Ausgänge

- `OUT` (DATE_AND_TIME): Berechneter Zeitpunkt (IN1 - IN2)

Funktionsprinzip

1. **Berechnungsauslösung**:
   - `REQ`-Ereignis mit DATE_AND_TIME und TIME-Wert
   - Zeitbereich: DT#1970-01-01-00:00:00 bis DT#2106-02-07-06:28:15
2. **Zeitpunktberechnung**:
   - Konvertierung in Epochen-Mikrosekunden
   - Subtraktion des Zeitintervalls
   - Kalendergenaue Rückrechnung
3. **Ergebnisausgabe**:
   - `CNF`-Ereignis mit neuem Zeitstempel
   - Automatische Kalenderkorrektur (Schaltjahre, Monatsenden)

Technische Besonderheiten

✓ **Mikrosekundengenauigkeit** ✓ **Vollständige Kalenderarithmetik** ✓ **Negative Zeitintervalle** (Zeitpunktverschiebung in die Zukunft) ✓ **Plattformübergreifende Konsistenz**

Anwendungsszenarien

- **Terminplanung**: Fristenberechnung
- **Prozesssteuerung**: Vorherige Startzeitpunktbestimmung
- **Wartungssysteme**: Planung vorbeugender Wartungen
- **Datenanalyse**: Historische Zeitpunktkorrektur

Fehlerbehandlung

- **Zeitbereichsüberschreitung**:
  - Bei Unterlauf: OUT = DT#1970-01-01-00:00:00
  - Bei Überlauf: OUT = DT#2106-02-07-06:28:15
- **Sonderfälle**:
  - IN2 = T#0s ⟶ OUT = IN1
  - IN2 negativ ⟶ Addition zum Zeitpunkt

⚖️ **Vergleich mit ähnlichen Bausteinen**

Feature	F_SUB_DT_TIME	F_ADD_DT_TIME	F_SUB_DT_DT
Operation	Zeitpunkt - Intervall	Zeitpunkt + Intervall	Zeitpunkt - Zeitpunkt
Ergebnis	DATE_AND_TIME	DATE_AND_TIME	TIME
Genauigkeit	Mikrosekunden	Mikrosekunden	Mikrosekunden

Fazit

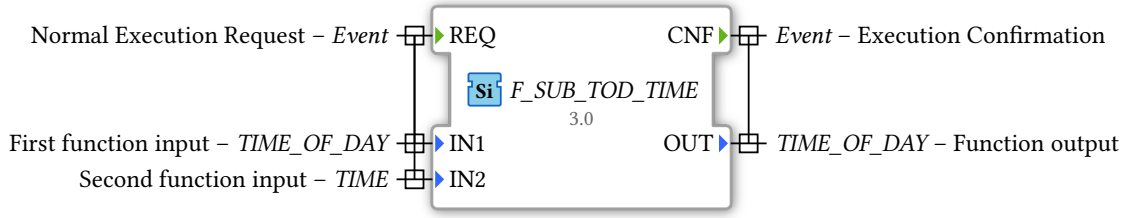
Der F_SUB_DT_TIME-Baustein bietet eine zuverlässige Lösung für Zeitpunktberechnungen:

- Industrietaugliche Präzision
- Volle Kalenderunterstützung
- Intuitive Handhabung von Zeitintervallen

Unverzichtbar für alle Anwendungen mit terminbezogenen Berechnungen und historischen Zeitanalysen.

F_SUB_TOD_TIME

• – ◦ • – ◦ • – ◦ •



Einführung

Der **F_SUB_TOD_TIME** ist ein standardkonformer Funktionsbaustein zur Subtraktion von Zeitintervallen von Tageszeiten (TIME_OF_DAY), entwickelt unter EPL-2.0 Lizenz. Version 1.0 ermöglicht tageszeitbezogene Berechnungen gemäß IEC 61131-3 Standard.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Berechnungsanforderung (mit Tageszeit und Zeitintervall)

Ereignis-Ausgänge

- `CNF`: Berechnungsbestätigung (mit korrigierter Tageszeit)

Daten-Eingänge

- `IN1` (TIME_OF_DAY): Basis-Tageszeit (TOD#hh:mm:ss)
- `IN2` (TIME): Abzuziehendes Zeitintervall

Daten-Ausgänge

- `OUT` (TIME_OF_DAY): Berechnete Tageszeit (IN1 - IN2)

Funktionsprinzip

1. **Berechnungsauslösung**:
   - `REQ`-Ereignis mit TIME_OF_DAY und TIME-Wert
   - Gültiger Bereich: TOD#00:00:00 bis TOD#23:59:59.999999
2. **Zeitpunktberechnung**:
   - Umrechnung in Sekunden seit Mitternacht
   - Subtraktion des Zeitintervalls
   - Tageszeitliche Rückrechnung mit Überlaufbehandlung
3. **Ergebnisausgabe**:
   - `CNF`-Ereignis mit neuer Tageszeit
   - Automatischer 24h-Überlauf (zirkuläre Berechnung)

Technische Besonderheiten

✓ **Mikrosekundengenauigkeit** ✓ **24h-Überlaufbehandlung** (zirkuläre Tageszeit) ✓ **Negative Intervalle** (Addition von Zeit) ✓ **Echtzeitfähige Berechnung**

Anwendungsszenarien

- **Schichtplanung**: Arbeitsbeginn berechnen
- **Lichtsteuerung**: Sonnenaufgangszeiten korrigieren
- **Produktionsplanung**: Maschinenlaufzeiten rückrechnen
- **Verkehrsleitsysteme**: Fahrplananpassungen

Fehlerbehandlung

- **Überlaufbehandlung**:
  - Bei Überschreitung: zirkuläre Berechnung (modulo 24h)
  - Beispiel: TOD#01:00:00 - T#2h = TOD#23:00:00
- **Sonderfälle**:

- IN2 = T#0s ⟶ OUT = IN1
- IN2 negativ ⟶ Addition der Zeit (TOD#10:00 + T#−2h = TOD#12:00)

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	F_SUB_TOD_TIME	F_ADD_TOD_TIME	F_SUB_DT_TIME
Operation	Tageszeit - Intervall	Tageszeit + Intervall	Zeitstempel - Intervall
Ergebnis	TIME_OF_DAY	TIME_OF_DAY	DATE_AND_TIME
Überlauf	Zirkulär 24h	Zirkulär 24h	Kalenderbasiert

Fazit

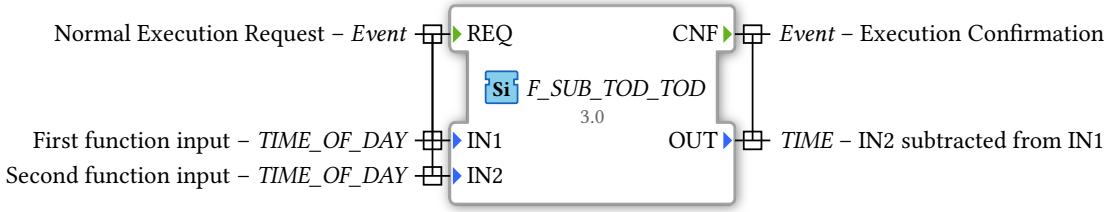
Der F_SUB_TOD_TIME-Baustein bietet eine optimierte Lösung für tageszeitliche Berechnungen:

- Präzise zirkuläre Zeitrechnung
- Einfache Handhabung von Schicht- und Betriebszeiten
- Intuitive Behandlung von Tagesübergängen

Ideale Wahl für alle Anwendungen mit tageszeitabhängigen Steuerungen.

F_SUB_TOD_TOD

• − ◦ • − ◦ • − ◦ •



Einführung

Der **F_SUB_TOD_TOD** ist ein standardkonformer Funktionsbaustein zur Berechnung von Zeitdifferenzen zwischen zwei Tageszeiten (TIME_OF_DAY), entwickelt unter EPL-2.0 Lizenz. Version 1.0 ermöglicht präzise Tageszeitberechnungen gemäß IEC 61131-3 Standard.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Berechnungsanforderung (mit Start- und End-Tageszeit)

Ereignis-Ausgänge

- `CNF`: Berechnungsbestätigung (mit Zeitdifferenz)

Daten-Eingänge

- `IN1` (TIME_OF_DAY): Start-Tageszeit (TOD#hh:mm:ss)
- `IN2` (TIME_OF_DAY): End-Tageszeit

Daten-Ausgänge

- `OUT` (TIME): Berechnetes Zeitintervall (IN1 - IN2)

Funktionsprinzip

1. **Berechnungsauslösung**:
   - `REQ`-Ereignis mit zwei TIME_OF_DAY-Werten
   - Gültiger Bereich: TOD#00:00:00 bis TOD#23:59:59.999999
2. **Differenzberechnung**:
   - Umrechnung beider Werte in Sekunden seit Mitternacht
   - Differenzbildung mit 24h-Überlaufberücksichtigung

- Ergebnis als TIME-Datentyp

3. **Ergebnisausgabe**:
   - `CNF`-Ereignis mit berechnetem Intervall
   - Positive Werte für IN1 > IN2 (innerhalb desselben Tages)
   - Negative Werte bei Tagesübergang

Technische Besonderheiten

✓ Mikrosekundengenauigkeit ✓ Automatische 24h-Überlaufbehandlung ✓ Bidirektionale Differenzberechnung ✓ Echtzeitfähige Ausführung

Anwendungsszenarien

- **Arbeitszeiterfassung**: Schichtdauerberechnung
- **Energiemanagement**: Lastspitzenzeiten-Analyse
- **Produktionslogistik**: Durchlaufzeitmessung
- **Gebäudeautomation**: Tageslichtnutzungszeiten

Fehlerbehandlung

- **Sonderfälle**:
  - IN1 = IN2 ⟶ OUT = T#0s
  - Über Mitternacht: OUT = -(24h - Differenz)
  - Beispiel: TOD#01:00:00 - TOD#23:00:00 = T#−22h
- **Grenzwerte**:
  - Maximale Differenz: ±T#23:59:59.999999
  - Keine Überlaufausnahme

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	F_SUB_TOD_TOD	F_SUB_DT_DT	F_SUB_TIME
Eingangstypen	TOD - TOD	DT - DT	TIME - TIME
Ausgangstyp	TIME	TIME	TIME
Besonderheit	24h-zirkulär	Kalenderbasiert	Einfache Differenz

Fazit

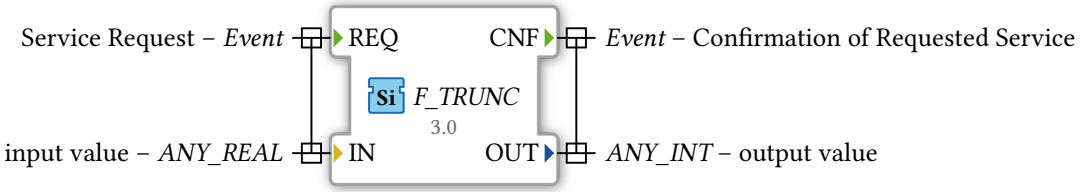
Der F_SUB_TOD_TOD-Baustein bietet die optimale Lösung für tageszeitliche Differenzberechnungen:

- Präzise Behandlung von Tagesübergängen
- Einfache Messung von Betriebszeitintervallen
- Standardkonforme Implementierung

Besonders wertvoll für alle Anwendungen mit tageszeitabhängigen Prozessanalysen.

F_TRUNC

• − ◦ • − ◦ • − ◦ •



Einführung

Der **F_TRUNC** ist ein standardkonformer Funktionsbaustein zur Abrundung von Fließkommazahlen in Richtung Null (truncation toward zero), entwickelt unter EPL-2.0 Lizenz. Version 1.0 ermöglicht die typsichere Konvertierung gemäß IEC 61131-3 Standard.

Schnittstellenstruktur

Ereignis-Eingänge
- `REQ`: Berechnungsanforderung (mit Eingabewert)

Ereignis-Ausgänge
- `CNF`: Berechnungsbestätigung (mit abgerundetem Wert)

Daten-Eingänge
- `IN` (ANY_REAL): Eingabewert (Fließkommazahl)

Daten-Ausgänge
- `OUT` (ANY_INT): Abgerundeter Ganzzahlwert

Funktionsprinzip
1. **Berechnungsauslösung**:
   - `REQ`-Ereignis mit einem ANY_REAL-Wert
   - Unterstützte Typen: REAL, LREAL
2. **Trunkierungsoperation**:
   - Entfernt Nachkommastellen ohne Rundung
   - Beispiel: 3.9 ⟶ 3, −2.7 ⟶ −2
   - Keine mathematische Rundung (≠ F_ROUND)
3. **Ergebnisausgabe**:
   - `CNF`-Ereignis mit ganzzahligem Ergebnis
   - Automatische Typanpassung (INT, DINT, etc.)

Technische Besonderheiten
✓ Verlustfreie Typkonvertierung ✓ Konsistente Null-Richtung (truncation toward zero) **✓ Deterministische Ausführung ✓ Keine Überlaufausnahmen**

Anwendungsszenarien
- **Datenvorverarbeitung**: Indexberechnungen
- **Messwertverarbeitung**: Diskretisierung
- **Steuerungslogik**: Ganzzahlige Positionsberechnungen
- **Finanzberechnungen**: Währungsumrechnungen

Fehlerbehandlung
- **Wertebereichsüberschreitung**:
  - Bei zu großen Werten: Begrenzung auf Zieltyp-Maximum
  - Keine Exception-Generierung
- **Sonderfälle**:
  - NaN ⟶ 0 ±INF ⟶ Maximalwert des Zieltyps

⚖️ Vergleich mit ähnlichen Bausteinen

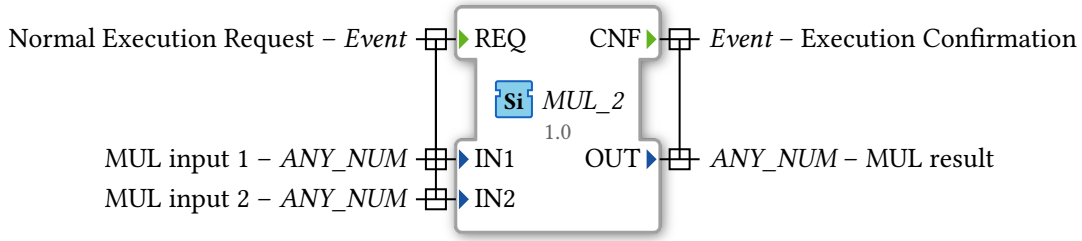
Feature	F_TRUNC	F_ROUND	F_FLOOR
Operation	Richtung Null	Mathematisch	Abrunden
Beispiel	3.9 ⟶ 3	3.9 ⟶ 4	3.9 ⟶ 3
Beispiel	−2.7 ⟶ −2	−2.7 ⟶ −3	−2.7 ⟶ −3

Fazit
Der F_TRUNC-Baustein bietet eine präzise Lösung für die Ganzzahlkonvertierung:
- Vorhersagbares Verhalten (truncation toward zero)

- Typsichere Fließkomma-zu-Ganzzahl Konvertierung
- Robuste Behandlung von Extremwerten

Unverzichtbar für alle Anwendungen mit exakten Bereichsgrenzen und Indexberechnungen.

MUL_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **MUL_2** ist ein generischer Funktionsbaustein, der das arithmetische Produkt zweier numerischer Eingänge berechnet (`OUT = IN1 * IN2`). Die Datentypen der Ports (`ANY_NUM`) werden bei der Instanziierung festgelegt; die native Implementierung ist `GEN_MUL`.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: normale Ausführungsanforderung (mit `IN1`, `IN2`)

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung (mit `OUT`)

Daten-Eingänge

- **IN1** (ANY_NUM): erster Faktor
- **IN2** (ANY_NUM): zweiter Faktor

Daten-Ausgänge

- **OUT** (ANY_NUM): Produkt

Adapter

Es werden keine Adapter verwendet.

Funktionsweise

Bei `REQ` werden `IN1` und `IN2` gelesen, multipliziert und das Produkt an `OUT` zusammen mit dem Ereignis `CNF` ausgegeben.

Technische Besonderheiten

- Generischer Baustein (Generic Class `GEN_MUL`): Die Typen der Ports sind `ANY_NUM` und werden pro Instanz festgelegt
- Zustandsloser Service-Interface-Baustein

Anwendungsszenarien

- Skalierung eines Werts mit einem Faktor in einem Ereignis-Netzwerk
- Flächen-, Leistungs- oder Mengenberechnungen aus zwei Größen

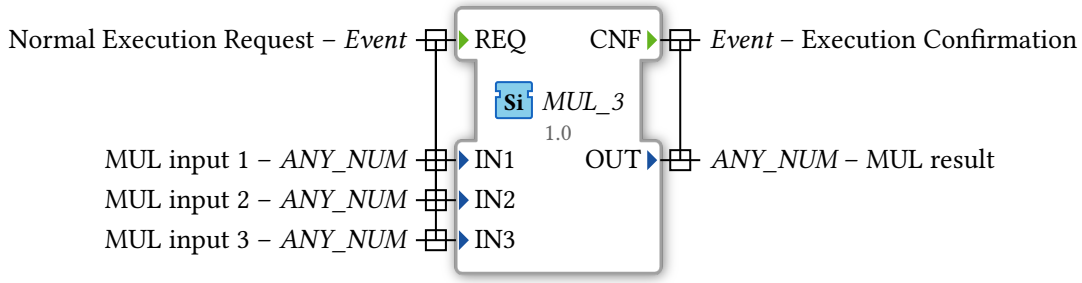
⚖️ Vergleich mit ähnlichen Bausteinen

`ADD_2` bildet entsprechend die Summe zweier Werte; für die Multiplikation mit Adaptern gibt es die AR-basierten Varianten der Adapter-Bibliothek.

Fazit

`MUL_2` multipliziert zwei numerische Werte beliebigen `ANY_NUM`-Typs ereignisgesteuert.

MUL_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **MUL_3** ist ein generischer Funktionsbaustein, der das arithmetische Produkt dreier numerischer Eingänge berechnet (`OUT = IN1 * IN2 * IN3`). Die Datentypen der Ports (`ANY_NUM`) werden bei der Instanziierung festgelegt; die native Implementierung ist `GEN_MUL`.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: normale Ausführungsanforderung (mit `IN1`, `IN2`, `IN3`)

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung (mit `OUT`)

Daten-Eingänge

- **IN1** (ANY_NUM): erster Faktor
- **IN2** (ANY_NUM): zweiter Faktor
- **IN3** (ANY_NUM): dritter Faktor

Daten-Ausgänge

- **OUT** (ANY_NUM): Produkt

Adapter

Es werden keine Adapter verwendet.

Funktionsweise

Bei `REQ` werden `IN1`, `IN2` und `IN3` gelesen, multipliziert und das Produkt an `OUT` zusammen mit dem Ereignis `CNF` ausgegeben.

Technische Besonderheiten

- Generischer Baustein (Generic Class `GEN_MUL`): Die Typen der Ports sind `ANY_NUM` und werden pro Instanz festgelegt
- Zustandsloser Service-Interface-Baustein

Anwendungsszenarien

- Produkt dreier Größen in einem Ereignis-Netzwerk (z. B. Volumenberechnung aus Länge, Breite, Höhe)

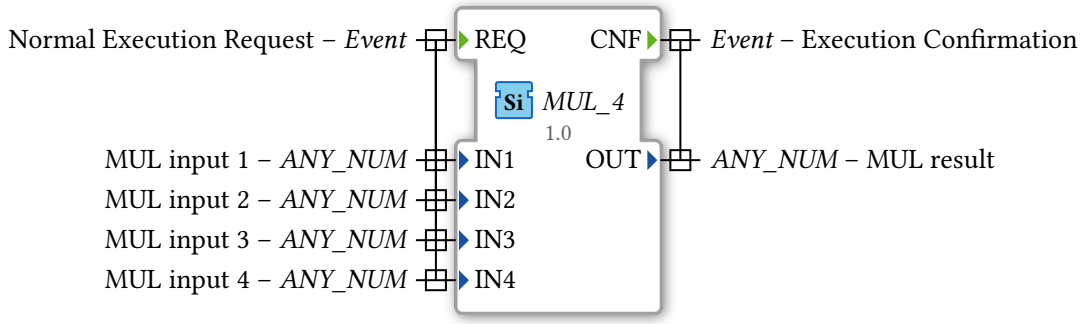
⚖️ Vergleich mit ähnlichen Bausteinen

`MUL_2` multipliziert zwei Werte, `MUL_3` drei.

Fazit

`MUL_3` multipliziert drei numerische Werte beliebigen `ANY_NUM`-Typs ereignisgesteuert.

MUL_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der **MUL_4** ist ein generischer Funktionsbaustein, der das arithmetische Produkt vierer numerischer Eingänge berechnet (`OUT = IN1 * IN2 * IN3 * IN4`). Die Datentypen der Ports (`ANY_NUM`) werden bei der Instanziierung festgelegt; die native Implementierung ist `GEN_MUL`.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: normale Ausführungsanforderung (mit `IN1`, `IN2`, `IN3`, `IN4`)

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung (mit `OUT`)

Daten-Eingänge

- **IN1** (ANY_NUM): erster Faktor
- **IN2** (ANY_NUM): zweiter Faktor
- **IN3** (ANY_NUM): dritter Faktor
- **IN4** (ANY_NUM): vierter Faktor

Daten-Ausgänge

- **OUT** (ANY_NUM): Produkt

Adapter

Es werden keine Adapter verwendet.

Funktionsweise

Bei `REQ` werden `IN1`, `IN2`, `IN3` und `IN4` gelesen, multipliziert und das Produkt an `OUT` zusammen mit dem Ereignis `CNF` ausgegeben.

Technische Besonderheiten

- Generischer Baustein (Generic Class `GEN_MUL`): Die Typen der Ports sind `ANY_NUM` und werden pro Instanz festgelegt
- Zustandsloser Service-Interface-Baustein
- Version 1.0 (2026-08-28)

Anwendungsszenarien

- Produkt vierer Größen in einem Ereignis-Netzwerk

⚖️ Vergleich mit ähnlichen Bausteinen

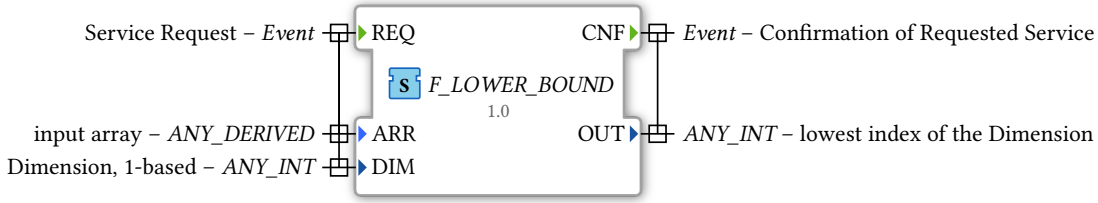
`MUL_2` multipliziert zwei Werte, `MUL_3` drei, `MUL_4` vier.

Fazit

`MUL_4` multipliziert vier numerische Werte beliebigen `ANY_NUM`-Typs ereignisgesteuert.

2.4.1.2 Arrays

F_LOWER_BOUND



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `F_LOWER_BOUND` liefert die untere Indexgrenze einer gewählten Dimension eines Arrays beliebigen Typs. Zusammen mit F_UPPER_BOUND bildet er die Grundlage für generische, größenunabhängige Array-Verarbeitung — z. B. wird `F_LEN_ARRAY` (siehe `utils::arrays`) aus beiden Grenzen berechnet.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Abfrage aus, trägt `ARR` und `DIM`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT`.

Daten-Eingänge

- **ARR** (ANY_DERIVED): Das Array, dessen Indexgrenze ermittelt werden soll.
- **DIM** (ANY_INT, Default `UDINT#1`): Die zu betrachtende Array-Dimension, 1-basiert.

Daten-Ausgänge

- **OUT** (ANY_INT): Der niedrigsten Index der gewählten Dimension.

Funktionsweise

Bei Eintreffen von `REQ` ermittelt der Baustein die untere Indexgrenze der über `DIM` gewählten Dimension von `ARR` und gibt sie über `OUT` aus. Anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **`ANY_DERIVED`-Eingang**: `ARR` akzeptiert Arrays beliebigen Elementtyps und beliebiger Dimensionalität.
- **1-basierte Dimensionsauswahl**: `DIM = 1` bezeichnet die erste Dimension; bei mehrdimensionalen Arrays lässt sich so gezielt eine bestimmte Dimension abfragen.
- **Nicht zwingend `0`/größer-`0`**: IEC-61131-3-Arrays können mit beliebigen Indexgrenzen deklariert werden (auch negativ oder nicht bei `0` beginnend) — `F_LOWER_BOUND` liefert die tatsächlich deklarierte Grenze, keine Annahme über deren Wert.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur Abfrage und zu `CNF`.

Anwendungsszenarien

- **Generische Array-Verarbeitung**: Schleifen, die unabhängig von der tatsächlich deklarierten Größe/den Indexgrenzen eines Arrays über dessen Elemente iterieren.
- **Baustein für Längenberechnung**: `F_LEN_ARRAY` (`utils::arrays`) kombiniert `F_LOWER_BOUND` mit `F_UPPER_BOUND`, um die Elementanzahl einer Dimension zu berechnen.

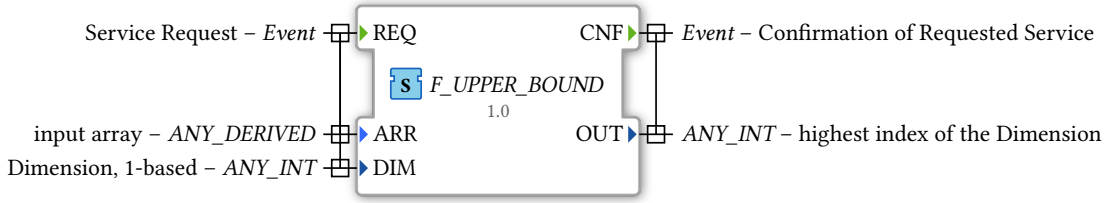
⚖️ Vergleich mit ähnlichen Bausteinen

- F_UPPER_BOUND: die jeweils andere Indexgrenze derselben Dimension.
- **`F_LEN_ARRAY`** (`utils::arrays`): berechnet aus `F_LOWER_BOUND` und `F_UPPER_BOUND` die Elementanzahl.

Fazit

`F_LOWER_BOUND` liefert die untere Indexgrenze einer Array-Dimension und ist ein grundlegender Baustein für generische, größenunabhängige Array-Verarbeitung in IEC-61131-3-Applikationen.

F_UPPER_BOUND



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `F_UPPER_BOUND` liefert die obere Indexgrenze einer gewählten Dimension eines Arrays beliebigen Typs. Zusammen mit F_LOWER_BOUND bildet er die Grundlage für generische, größenunabhängige Array-Verarbeitung — z. B. wird `F_LEN_ARRAY` (siehe `utils::arrays`) aus beiden Grenzen berechnet.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Abfrage aus, trägt `ARR` und `DIM`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT`.

Daten-Eingänge

- **ARR** (ANY_DERIVED): Das Array, dessen Indexgrenze ermittelt werden soll.
- **DIM** (ANY_INT, Default `UDINT#1`): Die zu betrachtende Array-Dimension, 1-basiert.

Daten-Ausgänge

- **OUT** (ANY_INT): Der höchsten Index der gewählten Dimension.

Funktionsweise

Bei Eintreffen von `REQ` ermittelt der Baustein die obere Indexgrenze der über `DIM` gewählten Dimension von `ARR` und gibt sie über `OUT` aus. Anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **`ANY_DERIVED`-Eingang**: `ARR` akzeptiert Arrays beliebigen Elementtyps und beliebiger Dimensionalität.
- **1-basierte Dimensionsauswahl**: `DIM = 1` bezeichnet die erste Dimension; bei mehrdimensionalen Arrays lässt sich so gezielt eine bestimmte Dimension abfragen.
- **Nicht zwingend `0`/größer-`0`**: IEC-61131-3-Arrays können mit beliebigen Indexgrenzen deklariert werden (auch negativ oder nicht bei `0` beginnend) — `F_UPPER_BOUND` liefert die tatsächlich deklarierte Grenze, keine Annahme über deren Wert.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur Abfrage und zu `CNF`.

Anwendungsszenarien

- **Generische Array-Verarbeitung**: Schleifen, die unabhängig von der tatsächlich deklarierten Größe/den Indexgrenzen eines Arrays über dessen Elemente iterieren.
- **Baustein für Längenberechnung**: `F_LEN_ARRAY` (`utils::arrays`) kombiniert `F_UPPER_BOUND` mit `F_LOWER_BOUND`, um die Elementanzahl einer Dimension zu berechnen.

⚖️ Vergleich mit ähnlichen Bausteinen

- F_LOWER_BOUND: die jeweils andere Indexgrenze derselben Dimension.
- **`F_LEN_ARRAY`** (`utils::arrays`): berechnet aus `F_UPPER_BOUND` und `F_LOWER_BOUND` die Elementanzahl.

Fazit

`F_UPPER_BOUND` liefert die obere Indexgrenze einer Array-Dimension und ist ein grundlegender Baustein für generische, größenunabhängige Array-Verarbeitung in IEC-61131-3-Applikationen.

2.4.1.3 Bistabile Elemente

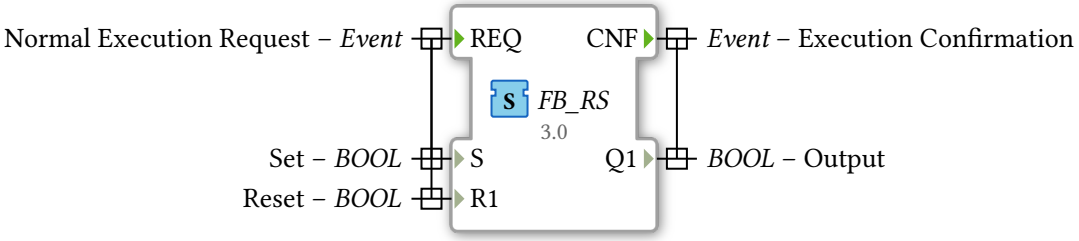
Hier finden Sie Funktionsbausteine für bistabile Elemente (Flip-Flops).

FB_RS

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock FB_RS ist ein bistabiles Element, das als Set-Reset-Flipflop (RS-Flipflop) fungiert. Es speichert einen Zustand und ändert diesen basierend auf den Eingangssignalen S (Set) und R1 (Reset). Der Block ist gemäß dem IEC 61499-1 Standard implementiert und eignet sich für Steuerungsanwendungen, die eine Zustandsspeicherung erfordern.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Löst die Ausführung des Funktionsblocks aus. Verknüpft mit den Daten-Eingängen S und R1.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Wird nach der Verarbeitung der Eingänge ausgegeben. Verknüpft mit dem Daten-Ausgang Q1.

Daten-Eingänge

- **S (BOOL)**: Set-Eingang. Setzt den Ausgang Q1 auf TRUE, wenn S TRUE ist und R1 FALSE ist.
- **R1 (BOOL)**: Reset-Eingang. Setzt den Ausgang Q1 auf FALSE, wenn R1 TRUE ist.

Daten-Ausgänge

- **Q1 (BOOL)**: Ausgang. Repräsentiert den gespeicherten Zustand des RS-Flipflops.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der FB_RS verarbeitet die Eingänge S und R1 bei jedem REQ-Ereignis. Der Ausgang Q1 wird nach folgender Logik aktualisiert:

- Wenn R1 TRUE ist, wird Q1 auf FALSE gesetzt (Reset hat Priorität).
- Wenn S TRUE ist und R1 FALSE ist, wird Q1 auf TRUE gesetzt.
- Wenn beide Eingänge FALSE sind, behält Q1 seinen vorherigen Zustand bei.

Die Logik wird durch den folgenden Algorithmus implementiert:

```
ALGORITHM REQ
Q1 := (NOT R1) AND (S OR Q1);
END_ALGORITHM
```

Technische Besonderheiten

- **IEC 61131-3 Konformität**: Dieser Baustein bildet das Verhalten des klassischen `RS`-Flipflops exakt nach. Da die Eingänge `S` und `R1` Daten-Eingänge sind, die beim `REQ`-Ereignis gleichzeitig abgetastet werden, ist eine logische Priorisierung notwendig.
- **Rücksetz-Dominanz (Priorität)**: Der Reset-Eingang (`R1`) hat Vorrang. Wenn `S` und `R1` gleichzeitig `TRUE` sind, wird der Ausgang `Q1` auf `FALSE` gesetzt.
- **Speicherverhalten**: Der Block behält seinen Zustand bei, solange keine Änderung an `S` oder `R1` erfolgt (oder kein `REQ` Ereignis eintritt).

Zustandsübersicht

Der `FB_RS` wertet die Eingänge bei `REQ` aus:

1. **Q1 = FALSE**: Wenn `R1` TRUE ist (unabhängig von `S`). -> **Rücksetz-Dominanz**
2. **Q1 = TRUE**: Wenn `S` TRUE ist UND `R1` FALSE ist.
3. **Q1 unverändert**: Wenn beide FALSE sind.

Anwendungsszenarien

- Speicherung von Schaltzuständen in Steuerungsanwendungen nach IEC 61131-3 Logik.
- Implementierung von Verriegelungslogik (z.B. Not-Aus-Schaltungen), wo das Abschalten (Reset) immer Vorrang haben muss.

⚖️ Vergleich mit ähnlichen Bausteinen

- **E_RS**: Der `E_RS` ist rein ereignisgesteuert („Last Event Wins“). Der `FB_RS` hingegen wertet statische Signale zum Zeitpunkt `REQ` aus und erzwingt die Rücksetz-Dominanz.
- **FB_SR**: Das Gegenstück mit **Setz-Dominanz**.

🛠️ Zugehörige Übungen

- Uebung_006e2
- Uebung_006e2_AX
- Uebung_160b2
- Uebung_160b2_AX

Fazit

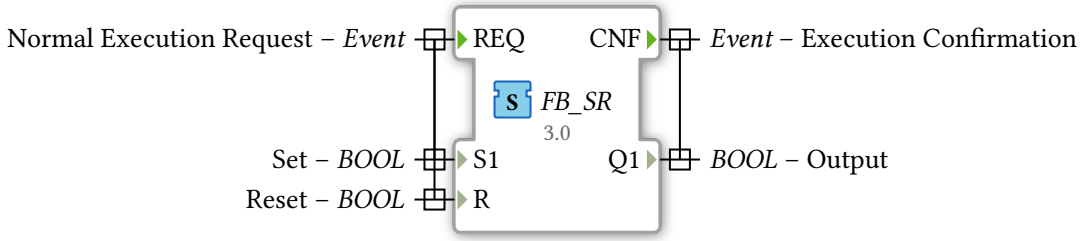
Der FB_RS ist ein grundlegender und robuster Funktionsblock für Zustandsspeicherung in IEC 61499-basierten Steuerungssystemen. Seine einfache und klare Logik macht ihn besonders geeignet für Anwendungen, bei denen ein zuverlässiges Set-Reset-Verhalten erforderlich ist. Die Priorisierung des Reset-Eingangs stellt sicher, dass Sicherheitsfunktionen zuverlässig ausgeführt werden können.

FB_SR

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `FB_SR` ist ein bistabiles Element, das als Set-Reset-Flipflop (SR-Flipflop) fungiert. Es speichert einen Zustand basierend auf den Eingangssignalen und behält diesen bei, bis er durch neue Eingangssignale geändert wird.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung (Trigger für die Ausführung des Funktionsblocks)

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung (wird nach der Verarbeitung der Eingänge ausgelöst)

Daten-Eingänge

- **S1** (BOOL): Set-Eingang (setzt den Ausgang `Q1` auf `TRUE`, wenn `TRUE`)
- **R** (BOOL): Reset-Eingang (setzt den Ausgang `Q1` auf `FALSE`, wenn `TRUE`)

Daten-Ausgänge

- **Q1** (BOOL): Ausgang (gespeicherter Zustand des Flipflops)

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock `FB_SR` verarbeitet die Eingänge `S1` (Set) und `R` (Reset) gemäß der folgenden Logik:

- Wenn `S1` `TRUE` ist, wird `Q1` auf `TRUE` gesetzt.
- Wenn `R` `TRUE` ist, wird `Q1` auf `FALSE` gesetzt.
- Wenn beide Eingänge `FALSE` sind, behält `Q1` seinen vorherigen Zustand bei.

Die Logik wird durch den folgenden Algorithmus implementiert:

```
ALGORITHM REQ
Q1 := S1 OR ((NOT R) AND Q1);
END_ALGORITHM
```

Technische Besonderheiten

- **IEC 61131-3 Konformität**: Dieser Baustein bildet das Verhalten des klassischen `SR`-Flipflops exakt nach. Da die Eingänge `S1` und `R` Daten-Eingänge sind, die beim `REQ`-Ereignis gleichzeitig abgetastet werden, ist eine logische Priorisierung notwendig.
- **Setz-Dominanz**: Der Baustein implementiert eine **Setz-Priorität**. Wenn `S1` und `R` gleichzeitig `TRUE` sind, wird der Ausgang `Q1` auf `TRUE` gesetzt.
- **Trigger**: Die Ausführung (Logikberechnung) erfolgt nur beim Ereignis `REQ`.

Zustandsübersicht

Der Zustand des Flipflops (`Q1`) hängt von den aktuellen Werten von `S1` und `R` ab (zum Zeitpunkt des `REQ`):

- `S1 = TRUE`: `Q1` wird auf `TRUE` gesetzt (unabhängig von `R`). -> **Setz-Dominanz**
- `R = TRUE` UND `S1 = FALSE`: `Q1` wird auf `FALSE` gesetzt.
- `S1 = FALSE` und `R = FALSE`: `Q1` bleibt unverändert.

Anwendungsszenarien

- Speicherung von binären Zuständen in Steuerungsanwendungen, wo IEC 61131-3 Verhalten gefordert ist.
- Verwendung in Schaltnetzen, bei denen die Dominanz des Setz-Signals sicherheitsrelevant oder prozessbedingt notwendig ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- E_SR: Der `E_SR` ist rein ereignisgesteuert. Dort gibt es keine gleichzeitigen Signale und somit keine Dominanz in diesem Sinne (das letzte Ereignis gewinnt). Der `FB_SR` hingegen wertet statische Signale zum Zeitpunkt `REQ` aus und erzwingt die Setz-Dominanz.
- FB_RS: Das Gegenstück mit **Rücksetz-Dominanz** (Reset Priority).

🛠️ Zugehörige Übungen

- Uebung_006e1
- Uebung_006e1_AX

Fazit

Der `FB_SR` ist ein grundlegender und effizienter Funktionsblock für die Speicherung binärer Zustände in Steuerungsanwendungen. Seine einfache Logik und klare Schnittstelle machen ihn ideal für den Einsatz in verschiedenen industriellen Automatisierungslösungen.

2.4.1.4 Bitweise Operatoren

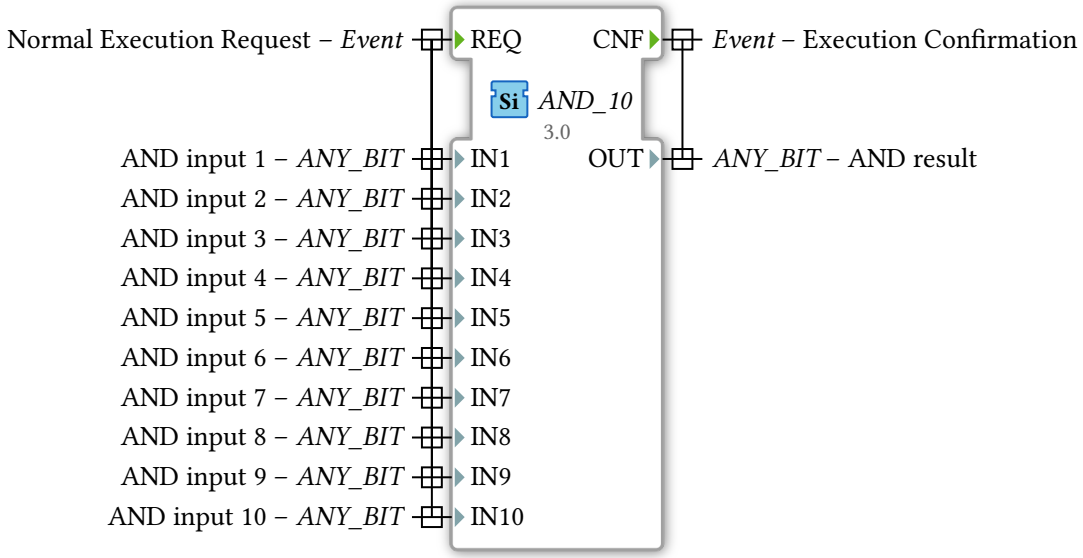
Hier finden Sie Funktionen für bitweise logische Operationen (AND, OR, XOR, Schieben, Rotieren).

AND_10

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `AND_10` Funktionsblock ist ein generischer Baustein zur Berechnung einer bitweisen logischen UND-Verknüpfung. Er unterstützt bis zu 10 Eingangsvariablen und ist gemäß der IEC 61131-3 Norm klassifiziert. Der FB ist Teil des `iec61131::bitwiseOperators` Pakets und eignet sich für Anwendungen, die logische Operationen auf Bit-Ebene erfordern.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung des Funktionsblocks. Wird mit allen Eingängen (`IN1` bis `IN10`) verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Berechnung. Wird mit dem Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN1** bis **IN10** (`ANY_BIT`): Bis zu 10 Eingangsvariablen für die bitweise UND-Verknüpfung. Jeder Eingang kann einen beliebigen Bit-Datentyp (z.B. BOOL, BYTE, WORD, DWORD) annehmen.

Daten-Ausgänge

- **OUT** (`ANY_BIT`): Ergebnis der bitweisen UND-Verknüpfung aller Eingänge. Der Datentyp entspricht dem der Eingänge.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der `AND_10` FB führt eine bitweise logische UND-Operation auf den Werten aller aktiven Eingänge (`IN1` bis `IN10`) aus. Das Ergebnis wird am Ausgang `OUT` ausgegeben, sobald das `REQ`-Ereignis empfangen wird. Die Berechnung wird durch das `CNF`-Ereignis bestätigt.

Technische Besonderheiten

- **Generische Implementierung**: Der FB nutzt die generische Klasse `GEN_AND`, was eine flexible Verwendung mit verschiedenen Bit-Datentypen ermöglicht.
- **Skalierbarkeit**: Unterstützt bis zu 10 Eingänge, was komplexere logische Verknüpfungen ermöglicht.

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf das `REQ`-Ereignis.
2. **Berechnungszustand**: Führt die UND-Verknüpfung aus, sobald `REQ` aktiviert wird.
3. **Ausgabezustand**: Gibt das Ergebnis über `OUT` aus und signalisiert den Abschluss mit `CNF`.

Anwendungsszenarien

- Logische Filterung von Signalgruppen.
- Bitweise Maskierung von Datenwerten.
- Steuerungslogik in industriellen Automatisierungssystemen.

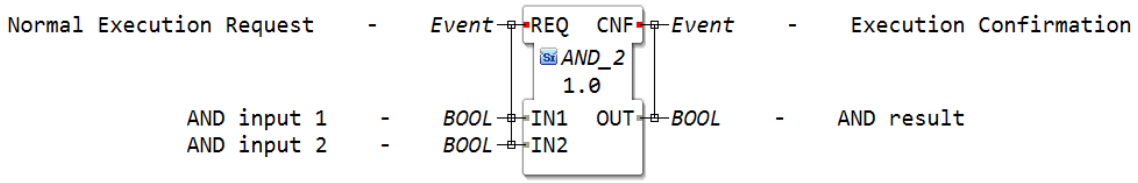
⚖️ Vergleich mit ähnlichen Bausteinen

- **Standard-AND-FB**: Unterstützt typischerweise nur 2 Eingänge. `AND_10` bietet mehr Flexibilität durch 10 Eingänge.
- **Generic Logic Blocks**: Ähnlich generische Blöcke könnten weniger Eingänge oder spezifischere Datentypen unterstützen.

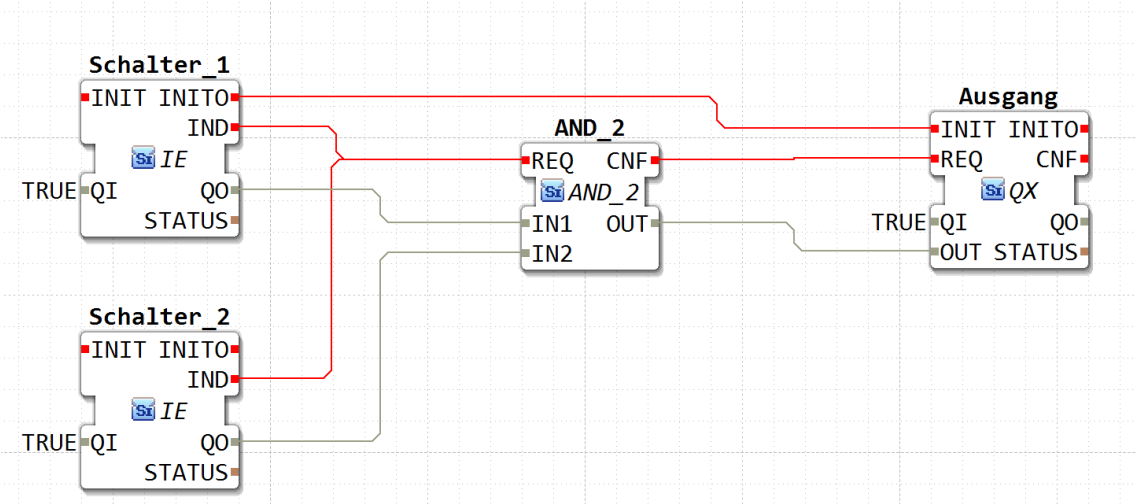
Fazit

Der `AND_10` Funktionsblock ist ein vielseitiges Werkzeug für bitweise logische Operationen in IEC 61131-3-basierten Steuerungssystemen. Seine generische Natur und die Unterstützung für bis zu 10 Eingänge machen ihn besonders nützlich für komplexe Verknüpfungsaufgaben.

AND_2



OUT wird nur geschalten wenn an IN1 und IN2 ein Bool (Signal anliegt) Z.b. von einem Schalter

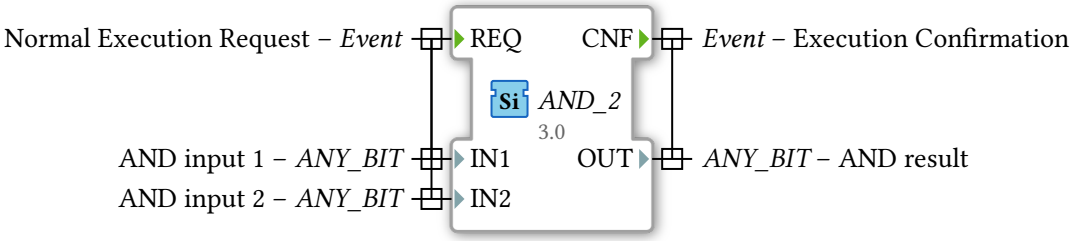


bei allen anderen „AND“ Bausteinen selbe prinzip nur dass immer alle IN eingänge immer True sein müssen

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AND_2 Funktionsblock ist ein generischer Baustein zur Berechnung der bitweisen booleschen UND-Verknüpfung. Er implementiert eine Standard-Funktion gemäß IEC 61131-3 und kann mit beliebigen Bit-Datentypen verwendet werden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung - löst die Berechnung der UND-Verknüpfung aus

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung - signalisiert den Abschluss der Berechnung

Daten-Eingänge

- **IN1** (ANY_BIT): UND-Eingang 1 - erster Operand für die bitweise UND-Verknüpfung
- **IN2** (ANY_BIT): UND-Eingang 2 - zweiter Operand für die bitweise UND-Verknüpfung

Daten-Ausgänge

- **OUT** (ANY_BIT): UND-Ergebnis - Resultat der bitweisen UND-Verknüpfung

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Bei Eintreffen des REQ-Ereignisses führt der Funktionsblock eine bitweise UND-Verknüpfung zwischen den beiden Eingangswerten IN1 und IN2 durch. Das Ergebnis wird am Datenausgang OUT ausgegeben und gleichzeitig wird das CNF-Ereignis zur Bestätigung der Ausführung generiert.

Die Operation wird für jedes Bit der Eingangswerte separat durchgeführt:

- Bit = 1, wenn beide entsprechenden Bits der Eingänge 1 sind
- Bit = 0, wenn mindestens eines der entsprechenden Bits der Eingänge 0 ist

Technische Besonderheiten

- **Generischer Datentyp**: Verwendet ANY_BIT, was die Verwendung mit verschiedenen Bit-Datentypen (BOOL, BYTE, WORD, DWORD, LWORD) ermöglicht
- **Bitweise Operation**: Führt die UND-Operation auf Bitebene durch
- **Ereignisgesteuert**: Die Ausführung wird durch Ereignisse gesteuert, nicht durch Zykluszeiten

Zustandsübergang

1. **Wartezustand**: Block wartet auf REQ-Ereignis
2. **Berechnungszustand**: Bei REQ-Ereignis werden IN1 und IN2 verarbeitet
3. **Ausgabezustand**: Ergebnis wird an OUT ausgegeben und CNF-Ereignis wird generiert
4. Rückkehr in Wartezustand

Anwendungsszenarien

- Bitmasken-Operationen in Steuerungsanwendungen
- Filterung von Daten durch Maskierung unerwünschter Bits
- Logische Verknüpfungen in digitalen Schaltungen
- Datenvorverarbeitung in Kommunikationsprotokollen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen logischen Verknüpfungsblöcken:

- AND_2 ist generisch und unterstützt verschiedene Bit-Datentypen
- Arbeitet bitweise im Gegensatz zu rein booleschen UND-Blöcken

- Ereignisgesteuert im Gegensatz zu zyklusgesteuerten Blöcken

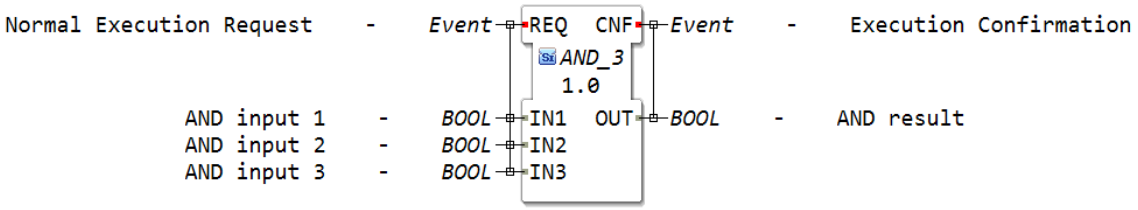
🛠️ Zugehörige Übungen

- Uebung_002a
- Uebung_002a_AX
- Uebung_002b2
- Uebung_002b3_AX
- Uebung_006a3_AX
- Uebung_088

Fazit

Der AND_2 Funktionsblock bietet eine flexible und standardkonforme Lösung für bitweise UND-Operationen in IEC 61131-3 basierten Steuerungssystemen. Seine generische Natur ermöglicht den Einsatz in vielfältigen Anwendungsbereichen, während die ereignisgesteuerte Architektur eine effiziente Ressourcennutzung gewährleistet.

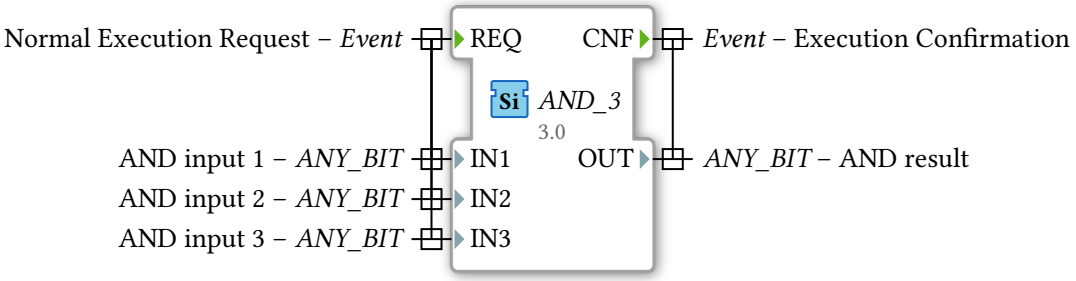
AND_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AND_3-Funktionsblock ist ein generischer Baustein zur Berechnung der bitweisen logischen UND-Verknüpfung mit drei Eingängen. Er führt eine bitweise UND-Operation auf drei Eingabewerten durch und gibt das Ergebnis aus. Der Baustein unterstützt alle ANY_BIT-Datentypen und ist damit flexibel für verschiedene Bitbreiten einsetzbar.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung - löst die Berechnung der UND-Verknüpfung aus

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung - signalisiert die Beendigung der Berechnung

Daten-Eingänge

- **IN1**: UND-Eingang 1 (ANY_BIT)
- **IN2**: UND-Eingang 2 (ANY_BIT)
- **IN3**: UND-Eingang 3 (ANY_BIT)

Daten-Ausgänge

- **OUT**: UND-Ergebnis (ANY_BIT)

Adapter

Keine Adapter vorhanden.

Funktionsweise

Bei Eintreffen des REQ-Ereignisses führt der AND_3-Baustein eine bitweise UND-Verknüpfung der drei Eingangswerte IN1, IN2 und IN3 durch. Das Ergebnis wird am Datenausgang OUT ausgegeben und gleichzeitig wird das CNF-Ereignis generiert, um die erfolgreiche Berechnung zu signalisieren.

Die Operation erfolgt nach der Formel: OUT = IN1 ∧ IN2 ∧ IN3

Technische Besonderheiten

- Generischer Baustein, der mit allen ANY_BIT-Datentypen (BOOL, BYTE, WORD, DWORD, LWORD) funktioniert
- Unterstützt drei Eingänge für erweiterte UND-Verknüpfungen
- Typensicherheit durch konsistente Verwendung des gleichen Datentyps für alle Ein- und Ausgänge

Zustandsübergang

1. **Wartezustand**: Baustein wartet auf REQ-Ereignis
2. **Berechnungszustand**: Bei REQ werden die Eingänge verarbeitet
3. **Ausgabezustand**: Ergebnis wird an OUT ausgegeben und CNF-Ereignis wird generiert
4. Rückkehr in Wartezustand

Anwendungsszenarien

- Bitmasken-Operationen in Steuerungsanwendungen
- Mehrfache Bedingungsprüfungen in logischen Schaltungen
- Signalverarbeitung mit mehreren Eingangssignalen
- Filteroperationen in Datenverarbeitungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-UND-Bausteinen mit nur zwei Eingängen bietet AND_3 die Möglichkeit, drei Eingänge gleichzeitig zu verarbeiten, was die Anzahl benötigter Bausteine in komplexeren Schaltungen reduziert. Gegenüber Bausteinen mit fester Bitbreite bietet AND_3 durch die generische ANY_BIT-Typisierung größere Flexibilität.

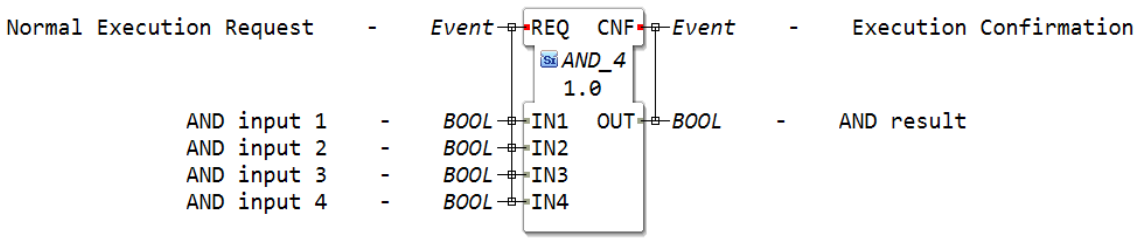
🛠️ Zugehörige Übungen

- Uebung_002a6_AX

Fazit

Der AND_3-Funktionsblock ist ein vielseitiger und effizienter Baustein für bitweise UND-Operationen mit drei Eingängen. Seine generische Natur macht ihn für verschiedene Anwendungsfälle und Datentypen geeignet, während die klare Ereignissteuerung eine zuverlässige Integration in Steuerungssysteme ermöglicht.

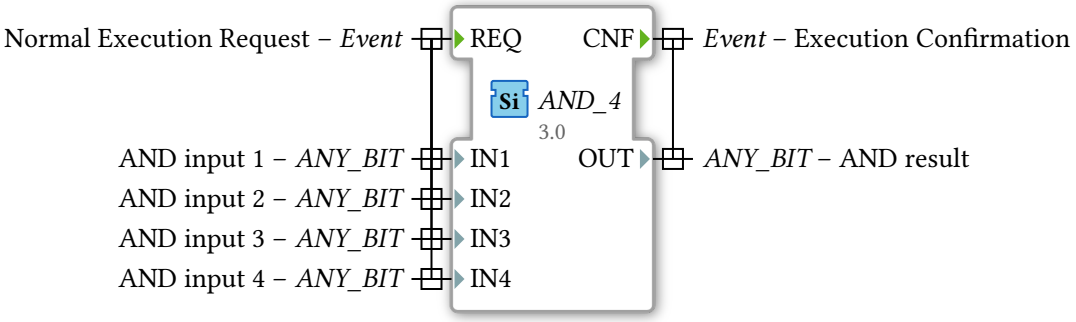
AND_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AND_4 Funktionsblock ist ein generischer Baustein zur Berechnung der bitweisen booleschen UND-Verknüpfung. Er verarbeitet vier Eingangswerte und führt eine logische UND-Operation auf Bitebene durch. Der Baustein ist nach dem IEC 61131-3 Standard klassifiziert und unterstützt beliebige Bit-Datentypen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung - löst die Berechnung aus

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung - signalisiert den Abschluss der Berechnung

Daten-Eingänge

- **IN1**: AND-Eingang 1 (ANY_BIT Datentyp)
- **IN2**: AND-Eingang 2 (ANY_BIT Datentyp)
- **IN3**: AND-Eingang 3 (ANY_BIT Datentyp)
- **IN4**: AND-Eingang 4 (ANY_BIT Datentyp)

Daten-Ausgänge

- **OUT**: AND-Ergebnis (ANY_BIT Datentyp)

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Bei Empfang des REQ-Ereignisses führt der AND_4-Baustein eine bitweise UND-Verknüpfung aller vier Eingangswerte durch. Das Ergebnis wird am Datenausgang OUT bereitgestellt und gleichzeitig wird das CNF-Ereignis zur Bestätigung der erfolgreichen Berechnung ausgegeben. Die Operation wird für jedes Bit der Eingangswerte separat durchgeführt.

Technische Besonderheiten

- Unterstützt den generischen Datentyp ANY_BIT, was die Verwendung mit verschiedenen Bit-Datentypen wie BOOL, BYTE, WORD, DWORD etc. ermöglicht
- Vier unabhängige Eingänge für flexible Verknüpfungsmöglichkeiten
- Ereignisgesteuerte Ausführung für deterministisches Verhalten
- Kompatibel mit IEC 61131-3 Standard

Zustandsübergang

1. **Wartezustand**: Block wartet auf REQ-Ereignis
2. **Berechnungszustand**: Bei REQ-Empfang werden alle vier Eingänge bitweise UND-verknüpft
3. **Ausgabezustand**: Ergebnis wird an OUT ausgegeben und CNF-Ereignis wird getriggert
4. Rückkehr in Wartezustand

Anwendungsszenarien

- Bitmasken-Operationen in Steuerungsanwendungen
- Paritätsprüfungen und Fehlererkennung
- Signalverarbeitung in industriellen Automatisierungssystemen
- Logische Verknüpfungen mehrerer digitaler Signale
- Filteroperationen in Datenverarbeitungssystemen

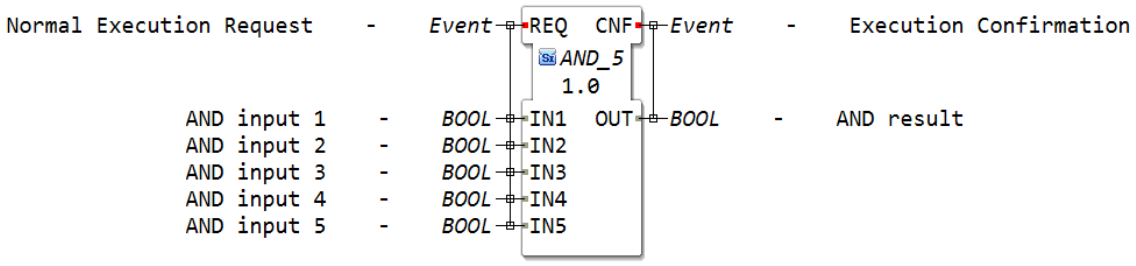
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-AND-Blöcken mit weniger Eingängen bietet AND_4 die Möglichkeit, vier Signale gleichzeitig zu verknüpfen, was komplexere logische Operationen in einem einzigen Schritt ermöglicht. Gegenüber sequenziellen AND-Verknüpfungen mehrerer Blöcke bietet AND_4 eine kompaktere Lösung mit besserer Übersichtlichkeit.

Fazit

Der AND_4 Funktionsblock stellt eine effiziente und flexible Lösung für bitweise UND-Verknüpfungen mit vier Eingängen dar. Seine generische Natur ermöglicht den Einsatz in verschiedenen Anwendungsbereichen, während die ereignisgesteuerte Architektur eine zuverlässige Integration in komplexere Steuerungssysteme gewährleistet.

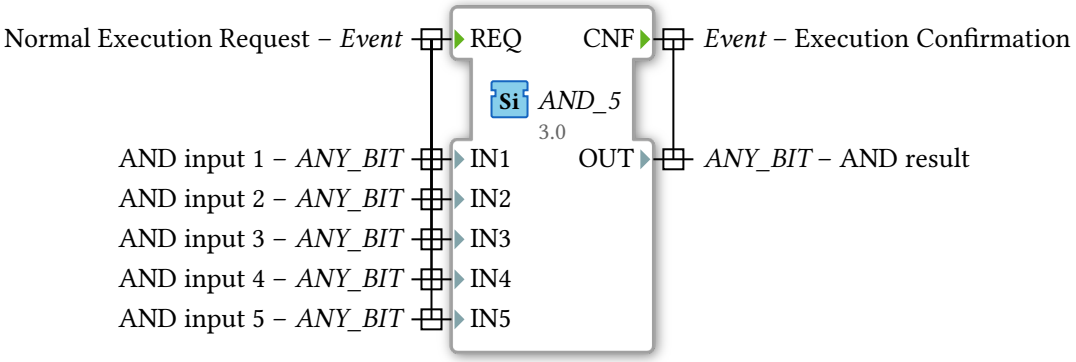
AND_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AND_5-Funktionsblock ist ein generischer Baustein zur Berechnung der bitweisen logischen UND-Verknüpfung mit fünf Eingängen. Er implementiert eine Standard-Bitoperation gemäß IEC 61131-3 und kann mit beliebigen Bit-Datentypen arbeiten.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung - löst die Berechnung der UND-Verknüpfung aus

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung - signalisiert den Abschluss der Berechnung

Daten-Eingänge

- **IN1**: UND-Eingang 1 (ANY_BIT Datentyp)
- **IN2**: UND-Eingang 2 (ANY_BIT Datentyp)
- **IN3**: UND-Eingang 3 (ANY_BIT Datentyp)
- **IN4**: UND-Eingang 4 (ANY_BIT Datentyp)
- **IN5**: UND-Eingang 5 (ANY_BIT Datentyp)

Daten-Ausgänge

- **OUT**: UND-Ergebnis (ANY_BIT Datentyp)

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Bei Eintreffen des REQ-Ereignisses führt der Funktionsblock eine bitweise UND-Operation über alle fünf Eingänge durch. Das Ergebnis wird am Datenausgang OUT ausgegeben und gleichzeitig wird das CNF-Ereignis zur Bestätigung der abgeschlossenen Berechnung gesendet.

Die Operation wird für jedes Bitposition separat durchgeführt: $OUT_i = IN1_i \wedge IN2_i \wedge IN3_i \wedge IN4_i \wedge IN5_i$

Technische Besonderheiten

- **Generischer Datentyp**: Verwendet ANY_BIT, wodurch der Baustein mit verschiedenen Bit-Datentypen (BOOL, BYTE, WORD, DWORD, LWORD) kompatibel ist
- **Fünf Eingänge**: Ermöglicht die Verknüpfung von bis zu fünf verschiedenen Bitwerten
- **Ereignisgesteuert**: Die Berechnung wird nur bei Anforderung durch das REQ-Ereignis durchgeführt

Zustandsübergang

1. **Wartezustand**: Block wartet auf REQ-Ereignis
2. **Berechnungszustand**: Bei REQ-Eingang werden alle Eingänge bitweise UND-verknüpft
3. **Ausgabezustand**: Ergebnis wird an OUT ausgegeben und CNF-Ereignis wird gesendet
4. Rückkehr in Wartezustand

Anwendungsszenarien

- Bitmasken-Operationen in Steuerungsanwendungen
- Mehrfache Bedingungsprüfungen in Sicherheitssystemen
- Signalverarbeitung mit mehreren Eingangssignalen
- Filteroperationen in Datenverarbeitungssystemen

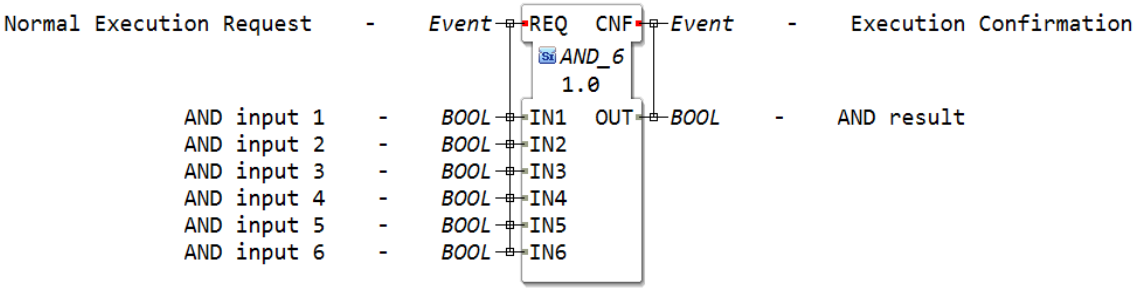
⚖️ Vergleich mit ähnlichen Bausteinen

- **AND_2**: Bietet nur zwei Eingänge für die UND-Verknüpfung
- **OR_5**: Führt bitweise ODER-Verknüpfungen mit fünf Eingängen durch
- **XOR_5**: Führt bitweise exklusive ODER-Verknüpfungen durch

Fazit

Der AND_5-Funktionsblock bietet eine flexible und effiziente Lösung für bitweise UND-Operationen mit fünf Eingängen. Seine generische Natur ermöglicht den Einsatz mit verschiedenen Bit-Datentypen, was ihn zu einem vielseitigen Baustein für komplexe logische Verknüpfungen in Automatisierungsanwendungen macht.

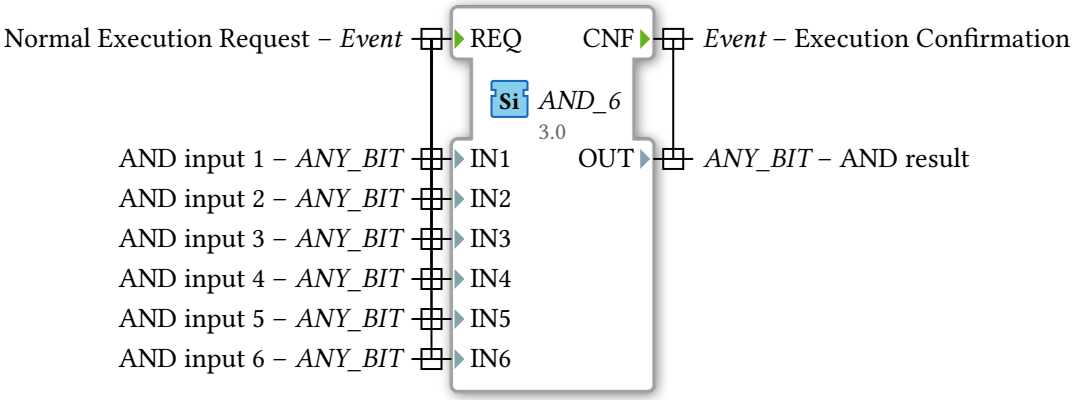
AND_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der AND_6 Funktionsblock ist ein generischer Baustein zur Berechnung der bitweisen booleschen UND-Verknüpfung mit sechs Eingängen. Er implementiert eine Standard-Funktion gemäß IEC 61131-3 und führt eine logische UND-Operation auf Bit-Ebene für bis zu sechs Eingangswerte durch.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Normal Execution Request): Startet die Ausführung der UND-Operation. Löst die Verarbeitung der Eingangsdaten aus.

Ereignis-Ausgänge

- **CNF** (Execution Confirmation): Signalisiert den Abschluss der Berechnung und die Verfügbarkeit des Ergebnisses.

Daten-Eingänge

- **IN1** (ANY_BIT): UND-Eingang 1
- **IN2** (ANY_BIT): UND-Eingang 2
- **IN3** (ANY_BIT): UND-Eingang 3
- **IN4** (ANY_BIT): UND-Eingang 4
- **IN5** (ANY_BIT): UND-Eingang 5
- **IN6** (ANY_BIT): UND-Eingang 6

Daten-Ausgänge

- **OUT** (ANY_BIT): UND-Ergebnis der bitweisen Verknüpfung aller sechs Eingänge

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Bei Aktivierung des REQ-Ereignisses führt der AND_6-Block eine bitweise UND-Verknüpfung aller sechs Eingangswerte durch. Die Operation wird für jedes Bitposition separat ausgeführt: Das Ergebnisbit an Position i ist nur dann 1, wenn alle sechs Eingangsbits an Position i den Wert 1 haben. Nach Abschluss der Berechnung wird das CNF-Ereignis ausgelöst und das Ergebnis am OUT-Ausgang verfügbar gemacht.

Technische Besonderheiten

- Verwendet den generischen Datentyp ANY_BIT, der verschiedene Bit-Datentypen (BOOL, BYTE, WORD, DWORD, LWORD) unterstützt
- Alle Eingänge müssen denselben Datentyp aufweisen
- Das Ergebnis hat denselben Datentyp wie die Eingänge
- Implementiert als generischer Funktionsblock mit der Klasse ‚GEN_AND‘

Zustandsübergang

1. **Wartezustand**: Block wartet auf REQ-Ereignis
2. **Berechnungszustand**: Bei REQ werden alle Eingänge bitweise UND-verknüpft
3. **Ausgabezustand**: Ergebnis wird an OUT ausgegeben und CNF-Ereignis wird ausgelöst
4. Rückkehr in Wartezustand

Anwendungsszenarien

- Bitmasken-Operationen in Steuerungsanwendungen
- Mehrfache Bedingungsprüfungen in logischen Schaltungen
- Signalverarbeitung mit mehreren Eingangssignalen
- Sicherheitsrelevante Schaltungen mit mehreren Überwachungssignalen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren UND-Blöcken mit weniger Eingängen bietet AND_6 die Möglichkeit, bis zu sechs Signale gleichzeitig zu verknüpfen, was die Programmstruktur vereinfacht und weniger Verknüpfungsbausteine erfordert. Gegenüber seriellen UND-Verknüpfungen bietet er eine bessere Übersichtlichkeit und geringere Latenz.

Fazit

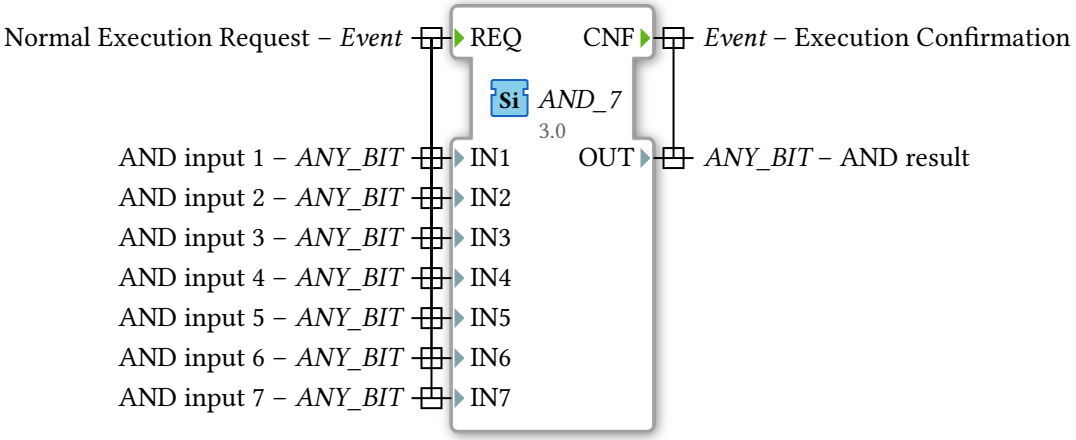
Der AND_6 Funktionsblock stellt eine effiziente und übersichtliche Lösung für komplexe bitweise UND-Verknüpfungen mit bis zu sechs Eingängen dar. Seine generische Implementierung ermöglicht den Einsatz mit verschiedenen Bit-Datentypen und macht ihn zu einem vielseitigen Werkzeug in der Automatisierungstechnik.

AND_7

• – ◦ • – ◦ • – ◦ •

Einleitung

Der AND_7 Funktionsblock führt eine bitweise logische UND-Verknüpfung (AND) auf bis zu sieben Eingangsvariablen durch. Es handelt sich um einen generischen Funktionsblock, der mit verschiedenen Bit-Datentypen (ANY_BIT) arbeiten kann. Der Block ist gemäß der IEC 61131-3 Norm klassifiziert und eignet sich für den Einsatz in Steuerungs- und Automatisierungssystemen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Normal Execution Request): Startet die Ausführung des Funktionsblocks. Wird mit allen sieben Eingangsvariablen (IN1 bis IN7) verknüpft.

Ereignis-Ausgänge

- **CNF** (Execution Confirmation): Signalisiert die erfolgreiche Ausführung und liefert das Ergebnis der UND-Verknüpfung über die Variable OUT.

Daten-Eingänge

- **IN1** (ANY_BIT): Erster Eingang für die UND-Verknüpfung
- **IN2** (ANY_BIT): Zweiter Eingang für die UND-Verknüpfung
- **IN3** (ANY_BIT): Dritter Eingang für die UND-Verknüpfung
- **IN4** (ANY_BIT): Vierter Eingang für die UND-Verknüpfung
- **IN5** (ANY_BIT): Fünfter Eingang für die UND-Verknüpfung

- **IN6** (ANY_BIT): Sechster Eingang für die UND-Verknüpfung
- **IN7** (ANY_BIT): Siebter Eingang für die UND-Verknüpfung

Daten-Ausgänge

- **OUT** (ANY_BIT): Ergebnis der bitweisen UND-Verknüpfung aller Eingangswerte

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der AND_7 Block führt bei jedem REQ-Ereignis eine bitweise UND-Verknüpfung aller aktiven Eingänge (IN1 bis IN7) durch. Das Ergebnis wird an OUT ausgegeben und das CNF-Ereignis ausgelöst. Der Block ist generisch und kann mit verschiedenen Bit-Datentypen (wie BOOL, BYTE, WORD, DWORD, LWORD) arbeiten.

Technische Besonderheiten

- Unterstützt den generischen Datentyp ANY_BIT
- Implementiert als generischer Funktionsblock (GEN_AND)
- Teil des iec61131::bitwiseOperators-Pakets
- EPL-2.0 Lizenz

Zustandsübersicht

Der Block hat keine internen Zustände. Bei jedem REQ-Ereignis wird die Operation sofort ausgeführt und das Ergebnis über CNF zurückgemeldet.

Anwendungsszenarien

- Bitweise Maskenoperationen in Steuerungsanwendungen
- Logische Verknüpfung mehrerer digitaler Signale
- Signalverarbeitung in Automatisierungssystemen
- Generische Logikoperationen in IEC 61131-3 kompatiblen Systemen

⚖️ Vergleich mit ähnlichen Bausteinen

- Standard AND-Blöcke bieten typischerweise nur 2-4 Eingänge
- AND_7 ermöglicht die Verarbeitung von bis zu sieben Eingängen in einer Operation
- Durch die generische Implementierung flexibler als typenspezifische Blöcke

Fazit

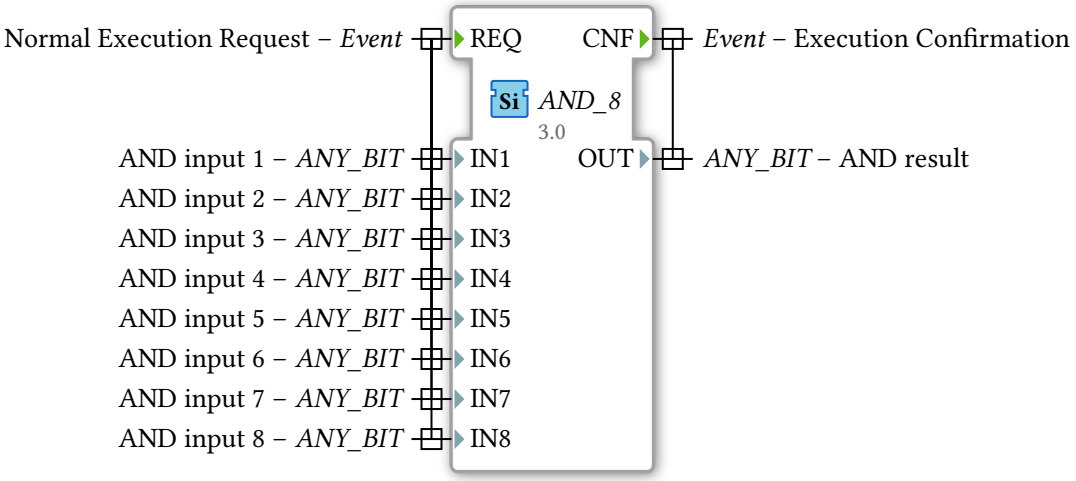
Der AND_7 Funktionsblock bietet eine erweiterte UND-Verknüpfungsfunktionalität für Systeme, die die Verarbeitung mehrerer Eingangssignale in einer einzigen Operation erfordern. Seine generische Natur und IEC 61131-3 Kompatibilität machen ihn zu einem vielseitigen Werkzeug für Automatisierungsanwendungen.

AND_8

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AND_8` führt eine bitweise logische UND-Operation auf bis zu acht Eingängen durch. Es handelt sich um einen generischen Funktionsblock, der mit verschiedenen Bit-Datentypen arbeiten kann. Der Block ist gemäß der Norm IEC 61131-3 klassifiziert und dient zur Verarbeitung von booleschen Operationen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung des Funktionsblocks. Die Operation wird ausgeführt, wenn dieses Ereignis eintritt. Es ist mit allen Daten-Eingängen verknüpft (`IN1` bis `IN8`).

Ereignis-Ausgänge

- **CNF**: Bestätigt die Ausführung des Funktionsblocks. Dieses Ereignis wird nach der Berechnung ausgegeben und ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN1** bis **IN8** (`ANY_BIT`): Acht Eingänge für die bitweise UND-Operation. Jeder Eingang kann einen beliebigen Bit-Datentyp (z.B. BOOL, BYTE, WORD, DWORD, LWORD) annehmen.

Daten-Ausgänge

- **OUT** (`ANY_BIT`): Das Ergebnis der bitweisen UND-Operation aller Eingänge. Der Datentyp entspricht dem der Eingänge.

Adapter

Der Funktionsblock verfügt über keine Adapter.

Funktionsweise

Der `AND_8`-Block führt eine bitweise UND-Operation auf den Werten aller aktiven Eingänge (`IN1` bis `IN8`) aus. Das Ergebnis wird am Ausgang `OUT` ausgegeben. Die Operation wird durch das Ereignis `REQ` gestartet und durch `CNF` bestätigt.

Technische Besonderheiten

- **Generische Implementierung**: Der Block kann mit verschiedenen Bit-Datentypen arbeiten, da die Eingänge und Ausgänge vom Typ `ANY_BIT` sind.
- **Flexible Eingangsanzahl**: Bis zu acht Eingänge können verwendet werden, wobei nicht alle zwingend belegt sein müssen.

Zustandsübersicht

1. **Idle-Zustand**: Der Block wartet auf das Ereignis `REQ`.
2. **Ausführungszustand**: Bei `REQ` werden die Eingänge verarbeitet und das Ergebnis berechnet.
3. **Bestätigungszustand**: Nach der Berechnung wird `CNF` ausgelöst und das Ergebnis an `OUT` ausgegeben.

Anwendungsszenarien

- Logische Verknüpfung mehrerer Bit-Signale in Steuerungsanwendungen.
- Filterung von Signalen durch bitweise Maskierung.
- Generische Verarbeitung von Bit-Daten in komplexen Automatisierungssystemen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **AND (Standard)**: Standard-UND-Block mit typischerweise zwei Eingängen. `AND_8` bietet mehr Flexibilität durch acht Eingänge und generische Typen.
- **GEN_AND**: Generische UND-Implementierung, die als Basis für `AND_8` dient. `AND_8` ist eine spezialisierte Version mit fester Eingangsanzahl.

Fazit

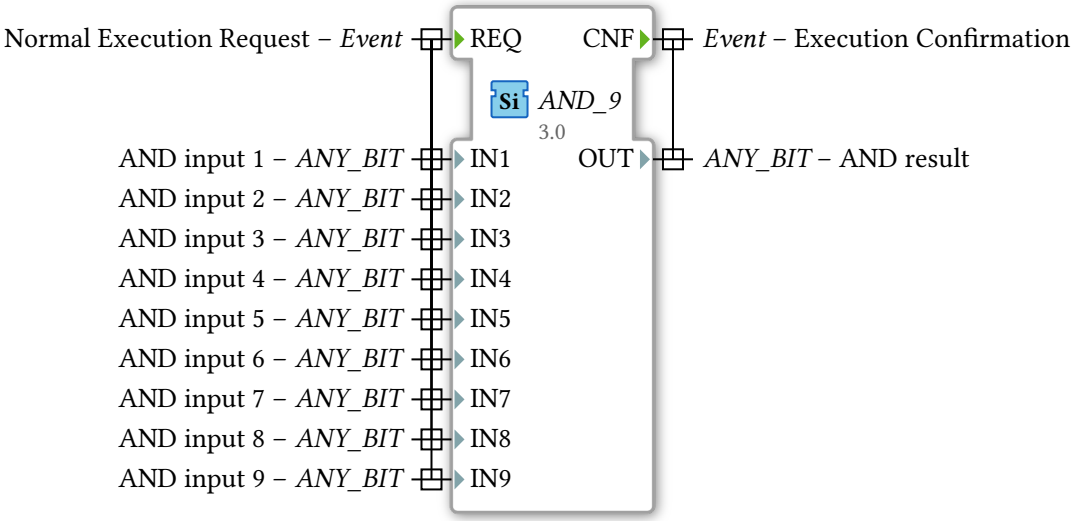
Der `AND_8`-Funktionsblock ist ein leistungsfähiges Werkzeug für bitweise logische Operationen in IEC 61131-3-konformen Steuerungssystemen. Seine generische Natur und flexible Eingangsanzahl machen ihn besonders vielseitig einsetzbar.

AND_9

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `AND_9` führt eine bitweise logische UND-Operation (AND) auf bis zu neun Eingangsvariablen durch. Es handelt sich um einen generischen Funktionsblock, der mit verschiedenen Bit-Datentypen arbeiten kann. Der Block ist gemäß dem Standard IEC 61131-3 klassifiziert und eignet sich für Anwendungen in der Automatisierungstechnik.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ` (Normal Execution Request): Startet die Ausführung des Funktionsblocks. Die Operation wird ausgeführt, wenn dieses Ereignis eintritt.

Ereignis-Ausgänge

- `CNF` (Execution Confirmation): Signalisiert die erfolgreiche Beendigung der Operation.

Daten-Eingänge

- `IN1` bis `IN9` (ANY_BIT): Bis zu neun Eingangsvariablen, auf denen die bitweise UND-Operation durchgeführt wird. Jeder Eingang kann einen beliebigen Bit-Datentyp (z.B. BOOL, BYTE, WORD, DWORD, LWORD) haben.

Daten-Ausgänge

- `OUT` (ANY_BIT): Das Ergebnis der bitweisen UND-Operation aller Eingangsvariablen. Der Datentyp entspricht dem der Eingangsvariablen.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock `AND_9` führt eine bitweise UND-Operation auf den Werten der Eingangsvariablen `IN1` bis `IN9` durch. Das Ergebnis wird an den Ausgang `OUT` ausgegeben. Die Operation wird durch das Ereignis `REQ` gestartet und nach Abschluss durch das Ereignis `CNF` bestätigt.

Technische Besonderheiten

- **Generische Implementierung**: Der Funktionsblock kann mit verschiedenen Bit-Datentypen arbeiten, was eine hohe Flexibilität ermöglicht.
- **Skalierbarkeit**: Unterstützt bis zu neun Eingangsvariablen, was komplexere logische Operationen ermöglicht.

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf das Ereignis `REQ`.
2. **Ausführungszustand**: Führt die bitweise UND-Operation durch.
3. **Bestätigungszustand**: Sendet das Ereignis `CNF` und gibt das Ergebnis an `OUT` aus.

Anwendungsszenarien

- Logische Verknüpfung mehrerer digitaler Signale in der Automatisierungstechnik.
- Filterung von Signalen durch logische Masken.
- Implementierung von Sicherheitsfunktionen, bei denen mehrere Bedingungen gleichzeitig erfüllt sein müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **AND (Standard)**: Standardmäßig unterstützen viele PLC-Systeme nur UND-Operationen mit zwei Eingängen. `AND_9` erweitert diese Funktionalität auf bis zu neun Eingänge.
- **GEN_AND**: Der generische AND-Block, auf dem `AND_9` basiert, kann für verschiedene Datentypen und eine variable Anzahl von Eingängen konfiguriert werden.

Fazit

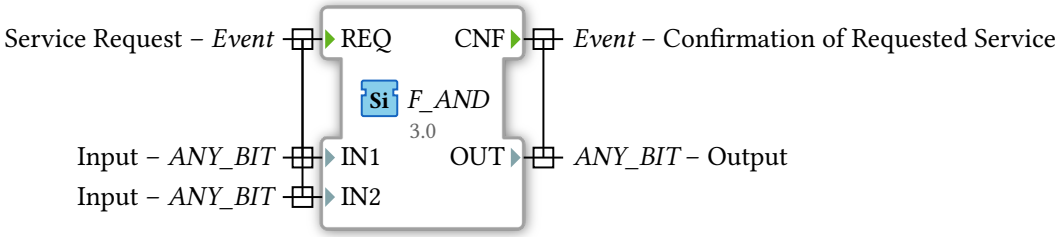
Der `AND_9`-Funktionsblock ist ein leistungsfähiges Werkzeug für bitweise logische Operationen in der Automatisierungstechnik. Seine generische Natur und die Unterstützung für bis zu neun Eingänge machen ihn besonders flexibel und vielseitig einsetzbar. Ideal für Anwendungen, bei denen komplexe logische Verknüpfungen erforderlich sind.

F_AND

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_AND` ist ein standardisierter bitweiser boolescher UND-Operator gemäß IEC 61131-3. Er führt eine bitweise UND-Verknüpfung auf zwei Eingangsvariablen durch und gibt das Ergebnis an den Ausgang weiter. Dieser Block ist besonders nützlich für logische Operationen in Steuerungsanwendungen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Dienstaufforderung. Löst die Ausführung der UND-Operation aus. Verknüpft mit den Daten-Eingängen `IN1` und `IN2`.

Ereignis-Ausgänge

- **CNF**: Bestätigung der angeforderten Dienstleistung. Wird nach erfolgreicher Ausführung der Operation ausgelöst. Verknüpft mit dem Daten-Ausgang `OUT`.

Daten-Eingänge

- **IN1**: Erster Eingang für die bitweise UND-Operation. Typ: `ANY_BIT`.
- **IN2**: Zweiter Eingang für die bitweise UND-Operation. Typ: `ANY_BIT`.

Daten-Ausgänge

- **OUT**: Ergebnis der bitweisen UND-Operation. Typ: `ANY_BIT`.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der `F_AND`-Block führt eine bitweise UND-Verknüpfung zwischen den Eingangsvariablen `IN1` und `IN2` durch, wenn das Ereignis `REQ` ausgelöst wird. Das Ergebnis wird an den Ausgang `OUT` weitergegeben und das Ereignis `CNF` wird ausgelöst, um die erfolgreiche Ausführung zu bestätigen.

Technische Besonderheiten

- Unterstützt den Datentyp `ANY_BIT`, was bedeutet, dass verschiedene Bit-Typen wie `BOOL`, `BYTE`, `WORD`, `DWORD`, etc. verwendet werden können.
- Der Block ist standardkonform nach IEC 61131-3 und in der Kategorie „standard bitwise boolean function“ klassifiziert.

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf das `REQ`-Ereignis.
2. **Ausführungszustand**: Führt die bitweise UND-Operation aus, wenn `REQ` empfangen wird.
3. **Bestätigungszustand**: Sendet das `CNF`-Ereignis und gibt das Ergebnis an `OUT` aus.

Anwendungsszenarien

- Logische Steuerungen in industriellen Automatisierungssystemen.
- Bitweise Maskenoperationen in Datenverarbeitungsanwendungen.
- Kombination mit anderen logischen Blöcken für komplexe Steuerungslogik.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_OR**: Führt eine bitweise ODER-Verknüpfung durch.
- **F_XOR**: Führt eine bitweise exklusive ODER-Verknüpfung durch.
- **F_NOT**: Führt eine bitweise Negation durch.

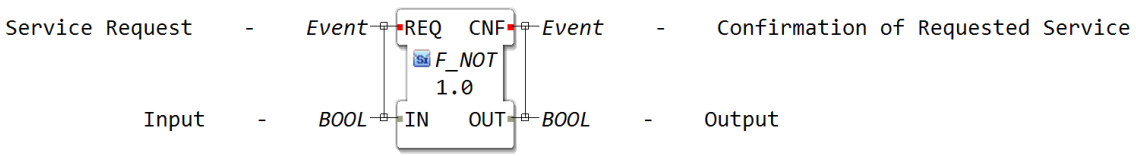
🛠️ Zugehörige Übungen

- Uebung_002a2
- Uebung_002a2_AX

Fazit

Der `F_AND`-Funktionsblock ist ein grundlegender Baustein für bitweise logische Operationen in IEC 61131-3-konformen Steuerungssystemen. Seine einfache und standardisierte Schnittstelle macht ihn zuverlässig und vielseitig einsetzbar in verschiedenen Automatisierungsanwendungen.

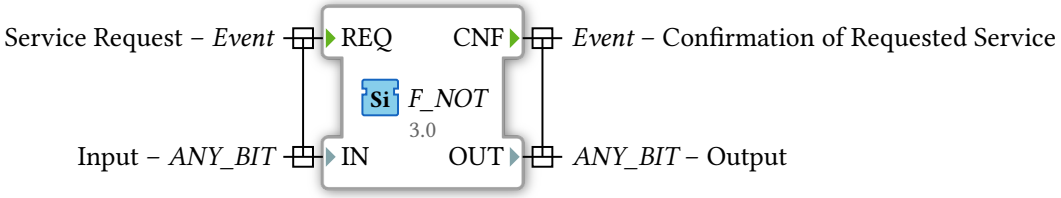
F_NOT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der F_NOT Funktionsblock implementiert eine bitweise logische NOT-Operation gemäß dem IEC 61131-3 Standard. Er führt eine bitweise Invertierung des Eingangswertes durch und gibt das Ergebnis am Ausgang aus.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Service Request - Löst die Berechnung der NOT-Operation aus

Ereignis-Ausgänge

- **CNF**: Confirmation of Requested Service - Signalisiert die Beendigung der Berechnung

Daten-Eingänge

- **IN**: Eingangswert vom Typ ANY_BIT - Der zu invertierende Wert

Daten-Ausgänge

- **OUT**: Ausgangswert vom Typ ANY_BIT - Das invertierte Ergebnis

Adapter

Keine Adapter vorhanden

Funktionsweise

Bei Empfang eines REQ-Ereignisses führt der F_NOT-Block eine bitweise logische NOT-Operation auf den Eingangswert IN aus. Jedes Bit des Eingangswertes wird invertiert (0 wird zu 1, 1 wird zu 0). Das Ergebnis wird am Ausgang OUT ausgegeben und durch das CNF-Ereignis bestätigt.

Technische Besonderheiten

- Unterstützt den ANY_BIT-Datentyp, was die Verwendung mit verschiedenen Bit-Datentypen (BOOL, BYTE, WORD, DWORD, etc.) ermöglicht
- Echtzeitfähige Ausführung
- Standardkonforme Implementierung nach IEC 61131-3

Zustandsübergang

1. Wartezustand auf REQ-Ereignis
2. Bei REQ: Ausführung der NOT-Operation
3. Ausgabe des Ergebnisses an OUT
4. Senden des CNF-Ereignisses
5. Rückkehr in Wartezustand

Anwendungsszenarien

- Bitmanipulation in Steuerungsprogrammen
- Invertierung von Statusbits
- Logische Verneinung von Bedingungen
- Signalumkehr in digitalen Schaltungen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen logischen Operationen wie F_AND oder F_OR führt F_NOT ausschließlich eine unäre Operation (ein Operand) durch, während andere logische Bausteine typischerweise binäre Operationen (zwei Operanden) implementieren.

🛠️ Zugehörige Übungen
- LinksRechts_T_FF_Event

Fazit

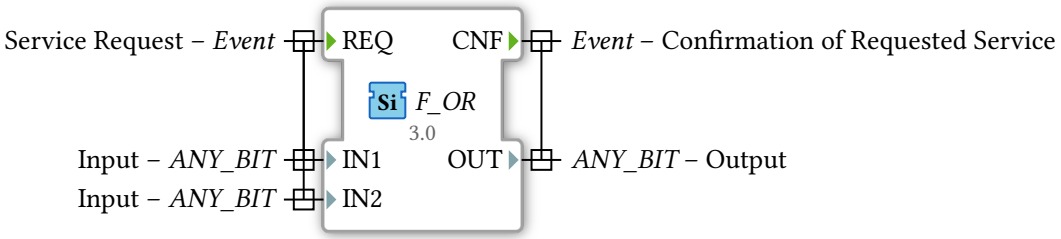
Der F_NOT Funktionsblock bietet eine standardkonforme, effiziente Lösung für bitweise Invertierungsoperationen in IEC 61131-3 kompatiblen Steuerungssystemen. Seine einfache Schnittstelle und universelle ANY_BIT-Typunterstützung machen ihn zu einem vielseitig einsetzbaren Baustein für verschiedene Anwendungsfälle.

F_OR

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_OR` implementiert eine bitweise ODER-Verknüpfung gemäß dem IEC 61131-3 Standard. Er führt eine logische ODER-Operation auf zwei Eingangsvariablen vom Typ `ANY_BIT` durch und gibt das Ergebnis auf dem Ausgang aus.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung der ODER-Operation. Bei diesem Ereignis werden die Eingänge `IN1` und `IN2` ausgewertet.

Ereignis-Ausgänge

- **CNF**: Signalisiert die erfolgreiche Beendigung der Operation. Dieses Ereignis wird zusammen mit dem Ergebnis `OUT` ausgegeben.

Daten-Eingänge

- **IN1**: Erster Eingang für die bitweise ODER-Verknüpfung (Typ: `ANY_BIT`).
- **IN2**: Zweiter Eingang für die bitweise ODER-Verknüpfung (Typ: `ANY_BIT`).

Daten-Ausgänge

- **OUT**: Ergebnis der bitweisen ODER-Verknüpfung (Typ: `ANY_BIT`).

Adapter

Der Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Auslösung des `REQ`-Ereignisses führt der `F_OR`-Block eine bitweise ODER-Verknüpfung der Werte an `IN1` und `IN2` durch. Das Ergebnis wird an `OUT` ausgegeben und das `CNF`-Ereignis signalisiert die erfolgreiche Berechnung.

Technische Besonderheiten

- **Typflexibilität**: Unterstützt alle `ANY_BIT`-Datentypen (z.B. BOOL, BYTE, WORD, DWORD, LWORD).
- **Einfache Implementierung**: Keine Zustandsmaschine erforderlich, da es sich um eine statische Operation handelt.

Zustandsübersicht

Der Funktionsblock besitzt keine interne Zustandsmaschine, da die Operation direkt bei Auftreten des `REQ`-Ereignisses durchgeführt wird.

Anwendungsszenarien

- Bitweise Verknüpfung in Steuerungsalgorithmen
- Maskenoperationen in der Signalverarbeitung
- Logische Kombination von digitalen Signalen

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_AND**: Führt eine bitweise UND-Verknüpfung durch
- **F_XOR**: Führt eine bitweise exklusive ODER-Verknüpfung durch
- **F_NOT**: Führt eine bitweise Negation durch

Fazit

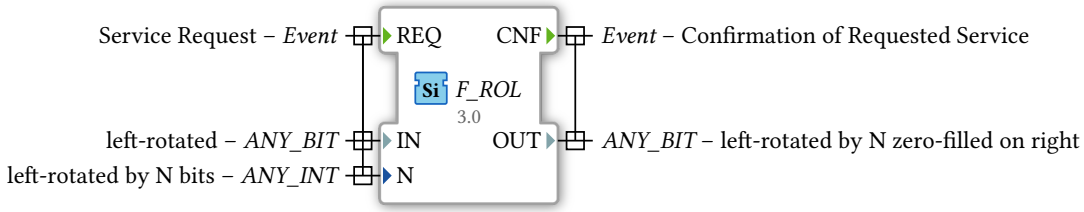
Der `F_OR`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, bitweise ODER-Operationen in IEC 61131-3-konformen Steuerungssystemen durchzuführen. Seine Typflexibilität und direkte Funktionsweise machen ihn zu einem universell einsetzbaren Baustein für logische Verknüpfungen.

F_ROL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_ROL` (Rotate Left) führt eine bitweise Linksrotation durch. Er rotiert den Eingabewert um eine angegebene Anzahl von Bits nach links. Dabei werden die Bits, die auf der linken Seite herausrotiert werden, auf der rechten Seite wieder eingefügt. Dies ist eine zyklische Operation, bei der keine Bits verloren gehen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Dienstaufforderung, die die Ausführung der Linksrotation auslöst. Wird mit den Daten-Eingängen `IN` und `N` verknüpft.

Ereignis-Ausgänge

- **CNF**: Bestätigung der durchgeführten Dienstleistung. Wird mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingabewert, der rotiert werden soll. Der Datentyp ist `ANY_BIT` (beliebiger Bit-Typ).
- **N**: Die Anzahl der Bits, um die der Eingabewert rotiert werden soll. Der Datentyp ist `ANY_INT` (beliebiger Integer-Typ).

Daten-Ausgänge

- **OUT**: Das Ergebnis der Linksrotation. Der Datentyp ist `ANY_BIT` (derselbe wie der Eingabewert).

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock `F_ROL` führt eine bitweise Linksrotation des Eingabewerts `IN` um `N` Bits durch. Dabei werden die Bits, die auf der linken Seite aus dem Wertebereich herausfallen, auf der rechten Seite wieder eingefügt. Es handelt sich um eine echte Rotation, bei der keine Stellen mit Nullen aufgefüllt werden.

Technische Besonderheiten

- Unterstützt beliebige Bit-Typen (`ANY_BIT`) für den Eingabewert `IN` und den Ausgabewert `OUT`.

- Die Anzahl der Rotationen `N` kann beliebige Integer-Werte annehmen (`ANY_INT`).
- Die Operation ist zyklisch, d.h., wenn `N` größer als die Bitlänge von `IN` ist, wird die Rotation modulo der Bitlänge durchgeführt.
- Im Gegensatz zu einer Verschiebung (Shift) gehen bei der Rotation keine Bits verloren.

Zustandsübersicht

1. **Initialzustand**: Wartet auf das Ereignis `REQ`.
2. **Ausführungszustand**: Bei Empfang von `REQ` wird die Linksrotation durchgeführt und das Ergebnis über `OUT` ausgegeben. Das Ereignis `CNF` wird ausgelöst.
3. **Rückkehr zum Initialzustand**: Nach Auslösen von `CNF` kehrt der Baustein in den Initialzustand zurück.

Anwendungsszenarien

- Bitweise Manipulation von Daten in Steuerungsanwendungen.
- Kryptographische Operationen, die bitweise Rotationen erfordern.
- Signalverarbeitung, bei denen Datenbits zyklisch verschoben werden müssen.
- Prüfsummenberechnungen und Datenkodierungen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_SHL (Shift Left)**: Verschiebt Bits nach links und füllt freiwerdende Stellen rechts mit Nullen auf. Dies ist keine zyklische Operation.
- **F_ROR (Rotate Right)**: Führt eine Rechtsrotation durch, bei der Bits von rechts nach links rotiert werden.
- **F_ROL vs. F_SHL**: Während F_SHL eine Verschiebung mit Nullauffüllung durchführt, führt F_ROL eine zyklische Rotation ohne Nullauffüllung durch.

Fazit

Der Funktionsblock `F_ROL` ist ein effizientes Werkzeug für bitweise Linksrotationen in Steuerungsanwendungen. Seine Flexibilität bei der Unterstützung verschiedener Bit- und Integer-Typen macht ihn vielseitig einsetzbar. Die zyklische Natur der Rotation stellt sicher, dass keine Datenbits verloren gehen, was ihn besonders für Anwendungen mit strengen Datenintegritätsanforderungen geeignet macht. Die klare Unterscheidung zwischen Rotation (zyklisch) und Verschiebung (mit Nullauffüllung) ist für die korrekte Anwendung entscheidend.

Änderungen im Überblick:

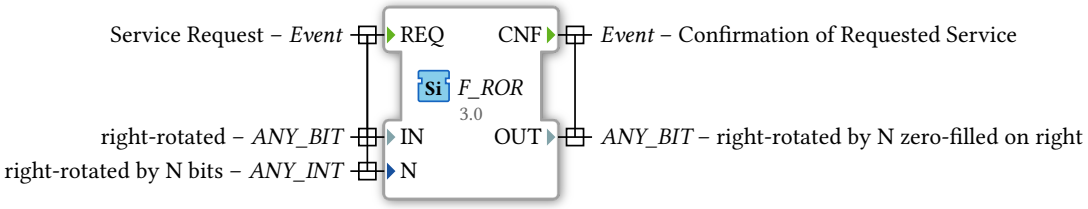
- Korrektur der Beschreibung: Bei ROL werden die herausrotierten Bits rechts wieder eingefügt, nicht mit Nullen aufgefüllt
- Klarstellung, dass es sich um eine echte zyklische Rotation handelt
- Verbesserte Abgrenzung zwischen Rotation (ROL/ROR) und Verschiebung (SHL/SHR)
- Konsistente Formulierung im gesamten Dokument

F_ROR

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_ROR` (Rotate Right) führt eine bitweise Rechtsrotation durch. Er gehört zur Kategorie der standardisierten Bitverschiebungsfunktionen gemäß IEC 61131-3. Der Block rotiert die Eingabedaten um eine angegebene Anzahl von Bits nach rechts und füllt die frei werdenden Bits mit den herausrotierenden Bits auf.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Service-Anfrage, die die Operation auslöst. Wird mit den Daten-Eingängen `IN` und `N` verknüpft.

Ereignis-Ausgänge

- `CNF`: Bestätigung der durchgeführten Operation. Wird mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- `IN`: Der Eingabewert, der rotiert werden soll (Typ: `ANY_BIT`).
- `N`: Die Anzahl der Bits, um die der Eingabewert nach rechts rotiert werden soll (Typ: `ANY_INT`).

Daten-Ausgänge

- `OUT`: Das Ergebnis der Rechtsrotation (Typ: `ANY_BIT`).

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt eine bitweise Rechtsrotation des Eingabewertes `IN` um `N` Bits durch. Dabei werden die herausrotierenden Bits auf der linken Seite wieder eingefügt. Dies unterscheidet sich von einer einfachen Rechtsverschiebung, bei der die frei werdenden Bits mit Nullen aufgefüllt werden.

Technische Besonderheiten

- Unterstützt alle `ANY_BIT`-Datentypen für den Eingabewert `IN` und den Ausgabewert `OUT`.
- Der Parameter `N` kann jeden `ANY_INT`-Datentyp annehmen.
- Die Operation ist unabhängig von der Wortbreite des Eingabewertes.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Operation wird bei jedem `REQ`-Ereignis neu ausgeführt.

Anwendungsszenarien

- Kryptografische Operationen, die bitweise Rotationen erfordern.
- Signalverarbeitung, bei denen Daten bitweise manipuliert werden müssen.
- Low-Level-Hardware-Steuerungen, die bitweise Operationen benötigen.

⚖️ Vergleich mit ähnlichen Bausteinen

- `F_SHR` (Shift Right): Führt eine Rechtsverschiebung durch, füllt aber die frei werdenden Bits mit Nullen auf.
- `F_ROL` (Rotate Left): Führt eine Linksrotation durch, ähnlich wie `F_ROR`, aber in die entgegengesetzte Richtung.

Fazit

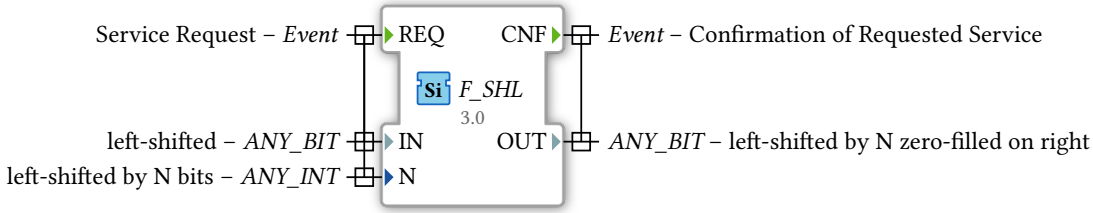
Der `F_ROR`-Funktionsblock ist ein nützliches Werkzeug für Anwendungen, die bitweise Rotationen erfordern. Seine einfache Schnittstelle und die Unterstützung verschiedener Datentypen machen ihn flexibel einsetzbar. Im Gegensatz zu einfachen Verschiebungsoperationen behält er alle Bits des Eingabewertes bei, was ihn für bestimmte Anwendungen unverzichtbar macht.

F_SHL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_SHL` führt eine Linksverschiebung (Shift Left) auf einen beliebigen Bit-Datentyp aus. Dabei werden die Bits des Eingabewertes um eine bestimmte Anzahl von Stellen nach links verschoben und die freiwerdenden Stellen rechts mit Nullen aufgefüllt. Dieser Baustein ist gemäß dem Standard IEC 61131-3 klassifiziert und eignet sich für bitweise Operationen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Dienst-Anforderung. Löst die Linksverschiebung aus. Verknüpft mit den Daten-Eingängen `IN` und `N`.

Ereignis-Ausgänge

- **CNF**: Bestätigung der angeforderten Dienstleistung. Signalisiert die Beendigung der Operation. Verknüpft mit dem Daten-Ausgang `OUT`.

Daten-Eingänge

- **IN**: Der Eingabewert, der linksverschoben werden soll. Der Datentyp ist `ANY_BIT` (beliebiger Bit-Datentyp).
- **N**: Die Anzahl der Stellen, um die verschoben werden soll. Der Datentyp ist `ANY_INT` (beliebiger Integer-Datentyp).

Daten-Ausgänge

- **OUT**: Das Ergebnis der Linksverschiebung. Der Datentyp entspricht dem des Eingabewertes `IN`. Die freiwerdenden Stellen rechts werden mit Nullen aufgefüllt.

Adapter

Es sind keine Adapter definiert.

Funktionsweise

Der Funktionsblock `F_SHL` führt eine bitweise Linksverschiebung des Eingabewertes `IN` um `N` Stellen durch. Dabei werden die Bits nach links verschoben und die neu entstehenden Stellen auf der rechten Seite mit Nullen aufgefüllt. Die Operation wird durch das Ereignis `REQ` ausgelöst und durch `CNF` bestätigt.

Technische Besonderheiten

- Unterstützt beliebige Bit-Datentypen (`ANY_BIT`) für den Eingabewert `IN`.
- Die Verschiebungsanzahl `N` kann von jedem Integer-Datentyp (`ANY_INT`) sein.
- Die Operation ist gemäß IEC 61131-3 standardisiert.

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf das Ereignis `REQ`.
2. **Ausführungszustand**: Führt die Linksverschiebung durch, wenn `REQ` empfangen wird.
3. **Bestätigungszustand**: Sendet `CNF` nach Abschluss der Operation.

Anwendungsszenarien

- Bitmanipulation in Steuerungsanwendungen.
- Verschiebeoperationen in digitalen Signalverarbeitungsalgorithmen.
- Generelle bitweise Operationen in IEC 61131-3-konformen Steuerungssystemen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_SHR**: Führt eine Rechtsverschiebung durch, im Gegensatz zur Linksverschiebung von `F_SHL`.
- **F_ROL** und **F_ROR**: Rotieren Bits statt sie zu verschieben, wobei überlaufende Bits auf der anderen Seite wieder eingefügt werden.

🛠️ Zugehörige Übungen

- Uebung_034

Fazit

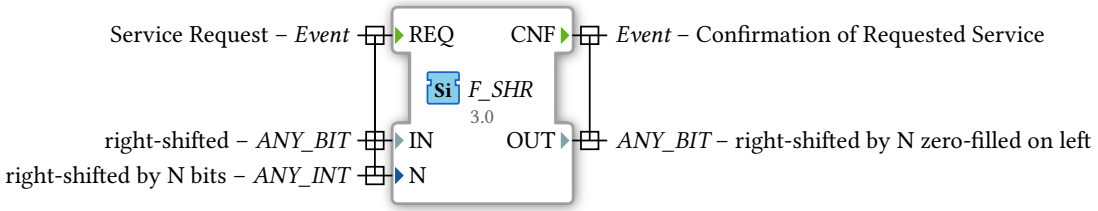
Der F_SHL-Funktionsblock ist ein effizientes Werkzeug für bitweise Linksverschiebungen in Steuerungssystemen. Seine Flexibilität bei der Unterstützung verschiedener Bit- und Integer-Datentypen macht ihn vielseitig einsetzbar. Die klare Ereignissteuerung und Standardkonformität nach IEC 61131-3 gewährleisten eine zuverlässige Integration in bestehende Systeme.

F_SHR

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock F_SHR führt eine bitweise Rechtsverschiebung (Shift Right) durch. Er ist Teil der Standard-Bit-Operatoren gemäß IEC 61131-3 und wird für logische und arithmetische Operationen in Steuerungsanwendungen eingesetzt.



Schnittstellenstruktur

Ereignis-Eingänge

- REQ: Startet die Rechtsverschiebung. Wird mit den Daten IN und N verknüpft.

Ereignis-Ausgänge

- CNF: Bestätigt die abgeschlossene Rechtsverschiebung. Liefert das Ergebnis OUT.

Daten-Eingänge

- IN (ANY_BIT): Der Eingabewert, der rechtsverschoben werden soll.
- N (ANY_INT): Die Anzahl der Bits, um die verschoben werden soll.

Daten-Ausgänge

- OUT (ANY_BIT): Das Ergebnis der Rechtsverschiebung. Die freiwerdenden Bits links werden mit Nullen aufgefüllt.

Adapter

Keine Adapter vorhanden.

Funktionsweise

1. Bei einem REQ-Ereignis wird der Wert IN um N Bits nach rechts verschoben.
2. Die freiwerdenden Bits auf der linken Seite werden mit Nullen aufgefüllt (logische Rechtsverschiebung).
3. Das Ergebnis wird über OUT ausgegeben und das CNF-Ereignis ausgelöst.

Technische Besonderheiten

- Unterstützt alle Bit-Datentypen (ANY_BIT), z. B. BOOL, BYTE, WORD, DWORD, LWORD.
- N kann jeden ganzzahligen Typ (ANY_INT) verwenden, z. B. SINT, INT, DINT, LINT, USINT, UINT, UDINT, ULINT.
- Die Rechtsverschiebung ist unabhängig vom Vorzeichen (logische Verschiebung).

Zustandsübersicht

- **Idle**: Wartet auf REQ.
- **Processing**: Führt die Rechtsverschiebung durch.
- **Completed**: Sendet CNF mit dem Ergebnis.

Anwendungsszenarien

- Bitmanipulation in Steuerungsalgorithmen.
- Datenkompression oder -extraktion.
- Schnelle Division durch Zweierpotenzen (bei vorzeichenlosen Zahlen).

⚖️ Vergleich mit ähnlichen Bausteinen

- `F_SHL`: Führt eine Linksverschiebung durch.
- `F_ROR`/`F_ROL`: Rotationsoperationen statt Verschiebungen.
- `F_SHR` ist spezifisch für logische Rechtsverschiebungen (Null-Auffüllung).

Fazit

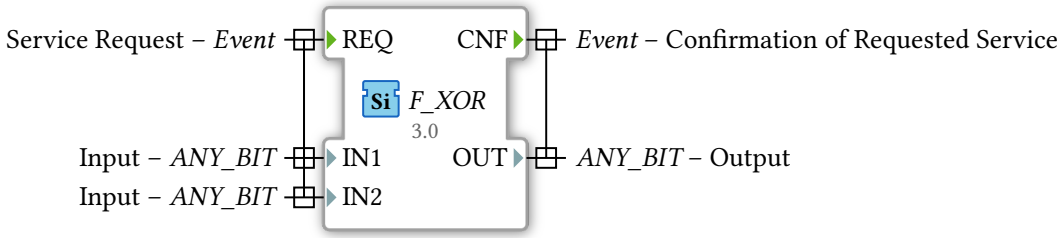
Der `F_SHR`-Funktionsblock ist ein effizientes Werkzeug für bitweise Operationen in IEC 61131-3-Umgebungen. Seine einfache Schnittstelle und Standardkonformität machen ihn ideal für Steuerungsanwendungen mit Bitmanipulationen.

F_XOR

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_XOR` führt eine bitweise XOR-Operation (exklusives ODER) auf zwei Eingabewerten durch. Er ist Teil der Standard-Bitoperationen gemäß IEC 61131-3 und eignet sich für logische Verarbeitungen in Steuerungsanwendungen.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Startet die XOR-Operation. Bei Aktivierung werden die Eingabedaten `IN1` und `IN2` verarbeitet.

Ereignis-Ausgänge

- `CNF`: Bestätigt die erfolgreiche Durchführung der XOR-Operation und liefert das Ergebnis über den Ausgang `OUT`.

Daten-Eingänge

- `IN1`: Erster Eingabewert für die XOR-Operation (Typ: `ANY_BIT`).
- `IN2`: Zweiter Eingabewert für die XOR-Operation (Typ: `ANY_BIT`).

Daten-Ausgänge

- `OUT`: Ergebnis der bitweisen XOR-Operation (Typ: `ANY_BIT`).

Adapter

Keine Adapter vorhanden.

Funktionsweise

Bei Aktivierung des Ereigniseingangs `REQ` führt der Funktionsblock eine bitweise XOR-Operation zwischen den Werten `IN1` und `IN2` durch. Das Ergebnis wird über den Datenausgang `OUT` ausgegeben und das Ereignis `CNF` wird ausgelöst, um die Fertigstellung der Operation zu signalisieren.

Technische Besonderheiten

- Unterstützt den Datentyp `ANY_BIT`, was bedeutet, dass verschiedene Bit-Datentypen (z.B. BOOL, BYTE, WORD, DWORD) verarbeitet werden können.

- Die Initialwerte der Eingänge sind leer, was eine explizite Zuweisung der Eingabewerte erfordert.

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf das Ereignis `REQ`.
2. **Verarbeitungszustand**: Führt die XOR-Operation durch, wenn `REQ` aktiviert wird.
3. **Ausgabe-Zustand**: Sendet das Ergebnis über `OUT` und löst `CNF` aus.

Anwendungsszenarien

- Logische Verarbeitung in Steuerungssystemen.
- Bitweise Manipulation von Daten in industriellen Automatisierungsprozessen.
- Implementierung von Sicherheitschecks oder Prüfsummenberechnungen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_AND / F_OR**: Führen bitweise AND- bzw. OR-Operationen durch, während `F_XOR` eine exklusive ODER-Operation bietet.
- **F_NOT**: Invertiert Bits, während `F_XOR` zwei Eingabewerte vergleicht.

Fazit

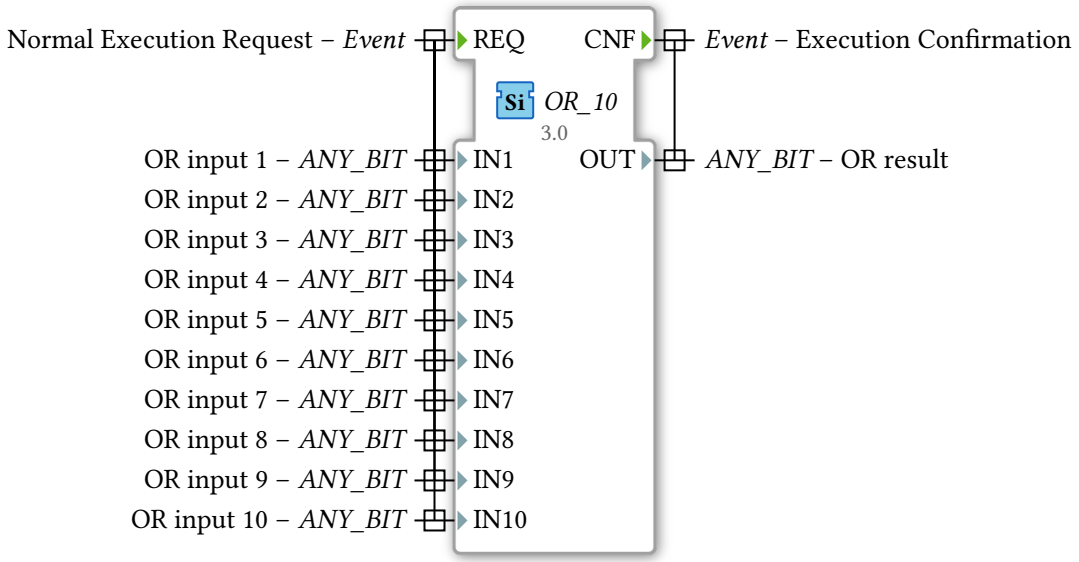
Der `F_XOR`-Funktionsblock ist ein effizientes Werkzeug für bitweise XOR-Operationen in IEC 61131-3-konformen Steuerungssystemen. Seine Flexibilität mit dem `ANY_BIT`-Datentyp macht ihn vielseitig einsetzbar in verschiedenen Automatisierungsanwendungen.

OR_10

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `OR_10` führt eine bitweise ODER-Verknüpfung auf bis zu 10 Eingangsvariablen durch. Es handelt sich um einen generischen Funktionsblock, der mit beliebigen Bit-Datentypen (ANY_BIT) arbeiten kann. Der Block ist nach dem IEC 61131-3 Standard klassifiziert und dient zur einfachen Durchführung von logischen ODER-Operationen in Steuerungsanwendungen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung des Funktionsblocks. Bei Auslösung dieses Ereignisses werden alle Eingangsvariablen (IN1 bis IN10) gelesen und die ODER-Operation durchgeführt.

Ereignis-Ausgänge

- **CNF**: Signalisiert die erfolgreiche Beendigung der ODER-Operation. Dieses Ereignis wird zusammen mit dem Ergebnis (OUT) ausgegeben.

Daten-Eingänge

- **IN1 bis IN10**: Bis zu 10 Eingangsvariablen vom Typ ANY_BIT, die die zu verknüpfenden Werte enthalten. Jede Variable kann einen beliebigen Bit-Datentyp (z.B. BOOL, BYTE, WORD, DWORD, etc.) haben.

Daten-Ausgänge

- **OUT**: Das Ergebnis der bitweisen ODER-Verknüpfung aller Eingangsvariablen. Der Datentyp entspricht dem der Eingangsvariablen.

Adapter

Es sind keine Adapter für diesen Funktionsblock definiert.

Funktionsweise

Der Funktionsblock führt eine bitweise ODER-Verknüpfung aller aktiven Eingangsvariablen durch. Das bedeutet, dass für jedes Bit der Eingangsvariablen eine logische ODER-Operation durchgeführt wird. Das Ergebnis wird an der Ausgangsvariable OUT ausgegeben. Die Operation wird durch das REQ-Ereignis gestartet und durch das CNF-Ereignis bestätigt.

Technische Besonderheiten

- **Generische Implementierung**: Der Funktionsblock kann mit beliebigen Bit-Datentypen arbeiten, was eine hohe Flexibilität ermöglicht.
- **Skalierbarkeit**: Bis zu 10 Eingangsvariablen können verarbeitet werden, was den Block für komplexere Anwendungen geeignet macht.

Zustandsübersicht

1. **Idle-Zustand**: Der Block wartet auf das REQ-Ereignis.
2. **Ausführungszustand**: Bei Empfang von REQ werden die Eingangsvariablen gelesen und die ODER-Operation durchgeführt.
3. **Bestätigungszustand**: Nach erfolgreicher Operation wird das CNF-Ereignis ausgelöst und der Block kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- Logische Verknüpfungen in Steuerungsanwendungen
- Signalverarbeitung in industriellen Automatisierungssystemen
- Kombination mehrerer binärer Signale zu einem Ergebnis

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen ODER-Blöcken mit weniger Eingängen bietet OR_10 die Möglichkeit, bis zu 10 Signale zu verknüpfen, was ihn für komplexere Anwendungen geeignet macht. Andere Blöcke sind oft auf weniger Eingänge beschränkt oder nicht generisch implementiert.

Fazit

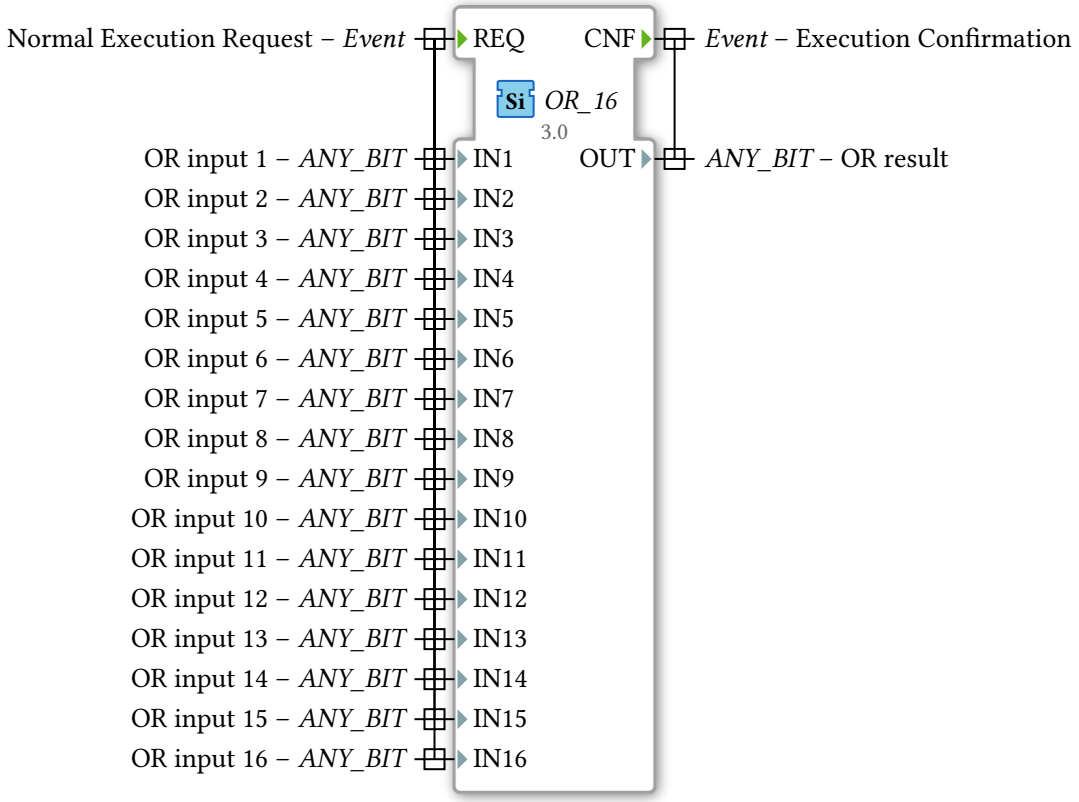
Der OR_10 Funktionsblock ist ein flexibles und leistungsfähiges Werkzeug für bitweise ODER-Operationen in IEC 61131-3-basierten Steuerungssystemen. Seine generische Natur und die Unterstützung für bis zu 10 Eingangsvariablen machen ihn besonders für anspruchsvolle Anwendungen geeignet.

OR_16

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `OR_16` führt eine bitweise logische ODER-Operation auf bis zu 16 Eingangsvariablen durch. Es handelt sich um einen generischen Funktionsblock, der mit verschiedenen Bit-Datentypen (ANY_BIT) arbeiten kann. Der Block ist Teil des Pakets `iec61131::bitwiseOperators` und entspricht dem Standard IEC 61131-3.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ` (Normal Execution Request): Löst die Berechnung aus. Wird mit allen 16 Eingangsvariablen verknüpft.

Ereignis-Ausgänge

- `CNF` (Execution Confirmation): Bestätigt die Ausführung der Operation. Wird mit dem Ergebnis `OUT` verknüpft.

Daten-Eingänge

16 Eingänge vom Typ `ANY_BIT`:

- `IN1` bis `IN16`: Die Eingangswerte für die ODER-Operation. Jeder Eingang ist kommentiert als „OR input X“ (wobei X die Nummer des Eingangs ist).

Daten-Ausgänge

- `OUT` (ANY_BIT): Das Ergebnis der bitweisen ODER-Operation aller Eingangswerte.

Adapter

Der Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Empfang des `REQ`-Ereignisses führt der Block eine bitweise ODER-Verknüpfung aller aktiven Eingangswerte (`IN1` bis `IN16`) durch. Das Ergebnis wird an `OUT` ausgegeben und das `CNF`-Ereignis wird ausgelöst.

Technische Besonderheiten

- Generische Implementierung: Arbeitet mit allen `ANY_BIT`-Datentypen (z.B. BOOL, BYTE, WORD, DWORD, LWORD)
- Unterstützt bis zu 16 Eingangsvariablen
- Kompatibel mit IEC 61131-3 Standard

Zustandsübersicht

1. Wartezustand: Auf `REQ`-Ereignis warten
2. Ausführungszustand: Bei `REQ` werden Eingänge verarbeitet und Ergebnis berechnet
3. Bestätigungszustand: Ausgabe des Ergebnisses und Trigger von `CNF`

Anwendungsszenarien

- Bitweise Logikoperationen in Steuerungsanwendungen
- Signalverarbeitung mit Mehrfachbedingungen
- Generische Logikbausteine in Bibliotheken

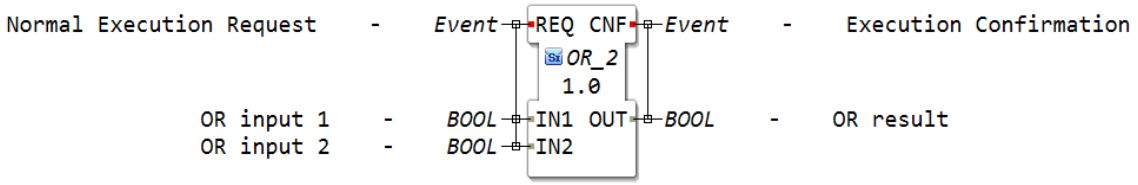
⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu einfacheren ODER-Blöcken (z.B. OR_2) bietet OR_16 deutlich mehr Eingänge
- Ähnlich zu AND_16 oder XOR_16, aber mit anderer logischer Operation
- Generische Implementierung ermöglicht flexiblere Nutzung als typspezifische Blöcke

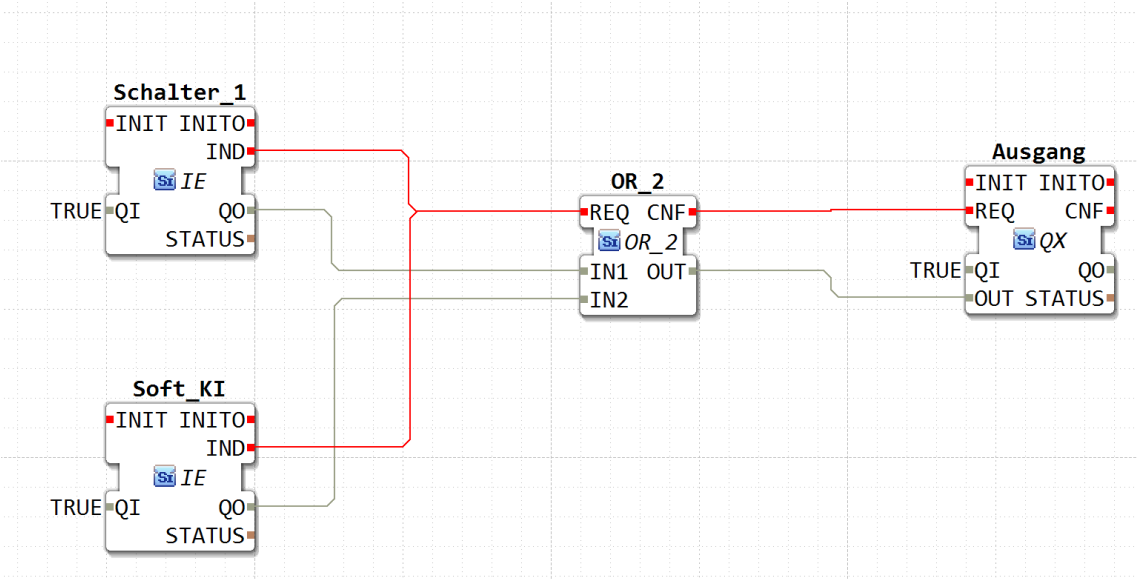
Fazit

Der OR_16 Funktionsblock bietet eine flexible und standardkonforme Lösung für bitweise ODER-Operationen mit bis zu 16 Eingängen. Seine generische Natur macht ihn vielseitig einsetzbar, während die klare Schnittstellenstruktur die Integration in verschiedene Steuerungsanwendungen erleichtert.

OR_2



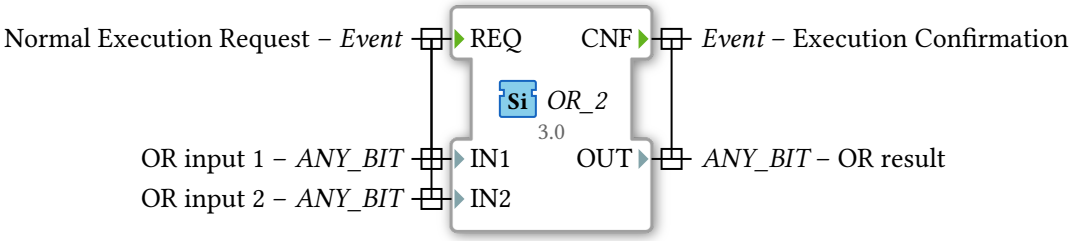
Ein häufiger Anwendungsfall ist ein Soft KI und ein AUX Button
TRUE dominiert, also sobald an einem IN ein TRUE anliegt wird der OUT auf TRUE geschalten



• – ◦ • – ◦ • – ◦ •

Einleitung

Der OR_2-Funktionsblock ist ein generischer Baustein zur Berechnung der bitweisen booleschen ODER-Verknüpfung. Er führt eine logische ODER-Operation auf zwei Eingabewerten durch und gibt das Ergebnis aus. Der Baustein ist gemäß der Norm IEC 61131-3 klassifiziert und unterstützt generische Datentypen für maximale Flexibilität.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung - löst die Berechnung der ODER-Operation aus

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung - signalisiert die Beendigung der Berechnung

Daten-Eingänge

- **IN1**: ODER-Eingang 1 (Typ: ANY_BIT) - erster Operand für die ODER-Operation
- **IN2**: ODER-Eingang 2 (Typ: ANY_BIT) - zweiter Operand für die ODER-Operation

Daten-Ausgänge

- **OUT**: ODER-Ergebnis (Typ: ANY_BIT) - Ergebnis der bitweisen ODER-Verknüpfung

Adapter

Der Baustein verwendet keine Adapter.

Funktionsweise

Bei Eintreffen des REQ-Ereignisses führt der OR_2-Baustein eine bitweise ODER-Verknüpfung zwischen den beiden Eingabewerten IN1 und IN2 durch. Das Ergebnis wird am Datenausgang OUT ausgegeben und gleichzeitig wird das CNF-Ereignis zur Bestätigung der erfolgreichen Berechnung generiert.

Die ODER-Operation wird auf Bitebene durchgeführt, wobei jedes Bit der Eingabewerte separat verarbeitet wird. Das Ergebnisbit ist 1, wenn mindestens eines der entsprechenden Eingabebits 1 ist.

Technische Besonderheiten

- **Generische Typunterstützung**: Verwendet den ANY_BIT-Datentyp, was bedeutet, dass der Baustein mit verschiedenen Bit-Datentypen (BOOL, BYTE, WORD, DWORD, LWORD) arbeiten kann
- **Standardkonforme Implementierung**: Entspricht der IEC 61131-3 Norm für bitweise boolesche Funktionen
- **Ereignisgesteuerte Ausführung**: Nur bei aktivem REQ-Ereignis wird die Berechnung durchgeführt

Zustandsübergänge

1. **Wartezustand**: Block wartet auf REQ-Ereignis
2. **Berechnungszustand**: Bei REQ-Ereignis werden IN1 und IN2 verarbeitet
3. **Ausgabezustand**: Ergebnis wird an OUT ausgegeben und CNF-Ereignis wird generiert
4. **Rückkehr**: Block kehrt in Wartezustand zurück

Anwendungsszenarien

- Bitweise Maskenoperationen in Steuerungsanwendungen
- Kombination von Statusflags in Prozesssteuerungen
- Logische Verknüpfungen in digitalen Schaltungen
- Datenmanipulation in Kommunikationsprotokollen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen booleschen Operationen wie AND oder XOR bietet OR_2 die spezifische Eigenschaft, dass das Ergebnis wahr ist, wenn mindestens einer der Operanden wahr ist. Im Gegensatz zu einfachen booleschen ODER-Bausteinen unterstützt OR_2 generische Bit-Datentypen und ermöglicht somit die Verarbeitung verschiedener Datenbreiten.

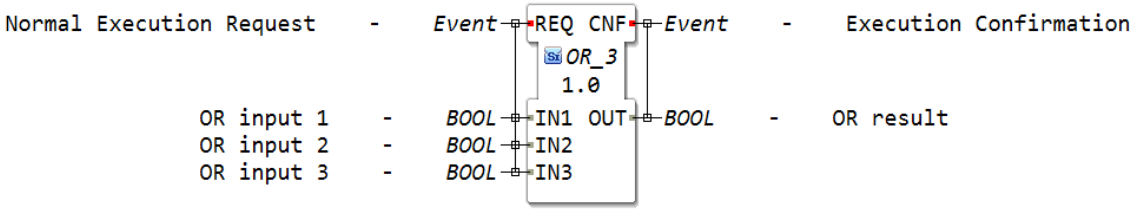
🛠️ Zugehörige Übungen

- Uebung_002a3
- Uebung_002a3_AX
- Uebung_002b2
- Uebung_002b3_AX
- Uebung_035a
- Uebung_035a2
- Uebung_035a2_AX
- Uebung_035a3
- Uebung_035a3_AX
- SoftkeyOverride_TO_QX_BG
- Uebung_089
- Uebung_160_AX
- Uebung_160b2_AX
- Uebung_160b_AX

Fazit

Der OR_2-Funktionsblock stellt eine flexible und normkonforme Lösung für bitweise ODER-Operationen in Automatisierungssystemen dar. Seine generische Natur ermöglicht den Einsatz in verschiedenen Anwendungskontexten mit unterschiedlichen Datentypen, während die ereignisgesteuerte Architektur eine effiziente Ressourcennutzung gewährleistet.

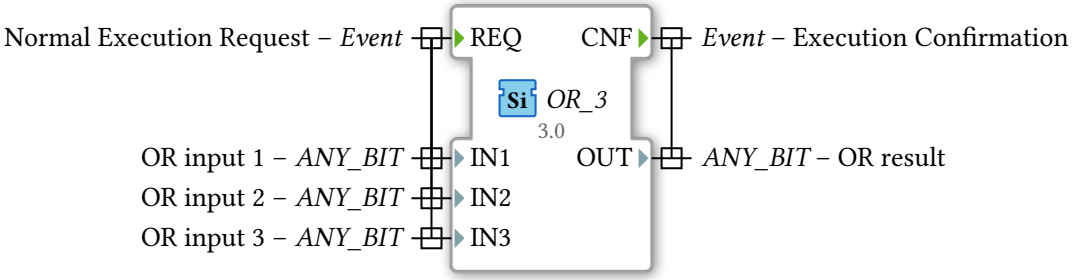
OR_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der OR_3 Funktionsblock ist ein generischer Baustein zur Berechnung der bitweisen booleschen ODER-Verknüpfung mit drei Eingängen. Er führt eine logische ODER-Operation auf Bitebene durch und unterstützt verschiedene Bit-Datentypen gemäß der IEC 61131-3 Norm.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung - löst die Berechnung der ODER-Verknüpfung aus

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung - signalisiert den Abschluss der Berechnung

Daten-Eingänge

- **IN1**: OR-Eingang 1 (ANY_BIT Datentyp)
- **IN2**: OR-Eingang 2 (ANY_BIT Datentyp)

- **IN3**: OR-Eingang 3 (ANY_BIT Datentyp)

Daten-Ausgänge

- **OUT**: OR-Ergebnis (ANY_BIT Datentyp)

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der OR_3 Funktionsblock führt bei jedem REQ-Ereignis eine bitweise ODER-Verknüpfung der drei Eingangswerte IN1, IN2 und IN3 durch. Das Ergebnis wird am Datenausgang OUT ausgegeben und gleichzeitig wird das CNF-Ereignis generiert, um den Abschluss der Operation zu signalisieren.

Die Operation wird für jeden Bit-Position separat durchgeführt: OUTi = IN1i OR IN2i OR IN3i

Technische Besonderheiten

- Unterstützt den generischen ANY_BIT Datentyp, was die Verwendung mit verschiedenen Bit-Datentypen wie BOOL, BYTE, WORD, DWORD etc. ermöglicht
- Implementiert als generischer Funktionsblock gemäß IEC 61131-3 Standard
- Gehört zum Package „iec61131::bitwiseOperators“

Zustandsübersicht

1. **Wartezustand**: Block wartet auf REQ-Ereignis
2. **Berechnungszustand**: Bei REQ-Ereignis werden die Eingänge verarbeitet
3. **Ausgabezustand**: Ergebnis wird an OUT ausgegeben und CNF-Ereignis wird generiert

Anwendungsszenarien

- Kombination mehrerer digitaler Signale in Steuerungsanwendungen
- Maskenoperationen in der Datenverarbeitung
- Parallele Verarbeitung mehrerer Bedingungen
- Signalverknüpfung in Automatisierungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen ODER-Bausteinen mit nur zwei Eingängen bietet OR_3 die Möglichkeit, drei Eingänge direkt zu verknüpfen, was die Programmstruktur vereinfacht und zusätzliche Verkettungen vermeidet. Gegenüber arithmetischen Bausteinen arbeitet OR_3 ausschließlich auf Bitebene.

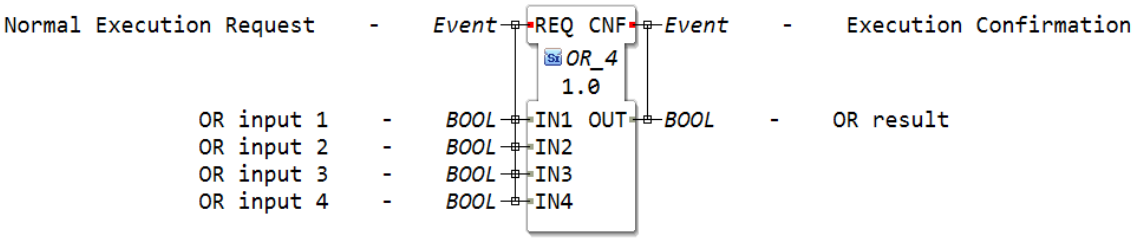
🛠️ Zugehörige Übungen

- Uebung_002a5_AX
- Uebung_002a5b_AX

Fazit

Der OR_3 Funktionsblock stellt eine effiziente und flexible Lösung für bitweise ODER-Operationen mit drei Eingängen dar. Seine generische Natur ermöglicht den Einsatz in verschiedenen Anwendungsbereichen und mit unterschiedlichen Bit-Datentypen, was ihn zu einem vielseitigen Werkzeug in der Automatisierungstechnik macht.

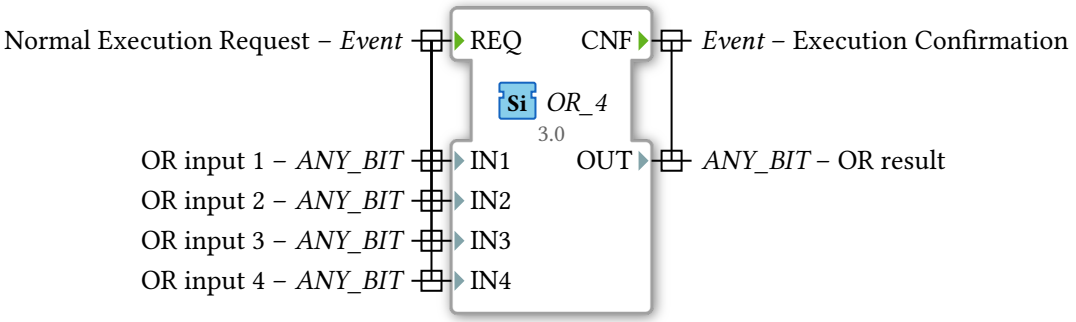
OR_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der OR_4 Funktionsblock ist ein generischer Baustein zur Berechnung der bitweisen logischen ODER-Verknüpfung mit vier Eingängen. Er führt eine bitweise ODER-Operation auf den Eingabedaten durch und gibt das Ergebnis an seinem Ausgang aus. Der Baustein ist für alle ANY_BIT Datentypen geeignet und ermöglicht somit eine flexible Verwendung mit verschiedenen Bit-Datentypen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung - löst die Berechnung der ODER-Verknüpfung aus

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung - signalisiert die Beendigung der Berechnung

Daten-Eingänge

- **IN1**: ODER-Eingang 1 (ANY_BIT)
- **IN2**: ODER-Eingang 2 (ANY_BIT)
- **IN3**: ODER-Eingang 3 (ANY_BIT)
- **IN4**: ODER-Eingang 4 (ANY_BIT)

Daten-Ausgänge

- **OUT**: ODER-Ergebnis (ANY_BIT)

Adapter

Keine Adapter vorhanden.

Funktionsweise

Bei Eintreffen des REQ-Ereignisses führt der OR_4-Baustein eine bitweise ODER-Verknüpfung aller vier Eingänge durch. Die Operation wird für jedes Bitposition separat ausgeführt: OUTi = IN1i OR IN2i OR IN3i OR IN4i. Nach Abschluss der Berechnung wird das CNF-Ereignis zusammen mit dem Ergebnis ausgegeben.

Technische Besonderheiten

- Generischer Baustein für ANY_BIT Datentypen
- Unterstützt verschiedene Bit-Datentypen (BOOL, BYTE, WORD, DWORD, LWORD)
- Bitweise Verarbeitung aller Eingänge
- Vier unabhängige Eingänge für maximale Flexibilität

Zustandsübergang

1. **Wartezustand**: Block wartet auf REQ-Ereignis
2. **Berechnungszustand**: Bei REQ-Ereignis führt Block ODER-Berechnung durch
3. **Ausgabezustand**: Sendet CNF-Ereignis mit Ergebnis und kehrt in Wartezustand zurück

Anwendungsszenarien

- Bitmasken-Operationen in Steuerungssystemen
- Kombination mehrerer Statusbits zu einem Gesamtstatus
- Parallele Verarbeitung mehrerer Bedingungen

- Signalverarbeitung in industriellen Automatisierungssystemen

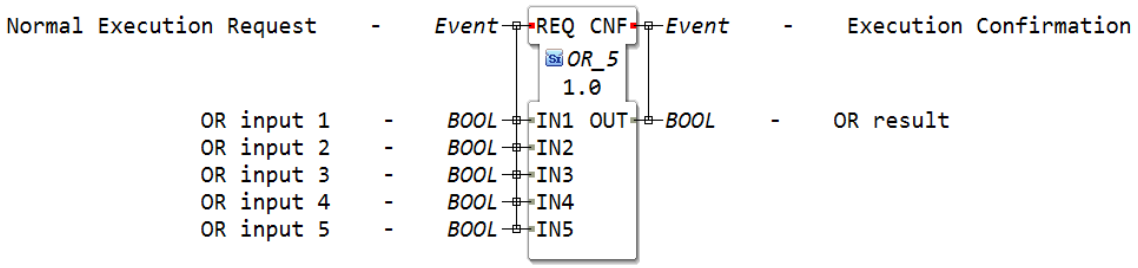
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren ODER-Bausteinen mit weniger Eingängen bietet OR_4 die Möglichkeit, vier Eingänge parallel zu verarbeiten, was die Anzahl benötigter Bausteine in komplexen Schaltungen reduziert. Gegenüber Bausteinen mit festen Datentypen bietet OR_4 durch die Verwendung von ANY_BIT eine höhere Flexibilität.

Fazit

Der OR_4 Funktionsblock ist ein vielseitiger und effizienter Baustein für bitweise ODER-Operationen mit vier Eingängen. Seine generische Natur ermöglicht den Einsatz in verschiedenen Anwendungsfällen, während die klare Ereignissteuerung eine zuverlässige Integration in komplexere Steuerungsabläufe gewährleistet.

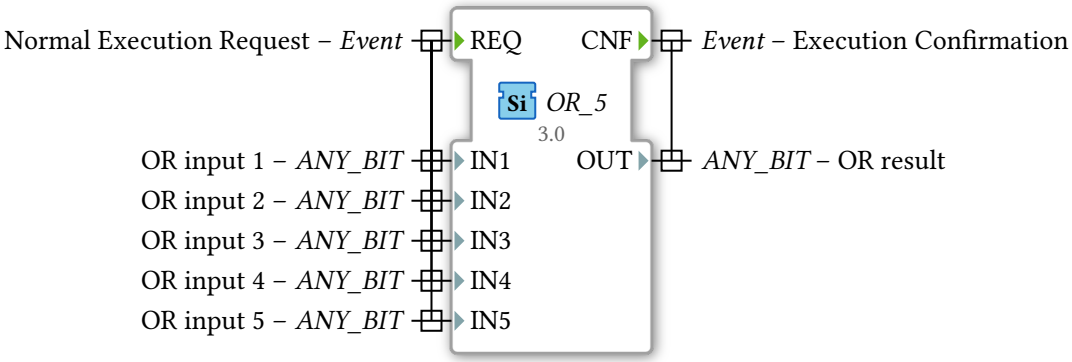
OR_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der OR_5 Funktionsblock ist ein generischer Baustein zur Durchführung einer bitweisen ODER-Verknüpfung mit fünf Eingängen. Er implementiert eine Standard-Bitfunktion gemäß IEC 61131-3 und kann mit beliebigen Bit-Datentypen arbeiten.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung - löst die Berechnung der ODER-Verknüpfung aus

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung - signalisiert den Abschluss der Berechnung

Daten-Eingänge

- **IN1**: ODER-Eingang 1 (ANY_BIT Datentyp)
- **IN2**: ODER-Eingang 2 (ANY_BIT Datentyp)
- **IN3**: ODER-Eingang 3 (ANY_BIT Datentyp)
- **IN4**: ODER-Eingang 4 (ANY_BIT Datentyp)
- **IN5**: ODER-Eingang 5 (ANY_BIT Datentyp)

Daten-Ausgänge

- **OUT**: ODER-Ergebnis (ANY_BIT Datentyp)

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der OR_5 Funktionsblock führt eine bitweise ODER-Verknüpfung aller fünf Eingangswerte durch. Bei Aktivierung des REQ-Ereignisses werden die Werte an IN1 bis IN5 verarbeitet und das Ergebnis an OUT ausgegeben. Das CNF-Ereignis signalisiert den Abschluss der Operation.

Die ODER-Operation wird nach folgender Logik durchgeführt:

```
OUT = IN1 OR IN2 OR IN3 OR IN4 OR IN5
```

Technische Besonderheiten

- **Generischer Datentyp**: Verwendet ANY_BIT, was die Verwendung mit verschiedenen Bit-Datentypen (BOOL, BYTE, WORD, DWORD, LWORD) ermöglicht
- **Fünf Eingänge**: Bietet mehr Flexibilität als Standard-ODER-Bausteine mit typischerweise zwei Eingängen
- **Ereignisgesteuert**: Arbeitet nach dem Echtzeit-Prinzip mit REQ/CNF-Signalisierung

Zustandsübergang

1. **Warte-Zustand**: Block wartet auf REQ-Ereignis
2. **Berechnungs-Zustand**: Bei REQ werden alle Eingänge verarbeitet
3. **Ausgabe-Zustand**: Ergebnis wird an OUT gesetzt und CNF wird ausgelöst
4. Rückkehr zu Warte-Zustand

Anwendungsszenarien

- Kombination mehrerer binärer Signale in Steuerungsanwendungen
- Überwachungssysteme mit mehreren Eingangsbedingungen
- Signalverarbeitung in industriellen Automatisierungssystemen
- Verwendung in Sicherheitsschaltungen mit mehreren Überwachungssensoren

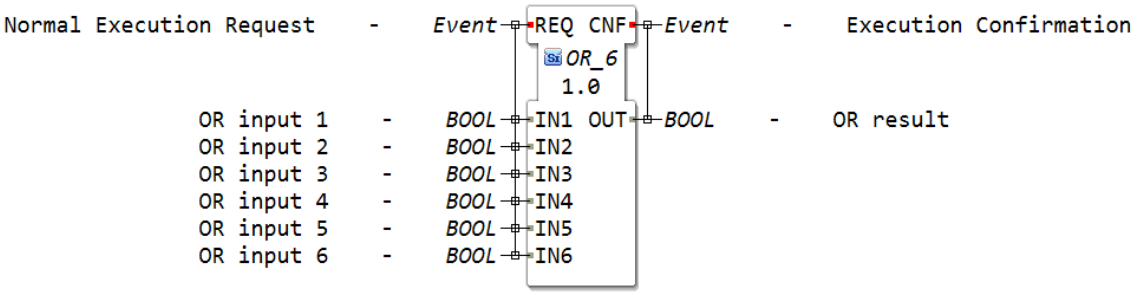
⚖️ Vergleich mit ähnlichen Bausteinen

- **Standard OR-Bausteine**: Typischerweise nur zwei Eingänge, OR_5 bietet fünf Eingänge
- **AND-Bausteine**: Führen UND-Verknüpfungen statt ODER-Verknüpfungen durch
- **XOR-Bausteine**: Führen exklusive ODER-Operationen durch

Fazit

Der OR_5 Funktionsblock ist ein vielseitiger und leistungsstarker Baustein für bitweise ODER-Operationen mit fünf Eingängen. Seine generische Natur ermöglicht den Einsatz in verschiedenen Anwendungsbereichen, während die ereignisgesteuerte Architektur eine präzise Steuerung der Ausführungszeitpunkte gewährleistet.

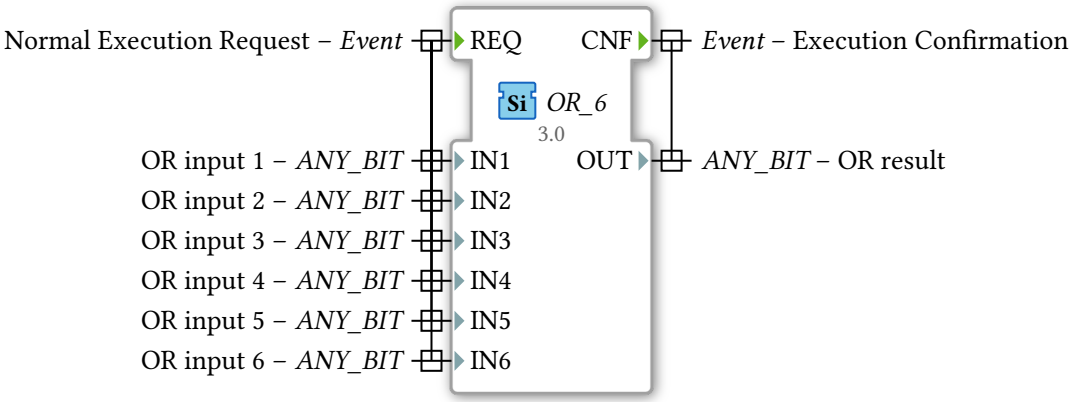
OR_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der OR_6 Funktionsblock ist ein generischer Bitweise-ODER-Baustein, der sechs Eingangswerte verarbeitet. Er führt eine bitweise logische ODER-Operation auf den sechs Eingangswerten durch und gibt das Ergebnis aus. Der Baustein ist nach der IEC 61131-3 Norm klassifiziert und kann mit beliebigen Bit-Datentypen (ANY_BIT) arbeiten.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung - löst die Berechnung der ODER-Operation aus

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung - signalisiert die Beendigung der Berechnung

Daten-Eingänge

- **IN1**: ODER-Eingang 1 (ANY_BIT)
- **IN2**: ODER-Eingang 2 (ANY_BIT)
- **IN3**: ODER-Eingang 3 (ANY_BIT)
- **IN4**: ODER-Eingang 4 (ANY_BIT)
- **IN5**: ODER-Eingang 5 (ANY_BIT)
- **IN6**: ODER-Eingang 6 (ANY_BIT)

Daten-Ausgänge

- **OUT**: ODER-Ergebnis (ANY_BIT)

Adapter

Keine Adapter vorhanden.

Funktionsweise

Bei Empfang des REQ-Ereignisses führt der OR_6-Baustein eine bitweise ODER-Operation auf allen sechs Eingangswerten durch. Die Operation wird nach folgender Logik ausgeführt:
`OUT = IN1 OR IN2 OR IN3 OR IN4 OR IN5 OR IN6`
Nach Abschluss der Berechnung wird das CNF-Ereignis zusammen mit dem Ergebniswert ausgegeben.

Technische Besonderheiten

- **Generischer Datentyp**: Verwendet ANY_BIT, was die Verwendung mit verschiedenen Bit-Datentypen (BOOL, BYTE, WORD, DWORD, LWORD) ermöglicht
- **Sechs Eingänge**: Verarbeitet bis zu sechs Eingangswerte in einer einzigen Operation
- **Bitweise Operation**: Führt die ODER-Operation auf Bitebene durch
- **Ereignisgesteuert**: Nur bei REQ-Ereignis wird die Berechnung durchgeführt

Zustandsübergänge

1. **Wartezustand**: Block wartet auf REQ-Ereignis
2. **Berechnungszustand**: Bei REQ-Ereignis werden alle Eingänge verarbeitet
3. **Ausgabezustand**: Nach Berechnung wird CNF-Ereignis mit Ergebnis ausgegeben

4. **Rückkehr**: Rückkehr in Wartezustand

Anwendungsszenarien

- **Bitmasken-Operationen**: Kombination mehrerer Bitmasken
- **Steuerungslogik**: Zusammenfassung mehrerer Bedingungen
- **Signalverarbeitung**: Verknüpfung mehrerer digitaler Signale
- **Datenfilterung**: Selektive Aktivierung von Bits aus verschiedenen Quellen

⚖️ Vergleich mit ähnlichen Bausteinen

- **OR_2**: Verarbeitet nur zwei Eingänge
- **OR_4**: Verarbeitet vier Eingänge
- **GEN_OR**: Generische Basisimplementierung, die OR_6 erweitert
- **XOR-Bausteine**: Führen exklusive ODER-Operationen durch

Fazit

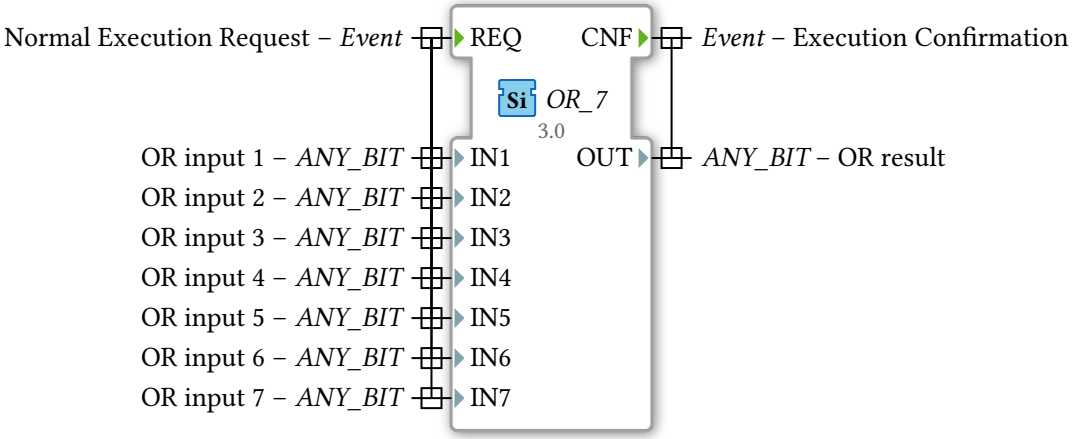
Der OR_6 Funktionsblock bietet eine effiziente Lösung für bitweise ODER-Operationen mit bis zu sechs Eingangswerten. Seine generische Natur ermöglicht maximale Flexibilität bei der Verwendung verschiedener Bit-Datentypen, während die ereignisgesteuerte Architektur eine präzise Steuerung der Berechnungszeitpunkte gewährleistet.

OR_7

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `OR_7` führt eine bitweise logische ODER-Operation auf bis zu sieben Eingangsvariablen durch. Es handelt sich um einen generischen Funktionsblock, der mit verschiedenen Bit-Datentypen (`ANY_BIT`) arbeiten kann. Der Block ist Teil der `iec61131::bitwiseOperators`-Bibliothek und implementiert eine Standard-Bitoperation gemäß IEC 61131-3.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung des Funktionsblocks. Mit diesem Ereignis werden alle Eingangsvariablen (`IN1` bis `IN7`) übergeben.

Ereignis-Ausgänge

- **CNF**: Signalisiert die erfolgreiche Ausführung der ODER-Operation und liefert das Ergebnis über die Variable `OUT`.

Daten-Eingänge

- **IN1** bis **IN7** (`ANY_BIT`): Bis zu sieben Eingangsvariablen, auf denen die bitweise ODER-Operation durchgeführt wird. Jeder Eingang kann einen beliebigen Bit-Datentyp (z.B. `BOOL`, `BYTE`, `WORD`, `DWORD`, `LWORD`) haben.

Daten-Ausgänge

- **OUT** (`ANY_BIT`): Das Ergebnis der bitweisen ODER-Operation. Der Datentyp entspricht dem der Eingangsvariablen.

Adapter

Der Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Auslösung des `REQ`-Ereignisses führt der Funktionsblock eine bitweise ODER-Operation auf den Werten der Eingangsvariablen `IN1` bis `IN7` durch. Das Ergebnis wird in der Variable `OUT` gespeichert und das `CNF`-Ereignis ausgelöst, um die erfolgreiche Berechnung zu signalisieren.

Technische Besonderheiten

- **Generische Implementierung**: Der Funktionsblock kann mit verschiedenen Bit-Datentypen (`ANY_BIT`) umgehen, was ihn flexibel einsetzbar macht.
- **Skalierbarkeit**: Unterstützt bis zu sieben Eingangsvariablen, was komplexere logische Verknüpfungen ermöglicht.

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf das `REQ`-Ereignis.
2. **Ausführungszustand**: Führt die ODER-Operation durch und berechnet das Ergebnis.
3. **Bestätigungszustand**: Löst das `CNF`-Ereignis aus und gibt das Ergebnis über `OUT` aus.

Anwendungsszenarien

- **Logische Verknüpfungen**: Kombination mehrerer binärer Signale zu einem gemeinsamen Ergebnis.
- **Steuerungstechnik**: Verwendung in Steuerungsalgorithmen, wo mehrere Bedingungen gleichzeitig erfüllt sein müssen (ODER-Verknüpfung).
- **Datenverarbeitung**: Bitweise Operationen auf Byte-, Word- oder DWord-Ebene.

⚖️ Vergleich mit ähnlichen Bausteinen

- **OR (Standard)**: Standard-ODER-Bausteine in IEC 61131-3 unterstützen typischerweise nur zwei Eingänge. `OR_7` erweitert diese Funktionalität auf bis zu sieben Eingänge.
- **GEN_OR**: Der generische ODER-Baustein, auf dem `OR_7` basiert, kann theoretisch beliebig viele Eingänge verarbeiten, ist aber weniger spezifisch in der Implementierung.

Fazit

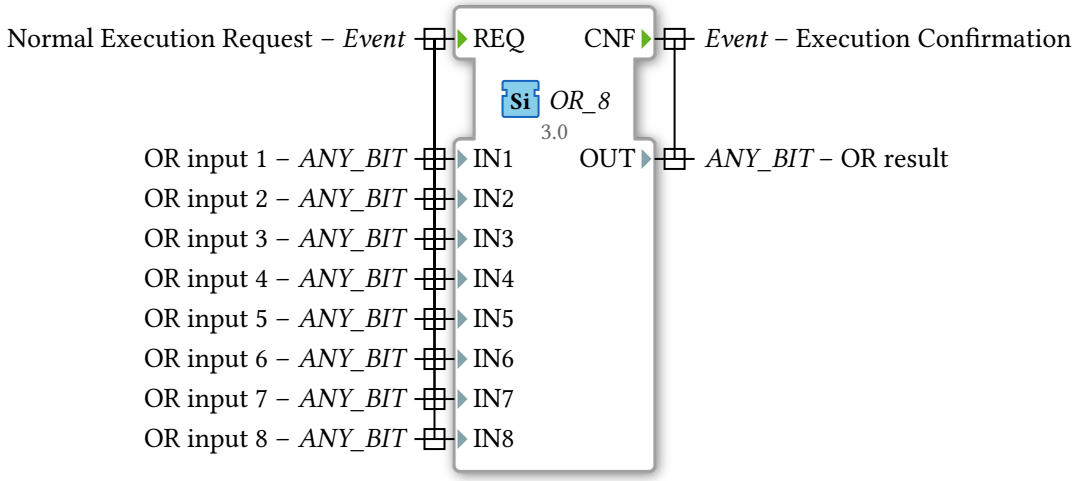
Der `OR_7`-Funktionsblock bietet eine flexible und leistungsfähige Lösung für bitweise ODER-Operationen mit bis zu sieben Eingangsvariablen. Seine generische Natur ermöglicht den Einsatz mit verschiedenen Bit-Datentypen, was ihn zu einem vielseitigen Werkzeug in der Steuerungs- und Automatisierungstechnik macht.

OR_8

• – ◦ • – ◦ • – ◦ •

Einleitung

Der OR_8 Funktionsblock ist ein generischer Baustein zur Berechnung einer bitweisen ODER-Verknüpfung. Er unterstützt bis zu 8 Eingänge und ist gemäß der Norm IEC 61131-3 klassifiziert. Der Baustein ist besonders für Anwendungen geeignet, in denen mehrere binäre Signale logisch verknüpft werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung des Funktionsblocks. Wird mit allen Daten-Eingängen verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert die erfolgreiche Ausführung des Funktionsblocks. Wird mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN1** bis **IN8**: Jeder dieser Eingänge nimmt einen beliebigen Bit-Typ (`ANY_BIT`) entgegen. Diese stellen die Operanden für die ODER-Verknüpfung dar.

Daten-Ausgänge

- **OUT**: Gibt das Ergebnis der bitweisen ODER-Verknüpfung der Eingänge `IN1` bis `IN8` zurück. Der Datentyp entspricht dem der Eingänge (`ANY_BIT`).

Adapter

Der Funktionsblock verfügt über keine Adapter.

Funktionsweise

Der OR_8 Funktionsblock führt eine bitweise ODER-Verknüpfung der Werte an den Eingängen `IN1` bis `IN8` durch. Das Ergebnis wird am Ausgang `OUT` ausgegeben, sobald das Ereignis `REQ` empfangen wird. Die Ausführung wird durch das Ereignis `CNF` bestätigt.

Technische Besonderheiten

- **Generische Implementierung**: Der Baustein ist generisch und kann mit verschiedenen Bit-Typen (`ANY_BIT`) arbeiten.
- **Flexible Eingangsanzahl**: Unterstützt bis zu 8 Eingänge, was ihn für komplexe Verknüpfungen geeignet macht.

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf das Ereignis `REQ`.
2. **Ausführungszustand**: Führt die ODER-Verknüpfung durch und setzt den Ausgang `OUT`.
3. **Bestätigungszustand**: Sendet das Ereignis `CNF` zur Bestätigung der Ausführung.

Anwendungsszenarien

- Logische Verknüpfung mehrerer Sensordaten.
- Steuerungslogik in industriellen Automatisierungssystemen.
- Kombination von binären Steuersignalen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **OR_2/OR_4**: Ähnliche Funktionsblöcke mit weniger Eingängen (2 bzw. 4). OR_8 bietet mehr Flexibilität bei komplexeren Anwendungen.
- **GEN_OR**: Generische Basisimplementierung, die durch OR_8 spezialisiert wird.

Fazit

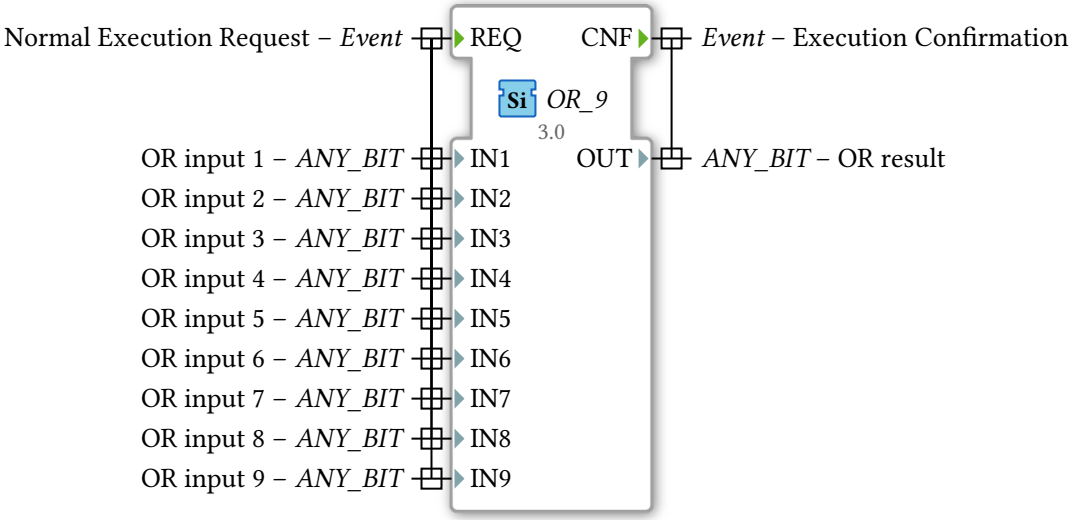
Der OR_8 Funktionsblock ist ein leistungsstarkes Werkzeug für bitweise ODER-Verknüpfungen in der industriellen Automatisierung. Seine generische Natur und die Unterstützung für bis zu 8 Eingänge machen ihn besonders vielseitig einsetzbar.

OR_9

• – ◦ • – ◦ • – ◦ •

Einleitung

Der OR_9 Funktionsblock ist ein generischer Baustein zur Berechnung einer bitweisen logischen ODER-Verknüpfung. Er unterstützt bis zu neun Eingangsvariablen und gibt das Ergebnis der ODER-Operation aus. Der Baustein ist Teil der IEC 61131-3 Standardbibliothek für bitweise boolesche Funktionen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderungseingang. Löst die Berechnung der ODER-Verknüpfung aus. Verknüpft mit allen Daten-Eingängen (IN1 bis IN9).

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Wird nach der Berechnung der ODER-Verknüpfung ausgelöst. Verknüpft mit dem Daten-Ausgang OUT.

Daten-Eingänge

- **IN1** (ANY_BIT): ODER-Eingang 1
- **IN2** (ANY_BIT): ODER-Eingang 2
- **IN3** (ANY_BIT): ODER-Eingang 3
- **IN4** (ANY_BIT): ODER-Eingang 4
- **IN5** (ANY_BIT): ODER-Eingang 5
- **IN6** (ANY_BIT): ODER-Eingang 6
- **IN7** (ANY_BIT): ODER-Eingang 7
- **IN8** (ANY_BIT): ODER-Eingang 8
- **IN9** (ANY_BIT): ODER-Eingang 9

Daten-Ausgänge

- **OUT** (ANY_BIT): Ergebnis der ODER-Verknüpfung aller Eingänge

Adapter

Der Baustein verfügt über keine Adapter.

Funktionsweise

Der OR_9 Funktionsblock führt eine bitweise logische ODER-Verknüpfung aller aktiven Eingänge (IN1 bis IN9) durch, wenn das REQ-Ereignis empfangen wird. Das Ergebnis wird am OUT-Ausgang ausgegeben und gleichzeitig das CNF-Ereignis ausgelöst.

Technische Besonderheiten

- Unterstützt den generischen Datentyp ANY_BIT für alle Eingänge und den Ausgang
- Kann mit verschiedenen Bitlängen (z.B. BYTE, WORD, DWORD) arbeiten
- Implementiert als generischer Baustein (GEN_OR)

Zustandsübersicht

1. Wartezustand: Auf REQ-Ereignis warten
2. Berechnungszustand: Führt ODER-Verknüpfung durch
3. Ausgabezustand: Setzt OUT und löst CNF aus
4. Rückkehr in Wartezustand

Anwendungsszenarien

- Bitweise Signalverarbeitung
- Kombinatorische Logikschaltungen
- Steuerungslogik mit mehreren Eingangsbedingungen
- Signalverarbeitung in industriellen Automatisierungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu einfacheren ODER-Bausteinen bietet OR_9 die Möglichkeit, bis zu neun Eingänge zu verarbeiten
- Ähnlich zu anderen bitweisen Operatoren, aber speziell für ODER-Verknüpfungen optimiert
- Generische Implementierung ermöglicht flexiblere Anwendung als typspezifische Varianten

Fazit

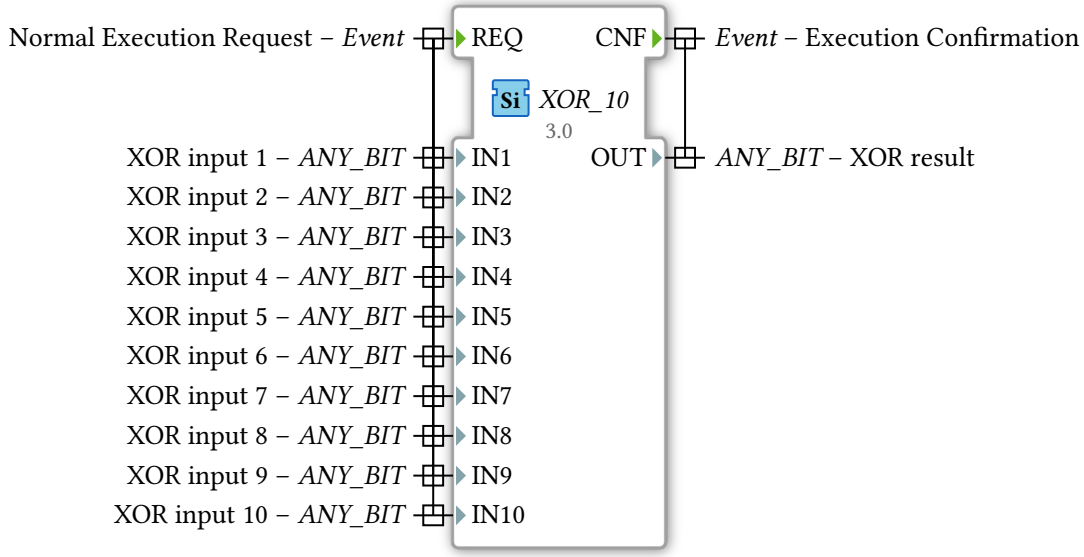
Der OR_9 Funktionsblock ist ein vielseitig einsetzbarer Baustein für bitweise ODER-Verknüpfungen in industriellen Steuerungssystemen. Seine generische Implementierung und Unterstützung für bis zu neun Eingänge machen ihn besonders flexibel für verschiedene Anwendungsfälle. Die klare Schnittstellenstruktur und Standardkonformität nach IEC 61131-3 ermöglichen eine einfache Integration in bestehende Automatisierungslösungen.

XOR_10

• – ◦ • – ◦ • – ◦ •

Einleitung

Der XOR_10 Funktionsblock ist ein generischer Baustein zur Berechnung einer bitweisen XOR-Verknüpfung mit bis zu 10 Eingängen. Er ist Teil der IEC 61131-3 Standardbibliothek für bitweise boolesche Operationen und kann mit verschiedenen Bit-Datentypen (ANY_BIT) verwendet werden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Berechnung der XOR-Verknüpfung. Verknüpft mit allen Daten-Eingängen (IN1 bis IN10).

Ereignis-Ausgänge

- **CNF**: Signalisiert die Beendigung der Berechnung und liefert das Ergebnis über den OUT-Ausgang.

Daten-Eingänge

- **IN1** bis **IN10** (ANY_BIT): Bis zu 10 Eingangswerte für die XOR-Berechnung. Jeder Eingang kann einen beliebigen Bit-Datentyp (BOOL, BYTE, WORD, DWORD, LWORD) enthalten.

Daten-Ausgänge

- **OUT** (ANY_BIT): Ergebnis der bitweisen XOR-Verknüpfung aller aktiven Eingänge.

Adapter

Dieser FB verwendet keine Adapter.

Funktionsweise

Bei Empfang des REQ-Ereignisses berechnet der FB eine bitweise XOR-Verknüpfung aller aktiven Eingänge (IN1 bis IN10). Das Ergebnis wird am OUT-Ausgang ausgegeben und das CNF-Ereignis ausgelöst.

Die XOR-Operation wird für jedes Bitposition separat durchgeführt:

- Ergebnisbit = 1, wenn eine ungerade Anzahl von Eingangsbits an dieser Position 1 ist
- Ergebnisbit = 0, wenn eine gerade Anzahl von Eingangsbits an dieser Position 1 ist

Technische Besonderheiten

- Unterstützt generische Bit-Datentypen (ANY_BIT)
- Verarbeitet bis zu 10 Eingangswerte
- Alle Eingänge müssen denselben Datentyp haben
- Nicht verwendete Eingänge werden als 0 behandelt
- Teil des `iec61131::bitwiseOperators` Pakets

Zustandsübersicht

1. **Wartezustand**: Wartet auf REQ-Ereignis
2. **Berechnungszustand**: Führt XOR-Operation durch
3. **Ausgabezustand**: Sendet Ergebnis und CNF-Ereignis
4. Rückkehr in Wartezustand

Anwendungsszenarien

- Bitweise Verschlüsselungsoperationen
- Paritätsprüfungen
- Fehlererkennung in digitalen Systemen
- Steuerungslogik mit mehreren Eingangssignalen
- Generische Logikoperationen in Automatisierungssystemen

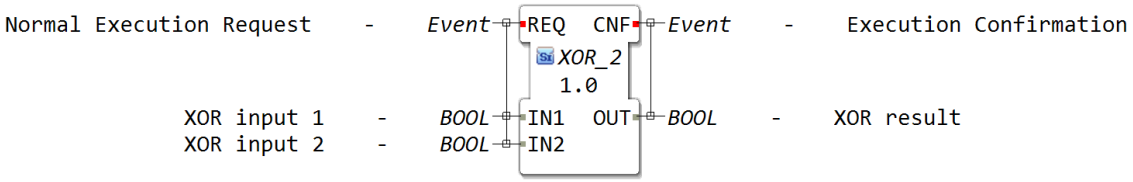
⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu einfachen XOR-Blöcken mit 2 Eingängen bietet XOR_10 mehr Flexibilität
- Ähnlich zu OR_10/AND_10 Blöcken, aber mit XOR-Logik
- Generische Implementierung im Gegensatz zu typ-spezifischen Blöcken (z.B. XOR_BOOL, XOR_WORD)

Fazit

Der XOR_10 Funktionsblock bietet eine flexible Lösung für komplexe bitweise XOR-Operationen in IEC 61131-3 basierten Steuerungssystemen. Seine generische Natur und Unterstützung für bis zu 10 Eingänge machen ihn besonders geeignet für Anwendungen, die mehr als die traditionellen zwei Eingänge erfordern.

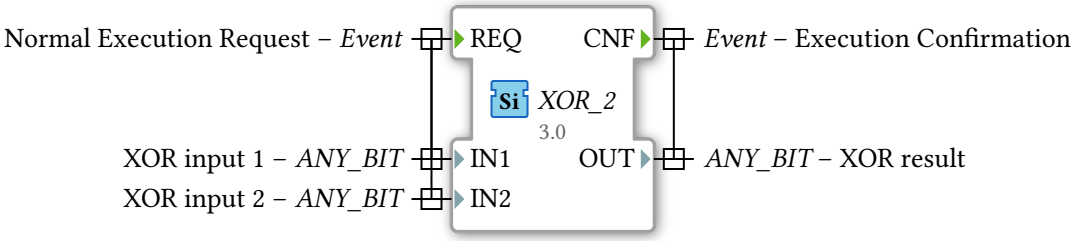
XOR_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der XOR_2-Funktionsblock ist ein generischer Baustein zur Berechnung der bitweisen booleschen XOR-Operation. Er führt eine exklusive ODER-Verknüpfung zwischen zwei Eingabewerten durch und gibt das Ergebnis aus. Der Baustein ist für alle ANY_BIT-Datentypen geeignet und ermöglicht somit eine flexible Verwendung mit verschiedenen Bit-Längen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung - löst die Berechnung der XOR-Operation aus

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung - signalisiert die Beendigung der Berechnung

Daten-Eingänge

- **IN1** (ANY_BIT): XOR-Eingang 1 - erster Operand für die XOR-Operation
- **IN2** (ANY_BIT): XOR-Eingang 2 - zweiter Operand für die XOR-Operation

Daten-Ausgänge

- **OUT** (ANY_BIT): XOR-Ergebnis - Resultat der bitweisen XOR-Operation

Adapter

Der Baustein verfügt über keine Adapter-Schnittstellen.

Funktionsweise

Bei Eintreffen des REQ-Ereignisses führt der XOR_2-Baustein eine bitweise XOR-Operation zwischen den beiden Eingabewerten IN1 und IN2 durch. Die Operation wird für jedes Bitpaar der Eingabewerte separat ausgeführt:

- Wenn beide Bits gleich sind (beide 0 oder beide 1), wird das Ergebnisbit auf 0 gesetzt
- Wenn die Bits unterschiedlich sind, wird das Ergebnisbit auf 1 gesetzt

Nach Abschluss der Berechnung wird das Ergebnis über den OUT-Ausgang ausgegeben und das CNF-Ereignis zur Bestätigung der Ausführung gesendet.

Technische Besonderheiten

- **Generische Implementierung**: Unterstützt alle ANY_BIT-Datentypen (BOOL, BYTE, WORD, DWORD, LWORD)
- **Bitweise Operation**: Führt die XOR-Operation auf Bitebene durch
- **Ereignisgesteuerte Ausführung**: Synchronisation über REQ/CNF-Ereignisse
- **Typensicherheit**: Garantiert korrekte Operation für alle unterstützten Datentypen

Zustandsübergänge

1. **Wartezustand**: Baustein wartet auf REQ-Ereignis
2. **Berechnungszustand**: Bei REQ-Ereignis führt Baustein XOR-Operation aus
3. **Ausgabezustand**: Nach Berechnung wird OUT gesetzt und CNF-Ereignis gesendet
4. **Rückkehr**: Baustein kehrt in Wartezustand zurück

Anwendungsszenarien

- **Bitmanipulation**: Umkehrung spezifischer Bits in Datenwörtern
- **Paritätsprüfung**: Erstellung von Paritätsbits für Fehlererkennung
- **Datenverschlüsselung**: Einfache kryptographische Operationen
- **Steuerungstechnik**: Logische Verknüpfungen in Automatisierungssystemen
- **Datenvergleich**: Erkennung von Unterschieden zwischen Datenwerten

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen logischen Bausteinen bietet XOR_2:

- **Flexibilität**: Unterstützung verschiedener Bit-Längen durch ANY_BIT-Typ
- **Spezifische Logik**: Exklusive ODER-Operation im Gegensatz zu AND/OR-Operationen
- **Generische Implementierung**: Wiederverwendbarkeit über verschiedene Datentypen hinweg

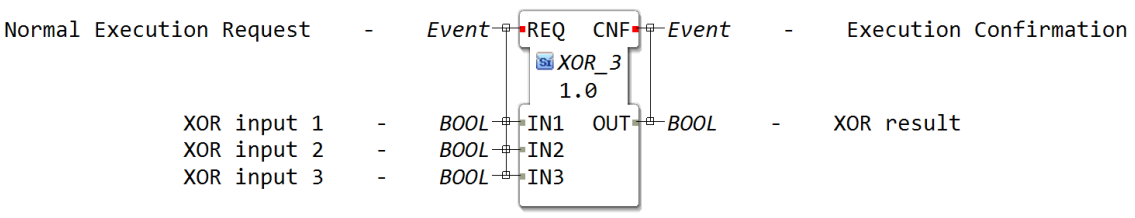
Fazit

Der XOR_2-Funktionsblock stellt eine effiziente und flexible Lösung für bitweise XOR-Operationen in 4diac-Systemen dar. Seine generische Natur ermöglicht den Einsatz in vielfältigen Anwendungsbereichen, während die ereignisgesteuerte Architektur eine zuverlässige Integration in komplexe Steuerungssysteme gewährleistet.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

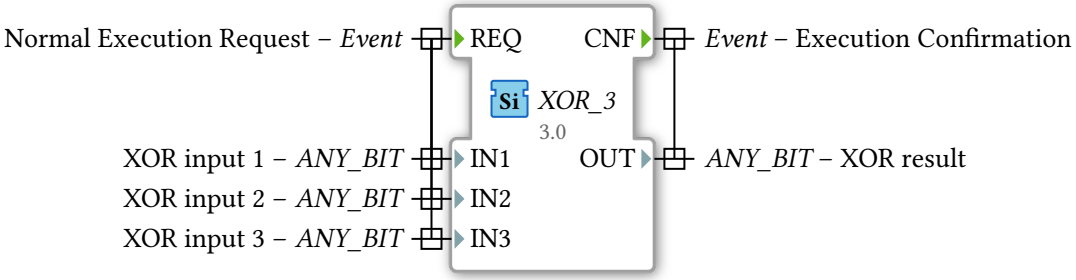
XOR_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der XOR_3 Funktionsblock ist ein generischer Baustein zur Berechnung der bitweisen XOR-Verknüpfung mit drei Eingängen. Er implementiert die logische Exklusiv-ODER-Operation auf Bitebene gemäß dem IEC 61131-3 Standard und kann mit verschiedenen Bit-Datentypen verwendet werden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung - löst die Berechnung der XOR-Operation aus

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung - signalisiert den Abschluss der Berechnung

Daten-Eingänge

- **IN1**: XOR-Eingang 1 (ANY_BIT Datentyp)
- **IN2**: XOR-Eingang 2 (ANY_BIT Datentyp)
- **IN3**: XOR-Eingang 3 (ANY_BIT Datentyp)

Daten-Ausgänge

- **OUT**: XOR-Ergebnis (ANY_BIT Datentyp)

Adapter

Der Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Eintreffen des REQ-Ereignisses führt der Baustein eine bitweise XOR-Operation auf den drei Eingangswerten IN1, IN2 und IN3 durch. Die Operation wird für jedes Bitposition separat ausgeführt und entspricht der logischen Funktion: OUT = IN1 XOR IN2 XOR IN3. Nach Abschluss der Berechnung wird das CNF-Ereignis zusammen mit dem Ergebniswert ausgegeben.

Technische Besonderheiten

- Verwendet den generischen Datentyp ANY_BIT, der mit verschiedenen Bit-Datentypen (BOOL, BYTE, WORD, DWORD, LWORD) kompatibel ist
- Implementiert als generischer Funktionsblock mit der Klasse ‚GEN_XOR‘
- Unterstützt genau drei Eingänge für die XOR-Operation
- Entspricht der IEC 61131-3 Klassifikation für standardisierte bitweise boolesche Funktionen

Zustandsübersicht

Der Baustein arbeitet zustandslos:

1. Wartezustand: Keine aktive Berechnung
2. Berechnungszustand: Bei REQ-Ereignis wird die XOR-Operation ausgeführt
3. Ausgabezustand: CNF-Ereignis mit Ergebnis wird ausgegeben

Anwendungsszenarien

- Bitweise Verschlüsselungsoperationen
- Paritätsprüfungen in Kommunikationsprotokollen
- Fehlererkennung in digitalen Systemen
- Steuerungslogik mit mehreren Bedingungen

- Datenverarbeitung in industriellen Automatisierungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-XOR-Bausteinen mit zwei Eingängen bietet XOR_3 die Möglichkeit, drei Eingänge gleichzeitig zu verarbeiten. Gegenüber AND/OR-Operationen liefert XOR ein exklusives Ergebnis, bei dem das Ausgangsbit nur dann 1 ist, wenn eine ungerade Anzahl der Eingangsbits 1 ist.

🛠️ Zugehörige Übungen

- Uebung_002a7_AX

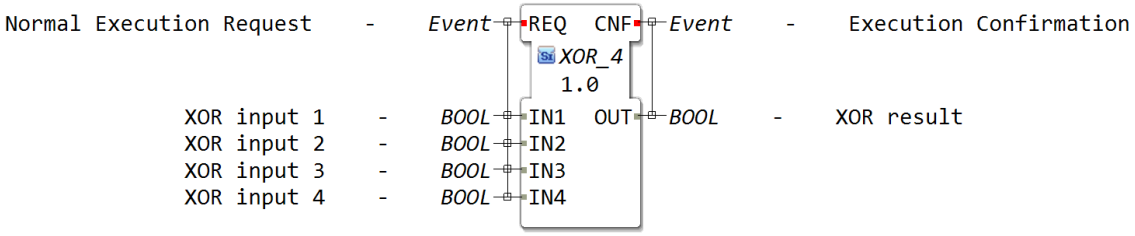
Fazit

Der XOR_3 Funktionsblock stellt eine effiziente und flexible Lösung für bitweise XOR-Operationen mit drei Eingängen in 4diac-Systemen dar. Seine generische Natur ermöglicht den Einsatz mit verschiedenen Bit-Datentypen, was ihn vielseitig in industriellen Automatisierungsanwendungen einsetzbar macht.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

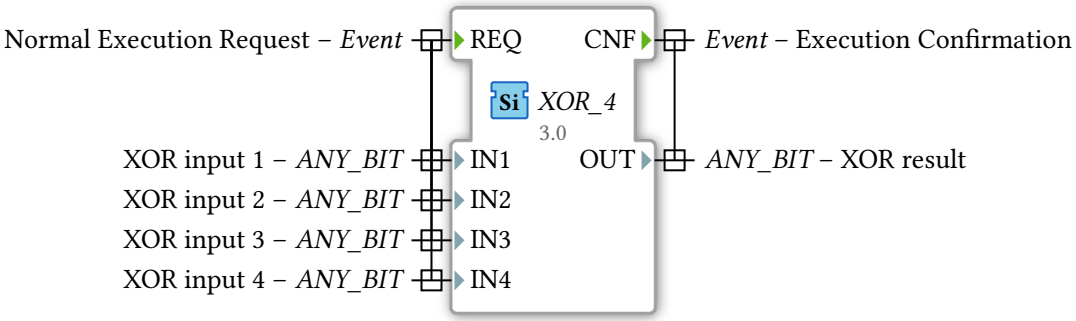
XOR_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der XOR_4 Funktionsblock ist ein generischer Baustein zur Berechnung der bitweisen XOR-Verknüpfung mit vier Eingängen. Er implementiert die exklusive ODER-Operation auf Bitebene gemäß dem IEC 61131-3 Standard und kann mit verschiedenen Bit-Datentypen arbeiten.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung - löst die Berechnung der XOR-Operation aus

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung - signalisiert den Abschluss der Berechnung

Daten-Eingänge

- **IN1**: XOR-Eingang 1 (ANY_BIT Datentyp)
- **IN2**: XOR-Eingang 2 (ANY_BIT Datentyp)
- **IN3**: XOR-Eingang 3 (ANY_BIT Datentyp)

- **IN4**: XOR-Eingang 4 (ANY_BIT Datentyp)

Daten-Ausgänge
- **OUT**: XOR-Ergebnis (ANY_BIT Datentyp)

Adapter
Der Funktionsblock verwendet keine Adapter.

Funktionsweise
Bei Eintreffen des REQ-Ereignisses führt der XOR_4-Block eine bitweise XOR-Operation auf allen vier Eingängen durch. Die Operation wird gemäß der folgenden Logik ausgeführt:
`OUT = IN1 XOR IN2 XOR IN3 XOR IN4`
Nach Abschluss der Berechnung wird das CNF-Ereignis zusammen mit dem Ergebnis an OUT ausgegeben.

Technische Besonderheiten
- **Generischer Datentyp**: Verwendet ANY_BIT, was die Verwendung mit verschiedenen Bit-Datentypen (BOOL, BYTE, WORD, DWORD, LWORD) ermöglicht
- **Vierfache XOR-Operation**: Verarbeitet vier Eingänge in einer einzigen Operation
- **Ereignisgesteuert**: Die Ausführung wird durch das REQ-Ereignis getriggert
- **Standardkonform**: Entspricht der IEC 61131-3 Norm für bitweise boolesche Funktionen

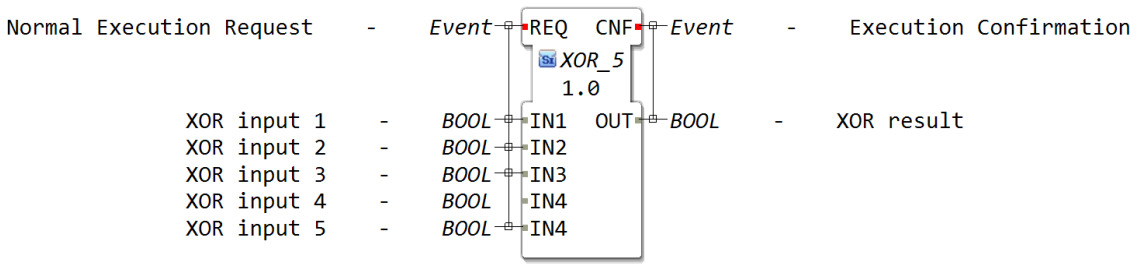
Zustandsübergänge
1. **Wartezustand**: Block wartet auf REQ-Ereignis
2. **Berechnungszustand**: Bei REQ-Ereignis werden alle Eingänge verarbeitet
3. **Ausgabezustand**: CNF-Ereignis wird mit dem berechneten Ergebnis ausgegeben
4. Rückkehr in Wartezustand

Anwendungsszenarien
- **Bitmanipulation**: Verwendung in Steuerungsanwendungen für bitweise Operationen
- **Paritätsprüfung**: Implementierung von Paritätsbits oder Fehlererkennung
- **Datenverschlüsselung**: Einfache kryptografische Operationen
- **Signalverarbeitung**: Kombination mehrerer digitaler Signale

⚖️ Vergleich mit ähnlichen Bausteinen
- **AND/OR-Bausteine**: Verarbeiten logische UND/ODER-Operationen statt exklusivem ODER
- **XOR mit weniger Eingängen**: Dieser Block bietet vier Eingänge im Vergleich zu Standard-XOR-Blöcken mit typischerweise zwei Eingängen
- **Generische vs. spezifische Blöcke**: Kann verschiedene Bit-Datentypen verarbeiten, während spezifische Blöcke auf einen Datentyp beschränkt sind

Fazit
Der XOR_4 Funktionsblock bietet eine flexible und standardkonforme Lösung für bitweise XOR-Operationen mit vier Eingängen. Seine generische Natur ermöglicht den Einsatz in verschiedenen Anwendungsbereichen, während die ereignisgesteuerte Architektur eine effiziente Integration in Steuerungssysteme gewährleistet.

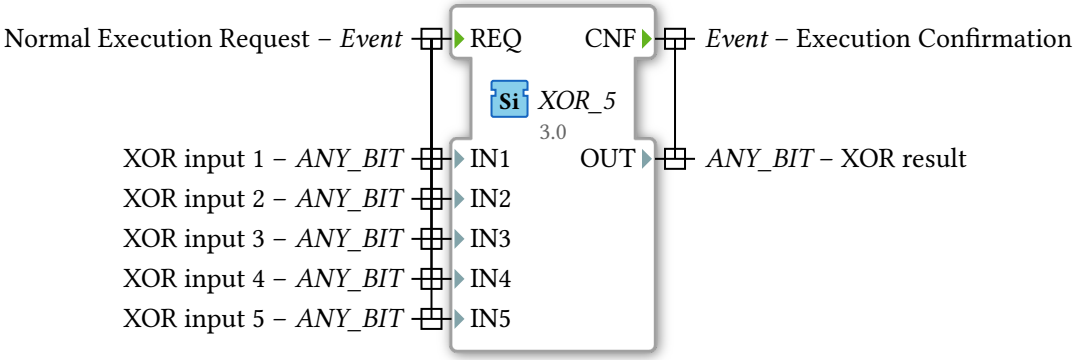
XOR_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der XOR_5 Funktionsblock ist ein generischer Baustein zur Berechnung der bitweisen booleschen XOR-Operation mit fünf Eingängen. Er implementiert die exklusive ODER-Verknüpfung für beliebige Bit-Datentypen gemäß dem IEC 61131-3 Standard.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung - löst die Berechnung der XOR-Operation aus

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung - signalisiert den Abschluss der Berechnung

Daten-Eingänge

- **IN1**: XOR Eingang 1 (ANY_BIT Datentyp)
- **IN2**: XOR Eingang 2 (ANY_BIT Datentyp)
- **IN3**: XOR Eingang 3 (ANY_BIT Datentyp)
- **IN4**: XOR Eingang 4 (ANY_BIT Datentyp)
- **IN5**: XOR Eingang 5 (ANY_BIT Datentyp)

Daten-Ausgänge

- **OUT**: XOR Ergebnis (ANY_BIT Datentyp)

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Bei Eintreffen des REQ-Ereignisses führt der Funktionsblock eine bitweise XOR-Operation über alle fünf Eingangswerte durch. Das Ergebnis wird an den OUT-Ausgang ausgegeben und gleichzeitig wird das CNF-Ereignis zur Bestätigung der abgeschlossenen Berechnung generiert.

Die XOR-Operation wird nach folgendem Prinzip durchgeführt:

- Jedes Bitposition wird einzeln verarbeitet
- Das Ergebnisbit ist 1, wenn eine ungerade Anzahl der Eingangsbits an dieser Position 1 ist
- Das Ergebnisbit ist 0, wenn eine gerade Anzahl der Eingangsbits an dieser Position 1 ist

Technische Besonderheiten

- Verwendet den generischen ANY_BIT Datentyp, der mit verschiedenen Bit-Datentypen (BOOL, BYTE, WORD, DWORD, LWORD) kompatibel ist
- Unterstützt die Verarbeitung von fünf Eingangswerten
- Implementiert als generischer Funktionsblock gemäß IEC 61131-3 Standard
- Package-Zuordnung: iec61131::bitwiseOperators

Zustandsübersicht

Der Funktionsblock besitzt einen einfachen Zustandsautomaten:

1. **Wartezustand**: Warten auf REQ-Ereignis

2. **Berechnungszustand**: Verarbeitung der XOR-Operation
3. **Ausgabezustand**: Setzen des OUT-Wertes und Auslösen des CNF-Ereignisses

Anwendungsszenarien

- Bitweise Datenverarbeitung in Steuerungssystemen
- Paritätsprüfungen und Fehlererkennung
- Kryptografische Operationen
- Datenmanipulation und -transformation
- Signalverarbeitung mit mehreren Eingangssignalen

⚖️ Vergleich mit ähnlichen Bausteinen

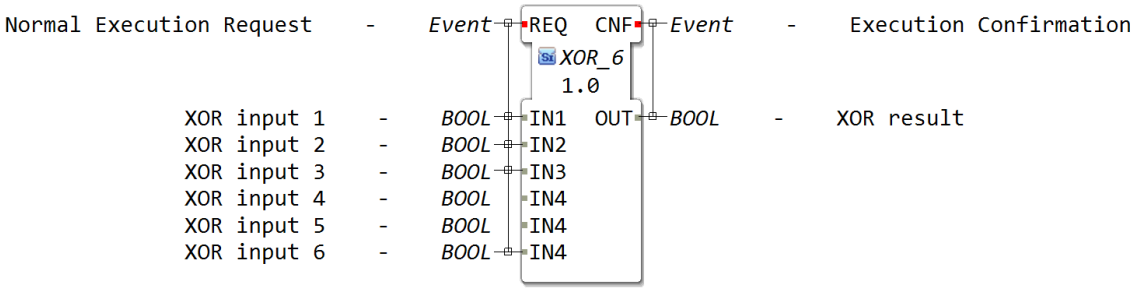
Im Vergleich zu Standard-XOR-Bausteinen mit weniger Eingängen bietet XOR_5:

- Erweiterte Funktionalität mit fünf statt zwei Eingängen
- Höhere Flexibilität bei komplexeren logischen Verknüpfungen
- Reduzierte Verkettungstiefe gegenüber seriellen XOR-Verknüpfungen

Fazit

Der XOR_5 Funktionsblock stellt eine leistungsstarke und flexible Lösung für bitweise XOR-Operationen mit fünf Eingängen dar. Seine generische Implementierung ermöglicht den Einsatz mit verschiedenen Bit-Datentypen und macht ihn zu einem vielseitigen Werkzeug für verschiedene Anwendungsbereiche in der Automatisierungstechnik.

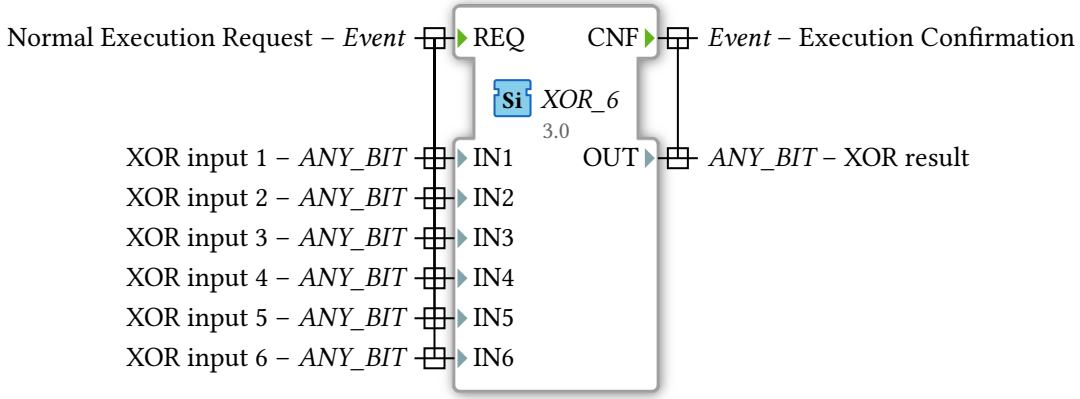
XOR_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der XOR_6 Funktionsblock ist ein generischer Baustein zur Berechnung der bitweisen XOR-Verknüpfung mit sechs Eingängen. Er implementiert die boolesche Exklusiv-ODER-Operation auf Bitebene gemäß dem IEC 61131-3 Standard und kann mit verschiedenen Bit-Datentypen arbeiten.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung - löst die Berechnung der XOR-Operation aus

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung - signalisiert den Abschluss der Berechnung

Daten-Eingänge

- **IN1**: XOR-Eingang 1 (ANY_BIT Datentyp)
- **IN2**: XOR-Eingang 2 (ANY_BIT Datentyp)
- **IN3**: XOR-Eingang 3 (ANY_BIT Datentyp)
- **IN4**: XOR-Eingang 4 (ANY_BIT Datentyp)
- **IN5**: XOR-Eingang 5 (ANY_BIT Datentyp)
- **IN6**: XOR-Eingang 6 (ANY_BIT Datentyp)

Daten-Ausgänge

- **OUT**: XOR-Ergebnis (ANY_BIT Datentyp)

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der XOR_6 Block führt eine bitweise XOR-Operation auf allen sechs Eingängen durch. Bei Aktivierung des REQ-Ereignisses werden die Werte an IN1 bis IN6 verarbeitet und das Ergebnis an OUT ausgegeben. Die Operation folgt der mathematischen Definition der XOR-Funktion, wobei das Ergebnis für jedes Bit wahr ist, wenn eine ungerade Anzahl der entsprechenden Eingangsbits wahr ist.

Die Berechnung erfolgt nach der Formel: OUT = IN1 ⊕ IN2 ⊕ IN3 ⊕ IN4 ⊕ IN5 ⊕ IN6

Technische Besonderheiten

- **Generischer Datentyp**: Verwendet ANY_BIT, was Kompatibilität mit verschiedenen Bit-Datentypen (BOOL, BYTE, WORD, DWORD, LWORD) ermöglicht
- **Sechs Eingänge**: Bietet erweiterte Funktionalität gegenüber Standard-XOR-Blöcken mit weniger Eingängen
- **Ereignisgesteuerte Ausführung**: Folgt dem 4diac-Execution Model mit REQ/CNF Ereignispaar

Zustandsübersicht

Der Funktionsblock besitzt einen einfachen Zustandsautomaten:

1. **Wartezustand**: Block wartet auf REQ-Ereignis
2. **Berechnungszustand**: Verarbeitet die Eingänge bei REQ-Aktivierung
3. **Ausgabezustand**: Sendet CNF-Ereignis mit berechnetem Ergebnis

Anwendungsszenarien

- **Paritätsprüfung**: Überprüfung der Datenintegrität durch XOR-Verknüpfung mehrerer Datenworte
- **Datenverschlüsselung**: Einfache kryptographische Operationen
- **Fehlererkennung**: Generierung von Prüfsummen in Kommunikationsprotokollen
- **Steuerungstechnik**: Logische Verknüpfungen in Automatisierungsanwendungen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-XOR-Blöcken mit zwei Eingängen bietet XOR_6:

- Erweiterte Funktionalität mit sechs direkten Eingängen
- Vermeidung von Kaskadierung mehrerer XOR-Blöcke
- Kompaktere Implementierung für komplexe XOR-Operationen

Fazit

Der XOR_6 Funktionsblock stellt eine leistungsstarke und flexible Lösung für bitweise XOR-Operationen mit mehreren Eingängen dar. Seine generische Natur ermöglicht breite Anwendbarkeit in verschiedenen Automatisierungsprojekten, während die sechsfache Eingangsfunktionalität effiziente Implementierungen komplexer logischer Verknüpfungen unterstützt.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

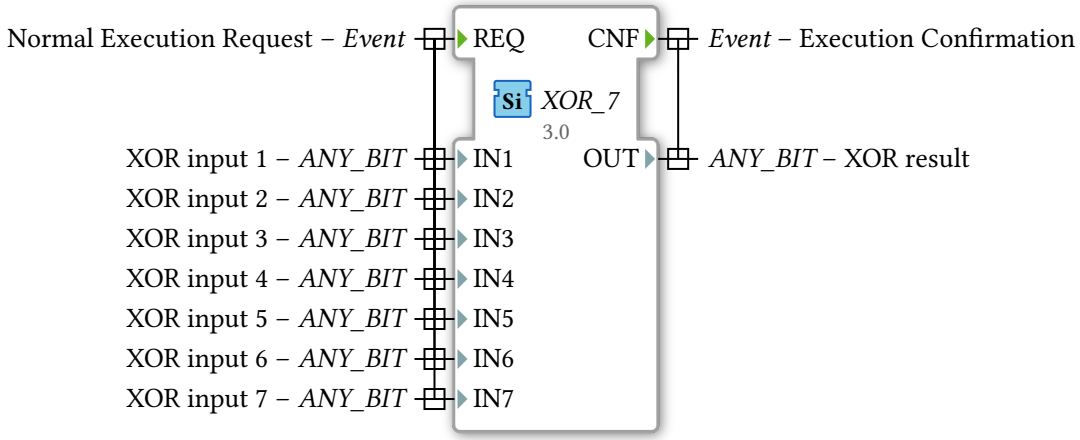
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

XOR_7

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `XOR_7` führt eine bitweise XOR-Operation auf sieben Eingabewerten durch. Es handelt sich um einen generischen Funktionsblock, der mit verschiedenen Bit-Datentypen (z.B. BOOL, BYTE, WORD, DWORD, LWORD) verwendet werden kann. Der Block ist gemäß dem IEC 61131-3 Standard klassifiziert und dient zur Verarbeitung boolescher Operationen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Normal Execution Request): Löst die Berechnung der XOR-Operation aus. Wird mit allen Daten-Eingängen (`IN1` bis `IN7`) verknüpft.

Ereignis-Ausgänge

- **CNF** (Execution Confirmation): Signalisiert die erfolgreiche Berechnung und liefert das Ergebnis über den Daten-Ausgang `OUT`.

Daten-Eingänge

- **IN1** (XOR input 1): Erster Eingabewert (Typ: `ANY_BIT`).
- **IN2** (XOR input 2): Zweiter Eingabewert (Typ: `ANY_BIT`).
- **IN3** (XOR input 3): Dritter Eingabewert (Typ: `ANY_BIT`).
- **IN4** (XOR input 4): Vierter Eingabewert (Typ: `ANY_BIT`).
- **IN5** (XOR input 5): Fünfter Eingabewert (Typ: `ANY_BIT`).
- **IN6** (XOR input 6): Sechster Eingabewert (Typ: `ANY_BIT`).
- **IN7** (XOR input 7): Siebter Eingabewert (Typ: `ANY_BIT`).

Daten-Ausgänge

- **OUT** (XOR result): Ergebnis der bitweisen XOR-Operation (Typ: `ANY_BIT`).

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt eine bitweise XOR-Operation auf den sieben Eingabewerten (`IN1` bis `IN7`) aus. Die Operation wird durch das Ereignis `REQ` ausgelöst. Das Ergebnis wird über den Ausgang `OUT` ausgegeben und durch das Ereignis `CNF` bestätigt.

Technische Besonderheiten

- **Generische Implementierung**: Unterstützt verschiedene Bit-Datentypen (`ANY_BIT`).
- **Flexibilität**: Kann mit bis zu sieben Eingabewerten arbeiten.
- **Standardkonform**: Entspricht dem IEC 61131-3 Standard.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Operation wird bei jedem `REQ`-Ereignis neu ausgeführt.

Anwendungsszenarien

- Bitweise Verschlüsselung.
- Fehlererkennung in Datenübertragungen.
- Logische Steuerungen in Automatisierungssystemen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **XOR (Standard)**: Bietet typischerweise nur zwei Eingänge, während `XOR_7` sieben Eingänge unterstützt.
- **AND/OR-Blöcke**: Führen andere logische Operationen durch, während `XOR_7` spezifisch für die XOR-Operation optimiert ist.

Fazit

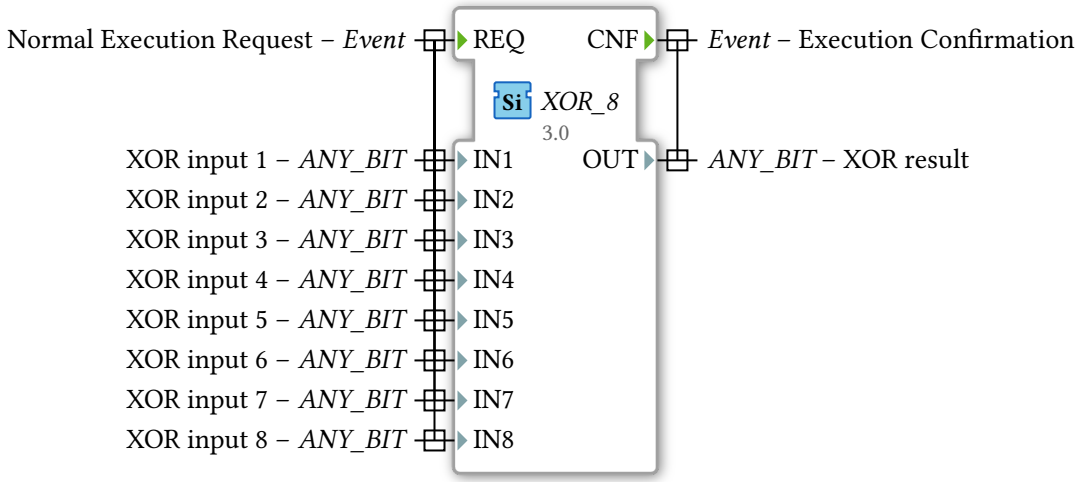
Der `XOR_7`-Funktionsblock ist ein leistungsfähiges Werkzeug für bitweise XOR-Operationen mit bis zu sieben Eingabewerten. Seine generische Natur und Standardkonformität machen ihn vielseitig einsetzbar in verschiedenen Automatisierungs- und Steuerungsszenarien.

XOR_8

• – ◦ • – ◦ • – ◦ •

Einleitung

Der XOR_8 Funktionsblock führt eine bitweise XOR-Operation auf bis zu 8 Eingangsvariablen durch. Es handelt sich um einen generischen Funktionsblock, der mit verschiedenen Bit-Datentypen (ANY_BIT) arbeiten kann. Der Block ist gemäß dem IEC 61131-3 Standard klassifiziert und bietet eine einfache Möglichkeit, XOR-Operationen in Steuerungsanwendungen durchzuführen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Normal Execution Request): Startet die Berechnung der XOR-Operation. Wird mit allen Daten-Eingängen verknüpft.

Ereignis-Ausgänge

- **CNF** (Execution Confirmation): Signalisiert den Abschluss der Berechnung. Wird mit dem Daten-Ausgang OUT verknüpft.

Daten-Eingänge

- **IN1** (ANY_BIT): XOR-Eingang 1
- **IN2** (ANY_BIT): XOR-Eingang 2

- **IN3** (ANY_BIT): XOR-Eingang 3
- **IN4** (ANY_BIT): XOR-Eingang 4
- **IN5** (ANY_BIT): XOR-Eingang 5
- **IN6** (ANY_BIT): XOR-Eingang 6
- **IN7** (ANY_BIT): XOR-Eingang 7
- **IN8** (ANY_BIT): XOR-Eingang 8

Daten-Ausgänge

- **OUT** (ANY_BIT): Ergebnis der XOR-Operation

Adapter

Der Funktionsblock verfügt über keine Adapter-Schnittstellen.

Funktionsweise

Bei Empfang des REQ-Ereignisses führt der Funktionsblock eine bitweise XOR-Operation auf allen aktiven Eingangsvariablen (IN1 bis IN8) durch. Das Ergebnis wird am Ausgang OUT ausgegeben und das CNF-Ereignis wird ausgelöst, um den Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Unterstützt den generischen Datentyp ANY_BIT, was bedeutet, dass verschiedene Bit-Datentypen (z.B. BOOL, BYTE, WORD, DWORD, LWORD) verwendet werden können.
- Die Operation wird streng bitweise durchgeführt.
- Der Funktionsblock ist Teil des Pakets „iec61131::bitwiseOperators“.

Zustandsübersicht

Der Funktionsblock hat keine komplexen Zustände. Er reagiert auf das REQ-Ereignis mit einer sofortigen Berechnung und Ausgabe des Ergebnisses.

Anwendungsszenarien

- Bitweise Prüfsummenberechnung
- Datenverschlüsselungsoperationen
- Fehlererkennung in Datenübertragungen
- Steuerungslogik mit bitweisen Operationen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren XOR-Blöcken (z.B. mit nur 2 Eingängen) bietet XOR_8 die Möglichkeit, bis zu 8 Eingänge gleichzeitig zu verarbeiten. Andere bitweise Operatoren (wie AND_8 oder OR_8) führen ähnliche Operationen mit unterschiedlichen logischen Verknüpfungen durch.

Fazit

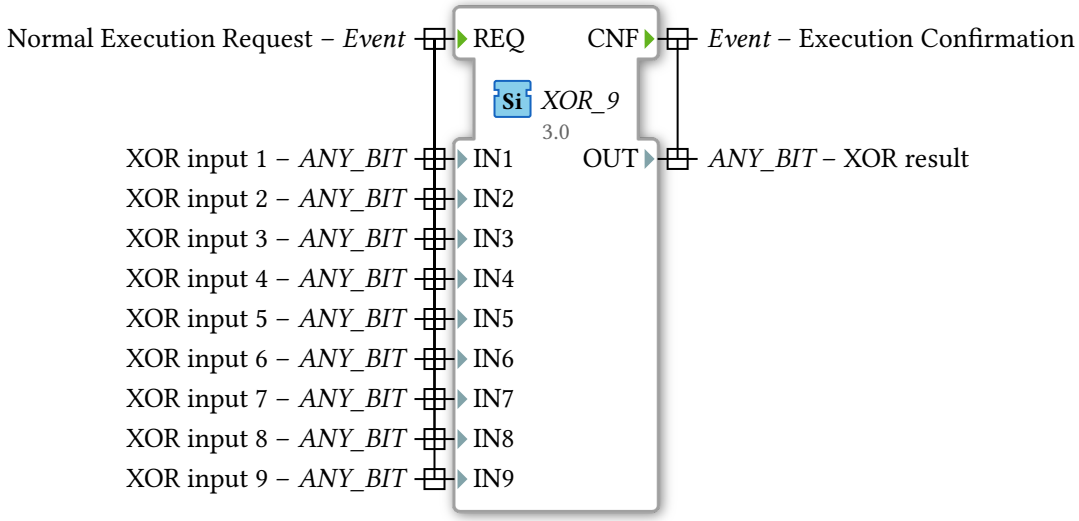
Der XOR_8 Funktionsblock bietet eine flexible und standardkonforme Lösung für bitweise XOR-Operationen in Steuerungssystemen. Seine generische Natur ermöglicht den Einsatz mit verschiedenen Bit-Datentypen, während die Unterstützung für bis zu 8 Eingänge komplexere logische Operationen ermöglicht. Die einfache Schnittstelle und deterministische Ausführung machen ihn zu einem zuverlässigen Baustein für Steuerungsanwendungen.

XOR_9

• − ◦ • − ◦ • − ◦ •

Einleitung

Der Funktionsblock `XOR_9` dient zur bitweisen Berechnung der XOR-Verknüpfung mit bis zu neun Eingängen. Er ist Teil der Standard-Bitoperationen gemäß IEC 61131-3 und ermöglicht die Verarbeitung beliebiger Bit-Datentypen (`ANY_BIT`).



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ` (Normal Execution Request): Startet die Berechnung der XOR-Verknüpfung. Löst die Auswertung aller verknüpften Daten-Eingänge aus.

Ereignis-Ausgänge

- `CNF` (Execution Confirmation): Signalisiert den Abschluss der Berechnung und liefert das Ergebnis über den Daten-Ausgang `OUT`.

Daten-Eingänge

- `IN1` bis `IN9` (XOR input 1-9): Bis zu neun Eingänge für die bitweise XOR-Verknüpfung. Jeder Eingang unterstützt den Datentyp `ANY_BIT` (z.B. BOOL, BYTE, WORD, DWORD, LWORD).

Daten-Ausgänge

- `OUT` (XOR result): Ergebnis der bitweisen XOR-Verknüpfung aller aktiven Eingänge. Der Datentyp entspricht dem der Eingänge (`ANY_BIT`).

Adapter

Der Funktionsblock verfügt über keine Adapter.

Funktionsweise

Bei Auslösung des `REQ`-Ereignisses berechnet der FB die XOR-Verknüpfung aller übergebenen Eingangswerte (`IN1` bis `IN9`). Das Ergebnis wird an `OUT` ausgegeben und das `CNF`-Ereignis signalisiert die Bereitschaft des Ergebnisses.

Beispiel (für BOOL-Eingänge): `OUT = IN1 XOR IN2 XOR ... XOR IN9`

Technische Besonderheiten

- **Generische Implementierung:** Unterstützt alle `ANY_BIT`-Datentypen durch Verwendung der generischen Klasse `GEN_XOR`.
- **Flexible Eingangsanzahl:** Nicht genutzte Eingänge werden ignoriert (wirken als neutrale Elemente).
- **Bitweise Operation:** Die Verknüpfung erfolgt für jedes Bit separat (bei mehrbitigen Datentypen).

Zustandsübersicht

1. **Idle:** Wartet auf `REQ`-Ereignis.
2. **Processing:** Berechnet XOR und setzt `OUT`.
3. **Ready:** Sendet `CNF` und kehrt nach Idle zurück.

Anwendungsszenarien

- Paritätsprüfung in Kommunikationsprotokollen

- Toggeln von Steuersignalen (z.B. Wechsel zwischen Zuständen)
- Kryptografische Grundoperationen
- Fehlererkennung in Binärdaten

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	XOR_9	Standard-XOR (2 Eingänge)
Eingangsanzahl	9	2
Datentyp	ANY_BIT	Typabhängig (z.B. BOOL)
Flexibilität	Hoch	Gering

Fazit

Der `XOR_9`-FB erweitert die klassische XOR-Funktionalität durch Unterstützung mehrerer Eingänge und generischer Datentypen. Ideal für Anwendungen, die komplexere bitweise Verknüpfungen erfordern, ohne individuelle FB-Ketten aufbauen zu müssen. Die IEC 61131-3-Konformität gewährleistet breite Kompatibilität.

2.4.1.5 Zeichenketten Funktionen

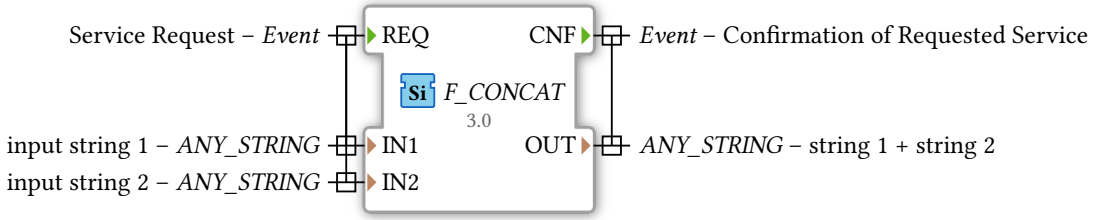
Hier finden Sie Funktionen zur Bearbeitung von Zeichenketten (Strings).

F_CONCAT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_CONCAT` dient der Verkettung (Konkatenation) von zwei Zeichenketten (Strings). Er ist Teil der Standard-Bibliothek für Zeichen- und String-Operationen gemäß IEC 61131-3.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ` (Service Request): Startet die Verkettungsoperation. Der Block benötigt beide Eingabestrings (`IN1` und `IN2`), um die Operation auszuführen.

Ereignis-Ausgänge

- `CNF` (Confirmation of Requested Service): Signalisiert die erfolgreiche Beendigung der Verkettungsoperation. Das Ergebnis wird über den Datenausgang `OUT` bereitgestellt.

Daten-Eingänge

- `IN1` (input string 1): Erster Eingabestring vom Typ `ANY_STRING`.
- `IN2` (input string 2): Zweiter Eingabestring vom Typ `ANY_STRING`.

Daten-Ausgänge

- `OUT` (string 1 + string 2): Ergebnis der Verkettung von `IN1` und `IN2` als `ANY_STRING`.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Bei Auslösung des Ereignisses `REQ` werden die beiden Eingabestrings `IN1` und `IN2` verkettet. Das Ergebnis wird über den Ausgang `OUT` ausgegeben und das Ereignis `CNF` signalisiert die erfolgreiche Beendigung der Operation.

Technische Besonderheiten

- Unterstützt den Datentyp `ANY_STRING`, was eine flexible Verwendung mit verschiedenen String-Typen ermöglicht.
- Einfache und direkte Implementierung ohne zusätzliche Parameter oder komplexe Zustandslogik.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Operation wird bei jedem `REQ`-Ereignis neu ausgeführt.

Anwendungsszenarien

- Kombination von Textteilen in automatisierten Nachrichtensystemen.
- Erstellung dynamischer Strings in Steuerungslogiken, z.B. für Protokollierung oder Benachrichtigungen.

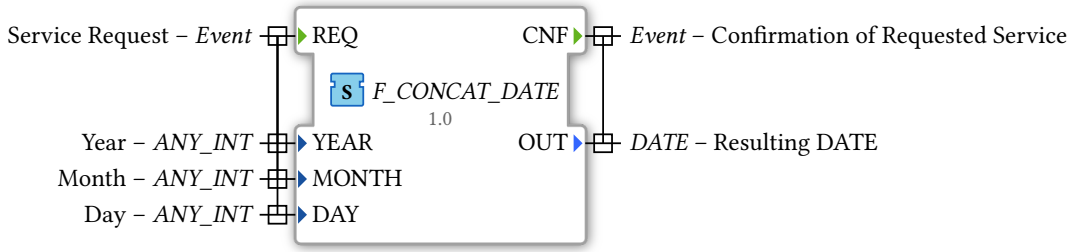
⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu `F_STRING_TO_*`-Blöcken, die Strings konvertieren, führt `F_CONCAT` keine Typumwandlung durch.
- Einfacher als Blöcke mit zusätzlichen Formatierungsoptionen wie `F_STRING_FORMAT`.

Fazit

`F_CONCAT` ist ein grundlegender und effizienter Baustein für die String-Verarbeitung in IEC 61131-3-Umgebungen. Seine Einfachheit und direkte Funktionsweise machen ihn ideal für grundlegende Verkettungsaufgaben.

F_CONCAT_DATE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `F_CONCAT_DATE` fügt einzelne Zeit-/Datumsbestandteile (`YEAR`, `MONTH`, `DAY`) zu einem zusammengesetzten `DATE`-Wert zusammen. Er ist die Umkehrung von F_SPLIT_DATE, das denselben `DATE`-Wert wieder in seine Einzelbestandteile zerlegt.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Zusammenführung aus, trägt `YEAR`, `MONTH`, `DAY`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT`.

Daten-Eingänge

- **YEAR** (ANY_INT): Jahr.
- **MONTH** (ANY_INT): Monat.
- **DAY** (ANY_INT): Tag.

Daten-Ausgänge

- **OUT** (`DATE`): Der aus den Einzelbestandteilen zusammengesetzte Wert.

Funktionsweise

Bei Eintreffen von `REQ` werden die Eingangswerte `YEAR`, `MONTH`, `DAY` zu einem `DATE`-Wert kombiniert und über `OUT` ausgegeben. Anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **`ANY_INT`-Eingänge**: Die Zeit-/Datumsbestandteile akzeptieren beliebige Ganzzahltypen, was die Verdrahtung mit unterschiedlich typisierten Quellwerten vereinfacht.
- **Keine Bereichsprüfung dokumentiert**: Der Baustein geht von plausiblen Eingabewerten aus (z. B. `MONTH` 1–12); eine explizite Validierung obliegt dem Aufrufer.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur Zusammenführung und zu `CNF`.

Anwendungsszenarien

- **Aufbau von Zeitstempeln** aus separat erfassten oder berechneten Einzelwerten, z. B. aus Sensordaten, Benutzereingaben oder Kommunikationsprotokollen.
- **Konfigurationsauswertung**: Zusammenführen von in einzelnen Variablen abgelegten Datums-/Zeitangaben zu einem verwendbaren `DATE`-Wert.

⚖️ Vergleich mit ähnlichen Bausteinen

- F_SPLIT_DATE: die Umkehrrichtung — zerlegt einen `DATE`-Wert in seine Einzelbestandteile.
- **`F_CONCAT_DATE_TOD`**: kombiniert stattdessen einen bereits fertigen `DATE`- und `TIME_OF_DAY`-Wert zu `DATE_AND_TIME`, statt aus Einzelfeldern.

Fazit

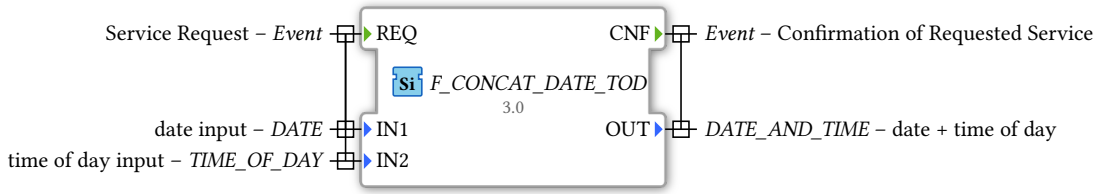
`F_CONCAT_DATE` liefert eine einfache, direkte Zusammenführung von Zeit-/Datumseinzelwerten zu einem `DATE`-Wert und ergänzt damit die entsprechende Zerlegungsfunktion F_SPLIT_DATE.

F_CONCAT_DATE_TOD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_CONCAT_DATE_TOD` dient zur Verkettung eines Datums (`DATE`) und einer Tageszeit (`TIME_OF_DAY`) zu einem kombinierten Datums- und Zeitstempel (`DATE_AND_TIME`). Dieser Baustein ist besonders nützlich in Szenarien, wo separate Datums- und Zeitinformationen zu einem einzigen Zeitstempel zusammengeführt werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Verarbeitung des Funktionsblocks. Bei Auslösung dieses Ereignisses werden die Eingabedaten `IN1` (Datum) und `IN2` (Tageszeit) verarbeitet.

Ereignis-Ausgänge

- **CNF**: Signalisiert die erfolgreiche Verarbeitung und gibt den kombinierten Zeitstempel `OUT` aus.

Daten-Eingänge

- **IN1** (`DATE`): Eingabe für das Datum.
- **IN2** (`TIME_OF_DAY`): Eingabe für die Tageszeit.

Daten-Ausgänge

- **OUT** (`DATE_AND_TIME`): Der kombinierte Datums- und Zeitstempel, der aus den Eingaben `IN1` und `IN2` erzeugt wird.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Bei Auslösung des Ereignisses `REQ` werden die Eingabedaten `IN1` (Datum) und `IN2` (Tageszeit) gelesen und zu einem `DATE_AND_TIME`-Wert kombiniert. Das Ergebnis wird über die Variable `OUT` ausgegeben und das Ereignis `CNF` signalisiert die erfolgreiche Verarbeitung.

Technische Besonderheiten

- Der Funktionsblock unterstützt die standardisierten Datentypen `DATE`, `TIME_OF_DAY` und `DATE_AND_TIME` gemäß IEC 61131-3.
- Die Verarbeitung erfolgt synchron mit dem Auslösen des `REQ`-Ereignisses.

Zustandsübersicht

1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Processing**: Verarbeitet die Eingabedaten und kombiniert sie zu einem `DATE_AND_TIME`-Wert.
3. **Output**: Gibt das Ergebnis über `OUT` aus und signalisiert `CNF`.

Anwendungsszenarien

- Kombination von separat erfassten Datums- und Zeitinformationen in Steuerungssystemen.
- Erzeugung von Zeitstempeln für Protokollierungs- und Überwachungszwecke.

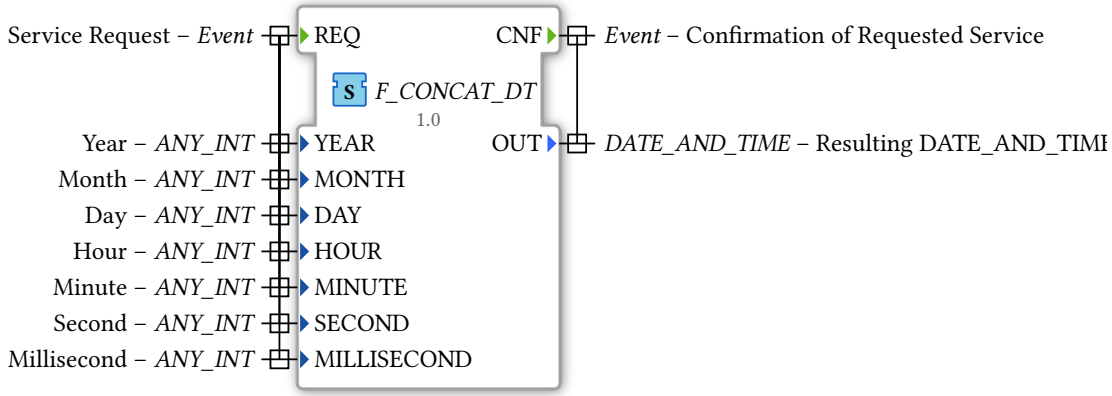
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Bausteinen, die ähnliche Funktionen bieten, zeichnet sich `F_CONCAT_DATE_TOD` durch seine Einfachheit und direkte Verarbeitung aus. Es gibt keine zusätzlichen Parameter oder komplexen Konfigurationsmöglichkeiten, was ihn besonders für einfache Anwendungsfälle geeignet macht.

Fazit

Der Funktionsblock `F_CONCAT_DATE_TOD` ist ein effizientes Werkzeug zur Kombination von Datums- und Zeitinformationen in Steuerungssystemen. Seine einfache Handhabung und klare Schnittstellenstruktur machen ihn zu einer zuverlässigen Lösung für grundlegende Zeitstempel-Operationen.

F_CONCAT_DT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `F_CONCAT_DT` fügt einzelne Zeit-/Datumsbestandteile (`YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`) zu einem zusammengesetzten `DATE_AND_TIME`-Wert zusammen. Er ist die Umkehrung von F_SPLIT_DT, das denselben `DATE_AND_TIME`-Wert wieder in seine Einzelbestandteile zerlegt.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Zusammenführung aus, trägt `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT`.

Daten-Eingänge

- **YEAR** (ANY_INT): Jahr.
- **MONTH** (ANY_INT): Monat.
- **DAY** (ANY_INT): Tag.
- **HOUR** (ANY_INT): Stunde.
- **MINUTE** (ANY_INT): Minute.
- **SECOND** (ANY_INT): Sekunde.
- **MILLISECOND** (ANY_INT): Millisekunde.

Daten-Ausgänge

- **OUT** (`DATE_AND_TIME`): Der aus den Einzelbestandteilen zusammengesetzte Wert.

Funktionsweise

Bei Eintreffen von `REQ` werden die Eingangswerte `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND` zu einem `DATE_AND_TIME`-Wert kombiniert und über `OUT` ausgegeben. Anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **`ANY_INT`-Eingänge**: Die Zeit-/Datumsbestandteile akzeptieren beliebige Ganzzahltypen, was die Verdrahtung mit unterschiedlich typisierten Quellwerten vereinfacht.
- **Keine Bereichsprüfung dokumentiert**: Der Baustein geht von plausiblen Eingabewerten aus (z. B. `MONTH` 1–12); eine explizite Validierung obliegt dem Aufrufer.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur Zusammenführung und zu `CNF`.

Anwendungsszenarien

- **Aufbau von Zeitstempeln** aus separat erfassten oder berechneten Einzelwerten, z. B. aus Sensordaten, Benutzereingaben oder Kommunikationsprotokollen.
- **Konfigurationsauswertung**: Zusammenführen von in einzelnen Variablen abgelegten Datums-/Zeitangaben zu einem verwendbaren `DATE_AND_TIME`-Wert.

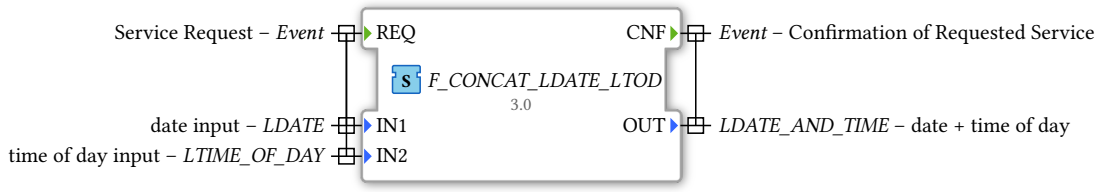
⚖️ Vergleich mit ähnlichen Bausteinen

- F_SPLIT_DT: die Umkehrrichtung — zerlegt einen `DATE_AND_TIME`-Wert in seine Einzelbestandteile.
- **`F_CONCAT_DATE_TOD`**: kombiniert stattdessen einen bereits fertigen `DATE`- und `TIME_OF_DAY`-Wert zu `DATE_AND_TIME`, statt aus Einzelfeldern.

Fazit

`F_CONCAT_DT` liefert eine einfache, direkte Zusammenführung von Zeit-/Datumseinzelwerten zu einem `DATE_AND_TIME`-Wert und ergänzt damit die entsprechende Zerlegungsfunktion F_SPLIT_DT.

F_CONCAT_LDATE_LTOD



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `F_CONCAT_LDATE_LTOD` verkettet ein Datum (`LDATE`) und eine Tageszeit (`LTIME_OF_DAY`) zu einem kombinierten `LDATE_AND_TIME`-Zeitstempel. Er ist das `L`-präfigierte (64-Bit-Auflösung) Gegenstück zu `F_CONCAT_DATE_TOD`.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Verkettung aus, trägt `IN1` und `IN2`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT`.

Daten-Eingänge

- **IN1** (LDATE): Das Datum.
- **IN2** (LTIME_OF_DAY): Die Tageszeit.

Daten-Ausgänge

- **OUT** (LDATE_AND_TIME): Der kombinierte Datums- und Zeitstempel.

Funktionsweise

Bei Eintreffen von `REQ` werden `IN1` (Datum) und `IN2` (Tageszeit) zu einem `LDATE_AND_TIME`-Wert kombiniert und über `OUT` ausgegeben. Anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **`L`-Datentypen**: Verwendet die hochauflösenden `LDATE`/`LTIME_OF_DAY`/`LDATE_AND_TIME`-Typen mit Nanosekunden-Auflösung statt der klassischen `DATE`/`TIME_OF_DAY`/`DATE_AND_TIME`-Typen.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur Verkettung und zu `CNF`.

Anwendungsszenarien

- **Kombination separat erfasster hochauflösender Datums- und Zeitinformationen** in Steuerungssystemen, die durchgängig mit `L`-Zeittypen arbeiten.

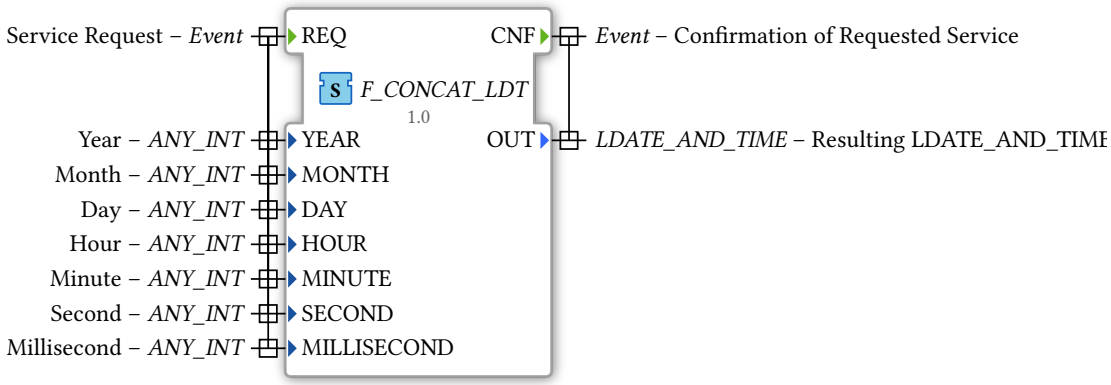
⚖️ Vergleich mit ähnlichen Bausteinen

- **`F_CONCAT_DATE_TOD`**: dieselbe Funktion mit den klassischen `DATE`/`TIME_OF_DAY`/`DATE_AND_TIME`-Typen.
- F_CONCAT_LDT: kombiniert stattdessen Einzelfelder (`YEAR`, `MONTH`, …) statt bereits fertiger `LDATE`/`LTIME_OF_DAY`-Werte.

Fazit

`F_CONCAT_LDATE_LTOD` liefert die hochauflösende Variante der Datums-/Zeit-Verkettung und eignet sich für Applikationen, die konsequent mit `L`-Zeittypen arbeiten.

F_CONCAT_LDT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `F_CONCAT_LDT` fügt einzelne Zeit-/Datumsbestandteile (`YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`) zu einem zusammengesetzten `LDATE_AND_TIME`-Wert zusammen. Er ist die Umkehrung von F_SPLIT_LDT, das denselben `LDATE_AND_TIME`-Wert wieder in seine Einzelbestandteile zerlegt.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Zusammenführung aus, trägt `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT`.

Daten-Eingänge

- **YEAR** (ANY_INT): Jahr.
- **MONTH** (ANY_INT): Monat.
- **DAY** (ANY_INT): Tag.
- **HOUR** (ANY_INT): Stunde.
- **MINUTE** (ANY_INT): Minute.
- **SECOND** (ANY_INT): Sekunde.
- **MILLISECOND** (ANY_INT): Millisekunde.

Daten-Ausgänge

- **OUT** (`LDATE_AND_TIME`): Der aus den Einzelbestandteilen zusammengesetzte Wert.

Funktionsweise

Bei Eintreffen von `REQ` werden die Eingangswerte `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND` zu einem `LDATE_AND_TIME`-Wert kombiniert und über `OUT` ausgegeben. Anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **`ANY_INT`-Eingänge**: Die Zeit-/Datumsbestandteile akzeptieren beliebige Ganzzahltypen, was die Verdrahtung mit unterschiedlich typisierten Quellwerten vereinfacht.
- **Keine Bereichsprüfung dokumentiert**: Der Baustein geht von plausiblen Eingabewerten aus (z. B. `MONTH` 1–12); eine explizite Validierung obliegt dem Aufrufer.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur Zusammenführung und zu `CNF`.

Anwendungsszenarien

- **Aufbau von Zeitstempeln** aus separat erfassten oder berechneten Einzelwerten, z. B. aus Sensordaten, Benutzereingaben oder Kommunikationsprotokollen.
- **Konfigurationsauswertung**: Zusammenführen von in einzelnen Variablen abgelegten Datums-/Zeitangaben zu einem verwendbaren `LDATE_AND_TIME`-Wert.

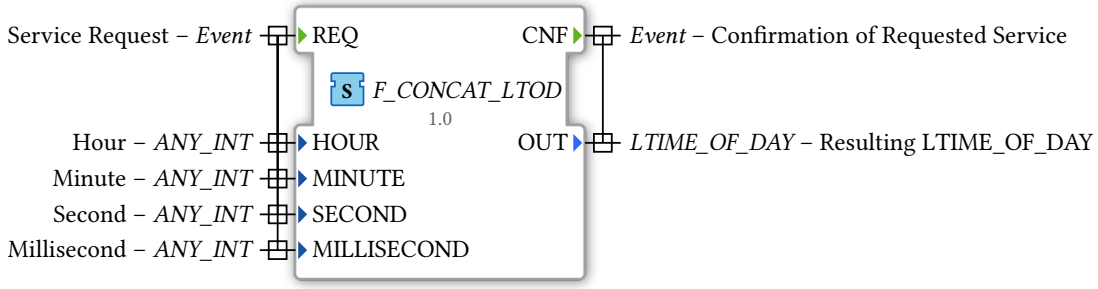
⚖️ Vergleich mit ähnlichen Bausteinen

- **F_SPLIT_LDT**: die Umkehrrichtung — zerlegt einen `LDATE_AND_TIME`-Wert in seine Einzelbestandteile.
- **`F_CONCAT_DATE_TOD`**: kombiniert stattdessen einen bereits fertigen `DATE`- und `TIME_OF_DAY`-Wert zu `DATE_AND_TIME`, statt aus Einzelfeldern.

Fazit

`F_CONCAT_LDT` liefert eine einfache, direkte Zusammenführung von Zeit-/Datumseinzelwerten zu einem `LDATE_AND_TIME`-Wert und ergänzt damit die entsprechende Zerlegungsfunktion F_SPLIT_LDT.

F_CONCAT_LTOD



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `F_CONCAT_LTOD` fügt einzelne Zeit-/Datumsbestandteile (`HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`) zu einem zusammengesetzten `LTIME_OF_DAY`-Wert zusammen. Er ist die Umkehrung von F_SPLIT_LTOD, das denselben `LTIME_OF_DAY`-Wert wieder in seine Einzelbestandteile zerlegt.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Zusammenführung aus, trägt `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT`.

Daten-Eingänge

- **HOUR** (ANY_INT): Stunde.
- **MINUTE** (ANY_INT): Minute.
- **SECOND** (ANY_INT): Sekunde.
- **MILLISECOND** (ANY_INT): Millisekunde.

Daten-Ausgänge

- **OUT** (`LTIME_OF_DAY`): Der aus den Einzelbestandteilen zusammengesetzte Wert.

Funktionsweise

Bei Eintreffen von `REQ` werden die Eingangswerte `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND` zu einem `LTIME_OF_DAY`-Wert kombiniert und über `OUT` ausgegeben. Anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **`ANY_INT`-Eingänge**: Die Zeit-/Datumsbestandteile akzeptieren beliebige Ganzzahltypen, was die Verdrahtung mit unterschiedlich typisierten Quellwerten vereinfacht.
- **Keine Bereichsprüfung dokumentiert**: Der Baustein geht von plausiblen Eingabewerten aus (z. B. `MONTH` 1–12); eine explizite Validierung obliegt dem Aufrufer.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur Zusammenführung und zu `CNF`.

Anwendungsszenarien

- **Aufbau von Zeitstempeln** aus separat erfassten oder berechneten Einzelwerten, z. B. aus Sensordaten, Benutzereingaben oder Kommunikationsprotokollen.
- **Konfigurationsauswertung**: Zusammenführen von in einzelnen Variablen abgelegten Datums-/Zeitangaben zu einem verwendbaren `LTIME_OF_DAY`-Wert.

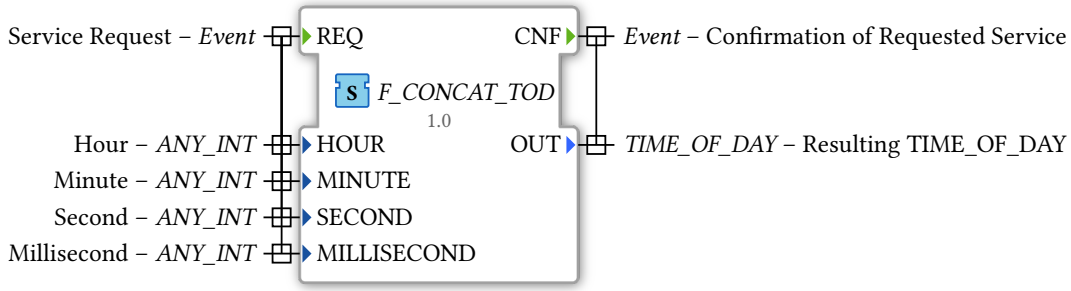
⚖️ Vergleich mit ähnlichen Bausteinen

- F_SPLIT_LTOD: die Umkehrrichtung — zerlegt einen `LTIME_OF_DAY`-Wert in seine Einzelbestandteile.
- **`F_CONCAT_DATE_TOD`**: kombiniert stattdessen einen bereits fertigen `DATE`- und `TIME_OF_DAY`-Wert zu `DATE_AND_TIME`, statt aus Einzelfeldern.

Fazit

`F_CONCAT_LTOD` liefert eine einfache, direkte Zusammenführung von Zeit-/Datumseinzelwerten zu einem `LTIME_OF_DAY`-Wert und ergänzt damit die entsprechende Zerlegungsfunktion F_SPLIT_LTOD.

F_CONCAT_TOD



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `F_CONCAT_TOD` fügt einzelne Zeit-/Datumsbestandteile (`HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`) zu einem zusammengesetzten `TIME_OF_DAY`-Wert zusammen. Er ist die Umkehrung von F_SPLIT_TOD, das denselben `TIME_OF_DAY`-Wert wieder in seine Einzelbestandteile zerlegt.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Zusammenführung aus, trägt `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT`.

Daten-Eingänge

- **HOUR** (ANY_INT): Stunde.
- **MINUTE** (ANY_INT): Minute.
- **SECOND** (ANY_INT): Sekunde.
- **MILLISECOND** (ANY_INT): Millisekunde.

Daten-Ausgänge

- **OUT** (`TIME_OF_DAY`): Der aus den Einzelbestandteilen zusammengesetzte Wert.

Funktionsweise

Bei Eintreffen von `REQ` werden die Eingangswerte `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND` zu einem `TIME_OF_DAY`-Wert kombiniert und über `OUT` ausgegeben. Anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **`ANY_INT`-Eingänge**: Die Zeit-/Datumsbestandteile akzeptieren beliebige Ganzzahltypen, was die Verdrahtung mit unterschiedlich typisierten Quellwerten vereinfacht.
- **Keine Bereichsprüfung dokumentiert**: Der Baustein geht von plausiblen Eingabewerten aus (z. B. `MONTH` 1–12); eine explizite Validierung obliegt dem Aufrufer.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur Zusammenführung und zu `CNF`.

Anwendungsszenarien

- **Aufbau von Zeitstempeln** aus separat erfassten oder berechneten Einzelwerten, z. B. aus Sensordaten, Benutzereingaben oder Kommunikationsprotokollen.
- **Konfigurationsauswertung**: Zusammenführen von in einzelnen Variablen abgelegten Datums-/Zeitangaben zu einem verwendbaren `TIME_OF_DAY`-Wert.

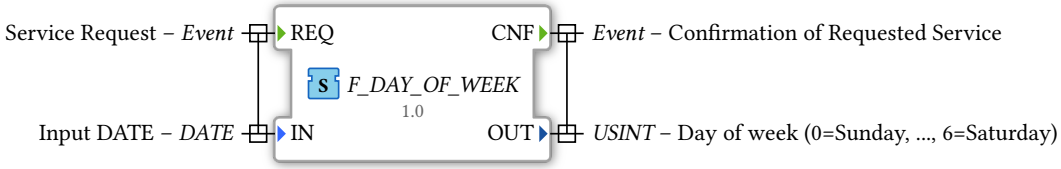
⚖️ Vergleich mit ähnlichen Bausteinen

- F_SPLIT_TOD: die Umkehrrichtung — zerlegt einen TIME_OF_DAY-Wert in seine Einzelbestandteile.
- **F_CONCAT_DATE_TOD**: kombiniert stattdessen einen bereits fertigen DATE- und TIME_OF_DAY-Wert zu DATE_AND_TIME, statt aus Einzelfeldern.

Fazit

F_CONCAT_TOD liefert eine einfache, direkte Zusammenführung von Zeit-/Datumseinzelwerten zu einem TIME_OF_DAY-Wert und ergänzt damit die entsprechende Zerlegungsfunktion F_SPLIT_TOD.

F_DAY_OF_WEEK



• – ◦ • – ◦ • – ◦ •

Einleitung

Der F_DAY_OF_WEEK ermittelt aus einem Datum (DATE) den zugehörigen Wochentag als Zahl. Er ist ein einfacher Funktionsbaustein zur Kalenderberechnung nach IEC 61131-3.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Berechnung aus, trägt IN.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt OUT.

Daten-Eingänge

- **IN** (DATE): Das Eingabedatum.

Daten-Ausgänge

- **OUT** (USINT): Der Wochentag (0 = Sonntag, 1 = Montag, ..., 6 = Samstag).

Funktionsweise

Bei Eintreffen von REQ berechnet die Algorithmus-ST-Anweisung OUT := DAY_OF_WEEK(IN) den Wochentag von IN und gibt ihn über OUT aus. Anschließend wird CNF ausgelöst.

Technische Besonderheiten

- **Kodierung 0–6**: Sonntag ist 0, Samstag ist 6 — diese Kodierung sollte beim Vergleich mit anderen, ggf. Montag-basierten Wochentagskonventionen beachtet werden.

Zustandsübersicht

Zustandslos: jedes REQ führt unmittelbar zur Berechnung und zu CNF.

Anwendungsszenarien

- **Wochentagsabhängige Steuerungslogik**: z. B. unterschiedliche Betriebsprogramme an Werktagen vs. Wochenenden.
- **Anzeige/Protokollierung**: Ausgabe des Wochentags neben einem Datum in Anzeigen oder Log-Einträgen.
- **Terminplanung**: Prüfung, ob ein berechnetes Datum auf einen bestimmten Wochentag fällt.

⚖️ Vergleich mit ähnlichen Bausteinen

- F_SPLIT_DATE: zerlegt ein Datum in Jahr/Monat/Tag, berechnet aber keinen Wochentag.

Fazit

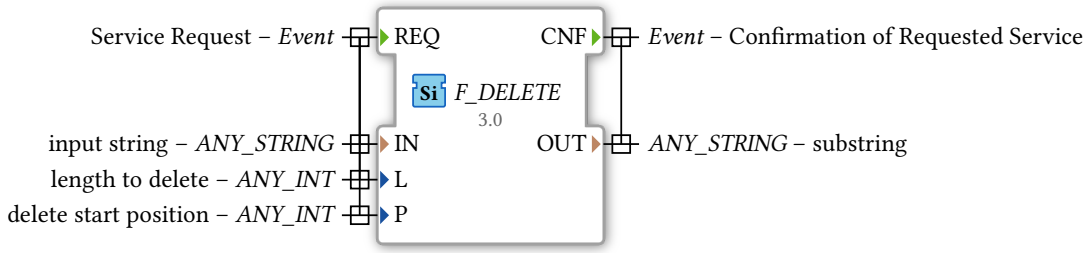
F_DAY_OF_WEEK liefert eine einfache Kalenderberechnung zur Ermittlung des Wochentags aus einem DATE-Wert und eignet sich für wochentagsabhängige Steuerungs- und Anzeigelogik.

F_DELETE

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock F_DELETE dient zur Manipulation von Zeichenketten. Er entfernt eine bestimmte Anzahl von Zeichen (L) aus einer Eingabezeichenkette (IN), beginnend an einer definierten Position (P). Das Ergebnis wird als neue Zeichenkette (OUT) ausgegeben.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Verarbeitung des Funktionsblocks. Die zugehörigen Daten sind IN, L und P.

Ereignis-Ausgänge

- **CNF**: Signalisiert die erfolgreiche Verarbeitung und liefert das Ergebnis OUT.

Daten-Eingänge

- **IN**: Die Eingabezeichenkette, aus der Zeichen entfernt werden sollen (Typ: ANY_STRING).
- **L**: Die Anzahl der zu entfernenden Zeichen (Typ: ANY_INT).
- **P**: Die Startposition, ab der Zeichen entfernt werden sollen (Typ: ANY_INT).

Daten-Ausgänge

- **OUT**: Die resultierende Zeichenkette nach dem Entfernen der Zeichen (Typ: ANY_STRING).

Adapter

Keine Adapter vorhanden.

Funktionsweise

1. Der Funktionsblock wird durch das Ereignis REQ aktiviert.
2. Aus der Zeichenkette IN werden L Zeichen beginnend an Position P entfernt.
3. Das Ergebnis wird in OUT gespeichert und das Ereignis CNF ausgelöst.

Technische Besonderheiten

- Der Funktionsblock unterstützt verschiedene String-Typen (ANY_STRING), was eine flexible Verwendung ermöglicht.
- Die Position P und die Länge L müssen gültige Werte innerhalb der Länge von IN sein, um korrekte Ergebnisse zu gewährleisten.

Zustandsübersicht

1. **Idle**: Wartet auf das Ereignis REQ.
2. **Processing**: Verarbeitet die Eingabedaten und entfernt die Zeichen.
3. **Completed**: Gibt das Ergebnis aus und signalisiert CNF.

Anwendungsszenarien

- Bereinigung von Benutzereingaben.
- Verarbeitung von Protokoll- oder Log-Daten.

- Manipulation von Textdaten in automatisierten Prozessen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_INSERT**: Fügt Zeichen an einer bestimmten Position ein.
- **F_REPLACE**: Ersetzt Zeichen in einer Zeichenkette.
- **F_SUBSTRING**: Extrahiert einen Teilstring aus einer Zeichenkette.

Fazit

Der `F_DELETE` Funktionsblock ist ein nützliches Werkzeug zur Bearbeitung von Zeichenketten in der 4diac-IDE. Durch seine einfache und flexible Schnittstelle kann er in verschiedenen Anwendungen eingesetzt werden, um Zeichen aus Strings zu entfernen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

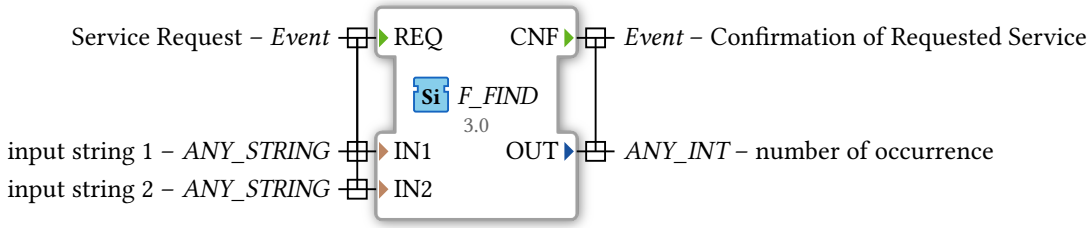
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

F_FIND

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_FIND` dient zur Ermittlung der Position des ersten Auftretens einer Zeichenkette (IN2) innerhalb einer anderen Zeichenkette (IN1). Falls die gesuchte Zeichenkette nicht gefunden wird, gibt der Block den Wert 0 zurück.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Suche nach der Zeichenkette. Die Eingänge IN1 und IN2 werden mit diesem Ereignis verknüpft.

Ereignis-Ausgänge

- **CNF**: Bestätigt die Beendigung der Suchoperation. Der Ausgang OUT wird mit diesem Ereignis verknüpft.

Daten-Eingänge

- **IN1** (ANY_STRING): Die Zeichenkette, in der gesucht wird.
- **IN2** (ANY_STRING): Die Zeichenkette, nach der gesucht wird.

Daten-Ausgänge

- **OUT** (ANY_INT): Die Position des ersten Auftretens von IN2 in IN1. Falls nicht gefunden, wird 0 zurückgegeben.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock `F_FIND` durchsucht die Zeichenkette IN1 nach dem ersten Auftreten der Zeichenkette IN2. Die Position des ersten Zeichens der gefundenen Zeichenkette wird als Ergebnis zurückgegeben. Falls IN2 nicht in IN1 enthalten ist, wird 0 zurückgegeben.

Technische Besonderheiten

- Unterstützt beliebige Zeichenketten-Typen (ANY_STRING).
- Gibt einen beliebigen Integer-Typ (ANY_INT) zurück.
- Die Suche ist case-sensitive (Groß-/Kleinschreibung wird beachtet).

Zustandsübersicht

1. **Idle**: Wartet auf das REQ-Ereignis.
2. **Processing**: Durchsucht IN1 nach IN2.
3. **Completed**: Sendet das CNF-Ereignis mit dem Ergebnis.

Anwendungsszenarien

- Suche nach Teilzeichenketten in Texten.
- Validierung von Eingabedaten.
- Verarbeitung von Zeichenketten in Automatisierungsprozessen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu `F_FIND_STRING` bietet `F_FIND` eine ähnliche Funktionalität, ist jedoch spezifisch für die Suche nach der ersten Position einer Teilzeichenkette.
- Andere Blöcke wie `F_MID` oder `F_LEFT` bieten zusätzliche Funktionen zur Zeichenkettenmanipulation, aber keine Suchfunktion.

Fazit

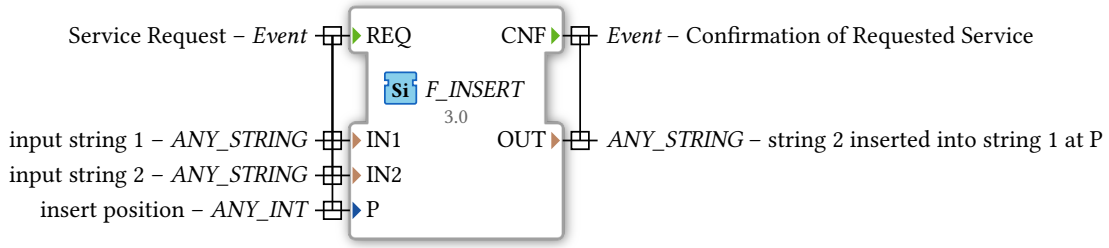
Der `F_FIND` Funktionsblock ist ein nützliches Werkzeug zur Suche nach Teilzeichenketten in Automatisierungsanwendungen. Seine einfache und effiziente Implementierung macht ihn ideal für die Integration in komplexere Steuerungslogiken.

F_INSERT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_INSERT` dient zur Manipulation von Zeichenketten. Er fügt eine Zeichenkette (`IN2`) in eine andere Zeichenkette (`IN1`) an einer bestimmten Position (`P`) ein. Der Block ist Teil der Standard-Charakter- und String-Funktionen gemäß IEC 61131-3.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Dienst-Anfrage. Löst die Ausführung des Funktionsblocks aus. Wird mit den Daten-Eingängen `IN1`, `IN2` und `P` verknüpft.

Ereignis-Ausgänge

- `CNF`: Bestätigung der angeforderten Dienstleistung. Wird mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- `IN1` (`ANY_STRING`): Die erste Eingabezeichenkette, in die `IN2` eingefügt wird.
- `IN2` (`ANY_STRING`): Die zweite Eingabezeichenkette, die in `IN1` eingefügt wird.
- `P` (`ANY_INT`): Die Position in `IN1`, an der `IN2` eingefügt werden soll.

Daten-Ausgänge

- `OUT` (`ANY_STRING`): Die resultierende Zeichenkette nach dem Einfügevorgang.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Bei Auslösung des Ereignisses `REQ` wird die Zeichenkette `IN2` in die Zeichenkette `IN1` an der durch `P` definierten Position eingefügt. Das Ergebnis dieser Operation wird über den Daten-Ausgang `OUT` ausgegeben und das Ereignis `CNF` wird ausgelöst, um die erfolgreiche Bearbeitung zu signalisieren.

Technische Besonderheiten

- Unterstützt alle String-Typen (`ANY_STRING`), was eine flexible Verwendung ermöglicht.
- Die Position `P` kann jeder ganzzahlige Wert sein (`ANY_INT`), muss jedoch innerhalb der Länge von `IN1` liegen, um ein definiertes Verhalten zu gewährleisten.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Operation wird bei jedem `REQ`-Ereignis neu ausgeführt.

Anwendungsszenarien

- Textverarbeitung in Automatisierungsanwendungen.
- Dynamische Erstellung von Log-Nachrichten oder Benachrichtigungen.
- Manipulation von Datenstrings vor der Weiterverarbeitung oder Ausgabe.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnlich zu `F_CONCAT`, das jedoch zwei Strings nur aneinanderhängt, ohne eine spezifische Einfügeposition zu berücksichtigen.
- Im Gegensatz zu `F_MID`, das Teile eines Strings extrahiert, fügt `F_INSERT` einen String in einen anderen ein.

Fazit

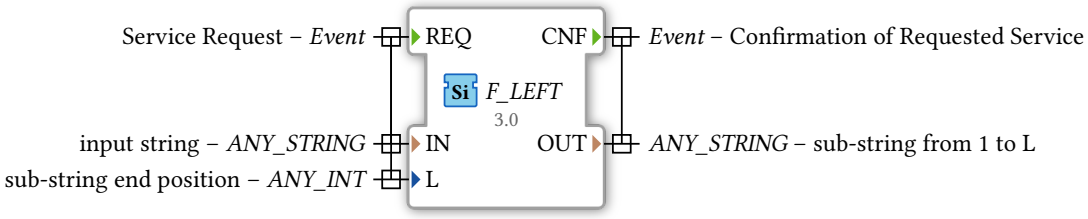
`F_INSERT` ist ein nützlicher Funktionsblock für Anwendungen, die eine präzise Manipulation von Zeichenketten erfordern. Seine einfache, aber effektive Funktionalität macht ihn zu einem wertvollen Werkzeug in der Automatisierungstechnik.

F_LEFT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LEFT` extrahiert die linken `L` Zeichen aus der Eingabezeichenkette `IN`. Er ist Teil der Standard-Character- und String-Funktionen gemäß IEC 61131-3.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Extraktion der Zeichenkette. Wird mit den Daten-Eingängen `IN` und `L` verknüpft.

Ereignis-Ausgänge

- **CNF**: Bestätigt die erfolgreiche Extraktion. Wird mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (`ANY_STRING`): Die Eingabezeichenkette, aus der die linken Zeichen extrahiert werden sollen.
- **L** (`ANY_INT`): Die Anzahl der Zeichen, die von links extrahiert werden sollen.

Daten-Ausgänge

- **OUT** (`ANY_STRING`): Die extrahierte Teilzeichenkette, bestehend aus den linken `L` Zeichen von `IN`.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Bei Auslösung des Ereignisses `REQ` extrahiert der Funktionsblock die ersten `L` Zeichen aus der Eingabezeichenkette `IN` und gibt das Ergebnis über `OUT` aus. Das Ereignis `CNF` signalisiert die erfolgreiche Ausführung.

Technische Besonderheiten

- Unterstützt alle String-Typen (`ANY_STRING`).
- `L` muss ein gültiger Integer-Wert sein. Ist `L` größer als die Länge von `IN`, wird die gesamte Zeichenkette zurückgegeben.

Zustandsübersicht

1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Processing**: Verarbeitet die Eingabe und extrahiert die Zeichen.
3. **Completed**: Sendet `CNF` mit dem Ergebnis.

Anwendungsszenarien

- Extraktion von Präfixen aus Zeichenketten.
- Verkürzung von Strings für spezifische Anforderungen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_RIGHT**: Extrahiert Zeichen von rechts.
- **F_MID**: Extrahiert Zeichen aus der Mitte einer Zeichenkette.

Fazit

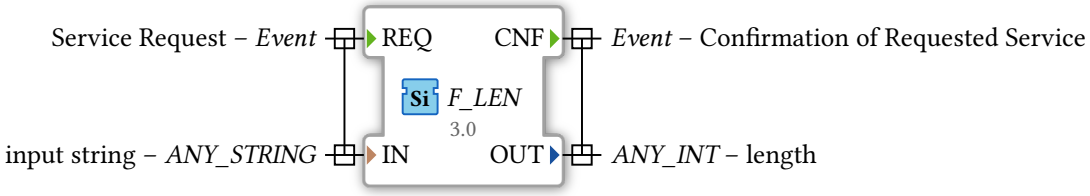
`F_LEFT` ist ein einfacher und effizienter Funktionsblock zur Extraktion von Teilzeichenketten aus dem Anfang eines Strings. Seine Verwendung ist intuitiv und erfüllt die Anforderungen gemäß IEC 61131-3.

F_LEN

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LEN` dient zur Bestimmung der Länge eines Eingabe-Strings. Er ist Teil der Standard-Bibliothek für Zeichen- und String-Funktionen gemäß IEC 61131-3.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Service-Anfrage. Wird dieses Ereignis ausgelöst, beginnt die Berechnung der String-Länge.

Ereignis-Ausgänge

- `CNF`: Bestätigung der Service-Anfrage. Wird ausgelöst, sobald die Berechnung abgeschlossen ist.

Daten-Eingänge

- `IN` (ANY_STRING): Der Eingabe-String, dessen Länge bestimmt werden soll.

Daten-Ausgänge

- `OUT` (ANY_INT): Die Länge des Eingabe-Strings.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock `F_LEN` berechnet die Länge des übergebenen Strings (`IN`) und gibt das Ergebnis als Integer (`OUT`) aus. Die Berechnung wird durch das Ereignis `REQ` gestartet und durch `CNF` bestätigt.

Technische Besonderheiten

- Unterstützt alle String-Typen (`ANY_STRING`).
- Gibt die Länge als beliebigen Integer-Typ (`ANY_INT`) zurück.

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf das `REQ`-Ereignis.
2. **Berechnungszustand**: Berechnet die Länge des Strings.
3. **Bestätigungszustand**: Sendet das `CNF`-Ereignis mit der berechneten Länge.

Anwendungsszenarien

- Überprüfung der Länge von Benutzereingaben.
- Validierung von String-Daten vor der weiteren Verarbeitung.
- Logging und Debugging von String-Operationen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnlich zu `LEN` in anderen Programmiersprachen, jedoch als standardisierter Funktionsblock in IEC 61131-3.
- Im Gegensatz zu `F_MID` oder `F_LEFT`, die Teile eines Strings extrahieren, liefert `F_LEN` nur die Länge.

Fazit

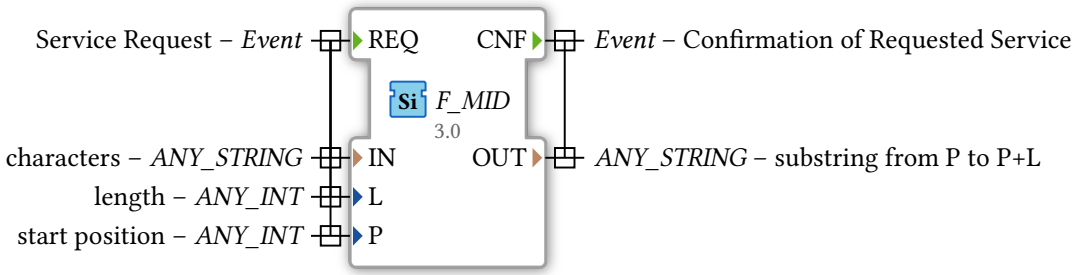
Der `F_LEN`-Funktionsblock ist ein einfaches, aber essentielles Werkzeug zur String-Verarbeitung in IEC 61131-3-Umgebungen. Seine klare Schnittstelle und einfache Funktionsweise machen ihn zu einem zuverlässigen Baustein für diverse Anwendungen.

F_MID

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_MID` dient zur Extraktion einer Teilzeichenkette aus einer gegebenen Zeichenkette. Dabei wird eine bestimmte Anzahl von Zeichen ab einer definierten Startposition extrahiert. Der FB ist Teil der Standard-Charakter- und String-Funktionen gemäß IEC 61131-3.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Service-Anfrage zur Ausführung der Extraktion. Wird mit den Daten-Eingängen `IN`, `L` und `P` verknüpft.

Ereignis-Ausgänge

- **CNF**: Bestätigung der erfolgreichen Ausführung der Anfrage. Wird mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Die Eingabezeichenkette, aus der die Teilzeichenkette extrahiert werden soll (Typ: `ANY_STRING`).
- **L**: Die Länge der zu extrahierenden Teilzeichenkette (Typ: `ANY_INT`).
- **P**: Die Startposition, ab der die Extraktion beginnen soll (Typ: `ANY_INT`).

Daten-Ausgänge

- **OUT**: Die extrahierte Teilzeichenkette (Typ: `ANY_STRING`).

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock `F_MID` extrahiert eine Teilzeichenkette aus der Eingabezeichenkette `IN`. Die Extraktion beginnt an der Position `P` und umfasst `L` Zeichen. Das Ergebnis wird über den Ausgang `OUT` ausgegeben, sobald der Ereignisausgang `CNF` ausgelöst wird.

Technische Besonderheiten

- Der FB unterstützt beliebige String-Typen (`ANY_STRING`) und Integer-Typen (`ANY_INT`).
- Die Startposition `P` und die Länge `L` müssen gültige Werte innerhalb der Länge der Eingabezeichenkette sein, um korrekte Ergebnisse zu gewährleisten.

Zustandsübersicht

1. **Initialisierung**: Der FB wartet auf eine Anfrage über `REQ`.
2. **Verarbeitung**: Bei Empfang von `REQ` werden die Eingabedaten verarbeitet und die Teilzeichenkette extrahiert.
3. **Ausgabe**: Die extrahierte Teilzeichenkette wird über `OUT` ausgegeben und `CNF` wird ausgelöst.

Anwendungsszenarien

- Extraktion von Teilzeichenketten aus längeren Texten oder Datenstrings.
- Verwendung in String-Verarbeitungsroutinen, z.B. zur Parsing von Datenformaten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_LEFT` oder `F_RIGHT` extrahieren Teilzeichenketten vom Anfang oder Ende einer Zeichenkette, während `F_MID` eine flexible Extraktion ab einer beliebigen Position ermöglicht.

Fazit

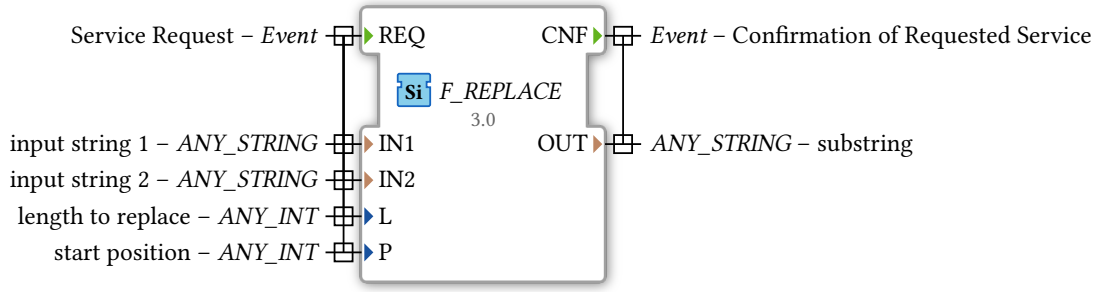
Der `F_MID`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, Teilzeichenketten aus einer gegebenen Zeichenkette zu extrahieren. Seine Flexibilität und Kompatibilität mit verschiedenen String- und Integer-Typen machen ihn zu einem nützlichen Werkzeug in der String-Verarbeitung.

F_REPLACE

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock F_REPLACE dient zur Manipulation von Zeichenketten. Er ersetzt eine bestimmte Anzahl von Zeichen (L) in der ersten Eingabezeichenkette (IN1) durch die zweite Eingabezeichenkette (IN2), beginnend an einer spezifizierten Position (P).



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Dienst-Anforderung. Löst die Ersetzungsoperation aus. Verknüpft mit den Daten-Eingängen IN1, IN2, L und P.

Ereignis-Ausgänge

- **CNF**: Bestätigung der angeforderten Dienstleistung. Wird ausgelöst, wenn die Ersetzung abgeschlossen ist. Verknüpft mit dem Daten-Ausgang OUT.

Daten-Eingänge

- **IN1** (ANY_STRING): Die erste Eingabezeichenkette, in der die Ersetzung vorgenommen wird.
- **IN2** (ANY_STRING): Die zweite Eingabezeichenkette, die als Ersatz verwendet wird.
- **L** (ANY_INT): Die Anzahl der zu ersetzenden Zeichen.
- **P** (ANY_INT): Die Startposition für die Ersetzung in IN1.

Daten-Ausgänge

- **OUT** (ANY_STRING): Die resultierende Zeichenkette nach der Ersetzung.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die folgende Operation durch:

1. Beim Empfang des REQ-Ereignisses werden die Eingabewerte IN1, IN2, L und P gelesen.
2. In der Zeichenkette IN1 werden ab Position P genau L Zeichen durch die Zeichenkette IN2 ersetzt.
3. Das Ergebnis wird in OUT gespeichert und das CNF-Ereignis ausgelöst.

Technische Besonderheiten

- Unterstützt beliebige Zeichenketten-Typen (ANY_STRING).
- Die Position P und die Länge L müssen gültige Indizes innerhalb der Zeichenkette IN1 sein, um korrekte Ergebnisse zu gewährleisten.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Operation wird bei jedem REQ-Ereignis neu ausgeführt.

Anwendungsszenarien

- Textverarbeitung in Automatisierungsanwendungen.
- Manipulation von Zeichenketten in Protokollnachrichten.
- Dynamische Anpassung von Konfigurationsstrings.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnlich zu F_INSERT, jedoch ersetzt F_REPLACE bestehende Zeichen statt neue einzufügen.
- Im Vergleich zu F_DELETE wird hier nicht nur gelöscht, sondern durch andere Zeichen ersetzt.

Fazit

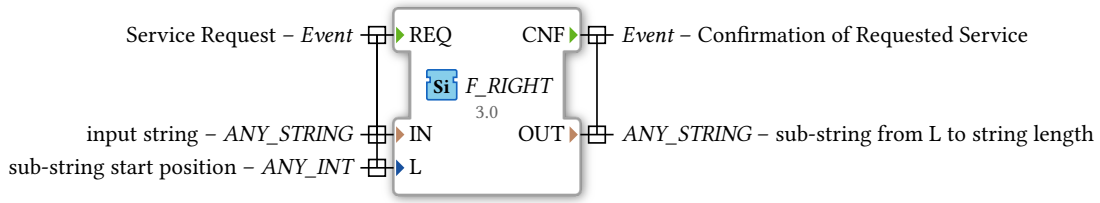
F_REPLACE ist ein nützlicher Funktionsblock für die Zeichenkettenmanipulation in IEC 61131-3 basierten Steuerungen. Seine einfache und klare Funktionsweise macht ihn ideal für verschiedene Anwendungen in der Automatisierungstechnik.

F_RIGHT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_RIGHT` ist ein Standard-Baustein für die Verarbeitung von Zeichenketten (Strings) nach der Norm IEC 61131-3. Er extrahiert die rechten `L` Zeichen aus der Eingabezeichenkette `IN` und gibt diese als Ergebnis `OUT` aus. Dieser Baustein ist besonders nützlich in Anwendungen, bei denen Teile von Zeichenketten verarbeitet oder analysiert werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Dienst-Anforderung. Löst die Extraktion der rechten `L` Zeichen aus. Wird mit den Daten `IN` und `L` verknüpft.

Ereignis-Ausgänge

- **CNF**: Bestätigung der angeforderten Dienstleistung. Wird mit dem Ergebnis `OUT` verknüpft.

Daten-Eingänge

- **IN**: Eingabezeichenkette, aus der die rechten Zeichen extrahiert werden sollen. Der Datentyp ist `ANY_STRING`.
- **L**: Anzahl der Zeichen, die von rechts aus der Eingabezeichenkette extrahiert werden sollen. Der Datentyp ist `ANY_INT`.

Daten-Ausgänge

- **OUT**: Ergebniszeichenkette, die die extrahierten Zeichen enthält. Der Datentyp ist `ANY_STRING`.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Bei Auslösung des Ereignisses `REQ` wird die Eingabezeichenkette `IN` verarbeitet. Der Funktionsblock extrahiert die rechten `L` Zeichen aus `IN` und gibt das Ergebnis über den Ausgang `OUT` aus. Das Ereignis `CNF` signalisiert die erfolgreiche Verarbeitung und stellt das Ergebnis bereit.

Technische Besonderheiten

- Unterstützt verschiedene String-Typen (`ANY_STRING`), was eine flexible Verwendung ermöglicht.
- Die Position `L` kann mit verschiedenen Integer-Typen (`ANY_INT`) angegeben werden.
- Der Baustein ist Teil des Pakets `iec61131::charString`.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Verarbeitung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Extraktion von Dateierweiterungen aus Dateinamen.

- Verarbeitung von festen Längenformaten, bei denen relevante Informationen am Ende der Zeichenkette stehen.
- Allgemeine Stringmanipulation in Steuerungs- und Automatisierungsanwendungen.

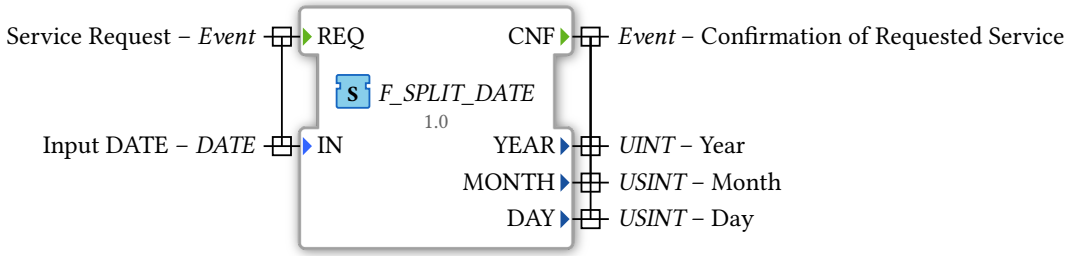
⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu `F_LEFT`, der die linken Zeichen einer Zeichenkette extrahiert, bietet `F_RIGHT` die umgekehrte Funktionalität.
- Andere String-Funktionsblöcke wie `F_MID` ermöglichen die Extraktion von Teilzeichenketten aus der Mitte einer Zeichenkette, während `F_RIGHT` speziell auf die rechten Zeichen fokussiert ist.

Fazit

Der Funktionsblock `F_RIGHT` ist ein einfacher und effektiver Baustein für die Extraktion von Zeichen vom Ende einer Zeichenkette. Seine Flexibilität in Bezug auf die unterstützten Datentypen macht ihn zu einem vielseitigen Werkzeug in der Stringverarbeitung.

F_SPLIT_DATE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `F_SPLIT_DATE` zerlegt einen `DATE`-Wert in seine einzelnen Zeit-/Datumsbestandteile (`YEAR`, `MONTH`, `DAY`). Er ist die Umkehrung von F_CONCAT_DATE, das dieselben Einzelbestandteile wieder zu einem `DATE`-Wert zusammenfügt.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Zerlegung aus, trägt `IN`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `YEAR`, `MONTH`, `DAY`.

Daten-Eingänge

- **IN** (`DATE`): Der zu zerlegende Zeit-/Datumswert.

Daten-Ausgänge

- **YEAR** (UINT): Jahr.
- **MONTH** (USINT): Monat.
- **DAY** (USINT): Tag.

Funktionsweise

Bei Eintreffen von `REQ` wird `IN` in seine Einzelbestandteile zerlegt, die über `YEAR`, `MONTH`, `DAY` ausgegeben werden. Anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **Feste Ausgabetypen**: Anders als bei F_CONCAT_DATE (generisch `ANY_INT`) sind die Ausgänge hier auf konkrete, für den jeweiligen Wertebereich passend gewählte Ganzzahltypen festgelegt (z. B. `USINT` für Monat/Tag, deren Wertebereich klein ist).

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur Zerlegung und zu `CNF`.

Anwendungsszenarien

- **Anzeige/Weiterverarbeitung einzelner Zeit-/Datumsbestandteile**, z. B. für eine formatierte Anzeige auf einem Panel oder für Berechnungen, die nur einen bestimmten Bestandteil benötigen.
- **Protokollierung**: Aufschlüsselung eines Zeitstempels in seine Einzelwerte für strukturierte Log-Ausgaben.

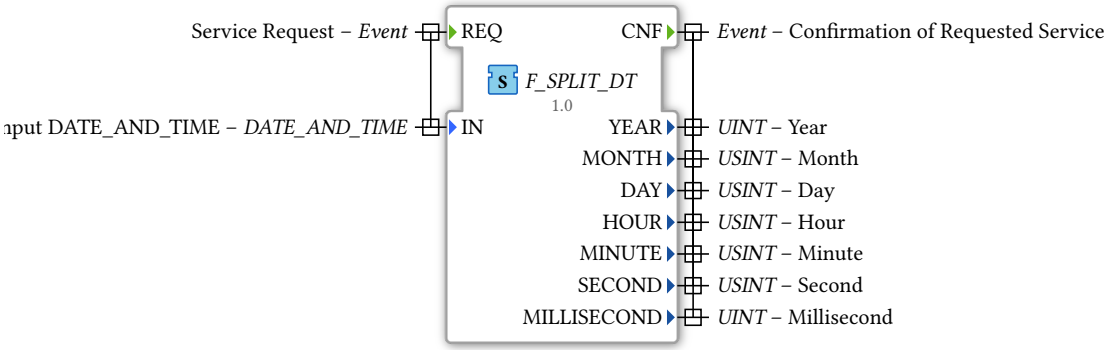
⚖️ Vergleich mit ähnlichen Bausteinen

- F_CONCAT_DATE: die Umkehrrichtung — fügt Einzelbestandteile zu einem `DATE`-Wert zusammen.

Fazit

`F_SPLIT_DATE` liefert eine einfache, direkte Zerlegung eines `DATE`-Werts in seine Einzelbestandteile und ergänzt damit die entsprechende Zusammenführungsfunktion F_CONCAT_DATE.

F_SPLIT_DT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `F_SPLIT_DT` zerlegt einen `DATE_AND_TIME`-Wert in seine einzelnen Zeit-/Datumsbestandteile (`YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`). Er ist die Umkehrung von F_CONCAT_DT, das dieselben Einzelbestandteile wieder zu einem `DATE_AND_TIME`-Wert zusammenfügt.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Zerlegung aus, trägt `IN`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`.

Daten-Eingänge

- **IN** (`DATE_AND_TIME`): Der zu zerlegende Zeit-/Datumswert.

Daten-Ausgänge

- **YEAR** (UINT): Jahr.
- **MONTH** (USINT): Monat.
- **DAY** (USINT): Tag.
- **HOUR** (USINT): Stunde.
- **MINUTE** (USINT): Minute.
- **SECOND** (USINT): Sekunde.
- **MILLISECOND** (UINT): Millisekunde.

Funktionsweise

Bei Eintreffen von `REQ` wird `IN` in seine Einzelbestandteile zerlegt, die über `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND` ausgegeben werden. Anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **Feste Ausgabetypen**: Anders als bei F_CONCAT_DT (generisch `ANY_INT`) sind die Ausgänge hier auf konkrete, für den jeweiligen Wertebereich passend gewählte Ganzzahltypen festgelegt (z. B. `USINT` für Monat/Tag, deren Wertebereich klein ist).

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur Zerlegung und zu `CNF`.

Anwendungsszenarien

- **Anzeige/Weiterverarbeitung einzelner Zeit-/Datumsbestandteile**, z. B. für eine formatierte Anzeige auf einem Panel oder für Berechnungen, die nur einen bestimmten Bestandteil benötigen.
- **Protokollierung**: Aufschlüsselung eines Zeitstempels in seine Einzelwerte für strukturierte Log-Ausgaben.

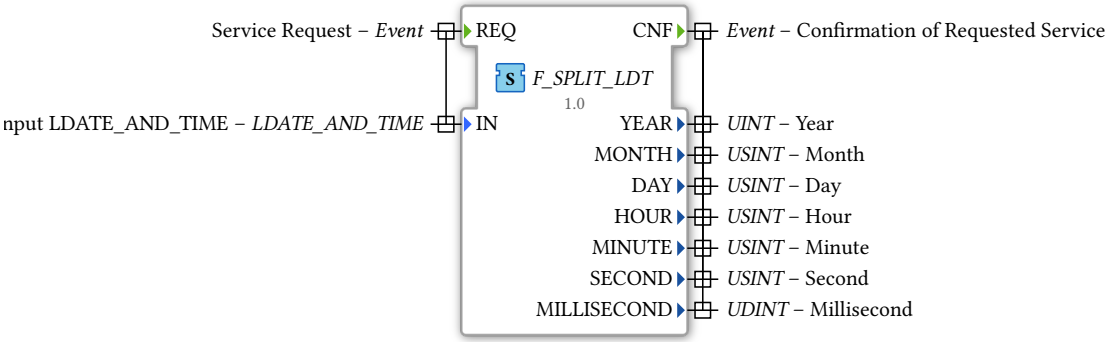
⚖️ Vergleich mit ähnlichen Bausteinen

- **F_CONCAT_DT**: die Umkehrrichtung — fügt Einzelbestandteile zu einem `DATE_AND_TIME`-Wert zusammen.

Fazit

`F_SPLIT_DT` liefert eine einfache, direkte Zerlegung eines `DATE_AND_TIME`-Werts in seine Einzelbestandteile und ergänzt damit die entsprechende Zusammenführungsfunktion F_CONCAT_DT.

F_SPLIT_LDT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `F_SPLIT_LDT` zerlegt einen `LDATE_AND_TIME`-Wert in seine einzelnen Zeit-/Datumsbestandteile (`YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`). Er ist die Umkehrung von F_CONCAT_LDT, das dieselben Einzelbestandteile wieder zu einem `LDATE_AND_TIME`-Wert zusammenfügt.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Zerlegung aus, trägt `IN`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`.

Daten-Eingänge

- **IN** (`LDATE_AND_TIME`): Der zu zerlegende Zeit-/Datumswert.

Daten-Ausgänge

- **YEAR** (UINT): Jahr.
- **MONTH** (USINT): Monat.
- **DAY** (USINT): Tag.
- **HOUR** (USINT): Stunde.
- **MINUTE** (USINT): Minute.
- **SECOND** (USINT): Sekunde.
- **MILLISECOND** (UDINT): Millisekunde.

Funktionsweise

Bei Eintreffen von `REQ` wird `IN` in seine Einzelbestandteile zerlegt, die über `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND` ausgegeben werden. Anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **Feste Ausgabetypen**: Anders als bei F_CONCAT_LDT (generisch `ANY_INT`) sind die Ausgänge hier auf konkrete, für den jeweiligen Wertebereich passend gewählte Ganzzahltypen festgelegt (z. B. `USINT` für Monat/Tag, deren Wertebereich klein ist).

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur Zerlegung und zu `CNF`.

Anwendungsszenarien

- **Anzeige/Weiterverarbeitung einzelner Zeit-/Datumsbestandteile**, z. B. für eine formatierte Anzeige auf einem Panel oder für Berechnungen, die nur einen bestimmten Bestandteil benötigen.
- **Protokollierung**: Aufschlüsselung eines Zeitstempels in seine Einzelwerte für strukturierte Log-Ausgaben.

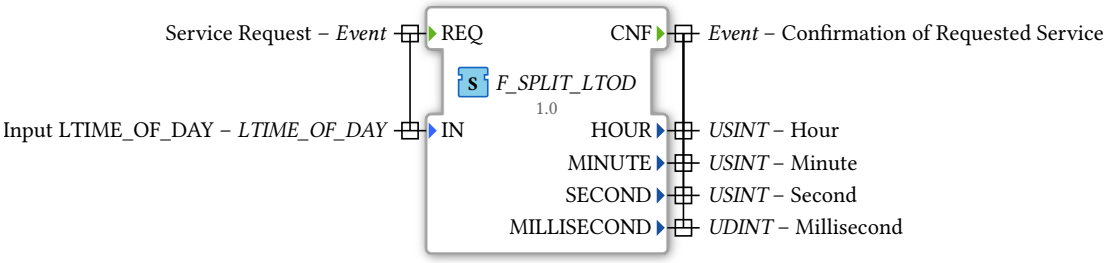
⚖️ Vergleich mit ähnlichen Bausteinen

- **F_CONCAT_LDT**: die Umkehrrichtung — fügt Einzelbestandteile zu einem `LDATE_AND_TIME`-Wert zusammen.

Fazit

`F_SPLIT_LDT` liefert eine einfache, direkte Zerlegung eines `LDATE_AND_TIME`-Werts in seine Einzelbestandteile und ergänzt damit die entsprechende Zusammenführungsfunktion F_CONCAT_LDT.

F_SPLIT_LTOD



• – ○ • – ○ • – ○ •

Einleitung

Der `F_SPLIT_LTOD` zerlegt einen `LTIME_OF_DAY`-Wert in seine einzelnen Zeit-/Datumsbestandteile (`HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`). Er ist die Umkehrung von F_CONCAT_LTOD, das dieselben Einzelbestandteile wieder zu einem `LTIME_OF_DAY`-Wert zusammenfügt.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Zerlegung aus, trägt `IN`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`.

Daten-Eingänge

- **IN** (`LTIME_OF_DAY`): Der zu zerlegende Zeit-/Datumswert.

Daten-Ausgänge

- **HOUR** (USINT): Stunde.
- **MINUTE** (USINT): Minute.
- **SECOND** (USINT): Sekunde.
- **MILLISECOND** (UDINT): Millisekunde.

Funktionsweise

Bei Eintreffen von `REQ` wird `IN` in seine Einzelbestandteile zerlegt, die über `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND` ausgegeben werden. Anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **Feste Ausgabetypen**: Anders als bei F_CONCAT_LTOD (generisch `ANY_INT`) sind die Ausgänge hier auf konkrete, für den jeweiligen Wertebereich passend gewählte Ganzzahltypen festgelegt (z. B. `USINT` für Monat/Tag, deren Wertebereich klein ist).

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur Zerlegung und zu `CNF`.

Anwendungsszenarien

- **Anzeige/Weiterverarbeitung einzelner Zeit-/Datumsbestandteile**, z. B. für eine formatierte Anzeige auf einem Panel oder für Berechnungen, die nur einen bestimmten Bestandteil benötigen.
- **Protokollierung**: Aufschlüsselung eines Zeitstempels in seine Einzelwerte für strukturierte Log-Ausgaben.

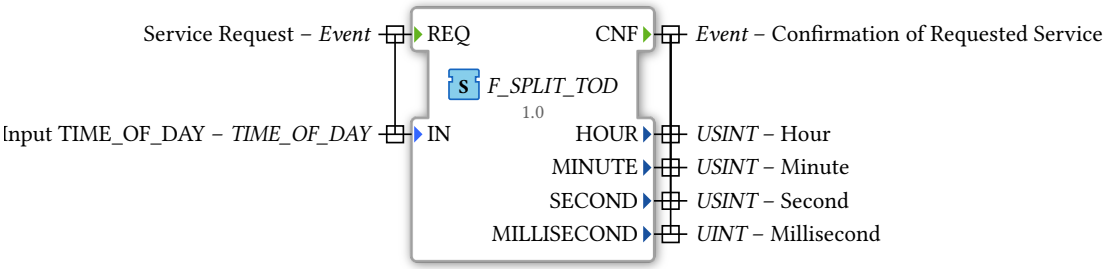
⚖️ Vergleich mit ähnlichen Bausteinen

- F_CONCAT_LTOD: die Umkehrrichtung — fügt Einzelbestandteile zu einem `LTIME_OF_DAY`-Wert zusammen.

Fazit

`F_SPLIT_LTOD` liefert eine einfache, direkte Zerlegung eines `LTIME_OF_DAY`-Werts in seine Einzelbestandteile und ergänzt damit die entsprechende Zusammenführungsfunktion F_CONCAT_LTOD.

F_SPLIT_TOD



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `F_SPLIT_TOD` zerlegt einen `TIME_OF_DAY`-Wert in seine einzelnen Zeit-/Datumsbestandteile (`HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`). Er ist die Umkehrung von F_CONCAT_TOD, das dieselben Einzelbestandteile wieder zu einem `TIME_OF_DAY`-Wert zusammenfügt.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Zerlegung aus, trägt `IN`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`.

Daten-Eingänge

- **IN** (`TIME_OF_DAY`): Der zu zerlegende Zeit-/Datumswert.

Daten-Ausgänge

- **HOUR** (USINT): Stunde.
- **MINUTE** (USINT): Minute.
- **SECOND** (USINT): Sekunde.
- **MILLISECOND** (UINT): Millisekunde.

Funktionsweise

Bei Eintreffen von `REQ` wird `IN` in seine Einzelbestandteile zerlegt, die über `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND` ausgegeben werden. Anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **Feste Ausgabetypen**: Anders als bei F_CONCAT_TOD (generisch `ANY_INT`) sind die Ausgänge hier auf konkrete, für den jeweiligen Wertebereich passend gewählte Ganzzahltypen festgelegt (z. B. `USINT` für Monat/Tag, deren Wertebereich klein ist).

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur Zerlegung und zu `CNF`.

Anwendungsszenarien

- **Anzeige/Weiterverarbeitung einzelner Zeit-/Datumsbestandteile**, z. B. für eine formatierte Anzeige auf einem Panel oder für Berechnungen, die nur einen bestimmten Bestandteil benötigen.
- **Protokollierung**: Aufschlüsselung eines Zeitstempels in seine Einzelwerte für strukturierte Log-Ausgaben.

⚖️ Vergleich mit ähnlichen Bausteinen

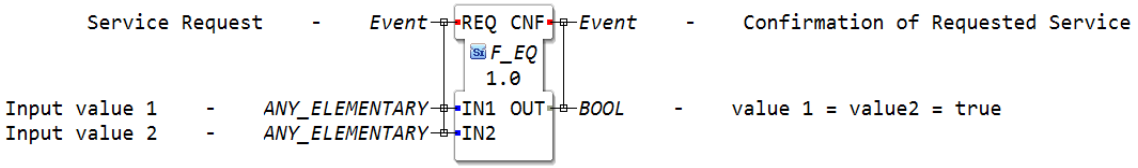
- **F_CONCAT_TOD**: die Umkehrrichtung — fügt Einzelbestandteile zu einem `TIME_OF_DAY`-Wert zusammen.

Fazit

`F_SPLIT_TOD` liefert eine einfache, direkte Zerlegung eines `TIME_OF_DAY`-Werts in seine Einzelbestandteile und ergänzt damit die entsprechende Zusammenführungsfunktion F_CONCAT_TOD.

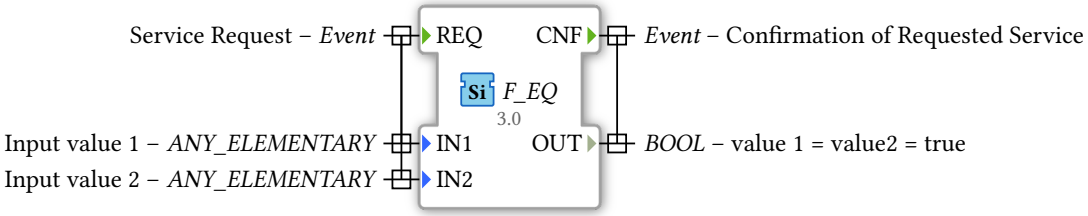
2.4.1.6 ⚖️ Vergleichsfunktionen

F_EQ



https://content.helpme-codesys.com/en/CODESYS%20Development%20System/_cds_operator_eq.html

• – ◦ • – ◦ • – ◦ •



Schnittstelle

Ereignis-Eingänge

- `REQ`: Dienstanforderung, trägt `IN1` und `IN2`.

Ereignis-Ausgänge

- `CNF`: Bestätigung des angeforderten Dienstes, trägt `OUT`.

Daten-Eingänge

- `IN1` (`ANY_ELEMENTARY`): Eingangswert 1.
- `IN2` (`ANY_ELEMENTARY`): Eingangswert 2.

Daten-Ausgänge

- `OUT` (`BOOL`): `TRUE`, wenn Wert 1 = Wert 2.

Folgender Text wurde erzeugt von https://chat.deepseek.com/ am 05.02.2025 und noch nicht korrekturgelesen (HF)

Der EQ-Operator in IEC 61499: Eine Einführung

Die IEC 61499 ist eine Norm, die sich mit der Funktionsbausteinmodellierung für industrielle Prozessmess- und Steuerungssysteme beschäftigt. Sie bietet ein Framework für die Entwicklung von verteilten Steuerungssystemen, das auf der Verwendung von Funktionsbausteinen basiert. Ein zentrales Element in dieser Norm ist die Definition von Operatoren, die in den Funktionsbausteinen verwendet werden, um logische und arithmetische Operationen durchzuführen. Einer dieser Operatoren ist der EQ-Operator, der in diesem Aufsatz näher betrachtet werden soll.

Was ist der EQ-Operator?

Der EQ-Operator, auch bekannt als Gleichheitsoperator, ist ein logischer Operator, der verwendet wird, um zu überprüfen, ob zwei Werte gleich sind. In der IEC 61499 wird dieser Operator häufig in den Funktionsbausteinen eingesetzt, um Bedingungen zu evaluieren und Entscheidungen zu treffen. Der EQ-Operator vergleicht zwei Eingangswerte und gibt einen booleschen Wert (TRUE oder FALSE) zurück, je nachdem, ob die Werte gleich sind oder nicht.

Syntax und Verwendung

In der IEC 61499 wird der EQ-Operator typischerweise in den Ausdrücken von Funktionsbausteinen verwendet. Die Syntax des EQ-Operators ist einfach:
`EQ(Eingang1, Eingang2)`
Hierbei sind `Eingang1` und `Eingang2` die beiden Werte, die verglichen werden sollen. Der Rückgabewert des Operators ist `TRUE`, wenn `Eingang1` und `Eingang2` gleich sind, andernfalls ist der Rückgabewert `FALSE`.

Anwendungsbeispiele

1. **Vergleich von Sensorwerten:** Angenommen, ein Funktionsbaustein soll überprüfen, ob zwei Sensoren den gleichen Wert liefern. Der EQ-Operator kann verwendet werden, um die Werte der beiden Sensoren zu vergleichen:

```
IF EQ(Sensor1, Sensor2) THEN
    // Aktion ausführen, wenn die Sensorwerte gleich sind
ELSE
    // Aktion ausführen, wenn die Sensorwerte unterschiedlich sind
END_IF
```

2. **Steuerung von Prozessschritten:** In einem Produktionsprozess kann der EQ-Operator verwendet werden, um zu überprüfen, ob ein bestimmter Prozessschritt abgeschlossen ist. Wenn der aktuelle Schritt gleich dem Zielschritt ist, kann der nächste Schritt eingeleitet werden:

```
IF EQ(AktuellerSchritt, ZielSchritt) THEN
    // Nächsten Prozessschritt einleiten
END_IF
```

3. **Fehlererkennung:** Der EQ-Operator kann auch zur Fehlererkennung eingesetzt werden. Wenn beispielsweise ein Sollwert und ein Istwert verglichen werden und diese nicht übereinstimmen, kann ein Fehlersignal generiert werden:

```
IF NOT EQ(Sollwert, Istwert) THEN
    // Fehlersignal generieren
END_IF
```

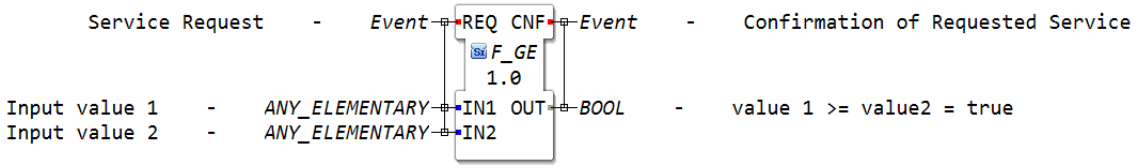
Vorteile des EQ-Operators

- **Einfachheit:** Der EQ-Operator ist einfach zu verstehen und zu verwenden, was die Entwicklung von Steuerungslogik erleichtert.
- **Flexibilität:** Der Operator kann auf verschiedene Datentypen angewendet werden, einschließlich Ganzzahlen, Gleitkommazahlen und sogar boolesche Werte.
- **Effizienz:** Da der EQ-Operator direkt in den Funktionsbausteinen verwendet werden kann, reduziert er den Bedarf an zusätzlicher Logik und macht den Code effizienter.

Fazit

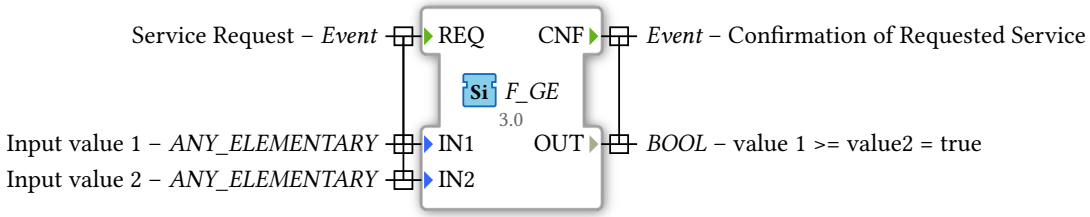
Der EQ-Operator ist ein grundlegendes Werkzeug in der IEC 61499, das die Implementierung von logischen Vergleichen in industriellen Steuerungssystemen vereinfacht. Durch seine einfache Syntax und vielseitige Anwendbarkeit trägt er dazu bei, die Entwicklung von robusten und effizienten Steuerungsalgorithmen zu unterstützen. Ob in der Sensorüberwachung, Prozesssteuerung oder Fehlererkennung – der EQ-Operator ist ein unverzichtbares Element in der Welt der industriellen Automatisierung.

F_GE



https://content.helpme-codesys.com/en/CODESYS%20Development%20System/_cds_operator_ge.html

• – ◦ • – ◦ • – ◦ •



Schnittstelle

Port	Typ	Beschreibung
REQ	Event	Service-Anforderung (mit IN1, IN2)
CNF	Event	Bestätigung der angeforderten Service-Operation (mit OUT)
IN1	ANY_ELEMENTARY	Eingangswert 1
IN2	ANY_ELEMENTARY	Eingangswert 2
OUT	BOOL	TRUE, wenn IN1 größer oder gleich IN2

Folgender Text wurde erzeugt von https://chat.deepseek.com/ am 05.02.2025 und noch nicht korrekturgelesen (HF)

Der GE-Operator in IEC 61499: Eine Einführung

Die IEC 61499 ist eine internationale Norm, die sich mit der Modellierung von Funktionsbausteinen für industrielle Automatisierungssysteme beschäftigt. Sie bietet ein standardisiertes Framework für die Entwicklung von verteilten Steuerungssystemen, wobei logische und arithmetische Operatoren eine zentrale Rolle spielen. Einer dieser Operatoren ist der **GE-Operator** (Greater Than or Equal To, „größer oder gleich"), der in diesem Aufsatz näher beleuchtet werden soll.

Was ist der GE-Operator?

Der GE-Operator ist ein Vergleichsoperator, der verwendet wird, um festzustellen, ob ein Wert größer oder gleich einem anderen Wert ist. In der IEC 61499 wird dieser Operator häufig in Funktionsbausteinen eingesetzt, um Bedingungen zu evaluieren und Entscheidungen basierend auf numerischen Vergleichen zu treffen. Der GE-Operator vergleicht zwei Eingangswerte und gibt einen booleschen Wert (`TRUE` oder `FALSE`) zurück, je nachdem, ob der erste Wert größer oder gleich dem zweiten Wert ist.

Syntax und Verwendung

In der IEC 61499 wird der GE-Operator in Ausdrücken von Funktionsbausteinen verwendet. Die Syntax des GE-Operators ist wie folgt:
`GE(Eingang1, Eingang2)`
Hierbei sind `Eingang1` und `Eingang2` die beiden Werte, die verglichen werden sollen. Der Rückgabewert des Operators ist `TRUE`, wenn `Eingang1` größer oder gleich `Eingang2` ist, andernfalls ist der Rückgabewert `FALSE`.

Anwendungsbeispiele

1. **Überwachung von Grenzwerten:** In industriellen Prozessen ist es oft notwendig, Grenzwerte zu überwachen. Der GE-Operator kann verwendet werden, um zu überprüfen, ob ein Messwert einen bestimmten Schwellenwert erreicht oder überschreitet:

   ```
   IF GE(Messwert, Schwellenwert) THEN
       // Alarm auslösen oder Korrekturmaßnahmen einleiten
   END_IF
   ```

2. **Steuerung von Produktionsprozessen:** In einem Produktionsprozess kann der GE-Operator verwendet werden, um zu bestimmen, ob eine bestimmte Menge an Material produziert wurde. Wenn die produzierte Menge größer oder gleich der Zielmenge ist, kann der Prozess gestoppt werden:

   ```
   IF GE(ProduzierteMenge, ZielMenge) THEN
       // Produktionsprozess stoppen
   END_IF
   ```

3. **Temperaturregelung:** In einem Heizsystem kann der GE-Operator verwendet werden, um zu überprüfen, ob die aktuelle Temperatur den Sollwert erreicht oder überschreitet. Wenn dies der Fall ist, kann die Heizung abgeschaltet werden:

   ```
   IF GE(AktuelleTemperatur, SollTemperatur) THEN
       // Heizung abschalten
   END_IF
   ```

4. **Energieverbrauchsüberwachung:** Der GE-Operator kann auch zur Überwachung des Energieverbrauchs eingesetzt werden. Wenn der Verbrauch einen bestimmten Wert überschreitet, können Maßnahmen zur Energieeinsparung eingeleitet werden:

   ```
   IF GE(EnergieVerbrauch, Grenzwert) THEN
       // Energieeinsparmaßnahmen aktivieren
   END_IF
   ```

Vorteile des GE-Operators

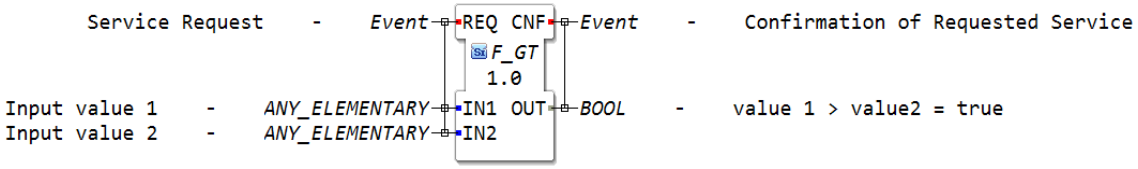
- **Einfache Logik:** Der GE-Operator ermöglicht es, komplexe Bedingungen in einfachen, leicht verständlichen Ausdrücken zu formulieren.
- **Flexibilität:** Der Operator kann auf verschiedene numerische Datentypen angewendet werden, einschließlich Ganzzahlen und Gleitkommazahlen.
- **Effizienz:** Durch die direkte Integration in Funktionsbausteine reduziert der GE-Operator den Bedarf an zusätzlicher Programmierung und macht den Code effizienter.

- **Robustheit:** Der GE-Operator trägt dazu bei, robuste Steuerungsalgorithmen zu entwickeln, die auf präzisen numerischen Vergleichen basieren.

Fazit

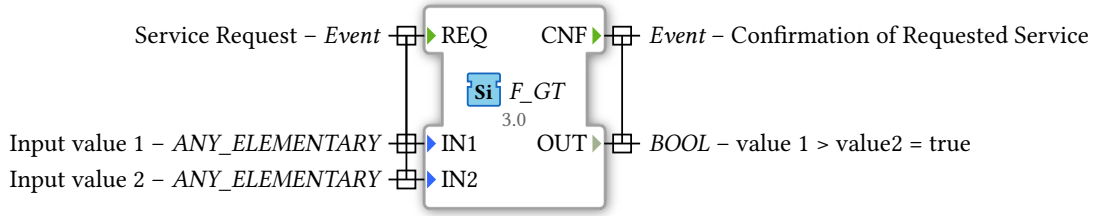
Der GE-Operator ist ein wichtiges Werkzeug in der IEC 61499, das die Implementierung von numerischen Vergleichen in industriellen Steuerungssystemen vereinfacht. Seine Fähigkeit, festzustellen, ob ein Wert größer oder gleich einem anderen Wert ist, macht ihn zu einem unverzichtbaren Bestandteil in der Prozessüberwachung, Regelung und Entscheidungsfindung. Ob in der Produktionssteuerung, Temperaturregelung oder Energieverbrauchsüberwachung – der GE-Operator ist ein leistungsstarkes und vielseitiges Element in der Welt der industriellen Automatisierung. Durch seine Einfachheit und Effizienz trägt er dazu bei, die Entwicklung von zuverlässigen und effektiven Steuerungssystemen zu unterstützen.

F_GT



https://content.helpme-codesys.com/en/CODESYS%20Development%20System/_cds_operator_gt.html

• – ◦ • – ◦ • – ◦ •



Folgender Text wurde erzeugt von https://chat.deepseek.com/ am 05.02.2025 und noch nicht korrekturgelesen (HF)

Der GT-Operator in IEC 61499: Eine Einführung

Die IEC 61499 ist eine internationale Norm, die sich mit der Modellierung von Funktionsbausteinen für industrielle Automatisierungssysteme beschäftigt. Sie bietet ein standardisiertes Framework für die Entwicklung von verteilten Steuerungssystemen, wobei logische und arithmetische Operatoren eine zentrale Rolle spielen. Einer dieser Operatoren ist der **GT-Operator** (Greater Than, „größer als“), der in diesem Aufsatz näher beleuchtet werden soll.

Schnittstelle

Ereignis-Eingänge

- `REQ`: Service-Anforderung (mit `IN1`, `IN2`)

Ereignis-Ausgänge

- `CNF`: Bestätigung der angeforderten Operation (mit `OUT`)

Daten-Eingänge

- `IN1` (ANY_ELEMENTARY): Eingangswert 1
- `IN2` (ANY_ELEMENTARY): Eingangswert 2

Daten-Ausgänge

- `OUT` (BOOL): Wert 1 > Wert 2 = TRUE

Was ist der GT-Operator?

Der GT-Operator ist ein Vergleichsoperator, der verwendet wird, um festzustellen, ob ein Wert größer als ein anderer Wert ist. In der IEC 61499 wird dieser Operator häufig in Funktionsbausteinen eingesetzt, um Bedingungen zu evaluieren und Entscheidungen basierend auf numerischen Vergleichen zu treffen. Der GT-Operator vergleicht zwei Eingangswerte und gibt einen booleschen Wert (`TRUE` oder `FALSE`) zurück, je nachdem, ob der erste Wert größer als der zweite Wert ist.

Syntax und Verwendung

In der IEC 61499 wird der GT-Operator in Ausdrücken von Funktionsbausteinen verwendet. Die Syntax des GT-Operators ist wie folgt:
`GT(Eingang1, Eingang2)`
Hierbei sind `Eingang1` und `Eingang2` die beiden Werte, die verglichen werden sollen. Der Rückgabewert des Operators ist `TRUE`, wenn `Eingang1` größer als `Eingang2` ist, andernfalls ist der Rückgabewert `FALSE`.

Anwendungsbeispiele

1. **Überwachung von Grenzwerten:** In industriellen Prozessen ist es oft notwendig, Grenzwerte zu überwachen. Der GT-Operator kann verwendet werden, um zu überprüfen, ob ein Messwert einen bestimmten Schwellenwert überschreitet:

   ```
   IF GT(Messwert, Schwellenwert) THEN
       // Alarm auslösen oder Korrekturmaßnahmen einleiten
   END_IF
   ```

2. **Steuerung von Produktionsprozessen:** In einem Produktionsprozess kann der GT-Operator verwendet werden, um zu bestimmen, ob eine bestimmte Menge an Material produziert wurde. Wenn die produzierte Menge größer als die Zielmenge ist, kann der Prozess gestoppt werden:

   ```
   IF GT(ProduzierteMenge, ZielMenge) THEN
       // Produktionsprozess stoppen
   END_IF
   ```

3. **Temperaturregelung:** In einem Heizsystem kann der GT-Operator verwendet werden, um zu überprüfen, ob die aktuelle Temperatur den Sollwert überschreitet. Wenn dies der Fall ist, kann die Heizung abgeschaltet werden:

   ```
   IF GT(AktuelleTemperatur, SollTemperatur) THEN
       // Heizung abschalten
   END_IF
   ```

4. **Energieverbrauchsüberwachung:** Der GT-Operator kann auch zur Überwachung des Energieverbrauchs eingesetzt werden. Wenn der Verbrauch einen bestimmten Wert überschreitet, können Maßnahmen zur Energieeinsparung eingeleitet werden:

   ```
   IF GT(EnergieVerbrauch, Grenzwert) THEN
       // Energieeinsparmaßnahmen aktivieren
   END_IF
   ```

Vorteile des GT-Operators

- **Einfache Logik:** Der GT-Operator ermöglicht es, komplexe Bedingungen in einfachen, leicht verständlichen Ausdrücken zu formulieren.
- **Flexibilität:** Der Operator kann auf verschiedene numerische Datentypen angewendet werden, einschließlich Ganzzahlen und Gleitkommazahlen.
- **Effizienz:** Durch die direkte Integration in Funktionsbausteine reduziert der GT-Operator den Bedarf an zusätzlicher Programmierung und macht den Code effizienter.
- **Robustheit:** Der GT-Operator trägt dazu bei, robuste Steuerungsalgorithmen zu entwickeln, die auf präzisen numerischen Vergleichen basieren.

🛠️ Zugehörige Übungen

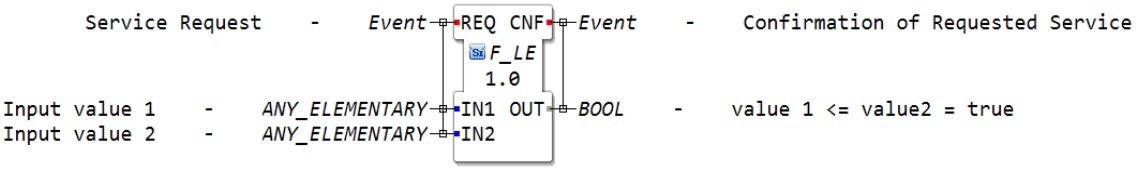
- Uebung_060
- Uebung_071
- Uebung_071a
- Uebung_110
- Uebung_111

Fazit

Der GT-Operator ist ein wichtiges Werkzeug in der IEC 61499, das die Implementierung von numerischen Vergleichen in industriellen Steuerungssystemen vereinfacht. Seine Fähigkeit, festzustellen, ob ein Wert größer als ein anderer Wert ist, macht ihn zu einem unverzichtbaren Bestandteil in der Prozessüberwachung, Regelung und Entscheidungsfindung. Ob in der Produktionssteuerung, Temperaturregelung oder Energieverbrauchsüberwachung – der GT-Operator ist ein leistungsstarkes und vielseitiges Element in

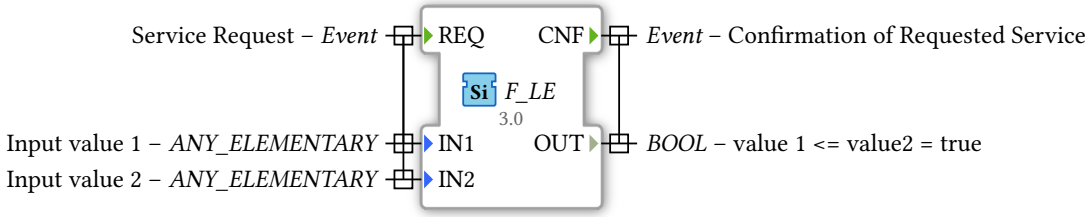
der Welt der industriellen Automatisierung. Durch seine Einfachheit und Effizienz trägt er dazu bei, die Entwicklung von zuverlässigen und effektiven Steuerungssystemen zu unterstützen.

F_LE



https://content.helpme-codesys.com/de/CODESYS%20Development%20System/_cds_operator_le.html
https://content.helpme-codesys.com/en/CODESYS%20Development%20System/_cds_operator_le.html

• – ◦ • – ◦ • – ◦ •



Schnittstelle

Ereignis-Eingänge

- `REQ`: Dienstanforderung, trägt `IN1` und `IN2`.

Ereignis-Ausgänge

- `CNF`: Bestätigung des angeforderten Dienstes, trägt `OUT`.

Daten-Eingänge

- `IN1` (`ANY_ELEMENTARY`): Eingangswert 1.
- `IN2` (`ANY_ELEMENTARY`): Eingangswert 2.

Daten-Ausgänge

- `OUT` (`BOOL`): `TRUE`, wenn Wert 1 <= Wert 2.

Folgender Text wurde erzeugt von https://chat.deepseek.com/ am 05.02.2025 und noch nicht korrekturgelesen (HF)

Der LE-Operator in IEC 61499: Eine Einführung

Die IEC 61499 ist eine internationale Norm, die sich mit der Modellierung von Funktionsbausteinen für industrielle Automatisierungssysteme beschäftigt. Sie bietet ein standardisiertes Framework für die Entwicklung von verteilten Steuerungssystemen, wobei logische und arithmetische Operatoren eine zentrale Rolle spielen. Einer dieser Operatoren ist der **LE-Operator** (Less Than or Equal To, „kleiner oder gleich“), der in diesem Aufsatz näher beleuchtet werden soll.

Was ist der LE-Operator?

Der LE-Operator ist ein Vergleichsoperator, der verwendet wird, um festzustellen, ob ein Wert kleiner oder gleich einem anderen Wert ist. In der IEC 61499 wird dieser Operator häufig in Funktionsbausteinen eingesetzt, um Bedingungen zu evaluieren und Entscheidungen basierend auf numerischen Vergleichen zu treffen. Der LE-Operator vergleicht zwei Eingangswerte und gibt einen booleschen Wert (`TRUE` oder `FALSE`) zurück, je nachdem, ob der erste Wert kleiner oder gleich dem zweiten Wert ist.

Syntax und Verwendung

In der IEC 61499 wird der LE-Operator in Ausdrücken von Funktionsbausteinen verwendet. Die Syntax des LE-Operators ist wie folgt:

```
LE(Eingang1, Eingang2)
```

Hierbei sind `Eingang1` und `Eingang2` die beiden Werte, die verglichen werden sollen. Der Rückgabewert des Operators ist `TRUE`, wenn `Eingang1` kleiner oder gleich `Eingang2` ist, andernfalls ist der Rückgabewert `FALSE`.

Anwendungsbeispiele

1. **Überwachung von Grenzwerten:** In industriellen Prozessen ist es oft notwendig, Grenzwerte zu überwachen. Der LE-Operator kann verwendet werden, um zu überprüfen, ob ein Messwert einen bestimmten Schwellenwert unterschreitet oder gleich diesem ist:

```
IF LE(Messwert, Schwellenwert) THEN
    // Aktion ausführen, wenn der Messwert den Schwellenwert nicht überschreitet
END_IF
```

2. **Steuerung von Produktionsprozessen:** In einem Produktionsprozess kann der LE-Operator verwendet werden, um zu bestimmen, ob eine bestimmte Menge an Material produziert wurde. Wenn die produzierte Menge kleiner oder gleich der Zielmenge ist, kann der Prozess fortgesetzt werden:

```
IF LE(ProduzierteMenge, ZielMenge) THEN
    // Produktionsprozess fortsetzen
END_IF
```

3. **Temperaturregelung:** In einem Kühlsystem kann der LE-Operator verwendet werden, um zu überprüfen, ob die aktuelle Temperatur kleiner oder gleich der Solltemperatur ist. Wenn dies der Fall ist, kann die Kühlung abgeschaltet werden:

```
IF LE(AktuelleTemperatur, SollTemperatur) THEN
    // Kühlung abschalten
END_IF
```

4. **Energieverbrauchsüberwachung:** Der LE-Operator kann auch zur Überwachung des Energieverbrauchs eingesetzt werden. Wenn der Verbrauch kleiner oder gleich einem bestimmten Wert ist, können Maßnahmen zur Energieeinsparung deaktiviert werden:

```
IF LE(EnergieVerbrauch, Grenzwert) THEN
    // Energieeinsparmaßnahmen deaktivieren
END_IF
```

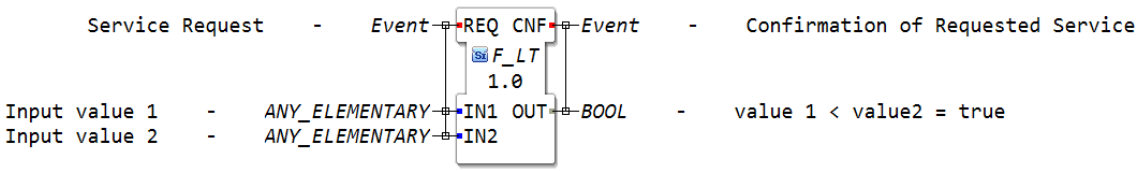
Vorteile des LE-Operators

- **Einfache Logik:** Der LE-Operator ermöglicht es, komplexe Bedingungen in einfachen, leicht verständlichen Ausdrücken zu formulieren.
- **Flexibilität:** Der Operator kann auf verschiedene numerische Datentypen angewendet werden, einschließlich Ganzzahlen und Gleitkommazahlen.
- **Effizienz:** Durch die direkte Integration in Funktionsbausteine reduziert der LE-Operator den Bedarf an zusätzlicher Programmierung und macht den Code effizienter.
- **Robustheit:** Der LE-Operator trägt dazu bei, robuste Steuerungsalgorithmen zu entwickeln, die auf präzisen numerischen Vergleichen basieren.

Fazit

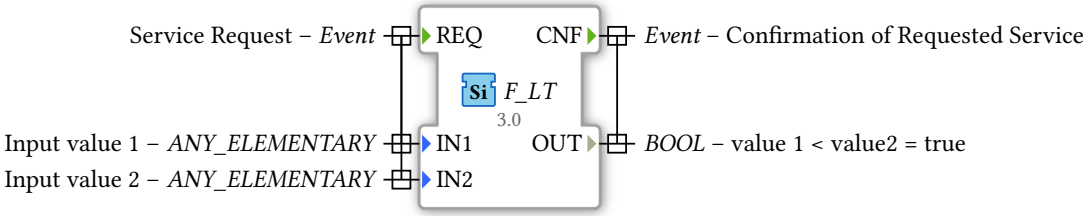
Der LE-Operator ist ein wichtiges Werkzeug in der IEC 61499, das die Implementierung von numerischen Vergleichen in industriellen Steuerungssystemen vereinfacht. Seine Fähigkeit, festzustellen, ob ein Wert kleiner oder gleich einem anderen Wert ist, macht ihn zu einem unverzichtbaren Bestandteil in der Prozessüberwachung, Regelung und Entscheidungsfindung. Ob in der Produktionssteuerung, Temperaturregelung oder Energieverbrauchsüberwachung – der LE-Operator ist ein leistungsstarkes und vielseitiges Element in der Welt der industriellen Automatisierung. Durch seine Einfachheit und Effizienz trägt er dazu bei, die Entwicklung von zuverlässigen und effektiven Steuerungssystemen zu unterstützen.

F_LT



https://content.helpme-codesys.com/de/CODESYS%20Development%20System/_cds_operator_lt.html

• – ◦ • – ◦ • – ◦ •



Schnittstelle

Port	Typ	Beschreibung
REQ	Event	Service-Anforderung (mit IN1, IN2)
CNF	Event	Bestätigung der angeforderten Service-Operation (mit OUT)
IN1	ANY_ELEMENTARY	Eingangswert 1
IN2	ANY_ELEMENTARY	Eingangswert 2
OUT	BOOL	TRUE, wenn IN1 kleiner als IN2

Folgender Text wurde erzeugt von https://chat.deepseek.com/ am 05.02.2025 und noch nicht korrekturgelesen (HF)

Der LT-Operator in IEC 61499: Eine Einführung

Die IEC 61499 ist eine internationale Norm, die sich mit der Modellierung von Funktionsbausteinen für industrielle Automatisierungssysteme beschäftigt. Sie bietet ein standardisiertes Framework für die Entwicklung von verteilten Steuerungssystemen, wobei logische und arithmetische Operatoren eine zentrale Rolle spielen. Einer dieser Operatoren ist der **LT-Operator** (Less Than, „kleiner als“), der in diesem Aufsatz näher beleuchtet werden soll.

Was ist der LT-Operator?

Der LT-Operator ist ein Vergleichsoperator, der verwendet wird, um festzustellen, ob ein Wert kleiner als ein anderer Wert ist. In der IEC 61499 wird dieser Operator häufig in Funktionsbausteinen eingesetzt, um Bedingungen zu evaluieren und Entscheidungen basierend auf numerischen Vergleichen zu treffen. Der LT-Operator vergleicht zwei Eingangswerte und gibt einen booleschen Wert (`TRUE` oder `FALSE`) zurück, je nachdem, ob der erste Wert kleiner als der zweite Wert ist.

Syntax und Verwendung

In der IEC 61499 wird der LT-Operator in Ausdrücken von Funktionsbausteinen verwendet. Die Syntax des LT-Operators ist wie folgt:

```
LT(Eingang1, Eingang2)
```

Hierbei sind `Eingang1` und `Eingang2` die beiden Werte, die verglichen werden sollen. Der Rückgabewert des Operators ist `TRUE`, wenn `Eingang1` kleiner als `Eingang2` ist, andernfalls ist der Rückgabewert `FALSE`.

Anwendungsbeispiele

1. **Überwachung von Grenzwerten:** In industriellen Prozessen ist es oft notwendig, Grenzwerte zu überwachen. Der LT-Operator kann verwendet werden, um zu überprüfen, ob ein Messwert einen bestimmten Schwellenwert unterschreitet:

   ```
   IF LT(Messwert, Schwellenwert) THEN
       // Aktion ausführen, wenn der Messwert den Schwellenwert unterschreitet
   END_IF
   ```

2. **Steuerung von Produktionsprozessen:** In einem Produktionsprozess kann der LT-Operator verwendet werden, um zu bestimmen, ob eine bestimmte Menge an Material produziert wurde. Wenn die produzierte Menge kleiner als die Zielmenge ist, kann der Prozess fortgesetzt werden:

   ```
   IF LT(ProduzierteMenge, ZielMenge) THEN
       // Produktionsprozess fortsetzen
   END_IF
   ```

3. **Temperaturregelung:** In einem Kühlsystem kann der LT-Operator verwendet werden, um zu überprüfen, ob die aktuelle Temperatur kleiner als die Solltemperatur ist. Wenn dies der Fall ist, kann die Kühlung weiterhin aktiviert bleiben:

```
IF LT(AktuelleTemperatur, SollTemperatur) THEN
    // Kühlung weiter aktivieren
END_IF
```

4. **Energieverbrauchsüberwachung:** Der LT-Operator kann auch zur Überwachung des Energieverbrauchs eingesetzt werden. Wenn der Verbrauch kleiner als ein bestimmter Wert ist, können Maßnahmen zur Energieeinsparung deaktiviert werden:

```
IF LT(EnergieVerbrauch, Grenzwert) THEN
    // Energieeinsparmaßnahmen deaktivieren
END_IF
```

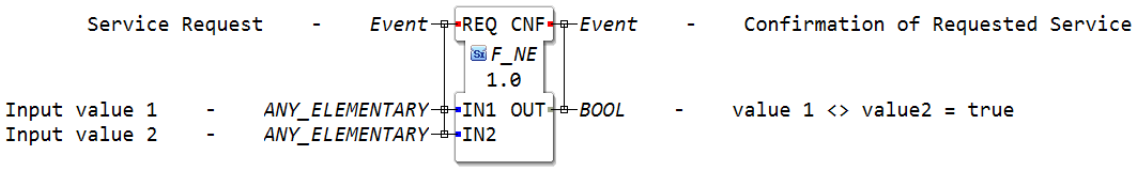
Vorteile des LT-Operators

- **Einfache Logik:** Der LT-Operator ermöglicht es, komplexe Bedingungen in einfachen, leicht verständlichen Ausdrücken zu formulieren.
- **Flexibilität:** Der Operator kann auf verschiedene numerische Datentypen angewendet werden, einschließlich Ganzzahlen und Gleitkommazahlen.
- **Effizienz:** Durch die direkte Integration in Funktionsbausteine reduziert der LT-Operator den Bedarf an zusätzlicher Programmierung und macht den Code effizienter.
- **Robustheit:** Der LT-Operator trägt dazu bei, robuste Steuerungsalgorithmen zu entwickeln, die auf präzisen numerischen Vergleichen basieren.

Fazit

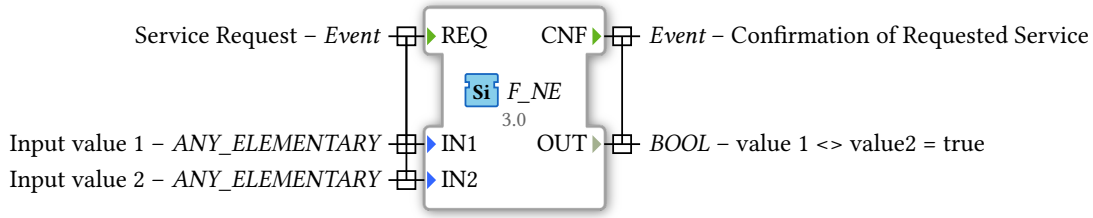
Der LT-Operator ist ein wichtiges Werkzeug in der IEC 61499, das die Implementierung von numerischen Vergleichen in industriellen Steuerungssystemen vereinfacht. Seine Fähigkeit, festzustellen, ob ein Wert kleiner als ein anderer Wert ist, macht ihn zu einem unverzichtbaren Bestandteil in der Prozessüberwachung, Regelung und Entscheidungsfindung. Ob in der Produktionssteuerung, Temperaturregelung oder Energieverbrauchsüberwachung – der LT-Operator ist ein leistungsstarkes und vielseitiges Element in der Welt der industriellen Automatisierung. Durch seine Einfachheit und Effizienz trägt er dazu bei, die Entwicklung von zuverlässigen und effektiven Steuerungssystemen zu unterstützen.

F_NE



https://content.helpme-codesys.com/de/CODESYS%20Development%20System/_cds_operator_ne.html https://content.helpme-codesys.com/en/CODESYS%20Development%20System/_cds_operator_ne.html

• – ◦ • – ◦ • – ◦ •



Folgender Text wurde erzeugt von https://chat.deepseek.com/ am 05.02.2025 und noch nicht korrekturgelesen (HF)

Der NE-Operator in IEC 61499: Eine Einführung

Die IEC 61499 ist eine internationale Norm, die sich mit der Modellierung von Funktionsbausteinen für industrielle Automatisierungssysteme beschäftigt. Sie bietet ein standardisiertes Framework für die Entwicklung von verteilten Steuerungssystemen, wobei logische und arithmetische Operatoren eine

zentrale Rolle spielen. Einer dieser Operatoren ist der **NE-Operator** (Not Equal, „ungleich“), der in diesem Aufsatz näher beleuchtet werden soll.

Schnittstelle

Ereignis-Eingänge

- `REQ`: Service-Anforderung (mit `IN1`, `IN2`)

Ereignis-Ausgänge

- `CNF`: Bestätigung der angeforderten Operation (mit `OUT`)

Daten-Eingänge

- `IN1` (ANY_ELEMENTARY): Eingangswert 1
- `IN2` (ANY_ELEMENTARY): Eingangswert 2

Daten-Ausgänge

- `OUT` (BOOL): Wert 1 <> Wert 2 = TRUE

Was ist der NE-Operator?

Der NE-Operator ist ein Vergleichsoperator, der verwendet wird, um festzustellen, ob zwei Werte ungleich sind. In der IEC 61499 wird dieser Operator häufig in Funktionsbausteinen eingesetzt, um Bedingungen zu evaluieren und Entscheidungen basierend auf Vergleichen zu treffen. Der NE-Operator vergleicht zwei Eingangswerte und gibt einen booleschen Wert (`TRUE` oder `FALSE`) zurück, je nachdem, ob die Werte ungleich sind.

Syntax und Verwendung

In der IEC 61499 wird der NE-Operator in Ausdrücken von Funktionsbausteinen verwendet. Die Syntax des NE-Operators ist wie folgt:
`NE(Eingang1, Eingang2)`
Hierbei sind `Eingang1` und `Eingang2` die beiden Werte, die verglichen werden sollen. Der Rückgabewert des Operators ist `TRUE`, wenn `Eingang1` ungleich `Eingang2` ist, andernfalls ist der Rückgabewert `FALSE`.

Anwendungsbeispiele

1. **Überwachung von Sensorwerten:** In industriellen Prozessen ist es oft notwendig, die Werte von Sensoren zu überwachen. Der NE-Operator kann verwendet werden, um zu überprüfen, ob zwei Sensoren unterschiedliche Werte liefern:

   ```
   IF NE(Sensor1, Sensor2) THEN
       // Alarm auslösen oder Korrekturmaßnahmen einleiten
   END_IF
   ```

2. **Steuerung von Produktionsprozessen:** In einem Produktionsprozess kann der NE-Operator verwendet werden, um zu bestimmen, ob ein bestimmter Prozessschritt noch nicht abgeschlossen ist. Wenn der aktuelle Schritt ungleich dem Zielschritt ist, kann der Prozess fortgesetzt werden:

   ```
   IF NE(AktuellerSchritt, ZielSchritt) THEN
       // Prozess fortsetzen
   END_IF
   ```

3. **Temperaturregelung:** In einem Heizsystem kann der NE-Operator verwendet werden, um zu überprüfen, ob die aktuelle Temperatur ungleich der Solltemperatur ist. Wenn dies der Fall ist, kann die Heizung weiterhin aktiviert bleiben:

   ```
   IF NE(AktuelleTemperatur, SollTemperatur) THEN
       // Heizung weiter aktivieren
   END_IF
   ```

4. **Fehlererkennung:** Der NE-Operator kann auch zur Fehlererkennung eingesetzt werden. Wenn beispielsweise ein Sollwert und ein Istwert verglichen werden und diese nicht übereinstimmen, kann ein Fehlersignal generiert werden:

   ```
   IF NE(Sollwert, Istwert) THEN
       // Fehlersignal generieren
   END_IF
   ```

Vorteile des NE-Operators

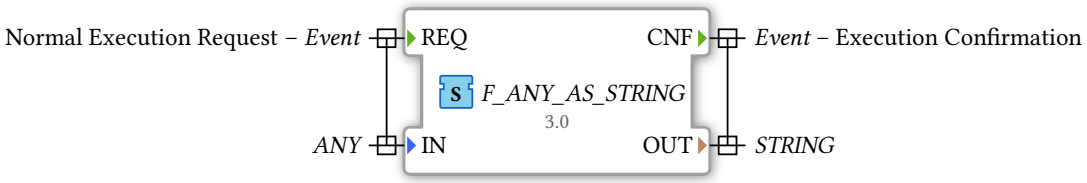
- **Einfache Logik:** Der NE-Operator ermöglicht es, komplexe Bedingungen in einfachen, leicht verständlichen Ausdrücken zu formulieren.
- **Flexibilität:** Der Operator kann auf verschiedene Datentypen angewendet werden, einschließlich Ganzzahlen, Gleitkommazahlen und sogar boolesche Werte.
- **Effizienz:** Durch die direkte Integration in Funktionsbausteine reduziert der NE-Operator den Bedarf an zusätzlicher Programmierung und macht den Code effizienter.
- **Robustheit:** Der NE-Operator trägt dazu bei, robuste Steuerungsalgorithmen zu entwickeln, die auf präzisen Vergleichen basieren.

Fazit

Der NE-Operator ist ein wichtiges Werkzeug in der IEC 61499, das die Implementierung von Vergleichen in industriellen Steuerungssystemen vereinfacht. Seine Fähigkeit, festzustellen, ob zwei Werte ungleich sind, macht ihn zu einem unverzichtbaren Bestandteil in der Prozessüberwachung, Regelung und Entscheidungsfindung. Ob in der Produktionssteuerung, Temperaturregelung oder Fehlererkennung – der NE-Operator ist ein leistungsstarkes und vielseitiges Element in der Welt der industriellen Automatisierung. Durch seine Einfachheit und Effizienz trägt er dazu bei, die Entwicklung von zuverlässigen und effektiven Steuerungssystemen zu unterstützen.

2.4.1.7 conversion

F_ANY_AS_STRING



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `F_ANY_AS_STRING` wandelt einen Wert beliebigen Datentyps (`ANY`) in seine textuelle Darstellung als `STRING` um. Er dient als generischer „Zu-Text"-Konverter, etwa für Diagnose- oder Logging-Zwecke, ohne für jeden Datentyp eine eigene, spezifische Konvertierungsfunktion aufrufen zu müssen.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Umwandlung aus, trägt `IN`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT`.

Daten-Eingänge

- **IN** (ANY): Der umzuwandelnde Wert, beliebiger Datentyp.

Daten-Ausgänge

- **OUT** (STRING): Die textuelle Darstellung von `IN`.

Funktionsweise

Bei Eintreffen von `REQ` erzeugt der Baustein aus dem generisch typisierten Eingang `IN` eine textuelle Darstellung und gibt sie über `OUT` aus, anschließend wird `CNF` ausgelöst. Die genaue Formatierung folgt der jeweiligen IEC-61131-3-Literal-Notation des zugewiesenen konkreten Datentyps.

Technische Besonderheiten

- **Generischer `ANY`-Eingang**: Der Baustein akzeptiert jeden IEC-61131-3-Datentyp, der zur Übersetzungszeit an `IN` gebunden wird.

- **Universeller Debug-/Logging-Baustein**: Da praktisch jeder Datentyp als `STRING` ausgegeben werden kann, eignet sich der Baustein besonders für generische Diagnoseausgaben.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur Umwandlung und zu `CNF`.

Anwendungsszenarien

- **Diagnose und Logging**: Ausgabe beliebiger Prozesswerte als lesbaren Text, z. B. für Log-Dateien oder Diagnoseanzeigen.
- **Generische Anzeigebausteine**, die unabhängig vom tatsächlichen Datentyp des angeschlossenen Werts eine textuelle Darstellung benötigen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **Typspezifische `*_TO_STRING`-Funktionen** (z. B. `DINT_TO_STRING`): erfordern für jeden Datentyp eine eigene Instanz, bieten aber ggf. feingranularere Formatierungskontrolle.

Fazit

`F_ANY_AS_STRING` liefert eine bequeme, generisch typisierte Text-Konvertierung für beliebige Datentypen und eignet sich besonders für Diagnose- und Logging-Anwendungen, bei denen der konkrete Datentyp variieren kann.

F_FROM_BIG_ENDIAN

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `F_FROM_BIG_ENDIAN` konvertiert einen Wert aus der Big-Endian (höchstwertiges Byte zuerst)-Byte-Reihenfolge. Er gehört zur Familie der Endian-Konvertierungsfunktionen, die für den plattformübergreifenden Austausch binärer Daten (z. B. über Netzwerkprotokolle oder mit Fremdsystemen) benötigt werden.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Konvertierung aus, trägt `IN`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT`.

Daten-Eingänge

- **IN** (ANY): Der zu konvertierende Wert, beliebiger Datentyp.

Daten-Ausgänge

- **OUT** (ANY): Der konvertierte Wert, gleicher Datentyp wie `IN`.

Funktionsweise

Bei Eintreffen von `REQ` interpretiert die Bytefolge von `IN` als Big-Endian (höchstwertiges Byte zuerst)-kodierten Wert und wandelt sie in die native Byte-Reihenfolge der Zielplattform um. Das Ergebnis wird über `OUT` ausgegeben, anschließend wird `CNF` ausgelöst. Die Bitbreite des Datentyps bleibt dabei unverändert — es wird ausschließlich die Byte-Reihenfolge angepasst.

Technische Besonderheiten

- **Generischer `ANY`-Datentyp**: `IN`/`OUT` passen sich an den zur Übersetzungszeit angeschlossenen konkreten Datentyp an; die Byte-Anzahl ergibt sich aus dessen Größe.
- **Reine Byte-Reihenfolge, keine Wertinterpretation**: Der Baustein ändert nur die Anordnung der Bytes, nicht deren Bedeutung oder Vorzeichen.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur Konvertierung und zu `CNF`.

Anwendungsszenarien

- **Netzwerkkommunikation**: Viele Netzwerkprotokolle (z. B. TCP/IP-Header) verwenden Big-Endian („Network Byte Order“); `F_FROM_BIG_ENDIAN` stellt die korrekte Byte-Reihenfolge beim Senden/Empfangen sicher.
- **Interoperabilität mit Fremdsystemen**: Austausch von Binärdaten mit Systemen, die eine andere native Byte-Reihenfolge verwenden als die Zielplattform.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`F_FROM_LITTLE_ENDIAN`**: dieselbe Konvertierungsrichtung (aus), aber für die jeweils andere Byte-Reihenfolge.
- **`F_TO_BIG_ENDIAN`**: die Gegenrichtung für dieselbe Byte-Reihenfolge.

Fazit

`F_FROM_BIG_ENDIAN` liefert eine einfache, generisch typisierte Byte-Reihenfolgen-Konvertierung und ist ein wichtiger Baustein für die korrekte Interoperabilität mit Netzwerkprotokollen und Fremdsystemen unterschiedlicher Endianness.

F_FROM_LITTLE_ENDIAN

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `F_FROM_LITTLE_ENDIAN` konvertiert einen Wert aus der Little-Endian (niedrigstwertiges Byte zuerst)-Byte-Reihenfolge. Er gehört zur Familie der Endian-Konvertierungsfunktionen, die für den plattformübergreifenden Austausch binärer Daten (z. B. über Netzwerkprotokolle oder mit Fremdsystemen) benötigt werden.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Konvertierung aus, trägt `IN`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT`.

Daten-Eingänge

- **IN** (ANY): Der zu konvertierende Wert, beliebiger Datentyp.

Daten-Ausgänge

- **OUT** (ANY): Der konvertierte Wert, gleicher Datentyp wie `IN`.

Funktionsweise

Bei Eintreffen von `REQ` interpretiert die Bytefolge von `IN` als Little-Endian (niedrigstwertiges Byte zuerst)-kodierten Wert und wandelt sie in die native Byte-Reihenfolge der Zielplattform um. Das Ergebnis wird

über `OUT` ausgegeben, anschließend wird `CNF` ausgelöst. Die Bitbreite des Datentyps bleibt dabei unverändert — es wird ausschließlich die Byte-Reihenfolge angepasst.

Technische Besonderheiten

- **Generischer `ANY`-Datentyp**: `IN`/`OUT` passen sich an den zur Übersetzungszeit angeschlossenen konkreten Datentyp an; die Byte-Anzahl ergibt sich aus dessen Größe.
- **Reine Byte-Reihenfolge, keine Wertinterpretation**: Der Baustein ändert nur die Anordnung der Bytes, nicht deren Bedeutung oder Vorzeichen.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur Konvertierung und zu `CNF`.

Anwendungsszenarien

- **Netzwerkkommunikation**: Viele Netzwerkprotokolle (z. B. TCP/IP-Header) verwenden Big-Endian („Network Byte Order“); `F_FROM_LITTLE_ENDIAN` stellt die korrekte Byte-Reihenfolge beim Senden/Empfangen sicher.
- **Interoperabilität mit Fremdsystemen**: Austausch von Binärdaten mit Systemen, die eine andere native Byte-Reihenfolge verwenden als die Zielplattform.

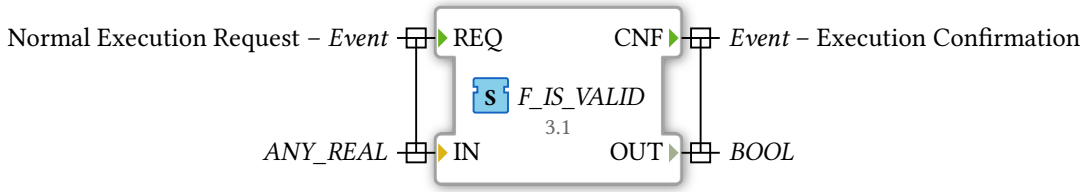
⚖️ Vergleich mit ähnlichen Bausteinen

- **`F_FROM_BIG_ENDIAN`**: dieselbe Konvertierungsrichtung (aus), aber für die jeweils andere Byte-Reihenfolge.
- **`F_TO_LITTLE_ENDIAN`**: die Gegenrichtung für dieselbe Byte-Reihenfolge.

Fazit

`F_FROM_LITTLE_ENDIAN` liefert eine einfache, generisch typisierte Byte-Reihenfolgen-Konvertierung und ist ein wichtiger Baustein für die korrekte Interoperabilität mit Netzwerkprotokollen und Fremdsystemen unterschiedlicher Endianness.

F_IS_VALID



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `F_IS_VALID` prüft, ob ein Gleitkommawert (`REAL`/`LREAL`) ein gültiger, endlicher Wert ist — also weder `NaN` (Not a Number) noch `+/-Infinity`. Er dient der Absicherung numerischer Berechnungen gegen ungültige Zwischenergebnisse.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Prüfung aus, trägt `IN`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT`.

Daten-Eingänge

- **IN** (ANY_REAL): Der zu prüfende Gleitkommawert.

Daten-Ausgänge

- **OUT** (BOOL): `TRUE`, wenn `IN` ein gültiger, endlicher Wert ist; `FALSE` bei `NaN` oder `Infinity`.

Funktionsweise

Bei Eintreffen von `REQ` prüft die Algorithmus-ST-Anweisung `OUT := IS_VALID(IN)`, ob `IN` weder `NaN` noch `+/-Infinity` ist, und gibt das Ergebnis über `OUT` aus. Anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **`ANY_REAL`-Eingang**: Akzeptiert sowohl `REAL` als auch `LREAL`.
- **Erkennt typische Fehlerresultate**: `NaN` entsteht z. B. bei `0.0/0.0`, `Infinity` bei Division durch `0.0` mit von-Null-verschiedenem Zähler — beides klassische Fehlerquellen in Gleitkommaberechnungen.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur Prüfung und zu `CNF`.

Anwendungsszenarien

- **Absicherung von Berechnungsketten**: Vor der Weiterverarbeitung eines Gleitkommaergebnisses prüfen, ob es gültig ist, um Folgefehler (z. B. `NaN`-Propagation) zu vermeiden.
- **Plausibilitätsprüfung von Sensordaten**: Erkennung fehlerhafter oder aus dem gültigen Bereich gefallener Messwerte.

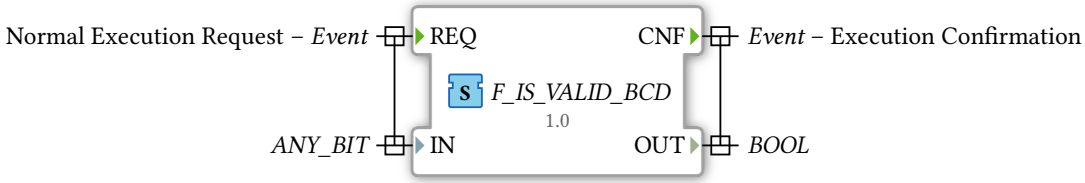
⚖️ Vergleich mit ähnlichen Bausteinen

- F_IS_VALID_BCD: dieselbe Grundidee der Gültigkeitsprüfung, jedoch für BCD-kodierte Bitfolgen statt Gleitkommazahlen.

Fazit

`F_IS_VALID` liefert eine einfache, zentrale Gültigkeitsprüfung für Gleitkommawerte und hilft, `NaN`/`Infinity`-bedingte Folgefehler in Berechnungsketten frühzeitig zu erkennen.

F_IS_VALID_BCD



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `F_IS_VALID_BCD` prüft, ob eine Bitfolge gültige BCD-Daten (Binary Coded Decimal) enthält — also ob jede Nibble (4-Bit-Gruppe) einen Wert zwischen `0` und `9` darstellt. Er dient der Validierung von BCD-kodierten Werten vor deren Konvertierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Prüfung aus, trägt `IN`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT`.

Daten-Eingänge

- **IN** (ANY_BIT): Die zu prüfende Bitfolge (`BYTE`, `WORD`, `DWORD` oder `LWORD`).

Daten-Ausgänge

- **OUT** (BOOL): `TRUE`, wenn jede Nibble von `IN` einen gültigen BCD-Wert (`0–9`) enthält; `FALSE`, wenn mindestens eine Nibble einen ungültigen Wert (`10–15`, entspricht `A–F`) enthält.

Funktionsweise

Bei Eintreffen von `REQ` prüft die Algorithmus-ST-Anweisung `OUT := IS_VALID_BCD(IN)` nibble-weise, ob alle 4-Bit-Gruppen von `IN` gültige Dezimalziffern (`0`–`9`) darstellen, und gibt das Ergebnis über `OUT` aus. Anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **`ANY_BIT`-Eingang**: Akzeptiert Bitfolgen beliebiger Breite (`BYTE`, `WORD`, `DWORD`, `LWORD`), die Anzahl der geprüften Nibbles ergibt sich aus deren Größe.
- **Vorabprüfung für BCD-Konvertierung**: Sinnvolle Ergänzung zu den `*_BCD_TO_*`-Konvertierungsbausteinen (vgl. `F_BYTE_BCD_TO_USINT` etc.), um ungültige Eingabedaten vor der eigentlichen Konvertierung abzufangen.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur Prüfung und zu `CNF`.

Anwendungsszenarien

- **Validierung von Feldbus-/Protokolldaten**, die BCD-kodiert übertragen werden, bevor sie in einen numerischen Wert umgewandelt werden.
- **Fehlererkennung** bei fehlerhaft übertragenen oder falsch interpretierten Bitfolgen, die als BCD interpretiert werden sollen.

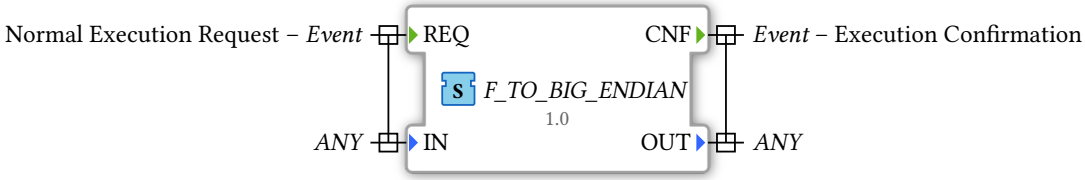
⚖️ Vergleich mit ähnlichen Bausteinen

- F_IS_VALID: dieselbe Grundidee der Gültigkeitsprüfung, jedoch für Gleitkommazahlen statt BCD-Bitfolgen.
- **`F_BYTE_BCD_TO_USINT`** und verwandte Konvertierungsbausteine: nutzen implizit dieselbe BCD-Gültigkeitsannahme; `F_IS_VALID_BCD` macht die Prüfung explizit und vorgelagert nutzbar.

Fazit

`F_IS_VALID_BCD` liefert eine einfache Vorabprüfung für BCD-kodierte Bitfolgen und schützt nachgeschaltete BCD-Konvertierungsbausteine vor ungültigen Eingabedaten.

F_TO_BIG_ENDIAN



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `F_TO_BIG_ENDIAN` konvertiert einen Wert in der Big-Endian (höchstwertiges Byte zuerst)-Byte-Reihenfolge. Er gehört zur Familie der Endian-Konvertierungsfunktionen, die für den plattformübergreifenden Austausch binärer Daten (z. B. über Netzwerkprotokolle oder mit Fremdsystemen) benötigt werden.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Konvertierung aus, trägt `IN`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT`.

Daten-Eingänge

- **IN** (ANY): Der zu konvertierende Wert, beliebiger Datentyp.

Daten-Ausgänge

- **OUT** (ANY): Der konvertierte Wert, gleicher Datentyp wie `IN`.

Funktionsweise

Bei Eintreffen von `REQ` wandelt den nativ kodierten Wert `IN` in eine Big-Endian (höchstwertiges Byte zuerst)-Bytefolge um. Das Ergebnis wird über `OUT` ausgegeben, anschließend wird `CNF` ausgelöst. Die Bitbreite des Datentyps bleibt dabei unverändert — es wird ausschließlich die Byte-Reihenfolge angepasst.

Technische Besonderheiten

- **Generischer `ANY`-Datentyp**: `IN`/`OUT` passen sich an den zur Übersetzungszeit angeschlossenen konkreten Datentyp an; die Byte-Anzahl ergibt sich aus dessen Größe.
- **Reine Byte-Reihenfolge, keine Wertinterpretation**: Der Baustein ändert nur die Anordnung der Bytes, nicht deren Bedeutung oder Vorzeichen.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur Konvertierung und zu `CNF`.

Anwendungsszenarien

- **Netzwerkkommunikation**: Viele Netzwerkprotokolle (z. B. TCP/IP-Header) verwenden Big-Endian („Network Byte Order“); `F_TO_BIG_ENDIAN` stellt die korrekte Byte-Reihenfolge beim Senden/Empfangen sicher.
- **Interoperabilität mit Fremdsystemen**: Austausch von Binärdaten mit Systemen, die eine andere native Byte-Reihenfolge verwenden als die Zielplattform.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`F_TO_LITTLE_ENDIAN`**: dieselbe Konvertierungsrichtung (in), aber für die jeweils andere Byte-Reihenfolge.
- **`F_FROM_BIG_ENDIAN`**: die Gegenrichtung für dieselbe Byte-Reihenfolge.

Fazit

`F_TO_BIG_ENDIAN` liefert eine einfache, generisch typisierte Byte-Reihenfolgen-Konvertierung und ist ein wichtiger Baustein für die korrekte Interoperabilität mit Netzwerkprotokollen und Fremdsystemen unterschiedlicher Endianness.

F_TO_LITTLE_ENDIAN

• – ◦ • – ◦ • – ◦ •

Einleitung

Der `F_TO_LITTLE_ENDIAN` konvertiert einen Wert in der Little-Endian (niedrigstwertiges Byte zuerst)-Byte-Reihenfolge. Er gehört zur Familie der Endian-Konvertierungsfunktionen, die für den plattformübergreifenden Austausch binärer Daten (z. B. über Netzwerkprotokolle oder mit Fremdsystemen) benötigt werden.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Konvertierung aus, trägt `IN`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT`.

Daten-Eingänge

- **IN** (ANY): Der zu konvertierende Wert, beliebiger Datentyp.

Daten-Ausgänge

- **OUT** (ANY): Der konvertierte Wert, gleicher Datentyp wie `IN`.

Funktionsweise

Bei Eintreffen von `REQ` wandelt den nativ kodierten Wert `IN` in eine Little-Endian (niedrigstwertiges Byte zuerst)-Bytefolge um. Das Ergebnis wird über `OUT` ausgegeben, anschließend wird `CNF` ausgelöst. Die Bitbreite des Datentyps bleibt dabei unverändert — es wird ausschließlich die Byte-Reihenfolge angepasst.

Technische Besonderheiten

- **Generischer `ANY`-Datentyp**: `IN`/`OUT` passen sich an den zur Übersetzungszeit angeschlossenen konkreten Datentyp an; die Byte-Anzahl ergibt sich aus dessen Größe.
- **Reine Byte-Reihenfolge, keine Wertinterpretation**: Der Baustein ändert nur die Anordnung der Bytes, nicht deren Bedeutung oder Vorzeichen.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur Konvertierung und zu `CNF`.

Anwendungsszenarien

- **Netzwerkkommunikation**: Viele Netzwerkprotokolle (z. B. TCP/IP-Header) verwenden Big-Endian („Network Byte Order“); `F_TO_LITTLE_ENDIAN` stellt die korrekte Byte-Reihenfolge beim Senden/Empfangen sicher.
- **Interoperabilität mit Fremdsystemen**: Austausch von Binärdaten mit Systemen, die eine andere native Byte-Reihenfolge verwenden als die Zielplattform.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`F_TO_BIG_ENDIAN`**: dieselbe Konvertierungsrichtung (in), aber für die jeweils andere Byte-Reihenfolge.
- **`F_FROM_LITTLE_ENDIAN`**: die Gegenrichtung für dieselbe Byte-Reihenfolge.

Fazit

`F_TO_LITTLE_ENDIAN` liefert eine einfache, generisch typisierte Byte-Reihenfolgen-Konvertierung und ist ein wichtiger Baustein für die korrekte Interoperabilität mit Netzwerkprotokollen und Fremdsystemen unterschiedlicher Endianness.

BCD Konvertierungen

🎧 Podcast

- Das Ingenieurs-Datenblatt des VBCD Ventils entschlüsselt
- Das VBCD DE A Ventil: Wie ein unsichtbarer Held Kräne, Bagger und Co. sicher steuert
- The VBCD DE-A Valve: Unpacking the Silent Guardian of Hydraulic Safety and Precision
- VBCD DE A: Il Cuore Idraulico per Sicurezza e Controllo di Carichi Estremi
- VBCD DE A [illegible]“[illegible]”[illegible]“[illegible]”

F_BYTE_BCD_TO_USINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_BYTE_BCD_TO_USINT` dient der Konvertierung eines BCD-codierten BYTE-Wertes in einen USINT-Wert. BCD (Binary Coded Decimal) ist eine Codierung, bei der jede Ziffer einer Dezimalzahl durch 4 Bits dargestellt wird. Dieser Funktionsblock ist besonders nützlich in Anwendungen, bei denen BCD-codierte Daten verarbeitet werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Eingang ausgelöst, wird der Algorithmus zur Umwandlung des BCD-codierten BYTE-Wertes in einen USINT-Wert ausgeführt.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ausgang wird aktiviert, sobald die Umwandlung abgeschlossen ist.

Daten-Eingänge

- **IN**: Der BCD-codierte BYTE-Wert, der in einen USINT-Wert umgewandelt werden soll.

Daten-Ausgänge

- **OUT**: Der resultierende USINT-Wert nach der Konvertierung.

Adapter

- Dieser Funktionsblock besitzt keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung eines BCD-codierten BYTE-Wertes in einen USINT-Wert durch, sobald das Ereignis `REQ` ausgelöst wird. Die Umwandlung erfolgt mittels der integrierten Funktion `BYTE_BCD_TO_USINT`. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst, um den Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne zusätzliche Verzögerung.
- Der Funktionsblock ist für die Verarbeitung von BCD-codierten Daten optimiert.

Zustandsübersicht

1. **Idle-Zustand**: Der Funktionsblock wartet auf das `REQ`-Ereignis.
2. **Konvertierungszustand**: Bei Auslösung von `REQ` wird die Konvertierung durchgeführt.
3. **Abschlusszustand**: Nach erfolgreicher Konvertierung wird `CNF` ausgelöst und der Funktionsblock kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- Verarbeitung von BCD-codierten Daten in industriellen Steuerungssystemen.
- Konvertierung von BCD-codierten Sensordaten in numerische Werte.
- Integration in Steuerungslogik, die BCD-Daten verarbeiten muss.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsfunktionsblöcken ist `F_BYTE_BCD_TO_USINT` spezialisiert auf die Umwandlung von BCD-codierten BYTE-Werten in USINT-Werte.
- Andere Funktionsblöcke könnten zusätzliche Konvertierungsoptionen bieten, sind jedoch möglicherweise weniger effizient für diesen spezifischen Anwendungsfall.

Fazit

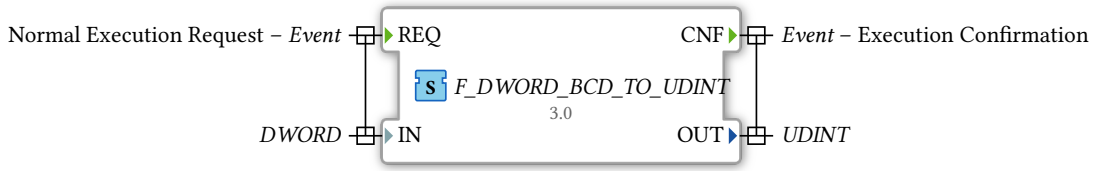
Der Funktionsblock `F_BYTE_BCD_TO_USINT` bietet eine effiziente und spezialisierte Lösung für die Konvertierung von BCD-codierten BYTE-Werten in USINT-Werte. Durch seine einfache und direkte Funktionsweise ist er ideal für Anwendungen, die eine schnelle und zuverlässige Umwandlung erfordern.

F_DWORD_BCD_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DWORD_BCD_TO_UDINT` dient der Konvertierung eines BCD-codierten DWORD-Werts in einen UDINT-Wert. BCD (Binary Coded Decimal) ist eine Codierungsmethode, bei der jede Ziffer einer Dezimalzahl durch vier Bits dargestellt wird. Dieser Funktionsblock ist besonders nützlich in Anwendungen, bei denen BCD-codierte Daten verarbeitet werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Startet die Konvertierung. Dieser Ereigniseingang ist mit dem Dateneingang `IN` verknüpft.

Ereignis-Ausgänge

- `CNF`: Signalisiert den Abschluss der Konvertierung. Dieser Ereignisausgang ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- `IN` (DWORD): Der BCD-codierte Eingangswert, der in einen UDINT-Wert konvertiert werden soll.

Daten-Ausgänge

- `OUT` (UDINT): Der konvertierte UDINT-Wert.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, wenn das `REQ`-Ereignis ausgelöst wird. Die Konvertierung erfolgt mittels der Funktion `DWORD_BCD_TO_UDINT`, die den BCD-codierten `IN`-Wert in einen UDINT-Wert umwandelt. Nach Abschluss der Konvertierung wird das `CNF`-Ereignis ausgelöst, und der konvertierte Wert steht am `OUT`-Ausgang zur Verfügung.

Technische Besonderheiten

- Der Funktionsblock ist einfach aufgebaut und enthält nur einen Algorithmus für die Konvertierung.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Zustandsverwaltung.

Zustandsübersicht

Der Funktionsblock hat keinen internen Zustand. Die Konvertierung wird bei jedem `REQ`-Ereignis neu durchgeführt.

Anwendungsszenarien

- Verarbeitung von BCD-codierten Daten aus externen Geräten oder Sensoren.
- Umwandlung von BCD-codierten Werten für weitere Berechnungen in UDINT-Format.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsbausteinen ist `F_DWORD_BCD_TO_UDINT` spezialisiert auf die Umwandlung von BCD-codierten DWORD-Werten in UDINT-Werte.
- Andere Konvertierungsbausteine könnten zusätzliche Funktionen oder Flexibilität bieten, sind aber möglicherweise weniger effizient für diesen spezifischen Anwendungsfall.

Fazit

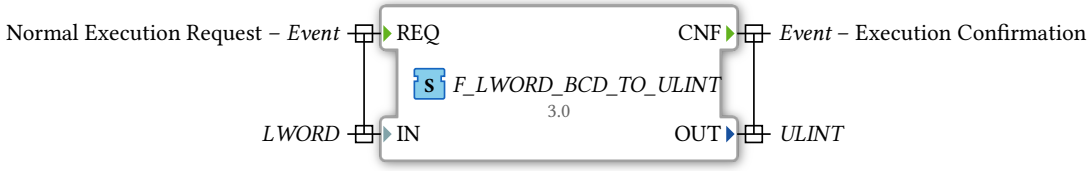
Der `F_DWORD_BCD_TO_UDINT`-Funktionsblock ist ein effizientes Werkzeug für die spezifische Aufgabe der BCD-zu-UDINT-Konvertierung. Seine Einfachheit und direkte Funktionsweise machen ihn zu einer zuverlässigen Lösung für Anwendungen, die diese Art der Konvertierung erfordern.

F_LWORD_BCD_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LWORD_BCD_TO_ULINT` dient der Konvertierung eines BCD-kodierten `LWORD`-Werts in einen `ULINT`-Wert. Dies ist besonders nützlich in Anwendungen, wo BCD-kodierte Daten verarbeitet werden müssen, wie z.B. in industriellen Steuerungen oder Kommunikationsprotokollen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingang für den BCD-kodierten `LWORD`-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgang für den konvertierten `ULINT`-Wert.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, wenn das `REQ`-Ereignis eintrifft. Dabei wird der BCD-kodierte `LWORD`-Wert am Eingang `IN` in einen `ULINT`-Wert umgewandelt und am Ausgang `OUT` bereitgestellt. Das `CNF`-Ereignis signalisiert die erfolgreiche Konvertierung.

Technische Besonderheiten

- Der Funktionsblock verwendet den BCD-zu-ULINT-Konvertierungsalgorithmus, der in der IEC 61131-3-Standardbibliothek definiert ist.
- Die Konvertierung erfolgt synchron mit dem `REQ`-Ereignis.

Zustandsübersicht

1. **Idle-Zustand**: Der Funktionsblock wartet auf das `REQ`-Ereignis.
2. **Konvertierungszustand**: Bei Eintreffen von `REQ` wird die Konvertierung durchgeführt und das `CNF`-Ereignis ausgelöst.

Anwendungsszenarien

- Verarbeitung von BCD-kodierten Daten in industriellen Steuerungen.
- Konvertierung von BCD-kodierten Werten in numerische Werte für weitere Berechnungen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_BCD_TO_ULINT**: Konvertiert BCD-kodierte Werte in `ULINT`, jedoch für andere Datentypen als `LWORD`.
- **F_LWORD_TO_ULINT**: Konvertiert `LWORD` in `ULINT`, jedoch ohne BCD-Kodierung.

Fazit

Der `F_LWORD_BCD_TO_ULINT`-Funktionsblock ist ein spezialisierter Baustein für die Konvertierung von BCD-kodierten `LWORD`-Werten in `ULINT`. Er ist einfach zu verwenden und eignet sich besonders für Anwendungen, die mit BCD-kodierten Daten arbeiten.

F_UDINT_TO_BCD_DWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UDINT_TO_BCD_DWORD` dient der Konvertierung eines vorzeichenlosen 32-Bit-Integer-Werts (UDINT) in ein BCD-kodiertes DWORD. Diese Konvertierung ist insbesondere in Anwendungen nützlich, bei denen numerische Werte in einem für Menschen lesbaren Format dargestellt oder verarbeitet werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Ein vorzeichenloser 32-Bit-Integer-Wert (UDINT), der in ein BCD-kodiertes DWORD umgewandelt werden soll.

Daten-Ausgänge

- **OUT**: Das Ergebnis der Konvertierung als BCD-kodiertes DWORD.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` empfangen wird. Der Algorithmus wandelt den Wert am Eingang `IN` von UDINT in ein BCD-kodiertes DWORD um und legt das Ergebnis am Ausgang `OUT` ab. Anschließend wird das Ereignis `CNF` ausgelöst, um den erfolgreichen Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Die Konvertierung erfolgt mittels der Funktion `UDINT_TO_BCD_DWORD`, die im Algorithmus des Funktionsblocks implementiert ist.
- Der Funktionsblock ist Teil des Pakets `iec61131::conversion`.

Zustandsübersicht

1. **Wartezustand**: Der Funktionsblock wartet auf das Ereignis `REQ`.

2. **Aktivzustand**: Bei Empfang von `REQ` wird die Konvertierung durchgeführt und das Ergebnis an `OUT` ausgegeben.
3. **Bestätigungszustand**: Nach erfolgreicher Konvertierung wird `CNF` ausgelöst und der Funktionsblock kehrt in den Wartezustand zurück.

Anwendungsszenarien

- **Anzeigesteuerung**: Konvertierung von numerischen Werten für die Anzeige auf siebensegmentigen Displays oder LCDs.
- **Datenkommunikation**: Vorbereitung von Daten für die Übertragung in BCD-kodierten Protokollen.
- **Druckersteuerung**: Umwandlung von numerischen Werten für den Druck in einem lesbaren Format.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_UDINT_TO_STRING**: Konvertiert einen UDINT in einen String, ist jedoch weniger effizient für BCD-spezifische Anwendungen.
- **F_UDINT_TO_DWORD**: Führt eine direkte Typumwandlung durch, ohne BCD-Kodierung.

Fazit

Der Funktionsblock `F_UDINT_TO_BCD_DWORD` bietet eine effiziente und spezialisierte Lösung für die Konvertierung von vorzeichenlosen 32-Bit-Integern in BCD-kodierte DWORDs. Seine einfache Schnittstelle und zuverlässige Funktionsweise machen ihn ideal für Anwendungen, die eine BCD-Kodierung erfordern.

F_UINT_TO_BCD_WORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UINT_TO_BCD_WORD` dient der Konvertierung eines vorzeichenlosen Integer-Werts (UINT) in ein BCD-kodiertes Wort (WORD). BCD (Binary Coded Decimal) ist eine Methode zur Darstellung von Dezimalzahlen, bei der jede Ziffer durch 4 Bits kodiert wird. Dieser Baustein ist besonders nützlich in Anwendungen, bei denen numerische Werte in einem für Menschen lesbaren Format ausgegeben werden müssen, wie z.B. in Anzeigesystemen oder Druckern.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der zu konvertierende vorzeichenlose Integer-Wert (UINT).

Daten-Ausgänge

- **OUT**: Das Ergebnis der Konvertierung als BCD-kodiertes Wort (WORD).

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Auslösung des Ereigniseingangs `REQ` wird der Algorithmus ausgeführt, der den Wert von `IN` in ein BCD-kodiertes `WORD` umwandelt und das Ergebnis an `OUT` ausgibt. Anschließend wird das Ereignis `CNF` ausgelöst, um den erfolgreichen Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Der Baustein implementiert eine direkte Konvertierung von UINT zu WORD unter Verwendung der BCD-Kodierung.
- Die Konvertierung erfolgt in einem einzigen Algorithmusschritt, was eine hohe Effizienz gewährleistet.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände außer der initialen Bereitschaft zur Verarbeitung des `REQ`-Ereignisses.

Anwendungsszenarien

- Anzeigesteuerungen, die numerische Werte in BCD-Format benötigen.
- Drucker und andere Ausgabegeräte, die BCD-kodierte Daten verarbeiten.
- Systeme, die eine Kompatibilität mit älteren BCD-basierten Komponenten erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu generischen Konvertierungsbausteinen bietet `F_UINT_TO_BCD_WORD` eine spezialisierte Lösung für die BCD-Kodierung, die effizienter und einfacher in der Handhabung sein kann als universelle Konvertierungsmethoden.

Fazit

Der `F_UINT_TO_BCD_WORD` Funktionsblock ist ein spezialisiertes Werkzeug für die effiziente Konvertierung von vorzeichenlosen Integer-Werten in das BCD-Format. Seine einfache Schnittstelle und direkte Funktionsweise machen ihn ideal für Anwendungen, die eine schnelle und zuverlässige BCD-Kodierung erfordern.

F_ULINT_TO_BCD_LWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_ULINT_TO_BCD_LWORD` dient der Konvertierung eines `ULINT`-Wertes (64-Bit unsigned integer) in einen `LWORD`-Wertyp im BCD-Format (Binary Coded Decimal). Dieser Block ist besonders nützlich in Anwendungen, bei denen numerische Werte in ein für BCD-Systeme kompatibles Format umgewandelt werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der zu konvertierende Wert wird über den Daten-Eingang `IN` übergeben.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der konvertierte Wert wird über den Daten-Ausgang `OUT` ausgegeben.

Daten-Eingänge

- **IN**: Der Eingang für den `ULINT`-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgang für den konvertierten `LWORD`-Wertyp im BCD-Format.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `ULINT` zu `LWORD` im BCD-Format durch, sobald das `REQ`-Ereignis ausgelöst wird. Die Konvertierung erfolgt über die integrierte Funktion `ULINT_TO_BCD_LWORD`. Nach Abschluss der Konvertierung wird das `CNF`-Ereignis ausgelöst, und der konvertierte Wert steht am `OUT`-Ausgang zur Verfügung.

Technische Besonderheiten

- Unterstützt die Konvertierung von 64-Bit unsigned integer (`ULINT`) zu 64-Bit BCD (`LWORD`).
- Keine zusätzlichen Parameter oder Einstellungen erforderlich.

Zustandsübersicht

1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Konvertierung**: Führt die Konvertierung durch, sobald `REQ` aktiviert wird.
3. **Abschluss**: Sendet das `CNF`-Ereignis und gibt den konvertierten Wert aus.

Anwendungsszenarien

- Industrielle Steuerungssysteme, die BCD-kodierte Daten verarbeiten.
- Kommunikation mit Peripheriegeräten, die BCD-Daten erwarten.
- Datenkonvertierung in Automatisierungsanwendungen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken bietet `F_ULINT_TO_BCD_LWORD` eine spezialisierte Lösung für die Umwandlung in das BCD-Format.
- Andere Blöcke könnten zusätzliche Konvertierungsoptionen bieten, sind jedoch möglicherweise weniger effizient für diesen spezifischen Anwendungsfall.

Fazit

Der `F_ULINT_TO_BCD_LWORD`-Funktionsblock ist eine effiziente und spezialisierte Lösung für die Konvertierung von `ULINT` zu `LWORD` im BCD-Format. Seine einfache Handhabung und direkte Funktionsweise machen ihn ideal für Anwendungen, die eine schnelle und zuverlässige Datenumwandlung erfordern.

F_USINT_TO_BCD_BYTE

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_USINT_TO_BCD_BYTE` dient der Konvertierung eines vorzeichenlosen 8-Bit-Integer-Werts (USINT) in ein BCD-kodiertes Byte (BYTE). BCD (Binary Coded Decimal) ist eine Kodierung, bei der jede Dezimalziffer durch vier Bits dargestellt wird.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Ein vorzeichenloser 8-Bit-Integer-Wert (USINT), der in BCD umgewandelt werden soll.

Daten-Ausgänge

- **OUT**: Das Ergebnis der Konvertierung als BCD-kodiertes Byte (BYTE).

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, wenn das Ereignis `REQ` eintritt. Der Algorithmus wandelt den Eingabewert `IN` von USINT in ein BCD-kodiertes Byte um und speichert das Ergebnis in `OUT`. Anschließend wird das Ereignis `CNF` ausgelöst, um den Abschluss der Konvertierung zu signalisieren.

Technische Besonderheiten

- Der Funktionsblock ist deterministisch und führt keine internen Zustandsänderungen durch.
- Die Konvertierung erfolgt direkt über die Funktion `USINT_TO_BCD_BYTE`.

Zustandsübersicht

Der Funktionsblock hat keinen internen Zustand. Die Konvertierung wird bei jedem `REQ`-Ereignis neu durchgeführt.

Anwendungsszenarien

- Anzeigesteuerung für siebensegmentige Displays.
- Kommunikation mit Geräten, die BCD-kodierte Daten erwarten.
- Datenkonvertierung in Steuerungssystemen, die BCD-Format erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu generischen Konvertierungsblöcken ist `F_USINT_TO_BCD_BYTE` spezialisiert auf die Umwandlung von USINT zu BCD.
- Andere Blöcke könnten zusätzliche Funktionen wie Fehlerbehandlung oder unterschiedliche Kodierungen bieten, sind aber oft komplexer.

Fazit

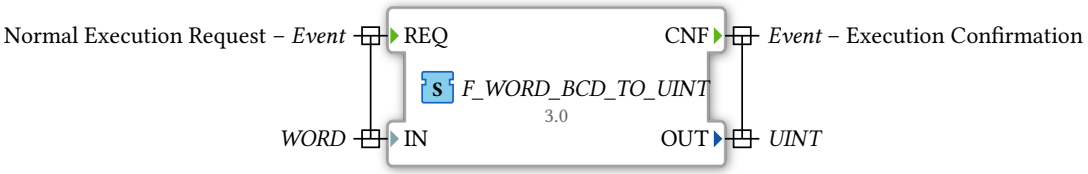
Der Funktionsblock `F_USINT_TO_BCD_BYTE` ist ein effizientes Werkzeug für die spezifische Aufgabe der Konvertierung von USINT zu BCD. Seine Einfachheit und Determinismus machen ihn zu einer zuverlässigen Wahl in Steuerungsanwendungen, die BCD-kodierte Daten erfordern.

F_WORD_BCD_TO_UINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_WORD_BCD_TO_UINT` dient der Konvertierung eines BCD-codierten WORD-Wertes in einen vorzeichenlosen Integer (UINT). Dies ist besonders nützlich in Anwendungen, bei denen Daten im BCD-Format verarbeitet oder angezeigt werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge
- `REQ`: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge
- `CNF`: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge
- `IN` (WORD): Der BCD-codierte Eingangswert, der konvertiert werden soll.

Daten-Ausgänge
- `OUT` (UINT): Der konvertierte vorzeichenlose Integer-Wert.

Adapter
Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise
Bei Auslösung des `REQ`-Ereignisses wird der BCD-codierte Wert am Eingang `IN` in einen vorzeichenlosen Integer-Wert konvertiert. Das Ergebnis wird am Ausgang `OUT` ausgegeben und das `CNF`-Ereignis signalisiert den Abschluss der Konvertierung.

Technische Besonderheiten
- Der Funktionsblock ist einfach und direkt, ohne komplexe Zustandslogik.
- Die Konvertierung erfolgt mittels der Funktion `WORD_BCD_TO_UINT`.

Zustandsübersicht
Da es sich um einen einfachen Funktionsblock handelt, gibt es keine komplexen Zustände. Der Block reagiert direkt auf das `REQ`-Ereignis mit einer Konvertierung und der Ausgabe des Ergebnisses.

Anwendungsszenarien
- Konvertierung von BCD-codierten Daten aus externen Geräten oder Sensoren.
- Verwendung in Steuerungssystemen, die BCD-Daten verarbeiten müssen.
- Anzeige von BCD-Daten in einem für Menschen lesbaren Format.

⚖️ Vergleich mit ähnlichen Bausteinen
- Im Vergleich zu generischen Konvertierungsblöcken ist `F_WORD_BCD_TO_UINT` spezialisiert auf die Konvertierung von BCD zu UINT.
- Andere Blöcke könnten zusätzliche Funktionen wie Fehlerbehandlung oder verschiedene Codierungsformate unterstützen, sind dafür aber komplexer.

Fazit
Der `F_WORD_BCD_TO_UINT` Funktionsblock ist ein effizientes Werkzeug für die spezifische Aufgabe der BCD-zu-UINT-Konvertierung. Seine Einfachheit und direkte Funktionsweise machen ihn ideal für Anwendungen, bei denen schnelle und zuverlässige Konvertierungen benötigt werden.

BOOL

BOOL Konvertierungen

🎧 Podcast
- George Boole: Wie ein Autodidakt mit 0 und 1 das Fundament unserer digitalen Welt legte

🛠️ Zugehörige Übungen
- Uebung_001_AX_b
- Uebung_002a2_AX
- Uebung_002a4
- Uebung_002a5b

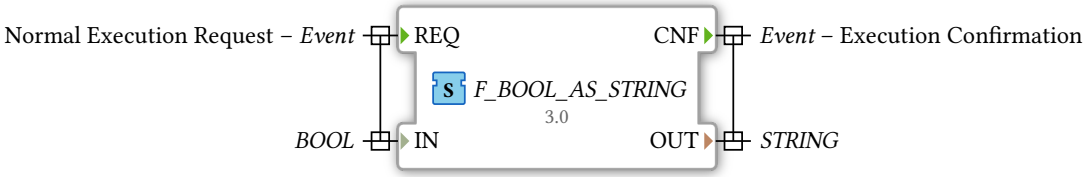
- Uebung_002b3
- Uebung_004b_AX
- Uebung_006a3
- LinksRechts_T_FF_Event
- Uebung_035a2
- Uebung_035a3
- SoftkeyOverride_TO_QX_BG
- Uebung_055
- Uebung_056
- Bool8_TO_logiBUS_QX8
- Uebung_094a_AX
- Uebung_160

F_BOOL_AS_STRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_BOOL_AS_STRING` dient zur Konvertierung eines booleschen Wertes (`BOOL`) in eine Zeichenkette (`STRING`). Diese Funktionalität ist besonders nützlich, wenn boolesche Werte für die Ausgabe oder Weiterverarbeitung in einem Textformat benötigt werden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung des Funktionsblocks. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN**: Der boolesche Wert (`BOOL`), der in eine Zeichenkette konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Die resultierende Zeichenkette (`STRING`), die den konvertierten booleschen Wert repräsentiert.

Adapter

Es sind keine Adapter für diesen Funktionsblock definiert.

Funktionsweise

Der Funktionsblock nutzt einen einfachen Algorithmus, um den booleschen Wert `IN` in eine Zeichenkette `OUT` umzuwandeln. Die Konvertierung erfolgt durch die Funktion `BOOL_AS_STRING(IN)`, die intern den Wert von `IN` in eine entsprechende Zeichenkette umsetzt (z.B. `TRUE` oder `FALSE`).

Technische Besonderheiten

- Der Funktionsblock ist einfach und effizient implementiert, ohne zusätzliche Zustände oder komplexe Logik.
- Die Konvertierung erfolgt synchron mit dem Auslösen des `REQ`-Ereignisses.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine expliziten Zustände. Die Ausführung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- **Protokollierung**: Konvertierung von booleschen Werten für Logging-Zwecke.
- **Benutzeroberflächen**: Darstellung von booleschen Werten in menschenlesbarer Form.
- **Datenkommunikation**: Vorbereitung von booleschen Werten für die Übertragung als Text.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_STRING_TO_BOOL**: Führt die umgekehrte Konvertierung von `STRING` zu `BOOL` durch.
- **F_BOOL_TO_***: Andere Konvertierungsbausteine, die boolesche Werte in verschiedene Datentypen umwandeln.

Fazit

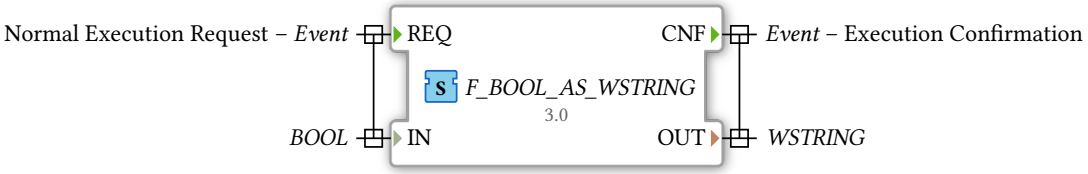
Der `F_BOOL_AS_STRING` Funktionsblock bietet eine einfache und effektive Möglichkeit, boolesche Werte in Zeichenketten umzuwandeln. Seine klare Schnittstelle und direkte Funktionsweise machen ihn zu einem nützlichen Werkzeug in vielen Anwendungsfällen.

F_BOOL_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_BOOL_AS_WSTRING` dient der Konvertierung eines booleschen Wertes (`BOOL`) in eine breite Zeichenkette (`WSTRING`). Diese Funktionalität ist besonders nützlich, wenn boolesche Werte in menschenlesbarer Form ausgegeben oder weiterverarbeitet werden sollen.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Startet die Konvertierung des booleschen Eingabewerts `IN`.

Ereignis-Ausgänge

- `CNF`: Signalisiert den Abschluss der Konvertierung und liefert das Ergebnis `OUT`.

Daten-Eingänge

- `IN` (`BOOL`): Der boolesche Wert, der konvertiert werden soll.

Daten-Ausgänge

- `OUT` (`WSTRING`): Der konvertierte Wert als breite Zeichenkette.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock konvertiert den booleschen Eingabewert `IN` in eine breite Zeichenkette (`WSTRING`). Die Konvertierung erfolgt mithilfe der Funktion `BOOL_AS_WSTRING`, die den Wert `TRUE` in `"TRUE"` und `FALSE` in `"FALSE"` umwandelt. Die Konvertierung wird durch das Ereignis `REQ` ausgelöst und das Ergebnis wird mit dem Ereignis `CNF` bestätigt.

Technische Besonderheiten

- Der Funktionsblock ist einfach aufgebaut und enthält nur einen Algorithmus zur Konvertierung.

- Die Konvertierung erfolgt direkt und ohne Verzögerung.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar nach dem Auslösen des `REQ`-Ereignisses.

Anwendungsszenarien

- Anzeige von booleschen Werten in Benutzeroberflächen.
- Protokollierung von booleschen Zuständen in Logdateien.
- Weiterverarbeitung von booleschen Werten in Systemen, die Zeichenketten erwarten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu `F_BOOL_TO_STRING` wird hier eine breite Zeichenkette (`WSTRING`) anstelle einer normalen Zeichenkette (`STRING`) erzeugt.
- Ähnliche Funktionsblöcke wie `F_INT_AS_WSTRING` oder `F_REAL_AS_WSTRING` konvertieren andere Datentypen in `WSTRING`.

Fazit

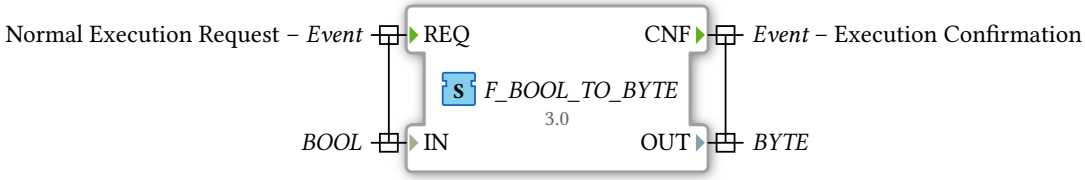
Der Funktionsblock `F_BOOL_AS_WSTRING` bietet eine einfache und effiziente Möglichkeit, boolesche Werte in breite Zeichenketten umzuwandeln. Durch seinen schlanken Aufbau ist er besonders für Anwendungen geeignet, in denen eine schnelle und unkomplizierte Konvertierung benötigt wird.

F_BOOL_TO_BYTE

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_BOOL_TO_BYTE` dient der Konvertierung eines booleschen Wertes (`BOOL`) in einen Byte-Wert (`BYTE`). Diese Konvertierung ist nützlich, wenn boolesche Signale in Systemen verarbeitet werden müssen, die Byte-Daten erfordern.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Dieser Ereigniseingang löst die Ausführung des Funktionsblocks aus. Er ist mit dem Dateneingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Dieser Ereignisausgang signalisiert die erfolgreiche Ausführung des Funktionsblocks. Er ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (`BOOL`): Der boolesche Eingangswert, der in ein Byte konvertiert werden soll.

Daten-Ausgänge

- **OUT** (`BYTE`): Der resultierende Byte-Wert nach der Konvertierung.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `BOOL` zu `BYTE` durch, sobald das Ereignis `REQ` ausgelöst wird. Der boolesche Wert `IN` wird in einen Byte-Wert umgewandelt, wobei `FALSE` zu `0` und `TRUE` zu `1`

konvertiert wird. Das Ergebnis wird über den Ausgang `OUT` ausgegeben, und das Ereignis `CNF` signalisiert die erfolgreiche Ausführung.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher Funktionsblock (`SimpleFB`) und verwendet einen ST-Algorithmus (`ST`) für die Konvertierung.
- Die Konvertierung erfolgt durch die Funktion `BOOL_TO_BYTE`, die intern den booleschen Wert in einen Byte-Wert umwandelt.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Ausführung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- **Signalverarbeitung**: Konvertierung von booleschen Steuersignalen in Byte-Werte für die Kommunikation mit Geräten, die Byte-Daten erwarten.
- **Datenprotokollierung**: Umwandlung von booleschen Statusinformationen in Byte-Format für die Speicherung oder Übertragung.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_BOOL_TO_INT**: Konvertiert einen booleschen Wert in einen Integer-Wert, während `F_BOOL_TO_BYTE` speziell für Byte-Ausgaben optimiert ist.
- **F_BOOL_TO_WORD**: Ähnlich wie `F_BOOL_TO_BYTE`, aber für größere Datenformate (Word) ausgelegt.

Fazit

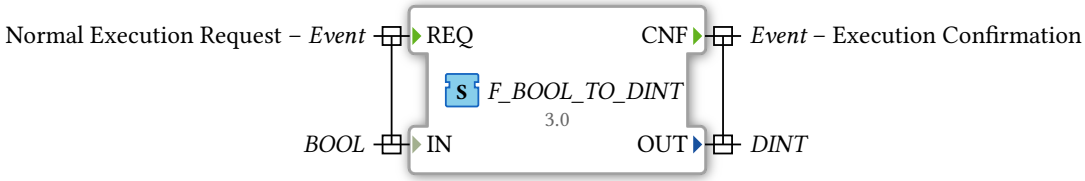
Der Funktionsblock `F_BOOL_TO_BYTE` ist ein einfaches und effizientes Werkzeug zur Konvertierung von booleschen Werten in Byte-Daten. Er eignet sich besonders für Anwendungen, die eine kompakte Darstellung von booleschen Signalen in Byte-Format erfordern. Durch seine einfache Schnittstelle und direkte Funktionsweise ist er leicht in bestehende Systeme integrierbar.

F_BOOL_TO_DINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_BOOL_TO_DINT` dient der Konvertierung eines booleschen Wertes (`BOOL`) in einen 32-Bit-Integer-Wert (`DINT`). Diese Konvertierung ist nützlich, wenn boolesche Signale in numerische Berechnungen oder Steuerungen integriert werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Bei Auslösung dieses Ereignisses wird der boolesche Eingangswert (`IN`) in einen `DINT`-Wert umgewandelt.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieses Ereignis wird zusammen mit dem konvertierten Ausgangswert (`OUT`) ausgegeben.

Daten-Eingänge

- **IN**: Der boolesche Eingangswert (`BOOL`), der in einen `DINT`-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der konvertierte 32-Bit-Integer-Wert (`DINT`). Der Wert ist `1`, wenn `IN` `TRUE` ist, andernfalls `0`.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `BOOL` zu `DINT` durch, indem er den booleschen Wert `TRUE` in `1` und `FALSE` in `0` umwandelt. Die Konvertierung wird durch das Ereignis `REQ` ausgelöst und das Ergebnis wird mit dem Ereignis `CNF` bestätigt.

Technische Besonderheiten

- Die Konvertierung erfolgt mittels der Funktion `BOOL_TO_DINT`, die im Algorithmus des Funktionsblocks implementiert ist.
- Der Funktionsblock ist einfach und deterministisch, da er keine internen Zustände oder komplexe Berechnungen beinhaltet.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Integration boolescher Signale in numerische Steuerungen.
- Umwandlung von Schaltzuständen (`TRUE`/`FALSE`) in numerische Werte für weitere Berechnungen.
- Schnittstellen zwischen booleschen und numerischen Systemkomponenten.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_BOOL_TO_INT**: Konvertiert `BOOL` in einen 16-Bit-Integer (`INT`). Im Gegensatz dazu liefert `F_BOOL_TO_DINT` einen 32-Bit-Integer.
- **F_BOOL_TO_REAL**: Konvertiert `BOOL` in einen Gleitkommawert (`REAL`). `F_BOOL_TO_DINT` liefert hingegen einen ganzzahligen Wert.

Fazit

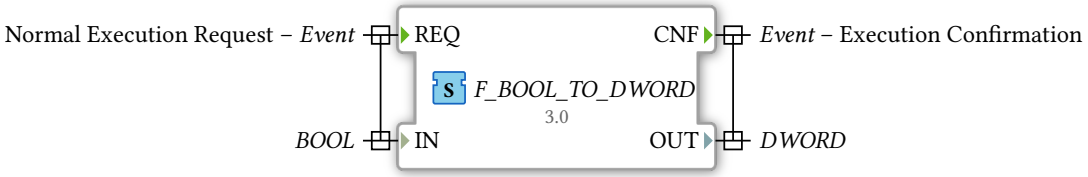
Der Funktionsblock `F_BOOL_TO_DINT` ist ein einfaches und effizientes Werkzeug zur Konvertierung boolescher Werte in 32-Bit-Integer. Er eignet sich besonders für Anwendungen, bei denen boolesche Signale in numerische Steuerungen integriert werden müssen. Die deterministische Funktionsweise und die klare Schnittstellenstruktur machen ihn zu einem zuverlässigen Baustein in der IEC 61499-basierten Automatisierung.

F_BOOL_TO_DWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_BOOL_TO_DWORD` dient der Konvertierung eines booleschen Wertes (`BOOL`) in einen 32-Bit-unsigned-Integer-Wert (`DWORD`). Der Block ist besonders nützlich in Szenarien, wo boolesche Signale in numerische Werte umgewandelt werden müssen, beispielsweise für die Kommunikation mit Systemen, die nur numerische Eingaben verarbeiten können.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der boolesche Wert (`BOOL`), der in ein `DWORD` konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der konvertierte 32-Bit-unsigned-Integer-Wert (`DWORD`). Der Wert ist `1` (TRUE) oder `0` (FALSE).

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` empfangen wird. Der boolesche Wert `IN` wird in ein `DWORD` umgewandelt:

- `TRUE` wird zu `1`
- `FALSE` wird zu `0`

Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst, und der konvertierte Wert steht am Ausgang `OUT` zur Verfügung.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne Verzögerung.
- Der Block ist deterministisch und hat keine internen Zustände.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung wird bei jedem `REQ`-Ereignis neu durchgeführt.

Anwendungsszenarien

- Umwandlung von Schaltzuständen (Ein/Aus) in numerische Steuersignale.
- Integration in Steuerungssysteme, die nur numerische Eingaben verarbeiten können.
- Protokollierung von booleschen Ereignissen als numerische Werte.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu `F_BOOL_TO_INT` oder `F_BOOL_TO_REAL` konvertiert dieser Block spezifisch in einen `DWORD`-Typ, der für 32-Bit-Systeme optimiert ist.
- Ähnliche Blöcke wie `F_DWORD_TO_BOOL` führen die umgekehrte Konvertierung durch.

Fazit

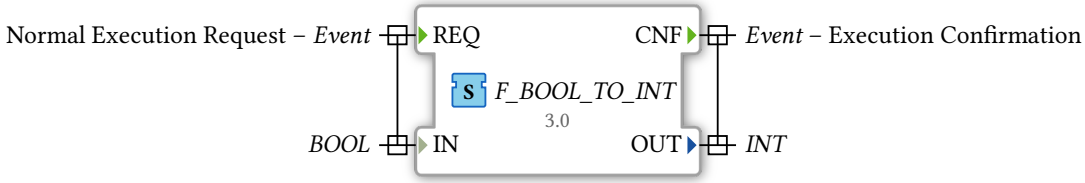
Der `F_BOOL_TO_DWORD`-Funktionsblock ist ein einfacher und effizienter Baustein für die Konvertierung von booleschen Werten in `DWORD`. Seine deterministische Funktionsweise und die klare Schnittstelle machen ihn zu einer zuverlässigen Komponente in Steuerungs- und Kommunikationsanwendungen.

F_BOOL_TO_INT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_BOOL_TO_INT` dient der Konvertierung eines booleschen Wertes (`BOOL`) in einen ganzzahligen Wert (`INT`). Dabei wird `TRUE` in `1` und `FALSE` in `0` umgewandelt. Der Block ist besonders nützlich in Szenarien, wo boolesche Logik in numerische Berechnungen integriert werden muss.



Schnittstellenstruktur

Ereignis-Eingänge
- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge
- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge
- **IN**: Der boolesche Wert (`BOOL`), der konvertiert werden soll.

Daten-Ausgänge
- **OUT**: Der konvertierte ganzzahlige Wert (`INT`). `TRUE` wird zu `1` und `FALSE` zu `0`.

Adapter
- Keine Adapter vorhanden.

Funktionsweise
Bei Auslösung des Ereignisses `REQ` wird der boolesche Wert am Eingang `IN` gelesen und in einen ganzzahligen Wert umgewandelt. Das Ergebnis wird am Ausgang `OUT` ausgegeben und das Ereignis `CNF` wird ausgelöst, um den Abschluss der Operation zu signalisieren.

Technische Besonderheiten
- Der Block ist ein einfacher Funktionsblock (`SimpleFB`) ohne interne Zustände.
- Die Konvertierung erfolgt direkt und ohne Verzögerung.

Zustandsübersicht
Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge oder interne Zustände. Die Operation wird bei jedem `REQ`-Ereignis neu ausgeführt.

Anwendungsszenarien
- Integration von boolescher Logik in numerische Steuerungen.
- Umwandlung von Schaltzuständen (`TRUE`/`FALSE`) in numerische Werte für weitere Berechnungen.
- Verwendung in Regelungstechnik, wo boolesche Signale in numerische Steuersignale umgewandelt werden müssen.

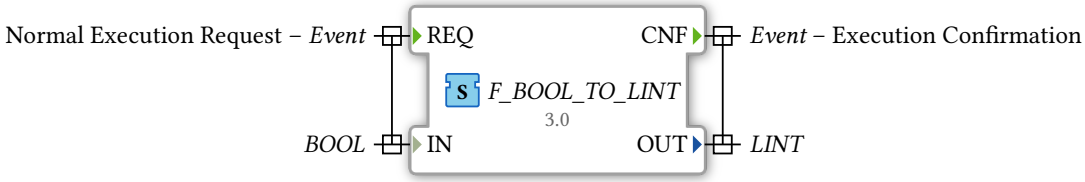
⚖️ Vergleich mit ähnlichen Bausteinen
- **F_BOOL_TO_REAL**: Konvertiert `BOOL` in `REAL` (Gleitkommazahl), ähnlich wie `F_BOOL_TO_INT`, aber mit Gleitkomma-Ausgabe.
- **F_INT_TO_BOOL**: Führt die umgekehrte Konvertierung durch (`INT` zu `BOOL`).

Fazit
Der `F_BOOL_TO_INT`-Funktionsblock ist ein einfaches, aber effektives Werkzeug zur Konvertierung von booleschen Werten in ganzzahlige Werte. Seine Einfachheit und direkte Funktionsweise machen ihn ideal für Anwendungen, die eine schnelle und unkomplizierte Umwandlung erfordern.

F_BOOL_TO_LINT

• – ◦ • – ◦ • – ◦ •

Einleitung
Der Funktionsblock `F_BOOL_TO_LINT` dient der Konvertierung eines booleschen Wertes (`BOOL`) in einen 64-Bit Integer-Wert (`LINT`). Dieser Block ist besonders nützlich in Szenarien, wo eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Dieser Eingang löst die Ausführung des Funktionsblocks aus. Er ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Dieser Ausgang signalisiert die erfolgreiche Ausführung des Funktionsblocks. Er ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingang für den booleschen Wert (`BOOL`), der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgang für den konvertierten 64-Bit Integer-Wert (`LINT`).

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `BOOL` zu `LINT` durch, wenn das Ereignis `REQ` empfangen wird. Die Konvertierung erfolgt mittels der Funktion `BOOL_TO_LINT(IN)`, wobei der boolesche Wert `IN` in einen `LINT`-Wert umgewandelt wird. Das Ergebnis wird am Ausgang `OUT` ausgegeben und das Ereignis `CNF` wird ausgelöst.

Technische Besonderheiten

- Der Block ist ein einfacher Funktionsblock (`SimpleFB`) und verwendet einen ST-Algorithmus für die Konvertierung.
- Die Konvertierung ist deterministisch und erfolgt ohne Verzögerung.

Zustandsübersicht

Der Funktionsblock hat keinen internen Zustand. Die Konvertierung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo boolesche Werte in numerische Werte umgewandelt werden müssen.
- Integration in größere Systeme, die eine einheitliche Datentypverarbeitung erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsblöcken wie `F_BOOL_TO_INT` oder `F_BOOL_TO_DINT` bietet dieser Block eine höhere Präzision durch die Verwendung des `LINT`-Datentyps (64-Bit).

Fazit

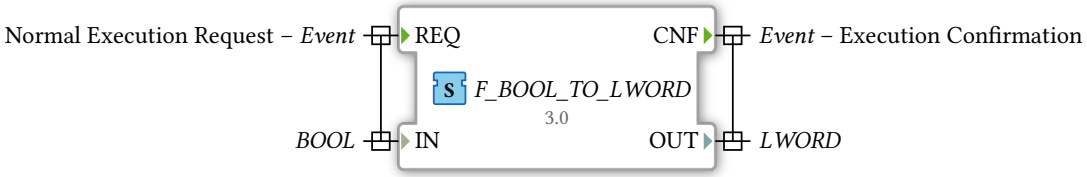
Der `F_BOOL_TO_LINT`-Funktionsblock ist ein effizientes Werkzeug für die Konvertierung von booleschen Werten in 64-Bit Integer-Werte. Seine einfache und deterministische Funktionsweise macht ihn zu einer zuverlässigen Komponente in Steuerungsanwendungen.

F_BOOL_TO_LWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_BOOL_TO_LWORD` dient der Konvertierung eines booleschen Wertes (`BOOL`) in einen 64-Bit langen vorzeichenlosen Integerwert (`LWORD`). Diese Konvertierung ist insbesondere in Szenarien nützlich, in denen boolesche Werte in größere Datenstrukturen eingebettet oder mit anderen langen Integerwerten verarbeitet werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Löst die Konvertierung aus. Wird mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Wird nach erfolgreicher Konvertierung ausgelöst. Wird mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der boolesche Wert (`BOOL`), der in einen `LWORD` konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der konvertierte 64-Bit lange vorzeichenlose Integerwert (`LWORD`).

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `BOOL` zu `LWORD` durch, wenn das `REQ`-Ereignis eintritt. Der boolesche Wert `IN` wird in einen `LWORD` umgewandelt, wobei `FALSE` zu `0` und `TRUE` zu `1` konvertiert wird. Das Ergebnis wird am Daten-Ausgang `OUT` ausgegeben, und das `CNF`-Ereignis wird ausgelöst, um die erfolgreiche Ausführung zu signalisieren.

Technische Besonderheiten

- Die Konvertierung erfolgt mittels der Funktion `BOOL_TO_LWORD`, die den booleschen Wert in einen 64-Bit-Wert umwandelt.
- Der Funktionsblock ist einfach und deterministisch, da er keine internen Zustände speichert.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock ohne interne Zustände handelt, gibt es keine Zustandsübergänge oder -verwaltung.

Anwendungsszenarien

- Einbettung boolescher Werte in größere Datenstrukturen, die `LWORD` verwenden.
- Verwendung in Systemen, die boolesche Werte als lange Integerwerte verarbeiten müssen, z.B. bei der Kommunikation mit externen Systemen oder bei der Speicherung in Datenbanken.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_BOOL_TO_**: Ähnliche Funktionsblöcke existieren für andere Datentypen (z.B. `F_BOOL_TO_BYTE`, `F_BOOL_TO_WORD`). Diese konvertieren boolesche Werte in andere Integer-Datentypen mit unterschiedlicher Bitbreite.
- **F__TO_LWORD**: Andere Konvertierungsblöcke wandeln verschiedene Datentypen (z.B. `INT`, `DINT`) in `LWORD` um, aber `F_BOOL_TO_LWORD` ist spezifisch für boolesche Werte.

Fazit

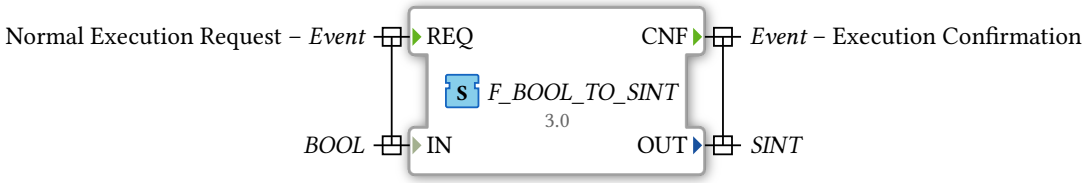
Der `F_BOOL_TO_LWORD`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, boolesche Werte in 64-Bit lange Integerwerte umzuwandeln. Seine deterministische Funktionsweise und klare Schnittstellenstruktur machen ihn zu einem nützlichen Baustein in Steuerungs- und Automatisierungssystemen, insbesondere dort, wo boolesche Werte in größere Datenstrukturen integriert werden müssen.

F_BOOL_TO_SINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_BOOL_TO_SINT` dient der Konvertierung eines booleschen Wertes (`BOOL`) in einen vorzeichenbehafteten 8-Bit-Integer (`SINT`). Diese Konvertierung ist nützlich, wenn boolesche Werte in Berechnungen oder Steuerungen verwendet werden sollen, die ganzzahlige Werte erfordern.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Ereigniseingang ausgelöst, wird der boolesche Wert am Daten-Eingang `IN` in einen `SINT`-Wert konvertiert.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieses Ereignis wird zusammen mit dem konvertierten Wert am Daten-Ausgang `OUT` ausgegeben.

Daten-Eingänge

- **IN**: Der boolesche Wert (`BOOL`), der in einen `SINT`-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der konvertierte Wert (`SINT`). Der Ausgangswert ist `1`, wenn der Eingang `IN` `TRUE` ist, andernfalls `0`.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock konvertiert den booleschen Eingangswert `IN` in einen `SINT`-Wert. Die Konvertierung erfolgt nach folgender Regel:

- `TRUE` wird zu `1` konvertiert.
- `FALSE` wird zu `0` konvertiert.

Die Konvertierung wird durch das Ereignis `REQ` ausgelöst. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst und der konvertierte Wert am Ausgang `OUT` ausgegeben.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher Algorithmus (`SimpleFB`), der direkt die Konvertierung durchführt.
- Die Konvertierung erfolgt ohne Verzögerung, sobald das Ereignis `REQ` eintrifft.

Zustandsübersicht

Der Funktionsblock hat keinen internen Zustand. Die Konvertierung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Umwandlung von booleschen Signalen in numerische Werte für mathematische Operationen.
- Integration von booleschen Steuerungssignalen in Systeme, die ganzzahlige Werte erwarten.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_BOOL_TO_INT**: Konvertiert `BOOL` in einen 16-Bit-Integer (`INT`). Dieser Baustein ist ähnlich, aber der Ausgangswert hat eine größere Bitbreite.
- **F_BOOL_TO_USINT**: Konvertiert `BOOL` in einen vorzeichenlosen 8-Bit-Integer (`USINT`). Der Unterschied liegt in der Interpretation des Ausgangswertes (vorzeichenbehaftet vs. vorzeichenlos).

Fazit

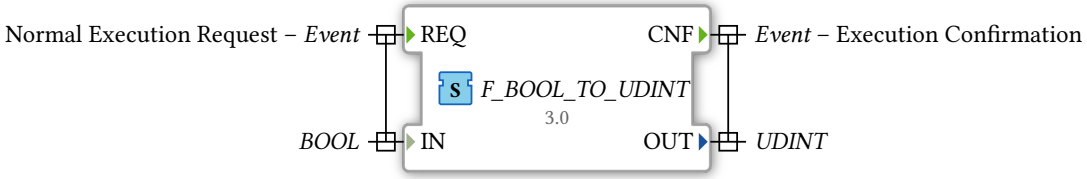
Der Funktionsblock `F_BOOL_TO_SINT` ist ein einfaches und effizientes Werkzeug zur Konvertierung von booleschen Werten in vorzeichenbehaftete 8-Bit-Integer. Er ist besonders nützlich in Steuerungssystemen, wo boolesche Signale in numerische Werte umgewandelt werden müssen.

F_BOOL_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_BOOL_TO_UDINT` dient der Konvertierung eines `BOOL`-Wertes in einen `UDINT`-Wert. Diese Konvertierung ist nützlich, wenn boolesche Werte in Berechnungen oder Speicherungen verwendet werden sollen, die einen größeren numerischen Datentyp erfordern.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingang vom Typ `BOOL`, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgang vom Typ `UDINT`, der das Ergebnis der Konvertierung enthält.

Adapter

Es werden keine Adapter unterstützt.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `BOOL` zu `UDINT` durch, wenn das Ereignis `REQ` ausgelöst wird. Der boolesche Wert `TRUE` wird in `1` und `FALSE` in `0` umgewandelt. Das Ergebnis wird am Daten-Ausgang `OUT` ausgegeben und das Ereignis `CNF` signalisiert die erfolgreiche Ausführung.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne Verzögerung.
- Der Funktionsblock ist für den Einsatz in Echtzeitsystemen geeignet.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Umwandlung von booleschen Schaltzuständen für numerische Berechnungen.
- Integration von booleschen Signalen in Systeme, die nur numerische Werte verarbeiten können.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen wie `F_BOOL_TO_INT` oder `F_BOOL_TO_DINT` bietet dieser Baustein eine Konvertierung in einen vorzeichenlosen 32-Bit-Integer.

Fazit

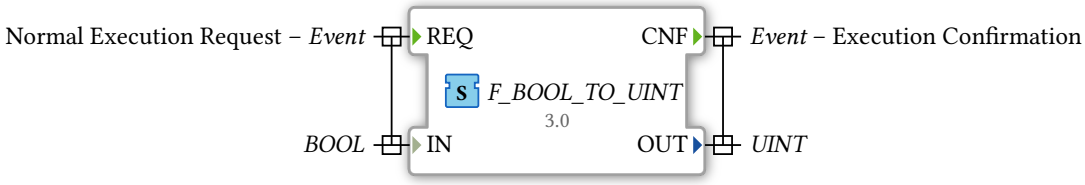
Der `F_BOOL_TO_UDINT` Funktionsblock ist ein einfacher und effizienter Baustein für die Konvertierung von booleschen Werten in vorzeichenlose 32-Bit-Integer. Er ist besonders nützlich in Szenarien, wo boolesche Signale in numerische Prozesse integriert werden müssen.

F_BOOL_TO_UINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_BOOL_TO_UINT` dient der Konvertierung eines booleschen Wertes (`BOOL`) in einen vorzeichenlosen Integerwert (`UINT`). Diese Konvertierung ist besonders nützlich, wenn boolesche Signale in numerische Werte umgewandelt werden müssen, z.B. für weitere Berechnungen oder Steuerungszwecke.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN**: Der boolesche Eingangswert (`BOOL`), der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der konvertierte vorzeichenlose Integerwert (`UINT`).

Adapter

- Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem er den booleschen Wert `IN` in einen vorzeichenlosen Integerwert `OUT` umwandelt. Die Konvertierung erfolgt gemäß der folgenden Regel:

- `FALSE` wird zu `0`
- `TRUE` wird zu `1`

Die Konvertierung wird durch das Ereignis `REQ` ausgelöst und durch das Ereignis `CNF` bestätigt.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher Funktionsblock (`SimpleFB`) und führt die Konvertierung in einem einzigen Algorithmus durch.
- Die Konvertierung erfolgt direkt und ohne Verzögerung.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung wird bei jedem `REQ`-Ereignis neu durchgeführt.

Anwendungsszenarien

- Umwandlung von booleschen Sensordaten in numerische Werte für weitere Verarbeitung.
- Integration in Steuerungssysteme, die numerische Werte anstelle von booleschen Werten erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_BOOL_TO_INT` oder `F_BOOL_TO_REAL` bieten Konvertierungen in andere Datentypen, aber `F_BOOL_TO_UINT` ist speziell für vorzeichenlose Integerwerte optimiert.

🛠️ Zugehörige Übungen

- Uebung_087
- Uebung_087a1
- Uebung_090a1
- Uebung_090a1_AX
- Uebung_090a2
- Uebung_090a2_AX

Fazit

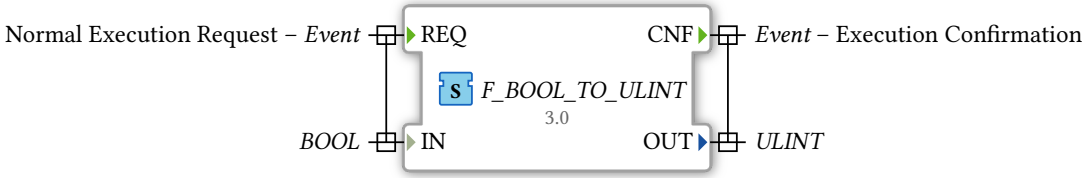
Der `F_BOOL_TO_UINT` Funktionsblock ist ein einfaches und effizientes Werkzeug zur Konvertierung von booleschen Werten in vorzeichenlose Integerwerte. Seine klare Schnittstelle und direkte Funktionsweise machen ihn zu einer zuverlässigen Komponente in Steuerungs- und Automatisierungssystemen.

F_BOOL_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_BOOL_TO_ULINT` dient der Konvertierung eines booleschen Wertes (`BOOL`) in einen vorzeichenlosen 64-Bit-Ganzzahlwert (`ULINT`). Diese Konvertierung ist insbesondere in Szenarien nützlich, in denen boolesche Zustände in numerische Werte umgewandelt werden müssen, beispielsweise für die Verarbeitung in mathematischen Operationen oder für die Speicherung in Datenbanken.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Löst die Konvertierung aus. Verknüpft mit dem Daten-Eingang `IN`.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Wird nach erfolgreicher Konvertierung ausgelöst. Verknüpft mit dem Daten-Ausgang `OUT`.

Daten-Eingänge

- **IN**: Eingang für den booleschen Wert (`BOOL`), der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Ausgang für den konvertierten vorzeichenlosen 64-Bit-Ganzzahlwert (`ULINT`).

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `BOOL` zu `ULINT` durch, wenn das Ereignis `REQ` empfangen wird. Der boolesche Wert `IN` wird in einen `ULINT`-Wert umgewandelt, wobei `FALSE` zu `0` und `TRUE` zu `1` konvertiert wird. Das Ergebnis wird am Ausgang `OUT` ausgegeben und das Ereignis `CNF` wird ausgelöst.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher Funktionsblock (`SimpleFB`) und verwendet einen ST-Algorithmus für die Konvertierung.
- Die Konvertierung erfolgt mittels der Funktion `BOOL_TO_ULINT`, die den booleschen Wert direkt in einen `ULINT`-Wert umwandelt.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Umwandlung von booleschen Schaltzuständen in numerische Werte für weitere Berechnungen.
- Integration in Steuerungssysteme, die boolesche Signale in numerische Datenbanken speichern müssen.
- Verwendung in Datenverarbeitungsketten, die eine Typkonvertierung erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_BOOL_TO_INT` oder `F_BOOL_TO_DINT` konvertieren boolesche Werte in andere numerische Typen, jedoch nicht in `ULINT`.
- `F_BOOL_TO_ULINT` ist spezifisch für die Konvertierung in den vorzeichenlosen 64-Bit-Typ und eignet sich daher für Anwendungen, die einen großen Wertebereich erfordern.

Fazit

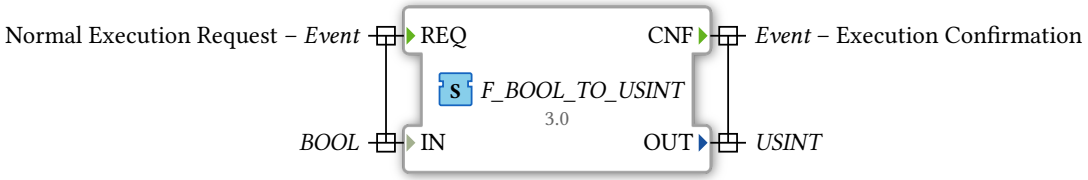
Der Funktionsblock `F_BOOL_TO_ULINT` bietet eine einfache und effiziente Möglichkeit, boolesche Werte in vorzeichenlose 64-Bit-Ganzzahlen umzuwandeln. Durch seine klare Schnittstelle und direkte Funktionsweise ist er ideal für Anwendungen, die eine Typkonvertierung mit hoher Genauigkeit und großem Wertebereich benötigen.

F_BOOL_TO_USINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_BOOL_TO_USINT` dient der Konvertierung eines booleschen Wertes (`BOOL`) in einen vorzeichenlosen 8-Bit-Integer (`USINT`). Diese Konvertierung ist nützlich, wenn boolesche Werte in numerischen Operationen oder für die Kommunikation mit Systemen verwendet werden müssen, die keine booleschen Werte direkt unterstützen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Dieser Ereigniseingang ist mit dem Dateneingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ereignisausgang ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der boolesche Eingangswert (`BOOL`), der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der konvertierte vorzeichenlose 8-Bit-Integer (`USINT`).

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem er den booleschen Wert `IN` in einen numerischen Wert umwandelt:

- `FALSE` wird zu `0` konvertiert.
- `TRUE` wird zu `1` konvertiert.

Die Konvertierung wird durch das Ereignis `REQ` ausgelöst und das Ergebnis wird über den Ausgang `OUT` ausgegeben, begleitet von dem Bestätigungsereignis `CNF`.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher Funktionsblock (`SimpleFB`) und führt die Konvertierung in einem einzigen Algorithmus aus.
- Die Konvertierung erfolgt direkt und ohne Verzögerung, sobald das `REQ`-Ereignis eintrifft.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Umwandlung von booleschen Schaltzuständen in numerische Werte für die Verarbeitung in mathematischen Operationen.
- Kommunikation mit Systemen oder Geräten, die boolesche Werte nicht direkt unterstützen, aber numerische Werte verarbeiten können.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsfunktionsblöcken wie `F_BOOL_TO_INT` oder `F_BOOL_TO_UINT` ist `F_BOOL_TO_USINT` spezifisch für die Konvertierung in einen vorzeichenlosen 8-Bit-Integer optimiert.
- Ähnliche Funktionsblöcke können größere oder vorzeichenbehaftete Integer-Typen verwenden, was je nach Anwendungsfall mehr oder weniger geeignet sein kann.

Fazit

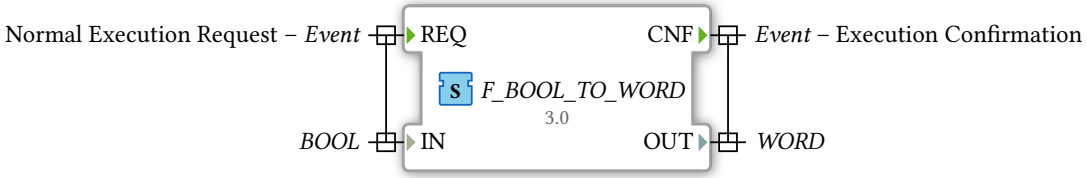
Der `F_BOOL_TO_USINT`-Funktionsblock ist ein einfaches und effizientes Werkzeug zur Konvertierung von booleschen Werten in vorzeichenlose 8-Bit-Integer. Seine direkte und verzögerungsfreie Funktionsweise macht ihn ideal für Echtzeitanwendungen, bei denen schnelle und zuverlässige Konvertierungen erforderlich sind.

F_BOOL_TO_WORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_BOOL_TO_WORD` dient der Konvertierung eines booleschen Wertes (`BOOL`) in einen 16-Bit-Wortwert (`WORD`). Diese Konvertierung ist insbesondere in Szenarien nützlich, in denen boolesche Signale in Wort-Datenstrukturen eingebettet oder weiterverarbeitet werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Löst die Konvertierung aus. Wird mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Wird nach erfolgreicher Konvertierung ausgelöst. Wird mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `BOOL`, der in ein `WORD` konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgangswert vom Typ `WORD`, der das Ergebnis der Konvertierung enthält.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `BOOL` zu `WORD` durch, wenn das Ereignis `REQ` empfangen wird. Der boolesche Wert `IN` wird in ein 16-Bit-Wort umgewandelt, wobei `FALSE` als `0` und `TRUE` als `1` interpretiert wird. Das Ergebnis wird über den Ausgang `OUT` ausgegeben, und das Ereignis `CNF` wird ausgelöst, um die erfolgreiche Konvertierung zu signalisieren.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher Funktionsblock (`SimpleFB`) und verwendet einen ST-Algorithmus für die Konvertierung.
- Die Konvertierung erfolgt direkt durch die Funktion `BOOL_TO_WORD(IN)`.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Einbettung boolescher Signale in Wort-Datenstrukturen für die Kommunikation mit anderen Systemen.
- Verwendung in Steuerungslogik, wo boolesche Werte als Teil größerer Datenwörter verarbeitet werden müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_BOOL_TO_BYTE` oder `F_BOOL_TO_DWORD` bieten Konvertierungen in andere Datentypen (`BYTE` oder `DWORD`). `F_BOOL_TO_WORD` ist spezifisch für 16-Bit-Wortwerte.

Fazit

Der Funktionsblock `F_BOOL_TO_WORD` bietet eine einfache und effiziente Möglichkeit, boolesche Werte in Wortwerte zu konvertieren. Seine klare Schnittstellenstruktur und direkte Funktionsweise machen ihn ideal für Anwendungen, die solche Konvertierungen erfordern.

BYTE

BYTE Konvertierungen

Hier finden Sie Funktionen zur Konvertierung des Datentyps BYTE.

🛠️ Zugehörige Übungen

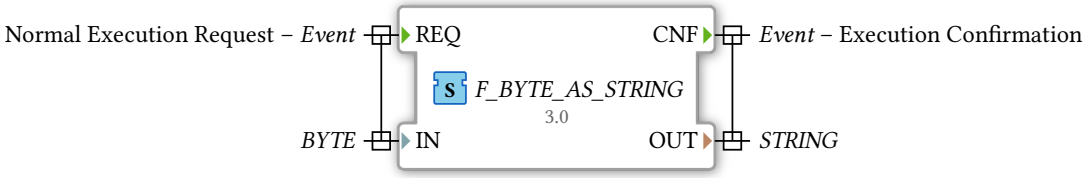
- SIN_TO_PCAN_Callback_Byte
- Uebung_128b
- Uebung_12x_sub

F_BYTE_AS_STRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_BYTE_AS_STRING` dient der Konvertierung eines BYTE-Wertes in einen STRING. Diese Funktionalität ist besonders nützlich, wenn numerische Daten in eine für die Anzeige oder Protokollierung geeignete Textform umgewandelt werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Löst die Konvertierung aus.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Wird nach erfolgreicher Konvertierung ausgelöst.

Daten-Eingänge

- **IN**: BYTE-Wert, der in einen STRING konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Ergebnis der Konvertierung als STRING.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Auslösung des REQ-Ereignisses wird der eingehende BYTE-Wert (IN) in einen STRING konvertiert und über den Ausgang OUT ausgegeben. Gleichzeitig wird das CNF-Ereignis ausgelöst, um die erfolgreiche Ausführung zu signalisieren.

Technische Besonderheiten

- Die Konvertierung erfolgt mittels der Funktion `BYTE_AS_STRING`.
- Der Funktionsblock ist einfach und hat keine internen Zustände oder komplexe Logik.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge oder -maschinen.

Anwendungsszenarien

- Konvertierung von Sensorwerten für die Anzeige auf einem HMI.
- Vorbereitung von Daten für die Protokollierung oder Kommunikation mit anderen Systemen, die String-Daten erwarten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen wie `F_WORD_AS_STRING` oder `F_DWORD_AS_STRING` ist dieser speziell für BYTE-Werte optimiert.
- Einfacher und spezialisierter als generische Konvertierungsbausteine.

Fazit

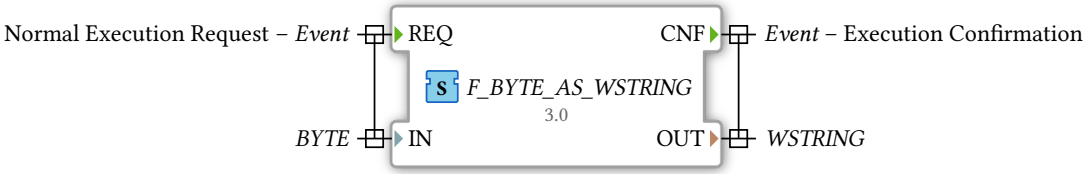
Der `F_BYTE_AS_STRING` Funktionsblock bietet eine einfache und effiziente Möglichkeit, BYTE-Werte in Strings umzuwandeln. Seine Einfachheit macht ihn zuverlässig und leicht in verschiedene Anwendungen integrierbar.

F_BYTE_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_BYTE_AS_WSTRING` dient der Konvertierung eines BYTE-Wertes in einen WSTRING-Wert. Dies ist besonders nützlich in Szenarien, wo Byte-Daten als Unicode-Strings weiterverarbeitet oder ausgegeben werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird mit dem Eingabewert `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Liefert den konvertierten Wert über `OUT`.

Daten-Eingänge

- **IN** (BYTE): Der Byte-Wert, der in einen WSTRING konvertiert werden soll.

Daten-Ausgänge

- **OUT** (WSTRING): Der resultierende Unicode-String nach der Konvertierung.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock nutzt die eingebaute Funktion `BYTE_AS_WSTRING`, um den eingegebenen Byte-Wert in einen WSTRING zu konvertieren. Die Konvertierung wird durch das Ereignis `REQ` ausgelöst und das Ergebnis über `CNF` zurückgemeldet.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne zusätzliche Parameter.
- Der Block ist für die Verwendung in IEC 61131-3-kompatiblen Umgebungen optimiert.

Zustandsübersicht

1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Processing**: Führt die Konvertierung durch.
3. **Completed**: Sendet das `CNF`-Ereignis mit dem Ergebnis.

Anwendungsszenarien

- Konvertierung von Byte-Daten für die Anzeige in Benutzeroberflächen.
- Verarbeitung von Byte-Streams in Unicode-kompatiblen Systemen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu `F_BYTE_AS_STRING` erzeugt dieser Block einen Unicode-String (WSTRING) statt eines ASCII-Strings (STRING).
- Ähnlich zu `F_WORD_AS_WSTRING`, aber für 8-Bit-Daten optimiert.

Fazit

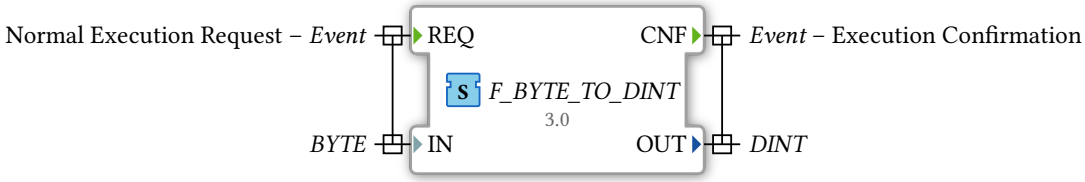
Der `F_BYTE_AS_WSTRING`-Block ist ein einfacher und effizienter Weg, um Byte-Daten in Unicode-Strings umzuwandeln. Seine klare Schnittstelle und direkte Funktionsweise machen ihn ideal für zahlreiche Anwendungen in der Automatisierungstechnik.

F_BYTE_TO_DINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_BYTE_TO_DINT` dient der Konvertierung eines `BYTE`-Wertes in einen `DINT`-Wert. Er ist Teil der `iec61131::conversion`-Bibliothek und ermöglicht eine einfache und effiziente Typumwandlung zwischen diesen beiden Datentypen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Dieser Eingang löst die Ausführung des Funktionsblocks aus. Er ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Dieser Ausgang signalisiert die erfolgreiche Ausführung des Funktionsblocks. Er ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingang erwartet einen `BYTE`-Wert, der in einen `DINT`-Wert umgewandelt werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgang liefert den konvertierten `DINT`-Wert.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `BYTE` zu `DINT` durch, sobald das Ereignis `REQ` empfangen wird. Die Umwandlung erfolgt mittels der Funktion `BYTE_TO_DINT(IN)`, die den Wert des Eingangs `IN` in den entsprechenden `DINT`-Wert umwandelt und am Ausgang `OUT` bereitstellt. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Der Funktionsblock ist einfach aufgebaut und enthält nur einen Algorithmus für die Konvertierung.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Verarbeitungsschritte.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Ausführung erfolgt einmalig bei jedem Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo `BYTE`-Daten in `DINT`-Daten umgewandelt werden müssen.
- Integration in größere Systeme, die eine Typkompatibilität zwischen verschiedenen Datenformaten erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen wie `F_WORD_TO_DINT` oder `F_INT_TO_DINT` ist dieser Funktionsblock speziell auf die Umwandlung von `BYTE` zu `DINT` ausgelegt.
- Er bietet eine einfache und direkte Lösung für diese spezifische Konvertierung.

Fazit

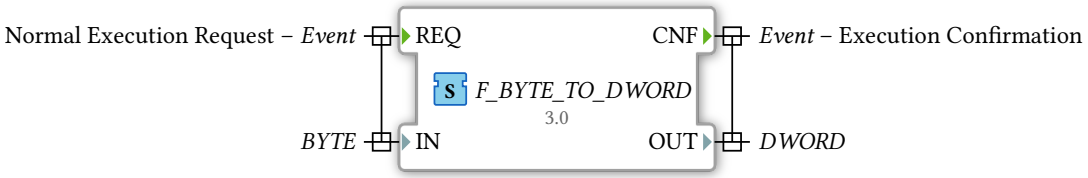
Der `F_BYTE_TO_DINT`-Funktionsblock ist ein nützliches Werkzeug für die Typumwandlung in IEC 61131-3-basierten Steuerungssystemen. Seine einfache und effiziente Funktionsweise macht ihn zu einer zuverlässigen Lösung für Anwendungen, die eine Konvertierung von `BYTE` zu `DINT` erfordern.

F_BYTE_TO_DWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_BYTE_TO_DWORD` dient der Konvertierung eines `BYTE`-Wertes in einen `DWORD`-Wert. Dieser Baustein ist besonders nützlich in Szenarien, bei denen eine Erweiterung der Datenbreite erforderlich ist, beispielsweise bei der Verarbeitung von Sensordaten oder der Kommunikation zwischen Systemen mit unterschiedlichen Datenformaten.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Dieser Eingang ist mit dem Daten-Eingang `IN` gekoppelt.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ausgang ist mit dem Daten-Ausgang `OUT` gekoppelt.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `BYTE`, der in einen `DWORD`-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgangswert vom Typ `DWORD`, der das Ergebnis der Konvertierung enthält.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Auslösung des Ereigniseingangs `REQ` wird der Algorithmus ausgeführt, der den `BYTE`-Wert am Eingang `IN` in einen `DWORD`-Wert konvertiert und das Ergebnis am Ausgang `OUT` ausgibt. Anschließend wird das Ereignis `CNF` ausgelöst, um den erfolgreichen Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Der Funktionsblock ist einfach und effizient, da er direkt die eingebaute Funktion `BYTE_TO_DWORD` nutzt.
- Keine zusätzlichen Zustände oder komplexe Logik erforderlich.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge oder interne Zustände. Die Operation wird atomar bei Auslösung von `REQ` durchgeführt.

Anwendungsszenarien

- Erweiterung von Datenbreiten in Kommunikationsprotokollen.
- Verarbeitung von Sensordaten, die als `BYTE` vorliegen, aber in `DWORD` weiterverarbeitet werden müssen.
- Kompatibilitätsanpassungen zwischen verschiedenen Systemkomponenten.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_WORD_TO_DWORD**: Konvertiert `WORD` zu `DWORD`, ähnlich in der Funktionsweise, aber mit einer anderen Eingangsdatengröße.
- **F_BYTE_TO_WORD**: Konvertiert `BYTE` zu `WORD`, bietet eine geringere Ausgangsdatengröße im Vergleich zu `F_BYTE_TO_DWORD`.

Fazit

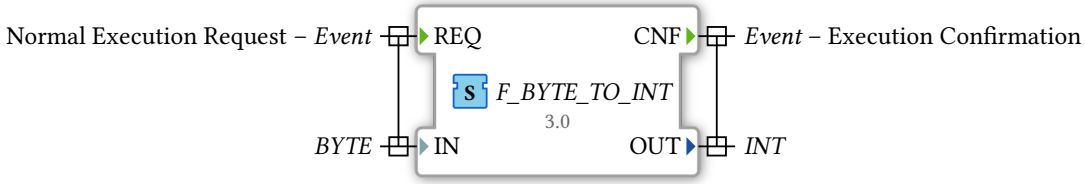
Der `F_BYTE_TO_DWORD` Funktionsblock ist ein einfaches und effizientes Werkzeug zur Konvertierung von `BYTE`- zu `DWORD`-Werten. Seine klare Schnittstelle und direkte Funktionsweise machen ihn ideal für Anwendungen, bei denen eine Datenbreitenerweiterung benötigt wird.

F_BYTE_TO_INT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_BYTE_TO_INT` dient der Konvertierung eines `BYTE`-Wertes in einen `INT`-Wert. Er ist Teil des `iec61131::conversion`-Pakets und ermöglicht eine einfache und effiziente Typumwandlung zwischen diesen beiden Datentypen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung des Funktionsblocks. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN**: Der Eingang vom Typ `BYTE`, der den zu konvertierenden Wert enthält.

Daten-Ausgänge

- **OUT**: Der Ausgang vom Typ `INT`, der das Ergebnis der Konvertierung enthält.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `BYTE` zu `INT` durch, sobald das Ereignis `REQ` empfangen wird. Die Konvertierung erfolgt mittels der integrierten Funktion `BYTE_TO_INT`. Nach erfolgreicher Umwandlung wird das Ergebnis über den Ausgang `OUT` ausgegeben und das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher FB ohne interne Zustandsverwaltung.
- Die Konvertierung erfolgt direkt und ohne Verzögerung.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge oder komplexe Zustandslogik.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo `BYTE`-Werte in `INT`-Werte umgewandelt werden müssen.
- Integration in größere Steuerungssysteme, die unterschiedliche Datentypen verarbeiten müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_WORD_TO_INT` oder `F_DWORD_TO_INT` bieten Konvertierungen für andere Datentypen, jedoch mit demselben Grundprinzip.
- Im Gegensatz zu komplexeren Konvertierungsblöcken bietet `F_BYTE_TO_INT` eine einfache und direkte Lösung für die spezifische Umwandlung von `BYTE` zu `INT`.

Fazit

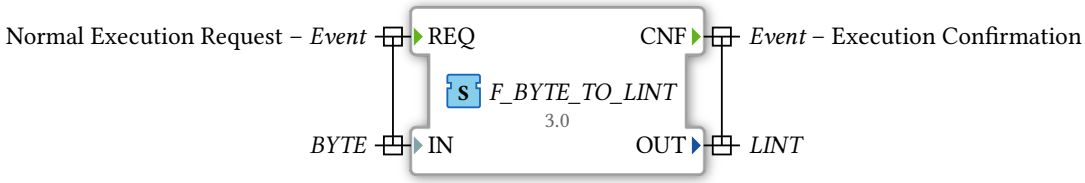
Der `F_BYTE_TO_INT`-Funktionsblock ist ein effizientes und einfach zu verwendendes Werkzeug für die Konvertierung von `BYTE`- in `INT`-Werte. Seine Einfachheit und direkte Funktionsweise machen ihn ideal für Anwendungen, die schnelle und zuverlässige Typumwandlungen erfordern.

F_BYTE_TO_LINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_BYTE_TO_LINT` dient der Konvertierung eines `BYTE`-Wertes in einen `LINT`-Wert. Dieser Baustein ist besonders nützlich in Szenarien, wo eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung der Konvertierung. Dieser Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (`BYTE`): Der Eingangswert, der in den `LINT`-Typ konvertiert werden soll.

Daten-Ausgänge

- **OUT** (`LINT`): Der konvertierte Ausgangswert.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Bei Auslösung des `REQ`-Ereignisses wird der Wert am Eingang `IN` vom Typ `BYTE` in den Typ `LINT` konvertiert und am Ausgang `OUT` ausgegeben. Das `CNF`-Ereignis signalisiert den erfolgreichen Abschluss der Konvertierung.

Technische Besonderheiten

- Der Funktionsblock nutzt die eingebaute Funktion `BYTE_TO_LINT` für die Konvertierung.
- Es handelt sich um einen einfachen Funktionsblock ohne Zustandsautomaten oder komplexe Algorithmen.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge oder komplexe Zustandslogik.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo `BYTE`-Werte in `LINT`-Werte umgewandelt werden müssen.
- Verwendung in Datenverarbeitungsketten, wo unterschiedliche Datentypen verarbeitet werden.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_WORD_TO_LINT` oder `F_DWORD_TO_LINT` bieten ähnliche Funktionalitäten, jedoch für andere Eingangstypen.
- Im Vergleich zu generischen Konvertierungsbausteinen ist `F_BYTE_TO_LINT` spezialisiert und optimiert für die Konvertierung von `BYTE` zu `LINT`.

Fazit

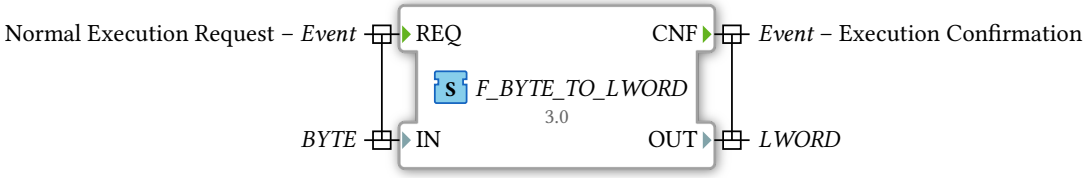
Der `F_BYTE_TO_LINT` Funktionsblock ist ein effizientes und spezialisiertes Werkzeug für die Konvertierung von `BYTE`-Werten in `LINT`-Werte. Seine Einfachheit und direkte Funktionsweise machen ihn zu einer zuverlässigen Wahl in entsprechenden Anwendungsfällen.

F_BYTE_TO_LWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_BYTE_TO_LWORD` dient der Konvertierung eines `BYTE`-Wertes in einen `LWORD`-Wert. Er ist Teil der `iec61131::conversion`-Bibliothek und ermöglicht eine einfache und effiziente Typumwandlung.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung des Funktionsblocks. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert die erfolgreiche Beendigung der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `BYTE`, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgangswert vom Typ `LWORD`, der das Ergebnis der Konvertierung enthält.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt bei einem eingehenden `REQ`-Ereignis die Konvertierung des `BYTE`-Wertes am Eingang `IN` in einen `LWORD`-Wert durch. Das Ergebnis wird am Ausgang `OUT` ausgegeben und durch ein `CNF`-Ereignis bestätigt.

Technische Besonderheiten

- Der Funktionsblock verwendet die `BYTE_TO_LWORD`-Funktion zur Konvertierung.
- Es handelt sich um einen einfachen Funktionsblock ohne Zustandsautomat.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge. Die Konvertierung erfolgt direkt bei einem `REQ`-Ereignis.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo `BYTE`-Werte in `LWORD`-Werte umgewandelt werden müssen.
- Einsatz in Datenverarbeitungsketten, wo unterschiedliche Datentypen verarbeitet werden.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke sind `F_WORD_TO_LWORD` oder `F_DWORD_TO_LWORD`, die andere Datentypen konvertieren.
- Im Gegensatz zu komplexeren Konvertierungsblöcken bietet `F_BYTE_TO_LWORD` eine einfache und direkte Lösung für die spezifische Konvertierung von `BYTE` zu `LWORD`.

Fazit

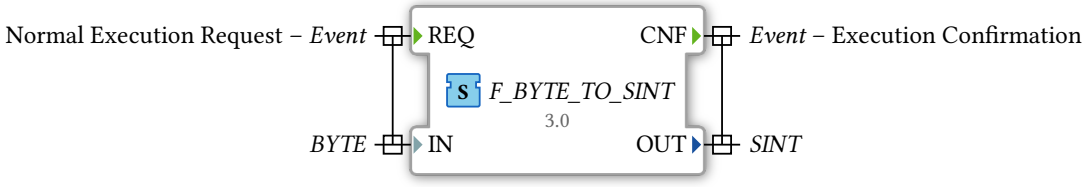
Der `F_BYTE_TO_LWORD`-Funktionsblock ist eine effiziente und einfache Lösung für die Konvertierung von `BYTE`- zu `LWORD`-Werten. Seine klare Schnittstellenstruktur und direkte Funktionsweise machen ihn ideal für den Einsatz in Steuerungsanwendungen.

F_BYTE_TO_SINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_BYTE_TO_SINT` dient der Konvertierung eines Datentyps `BYTE` in den Datentyp `SINT`. Dieser Block ist besonders nützlich in Szenarien, wo eine direkte Typumwandlung ohne zusätzliche Berechnungen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung des Funktionsblocks. Dieser Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert die erfolgreiche Beendigung der Konvertierung. Dieser Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Erwartet einen Eingangswert vom Typ `BYTE`, der in `SINT` umgewandelt werden soll.

Daten-Ausgänge

- **OUT**: Gibt den konvertierten Wert vom Typ `SINT` aus.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Der Funktionsblock führt bei einem eingehenden `REQ`-Ereignis die Konvertierung des `BYTE`-Wertes am Eingang `IN` in einen `SINT`-Wert durch und gibt das Ergebnis über den Ausgang `OUT` aus. Anschließend wird das `CNF`-Ereignis ausgelöst, um den erfolgreichen Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne zusätzliche Parameter oder Einstellungen.
- Der Block ist als einfacher Funktionsblock (SimpleFB) implementiert, was eine effiziente Ausführung garantiert.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Ausführung ist rein ereignisgesteuert und erfolgt bei jedem `REQ`-Ereignis.

Anwendungsszenarien

- Typkonvertierung in Steuerungsanwendungen, wo `BYTE`-Daten in `SINT` umgewandelt werden müssen.
- Integration in größere Funktionsblocknetzwerke, wo unterschiedliche Datentypen verarbeitet werden.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken ist `F_BYTE_TO_SINT` spezialisiert und bietet daher eine höhere Effizienz für diese spezifische Konvertierung.
- Andere Blöcke könnten zusätzliche Funktionen wie Bereichsprüfungen oder Skalierungen bieten, was hier nicht der Fall ist.

Fazit

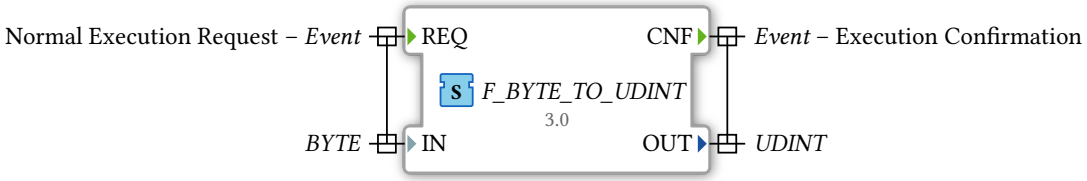
Der `F_BYTE_TO_SINT` Funktionsblock ist ein effizientes Werkzeug für die direkte Konvertierung von `BYTE` zu `SINT`. Seine Einfachheit und Spezialisierung machen ihn ideal für Anwendungen, wo eine schnelle und unkomplizierte Typumwandlung benötigt wird.

F_BYTE_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_BYTE_TO_UDINT` dient zur Konvertierung eines BYTE-Wertes in einen UDINT-Wert. Dieser Baustein ist besonders nützlich in Szenarien, bei denen eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Dieser Ereigniseingang löst die Konvertierung aus. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Dieser Ereignisausgang signalisiert die erfolgreiche Beendigung der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingang vom Typ `BYTE`, der den zu konvertierenden Wert enthält.

Daten-Ausgänge

- **OUT**: Der Ausgang vom Typ `UDINT`, der das Ergebnis der Konvertierung enthält.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Bei Auslösung des Ereigniseingangs `REQ` wird der Algorithmus ausgeführt, der den Wert am Eingang `IN` vom Typ `BYTE` in einen `UDINT`-Wert konvertiert und das Ergebnis am Ausgang `OUT` ausgibt. Anschließend wird das Ereignis `CNF` ausgelöst, um die erfolgreiche Beendigung der Operation zu signalisieren.

Technische Besonderheiten

- Der Funktionsblock verwendet die eingebaute Funktion `BYTE_TO_UDINT` zur Konvertierung.
- Die Konvertierung erfolgt ohne zusätzliche Parameter oder Konfiguration.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Ausführung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, bei denen BYTE-Werte in UDINT-Werte umgewandelt werden müssen.
- Integration in größere Steuerungssysteme, die eine einheitliche Datenverarbeitung erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen wie `F_WORD_TO_UDINT` oder `F_DWORD_TO_UDINT` ist dieser Baustein speziell für die Konvertierung von `BYTE` zu `UDINT` optimiert.
- Ähnliche Bausteine können unterschiedliche Eingangsdatentypen verarbeiten, sind aber ansonsten in ihrer Funktionsweise vergleichbar.

🛠️ Zugehörige Übungen

- Uebung_011a

Fazit

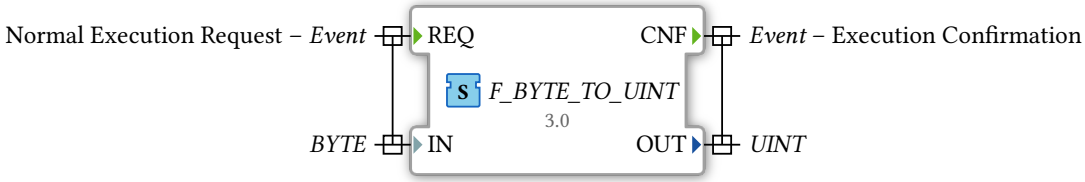
Der `F_BYTE_TO_UDINT` Funktionsblock bietet eine einfache und effiziente Möglichkeit, BYTE-Werte in UDINT-Werte umzuwandeln. Durch seine klare Schnittstellenstruktur und direkte Funktionsweise eignet er sich ideal für Anwendungen, die eine schnelle und zuverlässige Typumwandlung erfordern.

F_BYTE_TO_UINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_BYTE_TO_UINT` dient der Konvertierung eines `BYTE`-Wertes in einen `UINT`-Wert. Dies ist besonders nützlich, wenn Daten unterschiedlicher Typen in einem System verarbeitet werden müssen und eine Typumwandlung erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung des Funktionsblocks. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Bestätigt die erfolgreiche Ausführung des Funktionsblocks. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingang vom Typ `BYTE`, der in einen `UINT`-Wert umgewandelt werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgang vom Typ `UINT`, der das Ergebnis der Konvertierung enthält.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung des `BYTE`-Wertes am Eingang `IN` in einen `UINT`-Wert durch, sobald das Ereignis `REQ` ausgelöst wird. Das Ergebnis wird am Ausgang `OUT` ausgegeben und das Ereignis `CNF` signalisiert die erfolgreiche Ausführung.

Technische Besonderheiten

- Der Funktionsblock ist einfach aufgebaut und enthält nur einen Algorithmus, der die Konvertierung durchführt.
- Die Konvertierung erfolgt mittels der Funktion `BYTE_TO_UINT`, die den `BYTE`-Wert in einen `UINT`-Wert umwandelt.

Zustandsübersicht

Der Funktionsblock hat keinen internen Zustand. Die Ausführung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Typumwandlung in Steuerungs- und Automatisierungssystemen.
- Datenverarbeitung in Systemen, die unterschiedliche Datentypen verwenden.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_WORD_TO_UINT` oder `F_DWORD_TO_UINT` führen ähnliche Konvertierungen durch, jedoch mit anderen Eingangstypen.
- `F_BYTE_TO_UINT` ist spezialisiert auf die Konvertierung von `BYTE` zu `UINT` und daher effizienter für diesen spezifischen Anwendungsfall.

🛠️ Zugehörige Übungen

- Uebung_006c

Fazit

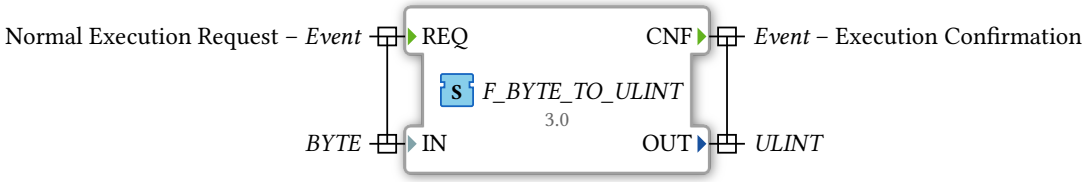
Der Funktionsblock `F_BYTE_TO_UINT` ist ein einfaches und effizientes Werkzeug zur Konvertierung von `BYTE`-Werten in `UINT`-Werte. Seine klare Schnittstellenstruktur und direkte Funktionsweise machen ihn ideal für Anwendungen, bei denen solche Typumwandlungen erforderlich sind.

F_BYTE_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_BYTE_TO_ULINT` dient der Konvertierung eines `BYTE`-Wertes in einen `ULINT`-Wert. Diese Konvertierung ist insbesondere in Szenarien nützlich, in denen Daten unterschiedlicher Größe verarbeitet oder weitergegeben werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Dieser Eingang löst die Ausführung des Funktionsblocks aus. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Dieser Ausgang signalisiert die erfolgreiche Ausführung des Funktionsblocks. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingang erwartet einen `BYTE`-Wert, der in einen `ULINT`-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgang liefert den konvertierten `ULINT`-Wert.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `BYTE` nach `ULINT` durch, sobald das Ereignis `REQ` empfangen wird. Die Konvertierung erfolgt mithilfe der Funktion `BYTE_TO_ULINT`, die den Eingabewert `IN` in den entsprechenden `ULINT`-Wert umwandelt. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst, und der konvertierte Wert steht am Ausgang `OUT` zur Verfügung.

Technische Besonderheiten

- Der Funktionsblock ist einfach aufgebaut und enthält nur einen Algorithmus zur Konvertierung.
- Die Konvertierung erfolgt ohne zusätzliche Parameter oder komplexe Logik.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Ausführung erfolgt unmittelbar nach Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Datenkonvertierung in Steuerungsanwendungen, wo `BYTE`-Werte in größere Datentypen wie `ULINT` umgewandelt werden müssen.
- Integration in größere Systeme, die eine einheitliche Datenverarbeitung in `ULINT` erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_WORD_TO_ULINT` oder `F_DWORD_TO_ULINT` führen Konvertierungen von anderen Datentypen nach `ULINT` durch. Der `F_BYTE_TO_ULINT` ist spezifisch für die Konvertierung von `BYTE`.

Fazit

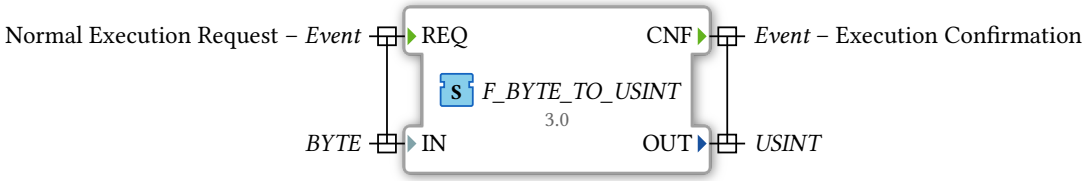
Der `F_BYTE_TO_ULINT`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, `BYTE`-Werte in `ULINT`-Werte umzuwandeln. Seine klare Schnittstelle und direkte Funktionsweise machen ihn zu einem nützlichen Baustein in der Automatisierungstechnik.

F_BYTE_TO_USINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_BYTE_TO_USINT` dient der Konvertierung eines `BYTE`-Wertes in einen `USINT`-Wert (Unsigned Short Integer). Dieser Baustein ist besonders nützlich in Szenarien, wo eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist, z.B. bei der Kommunikation zwischen verschiedenen Systemen oder bei der Datenverarbeitung.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Dieser Ereigniseingang ist mit dem Dateneingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ereignisausgang ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `BYTE`, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgangswert vom Typ `USINT`, der das Ergebnis der Konvertierung enthält.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem er den `BYTE`-Wert am Eingang `IN` in einen `USINT`-Wert umwandelt und das Ergebnis am Ausgang `OUT` bereitstellt. Die Konvertierung erfolgt synchron mit dem Auslösen des `REQ`-Ereignisses und wird durch das `CNF`-Ereignis bestätigt.

Technische Besonderheiten

- Der Funktionsblock verwendet die eingebaute Funktion `BYTE_TO_USINT` zur Konvertierung.
- Es handelt sich um einen einfachen Funktionsblock ohne Zustandsmaschine oder komplexe Logik.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge oder -verwaltung. Die Operation ist atomar und wird bei jedem `REQ`-Ereignis ausgeführt.

Anwendungsszenarien

- Typumwandlung in Kommunikationsprotokollen, wo verschiedene Systeme unterschiedliche Datentypen verwenden.
- Datenverarbeitung in Steuerungssystemen, wo eine Konvertierung zwischen `BYTE` und `USINT` erforderlich ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_WORD_TO_UINT` oder `F_DWORD_TO_UDINT` führen ähnliche Konvertierungen durch, jedoch mit anderen Datentypen.
- `F_BYTE_TO_USINT` ist spezifisch für die Konvertierung von `BYTE` zu `USINT` optimiert und daher effizienter für diesen Anwendungsfall.

Fazit

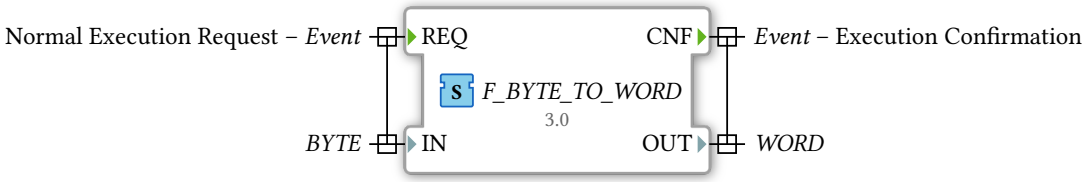
Der `F_BYTE_TO_USINT` Funktionsblock ist ein einfacher, aber effektiver Baustein für die Konvertierung zwischen `BYTE` und `USINT`. Seine Einfachheit und Effizienz machen ihn ideal für Anwendungen, die eine schnelle und zuverlässige Typumwandlung erfordern.

F_BYTE_TO_WORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_BYTE_TO_WORD` dient der Konvertierung eines `BYTE`-Datentyps in einen `WORD`-Datentyp. Diese Konvertierung ist nützlich, wenn Daten in verschiedenen Größen verarbeitet oder weitergegeben werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Dieser Ereigniseingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ereignisausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingang vom Typ `BYTE`, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgang vom Typ `WORD`, der das Ergebnis der Konvertierung enthält.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `BYTE` zu `WORD` durch, sobald das Ereignis `REQ` empfangen wird. Die Konvertierung erfolgt mittels der Funktion `BYTE_TO_WORD(IN)`, die den Eingangswert `IN` in den entsprechenden `WORD`-Wert umwandelt. Das Ergebnis wird am Ausgang `OUT` ausgegeben und das Ereignis `CNF` signalisiert den Abschluss der Operation.

Technische Besonderheiten

- Der Funktionsblock ist einfach aufgebaut und enthält nur einen Algorithmus zur Konvertierung.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Verzögerung.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung wird bei jedem `REQ`-Ereignis durchgeführt und das Ergebnis sofort ausgegeben.

Anwendungsszenarien

- **Datenkonvertierung**: Wenn ein `BYTE`-Wert in einem System verarbeitet werden muss, das `WORD`-Werte erwartet.
- **Schnittstellenanpassung**: Bei der Kommunikation zwischen Systemen, die unterschiedliche Datentypen verwenden.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_WORD_TO_BYTE**: Führt die umgekehrte Konvertierung durch, von `WORD` zu `BYTE`.
- **F_BYTE_TO_INT**: Konvertiert `BYTE` zu `INT`, was eine andere Art der Datentypumwandlung darstellt.

Fazit

Der `F_BYTE_TO_WORD`-Funktionsblock ist ein einfaches und effizientes Werkzeug zur Konvertierung von `BYTE`- zu `WORD`-Werten. Seine klare Schnittstellenstruktur und direkte Funktionsweise machen ihn ideal für Anwendungen, bei denen schnelle und unkomplizierte Datentypkonvertierungen erforderlich sind.

DINT

DINT Konvertierungen

🎧 Podcast

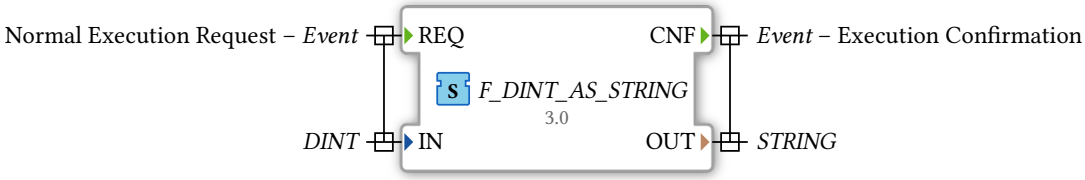
- SINT, INT, DINT: Warum die Wahl des Datentyps über Effizienz und Fehler entscheidet

F_DINT_AS_STRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DINT_AS_STRING` dient der Konvertierung eines 32-Bit-Ganzzahlwerts (DINT) in eine Zeichenkette (STRING). Diese Funktionalität ist besonders nützlich, wenn numerische Werte für die Anzeige oder Protokollierung in einem menschenlesbaren Format benötigt werden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Eingang ausgelöst, wird der Wert am Daten-Eingang `IN` verarbeitet.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang wird zusammen mit dem konvertierten Wert am Daten-Ausgang `OUT` aktiviert.

Daten-Eingänge

- **IN**: Erwartet einen 32-Bit-Ganzzahlwert (DINT), der in eine Zeichenkette umgewandelt werden soll.

Daten-Ausgänge

- **OUT**: Liefert den konvertierten Wert als Zeichenkette (STRING).

Adapter

Dieser Funktionsblock verfügt über keine Adapter-Schnittstellen.

Funktionsweise

Bei Auslösung des `REQ`-Ereignisses wird der Algorithmus gestartet, der den Wert von `IN` vom Typ DINT in einen STRING konvertiert. Das Ergebnis wird an `OUT` ausgegeben und das `CNF`-Ereignis signalisiert die erfolgreiche Konvertierung.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne zusätzliche Parameter.
- Der Funktionsblock ist Teil des Pakets `iec61131::conversion`.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Anzeige von numerischen Werten in Benutzeroberflächen.
- Protokollierung von Messwerten in Textdateien oder Datenbanken.
- Vorbereitung von Daten für die Kommunikation mit Systemen, die nur Zeichenketten verarbeiten können.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Konvertierungsbausteinen bietet `F_DINT_AS_STRING` eine spezialisierte und effiziente Lösung für die Umwandlung von DINT zu STRING. Andere Bausteine könnten zusätzliche Funktionen wie Formatierung oder Locale-Einstellungen bieten, sind jedoch oft komplexer in der Handhabung.

Fazit

Der `F_DINT_AS_STRING` Funktionsblock ist eine einfache und effektive Lösung für die Konvertierung von Ganzzahlen in Zeichenketten. Seine klare Schnittstelle und direkte Funktionsweise machen ihn ideal für Anwendungen, bei denen eine schnelle und unkomplizierte Umwandlung benötigt wird.

F_DINT_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DINT_AS_WSTRING` dient der Konvertierung eines 32-Bit-Ganzzahlwerts (DINT) in eine Unicode-Zeichenkette (WSTRING). Diese Funktionalität ist besonders nützlich, wenn numerische Werte für die Anzeige oder Weiterverarbeitung als Text benötigt werden.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung des Eingabewerts. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (DINT): Der zu konvertierende 32-Bit-Ganzzahlwert.

Daten-Ausgänge

- **OUT** (WSTRING): Die resultierende Unicode-Zeichenkette nach der Konvertierung.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Bei Auslösung des `REQ`-Ereignisses wird der Wert am Eingang `IN` gelesen und mittels der Funktion `DINT_AS_WSTRING` in eine Unicode-Zeichenkette umgewandelt. Das Ergebnis wird am Ausgang `OUT` ausgegeben und das `CNF`-Ereignis signalisiert die erfolgreiche Ausführung.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher Algorithmus ohne Zustandsautomaten.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Parameter.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge oder -verwaltung.

Anwendungsszenarien

- Anzeige von numerischen Werten in Benutzeroberflächen.
- Protokollierung von numerischen Daten in Textform.
- Vorbereitung von Daten für Text-basierte Kommunikationsprotokolle.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu `F_INT_AS_WSTRING` verarbeitet dieser Block 32-Bit statt 16-Bit Ganzzahlen.
- Ähnliche Blöcke wie `F_DINT_AS_STRING` konvertieren in ASCII- statt Unicode-Zeichenketten.

Fazit

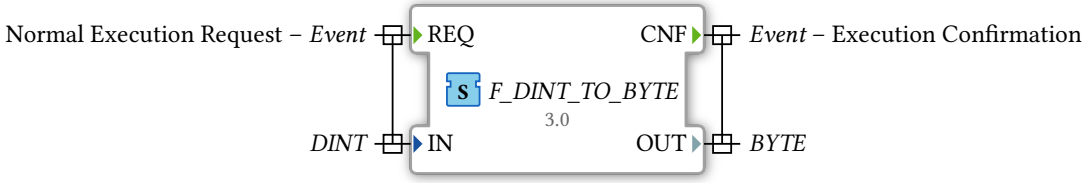
Der `F_DINT_AS_WSTRING`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, 32-Bit-Ganzzahlen in Unicode-Zeichenketten umzuwandeln. Seine klare Schnittstelle und direkte Funktionsweise machen ihn ideal für Anwendungen, die numerische Daten in Textform benötigen.

F_DINT_TO_BYTE

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DINT_TO_BYTE` dient der Konvertierung eines 32-Bit Ganzzahlwerts (DINT) in einen 8-Bit Ganzzahlwert (BYTE). Diese Konvertierung ist besonders nützlich, wenn Daten zwischen Systemen mit unterschiedlichen Wortbreiten ausgetauscht oder verarbeitet werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Eingang ausgelöst, wird der Wert am Daten-Eingang `IN` gelesen und konvertiert.

Ereignis-Ausgänge

- **CNF**: Signalisiert den erfolgreichen Abschluss der Konvertierung. Der konvertierte Wert steht am Daten-Ausgang `OUT` zur Verfügung.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `DINT` (32-Bit Ganzzahl), der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgangswert vom Typ `BYTE` (8-Bit Ganzzahl), der das Ergebnis der Konvertierung enthält.

Adapter

Dieser Funktionsblock verfügt über keine Adapter-Schnittstellen.

Funktionsweise

Bei Auslösung des Ereigniseingangs `REQ` wird der Algorithmus ausgeführt, der den Wert am Eingang `IN` vom Typ `DINT` in einen `BYTE`-Wert konvertiert. Das Ergebnis wird am Ausgang `OUT` ausgegeben und das Ereignis `CNF` signalisiert den Abschluss der Operation.

Der verwendete Algorithmus lautet:

```
OUT := DINT_TO_BYTE(IN);
```

Technische Besonderheiten

- Die Konvertierung von `DINT` zu `BYTE` kann zu Datenverlust führen, wenn der ursprüngliche Wert außerhalb des darstellbaren Bereichs eines `BYTE` (0 bis 255) liegt.
- Der Funktionsblock ist einfach und deterministisch, ohne interne Zustände oder komplexe Logik.

Zustandsübersicht

Der Funktionsblock hat keinen internen Zustand. Die Ausführung erfolgt ausschließlich bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Reduzierung der Datenbreite für die Kommunikation mit Peripheriegeräten, die nur BYTE-Daten verarbeiten können.
- Vorverarbeitung von Daten für Systeme mit begrenzter Speicherkapazität.
- Integration in größere Steuerungssysteme, wo unterschiedliche Datentypen konvertiert werden müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken ist `F_DINT_TO_BYTE` spezialisiert und optimiert für die spezifische Konvertierung von `DINT` zu `BYTE`.
- Ähnliche Blöcke wie `F_INT_TO_BYTE` oder `F_WORD_TO_BYTE` bieten Konvertierungen zwischen anderen Datentypen, aber mit ähnlicher Funktionsweise.

Fazit

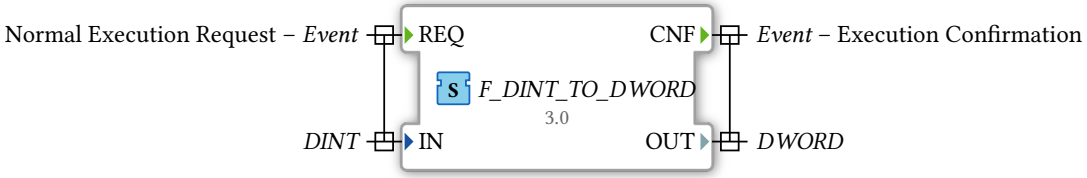
Der `F_DINT_TO_BYTE` Funktionsblock ist ein einfaches und effizientes Werkzeug zur Konvertierung von 32-Bit Ganzzahlen in 8-Bit Werte. Seine Einfachheit und Determinismus machen ihn zuverlässig, aber Anwender sollten potenzielle Datenverluste bei großen Eingangswerten berücksichtigen. Ideal für Anwendungen, wo platzsparende Datendarstellung erforderlich ist.

F_DINT_TO_DWORD

• − ◦ • − ◦ • − ◦ •

Einleitung

Der Funktionsblock `F_DINT_TO_DWORD` dient der Konvertierung eines 32-Bit Ganzzahlwerts (DINT) in einen 32-Bit vorzeichenlosen Wert (DWORD). Diese Konvertierung ist insbesondere dann notwendig, wenn vorzeichenbehaftete Werte in vorzeichenlose Werte umgewandelt werden müssen, beispielsweise für die weitere Verarbeitung in bestimmten Steuerungsalgorithmen oder bei der Kommunikation mit anderen Systemen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung des Eingangswerts `IN` von DINT zu DWORD.

Ereignis-Ausgänge

- **CNF**: Signalisiert den erfolgreichen Abschluss der Konvertierung und stellt den konvertierten Wert `OUT` bereit.

Daten-Eingänge

- **IN** (DINT): Der vorzeichenbehaftete 32-Bit Eingangswert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (DWORD): Der vorzeichenlose 32-Bit Ausgangswert nach der Konvertierung.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von DINT zu DWORD durch, wenn das Ereignis `REQ` ausgelöst wird. Dabei wird der Wert des Eingangs `IN` mittels der Funktion `DINT_TO_DWORD` umgewandelt und am Ausgang `OUT` bereitgestellt. Nach Abschluss der Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne zusätzliche Verzögerung.
- Der Funktionsblock ist für die Verwendung in Echtzeit-Steuerungssystemen optimiert.

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf das `REQ`-Ereignis.
2. **Konvertierungszustand**: Führt die Konvertierung durch und setzt den Ausgang `OUT`.
3. **Bestätigungszustand**: Löst das `CNF`-Ereignis aus und kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- Umwandlung von vorzeichenbehafteten Sensordaten in vorzeichenlose Werte für die weitere Verarbeitung.
- Vorbereitung von Daten für die Kommunikation mit Systemen, die vorzeichenlose Werte erwarten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu generischen Konvertierungsblöcken ist `F_DINT_TO_DWORD` spezialisiert auf die Umwandlung von DINT zu DWORD und bietet daher eine optimierte und fehlerfreie Konvertierung für diesen spezifischen Anwendungsfall.

Fazit

Der Funktionsblock `F_DINT_TO_DWORD` ist ein effizientes und zuverlässiges Werkzeug für die Konvertierung von vorzeichenbehafteten zu vorzeichenlosen 32-Bit Werten. Seine einfache Handhabung und direkte Integration in 4diac-IDE machen ihn zu einer idealen Lösung für entsprechende Anwendungsfälle.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

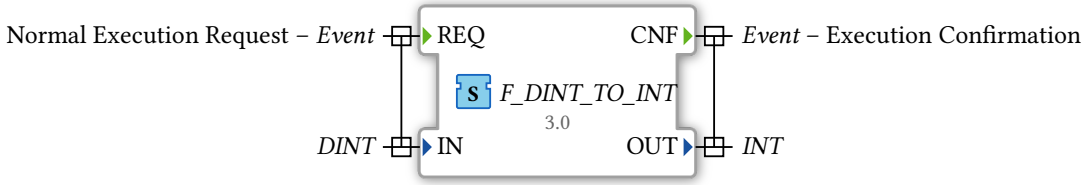
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

F_DINT_TO_INT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DINT_TO_INT` dient der Konvertierung eines 32-Bit-DINT-Wertes in einen 16-Bit-INT-Wert. Diese Konvertierung ist insbesondere in Szenarien erforderlich, in denen Daten zwischen Systemen mit unterschiedlicher Wortbreite ausgetauscht werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ` (Normal Execution Request): Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- `CNF` (Execution Confirmation): Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- `IN` (DINT): Der 32-Bit-DINT-Wert, der konvertiert werden soll.

Daten-Ausgänge

- `OUT` (INT): Der resultierende 16-Bit-INT-Wert nach der Konvertierung.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von einem 32-Bit-DINT-Wert (`IN`) in einen 16-Bit-INT-Wert (`OUT`) durch, sobald das Ereignis `REQ` ausgelöst wird. Die Konvertierung erfolgt durch die Funktion `DINT_TO_INT`. Nach Abschluss der Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Die Konvertierung kann zu einem Datenverlust führen, wenn der ursprüngliche DINT-Wert außerhalb des darstellbaren Bereichs eines INT-Wertes liegt (-32768 bis 32767).
- Der Funktionsblock ist Teil des Pakets `iec61131::conversion`.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Datenkonvertierung in Steuerungssystemen mit unterschiedlicher Wortbreite.
- Integration von Systemen, die unterschiedliche Datentypen verwenden.
- Reduzierung der Speicherbelegung, wenn die höhere Präzision eines DINT nicht benötigt wird.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsbausteinen ist `F_DINT_TO_INT` spezialisiert auf die Konvertierung von DINT zu INT und bietet daher eine einfache und direkte Lösung für diesen spezifischen Anwendungsfall.
- Andere Konvertierungsbausteine, wie `F_INT_TO_DINT`, führen entgegengesetzte Konvertierungen durch.

Fazit

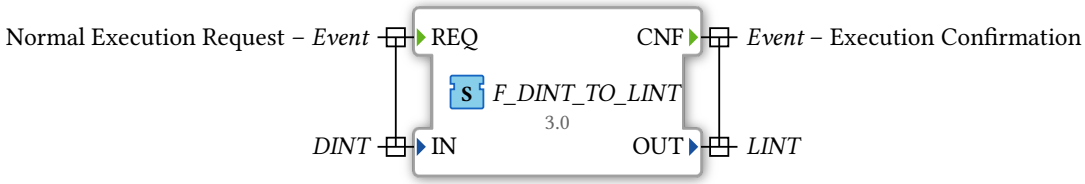
Der Funktionsblock `F_DINT_TO_INT` ist ein effizientes Werkzeug für die Konvertierung von 32-Bit-DINT-Werten in 16-Bit-INT-Werte. Seine einfache und spezialisierte Funktionsweise macht ihn ideal für Anwendungen, in denen solche Konvertierungen regelmäßig erforderlich sind. Bei der Verwendung ist jedoch auf den möglichen Datenverlust bei großen Werten zu achten.

F_DINT_TO_LINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DINT_TO_LINT` dient der Konvertierung eines 32-Bit Ganzzahlwerts (DINT) in einen 64-Bit Ganzzahlwert (LINT). Dieser Baustein ist besonders nützlich in Szenarien, bei denen eine Erweiterung des Wertebereichs oder eine Typanpassung für nachfolgende Verarbeitungsschritte erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Dieser Ereigniseingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den erfolgreichen Abschluss der Konvertierung. Dieser Ereignisausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (DINT): Der 32-Bit Ganzzahlwert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (LINT): Der resultierende 64-Bit Ganzzahlwert nach der Konvertierung.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Auslösung des `REQ`-Ereignisses wird der Wert des `IN`-Eingangs in den `OUT`-Ausgang kopiert. Dabei erfolgt eine automatische Typumwandlung von DINT zu LINT, wobei der numerische Wert erhalten bleibt.

Technische Besonderheiten

- Die Konvertierung erfolgt ohne Datenverlust, da der LINT-Typ einen größeren Wertebereich als DINT unterstützt.
- Der Algorithmus ist in Structured Text (ST) implementiert und besteht aus einer einfachen Zuweisung.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Erweiterung des Wertebereichs für nachfolgende Berechnungen
- Typanpassung in heterogenen Systemen
- Datenaufbereitung für Schnittstellen mit unterschiedlichen Anforderungen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu generischen Konvertierungsbausteinen bietet `F_DINT_TO_LINT` eine spezialisierte und typsichere Lösung für die spezifische Konvertierung zwischen DINT und LINT. Andere Bausteine könnten allgemeinere Konvertierungsmöglichkeiten bieten, sind aber möglicherweise weniger effizient oder erfordern zusätzliche Konfiguration.

Fazit

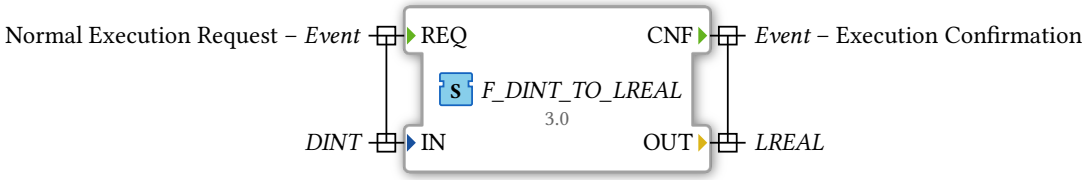
Der `F_DINT_TO_LINT` Funktionsblock stellt eine einfache und effiziente Lösung für die Konvertierung von 32-Bit zu 64-Bit Ganzzahlen dar. Seine klare Schnittstelle und direkte Funktionsweise machen ihn zu einem zuverlässigen Baustein in Steuerungssystemen, wo solche Typkonvertierungen benötigt werden.

F_DINT_TO_LREAL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DINT_TO_LREAL` dient der Konvertierung eines 32-Bit-Ganzzahlwertes (DINT) in einen 64-Bit-Gleitkommawert (LREAL). Diese Konvertierung ist insbesondere in Steuerungssystemen notwendig, wo unterschiedliche Datentypen in verschiedenen Teilen der Anwendung verarbeitet werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Eingang ausgelöst, wird der Wert am Daten-Eingang `IN` gelesen und konvertiert.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der konvertierte Wert steht am Daten-Ausgang `OUT` zur Verfügung.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `DINT`, der in einen `LREAL`-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgangswert vom Typ `LREAL`, der das Ergebnis der Konvertierung darstellt.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt eine direkte Typumwandlung durch. Bei Auslösung des Ereignisses `REQ` wird der Wert von `IN` gelesen und in den Datentyp `LREAL` umgewandelt. Das Ergebnis wird an `OUT` ausgegeben und das Ereignis `CNF` signalisiert die erfolgreiche Konvertierung.

Technische Besonderheiten

- Die Konvertierung erfolgt ohne zusätzliche Skalierung oder Offset-Anpassung.
- Der Funktionsblock ist deterministisch und hat keine internen Zustände.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Umwandlung von Ganzzahlen in Gleitkommazahlen für mathematische Berechnungen.
- Integration in Steuerungssysteme, wo unterschiedliche Datentypen verarbeitet werden müssen.
- Verwendung in Regelungstechnik, wo präzise Gleitkommaberechnungen erforderlich sind.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsbausteinen ist `F_DINT_TO_LREAL` spezialisiert auf die Umwandlung von `DINT` zu `LREAL` und bietet daher eine höhere Effizienz und Lesbarkeit im Code.
- Andere Bausteine könnten zusätzliche Funktionen wie Skalierung oder Begrenzung bieten, was hier nicht der Fall ist.

Fazit

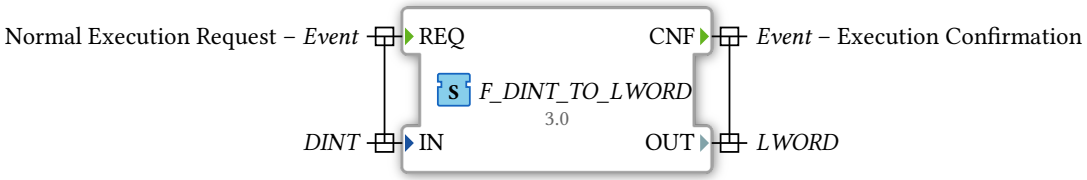
Der `F_DINT_TO_LREAL` Funktionsblock ist ein einfacher und effizienter Baustein zur Konvertierung von Ganzzahlen in Gleitkommazahlen. Seine klare Schnittstelle und deterministische Funktionsweise machen ihn ideal für den Einsatz in Steuerungs- und Regelungssystemen.

F_DINT_TO_LWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DINT_TO_LWORD` dient der Konvertierung eines 32-Bit Integer-Werts (DINT) in einen 64-Bit unsigned Integer-Wert (LWORD). Dieser Baustein ist besonders nützlich in Szenarien, wo eine Erweiterung des Wertebereichs oder eine Typanpassung für nachfolgende Verarbeitungsschritte erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Eingang ausgelöst, liest der Funktionsblock den Wert am Daten-Eingang `IN` und führt die Konvertierung durch.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Gleichzeitig wird der konvertierte Wert am Daten-Ausgang `OUT` ausgegeben.

Daten-Eingänge

- **IN** (DINT): Der 32-Bit Integer-Wert, der in einen 64-Bit unsigned Integer-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT** (LWORD): Der resultierende 64-Bit unsigned Integer-Wert nach der Konvertierung.

Adapter

Dieser Funktionsblock verfügt über keine Adapter-Schnittstellen.

Funktionsweise

Bei Auslösung des Ereigniseingangs `REQ` wird der Algorithmus des Funktionsblocks aktiviert. Dieser liest den Wert am Eingang `IN`, konvertiert ihn von DINT zu LWORD und legt das Ergebnis am Ausgang `OUT` ab. Anschließend wird das Ereignis `CNF` ausgelöst, um den erfolgreichen Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Die Konvertierung erfolgt durch die direkte Zuweisung des DINT-Werts zu einem LWORD, wobei der Wert entsprechend erweitert wird.
- Der Funktionsblock ist deterministisch und führt keine zusätzlichen Berechnungen oder Checks durch.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Operation wird bei jedem `REQ`-Ereignis neu ausgeführt.

Anwendungsszenarien

- Erweiterung des Wertebereichs für nachfolgende Operationen, die 64-Bit Werte erfordern.
- Typkonvertierung in heterogenen Systemen, wo verschiedene Datentypen verarbeitet werden müssen.
- Einsatz in Steuerungssystemen, wo eine höhere Präzision oder ein größerer Wertebereich benötigt wird.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu generischen Konvertierungsbausteinen ist `F_DINT_TO_LWORD` spezialisiert auf die Konvertierung von DINT zu LWORD.
- Ähnliche Bausteine wie `F_INT_TO_DINT` oder `F_LINT_TO_LWORD` bieten Konvertierungen zwischen anderen Integer-Typen.

Fazit

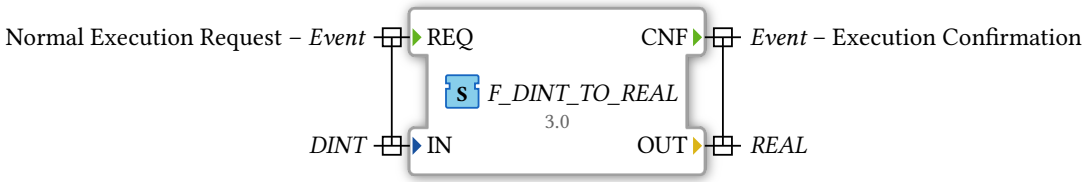
Der `F_DINT_TO_LWORD` Funktionsblock ist ein einfacher, aber effektiver Baustein für die Konvertierung von 32-Bit zu 64-Bit Integer-Werten. Seine deterministische Funktionsweise und klare Schnittstelle machen ihn zu einer verlässlichen Komponente in Steuerungs- und Automatisierungslösungen.

F_DINT_TO_REAL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DINT_TO_REAL` dient der Konvertierung eines ganzzahligen Wertes vom Typ `DINT` in einen Fließkommawert vom Typ `REAL`. Diese Konvertierung ist in vielen Steuerungs- und Automatisierungsanwendungen notwendig, wenn Daten zwischen verschiedenen Systemen oder Komponenten ausgetauscht werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Dieser Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `DINT`, der in einen `REAL`-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der konvertierte Ausgangswert vom Typ `REAL`.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` ausgelöst wird. Der Algorithmus `REQ` wandelt den Eingangswert `IN` vom Typ `DINT` in einen `REAL`-Wert um und speichert das Ergebnis in `OUT`. Anschließend wird das Ereignis `CNF` ausgelöst, um den Abschluss der Konvertierung zu signalisieren.

Technische Besonderheiten

- Der Funktionsblock ist einfach und effizient implementiert, ohne zusätzliche Zustände oder komplexe Logik.
- Die Konvertierung erfolgt direkt durch die Funktion `DINT_TO_REAL`.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt sofort bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Umwandlung von ganzzahligen Sensorwerten in Fließkommawerte für weitere Berechnungen.
- Datenkonvertierung zwischen Systemen, die unterschiedliche Datentypen verwenden.
- Integration in Steuerungsalgorithmen, die Fließkommawerte erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_INT_TO_REAL**: Konvertiert `INT` zu `REAL`, hat aber einen kleineren Eingangsbereich im Vergleich zu `DINT`.
- **F_LINT_TO_REAL**: Konvertiert `LINT` zu `REAL`, hat einen größeren Eingangsbereich, ist aber in den meisten Fällen nicht notwendig, wenn `DINT` ausreicht.

Fazit

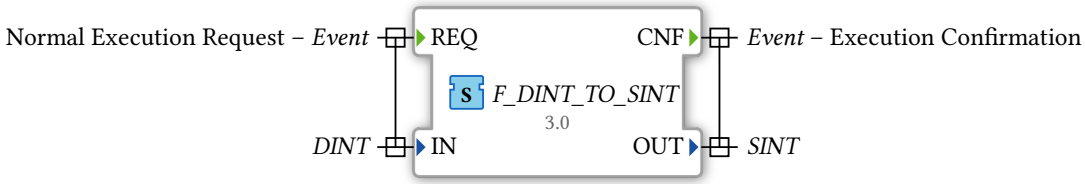
Der Funktionsblock `F_DINT_TO_REAL` ist ein einfaches und effizientes Werkzeug zur Konvertierung von `DINT`-Werten in `REAL`-Werte. Er ist besonders nützlich in Anwendungen, bei denen Daten zwischen verschiedenen Systemen oder Komponenten mit unterschiedlichen Datentypen ausgetauscht werden müssen. Die direkte und zustandslose Implementierung macht ihn zuverlässig und leicht integrierbar.

F_DINT_TO_SINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DINT_TO_SINT` dient der Konvertierung eines 32-Bit Ganzzahlwertes (DINT) in einen 8-Bit Ganzzahlwert (SINT). Dieser Baustein ist besonders nützlich in Szenarien, wo eine Typumwandlung zwischen unterschiedlich großen Integer-Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Konvertierung aus. Wird mit dem Eingabewert `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Liefert den konvertierten Wert über `OUT`.

Daten-Eingänge

- **IN** (DINT): Der 32-Bit Ganzzahlwert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (SINT): Der resultierende 8-Bit Ganzzahlwert nach der Konvertierung.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `DINT` zu `SINT` durch, sobald das Ereignis `REQ` ausgelöst wird. Der Algorithmus nutzt die eingebaute Funktion `DINT_TO_SINT`, um die Umwandlung durchzuführen. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` mit dem Ergebnis `OUT` ausgegeben.

Technische Besonderheiten

- **Typumwandlung**: Die Konvertierung kann zu Datenverlust führen, wenn der ursprüngliche `DINT`-Wert außerhalb des darstellbaren Bereichs von `SINT` (-128 bis 127) liegt.
- **Einfache Implementierung**: Der Baustein ist als einfacher Funktionsblock (SimpleFB) ohne interne Zustände implementiert.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Ausführung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- **Datenreduktion**: Wenn nur die unteren 8 Bit eines 32-Bit-Wertes relevant sind.
- **Kompatibilität**: Anpassung von Werten für Systeme oder Komponenten, die nur `SINT` verarbeiten können.
- **Protokollierung**: Speicherplatzoptimierte Speicherung von Werten, bei denen die höheren Bits nicht benötigt werden.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_INT_TO_SINT**: Ähnlicher Baustein, aber für 16-Bit zu 8-Bit Konvertierung.
- **F_ANY_TO_**: Generische Konvertierungsbausteine, die flexibler, aber möglicherweise weniger effizient sind.

Fazit

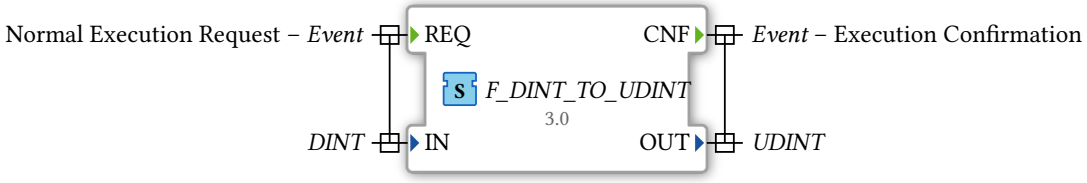
Der `F_DINT_TO_SINT` Funktionsblock bietet eine einfache und effiziente Möglichkeit, 32-Bit Ganzzahlen in 8-Bit Ganzzahlen umzuwandeln. Seine Verwendung ist besonders in ressourcenbeschränkten Umgebungen oder bei der Anpassung von Datentypen zwischen verschiedenen Systemkomponenten sinnvoll. Beachten Sie jedoch die möglichen Datenverluste bei großen Eingabewerten.

F_DINT_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DINT_TO_UDINT` dient der Konvertierung eines 32-Bit vorzeichenbehafteten Integer-Werts (DINT) in einen 32-Bit vorzeichenlosen Integer-Wert (UDINT). Diese Konvertierung ist nützlich, wenn Daten zwischen Systemen ausgetauscht werden, die unterschiedliche Datentypen verwenden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Eingang ausgelöst, wird der Wert am Daten-Eingang `IN` gelesen und konvertiert.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der konvertierte Wert steht am Daten-Ausgang `OUT` zur Verfügung.

Daten-Eingänge

- **IN**: Der zu konvertierende DINT-Wert (32-Bit vorzeichenbehafteter Integer).

Daten-Ausgänge

- **OUT**: Der konvertierte UDINT-Wert (32-Bit vorzeichenloser Integer).

Adapter

Es werden keine Adapter unterstützt.

Funktionsweise

Bei Auslösung des Ereigniseingangs `REQ` wird der Algorithmus des Funktionsblocks ausgeführt. Dabei wird der Wert am Eingang `IN` vom Typ `DINT` in einen `UDINT`-Wert konvertiert und am Ausgang `OUT` ausgegeben. Anschließend wird das Ereignis `CNF` ausgelöst, um den erfolgreichen Abschluss der Konvertierung zu signalisieren.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt durch die Funktion `DINT_TO_UDINT`.
- Es gibt keine zusätzliche Fehlerbehandlung für Überläufe oder ungültige Werte.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung von `REQ`.

Anwendungsszenarien

- Konvertierung von Sensorwerten, die als vorzeichenbehaftete Integer vorliegen, für Systeme, die vorzeichenlose Werte erwarten.
- Datenaufbereitung für Kommunikationsprotokolle, die vorzeichenlose Integer verwenden.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_INT_TO_UINT**: Konvertiert 16-Bit vorzeichenbehaftete Integer in vorzeichenlose Integer.
- **F_LINT_TO_ULINT**: Konvertiert 64-Bit vorzeichenbehaftete Integer in vorzeichenlose Integer.
- **F_SINT_TO_USINT**: Konvertiert 8-Bit vorzeichenbehaftete Integer in vorzeichenlose Integer.

🛠️ Zugehörige Übungen

- Uebung_009a

Fazit

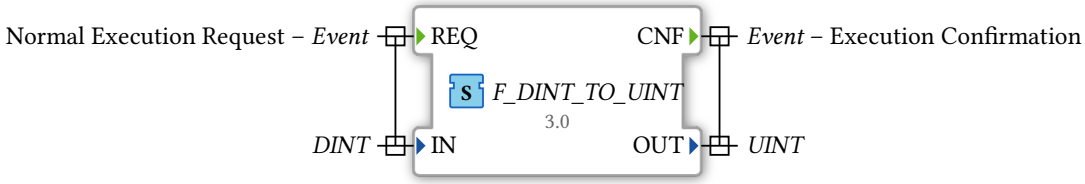
Der Funktionsblock `F_DINT_TO_UDINT` bietet eine einfache und effiziente Möglichkeit, vorzeichenbehaftete 32-Bit Integer in vorzeichenlose 32-Bit Integer zu konvertieren. Er ist besonders nützlich in Szenarien, wo Daten zwischen Systemen mit unterschiedlichen Datentypanforderungen ausgetauscht werden müssen.

F_DINT_TO_UINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DINT_TO_UINT` dient der Konvertierung eines 32-Bit Integer-Wertes (DINT) in einen vorzeichenlosen 32-Bit Integer-Wert (UINT). Dieser Block ist besonders nützlich in Szenarien, wo eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung des Funktionsblocks. Dieser Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert die erfolgreiche Ausführung der Konvertierung. Dieser Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `DINT`, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgangswert vom Typ `UINT`, der das Ergebnis der Konvertierung darstellt.

Adapter

Es sind keine Adapter für diesen Funktionsblock definiert.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `DINT` zu `UINT` durch, wenn das Ereignis `REQ` empfangen wird. Die Konvertierung erfolgt mittels der eingebauten Funktion `DINT_TO_UINT`. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst und der konvertierte Wert über den Ausgang `OUT` ausgegeben.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne zusätzliche Verzögerung.
- Der Block unterstützt keine Fehlerbehandlung für ungültige Eingabewerte.

Zustandsübersicht

1. **Idle**: Wartet auf das Ereignis `REQ`.
2. **Konvertierung**: Führt die Konvertierung durch und gibt das Ergebnis über `OUT` aus, gefolgt von einem `CNF`-Ereignis.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo vorzeichenbehaftete Werte in vorzeichenlose Werte umgewandelt werden müssen.
- Datenverarbeitung in industriellen Automatisierungssystemen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken ist `F_DINT_TO_UINT` spezialisiert auf die Umwandlung zwischen `DINT` und `UINT` und bietet daher eine einfachere und direktere Lösung für diesen spezifischen Anwendungsfall.

Fazit

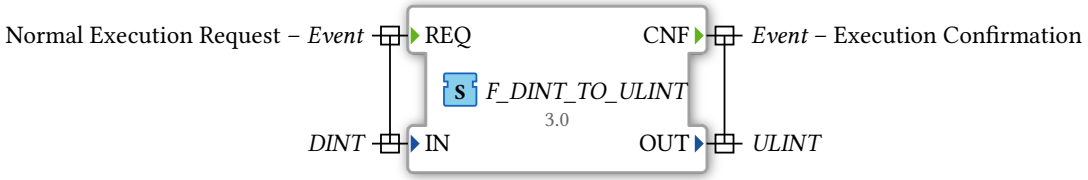
Der `F_DINT_TO_UINT` Funktionsblock ist ein effizientes Werkzeug für die spezifische Konvertierung zwischen `DINT` und `UINT` Datentypen. Seine Einfachheit und direkte Funktionsweise machen ihn ideal für Anwendungen, die eine schnelle und zuverlässige Typumwandlung erfordern.

F_DINT_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DINT_TO_ULINT` dient der Konvertierung eines 32-Bit vorzeichenbehafteten Integer-Werts (`DINT`) in einen 64-Bit vorzeichenlosen Integer-Wert (`ULINT`). Dieser Baustein ist besonders nützlich, wenn Daten zwischen Systemen mit unterschiedlichen Datentypen ausgetauscht werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Bei Aktivierung dieses Ereignisses wird der Wert am Daten-Eingang `IN` gelesen und konvertiert.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieses Ereignis wird zusammen mit dem konvertierten Wert am Daten-Ausgang `OUT` ausgegeben.

Daten-Eingänge

- **IN** (`DINT`): Der vorzeichenbehaftete 32-Bit Integer-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (`ULINT`): Der vorzeichenlose 64-Bit Integer-Wert, der das Ergebnis der Konvertierung enthält.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `DINT` zu `ULINT` durch, sobald das Ereignis `REQ` empfangen wird. Der Algorithmus `REQ` wird ausgeführt, wobei der Wert von `IN` mittels der Funktion `DINT_TO_ULINT` konvertiert und an `OUT` ausgegeben wird. Anschließend wird das Ereignis `CNF` ausgelöst, um den erfolgreichen Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Der Baustein unterstützt die direkte Konvertierung ohne zusätzliche Parameter.
- Die Konvertierung erfolgt ohne Datenverlust, da der `ULINT`-Typ einen größeren Wertebereich als `DINT` abdeckt.

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf das `REQ`-Ereignis.
2. **Konvertierungszustand**: Führt die Konvertierung durch und setzt den Ausgang `OUT`.
3. **Bestätigungszustand**: Löst das `CNF`-Ereignis aus und kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- Datenkonvertierung in Steuerungssystemen, die unterschiedliche Datentypen verwenden.
- Integration von Komponenten, die vorzeichenlose 64-Bit Werte erfordern.
- Erweiterung der Kompatibilität zwischen verschiedenen Systemen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_INT_TO_ULINT**: Konvertiert einen kleineren vorzeichenbehafteten Integer (`INT`) zu `ULINT`.
- **F_LINT_TO_ULINT**: Konvertiert einen 64-Bit vorzeichenbehafteten Integer (`LINT`) zu `ULINT`.
- **F_DINT_TO_LINT**: Konvertiert `DINT` zu einem vorzeichenbehafteten 64-Bit Integer (`LINT`).

Fazit

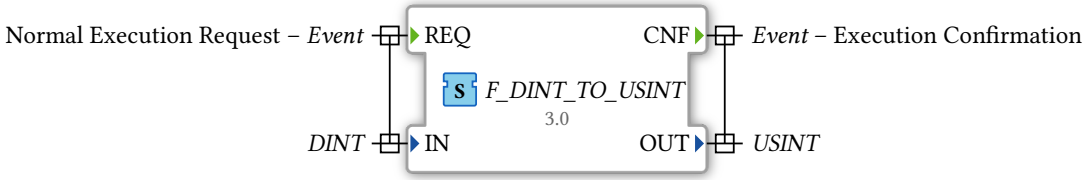
Der `F_DINT_TO_ULINT`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, vorzeichenbehaftete 32-Bit Integer-Werte in vorzeichenlose 64-Bit Integer-Werte umzuwandeln. Durch seine klare Schnittstelle und zuverlässige Funktionsweise eignet er sich ideal für Anwendungen, die eine sichere und verlustfreie Datentypkonvertierung erfordern.

F_DINT_TO_USINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DINT_TO_USINT` dient der Konvertierung eines 32-Bit Ganzzahlwerts (DINT) in einen 8-Bit vorzeichenlosen Ganzzahlwert (USINT). Diese Konvertierung ist nützlich, wenn Daten zwischen Systemen oder Komponenten ausgetauscht werden müssen, die unterschiedliche Datentypen verwenden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Dieser Ereigniseingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ereignisausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `DINT` (32-Bit Ganzzahl), der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgangswert vom Typ `USINT` (8-Bit vorzeichenlose Ganzzahl), der das Ergebnis der Konvertierung enthält.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Auslösung des Ereigniseingangs `REQ` wird der Algorithmus ausgeführt, der den Wert am Eingang `IN` vom Typ `DINT` in einen Wert vom Typ `USINT` konvertiert und das Ergebnis am Ausgang `OUT` ausgibt. Anschließend wird der Ereignisausgang `CNF` aktiviert, um den erfolgreichen Abschluss der Konvertierung zu signalisieren.

Technische Besonderheiten

- Die Konvertierung erfolgt durch die eingebaute Funktion `DINT_TO_USINT`.
- Es ist zu beachten, dass bei der Konvertierung von `DINT` zu `USINT` ein möglicher Datenverlust auftreten kann, da der Wertebereich von `USINT` (0 bis 255) kleiner ist als der von `DINT` (-2.147.483.648 bis 2.147.483.647).

Zustandsübersicht

1. **Idle-Zustand**: Der Funktionsblock wartet auf das Ereignis `REQ`.
2. **Aktiver Zustand**: Bei Empfang von `REQ` wird die Konvertierung durchgeführt und `CNF` ausgelöst.
3. **Abschlusszustand**: Nach der Konvertierung kehrt der Funktionsblock in den Idle-Zustand zurück.

Anwendungsszenarien

- Datenkonvertierung in Steuerungssystemen, wenn unterschiedliche Datentypen verwendet werden.
- Reduzierung der Speicherbelegung, wenn nur kleine vorzeichenlose Werte benötigt werden.
- Schnittstellenanpassung zwischen verschiedenen Systemkomponenten.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_INT_TO_USINT**: Konvertiert einen 16-Bit Ganzzahlwert (INT) in einen 8-Bit vorzeichenlosen Ganzzahlwert (USINT). Im Vergleich dazu verarbeitet `F_DINT_TO_USINT` größere Eingangswerte.
- **F_LINT_TO_USINT**: Konvertiert einen 64-Bit Ganzzahlwert (LINT) in einen 8-Bit vorzeichenlosen Ganzzahlwert (USINT). `F_DINT_TO_USINT` ist für kleinere Eingangsbereiche ausgelegt.

Fazit

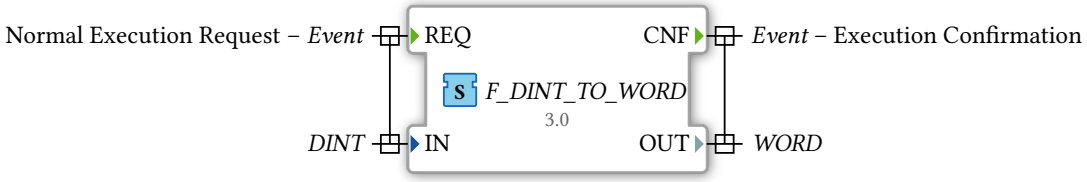
Der Funktionsblock `F_DINT_TO_USINT` bietet eine einfache und effiziente Möglichkeit, 32-Bit Ganzzahlen in 8-Bit vorzeichenlose Ganzzahlen zu konvertieren. Er ist besonders nützlich in Szenarien, wo Speicherplatz optimiert oder Daten zwischen Systemen mit unterschiedlichen Datentypen ausgetauscht werden müssen. Aufgrund des möglichen Datenverlusts sollte jedoch sichergestellt werden, dass die Eingangswerte innerhalb des zulässigen Bereichs von `USINT` liegen.

F_DINT_TO_WORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DINT_TO_WORD` dient der Konvertierung eines 32-Bit Ganzzahlwertes (DINT) in einen 16-Bit Wortwert (WORD). Diese Konvertierung ist besonders nützlich, wenn Daten zwischen Systemen mit unterschiedlicher Wortbreite ausgetauscht werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Eingang ausgelöst, wird der Wert am Daten-Eingang `IN` gelesen und konvertiert.

Ereignis-Ausgänge

- **CNF**: Signalisiert die erfolgreiche Beendigung der Konvertierung. Der konvertierte Wert steht am Daten-Ausgang `OUT` zur Verfügung.

Daten-Eingänge

- **IN**: Der zu konvertierende 32-Bit Ganzzahlwert (DINT).

Daten-Ausgänge

- **OUT**: Der konvertierte 16-Bit Wortwert (WORD).

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Bei Auslösung des Ereigniseingangs `REQ` wird der Algorithmus ausgeführt, der den Wert von `IN` von DINT in WORD konvertiert und das Ergebnis an `OUT` ausgibt. Anschließend wird das Ereignis `CNF` ausgelöst, um den Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Die Konvertierung schneidet die oberen 16 Bit des DINT-Wertes ab, da ein WORD nur 16 Bit speichern kann. Es ist daher wichtig sicherzustellen, dass der ursprüngliche DINT-Wert innerhalb des darstellbaren Bereichs eines WORD liegt (0 bis 65535), um Datenverlust zu vermeiden.
- Der Funktionsblock ist deterministisch und hat keine internen Zustände.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock ohne interne Zustände handelt, gibt es keine Zustandsübergänge. Der Block führt die Konvertierung bei jedem `REQ`-Ereignis aus und signalisiert den Abschluss mit `CNF`.

Anwendungsszenarien

- Kommunikation zwischen Systemen, die unterschiedliche Wortbreiten verwenden.
- Anpassung von Daten für Peripheriegeräte, die nur WORD-Daten verarbeiten können.
- Einsatz in Steuerungssystemen, wo Speicherplatz optimiert werden muss.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken wie `F_ANY_TO_ANY` ist `F_DINT_TO_WORD` spezialisiert und daher effizienter, wenn genau diese Konvertierung benötigt wird.
- Ähnliche Blöcke wie `F_INT_TO_WORD` arbeiten mit kleineren Ganzzahltypen, während `F_DINT_TO_WORD` mit dem größeren DINT-Typ umgeht.

Fazit

Der `F_DINT_TO_WORD` Funktionsblock ist ein spezialisiertes und effizientes Werkzeug für die Konvertierung von 32-Bit Ganzzahlen in 16-Bit Wörter. Seine Einfachheit und Determinismus machen ihn zu einer verlässlichen Komponente in Steuerungsanwendungen, insbesondere bei der Datenanpassung zwischen unterschiedlichen Systemen.

DT

DT Konvertierungen

🎧 Podcast

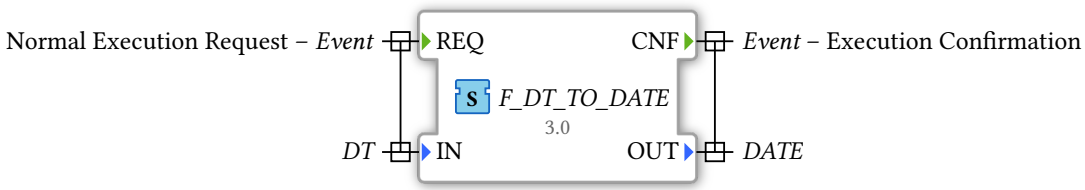
- ISOBUS Button: Mehr als nur ein Klick – Die Standardisierung der Landtechnik
- Schalterlogik verstehen: So funktioniert ein Toggle Flip-Flop mit logiBUS® – einfache Steuerung in der Landtechnik
- Als Landtechnik-Spezialist durch die Hölle: Wie Lanz-Wery Krieg, Besatzung und Hyperinflation überlebte – Einblicke in Original-Geschäftsberichte 1915-1922
- Altbayerisch für Einsteiger: Von Gratler-Schnupfen und Stadthodern – Eine Laute-Reise durch Lektion 3C
- CAN-Bus-Diagnose: Jenseits der Daten – Die Wahrheit über Fehler in der Landtechnik

F_DT_TO_DATE

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DT_TO_DATE` dient der Konvertierung eines Datums-Zeit-Werts (DT) in ein reines Datum (DATE). Diese Funktionalität ist besonders nützlich, wenn in Steuerungsanwendungen nur das Datum ohne Zeitinformation benötigt wird.



Schnittstellenstruktur

Ereignis-Eingänge
- `REQ`: Startet die Ausführung der Konvertierung. Verbunden mit dem Daten-Eingang `IN`.

Ereignis-Ausgänge
- `CNF`: Signalisiert den Abschluss der Konvertierung. Verbunden mit dem Daten-Ausgang `OUT`.

Daten-Eingänge
- `IN` (DT): Der Eingang für den zu konvertierenden Datums-Zeit-Wert.

Daten-Ausgänge
- `OUT` (DATE): Der Ausgang für das konvertierte Datum.

Adapter
Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise
Bei Empfang eines `REQ`-Ereignisses wird der DT-Wert am Eingang `IN` in ein DATE konvertiert und am Ausgang `OUT` ausgegeben. Gleichzeitig wird das `CNF`-Ereignis ausgelöst, um den Abschluss der Operation zu signalisieren.

Technische Besonderheiten
- Der Funktionsblock implementiert die Konvertierung mittels der Funktion `DT_TO_DATE`.
- Die Konvertierung ist atomar und erfolgt in einem einzigen Ausführungsschritt.

Zustandsübersicht
Der Funktionsblock besitzt keine internen Zustände. Jede `REQ`-Anfrage wird unabhängig verarbeitet.

Anwendungsszenarien
- Extraktion des Datums aus Zeitstempeln in Logging-Systemen
- Vorverarbeitung von Datumsinformationen für Berichte oder Visualisierungen
- Schnittstellen zu Systemen, die nur DATE-Datentypen verarbeiten können

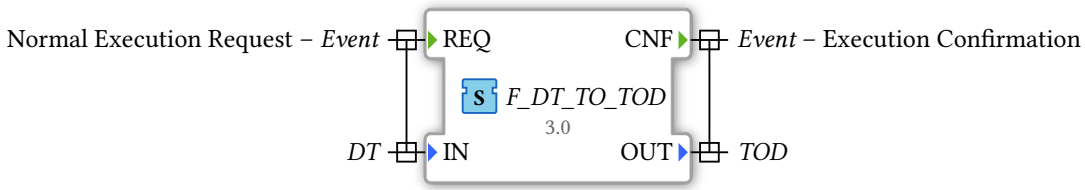
⚖️ Vergleich mit ähnlichen Bausteinen
- Im Vergleich zu `F_DT_TO_TIME` konvertiert dieser Baustein nur den Datumsteil
- Ähnlich zu `F_DT_TO_TOD`, aber mit Fokus auf das Datum statt der Tageszeit

Fazit
Der `F_DT_TO_DATE` Funktionsblock bietet eine einfache und effiziente Möglichkeit, DT-Werte in reine Datumsangaben umzuwandeln. Seine schlanke Implementierung macht ihn besonders geeignet für Echtzeitanwendungen, wo Performance kritisch ist.

F_DT_TO_TOD

• – ◦ • – ◦ • – ◦ •

Einleitung
Der Funktionsblock `F_DT_TO_TOD` dient der Konvertierung eines Datums- und Zeitwerts (DT) in einen reinen Zeitwert (TOD - Time of Day). Diese Funktion ist besonders nützlich, wenn nur die Tageszeit aus einem kombinierten Datums- und Zeitstempel extrahiert werden soll.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Eingang ausgelöst, führt der Funktionsblock die Umwandlung durch.

Ereignis-Ausgänge

- **CNF**: Signalisiert die erfolgreiche Beendigung der Konvertierung.

Daten-Eingänge

- **IN**: Der Eingang vom Typ `DT`, der den zu konvertierenden Datums- und Zeitwert enthält.

Daten-Ausgänge

- **OUT**: Der Ausgang vom Typ `TOD`, der den konvertierten Zeitwert (ohne Datum) bereitstellt.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Auslösung des `REQ`-Ereignisses wird der Algorithmus ausgeführt, der den Eingabewert `IN` vom Typ `DT` in einen `TOD`-Wert umwandelt. Das Ergebnis wird am Ausgang `OUT` ausgegeben und das `CNF`-Ereignis signalisiert die erfolgreiche Beendigung der Operation.

Technische Besonderheiten

- Der Funktionsblock ist als einfacher FB (SimpleFB) implementiert, was eine schnelle und effiziente Ausführung gewährleistet.
- Die Konvertierung erfolgt über die eingebaute Funktion `DT_TO_TOD`.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung wird bei jedem `REQ`-Ereignis neu durchgeführt.

Anwendungsszenarien

- Extraktion der Tageszeit aus Zeitstempeln für Zeitsteuerungen.
- Vorverarbeitung von Daten für Systeme, die nur die Tageszeit benötigen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu `F_TOD_TO_DT` konvertiert dieser Baustein in die entgegengesetzte Richtung.
- Ähnliche Bausteine wie `F_DT_TO_DATE` oder `F_DT_TO_TIME` extrahieren andere Teile eines DT-Werts.

Fazit

Der `F_DT_TO_TOD`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, die Tageszeit aus einem kombinierten Datums- und Zeitwert zu extrahieren. Seine klare Schnittstelle und einfache Funktionsweise machen ihn zu einem nützlichen Werkzeug in Steuerungsanwendungen.

DWORD

DWORD Konvertierungen

Hier finden Sie Funktionen zur Konvertierung des Datentyps DWORD.

🛠️ Zugehörige Übungen

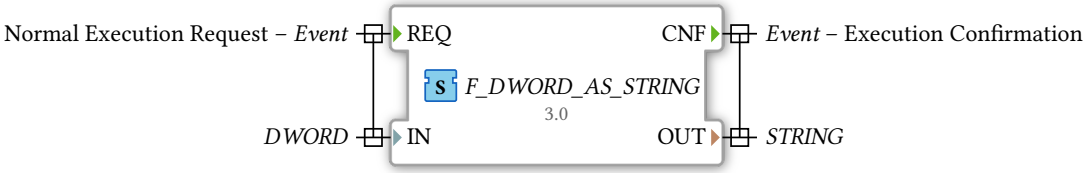
- Uebung_034b
- SIN_TO_PCAN_Callback_DWord
- Uebung_152
- Uebung_153

F_DWORD_AS_STRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DWORD_AS_STRING` dient der Konvertierung eines DWORD-Wertes in einen STRING. Dieser Baustein ist besonders nützlich, wenn numerische Werte in eine für Menschen lesbare Form gebracht oder für die Ausgabe in Textform vorbereitet werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung des DWORD-Wertes in einen STRING. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN**: Der DWORD-Wert, der in einen STRING konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der resultierende STRING nach der Konvertierung des DWORD-Wertes.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` ausgelöst wird. Dabei wird der Wert am Eingang `IN` (DWORD) in einen STRING umgewandelt und am Ausgang `OUT` ausgegeben. Die erfolgreiche Konvertierung wird durch das Ereignis `CNF` signalisiert.

Technische Besonderheiten

- Der Baustein verwendet die Funktion `DWORD_AS_STRING` zur Konvertierung.
- Die Konvertierung erfolgt synchron mit dem Auslösen des `REQ`-Ereignisses.

Zustandsübersicht

1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Processing**: Führt die Konvertierung durch.
3. **Complete**: Signalisiert den Abschluss mit `CNF` und gibt das Ergebnis an `OUT` aus.

Anwendungsszenarien

- Umwandlung von numerischen IDs oder Statuswerten in lesbare Textform.
- Vorbereitung von Daten für die Protokollierung oder Benutzeranzeige.
- Integration in Systeme, die String-basierte Kommunikation erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsbausteinen ist `F_DWORD_AS_STRING` spezialisiert auf die Umwandlung von DWORD zu STRING und bietet daher eine einfache und direkte Lösung für diesen spezifischen Anwendungsfall.
- Andere Bausteine könnten zusätzliche Konvertierungsoptionen oder Parameter bieten, sind jedoch möglicherweise komplexer in der Handhabung.

Fazit

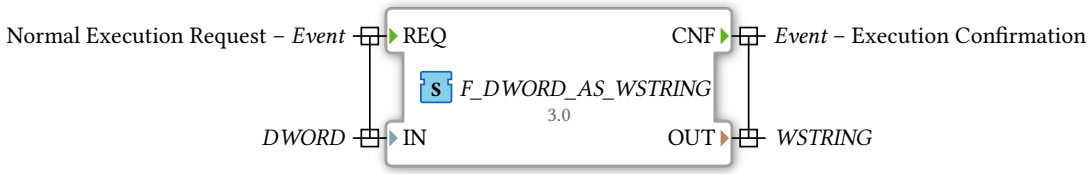
Der `F_DWORD_AS_STRING` Funktionsblock ist ein effizientes und einfach zu handhabendes Werkzeug für die Konvertierung von DWORD-Werten in Strings. Seine klare Schnittstellenstruktur und direkte Funktionsweise machen ihn ideal für Anwendungen, bei denen eine schnelle und zuverlässige Umwandlung benötigt wird.

F_DWORD_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DWORD_AS_WSTRING` dient der Konvertierung eines DWORD-Wertes in einen WSTRING-Wert. Dieser Baustein ist besonders nützlich in Szenarien, wo numerische Werte in eine Zeichenkette umgewandelt werden müssen, beispielsweise für die Anzeige oder Protokollierung.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Dieser Eingang löst die Konvertierung aus.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Dieser Ausgang signalisiert den Abschluss der Konvertierung.

Daten-Eingänge

- **IN**: Eingang für den DWORD-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Ausgang für den resultierenden WSTRING-Wert nach der Konvertierung.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von DWORD zu WSTRING durch, sobald das Ereignis `REQ` empfangen wird. Die Konvertierung erfolgt mittels der Funktion `DWORD_AS_WSTRING`. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst, und der konvertierte Wert steht am Ausgang `OUT` zur Verfügung.

Technische Besonderheiten

- Der Baustein ist einfach und effizient implementiert, ohne zusätzliche Zustandsverwaltung.
- Die Konvertierung erfolgt direkt und ohne Verzögerung.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung wird bei jedem `REQ`-Ereignis neu ausgeführt.

Anwendungsszenarien

- Anzeige von numerischen Werten in einer Benutzeroberfläche.
- Protokollierung von numerischen Daten in Textform.
- Kommunikation mit Systemen, die Zeichenketten erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsbausteinen ist `F_DWORD_AS_WSTRING` spezialisiert auf die Konvertierung von DWORD zu WSTRING und daher effizienter in diesem spezifischen Anwendungsfall.
- Andere Bausteine könnten zusätzliche Parameter oder Zustände haben, was hier nicht notwendig ist.

Fazit

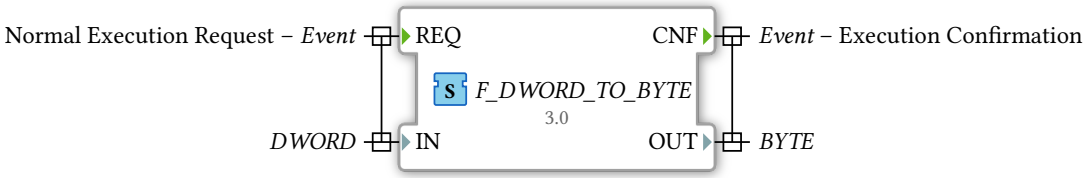
Der `F_DWORD_AS_WSTRING` Funktionsblock ist ein einfacher und effizienter Baustein für die Konvertierung von DWORD-Werten in WSTRING. Seine Einfachheit macht ihn besonders geeignet für Anwendungen, wo eine schnelle und direkte Konvertierung benötigt wird.

F_DWORD_TO_BYTE

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DWORD_TO_BYTE` dient der Konvertierung eines 32-Bit-Wertes (DWORD) in einen 8-Bit-Wert (BYTE). Diese Konvertierung ist nützlich in Szenarien, wo eine Datenreduktion oder Typanpassung erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung des Funktionsblocks. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert die erfolgreiche Beendigung der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `DWORD` (32-Bit), der in ein `BYTE` konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgangswert vom Typ `BYTE` (8-Bit), der das Ergebnis der Konvertierung enthält.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `DWORD` zu `BYTE` durch, sobald das Ereignis `REQ` empfangen wird. Die Konvertierung erfolgt mittels der Funktion `DWORD_TO_BYTE`. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Die Konvertierung kann zu einem Datenverlust führen, wenn der ursprüngliche `DWORD`-Wert außerhalb des darstellbaren Bereichs eines `BYTE` (0-255) liegt.
- Der Funktionsblock ist als einfacher FB (SimpleFB) implementiert und verwendet einen ST-Algorithmus für die Konvertierung.

Zustandsübersicht

1. **Idle**: Wartet auf das Ereignis `REQ`.
2. **Processing**: Führt die Konvertierung durch.

3. **Completed**: Sendet das Ereignis `CNF` und kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- Datenreduktion in Steuerungssystemen.
- Typkonvertierung in Kommunikationsprotokollen.
- Anpassung von Daten für spezifische Hardware-Schnittstellen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_WORD_TO_BYTE**: Konvertiert einen 16-Bit-Wert (WORD) in ein BYTE.
- **F_DWORD_TO_WORD**: Konvertiert einen 32-Bit-Wert (DWORD) in ein 16-Bit-Wert (WORD).
- **F_BYTE_TO_DWORD**: Führt die umgekehrte Konvertierung durch (BYTE zu DWORD).

Fazit

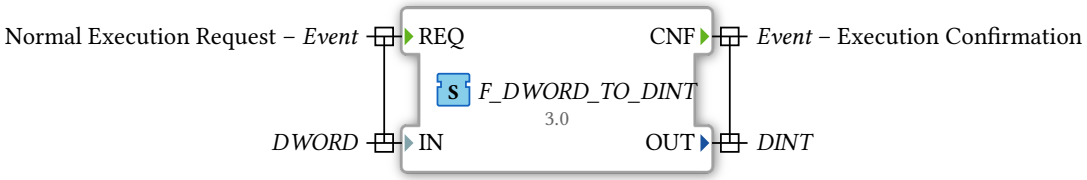
Der Funktionsblock `F_DWORD_TO_BYTE` bietet eine einfache und effiziente Möglichkeit, 32-Bit-Werte in 8-Bit-Werte zu konvertieren. Aufgrund des möglichen Datenverlusts bei großen Werten sollte die Verwendung jedoch sorgfältig abgewogen werden.

F_DWORD_TO_DINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DWORD_TO_DINT` dient der Konvertierung eines `DWORD`-Wertes in einen `DINT`-Wert. Diese Konvertierung ist insbesondere dann notwendig, wenn Daten zwischen Systemen ausgetauscht werden, die unterschiedliche Datentypen verwenden. Der Block ist Teil des Pakets `iec61131::conversion` und folgt dem Standard 61499-1.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN**: Der Eingang erwartet einen `DWORD`-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgang liefert den konvertierten `DINT`-Wert.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `DWORD` zu `DINT` durch, sobald das Ereignis `REQ` empfangen wird. Die Konvertierung erfolgt mittels der Funktion `DWORD_TO_DINT(IN)`. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst und der konvertierte Wert über den Ausgang `OUT` ausgegeben.

Technische Besonderheiten

- Der Block ist als einfacher Funktionsblock (`SimpleFB`) implementiert.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Zustandsverwaltung.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine expliziten Zustände. Die Konvertierung wird bei jedem `REQ`-Ereignis durchgeführt.

Anwendungsszenarien

- Konvertierung von `DWORD`-Daten, die von einem System geliefert werden, in `DINT`-Daten für ein anderes System.
- Verwendung in Steuerungssystemen, wo Datentypen harmonisiert werden müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Blöcke könnten `F_WORD_TO_INT` oder `F_LWORD_TO_LINT` sein, die jedoch andere Datentypen konvertieren.
- Im Gegensatz zu komplexeren Konvertierungsblöcken bietet `F_DWORD_TO_DINT` eine direkte und einfache Lösung für die spezifische Konvertierung.

Fazit

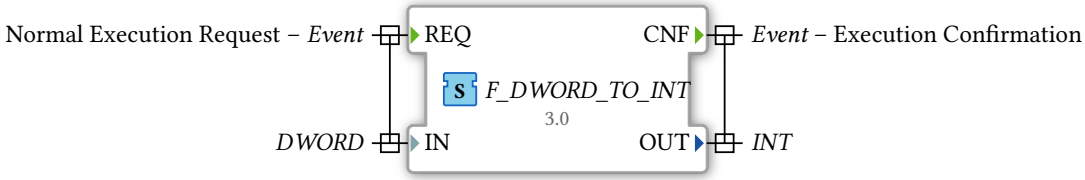
Der Funktionsblock `F_DWORD_TO_DINT` bietet eine effiziente und einfache Möglichkeit, `DWORD`-Werte in `DINT`-Werte zu konvertieren. Durch seine klare Schnittstelle und direkte Funktionsweise ist er ideal für Anwendungen, die eine schnelle und zuverlässige Typkonvertierung erfordern.

F_DWORD_TO_INT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DWORD_TO_INT` dient der Konvertierung eines DWORD-Wertes in einen INT-Wert. Er ist Teil des Pakets `iec61131::conversion` und ermöglicht die einfache Umwandlung zwischen diesen beiden Datentypen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung der Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingang vom Typ `DWORD`, der den zu konvertierenden Wert enthält.

Daten-Ausgänge

- **OUT**: Der Ausgang vom Typ `INT`, der das Ergebnis der Konvertierung liefert.

Adapter

Der Funktionsblock verfügt über keine Adapter.

Funktionsweise

Bei Empfang eines Ereignisses am Eingang `REQ` wird der Algorithmus ausgeführt, der den Wert von `IN` vom Typ `DWORD` in einen Wert vom Typ `INT` konvertiert. Das Ergebnis wird am Ausgang `OUT` ausgegeben und durch das Ereignis `CNF` bestätigt.

Technische Besonderheiten

- Der Funktionsblock verwendet die Standardfunktion `DWORD_TO_INT` für die Konvertierung.
- Es handelt sich um einen einfachen Funktionsblock ohne interne Zustände oder komplexe Logik.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Konvertierung von DWORD-Werten, die von Hardware oder anderen Systemen geliefert werden, in INT-Werte für die weitere Verarbeitung.
- Integration in Steuerungssysteme, die unterschiedliche Datentypen verarbeiten müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen wie `F_WORD_TO_INT` oder `F_BYTE_TO_INT` ist dieser Funktionsblock speziell für die Konvertierung von DWORD zu INT optimiert.
- Er bietet eine einfache und direkte Lösung ohne zusätzliche Funktionen oder Parameter.

Fazit

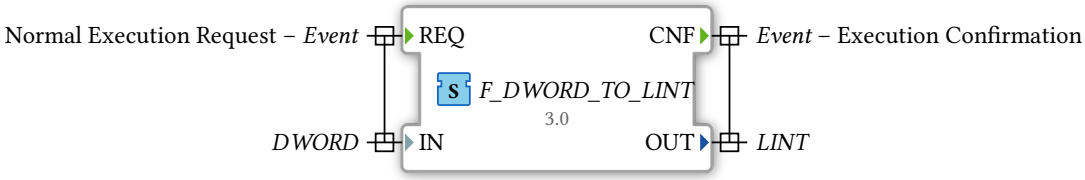
Der `F_DWORD_TO_INT` Funktionsblock ist ein nützliches Werkzeug für die Konvertierung zwischen DWORD- und INT-Datentypen. Seine Einfachheit und direkte Funktionsweise machen ihn zu einer zuverlässigen Lösung in verschiedenen Anwendungsfällen.

F_DWORD_TO_LINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DWORD_TO_LINT` dient der Konvertierung eines `DWORD`-Wertes in einen `LINT`-Wert. Diese Konvertierung ist besonders nützlich, wenn Daten zwischen Systemen oder Komponenten ausgetauscht werden müssen, die unterschiedliche Datentypen verwenden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Dieser Ereigniseingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ereignisausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `DWORD`, der in einen `LINT`-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgangswert vom Typ `LINT`, der das Ergebnis der Konvertierung enthält.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Empfang eines Ereignisses am Eingang `REQ` wird der Algorithmus ausgeführt, der den `DWORD`-Wert am Eingang `IN` in einen `LINT`-Wert konvertiert und das Ergebnis am Ausgang `OUT` ausgibt. Anschließend wird das Ereignis `CNF` ausgelöst, um den Abschluss der Konvertierung zu signalisieren.

Technische Besonderheiten

- Der Funktionsblock ist einfach aufgebaut und enthält nur einen Algorithmus, der die Konvertierung durchführt.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Verarbeitungsschritte.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung wird bei jedem `REQ`-Ereignis neu ausgeführt.

Anwendungsszenarien

- Konvertierung von Daten zwischen Systemen, die unterschiedliche Datentypen verwenden.
- Verwendung in Steuerungssystemen, wo eine Typumwandlung erforderlich ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungs-Funktionsblöcken ist `F_DWORD_TO_LINT` spezialisiert auf die Konvertierung von `DWORD` zu `LINT`.
- Andere Funktionsblöcke könnten ähnliche Konvertierungen für andere Datentypen durchführen, wie z.B. `F_WORD_TO_INT`.

Fazit

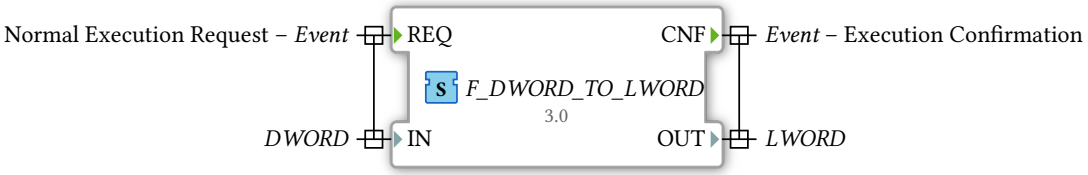
Der Funktionsblock `F_DWORD_TO_LINT` ist ein einfaches und effizientes Werkzeug zur Konvertierung von `DWORD`-Werten in `LINT`-Werte. Seine klare Schnittstellenstruktur und direkte Funktionsweise machen ihn zu einer zuverlässigen Lösung für Typumwandlungen in Steuerungssystemen.

F_DWORD_TO_LWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DWORD_TO_LWORD` dient der Konvertierung eines 32-Bit-Datenwerts (DWORD) in einen 64-Bit-Datenwert (LWORD). Diese Konvertierung ist besonders in Szenarien nützlich, in denen eine Erweiterung der Datenbreite erforderlich ist, um Kompatibilität mit nachfolgenden Verarbeitungsschritten zu gewährleisten.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Typ: Event): Startet die Ausführung des Funktionsblocks. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF** (Typ: Event): Signalisiert den erfolgreichen Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (Typ: DWORD): Der 32-Bit-Eingangswert, der in einen 64-Bit-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT** (Typ: LWORD): Der resultierende 64-Bit-Ausgangswert nach der Konvertierung.

Adapter

Der Funktionsblock verfügt über keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung von DWORD zu LWORD durch, indem der 32-Bit-Eingangswert (`IN`) in einen 64-Bit-Wert (`OUT`) umgewandelt wird. Die Konvertierung erfolgt automatisch bei Empfang eines Ereignisses am Eingang `REQ`. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Der Funktionsblock ist als einfacher FB (SimpleFB) implementiert und verwendet einen ST-Algorithmus für die Konvertierung.
- Die Konvertierung erfolgt ohne Datenverlust, da der 32-Bit-Wert in den 64-Bit-Wert eingebettet wird.

Zustandsübersicht

Der Funktionsblock hat keinen internen Zustand. Die Konvertierung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Erweiterung der Datenbreite für nachfolgende Verarbeitungsschritte.
- Kompatibilitätsanpassung zwischen Systemen, die unterschiedliche Datenbreiten verwenden.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen (z.B. `F_WORD_TO_DWORD`) ist dieser Funktionsblock speziell für die Konvertierung von 32-Bit- zu 64-Bit-Werten optimiert.
- Ähnliche Bausteine könnten zusätzliche Funktionen wie Skalierung oder Offset-Anpassung bieten, was hier nicht der Fall ist.

Fazit

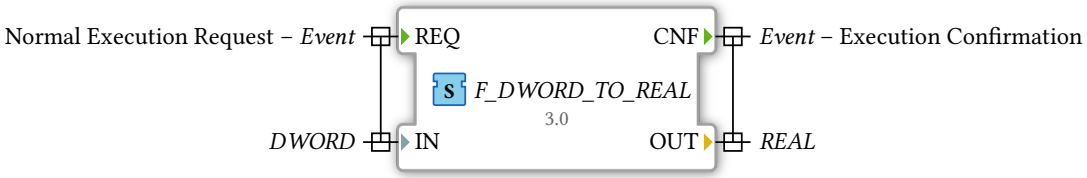
Der `F_DWORD_TO_LWORD` Funktionsblock ist ein einfaches und effizientes Werkzeug zur Konvertierung von 32-Bit- zu 64-Bit-Werten. Seine klare Schnittstellenstruktur und direkte Funktionsweise machen ihn ideal für Anwendungen, bei denen eine reine Datenbreitenerweiterung benötigt wird.

F_DWORD_TO_REAL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DWORD_TO_REAL` dient der Konvertierung eines DWORD-Wertes (32-Bit-Ganzzahl ohne Vorzeichen) in einen REAL-Wert (32-Bit-Gleitkommazahl). Dieser Baustein ist besonders nützlich in Szenarien, wo eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Eingang ausgelöst, wird der Wert am Daten-Eingang `IN` verarbeitet.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ausgang wird aktiviert, sobald die Umwandlung abgeschlossen ist und der Ergebniswert am Daten-Ausgang `OUT` verfügbar ist.

Daten-Eingänge

- **IN**: DWORD-Wert, der in einen REAL-Wert umgewandelt werden soll.

Daten-Ausgänge

- **OUT**: Der konvertierte REAL-Wert.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` ausgelöst wird. Dabei wird der Wert am Eingang `IN` mittels der Funktion `DWORD_TO_REAL` in einen REAL-Wert umgewandelt und am Ausgang `OUT` ausgegeben. Abschließend wird das Ereignis `CNF` aktiviert, um den erfolgreichen Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne zusätzliche Verzögerung.
- Es werden keine speziellen Fehlerbehandlungen durchgeführt, z.B. bei ungültigen Eingabewerten.

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf das `REQ`-Ereignis.
2. **Konvertierungs-Zustand**: Führt die Umwandlung durch und setzt den Ausgang `OUT`.
3. **Bestätigungs-Zustand**: Aktiviert `CNF` und kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo DWORD-Daten in Gleitkommazahlen umgewandelt werden müssen.
- Integration in größere Funktionsblöcke, die eine Typkonvertierung erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_WORD_TO_REAL` oder `F_INT_TO_REAL` bieten ähnliche Funktionalitäten, jedoch für andere Eingabetypen.
- `F_DWORD_TO_REAL` ist spezifisch für die Umwandlung von DWORD zu REAL optimiert.

Fazit

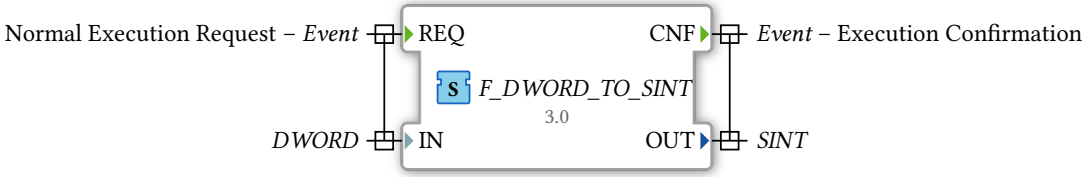
Der `F_DWORD_TO_REAL` Funktionsblock ist ein einfacher und effizienter Baustein zur Konvertierung von DWORD- in REAL-Werte. Seine klare Schnittstelle und direkte Funktionsweise machen ihn zu einer zuverlässigen Komponente in Steuerungsanwendungen.

F_DWORD_TO_SINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DWORD_TO_SINT` dient der Konvertierung eines 32-Bit-DWORD-Wertes in einen 8-Bit-SINT-Wert (vorzeichenbehaftete Ganzzahl). Dieser Baustein ist Teil der IEC 61131-Konvertierungsbibliothek und ermöglicht eine einfache Typumwandlung zwischen diesen Datentypen.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Startet die Ausführung der Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- `CNF`: Signalisiert den erfolgreichen Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- `IN` (DWORD): Der 32-Bit-Eingangswert, der in einen SINT-Wert konvertiert werden soll.

Daten-Ausgänge

- `OUT` (SINT): Der konvertierte 8-Bit-SINT-Wert.

Adapter

Der Funktionsblock verfügt über keine Adapter-Schnittstellen.

Funktionsweise

Bei Auslösung des `REQ`-Ereignisses wird der Algorithmus ausgeführt, der den `IN`-Wert vom Typ DWORD in einen SINT-Wert umwandelt. Die Konvertierung erfolgt durch die eingebaute Funktion `DWORD_TO_SINT`. Nach Abschluss der Konvertierung wird das `CNF`-Ereignis ausgelöst, und der konvertierte Wert steht am `OUT`-Ausgang zur Verfügung.

Technische Besonderheiten

- Die Konvertierung kann zu einem Datenverlust führen, wenn der DWORD-Wert außerhalb des darstellbaren Bereichs eines SINT (-128 bis 127) liegt.
- Der Funktionsblock ist ein einfacher Baustein ohne interne Zustandsverwaltung.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Ausführung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo DWORD-Werte in kompaktere SINT-Werte umgewandelt werden müssen.
- Datenreduktion, wenn nur der niederwertigste Teil eines DWORD-Wertes benötigt wird.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu `F_DWORD_TO_INT` oder `F_DWORD_TO_USINT` konvertiert dieser Baustein in einen vorzeichenbehafteten 8-Bit-Wert.
- Ähnliche Bausteine für andere Zieltypen sind in der IEC 61131-Konvertierungsbibliothek verfügbar.

Fazit

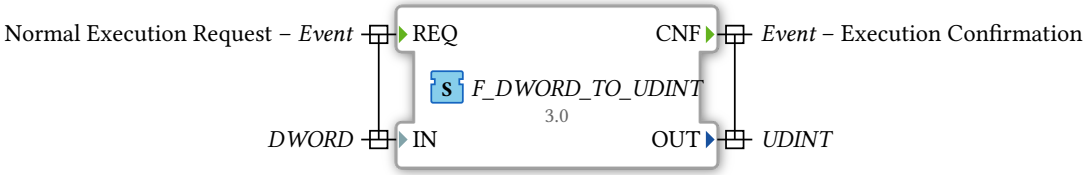
Der `F_DWORD_TO_SINT`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, DWORD-Werte in SINT-Werte umzuwandeln. Er ist besonders nützlich in Anwendungen, wo Speicherplatz oder Datenbandbreite optimiert werden müssen. Bei der Verwendung ist auf den möglichen Wertebereich des SINT-Datentyps zu achten, um unerwünschte Datenverluste zu vermeiden.

F_DWORD_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DWORD_TO_UDINT` dient der Konvertierung eines `DWORD`-Werts in einen `UDINT`-Wert. Dieser Baustein ist Teil des `iec61131::conversion`-Pakets und ermöglicht eine einfache und effiziente Typumwandlung.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Dieser Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingang vom Typ `DWORD`, der den zu konvertierenden Wert enthält.

Daten-Ausgänge

- **OUT**: Der Ausgang vom Typ `UDINT`, der das Ergebnis der Konvertierung liefert.

Adapter

- Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `DWORD` zu `UDINT` durch, sobald das `REQ`-Ereignis ausgelöst wird. Die Umwandlung erfolgt mithilfe der integrierten Funktion `DWORD_TO_UDINT`. Nach Abschluss der Konvertierung wird das `CNF`-Ereignis ausgelöst, und der konvertierte Wert steht am Ausgang `OUT` zur Verfügung.

Technische Besonderheiten

- Der Funktionsblock ist einfach aufgebaut und enthält nur einen Algorithmus (`REQ`), der die Konvertierung durchführt.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Verzögerung.

Zustandsübersicht

Der Funktionsblock hat keinen internen Zustand. Die Konvertierung wird bei jedem `REQ`-Ereignis neu ausgeführt.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, bei denen `DWORD`-Werte in `UDINT`-Werte umgewandelt werden müssen.
- Integration in größere Steuerungssysteme, die eine Typkompatibilität zwischen verschiedenen Datenformaten erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_WORD_TO_UINT` oder `F_BYTE_TO_USINT` führen ähnliche Typumwandlungen durch, jedoch für andere Datentypen.
- Im Gegensatz zu generischen Konvertierungsblöcken ist `F_DWORD_TO_UDINT` spezialisiert und optimiert für die Umwandlung von `DWORD` zu `UDINT`.

🛠️ Zugehörige Übungen

- Uebung_011
- Uebung_011a2
- Uebung_012
- NumericValue_ID_TO_NVS
- Uebung_012b

- Uebung_012c
- Uebung_020c2_sub
- Uebung_028
- Uebung_034
- Uebung_034a1_Q1
- Uebung_034a1_Q2
- Uebung_034a1_Q4
- Uebung_103
- Uebung_151
- Uebung_151_AX
- Uebung_152
- Uebung_153

Fazit

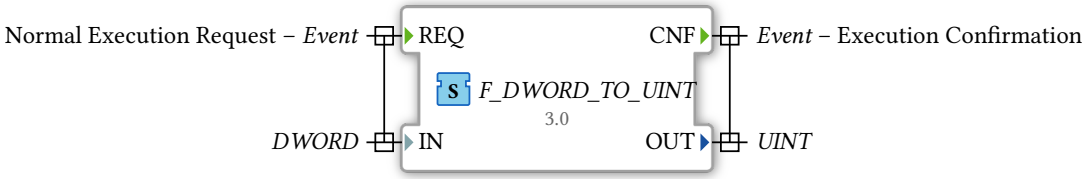
Der `F_DWORD_TO_UDINT`-Funktionsblock bietet eine einfache und effiziente Lösung für die Konvertierung von `DWORD`-Werten in `UDINT`-Werte. Seine klare Schnittstellenstruktur und direkte Funktionsweise machen ihn zu einem zuverlässigen Baustein in Steuerungsanwendungen.

F_DWORD_TO_UINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DWORD_TO_UINT` dient der Konvertierung eines `DWORD`-Wertes in einen `UINT`-Wert. Dieser Baustein ist besonders nützlich in Szenarien, wo eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist, beispielsweise bei der Kommunikation zwischen verschiedenen Systemen oder bei der Verarbeitung von Daten aus unterschiedlichen Quellen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Dieser Eingang löst die Ausführung des Funktionsblocks aus. Der zugehörige Datenwert `IN` wird bei diesem Ereignis verarbeitet.

Ereignis-Ausgänge

- **CNF**: Dieser Ausgang wird nach erfolgreicher Konvertierung aktiviert und bestätigt die Ausführung. Der konvertierte Wert `OUT` wird mit diesem Ereignis ausgegeben.

Daten-Eingänge

- **IN** (`DWORD`): Der Eingangswert, der von `DWORD` in `UINT` konvertiert werden soll.

Daten-Ausgänge

- **OUT** (`UINT`): Der Ausgangswert nach der Konvertierung.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `DWORD` zu `UINT` durch, sobald das Ereignis `REQ` empfangen wird. Die Konvertierung erfolgt mittels der Funktion `DWORD_TO_UINT`, die den Eingabewert `IN` umwandelt und das Ergebnis an `OUT` ausgibt. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher Algorithmus (`SimpleFB`) und führt keine komplexen Zustandsübergänge durch.
- Die Konvertierung erfolgt direkt und ohne Verzögerung.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine komplexen Zustandsübergänge. Der Block reagiert ausschließlich auf das `REQ`-Ereignis mit der Ausgabe des konvertierten Wertes und dem `CNF`-Ereignis.

Anwendungsszenarien

- Typumwandlung in Kommunikationsschnittstellen, wo unterschiedliche Systeme verschiedene Datentypen verwenden.
- Datenverarbeitung in Steuerungssystemen, wo eine Konvertierung zwischen `DWORD` und `UINT` erforderlich ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen wie `F_WORD_TO_UINT` oder `F_BYTE_TO_UINT` ist dieser speziell für die Umwandlung von `DWORD` zu `UINT` optimiert.
- Ähnliche Bausteine könnten zusätzliche Funktionen wie Bereichsprüfungen oder Fehlerbehandlungen bieten, was dieser Funktionsblock nicht tut.

Fazit

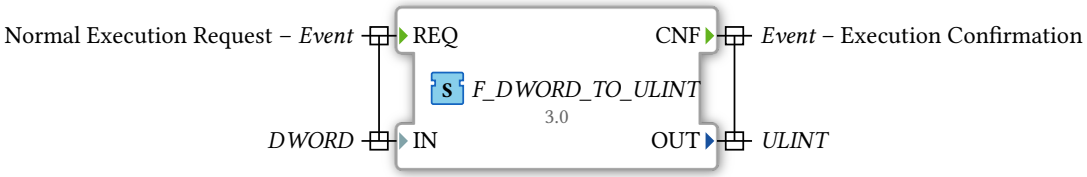
Der `F_DWORD_TO_UINT` Funktionsblock ist ein einfaches und effizientes Werkzeug zur Konvertierung von `DWORD`-Werten in `UINT`-Werte. Seine Einfachheit macht ihn besonders geeignet für Anwendungen, wo eine schnelle und direkte Typumwandlung benötigt wird.

F_DWORD_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DWORD_TO_ULINT` dient der Konvertierung eines `DWORD`-Wertes in einen `ULINT`-Wert. Er ist Teil des `iec61131::conversion`-Pakets und wird typischerweise in Automatisierungsanwendungen eingesetzt, wo eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Dieser Eingang löst die Ausführung des Funktionsblocks aus. Er ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Dieser Ausgang signalisiert die erfolgreiche Ausführung des Funktionsblocks. Er ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingang erwartet einen `DWORD`-Wert, der in einen `ULINT`-Wert umgewandelt werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgang liefert den konvertierten `ULINT`-Wert.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `DWORD` zu `ULINT` durch, sobald das `REQ`-Ereignis eintrifft. Der Algorithmus innerhalb des Funktionsblocks nutzt die Funktion `DWORD_TO_ULINT`, um die Umwandlung durchzuführen. Nach erfolgreicher Konvertierung wird das `CNF`-Ereignis ausgelöst, und der konvertierte Wert steht am `OUT`-Ausgang zur Verfügung.

Technische Besonderheiten

- Der Funktionsblock ist als einfacher Funktionsblock (`SimpleFB`) implementiert.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Verzögerung.
- Der Funktionsblock ist unter der Eclipse Public License 2.0 lizenziert.

Zustandsübersicht

Der Funktionsblock hat keinen internen Zustand. Die Ausführung erfolgt ausschließlich bei Eintreffen des `REQ`-Ereignisses.

Anwendungsszenarien

- Typumwandlung in Automatisierungsanwendungen, wo `DWORD`-Werte in `ULINT`-Werte umgewandelt werden müssen.
- Integration in größere Steuerungssysteme, die unterschiedliche Datentypen verarbeiten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_WORD_TO_UINT` oder `F_BYTE_TO_USINT` führen ähnliche Konvertierungen durch, jedoch für andere Datentypen.
- `F_DWORD_TO_ULINT` ist spezialisiert auf die Umwandlung von `DWORD` zu `ULINT` und bietet keine zusätzliche Funktionalität.

Fazit

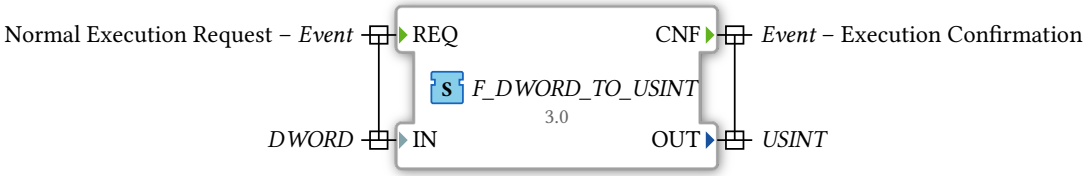
Der `F_DWORD_TO_ULINT`-Funktionsblock ist ein einfaches und effizientes Werkzeug zur Typumwandlung in IEC 61131-3-basierten Steuerungssystemen. Seine klare Schnittstelle und direkte Funktionsweise machen ihn zu einer zuverlässigen Komponente in Automatisierungsprojekten.

F_DWORD_TO_USINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DWORD_TO_USINT` dient der Konvertierung eines 32-Bit-Wertes (DWORD) in einen 8-Bit-unsigned-Integer-Wert (USINT). Diese Konvertierung ist besonders nützlich, wenn Daten zwischen Systemen mit unterschiedlichen Wortbreiten ausgetauscht werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Eingang ausgelöst, wird der Wert am Daten-Eingang `IN` verarbeitet.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der konvertierte Wert steht am Daten-Ausgang `OUT` zur Verfügung.

Daten-Eingänge

- **IN** (DWORD): Der 32-Bit-Wert, der in einen 8-Bit-unsigned-Integer-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT** (USINT): Der resultierende 8-Bit-unsigned-Integer-Wert nach der Konvertierung.

Adapter

Dieser Funktionsblock verfügt über keine Adapter-Schnittstellen.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem der 32-Bit-Wert am Eingang `IN` in einen 8-Bit-unsigned-Integer-Wert umgewandelt wird. Die Konvertierung erfolgt durch die Funktion `DWORD_TO_USINT`, die den Wert entsprechend umwandelt. Bei der Konvertierung wird nur das niederwertigste Byte des DWORD-Wertes berücksichtigt, während die höherwertigen Bytes ignoriert werden.

Technische Besonderheiten

- Die Konvertierung schneidet die höherwertigen Bytes ab, was zu einem Datenverlust führen kann, wenn der ursprüngliche DWORD-Wert größer als 255 ist.
- Der Funktionsblock ist deterministisch und führt die Konvertierung in einem einzigen Ausführungsschritt durch.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung wird bei jedem Auslösen des `REQ`-Ereignisses neu durchgeführt.

Anwendungsszenarien

- Datenreduktion in Kommunikationsprotokollen, wo nur 8-Bit-Werte übertragen werden können.
- Anpassung von Datenformaten zwischen unterschiedlichen Systemkomponenten.
- Verwendung in Steuerungen, wo spezifische Hardware nur 8-Bit-Daten verarbeiten kann.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_WORD_TO_USINT**: Konvertiert einen 16-Bit-Wert (WORD) in einen 8-Bit-unsigned-Integer-Wert. Im Gegensatz dazu verarbeitet `F_DWORD_TO_USINT` 32-Bit-Werte.
- **F_DWORD_TO_BYTE**: Ähnlich in der Funktionalität, aber der Ausgangstyp ist BYTE, was ebenfalls 8 Bit umfasst, aber ohne Vorzeichen.

Fazit

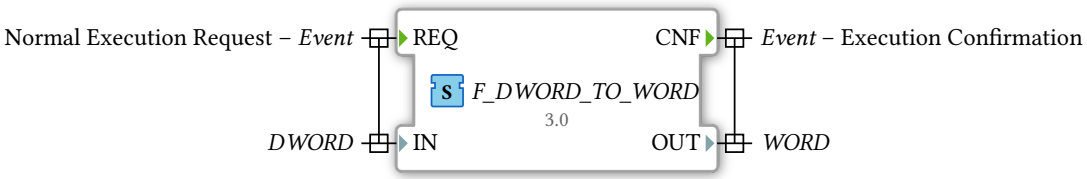
Der Funktionsblock `F_DWORD_TO_USINT` ist ein einfaches und effizientes Werkzeug zur Konvertierung von 32-Bit-Werten in 8-Bit-unsigned-Integer-Werte. Er eignet sich besonders für Anwendungen, bei denen Daten auf eine kleinere Wortbreite reduziert werden müssen. Aufgrund der möglichen Datenverluste bei großen Eingangswerten sollte seine Verwendung jedoch sorgfältig geplant werden.

F_DWORD_TO_WORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_DWORD_TO_WORD` dient der Konvertierung eines 32-Bit-DWORD-Wertes in einen 16-Bit-WORD-Wert. Diese Konvertierung ist nützlich, wenn Daten zwischen Systemen mit unterschiedlichen Wortbreiten ausgetauscht werden müssen oder wenn Speicherplatz optimiert werden soll.



Schnittstellenstruktur

Ereignis-Eingänge
- **REQ**: Startet die Konvertierung. Dieser Ereigniseingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge
- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ereignisausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge
- **IN** (DWORD): Der 32-Bit-Eingangswert, der konvertiert werden soll.

Daten-Ausgänge
- **OUT** (WORD): Der konvertierte 16-Bit-Ausgangswert.

Adapter
Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise
Bei Auslösung des `REQ`-Ereignisses wird der Algorithmus ausgeführt, der den DWORD-Wert am Eingang `IN` in einen WORD-Wert konvertiert und das Ergebnis an `OUT` ausgibt. Anschließend wird das `CNF`-Ereignis ausgelöst, um den Abschluss der Konvertierung zu signalisieren.

Technische Besonderheiten
- Die Konvertierung erfolgt durch Abschneiden der höherwertigen 16 Bit des DWORD-Wertes.
- Es findet keine Überprüfung auf möglichen Datenverlust durch diese Konvertierung statt.

Zustandsübersicht
Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien
- Kommunikation zwischen Systemen mit unterschiedlichen Wortbreiten
- Speicheroptimierung in eingebetteten Systemen
- Datenvorverarbeitung in Steuerungssystemen

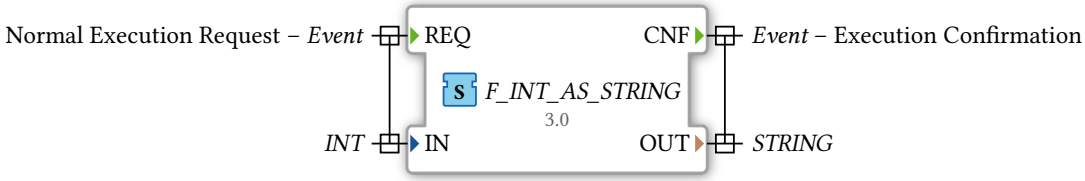
⚖️ Vergleich mit ähnlichen Bausteinen
- Im Gegensatz zu `F_DWORD_TO_BYTE` konvertiert dieser Baustein in einen WORD-Wert statt in ein BYTE.
- Ähnlich wie `F_WORD_TO_BYTE`, aber mit anderen Datentypen.

Fazit
Der `F_DWORD_TO_WORD` Funktionsblock bietet eine einfache und effiziente Möglichkeit, DWORD-Werte in WORD-Werte zu konvertieren. Er ist besonders nützlich in Systemen, die mit verschiedenen Wortbreiten arbeiten müssen. Beachten Sie jedoch den möglichen Datenverlust bei der Konvertierung.

INT

F_INT_AS_STRING
• – ◦ • – ◦ • – ◦ •

Einleitung
Der Funktionsblock `F_INT_AS_STRING` dient der Konvertierung eines ganzzahligen Wertes (`INT`) in eine Zeichenkette (`STRING`). Diese Funktionalität ist besonders nützlich, wenn numerische Werte in einer für den Benutzer lesbaren Form ausgegeben oder in einer Textverarbeitung weiterverarbeitet werden sollen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung des Eingabewertes. Dieser Ereigniseingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ereignisausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (`INT`): Der ganzzahlige Wert, der in eine Zeichenkette umgewandelt werden soll.

Daten-Ausgänge

- **OUT** (`STRING`): Die resultierende Zeichenkette nach der Konvertierung des Eingabewertes.

Adapter

- Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Bei Auslösung des Ereignisses `REQ` wird der Algorithmus ausgeführt, der den ganzzahligen Wert `IN` in eine Zeichenkette konvertiert und das Ergebnis an `OUT` ausgibt. Anschließend wird das Ereignis `CNF` ausgelöst, um den erfolgreichen Abschluss der Konvertierung zu signalisieren.

Technische Besonderheiten

- Die Konvertierung erfolgt mittels der Funktion `INT_AS_STRING`, die den `INT`-Wert direkt in einen `STRING` umwandelt.
- Der Funktionsblock ist einfach und effizient, da er keine komplexen Zustandsmaschinen oder zusätzliche Verarbeitungsschritte erfordert.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine expliziten Zustände. Die Funktionalität wird direkt bei Auslösung des `REQ`-Ereignisses ausgeführt.

Anwendungsszenarien

- Anzeige von numerischen Werten in Benutzeroberflächen oder Logs.
- Vorbereitung von Daten für die Ausgabe in Textformaten (z.B. CSV, JSON).
- Integration in Systeme, die Zeichenketten zur weiteren Verarbeitung benötigen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken ist `F_INT_AS_STRING` spezialisiert auf die Umwandlung von `INT` zu `STRING` und daher effizienter und einfacher in der Handhabung.
- Andere Blöcke könnten zusätzliche Parameter wie Formatierung oder Locale-Einstellungen unterstützen, was hier nicht der Fall ist.

Fazit

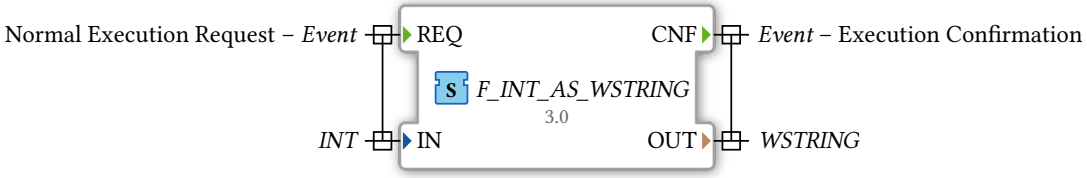
Der `F_INT_AS_STRING` Funktionsblock bietet eine einfache und effektive Möglichkeit, ganzzahlige Werte in Zeichenketten umzuwandeln. Seine Einfachheit und direkte Funktionsweise machen ihn ideal für Anwendungen, bei denen eine schnelle und unkomplizierte Konvertierung benötigt wird.

F_INT_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_INT_AS_WSTRING` dient der Konvertierung eines ganzzahligen Wertes (`INT`) in eine breite Zeichenkette (`WSTRING`). Diese Funktionalität ist besonders nützlich, wenn numerische Werte für die Anzeige oder Weiterverarbeitung als Text benötigt werden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Eingang ausgelöst, wird der Wert am Daten-Eingang `IN` verarbeitet.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der konvertierte Wert steht am Daten-Ausgang `OUT` zur Verfügung.

Daten-Eingänge

- **IN**: Der ganzzahlige Wert (`INT`), der in eine breite Zeichenkette konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Die resultierende breite Zeichenkette (`WSTRING`) nach der Konvertierung.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock nutzt die eingebaute Funktion `INT_AS_WSTRING`, um den ganzzahligen Eingabewert in eine breite Zeichenkette umzuwandeln. Die Konvertierung wird durch das Ereignis `REQ` initiiert und durch das Ereignis `CNF` bestätigt.

Technische Besonderheiten

- Der Funktionsblock ist einfach aufgebaut und führt nur eine grundlegende Datentypkonvertierung durch.
- Es werden keine zusätzlichen Parameter oder komplexen Algorithmen benötigt.

Zustandsübersicht

1. **Bereit**: Wartet auf das Ereignis `REQ`.
2. **Aktiv**: Führt die Konvertierung durch und setzt den Ausgang `OUT`.
3. **Bestätigt**: Sendet das Ereignis `CNF` und kehrt in den Bereitschaftszustand zurück.

Anwendungsszenarien

- Anzeige von numerischen Werten in Benutzeroberflächen.
- Protokollierung von numerischen Daten in Textform.
- Vorbereitung von Daten für die Kommunikation mit Systemen, die Textformate erwarten.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_DINT_AS_WSTRING**: Konvertiert einen längeren ganzzahligen Wert (`DINT`) in eine breite Zeichenkette.
- **F_REAL_AS_WSTRING**: Konvertiert einen Fließkommawert (`REAL`) in eine breite Zeichenkette.
- **F_STRING_AS_INT**: Führt die umgekehrte Konvertierung durch (Zeichenkette zu Ganzzahl).

Fazit

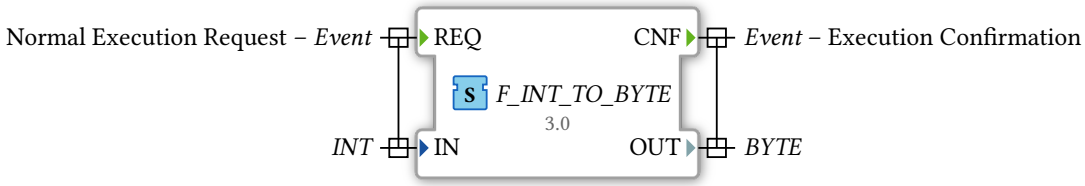
Der `F_INT_AS_WSTRING` Funktionsblock ist ein einfaches und effizientes Werkzeug für die Konvertierung von Ganzzahlen in breite Zeichenketten. Seine klare Schnittstelle und direkte Funktionsweise machen ihn zu einer zuverlässigen Lösung für grundlegende Datentypumwandlungen.

F_INT_TO_BYTE

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_INT_TO_BYTE` dient zur Konvertierung eines Integer-Werts (`INT`) in einen Byte-Wert (`BYTE`). Er ist Teil des Pakets `iec61131::conversion` und bietet eine einfache und effiziente Möglichkeit, Datentypen innerhalb von IEC 61499-Systemen umzuwandeln.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Dieser Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN**: Der Integer-Wert (`INT`), der in einen Byte-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der resultierende Byte-Wert (`BYTE`) nach der Konvertierung.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Empfang eines Ereignisses am Eingang `REQ` wird der Algorithmus ausgeführt, der den Wert von `IN` von `INT` zu `BYTE` konvertiert und das Ergebnis an `OUT` ausgibt. Anschließend wird das Ereignis `CNF` ausgelöst, um den erfolgreichen Abschluss der Konvertierung zu signalisieren.

Der Algorithmus ist wie folgt definiert:

```
ALGORITHM REQ
OUT := INT_TO_BYTE(IN);
END_ALGORITHM
```

Technische Besonderheiten

- Der Funktionsblock ist einfach und effizient implementiert, ohne zusätzliche Zustände oder komplexe Logik.
- Die Konvertierung erfolgt direkt und ohne Verzögerung.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine expliziten Zustände. Die Ausführung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Typkonvertierung in Steuerungsanwendungen, wo Integer-Werte in Byte-Werte umgewandelt werden müssen.

- Integration in größere Systeme, die eine strenge Typenkontrolle erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_DINT_TO_BYTE` oder `F_WORD_TO_BYTE` bieten Konvertierungen zwischen anderen Datentypen, sind aber für unterschiedliche Quelltypen ausgelegt.
- `F_INT_TO_BYTE` ist spezifisch für die Konvertierung von `INT` zu `BYTE` optimiert.

Fazit

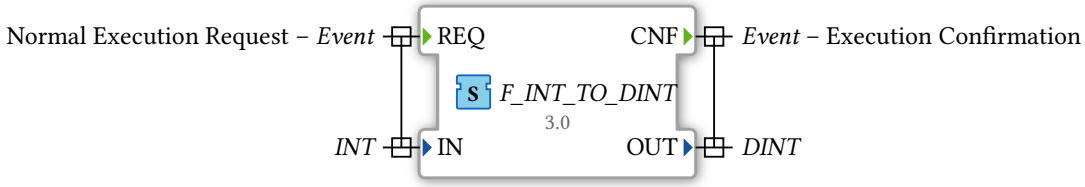
Der Funktionsblock `F_INT_TO_BYTE` ist ein nützliches Werkzeug für die Typkonvertierung in IEC 61499-basierten Systemen. Seine einfache und effiziente Implementierung macht ihn ideal für Anwendungen, die schnelle und zuverlässige Konvertierungen erfordern.

F_INT_TO_DINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_INT_TO_DINT` dient zur Konvertierung eines 16-Bit Integer-Werts (`INT`) in einen 32-Bit Integer-Wert (`DINT`). Diese Konvertierung ist nützlich, wenn Daten zwischen Systemen oder Komponenten ausgetauscht werden sollen, die unterschiedliche Integer-Formate verwenden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (`INT`): Der 16-Bit Integer-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (`DINT`): Der resultierende 32-Bit Integer-Wert nach der Konvertierung.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt eine direkte Typumwandlung durch. Wenn das `REQ`-Ereignis eintrifft, wird der Wert von `IN` nach `OUT` kopiert. Dabei wird der 16-Bit `INT`-Wert automatisch in einen 32-Bit `DINT`-Wert erweitert. Nach Abschluss der Konvertierung wird das `CNF`-Ereignis ausgelöst.

Technische Besonderheiten

- Die Konvertierung ist verlustfrei, da ein `INT`-Wert problemlos in einen `DINT`-Wert umgewandelt werden kann.
- Der Funktionsblock ist deterministisch und hat keine internen Zustände.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Er reagiert ausschließlich auf das `REQ`-Ereignis mit einer sofortigen Konvertierung und der Ausgabe des `CNF`-Ereignisses.

Anwendungsszenarien

- Datenkommunikation zwischen Systemen, die unterschiedliche Integer-Formate verwenden.
- Erweiterung von Integer-Werten für weitere Berechnungen oder Speicherung.
- Kompatibilität mit Bibliotheken oder Funktionen, die `DINT`-Werte erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken ist `F_INT_TO_DINT` spezialisiert und optimiert für die Umwandlung von `INT` nach `DINT`.
- Andere Konvertierungsblöcke könnten zusätzliche Funktionen wie Überlaufprüfungen oder Skalierungen bieten, was hier nicht notwendig ist.

Fazit

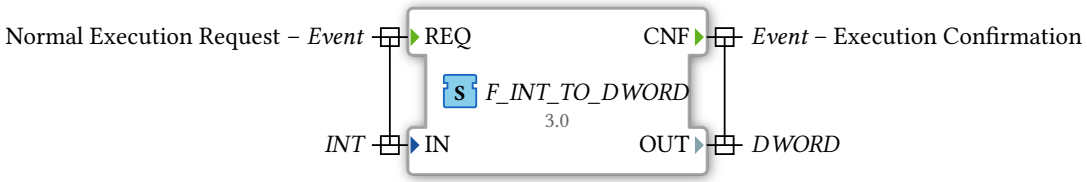
Der `F_INT_TO_DINT`-Funktionsblock ist ein einfacher und effizienter Baustein zur verlustfreien Konvertierung von 16-Bit zu 32-Bit Integer-Werten. Er ist besonders nützlich in Szenarien, wo eine Typanpassung ohne zusätzliche Berechnungen erforderlich ist.

F_INT_TO_DWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_INT_TO_DWORD` dient der Konvertierung eines Integer-Werts (INT) in einen Double-Word-Wert (DWORD). Diese Konvertierung ist besonders nützlich, wenn Daten zwischen Systemen ausgetauscht werden, die unterschiedliche Datentypen verwenden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Dieser Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Bestätigt die erfolgreiche Ausführung der Konvertierung. Dieser Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (INT): Der Integer-Wert, der in einen DWORD-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT** (DWORD): Der resultierende DWORD-Wert nach der Konvertierung.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung von INT zu DWORD durch, sobald das Ereignis `REQ` ausgelöst wird. Die Konvertierung erfolgt mittels der Funktion `INT_TO_DWORD`, die den Eingabewert `IN` in den Ausgabewert `OUT` umwandelt. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher Funktionsblock (SimpleFB) und verwendet einen ST-Algorithmus für die Konvertierung.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Verzögerung.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar nach Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Datenkonvertierung in Steuerungssystemen, die unterschiedliche Datentypen verwenden.
- Integration von Systemen, die INT-Werte erzeugen, aber DWORD-Werte benötigen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen wie `F_INT_TO_REAL` oder `F_DWORD_TO_INT` ist dieser Funktionsblock spezialisiert auf die Konvertierung von INT zu DWORD.
- Einfacher und direkter als universelle Konvertierungsbausteine, die mehrere Datentypen unterstützen.

Fazit

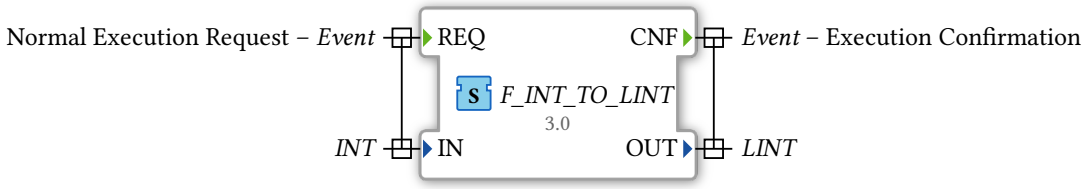
Der `F_INT_TO_DWORD` Funktionsblock ist ein effizientes und einfaches Werkzeug für die Konvertierung von Integer- zu Double-Word-Werten. Seine klare Schnittstelle und direkte Funktionsweise machen ihn ideal für Anwendungen, die eine schnelle und zuverlässige Datentypkonvertierung erfordern.

F_INT_TO_LINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_INT_TO_LINT` dient der Konvertierung eines 16-Bit Integer-Werts (`INT`) in einen 64-Bit Long Integer-Wert (`LINT`). Dieser Baustein ist besonders nützlich in Szenarien, wo eine Erweiterung des Wertebereichs oder eine Typanpassung für nachfolgende Verarbeitungsschritte erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (`INT`): Der 16-Bit Integer-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (`LINT`): Der resultierende 64-Bit Long Integer-Wert nach der Konvertierung.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt eine direkte Zuweisung des Eingabewerts `IN` vom Typ `INT` zum Ausgabewert `OUT` vom Typ `LINT` durch. Die Konvertierung erfolgt implizit durch die Typumwandlung, wobei der numerische Wert erhalten bleibt.

Technische Besonderheiten

- **Einfache Implementierung**: Der Algorithmus besteht aus einer einzigen Zuweisung.

- **Keine Datenverluste**: Da `LINT` einen größeren Wertebereich als `INT` besitzt, gibt es keine Überlaufprobleme.
- **Deterministisches Verhalten**: Die Ausführung ist vorhersehbar und schnell.

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand. Jeder Aufruf über `REQ` führt zur sofortigen Ausführung und Ausgabe über `CNF`.

Anwendungsszenarien

- **Datenverarbeitung**: Erweiterung von Integer-Werten für nachfolgende Berechnungen mit größerem Wertebereich.
- **Schnittstellenanpassung**: Anpassung von Daten für Komponenten, die `LINT` erwarten.
- **Protokollierung**: Speicherung von `INT`-Werten in Systemen, die `LINT` verwenden.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_INT_TO_DINT**: Konvertiert `INT` zu `DINT` (32-Bit), bietet weniger Wertebereich als `LINT`.
- **F_DINT_TO_LINT**: Konvertiert `DINT` zu `LINT`, ähnlich aber mit anderem Eingangstyp.
- **F_LINT_TO_INT**: Umgekehrte Konvertierung, potenziell mit Datenverlust.

Fazit

Der `F_INT_TO_LINT` Funktionsblock ist ein einfaches und effizientes Werkzeug zur Typkonvertierung in 4diac-IDE. Seine Stärke liegt in der direkten und verlustfreien Umwandlung, was ihn ideal für Anwendungen mit erweitertem Wertebedarf macht.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

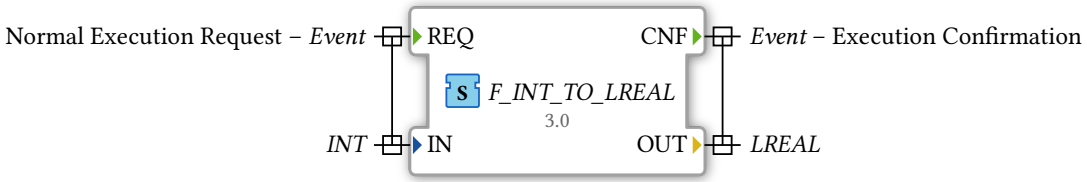
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

F_INT_TO_LREAL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_INT_TO_LREAL` dient der Konvertierung eines ganzzahligen Wertes (INT) in eine 64-Bit Gleitkommazahl (LREAL). Dieser Baustein ist besonders nützlich in Anwendungen, bei denen eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Dieser Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `INT`, der in einen `LREAL`-Wert umgewandelt werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgangswert vom Typ `LREAL`, der das Ergebnis der Konvertierung enthält.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem der ganzzahlige Eingangswert `IN` direkt dem Gleitkomma-Ausgangswert `OUT` zugewiesen wird. Die Konvertierung erfolgt automatisch beim Eintreffen eines Ereignisses am `REQ`-Eingang. Nach Abschluss der Konvertierung wird das `CNF`-Ereignis ausgelöst.

Technische Besonderheiten

- Die Konvertierung erfolgt ohne zusätzliche Parameter oder Konfiguration.
- Der Baustein ist für die Verwendung in der 4diac-IDE optimiert und unterstützt die Echtzeitverarbeitung.

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf ein `REQ`-Ereignis.
2. **Konvertierungszustand**: Führt die Typumwandlung durch.
3. **Bestätigungszustand**: Sendet das `CNF`-Ereignis und kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, bei denen Gleitkommaoperationen erforderlich sind.
- Integration in größere Steuerungssysteme, die gemischte Datentypen verarbeiten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsbausteinen ist `F_INT_TO_LREAL` spezialisiert auf die Umwandlung von `INT` zu `LREAL` und daher effizienter in dieser spezifischen Aufgabe.
- Andere Bausteine könnten zusätzliche Funktionen wie Skalierung oder Bereichsprüfung bieten, was hier nicht der Fall ist.

Fazit

Der `F_INT_TO_LREAL`-Funktionsblock ist ein einfacher und effizienter Baustein für die Konvertierung von ganzzahligen Werten in Gleitkommazahlen. Seine Einfachheit und direkte Funktionsweise machen ihn zu einer zuverlässigen Wahl für grundlegende Typumwandlungen in der Automatisierungstechnik.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

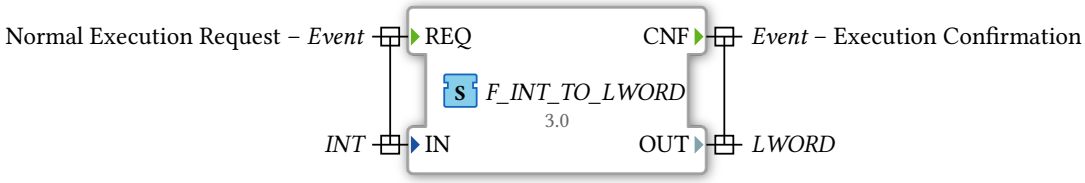
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

F_INT_TO_LWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_INT_TO_LWORD` dient der Konvertierung eines Integer-Werts (`INT`) in einen 64-Bit langen vorzeichenlosen Integer-Wert (`LWORD`). Dieser Baustein ist besonders nützlich in Szenarien, wo eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Ausführung des Funktionsblocks aus. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert die erfolgreiche Ausführung des Funktionsblocks. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN** (`INT`): Der Eingangswert, der in einen `LWORD`-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT** (`LWORD`): Der konvertierte Ausgangswert.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt bei einem eingehenden `REQ`-Ereignis die Konvertierung des `INT`-Werts am Eingang `IN` in einen `LWORD`-Wert durch. Das Ergebnis wird am Ausgang `OUT` ausgegeben und das `CNF`-Ereignis signalisiert die erfolgreiche Ausführung.

Der Algorithmus ist wie folgt definiert:

```
ALGORITHM REQ
OUT := INT_TO_LWORD(IN);
END_ALGORITHM
```

Technische Besonderheiten

- Der Funktionsblock ist einfach und effizient, da er direkt die eingebaute `INT_TO_LWORD`-Funktion nutzt.
- Keine zusätzlichen Zustände oder komplexe Logik erforderlich.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Ausführung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo `INT`-Werte in `LWORD`-Werte konvertiert werden müssen.
- Integration in größere Systeme, die unterschiedliche Datentypen verarbeiten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_DINT_TO_LWORD` oder `F_SINT_TO_LWORD` bieten Konvertierungen von anderen Integer-Typen zu `LWORD`.
- `F_INT_TO_LWORD` ist spezifisch für die Konvertierung von `INT` zu `LWORD` optimiert.

Fazit

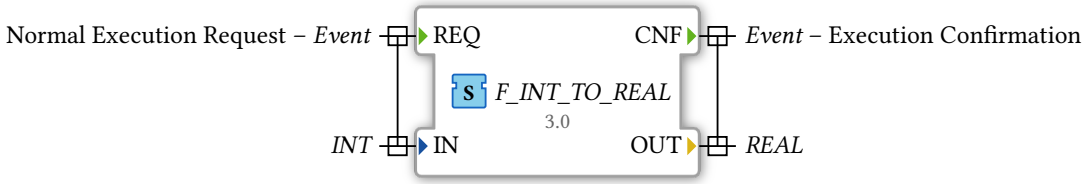
Der `F_INT_TO_LWORD`-Funktionsblock bietet eine einfache und effiziente Lösung für die Konvertierung von `INT`- zu `LWORD`-Werten. Seine klare Schnittstelle und direkte Funktionsweise machen ihn ideal für den Einsatz in Steuerungsanwendungen.

F_INT_TO_REAL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_INT_TO_REAL` dient der Konvertierung eines ganzzahligen Wertes (`INT`) in eine Fließkommazahl (`REAL`). Diese Funktionalität ist besonders nützlich, wenn in Steuerungsanwendungen Daten zwischen verschiedenen numerischen Darstellungen umgewandelt werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Dieser Eingang löst die Ausführung der Konvertierung aus. Er ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Dieser Ausgang signalisiert den Abschluss der Konvertierung. Er ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (`INT`): Der ganzzahlige Wert, der in eine Fließkommazahl umgewandelt werden soll.

Daten-Ausgänge

- **OUT** (`REAL`): Der resultierende Fließkommawert nach der Konvertierung.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Auslösung des Ereignisses `REQ` wird der Wert des Daten-Eingangs `IN` gelesen und in den Typ `REAL` konvertiert. Das Ergebnis wird am Daten-Ausgang `OUT` ausgegeben und das Ereignis `CNF` signalisiert den erfolgreichen Abschluss der Operation.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne zusätzliche Parameter oder Einstellungen.
- Der Funktionsblock ist einfach und effizient implementiert, was eine schnelle Ausführung gewährleistet.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Umwandlung von Sensorwerten (z.B. als `INT` erfasst) in Fließkommazahlen für weitere Berechnungen.
- Integration in Steuerungslogiken, die gemischte Datentypen verarbeiten müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsbausteinen ist `F_INT_TO_REAL` spezialisiert und daher effizienter für diesen spezifischen Anwendungsfall.
- Andere Bausteine könnten zusätzliche Funktionen wie Bereichsprüfungen oder Skalierungen bieten, was hier nicht erforderlich ist.

Fazit

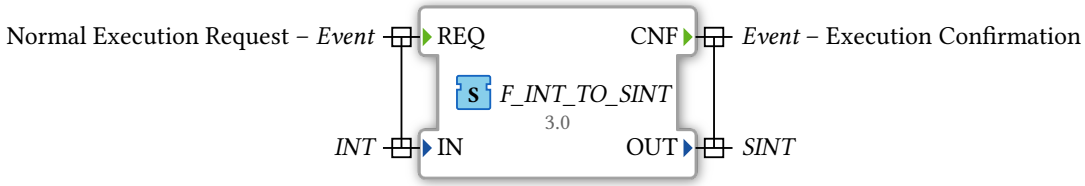
Der `F_INT_TO_REAL` Funktionsblock ist ein einfaches und effektives Werkzeug zur Konvertierung von ganzzahligen Werten in Fließkommazahlen. Seine klare Schnittstelle und direkte Funktionsweise machen ihn ideal für Anwendungen, bei denen eine schnelle und zuverlässige Typumwandlung benötigt wird.

F_INT_TO_SINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_INT_TO_SINT` dient der Konvertierung eines ganzzahligen Wertes vom Typ `INT` in einen vorzeichenbehafteten 8-Bit-Integer-Wert vom Typ `SINT`. Diese Konvertierung ist nützlich, wenn Daten zwischen Systemen mit unterschiedlichen Datentypen ausgetauscht oder verarbeitet werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `INT`, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der konvertierte Ausgangswert vom Typ `SINT`.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `INT` zu `SINT` durch, sobald das Ereignis `REQ` ausgelöst wird. Der Algorithmus innerhalb des Blocks nutzt die Funktion `INT_TO_SINT`, um die Konvertierung durchzuführen. Nach Abschluss der Konvertierung wird das Ereignis `CNF` ausgelöst, und der konvertierte Wert steht am Ausgang `OUT` zur Verfügung.

Technische Besonderheiten

- Der Funktionsblock ist einfach aufgebaut und enthält nur einen Algorithmus für die Konvertierung.
- Es gibt keine zusätzliche Fehlerbehandlung für Überlauf- oder Unterlaufbedingungen. Der Nutzer muss sicherstellen, dass der Eingabewert innerhalb des gültigen Bereichs für `SINT` liegt (-128 bis 127).

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar nach Auslösen des `REQ`-Ereignisses.

Anwendungsszenarien

- Datenkonvertierung in Steuerungssystemen, wo unterschiedliche Datentypen verarbeitet werden müssen.
- Kommunikation zwischen Systemen, die unterschiedliche Datentypen verwenden.
- Reduzierung der Speicherbelegung, wenn ein kleinerer Datentyp ausreichend ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_INT_TO_USINT**: Konvertiert `INT` zu `USINT` (vorzeichenloser 8-Bit-Integer).
- **F_INT_TO_DINT**: Konvertiert `INT` zu `DINT` (32-Bit-Integer), was keine Datenreduktion bedeutet.
- **F_INT_TO_REAL**: Konvertiert `INT` zu `REAL` (Gleitkommazahl).

Fazit

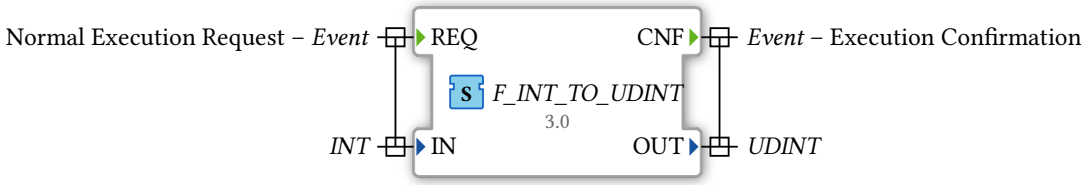
Der `F_INT_TO_SINT`-Funktionsblock ist ein einfaches und effizientes Werkzeug zur Konvertierung von `INT`- zu `SINT`-Werten. Er eignet sich besonders für Anwendungen, bei denen Speicherplatz optimiert oder Daten zwischen Systemen mit unterschiedlichen Datentypen ausgetauscht werden müssen. Nutzer sollten jedoch darauf achten, dass die Eingabewerte innerhalb des gültigen Bereichs für `SINT` liegen, um unerwartete Ergebnisse zu vermeiden.

F_INT_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_INT_TO_UDINT` dient der Konvertierung eines vorzeichenbehafteten 16-Bit Integer-Werts (`INT`) in einen vorzeichenlosen 32-Bit Integer-Wert (`UDINT`). Diese Konvertierung ist insbesondere in Szenarien erforderlich, in denen Daten zwischen Systemen mit unterschiedlichen Datentypen ausgetauscht werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der vorzeichenbehaftete 16-Bit Integer-Wert (`INT`), der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der resultierende vorzeichenlose 32-Bit Integer-Wert (`UDINT`).

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `INT` zu `UDINT` durch, sobald das Ereignis `REQ` empfangen wird. Die Konvertierung erfolgt mittels der Funktion `INT_TO_UDINT(IN)`, die den Eingabewert `IN` in den entsprechenden `UDINT`-Wert umwandelt. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst, und der konvertierte Wert steht am Ausgang `OUT` zur Verfügung.

Technische Besonderheiten

- Die Konvertierung erfolgt ohne zusätzliche Parameter oder Konfiguration.
- Der Funktionsblock ist für den Einsatz in Echtzeitsystemen geeignet, da er deterministisch arbeitet.

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf das Ereignis `REQ`.
2. **Konvertierungszustand**: Führt die Konvertierung durch und setzt den Ausgang `OUT`.
3. **Bestätigungszustand**: Löst das Ereignis `CNF` aus und kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- Datenkonvertierung in Steuerungssystemen, die unterschiedliche Datentypen verwenden.
- Integration von Sensordaten, die als `INT` vorliegen, in Systeme, die `UDINT` erfordern.
- Kompatibilitätslayer zwischen verschiedenen Hardware- oder Softwarekomponenten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_DINT_TO_UDINT` oder `F_SINT_TO_UDINT` bieten Konvertierungen zwischen anderen Datentypen, jedoch nicht speziell für `INT` zu `UDINT`.
- Im Gegensatz zu generischen Konvertierungsblöcken ist `F_INT_TO_UDINT` auf eine spezifische Konvertierung optimiert.

Fazit

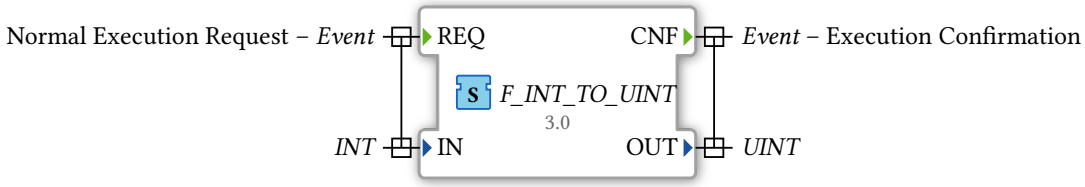
Der Funktionsblock `F_INT_TO_UDINT` ist ein spezialisiertes und effizientes Werkzeug zur Konvertierung von vorzeichenbehafteten 16-Bit Integer-Werten in vorzeichenlose 32-Bit Integer-Werte. Seine einfache Schnittstelle und deterministische Funktionsweise machen ihn ideal für den Einsatz in Echtzeitanwendungen.

F_INT_TO_UINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_INT_TO_UINT` dient der Konvertierung eines vorzeichenbehafteten Integer-Werts (`INT`) in einen vorzeichenlosen Integer-Wert (`UINT`). Diese Konvertierung ist nützlich, wenn Daten zwischen Systemen ausgetauscht werden, die unterschiedliche Datentypen verwenden, oder wenn eine vorzeichenlose Verarbeitung erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN** (`INT`): Der vorzeichenbehaftete Integer-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (`UINT`): Der resultierende vorzeichenlose Integer-Wert nach der Konvertierung.

Adapter

Es werden keine Adapter verwendet.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `INT` zu `UINT` durch, wenn das Ereignis `REQ` ausgelöst wird. Die Konvertierung erfolgt mittels der Funktion `INT_TO_UINT(IN)`. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst, und der konvertierte Wert steht am Ausgang `OUT` zur Verfügung.

Technische Besonderheiten

- Die Konvertierung erfolgt ohne zusätzliche Parameter oder Konfiguration.
- Der Block ist als einfacher Funktionsblock (`SimpleFB`) implementiert und verwendet einen ST-Algorithmus für die Konvertierung.

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf das `REQ`-Ereignis.
2. **Konvertierungszustand**: Führt die Konvertierung durch und setzt den Ausgang `OUT`.
3. **Bestätigungszustand**: Löst das `CNF`-Ereignis aus und kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- **Datenverarbeitung**: Konvertierung von Sensorwerten oder Steuerungsdaten, die als `INT` vorliegen, für Systeme, die `UINT` erwarten.

- **Kommunikation**: Vorbereitung von Daten für die Übertragung zwischen Systemen mit unterschiedlichen Datentypanforderungen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_UINT_TO_INT**: Führt die umgekehrte Konvertierung von `UINT` zu `INT` durch.
- **Generische Konvertierungsblöcke**: Bieten möglicherweise mehr Flexibilität, sind aber auch komplexer in der Handhabung.

Fazit

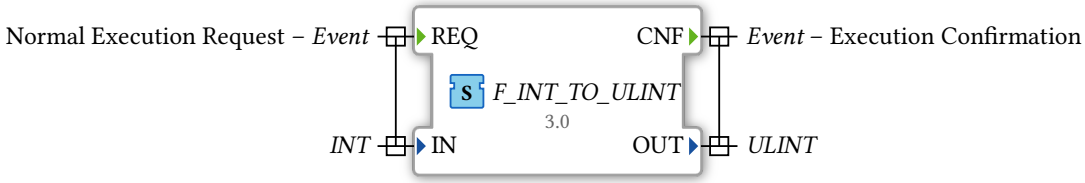
Der `F_INT_TO_UINT`-Funktionsblock ist ein einfaches und effizientes Werkzeug für die Konvertierung von vorzeichenbehafteten zu vorzeichenlosen Integer-Werten. Seine klare Schnittstelle und einfache Funktionsweise machen ihn ideal für Anwendungen, bei denen eine schnelle und zuverlässige Typumwandlung erforderlich ist.

F_INT_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_INT_TO_ULINT` dient der Konvertierung eines ganzzahligen Wertes vom Datentyp `INT` in einen vorzeichenlosen 64-Bit-Ganzzahlwert vom Datentyp `ULINT`. Diese Konvertierung ist besonders nützlich, wenn Werte zwischen unterschiedlichen Systemen oder Komponenten ausgetauscht werden müssen, die unterschiedliche Datentypen verwenden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung des Eingabewertes. Dieser Ereigniseingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung und gibt den konvertierten Wert über den Daten-Ausgang `OUT` aus.

Daten-Eingänge

- **IN**: Der Eingabewert vom Datentyp `INT`, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der konvertierte Ausgabewert vom Datentyp `ULINT`.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` empfangen wird. Der Algorithmus innerhalb des Funktionsblocks wandelt den `INT`-Wert am Eingang `IN` in einen `ULINT`-Wert um und stellt das Ergebnis am Ausgang `OUT` bereit. Die erfolgreiche Konvertierung wird durch das Auslösen des Ereignisses `CNF` signalisiert.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne zusätzliche Verzögerung.
- Der Funktionsblock ist für die Verwendung in Echtzeitsystemen geeignet.
- Die Implementierung erfolgt gemäß dem IEC 61131-3 Standard.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung wird bei jedem Empfang des `REQ`-Ereignisses durchgeführt.

Anwendungsszenarien

- Konvertierung von Sensorwerten für die Verarbeitung in Systemen, die `ULINT`-Werte erfordern.
- Integration von Komponenten, die unterschiedliche Datentypen verwenden.
- Verwendung in Steuerungssystemen, die eine Typumwandlung benötigen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen wie `F_DINT_TO_ULINT` oder `F_SINT_TO_ULINT` ist dieser Funktionsblock speziell für die Konvertierung von `INT` zu `ULINT` optimiert.
- Ähnliche Bausteine könnten zusätzliche Funktionen wie Bereichsprüfungen oder Fehlerbehandlung bieten, was dieser Funktionsblock nicht tut.

Fazit

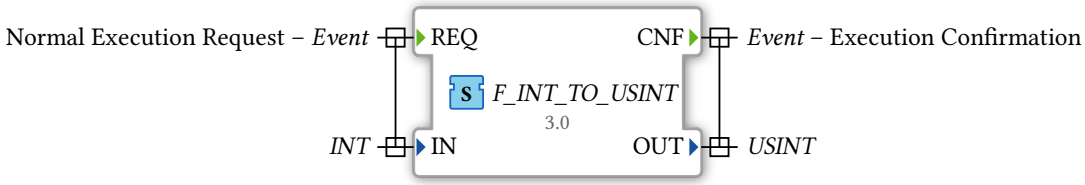
Der Funktionsblock `F_INT_TO_ULINT` ist ein einfacher und effizienter Baustein zur Konvertierung von `INT`-Werten in `ULINT`-Werte. Er ist besonders in Szenarien nützlich, wo eine direkte und schnelle Typumwandlung benötigt wird. Die klare Schnittstellenstruktur und die einfache Funktionsweise machen ihn zu einem zuverlässigen Werkzeug in der Automatisierungstechnik.

F_INT_TO_USINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_INT_TO_USINT` dient der Konvertierung eines ganzzahligen Wertes vom Datentyp `INT` (16-Bit Ganzzahl mit Vorzeichen) in den Datentyp `USINT` (8-Bit Ganzzahl ohne Vorzeichen). Dieser Baustein ist besonders nützlich in Szenarien, wo eine Typumwandlung zwischen unterschiedlich großen numerischen Werten erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Typ: `Event`): Startet die Ausführung der Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF** (Typ: `Event`): Signalisiert den erfolgreichen Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (Typ: `INT`): Der Eingangswert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (Typ: `USINT`): Das Ergebnis der Konvertierung von `INT` zu `USINT`.

Adapter

Dieser Funktionsblock verfügt über keine Adapter-Schnittstellen.

Funktionsweise

Bei Auslösung des Ereigniseingangs `REQ` wird der Algorithmus des Funktionsblocks ausgeführt. Dieser wandelt den Wert am Daten-Eingang `IN` vom Typ `INT` in den Typ `USINT` um und speichert das Ergebnis

im Daten-Ausgang `OUT`. Anschließend wird das Ereignis `CNF` ausgelöst, um den erfolgreichen Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Die Konvertierung erfolgt durch die Funktion `INT_TO_USINT`, die den `INT`-Wert in einen `USINT`-Wert umwandelt. Dabei kann es zu einem Datenverlust kommen, wenn der ursprüngliche `INT`-Wert außerhalb des darstellbaren Bereichs von `USINT` (0 bis 255) liegt.
- Der Funktionsblock ist Teil des Pakets `iec61131::conversion`, das verschiedene Typkonvertierungen bereitstellt.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Ausführung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Typkonvertierung in Steuerungsanwendungen, wo unterschiedliche Datentypen verarbeitet werden müssen.
- Reduzierung der Speicherbelegung durch Konvertierung in kleinere Datentypen, wo möglich.
- Vorbereitung von Daten für die Kommunikation mit Systemen, die nur `USINT`-Werte verarbeiten können.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_DINT_TO_USINT` oder `F_SINT_TO_USINT` bieten Konvertierungen von anderen Ganzzahltypen zu `USINT`. `F_INT_TO_USINT` ist spezifisch für die Umwandlung von `INT` zu `USINT` optimiert.
- Im Gegensatz zu generischen Konvertierungsblöcken bietet dieser Baustein eine direkte und effiziente Lösung für die spezifische Typumwandlung.

Fazit

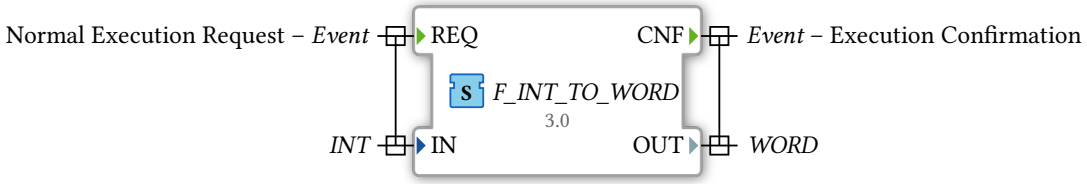
Der `F_INT_TO_USINT` Funktionsblock ist ein einfacher und effizienter Baustein zur Konvertierung von `INT`- zu `USINT`-Werten. Er eignet sich ideal für Anwendungen, die eine präzise und spezifische Typumwandlung erfordern, insbesondere in Umgebungen mit begrenzten Ressourcen oder spezifischen Datentypanforderungen.

F_INT_TO_WORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_INT_TO_WORD` dient der Konvertierung eines ganzzahligen Wertes (INT) in einen 16-Bit-Wortwert (WORD). Diese Konvertierung ist besonders nützlich in Szenarien, wo Daten zwischen Systemen mit unterschiedlichen Datentypen ausgetauscht werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Eingang ausgelöst, wird der Wert am Daten-Eingang `IN` gelesen und konvertiert.

Ereignis-Ausgänge

- **CNF**: Signalisiert den erfolgreichen Abschluss der Konvertierung. Gleichzeitig wird der konvertierte Wert am Daten-Ausgang `OUT` ausgegeben.

Daten-Eingänge

- **IN**: Der Eingang für den ganzzahligen Wert (INT), der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgang für den konvertierten 16-Bit-Wortwert (WORD).

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` ausgelöst wird. Dabei wird der Wert am Eingang `IN` gelesen und mit der Funktion `INT_TO_WORD` in einen WORD-Wert umgewandelt. Das Ergebnis wird am Ausgang `OUT` ausgegeben und gleichzeitig das Ereignis `CNF` ausgelöst, um den erfolgreichen Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Der Funktionsblock ist einfach und effizient implementiert, ohne zusätzliche Zustände oder komplexe Logik.
- Die Konvertierung erfolgt direkt und ohne Verzögerung.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Datenkonvertierung in Steuerungssystemen, wo unterschiedliche Datentypen verarbeitet werden müssen.
- Schnittstellen zwischen Systemen, die unterschiedliche Datentypen verwenden.
- Protokollierung und Überwachung, wo Daten in einheitlichen Formaten benötigt werden.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken ist `F_INT_TO_WORD` spezialisiert auf die Konvertierung von INT zu WORD und daher effizienter in dieser spezifischen Aufgabe.
- Andere Blöcke wie `F_DINT_TO_WORD` oder `F_SINT_TO_WORD` bieten ähnliche Funktionalität, aber für unterschiedliche Quell-Datentypen.

Fazit

Der `F_INT_TO_WORD` Funktionsblock ist ein einfaches und effektives Werkzeug für die Konvertierung von ganzzahligen Werten in 16-Bit-Wortwerte. Seine klare Schnittstelle und direkte Funktionsweise machen ihn zu einer zuverlässigen Komponente in Steuerungs- und Automatisierungssystemen.

INT Konvertierungen

🎧 Podcast

- Eclipse 4diac 3.0: ST-Interpreter, FBE und 7200 Commits – Der Turbo für verteilte Automatisierung
- Von der SPS zur verteilten Intelligenz: IEC 61499 für Industrie 4.0 verstehen
- Simplifying Industrial Control: Your Deep Dive into Eclipse 4diac and IEC 61499
- Ereignisse und IEC 61499: Der Startschuss für intelligente Systeme
- ERP-Systeme: Das digitale Gehirn – Chancen, Risiken und der Weg zur intelligenten Produktion

🛠️ Zugehörige Übungen

- SR_TO_QX_TRIG_EO
- SINT_TO_Q_NumericValue
- Softkey_T_FF_ILOCK_TO_QX_BG
- Bool8_TO_logiBUS_QX8

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

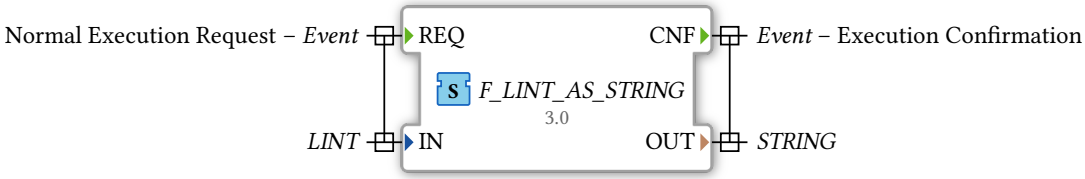
LINT

F_LINT_AS_STRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LINT_AS_STRING` dient der Konvertierung eines LINT-Datentyps (64-Bit-Ganzzahl) in einen STRING-Datentyp (Zeichenkette). Dieser Baustein ist besonders nützlich, wenn numerische Werte für die Ausgabe oder Weiterverarbeitung als Text benötigt werden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Bei Aktivierung dieses Ereignisses wird der Wert am Daten-Eingang `IN` verarbeitet.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieses Ereignis wird zusammen mit dem konvertierten Wert am Daten-Ausgang `OUT` ausgegeben.

Daten-Eingänge

- **IN** (LINT): Der Eingang für den zu konvertierenden LINT-Wert.

Daten-Ausgänge

- **OUT** (STRING): Der Ausgang für den konvertierten STRING-Wert.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt bei Aktivierung des `REQ`-Ereignisses die Konvertierung des LINT-Wertes am Eingang `IN` in einen STRING-Wert durch. Das Ergebnis wird am Ausgang `OUT` ausgegeben und durch das `CNF`-Ereignis bestätigt.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher Baustein (SimpleFB) und verwendet einen ST-Algorithmus für die Konvertierung.
- Die Konvertierung erfolgt direkt durch die Funktion `LINT_AS_STRING(IN)`.

Zustandsübersicht

1. **Initialisierung**: Der Funktionsblock wartet auf das `REQ`-Ereignis.
2. **Konvertierung**: Bei `REQ` wird der LINT-Wert konvertiert.
3. **Bestätigung**: Das `CNF`-Ereignis wird ausgelöst und der konvertierte Wert ausgegeben.

Anwendungsszenarien

- Anzeige von numerischen Werten in Benutzeroberflächen.
- Protokollierung von numerischen Daten in Textdateien.
- Vorbereitung von Daten für die Kommunikation über Text-basierte Protokolle.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen wie `F_INT_AS_STRING` oder `F_DINT_AS_STRING` unterstützt dieser Baustein den größeren LINT-Datentyp (64-Bit).

Fazit

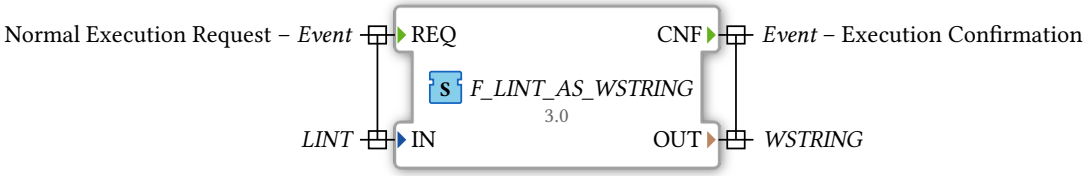
Der `F_LINT_AS_STRING` Funktionsblock bietet eine einfache und effiziente Möglichkeit, große Ganzzahlen in lesbare Zeichenketten umzuwandeln. Durch seine klare Schnittstelle und einfache Funktionsweise ist er ideal für Anwendungen, die eine Konvertierung zwischen diesen Datentypen erfordern.

F_LINT_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LINT_AS_WSTRING` dient zur Konvertierung eines LINT-Wertes (64-Bit-Ganzzahl) in einen WSTRING (Wide-String). Dieser Baustein ist besonders nützlich in Szenarien, wo numerische Werte als Zeichenketten ausgegeben oder weiterverarbeitet werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung des LINT-Wertes zu einem WSTRING. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN** (LINT): Der LINT-Wert, der in einen WSTRING konvertiert werden soll.

Daten-Ausgänge

- **OUT** (WSTRING): Der resultierende WSTRING nach der Konvertierung.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das `REQ`-Ereignis eintrifft. Dabei wird der LINT-Wert von `IN` in einen WSTRING umgewandelt und an `OUT` ausgegeben. Die Konvertierung wird durch das `CNF`-Ereignis bestätigt.

Technische Besonderheiten

- Der Baustein nutzt die `LINT_AS_WSTRING`-Funktion zur Konvertierung.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Parameter.

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf das `REQ`-Ereignis.
2. **Konvertierungszustand**: Führt die Konvertierung durch und gibt das Ergebnis aus.
3. **Bestätigungszustand**: Sendet das `CNF`-Ereignis zur Bestätigung der Konvertierung.

Anwendungsszenarien

- Anzeige von numerischen Werten in Benutzeroberflächen.

- Protokollierung von numerischen Daten in Textdateien.
- Kommunikation mit Systemen, die String-Daten erwarten.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_INT_AS_WSTRING**: Konvertiert INT (16-Bit) zu WSTRING.
- **F_DINT_AS_WSTRING**: Konvertiert DINT (32-Bit) zu WSTRING.
- **F_LINT_AS_STRING**: Konvertiert LINT zu STRING (nicht Wide-String).

Fazit

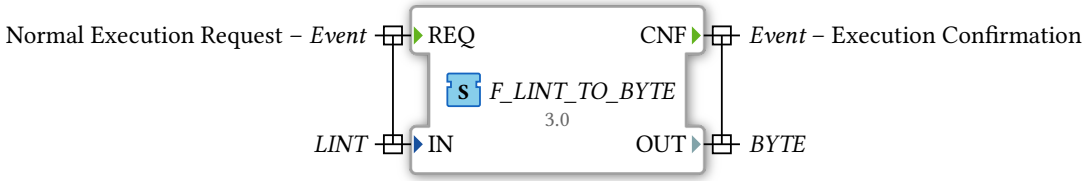
Der `F_LINT_AS_WSTRING`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, 64-Bit-Ganzzahlen in Wide-Strings umzuwandeln. Durch seine klare Schnittstelle und direkte Funktionsweise ist er ideal für Anwendungen, die eine Konvertierung zwischen diesen Datentypen erfordern.

F_LINT_TO_BYTE

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LINT_TO_BYTE` dient der Konvertierung eines LINT-Datentyps (64-Bit Ganzzahl) in einen BYTE-Datentyp (8-Bit Ganzzahl). Dieser Baustein ist besonders nützlich in Szenarien, wo eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Ausführung des Funktionsblocks aus. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert die erfolgreiche Ausführung der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Eingang für den LINT-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Ausgang für den konvertierten BYTE-Wert.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von LINT zu BYTE durch, sobald das Ereignis `REQ` ausgelöst wird. Die Konvertierung erfolgt mittels der Funktion `LINT_TO_BYTE(IN)`, wobei der Wert von `IN` in den BYTE-Typ umgewandelt und an `OUT` ausgegeben wird. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher Algorithmus (SimpleFB) und führt keine komplexen Zustandsübergänge durch.
- Die Konvertierung kann zu Datenverlust führen, wenn der LINT-Wert außerhalb des darstellbaren Bereichs eines BYTE (0 bis 255) liegt.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge. Die Ausführung erfolgt direkt bei Auslösung von `REQ`.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo LINT-Daten in BYTE-Daten umgewandelt werden müssen.
- Verwendung in Datenverarbeitungsketten, wo unterschiedliche Datentypen kompatibel gemacht werden müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_INT_TO_BYTE` oder `F_DINT_TO_BYTE` führen ähnliche Konvertierungen durch, jedoch mit anderen Ausgangsdatentypen.
- `F_LINT_TO_BYTE` ist spezifisch für die Konvertierung von 64-Bit zu 8-Bit Ganzzahlen.

Fazit

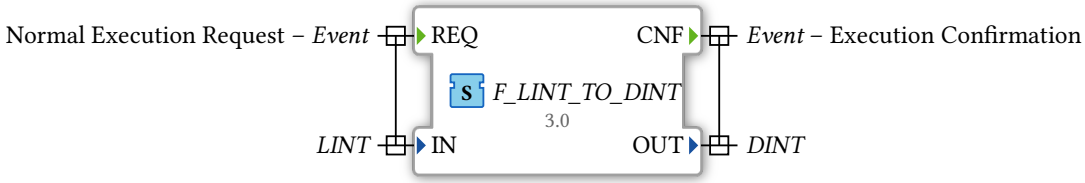
Der `F_LINT_TO_BYTE` Funktionsblock bietet eine einfache und effiziente Möglichkeit, LINT-Daten in BYTE-Daten umzuwandeln. Seine Verwendung ist in Szenarien mit Typkonvertierungen unerlässlich, jedoch ist auf mögliche Datenverluste bei großen Werten zu achten.

F_LINT_TO_DINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LINT_TO_DINT` dient der Konvertierung eines 64-Bit Integer-Werts (LINT) in einen 32-Bit Integer-Wert (DINT). Diese Konvertierung ist nützlich, wenn Daten zwischen Systemen mit unterschiedlicher Wortbreite ausgetauscht werden müssen oder wenn Speicherplatz optimiert werden soll.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung der Konvertierung. Dieser Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (LINT): Der 64-Bit Integer-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (DINT): Der resultierende 32-Bit Integer-Wert nach der Konvertierung.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem der Algorithmus `REQ` aufgerufen wird, sobald das Ereignis `REQ` eintrifft. Der Algorithmus nutzt die Funktion `LINT_TO_DINT`, um den Wert von `IN` nach `OUT` zu konvertieren. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Der Funktionsblock ist Teil des Pakets `iec61131::conversion`.
- Es handelt sich um einen einfachen Funktionsblock ohne interne Zustände oder komplexe Logik.
- Die Konvertierung kann zu einem Datenverlust führen, wenn der ursprüngliche LINT-Wert außerhalb des darstellbaren Bereichs eines DINT liegt.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge oder interne Zustände. Die Logik wird direkt bei Eintreffen des `REQ`-Ereignisses ausgeführt.

Anwendungsszenarien

- Datenkompatibilität zwischen Systemen mit unterschiedlicher Wortbreite.
- Speicheroptimierung, wenn 64-Bit Werte nicht benötigt werden.
- Integration in größere Steuerungssysteme, die DINT-Werte erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu generischen Konvertierungsblöcken ist `F_LINT_TO_DINT` spezialisiert auf die Konvertierung von LINT zu DINT.
- Andere Blöcke könnten zusätzliche Funktionen wie Bereichsprüfungen oder Fehlerbehandlung bieten, was hier nicht der Fall ist.

Fazit

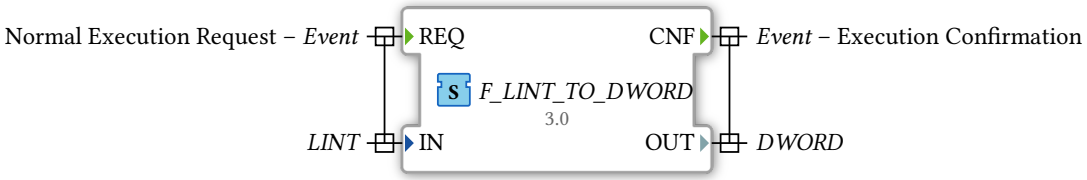
Der `F_LINT_TO_DINT` Funktionsblock bietet eine einfache und effiziente Möglichkeit, 64-Bit Integer-Werte in 32-Bit Integer-Werte umzuwandeln. Seine Einfachheit macht ihn zuverlässig und leicht integrierbar, jedoch sollten Anwender potenzielle Datenverluste bei der Konvertierung großer Werte berücksichtigen.

F_LINT_TO_DWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LINT_TO_DWORD` dient der Konvertierung eines LINT-Wertes (64-Bit Ganzzahl) in einen DWORD-Wert (32-Bit vorzeichenlose Ganzzahl). Dieser Baustein ist besonders nützlich in Szenarien, wo eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung der Konvertierung. Dieser Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingang vom Typ `LINT`, der den zu konvertierenden Wert enthält.

Daten-Ausgänge

- **OUT**: Der Ausgang vom Typ `DWORD`, der das Ergebnis der Konvertierung enthält.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem er den `LINT`-Wert am Eingang `IN` mittels der Funktion `LINT_TO_DWORD` in einen `DWORD`-Wert umwandelt. Das Ergebnis wird am Ausgang `OUT` ausgegeben. Die Konvertierung wird durch das Ereignis `REQ` gestartet und durch `CNF` bestätigt.

Technische Besonderheiten

- Die Konvertierung schneidet die höherwertigen 32 Bits des `LINT`-Wertes ab, da `DWORD` nur 32 Bits speichern kann.
- Es findet keine Überprüfung auf Überlauf statt, was bei großen `LINT`-Werten zu Datenverlust führen kann.

Zustandsübersicht

1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Processing**: Führt die Konvertierung durch.
3. **Complete**: Sendet das `CNF`-Ereignis und gibt das Ergebnis aus.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo unterschiedliche Datentypen verarbeitet werden müssen.
- Datenreduktion, wenn nur die unteren 32 Bits eines `LINT`-Wertes relevant sind.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_INT_TO_DINT` oder `F_DINT_TO_LINT` führen ähnliche Typumwandlungen durch, jedoch zwischen anderen Datentypen.
- `F_LINT_TO_DWORD` ist spezifisch für die Konvertierung von `LINT` zu `DWORD`.

Fazit

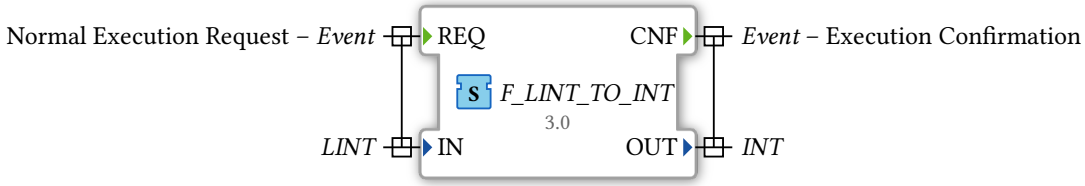
Der `F_LINT_TO_DWORD` Funktionsblock bietet eine einfache und effiziente Möglichkeit, `LINT`-Werte in `DWORD`-Werte umzuwandeln. Seine Verwendung ist in Szenarien mit Typumwandlungsbedarf zwischen diesen spezifischen Datentypen empfehlenswert, wobei auf mögliche Datenverluste bei großen Werten geachtet werden sollte.

F_LINT_TO_INT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LINT_TO_INT` dient zur Konvertierung eines 64-Bit Integer-Werts (LINT) in einen 16-Bit Integer-Wert (INT). Diese Konvertierung ist nützlich, wenn Daten zwischen Systemen mit unterschiedlicher Wortbreite ausgetauscht oder verarbeitet werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung der Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert die erfolgreiche Beendigung der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (LINT): Der 64-Bit Integer-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (INT): Der resultierende 16-Bit Integer-Wert nach der Konvertierung.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von einem LINT- zu einem INT-Wert durch, sobald das Ereignis `REQ` empfangen wird. Die Konvertierung erfolgt durch die Funktion `LINT_TO_INT(IN)`, die den Wert von `IN` in einen INT-Wert umwandelt und das Ergebnis an `OUT` ausgibt. Anschließend wird das Ereignis `CNF` ausgelöst, um die erfolgreiche Konvertierung zu signalisieren.

Technische Besonderheiten

- Die Konvertierung kann zu einem Datenverlust führen, wenn der ursprüngliche LINT-Wert außerhalb des darstellbaren Bereichs eines INT-Werts (-32.768 bis 32.767) liegt.
- Der Funktionsblock ist einfach aufgebaut und enthält nur einen Algorithmus für die Konvertierung.

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf das Ereignis `REQ`.
2. **Konvertierungszustand**: Führt die Konvertierung durch und gibt das Ergebnis aus.
3. **Bestätigungszustand**: Sendet das Ereignis `CNF` und kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- Datenkommunikation zwischen Systemen mit unterschiedlicher Wortbreite.
- Reduzierung der Speicherbelegung, wenn die volle 64-Bit-Präzision nicht benötigt wird.
- Integration in Steuerungssysteme, die nur 16-Bit-Integer verarbeiten können.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_LINT_TO_DINT**: Konvertiert LINT zu DINT (32-Bit Integer).
- **F_LINT_TO_UDINT**: Konvertiert LINT zu UDINT (32-Bit vorzeichenloser Integer).
- **F_LINT_TO_ULINT**: Konvertiert LINT zu ULINT (64-Bit vorzeichenloser Integer).

Fazit

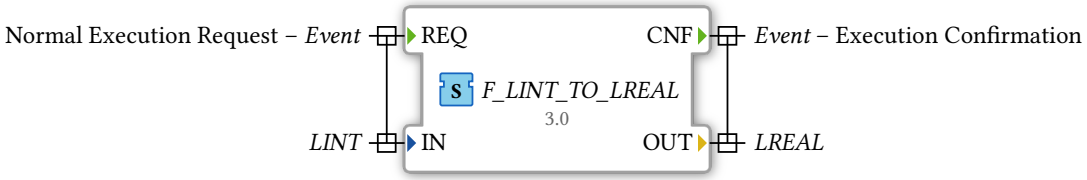
Der `F_LINT_TO_INT` Funktionsblock bietet eine einfache und effiziente Möglichkeit, 64-Bit Integer-Werte in 16-Bit Integer-Werte umzuwandeln. Er ist besonders nützlich in Szenarien, wo die Reduzierung der Datenbreite notwendig ist, jedoch sollte die Möglichkeit eines Datenverlusts bei der Konvertierung beachtet werden.

F_LINT_TO_LREAL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LINT_TO_LREAL` dient der Konvertierung eines ganzzahligen Wertes des Typs `LINT` (64-Bit Ganzzahl) in einen Fließkommawert des Typs `LREAL` (64-Bit Fließkomma). Diese Konvertierung ist insbesondere dann notwendig, wenn ganzzahlige Werte in Berechnungen mit Fließkommazahlen verwendet werden sollen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `LINT`, der in einen `LREAL`-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der konvertierte Ausgangswert vom Typ `LREAL`.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Auslösung des Ereignisses `REQ` wird der Algorithmus ausgeführt, der den Wert von `IN` vom Typ `LINT` in den Typ `LREAL` konvertiert und das Ergebnis an `OUT` ausgibt. Anschließend wird das Ereignis `CNF` ausgelöst, um den Abschluss der Konvertierung zu signalisieren.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne zusätzliche Parameter.
- Der Funktionsblock ist für die Verwendung in Echtzeitsystemen geeignet, da er deterministisch arbeitet.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung von `REQ`.

Anwendungsszenarien

- Umwandlung von großen Ganzzahlen in Fließkommazahlen für mathematische Berechnungen.
- Integration in Steuerungssysteme, die gemischte Datentypen verarbeiten müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen wie `F_INT_TO_REAL` oder `F_DINT_TO_REAL` handelt `F_LINT_TO_LREAL` mit größeren Zahlenbereichen (64-Bit).

Fazit

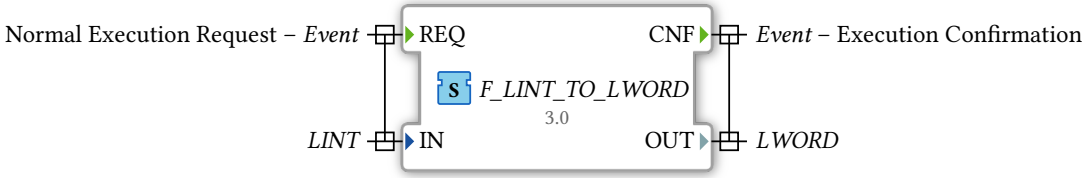
Der `F_LINT_TO_LREAL`-Funktionsblock bietet eine effiziente und zuverlässige Methode zur Konvertierung von 64-Bit Ganzzahlen in 64-Bit Fließkommazahlen. Seine einfache Schnittstelle und deterministische Funktionsweise machen ihn ideal für den Einsatz in industriellen Steuerungssystemen.

F_LINT_TO_LWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LINT_TO_LWORD` dient der Konvertierung eines LINT-Datentyps (64-Bit Ganzzahl mit Vorzeichen) in einen LWORD-Datentyp (64-Bit Bitstring). Dieser Baustein ist besonders nützlich in Szenarien, wo numerische Werte in ihre binäre Darstellung umgewandelt werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Ereigniseingang ausgelöst, liest der Funktionsblock den Wert am Daten-Eingang `IN` und führt die Konvertierung durch.

Ereignis-Ausgänge

- **CNF**: Signalisiert den erfolgreichen Abschluss der Konvertierung. Dieses Ereignis wird zusammen mit dem konvertierten Wert am Daten-Ausgang `OUT` ausgegeben.

Daten-Eingänge

- **IN**: Erwartet einen Wert vom Typ `LINT` (64-Bit Ganzahl mit Vorzeichen), der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Gibt den konvertierten Wert vom Typ `LWORD` (64-Bit Bitstring) aus.

Adapter

Dieser Funktionsblock verfügt über keine Adapter-Schnittstellen.

Funktionsweise

Bei Auslösung des `REQ`-Ereignisses wird der Wert am Eingang `IN` gelesen und mittels der Funktion `LINT_TO_LWORD` in einen `LWORD`-Wert umgewandelt. Das Ergebnis wird am Ausgang `OUT` bereitgestellt und gleichzeitig das `CNF`-Ereignis ausgelöst.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne zusätzliche Verzögerung.
- Der Funktionsblock ist deterministisch und hat keine internen Zustände.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge oder -maschinen. Die Konvertierung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Umwandlung von numerischen Werten in ihre binäre Repräsentation für bitweise Operationen.
- Verwendung in Protokollimplementierungen, wo numerische Werte als Bitstrings übertragen werden müssen.
- Integration in Systeme, die mit hardwarenahen Bitmanipulationen arbeiten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken ist `F_LINT_TO_LWORD` spezialisiert auf die Umwandlung von `LINT` zu `LWORD` und bietet daher eine optimierte und typsichere Lösung.
- Ähnliche Blöcke wie `INT_TO_WORD` oder `DINT_TO_DWORD` arbeiten mit kleineren Datentypen.

Fazit

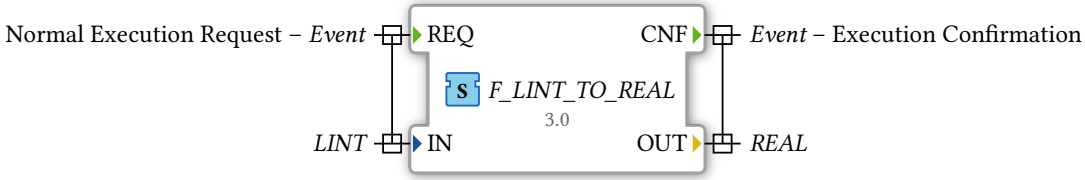
Der `F_LINT_TO_LWORD` Funktionsblock bietet eine effiziente und zuverlässige Methode zur Konvertierung von 64-Bit Ganzzahlen in 64-Bit Bitstrings. Seine Einfachheit und Determinismus machen ihn ideal für Echtzeitanwendungen, wo schnelle und präzise Typumwandlungen erforderlich sind.

F_LINT_TO_REAL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LINT_TO_REAL` konvertiert einen ganzzahligen Wert des Typs `LINT` (64-Bit Integer) in einen Fließkommawert des Typs `REAL`. Diese Konvertierung ist nützlich, wenn Daten aus einem ganzzahligen Format in ein Fließkommaformat umgewandelt werden müssen, beispielsweise für weitere Berechnungen oder Anzeigen.



Schnittstellenstruktur

Ereignis-Eingänge
- **REQ**: Startet die Konvertierung. Bei Auslösung dieses Ereignisses wird der Wert am Daten-Eingang `IN` gelesen und konvertiert.

Ereignis-Ausgänge
- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieses Ereignis wird zusammen mit dem konvertierten Wert am Daten-Ausgang `OUT` ausgegeben.

Daten-Eingänge
- **IN**: Der Eingang vom Typ `LINT`, der den zu konvertierenden ganzzahligen Wert bereitstellt.

Daten-Ausgänge
- **OUT**: Der Ausgang vom Typ `REAL`, der das Ergebnis der Konvertierung liefert.

Adapter
- Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise
Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` empfangen wird. Der Algorithmus liest den Wert von `IN`, konvertiert ihn mit der Funktion `LINT_TO_REAL` und schreibt das Ergebnis in `OUT`. Anschließend wird das Ereignis `CNF` ausgelöst, um den erfolgreichen Abschluss der Operation zu signalisieren.

Technische Besonderheiten
- Die Konvertierung erfolgt ohne Rundung oder Skalierung, da es sich um eine direkte Typumwandlung handelt.
- Der Funktionsblock ist für die Verwendung in Echtzeitsystemen geeignet, da er keine Verzögerungen oder komplexen Berechnungen beinhaltet.

Zustandsübersicht
Der Funktionsblock besitzt keine internen Zustände. Die Operation wird bei jedem `REQ`-Ereignis neu ausgeführt.

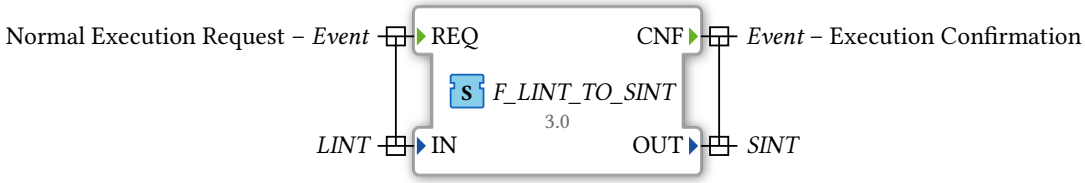
Anwendungsszenarien
- Umwandlung von Sensorwerten, die als `LINT` vorliegen, in `REAL` für weitere Verarbeitung.
- Integration in Systeme, die Fließkommaoperationen erfordern, aber Eingangsdaten als Ganzzahlen erhalten.

⚖️ Vergleich mit ähnlichen Bausteinen
- Ähnliche Funktionsblöcke wie `F_INT_TO_REAL` oder `F_DINT_TO_REAL` führen ähnliche Konvertierungen durch, jedoch mit kleineren Integer-Typen (`INT` oder `DINT`).
- `F_LINT_TO_REAL` ist speziell für 64-Bit Integer (`LINT`) optimiert.

Fazit
Der `F_LINT_TO_REAL` Funktionsblock bietet eine einfache und effiziente Möglichkeit, 64-Bit Integer-Werte in Fließkommazahlen umzuwandeln. Durch seine klare Schnittstelle und direkte Funktionsweise ist er ideal für Anwendungen, die eine Typkonvertierung ohne zusätzliche Verarbeitung erfordern.

F_LINT_TO_SINT
• – ◦ • – ◦ • – ◦ •

Einleitung
Der Funktionsblock `F_LINT_TO_SINT` dient der Konvertierung eines 64-Bit-Ganzzahlwerts (`LINT`) in einen 8-Bit-Ganzzahlwert (`SINT`). Diese Konvertierung ist nützlich, wenn Daten zwischen Systemen mit unterschiedlicher Wortbreite ausgetauscht oder verarbeitet werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN** (`LINT`): Der 64-Bit-Ganzzahlwert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (`SINT`): Der resultierende 8-Bit-Ganzzahlwert nach der Konvertierung.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Bei Auslösung des Ereignisses `REQ` wird der Algorithmus ausgeführt, der den Wert von `IN` vom Typ `LINT` in einen `SINT`-Wert konvertiert und das Ergebnis an `OUT` ausgibt. Anschließend wird das Ereignis `CNF` ausgelöst, um den Abschluss der Konvertierung zu signalisieren.

Technische Besonderheiten

- Die Konvertierung kann zu einem Datenverlust führen, wenn der ursprüngliche `LINT`-Wert außerhalb des darstellbaren Bereichs eines `SINT` (-128 bis 127) liegt.
- Der Funktionsblock ist einfach und deterministisch, ohne interne Zustände oder komplexe Logik.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Ausführung erfolgt ausschließlich als Reaktion auf das `REQ`-Ereignis.

Anwendungsszenarien

- Datenreduktion bei der Übertragung zwischen Systemen mit unterschiedlicher Wortbreite.
- Verarbeitung von Sensordaten, bei denen nur die unteren 8 Bit relevant sind.
- Kompatibilität mit älteren Systemen, die nur 8-Bit-Daten verarbeiten können.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken ist `F_LINT_TO_SINT` spezialisiert und optimiert für die Konvertierung von `LINT` zu `SINT`.
- Andere Konvertierungsblöcke könnten zusätzliche Funktionen wie Überlaufbehandlung oder Skalierung bieten, was hier nicht der Fall ist.

Fazit

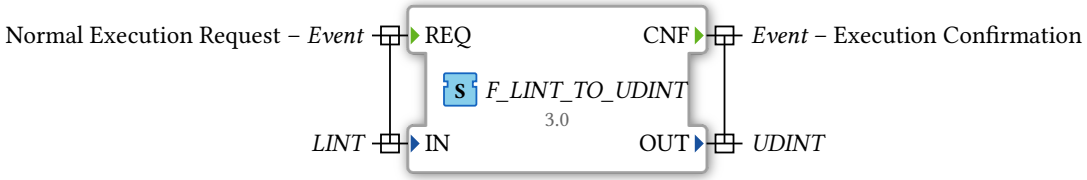
Der `F_LINT_TO_SINT`-Funktionsblock ist ein einfaches und effizientes Werkzeug zur Konvertierung von 64-Bit- zu 8-Bit-Ganzzahlen. Seine Einfachheit macht ihn robust und leicht verständlich, allerdings ohne erweiterte Features wie Überlaufkontrolle. Ideal für Anwendungen, bei denen eine direkte und schnelle Konvertierung benötigt wird.

F_LINT_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LINT_TO_UDINT` dient zur Konvertierung eines 64-Bit vorzeichenbehafteten Integer-Werts (`LINT`) in einen 32-Bit vorzeichenlosen Integer-Wert (`UDINT`). Der Block ist Teil der `iec61131::conversion`-Bibliothek und eignet sich für Anwendungen, bei denen eine Typumwandlung erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (`LINT`): Der 64-Bit vorzeichenbehaftete Integer-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (`UDINT`): Der resultierende 32-Bit vorzeichenlose Integer-Wert nach der Konvertierung.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Bei Auslösung des Ereignisses `REQ` wird der Algorithmus ausgeführt, der den Wert von `IN` von `LINT` in `UDINT` konvertiert und das Ergebnis an `OUT` ausgibt. Anschließend wird das Ereignis `CNF` ausgelöst, um den Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Der Block nutzt die eingebaute Funktion `LINT_TO_UDINT` für die Konvertierung.
- Es handelt sich um einen einfachen Funktionsblock ohne Zustandsautomaten oder komplexe Logik.

Zustandsübersicht

Der Block hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung von `REQ`.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo 64-Bit-Werte in 32-Bit-Werte umgewandelt werden müssen.
- Verwendung in Datenverarbeitungsketten, wo unterschiedliche Datentypen kompatibel gemacht werden müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Blöcke wie `F_DINT_TO_UDINT` oder `F_INT_TO_UINT` bieten Konvertierungen zwischen anderen Integer-Typen, jedoch nicht für `LINT` zu `UDINT`.
- Im Gegensatz zu generischen Konvertierungsblöcken ist `F_LINT_TO_UDINT` spezialisiert und optimiert für diese spezifische Umwandlung.

Fazit

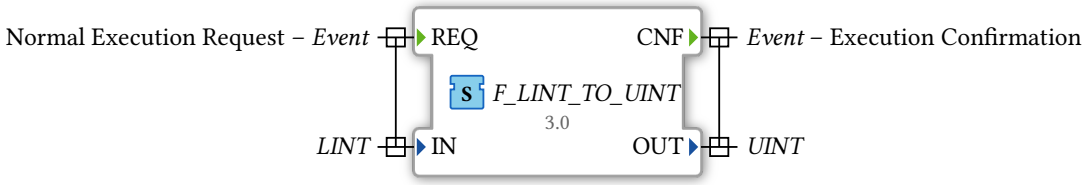
Der `F_LINT_TO_UDINT`-Block ist ein effizientes Werkzeug für die Konvertierung von 64-Bit vorzeichenbehafteten zu 32-Bit vorzeichenlosen Integer-Werten. Seine Einfachheit und direkte Funktionsweise machen ihn zu einer zuverlässigen Komponente in Steuerungsanwendungen.

F_LINT_TO_UINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LINT_TO_UINT` dient zur Konvertierung eines 64-Bit-Ganzzahlwerts (`LINT`) in einen 16-Bit-unsigned-Ganzzahlwert (`UINT`). Dieser Baustein ist besonders nützlich in Szenarien, wo eine Typumwandlung zwischen unterschiedlich großen numerischen Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingang erwartet einen Wert vom Typ `LINT` (64-Bit-Ganzzahl), der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgang liefert den konvertierten Wert vom Typ `UINT` (16-Bit-unsigned-Ganzzahl).

Adapter

Es sind keine Adapter definiert.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `LINT` zu `UINT` durch, sobald das Ereignis `REQ` ausgelöst wird. Der Algorithmus `REQ` nutzt die Funktion `LINT_TO_UINT`, um den Eingabewert umzuwandeln. Das Ergebnis wird über den Ausgang `OUT` ausgegeben, und das Ereignis `CNF` signalisiert die erfolgreiche Ausführung.

Technische Besonderheiten

- **Typumwandlung**: Es ist wichtig zu beachten, dass bei der Konvertierung von einem größeren zu einem kleineren Datentyp Datenverlust auftreten kann, wenn der ursprüngliche Wert außerhalb des darstellbaren Bereichs von `UINT` liegt.
- **Effizienz**: Die Konvertierung erfolgt direkt und ohne zusätzliche Verzögerung.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung wird bei jedem `REQ`-Ereignis neu ausgeführt.

Anwendungsszenarien

- **Datenverarbeitung**: In Steuerungssystemen, wo unterschiedliche Datentypen harmonisiert werden müssen.
- **Schnittstellenkompatibilität**: Bei der Kommunikation zwischen Systemen, die unterschiedliche Datentypen verwenden.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_LINT_TO_INT**: Konvertiert `LINT` zu `INT`, behält aber das Vorzeichen bei.
- **F_LINT_TO_ULINT**: Konvertiert `LINT` zu `ULINT`, ohne die Bitbreite zu reduzieren.

Fazit

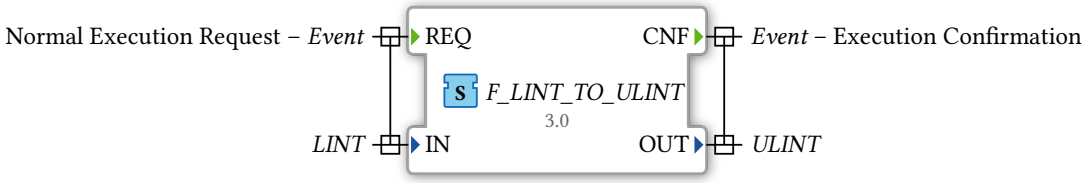
Der `F_LINT_TO_UINT`-Funktionsblock ist ein einfaches, aber effektives Werkzeug zur Typumwandlung in IEC 61499-basierten Systemen. Seine klare Schnittstelle und direkte Funktionsweise machen ihn zu einer zuverlässigen Lösung für Konvertierungsaufgaben. Beachten Sie jedoch mögliche Datenverluste bei der Umwandlung.

F_LINT_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LINT_TO_ULINT` dient der Konvertierung eines vorzeichenbehafteten 64-Bit-Integer-Werts (`LINT`) in einen vorzeichenlosen 64-Bit-Integer-Wert (`ULINT`). Dieser Baustein ist besonders nützlich in Szenarien, wo eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Dieser Eingang löst die Konvertierung aus. Er ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Dieser Ausgang signalisiert den Abschluss der Konvertierung. Er ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (`LINT`): Der vorzeichenbehaftete 64-Bit-Integer-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (`ULINT`): Der vorzeichenlose 64-Bit-Integer-Wert, der als Ergebnis der Konvertierung ausgegeben wird.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `LINT` zu `ULINT` durch, sobald das Ereignis `REQ` ausgelöst wird. Die Konvertierung erfolgt mittels der Funktion `LINT_TO_ULINT(IN)`, die den Wert von `IN` in den entsprechenden vorzeichenlosen Typ umwandelt. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst, und der konvertierte Wert steht am Ausgang `OUT` zur Verfügung.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher Baustein ohne interne Zustandsverwaltung.
- Die Konvertierung erfolgt direkt und ohne Verzögerung.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge oder komplexe Zustandslogik.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo vorzeichenbehaftete und vorzeichenlose Werte verarbeitet werden müssen.
- Datenverarbeitung in industriellen Automatisierungssystemen, wo unterschiedliche Datentypen kompatibel gemacht werden müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Bausteine könnten `F_INT_TO_UINT` oder `F_DINT_TO_UDINT` sein, die jedoch für kleinere Integer-Typen konzipiert sind.
- Im Gegensatz zu generischen Konvertierungsbausteinen ist `F_LINT_TO_ULINT` spezialisiert auf die 64-Bit-Varianten der Datentypen.

Fazit

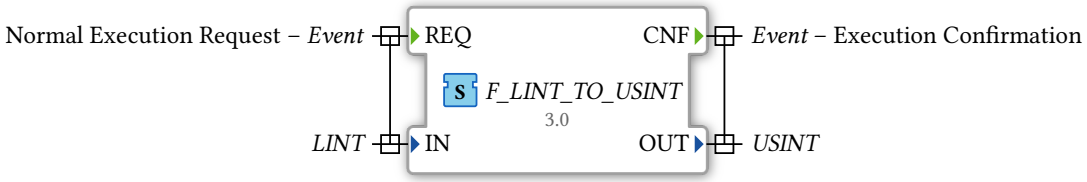
Der `F_LINT_TO_ULINT` Funktionsblock bietet eine einfache und effiziente Möglichkeit, vorzeichenbehaftete 64-Bit-Integer-Werte in vorzeichenlose umzuwandeln. Seine klare Schnittstellenstruktur und direkte Funktionsweise machen ihn zu einem zuverlässigen Werkzeug in der industriellen Automatisierung.

F_LINT_TO_USINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LINT_TO_USINT` dient der Konvertierung eines 64-Bit Ganzzahlwertes (`LINT`) in einen 8-Bit vorzeichenlosen Ganzzahlwert (`USINT`). Dieser Baustein ist Teil der IEC 61131-Konvertierungsbibliothek und wird typischerweise in Automatisierungsanwendungen eingesetzt, wo Datentypen unterschiedlicher Größe verarbeitet werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Dieser Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (`LINT`): Der 64-Bit Ganzzahlwert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (`USINT`): Der resultierende 8-Bit vorzeichenlose Ganzahlwert nach der Konvertierung.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, wenn das Ereignis `REQ` ausgelöst wird. Dabei wird der Wert von `IN` vom Typ `LINT` in den Typ `USINT` umgewandelt und das Ergebnis an `OUT` ausgegeben. Die Konvertierung wird durch das Ereignis `CNF` bestätigt.

Technische Besonderheiten

- Die Konvertierung schneidet höherwertige Bits ab, falls der `LINT`-Wert außerhalb des darstellbaren Bereichs von `USINT` (0 bis 255) liegt.
- Der Block ist deterministisch und hat keine internen Zustände.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Ausführung erfolgt unmittelbar nach dem Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Konvertierung von Sensorwerten oder Zählern, die als `LINT` vorliegen, für die Verarbeitung in Systemen, die nur `USINT` unterstützen.
- Verwendung in Steuerungssystemen, wo Speicherplatz optimiert werden muss.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Konvertierungsblöcke wie `F_INT_TO_USINT` oder `F_DINT_TO_USINT` arbeiten mit kleineren Eingangsdatentypen (`INT` bzw. `DINT`).
- Im Gegensatz zu generischen Konvertierungsblöcken ist `F_LINT_TO_USINT` spezialisiert auf die Konvertierung von `LINT` zu `USINT`.

Fazit

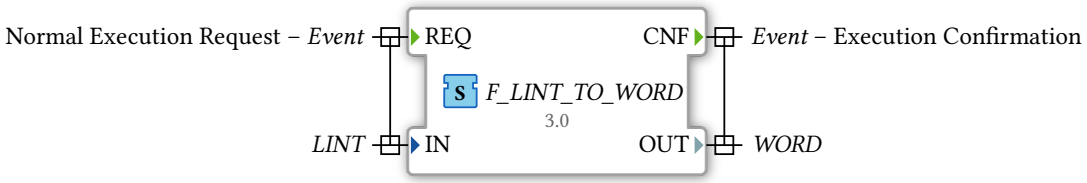
Der `F_LINT_TO_USINT` Funktionsblock ist ein spezialisiertes und effizientes Werkzeug für die Konvertierung von 64-Bit Ganzzahlen in 8-Bit vorzeichenlose Werte. Seine einfache und deterministische Funktionsweise macht ihn zuverlässig in Automatisierungsanwendungen.

F_LINT_TO_WORD

• − ◦ • − ◦ • − ◦ •

Einleitung

Der Funktionsblock `F_LINT_TO_WORD` dient der Konvertierung eines LINT-Datentyps (64-Bit Ganzzahl) in einen WORD-Datentyp (16-Bit Ganzahl). Diese Konvertierung ist nützlich, wenn Daten zwischen Systemen mit unterschiedlicher Wortbreite ausgetauscht werden müssen oder wenn eine explizite Typumwandlung erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ` (Normal Execution Request): Startet die Ausführung der Konvertierung. Wird mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- `CNF` (Execution Confirmation): Signalisiert den erfolgreichen Abschluss der Konvertierung. Wird mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- `IN` (LINT): Der Eingangswert, der von LINT in WORD konvertiert werden soll.

Daten-Ausgänge

- `OUT` (WORD): Der konvertierte Ausgangswert im WORD-Format.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung von LINT zu WORD durch, wenn das Ereignis `REQ` ausgelöst wird. Die Konvertierung erfolgt mittels der eingebauten Funktion `LINT_TO_WORD`, die den Wert des Eingangs `IN` nimmt und ihn in den WORD-Datentyp umwandelt. Das Ergebnis wird am Ausgang `OUT` ausgegeben, und das Ereignis `CNF` signalisiert die erfolgreiche Ausführung.

Technische Besonderheiten

- Die Konvertierung schneidet die höherwertigen Bits des LINT-Werts ab, da WORD nur 16 Bit speichern kann.
- Es gibt keine Überprüfung auf Überlauf oder Unterlauf während der Konvertierung.

Zustandsübersicht

Der Funktionsblock ist zustandslos und führt die Konvertierung bei jedem `REQ`-Ereignis neu aus.

Anwendungsszenarien

- Datenkommunikation zwischen Systemen mit unterschiedlicher Wortbreite.
- Explizite Typumwandlung in Steuerungsanwendungen, wo eine 64-Bit-Zahl auf eine 16-Bit-Zahl reduziert werden muss.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_DINT_TO_WORD` oder `F_INT_TO_WORD` führen ähnliche Konvertierungen durch, jedoch mit anderen Ausgangsdatentypen.
- `F_LINT_TO_WORD` ist spezifisch für die Konvertierung von 64-Bit zu 16-Bit und bietet keine zusätzliche Funktionalität wie Überlaufprüfung.

Fazit

Der `F_LINT_TO_WORD` Funktionsblock ist ein einfaches, aber effektives Werkzeug zur Konvertierung von LINT- zu WORD-Daten. Er ist besonders nützlich in Szenarien, wo eine Reduzierung der Datenbreite erforderlich ist, ohne dass zusätzliche Überprüfungen notwendig sind.

LINT Konvertierungen

Hier finden Sie Funktionen zur Konvertierung des Datentyps LINT.

🛠️ Zugehörige Übungen

- Uebung_072c

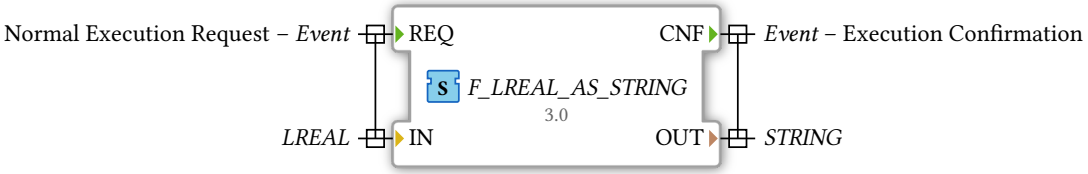
LREAL

F_LREAL_AS_STRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LREAL_AS_STRING` dient zur Konvertierung eines LREAL-Wertes (64-Bit Gleitkommazahl) in einen STRING. Diese Funktionalität ist besonders nützlich, wenn numerische Werte für die Anzeige, Protokollierung oder Kommunikation als Zeichenketten benötigt werden.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- `CNF`: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- `IN` (LREAL): Der Eingangswert, der in einen STRING konvertiert werden soll.

Daten-Ausgänge

- `OUT` (STRING): Der Ergebnis-STRING nach der Konvertierung.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Auslösung des `REQ`-Ereignisses wird der Algorithmus ausgeführt, der den LREAL-Wert `IN` in einen STRING konvertiert und das Ergebnis über `OUT` ausgibt. Das `CNF`-Ereignis signalisiert die erfolgreiche Beendigung der Konvertierung.

Technische Besonderheiten

- Die Konvertierung erfolgt mittels der Funktion `LREAL_AS_STRING`.
- Der Funktionsblock ist einfach und hat keine internen Zustände oder komplexe Logik.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge oder -maschinen.

Anwendungsszenarien

- Anzeige von Gleitkommazahlen in Benutzeroberflächen.
- Protokollierung von Messwerten in Textdateien oder Datenbanken.
- Vorbereitung von Daten für die Kommunikation über Textprotokolle.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu `F_STRING_TO_LREAL` führt dieser Baustein die umgekehrte Konvertierung durch.
- Im Vergleich zu generischen Konvertierungsbausteinen ist dieser spezialisiert auf LREAL-zu-STRING-Konvertierungen.

Fazit

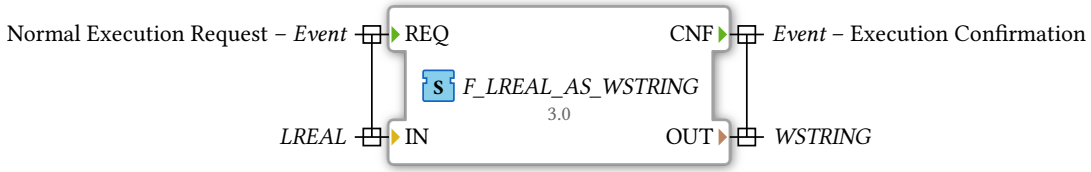
Der `F_LREAL_AS_STRING` Funktionsblock bietet eine einfache und effiziente Möglichkeit, Gleitkommazahlen in Zeichenketten umzuwandeln. Seine klare Schnittstelle und einfache Funktionsweise machen ihn zu einem nützlichen Baustein in vielen Automatisierungsanwendungen.

F_LREAL_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LREAL_AS_WSTRING` dient der Konvertierung eines LREAL-Wertes (64-Bit Fließkommazahl) in einen WSTRING (Wide-String). Dies ist besonders nützlich, wenn numerische Werte für die Anzeige oder Weiterverarbeitung als Text benötigt werden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung des Eingabewertes. Der Eingabewert `IN` wird mit diesem Ereignis verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgabewert `OUT` wird mit diesem Ereignis ausgegeben.

Daten-Eingänge

- **IN** (LREAL): Der Eingabewert, der in einen WSTRING konvertiert werden soll.

Daten-Ausgänge

- **OUT** (WSTRING): Der Ergebnisstring nach der Konvertierung.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Auslösung des `REQ`-Ereignisses wird der Eingabewert `IN` vom Typ LREAL mit der Funktion `LREAL_AS_WSTRING` in einen WSTRING konvertiert und am Ausgang `OUT` bereitgestellt. Das `CNF`-Ereignis signalisiert die erfolgreiche Konvertierung.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne zusätzliche Parameter.
- Der Funktionsblock ist für die Verwendung in Echtzeitsystemen geeignet.

Zustandsübersicht

1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Processing**: Konvertiert den Eingabewert.
3. **Done**: Sendet das `CNF`-Ereignis mit dem konvertierten Wert.

Anwendungsszenarien

- Anzeige von Fließkommawerten auf Benutzeroberflächen.
- Protokollierung von Messwerten in Textdateien.
- Vorbereitung von Daten für die Kommunikation mit Systemen, die String-Daten erwarten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu `F_REAL_AS_WSTRING` verarbeitet dieser Baustein 64-Bit Fließkommazahlen statt 32-Bit.
- Ähnlich zu `F_DINT_AS_WSTRING`, aber für Fließkommazahlen statt Ganzzahlen.

Fazit

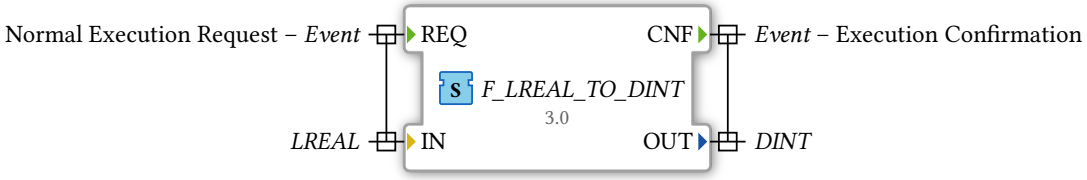
Der `F_LREAL_AS_WSTRING` Funktionsblock bietet eine einfache und effiziente Möglichkeit, LREAL-Werte in WSTRINGs zu konvertieren. Seine klare Schnittstelle und direkte Funktionsweise machen ihn zu einem nützlichen Werkzeug in verschiedenen Anwendungsfällen.

F_LREAL_TO_DINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LREAL_TO_DINT` dient der Konvertierung eines LREAL-Wertes (64-Bit Fließkommazahl) in einen DINT-Wert (32-Bit Ganzzahl). Der Block ist Teil des `iec61131::conversion` Pakets und implementiert eine einfache Typumwandlung.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung der Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (LREAL): Der Eingangswert, der von LREAL in DINT konvertiert werden soll.

Daten-Ausgänge

- **OUT** (DINT): Das Ergebnis der Konvertierung als DINT-Wert.

Adapter

Der Funktionsblock verfügt über keine Adapter.

Funktionsweise

Bei Auslösung des `REQ`-Ereignisses wird der Algorithmus ausgeführt, der den `IN`-Wert von LREAL in DINT konvertiert und das Ergebnis an `OUT` ausgibt. Anschließend wird das `CNF`-Ereignis ausgelöst, um den erfolgreichen Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Die Konvertierung erfolgt durch die eingebaute Funktion `LREAL_TO_DINT`.
- Es handelt sich um einen einfachen Funktionsblock ohne Zustandsverwaltung oder komplexe Logik.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Ausführung ist rein ereignisgesteuert und erfolgt bei jedem `REQ`-Ereignis.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo Fließkommazahlen in Ganzzahlen umgewandelt werden müssen.
- Einsatz in Berechnungen, wo eine Ganzzahldarstellung erforderlich ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsblöcken wie `F_REAL_TO_INT` oder `F_LREAL_TO_LINT` bietet dieser Block eine spezifische Konvertierung von LREAL zu DINT.
- Ähnliche Blöcke können unterschiedliche Quell- oder Zieltypen haben, wie z.B. `F_DINT_TO_LREAL` für die umgekehrte Konvertierung.

Fazit

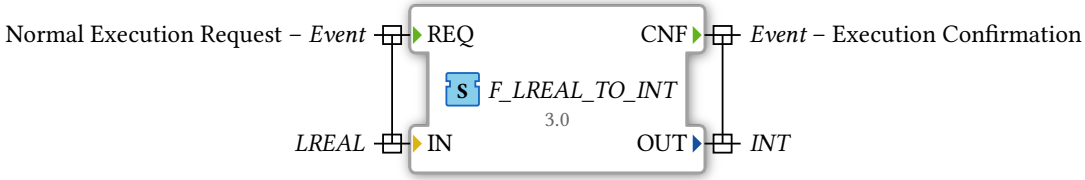
Der `F_LREAL_TO_DINT` Funktionsblock ist ein einfaches und effizientes Werkzeug zur Typumwandlung in IEC 61499-basierten Steuerungssystemen. Seine klare Schnittstelle und direkte Funktionsweise machen ihn ideal für Anwendungen, die eine schnelle Konvertierung von Fließkommazahlen in Ganzzahlen erfordern.

F_LREAL_TO_INT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LREAL_TO_INT` dient zur Konvertierung eines LREAL-Wertes (64-Bit Fließkommazahl) in einen INT-Wert (16-Bit Ganzzahl). Diese Konvertierung erfolgt durch Abrundung des LREAL-Wertes auf den nächstkleineren ganzzahligen Wert.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Startet die Konvertierung des Eingabewertes. Der Eingabewert wird mit diesem Ereignis übergeben.

Ereignis-Ausgänge

- `CNF`: Signalisiert den Abschluss der Konvertierung und gibt den konvertierten Wert aus.

Daten-Eingänge

- `IN` (LREAL): Der Eingabewert, der in einen INT-Wert konvertiert werden soll.

Daten-Ausgänge

- `OUT` (INT): Der konvertierte Ganzzahlwert.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Empfang des `REQ`-Ereignisses wird der Algorithmus ausgeführt, der den `IN`-Wert von LREAL zu INT konvertiert. Die Konvertierung erfolgt durch die Funktion `LREAL_TO_INT`, die den LREAL-Wert auf den nächstkleineren ganzzahligen Wert abrundet. Das Ergebnis wird an `OUT` ausgegeben und das `CNF`-Ereignis wird ausgelöst.

Technische Besonderheiten

- Die Konvertierung schneidet Nachkommastellen ab (keine Rundung, sondern Abrundung).
- Bei Werten außerhalb des INT-Bereichs kann es zu Überlaufproblemen kommen.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Konvertierung von Sensorwerten (z.B. Temperaturmessungen) für die Verarbeitung in Ganzzahl-basierten Systemen.
- Vorbereitung von Fließkommawerten für die Ausgabe auf Ganzzahl-Displays oder -Schnittstellen.

⚖️ Vergleich mit ähnlichen Bausteinen

- `F_REAL_TO_INT`: Konvertiert REAL (32-Bit Fließkommazahl) zu INT.
- `F_LREAL_TO_DINT`: Konvertiert LREAL zu DINT (32-Bit Ganzzahl) für einen größeren Wertebereich.

Fazit

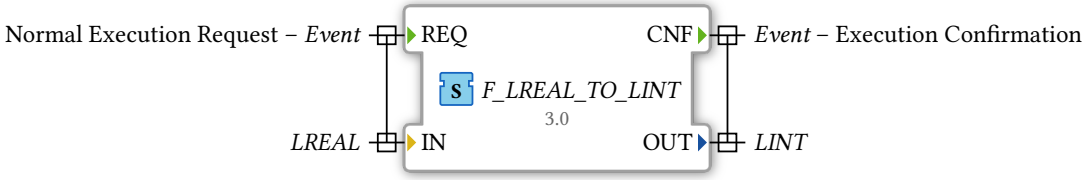
Der `F_LREAL_TO_INT`-Funktionsblock ist ein einfaches, aber effektives Werkzeug zur Konvertierung von Fließkommazahlen in Ganzzahlen. Er eignet sich besonders für Anwendungen, bei denen eine schnelle und direkte Konvertierung ohne zusätzliche Logik erforderlich ist. Bei Werten, die den INT-Bereich überschreiten, sollten jedoch alternative Bausteine wie `F_LREAL_TO_DINT` in Betracht gezogen werden.

F_LREAL_TO_LINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LREAL_TO_LINT` dient der Konvertierung eines LREAL-Wertes (64-Bit Gleitkommazahl) in einen LINT-Wert (64-Bit Ganzzahl). Diese Konvertierung ist insbesondere in Szenarien erforderlich, in denen Gleitkommazahlen in Ganzzahlen umgewandelt werden müssen, beispielsweise bei der Verarbeitung von Sensorwerten oder bei der Steuerung von Aktoren.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Dieser Eingang löst die Konvertierung aus.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Dieser Ausgang signalisiert den Abschluss der Konvertierung.

Daten-Eingänge

- **IN**: LREAL (64-Bit Gleitkommazahl). Der Eingabewert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: LINT (64-Bit Ganzzahl). Der konvertierte Ausgabewert.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von LREAL zu LINT durch, sobald das Ereignis `REQ` empfangen wird. Die Konvertierung erfolgt mithilfe der Funktion `LREAL_TO_LINT`, die den Eingabewert `IN` in den Ausgabewert `OUT` umwandelt. Nach Abschluss der Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Die Konvertierung schneidet den Nachkommateil der Gleitkommazahl ab (Trunkierung).
- Es erfolgt keine Rundung des Wertes.
- Bei Werten außerhalb des darstellbaren Bereichs von LINT kann es zu Überlaufproblemen kommen.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Umwandlung von Sensorwerten (z.B. Temperaturmessungen) in Ganzzahlen für weitere Verarbeitung.
- Steuerung von Aktoren, die Ganzzahlwerte erwarten.
- Datenvorverarbeitung in Steuerungssystemen.

⚖️ Vergleich mit ähnlichen Bausteinen

- `F_REAL_TO_INT`: Konvertiert REAL (32-Bit Gleitkommazahl) zu INT (16-Bit Ganzzahl).
- `F_LREAL_TO_DINT`: Konvertiert LREAL (64-Bit Gleitkommazahl) zu DINT (32-Bit Ganzzahl).
- `F_LREAL_TO_LINT` bietet eine höhere Genauigkeit durch die Verwendung von 64-Bit Ganzahlen.

Fazit

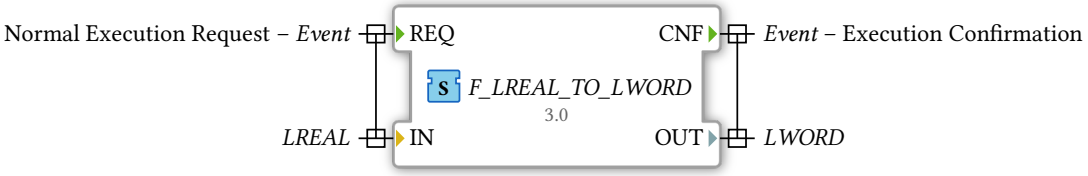
Der Funktionsblock `F_LREAL_TO_LINT` ist ein einfacher und effizienter Baustein zur Konvertierung von 64-Bit Gleitkommazahlen in 64-Bit Ganzahlen. Er eignet sich besonders für Anwendungen, die eine hohe Genauigkeit und einen großen Wertebereich erfordern. Durch seine einfache Schnittstelle und direkte Funktionsweise ist er leicht in bestehende Steuerungssysteme integrierbar.

F_LREAL_TO_LWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LREAL_TO_LWORD` dient der Konvertierung eines LREAL-Wertes (64-Bit Gleitkommazahl) in einen LWORD-Wert (64-Bit vorzeichenlose Ganzzahl). Diese Konvertierung ist insbesondere in Szenarien erforderlich, in denen Gleitkommazahlen in eine binäre Darstellung umgewandelt werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Eingang ausgelöst, wird der Wert am Daten-Eingang `IN` gelesen und konvertiert.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der konvertierte Wert steht am Daten-Ausgang `OUT` zur Verfügung.

Daten-Eingänge

- **IN** (LREAL): Der Eingangswert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (LWORD): Der konvertierte Ausgangswert.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem der LREAL-Wert am Eingang `IN` in einen LWORD-Wert umgewandelt wird. Die Konvertierung erfolgt direkt und ohne zusätzliche Parameter. Bei Auslösung des Ereignisses `REQ` wird der Algorithmus ausgeführt, der den Wert konvertiert und das Ereignis `CNF` ausgibt.

Technische Besonderheiten

- Die Konvertierung erfolgt mittels der Funktion `LREAL_TO_LWORD`, die im Algorithmus des Funktionsblocks implementiert ist.
- Der Funktionsblock ist einfach und hat keine internen Zustände oder komplexe Logik.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge oder interne Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Konvertierung von Gleitkommazahlen in eine binäre Darstellung für die weitere Verarbeitung in anderen Systemen.
- Verwendung in Steuerungssystemen, wo Gleitkommawerte in eine für die Hardware verständliche Form umgewandelt werden müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_REAL_TO_WORD` oder `F_DINT_TO_DWORD` führen ähnliche Konvertierungen durch, jedoch mit anderen Datentypen.
- `F_LREAL_TO_LWORD` ist spezifisch für die Konvertierung von 64-Bit Gleitkommazahlen in 64-Bit vorzeichenlose Ganzzahlen.

Fazit

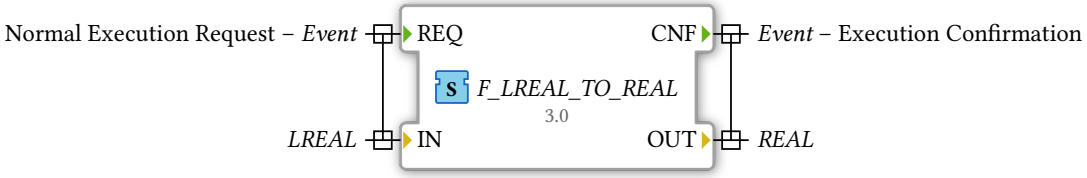
Der Funktionsblock `F_LREAL_TO_LWORD` bietet eine einfache und effiziente Möglichkeit, Gleitkommazahlen in eine binäre Darstellung umzuwandeln. Durch seine klare Schnittstelle und direkte Funktionsweise ist er ideal für Anwendungen, die solche Konvertierungen erfordern.

F_LREAL_TO_REAL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LREAL_TO_REAL` dient der Konvertierung eines LREAL-Wertes (64-Bit Fließkommazahl) in einen REAL-Wert (32-Bit Fließkommazahl). Dieser Baustein ist besonders nützlich in Szenarien, wo eine Typumwandlung zwischen diesen beiden numerischen Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung der Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ LREAL, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgangswert vom Typ REAL, der das Ergebnis der Konvertierung enthält.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Bei Auslösung des Ereigniseingangs `REQ` wird der Algorithmus ausgeführt, der den Wert von `IN` vom Typ LREAL in einen REAL-Wert konvertiert und das Ergebnis an `OUT` ausgibt. Anschließend wird das Ereignis `CNF` ausgelöst, um den Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Der Funktionsblock führt eine direkte Typumwandlung durch, wobei mögliche Genauigkeitsverluste aufgrund der unterschiedlichen Bitbreiten der Datentypen zu beachten sind.
- Die Konvertierung erfolgt mittels der eingebauten Funktion `LREAL_TO_REAL`.

Zustandsübersicht

Der Funktionsblock ist zustandslos und führt bei jedem `REQ`-Ereignis die Konvertierung durch.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo LREAL-Werte aus hochpräzisen Berechnungen in REAL-Werte für weitere Verarbeitungsschritte umgewandelt werden müssen.
- Kompatibilität mit Systemen oder Bausteinen, die nur REAL-Datentypen verarbeiten können.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsbausteinen ist `F_LREAL_TO_REAL` spezialisiert auf die Umwandlung zwischen LREAL und REAL und bietet daher eine klar definierte und effiziente Lösung für diesen spezifischen Anwendungsfall.

Fazit

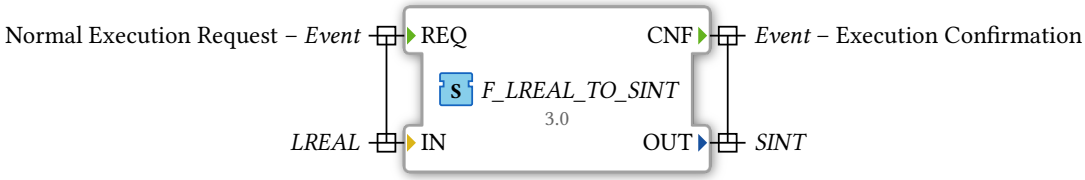
Der `F_LREAL_TO_REAL` Funktionsblock ist ein einfacher, aber effektiver Baustein für die Konvertierung zwischen LREAL- und REAL-Datentypen. Seine klare Schnittstelle und direkte Funktionsweise machen ihn zu einer zuverlässigen Komponente in Steuerungsanwendungen, wo solche Typumwandlungen erforderlich sind.

F_LREAL_TO_SINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LREAL_TO_SINT` dient der Konvertierung eines `LREAL`-Wertes (64-Bit Gleitkommazahl) in einen `SINT`-Wert (8-Bit Ganzahl). Diese Konvertierung ist nützlich, wenn Gleitkommazahlen in eine kompakte Ganzzahl-Darstellung umgewandelt werden müssen, z.B. für die Speicherung oder Übertragung in Systemen mit begrenzter Bandbreite.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (`LREAL`): Der Eingangswert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (`SINT`): Der konvertierte Ausgangswert.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` empfangen wird. Dabei wird der `LREAL`-Wert am Eingang `IN` in einen `SINT`-Wert umgewandelt und am Ausgang `OUT` ausgegeben. Das Ereignis `CNF` signalisiert den Abschluss der Konvertierung.

Technische Besonderheiten

- Die Konvertierung erfolgt durch die Funktion `LREAL_TO_SINT`, die den `LREAL`-Wert in einen `SINT`-Wert umwandelt.
- Es handelt sich um einen einfachen Funktionsblock ohne Zustandsverwaltung oder komplexe Logik.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung wird bei jedem `REQ`-Ereignis neu ausgeführt.

Anwendungsszenarien

- Konvertierung von Gleitkommazahlen in Ganzzahlen für die Speicherung in begrenzten Speicherbereichen.
- Verwendung in Steuerungssystemen, wo Gleitkommazahlen in kompakte Ganzzahlformate umgewandelt werden müssen.
- Integration in Datenverarbeitungsketten, wo eine Reduktion der Datenbreite erforderlich ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_REAL_TO_SINT**: Konvertiert `REAL` (32-Bit Gleitkommazahl) in `SINT`. Im Vergleich zu `F_LREAL_TO_SINT` wird hier eine geringere Präzision der Eingangsdaten verarbeitet.
- **F_LREAL_TO_INT**: Konvertiert `LREAL` in `INT` (16-Bit Ganzzahl). Bietet eine größere Ausgangsbreite im Vergleich zu `F_LREAL_TO_SINT`.

Fazit

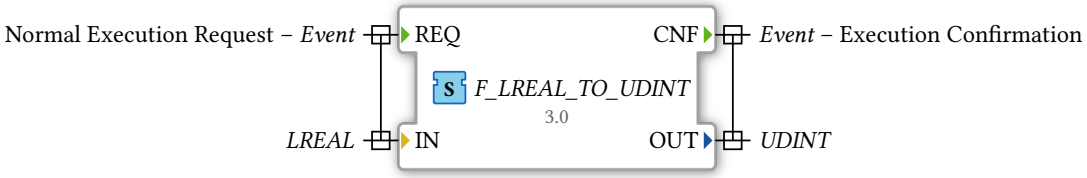
Der Funktionsblock `F_LREAL_TO_SINT` ist ein einfaches und effizientes Werkzeug zur Konvertierung von 64-Bit Gleitkommazahlen in 8-Bit Ganzzahlen. Seine einfache Schnittstelle und direkte Funktionsweise machen ihn ideal für Anwendungen, wo Platz- oder Bandbreitenbeschränkungen eine Rolle spielen.

F_LREAL_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LREAL_TO_UDINT` dient zur Konvertierung eines 64-Bit Gleitkommawerts (LREAL) in einen 32-Bit vorzeichenlosen Integerwert (UDINT). Diese Konvertierung ist nützlich, wenn Gleitkommazahlen in vorzeichenlose Integerwerte umgewandelt werden müssen, z.B. für spezifische Steuerungs- oder Kommunikationsanwendungen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Dieser Ereigniseingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ereignisausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (LREAL): Der Eingangswert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (UDINT): Der konvertierte Ausgangswert.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung von einem LREAL- zu einem UDINT-Wert durch, sobald das `REQ`-Ereignis empfangen wird. Die Konvertierung erfolgt mittels der eingebauten Funktion `LREAL_TO_UDINT`. Nach Abschluss der Konvertierung wird das `CNF`-Ereignis ausgelöst, und der konvertierte Wert steht am Ausgang `OUT` zur Verfügung.

Technische Besonderheiten

- Die Konvertierung schneidet den Gleitkommawert auf den nächsten ganzzahligen Wert ab.
- Es gibt keine Überprüfung auf Überlauf oder Unterlauf. Werte außerhalb des UDINT-Bereichs (0 bis 4.294.967.295) führen zu undefiniertem Verhalten.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Konvertierung von Sensordaten (z.B. Temperaturmessungen) in vorzeichenlose Integerwerte für die weitere Verarbeitung.
- Vorbereitung von Gleitkommawerten für die Kommunikation mit Systemen, die nur vorzeichenlose Integerwerte unterstützen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_REAL_TO_UINT**: Konvertiert 32-Bit Gleitkommawerte (REAL) in 16-Bit vorzeichenlose Integerwerte (UINT).
- **F_LREAL_TO_DINT**: Konvertiert 64-Bit Gleitkommawerte (LREAL) in 32-Bit vorzeichenbehaftete Integerwerte (DINT).
- **F_LREAL_TO_UDINT** ist spezifisch für die Konvertierung zu vorzeichenlosen 32-Bit Integern.

🛠️ Zugehörige Übungen

- Uebung_152
- Uebung_153

Fazit

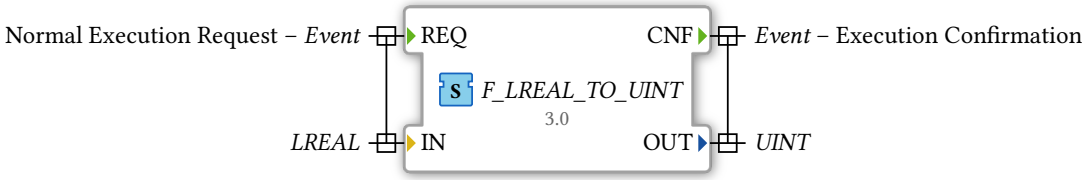
Der Funktionsblock `F_LREAL_TO_UDINT` bietet eine einfache und effiziente Möglichkeit, Gleitkommawerte in vorzeichenlose Integerwerte umzuwandeln. Aufgrund der fehlenden Überlaufprüfung sollte er jedoch nur in Szenarien eingesetzt werden, in denen die Eingabewerte bekannt und innerhalb des zulässigen Bereichs liegen.

F_LREAL_TO_UINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LREAL_TO_UINT` dient der Konvertierung eines LREAL-Wertes (64-Bit Gleitkommazahl) in einen UINT-Wert (vorzeichenlose 16-Bit Ganzzahl). Dieser Baustein ist Teil der `iec61131::conversion`-Bibliothek und ermöglicht eine einfache und effiziente Typumwandlung in Automatisierungsanwendungen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Beim Auftreten dieses Ereignisses wird der Algorithmus zur Umwandlung des Eingabewertes ausgeführt.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieses Ereignis wird nach erfolgreicher Umwandlung des Wertes ausgegeben.

Daten-Eingänge

- **IN**: Der Eingang erwartet eine LREAL-Zahl (64-Bit Gleitkommazahl), die in einen UINT-Wert umgewandelt werden soll.

Daten-Ausgänge

- **OUT**: Gibt das Ergebnis der Konvertierung als UINT-Wert (vorzeichenlose 16-Bit Ganzzahl) aus.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Beim Auftreten des `REQ`-Ereignisses wird der Algorithmus zur Konvertierung des LREAL-Wertes am Eingang `IN` in einen UINT-Wert ausgeführt. Das Ergebnis wird am Ausgang `OUT` ausgegeben und das `CNF`-Ereignis signalisiert den erfolgreichen Abschluss der Operation.

Der Algorithmus verwendet die Funktion `LREAL_TO_UINT` zur direkten Umwandlung des Eingabewertes. Es ist zu beachten, dass bei der Konvertierung von Gleitkommazahlen in Ganzzahlen eine mögliche Abrundung oder Überlauf auftreten kann.

Technische Besonderheiten

- **Datentypen**: Der Funktionsblock verarbeitet 64-Bit Gleitkommazahlen (LREAL) und gibt 16-Bit vorzeichenlose Ganzzahlen (UINT) aus.
- **Algorithmus**: Die Konvertierung erfolgt direkt durch die eingebaute `LREAL_TO_UINT`-Funktion.
- **Fehlerbehandlung**: Der Baustein verfügt über keine integrierte Fehlerbehandlung für Überlauf oder ungültige Eingabewerte.

Zustandsübersicht

1. **Wartezustand**: Der Funktionsblock wartet auf das `REQ`-Ereignis.
2. **Ausführungszustand**: Bei `REQ` wird die Konvertierung durchgeführt.
3. **Bestätigungszustand**: Nach erfolgreicher Konvertierung wird `CNF` ausgegeben und der Baustein kehrt in den Wartezustand zurück.

Anwendungsszenarien

- Konvertierung von Sensorwerten (z.B. Temperaturmessungen) in ganzzahlige Werte für die weitere Verarbeitung.
- Typumwandlung in Steuerungsanwendungen, wo Gleitkommazahlen in Ganzzahlen für bestimmte Operationen benötigt werden.
- Schnittstellenanpassung zwischen Systemkomponenten mit unterschiedlichen Datentypen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken ist `F_LREAL_TO_UINT` spezialisiert auf die Umwandlung von LREAL zu UINT.
- Andere Konvertierungsblöcke könnten zusätzliche Funktionen wie Bereichsprüfung oder Fehlerbehandlung bieten, sind aber komplexer in der Handhabung.

Fazit

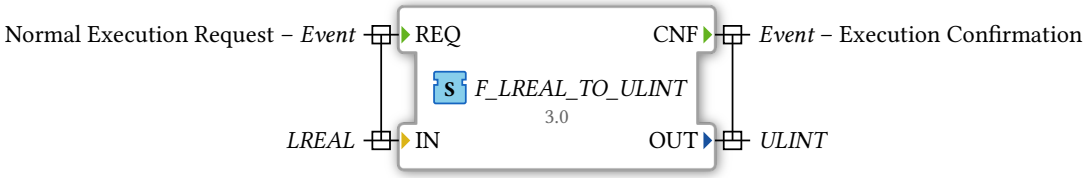
Der `F_LREAL_TO_UINT` Funktionsblock bietet eine einfache und effiziente Lösung für die spezifische Konvertierung von 64-Bit Gleitkommazahlen in 16-Bit vorzeichenlose Ganzahlen. Seine einfache Schnittstelle und direkte Funktionsweise machen ihn ideal für Anwendungen, wo eine schnelle Typumwandlung ohne zusätzliche Funktionen benötigt wird. Für Anwendungen mit kritischen Werten sollten jedoch zusätzliche Überprüfungen implementiert werden.

F_LREAL_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LREAL_TO_ULINT` dient der Konvertierung eines LREAL-Wertes (64-Bit Gleitkommazahl) in einen ULINT-Wert (64-Bit vorzeichenlose Ganzzahl). Diese Konvertierung ist insbesondere in Szenarien erforderlich, in denen Gleitkommazahlen in vorzeichenlose Ganzzahlen umgewandelt werden müssen, z.B. für bestimmte Berechnungen oder zur Speicherung in spezifischen Datenformaten.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ` (Normal Execution Request): Startet die Konvertierung des Eingabewertes. Wird mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- `CNF` (Execution Confirmation): Signalisiert den Abschluss der Konvertierung. Wird mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- `IN` (LREAL): Der Eingabewert, der konvertiert werden soll.

Daten-Ausgänge

- `OUT` (ULINT): Der konvertierte Ausgabewert.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` empfangen wird. Der Algorithmus `REQ` wandelt den LREAL-Wert `IN` mithilfe der Funktion `LREAL_TO_ULINT` in einen ULINT-Wert um und speichert das Ergebnis in `OUT`. Anschließend wird das Ereignis `CNF` ausgelöst, um den Abschluss der Konvertierung zu signalisieren.

Technische Besonderheiten

- Die Konvertierung erfolgt ohne Rundung, d.h. der Nachkommaanteil wird abgeschnitten.
- Bei Werten außerhalb des darstellbaren Bereichs von ULINT (0 bis 2^64-1) kann es zu Überlauf oder unerwarteten Ergebnissen kommen.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Konvertierung von Sensorwerten (z.B. Temperaturmessungen) in vorzeichenlose Ganzzahlen für weitere Verarbeitung.
- Umwandlung von Gleitkommazahlen für die Speicherung in Datenbanken oder Protokollen, die nur Ganzzahlen unterstützen.
- Verwendung in Steuerungsalgorithmen, die vorzeichenlose Ganzzahlen erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- `F_REAL_TO_UINT`: Konvertiert REAL (32-Bit Gleitkommazahl) in UINT (16-Bit vorzeichenlose Ganzzahl).
- `F_LREAL_TO_LINT`: Konvertiert LREAL in LINT (64-Bit vorzeichenbehaftete Ganzzahl).
- `F_LREAL_TO_ULINT` ist spezifisch für 64-Bit vorzeichenlose Konvertierungen und bietet somit eine höhere Genauigkeit und einen größeren Wertebereich im Vergleich zu 16- oder 32-Bit Varianten.

Fazit

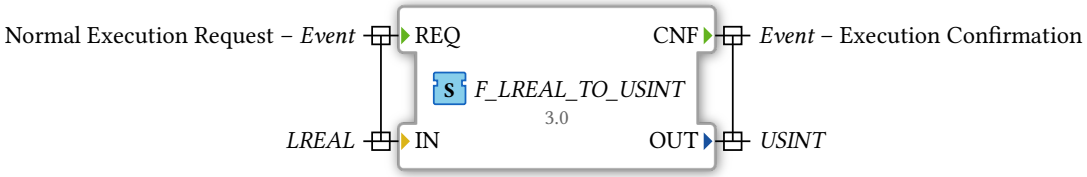
Der Funktionsblock `F_LREAL_TO_ULINT` bietet eine einfache und effiziente Möglichkeit, Gleitkommazahlen in vorzeichenlose Ganzzahlen umzuwandeln. Durch seine klare Schnittstelle und direkte Funktionsweise eignet er sich ideal für Anwendungen, die eine präzise Konvertierung ohne Rundung erfordern.

F_LREAL_TO_USINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LREAL_TO_USINT` dient der Konvertierung eines LREAL-Wertes (64-Bit Gleitkommazahl) in einen USINT-Wert (8-Bit vorzeichenlose Ganzzahl). Diese Konvertierung ist besonders nützlich, wenn Gleitkommazahlen in vorzeichenlose Ganzzahlen umgewandelt werden müssen, z.B. für die Steuerung von Hardware-Registern oder die Kommunikation mit Systemen, die nur Ganzzahlen verarbeiten können.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: LREAL (64-Bit Gleitkommazahl) – Der Eingabewert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: USINT (8-Bit vorzeichenlose Ganzzahl) – Das Ergebnis der Konvertierung.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem der Algorithmus `REQ` ausgeführt wird, sobald das Ereignis `REQ` eintrifft. Der Algorithmus wandelt den LREAL-Wert `IN` mit der Funktion `LREAL_TO_USINT` in einen USINT-Wert um und speichert das Ergebnis in `OUT`. Anschließend wird das Ereignis `CNF` ausgelöst, um den Abschluss der Konvertierung zu signalisieren.

Technische Besonderheiten

- Die Konvertierung schneidet den Gleitkommawert auf einen 8-Bit vorzeichenlosen Integer zu. Es ist zu beachten, dass Werte außerhalb des Bereichs von 0 bis 255 zu unerwarteten Ergebnissen führen können (Überlauf/Unterlauf).
- Der Block ist als einfacher Funktionsblock (SimpleFB) implementiert und verwendet einen ST-Algorithmus für die Konvertierung.

Zustandsübersicht

Der Funktionsblock hat keinen internen Zustand. Die Konvertierung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Umwandlung von Sensorwerten (z.B. Temperatur, Spannung) in vorzeichenlose Ganzzahlen für die Ansteuerung von Hardware.
- Kommunikation mit Systemen, die nur Ganzzahlen verarbeiten können (z.B. ältere Steuerungen oder Protokolle).
- Datenreduktion, wenn eine hohe Präzision nicht erforderlich ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu `F_REAL_TO_USINT` (Konvertierung von 32-Bit Gleitkommazahlen) bietet dieser Block eine höhere Eingangspräzision durch die Verwendung von 64-Bit Gleitkommazahlen.
- Ähnliche Blöcke wie `F_LREAL_TO_INT` oder `F_LREAL_TO_UINT` bieten Konvertierungen in andere Ganzzahltypen mit unterschiedlichen Wertebereichen.

🛠️ Zugehörige Übungen

- SIN_TO_PCAN_Callback_Byte

Fazit

Der `F_LREAL_TO_USINT`-Funktionsblock ist ein einfaches, aber effektives Werkzeug zur Konvertierung von 64-Bit Gleitkommazahlen in 8-Bit vorzeichenlose Ganzzahlen. Er eignet sich besonders für Anwendungen, bei denen eine begrenzte Ausgabegröße erforderlich ist, jedoch sollte der Wertebereich des Eingangssignals beachtet werden, um Überläufe zu vermeiden.

LREAL Konvertierungen

Hier finden Sie Funktionen zur Konvertierung des Datentyps LREAL.

🛠️ Zugehörige Übungen

- Uebung_003a0
- Uebung_003a0_AX
- Uebung_010c2
- Uebung_010c2_AX
- Uebung_035a
- Uebung_035a2
- Uebung_035a3

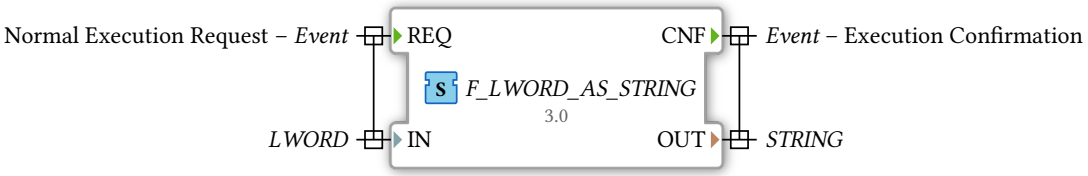
LWORD

F_LWORD_AS_STRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LWORD_AS_STRING` dient der Konvertierung eines `LWORD`-Datentyps in einen `STRING`-Datentyp. Dieser Baustein ist besonders nützlich, wenn große binäre Daten (64-Bit) in eine lesbare Zeichenkette umgewandelt werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Startet die Konvertierung. Dieser Ereigniseingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- `CNF`: Signalisiert den Abschluss der Konvertierung. Dieser Ereignisausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- `IN` (LWORD): Der Eingangswert, der konvertiert werden soll.

Daten-Ausgänge

- `OUT` (STRING): Das Ergebnis der Konvertierung.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das `REQ`-Ereignis ausgelöst wird. Dabei wird der `LWORD`-Wert am Eingang `IN` in einen `STRING` umgewandelt und am Ausgang `OUT` ausgegeben. Die erfolgreiche Konvertierung wird durch das `CNF`-Ereignis signalisiert.

Technische Besonderheiten

- Der Funktionsblock verwendet den Standardalgorithmus `LWORD_AS_STRING` zur Konvertierung.
- Der Baustein ist Teil des Pakets `iec61131::conversion`.

Zustandsübersicht

1. **Idle-Zustand**: Der Funktionsblock wartet auf das `REQ`-Ereignis.
2. **Konvertierungszustand**: Bei Empfang von `REQ` wird die Konvertierung durchgeführt.
3. **Abschlusszustand**: Nach erfolgreicher Konvertierung wird das `CNF`-Ereignis ausgelöst.

Anwendungsszenarien

- Konvertierung von großen binären Daten in lesbare Zeichenketten.
- Verwendung in Systemen, die mit binären und textuellen Daten arbeiten müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu anderen Konvertierungsbausteinen, die möglicherweise andere Datentypen verarbeiten, ist dieser speziell für die Umwandlung von `LWORD` zu `STRING` optimiert.

Fazit

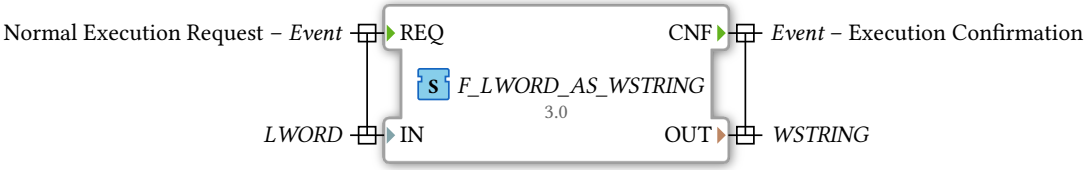
Der `F_LWORD_AS_STRING`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, große binäre Daten in Zeichenketten umzuwandeln. Seine klare Schnittstellenstruktur und einfache Funktionsweise machen ihn zu einem wertvollen Baustein in der Automatisierungstechnik.

F_LWORD_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LWORD_AS_WSTRING` dient zur Konvertierung eines LWORD-Datentyps in einen WSTRING-Datentyp. Diese Funktionalität ist besonders nützlich, wenn Daten zwischen Systemen ausgetauscht werden müssen, die unterschiedliche Datentypen verwenden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Dieser Ereigniseingang löst die Konvertierung aus.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Dieses Ereignis wird ausgegeben, sobald die Konvertierung abgeschlossen ist.

Daten-Eingänge

- **IN**: Eingangsvariable vom Typ LWORD, die den zu konvertierenden Wert enthält.

Daten-Ausgänge

- **OUT**: Ausgangsvariable vom Typ WSTRING, die den konvertierten Wert enthält.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von LWORD zu WSTRING durch, sobald das Ereignis `REQ` empfangen wird. Die Konvertierung erfolgt mittels der integrierten Funktion `LWORD_AS_WSTRING`. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgegeben.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher FB (SimpleFB) und enthält einen einzigen Algorithmus.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Verzögerung.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Ausführung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Datenkonvertierung in industriellen Steuerungssystemen.
- Austausch von Daten zwischen Systemen, die unterschiedliche Datentypen verwenden.
- Integration in größere Steuerungsanwendungen, wo eine Typumwandlung erforderlich ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen ist `F_LWORD_AS_WSTRING` spezialisiert auf die Umwandlung von LWORD zu WSTRING.
- Andere Bausteine könnten allgemeinere Konvertierungen unterstützen, sind aber möglicherweise weniger effizient für diesen spezifischen Anwendungsfall.

Fazit

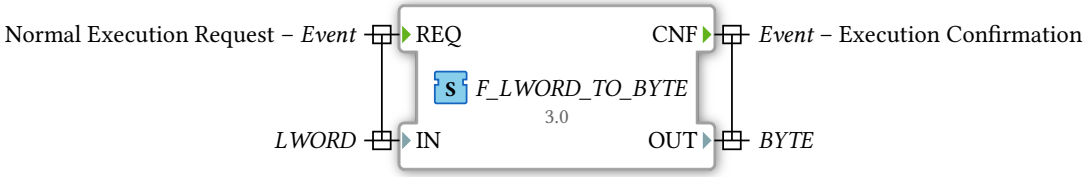
Der Funktionsblock `F_LWORD_AS_WSTRING` bietet eine effiziente und direkte Methode zur Konvertierung von LWORD zu WSTRING. Seine einfache Struktur und klare Funktionsweise machen ihn zu einem zuverlässigen Werkzeug in Steuerungsanwendungen.

F_LWORD_TO_BYTE

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LWORD_TO_BYTE` dient der Konvertierung eines 64-Bit-Wertes (LWORD) in einen 8-Bit-Wert (BYTE). Diese Konvertierung ist besonders nützlich, wenn Daten zwischen Systemen mit unterschiedlichen Wortbreiten ausgetauscht oder verarbeitet werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Eingang ausgelöst, wird der Wert am Daten-Eingang `IN` gelesen und konvertiert.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der konvertierte Wert steht am Daten-Ausgang `OUT` zur Verfügung.

Daten-Eingänge

- **IN**: Der Eingang für den 64-Bit-Wert (LWORD), der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgang für den konvertierten 8-Bit-Wert (BYTE).

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` ausgelöst wird. Dabei wird der Wert am Eingang `IN` gelesen und mit der Funktion `LWORD_TO_BYTE` in einen BYTE-Wert umgewandelt. Das Ergebnis wird am Ausgang `OUT` ausgegeben und das Ereignis `CNF` signalisiert den Abschluss der Operation.

Technische Besonderheiten

- Die Konvertierung kann zu Datenverlust führen, da ein 64-Bit-Wert in einen 8-Bit-Wert umgewandelt wird. Es ist wichtig sicherzustellen, dass der ursprüngliche Wert innerhalb des darstellbaren Bereichs eines BYTE (0 bis 255) liegt.
- Der Funktionsblock ist als einfacher Funktionsblock (SimpleFB) implementiert und verwendet einen ST-Algorithmus für die Konvertierung.

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf das Ereignis `REQ`.
2. **Aktiv-Zustand**: Führt die Konvertierung durch und gibt das Ergebnis aus.
3. **Abschluss-Zustand**: Signalisiert den Abschluss mit `CNF` und kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- Datenreduktion: Wenn nur die unteren 8 Bit eines 64-Bit-Wertes relevant sind.
- Kommunikation mit Geräten, die nur BYTE-Daten verarbeiten können.
- Speicheroptimierung in Systemen mit begrenzten Ressourcen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_WORD_TO_BYTE**: Konvertiert einen 16-Bit-Wert (WORD) in einen BYTE.
- **F_DWORD_TO_BYTE**: Konvertiert einen 32-Bit-Wert (DWORD) in einen BYTE.
- **F_LWORD_TO_WORD**: Konvertiert einen 64-Bit-Wert in einen 16-Bit-Wert.

Fazit

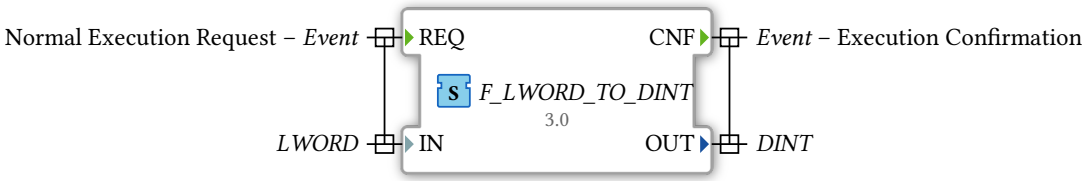
Der Funktionsblock `F_LWORD_TO_BYTE` bietet eine einfache und effiziente Möglichkeit, 64-Bit-Daten in 8-Bit-Daten umzuwandeln. Er ist besonders nützlich in Szenarien, wo nur ein Teil der Daten benötigt wird oder die Zielplattform nur begrenzte Datentypen unterstützt. Beachten Sie jedoch die möglichen Datenverluste bei der Konvertierung.

F_LWORD_TO_DINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LWORD_TO_DINT` dient der Konvertierung eines `LWORD`-Datentyps in einen `DINT`-Datentyp. Dieser Baustein ist besonders nützlich in Szenarien, wo eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist, beispielsweise bei der Verarbeitung von großen binären Daten oder bei der Kommunikation zwischen Systemen mit unterschiedlichen Datentypanforderungen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Dieser Ereigniseingang löst die Ausführung des Funktionsblocks aus. Er ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Dieser Ereignisausgang signalisiert die erfolgreiche Ausführung des Funktionsblocks. Er ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN** (LWORD): Der Eingang für den zu konvertierenden `LWORD`-Wert.

Daten-Ausgänge

- **OUT** (DINT): Der Ausgang für den konvertierten `DINT`-Wert.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock konvertiert den eingehenden `LWORD`-Wert (`IN`) in einen `DINT`-Wert (`OUT`), sobald das Ereignis `REQ` ausgelöst wird. Die Konvertierung erfolgt mithilfe der eingebauten Funktion `LWORD_TO_DINT`. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst, um die Fertigstellung der Operation anzuzeigen.

Technische Besonderheiten

- Der Funktionsblock ist einfach aufgebaut und besteht aus einem einzigen Algorithmus, der die Konvertierung durchführt.
- Es gibt keine Zustandsmaschine oder komplexe Logik, da die Konvertierung direkt und ohne Zwischenschritte erfolgt.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge oder komplexe Zustandslogik. Der Block führt die Konvertierung bei jedem `REQ`-Ereignis aus und signalisiert die Fertigstellung mit `CNF`.

Anwendungsszenarien

- **Datenkonvertierung**: In Systemen, wo `LWORD`-Daten in `DINT` umgewandelt werden müssen, z.B. bei der Kommunikation zwischen unterschiedlichen Hardwarekomponenten.
- **Datenverarbeitung**: Bei der Verarbeitung von großen binären Daten, die in einem kleineren Format weiterverarbeitet werden sollen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_LWORD_TO_INT**: Ähnlicher Funktionsblock, aber konvertiert in den kleineren `INT`-Datentyp.
- **F_LWORD_TO_UDINT**: Konvertiert in einen vorzeichenlosen `DINT`-Datentyp (`UDINT`).

Fazit

Der `F_LWORD_TO_DINT`-Funktionsblock ist ein einfaches und effizientes Werkzeug zur Konvertierung von `LWORD`- in `DINT`-Daten. Seine Einfachheit und direkte Funktionsweise machen ihn zu einer zuverlässigen Lösung für Typumwandlungen in 4diac-IDE-basierten Systemen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

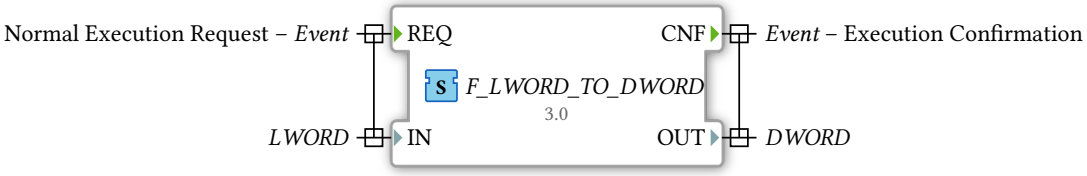
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

F_LWORD_TO_DWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LWORD_TO_DWORD` dient der Konvertierung eines LWORD-Datentyps in einen DWORD-Datentyp. Diese Konvertierung ist nützlich, wenn Daten zwischen Systemen oder Komponenten ausgetauscht werden müssen, die unterschiedliche Datentypen verwenden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Dieser Ereigniseingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ereignisausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Eingang für den zu konvertierenden LWORD-Wert.

Daten-Ausgänge

- **OUT**: Ausgang für den konvertierten DWORD-Wert.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von LWORD zu DWORD durch, wenn das Ereignis `REQ` ausgelöst wird. Der Algorithmus innerhalb des Funktionsblocks nutzt die Funktion `LWORD_TO_DWORD`, um die Konvertierung durchzuführen. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst, um den Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher Funktionsblock (SimpleFB) und führt eine direkte Konvertierung durch.
- Es gibt keine zusätzlichen Zustände oder komplexe Logiken, da die Konvertierung in einem Schritt erfolgt.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge oder Zustandsmaschinen. Der Funktionsblock reagiert direkt auf das `REQ`-Ereignis und gibt das Ergebnis über `CNF` aus.

Anwendungsszenarien

- **Datenkompatibilität**: Wenn ein System LWORD-Daten erzeugt, ein anderes System aber nur DWORD-Daten verarbeiten kann.
- **Protokollumsetzung**: Bei der Kommunikation zwischen verschiedenen Protokollen oder Schnittstellen, die unterschiedliche Datentypen verwenden.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_DWORD_TO_LWORD**: Führt die umgekehrte Konvertierung durch (DWORD zu LWORD).
- **F_WORD_TO_DWORD**: Konvertiert WORD zu DWORD, hat aber eine geringere Eingangsgröße im Vergleich zu LWORD.

Fazit

Der `F_LWORD_TO_DWORD` Funktionsblock ist ein einfaches und effizientes Werkzeug zur Konvertierung von LWORD- zu DWORD-Daten. Seine Einfachheit und direkte Funktionsweise machen ihn zu einer zuverlässigen Lösung für Datenkonvertierungsaufgaben in 4diac-IDE-basierten Systemen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

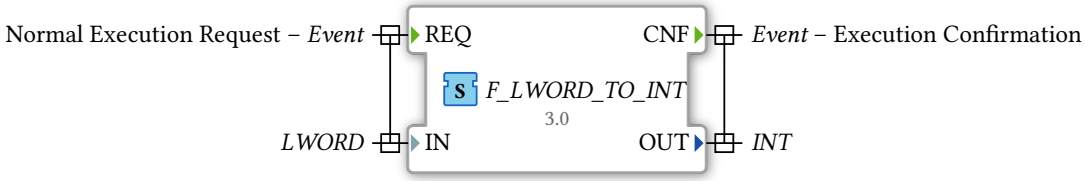
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

F_LWORD_TO_INT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LWORD_TO_INT` dient der Konvertierung eines `LWORD`-Wertes (64-Bit unsigned integer) in einen `INT`-Wert (16-Bit signed integer). Diese Konvertierung ist nützlich, wenn Daten zwischen Systemen mit unterschiedlichen Wortbreiten ausgetauscht oder verarbeitet werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `LWORD` (64-Bit unsigned integer), der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgangswert vom Typ `INT` (16-Bit signed integer), der das Ergebnis der Konvertierung enthält.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem der `LWORD`-Wert am Eingang `IN` in einen `INT`-Wert umgewandelt und am Ausgang `OUT` ausgegeben wird. Die Konvertierung erfolgt bei einem Ereignis am Eingang `REQ` und wird durch ein Ereignis am Ausgang `CNF` bestätigt.

Technische Besonderheiten

- Die Konvertierung von einem 64-Bit-Wert in einen 16-Bit-Wert kann zu einem Datenverlust führen, wenn der ursprüngliche Wert außerhalb des darstellbaren Bereichs von `INT` liegt.
- Der Funktionsblock ist als einfacher Funktionsblock (`SimpleFB`) implementiert und verwendet einen ST-Algorithmus für die Konvertierung.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei einem Ereignis am Eingang `REQ`.

Anwendungsszenarien

- Konvertierung von 64-Bit-Datenwerten für Systeme, die nur 16-Bit-Daten verarbeiten können.
- Datenreduktion, wenn nur der untere 16-Bit-Teil eines 64-Bit-Werts relevant ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- `F_DWORD_TO_INT`: Konvertiert einen 32-Bit-Wert in einen 16-Bit-Wert.
- `F_LWORD_TO_DINT`: Konvertiert einen 64-Bit-Wert in einen 32-Bit-Wert.
- `F_LWORD_TO_SINT`: Konvertiert einen 64-Bit-Wert in einen 8-Bit-Wert.

Fazit

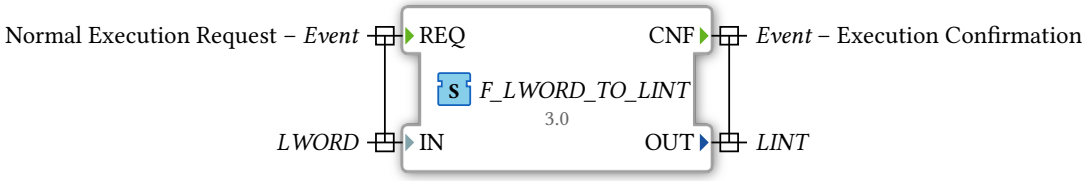
Der Funktionsblock `F_LWORD_TO_INT` bietet eine einfache und effiziente Möglichkeit, 64-Bit-Datenwerte in 16-Bit-Datenwerte zu konvertieren. Bei der Verwendung ist jedoch auf den möglichen Datenverlust zu achten, wenn der ursprüngliche Wert außerhalb des darstellbaren Bereichs von `INT` liegt.

F_LWORD_TO_LINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LWORD_TO_LINT` dient der Konvertierung eines `LWORD`-Wertes (64-Bit unsigned) in einen `LINT`-Wert (64-Bit signed). Dieser Baustein ist besonders nützlich in Szenarien, wo eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Dieser Ereigniseingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ereignisausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Eingang für den `LWORD`-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Ausgang für den konvertierten `LINT`-Wert.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `LWORD` zu `LINT` durch, sobald das Ereignis `REQ` ausgelöst wird. Die Konvertierung erfolgt mittels der Funktion `LWORD_TO_LINT(IN)`. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher FB (SimpleFB) und verwendet einen ST-Algorithmus für die Konvertierung.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Zustandsverwaltung.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine komplexen Zustandsübergänge. Der Block reagiert auf `REQ` mit der Ausführung des Algorithmus und signalisiert den Abschluss mit `CNF`.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo `LWORD`-Daten in `LINT` umgewandelt werden müssen.
- Integration in größere Systeme, die mit unterschiedlichen Datentypen arbeiten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Bausteine könnten `F_DWORD_TO_DINT` oder `F_WORD_TO_INT` sein, die jedoch für kleinere Datentypen ausgelegt sind.
- Im Gegensatz zu generischen Konvertierungsblöcken ist dieser speziell für die Umwandlung von `LWORD` zu `LINT` optimiert.

Fazit

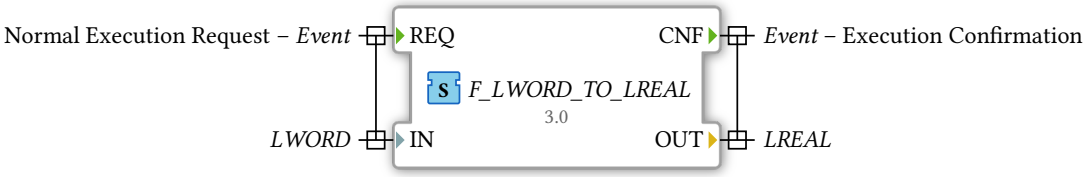
Der `F_LWORD_TO_LINT`-Funktionsblock bietet eine effiziente und direkte Methode zur Konvertierung von `LWORD` zu `LINT`. Seine Einfachheit und Spezialisierung machen ihn zu einer zuverlässigen Komponente in Steuerungsanwendungen.

F_LWORD_TO_LREAL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LWORD_TO_LREAL` dient der Konvertierung eines LWORD-Datenwerts (64-Bit Ganzzahl ohne Vorzeichen) in einen LREAL-Datenwert (64-Bit Gleitkommazahl nach IEEE 754). Dieser Baustein ist besonders nützlich in Anwendungen, wo eine direkte Typumwandlung zwischen diesen beiden Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Bei diesem Ereignis wird der Eingangswert `IN` verarbeitet.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung und gibt den konvertierten Wert `OUT` aus.

Daten-Eingänge

- **IN** (LWORD): Der Eingangswert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (LREAL): Der Ergebniswert der Konvertierung.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Bei Auslösung des `REQ`-Ereignisses wird der Algorithmus ausgeführt, der den `IN`-Wert vom Typ LWORD in einen LREAL-Wert konvertiert. Das Ergebnis wird über den `OUT`-Ausgang ausgegeben und das `CNF`-Ereignis signalisiert die erfolgreiche Konvertierung.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne zusätzliche Parameter.

- Der Baustein ist für die Verwendung in Echtzeitsystemen optimiert.

Zustandsübersicht

Der Funktionsblock hat einen einfachen Zustandsautomaten:

1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Processing**: Führt die Konvertierung durch.
3. **Complete**: Sendet das `CNF`-Ereignis mit dem Ergebnis.

Anwendungsszenarien

- Datenverarbeitung in Steuerungssystemen, wo LWORD-Werte in Gleitkommazahlen umgewandelt werden müssen.
- Integration in größere Funktionsblocknetzwerke, die gemischte Datentypen verarbeiten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Bausteine wie `F_DWORD_TO_REAL` oder `F_WORD_TO_REAL` bieten Konvertierungen für kleinere Datentypen.
- `F_LWORD_TO_LREAL` ist spezifisch für 64-Bit Konvertierungen optimiert.

Fazit

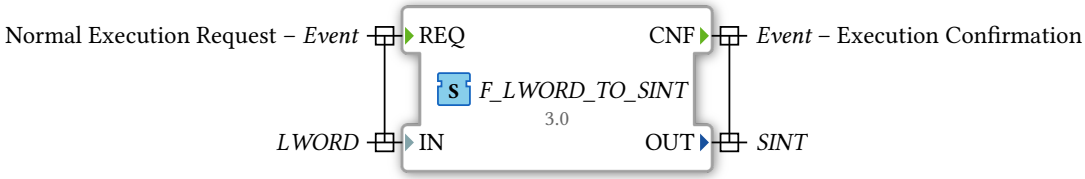
Der `F_LWORD_TO_LREAL` Funktionsblock ist ein effizientes Werkzeug zur direkten Konvertierung von 64-Bit Ganzahlen in Gleitkommazahlen. Seine einfache Schnittstelle und zuverlässige Funktionsweise machen ihn ideal für den Einsatz in komplexen Automatisierungslösungen.

F_LWORD_TO_SINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LWORD_TO_SINT` dient der Konvertierung eines `LWORD`-Wertes (64-Bit unsigned integer) in einen `SINT`-Wert (8-Bit signed integer). Diese Konvertierung ist nützlich, wenn Daten aus einem größeren Wertebereich in einen kleineren, spezifischeren Bereich übertragen werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung des Funktionsblocks. Bei Aktivierung wird der Eingangswert `IN` verarbeitet.

Ereignis-Ausgänge

- **CNF**: Signalisiert die erfolgreiche Beendigung der Konvertierung und stellt den Ergebniswert `OUT` bereit.

Daten-Eingänge

- **IN** (`LWORD`): Der Eingangswert, der von `LWORD` in `SINT` konvertiert wird.

Daten-Ausgänge

- **OUT** (`SINT`): Das Ergebnis der Konvertierung.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt eine direkte Typumwandlung des Eingangswertes `IN` vom Typ `LWORD` in den Typ `SINT` durch. Die Konvertierung erfolgt durch die Funktion `LWORD_TO_SINT(IN)`, die im Algorithmus `REQ` implementiert ist. Bei erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- **Typumwandlung**: Da `LWORD` einen größeren Wertebereich als `SINT` abdeckt, kann es zu einem Datenverlust kommen, wenn der Eingangswert außerhalb des darstellbaren Bereichs von `SINT` (-128 bis 127) liegt.
- **Algorithmus**: Der Algorithmus ist in Structured Text (ST) implementiert und führt eine direkte Konvertierung durch.

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf das Ereignis `REQ`.
2. **Ausführungszustand**: Bei `REQ` wird die Konvertierung durchgeführt.
3. **Bestätigungszustand**: Nach erfolgreicher Konvertierung wird `CNF` ausgelöst und der Block kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- **Datenreduktion**: Wenn nur die unteren 8 Bit eines 64-Bit-Wertes relevant sind.
- **Kommunikation**: Bei der Übertragung von Daten zwischen Systemen mit unterschiedlichen Wortgrößen.
- **Speicheroptimierung**: Wenn Speicherplatz begrenzt ist und kleinere Datentypen bevorzugt werden.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_LWORD_TO_INT**: Konvertiert `LWORD` in `INT` (16-Bit signed integer).
- **F_LWORD_TO_DINT**: Konvertiert `LWORD` in `DINT` (32-Bit signed integer).
- **F_LWORD_TO_USINT**: Konvertiert `LWORD` in `USINT` (8-Bit unsigned integer).

Im Gegensatz zu diesen Blöcken ist `F_LWORD_TO_SINT` spezifisch für die Konvertierung in einen 8-Bit signed integer.

Fazit

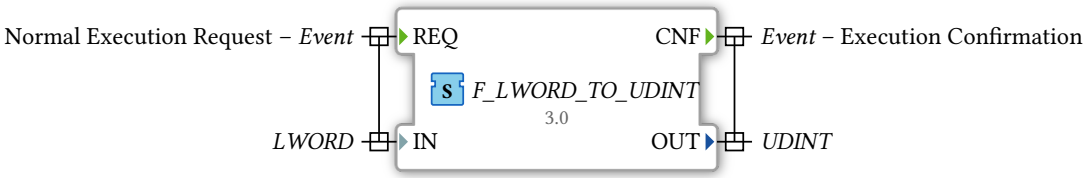
Der Funktionsblock `F_LWORD_TO_SINT` bietet eine einfache und effiziente Möglichkeit, 64-Bit-Werte in 8-Bit signed integers umzuwandeln. Er ist besonders nützlich in Szenarien, wo Speichereffizienz oder die Verarbeitung von Teilen eines größeren Datentyps erforderlich ist. Bei der Verwendung ist jedoch auf den möglichen Datenverlust bei Werten außerhalb des `SINT`-Bereichs zu achten.

F_LWORD_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LWORD_TO_UDINT` dient der Konvertierung eines `LWORD`-Wertes (64-Bit unsigned integer) in einen `UDINT`-Wert (32-Bit unsigned integer). Diese Konvertierung ist nützlich, wenn Daten zwischen Systemen mit unterschiedlichen Wortbreiten übertragen oder verarbeitet werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Startet die Konvertierung. Dieser Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- `CNF`: Signalisiert den Abschluss der Konvertierung. Dieser Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- `IN` (LWORD): Der zu konvertierende 64-Bit unsigned integer Wert.

Daten-Ausgänge

- `OUT` (UDINT): Der resultierende 32-Bit unsigned integer Wert nach der Konvertierung.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Auslösung des `REQ`-Ereignisses wird der Wert von `IN` von `LWORD` in `UDINT` konvertiert und das Ergebnis an `OUT` ausgegeben. Das `CNF`-Ereignis signalisiert die erfolgreiche Beendigung der Konvertierung.

Die Konvertierung erfolgt durch die Funktion `LWORD_TO_UDINT`, welche die unteren 32 Bit des `LWORD`-Wertes als `UDINT` zurückgibt.

Technische Besonderheiten

- Der Funktionsblock ist für die Verwendung in Echtzeitsystemen geeignet.
- Die Konvertierung ist deterministisch und hat eine konstante Ausführungszeit.
- Bei Werten größer als der maximale `UDINT`-Wert (2^32 - 1) wird der Wert abgeschnitten.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Kommunikation zwischen Systemen mit unterschiedlichen Wortbreiten.
- Verarbeitung von Daten aus externen Quellen, die 64-Bit Werte liefern, aber nur 32-Bit Werte benötigt werden.
- Reduzierung der Speichernutzung bei bekannten kleinen Wertebereichen.

⚖️ Vergleich mit ähnlichen Bausteinen

- `F_LWORD_TO_DINT`: Konvertiert in einen 32-Bit signed integer.
- `F_LWORD_TO_UINT`: Konvertiert in einen 16-Bit unsigned integer.
- `F_LWORD_TO_WORD`: Konvertiert in einen 16-Bit Wert ohne Vorzeichen.

Fazit

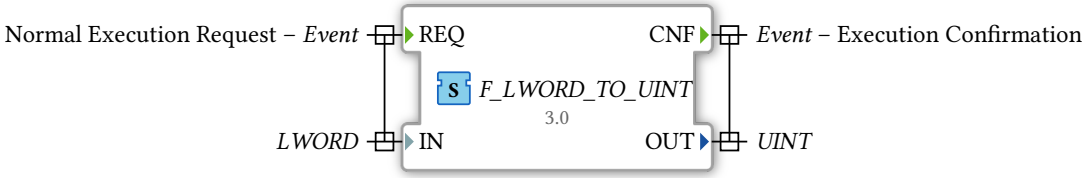
Der `F_LWORD_TO_UDINT` Funktionsblock bietet eine einfache und effiziente Möglichkeit, 64-Bit Werte in 32-Bit Werte zu konvertieren. Er ist besonders nützlich in Systemen, die mit unterschiedlichen Wortbreiten arbeiten müssen.

F_LWORD_TO_UINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LWORD_TO_UINT` dient der Konvertierung eines `LWORD`-Datentyps in einen `UINT`-Datentyp. Dieser Baustein ist besonders nützlich in Szenarien, wo eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist, beispielsweise bei der Kommunikation zwischen Systemen mit unterschiedlichen Datenformaten.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Dieser Ereigniseingang löst die Konvertierung aus. Er ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Dieses Ereignis wird nach erfolgreicher Konvertierung ausgelöst und ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Eingangsdaten vom Typ `LWORD`, die in den `UINT`-Datentyp konvertiert werden sollen.

Daten-Ausgänge

- **OUT**: Ausgangsdaten vom Typ `UINT`, die das Ergebnis der Konvertierung enthalten.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `LWORD` zu `UINT` durch, wenn das Ereignis `REQ` empfangen wird. Die Konvertierung erfolgt mittels der Funktion `LWORD_TO_UINT(IN)`, die den Wert von `IN` umwandelt und das Ergebnis an `OUT` weitergibt. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher Algorithmus (SimpleFB) und führt keine komplexen Zustandsübergänge durch.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Bedingungen oder Verzögerungen.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine expliziten Zustände. Die Funktionalität beschränkt sich auf die direkte Reaktion auf das `REQ`-Ereignis mit anschließender Ausgabe des konvertierten Wertes und Auslösung des `CNF`-Ereignisses.

Anwendungsszenarien

- Typumwandlung in Kommunikationsschnittstellen zwischen Systemen, die unterschiedliche Datentypen verwenden.
- Datenverarbeitung in Steuerungssystemen, wo eine Konvertierung von `LWORD` zu `UINT` erforderlich ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke könnten `F_DWORD_TO_UINT` oder `F_WORD_TO_UINT` sein, die jedoch andere Eingangsdatentypen verarbeiten.
- Der Hauptunterschied liegt im Eingangsdatentyp (`LWORD`), der eine größere Bitbreite als `DWORD` oder `WORD` aufweist.

Fazit

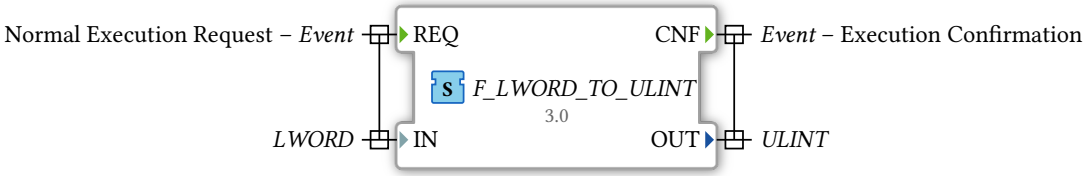
Der `F_LWORD_TO_UINT` Funktionsblock bietet eine einfache und effiziente Möglichkeit, `LWORD`-Daten in den `UINT`-Datentyp umzuwandeln. Seine klare Schnittstellenstruktur und direkte Funktionsweise machen ihn zu einem nützlichen Baustein in verschiedenen Anwendungsfällen, insbesondere dort, wo Typkonvertierungen zwischen diesen Datentypen erforderlich sind.

F_LWORD_TO_ULINT

• − ◦ • − ◦ • − ◦ •

Einleitung

Der Funktionsblock `F_LWORD_TO_ULINT` dient der Konvertierung eines LWORD-Datentyps in einen ULINT-Datentyp. Diese Konvertierung ist insbesondere in Szenarien erforderlich, in denen Daten unterschiedlicher Bitlängen verarbeitet oder weitergegeben werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Dieser Ereigniseingang löst die Konvertierung aus. Er ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Dieser Ereignisausgang signalisiert die erfolgreiche Beendigung der Konvertierung. Er ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingang erwartet eine Variable vom Typ `LWORD`, die konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgang liefert das Ergebnis der Konvertierung als `ULINT`.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Bei Auslösung des Ereigniseingangs `REQ` wird der Algorithmus gestartet, der den Wert von `IN` vom Typ `LWORD` in den Typ `ULINT` konvertiert und das Ergebnis an `OUT` ausgibt. Anschließend wird das Ereignis `CNF` ausgelöst, um die erfolgreiche Konvertierung zu signalisieren.

Technische Besonderheiten

- Der Funktionsblock ist besonders effizient, da er direkt die eingebaute Funktion `LWORD_TO_ULINT` nutzt.
- Es gibt keine besonderen Randbedingungen oder Einschränkungen bei der Verwendung dieses Funktionsblocks.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Datenkonvertierung in Steuerungssystemen, wo unterschiedliche Datentypen verarbeitet werden müssen.
- Integration in größere Systeme, die eine Typumwandlung zwischen `LWORD` und `ULINT` erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Konvertierungsbausteinen ist `F_LWORD_TO_ULINT` spezialisiert auf die Umwandlung von `LWORD` zu `ULINT` und bietet daher eine optimierte Lösung für diesen spezifischen Anwendungsfall.

Fazit

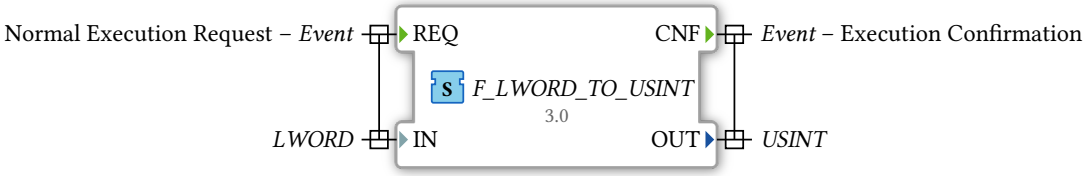
Der `F_LWORD_TO_ULINT` Funktionsblock ist ein einfacher und effizienter Baustein zur Konvertierung zwischen den Datentypen `LWORD` und `ULINT`. Seine klare Schnittstellenstruktur und direkte Funktionsweise machen ihn zu einer zuverlässigen Komponente in Steuerungsanwendungen.

F_LWORD_TO_USINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LWORD_TO_USINT` dient der Konvertierung eines 64-Bit-Wertes (LWORD) in einen 8-Bit-unsigned-Integer-Wert (USINT). Diese Konvertierung ist besonders nützlich, wenn Daten aus einem größeren Wertebereich in einen kleineren, spezifischeren Bereich umgewandelt werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Dieser Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung und liefert das Ergebnis über den Daten-Ausgang `OUT`.

Daten-Eingänge

- **IN (LWORD)**: Der 64-Bit-Wert, der in einen USINT-Wert umgewandelt werden soll.

Daten-Ausgänge

- **OUT (USINT)**: Der resultierende 8-Bit-unsigned-Integer-Wert nach der Konvertierung.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` ausgelöst wird. Der Algorithmus wandelt den 64-Bit-Eingangswert `IN` in einen 8-Bit-unsigned-Integer-Wert um und speichert das Ergebnis in `OUT`. Anschließend wird das Ereignis `CNF` ausgelöst, um den erfolgreichen Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Die Konvertierung erfolgt durch die eingebaute Funktion `LWORD_TO_USINT`.
- Es ist zu beachten, dass bei der Konvertierung von einem größeren zu einem kleineren Datentyp Daten verloren gehen können, wenn der ursprüngliche Wert außerhalb des darstellbaren Bereichs des Zieltyps liegt.

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf das `REQ`-Ereignis.
2. **Konvertierungs-Zustand**: Führt die Konvertierung durch.
3. **Bestätigungs-Zustand**: Sendet das `CNF`-Ereignis und kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- Verarbeitung von 64-Bit-Daten, die auf 8-Bit reduziert werden müssen.
- Kommunikation zwischen Systemen mit unterschiedlichen Datenbreiten.
- Speicherplatzoptimierung in eingebetteten Systemen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_LWORD_TO_BYTE**: Konvertiert ebenfalls LWORD in einen 8-Bit-Wert, jedoch ohne Vorzeichen (BYTE).
- **F_LWORD_TO_INT**: Konvertiert in einen 16-Bit-signed-Integer, bietet einen größeren Wertebereich, aber mit Vorzeichen.

Fazit

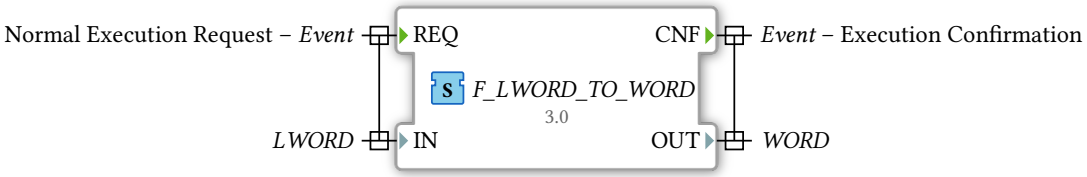
Der `F_LWORD_TO_USINT` Funktionsblock ist ein einfaches und effizientes Werkzeug zur Konvertierung von 64-Bit-Daten in 8-Bit-unsigned-Integer-Werte. Er ist besonders in Szenarien nützlich, wo Speicherplatz oder Datenbandbreite begrenzt sind. Es sollte jedoch sichergestellt werden, dass die ursprünglichen Daten innerhalb des darstellbaren Bereichs des Zieltyps liegen, um Datenverluste zu vermeiden.

F_LWORD_TO_WORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_LWORD_TO_WORD` dient der Konvertierung eines 64-Bit-Wertes (LWORD) in einen 16-Bit-Wert (WORD). Diese Konvertierung ist insbesondere in Szenarien erforderlich, in denen Daten zwischen Systemen mit unterschiedlicher Wortbreite ausgetauscht oder verarbeitet werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Signalisiert den Beginn der Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: LWORD (64-Bit) – Der Eingangswert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: WORD (16-Bit) – Der konvertierte Ausgangswert.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von einem 64-Bit-Wert (LWORD) zu einem 16-Bit-Wert (WORD) durch. Bei der Konvertierung werden die unteren 16 Bit des Eingangswertes übernommen, während die oberen 48 Bit verworfen werden. Die Konvertierung erfolgt synchron zum `REQ`-Ereignis, und das Ergebnis wird mit dem `CNF`-Ereignis bestätigt.

Technische Besonderheiten

- Die Konvertierung erfolgt ohne Rundung oder Skalierung, da es sich um eine direkte Bit-Übernahme handelt.
- Es findet keine Überprüfung auf möglichen Datenverlust durch die Verkürzung der Wortbreite statt. Der Anwender muss sicherstellen, dass der Eingangswert im zulässigen Bereich liegt.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung wird bei jedem `REQ`-Ereignis neu ausgeführt.

Anwendungsszenarien

- Kommunikation zwischen Systemen mit unterschiedlicher Wortbreite.
- Verarbeitung von Daten, die aus 64-Bit-Quellen stammen, aber in 16-Bit-Systemen weiterverwendet werden sollen.

- Reduktion der Datenbreite zur Speicherplatzoptimierung, wenn die höherwertigen Bits nicht benötigt werden.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_LWORD_TO_DWORD**: Konvertiert LWORD zu DWORD (32-Bit), behält mehr Datenbits bei.
- **F_LWORD_TO_BYTE**: Konvertiert LWORD zu BYTE (8-Bit), reduziert die Datenbreite noch stärker.
- **F_LWORD_TO_**: Ähnliche Bausteine für andere Ziel-Datentypen, jeweils mit unterschiedlicher Bit-Breite im Ergebnis.

Fazit

Der `F_LWORD_TO_WORD`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, 64-Bit-Daten auf 16-Bit-Daten zu reduzieren. Er ist besonders nützlich in heterogenen Systemumgebungen, in denen Daten zwischen Komponenten mit unterschiedlicher Wortbreite ausgetauscht werden müssen. Aufgrund der fehlenden Überprüfung auf Datenverlust sollte der Anwender jedoch sicherstellen, dass die Konvertierung sinnvoll und sicher ist.

LWORD Konvertierungen

Hier finden Sie Funktionen zur Konvertierung des Datentyps LWORD.

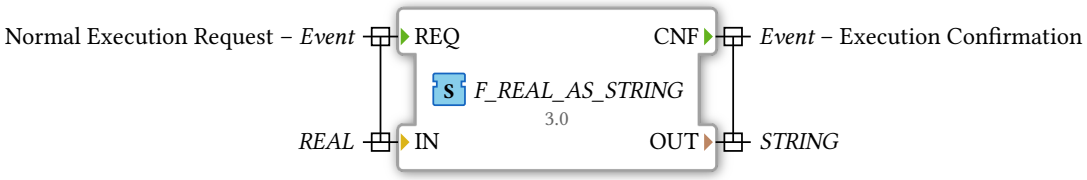
REAL

F_REAL_AS_STRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_REAL_AS_STRING` dient der Konvertierung eines REAL-Wertes in einen STRING-Wert. Er ist Teil des Pakets `iec61131::conversion` und ermöglicht die einfache Umwandlung von numerischen Werten in eine Zeichenkette.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung des REAL-Wertes in einen STRING. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der REAL-Wert, der in einen STRING konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der konvertierte STRING-Wert des Eingangs `IN`.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Bei einem anliegenden Ereignis am Eingang `REQ` wird der Algorithmus ausgeführt, der den REAL-Wert von `IN` in einen STRING konvertiert und das Ergebnis an `OUT` ausgibt. Anschließend wird das Ereignis `CNF` ausgelöst, um den erfolgreichen Abschluss der Konvertierung zu signalisieren.

Technische Besonderheiten

- Der Funktionsblock verwendet die Funktion `REAL_AS_STRING` zur Konvertierung.
- Es handelt sich um einen einfachen Funktionsblock ohne Zustandsautomat.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge. Die Konvertierung erfolgt direkt bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Konvertierung von Sensorwerten für die Anzeige auf einem HMI.
- Umwandlung von numerischen Werten für die Protokollierung in Textdateien.
- Integration in Systeme, die String-basierte Kommunikation erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen wie `F_INT_AS_STRING` oder `F_BOOL_AS_STRING` ist dieser speziell für REAL-Werte ausgelegt.
- Einfacher und direkter als universelle Konvertierungsbausteine, die möglicherweise zusätzliche Parameter benötigen.

Fazit

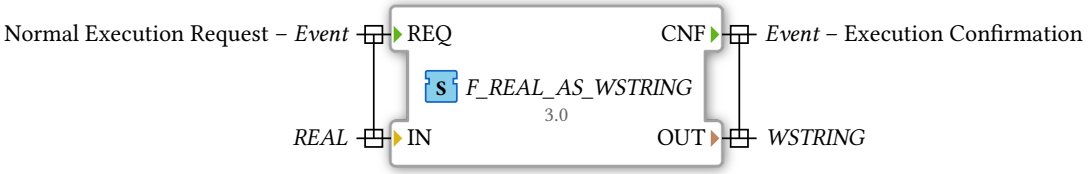
Der `F_REAL_AS_STRING` Funktionsblock bietet eine effiziente und unkomplizierte Möglichkeit, REAL-Werte in STRINGs umzuwandeln. Seine einfache Schnittstelle und direkte Funktionsweise machen ihn ideal für Anwendungen, bei denen numerische Werte als Zeichenketten verarbeitet werden müssen.

F_REAL_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_REAL_AS_WSTRING` dient der Konvertierung eines `REAL`-Wertes in einen `WSTRING`-Wert. Dieser Baustein ist besonders nützlich, wenn numerische Werte in eine für die Ausgabe oder Weiterverarbeitung geeignete Zeichenkette umgewandelt werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Dieser Eingang löst die Konvertierung aus. Bei einem ankommenden Ereignis wird der Algorithmus zur Umwandlung des `REAL`-Wertes in einen `WSTRING`-Wert gestartet.

Ereignis-Ausgänge

- **CNF**: Dieses Ereignis signalisiert die erfolgreiche Beendigung der Konvertierung und liefert das Ergebnis als `WSTRING`-Wert.

Daten-Eingänge

- **IN**: Der Eingang erwartet einen `REAL`-Wert, der in einen `WSTRING` umgewandelt werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgang liefert den konvertierten `WSTRING`-Wert nach Abschluss der Konvertierung.

Adapter

Dieser Funktionsblock verfügt über keine Adapter-Schnittstellen.

Funktionsweise

Der Funktionsblock `F_REAL_AS_WSTRING` führt bei einem eingehenden `REQ`-Ereignis die Konvertierung des `REAL`-Wertes am Eingang `IN` in einen `WSTRING`-Wert durch. Das Ergebnis wird am Ausgang `OUT` bereitgestellt und durch ein `CNF`-Ereignis bestätigt.

Der Konvertierungsalgorithmus ist als einfache ST (Structured Text)-Anweisung implementiert:

```
ALGORITHM REQ
OUT := REAL_AS_WSTRING(IN);
END_ALGORITHM
```

Technische Besonderheiten

- Der Funktionsblock ist Teil des Pakets `iec61131::conversion`.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Parameter oder Einstellmöglichkeiten.

Zustandsübersicht

1. **Idle**: Wartet auf ein `REQ`-Ereignis.
2. **Processing**: Führt die Konvertierung durch.
3. **Completed**: Sendet das `CNF`-Ereignis mit dem Ergebnis.

Anwendungsszenarien

- **Datenausgabe**: Konvertierung von numerischen Werten für die Anzeige auf einem Bedienpanel oder in einer Log-Datei.
- **Datenkommunikation**: Vorbereitung von numerischen Werten für die Übertragung als Zeichenketten in Netzwerkprotokollen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_REAL_TO_STRING**: Ähnlicher Funktionsblock, der jedoch in einen `STRING` statt in einen `WSTRING` konvertiert. `F_REAL_AS_WSTRING` ist für Unicode-fähige Umgebungen besser geeignet.
- **F_WSTRING_AS_REAL**: Führt die umgekehrte Konvertierung durch (von `WSTRING` zu `REAL`).

Fazit

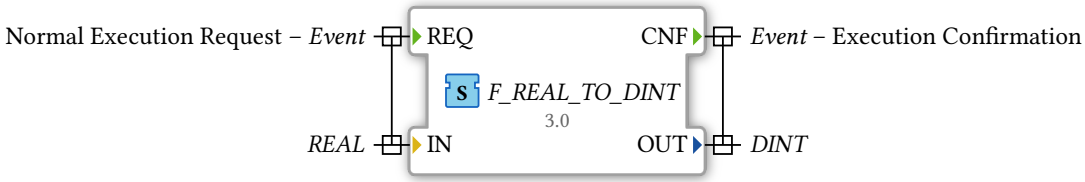
Der Funktionsblock `F_REAL_AS_WSTRING` bietet eine einfache und effiziente Möglichkeit, `REAL`-Werte in `WSTRING`-Werte umzuwandeln. Durch seine klare Schnittstelle und direkte Funktionsweise ist er ideal für Anwendungen, die eine Unicode-kompatible Zeichenkettenrepräsentation von Gleitkommazahlen benötigen.

F_REAL_TO_DINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_REAL_TO_DINT` dient der Konvertierung eines REAL-Wertes (Fließkommazahl) in einen DINT-Wert (32-Bit Ganzzahl). Diese Konvertierung erfolgt durch Abrundung des REAL-Wertes auf die nächstkleinere Ganzzahl.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Löst die Konvertierung aus.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Wird nach erfolgreicher Konvertierung ausgelöst.

Daten-Eingänge

- **IN**: Eingang für den REAL-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Ausgang für den konvertierten DINT-Wert.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Bei Auslösung des `REQ`-Ereignisses wird der Algorithmus ausgeführt, der den `IN`-Wert von REAL nach DINT konvertiert. Das Ergebnis wird am `OUT`-Ausgang ausgegeben und das `CNF`-Ereignis wird ausgelöst.

Technische Besonderheiten

- Die Konvertierung erfolgt durch Abrundung des REAL-Wertes.
- Der Funktionsblock ist einfach und hat keine internen Zustände.

Zustandsübersicht

Keine Zustände vorhanden, da der Block stateless ist.

Anwendungsszenarien

- Konvertierung von Sensorwerten (Fließkomma) in ganzzahlige Werte für Steuerungslogik.
- Datenverarbeitung in Steuerungssystemen, wo ganzzahlige Werte benötigt werden.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke könnten andere Konvertierungsmethoden anbieten, z.B. mit Rundung oder Aufrundung.
- Im Vergleich zu generischen Konvertierungsblöcken ist dieser spezialisiert auf REAL-zu-DINT Konvertierung.

Fazit

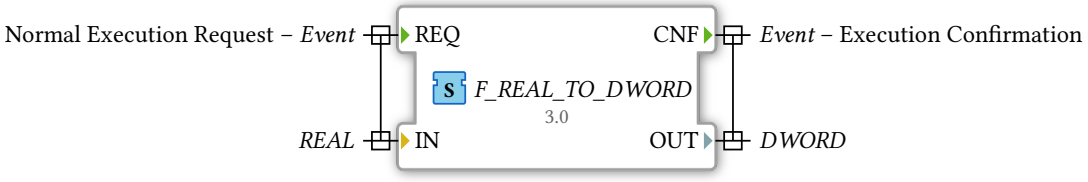
Der `F_REAL_TO_DINT` Funktionsblock ist ein einfacher und effizienter Baustein für die Konvertierung von Fließkommazahlen in Ganzzahlen. Er ist besonders nützlich in Szenarien, wo eine schnelle und direkte Konvertierung benötigt wird.

F_REAL_TO_DWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_REAL_TO_DWORD` dient der Konvertierung eines REAL-Wertes (Fließkommazahl) in einen DWORD-Wert (32-Bit vorzeichenlose Ganzzahl). Diese Konvertierung ist besonders nützlich, wenn Fließkommawerte in Systemen verarbeitet werden müssen, die nur Ganzzahloperationen unterstützen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der zu konvertierende Wert wird über den Daten-Eingang `IN` bereitgestellt.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der konvertierte Wert wird über den Daten-Ausgang `OUT` ausgegeben.

Daten-Eingänge

- **IN**: REAL-Wert, der in einen DWORD-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der resultierende DWORD-Wert nach der Konvertierung.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Empfang eines Ereignisses am Eingang `REQ` wird der Algorithmus ausgeführt, der den REAL-Wert von `IN` in einen DWORD-Wert konvertiert und das Ergebnis an `OUT` ausgibt. Anschließend wird das Ereignis `CNF` ausgelöst, um den Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Die Konvertierung erfolgt durch die Funktion `REAL_TO_DWORD`, die den REAL-Wert direkt in einen DWORD-Wert umwandelt. Dabei wird der Fließkommawert auf die nächstliegende Ganzzahl gerundet.
- Es ist zu beachten, dass bei sehr großen oder kleinen Werten Datenverlust auftreten kann, da der DWORD-Typ nur eine begrenzte Genauigkeit bietet.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Verarbeitung von Sensordaten, die als Fließkommawerte vorliegen, aber in Ganzzahlform weiterverarbeitet werden müssen.
- Integration in Steuerungssysteme, die nur Ganzzahloperationen unterstützen.
- Datenkonvertierung für Kommunikationsprotokolle, die DWORD-Werte erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_REAL_TO_INT**: Konvertiert REAL in INT (vorzeichenbehaftete Ganzzahl). Im Gegensatz dazu liefert `F_REAL_TO_DWORD` einen vorzeichenlosen Wert.
- **F_REAL_TO_LREAL**: Konvertiert REAL in LREAL (höhere Genauigkeit), während `F_REAL_TO_DWORD` den Wert in einen Ganzzahltyp umwandelt.

🛠️ Zugehörige Übungen

- SIN_TO_PCAN_Callback_DWord

Fazit

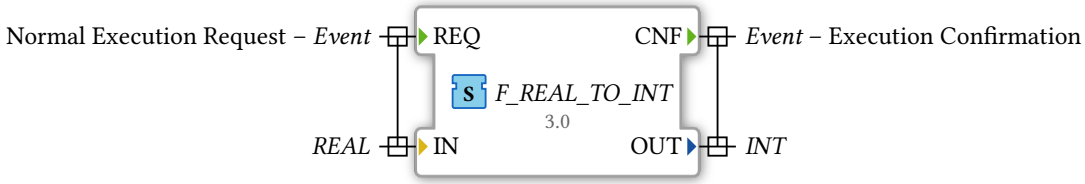
Der Funktionsblock `F_REAL_TO_DWORD` bietet eine einfache und effiziente Möglichkeit, Fließkommawerte in vorzeichenlose Ganzzahlen umzuwandeln. Er ist besonders nützlich in Umgebungen, die nur Ganzzahloperationen unterstützen oder wenn Daten in einem bestimmten Format weiterverarbeitet werden müssen. Bei der Verwendung ist jedoch auf mögliche Datenverluste durch die Konvertierung zu achten.

F_REAL_TO_INT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_REAL_TO_INT` dient der Konvertierung eines REAL-Wertes (Fließkommazahl) in einen INT-Wert (Ganzzahl). Diese Konvertierung ist besonders nützlich, wenn Daten zwischen Systemen ausgetauscht werden müssen, die unterschiedliche Datentypen verwenden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN**: Der Eingang für den REAL-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgang für den konvertierten INT-Wert.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem er den REAL-Wert am Eingang `IN` in einen INT-Wert umwandelt und das Ergebnis am Ausgang `OUT` ausgibt. Die Konvertierung wird durch das Ereignis `REQ` ausgelöst und durch das Ereignis `CNF` bestätigt.

Technische Besonderheiten

- Die Konvertierung erfolgt mittels der Funktion `REAL_TO_INT`, die den REAL-Wert in einen INT-Wert umwandelt.
- Der Funktionsblock ist einfach und effizient, da er keine komplexen Algorithmen oder Zustandsmaschinen verwendet.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung erfolgt unmittelbar nach dem Auslösen des `REQ`-Ereignisses.

Anwendungsszenarien

- **Datenkonvertierung**: Umwandlung von Sensorwerten (Fließkommazahlen) in Ganzzahlen für weitere Verarbeitung.
- **Schnittstellenkompatibilität**: Anpassung von Daten zwischen Systemen, die unterschiedliche Datentypen erwarten.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_TRUNC**: Konvertiert REAL zu INT durch Abschneiden der Nachkommastellen, während `F_REAL_TO_INT` eine standardmäßige Rundung durchführt.
- **F_ROUND**: Rundet REAL zu INT, ähnlich wie `F_REAL_TO_INT`, aber mit spezifischen Rundungsregeln.

Fazit

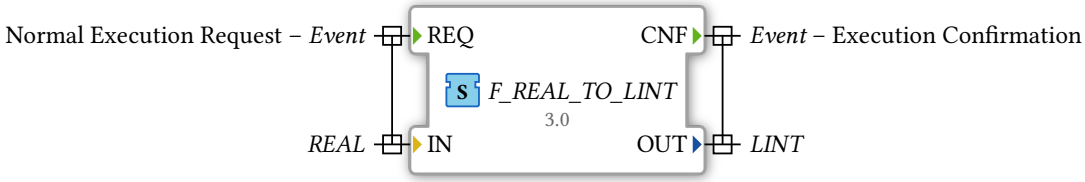
Der `F_REAL_TO_INT`-Funktionsblock ist ein einfaches und effektives Werkzeug zur Konvertierung von Fließkommazahlen in Ganzzahlen. Seine Einfachheit und direkte Funktionsweise machen ihn ideal für Anwendungen, bei denen eine schnelle und unkomplizierte Datentypumwandlung erforderlich ist.

F_REAL_TO_LINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_REAL_TO_LINT` dient der Konvertierung eines REAL-Wertes (Fließkommazahl) in einen LINT-Wert (64-Bit Ganzzahl). Diese Konvertierung ist besonders nützlich, wenn numerische Berechnungen mit unterschiedlichen Datentypen durchgeführt werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Ereigniseingang ausgelöst, wird der Wert am Daten-Eingang `IN` gelesen und konvertiert.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieses Ereignis wird zusammen mit dem konvertierten Wert am Daten-Ausgang `OUT` ausgegeben.

Daten-Eingänge

- **IN**: Der Eingang für den REAL-Wert, der in einen LINT-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgang, an dem der konvertierte LINT-Wert ausgegeben wird.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Bei Auslösung des `REQ`-Ereignisses wird der Algorithmus `REQ` ausgeführt. Dieser konvertiert den REAL-Wert am Eingang `IN` mittels der Funktion `REAL_TO_LINT` in einen LINT-Wert und schreibt das Ergebnis an den Ausgang `OUT`. Anschließend wird das `CNF`-Ereignis ausgelöst, um den erfolgreichen Abschluss der Konvertierung zu signalisieren.

Technische Besonderheiten

- Die Konvertierung von REAL zu LINT kann zu Datenverlust führen, wenn der REAL-Wert Nachkommastellen besitzt, da diese bei der Konvertierung abgeschnitten werden.
- Der Funktionsblock ist Teil des Pakets `iec61131::conversion`.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung erfolgt unmittelbar nach Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Konvertierung von Sensorwerten (z.B. Temperaturmessungen) für weitere Verarbeitung in Ganzzahl-Operationen.
- Integration in Steuerungssysteme, die gemischte Datentypen verarbeiten müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen (z.B. `F_REAL_TO_INT`) bietet `F_REAL_TO_LINT` eine höhere Präzision durch die Verwendung des 64-Bit LINT-Datentyps.
- Ähnliche Bausteine wie `F_LINT_TO_REAL` führen die umgekehrte Konvertierung durch.

Fazit

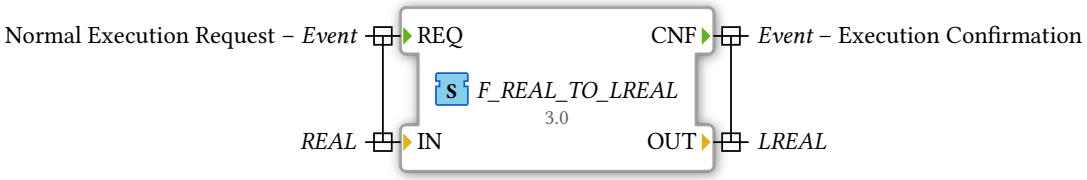
Der Funktionsblock `F_REAL_TO_LINT` ist ein einfacher und effizienter Baustein zur Konvertierung von Fließkommazahlen in 64-Bit Ganzzahlen. Seine Verwendung ist besonders in Szenarien mit hohen Präzisionsanforderungen zu empfehlen, wobei auf mögliche Datenverluste bei der Konvertierung zu achten ist.

F_REAL_TO_LREAL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_REAL_TO_LREAL` dient der Konvertierung eines REAL-Wertes in einen LREAL-Wert. Dieser Baustein ist Teil des `iec61131::conversion`-Pakets und ermöglicht eine einfache Typumwandlung zwischen den beiden numerischen Datentypen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Löst die Konvertierung aus. Wird mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Wird nach erfolgreicher Konvertierung ausgelöst. Wird mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Eingangswert vom Typ `REAL`, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Ausgangswert vom Typ `LREAL`, der das Ergebnis der Konvertierung enthält.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt bei Empfang des Ereignisses `REQ` die Konvertierung des REAL-Wertes `IN` in einen LREAL-Wert durch und weist das Ergebnis dem Ausgang `OUT` zu. Anschließend wird das Ereignis `CNF` ausgelöst, um die erfolgreiche Ausführung zu bestätigen.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher Baustein ohne interne Zustände oder komplexe Logik.
- Die Konvertierung erfolgt direkt durch Zuweisung, da `LREAL` eine höhere Genauigkeit als `REAL` bietet und somit alle Werte ohne Verlust darstellen kann.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge oder -verwaltung. Die Ausführung erfolgt atomar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo höhere Genauigkeit erforderlich ist.
- Integration in bestehende Systeme, die unterschiedliche Datentypen verwenden.
- Verwendung in mathematischen Berechnungen, die eine höhere Präzision erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Bausteine wie `F_INT_TO_REAL` oder `F_DINT_TO_LINT` führen ebenfalls Typumwandlungen durch, jedoch zwischen anderen Datentypen.
- Im Gegensatz zu komplexen Konvertierungsbausteinen bietet `F_REAL_TO_LREAL` eine einfache und direkte Umwandlung ohne zusätzliche Parameter oder Logik.

Fazit

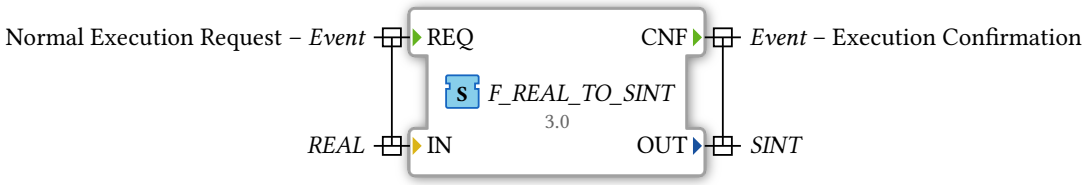
Der `F_REAL_TO_LREAL`-Funktionsblock ist ein nützliches Werkzeug für die einfache und effiziente Konvertierung von REAL- zu LREAL-Werten. Seine Einfachheit und direkte Funktionsweise machen ihn ideal für Anwendungen, die eine schnelle Typumwandlung ohne zusätzliche Komplexität erfordern.

F_REAL_TO_SINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_REAL_TO_SINT` dient der Konvertierung eines `REAL`-Wertes (Fließkommazahl) in einen `SINT`-Wert (8-Bit Ganzzahl mit Vorzeichen). Dieser Baustein ist besonders nützlich in Steuerungsanwendungen, bei denen eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Bei diesem Ereignis wird der Wert am Eingang `IN` verarbeitet.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieses Ereignis wird zusammen mit dem konvertierten Wert am Ausgang `OUT` ausgegeben.

Daten-Eingänge

- **IN**: Eingang für den `REAL`-Wert, der in einen `SINT`-Wert umgewandelt werden soll.

Daten-Ausgänge

- **OUT**: Ausgang für den konvertierten `SINT`-Wert.

Adapter

- Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem er den `REAL`-Wert am Eingang `IN` mittels der Funktion `REAL_TO_SINT` in einen `SINT`-Wert umwandelt. Die Konvertierung erfolgt bei Auftreten des Ereignisses `REQ`, und das Ergebnis wird mit dem Ereignis `CNF` ausgegeben.

Technische Besonderheiten

- Der Baustein ist als einfacher Funktionsblock (`SimpleFB`) implementiert, der nur einen Algorithmus enthält.
- Die Konvertierung schneidet den `REAL`-Wert auf den Wertebereich von `SINT` (-128 bis 127) zu. Werte außerhalb dieses Bereichs führen zu einem Überlauf.

Zustandsübersicht

Der Funktionsblock hat keinen internen Zustand. Die Konvertierung erfolgt unmittelbar bei Auftreten des `REQ`-Ereignisses.

Anwendungsszenarien

- Typumwandlung in Steuerungsalgorithmen, wo Fließkommawerte in Ganzzahlen umgewandelt werden müssen.
- Signalverarbeitung, bei der `REAL`-Sensordaten auf `SINT` reduziert werden müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Bausteine wie `F_REAL_TO_INT` oder `F_REAL_TO_USINT` bieten Konvertierungen in andere Ganzzahltypen. `F_REAL_TO_SINT` ist spezifisch für 8-Bit Ganzzahlen mit Vorzeichen.

Fazit

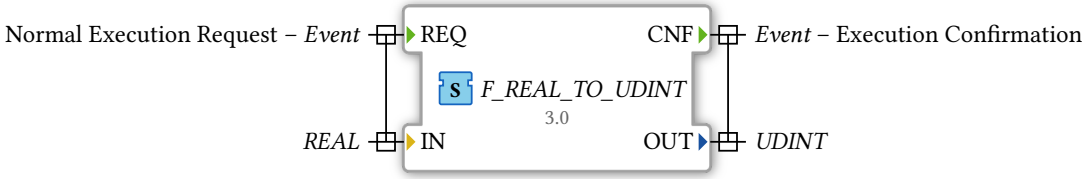
Der `F_REAL_TO_SINT`-Funktionsblock ist ein einfacher und effizienter Baustein für die Konvertierung von Fließkommazahlen in 8-Bit Ganzzahlen mit Vorzeichen. Seine Verwendung ist in Szenarien sinnvoll, wo eine solche Typumwandlung erforderlich ist, insbesondere in Steuerungs- und Signalverarbeitungsanwendungen.

F_REAL_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_REAL_TO_UDINT` dient der Konvertierung eines REAL-Wertes in einen UDINT-Wert. Dieser Baustein ist Teil des `iec61131::conversion`-Pakets und implementiert eine einfache, aber wichtige Typumwandlung für die Verwendung in Steuerungsanwendungen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung der Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (REAL): Der Eingangswert, der von REAL in UDINT konvertiert werden soll.

Daten-Ausgänge

- **OUT** (UDINT): Der Ergebniswert der Konvertierung.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Empfang eines Ereignisses am Eingang `REQ` wird der Algorithmus ausgeführt, der den REAL-Wert am Eingang `IN` in einen UDINT-Wert konvertiert und das Ergebnis am Ausgang `OUT` ausgibt. Anschließend wird das Ereignis `CNF` ausgelöst, um den erfolgreichen Abschluss der Konvertierung zu signalisieren.

Technische Besonderheiten

- Die Konvertierung erfolgt mittels der eingebauten Funktion `REAL_TO_UDINT`.
- Der Funktionsblock ist deterministisch und hat keine internen Zustände.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge oder -maschinen. Der Block führt die Konvertierung bei jedem `REQ`-Ereignis aus und signalisiert den Abschluss mit `CNF`.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo REAL-Werte als UDINT verarbeitet werden müssen.
- Integration in größere Funktionsblocknetzwerke, die unterschiedliche Datentypen verarbeiten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Konvertierungsblöcke wie `F_REAL_TO_INT` oder `F_REAL_TO_LREAL` bieten Konvertierungen in andere Datentypen, aber nicht spezifisch in UDINT.
- Im Gegensatz zu komplexeren Blöcken bietet `F_REAL_TO_UDINT` eine einfache und direkte Lösung für diese spezifische Konvertierung.

🛠️ Zugehörige Übungen

- Uebung_072c

Fazit

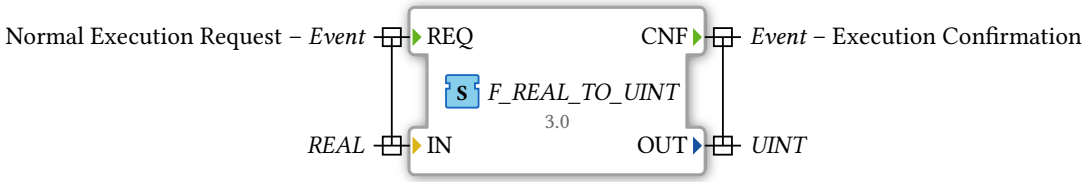
Der `F_REAL_TO_UDINT`-Funktionsblock ist ein einfacher, aber essenzieller Baustein für die Typumwandlung in IEC 61499-basierten Steuerungsanwendungen. Seine deterministische und zustandslose Natur macht ihn zuverlässig und leicht integrierbar in verschiedene Anwendungen.

F_REAL_TO_UINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_REAL_TO_UINT` dient der Konvertierung eines REAL-Wertes (Fließkommazahl) in einen UINT-Wert (vorzeichenlose Ganzzahl). Diese Konvertierung ist besonders nützlich, wenn Daten zwischen Systemen ausgetauscht werden, die unterschiedliche Datentypen verwenden oder wenn eine explizite Typumwandlung erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Ereigniseingang ausgelöst, wird der Algorithmus zur Umwandlung des REAL-Wertes in einen UINT-Wert ausgeführt.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieses Ereignis wird ausgelöst, sobald der Algorithmus erfolgreich durchlaufen wurde.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ REAL, der in einen UINT-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgangswert vom Typ UINT, der das Ergebnis der Konvertierung darstellt.

Adapter

Dieser Funktionsblock verfügt über keine Adapter-Schnittstellen.

Funktionsweise

Der Funktionsblock `F_REAL_TO_UINT` führt bei Auslösung des `REQ`-Ereignisses die Konvertierung des REAL-Wertes am Eingang `IN` in einen UINT-Wert durch. Das Ergebnis wird am Ausgang `OUT` ausgegeben und das `CNF`-Ereignis signalisiert den erfolgreichen Abschluss der Operation.

Der zugrundeliegende Algorithmus lautet:

```
ALGORITHM REQ
OUT := REAL_TO_UINT(IN);
END_ALGORITHM
```

Technische Besonderheiten

- Die Konvertierung erfolgt durch die Standardfunktion `REAL_TO_UINT`, die den REAL-Wert in den nächstgelegenen UINT-Wert rundet.
- Es ist zu beachten, dass bei Werten außerhalb des darstellbaren Bereichs von UINT (0 bis 65535) undefiniertes Verhalten auftreten kann.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo Fließkommawerte in Ganzzahlen umgewandelt werden müssen.
- Datenaufbereitung für Systeme, die nur vorzeichenlose Ganzzahlen verarbeiten können.
- Schnittstellen zwischen unterschiedlichen Systemkomponenten mit verschiedenen Datentypen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen wie `F_REAL_TO_INT` oder `F_REAL_TO_UDINT` ist dieser Funktionsblock spezialisiert auf die Umwandlung in vorzeichenlose 16-Bit-Ganzzahlen (UINT).
- Ähnliche Bausteine könnten zusätzliche Parameter wie Rundungsmodi oder Bereichsprüfungen bieten, was hier nicht der Fall ist.

Fazit

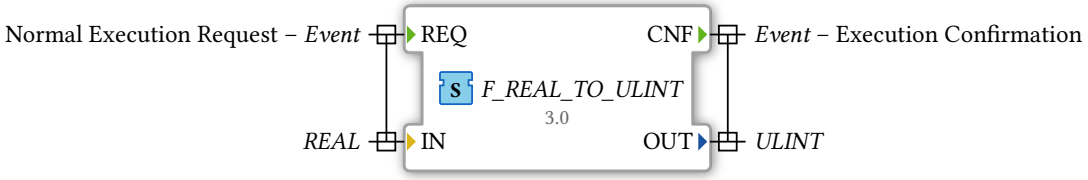
Der `F_REAL_TO_UINT` Funktionsblock bietet eine einfache und effiziente Möglichkeit, Fließkommazahlen in vorzeichenlose Ganzzahlen umzuwandeln. Seine klare Schnittstellenstruktur und direkte Funktionsweise machen ihn ideal für Anwendungen, bei denen eine schnelle und unkomplizierte Typumwandlung benötigt wird. Bei der Verwendung ist jedoch auf den gültigen Wertebereich zu achten, um unerwartetes Verhalten zu vermeiden.

F_REAL_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_REAL_TO_ULINT` dient der Konvertierung eines `REAL`-Wertes (Fließkommazahl) in einen `ULINT`-Wert (64-Bit vorzeichenlose Ganzzahl). Dieser Baustein ist besonders nützlich in Szenarien, wo eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Eingang für den `REAL`-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Ausgang für den konvertierten `ULINT`-Wert.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` ausgelöst wird. Dabei wird der `REAL`-Wert am Eingang `IN` in einen `ULINT`-Wert umgewandelt und am Ausgang `OUT` ausgegeben. Die Konvertierung wird mit dem Ereignis `CNF` bestätigt.

Technische Besonderheiten

- Die Konvertierung erfolgt mittels der Funktion `REAL_TO_ULINT`.
- Der Baustein ist Teil des Pakets `iec61131::conversion`.

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf das `REQ`-Ereignis.
2. **Konvertierungszustand**: Führt die Umwandlung durch und gibt das Ergebnis aus.
3. **Bestätigungszustand**: Sendet das `CNF`-Ereignis zur Bestätigung der Konvertierung.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo Fließkommazahlen in vorzeichenlose Ganzzahlen umgewandelt werden müssen.
- Datenverarbeitung in industriellen Automatisierungssystemen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Bausteine wie `F_REAL_TO_INT` oder `F_REAL_TO_LINT` führen Konvertierungen in andere Ganzzahltypen durch. `F_REAL_TO_ULINT` ist spezifisch für 64-Bit vorzeichenlose Ganzzahlen.

Fazit

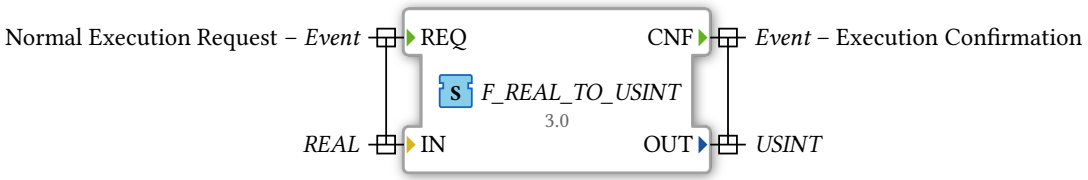
Der `F_REAL_TO_ULINT`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, `REAL`-Werte in `ULINT`-Werte umzuwandeln. Seine klare Schnittstellenstruktur und einfache Funktionsweise machen ihn zu einem wertvollen Baustein in der industriellen Automatisierung.

F_REAL_TO_USINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_REAL_TO_USINT` dient der Konvertierung eines REAL-Wertes (Fließkommazahl) in einen USINT-Wert (vorzeichenlose 8-Bit Ganzzahl). Diese Konvertierung ist besonders nützlich, wenn Fließkommazahlen in einem Bereich verarbeitet werden müssen, der nur ganzzahlige Werte erfordert.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Eingang ausgelöst, wird der Wert am Daten-Eingang `IN` verarbeitet.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Wird ausgelöst, sobald der konvertierte Wert am Daten-Ausgang `OUT` verfügbar ist.

Daten-Eingänge

- **IN**: REAL-Wert, der in einen USINT-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der konvertierte USINT-Wert.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Der Funktionsblock führt eine direkte Konvertierung des REAL-Wertes am Eingang `IN` in einen USINT-Wert durch. Die Konvertierung erfolgt durch den Aufruf der internen Funktion `REAL_TO_USINT`. Sobald die Konvertierung abgeschlossen ist, wird das Ereignis `CNF` ausgelöst, und der konvertierte Wert steht am Ausgang `OUT` zur Verfügung.

Technische Besonderheiten

- Die Konvertierung schneidet den Nachkommaanteil ab (keine Rundung).
- Der Eingangswert sollte im Bereich von 0 bis 255 liegen, da USINT nur Werte in diesem Bereich darstellen kann. Werte außerhalb dieses Bereichs führen zu undefiniertem Verhalten oder Überlauf.

Zustandsübersicht

1. **Idle**: Wartet auf das Ereignis `REQ`.
2. **Processing**: Führt die Konvertierung durch.
3. **Completed**: Sendet das Ereignis `CNF` und stellt den konvertierten Wert bereit.

Anwendungsszenarien

- Konvertierung von Sensorwerten (z.B. Temperatur, Druck) in ganzzahlige Werte für die weitere Verarbeitung.
- Vorverarbeitung von Daten für Systeme, die nur Ganzzahlen verarbeiten können.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_REAL_TO_INT**: Konvertiert REAL in INT (vorzeichenbehaftete 16-Bit Ganzahl).
- **F_REAL_TO_UINT**: Konvertiert REAL in UINT (vorzeichenlose 16-Bit Ganzahl).
- **F_REAL_TO_ULINT**: Konvertiert REAL in ULINT (vorzeichenlose 64-Bit Ganzahl).

Fazit

Der Funktionsblock `F_REAL_TO_USINT` ist ein einfaches und effizientes Werkzeug zur Konvertierung von Fließkommazahlen in vorzeichenlose 8-Bit Ganzzahlen. Er eignet sich besonders für Anwendungen, bei denen Speicherplatz oder Verarbeitungsressourcen begrenzt sind.

REAL Konvertierungen

🎧 Podcast

- IEC 61499: Fabrikautomation im Umbruch – Von der Vision zur industriellen Realität?
- From „Get Away with Electronics“ to Essential: How Agricultural Tech Earned Trust and Delivered Real Relief

🛠️ Zugehörige Übungen
- Uebung_072c
- Uebung_151_AX
- Uebung_152
- Uebung_153

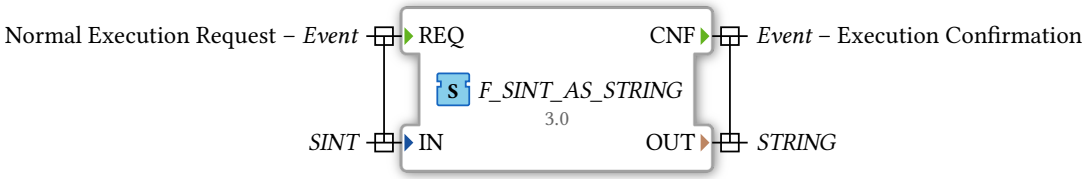
SINT

F_SINT_AS_STRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_SINT_AS_STRING` dient der Konvertierung eines `SINT`-Wertes (8-Bit Ganzzahl) in einen `STRING` (Zeichenkette). Dieser Baustein ist besonders nützlich, wenn numerische Werte für die Anzeige oder Weiterverarbeitung als Text benötigt werden.



Schnittstellenstruktur

Ereignis-Eingänge
- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge
- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge
- **IN**: Der zu konvertierende `SINT`-Wert (8-Bit Ganzzahl).

Daten-Ausgänge
- **OUT**: Der konvertierte `STRING`-Wert.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das `REQ`-Ereignis eintrifft. Der `SINT`-Wert am Eingang `IN` wird in einen `STRING` umgewandelt und am Ausgang `OUT` ausgegeben. Die Konvertierung erfolgt mittels der Funktion `SINT_AS_STRING`. Nach Abschluss der Konvertierung wird das `CNF`-Ereignis ausgelöst.

Technische Besonderheiten
- Die Konvertierung erfolgt direkt und ohne zusätzliche Parameter.
- Der Funktionsblock ist einfach und effizient, da er nur eine grundlegende Datentypumwandlung durchführt.

Zustandsübersicht
1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Konvertierung**: Führt die Umwandlung durch und gibt das Ergebnis aus.
3. **Bestätigung**: Löst das `CNF`-Ereignis aus und kehrt in den Idle-Zustand zurück.

Anwendungsszenarien
- Anzeige von numerischen Werten in Benutzeroberflächen.

- Protokollierung von numerischen Daten als Text.
- Vorbereitung von Daten für die Kommunikation über textbasierte Protokolle.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_INT_AS_STRING**: Konvertiert einen `INT`-Wert (16-Bit) in einen `STRING`.
- **F_DINT_AS_STRING**: Konvertiert einen `DINT`-Wert (32-Bit) in einen `STRING`.
- **F_LINT_AS_STRING**: Konvertiert einen `LINT`-Wert (64-Bit) in einen `STRING`.

Im Vergleich zu diesen Bausteinen ist `F_SINT_AS_STRING` spezialisiert auf die Konvertierung von 8-Bit Ganzzahlen.

Fazit

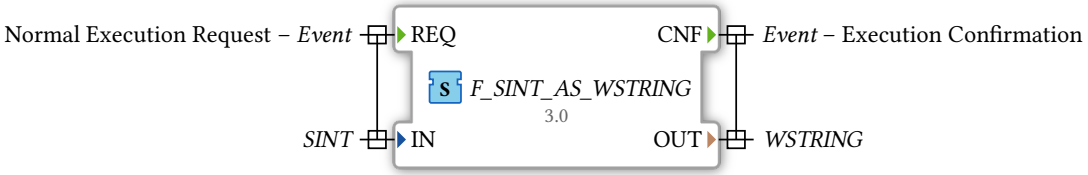
Der `F_SINT_AS_STRING`-Funktionsblock ist ein einfaches und effizientes Werkzeug zur Konvertierung von `SINT`-Werten in `STRING`s. Seine klare Schnittstelle und direkte Funktionsweise machen ihn ideal für Anwendungen, bei denen kleine Ganzzahlen als Text dargestellt werden müssen.

F_SINT_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_SINT_AS_WSTRING` dient zur Konvertierung eines SINT-Wertes (8-Bit Ganzzahl mit Vorzeichen) in einen WSTRING (Wide-String). Dieser Baustein ist besonders nützlich, wenn numerische Werte in eine für die Anzeige oder Weiterverarbeitung geeignete Textform umgewandelt werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN**: Der SINT-Wert (8-Bit Ganzzahl mit Vorzeichen), der in einen WSTRING konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der resultierende WSTRING nach der Konvertierung des SINT-Wertes.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung des SINT-Wertes (`IN`) in einen WSTRING (`OUT`) durch, sobald das Ereignis `REQ` ausgelöst wird. Die Konvertierung erfolgt mittels der integrierten Funktion `SINT_AS_WSTRING`. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Der Funktionsblock ist einfach aufgebaut und enthält nur einen Algorithmus zur Konvertierung.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Parameter oder komplexe Logik.

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf das `REQ`-Ereignis.

2. **Konvertierungszustand**: Führt die Konvertierung durch und gibt das Ergebnis über `OUT` aus.
3. **Bestätigungszustand**: Löst das `CNF`-Ereignis aus und kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- Anzeige von numerischen Werten in Benutzeroberflächen.
- Protokollierung von numerischen Daten in Textform.
- Weiterverarbeitung von numerischen Werten in Text-basierten Systemen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_INT_AS_WSTRING**: Konvertiert einen INT-Wert (16-Bit) in einen WSTRING.
- **F_DINT_AS_WSTRING**: Konvertiert einen DINT-Wert (32-Bit) in einen WSTRING.
- **F_LINT_AS_WSTRING**: Konvertiert einen LINT-Wert (64-Bit) in einen WSTRING.

Fazit

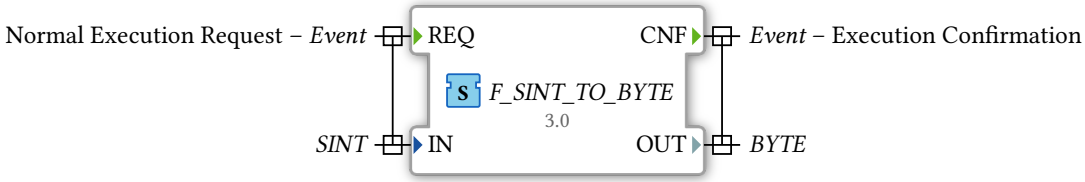
Der `F_SINT_AS_WSTRING` Funktionsblock bietet eine einfache und effiziente Möglichkeit, SINT-Werte in WSTRINGs zu konvertieren. Seine klare Schnittstelle und einfache Funktionsweise machen ihn zu einem nützlichen Baustein in verschiedenen Anwendungen, insbesondere dort, wo numerische Werte in Textform dargestellt oder verarbeitet werden müssen.

F_SINT_TO_BYTE

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_SINT_TO_BYTE` dient der Konvertierung eines vorzeichenbehafteten 8-Bit-Integer-Werts (SINT) in einen vorzeichenlosen 8-Bit-Integer-Wert (BYTE). Diese Konvertierung ist nützlich, wenn Daten zwischen Systemen ausgetauscht werden, die unterschiedliche Darstellungen für 8-Bit-Werte verwenden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Eingang ausgelöst, wird der Wert am Daten-Eingang `IN` gelesen und konvertiert.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der konvertierte Wert steht am Daten-Ausgang `OUT` zur Verfügung.

Daten-Eingänge

- **IN** (SINT): Der vorzeichenbehaftete 8-Bit-Integer-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (BYTE): Der vorzeichenlose 8-Bit-Integer-Wert, der das Ergebnis der Konvertierung darstellt.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` ausgelöst wird. Dabei wird der Wert am Eingang `IN` vom Typ SINT in einen BYTE-Wert umgewandelt und am Ausgang `OUT` ausgegeben. Das Ereignis `CNF` signalisiert die erfolgreiche Beendigung der Konvertierung.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt durch die Funktion `SINT_TO_BYTE`.
- Der Block ist einfach und deterministisch, ohne interne Zustände oder komplexe Logik.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar nach Auslösen von `REQ`, gefolgt von der Ausgabe des Ergebnisses und der Bestätigung durch `CNF`.

Anwendungsszenarien

- Kommunikation zwischen Systemen, die vorzeichenbehaftete und vorzeichenlose 8-Bit-Werte verwenden.
- Datenverarbeitung in Steuerungssystemen, wo eine Typumwandlung erforderlich ist.
- Integration in größere Funktionsblocknetzwerke, die unterschiedliche Datentypen verarbeiten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken ist `F_SINT_TO_BYTE` spezialisiert auf die Umwandlung von SINT zu BYTE und daher effizienter für diesen spezifischen Anwendungsfall.
- Andere Blöcke wie `F_INT_TO_BYTE` oder `F_DINT_TO_BYTE` sind für größere Integer-Typen ausgelegt.

Fazit

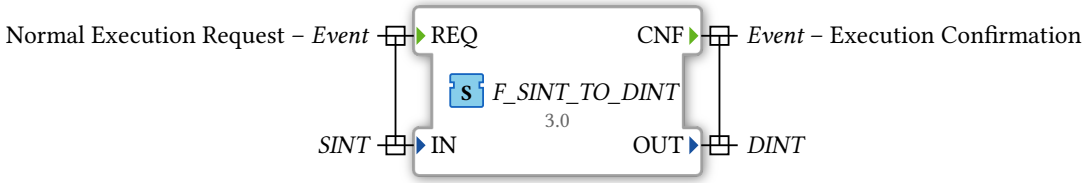
Der `F_SINT_TO_BYTE` Funktionsblock ist ein einfaches, aber effektives Werkzeug zur Konvertierung zwischen vorzeichenbehafteten und vorzeichenlosen 8-Bit-Integern. Seine Einfachheit und Determinismus machen ihn zuverlässig für Anwendungen, die solche Typumwandlungen erfordern.

F_SINT_TO_DINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_SINT_TO_DINT` dient der Konvertierung eines `SINT`-Wertes (8-Bit Ganzahl mit Vorzeichen) in einen `DINT`-Wert (32-Bit Ganzahl mit Vorzeichen). Dieser Baustein ist besonders nützlich in Szenarien, wo eine Typumwandlung erforderlich ist, um Daten für weitere Verarbeitungsschritte vorzubereiten.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `SINT`, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der konvertierte Ausgangswert vom Typ `DINT`.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Bei Auslösung des Ereigniseingangs `REQ` wird der Wert am Daten-Eingang `IN` gelesen und in den Typ `DINT` konvertiert. Das Ergebnis wird am Daten-Ausgang `OUT` ausgegeben und das Ereignis `CNF` signalisiert den erfolgreichen Abschluss der Konvertierung.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt durch Zuweisung (`OUT := IN`), da `SINT` automatisch in `DINT` umgewandelt wird.
- Der Baustein ist einfach und effizient, da er keine komplexen Berechnungen durchführt.

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf das Ereignis `REQ`.
2. **Konvertierungszustand**: Bei Empfang von `REQ` wird die Konvertierung durchgeführt und `CNF` ausgelöst.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo `SINT`-Daten in `DINT` umgewandelt werden müssen.
- Vorbereitung von Daten für weitere Verarbeitungsschritte, die `DINT` erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Bausteine wie `F_INT_TO_DINT` oder `F_USINT_TO_UDINT` führen ähnliche Konvertierungen durch, aber mit anderen Ausgangs- bzw. Eingangstypen.
- `F_SINT_TO_DINT` ist spezifisch für die Konvertierung von `SINT` zu `DINT`.

Fazit

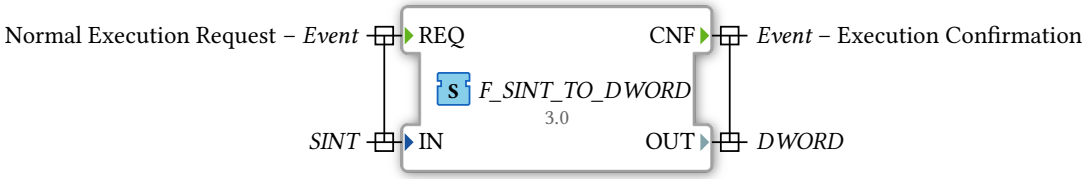
Der `F_SINT_TO_DINT` Funktionsblock ist ein einfacher und effizienter Baustein für die Typumwandlung von `SINT` zu `DINT`. Seine klare Schnittstellenstruktur und einfache Funktionsweise machen ihn zu einer zuverlässigen Komponente in Steuerungsanwendungen.

F_SINT_TO_DWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_SINT_TO_DWORD` dient der Konvertierung eines vorzeichenbehafteten 8-Bit-Integer-Werts (SINT) in einen vorzeichenlosen 32-Bit-Integer-Wert (DWORD). Dieser Baustein ist besonders nützlich in Szenarien, wo eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Dieser Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `SINT`, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgangswert vom Typ `DWORD`, der das Ergebnis der Konvertierung enthält.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` ausgelöst wird. Der Algorithmus wandelt den `SINT`-Wert am Eingang `IN` in einen `DWORD`-Wert um und legt das Ergebnis am Ausgang `OUT` ab. Anschließend wird das Ereignis `CNF` ausgelöst, um den erfolgreichen Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt durch die Funktion `SINT_TO_DWORD(IN)`.
- Der Baustein ist als einfacher Funktionsblock (SimpleFB) implementiert, was eine schnelle und effiziente Ausführung ermöglicht.

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf das `REQ`-Ereignis.
2. **Konvertierungszustand**: Führt die Konvertierung durch und setzt den Ausgang `OUT`.
3. **Bestätigungszustand**: Löst das `CNF`-Ereignis aus und kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo `SINT`-Werte in `DWORD`-Werte umgewandelt werden müssen.
- Integration in größere Steuerungssysteme, die unterschiedliche Datentypen verarbeiten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_INT_TO_DINT` oder `F_BYTE_TO_WORD` bieten Konvertierungen zwischen anderen Datentypen, jedoch nicht speziell zwischen `SINT` und `DWORD`.
- Im Gegensatz zu generischen Konvertierungsblöcken ist `F_SINT_TO_DWORD` spezialisiert und daher effizienter für diesen spezifischen Anwendungsfall.

Fazit

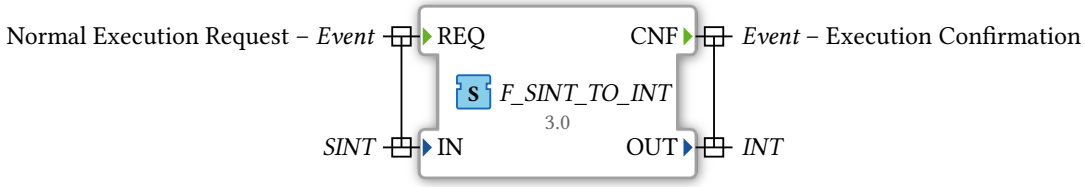
Der `F_SINT_TO_DWORD`-Funktionsblock ist ein spezialisiertes und effizientes Werkzeug zur Konvertierung von `SINT`- in `DWORD`-Werte. Seine einfache Schnittstelle und direkte Funktionsweise machen ihn ideal für Anwendungen, die eine schnelle und zuverlässige Typumwandlung erfordern.

F_SINT_TO_INT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_SINT_TO_INT` dient zur Konvertierung eines `SINT`-Wertes (8-Bit Ganzzahl mit Vorzeichen) in einen `INT`-Wert (16-Bit Ganzahl mit Vorzeichen). Dieser Baustein ist besonders nützlich in Szenarien, wo eine Typumwandlung zwischen unterschiedlichen Integer-Formaten erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung der Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (`SINT`): Der Eingangswert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (`INT`): Der konvertierte Ausgangswert.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt eine direkte Zuweisung des `SINT`-Eingabewertes an den `INT`-Ausgang durch. Da `SINT` ein kleinerer Datentyp als `INT` ist, erfolgt die Konvertierung ohne Datenverlust.

Der Algorithmus ist wie folgt definiert:

```
ALGORITHM REQ
OUT := IN;
END_ALGORITHM
```

Technische Besonderheiten

- Die Konvertierung ist verlustfrei, da der Wertebereich von `SINT` (-128 bis 127) vollständig in den Wertebereich von `INT` (-32768 bis 32767) passt.
- Der Funktionsblock ist deterministisch und hat keine internen Zustände.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Ausführung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo verschiedene Integer-Formate verarbeitet werden müssen.
- Integration in größere Funktionsblock-Netzwerke, wo eine einheitliche Datentypverarbeitung erforderlich ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_INT_TO_SINT` führen die umgekehrte Konvertierung durch, jedoch mit potenziellem Datenverlust.
- Im Gegensatz zu generischen Konvertierungsblöcken ist `F_SINT_TO_INT` spezialisiert und optimiert für diese spezifische Umwandlung.

Fazit

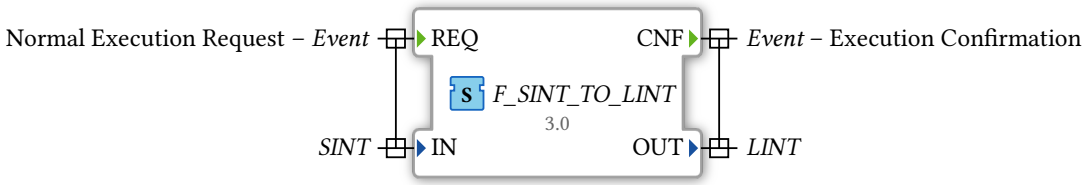
Der `F_SINT_TO_INT` Funktionsblock bietet eine einfache und effiziente Möglichkeit, `SINT`-Werte in `INT`-Werte umzuwandeln. Seine deterministische und verlustfreie Funktionsweise macht ihn zu einem verlässlichen Baustein in Steuerungsanwendungen.

F_SINT_TO_LINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_SINT_TO_LINT` dient der Konvertierung eines `SINT`-Wertes (8-Bit Ganzzahl) in einen `LINT`-Wert (64-Bit Ganzahl). Diese Konvertierung ist nützlich, wenn Daten mit unterschiedlicher Bitbreite verarbeitet werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge
- `REQ`: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge
- `CNF`: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge
- `IN` (`SINT`): Der zu konvertierende 8-Bit Ganzzahlwert.

Daten-Ausgänge
- `OUT` (`LINT`): Der konvertierte 64-Bit Ganzzahlwert.

Adapter
Keine Adapter vorhanden.

Funktionsweise
Der Funktionsblock führt eine direkte Zuweisung des Eingabewertes `IN` zum Ausgabewert `OUT` durch. Da `LINT` eine größere Bitbreite als `SINT` hat, erfolgt die Konvertierung ohne Datenverlust.

Technische Besonderheiten
- Die Konvertierung ist verlustfrei, da der `LINT`-Datentyp alle möglichen Werte des `SINT`-Datentyps darstellen kann.
- Der Funktionsblock ist deterministisch und hat keine internen Zustände.

Zustandsübersicht
Der Funktionsblock hat keine internen Zustände. Die Ausführung erfolgt sofort bei Empfang des `REQ`-Ereignisses.

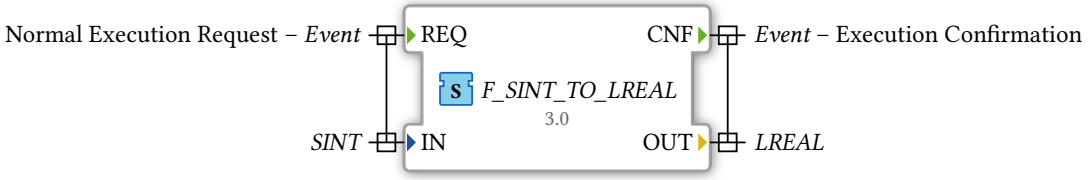
Anwendungsszenarien
- Verarbeitung von 8-Bit Sensordaten in 64-Bit Systemen
- Kompatibilität zwischen Systemen mit unterschiedlicher Bitbreite
- Datenkonvertierung in Automatisierungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen
- Im Vergleich zu anderen Konvertierungsblöcken (z.B. `F_INT_TO_LINT`) arbeitet dieser Block speziell mit 8-Bit Eingabedaten.
- Ähnlich wie `F_BYTE_TO_LWORD`, aber für vorzeichenbehaftete Ganzzahlen.

Fazit
Der `F_SINT_TO_LINT` Funktionsblock bietet eine einfache und effiziente Möglichkeit, 8-Bit Ganzzahlen in 64-Bit Ganzzahlen zu konvertieren. Seine einfache Struktur und deterministische Arbeitsweise machen ihn zu einem zuverlässigen Baustein in Automatisierungslösungen.

F_SINT_TO_LREAL
• − ◦ • − ◦ • − ◦ •

Einleitung
Der Funktionsblock `F_SINT_TO_LREAL` dient der Konvertierung eines ganzzahligen Wertes vom Typ `SINT` (8-Bit Ganzzahl) in einen Fließkommawert vom Typ `LREAL` (64-Bit Fließkomma). Dieser Baustein ist besonders nützlich in Anwendungen, bei denen eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Eingang für den zu konvertierenden `SINT`-Wert.

Daten-Ausgänge

- **OUT**: Ausgang für den konvertierten `LREAL`-Wert.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt eine direkte Typumwandlung durch. Bei Eintreffen eines `REQ`-Ereignisses wird der Wert von `IN` gelesen, in den Typ `LREAL` konvertiert und an `OUT` ausgegeben. Anschließend wird das `CNF`-Ereignis ausgelöst.

Technische Besonderheiten

- Die Konvertierung erfolgt ohne zusätzliche Skalierung oder Rundung.
- Der Baustein ist einfach und effizient implementiert, da er nur eine Zuweisung benötigt.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Er reagiert ausschließlich auf das `REQ`-Ereignis mit einer sofortigen Konvertierung und Ausgabe.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo `SINT`-Werte in Berechnungen mit `LREAL`-Werten verwendet werden müssen.
- Datenaufbereitung für Visualisierung oder Protokollierung, wo höhere Genauigkeit erforderlich ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Bausteine wie `F_INT_TO_REAL` oder `F_DINT_TO_LREAL` bieten Konvertierungen zwischen anderen Datentypen, aber nicht spezifisch von `SINT` zu `LREAL`.
- Dieser Baustein ist spezialisiert auf die Konvertierung von 8-Bit Ganzzahlen zu 64-Bit Fließkommazahlen.

Fazit

Der `F_SINT_TO_LREAL`-Funktionsblock ist ein einfaches und effizientes Werkzeug für die Typumwandlung in 4diac-IDE. Seine klare Schnittstelle und direkte Funktionsweise machen ihn zu einer zuverlässigen Komponente in Steuerungsanwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

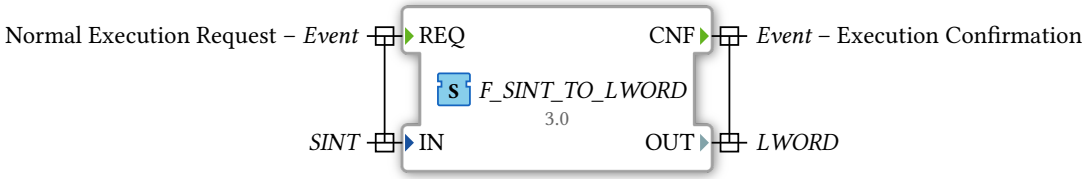
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

F_SINT_TO_LWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_SINT_TO_LWORD` dient der Konvertierung eines vorzeichenbehafteten 8-Bit-Integer-Werts (`SINT`) in einen vorzeichenlosen 64-Bit-Integer-Wert (`LWORD`). Dieser Baustein ist Teil des `iec61131::conversion`-Pakets und wird typischerweise in Automatisierungsanwendungen eingesetzt, wo Datentypkonvertierungen erforderlich sind.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Normal Execution Request): Startet die Ausführung des Funktionsblocks. Wird mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF** (Execution Confirmation): Signalisiert den Abschluss der Konvertierung. Wird mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (`SINT`): Der vorzeichenbehaftete 8-Bit-Integer-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (`LWORD`): Der resultierende vorzeichenlose 64-Bit-Integer-Wert nach der Konvertierung.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `SINT` zu `LWORD` durch, sobald das Ereignis `REQ` ausgelöst wird. Die Konvertierung erfolgt mittels der integrierten Funktion `SINT_TO_LWORD`. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst, und der konvertierte Wert steht am Ausgang `OUT` zur Verfügung.

Technische Besonderheiten

- Die Konvertierung erfolgt ohne zusätzliche Parameter oder Konfiguration.
- Der Baustein ist deterministisch und hat keine internen Zustände.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge oder interne Zustände. Die Ausführung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Typkonvertierung in Steuerungsanwendungen, wo unterschiedliche Datentypen verarbeitet werden müssen.
- Integration in größere Funktionsblocknetzwerke, wo Daten von einem niedrigeren zu einem höheren Präzisionsformat konvertiert werden müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_INT_TO_LWORD` oder `F_DINT_TO_LWORD` führen Konvertierungen von anderen Integer-Typen zu `LWORD` durch. `F_SINT_TO_LWORD` ist spezifisch für die Konvertierung von 8-Bit-Integern.

- Im Gegensatz zu generischen Konvertierungsblöcken bietet dieser Baustein eine typsichere und optimierte Lösung für die spezifische Konvertierung.

Fazit

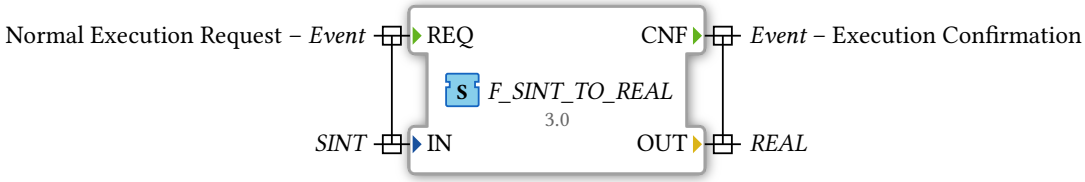
Der `F_SINT_TO_LWORD`-Funktionsblock bietet eine effiziente und einfache Möglichkeit, `SINT`-Werte in `LWORD`-Werte zu konvertieren. Durch seine deterministische Arbeitsweise und klare Schnittstellen eignet er sich ideal für den Einsatz in Automatisierungsprojekten, wo solche Typkonvertierungen regelmäßig benötigt werden.

F_SINT_TO_REAL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_SINT_TO_REAL` dient der Konvertierung eines vorzeichenbehafteten 8-Bit-Integer-Werts (`SINT`) in einen Fließkommawert (`REAL`). Diese Konvertierung ist insbesondere in Steuerungsanwendungen nützlich, wo unterschiedliche Datentypen verarbeitet werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (`SINT`): Der vorzeichenbehaftete 8-Bit-Integer-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (`REAL`): Der resultierende Fließkommawert nach der Konvertierung.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt eine direkte Typumwandlung durch. Bei Eintreffen eines Ereignisses am Eingang `REQ` wird der Wert von `IN` in den Datentyp `REAL` konvertiert und an `OUT` ausgegeben. Anschließend wird das Ereignis `CNF` ausgelöst, um den erfolgreichen Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- **Einfache Konvertierung**: Die Umwandlung erfolgt ohne zusätzliche Berechnungen oder Skalierungen.
- **Keine Fehlerbehandlung**: Es gibt keine spezielle Behandlung von Überlauf- oder Unterlaufbedingungen.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- **Messwertverarbeitung**: Umwandlung von Sensordaten, die als `SINT` vorliegen, in `REAL` für weitere Berechnungen.
- **Steuerungsalgorithmen**: Integration in Regelkreise, wo Fließkommawerte benötigt werden.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_INT_TO_REAL**: Konvertiert einen 16-Bit-Integer (`INT`) in `REAL`.
- **F_DINT_TO_REAL**: Konvertiert einen 32-Bit-Integer (`DINT`) in `REAL`.
- **F_SINT_TO_REAL** ist spezifisch für 8-Bit-Integer und bietet damit eine schlankere Lösung für diesen Datentyp.

Fazit

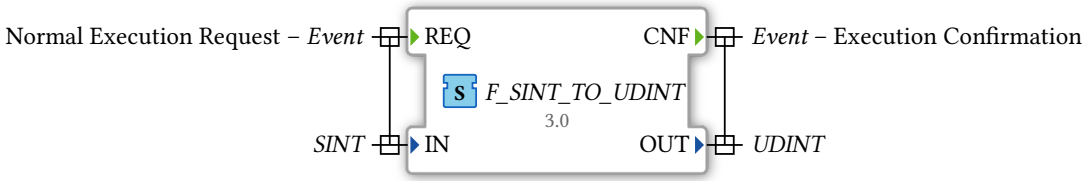
Der `F_SINT_TO_REAL`-Funktionsblock ist ein einfaches und effizientes Werkzeug zur Konvertierung von 8-Bit-Integer-Werten in Fließkommazahlen. Seine klare Schnittstelle und direkte Funktionsweise machen ihn ideal für Anwendungen, die eine schnelle und unkomplizierte Typumwandlung erfordern.

F_SINT_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_SINT_TO_UDINT` dient der Konvertierung eines vorzeichenbehafteten 8-Bit-Integer-Werts (SINT) in einen vorzeichenlosen 32-Bit-Integer-Wert (UDINT). Diese Konvertierung ist besonders nützlich, wenn Daten zwischen Systemen ausgetauscht werden, die unterschiedliche Datentypen verwenden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Dieser Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `SINT`, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgangswert vom Typ `UDINT`, der das Ergebnis der Konvertierung enthält.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem er den `SINT`-Wert am Eingang `IN` in einen `UDINT`-Wert umwandelt und das Ergebnis am Ausgang `OUT` bereitstellt. Die Konvertierung wird durch ein Ereignis am Eingang `REQ` ausgelöst und durch ein Ereignis am Ausgang `CNF` bestätigt.

Technische Besonderheiten

- Die Konvertierung erfolgt mittels der Funktion `SINT_TO_UDINT`.
- Der Funktionsblock ist Teil des Pakets `iec61131::conversion`.

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf ein Ereignis am Eingang `REQ`.
2. **Aktiv-Zustand**: Führt die Konvertierung durch und signalisiert den Abschluss mit einem Ereignis am Ausgang `CNF`.

Anwendungsszenarien

- Konvertierung von Sensorwerten, die als `SINT` vorliegen, in einen `UDINT`-Wert für weitere Verarbeitung.
- Datenaufbereitung für Systeme, die vorzeichenlose Integer-Werte erwarten.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_INT_TO_UDINT**: Konvertiert einen vorzeichenbehafteten 16-Bit-Integer (`INT`) in einen vorzeichenlosen 32-Bit-Integer (`UDINT`).
- **F_DINT_TO_UDINT**: Konvertiert einen vorzeichenbehafteten 32-Bit-Integer (`DINT`) in einen vorzeichenlosen 32-Bit-Integer (`UDINT`).

Fazit

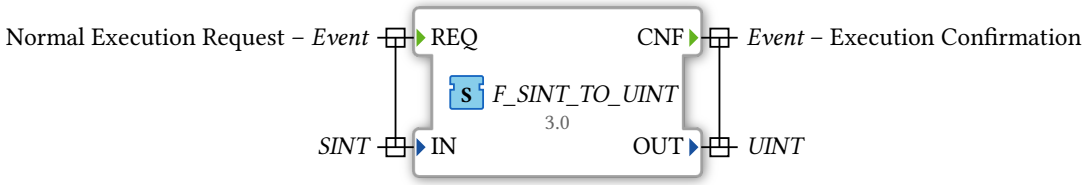
Der Funktionsblock `F_SINT_TO_UDINT` bietet eine einfache und effiziente Möglichkeit, `SINT`-Werte in `UDINT`-Werte umzuwandeln. Durch seine klare Schnittstellenstruktur und einfache Funktionsweise ist er ideal für Anwendungen, die eine solche Konvertierung erfordern.

F_SINT_TO_UINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_SINT_TO_UINT` dient der Konvertierung eines vorzeichenbehafteten 8-Bit-Integer-Werts (`SINT`) in einen vorzeichenlosen 16-Bit-Integer-Wert (`UINT`). Diese Konvertierung ist insbesondere dann notwendig, wenn Daten zwischen Systemen ausgetauscht werden, die unterschiedliche Datentypen verwenden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN**: Der Eingang erwartet einen Wert vom Typ `SINT` (vorzeichenbehafteter 8-Bit-Integer).

Daten-Ausgänge

- **OUT**: Der Ausgang liefert den konvertierten Wert als `UINT` (vorzeichenloser 16-Bit-Integer).

Adapter

- Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` empfangen wird. Dabei wird der Wert am Eingang `IN` vom Typ `SINT` in einen `UINT`-Wert umgewandelt und am Ausgang `OUT` ausgegeben. Die erfolgreiche Konvertierung wird durch das Ereignis `CNF` signalisiert.

Technische Besonderheiten

- Die Konvertierung erfolgt mittels der Funktion `SINT_TO_UINT`, die im Algorithmus des Funktionsblocks implementiert ist.
- Der Funktionsblock ist einfach und benötigt keine zusätzliche Zustandsverwaltung.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine komplexen Zustände. Der Block reagiert direkt auf das `REQ`-Ereignis mit der Ausführung der Konvertierung und der Ausgabe des Ergebnisses.

Anwendungsszenarien

- Konvertierung von Sensorwerten, die als `SINT` vorliegen, für Systeme, die `UINT` erwarten.
- Datenaufbereitung für Kommunikationsprotokolle, die vorzeichenlose Werte benötigen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken ist `F_SINT_TO_UINT` spezialisiert und optimiert für die Konvertierung von `SINT` zu `UINT`.
- Andere Blöcke wie `F_INT_TO_UINT` oder `F_DINT_TO_UDINT` bieten ähnliche Funktionalität, jedoch für andere Datentypen.

🛠️ Zugehörige Übungen

- Uebung_035
- Uebung_035b
- Uebung_035c
- Uebung_036
- Uebung_037
- Uebung_038
- Uebung_038_AX
- SINT_TO_Q_NumericValue
- Uebung_040
- Uebung_040_2
- Uebung_040_AX
- Uebung_041

Fazit

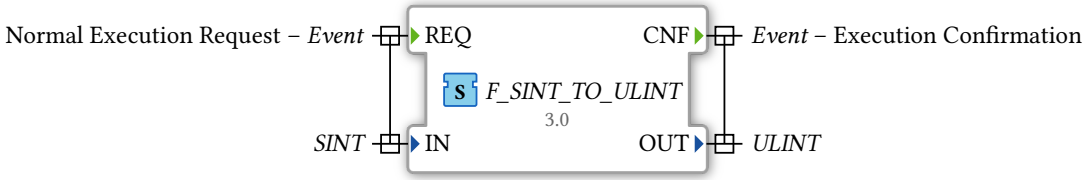
Der `F_SINT_TO_UINT` Funktionsblock ist ein effizientes Werkzeug für die spezifische Konvertierung von vorzeichenbehafteten zu vorzeichenlosen Integer-Werten. Seine Einfachheit und direkte Funktionsweise machen ihn zu einer zuverlässigen Lösung in Steuerungs- und Automatisierungssystemen.

F_SINT_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_SINT_TO_ULINT` dient der Konvertierung eines vorzeichenbehafteten 8-Bit-Integer-Werts (`SINT`) in einen vorzeichenlosen 64-Bit-Integer-Wert (`ULINT`). Dieser Baustein ist besonders nützlich in Szenarien, bei denen eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Dieser Ereigniseingang ist mit dem Dateneingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ereignisausgang ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `SINT`, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der konvertierte Ausgangswert vom Typ `ULINT`.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, wenn das Ereignis `REQ` empfangen wird. Dabei wird der Wert am Eingang `IN` vom Typ `SINT` in einen `ULINT`-Wert umgewandelt und am Ausgang `OUT` ausgegeben. Die erfolgreiche Konvertierung wird durch das Ereignis `CNF` signalisiert.

Technische Besonderheiten

- Die Konvertierung erfolgt durch die eingebaute Funktion `SINT_TO_ULINT`.
- Der Baustein ist einfach und deterministisch, da er keine internen Zustände speichert.

Zustandsübersicht

Der Funktionsblock hat keinen internen Zustand. Die Konvertierung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo `SINT`-Werte in größere vorzeichenlose Integer-Typen umgewandelt werden müssen.
- Datenverarbeitung in Systemen, die mit unterschiedlichen Integer-Formaten arbeiten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Bausteine wie `F_INT_TO_ULINT` oder `F_DINT_TO_ULINT` führen ähnliche Konvertierungen durch, jedoch mit anderen Eingangstypen.
- `F_SINT_TO_ULINT` ist spezifisch für die Konvertierung von `SINT` zu `ULINT` optimiert.

Fazit

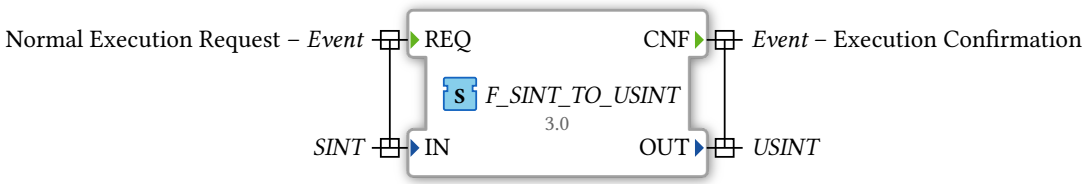
Der `F_SINT_TO_ULINT`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, `SINT`-Werte in `ULINT` umzuwandeln. Er ist besonders in Anwendungen nützlich, bei denen eine sichere und schnelle Typumwandlung erforderlich ist.

F_SINT_TO_USINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_SINT_TO_USINT` dient der Konvertierung eines vorzeichenbehafteten 8-Bit-Integer-Werts (SINT) in einen vorzeichenlosen 8-Bit-Integer-Wert (USINT). Diese Konvertierung ist nützlich, wenn Daten zwischen Systemen ausgetauscht werden, die unterschiedliche Darstellungen für numerische Werte verwenden.



Schnittstellenstruktur

Ereignis-Eingänge
- **REQ**: Startet die Konvertierung. Dieser Ereigniseingang ist mit dem Dateneingang `IN` verknüpft.

Ereignis-Ausgänge
- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ereignisausgang ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge
- **IN** (SINT): Der vorzeichenbehaftete 8-Bit-Integer-Wert, der konvertiert werden soll.

Daten-Ausgänge
- **OUT** (USINT): Der resultierende vorzeichenlose 8-Bit-Integer-Wert nach der Konvertierung.

Adapter
Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise
Bei Auslösung des Ereignisses `REQ` wird der Algorithmus ausgeführt, der den Wert von `IN` von SINT zu USINT konvertiert und das Ergebnis an `OUT` ausgibt. Anschließend wird das Ereignis `CNF` ausgelöst, um den erfolgreichen Abschluss der Konvertierung zu signalisieren.

Technische Besonderheiten
- Der Funktionsblock führt eine direkte Typumwandlung durch, wobei der vorzeichenbehaftete Wert in einen vorzeichenlosen Wert umgewandelt wird.
- Es gibt keine zusätzliche Logik zur Behandlung von Überlaufbedingungen oder ungültigen Eingaben.

Zustandsübersicht
Der Funktionsblock hat einen einfachen Zustandsautomaten:

1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Processing**: Führt die Konvertierung durch und löst `CNF` aus.
3. Keine weiteren Zustände, da es sich um einen einfachen, nicht-zustandsbehafteten Block handelt.

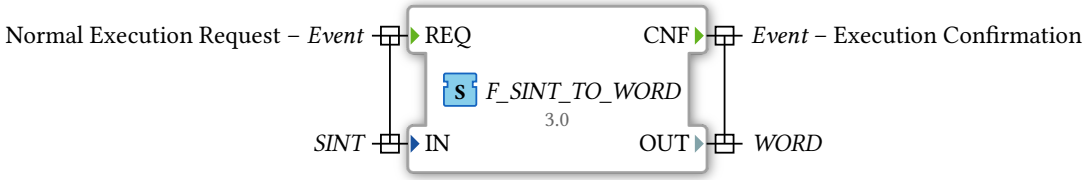
Anwendungsszenarien
- Konvertierung von Sensorwerten, die als vorzeichenbehaftete Werte geliefert werden, aber als vorzeichenlose Werte weiterverarbeitet werden sollen.
- Datenaufbereitung für Kommunikationsprotokolle, die vorzeichenlose Werte erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen
- **F_INT_TO_UINT**: Konvertiert vorzeichenbehaftete 16-Bit-Integer-Werte in vorzeichenlose 16-Bit-Integer-Werte.
- **F_DINT_TO_UDINT**: Konvertiert vorzeichenbehaftete 32-Bit-Integer-Werte in vorzeichenlose 32-Bit-Integer-Werte.
- **F_SINT_TO_USINT** ist spezifisch für 8-Bit-Werte und daher effizienter für diese Datengröße.

Fazit
Der `F_SINT_TO_USINT`-Funktionsblock ist ein einfaches, aber effektives Werkzeug zur Konvertierung von vorzeichenbehafteten zu vorzeichenlosen 8-Bit-Integer-Werten. Seine Einfachheit und Effizienz machen ihn ideal für Anwendungen, bei denen solche Konvertierungen regelmäßig durchgeführt werden müssen.

F_SINT_TO_WORD
• − ◦ • − ◦ • − ◦ •

Einleitung
Der Funktionsblock `F_SINT_TO_WORD` dient der Konvertierung eines SINT-Wertes (8-Bit Ganzahl mit Vorzeichen) in einen WORD-Wert (16-Bit Ganzahl ohne Vorzeichen). Dieser Baustein ist besonders nützlich in Szenarien, wo eine Typumwandlung zwischen unterschiedlichen Integer-Formaten erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ` (Normal Execution Request): Startet die Ausführung der Konvertierung. Wird mit dem Eingabewert `IN` verknüpft.

Ereignis-Ausgänge

- `CNF` (Execution Confirmation): Signalisiert den Abschluss der Konvertierung. Liefert den konvertierten Wert über `OUT`.

Daten-Eingänge

- `IN` (SINT): Der Eingabewert, der von SINT zu WORD konvertiert werden soll.

Daten-Ausgänge

- `OUT` (WORD): Der konvertierte WORD-Wert.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Auslösung des `REQ`-Ereignisses wird der Algorithmus ausgeführt, der den `IN`-Wert von SINT zu WORD konvertiert. Das Ergebnis wird über `OUT` ausgegeben und das `CNF`-Ereignis signalisiert die erfolgreiche Konvertierung.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt durch die Funktion `SINT_TO_WORD(IN)`.
- Keine zusätzliche Fehlerbehandlung für Überlauf oder Unterlauf, da SINT (8-Bit) in WORD (16-Bit) konvertiert wird und somit keine Werte verloren gehen.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung von `REQ`.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo unterschiedliche Integer-Formate verarbeitet werden müssen.
- Datenvorverarbeitung für Kommunikationsprotokolle oder Sensordaten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_INT_TO_WORD` oder `F_DINT_TO_WORD` bieten Konvertierungen von anderen Integer-Typen zu WORD.
- `F_SINT_TO_WORD` ist spezifisch für die Konvertierung von 8-Bit vorzeichenbehafteten zu 16-Bit vorzeichenlosen Integern optimiert.

Fazit

Der `F_SINT_TO_WORD` Funktionsblock ist ein einfacher und effizienter Baustein für die Typumwandlung von SINT zu WORD. Seine klare Schnittstelle und direkte Funktionsweise machen ihn ideal für Anwendungen, die eine schnelle und zuverlässige Konvertierung benötigen.

SINT Konvertierungen

🎧 Podcast
- SINT, INT, DINT: Warum die Wahl des Datentyps über Effizienz und Fehler entscheidet

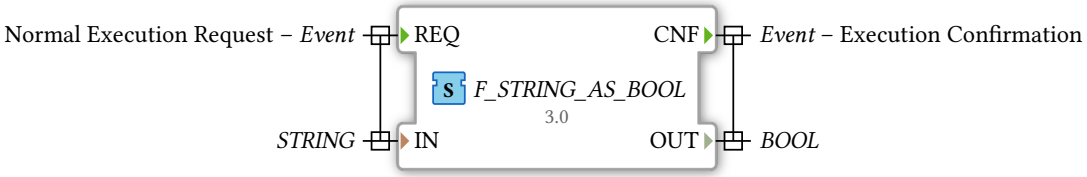
🛠️ Zugehörige Übungen
- SINT_TO_Q_NumericValue

STRING

F_STRING_AS_BOOL

• – ◦ • – ◦ • – ◦ •

Einleitung
Der Funktionsblock `F_STRING_AS_BOOL` dient der Konvertierung eines `STRING`-Werts in einen `BOOL`-Wert. Dieser Baustein ist besonders nützlich, wenn Zeichenketten in boolesche Werte umgewandelt werden müssen, beispielsweise bei der Verarbeitung von Benutzereingaben oder der Interpretation von Textdaten.



Schnittstellenstruktur

Ereignis-Eingänge
- **REQ**: Normaler Ausführungsanforderung. Dieser Eingang löst die Konvertierung aus.

Ereignis-Ausgänge
- **CNF**: Ausführungsbestätigung. Dieser Ausgang signalisiert die erfolgreiche Beendigung der Konvertierung.

Daten-Eingänge
- **IN** (`STRING`): Der Eingabe-String, der in einen booleschen Wert konvertiert werden soll.

Daten-Ausgänge
- **OUT** (`BOOL`): Der resultierende boolesche Wert nach der Konvertierung.

Adapter
Es sind keine Adapter vorhanden.

Funktionsweise
Der Funktionsblock nutzt die Funktion `STRING_AS_BOOL`, um den Eingabe-String `IN` in einen booleschen Wert `OUT` umzuwandeln. Die Konvertierung wird durch das Ereignis `REQ` ausgelöst, und nach Abschluss der Operation wird das Ereignis `CNF` ausgegeben.

Technische Besonderheiten
- Die Konvertierung erfolgt gemäß der IEC 61131-3 Spezifikation für die Umwandlung von Strings in boolesche Werte.
- Der Baustein ist Teil des Pakets `iec61131::conversion`.

Zustandsübersicht
1. **Initialisierung**: Der Funktionsblock wartet auf das `REQ`-Ereignis.
2. **Ausführung**: Bei Empfang von `REQ` wird der String `IN` in einen booleschen Wert konvertiert und das Ergebnis an `OUT` ausgegeben.
3. **Bestätigung**: Nach erfolgreicher Konvertierung wird das `CNF`-Ereignis ausgegeben.

Anwendungsszenarien

- Verarbeitung von Benutzereingaben, die als String vorliegen (z.B. „true“ oder „false“).
- Interpretation von Textdaten aus externen Quellen (z.B. Konfigurationsdateien oder Netzwerkkommunikation).
- Integration in Systeme, die boolesche Werte als Strings speichern oder übertragen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_STRING_TO_BOOL**: Ein ähnlicher Baustein, der möglicherweise zusätzliche Optionen für die String-Interpretation bietet.
- **F_STRING_AS_***: Andere Konvertierungsbausteine für verschiedene Datentypen (z.B. `F_STRING_AS_INT`).

Fazit

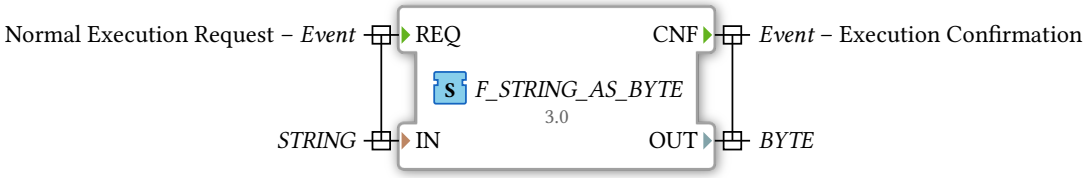
Der `F_STRING_AS_BOOL` Funktionsblock bietet eine einfache und effiziente Möglichkeit, String-Werte in boolesche Werte umzuwandeln. Durch seine klare Schnittstellenstruktur und Zuverlässigkeit ist er ideal für Anwendungen, die eine solche Konvertierung erfordern.

F_STRING_AS_BYTE

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_STRING_AS_BYTE` dient der Konvertierung eines `STRING`-Wertes in einen `BYTE`-Wert. Diese Funktionalität ist besonders nützlich, wenn Zeichenketten in ihre entsprechenden Byte-Werte umgewandelt werden müssen, beispielsweise für die Verarbeitung in Protokollen oder bei der Kommunikation mit Hardware.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Dieser Ereigniseingang löst die Konvertierung aus. Er ist mit dem Dateneingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Dieses Ereignis wird nach erfolgreicher Konvertierung ausgelöst und ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Eingabe vom Typ `STRING`, der in einen `BYTE`-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Ausgabe vom Typ `BYTE`, die das Ergebnis der Konvertierung enthält.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung des `STRING`-Wertes `IN` in einen `BYTE`-Wert `OUT` durch, sobald das Ereignis `REQ` empfangen wird. Die Konvertierung erfolgt mittels der Funktion `STRING_AS_BYTE`. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher Funktionsblock (`SimpleFB`) und verwendet einen ST-Algorithmus für die Konvertierung.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Verzögerung.

Zustandsübersicht

1. **Wartezustand**: Der Funktionsblock wartet auf das Ereignis `REQ`.
2. **Ausführungszustand**: Bei Empfang von `REQ` wird die Konvertierung durchgeführt.
3. **Bestätigungszustand**: Nach erfolgreicher Konvertierung wird `CNF` ausgelöst und der Funktionsblock kehrt in den Wartezustand zurück.

Anwendungsszenarien

- Konvertierung von Zeichenketten in Byte-Werte für die Kommunikation mit seriellen Schnittstellen.
- Verarbeitung von ASCII-Zeichen in Byte-Form für Protokolle oder Hardware-Steuerungen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen wie `F_STRING_TO_INT` oder `F_STRING_TO_REAL` ist `F_STRING_AS_BYTE` speziell auf die Umwandlung in einen `BYTE`-Wert ausgelegt.
- Ein ähnlicher Baustein könnte `F_CHAR_AS_BYTE` sein, der jedoch einzelne Zeichen konvertiert, während `F_STRING_AS_BYTE` auf Zeichenketten arbeitet.

Fazit

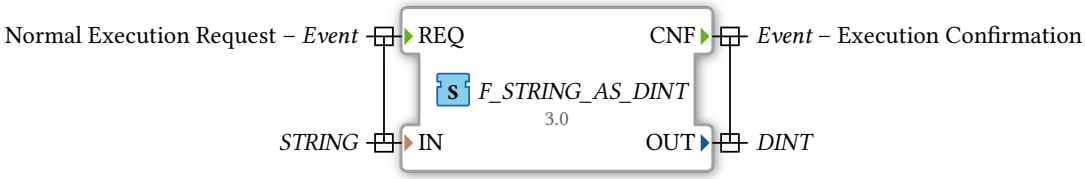
Der Funktionsblock `F_STRING_AS_BYTE` bietet eine einfache und effiziente Möglichkeit, Zeichenketten in Byte-Werte umzuwandeln. Durch seine klare Schnittstellenstruktur und direkte Funktionsweise eignet er sich ideal für Anwendungen, die eine solche Konvertierung erfordern.

F_STRING_AS_DINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_STRING_AS_DINT` dient der Konvertierung eines `STRING`-Wertes in einen `DINT`-Wert. Diese Funktion ist besonders nützlich, wenn Zeichenketten in numerische Werte umgewandelt werden müssen, beispielsweise bei der Verarbeitung von Benutzereingaben oder der Interpretation von Textdaten.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Dieser Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (`STRING`): Der zu konvertierende Zeichenkettenwert.

Daten-Ausgänge

- **OUT** (`DINT`): Das Ergebnis der Konvertierung als 32-Bit-Ganzzahl.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` ausgelöst wird. Dabei wird der `STRING`-Wert am Eingang `IN` in einen `DINT`-Wert umgewandelt und am Ausgang `OUT` ausgegeben. Die Konvertierung erfolgt mithilfe der Funktion `STRING_AS_DINT`. Nach Abschluss der Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Die Konvertierung kann fehlschlagen, wenn der `STRING`-Wert keine gültige numerische Repräsentation enthält. In solchen Fällen kann der Ausgang `OUT` einen undefinierten Wert annehmen.
- Der Funktionsblock ist Teil des Pakets `iec61131::conversion`.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Verarbeitung von Benutzereingaben, die als Zeichenketten vorliegen.
- Interpretation von Textdaten aus Dateien oder Netzwerkquellen.
- Umwandlung von konfigurierbaren Parametern, die als Zeichenketten gespeichert sind.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu `F_STRING_TO_*`-Funktionsblöcken, die explizite Typkonvertierungen durchführen, interpretiert `F_STRING_AS_DINT` den `STRING`-Wert direkt als `DINT`.
- Ähnliche Funktionsblöcke wie `F_STRING_TO_DINT` könnten zusätzliche Parameter für die Formatierung bieten, sind jedoch in der Standardbibliothek nicht enthalten.

Fazit

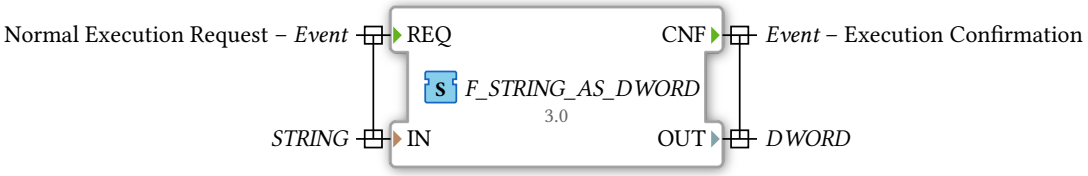
Der `F_STRING_AS_DINT`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, Zeichenketten in 32-Bit-Ganzzahlen umzuwandeln. Seine Verwendung ist besonders in Szenarien sinnvoll, in denen eine direkte Interpretation der Zeichenkette als numerischer Wert erforderlich ist.

F_STRING_AS_DWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_STRING_AS_DWORD` dient der Konvertierung eines `STRING`-Wertes in einen `DWORD`-Wert. Diese Funktionalität ist besonders nützlich, wenn Zeichenketten in numerische Werte umgewandelt werden müssen, beispielsweise für die weitere Verarbeitung in Steuerungsalgorithmen.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Startet die Ausführung der Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- `CNF`: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- `IN` (`STRING`): Der Eingabewert, der in einen `DWORD`-Wert umgewandelt werden soll.

Daten-Ausgänge

- `OUT` (`DWORD`): Der Ergebniswert der Konvertierung.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung des `STRING`-Eingabewertes `IN` in einen `DWORD`-Wert durch, sobald das Ereignis `REQ` ausgelöst wird. Die Konvertierung erfolgt mithilfe der Funktion `STRING_AS_DWORD`. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst, und der Ergebniswert steht am Ausgang `OUT` zur Verfügung.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher Funktionsblock (`SimpleFB`) und führt die Konvertierung in einem einzigen Algorithmus durch.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Zustandsverwaltung.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Ausführung erfolgt direkt bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Konvertierung von Benutzereingaben oder Kommunikationsdaten in numerische Werte.
- Verwendung in Steuerungsalgorithmen, die numerische Werte anstelle von Zeichenketten erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen wie `F_STRING_TO_INT` oder `F_STRING_TO_REAL` ist dieser Baustein speziell auf die Konvertierung in den `DWORD`-Datentyp ausgelegt.
- Die direkte Verwendung der `STRING_AS_DWORD`-Funktion ermöglicht eine effiziente und spezialisierte Konvertierung.

Fazit

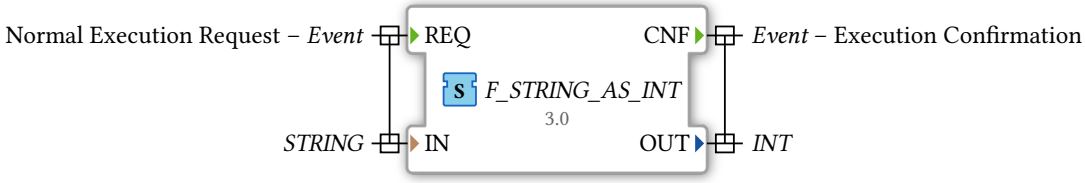
Der `F_STRING_AS_DWORD`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, Zeichenketten in `DWORD`-Werte umzuwandeln. Durch seine klare Schnittstellenstruktur und direkte Funktionsweise ist er ideal für Anwendungen geeignet, die eine schnelle und zuverlässige Konvertierung erfordern.

F_STRING_AS_INT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_STRING_AS_INT` dient der Konvertierung eines `STRING`-Wertes in einen `INT`-Wert. Diese Funktionalität ist besonders nützlich, wenn Zeichenketten, die numerische Werte repräsentieren, in ganzzahlige Werte umgewandelt werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Ereigniseingang ausgelöst, wird der `STRING`-Wert am Daten-Eingang `IN` in einen `INT`-Wert umgewandelt.

Ereignis-Ausgänge

- **CNF**: Signalisiert die erfolgreiche Beendigung der Konvertierung. Dieses Ereignis wird zusammen mit dem konvertierten `INT`-Wert am Daten-Ausgang `OUT` ausgegeben.

Daten-Eingänge

- **IN**: Der `STRING`-Wert, der in einen `INT`-Wert umgewandelt werden soll.

Daten-Ausgänge

- **OUT**: Der konvertierte `INT`-Wert.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das `REQ`-Ereignis empfangen wird. Dabei wird die eingebaute Funktion `STRING_AS_INT` verwendet, um den `STRING`-Wert am Eingang `IN` in einen `INT`-Wert umzuwandeln. Das Ergebnis wird am Ausgang `OUT` ausgegeben und durch das `CNF`-Ereignis bestätigt.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne zusätzliche Parameter.
- Der Funktionsblock ist einfach und effizient implementiert, ohne komplexe Zustandslogik.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Verarbeitung von Benutzereingaben, die als Zeichenketten vorliegen, aber als numerische Werte benötigt werden.
- Umwandlung von Daten aus Textdateien oder Kommunikationsprotokollen in numerische Werte.
- Integration in Systeme, die Zeichenketten-basierte Datenquellen nutzen, aber numerische Verarbeitung erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsbausteinen ist `F_STRING_AS_INT` spezialisiert auf die Umwandlung von `STRING` zu `INT`.
- Andere Bausteine könnten zusätzliche Funktionen wie Fehlerbehandlung oder Unterstützung für verschiedene numerische Formate bieten, was hier nicht der Fall ist.

Fazit

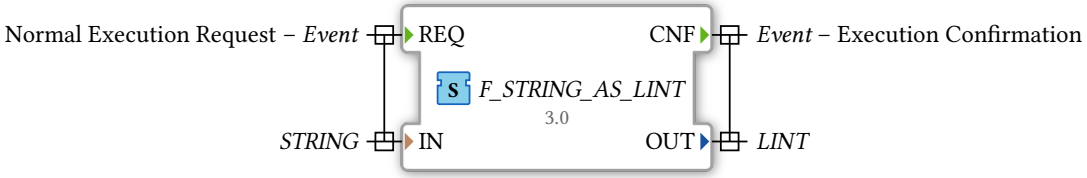
Der `F_STRING_AS_INT`-Funktionsblock ist ein einfaches und effektives Werkzeug zur Konvertierung von Zeichenketten in ganzzahlige Werte. Seine Stärke liegt in der Einfachheit und direkten Anwendbarkeit, ohne zusätzliche Komplexität. Für Anwendungen, die eine schnelle und unkomplizierte Umwandlung benötigen, ist dieser Baustein ideal geeignet.

F_STRING_AS_LINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_STRING_AS_LINT` dient der Konvertierung eines `STRING`-Wertes in einen `LINT`-Wert. Diese Funktionalität ist insbesondere dann nützlich, wenn Zeichenketten in numerische Werte umgewandelt werden müssen, um sie in weiteren Berechnungen oder Steuerungslogiken zu verwenden.



Schnittstellenstruktur

Ereignis-Eingänge
- **REQ**: Startet die Konvertierung des `STRING`-Wertes zu einem `LINT`-Wert. Dieser Ereigniseingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge
- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ereignisausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge
- **IN**: Der Eingang für den `STRING`-Wert, der in einen `LINT`-Wert konvertiert werden soll.

Daten-Ausgänge
- **OUT**: Der Ausgang für den konvertierten `LINT`-Wert.

Adapter
Es sind keine Adapter vorhanden.

Funktionsweise
Der Funktionsblock führt die Konvertierung des `STRING`-Wertes zu einem `LINT`-Wert durch, sobald das `REQ`-Ereignis ausgelöst wird. Die Konvertierung erfolgt mithilfe der Funktion `STRING_AS_LINT()`. Nach erfolgreicher Konvertierung wird das `CNF`-Ereignis ausgelöst, und der konvertierte Wert steht am `OUT`-Ausgang zur Verfügung.

Technische Besonderheiten
- Der Funktionsblock ist ein einfacher Funktionsblock (SimpleFB) und führt die Konvertierung in einem einzigen Algorithmus durch.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Zustandsverwaltung.

Zustandsübersicht
Da es sich um einen einfachen Funktionsblock handelt, gibt es keine komplexen Zustandsübergänge. Der Block führt die Konvertierung bei jedem `REQ`-Ereignis aus und signalisiert den Abschluss mit dem `CNF`-Ereignis.

Anwendungsszenarien
- Konvertierung von Benutzereingaben in numerische Werte.
- Verarbeitung von Zeichenketten, die numerische Daten enthalten, in Steuerungssystemen.
- Integration in größere Steuerungslogiken, die numerische Berechnungen erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen
- **F_STRING_AS_INT**: Konvertiert einen `STRING` in einen `INT`-Wert, ist jedoch auf kleinere Zahlenbereiche beschränkt.
- **F_STRING_AS_REAL**: Konvertiert einen `STRING` in einen `REAL`-Wert, ist für Gleitkommazahlen geeignet.
- **F_LINT_AS_STRING**: Führt die umgekehrte Konvertierung durch, also von `LINT` zu `STRING`.

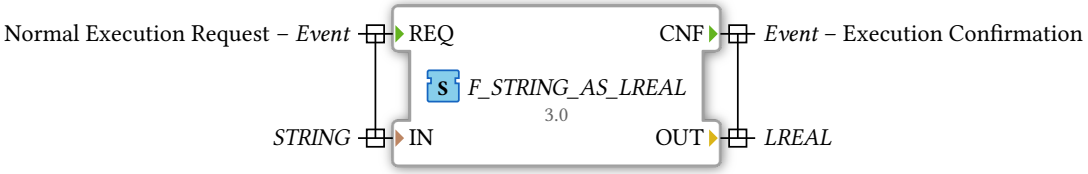
Fazit
Der Funktionsblock `F_STRING_AS_LINT` bietet eine einfache und effiziente Möglichkeit, Zeichenketten in große numerische Werte (`LINT`) umzuwandeln. Durch seine direkte Funktionsweise und klare Schnittstellenstruktur ist er ideal für den Einsatz in Steuerungssystemen, die eine solche Konvertierung erfordern.

F_STRING_AS_LREAL

• − ◦ • − ◦ • − ◦ •

Einleitung
Der Funktionsblock `F_STRING_AS_LREAL` dient der Konvertierung eines STRING-Wertes in einen LREAL-Wert (64-Bit Fließkommazahl). Dieser Baustein ist besonders nützlich, wenn Zeichenketten in numerische

Werte umgewandelt werden müssen, z.B. bei der Verarbeitung von Benutzereingaben oder beim Lesen von Daten aus externen Quellen.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- `CNF`: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- `IN` (STRING): Der zu konvertierende Zeichenkettenwert.

Daten-Ausgänge

- `OUT` (LREAL): Das Ergebnis der Konvertierung als 64-Bit Fließkommazahl.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Auslösung des `REQ`-Ereignisses wird der Wert am Eingang `IN` als Zeichenkette eingelesen und in einen LREAL-Wert konvertiert. Das Ergebnis wird am Ausgang `OUT` ausgegeben und das `CNF`-Ereignis signalisiert die erfolgreiche Ausführung.

Der Konvertierungsalgorithmus ist als ST-Code implementiert:

```
ALGORITHM REQ
OUT := STRING_AS_LREAL(IN);
END_ALGORITHM
```

Technische Besonderheiten

- Die Konvertierung folgt den Standardregeln für die Umwandlung von STRING zu LREAL in IEC 61131-3.
- Bei ungültigen Eingabewerten (z.B. nicht-numerische Zeichenketten) kann das Verhalten abhängig von der Laufzeitumgebung variieren.

Zustandsübersicht

Der Funktionsblock besitzt einen einfachen Zustandsautomaten:

1. Wartet auf `REQ`-Ereignis
2. Führt Konvertierung durch
3. Sendet `CNF`-Ereignis
4. Kehrt in Wartezustand zurück

Anwendungsszenarien

- Verarbeitung von Benutzereingaben in numerischen Steuerungen
- Konvertierung von Textdaten aus Dateien oder Netzwerkquellen
- Schnittstellen zu Systemen, die numerische Daten als Zeichenketten übertragen

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsbausteinen ist dieser Funktionsblock spezialisiert auf die STRING-zu-LREAL-Konvertierung.
- Andere ähnliche Bausteine könnten zusätzliche Parameter für die Zahlenformatierung bieten.

Fazit

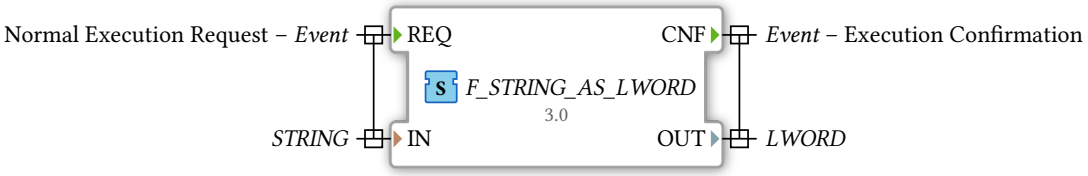
Der `F_STRING_AS_LREAL` Funktionsblock bietet eine einfache und effiziente Möglichkeit, Zeichenketten in 64-Bit Fließkommazahlen umzuwandeln. Seine klare Schnittstelle und einfache Funktionsweise machen ihn besonders geeignet für Standardkonvertierungsaufgaben in Automatisierungsprojekten.

F_STRING_AS_LWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_STRING_AS_LWORD` dient der Konvertierung eines `STRING`-Werts in einen `LWORD`-Wert. Er ist Teil des `iec61131::conversion`-Pakets und ermöglicht die einfache Umwandlung zwischen diesen beiden Datentypen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung der Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (`STRING`): Der Eingangsstring, der in einen `LWORD`-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT** (`LWORD`): Der konvertierte `LWORD`-Wert.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung aus, wenn das Ereignis `REQ` empfangen wird. Dabei wird der Eingangsstring `IN` mit der Funktion `STRING_AS_LWORD` in einen `LWORD`-Wert umgewandelt und am Ausgang `OUT` ausgegeben. Nach Abschluss der Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne zusätzliche Parameter.
- Der Block ist als einfacher Funktionsblock (`SimpleFB`) implementiert, der nur einen Algorithmus enthält.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung wird bei jedem `REQ`-Ereignis neu ausgeführt.

Anwendungsszenarien

- Umwandlung von String-Daten in `LWORD`-Werte für die weitere Verarbeitung in Steuerungssystemen.
- Integration in Systeme, die `LWORD`-Werte erfordern, aber String-Eingaben erhalten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen ist `F_STRING_AS_LWORD` spezialisiert auf die Umwandlung von `STRING` zu `LWORD`.

- Andere Bausteine wie `F_STRING_TO_*` könnten ähnliche Funktionen für andere Datentypen bieten.

Fazit

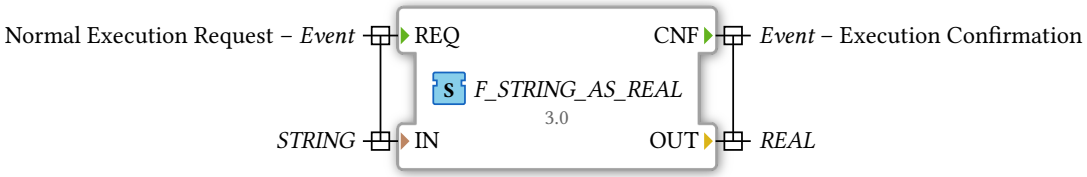
Der `F_STRING_AS_LWORD`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, String-Daten in `LWORD`-Werte umzuwandeln. Er ist besonders nützlich in Szenarien, wo solche Konvertierungen regelmäßig benötigt werden.

F_STRING_AS_REAL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_STRING_AS_REAL` dient der Konvertierung eines `STRING`-Wertes in einen `REAL`-Wert. Er ist besonders nützlich, wenn Daten aus einer Zeichenkette in eine numerische Darstellung umgewandelt werden müssen, beispielsweise bei der Verarbeitung von Benutzereingaben oder der Interpretation von Textdateien.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Bei Auslösung dieses Ereignisses wird der Algorithmus zur Umwandlung des `STRING`-Wertes in einen `REAL`-Wert ausgeführt.

Ereignis-Ausgänge

- **CNF**: Signalisiert den erfolgreichen Abschluss der Konvertierung. Dieses Ereignis wird zusammen mit dem konvertierten `REAL`-Wert ausgegeben.

Daten-Eingänge

- **IN** (`STRING`): Der Eingabewert, der in einen `REAL`-Wert umgewandelt werden soll.

Daten-Ausgänge

- **OUT** (`REAL`): Der Ergebniswert der Konvertierung.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt bei Auslösung des `REQ`-Ereignisses die Konvertierung des `STRING`-Wertes `IN` in einen `REAL`-Wert durch. Die Umwandlung erfolgt mittels der eingebauten Funktion `STRING_AS_REAL`. Nach erfolgreicher Konvertierung wird das `CNF`-Ereignis ausgelöst und der Ergebniswert über `OUT` ausgegeben.

Technische Besonderheiten

- Die Konvertierung kann fehlschlagen, wenn der Eingabe-`STRING` keine gültige `REAL`-Zahl darstellt. In diesem Fall kann es zu einem Laufzeitfehler kommen.
- Der Funktionsblock ist einfach gehalten und enthält keine zusätzliche Fehlerbehandlung.

Zustandsübersicht

1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Processing**: Führt die Konvertierung durch.
3. **Completed**: Sendet das `CNF`-Ereignis mit dem Ergebnis.

Anwendungsszenarien

- Verarbeitung von Benutzereingaben, die als Zeichenketten vorliegen.

- Interpretation von Daten aus Textdateien oder Netzwerkkommunikation.
- Umwandlung von Konfigurationswerten, die als Zeichenketten gespeichert sind.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken ist `F_STRING_AS_REAL` spezialisiert auf die Umwandlung von `STRING` zu `REAL`.
- Andere Blöcke könnten zusätzliche Funktionen wie Fehlerbehandlung oder Formatierung bieten, sind dafür aber komplexer.

Fazit

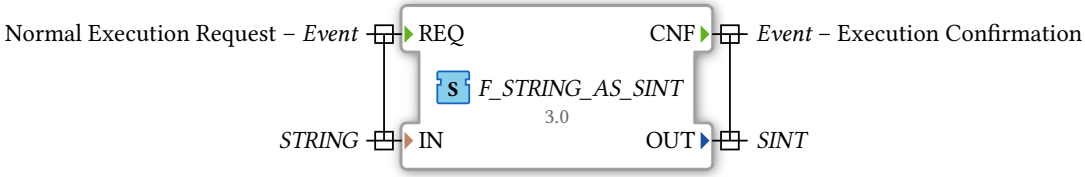
Der `F_STRING_AS_REAL`-Funktionsblock ist ein einfaches und effizientes Werkzeug zur Konvertierung von Zeichenketten in Gleitkommazahlen. Seine Stärke liegt in der Einfachheit und direkten Anwendbarkeit, allerdings ohne integrierte Fehlerbehandlung. Für robuste Anwendungen sollten zusätzliche Sicherheitsmaßnahmen implementiert werden.

F_STRING_AS_SINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_STRING_AS_SINT` dient der Konvertierung eines `STRING`-Wertes in einen `SINT`-Wert (8-Bit Ganzzahl mit Vorzeichen). Dieser Baustein ist besonders nützlich, wenn Zeichenketten in numerische Werte umgewandelt werden müssen, z.B. bei der Verarbeitung von Benutzereingaben oder der Interpretation von Textdaten.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Löst die Ausführung der Konvertierung aus. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- `CNF`: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- `IN` (`STRING`): Die Zeichenkette, die in eine `SINT`-Zahl umgewandelt werden soll.

Daten-Ausgänge

- `OUT` (`SINT`): Das Ergebnis der Konvertierung als 8-Bit Ganzzahl mit Vorzeichen.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Auslösung des `REQ`-Ereignisses wird die Funktion `STRING_AS_SINT` auf den Wert des `IN`-Eingangs angewendet. Das Ergebnis wird im `OUT`-Ausgang gespeichert und das `CNF`-Ereignis signalisiert den erfolgreichen Abschluss der Operation.

Technische Besonderheiten

- Die Konvertierung erfolgt gemäß der IEC 61131-3 Spezifikation für die Umwandlung von Zeichenketten in numerische Werte.
- Der Baustein ist Teil des `iec61131::conversion`-Pakets.

Zustandsübersicht

1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Processing**: Führt die Konvertierung durch.
3. **Completed**: Signalisiert den Abschluss mit `CNF` und kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- Interpretation von Benutzereingaben in numerische Steuerbefehle.
- Verarbeitung von Textdaten aus externen Quellen (z.B. Dateien oder Netzwerkkommunikation).
- Konvertierung von Konfigurationswerten in Steuerungsparameter.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken ist `F_STRING_AS_SINT` spezialisiert auf die Umwandlung in 8-Bit Ganzzahlen.
- Ähnliche Blöcke wie `F_STRING_AS_INT` oder `F_STRING_AS_DINT` bieten Konvertierungen in größere Ganzzahltypen.

Fazit

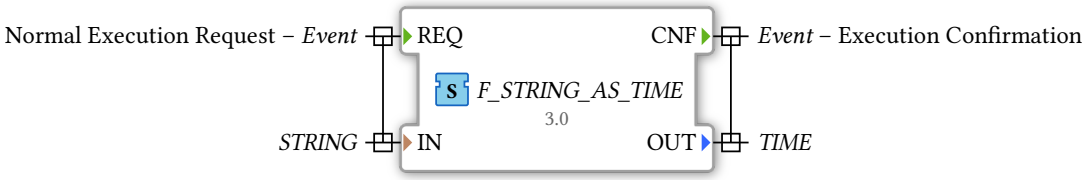
`F_STRING_AS_SINT` ist ein effizienter und spezialisierter Funktionsblock für die Konvertierung von Zeichenketten in 8-Bit Ganzzahlen. Seine einfache Schnittstelle und klare Funktionsweise machen ihn ideal für Steuerungsanwendungen, die mit textbasierten Eingaben arbeiten.

F_STRING_AS_TIME

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_STRING_AS_TIME` dient der Konvertierung eines Zeitwertes im STRING-Format in das TIME-Format. Dies ist besonders nützlich, wenn Zeitwerte als Text vorliegen und für weitere Verarbeitung in ein maschinenlesbares Format umgewandelt werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Startet die Konvertierung des Zeitwertes. Wird dieser Ereigniseingang ausgelöst, beginnt der Funktionsblock mit der Umwandlung des an `IN` übergebenen Strings.

Ereignis-Ausgänge

- `CNF`: Signalisiert die erfolgreiche Beendigung der Konvertierung. Dieses Ereignis wird zusammen mit dem konvertierten Zeitwert am Datenausgang `OUT` ausgegeben.

Daten-Eingänge

- `IN` (STRING): Der Eingang für den Zeitwert im STRING-Format, der in das TIME-Format umgewandelt werden soll.

Daten-Ausgänge

- `OUT` (TIME): Der Ausgang, an dem der konvertierte Zeitwert im TIME-Format ausgegeben wird.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock nutzt die eingebaute Funktion `STRING_AS_TIME`, um den über den Eingang `IN` empfangenen String in einen TIME-Wert umzuwandeln. Die Konvertierung wird durch das Ereignis `REQ` ausgelöst und bei erfolgreicher Umwandlung durch das Ereignis `CNF` bestätigt.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne Zwischenschritte.
- Der Funktionsblock ist Teil des Pakets `iec61131::conversion`.

Zustandsübersicht

1. **Idle**: Der Funktionsblock wartet auf das Ereignis `REQ`.
2. **Processing**: Bei Empfang von `REQ` wird die Konvertierung durchgeführt.
3. **Completion**: Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst und der Zustand wechselt zurück zu Idle.

Anwendungsszenarien

- Verarbeitung von Zeitwerten, die als Text in Benutzereingaben oder externen Datenquellen vorliegen.
- Integration in Systeme, die Zeitwerte im TIME-Format für Steuerungs- oder Regelungsaufgaben benötigen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen ist `F_STRING_AS_TIME` spezialisiert auf die Umwandlung von Zeitwerten und bietet daher eine direkte und effiziente Lösung für diesen spezifischen Anwendungsfall.

Fazit

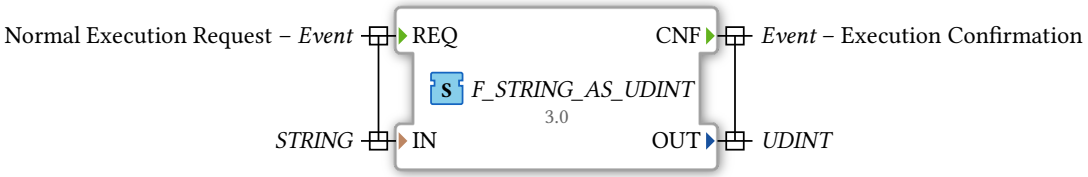
Der `F_STRING_AS_TIME` Funktionsblock ist ein effizientes Werkzeug zur Konvertierung von Zeitwerten im STRING-Format in das TIME-Format. Durch seine einfache und direkte Funktionsweise eignet er sich ideal für Anwendungen, die eine schnelle und zuverlässige Umwandlung benötigen.

F_STRING_AS_UDINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_STRING_AS_UDINT` dient der Konvertierung eines `STRING`-Wertes in einen `UDINT`-Wert (unsigned double integer). Er ist Teil des Pakets `iec61131::conversion` und ermöglicht die einfache Umwandlung von Zeichenketten in numerische Werte.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN**: Der Eingang erwartet einen `STRING`-Wert, der in einen `UDINT`-Wert umgewandelt werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgang liefert den konvertierten `UDINT`-Wert.

Adapter

Der FB verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung mithilfe der eingebauten Funktion `STRING_AS_UDINT` durch. Bei Auslösung des Ereignisses `REQ` wird der `STRING`-Wert am Eingang `IN` in einen `UDINT`-Wert umgewandelt und am Ausgang `OUT` ausgegeben. Anschließend wird das Ereignis `CNF` ausgelöst, um den erfolgreichen Abschluss der Konvertierung zu signalisieren.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne zusätzliche Parameter.
- Der FB ist für einfache Umwandlungen optimiert und unterstützt keine komplexen Formatierungen oder Fehlerbehandlungen.

Zustandsübersicht

1. **Idle**: Wartet auf das Ereignis `REQ`.
2. **Konvertierung**: Führt die Umwandlung durch und gibt das Ergebnis aus.
3. **Bestätigung**: Löst das Ereignis `CNF` aus und kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- Umwandlung von Benutzereingaben oder Kommunikationsdaten in numerische Werte.
- Integration in Systeme, die numerische Verarbeitung erfordern, aber Zeichenketten als Eingabe erhalten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungs-FBs bietet `F_STRING_AS_UDINT` eine spezialisierte und effiziente Lösung für die Umwandlung von `STRING` zu `UDINT`.
- Andere FBs könnten zusätzliche Funktionen wie Fehlerbehandlung oder Formatierung bieten, sind jedoch komplexer in der Handhabung.

Fazit

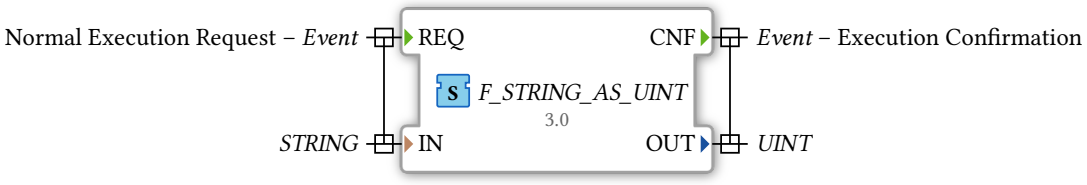
Der `F_STRING_AS_UDINT`-Funktionsblock ist eine einfache und effiziente Lösung für die Konvertierung von Zeichenketten in numerische Werte. Er eignet sich besonders für Anwendungen, bei denen eine schnelle und direkte Umwandlung benötigt wird.

F_STRING_AS_UINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_STRING_AS_UINT` dient der Konvertierung eines `STRING`-Wertes in einen `UINT`-Wert (unsigned integer). Diese Funktion ist besonders nützlich, wenn numerische Werte als Zeichenketten vorliegen und für weitere Berechnungen oder Steuerungsaufgaben in numerischer Form benötigt werden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung des `STRING`-Wertes zu `UINT`. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (`STRING`): Der Eingang für den zu konvertierenden Zeichenkettenwert.

Daten-Ausgänge

- **OUT** (`UINT`): Der Ausgang für den konvertierten numerischen Wert.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Bei Auslösung des Ereignisses `REQ` wird der Algorithmus ausgeführt, der den `STRING`-Wert von `IN` mittels der Funktion `STRING_AS_UINT` in einen `UINT`-Wert umwandelt. Das Ergebnis wird an `OUT` ausgegeben und das Ereignis `CNF` signalisiert den erfolgreichen Abschluss der Konvertierung.

Technische Besonderheiten

- Der Funktionsblock ist einfach aufgebaut und enthält nur einen Algorithmus für die Konvertierung.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Parameter oder komplexe Zustandslogik.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Ausführung erfolgt unmittelbar bei Auslösung von `REQ`.

Anwendungsszenarien

- Verarbeitung von Benutzereingaben, die als Zeichenketten vorliegen.
- Konvertierung von Daten aus Textdateien oder Kommunikationsprotokollen für die weitere numerische Verarbeitung.
- Integration in Steuerungssysteme, die numerische Werte aus String-Quellen benötigen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken ist `F_STRING_AS_UINT` spezialisiert auf die Umwandlung von `STRING` zu `UINT` und bietet daher eine einfache und direkte Lösung für diesen spezifischen Anwendungsfall.
- Andere Blöcke könnten zusätzliche Funktionen wie Fehlerbehandlung oder verschiedene numerische Formate unterstützen, was hier nicht der Fall ist.

Fazit

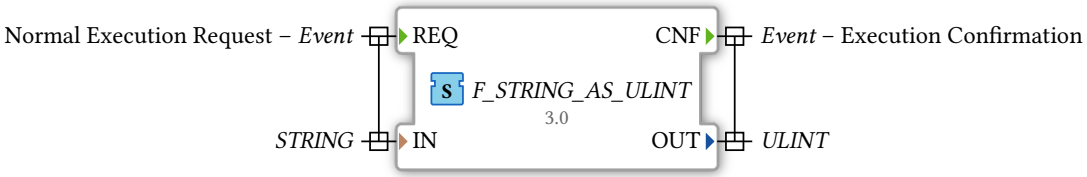
Der `F_STRING_AS_UINT` Funktionsblock ist eine effiziente und unkomplizierte Lösung für die Konvertierung von Zeichenketten in unsigned integer Werte. Seine Einfachheit macht ihn besonders geeignet für Anwendungen, bei denen eine schnelle und direkte Umwandlung benötigt wird.

F_STRING_AS_ULINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_STRING_AS_ULINT` dient der Konvertierung eines `STRING`-Wertes in einen `ULINT`-Wert (unsigned 64-bit integer). Dieser Baustein ist besonders nützlich, wenn Zeichenketten in numerische Werte umgewandelt werden müssen, beispielsweise bei der Verarbeitung von Benutzereingaben oder beim Parsen von Daten aus externen Quellen.



Schnittstellenstruktur

Ereignis-Eingänge
- **REQ**: Startet die Konvertierung. Dieser Ereigniseingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge
- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ereignisausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge
- **IN**: Der Eingang vom Typ `STRING`, der die zu konvertierende Zeichenkette enthält.

Daten-Ausgänge
- **OUT**: Der Ausgang vom Typ `ULINT`, der das Ergebnis der Konvertierung enthält.

Adapter
Es sind keine Adapter vorhanden.

Funktionsweise
Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` ausgelöst wird. Dabei wird die eingebaute Funktion `STRING_AS_ULINT` verwendet, um den `STRING`-Eingang `IN` in einen `ULINT`-Wert umzuwandeln. Das Ergebnis wird am Ausgang `OUT` ausgegeben, und das Ereignis `CNF` signalisiert die erfolgreiche Beendigung der Operation.

Technische Besonderheiten
- Der Funktionsblock ist Teil des Pakets `iec61131::conversion`.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Parameter oder Einstellungen.

Zustandsübersicht
Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung wird bei jedem `REQ`-Ereignis neu ausgeführt.

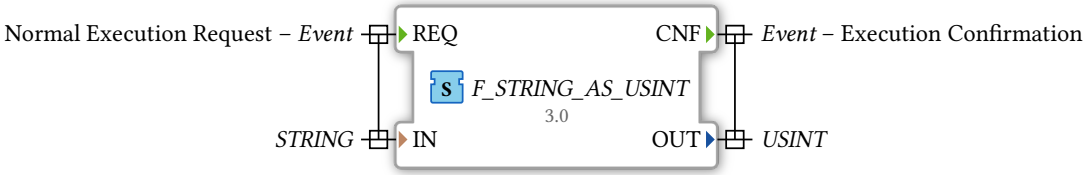
Anwendungsszenarien
- **Datenverarbeitung**: Umwandlung von Zeichenketten in numerische Werte für Berechnungen oder Speicherung.
- **Benutzereingaben**: Verarbeitung von Eingaben, die als Zeichenketten vorliegen, aber als Zahlen interpretiert werden sollen.
- **Protokollierung**: Konvertierung von logischen oder textuellen Daten in numerische Formate für die weitere Analyse.

⚖️ Vergleich mit ähnlichen Bausteinen
- **F_STRING_AS_INT**: Konvertiert einen `STRING` in einen `INT` (signed 32-bit integer). Im Gegensatz dazu arbeitet `F_STRING_AS_ULINT` mit einem größeren Wertebereich und ohne Vorzeichen.
- **F_STRING_AS_LINT**: Ähnlich, aber für `LINT` (signed 64-bit integer). `F_STRING_AS_ULINT` ist für vorzeichenlose Werte optimiert.

Fazit
Der Funktionsblock `F_STRING_AS_ULINT` bietet eine einfache und effiziente Möglichkeit, Zeichenketten in vorzeichenlose 64-bit Integer-Werte umzuwandeln. Durch seine klare Schnittstelle und direkte Funktionsweise ist er ideal für Anwendungen, die eine schnelle und zuverlässige Konvertierung erfordern.

F_STRING_AS_USINT
• – ◦ • – ◦ • – ◦ •

Einleitung
Der Funktionsblock `F_STRING_AS_USINT` dient der Konvertierung eines `STRING`-Wertes in einen `USINT`-Wert (Unsigned Short Integer). Der Block ist Teil des `iec61131::conversion`-Pakets und ermöglicht die einfache Umwandlung von Zeichenketten in numerische Werte.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingang erwartet einen `STRING`-Wert, der in einen `USINT`-Wert umgewandelt werden soll.

Daten-Ausgänge

- **OUT**: Gibt den konvertierten `USINT`-Wert aus.

Adapter

Der Funktionsblock verfügt über keine Adapter.

Funktionsweise

Bei Auslösung des `REQ`-Ereignisses wird der Algorithmus ausgeführt, der den `STRING`-Wert am Eingang `IN` in einen `USINT`-Wert umwandelt. Das Ergebnis wird am Ausgang `OUT` ausgegeben und das `CNF`-Ereignis signalisiert den Abschluss der Konvertierung.

Technische Besonderheiten

- Der Block verwendet die Funktion `STRING_AS_USINT` zur Konvertierung.
- Die Konvertierung erfolgt synchron mit dem Auslösen des `REQ`-Ereignisses.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar nach Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Umwandlung von Benutzereingaben oder Kommunikationsdaten in numerische Werte.
- Verwendung in Steuerungslogik, wo Zeichenketten in numerische Werte umgewandelt werden müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsblöcken wie `F_STRING_AS_INT` oder `F_STRING_AS_UINT` ist dieser Block speziell für die Umwandlung in `USINT`-Werte optimiert.

Fazit

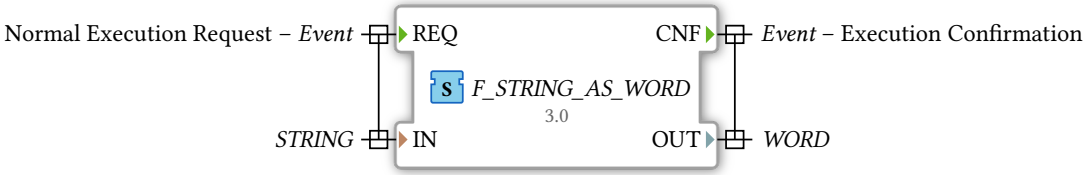
Der `F_STRING_AS_USINT`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, `STRING`-Werte in `USINT`-Werte umzuwandeln. Er ist besonders nützlich in Anwendungen, wo Zeichenketten in numerische Werte konvertiert werden müssen.

F_STRING_AS_WORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_STRING_AS_WORD` dient zur Konvertierung eines `STRING`-Wertes in einen `WORD`-Wert. Diese Funktionalität ist insbesondere in Szenarien nützlich, in denen Zeichenketten in numerische Werte umgewandelt werden müssen, um sie in weiteren Berechnungen oder Steuerungslogiken zu verwenden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderungseingang. Wird verwendet, um die Konvertierung zu starten. Ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Wird nach erfolgreicher Konvertierung ausgelöst. Ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN**: Eingabe vom Typ `STRING`. Enthält den zu konvertierenden Zeichenkettenwert.

Daten-Ausgänge

- **OUT**: Ausgabe vom Typ `WORD`. Enthält das Ergebnis der Konvertierung des Eingabestrings.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, wenn das Ereignis `REQ` empfangen wird. Dabei wird der Eingabestring `IN` mittels der Funktion `STRING_AS_WORD` in einen `WORD`-Wert umgewandelt und das Ergebnis an `OUT` ausgegeben. Anschließend wird das Bestätigungsereignis `CNF` ausgelöst.

Technische Besonderheiten

- Der Funktionsblock ist einfach aufgebaut und enthält nur einen Algorithmus zur Konvertierung.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Zustandsverwaltung.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Ausführung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Konvertierung von Benutzereingaben oder Kommunikationsdaten in numerische Werte.
- Verwendung in Steuerungslogiken, die numerische Verarbeitung erfordern.
- Integration in größere Automatisierungssysteme, die String- und Wort-Daten verarbeiten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen wie `F_STRING_TO_INT` oder `F_STRING_TO_REAL` konvertiert dieser Baustein spezifisch in den `WORD`-Datentyp.
- Einfacher und direkter als Bausteine mit zusätzlichen Funktionen wie Fehlerbehandlung oder Formatierung.

Fazit

Der `F_STRING_AS_WORD`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, Zeichenketten in `WORD`-Werte umzuwandeln. Seine klare Schnittstelle und direkte Funktionsweise machen ihn ideal für den Einsatz in verschiedenen Automatisierungs- und Steuerungsszenarien.

F_STRING_TO_WSTRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_STRING_TO_WSTRING` dient der Konvertierung eines `STRING`-Datentyps in einen `WSTRING`-Datentyp. Dies ist besonders nützlich in Umgebungen, wo eine Umwandlung zwischen schmalen (8-Bit) und breiten (16-Bit) Zeichenketten erforderlich ist, beispielsweise bei der Internationalisierung von Anwendungen oder der Kommunikation mit Systemen, die unterschiedliche Zeichenkodierungen verwenden.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Initiiert die Konvertierung. Bei Auslösung dieses Ereignisses wird der Eingabewert `IN` gelesen und verarbeitet.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieses Ereignis wird zusammen mit dem konvertierten Ausgabewert `OUT` ausgegeben.

Daten-Eingänge

- **IN** (`STRING`): Die Eingabezeichenkette, die in einen `WSTRING` umgewandelt werden soll.

Daten-Ausgänge

- **OUT** (`WSTRING`): Die resultierende breite Zeichenkette nach der Konvertierung.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das `REQ`-Ereignis eintrifft. Dabei wird der `STRING`-Wert von `IN` mittels der Funktion `STRING_TO_WSTRING` in einen `WSTRING` umgewandelt und das Ergebnis an `OUT` ausgegeben. Gleichzeitig wird das `CNF`-Ereignis ausgelöst, um den erfolgreichen Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Die Konvertierung erfolgt ohne zusätzliche Parameter oder Konfiguration.
- Der Block ist als einfacher Funktionsblock (`SimpleFB`) implementiert, was eine effiziente Ausführung garantiert.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung wird bei jedem `REQ`-Ereignis neu durchgeführt.

Anwendungsszenarien

- **Internationalisierung**: Umwandlung von lokalen Zeichenketten in Unicode-fähige `WSTRING`-Werte.
- **Systemkommunikation**: Austausch von Daten zwischen Systemen, die unterschiedliche Zeichensätze verwenden.
- **Datenverarbeitung**: Verarbeitung von Textdaten, die in verschiedenen Kodierungen vorliegen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken ist `F_STRING_TO_WSTRING` spezialisiert auf die Umwandlung zwischen `STRING` und `WSTRING`.
- Andere Blöcke wie `F_WSTRING_TO_STRING` bieten die umgekehrte Funktionalität.

Fazit

Der `F_STRING_TO_WSTRING`-Funktionsblock ist ein einfaches und effizientes Werkzeug zur Konvertierung von Zeichenketten zwischen unterschiedlichen Kodierungen. Seine klare Schnittstelle und direkte Funktionsweise machen ihn besonders geeignet für Anwendungen, die eine schnelle und zuverlässige Umwandlung erfordern.

STRING Konvertierungen

Hier finden Sie Funktionen zur Konvertierung des Datentyps STRING.

🛠️ Zugehörige Übungen

- NumericValue_ID_TO_NVS
- Uebung_020c2_sub
- Uebung_051
- Uebung_052
- Uebung_120
- Uebung_121
- Uebung_122
- Uebung_122b
- Uebung_123
- Uebung_124
- Uebung_125
- Uebung_126
- SIN_TO_PCAN_Callback_DWord
- SIN_TO_PCAN_Callback_Byte
- Uebung_127
- Uebung_128
- Uebung_128b
- Uebung_12x_sub
- Uebung_130
- Uebung_131
- Uebung_132
- Uebung_133
- Uebung_134

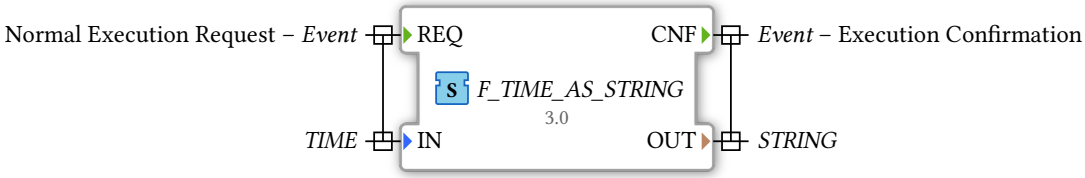
TIME

F_TIME_AS_STRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_TIME_AS_STRING` dient zur Konvertierung eines `TIME`-Wertes in einen `STRING`-Wert. Er ist besonders nützlich, wenn Zeitwerte für die Anzeige oder Protokollierung in einer textuellen Form benötigt werden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Ausführung des Funktionsblocks aus. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert die erfolgreiche Ausführung des Funktionsblocks. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (`TIME`): Der Zeitwert, der in einen String konvertiert werden soll.

Daten-Ausgänge

- **OUT** (`STRING`): Der resultierende String, der den konvertierten Zeitwert repräsentiert.

Adapter

Es sind keine Adapter für diesen Funktionsblock definiert.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` empfangen wird. Dabei wird der `TIME`-Wert am Eingang `IN` in einen `STRING`-Wert umgewandelt und am Ausgang `OUT` ausgegeben. Die Konvertierung erfolgt mithilfe der Funktion `TIME_AS_STRING`. Nach Abschluss der Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher Funktionsblock (SimpleFB) und führt keine komplexen Zustandsübergänge durch.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Parameter.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge. Der Block führt die Konvertierung bei jedem `REQ`-Ereignis durch und signalisiert die Fertigstellung mit `CNF`.

Anwendungsszenarien

- Anzeige von Zeitwerten in Benutzeroberflächen.
- Protokollierung von Zeitstempeln in textuellen Logdateien.
- Integration in Systeme, die Zeitwerte als Strings verarbeiten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen wie `F_INT_AS_STRING` oder `F_REAL_AS_STRING` ist dieser Block speziell für die Konvertierung von `TIME`-Werten optimiert.
- Ähnliche Funktionalität könnte auch mit einem benutzerdefinierten Algorithmus erreicht werden, dieser Block bietet jedoch eine standardisierte und wiederverwendbare Lösung.

Fazit

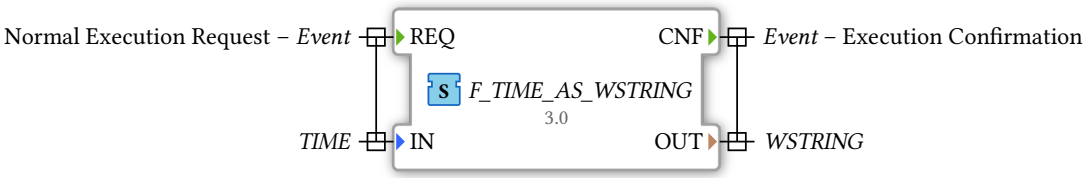
Der `F_TIME_AS_STRING`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, Zeitwerte in Strings umzuwandeln. Seine klare Schnittstelle und direkte Funktionsweise machen ihn zu einem nützlichen Baustein für verschiedene Anwendungen, bei denen Zeitwerte in textueller Form benötigt werden.

F_TIME_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_TIME_AS_WSTRING` dient der Konvertierung eines TIME-Wertes in einen WSTRING-Wert. Dies ist besonders nützlich, wenn Zeitwerte für die Anzeige oder Protokollierung in einer menschenlesbaren Form benötigt werden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsaufruf. Wird dieser Eingang ausgelöst, führt der Funktionsblock die Konvertierung durch.

Ereignis-Ausgänge

- **CNF**: Bestätigung der Ausführung. Wird nach erfolgreicher Konvertierung ausgelöst.

Daten-Eingänge

- **IN**: Eingang für den TIME-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Ausgang für den konvertierten WSTRING-Wert.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Der Funktionsblock konvertiert den eingehenden TIME-Wert (`IN`) in einen WSTRING-Wert (`OUT`), sobald das Ereignis `REQ` ausgelöst wird. Die Konvertierung erfolgt mittels der Funktion `TIME_AS_WSTRING`. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Der Funktionsblock ist einfach aufgebaut und enthält nur einen Algorithmus für die Konvertierung.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Parameter.

Zustandsübersicht

Der Funktionsblock hat keine komplexen Zustände. Er reagiert auf das `REQ`-Ereignis mit der Ausführung des Algorithmus und bestätigt die Ausführung mit dem `CNF`-Ereignis.

Anwendungsszenarien

- Anzeige von Zeitwerten in Benutzeroberflächen.
- Protokollierung von Zeitstempeln in lesbarer Form.
- Integration in Systeme, die Zeitwerte als Zeichenketten benötigen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen ist `F_TIME_AS_WSTRING` spezialisiert auf die Umwandlung von TIME zu WSTRING.
- Andere Bausteine könnten zusätzliche Formatierungsoptionen bieten, dieser Baustein ist jedoch einfach und effizient.

Fazit

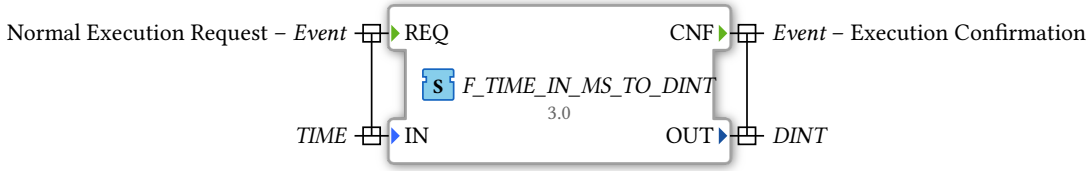
Der `F_TIME_AS_WSTRING` Funktionsblock ist ein einfaches und effektives Werkzeug zur Konvertierung von TIME-Werten in WSTRING. Er eignet sich besonders für Anwendungen, bei denen Zeitwerte in einer lesbaren Form dargestellt oder weiterverarbeitet werden müssen.

F_TIME_IN_MS_TO_DINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_TIME_IN_MS_TO_DINT` dient der Konvertierung eines TIME-Werts in Millisekunden in einen DINT-Wert. Dies ist besonders nützlich, wenn Zeitwerte in numerische Werte umgewandelt werden müssen, um sie in Berechnungen oder für Steuerungszwecke zu verwenden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderungs-Event. Wird verwendet, um die Konvertierung zu starten. Ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigungs-Event. Signalisiert den Abschluss der Konvertierung. Ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (TIME): Der Eingangswert in Millisekunden, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (DINT): Der konvertierte numerische Wert.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem er den TIME-Wert in Millisekunden in einen DINT-Wert umwandelt. Die Konvertierung erfolgt, wenn das Ereignis `REQ` ausgelöst wird. Nach Abschluss der Konvertierung wird das Ereignis `CNF` ausgelöst, und der konvertierte Wert steht am Ausgang `OUT` zur Verfügung.

Technische Besonderheiten

- Der Funktionsblock verwendet einen einfachen Algorithmus zur direkten Konvertierung von TIME in DINT.
- Die Konvertierung ist deterministisch und erfolgt ohne Verzögerung.

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf das `REQ`-Ereignis.
2. **Konvertierungszustand**: Führt die Konvertierung durch und löst das `CNF`-Ereignis aus.

Anwendungsszenarien

- Umwandlung von Zeitwerten für die Verwendung in numerischen Berechnungen.
- Integration in Steuerungssysteme, die numerische Zeitwerte benötigen.
- Verwendung in Protokollierungs- und Überwachungssystemen, die Zeitstempel in numerischer Form speichern.

Umgekehrte Konvertierung (Numerisch zu TIME)

Oft wird nach einer Funktion wie `DINT_TO_TIME` gesucht, um einen numerischen Wert (z.B. 500) wieder in einen Zeitwert (z.B. 500ms) zurückzuwandeln. Einen solchen expliziten Konvertierungsbaustein gibt es in der Regel nicht, da dies elegant über eine Multiplikation gelöst wird.

Verwenden Sie hierfür den Baustein **`F_MULTIME`** (aus der Kategorie `arithmetic`):

- Multiplizieren Sie Ihren numerischen Wert mit der gewünschten Zeitbasis.
- **Beispiel:** `500 * T#1ms = T#500ms`
- Dies ermöglicht eine flexible Skalierung (z.B. * T#1s für Sekunden, * T#100ms für Zehntelsekunden).

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_TIME_TO_UDINT**: Konvertiert TIME in UDINT, während `F_TIME_IN_MS_TO_DINT` in DINT konvertiert.
- **F_TIME_IN_US_TO_DINT**: Konvertiert TIME in Mikrosekunden in DINT, während dieser Baustein Millisekunden verwendet.

Fazit

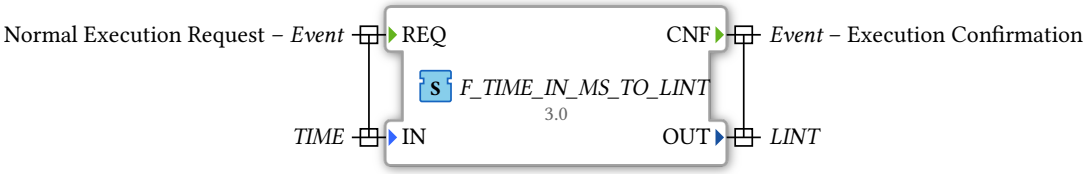
Der Funktionsblock `F_TIME_IN_MS_TO_DINT` ist ein einfaches und effizientes Werkzeug zur Konvertierung von Zeitwerten in Millisekunden in numerische DINT-Werte. Er eignet sich besonders für Anwendungen, die eine schnelle und deterministische Umwandlung erfordern.

F_TIME_IN_MS_TO_LINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_TIME_IN_MS_TO_LINT` dient der Konvertierung eines Zeitwertes (`TIME`) in Millisekunden in einen 64-Bit-Ganzzahlwert (`LINT`). Diese Funktionalität ist besonders nützlich, wenn Zeitwerte für Berechnungen oder Vergleiche in einer numerischen Form benötigt werden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung des Zeitwertes. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingang erwartet einen Zeitwert (`TIME`), der in Millisekunden umgewandelt werden soll.

Daten-Ausgänge

- **OUT**: Gibt den konvertierten Zeitwert als 64-Bit-Ganzzahl (`LINT`) aus.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Bei Auslösung des Ereigniseingangs `REQ` wird der Algorithmus ausgeführt, der den Zeitwert `IN` in Millisekunden umrechnet und das Ergebnis als `LINT` über den Ausgang `OUT` ausgibt. Das Ereignis `CNF` signalisiert den erfolgreichen Abschluss der Konvertierung.

Technische Besonderheiten

- Der Funktionsblock nutzt die Funktion `TIME_IN_MS_TO_LINT` zur Umrechnung des Zeitwertes.
- Die Konvertierung erfolgt mit einer hohen Genauigkeit, da der `LINT`-Datentyp 64 Bit breit ist.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Zeitmessungen, bei denen die Ergebnisse in numerischer Form weiterverarbeitet werden müssen.

- Steuerungsanwendungen, die präzise Zeitintervalle in Millisekunden benötigen.

Umgekehrte Konvertierung (Numerisch zu TIME)

Oft wird nach einer Funktion wie `DINT_TO_TIME` gesucht, um einen numerischen Wert (z.B. 500) wieder in einen Zeitwert (z.B. 500ms) zurückzuwandeln. Einen solchen expliziten Konvertierungsbaustein gibt es in der Regel nicht, da dies elegant über eine Multiplikation gelöst wird.

Verwenden Sie hierfür den Baustein **`F_MULTIME`** (aus der Kategorie `arithmetic`):

- Multiplizieren Sie Ihren numerischen Wert mit der gewünschten Zeitbasis.
- **Beispiel:** `500 * T#1ms = T#500ms`
- Dies ermöglicht eine flexible Skalierung (z.B. * T#1s für Sekunden, * T#100ms für Zehntelsekunden).

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen, die möglicherweise nur in kleinere Ganzzahltypen konvertieren, bietet dieser Baustein eine höhere Genauigkeit durch die Verwendung des `LINT`-Datentyps.
- Ähnliche Bausteine könnten die Zeit in anderen Einheiten (z.B. Sekunden oder Mikrosekunden) konvertieren, während dieser Baustein speziell auf Millisekunden ausgelegt ist.

Fazit

Der `F_TIME_IN_MS_TO_LINT` Funktionsblock ist ein effizientes Werkzeug zur präzisen Konvertierung von Zeitwerten in Millisekunden in einen 64-Bit-Ganzzahlwert. Seine einfache und direkte Funktionsweise macht ihn besonders geeignet für Anwendungen, die genaue Zeitmessungen und -berechnungen erfordern.

F_TIME_IN_MS_TO_LREAL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_TIME_IN_MS_TO_LREAL` dient zur Konvertierung eines TIME-Werts in Millisekunden in einen LREAL-Wert. Dies ist besonders nützlich, wenn Zeitwerte für weitere Berechnungen oder Analysen in einer höheren Präzision benötigt werden.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Löst die Konvertierung aus.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Wird nach erfolgreicher Konvertierung ausgelöst.

Daten-Eingänge

- **IN**: Eingangsvariable vom Typ `TIME`. Der zu konvertierende Zeitwert in Millisekunden.

Daten-Ausgänge

- **OUT**: Ausgangsvariable vom Typ `LREAL`. Der konvertierte Wert als Gleitkommazahl.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem der eingehende TIME-Wert in Millisekunden in einen LREAL-Wert umgewandelt wird. Die Konvertierung erfolgt über den Algorithmus `REQ`, der die Funktion `TIME_IN_MS_TO_LREAL` verwendet.

Technische Besonderheiten

- Der Funktionsblock ist einfach aufgebaut und enthält nur einen Algorithmus für die Konvertierung.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Verzögerung.

Zustandsübersicht

Der Funktionsblock hat keinen internen Zustand. Die Konvertierung wird bei jedem `REQ`-Ereignis neu ausgeführt.

Anwendungsszenarien

- Umwandlung von Zeitwerten für präzise Berechnungen.
- Integration in Steuerungssysteme, die Gleitkommawerte für Zeitmessungen benötigen.

Umgekehrte Konvertierung (Numerisch zu TIME)

Oft wird nach einer Funktion wie `DINT_TO_TIME` gesucht, um einen numerischen Wert (z.B. 500) wieder in einen Zeitwert (z.B. 500ms) zurückzuwandeln. Einen solchen expliziten Konvertierungsbaustein gibt es in der Regel nicht, da dies elegant über eine Multiplikation gelöst wird.

Verwenden Sie hierfür den Baustein **`F_MULTIME`** (aus der Kategorie `arithmetic`):

- Multiplizieren Sie Ihren numerischen Wert mit der gewünschten Zeitbasis.
- **Beispiel:** `500 * T#1ms = T#500ms`
- Dies ermöglicht eine flexible Skalierung (z.B. * T#1s für Sekunden, * T#100ms für Zehntelsekunden).

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen ist dieser speziell auf die Umwandlung von TIME in LREAL ausgelegt und bietet daher eine einfache und direkte Lösung für diesen Anwendungsfall.

Fazit

Der `F_TIME_IN_MS_TO_LREAL` Funktionsblock ist ein einfaches und effizientes Werkzeug zur Konvertierung von Zeitwerten in Millisekunden in Gleitkommazahlen. Seine klare Schnittstelle und direkte Funktionsweise machen ihn zu einer zuverlässigen Komponente in Steuerungssystemen.

F_TIME_IN_MS_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_TIME_IN_MS_TO_UDINT` dient der Konvertierung eines `TIME`-Werts in Millisekunden in einen `UDINT`-Wert. Dies ist besonders nützlich, wenn Zeitwerte in numerischen Berechnungen oder für Steuerungszwecke verwendet werden sollen.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Löst die Konvertierung aus.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Wird nach erfolgreicher Konvertierung ausgelöst.

Daten-Eingänge

- **IN**: Eingang vom Typ `TIME`. Der Zeitwert in Millisekunden, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Ausgang vom Typ `UDINT`. Der konvertierte numerische Wert.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem er den `TIME`-Wert in Millisekunden in einen `UDINT`-Wert umwandelt. Die Konvertierung erfolgt synchron mit dem Auslösen des `REQ`-Ereignisses. Nach Abschluss der Konvertierung wird das `CNF`-Ereignis ausgelöst.

Technische Besonderheiten

- Der Funktionsblock ist einfach und hat keine internen Zustände.
- Die Konvertierung erfolgt direkt und ohne Verzögerung.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge oder komplexe Zustandslogik.

Anwendungsszenarien

- Umwandlung von Zeitwerten für numerische Berechnungen.
- Integration in Steuerungslogiken, die numerische Zeitwerte benötigen.

Umgekehrte Konvertierung (Numerisch zu TIME)

Oft wird nach einer Funktion wie `DINT_TO_TIME` gesucht, um einen numerischen Wert (z.B. 500) wieder in einen Zeitwert (z.B. 500ms) zurückzuwandeln. Einen solchen expliziten Konvertierungsbaustein gibt es in der Regel nicht, da dies elegant über eine Multiplikation gelöst wird.

Verwenden Sie hierfür den Baustein **`F_MULTIME`** (aus der Kategorie `arithmetic`):

- Multiplizieren Sie Ihren numerischen Wert mit der gewünschten Zeitbasis.
- **Beispiel:** `500 * T#1ms = T#500ms`
- Dies ermöglicht eine flexible Skalierung (z.B. * T#1s für Sekunden, * T#100ms für Zehntelsekunden).

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen ist dieser speziell auf die Umwandlung von `TIME` in `UDINT` ausgelegt.
- Andere Bausteine könnten zusätzliche Funktionen wie Skalierung oder Filterung bieten, was hier nicht der Fall ist.

Fazit

Der `F_TIME_IN_MS_TO_UDINT`-Funktionsblock ist ein einfaches und effizientes Werkzeug zur Konvertierung von Zeitwerten in Millisekunden in numerische Werte. Er eignet sich besonders für Anwendungen, bei denen eine schnelle und direkte Umwandlung benötigt wird.

F_TIME_IN_MS_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_TIME_IN_MS_TO_ULINT` dient der Konvertierung eines Zeitwerts in Millisekunden (`TIME`) in einen vorzeichenlosen 64-Bit-Integer-Wert (`ULINT`). Dieser Baustein ist besonders nützlich, wenn Zeitwerte für weitere Berechnungen oder Vergleiche in numerischer Form benötigt werden.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingang erwartet einen Zeitwert vom Typ `TIME`, der in Millisekunden angegeben ist.

Daten-Ausgänge

- **OUT**: Gibt den konvertierten Wert als vorzeichenlosen 64-Bit-Integer (`ULINT`) zurück.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem er den `TIME`-Wert in Millisekunden in einen `ULINT`-Wert umwandelt. Die Konvertierung erfolgt synchron bei Auftreten des `REQ`-Ereignisses. Nach Abschluss der Konvertierung wird das `CNF`-Ereignis ausgelöst, und der konvertierte Wert steht am Ausgang `OUT` zur Verfügung.

Technische Besonderheiten

- Der Baustein nutzt die Funktion `TIME_IN_MS_TO_ULINT` zur Konvertierung.
- Die Implementierung erfolgt in Structured Text (ST).
- Der Baustein ist Teil des Pakets `iec61131::conversion`.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auftreten des `REQ`-Ereignisses.

Anwendungsszenarien

- Umrechnung von Zeitwerten für mathematische Operationen.
- Verwendung in Steuerungsalgorithmen, die numerische Zeitwerte benötigen.
- Integration in Systeme, die Zeitstempel als numerische Werte speichern oder verarbeiten.

Umgekehrte Konvertierung (Numerisch zu TIME)

Oft wird nach einer Funktion wie `DINT_TO_TIME` gesucht, um einen numerischen Wert (z.B. 500) wieder in einen Zeitwert (z.B. 500ms) zurückzuwandeln. Einen solchen expliziten Konvertierungsbaustein gibt es in der Regel nicht, da dies elegant über eine Multiplikation gelöst wird.

Verwenden Sie hierfür den Baustein **`F_MULTIME`** (aus der Kategorie `arithmetic`):

- Multiplizieren Sie Ihren numerischen Wert mit der gewünschten Zeitbasis.
- **Beispiel:** `500 * T#1ms = T#500ms`
- Dies ermöglicht eine flexible Skalierung (z.B. * T#1s für Sekunden, * T#100ms für Zehntelsekunden).

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu generischen Konvertierungsbausteinen ist dieser Funktionsblock spezialisiert auf die Umwandlung von `TIME` in `ULINT`.

- Andere Bausteine könnten zusätzliche Optionen wie die Konvertierung in andere Datentypen bieten, sind jedoch weniger spezifisch.

Fazit

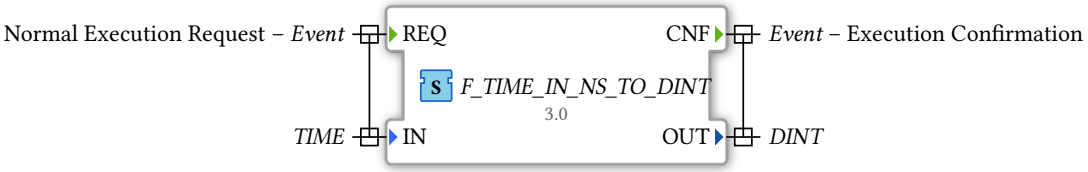
Der `F_TIME_IN_MS_TO_ULINT`-Funktionsblock bietet eine effiziente und spezialisierte Lösung für die Konvertierung von Zeitwerten in Millisekunden in vorzeichenlose 64-Bit-Integer. Seine einfache Schnittstelle und direkte Funktionsweise machen ihn ideal für Anwendungen, die numerische Zeitwerte benötigen.

F_TIME_IN_NS_TO_DINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_TIME_IN_NS_TO_DINT` dient zur Konvertierung eines TIME-Werts in Nanosekunden in einen DINT-Wert. Diese Konvertierung ist nützlich, wenn Zeitwerte in numerischen Berechnungen oder für weitere Verarbeitungsschritte benötigt werden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung des Funktionsblocks. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert die erfolgreiche Beendigung der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (TIME): Der Eingang für den Zeitwert in Nanosekunden, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (DINT): Der Ausgang für den konvertierten numerischen Wert.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt bei einem eingehenden `REQ`-Ereignis die Konvertierung des TIME-Werts `IN` in einen DINT-Wert `OUT` durch. Die Konvertierung erfolgt mittels der Funktion `TIME_IN_NS_TO_DINT`. Nach erfolgreicher Konvertierung wird das `CNF`-Ereignis ausgelöst.

Technische Besonderheiten

- Der Funktionsblock ist einfach aufgebaut und enthält nur einen Algorithmus für die Konvertierung.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Verzögerung.

Zustandsübersicht

1. **Warten auf REQ**: Der Funktionsblock wartet auf ein eingehendes `REQ`-Ereignis.
2. **Konvertierung**: Bei `REQ` wird der TIME-Wert `IN` in DINT konvertiert.
3. **Ausgabe**: Nach erfolgreicher Konvertierung wird das `CNF`-Ereignis ausgelöst.

Anwendungsszenarien

- Umwandlung von Zeitwerten für numerische Berechnungen.
- Integration in Steuerungssysteme, wo Zeitwerte als numerische Werte benötigt werden.

Umgekehrte Konvertierung (Numerisch zu TIME)

Oft wird nach einer Funktion wie `DINT_TO_TIME` gesucht, um einen numerischen Wert (z.B. 500) wieder in einen Zeitwert (z.B. 500ms) zurückzuwandeln. Einen solchen expliziten Konvertierungsbaustein gibt es in der Regel nicht, da dies elegant über eine Multiplikation gelöst wird.

Verwenden Sie hierfür den Baustein **`F_MULTIME`** (aus der Kategorie `arithmetic`):

- Multiplizieren Sie Ihren numerischen Wert mit der gewünschten Zeitbasis.
- **Beispiel:** `500 * T#1ms = T#500ms`
- Dies ermöglicht eine flexible Skalierung (z.B. * T#1s für Sekunden, * T#100ms für Zehntelsekunden).

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen ist `F_TIME_IN_NS_TO_DINT` spezialisiert auf die Umwandlung von TIME in DINT.
- Ähnliche Bausteine könnten andere Datentypen oder zusätzliche Funktionen bieten, wie z.B. Skalierung oder Filterung.

Fazit

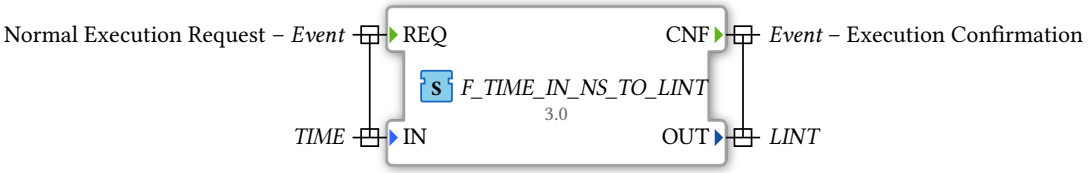
Der Funktionsblock `F_TIME_IN_NS_TO_DINT` bietet eine einfache und effiziente Möglichkeit, Zeitwerte in Nanosekunden in numerische DINT-Werte umzuwandeln. Er ist besonders nützlich in Anwendungen, wo Zeitwerte in Berechnungen oder Steuerungslogiken integriert werden müssen.

F_TIME_IN_NS_TO_LINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_TIME_IN_NS_TO_LINT` dient zur Konvertierung eines Zeitwerts in Nanosekunden (`TIME`) in einen 64-Bit-Ganzzahlwert (`LINT`). Diese Konvertierung ist nützlich, wenn Zeitwerte für Berechnungen oder Vergleiche in einer numerischen Form benötigt werden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung des Funktionsblocks. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingang erwartet einen Zeitwert vom Typ `TIME`, der in Nanosekunden angegeben ist.

Daten-Ausgänge

- **OUT**: Der Ausgang liefert den konvertierten Wert als 64-Bit-Ganzzahl (`LINT`).

Adapter

- Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, wenn das Ereignis `REQ` ausgelöst wird. Der Algorithmus wandelt den Zeitwert `IN` in Nanosekunden in einen `LINT`-Wert um und speichert das Ergebnis in `OUT`. Anschließend wird das Ereignis `CNF` ausgelöst, um den Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Der Funktionsblock ist als *SimpleFB* implementiert, was bedeutet, dass er keinen internen Zustand hat und bei jedem Aufruf von `REQ` neu ausgeführt wird.
- Die Konvertierung erfolgt direkt über die Funktion `TIME_IN_NS_TO_LINT`, die den `TIME`-Wert in einen `LINT`-Wert umwandelt.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge oder -maschinen.

Anwendungsszenarien

- Zeitmessung und -verarbeitung in Steuerungssystemen, wo numerische Operationen auf Zeitwerten erforderlich sind.
- Integration in Systeme, die Zeitstempel in einer numerischen Form speichern oder übertragen müssen.

Umgekehrte Konvertierung (Numerisch zu TIME)

Oft wird nach einer Funktion wie `DINT_TO_TIME` gesucht, um einen numerischen Wert (z.B. 500) wieder in einen Zeitwert (z.B. 500ms) zurückzuwandeln. Einen solchen expliziten Konvertierungsbaustein gibt es in der Regel nicht, da dies elegant über eine Multiplikation gelöst wird.

Verwenden Sie hierfür den Baustein **`F_MULTIME`** (aus der Kategorie `arithmetic`):

- Multiplizieren Sie Ihren numerischen Wert mit der gewünschten Zeitbasis.
- **Beispiel:** `500 * T#1ms = T#500ms`
- Dies ermöglicht eine flexible Skalierung (z.B. * T#1s für Sekunden, * T#100ms für Zehntelsekunden).

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu generischen Konvertierungsblöcken ist dieser Funktionsblock spezialisiert auf die Umwandlung von `TIME` in `LINT`.
- Andere Blöcke könnten zusätzliche Funktionen wie Skalierung oder Formatierung bieten, was hier nicht der Fall ist.

Fazit

Der `F_TIME_IN_NS_TO_LINT`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, Zeitwerte in Nanosekunden in 64-Bit-Ganzzahlen umzuwandeln. Er ist besonders nützlich in Anwendungen, die präzise Zeitmessungen und -berechnungen erfordern.

F_TIME_IN_NS_TO_LREAL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_TIME_IN_NS_TO_LREAL` dient zur Konvertierung eines Zeitwertes in Nanosekunden (`TIME`) in einen Fließkommawert (`LREAL`). Diese Umwandlung ist insbesondere in Steuerungsanwendungen nützlich, wo Zeitwerte in mathematischen Berechnungen verwendet werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung des Funktionsblocks. Der Eingang ist mit dem Dateneingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingang erwartet einen Zeitwert in Nanosekunden (`TIME`).

Daten-Ausgänge

- **OUT**: Gibt den konvertierten Fließkommawert (`LREAL`) zurück.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung des Zeitwertes `IN` von Nanosekunden (`TIME`) in einen Fließkommawert (`LREAL`) durch. Die Umwandlung erfolgt bei jedem Auftreten des Ereignisses `REQ`. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst, und der Ergebniswert steht am Ausgang `OUT` zur Verfügung.

Technische Besonderheiten

- Der Funktionsblock ist als einfacher FB (`SimpleFB`) implementiert.
- Die Konvertierung erfolgt mithilfe der eingebauten Funktion `TIME_IN_NS_TO_LREAL`.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Ausführung erfolgt unmittelbar bei Auftreten des `REQ`-Ereignisses.

Anwendungsszenarien

- Zeitmessungen in Steuerungssystemen, die eine weitere Verarbeitung in mathematischen Ausdrücken erfordern.
- Integration in Regelkreise, wo Zeitwerte als Fließkommazahlen benötigt werden.

Umgekehrte Konvertierung (Numerisch zu TIME)

Oft wird nach einer Funktion wie `DINT_TO_TIME` gesucht, um einen numerischen Wert (z.B. 500) wieder in einen Zeitwert (z.B. 500ms) zurückzuwandeln. Einen solchen expliziten Konvertierungsbaustein gibt es in der Regel nicht, da dies elegant über eine Multiplikation gelöst wird.

Verwenden Sie hierfür den Baustein **`F_MULTIME`** (aus der Kategorie `arithmetic`):

- Multiplizieren Sie Ihren numerischen Wert mit der gewünschten Zeitbasis.
- **Beispiel:** `500 * T#1ms = T#500ms`
- Dies ermöglicht eine flexible Skalierung (z.B. * T#1s für Sekunden, * T#100ms für Zehntelsekunden).

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu generischen Konvertierungsblöcken ist `F_TIME_IN_NS_TO_LREAL` speziell auf die Umwandlung von Zeitwerten in Nanosekunden zu Fließkommazahlen optimiert.
- Andere Blöcke wie `F_TIME_TO_LREAL` könnten eine ähnliche Funktionalität bieten, jedoch ohne die explizite Angabe der Nanosekunden-Einheit.

Fazit

Der Funktionsblock `F_TIME_IN_NS_TO_LREAL` bietet eine effiziente und spezialisierte Lösung für die Konvertierung von Zeitwerten in Fließkommazahlen. Seine einfache Handhabung und direkte Integration machen ihn ideal für Anwendungen, die präzise Zeitverarbeitung erfordern.

F_TIME_IN_NS_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_TIME_IN_NS_TO_UDINT` dient der Konvertierung eines Zeitwertes in Nanosekunden (`TIME`) in einen vorzeichenlosen 32-Bit-Integer-Wert (`UDINT`). Diese Konvertierung ist insbesondere dann nützlich, wenn Zeitwerte für weitere Berechnungen oder Steuerungszwecke in einem numerischen Format benötigt werden.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Ereigniseingang ausgelöst, führt der Funktionsblock die Umwandlung des an `IN` anliegenden Zeitwertes durch.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieses Ereignis wird ausgelöst, sobald der konvertierte Wert an `OUT` ausgegeben wurde.

Daten-Eingänge

- **IN** (`TIME`): Der Eingang für den Zeitwert in Nanosekunden, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (`UDINT`): Der Ausgang, an dem der konvertierte vorzeichenlose 32-Bit-Integer-Wert ausgegeben wird.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem er den an `IN` anliegenden Zeitwert in Nanosekunden (`TIME`) mittels der Funktion `TIME_IN_NS_TO_UDINT` in einen vorzeichenlosen 32-Bit-Integer-Wert (`UDINT`) umwandelt. Die Konvertierung erfolgt synchron zum Auslösen des `REQ`-Ereignisses. Nach erfolgreicher Umwandlung wird das `CNF`-Ereignis ausgelöst und der konvertierte Wert an `OUT` ausgegeben.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne Verzögerung.
- Der Funktionsblock ist für die Verwendung in Echtzeitanwendungen geeignet.
- Es werden keine zusätzlichen Parameter oder Konfigurationen benötigt.

Zustandsübersicht

1. **Idle-Zustand**: Der Funktionsblock wartet auf das `REQ`-Ereignis.
2. **Konvertierungszustand**: Bei Auslösen von `REQ` wird der Zeitwert konvertiert.
3. **Bestätigungszustand**: Nach erfolgreicher Konvertierung wird `CNF` ausgelöst und der Funktionsblock kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- Umwandlung von Zeitstempeln für die Verwendung in numerischen Berechnungen.
- Integration in Steuerungslogik, die Zeitwerte in einem numerischen Format verarbeiten muss.
- Verwendung in Diagnosesystemen zur Darstellung von Zeitdauern in einem lesbaren Format.

Umgekehrte Konvertierung (Numerisch zu TIME)

Oft wird nach einer Funktion wie `DINT_TO_TIME` gesucht, um einen numerischen Wert (z.B. 500) wieder in einen Zeitwert (z.B. 500ms) zurückzuwandeln. Einen solchen expliziten Konvertierungsbaustein gibt es in der Regel nicht, da dies elegant über eine Multiplikation gelöst wird.

Verwenden Sie hierfür den Baustein **`F_MULTIME`** (aus der Kategorie `arithmetic`):

- Multiplizieren Sie Ihren numerischen Wert mit der gewünschten Zeitbasis.
- **Beispiel:** `500 * T#1ms = T#500ms`
- Dies ermöglicht eine flexible Skalierung (z.B. * T#1s für Sekunden, * T#100ms für Zehntelsekunden).

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu generischen Konvertierungsbausteinen ist dieser Funktionsblock speziell auf die Umwandlung von Zeitwerten in Nanosekunden optimiert.
- Ähnliche Bausteine könnten zusätzliche Parameter oder Konfigurationsmöglichkeiten bieten, dieser Funktionsblock ist jedoch auf Einfachheit und Effizienz ausgelegt.

Fazit

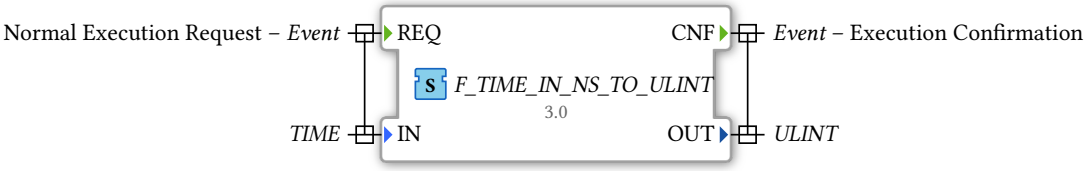
Der `F_TIME_IN_NS_TO_UDINT` Funktionsblock bietet eine einfache und effiziente Möglichkeit, Zeitwerte in Nanosekunden in vorzeichenlose 32-Bit-Integer-Werte umzuwandeln. Durch seine klare Schnittstelle und direkte Funktionsweise eignet er sich ideal für Anwendungen, die eine schnelle und zuverlässige Konvertierung erfordern.

F_TIME_IN_NS_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_TIME_IN_NS_TO_ULINT` dient der Konvertierung eines `TIME`-Werts in Nanosekunden in einen `ULINT`-Wert. Diese Konvertierung ist besonders nützlich, wenn Zeitwerte in numerischen Berechnungen oder für weitere Verarbeitungsschritte benötigt werden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung des Funktionsblocks. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingang erwartet einen `TIME`-Wert in Nanosekunden, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Gibt den konvertierten `ULINT`-Wert aus.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung des `TIME`-Werts in Nanosekunden (`IN`) in einen `ULINT`-Wert (`OUT`) durch, sobald das Ereignis `REQ` ausgelöst wird. Die Konvertierung erfolgt mithilfe der Funktion `TIME_IN_NS_TO_ULINT`. Nach Abschluss der Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Der Funktionsblock ist einfach aufgebaut und enthält nur einen Algorithmus für die Konvertierung.
- Die Konvertierung ist direkt und ohne zusätzliche Verzögerung.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Umwandlung von Zeitwerten für die Verwendung in numerischen Berechnungen.
- Integration in Steuerungssysteme, wo Zeitwerte als numerische Werte weiterverarbeitet werden müssen.

Umgekehrte Konvertierung (Numerisch zu TIME)

Oft wird nach einer Funktion wie `DINT_TO_TIME` gesucht, um einen numerischen Wert (z.B. 500) wieder in einen Zeitwert (z.B. 500ms) zurückzuwandeln. Einen solchen expliziten Konvertierungsbaustein gibt es in der Regel nicht, da dies elegant über eine Multiplikation gelöst wird.

Verwenden Sie hierfür den Baustein **`F_MULTIME`** (aus der Kategorie `arithmetic`):

- Multiplizieren Sie Ihren numerischen Wert mit der gewünschten Zeitbasis.
- **Beispiel:** `500 * T#1ms = T#500ms`
- Dies ermöglicht eine flexible Skalierung (z.B. * T#1s für Sekunden, * T#100ms für Zehntelsekunden).

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen ist dieser speziell auf die Umwandlung von `TIME` in `ULINT` ausgelegt.
- Ähnliche Bausteine könnten andere Datentypen oder Einheiten verwenden, dieser ist jedoch auf Nanosekunden optimiert.

Fazit

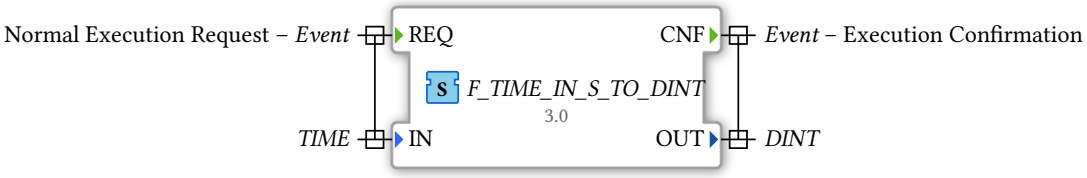
Der `F_TIME_IN_NS_TO_ULINT`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, Zeitwerte in Nanosekunden in `ULINT`-Werte umzuwandeln. Seine klare Schnittstelle und direkte Funktionsweise machen ihn zu einem nützlichen Baustein in der Automatisierungstechnik.

F_TIME_IN_S_TO_DINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_TIME_IN_S_TO_DINT` dient der Konvertierung eines Zeitwerts in Sekunden (`TIME`) in einen vorzeichenbehafteten Ganzzahlwert (`DINT`). Diese Konvertierung ist nützlich, wenn Zeitwerte in numerischen Berechnungen oder für Steuerungszwecke verwendet werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingang erwartet einen Zeitwert vom Typ `TIME` in Sekunden.

Daten-Ausgänge

- **OUT**: Der Ausgang liefert den konvertierten Wert als vorzeichenbehaftete Ganzzahl (`DINT`).

Adapter

Es werden keine Adapter verwendet.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, wenn das Ereignis `REQ` ausgelöst wird. Der Algorithmus wandelt den Zeitwert `IN` in Sekunden in einen `DINT`-Wert um und speichert das Ergebnis in `OUT`. Nach Abschluss der Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Der Funktionsblock nutzt die Funktion `TIME_IN_S_TO_DINT` für die Konvertierung.
- Der Block ist einfach und effizient, da er direkt auf dem ST-Algorithmus basiert.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar nach dem Auslösen von `REQ`.

Anwendungsszenarien

- Umrechnung von Zeitwerten für numerische Berechnungen.
- Integration in Steuerungslogiken, die numerische Zeitwerte benötigen.

Umgekehrte Konvertierung (Numerisch zu TIME)

Oft wird nach einer Funktion wie `DINT_TO_TIME` gesucht, um einen numerischen Wert (z.B. 500) wieder in einen Zeitwert (z.B. 500ms) zurückzuwandeln. Einen solchen expliziten Konvertierungsbaustein gibt es in der Regel nicht, da dies elegant über eine Multiplikation gelöst wird.

Verwenden Sie hierfür den Baustein **`F_MULTIME`** (aus der Kategorie `arithmetic`):

- Multiplizieren Sie Ihren numerischen Wert mit der gewünschten Zeitbasis.
- **Beispiel:** `500 * T#1ms = T#500ms`
- Dies ermöglicht eine flexible Skalierung (z.B. * T#1s für Sekunden, * T#100ms für Zehntelsekunden).

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsblöcken ist dieser speziell auf die Umwandlung von `TIME` in `DINT` ausgelegt.
- Einfacher und direkter als allgemeine Konvertierungsblöcke, die möglicherweise zusätzliche Parameter benötigen.

Fazit

Der `F_TIME_IN_S_TO_DINT` Funktionsblock bietet eine effiziente und einfache Lösung für die Konvertierung von Zeitwerten in Sekunden in vorzeichenbehaftete Ganzzahlen. Seine klare Schnittstelle und direkte Funktionsweise machen ihn ideal für Steuerungsanwendungen.

F_TIME_IN_S_TO_LINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_TIME_IN_S_TO_LINT` dient zur Konvertierung eines Zeitwertes in Sekunden (`TIME`) in einen 64-Bit Ganzzahlwert (`LINT`). Diese Konvertierung ist besonders nützlich, wenn Zeitwerte für Berechnungen oder Vergleiche in einer numerischen Form benötigt werden.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung des Zeitwertes. Dieser Ereigniseingang ist mit dem Dateneingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ereignisausgang ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingang für den Zeitwert in Sekunden (`TIME`), der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgang für den konvertierten 64-Bit Ganzzahlwert (`LINT`).

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem er den Zeitwert in Sekunden (`TIME`) in einen 64-Bit Ganzzahlwert (`LINT`) umwandelt. Die Konvertierung erfolgt bei Auslösung des `REQ`-Ereignisses und das Ergebnis wird über den `OUT`-Datenausgang ausgegeben, gefolgt von einem `CNF`-Ereignis.

Technische Besonderheiten

- Der Funktionsblock verwendet den Standard `61499-1`.
- Die Konvertierung erfolgt mittels der Funktion `TIME_IN_S_TO_LINT`, die im Algorithmus des Funktionsblocks implementiert ist.
- Der Funktionsblock ist Teil des Pakets `iec61131::conversion`.

Zustandsübersicht

Der Funktionsblock besitzt keine komplexen Zustände. Er reagiert auf das `REQ`-Ereignis, führt die Konvertierung durch und signalisiert den Abschluss mit dem `CNF`-Ereignis.

Anwendungsszenarien

- Umwandlung von Zeitwerten für numerische Berechnungen.
- Integration in Steuerungssysteme, die Zeitwerte in einer numerischen Form benötigen.
- Verwendung in Systemen, die eine hohe Präzision bei der Zeitmessung erfordern.

Umgekehrte Konvertierung (Numerisch zu TIME)

Oft wird nach einer Funktion wie `DINT_TO_TIME` gesucht, um einen numerischen Wert (z.B. 500) wieder in einen Zeitwert (z.B. 500ms) zurückzuwandeln. Einen solchen expliziten Konvertierungsbaustein gibt es in der Regel nicht, da dies elegant über eine Multiplikation gelöst wird.

Verwenden Sie hierfür den Baustein **`F_MULTIME`** (aus der Kategorie `arithmetic`):

- Multiplizieren Sie Ihren numerischen Wert mit der gewünschten Zeitbasis.
- **Beispiel:** `500 * T#1ms = T#500ms`
- Dies ermöglicht eine flexible Skalierung (z.B. * T#1s für Sekunden, * T#100ms für Zehntelsekunden).

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen, die möglicherweise andere Datentypen unterstützen, ist dieser Funktionsblock speziell auf die Konvertierung von `TIME` zu `LINT` ausgelegt.
- Andere Bausteine könnten eine Konvertierung in andere Ganzzahltypen (z.B. `INT` oder `DINT`) anbieten, aber dieser Baustein bietet eine höhere Präzision durch die Verwendung von `LINT`.

Fazit

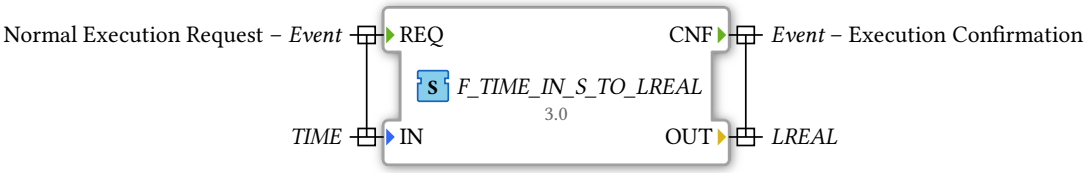
Der Funktionsblock `F_TIME_IN_S_TO_LINT` ist ein effizientes Werkzeug zur Konvertierung von Zeitwerten in Sekunden in 64-Bit Ganzzahlen. Seine einfache und direkte Funktionsweise macht ihn zu einer zuverlässigen Komponente in Steuerungssystemen, die präzise Zeitmessungen und -berechnungen erfordern.

F_TIME_IN_S_TO_LREAL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_TIME_IN_S_TO_LREAL` dient zur Konvertierung eines Zeitwerts in Sekunden (`TIME`) in einen Fließkommawert (`LREAL`). Diese Konvertierung ist nützlich, wenn Zeitwerte in mathematischen Berechnungen oder für weitere Verarbeitungsschritte benötigt werden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingang erwartet einen Zeitwert vom Typ `TIME` in Sekunden, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Gibt den konvertierten Wert als Fließkommazahl (`LREAL`) aus.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `TIME` zu `LREAL` durch, sobald das Ereignis `REQ` ausgelöst wird. Die Konvertierung erfolgt mithilfe der Funktion `TIME_IN_S_TO_LREAL`, die den Zeitwert in Sekunden in einen entsprechenden Fließkommawert umwandelt. Nach Abschluss der Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher Algorithmus (`SimpleFB`) und führt keine komplexen Zustandsübergänge durch.
- Die Konvertierung ist direkt und ohne zusätzliche Verzögerung.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge. Die Konvertierung erfolgt sofort bei Auslösung von `REQ`.

Anwendungsszenarien

- Umrechnung von Zeitwerten für mathematische Operationen.
- Integration in Steuerungslogik, wo Zeitwerte als Fließkommazahlen benötigt werden.
- Verwendung in Regelungstechnik oder Datenverarbeitung, wo präzise Zeitmessungen erforderlich sind.

Umgekehrte Konvertierung (Numerisch zu TIME)

Oft wird nach einer Funktion wie `DINT_TO_TIME` gesucht, um einen numerischen Wert (z.B. 500) wieder in einen Zeitwert (z.B. 500ms) zurückzuwandeln. Einen solchen expliziten Konvertierungsbaustein gibt es in der Regel nicht, da dies elegant über eine Multiplikation gelöst wird.

Verwenden Sie hierfür den Baustein **`F_MULTIME`** (aus der Kategorie `arithmetic`):

- Multiplizieren Sie Ihren numerischen Wert mit der gewünschten Zeitbasis.
- **Beispiel:** `500 * T#1ms = T#500ms`
- Dies ermöglicht eine flexible Skalierung (z.B. * T#1s für Sekunden, * T#100ms für Zehntelsekunden).

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu anderen Konvertierungsbausteinen, die möglicherweise zusätzliche Parameter oder Zustände benötigen, ist `F_TIME_IN_S_TO_LREAL` auf eine spezifische und direkte Konvertierung ausgelegt.
- Ähnliche Bausteine könnten z.B. `TIME_TO_STRING` oder `TIME_TO_INT` sein, die jedoch andere Ausgabetypen liefern.

Fazit

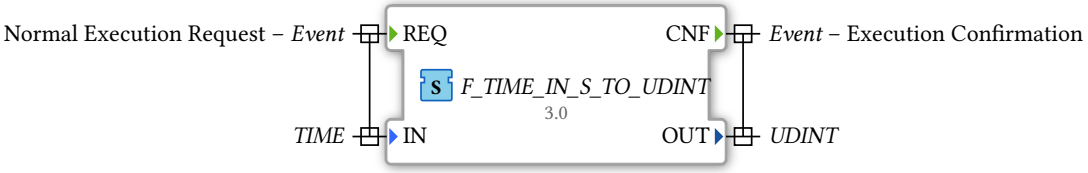
Der Funktionsblock `F_TIME_IN_S_TO_LREAL` bietet eine effiziente und direkte Möglichkeit, Zeitwerte in Sekunden in Fließkommazahlen umzuwandeln. Durch seine einfache Struktur und schnelle Ausführung eignet er sich ideal für Anwendungen, die eine präzise und unkomplizierte Konvertierung erfordern.

F_TIME_IN_S_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_TIME_IN_S_TO_UDINT` dient der Konvertierung eines TIME-Werts in Sekunden in einen UDINT-Wert. Diese Konvertierung ist nützlich, wenn Zeitwerte in numerischen Operationen oder für weitere Verarbeitungsschritte benötigt werden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Löst die Konvertierung aus.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Wird nach erfolgreicher Konvertierung ausgelöst.

Daten-Eingänge

- **IN**: TIME-Wert in Sekunden, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Konvertierter UDINT-Wert.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock konvertiert den eingehenden TIME-Wert (IN) in Sekunden in einen UDINT-Wert (OUT). Die Konvertierung erfolgt durch den Aufruf der Funktion `TIME_IN_S_TO_UDINT(IN)` im Algorithmus. Bei Auslösung des REQ-Ereignisses wird die Konvertierung durchgeführt und das CNF-Ereignis ausgelöst.

Technische Besonderheiten

- Der Funktionsblock ist einfach und effizient, da er direkt die eingebaute Funktion `TIME_IN_S_TO_UDINT` nutzt.
- Keine zusätzlichen Zustände oder komplexe Logik erforderlich.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt sofort bei Auslösung des REQ-Ereignisses.

Anwendungsszenarien

- Konvertierung von Zeitwerten für numerische Berechnungen.
- Integration in Steuerungssysteme, wo Zeitwerte als numerische Werte benötigt werden.

Umgekehrte Konvertierung (Numerisch zu TIME)

Oft wird nach einer Funktion wie `DINT_TO_TIME` gesucht, um einen numerischen Wert (z.B. 500) wieder in einen Zeitwert (z.B. 500ms) zurückzuwandeln. Einen solchen expliziten Konvertierungsbaustein gibt es in der Regel nicht, da dies elegant über eine Multiplikation gelöst wird.

Verwenden Sie hierfür den Baustein **`F_MULTIME`** (aus der Kategorie `arithmetic`):

- Multiplizieren Sie Ihren numerischen Wert mit der gewünschten Zeitbasis.
- **Beispiel:** `500 * T#1ms = T#500ms`
- Dies ermöglicht eine flexible Skalierung (z.B. * T#1s für Sekunden, * T#100ms für Zehntelsekunden).

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen ist dieser speziell für die Umwandlung von TIME in UDINT optimiert.
- Andere Bausteine könnten zusätzliche Funktionen wie Skalierung oder Grenzwertprüfung bieten, was hier nicht der Fall ist.

Fazit

Der `F_TIME_IN_S_TO_UDINT` Funktionsblock ist ein einfacher und effizienter Baustein zur Konvertierung von TIME-Werten in UDINT. Er eignet sich ideal für Anwendungen, bei denen Zeitwerte in numerischer Form benötigt werden.

F_TIME_IN_S_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_TIME_IN_S_TO_ULINT` dient der Konvertierung eines TIME-Werts in Sekunden in einen ULINT-Wert. Dies ist besonders nützlich, wenn Zeitwerte in numerische Formate umgewandelt werden müssen, um sie in Berechnungen oder für weitere Verarbeitungsschritte zu verwenden.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderungs-Event. Dieser Event löst die Konvertierung aus und ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Bestätigungs-Event nach erfolgreicher Ausführung. Dieser Event wird zusammen mit dem Daten-Ausgang `OUT` ausgegeben.

Daten-Eingänge

- **IN**: Eingang vom Typ `TIME`, der den zu konvertierenden Zeitwert in Sekunden enthält.

Daten-Ausgänge

- **OUT**: Ausgang vom Typ `ULINT`, der den konvertierten numerischen Wert enthält.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem er den TIME-Wert (in Sekunden) in einen ULINT-Wert umwandelt. Die Konvertierung erfolgt bei Auftreten des `REQ`-Events, und das Ergebnis wird über den `OUT`-Ausgang ausgegeben, begleitet vom `CNF`-Event.

Technische Besonderheiten

- Der Funktionsblock verwendet den Standardalgorithmus `TIME_IN_S_TO_ULINT` für die Konvertierung.
- Der FB ist Teil des Pakets `iec61131::conversion`.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Ausführung erfolgt unmittelbar bei Auftreten des `REQ`-Events.

Anwendungsszenarien

- Umwandlung von Zeitwerten für die Verwendung in numerischen Berechnungen.
- Integration in Steuerungssysteme, die Zeitwerte in numerischer Form verarbeiten müssen.

Umgekehrte Konvertierung (Numerisch zu TIME)

Oft wird nach einer Funktion wie `DINT_TO_TIME` gesucht, um einen numerischen Wert (z.B. 500) wieder in einen Zeitwert (z.B. 500ms) zurückzuwandeln. Einen solchen expliziten Konvertierungsbaustein gibt es in der Regel nicht, da dies elegant über eine Multiplikation gelöst wird.

Verwenden Sie hierfür den Baustein **`F_MULTIME`** (aus der Kategorie `arithmetic`):

- Multiplizieren Sie Ihren numerischen Wert mit der gewünschten Zeitbasis.
- **Beispiel:** `500 * T#1ms = T#500ms`
- Dies ermöglicht eine flexible Skalierung (z.B. * T#1s für Sekunden, * T#100ms für Zehntelsekunden).

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu generischen Konvertierungsblöcken ist dieser FB spezialisiert auf die Umwandlung von TIME in ULINT.
- Andere Blöcke könnten unterschiedliche Ausgabetypen oder zusätzliche Funktionen bieten, wie z.B. die Umwandlung in andere numerische Formate.

Fazit

Der `F_TIME_IN_S_TO_ULINT` Funktionsblock bietet eine einfache und effiziente Möglichkeit, Zeitwerte in numerische ULINT-Werte umzuwandeln. Seine klare Schnittstelle und direkte Funktionsweise machen ihn ideal für Anwendungen, die solche Konvertierungen erfordern.

F_TIME_IN_US_TO_DINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_TIME_IN_US_TO_DINT` dient zur Konvertierung eines TIME-Wertes in Mikrosekunden (µs) in einen DINT-Wert. Diese Konvertierung ist insbesondere dann nützlich, wenn Zeitwerte in numerischen Berechnungen oder für Steuerungszwecke verwendet werden sollen.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Bei diesem Ereignis wird der Eingangswert `IN` verarbeitet und das Ergebnis am Ausgang `OUT` ausgegeben.

Ereignis-Ausgänge

- **CNF**: Signalisiert die erfolgreiche Beendigung der Konvertierung. Dieses Ereignis wird zusammen mit dem konvertierten Wert am Ausgang `OUT` ausgegeben.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `TIME`, der in Mikrosekunden angegeben ist und in einen DINT-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der konvertierte Wert vom Typ `DINT`, der das Ergebnis der Umwandlung des TIME-Wertes in Mikrosekunden darstellt.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem der TIME-Wert in Mikrosekunden in einen DINT-Wert umgewandelt wird. Die Konvertierung erfolgt synchron mit dem Eintreffen des `REQ`-Ereignisses. Das Ergebnis wird sofort am Ausgang `OUT` ausgegeben und durch das `CNF`-Ereignis bestätigt.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne Verzögerung.
- Der Funktionsblock ist für die Verwendung in Echtzeitanwendungen geeignet.
- Es werden keine zusätzlichen Parameter oder Konfigurationen benötigt.

Zustandsübersicht

1. **Idle-Zustand**: Der Funktionsblock wartet auf das `REQ`-Ereignis.
2. **Aktiv-Zustand**: Bei Eintreffen von `REQ` wird die Konvertierung durchgeführt und das Ergebnis ausgegeben.
3. **Bestätigungs-Zustand**: Das `CNF`-Ereignis wird ausgelöst, um die erfolgreiche Konvertierung zu signalisieren.

Anwendungsszenarien

- Umwandlung von Zeitwerten für die Verwendung in numerischen Berechnungen.
- Integration in Steuerungsalgorithmen, die Zeitwerte in einer anderen Form benötigen.
- Verwendung in Systemen, die Zeitstempel oder Zeitdauern in Mikrosekunden verarbeiten.

Umgekehrte Konvertierung (Numerisch zu TIME)

Oft wird nach einer Funktion wie `DINT_TO_TIME` gesucht, um einen numerischen Wert (z.B. 500) wieder in einen Zeitwert (z.B. 500ms) zurückzuwandeln. Einen solchen expliziten Konvertierungsbaustein gibt es in der Regel nicht, da dies elegant über eine Multiplikation gelöst wird.

Verwenden Sie hierfür den Baustein **`F_MULTIME`** (aus der Kategorie `arithmetic`):

- Multiplizieren Sie Ihren numerischen Wert mit der gewünschten Zeitbasis.
- **Beispiel:** `500 * T#1ms = T#500ms`
- Dies ermöglicht eine flexible Skalierung (z.B. * T#1s für Sekunden, * T#100ms für Zehntelsekunden).

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsbausteinen ist dieser Funktionsblock spezialisiert auf die Umwandlung von TIME in DINT.
- Andere Bausteine könnten zusätzliche Funktionen wie Skalierung oder Filterung bieten, was hier nicht der Fall ist.

Fazit

Der `F_TIME_IN_US_TO_DINT` Funktionsblock ist ein einfaches und effizientes Werkzeug zur Konvertierung von Zeitwerten in Mikrosekunden in einen DINT-Wert. Seine einfache Schnittstelle und direkte Funktionsweise machen ihn ideal für Anwendungen, die eine schnelle und zuverlässige Umwandlung benötigen.

F_TIME_IN_US_TO_LINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_TIME_IN_US_TO_LINT` dient der Konvertierung eines TIME-Wertes in Mikrosekunden (µs) in einen LINT-Wert (64-Bit Ganzzahl). Diese Konvertierung ist nützlich, wenn Zeitwerte in numerischen Berechnungen oder für weitere Verarbeitungsschritte benötigt werden.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ` (Normal Execution Request): Startet die Ausführung des Funktionsblocks. Wird mit dem Eingang `IN` verknüpft.

Ereignis-Ausgänge

- `CNF` (Execution Confirmation): Signalisiert den Abschluss der Konvertierung. Wird mit dem Ausgang `OUT` verknüpft.

Daten-Eingänge

- `IN` (TIME): Der Eingangswert in Mikrosekunden (µs), der in einen LINT-Wert umgewandelt werden soll.

Daten-Ausgänge

- `OUT` (LINT): Der konvertierte 64-Bit Ganzzahlwert.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem er den TIME-Eingangswert (`IN`) in Mikrosekunden in einen LINT-Wert (`OUT`) umwandelt. Die Konvertierung erfolgt bei Auslösung des `REQ`-Ereignisses und wird durch das `CNF`-Ereignis bestätigt.

Technische Besonderheiten

- Der Algorithmus verwendet die Funktion `TIME_IN_US_TO_LINT`, um die Konvertierung durchzuführen.
- Der Funktionsblock ist für die Verwendung in Echtzeitanwendungen geeignet.

Zustandsübersicht

Der Funktionsblock hat einen einfachen Zustandsautomaten:

1. **Idle-Zustand**: Wartet auf das `REQ`-Ereignis.
2. **Ausführungszustand**: Führt die Konvertierung durch und löst das `CNF`-Ereignis aus.

Anwendungsszenarien

- Zeitmessung und -verarbeitung in Steuerungssystemen.
- Integration von Zeitwerten in numerische Berechnungen.
- Protokollierung und Analyse von Zeitdauern.

Umgekehrte Konvertierung (Numerisch zu TIME)

Oft wird nach einer Funktion wie `DINT_TO_TIME` gesucht, um einen numerischen Wert (z.B. 500) wieder in einen Zeitwert (z.B. 500ms) zurückzuwandeln. Einen solchen expliziten Konvertierungsbaustein gibt es in der Regel nicht, da dies elegant über eine Multiplikation gelöst wird.

Verwenden Sie hierfür den Baustein **`F_MULTIME`** (aus der Kategorie `arithmetic`):

- Multiplizieren Sie Ihren numerischen Wert mit der gewünschten Zeitbasis.
- **Beispiel:** `500 * T#1ms = T#500ms`
- Dies ermöglicht eine flexible Skalierung (z.B. * T#1s für Sekunden, * T#100ms für Zehntelsekunden).

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen ist dieser speziell auf die Umwandlung von TIME in LINT optimiert.
- Ähnliche Bausteine könnten andere Datentypen oder Einheiten verwenden, z.B. Millisekunden statt Mikrosekunden.

Fazit

Der `F_TIME_IN_US_TO_LINT` Funktionsblock bietet eine effiziente und zuverlässige Möglichkeit, Zeitwerte in Mikrosekunden in 64-Bit Ganzahlen umzuwandeln. Seine einfache Schnittstelle und klare Funktionsweise machen ihn ideal für eine Vielzahl von Anwendungen in der Automatisierungstechnik.

F_TIME_IN_US_TO_LREAL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_TIME_IN_US_TO_LREAL` dient zur Konvertierung eines Zeitwertes in Mikrosekunden (`TIME`) in eine Fließkommazahl mit doppelter Genauigkeit (`LREAL`). Diese Umwandlung ist insbesondere in Steuerungsanwendungen nützlich, wo Zeitwerte für weitere Berechnungen benötigt werden.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Auslöser für die Konvertierung. Bei diesem Ereignis wird der Eingangswert `IN` verarbeitet und das Ergebnis an `OUT` ausgegeben.

Ereignis-Ausgänge

- **CNF**: Bestätigt die erfolgreiche Ausführung der Konvertierung. Dieses Ereignis wird zusammen mit dem konvertierten Wert `OUT` ausgegeben.

Daten-Eingänge

- **IN** (`TIME`): Der Eingangswert in Mikrosekunden, der in einen `LREAL`-Wert umgewandelt werden soll.

Daten-Ausgänge

- **OUT** (`LREAL`): Der konvertierte Wert als Fließkommazahl mit doppelter Genauigkeit.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem er den `TIME`-Eingangswert `IN` mittels der Funktion `TIME_IN_US_TO_LREAL` in einen `LREAL`-Wert umwandelt. Das Ergebnis wird an den Ausgang `OUT` übergeben und durch das Ereignis `CNF` bestätigt.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne zusätzliche Verzögerung.
- Der Funktionsblock ist für Echtzeitanwendungen optimiert und unterstützt präzise Zeitmessungen.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung wird bei jedem Auftreten des `REQ`-Ereignisses durchgeführt.

Anwendungsszenarien

- Umwandlung von Zeitdauern für mathematische Berechnungen.
- Integration in Steuerungssysteme, die Fließkommawerte für Regelungsalgorithmen benötigen.

Umgekehrte Konvertierung (Numerisch zu TIME)

Oft wird nach einer Funktion wie `DINT_TO_TIME` gesucht, um einen numerischen Wert (z.B. 500) wieder in einen Zeitwert (z.B. 500ms) zurückzuwandeln. Einen solchen expliziten Konvertierungsbaustein gibt es in der Regel nicht, da dies elegant über eine Multiplikation gelöst wird.

Verwenden Sie hierfür den Baustein **`F_MULTIME`** (aus der Kategorie `arithmetic`):

- Multiplizieren Sie Ihren numerischen Wert mit der gewünschten Zeitbasis.
- **Beispiel:** `500 * T#1ms = T#500ms`
- Dies ermöglicht eine flexible Skalierung (z.B. * T#1s für Sekunden, * T#100ms für Zehntelsekunden).

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu generischen Konvertierungsblöcken ist dieser Funktionsblock spezialisiert auf die Umwandlung von `TIME` in `LREAL` und bietet daher eine optimierte Leistung.
- Andere Blöcke könnten zusätzliche Parameter oder Zustände besitzen, was hier nicht der Fall ist.

Fazit

Der `F_TIME_IN_US_TO_LREAL`-Funktionsblock ist ein effizientes Werkzeug zur präzisen Konvertierung von Zeitwerten in Fließkommazahlen. Seine einfache und direkte Funktionsweise macht ihn ideal für Anwendungen, die schnelle und genaue Umrechnungen erfordern.

F_TIME_IN_US_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_TIME_IN_US_TO_UDINT` dient der Konvertierung eines TIME-Werts in Mikrosekunden (µs) in einen UDINT-Wert. Diese Konvertierung ist insbesondere in Anwendungen nützlich, bei denen Zeitwerte in numerischer Form weiterverarbeitet werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingang vom Typ `TIME`, der den zu konvertierenden Zeitwert in Mikrosekunden enthält.

Daten-Ausgänge

- **OUT**: Der Ausgang vom Typ `UDINT`, der das Ergebnis der Konvertierung enthält.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock konvertiert den an `IN` übergebenen TIME-Wert (in µs) in einen UDINT-Wert. Die Konvertierung erfolgt durch den Aufruf der Funktion `TIME_IN_US_TO_UDINT(IN)`. Nach erfolgreicher Konvertierung wird das Ergebnis an `OUT` ausgegeben und das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Der Funktionsblock ist einfach aufgebaut und enthält nur einen Algorithmus (`REQ`), der die Konvertierung durchführt.
- Es gibt keine Zustandsmaschine oder komplexe Logik, da der Block ausschließlich für die Konvertierung zuständig ist.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge oder -maschinen. Der Block führt die Konvertierung bei jedem `REQ`-Ereignis durch und signalisiert den Abschluss mit `CNF`.

Anwendungsszenarien

- Zeitmessung und -verarbeitung in Steuerungssystemen.
- Umwandlung von Zeitwerten für die Speicherung oder Übertragung in numerischer Form.
- Integration in Systeme, die Zeitwerte in µs als UDINT verarbeiten müssen.

Umgekehrte Konvertierung (Numerisch zu TIME)

Oft wird nach einer Funktion wie `DINT_TO_TIME` gesucht, um einen numerischen Wert (z.B. 500) wieder in einen Zeitwert (z.B. 500ms) zurückzuwandeln. Einen solchen expliziten Konvertierungsbaustein gibt es in der Regel nicht, da dies elegant über eine Multiplikation gelöst wird.

Verwenden Sie hierfür den Baustein **`F_MULTIME`** (aus der Kategorie `arithmetic`):

- Multiplizieren Sie Ihren numerischen Wert mit der gewünschten Zeitbasis.
- **Beispiel:** `500 * T#1ms = T#500ms`
- Dies ermöglicht eine flexible Skalierung (z.B. * T#1s für Sekunden, * T#100ms für Zehntelsekunden).

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen ist `F_TIME_IN_US_TO_UDINT` spezialisiert auf die Umwandlung von TIME in UDINT.
- Ähnliche Blöcke könnten andere Zeitformate oder andere Ziel-Datentypen unterstützen, z.B. TIME in REAL oder TIME in STRING.

Fazit

Der Funktionsblock `F_TIME_IN_US_TO_UDINT` bietet eine einfache und effiziente Möglichkeit, Zeitwerte in Mikrosekunden in UDINT-Werte umzuwandeln. Durch seinen schlanken Aufbau ist er besonders für Anwendungen geeignet, bei denen Performance und Einfachheit im Vordergrund stehen.

F_TIME_IN_US_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_TIME_IN_US_TO_ULINT` dient der Konvertierung eines Zeitwerts in Mikrosekunden (`TIME`) in einen vorzeichenlosen 64-Bit-Integer-Wert (`ULINT`). Dies ist besonders nützlich, wenn Zeitwerte in numerische Berechnungen oder für weitere Verarbeitungsschritte umgewandelt werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN**: Der Eingang erwartet einen Zeitwert vom Typ `TIME` (in Mikrosekunden), der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Gibt den konvertierten Wert als vorzeichenlosen 64-Bit-Integer (`ULINT`) zurück.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock nutzt den Algorithmus `REQ`, um den eingehenden `TIME`-Wert in Mikrosekunden in einen `ULINT`-Wert umzuwandeln. Die Konvertierung erfolgt durch die Funktion `TIME_IN_US_TO_ULINT(IN)`. Sobal die Konvertierung abgeschlossen ist, wird das Ereignis `CNF` ausgelöst, um den erfolgreichen Abschluss zu signalisieren.

Technische Besonderheiten

- Der Funktionsblock ist einfach aufgebaut und enthält nur einen Algorithmus.
- Die Konvertierung ist direkt und ohne zusätzliche Parameter oder Zustandsverwaltung.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine komplexen Zustandsübergänge. Der Block reagiert auf das `REQ`-Ereignis, führt die Konvertierung durch und signalisiert den Abschluss mit `CNF`.

Anwendungsszenarien

- Zeitmessung und -verarbeitung in Steuerungssystemen.
- Umwandlung von Zeitwerten für die Verwendung in Berechnungen oder Logging-Systemen.
- Integration in größere Steuerungsalgorithmen, die numerische Zeitwerte benötigen.

Umgekehrte Konvertierung (Numerisch zu TIME)

Oft wird nach einer Funktion wie `DINT_TO_TIME` gesucht, um einen numerischen Wert (z.B. 500) wieder in einen Zeitwert (z.B. 500ms) zurückzuwandeln. Einen solchen expliziten Konvertierungsbaustein gibt es in der Regel nicht, da dies elegant über eine Multiplikation gelöst wird.

Verwenden Sie hierfür den Baustein **`F_MULTIME`** (aus der Kategorie `arithmetic`):

- Multiplizieren Sie Ihren numerischen Wert mit der gewünschten Zeitbasis.
- **Beispiel:** `500 * T#1ms = T#500ms`
- Dies ermöglicht eine flexible Skalierung (z.B. * T#1s für Sekunden, * T#100ms für Zehntelsekunden).

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken ist `F_TIME_IN_US_TO_ULINT` spezialisiert auf die Umwandlung von `TIME` zu `ULINT`.
- Andere Blöcke könnten zusätzliche Funktionen wie Skalierung oder Filterung anbieten, was hier nicht der Fall ist.

Fazit

Der Funktionsblock `F_TIME_IN_US_TO_ULINT` ist ein einfaches, aber effektives Werkzeug zur Konvertierung von Zeitwerten in Mikrosekunden in numerische Werte. Seine Einfachheit und direkte Funktionsweise machen ihn zu einer zuverlässigen Komponente in Steuerungsanwendungen, die präzise Zeitverarbeitung erfordern.

TIME Konvertierungen

🎧 Podcast

- Die drei Timer der DIN EN 61131-3 entschlüsselt – TP, TON & TOF präzise erklärt

- E_TOF vs. FB_TOF: Der Event-Timer, der nicht zyklisch tickt – Revolution für Automatisierungssysteme?
- E_TON in der Industrieautomation: Wie ein simpler Timer Sicherheit und Stabilität schafft
- Ereignisgesteuerte Timer: Wann welcher am Lüfter Sinn macht – ETON, ETOF, ETP & mehr entschlüsselt
- FB_TOF und E_TOF: Verzögerungstimer in IEC 61131-3 und 61499

🛠️ Zugehörige Übungen

- Uebung_072c

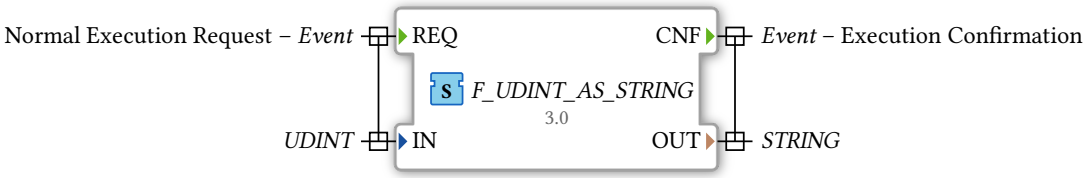
UDINT

F_UDINT_AS_STRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UDINT_AS_STRING` dient der Konvertierung eines vorzeichenlosen 32-Bit-Integer-Werts (UDINT) in einen String (STRING). Dieser Baustein ist besonders nützlich, wenn numerische Werte für die Anzeige oder Protokollierung in eine lesbare Textform umgewandelt werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der vorzeichenlose 32-Bit-Integer-Wert (UDINT), der in einen String umgewandelt werden soll.

Daten-Ausgänge

- **OUT**: Der resultierende String (STRING) nach der Konvertierung.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Bei Auslösung des Ereignisses `REQ` wird der Algorithmus ausgeführt, der den Wert des Eingangs `IN` in einen String umwandelt und das Ergebnis an `OUT` ausgibt. Anschließend wird das Ereignis `CNF` ausgelöst, um den erfolgreichen Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Der Funktionsblock verwendet die Funktion `UDINT_AS_STRING` zur Konvertierung.
- Die Implementierung erfolgt in Structured Text (ST).

Zustandsübersicht

1. **Initialisierung**: Der Funktionsblock ist bereit und wartet auf das `REQ`-Ereignis.
2. **Konvertierung**: Bei Empfang von `REQ` wird der Wert von `IN` konvertiert und an `OUT` ausgegeben.
3. **Bestätigung**: Das `CNF`-Ereignis wird ausgelöst, um den Abschluss der Konvertierung zu signalisieren.

Anwendungsszenarien

- Anzeige von numerischen Werten in Benutzeroberflächen.
- Protokollierung von Messwerten in Textdateien oder Datenbanken.
- Vorbereitung von Daten für die Kommunikation über Textprotokolle (z.B. HTTP, MQTT).

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsbausteinen bietet `F_UDINT_AS_STRING` eine spezialisierte Lösung für die Umwandlung von UDINT zu STRING.
- Andere Bausteine könnten zusätzliche Parameter wie Formatierung oder Längenbegrenzung unterstützen, was hier nicht der Fall ist.

Fazit

Der `F_UDINT_AS_STRING` Funktionsblock ist ein einfacher und effizienter Baustein für die Konvertierung von vorzeichenlosen 32-Bit-Integern in Strings. Seine klare Schnittstelle und einfache Funktionsweise machen ihn ideal für grundlegende Konvertierungsaufgaben in Automatisierungsprojekten.

F_UDINT_AS_WSTRING

• − ◦ • − ◦ • − ◦ •

Einleitung

Der Funktionsblock `F_UDINT_AS_WSTRING` dient der Konvertierung eines vorzeichenlosen 32-Bit-Integer-Werts (`UDINT`) in eine Unicode-Zeichenkette (`WSTRING`). Diese Funktionalität ist besonders nützlich, wenn numerische Werte für die Anzeige oder Weiterverarbeitung als Text benötigt werden.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der vorzeichenlose 32-Bit-Integer-Wert (`UDINT`), der in eine Unicode-Zeichenkette konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Die resultierende Unicode-Zeichenkette (`WSTRING`) nach der Konvertierung.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` ausgelöst wird. Der Algorithmus wandelt den Wert von `IN` in eine `WSTRING`-Darstellung um und speichert das Ergebnis in `OUT`. Anschließend wird das Ereignis `CNF` ausgelöst, um den Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne zusätzliche Parameter.
- Der Funktionsblock ist für die Verwendung in IEC 61131-konformen Umgebungen optimiert.

Zustandsübersicht

1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Processing**: Führt die Konvertierung durch.
3. **Completed**: Signalisiert den Abschluss mit `CNF` und kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- Anzeige von numerischen Werten in Benutzeroberflächen.
- Protokollierung von numerischen Daten in Textdateien.
- Vorbereitung von Daten für die Kommunikation über Text-basierte Protokolle.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu `F_DINT_AS_WSTRING` verarbeitet dieser Baustein vorzeichenlose Werte.
- Ähnlich wie `F_UDINT_AS_STRING`, aber mit Unicode-Unterstützung.

Fazit

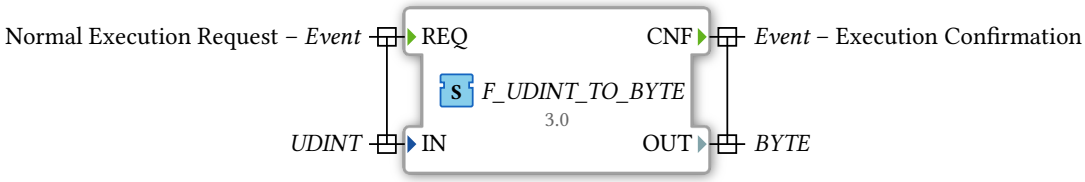
Der `F_UDINT_AS_WSTRING`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, vorzeichenlose 32-Bit-Integer-Werte in Unicode-Zeichenketten umzuwandeln. Seine klare Schnittstelle und direkte Funktionsweise machen ihn zu einem wertvollen Werkzeug in vielen Automatisierungsanwendungen.

F_UDINT_TO_BYTE

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UDINT_TO_BYTE` dient der Konvertierung eines 32-Bit-unsigned-Integer-Werts (UDINT) in einen 8-Bit-unsigned-Integer-Wert (BYTE). Diese Konvertierung ist nützlich, wenn Daten zwischen Systemen mit unterschiedlichen Wortbreiten ausgetauscht werden müssen oder wenn Speicherplatz optimiert werden soll.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der zu konvertierende 32-Bit-unsigned-Integer-Wert (UDINT).

Daten-Ausgänge

- **OUT**: Der resultierende 8-Bit-unsigned-Integer-Wert (BYTE) nach der Konvertierung.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, wenn das Ereignis `REQ` empfangen wird. Dabei wird der Wert am Eingang `IN` vom Typ UDINT in einen BYTE-Wert umgewandelt und am Ausgang `OUT` ausgegeben. Das Ereignis `CNF` signalisiert den erfolgreichen Abschluss der Konvertierung.

Technische Besonderheiten

- Die Konvertierung erfolgt durch die Funktion `UDINT_TO_BYTE`, die den UDINT-Wert auf einen BYTE-Wert abbildet. Dabei kann es zu einem Datenverlust kommen, wenn der ursprüngliche Wert außerhalb des darstellbaren Bereichs eines BYTE (0 bis 255) liegt.
- Der Funktionsblock ist als einfacher Funktionsblock (SimpleFB) implementiert und verwendet einen ST-Algorithmus für die Konvertierung.

Zustandsübersicht

1. **Idle-Zustand**: Der Funktionsblock wartet auf das Ereignis `REQ`.
2. **Konvertierungszustand**: Bei Empfang von `REQ` wird die Konvertierung durchgeführt und `CNF` ausgelöst.

Anwendungsszenarien

- **Datenreduktion**: Wenn nur die unteren 8 Bits eines 32-Bit-Werts relevant sind.
- **Schnittstellenanpassung**: Bei der Kommunikation mit Systemen, die nur BYTE-Daten verarbeiten können.
- **Speicheroptimierung**: In Umgebungen mit begrenztem Speicherplatz.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_DINT_TO_BYTE**: Konvertiert einen 32-Bit-signed-Integer (DINT) in einen BYTE-Wert.
- **F_INT_TO_BYTE**: Konvertiert einen 16-Bit-signed-Integer (INT) in einen BYTE-Wert.
- **F_UDINT_TO_WORD**: Konvertiert einen 32-Bit-unsigned-Integer (UDINT) in einen 16-Bit-unsigned-Integer (WORD).

Fazit

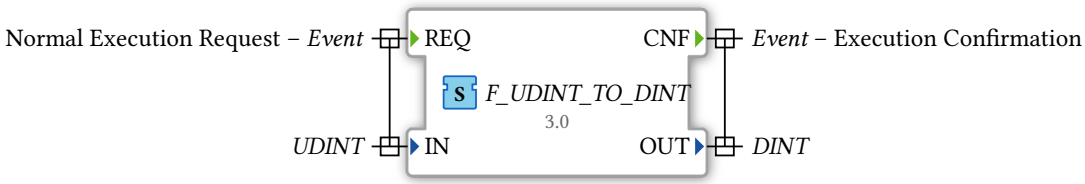
Der Funktionsblock `F_UDINT_TO_BYTE` bietet eine einfache und effiziente Möglichkeit, 32-Bit-unsigned-Integer-Werte in 8-Bit-unsigned-Integer-Werte umzuwandeln. Er ist besonders nützlich in Szenarien, bei denen Speicherplatz oder Kommunikationsbandbreite optimiert werden müssen. Beachten Sie jedoch die möglichen Datenverluste bei Werten außerhalb des BYTE-Bereichs.

F_UDINT_TO_DINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UDINT_TO_DINT` dient der Konvertierung eines vorzeichenlosen 32-Bit-Integer-Werts (UDINT) in einen vorzeichenbehafteten 32-Bit-Integer-Wert (DINT). Diese Konvertierung ist in Szenarien nützlich, in denen Daten zwischen Systemen ausgetauscht werden, die unterschiedliche Datentypen verwenden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (UDINT): Der vorzeichenlose 32-Bit-Integer-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (DINT): Der resultierende vorzeichenbehaftete 32-Bit-Integer-Wert nach der Konvertierung.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem er den Wert des Eingangs `IN` vom Typ `UDINT` in den Typ `DINT` umwandelt. Die Konvertierung erfolgt synchron bei Auslösung des Ereignisses `REQ`. Nach Abschluss der Konvertierung wird das Ereignis `CNF` ausgelöst, und der konvertierte Wert steht am Ausgang `OUT` zur Verfügung.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne zusätzliche Verarbeitung.
- Der Funktionsblock ist für den Einsatz in Echtzeitsystemen geeignet.
- Die Implementierung erfolgt in Structured Text (ST).

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf das Ereignis `REQ`.
2. **Konvertierungszustand**: Führt die Konvertierung durch, wenn `REQ` ausgelöst wird.
3. **Bestätigungszustand**: Löst `CNF` aus, sobald die Konvertierung abgeschlossen ist.

Anwendungsszenarien

- Datenkonvertierung in Steuerungssystemen.
- Schnittstellen zwischen Systemen, die unterschiedliche Datentypen verwenden.
- Echtzeit-Datenverarbeitung, bei der Typkonvertierungen erforderlich sind.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsbausteinen ist `F_UDINT_TO_DINT` spezialisiert auf die Konvertierung zwischen `UDINT` und `DINT`.
- Andere Bausteine könnten zusätzliche Funktionen wie Bereichsprüfungen oder Fehlerbehandlung bieten, was hier nicht der Fall ist.

Fazit

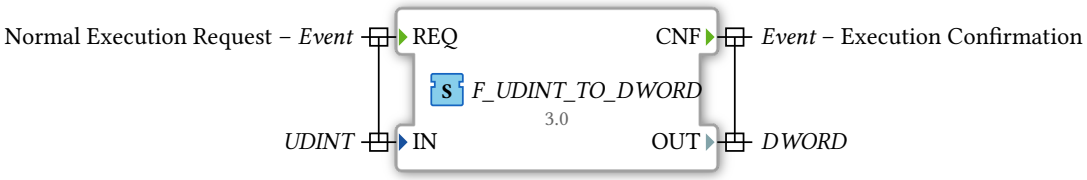
Der Funktionsblock `F_UDINT_TO_DINT` ist ein einfacher und effizienter Baustein für die Konvertierung zwischen vorzeichenlosen und vorzeichenbehafteten 32-Bit-Integern. Er ist besonders in Szenarien nützlich, in denen eine schnelle und direkte Typumwandlung erforderlich ist.

F_UDINT_TO_DWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UDINT_TO_DWORD` dient der Konvertierung eines `UDINT`-Wertes (32-Bit unsignierte Ganzzahl) in einen `DWORD`-Wert (32-Bit Datenwort). Diese Konvertierung ist insbesondere in Szenarien nützlich, in denen numerische Werte als Bitmuster weiterverarbeitet oder übertragen werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (`UDINT`): Der zu konvertierende 32-Bit unsignierte Ganzzahlwert.

Daten-Ausgänge

- **OUT** (`DWORD`): Das Ergebnis der Konvertierung als 32-Bit Datenwort.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Bei Auslösung des `REQ`-Ereignisses wird der Wert am Eingang `IN` vom Typ `UDINT` in einen `DWORD`-Wert konvertiert und am Ausgang `OUT` ausgegeben. Das `CNF`-Ereignis signalisiert den erfolgreichen Abschluss der Konvertierung.

Der zugrundeliegende Algorithmus ist in Structured Text (ST) implementiert:

```
ALGORITHM REQ
OUT := UDINT_TO_DWORD(IN);
END_ALGORITHM
```

Technische Besonderheiten

- Die Konvertierung erfolgt direkt durch eine Typumwandlung, ohne Änderung der Bitrepräsentation.
- Der Funktionsblock ist deterministisch und hat keine internen Zustände.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Umwandlung von numerischen Werten für die bitweise Verarbeitung.
- Kompatibilität mit Systemen, die `DWORD`-Daten erwarten.
- Datenvorbereitung für Kommunikationsprotokolle oder Hardware-Register.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu arithmetischen Funktionsblöcken ändert `F_UDINT_TO_DWORD` nicht den Wert, sondern nur den Datentyp.
- Ähnliche Konvertierungsblöcke existieren für andere Datentypen (z.B. `INT_TO_DINT`), jedoch mit unterschiedlichen Quell- und Zieltypen.

🛠️ Zugehörige Übungen

- Uebung_034b
- Uebung_152
- Uebung_153

Fazit

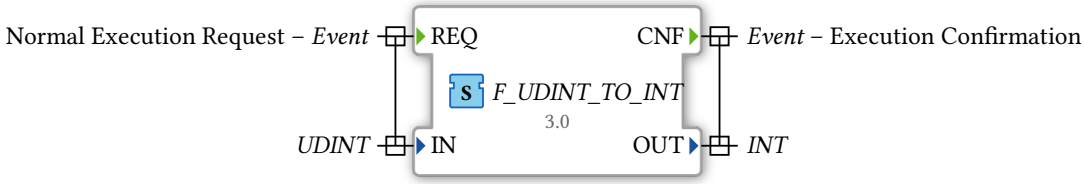
`F_UDINT_TO_DWORD` ist ein einfacher, aber essentieller Funktionsblock für Typkonvertierungen in IEC 61499-basierten Steuerungssystemen. Seine deterministische Funktionsweise und klare Schnittstelle machen ihn zu einem verlässlichen Werkzeug für Datenumwandlungen.

F_UDINT_TO_INT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UDINT_TO_INT` dient der Konvertierung eines `UDINT`-Wertes (32-Bit vorzeichenlose Ganzzahl) in einen `INT`-Wert (16-Bit vorzeichenbehaftete Ganzzahl). Diese Konvertierung ist nützlich, wenn Daten zwischen Systemen oder Komponenten ausgetauscht werden müssen, die unterschiedliche Datentypen verwenden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Löst die Konvertierung aus.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Wird nach erfolgreicher Konvertierung ausgelöst.

Daten-Eingänge

- **IN** (`UDINT`): Der Eingabewert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (`INT`): Der konvertierte Ausgabewert.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `UDINT` zu `INT` durch, wenn das `REQ`-Ereignis empfangen wird. Der Algorithmus nutzt die eingebaute Funktion `UDINT_TO_INT`, um die Konvertierung durchzuführen. Nach erfolgreicher Konvertierung wird das `CNF`-Ereignis ausgelöst, und der konvertierte Wert steht am `OUT`-Ausgang zur Verfügung.

Technische Besonderheiten

- Die Konvertierung kann zu Datenverlust führen, wenn der `UDINT`-Wert außerhalb des darstellbaren Bereichs von `INT` liegt (d.h., größer als 32767 oder kleiner als –32768).
- Der Funktionsblock ist einfach und hat keine internen Zustände oder komplexe Logik.

Zustandsübersicht

Der Funktionsblock hat keinen internen Zustand. Er reagiert ausschließlich auf das `REQ`-Ereignis und führt die Konvertierung unmittelbar durch.

Anwendungsszenarien

- Datenkonvertierung in Steuerungssystemen, wo unterschiedliche Datentypen verwendet werden.
- Integration von Komponenten, die unterschiedliche Datentypen erwarten oder liefern.
- Protokollierung oder Anzeige von Daten, die ursprünglich als `UDINT` gespeichert sind, aber als `INT` dargestellt werden sollen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_DINT_TO_INT**: Konvertiert `DINT` (32-Bit vorzeichenbehaftet) zu `INT`.
- **F_UDINT_TO_DINT: `Konvertiert UDINTzuDINT(32-Bit vorzeichenbehaftet), ohne Datenverlust, da der Wertebereich vonDINT` größer ist.
- **F_INT_TO_UDINT**: Führt die umgekehrte Konvertierung durch (`INT` zu `UDINT`).

Fazit

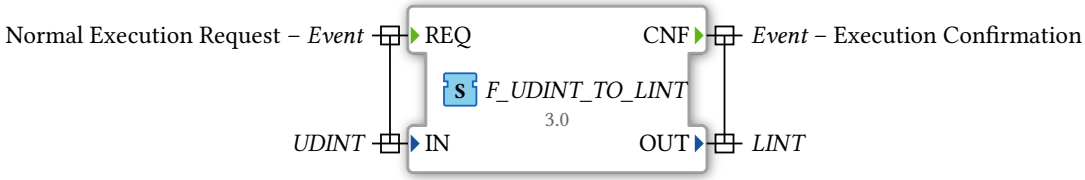
Der `F_UDINT_TO_INT`-Funktionsblock ist ein einfaches und effizientes Werkzeug zur Konvertierung von `UDINT`- zu `INT`-Werten. Er ist besonders nützlich in Szenarien, wo eine solche Typumwandlung erforderlich ist, jedoch sollte auf möglichen Datenverlust bei großen Werten geachtet werden.

F_UDINT_TO_LINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UDINT_TO_LINT` dient der Konvertierung eines 32-Bit vorzeichenlosen Integer-Werts (UDINT) in einen 64-Bit vorzeichenbehafteten Integer-Wert (LINT). Dieser Baustein ist Teil des `iec61131::conversion`-Pakets und ermöglicht eine einfache und effiziente Typumwandlung zwischen diesen beiden Datentypen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung der Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (UDINT): Der 32-Bit vorzeichenlose Integer-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (LINT): Der resultierende 64-Bit vorzeichenbehaftete Integer-Wert nach der Konvertierung.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Auslösung des Ereigniseingangs `REQ` wird der Wert des Daten-Eingangs `IN` (UDINT) in den Daten-Ausgang `OUT` (LINT) kopiert. Die Konvertierung erfolgt direkt, da LINT einen größeren Wertebereich als UDINT abdeckt und somit alle möglichen UDINT-Werte ohne Datenverlust darstellen kann. Nach Abschluss der Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- **Algorithmus**: Der Funktionsblock verwendet einen einfachen ST-Algorithmus, der den Eingabewert direkt dem Ausgabewert zuweist.
- **Datenverlust**: Da LINT einen größeren Wertebereich als UDINT hat, gibt es bei dieser Konvertierung keinen Datenverlust.
- **Performance**: Die Konvertierung ist sehr effizient und erfolgt in konstanter Zeit.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- **Typumwandlung**: In Szenarien, wo ein vorzeichenloser 32-Bit-Wert in einen vorzeichenbehafteten 64-Bit-Wert umgewandelt werden muss, z.B. bei der Verarbeitung von Sensordaten oder in mathematischen Berechnungen.
- **Datenkompatibilität**: Bei der Integration von Systemen, die unterschiedliche Integer-Typen verwenden.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_UDINT_TO_DINT**: Konvertiert UDINT zu DINT (32-Bit vorzeichenbehaftet), kann jedoch zu Überlauf führen, da DINT einen kleineren Wertebereich hat.

- **F_UDINT_TO_ULINT**: Konvertiert UDINT zu ULINT (64-Bit vorzeichenlos), behält jedoch den vorzeichenlosen Charakter bei.

🛠️ Zugehörige Übungen

- Uebung_072c

Fazit

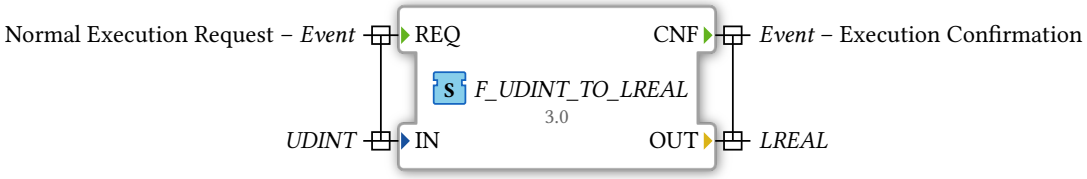
Der `F_UDINT_TO_LINT`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, vorzeichenlose 32-Bit-Integer-Werte in vorzeichenbehaftete 64-Bit-Integer-Werte umzuwandeln. Durch seine direkte und verlustfreie Konvertierung eignet er sich ideal für Anwendungen, die eine Erweiterung des Wertebereichs erfordern.

F_UDINT_TO_LREAL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UDINT_TO_LREAL` dient der Konvertierung eines 32-Bit vorzeichenlosen Integer-Werts (UDINT) in einen 64-Bit Fließkommawert (LREAL). Dieser Baustein ist besonders nützlich in Anwendungen, wo numerische Werte unterschiedlicher Genauigkeit verarbeitet werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Eingang ausgelöst, wird der Wert am Daten-Eingang `IN` verarbeitet.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der konvertierte Wert steht am Daten-Ausgang `OUT` zur Verfügung.

Daten-Eingänge

- **IN** (UDINT): Der vorzeichenlose 32-Bit Integer-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (LREAL): Der resultierende 64-Bit Fließkommawert nach der Konvertierung.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Auslösung des Ereigniseingangs `REQ` wird der Wert am Eingang `IN` vom Typ UDINT gelesen und in einen LREAL-Wert konvertiert. Das Ergebnis wird am Ausgang `OUT` ausgegeben und das Ereignis `CNF` signalisiert die erfolgreiche Konvertierung.

Technische Besonderheiten

- Der Algorithmus ist direkt in ST (Structured Text) implementiert und führt eine einfache Typumwandlung durch.
- Keine zusätzliche Fehlerbehandlung oder Grenzwertprüfung, da UDINT-Werte direkt in LREAL umgewandelt werden können.

Zustandsübersicht

1. **Idle**: Wartet auf das `REQ`-Ereignis.

2. **Processing**: Konvertiert `IN` zu `OUT`.
3. **Completed**: Sendet `CNF` und geht zurück in den Idle-Zustand.

Anwendungsszenarien

- Datenverarbeitung in Steuerungssystemen, wo unterschiedliche numerische Formate verarbeitet werden müssen.
- Integration in größere Systeme, die sowohl Integer- als auch Fließkommaoperationen erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsbausteinen ist `F_UDINT_TO_LREAL` spezialisiert auf die Umwandlung von UDINT zu LREAL und daher effizienter in dieser spezifischen Anwendung.
- Andere Bausteine könnten zusätzliche Funktionalitäten wie Bereichsprüfungen oder Skalierungen bieten, was hier nicht notwendig ist.

Fazit

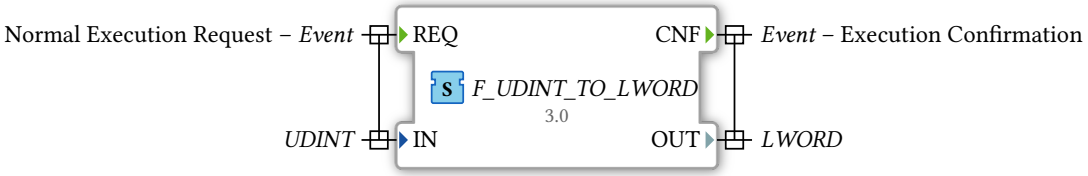
Der `F_UDINT_TO_LREAL` Funktionsblock ist ein einfacher, aber effektiver Baustein für die spezifische Konvertierung von vorzeichenlosen 32-Bit Integern zu 64-Bit Fließkommazahlen. Seine Einfachheit und direkte Implementierung machen ihn zu einer zuverlässigen Wahl für entsprechende Anwendungen.

F_UDINT_TO_LWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UDINT_TO_LWORD` dient der Konvertierung eines 32-Bit unsigned integer Wertes (UDINT) in einen 64-Bit unsigned integer Wert (LWORD). Dieser Baustein ist besonders nützlich in Szenarien, wo eine Erweiterung der Bitbreite von Daten erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung der Konvertierung. Dieser Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN**: Der Eingang für den zu konvertierenden UDINT-Wert.

Daten-Ausgänge

- **OUT**: Der Ausgang für den konvertierten LWORD-Wert.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von UDINT zu LWORD durch, indem der eingehende 32-Bit Wert in einen 64-Bit Wert erweitert wird. Die Konvertierung erfolgt bei jedem Auftreten des `REQ`-Ereignisses und wird durch das `CNF`-Ereignis bestätigt.

Technische Besonderheiten

- Der Funktionsblock verwendet die eingebaute Funktion `UDINT_TO_LWORD` zur Konvertierung.
- Es handelt sich um einen einfachen Funktionsblock ohne Zustandsmaschine oder komplexe Logik.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge oder interne Zustände.

Anwendungsszenarien

- Erweiterung von Datenbreiten in Steuerungssystemen.
- Kompatibilität mit Systemen, die unterschiedliche Bitbreiten verwenden.
- Datenverarbeitung in industriellen Automatisierungslösungen.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Konvertierungsbausteinen wie `F_DINT_TO_LWORD` oder `F_UINT_TO_LWORD` ist dieser speziell für die Konvertierung von UDINT zu LWORD optimiert. Er bietet eine direkte und effiziente Lösung für diese spezifische Konvertierung.

Fazit

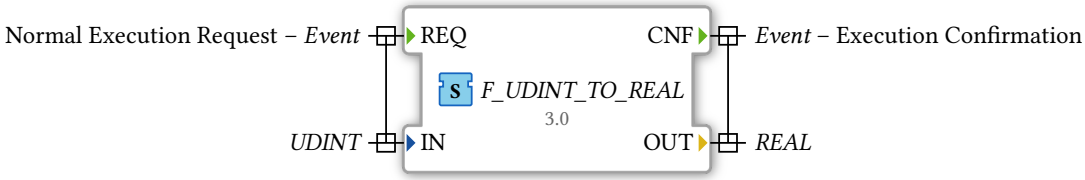
Der `F_UDINT_TO_LWORD` Funktionsblock ist ein einfacher und effizienter Baustein für die Konvertierung von 32-Bit unsigned integer Werten zu 64-Bit unsigned integer Werten. Seine klare Schnittstellenstruktur und einfache Funktionsweise machen ihn zu einer zuverlässigen Komponente in industriellen Automatisierungslösungen.

F_UDINT_TO_REAL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UDINT_TO_REAL` dient der Konvertierung eines vorzeichenlosen 32-Bit-Integer-Werts (`UDINT`) in einen 32-Bit-Gleitkommawert (`REAL`). Dieser Baustein ist besonders nützlich in Szenarien, wo numerische Werte unterschiedlicher Datentypen verarbeitet werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Eingang ausgelöst, wird der Wert am Daten-Eingang `IN` verarbeitet.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der konvertierte Wert steht am Daten-Ausgang `OUT` zur Verfügung.

Daten-Eingänge

- **IN**: Erwartet einen Eingabewert vom Typ `UDINT` (vorzeichenloser 32-Bit-Integer).

Daten-Ausgänge

- **OUT**: Gibt den konvertierten Wert als `REAL` (32-Bit-Gleitkommazahl) aus.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Bei Auslösung des Ereigniseingangs `REQ` wird der Algorithmus gestartet, der den `UDINT`-Wert am Eingang `IN` in einen `REAL`-Wert konvertiert und das Ergebnis am Ausgang `OUT` bereitstellt. Anschließend wird das Ereignis `CNF` ausgelöst, um den erfolgreichen Abschluss der Konvertierung zu signalisieren.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt mittels der eingebauten Funktion `UDINT_TO_REAL`.
- Der Funktionsblock ist einfach und effizient, da er keine komplexen Zustandsmaschinen oder zusätzliche Verarbeitungsschritte benötigt.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Verarbeitung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Umwandlung von Rohdaten (z.B. aus Sensoren oder Kommunikationsprotokollen) in Gleitkommawerte für weitere Berechnungen.
- Integration in Steuerungssysteme, wo unterschiedliche Datentypen verarbeitet werden müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsbausteinen ist `F_UDINT_TO_REAL` spezialisiert und optimiert für die spezifische Konvertierung von `UDINT` zu `REAL`.
- Ähnliche Bausteine könnten zusätzliche Funktionen wie Bereichsprüfungen oder Skalierungen bieten, was hier nicht der Fall ist.

🛠️ Zugehörige Übungen

- Uebung_151
- Uebung_151_AX
- Uebung_152
- Uebung_153

Fazit

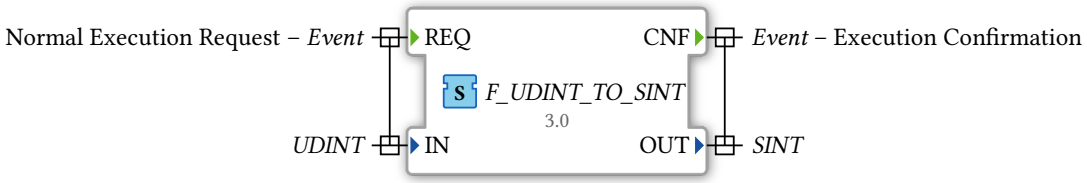
Der `F_UDINT_TO_REAL`-Funktionsblock ist ein einfacher und effizienter Baustein für die Konvertierung von vorzeichenlosen 32-Bit-Integern in Gleitkommazahlen. Seine Einfachheit macht ihn besonders geeignet für Anwendungen, wo eine schnelle und direkte Konvertierung benötigt wird.

F_UDINT_TO_SINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UDINT_TO_SINT` dient der Konvertierung eines 32-Bit vorzeichenlosen Integer-Werts (UDINT) in einen 8-Bit vorzeichenbehafteten Integer-Wert (SINT). Diese Konvertierung ist nützlich, wenn Daten zwischen Systemen oder Komponenten ausgetauscht werden müssen, die unterschiedliche Datentypen verwenden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN** (UDINT): Der vorzeichenlose 32-Bit Integer-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (SINT): Der resultierende vorzeichenbehaftete 8-Bit Integer-Wert nach der Konvertierung.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, wenn das Ereignis `REQ` ausgelöst wird. Der Algorithmus wandelt den Wert von `IN` (UDINT) in einen SINT-Wert um und speichert das Ergebnis in `OUT`. Anschließend wird das Ereignis `CNF` ausgelöst, um den Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Der Funktionsblock ist einfach und deterministisch, da er nur eine direkte Typumwandlung durchführt.
- Es gibt keine zusätzliche Logik oder Fehlerbehandlung für Werte, die außerhalb des darstellbaren Bereichs von SINT liegen.

Zustandsübersicht

Der Funktionsblock hat keinen internen Zustand. Die Konvertierung erfolgt unmittelbar bei Auslösung von `REQ`.

Anwendungsszenarien

- Datenkonvertierung in Steuerungssystemen, wo unterschiedliche Datentypen verwendet werden.
- Schnittstellen zwischen Systemen, die unterschiedliche Integer-Formate erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_DINT_TO_SINT` oder `F_INT_TO_SINT` führen ähnliche Konvertierungen durch, jedoch mit anderen Ausgangs- oder Eingangsdatentypen.
- `F_UDINT_TO_SINT` ist spezifisch für die Konvertierung von UDINT zu SINT.

Fazit

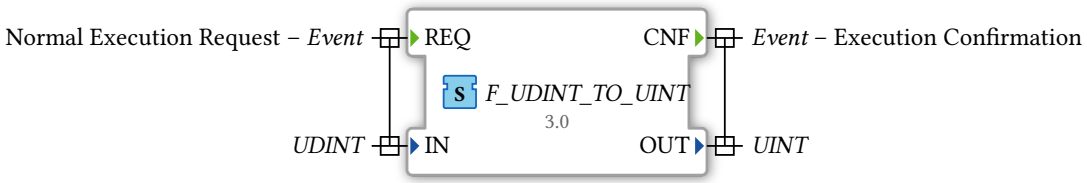
Der `F_UDINT_TO_SINT` Funktionsblock ist ein einfaches und effizientes Werkzeug zur Konvertierung von vorzeichenlosen 32-Bit Integern in vorzeichenbehaftete 8-Bit Integer. Er ist besonders nützlich in Szenarien, wo solche Typumwandlungen regelmäßig benötigt werden.

F_UDINT_TO_UINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UDINT_TO_UINT` dient der Konvertierung eines 32-Bit unsigned integer-Werts (UDINT) in einen 16-Bit unsigned integer-Wert (UINT). Der Block ist Teil des `iec61131::conversion` Pakets.



Schnittstellenstruktur

Ereignis-Eingänge
- **REQ**: Startet die Konvertierung des Eingabewerts. Der Eingabewert `IN` wird mit diesem Event übergeben.

Ereignis-Ausgänge
- **CNF**: Signalisiert die erfolgreiche Konvertierung und gibt den konvertierten Wert `OUT` aus.

Daten-Eingänge
- **IN** (UDINT): Der 32-Bit unsigned integer-Wert, der konvertiert werden soll.

Daten-Ausgänge
- **OUT** (UINT): Der konvertierte 16-Bit unsigned integer-Wert.

Adapter
Der Funktionsblock verfügt über keine Adapter.

Funktionsweise
Bei Empfang des Events `REQ` wird der Eingabewert `IN` vom Typ UDINT in den Typ UINT konvertiert und über den Ausgang `OUT` ausgegeben. Gleichzeitig wird das Event `CNF` ausgelöst, um den erfolgreichen Abschluss der Konvertierung zu signalisieren.

Technische Besonderheiten
- Die Konvertierung erfolgt durch eine direkte Typumwandlung, wobei die höherwertigen Bits des UDINT-Werts abgeschnitten werden, falls der Wert außerhalb des darstellbaren Bereichs von UINT liegt.
- Der Block ist für die Verwendung in Echtzeit-Systemen geeignet.

Zustandsübersicht
Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Empfang des `REQ`-Events.

Anwendungsszenarien
- Konvertierung von 32-Bit-Werten in 16-Bit-Werten in Steuerungsanwendungen.
- Verwendung in Systemen, die mit unterschiedlichen Integer-Größen arbeiten müssen.

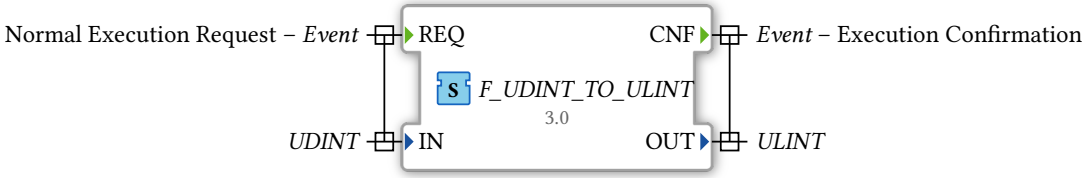
⚖️ Vergleich mit ähnlichen Bausteinen
- Im Vergleich zu generischen Konvertierungsblöcken ist `F_UDINT_TO_UINT` spezialisiert auf die Konvertierung von UDINT zu UINT und daher effizienter in der Ausführung.
- Ähnliche Blöcke wie `F_DINT_TO_INT` konvertieren zwischen anderen Integer-Typen.

🛠️ Zugehörige Übungen
- Uebung_103

Fazit
Der Funktionsblock `F_UDINT_TO_UINT` ist ein spezialisierter und effizienter Baustein zur Konvertierung von 32-Bit- zu 16-Bit-unsigned integer-Werten. Er eignet sich besonders für Anwendungen, bei denen eine schnelle und direkte Typumwandlung erforderlich ist.

F_UDINT_TO_ULINT
• – ◦ • – ◦ • – ◦ •

Einleitung
Der Funktionsblock `F_UDINT_TO_ULINT` dient der Konvertierung eines 32-Bit-unsigned-Integer-Werts (UDINT) in einen 64-Bit-unsigned-Integer-Wert (ULINT). Dieser Baustein ist besonders nützlich, wenn Daten zwischen Systemen oder Komponenten ausgetauscht werden, die unterschiedliche Integer-Größen verwenden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Eingang ausgelöst, liest der Funktionsblock den Wert am Daten-Eingang `IN` und führt die Konvertierung durch.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ausgang wird zusammen mit dem konvertierten Wert am Daten-Ausgang `OUT` aktiviert.

Daten-Eingänge

- **IN**: Der Eingang für den 32-Bit-unsigned-Integer-Wert (UDINT), der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgang für den konvertierten 64-Bit-unsigned-Integer-Wert (ULINT).

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt eine direkte Typumwandlung durch. Bei Auslösung des `REQ`-Ereignisses wird der Wert von `IN` gelesen und ohne weitere Bearbeitung an `OUT` zugewiesen. Da ULINT einen größeren Wertebereich als UDINT abdeckt, ist diese Konvertierung immer sicher und verlustfrei.

Der Algorithmus ist wie folgt implementiert:

```
ALGORITHM REQ
OUT := IN;
END_ALGORITHM
```

Technische Besonderheiten

- **Verlustfreie Konvertierung**: Da ULINT alle Werte von UDINT darstellen kann, gibt es keine Datenverluste.
- **Einfache Implementierung**: Der Funktionsblock benötigt keine zusätzliche Logik oder Fehlerbehandlung.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- **Datenkompatibilität**: Wird verwendet, wenn Systeme mit unterschiedlichen Integer-Größen kommunizieren müssen.
- **Erweiterung des Wertebereichs**: Ermöglicht die Verwendung von UDINT-Werten in Kontexten, die ULINT erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_UDINT_TO_LINT**: Konvertiert UDINT in einen 64-Bit-signed-Integer (LINT). Im Gegensatz dazu behält `F_UDINT_TO_ULINT` den unsigned-Charakter bei.
- **F_DINT_TO_ULINT**: Konvertiert einen 32-Bit-signed-Integer (DINT) in ULINT. `F_UDINT_TO_ULINT` ist spezifisch für unsigned-Werte.

Fazit

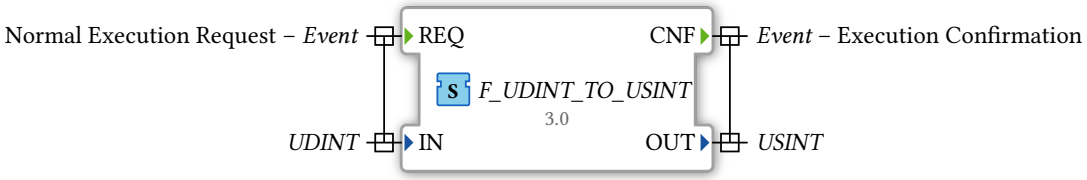
Der `F_UDINT_TO_ULINT`-Funktionsblock ist ein einfaches und effizientes Werkzeug zur sicheren Konvertierung von 32-Bit-unsigned-Integern in 64-Bit-unsigned-Integer. Seine verlustfreie Operation und einfache Handhabung machen ihn ideal für Anwendungen, die eine Erweiterung des Wertebereichs erfordern.

F_UDINT_TO_USINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UDINT_TO_USINT` dient der Konvertierung eines 32-Bit unsigned integer Wertes (UDINT) in einen 8-Bit unsigned integer Wert (USINT). Dieser Baustein ist besonders nützlich in Szenarien, wo eine Typumwandlung zwischen unterschiedlich großen numerischen Werten erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN**: Der Eingang erwartet einen 32-Bit unsigned integer Wert (UDINT), der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgang liefert den konvertierten 8-Bit unsigned integer Wert (USINT).

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` empfangen wird. Der Algorithmus wandelt den Wert von `IN` (UDINT) in den entsprechenden `OUT` (USINT) Wert um. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Die Konvertierung erfolgt durch die eingebaute Funktion `UDINT_TO_USINT`.
- Es wird keine Bereichsprüfung durchgeführt. Bei Werten außerhalb des USINT-Bereichs (0 bis 255) kann es zu unerwarteten Ergebnissen kommen.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung wird bei jedem `REQ` Ereignis neu durchgeführt.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo Speicherplatz optimiert werden muss.
- Kommunikation zwischen Systemen, die unterschiedliche Datentypen verwenden.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsbausteinen ist `F_UDINT_TO_USINT` spezialisiert und daher effizienter in der Ausführung.

- Andere Bausteine könnten zusätzliche Funktionalitäten wie Bereichsprüfungen bieten, was hier nicht der Fall ist.

Fazit

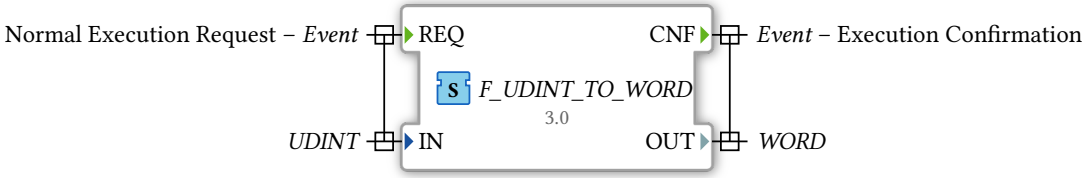
Der `F_UDINT_TO_USINT` Funktionsblock ist ein einfacher und effizienter Baustein für die spezifische Konvertierung von UDINT zu USINT Werten. Er eignet sich besonders für Anwendungen, wo eine schnelle und direkte Typumwandlung benötigt wird.

F_UDINT_TO_WORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UDINT_TO_WORD` dient der Konvertierung eines 32-Bit unsignierten Integer-Werts (UDINT) in einen 16-Bit unsignierten Integer-Wert (WORD). Diese Konvertierung ist insbesondere in Szenarien erforderlich, in denen Daten zwischen Systemen mit unterschiedlichen Wortbreiten ausgetauscht werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (UDINT): Der 32-Bit unsignierte Integer-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (WORD): Der resultierende 16-Bit unsignierte Integer-Wert nach der Konvertierung.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von UDINT zu WORD durch, indem der Algorithmus `REQ` ausgeführt wird. Dabei wird der Eingangswert `IN` vom Typ UDINT in einen WORD-Wert umgewandelt und am Ausgang `OUT` ausgegeben. Die Konvertierung erfolgt synchron mit dem Ereignis `REQ`, und das Ergebnis wird mit dem Ereignis `CNF` bestätigt.

Technische Besonderheiten

- Die Konvertierung schneidet die höherwertigen 16 Bits des UDINT-Werts ab, da ein WORD nur 16 Bits umfasst.
- Der Funktionsblock ist Teil des Pakets `iec61131::conversion`.

Zustandsübersicht

Der Funktionsblock besitzt keinen internen Zustand. Die Konvertierung wird bei jedem Ereignis `REQ` neu ausgeführt.

Anwendungsszenarien

- Datenreduktion von 32-Bit auf 16-Bit in eingebetteten Systemen.

- Kompatibilität mit älteren Systemen, die nur 16-Bit-Daten verarbeiten können.
- Speicheroptimierung in ressourcenbeschränkten Umgebungen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu `F_DINT_TO_WORD` oder `F_INT_TO_WORD` verarbeitet dieser Baustein unsignierte Werte.
- Ähnliche Bausteine wie `F_UDINT_TO_DWORD` führen keine Reduktion der Bitbreite durch.

Fazit

Der `F_UDINT_TO_WORD` Funktionsblock bietet eine einfache und effiziente Möglichkeit, 32-Bit unsignierte Integer-Werte in 16-Bit unsignierte Integer-Werte umzuwandeln. Er ist besonders nützlich in Szenarien, wo eine Reduktion der Datenbreite erforderlich ist, ohne die grundlegende Semantik der Daten zu verändern.

UDINT Konvertierungen

🛠️ Zugehörige Übungen

- Uebung_009
- Uebung_009_AX
- Uebung_009a
- Uebung_011
- Uebung_011a
- Uebung_011a2
- Uebung_012
- NumericValue_ID_TO_NVS
- Uebung_012b
- Uebung_012c
- Uebung_015
- Uebung_015a
- Uebung_020c2_sub
- Uebung_028
- Uebung_034
- Uebung_034a1_Q1
- Uebung_034a1_Q2
- Uebung_034a1_Q4
- Uebung_034b
- Uebung_072c
- Uebung_083
- Uebung_083_AX
- Uebung_103
- Uebung_111
- Uebung_151_AX
- Uebung_152
- Uebung_153

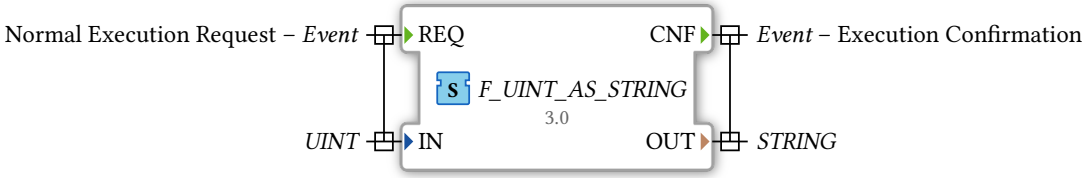
UINT

F_UINT_AS_STRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UINT_AS_STRING` dient der Konvertierung eines vorzeichenlosen Integer-Werts (`UINT`) in eine Zeichenkette (`STRING`). Diese Funktionalität ist besonders nützlich, wenn numerische Werte für die Ausgabe, Protokollierung oder Weiterverarbeitung als Text benötigt werden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Eingang ausgelöst, wird der Wert am Daten-Eingang `IN` verarbeitet.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang wird zusammen mit dem konvertierten Wert am Daten-Ausgang `OUT` aktiviert.

Daten-Eingänge

- **IN** (`UINT`): Der vorzeichenlose Integer-Wert, der in eine Zeichenkette umgewandelt werden soll.

Daten-Ausgänge

- **OUT** (`STRING`): Die Zeichenkette, die aus dem Eingangswert `IN` erzeugt wurde.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Bei Auslösung des Ereigniseingangs `REQ` wird der Algorithmus gestartet, der den Wert von `IN` mit der Funktion `UINT_AS_STRING` in einen `STRING` umwandelt. Das Ergebnis wird an `OUT` ausgegeben und das Ereignis `CNF` signalisiert die erfolgreiche Konvertierung.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne Verzögerung.
- Der Funktionsblock ist einfach und effizient implementiert, ohne zusätzliche Zustände oder komplexe Logik.

Zustandsübersicht

Der Funktionsblock hat keinen internen Zustand. Jede Ausführung ist unabhängig von vorherigen Aufrufen.

Anwendungsszenarien

- Anzeige von numerischen Werten in Benutzeroberflächen.
- Protokollierung von Messwerten in Textdateien.
- Vorbereitung von Daten für die Kommunikation über Text-basierte Protokolle.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu generischen Konvertierungsblöcken ist `F_UINT_AS_STRING` spezialisiert auf die Umwandlung von `UINT` zu `STRING`.
- Andere Blöcke könnten zusätzliche Parameter wie Formatierung oder Basis (dezimal, hexadezimal) unterstützen, was hier nicht der Fall ist.

Fazit

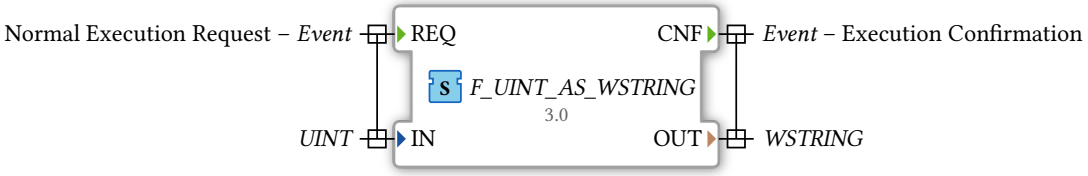
Der `F_UINT_AS_STRING` Funktionsblock ist ein einfaches und effektives Werkzeug für die Konvertierung von vorzeichenlosen Integer-Werten in Zeichenketten. Seine Stärke liegt in der Einfachheit und direkten Anwendbarkeit in Szenarien, wo eine schnelle und unkomplizierte Umwandlung benötigt wird.

F_UINT_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UINT_AS_WSTRING` dient der Konvertierung eines vorzeichenlosen Integer-Werts (`UINT`) in eine Wide-String-Darstellung (`WSTRING`). Dieser Baustein ist besonders nützlich in Szenarien, wo numerische Werte in eine für Menschen lesbare Textform umgewandelt werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Bei Auslösung dieses Ereignisses wird der Wert am Daten-Eingang `IN` verarbeitet.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieses Ereignis wird zusammen mit dem konvertierten Wert am Daten-Ausgang `OUT` ausgegeben.

Daten-Eingänge

- **IN**: Der Eingang für den zu konvertierenden `UINT`-Wert.

Daten-Ausgänge

- **OUT**: Der Ausgang für den konvertierten `WSTRING`-Wert.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Auslösung des `REQ`-Ereignisses wird der Algorithmus ausgeführt, der den `UINT`-Wert am Eingang `IN` in einen `WSTRING` umwandelt. Das Ergebnis wird am Ausgang `OUT` bereitgestellt und das `CNF`-Ereignis signalisiert die erfolgreiche Konvertierung.

Technische Besonderheiten

- Der Funktionsblock ist einfach und effizient implementiert, ohne zusätzliche Zustände oder komplexe Logik.
- Die Konvertierung erfolgt direkt über die eingebaute Funktion `UINT_AS_WSTRING`.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Verarbeitung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Anzeige von numerischen Werten in Benutzeroberflächen.
- Protokollierung und Debugging, wo numerische Werte als Text ausgegeben werden müssen.
- Kommunikation mit Systemen, die String-Daten erwarten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsbausteinen ist `F_UINT_AS_WSTRING` spezialisiert auf die Umwandlung von `UINT` zu `WSTRING` und daher effizienter für diesen spezifischen Anwendungsfall.
- Andere Bausteine könnten zusätzliche Funktionen wie Formatierung oder Locale-Anpassung bieten, was hier nicht der Fall ist.

Fazit

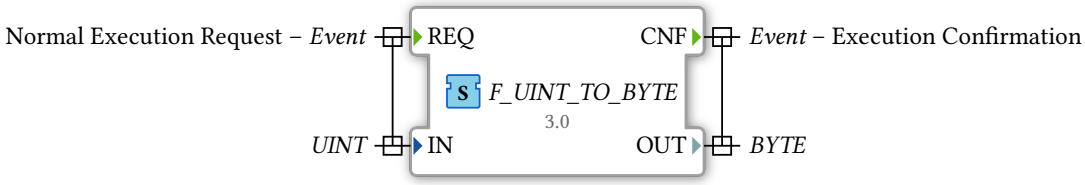
Der `F_UINT_AS_WSTRING` Funktionsblock ist ein einfacher und effektiver Baustein für die Konvertierung von vorzeichenlosen Integer-Werten in Wide-Strings. Seine Einfachheit und direkte Funktionsweise machen ihn ideal für Anwendungen, die eine schnelle und unkomplizierte Umwandlung benötigen.

F_UINT_TO_BYTE

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UINT_TO_BYTE` dient der Konvertierung eines vorzeichenlosen 16-Bit-Integer-Werts (UINT) in einen 8-Bit-Byte-Wert (BYTE). Dieser Baustein ist besonders nützlich in Szenarien, wo eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingang für den UINT-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgang für den konvertierten BYTE-Wert.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald ein Ereignis am Eingang `REQ` eintrifft. Dabei wird der Wert am Eingang `IN` vom Typ UINT in einen BYTE-Wert umgewandelt und am Ausgang `OUT` ausgegeben. Die Konvertierung erfolgt mittels der Funktion `UINT_TO_BYTE`. Nach Abschluss der Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Die Konvertierung erfolgt ohne zusätzliche Parameter oder Konfiguration.
- Der Funktionsblock ist deterministisch und hat keine internen Zustände.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung wird bei jedem `REQ`-Ereignis neu ausgeführt.

Anwendungsszenarien

- Typumwandlung in Kommunikationsprotokollen, wo BYTE-Daten erwartet werden.
- Datenreduktion, wenn nur die unteren 8 Bit eines UINT-Werts relevant sind.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsbausteinen ist `F_UINT_TO_BYTE` spezialisiert auf die Umwandlung von UINT zu BYTE.

- Andere Bausteine könnten zusätzliche Funktionen wie Bereichsprüfungen oder Skalierungen bieten, was hier nicht der Fall ist.

🛠️ Zugehörige Übungen

- Uebung_12x_sub

Fazit

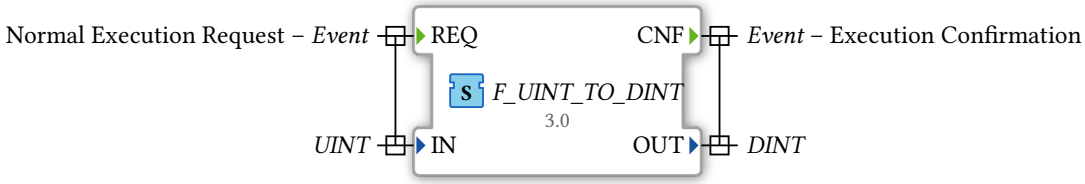
Der `F_UINT_TO_BYTE` Funktionsblock ist ein einfacher und effizienter Baustein für die spezifische Konvertierung von UINT zu BYTE. Er eignet sich besonders für Anwendungen, wo eine schnelle und direkte Typumwandlung benötigt wird.

F_UINT_TO_DINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UINT_TO_DINT` dient der Konvertierung eines vorzeichenlosen 16-Bit-Integer-Werts (UINT) in einen vorzeichenbehafteten 32-Bit-Integer-Wert (DINT). Diese Konvertierung ist nützlich, wenn Daten zwischen Systemen ausgetauscht werden, die unterschiedliche Datentypen verwenden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `UINT`, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgangswert vom Typ `DINT`, der das Ergebnis der Konvertierung enthält.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Empfang eines Ereignisses am Eingang `REQ` wird der Wert von `IN` gelesen und direkt in den Datentyp `DINT` konvertiert. Das Ergebnis wird an `OUT` ausgegeben und das Ereignis `CNF` wird ausgelöst, um den Abschluss der Konvertierung zu signalisieren.

Technische Besonderheiten

- Der Funktionsblock führt eine direkte Typumwandlung durch, ohne zusätzliche Berechnungen oder Skalierungen.
- Die Konvertierung ist verlustfrei, da der `DINT`-Typ einen größeren Wertebereich als `UINT` abdeckt.

Zustandsübersicht

1. **Idle**: Wartet auf ein Ereignis am Eingang `REQ`.
2. **Converting**: Führt die Konvertierung durch und gibt das Ergebnis aus.
3. **Completed**: Signalisiert den Abschluss der Konvertierung mit dem Ereignis `CNF`.

Anwendungsszenarien

- Datenkonvertierung in Steuerungssystemen, die unterschiedliche Datentypen verwenden.
- Integration von Sensordaten, die als `UINT` vorliegen, in Systeme, die `DINT` erwarten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken ist `F_UINT_TO_DINT` spezialisiert und effizient für die Konvertierung von `UINT` zu `DINT`.
- Andere Blöcke könnten zusätzliche Funktionen wie Skalierung oder Bereichsprüfung bieten, was hier nicht erforderlich ist.

Fazit

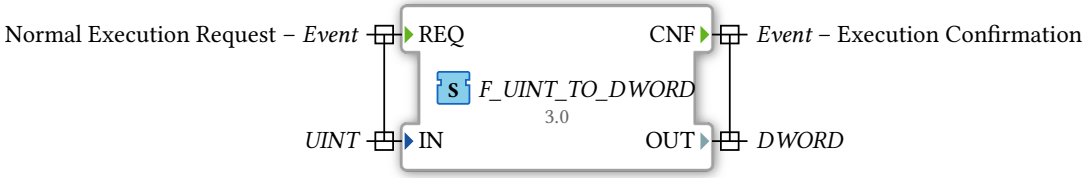
Der `F_UINT_TO_DINT`-Funktionsblock ist ein einfacher und effizienter Baustein für die direkte Konvertierung von `UINT` zu `DINT`. Er ist besonders nützlich in Szenarien, wo eine verlustfreie Typumwandlung benötigt wird.

F_UINT_TO_DWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UINT_TO_DWORD` dient der Konvertierung eines UINT-Wertes (16-Bit unsigned integer) in einen DWORD-Wert (32-Bit unsigned integer). Diese Konvertierung ist insbesondere in Szenarien erforderlich, in denen Daten unterschiedlicher Bitbreiten verarbeitet oder übertragen werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: UINT-Wert (16-Bit), der in einen DWORD-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT**: DWORD-Wert (32-Bit), der das Ergebnis der Konvertierung enthält.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald ein Ereignis am Eingang `REQ` eintrifft. Der Algorithmus `REQ` wandelt den UINT-Wert am Eingang `IN` in einen DWORD-Wert um und speichert das Ergebnis in `OUT`. Anschließend wird das Ereignis `CNF` ausgelöst, um den Abschluss der Konvertierung zu signalisieren.

Technische Besonderheiten

- Die Konvertierung erfolgt ohne Datenverlust, da der Wertebereich von UINT (0 bis 65535) vollständig in DWORD (0 bis 4294967295) abgebildet werden kann.
- Der Funktionsblock ist Teil des Pakets `iec61131::conversion`.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung wird bei jedem `REQ`-Ereignis neu ausgeführt.

Anwendungsszenarien

- Datenkonvertierung in Steuerungssystemen, wenn unterschiedliche Bitbreiten harmonisiert werden müssen.
- Integration in größere Funktionsblocknetzwerke, die sowohl UINT- als auch DWORD-Daten verarbeiten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen (z.B. `F_INT_TO_DINT`) ist `F_UINT_TO_DWORD` spezialisiert auf die Konvertierung von 16-Bit zu 32-Bit unsigned integers.
- Ähnliche Bausteine könnten zusätzliche Funktionen wie Skalierung oder Grenzwertprüfung bieten, was hier nicht der Fall ist.

Fazit

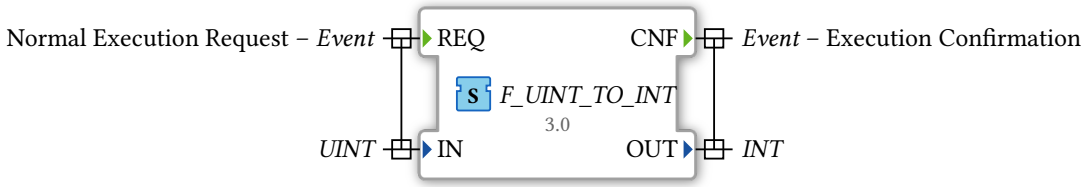
Der Funktionsblock `F_UINT_TO_DWORD` bietet eine einfache und effiziente Möglichkeit, UINT-Werte in DWORD-Werte umzuwandeln. Durch seine klare Schnittstelle und deterministische Funktionsweise eignet er sich ideal für den Einsatz in Steuerungs- und Automatisierungslösungen.

F_UINT_TO_INT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UINT_TO_INT` dient der Konvertierung eines vorzeichenlosen Integer-Werts (UINT) in einen vorzeichenbehafteten Integer-Wert (INT). Diese Funktionalität ist besonders nützlich, wenn Daten zwischen Systemen ausgetauscht werden, die unterschiedliche Integer-Formate verwenden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Eingang ausgelöst, liest der Funktionsblock den Wert am Daten-Eingang `IN` und führt die Konvertierung durch.

Ereignis-Ausgänge

- **CNF**: Signalisiert den erfolgreichen Abschluss der Konvertierung. Gleichzeitig wird der konvertierte Wert am Daten-Ausgang `OUT` ausgegeben.

Daten-Eingänge

- **IN**: Hier wird der vorzeichenlose Integer-Wert (UINT) angelegt, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Gibt den konvertierten vorzeichenbehafteten Integer-Wert (INT) aus.

Adapter

Dieser Funktionsblock verfügt über keine Adapter-Schnittstellen.

Funktionsweise

Bei Auslösung des Ereigniseingangs `REQ` wird der Algorithmus des Funktionsblocks aktiviert. Dieser liest den Wert am Eingang `IN`, konvertiert ihn von UINT nach INT und legt das Ergebnis am Ausgang `OUT`

ab. Anschließend wird das Ereignis `CNF` ausgelöst, um den erfolgreichen Abschluss der Konvertierung zu signalisieren.

Die Konvertierung erfolgt mittels der eingebauten Funktion `UINT_TO_INT()`.

Technische Besonderheiten

- Der Funktionsblock ist einfach aufgebaut und enthält nur einen Algorithmus.
- Es werden keine zusätzlichen Bibliotheken oder Adapter benötigt.
- Die Konvertierung erfolgt direkt und ohne Verzögerung.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung wird bei jedem Auslösen von `REQ` neu durchgeführt.

Anwendungsszenarien

- Datenkonvertierung zwischen Systemen, die unterschiedliche Integer-Formate verwenden.
- Vorverarbeitung von Sensordaten, die als UINT geliefert werden, aber als INT weiterverarbeitet werden sollen.
- Kompatibilitätsherstellung zwischen verschiedenen Protokollen oder Schnittstellen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken ist `F_UINT_TO_INT` spezialisiert und daher effizienter.
- Ähnliche Blöcke wie `F_INT_TO_UINT` führen die umgekehrte Konvertierung durch.
- Universalblöcke wie `F_ANY_TO_ANY` bieten mehr Flexibilität, sind aber unter Umständen weniger performant.

Fazit

Der `F_UINT_TO_INT` Funktionsblock ist ein einfaches, aber effektives Werkzeug zur Konvertierung zwischen vorzeichenlosen und vorzeichenbehafteten Integer-Werten. Seine Einfachheit und direkte Funktionsweise machen ihn zu einer zuverlässigen Lösung für entsprechende Anwendungsfälle.

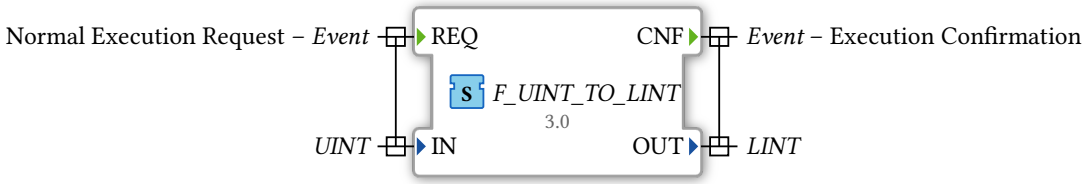
F_UINT_TO_LINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UINT_TO_LINT` dient der Konvertierung eines vorzeichenlosen 16-Bit-Integer-Werts (UINT) in einen vorzeichenbehafteten 64-Bit-Integer-Wert (LINT). Diese Konvertierung ist besonders nützlich, wenn Daten zwischen Systemen mit unterschiedlicher Wortbreite ausgetauscht oder verarbeitet werden müssen.

• – ◦ • – ◦ • – ◦ •



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung der Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `UINT`, der in einen `LINT`-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der konvertierte Ausgangswert vom Typ `LINT`.

Adapter

- Es sind keine Adapter vorhanden.

• – ◦ • – ◦ • – ◦ •

Funktionsweise

Der Funktionsblock führt eine direkte Zuweisung des Eingangswerts `IN` zum Ausgang `OUT` durch. Da `LINT` einen größeren Wertebereich als `UINT` besitzt, ist die Konvertierung verlustfrei. Der Block wird durch das Ereignis `REQ` aktiviert und signalisiert den Abschluss der Operation mit dem Ereignis `CNF`.

• – ◦ • – ◦ • – ◦ •

Technische Besonderheiten

- **Verlustfreie Konvertierung**: Da `LINT` (64 Bit) einen größeren Wertebereich als `UINT` (16 Bit) abdeckt, geht bei der Konvertierung keine Information verloren.
- **Einfache Implementierung**: Der Block verwendet einen einfachen Algorithmus zur direkten Zuweisung der Werte.

• – ◦ • – ◦ • – ◦ •

Zustandsübersicht

1. **Idle-Zustand**: Der Block wartet auf das Ereignis `REQ`.
2. **Aktivierungszustand**: Bei Empfang von `REQ` wird der Eingangswert `IN` gelesen und konvertiert.
3. **Bestätigungszustand**: Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst und der Block kehrt in den Idle-Zustand zurück.

• – ◦ • – ◦ • – ◦ •

Anwendungsszenarien

- **Datenkonvertierung**: In Systemen, die sowohl 16-Bit- als auch 64-Bit-Daten verarbeiten müssen.
- **Schnittstellenanpassung**: Bei der Kommunikation zwischen Geräten mit unterschiedlicher Wortbreite.
- **Datenverarbeitung**: In Algorithmen, die höhere Präzision für Berechnungen benötigen.

• – ◦ • – ◦ • – ◦ •

⚖️ **Vergleich mit ähnlichen Bausteinen**

- **F_UINT_TO_INT**: Konvertiert `UINT` zu `INT` (16 Bit), aber mit Vorzeichenbehaftung.
- **F_UINT_TO_DINT**: Konvertiert `UINT` zu `DINT` (32 Bit), bietet mehr Wertebereich als `INT`, aber weniger als `LINT`.
- **F_UINT_TO_ULINT**: Konvertiert `UINT` zu `ULINT` (64 Bit), bleibt vorzeichenlos.

• – ◦ • – ◦ • – ◦ •

Fazit

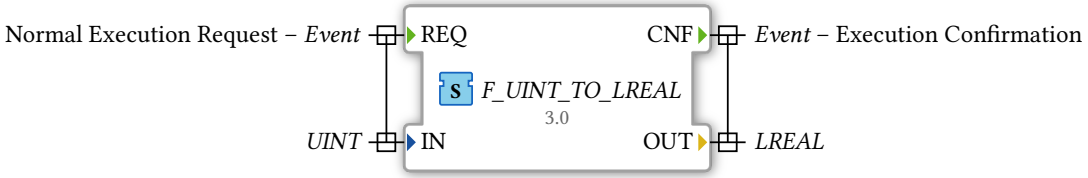
Der `F_UINT_TO_LINT`-Funktionsblock ist ein einfaches und effizientes Werkzeug zur Konvertierung von 16-Bit- vorzeichenlosen Integer-Werten in 64-Bit- vorzeichenbehaftete Integer-Werte. Seine verlustfreie Konvertierung und einfache Handhabung machen ihn ideal für Anwendungen, die eine höhere Präzision oder Kompatibilität mit 64-Bit-Systemen erfordern.

F_UINT_TO_LREAL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UINT_TO_LREAL` dient der Konvertierung eines vorzeichenlosen 16-Bit-Integer-Werts (UINT) in einen 64-Bit-Gleitkommawert (LREAL). Diese Konvertierung ist insbesondere in Steuerungsanwendungen nützlich, wo unterschiedliche Datentypen verarbeitet werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Bei diesem Ereignis wird der Wert am Eingang `IN` gelesen und konvertiert.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der konvertierte Wert steht am Ausgang `OUT` zur Verfügung.

Daten-Eingänge

- **IN** (UINT): Der vorzeichenlose 16-Bit-Integer-Wert, der in einen LREAL-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT** (LREAL): Der resultierende 64-Bit-Gleitkommawert nach der Konvertierung.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem der UINT-Wert von `IN` direkt in einen LREAL-Wert umgewandelt und an `OUT` ausgegeben wird. Die Konvertierung erfolgt bei Auftreten des Ereignisses `REQ`, und das Ereignis `CNF` signalisiert die Fertigstellung.

Technische Besonderheiten

- **Algorithmus**: Der Algorithmus ist in Structured Text (ST) implementiert und besteht aus einer einfachen Zuweisung (`OUT := IN`).
- **Datentypen**: Unterstützt die Konvertierung von UINT (0 bis 65535) zu LREAL (64-Bit-Gleitkomma).

Zustandsübersicht

1. **Idle**: Wartet auf das Ereignis `REQ`.
2. **Konvertierung**: Bei `REQ` wird der Wert von `IN` gelesen und konvertiert.
3. **Abschluss**: Das Ereignis `CNF` wird ausgelöst, und der konvertierte Wert steht an `OUT` zur Verfügung.

Anwendungsszenarien

- **Messwertverarbeitung**: Umwandlung von diskreten Sensorwerten (z.B. ADC-Rohdaten) in Gleitkommawerte für weitere Berechnungen.
- **Datenkommunikation**: Konvertierung von ganzzahligen Werten für die Verwendung in Gleitkommaoperationen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_INT_TO_LREAL**: Konvertiert einen vorzeichenbehafteten Integer (INT) zu LREAL.
- **F_DINT_TO_LREAL**: Konvertiert einen 32-Bit-Integer (DINT) zu LREAL.
- **F_ULINT_TO_LREAL**: Konvertiert einen 64-Bit-unsigned Integer (ULINT) zu LREAL.

Fazit

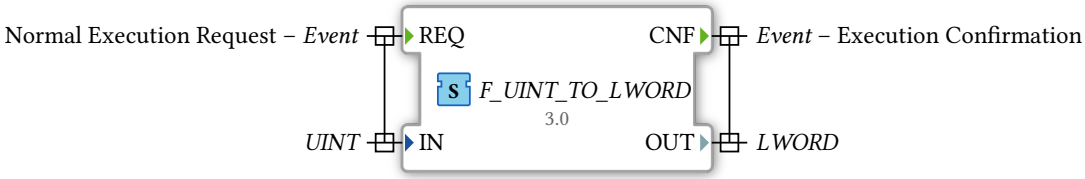
Der `F_UINT_TO_LREAL` Funktionsblock bietet eine einfache und effiziente Möglichkeit, vorzeichenlose 16-Bit-Integer-Werte in 64-Bit-Gleitkommawerte umzuwandeln. Seine klare Schnittstelle und direkte Funktionsweise machen ihn ideal für Anwendungen, die eine schnelle und zuverlässige Typkonvertierung erfordern.

F_UINT_TO_LWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UINT_TO_LWORD` dient der Konvertierung eines `UINT`-Wertes (16-Bit unsigned integer) in einen `LWORD`-Wert (64-Bit unsigned integer). Diese Konvertierung ist insbesondere in Szenarien erforderlich, in denen Daten unterschiedlicher Bitbreiten verarbeitet oder übertragen werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `UINT`, der in einen `LWORD`-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der konvertierte Ausgangswert vom Typ `LWORD`.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung mittels des Algorithmus `REQ` durch, der bei Auslösung des Ereignisses `REQ` aktiviert wird. Der Algorithmus wandelt den `UINT`-Eingangswert `IN` in einen `LWORD`-Wert um und weist diesen dem Ausgang `OUT` zu. Nach Abschluss der Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Die Konvertierung erfolgt ohne Datenverlust, da der `LWORD`-Typ einen größeren Wertebereich als `UINT` abdeckt.
- Der Funktionsblock ist deterministisch und hat keine internen Zustände.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge oder interne Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Datenkonvertierung in Steuerungssystemen, wo unterschiedliche Datentypen verarbeitet werden müssen.
- Integration in größere Funktionsblock-Netzwerke, die Daten unterschiedlicher Bitbreiten verarbeiten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken ist `F_UINT_TO_LWORD` spezialisiert auf die Konvertierung von `UINT` zu `LWORD` und bietet somit eine optimierte und typsichere Lösung.
- Andere Konvertierungsblöcke könnten zusätzliche Funktionen wie Fehlerbehandlung oder Überlaufprüfung bieten, was hier nicht der Fall ist.

Fazit

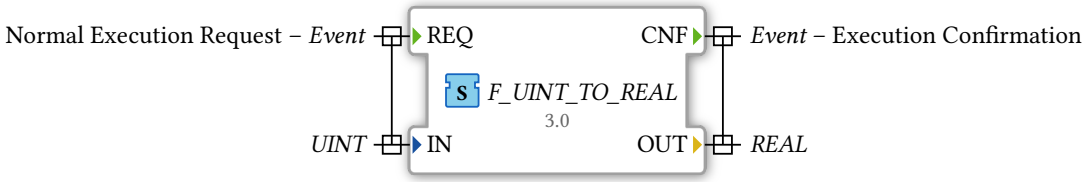
Der `F_UINT_TO_LWORD` Funktionsblock bietet eine einfache und effiziente Möglichkeit, `UINT`-Werte in `LWORD`-Werte umzuwandeln. Seine Einfachheit und Determinismus machen ihn zu einer zuverlässigen Komponente in Steuerungsanwendungen. Für komplexere Anforderungen, wie Fehlerbehandlung, sollten jedoch erweiterte Funktionsblöcke in Betracht gezogen werden.

F_UINT_TO_REAL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UINT_TO_REAL` dient der Konvertierung eines vorzeichenlosen Integer-Werts (`UINT`) in eine Fließkommazahl (`REAL`). Diese Funktionalität ist besonders nützlich, wenn Daten zwischen Systemen ausgetauscht werden, die unterschiedliche Datentypen verwenden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `UINT`, der in einen `REAL`-Wert umgewandelt werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgangswert vom Typ `REAL`, der das Ergebnis der Konvertierung darstellt.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` ausgelöst wird. Der Algorithmus innerhalb des Blocks weist den Wert von `IN` direkt `OUT` zu, wobei eine implizite Typumwandlung von `UINT` zu `REAL` stattfindet. Nach Abschluss der Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Die Konvertierung erfolgt ohne zusätzliche Parameter oder Einstellungen.
- Der Block ist Teil des Pakets `iec61131::conversion`.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung von `REQ`.

Anwendungsszenarien

- Umwandlung von Sensorwerten, die als `UINT` vorliegen, in `REAL` für weitere Berechnungen.
- Kompatibilität zwischen Systemen, die unterschiedliche Datentypen verwenden.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu komplexeren Konvertierungsblöcken bietet `F_UINT_TO_REAL` eine einfache, direkte Umwandlung ohne zusätzliche Funktionalität.
- Ähnliche Blöcke könnten zusätzliche Features wie Skalierung oder Bereichsprüfungen bieten, was hier nicht der Fall ist.

🛠️ Zugehörige Übungen

- Uebung_072c

Fazit

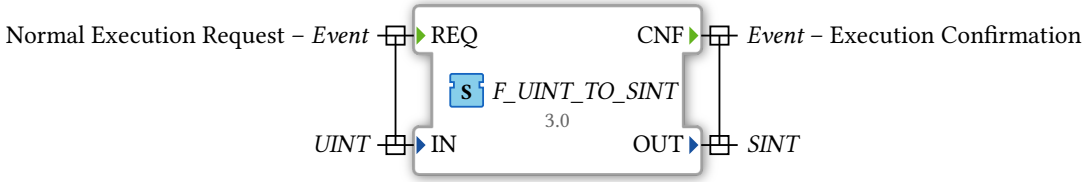
Der `F_UINT_TO_REAL`-Funktionsblock ist ein einfaches und effizientes Werkzeug für die Typumwandlung von `UINT` zu `REAL`. Seine Stärke liegt in der Einfachheit und direkten Anwendbarkeit, ohne Overhead durch zusätzliche Funktionen.

F_UINT_TO_SINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UINT_TO_SINT` dient der Konvertierung eines vorzeichenlosen 16-Bit-Integer-Werts (UINT) in einen vorzeichenbehafteten 8-Bit-Integer-Wert (SINT). Dieser Baustein ist besonders nützlich in Szenarien, wo eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Bei Aktivierung wird der Eingangswert `IN` gelesen und verarbeitet.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgangswert `OUT` steht dann zur Verfügung.

Daten-Eingänge

- **IN** (UINT): Der vorzeichenlose 16-Bit-Integer-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (SINT): Der resultierende vorzeichenbehaftete 8-Bit-Integer-Wert nach der Konvertierung.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem der `UINT`-Wert am Eingang `IN` in einen `SINT`-Wert umgewandelt und am Ausgang `OUT` ausgegeben wird. Die Konvertierung erfolgt bei Aktivierung des `REQ`-Ereignisses, und das `CNF`-Ereignis signalisiert die Fertigstellung.

Technische Besonderheiten

- Die Konvertierung kann zu Datenverlust führen, wenn der `UINT`-Wert außerhalb des darstellbaren Bereichs von `SINT` (0 bis 127) liegt.

- Der Baustein ist als einfacher Funktionsblock (SimpleFB) implementiert und verwendet einen ST-Algorithmus für die Konvertierung.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Aktivierung des `REQ`-Ereignisses.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo `UINT`-Werte in `SINT`-Werte umgewandelt werden müssen.
- Datenverarbeitung in industriellen Automatisierungssystemen, wo unterschiedliche Datentypen verarbeitet werden.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsbausteinen ist `F_UINT_TO_SINT` spezialisiert auf die Umwandlung von `UINT` zu `SINT`.
- Andere Konvertierungsbausteine könnten zusätzliche Funktionen wie Bereichsprüfungen oder Fehlerbehandlung bieten.

Fazit

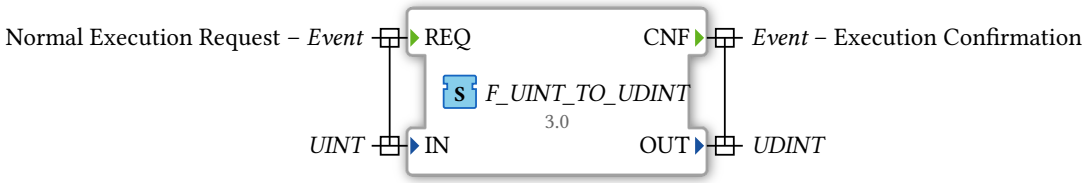
Der `F_UINT_TO_SINT`-Funktionsblock ist ein effizientes Werkzeug für die spezifische Konvertierung von `UINT` zu `SINT`. Seine Einfachheit und direkte Funktionsweise machen ihn ideal für Anwendungen, die eine schnelle und unkomplizierte Typumwandlung erfordern.

F_UINT_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UINT_TO_UDINT` dient der Konvertierung eines 16-Bit vorzeichenlosen Integer-Werts (`UINT`) in einen 32-Bit vorzeichenlosen Integer-Wert (`UDINT`). Der Block ist Teil des `iec61131::conversion`-Pakets und eignet sich für Anwendungen, bei denen eine Erweiterung der Bitbreite von numerischen Werten erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der zu konvertierende 16-Bit vorzeichenlose Integer-Wert (`UINT`).

Daten-Ausgänge

- **OUT**: Der konvertierte 32-Bit vorzeichenlose Integer-Wert (`UDINT`).

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Auslösung des Ereigniseingangs `REQ` wird der Wert des Daten-Eingangs `IN` gelesen und direkt dem Daten-Ausgang `OUT` zugewiesen. Da `UDINT` eine größere Bitbreite als `UINT` hat, erfolgt die Konvertierung ohne Datenverlust. Anschließend wird das Ereignis `CNF` ausgelöst, um den Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Die Konvertierung ist verlustfrei, da `UDINT` den Wertebereich von `UINT` vollständig abdeckt.
- Der Block ist deterministisch und hat keine internen Zustände.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Operation wird bei jedem `REQ`-Ereignis neu ausgeführt.

Anwendungsszenarien

- Erweiterung der Bitbreite von Sensordaten für weitere Verarbeitung.
- Kompatibilitätssicherung zwischen Systemen, die unterschiedliche Integer-Formate verwenden.
- Vorverarbeitung von Daten für Algorithmen, die `UDINT` erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken ist `F_UINT_TO_UDINT` spezialisiert und optimiert für die spezifische Konvertierung von `UINT` zu `UDINT`.
- Andere Blöcke wie `F_INT_TO_DINT` behandeln vorzeichenbehaftete Integer, während dieser Block vorzeichenlose Werte verarbeitet.

🛠️ Zugehörige Übungen

- Uebung_015
- Uebung_015a

Fazit

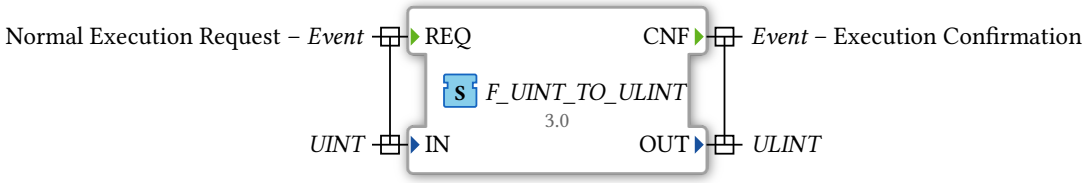
Der `F_UINT_TO_UDINT`-Funktionsblock ist ein effizientes und zuverlässiges Werkzeug für die Konvertierung von 16-Bit zu 32-Bit vorzeichenlosen Integer-Werten. Seine Einfachheit und Determinismus machen ihn ideal für Echtzeitanwendungen, bei denen Leistung und Präzision kritisch sind.

F_UINT_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UINT_TO_ULINT` dient der Konvertierung eines vorzeichenlosen 16-Bit-Integer-Werts (`UINT`) in einen vorzeichenlosen 64-Bit-Integer-Wert (`ULINT`). Diese Konvertierung ist insbesondere dann notwendig, wenn Werte zwischen Systemen mit unterschiedlicher Wortbreite ausgetauscht werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `UINT`, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der konvertierte Ausgangswert vom Typ `ULINT`.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt eine direkte Konvertierung des Eingangswerts `IN` vom Typ `UINT` in den Ausgangswert `OUT` vom Typ `ULINT` durch. Die Konvertierung erfolgt ohne Änderung des numerischen Werts, lediglich der Datentyp wird erweitert.

Technische Besonderheiten

- Die Konvertierung ist verlustfrei, da der `ULINT`-Typ einen größeren Wertebereich als `UINT` abdeckt.
- Der Funktionsblock ist deterministisch und hat keine internen Zustände.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Integration von Systemen mit unterschiedlicher Wortbreite
- Datenverarbeitung in Steuerungen, die sowohl 16-Bit- als auch 64-Bit-Daten verarbeiten müssen
- Erweiterung von Daten für nachfolgende Berechnungen, die höhere Präzision erfordern

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu `F_INT_TO_LINT` werden hier vorzeichenlose Datentypen verarbeitet
- Ähnliche Konvertierungsbausteine existieren für andere Datentypkombinationen (z.B. `F_BYTE_TO_WORD`)

Fazit

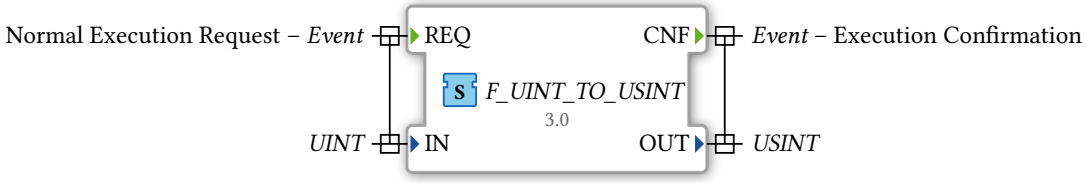
Der `F_UINT_TO_ULINT` Funktionsblock bietet eine einfache und effiziente Möglichkeit, 16-Bit vorzeichenlose Integer in 64-Bit vorzeichenlose Integer umzuwandeln. Seine einfache Struktur und deterministische Funktionsweise machen ihn zu einem zuverlässigen Baustein für Systeme mit gemischten Wortbreiten.

F_UINT_TO_USINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UINT_TO_USINT` dient der Konvertierung eines vorzeichenlosen 16-Bit-Integer-Werts (UINT) in einen vorzeichenlosen 8-Bit-Integer-Wert (USINT). Der Baustein ist Teil der `iec61131::conversion`-Bibliothek und implementiert eine einfache Typumwandlung.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung der Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (UINT): Der vorzeichenlose 16-Bit-Integer-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (USINT): Der resultierende vorzeichenlose 8-Bit-Integer-Wert nach der Konvertierung.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Auslösung des `REQ`-Ereignisses wird der Algorithmus ausgeführt, der den `IN`-Wert von UINT zu USINT konvertiert und das Ergebnis an `OUT` ausgibt. Anschließend wird das `CNF`-Ereignis ausgelöst, um den erfolgreichen Abschluss der Operation zu signalisieren. Die Konvertierung erfolgt durch Abschneiden der höherwertigen Bits, falls der Eingabewert den Wertebereich von USINT überschreitet.

Technische Besonderheiten

- **Wertebereichsbehandlung**: Da USINT nur Werte von 0 bis 255 darstellen kann, werden bei Eingabewerten außerhalb dieses Bereichs die höherwertigen Bits verworfen. Dies kann zu Datenverlust führen.
- **Deterministisches Verhalten**: Die Konvertierung erfolgt in einem einzigen Ausführungsschritt ohne Zustandsverwaltung.

Zustandsübersicht

Der Funktionsblock ist zustandslos und führt die Konvertierung bei jedem `REQ`-Ereignis neu aus.

Anwendungsszenarien

- **Datenreduktion**: Verwendung in Systemen, wo Speicherplatz begrenzt ist und eine Reduktion der Datengröße erforderlich ist.
- **Protokollanpassung**: Anpassung von Datenformaten für Kommunikationsprotokolle, die nur 8-Bit-Daten unterstützen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_UINT_TO_SINT**: Konvertiert UINT in einen vorzeichenbehafteten 8-Bit-Integer (SINT), wobei der Wertebereich anders behandelt wird.
- **F_UINT_TO_INT**: Behält die 16-Bit-Größe bei, konvertiert aber in einen vorzeichenbehafteten Typ.

Fazit

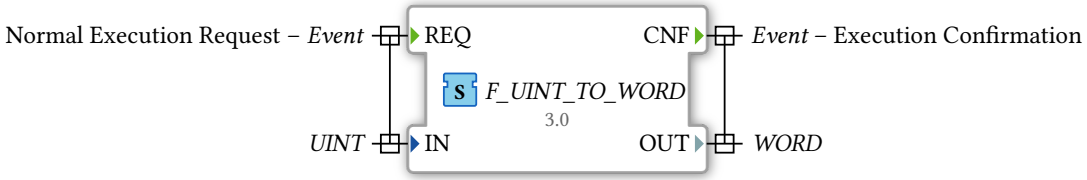
`F_UINT_TO_USINT` ist ein einfacher und effizienter Funktionsblock für die Konvertierung von 16-Bit- zu 8-Bit-Daten ohne Vorzeichen. Seine Verwendung ist ideal in Szenarien, wo eine platzsparende Darstellung erforderlich ist, jedoch sollte die Möglichkeit eines Datenverlusts bei großen Eingabewerten berücksichtigt werden.

F_UINT_TO_WORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_UINT_TO_WORD` dient der Konvertierung eines vorzeichenlosen 16-Bit-Integer-Werts (UINT) in ein 16-Bit-Wort (WORD). Diese Konvertierung ist insbesondere in Szenarien erforderlich, in denen Daten zwischen unterschiedlichen Darstellungsformen oder Systemen ausgetauscht werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der zu konvertierende UINT-Wert.

Daten-Ausgänge

- **OUT**: Der konvertierte WORD-Wert.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem er den UINT-Wert von `IN` mittels der Funktion `UINT_TO_WORD` in ein WORD umwandelt und das Ergebnis an `OUT` ausgibt. Die Konvertierung wird durch das Ereignis `REQ` ausgelöst und durch `CNF` bestätigt.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne zusätzliche Verarbeitung.
- Der Funktionsblock ist deterministisch und hat keine internen Zustände.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Ausführung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Datenkonvertierung in Steuerungssystemen.
- Kommunikation zwischen Systemen, die unterschiedliche Datentypen verwenden.
- Integration von Legacy-Systemen, die WORD-Datentypen erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen wie `F_INT_TO_WORD` oder `F_BYTE_TO_WORD` ist `F_UINT_TO_WORD` speziell auf die Konvertierung von vorzeichenlosen 16-Bit-Integern ausgelegt.
- Der Baustein ist einfacher aufgebaut als generische Konvertierungsbausteine, die mehrere Datentypen unterstützen.

Fazit

Der `F_UINT_TO_WORD`-Funktionsblock bietet eine effiziente und direkte Lösung für die Konvertierung von UINT- zu WORD-Werten. Seine Einfachheit und Determinismus machen ihn zu einer zuverlässigen Wahl in Steuerungs- und Kommunikationsanwendungen.

UINT Konvertierungen

Hier finden Sie Funktionen zur Konvertierung des Datentyps UINT.

🛠️ Zugehörige Übungen

- Uebung_006c
- Uebung_010b4_sub_AX
- Uebung_010b5_sub_AX
- Uebung_010c2
- Uebung_010c2_AX
- AuxIXA_TO_logiBUS_QXA_BG
- NumericValue_ID_TO_NVS
- Uebung_020c2_sub
- IX_TO_logiBUS_LED_strip_QX
- Uebung_035
- Uebung_036
- Uebung_037
- Uebung_038
- Uebung_038_AX
- SINT_TO_Q_NumericValue
- SoftkeyOverride_TO_QX_BG
- Softkey_T_FF_ILOCK_TO_QX_BG
- Uebung_040
- Uebung_040_AX
- Uebung_041
- Uebung_087
- Uebung_087a1
- Uebung_090a1
- Uebung_090a1_AX
- Uebung_090a2
- Uebung_090a2_AX
- Uebung_103

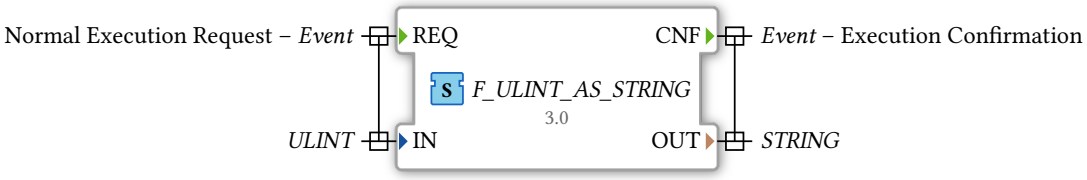
ULINT

F_ULINT_AS_STRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_ULINT_AS_STRING` dient der Konvertierung eines `ULINT`-Wertes (64-Bit unsigned integer) in einen `STRING`. Diese Funktionalität ist besonders nützlich, wenn numerische Werte für die Ausgabe, Protokollierung oder weitere Verarbeitung als Zeichenketten benötigt werden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (`ULINT`): Der zu konvertierende 64-Bit unsigned integer Wert.

Daten-Ausgänge

- **OUT** (`STRING`): Der Ergebnis-String nach der Konvertierung.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, wenn das Ereignis `REQ` empfangen wird. Dabei wird der `ULINT`-Wert am Eingang `IN` in einen `STRING` umgewandelt und am Ausgang `OUT` ausgegeben. Die erfolgreiche Konvertierung wird durch das Ereignis `CNF` signalisiert.

Technische Besonderheiten

- Der Funktionsblock ist einfach aufgebaut und enthält nur einen Algorithmus, der die Konvertierung durchführt.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Parameter oder komplexe Logik.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung wird bei jedem `REQ`-Ereignis neu durchgeführt.

Anwendungsszenarien

- **Protokollierung**: Konvertierung von numerischen Werten für die Ausgabe in Log-Dateien.
- **Benutzeroberflächen**: Darstellung von numerischen Werten als Text in Benutzeroberflächen.
- **Datenkommunikation**: Vorbereitung von numerischen Werten für die Übertragung als Zeichenketten.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_INT_AS_STRING**: Konvertiert einen `INT`-Wert in einen `STRING`. Im Gegensatz dazu unterstützt `F_ULINT_AS_STRING` den größeren `ULINT`-Datentyp.
- **F_DINT_AS_STRING**: Konvertiert einen `DINT`-Wert in einen `STRING`. Ähnlich, aber für 32-Bit signed integers.

Fazit

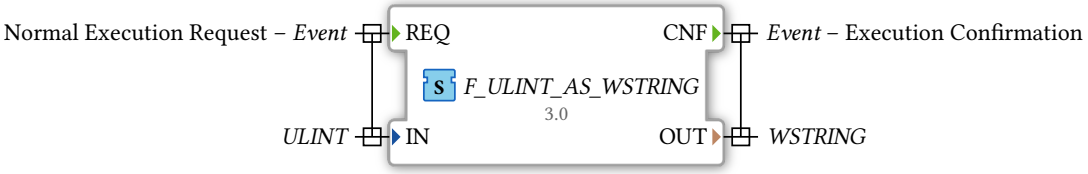
Der Funktionsblock `F_ULINT_AS_STRING` bietet eine einfache und effiziente Möglichkeit, `ULINT`-Werte in `STRING`-Werte umzuwandeln. Durch seine klare Schnittstelle und direkte Funktionsweise ist er ideal für Anwendungen, die eine solche Konvertierung erfordern.

F_ULINT_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_ULINT_AS_WSTRING` dient der Konvertierung eines vorzeichenlosen 64-Bit-Integer-Werts (`ULINT`) in einen Unicode-String (`WSTRING`). Diese Funktionalität ist besonders nützlich in Szenarien, wo numerische Werte für die Anzeige oder Weiterverarbeitung als Text benötigt werden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN**: Der vorzeichenlose 64-Bit-Integer-Wert (`ULINT`), der in einen Unicode-String konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der resultierende Unicode-String (`WSTRING`) nach der Konvertierung.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, wenn das Ereignis `REQ` ausgelöst wird. Dabei wird der Wert am Eingang `IN` mittels der Funktion `ULINT_AS_WSTRING` in einen Unicode-String umgewandelt und am Ausgang `OUT` ausgegeben. Das Ereignis `CNF` signalisiert den erfolgreichen Abschluss der Konvertierung.

Technische Besonderheiten

- Verwendet die Funktion `ULINT_AS_WSTRING` zur Konvertierung.
- Unterstützt den vollständigen Wertebereich von `ULINT` (0 bis 18.446.744.073.709.551.615).

Zustandsübersicht

- **Idle**: Wartet auf das `REQ`-Ereignis.
- **Converting**: Führt die Konvertierung durch und gibt das Ergebnis über `CNF` aus.

Anwendungsszenarien

- Anzeige von großen numerischen Werten in Benutzeroberflächen.
- Protokollierung von numerischen Daten in Textform.
- Vorbereitung von Daten für die Kommunikation mit Systemen, die String-Daten erwarten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu `F_INT_AS_WSTRING` oder `F_DINT_AS_WSTRING` unterstützt dieser Baustein den größeren Wertebereich von `ULINT`.
- Ähnlich zu `F_ULINT_TO_WSTRING`, jedoch mit direkter Konvertierung ohne zusätzliche Formatierungsoptionen.

Fazit

Der `F_ULINT_AS_WSTRING` Funktionsblock bietet eine einfache und effiziente Möglichkeit, große vorzeichenlose Integer-Werte in Unicode-Strings umzuwandeln. Seine klare Schnittstelle und zuverlässige Funktionsweise machen ihn zu einem wertvollen Baustein in der 4diac-IDE.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

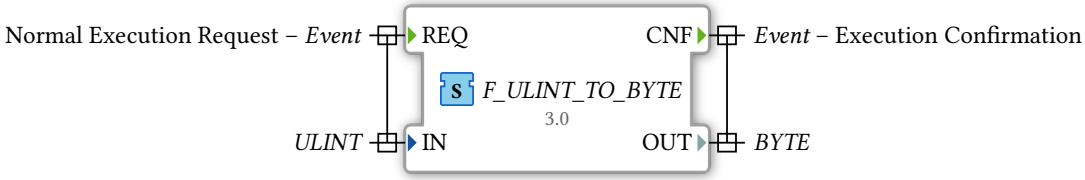
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

F_ULINT_TO_BYTE

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_ULINT_TO_BYTE` dient der Konvertierung eines 64-Bit-unsigned-Integer-Werts (ULINT) in einen 8-Bit-unsigned-Integer-Wert (BYTE). Dieser Baustein ist besonders nützlich in Szenarien, wo eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Wird mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (ULINT): Der 64-Bit-unsigned-Integer-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (BYTE): Der resultierende 8-Bit-unsigned-Integer-Wert nach der Konvertierung.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von ULINT zu BYTE durch, wenn das `REQ`-Ereignis ausgelöst wird. Die Konvertierung erfolgt mittels der Funktion `ULINT_TO_BYTE(IN)`. Nach erfolgreicher Konvertierung wird das `CNF`-Ereignis ausgelöst, und der konvertierte Wert steht am Ausgang `OUT` zur Verfügung.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher Algorithmus ohne Zustandsverwaltung.
- Die Konvertierung kann zu Datenverlust führen, wenn der ULINT-Wert außerhalb des darstellbaren Bereichs eines BYTE (0 bis 255) liegt.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge oder komplexe Zustandslogik.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo ULINT-Werte auf BYTE reduziert werden müssen.
- Datenverarbeitung in eingebetteten Systemen mit begrenzter Speicherkapazität.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_DINT_TO_BYTE` oder `F_UINT_TO_BYTE` führen ähnliche Konvertierungen durch, jedoch mit anderen Ausgangsdatentypen.
- `F_ULINT_TO_BYTE` ist spezifisch für die Konvertierung von 64-Bit-unsigned-Integer zu 8-Bit-unsigned-Integer.

Fazit

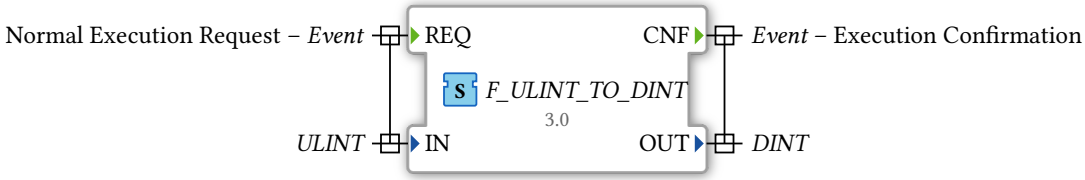
Der `F_ULINT_TO_BYTE` Funktionsblock bietet eine einfache und effiziente Möglichkeit, ULINT-Werte in BYTE-Werte umzuwandeln. Er ist besonders nützlich in Anwendungen, wo eine Reduktion der Datenbreite erforderlich ist, jedoch sollte auf mögliche Datenverluste geachtet werden.

F_ULINT_TO_DINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_ULINT_TO_DINT` dient der Konvertierung eines `ULINT`-Wertes (64-Bit unsignierte Ganzzahl) in einen `DINT`-Wert (32-Bit signierte Ganzzahl). Diese Konvertierung ist besonders in Szenarien relevant, in denen Daten zwischen Systemen mit unterschiedlichen Datentypen ausgetauscht werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN** (`ULINT`): Der unsignierte 64-Bit Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (`DINT`): Das Ergebnis der Konvertierung als signierter 32-Bit Wert.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Bei Aktivierung des `REQ`-Ereignisses wird der Algorithmus ausgeführt, der den `ULINT`-Wert am Eingang `IN` in einen `DINT`-Wert konvertiert und das Ergebnis am Ausgang `OUT` ausgibt. Anschließend wird das `CNF`-Ereignis ausgelöst, um den Abschluss der Konvertierung zu signalisieren.

Technische Besonderheiten

- Die Konvertierung kann zu einem Datenverlust führen, wenn der ursprüngliche `ULINT`-Wert außerhalb des darstellbaren Bereichs von `DINT` liegt (d.h. größer als 2^{31}-1 ist).
- Der Funktionsblock ist Teil des `iec61131::conversion`-Pakets.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Eingang des `REQ`-Ereignisses.

Anwendungsszenarien

- Datenkonvertierung in Steuerungssystemen, die unterschiedliche Datentypen verwenden.
- Integration von Systemen, die unterschiedliche Bitbreiten für Ganzzahlen verwenden.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken ist `F_ULINT_TO_DINT` spezialisiert auf die Konvertierung von `ULINT` zu `DINT`.
- Andere Konvertierungsblöcke könnten andere Datentypen oder Konvertierungsrichtungen unterstützen.

Fazit

Der `F_ULINT_TO_DINT`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, unsignierte 64-Bit Werte in signierte 32-Bit Werte zu konvertieren. Seine Verwendung ist in Szenarien mit unterschiedli-

chen Datentypanforderungen sinnvoll, jedoch sollte auf mögliche Datenverluste bei der Konvertierung geachtet werden.

F_ULINT_TO_DWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_ULINT_TO_DWORD` dient der Konvertierung eines `ULINT`-Wertes (64-Bit unsigned integer) in einen `DWORD`-Wert (32-Bit unsigned integer). Dieser Baustein ist Teil des `iec61131::conversion`-Pakets und ermöglicht eine einfache und effiziente Typumwandlung.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN** (`ULINT`): Der 64-Bit unsigned integer Wert, der in einen `DWORD` umgewandelt werden soll.

Daten-Ausgänge

- **OUT** (`DWORD`): Der resultierende 32-Bit unsigned integer Wert nach der Konvertierung.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das `REQ`-Ereignis empfangen wird. Dabei wird der `ULINT`-Wert am Eingang `IN` in einen `DWORD`-Wert umgewandelt und am Ausgang `OUT` ausgegeben. Die Konvertierung wird durch das `CNF`-Ereignis bestätigt.

Technische Besonderheiten

- Die Konvertierung kann zu einem Datenverlust führen, wenn der `ULINT`-Wert größer als der maximale darstellbare Wert eines `DWORD` ist (2^{32} - 1).
- Der Funktionsblock ist als einfacher FB (`SimpleFB`) implementiert und verwendet einen ST-Algorithmus für die Konvertierung.

Zustandsübersicht

1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Konvertierung**: Führt die Umwandlung durch und gibt das Ergebnis über `OUT` aus.
3. **Bestätigung**: Sendet das `CNF`-Ereignis und kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo 64-Bit Werte in 32-Bit Werte konvertiert werden müssen.
- Datenverarbeitung in Systemen mit unterschiedlichen Wortbreiten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_DINT_TO_DWORD` oder `F_LINT_TO_DWORD` bieten Konvertierungen zwischen anderen Datentypen, jedoch nicht speziell von `ULINT` zu `DWORD`.
- Im Gegensatz zu generischen Konvertierungsblöcken ist dieser spezialisiert und optimiert für die `ULINT`-zu-`DWORD`-Konvertierung.

Fazit

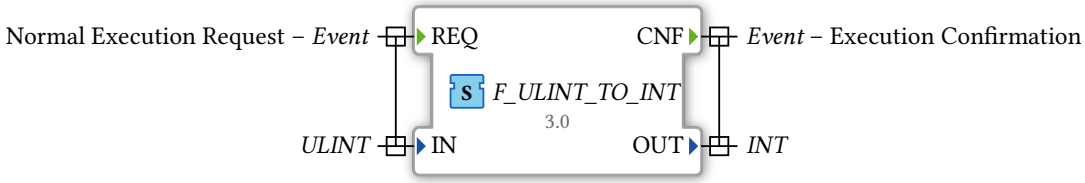
Der `F_ULINT_TO_DWORD`-Funktionsblock bietet eine einfache und effiziente Lösung für die Konvertierung von 64-Bit unsigned integers in 32-Bit unsigned integers. Seine klare Schnittstellenstruktur und einfache Funktionsweise machen ihn ideal für Anwendungen, die solche Typumwandlungen erfordern. Beachten Sie jedoch die möglichen Datenverluste bei großen Eingabewerten.

F_ULINT_TO_INT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_ULINT_TO_INT` dient der Konvertierung eines 64-Bit vorzeichenlosen Integer-Werts (ULINT) in einen 16-Bit vorzeichenbehafteten Integer-Wert (INT). Diese Konvertierung ist insbesondere dann notwendig, wenn Daten zwischen Systemen mit unterschiedlichen Datentypen ausgetauscht werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Dieser Eingang löst die Konvertierung aus. Beim Eintreffen dieses Ereignisses wird der Wert am Daten-Eingang `IN` gelesen und konvertiert.

Ereignis-Ausgänge

- **CNF**: Dieses Ereignis signalisiert den Abschluss der Konvertierung. Es wird zusammen mit dem konvertierten Wert am Daten-Ausgang `OUT` ausgegeben.

Daten-Eingänge

- **IN** (ULINT): Hier wird der 64-Bit vorzeichenlose Integer-Wert bereitgestellt, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (INT): Dieser Ausgang liefert den konvertierten 16-Bit vorzeichenbehafteten Integer-Wert.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung von ULINT zu INT durch, sobald das Ereignis `REQ` eintrifft. Die Konvertierung erfolgt mittels der Funktion `ULINT_TO_INT(IN)`, die den Wert von `IN` in den Zieltyp `INT` umwandelt. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst, und der konvertierte Wert wird an `OUT` ausgegeben.

Technische Besonderheiten

- **Datentyp-Konvertierung**: Die Konvertierung von ULINT (64-Bit) zu INT (16-Bit) kann zu Datenverlust führen, wenn der ursprüngliche Wert außerhalb des darstellbaren Bereichs von INT liegt (d.h., größer als 32767 oder kleiner als –32768 ist).
- **Einfache Implementierung**: Der Funktionsblock ist als SimpleFB implementiert und verwendet einen einfachen ST-Algorithmus für die Konvertierung.

Zustandsübersicht

1. **Idle-Zustand**: Der Funktionsblock wartet auf das Ereignis `REQ`.
2. **Konvertierungszustand**: Bei Eintreffen von `REQ` wird die Konvertierung durchgeführt.
3. **Bestätigungszustand**: Nach der Konvertierung wird `CNF` ausgelöst, und der Funktionsblock kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- **Datenkompatibilität**: Wird verwendet, wenn Systeme mit unterschiedlichen Datentypen kommunizieren müssen.
- **Datenreduktion**: Kann eingesetzt werden, um Speicherplatz zu sparen, wenn die höhere Präzision von ULINT nicht benötigt wird.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_ULINT_TO_DINT**: Konvertiert ULINT zu DINT (32-Bit), bietet einen größeren Wertebereich als INT.
- **F_ULINT_TO_LINT**: Konvertiert ULINT zu LINT (64-Bit), behält die volle Präzision bei.

Fazit

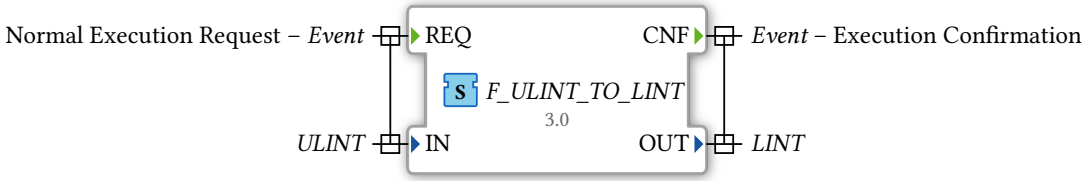
Der Funktionsblock `F_ULINT_TO_INT` ist ein einfaches, aber effektives Werkzeug zur Konvertierung von 64-Bit vorzeichenlosen Integer-Werten in 16-Bit vorzeichenbehaftete Integer-Werte. Er ist besonders nützlich in Szenarien, wo Speichereffizienz oder Kompatibilität mit älteren Systemen erforderlich ist. Beachten Sie jedoch die möglichen Datenverluste bei der Konvertierung großer Werte.

F_ULINT_TO_LINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_ULINT_TO_LINT` dient der Konvertierung eines vorzeichenlosen 64-Bit-Integer-Werts (ULINT) in einen vorzeichenbehafteten 64-Bit-Integer-Wert (LINT). Dieser Baustein ist besonders nützlich in Szenarien, wo eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (ULINT): Der vorzeichenlose 64-Bit-Integer-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (LINT): Der resultierende vorzeichenbehaftete 64-Bit-Integer-Wert nach der Konvertierung.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem er den Wert des Eingangs `IN` vom Typ `ULINT` in den Typ `LINT` umwandelt. Die Umwandlung erfolgt direkt und ohne weitere Bearbeitung. Das Ergebnis wird am Ausgang `OUT` ausgegeben, sobald das Ereignis `CNF` ausgelöst wird.

Technische Besonderheiten

- Die Konvertierung erfolgt mittels der Funktion `ULINT_TO_LINT`, die im Algorithmus des Funktionsblocks implementiert ist.
- Der Baustein ist für die Verwendung in Echtzeitsystemen geeignet, da er keine Verzögerungen oder komplexe Berechnungen beinhaltet.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung wird bei jedem Auftreten des `REQ`-Ereignisses durchgeführt und unmittelbar bestätigt.

Anwendungsszenarien

- Typumwandlung in Steuerungs- und Automatisierungssystemen.
- Datenverarbeitung in industriellen Anwendungen, wo vorzeichenlose und vorzeichenbehaftete Werte verarbeitet werden müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen, wie z.B. `F_DINT_TO_INT`, ist dieser speziell für 64-Bit-Datentypen ausgelegt.
- Ähnliche Bausteine könnten zusätzliche Funktionen wie Bereichsprüfungen oder Fehlerbehandlungen beinhalten, was hier nicht der Fall ist.

Fazit

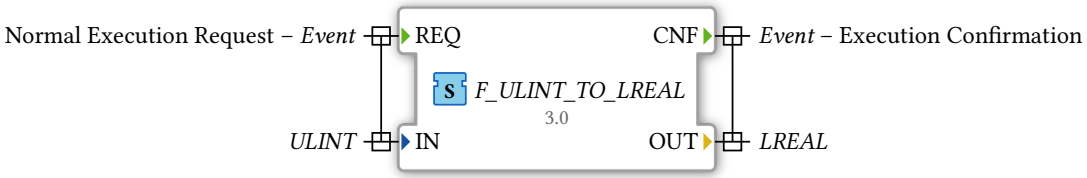
Der `F_ULINT_TO_LINT` Funktionsblock ist ein einfaches und effizientes Werkzeug zur Konvertierung zwischen den Datentypen `ULINT` und `LINT`. Seine Einfachheit macht ihn zuverlässig und leicht in bestehende Systeme integrierbar.

F_ULINT_TO_LREAL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_ULINT_TO_LREAL` dient der Konvertierung eines `ULINT`-Wertes (64-Bit unsigned integer) in einen `LREAL`-Wert (64-Bit floating point). Diese Konvertierung ist insbesondere in Szenarien erforderlich, in denen Ganzzahlwerte in Gleitkommazahlen umgewandelt werden müssen, um weitere Berechnungen durchzuführen.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Startet die Konvertierung. Dieser Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- `CNF`: Signalisiert den Abschluss der Konvertierung. Dieser Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- `IN` (`ULINT`): Der Eingangswert, der konvertiert werden soll.

Daten-Ausgänge

- `OUT` (`LREAL`): Der konvertierte Ausgangswert.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `ULINT` nach `LREAL` durch, wenn das `REQ`-Ereignis ausgelöst wird. Die Konvertierung erfolgt mittels der eingebauten Funktion `ULINT_TO_LREAL`. Nach Abschluss der Konvertierung wird das `CNF`-Ereignis ausgelöst und der konvertierte Wert am Ausgang `OUT` bereitgestellt.

Technische Besonderheiten

- Der Funktionsblock ist einfach aufgebaut und enthält nur einen Algorithmus (`REQ`), der die Konvertierung durchführt.
- Die Konvertierung ist verlustfrei, da `LREAL` einen größeren Wertebereich als `ULINT` abdeckt.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Umwandlung von Ganzzahlwerten in Gleitkommazahlen für weitere Berechnungen.
- Verwendung in Steuerungs- und Regelungssystemen, wo Gleitkommaoperationen erforderlich sind.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen (z.B. `F_INT_TO_REAL`) ist dieser speziell für die Konvertierung von 64-Bit unsigned integers zu 64-Bit floating points optimiert.
- Ähnliche Bausteine könnten eine geringere Genauigkeit oder einen kleineren Wertebereich aufweisen.

Fazit

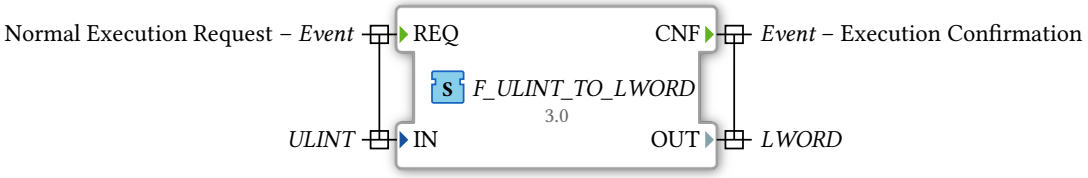
Der `F_ULINT_TO_LREAL`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, `ULINT`-Werte in `LREAL`-Werte umzuwandeln. Seine klare Schnittstelle und verlustfreie Konvertierung machen ihn zu einem nützlichen Baustein in verschiedenen Anwendungen.

F_ULINT_TO_LWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_ULINT_TO_LWORD` dient der Konvertierung eines ULINT-Datentyps (64-Bit unsigned integer) in einen LWORD-Datentyp (64-Bit bitstring). Dieser Baustein ist besonders nützlich in Anwendungen, wo eine direkte Typumwandlung zwischen diesen beiden Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung der Konvertierung. Bei Aktivierung wird der Eingabewert `IN` gelesen und verarbeitet.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgabewert `OUT` ist zu diesem Zeitpunkt verfügbar.

Daten-Eingänge

- **IN**: ULINT (64-Bit unsigned integer) - Der Eingabewert, der in einen LWORD konvertiert werden soll.

Daten-Ausgänge

- **OUT**: LWORD (64-Bit bitstring) - Das Ergebnis der Konvertierung.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem er den ULINT-Eingabewert `IN` in einen LWORD-Ausgabewert `OUT` umwandelt. Die Konvertierung erfolgt direkt und ohne weitere Verarbeitung. Bei Aktivierung des Ereigniseingangs `REQ` wird der Algorithmus ausgeführt und das Ergebnis über den Ereignisausgang `CNF` signalisiert.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne Datenverlust, da beide Datentypen 64 Bit breit sind.
- Der Funktionsblock ist deterministisch und hat keine internen Zustände.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Ausführung erfolgt unmittelbar bei Aktivierung des `REQ`-Ereignisses.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo ULINT-Werte als Bitstrings verarbeitet werden müssen.
- Kommunikation zwischen Systemen, die unterschiedliche Datentypen verwenden.
- Debugging und Protokollierung, wo Bitrepräsentationen von numerischen Werten benötigt werden.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen wie `F_DINT_TO_LWORD` oder `F_UDINT_TO_LWORD` arbeitet dieser Baustein mit dem größeren ULINT-Datentyp.
- Ähnliche Bausteine könnten zusätzliche Funktionen wie Skalierung oder Formatierung bieten, dieser Baustein ist jedoch auf die reine Typumwandlung spezialisiert.

Fazit

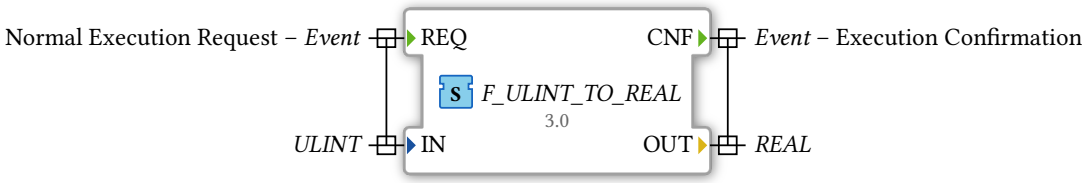
Der `F_ULINT_TO_LWORD` Funktionsblock ist ein einfacher und effizienter Baustein für die direkte Konvertierung von ULINT zu LWORD. Seine deterministische und zustandslose Natur macht ihn zuverlässig und leicht in bestehende Anwendungen integrierbar.

F_ULINT_TO_REAL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_ULINT_TO_REAL` dient der Konvertierung eines `ULINT`-Wertes (64-Bit vorzeichenlose Ganzzahl) in einen `REAL`-Wert (32-Bit Gleitkommazahl). Dieser Baustein ist besonders nützlich in Szenarien, wo eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `ULINT`, der in einen `REAL`-Wert umgewandelt werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgangswert vom Typ `REAL`, der das Ergebnis der Konvertierung darstellt.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` ausgelöst wird. Der Algorithmus innerhalb des Bausteins nutzt die Funktion `ULINT_TO_REAL`, um den `ULINT`-Wert in einen `REAL`-Wert umzuwandeln. Das Ergebnis wird über den Daten-Ausgang `OUT` ausgegeben und das Ereignis `CNF` signalisiert die erfolgreiche Ausführung.

Technische Besonderheiten

- Der Funktionsblock ist Teil des Pakets `iec61131::conversion`.
- Unterstützt die Konvertierung von 64-Bit vorzeichenlosen Ganzzahlen zu 32-Bit Gleitkommazahlen.
- Einfache und direkte Implementierung ohne zusätzliche Parameter oder komplexe Logik.

Zustandsübersicht

Der Funktionsblock hat einen einfachen Zustandsautomaten:

1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Processing**: Führt die Konvertierung durch.
3. **Complete**: Sendet das `CNF`-Ereignis und gibt das Ergebnis aus.

Anwendungsszenarien

- Datenverarbeitung in Steuerungssystemen, wo eine Typumwandlung erforderlich ist.
- Integration in größere Steuerungsalgorithmen, die Gleitkommaberechnungen mit ursprünglich als Ganzzahlen gespeicherten Werten durchführen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_DINT_TO_REAL` oder `F_LINT_TO_REAL` bieten Konvertierungen von anderen Ganzzahltypen zu `REAL`, jedoch mit unterschiedlichen Eingangsgrößen und Präzisionen.
- `F_ULINT_TO_REAL` ist spezifisch für 64-Bit vorzeichenlose Ganzzahlen optimiert.

Fazit

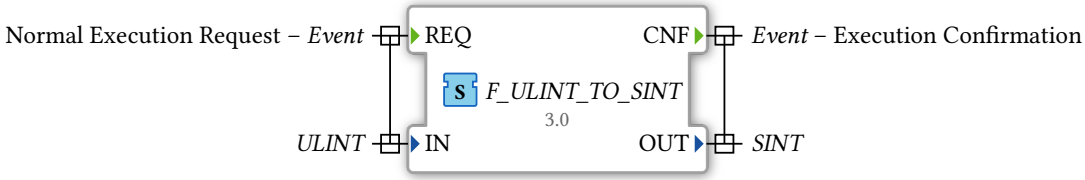
Der `F_ULINT_TO_REAL`-Funktionsblock ist ein einfacher, aber essentieller Baustein für die Typumwandlung in IEC 61499-basierten Steuerungssystemen. Seine klare Schnittstelle und effiziente Funktionsweise machen ihn zu einer zuverlässigen Lösung für Konvertierungsaufgaben.

F_ULINT_TO_SINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_ULINT_TO_SINT` dient der Konvertierung eines `ULINT`-Wertes (64-Bit unsigned integer) in einen `SINT`-Wert (8-Bit signed integer). Dieser Baustein ist besonders nützlich in Anwendungen, bei denen eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN**: Der Eingang erwartet einen `ULINT`-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgang liefert den konvertierten `SINT`-Wert.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` ausgelöst wird. Der Algorithmus wandelt den `ULINT`-Wert an `IN` in einen `SINT`-Wert um und speichert das Ergebnis in `OUT`. Anschließend wird das Ereignis `CNF` ausgelöst, um den Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt durch die Funktion `ULINT_TO_SINT`.
- Es ist zu beachten, dass bei der Konvertierung von einem größeren zu einem kleineren Datentyp Datenverlust auftreten kann, wenn der `ULINT`-Wert außerhalb des darstellbaren Bereichs von `SINT` liegt.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung wird bei jedem `REQ`-Ereignis durchgeführt.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen
- Datenverarbeitung, bei der unterschiedliche Datentypen verarbeitet werden müssen
- Kommunikation zwischen Systemen mit unterschiedlichen Datentypen

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_DINT_TO_SINT` oder `F_LINT_TO_SINT` führen ähnliche Konvertierungen durch, jedoch mit anderen Ausgangsdatentypen.
- `F_ULINT_TO_SINT` ist spezifisch für die Konvertierung von `ULINT` zu `SINT`.

Fazit

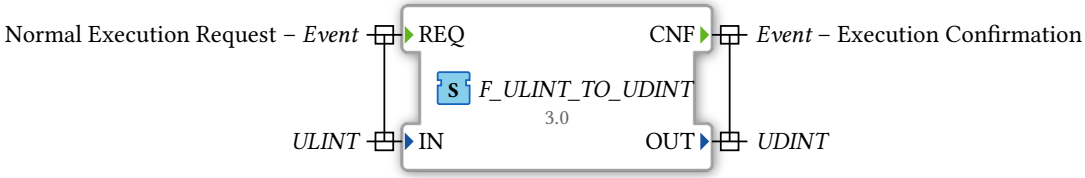
Der `F_ULINT_TO_SINT` Funktionsblock bietet eine einfache und effiziente Möglichkeit, `ULINT`-Werte in `SINT`-Werte umzuwandeln. Er ist besonders nützlich in Anwendungen, bei denen eine Typumwandlung erforderlich ist, jedoch sollte auf mögliche Datenverluste bei der Konvertierung geachtet werden.

F_ULINT_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_ULINT_TO_UDINT` dient der Konvertierung eines 64-Bit unsigned integer Wertes (ULINT) in einen 32-Bit unsigned integer Wert (UDINT). Diese Konvertierung ist nützlich, wenn Daten zwischen Systemen oder Komponenten ausgetauscht werden, die unterschiedliche Datentypen verwenden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Wird mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: ULINT (64-Bit unsigned integer) - Der Eingangswert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: UDINT (32-Bit unsigned integer) - Der konvertierte Ausgangswert.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt eine direkte Konvertierung eines ULINT-Wertes in einen UDINT-Wert durch. Die Konvertierung erfolgt durch den Aufruf der Funktion `ULINT_TO_UDINT` im Algorithmus `REQ`. Bei der Konvertierung wird der 64-Bit Wert auf 32-Bit gekürzt. Es ist zu beachten, dass dabei Datenverlust auftreten kann, wenn der ursprüngliche Wert größer als der maximale Wert eines UDINT ist.

Technische Besonderheiten

- Der Funktionsblock ist einfach und hat keine internen Zustände.
- Die Konvertierung erfolgt in einem einzigen Algorithmus (`REQ`), der bei jedem Aufruf des `REQ`-Ereignisses ausgeführt wird.
- Es gibt keine Fehlerbehandlung für Überlaufbedingungen. Der Benutzer muss sicherstellen, dass der Eingabewert innerhalb des zulässigen Bereichs für UDINT liegt.

Zustandsübersicht

Da der Funktionsblock zustandslos ist, gibt es keine Zustandsübergänge oder interne Zustände.

Anwendungsszenarien

- Datenkonvertierung in Steuerungssystemen, wo unterschiedliche Datentypen verwendet werden.
- Reduzierung der Speicheranforderungen, wenn 64-Bit Werte nicht benötigt werden.
- Kompatibilität mit älteren Systemen, die nur 32-Bit Werte unterstützen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_ULINT_TO_INT**: Konvertiert ULINT in INT (signed integer), wobei hier zusätzlich das Vorzeichen beachtet werden muss.
- **F_ULINT_TO_LINT**: Konvertiert ULINT in LINT (64-Bit signed integer), was keine Datenreduktion bedeutet, aber das Vorzeichen ändert.
- **F_UDINT_TO_ULINT**: Umgekehrte Konvertierung von UDINT zu ULINT, die keine Datenverluste verursacht.

Fazit

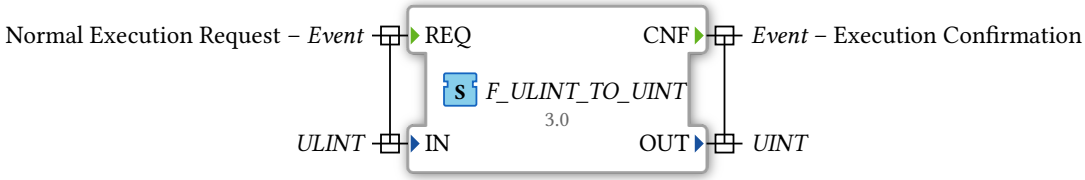
Der `F_ULINT_TO_UDINT` Funktionsblock ist ein einfaches und effizientes Werkzeug zur Konvertierung von 64-Bit unsigned integer Werten in 32-Bit unsigned integer Werte. Er ist besonders nützlich in Szenarien, wo Speichereffizienz oder Kompatibilität mit älteren Systemen erforderlich ist. Benutzer sollten jedoch darauf achten, dass keine Überläufe auftreten, da der Baustein keine automatische Fehlerbehandlung bietet.

F_ULINT_TO_UINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_ULINT_TO_UINT` dient zur Konvertierung eines 64-Bit unsigned integer Wertes (ULINT) in einen 16-Bit unsigned integer Wert (UINT). Der Baustein ist Teil des `iec61131::conversion` Pakets und implementiert eine einfache Typumwandlung.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung der Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN**: ULINT (64-Bit unsigned integer) - Der Eingabewert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: UINT (16-Bit unsigned integer) - Das Ergebnis der Konvertierung.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von ULINT zu UINT durch, wenn das `REQ`-Ereignis eintrifft. Die Konvertierung wird mit der Funktion `ULINT_TO_UINT` durchgeführt. Nach Abschluss der Konvertierung wird das `CNF`-Ereignis ausgelöst.

Technische Besonderheiten

- Der Baustein führt eine direkte Typumwandlung durch, wobei die höherwertigen Bits des ULINT-Wertes abgeschnitten werden, falls der Wert außerhalb des UINT-Bereiches liegt.
- Der Baustein ist deterministisch und hat keine internen Zustände.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo ULINT-Werte auf UINT reduziert werden müssen.
- Datenverarbeitung in eingebetteten Systemen mit begrenzter Speicherkapazität.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsbausteinen ist `F_ULINT_TO_UINT` spezialisiert und optimiert für die spezifische Konvertierung von ULINT zu UINT.
- Andere Bausteine könnten zusätzliche Funktionalitäten wie Bereichsprüfungen oder Skalierungen bieten, was hier nicht der Fall ist.

Fazit

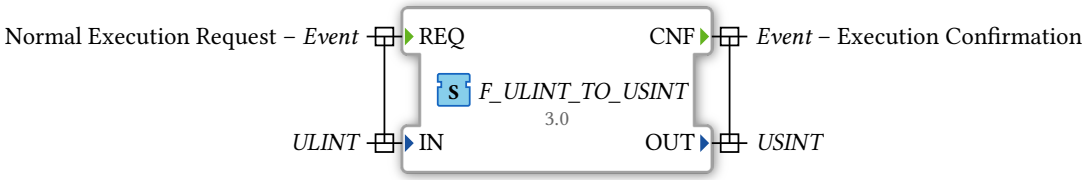
Der `F_ULINT_TO_UINT` Funktionsblock ist ein einfacher und effizienter Baustein zur Konvertierung von 64-Bit unsigned integer Werten in 16-Bit unsigned integer Werte. Er ist besonders nützlich in Anwendungen, wo eine direkte und schnelle Typumwandlung benötigt wird.

F_ULINT_TO_USINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_ULINT_TO_USINT` dient der Konvertierung eines 64-Bit unsigned integer Wertes (`ULINT`) in einen 8-Bit unsigned integer Wert (`USINT`). Dieser Baustein ist besonders nützlich in Szenarien, wo eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung des Funktionsblocks. Verbunden mit dem Daten-Eingang `IN`.

Ereignis-Ausgänge

- **CNF**: Signalisiert die erfolgreiche Beendigung der Konvertierung. Verbunden mit dem Daten-Ausgang `OUT`.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `ULINT` (64-Bit unsigned integer), der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgangswert vom Typ `USINT` (8-Bit unsigned integer), der das Ergebnis der Konvertierung enthält.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei Auslösung des Ereignisses `REQ` wird der Algorithmus ausgeführt, der den Wert von `IN` vom Typ `ULINT` in den Typ `USINT` konvertiert und das Ergebnis an `OUT` ausgibt. Anschließend wird das Ereignis `CNF` ausgelöst, um die erfolgreiche Beendigung der Operation zu signalisieren.

Technische Besonderheiten

- Die Konvertierung schneidet den Wert von `ULINT` auf die Größe von `USINT` zu. Es ist wichtig sicherzustellen, dass der Eingangswert innerhalb des Bereichs von `USINT` (0 bis 255) liegt, um unerwartete Ergebnisse zu vermeiden.
- Der Algorithmus ist in Structured Text (ST) implementiert.

Zustandsübersicht

Der Funktionsblock hat einen einfachen Zustandsautomaten:

1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Processing**: Führt die Konvertierung durch und löst `CNF` aus.

Anwendungsszenarien

- Typumwandlung in Steuerungssystemen, wo Daten in unterschiedlichen Formaten vorliegen.
- Reduzierung der Speichergröße von Daten, wenn nur kleine Wertebereiche benötigt werden.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_DINT_TO_INT` oder `F_LREAL_TO_REAL` führen ähnliche Typumwandlungen durch, jedoch für andere Datentypen.
- Im Gegensatz zu generischen Konvertierungsblöcken ist `F_ULINT_TO_USINT` spezialisiert und optimiert für die Konvertierung zwischen `ULINT` und `USINT`.

Fazit

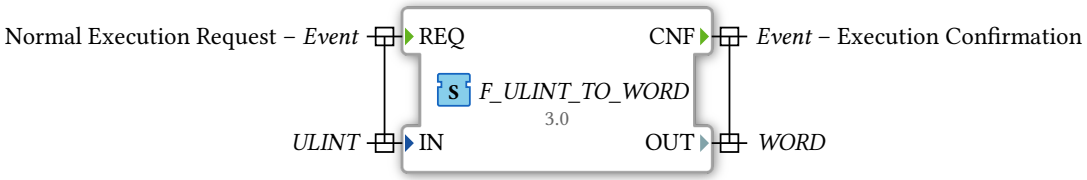
Der `F_ULINT_TO_USINT` Funktionsblock bietet eine effiziente und einfache Möglichkeit, Werte von `ULINT` nach `USINT` zu konvertieren. Seine klare Schnittstelle und einfache Funktionsweise machen ihn zu einem nützlichen Baustein in der IEC 61131-Programmierung.

F_ULINT_TO_WORD

• − ◦ • − ◦ • − ◦ •

Einleitung

Der Funktionsblock `F_ULINT_TO_WORD` dient der Konvertierung eines 64-Bit unsigned integer Wertes (ULINT) in einen 16-Bit unsigned integer Wert (WORD). Der Block ist Teil des `iec61131::conversion` Pakets und implementiert eine einfache Typumwandlung.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung der Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert den erfolgreichen Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN**: ULINT (64-Bit unsigned integer) - Der Eingangswert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: WORD (16-Bit unsigned integer) - Das Ergebnis der Konvertierung.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von ULINT zu WORD durch, wenn das `REQ`-Ereignis eintrifft. Die Konvertierung erfolgt mittels der eingebauten Funktion `ULINT_TO_WORD`. Nach erfolgreicher Konvertierung wird das `CNF`-Ereignis ausgelöst.

Technische Besonderheiten

- Der Block unterstützt keine Fehlerbehandlung bei Überlauf (da ULINT einen größeren Wertebereich als WORD hat).

- Die Konvertierung ist direkt und ohne zusätzliche Verzögerung.

Zustandsübersicht

Der Block hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo ULINT-Werte in WORD-Werte umgewandelt werden müssen.
- Datenreduktion, wenn nur die unteren 16 Bit eines ULINT-Wertes relevant sind.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Blöcke wie `F_DINT_TO_WORD` oder `F_LINT_TO_WORD` bieten Konvertierungen von anderen Integer-Typen zu WORD, jedoch mit unterschiedlichen Wertebereichen und Genauigkeiten.

Fazit

Der `F_ULINT_TO_WORD` Funktionsblock ist ein einfaches und effizientes Werkzeug zur Konvertierung von ULINT zu WORD Werten. Er ist besonders nützlich in Szenarien, wo eine Reduktion der Datenbreite erforderlich ist. Beachten Sie jedoch die möglichen Datenverluste bei der Konvertierung von größeren zu kleineren Datentypen.

ULINT Konvertierungen

Hier finden Sie Funktionen zur Konvertierung des Datentyps ULINT.

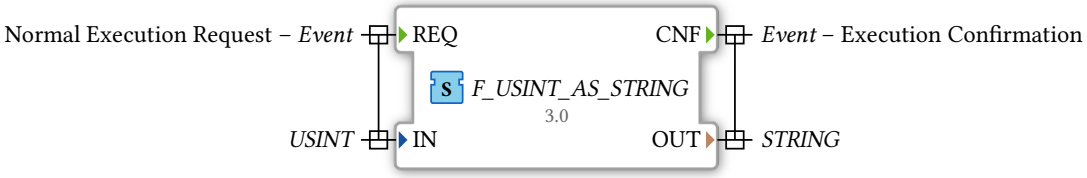
USINT

F_USINT_AS_STRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_USINT_AS_STRING` dient der Konvertierung eines vorzeichenlosen 8-Bit-Integer-Werts (USINT) in einen String (STRING). Diese Funktionalität ist besonders nützlich, wenn numerische Werte für die Anzeige oder Protokollierung in eine lesbare Textform umgewandelt werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Ein vorzeichenloser 8-Bit-Integer-Wert (USINT), der in einen String konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der resultierende String (STRING), der den konvertierten Wert enthält.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Bei Auslösung des Ereigniseingangs `REQ` wird der Algorithmus ausgeführt, der den USINT-Wert `IN` in einen String konvertiert und das Ergebnis an `OUT` ausgibt. Anschließend wird das Ereignis `CNF` ausgelöst, um den erfolgreichen Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Der Funktionsblock verwendet die Funktion `USINT_AS_STRING` zur Konvertierung.
- Es handelt sich um einen einfachen Funktionsblock ohne Zustandsautomat oder komplexe Logik.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge oder -verwaltung. Die Operation wird bei jedem `REQ`-Ereignis ausgeführt und mit `CNF` bestätigt.

Anwendungsszenarien

- Anzeige von numerischen Werten auf einem Bedienpanel.
- Protokollierung von Messwerten in einer Textdatei oder Datenbank.
- Vorbereitung von Daten für die Kommunikation über Textprotokolle (z.B. HTTP, MQTT).

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken ist `F_USINT_AS_STRING` spezialisiert auf die Konvertierung von USINT zu STRING und daher effizienter in dieser spezifischen Anwendung.
- Andere Blöcke könnten zusätzliche Optionen wie Formatierung oder Locale-Einstellungen bieten, was hier nicht der Fall ist.

Fazit

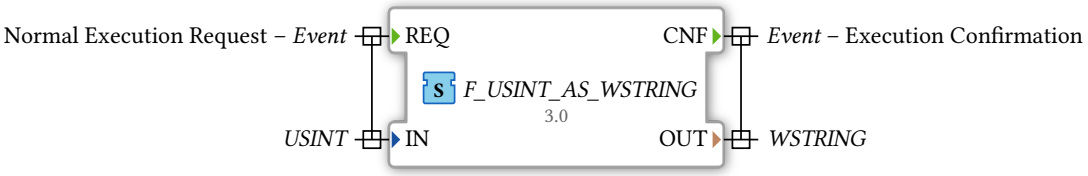
Der `F_USINT_AS_STRING` Funktionsblock ist ein einfaches und effizientes Werkzeug zur Konvertierung von vorzeichenlosen 8-Bit-Integern in Strings. Seine Einfachheit macht ihn besonders geeignet für Anwendungen, bei denen Performance und Zuverlässigkeit im Vordergrund stehen.

F_USINT_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_USINT_AS_WSTRING` dient der Konvertierung eines vorzeichenlosen 8-Bit-Integer-Werts (`USINT`) in eine Unicode-Zeichenkette (`WSTRING`). Diese Funktionalität ist besonders nützlich in Szenarien, wo numerische Werte als Text ausgegeben oder verarbeitet werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Eingang ausgelöst, wird der Wert am Daten-Eingang `IN` verarbeitet.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der konvertierte Wert ist am Daten-Ausgang `OUT` verfügbar.

Daten-Eingänge

- **IN**: Ein vorzeichenloser 8-Bit-Integer-Wert (`USINT`), der in eine Unicode-Zeichenkette konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Die resultierende Unicode-Zeichenkette (`WSTRING`) nach der Konvertierung.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` ausgelöst wird. Dabei wird der Wert am Eingang `IN` genommen und mittels der Funktion `USINT_AS_WSTRING` in eine Unicode-Zeichenkette umgewandelt. Das Ergebnis wird am Ausgang `OUT` bereitgestellt und das Ereignis `CNF` signalisiert die erfolgreiche Ausführung.

Technische Besonderheiten

- Verwendet die eingebaute Funktion `USINT_AS_WSTRING` zur Konvertierung.
- Unterstützt den vollständigen Wertebereich von `USINT` (0 bis 255).

Zustandsübersicht

1. **Idle**: Wartet auf das Ereignis `REQ`.
2. **Konvertierung**: Verarbeitet den Eingangswert und generiert die Ausgabe.
3. **Bestätigung**: Sendet das Ereignis `CNF` und kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- Anzeige von numerischen Werten in Benutzeroberflächen.
- Protokollierung von numerischen Daten in Textdateien.
- Kommunikation mit Systemen, die Textdaten erwarten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken ist `F_USINT_AS_WSTRING` spezialisiert auf die Konvertierung von `USINT` zu `WSTRING`.
- Andere Blöcke könnten andere Datentypen oder zusätzliche Formatierungsoptionen unterstützen.

Fazit

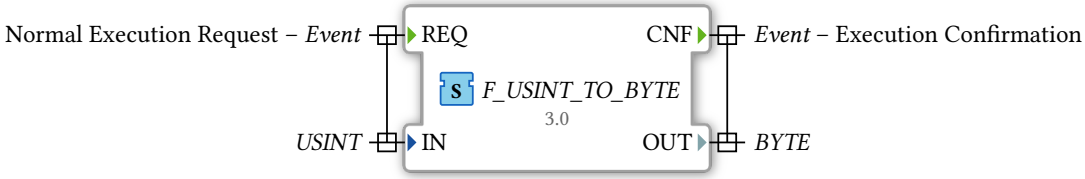
Der `F_USINT_AS_WSTRING` Funktionsblock bietet eine einfache und effiziente Möglichkeit, vorzeichenlose 8-Bit-Integer-Werte in Unicode-Zeichenketten umzuwandeln. Seine klare Schnittstelle und direkte Funktionsweise machen ihn ideal für Projekte, die solche Konvertierungen benötigen.

F_USINT_TO_BYTE

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_USINT_TO_BYTE` dient der Konvertierung eines vorzeichenlosen 8-Bit-Integer-Werts (USINT) in einen Byte-Wertyp (BYTE). Diese Konvertierung ist insbesondere in Szenarien erforderlich, wo Daten zwischen unterschiedlichen Formaten oder Systemen ausgetauscht werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Dieser Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `USINT` (vorzeichenloser 8-Bit-Integer), der in einen `BYTE`-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgangswert vom Typ `BYTE`, der das Ergebnis der Konvertierung darstellt.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` empfangen wird. Dabei wird der Wert am Eingang `IN` vom Typ `USINT` in einen `BYTE`-Wert umgewandelt und am Ausgang `OUT` ausgegeben. Das Ereignis `CNF` signalisiert die erfolgreiche Beendigung der Konvertierung.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne zusätzliche Verarbeitungsschritte.
- Der Block ist für die Verwendung in Echtzeitsystemen geeignet, da er keine zeitkritischen Operationen durchführt.

Zustandsübersicht

1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Konvertierung**: Führt die Konvertierung durch und gibt das Ergebnis über `OUT` aus.
3. **Bestätigung**: Sendet das `CNF`-Ereignis zur Bestätigung der erfolgreichen Konvertierung.

Anwendungsszenarien

- Datenkonvertierung in Steuerungssystemen.
- Kommunikation zwischen Systemen mit unterschiedlichen Datentypen.
- Vorverarbeitung von Sensordaten.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_BYTE_TO_USINT**: Führt die umgekehrte Konvertierung durch (BYTE zu USINT).
- **F_INT_TO_STRING**: Konvertiert einen Integer-Wert in einen String, ist jedoch komplexer in der Handhabung.

🛠️ Zugehörige Übungen

- SIN_TO_PCAN_Callback_Byte

Fazit

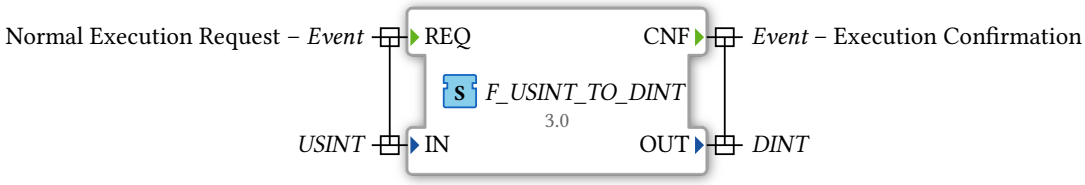
Der `F_USINT_TO_BYTE`-Funktionsblock ist ein einfaches und effizientes Werkzeug zur Konvertierung von `USINT`- zu `BYTE`-Werten. Seine klare Schnittstellenstruktur und direkte Funktionsweise machen ihn ideal für den Einsatz in verschiedenen industriellen Anwendungen.

F_USINT_TO_DINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_USINT_TO_DINT` dient der Konvertierung eines `USINT`-Wertes (8-Bit unsigned integer) in einen `DINT`-Werte (32-Bit signed integer). Diese Konvertierung ist nützlich, wenn Werte zwischen verschiedenen Datentypen umgewandelt werden müssen, insbesondere wenn eine Erweiterung der Bitbreite erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge
- **REQ**: Startet die Konvertierung. Bei Auslösung dieses Ereignisses wird der Wert von `IN` in `OUT` umgewandelt.

Ereignis-Ausgänge
- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieses Ereignis wird zusammen mit dem aktualisierten Wert von `OUT` ausgegeben.

Daten-Eingänge
- **IN** (`USINT`): Der Eingangswert, der konvertiert werden soll.

Daten-Ausgänge
- **OUT** (`DINT`): Der konvertierte Ausgangswert.

Adapter
Keine Adapter vorhanden.

Funktionsweise
Der Funktionsblock führt eine direkte Zuweisung des Eingangswertes `IN` (vom Typ `USINT`) an den Ausgang `OUT` (vom Typ `DINT`) durch. Da `DINT` eine größere Bitbreite hat als `USINT`, wird der Wert ohne Datenverlust erweitert.

Der Algorithmus wird durch das `REQ`-Ereignis ausgelöst und gibt nach der Konvertierung das `CNF`-Ereignis aus.

Technische Besonderheiten
- Der Funktionsblock ist deterministisch und hat keine internen Zustände.
- Die Konvertierung ist verlustfrei, da `DINT` alle Werte von `USINT` darstellen kann.

Zustandsübersicht
Der Funktionsblock hat keine internen Zustände. Die Ausführung erfolgt unmittelbar nach dem Auslösen des `REQ`-Ereignisses.

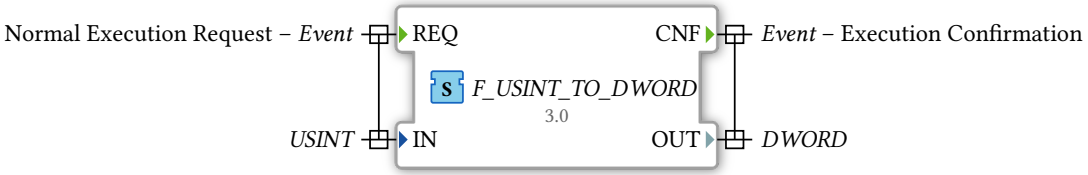
Anwendungsszenarien
- Erweiterung von 8-Bit unsigned Werten auf 32-Bit signed Werte in Steuerungsanwendungen.
- Kompatibilität zwischen verschiedenen Teilen eines Steuerungssystems, die unterschiedliche Datentypen verwenden.

⚖️ Vergleich mit ähnlichen Bausteinen
- Im Vergleich zu `F_SINT_TO_DINT` oder `F_INT_TO_DINT` behandelt dieser Baustein ausschließlich unsigned 8-Bit Werte.
- Ähnliche Konvertierungsbausteine existieren für andere Datentypen, wie z.B. `F_BOOL_TO_INT` oder `F_REAL_TO_LREAL`.

Fazit
Der `F_USINT_TO_DINT`-Funktionsblock ist ein einfacher und effizienter Baustein zur Konvertierung von 8-Bit unsigned Integer-Werten in 32-Bit signed Integer-Werte. Er ist besonders nützlich in Szenarien, wo eine Typenerweiterung ohne Datenverlust erforderlich ist.

F_USINT_TO_DWORD
• – ◦ • – ◦ • – ◦ •

Einleitung
Der Funktionsblock `F_USINT_TO_DWORD` dient der Konvertierung eines `USINT`-Wertes (8-Bit unsigned integer) in einen `DWORD`-Wert (32-Bit unsigned integer). Diese Konvertierung ist besonders nützlich, wenn Daten zwischen Systemen mit unterschiedlichen Wortbreiten ausgetauscht oder verarbeitet werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ` (Normal Execution Request): Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- `CNF` (Execution Confirmation): Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- `IN` (USINT): Der Eingangswert, der konvertiert werden soll. Der Datentyp ist `USINT` (8-Bit unsigned integer).

Daten-Ausgänge

- `OUT` (DWORD): Der konvertierte Ausgangswert. Der Datentyp ist `DWORD` (32-Bit unsigned integer).

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `USINT` nach `DWORD` durch, sobald das Ereignis `REQ` ausgelöst wird. Der Algorithmus `REQ` nutzt die Funktion `USINT_TO_DWORD`, um den Eingabewert `IN` in den Ausgabewert `OUT` umzuwandeln. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Der Funktionsblock ist einfach aufgebaut und enthält nur einen Algorithmus.
- Die Konvertierung erfolgt ohne Datenverlust, da der `DWORD`-Datentyp einen größeren Wertebereich als `USINT` abdeckt.

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf das Ereignis `REQ`.
2. **Konvertierungszustand**: Führt die Konvertierung durch.
3. **Bestätigungszustand**: Löst das Ereignis `CNF` aus und kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- Datenkonvertierung in Steuerungssystemen, die unterschiedliche Wortbreiten verwenden.
- Integration von Peripheriegeräten, die Daten in unterschiedlichen Formaten liefern.
- Datenverarbeitung in industriellen Automatisierungslösungen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_BYTE_TO_DWORD` oder `F_WORD_TO_DWORD` bieten Konvertierungen zwischen anderen Datentypen an.
- `F_USINT_TO_DWORD` ist spezialisiert auf die Konvertierung von 8-Bit unsigned integer zu 32-Bit unsigned integer.

Fazit

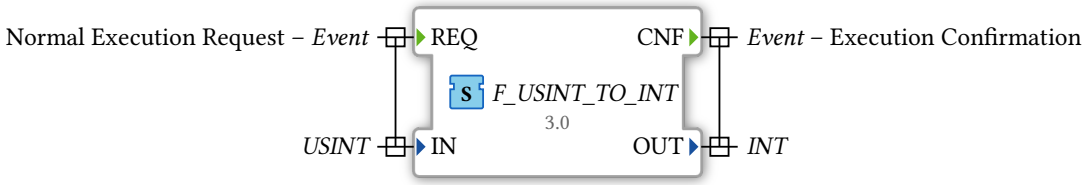
Der Funktionsblock `F_USINT_TO_DWORD` ist ein einfaches und effizientes Werkzeug zur Konvertierung von `USINT`-Werten in `DWORD`-Werte. Seine klare Schnittstellenstruktur und einfache Funktionsweise machen ihn zu einer zuverlässigen Lösung für Datenkonvertierungsaufgaben in industriellen Automatisierungssystemen.

F_USINT_TO_INT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_USINT_TO_INT` dient der Konvertierung eines vorzeichenlosen 8-Bit-Integer-Werts (USINT) in einen vorzeichenbehafteten 16-Bit-Integer-Wert (INT). Dieser Baustein ist Teil des `iec61131::conversion` Pakets und ermöglicht eine einfache Typumwandlung zwischen diesen beiden Datentypen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Dieser Ereigniseingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ereignisausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (USINT): Der vorzeichenlose 8-Bit-Integer-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (INT): Der resultierende vorzeichenbehaftete 16-Bit-Integer-Wert nach der Konvertierung.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem der Wert des Eingangs `IN` direkt dem Ausgang `OUT` zugewiesen wird. Die Konvertierung erfolgt bei Auftreten des Ereignisses `REQ` und wird durch das Ereignis `CNF` bestätigt.

Technische Besonderheiten

- **Einfache Typumwandlung**: Der Baustein führt keine zusätzliche Verarbeitung oder Prüfung der Daten durch.
- **Effiziente Implementierung**: Die Konvertierung erfolgt direkt in einem ST-Algorithmus ohne zusätzliche Logik.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auftreten des `REQ`-Ereignisses.

Anwendungsszenarien

- **Typumwandlung in Steuerungsanwendungen**: Wenn ein vorzeichenloser 8-Bit-Wert in einen vorzeichenbehafteten 16-Bit-Wert umgewandelt werden muss.
- **Datenkompatibilität**: In Szenarien, wo verschiedene Komponenten unterschiedliche Datentypen erwarten.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_USINT_TO_UINT**: Konvertiert USINT zu UINT (vorzeichenloser 16-Bit-Integer).
- **F_USINT_TO_DINT**: Konvertiert USINT zu DINT (vorzeichenbehafteter 32-Bit-Integer).
- **F_USINT_TO_UDINT**: Konvertiert USINT zu UDINT (vorzeichenloser 32-Bit-Integer).

Fazit

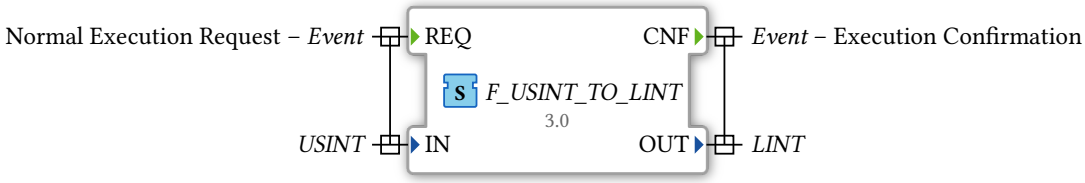
Der `F_USINT_TO_INT` Funktionsblock bietet eine einfache und effiziente Möglichkeit, vorzeichenlose 8-Bit-Integer-Werte in vorzeichenbehaftete 16-Bit-Integer-Werte umzuwandeln. Er ist besonders nützlich in Anwendungen, wo eine Typumwandlung ohne zusätzliche Verarbeitung erforderlich ist.

F_USINT_TO_LINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_USINT_TO_LINT` dient der Konvertierung eines `USINT`-Wertes (8-Bit unsigned integer) in einen `LINT`-Wert (64-Bit signed integer). Diese Konvertierung ist besonders nützlich, wenn Werte aus kleineren Datentypen in größere umgewandelt werden müssen, ohne dabei die ursprüngliche Information zu verlieren.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `USINT`, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der konvertierte Ausgangswert vom Typ `LINT`.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt eine direkte Zuweisung des Eingangswertes `IN` zum Ausgangswert `OUT` durch. Da `USINT` ein 8-Bit unsigned integer und `LINT` ein 64-Bit signed integer ist, wird der Wert ohne Änderung des numerischen Werts in den größeren Datentyp umgewandelt.

Technische Besonderheiten

- **Algorithmus**: Der Algorithmus besteht aus einer einfachen Zuweisung (`OUT := IN;`).
- **Datentyp-Kompatibilität**: Da `USINT` ein unsigned Datentyp ist und `LINT` signed, wird der Wert implizit erweitert, ohne dass es zu Datenverlust kommt.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt sofort bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- **Datenverarbeitung**: Wenn Werte aus kleineren in größere Datentypen umgewandelt werden müssen.
- **Kommunikation**: Bei der Übertragung von Daten zwischen Systemen mit unterschiedlichen Datentyp-Anforderungen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_USINT_TO_INT**: Konvertiert `USINT` zu `INT` (16-Bit signed integer). Im Gegensatz dazu bietet `F_USINT_TO_LINT` eine größere Reichweite durch den 64-Bit Ausgang.
- **F_USINT_TO_ULINT**: Konvertiert `USINT` zu `ULINT` (64-Bit unsigned integer). Hier wird der Ausgang ebenfalls als unsigned behandelt.

Fazit

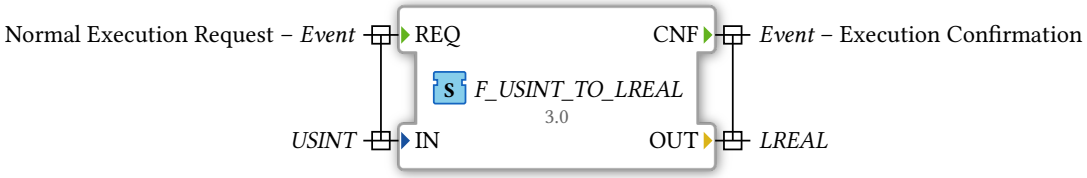
Der `F_USINT_TO_LINT` Funktionsblock ist ein einfacher, aber effektiver Baustein zur Konvertierung von 8-Bit unsigned integer Werten in 64-Bit signed integer Werte. Durch seine direkte und verlustfreie Umwandlung eignet er sich ideal für Anwendungen, bei denen die Erhaltung des numerischen Werts in einem größeren Datentyp erforderlich ist.

F_USINT_TO_LREAL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_USINT_TO_LREAL` dient der Konvertierung eines vorzeichenlosen 8-Bit-Integer-Werts (USINT) in einen 64-Bit-Gleitkommawert (LREAL). Diese Konvertierung ist besonders nützlich, wenn Daten zwischen Systemen mit unterschiedlichen Datentypen ausgetauscht oder verarbeitet werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Startet die Konvertierung. Dieser Ereigniseingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- `CNF`: Signalisiert den Abschluss der Konvertierung. Dieser Ereignisausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- `IN` (USINT): Der Eingangswert, der konvertiert werden soll.

Daten-Ausgänge

- `OUT` (LREAL): Der konvertierte Ausgangswert.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt eine direkte Typumwandlung durch. Bei Eintreffen eines `REQ`-Ereignisses wird der Wert von `IN` in den Datentyp LREAL umgewandelt und an `OUT` ausgegeben. Anschließend wird das `CNF`-Ereignis ausgelöst, um den Abschluss der Konvertierung zu signalisieren.

Technische Besonderheiten

- Die Konvertierung erfolgt ohne zusätzliche Skalierung oder Anpassung.
- Der Block ist für einfache Typumwandlungen optimiert und verfügt über keine komplexe Logik oder Zustandsverwaltung.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung wird bei jedem `REQ`-Ereignis neu ausgeführt.

Anwendungsszenarien

- Konvertierung von Sensordaten, die als USINT vorliegen, für Berechnungen, die LREAL erfordern.
- Schnittstellen zwischen Systemen mit unterschiedlichen Datentypanforderungen.
- Einfache Typumwandlungen in Automatisierungsanwendungen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken ist `F_USINT_TO_LREAL` spezialisiert und daher effizienter für diese spezifische Konvertierung.
- Andere Blöcke könnten zusätzliche Funktionen wie Skalierung oder Bereichsprüfungen bieten, was hier nicht erforderlich ist.

Fazit

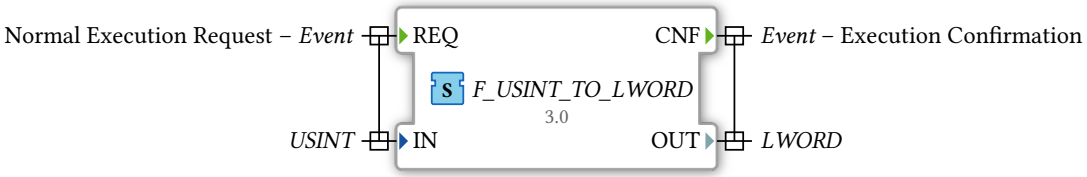
Der `F_USINT_TO_LREAL`-Funktionsblock ist ein einfacher und effizienter Baustein für die direkte Konvertierung von USINT zu LREAL. Seine Stärke liegt in der Einfachheit und Spezialisierung, was ihn ideal für spezifische Anwendungsfälle macht, bei denen keine zusätzliche Logik benötigt wird.

F_USINT_TO_LWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_USINT_TO_LWORD` dient der Konvertierung eines `USINT`-Wertes (8-Bit unsigned integer) in einen `LWORD`-Wert (64-Bit unsigned integer). Diese Konvertierung ist nützlich, wenn Daten unterschiedlicher Bitlängen verarbeitet werden müssen, insbesondere in Systemen, die eine einheitliche Datenbreite erfordern.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Eingang ausgelöst, wird der Wert am Daten-Eingang `IN` gelesen und konvertiert.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der konvertierte Wert steht am Daten-Ausgang `OUT` zur Verfügung.

Daten-Eingänge

- **IN**: Erwartet einen `USINT`-Wert (8-Bit unsigned integer), der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Liefert den konvertierten `LWORD`-Wert (64-Bit unsigned integer).

Adapter

Dieser Funktionsblock verfügt über keine Adapter-Schnittstellen.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` ausgelöst wird. Der Algorithmus liest den Wert von `IN`, konvertiert ihn von `USINT` zu `LWORD` und stellt das Ergebnis an `OUT` bereit. Anschließend wird das `CNF`-Ereignis ausgelöst, um den erfolgreichen Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Die Konvertierung erfolgt ohne Datenverlust, da der `USINT`-Wert in den größeren `LWORD`-Typ umgewandelt wird.
- Der Funktionsblock ist deterministisch und hat keine internen Zustände.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Ausführung erfolgt rein ereignisgesteuert durch das `REQ`-Ereignis.

Anwendungsszenarien

- Datenkonvertierung in Steuerungssystemen, wo unterschiedliche Datenbreiten verarbeitet werden müssen.
- Integration in größere Funktionsblocknetzwerke, die eine einheitliche Datenbreite erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen (z.B. `F_INT_TO_LWORD`) ist dieser speziell für die Konvertierung von 8-Bit unsigned Integern optimiert.
- Ähnliche Bausteine könnten zusätzliche Features wie Skalierung oder Fehlerbehandlung bieten, dieser Block ist jedoch auf die reine Konvertierung beschränkt.

Fazit

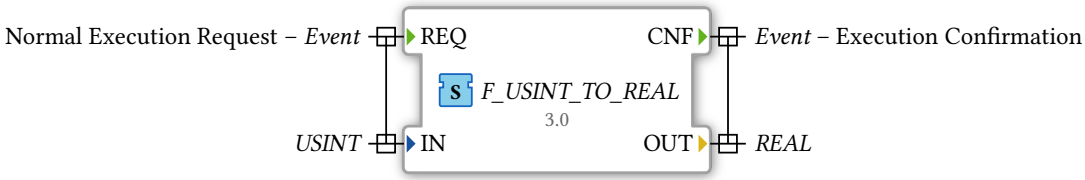
Der `F_USINT_TO_LWORD` Funktionsblock ist ein einfacher und effizienter Baustein zur Konvertierung von 8-Bit unsigned Integern in 64-Bit unsigned Integer. Seine Einfachheit und Determinismus machen ihn zu einer zuverlässigen Komponente in Steuerungssystemen.

F_USINT_TO_REAL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_USINT_TO_REAL` dient der Konvertierung eines vorzeichenlosen 8-Bit-Integer-Werts (USINT) in einen 32-Bit-Gleitkommawert (REAL). Diese Konvertierung ist insbesondere in Szenarien erforderlich, in denen numerische Werte unterschiedlicher Datentypen verarbeitet oder weiterverwendet werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Ein vorzeichenloser 8-Bit-Integer-Wert (USINT), der in einen REAL-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der resultierende 32-Bit-Gleitkommawert (REAL) nach der Konvertierung.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem der USINT-Wert von `IN` direkt in den REAL-Datentyp umgewandelt und an `OUT` ausgegeben wird. Die Konvertierung erfolgt synchron bei Empfang des `REQ`-Ereignisses und wird durch das `CNF`-Ereignis bestätigt.

Technische Besonderheiten

- Die Konvertierung erfolgt ohne zusätzliche Skalierung oder Rundung.
- Der Block ist für einfache Konvertierungen optimiert und bietet keine Fehlerbehandlung für ungültige Werte.

Zustandsübersicht

1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Processing**: Führt die Konvertierung durch.
3. **Completed**: Sendet das `CNF`-Ereignis und gibt den konvertierten Wert aus.

Anwendungsszenarien

- Konvertierung von Sensorwerten (z.B. ADC-Rohdaten) in Gleitkommawerte für weitere Berechnungen.
- Integration in Steuerungssysteme, die unterschiedliche Datentypen verarbeiten müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken ist `F_USINT_TO_REAL` spezialisiert und daher effizienter für diese spezifische Konvertierung.
- Andere Blöcke wie `F_INT_TO_REAL` oder `F_DINT_TO_REAL` sind für andere Integer-Datentypen ausgelegt.

Fazit

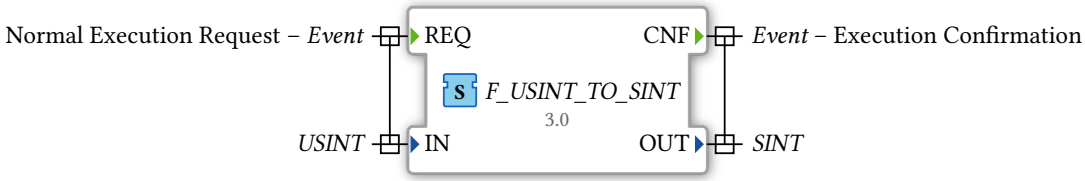
Der `F_USINT_TO_REAL`-Funktionsblock ist ein einfacher und effizienter Baustein für die Konvertierung von vorzeichenlosen 8-Bit-Integern in Gleitkommawerte. Seine Einfachheit und direkte Funktionsweise machen ihn ideal für Anwendungen, die eine schnelle und zuverlässige Typumwandlung erfordern.

F_USINT_TO_SINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_USINT_TO_SINT` dient der Konvertierung eines vorzeichenlosen 8-Bit-Integer-Werts (USINT) in einen vorzeichenbehafteten 8-Bit-Integer-Wert (SINT). Dieser Baustein ist besonders nützlich, wenn Daten zwischen Systemen ausgetauscht werden, die unterschiedliche Datentypen verwenden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Bei Aktivierung wird der Eingangswert `IN` verarbeitet.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgangswert `OUT` ist dann verfügbar.

Daten-Eingänge

- **IN** (USINT): Der vorzeichenlose 8-Bit-Integer-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (SINT): Der resultierende vorzeichenbehaftete 8-Bit-Integer-Wert nach der Konvertierung.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem er den USINT-Wert `IN` in einen SINT-Wert `OUT` umwandelt. Die Konvertierung erfolgt direkt bei Aktivierung des `REQ`-Ereignisses, und das Ergebnis wird über das `CNF`-Ereignis ausgegeben.

Technische Besonderheiten

- Die Konvertierung erfolgt ohne zusätzliche Parameter oder Konfiguration.
- Der Algorithmus ist in Structured Text (ST) implementiert und verwendet die Funktion `USINT_TO_SINT`.

Zustandsübersicht

1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Processing**: Führt die Konvertierung durch.
3. **Completed**: Sendet das `CNF`-Ereignis mit dem Ergebnis.

Anwendungsszenarien

- Datenkonvertierung in Steuerungssystemen.
- Kommunikation zwischen Geräten mit unterschiedlichen Datentypen.
- Vorverarbeitung von Sensordaten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Bausteine wie `F_INT_TO_REAL` oder `F_BOOL_TO_INT` bieten Konvertierungen zwischen anderen Datentypen.
- `F_USINT_TO_SINT` ist spezialisiert auf die Konvertierung zwischen 8-Bit-Integer-Typen.

Fazit

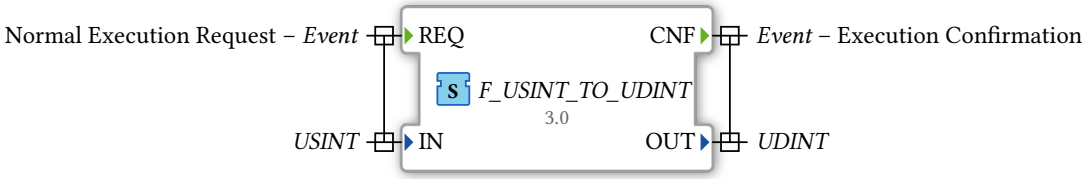
Der `F_USINT_TO_SINT`-Funktionsblock ist ein einfacher und effizienter Baustein für die Konvertierung zwischen vorzeichenlosen und vorzeichenbehafteten 8-Bit-Integer-Werten. Seine klare Schnittstelle und direkte Funktionsweise machen ihn ideal für den Einsatz in verschiedenen Steuerungs- und Kommunikationsanwendungen.

F_USINT_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_USINT_TO_UDINT` dient der Konvertierung eines `USINT`-Wertes (8-Bit unsigned integer) in einen `UDINT`-Wert (32-Bit unsigned integer). Dieser Baustein ist besonders nützlich in Szenarien, wo eine Typumwandlung ohne Datenverlust erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verbunden.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verbunden.

Daten-Eingänge

- **IN** (`USINT`): Der Eingangswert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (`UDINT`): Der konvertierte Ausgangswert.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt eine direkte Zuweisung des Eingangswertes `IN` vom Typ `USINT` zum Ausgangswert `OUT` vom Typ `UDINT` durch. Da `UDINT` einen größeren Wertebereich als `USINT` abdeckt, erfolgt die Konvertierung ohne Datenverlust.

Technische Besonderheiten

- **Algorithmus**: Der Algorithmus `REQ` führt die Zuweisung `OUT := IN` aus.
- **Datenintegrität**: Durch die Konvertierung in einen größeren Datentyp bleibt der ursprüngliche Wert erhalten.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auftreten des `REQ`-Ereignisses.

Anwendungsszenarien

- **Typumwandlung**: In Steuerungssystemen, wo unterschiedliche Datentypen verarbeitet werden müssen.
- **Datenverarbeitung**: Bei der Kommunikation zwischen Komponenten, die unterschiedliche Datentypen verwenden.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_USINT_TO_INT**: Konvertiert `USINT` zu `INT`, jedoch mit potenziellem Vorzeichenwechsel.
- **F_USINT_TO_UINT**: Konvertiert `USINT` zu `UINT` (16-Bit), was einen kleineren Wertebereich als `UDINT` bietet.

🛠️ Zugehörige Übungen

- Uebung_111

Fazit

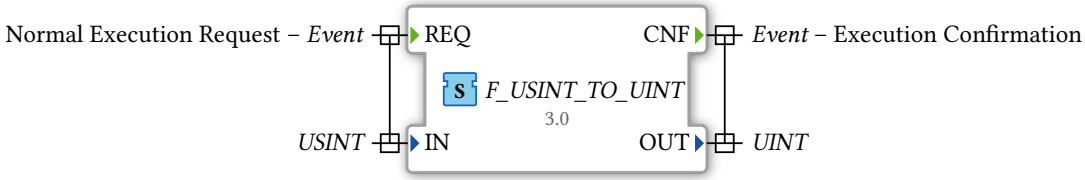
Der `F_USINT_TO_UDINT`-Funktionsblock ist ein einfacher und effizienter Baustein zur sicheren Konvertierung von `USINT` nach `UDINT`. Durch seine direkte Zuweisung und den größeren Wertebereich des Ausgangstyps ist er ideal für Anwendungen, bei denen Datenintegrität gewährleistet sein muss.

F_USINT_TO_UINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_USINT_TO_UINT` dient der Konvertierung eines `USINT`-Wertes (Unsigned Short Integer) in einen `UINT`-Wert (Unsigned Integer). Dieser Baustein ist besonders nützlich in Szenarien, wo eine Typumwandlung ohne Datenverlust erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Dieser Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `USINT`, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgangswert vom Typ `UINT`, der das Ergebnis der Konvertierung enthält.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt eine direkte Zuweisung des Eingangswertes `IN` vom Typ `USINT` zum Ausgangswert `OUT` vom Typ `UINT` durch. Da `USINT` einen kleineren Wertebereich als `UINT` hat, erfolgt die Konvertierung ohne Datenverlust.

Der Algorithmus wird durch das Ereignis `REQ` ausgelöst und gibt das Ergebnis über das Ereignis `CNF` aus.

Technische Besonderheiten

- Der Funktionsblock ist in der Programmiersprache ST (Structured Text) implementiert.
- Die Konvertierung erfolgt ohne zusätzliche Berechnungen oder Prüfungen, da `USINT` automatisch in `UINT` umgewandelt werden kann.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo `USINT`-Werte in `UINT` umgewandelt werden müssen.
- Verwendung in Datenverarbeitungsketten, wo verschiedene Datentypen kompatibel gemacht werden müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen wie `F_INT_TO_REAL` oder `F_BOOL_TO_STRING` ist dieser Baustein spezifisch für die Umwandlung zwischen zwei vorzeichenlosen Ganzzahltypen.
- Ähnliche Bausteine könnten zusätzliche Prüfungen oder Skalierungen durchführen, was hier nicht notwendig ist.

Fazit

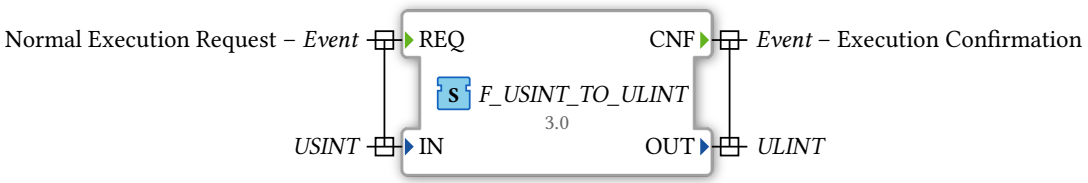
Der `F_USINT_TO_UINT`-Funktionsblock ist ein einfacher und effizienter Baustein zur Konvertierung von `USINT` nach `UINT`. Er eignet sich ideal für Anwendungen, bei denen eine schnelle und verlustfreie Typumwandlung erforderlich ist.

F_USINT_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_USINT_TO_ULINT` dient der Konvertierung eines `USINT`-Wertes (8-Bit unsigned integer) in einen `ULINT`-Wert (64-Bit unsigned integer). Diese Konvertierung ist nützlich, wenn Daten zwischen Systemen mit unterschiedlichen Wortbreiten ausgetauscht oder verarbeitet werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge
- `REQ`: Startet die Ausführung der Konvertierung. Dieser Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge
- `CNF`: Signalisiert den Abschluss der Konvertierung. Dieser Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge
- `IN` (USINT): Der Eingangswert, der konvertiert werden soll.

Daten-Ausgänge
- `OUT` (ULINT): Der konvertierte Ausgangswert.

Adapter
Keine Adapter vorhanden.

Funktionsweise
Der Funktionsblock führt eine direkte Konvertierung des Eingangswertes `IN` vom Typ `USINT` in den Ausgangswert `OUT` vom Typ `ULINT` durch. Die Konvertierung erfolgt ohne Datenverlust, da der Wertebereich von `USINT` (0 bis 255) vollständig in `ULINT` (0 bis 2^64-1) abgebildet werden kann.

Technische Besonderheiten
- Die Konvertierung erfolgt in einem einzigen Algorithmus (`REQ`), der bei Auslösung des `REQ`-Ereignisses ausgeführt wird.
- Der Algorithmus ist in ST (Structured Text) implementiert und besteht aus einer einfachen Zuweisung: `OUT := IN;`.

Zustandsübersicht
Der Funktionsblock hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

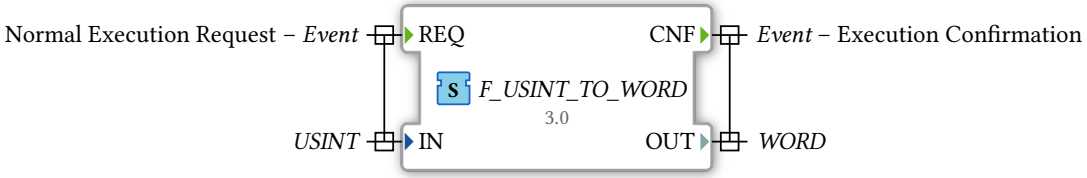
Anwendungsszenarien
- Datenkonvertierung in Steuerungssystemen, die unterschiedliche Wortbreiten verwenden.
- Integration von Peripheriegeräten, die Daten in unterschiedlichen Formaten liefern.
- Allgemeine Typkonvertierung in IEC 61131-3-basierten Programmen.

⚖️ Vergleich mit ähnlichen Bausteinen
- Ähnliche Funktionsblöcke wie `F_SINT_TO_LINT` oder `F_UINT_TO_UDINT` führen ähnliche Konvertierungen durch, jedoch zwischen anderen Datentypen.
- `F_USINT_TO_ULINT` ist spezifisch für die Konvertierung von 8-Bit zu 64-Bit unsigned integers optimiert.

Fazit
Der Funktionsblock `F_USINT_TO_ULINT` bietet eine einfache und effiziente Möglichkeit, Werte vom Typ `USINT` in `ULINT` zu konvertieren. Durch seine klare Schnittstelle und direkte Funktionsweise ist er ideal für Anwendungen, die eine Typkonvertierung ohne zusätzliche Logik erfordern.

F_USINT_TO_WORD
• – ◦ • – ◦ • – ◦ •

Einleitung
Der Funktionsblock `F_USINT_TO_WORD` dient der Konvertierung eines vorzeichenlosen 8-Bit-Integer-Werts (USINT) in einen 16-Bit-Wort-Wert (WORD). Diese Konvertierung ist insbesondere in Szenarien erforderlich, in denen Daten unterschiedlicher Bitbreiten verarbeitet oder übertragen werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (USINT): Der vorzeichenlose 8-Bit-Integer-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT** (WORD): Der resultierende 16-Bit-Wort-Wert nach der Konvertierung.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, indem er den USINT-Wert von `IN` nimmt und ihn in einen WORD-Wert umwandelt. Dies geschieht durch direkte Zuweisung unter Verwendung der Funktion `USINT_TO_WORD(IN)`. Die Konvertierung wird durch das Ereignis `REQ` ausgelöst und durch `CNF` bestätigt.

Technische Besonderheiten

- Die Konvertierung ist verlustfrei, da ein USINT (0-255) problemlos in ein WORD (0-65535) umgewandelt werden kann.
- Der Funktionsblock ist deterministisch und hat keine internen Zustände.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Ausführung ist rein ereignisgesteuert:

1. Bei Empfang von `REQ` wird die Konvertierung durchgeführt.
2. Nach Abschluss wird `CNF` ausgelöst.

Anwendungsszenarien

- Datenaufbereitung für Kommunikationsprotokolle, die WORD-Daten erwarten.
- Integration von Sensordaten, die als USINT vorliegen, in Systeme, die WORD verarbeiten.
- Allgemeine Typkonvertierung in Automatisierungsanwendungen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken ist `F_USINT_TO_WORD` spezialisiert und optimiert für die spezifische Konvertierung von USINT zu WORD.
- Andere Blöcke könnten zusätzliche Funktionen wie Skalierung oder Fehlerbehandlung bieten, sind aber komplexer.

Fazit

Der `F_USINT_TO_WORD` Funktionsblock ist ein einfacher und effizienter Baustein für die spezifische Konvertierung von USINT zu WORD. Seine Einfachheit und Determinismus machen ihn zu einer zuverlässigen Wahl in Szenarien, die diese Konvertierung erfordern.

USINT Konvertierungen

Hier finden Sie Funktionen zur Konvertierung des Datentyps USINT.

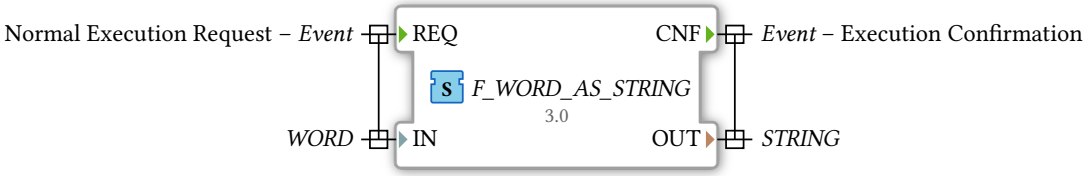
🛠️ Zugehörige Übungen
- Funk_IX_TO_DataPanel_MI_QX
- Funk_IXA_TO_DataPanel_MI_QXA
- IX_TO_Aux_QX
- IXA_TO_Aux_QXA
- IX_TO_logiBUS_LED_strip_QX
- SIN_TO_PCAN_Callback_Byte

WORD

F_WORD_AS_STRING

• – ◦ • – ◦ • – ◦ •

Einleitung
Der Funktionsblock `F_WORD_AS_STRING` dient der Konvertierung eines `WORD`-Datentyps in einen `STRING`. Dies ist besonders nützlich, wenn numerische Werte in einer für Menschen lesbaren Form ausgegeben oder weiterverarbeitet werden sollen.



Schnittstellenstruktur

Ereignis-Eingänge
- **REQ**: Startet die Konvertierung. Bei Auslösung dieses Ereignisses wird der Wert des Daten-Eingangs `IN` verarbeitet.

Ereignis-Ausgänge
- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieses Ereignis wird zusammen mit dem konvertierten Wert am Daten-Ausgang `OUT` ausgegeben.

Daten-Eingänge
- **IN**: Erwartet einen Eingabewert vom Typ `WORD`, der in einen `STRING` umgewandelt werden soll.

Daten-Ausgänge
- **OUT**: Gibt den konvertierten Wert als `STRING` aus.

Adapter
- Keine Adapter vorhanden.

Funktionsweise
Der Funktionsblock nutzt den Algorithmus `REQ`, um den `WORD`-Wert am Eingang `IN` in einen `STRING` umzuwandeln. Die Konvertierung erfolgt durch die Funktion `WORD_AS_STRING(IN)`. Nach erfolgreicher Umwandlung wird das Ereignis `CNF` ausgelöst und der konvertierte Wert am Ausgang `OUT` bereitgestellt.

Technische Besonderheiten
- Der Funktionsblock ist einfach aufgebaut und benötigt keine zusätzlichen Parameter oder komplexe Logik.
- Die Konvertierung erfolgt direkt und ohne Zwischenschritte.

Zustandsübersicht
1. **Initialisierung**: Der Funktionsblock wartet auf das Ereignis `REQ`.
2. **Verarbeitung**: Bei Empfang von `REQ` wird der Wert von `IN` konvertiert.
3. **Abschluss**: Nach der Konvertierung wird `CNF` ausgelöst und der Wert an `OUT` ausgegeben.

Anwendungsszenarien

- Anzeige von numerischen Werten in Benutzeroberflächen.
- Protokollierung von `WORD`-Werten in lesbarer Form.
- Weiterverarbeitung von numerischen Werten in Systemen, die `STRING`-Daten erwarten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken ist `F_WORD_AS_STRING` spezialisiert auf die Umwandlung von `WORD` zu `STRING` und daher effizienter für diesen spezifischen Anwendungsfall.
- Andere Blöcke könnten zusätzliche Parameter oder komplexere Logik beinhalten, was hier nicht notwendig ist.

Fazit

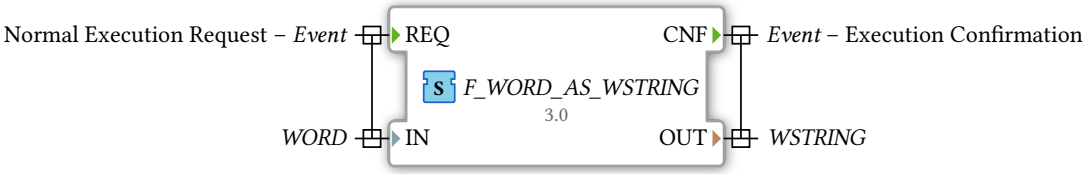
Der `F_WORD_AS_STRING`-Funktionsblock ist ein einfaches und effizientes Werkzeug zur Konvertierung von `WORD`-Werten in `STRING`. Seine klare Schnittstelle und direkte Funktionsweise machen ihn ideal für Anwendungen, bei denen eine schnelle und unkomplizierte Umwandlung benötigt wird.

F_WORD_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_WORD_AS_WSTRING` dient der Konvertierung eines `WORD`-Datentyps in einen `WSTRING`-Datentyp. Dieser Baustein ist besonders nützlich in Szenarien, wo numerische Werte in eine Zeichenkette umgewandelt werden müssen, beispielsweise für die Anzeige oder Protokollierung.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Eingang ausgelöst, wird der Wert am Daten-Eingang `IN` verarbeitet.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der konvertierte Wert steht am Daten-Ausgang `OUT` zur Verfügung.

Daten-Eingänge

- **IN** (`WORD`): Der Eingangswert, der in einen `WSTRING` umgewandelt werden soll.

Daten-Ausgänge

- **OUT** (`WSTRING`): Der Ausgangswert nach der Konvertierung.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Bei Auslösung des Ereigniseingangs `REQ` wird der Algorithmus ausgeführt, der den Wert am Eingang `IN` vom Typ `WORD` in einen `WSTRING` umwandelt. Das Ergebnis wird am Ausgang `OUT` bereitgestellt und das Ereignis `CNF` signalisiert die erfolgreiche Ausführung.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne Zwischenschritte.
- Der Baustein ist für einfache Konvertierungen optimiert und benötigt keine zusätzliche Konfiguration.

Zustandsübersicht

1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Processing**: Führt die Konvertierung durch.
3. **Completed**: Signalisiert mit `CNF` den Abschluss.

Anwendungsszenarien

- Anzeige von numerischen Werten in einer Benutzeroberfläche.
- Protokollierung von `WORD`-Werten als lesbare Zeichenketten.
- Datenkonvertierung für die Kommunikation mit Systemen, die `WSTRING` erwarten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsbausteinen ist `F_WORD_AS_WSTRING` spezialisiert und daher effizienter für diese spezifische Aufgabe.
- Andere Bausteine könnten zusätzliche Parameter oder komplexere Konvertierungen unterstützen, was hier nicht notwendig ist.

Fazit

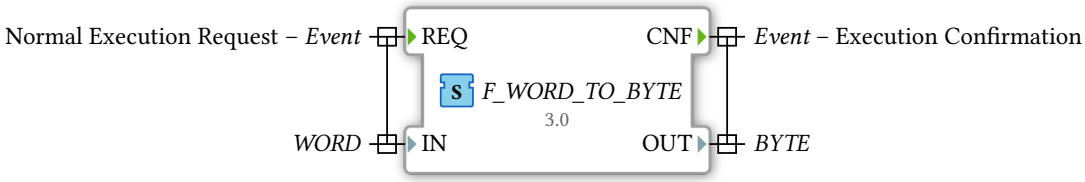
Der `F_WORD_AS_WSTRING`-Funktionsblock ist ein einfacher und effizienter Baustein für die Konvertierung von `WORD` zu `WSTRING`. Seine Stärke liegt in der Einfachheit und direkten Anwendbarkeit für spezifische Konvertierungsaufgaben.

F_WORD_TO_BYTE

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_WORD_TO_BYTE` dient zur Konvertierung eines `WORD`-Datentyps in einen `BYTE`-Datentyp. Er ist Teil des Pakets `iec61131::conversion` und eignet sich für Anwendungen, bei denen eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung des Funktionsblocks. Dieser Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (`WORD`): Der Eingangswert, der von `WORD` in `BYTE` konvertiert werden soll.

Daten-Ausgänge

- **OUT** (`BYTE`): Der konvertierte Ausgangswert vom Typ `BYTE`.

Adapter

Es sind keine Adapter für diesen Funktionsblock definiert.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, wenn das Ereignis `REQ` empfangen wird. Dabei wird der Wert am Eingang `IN` vom Typ `WORD` mithilfe der Funktion `WORD_TO_BYTE` in einen `BYTE`-Wert

umgewandelt und am Ausgang `OUT` ausgegeben. Nach Abschluss der Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher Funktionsblock (`SimpleFB`) und verwendet einen ST-Algorithmus für die Konvertierung.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Verzögerung.

Zustandsübersicht

1. **Idle-Zustand**: Der Funktionsblock wartet auf das Ereignis `REQ`.
2. **Konvertierungszustand**: Bei Empfang von `REQ` wird die Konvertierung durchgeführt und `CNF` ausgelöst.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, bei denen `WORD`-Daten in `BYTE`-Daten umgewandelt werden müssen.
- Integration in größere Steuerungssysteme, die eine kompatible Datentypkonvertierung erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsfunktionen bietet dieser Funktionsblock eine spezialisierte und optimierte Lösung für die Umwandlung von `WORD` zu `BYTE`.
- Andere ähnliche Blöcke könnten zusätzliche Funktionen wie Fehlerbehandlung oder optionale Parameter bieten, was hier nicht der Fall ist.

Fazit

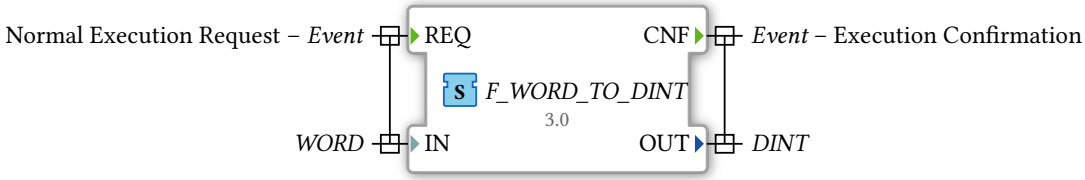
Der `F_WORD_TO_BYTE`-Funktionsblock ist eine effiziente und einfache Lösung für die Konvertierung von `WORD`- zu `BYTE`-Daten. Er ist besonders geeignet für Anwendungen, bei denen eine schnelle und direkte Typumwandlung erforderlich ist.

F_WORD_TO_DINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_WORD_TO_DINT` dient der Konvertierung eines `WORD`-Datentyps in einen `DINT`-Datentyp. Dieser Baustein ist besonders nützlich in Szenarien, bei denen eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist, z.B. bei der Verarbeitung von Daten aus verschiedenen Quellen oder bei der Anpassung von Daten für unterschiedliche Systemkomponenten.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Dieser Eingang löst die Konvertierung aus. Er ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Dieser Ausgang signalisiert den Abschluss der Konvertierung. Er ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Eingangsvariable vom Typ `WORD`, die den zu konvertierenden Wert enthält.

Daten-Ausgänge

- **OUT**: Ausgangsvariable vom Typ `DINT`, die das Ergebnis der Konvertierung enthält.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Der Funktionsblock `F_WORD_TO_DINT` führt die Konvertierung von einem `WORD`- zu einem `DINT`-Datentyp durch. Bei Auslösung des `REQ`-Ereignisses wird der Wert am Eingang `IN` gelesen und mit der Funktion `WORD_TO_DINT` konvertiert. Das Ergebnis wird an den Ausgang `OUT` ausgegeben und das `CNF`-Ereignis signalisiert den Abschluss der Operation.

Technische Besonderheiten

- Der Funktionsblock ist einfach aufgebaut und enthält nur einen Algorithmus für die Konvertierung.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Verzögerung, sobald das `REQ`-Ereignis eintrifft.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung wird bei jedem `REQ`-Ereignis neu ausgeführt.

Anwendungsszenarien

- Typumwandlung in Steuerungssystemen, bei denen Daten unterschiedlicher Typen verarbeitet werden müssen.
- Integration von Komponenten, die unterschiedliche Datentypen verwenden.
- Datenaufbereitung für weitere Verarbeitungsschritte.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen wie `F_INT_TO_REAL` oder `F_BOOL_TO_BYTE` ist `F_WORD_TO_DINT` spezifisch auf die Umwandlung von `WORD` zu `DINT` ausgelegt.
- Ähnliche Bausteine könnten zusätzliche Funktionen wie Bereichsprüfungen oder Skalierungen bieten, was hier nicht der Fall ist.

Fazit

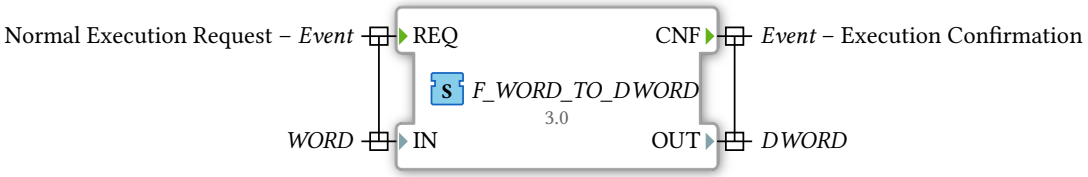
Der Funktionsblock `F_WORD_TO_DINT` ist ein einfaches und effizientes Werkzeug zur Konvertierung von `WORD`- zu `DINT`-Werten. Seine klare Schnittstellenstruktur und direkte Funktionsweise machen ihn zu einer zuverlässigen Komponente in Steuerungsanwendungen, bei denen solche Typumwandlungen erforderlich sind.

F_WORD_TO_DWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_WORD_TO_DWORD` dient der Konvertierung eines `WORD`-Wertes in einen `DWORD`-Wert. Er ist Teil des Pakets `iec61131::conversion` und ermöglicht eine einfache und effiziente Typumwandlung.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung des Funktionsblocks. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingang vom Typ `WORD`, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgang vom Typ `DWORD`, der das Ergebnis der Konvertierung enthält.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `WORD` zu `DWORD` durch, sobald das Ereignis `REQ` empfangen wird. Die Konvertierung erfolgt mittels der Funktion `WORD_TO_DWORD(IN)`. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher Funktionsblock (`SimpleFB`) und verwendet einen ST-Algorithmus für die Konvertierung.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Verarbeitungsschritte.

Zustandsübersicht

1. **Idle**: Wartet auf das Ereignis `REQ`.
2. **Processing**: Führt die Konvertierung durch.
3. **Completed**: Sendet das Ereignis `CNF` und kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo `WORD`-Werte in `DWORD`-Werte umgewandelt werden müssen.
- Integration in größere Steuerungssysteme, die eine Typkompatibilität erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen bietet `F_WORD_TO_DWORD` eine spezialisierte Funktionalität für die Umwandlung von `WORD` zu `DWORD` ohne zusätzliche Features oder Komplexität.

Fazit

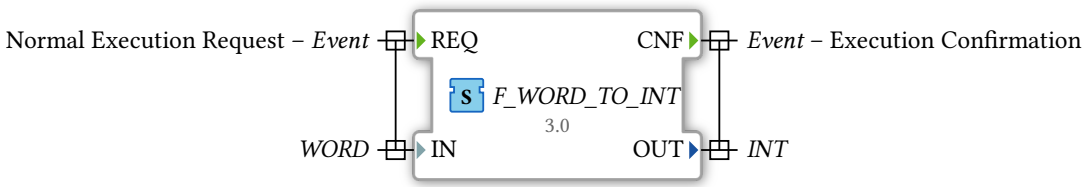
Der `F_WORD_TO_DWORD` Funktionsblock ist ein einfaches und effizientes Werkzeug für die Typumwandlung in IEC 61131-3-basierten Steuerungssystemen. Seine klare Schnittstellenstruktur und direkte Funktionsweise machen ihn zu einer zuverlässigen Lösung für Konvertierungsaufgaben.

F_WORD_TO_INT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_WORD_TO_INT` dient der Konvertierung eines `WORD`-Datentyps in einen `INT`-Datentyp. Er ist Teil des Pakets `iec61131::conversion` und ermöglicht eine einfache und effiziente Typumwandlung innerhalb von IEC 61131-3-basierten Steuerungssystemen.



Schnittstellenstruktur

Ereignis-Eingänge
- **REQ**: Startet die Konvertierung. Dieser Ereigniseingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge
- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ereignisausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge
- **IN**: Der Eingang vom Typ `WORD`, der in einen `INT`-Wert umgewandelt werden soll.

Daten-Ausgänge
- **OUT**: Der Ausgang vom Typ `INT`, der das Ergebnis der Konvertierung enthält.

Adapter
- Keine Adapter vorhanden.

Funktionsweise
Der Funktionsblock führt die Konvertierung aus, sobald das Ereignis `REQ` ausgelöst wird. Die Umwandlung erfolgt mithilfe der Funktion `WORD_TO_INT`, die den Wert des Eingangs `IN` in einen `INT`-Wert konvertiert und das Ergebnis an den Ausgang `OUT` weitergibt. Nach Abschluss der Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten
- Der Funktionsblock ist ein einfacher Funktionsblock (`SimpleFB`) und verwendet einen ST-Algorithmus für die Konvertierung.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Verzögerung.

Zustandsübersicht
Der Funktionsblock besitzt keine internen Zustände. Die Konvertierung wird bei jedem `REQ`-Ereignis neu ausgeführt.

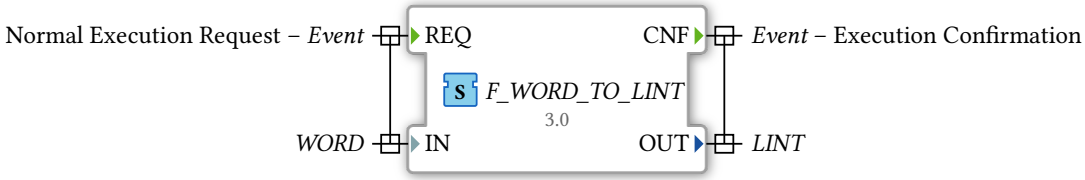
Anwendungsszenarien
- Typumwandlung in Steuerungsanwendungen, wo `WORD`-Daten in `INT`-Daten umgewandelt werden müssen.
- Integration in größere Steuerungssysteme, die eine Typkompatibilität erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen
- Ähnliche Funktionsblöcke könnten `F_BYTE_TO_INT` oder `F_DWORD_TO_INT` sein, die jedoch andere Datentypen konvertieren.
- Im Gegensatz zu komplexeren Konvertierungsblöcken bietet `F_WORD_TO_INT` eine einfache und direkte Lösung für die spezifische Umwandlung von `WORD` zu `INT`.

Fazit
Der `F_WORD_TO_INT`-Funktionsblock ist ein effizientes und einfach zu verwendendes Werkzeug für die Typumwandlung in IEC 61131-3-basierten Systemen. Seine klare Schnittstelle und direkte Funktionsweise machen ihn ideal für Anwendungen, die eine schnelle und zuverlässige Konvertierung erfordern.

F_WORD_TO_LINT
• – ◦ • – ◦ • – ◦ •

Einleitung
Der Funktionsblock `F_WORD_TO_LINT` dient der Konvertierung eines `WORD`-Datentyps in einen `LINT`-Datentyp. Er ist Teil des Pakets `iec61131::conversion` und ermöglicht eine einfache und effiziente Typumwandlung innerhalb von IEC 61499-basierten Steuerungssystemen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung der Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den erfolgreichen Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingang vom Typ `WORD`, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgang vom Typ `LINT`, der das Ergebnis der Konvertierung enthält.

Adapter

- Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `WORD` zu `LINT` durch, sobald das Ereignis `REQ` ausgelöst wird. Die Konvertierung erfolgt mittels der Funktion `WORD_TO_LINT`, die den Eingabewert `IN` umwandelt und das Ergebnis an `OUT` ausgibt. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher Algorithmus ohne Zustandsmaschine.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Verzögerung.
- Der Block ist für den Einsatz in Echtzeit-Steuerungssystemen optimiert.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge oder komplexe Zustandslogik. Die Ausführung erfolgt direkt bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo `WORD`-Daten in `LINT` umgewandelt werden müssen.
- Integration in größere Steuerungssysteme, die unterschiedliche Datentypen verarbeiten.
- Verwendung in Datenverarbeitungsketten, wo eine Erweiterung der Bitbreite erforderlich ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken ist `F_WORD_TO_LINT` spezialisiert und daher effizienter.
- Andere Blöcke wie `F_INT_TO_LINT` oder `F_DWORD_TO_LINT` bieten ähnliche Funktionalität, aber für andere Datentypen.

Fazit

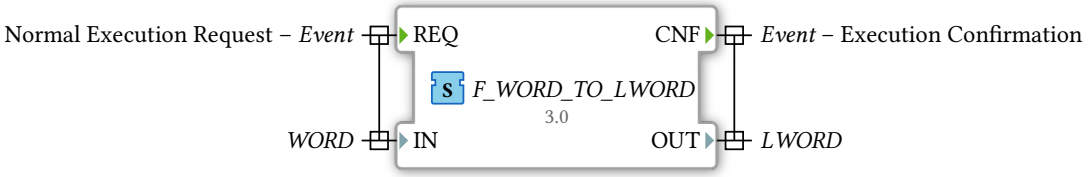
Der `F_WORD_TO_LINT`-Funktionsblock ist ein einfacher und effizienter Baustein für die Typumwandlung von `WORD` zu `LINT`. Er eignet sich ideal für Anwendungen, die eine schnelle und direkte Konvertierung erfordern, und ist leicht in bestehende Steuerungssysteme integrierbar.

F_WORD_TO_LWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_WORD_TO_LWORD` dient der Konvertierung eines `WORD`-Datentyps in einen `LWORD`-Datentyp. Dieser Baustein ist besonders nützlich in Szenarien, wo eine Erweiterung der Datenbreite erforderlich ist, beispielsweise bei der Kommunikation zwischen Systemen mit unterschiedlichen Datenwortgrößen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Dieser Ereigniseingang ist mit dem Dateneingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ereignisausgang ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Erwartet eine Eingabe vom Typ `WORD`, die in einen `LWORD` konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Gibt das konvertierte Ergebnis vom Typ `LWORD` aus.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` empfangen wird. Die Konvertierung erfolgt mittels der Funktion `WORD_TO_LWORD`, welche den Eingabewert `IN` vom Typ `WORD` in den Ausgabewert `OUT` vom Typ `LWORD` umwandelt. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Die Konvertierung ist direkt und ohne zusätzliche Parameter oder Zustandsverwaltung.
- Der Baustein ist für die Verwendung in Echtzeitsystemen geeignet, da er keine Verzögerungen oder komplexe Berechnungen beinhaltet.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Operation ist stateless und wird bei jedem `REQ`-Ereignis neu ausgeführt.

Anwendungsszenarien

- Erweiterung der Datenbreite für nachfolgende Verarbeitungsschritte.
- Kompatibilitätssicherung zwischen Systemen mit unterschiedlichen Wortgrößen.
- Einfache Datentypkonvertierung in Steuerungs- und Automatisierungssystemen.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu generischen Konvertierungsbausteinen bietet `F_WORD_TO_LWORD` eine spezialisierte und effiziente Lösung für die Konvertierung von `WORD` zu `LWORD`. Andere Bausteine könnten allgemeiner sein, aber weniger optimiert für diese spezifische Aufgabe.

Fazit

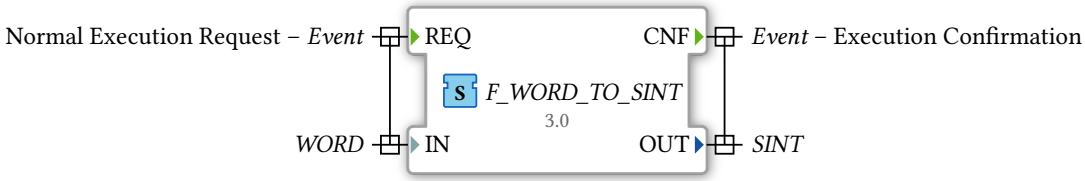
Der `F_WORD_TO_LWORD` Funktionsblock ist ein einfacher, aber effektiver Baustein für die spezifische Konvertierung von `WORD` zu `LWORD`. Seine Einfachheit und direkte Funktionsweise machen ihn zu einer zuverlässigen Wahl in vielen Automatisierungs- und Steuerungsszenarien.

F_WORD_TO_SINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_WORD_TO_SINT` dient der Konvertierung eines `WORD`-Datentyps in einen `SINT`-Datentyp. Er ist Teil des `iec61131::conversion`-Pakets und ermöglicht eine einfache und effiziente Typumwandlung innerhalb von 4diac-IDE-Anwendungen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung der Konvertierung. Dieser Ereigniseingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ereignisausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (`WORD`): Der Eingangswert, der von `WORD` zu `SINT` konvertiert werden soll.

Daten-Ausgänge

- **OUT** (`SINT`): Der konvertierte Ausgangswert vom Typ `SINT`.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Bei Auslösung des `REQ`-Ereignisses wird der Algorithmus ausgeführt, der den `WORD`-Wert am Eingang `IN` in einen `SINT`-Wert konvertiert und das Ergebnis am Ausgang `OUT` ausgibt. Anschließend wird das `CNF`-Ereignis ausgelöst, um den erfolgreichen Abschluss der Konvertierung zu signalisieren.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher Funktionsblock (`SimpleFB`) und führt die Konvertierung in einem einzigen Algorithmusschritt durch.
- Die Konvertierung erfolgt mittels der eingebauten Funktion `WORD_TO_SINT`.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Ausführung erfolgt direkt bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo `WORD`-Daten in `SINT`-Daten umgewandelt werden müssen.
- Integration in größere Steuerungssysteme, die unterschiedliche Datentypen verarbeiten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_INT_TO_SINT` oder `F_DWORD_TO_SINT` bieten Konvertierungen zwischen anderen Datentypen, sind aber in ihrer Funktionsweise vergleichbar.

Fazit

Der `F_WORD_TO_SINT`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, `WORD`-Daten in `SINT`-Daten umzuwandeln. Durch seine klare Schnittstellenstruktur und direkte Funktionsweise ist er ideal für Anwendungen, die eine schnelle und zuverlässige Typumwandlung erfordern.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

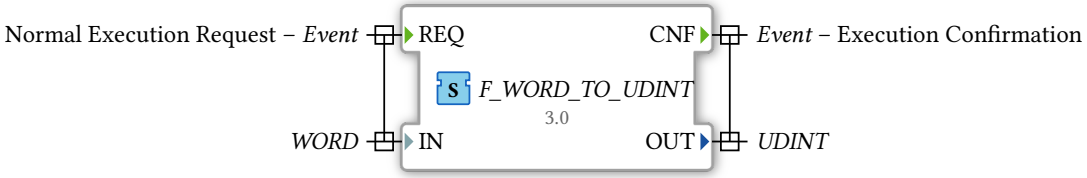
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

F_WORD_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_WORD_TO_UDINT` dient der Konvertierung eines `WORD`-Datentyps in einen `UDINT`-Datentyp. Dieser Baustein ist besonders nützlich in Szenarien, bei denen eine Typumwandlung erforderlich ist, um Daten für weitere Verarbeitungsschritte vorzubereiten.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Bei Auslösung dieses Ereignisses wird der Eingangswert `IN` verarbeitet.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung und gibt das Ergebnis über den Ausgang `OUT` aus.

Daten-Eingänge

- **IN**: Der Eingangswert vom Typ `WORD`, der in einen `UDINT` umgewandelt werden soll.

Daten-Ausgänge

- **OUT**: Das Ergebnis der Konvertierung als `UDINT`.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Bei Auslösung des `REQ`-Ereignisses wird der Algorithmus gestartet, der den `WORD`-Wert am Eingang `IN` in einen `UDINT`-Wert konvertiert. Das Ergebnis wird am Ausgang `OUT` bereitgestellt und das `CNF`-Ereignis signalisiert die erfolgreiche Ausführung.

Technische Besonderheiten

- Der Funktionsblock verwendet die eingebaute Funktion `WORD_TO_UDINT` zur Konvertierung.
- Die Konvertierung erfolgt ohne zusätzliche Parameter oder komplexe Logik, was eine schnelle und effiziente Verarbeitung ermöglicht.

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf das `REQ`-Ereignis.
2. **Verarbeitungszustand**: Führt die Konvertierung durch.
3. **Abschlusszustand**: Sendet das `CNF`-Ereignis und gibt das Ergebnis aus.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen.
- Vorbereitung von Daten für weitere Berechnungen oder Speicherung.
- Integration in größere Automatisierungslösungen, bei denen unterschiedliche Datentypen verarbeitet werden müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_INT_TO_REAL` oder `F_BOOL_TO_BYTE` bieten ähnliche Funktionalitäten für andere Datentypen.
- `F_WORD_TO_UDINT` ist spezialisiert auf die Konvertierung von `WORD` zu `UDINT` und bietet daher eine optimierte Lösung für diesen spezifischen Anwendungsfall.

Fazit

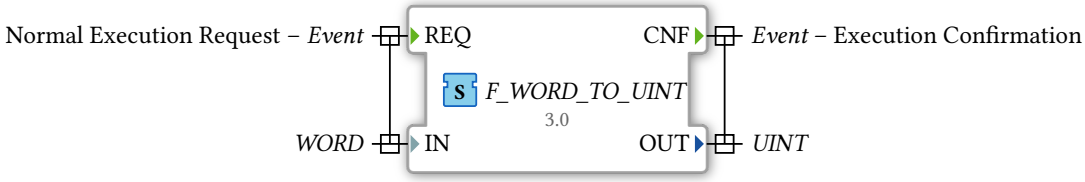
Der `F_WORD_TO_UDINT`-Funktionsblock ist ein einfacher und effizienter Baustein zur Konvertierung von `WORD`- in `UDINT`-Werte. Seine klare Schnittstelle und einfache Funktionsweise machen ihn zu einer zuverlässigen Komponente in Automatisierungsprojekten.

F_WORD_TO_UINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_WORD_TO_UINT` dient der Konvertierung eines `WORD`-Datentyps in einen `UINT`-Datentyp. Diese Konvertierung ist besonders nützlich in Szenarien, wo eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist, z.B. bei der Kommunikation zwischen verschiedenen Systemen oder bei der Verarbeitung von Daten unterschiedlicher Quellen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Dieser Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Der Eingang vom Typ `WORD`, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Der Ausgang vom Typ `UINT`, der das Ergebnis der Konvertierung enthält.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `WORD` zu `UINT` durch, sobald das Ereignis `REQ` empfangen wird. Die Konvertierung erfolgt mittels der Funktion `WORD_TO_UINT(IN)`. Nach erfolgreicher Konvertierung wird das Ergebnis über den Ausgang `OUT` ausgegeben und das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Der Funktionsblock ist einfach aufgebaut und enthält nur einen Algorithmus für die Konvertierung.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Verarbeitungsschritte.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Konvertierung wird bei jedem `REQ`-Ereignis neu ausgeführt.

Anwendungsszenarien

- Typumwandlung in Steuerungssystemen.
- Datenverarbeitung in industriellen Automatisierungslösungen.
- Kommunikation zwischen Systemen mit unterschiedlichen Datentypen.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Konvertierungsbausteinen ist `F_WORD_TO_UINT` spezifisch auf die Umwandlung von `WORD` zu `UINT` ausgelegt. Andere Bausteine könnten zusätzliche Funktionen oder eine breitere Palette von Konvertierungen anbieten, sind dafür aber möglicherweise komplexer im Aufbau.

Fazit

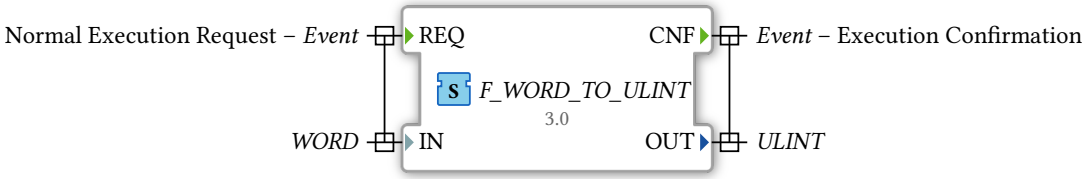
Der `F_WORD_TO_UINT` Funktionsblock ist ein einfaches und effizientes Werkzeug für die spezifische Konvertierung von `WORD` zu `UINT`. Seine Einfachheit macht ihn besonders geeignet für Anwendungen, bei denen eine schnelle und direkte Typumwandlung erforderlich ist.

F_WORD_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_WORD_TO_ULINT` dient der Konvertierung eines `WORD`-Datentyps in einen `ULINT`-Datentyp. Dieser Baustein ist besonders nützlich in Szenarien, bei denen eine Erweiterung oder Umwandlung von Datenformaten erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Dieser Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Erwartet einen `WORD`-Wert, der konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Gibt den konvertierten `ULINT`-Wert aus.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `WORD` nach `ULINT` durch, sobald das `REQ`-Ereignis eintritt. Die Konvertierung erfolgt mittels der integrierten Funktion `WORD_TO_ULINT`. Nach erfolgreicher Konvertierung wird das `CNF`-Ereignis ausgelöst.

Technische Besonderheiten

- Der Baustein ist Teil des Pakets `iec61131::conversion`.

- Die Konvertierung erfolgt ohne zusätzliche Parameter oder Einstellungen.

Zustandsübersicht

1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Processing**: Führt die Konvertierung durch.
3. **Done**: Sendet das `CNF`-Ereignis und gibt das Ergebnis aus.

Anwendungsszenarien

- Datenverarbeitung in Steuerungssystemen, wo unterschiedliche Datentypen kompatibel gemacht werden müssen.
- Erweiterung von Daten für weitere Berechnungen oder Speicherung.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Bausteine könnten direkte Konvertierungen zwischen anderen Datentypen anbieten, z.B. `F_WORD_TO_INT`.
- Dieser Baustein spezialisiert sich auf die Konvertierung zu `ULINT`, was besonders für große, vorzeichenlose Zahlen relevant ist.

Fazit

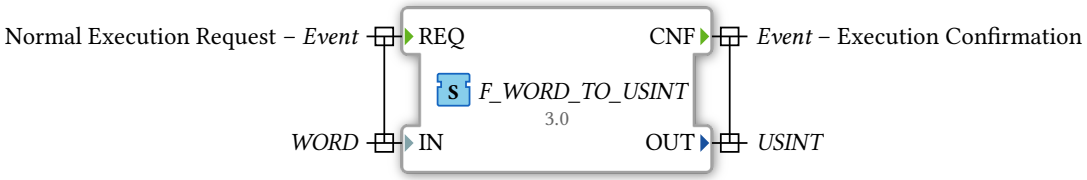
Der `F_WORD_TO_ULINT`-Funktionsblock ist ein einfacher, aber effektiver Baustein für die Konvertierung von `WORD` zu `ULINT`. Seine klare Schnittstelle und einfache Funktionsweise machen ihn zu einer zuverlässigen Lösung in vielen Automatisierungsszenarien.

F_WORD_TO_USINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_WORD_TO_USINT` dient der Konvertierung eines `WORD`-Datentyps in einen `USINT`-Datentyp. Er ist Teil des Pakets `iec61131::conversion` und eignet sich für Anwendungen, bei denen eine Typumwandlung zwischen diesen beiden Datentypen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ` (Event): Startet die Ausführung des Funktionsblocks. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- `CNF` (Event): Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- `IN` (WORD): Der Eingangswert, der konvertiert werden soll.

Daten-Ausgänge

- `OUT` (USINT): Der konvertierte Ausgangswert.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von `WORD` zu `USINT` durch, sobald das Ereignis `REQ` empfangen wird. Die Konvertierung erfolgt mittels der Funktion `WORD_TO_USINT`, die den Eingangswert `IN` in den Ausgangswert `OUT` umwandelt. Nach Abschluss der Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher Funktionsblock (SimpleFB) und verwendet einen ST-Algorithmus für die Konvertierung.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Verzögerung.

Zustandsübersicht

1. **Idle-Zustand**: Der Funktionsblock wartet auf das Ereignis `REQ`.
2. **Ausführungszustand**: Bei Empfang von `REQ` wird die Konvertierung durchgeführt.
3. **Bestätigungszustand**: Nach erfolgreicher Konvertierung wird `CNF` ausgelöst und der Funktionsblock kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- Typumwandlung in Steuerungsanwendungen, wo ein `WORD`-Wert in einen `USINT`-Wert umgewandelt werden muss.
- Integration in größere Steuerungssysteme, die unterschiedliche Datentypen verarbeiten müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen wie `F_INT_TO_REAL` oder `F_BOOL_TO_STRING` ist `F_WORD_TO_USINT` spezialisiert auf die Umwandlung von `WORD` zu `USINT`.
- Ähnliche Bausteine könnten `F_USINT_TO_WORD` sein, welcher die umgekehrte Konvertierung durchführt.

Fazit

Der Funktionsblock `F_WORD_TO_USINT` bietet eine einfache und effiziente Möglichkeit, `WORD`-Werte in `USINT`-Werte umzuwandeln. Durch seine klare Schnittstellenstruktur und einfache Funktionsweise eignet er sich ideal für Anwendungen, die eine solche Typumwandlung erfordern.

WORD Konvertierungen

Hier finden Sie Funktionen zur Konvertierung des Datentyps WORD.

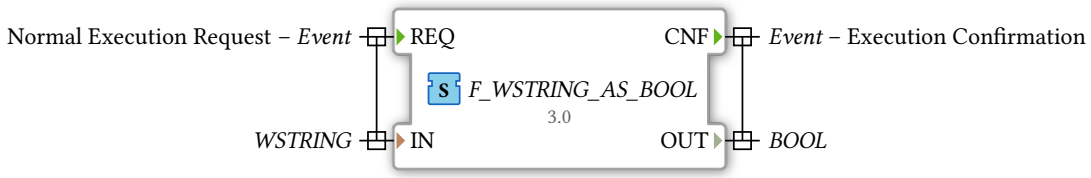
WSTRING

F_WSTRING_AS_BOOL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_WSTRING_AS_BOOL` dient der Konvertierung eines WSTRING-Wertes in einen BOOL-Wert. Dies ist besonders in Szenarien nützlich, in denen Zeichenketten in boolesche Werte umgewandelt werden müssen, beispielsweise bei der Verarbeitung von Benutzereingaben oder der Interpretation von Konfigurationsdaten.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Löst die Konvertierung aus.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Wird nach erfolgreicher Konvertierung ausgelöst.

Daten-Eingänge

- **IN**: WSTRING. Der Eingabewert, der in einen BOOL-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT**: BOOL. Der Ergebniswert der Konvertierung.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, wenn das Ereignis `REQ` empfangen wird. Die Konvertierung erfolgt mittels der Funktion `WSTRING_AS_BOOL`, die den WSTRING-Eingabewert `IN` in einen BOOL-Wert `OUT` umwandelt. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne zusätzliche Verzögerung.
- Der Funktionsblock ist als einfacher Funktionsblock (SimpleFB) implementiert und verwendet einen ST-Algorithmus für die Konvertierung.

Zustandsübersicht

1. **Idle**: Wartet auf das Ereignis `REQ`.
2. **Processing**: Führt die Konvertierung durch.
3. **Complete**: Löst das Ereignis `CNF` aus und kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- Verarbeitung von Benutzereingaben, die als Zeichenketten vorliegen.
- Interpretation von Konfigurationsdateien, die boolesche Werte als Zeichenketten speichern.
- Schnittstellen zwischen Systemen, die unterschiedliche Datentypen verwenden.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_STRING_AS_BOOL**: Konvertiert STRING statt WSTRING in BOOL.
- **F_BOOL_AS_WSTRING**: Führt die umgekehrte Konvertierung durch (BOOL zu WSTRING).

Fazit

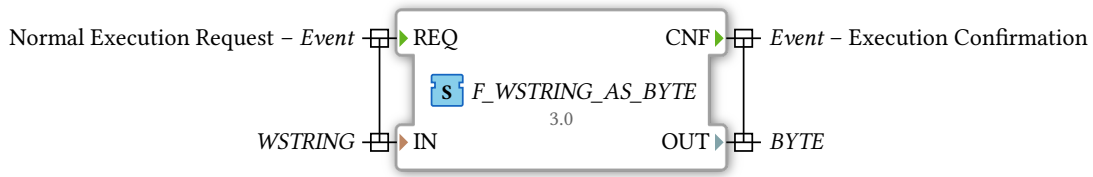
Der `F_WSTRING_AS_BOOL` Funktionsblock bietet eine einfache und effiziente Möglichkeit, WSTRING-Werte in BOOL-Werte umzuwandeln. Seine klare Schnittstellenstruktur und direkte Funktionsweise machen ihn zu einem nützlichen Werkzeug in verschiedenen Anwendungsszenarien.

F_WSTRING_AS_BYTE

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_WSTRING_AS_BYTE` dient der Konvertierung eines WSTRING-Datentyps in einen BYTE-Datentyp. Dieser Baustein ist besonders nützlich in Szenarien, wo eine Umwandlung von breiten Zeichenketten in einzelne Bytes erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge
- **REQ**: Startet die Ausführung des Funktionsblocks. Bei diesem Ereignis wird der Eingangswert `IN` verarbeitet.

Ereignis-Ausgänge
- **CNF**: Signalisiert den Abschluss der Verarbeitung und gibt das Ergebnis über den Ausgang `OUT` aus.

Daten-Eingänge
- **IN** (WSTRING): Der Eingangswert, der in einen BYTE-Wert konvertiert werden soll.

Daten-Ausgänge
- **OUT** (BYTE): Das Ergebnis der Konvertierung.

Adapter
- Keine Adapter vorhanden.

Funktionsweise
Der Funktionsblock führt bei Auftreten des `REQ`-Ereignisses die Konvertierung des WSTRING-Eingangs `IN` in einen BYTE-Wert durch. Das Ergebnis wird über den Ausgang `OUT` ausgegeben und das `CNF`-Ereignis signalisiert den erfolgreichen Abschluss der Operation.

Technische Besonderheiten
- Der Funktionsblock nutzt die Funktion `WSTRING_AS_BYTE` zur Konvertierung.
- Es handelt sich um einen einfachen Funktionsblock ohne interne Zustandsverwaltung.

Zustandsübersicht
Der Funktionsblock hat keine internen Zustände. Die Verarbeitung erfolgt direkt bei Auftreten des `REQ`-Ereignisses.

Anwendungsszenarien
- Konvertierung von WSTRING-Daten in BYTE-Werte für die weitere Verarbeitung in Systemen, die nur BYTE-Daten unterstützen.
- Verwendung in Kommunikationsprotokollen, wo Zeichenketten in Bytes umgewandelt werden müssen.

⚖️ Vergleich mit ähnlichen Bausteinen
- Im Vergleich zu anderen Konvertierungsbausteinen wie `F_STRING_AS_BYTE` ist dieser speziell für WSTRING-Datentypen ausgelegt.
- Ähnliche Bausteine könnten zusätzliche Parameter oder Zustände haben, dieser Funktionsblock ist jedoch auf die einfache Konvertierung beschränkt.

Fazit
Der `F_WSTRING_AS_BYTE` Funktionsblock bietet eine einfache und effiziente Möglichkeit, WSTRING-Daten in BYTE-Werte umzuwandeln. Seine klare Schnittstellenstruktur und direkte Funktionsweise machen ihn zu einer zuverlässigen Lösung für entsprechende Anwendungsfälle.

F_WSTRING_AS_DINT

• – ◦ • – ◦ • – ◦ •

Einleitung
Der Funktionsblock `F_WSTRING_AS_DINT` dient der Konvertierung eines WSTRING-Wertes in einen DINT-Wert. Er ist Teil des Pakets `iec61131::conversion` und ermöglicht die einfache Umwandlung von breiten Zeichenketten in ganzzahlige Werte.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Löst die Konvertierung aus.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Wird nach erfolgreicher Konvertierung ausgelöst.

Daten-Eingänge

- **IN**: Eingangsvariable vom Typ `WSTRING`. Enthält die zu konvertierende Zeichenkette.

Daten-Ausgänge

- **OUT**: Ausgangsvariable vom Typ `DINT`. Enthält das Ergebnis der Konvertierung.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock konvertiert den Wert der Eingangsvariable `IN` vom Typ `WSTRING` in einen Wert vom Typ `DINT` und speichert das Ergebnis in der Ausgangsvariable `OUT`. Die Konvertierung erfolgt durch den Aufruf der Funktion `WSTRING_AS_DINT` im Algorithmus `REQ`.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt über die eingebaute Funktion `WSTRING_AS_DINT`.
- Der Funktionsblock ist einfach und hat keine zusätzlichen Zustände oder komplexe Logik.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Ausführung erfolgt direkt bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Umwandlung von Benutzereingaben in numerische Werte.
- Verarbeitung von Zeichenketten, die numerische Daten enthalten, in Steuerungssystemen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke wie `F_STRING_AS_DINT` konvertieren einfache Zeichenketten (`STRING`) in `DINT`, während dieser Block breite Zeichenketten (`WSTRING`) verarbeitet.
- Im Vergleich zu generischen Konvertierungsblöcken ist dieser spezialisiert auf die Umwandlung von `WSTRING` zu `DINT`.

Fazit

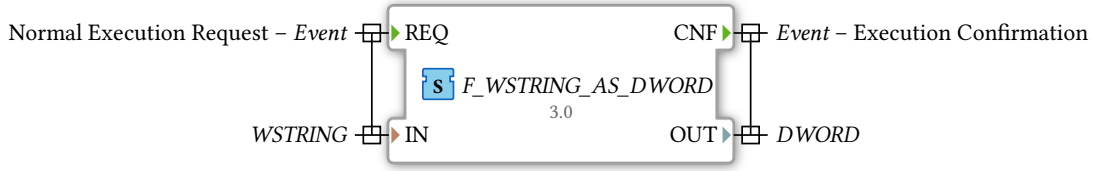
Der `F_WSTRING_AS_DINT` Funktionsblock bietet eine einfache und effiziente Möglichkeit, breite Zeichenketten in ganzzahlige Werte umzuwandeln. Er ist besonders nützlich in Szenarien, wo breite Zeichenketten verarbeitet werden müssen.

F_WSTRING_AS_DWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_WSTRING_AS_DWORD` dient zur Konvertierung eines WSTRING-Wertes in einen DWORD-Wert. Diese Funktionalität ist besonders nützlich, wenn Zeichenketten in numerische Werte umgewandelt werden müssen, beispielsweise für die weitere Verarbeitung in Steuerungsalgorithmen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Löst die Konvertierung aus.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Wird nach erfolgreicher Konvertierung ausgelöst.

Daten-Eingänge

- **IN**: Eingabe vom Typ WSTRING. Die Zeichenkette, die in einen DWORD-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Ausgabe vom Typ DWORD. Der konvertierte numerische Wert.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` empfangen wird. Die Konvertierung erfolgt mittels der Funktion `WSTRING_AS_DWORD`, die den WSTRING-Eingabewert in einen DWORD-Wert umwandelt. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst, und der konvertierte Wert steht am Ausgang `OUT` zur Verfügung.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher Funktionsblock (SimpleFB) und verfügt über einen einzigen Algorithmus, der bei Auslösung von `REQ` ausgeführt wird.
- Die Konvertierung erfolgt direkt und ohne zusätzliche Zustandsverwaltung.

Zustandsübersicht

Der Funktionsblock hat keinen internen Zustand. Die Konvertierung wird bei jedem `REQ`-Ereignis neu durchgeführt.

Anwendungsszenarien

- Konvertierung von Benutzereingaben (z.B. über HMI) in numerische Werte für die Steuerung.
- Verarbeitung von Zeichenketten, die numerische Informationen enthalten, in Steuerungsalgorithmen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen wie `F_STRING_AS_DWORD` ist dieser speziell für WSTRING-Eingaben ausgelegt.
- Ähnliche Bausteine könnten zusätzliche Parameter oder Fehlerbehandlung bieten, dieser Funktionsblock ist jedoch auf die grundlegende Konvertierung beschränkt.

Fazit

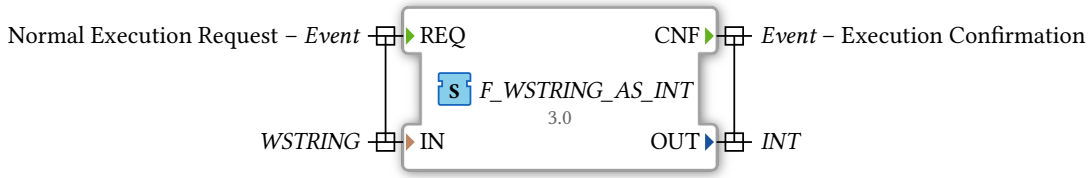
Der `F_WSTRING_AS_DWORD` Funktionsblock bietet eine einfache und effiziente Möglichkeit, WSTRING-Werte in DWORD-Werte umzuwandeln. Seine Einfachheit macht ihn besonders geeignet für Anwendungen, bei denen eine direkte Konvertierung ohne zusätzliche Komplexität erforderlich ist.

F_WSTRING_AS_INT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_WSTRING_AS_INT` dient zur Konvertierung eines WSTRING-Wertes in einen INT-Wert. Dieser Block ist besonders nützlich, wenn Zeichenketten in numerische Werte umgewandelt werden müssen, beispielsweise bei der Verarbeitung von Benutzereingaben oder der Interpretation von Textdaten.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Dieser Ereigniseingang löst die Ausführung des Funktionsblocks aus. Er ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Dieser Ereignisausgang signalisiert die erfolgreiche Ausführung des Funktionsblocks. Er ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Dieser Eingang erwartet einen WSTRING-Wert, der in einen INT-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Dieser Ausgang liefert den konvertierten INT-Wert.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` empfangen wird. Die Konvertierung erfolgt mithilfe der Funktion `WSTRING_AS_INT`, die den WSTRING-Wert von `IN` in einen INT-Wert umwandelt und das Ergebnis an `OUT` ausgibt. Anschließend wird das Ereignis `CNF` ausgelöst, um die erfolgreiche Ausführung zu signalisieren.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher Block ohne interne Zustände oder komplexe Logik.
- Die Konvertierung erfolgt direkt und ohne Verzögerung.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine Zustandsübergänge oder interne Zustände.

Anwendungsszenarien

- Verarbeitung von Benutzereingaben, die als Zeichenketten vorliegen, aber als numerische Werte interpretiert werden müssen.
- Interpretation von Textdaten aus externen Quellen, die in numerische Werte umgewandelt werden sollen.
- Integration in Systeme, die Zeichenketten und numerische Werte verarbeiten müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsblöcken wie `F_STRING_AS_INT` oder `F_WSTRING_AS_REAL` ist dieser Block speziell auf die Konvertierung von WSTRING zu INT ausgelegt.

- Andere Blöcke könnten zusätzliche Funktionen wie Fehlerbehandlung oder Formatierung bieten, dieser Block ist jedoch auf die grundlegende Konvertierung beschränkt.

Fazit

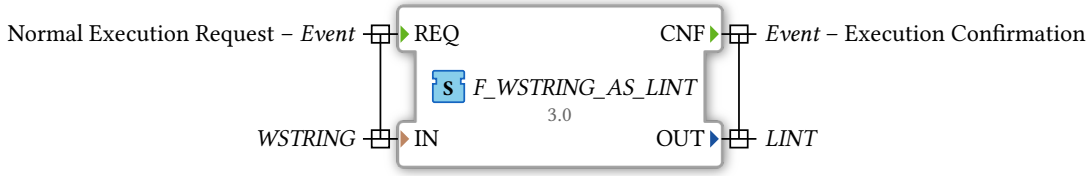
Der `F_WSTRING_AS_INT` Funktionsblock ist ein einfaches und effizientes Werkzeug zur Konvertierung von WSTRING-Werten in INT-Werte. Seine klare Schnittstelle und direkte Funktionsweise machen ihn ideal für Anwendungen, bei denen eine schnelle und unkomplizierte Konvertierung erforderlich ist.

F_WSTRING_AS_LINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_WSTRING_AS_LINT` dient der Konvertierung eines WSTRING-Wertes in einen LINT-Wert. Dieser Baustein ist besonders nützlich in Szenarien, wo Zeichenketten in numerische Werte umgewandelt werden müssen, beispielsweise bei der Verarbeitung von Benutzereingaben oder der Interpretation von Textdaten.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Dieser Ereigniseingang löst die Konvertierung aus. Er ist mit dem Dateneingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Dieses Ereignis wird nach erfolgreicher Konvertierung ausgegeben und ist mit dem Datenausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (WSTRING): Der Eingang für die Zeichenkette, die in einen LINT-Wert umgewandelt werden soll.

Daten-Ausgänge

- **OUT** (LINT): Der Ausgang für den konvertierten LINT-Wert.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock konvertiert den WSTRING-Wert am Eingang `IN` in einen LINT-Wert am Ausgang `OUT`, sobald das Ereignis `REQ` empfangen wird. Die Konvertierung erfolgt mithilfe der Funktion `WSTRING_AS_LINT`. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgegeben.

Technische Besonderheiten

- Der Funktionsblock ist einfach aufgebaut und enthält nur einen Algorithmus zur Konvertierung.
- Die Konvertierung kann fehlschlagen, wenn die Zeichenkette keine gültige numerische Repräsentation enthält. In diesem Fall wird das Ereignis `CNF` nicht ausgegeben.

Zustandsübersicht

1. **Idle**: Der Funktionsblock wartet auf das Ereignis `REQ`.
2. **Processing**: Bei Empfang von `REQ` wird die Konvertierung durchgeführt.
3. **Confirmation**: Nach erfolgreicher Konvertierung wird `CNF` ausgegeben, und der Funktionsblock kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- Verarbeitung von Benutzereingaben, die als Zeichenketten vorliegen, aber als numerische Werte interpretiert werden sollen.
- Interpretation von Textdaten aus externen Quellen, die numerische Informationen enthalten.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_STRING_AS_LINT**: Ähnlicher Funktionsblock, aber für den Datentyp STRING statt WSTRING.
- **F_WSTRING_AS_INT**: Konvertiert WSTRING in INT, bietet jedoch eine geringere Wertebereich als LINT.

Fazit

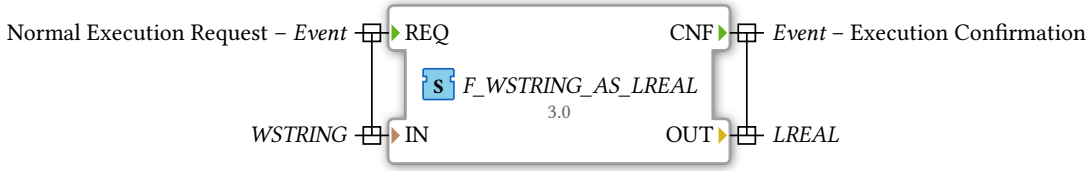
Der Funktionsblock `F_WSTRING_AS_LINT` ist ein einfaches und effektives Werkzeug zur Konvertierung von Zeichenketten in große numerische Werte. Seine klare Schnittstelle und einfache Funktionsweise machen ihn zu einer zuverlässigen Lösung für entsprechende Anwendungsfälle.

F_WSTRING_AS_LREAL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_WSTRING_AS_LREAL` dient der Konvertierung eines WSTRING-Wertes in einen LREAL-Wert. Dieser Baustein ist besonders nützlich, wenn Zeichenketten in numerische Werte umgewandelt werden müssen, beispielsweise für mathematische Berechnungen oder Datenverarbeitung.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung des Funktionsblocks. Wird dieser Eingang aktiviert, beginnt die Konvertierung des WSTRING-Wertes am Eingang `IN` in einen LREAL-Wert.

Ereignis-Ausgänge

- **CNF**: Signalisiert den erfolgreichen Abschluss der Konvertierung. Der konvertierte LREAL-Wert ist am Ausgang `OUT` verfügbar.

Daten-Eingänge

- **IN** (WSTRING): Der Eingang für die Zeichenkette, die in einen LREAL-Wert umgewandelt werden soll.

Daten-Ausgänge

- **OUT** (LREAL): Der Ausgang für den konvertierten LREAL-Wert.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Der Funktionsblock führt bei Aktivierung des `REQ`-Ereignisses die Konvertierung des WSTRING-Wertes am Eingang `IN` in einen LREAL-Wert durch. Die Konvertierung erfolgt mittels der Funktion `WSTRING_AS_LREAL`. Nach erfolgreicher Konvertierung wird das `CNF`-Ereignis ausgelöst und der konvertierte Wert am Ausgang `OUT` bereitgestellt.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne zusätzliche Parameter.
- Der Funktionsblock ist für die Verwendung in Echtzeitsystemen optimiert.

Zustandsübersicht

1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Processing**: Führt die Konvertierung durch.
3. **Complete**: Signalisiert den Abschluss mit dem `CNF`-Ereignis.

Anwendungsszenarien

- Konvertierung von Benutzereingaben in numerische Werte.
- Verarbeitung von Textdaten aus externen Quellen für Berechnungen.
- Integration in Systeme, die sowohl Zeichenketten als auch numerische Werte verarbeiten müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen ist `F_WSTRING_AS_LREAL` speziell auf die Umwandlung von WSTRING in LREAL ausgelegt.
- Andere Bausteine könnten zusätzliche Parameter oder Fehlerbehandlung bieten, dieser Baustein ist jedoch auf Einfachheit und Effizienz optimiert.

Fazit

Der `F_WSTRING_AS_LREAL` Funktionsblock bietet eine einfache und effiziente Möglichkeit, WSTRING-Werte in LREAL-Werte umzuwandeln. Seine klare Schnittstellenstruktur und direkte Funktionsweise machen ihn zu einem wertvollen Baustein in vielen Anwendungen.

F_WSTRING_AS_LWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_WSTRING_AS_LWORD` dient der Konvertierung eines WSTRING-Datentyps in einen LWORD-Datentyp. Diese Funktionalität ist insbesondere in Szenarien nützlich, in denen Zeichenketten in eine für die weitere Verarbeitung geeignete binäre Form umgewandelt werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Dieser Ereigniseingang löst die Konvertierung aus.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Dieses Ereignis wird nach erfolgreicher Konvertierung ausgegeben.

Daten-Eingänge

- **IN**: Eingabe vom Typ WSTRING. Dies ist die Zeichenkette, die konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Ausgabe vom Typ LWORD. Dies ist das Ergebnis der Konvertierung.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung von WSTRING zu LWORD durch, sobald das Ereignis `REQ` empfangen wird. Die Konvertierung erfolgt mittels der integrierten Funktion `WSTRING_AS_LWORD`. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgegeben, zusammen mit dem konvertierten Wert.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher FB (SimpleFB) und verwendet einen ST-Algorithmus für die Konvertierung.
- Die Konvertierung erfolgt direkt und ohne weitere Verzögerung.

Zustandsübersicht

1. **Idle**: Wartet auf das Ereignis `REQ`.
2. **Processing**: Führt die Konvertierung durch.
3. **Completed**: Gibt das Ereignis `CNF` aus und kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- Konvertierung von Zeichenketten in binäre Daten für die weitere Verarbeitung in Steuerungssystemen.
- Verwendung in Protokollumsetzungen, wo Zeichenketten in eine binäre Form gebracht werden müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen wie `F_STRING_AS_LWORD` arbeitet dieser FB spezifisch mit dem WSTRING-Datentyp, der Unicode-Zeichen unterstützt.

Fazit

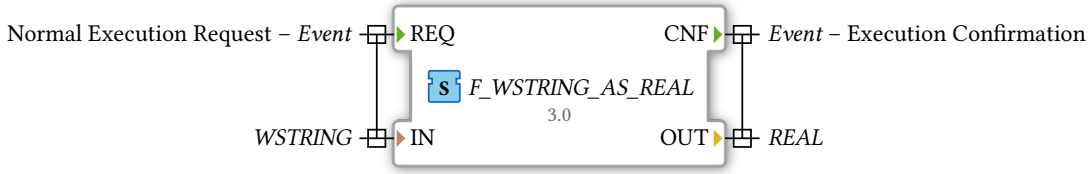
Der `F_WSTRING_AS_LWORD` Funktionsblock bietet eine einfache und effiziente Möglichkeit, WSTRING-Daten in LWORD-Daten umzuwandeln. Seine klare Schnittstellenstruktur und direkte Funktionsweise machen ihn zu einem nützlichen Werkzeug in verschiedenen Anwendungsfällen.

F_WSTRING_AS_REAL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_WSTRING_AS_REAL` dient der Konvertierung eines WSTRING-Wertes in einen REAL-Wert. Dieser Baustein ist besonders nützlich, wenn Zeichenketten, die numerische Werte repräsentieren, in numerische Datentypen umgewandelt werden müssen, um sie für weitere Berechnungen oder Steuerungslogik zu verwenden.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Eingang ausgelöst, wird der im Daten-Eingang `IN` enthaltene WSTRING-Wert in einen REAL-Wert umgewandelt.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ausgang wird aktiviert, sobald die Umwandlung abgeschlossen ist und der Ergebniswert im Daten-Ausgang `OUT` verfügbar ist.

Daten-Eingänge

- **IN**: Hier wird der WSTRING-Wert eingegeben, der in einen REAL-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT**: Enthält den konvertierten REAL-Wert nach erfolgreicher Ausführung des Funktionsblocks.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` ausgelöst wird. Die Umwandlung erfolgt mithilfe der internen Funktion `WSTRING_AS_REAL`, die den WSTRING-Eingang `IN` in einen REAL-Wert umwandelt und das Ergebnis im Ausgang `OUT` speichert. Anschließend wird das Ereignis `CNF` ausgelöst, um den erfolgreichen Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Der Funktionsblock unterstützt die Konvertierung von WSTRING nach REAL, was besonders in internationalen Anwendungen nützlich ist, da WSTRING Unicode-Zeichen unterstützt.
- Die Konvertierung erfolgt in einem einzigen Algorithmusschritt, was eine effiziente Ausführung gewährleistet.

Zustandsübersicht

1. **Idle-Zustand**: Der Funktionsblock wartet auf das Auslösen des `REQ`-Ereignisses.
2. **Konvertierungszustand**: Bei Auslösen von `REQ` wird die Konvertierung durchgeführt.
3. **Bestätigungszustand**: Nach erfolgreicher Konvertierung wird `CNF` ausgelöst und der Funktionsblock kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- **Datenverarbeitung**: Umwandlung von Benutzereingaben oder gelesenen Zeichenketten in numerische Werte für Berechnungen.
- **Schnittstellenkommunikation**: Konvertierung von Zeichenketten, die über Netzwerke oder Schnittstellen empfangen werden, in numerische Werte für die Steuerungslogik.
- **Internationale Anwendungen**: Verarbeitung von Unicode-kodierten Zeichenketten, die numerische Daten enthalten.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_STRING_AS_REAL**: Ähnlicher Funktionsblock, jedoch für den Datentyp STRING anstelle von WSTRING. `F_WSTRING_AS_REAL` ist vorzuziehen, wenn Unicode-Unterstützung erforderlich ist.
- **F_WSTRING_TO_***: Andere Konvertierungsbausteine für WSTRING, die jedoch in andere Datentypen wie INT oder BOOL umwandeln.

Fazit

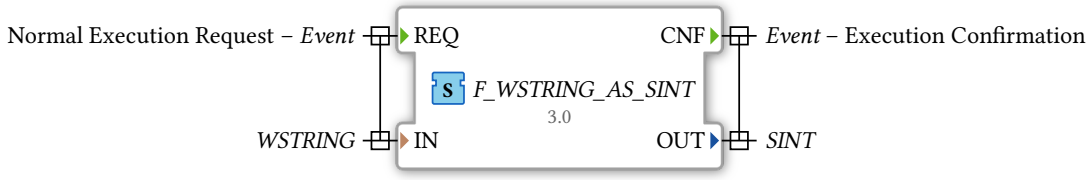
Der `F_WSTRING_AS_REAL`-Funktionsblock ist ein effizientes Werkzeug zur Konvertierung von Unicode-Zeichenketten in numerische REAL-Werte. Seine einfache Schnittstelle und zuverlässige Funktionsweise machen ihn ideal für Anwendungen, die internationale Zeichensätze oder komplexe Datenverarbeitung erfordern.

F_WSTRING_AS_SINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_WSTRING_AS_SINT` dient der Konvertierung eines WSTRING-Wertes in einen SINT-Wert (8-Bit Ganzzahl). Der Block ist besonders nützlich in Szenarien, wo Zeichenketten in numerische Werte umgewandelt werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Auslösendes Ereignis für die Konvertierung. Wird mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Bestätigt die erfolgreiche Ausführung der Konvertierung. Wird mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (WSTRING): Die Eingabezeichenkette, die in einen SINT-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT** (SINT): Der resultierende 8-Bit Ganzzahlwert nach der Konvertierung.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das `REQ`-Ereignis eintrifft. Dabei wird die Funktion `WSTRING_AS_SINT` auf den Eingabewert `IN` angewendet und das Ergebnis an `OUT` ausgegeben. Die erfolgreiche Ausführung wird durch das `CNF`-Ereignis signalisiert.

Technische Besonderheiten

- Die Konvertierung erfolgt mittels der eingebauten Funktion `WSTRING_AS_SINT`.
- Der Block ist für die Verwendung in IEC 61131-3-konformen Umgebungen optimiert.

Zustandsübersicht

Der Block hat keine internen Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- Konvertierung von Benutzereingaben in numerische Werte.
- Verarbeitung von Zeichenketten aus externen Quellen (z.B. Sensoren oder Netzwerkkommunikation).

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsblöcken ist `F_WSTRING_AS_SINT` spezialisiert auf die Umwandlung von WSTRING zu SINT.
- Andere Blöcke könnten zusätzliche Funktionen wie Fehlerbehandlung oder Typüberprüfung bieten.

Fazit

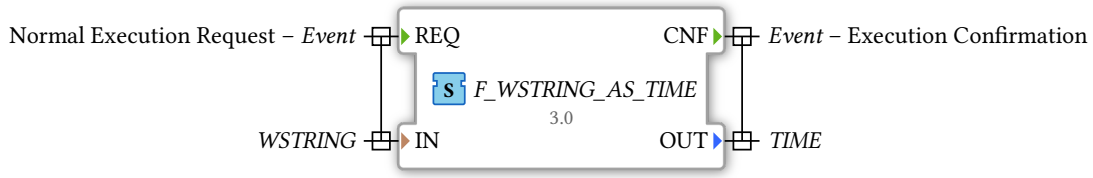
Der `F_WSTRING_AS_SINT`-Block bietet eine effiziente und direkte Methode zur Konvertierung von WSTRING-Werten in SINT-Werte. Seine Einfachheit und Spezialisierung machen ihn ideal für spezifische Anwendungsfälle.

F_WSTRING_AS_TIME

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_WSTRING_AS_TIME` dient zur Konvertierung einer Zeichenkette im WSTRING-Format in einen TIME-Wert. Dieser Baustein ist besonders nützlich, wenn Zeitwerte als Zeichenketten vorliegen und in das TIME-Format umgewandelt werden müssen, um sie in weiteren Berechnungen oder Steuerungslogiken verwenden zu können.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird dieser Ereigniseingang ausgelöst, wird der im Daten-Eingang `IN` enthaltene WSTRING-Wert in einen TIME-Wert umgewandelt.

Ereignis-Ausgänge

- **CNF**: Signalisiert die erfolgreiche Beendigung der Konvertierung. Dieses Ereignis wird zusammen mit dem konvertierten TIME-Wert am Daten-Ausgang `OUT` ausgegeben.

Daten-Eingänge

- **IN** (WSTRING): Die Zeichenkette, die in einen TIME-Wert umgewandelt werden soll.

Daten-Ausgänge

- **OUT** (TIME): Der konvertierte TIME-Wert.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Bei Auslösung des Ereigniseingangs `REQ` wird der im Daten-Eingang `IN` enthaltene WSTRING-Wert mithilfe der Funktion `WSTRING_AS_TIME` in einen TIME-Wert konvertiert. Das Ergebnis wird am Daten-Ausgang `OUT` ausgegeben und das Ereignis `CNF` signalisiert die erfolgreiche Beendigung der Konvertierung.

Technische Besonderheiten

- Der Funktionsblock ist einfach aufgebaut und enthält nur einen Algorithmus, der die Konvertierung durchführt.
- Es gibt keine zusätzlichen Zustände oder komplexe Logiken, da die Konvertierung direkt und ohne Zwischenschritte erfolgt.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine expliziten Zustände. Die Konvertierung erfolgt unmittelbar bei Auslösung des `REQ`-Ereignisses.

Anwendungsszenarien

- **Datenkonvertierung**: Wenn Zeitwerte als Zeichenketten vorliegen (z.B. aus einer Benutzereingabe oder einer externen Datenquelle) und in das TIME-Format umgewandelt werden müssen.
- **Schnittstellenkommunikation**: Bei der Kommunikation mit Systemen, die Zeitwerte als Zeichenketten übermitteln.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_STRING_AS_TIME**: Ähnlicher Funktionsblock, der jedoch mit STRING anstelle von WSTRING arbeitet. `F_WSTRING_AS_TIME` ist für Unicode-Zeichenketten geeignet.
- **F_TIME_AS_WSTRING**: Führt die umgekehrte Konvertierung durch (TIME zu WSTRING).

Fazit

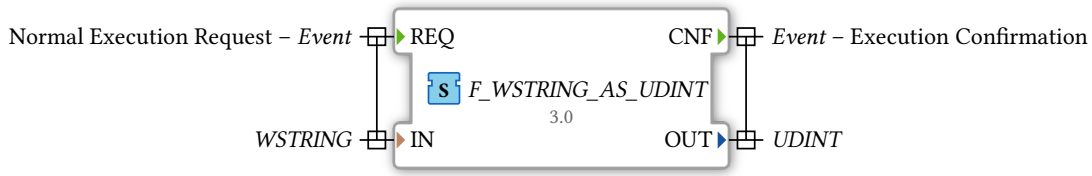
Der Funktionsblock `F_WSTRING_AS_TIME` ist ein einfaches und effektives Werkzeug zur Konvertierung von WSTRING-Zeichenketten in TIME-Werte. Seine klare Schnittstelle und direkte Funktionsweise machen ihn zu einer zuverlässigen Lösung für entsprechende Anwendungsfälle.

F_WSTRING_AS_UDINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_WSTRING_AS_UDINT` dient der Konvertierung eines WSTRING-Wertes in einen UDINT-Wert. Dieser Baustein ist besonders nützlich in Szenarien, wo Zeichenketten in numerische Werte umgewandelt werden müssen, beispielsweise bei der Verarbeitung von Benutzereingaben oder der Interpretation von Daten aus externen Quellen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Dieser Eingang löst die Konvertierung aus.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Dieser Ausgang signalisiert den Abschluss der Konvertierung.

Daten-Eingänge

- **IN**: WSTRING. Die Eingabezeichenkette, die in einen UDINT-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT**: UDINT. Der konvertierte numerische Wert.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, wenn das Ereignis `REQ` empfangen wird. Die Konvertierung erfolgt mittels der Funktion `WSTRING_AS_UDINT`, die den WSTRING-Eingang `IN` in einen UDINT-Wert umwandelt und das Ergebnis an `OUT` ausgibt. Nach Abschluss der Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Der Funktionsblock ist einfach und effizient, da er direkt die eingebaute Konvertierungsfunktion nutzt.
- Es gibt keine zusätzliche Fehlerbehandlung für ungültige Eingaben; die Konvertierung erfolgt gemäß den Regeln der IEC 61131-3.

Zustandsübersicht

Der Funktionsblock hat einen einfachen Zustandsautomaten:

1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Processing**: Führt die Konvertierung durch.
3. **Complete**: Sendet das `CNF`-Ereignis und kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- **Datenverarbeitung**: Konvertierung von Benutzereingaben oder externen Daten in numerische Werte.
- **Protokollierung**: Umwandlung von Zeichenketten in numerische IDs für die Speicherung oder Weiterverarbeitung.
- **Schnittstellenkommunikation**: Vorbereitung von Daten für die Übertragung über Protokolle, die numerische Werte erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_STRING_AS_UDINT**: Ähnlicher Funktionsblock, jedoch für STRING anstelle von WSTRING.
- **F_UDINT_AS_WSTRING**: Führt die umgekehrte Konvertierung durch (UDINT zu WSTRING).

Fazit

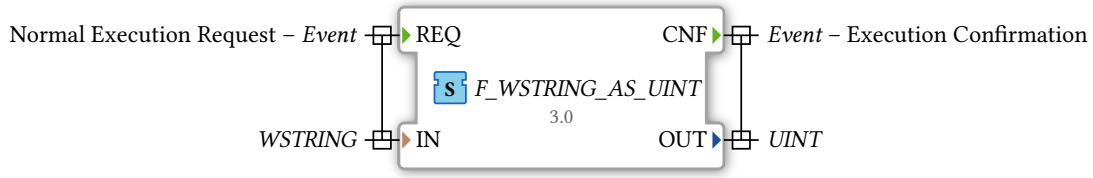
Der `F_WSTRING_AS_UDINT` Funktionsblock ist ein einfaches und effektives Werkzeug für die Konvertierung von WSTRING-Werten in UDINT. Seine klare Schnittstelle und direkte Funktionsweise machen ihn ideal für Anwendungen, die solche Konvertierungen erfordern.

F_WSTRING_AS_UINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_WSTRING_AS_UINT` dient der Konvertierung eines WSTRING-Wertes in einen UINT-Wert. Diese Funktion ist besonders nützlich, wenn Zeichenketten in numerische Werte umgewandelt werden müssen, beispielsweise für weitere Berechnungen oder Steuerungsaufgaben.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Dieser Eingang löst die Konvertierung aus.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Dieser Ausgang signalisiert den Abschluss der Konvertierung.

Daten-Eingänge

- **IN**: WSTRING-Wert, der in einen UINT-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT**: UINT-Wert, der aus der Konvertierung des WSTRING-Eingangs resultiert.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung des WSTRING-Eingangs `IN` in einen UINT-Wert `OUT` durch, sobald das Ereignis `REQ` ausgelöst wird. Die Konvertierung erfolgt mithilfe der integrierten Funktion `WSTRING_AS_UINT`. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst, um den Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Der Funktionsblock ist ein einfacher Funktionsblock (SimpleFB) und führt die Konvertierung in einem einzigen Algorithmus durch.
- Die Konvertierung ist direkt und erfolgt ohne zusätzliche Zustandsverwaltung.

Zustandsübersicht

Da es sich um einen einfachen Funktionsblock handelt, gibt es keine komplexen Zustandsübergänge. Der Block reagiert auf das `REQ`-Ereignis, führt die Konvertierung durch und signalisiert den Abschluss mit `CNF`.

Anwendungsszenarien

- Konvertierung von Benutzereingaben in numerische Werte.
- Verarbeitung von Zeichenketten, die numerische Daten enthalten, für Steuerungszwecke.
- Integration in größere Steuerungssysteme, die sowohl Zeichenketten als auch numerische Werte verarbeiten müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Konvertierungsbausteinen wie `F_STRING_AS_INT` oder `F_WSTRING_AS_INT` konvertiert dieser Block speziell in einen UINT-Wert, was für nicht-negative Zahlen geeignet ist.
- Die Verwendung von WSTRING ermöglicht die Verarbeitung von Unicode-Zeichenketten, was im Vergleich zu einfachen STRING-Konvertierungen flexibler ist.

Fazit

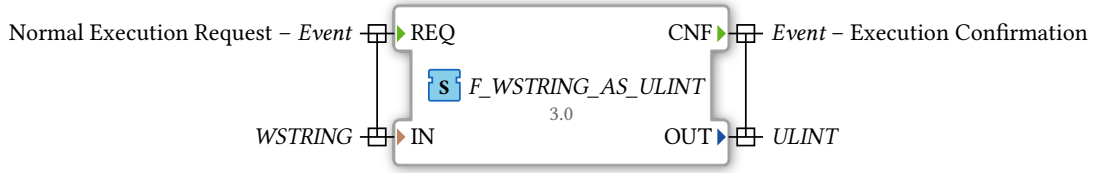
Der `F_WSTRING_AS_UINT`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, WSTRING-Werte in UINT-Werte zu konvertieren. Seine direkte Funktionsweise und klare Schnittstelle machen ihn zu einem nützlichen Baustein in Steuerungsanwendungen, die mit Zeichenketten und numerischen Werten arbeiten.

F_WSTRING_AS_ULINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_WSTRING_AS_ULINT` dient der Konvertierung eines WSTRING-Datentyps in einen ULINT-Datentyp. Dieser Baustein ist besonders nützlich in Szenarien, wo Zeichenketten in numerische Werte umgewandelt werden müssen, beispielsweise bei der Verarbeitung von Benutzereingaben oder beim Parsen von Daten aus externen Quellen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Dieser Ereigniseingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Dieser Ereignisausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (WSTRING): Die Eingabezeichenkette, die in einen ULINT-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT** (ULINT): Der konvertierte numerische Wert.

Adapter

Es sind keine Adapter für diesen Funktionsblock definiert.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, wenn das `REQ`-Ereignis empfangen wird. Die Konvertierung erfolgt mittels der Funktion `WSTRING_AS_ULINT`, die die Zeichenkette `IN` in einen vorzeichenlosen 64-Bit-Integer (`ULINT`) umwandelt. Nach erfolgreicher Konvertierung wird das `CNF`-Ereignis ausgelöst, und der konvertierte Wert steht am Ausgang `OUT` zur Verfügung.

Technische Besonderheiten

- Die Konvertierung erfolgt streng typisiert, d.h., die Eingabe muss eine gültige numerische Repräsentation in der Zeichenkette enthalten, um korrekte Ergebnisse zu liefern.
- Der Funktionsblock unterstützt keine Fehlerbehandlung für ungültige Eingaben. Es liegt in der Verantwortung des Anwenders, sicherzustellen, dass die Eingabe korrekt formatiert ist.

Zustandsübersicht

1. **Idle**: Der Funktionsblock wartet auf das `REQ`-Ereignis.
2. **Processing**: Bei Empfang von `REQ` wird die Konvertierung durchgeführt.
3. **Completed**: Nach der Konvertierung wird das `CNF`-Ereignis ausgelöst, und der Funktionsblock kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- Konvertierung von Benutzereingaben in numerische Werte.
- Parsen von Daten aus externen Quellen (z.B. Dateien, Netzwerk) in numerische Formate.
- Verwendung in Steuerungsanwendungen, wo Zeichenketten in Steuerungswerte umgewandelt werden müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsbausteinen ist `F_WSTRING_AS_ULINT` spezialisiert auf die Konvertierung von WSTRING zu ULINT und bietet daher eine optimierte und typsichere Lösung für diesen spezifischen Anwendungsfall.
- Andere Bausteine wie `F_STRING_AS_INT` oder `F_WSTRING_AS_LINT` bieten ähnliche Funktionalitäten, jedoch für andere Datentypen.

Fazit

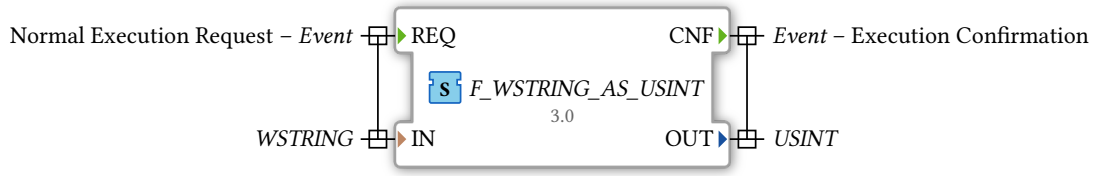
Der `F_WSTRING_AS_ULINT`-Funktionsblock ist ein effizientes Werkzeug für die Konvertierung von Zeichenketten in vorzeichenlose 64-Bit-Integer-Werte. Seine Einfachheit und Spezialisierung machen ihn ideal für Anwendungen, die eine schnelle und typsichere Konvertierung erfordern. Beachten Sie jedoch, dass keine integrierte Fehlerbehandlung für ungültige Eingaben vorhanden ist.

F_WSTRING_AS_USINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_WSTRING_AS_USINT` dient der Konvertierung eines WSTRING-Wertes in einen USINT-Wert. Er ist Teil des Pakets `iec61131::conversion` und ermöglicht die einfache Umwandlung von breiten Zeichenketten in vorzeichenlose 8-Bit-Ganzzahlen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung der Konvertierung. Der Eingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung. Der Ausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN** (WSTRING): Die Eingabezeichenkette, die in einen USINT-Wert konvertiert werden soll.

Daten-Ausgänge

- **OUT** (USINT): Der Ergebniswert der Konvertierung als vorzeichenlose 8-Bit-Ganzzahl.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, wenn das Ereignis `REQ` ausgelöst wird. Die Konvertierung erfolgt mittels der Funktion `WSTRING_AS_USINT`, die den WSTRING-Eingabewert in einen USINT-Wert umwandelt. Nach Abschluss der Konvertierung wird das Ereignis `CNF` ausgelöst, und der Ergebniswert steht am Ausgang `OUT` zur Verfügung.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne zusätzliche Parameter.
- Der Funktionsblock ist für die Verwendung in IEC 61499-konformen Umgebungen optimiert.

Zustandsübersicht

1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Processing**: Führt die Konvertierung durch.
3. **Done**: Sendet das `CNF`-Ereignis und stellt das Ergebnis bereit.

Anwendungsszenarien

- Konvertierung von Zeichenketten in numerische Werte für Steuerungsanwendungen.
- Verwendung in Systemen, die Zeichenketten als Eingabe verwenden, aber numerische Verarbeitung erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu generischen Konvertierungsbausteinen ist `F_WSTRING_AS_USINT` spezialisiert auf die Umwandlung von WSTRING zu USINT.
- Andere Bausteine könnten zusätzliche Parameter oder Fehlerbehandlung bieten, dieser Baustein ist jedoch auf Einfachheit und Effizienz ausgelegt.

Fazit

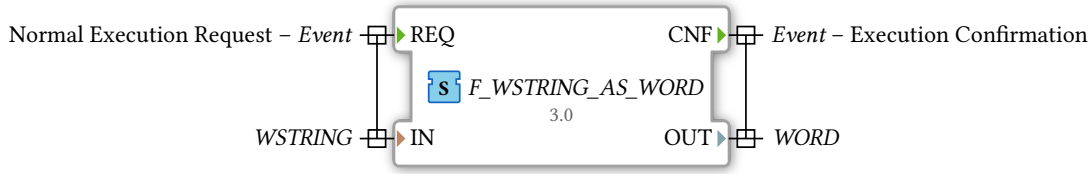
Der `F_WSTRING_AS_USINT`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, WSTRING-Werte in USINT-Werte umzuwandeln. Seine klare Schnittstelle und direkte Funktionsweise machen ihn ideal für Anwendungen, die schnelle und unkomplizierte Konvertierungen benötigen.

F_WSTRING_AS_WORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_WSTRING_AS_WORD` dient der Konvertierung eines WSTRING-Datentyps in einen WORD-Datentyp. Diese Funktionalität ist besonders nützlich, wenn Zeichenketten in numerische Werte umgewandelt werden müssen, beispielsweise für die weitere Verarbeitung in Steuerungsalgorithmen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung des Funktionsblocks. Dieser Ereigniseingang ist mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert die erfolgreiche Beendigung der Konvertierung. Dieser Ereignisausgang ist mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- **IN**: Erwartet eine Eingabe vom Typ `WSTRING`, die in einen `WORD`-Wert umgewandelt werden soll.

Daten-Ausgänge

- **OUT**: Gibt das Ergebnis der Konvertierung als `WORD` zurück.

Adapter

Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` empfangen wird. Die Umwandlung erfolgt mithilfe der Funktion `WSTRING_AS_WORD`, die den `WSTRING`-Eingang `IN` in einen `WORD`-Wert konvertiert und das Ergebnis über den Ausgang `OUT` ausgibt. Die erfolgreiche Ausführung wird durch das Ereignis `CNF` signalisiert.

Technische Besonderheiten

- Die Konvertierung erfolgt direkt und ohne zusätzliche Parameter.
- Der Funktionsblock ist für die Verwendung in IEC 61131-konformen Umgebungen optimiert.

Zustandsübersicht

Der Funktionsblock besitzt keine internen Zustände. Die Ausführung erfolgt unmittelbar bei Empfang des `REQ`-Ereignisses.

Anwendungsszenarien

- Umwandlung von Zeichenketten in numerische Werte für Steuerungsbefehle.
- Verarbeitung von Texteingaben in numerische Steuerungssysteme.
- Integration in Systeme, die eine Konvertierung zwischen unterschiedlichen Datentypen erfordern.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Konvertierungsbausteinen ist `F_WSTRING_AS_WORD` spezialisiert auf die Umwandlung von `WSTRING` zu `WORD`.
- Andere Bausteine könnten zusätzliche Parameter oder Fehlerbehandlungen bieten, dieser Funktionsblock ist jedoch auf Einfachheit und Effizienz ausgelegt.

Fazit

Der `F_WSTRING_AS_WORD`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, `WSTRING`-Daten in `WORD`-Werte umzuwandeln. Seine klare Schnittstelle und direkte Funktionsweise machen ihn ideal für Anwendungen, die schnelle und zuverlässige Konvertierungen erfordern.

F_WSTRING_TO_STRING

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_WSTRING_TO_STRING` dient der Konvertierung eines WSTRING-Datentyps in einen STRING-Datentyp. Dieser Baustein ist besonders nützlich in Szenarien, wo eine Umwandlung von breiten Zeichenketten (Unicode) in einfache Zeichenketten (ASCII oder ähnlich) erforderlich ist.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Bei Auslösung dieses Ereignisses wird der WSTRING-Eingang `IN` verarbeitet.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Konvertierung und gibt das Ergebnis über den STRING-Ausgang `OUT` aus.

Daten-Eingänge

- **IN** (WSTRING): Die Eingabezeichenkette im WSTRING-Format, die konvertiert werden soll.

Daten-Ausgänge

- **OUT** (STRING): Die konvertierte Zeichenkette im STRING-Format.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt die Konvertierung durch, sobald das Ereignis `REQ` ausgelöst wird. Die Umwandlung erfolgt mittels der integrierten Funktion `WSTRING_TO_STRING`, die den WSTRING-Eingang `IN` in einen STRING-Ausgang `OUT` umwandelt. Nach erfolgreicher Konvertierung wird das Ereignis `CNF` ausgelöst.

Technische Besonderheiten

- Der FB unterstützt keine zusätzlichen Parameter oder Konfigurationen für die Konvertierung.
- Die Konvertierung erfolgt direkt und ohne Zwischenspeicherung.

Zustandsübersicht

1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Processing**: Führt die Konvertierung durch.
3. **Complete**: Löst `CNF` aus und gibt das Ergebnis aus.

Anwendungsszenarien

- Konvertierung von Unicode-Texten in ASCII-Texten.
- Verwendung in Systemen, die keine breiten Zeichenketten unterstützen.
- Integration in bestehende Systeme, die nur mit STRING-Datentypen arbeiten.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu generischen Konvertierungsbausteinen ist `F_WSTRING_TO_STRING` spezialisiert auf die Umwandlung von WSTRING zu STRING.
- Ähnliche Bausteine könnten zusätzliche Optionen wie Zeichensatz-Konvertierung bieten, dieser FB ist jedoch auf die grundlegende Funktionalität beschränkt.

Fazit

Der `F_WSTRING_TO_STRING` Funktionsblock bietet eine einfache und effiziente Möglichkeit, WSTRING-Daten in STRING-Daten umzuwandeln. Seine klare Schnittstelle und direkte Funktionsweise machen ihn ideal für Projekte, die eine schnelle und unkomplizierte Konvertierung benötigen.

WSTRING Konvertierungen

Hier finden Sie Funktionen zur Konvertierung des Datentyps WSTRING in andere Datentypen.

2.4.1.8 Zähler

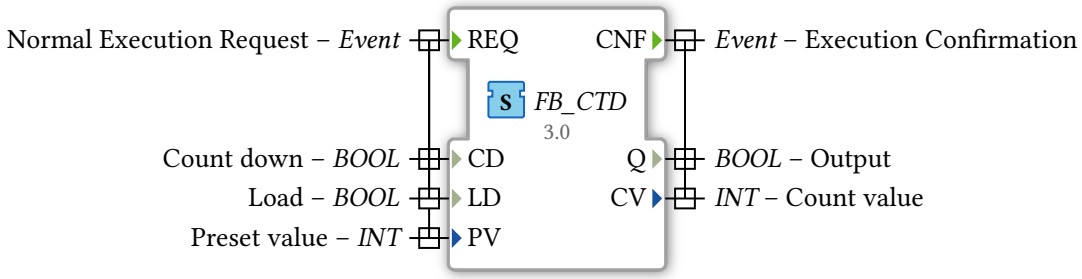
Hier finden Sie Funktionsbausteine für verschiedene Zählertypen (Vorwärts, Rückwärts, etc.).

FB_CTD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der FB_CTD (Down Counter) ist ein Funktionsblock, der als Abwärtszähler fungiert. Er zählt bei jedem Trigger-Ereignis den internen Zählerstand herunter und gibt ein Signal aus, wenn der Zählerstand einen bestimmten Wert erreicht oder unterschreitet.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsrequest, der den Zählvorgang startet.

Ereignis-Ausgänge

- **CNF**: Bestätigung der Ausführung, wird nach dem Zählvorgang ausgelöst.

Daten-Eingänge

- **CD (BOOL)**: Count Down - Signal zum Herunterzählen des Zählers.
- **LD (BOOL)**: Load - Signal zum Laden des Preset-Werts in den Zähler.
- **PV (INT)**: Preset Value - Voreingestellter Wert, der in den Zähler geladen wird.

Daten-Ausgänge

- **Q (BOOL)**: Output - Signal, das aktiviert wird, wenn der Zählerstand ≤ 0 ist.
- **CV (INT)**: Count Value - Aktueller Zählerstand.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der FB_CTD führt folgende Operationen aus:

1. Wenn das **LD**-Signal aktiv ist, wird der **PV**-Wert in den Zähler (**CV**) geladen.
2. Wenn das **CD**-Signal aktiv ist und der Zählerstand (**CV**) größer als −32768 ist, wird der Zähler um 1 verringert.
3. Das Ausgangssignal **Q** wird auf `TRUE` gesetzt, wenn der Zählerstand (**CV**) ≤ 0 ist.

Technische Besonderheiten

- Der Zählerstand (**CV**) ist ein 16-Bit-Integer mit einem Wertebereich von −32768 bis 32767.
- Der FB_CTD ist ein einfacher Funktionsblock ohne interne Zustandsmaschine.

Zustandsübersicht

Der FB_CTD hat keine expliziten Zustände, sondern reagiert direkt auf die Eingangssignale.

Anwendungsszenarien

- Steuerung von Abläufen, die eine bestimmte Anzahl von Schritten erfordern.
- Überwachung von Zeitintervallen oder Ereigniszählungen.
- Einsatz in Automatisierungssystemen, wo ein herunterzählender Zähler benötigt wird.

⚖️ Vergleich mit ähnlichen Bausteinen

- **FB_CTU (Up Counter)**: Zählt nach oben statt nach unten.
- **FB_CTUD (Up-Down Counter)**: Kombiniert Auf- und Abwärtszählen in einem Baustein.

Fazit

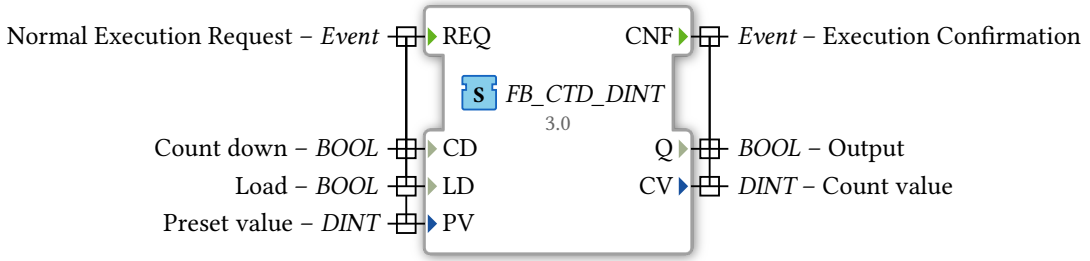
Der FB_CTD ist ein einfacher und effizienter Funktionsblock für Abwärtszählvorgänge in Automatisierungsanwendungen. Seine klare Schnittstelle und einfache Funktionsweise machen ihn ideal für grundlegende Zählaufgaben.

FB_CTD_DINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `FB_CTD_DINT` ist ein Abwärtszähler (Down Counter) für 32-Bit Ganzzahlen (DINT). Er wird verwendet, um einen Wert schrittweise zu verringern und bei Erreichen eines bestimmten Schwellwerts ein Signal auszugeben.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Normaler Ausführungsanforderung (Trigger für den Zählvorgang)

Ereignis-Ausgänge

- `CNF`: Ausführungsbestätigung (wird nach Abschluss des Zählvorgangs ausgelöst)

Daten-Eingänge

- `CD` (BOOL): Zählbefehl nach unten (Count Down)
- `LD` (BOOL): Ladebefehl (Load)
- `PV` (DINT): Voreingestellter Wert (Preset Value)

Daten-Ausgänge

- `Q` (BOOL): Ausgangssignal (wird TRUE, wenn CV ≤ 0)
- `CV` (DINT): Aktueller Zählwert (Count Value)

Funktionsweise

1. Bei Aktivierung des `REQ`-Ereignisses wird der Algorithmus ausgeführt:
   - Wenn `LD` TRUE ist, wird der Zähler auf den Wert von `PV` gesetzt
   - Wenn `CD` TRUE ist und der aktuelle Wert (`CV`) größer als –2147483648 ist, wird `CV` um 1 verringert
2. Das Ausgangssignal `Q` wird auf TRUE gesetzt, wenn `CV` kleiner oder gleich 0 ist
3. Nach Abschluss wird das `CNF`-Ereignis ausgelöst

Technische Besonderheiten

- Verwendet 32-Bit Ganzzahlen (DINT) mit einem Wertebereich von –2147483648 bis 2147483647
- Der Zähler stoppt bei Erreichen des minimalen DINT-Werts (-2147483648)
- Der Block ist in der IEC 61131-3 Standardbibliothek unter `iec61131::counters` enthalten

Zustandsübersicht

Der Funktionsblock hat keine expliziten Zustände, sondern arbeitet ereignisgesteuert:

1. Wartet auf `REQ`-Ereignis
2. Führt Zähloperation durch
3. Sendet `CNF`-Ereignis

Anwendungsszenarien

- Abwärtszählung in Produktionsprozessen
- Zeitsteuerungen mit Countdown-Funktion
- Überwachung von Restmengen oder verbleibenden Schritten

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu Aufwärtszählern (CTU) verringert dieser Block den Zählerwert
- Im Vergleich zu 16-Bit Zählern (INT) bietet dieser Block einen größeren Wertebereich
- Ähnlich wie Timer, aber für allgemeine Zählzwecke ohne Zeitbezug

Fazit

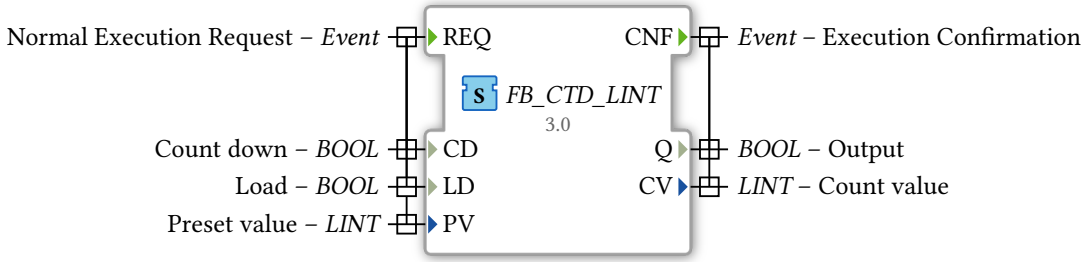
Der `FB_CTD_DINT` ist ein robuster und einfach zu verwendender Abwärtszähler für 32-Bit Werte. Seine klare Schnittstelle und deterministische Funktionsweise machen ihn ideal für Steuerungsaufgaben, bei denen eine zuverlässige Abwärtszählung benötigt wird.

FB_CTD_LINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `FB_CTD_LINT` ist ein Abwärtszähler (Counter Down) für 64-Bit-Ganzzahlen (LINT). Er ermöglicht das Herunterzählen eines internen Zählwerts und bietet Funktionen zum Laden eines vordefinierten Werts. Der Block ist besonders für Anwendungen geeignet, die mit großen Zahlenbereichen arbeiten müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Startet die normale Ausführung des Zählvorgangs. Verknüpft mit den Daten `CD`, `PV` und `LD`.

Ereignis-Ausgänge

- `CNF`: Bestätigt die Ausführung und liefert die aktuellen Werte von `Q` und `CV`.

Daten-Eingänge

- `CD` (BOOL): Signalisiert, ob der Zähler heruntergezählt werden soll.
- `LD` (BOOL): Lädt den vordefinierten Wert (`PV`) in den Zähler.
- `PV` (LINT): Der vordefinierte Wert, der geladen wird, wenn `LD` aktiv ist.

Daten-Ausgänge

- `Q` (BOOL): Signalisiert, ob der Zählerwert (`CV`) kleiner oder gleich Null ist.
- `CV` (LINT): Der aktuelle Zählerwert.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock arbeitet nach folgender Logik:

1. Wenn `LD` aktiv ist, wird der vordefinierte Wert (`PV`) in den Zähler (`CV`) geladen.
2. Wenn `CD` aktiv ist und der Zählerwert (`CV`) nicht den minimalen 64-Bit-Wert erreicht hat, wird `CV` um 1 verringert.
3. Der Ausgang `Q` wird auf `TRUE` gesetzt, wenn `CV` kleiner oder gleich Null ist.

Technische Besonderheiten

- Verwendet den Datentyp `LINT` (64-Bit-Ganzzahl), was einen sehr großen Zahlenbereich ermöglicht.
- Der minimale Wert für `LINT` wird berücksichtigt, um Überläufe zu vermeiden.

Zustandsübersicht

- **Initialisierung**: Keine spezifische Initialisierung, `CV` startet mit einem undefinierten Wert.
- **Laden**: Wenn `LD` aktiv ist, wird `CV` mit `PV` überschrieben.
- **Zählen**: Bei aktivem `CD` wird `CV` dekrementiert, sofern nicht der Minimalwert erreicht ist.
- **Ausgabe**: `Q` wird basierend auf dem aktuellen `CV`-Wert gesetzt.

Anwendungsszenarien

- Steuerung von Prozessen mit großen Zählbereichen, z.B. in der Produktionsautomatisierung.
- Überwachung von Zeitintervallen oder Ereigniszählungen in industriellen Steuerungen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnlich zu `FB_CTD_INT`, aber mit erweitertem Zahlenbereich (`LINT` statt `INT`).
- Bietet im Vergleich zu einfacheren Zählern mehr Flexibilität durch den großen Wertebereich und die Ladefunktion.

Fazit

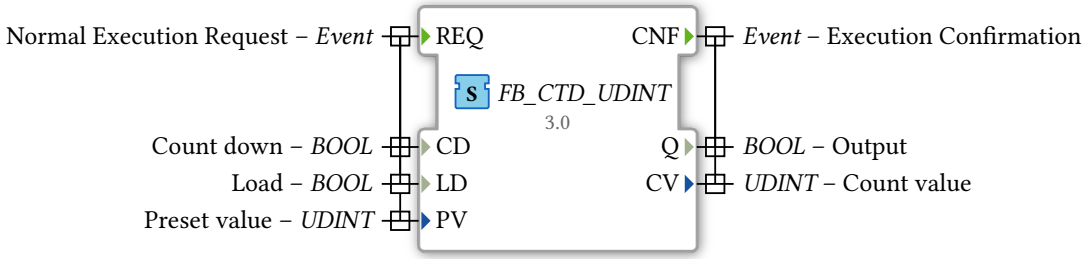
Der `FB_CTD_LINT` ist ein leistungsfähiger Abwärtszähler für Anwendungen, die große Zahlenbereiche erfordern. Seine einfache Handhabung und die robuste Implementierung machen ihn zu einer zuverlässigen Wahl für komplexe Steuerungsaufgaben.

FB_CTD_UDINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `FB_CTD_UDINT` ist ein Abwärtszähler (Down Counter) für vorzeichenlose 32-Bit-Ganzzahlen (UDINT). Er wird verwendet, um einen Zählerwert bei jedem Ereignis herunterzuzählen, bis er Null erreicht.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Wird verwendet, um den Zählvorgang oder das Laden eines neuen Wertes auszulösen.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Wird nach der Bearbeitung der Anforderung ausgelöst und gibt den aktuellen Zählerstand und den Ausgangszustand zurück.

Daten-Eingänge

- **CD** (BOOL): Zählt herunter, wenn `TRUE` und der Zählerwert größer als Null ist.
- **LD** (BOOL): Lädt den vorgegebenen Wert (`PV`) in den Zähler, wenn `TRUE`.
- **PV** (UDINT): Vorgegebener Wert, der in den Zähler geladen wird, wenn `LD` aktiviert ist.

Daten-Ausgänge

- **Q** (BOOL): Wird `TRUE`, wenn der Zählerwert kleiner oder gleich Null ist.
- **CV** (UDINT): Aktueller Zählerwert.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock verarbeitet die Eingabeereignisse und -daten wie folgt:

1. Wenn `LD` aktiviert ist (`TRUE`), wird der vorgegebene Wert (`PV`) in den Zähler (`CV`) geladen.
2. Wenn `CD` aktiviert ist (`TRUE`) und der Zählerwert (`CV`) größer als Null ist, wird der Zählerwert um eins verringert.
3. Der Ausgang `Q` wird auf `TRUE` gesetzt, wenn der Zählerwert (`CV`) kleiner oder gleich Null ist.
4. Nach der Bearbeitung wird das Ausgabeereignis `CNF` ausgelöst, um den neuen Zustand zu bestätigen.

Technische Besonderheiten

- Der Zähler arbeitet mit vorzeichenlosen 32-Bit-Ganzzahlen (UDINT).
- Der Zählerwert wird nur verringert, wenn `CD` aktiviert ist und der aktuelle Wert größer als Null ist.
- Der Ausgang `Q` signalisiert, wenn der Zählerwert Null oder darunter erreicht hat.

Zustandsübersicht

- **Initialisierung**: Der Zählerwert (`CV`) ist undefiniert, bis ein `LD`-Signal empfangen wird.
- **Laden**: Bei aktiviertem `LD` wird `PV` in `CV` geladen.
- **Herunterzählen**: Bei aktiviertem `CD` und `CV > 0` wird `CV` um eins verringert.
- **Ausgangszustand**: `Q` wird `TRUE`, wenn `CV <= 0`.

Anwendungsszenarien

- Steuerung von Prozessen, die eine bestimmte Anzahl von Schritten oder Ereignissen erfordern.
- Zeitgesteuerte Abläufe, bei denen ein Countdown benötigt wird.
- Überwachung von Zyklen oder Wiederholungen in automatisierten Systemen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu einem Aufwärtszähler (CTU) verringert dieser Baustein den Zählerwert.
- Ähnlich wie andere Zählerbausteine, aber spezialisiert auf vorzeichenlose 32-Bit-Ganzzahlen.

Fazit

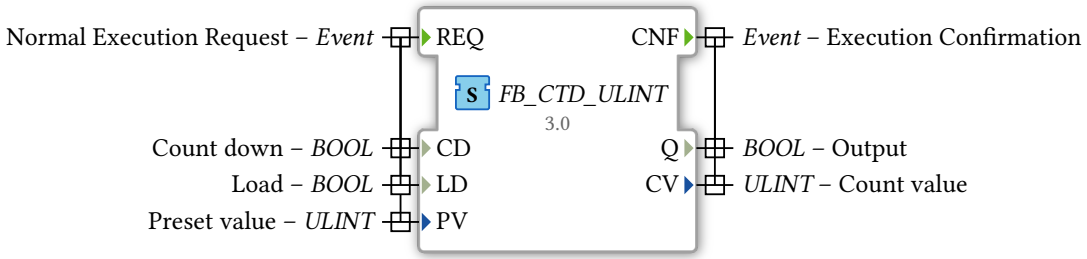
Der `FB_CTD_UDINT` ist ein effizienter und einfacher Abwärtszähler für Anwendungen, die eine Zählfunktion mit vorzeichenlosen 32-Bit-Ganzzahlen erfordern. Seine klare Schnittstelle und einfache Funktionsweise machen ihn zu einem nützlichen Baustein in der Automatisierungstechnik.

FB_CTD_ULINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der FB_CTD_ULINT ist ein Abwärtszähler (Down Counter) für den Datentyp ULINT (Unsigned Long Integer). Er wird verwendet, um Werte von einem vorgegebenen Startwert aus abwärts zu zählen und signalisiert, wenn der Zählerstand den Wert 0 erreicht oder unterschreitet.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Wird verwendet, um den Zähler zu aktualisieren. Verknüpft mit den Daten-Eingängen CD, PV und LD.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Wird ausgelöst, nachdem der Zähler aktualisiert wurde. Verknüpft mit den Daten-Ausgängen Q und CV.

Daten-Eingänge

- **CD (BOOL)**: Signalisiert, dass der Zähler um 1 verringert werden soll (Count Down).
- **LD (BOOL)**: Signalisiert, dass der Zähler mit dem Wert von PV geladen werden soll (Load).
- **PV (ULINT)**: Der voreingestellte Wert (Preset Value), mit dem der Zähler geladen wird, wenn LD aktiviert ist.

Daten-Ausgänge

- **Q (BOOL)**: Signalisiert, ob der Zählerstand kleiner oder gleich 0 ist.
- **CV (ULINT)**: Der aktuelle Zählerstand (Count Value).

Adapter

Es sind keine Adapter für diesen Funktionsblock definiert.

Funktionsweise

Der Funktionsblock arbeitet nach folgenden Regeln:

1. Wenn LD (Load) aktiviert ist, wird der Zählerstand (CV) auf den vorgegebenen Wert (PV) gesetzt.
2. Wenn CD (Count Down) aktiviert ist und der Zählerstand (CV) größer als 0 ist, wird der Zählerstand um 1 verringert.
3. Der Ausgang Q wird auf TRUE gesetzt, wenn der Zählerstand (CV) kleiner oder gleich 0 ist.

Technische Besonderheiten

- Der Zähler arbeitet mit dem Datentyp ULINT, was sehr große positive Zahlen ermöglicht (0 bis 18.446.744.073.709.551.615).
- Der Zähler wird nur aktualisiert, wenn das REQ-Ereignis ausgelöst wird.

Zustandsübersicht

- **Initialisierung**: Der Zählerstand (CV) ist undefiniert bis zum ersten REQ-Ereignis.
- **Laden (LD)**: CV wird auf PV gesetzt.
- **Zählen (CD)**: CV wird um 1 verringert, solange CV > 0.
- **Ausgang Q**: TRUE, wenn CV ≤ 0.

Anwendungsszenarien

- Steuerung von Produktionsprozessen mit Abwärtszählung (z.B. Verbleibende Teile in einem Lager).
- Zeitsteuerung mit großen Zeitintervallen.
- Überwachung von Ereignissen mit begrenzter Anzahl.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Zählern (z.B. FB_CTD_INT) bietet dieser Baustein eine größere Zahlenreichweite durch ULINT.
- Ähnlich wie FB_CTD, aber speziell für ULINT-Werte optimiert.

Fazit

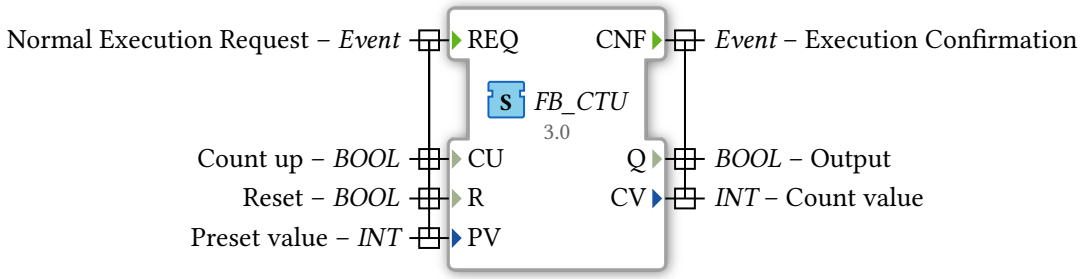
Der FB_CTD_ULINT ist ein leistungsfähiger Abwärtszähler für Anwendungen, die sehr große Zählbereiche erfordern. Seine einfache und klare Schnittstelle macht ihn leicht integrierbar in verschiedene Steuerungssysteme.

FB_CTU

• – ◦ • – ◦ • – ◦ •

Einleitung

Der FB_CTU (Up Counter) ist ein Funktionsblock, der als Aufwärtszähler fungiert. Er zählt Ereignisse hoch und kann zurückgesetzt werden. Der Zählerwert wird mit einem vordefinierten Wert (PV) verglichen, und ein Ausgangssignal (Q) wird aktiviert, sobald der Zählerwert den vordefinierten Wert erreicht oder überschreitet.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Wird verwendet, um den Zählvorgang auszulösen. Verknüpft mit den Daten-Eingängen `CU` und `R`.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Wird ausgelöst, nachdem der Zählvorgang abgeschlossen ist. Verknüpft mit den Daten-Ausgängen `Q` und `CV`.

Daten-Eingänge

- **CU** (BOOL): Zählt hoch, wenn `TRUE`. Wird nur ausgewertet, wenn `R` `FALSE` ist.
- **R** (BOOL): Setzt den Zählerwert (`CV`) auf `0` zurück, wenn `TRUE`.
- **PV** (INT): Vordefinierter Wert, mit dem der Zählerwert (`CV`) verglichen wird.

Daten-Ausgänge

- **Q** (BOOL): Wird `TRUE`, wenn der Zählerwert (`CV`) den vordefinierten Wert (`PV`) erreicht oder überschreitet.
- **CV** (INT): Aktueller Zählerwert.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der FB_CTU führt folgende Schritte aus, wenn das Ereignis `REQ` eintritt:

1. Wenn `R` `TRUE` ist, wird der Zählerwert (`CV`) auf `0` zurückgesetzt.
2. Wenn `CU` `TRUE` ist und der Zählerwert (`CV`) kleiner als 32767 ist, wird `CV` um 1 erhöht.
3. Der Ausgang `Q` wird auf `TRUE` gesetzt, wenn `CV` größer oder gleich `PV` ist.

Technische Besonderheiten

- Der Zählerwert (`CV`) ist auf einen maximalen Wert von 32767 begrenzt.
- Der FB_CTU ist ein einfacher Funktionsblock ohne interne Zustandsverwaltung zwischen den Aufrufen.

Zustandsübersicht

Der FB_CTU hat keine internen Zustände. Die Logik wird bei jedem `REQ`-Ereignis neu ausgeführt.

Anwendungsszenarien

- Zählen von Ereignissen, z.B. Produktionsschritte in einer Fertigungslinie.
- Überwachung von Prozessen, bei denen eine bestimmte Anzahl von Schritten erreicht werden muss.
- Steuerung von Abläufen, die nach einer bestimmten Anzahl von Wiederholungen eine Aktion auslösen sollen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **FB_CTD**: Ein Abwärtszähler, der im Gegensatz zum FB_CTU herunterzählt.
- **FB_CTUD**: Ein kombiniertes Auf- und Abwärtszähler, der sowohl hoch- als auch herunterzählen kann.

Fazit

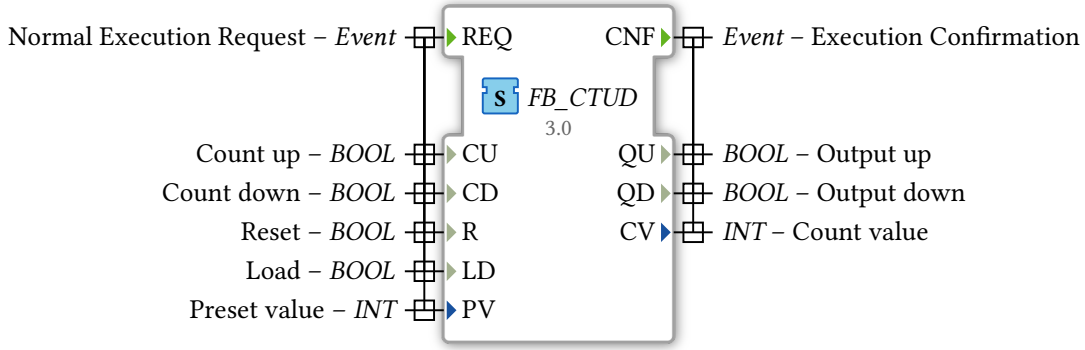
Der FB_CTU ist ein einfacher und effektiver Funktionsblock für Aufwärtszählvorgänge. Er eignet sich besonders für Anwendungen, bei denen Ereignisse gezählt und bei Erreichen eines bestimmten Werts eine Aktion ausgelöst werden soll. Die klare Schnittstellenstruktur und die einfache Funktionsweise machen ihn zu einem nützlichen Baustein in der Steuerungstechnik.

FB_CTUD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der FB_CTUD (Up Down Counter) ist ein Funktionsblock, der als Zähler mit Aufwärts- und Abwärtszählfunktion dient. Er ermöglicht das Zählen von Ereignissen in beide Richtungen und bietet zusätzliche Funktionen wie Reset und Laden eines vordefinierten Wertes.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Dieser Ereigniseingang löst die Ausführung des Zählvorgangs aus und ist mit den Daten-Eingängen CU, CD, R, LD und PV verknüpft.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Dieses Ereignis wird nach Abschluss des Zählvorgangs ausgegeben und ist mit den Daten-Ausgängen QU, QD und CV verknüpft.

Daten-Eingänge

- **CU** (BOOL): Zählt nach oben (Count Up).
- **CD** (BOOL): Zählt nach unten (Count Down).
- **R** (BOOL): Setzt den Zählerwert auf 0 zurück (Reset).
- **LD** (BOOL): Lädt den vordefinierten Wert PV in den Zähler (Load).
- **PV** (INT): Vordefinierter Wert, der bei aktiviertem LD in den Zähler geladen wird (Preset Value).

Daten-Ausgänge

- **QU** (BOOL): Signalisiert, ob der Zählerwert größer oder gleich dem vordefinierten Wert PV ist (Output Up).
- **QD** (BOOL): Signalisiert, ob der Zählerwert kleiner oder gleich 0 ist (Output Down).
- **CV** (INT): Aktueller Zählerwert (Count Value).

Adapter

Der FB_CTUD verfügt über keine Adapter.

Funktionsweise

Der Funktionsblock FB_CTUD führt folgende Operationen aus:

1. **Reset (R)**: Wenn R aktiviert ist, wird der Zählerwert (CV) auf 0 gesetzt.
2. **Laden (LD)**: Wenn LD aktiviert ist, wird der vordefinierte Wert (PV) in den Zähler (CV) geladen.
3. **Zählen (CU/CD)**:
   - Wenn CU aktiviert ist und der Zählerwert (CV) kleiner als 32767 ist, wird CV um 1 erhöht.

- Wenn CD aktiviert ist und der Zählerwert (CV) größer als −32768 ist, wird CV um 1 verringert.
- CU und CD dürfen nicht gleichzeitig aktiviert sein.

4. **Ausgänge (QU/QD)**:
   - QU wird aktiviert, wenn CV größer oder gleich PV ist.
   - QD wird aktiviert, wenn CV kleiner oder gleich 0 ist.

Technische Besonderheiten

- Der Zählerwert (CV) ist vom Typ INT und hat einen Wertebereich von −32768 bis 32767.
- Der FB_CTUD ist ein einfacher Funktionsblock (SimpleFB) und verwendet einen ST-Algorithmus (Structured Text) zur Implementierung der Logik.

Zustandsübersicht

Der FB_CTUD hat keine expliziten Zustände, aber der Zählerwert (CV) kann als Zustand betrachtet werden, der durch die Eingänge CU, CD, R und LD verändert wird.

Anwendungsszenarien

- **Produktionslinien**: Zählen von produzierten Einheiten (Aufwärtszählen) oder verbleibenden Einheiten (Abwärtszählen).
- **Lagerverwaltung**: Überwachung des Lagerbestands.
- **Steuerungstechnik**: Zeitgesteuerte Prozesse oder Ereigniszählung.

⚖️ Vergleich mit ähnlichen Bausteinen

- **FB_CTU**: Einfacher Aufwärtszähler ohne Abwärtszählfunktion.
- **FB_CTD**: Einfacher Abwärtszähler ohne Aufwärtszählfunktion.
- **FB_CTUD**: Kombiniert beide Funktionen in einem Baustein und bietet zusätzlich Reset- und Lade-Funktionen.

Fazit

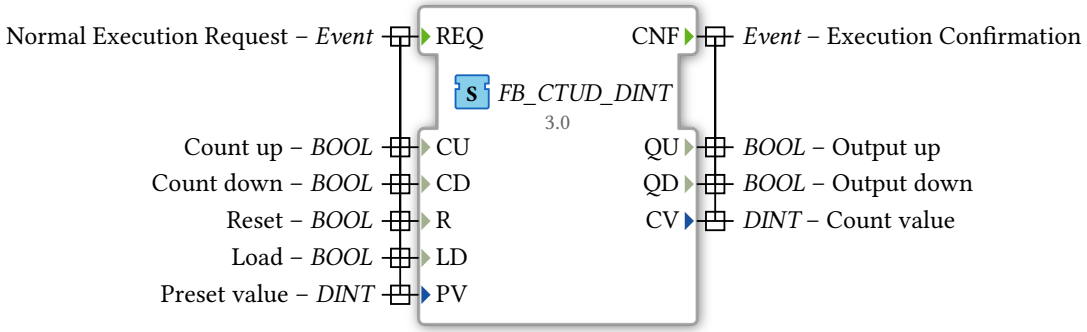
Der FB_CTUD ist ein vielseitiger Funktionsblock, der sowohl Aufwärts- als auch Abwärtszählfunktionen in einem Baustein vereint. Durch die zusätzlichen Funktionen wie Reset und Laden eines vordefinierten Wertes eignet er sich für eine Vielzahl von Anwendungen in der Steuerungstechnik. Seine einfache Handhabung und klare Schnittstellenstruktur machen ihn zu einer praktischen Lösung für Zählaufgaben.

FB_CTUD_DINT

• − ◦ • − ◦ • − ◦ •

Einleitung

Der FB_CTUD_DINT ist ein Funktionsblock, der als Aufwärts-/Abwärtszähler (Up Down Counter) mit DINT-Zählwert fungiert. Er ermöglicht das Zählen in beide Richtungen sowie das Zurücksetzen und Laden eines vordefinierten Wertes. Der Zähler ist besonders für Steuerungsanwendungen geeignet, bei denen eine flexible Zählfunktion benötigt wird.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung (triggert die Zähloperation)

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung (wird nach Abschluss der Zähloperation ausgelöst)

Daten-Eingänge

- **CU** (BOOL): Zählt hoch (Count up)
- **CD** (BOOL): Zählt runter (Count down)
- **R** (BOOL): Setzt den Zähler zurück (Reset)
- **LD** (BOOL): Lädt den Preset-Wert (Load)
- **PV** (DINT): Preset-Wert (Preset value)

Daten-Ausgänge

- **QU** (BOOL): Signalisiert, ob der Zählwert den Preset-Wert erreicht oder überschritten hat (Output up)
- **QD** (BOOL): Signalisiert, ob der Zählwert 0 erreicht oder unterschritten hat (Output down)
- **CV** (DINT): Aktueller Zählwert (Count value)

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt folgende Operationen aus, wenn das REQ-Ereignis eintritt:

1. **Reset (R)**: Setzt den Zählwert (CV) auf 0, wenn R aktiv ist.
2. **Load (LD)**: Lädt den Preset-Wert (PV) in den Zählwert (CV), wenn LD aktiv ist.
3. **Zählen (CU/CD)**:
   - Wenn CU aktiv ist und der Zählwert nicht den Maximalwert (2147483647) erreicht hat, wird der Zählwert um 1 erhöht.
   - Wenn CD aktiv ist und der Zählwert nicht den Minimalwert (-2147483648) erreicht hat, wird der Zählwert um 1 verringert.
4. **Ausgänge setzen**:
   - QU wird auf TRUE gesetzt, wenn der Zählwert den Preset-Wert erreicht oder überschreitet.
   - QD wird auf TRUE gesetzt, wenn der Zählwert 0 erreicht oder unterschreitet.

Technische Besonderheiten

- Der Zählwert (CV) ist vom Typ DINT (32-Bit Ganzzahl mit Vorzeichen), was einen großen Zählbereich ermöglicht.
- Der Funktionsblock verhindert ein Überlaufen des Zählwerts durch Prüfung der Grenzwerte.
- Gleichzeitiges Aktivieren von CU und CD führt zu keiner Änderung des Zählwerts.

Zustandsübersicht

Der Funktionsblock hat keinen internen Zustand im Sinne eines Zustandsautomaten. Die Logik wird bei jedem REQ-Ereignis neu ausgeführt.

Anwendungsszenarien

- Produktionszählung (z.B. Stückzählung auf einem Fließband)
- Positionserfassung (z.B. in linearen Achsen)
- Ereigniszählung mit Möglichkeit zur Richtungsumkehr

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu einfachen Zählern (z.B. CTU oder CTD) bietet dieser Baustein beide Zählrichtungen in einem Block.
- Gegenüber Zählern mit kleinerem Datentyp (z.B. INT) bietet der DINT-Typ einen größeren Zählbereich.

Fazit

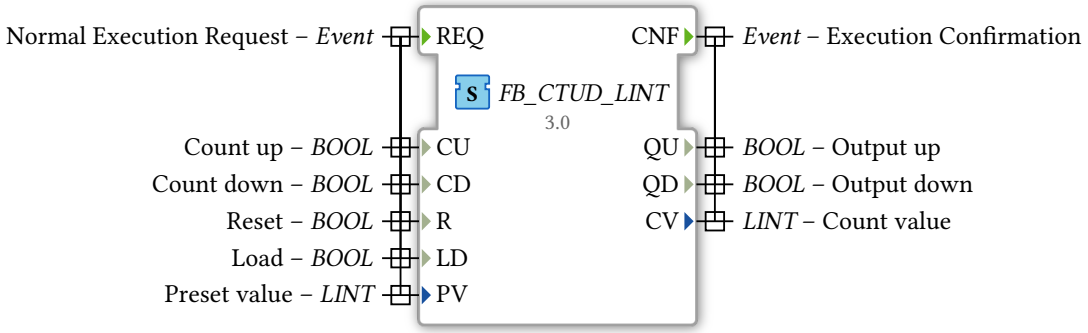
Der FB_CTUD_DINT ist ein vielseitiger Zählbaustein für Anwendungen, die bidirektionale Zählfunktionen mit großem Zählbereich erfordern. Durch die integrierten Reset- und Load-Funktionen sowie die Statusausgänge QU und QD eignet er sich besonders für komplexe Steuerungsaufgaben.

FB_CTUD_LINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der FB_CTUD_LINT ist ein Funktionsblock für einen hoch- und runterzählenden Zähler mit 64-Bit-Ganzzahlwerten (LINT). Er bietet Funktionen zum Zählen nach oben und unten, zum Zurücksetzen des Zählers sowie zum Laden eines vordefinierten Wertes. Der Zähler ist besonders für Anwendungen geeignet, die große Zahlenbereiche erfordern.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Wird verwendet, um den Zählvorgang zu starten. Verknüpft mit den Daten-Eingängen CU, CD, R, LD und PV.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Wird nach Abschluss der Zähloperation ausgegeben. Verknüpft mit den Daten-Ausgängen QU, QD und CV.

Daten-Eingänge

- **CU (BOOL)**: Signal zum Hochzählen (Count Up).
- **CD (BOOL)**: Signal zum Runterzählen (Count Down).
- **R (BOOL)**: Signal zum Zurücksetzen des Zählers (Reset).
- **LD (BOOL)**: Signal zum Laden des vordefinierten Wertes (Load).
- **PV (LINT)**: Vordefinierter Wert (Preset Value), der geladen wird, wenn LD aktiv ist.

Daten-Ausgänge

- **QU (BOOL)**: Signal, das anzeigt, ob der Zählerstand größer oder gleich dem vordefinierten Wert ist.
- **QD (BOOL)**: Signal, das anzeigt, ob der Zählerstand kleiner oder gleich 0 ist.
- **CV (LINT)**: Aktueller Zählerstand (Count Value).

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt folgende Operationen aus, wenn das REQ-Ereignis eintritt:

1. **Reset (R)**: Setzt den Zählerstand (CV) auf 0 zurück.
2. **Load (LD)**: Lädt den vordefinierten Wert (PV) in den Zählerstand (CV).
3. **Count Up (CU)**: Erhöht den Zählerstand um 1, solange der maximale Wert (9223372036854775807) nicht überschritten wird.
4. **Count Down (CD)**: Verringert den Zählerstand um 1, solange der minimale Wert (-9223372036854775808) nicht unterschritten wird.
5. **Ausgänge QU und QD**: Werden basierend auf dem aktuellen Zählerstand aktualisiert.

Technische Besonderheiten

- Verwendet 64-Bit-Ganzzahlen (LINT) für den Zählerstand und den vordefinierten Wert.
- Unterstützt sowohl Hoch- als auch Runterzählen mit Überlauf- und Unterlaufschutz.
- Keine gleichzeitige Aktivierung von CU und CD möglich (verhindert Konflikte).

Zustandsübersicht

Der Funktionsblock hat keinen internen Zustand außer dem aktuellen Zählerstand (CV). Die Logik wird bei jedem REQ-Ereignis neu ausgeführt.

Anwendungsszenarien

- Industrielle Steuerungen mit großen Zählbereichen.
- Überwachung von Produktionsprozessen.
- Zeitmessung oder Ereigniszählung in langen Zeiträumen.

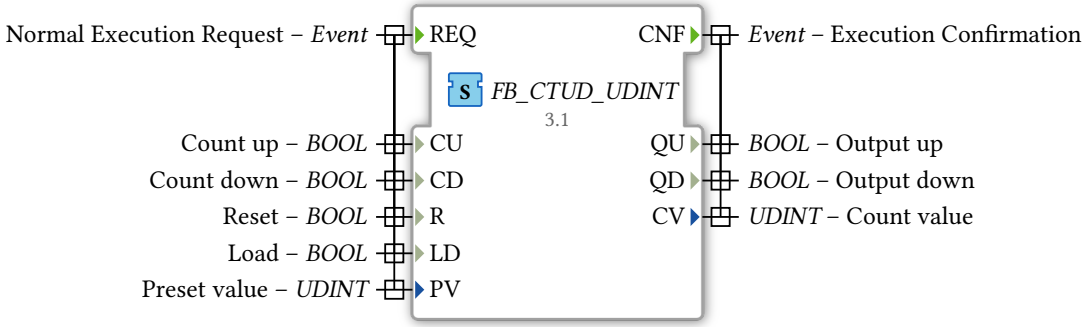
⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu einfachen Zählern bietet FB_CTUD_LINT eine größere Zahlenreichweite (64-Bit).
- Ähnlich zu FB_CTUD, aber mit LINT-Datentyp für erweiterte Anwendungen.

Fazit

Der FB_CTUD_LINT ist ein leistungsstarker Zähler für Anwendungen, die große Zahlenbereiche erfordern. Seine Flexibilität und Robustheit machen ihn ideal für industrielle Steuerungen und komplexe Zählaufgaben.

FB_CTUD_UDINT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `FB_CTUD_UDINT` ist ein kombinierter Auf-/Abwärtszähler (Up/Down Counter) für den Datentyp `UDINT` (32-Bit-Ganzzahl, vorzeichenlos). Er zählt bei `CU` hoch und bei `CD` herunter, kann über `LD` auf einen Vorgabewert `PV` geladen und über `R` auf `0` zurückgesetzt werden.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung, verknüpft mit `CU`, `CD`, `R`, `LD`, `PV`.

Ereignis-Ausgänge

- **CNF**: Bestätigt die Ausführung, verknüpft mit `QU`, `QD`, `CV`.

Daten-Eingänge

- **CU** (BOOL): Aufwärtszählen bei `TRUE`.
- **CD** (BOOL): Abwärtszählen bei `TRUE`.
- **R** (BOOL): Reset — setzt `CV` auf `0`, hat Vorrang vor `LD`/`CU`/`CD`.
- **LD** (BOOL): Load — lädt `PV` in `CV`, sofern `R` nicht aktiv ist.
- **PV** (UDINT): Vorgabewert (Preset Value), gegen den `CV` verglichen wird.

Daten-Ausgänge

- **QU** (BOOL): `TRUE`, wenn `CV >= PV`.
- **QD** (BOOL): `TRUE`, wenn `CV <= 0`.
- **CV** (UDINT): Aktueller Zählerstand.

Funktionsweise

Bei jedem REQ-Ereignis wertet der Baustein die Eingänge in folgender Priorität aus: Ist R aktiv, wird CV := 0. Andernfalls wird bei aktivem LD CV := PV gesetzt. Ist weder R noch LD aktiv und sind CU/CD nicht gleichzeitig aktiv, wird CV bei CU (sofern CV < 4294967295) um 1 erhöht bzw. bei CD (sofern CV > 0) um 1 verringert. Abschließend werden QU := CV >= PV und QD := CV <= 0 berechnet.

Technische Besonderheiten

- **Priorisierung**: R (Reset) hat Vorrang vor LD (Load), das wiederum Vorrang vor der eigentlichen Zähloperation hat.
- **Gleichzeitiges CU+CD**: Sind beide gleichzeitig TRUE, findet keine Zähländerung statt (NOT(CU AND CD)-Bedingung in der Algorithmus-ST).
- **Wertebereichsgrenze**: Der Zählerstand wird bei 4294967295 (Maximalwert von UDINT) nach oben und bei 0 nach unten begrenzt.

Zustandsübersicht

1. **Reset**: R = TRUE ⟶ CV := 0.
2. **Load**: R = FALSE, LD = TRUE ⟶ CV := PV.
3. **Zählen**: R = FALSE, LD = FALSE ⟶ CV wird je nach CU/CD angepasst.
4. **Ausgabe**: QU/QD werden anhand des aktuellen CV neu berechnet, CNF wird ausgelöst.

Anwendungsszenarien

- **Bidirektionale Zählaufgaben mit großem Wertebereich**, z. B. Stückzahlen oder Positionswerte, die sowohl inkrementiert als auch dekrementiert werden müssen und den Wertebereich von UINT/DINT überschreiten können.
- **Füllstands- oder Pufferzählung**, bei der QU/QD als Ober-/Untergrenzen-Signal für nachgeschaltete Logik dienen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **FB_CTUD**: dieselbe Funktion für den Datentyp INT.
- **FB_CTUD_DINT/FB_CTUD_LINT/FB_CTUD_ULINT**: dieselbe Funktion für andere Ganzzahltypen.
- **FB_CTU_UDINT/FB_CTD_UDINT**: die reinen Auf- bzw. Abwärtszähler-Varianten für UDINT, ohne kombinierte LD-Funktion.

Fazit

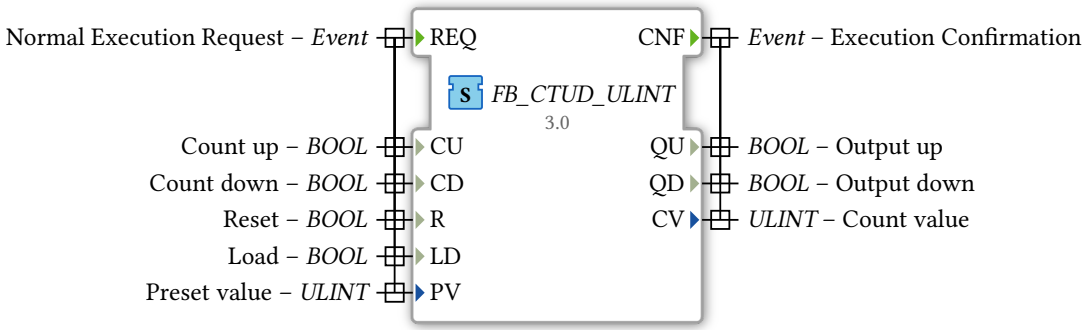
FB_CTUD_UDINT liefert einen vollständigen Auf-/Abwärtszähler mit Lade- und Reset-Funktion für den UDINT-Wertebereich und schließt die Lücke zwischen den bereits vorhandenen FB_CTUD_*-Varianten für andere Datentypen.

FB_CTUD_ULINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock FB_CTUD_ULINT ist ein Zähler mit Aufwärts- und Abwärtszählfunktion für den Datentyp ULINT (unsigned 64-bit integer). Er bietet zusätzliche Funktionen wie Reset, Laden eines Vorgabewerts und Statusausgaben für die Zählrichtung.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Startet die normale Ausführung des Zählvorgangs. Wird mit den Daten-Eingängen `CU`, `CD`, `R`, `LD` und `PV` verknüpft.

Ereignis-Ausgänge

- `CNF`: Bestätigt die Ausführung und liefert die aktuellen Zählerwerte über die Daten-Ausgänge `QU`, `QD` und `CV`.

Daten-Eingänge

- `CU` (BOOL): Signal zum Hochzählen (Count Up)
- `CD` (BOOL): Signal zum Herunterzählen (Count Down)
- `R` (BOOL): Reset-Signal (setzt den Zähler auf 0)
- `LD` (BOOL): Load-Signal (lädt den Vorgabewert `PV` in den Zähler)
- `PV` (ULINT): Preset Value (Vorgabewert für den Zähler)

Daten-Ausgänge

- `QU` (BOOL): Signalisiert, ob der Zählerwert größer oder gleich dem Vorgabewert ist
- `QD` (BOOL): Signalisiert, ob der Zählerwert kleiner oder gleich 0 ist
- `CV` (ULINT): Aktueller Zählerwert (Count Value)

Funktionsweise

Der Zähler führt folgende Operationen beim `REQ`-Ereignis aus:

1. Bei aktivem `R` (Reset) wird der Zähler auf 0 gesetzt
2. Bei aktivem `LD` (Load) wird der Vorgabewert `PV` in den Zähler geladen
3. Wenn weder Reset noch Load aktiv sind:
   - Bei aktivem `CU` (und nicht gleichzeitig `CD`) wird hochgezählt (bis zum Maximalwert 18446744073709551615)
   - Bei aktivem `CD` (und nicht gleichzeitig `CU`) wird heruntergezählt (bis zum Minimalwert 0)
4. Die Ausgänge `QU` und `QD` werden entsprechend dem aktuellen Zählerstand aktualisiert

Technische Besonderheiten

- Verwendet den 64-bit unsigned integer Datentyp (ULINT)
- Maximaler Zählerwert: 18446744073709551615
- Besondere Behandlung der gleichzeitigen Aktivierung von `CU` und `CD` (keine Zähloperation)
- Statusausgänge für Über-/Unterschreitung

Anwendungsszenarien

- Industrielle Zählaufgaben mit großen Zahlenbereichen
- Positionserfassung in großen Systemen
- Materialflussüberwachung mit hohen Stückzahlen
- Generelle Steuerungsaufgaben mit Aufwärts-/Abwärtszählfunktion

⚖️ Vergleich mit ähnlichen Bausteinen

- Gegenüber einfachen Zählern bietet dieser FB beide Zählrichtungen
- Im Vergleich zu kleineren Datentypen (z.B. INT) ermöglicht ULINT extrem große Zählbereiche
- Integrierte Statusausgänge (`QU`, `QD`) vereinfachen die Überwachung

Fazit

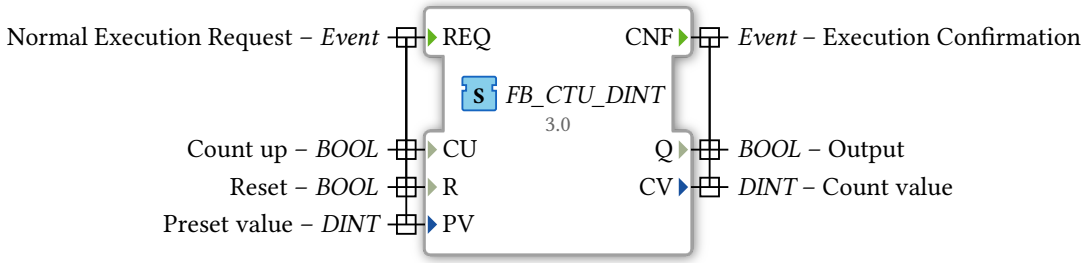
Der `FB_CTUD_ULINT` ist ein leistungsfähiger Zählbaustein für Anwendungen, die große Zahlenbereiche und flexible Zählfunktionen benötigen. Durch die kombinierte Aufwärts-/Abwärtszählfunktion und die zusätzlichen Steuer- und Statusmöglichkeiten eignet er sich für komplexe Zählaufgaben in industriellen Steuerungen.

FB_CTU_DINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der FB_CTU_DINT ist ein Aufwärtszähler (Counter Up) für den Datentyp DINT (32-Bit Ganzahl). Er zählt bei jedem Zählimpuls hoch und kann auf einen vordefinierten Wert zurückgesetzt werden. Der Baustein ist besonders geeignet für Anwendungen, die große Zählbereiche erfordern.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsrequest (triggert den Zählvorgang)

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung (wird nach Abschluss des Zählvorgangs ausgelöst)

Daten-Eingänge

- **CU** (BOOL): Zähl-Eingang (bei TRUE wird hochgezählt)
- **R** (BOOL): Reset-Eingang (bei TRUE wird der Zählerwert auf 0 gesetzt)
- **PV** (DINT): Preset-Wert (definiert den Vergleichswert für den Q-Ausgang)

Daten-Ausgänge

- **Q** (BOOL): Statusausgang (wird TRUE, wenn CV ≥ PV)
- **CV** (DINT): Aktueller Zählerwert (Count Value)

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der FB_CTU_DINT arbeitet nach folgenden Regeln:

1. Bei aktivem Reset (R=TRUE) wird der Zählerwert (CV) auf 0 zurückgesetzt
2. Bei aktivem Zähleingang (CU=TRUE) und nicht aktivem Reset wird der Zählerwert um 1 erhöht (bis zum Maximalwert von 2147483647)
3. Der Ausgang Q wird auf TRUE gesetzt, sobald der Zählerwert (CV) den Preset-Wert (PV) erreicht oder überschreitet

Technische Besonderheiten

- Verwendet 32-Bit Integer (DINT) für Zählwerte
- Maximaler Zählerwert: 2.147.483.647
- Implementiert als einfacher Funktionsblock (SimpleFB)
- Unterstützt den IEC 61131-3 Standard

Zustandsübersicht

Der Baustein hat keinen internen Zustandsautomaten. Die Logik wird bei jedem REQ-Ereignis ausgeführt.

Anwendungsszenarien

- Produktionszählung in Fertigungsanlagen
- Ereigniszählung in Prozesssteuerungen
- Überwachung von Zykluszeiten
- Allgemeine Zählaufgaben in der Automatisierungstechnik

⚖️ Vergleich mit ähnlichen Bausteinen

- Gegenüber FB_CTU_INT bietet FB_CTU_DINT einen größeren Zählbereich (32-Bit vs. 16-Bit)
- Im Vergleich zu FB_CTU_LINT hat FB_CTU_DINT einen kleineren Wertebereich, aber geringeren Speicherbedarf
- Ähnlich zu IEC 61131-3 CTU-Funktionsblöcken, aber in 4diac-IDE Implementierung

Fazit

Der FB_CTU_DINT ist ein robuster und einfach zu verwendender Aufwärtszähler für Anwendungen mit mittleren bis großen Zählbereichen. Durch seine klare Schnittstelle und einfache Funktionsweise eignet er sich besonders für Standard-Zählaufgaben in der Automatisierungstechnik.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

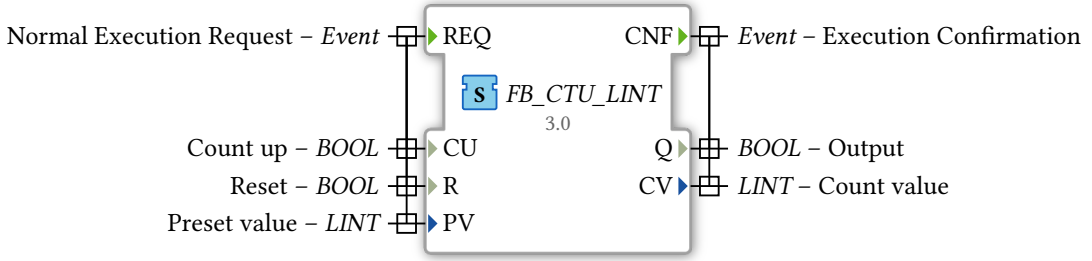
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

FB_CTU_LINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `FB_CTU_LINT` ist ein Aufwärtszähler (Counter Up) für große Ganzzahlen. Er zählt bei jedem eintreffenden Zählereignis hoch und kann auf einen vordefinierten Wert zurückgesetzt werden. Der Block verwendet den Datentyp `LINT` (64-Bit Ganzzahl) für den Zählwert und den Sollwert.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsrequest, löst die Zähloperation aus. Wird mit den Daten `CU` und `R` verknüpft.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung, wird nach der Zähloperation ausgegeben. Wird mit den Daten `Q` und `CV` verknüpft.

Daten-Eingänge

- **CU** (BOOL): Zählbefehl (Count Up). Bei TRUE wird der Zähler hochgezählt.
- **R** (BOOL): Reset. Bei TRUE wird der Zähler auf 0 zurückgesetzt.
- **PV** (LINT): Preset Value. Der Sollwert, bei dessen Erreichen der Ausgang `Q` auf TRUE gesetzt wird.

Daten-Ausgänge

- **Q** (BOOL): Ausgang. Wird TRUE, wenn der Zählerstand (`CV`) den Sollwert (`PV`) erreicht oder überschreitet.
- **CV** (LINT): Current Value. Aktueller Zählerstand.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock führt bei jedem `REQ`-Ereignis folgende Operationen durch:

1. Wenn `R` TRUE ist, wird der Zähler (`CV`) auf 0 zurückgesetzt.

2. Wenn `CU` TRUE ist und der Zähler noch nicht den Maximalwert (9223372036854775807) erreicht hat, wird der Zähler um 1 erhöht.
3. Der Ausgang `Q` wird auf TRUE gesetzt, wenn der Zählerstand (`CV`) den Sollwert (`PV`) erreicht oder überschreitet.

Technische Besonderheiten

- Verwendet 64-Bit Ganzzahlen (`LINT`) für Zählerstand und Sollwert
- Maximaler Zählerstand: 9223372036854775807 (2^63-1)
- Implementierung in Structured Text (ST)

Zustandsübersicht

Der Funktionsblock hat keinen internen Zustandsautomaten. Die Logik wird bei jedem `REQ`-Ereignis neu ausgeführt.

Anwendungsszenarien

- Zählung von Ereignissen in großen Zahlenbereichen
- Überwachung von Produktionsmengen
- Steuerung von Prozessen mit hohen Zählwerten

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu `FB_CTU` (Standard-Counter) unterstützt dieser Block größere Zahlenbereiche (`LINT` statt `INT`)
- Ähnlich zu `FB_CTUD`, aber ohne Abwärtszählfunktion

Fazit

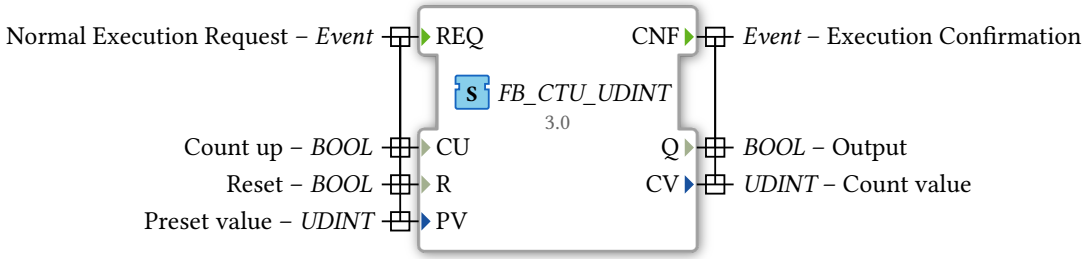
Der `FB_CTU_LINT` ist ein leistungsfähiger Aufwärtszähler für Anwendungen, die große Zählbereiche erfordern. Seine einfache Schnittstelle und robuste Implementierung machen ihn zu einem nützlichen Baustein für industrielle Steuerungsanwendungen.

FB_CTU_UDINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `FB_CTU_UDINT` ist ein Aufwärtszähler (Counter Up) für den Datentyp `UDINT` (unsigned double integer). Er zählt bei jedem Ereignis `REQ` hoch, solange der Eingang `CU` aktiv ist, und kann über den Eingang `R` zurückgesetzt werden. Der Zähler vergleicht den aktuellen Zählerstand mit einem vorgegebenen Sollwert (`PV`) und gibt über den Ausgang `Q` ein Signal aus, wenn dieser erreicht oder überschritten wird.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung des Funktionsblocks. Wird mit den Daten-Eingängen `CU` und `R` verknüpft.

Ereignis-Ausgänge

- **CNF**: Bestätigt die Ausführung des Funktionsblocks. Wird mit den Daten-Ausgängen `Q` und `CV` verknüpft.

Daten-Eingänge

- **CU** (`BOOL`): Signal zum Hochzählen. Bei `TRUE` wird der Zählerstand erhöht.
- **R** (`BOOL`): Signal zum Zurücksetzen des Zählers. Bei `TRUE` wird der Zählerstand auf `0` gesetzt.
- **PV** (`UDINT`): Sollwert (Preset Value), mit dem der aktuelle Zählerstand verglichen wird.

Daten-Ausgänge

- **Q** (`BOOL`): Signal, das `TRUE` wird, wenn der Zählerstand (`CV`) den Sollwert (`PV`) erreicht oder überschreitet.
- **CV** (`UDINT`): Aktueller Zählerstand (Count Value).

Funktionsweise

Der Funktionsblock führt bei jedem `REQ`-Ereignis folgende Schritte aus:

1. Wenn `R` aktiv ist (`TRUE`), wird der Zählerstand (`CV`) auf `0` zurückgesetzt.
2. Wenn `CU` aktiv ist (`TRUE`) und der Zählerstand (`CV`) kleiner als der Maximalwert (`4294967295`) ist, wird der Zählerstand um `1` erhöht.
3. Der Ausgang `Q` wird auf `TRUE` gesetzt, wenn der Zählerstand (`CV`) den Sollwert (`PV`) erreicht oder überschreitet.

Technische Besonderheiten

- Der Zählerstand (`CV`) ist vom Typ `UDINT` und kann Werte zwischen `0` und `4294967295` annehmen.
- Der Funktionsblock ist deterministisch und führt die Zähloperation bei jedem `REQ`-Ereignis aus.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände außer dem aktuellen Zählerstand (`CV`). Die Logik wird bei jedem `REQ`-Ereignis neu ausgewertet.

Anwendungsszenarien

- Zählen von Ereignissen in Steuerungsanwendungen (z. B. Produktionszähler).
- Überwachung von Prozessen, bei denen eine bestimmte Anzahl von Schritten erreicht werden muss.
- Als Teil einer größeren Steuerungslogik, die auf Zählerstände reagiert.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Zählern (z. B. `FB_CTU_INT`) arbeitet dieser Funktionsblock mit dem Datentyp `UDINT`, was einen größeren Wertebereich ermöglicht.
- Ähnliche Funktionsblöcke könnten zusätzliche Funktionen wie Abwärtszählen oder eine kombinierte Auf-/Abwärtszählung bieten.

Fazit

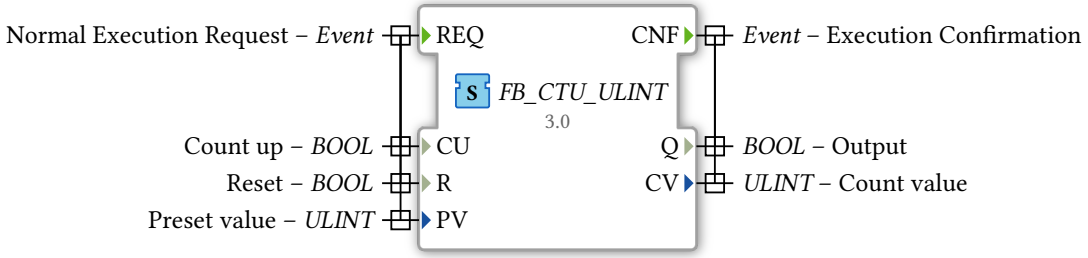
Der `FB_CTU_UDINT` ist ein einfacher und effizienter Aufwärtszähler für Anwendungen, die einen großen Wertebereich benötigen. Seine klare Schnittstelle und deterministische Funktionsweise machen ihn zu einem zuverlässigen Baustein in Steuerungsanwendungen.

FB_CTU_ULINT

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `FB_CTU_ULINT` ist ein Aufwärtszähler (Counter Up) für den Datentyp `ULINT` (Unsigned Long Integer). Er zählt bei jedem Zählereignis hoch und kann auf einen vordefinierten Wert zurückgesetzt werden. Der Block ist besonders für Anwendungen geeignet, die große Zählbereiche erfordern.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Wird verwendet, um den Zählvorgang auszulösen. Verknüpft mit den Daten `CU` und `R`.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Wird nach der Verarbeitung des Zählvorgangs ausgelöst. Verknüpft mit den Daten `Q` und `CV`.

Daten-Eingänge

- **CU** (BOOL): Zählaufforderung. Bei `TRUE` wird der Zähler erhöht, sofern nicht `R` aktiv ist.
- **R** (BOOL): Reset. Setzt den Zählerwert (`CV`) auf `0` zurück, wenn `TRUE`.
- **PV** (ULINT): Voreingestellter Wert. Definiert den Schwellwert, bei dem der Ausgang `Q` aktiviert wird.

Daten-Ausgänge

- **Q** (BOOL): Ausgang. Wird `TRUE`, wenn der Zählerwert (`CV`) den voreingestellten Wert (`PV`) erreicht oder überschreitet.
- **CV** (ULINT): Aktueller Zählerwert. Gibt den aktuellen Stand des Zählers zurück.

Funktionsweise

Der Funktionsblock erhöht den Zählerwert (`CV`) um 1, wenn der Eingang `CU` aktiviert ist (`TRUE`) und der Zählerwert nicht den Maximalwert (`18446744073709551615`) erreicht hat. Wenn der Eingang `R` aktiviert ist (`TRUE`), wird der Zählerwert auf `0` zurückgesetzt. Der Ausgang `Q` wird auf `TRUE` gesetzt, sobald der Zählerwert (`CV`) den voreingestellten Wert (`PV`) erreicht oder überschreitet.

Technische Besonderheiten

- Der Zählerwert (`CV`) ist vom Typ `ULINT`, was einen sehr großen Zählbereich ermöglicht (0 bis 18446744073709551615).
- Der Block ist deterministisch und eignet sich für Echtzeitanwendungen.

Zustandsübersicht

- **Reset-Zustand**: `CV = 0` (wenn `R = TRUE`).
- **Zählzustand**: `CV` wird um 1 erhöht (wenn `CU = TRUE` und `R = FALSE`).
- **Schwellwert erreicht**: `Q = TRUE` (wenn `CV >= PV`).

Anwendungsszenarien

- Zählung von Produktionsstücken in Hochgeschwindigkeitsanlagen.
- Überwachung von Ereignissen in lang laufenden Systemen.
- Steuerung von Prozessen mit großen Zählbereichen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu `FB_CTU_INT` oder `FB_CTU_DINT` bietet `FB_CTU_ULINT` einen deutlich größeren Zählbereich.
- Ähnlich zu `FB_CTU`, aber spezialisiert auf den `ULINT`-Datentyp.

Fazit

Der `FB_CTU_ULINT` ist ein leistungsstarker Aufwärtszähler für Anwendungen, die große Zählbereiche erfordern. Seine einfache Handhabung und deterministische Funktionsweise machen ihn zu einer zuverlässigen Wahl für industrielle Steuerungsanwendungen.

2.4.1.9 Flankenerkennung

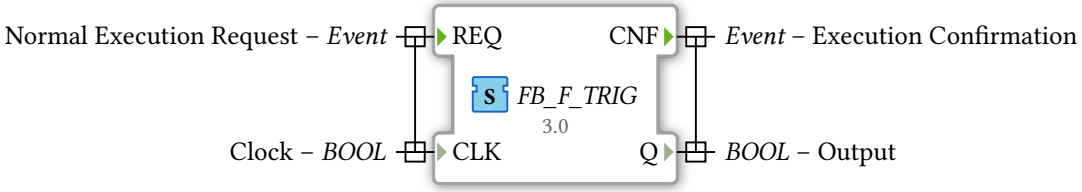
Hier finden Sie Funktionsbausteine zur Erkennung von Signalflanken (Steigend, Fallend).

FB_F_TRIG

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `FB_F_TRIG` dient zur Erkennung einer fallenden Flanke eines booleschen Signals. Er wird häufig in Steuerungsanwendungen eingesetzt, um Zustandsänderungen von `TRUE` nach `FALSE` zu detektieren und entsprechende Aktionen auszulösen.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ` (Normal Execution Request): Startet die Ausführung des Funktionsblocks. Wird mit dem Daten-Eingang `CLK` verknüpft.

Ereignis-Ausgänge

- `CNF` (Execution Confirmation): Bestätigt die Ausführung des Funktionsblocks. Wird mit dem Daten-Ausgang `Q` verknüpft.

Daten-Eingänge

- `CLK` (BOOL): Das zu überwachende Signal, dessen fallende Flanke erkannt werden soll.

Daten-Ausgänge

- `Q` (BOOL): Gibt `TRUE` aus, wenn eine fallende Flanke erkannt wurde, ansonsten `FALSE`.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock speichert den vorherigen Zustand des `CLK`-Signals in einer internen Variable `MEM`. Bei jedem Aufruf über das `REQ`-Ereignis wird geprüft, ob das aktuelle `CLK`-Signal `FALSE` ist und der vorherige Zustand (`MEM`) `TRUE` war. In diesem Fall wird `Q` auf `TRUE` gesetzt, ansonsten auf `FALSE`. Der aktuelle Zustand von `CLK` wird dann in `MEM` gespeichert.

Technische Besonderheiten

- Der Funktionsblock verwendet eine interne Speichervariable (`MEM`), um den vorherigen Zustand des `CLK`-Signals zu speichern.
- Die Logik zur Erkennung der fallenden Flanke wird in einem ST-Algorithmus implementiert.

Zustandsübersicht

1. **Initialisierung**: `MEM` wird mit `TRUE` initialisiert.
2. **Erkennung einer fallenden Flanke**:
   - Wenn `CLK` von `TRUE` auf `FALSE` wechselt, wird `Q` auf `TRUE` gesetzt.
   - In allen anderen Fällen wird `Q` auf `FALSE` gesetzt.
3. **Speichern des aktuellen Zustands**: Der aktuelle Zustand von `CLK` wird in `MEM` gespeichert.

Anwendungsszenarien

- Erkennung von Taster- oder Schalterbetätigungen in Steuerungssystemen.
- Synchronisation von Prozessen bei Signaländerungen.
- Triggerung von Aktionen bei fallenden Flanken in zeitkritischen Anwendungen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **FB_R_TRIG**: Erkennung einer steigenden Flanke (von `FALSE` nach `TRUE`).
- **FB_R_F_TRIG**: Kombinierte Erkennung von steigenden und fallenden Flanken.

- **FB_F_TRIG** ist spezialisiert auf die Erkennung von fallenden Flanken und daher effizienter in dieser spezifischen Anwendung.

🛠️ Zugehörige Übungen

- Uebung_178
- Uebung_178_AX

Fazit

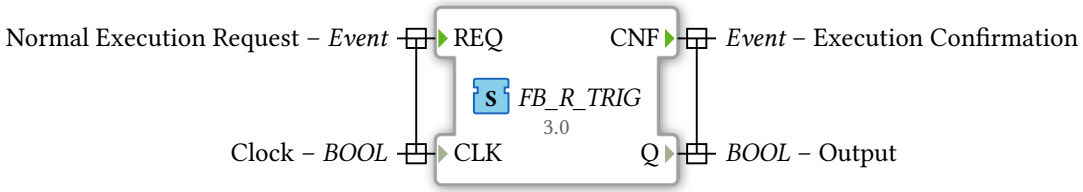
Der `FB_F_TRIG` ist ein einfacher und effizienter Funktionsblock zur Erkennung von fallenden Flanken in booleschen Signalen. Durch seine klare Schnittstelle und einfache Funktionsweise eignet er sich ideal für Steuerungsanwendungen, bei denen Zustandsänderungen detektiert werden müssen.

FB_R_TRIG

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `FB_R_TRIG` dient der Erkennung einer steigenden Flanke eines booleschen Signals. Er wird typischerweise verwendet, um Zustandsänderungen von `FALSE` nach `TRUE` zu detektieren und darauf zu reagieren.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normaler Ausführungsanforderung. Wird verwendet, um die Flankenerkennung auszulösen. Ist mit dem Daten-Eingang `CLK` verknüpft.

Ereignis-Ausgänge

- **CNF**: Ausführungsbestätigung. Wird ausgelöst, nachdem die Flankenerkennung abgeschlossen ist. Ist mit dem Daten-Ausgang `Q` verknüpft.

Daten-Eingänge

- **CLK** (BOOL): Das Eingangssignal, dessen steigende Flanke erkannt werden soll.

Daten-Ausgänge

- **Q** (BOOL): Das Ausgangssignal, das `TRUE` wird, wenn eine steigende Flanke am Eingang `CLK` erkannt wird.

Adapter

Der Funktionsblock verwendet keine Adapter.

Funktionsweise

Der `FB_R_TRIG` speichert den vorherigen Zustand des Eingangssignals `CLK` in einer internen Variable `MEM`. Bei jedem Aufruf des Algorithmus über das Ereignis `REQ` wird geprüft, ob der aktuelle Zustand von `CLK` `TRUE` und der vorherige Zustand `FALSE` war. Ist dies der Fall, wird der Ausgang `Q` auf `TRUE` gesetzt, andernfalls auf `FALSE`. Der aktuelle Zustand von `CLK` wird dann in `MEM` gespeichert für den nächsten Durchlauf.

Technische Besonderheiten

- Der Funktionsblock ist gemäß dem Standard IEC 61499-1 implementiert.
- Er verwendet einen einfachen Algorithmus zur Flankenerkennung, der effizient und schnell ausgeführt werden kann.

Zustandsübersicht

Der Funktionsblock hat keinen komplexen Zustandsautomaten. Der Zustand wird lediglich durch den Wert der internen Variable `MEM` repräsentiert, die den vorherigen Zustand von `CLK` speichert.

Anwendungsszenarien

- Erkennung von Tastendrücken oder Schalterbetätigungen.
- Synchronisation von Prozessen auf steigende Flanken von Steuersignalen.
- Triggerung von Aktionen bei bestimmten Ereignissen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **FB_F_TRIG**: Erkennung einer fallenden Flanke (von `TRUE` nach `FALSE`).
- **FB_R_TRIG** ist spezifisch für steigende Flanken und daher komplementär zu `FB_F_TRIG`.

🛠️ Zugehörige Übungen

- Uebung_177
- Uebung_177_AX

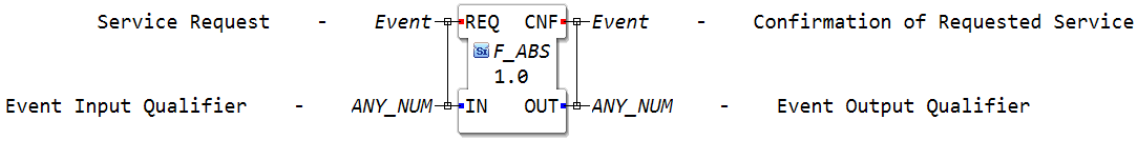
Fazit

Der `FB_R_TRIG` ist ein einfacher und effektiver Funktionsblock zur Erkennung steigender Flanken. Seine klare Schnittstellenstruktur und einfache Funktionsweise machen ihn zu einem nützlichen Baustein in vielen Steuerungsanwendungen.

2.4.1.10 Numerische Funktionen

Hier finden Sie mathematische Funktionen wie Winkelfunktionen, Logarithmus, Wurzel etc.

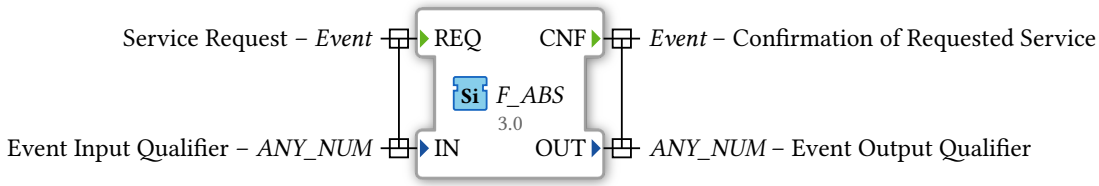
F_ABS



• – ◦ • – ◦ • – ◦ •

Einleitung

Der F_ABS Funktionsblock berechnet den absoluten Wert einer numerischen Eingabe. Er gehört zur Kategorie der Standardfunktionen für numerische Variablen gemäß IEC 61131-3 und ist Teil des iec61131::numerical Pakets.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Service-Anfrage - löst die Berechnung des absoluten Wertes aus

Ereignis-Ausgänge

- **CNF**: Bestätigung der angeforderten Serviceleistung - signalisiert die Beendigung der Berechnung

Daten-Eingänge

- **IN**: Numerische Eingabe (ANY_NUM) - der Wert, dessen absoluter Wert berechnet werden soll

Daten-Ausgänge

- **OUT**: Numerische Ausgabe (ANY_NUM) - der berechnete absolute Wert der Eingabe

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Bei Empfang eines REQ-Ereignisses berechnet der F_ABS Block den absoluten Wert des an IN übergebenen numerischen Wertes. Das Ergebnis wird an OUT ausgegeben und durch ein CNF-Ereignis bestätigt. Der Block unterstützt alle numerischen Datentypen (ANY_NUM).

Technische Besonderheiten

- Unterstützt alle numerischen Datentypen gemäß IEC 61131-3
- Einfache, deterministische Funktionsweise ohne interne Zustände
- Sofortige Berechnung bei REQ-Ereignis

Zustandsübersicht

Der F_ABS Block besitzt keine internen Zustände. Er verhält sich rein funktional und reagiert sofort auf eingehende REQ-Ereignisse.

Anwendungsszenarien

- Signalverarbeitung zur Entfernung von Vorzeichen
- Betragsberechnungen in mathematischen Operationen
- Verwendung in Regelungstechnik für Abstandsberechnungen
- Vorverarbeitung von Messwerten

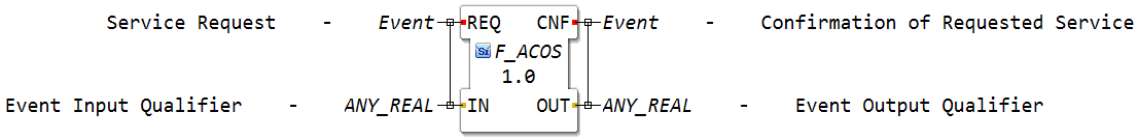
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen mathematischen Funktionsblöcken zeichnet sich F_ABS durch seine Einfachheit und spezifische Funktionalität aus. Während Blöcke wie F_ADD oder F_MUL komplexere Operationen durchführen, konzentriert sich F_ABS ausschließlich auf die Betragsbildung.

Fazit

Der F_ABS Funktionsblock bietet eine zuverlässige und standardkonforme Lösung zur Berechnung absoluter Werte in IEC 61131-3-basierten Steuerungssystemen. Seine einfache Schnittstelle und deterministische Funktionsweise machen ihn zu einem robusten Baustein für numerische Operationen.

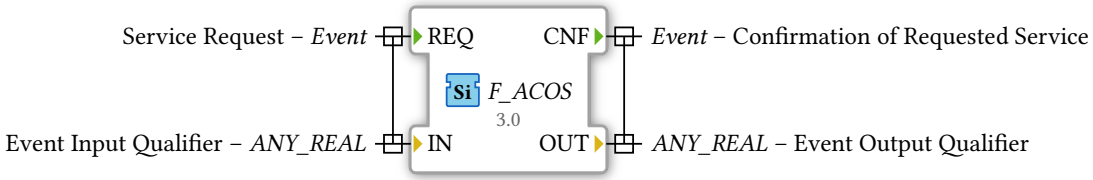
F_ACOS



• — ○ • — ○ • — ○ •

Einleitung

Der F_ACOS Funktionsblock berechnet den Hauptwert des Arkuskosinus (Arccosinus) einer numerischen Eingabe. Er implementiert die mathematische Funktion zur Bestimmung des Winkels, dessen Kosinus dem Eingabewert entspricht. Der Block gehört zur Kategorie der Standardfunktionen für numerische Variablen gemäß IEC 61131-3.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Service-Anfrage - löst die Berechnung des Arkuskosinus aus

Ereignis-Ausgänge

- **CNF**: Bestätigung der angeforderten Serviceleistung - signalisiert den Abschluss der Berechnung

Daten-Eingänge

- **IN**: Eingabewert vom Typ ANY_REAL - der Wert, für den der Arkuskosinus berechnet werden soll

Daten-Ausgänge

- **OUT**: Ausgabewert vom Typ ANY_REAL - das Ergebnis der Arkuskosinus-Berechnung

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Bei Eintreffen eines REQ-Ereignisses berechnet der F_ACOS-Block den Arkuskosinus des aktuellen IN-Wertes. Das Ergebnis wird am OUT-Ausgang ausgegeben und gleichzeitig wird ein CNF-Ereignis generiert, um den erfolgreichen Abschluss der Operation zu signalisieren.

Die Funktion berechnet den Hauptwert des Arkuskosinus, der im Bereich von 0 bis π (0° bis 180°) liegt.

Technische Besonderheiten

- Unterstützt den ANY_REAL-Datentyp für maximale Flexibilität bei numerischen Eingaben
- Einhaltung des IEC 61131-3 Standards für Programmierbare Steuerungen
- Liefert den mathematisch korrekten Hauptwert des Arkuskosinus
- Garantiert definiertes Verhalten für gültige Eingabewerte im Bereich [−1, 1]

Zustandsübergänge

1. **Idle-Zustand**: Wartet auf REQ-Ereignis
2. **Berechnungs-Zustand**: Verarbeitet die Arkuskosinus-Funktion bei REQ-Eingang
3. **Ausgabe-Zustand**: Sendet CNF-Ereignis mit berechnetem OUT-Wert

Anwendungsszenarien

- Winkelmessung und -berechnung in technischen Systemen
- Robotik und Bewegungssteuerung
- Signalverarbeitung und Regelungstechnik
- Wissenschaftliche Berechnungen und Simulationen
- Geometrische Transformationen in CAD/CAM-Systemen

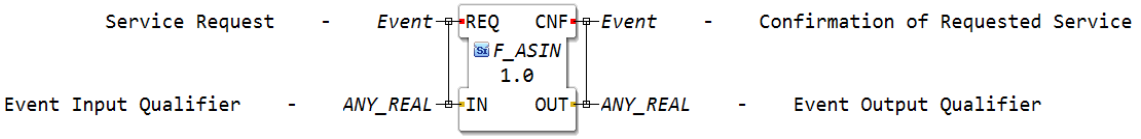
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen trigonometrischen Funktionen wie F_SIN (Sinus) oder F_COS (Kosinus) bietet F_ACOS die inverse Operation. Während F_COS einen Winkel auf einen Wert zwischen −1 und 1 abbildet, bildet F_ACOS einen Wert zwischen −1 und 1 auf einen Winkel zwischen 0 und π ab.

Fazit

Der F_ACOS Funktionsblock stellt eine zuverlässige und standardkonforme Implementierung der Arkuskosinus-Funktion für industrielle Steuerungssysteme bereit. Seine einfache Schnittstelle und deterministische Funktionsweise machen ihn ideal für Echtzeitanwendungen in Automatisierungslösungen.

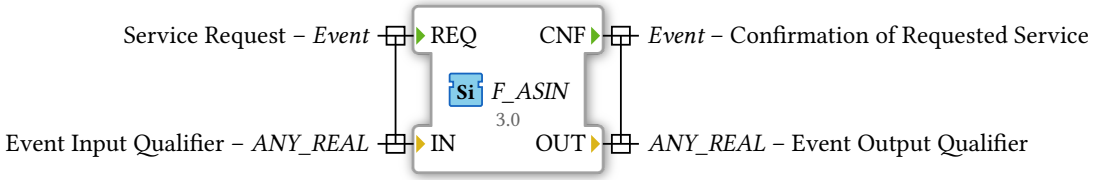
F_ASIN



• — ○ • — ○ • — ○ •

Einleitung

Der F_ASIN Funktionsblock ist ein Standard-Funktionsbaustein gemäß IEC 61131-3, der den Arkussinus (Arcussinus) einer numerischen Eingabe berechnet. Er gehört zur Kategorie der mathematischen Standardfunktionen für eine numerische Variable und liefert den Hauptwert der Arkussinus-Funktion.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Service-Anfrage - löst die Berechnung des Arkussinus aus

Ereignis-Ausgänge

- **CNF**: Bestätigung der angeforderten Serviceleistung - signalisiert die Beendigung der Berechnung

Daten-Eingänge

- **IN**: Eingabewert vom Typ ANY_REAL - der Wert, für den der Arkussinus berechnet werden soll

Daten-Ausgänge

- **OUT**: Ausgabewert vom Typ ANY_REAL - das Ergebnis der Arkussinus-Berechnung

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der F_ASIN Funktionsblock berechnet den Arkussinus (inverse Sinusfunktion) des Eingabewertes IN. Bei Aktivierung des REQ-Ereignisses wird die Berechnung durchgeführt. Nach Abschluss der Berechnung wird das CNF-Ereignis zusammen mit dem Ergebniswert OUT ausgegeben.

Die Funktion arbeitet nach der mathematischen Definition: OUT = arcsin(IN), wobei das Ergebnis im Bereich $[-\pi/2, \pi/2]$ Radiant liegt.

Technische Besonderheiten

- Unterstützt den ANY_REAL Datentyp für maximale Flexibilität mit verschiedenen realen Zahlentypen
- Einhaltung des IEC 61131-3 Standards für numerische Funktionen
- Einfache ereignisgesteuerte Verarbeitung
- Liefert den Hauptwert der Arkussinus-Funktion

Zustandsübersicht

Der Funktionsblock besitzt einen einfachen Zustandsautomaten:

1. **Wartezustand**: Wartet auf REQ-Ereignis
2. **Berechnungszustand**: Führt Arkussinus-Berechnung durch
3. **Ausgabezustand**: Sendet CNF-Ereignis mit Ergebnis

Anwendungsszenarien

- Berechnung von Winkeln aus Sinuswerten in trigonometrischen Anwendungen
- Signalverarbeitung in Regelungstechnik und Automatisierung
- Geometrische Berechnungen in technischen Systemen
- Wissenschaftliche und ingenieurtechnische Berechnungen

⚖️ Vergleich mit ähnlichen Bausteinen

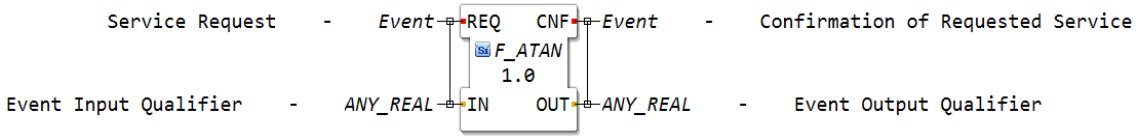
Im Vergleich zu anderen trigonometrischen Funktionen:

- F_ASIN ist die inverse Funktion zu F_SIN
- Ähnlich zu F_ACOS (Arkuskosinus) und F_ATAN (Arkustangens)
- Liefert Ergebnisse im spezifischen Wertebereich für Arkussinus

Fazit

Der F_ASIN Funktionsblock bietet eine standardkonforme, zuverlässige Implementierung der Arkussinus-Funktion für Automatisierungsanwendungen. Seine einfache Schnittstelle und ereignisgesteuerte Verarbeitung machen ihn ideal für den Einsatz in komplexeren Steuerungsalgorithmen, die trigonometrische Berechnungen erfordern.

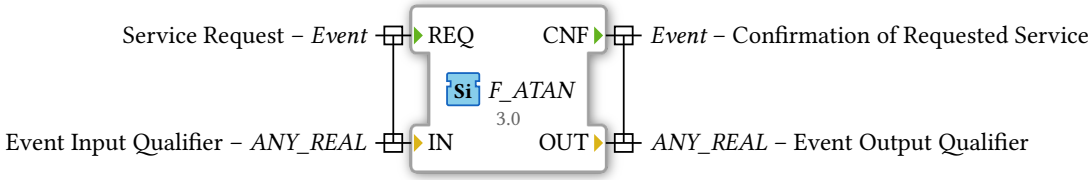
F_ATAN



• – ◦ • – ◦ • – ◦ •

Einleitung

Der F_ATAN Funktionsblock berechnet den Arkustangens (Arcustangens) einer reellen Zahl. Es handelt sich um eine Standardfunktion für eine numerische Variable gemäß IEC 61131-3. Der Funktionsblock wandelt einen Tangenswert in den entsprechenden Winkel im Bogenmaß um.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Service-Anfrage - löst die Berechnung des Arkustangens aus

Ereignis-Ausgänge

- **CNF**: Bestätigung der angeforderten Serviceleistung - signalisiert die Beendigung der Berechnung

Daten-Eingänge

- **IN**: Eingabewert vom Typ ANY_REAL - der Tangenswert, für den der Arkustangens berechnet werden soll

Daten-Ausgänge

- **OUT**: Ausgabewert vom Typ ANY_REAL - der berechnete Arkustangens-Wert im Bogenmaß

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Bei Empfang eines REQ-Ereignisses berechnet der F_ATAN Funktionsblock den Arkustangens des an IN übergebenen Wertes. Das Ergebnis wird an OUT ausgegeben und gleichzeitig wird das CNF-Ereignis generiert, um den Abschluss der Berechnung zu signalisieren.

Die Funktion berechnet den Hauptwert des Arkustangens, der im Bereich von $-\pi/2$ bis $\pi/2$ liegt.

Technische Besonderheiten

- Unterstützt den ANY_REAL Datentyp, was verschiedene reelle Datentypen (REAL, LREAL) ermöglicht
- Einhaltung des IEC 61131-3 Standards
- Deterministisches Verhalten bei der Berechnung
- Geeignet für Echtzeitanwendungen

Zustandsübergänge

1. **Initialzustand**: Warten auf REQ-Ereignis
2. **Berechnungszustand**: Bei REQ-Ereignis wird ATAN(IN) berechnet

3. **Ausgabezustand**: Ergebnis wird an OUT gesendet und CNF-Ereignis wird ausgelöst
4. Rückkehr zum Initialzustand

Anwendungsszenarien

- Winkelmessung und -berechnung in Steuerungssystemen
- Koordinatentransformationen in Robotikanwendungen
- Regelungstechnik bei Positionssteuerungen
- Signalverarbeitung in mathematischen Berechnungen
- Navigation und Orientierungssysteme

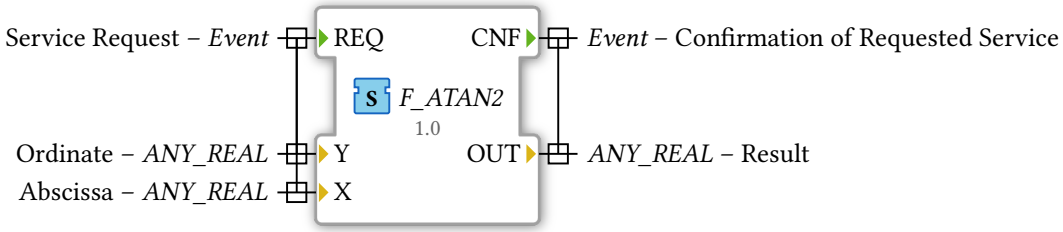
⚖️ Vergleich mit ähnlichen Bausteinen

- **F_ATAN2**: Berechnet den Arkustangens von zwei Koordinaten (x,y) und liefert einen Winkel im Bereich von -π bis π
- **F_SIN/F_COS**: Trigonometrische Grundfunktionen für Sinus und Cosinus
- **F_TAN**: Berechnet den Tangens eines Winkels (Umkehrfunktion zu F_ATAN)

Fazit

Der F_ATAN Funktionsblock bietet eine zuverlässige und standardkonforme Implementierung der Arkustangens-Funktion für Automatisierungsanwendungen. Seine einfache Schnittstelle und deterministische Funktionsweise machen ihn ideal für Echtzeitanwendungen in der Industrieautomation, wo trigonometrische Berechnungen benötigt werden.

F_ATAN2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `F_ATAN2` berechnet den Hauptwert des Arkustangens von `Y/X` unter Berücksichtigung der Vorzeichen beider Argumente (Zwei-Parameter-Arkustangens). Im Gegensatz zur einfachen `ATAN`-Funktion (die nur ein Argument entgegennimmt) liefert `F_ATAN2` den Winkel im korrekten Quadranten über den vollen Winkelbereich von `-π` bis `+π`.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Berechnung aus, trägt `Y` und `X`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT`.

Daten-Eingänge

- **Y** (ANY_REAL): Ordinate (Y-Koordinate).
- **X** (ANY_REAL): Abszisse (X-Koordinate).

Daten-Ausgänge

- **OUT** (ANY_REAL): Der berechnete Winkel in Radiant, im Bereich `(-π, π]`.

Funktionsweise

Bei Eintreffen von `REQ` berechnet der Baustein den Winkel zwischen der positiven X-Achse und dem Punkt `(X, Y)` — äquivalent zu `ATAN(Y/X)`, jedoch unter Auswertung der Vorzeichen von `X` und `Y` einzeln, um

den korrekten Quadranten zu bestimmen (inklusive der Sonderfälle `X = 0`). Das Ergebnis wird über `OUT` ausgegeben, anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **Quadrantenkorrektheit**: Im Gegensatz zu `ATAN(Y/X)` liefert `F_ATAN2` auch bei negativem `X` (2. und 3. Quadrant) den korrekten Winkel, ohne dass der Aufrufer den Quadranten manuell nachkorrigieren muss.
- **`ANY_REAL`-Typisierung**: Akzeptiert sowohl `REAL` als auch `LREAL` für `Y`, `X` und `OUT`.
- **Sonderfall `X = 0`**: Liefert `+π/2` bzw. `-π/2` je nach Vorzeichen von `Y`, statt einer Division-durch-Null wie bei direkter Verwendung von `ATAN(Y/X)`.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur Berechnung und zu `CNF`.

Anwendungsszenarien

- **Winkelberechnung aus kartesischen Koordinaten**: z. B. Ermittlung der Blickrichtung oder Ausrichtung eines Objekts aus dessen X/Y-Position relativ zu einem Referenzpunkt.
- **Regelungstechnik**: Berechnung von Phasenwinkeln oder Lenkwinkeln aus zwei Komponentensignalen.
- **Navigation**: Kursberechnung aus Geschwindigkeitskomponenten.

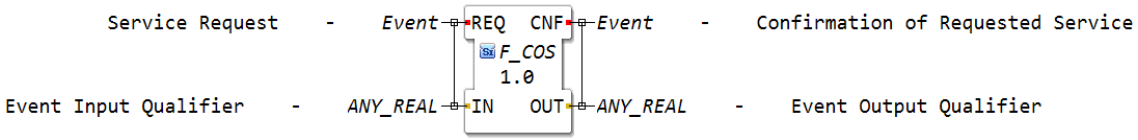
⚖️ Vergleich mit ähnlichen Bausteinen

- **`F_ATAN`**: berechnet den Arkustangens eines einzelnen Werts, liefert nur Winkel im Bereich `(-π/2, π/2)` und benötigt eine manuelle Quadrantenkorrektur.

Fazit

`F_ATAN2` liefert eine quadrantenkorrekte Winkelberechnung aus zwei kartesischen Komponenten und ist der Standardbaustein für Winkel-/Richtungsberechnungen, bei denen `ATAN` allein nicht ausreicht.

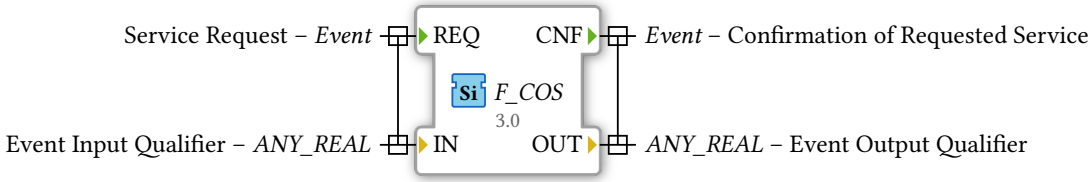
F_COS



• – ◦ • – ◦ • – ◦ •

Einleitung

Der F_COS Funktionsblock berechnet den Kosinus eines Eingangswertes in Bogenmaß (Radiant). Es handelt sich um eine Standardfunktion für numerische Berechnungen gemäß IEC 61131-3.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Service-Anfrage - löst die Berechnung des Kosinus aus

Ereignis-Ausgänge

- **CNF**: Bestätigung der angeforderten Serviceleistung - signalisiert die Beendigung der Berechnung

Daten-Eingänge

- **IN**: Eingangswert in Radiant (ANY_REAL Datentyp)

Daten-Ausgänge

- **OUT**: Berechneter Kosinuswert des Eingangswertes (ANY_REAL Datentyp)

Adapter

Keine Adapter vorhanden.

Funktionsweise

Bei Empfang eines REQ-Ereignisses berechnet der F_COS-Block den Kosinus des an IN übergebenen Wertes. Der Eingangswert wird als Winkel in Radiant interpretiert. Nach Abschluss der Berechnung wird das CNF-Ereignis zusammen mit dem Ergebnis an OUT ausgegeben.

Technische Besonderheiten

- Unterstützt den ANY_REAL Datentyp für maximale Flexibilität
- Berechnung erfolgt in Radiant (Bogenmaß)
- Einhaltung des IEC 61131-3 Standards
- Teil des iec61131::numerical Pakets

Zustandsübergänge

1. **Wartezustand**: Block wartet auf REQ-Ereignis
2. **Berechnungszustand**: Bei REQ-Ereignis wird Kosinus berechnet
3. **Ausgabezustand**: CNF-Ereignis mit Ergebnis wird ausgegeben
4. **Rückkehr**: Zurück in Wartezustand

Anwendungsszenarien

- Signalverarbeitung in Regelungstechnik
- Mathematische Berechnungen in Steuerungssystemen
- Winkelfunktionen in technischen Anwendungen
- Berechnungen in Robotik und Motion Control

⚖️ Vergleich mit ähnlichen Bausteinen

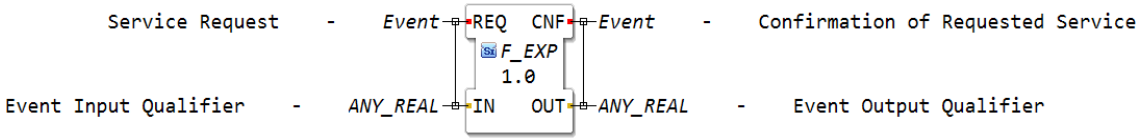
Im Vergleich zu anderen trigonometrischen Funktionen:

- F_SIN: Berechnet Sinus statt Kosinus
- F_TAN: Berechnet Tangens
- F_ACOS: Berechnet Arkuskosinus (Umkehrfunktion)

Fazit

Der F_COS Funktionsblock bietet eine zuverlässige und standardkonforme Implementierung der Kosinus-Funktion für industrielle Steuerungssysteme. Seine einfache Schnittstelle und flexible Datentypunterstützung machen ihn ideal für verschiedene Anwendungen in der Automatisierungstechnik.

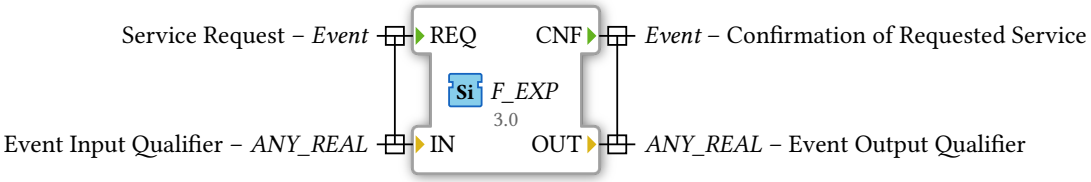
F_EXP



• – ◦ • – ◦ • – ◦ •

Einleitung

Der F_EXP Funktionsblock berechnet die natürliche Exponentialfunktion (e^x) für numerische Eingabewerte. Er gehört zu den Standardfunktionen für numerische Variablen nach IEC 61131-3 und wird für mathematische Berechnungen in Steuerungsanwendungen eingesetzt.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Service Request - Löst die Berechnung der Exponentialfunktion aus

Ereignis-Ausgänge

- **CNF**: Confirmation of Requested Service - Bestätigt die erfolgreiche Berechnung

Daten-Eingänge

- **IN**: ANY_REAL - Numerischer Eingabewert, für den die Exponentialfunktion berechnet werden soll

Daten-Ausgänge

- **OUT**: ANY_REAL - Ergebnis der Exponentialfunktion (e^IN)

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Bei Eintreffen eines REQ-Ereignisses berechnet der F_EXP-Block die natürliche Exponentialfunktion des an IN übergebenen Wertes. Das Ergebnis wird an OUT ausgegeben und durch ein CNF-Ereignis bestätigt. Die Funktion entspricht mathematisch der Berechnung von e^x, wobei x der Eingabewert ist.

Technische Besonderheiten

- Unterstützt den ANY_REAL Datentyp für maximale Flexibilität bei numerischen Eingaben
- Echtzeitfähige Berechnung der Exponentialfunktion
- Einfache und direkte Schnittstelle ohne zusätzliche Parameter
- Konforme Implementierung nach IEC 61131-3 Standard

Zustandsübergänge

1. **Wartezustand**: Block wartet auf REQ-Ereignis
2. **Berechnungszustand**: Bei REQ-Ereignis wird e^IN berechnet
3. **Ausgabezustand**: Ergebnis wird an OUT gesetzt und CNF-Ereignis ausgelöst
4. Rückkehr in Wartezustand

Anwendungsszenarien

- Mathematische Modellierung und Simulation
- Verstärkungsberechnungen in Regelungstechnik
- Wachstumsprozesse und exponentielle Entwicklungen
- Signalverarbeitung und Filterung
- Wissenschaftliche Berechnungen in industriellen Steuerungen

⚖️ Vergleich mit ähnlichen Bausteinen

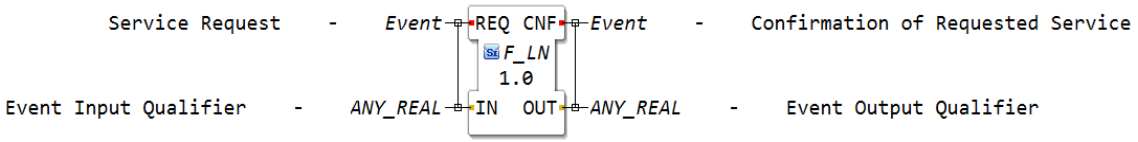
Im Vergleich zu anderen mathematischen Funktionenblöcken wie F_LN (natürlicher Logarithmus) oder F_POWER (Potenzfunktion) bietet F_EXP eine spezialisierte Implementierung für die Exponentialfunktion mit optimierter Performance. Die Verwendung von ANY_REAL ermöglicht breitere Kompatibilität als typspezifische Implementierungen.

Fazit

Der F_EXP Funktionsblock stellt eine zuverlässige und normkonforme Implementierung der natürlichen Exponentialfunktion für industrielle Steuerungssysteme bereit. Durch seine einfache Schnittstelle und

flexible Datentypunterstützung eignet er sich ideal für verschiedene mathematische Anwendungen in Automatisierungsprojekten.

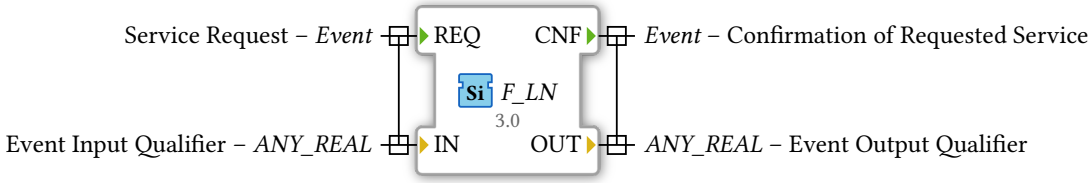
F_LN



• – ◦ • – ◦ • – ◦ •

Einleitung

Der F_LN Funktionsblock berechnet den natürlichen Logarithmus (Logarithmus zur Basis e) eines numerischen Eingangswerts. Er gehört zur Kategorie der Standardfunktionen für eine numerische Variable gemäß IEC 61131-3 Standard.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Service-Anfrage - löst die Berechnung des natürlichen Logarithmus aus

Ereignis-Ausgänge

- **CNF**: Bestätigung der angeforderten Serviceleistung - signalisiert den Abschluss der Berechnung

Daten-Eingänge

- **IN**: Numerischer Eingangswert vom Typ ANY_REAL, für den der natürliche Logarithmus berechnet werden soll

Daten-Ausgänge

- **OUT**: Berechneter natürlicher Logarithmus des Eingangswerts vom Typ ANY_REAL

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Bei Empfang eines REQ-Ereignisses berechnet der F_LN-Block den natürlichen Logarithmus (ln) des an IN übergebenen Wertes. Das Ergebnis wird an OUT ausgegeben und gleichzeitig wird das CNF-Ereignis generiert, um den erfolgreichen Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Unterstützt den ANY_REAL Datentyp, was verschiedene numerische Datentypen wie REAL, LREAL etc. umfasst
- Implementiert gemäß IEC 61131-3 Standard für numerische Funktionen
- Echtzeitfähige Ausführung bei Ereignissteuerung

Zustandsübersicht

Der Funktionsblock verfügt über einen einfachen Zustandsautomaten:

1. **Wartezustand**: Block wartet auf REQ-Ereignis
2. **Berechnungszustand**: Verarbeitet die Logarithmusberechnung
3. **Ausgabezustand**: Sendet CNF-Ereignis mit berechnetem Ergebnis

Anwendungsszenarien

- Wissenschaftliche Berechnungen und mathematische Modellierung
- Signalverarbeitung und Datenanalyse
- Steuerungstechnik bei exponentiellen Prozessen
- Skalierung von Messwerten in logarithmischen Darstellungen

⚖️ Vergleich mit ähnlichen Bausteinen

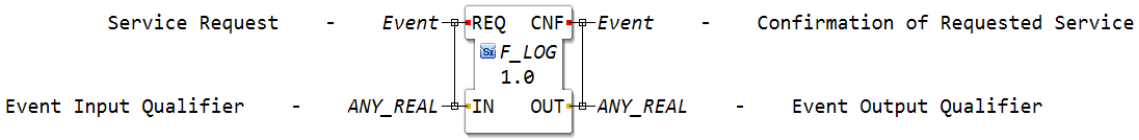
Im Vergleich zu anderen mathematischen Funktionsblöcken:

- F_LN spezialisiert auf natürliche Logarithmen (Basis e)
- F_LOG berechnet Logarithmen zu beliebiger Basis
- F_EXP berechnet die Exponentialfunktion (Umkehrfunktion zu LN)
- Einfacherer Aufbau als allgemeine Logarithmus-Funktionen

Fazit

Der F_LN Funktionsblock bietet eine effiziente und standardkonforme Implementierung zur Berechnung des natürlichen Logarithmus. Durch seine ereignisgesteuerte Architektur eignet er sich besonders für Echtzeitanwendungen, bei denen mathematische Operationen präzise und zuverlässig durchgeführt werden müssen.

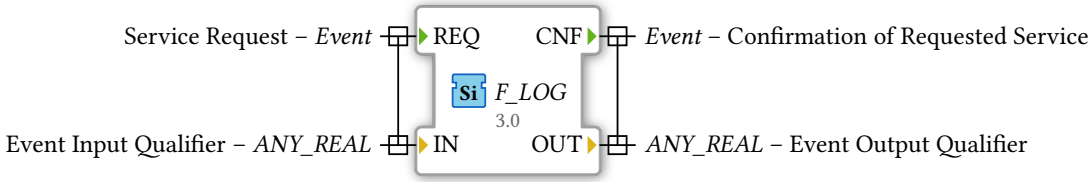
F_LOG



• — ○ • — ○ • — ○ •

Einleitung

Der F_LOG Funktionsblock berechnet den Logarithmus zur Basis 10 (dekadischer Logarithmus) eines numerischen Eingangswerts. Er gehört zur Kategorie der Standardfunktionen für eine numerische Variable gemäß IEC 61131-3 und ist Teil des Pakets `iec61131::numerical`.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Service-Anfrage - löst die Berechnung des Logarithmus aus

Ereignis-Ausgänge

- **CNF**: Bestätigung der angeforderten Serviceleistung - signalisiert die Beendigung der Berechnung

Daten-Eingänge

- **IN**: Numerischer Eingangswert vom Typ ANY_REAL, für den der Logarithmus berechnet werden soll

Daten-Ausgänge

- **OUT**: Berechneter Logarithmus-Wert vom Typ ANY_REAL

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Bei Eintreffen eines REQ-Ereignisses berechnet der F_LOG-Block den dekadischen Logarithmus ($\log_{10}$) des aktuellen IN-Werts. Das Ergebnis wird am OUT-Ausgang ausgegeben und gleichzeitig wird ein CNF-Ereignis generiert, um die erfolgreiche Berechnung zu signalisieren.

Technische Besonderheiten

- Unterstützt den ANY_REAL-Datentyp, was verschiedene numerische Datentypen wie REAL, LREAL etc. ermöglicht
- Einhaltung der IEC 61131-3 Standard-Spezifikation
- Einfache und deterministische Funktionsweise ohne interne Zustände

Zustandsübersicht

Der Funktionsblock besitzt keine komplexen internen Zustände. Er verhält sich rein funktional: Bei REQ-Ereignis ⟶ Berechnung ⟶ CNF-Ereignis.

Anwendungsszenarien

- Wissenschaftliche Berechnungen und Messwertverarbeitung
- Signalverarbeitung in der Automatisierungstechnik
- Skalierung von Messwerten in logarithmischen Darstellungen
- Audio- und Akustik-Anwendungen (z.B. dB-Berechnungen)

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen logarithmischen Funktionen:

- F_LN: Berechnet den natürlichen Logarithmus (Basis e)
- F_LOG: Berechnet spezifisch den dekadischen Logarithmus (Basis 10)
- Beide Blöcke folgen demselben Ereignis-/Datenfluss-Paradigma

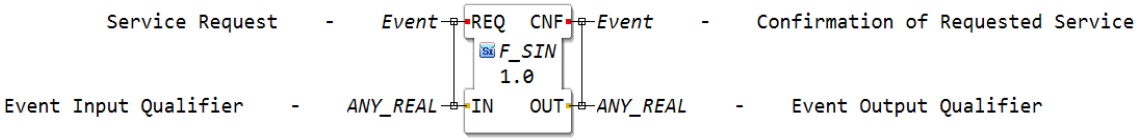
Fazit

Der F_LOG Funktionsblock bietet eine standardisierte, zuverlässige Implementierung für die Berechnung des dekadischen Logarithmus in 4diac-Steuerungssystemen. Seine einfache Schnittstelle und deterministische Funktionsweise machen ihn ideal für den Einsatz in Echtzeit-Automatisierungsanwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

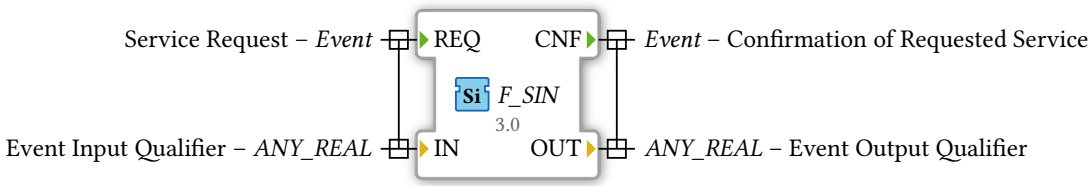
F_SIN



• – ◦ • – ◦ • – ◦ •

Einleitung

Der F_SIN Funktionsblock berechnet den Sinus eines Eingangswerts in Radiant. Es handelt sich um eine Standardfunktion für numerische Berechnungen gemäß IEC 61131-3, die mathematische Sinusoperationen in Steuerungsanwendungen ermöglicht.



Schnittstellenstruktur

Ereignis-Eingänge
- **REQ**: Service-Anfrage - löst die Berechnung des Sinus aus

Ereignis-Ausgänge
- **CNF**: Bestätigung der angeforderten Serviceleistung - signalisiert die Beendigung der Berechnung

Daten-Eingänge
- **IN**: Eingabewert in Radiant (Typ: ANY_REAL) - der Winkel, dessen Sinus berechnet werden soll

Daten-Ausgänge
- **OUT**: Ausgabewert (Typ: ANY_REAL) - das Ergebnis der Sinusberechnung

Adapter
Keine Adapter-Schnittstellen vorhanden.

Funktionsweise
Bei Empfang eines REQ-Ereignisses berechnet der Funktionsblock den Sinus des an IN übergebenen Wertes. Das Ergebnis wird an OUT ausgegeben und gleichzeitig wird ein CNF-Ereignis generiert, um die erfolgreiche Berechnung zu signalisieren. Die Berechnung erfolgt nach der mathematischen Standarddefinition der Sinusfunktion.

Technische Besonderheiten
- Unterstützt den ANY_REAL Datentyp für maximale Flexibilität bei numerischen Eingaben
- Einhaltung des IEC 61131-3 Standards für Programmierbare Steuerungen
- Einfache ereignisgesteuerte Verarbeitung
- Direkte Abbildung der mathematischen Sinusfunktion

Zustandsübergänge
Der Funktionsblock verfügt über zwei Hauptzustände:

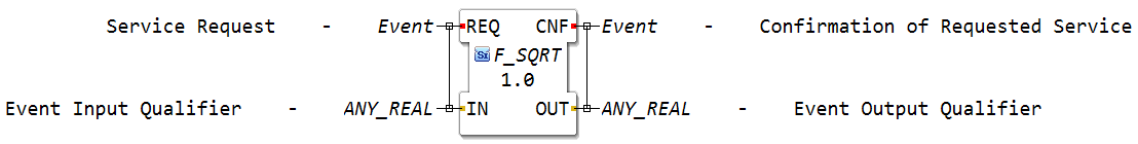
1. **Wartezustand**: Wartet auf REQ-Ereignis
2. **Berechnungszustand**: Verarbeitet die Sinusberechnung und generiert CNF-Ereignis

Anwendungsszenarien
- Signalverarbeitung in Regelungstechnik
- Berechnung von Schwingungen und periodischen Vorgängen
- Mathematische Transformationen in technischen Anwendungen
- Positionsberechnungen in Robotik und Motion Control

⚖️ Vergleich mit ähnlichen Bausteinen
Im Vergleich zu anderen trigonometrischen Funktionen wie F_COS (Kosinus) oder F_TAN (Tangens) bietet F_SIN die spezifische Sinusberechnung. Gegenüber universellen mathematischen Blöcken zeichnet sich F_SIN durch seine spezialisierte und optimierte Implementierung aus.

Fazit
Der F_SIN Funktionsblock stellt eine zuverlässige und standardkonforme Lösung für Sinusberechnungen in IEC 61131-3 basierten Steuerungssystemen dar. Seine einfache Schnittstelle und direkte Funktionalität machen ihn ideal für zahlreiche technische Anwendungen, die trigonometrische Berechnungen erfordern.

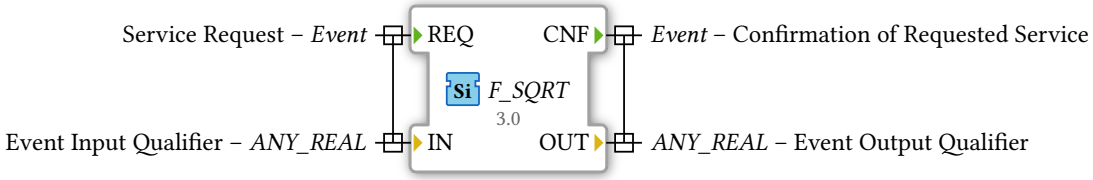
F_SQRT



• — ◦ • — ◦ • — ◦ •

Einleitung

Der F_SQRT Funktionsblock ist ein Standardfunktionsblock gemäß IEC 61131-3, der die Quadratwurzel einer numerischen Eingabe berechnet. Er gehört zur Kategorie der numerischen Funktionen und dient zur mathematischen Berechnung der Quadratwurzel eines beliebigen reellen Eingabewertes.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Service-Anfrage - löst die Berechnung der Quadratwurzel aus

Ereignis-Ausgänge

- **CNF**: Bestätigung der angeforderten Serviceleistung - signalisiert den Abschluss der Berechnung

Daten-Eingänge

- **IN** (ANY_REAL): Eingabewert, von dem die Quadratwurzel berechnet werden soll

Daten-Ausgänge

- **OUT** (ANY_REAL): Ergebnis der Quadratwurzelberechnung

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der F_SQRT Funktionsblock berechnet die Quadratwurzel des Eingabewertes IN. Bei Aktivierung durch das REQ-Ereignis wird die Berechnung durchgeführt. Nach Abschluss der Berechnung wird das CNF-Ereignis ausgelöst und das Ergebnis über die OUT-Variable ausgegeben.

Technische Besonderheiten

- Verwendet den ANY_REAL Datentyp für maximale Flexibilität mit verschiedenen realen Zahlentypen
- Einhaltung der IEC 61131-3 Standard-Spezifikation
- Einfache und direkte Funktionalität ohne zusätzliche Parameter

Zustandsübergänge

1. **Wartezustand**: Block wartet auf REQ-Ereignis
2. **Berechnungszustand**: Bei REQ-Ereignis wird die Quadratwurzel berechnet
3. **Ausgabezustand**: Nach Berechnung wird CNF-Ereignis ausgelöst und Ergebnis ausgegeben

Anwendungsszenarien

- Mathematische Berechnungen in Steuerungssystemen
- Signalverarbeitung und Filterung
- Wissenschaftliche und technische Anwendungen
- Regelungstechnik und Prozesssteuerung

⚖️ Vergleich mit ähnlichen Bausteinen

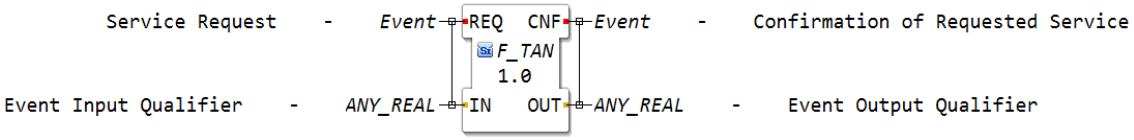
Im Vergleich zu anderen mathematischen Funktionsblöcken bietet F_SQRT eine spezialisierte Funktion für die Quadratwurzelberechnung. Andere Blöcke wie F_ADD oder F_MUL bieten grundlegende arithmetische Operationen, während F_SQRT eine spezifische mathematische Funktion bereitstellt.

Fazit

Der F_SQRT Funktionsblock ist ein essentielles Werkzeug für numerische Berechnungen in IEC 61131-3 basierten Steuerungssystemen. Seine einfache Schnittstelle und zuverlässige Funktionalität machen ihn

zu einer wertvollen Komponente für mathematische Operationen in industriellen Automatisierungslösungen.

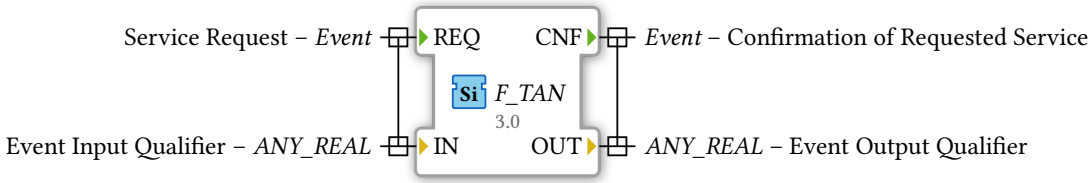
F_TAN



• – ◦ • – ◦ • – ◦ •

Einleitung

Der F_TAN Funktionsblock berechnet den Tangens eines Eingabewertes in Radiant. Es handelt sich um eine Standardfunktion für numerische Berechnungen gemäß IEC 61131-3.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Service-Anfrage - löst die Berechnung des Tangens aus

Ereignis-Ausgänge

- **CNF**: Bestätigung der angeforderten Serviceleistung - signalisiert die Beendigung der Berechnung

Daten-Eingänge

- **IN**: Eingabewert in Radiant (ANY_REAL Datentyp)

Daten-Ausgänge

- **OUT**: Berechneter Tangens-Wert (ANY_REAL Datentyp)

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Bei Empfang eines REQ-Ereignisses berechnet der F_TAN-Block den Tangens des an IN übergebenen Winkels in Radiant. Das Ergebnis wird an OUT ausgegeben und gleichzeitig wird das CNF-Ereignis generiert, um den Abschluss der Berechnung zu signalisieren.

Technische Besonderheiten

- Unterstützt den ANY_REAL Datentyp für maximale Flexibilität bei numerischen Eingaben
- Berechnung erfolgt in Radiant (nicht in Grad)
- Implementiert nach IEC 61131-3 Standard für numerische Funktionen
- Mathematische Besonderheit: Bei Winkeln nahe $\pi/2 + k\pi$ ($k \in \mathbb{Z}$) können sehr große Werte oder Überläufe auftreten

Zustandsübergang

1. **Wartezustand**: Block wartet auf REQ-Ereignis
2. **Berechnungszustand**: Bei REQ-Ereignis wird Tangens berechnet
3. **Ausgabezustand**: Ergebnis wird an OUT gesendet und CNF-Ereignis generiert
4. Rückkehr in Wartezustand

Anwendungsszenarien

- Trigonometrische Berechnungen in Steuerungssystemen

- Signalverarbeitung und Filterung
- Robotik und Bewegungssteuerung
- Mathematische Modellierung und Simulation
- Regelungstechnik und Positionsbestimmung

⚖️ Vergleich mit ähnlichen Bausteinen

- **F_SIN**: Berechnet Sinus statt Tangens
- **F_COS**: Berechnet Kosinus statt Tangens
- **F_ATAN**: Inverse Funktion (Arcustangens)
- **F_TAN** bietet spezifisch die Tangens-Funktion für trigonometrische Anwendungen

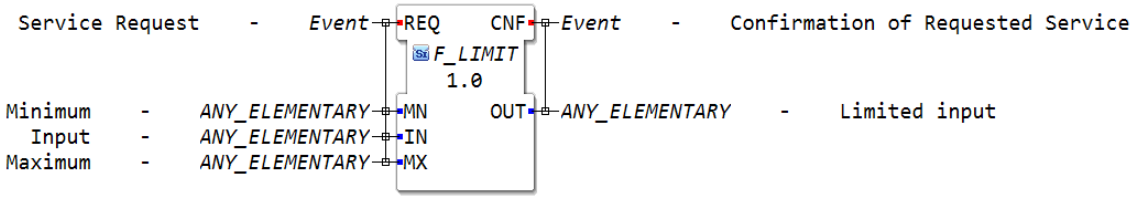
Fazit

Der F_TAN Funktionsblock ist ein essentielles Element für trigonometrische Berechnungen in IEC 61131-3 basierten Steuerungssystemen. Seine einfache und standardkonforme Schnittstelle ermöglicht eine zuverlässige Integration in komplexere Steuerungsalgorithmen, insbesondere in Anwendungen, die Winkelfunktionen erfordern.

2.4.1.11 Auswahlfunktionen

Hier finden Sie Funktionen zur Auswahl von Werten (Max, Min, Limit, Mux etc.).

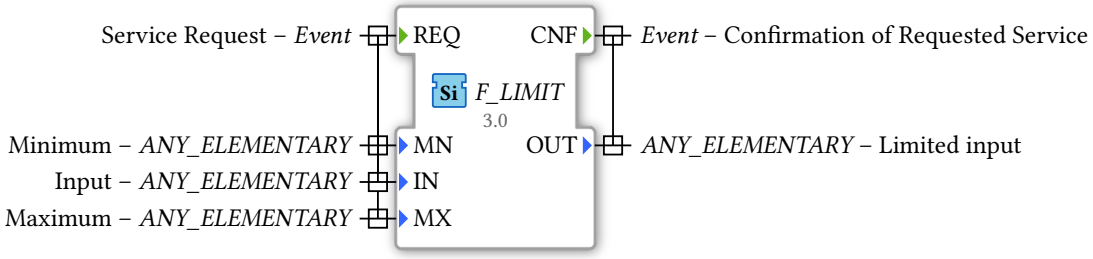
F_LIMIT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der F_LIMIT Funktionsblock ist ein Standard-Selektionsfunktionsblock gemäß IEC 61131-3, der als Begrenzer (Limiter) fungiert. Er begrenzt einen Eingangswert auf einen definierten Wertebereich zwischen einem unteren und oberen Grenzwert.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Service-Anfrage - löst die Begrenzungsoperation aus

Ereignis-Ausgänge

- **CNF**: Bestätigung der angeforderten Service-Operation

Daten-Eingänge

- **MN**: Minimum - unterer Grenzwert für die Begrenzung
- **IN**: Input - Eingangswert, der begrenzt werden soll
- **MX**: Maximum - oberer Grenzwert für die Begrenzung

Daten-Ausgänge

- **OUT**: Limited input - der begrenzte Ausgangswert

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der F_LIMIT Funktionsblock nimmt einen Eingangswert (IN) und begrenzt ihn auf den Bereich zwischen dem Minimum (MN) und Maximum (MX). Bei Aktivierung durch das REQ-Ereignis wird folgende Operation durchgeführt:

- Wenn IN < MN, dann OUT = MN
- Wenn IN > MX, dann OUT = MX
- Wenn MN ≤ IN ≤ MX, dann OUT = IN

Nach Abschluss der Operation wird das CNF-Ereignis mit dem berechneten OUT-Wert ausgegeben.

Technische Besonderheiten

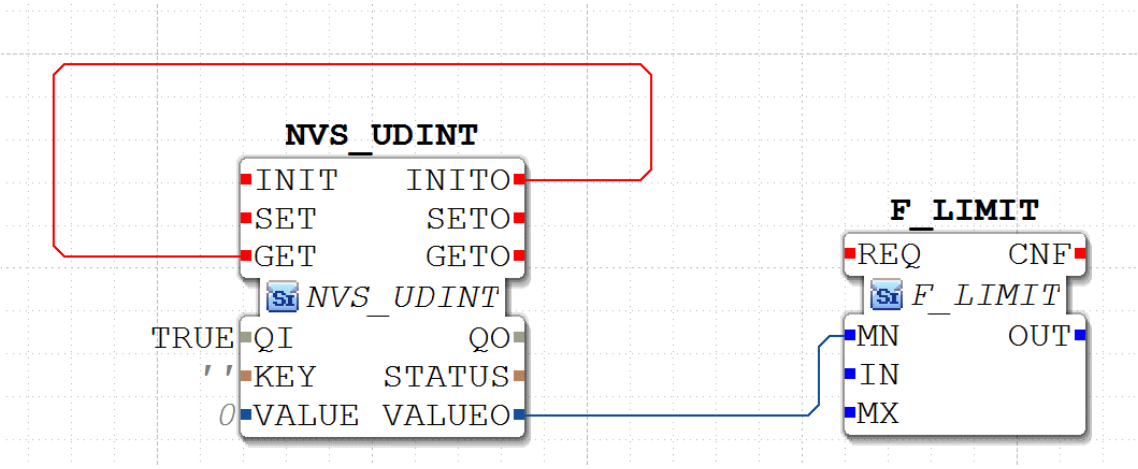
- Verwendet den generischen Datentyp ANY_ELEMENTARY, was bedeutet, dass der Block mit verschiedenen elementaren Datentypen (INT, REAL, etc.) verwendet werden kann
- Alle Daten-Eingänge und der Daten-Ausgang verwenden denselben Datentyp für konsistente Operation
- EPL-2.0 Lizenz (Eclipse Public License)

Zustandsübergang

1. **Wartezustand**: Block wartet auf REQ-Ereignis
2. **Aktiv**: Bei REQ-Ereignis wird die Begrenzungsoperation durchgeführt
3. **Ausgabe**: Nach Berechnung wird CNF-Ereignis mit OUT-Wert ausgegeben
4. **Rückkehr**: Rückkehr in Wartezustand

Anwendungsszenarien

- Signalbegrenzung in Regelungstechnik
- Schutz vor Über- und Unterschreitung von Grenzwerten
- Normierung von Messwerten auf einen zulässigen Bereich
- Sicherheitsbegrenzung in Prozesssteuerungen



⚖️ Vergleich mit ähnlichen Bausteinen

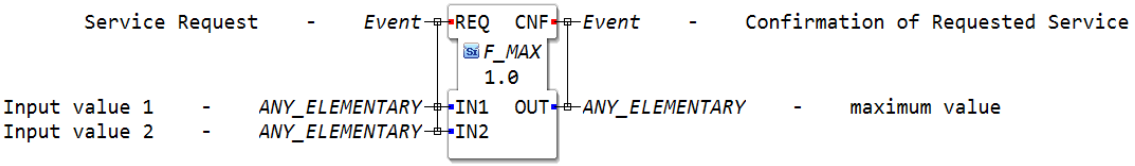
Im Vergleich zu anderen Selektionsfunktionsblöcken:

- F_LIMIT speziell für Wertebegrenzung optimiert
- Einfacher als komplexe Vergleichsblöcke mit mehreren Ausgängen
- Universeller einsetzbar durch ANY_ELEMENTARY Datentyp

Fazit

Der F_LIMIT Funktionsblock bietet eine einfache und effiziente Lösung zur Wertebegrenzung in IEC 61131-3 basierten Steuerungssystemen. Seine generische Typunterstützung macht ihn flexibel einsetzbar für verschiedene Anwendungsfälle, bei denen Eingangswerte auf einen definierten Bereich beschränkt werden müssen.

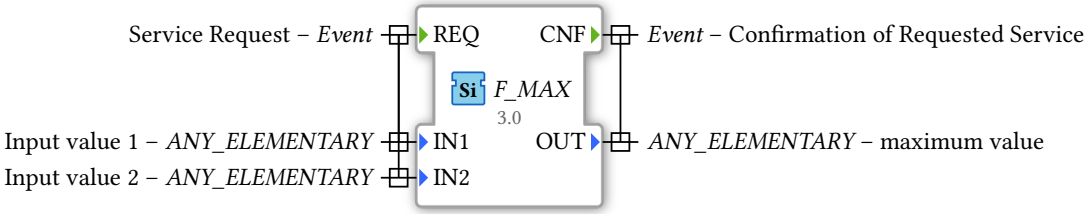
F_MAX



• – ◦ • – ◦ • – ◦ •

Einleitung

Der F_MAX Funktionsblock ist ein Standard-Selektionsfunktionsblock gemäß IEC 61131-3, der den maximalen Wert aus zwei Eingangswerten ermittelt. Der Baustein vergleicht zwei beliebige elementare Datentypen und gibt den größeren der beiden Werte als Ergebnis zurück.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Service-Anfrage - löst die Berechnung des Maximums aus

Ereignis-Ausgänge

- **CNF**: Bestätigung der angeforderten Serviceleistung - signalisiert die Fertigstellung der Berechnung

Daten-Eingänge

- **IN1**: Eingabewert 1 (beliebiger elementarer Datentyp)
- **IN2**: Eingabewert 2 (beliebiger elementarer Datentyp)

Daten-Ausgänge

- **OUT**: Maximalwert (gleicher Datentyp wie die Eingänge)

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Bei Eintreffen eines REQ-Ereignisses vergleicht der F_MAX-Block die beiden Eingangswerte IN1 und IN2. Der größere der beiden Werte wird am Ausgang OUT ausgegeben. Gleichzeitig wird das CNF-Ereignis generiert, um den Abschluss der Operation zu signalisieren. Der Block unterstützt alle elementaren Datentypen (ANY_ELEMENTARY), wodurch er universell für verschiedene numerische und nicht-numerische Vergleiche einsetzbar ist.

Technische Besonderheiten

- Verwendung des generischen Datentyps ANY_ELEMENTARY für maximale Flexibilität
- Automatische Typanpassung zwischen Eingängen und Ausgang
- Echtzeitfähige Ausführung
- Standardkonforme Implementierung nach IEC 61131-3

Zustandsübergänge

1. **Wartezustand**: Block wartet auf REQ-Ereignis
2. **Berechnungszustand**: Bei REQ-Ereignis werden IN1 und IN2 verglichen
3. **Ausgabezustand**: Maximum wird an OUT ausgegeben und CNF-Ereignis generiert
4. Rückkehr in Wartezustand

Anwendungsszenarien

- Ermittlung des Maximums in Regelungs- und Steuerungsalgorithmen
- Grenzwertüberwachung in Prozessleitsystemen
- Signalverarbeitung und Filterung
- Qualitätssicherung bei Messwerten
- Priorisierungslogik in Steuerungen

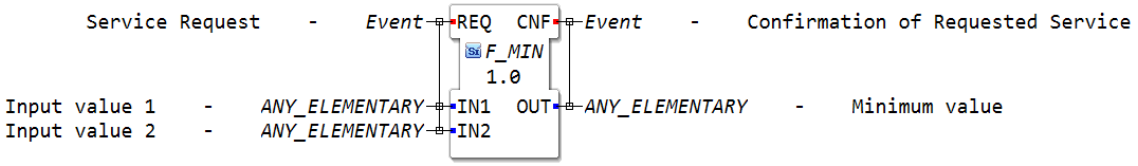
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu spezifischen Maximum-Blöcken (wie MAX_INT, MAX_REAL etc.) bietet F_MAX den Vorteil der Typunabhängigkeit. Während spezifische Blöcke nur einen Datentyp verarbeiten können, ist F_MAX universell für alle elementaren Datentypen einsetzbar.

Fazit

Der F_MAX Funktionsblock stellt eine flexible und standardkonforme Lösung zur Ermittlung von Maximumwerten in Automatisierungssystemen dar. Seine Typunabhängigkeit macht ihn besonders vielseitig einsetzbar, während die einfache Schnittstellenstruktur eine intuitive Integration in verschiedene Applikationen ermöglicht.

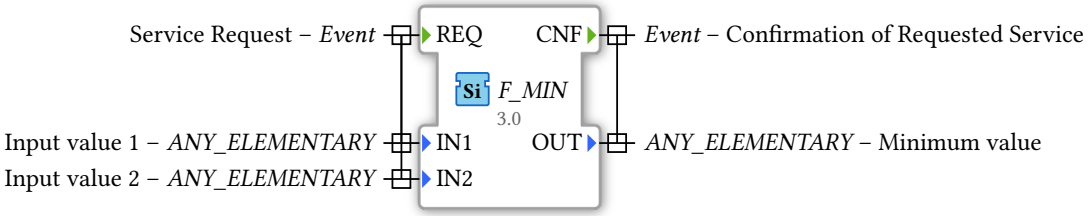
F_MIN



• – ○ • – ○ • – ○ •

Einleitung

Der F_MIN Funktionsblock ist ein Standard-Selektionsfunktionsblock gemäß IEC 61131-3, der das Minimum zweier Eingangswerte ermittelt. Der Baustein vergleicht zwei beliebige elementare Datentypen und gibt den kleineren der beiden Werte als Ergebnis zurück.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Service-Anfrage - löst die Berechnung des Minimums aus

Ereignis-Ausgänge

- **CNF**: Bestätigung der angeforderten Serviceleistung - signalisiert die Beendigung der Berechnung

Daten-Eingänge

- **IN1**: Eingabewert 1 (beliebiger elementarer Datentyp)
- **IN2**: Eingabewert 2 (beliebiger elementarer Datentyp)

Daten-Ausgänge

- **OUT**: Minimalwert (gleicher Datentyp wie die Eingänge)

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Bei Aktivierung des REQ-Ereignisses vergleicht der F_MIN-Block die beiden Eingangswerte IN1 und IN2. Der kleinere der beiden Werte wird am Ausgang OUT ausgegeben. Gleichzeitig wird das CNF-Ereignis generiert, um den Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Unterstützt beliebige elementare Datentypen (ANY_ELEMENTARY)
- Typsicherheit: Ausgang hat denselben Datentyp wie die Eingänge
- Echtzeitfähige Ausführung
- Einfache und deterministische Funktionsweise

Zustandsübergänge

1. **Wartezustand**: Block wartet auf REQ-Ereignis
2. **Berechnungszustand**: Bei REQ-Ereignis werden IN1 und IN2 verglichen
3. **Ausgabezustand**: Minimalwert wird an OUT ausgegeben und CNF-Ereignis generiert
4. Rückkehr in Wartezustand

Anwendungsszenarien

- Grenzwertüberwachung in Regelungstechnik
- Minimum-Selektion in Datenverarbeitungsketten
- Sicherheitsrelevante Begrenzungen
- Signalverarbeitung mit dynamischen Eingangswerten

⚖️ Vergleich mit ähnlichen Bausteinen

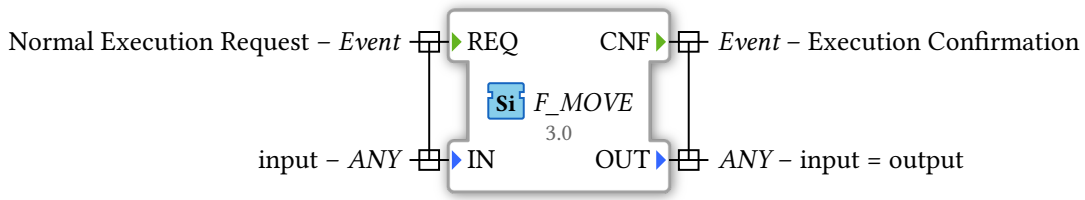
Im Vergleich zu F_MAX (Maximum-Funktion) ermittelt F_MIN den kleineren statt den größeren Wert. Beide Blöcke teilen die gleiche Schnittstellenstruktur und arbeiten mit beliebigen elementaren Datentypen.

Fazit

Der F_MIN-Block stellt eine einfache, zuverlässige und standardkonforme Lösung zur Minimum-Bestimmung in IEC 61131-3-basierten Steuerungssystemen dar. Seine Flexibilität bezüglich der unterstützten Datentypen macht ihn zu einem universell einsetzbaren Baustein in verschiedenen Automatisierungsanwendungen.

F_MOVE

• – ◦ • – ◦ • – ◦ •



Einführung

Der **F_MOVE** ist ein universeller Funktionsbaustein zur typsicheren Datenübertragung, entwickelt unter EPL-2.0 Lizenz. Version 1.0 ermöglicht die Übertragung beliebiger Datentypen (ANY) zwischen Variablen in Steuerungsanwendungen.

Schnittstellenstruktur

Ereignis-Eingänge
- `REQ`: Übertragungsanforderung (mit Eingabewert)

Ereignis-Ausgänge
- `CNF`: Übertragungsbestätigung (mit Ausgabewert)

Daten-Eingänge
- `IN` (ANY): Quelldaten (beliebiger Typ)

Daten-Ausgänge
- `OUT` (ANY): Zieldaten (identischer Typ wie IN)

Funktionsweise
1. **Aktivierung**:
   - Durch `REQ`-Ereignis ausgelöst
   - Verarbeitet beliebige Datentypen (ANY)
2. **Datenübertragung**:
   - 1:1 Kopie des Eingabewerts
   - Typerhaltende Übertragung
   - Unterstützt alle IEC 61131-3 Datentypen
3. **Bestätigung**:
   - `CNF`-Ereignis nach erfolgreicher Übertragung
   - OUT erhält exakte Kopie von IN

Technische Spezifikationen
✓ Generische Typunterstützung (ANY) **✓ Bitgenaue Datenkopie ✓ Einzyklus-Latenzzeit ✓ IEC 61131-3 konform**

Anwendungsbeispiele
- **Datenkonvertierung**: Typumwandlungen
- **Signalweiterleitung**: Prozessdatenverteilung
- **Pufferoperationen**: Zwischenspeicherung
- **Schnittstellen**: Datenkapselung

Besondere Hinweise
- **Typkompatibilität**:
  - Eingangs- und Ausgangstyp müssen identisch sein
  - Keine automatische Typkonvertierung
- **Performance**:
  - Konstant schnelle Ausführung
  - Unabhängig vom Datentyp

⚖️ Vergleich mit ähnlichen Bausteinen

Merkmal	F_MOVE	F_BLOCK_MOVE	F_CAST
Funktionsumfang	Einzelwert	Speicherblock	Typkonvertierung
Typunterstützung	ANY	BYTE-Array	ANY
Latenz	1 Zyklus	Variabel	1 Zyklus

🛠️ Zugehörige Übungen
- Uebung_002b2

- Uebung_103c
- Uebung_110
- Uebung_111
- Uebung_171

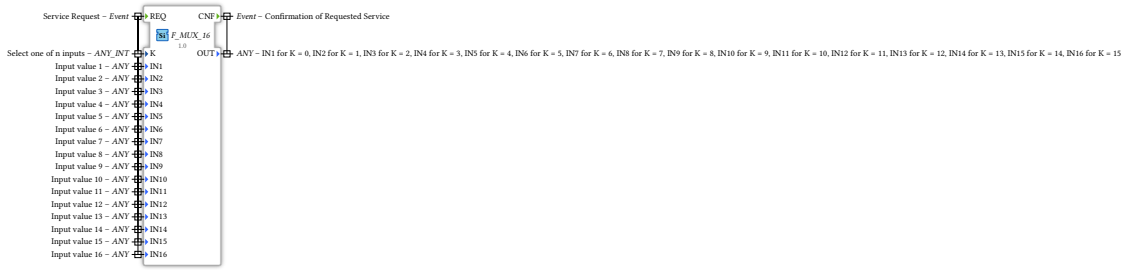
Fazit

Der F_MOVE-Baustein bietet essentielle Datenübertragungsfunktionen:

- Universelle Typunterstützung
- Zuverlässige 1:1-Datenkopie
- Minimaler Ressourcenverbrauch

Unverzichtbar für Steuerungsanwendungen mit komplexen Datenstrukturen oder typsensiblen Operationen. Die generische Implementierung als GEN_FORTE_F_MOVE ermöglicht maximale Flexibilität in heterogenen Automatisierungsumgebungen.

F_MUX_16



• — ○ • — ○ • — ○ •

Einleitung

Der `F_MUX_16` ist ein Multiplexer nach IEC 61131-3 (Standard-Auswahlfunktion, generischer Baustein), der anhand des Selektors `K` einen von **16 Eingängen** auf den Ausgang `OUT` durchschaltet. Er ist die 16-Kanal-Variante von `F_MUX_8` und `F_MUX_4`.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Serviceanforderung. Verbundene Daten: `IN1` bis `IN16`, `K`

Ereignis-Ausgänge

- **CNF**: Bestätigung des angeforderten Dienstes. Verbundene Daten: `OUT`

Daten-Eingänge

- **K** (`ANY_INT`): Wählt einen der Eingänge aus
- **IN1** bis **IN16** (`ANY`): Eingangswerte 1 bis 16

Daten-Ausgänge

- **OUT** (`ANY`): Gewählter Eingangswert (siehe Tabelle)

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Bei `REQ` wird der durch `K` gewählte Eingang an `OUT` ausgegeben, anschließend wird `CNF` ausgelöst. `K` zählt ab 0:

K	OUT	K	OUT
0	`IN1`	8	`IN9`
1	`IN2`	9	`IN10`

2	IN3	10	IN11
3	IN4	11	IN12
4	IN5	12	IN13
5	IN6	13	IN14
6	IN7	14	IN15
7	IN8	15	IN16

Technische Besonderheiten

- Generischer Baustein (`GEN_F_MUX`, FORTE); die Datentypen der Eingänge sind `ANY` und müssen zu `OUT` passen.
- Gehört zum Paket `iec61131::selection`.

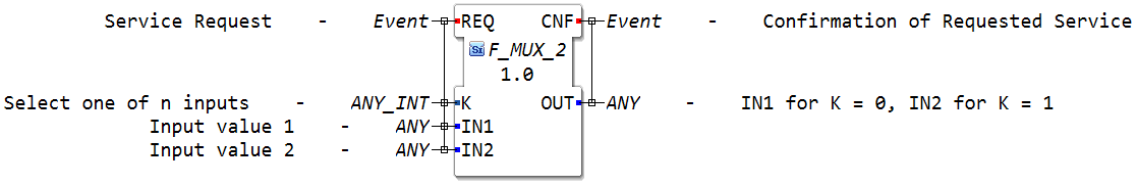
Anwendungsszenarien

- Auswahl eines von bis zu 16 Werten, z. B. Konstanten, Sollwerte oder Objekt-IDs, über einen Index

Fazit

`F_MUX_16` schaltet anhand des Index `K` (0 bis 15) einen von 16 Eingangswerten auf den Ausgang.

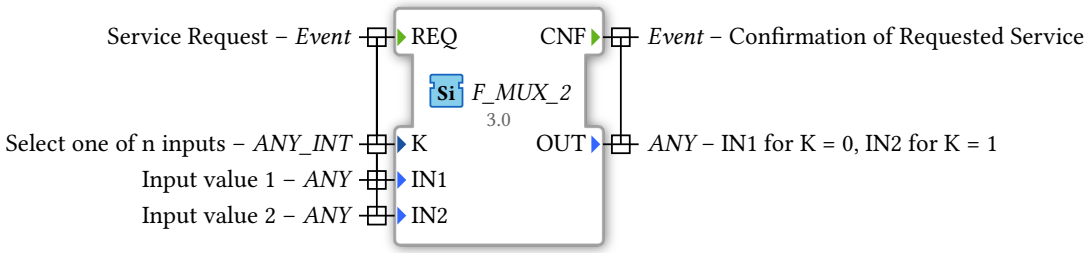
F_MUX_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der F_MUX_2 ist ein Multiplexer-Funktionsblock nach IEC 61131-3, der als Standard-Selektionsfunktion klassifiziert ist. Der Baustein ermöglicht die Auswahl zwischen zwei Eingangssignalen basierend auf einem Steuersignal.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Service-Anfrage - löst die Verarbeitung des Multiplexers aus

Ereignis-Ausgänge

- **CNF**: Bestätigung der angeforderten Serviceleistung - signalisiert die Beendigung der Verarbeitung

Daten-Eingänge

- **K**: Steuereingang zur Auswahl eines von n Eingängen (ANY_INT Typ)
- **IN1**: Eingangswert 1 (ANY Typ)
- **IN2**: Eingangswert 2 (ANY Typ)

Daten-Ausgänge

- **OUT**: Ausgangswert - liefert IN1 für K = 0, IN2 für K = 1 (ANY Typ)

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der F_MUX_2 arbeitet als 2:1 Multiplexer. Bei Eintreffen eines REQ-Ereignisses wertet der Baustein den Steuereingang K aus:

- Bei K = 0 wird der Wert von IN1 an OUT weitergegeben
- Bei K = 1 wird der Wert von IN2 an OUT weitergegeben

Nach erfolgreicher Verarbeitung wird das CNF-Ereignis ausgelöst, um den Abschluss der Operation zu signalisieren.

Technische Besonderheiten

- Verwendung von generischen Datentypen (ANY und ANY_INT) für maximale Flexibilität
- Unterstützung verschiedener Datentypen für die Eingangswerte
- Einfache und deterministische Verarbeitung
- EPL-2.0 Lizenz

Zustandsübergänge

1. **Wartezeit**: Keine aktive Verarbeitung
2. **Verarbeitung**: Bei REQ-Ereignis - Auswertung von K und Auswahl des entsprechenden Eingangs
3. **Abschluss**: Auslösung von CNF nach erfolgreicher Datenweitergabe

Anwendungsszenarien

- Signalweiche in Steuerungsanwendungen
- Auswahl zwischen verschiedenen Betriebsmodi
- Umschaltung zwischen redundanten Signalquellen
- Verarbeitungspfad-Auswahl in komplexen Steuerungen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Multiplexern bietet F_MUX_2:

- Spezifisch für 2 Eingänge optimiert
- Einfacherer Aufbau als Multiplexer mit mehr Eingängen
- Direkte und klare Steuerlogik
- Geringerer Ressourcenverbrauch bei 2-Wege-Auswahl

🛠️ Zugehörige Übungen

- Uebung_090a1

Fazit

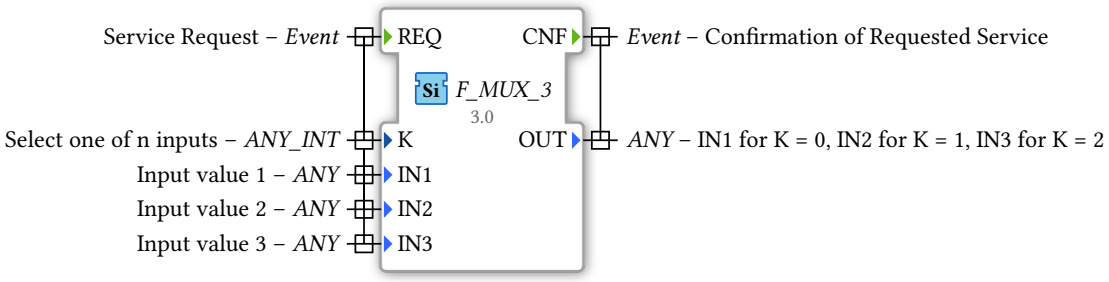
Der F_MUX_2 ist ein effizienter und zuverlässiger Multiplexer für Anwendungen, die eine Auswahl zwischen zwei Eingangssignalen erfordern. Seine einfache Struktur und die Verwendung generischer Datentypen machen ihn zu einer vielseitigen Lösung für verschiedene Steuerungsaufgaben.

F_MUX_3

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_MUX_3` ist ein Multiplexer, der drei Eingangssignale basierend auf einem Steuersignal auswählt und an den Ausgang weiterleitet. Er ist Teil der IEC 61131-3 Standardbibliothek und dient zur Signalauswahl in Automatisierungsanwendungen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Service Request – löst die Auswahl und Weiterleitung des entsprechenden Eingangssignals aus. Verknüpft mit den Daten-Eingängen `IN1`, `IN2`, `IN3` und `K`.

Ereignis-Ausgänge

- **CNF**: Confirmation of Requested Service – bestätigt die erfolgreiche Auswahl und Weiterleitung des Signals. Verknüpft mit dem Daten-Ausgang `OUT`.

Daten-Eingänge

- **K** (ANY_INT): Steuersignal, das angibt, welcher Eingang ausgewählt wird (0 für `IN1`, 1 für `IN2`, 2 für `IN3`).
- **IN1** (ANY): Eingangswert 1.
- **IN2** (ANY): Eingangswert 2.
- **IN3** (ANY): Eingangswert 3.

Daten-Ausgänge

- **OUT** (ANY): Ausgangswert, der dem ausgewählten Eingang entspricht (`IN1` für `K = 0`, `IN2` für `K = 1`, `IN3` für `K = 2`).

Adapter

Keine Adapter vorhanden.

Funktionsweise

Bei einem `REQ`-Ereignis wertet der Funktionsblock den Wert von `K` aus und leitet den entsprechenden Eingang (`IN1`, `IN2` oder `IN3`) an den Ausgang `OUT` weiter. Anschließend wird das `CNF`-Ereignis ausgelöst, um die erfolgreiche Operation zu bestätigen.

Technische Besonderheiten

- Unterstützt beliebige Datentypen (`ANY`) für die Eingänge und den Ausgang.
- Der Steuereingang `K` muss ein ganzzahliger Typ (`ANY_INT`) sein.
- Initialwerte für die Eingänge sind nicht vordefiniert.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Auswahl und Weiterleitung erfolgt unmittelbar bei einem `REQ`-Ereignis.

Anwendungsszenarien

- Signalweiterleitung in Steuerungsanwendungen.
- Auswahl zwischen verschiedenen Sensordaten.
- Dynamische Konfiguration von Signalpfaden.

⚖️ Vergleich mit ähnlichen Bausteinen

- `F_MUX_2`: Ein einfacherer Multiplexer mit nur zwei Eingängen.
- `F_SEL`: Ein allgemeinerer Selektor, der möglicherweise komplexere Auswahlkriterien unterstützt.

🛠️ Zugehörige Übungen

- Uebung_090a2

Fazit

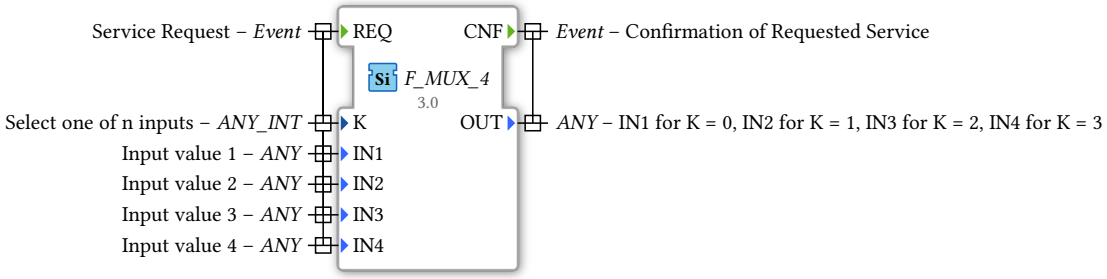
Der `F_MUX_3` ist ein einfacher und effizienter Multiplexer für Anwendungen, bei denen zwischen drei Signalen ausgewählt werden muss. Seine Flexibilität in Bezug auf Datentypen macht ihn vielseitig einsetzbar.

F_MUX_4

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_MUX_4` ist ein Multiplexer mit vier Eingängen, der einen von vier Eingangswerten basierend auf einem Steuersignal auswählt und am Ausgang ausgibt. Er ist Teil der IEC 61131-3 Standardbibliothek und wird für Auswahloperationen in Steuerungsanwendungen verwendet.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Dienst-Anforderung. Löst die Auswahl und Ausgabe des entsprechenden Eingangswerts aus. Wird mit den Daten-Eingängen `IN1`, `IN2`, `IN3`, `IN4` und `K` verknüpft.

Ereignis-Ausgänge

- `CNF`: Bestätigung der angeforderten Dienstleistung. Wird ausgelöst, nachdem der Ausgangswert gesetzt wurde. Wird mit dem Daten-Ausgang `OUT` verknüpft.

Daten-Eingänge

- `K` (`ANY_INT`): Steuersignal, das den auszuwählenden Eingang bestimmt.
  - `K = 0`: Wählt `IN1`
  - `K = 1`: Wählt `IN2`
  - `K = 2`: Wählt `IN3`
  - `K = 3`: Wählt `IN4`
- `IN1` (`ANY`): Eingangswert 1.
- `IN2` (`ANY`): Eingangswert 2.
- `IN3` (`ANY`): Eingangswert 3.
- `IN4` (`ANY`): Eingangswert 4.

Daten-Ausgänge

- `OUT` (`ANY`): Ausgangswert, der dem durch `K` ausgewählten Eingang entspricht.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Bei Empfang des Ereignisses `REQ` wertet der Funktionsblock den Wert von `K` aus und gibt den entsprechenden Eingangswert (`IN1` bis `IN4`) am Ausgang `OUT` aus. Anschließend wird das Ereignis `CNF` ausgelöst, um die erfolgreiche Auswahl und Ausgabe zu bestätigen.

Technische Besonderheiten

- Unterstützt beliebige Datentypen (`ANY`) für die Eingänge und den Ausgang.
- Der Steuereingang `K` muss ein ganzzahliger Wert sein (`ANY_INT`).
- Die Initialwerte der Eingänge sind leer, es werden keine Standardwerte vorgegeben.

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf das Ereignis `REQ`.
2. **Auswahlzustand**: Wertet `K` aus und wählt den entsprechenden Eingang aus.
3. **Ausgabezustand**: Setzt `OUT` auf den ausgewählten Wert und löst `CNF` aus.

Anwendungsszenarien

- Auswahl zwischen verschiedenen Sensordaten basierend auf einer Steuerlogik.
- Umschaltung zwischen verschiedenen Betriebsmodi in einer Steuerung.
- Dynamische Auswahl von Datenquellen in Abhängigkeit von externen Bedingungen.

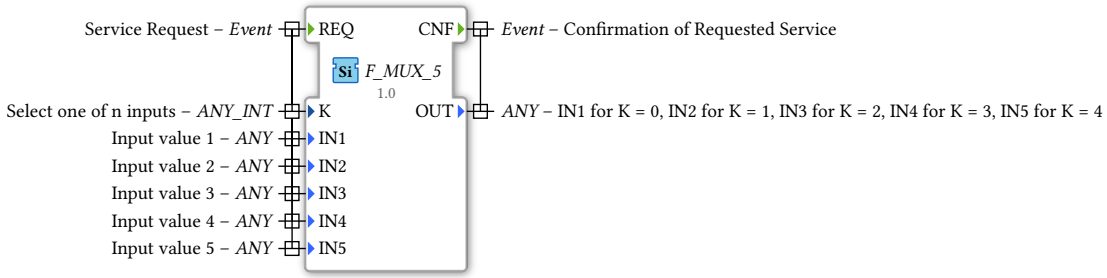
⚖️ Vergleich mit ähnlichen Bausteinen

- `F_MUX_2`: Einfacher Multiplexer mit nur zwei Eingängen.
- `F_MUX_3`: Multiplexer mit drei Eingängen.
- `F_MUX_4` bietet eine Erweiterung auf vier Eingänge, was mehr Flexibilität bei der Auswahl ermöglicht.

Fazit

Der `F_MUX_4` ist ein vielseitiger und einfach zu verwendender Multiplexer, der sich ideal für Anwendungen eignet, bei denen zwischen vier verschiedenen Eingangswerten ausgewählt werden muss. Seine Unterstützung beliebiger Datentypen und die klare Ereignissteuerung machen ihn zu einem wertvollen Baustein in der IEC 61131-3 Standardbibliothek.

F_MUX_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `F_MUX_5` ist die generische 5-Eingangs-Variante des Standard-Multiplexer-Bausteins `F_MUX` (vgl. `F_MUX_2`/`F_MUX_3`/`F_MUX_4`). Er wählt anhand des Selektors `K` einen von 5 Eingangswerten (`IN1`, `IN2`, `IN3`, `IN4`, `IN5`) aus und gibt ihn über `OUT` aus.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Auswahl aus, trägt `K` und `IN1`, `IN2`, `IN3`, `IN4`, `IN5`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT`.

Daten-Eingänge

- **K** (ANY_INT): Wählt einen der 5 Eingänge aus (`0`-basiert).
- `IN1`, `IN2`, `IN3`, `IN4`, `IN5` (ANY): Die 5 zur Auswahl stehenden Eingangswerte.

Daten-Ausgänge

- **OUT** (ANY): Der ausgewählte Eingangswert: `IN1` bei `K = 0`, `IN2` bei `K = 1`, `IN3` bei `K = 2`, `IN4` bei `K = 3`, `IN5` bei `K = 4`.

Funktionsweise

Bei Eintreffen von `REQ` wird abhängig vom Wert von `K` der entsprechende Eingang (`IN1`, `IN2`, `IN3`, `IN4`, `IN5`) auf `OUT` durchgeschaltet: `IN1` bei `K = 0`, `IN2` bei `K = 1`, `IN3` bei `K = 2`, `IN4` bei `K = 3`, `IN5` bei `K = 4`. Anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_F_MUX'`, dieselbe C++-Basis wie `F_MUX_2`/`F_MUX_3`/`F_MUX_4`; nur die Anzahl der Eingänge unterscheidet sich pro Instanziierung.
- **`ANY`-Datentyp**: Sowohl die Eingänge `IN1`…`IN5` als auch `OUT` sind generisch typisiert und passen sich beim Verdrahten an den angeschlossenen Datentyp an.
- **`0`-basierter Selektor**: `K = 0` wählt `IN1`, `K = 4` wählt `IN5`.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur Auswahl und zu `CNF`.

Anwendungsszenarien

- **Mehrfachauswahl mit mehr als 4 Quellen**: Überall dort, wo zwischen mehr als den von `F_MUX_4` unterstützten 4 Eingängen ausgewählt werden muss, ohne mehrere `F_MUX`-Bausteine zu kaskadieren.
- **Signalumschaltung**: Auswahl eines von mehreren Sensor- oder Quellsignalen anhand eines Steuerwerts `K`.

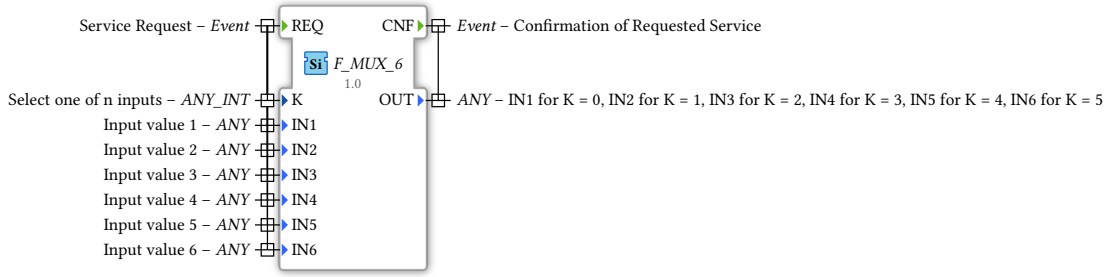
⚖️ Vergleich mit ähnlichen Bausteinen

- **`F_MUX_4`**: dieselbe generische Implementierung mit 4 statt 5 Eingängen.
- **`F_SEL`**: die binäre Auswahl zwischen genau zwei Werten anhand eines `BOOL`-Selektors, funktional ein Spezialfall von `F_MUX_2`.

Fazit

`F_MUX_5` erweitert das Multiplexer-Prinzip von `F_MUX_2`/`F_MUX_3`/`F_MUX_4` auf 5 wählbare Eingänge und ist die generische, für 5 Eingänge instanziierte Variante derselben zugrunde liegenden Implementierung.

F_MUX_6



• – ○ • – ○ • – ○ •

Einleitung

Der `F_MUX_6` ist die generische 6-Eingangs-Variante des Standard-Multiplexer-Bausteins `F_MUX` (vgl. `F_MUX_2`/`F_MUX_3`/`F_MUX_4`). Er wählt anhand des Selektors `K` einen von 6 Eingangswerten (`IN1`, `IN2`, `IN3`, `IN4`, `IN5`, `IN6`) aus und gibt ihn über `OUT` aus.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Auswahl aus, trägt `K` und `IN1`, `IN2`, `IN3`, `IN4`, `IN5`, `IN6`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT`.

Daten-Eingänge

- **K** (ANY_INT): Wählt einen der 6 Eingänge aus (`0`-basiert).
- `IN1`, `IN2`, `IN3`, `IN4`, `IN5`, `IN6` (ANY): Die 6 zur Auswahl stehenden Eingangswerte.

Daten-Ausgänge

- **OUT** (ANY): Der ausgewählte Eingangswert: `IN1` bei `K = 0`, `IN2` bei `K = 1`, `IN3` bei `K = 2`, `IN4` bei `K = 3`, `IN5` bei `K = 4`, `IN6` bei `K = 5`.

Funktionsweise

Bei Eintreffen von `REQ` wird abhängig vom Wert von `K` der entsprechende Eingang (`IN1`, `IN2`, `IN3`, `IN4`, `IN5`, `IN6`) auf `OUT` durchgeschaltet: `IN1` bei `K = 0`, `IN2` bei `K = 1`, `IN3` bei `K = 2`, `IN4` bei `K = 3`, `IN5` bei `K = 4`, `IN6` bei `K = 5`. Anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_F_MUX'`, dieselbe C++-Basis wie `F_MUX_2`/`F_MUX_3`/`F_MUX_4`; nur die Anzahl der Eingänge unterscheidet sich pro Instanziierung.
- **`ANY`-Datentyp**: Sowohl die Eingänge `IN1`…`IN6` als auch `OUT` sind generisch typisiert und passen sich beim Verdrahten an den angeschlossenen Datentyp an.
- **`0`-basierter Selektor**: `K = 0` wählt `IN1`, `K = 5` wählt `IN6`.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur Auswahl und zu `CNF`.

Anwendungsszenarien

- **Mehrfachauswahl mit mehr als 4 Quellen**: Überall dort, wo zwischen mehr als den von `F_MUX_4` unterstützten 4 Eingängen ausgewählt werden muss, ohne mehrere `F_MUX`-Bausteine zu kaskadieren.
- **Signalumschaltung**: Auswahl eines von mehreren Sensor- oder Quellsignalen anhand eines Steuerwerts `K`.

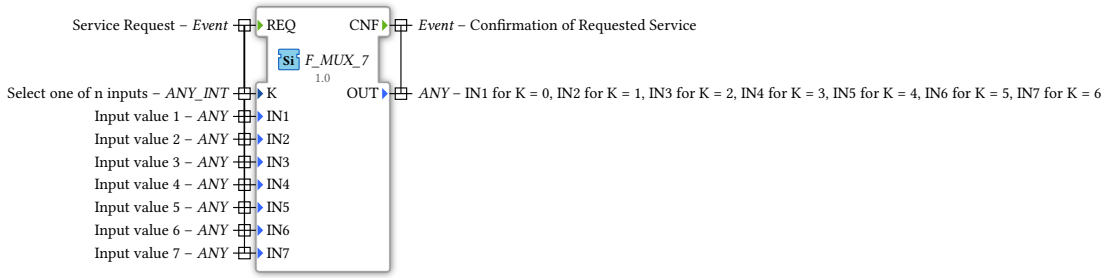
⚖️ Vergleich mit ähnlichen Bausteinen

- **`F_MUX_5`**: dieselbe generische Implementierung mit 5 statt 6 Eingängen.
- **`F_SEL`**: die binäre Auswahl zwischen genau zwei Werten anhand eines `BOOL`-Selektors, funktional ein Spezialfall von `F_MUX_2`.

Fazit

`F_MUX_6` erweitert das Multiplexer-Prinzip von `F_MUX_2`/`F_MUX_3`/`F_MUX_4` auf 6 wählbare Eingänge und ist die generische, für 6 Eingänge instanziierte Variante derselben zugrunde liegenden Implementierung.

F_MUX_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der F_MUX_7 ist die generische 7-Eingangs-Variante des Standard-Multiplexer-Bausteins F_MUX (vgl. F_MUX_2/F_MUX_3/F_MUX_4). Er wählt anhand des Selektors K einen von 7 Eingangswerten (IN1, IN2, IN3, IN4, IN5, IN6, IN7) aus und gibt ihn über OUT aus.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Auswahl aus, trägt K und IN1, IN2, IN3, IN4, IN5, IN6, IN7.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt OUT.

Daten-Eingänge

- **K** (ANY_INT): Wählt einen der 7 Eingänge aus (0-basiert).
- IN1, IN2, IN3, IN4, IN5, IN6, IN7 (ANY): Die 7 zur Auswahl stehenden Eingangswerte.

Daten-Ausgänge

- **OUT** (ANY): Der ausgewählte Eingangswert: IN1 bei K = 0, IN2 bei K = 1, IN3 bei K = 2, IN4 bei K = 3, IN5 bei K = 4, IN6 bei K = 5, IN7 bei K = 6.

Funktionsweise

Bei Eintreffen von REQ wird abhängig vom Wert von K der entsprechende Eingang (IN1, IN2, IN3, IN4, IN5, IN6, IN7) auf OUT durchgeschaltet: IN1 bei K = 0, IN2 bei K = 1, IN3 bei K = 2, IN4 bei K = 3, IN5 bei K = 4, IN6 bei K = 5, IN7 bei K = 6. Anschließend wird CNF ausgelöst.

Technische Besonderheiten

- **Generische Implementierung**: eclipse4diac::core::GenericClassName = 'GEN_F_MUX', dieselbe C++-Basis wie F_MUX_2/F_MUX_3/F_MUX_4; nur die Anzahl der Eingänge unterscheidet sich pro Instanziierung.
- **ANY-Datentyp**: Sowohl die Eingänge IN1…IN7 als auch OUT sind generisch typisiert und passen sich beim Verdrahten an den angeschlossenen Datentyp an.
- **0-basierter Selektor**: K = 0 wählt IN1, K = 6 wählt IN7.

Zustandsübersicht

Zustandslos: jedes REQ führt unmittelbar zur Auswahl und zu CNF.

Anwendungsszenarien

- **Mehrfachauswahl mit mehr als 4 Quellen**: Überall dort, wo zwischen mehr als den von F_MUX_4 unterstützten 4 Eingängen ausgewählt werden muss, ohne mehrere F_MUX-Bausteine zu kaskadieren.
- **Signalumschaltung**: Auswahl eines von mehreren Sensor- oder Quellsignalen anhand eines Steuerwerts K.

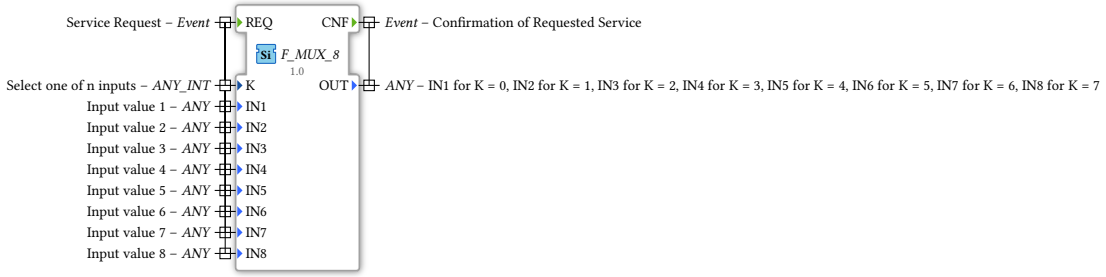
⚖️ Vergleich mit ähnlichen Bausteinen

- **F_MUX_6**: dieselbe generische Implementierung mit 6 statt 7 Eingängen.
- **F_SEL**: die binäre Auswahl zwischen genau zwei Werten anhand eines BOOL-Selektors, funktional ein Spezialfall von F_MUX_2.

Fazit

F_MUX_7 erweitert das Multiplexer-Prinzip von F_MUX_2/F_MUX_3/F_MUX_4 auf 7 wählbare Eingänge und ist die generische, für 7 Eingänge instanziierte Variante derselben zugrunde liegenden Implementierung.

F_MUX_8



• – ○ • – ○ • – ○ •

Einleitung

Der `F_MUX_8` ist die generische 8-Eingangs-Variante des Standard-Multiplexer-Bausteins `F_MUX` (vgl. `F_MUX_2`/`F_MUX_3`/`F_MUX_4`). Er wählt anhand des Selektors `K` einen von 8 Eingangswerten (`IN1`, `IN2`, `IN3`, `IN4`, `IN5`, `IN6`, `IN7`, `IN8`) aus und gibt ihn über `OUT` aus.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Auswahl aus, trägt `K` und `IN1`, `IN2`, `IN3`, `IN4`, `IN5`, `IN6`, `IN7`, `IN8`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT`.

Daten-Eingänge

- **K** (ANY_INT): Wählt einen der 8 Eingänge aus (`0`-basiert).
- `IN1`, `IN2`, `IN3`, `IN4`, `IN5`, `IN6`, `IN7`, `IN8` (ANY): Die 8 zur Auswahl stehenden Eingangswerte.

Daten-Ausgänge

- **OUT** (ANY): Der ausgewählte Eingangswert: `IN1` bei `K = 0`, `IN2` bei `K = 1`, `IN3` bei `K = 2`, `IN4` bei `K = 3`, `IN5` bei `K = 4`, `IN6` bei `K = 5`, `IN7` bei `K = 6`, `IN8` bei `K = 7`.

Funktionsweise

Bei Eintreffen von `REQ` wird abhängig vom Wert von `K` der entsprechende Eingang (`IN1`, `IN2`, `IN3`, `IN4`, `IN5`, `IN6`, `IN7`, `IN8`) auf `OUT` durchgeschaltet: `IN1` bei `K = 0`, `IN2` bei `K = 1`, `IN3` bei `K = 2`, `IN4` bei `K = 3`, `IN5` bei `K = 4`, `IN6` bei `K = 5`, `IN7` bei `K = 6`, `IN8` bei `K = 7`. Anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_F_MUX'`, dieselbe C++-Basis wie `F_MUX_2`/`F_MUX_3`/`F_MUX_4`; nur die Anzahl der Eingänge unterscheidet sich pro Instanziierung.
- **`ANY`-Datentyp**: Sowohl die Eingänge `IN1`…`IN8` als auch `OUT` sind generisch typisiert und passen sich beim Verdrahten an den angeschlossenen Datentyp an.
- **`0`-basierter Selektor**: `K = 0` wählt `IN1`, `K = 7` wählt `IN8`.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur Auswahl und zu `CNF`.

Anwendungsszenarien

- **Mehrfachauswahl mit mehr als 4 Quellen**: Überall dort, wo zwischen mehr als den von `F_MUX_4` unterstützten 4 Eingängen ausgewählt werden muss, ohne mehrere `F_MUX`-Bausteine zu kaskadieren.
- **Signalumschaltung**: Auswahl eines von mehreren Sensor- oder Quellsignalen anhand eines Steuerwerts `K`.

⚖️ Vergleich mit ähnlichen Bausteinen

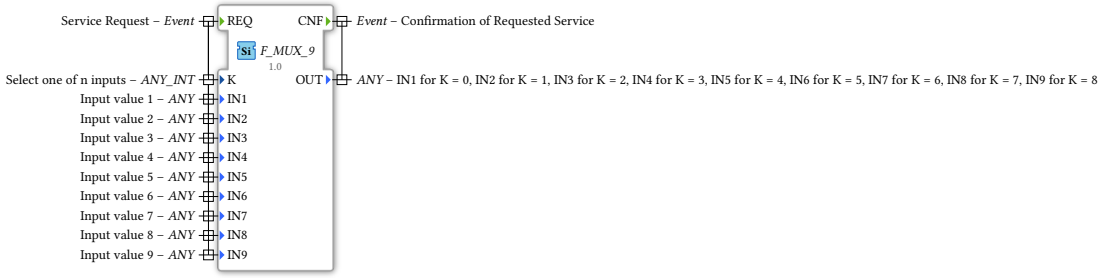
- **`F_MUX_7`**: dieselbe generische Implementierung mit 7 statt 8 Eingängen.

- **F_SEL**: die binäre Auswahl zwischen genau zwei Werten anhand eines BOOL-Selektors, funktional ein Spezialfall von F_MUX_2.

Fazit

F_MUX_8 erweitert das Multiplexer-Prinzip von F_MUX_2/F_MUX_3/F_MUX_4 auf 8 wählbare Eingänge und ist die generische, für 8 Eingänge instanziierte Variante derselben zugrunde liegenden Implementierung.

F_MUX_9



• – ○ • – ○ • – ○ •

Einleitung

Der F_MUX_9 ist die generische 9-Eingangs-Variante des Standard-Multiplexer-Bausteins F_MUX (vgl. F_MUX_2/F_MUX_3/F_MUX_4). Er wählt anhand des Selektors K einen von 9 Eingangswerten (IN1, IN2, IN3, IN4, IN5, IN6, IN7, IN8, IN9) aus und gibt ihn über OUT aus.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Auswahl aus, trägt K und IN1, IN2, IN3, IN4, IN5, IN6, IN7, IN8, IN9.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt OUT.

Daten-Eingänge

- **K** (ANY_INT): Wählt einen der 9 Eingänge aus (0-basiert).
- IN1, IN2, IN3, IN4, IN5, IN6, IN7, IN8, IN9 (ANY): Die 9 zur Auswahl stehenden Eingangswerte.

Daten-Ausgänge

- **OUT** (ANY): Der ausgewählte Eingangswert: IN1 bei K = 0, IN2 bei K = 1, IN3 bei K = 2, IN4 bei K = 3, IN5 bei K = 4, IN6 bei K = 5, IN7 bei K = 6, IN8 bei K = 7, IN9 bei K = 8.

Funktionsweise

Bei Eintreffen von REQ wird abhängig vom Wert von K der entsprechende Eingang (IN1, IN2, IN3, IN4, IN5, IN6, IN7, IN8, IN9) auf OUT durchgeschaltet: IN1 bei K = 0, IN2 bei K = 1, IN3 bei K = 2, IN4 bei K = 3, IN5 bei K = 4, IN6 bei K = 5, IN7 bei K = 6, IN8 bei K = 7, IN9 bei K = 8. Anschließend wird CNF ausgelöst.

Technische Besonderheiten

- **Generische Implementierung**: eclipse4diac::core::GenericClassName = 'GEN_F_MUX', dieselbe C++-Basis wie F_MUX_2/F_MUX_3/F_MUX_4; nur die Anzahl der Eingänge unterscheidet sich pro Instanziierung.
- **ANY-Datentyp**: Sowohl die Eingänge IN1…IN9 als auch OUT sind generisch typisiert und passen sich beim Verdrahten an den angeschlossenen Datentyp an.
- **0-basierter Selektor**: K = 0 wählt IN1, K = 8 wählt IN9.

Zustandsübersicht

Zustandslos: jedes REQ führt unmittelbar zur Auswahl und zu CNF.

Anwendungsszenarien

- **Mehrfachauswahl mit mehr als 4 Quellen**: Überall dort, wo zwischen mehr als den von `F_MUX_4` unterstützten 4 Eingängen ausgewählt werden muss, ohne mehrere `F_MUX`-Bausteine zu kaskadieren.
- **Signalumschaltung**: Auswahl eines von mehreren Sensor- oder Quellsignalen anhand eines Steuerwerts `K`.

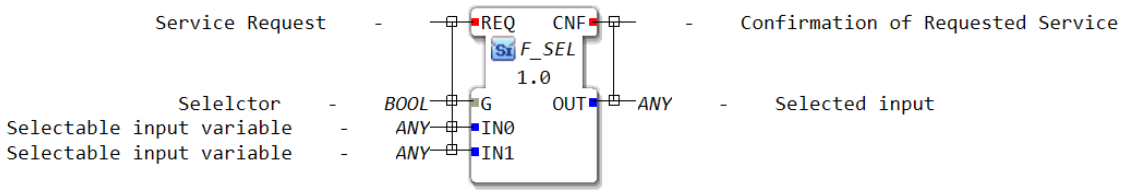
⚖️ Vergleich mit ähnlichen Bausteinen

- **`F_MUX_8`**: dieselbe generische Implementierung mit 8 statt 9 Eingängen.
- **`F_SEL`**: die binäre Auswahl zwischen genau zwei Werten anhand eines `BOOL`-Selektors, funktional ein Spezialfall von `F_MUX_2`.

Fazit

`F_MUX_9` erweitert das Multiplexer-Prinzip von `F_MUX_2`/`F_MUX_3`/`F_MUX_4` auf 9 wählbare Eingänge und ist die generische, für 9 Eingänge instanziierte Variante derselben zugrunde liegenden Implementierung.

F_SEL

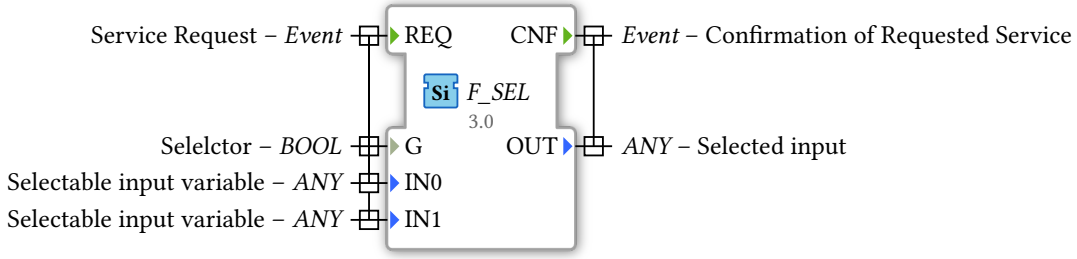


https://content.helpme-codesys.com/en/CODESYS%20Development%20System/_cds_operator_sel.html

• – ◦ • – ◦ • – ◦ •

Einleitung

Der F_SEL Funktionsblock ist ein binärer Selektor gemäß IEC 61131-3 Standard. Er ermöglicht die Auswahl zwischen zwei Eingangswerten basierend auf einem Steuersignal. Der Baustein dient zur bedingten Datenauswahl in Steuerungsanwendungen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Service-Anfrage - löst die Auswahloperation aus

Ereignis-Ausgänge

- **CNF**: Bestätigung der angeforderten Service-Operation

Daten-Eingänge

- **G**: Selektor (BOOL) - Steuersignal für die Auswahl
- **IN0**: Auswählbare Eingangsvariable (ANY) - Erster Eingang
- **IN1**: Auswählbare Eingangsvariable (ANY) - Zweiter Eingang

Daten-Ausgänge

- **OUT**: Ausgewählter Eingang (ANY) - Ergebnis der Selektion

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Bei Eintreffen eines REQ-Ereignisses wertet der F_SEL-Block den Selektor-Eingang G aus:

- Wenn G = FALSE (0): OUT wird auf den Wert von IN0 gesetzt
- Wenn G = TRUE (1): OUT wird auf den Wert von IN1 gesetzt

Nach Abschluss der Auswahloperation wird das CNF-Ereignis mit dem berechneten Ausgangswert ausgegeben.

Technische Besonderheiten

- Verwendet generische Datentypen (ANY) für die Ein- und Ausgänge IN0, IN1 und OUT
- Ermöglicht damit die Verarbeitung verschiedener Datentypen
- Echtzeitfähige Ausführung bei Ereignisauslösung
- Einfache und deterministische Funktionsweise

Zustandsübergänge

1. **Warte-Zustand**: Block wartet auf REQ-Ereignis
2. **Ausführungs-Zustand**: Bei REQ-Ereignis erfolgt Auswertung von G und Datenauswahl
3. **Ausgabe-Zustand**: CNF-Ereignis mit Ergebnis wird ausgegeben, Rückkehr in Warte-Zustand

Anwendungsszenarien

- Umschaltung zwischen zwei Sensoren oder Messwerten
- Auswahl zwischen verschiedenen Betriebsmodi
- Bedingte Datenweiterleitung in Prozessketten
- Implementierung von Multiplexer-Funktionalität

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Selektionsbausteinen bietet F_SEL:

- Einfache binäre Auswahl (nur zwei Eingänge)
- Flexible Typunterstützung durch ANY-Datentyp
- Standardisierte Ereignissteuerung gemäß IEC 61131-3

🛠️ Zugehörige Übungen

- Uebung_015
- Uebung_016
- Uebung_060

Fazit

Der F_SEL-Block stellt eine grundlegende und vielseitige Selektionsfunktion für Steuerungsanwendungen bereit. Seine einfache Struktur und die flexible Typunterstützung machen ihn zu einem wertvollen Baustein für verschiedene Auswahl- und Umschaltaufgaben in industriellen Automatisierungssystemen.

2.4.1.12 Zeitgeber (Timer)

FB_TOF

🎧 Podcast

- E_TOF vs. FB_TOF: Der Event-Timer, der nicht zyklisch tickt – Revolution für Automatisierungssysteme?
- FB_TOF und E_TOF: Verzögerungstimer in IEC 61131-3 und 61499

• – ◦ • – ◦ • – ◦ •

Folgender Text wurde erzeugt von https://chat.deepseek.com/ am 05.02.2025 und noch nicht korrekturgelesen (HF)

Einleitung

Wichtiger Hinweis: Dieser Baustein funktioniert nur korrekt, wenn er zyklisch aufgerufen wird.

Der Baustein FB_TOF in der IEC 61131-3: Ein Überblick

Überblick

Die IEC 61131-3 ist ein internationaler Standard, der die Programmierung von speicherprogrammierbaren Steuerungen (SPS) regelt. Ein zentrales Konzept in diesem Standard ist der Funktionsbaustein (FB), der als wiederverwendbare Komponente für die Steuerungslogik dient. Der Baustein **FB_TOF** (Off-Delay Timer) ist ein Beispiel für einen solchen Funktionsbaustein, der in der IEC 61131-3 verwendet wird, um eine verzögerte Abschaltfunktion zu modellieren. Dieser Aufsatz beschreibt die Struktur, das Verhalten und die Anwendung des FB_TOF-Bausteins anhand des bereitgestellten Quelltextes. Ein wichtiger Aspekt, der besonders hervorgehoben werden soll, ist, dass dieser Baustein nur dann korrekt funktioniert, wenn er **zyklisch aufgerufen** wird.

Struktur des FB_TOF-Bausteins

Der FB_TOF-Baustein ist ein standardisierter Funktionsbaustein gemäß der IEC 61131-3. Er besteht aus einer Schnittstelle (Interface), die Eingangs- und Ausgangsvariablen sowie Ereignisse definiert.

Schnittstelle (Interface)

Die Schnittstelle des FB_TOF-Bausteins besteht aus:

- **Eingangsereignis (Event Input)**:
  - **REQ (Request)**: Dieses Ereignis löst die Ausführung des Bausteins aus. Es ist mit den Eingangsvariablen **IN** und **PT** verknüpft.
- **Ausgangsereignis (Event Output)**:
  - **CNF (Confirmation)**: Dieses Ereignis wird ausgelöst, wenn die Ausführung des Bausteins abgeschlossen ist. Es ist mit den Ausgangsvariablen **Q** und **ET** verknüpft.
- **Eingangsvariablen (Input Variables)**:
  - **IN (Input)**: Dies ist eine boolesche Variable, die den Zustand des Timers steuert. Wenn **IN** auf **TRUE** gesetzt ist, startet der Timer. Wenn **IN** auf **FALSE** gesetzt wird, beginnt die Verzögerungszeit.
  - **PT (Process Time)**: Dies ist eine Zeitvariable (TIME), die die Verzögerungszeit definiert, nach der der Ausgang **Q** auf **FALSE** gesetzt wird.
- **Ausgangsvariablen (Output Variables)**:
  - **Q (Output)**: Dies ist eine boolesche Variable, die den Zustand des Timers anzeigt. Sie bleibt solange **TRUE**, bis die Verzögerungszeit **PT** abgelaufen ist.
  - **ET (Elapsed Time)**: Dies ist eine Zeitvariable (TIME), die die verstrichene Zeit seit dem Start der Verzögerung anzeigt.

Verhalten des FB_TOF-Bausteins

Der FB_TOF-Baustein verhält sich wie ein Off-Delay-Timer, der eine verzögerte Abschaltfunktion realisiert. Das Verhalten des Bausteins kann wie folgt beschrieben werden:

1. **Start des Timers**: Wenn das **REQ**-Ereignis eintritt und die Eingangsvariable **IN** auf **TRUE** gesetzt ist, startet der Timer. Der Ausgang **Q** wird auf **TRUE** gesetzt, und die verstrichene Zeit **ET** beginnt zu zählen.
2. **Verzögerungszeit**: Wenn die Eingangsvariable **IN** auf **FALSE** gesetzt wird, beginnt die Verzögerungszeit **PT**. Der Ausgang **Q** bleibt solange **TRUE**, bis die Verzögerungszeit abgelaufen ist.
3. **Ablauf der Verzögerungszeit**: Sobald die Verzögerungszeit **PT** abgelaufen ist, wird der Ausgang **Q** auf **FALSE** gesetzt, und das **CNF**-Ereignis wird ausgelöst, um den Abschluss der Verzögerung zu signalisieren.

Zyklischer Aufruf des FB_TOF-Bausteins

Ein kritischer Aspekt des FB_TOF-Bausteins ist, dass er **zyklisch aufgerufen** werden muss, um korrekt zu funktionieren. Das bedeutet, dass der Baustein in jedem Zyklus des SPS-Programms ausgeführt werden muss, um den Timer korrekt zu aktualisieren. Wenn der Baustein nicht zyklisch aufgerufen wird, kann die verstrichene Zeit **ET** nicht korrekt berechnet werden, und der Ausgang **Q** wird möglicherweise nicht rechtzeitig auf **FALSE** gesetzt.

Der zyklische Aufruf ist notwendig, weil der Baustein den Zustand des Timers (d.h. die verstrichene Zeit **ET**) in jedem Zyklus aktualisieren muss. Wenn der Baustein nicht regelmäßig aufgerufen wird, kann der Timer nicht korrekt arbeiten, und die Verzögerungszeit **PT** wird möglicherweise nicht eingehalten.

Anwendungsbeispiele

Der FB_TOF-Baustein kann in verschiedenen industriellen Steuerungsanwendungen eingesetzt werden, insbesondere in Szenarien, in denen eine verzögerte Abschaltfunktion erforderlich ist. Einige Beispiele sind:

- **Maschinensteuerung**: Der Baustein kann verwendet werden, um sicherzustellen, dass eine Maschine nach dem Ausschalten noch für eine bestimmte Zeit weiterläuft, z.B. um Prozesse abzuschließen oder Sicherheitsvorkehrungen zu treffen.
- **Beleuchtungssteuerung**: Der Baustein kann in Beleuchtungssystemen eingesetzt werden, um das Licht nach dem Ausschalten des Schalters noch für eine bestimmte Zeit brennen zu lassen.
- **Ventilsteuerung**: Der Baustein kann verwendet werden, um ein Ventil nach dem Schließen noch für eine bestimmte Zeit offen zu halten, um z.B. Druck abzulassen.

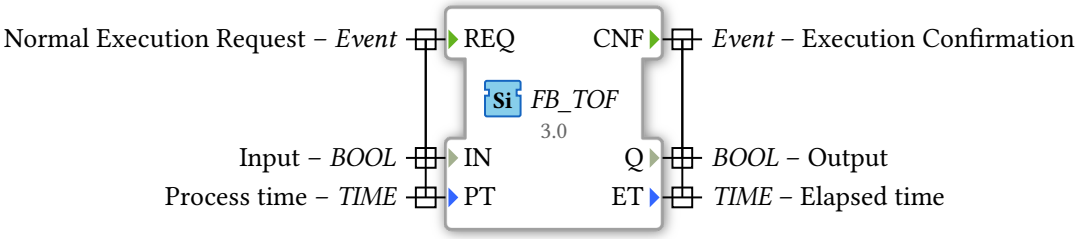
🛠️ Zugehörige Übungen

- Uebung_020e2
- Uebung_020e2_AX

Fazit

Der FB_TOF-Baustein ist ein einfacher, aber leistungsfähiger Funktionsbaustein in der IEC 61131-3, der eine verzögerte Abschaltfunktion realisiert. Durch seine klare Struktur und einfache Bedienung eignet er sich ideal für Anwendungen, in denen eine zeitverzögerte Steuerung erforderlich ist. Ein wichtiger Aspekt, der beachtet werden muss, ist, dass der Baustein **zyklisch aufgerufen** werden muss, um korrekt zu funktionieren. Nur durch den regelmäßigen Aufruf in jedem SPS-Zyklus kann der Timer die verstrichene Zeit korrekt berechnen und die Verzögerungszeit einhalten.

Der bereitgestellte Quelltext zeigt, wie der FB_TOF-Baustein in der Praxis implementiert werden kann, und bietet eine solide Grundlage für die Entwicklung ähnlicher Bausteine in industriellen Steuerungssystemen. Der zyklische Aufruf ist dabei ein entscheidender Faktor für die korrekte Funktionsweise des Bausteins.



⚖️ Vergleich E_TOF zu FB_TOF

Kapitel 1: Der FB_TOF-Baustein

Der **FB_TOF** (Off-Delay Timer) ist ein standardisierter Funktionsbaustein gemäß der IEC 61131-3, der eine verzögerte Abschaltfunktion realisiert. Ein wesentliches Merkmal dieses Bausteins ist der Ausgang **ET** (Elapsed Time), der die verstrichene Zeit seit dem Start der Verzögerung anzeigt. Dieser Ausgang ist besonders nützlich, um den Fortschritt der Verzögerungszeit zu überwachen und in Steuerungslogiken zu verwenden.

Ein kritischer Aspekt des FB_TOF-Bausteins ist jedoch, dass er **zyklisch aufgerufen werden muss**, um korrekt zu funktionieren. Das bedeutet, dass der Baustein in jedem Zyklus des SPS-Programms ausgeführt werden muss, um die verstrichene Zeit **ET** korrekt zu berechnen und die Verzögerungszeit **PT** einzuhalten. Wenn der Baustein nicht regelmäßig aufgerufen wird, kann der Timer nicht korrekt arbeiten, und die Verzögerungszeit wird möglicherweise nicht eingehalten. Diese Abhängigkeit vom zyklischen Aufruf macht den FB_TOF-Baustein ideal für traditionelle SPS-Systeme, die in festen Zyklen arbeiten.

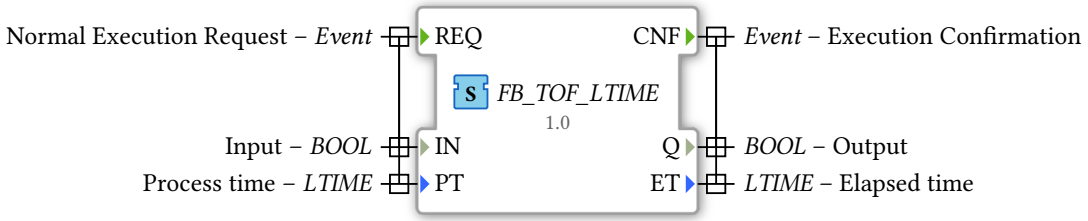
Kapitel 2: Der E_TOF-Baustein

Der **E_TOF** (Event-driven Off-Delay Timer) ist ein Funktionsbaustein gemäß der IEC 61499, der ebenfalls eine verzögerte Abschaltfunktion realisiert. Im Gegensatz zum FB_TOF besitzt der E_TOF **keinen Ausgang ET**, der die verstrichene Zeit anzeigt. Stattdessen basiert der E_TOF intern auf dem **E_DELAY**-Baustein, der in der IEC 61499 spezifiziert ist und eine ereignisgesteuerte Verzögerungsfunktion bietet.

Ein entscheidender Vorteil des E_TOF-Bausteins ist, dass er **nicht zyklisch aufgerufen werden muss**, um korrekt zu funktionieren. Da der interne **E_DELAY**-Baustein unabhängig vom zyklischen Aufruf des übergeordneten Bausteins arbeitet, kann der E_TOF auch in Systemen eingesetzt werden, die nicht zyklisch arbeiten, z.B. in ereignisgesteuerten oder verteilten Steuerungssystemen. Dies macht den E_TOF-Baustein besonders flexibel und geeignet für moderne, verteilte Steuerungsarchitekturen, bei denen zyklische Aufrufe nicht immer gewährleistet sind.

Zusammenfassend lässt sich sagen, dass der FB_TOF durch seinen Ausgang **ET** eine detaillierte Überwachung der verstrichenen Zeit ermöglicht, jedoch einen zyklischen Aufruf erfordert. Der E_TOF hingegen verzichtet auf den Ausgang **ET**, bietet dafür aber die Flexibilität, unabhängig von zyklischen Aufrufen zu arbeiten, was ihn ideal für ereignisgesteuerte Systeme macht.

FB_TOF_LTIME



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `FB_TOF_LTIME` ist die `LTIME`-Variante (64-Bit-Zeitdauer, Nanosekunden-Auflösung) des Standard-Timer-Bausteins `FB_TOF` (Ausschaltverzögerung). Er verhält sich funktional identisch zu `FB_TOF`, verwendet für `PT`/`ET` jedoch den Datentyp `LTIME` statt `TIME`.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung, verknüpft mit `IN` und `PT`.

Ereignis-Ausgänge

- **CNF**: Bestätigt die Ausführung, verknüpft mit `Q` und `ET`.

Daten-Eingänge

- **IN** (BOOL): Eingangssignal.
- **PT** (LTIME): Vorgabezeit.

Daten-Ausgänge

- **Q** (BOOL): Ausgangssignal, siehe Funktionsweise.
- **ET** (LTIME): Verstrichene Zeit seit Start der Zeitmessung.

Funktionsweise

Q wird sofort TRUE, wenn IN auf TRUE wechselt. Wechselt IN auf FALSE, bleibt Q noch für die Dauer PT TRUE, bevor es auf FALSE zurückfällt. Intern hält der Baustein (analog zu `FB_TOF`) die internen Hilfsvariablen `MEM` und `StartTime`, um Flankenwechsel von `IN` und den Startzeitpunkt der Zeitmessung zu verfolgen — mit `LTIME`-Auflösung statt `TIME`.

Technische Besonderheiten

- **`LTIME` statt `TIME`**: Bietet 64-Bit-Auflösung in Nanosekunden statt der 32-Bit-Millisekundenauflösung von `TIME`, relevant für sehr lange oder sehr präzise Zeitmessungen.

- **Funktional identisch zu `FB_TOF`**: Gleiches Timing-Verhalten, lediglich der Datentyp von `PT`/`ET` unterscheidet sich.

Zustandsübersicht

Siehe `FB_TOF` — identisches Zustandsverhalten, nur mit `LTIME`-Zeitwerten statt `TIME`.

Anwendungsszenarien

- **Zeitmessungen mit sehr hoher Präzision oder sehr langer Dauer**, bei denen die Millisekundenauflösung bzw. der Wertebereich von `TIME` nicht ausreicht.
- **Konsistente `LTIME`-Verwendung** in Applikationen, die durchgehend mit `LTIME` statt `TIME` arbeiten, um Datentyp-Konvertierungen zu vermeiden.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`FB_TOF`**: dieselbe Funktion mit `TIME` statt `LTIME`.

Fazit

`FB_TOF_LTIME` liefert dieselbe Ausschaltverzögerung-Funktionalität wie `FB_TOF`, jedoch mit `LTIME`-Auflösung, und eignet sich für Applikationen, die durchgängig mit hochauflösenden Zeitwerten arbeiten.

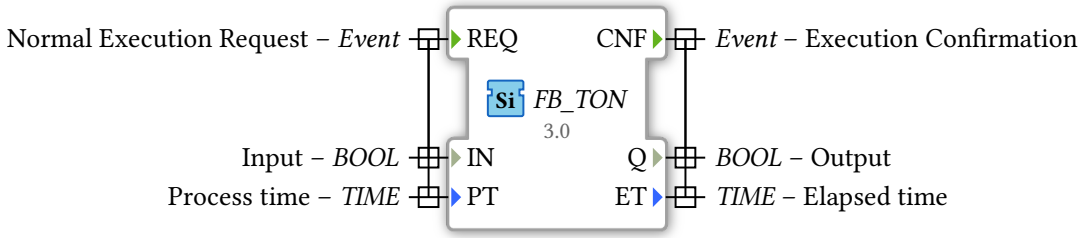
FB_TON

• – ◦ • – ◦ • – ◦ •

Einleitung

Wichtiger Hinweis: Dieser Baustein funktioniert nur korrekt, wenn er zyklisch aufgerufen wird.

Der FB_TON ist ein standardisierter Timer-Funktionsblock gemäß IEC 61131-3, der eine Einschaltverzögerung (On-Delay) realisiert. Dieser Baustein wird verwendet, um zeitgesteuerte Verzögerungen in Steuerungsprogrammen zu implementieren.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Normal Execution Request - Startet die Zeitüberwachung bei aktivem IN-Signal

Ereignis-Ausgänge

- **CNF**: Execution Confirmation - Bestätigt den Abschluss der Zeitüberwachung

Daten-Eingänge

- **IN**: Input - Boolescher Eingang, der den Timer startet (TRUE = Timer läuft)
- **PT**: Process Time - Zeitwert, der die Verzögerungsdauer definiert

Daten-Ausgänge

- **Q**: Output - Boolescher Ausgang, der nach Ablauf der Verzögerungszeit TRUE wird
- **ET**: Elapsed Time - Zeigt die bereits verstrichene Zeit an

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der FB_TON startet die Zeitmessung, wenn der Eingang IN von FALSE auf TRUE wechselt. Sobald die vorgegebene Zeit PT verstrichen ist, wird der Ausgang Q auf TRUE gesetzt. Der Ausgang ET zeigt kontinuierlich die bereits verstrichene Zeit an. Wird IN während der Zeitmessung auf FALSE gesetzt, wird der Timer zurückgesetzt und Q bleibt FALSE.

Technische Besonderheiten

- Zeitwerte werden im TIME-Datentyp verarbeitet
- Der Timer wird bei IN=FALSE sofort zurückgesetzt
- ET zeigt immer die aktuelle verstrichene Zeit an, auch wenn der Timer noch läuft
- Konforme Implementierung nach IEC 61131-3 Standard

Zustandsübergänge

1. **Inaktiv**: IN = FALSE, Q = FALSE, ET = 0
2. **Aktiv (Timer läuft)**: IN = TRUE, Q = FALSE, ET zählt hoch
3. **Zeit abgelaufen**: IN = TRUE, Q = TRUE, ET = PT

Anwendungsszenarien

- Verzögerte Schaltvorgänge in Maschinensteuerungen
- Sicherheitszeitüberwachungen
- Prozesssteuerungen mit definierten Wartezeiten
- Verzögerte Alarmauslösungen

⚖️ Vergleich mit ähnlichen Bausteinen

- **FB_TOF**: Ausschaltverzögerung (Off-Delay) statt Einschaltverzögerung
- **FB_TP**: Impulstimer mit fester Pulsdauer
- **FB_TON_EXT**: Erweiterte Timer-Varianten mit zusätzlichen Funktionen

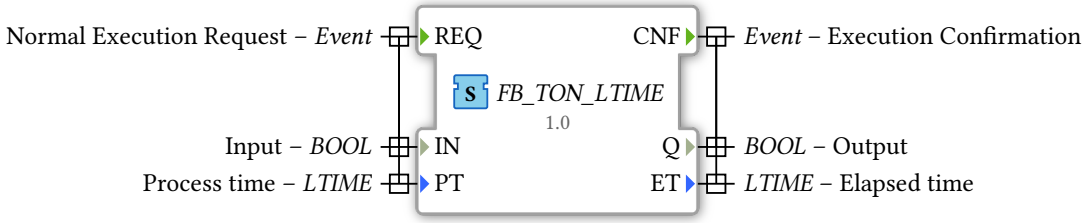
🛠️ Zugehörige Übungen

- Uebung_020c3
- Uebung_020c3_AX

Fazit

Der FB_TON ist ein grundlegender und robuster Timer-Baustein für Standard-Einschaltverzögerungen in industriellen Steuerungssystemen. Seine einfache und intuitive Bedienung sowie die Konformität zum IEC 61131-3 Standard machen ihn zu einer zuverlässigen Wahl für zeitgesteuerte Anwendungen.

FB_TON_LTIME



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `FB_TON_LTIME` ist die `LTIME`-Variante (64-Bit-Zeitdauer, Nanosekunden-Auflösung) des Standard-Timer-Bausteins `FB_TON` (Einschaltverzögerung). Er verhält sich funktional identisch zu `FB_TON`, verwendet für `PT`/`ET` jedoch den Datentyp `LTIME` statt `TIME`.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung, verknüpft mit `IN` und `PT`.

Ereignis-Ausgänge

- **CNF**: Bestätigt die Ausführung, verknüpft mit `Q` und `ET`.

Daten-Eingänge

- **IN** (BOOL): Eingangssignal.
- **PT** (LTIME): Vorgabezeit.

Daten-Ausgänge

- **Q** (BOOL): Ausgangssignal, siehe Funktionsweise.
- **ET** (LTIME): Verstrichene Zeit seit Start der Zeitmessung.

Funktionsweise

Q wird erst TRUE, wenn IN seit PT ununterbrochen TRUE war. Fällt IN vor Ablauf von PT zurück auf FALSE, bleibt Q FALSE. Intern hält der Baustein (analog zu `FB_TON`) die internen Hilfsvariablen `MEM` und `StartTime`, um Flankenwechsel von `IN` und den Startzeitpunkt der Zeitmessung zu verfolgen — mit `LTIME`-Auflösung statt `TIME`.

Technische Besonderheiten

- **`LTIME` statt `TIME`**: Bietet 64-Bit-Auflösung in Nanosekunden statt der 32-Bit-Millisekundenauflösung von `TIME`, relevant für sehr lange oder sehr präzise Zeitmessungen.
- **Funktional identisch zu `FB_TON`**: Gleiches Timing-Verhalten, lediglich der Datentyp von `PT`/`ET` unterscheidet sich.

Zustandsübersicht

Siehe `FB_TON` — identisches Zustandsverhalten, nur mit `LTIME`-Zeitwerten statt `TIME`.

Anwendungsszenarien

- **Zeitmessungen mit sehr hoher Präzision oder sehr langer Dauer**, bei denen die Millisekundenauflösung bzw. der Wertebereich von `TIME` nicht ausreicht.
- **Konsistente `LTIME`-Verwendung** in Applikationen, die durchgehend mit `LTIME` statt `TIME` arbeiten, um Datentyp-Konvertierungen zu vermeiden.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`FB_TON`**: dieselbe Funktion mit `TIME` statt `LTIME`.

Fazit

`FB_TON_LTIME` liefert dieselbe Einschaltverzögerung-Funktionalität wie `FB_TON`, jedoch mit `LTIME`-Auflösung, und eignet sich für Applikationen, die durchgängig mit hochauflösenden Zeitwerten arbeiten.

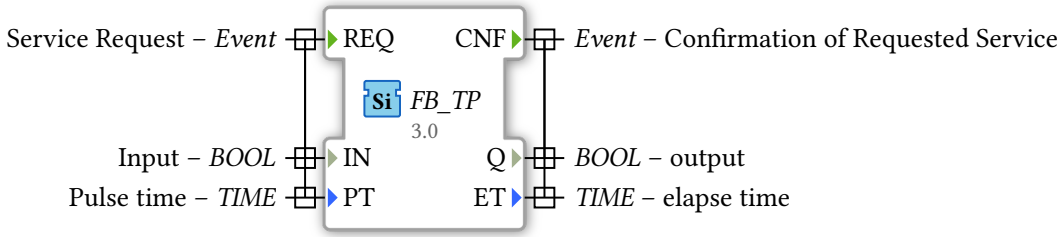
FB_TP

• – ◦ • – ◦ • – ◦ •

Einleitung

Wichtiger Hinweis: Dieser Baustein funktioniert nur korrekt, wenn er zyklisch aufgerufen wird.

Der FB_TP (Timer Pulse) ist ein standardisierter Zeitgeber-Funktionsblock nach IEC 61131-3, der eine pulsierende Ausgangsfunktion bereitstellt. Der Baustein erzeugt einen Impuls mit einer definierten Zeitdauer, unabhängig von der Dauer des Eingangssignals.



Schnittstellenstruktur

Ereignis-Eingänge
- **REQ**: Service-Anforderung - Startet den Zeitgeber

Ereignis-Ausgänge
- **CNF**: Bestätigung der angeforderten Serviceleistung - Signalisiert das Ende des Zeitimpulses

Daten-Eingänge
- **IN**: Eingangssignal (BOOL) - Steuert den Start des Zeitgebers
- **PT**: Impulszeit (TIME) - Definiert die Dauer des Ausgangsimpulses

Daten-Ausgänge
- **Q**: Ausgangssignal (BOOL) - Gibt den aktuellen Zustand des Zeitgebers aus
- **ET**: Verstrichene Zeit (TIME) - Zeigt die bereits verstrichene Zeit des laufenden Impulses an

Adapter
Keine Adapter-Schnittstellen vorhanden.

Funktionsweise
Der FB_TP funktioniert als monostabiler Multivibrator (One-Shot). Bei einem positiven Flanke am IN-Eingang startet der Zeitgeber und setzt den Q-Ausgang für die in PT definierte Zeitdauer auf TRUE. Die verstrichene Zeit wird kontinuierlich am ET-Ausgang ausgegeben. Nach Ablauf der eingestellten Zeit PT wird Q automatisch auf FALSE zurückgesetzt, unabhängig vom Zustand des IN-Eingangs.

Technische Besonderheiten
- Zeitmessung erfolgt unabhängig von der Dauer des Eingangssignals
- ET-Ausgang zeigt die aktuelle Laufzeit des Impulses
- Sofortige Reaktion auf positive Flanke am IN-Eingang
- Automatische Rücksetzung nach Zeitablauf

Zustandsübergänge
1. **Ruhezustand**: IN = FALSE, Q = FALSE, ET = 0
2. **Start**: Positive Flanke an IN ⟶ Q = TRUE, ET-Zählung beginnt
3. **Laufzeit**: Q bleibt TRUE während ET < PT
4. **Ende**: Wenn ET ≥ PT ⟶ Q = FALSE, ET-Zählung stoppt

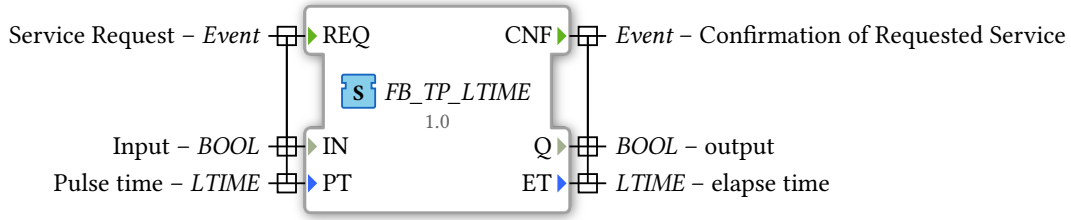
Anwendungsszenarien
- Erzeugung von genau definierten Impulsen
- Zeitgesteuerte Steuerungsabläufe
- Verzögerungsschaltungen
- Taktgenerierung für Prozesssteuerungen
- Sicherheitszeitüberwachung

⚖️ Vergleich mit ähnlichen Bausteinen
- **FB_TON**: Einschaltverzögerung - Q wird erst nach Ablauf von PT aktiv
- **FB_TOF**: Ausschaltverzögerung - Q bleibt nach IN-Rücksetzung noch für PT aktiv
- **FB_TP**: Impulsgeber - Erzeugt einen festen Impuls unabhängig von IN

🛠️ Zugehörige Übungen
- Uebung_020f2
- Uebung_020f2_AX

Fazit
Der FB_TP ist ein essentieller Baustein für zeitgesteuerte Anwendungen in der Automatisierungstechnik. Seine einfache Handhabung und zuverlässige Funktionsweise machen ihn ideal für Anwendungen, bei denen präzise Zeitimpulse benötigt werden. Die klare Trennung von Startimpuls und Impulsdauer ermöglicht flexible Steuerungskonzepte.

FB_TP_LTIME



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `FB_TP_LTIME` ist die `LTIME`-Variante (64-Bit-Zeitdauer, Nanosekunden-Auflösung) des Standard-Timer-Bausteins `FB_TP` (Impuls). Er verhält sich funktional identisch zu `FB_TP`, verwendet für `PT`/`ET` jedoch den Datentyp `LTIME` statt `TIME`.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung, verknüpft mit `IN` und `PT`.

Ereignis-Ausgänge

- **CNF**: Bestätigt die Ausführung, verknüpft mit `Q` und `ET`.

Daten-Eingänge

- **IN** (BOOL): Eingangssignal.
- **PT** (LTIME): Vorgabezeit.

Daten-Ausgänge

- **Q** (BOOL): Ausgangssignal, siehe Funktionsweise.
- **ET** (LTIME): Verstrichene Zeit seit Start der Zeitmessung.

Funktionsweise

Ein steigender Flankenwechsel von IN löst einen TRUE-Impuls der festen Dauer PT an Q aus, unabhängig davon, wie lange IN TRUE bleibt. Intern hält der Baustein (analog zu `FB_TP`) die internen Hilfsvariablen `MEM` und `StartTime`, um Flankenwechsel von `IN` und den Startzeitpunkt der Zeitmessung zu verfolgen — mit `LTIME`-Auflösung statt `TIME`.

Technische Besonderheiten

- **`LTIME` statt `TIME`**: Bietet 64-Bit-Auflösung in Nanosekunden statt der 32-Bit-Millisekundenauflösung von `TIME`, relevant für sehr lange oder sehr präzise Zeitmessungen.
- **Funktional identisch zu `FB_TP`**: Gleiches Timing-Verhalten, lediglich der Datentyp von `PT`/`ET` unterscheidet sich.

Zustandsübersicht

Siehe `FB_TP` — identisches Zustandsverhalten, nur mit `LTIME`-Zeitwerten statt `TIME`.

Anwendungsszenarien

- **Zeitmessungen mit sehr hoher Präzision oder sehr langer Dauer**, bei denen die Millisekundenauflösung bzw. der Wertebereich von `TIME` nicht ausreicht.
- **Konsistente `LTIME`-Verwendung** in Applikationen, die durchgehend mit `LTIME` statt `TIME` arbeiten, um Datentyp-Konvertierungen zu vermeiden.

⚖️ Vergleich mit ähnlichen Bausteinen

- **`FB_TP`**: dieselbe Funktion mit `TIME` statt `LTIME`.

Fazit

`FB_TP_LTIME` liefert dieselbe Impuls-Funktionalität wie `FB_TP`, jedoch mit `LTIME`-Auflösung, und eignet sich für Applikationen, die durchgängig mit hochauflösenden Zeitwerten arbeiten.

2.4.2 🔌 io

2.4.2.1 📺 Video

- Automation der Zukunft
- Die große Migration
- Digital decisions
- Ideale Dioden
- The Dual Wi Fi Solution

2.4.2.2 🎧 Podcast

- 4diac IDE: Wie der IEC 61499 Standard die Industrieautomatisierung revolutioniert
- 4diac-Präsentation: Zielgruppen, Struktur und Alleinstellungsmerkmal Schärfen
- Den Software-Drachen zähmen: Industrielle Automatisierung und die Zukunft der Produktion
- Eclipse 4diac: Innovationsmotor Forschung vs. Nutzerbedürfnisse – Was treibt die Entwicklung wirklich voran?
- Software-Drachen zähmen: Warum IEC 61499 die Industrie revolutionieren könnte

---

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 Diode & Halbleitergrundlagen auf ms-muc-docs.de
- 🌐 Smart Switch vs. Relais auf ms-muc-docs.de

2.4.2.3 IB

🎧 Podcast

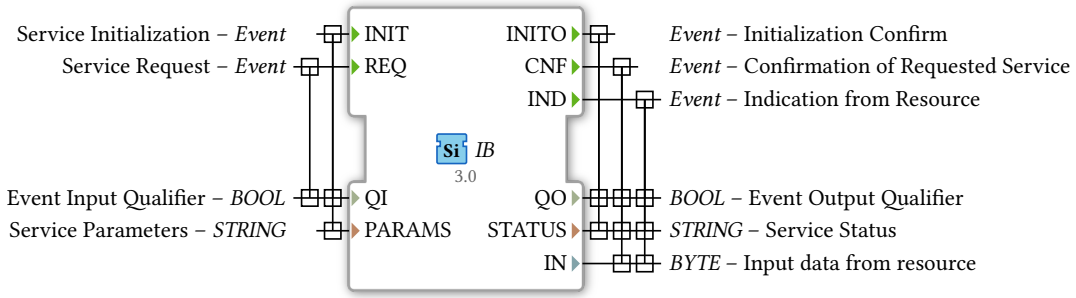
- Eclipse 4diac: Innovationsmotor Forschung vs. Nutzerbedürfnisse – Was treibt die Entwicklung wirklich voran?
- DIN EN 61499-1: Der digitale Lego-Baukasten für flexible Automatisierung und smarte Zukunft
- DIN EN 61499-1: Die Lego-Steine für flexible und ereignisgesteuerte Industriesteuerungen
- IEC 61499: The DNA of Flexible Automation – Powering Industry 4.0 from Factories to the High Seas
- IEC 61499 vs. 61131: Unpacking the Architectural Revolution of Distributed Control and Industry 4.0

📺 Video

- LogiBUS Projekt Update

Einleitung

Der IB-Funktionsblock (Input Byte) ist ein Dienstschnittstellen-Funktionsblock für die Verarbeitung von Byte-Eingabedaten. Er dient als Schnittstelle zwischen einem IEC 61499-Steuerungssystem und externen Eingabegeräten oder Ressourcen, die Byte-Daten liefern.



Schnittstellenstruktur

Ereignis-Eingänge
- **INIT**: Dienstinitialisierung - Initialisiert den Funktionsblock und konfiguriert die Schnittstellenparameter
- **REQ**: Dienstanforderung - Fordert eine aktuelle Dateneingabe an

Ereignis-Ausgänge
- **INITO**: Initialisierungsbestätigung - Bestätigt die erfolgreiche Initialisierung
- **CNF**: Dienstbestätigung - Bestätigt die angeforderten Dienstoperationen
- **IND**: Indikation von der Ressource - Signalisiert eingehende Daten von der externen Ressource

Daten-Eingänge
- **QI**: Ereigniseingangsqualifizierer (BOOL) - Steuert die Ausführung der Ereignisse
- **PARAMS**: Dienstparameter (STRING) - Konfigurationsparameter für die Schnittstelle

Daten-Ausgänge
- **QO**: Ereignisausgangsqualifizierer (BOOL) - Statusindikator für die Ausgangsereignisse
- **STATUS**: Dienststatus (STRING) - Statusinformationen über die aktuelle Operation
- **IN**: Eingabedaten von der Ressource (BYTE) - Die empfangenen Byte-Daten

Adapter
Keine Adapter-Schnittstellen vorhanden.

Funktionsweise
Der IB-Funktionsblock arbeitet als bidirektionale Schnittstelle für Byte-Eingabedaten. Bei Initialisierung (INIT) werden die Schnittstellenparameter konfiguriert. Über REQ-Ereignisse können Daten explizit angefordert werden, während IND-Ereignisse asynchrone Dateneingänge signalisieren. Die empfangenen Byte-Daten werden über den IN-Ausgang bereitgestellt.

Technische Besonderheiten
- Unterstützt sowohl anforderungsbasierte (REQ/CNF) als auch ereignisbasierte (IND) Datenerfassung
- BYTE-Datentyp für kompakte 8-Bit-Datenverarbeitung
- String-basierte Parameterkonfiguration für flexible Anpassung
- Statusrückmeldung über SERVICE-Statusausgang

Zustandsübersicht
Der Funktionsblock durchläuft folgende Hauptzustände:

1. **Nicht initialisiert**: Wartet auf INIT-Ereignis
2. **Initialisiert**: Bereit für Datenerfassung
3. **Datenanforderung**: Verarbeitet REQ-Ereignisse
4. **Datenempfang**: Verarbeitet eingehende IND-Ereignisse

Anwendungsszenarien
- Einlesen von 8-Bit-Sensordaten
- Schnittstelle zu einfachen digitalen Eingabegeräten
- Kommunikation mit Peripheriegeräten, die Byte-Daten liefern
- Protokollimplementierungen mit Byte-orientierten Datenformaten

⚖️ Vergleich mit ähnlichen Bausteinen
Im Vergleich zu anderen Eingabebausteinen:

- IB spezialisiert auf BYTE-Datentyp (8-Bit)
- Bietet sowohl pollende (REQ) als auch interrupt-artige (IND) Betriebsmodi
- Einfacher als komplexere Protokollbausteine, aber spezifischer für Byte-Daten

🛠️ Zugehörige Übungen
- Uebung_006c
- Uebung_011a

Fazit

Der IB-Funktionsblock bietet eine effiziente und flexible Lösung für die Integration von Byte-basierten Eingabedaten in IEC 61499-Steuerungssysteme. Seine duale Betriebsart (anforderungsbasiert und ereignisgesteuert) macht ihn vielseitig einsetzbar für verschiedene Anwendungsfälle in der industriellen Automatisierung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.2.4 ID

🎧 Podcast

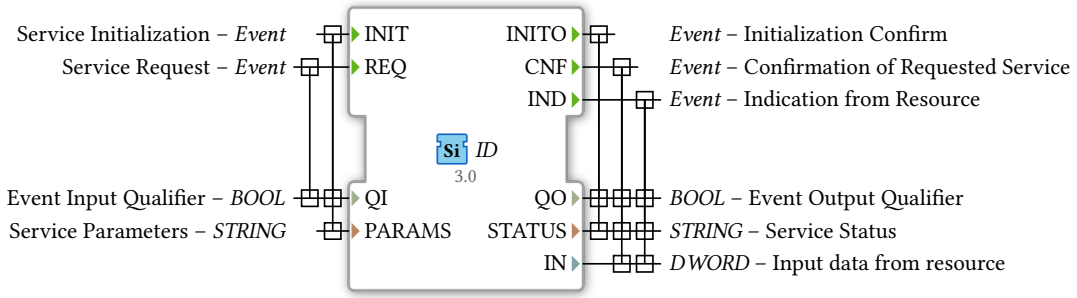
- 4diac IDE: Dein „Hello World“ der Automatisierung – Das Blinking Tutorial Lokal
- 4diac IDE: Dein Open-Source-Werkzeugkasten für verteilte Industrieautomatisierung nach IEC 61499
- 4diac IDE: Wie der IEC 61499 Standard die Industrieautomatisierung revolutioniert
- From Pyramid to Plug-and-Play: The Rise of Self-Configurable Industrial Automation
- Building Tomorrow’s Factories: Bridging OT and IT with IEC 61499

📺 Video

- Ideale Dioden

Einleitung

Der ID-Funktionsblock ist ein Eingabeservice-Interface-Baustein für Doppelwort-Eingabedaten (DWORD). Er dient als Schnittstelle zwischen der Steuerungslogik und externen Eingabegeräten oder Ressourcen und ermöglicht die Abfrage von 32-Bit-Eingabedaten.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung - Initialisiert den Funktionsblock und konfiguriert die Parameter
- **REQ**: Service-Anfrage - Löst eine Abfrage der Eingabedaten aus

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung - Bestätigt die erfolgreiche Initialisierung
- **CNF**: Bestätigung der angeforderten Service - Bestätigt eine erfolgreiche Service-Anfrage
- **IND**: Indikation von der Ressource - Signalisiert eingehende Daten von der Ressource

Daten-Eingänge

- **QI**: Event-Input-Qualifier (BOOL) - Aktiviert/deaktiviert den Service
- **PARAMS**: Service-Parameter (STRING) - Konfigurationsparameter für den Service

Daten-Ausgänge

- **QO**: Event-Output-Qualifier (BOOL) - Status des Service-Ausgangs
- **STATUS**: Service-Status (STRING) - Statusinformationen über den Service
- **IN**: Eingabedaten von der Ressource (DWORD) - Die gelesenen 32-Bit-Eingabedaten

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der ID-Baustein arbeitet als Service-Interface für Doppelwort-Eingaben. Bei Initialisierung (INIT) werden die Service-Parameter konfiguriert. Anschließend können über REQ-Ereignisse gezielt Eingabedaten von der angeschlossenen Ressource abgefragt werden. Der Baustein liefert die gelesenen Daten über die Ausgänge IN zusammen mit Statusinformationen zurück.

Technische Besonderheiten

- Verarbeitet 32-Bit-Daten (DWORD)
- Unterstützt sowohl angeforderte (REQ/CNF) als auch spontane (IND) Datenübertragungen
- Flexible Parameterkonfiguration über STRING-Parameter
- Qualifier-basierte Steuerung (QI/QO) für Service-Aktivierung

Zustandsübersicht

Der Baustein durchläuft folgende Hauptzustände:

1. **Nicht initialisiert**: Vor der INIT-Verarbeitung
2. **Initialisiert**: Nach erfolgreicher INIT-Verarbeitung, bereit für Datenabfragen
3. **Datenabfrage**: Während der Verarbeitung von REQ-Ereignissen
4. **Datenempfang**: Bei spontan eingehenden Daten (IND)

Anwendungsszenarien

- Abfrage von 32-Bit-Sensordaten
- Einlesen von digitalen Eingangssignalen in Gruppen
- Kommunikation mit Peripheriegeräten, die Doppelwort-Daten liefern
- Integration von externen Messsystemen in 4diac-Steuerungen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren Eingabebausteinen bietet ID:

- Erweiterte Statusrückmeldungen
- Konfigurierbare Service-Parameter
- Unterstützung für beide Betriebsmodi (anforderungsbasiert und spontan)
- 32-Bit-Datenbreite statt einfacher BOOL- oder BYTE-Werte

🛠️ Zugehörige Übungen

- Uebung_011
- Uebung_011a2
- Uebung_012
- NumericValue_ID_TO_NVS
- Uebung_012b
- Uebung_012c
- Uebung_020c2_sub
- Uebung_028
- Uebung_034
- Uebung_034a1_Q1
- Uebung_034a1_Q2
- Uebung_034a1_Q4
- Uebung_060
- Uebung_103
- Uebung_150
- Uebung_150_AX
- Uebung_151
- Uebung_151_AX
- Uebung_152
- Uebung_153

Fazit

Der ID-Funktionsblock stellt eine leistungsfähige und flexible Schnittstelle für Doppelwort-Eingabedaten bereit. Durch seine umfangreiche Statusrückmeldung und konfigurierbaren Parameter eignet er sich besonders für anspruchsvolle Anwendungen, bei denen zuverlässige und informative Eingabedatenverarbeitung erforderlich ist.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 Diode & Halbleitergrundlagen auf ms-muc-docs.de
- 🌐 Smart Switch vs. Relais auf ms-muc-docs.de

2.4.2.5 IE

🎧 Podcast

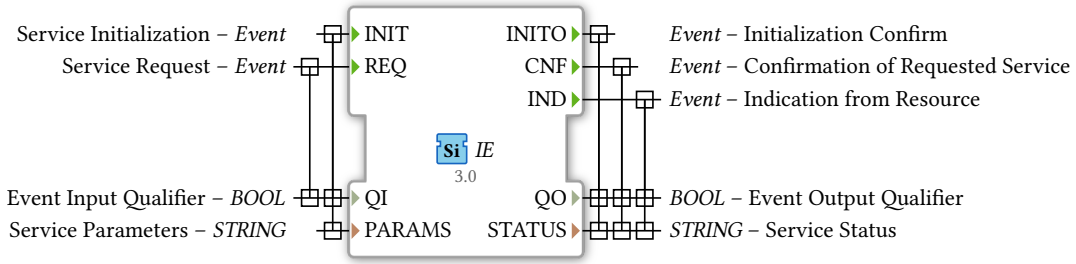
- 4diac IDE: Dein „Hello World“ der Automatisierung – Das Blinking Tutorial Lokal
- 4diac IDE: Dein Open-Source-Werkzeugkasten für verteilte Industrieautomatisierung nach IEC 61499
- 4diac IDE: Wie der IEC 61499 Standard die Industrieautomatisierung revolutioniert
- 4diac-Präsentation: Zielgruppen, Struktur und Alleinstellungsmerkmal Schärfen
- Den Software-Drachen zähmen: Industrielle Automatisierung und die Zukunft der Produktion

📺 Video

- Die große Migration
- Die Kunst des Lötens
- Die Welt der Normung
- Löten wie ein Profi
- The secret of the field

Einleitung

Der IE-Funktionsblock (Input Event) ist ein Service-Interface-Funktionsblock für die Verarbeitung von Ereigniseingangsdaten. Er dient als Schnittstelle zwischen der Steuerungslogik und externen Eingabegeräten oder -signalen und ermöglicht die Initialisierung, Abfrage und Indikation von Eingabeereignissen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung - Initialisiert den Funktionsblock mit Parametern
- **REQ**: Service-Anfrage - Löst eine Abfrage des Eingabezustands aus

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung - Bestätigt die erfolgreiche Initialisierung
- **CNF**: Service-Bestätigung - Bestätigt die verarbeitete Service-Anfrage
- **IND**: Indikation von der Ressource - Signalisiert eingehende Ereignisse von der Hardware

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangs-Qualifier - Aktiviert/deaktiviert die Ereignisverarbeitung
- **PARAMS** (STRING): Service-Parameter - Konfigurationsparameter für den Service

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangs-Qualifier - Status der Ereignisverarbeitung
- **STATUS** (STRING): Service-Status - Rückmeldung über den aktuellen Betriebszustand

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der IE-Funktionsblock arbeitet als bidirektionale Schnittstelle für Eingabeereignisse. Bei der Initialisierung (INIT) werden die Service-Parameter konfiguriert. Über REQ-Ereignisse können gezielte Abfragen des Eingabezustands durchgeführt werden. Gleichzeitig kann der Block asynchron IND-Ereignisse generieren, wenn von der Hardware spontan Eingabeereignisse erkannt werden.

Technische Besonderheiten

- Unterstützt sowohl poll-basierte (REQ/CNF) als auch interrupt-basierte (IND) Betriebsmodi
- String-basierte Parameter- und Statusübertragung für flexible Konfiguration
- Separate Qualifier für Eingangs- und Ausgangsereignisse (QI/QO)
- Robuste Fehlerbehandlung durch STATUS-Rückmeldungen

Zustandsübersicht

Der Funktionsblock durchläuft folgende Hauptzustände:

1. **Nicht initialisiert**: Block wartet auf INIT-Ereignis
2. **Initialisiert**: Block ist betriebsbereit und kann REQ- und IND-Ereignisse verarbeiten
3. **Abfrage aktiv**: Verarbeitung einer REQ-Anfrage
4. **Indikation aktiv**: Verarbeitung eines spontanen Eingabeereignisses

Anwendungsszenarien

- Abfrage von digitalen Eingängen (z.B. Taster, Schalter)
- Überwachung von Sensorsignalen
- Schnittstelle zu externen Eingabegeräten
- Ereignisgesteuerte Steuerungsanwendungen
- Hardware-nahe E/A-Verwaltung in Automatisierungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren Eingabeblöcken bietet IE erweiterte Funktionalität:

- Gegenüber reinen E/A-Blöcken: Unterstützt sowohl poll- als auch event-basierte Abfragen
- Gegenüber statischen Eingabeblöcken: Dynamische Parametrierung zur Laufzeit
- Erweiterte Statusrückmeldungen für verbesserte Fehlerdiagnose

🛠️ Zugehörige Übungen

- Uebung_004a
- Uebung_004a2
- Uebung_004a2_2
- Uebung_004a2_3
- Uebung_004a2_AX
- Uebung_004a3
- Uebung_004a3_AX
- Uebung_004a4
- Uebung_004a4_AX
- Uebung_004a5
- Uebung_004a5_AX
- Uebung_004a6
- Uebung_004a6_AX
- Uebung_004a7
- Uebung_004a7_AX
- Uebung_004a8
- Uebung_004a8_AX

- Uebung_004a9
- Uebung_004a9_AX
- Uebung_004a_AX
- Uebung_004b
- Uebung_004b2
- Uebung_004b3
- Uebung_004b_AX
- Uebung_004b_AX_ASR
- Uebung_004b_AX_ASR_X
- Uebung_004c1
- Uebung_004c1_AX
- Uebung_004c2
- Uebung_004c2_AX
- Uebung_004c3
- Uebung_004c3_AX
- Uebung_004c4
- Uebung_004c4_AX
- Uebung_004c5
- Uebung_004c5_AX
- Uebung_006
- Uebung_006_AX
- Uebung_006a
- Uebung_006a2
- Uebung_006a2_AX
- Uebung_006a3
- Uebung_006a3_AX
- Uebung_006a4
- Uebung_006a4_AX
- Uebung_006a_AX
- Uebung_006b
- Uebung_006b_AX
- Uebung_006d
- Uebung_006d_AX
- Uebung_007a1
- Uebung_007a1_AX
- Uebung_007a2
- Uebung_007a2_AX
- Uebung_007a3
- Uebung_007a3_AX
- Uebung_009a
- Uebung_010b2
- Uebung_010b2_AX
- Uebung_010b3
- Uebung_010b3_AX
- Uebung_010b6
- Uebung_010b6_AX
- Uebung_010b7
- Uebung_010b7_AX
- Uebung_010b8
- Uebung_010b8_AX
- Uebung_010b9
- Uebung_010b9_AX
- Uebung_010bA
- Uebung_010bA2
- Uebung_010bA2_AX
- Uebung_010bA3
- Uebung_010bA3_AX
- Uebung_010bA4
- Uebung_010bA4_AX

- Uebung_010bA_AX
- Uebung_013
- Uebung_013_AX
- Uebung_014
- Uebung_015
- Uebung_015a
- Uebung_016
- Uebung_016a
- Uebung_017
- Uebung_018
- Uebung_018a
- Uebung_019
- Uebung_019a
- Uebung_019b
- Uebung_019c
- Uebung_020f3
- Uebung_020f3_AX
- Uebung_020h
- Uebung_020h_AX
- Uebung_020i
- Uebung_020i_AX
- Uebung_021
- Uebung_022
- Uebung_023
- Uebung_024
- Uebung_025
- Uebung_026
- Uebung_031
- Uebung_034b
- Uebung_035
- Uebung_035a
- Uebung_035a1_AX
- Uebung_035a1b_AX
- Uebung_035a2
- Uebung_035a2_AX
- Uebung_035a3
- Uebung_035a3_AX
- Uebung_035b
- Uebung_035c
- Uebung_036
- Uebung_037
- Uebung_038
- Uebung_038_AX
- Uebung_039
- Uebung_039a
- Softkey_T_FF_ILOCK_TO_QX_BG
- Uebung_040
- Uebung_040_2
- Uebung_040_AX
- Uebung_041
- Uebung_042
- Uebung_043
- Uebung_080
- Uebung_080b
- Uebung_080c
- Uebung_081
- Uebung_082
- Uebung_083
- Uebung_083_AX

- Uebung_084
- Uebung_085
- Uebung_087
- Uebung_087a1
- Uebung_091
- Uebung_093
- Uebung_093b
- Uebung_094
- Uebung_094a
- Uebung_094a_AX
- Uebung_095
- Uebung_095_AX
- Uebung_110
- Uebung_111
- Uebung_124
- Uebung_127
- Uebung_128
- Uebung_128b
- Uebung_132
- Uebung_150_AX
- Uebung_151_AX
- Uebung_152
- Uebung_153
- Uebung_160b
- Uebung_160b_AX
- Uebung_171_AX
- Uebung_179
- Uebung_180

Fazit

Der IE-Funktionsblock stellt eine flexible und robuste Lösung für die Behandlung von Eingabeereignissen in 4diac-basierten Steuerungssystemen dar. Seine Fähigkeit, sowohl synchrone Abfragen als auch asynchrone Indikationen zu verarbeiten, macht ihn besonders geeignet für Anwendungen, die sowohl reaktive als auch proaktive Eingabeverarbeitung erfordern.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

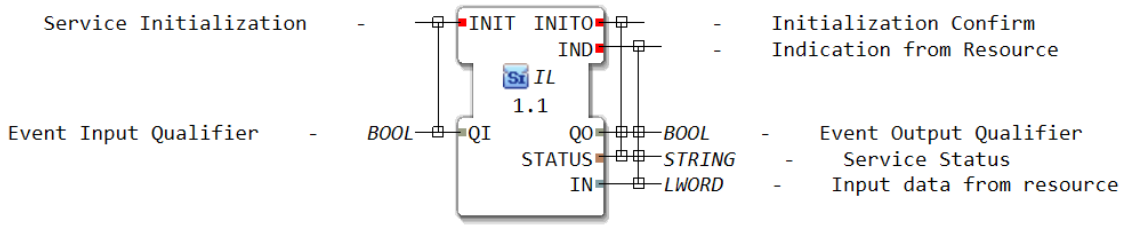
2.4.2.6 IL

📺 Video

- The silent failure

🎧 Podcast

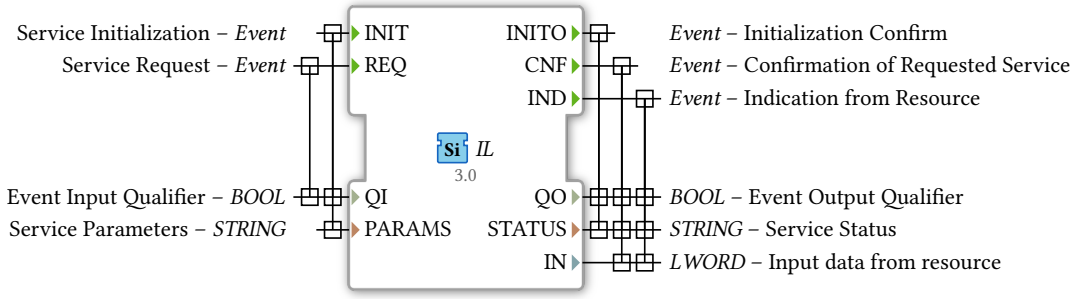
- 4diac IDE: Dein Open-Source-Werkzeugkasten für verteilte Industrieautomatisierung nach IEC 61499
- Eclipse 4diac 3.0: ST-Interpreter, FBE und 7200 Commits – Der Turbo für verteilte Automatisierung
- Eclipse 4diac: Ihr Fahrrad für die Industrie 4.0 – Software-Aufwand meistern und verteilte Systeme steuern
- IEC 61499: Die Zukunft der industriellen Automatisierung und verteilten Systeme
- Schritt 2 - 4diac Anwendungen verteilen



• – ◦ • – ◦ • – ◦ •

Einleitung

Der IL-Funktionsblock (Input Long Word) ist ein Service Interface Function Block für die Verarbeitung von Long Word-Eingabedaten. Er dient als Schnittstelle zwischen der Steuerungslogik und externen Eingabegeräten oder Ressourcen, die 64-Bit-Datenwerte bereitstellen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierungsereignis
- **REQ**: Service-Anfrageereignis

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
- **CNF**: Bestätigung der angeforderten Service-Operation
- **IND**: Indikation von der Ressource

Daten-Eingänge

- **QI** (BOOL): Event Input Qualifier - Steuert die Ausführung des Service
- **PARAMS** (STRING): Service-Parameter für die Konfiguration

Daten-Ausgänge

- **QO** (BOOL): Event Output Qualifier - Zeigt den Status der Operation an
- **STATUS** (STRING): Service-Statusinformationen
- **IN** (LWORD): Eingabedaten von der Ressource (64-Bit Wert)

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der IL-Block arbeitet als Eingabeservice-Interface für Long Word-Daten. Bei Empfang des INIT-Ereignisses wird der Service initialisiert und die Konfigurationsparameter werden über PARAMS übergeben. Das REQ-Ereignis löst eine Leseoperation aus, bei der Daten von der angeschlossenen Ressource gelesen werden. Der Block unterstützt sowohl angeforderte Lesevorgänge (CNF) als auch spontane Datenaktualisierungen (IND) von der Ressource.

Technische Besonderheiten

- Verarbeitet 64-Bit Long Word-Datentypen (LWORD)
- Unterstützt sowohl poll-basierte als auch event-basierte Datenabfragen

- Flexible Parameterkonfiguration über STRING-Parameter
- Statusrückmeldung über STATUS-Ausgang

Zustandsübersicht

Der Block durchläuft folgende Zustände:

1. **Nicht initialisiert**: Vor der INIT-Operation
2. **Initialisiert**: Nach erfolgreicher INIT-Operation, bereit für Datenabfragen
3. **Aktiv**: Während der Datenabfrage oder bei empfangenen Indikationen

Anwendungsszenarien

- Einlesen von 64-Bit Sensordaten
- Kommunikation mit externen Geräten, die Long Word-Daten liefern
- Datenerfassungssysteme mit hoher Präzision
- Steuerungssysteme, die große Zahlenwerte verarbeiten müssen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Eingabeblöcken wie IB (Byte), IW (Word) oder ID (Double Word) verarbeitet IL die größte Datengröße (64 Bit). Er bietet damit die höchste Präzision für numerische Eingabewerte in der 4diac-IO-Bibliothek.

Fazit

Der IL-Funktionsblock bietet eine robuste und flexible Schnittstelle für die Verarbeitung von 64-Bit Eingabedaten in IEC 61499-Steuerungssystemen. Seine Unterstützung für verschiedene Betriebsmodi (polling und event-driven) macht ihn vielseitig einsetzbar in verschiedenen industriellen Anwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

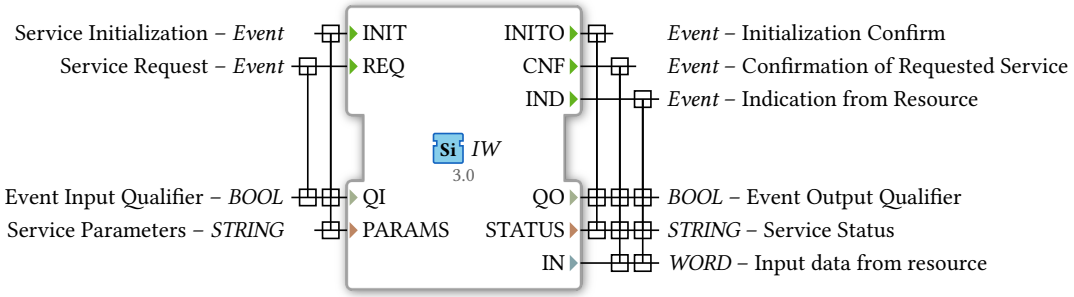
2.4.2.7 IW

🎧 Podcast

- Miniware TS101: Das mobile Löt-Multitalent – Stärken, Schwächen und die USB-C Revolution

Einleitung

Der IW-Funktionsblock (Input Word) ist ein Dienstschnittstellen-Funktionsblock für die Verarbeitung von Wort-Eingabedaten in 4diac-Systemen. Er dient als Schnittstelle zwischen der Steuerungslogik und externen Eingabegeräten oder Ressourcen, die Wort-Daten (16-Bit) bereitstellen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Dienstinitialisierung - Initialisiert den Funktionsblock und konfiguriert die Hardware-Parameter
- **REQ**: Dienstanforderung - Fordert eine aktuelle Eingabe vom angeschlossenen Gerät an

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung - Bestätigt den Abschluss der Initialisierung
- **CNF**: Bestätigung der angeforderten Dienstleistung - Quittiert eine erfolgreiche REQ-Anfrage

- **IND**: Indikation von der Ressource - Signalisiert eingehende Daten ohne vorherige Anfrage

Daten-Eingänge

- **QI**: Ereignis-Eingabe-Qualifizierer (BOOL) - Aktiviert/deaktiviert die Verarbeitung
- **PARAMS**: Dienstparameter (STRING) - Konfigurationsparameter für die Hardware-Schnittstelle

Daten-Ausgänge

- **QO**: Ereignis-Ausgabe-Qualifizierer (BOOL) - Status der Verarbeitung
- **STATUS**: Dienststatus (STRING) - Fehlermeldungen oder Statusinformationen
- **IN**: Eingabedaten von der Ressource (WORD) - Die gelesenen 16-Bit-Daten

Funktionsweise

Der IW-Block arbeitet als bidirektionale Schnittstelle für Wort-Eingabedaten. Bei der Initialisierung (INIT) werden die Hardware-Parameter konfiguriert. Der Block kann sowohl auf explizite Anfragen (REQ) reagieren als auch asynchrone Datenindikationen (IND) von der angeschlossenen Ressource verarbeiten. Die gelesenen WORD-Daten werden an den Ausgang IN ausgegeben.

Technische Besonderheiten

- Verarbeitet 16-Bit WORD-Datentypen
- Unterstützt sowohl anforderungsbasierte (REQ/CNF) als auch ereignisbasierte (IND) Kommunikation
- String-basierte Parameterkonfiguration für flexible Hardware-Anbindung
- Umfassende Statusrückmeldung über STATUS-Ausgang

Zustandsübersicht

Der Block durchläuft folgende Hauptzustände:

- **Nicht initialisiert**: Block wartet auf INIT-Ereignis
- **Initialisiert**: Block ist betriebsbereit und kann Daten verarbeiten
- **Datenanforderung**: Verarbeitung einer REQ-Anfrage
- **Datenempfang**: Verarbeitung eingehender IND-Daten

Anwendungsszenarien

- Anbindung von 16-Bit Eingabegeräten (Sensoren, Encoder, Schaltergruppen)
- Lesen von digitalen Eingangswörtern aus Prozessperipherie
- Schnittstelle zu IO-Modulen mit Wort-basierten Eingängen
- Überwachung von Statuswörtern aus verteilten Systemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu IB (Input Byte) verarbeitet IW 16-Bit anstelle von 8-Bit Daten. Gegenüber ID (Input Double Word) bietet IW eine geringere Datenbreite. Der Block ähnelt in der Struktur anderen IO-Schnittstellenblöcken, ist jedoch speziell auf WORD-Datentypen optimiert.

Fazit

Der IW-Funktionsblock bietet eine robuste und flexible Schnittstelle für die Verarbeitung von 16-Bit Eingabedaten in 4diac-Systemen. Seine klare Ereignisstruktur und umfassende Statusrückmeldung machen ihn zu einer zuverlässigen Lösung für die Anbindung von Wort-basierten Eingabegeräten in industriellen Automatisierungsanwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

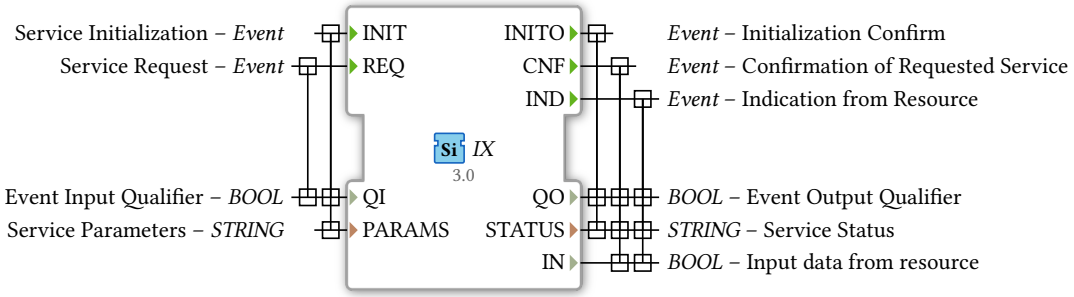
2.4.2.8 IX

🎧 Podcast

- Infineon MOTIX BTM9020/9021EP: Datenblatt-Analyse für Automotive – Robuster Motortreiber mit intelligenter Diagnose (HW vs. SPI)
- integrierten Vollbrücken-ICs MOTIX™ BTM9020EP

Einleitung

Der IX-Funktionsblock ist ein Service-Interface-Funktionsblock für boolesche Eingabedaten. Er dient als Schnittstelle zur Kommunikation mit Eingabegeräten und ermöglicht die Abfrage und Verarbeitung von digitalen Eingangssignalen in 4diac-Systemen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung - Initialisiert den Funktionsblock und konfiguriert die Hardware-Parameter
- **REQ**: Service-Anfrage - Fordert eine Abfrage des aktuellen Eingangswerts an

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung - Bestätigt die erfolgreiche Initialisierung
- **CNF**: Bestätigung der angeforderten Service - Bestätigt eine erfolgreiche Abfrage
- **IND**: Indikation von der Ressource - Signalisiert eine Zustandsänderung des Eingangssignals

Daten-Eingänge

- **QI**: Event-Input-Qualifier (BOOL) - Steuert die Aktivierung der Service-Funktionalität
- **PARAMS**: Service-Parameter (STRING) - Enthält Konfigurationsparameter für die Hardware-Schnittstelle

Daten-Ausgänge

- **QO**: Event-Output-Qualifier (BOOL) - Zeigt den Status der Service-Ausführung an
- **STATUS**: Service-Status (STRING) - Liefert Statusinformationen über die Service-Ausführung
- **IN**: Eingabedaten von der Ressource (BOOL) - Enthält den aktuellen Wert des digitalen Eingangs

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der IX-Funktionsblock dient als Vermittler zwischen der Steuerungslogik und physischen Eingabegeräten. Bei der Initialisierung (INIT) werden die Hardware-Parameter konfiguriert. Anschließend kann über REQ-Ereignisse der aktuelle Zustand des Eingangs abgefragt werden. Der Block kann sowohl poll-basierte Abfragen (REQ/CNF) als auch ereignisbasierte Benachrichtigungen (IND) bei Zustandsänderungen verarbeiten.

Technische Besonderheiten

- Unterstützt sowohl anforderungsbasierte als auch ereignisgesteuerte Betriebsmodi
- Boolescher Datentyp für einfache digitale Eingänge
- Flexible Parameterkonfiguration über STRING-Parameter
- Umfassende Statusrückmeldung für Fehlerdiagnose

Zustandsübersicht

Der Funktionsblock durchläuft folgende Hauptzustände:

1. **Nicht initialisiert**: Block ist inaktiv
2. **Initialisiert**: Block ist betriebsbereit nach erfolgreicher INIT-Verarbeitung
3. **Abfrage aktiv**: Verarbeitung einer REQ-Anfrage

4. **Indikationsbereit**: Bereit für ereignisgesteuerte Benachrichtigungen

Anwendungsszenarien

- Abfrage von digitalen Sensoren (Endschalter, Näherungsschalter)
- Überwachung von Taster-Eingängen
- Lesen von Schalterstellungen
- Digitale Signalverarbeitung in Automatisierungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Eingabe-Bausteinen bietet IX eine erweiterte Service-Schnittstelle mit sowohl poll-basierten als auch ereignisgesteuerten Betriebsmodi. Während einfachere Eingabeblöcke oft nur direkte Werte liefern, bietet IX zusätzliche Statusinformationen und Fehlerbehandlung.

🛠️ Zugehörige Übungen

- Uebung_001
- Uebung_001c
- Uebung_002
- Uebung_002a
- Uebung_002a2
- Uebung_002a3
- Uebung_002a4
- Uebung_002a5b
- Uebung_002b2
- Uebung_002b3
- Uebung_003
- Uebung_003a0
- Uebung_003a_sub
- Funk_IX_TO_DataPanel_MI_QX
- IX_TO_Aux_QX
- Uebung_003d
- Uebung_005
- Uebung_006e1
- Uebung_006e2
- Uebung_010
- Uebung_010a
- Uebung_010a2
- Uebung_010a3
- Uebung_010a4
- Uebung_010b1
- Uebung_010c
- Uebung_010c2
- Uebung_019c
- Uebung_020a
- Uebung_020b
- Uebung_020c
- Uebung_020c2
- Uebung_020c3
- Uebung_020d
- Uebung_020e
- Uebung_020e2
- Uebung_020f
- Uebung_020f2
- Uebung_020g
- Uebung_020i
- Uebung_028
- Uebung_029
- Uebung_030
- Uebung_032
- IX_TO_logiBUS_LED_strip_QX

- SoftkeyOverride_TO_QX_BG
- Uebung_039b
- Uebung_049
- Uebung_051
- Uebung_052
- Uebung_053
- Uebung_054
- Uebung_055
- Uebung_056
- Uebung_085
- Uebung_086
- Uebung_087
- Uebung_087a1
- Uebung_087a2
- Uebung_088
- Uebung_089
- Uebung_090a1
- Uebung_090a1_AX
- Uebung_090a2
- Uebung_090a2_AX
- Uebung_094
- Uebung_094a
- Uebung_095
- Uebung_160
- Uebung_160b2
- Uebung_177
- Uebung_178

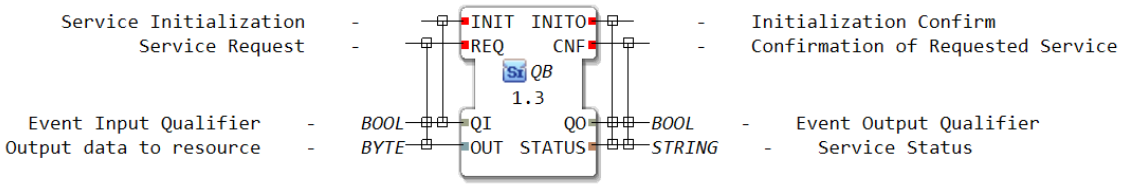
Fazit

Der IX-Funktionsblock stellt eine robuste und flexible Lösung für die Integration boolescher Eingabedaten in 4diac-Systeme dar. Seine umfassende Fehlerbehandlung und flexible Betriebsmodi machen ihn besonders geeignet für zuverlässige Automatisierungsanwendungen mit digitalen Eingangssignalen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

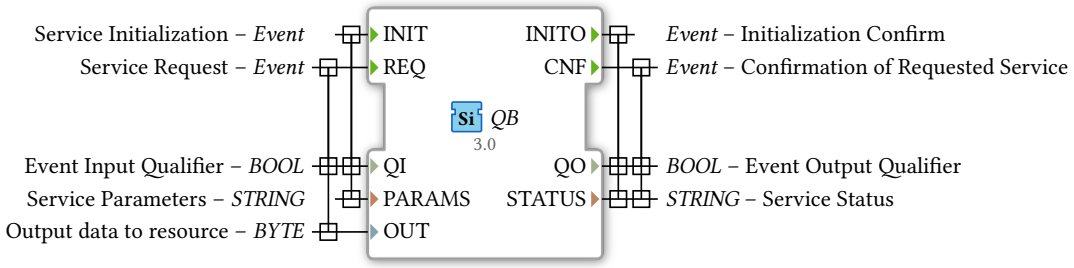
2.4.2.9 QB



• – ◦ • – ◦ • – ◦ •

Einleitung

Der QB-Funktionsblock ist ein Ausgabeservice-Interface für Byte-Ausgabedaten. Er dient als Schnittstelle zwischen der Steuerungslogik und physischen Ausgabegeräten oder Ressourcen und ermöglicht die Ausgabe von Byte-Daten an externe Hardware.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierungsereignis
- **REQ**: Service-Anfrageereignis

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
- **CNF**: Bestätigung der angeforderten Service-Operation

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangsqualifizierer
- **PARAMS** (STRING): Service-Parameter für Konfiguration
- **OUT** (BYTE): Ausgabedaten für die Ressource

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifizierer
- **STATUS** (STRING): Service-Statusinformationen

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der QB-Block initialisiert sich über das INIT-Ereignis mit den entsprechenden PARAMS. Nach erfolgreicher Initialisierung bestätigt er dies mit INITO. Über das REQ-Ereignis können Byte-Daten (OUT) an die angeschlossene Ressource ausgegeben werden. Die Verarbeitung wird mit dem CNF-Ereignis und Statusinformationen quittiert.

Technische Besonderheiten

- Spezialisiert auf Byte-Datenausgabe
- Unterstützt parametrierbare Initialisierung
- Liefert detaillierte Statusinformationen über STRING-Ausgang
- Verwendet Qualifizierer (QI/QO) für Ereignissteuerung

Zustandsübersicht

1. **Nicht initialisiert**: Block wartet auf INIT-Ereignis
2. **Initialisiert**: Block bereit für REQ-Anfragen
3. **Aktiv**: Verarbeitet Ausgabeanfragen
4. **Fehler**: Statusmeldung bei Problemen

Anwendungsszenarien

- Ansteuerung von digitalen Ausgabemodulen
- Kommunikation mit Byte-basierten Peripheriegeräten
- Integration in IO-Subsysteme
- Steuerung von Hardware mit Byte-Kommandos

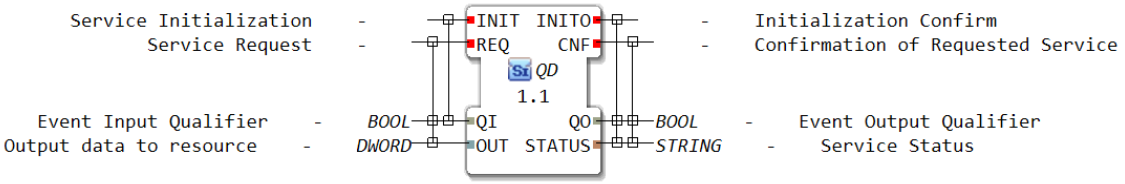
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu allgemeineren Ausgabeblöcken ist QB speziell auf Byte-Daten optimiert. Andere Blöcke wie z.B. WORD- oder DWORD-Ausgabeblöcke arbeiten mit größeren Datentypen, während QB die effiziente Verarbeitung von 8-Bit-Daten ermöglicht.

Fazit

Der QB-Funktionsblock bietet eine robuste und standardisierte Schnittstelle für Byte-Ausgabedaten in industriellen Steuerungssystemen. Durch seine klare Ereignissteuerung und Statusrückmeldungen eignet er sich besonders für zuverlässige IO-Operationen in automatisierten Prozessen.

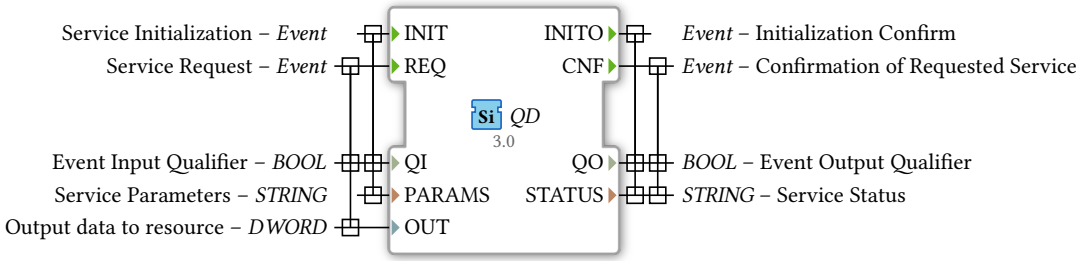
2.4.2.10 QD



• – ◦ • – ◦ • – ◦ •

Einleitung

Der QD-Funktionsblock ist ein Ausgabeservice-Interface-Baustein für die Ausgabe von Doppelwort-Daten (DWORD). Er dient als Schnittstelle zwischen der Anwendungslogik und den physikalischen Ausgabegeräten eines Ressourcen-Managers.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung - initialisiert den Funktionsblock mit den erforderlichen Parametern
- **REQ**: Service-Anforderung - löst die Datenausgabe an die Ressource aus

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung - bestätigt den Abschluss der Initialisierung
- **CNF**: Bestätigung der angeforderten Service-Ausführung - signalisiert den Abschluss der Datenausgabe

Daten-Eingänge

- **QI**: Event-Input-Qualifier (BOOL) - steuert die Ausführung der Service-Anforderungen
- **PARAMS**: Service-Parameter (STRING) - Konfigurationsparameter für den Service
- **OUT**: Ausgabedaten (DWORD) - die an die Ressource zu übertragenden Doppelwort-Daten

Daten-Ausgänge

- **QO**: Event-Output-Qualifier (BOOL) - Status der Service-Ausführung
- **STATUS**: Service-Status (STRING) - detaillierte Statusinformationen über die Service-Ausführung

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der QD-Block arbeitet als zyklischer Service-Interface-Baustein. Bei Empfang des INIT-Ereignisses wird der Block mit den über PARAMS übergebenen Konfigurationsdaten initialisiert. Die eigentliche Datenausgabe wird durch das REQ-Ereignis ausgelöst, wobei die im OUT-Eingang bereitgestellten DWORD-Daten an die zugeordnete Ressource übertragen werden. Jede Service-Anforderung wird durch entsprechende Bestätigungsereignisse (INITO bzw. CNF) quittiert.

Technische Besonderheiten

- Verarbeitet 32-Bit-Daten (DWORD)
- Unterstützt parametrierbare Service-Konfiguration
- Bietet detaillierte Statusrückmeldungen
- Arbeitet mit Qualifier-Flags für zuverlässige Service-Steuerung

Zustandsübersicht

Der Block durchläuft folgende Zustände:

1. **Nicht initialisiert**: Block wartet auf INIT-Ereignis
2. **Initialisiert**: Block ist betriebsbereit und wartet auf REQ-Ereignisse
3. **Service-Ausführung**: Verarbeitung einer REQ-Anforderung
4. **Bestätigung**: Rückmeldung des Service-Ergebnisses

Anwendungsszenarien

- Ansteuerung von digitalen Ausgabemodulen mit 32-Bit-Datenbreite
- Steuerung von komplexen Aktoren mit Mehrfach-Parameterübergabe
- Integration in verteilte Steuerungssysteme mit Ressourcen-Management
- Anwendungen mit hohen Anforderungen an die Ausgabedatenintegrität

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren Ausgabeblöcken bietet QD:

- Erweiterte Parametrierbarkeit durch STRING-Parameter
- DWORD-Datenbreite statt einfacher BOOL- oder BYTE-Werte
- Detaillierte Statusrückmeldungen für Fehlerdiagnose
- Vollständige Service-Interface-Charakteristik mit Initialisierungsprozedur

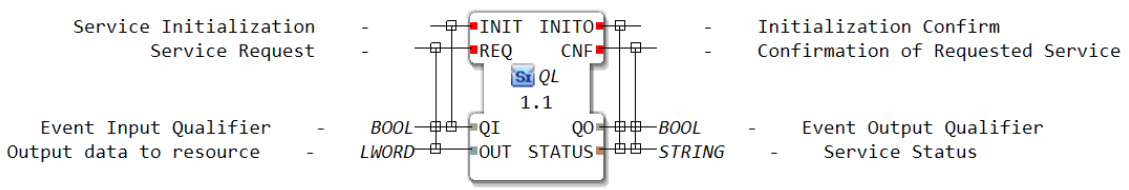
🛠️ Zugehörige Übungen

- Uebung_060

Fazit

Der QD-Funktionsblock stellt eine leistungsstarke und flexible Lösung für die Ausgabe von Doppelwort-Daten in IEC 61499-Systemen dar. Seine umfassende Parametrierbarkeit und detaillierte Statusrückmeldungen machen ihn besonders geeignet für anspruchsvolle Automatisierungsanwendungen, die zuverlässige und konfigurierbare Ausgabeservices erfordern.

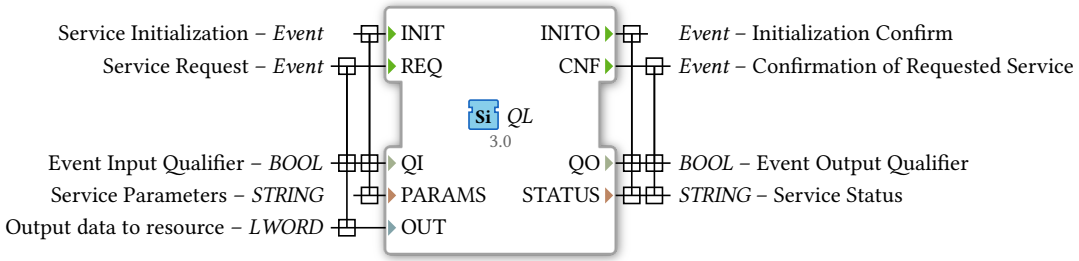
2.4.2.11 QL



• – ◦ • – ◦ • – ◦ •

Einleitung

Der QL-Funktionsblock ist ein Ausgabeservice-Interface-Baustein für die Verarbeitung von Long-Word-Ausgabedaten. Er dient als Schnittstelle zwischen der Steuerungslogik und externen Ressourcen und ermöglicht die Ausgabe von 64-Bit-Datenwerten.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierungsereignis
- **REQ**: Service-Anfrageereignis

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
- **CNF**: Bestätigung der angeforderten Service-Operation

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangsqualifizierer
- **PARAMS** (STRING): Service-Parameter
- **OUT** (LWORD): Ausgabedaten für die Ressource (64-Bit)

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifizierer
- **STATUS** (STRING): Service-Statusinformation

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der QL-Block verarbeitet zwei Hauptoperationen: Initialisierung und Datenausgabe. Bei der INIT-Operation werden die Service-Parameter konfiguriert und die Verbindung zur Ressource initialisiert. Die REQ-Operation führt die eigentliche Datenausgabe des LWORD-Wertes an die angeschlossene Ressource durch. Jede Operation wird durch entsprechende Bestätigungsereignisse quittiert.

Technische Besonderheiten

- Verarbeitung von 64-Bit-Datenwerten (LWORD)
- String-basierte Parameterkonfiguration
- Statusrückmeldung über STRING-Variable
- Qualifizierer für Ereignissteuerung (QI/QO)

Zustandsübergänge

Der Block verfügt über zwei Hauptzustände:

1. **Initialisierungszustand**: Verarbeitung der INIT-Anfrage mit PARAMS
2. **Betriebszustand**: Verarbeitung von REQ-Anfragen mit OUT-Daten

Anwendungsszenarien

- Ausgabe von 64-Bit-Steuerwerten an Hardware-Schnittstellen
- Kommunikation mit externen Peripheriegeräten
- Integration in verteilte Steuerungssysteme
- Anbindung an spezielle IO-Module

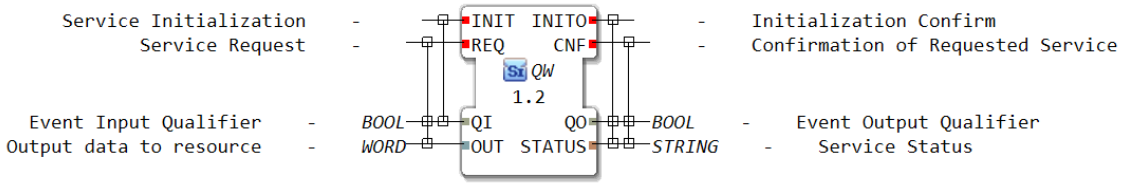
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren Ausgabeblöcken bietet QL die Verarbeitung von 64-Bit-Daten und eine erweiterte Parameterkonfiguration. Gegenüber universelleren Kommunikationsblöcken ist QL speziell auf die Ausgabefunktionalität optimiert.

Fazit

Der QL-Funktionsblock stellt eine leistungsfähige Schnittstelle für die Ausgabe von Long-Word-Daten dar und bietet durch seine strukturierte Ereignissteuerung und Statusrückmeldung eine zuverlässige Grundlage für die Anbindung externer Ressourcen in Automatisierungssystemen.

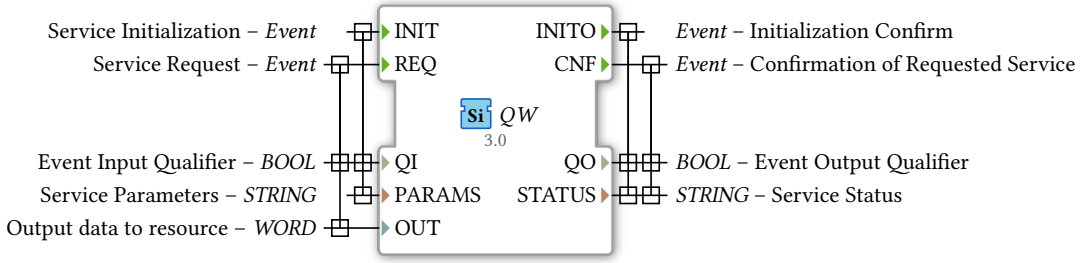
2.4.2.12 QW



• – ◦ • – ◦ • – ◦ •

Einleitung

Der QW-Funktionsblock ist ein Ausgabeservice-Interface-Baustein für die Ausgabe von Wort-Daten (WORD). Er dient als Schnittstelle zwischen der Steuerungslogik und physikalischen Ausgabegeräten oder Ressourcen und ermöglicht die sichere Übertragung von 16-Bit-Daten an externe Hardware.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierungsereignis
- **REQ**: Service-Anfrageereignis

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
- **CNF**: Bestätigung der angeforderten Service-Operation

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangsqualifizierer
- **PARAMS** (STRING): Service-Parameter für Konfiguration
- **OUT** (WORD): Ausgabedaten für die Ressource

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifizierer
- **STATUS** (STRING): Service-Statusinformationen

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der QW-Block initialisiert bei Empfang des INIT-Ereignisses die Hardware-Schnittstelle mit den übergebenen PARAMS. Nach erfolgreicher Initialisierung bestätigt er dies mit INITO. Bei einer REQ-Anfrage werden die an OUT anliegenden WORD-Daten an die konfigurierte Ressource ausgegeben. Die Operation wird mit CNF bestätigt, wobei STATUS Informationen über den Erfolg oder Fehler der Operation liefert.

Technische Besonderheiten

- Verarbeitet 16-Bit WORD-Daten
- String-basierte Parameterkonfiguration über PARAMS
- Statusrückmeldung über STRING-Variable
- Qualifier-Bits (QI/QO) für Ereignissteuerung

Zustandsübersicht

1. **Nicht initialisiert**: Block wartet auf INIT-Ereignis
2. **Initialisiert**: Block ist betriebsbereit und wartet auf REQ-Ereignisse
3. **Aktiv**: Verarbeitet Ausgabeanfragen und generiert Bestätigungen

Anwendungsszenarien

- Ansteuerung von digitalen Ausgabemodulen
- Kommunikation mit I/O-Systemen
- Steuerung von Hardware mit 16-Bit-Datenworten
- Integration in verteilte Steuerungssysteme

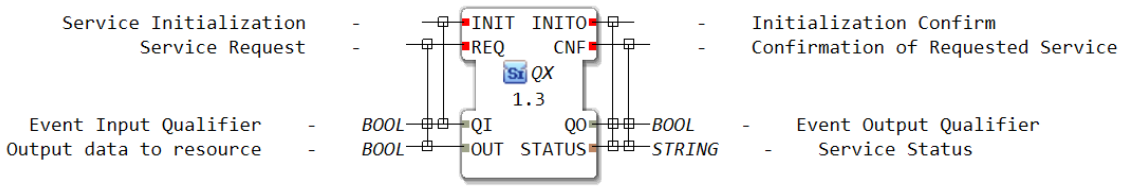
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu QB (Byte-Ausgabe) verarbeitet QW 16-Bit statt 8-Bit Daten. Gegenüber QD (Double-Word) bietet er eine geringere Datenbreite bei reduziertem Ressourcenbedarf. Die Struktur ähnelt anderen IO-Service-Blöcken, ist jedoch spezifisch auf WORD-Daten ausgelegt.

Fazit

Der QW-Funktionsblock bietet eine robuste und standardisierte Schnittstelle für die Ausgabe von 16-Bit-Daten an externe Ressourcen. Durch seine klare Ereignissteuerung und Statusrückmeldung eignet er sich ideal für zuverlässige Echtzeit-I/O-Operationen in industriellen Steuerungssystemen.

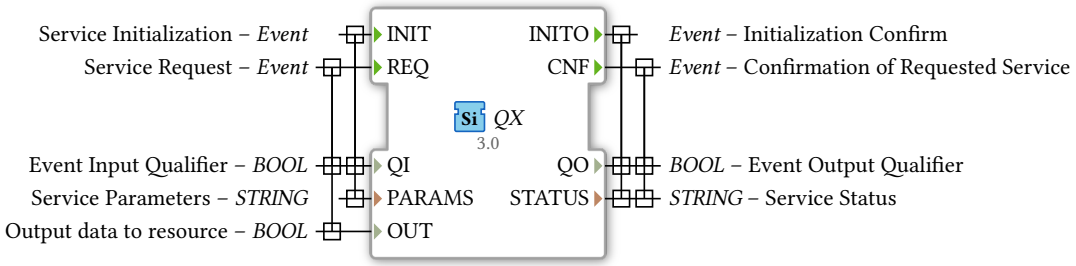
2.4.2.13 QX



• – ◦ • – ◦ • – ◦ •

Einleitung

Der QX-Funktionsblock ist ein Ausgabeservice-Schnittstellen-Funktionsblock für boolesche Ausgabedaten. Er dient als Schnittstelle zwischen der Steuerungslogik und physikalischen Ausgabegeräten und ermöglicht die Ausgabe von digitalen Signalen an externe Ressourcen.



Schnittstellenstruktur

Ereignis-Eingänge
- **INIT**: Service-Initialisierungsereignis
- **REQ**: Service-Anfrageereignis

Ereignis-Ausgänge
- **INITO**: Initialisierungsbestätigung
- **CNF**: Bestätigung der angeforderten Service-Operation

Daten-Eingänge
- **QI** (BOOL): Ereignis-Eingangsqualifizierer
- **PARAMS** (STRING): Service-Parameter für Konfiguration
- **OUT** (BOOL): Ausgabedaten für die Ressource

Daten-Ausgänge
- **QO** (BOOL): Ereignis-Ausgangsqualifizierer
- **STATUS** (STRING): Service-Statusinformationen

Adapter
Keine Adapter-Schnittstellen vorhanden.

Funktionsweise
Der QX-Block verarbeitet zwei Hauptoperationen: Initialisierung und Service-Anfragen. Bei INIT-Ereignis wird der Service mit den angegebenen PARAMS initialisiert. Bei REQ-Ereignis wird der boolesche OUT-Wert an die angeschlossene Ressource ausgegeben. Jede Operation generiert eine entsprechende Bestätigung (INITO oder CNF) mit Statusinformationen.

Technische Besonderheiten
- Spezialisiert auf boolesche Ausgabedaten
- Unterstützt parametrisierbare Service-Konfiguration
- Liefert detaillierte Statusinformationen über den STRING-Ausgang STATUS
- Verwendet Qualifizierer (QI/QO) für Ereignissteuerung

Zustandsübersicht
Der Block wechselt zwischen folgenden Zuständen:

1. **Nicht initialisiert**: Vor der ersten INIT-Operation
2. **Initialisiert**: Nach erfolgreicher INIT-Operation, bereit für REQ-Operationen
3. **Aktiv**: Während der Verarbeitung von REQ-Operationen

Anwendungsszenarien
- Ansteuerung von digitalen Ausgängen (Relais, LEDs, Ventile)
- Schnittstelle zu Aktoren in Automatisierungssystemen
- Integration in IO-Subsysteme für verteilte Steuerungen
- Test- und Simulationsumgebungen für Ausgabesignale

⚖️ Vergleich mit ähnlichen Bausteinen
Im Vergleich zu generischen Ausgabeblöcken bietet QX spezifische Service-Schnittstellenfunktionalität mit parametrisierbarer Konfiguration und detailliertem Status-Reporting. Andere Blöcke wie einfache BOOL-Ausgaben haben typischerweise weniger Konfigurationsmöglichkeiten und Statusinformationen.

🛠️ Zugehörige Übungen
- Uebung_001
- Uebung_001c
- Uebung_002
- Uebung_002a
- Uebung_002a2
- Uebung_002a3

- Uebung_002a4
- Uebung_002a5b
- Uebung_002b2
- Uebung_002b3
- Uebung_003
- Uebung_003a0
- Uebung_003a_sub
- Funk_IX_TO_DataPanel_MI_QX
- IX_TO_Aux_QX
- Uebung_003d
- Uebung_004a
- Uebung_004a2
- Uebung_004a2_2
- Uebung_004a2_3
- Uebung_004a3
- Uebung_004a4
- Uebung_004a5
- Uebung_004a6
- Uebung_004a7
- Uebung_004a8
- Uebung_004a9
- Uebung_004b
- Uebung_004b2
- Uebung_004b3
- Uebung_004c1
- Uebung_004c2
- Uebung_004c3
- Uebung_004c4
- Uebung_004c5
- Uebung_004c6
- Uebung_004c7
- Uebung_005
- Uebung_006
- Uebung_006a
- Uebung_006a2
- Uebung_006a3
- Uebung_006a4
- Uebung_006b
- Uebung_006c
- Uebung_006d
- Uebung_006e1
- Uebung_006e2
- Uebung_007
- Uebung_007a1
- Uebung_007a2
- Uebung_007a3
- Uebung_008
- Uebung_009
- Uebung_010
- Uebung_010a
- Uebung_010a2
- Uebung_010a3
- Uebung_010a4
- Uebung_010b1
- Uebung_010b2
- Uebung_010b3
- Uebung_010b6
- Uebung_010b7
- Uebung_010b8

- Uebung_010b9
- Uebung_010bA
- Uebung_010bA2
- Uebung_010bA3
- Uebung_010bA4
- Uebung_010c
- Uebung_010c2
- Uebung_013
- Uebung_019b
- Uebung_019c
- Uebung_020a
- Uebung_020b
- Uebung_020c
- Uebung_020c2
- Uebung_020c3
- Uebung_020d
- Uebung_020e
- Uebung_020e2
- Uebung_020f
- Uebung_020f2
- Uebung_020f3
- Uebung_020g
- Uebung_020h
- Uebung_020i
- Uebung_021
- Uebung_022
- Uebung_023
- Uebung_024
- Uebung_025
- SR_TO_QX_TRIG_EO
- Uebung_028
- Uebung_029
- Uebung_030
- Uebung_032
- IX_TO_logiBUS_LED_strip_QX
- Uebung_035
- Uebung_035a
- Uebung_035a2
- Uebung_035a3
- Uebung_035b
- Uebung_035c
- Uebung_036
- Uebung_037
- Uebung_038
- SoftkeyOverride_TO_QX_BG
- Softkey_T_FF_ILOCK_TO_QX_BG
- Uebung_039b
- Uebung_040
- Uebung_040_2
- Uebung_041
- Uebung_049
- Uebung_051
- Uebung_052
- Uebung_053
- Uebung_054
- Uebung_055
- Uebung_056
- Bool8_TO_logiBUS_QX8
- Uebung_071

- Uebung_071a
- Uebung_071b
- Uebung_072b
- Uebung_080
- Uebung_080b
- Uebung_080c
- Uebung_081
- Uebung_082
- Uebung_083
- Uebung_084
- Uebung_085
- Uebung_087
- Uebung_087a1
- Uebung_087a2
- Uebung_088
- Uebung_089
- Uebung_090a1
- Uebung_090a2
- Uebung_091
- Uebung_093
- Uebung_093b
- Uebung_094
- Uebung_094a
- Uebung_095
- Uebung_110
- Uebung_111
- Uebung_160
- Uebung_160b
- Uebung_160b2
- Uebung_177
- Uebung_178
- Uebung_179
- Uebung_180

Fazit

Der QX-Funktionsblock stellt eine robuste und konfigurierbare Lösung für boolesche Ausgabeservices dar. Seine strukturierte Ereignisbehandlung und detaillierte Statusrückmeldung machen ihn besonders geeignet für zuverlässige Automatisierungsanwendungen, bei denen Transparenz über den IO-Status erforderlich ist.

2.4.2.14 ADS

🎧 Podcast

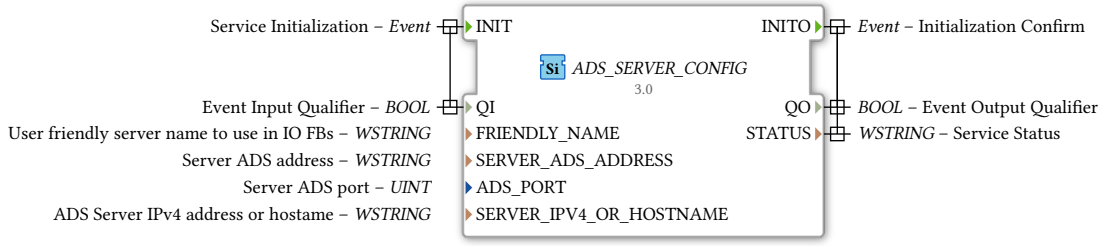
- Germany's Green Energy Crossroads: Navigating the End of Renewable Subsidies with Innovative Solutions

ADS_SERVER_CONFIG

• – ◦ • – ◦ • – ◦ •

Einleitung

Der ADS_SERVER_CONFIG Funktionsblock dient zur Konfiguration von ADS-Server-Verbindungen in 4diac-Systemen. Er ermöglicht die Initialisierung und Parametrisierung von Verbindungseinstellungen für ADS-Kommunikation, die in nachgelagerten IO-Funktionsblöcken verwendet werden kann.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierungsereignis - Startet die Konfiguration des ADS-Servers

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung - Signalisiert den erfolgreichen Abschluss der Konfiguration

Daten-Eingänge

- **QI** (BOOL): Ereigniseingangsqualifizierer - Aktiviert/deaktiviert die Initialisierung
- **FRIENDLY_NAME** (WSTRING): Benutzerfreundlicher Servername für Verwendung in IO-Funktionsblöcken
- **SERVER_ADS_ADDRESS** (WSTRING): ADS-Adresse des Servers
- **ADS_PORT** (UINT): ADS-Port des Servers
- **SERVER_IPV4_OR_HOSTNAME** (WSTRING): IPv4-Adresse oder Hostname des ADS-Servers

Daten-Ausgänge

- **QO** (BOOL): Ereignisausgangsqualifizierer - Zeigt den Status der Initialisierung an
- **STATUS** (WSTRING): Service-Statusinformationen

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Funktionsblock wird durch das INIT-Ereignis aktiviert und verarbeitet die bereitgestellten Konfigurationsparameter. Bei erfolgreicher Verarbeitung gibt er das INITO-Ereignis aus und stellt die konfigurierten Parameter für nachgelagerte ADS-IO-Funktionsblöcke bereit. Der STATUS-Ausgang liefert detaillierte Informationen über den Konfigurationsprozess.

Technische Besonderheiten

- Verwendet WSTRING-Datentypen für internationale Zeichenunterstützung
- Unterstützt sowohl IPv4-Adressen als auch Hostnamen für maximale Flexibilität
- Separate Parameter für ADS-Adresse und Netzwerkadresse ermöglichen komplexe Netzwerkkonfigurationen
- Port-Konfiguration als UINT für numerische Port-Spezifikation

Zustandsübersicht

1. **Inaktiv**: Wartet auf INIT-Ereignis
2. **Konfiguration**: Verarbeitet Eingangsparameter
3. **Aktiv**: Konfiguration erfolgreich abgeschlossen, Parameter stehen zur Verfügung
4. **Fehler**: Konfiguration fehlgeschlagen, STATUS enthält Fehlerinformationen

Anwendungsszenarien

- Initiale Systemkonfiguration in ADS-basierten Automatisierungssystemen
- Zentrale Verwaltung von ADS-Server-Verbindungsparametern
- Dynamische Rekonfiguration von Netzwerkeinstellungen
- Integration in größere IO-Management-Systeme

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen ADS-IO-Blöcken bietet ADS_SERVER_CONFIG eine zentrale Konfigurationsmöglichkeit für mehrere nachgelagerte Blöcke. Er separiert die Konfigurationslogik von der eigentlichen Datenkommunikation und ermöglicht so wiederverwendbare Server-Einstellungen.

Fazit

ADS_SERVER_CONFIG ist ein essentieller Baustein für strukturierte ADS-Kommunikation in 4diac-Systemen. Durch die zentrale Konfigurationsverwaltung vereinfacht er die Wartung und Änderung von Netzwerkeinstellungen und fördert konsistente Verbindungsparameter über verschiedene IO-Funktionsblöcke hinweg.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

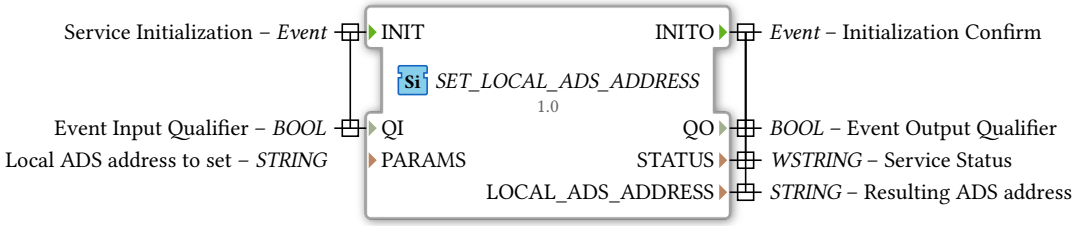
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SET_LOCAL_ADS_ADDRESS

• – ◦ • – ◦ • – ◦ •

Einleitung

Der SET_LOCAL_ADS_ADDRESS Funktionsblock ist ein Service Interface Function Block zur Konfiguration der lokalen ADS-Adresse. ADS (Automation Device Specification) ist ein Kommunikationsprotokoll, das häufig in der Automatisierungstechnik verwendet wird. Dieser Baustein ermöglicht die dynamische Einstellung der lokalen ADS-Adresse während der Laufzeit.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service Initialisierung - Startet den Konfigurationsprozess für die lokale ADS-Adresse

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung - Signalisiert den Abschluss des Konfigurationsvorgangs

Daten-Eingänge

- **QI** (BOOL): Event Input Qualifier - Aktiviert/deaktiviert den Initialisierungsvorgang
- **PARAMS** (STRING): Lokale ADS-Adresse, die gesetzt werden soll

Daten-Ausgänge

- **QO** (BOOL): Event Output Qualifier - Bestätigt den erfolgreichen Abschluss der Operation
- **STATUS** (WSTRING): Service Status - Enthält Statusinformationen über den durchgeführten Vorgang
- **LOCAL_ADS_ADDRESS** (STRING): Resultierende ADS-Adresse - Die tatsächlich gesetzte ADS-Adresse

Funktionsweise

Der Funktionsblock wird durch das INIT-Ereignis aktiviert. Bei Aktivierung liest er den PARAMS-Eingang, der die gewünschte lokale ADS-Adresse im STRING-Format enthält. Der Baustein konfiguriert daraufhin die lokale ADS-Schnittstelle mit der angegebenen Adresse. Nach Abschluss des Vorgangs wird das INITO-Ereignis ausgelöst und die resultierende ADS-Adresse sowie Statusinformationen an den Ausgängen bereitgestellt.

Technische Besonderheiten

- Verwendet STRING-Datentyp für die ADS-Adressen

- Bietet umfangreiche Statusinformationen im WSTRING-Format
- Unterstützt dynamische Adresskonfiguration zur Laufzeit
- Implementiert ein standardisiertes Initialisierungsprotokoll

Zustandsübergänge

1. **Initialzustand**: Wartet auf INIT-Ereignis
2. **Konfigurationszustand**: Verarbeitet die ADS-Adressen-Parameter
3. **Abschlusszustand**: Sendet INITO-Ereignis mit Ergebnisdaten

Anwendungsszenarien

- Dynamische Konfiguration von ADS-Geräten in verteilten Automatisierungssystemen
- Adressierung von ADS-Komponenten in flexiblen Produktionsanlagen
- Integration in Systeme mit variablen Netzwerkkonfigurationen
- Wartung und Inbetriebnahme von ADS-basierten Steuerungen

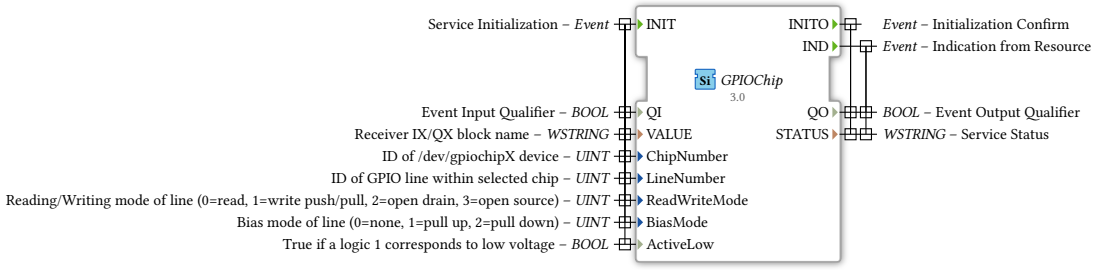
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu statischen ADS-Konfigurationsbausteinen ermöglicht SET_LOCAL_ADS_ADDRESS eine dynamische Adressverwaltung während der Laufzeit. Der Baustein bietet zudem detaillierte Statusrückmeldungen, was die Fehlerdiagnose erleichtert.

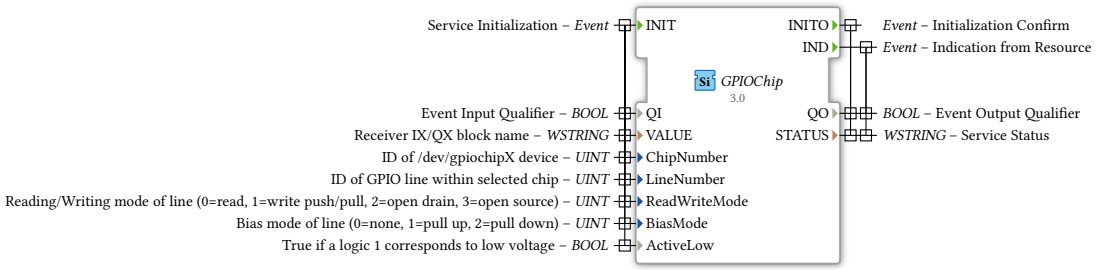
Fazit

Der SET_LOCAL_ADS_ADDRESS Funktionsblock stellt eine flexible Lösung für die dynamische Konfiguration von ADS-Adressen in IEC 61499-basierten Steuerungssystemen dar. Durch seine klare Schnittstellendefinition und umfassende Statusrückmeldung eignet er sich besonders für Anwendungen, die eine flexible Netzwerkkonfiguration erfordern.

2.4.2.15 GPIOChip



GPIOChip



• – ○ • – ○ • – ○ •

Einleitung

Der GPIOChip Funktionsblock ist ein Service Interface Function Block zur Ansteuerung von GPIO-Chips unter Linux-Systemen. Er ermöglicht den Zugriff auf GPIO-Lines über das Linux GPIO Character Device Interface (/dev/gpiochipX) und unterstützt verschiedene Betriebsmodi für das Lesen und Schreiben von GPIO-Signalen.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service Initialisierungs-Ereignis mit den zugehörigen Daten QI, VALUE, ChipNumber, LineNumber, ReadWriteMode, BiasMode und ActiveLow

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung mit QO und STATUS
- **IND**: Indikation vom Resource mit QO und STATUS

Daten-Eingänge

- **QI** (BOOL): Event Input Qualifier
- **VALUE** (WSTRING): Receiver IX/QX block name
- **ChipNumber** (UINT): ID des /dev/gpiochipX Devices (Initialwert: 0)
- **LineNumber** (UINT): ID der GPIO-Line innerhalb des ausgewählten Chips (Initialwert: 0)
- **ReadWriteMode** (UINT): Lese-/Schreibmodus der Line (0=read, 1=write push/pull, 2=open drain, 3=open source) (Initialwert: 0)
- **BiasMode** (UINT): Bias-Modus der Line (0=none, 1=pull up, 2=pull down) (Initialwert: 0)
- **ActiveLow** (BOOL): True, wenn logisch 1 niedriger Spannung entspricht (Initialwert: 0)

Daten-Ausgänge

- **QO** (BOOL): Event Output Qualifier
- **STATUS** (WSTRING): Service Status

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der GPIOChip FB initialisiert bei Empfang des INIT-Ereignisses eine Verbindung zu einem spezifischen GPIO-Chip und einer definierten GPIO-Line. Die Konfiguration umfasst den Betriebsmodus (Lesen/Schreiben), den Bias-Modus und die Polaritätseinstellung (ActiveLow). Nach erfolgreicher Initialisierung bestätigt der Block mit INITO+, bei Fehlern mit INITO- und einem entsprechenden STATUS.

Technische Besonderheiten

- Unterstützt verschiedene GPIO-Betriebsmodi: Lesen, Schreiben (Push/Pull), Open Drain und Open Source
- Konfigurierbare Bias-Einstellungen: None, Pull-Up, Pull-Down
- Aktiv-Niedrig Modus für invertierte Logik
- Linux GPIO Character Device Interface basiert
- Automatische Fehlerbehandlung mit Statusrückmeldung

Zustandsübersicht

Der Funktionsblock durchläuft folgende Zustände:

1. **Nicht initialisiert**: Ausgangszustand vor INIT
2. **Initialisierung**: Während der GPIO-Chip Konfiguration
3. **Bereit**: Nach erfolgreicher Initialisierung (INITO+)
4. **Fehler**: Bei Initialisierungsfehlern (INITO-)

Anwendungsszenarien

- Ansteuerung von GPIO-Pins in Embedded Linux Systemen
- Hardware-nahe E/A-Operationen in Automatisierungssystemen
- Schnittstelle zu externen Sensoren und Aktoren
- Prototyping und Entwicklung von IoT-Geräten
- Industrielle Steuerungsanwendungen

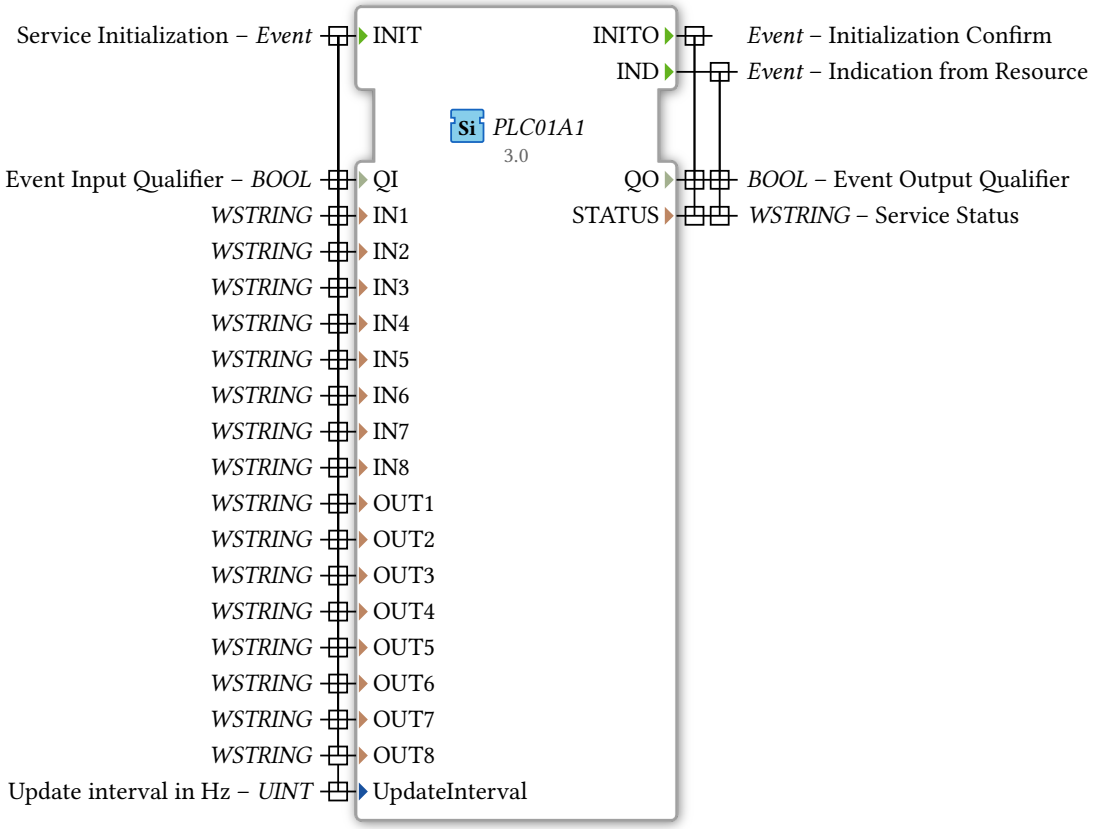
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen I/O-Blöcken bietet GPIOChip erweiterte Funktionen wie konfigurierbare Bias-Modi, verschiedene Schreibmodi und direkten Zugriff auf das Linux GPIO-Subsystem. Er ist spezifisch für Linux-Systeme mit GPIO-Chip Unterstützung ausgelegt.

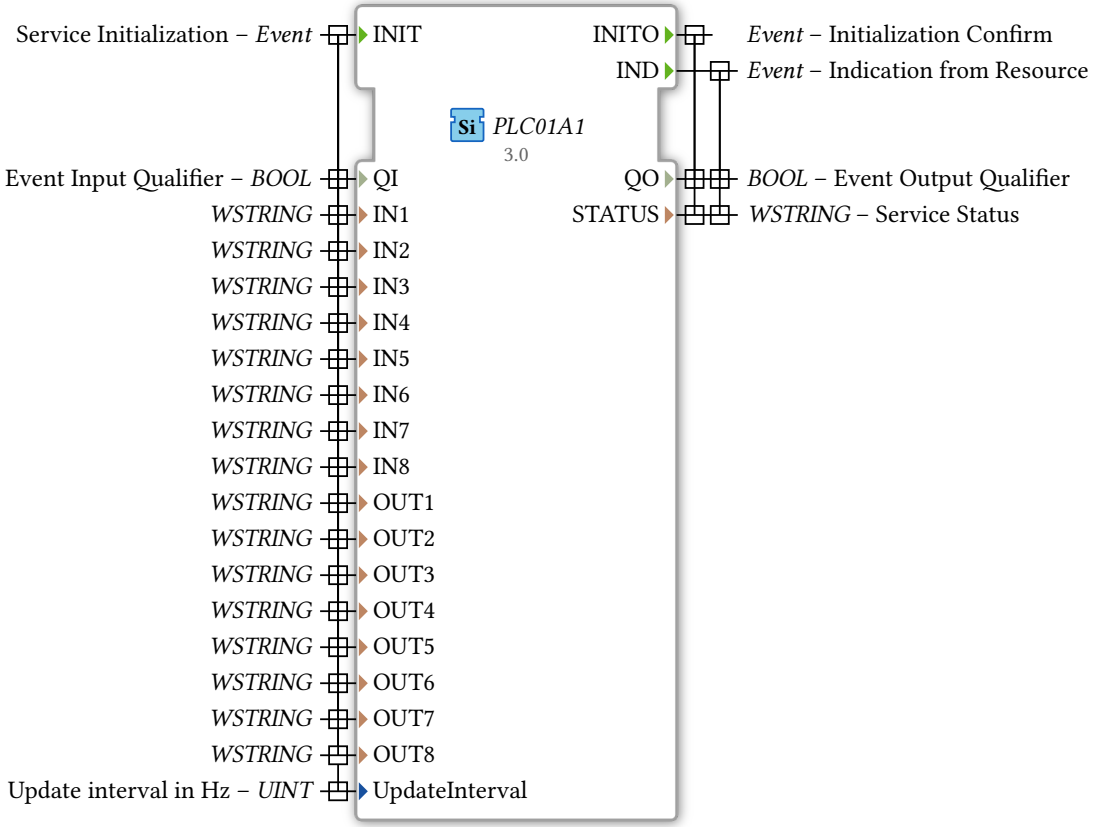
Fazit

Der GPIOChip Funktionsblock stellt eine leistungsfähige Schnittstelle zur GPIO-Hardwaresteuerung unter Linux bereit. Durch die umfangreiche Konfigurierbarkeit und das standardisierte Service Interface eignet er sich ideal für anspruchsvolle Embedded-Anwendungen, die direkten Hardwarezugriff erfordern.

2.4.2.16 PLC01A1



PLC01A1



• – ◦ • – ◦ • – ◦ •

Einleitung

Der PLC01A1 Funktionsblock dient als Schnittstelle für den Zugriff auf die Ein- und Ausgänge des PLC01A1-Moduls. Dieser Baustein ermöglicht die Ansteuerung von 8 digitalen Ausgängen und das Auslesen von 8 digitalen Eingängen über eine standardisierte 4diac-Schnittstelle.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierungsereignis
  - Verknüpft mit: QI, UpdateInterval, IN1-IN8, OUT1-OUT8

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
  - Verknüpft mit: QO, STATUS
- **IND**: Indikation vom Ressource
  - Verknüpft mit: QO, STATUS

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangsqualifizierer
- **IN1-IN8** (WSTRING): 8 digitale Eingänge
- **OUT1-OUT8** (WSTRING): 8 digitale Ausgänge
- **UpdateInterval** (UINT): Aktualisierungsintervall in Hz (Initialwert: 25)

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifizierer
- **STATUS** (WSTRING): Service-Statusinformation

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der PLC01A1 Funktionsblock initialisiert die Hardware-Schnittstelle beim Empfang des INIT-Ereignisses. Nach erfolgreicher Initialisierung wird das INITO-Ereignis ausgegeben. Der Baustein überwacht kontinuierlich die Eingänge IN1-IN8 und aktualisiert die Ausgänge OUT1-OUT8 entsprechend der Konfiguration. Das Aktualisierungsintervall kann über den UpdateInterval-Parameter angepasst werden.

Technische Besonderheiten

- Verwendet WSTRING-Datentyp für alle Ein-/Ausgänge
- Konfigurierbares Aktualisierungsintervall (Standard: 25 Hz)
- Separate Initialisierungs- und Betriebszustände
- Statusrückmeldung über STATUS-Ausgang

Zustandsübersicht

1. **Nicht initialisiert**: Wartet auf INIT-Ereignis
2. **Initialisierung**: Verarbeitet INIT mit Konfigurationsdaten
3. **Betrieb**: Überwacht Eingänge und steuert Ausgänge
4. **Fehler**: Gibt Fehlerstatus über STATUS aus

Anwendungsszenarien

- Anbindung von PLC01A1-Hardware an 4diac-Steuerungen
- Digitale Signalverarbeitung in Automatisierungssystemen
- IO-Management in verteilten Steuerungssystemen
- Prototypenentwicklung mit PLC01A1-Modulen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu generischen IO-Bausteinen bietet PLC01A1 eine spezifische Anpassung für das PLC01A1-Modul mit vordefinierten 8 Ein- und 8 Ausgängen. Die Verwendung von WSTRING ermöglicht eine flexible Datenrepräsentation, unterscheidet sich jedoch von typischen BOOL-basierten IO-Bausteinen.

Fazit

Der PLC01A1 Funktionsblock stellt eine zuverlässige und konfigurierbare Schnittstelle für die PLC01A1-Hardware in 4diac-basierten Steuerungssystemen dar. Durch die klare Trennung von Initialisierungs- und Betriebsphase sowie die konfigurierbare Aktualisierungsrate eignet sich der Baustein sowohl für Echtzeit-Anwendungen als auch für allgemeine Automatisierungsaufgaben.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

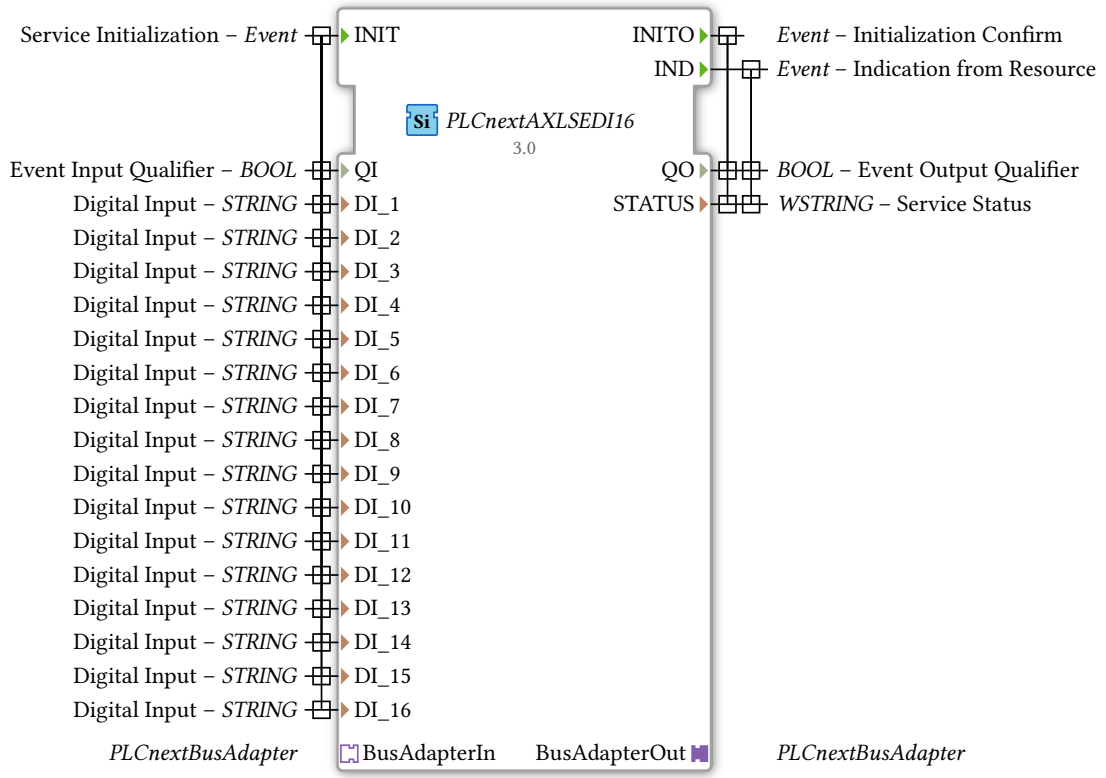
2.4.2.17 PLCnext

PLCnextAXLSEDI16

• – ◦ • – ◦ • – ◦ •

Einleitung

Der PLCnextAXLSEDI16 ist ein Service Interface Function Block für die Anbindung an PLCnext-Systeme. Dieser Funktionsblock dient als Schnittstelle für digitale Eingänge und ermöglicht die Kommunikation mit der PLCnext-Bus-Architektur. Er unterstützt 16 digitale Eingangskanäle und bietet standardisierte Initialisierungs- und Statusrückmeldungen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service Initialization - Initialisiert den Funktionsblock mit den konfigurierten Parametern

Ereignis-Ausgänge

- **INITO**: Initialization Confirm - Bestätigt die erfolgreiche Initialisierung
- **IND**: Indication from Resource - Signalisiert Statusänderungen oder Ereignisse von der Ressource

Daten-Eingänge

- **QI** (BOOL): Event Input Qualifier - Steuert die Initialisierung
- **DI_1 bis DI_16** (STRING): Digital Input - 16 digitale Eingangskanäle für die Konfiguration

Daten-Ausgänge

- **QO** (BOOL): Event Output Qualifier - Status der Ereignisausgabe
- **STATUS** (WSTRING): Service Status - Detaillierte Statusinformationen

Adapter

- **BusAdapterOut** (Plug): Ausgehender Bus-Adapter für PLCnext-Kommunikation
- **BusAdapterIn** (Socket): Eingehender Bus-Adapter für PLCnext-Kommunikation

Funktionsweise

Der Funktionsblock initialisiert sich über das INIT-Ereignis und konfiguriert die 16 digitalen Eingangskanäle basierend auf den DI_Parameters. Nach erfolgreicher Initialisierung bestätigt er dies über INITO. Während des Betriebs überwacht er kontinuierlich die Eingänge und signalisiert Änderungen über das IND-Ereignis. Die Kommunikation mit dem PLCnext-Bus erfolgt über die integrierten Adapter-Schnittstellen.

Technische Besonderheiten

- Unterstützt 16 unabhängige digitale Eingangskanäle
- String-basierte Konfiguration der Eingänge
- Unicode-fähige Statusrückmeldungen (WSTRING)

- Integrierte PLCnext-Bus-Adapter-Kommunikation
- Standardkonforme IEC 61499-2-Implementierung

Zustandsübersicht

1. **Nicht initialisiert**: Funktionsblock wartet auf INIT-Ereignis
2. **Initialisierung**: Verarbeitung der Konfigurationsparameter
3. **Betriebsbereit**: Überwachung der digitalen Eingänge aktiv
4. **Fehlerzustand**: Bei Problemen mit Statusrückmeldung

Anwendungsszenarien

- Anbindung digitaler Sensoren an PLCnext-Systeme
- Industrielle Automatisierungssteuerungen
- Überwachungssysteme mit multiplen Eingangssignalen
- PLCnext-basierte Steuerungsarchitekturen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Eingangsbausteinen bietet PLCnextAXLSEDI16 eine erweiterte Konfigurierbarkeit durch String-Parameter und eine integrierte Bus-Kommunikation für PLCnext-Systeme. Die 16 Kanäle ermöglichen eine höhere Eingangsdichte als Standardbausteine.

Fazit

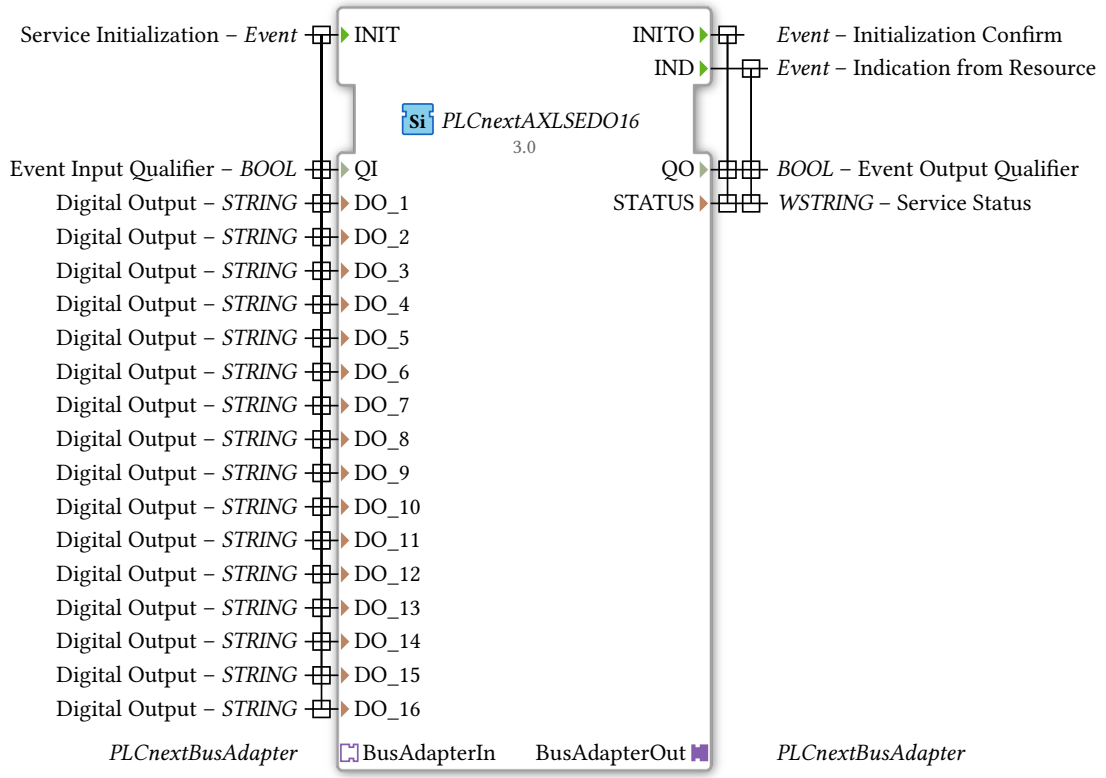
Der PLCnextAXLSEDI16 ist ein leistungsstarker Funktionsblock für die Integration digitaler Eingänge in PLCnext-basierte Automatisierungssysteme. Seine flexible Konfiguration und robuste Bus-Kommunikation machen ihn ideal für industrielle Anwendungen mit multiplen digitalen Signalen.

PLCnextAXLSEDO16

• – ◦ • – ◦ • – ◦ •

Einleitung

Der PLCnextAXLSEDO16 ist ein Service Interface Function Block für die Ansteuerung von 16 digitalen Ausgängen in PLCnext-Systemen. Der Baustein dient als Schnittstelle zwischen der IEC 61499-basierten Steuerungslogik und den physikalischen Ausgängen des PLCnext-AXLSEDO16-Moduls.



Schnittstellenstruktur

Ereignis-Eingänge
- **INIT**: Service Initialization - Initialisiert den Baustein und konfiguriert die digitalen Ausgänge

Ereignis-Ausgänge
- **INITO**: Initialization Confirm - Bestätigt die erfolgreiche Initialisierung
- **IND**: Indication from Resource - Signalisiert Statusänderungen oder Fehler vom Ressourcen-Adapter

Daten-Eingänge
- **QI** (BOOL): Event Input Qualifier - Steuert die Initialisierung
- **DO_1** bis **DO_16** (STRING): Digitale Ausgänge - Konfigurieren die jeweiligen Ausgangskanäle

Daten-Ausgänge
- **QO** (BOOL): Event Output Qualifier - Status der Ereignisausgabe
- **STATUS** (WSTRING): Service Status - Detaillierte Statusinformationen

Adapter
- **BusAdapterOut** (Plug): Ausgehender Bus-Adapter für die Kommunikation mit der Hardware
- **BusAdapterIn** (Socket): Eingehender Bus-Adapter für Rückmeldungen von der Hardware

Funktionsweise
Der Baustein initialisiert bei Empfang des INIT-Ereignisses die 16 digitalen Ausgänge basierend auf den Konfigurationswerten in DO_1 bis DO_16. Die Initialisierung wird durch INITO bestätigt. Während des Betriebs können Statusänderungen und Fehler über das IND-Ereignis gemeldet werden. Die Kommunikation mit der physikalischen Hardware erfolgt über die Bus-Adapter-Schnittstellen.

Technische Besonderheiten
- Unterstützt 16 unabhängige digitale Ausgänge
- Verwendet STRING-Typ für die Ausgangskonfiguration, was flexible Parametrierung ermöglicht
- Integriert sich nahtlos in PLCnext-Systeme über spezifische Bus-Adapter
- Bietet umfassende Statusrückmeldung über WSTRING-Variable

Zustandsübersicht
1. **Nicht initialisiert**: Baustein wartet auf INIT-Ereignis
2. **Initialisierung läuft**: Verarbeitung der Konfigurationsparameter
3. **Betriebsbereit**: Ausgänge sind konfiguriert und bereit für den Betrieb
4. **Fehlerzustand**: Bei Problemen wird STATUS mit Fehlerinformationen gefüllt

Anwendungsszenarien
- Ansteuerung von Relais und Aktoren in Automatisierungsanlagen
- Steuerung von Beleuchtungssystemen
- Kontrolle von Ventilen und Motoren
- Signalausgabe in Prozessleitsystemen

⚖️ Vergleich mit ähnlichen Bausteinen
Im Vergleich zu einfacheren digitalen Ausgangsbausteinen bietet PLCnextAXLSEDO16:

- Höhere Kanalanzahl (16 statt typischerweise 8)
- Spezifische Integration für PLCnext-Hardware
- Erweiterte Statusrückmeldungen
- Flexible String-basierte Konfiguration

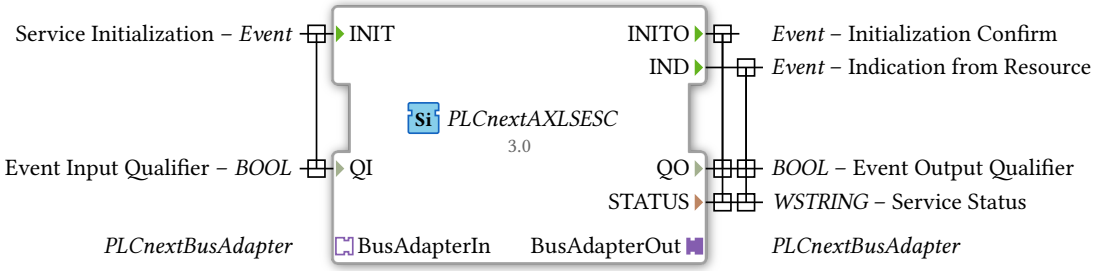
Fazit
Der PLCnextAXLSEDO16 ist ein leistungsstarker Funktionsbaustein für die Ansteuerung digitaler Ausgänge in PLCnext-Umgebungen. Seine 16 Kanäle, die flexible Konfigurierbarkeit und die umfassende Statusrückmeldung machen ihn ideal für komplexe Automatisierungsaufgaben, bei denen mehrere Aktoren gleichzeitig gesteuert werden müssen.

PLCnextAXLSESC

• – ◦ • – ◦ • – ◦ •

Einleitung

Der PLCnextAXLSESC ist ein Service Interface Function Block, der als Schnittstelle für PLCnext-Bussysteme dient. Dieser Baustein ermöglicht die Kommunikation zwischen der 4diac-Umgebung und PLCnext-Hardwarekomponenten über spezielle Busadapter.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service Initialisierung - Initialisiert den Service mit dem Qualifier QI

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung - Bestätigt die erfolgreiche Initialisierung mit QO und STATUS
- **IND**: Indikation von der Ressource - Signalisiert Statusänderungen mit QO und STATUS

Daten-Eingänge

- **QI** (BOOL): Event Input Qualifier - Steuert die Initialisierung des Services

Daten-Ausgänge

- **QO** (BOOL): Event Output Qualifier - Zeigt den Status der Service-Ausgabe an
- **STATUS** (WSTRING): Service Status - Enthält detaillierte Statusinformationen als Unicode-String

Adapter

- **BusAdapterOut** (Plug): Ausgehender Busadapter für PLCnext-Kommunikation
- **BusAdapterIn** (Socket): Eingehender Busadapter für PLCnext-Kommunikation

Funktionsweise

Der PLCnextAXLSESC-Baustein dient als Vermittler zwischen der 4diac-Steuerung und PLCnext-Hardware. Bei Erhalt eines INIT-Ereignisses mit aktiviertem QI initialisiert der Baustein die Kommunikation über die Busadapter. Nach erfolgreicher Initialisierung wird dies durch das INITO-Ereignis bestätigt. Während des Betriebs können über das IND-Ereignis Statusänderungen und Indikationen von der Hardware gemeldet werden.

Technische Besonderheiten

- Verwendet WSTRING für STATUS-Ausgaben, was internationale Zeichensätze unterstützt
- Implementiert ein bidirektionales Adapter-System für flexible Buskommunikation
- Separate Ereignisse für Initialisierung und laufende Indikationen
- Qualifier-Bits (QI/QO) für zuverlässige Ereignissteuerung

Zustandsübersicht

1. **Nicht initialisiert**: Wartet auf INIT-Ereignis mit QI=TRUE
2. **Initialisierungsphase**: Verarbeitet Busadapter-Kommunikation
3. **Betriebsbereit**: Sendet INITO-Bestätigung und ist bereit für IND-Ereignisse
4. **Aktiver Betrieb**: Verarbeitet eingehende Indikationen und gibt STATUS aus

Anwendungsszenarien

- Integration von PLCnext-Hardware in 4diac-basierte Steuerungssysteme

- Kommunikation mit PLCnext-I/O-Modulen und Peripheriegeräten
- Überwachung und Steuerung von industriellen Automatisierungsprozessen
- Bridge-Funktionalität zwischen 4diac und PLCnext-Ökosystem

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu generischen IO-Bausteinen bietet PLCnextAXLSESC spezifische Anpassungen für die PLCnext-Plattform, insbesondere durch die integrierten Busadapter. Während Standard-IO-Bausteine allgemeine Schnittstellen bereitstellen, ist dieser Baustein speziell auf die Anforderungen der PLCnext-Architektur optimiert.

Fazit

Der PLCnextAXLSESC-Funktionsblock stellt eine zuverlässige und spezialisierte Schnittstelle für die Integration von PLCnext-Hardware in 4diac-Steuerungssysteme dar. Durch seine klare Ereignisstruktur und flexible Adapter-Architektur ermöglicht er eine robuste Kommunikation in industriellen Automatisierungsanwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

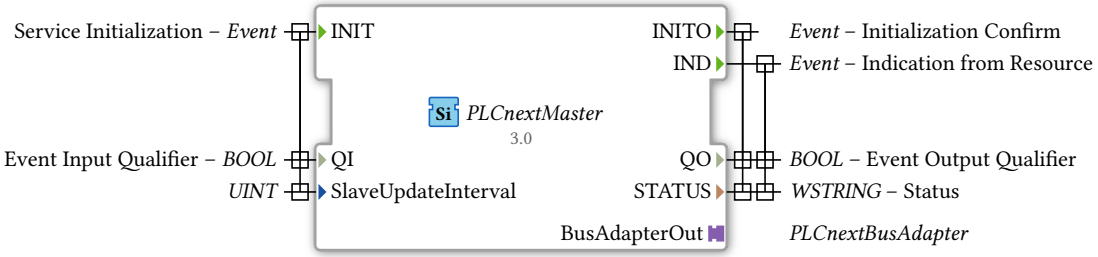
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

PLCnextMaster

• – ◦ • – ◦ • – ◦ •

Einleitung

Der PLCnextMaster ist ein Service Interface Function Block, der als Master-Komponente für die Kommunikation mit PLCnext-Systemen dient. Dieser Baustein ermöglicht die Initialisierung und Steuerung der Bus-Kommunikation in PLCnext-basierten Automatisierungslösungen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service Initialisierung
  - Verknüpft mit: QI, SlaveUpdateInterval

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
  - Verknüpft mit: QO, STATUS
- **IND**: Indikation von der Resource
  - Verknüpft mit: QO, STATUS

Daten-Eingänge

- **QI** (BOOL): Event Input Qualifier - Steuert die Aktivierung des Service
- **SlaveUpdateInterval** (UINT): Aktualisierungsintervall für Slaves mit Initialwert 25

Daten-Ausgänge

- **QO** (BOOL): Event Output Qualifier - Bestätigt den Status der Operation
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

- **BusAdapterOut**: PLCnextBusAdapter für die Bus-Kommunikation

- Typ: eclipse4diac::io::plcnext::PLCnextBusAdapter

Funktionsweise

Der PLCnextMaster initialisiert und verwaltet die Kommunikation mit PLCnext-Systemen. Bei Aktivierung des INIT-Events mit QI=TRUE wird der Service gestartet. Das SlaveUpdateInterval definiert das Aktualisierungsintervall für angeschlossene Slave-Komponenten. Nach erfolgreicher Initialisierung bestätigt INITO den Betriebszustand, während IND-Events asynchrone Statusänderungen anzeigen.

Technische Besonderheiten

- Unterstützt Unicode-Statusmeldungen (WSTRING)
- Konfigurierbares Slave-Update-Intervall
- Adapter-basierte Bus-Kommunikation
- Initialwert für SlaveUpdateInterval auf 25 gesetzt

Zustandsübersicht

1. **Inaktiv**: Vor INIT-Aktivierung
2. **Initialisierung**: Während INIT-Verarbeitung
3. **Aktiv**: Nach erfolgreicher INITO-Bestätigung
4. **Bereit**: Für IND-Events bei aktiver Kommunikation

Anwendungsszenarien

- PLCnext-basierte Automatisierungssysteme
- Industrie 4.0 Anwendungen
- Vernetzte Steuerungssysteme
- IoT-Edge-Computing-Lösungen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu generischen Bus-Master-Bausteinen bietet der PLCnextMaster spezifische Optimierungen für PLCnext-Hardware und unterstützt die spezifische Bus-Architektur von PLCnext-Systemen.

Fazit

Der PLCnextMaster stellt einen spezialisierten Funktionsbaustein für die Integration von PLCnext-Systemen in 4diac-basierte Automatisierungslösungen dar. Durch seine konfigurierbaren Parameter und Adapter-Schnittstelle ermöglicht er eine flexible und zuverlässige Kommunikation in industriellen Steuerungsumgebungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

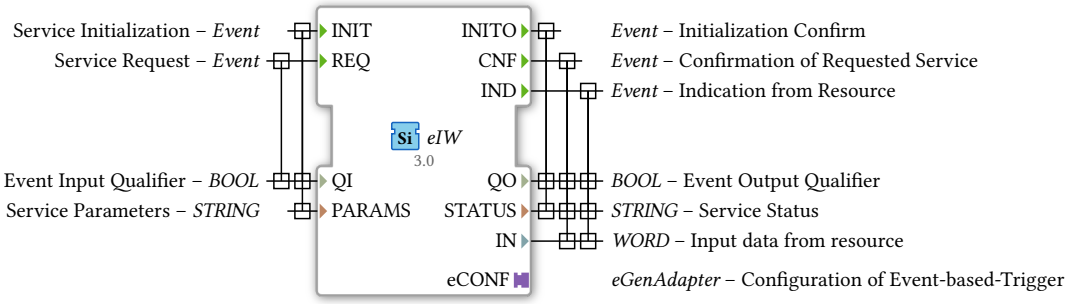
2.4.2.18 eIO

eIW

• – ◦ • – ◦ • – ◦ •

Einleitung

Der eIW-Funktionsblock ist ein Eingabeservice-Interface für Wort-Eingabedaten (WORD) mit ereignisbasierter Auslösung. Er dient als Schnittstelle zwischen der Steuerungsanwendung und physikalischen Eingabegeräten und ermöglicht die Abfrage von 16-Bit-Datenwörtern.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung - initialisiert den Funktionsblock und konfiguriert die Hardware-Parameter
- **REQ**: Service-Anfrage - löst eine Abfrage der Eingabedaten aus

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung - bestätigt die erfolgreiche Initialisierung
- **CNF**: Bestätigung der angeforderten Service-Anfrage - liefert die angeforderten Eingabedaten
- **IND**: Indikation von der Ressource - signalisiert asynchrone Zustandsänderungen der Eingabedaten

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangsqualifikator - steuert die Ausführung der Service-Anfragen
- **PARAMS** (STRING): Service-Parameter - Konfigurationsparameter für die Hardware-Schnittstelle

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifikator - zeigt den Erfolg oder Fehler der Operation an
- **STATUS** (STRING): Service-Status - detaillierte Statusinformationen über die ausgeführte Operation
- **IN** (WORD): Eingabedaten von der Ressource - die gelesenen 16-Bit-Daten

Adapter

- **eCONF** (eclipse4diac::io::eio::eGenAdapter): Konfiguration der ereignisbasierten Auslösung - ermöglicht die Anpassung des Trigger-Verhaltens

Funktionsweise

Der eIW-Funktionsblock arbeitet als Vermittler zwischen der Steuerungsanwendung und physikalischen Eingabegeräten. Bei der Initialisierung (INIT) werden die Hardware-Parameter konfiguriert. Anschließend können über REQ-Ereignisse gezielt Eingabedaten abgefragt werden. Der Block unterstützt sowohl synchrone Abfragen (CNF) als auch asynchrone Benachrichtigungen bei Zustandsänderungen (IND).

Technische Besonderheiten

- Verarbeitet 16-Bit-Wort-Daten (WORD-Datentyp)
- Unterstützt ereignisbasierte und polling-basierte Abfragemodi
- Bietet detaillierte Statusrückmeldungen über den STATUS-Ausgang
- Ermöglicht flexible Hardware-Konfiguration über PARAMS-Eingang
- Integriert Adapter-Schnittstelle für erweiterte Trigger-Konfiguration

Zustandsübersicht

Der Funktionsblock durchläuft folgende Zustände:

1. **Nicht initialisiert**: Block ist betriebsbereit, aber nicht konfiguriert
2. **Initialisiert**: Hardware-Parameter sind konfiguriert, Block ist betriebsbereit
3. **Abfrage aktiv**: Service-Anfrage wird verarbeitet
4. **Daten verfügbar**: Eingabedaten stehen zur Verfügung und werden ausgegeben

Anwendungsszenarien

- Abfrage von digitalen Eingangsgruppen (16-Bit)

- Lesen von Encoder-Werten
- Überwachung von Statusregistern
- Erfassung von Sensor-Datenwörtern
- Schnittstelle zu I/O-Modulen mit Wort-Zugriff

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Eingabeblöcken bietet eIW:

- Erweiterte 16-Bit-Datenverarbeitung
- Detaillierte Statusrückmeldungen
- Flexible Konfigurationsmöglichkeiten
- Unterstützung für ereignisbasierte und polling-basierte Betriebsarten
- Adapter-Schnittstelle für erweiterte Funktionalität

Fazit

Der eIW-Funktionsblock stellt eine leistungsstarke und flexible Schnittstelle für die Verarbeitung von Wort-Eingabedaten in IEC 61499-Steuerungssystemen dar. Seine kombinierte Unterstützung für synchrone Abfragen und asynchrone Benachrichtigungen macht ihn besonders geeignet für Anwendungen, die sowohl deterministische Abfragen als auch zeitkritische Zustandsänderungen erfordern.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

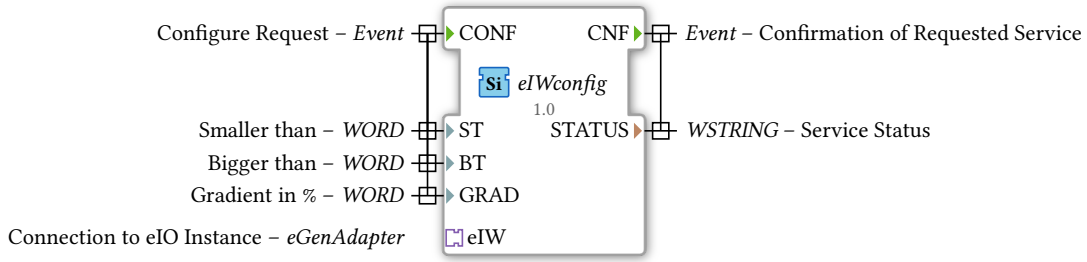
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

eIWconfig

• – ◦ • – ◦ • – ◦ •

Einleitung

Der eIWconfig Funktionsblock dient als Service Interface für die Konfiguration von eIO-Instanzen. Er ermöglicht die Einstellung von Schwellwerten und Gradienten für industrielle IO-Operationen.



Schnittstellenstruktur

Ereignis-Eingänge

- **CONF**: Konfigurationsanforderung - löst die Konfiguration der eIO-Instanz aus

Ereignis-Ausgänge

- **CNF**: Bestätigung des angeforderten Services - signalisiert den Abschluss der Konfiguration

Daten-Eingänge

- **ST** (WORD): „Smaller than“ - Unterer Schwellwert
- **BT** (WORD): „Bigger than“ - Oberer Schwellwert
- **GRAD** (WORD): Gradient in Prozent - Steilheit der Übergänge

Daten-Ausgänge

- **STATUS** (WSTRING): Service Status - Rückmeldung über den Konfigurationsvorgang

Adapter

- **eIW**: Verbindung zur eIO-Instanz (Typ: eclipse4diac::io::eio::eGenAdapter)

Funktionsweise

Der Funktionsblock empfängt über das CONF-Ereignis eine Konfigurationsanforderung zusammen mit den Parametern ST, BT und GRAD. Diese Parameter werden über den eIW-Adapter an die eIO-Instanz übertragen. Nach erfolgreicher Konfiguration bestätigt der Block den Vorgang durch das CNF-Ereignis und liefert über STATUS eine Statusrückmeldung.

Technische Besonderheiten

- Verwendet WORD-Datentypen für alle numerischen Eingänge
- STATUS-Ausgang als WSTRING für detaillierte Statusinformationen
- Adapter-basierte Architektur für lose Kopplung zur eIO-Instanz

Zustandsübersicht

1. **Bereit**: Wartet auf CONF-Ereignis
2. **Konfigurierend**: Verarbeitet Konfigurationsparameter
3. **Bestätigend**: Sendet CNF-Ereignis mit Statusinformation

Anwendungsszenarien

- Konfiguration von IO-Schwellwerten in Automatisierungssystemen
- Einstellung von Filterparametern für Sensorsignale
- Dynamische Anpassung von Grenzwerten in Regelungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen IO-Konfigurationsblöcken bietet eIWconfig eine erweiterte Funktionalität mit Gradientensteuerung und Adapter-basierter Architektur für höhere Flexibilität.

Fazit

Der eIWconfig Funktionsblock stellt eine flexible und erweiterbare Lösung für die Konfiguration von eIO-Instanzen dar, die durch ihre Adapter-basierte Architektur und umfangreiche Parametrierungsmöglichkeiten besticht.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

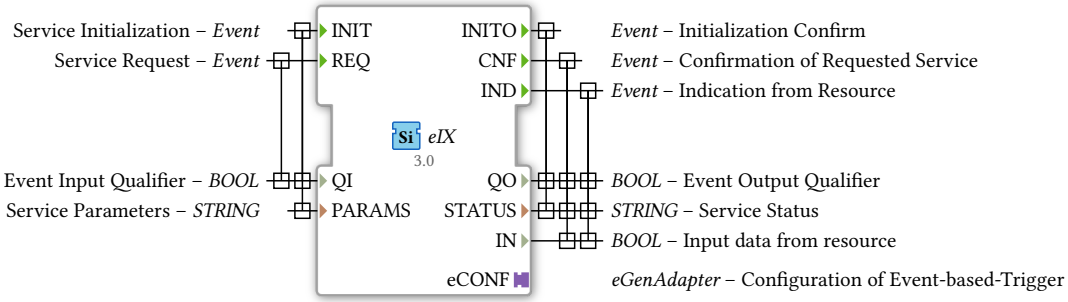
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

eIX

• – ◦ • – ◦ • – ◦ •

Einleitung

Der eIX-Funktionsblock ist ein Eingabeservice-Interface für boolesche Eingabedaten mit ereignisbasierter Auslösung. Er dient als Schnittstelle zwischen der Steuerungslogik und physikalischen Eingabegeräten und ermöglicht die Abfrage von digitalen Eingangssignalen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung - Initialisiert den Funktionsblock mit Parametern
- **REQ**: Service-Anfrage - Löst eine Abfrage des Eingangssignals aus

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung - Bestätigt die erfolgreiche Initialisierung
- **CNF**: Service-Bestätigung - Bestätigt die verarbeitete Service-Anfrage
- **IND**: Indikation von der Ressource - Signalisiert eine Statusänderung des Eingangs

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangsqualifizierer - Aktiviert/deaktiviert die Verarbeitung
- **PARAMS** (STRING): Service-Parameter - Konfigurationsparameter für den Service

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifizierer - Status der Verarbeitung
- **STATUS** (STRING): Service-Status - Fehler- oder Statusinformationen
- **IN** (BOOL): Eingabedaten von der Ressource - Aktueller Wert des digitalen Eingangs

Adapter

- **eCONF**: Konfiguration der ereignisbasierten Auslösung - Adapter für die Trigger-Konfiguration

Funktionsweise

Der eIX-Block arbeitet als Service-Interface für digitale Eingänge. Bei Initialisierung (INIT) werden die Service-Parameter konfiguriert. Über REQ-Ereignisse können gezielte Abfragen des Eingangssignals ausgelöst werden. Der Block liefert über CNF die aktuellen Eingangswerte zurück und kann über IND asynchrone Statusänderungen melden.

Technische Besonderheiten

- Ereignisbasierte Abfragemechanismen
- Separater Initialisierungs- und Betriebsmodus
- String-basierte Statusrückmeldungen
- Adapter-basierte Konfiguration der Trigger-Logik
- Unterstützung sowohl für angefragte als auch indizierte Operationen

Zustandsübersicht

1. **Nicht initialisiert**: Block wartet auf INIT-Ereignis
2. **Initialisiert**: Block ist betriebsbereit und kann REQ-Ereignisse verarbeiten
3. **Aktiv**: Verarbeitung von Service-Anfragen und Rückmeldung von Eingangswerten
4. **Fehler**: Bei Problemen wird STATUS mit Fehlerinformationen gefüllt

Anwendungsszenarien

- Abfrage von digitalen Sensoren (Endschalter, Näherungsschalter)
- Überwachung von Schaltzuständen
- Integration von Tastern und Bedienelementen
- Signalverarbeitung in Automatisierungssystemen
- Zustandsüberwachung von Maschinen und Anlagen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Eingangsblöcken bietet eIX erweiterte Funktionalität:

- Ereignisbasierte statt zyklische Abfrage
- Detaillierte Statusrückmeldungen
- Konfigurierbare Trigger-Mechanismen
- Separate Initialisierungs- und Betriebsphasen

Fazit

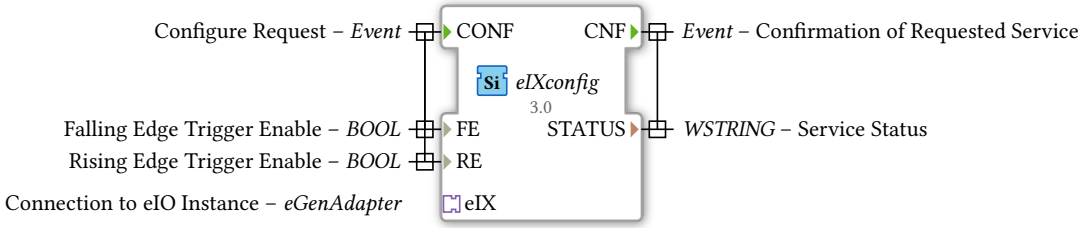
Der eIX-Funktionsblock stellt eine flexible und robuste Lösung für die Integration digitaler Eingänge in IEC 61499-basierten Steuerungssystemen dar. Durch seine ereignisbasierte Architektur und umfangreiche Statusrückmeldungen eignet er sich besonders für Anwendungen, die zuverlässige und konfigurierbare Eingabeverarbeitung erfordern.

eIXconfig

• – ◦ • – ◦ • – ◦ •

Einleitung

Der eIXconfig Funktionsblock dient als Service Interface Function Block zur Konfiguration von eIO-Instanzen. Er ermöglicht die Einstellung von Trigger-Eigenschaften für Ein- und Ausgänge und stellt eine Verbindung zu eIO-Instanzen über einen Adapter bereit.



Schnittstellenstruktur

Ereignis-Eingänge

- **CONF**: Konfigurationsanforderung - löst die Konfiguration der eIO-Instanz aus

Ereignis-Ausgänge

- **CNF**: Bestätigung des angeforderten Services - signalisiert den Abschluss der Konfiguration

Daten-Eingänge

- **FE** (BOOL): Falling Edge Trigger Enable - aktiviert die Flankenauslösung bei fallender Flanke
- **RE** (BOOL): Rising Edge Trigger Enable - aktiviert die Flankenauslösung bei steigender Flanke

Daten-Ausgänge

- **STATUS** (WSTRING): Service Status - liefert Statusinformationen über den Konfigurationsvorgang

Adapter

- **eIX**: Verbindung zur eIO-Instanz - stellt die Kommunikation mit der eIO-Hardware her

Funktionsweise

Der eIXconfig Block empfängt über das CONF-Ereignis eine Konfigurationsanforderung. Zusammen mit den Daten-Eingängen FE und RE werden die Trigger-Einstellungen für die eIO-Instanz konfiguriert. Nach erfolgreicher Konfiguration wird das CNF-Ereignis mit dem entsprechenden STATUS ausgegeben. Die tatsächliche Kommunikation mit der Hardware erfolgt über den eIX-Adapter.

Technische Besonderheiten

- Verwendet WSTRING für Statusausgaben, was umfangreiche Statusinformationen ermöglicht
- Unterstützt separate Konfiguration von steigender und fallender Flankenerkennung
- Implementiert als Service Interface Function Block gemäß IEC 61499-2 Standard

Zustandsübergänge

1. **Initialisierungszustand**: Block wartet auf CONF-Ereignis
2. **Konfigurationszustand**: Verarbeitung der Trigger-Einstellungen und Übertragung an eIO-Instanz
3. **Bestätigungszustand**: Ausgabe von CNF mit Statusinformation

Anwendungsszenarien

- Konfiguration von eIO-Hardwarekomponenten in industriellen Steuerungssystemen
- Einstellung von Trigger-Eigenschaften für digitale Eingänge
- Integration in größere Automatisierungssysteme mit eIO-Komponenten

⚖️ Vergleich mit ähnlichen Bausteinen

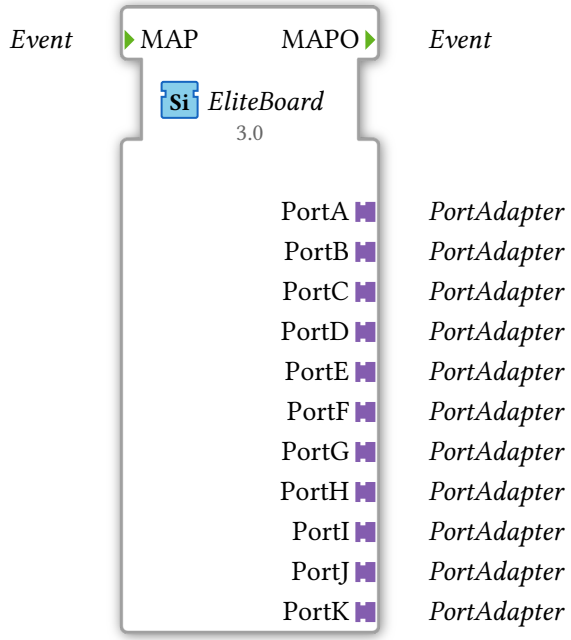
Im Vergleich zu einfachen IO-Konfigurationsblöcken bietet eIXconfig spezifische eIO-Integration und erweiterte Trigger-Konfigurationsmöglichkeiten. Die Verwendung eines Adapters ermöglicht eine saubere Trennung zwischen Funktionalität und Hardwarekommunikation.

Fazit

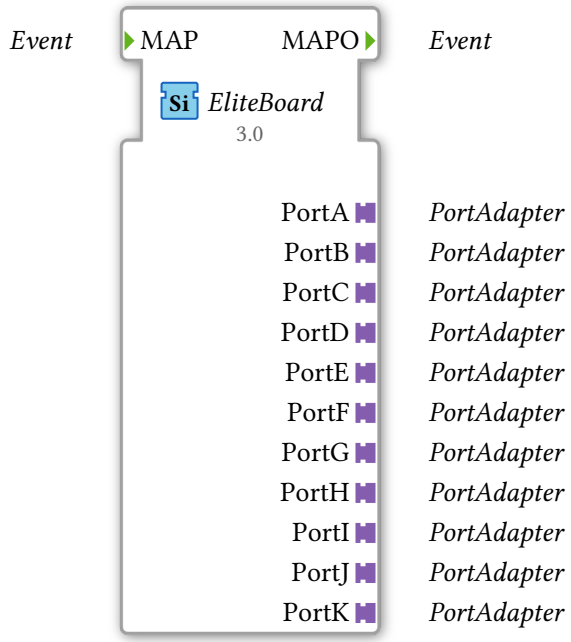
Der eIXconfig Funktionsblock stellt eine spezialisierte Lösung für die Konfiguration von eIO-Instanzen dar. Durch die klare Trennung von Konfigurationslogik und Hardwarekommunikation sowie die flexible Trigger-Einstellung eignet er sich ideal für anspruchsvolle industrielle Anwendungen.

2.4.2.19 Eliteboard

eliteboard



EliteBoard



• – ◦ • – ◦ • – ◦ •

Einleitung

Der EliteBoard-Funktionsblock ist ein Service Interface Function Block Type, der als Schnittstelle für die EliteBoard-Hardware dient. Dieser Baustein ermöglicht die Anbindung und Steuerung von bis zu 11 verschiedenen Ports über standardisierte Adapter-Schnittstellen.

Schnittstellenstruktur

Ereignis-Eingänge

- **MAP**: Initialisiert den Mapping-Prozess für die konfigurierten Ports

Ereignis-Ausgänge

- **MAPO**: Signalisiert den erfolgreichen Abschluss des Mapping-Prozesses

Daten-Eingänge

Keine Daten-Eingänge vorhanden

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

Der Funktionsblock verfügt über 11 Adapter-Schnittstellen (Plugs):

- **PortA** bis **PortK**: Jeweils vom Typ `eclipse4diac::io::eliteboard::PortAdapter`

Diese Adapter ermöglichen die Verbindung mit verschiedenen Port-Schnittstellen der EliteBoard-Hardware.

Funktionsweise

Beim Empfang des MAP-Ereignisses initialisiert der EliteBoard-Funktionsblock die konfigurierten Port-Adapter. Nach erfolgreicher Initialisierung aller angeschlossenen Ports wird das MAPO-Ereignis ausgelöst. Jeder Port-Adapter (PortA bis PortK) kann individuell für spezifische Ein-/Ausgabe-Funktionen konfiguriert werden.

Technische Besonderheiten

- Unterstützt bis zu 11 unabhängige Port-Schnittstellen
- Verwendet standardisierte PortAdapter-Schnittstellen
- Einfache Initialisierung über ein einzelnes Ereignis
- Robuste Fehlerbehandlung durch zentralisiertes Mapping

Zustandsübersicht

1. **Initialzustand**: Wartet auf MAP-Ereignis
2. **Mapping-Zustand**: Initialisiert alle konfigurierten Port-Adapter
3. **Bereit-Zustand**: Alle Ports sind initialisiert, MAPO-Ereignis wird ausgelöst

Anwendungsszenarien

- Steuerung von EliteBoard-Hardware in industriellen Automatisierungssystemen
- Anbindung multipler Ein-/Ausgabe-Schnittstellen in Embedded-Systemen
- Integration in verteilte Steuerungssysteme nach IEC 61499
- Hardware-nahe Steuerungsaufgaben mit modularer Port-Verwaltung

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen IO-Bausteinen bietet EliteBoard eine strukturierte Verwaltung multipler Ports über standardisierte Adapter-Schnittstellen. Während einfache IO-Bausteine typischerweise direkte Daten-Ein-/Ausgänge verwenden, setzt EliteBoard auf Adapter-basierte Architektur für bessere Modularität und Wiederverwendbarkeit.

Fazit

Der EliteBoard-Funktionsblock stellt eine flexible und skalierbare Lösung für die Anbindung von EliteBoard-Hardware in 4diac-basierten Steuerungssystemen dar. Durch die Verwendung standardisierter

Adapter-Schnittstellen und die Unterstützung multipler Ports ermöglicht er eine modulare und wartbare Systemarchitektur für industrielle Automatisierungsanwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

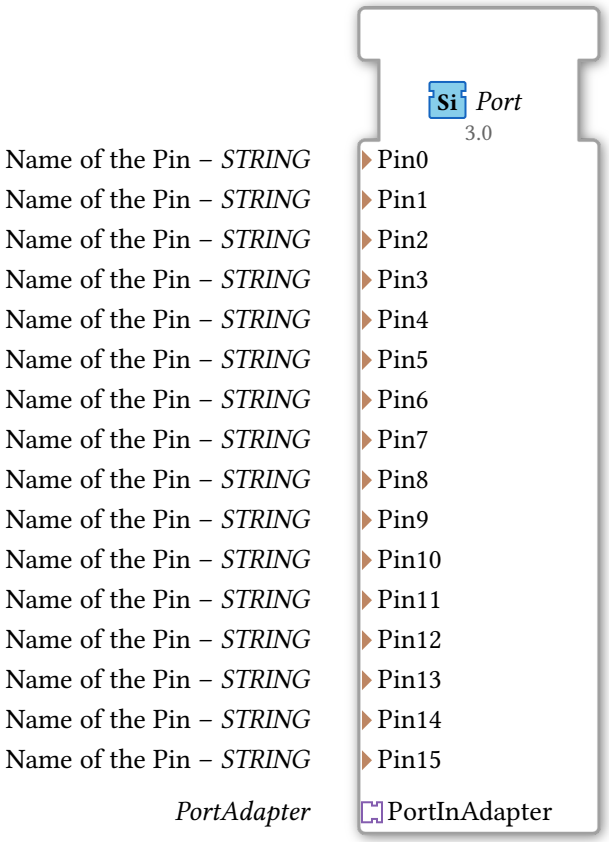
Port

🎧 Podcast

- Das Technologie-Panorama von 1863: Lanz & Comp. und die Revolution der deutschen Landwirtschaft durch Import, Innovation und Guano
- ESP32-S3-DevKitC-1 Doku-Analyse: Das Speicher-Monster (32MB Flash/16MB PSRAM) und die Macht der Dual-USB-Ports

Einleitung

Der Port-Funktionsblock dient als Service Interface Function Block für die Konfiguration und Verwaltung von digitalen Ein-/Ausgängen auf einem EliteBoard-System. Er ermöglicht die Zuordnung von Pin-Namen zu physischen Ports über einen Adapter-Mechanismus.



Schnittstellenstruktur

Ereignis-Eingänge

Keine Ereignis-Eingänge vorhanden

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden

Daten-Eingänge

- **Pin0** (STRING): Name des Pins 0
- **Pin1** (STRING): Name des Pins 1

- **Pin2** (STRING): Name des Pins 2
- **Pin3** (STRING): Name des Pins 3
- **Pin4** (STRING): Name des Pins 4
- **Pin5** (STRING): Name des Pins 5
- **Pin6** (STRING): Name des Pins 6
- **Pin7** (STRING): Name des Pins 7
- **Pin8** (STRING): Name des Pins 8
- **Pin9** (STRING): Name des Pins 9
- **Pin10** (STRING): Name des Pins 10
- **Pin11** (STRING): Name des Pins 11
- **Pin12** (STRING): Name des Pins 12
- **Pin13** (STRING): Name des Pins 13
- **Pin14** (STRING): Name des Pins 14
- **Pin15** (STRING): Name des Pins 15

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden

Adapter

- **PortInAdapter** (eclipse4diac::io::eliteboard::PortAdapter): Socket-Adapter für die Kommunikation mit dem Port-Subsystem

Funktionsweise

Der Port-FB dient als Konfigurationsschnittstelle für bis zu 16 digitale Pins. Über die STRING-Eingänge können individuelle Pin-Namen vergeben werden, die über den PortAdapter an das darunterliegende System weitergegeben werden. Der Funktionsblock selbst führt keine direkten Ein-/Ausgabeoperationen durch, sondern stellt lediglich die Konfigurationsparameter bereit.

Technische Besonderheiten

- Unterstützt Konfiguration von 16 digitalen Pins
- Verwendet STRING-Datentyp für Pin-Namen
- Implementiert als Service Interface Function Block
- Nutzt Adapter-Pattern für Hardware-Abstraktion
- Teil des eclipse4diac::io::eliteboard-Pakets

Zustandsübersicht

Da es sich um einen reinen Konfigurationsbaustein ohne Ereignissteuerung handelt, besitzt der Port-FB keine expliziten Zustände. Die Konfiguration wird statisch über die Daten-Eingänge festgelegt.

Anwendungsszenarien

- Konfiguration von digitalen Ein-/Ausgängen auf EliteBoard-Systemen
- Abstraktion von Hardware-Pins durch logische Namen
- Integration in größere Steuerungssysteme mit einheitlicher Pin-Verwaltung
- Wiederverwendbare Pin-Konfigurationen in verschiedenen Applikationen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu direkten GPIO-Bausteinen bietet der Port-FB eine höhere Abstraktionsebene durch die Verwendung von logischen Pin-Namen statt direkter Hardware-Adressierung. Er dient primär der Konfiguration, während andere Bausteine typischerweise die eigentliche Ein-/Ausgabe-Funktionalität bereitstellen.

Fazit

Der Port-Funktionsblock stellt eine elegante Lösung für die konfigurationsbasierte Pin-Verwaltung in Eclipse 4diac-Systemen dar. Durch die Verwendung von Adaptern ermöglicht er eine klare Trennung zwischen Konfiguration und Hardware-Zugriff, was die Wartbarkeit und Portierbarkeit von Steuerungsapplikationen verbessert.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 ESP32 & ESP32-S3 DevKit auf ms-muc-docs.de

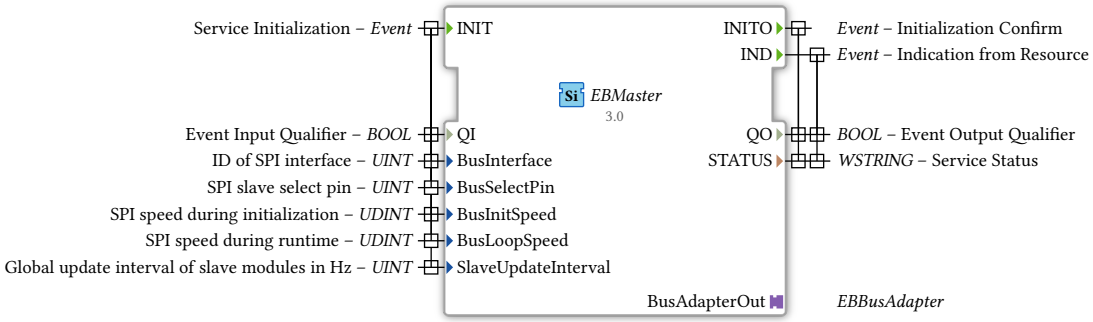
2.4.2.20 embrick

EBMaster

• – ◦ • – ◦ • – ◦ •

Einleitung

Der EBMaster Funktionsblock ist ein Service Interface Function Block für die Kommunikation mit EtherBrick-Modulen über SPI (Serial Peripheral Interface). Er dient als Master-Komponente zur Steuerung und Konfiguration der SPI-Bus-Kommunikation mit angeschlossenen Slave-Modulen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service Initialisierung - Initialisiert den SPI-Bus und konfiguriert die Kommunikationsparameter

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung - Bestätigt den erfolgreichen Abschluss der Initialisierung
- **IND**: Indikation von der Ressource - Signalisiert Statusänderungen oder Ereignisse von den angeschlossenen Modulen

Daten-Eingänge

- **QI** (BOOL): Event Input Qualifier - Aktiviert/Deaktiviert den Funktionsblock
- **BusInterface** (UINT): ID des SPI-Interface (Initialwert: 1)
- **BusSelectPin** (UINT): SPI Slave Select Pin (Initialwert: 49)
- **BusInitSpeed** (UDINT): SPI-Geschwindigkeit während der Initialisierung in Hz (Initialwert: 300000)
- **BusLoopSpeed** (UDINT): SPI-Geschwindigkeit während des Betriebs in Hz (Initialwert: 700000)
- **SlaveUpdateInterval** (UINT): Globale Update-Intervall der Slave-Module in Hz (Initialwert: 25)

Daten-Ausgänge

- **QO** (BOOL): Event Output Qualifier - Status der Ereignisausgabe
- **STATUS** (WSTRING): Service Status - Detaillierte Statusinformationen

Adapter

- **BusAdapterOut**: Adapter für die Kommunikation mit dem EtherBrick-Bus-System

Funktionsweise

Der EBMaster verwaltet die SPI-Kommunikation als Master-Komponente. Bei der Initialisierung (INIT-Event) konfiguriert er das SPI-Interface mit den angegebenen Parametern. Während des Betriebs steuert er die Datenübertragung zu den angeschlossenen Slave-Modulen mit dem konfigurierten Update-Intervall. Der Block ermöglicht unterschiedliche Geschwindigkeiten für Initialisierungs- und Betriebsphase.

Technische Besonderheiten

- Unterstützung verschiedener SPI-Geschwindigkeiten für Initialisierung und Betrieb

- Konfigurierbare Slave-Select-Pins
- Einstellbares globales Update-Intervall für Slave-Module
- WSTRING-basierte Statusrückmeldungen für detaillierte Fehlerdiagnose

Zustandsübersicht

Der Funktionsblock durchläuft folgende Zustände:

1. **Nicht initialisiert**: Wartet auf INIT-Event
2. **Initialisierung**: Konfiguriert SPI-Parameter bei INIT-Event
3. **Betriebsbereit**: Sendet INITO bei erfolgreicher Initialisierung
4. **Aktiv**: Kommuniziert mit Slaves gemäß konfiguriertem Intervall
5. **Fehler**: Sendet IND-Event mit STATUS-Information bei Problemen

Anwendungsszenarien

- Steuerung von EtherBrick-I/O-Modulen in Automatisierungssystemen
- SPI-basierte Kommunikation in Embedded-Systemen
- Industrielle Steuerungen mit modularen I/O-Erweiterungen
- Echtzeit-Datenerfassung mit konfigurierbaren Update-Raten

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu generischen SPI-Master-Blocks bietet EBMaster spezifische Optimierungen für EtherBrick-Module, einschließlich vorkonfigurierter Parameterwerte und spezialisierter Bus-Adapter-Schnittstellen.

Fazit

Der EBMaster Funktionsblock bietet eine robuste und konfigurierbare Lösung für die SPI-Master-Kommunikation mit EtherBrick-Modulen. Durch die Trennung von Initialisierungs- und Betriebsgeschwindigkeit sowie das konfigurierbare Update-Intervall ermöglicht er eine optimale Anpassung an verschiedene Anwendungsanforderungen.

EBSlave2181

• – ◦ • – ◦ • – ◦ •

Einleitung

Der EBSlave2181 ist ein Service Interface Function Block für die Kommunikation mit EtherBrick-Slave-Modulen. Dieser Funktionsblock ermöglicht die Konfiguration und Überwachung von digitalen Ein- und Ausgängen eines EtherBrick-Slave-Moduls und dient als Schnittstelle zwischen der Steuerungslogik und der physikalischen Feldbus-Kommunikation.

Schnittstellenstruktur

Ereignis-Eingänge
- **MAP**: Wird zur Konfiguration der Pin-Zuordnungen verwendet. Löst die Zuordnung der digitalen Ein- und Ausgänge aus.

Ereignis-Ausgänge
- **MAPO**: Bestätigt die erfolgreiche Pin-Zuordnung.
- **IND**: Zeigt Statusänderungen oder Fehler vom Slave-Modul an.

Daten-Eingänge
- **QI** (BOOL): Event Input Qualifier - Aktiviert/deaktiviert den Funktionsblock
- **DigitalInput_1** bis **DigitalInput_8** (WSTRING): Konfiguration der digitalen Eingänge 1-8
- **DigitalOutput_1** bis **DigitalOutput_8** (WSTRING): Konfiguration der digitalen Ausgänge 1-8
- **UpdateInterval** (UINT): Aktualisierungsintervall des Slave-Moduls in Hz (vom Master geerbt)

Daten-Ausgänge
- **QO** (BOOL): Event Output Qualifier - Status des Funktionsblocks
- **STATUS** (WSTRING): Service Status - Enthält Statusinformationen oder Fehlermeldungen

Adapter
- **BusAdapterOut** (Plug): Ausgehender Bus-Adapter für EtherBrick-Kommunikation
- **BusAdapterIn** (Socket): Eingehender Bus-Adapter für EtherBrick-Kommunikation

Funktionsweise
Der EBSlave2181 dient als Vermittler zwischen der IEC 61499-Steuerungslogik und einem EtherBrick-Slave-Modul. Beim Empfang des MAP-Ereignisses werden die konfigurierten digitalen Ein- und Ausgänge dem Slave-Modul zugeordnet. Das MAPO-Ereignis bestätigt den erfolgreichen Abschluss dieser Zuordnung. Das IND-Ereignis signalisiert Statusänderungen oder Fehlerzustände des Slave-Moduls.

Technische Besonderheiten
- Unterstützt 8 digitale Eingänge und 8 digitale Ausgänge
- Verwendet WSTRING-Datentyp für Pin-Konfiguration
- Ermöglicht konfigurierbare Aktualisierungsintervalle
- Integriert sich über standardisierte Bus-Adapter in die EtherBrick-Architektur
- Implementiert Qualifier-Pattern (QI/QO) für zuverlässige Statusverwaltung

Zustandsübersicht
Der Funktionsblock verfügt über folgende Betriebszustände:

- **Inaktiv**: QI = FALSE, Funktionsblock reagiert nicht auf Ereignisse
- **Konfigurierend**: Verarbeitung des MAP-Ereignisses und Zuordnung der Pins
- **Aktiv**: Erfolgreich konfiguriert, wartet auf IND-Ereignisse vom Slave
- **Fehler**: STATUS enthält Fehlerinformationen, QO kann FALSE sein

Anwendungsszenarien
- Anbindung von EtherBrick-Slave-Modulen in verteilten Automatisierungssystemen
- Konfiguration digitaler I/O-Module in Industrieanlagen
- Integration in Steuerungssysteme mit EtherBrick-Feldbus-Kommunikation
- Überwachung und Diagnose von Slave-Modulen in Echtzeit

⚖️ Vergleich mit ähnlichen Bausteinen
Im Vergleich zu generischen I/O-Funktionsblöcken bietet der EBSlave2181 spezifische Integration für EtherBrick-Systeme mit vordefinierten Bus-Adapter-Schnittstellen. Er ist spezialisiert auf die Konfiguration und Überwachung von Slave-Modulen und bietet direkte Unterstützung für die EtherBrick-Kommunikationsprotokolle.

Fazit

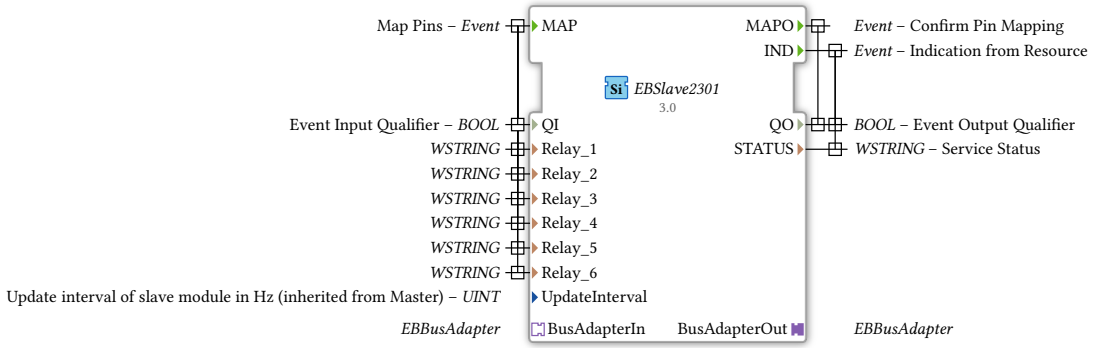
Der EBSlave2181 ist ein spezialisierter Service Interface Function Block, der eine zuverlässige und standardisierte Schnittstelle für die Integration von EtherBrick-Slave-Modulen in IEC 61499-basierte Steuerungssysteme bereitstellt. Seine strukturierte Schnittstelle und klare Zustandsverwaltung machen ihn zu einer robusten Lösung für industrielle Feldbus-Kommunikation.

EBSlave2301

• – ◦ • – ◦ • – ◦ •

Einleitung

Der EBSlave2301 ist ein Service Interface Function Block für die Kommunikation mit einem EtherBrick Slave-Modul. Dieser Funktionsblock dient als Schnittstelle zur Konfiguration und Steuerung von bis zu sechs Relais-Ausgängen über ein EtherBrick-Bussystem.



Schnittstellenstruktur

Ereignis-Eingänge

- **MAP**: Wird ausgelöst, um die Pin-Zuordnungen zu konfigurieren. Begleitet von den Datenvariablen Relay_1 bis Relay_6 und QI.

Ereignis-Ausgänge

- **MAPO**: Bestätigt die erfolgreiche Pin-Zuordnung. Wird mit QO ausgegeben.
- **IND**: Meldung vom Ressourcen-Interface. Wird mit QO und STATUS ausgegeben.

Daten-Eingänge

- **QI** (BOOL): Event Input Qualifier - Aktiviert/deaktiviert den Funktionsblock
- **Relay_1** bis **Relay_6** (WSTRING): Konfiguration der Relais-Pins
- **UpdateInterval** (UINT): Aktualisierungsintervall des Slave-Moduls in Hz (vom Master geerbt)

Daten-Ausgänge

- **QO** (BOOL): Event Output Qualifier - Status der Operation
- **STATUS** (WSTRING): Service-Statusinformationen

Adapter

- **BusAdapterOut** (Plug): Ausgehender Bus-Adapter vom Typ EBBusAdapter
- **BusAdapterIn** (Socket): Eingehender Bus-Adapter vom Typ EBBusAdapter

Funktionsweise

Der EBSlave2301 ermöglicht die Konfiguration von bis zu sechs Relais-Ausgängen über das MAP-Ereignis. Bei Aktivierung durch QI werden die Relais-Parameter über die Bus-Adapter an das EtherBrick-System übertragen. Das MAPO-Ereignis bestätigt den erfolgreichen Abschluss der Konfiguration, während IND-Ereignisse Statusinformationen vom Ressourcen-Interface liefern.

Technische Besonderheiten

- Unterstützt WSTRING-Datentypen für Relais-Konfiguration
- Verwendet ein Adapter-basiertes Bussystem für die Kommunikation
- UpdateInterval wird vom Master-System geerbt

- Bietet qualifizierte Ereignissteuerung über QI und QO

Zustandsübersicht

Der Funktionsblock wechselt zwischen folgenden Zuständen:

- Inaktiv: QI = FALSE
- Konfigurationsmodus: Bei MAP-Ereignis mit QI = TRUE
- Betriebsbereit: Nach erfolgreicher MAPO-Bestätigung
- Fehlerzustand: Bei IND-Ereignis mit Statusinformationen

Anwendungsszenarien

- Steuerung von Relais-basierten Aktoren in Automatisierungssystemen
- Integration in EtherBrick-basierte IO-Systeme
- Verteiltes Steuerungssystem mit Master-Slave-Architektur
- Anwendungen mit konfigurierbaren Relais-Zuordnungen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen digitalen Ausgangsbausteinen bietet der EBSlave2301:

- Konfigurierbare Pin-Zuordnungen zur Laufzeit
- Adapter-basierte Buskommunikation
- Qualifizierte Ereignissteuerung
- Statusrückmeldungen vom Ressourcen-Interface

Fazit

Der EBSlave2301 ist ein spezialisierter Funktionsblock für die flexible Steuerung von Relais-Ausgängen in EtherBrick-Systemen. Durch seine konfigurierbaren Schnittstellen und Adapter-basierte Architektur eignet er sich besonders für anspruchsvolle Automatisierungsanwendungen mit dynamischen IO-Konfigurationen.

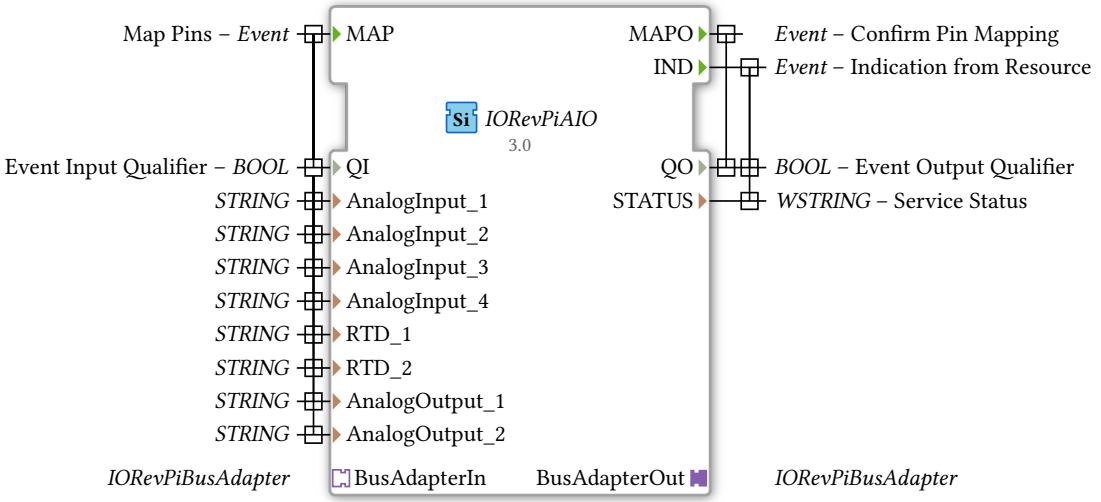
2.4.2.21 revpi

🔌 IORevPiAIO

• – ◦ • – ◦ • – ◦ •

Einleitung

Der IORevPiAIO Funktionsblock stellt eine Schnittstelle für das Analog I/O-Modul des Revolution Pi von KUNBUS GmbH dar. Dieser Baustein ermöglicht die Ansteuerung und Auslesung von analogen Eingängen und Ausgängen sowie RTD-Sensoren (Resistance Temperature Detector) über das Revolution Pi System.



Schnittstellenstruktur

Ereignis-Eingänge
- **MAP**: Startet die Pin-Zuordnung für alle konfigurierten analogen Ein- und Ausgänge

Ereignis-Ausgänge
- **MAPO**: Bestätigt die erfolgreiche Pin-Zuordnung
- **IND**: Zeigt Statusinformationen vom Ressourcen-Manager an

Daten-Eingänge
- **QI** (BOOL): Event Input Qualifier - Aktiviert/Deaktiviert den Funktionsblock
- **AnalogInput_1** (STRING): Konfiguration für analogen Eingang 1
- **AnalogInput_2** (STRING): Konfiguration für analogen Eingang 2
- **AnalogInput_3** (STRING): Konfiguration für analogen Eingang 3
- **AnalogInput_4** (STRING): Konfiguration für analogen Eingang 4
- **RTD_1** (STRING): Konfiguration für RTD-Sensor 1
- **RTD_2** (STRING): Konfiguration für RTD-Sensor 2
- **AnalogOutput_1** (STRING): Konfiguration für analogen Ausgang 1
- **AnalogOutput_2** (STRING): Konfiguration für analogen Ausgang 2

Daten-Ausgänge
- **QO** (BOOL): Event Output Qualifier - Status der Operation
- **STATUS** (WSTRING): Detaillierte Statusinformationen des Dienstes

Adapter
- **BusAdapterIn** (Socket): Eingangsadapter für Revolution Pi Bus-Kommunikation
- **BusAdapterOut** (Plug): Ausgangsadapter für Revolution Pi Bus-Kommunikation

Funktionsweise
Der IORevPiAIO Funktionsblock verwaltet die Kommunikation mit dem Revolution Pi Analog I/O-Modul. Beim Empfang des MAP-Ereignisses werden alle konfigurierten analogen Ein- und Ausgänge sowie RTD-Sensoren gemäß den String-Parametern initialisiert und zugeordnet. Der Block verwendet spezielle Bus-Adapter für die Kommunikation mit der Revolution Pi Hardware.

Technische Besonderheiten
- Unterstützt bis zu 4 analoge Eingänge
- Unterstützt bis zu 2 analoge Ausgänge
- Integrierte RTD-Sensorunterstützung (2 Kanäle)
- String-basierte Konfiguration der Pin-Zuordnung
- Bus-Adapter-Architektur für hardwarenahe Kommunikation

Zustandsübersicht
Der Funktionsblock verfügt über folgende Betriebszustände:

- **Inaktiv**: QI = FALSE, keine Operationen
- **Bereit**: QI = TRUE, wartet auf MAP-Ereignis
- **Mapping**: Verarbeitet Pin-Zuordnung nach MAP-Ereignis
- **Aktiv**: Erfolgreich konfiguriert, bereit für Datenoperationen

Anwendungsszenarien
- Industrielle Prozessautomation mit analogen Sensoren
- Temperaturmessung mit RTD-Sensoren
- Analoges Signalverarbeitung in Steuerungssystemen
- Revolution Pi-basierte Automatisierungslösungen

⚖️ Vergleich mit ähnlichen Bausteinen
Im Vergleich zu generischen Analog-I/O-Blöcken bietet IORevPiAIO spezifische Integration für die Revolution Pi Hardware und unterstützt zusätzlich RTD-Temperatursensoren. Die Bus-Adapter-Architektur ermöglicht eine effiziente Kommunikation mit dem Revolution Pi System.

Fazit

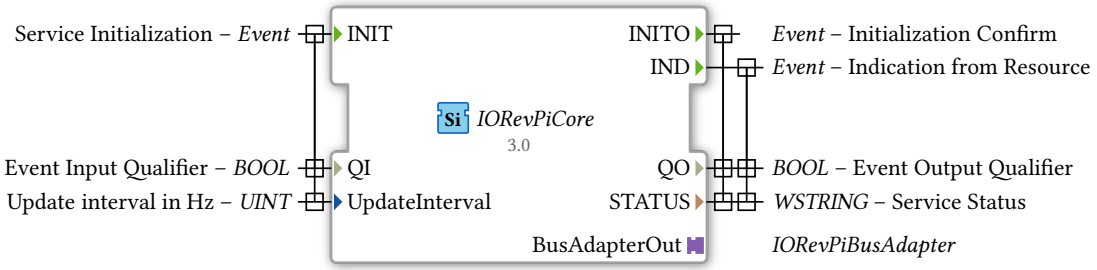
Der IORevPiAIO Funktionsblock bietet eine zuverlässige und spezialisierte Schnittstelle für Analog-I/O-Operationen auf Revolution Pi Systemen. Durch die integrierte Unterstützung für RTD-Sensoren und die flexible Konfiguration über String-Parameter eignet er sich besonders für industrielle Automatisierungsanwendungen mit analogen Mess- und Steueraufgaben.

🔌 IORevPiCore

• – ◦ • – ◦ • – ◦ •

Einleitung

Der IORevPiCore Funktionsblock dient als Kernmodul für die Revolution Pi Hardware von KUNBUS GmbH. Er ermöglicht die grundlegende Kommunikation und Konfiguration der Revolution Pi I/O-Module innerhalb einer 4diac-basierten Steuerungsanwendung.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsereignis zur Aktivierung des Funktionsblocks

Ereignis-Ausgänge

- **INITO**: Bestätigung der erfolgreichen Initialisierung
- **IND**: Indikationsereignis bei Statusänderungen vom Ressourcen

Daten-Eingänge

- **QI** (BOOL): Ereigniseingangsqualifizierer
- **UpdateInterval** (UINT): Aktualisierungsintervall in Hz (Standardwert: 25)

Daten-Ausgänge

- **QO** (BOOL): Ereignisausgangsqualifizierer
- **STATUS** (WSTRING): Servicestatus als Unicode-String

Adapter

- **BusAdapterOut**: Ausgangsadapter für die Revolution Pi Bus-Kommunikation (Typ: eclipse4diac::io::revpi::IORevPiBusAdapter)

Funktionsweise

Der IORevPiCore Block initialisiert die Kommunikation mit der Revolution Pi Hardware beim Empfang des INIT-Ereignisses. Nach erfolgreicher Initialisierung wird INITO mit dem entsprechenden Status ausgegeben. Der Block überwacht kontinuierlich den Hardwarezustand und gibt bei Änderungen IND-Ereignisse aus. Das UpdateInterval bestimmt die Abtastrate für die Hardwareüberwachung.

Technische Besonderheiten

- Standard-Updateintervall von 25 Hz
- Unicode-String-Ausgabe für Statusinformationen
- Integration über spezifischen Bus-Adapter für Revolution Pi Hardware
- Automatische Hardware-Erkennung und -Initialisierung

Zustandsübersicht

1. **Inaktiv**: Block wartet auf INIT-Ereignis

2. **Initialisierung**: Verarbeitung der Konfigurationsparameter
3. **Aktiv**: Überwachung der Revolution Pi Hardware und Ausgabe von IND-Ereignissen bei Statusänderungen

Anwendungsszenarien

- Steuerung von Revolution Pi I/O-Modulen in industriellen Automatisierungslösungen
- Integration von Revolution Pi Hardware in 4diac-basierte Steuerungssysteme
- Überwachung und Diagnose von Revolution Pi Komponenten

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu generischen I/O-Blöcken bietet IORevPiCore spezifische Optimierungen für die Revolution Pi Hardware-Plattform, einschließlich spezialisierter Bus-Kommunikation und hardwarenaher Konfigurationsparameter.

Fazit

Der IORevPiCore Funktionsblock stellt eine zuverlässige und einfach zu konfigurierende Schnittstelle für die Integration von Revolution Pi Hardware in 4diac-basierte Automatisierungslösungen dar. Durch die standardisierte Schnittstelle und flexible Konfigurationsmöglichkeiten ermöglicht er eine effiziente Anbindung industrieller I/O-Komponenten.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

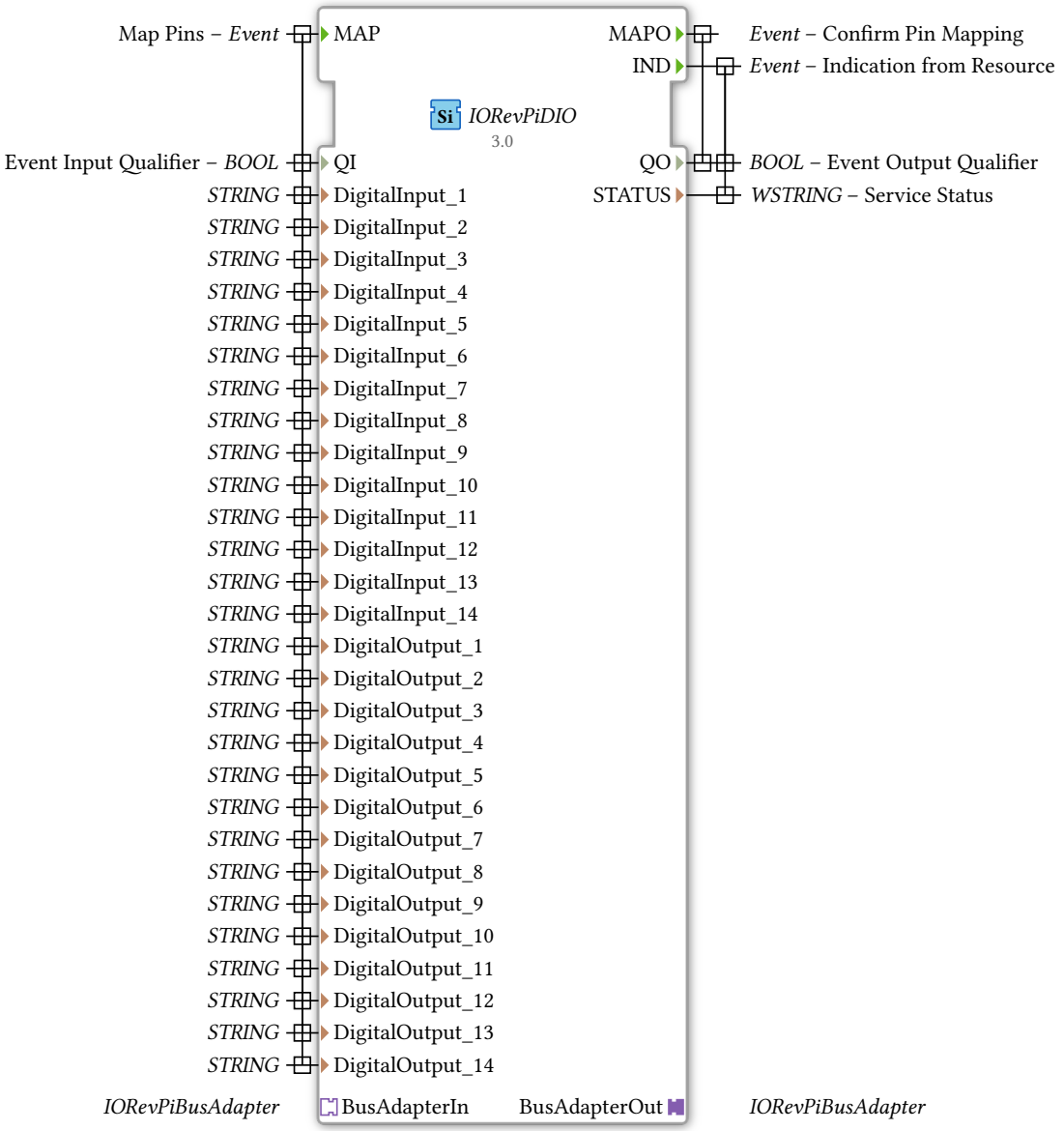
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

🔌 IORevPiDIO

• – ◦ • – ◦ • – ◦ •

Einleitung

Der IORevPiDIO Funktionsblock ist ein digitales Ein-/Ausgabemodul für Revolution Pi Systeme von KUNBUS GmbH. Er ermöglicht die Ansteuerung und Überwachung digitaler Eingänge und Ausgänge über das Revolution Pi System und dient als Schnittstelle zwischen 4diac-IDE und der Revolution Pi Hardware.



Schnittstellenstruktur

Ereignis-Eingänge

- **MAP**: Ereignis zum Mappen der Pins mit allen DigitalInput- und DigitalOutput-Datenvariablen sowie QI

Ereignis-Ausgänge

- **MAPO**: Bestätigungsereignis für erfolgreiches Pin-Mapping mit QO
- **IND**: Indikationsereignis vom Resource mit QO und STATUS

Daten-Eingänge

- **QI** (BOOL): Event Input Qualifier
- **DigitalInput_1** bis **DigitalInput_14** (STRING): Konfiguration der digitalen Eingänge 1-14
- **DigitalOutput_1** bis **DigitalOutput_14** (STRING): Konfiguration der digitalen Ausgänge 1-14

Daten-Ausgänge

- **QO** (BOOL): Event Output Qualifier
- **STATUS** (WSTRING): Service Status

Adapter

- **BusAdapterOut** (Plug): Ausgehender Bus-Adapter vom Typ IORevPiBusAdapter
- **BusAdapterIn** (Socket): Eingehender Bus-Adapter vom Typ IORevPiBusAdapter

Funktionsweise

Der Funktionsblock ermöglicht die Konfiguration und Steuerung von bis zu 14 digitalen Eingängen und 14 digitalen Ausgängen eines Revolution Pi Systems. Durch das MAP-Ereignis werden die Pin-Konfigurationen übergeben und mit MAPO bestätigt. Der IND-Ausgang signalisiert Statusänderungen und Fehlerzustände. Die Kommunikation mit der Hardware erfolgt über die IORevPiBusAdapter-Schnittstelle.

Technische Besonderheiten

- Unterstützt bis zu 14 digitale Eingänge und 14 digitale Ausgänge
- Verwendet STRING-Typ für Pin-Konfiguration
- Integriert Bus-Adapter für Revolution Pi Kommunikation
- Bietet umfassende Statusrückmeldung über WSTRING

Zustandsübergänge

1. **Initialisierung**: Warten auf MAP-Ereignis mit Konfigurationsdaten
2. **Konfiguration**: Verarbeitung der Pin-Zuordnungen und Bestätigung via MAPO
3. **Betrieb**: Überwachung der digitalen Ein-/Ausgänge und Signalgebung via IND
4. **Fehlerbehandlung**: Statusmeldung bei Kommunikationsproblemen mit der Hardware

Anwendungsszenarien

- Industrielle Automatisierung mit Revolution Pi
- Digitale Signalverarbeitung in Steuerungssystemen
- Anbindung von Sensoren und Aktoren an 4diac-basierte Steuerungen
- Prototyping und Entwicklung von IoT-Lösungen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen I/O-Bausteinen bietet IORevPiDIO spezifische Unterstützung für Revolution Pi Hardware mit einer hohen Anzahl an konfigurierbaren Ein-/Ausgängen. Die Integration des Bus-Adapters ermöglicht eine direkte Kommunikation mit der Revolution Pi Plattform.

Fazit

Der IORevPiDIO Funktionsblock stellt eine leistungsstarke Schnittstelle für digitale Ein-/Ausgänge in Revolution Pi Systemen dar und ermöglicht eine einfache Integration in 4diac-basierte Automatisierungslösungen. Die umfangreiche Konfigurationsmöglichkeit und zuverlässige Statusrückmeldung machen ihn zu einer robusten Lösung für industrielle Anwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

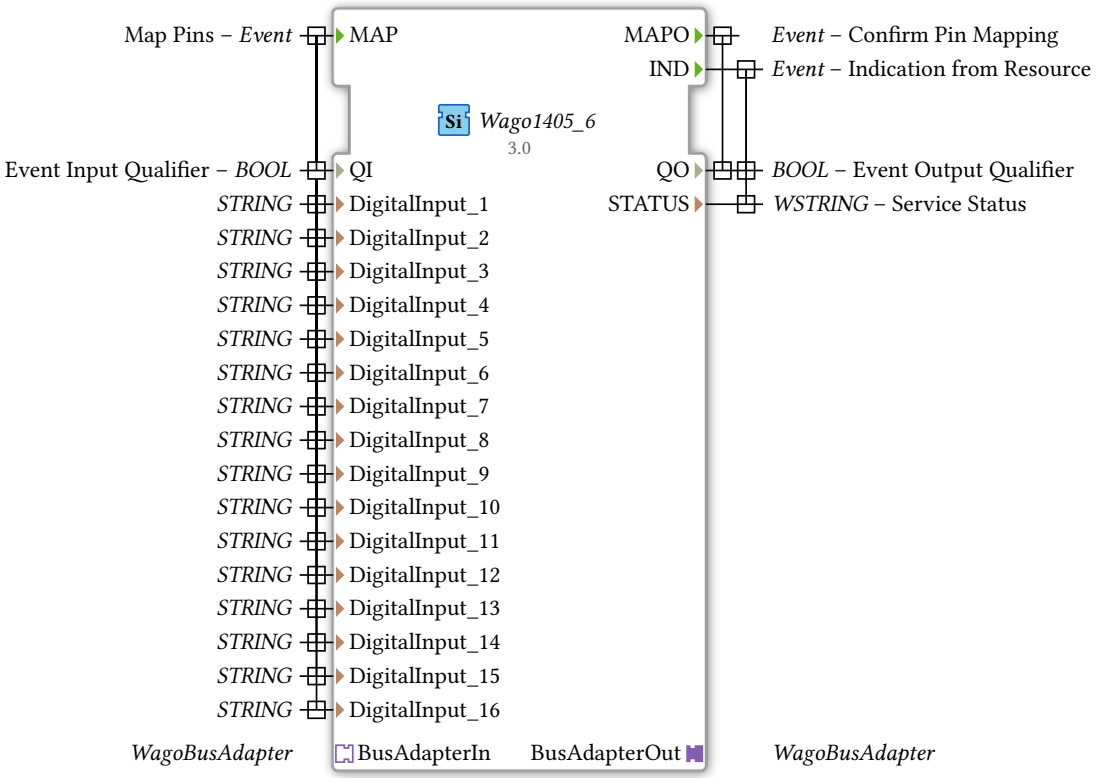
2.4.2.22 wago

Wago1405_6

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Wago1405_6 ist ein Service Interface Function Block für die Anbindung an Wago-Bussysteme. Dieser Funktionsblock dient als Schnittstelle zur Konfiguration und Steuerung von digitalen Eingängen eines Wago-IO-Systems und ermöglicht die Kommunikation über spezielle Busadapter.



Schnittstellenstruktur

Ereignis-Eingänge

- **MAP**: Map Pins - Dient zur Konfiguration der Pin-Zuordnungen für digitale Eingänge

Ereignis-Ausgänge

- **MAPO**: Confirm Pin Mapping - Bestätigt die erfolgreiche Pin-Zuordnung
- **IND**: Indication from Resource - Meldung vom Ressourcen-Interface

Daten-Eingänge

- **QI**: Event Input Qualifier (BOOL) - Qualifiziert die Ereigniseingänge
- **DigitalInput_1** bis **DigitalInput_16**: Pin-Konfiguration (STRING) - Zuordnung der digitalen Eingänge 1-16

Daten-Ausgänge

- **QO**: Event Output Qualifier (BOOL) - Qualifiziert die Ereignisausgänge
- **STATUS**: Service Status (WSTRING) - Statusinformationen des Service-Interfaces

Adapter

- **BusAdapterOut**: Ausgehender Busadapter (Plug) vom Typ eclipse4diac::io::wago::WagoBusAdapter
- **BusAdapterIn**: Eingehender Busadapter (Socket) vom Typ eclipse4diac::io::wago::WagoBusAdapter

Funktionsweise

Der Wago1405_6 Funktionsblock ermöglicht die Konfiguration von 16 digitalen Eingängen über das MAP-Ereignis. Bei Aktivierung des MAP-Ereignisses werden die Pin-Zuordnungen über die DigitalInput_x Parameter konfiguriert. Nach erfolgreicher Konfiguration erfolgt eine Bestätigung über das MAPO-Ereignis. Das IND-Ereignis dient zur asynchronen Kommunikation mit dem zugrundeliegenden Ressourcen-Interface und liefert Statusinformationen.

Technische Besonderheiten

- Unterstützt 16 digitale Eingänge
- Verwendet STRING-Typ für Pin-Konfiguration

- Implementiert bidirektionale Busadapter-Kommunikation
- Bietet umfangreiche Statusrückmeldungen über WSTRING

Zustandsübersicht

Der Funktionsblock arbeitet mit folgenden Hauptzuständen:

1. **Inaktiv**: Warte auf MAP-Ereignis
2. **Konfiguration**: Verarbeitung der Pin-Zuordnungen
3. **Bestätigung**: Senden der MAPO-Bestätigung
4. **Betrieb**: Asynchrone Statusmeldungen über IND

Anwendungsszenarien

- Anbindung von Wago-750-1405/750-1406 IO-Systemen
- Konfiguration digitaler Eingänge in Automatisierungssystemen
- Integration in verteilte Steuerungssysteme mit 4diac
- Industrielle IO-Konfiguration und -Überwachung

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu generischen IO-Funktionsblöcken bietet der Wago1405_6 spezifische Anpassungen für Wago-Bussysteme und eine umfangreichere Konfigurationsmöglichkeit für digitale Eingänge. Die Integration der Busadapter ermöglicht eine nahtlose Einbindung in Wago-spezifische Kommunikationsarchitekturen.

Fazit

Der Wago1405_6 Funktionsblock stellt eine spezialisierte Lösung für die Anbindung und Konfiguration von Wago-IO-Systemen dar. Durch die umfangreiche Konfigurationsmöglichkeit für 16 digitale Eingänge und die integrierte Busadapter-Schnittstelle bietet er eine effiziente Lösung für industrielle Automatisierungsanwendungen mit Wago-Komponenten.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

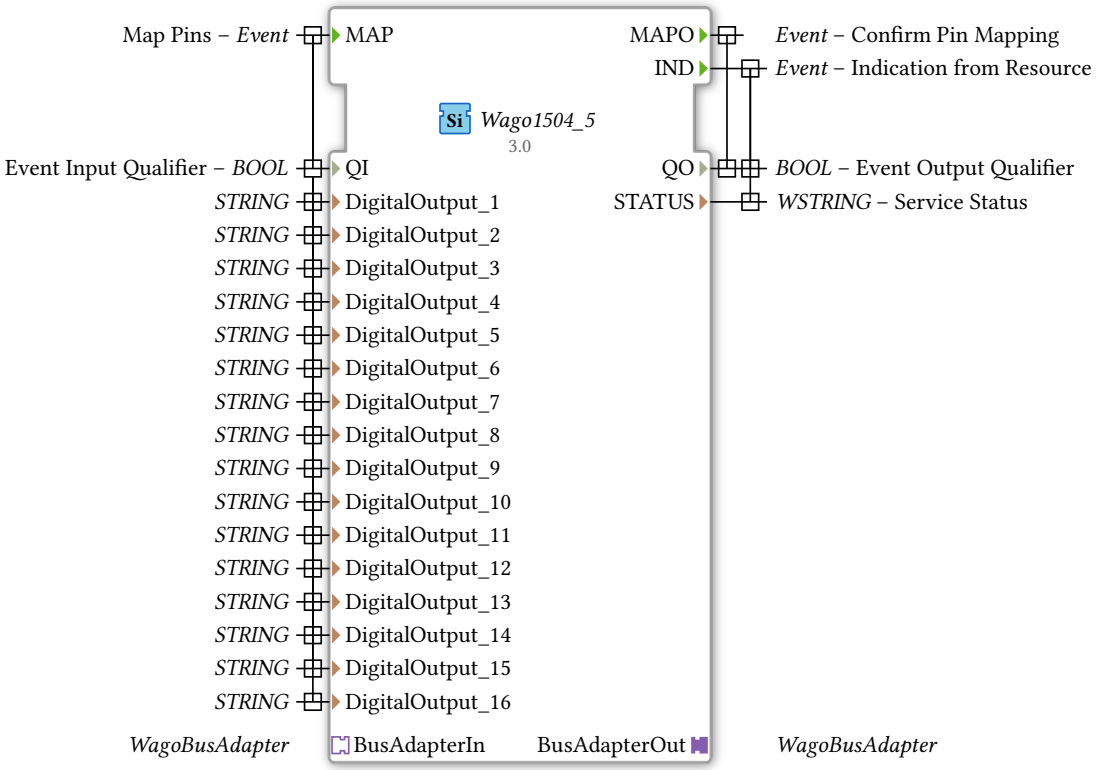
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Wago1504_5

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Wago1504_5 ist ein Service Interface Function Block für die Ansteuerung digitaler Ausgänge eines Wago 750-1504/005-000 Moduls. Dieser Funktionsblock ermöglicht die Konfiguration und Steuerung von 16 digitalen Ausgängen über eine Busadapter-Schnittstelle.



Schnittstellenstruktur

Ereignis-Eingänge

- **MAP**: Map Pins - Initialisiert die Pin-Zuordnung der digitalen Ausgänge

Ereignis-Ausgänge

- **MAPO**: Confirm Pin Mapping - Bestätigt die erfolgreiche Pin-Zuordnung
- **IND**: Indication from Resource - Meldung vom Ressourcen-Adapter

Daten-Eingänge

- **QI**: Event Input Qualifier (BOOL) - Qualifiziert die Ereigniseingänge
- **DigitalOutput_1** bis **DigitalOutput_16** (STRING) - Konfigurationsstrings für die jeweiligen digitalen Ausgänge

Daten-Ausgänge

- **QO**: Event Output Qualifier (BOOL) - Qualifiziert die Ereignisausgänge
- **STATUS**: Service Status (WSTRING) - Statusinformationen des Service

Adapter

- **BusAdapterOut** (Plug): Ausgehender WagoBusAdapter
- **BusAdapterIn** (Socket): Eingehender WagoBusAdapter

Funktionsweise

Der Wago1504_5 Funktionsblock dient als Schnittstelle zwischen der 4diac-Steuerung und dem Wago 750-1504/005-000 E/A-Modul. Beim Empfang des MAP-Ereignisses werden die 16 digitalen Ausgänge gemäß der übergebenen String-Konfiguration zugeordnet. Die erfolgreiche Zuordnung wird durch das MAPO-Ereignis bestätigt. Der IND-Ausgang dient zur Rückmeldung von Statusinformationen aus dem Ressourcen-Adapter.

Technische Besonderheiten

- Unterstützt 16 unabhängige digitale Ausgänge
- Verwendet String-basierte Konfiguration für die Pin-Zuordnung

- Implementiert Busadapter-Kommunikation für die Anbindung an Wago-Hardware
- Bietet Statusrückmeldungen über WSTRING-Datentyp

Zustandsübersicht

Der Funktionsblock wechselt zwischen folgenden Zuständen:

1. **Inaktiv**: Warten auf MAP-Ereignis
2. **Mapping**: Verarbeitung der Pin-Zuordnung
3. **Bestätigung**: Senden von MAPO bei erfolgreicher Zuordnung
4. **Bereit**: Betriebsbereit für IND-Meldungen

Anwendungsszenarien

- Ansteuerung von Relais und Aktoren in Automatisierungsanlagen
- Digitale Signalausgabe in Prozesssteuerungen
- Integration von Wago 750-1504/005-000 Modulen in 4diac-basierte Steuerungen
- Industrielle Automatisierung mit verteilten E/A-Systemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu generischen digitalen Ausgangs-Blöcken bietet Wago1504_5:

- Spezifische Anpassung für Wago-Hardware
- Erweiterte Konfigurationsmöglichkeiten über String-Parameter
- Integrierte Busadapter-Kommunikation
- Statusrückmeldungen mit detaillierten Informationen

Fazit

Der Wago1504_5 Funktionsblock stellt eine zuverlässige und spezialisierte Schnittstelle für die Ansteuerung von Wago 750-1504/005-000 Digitalausgangsmodulen dar. Durch die strukturierte Ereignissteuerung und umfassende Statusrückmeldungen eignet er sich besonders für robuste industrielle Automatisierungslösungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

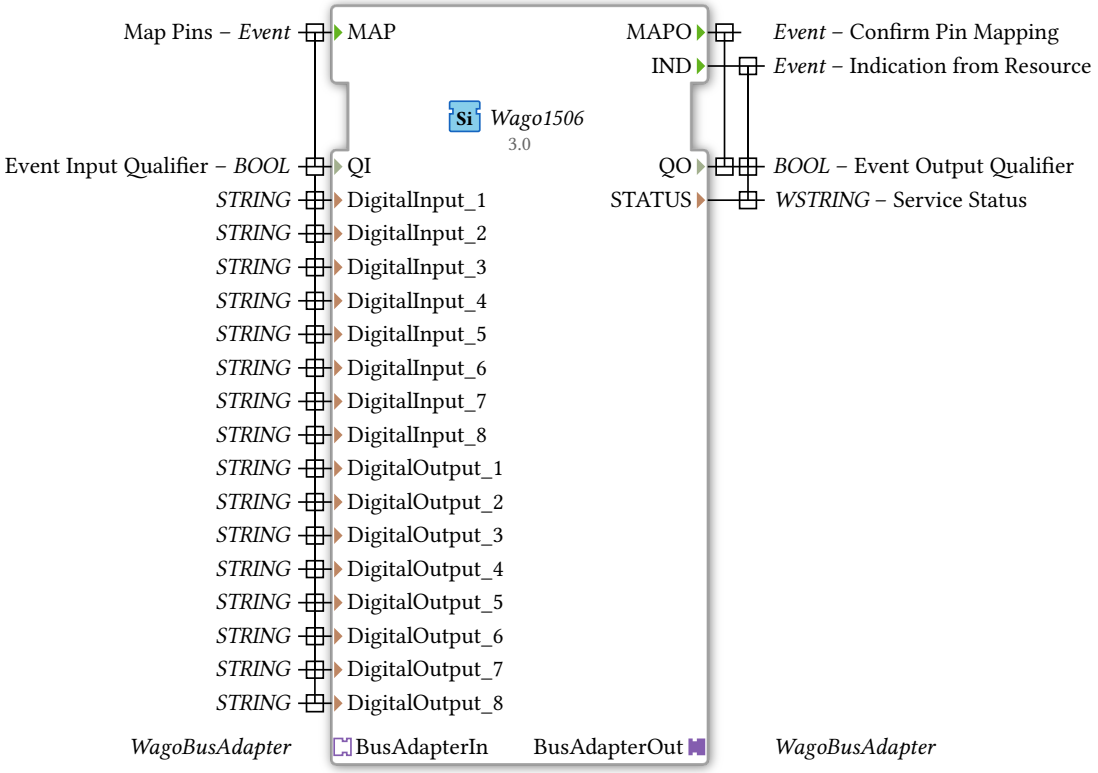
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

Wago1506

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Wago1506 ist ein Service Interface Function Block für die Ansteuerung von Wago 750-1506 I/O-Modulen. Dieser Funktionsblock ermöglicht die Konfiguration und Steuerung von digitalen Ein- und Ausgängen über einen Busadapter und dient als Schnittstelle zwischen der Steuerungslogik und der physischen Hardware.



Schnittstellenstruktur

Ereignis-Eingänge

- **MAP**: Wird verwendet, um die Pin-Belegung zu konfigurieren. Löst die Zuordnung der digitalen Ein- und Ausgänge aus.

Ereignis-Ausgänge

- **MAPO**: Bestätigt die erfolgreiche Pin-Zuordnung.
- **IND**: Zeigt Statusinformationen von der Ressource an.

Daten-Eingänge

- **QI** (BOOL): Event Input Qualifier - Aktiviert/deaktiviert den Funktionsblock
- **DigitalInput_1** bis **DigitalInput_8** (STRING): Konfiguration der digitalen Eingänge 1-8
- **DigitalOutput_1** bis **DigitalOutput_8** (STRING): Konfiguration der digitalen Ausgänge 1-8

Daten-Ausgänge

- **QO** (BOOL): Event Output Qualifier - Status der Operation
- **STATUS** (WSTRING): Detaillierte Statusinformationen des Service

Adapter

- **BusAdapterOut** (Plug): Ausgehender Busadapter für Wago-Kommunikation
- **BusAdapterIn** (Socket): Eingehender Busadapter für Wago-Kommunikation

Funktionsweise

Der Wago1506 FB verwaltet die Kommunikation mit einem Wago 750-1506 I/O-Modul. Beim Empfang des MAP-Ereignisses werden die konfigurierten Pin-Zuordnungen an das Bus-Adapter-Interface übergeben. Der FB überwacht den Status der Hardware und gibt bei Änderungen IND-Ereignisse mit entsprechenden Statusinformationen aus.

Technische Besonderheiten

- Unterstützt 8 digitale Eingänge und 8 digitale Ausgänge
- Verwendet STRING-Typen für die Pin-Konfiguration, was flexible Adressierung ermöglicht

- Integriert spezifische Wago-Busadapter für zuverlässige Hardwarekommunikation
- Bietet umfangreiche Statusrückmeldungen über WSTRING

Zustandsübersicht

1. **Inaktiv**: QI = FALSE, FB wartet auf Aktivierung
2. **Konfiguriert**: Nach MAP-Ereignis mit erfolgreicher Pin-Zuordnung
3. **Aktiv**: QI = TRUE, FB kommuniziert mit Hardware
4. **Fehler**: Bei Kommunikationsproblemen oder ungültigen Konfigurationen

Anwendungsszenarien

- Ansteuerung von Wago 750-1506 Digital-I/O-Modulen
- Industrielle Automatisierungsanwendungen
- Maschinensteuerung mit digitalen Signalen
- Prozessautomation mit verteilten I/Os

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu generischen I/O-Funktionsblöcken bietet der Wago1506 spezifische Optimierungen für Wago-Hardware, insbesondere für das 750-1506 Modul. Die Integration spezieller Busadapter gewährleistet eine zuverlässigere Kommunikation als universelle Lösungen.

Fazit

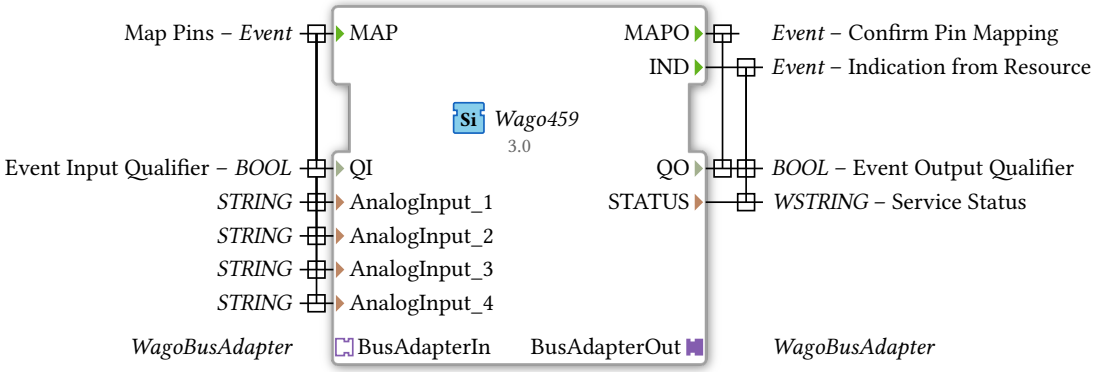
Der Wago1506 Funktionsblock stellt eine robuste und spezialisierte Lösung für die Ansteuerung von Wago 750-1506 I/O-Modulen dar. Durch die klare Trennung von Konfiguration (MAP) und Betrieb (IND) sowie die umfassende Statusrückmeldung eignet er sich ideal für industrielle Automatisierungsprojekte mit hohen Zuverlässigkeitsanforderungen.

Wago459

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Wago459 ist ein Service Interface Function Block für die Anbindung von Wago-750-459 Analog-Eingangsmodulen in 4diac-Systemen. Dieser Funktionsblock ermöglicht die Konfiguration und Überwachung von bis zu vier analogen Eingangskanälen und stellt die Kommunikation mit dem Wago-Bussystem her.



Schnittstellenstruktur

Ereignis-Eingänge

- **MAP**: Startet die Pin-Zuordnung für die analogen Eingänge. Wird mit den Datenvariablen AnalogInput_1 bis AnalogInput_4 und QI verknüpft.

Ereignis-Ausgänge

- **MAPO**: Bestätigt die erfolgreiche Pin-Zuordnung. Wird mit QO verknüpft.
- **IND**: Liefert Statusinformationen vom Ressourcen-Manager. Wird mit QO und STATUS verknüpft.

Daten-Eingänge

- **QI** (BOOL): Event Input Qualifier - Steuert die Aktivierung des Funktionsblocks
- **AnalogInput_1** (STRING): Konfiguration für analogen Eingang 1
- **AnalogInput_2** (STRING): Konfiguration für analogen Eingang 2
- **AnalogInput_3** (STRING): Konfiguration für analogen Eingang 3
- **AnalogInput_4** (STRING): Konfiguration für analogen Eingang 4

Daten-Ausgänge

- **QO** (BOOL): Event Output Qualifier - Zeigt den Betriebszustand an
- **STATUS** (WSTRING): Detaillierte Statusinformationen des Service

Adapter

- **BusAdapterOut** (Plug): Ausgehende Bus-Adapter-Verbindung vom Typ WagoBusAdapter
- **BusAdapterIn** (Socket): Eingehende Bus-Adapter-Verbindung vom Typ WagoBusAdapter

Funktionsweise

Der Wago459-Funktionsblock dient als Schnittstelle zwischen der 4diac-Steuerung und dem Wago-750-459 Analog-Eingangsmodul. Beim Empfang des MAP-Ereignisses werden die konfigurierten Pin-Zuordnungen (AnalogInput_1 bis AnalogInput_4) an das Bus-System übertragen. Der Block bestätigt erfolgreiche Zuordnungen über MAPO und liefert kontinuierlich Statusinformationen über IND-Ereignisse.

Technische Besonderheiten

- Unterstützt bis zu vier analoge Eingangskanäle
- Verwendet STRING-Datentypen für flexible Pin-Konfiguration
- Bietet umfangreiche Statusrückmeldungen via WSTRING
- Implementiert ein bidirektionales Adapter-System für Bus-Kommunikation
- Enthält Qualifier-Variablen (QI/QO) für zuverlässige Zustandsverwaltung

Zustandsübersicht

Der Funktionsblock verfügt über folgende Betriebszustände:

- Inaktiv: Wartet auf MAP-Ereignis mit aktiviertem QI
- Konfiguration: Verarbeitet Pin-Zuordnungen nach MAP-Ereignis
- Betrieb: Überwacht kontinuierlich den Modulstatus und sendet IND-Ereignisse
- Fehler: Bei Problemen wird STATUS mit Fehlerinformationen gefüllt

Anwendungsszenarien

- Anbindung von Wago-750-459 Analog-Eingangsmodulen in Automatisierungssystemen
- Messwerterfassung von analogen Sensoren (0-10V, 0-20mA, 4-20mA)
- Prozessüberwachung in industriellen Steuerungssystemen
- Integration in dezentrale Peripheriesysteme mit Wago-Busarchitektur

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu generischen IO-Funktionsblöcken bietet Wago459:

- Spezifische Optimierung für Wago-750-459 Hardware
- Vorkonfigurierte Adapter für Wago-Bussysteme
- Direkte Unterstützung für alle vier Kanäle des Moduls
- Integrierte Statusüberwachung und Fehlerbehandlung

Fazit

Der Wago459-Funktionsblock stellt eine zuverlässige und spezialisierte Lösung für die Integration von Wago-750-459 Analog-Eingangsmodulen in 4diac-basierte Steuerungssysteme dar. Durch seine klare Schnittstellendefinition und umfassende Statusrückmeldungen ermöglicht er eine effiziente Implementierung analoger Messwertverarbeitung in industriellen Automatisierungsprojekten.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

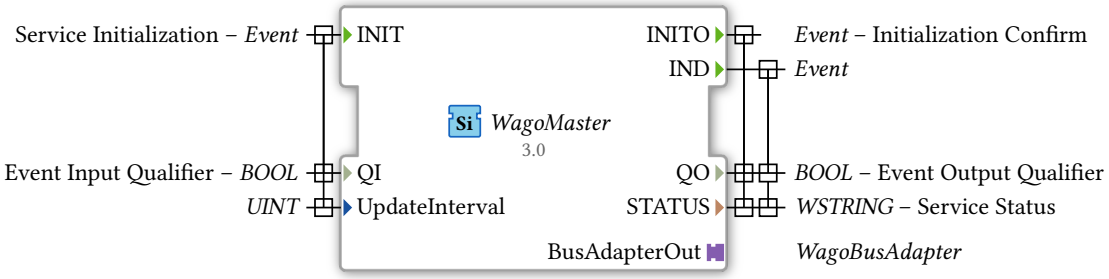
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

WagoMaster

• – ◦ • – ◦ • – ◦ •

Einleitung

Der WagoMaster ist ein Service Interface Function Block zur Steuerung und Überwachung von Wago-Bussystemen. Dieser Baustein dient als Master-Komponente für die Kommunikation mit Wago-IO-Systemen und ermöglicht die Initialisierung und Verwaltung der Busverbindung.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service Initialisierung
  - Verknüpft mit: QI, UpdateInterval

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
  - Verknüpft mit: QO, STATUS
- **IND**: Indikationsereignis
  - Verknüpft mit: STATUS, QO

Daten-Eingänge

- **QI** (BOOL): Event Input Qualifier - Steuert die Aktivierung des Service
- **UpdateInterval** (UINT): Definiert das Aktualisierungsintervall für die Buskommunikation

Daten-Ausgänge

- **QO** (BOOL): Event Output Qualifier - Zeigt den Status der Service-Ausführung an
- **STATUS** (WSTRING): Service Status - Enthält Statusinformationen und Fehlermeldungen

Adapter

- **BusAdapterOut**: WagoBusAdapter - Stellt die Verbindung zum Wago-Bussystem her

Funktionsweise

Der WagoMaster initialisiert und verwaltet die Kommunikation mit Wago-Bussystemen. Bei Aktivierung durch das INIT-Ereignis wird der Service gestartet und über das definierte UpdateInterval die Buskommunikation periodisch durchgeführt. Der Baustein meldet den Initialisierungserfolg über INITO und liefert kontinuierlich Statusinformationen über IND-Ereignisse.

Technische Besonderheiten

- Unterstützt WSTRING für detaillierte Statusmeldungen
- Konfigurierbares Update-Intervall für flexible Anpassung an verschiedene Anwendungsanforderungen
- Integrierte Qualifier-Variablen (QI/QO) für zuverlässige Service-Steuerung
- Spezialisierter WagoBusAdapter für optimierte Buskommunikation

Zustandsübersicht

Der Baustein durchläuft folgende Zustände:

1. **Inaktiv**: Vor der Initialisierung
2. **Initialisierung**: Bei INIT-Ereignis mit QI=TRUE
3. **Aktiv**: Nach erfolgreicher Initialisierung (QO=TRUE)
4. **Fehler**: Bei Kommunikationsproblemen (STATUS mit Fehlermeldung)

Anwendungsszenarien

- Steuerung von Wago-750-Serie IO-Modulen
- Integration in Automatisierungssysteme mit Wago-Controller
- Überwachung und Diagnose von Wago-Busnetzwerken
- Industrie 4.0-Anwendungen mit verteilten Wago-IO-Systemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu generischen Bus-Master-Bausteinen bietet der WagoMaster spezifische Optimierungen für Wago-Bussysteme und eine integrierte Statusdiagnose über WSTRING. Die spezialisierte Adapter-Schnittstelle ermöglicht eine effizientere Kommunikation als universelle Bus-Adapter.

Fazit

Der WagoMaster stellt einen zuverlässigen und spezialisierten Baustein für die Integration von Wago-Bussystemen in 4diac-basierte Steuerungssysteme dar. Durch seine konfigurierbaren Parameter und detaillierte Statusrückmeldung eignet er sich besonders für anspruchsvolle Industrieautomationsanwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

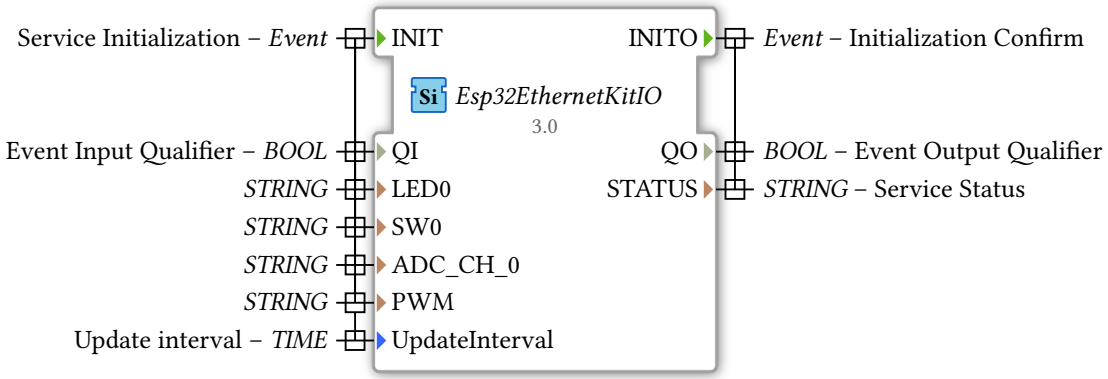
2.4.2.23 zephyr

Esp32EthernetKitIO

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock Esp32EthernetKitIO dient als Template für modulare Ein-/Ausgabe-Operationen mit dem Esp32EthernetKit-Board. Dieser Baustein ermöglicht die Steuerung und Überwachung verschiedener Hardware-Komponenten des Boards über eine einheitliche Schnittstelle.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung - löst den Initialisierungsprozess des Bausteins aus

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung - signalisiert den erfolgreichen Abschluss der Initialisierung

Daten-Eingänge

- **QI** (BOOL): Event Input Qualifier - steuert die Ausführung des Bausteins
- **LED0** (STRING): Konfiguration für LED0
- **SW0** (STRING): Konfiguration für Schalter SW0
- **ADC_CH_0** (STRING): Konfiguration für Analog-Digital-Wandler Kanal 0
- **PWM** (STRING): Konfiguration für Pulsweitenmodulation
- **UpdateInterval** (TIME): Aktualisierungsintervall mit Standardwert T#40ms

Daten-Ausgänge

- **QO** (BOOL): Event Output Qualifier - Status der Bausteinausführung
- **STATUS** (STRING): Service-Status - liefert detaillierte Statusinformationen

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Esp32EthernetKitIO-Baustein initialisiert und verwaltet die Hardware-Komponenten des Esp32EthernetKit-Boards. Bei Empfang des INIT-Ereignisses werden die konfigurierten Ein-/Ausgabekomponenten (LEDs, Schalter, ADC, PWM) entsprechend den übergebenen String-Parametern eingerichtet. Das Aktualisierungsintervall bestimmt, wie häufig die Ein-/Ausgabezustände aktualisiert werden.

Technische Besonderheiten

- Verwendet String-Parameter zur flexiblen Konfiguration der Hardware-Komponenten
- Unterstützt gemischte Signaltypen (digital, analog, PWM)
- Bietet ein konfigurierbares Aktualisierungsintervall für Echtzeit-Anwendungen
- Implementiert im Zephyr-RTOS-Framework

Zustandsübergänge

1. **Initialisierungszustand**: Wartet auf INIT-Ereignis
2. **Aktivierungszustand**: Verarbeitet INIT mit Konfigurationsparametern
3. **Betriebszustand**: Führt kontinuierliche Ein-/Ausgabeoperationen durch
4. **Statusmeldung**: Sendet INITO mit Statusinformationen

Anwendungsszenarien

- Steuerung von Beleuchtungssystemen mit LEDs
- Erfassung von Benutzereingaben über Taster/Schalter
- Messung analoger Sensordaten über ADC
- Regelung von Motoren oder Aktoren mittels PWM
- IoT-Geräte mit Esp32EthernetKit-Board

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen Ein-/Ausgabeblöcken bietet Esp32EthernetKitIO eine integrierte Lösung für das spezifische Hardware-Board mit Unterstützung für verschiedene Signaltypen und konfigurierbaren Aktualisierungsintervallen.

Fazit

Der Esp32EthernetKitIO-Funktionsblock stellt eine flexible und leistungsfähige Lösung für die Ansteuerung von Esp32EthernetKit-Boards in 4diac-basierten Steuerungssystemen dar. Durch die modulare Konfiguration über String-Parameter ermöglicht er eine vielseitige Nutzung in verschiedenen industriellen und IoT-Anwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

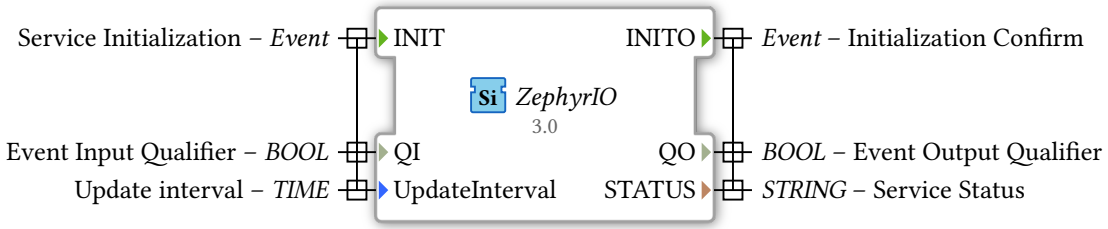
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 Das PWM-Signal & Infografik auf ms-muc-docs.de

ZephyrIO

• – ◦ • – ◦ • – ◦ •

Einleitung

Der ZephyrIO Funktionsblock dient als Template für modulare Ein-/Ausgabe-Funktionalität mit Boards, die das Zephyr Betriebssystem verwenden. Dieser Baustein ermöglicht die Initialisierung und Konfiguration von IO-Services in verteilten Automatisierungssystemen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierungsereignis
  - Verknüpft mit: QI, UpdateInterval

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung
  - Verknüpft mit: QO, STATUS

Daten-Eingänge

- **QI** (BOOL): Event Input Qualifier - Steuert die Ausführung des Initialisierungsvorgangs
- **UpdateInterval** (TIME): Aktualisierungsintervall für den Service (Standardwert: T#40ms)

Daten-Ausgänge

- **QO** (BOOL): Event Output Qualifier - Zeigt den Status der Initialisierung an
- **STATUS** (STRING): Service-Statusinformationen

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der ZephyrIO Funktionsblock initialisiert IO-Services für Zephyr-basierte Hardware. Bei Empfang des INIT-Ereignisses wird der Service mit dem konfigurierten Update-Intervall gestartet. Nach erfolgreicher Initialisierung wird das INITO-Ereignis mit entsprechenden Statusinformationen ausgegeben.

Technische Besonderheiten

- Speziell für Zephyr RTOS optimiert
- Konfigurierbares Update-Intervall für periodische Operationen
- Standard-Update-Intervall von 40 Millisekunden
- Statusrückmeldung über STRING-Variable

Zustandsübersicht

1. **Nicht initialisiert**: Wartet auf INIT-Ereignis
2. **Initialisierung läuft**: Verarbeitet INIT mit QI=true
3. **Initialisiert**: Sendet INITO mit Statusinformationen

Anwendungsszenarien

- Einbindung von Zephyr-basierten IO-Boards in 4diac-Systeme
- Modulare IO-Erweiterungen für Embedded-Systeme
- Hardware-nahe Echtzeit-Steuerungen mit Zephyr RTOS

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu generischen IO-Bausteinen bietet ZephyrIO spezifische Optimierungen für das Zephyr Betriebssystem und ermöglicht eine nahtlose Integration von Zephyr-basierter Hardware in 4diac-Umgebungen.

Fazit

Der ZephyrIO Funktionsblock stellt eine spezialisierte Lösung für die Integration von Zephyr-basierten IO-Komponenten in IEC 61499-Systeme dar und bietet eine standardisierte Schnittstelle für modulare Hardware-Erweiterungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.3 net

2.4.3.1 🎧 Podcast

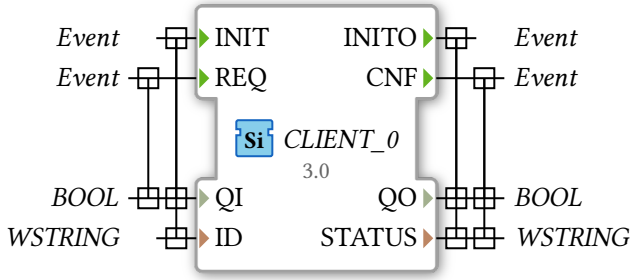
- Industrielle Netzwerke: Das Nervensystem der modernen Fabrik – OT, IT & die Zukunft der Automatisierung
- ISOBUS: Die Geheimsprache der Landmaschinen – Von Plug-and-Play zu High-Speed-Vernetzung
- Magnetventile schützen: Dein Leitfaden gegen Spannungsspitzen in Land- und Baumaschinen
- Open Source in der Industrie: Mehr als Code – Ein Netzwerk für Innovation und Kollaboration
- Smart Power: Wie dezentrale Steuerung die Energiekosten in Industrieanlagen senkt und das Netz stabilisiert

2.4.3.2 CLIENT_0

• – ◦ • – ◦ • – ◦ •

Einleitung

Der CLIENT_0 Funktionsblock dient zur Kommunikation mit einem SERVER_0 Block über eine Netzwerkverbindung. Er ermöglicht das Aufbauen und Schließen von Verbindungen. Im Gegensatz zu CLIENT_1 findet kein expliziter Datenaustausch über Daten-Ein-/Ausgänge statt, die Synchronisation erfolgt primär über Ereignisse.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE)
- **REQ**: Fordert eine Interaktion mit dem Server an

Ereignis-Ausgänge

- **INITO**: Bestätigt das Herstellen einer neuen Verbindung (QI = TRUE) oder das Schließen einer Verbindung (QI = FALSE)
- **CNF**: Bestätigt die Ausführung der Anforderung

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (TRUE = Verbindung öffnen, FALSE = Verbindung schließen)
- **ID** (WSTRING): Identifikator für die Verbindung

Daten-Ausgänge

- **QO** (BOOL): Status der Verbindung (TRUE = verbunden, FALSE = getrennt)
- **STATUS** (WSTRING): Statusinformationen über die Verbindung

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der CLIENT_0 Block initialisiert eine Netzwerkverbindung zum SERVER_0 Block über das INIT-Ereignis. Bei erfolgreicher Verbindung wird das INITO-Ereignis mit QO=TRUE ausgelöst. Über das REQ-Ereignis kann eine Interaktion mit dem Server angestoßen werden, deren Abschluss durch CNF bestätigt wird.

Technische Besonderheiten

- Keine Daten-Ein-/Ausgänge (SD_x, RD_x) vorhanden
- WSTRING-Datentyp für ID und STATUS
- Generische Implementierung durch GEN_CLIENT Klassennamen

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, QO = FALSE
2. **Verbindungsaufbau**: INIT mit QI = TRUE verarbeitet
3. **Verbunden**: INITO mit QO = TRUE bestätigt
4. **Interaktion**: REQ/CNF-Zyklus
5. **Verbindungstrennung**: INIT mit QI = FALSE verarbeitet

Anwendungsszenarien

- Reine Ereignis-Synchronisation zwischen Systemen
- „Heartbeat“-Überwachung ohne Nutzdaten
- Trigger-Signale über Netzwerk

Metadaten

Attribut	Wert
Copyright	(c) 2025 Monika Wenger
Lizenz	EPL-2.0
Version	3.0 (2025-12-05, Monika Wenger)
4diac-Paket	iec61499::net

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

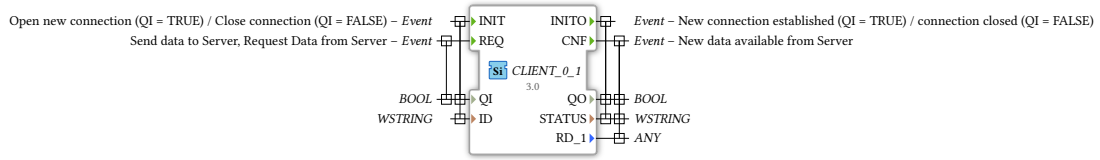
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.3.3 CLIENT_0_1

• – ◦ • – ◦ • – ◦ •

Einleitung

Der CLIENT_0_1 Funktionsblock dient zur Kommunikation mit einem entsprechenden Server-Block (z.B. SERVER_1_0) über eine Netzwerkverbindung. Er ist spezialisiert auf den **Empfang** von Daten vom Server, ohne selbst Daten zu senden (Receive Only Client).



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE)
- **REQ**: Fordert Daten vom Server an

Ereignis-Ausgänge

- **INITO**: Bestätigt das Herstellen einer neuen Verbindung (QI = TRUE) oder das Schließen einer Verbindung (QI = FALSE)
- **CNF**: Signalisiert, dass neue Daten vom Server empfangen wurden und an RD_1 bereitstehen

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (TRUE = Verbindung öffnen, FALSE = Verbindung schließen)
- **ID** (WSTRING): Identifikator für die Verbindung

Daten-Ausgänge

- **QO** (BOOL): Status der Verbindung (TRUE = verbunden, FALSE = getrennt)
- **STATUS** (WSTRING): Statusinformationen über die Verbindung
- **RD_1** (ANY): Vom Server empfangene Daten

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der CLIENT_0_1 Block baut über INIT eine Verbindung auf. Mit dem REQ-Ereignis fordert er Daten vom Server an. Wenn der Server antwortet, werden die Daten am Ausgang RD_1 bereitgestellt und das CNF-Ereignis ausgelöst. Es gibt keinen Dateneingang zum Senden von Nutzdaten.

Technische Besonderheiten

- Unidirektionale Datenübertragung (Empfang)
- Verwendung des ANY-Datentyps für RD_1 ermöglicht flexible Datentypen
- WSTRING-Datentyp für ID und STATUS

Anwendungsszenarien

- Abfragen von Sensorwerten oder Statusinformationen von einem entfernten System
- Überwachungssysteme (Monitoring), die nur Daten lesen

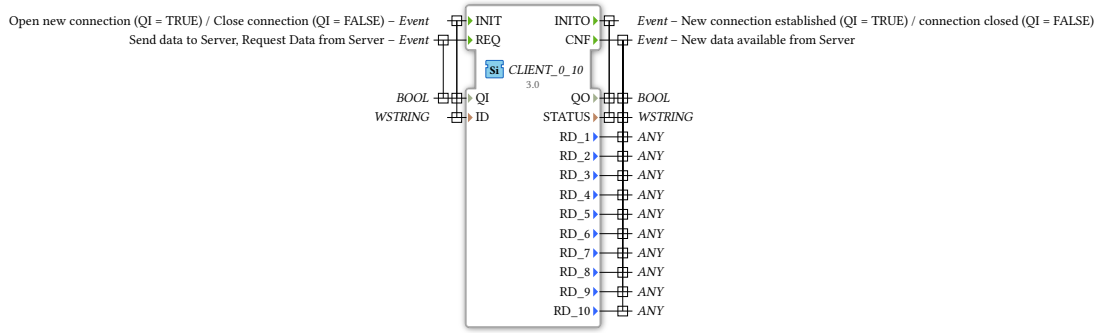
Metadaten

Attribut	Wert
Copyright	(c) 2017, 2025 fortiss GmbH, JKU Linz
Lizenz	EPL-2.0
Version	3.0 (2025-19-26, Alois Zoitl)
4diac-Paket	iec61499::net

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.3.4 CLIENT_0_10



• – ○ • – ○ • – ○ •

Einleitung

Der CLIENT_0_10-Funktionsblock ist die generische Client-Variante mit 0 Sende- und 10 Empfangsdatenfeldern für die Kommunikation mit einem passenden SERVER_10_0-Block. Er empfängt 10 Datenwerte (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9 RD_10) vom Server, ohne selbst Nutzdaten zu senden. Wie alle CLIENT_*-Bausteine basiert er auf der generischen GEN_CLIENT-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE), trägt QI und ID.
- **REQ**: Sendet Daten an den Server und fordert Daten vom Server an, trägt QI.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt QO und STATUS.
- **CNF**: Signalisiert, dass neue Daten vom Server verfügbar sind, trägt QO, STATUS sowie RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9 RD_10.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (TRUE = Verbindung öffnen, FALSE = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- Keine Sendedaten (CLIENT_0_10 überträgt keine Nutzdaten mit REQ, nur die Verbindungssteuerung).

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit CNF vom Server geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit CNF vom Server geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit CNF vom Server geliefert.
- **RD_4** (ANY): Empfangsdatum 4, wird mit CNF vom Server geliefert.
- **RD_5** (ANY): Empfangsdatum 5, wird mit CNF vom Server geliefert.
- **RD_6** (ANY): Empfangsdatum 6, wird mit CNF vom Server geliefert.
- **RD_7** (ANY): Empfangsdatum 7, wird mit CNF vom Server geliefert.
- **RD_8** (ANY): Empfangsdatum 8, wird mit CNF vom Server geliefert.
- **RD_9** (ANY): Empfangsdatum 9, wird mit CNF vom Server geliefert.
- **RD_10** (ANY): Empfangsdatum 10, wird mit CNF vom Server geliefert.

Funktionsweise

CLIENT_0_10 initialisiert über INIT eine Verbindung zum passenden SERVER_10_0-Block (bei QI = TRUE) bzw. schließt sie (bei QI = FALSE); der Abschluss wird über INITO bestätigt. Über REQ wird eine Anfrage ohne Nutzdaten gesendet. Sobald Antwortdaten (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9 RD_10) vorliegen, löst der Baustein CNF aus und stellt RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9 RD_10 bereit.

Technische Besonderheiten

- **Generische Implementierung**: eclipse4diac::core::GenericClassName = 'GEN_CLIENT', dieselbe C++-Basis wie alle anderen CLIENT_*-Varianten; Anzahl und Typ (ANY) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **ANY-Datenfelder**: Alle SD_i/RD_i sind generisch (ANY) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück SERVER_10_0**: CLIENT_0_10 ist funktional nur mit einem SERVER_10_0-Block auf der Server-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, QO = FALSE.
2. **Verbindungsaufbau**: INIT mit QI = TRUE wird verarbeitet.
3. **Verbunden**: INITO mit QO = TRUE bestätigt die Verbindung.

4. **Datenaustausch**: `REQ`/`CNF`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 0 Werte gesendet und 10 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Client-seitige Anbindung** eines Kommunikationspartners, der bereits als `SERVER_10_0` mit passender Feldanzahl ausgelegt ist.

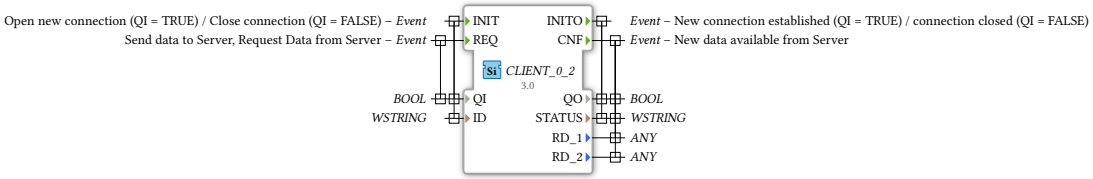
⚖️ **Vergleich mit ähnlichen Bausteinen**

- **SERVER_10_0**: das direkte Gegenstück auf der Server-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`CLIENT_0_10` liefert die generische, auf 0 Sende- und 10 Empfangsfelder zugeschnittene Client-Variante der `GEN_CLIENT`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.5 CLIENT_0_2



• – ○ • – ○ • – ○ •

Einleitung

Der `CLIENT_0_2`-Funktionsblock ist die generische Client-Variante mit 0 Sende- und 2 Empfangsdatenfeldern für die Kommunikation mit einem passenden SERVER_2_0-Block. Er empfängt 2 Datenwerte (`RD_1` `RD_2`) vom Server, ohne selbst Nutzdaten zu senden. Wie alle `CLIENT_*`-Bausteine basiert er auf der generischen `GEN_CLIENT`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE), trägt `QI` und `ID`.
- **REQ**: Sendet Daten an den Server und fordert Daten vom Server an, trägt `QI`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **CNF**: Signalisiert, dass neue Daten vom Server verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1` `RD_2`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- Keine Sendedaten (`CLIENT_0_2` überträgt keine Nutzdaten mit `REQ`, nur die Verbindungssteuerung).

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `CNF` vom Server geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `CNF` vom Server geliefert.

Funktionsweise

CLIENT_0_2 initialisiert über INIT eine Verbindung zum passenden SERVER_2_0-Block (bei QI = TRUE) bzw. schließt sie (bei QI = FALSE); der Abschluss wird über INITO bestätigt. Über REQ wird eine Anfrage ohne Nutzdaten gesendet. Sobald Antwortdaten (RD_1 RD_2) vorliegen, löst der Baustein CNF aus und stellt RD_1 RD_2 bereit.

Technische Besonderheiten

- **Generische Implementierung**: eclipse4diac::core::GenericClassName = 'GEN_CLIENT', dieselbe C++-Basis wie alle anderen CLIENT_*-Varianten; Anzahl und Typ (ANY) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **ANY-Datenfelder**: Alle SD_i/RD_i sind generisch (ANY) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück SERVER_2_0**: CLIENT_0_2 ist funktional nur mit einem SERVER_2_0-Block auf der Server-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, QO = FALSE.
2. **Verbindungsaufbau**: INIT mit QI = TRUE wird verarbeitet.
3. **Verbunden**: INITO mit QO = TRUE bestätigt die Verbindung.
4. **Datenaustausch**: REQ/CNF-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: INIT mit QI = FALSE wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 0 Werte gesendet und 2 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Client-seitige Anbindung** eines Kommunikationspartners, der bereits als SERVER_2_0 mit passender Feldanzahl ausgelegt ist.

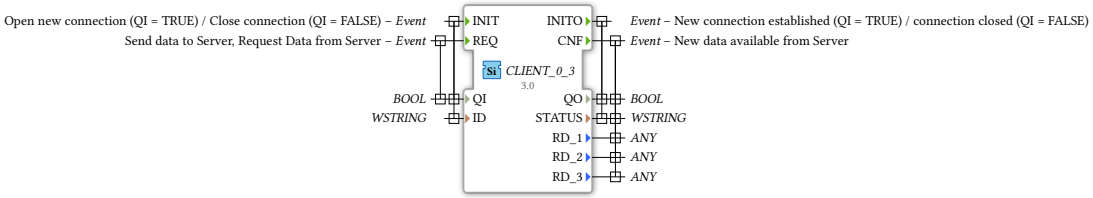
⚖️ Vergleich mit ähnlichen Bausteinen

- SERVER_2_0: das direkte Gegenstück auf der Server-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- CLIENT_1 / SERVER_1: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

CLIENT_0_2 liefert die generische, auf 0 Sende- und 2 Empfangsfelder zugeschnittene Client-Variante der GEN_CLIENT-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.6 CLIENT_0_3



• – ○ • – ○ • – ○ •

Einleitung

Der CLIENT_0_3-Funktionsblock ist die generische Client-Variante mit 0 Sende- und 3 Empfangsdatenfeldern für die Kommunikation mit einem passenden SERVER_3_0-Block. Er empfängt 3 Datenwerte (RD_1 RD_2 RD_3) vom Server, ohne selbst Nutzdaten zu senden. Wie alle CLIENT_*-Bausteine basiert er auf der generischen GEN_CLIENT-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE), trägt `QI` und `ID`.
- **REQ**: Sendet Daten an den Server und fordert Daten vom Server an, trägt `QI`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **CNF**: Signalisiert, dass neue Daten vom Server verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1` `RD_2` `RD_3`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- Keine Sendedaten (`CLIENT_0_3` überträgt keine Nutzdaten mit `REQ`, nur die Verbindungssteuerung).

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `CNF` vom Server geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `CNF` vom Server geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit `CNF` vom Server geliefert.

Funktionsweise

`CLIENT_0_3` initialisiert über `INIT` eine Verbindung zum passenden `SERVER_3_0`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `REQ` wird eine Anfrage ohne Nutzdaten gesendet. Sobald Antwortdaten (`RD_1` `RD_2` `RD_3`) vorliegen, löst der Baustein `CNF` aus und stellt `RD_1` `RD_2` `RD_3` bereit.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, dieselbe C++-Basis wie alle anderen `CLIENT_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `SERVER_3_0`**: `CLIENT_0_3` ist funktional nur mit einem `SERVER_3_0`-Block auf der Server-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `REQ`/`CNF`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 0 Werte gesendet und 3 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Client-seitige Anbindung** eines Kommunikationspartners, der bereits als `SERVER_3_0` mit passender Feldanzahl ausgelegt ist.

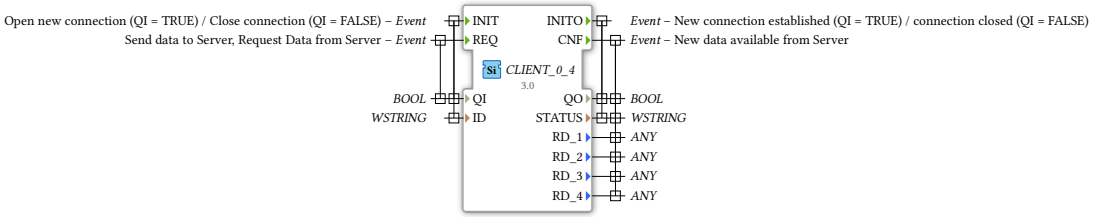
⚖️ Vergleich mit ähnlichen Bausteinen

- **SERVER_3_0**: das direkte Gegenstück auf der Server-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`CLIENT_0_3` liefert die generische, auf 0 Sende- und 3 Empfangsfelder zugeschnittene Client-Variante der `GEN_CLIENT`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.7 CLIENT_0_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `CLIENT_0_4`-Funktionsblock ist die generische Client-Variante mit 0 Sende- und 4 Empfangsdatenfeldern für die Kommunikation mit einem passenden SERVER_4_0-Block. Er empfängt 4 Datenwerte (`RD_1` `RD_2` `RD_3` `RD_4`) vom Server, ohne selbst Nutzdaten zu senden. Wie alle `CLIENT_*`-Bausteine basiert er auf der generischen `GEN_CLIENT`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE), trägt `QI` und `ID`.
- **REQ**: Sendet Daten an den Server und fordert Daten vom Server an, trägt `QI`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **CNF**: Signalisiert, dass neue Daten vom Server verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1` `RD_2` `RD_3` `RD_4`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- Keine Sendedaten (`CLIENT_0_4` überträgt keine Nutzdaten mit `REQ`, nur die Verbindungssteuerung).

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `CNF` vom Server geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `CNF` vom Server geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit `CNF` vom Server geliefert.
- **RD_4** (ANY): Empfangsdatum 4, wird mit `CNF` vom Server geliefert.

Funktionsweise

`CLIENT_0_4` initialisiert über `INIT` eine Verbindung zum passenden `SERVER_4_0`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `REQ` wird eine Anfrage ohne Nutzdaten gesendet. Sobald Antwortdaten (`RD_1` `RD_2` `RD_3` `RD_4`) vorliegen, löst der Baustein `CNF` aus und stellt `RD_1` `RD_2` `RD_3` `RD_4` bereit.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, dieselbe C++-Basis wie alle anderen `CLIENT_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.

- **ANY-Datenfelder**: Alle SD_i/RD_i sind generisch (ANY) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück SERVER_4_0**: CLIENT_0_4 ist funktional nur mit einem SERVER_4_0-Block auf der Server-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, QO = FALSE.
2. **Verbindungsaufbau**: INIT mit QI = TRUE wird verarbeitet.
3. **Verbunden**: INITO mit QO = TRUE bestätigt die Verbindung.
4. **Datenaustausch**: REQ/CNF-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: INIT mit QI = FALSE wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 0 Werte gesendet und 4 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Client-seitige Anbindung** eines Kommunikationspartners, der bereits als SERVER_4_0 mit passender Feldanzahl ausgelegt ist.

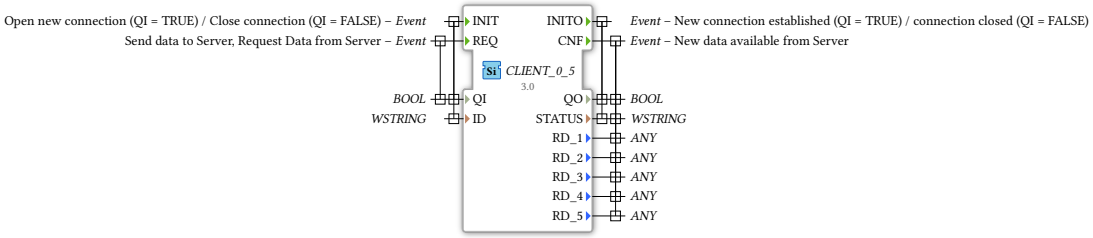
⚖️ Vergleich mit ähnlichen Bausteinen

- SERVER_4_0: das direkte Gegenstück auf der Server-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- CLIENT_1 / SERVER_1: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

CLIENT_0_4 liefert die generische, auf 0 Sende- und 4 Empfangsfelder zugeschnittene Client-Variante der GEN_CLIENT-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.8 CLIENT_0_5



• – ○ • – ○ • – ○ •

Einleitung

Der CLIENT_0_5-Funktionsblock ist die generische Client-Variante mit 0 Sende- und 5 Empfangsdatenfeldern für die Kommunikation mit einem passenden SERVER_5_0-Block. Er empfängt 5 Datenwerte (RD_1 RD_2 RD_3 RD_4 RD_5) vom Server, ohne selbst Nutzdaten zu senden. Wie alle CLIENT_*-Bausteine basiert er auf der generischen GEN_CLIENT-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE), trägt QI und ID.
- **REQ**: Sendet Daten an den Server und fordert Daten vom Server an, trägt QI.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt QO und STATUS.
- **CNF**: Signalisiert, dass neue Daten vom Server verfügbar sind, trägt QO, STATUS sowie RD_1 RD_2 RD_3 RD_4 RD_5.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- Keine Sendedaten (`CLIENT_0_5` überträgt keine Nutzdaten mit `REQ`, nur die Verbindungssteuerung).

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `CNF` vom Server geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `CNF` vom Server geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit `CNF` vom Server geliefert.
- **RD_4** (ANY): Empfangsdatum 4, wird mit `CNF` vom Server geliefert.
- **RD_5** (ANY): Empfangsdatum 5, wird mit `CNF` vom Server geliefert.

Funktionsweise

`CLIENT_0_5` initialisiert über `INIT` eine Verbindung zum passenden `SERVER_5_0`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `REQ` wird eine Anfrage ohne Nutzdaten gesendet. Sobald Antwortdaten (`RD_1 RD_2 RD_3 RD_4 RD_5`) vorliegen, löst der Baustein `CNF` aus und stellt `RD_1 RD_2 RD_3 RD_4 RD_5` bereit.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, dieselbe C++-Basis wie alle anderen `CLIENT_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `SERVER_5_0`**: `CLIENT_0_5` ist funktional nur mit einem `SERVER_5_0`-Block auf der Server-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `REQ`/`CNF`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 0 Werte gesendet und 5 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Client-seitige Anbindung** eines Kommunikationspartners, der bereits als `SERVER_5_0` mit passender Feldanzahl ausgelegt ist.

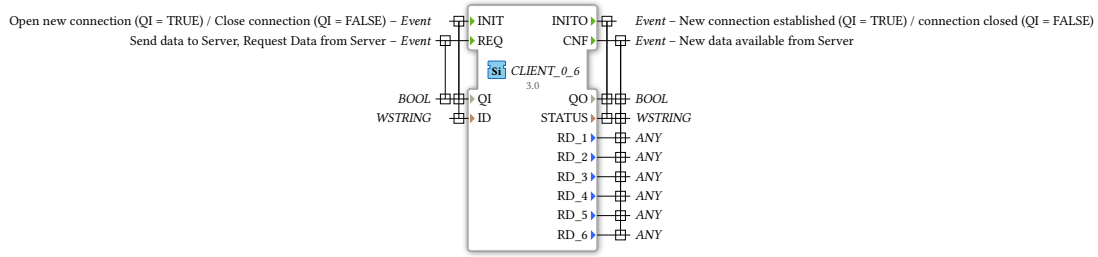
⚖️ Vergleich mit ähnlichen Bausteinen

- **SERVER_5_0**: das direkte Gegenstück auf der Server-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`CLIENT_0_5` liefert die generische, auf 0 Sende- und 5 Empfangsfelder zugeschnittene Client-Variante der `GEN_CLIENT`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.9 CLIENT_0_6



• – ○ • – ○ • – ○ •

Einleitung

Der `CLIENT_0_6`-Funktionsblock ist die generische Client-Variante mit 0 Sende- und 6 Empfangsdatenfeldern für die Kommunikation mit einem passenden SERVER_6_0-Block. Er empfängt 6 Datenwerte (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6`) vom Server, ohne selbst Nutzdaten zu senden. Wie alle `CLIENT_*`-Bausteine basiert er auf der generischen `GEN_CLIENT`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE), trägt `QI` und `ID`.
- **REQ**: Sendet Daten an den Server und fordert Daten vom Server an, trägt `QI`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **CNF**: Signalisiert, dass neue Daten vom Server verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- Keine Sendedaten (`CLIENT_0_6` überträgt keine Nutzdaten mit `REQ`, nur die Verbindungssteuerung).

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `CNF` vom Server geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `CNF` vom Server geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit `CNF` vom Server geliefert.
- **RD_4** (ANY): Empfangsdatum 4, wird mit `CNF` vom Server geliefert.
- **RD_5** (ANY): Empfangsdatum 5, wird mit `CNF` vom Server geliefert.
- **RD_6** (ANY): Empfangsdatum 6, wird mit `CNF` vom Server geliefert.

Funktionsweise

`CLIENT_0_6` initialisiert über `INIT` eine Verbindung zum passenden `SERVER_6_0`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `REQ` wird eine Anfrage ohne Nutzdaten gesendet. Sobald Antwortdaten (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6`) vorliegen, löst der Baustein `CNF` aus und stellt `RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` bereit.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, dieselbe C++-Basis wie alle anderen `CLIENT_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.

- **Gegenstück `SERVER_6_0`**: `CLIENT_0_6` ist funktional nur mit einem `SERVER_6_0`-Block auf der Server-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `REQ`/`CNF`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 0 Werte gesendet und 6 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Client-seitige Anbindung** eines Kommunikationspartners, der bereits als `SERVER_6_0` mit passender Feldanzahl ausgelegt ist.

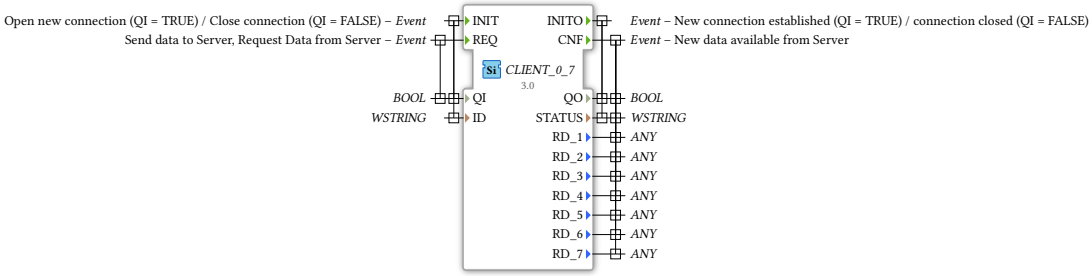
⚖️ Vergleich mit ähnlichen Bausteinen

- **SERVER_6_0**: das direkte Gegenstück auf der Server-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`CLIENT_0_6` liefert die generische, auf 0 Sende- und 6 Empfangsfelder zugeschnittene Client-Variante der `GEN_CLIENT`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.10 CLIENT_0_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `CLIENT_0_7`-Funktionsblock ist die generische Client-Variante mit 0 Sende- und 7 Empfangsdatenfeldern für die Kommunikation mit einem passenden SERVER_7_0-Block. Er empfängt 7 Datenwerte (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7`) vom Server, ohne selbst Nutzdaten zu senden. Wie alle `CLIENT_*`-Bausteine basiert er auf der generischen `GEN_CLIENT`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE), trägt `QI` und `ID`.
- **REQ**: Sendet Daten an den Server und fordert Daten vom Server an, trägt `QI`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **CNF**: Signalisiert, dass neue Daten vom Server verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- Keine Sendedaten (`CLIENT_0_7` überträgt keine Nutzdaten mit `REQ`, nur die Verbindungssteuerung).

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `CNF` vom Server geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `CNF` vom Server geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit `CNF` vom Server geliefert.
- **RD_4** (ANY): Empfangsdatum 4, wird mit `CNF` vom Server geliefert.
- **RD_5** (ANY): Empfangsdatum 5, wird mit `CNF` vom Server geliefert.
- **RD_6** (ANY): Empfangsdatum 6, wird mit `CNF` vom Server geliefert.
- **RD_7** (ANY): Empfangsdatum 7, wird mit `CNF` vom Server geliefert.

Funktionsweise

`CLIENT_0_7` initialisiert über `INIT` eine Verbindung zum passenden `SERVER_7_0`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `REQ` wird eine Anfrage ohne Nutzdaten gesendet. Sobald Antwortdaten (`RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7`) vorliegen, löst der Baustein `CNF` aus und stellt `RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7` bereit.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, dieselbe C++-Basis wie alle anderen `CLIENT_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `SERVER_7_0`**: `CLIENT_0_7` ist funktional nur mit einem `SERVER_7_0`-Block auf der Server-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `REQ`/`CNF`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 0 Werte gesendet und 7 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Client-seitige Anbindung** eines Kommunikationspartners, der bereits als `SERVER_7_0` mit passender Feldanzahl ausgelegt ist.

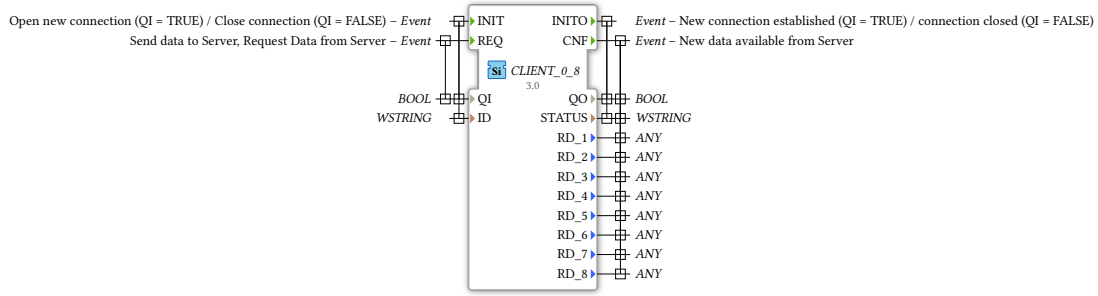
⚖️ Vergleich mit ähnlichen Bausteinen

- **SERVER_7_0**: das direkte Gegenstück auf der Server-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`CLIENT_0_7` liefert die generische, auf 0 Sende- und 7 Empfangsfelder zugeschnittene Client-Variante der `GEN_CLIENT`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.11 CLIENT_0_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `CLIENT_0_8`-Funktionsblock ist die generische Client-Variante mit 0 Sende- und 8 Empfangsdatenfeldern für die Kommunikation mit einem passenden SERVER_8_0-Block. Er empfängt 8 Datenwerte (`RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8`) vom Server, ohne selbst Nutzdaten zu senden. Wie alle `CLIENT_*`-Bausteine basiert er auf der generischen `GEN_CLIENT`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE), trägt `QI` und `ID`.
- **REQ**: Sendet Daten an den Server und fordert Daten vom Server an, trägt `QI`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **CNF**: Signalisiert, dass neue Daten vom Server verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- Keine Sendedaten (`CLIENT_0_8` überträgt keine Nutzdaten mit `REQ`, nur die Verbindungssteuerung).

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `CNF` vom Server geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `CNF` vom Server geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit `CNF` vom Server geliefert.
- **RD_4** (ANY): Empfangsdatum 4, wird mit `CNF` vom Server geliefert.
- **RD_5** (ANY): Empfangsdatum 5, wird mit `CNF` vom Server geliefert.
- **RD_6** (ANY): Empfangsdatum 6, wird mit `CNF` vom Server geliefert.
- **RD_7** (ANY): Empfangsdatum 7, wird mit `CNF` vom Server geliefert.
- **RD_8** (ANY): Empfangsdatum 8, wird mit `CNF` vom Server geliefert.

Funktionsweise

`CLIENT_0_8` initialisiert über `INIT` eine Verbindung zum passenden `SERVER_8_0`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `REQ` wird eine Anfrage ohne Nutzdaten gesendet. Sobald Antwortdaten (`RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8`) vorliegen, löst der Baustein `CNF` aus und stellt `RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8` bereit.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, dieselbe C++-Basis wie alle anderen `CLIENT_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `SERVER_8_0`**: `CLIENT_0_8` ist funktional nur mit einem `SERVER_8_0`-Block auf der Server-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `REQ`/`CNF`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 0 Werte gesendet und 8 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Client-seitige Anbindung** eines Kommunikationspartners, der bereits als `SERVER_8_0` mit passender Feldanzahl ausgelegt ist.

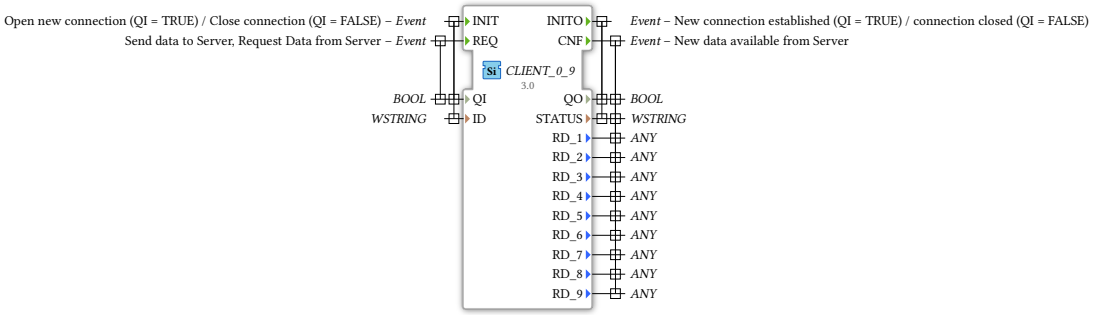
⚖️ Vergleich mit ähnlichen Bausteinen

- SERVER_8_0: das direkte Gegenstück auf der Server-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- CLIENT_1 / SERVER_1: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`CLIENT_0_8` liefert die generische, auf 0 Sende- und 8 Empfangsfelder zugeschnittene Client-Variante der `GEN_CLIENT`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.12 CLIENT_0_9



• – ○ • – ○ • – ○ •

Einleitung

Der `CLIENT_0_9`-Funktionsblock ist die generische Client-Variante mit 0 Sende- und 9 Empfangsdatenfeldern für die Kommunikation mit einem passenden SERVER_9_0-Block. Er empfängt 9 Datenwerte (`RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9`) vom Server, ohne selbst Nutzdaten zu senden. Wie alle `CLIENT_*`-Bausteine basiert er auf der generischen `GEN_CLIENT`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE), trägt `QI` und `ID`.
- **REQ**: Sendet Daten an den Server und fordert Daten vom Server an, trägt `QI`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **CNF**: Signalisiert, dass neue Daten vom Server verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- Keine Sendedaten (`CLIENT_0_9` überträgt keine Nutzdaten mit `REQ`, nur die Verbindungssteuerung).

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `CNF` vom Server geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `CNF` vom Server geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit `CNF` vom Server geliefert.
- **RD_4** (ANY): Empfangsdatum 4, wird mit `CNF` vom Server geliefert.
- **RD_5** (ANY): Empfangsdatum 5, wird mit `CNF` vom Server geliefert.
- **RD_6** (ANY): Empfangsdatum 6, wird mit `CNF` vom Server geliefert.
- **RD_7** (ANY): Empfangsdatum 7, wird mit `CNF` vom Server geliefert.
- **RD_8** (ANY): Empfangsdatum 8, wird mit `CNF` vom Server geliefert.
- **RD_9** (ANY): Empfangsdatum 9, wird mit `CNF` vom Server geliefert.

Funktionsweise

`CLIENT_0_9` initialisiert über `INIT` eine Verbindung zum passenden `SERVER_9_0`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `REQ` wird eine Anfrage ohne Nutzdaten gesendet. Sobald Antwortdaten (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9`) vorliegen, löst der Baustein `CNF` aus und stellt `RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9` bereit.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, dieselbe C++-Basis wie alle anderen `CLIENT_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `SERVER_9_0`**: `CLIENT_0_9` ist funktional nur mit einem `SERVER_9_0`-Block auf der Server-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `REQ`/`CNF`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 0 Werte gesendet und 9 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Client-seitige Anbindung** eines Kommunikationspartners, der bereits als `SERVER_9_0` mit passender Feldanzahl ausgelegt ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- **SERVER_9_0**: das direkte Gegenstück auf der Server-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

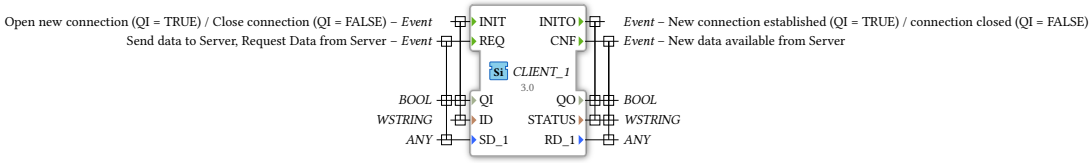
`CLIENT_0_9` liefert die generische, auf 0 Sende- und 9 Empfangsfelder zugeschnittene Client-Variante der `GEN_CLIENT`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.13 CLIENT_1

• – ○ • – ○ • – ○ •

Einleitung

Der CLIENT_1 Funktionsblock dient zur Kommunikation mit einem SERVER_1 Block über eine Netzwerkverbindung. Er ermöglicht das Aufbauen und Schließen von Verbindungen sowie den Datenaustausch zwischen Client und Server.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE)
- **REQ**: Sendet Daten an den Server und fordert Daten vom Server an

Ereignis-Ausgänge

- **INITO**: Bestätigt das Herstellen einer neuen Verbindung (QI = TRUE) oder das Schließen einer Verbindung (QI = FALSE)
- **CNF**: Signalisiert, dass neue Daten vom Server verfügbar sind

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (TRUE = Verbindung öffnen, FALSE = Verbindung schließen)
- **ID** (WSTRING): Identifikator für die Verbindung
- **SD_1** (ANY): Zu sendende Daten an den Server

Daten-Ausgänge

- **QO** (BOOL): Status der Verbindung (TRUE = verbunden, FALSE = getrennt)
- **STATUS** (WSTRING): Statusinformationen über die Verbindung
- **RD_1** (ANY): Vom Server empfangene Daten

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der CLIENT_1 Block initialisiert eine Netzwerkverbindung zum SERVER_1 Block über das INIT-Ereignis. Bei erfolgreicher Verbindung wird das INITO-Ereignis mit QO=TRUE ausgelöst. Über das REQ-Ereignis können Daten an den Server gesendet und gleichzeitig Daten vom Server angefordert werden. Empfangene Daten werden über das CNF-Ereignis mit den entsprechenden Datenausgängen bereitgestellt.

Technische Besonderheiten

- Verwendung des ANY-Datentyps für SD_1 und RD_1 ermöglicht flexible Datentypen
- WSTRING-Datentyp für ID und STATUS unterstützt Unicode-Zeichen
- Generische Implementierung durch GEN_CLIENT Klassennamen

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, QO = FALSE
2. **Verbindungsaufbau**: INIT mit QI = TRUE verarbeitet
3. **Verbunden**: INITO mit QO = TRUE bestätigt
4. **Datenübertragung**: REQ/CNF-Zyklus für Datenaustausch
5. **Verbindungstrennung**: INIT mit QI = FALSE verarbeitet

Anwendungsszenarien

- Kommunikation zwischen verteilten Steuerungssystemen
- Datenaustausch in Industrie-4.0-Anwendungen
- Client-Server-Architekturen in Automatisierungssystemen
- Vernetzte Steuerungskomponenten über TCP/IP oder andere Protokolle

⚖️ Vergleich mit ähnlichen Bausteinen

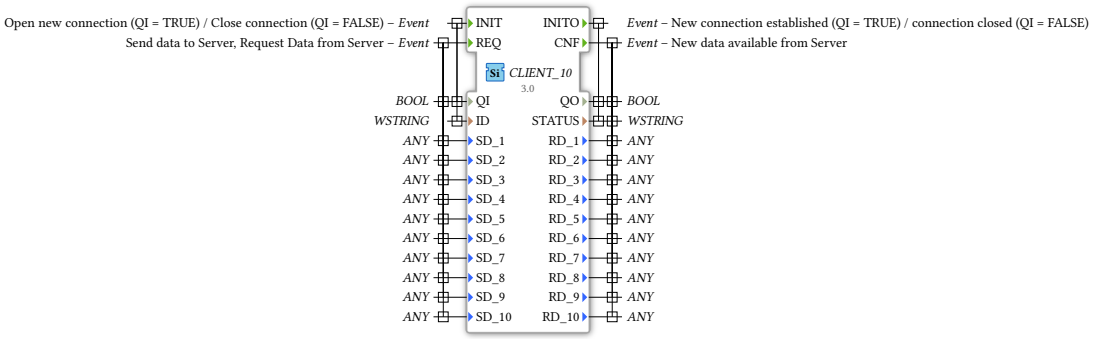
Im Vergleich zu einfacheren Kommunikationsblöcken bietet CLIENT_1:

- Bidirektionalen Datenaustausch in einer Transaktion
- Flexible Datentypen durch ANY-Typ
- Explizite Verbindungssteuerung
- Detaillierte Statusrückmeldungen

Fazit

Der CLIENT_1 Funktionsblock stellt eine robuste Lösung für Client-Server-Kommunikation in verteilten Automatisierungssystemen dar. Seine flexible Schnittstelle und explizite Verbindungssteuerung machen ihn besonders geeignet für anspruchsvolle Netzwerkanwendungen in industriellen Umgebungen.

2.4.3.14 CLIENT_10



• – ○ • – ○ • – ○ •

Einleitung

Der `CLIENT_10`-Funktionsblock ist die generische Client-Variante mit 10 Sende- und 10 Empfangsdatenfeldern für die Kommunikation mit einem passenden SERVER_10-Block. Er überträgt 10 Datenwerte (`SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7` `SD_8` `SD_9` `SD_10`) an den Server und empfängt 10 Datenwerte (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9` `RD_10`) zurück. Wie alle `CLIENT_*`-Bausteine basiert er auf der generischen `GEN_CLIENT`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE), trägt `QI` und `ID`.
- **REQ**: Sendet Daten an den Server und fordert Daten vom Server an, trägt `QI` sowie `SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7` `SD_8` `SD_9` `SD_10`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.

- **CNF**: Signalisiert, dass neue Daten vom Server verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9` `RD_10`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit `REQ` an den Server übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `REQ` an den Server übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit `REQ` an den Server übertragen.
- **SD_4** (ANY): Sendedatum 4, wird mit `REQ` an den Server übertragen.
- **SD_5** (ANY): Sendedatum 5, wird mit `REQ` an den Server übertragen.
- **SD_6** (ANY): Sendedatum 6, wird mit `REQ` an den Server übertragen.
- **SD_7** (ANY): Sendedatum 7, wird mit `REQ` an den Server übertragen.
- **SD_8** (ANY): Sendedatum 8, wird mit `REQ` an den Server übertragen.
- **SD_9** (ANY): Sendedatum 9, wird mit `REQ` an den Server übertragen.
- **SD_10** (ANY): Sendedatum 10, wird mit `REQ` an den Server übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `CNF` vom Server geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `CNF` vom Server geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit `CNF` vom Server geliefert.
- **RD_4** (ANY): Empfangsdatum 4, wird mit `CNF` vom Server geliefert.
- **RD_5** (ANY): Empfangsdatum 5, wird mit `CNF` vom Server geliefert.
- **RD_6** (ANY): Empfangsdatum 6, wird mit `CNF` vom Server geliefert.
- **RD_7** (ANY): Empfangsdatum 7, wird mit `CNF` vom Server geliefert.
- **RD_8** (ANY): Empfangsdatum 8, wird mit `CNF` vom Server geliefert.
- **RD_9** (ANY): Empfangsdatum 9, wird mit `CNF` vom Server geliefert.
- **RD_10** (ANY): Empfangsdatum 10, wird mit `CNF` vom Server geliefert.

Funktionsweise

`CLIENT_10` initialisiert über `INIT` eine Verbindung zum passenden `SERVER_10`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `REQ` werden `SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7` `SD_8` `SD_9` `SD_10` an den Server übertragen. Sobald Antwortdaten (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9` `RD_10`) vorliegen, löst der Baustein `CNF` aus und stellt `RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9` `RD_10` bereit.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, dieselbe C++-Basis wie alle anderen `CLIENT_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `SERVER_10`**: `CLIENT_10` ist funktional nur mit einem `SERVER_10`-Block auf der Server-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `REQ`/`CNF`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 10 Werte gesendet und 10 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.

- **Client-seitige Anbindung** eines Kommunikationspartners, der bereits als `SERVER_10` mit passender Feldanzahl ausgelegt ist.

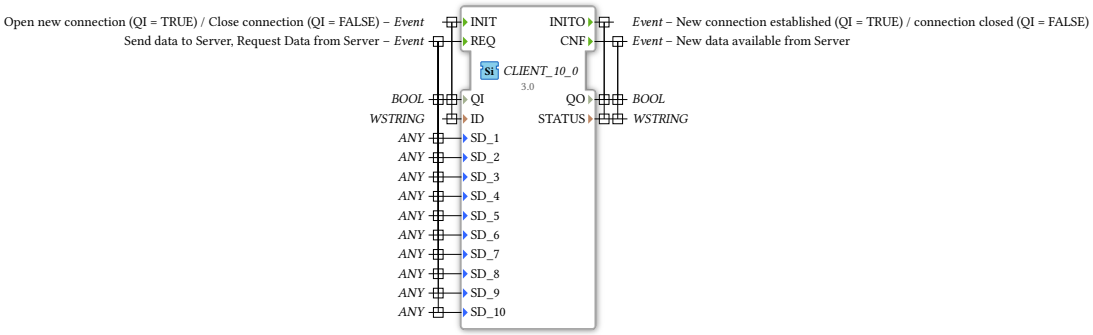
⚖️ Vergleich mit ähnlichen Bausteinen

- **SERVER_10**: das direkte Gegenstück auf der Server-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`CLIENT_10` liefert die generische, auf 10 Sende- und 10 Empfangsfelder zugeschnittene Client-Variante der `GEN_CLIENT`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.15 CLIENT_10_0



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `CLIENT_10_0`-Funktionsblock ist die generische Client-Variante mit 10 Sende- und 0 Empfangsdatenfeldern für die Kommunikation mit einem passenden SERVER_0_10-Block. Er überträgt 10 Datenwerte (`SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7` `SD_8` `SD_9` `SD_10`) an den Server, ohne Rückgabedaten zu empfangen. Wie alle `CLIENT_*`-Bausteine basiert er auf der generischen `GEN_CLIENT`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE), trägt `QI` und `ID`.
- **REQ**: Sendet Daten an den Server und fordert Daten vom Server an, trägt `QI` sowie `SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7` `SD_8` `SD_9` `SD_10`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **CNF**: Signalisiert, dass neue Daten vom Server verfügbar sind, trägt `QO`, `STATUS`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit `REQ` an den Server übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `REQ` an den Server übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit `REQ` an den Server übertragen.
- **SD_4** (ANY): Sendedatum 4, wird mit `REQ` an den Server übertragen.
- **SD_5** (ANY): Sendedatum 5, wird mit `REQ` an den Server übertragen.
- **SD_6** (ANY): Sendedatum 6, wird mit `REQ` an den Server übertragen.
- **SD_7** (ANY): Sendedatum 7, wird mit `REQ` an den Server übertragen.
- **SD_8** (ANY): Sendedatum 8, wird mit `REQ` an den Server übertragen.

- **SD_9** (ANY): Sendedatum 9, wird mit REQ an den Server übertragen.
- **SD_10** (ANY): Sendedatum 10, wird mit REQ an den Server übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- Keine Empfangsdaten (CLIENT_10_0 erhält keine Nutzdaten über CNF, nur Verbindungs-/Statusinformationen).

Funktionsweise

CLIENT_10_0 initialisiert über INIT eine Verbindung zum passenden SERVER_0_10-Block (bei QI = TRUE) bzw. schließt sie (bei QI = FALSE); der Abschluss wird über INITO bestätigt. Über REQ werden SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8 SD_9 SD_10 an den Server übertragen. Sobald eine Antwort vorliegen, löst der Baustein CNF aus.

Technische Besonderheiten

- **Generische Implementierung**: eclipse4diac::core::GenericClassName = 'GEN_CLIENT', dieselbe C++-Basis wie alle anderen CLIENT_*-Varianten; Anzahl und Typ (ANY) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **ANY-Datenfelder**: Alle SD_i/RD_i sind generisch (ANY) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück SERVER_0_10**: CLIENT_10_0 ist funktional nur mit einem SERVER_0_10-Block auf der Server-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, QO = FALSE.
2. **Verbindungsaufbau**: INIT mit QI = TRUE wird verarbeitet.
3. **Verbunden**: INITO mit QO = TRUE bestätigt die Verbindung.
4. **Datenaustausch**: REQ/CNF-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: INIT mit QI = FALSE wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 10 Werte gesendet und 0 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Client-seitige Anbindung** eines Kommunikationspartners, der bereits als SERVER_0_10 mit passender Feldanzahl ausgelegt ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- SERVER_0_10: das direkte Gegenstück auf der Server-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- CLIENT_1 / SERVER_1: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

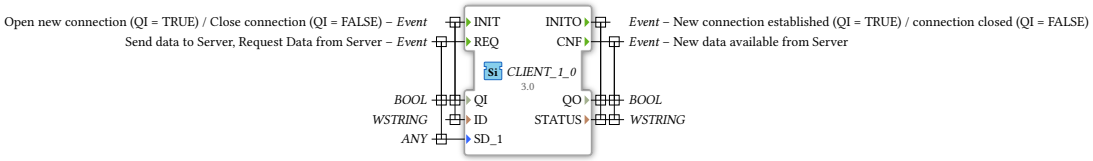
CLIENT_10_0 liefert die generische, auf 10 Sende- und 0 Empfangsfelder zugeschnittene Client-Variante der GEN_CLIENT-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.16 CLIENT_1_0

• – ◦ • – ◦ • – ◦ •

Einleitung

Der CLIENT_1_0 Funktionsblock dient zur Kommunikation mit einem entsprechenden Server-Block (z.B. SERVER_0_1) über eine Netzwerkverbindung. Er ist spezialisiert auf das **Senden** von Daten an den Server, ohne Daten zu empfangen (Send Only Client).



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE)
- **REQ**: Sendet die an SD_1 anliegenden Daten an den Server

Ereignis-Ausgänge

- **INITO**: Bestätigt das Herstellen einer neuen Verbindung (QI = TRUE) oder das Schließen einer Verbindung (QI = FALSE)
- **CNF**: Bestätigt, dass die Daten erfolgreich gesendet wurden

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (TRUE = Verbindung öffnen, FALSE = Verbindung schließen)
- **ID** (WSTRING): Identifikator für die Verbindung
- **SD_1** (ANY): Zu sendende Daten an den Server

Daten-Ausgänge

- **QO** (BOOL): Status der Verbindung (TRUE = verbunden, FALSE = getrennt)
- **STATUS** (WSTRING): Statusinformationen über die Verbindung

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der CLIENT_1_0 Block baut über INIT eine Verbindung auf. Wenn das REQ-Ereignis ausgelöst wird, sendet der Block den Wert von SD_1 an den Server. Der erfolgreiche Versand wird durch das CNF-Ereignis bestätigt. Es gibt keinen Datenausgang für Empfangsdaten.

Technische Besonderheiten

- Unidirektionale Datenübertragung (Senden)
- Verwendung des ANY-Datentyps für SD_1 ermöglicht flexible Datentypen
- WSTRING-Datentyp für ID und STATUS

Anwendungsszenarien

- Senden von Steuerbefehlen oder Sollwerten an ein entferntes System (ohne direkte Rückantwort als Datum)
- Logging-Clients, die Daten an einen zentralen Server pushen

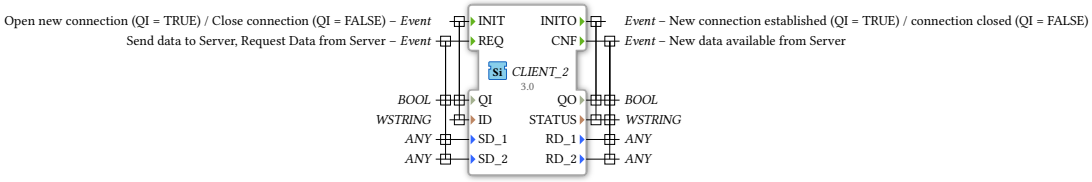
Metadaten

Attribut	Wert
Copyright	(c) 2017, 2025 fortiss GmbH, JKU Linz
Lizenz	EPL-2.0
Version	3.0 (2025-19-26, Alois Zoitl)
4diac-Paket	iec61499::net

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.3.17 CLIENT_2



• – ○ • – ○ • – ○ •

Einleitung

Der `CLIENT_2`-Funktionsblock ist die generische Client-Variante mit 2 Sende- und 2 Empfangsdatenfeldern für die Kommunikation mit einem passenden SERVER_2-Block. Er überträgt 2 Datenwerte (`SD_1` `SD_2`) an den Server und empfängt 2 Datenwerte (`RD_1` `RD_2`) zurück. Wie alle `CLIENT_*`-Bausteine basiert er auf der generischen `GEN_CLIENT`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE), trägt `QI` und `ID`.
- **REQ**: Sendet Daten an den Server und fordert Daten vom Server an, trägt `QI` sowie `SD_1` `SD_2`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **CNF**: Signalisiert, dass neue Daten vom Server verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1` `RD_2`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit `REQ` an den Server übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `REQ` an den Server übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `CNF` vom Server geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `CNF` vom Server geliefert.

Funktionsweise

`CLIENT_2` initialisiert über `INIT` eine Verbindung zum passenden `SERVER_2`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `REQ` werden `SD_1` `SD_2` an den Server übertragen. Sobald Antwortdaten (`RD_1` `RD_2`) vorliegen, löst der Baustein `CNF` aus und stellt `RD_1` `RD_2` bereit.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, dieselbe C++-Basis wie alle anderen `CLIENT_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `SERVER_2`**: `CLIENT_2` ist funktional nur mit einem `SERVER_2`-Block auf der Server-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.

3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `REQ`/`CNF`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 2 Werte gesendet und 2 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Client-seitige Anbindung** eines Kommunikationspartners, der bereits als `SERVER_2` mit passender Feldanzahl ausgelegt ist.

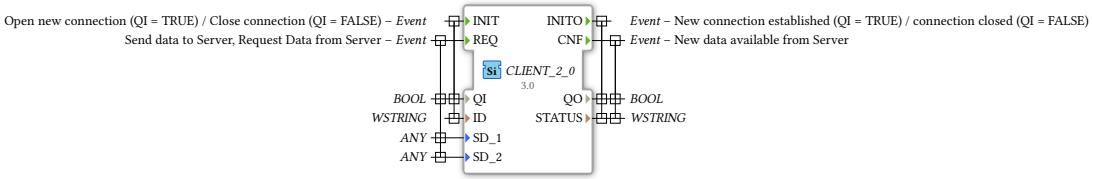
⚖️ Vergleich mit ähnlichen Bausteinen

- **SERVER_2**: das direkte Gegenstück auf der Server-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`CLIENT_2` liefert die generische, auf 2 Sende- und 2 Empfangsfelder zugeschnittene Client-Variante der `GEN_CLIENT`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.18 CLIENT_2_0



• – ○ • – ○ • – ○ •

Einleitung

Der `CLIENT_2_0`-Funktionsblock ist die generische Client-Variante mit 2 Sende- und 0 Empfangsdatenfeldern für die Kommunikation mit einem passenden SERVER_0_2-Block. Er überträgt 2 Datenwerte (`SD_1` `SD_2`) an den Server, ohne Rückgabedaten zu empfangen. Wie alle `CLIENT_*`-Bausteine basiert er auf der generischen `GEN_CLIENT`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE), trägt `QI` und `ID`.
- **REQ**: Sendet Daten an den Server und fordert Daten vom Server an, trägt `QI` sowie `SD_1` `SD_2`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **CNF**: Signalisiert, dass neue Daten vom Server verfügbar sind, trägt `QO`, `STATUS`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit `REQ` an den Server übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `REQ` an den Server übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- Keine Empfangsdaten (`CLIENT_2_0` erhält keine Nutzdaten über `CNF`, nur Verbindungs-/Statusinformationen).

Funktionsweise

`CLIENT_2_0` initialisiert über `INIT` eine Verbindung zum passenden `SERVER_0_2`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `REQ` werden `SD_1` `SD_2` an den Server übertragen. Sobald eine Antwort vorliegen, löst der Baustein `CNF` aus.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, dieselbe C++-Basis wie alle anderen `CLIENT_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `SERVER_0_2`**: `CLIENT_2_0` ist funktional nur mit einem `SERVER_0_2`-Block auf der Server-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `REQ`/`CNF`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 2 Werte gesendet und 0 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Client-seitige Anbindung** eines Kommunikationspartners, der bereits als `SERVER_0_2` mit passender Feldanzahl ausgelegt ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- **SERVER_0_2**: das direkte Gegenstück auf der Server-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

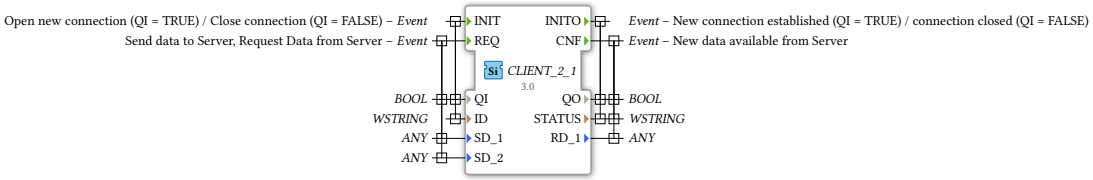
`CLIENT_2_0` liefert die generische, auf 2 Sende- und 0 Empfangsfelder zugeschnittene Client-Variante der `GEN_CLIENT`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.19 CLIENT_2_1

• – ○ • – ○ • – ○ •

Einleitung

Der CLIENT_2_1 Funktionsblock dient zur Kommunikation mit einem SERVER_1_2 Block über eine Netzwerkverbindung. Er ermöglicht das Aufbauen und Trennen von Verbindungen sowie den Datenaustausch zwischen Client und Server.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsereignis zum Öffnen (QI = TRUE) oder Schließen (QI = FALSE) einer Verbindung
- **REQ**: Anforderungsereignis zum Senden von Daten an den Server und Anfordern von Daten vom Server

Ereignis-Ausgänge

- **INITO**: Bestätigungsereignis für Verbindungsaufbau (QI = TRUE) oder Verbindungstrennung (QI = FALSE)
- **CNF**: Bestätigungsereignis bei neu verfügbaren Daten vom Server

Daten-Eingänge

- **QI** (BOOL): Qualitätsindikator für INIT-Ereignis (TRUE = Verbindung öffnen, FALSE = Verbindung schließen)
- **ID** (WSTRING): Identifikator für die Verbindung
- **SD_1** (ANY): Erster Sendeblock für Daten an den Server
- **SD_2** (ANY): Zweiter Sendeblock für Daten an den Server

Daten-Ausgänge

- **QO** (BOOL): Qualitätsindikator für Ausgangsereignisse
- **STATUS** (WSTRING): Statusinformation über die Verbindung oder Operation
- **RD_1** (ANY): Empfangener Datenblock vom Server

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der CLIENT_2_1 Block verwaltet die gesamte Client-Seite einer Client-Server-Kommunikation. Bei INIT mit QI=TRUE wird eine neue Verbindung zum angegebenen Server aufgebaut. Mit INIT und QI=FALSE wird eine bestehende Verbindung geschlossen. Das REQ-Ereignis löst gleichzeitig das Senden der Daten SD_1 und SD_2 an den Server sowie das Anfordern von Daten vom Server aus. Bei erfolgreicher Operation werden die empfangenen Daten über RD_1 ausgegeben.

Technische Besonderheiten

- Unterstützt den Datentyp ANY für flexible Datenübertragung
- Verwendet WSTRING für Statusmeldungen und Verbindungs-IDs
- Bietet bidirektionalen Datenaustausch in einer einzigen Operation
- Implementiert vollständige Verbindungsverwaltung (Aufbau/Trennung)

Zustandsübersicht

Der Block kann folgende Zustände annehmen:

- Nicht verbunden (initialer Zustand)
- Verbindungsaufbau läuft
- Verbunden und betriebsbereit
- Verbindungstrennung läuft

Anwendungsszenarien

- Industrielle Automatisierungssysteme mit verteilter Steuerung
- IoT-Anwendungen mit Client-Server-Architektur
- Datenaustausch zwischen verschiedenen Steuerungsebenen
- Remote-Überwachung und -Steuerung von Geräten

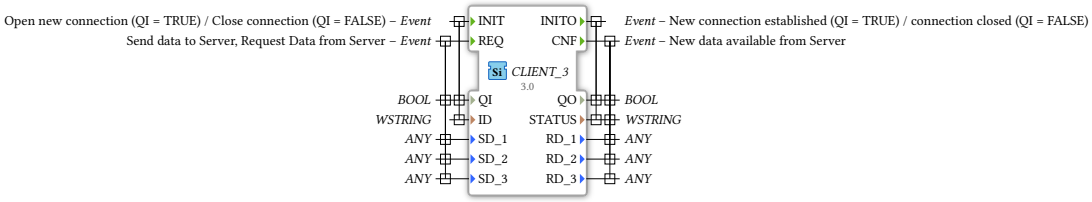
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren Client-Blöcken bietet CLIENT_2_1 die Möglichkeit, zwei verschiedene Datenblöcke gleichzeitig zu senden und einen Datenblock zu empfangen. Die Integration von Verbindungsmanagement und Datenaustausch in einem Block vereinfacht die Anwendungslogik.

Fazit

Der CLIENT_2_1 Funktionsblock stellt eine leistungsfähige Lösung für Client-Server-Kommunikation in verteilten Automatisierungssystemen dar. Durch die Kombination von Verbindungsmanagement und bidirektionalem Datenaustausch in einem einzigen Block ermöglicht er effiziente und übersichtliche Netzwerkanwendungen.

2.4.3.20 CLIENT_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `CLIENT_3`-Funktionsblock ist die generische Client-Variante mit 3 Sende- und 3 Empfangsdatenfeldern für die Kommunikation mit einem passenden SERVER_3-Block. Er überträgt 3 Datenwerte (`SD_1 SD_2 SD_3`) an den Server und empfängt 3 Datenwerte (`RD_1 RD_2 RD_3`) zurück. Wie alle `CLIENT_*`-Bausteine basiert er auf der generischen `GEN_CLIENT`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE), trägt `QI` und `ID`.
- **REQ**: Sendet Daten an den Server und fordert Daten vom Server an, trägt `QI` sowie `SD_1 SD_2 SD_3`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **CNF**: Signalisiert, dass neue Daten vom Server verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1 RD_2 RD_3`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit `REQ` an den Server übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `REQ` an den Server übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit `REQ` an den Server übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `CNF` vom Server geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `CNF` vom Server geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit `CNF` vom Server geliefert.

Funktionsweise

`CLIENT_3` initialisiert über `INIT` eine Verbindung zum passenden `SERVER_3`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `REQ` werden `SD_1 SD_2 SD_3` an den Server übertragen. Sobald Antwortdaten (`RD_1 RD_2 RD_3`) vorliegen, löst der Baustein `CNF` aus und stellt `RD_1 RD_2 RD_3` bereit.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, dieselbe C++-Basis wie alle anderen `CLIENT_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `SERVER_3`**: `CLIENT_3` ist funktional nur mit einem `SERVER_3`-Block auf der Server-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `REQ`/`CNF`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 3 Werte gesendet und 3 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Client-seitige Anbindung** eines Kommunikationspartners, der bereits als `SERVER_3` mit passender Feldanzahl ausgelegt ist.

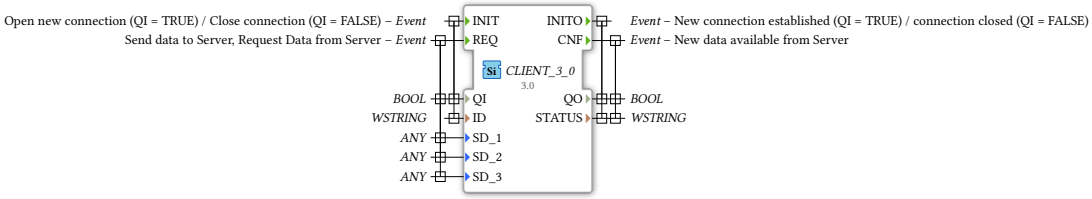
⚖️ Vergleich mit ähnlichen Bausteinen

- **SERVER_3**: das direkte Gegenstück auf der Server-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`CLIENT_3` liefert die generische, auf 3 Sende- und 3 Empfangsfelder zugeschnittene Client-Variante der `GEN_CLIENT`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.21 CLIENT_3_0



• – ○ • – ○ • – ○ •

Einleitung

Der `CLIENT_3_0`-Funktionsblock ist die generische Client-Variante mit 3 Sende- und 0 Empfangsdatenfeldern für die Kommunikation mit einem passenden SERVER_0_3-Block. Er überträgt 3 Datenwerte (`SD_1` `SD_2` `SD_3`) an den Server, ohne Rückgabedaten zu empfangen. Wie alle `CLIENT_*`-Bausteine basiert er auf der generischen `GEN_CLIENT`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE), trägt `QI` und `ID`.
- **REQ**: Sendet Daten an den Server und fordert Daten vom Server an, trägt `QI` sowie `SD_1` `SD_2` `SD_3`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **CNF**: Signalisiert, dass neue Daten vom Server verfügbar sind, trägt `QO`, `STATUS`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit `REQ` an den Server übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `REQ` an den Server übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit `REQ` an den Server übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- Keine Empfangsdaten (`CLIENT_3_0` erhält keine Nutzdaten über `CNF`, nur Verbindungs-/Statusinformationen).

Funktionsweise

`CLIENT_3_0` initialisiert über `INIT` eine Verbindung zum passenden `SERVER_0_3`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `REQ` werden `SD_1` `SD_2` `SD_3` an den Server übertragen. Sobald eine Antwort vorliegen, löst der Baustein `CNF` aus.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, dieselbe C++-Basis wie alle anderen `CLIENT_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `SERVER_0_3`**: `CLIENT_3_0` ist funktional nur mit einem `SERVER_0_3`-Block auf der Server-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `REQ`/`CNF`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 3 Werte gesendet und 0 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Client-seitige Anbindung** eines Kommunikationspartners, der bereits als `SERVER_0_3` mit passender Feldanzahl ausgelegt ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- **SERVER_0_3**: das direkte Gegenstück auf der Server-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

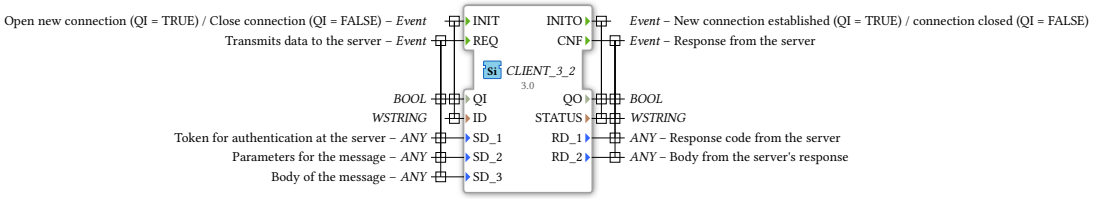
`CLIENT_3_0` liefert die generische, auf 3 Sende- und 0 Empfangsfelder zugeschnittene Client-Variante der `GEN_CLIENT`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.22 CLIENT_3_2

• – ◦ • – ◦ • – ◦ •

Einleitung

Der CLIENT_3_2 ist ein Funktionsblock für HTTP-Kommunikation, der als Client mit drei Eingängen und zwei Ausgängen konzipiert wurde. Er ermöglicht die Kommunikation mit einem Server über das HTTP-Protokoll und unterstützt sowohl das Öffnen und Schließen von Verbindungen als auch das Senden von Anfragen und Empfangen von Antworten.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (Typ: EInit) - Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine Verbindung (QI = FALSE)
- **REQ** (Typ: Event) - Sendet Daten an den Server

Ereignis-Ausgänge

- **INITO** (Typ: EInit) - Signalisiert, dass eine neue Verbindung aufgebaut wurde (QI = TRUE) oder eine Verbindung geschlossen wurde (QI = FALSE)
- **CNF** (Typ: Event) - Antwort vom Server

Daten-Eingänge

- **QI** (BOOL) - Qualitätsindikator für INIT- und REQ-Ereignisse
- **ID** (WSTRING) - Identifikator für die Verbindung
- **SD_1** (ANY) - Token für die Authentifizierung am Server
- **SD_2** (ANY) - Parameter für die Nachricht
- **SD_3** (ANY) - Body der Nachricht

Daten-Ausgänge

- **QO** (BOOL) - Qualitätsindikator für INITO- und CNF-Ereignisse
- **STATUS** (WSTRING) - Statusinformationen
- **RD_1** (ANY) - Antwortcode vom Server
- **RD_2** (ANY) - Body der Server-Antwort

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der CLIENT_3_2 Funktionsblock arbeitet in zwei Hauptphasen: Verbindungsmanagement und Datenübertragung. Über das INIT-Ereignis wird zunächst eine Verbindung zum Server aufgebaut oder geschlossen. Nach erfolgreichem Verbindungsaufbau können über das REQ-Ereignis Daten an den Server gesendet werden. Die Server-Antwort wird über das CNF-Ereignis zurückgemeldet.

Technische Besonderheiten

- Verwendung von WSTRING für ID und STATUS für internationale Zeichensatzunterstützung
- ANY-Typ für SD_1, SD_2, SD_3, RD_1 und RD_2 ermöglicht flexible Datentypen
- Separate Qualitätsindikatoren für Eingänge (QI) und Ausgänge (QO)
- Unterstützung von Authentifizierungstokens (SD_1)

Zustandsübersicht

1. **Initialisierungszustand**: Wartet auf INIT-Ereignis
2. **Verbindungsaufbau**: Verarbeitet INIT mit QI=TRUE
3. **Bereit**: Verbindung steht, wartet auf REQ-Ereignisse
4. **Datenübertragung**: Sendet REQ und wartet auf CNF
5. **Verbindungsabbau**: Verarbeitet INIT mit QI=FALSE

Anwendungsszenarien

- HTTP-REST-API-Kommunikation in Automatisierungssystemen
- Web-Service-Integration in industriellen Steuerungen
- Datenaustausch mit Cloud-Diensten

- Authentifizierte Kommunikation mit Backend-Servern

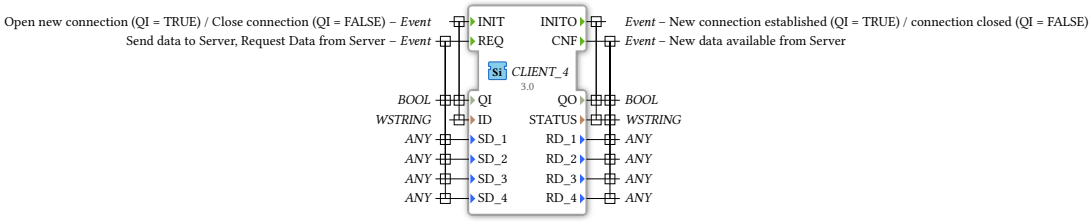
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren Client-Bausteinen bietet CLIENT_3_2 erweiterte Funktionalität durch drei separate Dateneingänge (SD_1-3) und zwei Datenausgänge (RD_1-2), was eine strukturierte Übertragung von Authentifizierungsdaten, Parametern und Nachrichteninhalten ermöglicht.

Fazit

Der CLIENT_3_2 ist ein vielseitiger HTTP-Client-Funktionsblock, der durch seine flexible Datenschnittstelle und umfassende Statusrückmeldungen eine robuste Kommunikation mit HTTP-Servern in industriellen Automatisierungsumgebungen ermöglicht.

2.4.3.23 CLIENT_4



• – ○ • – ○ • – ○ •

Einleitung

Der `CLIENT_4`-Funktionsblock ist die generische Client-Variante mit 4 Sende- und 4 Empfangsdatenfeldern für die Kommunikation mit einem passenden SERVER_4-Block. Er überträgt 4 Datenwerte (`SD_1` `SD_2` `SD_3` `SD_4`) an den Server und empfängt 4 Datenwerte (`RD_1` `RD_2` `RD_3` `RD_4`) zurück. Wie alle `CLIENT_*`-Bausteine basiert er auf der generischen `GEN_CLIENT`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE), trägt `QI` und `ID`.
- **REQ**: Sendet Daten an den Server und fordert Daten vom Server an, trägt `QI` sowie `SD_1` `SD_2` `SD_3` `SD_4`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **CNF**: Signalisiert, dass neue Daten vom Server verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1` `RD_2` `RD_3` `RD_4`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit `REQ` an den Server übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `REQ` an den Server übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit `REQ` an den Server übertragen.
- **SD_4** (ANY): Sendedatum 4, wird mit `REQ` an den Server übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `CNF` vom Server geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `CNF` vom Server geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit `CNF` vom Server geliefert.
- **RD_4** (ANY): Empfangsdatum 4, wird mit `CNF` vom Server geliefert.

Funktionsweise

`CLIENT_4` initialisiert über `INIT` eine Verbindung zum passenden `SERVER_4`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `REQ` werden `SD_1` `SD_2` `SD_3` `SD_4` an den Server übertragen. Sobald Antwortdaten (`RD_1` `RD_2` `RD_3` `RD_4`) vorliegen, löst der Baustein `CNF` aus und stellt `RD_1` `RD_2` `RD_3` `RD_4` bereit.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, dieselbe C++-Basis wie alle anderen `CLIENT_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `SERVER_4`**: `CLIENT_4` ist funktional nur mit einem `SERVER_4`-Block auf der Server-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `REQ`/`CNF`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 4 Werte gesendet und 4 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Client-seitige Anbindung** eines Kommunikationspartners, der bereits als `SERVER_4` mit passender Feldanzahl ausgelegt ist.

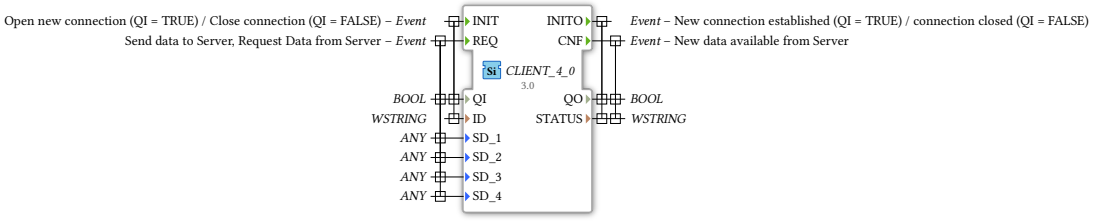
⚖️ Vergleich mit ähnlichen Bausteinen

- SERVER_4: das direkte Gegenstück auf der Server-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- CLIENT_1 / SERVER_1: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`CLIENT_4` liefert die generische, auf 4 Sende- und 4 Empfangsfelder zugeschnittene Client-Variante der `GEN_CLIENT`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.24 CLIENT_4_0



• – ○ • – ○ • – ○ •

Einleitung

Der `CLIENT_4_0`-Funktionsblock ist die generische Client-Variante mit 4 Sende- und 0 Empfangsdatenfeldern für die Kommunikation mit einem passenden SERVER_0_4-Block. Er überträgt 4 Datenwerte (`SD_1` `SD_2` `SD_3` `SD_4`) an den Server, ohne Rückgabedaten zu empfangen. Wie alle `CLIENT_*`-Bausteine basiert er auf der generischen `GEN_CLIENT`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE), trägt `QI` und `ID`.
- **REQ**: Sendet Daten an den Server und fordert Daten vom Server an, trägt `QI` sowie `SD_1` `SD_2` `SD_3` `SD_4`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **CNF**: Signalisiert, dass neue Daten vom Server verfügbar sind, trägt `QO`, `STATUS`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit `REQ` an den Server übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `REQ` an den Server übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit `REQ` an den Server übertragen.
- **SD_4** (ANY): Sendedatum 4, wird mit `REQ` an den Server übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- Keine Empfangsdaten (`CLIENT_4_0` erhält keine Nutzdaten über `CNF`, nur Verbindungs-/Statusinformationen).

Funktionsweise

`CLIENT_4_0` initialisiert über `INIT` eine Verbindung zum passenden `SERVER_0_4`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `REQ` werden `SD_1` `SD_2` `SD_3` `SD_4` an den Server übertragen. Sobald eine Antwort vorliegen, löst der Baustein `CNF` aus.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, dieselbe C++-Basis wie alle anderen `CLIENT_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `SERVER_0_4`**: `CLIENT_4_0` ist funktional nur mit einem `SERVER_0_4`-Block auf der Server-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `REQ`/`CNF`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 4 Werte gesendet und 0 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Client-seitige Anbindung** eines Kommunikationspartners, der bereits als `SERVER_0_4` mit passender Feldanzahl ausgelegt ist.

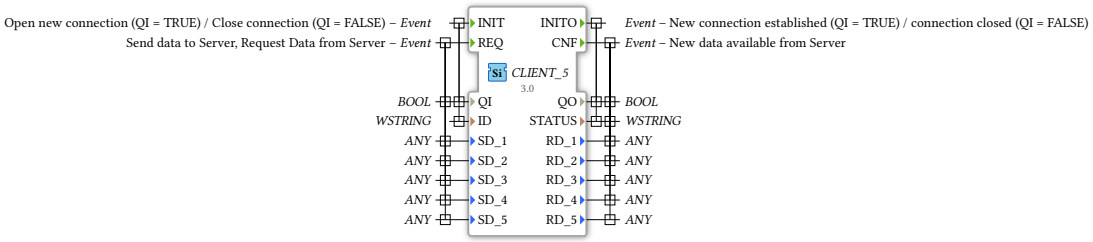
⚖️ Vergleich mit ähnlichen Bausteinen

- SERVER_0_4: das direkte Gegenstück auf der Server-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- CLIENT_1 / SERVER_1: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`CLIENT_4_0` liefert die generische, auf 4 Sende- und 0 Empfangsfelder zugeschnittene Client-Variante der `GEN_CLIENT`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.25 CLIENT_5



• – ○ • – ○ • – ○ •

Einleitung

Der `CLIENT_5`-Funktionsblock ist die generische Client-Variante mit 5 Sende- und 5 Empfangsdatenfeldern für die Kommunikation mit einem passenden SERVER_5-Block. Er überträgt 5 Datenwerte (`SD_1` `SD_2` `SD_3` `SD_4` `SD_5`) an den Server und empfängt 5 Datenwerte (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5`) zurück. Wie alle `CLIENT_*`-Bausteine basiert er auf der generischen `GEN_CLIENT`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE), trägt `QI` und `ID`.
- **REQ**: Sendet Daten an den Server und fordert Daten vom Server an, trägt `QI` sowie `SD_1` `SD_2` `SD_3` `SD_4` `SD_5`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **CNF**: Signalisiert, dass neue Daten vom Server verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1` `RD_2` `RD_3` `RD_4` `RD_5`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit `REQ` an den Server übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `REQ` an den Server übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit `REQ` an den Server übertragen.
- **SD_4** (ANY): Sendedatum 4, wird mit `REQ` an den Server übertragen.
- **SD_5** (ANY): Sendedatum 5, wird mit `REQ` an den Server übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `CNF` vom Server geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `CNF` vom Server geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit `CNF` vom Server geliefert.
- **RD_4** (ANY): Empfangsdatum 4, wird mit `CNF` vom Server geliefert.
- **RD_5** (ANY): Empfangsdatum 5, wird mit `CNF` vom Server geliefert.

Funktionsweise

`CLIENT_5` initialisiert über `INIT` eine Verbindung zum passenden `SERVER_5`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `REQ` werden `SD_1` `SD_2` `SD_3`

SD_4 SD_5 an den Server übertragen. Sobald Antwortdaten (RD_1 RD_2 RD_3 RD_4 RD_5) vorliegen, löst der Baustein CNF aus und stellt RD_1 RD_2 RD_3 RD_4 RD_5 bereit.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, dieselbe C++-Basis wie alle anderen CLIENT_*-Varianten; Anzahl und Typ (ANY) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **ANY-Datenfelder**: Alle SD_i/RD_i sind generisch (ANY) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück SERVER_5**: CLIENT_5 ist funktional nur mit einem SERVER_5-Block auf der Server-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, QO = FALSE.
2. **Verbindungsaufbau**: INIT mit QI = TRUE wird verarbeitet.
3. **Verbunden**: INITO mit QO = TRUE bestätigt die Verbindung.
4. **Datenaustausch**: REQ/CNF-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: INIT mit QI = FALSE wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 5 Werte gesendet und 5 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Client-seitige Anbindung** eines Kommunikationspartners, der bereits als SERVER_5 mit passender Feldanzahl ausgelegt ist.

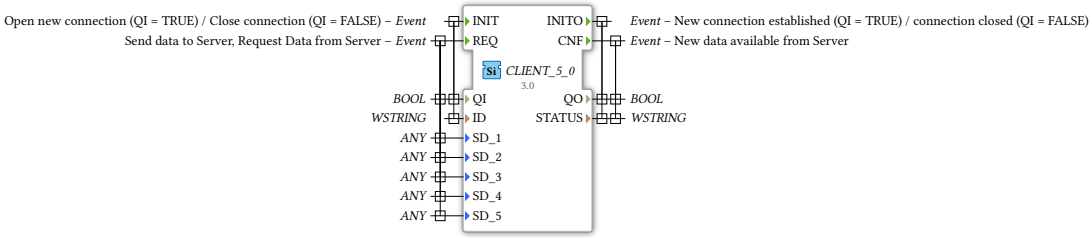
⚖️ Vergleich mit ähnlichen Bausteinen

- SERVER_5: das direkte Gegenstück auf der Server-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- CLIENT_1 / SERVER_1: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

CLIENT_5 liefert die generische, auf 5 Sende- und 5 Empfangsfelder zugeschnittene Client-Variante der GEN_CLIENT-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.26 CLIENT_5_0



• – ○ • – ○ • – ○ •

Einleitung

Der CLIENT_5_0-Funktionsblock ist die generische Client-Variante mit 5 Sende- und 0 Empfangsdatenfeldern für die Kommunikation mit einem passenden SERVER_0_5-Block. Er überträgt 5 Datenwerte (SD_1 SD_2 SD_3 SD_4 SD_5) an den Server, ohne Rückgabedaten zu empfangen. Wie alle CLIENT_*-Bausteine basiert er auf der generischen GEN_CLIENT-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE), trägt QI und ID.

- **REQ**: Sendet Daten an den Server und fordert Daten vom Server an, trägt `QI` sowie `SD_1` `SD_2` `SD_3` `SD_4` `SD_5`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **CNF**: Signalisiert, dass neue Daten vom Server verfügbar sind, trägt `QO`, `STATUS`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit `REQ` an den Server übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `REQ` an den Server übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit `REQ` an den Server übertragen.
- **SD_4** (ANY): Sendedatum 4, wird mit `REQ` an den Server übertragen.
- **SD_5** (ANY): Sendedatum 5, wird mit `REQ` an den Server übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- Keine Empfangsdaten (`CLIENT_5_0` erhält keine Nutzdaten über `CNF`, nur Verbindungs-/Statusinformationen).

Funktionsweise

`CLIENT_5_0` initialisiert über `INIT` eine Verbindung zum passenden `SERVER_0_5`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `REQ` werden `SD_1` `SD_2` `SD_3` `SD_4` `SD_5` an den Server übertragen. Sobald eine Antwort vorliegen, löst der Baustein `CNF` aus.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, dieselbe C++-Basis wie alle anderen `CLIENT_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `SERVER_0_5`**: `CLIENT_5_0` ist funktional nur mit einem `SERVER_0_5`-Block auf der Server-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `REQ`/`CNF`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 5 Werte gesendet und 0 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Client-seitige Anbindung** eines Kommunikationspartners, der bereits als `SERVER_0_5` mit passender Feldanzahl ausgelegt ist.

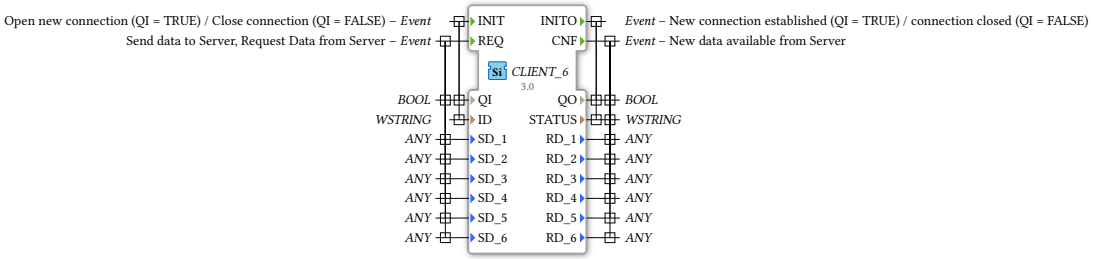
⚖️ Vergleich mit ähnlichen Bausteinen

- **SERVER_0_5**: das direkte Gegenstück auf der Server-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`CLIENT_5_0` liefert die generische, auf 5 Sende- und 0 Empfangsfelder zugeschnittene Client-Variante der `GEN_CLIENT`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.27 CLIENT_6



• – ○ • – ○ • – ○ •

Einleitung

Der `CLIENT_6`-Funktionsblock ist die generische Client-Variante mit 6 Sende- und 6 Empfangsdatenfeldern für die Kommunikation mit einem passenden SERVER_6-Block. Er überträgt 6 Datenwerte (`SD_1 SD_2 SD_3 SD_4 SD_5 SD_6`) an den Server und empfängt 6 Datenwerte (`RD_1 RD_2 RD_3 RD_4 RD_5 RD_6`) zurück. Wie alle `CLIENT_*`-Bausteine basiert er auf der generischen `GEN_CLIENT`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE), trägt `QI` und `ID`.
- **REQ**: Sendet Daten an den Server und fordert Daten vom Server an, trägt `QI` sowie `SD_1 SD_2 SD_3 SD_4 SD_5 SD_6`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **CNF**: Signalisiert, dass neue Daten vom Server verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1 RD_2 RD_3 RD_4 RD_5 RD_6`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit `REQ` an den Server übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `REQ` an den Server übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit `REQ` an den Server übertragen.
- **SD_4** (ANY): Sendedatum 4, wird mit `REQ` an den Server übertragen.
- **SD_5** (ANY): Sendedatum 5, wird mit `REQ` an den Server übertragen.
- **SD_6** (ANY): Sendedatum 6, wird mit `REQ` an den Server übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `CNF` vom Server geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `CNF` vom Server geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit `CNF` vom Server geliefert.
- **RD_4** (ANY): Empfangsdatum 4, wird mit `CNF` vom Server geliefert.
- **RD_5** (ANY): Empfangsdatum 5, wird mit `CNF` vom Server geliefert.
- **RD_6** (ANY): Empfangsdatum 6, wird mit `CNF` vom Server geliefert.

Funktionsweise

`CLIENT_6` initialisiert über `INIT` eine Verbindung zum passenden `SERVER_6`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `REQ` werden `SD_1 SD_2 SD_3 SD_4 SD_5 SD_6` an den Server übertragen. Sobald Antwortdaten (`RD_1 RD_2 RD_3 RD_4 RD_5 RD_6`) vorliegen, löst der Baustein `CNF` aus und stellt `RD_1 RD_2 RD_3 RD_4 RD_5 RD_6` bereit.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, dieselbe C++-Basis wie alle anderen `CLIENT_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `SERVER_6`**: `CLIENT_6` ist funktional nur mit einem `SERVER_6`-Block auf der Server-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `REQ`/`CNF`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 6 Werte gesendet und 6 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Client-seitige Anbindung** eines Kommunikationspartners, der bereits als `SERVER_6` mit passender Feldanzahl ausgelegt ist.

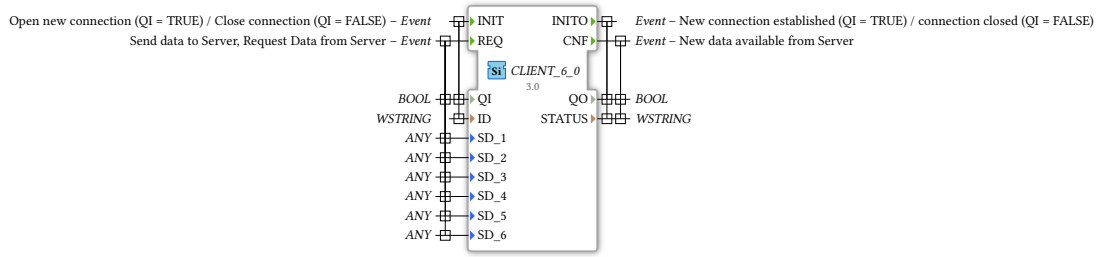
⚖️ Vergleich mit ähnlichen Bausteinen

- **SERVER_6**: das direkte Gegenstück auf der Server-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`CLIENT_6` liefert die generische, auf 6 Sende- und 6 Empfangsfelder zugeschnittene Client-Variante der `GEN_CLIENT`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.28 CLIENT_6_0



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `CLIENT_6_0`-Funktionsblock ist die generische Client-Variante mit 6 Sende- und 0 Empfangsdatenfeldern für die Kommunikation mit einem passenden SERVER_0_6-Block. Er überträgt 6 Datenwerte (`SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6`) an den Server, ohne Rückgabedaten zu empfangen. Wie alle `CLIENT_*`-Bausteine basiert er auf der generischen `GEN_CLIENT`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE), trägt `QI` und `ID`.
- **REQ**: Sendet Daten an den Server und fordert Daten vom Server an, trägt `QI` sowie `SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **CNF**: Signalisiert, dass neue Daten vom Server verfügbar sind, trägt `QO`, `STATUS`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit `REQ` an den Server übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `REQ` an den Server übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit `REQ` an den Server übertragen.
- **SD_4** (ANY): Sendedatum 4, wird mit `REQ` an den Server übertragen.
- **SD_5** (ANY): Sendedatum 5, wird mit `REQ` an den Server übertragen.
- **SD_6** (ANY): Sendedatum 6, wird mit `REQ` an den Server übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- Keine Empfangsdaten (`CLIENT_6_0` erhält keine Nutzdaten über `CNF`, nur Verbindungs-/Statusinformationen).

Funktionsweise

`CLIENT_6_0` initialisiert über `INIT` eine Verbindung zum passenden `SERVER_0_6`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `REQ` werden `SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` an den Server übertragen. Sobald eine Antwort vorliegen, löst der Baustein `CNF` aus.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, dieselbe C++-Basis wie alle anderen `CLIENT_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `SERVER_0_6`**: `CLIENT_6_0` ist funktional nur mit einem `SERVER_0_6`-Block auf der Server-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `REQ`/`CNF`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 6 Werte gesendet und 0 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Client-seitige Anbindung** eines Kommunikationspartners, der bereits als `SERVER_0_6` mit passender Feldanzahl ausgelegt ist.

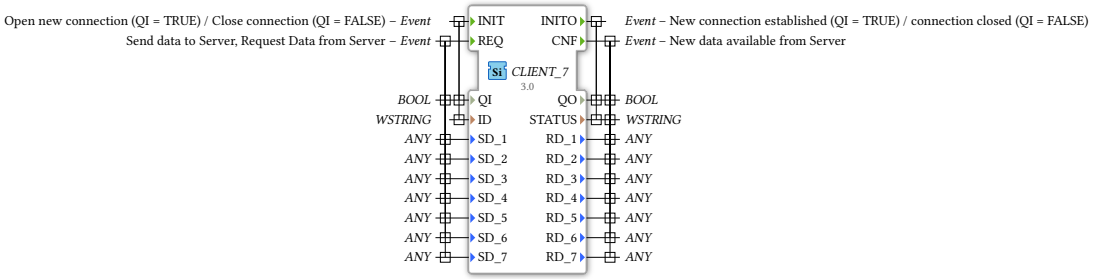
⚖️ Vergleich mit ähnlichen Bausteinen

- **SERVER_0_6**: das direkte Gegenstück auf der Server-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`CLIENT_6_0` liefert die generische, auf 6 Sende- und 0 Empfangsfelder zugeschnittene Client-Variante der `GEN_CLIENT`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.29 CLIENT_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `CLIENT_7`-Funktionsblock ist die generische Client-Variante mit 7 Sende- und 7 Empfangsdatenfeldern für die Kommunikation mit einem passenden SERVER_7-Block. Er überträgt 7 Datenwerte (`SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7`) an den Server und empfängt 7 Datenwerte (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7`) zurück. Wie alle `CLIENT_*`-Bausteine basiert er auf der generischen `GEN_CLIENT`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE), trägt `QI` und `ID`.
- **REQ**: Sendet Daten an den Server und fordert Daten vom Server an, trägt `QI` sowie `SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **CNF**: Signalisiert, dass neue Daten vom Server verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit `REQ` an den Server übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `REQ` an den Server übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit `REQ` an den Server übertragen.
- **SD_4** (ANY): Sendedatum 4, wird mit `REQ` an den Server übertragen.
- **SD_5** (ANY): Sendedatum 5, wird mit `REQ` an den Server übertragen.
- **SD_6** (ANY): Sendedatum 6, wird mit `REQ` an den Server übertragen.
- **SD_7** (ANY): Sendedatum 7, wird mit `REQ` an den Server übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `CNF` vom Server geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `CNF` vom Server geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit `CNF` vom Server geliefert.
- **RD_4** (ANY): Empfangsdatum 4, wird mit `CNF` vom Server geliefert.
- **RD_5** (ANY): Empfangsdatum 5, wird mit `CNF` vom Server geliefert.
- **RD_6** (ANY): Empfangsdatum 6, wird mit `CNF` vom Server geliefert.
- **RD_7** (ANY): Empfangsdatum 7, wird mit `CNF` vom Server geliefert.

Funktionsweise

`CLIENT_7` initialisiert über `INIT` eine Verbindung zum passenden `SERVER_7`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `REQ` werden `SD_1` `SD_2` `SD_3`

SD_4 SD_5 SD_6 SD_7 an den Server übertragen. Sobald Antwortdaten (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7) vorliegen, löst der Baustein CNF aus und stellt RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 bereit.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, dieselbe C++-Basis wie alle anderen CLIENT_*-Varianten; Anzahl und Typ (ANY) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **ANY-Datenfelder**: Alle SD_i/RD_i sind generisch (ANY) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück SERVER_7**: CLIENT_7 ist funktional nur mit einem SERVER_7-Block auf der Server-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, QO = FALSE.
2. **Verbindungsaufbau**: INIT mit QI = TRUE wird verarbeitet.
3. **Verbunden**: INITO mit QO = TRUE bestätigt die Verbindung.
4. **Datenaustausch**: REQ/CNF-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: INIT mit QI = FALSE wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 7 Werte gesendet und 7 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Client-seitige Anbindung** eines Kommunikationspartners, der bereits als SERVER_7 mit passender Feldanzahl ausgelegt ist.

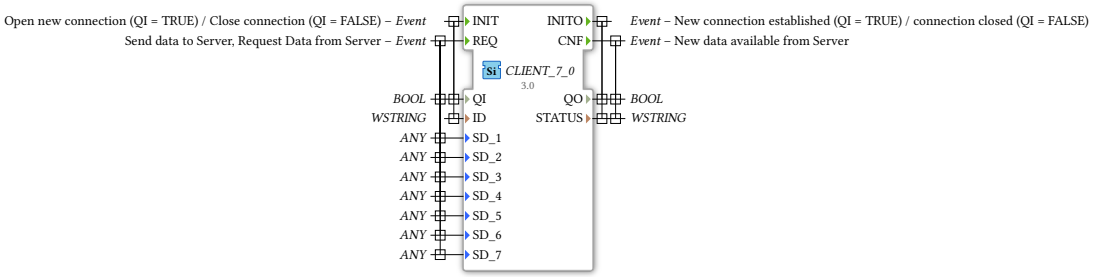
⚖️ Vergleich mit ähnlichen Bausteinen

- SERVER_7: das direkte Gegenstück auf der Server-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- CLIENT_1 / SERVER_1: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

CLIENT_7 liefert die generische, auf 7 Sende- und 7 Empfangsfelder zugeschnittene Client-Variante der GEN_CLIENT-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.30 CLIENT_7_0



• – ○ • – ○ • – ○ •

Einleitung

Der CLIENT_7_0-Funktionsblock ist die generische Client-Variante mit 7 Sende- und 0 Empfangsdatenfeldern für die Kommunikation mit einem passenden SERVER_0_7-Block. Er überträgt 7 Datenwerte (SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7) an den Server, ohne Rückgabedaten zu empfangen. Wie alle CLIENT_*-Bausteine basiert er auf der generischen GEN_CLIENT-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE), trägt `QI` und `ID`.
- **REQ**: Sendet Daten an den Server und fordert Daten vom Server an, trägt `QI` sowie `SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **CNF**: Signalisiert, dass neue Daten vom Server verfügbar sind, trägt `QO`, `STATUS`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit `REQ` an den Server übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `REQ` an den Server übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit `REQ` an den Server übertragen.
- **SD_4** (ANY): Sendedatum 4, wird mit `REQ` an den Server übertragen.
- **SD_5** (ANY): Sendedatum 5, wird mit `REQ` an den Server übertragen.
- **SD_6** (ANY): Sendedatum 6, wird mit `REQ` an den Server übertragen.
- **SD_7** (ANY): Sendedatum 7, wird mit `REQ` an den Server übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- Keine Empfangsdaten (`CLIENT_7_0` erhält keine Nutzdaten über `CNF`, nur Verbindungs-/Statusinformationen).

Funktionsweise

`CLIENT_7_0` initialisiert über `INIT` eine Verbindung zum passenden `SERVER_0_7`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `REQ` werden `SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7` an den Server übertragen. Sobald eine Antwort vorliegen, löst der Baustein `CNF` aus.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, dieselbe C++-Basis wie alle anderen `CLIENT_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `SERVER_0_7`**: `CLIENT_7_0` ist funktional nur mit einem `SERVER_0_7`-Block auf der Server-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `REQ`/`CNF`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 7 Werte gesendet und 0 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Client-seitige Anbindung** eines Kommunikationspartners, der bereits als `SERVER_0_7` mit passender Feldanzahl ausgelegt ist.

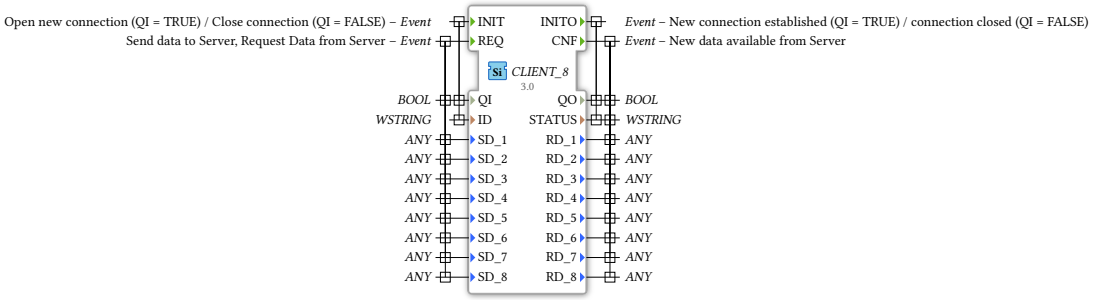
⚖️ Vergleich mit ähnlichen Bausteinen

- SERVER_0_7: das direkte Gegenstück auf der Server-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- CLIENT_1 / SERVER_1: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`CLIENT_7_0` liefert die generische, auf 7 Sende- und 0 Empfangsfelder zugeschnittene Client-Variante der `GEN_CLIENT`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.31 CLIENT_8



• – ○ • – ○ • – ○ •

Einleitung

Der `CLIENT_8`-Funktionsblock ist die generische Client-Variante mit 8 Sende- und 8 Empfangsdatenfeldern für die Kommunikation mit einem passenden SERVER_8-Block. Er überträgt 8 Datenwerte (`SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8`) an den Server und empfängt 8 Datenwerte (`RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8`) zurück. Wie alle `CLIENT_*`-Bausteine basiert er auf der generischen `GEN_CLIENT`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE), trägt `QI` und `ID`.
- **REQ**: Sendet Daten an den Server und fordert Daten vom Server an, trägt `QI` sowie `SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **CNF**: Signalisiert, dass neue Daten vom Server verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit `REQ` an den Server übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `REQ` an den Server übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit `REQ` an den Server übertragen.
- **SD_4** (ANY): Sendedatum 4, wird mit `REQ` an den Server übertragen.
- **SD_5** (ANY): Sendedatum 5, wird mit `REQ` an den Server übertragen.
- **SD_6** (ANY): Sendedatum 6, wird mit `REQ` an den Server übertragen.
- **SD_7** (ANY): Sendedatum 7, wird mit `REQ` an den Server übertragen.
- **SD_8** (ANY): Sendedatum 8, wird mit `REQ` an den Server übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.

- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `CNF` vom Server geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `CNF` vom Server geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit `CNF` vom Server geliefert.
- **RD_4** (ANY): Empfangsdatum 4, wird mit `CNF` vom Server geliefert.
- **RD_5** (ANY): Empfangsdatum 5, wird mit `CNF` vom Server geliefert.
- **RD_6** (ANY): Empfangsdatum 6, wird mit `CNF` vom Server geliefert.
- **RD_7** (ANY): Empfangsdatum 7, wird mit `CNF` vom Server geliefert.
- **RD_8** (ANY): Empfangsdatum 8, wird mit `CNF` vom Server geliefert.

Funktionsweise

`CLIENT_8` initialisiert über `INIT` eine Verbindung zum passenden `SERVER_8`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `REQ` werden `SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8` an den Server übertragen. Sobald Antwortdaten (`RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8`) vorliegen, löst der Baustein `CNF` aus und stellt `RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8` bereit.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, dieselbe C++-Basis wie alle anderen `CLIENT_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `SERVER_8`**: `CLIENT_8` ist funktional nur mit einem `SERVER_8`-Block auf der Server-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `REQ`/`CNF`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 8 Werte gesendet und 8 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Client-seitige Anbindung** eines Kommunikationspartners, der bereits als `SERVER_8` mit passender Feldanzahl ausgelegt ist.

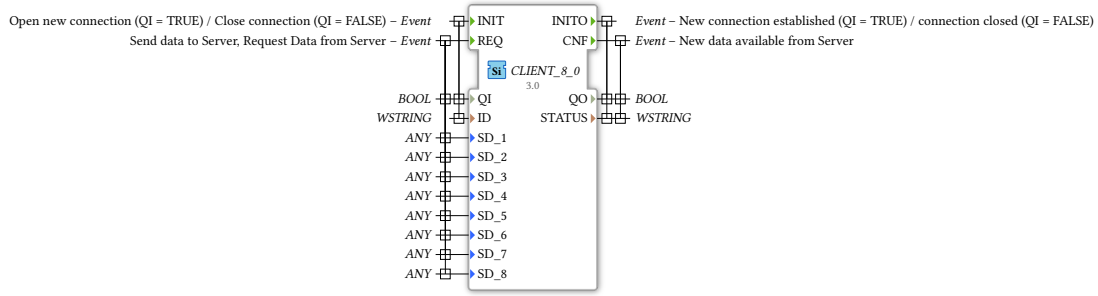
⚖️ Vergleich mit ähnlichen Bausteinen

- **SERVER_8**: das direkte Gegenstück auf der Server-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`CLIENT_8` liefert die generische, auf 8 Sende- und 8 Empfangsfelder zugeschnittene Client-Variante der `GEN_CLIENT`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.32 CLIENT_8_0



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `CLIENT_8_0`-Funktionsblock ist die generische Client-Variante mit 8 Sende- und 0 Empfangsdatenfeldern für die Kommunikation mit einem passenden SERVER_0_8-Block. Er überträgt 8 Datenwerte (`SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8`) an den Server, ohne Rückgabedaten zu empfangen. Wie alle `CLIENT_*`-Bausteine basiert er auf der generischen `GEN_CLIENT`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE), trägt `QI` und `ID`.
- **REQ**: Sendet Daten an den Server und fordert Daten vom Server an, trägt `QI` sowie `SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **CNF**: Signalisiert, dass neue Daten vom Server verfügbar sind, trägt `QO`, `STATUS`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit `REQ` an den Server übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `REQ` an den Server übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit `REQ` an den Server übertragen.
- **SD_4** (ANY): Sendedatum 4, wird mit `REQ` an den Server übertragen.
- **SD_5** (ANY): Sendedatum 5, wird mit `REQ` an den Server übertragen.
- **SD_6** (ANY): Sendedatum 6, wird mit `REQ` an den Server übertragen.
- **SD_7** (ANY): Sendedatum 7, wird mit `REQ` an den Server übertragen.
- **SD_8** (ANY): Sendedatum 8, wird mit `REQ` an den Server übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- Keine Empfangsdaten (`CLIENT_8_0` erhält keine Nutzdaten über `CNF`, nur Verbindungs-/Statusinformationen).

Funktionsweise

`CLIENT_8_0` initialisiert über `INIT` eine Verbindung zum passenden `SERVER_0_8`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `REQ` werden `SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8` an den Server übertragen. Sobald eine Antwort vorliegen, löst der Baustein `CNF` aus.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, dieselbe C++-Basis wie alle anderen `CLIENT_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `SERVER_0_8`**: `CLIENT_8_0` ist funktional nur mit einem `SERVER_0_8`-Block auf der Server-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `REQ`/`CNF`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 8 Werte gesendet und 0 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Client-seitige Anbindung** eines Kommunikationspartners, der bereits als `SERVER_0_8` mit passender Feldanzahl ausgelegt ist.

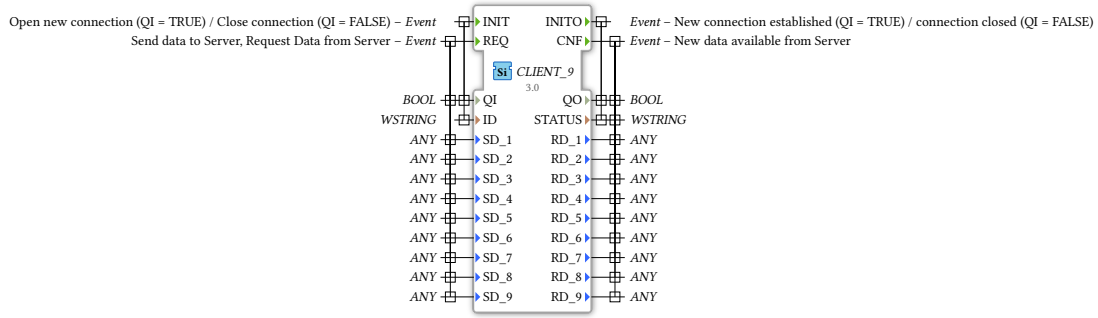
⚖️ Vergleich mit ähnlichen Bausteinen

- **SERVER_0_8**: das direkte Gegenstück auf der Server-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`CLIENT_8_0` liefert die generische, auf 8 Sende- und 0 Empfangsfelder zugeschnittene Client-Variante der `GEN_CLIENT`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.33 CLIENT_9



• – ○ • – ○ • – ○ •

Einleitung

Der `CLIENT_9`-Funktionsblock ist die generische Client-Variante mit 9 Sende- und 9 Empfangsdatenfeldern für die Kommunikation mit einem passenden SERVER_9-Block. Er überträgt 9 Datenwerte (`SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8 SD_9`) an den Server und empfängt 9 Datenwerte (`RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9`) zurück. Wie alle `CLIENT_*`-Bausteine basiert er auf der generischen `GEN_CLIENT`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE), trägt `QI` und `ID`.
- **REQ**: Sendet Daten an den Server und fordert Daten vom Server an, trägt `QI` sowie `SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8 SD_9`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **CNF**: Signalisiert, dass neue Daten vom Server verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit `REQ` an den Server übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `REQ` an den Server übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit `REQ` an den Server übertragen.
- **SD_4** (ANY): Sendedatum 4, wird mit `REQ` an den Server übertragen.
- **SD_5** (ANY): Sendedatum 5, wird mit `REQ` an den Server übertragen.
- **SD_6** (ANY): Sendedatum 6, wird mit `REQ` an den Server übertragen.
- **SD_7** (ANY): Sendedatum 7, wird mit `REQ` an den Server übertragen.
- **SD_8** (ANY): Sendedatum 8, wird mit `REQ` an den Server übertragen.
- **SD_9** (ANY): Sendedatum 9, wird mit `REQ` an den Server übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `CNF` vom Server geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `CNF` vom Server geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit `CNF` vom Server geliefert.
- **RD_4** (ANY): Empfangsdatum 4, wird mit `CNF` vom Server geliefert.
- **RD_5** (ANY): Empfangsdatum 5, wird mit `CNF` vom Server geliefert.
- **RD_6** (ANY): Empfangsdatum 6, wird mit `CNF` vom Server geliefert.
- **RD_7** (ANY): Empfangsdatum 7, wird mit `CNF` vom Server geliefert.
- **RD_8** (ANY): Empfangsdatum 8, wird mit `CNF` vom Server geliefert.
- **RD_9** (ANY): Empfangsdatum 9, wird mit `CNF` vom Server geliefert.

Funktionsweise

`CLIENT_9` initialisiert über `INIT` eine Verbindung zum passenden `SERVER_9`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `REQ` werden `SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8 SD_9` an den Server übertragen. Sobald Antwortdaten (`RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9`) vorliegen, löst der Baustein `CNF` aus und stellt `RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9` bereit.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, dieselbe C++-Basis wie alle anderen `CLIENT_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `SERVER_9`**: `CLIENT_9` ist funktional nur mit einem `SERVER_9`-Block auf der Server-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.

3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `REQ`/`CNF`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 9 Werte gesendet und 9 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Client-seitige Anbindung** eines Kommunikationspartners, der bereits als `SERVER_9` mit passender Feldanzahl ausgelegt ist.

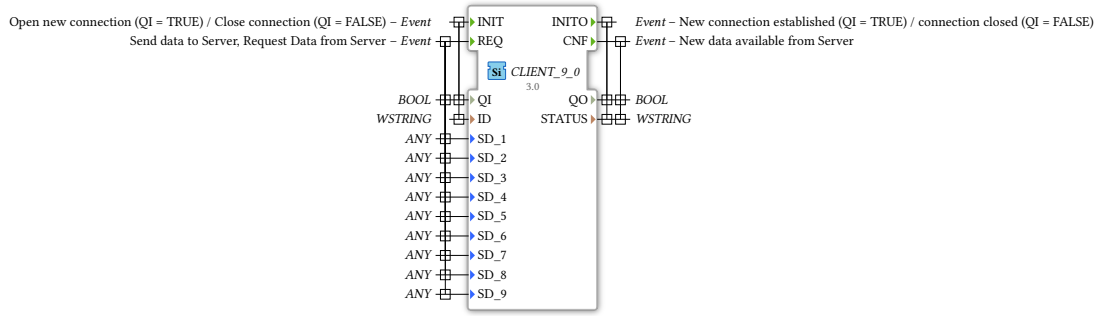
⚖️ Vergleich mit ähnlichen Bausteinen

- **SERVER_9**: das direkte Gegenstück auf der Server-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`CLIENT_9` liefert die generische, auf 9 Sende- und 9 Empfangsfelder zugeschnittene Client-Variante der `GEN_CLIENT`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.34 CLIENT_9_0



• – ○ • – ○ • – ○ •

Einleitung

Der `CLIENT_9_0`-Funktionsblock ist die generische Client-Variante mit 9 Sende- und 0 Empfangsdatenfeldern für die Kommunikation mit einem passenden SERVER_0_9-Block. Er überträgt 9 Datenwerte (`SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8 SD_9`) an den Server, ohne Rückgabedaten zu empfangen. Wie alle `CLIENT_*`-Bausteine basiert er auf der generischen `GEN_CLIENT`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Öffnet eine neue Verbindung (QI = TRUE) oder schließt eine bestehende Verbindung (QI = FALSE), trägt `QI` und `ID`.
- **REQ**: Sendet Daten an den Server und fordert Daten vom Server an, trägt `QI` sowie `SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8 SD_9`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **CNF**: Signalisiert, dass neue Daten vom Server verfügbar sind, trägt `QO`, `STATUS`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit `REQ` an den Server übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `REQ` an den Server übertragen.

- **SD_3** (ANY): Sendedatum 3, wird mit REQ an den Server übertragen.
- **SD_4** (ANY): Sendedatum 4, wird mit REQ an den Server übertragen.
- **SD_5** (ANY): Sendedatum 5, wird mit REQ an den Server übertragen.
- **SD_6** (ANY): Sendedatum 6, wird mit REQ an den Server übertragen.
- **SD_7** (ANY): Sendedatum 7, wird mit REQ an den Server übertragen.
- **SD_8** (ANY): Sendedatum 8, wird mit REQ an den Server übertragen.
- **SD_9** (ANY): Sendedatum 9, wird mit REQ an den Server übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- Keine Empfangsdaten (CLIENT_9_0 erhält keine Nutzdaten über CNF, nur Verbindungs-/Statusinformationen).

Funktionsweise

CLIENT_9_0 initialisiert über INIT eine Verbindung zum passenden SERVER_0_9-Block (bei QI = TRUE) bzw. schließt sie (bei QI = FALSE); der Abschluss wird über INITO bestätigt. Über REQ werden SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8 SD_9 an den Server übertragen. Sobald eine Antwort vorliegen, löst der Baustein CNF aus.

Technische Besonderheiten

- **Generische Implementierung**: eclipse4diac::core::GenericClassName = 'GEN_CLIENT', dieselbe C++-Basis wie alle anderen CLIENT_*-Varianten; Anzahl und Typ (ANY) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **ANY-Datenfelder**: Alle SD_i/RD_i sind generisch (ANY) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück SERVER_0_9**: CLIENT_9_0 ist funktional nur mit einem SERVER_0_9-Block auf der Server-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, QO = FALSE.
2. **Verbindungsaufbau**: INIT mit QI = TRUE wird verarbeitet.
3. **Verbunden**: INITO mit QO = TRUE bestätigt die Verbindung.
4. **Datenaustausch**: REQ/CNF-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: INIT mit QI = FALSE wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 9 Werte gesendet und 0 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Client-seitige Anbindung** eines Kommunikationspartners, der bereits als SERVER_0_9 mit passender Feldanzahl ausgelegt ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- SERVER_0_9: das direkte Gegenstück auf der Server-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- CLIENT_1 / SERVER_1: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

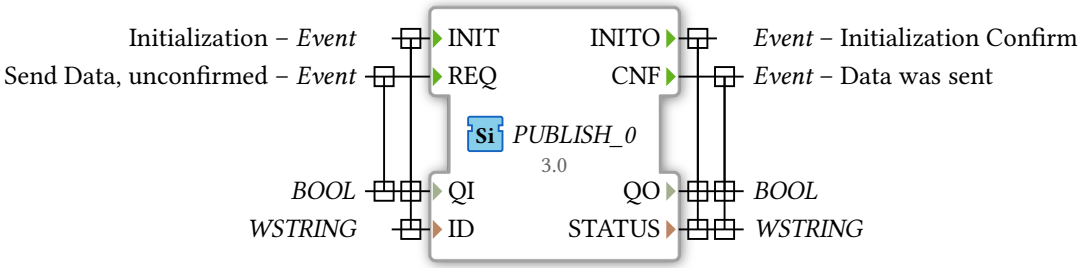
CLIENT_9_0 liefert die generische, auf 9 Sende- und 0 Empfangsfelder zugeschnittene Client-Variante der GEN_CLIENT-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.35 PUBLISH_0

• – ◦ • – ◦ • – ◦ •

Einleitung

Der PUBLISH_0-Funktionsblock dient zum Veröffentlichen von Daten für einen oder mehrere SUBSCRIBE_0-Blöcke. Er ermöglicht eine unbestätigte Kommunikation zwischen verschiedenen Komponenten in einem 4diac-System.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsereignis mit zugehörigen Daten QI und ID
- **REQ**: Sendeanforderung für Daten (unbestätigt) mit zugehörigem Daten QI

Ereignis-Ausgänge

- **INITO**: Bestätigung der Initialisierung mit zugehörigen Daten QO und STATUS
- **CNF**: Bestätigung, dass Daten gesendet wurden mit zugehörigen Daten QO und STATUS

Daten-Eingänge

- **QI** (BOOL): Qualitätsindikator für die Blockaktivierung
- **ID** (WSTRING): Identifikationsstring für die Kommunikation

Daten-Ausgänge

- **QO** (BOOL): Qualitätsindikator für die Blockausgabe
- **STATUS** (WSTRING): Statusinformationen über die ausgeführte Operation

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der PUBLISH_0-Block ermöglicht die Veröffentlichung von Daten über ein Publish-Subscribe-Muster. Bei Initialisierung (INIT) wird der Block mit einer spezifischen ID konfiguriert. Bei einer Sendeanforderung (REQ) werden die Daten an alle registrierten Subscriber verteilt. Die Kommunikation erfolgt unbestätigt, was bedeutet, dass keine Rückmeldung von den Empfängern erwartet wird.

Technische Besonderheiten

- Verwendet WSTRING-Datentypen für ID und STATUS für internationale Zeichenunterstützung
- Implementiert unbestätigte Kommunikation (unconfirmed)
- Unterstützt multiple Subscriber über eine einzige Publish-Instanz
- Generische Implementierung als GEN_PUBLISH

Zustandsübersicht

Der Block durchläuft folgende Hauptzustände:

1. **Nicht initialisiert**: Block ist betriebsbereit, aber nicht konfiguriert
2. **Initialisiert**: Block ist konfiguriert und bereit zum Senden
3. **Sendebereit**: Block kann Daten an Subscriber verteilen

Anwendungsszenarien

- Verteilung von Sensordaten an mehrere Verarbeitungsknoten
- Broadcast von Systemstatusinformationen
- Verteilung von Steuerbefehlen an multiple Aktoren
- Datenaustausch in verteilten Steuerungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu bestätigten Kommunikationsblöcken bietet PUBLISH_0:

- Höhere Performance durch unbestätigte Übertragung
- Geringere Latenzzeiten
- Einfacherere Implementierung
- Keine Garantie der Zustellung

Fazit

Der PUBLISH_0-Block ist ein effizientes Werkzeug für unbestätigte Kommunikation in verteilten Steuerungssystemen. Seine einfache Schnittstelle und die Unterstützung für multiple Empfänger machen ihn ideal für Szenarien, in denen Performance wichtiger ist als garantierte Zustellung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

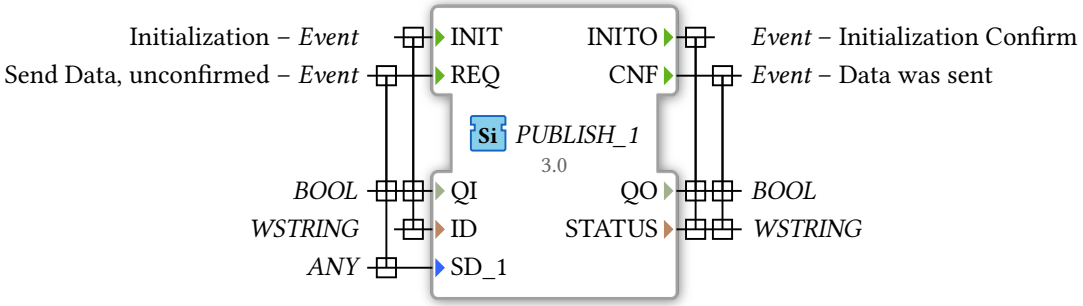
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.3.36 PUBLISH_1

• – ◦ • – ◦ • – ◦ •

Einleitung

Der PUBLISH_1-Funktionsblock dient zum Senden von Daten an einen oder mehrere SUBSCRIBE_1-Blöcke. Er implementiert ein Publish-Subscribe-Kommunikationsmuster, bei dem Daten unbestätigt an alle angemeldeten Empfänger verteilt werden.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsereignis mit zugehörigen Daten QI und ID
- **REQ**: Sendeanfrage für Daten (unbestätigt) mit zugehörigen Daten QI und SD_1

Ereignis-Ausgänge

- **INITO**: Bestätigung der Initialisierung mit zugehörigen Daten QO und STATUS
- **CNF**: Bestätigung, dass Daten gesendet wurden mit zugehörigen Daten QO und STATUS

Daten-Eingänge

- **QI** (BOOL): Qualifizierer für Initialisierung und Betrieb
- **ID** (WSTRING): Identifikator für den Publish-Kanal
- **SD_1** (ANY): Zu sendende Daten (beliebiger Datentyp)

Daten-Ausgänge

- **QO** (BOOL): Qualifizierer für Ausgangsstatus
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der PUBLISH_1-Block ermöglicht die Verteilung von Daten an mehrere Empfänger über einen Publish-Subscribe-Mechanismus. Bei Initialisierung (INIT-Ereignis) wird der Block mit einer spezifischen ID konfiguriert. Bei Empfang eines REQ-Ereignisses werden die an SD_1 übergebenen Daten an alle SUBSCRIBE_1-Blöcke mit derselben ID gesendet. Die Kommunikation erfolgt unbestätigt, d.h. es wird keine Rückmeldung von den Empfängern erwartet.

Technische Besonderheiten

- Verwendung des ANY-Datentyps für SD_1 ermöglicht das Senden beliebiger Datenstrukturen
- WSTRING-Datentyp für ID und STATUS unterstützt internationale Zeichensätze
- Unbestätigte Kommunikation für reduzierte Latenz
- Generische Implementierung durch GenericClassName-Attribut

Zustandsübersicht

1. **Nicht initialisiert**: Block wartet auf INIT-Ereignis
2. **Initialisiert**: Block bereit zum Empfang von REQ-Ereignissen
3. **Sendeaktiv**: Verarbeitung von REQ-Ereignissen und Datenverteilung

Anwendungsszenarien

- Verteilung von Sensordaten an mehrere Verarbeitungsknoten
- Broadcast von Systemstatusinformationen
- Verteilen von Konfigurationsdaten an mehrere Steuerungen
- Multi-Cast-Kommunikation in verteilten Automatisierungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu bestätigten Kommunikationsblöcken bietet PUBLISH_1 reduzierte Latenz durch unbestätigte Übertragung. Gegenüber Punkt-zu-Punkt-Verbindungen ermöglicht er effiziente 1-zu-n-Kommunikation ohne explizite Verbindungsaufbauten zu jedem Empfänger.

Fazit

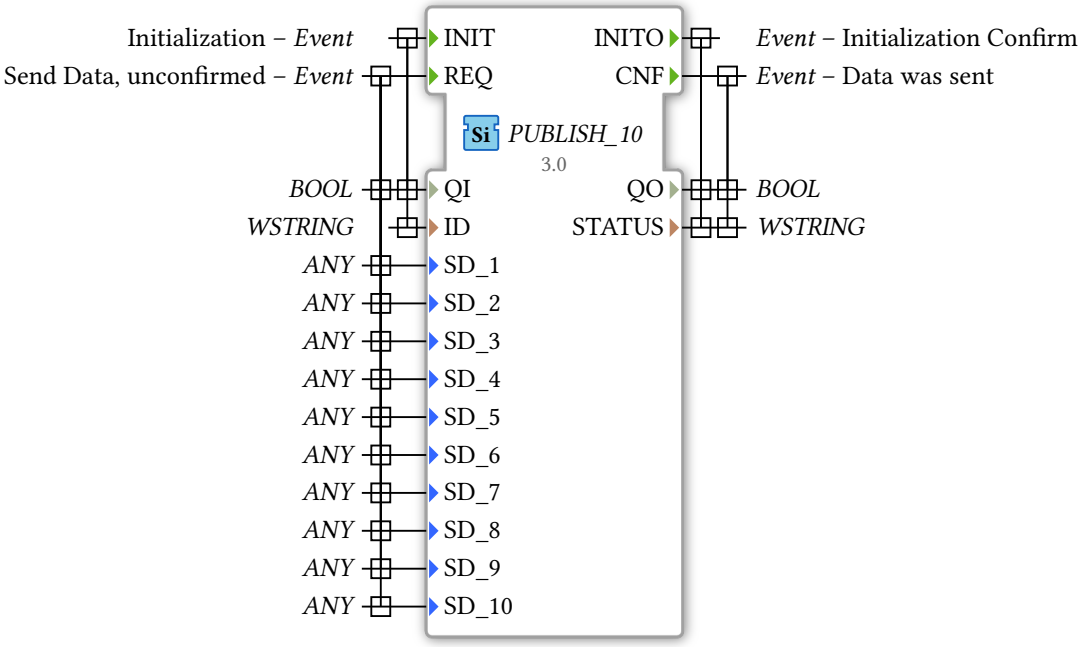
Der PUBLISH_1-Block ist ein effizientes Werkzeug für unbestätigte 1-zu-n-Kommunikation in verteilten Automatisierungssystemen. Seine flexible Datentypunterstützung und einfache Konfiguration machen ihn ideal für Szenarien, in denen Daten an mehrere Empfänger verteilt werden müssen, ohne dass Bestätigungen erforderlich sind.

2.4.3.37 PUBLISH_10

• − ◦ • − ◦ • − ◦ •

Einleitung

Der PUBLISH_10 Funktionsblock dient zum Verteilen von Daten an einen oder mehrere SUBSCRIBE_10 Blöcke. Er ermöglicht die unbestätigte Übertragung von bis zu 10 verschiedenen Datenwerten über ein Publish-Subscribe-Kommunikationsmuster.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsereignis mit zugehörigen Daten QI und ID
- **REQ**: Sendeanforderung für unbestätigte Datenübertragung mit 10 Datenvariablen

Ereignis-Ausgänge

- **INITO**: Bestätigung der Initialisierung mit Statusinformationen
- **CNF**: Bestätigung, dass Daten gesendet wurden

Daten-Eingänge

- **QI** (BOOL): Qualifizierer für Initialisierung (TRUE = aktivieren, FALSE = deaktivieren)
- **ID** (WSTRING): Identifikationsstring für den Kommunikationskanal
- **SD_1** bis **SD_10** (ANY): 10 verschiedene zu sendende Datenwerte beliebigen Typs

Daten-Ausgänge

- **QO** (BOOL): Qualifizierer-Ausgang (TRUE = erfolgreich, FALSE = fehlgeschlagen)
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der PUBLISH_10 Block arbeitet nach dem Publish-Subscribe-Prinzip. Nach erfolgreicher Initialisierung mit dem INIT-Ereignis können über das REQ-Ereignis bis zu 10 verschiedene Datenwerte gleichzeitig an alle angemeldeten Subscriber verteilt werden. Die Datenübertragung erfolgt unbestätigt, was bedeutet, dass der Block keine Rückmeldung von den Empfängern erwartet.

Technische Besonderheiten

- Unterstützt bis zu 10 verschiedene Datenquellen gleichzeitig
- Verwendet den generischen Datentyp ANY für maximale Flexibilität
- Unicode-String-Unterstützung für ID und STATUS
- Unbestätigte Kommunikation für reduzierte Latenz
- Generische Implementierung durch GEN_PUBLISH Basisklasse

Zustandsübersicht

1. **Nicht initialisiert**: Block ist inaktiv
2. **Initialisiert**: Block ist bereit für Datenverteilung
3. **Sendebereit**: Verarbeitet REQ-Ereignisse und verteilt Daten

Anwendungsszenarien

- Verteilung von Sensordaten an mehrere Verbraucher
- Broadcast-Kommunikation in verteilten Systemen
- Datenverteilung in Echtzeit-Steuerungssystemen
- Multi-Consumer Datenpipeline-Architekturen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu bestätigten Kommunikationsblöcken bietet PUBLISH_10 reduzierte Latenz durch unbestätigte Übertragung. Gegenüber Blöcken mit weniger Datenkanälen ermöglicht er die gleichzeitige Verteilung mehrerer Datenströme.

Fazit

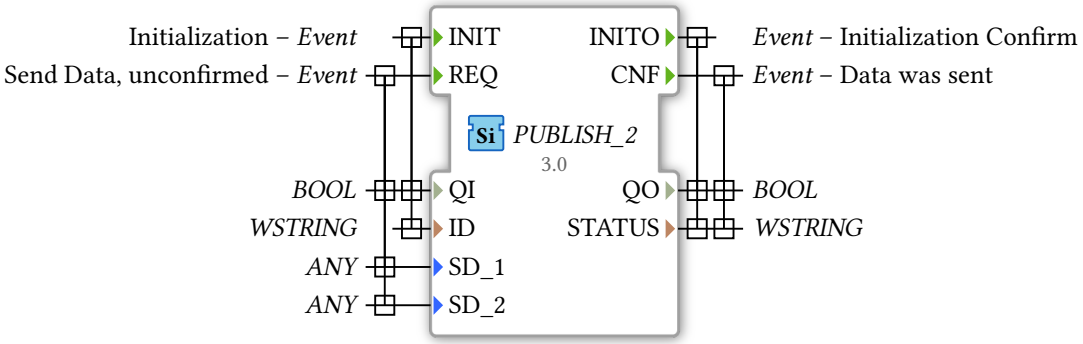
Der PUBLISH_10 Funktionsblock ist eine leistungsstarke Lösung für unbestätigte Multi-Daten-Publish-Subscribe-Kommunikation in IEC 61499-Systemen. Seine Flexibilität bei der Datenaufnahme und die Unterstützung für bis zu 10 parallele Datenkanäle machen ihn ideal für komplexe Datenverteilungsaufgaben in industriellen Automatisierungssystemen.

2.4.3.38 PUBLISH_2

• – ◦ • – ◦ • – ◦ •

Einleitung

Der PUBLISH_2-Funktionsblock dient zum Senden von Daten an einen oder mehrere SUBSCRIBE_2-Blöcke. Er ermöglicht eine unbestätigte Kommunikation zwischen verschiedenen Komponenten in einem verteilten System.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsereignis mit den zugehörigen Daten QI und ID
- **REQ**: Anforderungsereignis zum Senden von Daten mit den zugehörigen Daten QI, SD_1 und SD_2

Ereignis-Ausgänge

- **INITO**: Bestätigung der Initialisierung mit den zugehörigen Daten QO und STATUS
- **CNF**: Bestätigung, dass Daten gesendet wurden, mit den zugehörigen Daten QO und STATUS

Daten-Eingänge

- **QI** (BOOL): Qualifizierer für Initialisierung und Betrieb
- **ID** (WSTRING): Identifikator für die Kommunikation
- **SD_1** (ANY): Erster zu sendender Datenwert (beliebiger Datentyp)
- **SD_2** (ANY): Zweiter zu sendender Datenwert (beliebiger Datentyp)

Daten-Ausgänge

- **QO** (BOOL): Qualifizierer für den Ausgangszustand
- **STATUS** (WSTRING): Statusinformationen über den Betriebszustand

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der PUBLISH_2-Block initialisiert sich über das INIT-Ereignis und sendet anschließend Daten über das REQ-Ereignis. Die Daten SD_1 und SD_2 werden an alle angeschlossenen SUBSCRIBE_2-Blöcke verteilt. Die Kommunikation erfolgt unbestätigt, was bedeutet, dass keine Rückmeldung von den Empfängern erwartet wird.

Technische Besonderheiten

- Unterstützt beliebige Datentypen für SD_1 und SD_2
- Verwendet WSTRING für ID und STATUS für internationale Zeichenunterstützung
- Unbestätigter Kommunikationsmodus (Publish/Subscribe-Pattern)
- Flexible Datenübertragung mit zwei separaten Datenkanälen

Zustandsübersicht

1. **Nicht initialisiert**: Block wartet auf INIT-Ereignis
2. **Initialisiert**: Block ist betriebsbereit und kann REQ-Ereignisse verarbeiten
3. **Sendebereit**: Verarbeitet REQ-Ereignisse und sendet Daten an Subscriber

Anwendungsszenarien

- Verteilte Systeme mit mehreren Datenempfängern
- Echtzeit-Datenverteilung ohne Bestätigungsmechanismus
- Systeme mit variablen Datentypen
- Kommunikation zwischen verschiedenen Steuerungskomponenten

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu bestätigten Kommunikationsblöcken bietet PUBLISH_2 eine höhere Performance bei gleichzeitig geringerer Zuverlässigkeit. Die Unterstützung für beliebige Datentypen macht ihn flexibler als typspezifische Kommunikationsblöcke.

Fazit

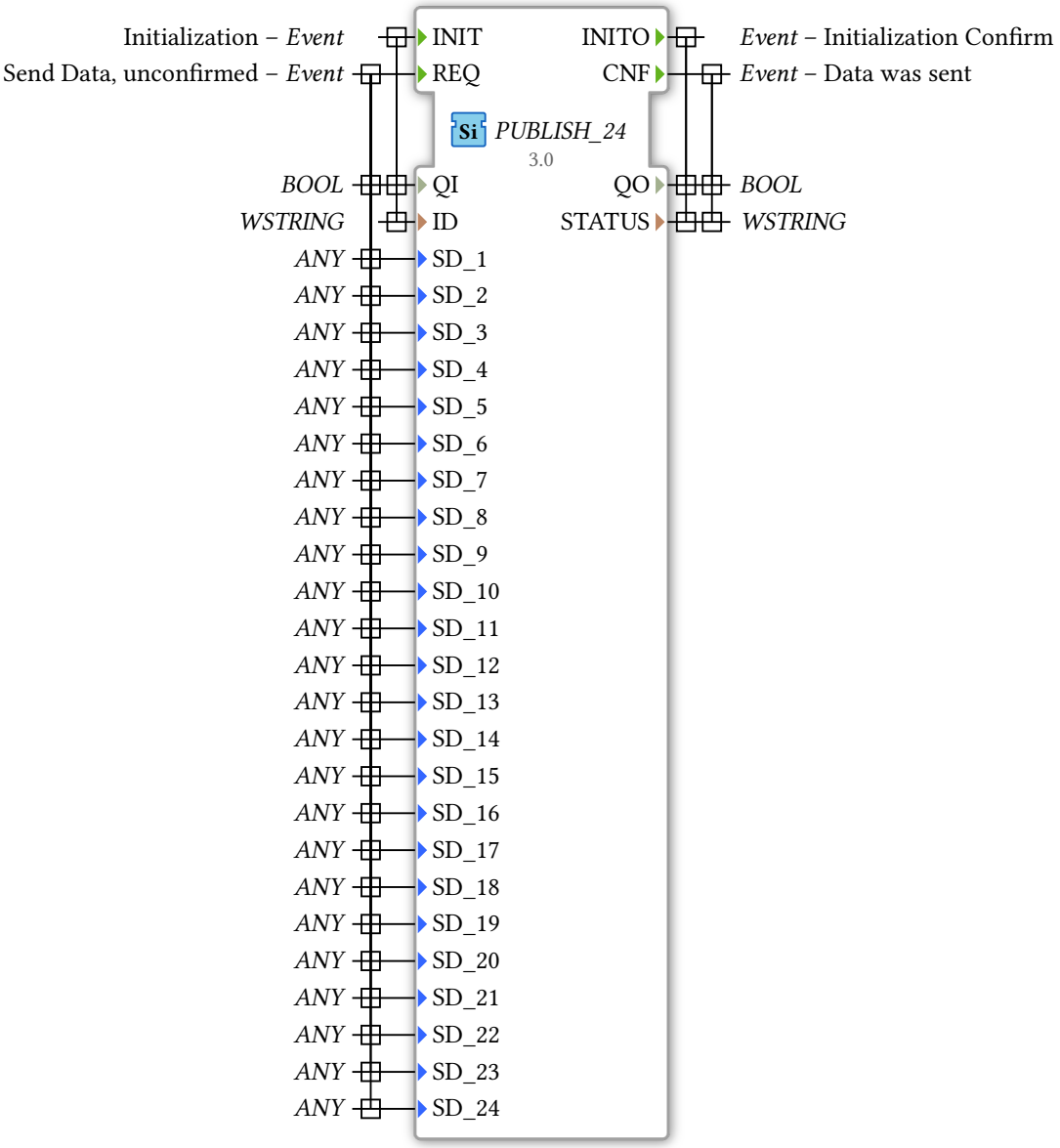
Der PUBLISH_2-Block ist ein effizientes Werkzeug für unbestätigte Datenverteilung in verteilten Systemen. Seine Flexibilität bei der Datentypunterstützung und die einfache Handhabung machen ihn ideal für Szenarien, in denen Performance wichtiger ist als garantierte Zustellung.

2.4.3.39 PUBLISH_24

• − ◦ • − ◦ • − ◦ •

Einleitung

Der Funktionsblock `PUBLISH_24` verteilt Daten an einen oder mehrere `SUBSCRIBE_24`-Blöcke. Er ermöglicht die unbestätigte Übertragung von bis zu 24 Datenwerten über ein Publish-Subscribe-Kommunikationsmuster.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierung. Verknüpft mit `QI`, `ID`.
- **REQ**: Daten senden, unbestätigt. Verknüpft mit `QI`, `SD_1` bis `SD_24`.

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung. Verknüpft mit `QO`, `STATUS`.
- **CNF**: Daten wurden gesendet. Verknüpft mit `QO`, `STATUS`.

Daten-Eingänge

- **QI** (`BOOL`): Ereignis-Eingangsqualifizierer
- **ID** (`WSTRING`): Identifikationsstring des Kommunikationskanals
- **SD_1** bis **SD_24** (`ANY`): zu sendende Datenwerte beliebigen Typs

Daten-Ausgänge

- **QO** (`BOOL`): Ereignis-Ausgangsqualifizierer
- **STATUS** (`WSTRING`): Statusinformation

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Nach der Initialisierung mit `INIT` werden mit `REQ` bis zu 24 Datenwerte an alle angemeldeten Subscriber verteilt. Die Übertragung ist unbestätigt; `CNF` meldet nur, dass die Daten gesendet wurden.

Technische Besonderheiten

- Generischer Baustein (Generic Class `GEN_PUBLISH`): Die Anzahl der Datenports ist hier 24, ihre Typen (`ANY`) werden bei der Instanziierung festgelegt
- Unicode-Strings (`WSTRING`) für `ID` und `STATUS`
- Gegenstück: `SUBSCRIBE_24` mit `RD_1` bis `RD_24`

Anwendungsszenarien

- Verteilung vieler zusammengehöriger Werte (z. B. Messwerte oder Zustände) mit einem einzigen Ereignis an andere Applikationen oder Geräte

⚖️ Vergleich mit ähnlichen Bausteinen

`PUBLISH_10` überträgt bis zu 10 Datenwerte, `PUBLISH_24` bis zu 24.

Fazit

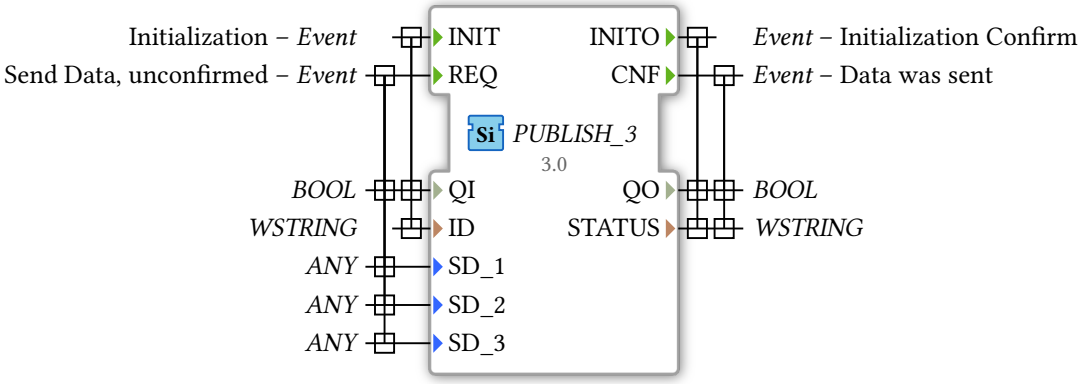
`PUBLISH_24` ist die unbestätigte Publish-Variante für bis zu 24 Datenwerte.

2.4.3.40 PUBLISH_3

• – ◦ • – ◦ • – ◦ •

Einleitung

Der PUBLISH_3-Funktionsblock dient zum Verteilen von Daten an einen oder mehrere SUBSCRIBE_3-Blöcke. Er ermöglicht die unbestätigte Übertragung von bis zu drei verschiedenen Datenwerten an Abonnenten in einem verteilten System.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsereignis mit den zugehörigen Daten QI und ID
- **REQ**: Sendeanforderung für Daten (unbestätigt) mit den zugehörigen Daten QI, SD_1, SD_2 und SD_3

Ereignis-Ausgänge

- **INITO**: Bestätigung der Initialisierung mit den zugehörigen Daten QO und STATUS
- **CNF**: Bestätigung, dass Daten gesendet wurden, mit den zugehörigen Daten QO und STATUS

Daten-Eingänge

- **QI** (BOOL): Qualifizierer für Initialisierung und Sendeoperationen
- **ID** (WSTRING): Identifikator für den Publisher
- **SD_1** (ANY): Erster zu sendender Datenwert (beliebiger Datentyp)

- **SD_2** (ANY): Zweiter zu sendender Datenwert (beliebiger Datentyp)
- **SD_3** (ANY): Dritter zu sendender Datenwert (beliebiger Datentyp)

Daten-Ausgänge

- **QO** (BOOL): Qualifizierer für Ausgangsereignisse
- **STATUS** (WSTRING): Statusinformationen über die ausgeführte Operation

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der PUBLISH_3-Block wird zunächst über das INIT-Ereignis initialisiert, wobei der ID-Parameter den eindeutigen Identifikator für diesen Publisher festlegt. Nach erfolgreicher Initialisierung bestätigt der Block mit INITO. Daten können anschließend über das REQ-Ereignis gesendet werden, wobei bis zu drei verschiedene Datenwerte (SD_1, SD_2, SD_3) gleichzeitig übertragen werden können. Die Übertragung erfolgt unbestätigt, jedoch gibt der Block nach dem Senden ein CNF-Ereignis aus.

Technische Besonderheiten

- Unterstützt den ANY-Datentyp für alle drei Datenkanäle, was maximale Flexibilität bei den zu übertragenden Datentypen bietet
- Verwendet WSTRING für ID und STATUS, ermöglicht somit internationale Zeichensätze
- Unbestätigte Kommunikation für reduzierte Latenz
- Kann Daten an mehrere Subscriber gleichzeitig verteilen

Zustandsübersicht

1. **Nicht initialisiert**: Block ist betriebsbereit, aber nicht für den Datenaustausch konfiguriert
2. **Initialisiert**: Block ist konfiguriert und bereit zum Senden von Daten
3. **Sendend**: Block verarbeitet eine Sendeanforderung
4. **Bereit**: Block hat Daten erfolgreich gesendet und ist bereit für weitere Anforderungen

Anwendungsszenarien

- Verteilung von Sensordaten an mehrere Verarbeitungsknoten
- Broadcast von Steuerinformationen in verteilten Systemen
- Datenverteilung in Produktionsanlagen mit mehreren Verbrauchern
- Messwertverteilung in Überwachungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren PUBLISH-Blöcken bietet PUBLISH_3 die Möglichkeit, bis zu drei verschiedene Datenwerte gleichzeitig zu übertragen, was die Effizienz in Systemen mit mehreren zusammenhängenden Datenpunkten erhöht. Die Verwendung von ANY-Datentypen bietet mehr Flexibilität als typspezifische Implementierungen.

Fazit

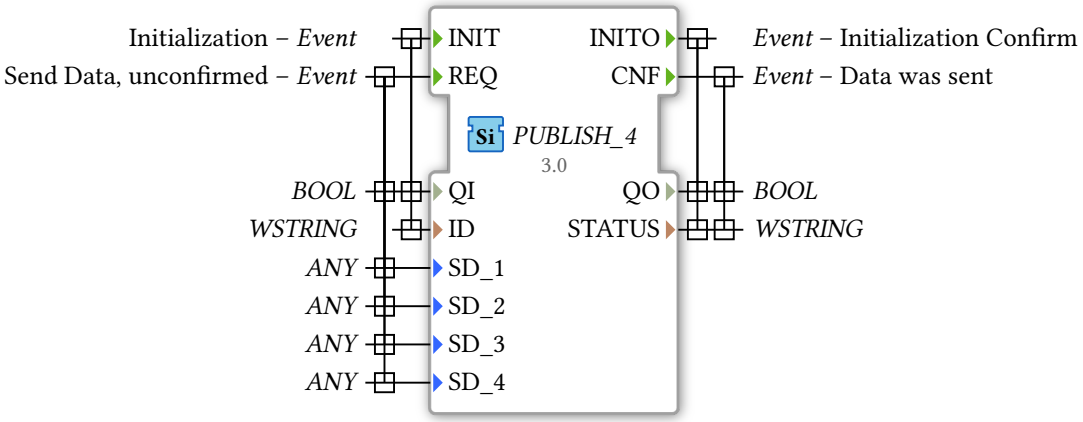
Der PUBLISH_3-Funktionsblock ist eine leistungsstarke Lösung für die Verteilung von Daten in verteilten Automatisierungssystemen. Seine Fähigkeit, bis zu drei verschiedene Datenwerte gleichzeitig zu übertragen, kombiniert mit der Flexibilität der ANY-Datentypen, macht ihn besonders geeignet für komplexe Anwendungen, bei denen mehrere zusammenhängende Informationen effizient verteilt werden müssen.

2.4.3.41 PUBLISH_4

• – ◦ • – ◦ • – ◦ •

Einleitung

Der PUBLISH_4 Funktionsblock dient zur Veröffentlichung von Daten an einen oder mehrere SUBSCRIBE_4 Blöcke. Er ermöglicht die unbestätigte Übertragung von bis zu vier verschiedenen Datenwerten über ein Publish-Subscribe-Kommunikationsmuster.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsereignis mit zugehörigen Daten QI und ID
- **REQ**: Sendeanforderung für Daten (unbestätigt) mit zugehörigen Daten QI, SD_1, SD_2, SD_3 und SD_4

Ereignis-Ausgänge

- **INITO**: Bestätigung der Initialisierung mit zugehörigen Daten QO und STATUS
- **CNF**: Bestätigung, dass Daten gesendet wurden, mit zugehörigen Daten QO und STATUS

Daten-Eingänge

- **QI** (BOOL): Qualifizierer für Initialisierung und Sendeoperationen
- **ID** (WSTRING): Identifikationsstring für die Kommunikation
- **SD_1** (ANY): Erster zu sendender Datenwert (beliebiger Datentyp)
- **SD_2** (ANY): Zweiter zu sendender Datenwert (beliebiger Datentyp)
- **SD_3** (ANY): Dritter zu sendender Datenwert (beliebiger Datentyp)
- **SD_4** (ANY): Vierter zu sendender Datenwert (beliebiger Datentyp)

Daten-Ausgänge

- **QO** (BOOL): Qualifizierer für Ausgangsereignisse
- **STATUS** (WSTRING): Statusinformationen als Wide-String

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der PUBLISH_4 Block initialisiert sich über das INIT-Ereignis und bestätigt dies mit INITO. Nach erfolgreicher Initialisierung können über das REQ-Ereignis bis zu vier verschiedene Datenwerte (SD_1 bis SD_4) gleichzeitig an alle verbundenen SUBSCRIBE_4 Blöcke gesendet werden. Die Datenübertragung erfolgt unbestätigt, wobei der Block lediglich bestätigt, dass die Daten gesendet wurden (CNF), nicht jedoch, dass sie empfangen wurden.

Technische Besonderheiten

- Unterstützt beliebige Datentypen (ANY) für alle vier Datenkanäle
- Verwendet Wide-Strings (WSTRING) für ID und STATUS
- Unbestätigter Sendemodus (fire-and-forget)
- Maximale Kapazität von vier gleichzeitigen Datenwerten
- Generische Implementierung als GEN_PUBLISH

Zustandsübersicht

1. **Nicht initialisiert**: Block wartet auf INIT-Ereignis
2. **Initialisiert**: Block bereit zum Empfang von REQ-Ereignissen
3. **Sendeaktiv**: Verarbeitung von REQ-Ereignissen und Auslösung von CNF

Anwendungsszenarien

- Verteilte Systeme mit Publish-Subscribe-Architektur
- Echtzeit-Datenverteilung an mehrere Empfänger
- Systeme mit unidirektionaler Datenkommunikation
- Anwendungen, die gleichzeitige Übertragung mehrerer Datenwerte benötigen

⚖️ **Vergleich mit ähnlichen Bausteinen**

Im Vergleich zu bestätigten Kommunikationsblöcken bietet PUBLISH_4 geringere Latenz durch den unbestätigten Sendemodus. Gegenüber Blöcken mit weniger Datenkanälen ermöglicht er die gleichzeitige Übertragung von bis zu vier verschiedenen Datenwerten.

Fazit

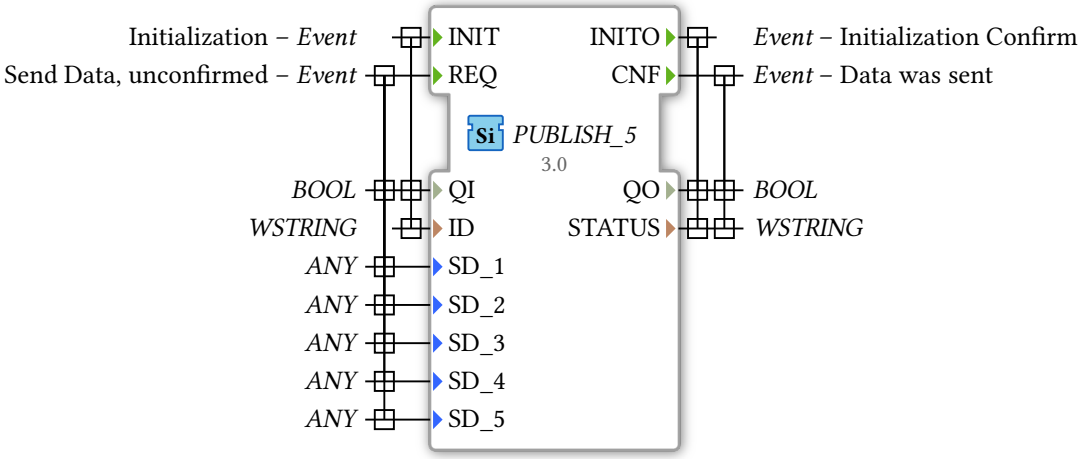
Der PUBLISH_4 Funktionsblock ist eine effiziente Lösung für unbestätigte Datenverteilung in verteilten Systemen. Seine Flexibilität bei der Unterstützung beliebiger Datentypen und die Möglichkeit, vier Datenwerte parallel zu senden, machen ihn besonders geeignet für komplexe Kommunikationsszenarien in industriellen Automatisierungssystemen.

2.4.3.42 PUBLISH_5

• – ◦ • – ◦ • – ◦ •

Einleitung

Der PUBLISH_5 Funktionsblock dient zum Veröffentlichen von Daten an einen oder mehrere SUBSCRIBE_5 Blöcke. Er ermöglicht die Übertragung von bis zu fünf verschiedenen Datenwerten über ein Publish-Subscribe-Kommunikationsmuster.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsereignis mit zugehörigen Daten QI und ID
- **REQ**: Anforderungsereignis zum Senden von Daten (unbestätigt) mit zugehörigen Daten QI, SD_1 bis SD_5

Ereignis-Ausgänge

- **INITO**: Bestätigung der Initialisierung mit zugehörigen Daten QO und STATUS
- **CNF**: Bestätigung, dass Daten gesendet wurden mit zugehörigen Daten QO und STATUS

Daten-Eingänge

- **QI** (BOOL): Qualifizierer für Initialisierung/Operation
- **ID** (WSTRING): Identifikator für den Publish-Kanal
- **SD_1** (ANY): Erster zu sendender Datenwert
- **SD_2** (ANY): Zweiter zu sendender Datenwert
- **SD_3** (ANY): Dritter zu sendender Datenwert

- **SD_4** (ANY): Vierter zu sendender Datenwert
- **SD_5** (ANY): Fünfter zu sendender Datenwert

Daten-Ausgänge

- **QO** (BOOL): Qualifizierer für Ausgangsstatus
- **STATUS** (WSTRING): Statusinformation über die Operation

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der PUBLISH_5 Block initialisiert sich über das INIT-Ereignis und konfiguriert den Publish-Kanal mit der angegebenen ID. Nach erfolgreicher Initialisierung kann über das REQ-Ereignis Daten an alle verbundenen SUBSCRIBE_5 Blöcke gesendet werden. Die Daten SD_1 bis SD_5 werden dabei gleichzeitig übertragen. Jede Sendung wird durch ein CNF-Ereignis bestätigt.

Technische Besonderheiten

- Unterstützt den Datentyp ANY für maximale Flexibilität bei den zu sendenden Daten
- Verwendet WSTRING für Statusmeldungen und Kanal-IDs
- Implementiert ein unbestätigtes Sendeverfahren (unconfirmed)
- Bietet Platz für bis zu fünf verschiedene Datenwerte pro Sendevorgang

Zustandsübersicht

1. **Nicht initialisiert**: Block wartet auf INIT-Ereignis
2. **Initialisiert**: Block ist bereit für REQ-Ereignisse
3. **Senden**: Verarbeitet REQ und sendet Daten an Subscriber
4. **Bestätigen**: Sendet CNF nach erfolgreicher Datenübertragung

Anwendungsszenarien

- Verteilte Systeme mit Publisher-Subscriber-Architektur
- Datenverteilung an mehrere Empfänger in Echtzeitsystemen
- Flexibles Messaging zwischen verschiedenen Steuerungskomponenten
- Systeme mit variablen Datenstrukturen (durch ANY-Typ Unterstützung)

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren PUBLISH-Blöcken bietet PUBLISH_5 die Möglichkeit, bis zu fünf verschiedene Datenwerte gleichzeitig zu senden. Die Verwendung des ANY-Datentyps ermöglicht größere Flexibilität als typspezifische Publish-Blöcke.

Fazit

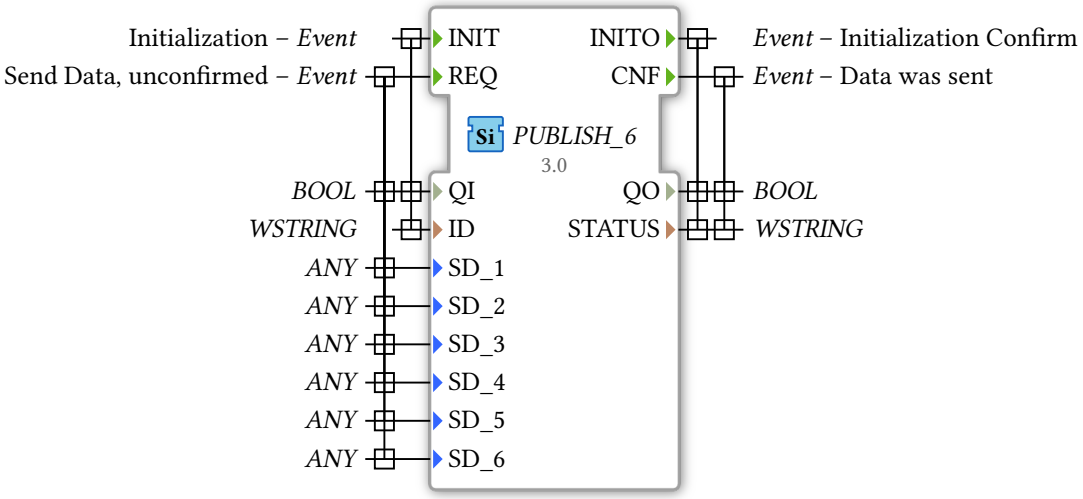
Der PUBLISH_5 Funktionsblock ist eine leistungsfähige Lösung für Publish-Subscribe-Kommunikation in verteilten Automatisierungssystemen. Seine Flexibilität durch den ANY-Datentyp und die Möglichkeit, mehrere Datenwerte gleichzeitig zu senden, machen ihn besonders geeignet für komplexe Datenverteilungsaufgaben in industriellen Steuerungssystemen.

2.4.3.43 PUBLISH_6

• – ◦ • – ◦ • – ◦ •

Einleitung

Der PUBLISH_6 Funktionsblock dient zum Veröffentlichen von Daten an einen oder mehrere SUBSCRIBE_6 Blöcke. Er ermöglicht die Übertragung von bis zu sechs verschiedenen Datenwerten über ein Publish-Subscribe-Kommunikationsmuster und bietet sowohl bestätigte als auch unbestätigte Sendeoperationen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsereignis mit zugehörigen Daten QI und ID
- **REQ**: Sendeanforderung für Daten (unbestätigt) mit zugehörigen Daten QI und SD_1 bis SD_6

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung mit zugehörigen Daten QO und STATUS
- **CNF**: Bestätigung, dass Daten gesendet wurden, mit zugehörigen Daten QO und STATUS

Daten-Eingänge

- **QI** (BOOL): Qualitätsindikator für Initialisierung und Sendeoperationen
- **ID** (WSTRING): Identifikator für den Kommunikationskanal
- **SD_1** bis **SD_6** (ANY): Sechs verschiedene zu sendende Datenwerte beliebigen Typs

Daten-Ausgänge

- **QO** (BOOL): Qualitätsindikator für Ausgangsoperationen
- **STATUS** (WSTRING): Statusinformation über die ausgeführte Operation

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der PUBLISH_6 Block initialisiert sich über das INIT-Ereignis und bestätigt die Initialisierung mit INITO. Daten können über das REQ-Ereignis gesendet werden, wobei bis zu sechs verschiedene Datenwerte (SD_1 bis SD_6) gleichzeitig übertragen werden können. Nach erfolgreichem Senden der Daten wird ein CNF-Ereignis ausgegeben. Der Block unterstützt das Publish-Subscribe-Pattern, bei dem mehrere Subscriber die gesendeten Daten empfangen können.

Technische Besonderheiten

- Unterstützung für bis zu sechs verschiedene Datenwerte beliebigen Typs (ANY)
- Verwendung von WSTRING für Statusmeldungen und Kanal-Identifikation
- Unbestätigte Sendeoperationen (REQ) mit nachfolgender Bestätigung (CNF)
- Generische Implementierung für flexible Wiederverwendung

Zustandsübersicht

1. **Nicht initialisiert**: Block wartet auf INIT-Ereignis
2. **Initialisiert**: Block bereit zum Empfang von REQ-Ereignissen
3. **Sendeoperation**: Verarbeitung der Sendeanforderung und Datenübertragung
4. **Bestätigung**: Ausgabe von CNF nach erfolgreichem Senden

Anwendungsszenarien

- Verteilung von Sensordaten an mehrere Verarbeitungsknoten
- Broadcast-Kommunikation in verteilten Steuerungssystemen
- Datenverteilung in Produktionsanlagen mit mehreren Verbrauchern
- Flexibles Messaging in IoT-Anwendungen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren Publish-Blöcken bietet PUBLISH_6 die Möglichkeit, bis zu sechs verschiedene Datenwerte gleichzeitig zu übertragen, was die Effizienz bei der Übertragung zusammengehöriger Daten erhöht. Die Verwendung des ANY-Datentyps ermöglicht maximale Flexibilität bei den zu übertragenden Datenformaten.

Fazit

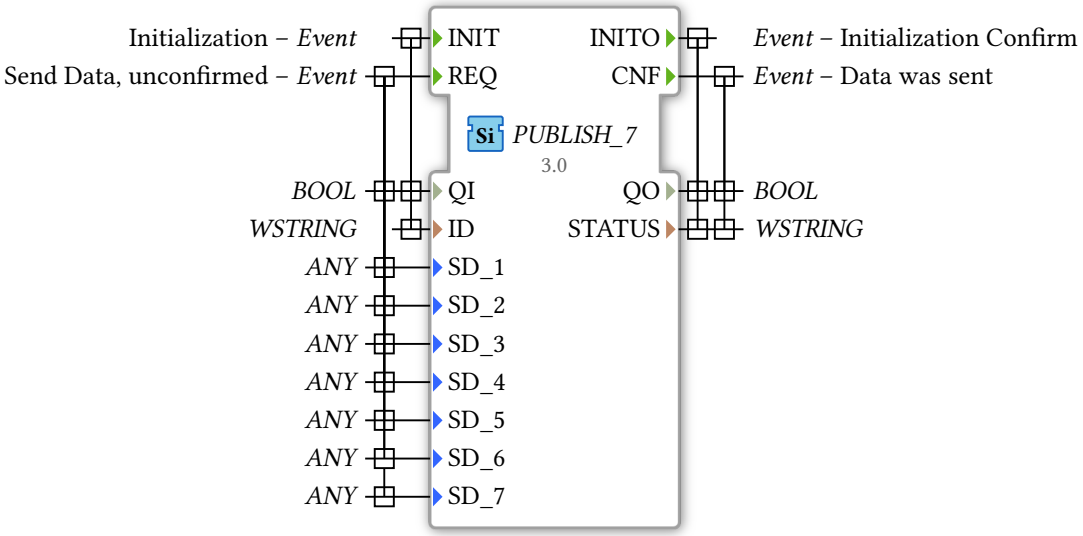
Der PUBLISH_6 Funktionsblock ist ein leistungsfähiges Werkzeug für Publish-Subscribe-Kommunikation in verteilten Automatisierungssystemen. Seine Fähigkeit, mehrere Datenwerte gleichzeitig zu übertragen, macht ihn besonders effizient für Anwendungen, bei denen zusammenhängende Datensätze verteilt werden müssen. Die flexible Typunterstützung und robuste Fehlerbehandlung durch Statusmeldungen machen ihn zu einer zuverlässigen Lösung für industrielle Kommunikationsanforderungen.

2.4.3.44 PUBLISH_7

• – ◦ • – ◦ • – ◦ •

Einleitung

Der PUBLISH_7 Funktionsblock dient zum Verteilen von Daten an einen oder mehrere SUBSCRIBE_7 Blöcke. Er ermöglicht die unbestätigte Übertragung von bis zu sieben verschiedenen Datenwerten über eine Publish-Subscribe-Architektur.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsereignis mit zugehörigen Daten QI und ID
- **REQ**: Sendeanforderung für Daten (unbestätigt) mit sieben Datenvariablen

Ereignis-Ausgänge

- **INITO**: Bestätigung der Initialisierung mit QO und STATUS
- **CNF**: Bestätigung, dass Daten gesendet wurden mit QO und STATUS

Daten-Eingänge

- **QI** (BOOL): Qualitätsindikator für Initialisierung
- **ID** (WSTRING): Identifikator für den Publish-Kanal

- **SD_1** bis **SD_7** (ANY): Sieben verschiedene Datenwerte beliebigen Typs

Daten-Ausgänge

- **QO** (BOOL): Qualitätsindikator für Ausgangszustand
- **STATUS** (WSTRING): Statusinformationen als Unicode-String

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der PUBLISH_7 Block initialisiert sich über das INIT-Ereignis mit einem spezifischen Kanal-ID. Nach erfolgreicher Initialisierung können über das REQ-Ereignis bis zu sieben Datenwerte (SD_1 bis SD_7) gleichzeitig an alle angeschlossenen SUBSCRIBE_7 Blöcke verteilt werden. Die Datenübertragung erfolgt unbestätigt, wobei der CNF-Ausgang lediglich bestätigt, dass die Daten gesendet wurden, nicht jedoch, dass sie empfangen wurden.

Technische Besonderheiten

- Unterstützt den ANY-Datentyp für maximale Flexibilität
- Verwendet WSTRING für ID und STATUS für internationale Zeichensatzunterstützung
- Bietet sieben unabhängige Datenkanäle
- Implementiert nach IEC 61499-2 Standard
- Verwendet GenericClassName ‚GEN_PUBLISH' für generische Implementierung

Zustandsübersicht

1. **Nicht initialisiert**: Block wartet auf INIT-Ereignis
2. **Initialisiert**: Block bereit für REQ-Ereignisse
3. **Sendebereit**: Verarbeitet REQ-Ereignisse und verteilt Daten

Anwendungsszenarien

- Verteilung von Sensordaten an mehrere Verbraucher
- Broadcast-Kommunikation in verteilten Systemen
- Datenverteilung in Produktionsanlagen
- Messwertverteilung in Überwachungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren PUBLISH-Blöcken bietet PUBLISH_7 die Möglichkeit, sieben verschiedene Datenwerte parallel zu verteilen, während einfachere Varianten typischerweise nur einen oder wenige Datenkanäle unterstützen.

Fazit

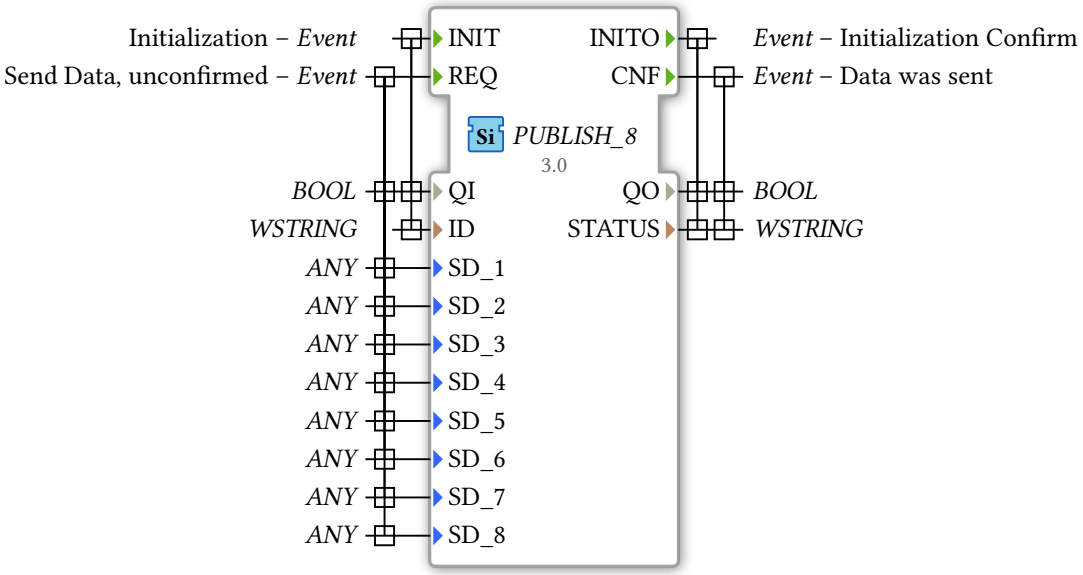
Der PUBLISH_7 Funktionsblock ist eine leistungsstarke Lösung für Publish-Subscribe-Kommunikation in IEC 61499-Systemen, die durch ihre sieben parallelen Datenkanäle und Flexibilität bei den Datentypen besonders für komplexe Datenverteilungsaufgaben geeignet ist.

2.4.3.45 PUBLISH_8

• – ◦ • – ◦ • – ◦ •

Einleitung

Der PUBLISH_8 Funktionsblock dient zum Senden von Daten an einen oder mehrere SUBSCRIBE_8 Blöcke. Er implementiert ein Publisher-Subscriber-Kommunikationsmuster und ermöglicht die Verteilung von bis zu acht verschiedenen Datenwerten an abonnierte Empfänger.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsereignis mit zugehörigen Daten QI und ID
- **REQ**: Sendeanforderung für Daten (unbestätigt) mit allen acht SD_x Datenwerten

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung mit QO und STATUS
- **CNF**: Bestätigung, dass Daten gesendet wurden, mit QO und STATUS

Daten-Eingänge

- **QI** (BOOL): Qualifizierer für Initialisierung
- **ID** (WSTRING): Identifikator für die Kommunikation
- **SD_1** bis **SD_8** (ANY): Acht verschiedene Sendedaten beliebigen Typs

Daten-Ausgänge

- **QO** (BOOL): Qualifizierer für Ausgangszustand
- **STATUS** (WSTRING): Statusinformationen

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der PUBLISH_8 Block ermöglicht die Verteilung von Daten über ein Publisher-Subscriber-Modell. Bei Empfang eines REQ-Ereignisses werden alle acht SD_x Datenwerte an alle abonnierten SUBSCRIBE_8 Blöcke gesendet. Die INIT-Initialisierung konfiguriert den Block für die Kommunikation, wobei der ID-Parameter zur Identifikation des Publishers dient.

Technische Besonderheiten

- Unterstützt bis zu acht verschiedene Datenwerte beliebigen Typs (ANY)
- Verwendet Wide-String (WSTRING) für ID und STATUS für internationale Zeichenunterstützung
- Implementiert unbestätigte Kommunikation (REQ ohne direkte Empfangsbestätigung)
- Generische Implementierung basierend auf GEN_PUBLISH

Zustandsübersicht

Der Block verfügt über zwei Hauptzustände:

1. **Nicht initialisiert**: Block ist betriebsbereit, aber nicht für Kommunikation konfiguriert
2. **Initialisiert**: Block ist konfiguriert und kann Daten an Subscriber verteilen

Anwendungsszenarien

- Verteilung von Prozessdaten an mehrere Anzeige- oder Steuerungssysteme
- Implementierung von Daten-Broadcasting in verteilten Systemen
- Einsatz in SPS-Systemen für Maschine-zu-Maschine-Kommunikation
- Verwendung in Industrie-4.0-Anwendungen für flexible Datenverteilung

⚖ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren PUBLISH-Blöcken bietet PUBLISH_8 die Möglichkeit, bis zu acht verschiedene Datenwerte parallel zu verteilen, was eine höhere Datendichte pro Kommunikationszyklus ermöglicht. Die Verwendung von ANY-Typen bietet maximale Flexibilität bei den zu übertragenden Datentypen.

Fazit

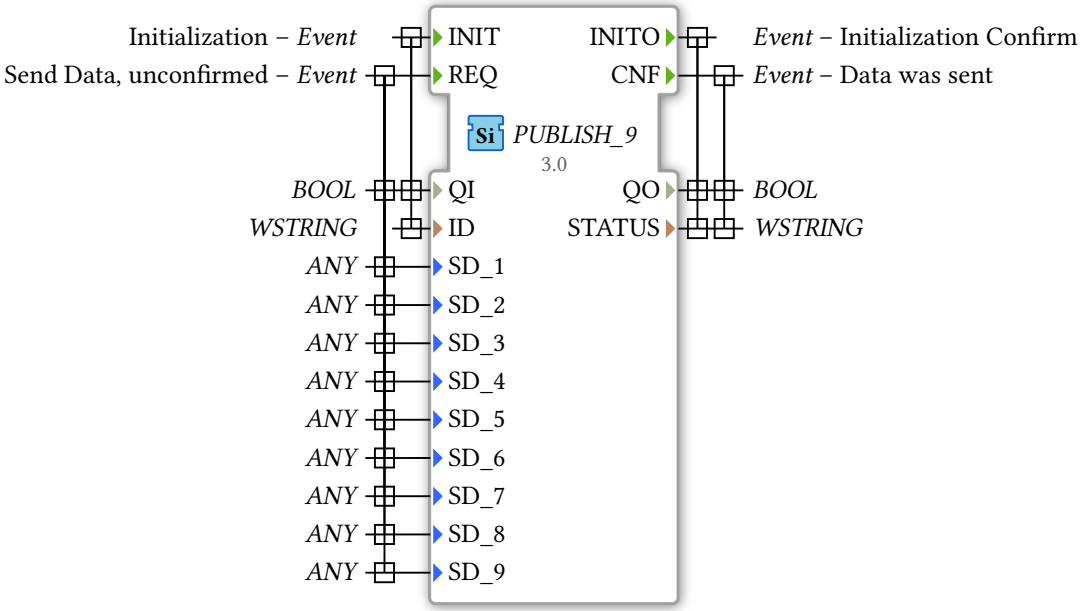
Der PUBLISH_8 Funktionsblock ist eine leistungsstarke Lösung für die Verteilung multipler Datenwerte in verteilten Automatisierungssystemen. Seine Flexibilität bei den Datentypen und die Unterstützung für acht parallele Datenkanäle machen ihn ideal für komplexe Kommunikationsszenarien in industriellen Anwendungen.

2.4.3.46 PUBLISH_9

• – ◦ • – ◦ • – ◦ •

Einleitung

Der PUBLISH_9 Funktionsblock dient zum Veröffentlichen von Daten an einen oder mehrere SUBSCRIBE_9 Blöcke. Er ermöglicht die unbestätigte Übertragung von bis zu neun verschiedenen Datenwerten über ein Kommunikationsnetzwerk.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsereignis mit zugehörigen Daten QI und ID
- **REQ**: Sendeanforderung für unbestätigte Datenübertragung mit neun Datenwerten

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung mit QO und STATUS
- **CNF**: Bestätigung, dass Daten gesendet wurden, mit QO und STATUS

Daten-Eingänge

- **QI** (BOOL): Qualifizierer für Initialisierung (TRUE = aktivieren, FALSE = deaktivieren)

- **ID** (WSTRING): Identifikator für den Kommunikationskanal
- **SD_1** bis **SD_9** (ANY): Neun verschiedene zu sendende Datenwerte beliebigen Typs

Daten-Ausgänge

- **QO** (BOOL): Qualifizierer für Ausgangszustand
- **STATUS** (WSTRING): Statusinformationen über den Betriebszustand

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der PUBLISH_9 Block wird zunächst über das INIT-Ereignis initialisiert, wobei der ID-Parameter den Kommunikationskanal definiert. Bei aktiviertem QI-Eingang (TRUE) können Daten über das REQ-Ereignis gesendet werden. Der Block überträgt dabei alle neun SD_x Datenwerte gleichzeitig an alle angeschlossenen SUBSCRIBE_9 Blöcke. Die Übertragung erfolgt unbestätigt, jedoch signalisiert das CNF-Ereignis die erfolgreiche Abarbeitung der Sendeanforderung.

Technische Besonderheiten

- Unterstützt bis zu neun verschiedene Datenwerte beliebigen Typs (ANY)
- Unbestätigte Kommunikation (fire-and-forget Prinzip)
- WSTRING-basierte Kanalidentifikation für flexible Adressierung
- Generische Implementierung durch GEN_PUBLISH Basisklasse

Zustandsübersicht

Der Block verfügt über zwei Hauptzustände: initialisiert und nicht initialisiert. Nach erfolgreicher Initialisierung (INIT mit QI=TRUE) wechselt der Block in den aktiven Zustand und kann Daten senden. Bei Deaktivierung (INIT mit QI=FALSE) wird der Block deinitialisiert.

Anwendungsszenarien

- Verteilte Systeme mit Publisher-Subscriber-Architektur
- Datenaustausch zwischen verschiedenen Steuerungskomponenten
- Broadcast-Kommunikation in Automatisierungsnetzwerken
- Systeme mit 1-zu-n Kommunikationsbeziehungen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu bestätigten Kommunikationsblöcken bietet PUBLISH_9 eine reduzierte Latenz durch den Verzicht auf Empfangsbestätigungen. Gegenüber Blöcken mit weniger Datenkanälen ermöglicht er die gleichzeitige Übertragung mehrerer Datenwerte, was die Systemeffizienz erhöht.

Fazit

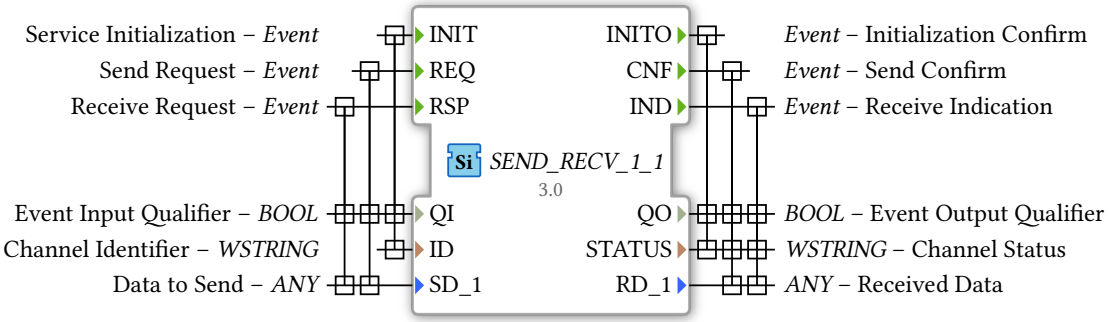
Der PUBLISH_9 Funktionsblock ist eine leistungsfähige Lösung für unbestätigte Mehrfachdatenübertragungen in verteilten Automatisierungssystemen. Seine Flexibilität bei der Datenübertragung und die Unterstützung für neun verschiedene Datenwerte machen ihn ideal für komplexe Kommunikationsszenarien in industriellen Steuerungssystemen.

2.4.3.47 SEND_RECV_1_1

• – ◦ • – ◦ • – ◦ •

Einleitung

Der SEND_RECV_1_1 Funktionsblock implementiert eine Duplex-Kommunikationsschnittstelle für den bidirektionalen Datenaustausch über Kommunikationskanäle. Der Baustein ermöglicht sowohl das Senden als auch das Empfangen von Daten über einen gemeinsamen Kanal und unterstützt generische Datentypen für maximale Flexibilität.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Service-Initialisierung mit zugehörigen Daten QI und ID
- **REQ**: Sende-Anfrage mit zugehörigen Daten QI und SD_1
- **RSP**: Empfangs-Anfrage mit zugehörigen Daten QI und SD_1

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung mit QO und STATUS
- **CNF**: Sende-Bestätigung mit QO, STATUS und RD_1
- **IND**: Empfangs-Indikation mit QO, STATUS und RD_1

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangsqualifizierer
- **ID** (WSTRING): Kanal-Identifikator
- **SD_1** (ANY): Zu sendende Daten (generischer Datentyp)

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifizierer
- **STATUS** (WSTRING): Kanal-Statusinformation
- **RD_1** (ANY): Empfangene Daten (generischer Datentyp)

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Funktionsblock arbeitet als bidirektionale Kommunikationsschnittstelle, die über den INIT-Ereigniseingang initialisiert wird. Nach erfolgreicher Initialisierung können über REQ Daten gesendet und über RSP Daten empfangen werden. Die Kommunikation erfolgt über einen durch ID identifizierten Kanal. Der generische ANY-Datentyp ermöglicht die Übertragung beliebiger Datenstrukturen.

Technische Besonderheiten

- Unterstützung generischer Datentypen (ANY) für maximale Flexibilität
- Bidirektionale Kommunikation über einen gemeinsamen Kanal
- Unicode-String-Unterstützung für Kanal-IDs und Statusmeldungen
- Separate Qualifizierer für Eingangs- und Ausgangsereignisse
- Statusrückmeldung über WSTRING für detaillierte Fehlerinformationen

Zustandsübersicht

Der Funktionsblock durchläuft folgende Zustände:

1. **Nicht initialisiert**: Vor INIT-Aufruf
2. **Initialisiert**: Nach erfolgreicher INIT/INITO-Sequenz
3. **Sende-Betrieb**: Während REQ/CNF-Operation
4. **Empfangs-Betrieb**: Während RSP/IND-Operation

Anwendungsszenarien

- Client-Server-Kommunikation in verteilten Systemen
- Bidirektionale Datenübertragung zwischen Steuerungskomponenten
- Protokollimplementierungen mit Anfrage-Antwort-Mustern
- Netzwerkkommunikation in industriellen Automatisierungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen SEND- oder RECV-Blöcken bietet SEND_RECV_1_1 die kombinierte Funktionalität beider Operationen in einem einzigen Baustein. Die Duplex-Fähigkeit unterscheidet ihn von Simplex-Kommunikationsblöcken und ermöglicht effizientere Kommunikationsmuster.

Fazit

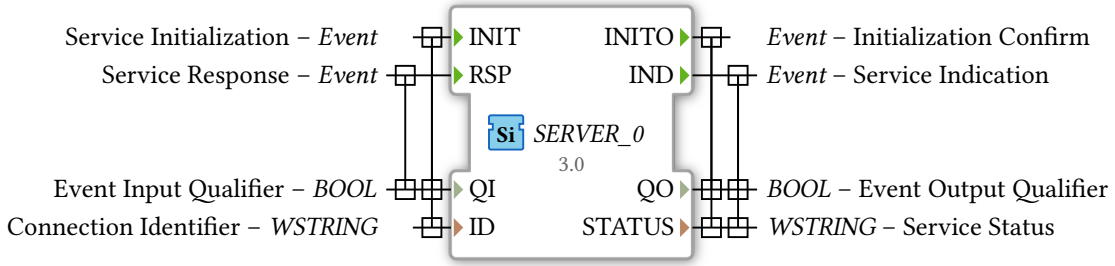
Der SEND_RECV_1_1 Funktionsblock stellt eine vielseitige Lösung für bidirektionale Kommunikationsanforderungen in verteilten Automatisierungssystemen dar. Durch die Unterstützung generischer Datentypen und die kombinierte Sende-/Empfangsfunktionalität bietet er eine effiziente Basis für verschiedene Kommunikationsszenarien.

2.4.3.48 SERVER_0

• – ◦ • – ◦ • – ◦ •

Einleitung

Der SERVER_0 Funktionsblock dient als Gegenstelle zu einem CLIENT_0 Block. Er ermöglicht die Einrichtung einer Server-Client-Verbindung ohne Austausch von Nutzdaten über Daten-Ein-/Ausgänge.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (Typ: Event) - Aktiviert den Server für eine Client-Verbindung (QI = TRUE) oder schließt den Server (QI = FALSE)
- **RSP** (Typ: Event) - Sendet eine Antwort/Bestätigung an den Client

Ereignis-Ausgänge

- **INITO** (Typ: Event) - Signalisiert Bereitschaft für Client-Verbindung (QI = TRUE) oder geschlossenen Zustand (QI = FALSE)
- **IND** (Typ: Event) - Zeigt an, dass eine Anfrage vom Client eingegangen ist

Daten-Eingänge

- **QI** (BOOL) - Qualifizierer für INIT-Operation
- **ID** (WSTRING) - Identifikator für die Server-Instanz

Daten-Ausgänge

- **QO** (BOOL) - Qualifizierer für Statusausgabe
- **STATUS** (WSTRING) - Statusinformationen

Funktionsweise

Der SERVER_0 Block verwaltet die Verbindung.

1. **Initialisierung**: Aktivierung über INIT.
2. **Synchronisation**:

- Wenn der Client eine Anfrage sendet (REQ), wird am Server das IND-Ereignis ausgelöst.
- Der Server kann daraufhin mit RSP antworten, was beim Client zum CNF-Ereignis führt. Es werden dabei keine Nutzdaten (SD_x, RD_x) übertragen.

Technische Besonderheiten
- Keine Daten-Ein-/Ausgänge vorhanden
- WSTRING-Typ für ID und STATUS
- Generische Implementierung durch ‚GEN_SERVER' Klassenattribut

Zustandsübersicht
- **Nicht initialisiert**: Server inaktiv
- **Initialisiert**: Server bereit
- **Indication (IND)**: Client hat angefragt
- **Response (RSP)**: Server antwortet

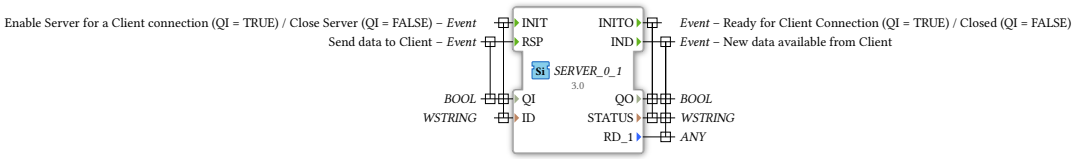
Anwendungsszenarien
- Ereignis-Synchronisation
- Signalisierung von Zuständen ohne Dateninhalt

Metadaten

Attribut	Wert
Copyright	(c) 2025 Monika Wenger
Lizenz	EPL-2.0
Version	3.0 (2025-12-05, Monika Wenger)
4diac-Paket	iec61499::net

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.3.49 SERVER_0_1



• – ○ • – ○ • – ○ •

Einleitung

Der `SERVER_0_1`-Funktionsblock ist die generische Server-Variante mit 0 Sende- und 1 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_1_0-Block. Er empfängt 1 Datenwerte (`RD_1`) vom Client, ohne selbst Nutzdaten zu senden. Wie alle `SERVER_*`-Bausteine basiert er auf der generischen `GEN_SERVER`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge
- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt `QI` und `ID`.
- **RSP**: Sendet Daten an den Client, trägt `QI`.

Ereignis-Ausgänge
- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.

- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- Keine Sendedaten (`SERVER_0_1` überträgt keine Nutzdaten mit `RSP`, nur die Verbindungssteuerung).

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `IND` vom Client geliefert.

Funktionsweise

`SERVER_0_1` initialisiert über `INIT` eine Verbindung zum passenden `CLIENT_1_0`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `RSP` wird eine Anfrage ohne Nutzdaten gesendet. Sobald Antwortdaten (`RD_1`) vorliegen, löst der Baustein `IND` aus und stellt `RD_1` bereit.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, dieselbe C++-Basis wie alle anderen `SERVER_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `CLIENT_1_0`**: `SERVER_0_1` ist funktional nur mit einem `CLIENT_1_0`-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `RSP`/`IND`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 0 Werte gesendet und 1 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als `CLIENT_1_0` mit passender Feldanzahl ausgelegt ist.

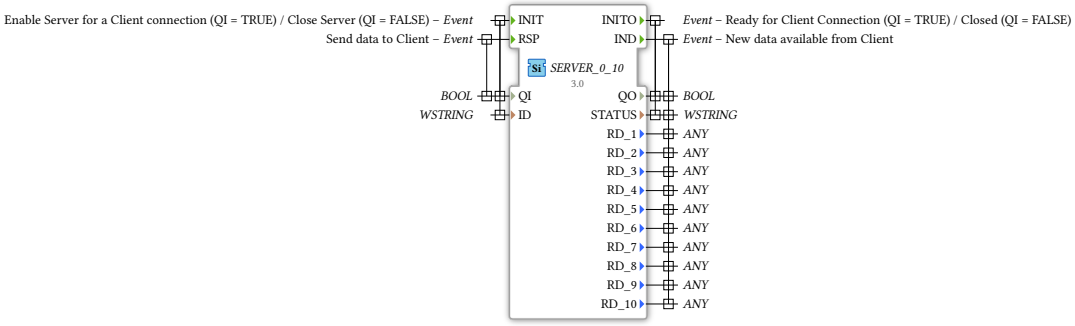
⚖️ Vergleich mit ähnlichen Bausteinen

- **CLIENT_1_0**: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`SERVER_0_1` liefert die generische, auf 0 Sende- und 1 Empfangsfelder zugeschnittene Server-Variante der `GEN_SERVER`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.50 SERVER_0_10



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `SERVER_0_10`-Funktionsblock ist die generische Server-Variante mit 0 Sende- und 10 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_10_0-Block. Er empfängt 10 Datenwerte (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9` `RD_10`) vom Client, ohne selbst Nutzdaten zu senden. Wie alle `SERVER_*`-Bausteine basiert er auf der generischen `GEN_SERVER`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt `QI` und `ID`.
- **RSP**: Sendet Daten an den Client, trägt `QI`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9` `RD_10`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- Keine Sendedaten (`SERVER_0_10` überträgt keine Nutzdaten mit `RSP`, nur die Verbindungssteuerung).

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `IND` vom Client geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `IND` vom Client geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit `IND` vom Client geliefert.
- **RD_4** (ANY): Empfangsdatum 4, wird mit `IND` vom Client geliefert.
- **RD_5** (ANY): Empfangsdatum 5, wird mit `IND` vom Client geliefert.
- **RD_6** (ANY): Empfangsdatum 6, wird mit `IND` vom Client geliefert.
- **RD_7** (ANY): Empfangsdatum 7, wird mit `IND` vom Client geliefert.
- **RD_8** (ANY): Empfangsdatum 8, wird mit `IND` vom Client geliefert.
- **RD_9** (ANY): Empfangsdatum 9, wird mit `IND` vom Client geliefert.
- **RD_10** (ANY): Empfangsdatum 10, wird mit `IND` vom Client geliefert.

Funktionsweise

`SERVER_0_10` initialisiert über `INIT` eine Verbindung zum passenden `CLIENT_10_0`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `RSP` wird eine Anfrage ohne Nutzdaten gesendet. Sobald Antwortdaten (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9` `RD_10`) vorliegen, löst der Baustein `IND` aus und stellt `RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9` `RD_10` bereit.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, dieselbe C++-Basis wie alle anderen `SERVER_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `CLIENT_10_0`**: `SERVER_0_10` ist funktional nur mit einem `CLIENT_10_0`-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `RSP`/`IND`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 0 Werte gesendet und 10 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als `CLIENT_10_0` mit passender Feldanzahl ausgelegt ist.

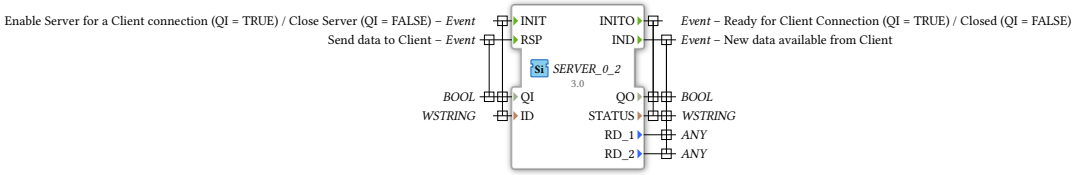
⚖️ Vergleich mit ähnlichen Bausteinen

- **CLIENT_10_0**: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`SERVER_0_10` liefert die generische, auf 0 Sende- und 10 Empfangsfelder zugeschnittene Server-Variante der `GEN_SERVER`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.51 SERVER_0_2



• – ○ • – ○ • – ○ •

Einleitung

Der `SERVER_0_2`-Funktionsblock ist die generische Server-Variante mit 0 Sende- und 2 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_2_0-Block. Er empfängt 2 Datenwerte (`RD_1` `RD_2`) vom Client, ohne selbst Nutzdaten zu senden. Wie alle `SERVER_*`-Bausteine basiert er auf der generischen `GEN_SERVER`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt `QI` und `ID`.
- **RSP**: Sendet Daten an den Client, trägt `QI`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.

- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1` `RD_2`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- Keine Sendedaten (`SERVER_0_2` überträgt keine Nutzdaten mit `RSP`, nur die Verbindungssteuerung).

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `IND` vom Client geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `IND` vom Client geliefert.

Funktionsweise

`SERVER_0_2` initialisiert über `INIT` eine Verbindung zum passenden `CLIENT_2_0`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `RSP` wird eine Anfrage ohne Nutzdaten gesendet. Sobald Antwortdaten (`RD_1` `RD_2`) vorliegen, löst der Baustein `IND` aus und stellt `RD_1` `RD_2` bereit.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, dieselbe C++-Basis wie alle anderen `SERVER_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `CLIENT_2_0`**: `SERVER_0_2` ist funktional nur mit einem `CLIENT_2_0`-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `RSP`/`IND`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 0 Werte gesendet und 2 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als `CLIENT_2_0` mit passender Feldanzahl ausgelegt ist.

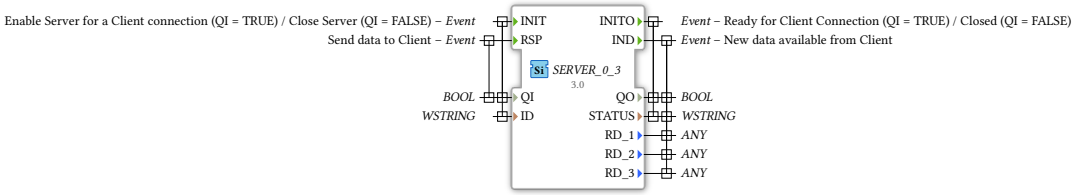
⚖️ Vergleich mit ähnlichen Bausteinen

- **CLIENT_2_0**: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`SERVER_0_2` liefert die generische, auf 0 Sende- und 2 Empfangsfelder zugeschnittene Server-Variante der `GEN_SERVER`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.52 SERVER_0_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `SERVER_0_3`-Funktionsblock ist die generische Server-Variante mit 0 Sende- und 3 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_3_0-Block. Er empfängt 3 Datenwerte (`RD_1` `RD_2` `RD_3`) vom Client, ohne selbst Nutzdaten zu senden. Wie alle `SERVER_*`-Bausteine basiert er auf der generischen `GEN_SERVER`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt `QI` und `ID`.
- **RSP**: Sendet Daten an den Client, trägt `QI`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1` `RD_2` `RD_3`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- Keine Sendedaten (`SERVER_0_3` überträgt keine Nutzdaten mit `RSP`, nur die Verbindungssteuerung).

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `IND` vom Client geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `IND` vom Client geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit `IND` vom Client geliefert.

Funktionsweise

`SERVER_0_3` initialisiert über `INIT` eine Verbindung zum passenden `CLIENT_3_0`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `RSP` wird eine Anfrage ohne Nutzdaten gesendet. Sobald Antwortdaten (`RD_1` `RD_2` `RD_3`) vorliegen, löst der Baustein `IND` aus und stellt `RD_1` `RD_2` `RD_3` bereit.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, dieselbe C++-Basis wie alle anderen `SERVER_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `CLIENT_3_0`**: `SERVER_0_3` ist funktional nur mit einem `CLIENT_3_0`-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.

2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `RSP`/`IND`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 0 Werte gesendet und 3 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als `CLIENT_3_0` mit passender Feldanzahl ausgelegt ist.

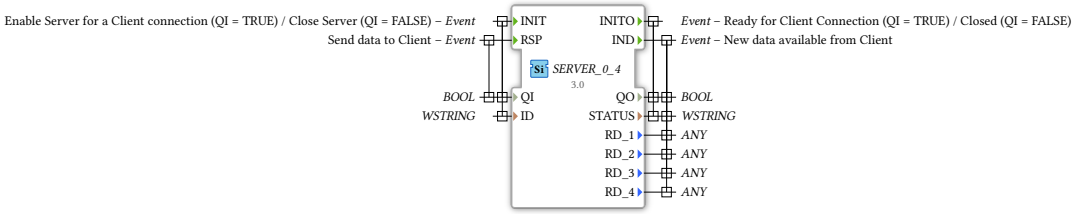
⚖️ Vergleich mit ähnlichen Bausteinen

- **CLIENT_3_0**: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`SERVER_0_3` liefert die generische, auf 0 Sende- und 3 Empfangsfelder zugeschnittene Server-Variante der `GEN_SERVER`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.53 SERVER_0_4



• – ○ • – ○ • – ○ •

Einleitung

Der `SERVER_0_4`-Funktionsblock ist die generische Server-Variante mit 0 Sende- und 4 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_4_0-Block. Er empfängt 4 Datenwerte (`RD_1` `RD_2` `RD_3` `RD_4`) vom Client, ohne selbst Nutzdaten zu senden. Wie alle `SERVER_*`-Bausteine basiert er auf der generischen `GEN_SERVER`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt `QI` und `ID`.
- **RSP**: Sendet Daten an den Client, trägt `QI`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1` `RD_2` `RD_3` `RD_4`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- Keine Sendedaten (`SERVER_0_4` überträgt keine Nutzdaten mit `RSP`, nur die Verbindungssteuerung).

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.

- **RD_1** (ANY): Empfangsdatum 1, wird mit `IND` vom Client geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `IND` vom Client geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit `IND` vom Client geliefert.
- **RD_4** (ANY): Empfangsdatum 4, wird mit `IND` vom Client geliefert.

Funktionsweise

`SERVER_0_4` initialisiert über `INIT` eine Verbindung zum passenden `CLIENT_4_0`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `RSP` wird eine Anfrage ohne Nutzdaten gesendet. Sobald Antwortdaten (`RD_1 RD_2 RD_3 RD_4`) vorliegen, löst der Baustein `IND` aus und stellt `RD_1 RD_2 RD_3 RD_4` bereit.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, dieselbe C++-Basis wie alle anderen `SERVER_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `CLIENT_4_0`**: `SERVER_0_4` ist funktional nur mit einem `CLIENT_4_0`-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `RSP`/`IND`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 0 Werte gesendet und 4 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als `CLIENT_4_0` mit passender Feldanzahl ausgelegt ist.

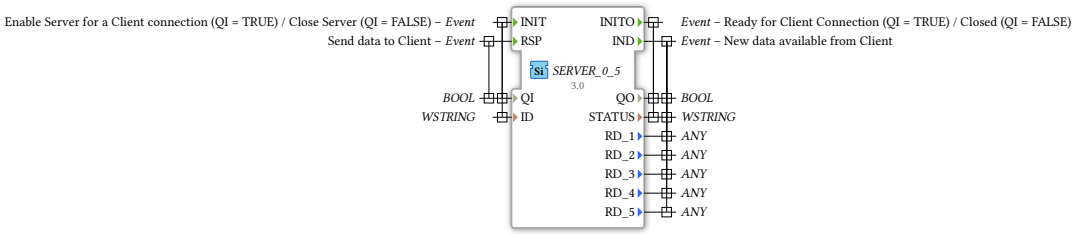
⚖️ Vergleich mit ähnlichen Bausteinen

- CLIENT_4_0: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- CLIENT_1 / SERVER_1: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`SERVER_0_4` liefert die generische, auf 0 Sende- und 4 Empfangsfelder zugeschnittene Server-Variante der `GEN_SERVER`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.54 SERVER_0_5



• – ○ • – ○ • – ○ •

Einleitung

Der `SERVER_0_5`-Funktionsblock ist die generische Server-Variante mit 0 Sende- und 5 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_5_0-Block. Er empfängt 5 Datenwerte (`RD_1`

RD_2 RD_3 RD_4 RD_5) vom Client, ohne selbst Nutzdaten zu senden. Wie alle SERVER_*-Bausteine basiert er auf der generischen GEN_SERVER-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt QI und ID.
- **RSP**: Sendet Daten an den Client, trägt QI.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt QO und STATUS.
- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt QO, STATUS sowie RD_1 RD_2 RD_3 RD_4 RD_5.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (TRUE = Verbindung öffnen, FALSE = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- Keine Sendedaten (SERVER_0_5 überträgt keine Nutzdaten mit RSP, nur die Verbindungssteuerung).

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit IND vom Client geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit IND vom Client geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit IND vom Client geliefert.
- **RD_4** (ANY): Empfangsdatum 4, wird mit IND vom Client geliefert.
- **RD_5** (ANY): Empfangsdatum 5, wird mit IND vom Client geliefert.

Funktionsweise

SERVER_0_5 initialisiert über INIT eine Verbindung zum passenden CLIENT_5_0-Block (bei QI = TRUE) bzw. schließt sie (bei QI = FALSE); der Abschluss wird über INITO bestätigt. Über RSP wird eine Anfrage ohne Nutzdaten gesendet. Sobald Antwortdaten (RD_1 RD_2 RD_3 RD_4 RD_5) vorliegen, löst der Baustein IND aus und stellt RD_1 RD_2 RD_3 RD_4 RD_5 bereit.

Technische Besonderheiten

- **Generische Implementierung**: eclipse4diac::core::GenericClassName = 'GEN_SERVER', dieselbe C++-Basis wie alle anderen SERVER_*-Varianten; Anzahl und Typ (ANY) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **ANY-Datenfelder**: Alle SD_i/RD_i sind generisch (ANY) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück CLIENT_5_0**: SERVER_0_5 ist funktional nur mit einem CLIENT_5_0-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, QO = FALSE.
2. **Verbindungsaufbau**: INIT mit QI = TRUE wird verarbeitet.
3. **Verbunden**: INITO mit QO = TRUE bestätigt die Verbindung.
4. **Datenaustausch**: RSP/IND-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: INIT mit QI = FALSE wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 0 Werte gesendet und 5 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als CLIENT_5_0 mit passender Feldanzahl ausgelegt ist.

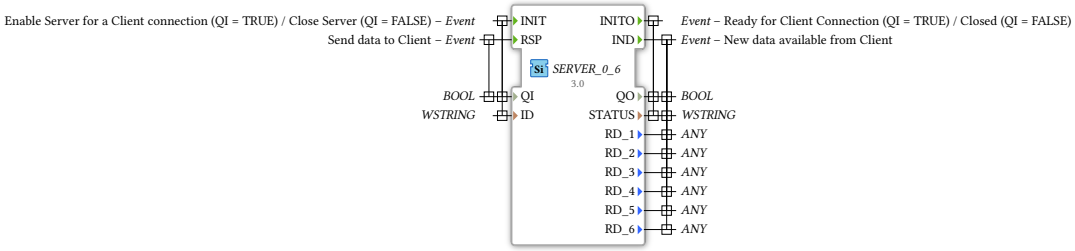
⚖️ Vergleich mit ähnlichen Bausteinen

- **CLIENT_5_0**: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`SERVER_0_5` liefert die generische, auf 0 Sende- und 5 Empfangsfelder zugeschnittene Server-Variante der `GEN_SERVER`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.55 SERVER_0_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `SERVER_0_6`-Funktionsblock ist die generische Server-Variante mit 0 Sende- und 6 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_6_0-Block. Er empfängt 6 Datenwerte (`RD_1 RD_2 RD_3 RD_4 RD_5 RD_6`) vom Client, ohne selbst Nutzdaten zu senden. Wie alle `SERVER_*`-Bausteine basiert er auf der generischen `GEN_SERVER`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt `QI` und `ID`.
- **RSP**: Sendet Daten an den Client, trägt `QI`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1 RD_2 RD_3 RD_4 RD_5 RD_6`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- Keine Sendedaten (`SERVER_0_6` überträgt keine Nutzdaten mit `RSP`, nur die Verbindungssteuerung).

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `IND` vom Client geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `IND` vom Client geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit `IND` vom Client geliefert.
- **RD_4** (ANY): Empfangsdatum 4, wird mit `IND` vom Client geliefert.
- **RD_5** (ANY): Empfangsdatum 5, wird mit `IND` vom Client geliefert.
- **RD_6** (ANY): Empfangsdatum 6, wird mit `IND` vom Client geliefert.

Funktionsweise

SERVER_0_6 initialisiert über INIT eine Verbindung zum passenden CLIENT_6_0-Block (bei QI = TRUE) bzw. schließt sie (bei QI = FALSE); der Abschluss wird über INITO bestätigt. Über RSP wird eine Anfrage ohne Nutzdaten gesendet. Sobald Antwortdaten (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6) vorliegen, löst der Baustein IND aus und stellt RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 bereit.

Technische Besonderheiten

- **Generische Implementierung**: eclipse4diac::core::GenericClassName = 'GEN_SERVER', dieselbe C++-Basis wie alle anderen SERVER_*-Varianten; Anzahl und Typ (ANY) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **ANY-Datenfelder**: Alle SD_i/RD_i sind generisch (ANY) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück CLIENT_6_0**: SERVER_0_6 ist funktional nur mit einem CLIENT_6_0-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, QO = FALSE.
2. **Verbindungsaufbau**: INIT mit QI = TRUE wird verarbeitet.
3. **Verbunden**: INITO mit QO = TRUE bestätigt die Verbindung.
4. **Datenaustausch**: RSP/IND-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: INIT mit QI = FALSE wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 0 Werte gesendet und 6 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als CLIENT_6_0 mit passender Feldanzahl ausgelegt ist.

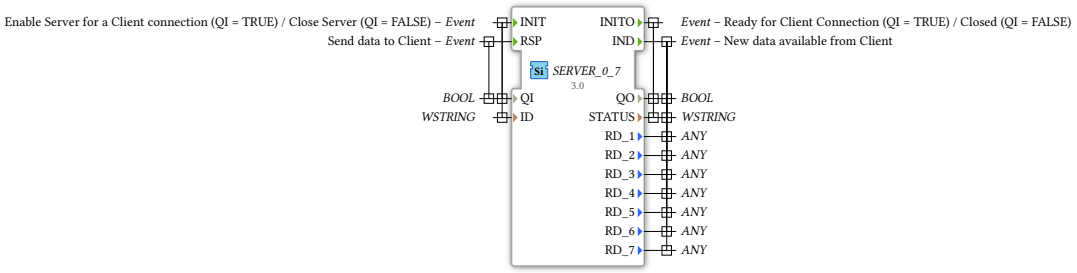
⚖️ Vergleich mit ähnlichen Bausteinen

- CLIENT_6_0: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- CLIENT_1 / SERVER_1: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

SERVER_0_6 liefert die generische, auf 0 Sende- und 6 Empfangsfelder zugeschnittene Server-Variante der GEN_SERVER-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.56 SERVER_0_7



• – ○ • – ○ • – ○ •

Einleitung

Der SERVER_0_7-Funktionsblock ist die generische Server-Variante mit 0 Sende- und 7 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_7_0-Block. Er empfängt 7 Datenwerte (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7) vom Client, ohne selbst Nutzdaten zu senden. Wie alle SERVER_*-Bausteine basiert er auf der generischen GEN_SERVER-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt `QI` und `ID`.
- **RSP**: Sendet Daten an den Client, trägt `QI`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- Keine Sendedaten (`SERVER_0_7` überträgt keine Nutzdaten mit `RSP`, nur die Verbindungssteuerung).

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `IND` vom Client geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `IND` vom Client geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit `IND` vom Client geliefert.
- **RD_4** (ANY): Empfangsdatum 4, wird mit `IND` vom Client geliefert.
- **RD_5** (ANY): Empfangsdatum 5, wird mit `IND` vom Client geliefert.
- **RD_6** (ANY): Empfangsdatum 6, wird mit `IND` vom Client geliefert.
- **RD_7** (ANY): Empfangsdatum 7, wird mit `IND` vom Client geliefert.

Funktionsweise

`SERVER_0_7` initialisiert über `INIT` eine Verbindung zum passenden `CLIENT_7_0`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `RSP` wird eine Anfrage ohne Nutzdaten gesendet. Sobald Antwortdaten (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7`) vorliegen, löst der Baustein `IND` aus und stellt `RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` bereit.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, dieselbe C++-Basis wie alle anderen `SERVER_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `CLIENT_7_0`**: `SERVER_0_7` ist funktional nur mit einem `CLIENT_7_0`-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `RSP`/`IND`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 0 Werte gesendet und 7 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als `CLIENT_7_0` mit passender Feldanzahl ausgelegt ist.

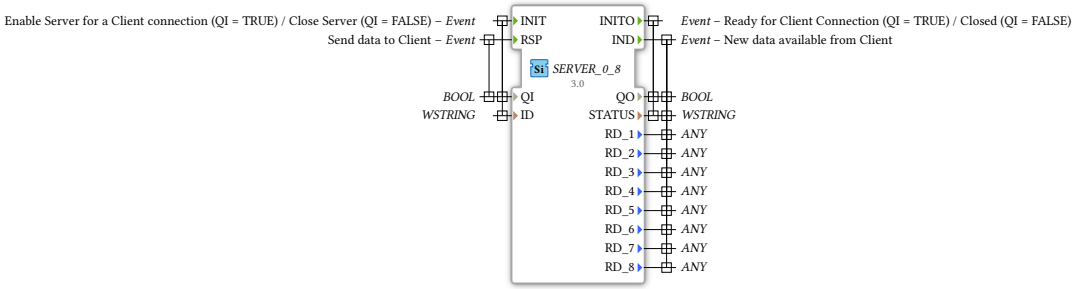
⚖️ Vergleich mit ähnlichen Bausteinen

- **CLIENT_7_0**: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`SERVER_0_7` liefert die generische, auf 0 Sende- und 7 Empfangsfelder zugeschnittene Server-Variante der `GEN_SERVER`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.57 SERVER_0_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `SERVER_0_8`-Funktionsblock ist die generische Server-Variante mit 0 Sende- und 8 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_8_0-Block. Er empfängt 8 Datenwerte (`RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8`) vom Client, ohne selbst Nutzdaten zu senden. Wie alle `SERVER_*`-Bausteine basiert er auf der generischen `GEN_SERVER`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt `QI` und `ID`.
- **RSP**: Sendet Daten an den Client, trägt `QI`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- Keine Sendedaten (`SERVER_0_8` überträgt keine Nutzdaten mit `RSP`, nur die Verbindungssteuerung).

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `IND` vom Client geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `IND` vom Client geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit `IND` vom Client geliefert.
- **RD_4** (ANY): Empfangsdatum 4, wird mit `IND` vom Client geliefert.
- **RD_5** (ANY): Empfangsdatum 5, wird mit `IND` vom Client geliefert.
- **RD_6** (ANY): Empfangsdatum 6, wird mit `IND` vom Client geliefert.
- **RD_7** (ANY): Empfangsdatum 7, wird mit `IND` vom Client geliefert.

- **RD_8** (ANY): Empfangsdatum 8, wird mit `IND` vom Client geliefert.

Funktionsweise

`SERVER_0_8` initialisiert über `INIT` eine Verbindung zum passenden `CLIENT_8_0`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `RSP` wird eine Anfrage ohne Nutzdaten gesendet. Sobald Antwortdaten (`RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8`) vorliegen, löst der Baustein `IND` aus und stellt `RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8` bereit.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, dieselbe C++-Basis wie alle anderen `SERVER_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `CLIENT_8_0`**: `SERVER_0_8` ist funktional nur mit einem `CLIENT_8_0`-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `RSP`/`IND`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 0 Werte gesendet und 8 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als `CLIENT_8_0` mit passender Feldanzahl ausgelegt ist.

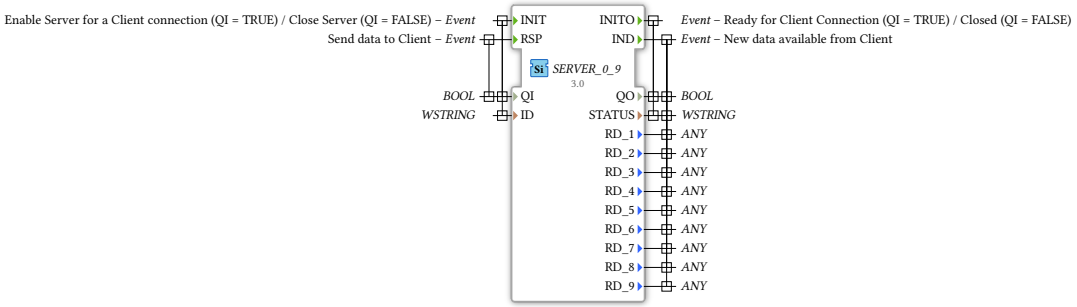
⚖️ Vergleich mit ähnlichen Bausteinen

- CLIENT_8_0: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- CLIENT_1 / SERVER_1: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`SERVER_0_8` liefert die generische, auf 0 Sende- und 8 Empfangsfelder zugeschnittene Server-Variante der `GEN_SERVER`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.58 SERVER_0_9



• – ○ • – ○ • – ○ •

Einleitung

Der `SERVER_0_9`-Funktionsblock ist die generische Server-Variante mit 0 Sende- und 9 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_9_0-Block. Er empfängt 9 Datenwerte

(RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9) vom Client, ohne selbst Nutzdaten zu senden. Wie alle SERVER_*-Bausteine basiert er auf der generischen GEN_SERVER-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt QI und ID.
- **RSP**: Sendet Daten an den Client, trägt QI.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt QO und STATUS.
- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt QO, STATUS sowie RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (TRUE = Verbindung öffnen, FALSE = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- Keine Sendedaten (SERVER_0_9 überträgt keine Nutzdaten mit RSP, nur die Verbindungssteuerung).

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit IND vom Client geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit IND vom Client geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit IND vom Client geliefert.
- **RD_4** (ANY): Empfangsdatum 4, wird mit IND vom Client geliefert.
- **RD_5** (ANY): Empfangsdatum 5, wird mit IND vom Client geliefert.
- **RD_6** (ANY): Empfangsdatum 6, wird mit IND vom Client geliefert.
- **RD_7** (ANY): Empfangsdatum 7, wird mit IND vom Client geliefert.
- **RD_8** (ANY): Empfangsdatum 8, wird mit IND vom Client geliefert.
- **RD_9** (ANY): Empfangsdatum 9, wird mit IND vom Client geliefert.

Funktionsweise

SERVER_0_9 initialisiert über INIT eine Verbindung zum passenden CLIENT_9_0-Block (bei QI = TRUE) bzw. schließt sie (bei QI = FALSE); der Abschluss wird über INITO bestätigt. Über RSP wird eine Anfrage ohne Nutzdaten gesendet. Sobald Antwortdaten (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9) vorliegen, löst der Baustein IND aus und stellt RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9 bereit.

Technische Besonderheiten

- **Generische Implementierung**: eclipse4diac::core::GenericClassName = 'GEN_SERVER', dieselbe C++-Basis wie alle anderen SERVER_*-Varianten; Anzahl und Typ (ANY) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **ANY-Datenfelder**: Alle SD_i/RD_i sind generisch (ANY) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück CLIENT_9_0**: SERVER_0_9 ist funktional nur mit einem CLIENT_9_0-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, QO = FALSE.
2. **Verbindungsaufbau**: INIT mit QI = TRUE wird verarbeitet.
3. **Verbunden**: INITO mit QO = TRUE bestätigt die Verbindung.
4. **Datenaustausch**: RSP/IND-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: INIT mit QI = FALSE wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 0 Werte gesendet und 9 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als `CLIENT_9_0` mit passender Feldanzahl ausgelegt ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- **CLIENT_9_0**: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

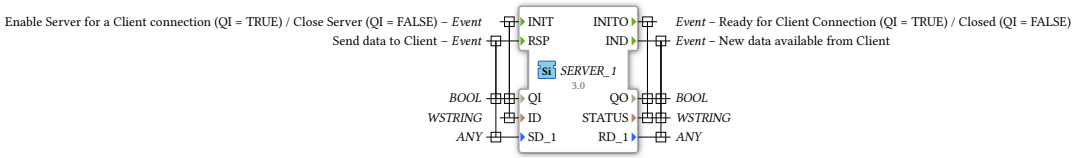
`SERVER_0_9` liefert die generische, auf 0 Sende- und 9 Empfangsfelder zugeschnittene Server-Variante der `GEN_SERVER`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.59 SERVER_1

• – ◦ • – ◦ • – ◦ •

Einleitung

Der SERVER_1 Funktionsblock dient zur Kommunikation mit einem CLIENT_1 Block und ermöglicht die Einrichtung einer Server-Client-Verbindung. Der Block kann Daten von Clients empfangen und an diese senden, wobei er eine flexible Kommunikationsschnittstelle für verschiedene Anwendungsfälle bereitstellt.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (Typ: EInit) - Aktiviert den Server für eine Client-Verbindung (QI = TRUE) oder schließt den Server (QI = FALSE)
- **RSP** (Typ: Event) - Sendet Daten an den Client

Ereignis-Ausgänge

- **INITO** (Typ: EInit) - Signalisiert Bereitschaft für Client-Verbindung (QI = TRUE) oder geschlossenen Zustand (QI = FALSE)
- **IND** (Typ: Event) - Zeigt neue verfügbare Daten vom Client an

Daten-Eingänge

- **QI** (BOOL) - Qualifizierer für INIT-Operation
- **ID** (WSTRING) - Identifikator für die Server-Instanz
- **SD_1** (ANY) - Sendedaten an den Client

Daten-Ausgänge

- **QO** (BOOL) - Qualifizierer für Statusausgabe
- **STATUS** (WSTRING) - Statusinformationen
- **RD_1** (ANY) - Empfangene Daten vom Client

Funktionsweise

Der SERVER_1 Block verwaltet Server-Client-Verbindungen über zwei Hauptoperationen:

1. **Initialisierung/Deaktivierung**: Durch das INIT-Ereignis mit entsprechendem QI-Wert wird der Server aktiviert oder deaktiviert. Bei erfolgreicher Operation wird INITO mit Statusinformationen ausgelöst.

2. **Datenkommunikation**:
   - Empfang: Eingehende Daten vom Client lösen das IND-Ereignis aus und stehen über RD_1 zur Verfügung
   - Sendung: Das RSP-Ereignis ermöglicht das Senden von Daten über SD_1 an den verbundenen Client

Technische Besonderheiten

- Verwendung des ANY-Datentyps für SD_1 und RD_1 ermöglicht flexible Datenformate
- WSTRING-Typ für ID und STATUS unterstützt Unicode-Strings
- Generische Implementierung durch ‚GEN_SERVER' Klassenattribut
- Package-Zugehörigkeit: iec61499::net

Zustandsübersicht

Der Block kann folgende Zustände annehmen:

- **Nicht initialisiert**: Server nicht betriebsbereit
- **Initialisiert**: Server bereit für Client-Verbindungen
- **Datenempfang**: Verarbeitung eingehender Daten
- **Datensendung**: Versand von Daten an Client

Anwendungsszenarien

- Industrielle Automatisierungssysteme
- Vernetzte Steuerungssysteme
- IoT-Kommunikationsanwendungen
- Distributed Control Systems (DCS)
- Maschine-zu-Maschine-Kommunikation

⚖️ Vergleich mit ähnlichen Bausteinen

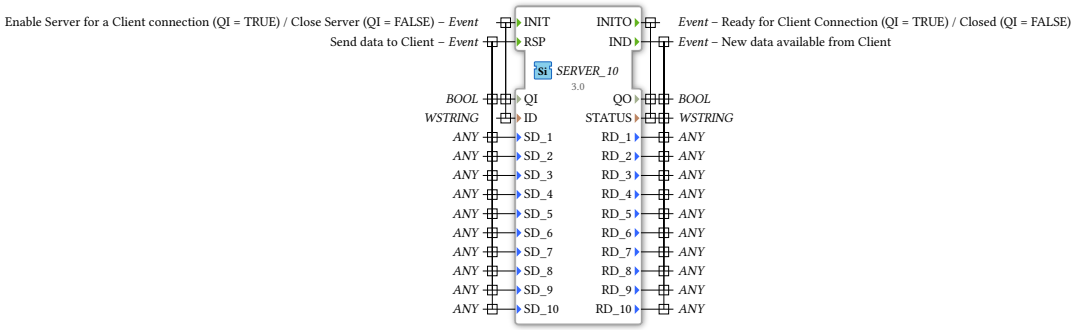
Im Vergleich zu anderen Kommunikationsblöcken bietet SERVER_1:

- Spezifische Server-Funktionalität für CLIENT_1-Blöcke
- Flexible Datenübertragung durch ANY-Typ
- Explizite Initialisierungs- und Deaktivierungssteuerung
- Statusrückmeldungen für verbesserte Fehlererkennung

Fazit

Der SERVER_1 Funktionsblock stellt eine robuste und flexible Lösung für Server-Client-Kommunikation in verteilten Automatisierungssystemen dar. Seine klare Schnittstellendefinition und flexible Datentypunterstützung machen ihn besonders geeignet für verschiedenste industrielle Kommunikationsanforderungen.

2.4.3.60 SERVER_10



• – ○ • – ○ • – ○ •

Einleitung

Der `SERVER_10`-Funktionsblock ist die generische Server-Variante mit 10 Sende- und 10 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_10-Block. Er überträgt 10 Datenwerte (`SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7` `SD_8` `SD_9` `SD_10`) an den Client und empfängt 10 Datenwerte (`RD_1`

RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9 RD_10) zurück. Wie alle SERVER_*-Bausteine basiert er auf der generischen GEN_SERVER-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt QI und ID.
- **RSP**: Sendet Daten an den Client, trägt QI sowie SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8 SD_9 SD_10.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt QO und STATUS.
- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt QO, STATUS sowie RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9 RD_10.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (TRUE = Verbindung öffnen, FALSE = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit RSP an den Client übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit RSP an den Client übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit RSP an den Client übertragen.
- **SD_4** (ANY): Sendedatum 4, wird mit RSP an den Client übertragen.
- **SD_5** (ANY): Sendedatum 5, wird mit RSP an den Client übertragen.
- **SD_6** (ANY): Sendedatum 6, wird mit RSP an den Client übertragen.
- **SD_7** (ANY): Sendedatum 7, wird mit RSP an den Client übertragen.
- **SD_8** (ANY): Sendedatum 8, wird mit RSP an den Client übertragen.
- **SD_9** (ANY): Sendedatum 9, wird mit RSP an den Client übertragen.
- **SD_10** (ANY): Sendedatum 10, wird mit RSP an den Client übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit IND vom Client geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit IND vom Client geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit IND vom Client geliefert.
- **RD_4** (ANY): Empfangsdatum 4, wird mit IND vom Client geliefert.
- **RD_5** (ANY): Empfangsdatum 5, wird mit IND vom Client geliefert.
- **RD_6** (ANY): Empfangsdatum 6, wird mit IND vom Client geliefert.
- **RD_7** (ANY): Empfangsdatum 7, wird mit IND vom Client geliefert.
- **RD_8** (ANY): Empfangsdatum 8, wird mit IND vom Client geliefert.
- **RD_9** (ANY): Empfangsdatum 9, wird mit IND vom Client geliefert.
- **RD_10** (ANY): Empfangsdatum 10, wird mit IND vom Client geliefert.

Funktionsweise

SERVER_10 initialisiert über INIT eine Verbindung zum passenden CLIENT_10-Block (bei QI = TRUE) bzw. schließt sie (bei QI = FALSE); der Abschluss wird über INITO bestätigt. Über RSP werden SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8 SD_9 SD_10 an den Client übertragen. Sobald Antwortdaten (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9 RD_10) vorliegen, löst der Baustein IND aus und stellt RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9 RD_10 bereit.

Technische Besonderheiten

- **Generische Implementierung**: eclipse4diac::core::GenericClassName = 'GEN_SERVER', dieselbe C++-Basis wie alle anderen SERVER_*-Varianten; Anzahl und Typ (ANY) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **ANY-Datenfelder**: Alle SD_i/RD_i sind generisch (ANY) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.

- **Gegenstück `CLIENT_10`**: `SERVER_10` ist funktional nur mit einem `CLIENT_10`-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `RSP`/`IND`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 10 Werte gesendet und 10 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als `CLIENT_10` mit passender Feldanzahl ausgelegt ist.

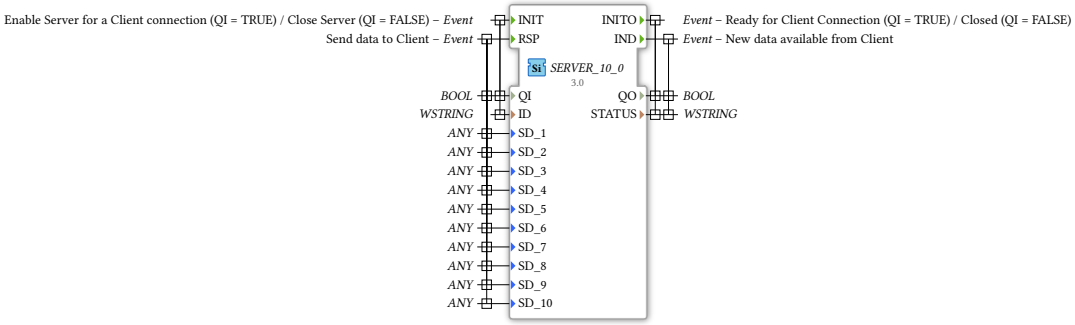
⚖️ Vergleich mit ähnlichen Bausteinen

- **CLIENT_10**: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`SERVER_10` liefert die generische, auf 10 Sende- und 10 Empfangsfelder zugeschnittene Server-Variante der `GEN_SERVER`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.61 SERVER_10_0



• – ○ • – ○ • – ○ •

Einleitung

Der `SERVER_10_0`-Funktionsblock ist die generische Server-Variante mit 10 Sende- und 0 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_0_10-Block. Er überträgt 10 Datenwerte (`SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7` `SD_8` `SD_9` `SD_10`) an den Client, ohne Rückgabedaten zu empfangen. Wie alle `SERVER_*`-Bausteine basiert er auf der generischen `GEN_SERVER`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt `QI` und `ID`.
- **RSP**: Sendet Daten an den Client, trägt `QI` sowie `SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7` `SD_8` `SD_9` `SD_10`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt `QO`, `STATUS`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit `RSP` an den Client übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `RSP` an den Client übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit `RSP` an den Client übertragen.
- **SD_4** (ANY): Sendedatum 4, wird mit `RSP` an den Client übertragen.
- **SD_5** (ANY): Sendedatum 5, wird mit `RSP` an den Client übertragen.
- **SD_6** (ANY): Sendedatum 6, wird mit `RSP` an den Client übertragen.
- **SD_7** (ANY): Sendedatum 7, wird mit `RSP` an den Client übertragen.
- **SD_8** (ANY): Sendedatum 8, wird mit `RSP` an den Client übertragen.
- **SD_9** (ANY): Sendedatum 9, wird mit `RSP` an den Client übertragen.
- **SD_10** (ANY): Sendedatum 10, wird mit `RSP` an den Client übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- Keine Empfangsdaten (`SERVER_10_0` erhält keine Nutzdaten über `IND`, nur Verbindungs-/Statusinformationen).

Funktionsweise

`SERVER_10_0` initialisiert über `INIT` eine Verbindung zum passenden `CLIENT_0_10`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `RSP` werden `SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7` `SD_8` `SD_9` `SD_10` an den Client übertragen. Sobald eine Antwort vorliegen, löst der Baustein `IND` aus.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, dieselbe C++-Basis wie alle anderen `SERVER_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `CLIENT_0_10`**: `SERVER_10_0` ist funktional nur mit einem `CLIENT_0_10`-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `RSP`/`IND`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 10 Werte gesendet und 0 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als `CLIENT_0_10` mit passender Feldanzahl ausgelegt ist.

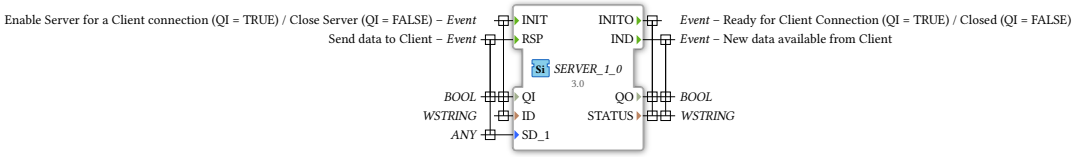
⚖️ Vergleich mit ähnlichen Bausteinen

- **CLIENT_0_10**: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`SERVER_10_0` liefert die generische, auf 10 Sende- und 0 Empfangsfelder zugeschnittene Server-Variante der `GEN_SERVER`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.62 SERVER_1_0



• – ◦ • – ◦ • – ◦ •

Einleitung

Der SERVER_1_0-Funktionsblock ist die generische Server-Variante mit 1 Sende- und 0 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_0_1-Block. Er überträgt 1 Datenwerte (SD_1) an den Client, ohne Rückgabedaten zu empfangen. Wie alle SERVER_*-Bausteine basiert er auf der generischen GEN_SERVER-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt QI und ID.
- **RSP**: Sendet Daten an den Client, trägt QI sowie SD_1.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt QO und STATUS.
- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt QO, STATUS.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (TRUE = Verbindung öffnen, FALSE = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit RSP an den Client übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- Keine Empfangsdaten (SERVER_1_0 erhält keine Nutzdaten über IND, nur Verbindungs-/Statusinformationen).

Funktionsweise

SERVER_1_0 initialisiert über INIT eine Verbindung zum passenden CLIENT_0_1-Block (bei QI = TRUE) bzw. schließt sie (bei QI = FALSE); der Abschluss wird über INITO bestätigt. Über RSP werden SD_1 an den Client übertragen. Sobald eine Antwort vorliegen, löst der Baustein IND aus.

Technische Besonderheiten

- **Generische Implementierung**: eclipse4diac::core::GenericClassName = 'GEN_SERVER', dieselbe C++-Basis wie alle anderen SERVER_*-Varianten; Anzahl und Typ (ANY) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **ANY-Datenfelder**: Alle SD_i/RD_i sind generisch (ANY) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück CLIENT_0_1**: SERVER_1_0 ist funktional nur mit einem CLIENT_0_1-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, QO = FALSE.
2. **Verbindungsaufbau**: INIT mit QI = TRUE wird verarbeitet.
3. **Verbunden**: INITO mit QO = TRUE bestätigt die Verbindung.
4. **Datenaustausch**: RSP/IND-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: INIT mit QI = FALSE wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 1 Werte gesendet und 0 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als `CLIENT_0_1` mit passender Feldanzahl ausgelegt ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- **CLIENT_0_1**: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

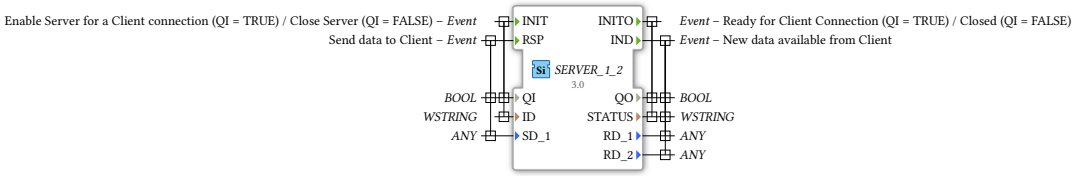
`SERVER_1_0` liefert die generische, auf 1 Sende- und 0 Empfangsfelder zugeschnittene Server-Variante der `GEN_SERVER`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.63 SERVER_1_2

• – ◦ • – ◦ • – ◦ •

Einleitung

Der SERVER_1_2 Funktionsblock dient zur Kommunikation mit einem CLIENT_2_1 Block und ermöglicht die Erstellung einer Server-Client-Verbindung. Der Block kann Daten empfangen und senden sowie die Verbindung initialisieren und beenden.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisiert den Server für eine Client-Verbindung (QI = TRUE) oder schließt den Server (QI = FALSE)
- **RSP**: Sendet Daten an den Client

Ereignis-Ausgänge

- **INITO**: Signalisiert Bereitschaft für Client-Verbindung (QI = TRUE) oder geschlossenen Zustand (QI = FALSE)
- **IND**: Zeigt neue verfügbare Daten vom Client an

Daten-Eingänge

- **QI** (BOOL): Qualifizierer für INIT-Ereignis (TRUE = aktivieren, FALSE = deaktivieren)
- **ID** (WSTRING): Identifikationsstring für die Server-Instanz
- **SD_1** (ANY): Zu sendende Daten an den Client

Daten-Ausgänge

- **QO** (BOOL): Statusausgabe (TRUE = aktiv, FALSE = inaktiv)
- **STATUS** (WSTRING): Statusinformationen als Zeichenkette
- **RD_1** (ANY): Empfangene Daten vom Client (Kanal 1)
- **RD_2** (ANY): Empfangene Daten vom Client (Kanal 2)

Funktionsweise

Der SERVER_1_2 Block arbeitet als Server in einer Client-Server-Architektur. Bei Initialisierung (INIT mit QI=TRUE) wird der Server für eingehende Verbindungen bereitgestellt. Bei eingehenden Daten vom Client wird das IND-Ereignis ausgelöst und die empfangenen Daten über RD_1 und RD_2 bereitgestellt. Über das RSP-Ereignis können Daten zurück an den Client gesendet werden.

Technische Besonderheiten

- Unterstützt den Datentyp ANY für flexible Datenübertragung
- Verwendet WSTRING für Statusmeldungen und Identifikation
- Bietet zwei separate Datenkanäle für den Empfang (RD_1 und RD_2)
- Implementiert als generischer Server-Baustein

Zustandsübersicht

- **Inaktiv**: Server nicht initialisiert (QO = FALSE)
- **Aktiv**: Server bereit für Client-Verbindungen (QO = TRUE)
- **Datenempfang**: Neue Daten vom Client verfügbar (IND-Ereignis)
- **Datensendung**: Daten werden an Client gesendet (RSP-Ereignis)

Anwendungsszenarien

- Netzwerkkommunikation in verteilten Automatisierungssystemen
- Datenaustausch zwischen verschiedenen Steuerungskomponenten
- Implementierung von Server-Diensten in IEC 61499-basierten Systemen
- Bidirektionale Kommunikation in industriellen Netzwerken

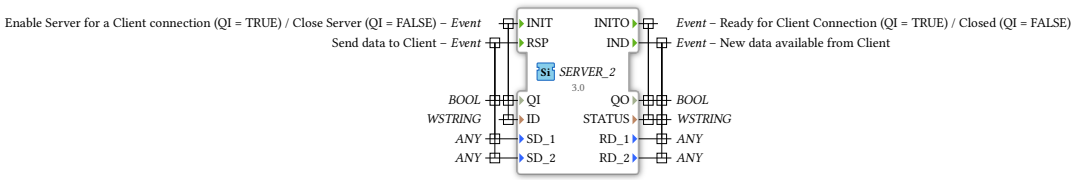
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren Kommunikationsbausteinen bietet SERVER_1_2 eine vollständige Server-Implementierung mit Initialisierungs- und Statusmanagement. Die zwei separaten Empfangskanäle ermöglichen eine strukturierte Datenverarbeitung.

Fazit

Der SERVER_1_2 Funktionsblock stellt eine robuste Lösung für Server-basierte Kommunikation in IEC 61499-Systemen dar. Durch die flexible Datentypunterstützung und klare Zustandsverwaltung eignet er sich ideal für verschiedene industrielle Kommunikationsszenarien.

2.4.3.64 SERVER_2



• – ○ • – ○ • – ○ •

Einleitung

Der `SERVER_2`-Funktionsblock ist die generische Server-Variante mit 2 Sende- und 2 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_2-Block. Er überträgt 2 Datenwerte (`SD_1` `SD_2`) an den Client und empfängt 2 Datenwerte (`RD_1` `RD_2`) zurück. Wie alle `SERVER_*`-Bausteine basiert er auf der generischen `GEN_SERVER`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt `QI` und `ID`.
- **RSP**: Sendet Daten an den Client, trägt `QI` sowie `SD_1` `SD_2`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1` `RD_2`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).

- **SD_1** (ANY): Sendedatum 1, wird mit `RSP` an den Client übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `RSP` an den Client übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `IND` vom Client geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `IND` vom Client geliefert.

Funktionsweise

`SERVER_2` initialisiert über `INIT` eine Verbindung zum passenden `CLIENT_2`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `RSP` werden `SD_1` `SD_2` an den Client übertragen. Sobald Antwortdaten (`RD_1` `RD_2`) vorliegen, löst der Baustein `IND` aus und stellt `RD_1` `RD_2` bereit.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, dieselbe C++-Basis wie alle anderen `SERVER_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `CLIENT_2`**: `SERVER_2` ist funktional nur mit einem `CLIENT_2`-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `RSP`/`IND`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 2 Werte gesendet und 2 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als `CLIENT_2` mit passender Feldanzahl ausgelegt ist.

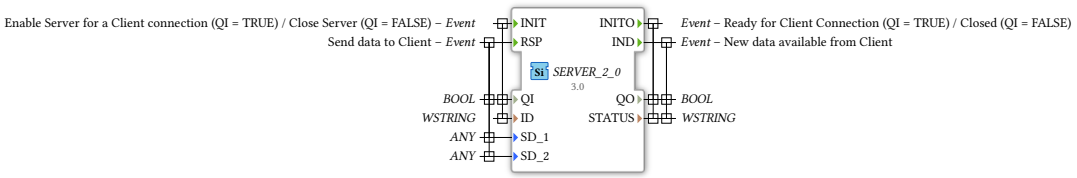
⚖️ Vergleich mit ähnlichen Bausteinen

- CLIENT_2: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- CLIENT_1 / SERVER_1: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`SERVER_2` liefert die generische, auf 2 Sende- und 2 Empfangsfelder zugeschnittene Server-Variante der `GEN_SERVER`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.65 SERVER_2_0



• – ◦ • – ◦ • – ◦ •

Einleitung

Der SERVER_2_0-Funktionsblock ist die generische Server-Variante mit 2 Sende- und 0 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_0_2-Block. Er überträgt 2 Datenwerte (SD_1 SD_2) an den Client, ohne Rückgabedaten zu empfangen. Wie alle SERVER_*-Bausteine basiert er auf der generischen GEN_SERVER-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt QI und ID.
- **RSP**: Sendet Daten an den Client, trägt QI sowie SD_1 SD_2.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt QO und STATUS.
- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt QO, STATUS.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (TRUE = Verbindung öffnen, FALSE = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit RSP an den Client übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit RSP an den Client übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- Keine Empfangsdaten (SERVER_2_0 erhält keine Nutzdaten über IND, nur Verbindungs-/Statusinformationen).

Funktionsweise

SERVER_2_0 initialisiert über INIT eine Verbindung zum passenden CLIENT_0_2-Block (bei QI = TRUE) bzw. schließt sie (bei QI = FALSE); der Abschluss wird über INITO bestätigt. Über RSP werden SD_1 SD_2 an den Client übertragen. Sobald eine Antwort vorliegen, löst der Baustein IND aus.

Technische Besonderheiten

- **Generische Implementierung**: eclipse4diac::core::GenericClassName = 'GEN_SERVER', dieselbe C++-Basis wie alle anderen SERVER_*-Varianten; Anzahl und Typ (ANY) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **ANY-Datenfelder**: Alle SD_i/RD_i sind generisch (ANY) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück CLIENT_0_2**: SERVER_2_0 ist funktional nur mit einem CLIENT_0_2-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, QO = FALSE.
2. **Verbindungsaufbau**: INIT mit QI = TRUE wird verarbeitet.
3. **Verbunden**: INITO mit QO = TRUE bestätigt die Verbindung.
4. **Datenaustausch**: RSP/IND-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: INIT mit QI = FALSE wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 2 Werte gesendet und 0 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als CLIENT_0_2 mit passender Feldanzahl ausgelegt ist.

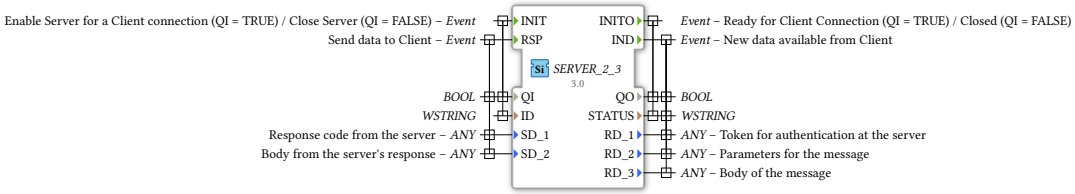
⚖️ Vergleich mit ähnlichen Bausteinen

- CLIENT_0_2: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- CLIENT_1 / SERVER_1: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

SERVER_2_0 liefert die generische, auf 2 Sende- und 0 Empfangsfelder zugeschnittene Server-Variante der GEN_SERVER-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.66 SERVER_2_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der SERVER_2_3-Funktionsblock ist die generische Server-Variante mit 2 Sende- und 3 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_3_2-Block. Er überträgt 2 Datenwerte (SD_1 SD_2) an den Client und empfängt 3 Datenwerte (RD_1 RD_2 RD_3) zurück. Wie alle SERVER_*-Bausteine basiert er auf der generischen GEN_SERVER-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt QI und ID.
- **RSP**: Sendet Daten an den Client, trägt QI sowie SD_1 SD_2.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt QO und STATUS.
- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt QO, STATUS sowie RD_1 RD_2 RD_3.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (TRUE = Verbindung öffnen, FALSE = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit RSP an den Client übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit RSP an den Client übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit IND vom Client geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit IND vom Client geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit IND vom Client geliefert.

Funktionsweise

SERVER_2_3 initialisiert über INIT eine Verbindung zum passenden CLIENT_3_2-Block (bei QI = TRUE) bzw. schließt sie (bei QI = FALSE); der Abschluss wird über INITO bestätigt. Über RSP werden SD_1 SD_2 an den Client übertragen. Sobald Antwortdaten (RD_1 RD_2 RD_3) vorliegen, löst der Baustein IND aus und stellt RD_1 RD_2 RD_3 bereit.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, dieselbe C++-Basis wie alle anderen `SERVER_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `CLIENT_3_2`**: `SERVER_2_3` ist funktional nur mit einem `CLIENT_3_2`-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `RSP`/`IND`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 2 Werte gesendet und 3 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als `CLIENT_3_2` mit passender Feldanzahl ausgelegt ist.

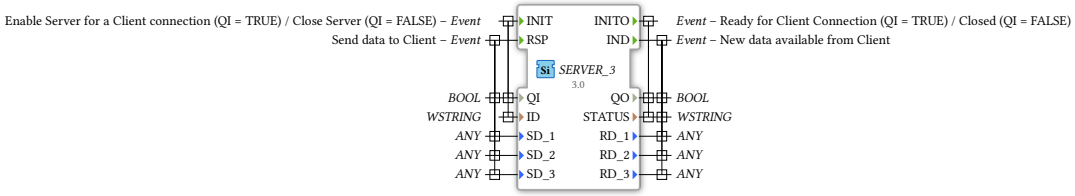
⚖️ Vergleich mit ähnlichen Bausteinen

- **CLIENT_3_2**: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`SERVER_2_3` liefert die generische, auf 2 Sende- und 3 Empfangsfelder zugeschnittene Server-Variante der `GEN_SERVER`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.67 SERVER_3



• – ○ • – ○ • – ○ •

Einleitung

Der `SERVER_3`-Funktionsblock ist die generische Server-Variante mit 3 Sende- und 3 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_3-Block. Er überträgt 3 Datenwerte (`SD_1` `SD_2` `SD_3`) an den Client und empfängt 3 Datenwerte (`RD_1` `RD_2` `RD_3`) zurück. Wie alle `SERVER_*`-Bausteine basiert er auf der generischen `GEN_SERVER`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt `QI` und `ID`.
- **RSP**: Sendet Daten an den Client, trägt `QI` sowie `SD_1` `SD_2` `SD_3`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1` `RD_2` `RD_3`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit `RSP` an den Client übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `RSP` an den Client übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit `RSP` an den Client übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `IND` vom Client geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `IND` vom Client geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit `IND` vom Client geliefert.

Funktionsweise

`SERVER_3` initialisiert über `INIT` eine Verbindung zum passenden `CLIENT_3`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `RSP` werden `SD_1` `SD_2` `SD_3` an den Client übertragen. Sobald Antwortdaten (`RD_1` `RD_2` `RD_3`) vorliegen, löst der Baustein `IND` aus und stellt `RD_1` `RD_2` `RD_3` bereit.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, dieselbe C++-Basis wie alle anderen `SERVER_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `CLIENT_3`**: `SERVER_3` ist funktional nur mit einem `CLIENT_3`-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `RSP`/`IND`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 3 Werte gesendet und 3 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als `CLIENT_3` mit passender Feldanzahl ausgelegt ist.

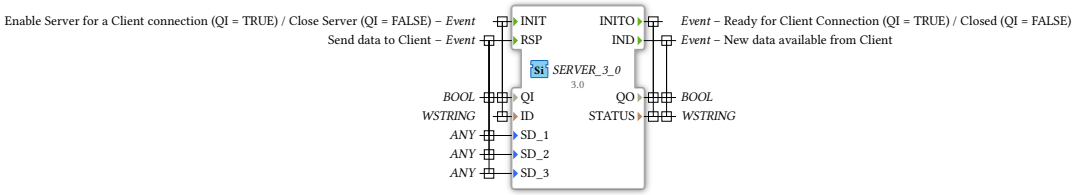
⚖️ Vergleich mit ähnlichen Bausteinen

- **CLIENT_3**: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`SERVER_3` liefert die generische, auf 3 Sende- und 3 Empfangsfelder zugeschnittene Server-Variante der `GEN_SERVER`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.68 SERVER_3_0



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `SERVER_3_0`-Funktionsblock ist die generische Server-Variante mit 3 Sende- und 0 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_0_3-Block. Er überträgt 3 Datenwerte (`SD_1` `SD_2` `SD_3`) an den Client, ohne Rückgabedaten zu empfangen. Wie alle `SERVER_*`-Bausteine basiert er auf der generischen `GEN_SERVER`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt `QI` und `ID`.
- **RSP**: Sendet Daten an den Client, trägt `QI` sowie `SD_1` `SD_2` `SD_3`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt `QO`, `STATUS`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit `RSP` an den Client übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `RSP` an den Client übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit `RSP` an den Client übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- Keine Empfangsdaten (`SERVER_3_0` erhält keine Nutzdaten über `IND`, nur Verbindungs-/Statusinformationen).

Funktionsweise

`SERVER_3_0` initialisiert über `INIT` eine Verbindung zum passenden `CLIENT_0_3`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `RSP` werden `SD_1` `SD_2` `SD_3` an den Client übertragen. Sobald eine Antwort vorliegen, löst der Baustein `IND` aus.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, dieselbe C++-Basis wie alle anderen `SERVER_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `CLIENT_0_3`**: `SERVER_3_0` ist funktional nur mit einem `CLIENT_0_3`-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.

2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `RSP`/`IND`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 3 Werte gesendet und 0 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als `CLIENT_0_3` mit passender Feldanzahl ausgelegt ist.

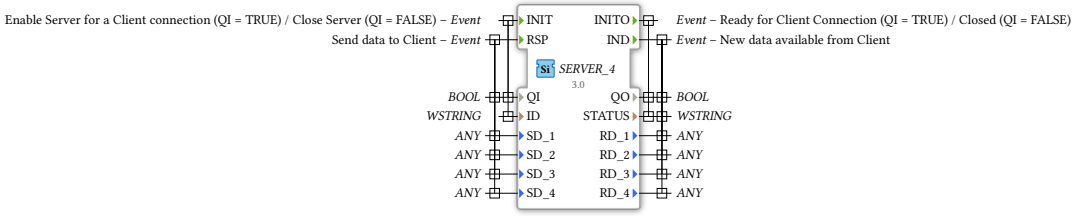
⚖️ Vergleich mit ähnlichen Bausteinen

- **CLIENT_0_3**: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`SERVER_3_0` liefert die generische, auf 3 Sende- und 0 Empfangsfelder zugeschnittene Server-Variante der `GEN_SERVER`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.69 SERVER_4



• – ○ • – ○ • – ○ •

Einleitung

Der `SERVER_4`-Funktionsblock ist die generische Server-Variante mit 4 Sende- und 4 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_4-Block. Er überträgt 4 Datenwerte (`SD_1 SD_2 SD_3 SD_4`) an den Client und empfängt 4 Datenwerte (`RD_1 RD_2 RD_3 RD_4`) zurück. Wie alle `SERVER_*`-Bausteine basiert er auf der generischen `GEN_SERVER`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt `QI` und `ID`.
- **RSP**: Sendet Daten an den Client, trägt `QI` sowie `SD_1 SD_2 SD_3 SD_4`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1 RD_2 RD_3 RD_4`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit `RSP` an den Client übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `RSP` an den Client übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit `RSP` an den Client übertragen.

- **SD_4** (ANY): Sendedatum 4, wird mit `RSP` an den Client übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `IND` vom Client geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `IND` vom Client geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit `IND` vom Client geliefert.
- **RD_4** (ANY): Empfangsdatum 4, wird mit `IND` vom Client geliefert.

Funktionsweise

`SERVER_4` initialisiert über `INIT` eine Verbindung zum passenden `CLIENT_4`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `RSP` werden `SD_1` `SD_2` `SD_3` `SD_4` an den Client übertragen. Sobald Antwortdaten (`RD_1` `RD_2` `RD_3` `RD_4`) vorliegen, löst der Baustein `IND` aus und stellt `RD_1` `RD_2` `RD_3` `RD_4` bereit.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, dieselbe C++-Basis wie alle anderen `SERVER_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `CLIENT_4`**: `SERVER_4` ist funktional nur mit einem `CLIENT_4`-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `RSP`/`IND`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 4 Werte gesendet und 4 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als `CLIENT_4` mit passender Feldanzahl ausgelegt ist.

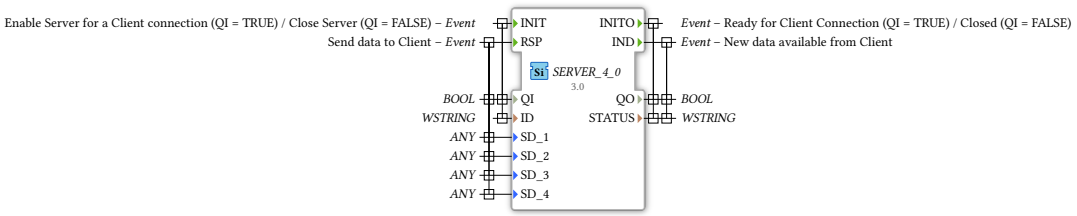
⚖️ Vergleich mit ähnlichen Bausteinen

- **CLIENT_4**: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`SERVER_4` liefert die generische, auf 4 Sende- und 4 Empfangsfelder zugeschnittene Server-Variante der `GEN_SERVER`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.70 SERVER_4_0



• – ◦ • – ◦ • – ◦ •

Einleitung

Der SERVER_4_0-Funktionsblock ist die generische Server-Variante mit 4 Sende- und 0 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_0_4-Block. Er überträgt 4 Datenwerte (SD_1 SD_2 SD_3 SD_4) an den Client, ohne Rückgabedaten zu empfangen. Wie alle SERVER_*-Bausteine basiert er auf der generischen GEN_SERVER-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt QI und ID.
- **RSP**: Sendet Daten an den Client, trägt QI sowie SD_1 SD_2 SD_3 SD_4.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt QO und STATUS.
- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt QO, STATUS.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (TRUE = Verbindung öffnen, FALSE = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit RSP an den Client übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit RSP an den Client übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit RSP an den Client übertragen.
- **SD_4** (ANY): Sendedatum 4, wird mit RSP an den Client übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- Keine Empfangsdaten (SERVER_4_0 erhält keine Nutzdaten über IND, nur Verbindungs-/Statusinformationen).

Funktionsweise

SERVER_4_0 initialisiert über INIT eine Verbindung zum passenden CLIENT_0_4-Block (bei QI = TRUE) bzw. schließt sie (bei QI = FALSE); der Abschluss wird über INITO bestätigt. Über RSP werden SD_1 SD_2 SD_3 SD_4 an den Client übertragen. Sobald eine Antwort vorliegen, löst der Baustein IND aus.

Technische Besonderheiten

- **Generische Implementierung**: eclipse4diac::core::GenericClassName = 'GEN_SERVER', dieselbe C++-Basis wie alle anderen SERVER_*-Varianten; Anzahl und Typ (ANY) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **ANY-Datenfelder**: Alle SD_i/RD_i sind generisch (ANY) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück CLIENT_0_4**: SERVER_4_0 ist funktional nur mit einem CLIENT_0_4-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, QO = FALSE.
2. **Verbindungsaufbau**: INIT mit QI = TRUE wird verarbeitet.
3. **Verbunden**: INITO mit QO = TRUE bestätigt die Verbindung.
4. **Datenaustausch**: RSP/IND-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: INIT mit QI = FALSE wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 4 Werte gesendet und 0 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als CLIENT_0_4 mit passender Feldanzahl ausgelegt ist.

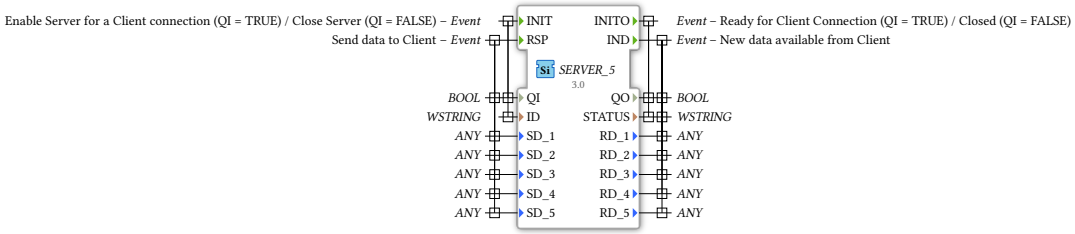
⚖️ Vergleich mit ähnlichen Bausteinen

- **CLIENT_0_4**: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`SERVER_4_0` liefert die generische, auf 4 Sende- und 0 Empfangsfelder zugeschnittene Server-Variante der `GEN_SERVER`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.71 SERVER_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der `SERVER_5`-Funktionsblock ist die generische Server-Variante mit 5 Sende- und 5 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_5-Block. Er überträgt 5 Datenwerte (`SD_1` `SD_2` `SD_3` `SD_4` `SD_5`) an den Client und empfängt 5 Datenwerte (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5`) zurück. Wie alle `SERVER_*`-Bausteine basiert er auf der generischen `GEN_SERVER`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt `QI` und `ID`.
- **RSP**: Sendet Daten an den Client, trägt `QI` sowie `SD_1` `SD_2` `SD_3` `SD_4` `SD_5`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1` `RD_2` `RD_3` `RD_4` `RD_5`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit `RSP` an den Client übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `RSP` an den Client übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit `RSP` an den Client übertragen.
- **SD_4** (ANY): Sendedatum 4, wird mit `RSP` an den Client übertragen.
- **SD_5** (ANY): Sendedatum 5, wird mit `RSP` an den Client übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `IND` vom Client geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `IND` vom Client geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit `IND` vom Client geliefert.
- **RD_4** (ANY): Empfangsdatum 4, wird mit `IND` vom Client geliefert.
- **RD_5** (ANY): Empfangsdatum 5, wird mit `IND` vom Client geliefert.

Funktionsweise

SERVER_5 initialisiert über INIT eine Verbindung zum passenden CLIENT_5-Block (bei QI = TRUE) bzw. schließt sie (bei QI = FALSE); der Abschluss wird über INITO bestätigt. Über RSP werden SD_1 SD_2 SD_3 SD_4 SD_5 an den Client übertragen. Sobald Antwortdaten (RD_1 RD_2 RD_3 RD_4 RD_5) vorliegen, löst der Baustein IND aus und stellt RD_1 RD_2 RD_3 RD_4 RD_5 bereit.

Technische Besonderheiten

- **Generische Implementierung**: eclipse4diac::core::GenericClassName = 'GEN_SERVER', dieselbe C++-Basis wie alle anderen SERVER_*-Varianten; Anzahl und Typ (ANY) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **ANY-Datenfelder**: Alle SD_i/RD_i sind generisch (ANY) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück CLIENT_5**: SERVER_5 ist funktional nur mit einem CLIENT_5-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, QO = FALSE.
2. **Verbindungsaufbau**: INIT mit QI = TRUE wird verarbeitet.
3. **Verbunden**: INITO mit QO = TRUE bestätigt die Verbindung.
4. **Datenaustausch**: RSP/IND-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: INIT mit QI = FALSE wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 5 Werte gesendet und 5 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als CLIENT_5 mit passender Feldanzahl ausgelegt ist.

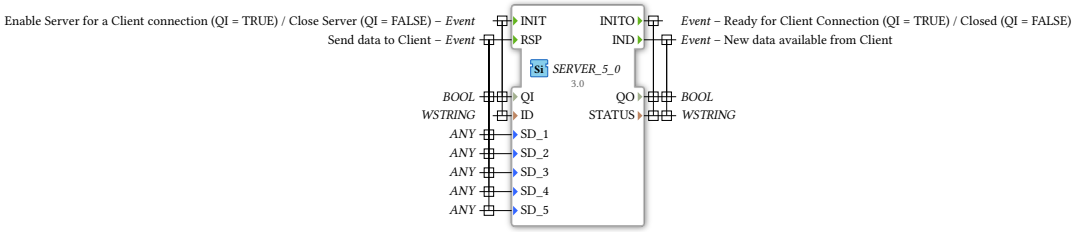
⚖️ Vergleich mit ähnlichen Bausteinen

- CLIENT_5: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- CLIENT_1 / SERVER_1: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

SERVER_5 liefert die generische, auf 5 Sende- und 5 Empfangsfelder zugeschnittene Server-Variante der GEN_SERVER-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.72 SERVER_5_0



• – ○ • – ○ • – ○ •

Einleitung

Der SERVER_5_0-Funktionsblock ist die generische Server-Variante mit 5 Sende- und 0 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_0_5-Block. Er überträgt 5 Datenwerte (SD_1 SD_2 SD_3 SD_4 SD_5) an den Client, ohne Rückgabedaten zu empfangen. Wie alle SERVER_*-Bausteine basiert er auf der generischen GEN_SERVER-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge
- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt `QI` und `ID`.
- **RSP**: Sendet Daten an den Client, trägt `QI` sowie `SD_1` `SD_2` `SD_3` `SD_4` `SD_5`.

Ereignis-Ausgänge
- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt `QO`, `STATUS`.

Daten-Eingänge
- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit `RSP` an den Client übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `RSP` an den Client übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit `RSP` an den Client übertragen.
- **SD_4** (ANY): Sendedatum 4, wird mit `RSP` an den Client übertragen.
- **SD_5** (ANY): Sendedatum 5, wird mit `RSP` an den Client übertragen.

Daten-Ausgänge
- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- Keine Empfangsdaten (`SERVER_5_0` erhält keine Nutzdaten über `IND`, nur Verbindungs-/Statusinformationen).

Funktionsweise
`SERVER_5_0` initialisiert über `INIT` eine Verbindung zum passenden `CLIENT_0_5`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `RSP` werden `SD_1` `SD_2` `SD_3` `SD_4` `SD_5` an den Client übertragen. Sobald eine Antwort vorliegen, löst der Baustein `IND` aus.

Technische Besonderheiten
- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, dieselbe C++-Basis wie alle anderen `SERVER_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `CLIENT_0_5`**: `SERVER_5_0` ist funktional nur mit einem `CLIENT_0_5`-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht
1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `RSP`/`IND`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

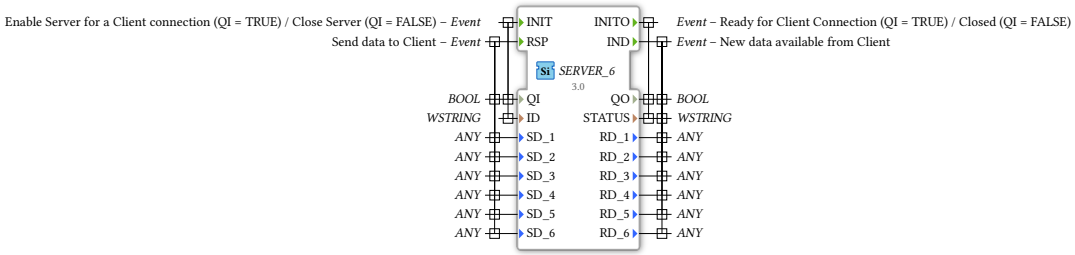
Anwendungsszenarien
- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 5 Werte gesendet und 0 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als `CLIENT_0_5` mit passender Feldanzahl ausgelegt ist.

⚖️ Vergleich mit ähnlichen Bausteinen
- **CLIENT_0_5**: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

SERVER_5_0 liefert die generische, auf 5 Sende- und 0 Empfangsfelder zugeschnittene Server-Variante der GEN_SERVER-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.73 SERVER_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der SERVER_6-Funktionsblock ist die generische Server-Variante mit 6 Sende- und 6 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_6-Block. Er überträgt 6 Datenwerte (SD_1 SD_2 SD_3 SD_4 SD_5 SD_6) an den Client und empfängt 6 Datenwerte (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6) zurück. Wie alle SERVER_*-Bausteine basiert er auf der generischen GEN_SERVER-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt QI und ID.
- **RSP**: Sendet Daten an den Client, trägt QI sowie SD_1 SD_2 SD_3 SD_4 SD_5 SD_6.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt QO und STATUS.
- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt QO, STATUS sowie RD_1 RD_2 RD_3 RD_4 RD_5 RD_6.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (TRUE = Verbindung öffnen, FALSE = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit RSP an den Client übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit RSP an den Client übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit RSP an den Client übertragen.
- **SD_4** (ANY): Sendedatum 4, wird mit RSP an den Client übertragen.
- **SD_5** (ANY): Sendedatum 5, wird mit RSP an den Client übertragen.
- **SD_6** (ANY): Sendedatum 6, wird mit RSP an den Client übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit IND vom Client geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit IND vom Client geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit IND vom Client geliefert.
- **RD_4** (ANY): Empfangsdatum 4, wird mit IND vom Client geliefert.
- **RD_5** (ANY): Empfangsdatum 5, wird mit IND vom Client geliefert.
- **RD_6** (ANY): Empfangsdatum 6, wird mit IND vom Client geliefert.

Funktionsweise

SERVER_6 initialisiert über INIT eine Verbindung zum passenden CLIENT_6-Block (bei QI = TRUE) bzw. schließt sie (bei QI = FALSE); der Abschluss wird über INITO bestätigt. Über RSP werden SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 an den Client übertragen. Sobald Antwortdaten (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6) vorliegen, löst der Baustein IND aus und stellt RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 bereit.

Technische Besonderheiten

- **Generische Implementierung**: eclipse4diac::core::GenericClassName = 'GEN_SERVER', dieselbe C++-Basis wie alle anderen SERVER_*-Varianten; Anzahl und Typ (ANY) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **ANY-Datenfelder**: Alle SD_i/RD_i sind generisch (ANY) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück CLIENT_6**: SERVER_6 ist funktional nur mit einem CLIENT_6-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, QO = FALSE.
2. **Verbindungsaufbau**: INIT mit QI = TRUE wird verarbeitet.
3. **Verbunden**: INITO mit QO = TRUE bestätigt die Verbindung.
4. **Datenaustausch**: RSP/IND-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: INIT mit QI = FALSE wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 6 Werte gesendet und 6 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als CLIENT_6 mit passender Feldanzahl ausgelegt ist.

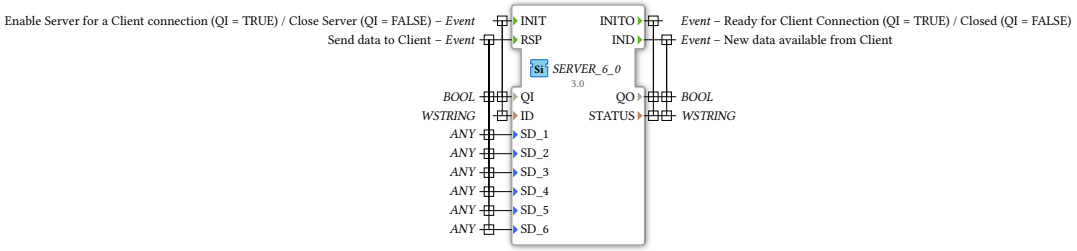
⚖️ Vergleich mit ähnlichen Bausteinen

- CLIENT_6: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- CLIENT_1 / SERVER_1: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

SERVER_6 liefert die generische, auf 6 Sende- und 6 Empfangsfelder zugeschnittene Server-Variante der GEN_SERVER-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.74 SERVER_6_0



• – ○ • – ○ • – ○ •

Einleitung

Der SERVER_6_0-Funktionsblock ist die generische Server-Variante mit 6 Sende- und 0 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_0_6-Block. Er überträgt 6 Datenwerte (SD_1 SD_2 SD_3 SD_4 SD_5 SD_6) an den Client, ohne Rückgabedaten zu empfangen. Wie alle SERVER_*-Bausteine basiert er auf der generischen GEN_SERVER-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt `QI` und `ID`.
- **RSP**: Sendet Daten an den Client, trägt `QI` sowie `SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt `QO`, `STATUS`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit `RSP` an den Client übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `RSP` an den Client übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit `RSP` an den Client übertragen.
- **SD_4** (ANY): Sendedatum 4, wird mit `RSP` an den Client übertragen.
- **SD_5** (ANY): Sendedatum 5, wird mit `RSP` an den Client übertragen.
- **SD_6** (ANY): Sendedatum 6, wird mit `RSP` an den Client übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- Keine Empfangsdaten (`SERVER_6_0` erhält keine Nutzdaten über `IND`, nur Verbindungs-/Statusinformationen).

Funktionsweise

`SERVER_6_0` initialisiert über `INIT` eine Verbindung zum passenden `CLIENT_0_6`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `RSP` werden `SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` an den Client übertragen. Sobald eine Antwort vorliegen, löst der Baustein `IND` aus.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, dieselbe C++-Basis wie alle anderen `SERVER_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `CLIENT_0_6`**: `SERVER_6_0` ist funktional nur mit einem `CLIENT_0_6`-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `RSP`/`IND`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 6 Werte gesendet und 0 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als `CLIENT_0_6` mit passender Feldanzahl ausgelegt ist.

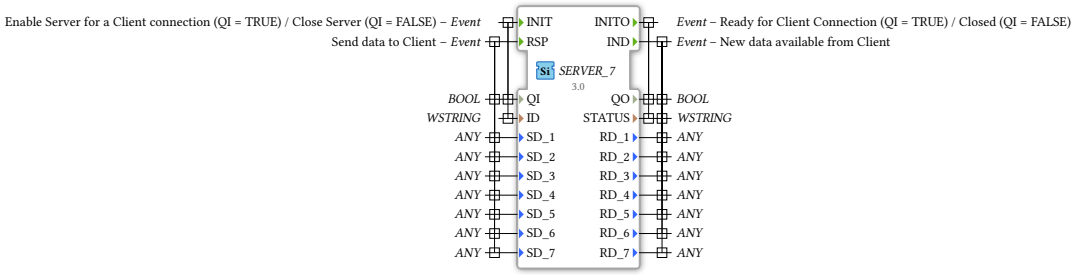
⚖️ Vergleich mit ähnlichen Bausteinen

- **CLIENT_0_6**: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`SERVER_6_0` liefert die generische, auf 6 Sende- und 0 Empfangsfelder zugeschnittene Server-Variante der `GEN_SERVER`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.75 SERVER_7



• – ○ • – ○ • – ○ •

Einleitung

Der `SERVER_7`-Funktionsblock ist die generische Server-Variante mit 7 Sende- und 7 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_7-Block. Er überträgt 7 Datenwerte (`SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7`) an den Client und empfängt 7 Datenwerte (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7`) zurück. Wie alle `SERVER_*`-Bausteine basiert er auf der generischen `GEN_SERVER`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt `QI` und `ID`.
- **RSP**: Sendet Daten an den Client, trägt `QI` sowie `SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit `RSP` an den Client übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `RSP` an den Client übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit `RSP` an den Client übertragen.
- **SD_4** (ANY): Sendedatum 4, wird mit `RSP` an den Client übertragen.
- **SD_5** (ANY): Sendedatum 5, wird mit `RSP` an den Client übertragen.
- **SD_6** (ANY): Sendedatum 6, wird mit `RSP` an den Client übertragen.
- **SD_7** (ANY): Sendedatum 7, wird mit `RSP` an den Client übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `IND` vom Client geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `IND` vom Client geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit `IND` vom Client geliefert.
- **RD_4** (ANY): Empfangsdatum 4, wird mit `IND` vom Client geliefert.
- **RD_5** (ANY): Empfangsdatum 5, wird mit `IND` vom Client geliefert.
- **RD_6** (ANY): Empfangsdatum 6, wird mit `IND` vom Client geliefert.
- **RD_7** (ANY): Empfangsdatum 7, wird mit `IND` vom Client geliefert.

Funktionsweise

SERVER_7 initialisiert über INIT eine Verbindung zum passenden CLIENT_7-Block (bei QI = TRUE) bzw. schließt sie (bei QI = FALSE); der Abschluss wird über INITO bestätigt. Über RSP werden SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 an den Client übertragen. Sobald Antwortdaten (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7) vorliegen, löst der Baustein IND aus und stellt RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 bereit.

Technische Besonderheiten

- **Generische Implementierung**: eclipse4diac::core::GenericClassName = 'GEN_SERVER', dieselbe C++-Basis wie alle anderen SERVER_*-Varianten; Anzahl und Typ (ANY) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **ANY-Datenfelder**: Alle SD_i/RD_i sind generisch (ANY) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück CLIENT_7**: SERVER_7 ist funktional nur mit einem CLIENT_7-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, QO = FALSE.
2. **Verbindungsaufbau**: INIT mit QI = TRUE wird verarbeitet.
3. **Verbunden**: INITO mit QO = TRUE bestätigt die Verbindung.
4. **Datenaustausch**: RSP/IND-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: INIT mit QI = FALSE wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 7 Werte gesendet und 7 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als CLIENT_7 mit passender Feldanzahl ausgelegt ist.

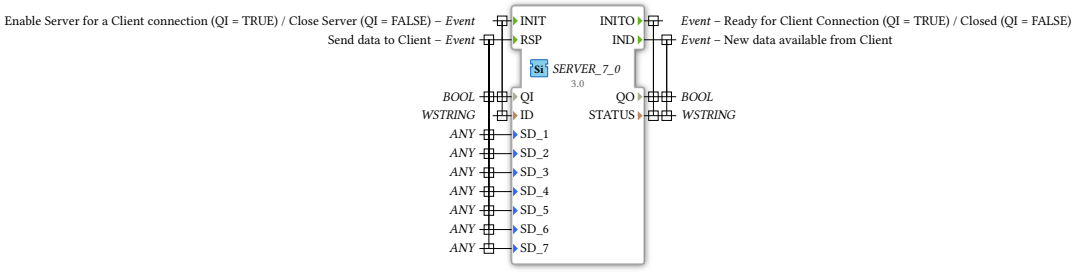
⚖️ Vergleich mit ähnlichen Bausteinen

- CLIENT_7: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- CLIENT_1 / SERVER_1: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

SERVER_7 liefert die generische, auf 7 Sende- und 7 Empfangsfelder zugeschnittene Server-Variante der GEN_SERVER-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.76 SERVER_7_0



• – ○ • – ○ • – ○ •

Einleitung

Der SERVER_7_0-Funktionsblock ist die generische Server-Variante mit 7 Sende- und 0 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_0_7-Block. Er überträgt 7 Datenwerte (SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7) an den Client, ohne Rückgabedaten zu empfangen. Wie alle SERVER_*-Bausteine basiert er auf der generischen GEN_SERVER-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt `QI` und `ID`.
- **RSP**: Sendet Daten an den Client, trägt `QI` sowie `SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt `QO`, `STATUS`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit `RSP` an den Client übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `RSP` an den Client übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit `RSP` an den Client übertragen.
- **SD_4** (ANY): Sendedatum 4, wird mit `RSP` an den Client übertragen.
- **SD_5** (ANY): Sendedatum 5, wird mit `RSP` an den Client übertragen.
- **SD_6** (ANY): Sendedatum 6, wird mit `RSP` an den Client übertragen.
- **SD_7** (ANY): Sendedatum 7, wird mit `RSP` an den Client übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- Keine Empfangsdaten (`SERVER_7_0` erhält keine Nutzdaten über `IND`, nur Verbindungs-/Statusinformationen).

Funktionsweise

`SERVER_7_0` initialisiert über `INIT` eine Verbindung zum passenden `CLIENT_0_7`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `RSP` werden `SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7` an den Client übertragen. Sobald eine Antwort vorliegen, löst der Baustein `IND` aus.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, dieselbe C++-Basis wie alle anderen `SERVER_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `CLIENT_0_7`**: `SERVER_7_0` ist funktional nur mit einem `CLIENT_0_7`-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `RSP`/`IND`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 7 Werte gesendet und 0 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als `CLIENT_0_7` mit passender Feldanzahl ausgelegt ist.

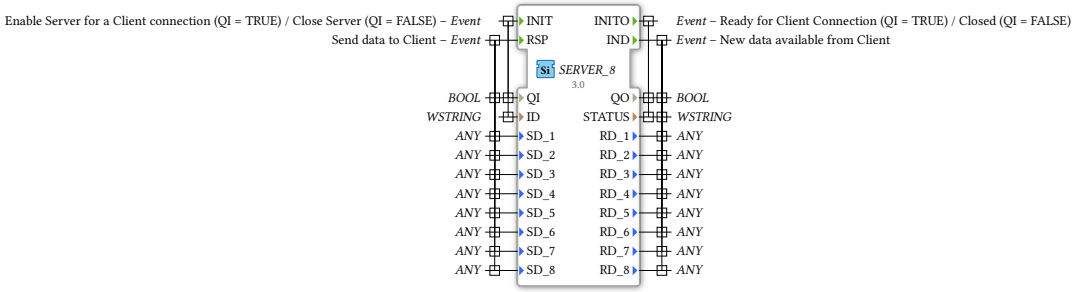
⚖️ Vergleich mit ähnlichen Bausteinen

- CLIENT_0_7: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- CLIENT_1 / SERVER_1: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

SERVER_7_0 liefert die generische, auf 7 Sende- und 0 Empfangsfelder zugeschnittene Server-Variante der GEN_SERVER-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.77 SERVER_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der SERVER_8-Funktionsblock ist die generische Server-Variante mit 8 Sende- und 8 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_8-Block. Er überträgt 8 Datenwerte (SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8) an den Client und empfängt 8 Datenwerte (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8) zurück. Wie alle SERVER_*-Bausteine basiert er auf der generischen GEN_SERVER-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt QI und ID.
- **RSP**: Sendet Daten an den Client, trägt QI sowie SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt QO und STATUS.
- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt QO, STATUS sowie RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (TRUE = Verbindung öffnen, FALSE = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit RSP an den Client übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit RSP an den Client übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit RSP an den Client übertragen.
- **SD_4** (ANY): Sendedatum 4, wird mit RSP an den Client übertragen.
- **SD_5** (ANY): Sendedatum 5, wird mit RSP an den Client übertragen.
- **SD_6** (ANY): Sendedatum 6, wird mit RSP an den Client übertragen.
- **SD_7** (ANY): Sendedatum 7, wird mit RSP an den Client übertragen.
- **SD_8** (ANY): Sendedatum 8, wird mit RSP an den Client übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.

- **RD_1** (ANY): Empfangsdatum 1, wird mit IND vom Client geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit IND vom Client geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit IND vom Client geliefert.
- **RD_4** (ANY): Empfangsdatum 4, wird mit IND vom Client geliefert.
- **RD_5** (ANY): Empfangsdatum 5, wird mit IND vom Client geliefert.
- **RD_6** (ANY): Empfangsdatum 6, wird mit IND vom Client geliefert.
- **RD_7** (ANY): Empfangsdatum 7, wird mit IND vom Client geliefert.
- **RD_8** (ANY): Empfangsdatum 8, wird mit IND vom Client geliefert.

Funktionsweise

SERVER_8 initialisiert über INIT eine Verbindung zum passenden CLIENT_8-Block (bei QI = TRUE) bzw. schließt sie (bei QI = FALSE); der Abschluss wird über INITO bestätigt. Über RSP werden SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8 an den Client übertragen. Sobald Antwortdaten (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8) vorliegen, löst der Baustein IND aus und stellt RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 bereit.

Technische Besonderheiten

- **Generische Implementierung**: eclipse4diac::core::GenericClassName = 'GEN_SERVER', dieselbe C++-Basis wie alle anderen SERVER_*-Varianten; Anzahl und Typ (ANY) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **ANY-Datenfelder**: Alle SD_i/RD_i sind generisch (ANY) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück CLIENT_8**: SERVER_8 ist funktional nur mit einem CLIENT_8-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, QO = FALSE.
2. **Verbindungsaufbau**: INIT mit QI = TRUE wird verarbeitet.
3. **Verbunden**: INITO mit QO = TRUE bestätigt die Verbindung.
4. **Datenaustausch**: RSP/IND-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: INIT mit QI = FALSE wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 8 Werte gesendet und 8 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als CLIENT_8 mit passender Feldanzahl ausgelegt ist.

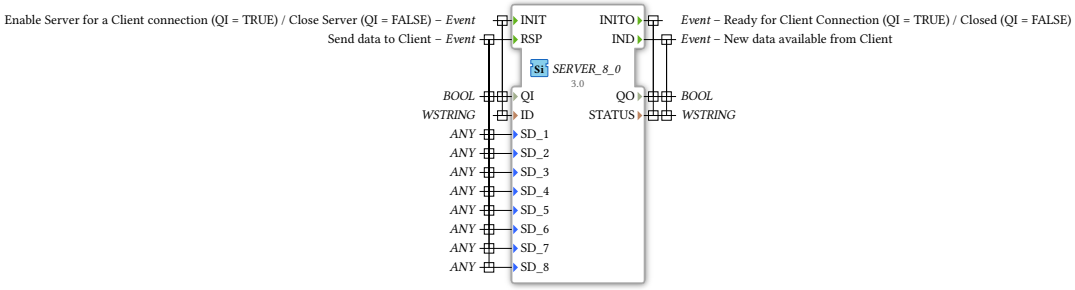
⚖️ Vergleich mit ähnlichen Bausteinen

- **CLIENT_8**: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

SERVER_8 liefert die generische, auf 8 Sende- und 8 Empfangsfelder zugeschnittene Server-Variante der GEN_SERVER-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.78 SERVER_8_0



• – ○ • – ○ • – ○ •

Einleitung

Der `SERVER_8_0`-Funktionsblock ist die generische Server-Variante mit 8 Sende- und 0 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_0_8-Block. Er überträgt 8 Datenwerte (`SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8`) an den Client, ohne Rückgabedaten zu empfangen. Wie alle `SERVER_*`-Bausteine basiert er auf der generischen `GEN_SERVER`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt `QI` und `ID`.
- **RSP**: Sendet Daten an den Client, trägt `QI` sowie `SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt `QO`, `STATUS`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit `RSP` an den Client übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `RSP` an den Client übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit `RSP` an den Client übertragen.
- **SD_4** (ANY): Sendedatum 4, wird mit `RSP` an den Client übertragen.
- **SD_5** (ANY): Sendedatum 5, wird mit `RSP` an den Client übertragen.
- **SD_6** (ANY): Sendedatum 6, wird mit `RSP` an den Client übertragen.
- **SD_7** (ANY): Sendedatum 7, wird mit `RSP` an den Client übertragen.
- **SD_8** (ANY): Sendedatum 8, wird mit `RSP` an den Client übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- Keine Empfangsdaten (`SERVER_8_0` erhält keine Nutzdaten über `IND`, nur Verbindungs-/Statusinformationen).

Funktionsweise

`SERVER_8_0` initialisiert über `INIT` eine Verbindung zum passenden `CLIENT_0_8`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `RSP` werden `SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8` an den Client übertragen. Sobald eine Antwort vorliegen, löst der Baustein `IND` aus.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, dieselbe C++-Basis wie alle anderen `SERVER_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `CLIENT_0_8`**: `SERVER_8_0` ist funktional nur mit einem `CLIENT_0_8`-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.
4. **Datenaustausch**: `RSP`/`IND`-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: `INIT` mit `QI = FALSE` wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 8 Werte gesendet und 0 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als `CLIENT_0_8` mit passender Feldanzahl ausgelegt ist.

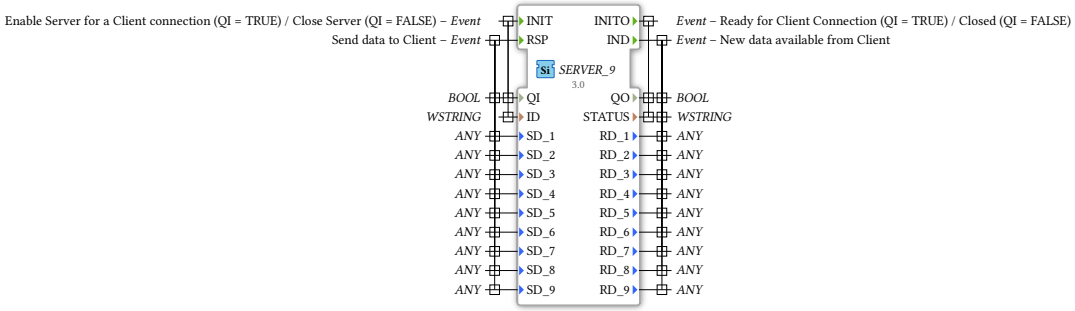
⚖️ Vergleich mit ähnlichen Bausteinen

- **CLIENT_0_8**: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- **CLIENT_1** / **SERVER_1**: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

`SERVER_8_0` liefert die generische, auf 8 Sende- und 0 Empfangsfelder zugeschnittene Server-Variante der `GEN_SERVER`-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.79 SERVER_9



• – ○ • – ○ • – ○ •

Einleitung

Der `SERVER_9`-Funktionsblock ist die generische Server-Variante mit 9 Sende- und 9 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_9-Block. Er überträgt 9 Datenwerte (`SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7` `SD_8` `SD_9`) an den Client und empfängt 9 Datenwerte (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9`) zurück. Wie alle `SERVER_*`-Bausteine basiert er auf der generischen `GEN_SERVER`-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt `QI` und `ID`.
- **RSP**: Sendet Daten an den Client, trägt `QI` sowie `SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8 SD_9`.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt `QO` und `STATUS`.
- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt `QO`, `STATUS` sowie `RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9`.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (`TRUE` = Verbindung öffnen, `FALSE` = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit `RSP` an den Client übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit `RSP` an den Client übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit `RSP` an den Client übertragen.
- **SD_4** (ANY): Sendedatum 4, wird mit `RSP` an den Client übertragen.
- **SD_5** (ANY): Sendedatum 5, wird mit `RSP` an den Client übertragen.
- **SD_6** (ANY): Sendedatum 6, wird mit `RSP` an den Client übertragen.
- **SD_7** (ANY): Sendedatum 7, wird mit `RSP` an den Client übertragen.
- **SD_8** (ANY): Sendedatum 8, wird mit `RSP` an den Client übertragen.
- **SD_9** (ANY): Sendedatum 9, wird mit `RSP` an den Client übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- **RD_1** (ANY): Empfangsdatum 1, wird mit `IND` vom Client geliefert.
- **RD_2** (ANY): Empfangsdatum 2, wird mit `IND` vom Client geliefert.
- **RD_3** (ANY): Empfangsdatum 3, wird mit `IND` vom Client geliefert.
- **RD_4** (ANY): Empfangsdatum 4, wird mit `IND` vom Client geliefert.
- **RD_5** (ANY): Empfangsdatum 5, wird mit `IND` vom Client geliefert.
- **RD_6** (ANY): Empfangsdatum 6, wird mit `IND` vom Client geliefert.
- **RD_7** (ANY): Empfangsdatum 7, wird mit `IND` vom Client geliefert.
- **RD_8** (ANY): Empfangsdatum 8, wird mit `IND` vom Client geliefert.
- **RD_9** (ANY): Empfangsdatum 9, wird mit `IND` vom Client geliefert.

Funktionsweise

`SERVER_9` initialisiert über `INIT` eine Verbindung zum passenden `CLIENT_9`-Block (bei `QI = TRUE`) bzw. schließt sie (bei `QI = FALSE`); der Abschluss wird über `INITO` bestätigt. Über `RSP` werden `SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8 SD_9` an den Client übertragen. Sobald Antwortdaten (`RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9`) vorliegen, löst der Baustein `IND` aus und stellt `RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9` bereit.

Technische Besonderheiten

- **Generische Implementierung**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, dieselbe C++-Basis wie alle anderen `SERVER_*`-Varianten; Anzahl und Typ (`ANY`) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **`ANY`-Datenfelder**: Alle `SD_i`/`RD_i` sind generisch (`ANY`) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück `CLIENT_9`**: `SERVER_9` ist funktional nur mit einem `CLIENT_9`-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, `QO = FALSE`.
2. **Verbindungsaufbau**: `INIT` mit `QI = TRUE` wird verarbeitet.
3. **Verbunden**: `INITO` mit `QO = TRUE` bestätigt die Verbindung.

4. **Datenaustausch**: RSP/IND-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: INIT mit QI = FALSE wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 9 Werte gesendet und 9 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als CLIENT_9 mit passender Feldanzahl ausgelegt ist.

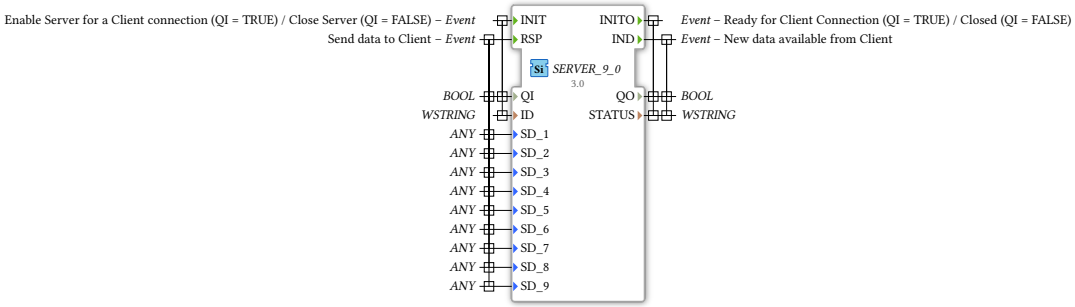
⚖️ Vergleich mit ähnlichen Bausteinen

- CLIENT_9: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- CLIENT_1 / SERVER_1: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

SERVER_9 liefert die generische, auf 9 Sende- und 9 Empfangsfelder zugeschnittene Server-Variante der GEN_SERVER-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.80 SERVER_9_0



• – ◦ • – ◦ • – ◦ •

Einleitung

Der SERVER_9_0-Funktionsblock ist die generische Server-Variante mit 9 Sende- und 0 Empfangsdatenfeldern für die Kommunikation mit einem passenden CLIENT_0_9-Block. Er überträgt 9 Datenwerte (SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8 SD_9) an den Client, ohne Rückgabedaten zu empfangen. Wie alle SERVER_*-Bausteine basiert er auf der generischen GEN_SERVER-Implementierung — dieselbe C++-Basis wie CLIENT_1/SERVER_1, lediglich die Anzahl der Sende-/Empfangsfelder unterscheidet sich pro Instanziierung.

Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Aktiviert den Server für eine Client-Verbindung (QI = TRUE) bzw. schließt ihn (QI = FALSE), trägt QI und ID.
- **RSP**: Sendet Daten an den Client, trägt QI sowie SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8 SD_9.

Ereignis-Ausgänge

- **INITO**: Bestätigt Verbindungsauf-/-abbau, trägt QO und STATUS.
- **IND**: Signalisiert, dass neue Daten vom Client verfügbar sind, trägt QO, STATUS.

Daten-Eingänge

- **QI** (BOOL): Steuert den Verbindungsstatus (TRUE = Verbindung öffnen, FALSE = Verbindung schließen).
- **ID** (WSTRING): Identifikator der Verbindung (z. B. Zieladresse/Port).
- **SD_1** (ANY): Sendedatum 1, wird mit RSP an den Client übertragen.
- **SD_2** (ANY): Sendedatum 2, wird mit RSP an den Client übertragen.
- **SD_3** (ANY): Sendedatum 3, wird mit RSP an den Client übertragen.
- **SD_4** (ANY): Sendedatum 4, wird mit RSP an den Client übertragen.

- **SD_5** (ANY): Sendedatum 5, wird mit RSP an den Client übertragen.
- **SD_6** (ANY): Sendedatum 6, wird mit RSP an den Client übertragen.
- **SD_7** (ANY): Sendedatum 7, wird mit RSP an den Client übertragen.
- **SD_8** (ANY): Sendedatum 8, wird mit RSP an den Client übertragen.
- **SD_9** (ANY): Sendedatum 9, wird mit RSP an den Client übertragen.

Daten-Ausgänge

- **QO** (BOOL): Aktueller Verbindungsstatus.
- **STATUS** (WSTRING): Statusinformationen zur Verbindung.
- Keine Empfangsdaten (SERVER_9_0 erhält keine Nutzdaten über IND, nur Verbindungs-/Statusinformationen).

Funktionsweise

SERVER_9_0 initialisiert über INIT eine Verbindung zum passenden CLIENT_0_9-Block (bei QI = TRUE) bzw. schließt sie (bei QI = FALSE); der Abschluss wird über INITO bestätigt. Über RSP werden SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8 SD_9 an den Client übertragen. Sobald eine Antwort vorliegen, löst der Baustein IND aus.

Technische Besonderheiten

- **Generische Implementierung**: eclipse4diac::core::GenericClassName = 'GEN_SERVER', dieselbe C++-Basis wie alle anderen SERVER_*-Varianten; Anzahl und Typ (ANY) der Sende-/Empfangsfelder werden pro Instanziierung über die Typdefinition festgelegt.
- **ANY-Datenfelder**: Alle SD_i/RD_i sind generisch (ANY) typisiert und passen sich beim Verdrahten an den jeweils angeschlossenen Datentyp an.
- **Gegenstück CLIENT_0_9**: SERVER_9_0 ist funktional nur mit einem CLIENT_0_9-Block auf der Client-Seite kompatibel — die Anzahl der Sendefelder der einen Seite muss der Anzahl der Empfangsfelder der Gegenseite entsprechen.

Zustandsübersicht

1. **Nicht verbunden**: Initialzustand, QO = FALSE.
2. **Verbindungsaufbau**: INIT mit QI = TRUE wird verarbeitet.
3. **Verbunden**: INITO mit QO = TRUE bestätigt die Verbindung.
4. **Datenaustausch**: RSP/IND-Zyklus für Sende-/Empfangsdaten.
5. **Verbindungstrennung**: INIT mit QI = FALSE wird verarbeitet.

Anwendungsszenarien

- **Kommunikation zwischen verteilten Steuerungssystemen**, bei denen genau 9 Werte gesendet und 0 Werte empfangen werden sollen, ohne überzählige, ungenutzte Datenfelder in Kauf zu nehmen.
- **Server-seitige Anbindung** eines Kommunikationspartners, der bereits als CLIENT_0_9 mit passender Feldanzahl ausgelegt ist.

⚖️ Vergleich mit ähnlichen Bausteinen

- CLIENT_0_9: das direkte Gegenstück auf der Client-Seite, mit vertauschter Sende-/Empfangsfeldanzahl.
- CLIENT_1 / SERVER_1: dieselbe generische Implementierung mit je einem Sende- und Empfangsfeld.

Fazit

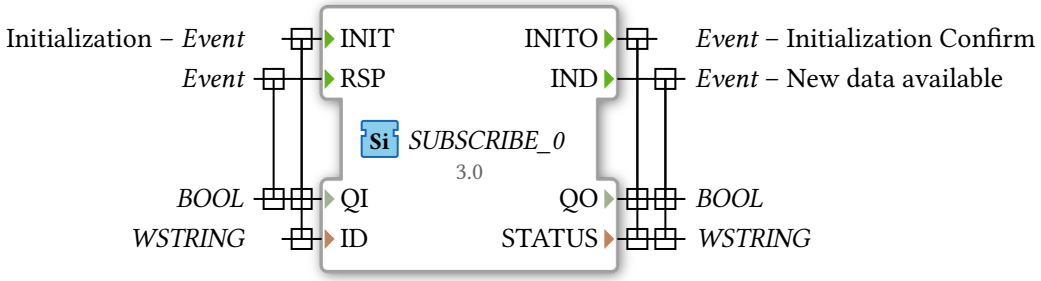
SERVER_9_0 liefert die generische, auf 9 Sende- und 0 Empfangsfelder zugeschnittene Server-Variante der GEN_SERVER-Familie und eignet sich für Netzwerkverbindungen, deren Nutzdatenanzahl von der Standardvariante mit einem Feld abweicht.

2.4.3.81 SUBSCRIBE_0

• – ◦ • – ◦ • – ◦ •

Einleitung

Der SUBSCRIBE_0 Funktionsblock dient zum Abonnieren von Daten eines PUBLISH_0 Blocks. Er ermöglicht die Kommunikation zwischen verschiedenen Komponenten in einem verteilten Automatisierungssystem, indem er Daten von einem Publisher empfängt und bei neuen Daten Ereignisse auslöst.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (Typ: EInit) - Initialisierungsereignis
  - Verknüpft mit: QI, ID
- **RSP** (Typ: Event) - Response-Ereignis
  - Verknüpft mit: QI

Ereignis-Ausgänge

- **INITO** (Typ: EInit) - Initialisierungsbestätigung
  - Verknüpft mit: QO, STATUS
- **IND** (Typ: Event) - Anzeige bei neuen verfügbaren Daten
  - Verknüpft mit: QO, STATUS

Daten-Eingänge

- **QI** (Typ: BOOL) - Qualifizierer für Initialisierung
- **ID** (Typ: WSTRING) - Identifikator für die Abonnement-Verbindung

Daten-Ausgänge

- **QO** (Typ: BOOL) - Qualifizierer für Ausgangszustand
- **STATUS** (Typ: WSTRING) - Statusinformationen

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der SUBSCRIBE_0 Block initialisiert sich über das INIT-Ereignis und stellt dabei eine Verbindung zu einem PUBLISH_0 Block her, der durch den ID-Parameter identifiziert wird. Nach erfolgreicher Initialisierung bestätigt der Block dies über das INITO-Ereignis. Bei empfangenen neuen Daten vom Publisher wird das IND-Ereignis ausgelöst, wodurch die Anwendung über verfügbare Daten informiert wird.

Technische Besonderheiten

- Verwendet WSTRING-Datentypen für ID und STATUS für internationale Zeichenunterstützung
- Implementiert ein Qualifizierungsmodell mit QI und QO für Zustandsmanagement
- Bietet Statusrückmeldungen über den STATUS-Ausgang

Zustandsübersicht

1. **Nicht initialisiert**: Ausgangszustand vor INIT
2. **Initialisierungsphase**: Während INIT-Verarbeitung
3. **Bereit**: Erfolgreich initialisiert und wartet auf Daten
4. **Datenempfang**: Verarbeitet eingehende Daten und löst IND aus

Anwendungsszenarien

- Verteile Automatisierungssysteme mit Datenverteilung
- IoT-Anwendungen mit Publisher-Subscriber-Muster
- Überwachungssysteme, die Statusdaten von verschiedenen Quellen sammeln
- Steuerungssysteme mit losgekoppelter Kommunikation zwischen Komponenten

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Kommunikationsbausteinen fokussiert sich SUBSCRIBE_0 spezifisch auf das Publisher-Subscriber-Muster und bietet eine einfache Schnittstelle für das Abonnieren von Datenströmen. Im Gegensatz zu Client-Server-Bausteinen arbeitet er asynchron und push-basiert.

Fazit

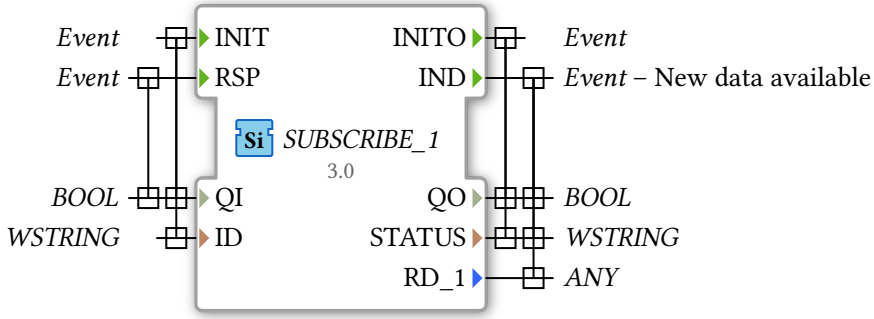
Der SUBSCRIBE_0 Funktionsblock bietet eine robuste Lösung für das Abonnieren von Daten in verteilten Automatisierungssystemen. Seine klare Schnittstelle und das etablierte Publisher-Subscriber-Muster machen ihn zu einer zuverlässigen Wahl für lose gekoppelte Kommunikationsszenarien.

2.4.3.82 SUBSCRIBE_1

• – ◦ • – ◦ • – ◦ •

Einleitung

Der SUBSCRIBE_1 Funktionsblock dient zum Abonnieren von Daten eines PUBLISH_1 Blocks. Er ermöglicht die Kommunikation zwischen verschiedenen Komponenten in einem verteilten System, indem er Daten von einem Publisher empfängt und bei neuen Daten verfügbar macht.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsereignis mit den zugehörigen Daten QI und ID
- **RSP**: Response-Ereignis mit zugehörigem Daten QI

Ereignis-Ausgänge

- **INITO**: Initialisierungsausgang mit den zugehörigen Daten QO und STATUS
- **IND**: Indikationsereignis bei neuen verfügbaren Daten mit zugehörigen Daten QO, STATUS und RD_1

Daten-Eingänge

- **QI** (BOOL): Qualifier Input - Steuert die Aktivierung des Blocks
- **ID** (WSTRING): Identifikator - Eindeutige ID für die Abonnement-Verbindung

Daten-Ausgänge

- **QO** (BOOL): Qualifier Output - Status der Blockausführung
- **STATUS** (WSTRING): Statusinformationen als Unicode-String
- **RD_1** (ANY): Empfangene Daten - Kann beliebigen Datentyp enthalten

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der SUBSCRIBE_1 Block initialisiert sich über das INIT-Ereignis und abonniert dabei Daten eines PUBLISH_1 Blocks mit der angegebenen ID. Bei erfolgreicher Initialisierung bestätigt er dies über INITO. Wenn neue Daten vom Publisher verfügbar sind, wird das IND-Ereignis ausgelöst und die empfangenen Daten über RD_1 bereitgestellt. Der STATUS-Ausgang liefert dabei Statusinformationen über den Abonnement-Vorgang.

Technische Besonderheiten

- Verwendet WSTRING für ID und STATUS für Unicode-Kompatibilität
- RD_1 unterstützt den ANY-Datentyp für maximale Flexibilität bei den empfangenen Daten
- Zwei getrennte Ereignispfade für Initialisierung und Datenempfang
- Qualifier-Eingänge (QI) und -Ausgänge (QO) für zuverlässige Zustandssteuerung

Zustandsübersicht

1. **Nicht initialisiert**: Block wartet auf INIT-Ereignis
2. **Initialisiert**: Abonnement aktiv, wartet auf Daten vom Publisher
3. **Datenempfang**: Verarbeitet eingehende Daten und triggert IND-Ereignis

Anwendungsszenarien

- Verteile Steuerungssysteme mit Datenverteilung
- IoT-Anwendungen mit Publisher-Subscriber-Architektur
- Industrie 4.0 Kommunikationsstrukturen
- Echtzeit-Datenaustausch zwischen verschiedenen Steuerungskomponenten

⚖️ **Vergleich mit ähnlichen Bausteinen**

Im Vergleich zu anderen Kommunikationsblöcken bietet SUBSCRIBE_1 eine spezialisierte Lösung für das Publisher-Subscriber-Muster mit Fokus auf Flexibilität durch den ANY-Datentyp und Unicode-Unterstützung.

Fazit

Der SUBSCRIBE_1 Funktionsblock ist ein essentieller Baustein für verteilte Systeme in 4diac, der eine robuste und flexible Kommunikation zwischen verschiedenen Komponenten ermöglicht. Seine einfache Schnittstelle und die Unterstützung beliebiger Datentypen machen ihn vielseitig einsetzbar in verschiedenen Automatisierungsanwendungen.

🌐 **Passende Themen-Unterseiten auf ms-muc-docs.de**

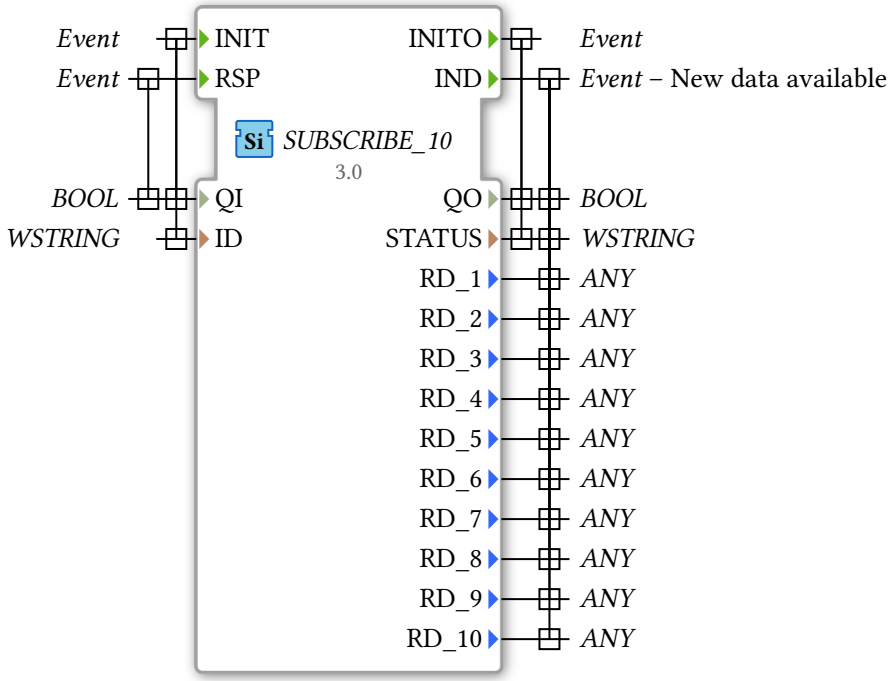
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.3.83 SUBSCRIBE_10

• – ◦ • – ◦ • – ◦ •

Einleitung

Der SUBSCRIBE_10 Funktionsblock dient als Abonnent in einem Publish-Subscribe-Kommunikationsmuster und ermöglicht das Empfangen von Daten von einem PUBLISH_10 Block. Der Block kann bis zu 10 verschiedene Datenwerte gleichzeitig empfangen und verarbeiten.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (Typ: EInit) - Initialisierungsereignis
  - Verknüpft mit: QI, ID
- **RSP** (Typ: Event) - Response-Ereignis
  - Verknüpft mit: QI

Ereignis-Ausgänge

- **INITO** (Typ: EInit) - Initialisierungsausgang
  - Verknüpft mit: QO, STATUS
- **IND** (Typ: Event) - Indikationsereignis (Neue Daten verfügbar)
  - Verknüpft mit: QO, STATUS, RD_1 bis RD_10

Daten-Eingänge

- **QI** (BOOL) - Qualifizierter Eingang (Aktivierung)
- **ID** (WSTRING) - Identifikationsstring für die Verbindung

Daten-Ausgänge

- **QO** (BOOL) - Qualifizierter Ausgang (Status)
- **STATUS** (WSTRING) - Statusinformation
- **RD_1** bis **RD_10** (ANY) - Empfangene Datenwerte 1-10

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der SUBSCRIBE_10 Block initialisiert sich über das INIT-Ereignis und verbindet sich mit einem entsprechenden PUBLISH_10 Block basierend auf der angegebenen ID. Sobald neue Daten vom Publisher verfügbar sind, wird das IND-Ereignis ausgelöst und die empfangenen Daten über die RD_1 bis RD_10 Ausgänge bereitgestellt.

Technische Besonderheiten

- Unterstützt den Datentyp ANY für alle empfangenen Daten, was maximale Flexibilität bei den Datentypen bietet

- Kann bis zu 10 verschiedene Datenwerte parallel empfangen
- Verwendet WSTRING für Status- und Identifikationsinformationen
- Implementiert ein zuverlässiges Publish-Subscribe-Kommunikationsmuster

Zustandsübersicht

1. **Nicht initialisiert**: Block ist inaktiv
2. **Initialisiert**: Verbindung zum Publisher hergestellt, wartet auf Daten
3. **Datenempfang**: Empfängt und verarbeitet eingehende Daten
4. **Fehlerzustand**: Bei Verbindungsproblemen oder Fehlern

Anwendungsszenarien

- Verteilte Steuerungssysteme
- Datenverteilung in Automatisierungsnetzwerken
- Kommunikation zwischen verschiedenen Steuerungskomponenten
- Überwachungssysteme mit mehreren Datenquellen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren Subscribe-Blöcken bietet SUBSCRIBE_10 die Möglichkeit, bis zu 10 verschiedene Datenwerte gleichzeitig zu empfangen, was ihn für komplexere Anwendungen mit mehreren Datenpunkten geeignet macht.

Fazit

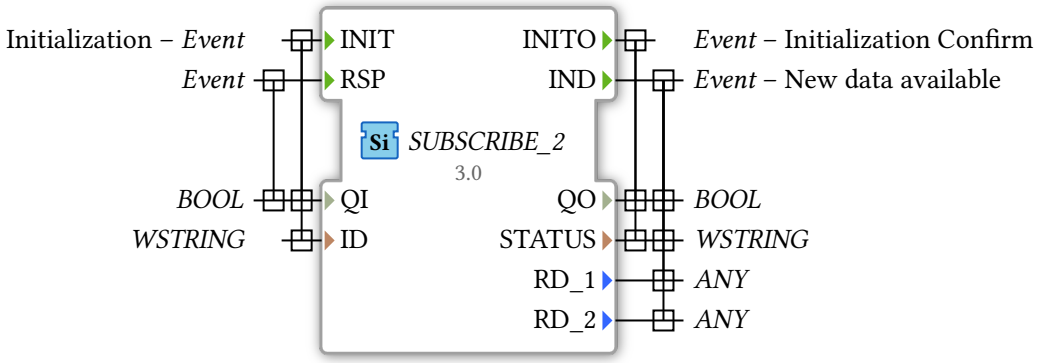
Der SUBSCRIBE_10 Funktionsblock ist ein leistungsstarkes Werkzeug für die verteilte Kommunikation in IEC 61499-Systemen. Seine Fähigkeit, bis zu 10 verschiedene Datenwerte zu verarbeiten und die flexible ANY-Datentypunterstützung machen ihn besonders geeignet für komplexe Automatisierungsanwendungen mit umfangreichen Datenaustauschanforderungen.

2.4.3.84 SUBSCRIBE_2

• – ◦ • – ◦ • – ◦ •

Einleitung

Der SUBSCRIBE_2-Funktionsblock dient als Abonnent für einen PUBLISH_2-Block und ermöglicht den Empfang von Daten über ein Publish-Subscribe-Kommunikationsmuster. Der Block kann zwei verschiedene Datenwerte von einem Publisher empfangen und weiterverarbeiten.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsereignis mit den zugehörigen Daten QI und ID
- **RSP**: Response-Ereignis mit zugehörigem Datenwert QI

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung mit den Daten QO und STATUS
- **IND**: Indikationsereignis bei neuen verfügbaren Daten mit QO, STATUS, RD_1 und RD_2

Daten-Eingänge

- **QI** (BOOL): Qualifier Input - Steuert die Aktivierung des Blocks
- **ID** (WSTRING): Identifikationsstring zur Zuordnung zum entsprechenden Publisher

Daten-Ausgänge

- **QO** (BOOL): Qualifier Output - Status der Blockoperation
- **STATUS** (WSTRING): Statusinformationen als Unicode-String
- **RD_1** (ANY): Empfangene Daten 1 (beliebiger Datentyp)
- **RD_2** (ANY): Empfangene Daten 2 (beliebiger Datentyp)

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der SUBSCRIBE_2-Block initialisiert sich über das INIT-Ereignis und abonniert dabei einen spezifischen Publisher mittels des ID-Identifikators. Nach erfolgreicher Initialisierung bestätigt der Block dies über INITO. Bei eingehenden Daten vom Publisher löst der Block das IND-Ereignis aus und stellt die empfangenen Daten über RD_1 und RD_2 zur Verfügung.

Technische Besonderheiten

- Unterstützt den Empfang von zwei verschiedenen Datenwerten gleichzeitig
- Verwendet WSTRING für Status- und Identifikationsinformationen
- Implementiert generische Datentypen (ANY) für maximale Flexibilität
- Basiert auf dem GEN_SUBSCRIBE-Generic-Template

Zustandsübersicht

Der Block durchläuft folgende Hauptzustände:

1. **Nicht initialisiert**: Block wartet auf INIT-Ereignis
2. **Initialisiert**: Block ist bereit zum Empfang von Daten
3. **Datenempfang**: Verarbeitung eingehender Daten und Auslösen von IND

Anwendungsszenarien

- Verteilte Steuerungssysteme mit Datenverteilung
- IoT-Anwendungen mit Publisher-Subscriber-Architektur
- Industrielle Automatisierung mit dezentraler Datenkommunikation
- Systeme mit flexiblen Datentypanforderungen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren SUBSCRIBE-Blöcken bietet SUBSCRIBE_2 die Möglichkeit, zwei verschiedene Datenwerte parallel zu empfangen. Die Verwendung von ANY-Datentypen ermöglicht eine höhere Flexibilität gegenüber typspezifischen Implementationen.

Fazit

Der SUBSCRIBE_2-Block stellt eine erweiterte Lösung für Publish-Subscribe-Kommunikation in 4diac-Systemen dar, die durch die Unterstützung zweier Datenkanäle und generischer Datentypen besonders flexibel einsetzbar ist.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

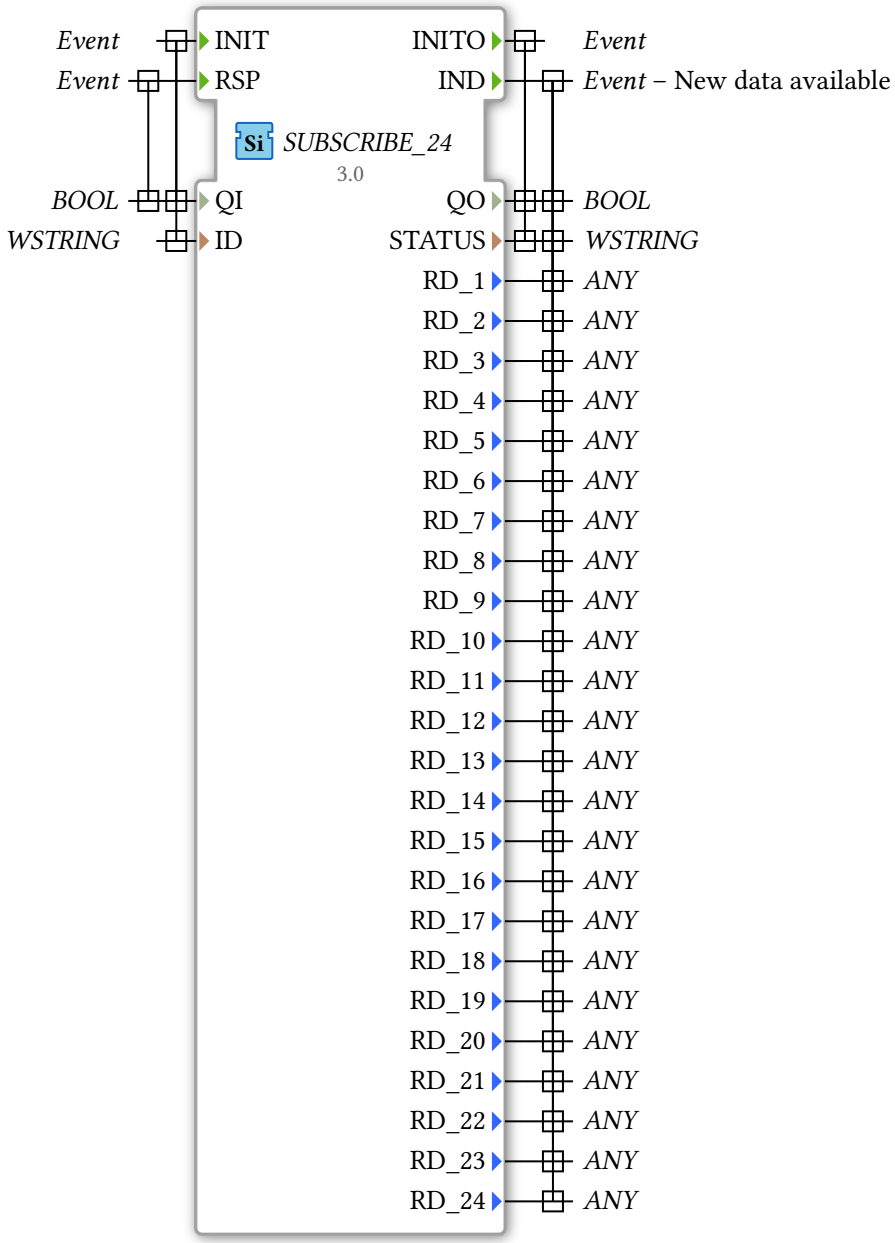
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.3.85 SUBSCRIBE_24

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `SUBSCRIBE_24` abonniert einen `PUBLISH_24`-Block und stellt die empfangenen Daten (bis zu 24 Werte) an seinen Ausgängen bereit.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (`EInit`): Initialisierung. Verknüpft mit `QI`, `ID`.
- **RSP**: Antwort (Response) auf eine Indikation. Verknüpft mit `QI`.

Ereignis-Ausgänge

- **INITO** (`EInit`): Initialisierungsbestätigung. Verknüpft mit `QO`, `STATUS`.
- **IND**: Neue Daten verfügbar. Verknüpft mit `QO`, `STATUS`, `RD_1` bis `RD_24`.

Daten-Eingänge

- **QI** (`BOOL`): Ereignis-Eingangsqualifizierer
- **ID** (`WSTRING`): Identifikationsstring des Kommunikationskanals (muss dem des `PUBLISH_24` entsprechen)

Daten-Ausgänge

- **QO** (`BOOL`): Ereignis-Ausgangsqualifizierer

- **STATUS** (`WSTRING`): Statusinformation
- **RD_1** bis **RD_24** (`ANY`): empfangene Datenwerte

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Nach der Initialisierung mit `INIT` meldet der Block bei jedem Empfang neuer Daten `IND` und stellt die Werte `RD_1` bis `RD_24` bereit. Der Dienst ist als Service-Interface (`APPLICATION` ↔ `RESOURCE`) ausgeführt; `RSP` beantwortet die Indikation.

Technische Besonderheiten

- Generischer Baustein (Generic Class `GEN_SUBSCRIBE`): Die Anzahl der Datenports ist hier 24, ihre Typen (`ANY`) werden bei der Instanziierung festgelegt
- Unicode-Strings (`WSTRING`) für `ID` und `STATUS`
- Gegenstück: `PUBLISH_24` mit `SD_1` bis `SD_24`

Anwendungsszenarien

- Empfang vieler zusammengehöriger Werte, die ein `PUBLISH_24` an anderer Stelle sendet

⚖️ Vergleich mit ähnlichen Bausteinen

`SUBSCRIBE_10` empfängt bis zu 10 Datenwerte, `SUBSCRIBE_24` bis zu 24.

Fazit

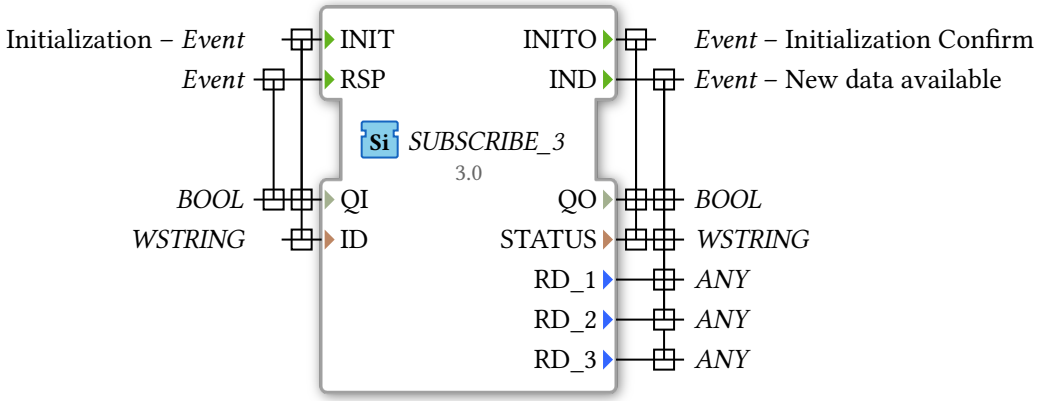
`SUBSCRIBE_24` ist das Empfangsgegenstück zu `PUBLISH_24` für bis zu 24 Datenwerte.

2.4.3.86 SUBSCRIBE_3

• – ◦ • – ◦ • – ◦ •

Einleitung

Der SUBSCRIBE_3 Funktionsblock dient zum Abonnieren von Daten eines PUBLISH_3 Blocks. Er ermöglicht den Empfang von drei verschiedenen Datenströmen über eine Netzwerkverbindung und stellt einen standardisierten Mechanismus für die Kommunikation zwischen verteilten Systemkomponenten bereit.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsereignis mit zugehörigen Daten QI und ID
- **RSP**: Response-Ereignis mit zugehörigem Daten QI

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung mit zugehörigen Daten QO und STATUS
- **IND**: Datenverfügbarkeitsereignis mit zugehörigen Daten QO, STATUS, RD_1, RD_3 und RD_2

Daten-Eingänge

- **QI** (BOOL): Qualifizierer für Initialisierung und Betrieb
- **ID** (WSTRING): Identifikationsstring für die Verbindung

Daten-Ausgänge

- **QO** (BOOL): Qualifizierer für Ausgangszustand
- **STATUS** (WSTRING): Statusinformationen der Verbindung
- **RD_1** (ANY): Empfangene Daten vom Typ 1
- **RD_2** (ANY): Empfangene Daten vom Typ 2
- **RD_3** (ANY): Empfangene Daten vom Typ 3

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der SUBSCRIBE_3 Block initialisiert sich über das INIT-Ereignis und stellt eine Verbindung zu einem entsprechenden PUBLISH_3 Block her. Nach erfolgreicher Initialisierung bestätigt er dies über INITO. Bei eingehenden Daten vom Publisher wird das IND-Ereignis ausgelöst und die empfangenen Daten über RD_1, RD_2 und RD_3 ausgegeben. Der STATUS-Parameter liefert dabei Informationen über den Verbindungszustand.

Technische Besonderheiten

- Unterstützt drei unabhängige Datenkanäle (RD_1, RD_2, RD_3) mit ANY-Datentyp
- Verwendet WSTRING für ID und STATUS für internationale Zeichenunterstützung
- Implementiert ein zuverlässiges Initialisierungsprotokoll mit QI/QO-Handshake
- Generische Implementierung durch GEN_SUBSCRIBE Basisklasse

Zustandsübersicht

1. **Nicht initialisiert**: Block wartet auf INIT-Ereignis
2. **Initialisierung**: Verarbeitung von INIT mit ID-Parameter
3. **Betriebsbereit**: Erfolgreiche Verbindung zum Publisher
4. **Datenempfang**: Verarbeitung eingehender Daten mit IND-Auslösung

Anwendungsszenarien

- Verteilte Steuerungssysteme in der Automatisierungstechnik
- Datenverteilung in IoT-Anwendungen
- Kommunikation zwischen verschiedenen Steuerungsebenen
- Maschine-zu-Maschine-Kommunikation in Industrie 4.0 Umgebungen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren SUBSCRIBE-Blöcken bietet SUBSCRIBE_3 die Möglichkeit, drei verschiedene Datenströme parallel zu empfangen. Die Verwendung von ANY-Datentypen ermöglicht größere Flexibilität bei den übertragenen Datenformaten.

Fazit

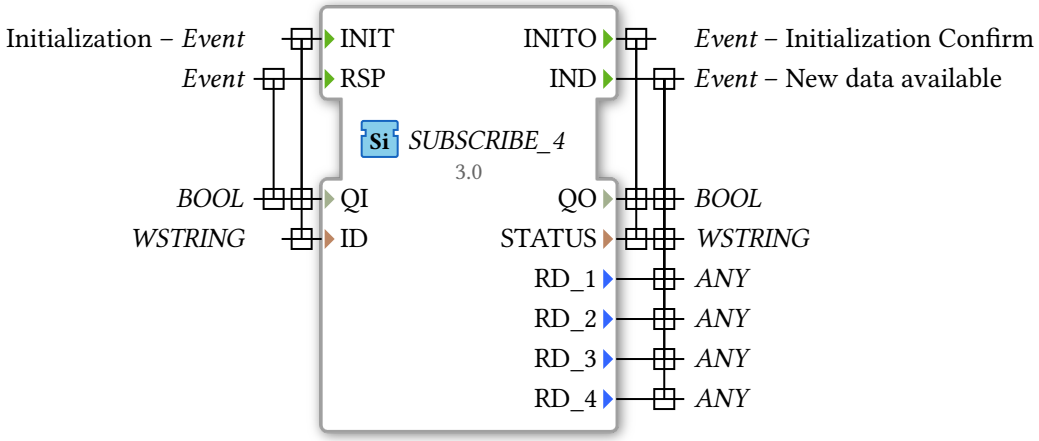
Der SUBSCRIBE_3 Funktionsblock stellt eine leistungsfähige Lösung für die Kommunikation in verteilten Automatisierungssystemen dar. Durch die Unterstützung mehrerer Datenkanäle und flexible Datentypen eignet er sich besonders für komplexe Anwendungen, bei mehrere Datenquellen parallel überwacht werden müssen.

2.4.3.87 SUBSCRIBE_4

• – ◦ • – ◦ • – ◦ •

Einleitung

Der SUBSCRIBE_4 Funktionsblock dient zum Abonnieren von Daten eines PUBLISH_4 Blocks. Er ermöglicht den Empfang von bis zu vier verschiedenen Datenpunkten über eine Kommunikationsverbindung und stellt diese bei Verfügbarkeit zur Verfügung.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** (Typ: EInit) - Initialisierungsereignis
  - Verknüpft mit: QI, ID
- **RSP** (Typ: Event) - Response-Ereignis
  - Verknüpft mit: QI

Ereignis-Ausgänge

- **INITO** (Typ: EInit) - Initialisierungsbestätigung
  - Verknüpft mit: QO, STATUS
- **IND** (Typ: Event) - Datenverfügbarkeitsmeldung
  - Verknüpft mit: QO, STATUS, RD_1, RD_3, RD_2, RD_4

Daten-Eingänge

- **QI** (BOOL) - Qualifizierter Input (Aktivierung/Deaktivierung)
- **ID** (WSTRING) - Identifikator für die Kommunikationsverbindung

Daten-Ausgänge

- **QO** (BOOL) - Qualifizierter Output (Statusrückmeldung)
- **STATUS** (WSTRING) - Statusinformationen
- **RD_1** (ANY) - Empfangene Daten 1
- **RD_2** (ANY) - Empfangene Daten 2
- **RD_3** (ANY) - Empfangene Daten 3
- **RD_4** (ANY) - Empfangene Daten 4

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der SUBSCRIBE_4 Block initialisiert bei Empfang des INIT-Ereignisses eine Abonnement-Verbindung zu einem PUBLISH_4 Block unter Verwendung der angegebenen ID. Nach erfolgreicher Initialisierung bestätigt er dies mit INITO. Bei Verfügbarkeit neuer Daten vom Publisher wird das IND-Ereignis ausgelöst und die empfangenen Daten über die RD_1 bis RD_4 Ausgänge bereitgestellt.

Technische Besonderheiten

- Unterstützt den Empfang von bis zu vier verschiedenen Datentypen (ANY-Typ)
- Verwendet WSTRING für Status- und Identifikationsinformationen
- Generische Implementierung durch GEN_SUBSCRIBE Basisklasse
- Robuste Fehlerbehandlung über STATUS-Ausgabe

Zustandsübersicht

1. **Inaktiv**: Block ist nicht initialisiert

2. **Initialisiert**: Abonnement aktiv, wartet auf Daten
3. **Datenempfang**: Verarbeitet eingehende Daten und gibt IND-Ereignis aus

Anwendungsszenarien

- Verteilte Automatisierungssysteme
- Datenverteilung in dezentralen Steuerungen
- Kommunikation zwischen verschiedenen Steuerungskomponenten
- Überwachungssysteme mit Datenabonnement

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren SUBSCRIBE-Blöcken bietet SUBSCRIBE_4 die Möglichkeit, bis zu vier verschiedene Datenpunkte gleichzeitig zu empfangen. Die erweiterte Funktionalität macht ihn besonders geeignet für komplexere Kommunikationsszenarien, bei denen mehrere Datenquellen konsolidiert werden müssen.

Fazit

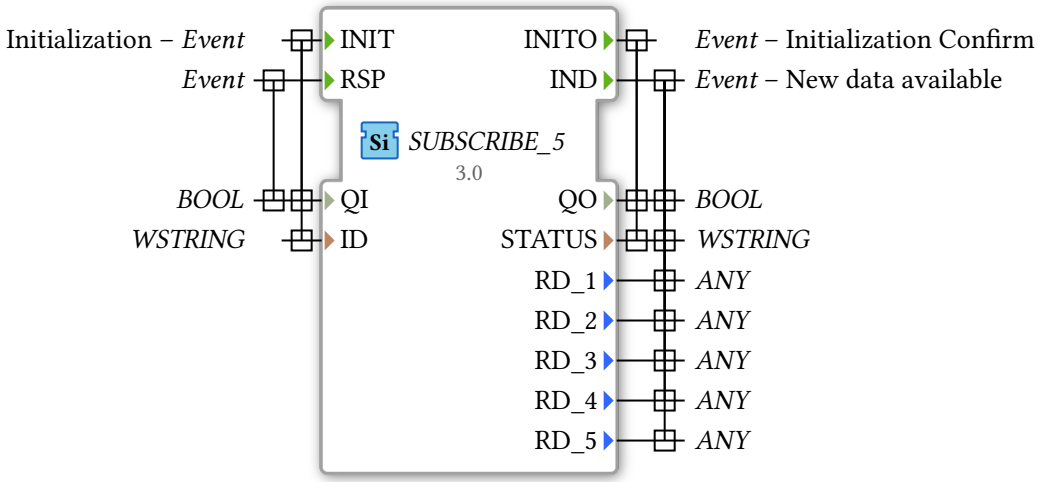
Der SUBSCRIBE_4 Funktionsblock stellt eine leistungsfähige Lösung für Abonnement-basierte Kommunikation in IEC 61499-Systemen dar. Seine Fähigkeit, mehrere Datenpunkte parallel zu verwalten, macht ihn zu einer flexiblen Wahl für anspruchsvolle Automatisierungsanwendungen mit verteilter Datenkommunikation.

2.4.3.88 SUBSCRIBE_5

• – ◦ • – ◦ • – ◦ •

Einleitung

Der SUBSCRIBE_5 Funktionsblock dient zum Abonnieren von Daten eines PUBLISH_5 Blocks. Er ermöglicht den Empfang von bis zu fünf verschiedenen Datenpunkten über eine Netzwerkverbindung und stellt eine standardisierte Schnittstelle für die Kommunikation zwischen verteilten Automatisierungskomponenten bereit.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsereignis mit den zugehörigen Daten QI und ID
- **RSP**: Response-Ereignis mit zugehörigem Datenpunkt QI

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung mit den Daten QO und STATUS
- **IND**: Indikationsereignis bei neuen verfügbaren Daten mit QO, STATUS und den fünf Datenpunkten RD_1 bis RD_5

Daten-Eingänge

- **QI** (BOOL): Qualifizierer für Initialisierung - aktiviert/deaktiviert den Block
- **ID** (WSTRING): Identifikationsstring zur Zuordnung zum entsprechenden PUBLISH_5 Block

Daten-Ausgänge

- **QO** (BOOL): Qualifizierer-Ausgang - zeigt den Betriebszustand an
- **STATUS** (WSTRING): Statusinformationen und Fehlermeldungen
- **RD_1** bis **RD_5** (ANY): Empfangene Datenpunkte 1 bis 5 mit beliebigem Datentyp

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der SUBSCRIBE_5 Block initialisiert sich über das INIT-Ereignis und stellt anschließend eine Verbindung zum entsprechenden PUBLISH_5 Block her. Bei erfolgreicher Initialisierung bestätigt er dies über INITO. Sobald neue Daten vom Publisher verfügbar sind, werden diese über das IND-Ereignis mit den entsprechenden Datenausgängen RD_1 bis RD_5 ausgegeben. Der STATUS-Ausgang liefert dabei Informationen über den Verbindungszustand und eventuelle Fehler.

Technische Besonderheiten

- Unterstützt bis zu fünf verschiedene Datenpunkte gleichzeitig
- Verwendet WSTRING für ID und STATUS für internationale Zeichensatzunterstützung
- ANY-Typ bei den Datenausgängen ermöglicht flexible Datentypen
- Generische Implementierung durch GEN_SUBSCRIBE Basisklasse

Zustandsübersicht

1. **Nicht initialisiert**: Block wartet auf INIT-Ereignis
2. **Initialisierungsphase**: Verarbeitung von INIT mit ID-Parameter
3. **Verbunden**: Erfolgreiche Verbindung zum Publisher, bereit für Datenempfang
4. **Datenempfang**: Verarbeitung eingehender Daten und Ausgabe über IND

Anwendungsszenarien

- Verteilte Automatisierungssysteme
- Datenverteilung in Produktionsanlagen
- Maschine-zu-Maschine-Kommunikation
- Überwachungssysteme mit mehreren Sensordaten
- SCADA-Systeme mit dezentralen Datenquellen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren SUBSCRIBE-Blöcken bietet SUBSCRIBE_5 die Möglichkeit, bis zu fünf verschiedene Datenpunkte parallel zu empfangen. Die Verwendung von ANY-Typen macht ihn flexibler als typspezifische Subscribe-Blöcke, erfordert jedoch eine korrekte Typzuordnung zum entsprechenden Publisher.

Fazit

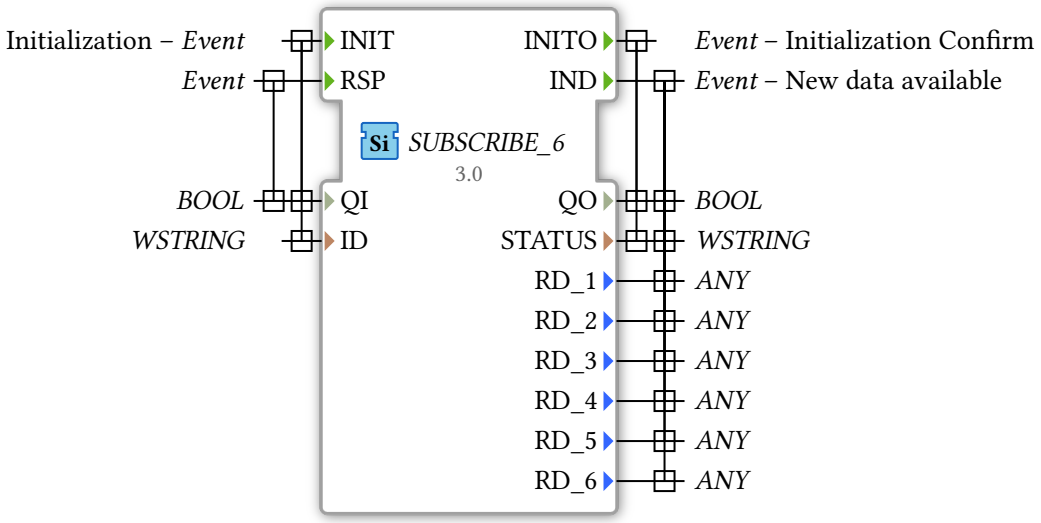
Der SUBSCRIBE_5 Funktionsblock stellt eine leistungsfähige und flexible Lösung für den Empfang multipler Datenströme in verteilten Automatisierungssystemen dar. Seine generische Implementierung und die Unterstützung verschiedener Datentypen machen ihn besonders geeignet für komplexe Anwendungen mit variablen Datenstrukturen.

2.4.3.89 SUBSCRIBE_6

• – ◦ • – ◦ • – ◦ •

Einleitung

Der SUBSCRIBE_6 Funktionsblock dient zum Abonnieren von Daten eines PUBLISH_6 Blocks. Er ermöglicht den Empfang von bis zu sechs verschiedenen Datenwerten über eine Netzwerkverbindung und stellt diese für die weitere Verarbeitung in einem IEC 61499-System zur Verfügung.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsereignis mit zugehörigen Daten QI und ID
- **RSP**: Response-Ereignis mit zugehörigem Daten QI

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung mit zugehörigen Daten QO und STATUS
- **IND**: Indikationsereignis bei neu verfügbaren Daten mit zugehörigen Daten QO, STATUS und RD_1 bis RD_6

Daten-Eingänge

- **QI** (BOOL): Qualifier Input - Steuert die Aktivierung des Blocks
- **ID** (WSTRING): Identifikationsstring zur Zuordnung zum entsprechenden PUBLISH_6 Block

Daten-Ausgänge

- **QO** (BOOL): Qualifier Output - Status der Blockoperation
- **STATUS** (WSTRING): Statusinformationen als Unicode-String
- **RD_1** bis **RD_6** (ANY): Empfangene Datenwerte (bis zu 6 verschiedene Datentypen)

Funktionsweise

Der SUBSCRIBE_6 Block initialisiert sich über das INIT-Ereignis und stellt eine Verbindung zu einem PUBLISH_6 Block mit der angegebenen ID her. Bei erfolgreicher Initialisierung bestätigt er dies mit INITO. Sobald neue Daten vom PUBLISH_6 Block verfügbar sind, löst der SUBSCRIBE_6 Block das IND-Ereignis aus und stellt die empfangenen Daten über die RD_1 bis RD_6 Ausgänge zur Verfügung.

Technische Besonderheiten

- Unterstützt bis zu sechs verschiedene Datenwerte gleichzeitig
- Verwendet WSTRING für Statusmeldungen und Identifikation
- Flexible Datentypen durch ANY-Typ bei den empfangenen Daten
- Netzwerkbasierte Kommunikation zwischen verteilten Systemen

Zustandsübersicht

Der Block durchläuft folgende Hauptzustände:

1. **Nicht initialisiert**: Block ist inaktiv
2. **Initialisierung**: Verarbeitung des INIT-Ereignisses
3. **Verbunden**: Erfolgreiche Verbindung zum Publisher
4. **Datenempfang**: Empfängt und verarbeitet eingehende Daten

Anwendungsszenarien

- Verteilte Steuerungssysteme mit Datenverteilung
- Überwachungssysteme mit zentraler Datenerfassung
- Industrie 4.0 Anwendungen mit Maschinendaten
- SCADA-Systeme mit dezentralen Datenquellen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren Subscribe-Blocks bietet SUBSCRIBE_6 die Möglichkeit, bis zu sechs verschiedene Datenwerte parallel zu empfangen, was eine höhere Datendurchsatzkapazität ermöglicht. Die Verwendung von WSTRING für Statusmeldungen ermöglicht detailliertere Fehlerinformationen.

Fazit

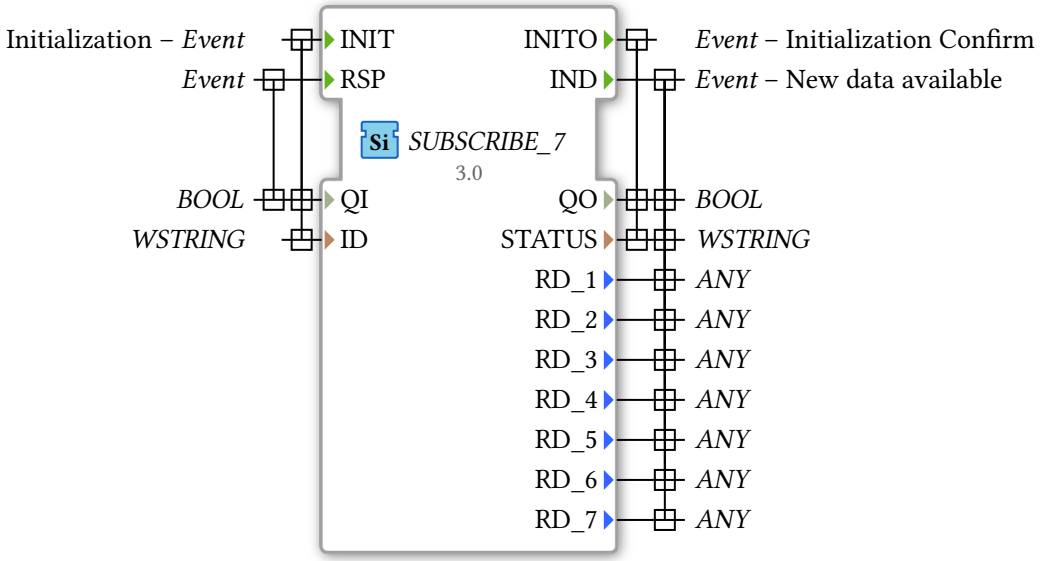
Der SUBSCRIBE_6 Funktionsblock ist eine leistungsstarke Lösung für verteilte Systeme, die den gleichzeitigen Empfang mehrerer Datenströme erfordern. Seine flexible Architektur und die Unterstützung verschiedener Datentypen machen ihn ideal für komplexe industrielle Automatisierungsanwendungen.

2.4.3.90 SUBSCRIBE_7

• – ◦ • – ◦ • – ◦ •

Einleitung

Der SUBSCRIBE_7 Funktionsblock dient als Abonnent für Daten eines PUBLISH_7 Blocks. Er ermöglicht den Empfang von bis zu sieben verschiedenen Datenpunkten über eine Kommunikationsverbindung. Der Block ist Teil der IEC 61499 Netzwerkkommunikationsfunktionalität und wird für verteilte Automatisierungssysteme eingesetzt.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsereignis mit zugehörigen Daten QI und ID
- **RSP**: Response-Ereignis mit zugehörigem Daten QI

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung mit zugehörigen Daten QO und STATUS
- **IND**: Indikationsereignis bei neuen verfügbaren Daten mit zugehörigen Daten QO, STATUS und RD_1 bis RD_7

Daten-Eingänge

- **QI** (BOOL): Qualifizierer für Initialisierung
- **ID** (WSTRING): Identifikator für die Verbindung

Daten-Ausgänge

- **QO** (BOOL): Qualifizierer für Ausgangszustand
- **STATUS** (WSTRING): Statusinformation der Verbindung
- **RD_1** bis **RD_7** (ANY): Empfangene Datenpunkte 1 bis 7

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der SUBSCRIBE_7 Block initialisiert sich über das INIT-Ereignis und stellt eine Verbindung zu einem entsprechenden PUBLISH_7 Block her. Nach erfolgreicher Initialisierung bestätigt er dies über INITO. Bei empfangenen Daten vom Publisher wird das IND-Ereignis ausgelöst und die empfangenen Daten über RD_1 bis RD_7 ausgegeben. Der RSP-Eingang ermöglicht die Behandlung von Antwortnachrichten.

Technische Besonderheiten

- Unterstützt bis zu sieben verschiedene Datenpunkte gleichzeitig
- Verwendet WSTRING für Status- und Identifikationsinformationen
- ANY-Datentyp für die empfangenen Daten ermöglicht flexible Datentypen
- Generische Implementierung durch GEN_SUBSCRIBE Basisklasse

Zustandsübersicht

1. **Nicht initialisiert**: Block wartet auf INIT-Ereignis
2. **Initialisierung**: Verarbeitung der INIT-Anfrage
3. **Bereit**: Erfolgreich verbunden, wartet auf Daten
4. **Datenempfang**: Verarbeitung eingehender Daten mit IND-Auslösung

Anwendungsszenarien

- Verteiltes Automatisierungssystem mit Datenverteilung
- Maschinenvernetzung in Industrie 4.0 Umgebungen
- Datenaustausch zwischen verschiedenen Steuerungskomponenten
- Überwachungssysteme mit zentraler Datensammlung

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren SUBSCRIBE-Blöcken bietet SUBSCRIBE_7 die Möglichkeit, bis zu sieben Datenpunkte parallel zu empfangen. Gegenüber individuellen Kommunikationslösungen bietet er standardisierte IEC 61499 Schnittstellen und vereinfachte Konfiguration.

Fazit

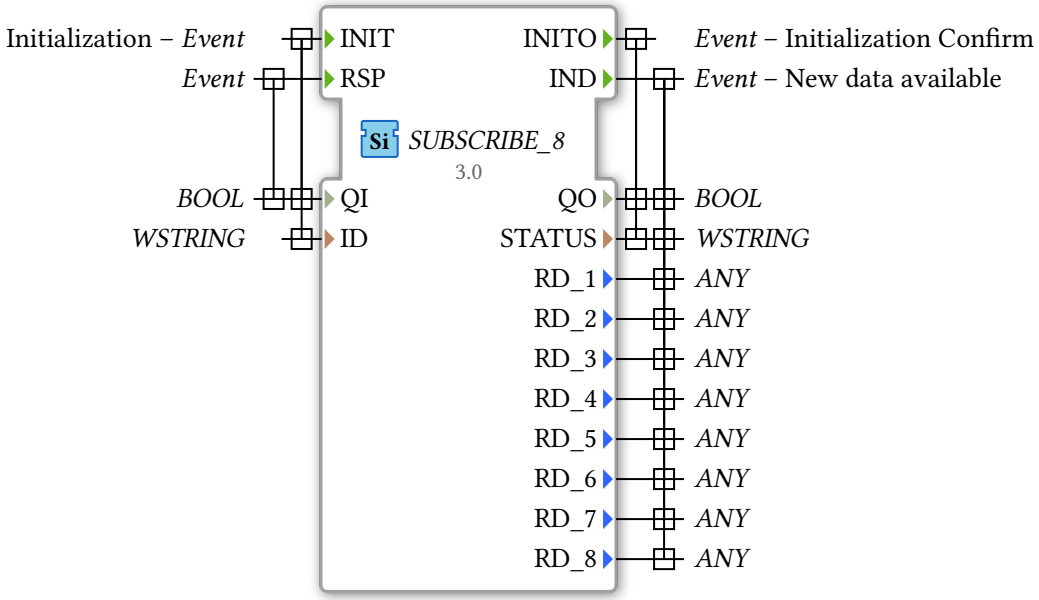
Der SUBSCRIBE_7 Block ist ein leistungsfähiger Kommunikationsbaustein für verteilte Automatisierungssysteme, der durch seine Flexibilität bei der Datenübertragung und standardisierte Schnittstellen eine zuverlässige Grundlage für industrielle Kommunikationsanwendungen bietet.

2.4.3.91 SUBSCRIBE_8

• – ◦ • – ◦ • – ◦ •

Einleitung

Der SUBSCRIBE_8 Funktionsblock dient als Abonnent für einen PUBLISH_8 Block und ermöglicht den Empfang von bis zu 8 verschiedenen Datenwerten über eine Kommunikationsverbindung. Der Block implementiert ein Publish-Subscribe-Kommunikationsmuster und ist Teil des iec61499::net Pakets.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT** - Initialisierungsereignis
  - Verknüpft mit: QI, ID
- **RSP** - Response-Ereignis
  - Verknüpft mit: QI

Ereignis-Ausgänge

- **INITO** - Initialisierungsbestätigung
  - Verknüpft mit: QO, STATUS
- **IND** - Indikationsereignis bei neuen Daten
  - Verknüpft mit: QO, STATUS, RD_1 bis RD_8

Daten-Eingänge

- **QI** (BOOL) - Qualifizierter Eingang für Initialisierung
- **ID** (WSTRING) - Identifikationsstring für die Verbindung

Daten-Ausgänge

- **QO** (BOOL) - Qualifizierter Ausgang
- **STATUS** (WSTRING) - Statusinformationen
- **RD_1** bis **RD_8** (ANY) - Empfangene Datenwerte 1 bis 8

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der SUBSCRIBE_8 Block initialisiert sich über das INIT-Ereignis und stellt dabei eine Verbindung zu einem entsprechenden PUBLISH_8 Block her. Nach erfolgreicher Initialisierung bestätigt er dies mit dem INITO-Ereignis. Bei empfangenen Daten vom Publisher wird das IND-Ereignis ausgelöst und die empfangenen Daten über die RD_1 bis RD_8 Ausgänge bereitgestellt.

Technische Besonderheiten

- Unterstützt den Datentyp ANY für alle Datenausgänge, was maximale Flexibilität bei den übertragenen Datentypen ermöglicht
- Verwendet WSTRING für Statusmeldungen und Identifikation
- Implementiert ein zuverlässiges Initialisierungsprotokoll mit Qualifikationsbits
- Kann bis zu 8 verschiedene Datenwerte parallel empfangen

Zustandsübersicht

1. **Nicht initialisiert** - Block wartet auf INIT-Ereignis
2. **Initialisierung** - Verarbeitung des INIT-Ereignisses
3. **Bereit** - Erfolgreich initialisiert, wartet auf Daten
4. **Datenempfang** - Verarbeitung eingehender Daten mit IND-Auslösung

Anwendungsszenarien

- Verteilte Steuerungssysteme mit Datenverteilung
- Überwachungssysteme mit zentraler Datenerfassung
- Kommunikation zwischen verschiedenen Steuerungskomponenten
- Systeme mit Publish-Subscribe-Architektur

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren Subscribe-Blöcken bietet SUBSCRIBE_8 die Möglichkeit, bis zu 8 verschiedene Datenwerte parallel zu empfangen. Die Verwendung des ANY-Datentyps ermöglicht eine höhere Flexibilität als typspezifische Subscribe-Blöcke.

Fazit

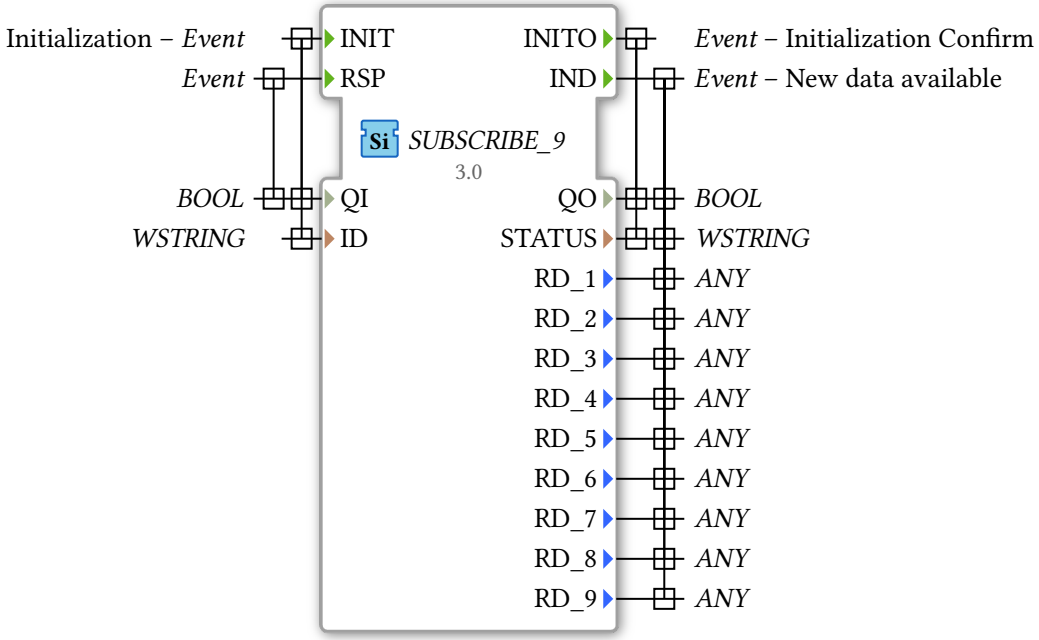
Der SUBSCRIBE_8 Funktionsblock stellt eine leistungsstarke und flexible Lösung für Publish-Subscribe-Kommunikation in verteilten Automatisierungssystemen dar. Seine Fähigkeit, multiple Datenwerte unterschiedlicher Typen zu empfangen, macht ihn besonders geeignet für komplexe Kommunikationsszenarien in industriellen Steuerungssystemen.

2.4.3.92 SUBSCRIBE_9

• – ◦ • – ◦ • – ◦ •

Einleitung

Der SUBSCRIBE_9 Funktionsblock dient zum Abonnieren von Daten eines PUBLISH_9 Blocks. Er ermöglicht den Empfang von bis zu 9 verschiedenen Datenpunkten über eine Netzwerkverbindung und stellt diese für die weitere Verarbeitung im Steuerungssystem bereit.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierungsereignis mit den zugehörigen Daten QI und ID
- **RSP**: Response-Ereignis mit zugehörigem Daten QI

Ereignis-Ausgänge

- **INITO**: Initialisierungsbestätigung mit QO und STATUS
- **IND**: Indikationsereignis bei neuen verfügbaren Daten mit QO, STATUS und allen 9 RD_x Datenausgängen

Daten-Eingänge

- **QI** (BOOL): Qualifizierer für Initialisierung (TRUE = aktivieren, FALSE = deaktivieren)
- **ID** (WSTRING): Identifikationsstring für die Verbindung zum Publisher

Daten-Ausgänge

- **QO** (BOOL): Qualifizierer-Ausgang für Betriebszustand
- **STATUS** (WSTRING): Statusinformationen als Unicode-String
- **RD_1** bis **RD_9** (ANY): Empfangene Datenpunkte 1-9 mit beliebigem Datentyp

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der SUBSCRIBE_9 Block initialisiert bei Empfang des INIT-Ereignisses eine Verbindung zu einem PUBLISH_9 Block basierend auf der angegebenen ID. Nach erfolgreicher Initialisierung bestätigt er dies mit dem INITO-Ereignis. Bei empfangenen Daten vom Publisher wird das IND-Ereignis ausgelöst und die Daten werden über die RD_1 bis RD_9 Ausgänge verfügbar gemacht.

Technische Besonderheiten

- Unterstützt bis zu 9 verschiedene Datenpunkte parallel
- Verwendet WSTRING für Status- und Identifikationsinformationen
- ANY-Datentypen für die empfangenen Daten ermöglichen flexible Datentypen
- Generische Implementierung als GEN_SUBSCRIBE

Zustandsübersicht

1. **Nicht initialisiert**: Block wartet auf INIT-Ereignis
2. **Initialisiert**: Verbindung zum Publisher hergestellt, bereit zum Empfang
3. **Datenempfang**: Verarbeitet eingehende Daten und triggert IND-Ereignis

Anwendungsszenarien

- Verteilte Steuerungssysteme mit Datenverteilung
- Maschinenvernetzung in Industrie 4.0 Umgebungen
- Überwachungssysteme mit zentraler Datensammlung
- Anlagen mit mehreren dezentralen Sensorknoten

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren SUBSCRIBE-Blöcken bietet SUBSCRIBE_9 die Möglichkeit, bis zu 9 verschiedene Datenpunkte gleichzeitig zu empfangen, was die Effizienz in komplexeren Netzwerkstrukturen erhöht.

Fazit

Der SUBSCRIBE_9 Funktionsblock ist eine leistungsstarke Lösung für den Empfang multipler Datenströme in verteilten Automatisierungssystemen und eignet sich besonders für Anwendungen mit hohem Datenaufkommen.

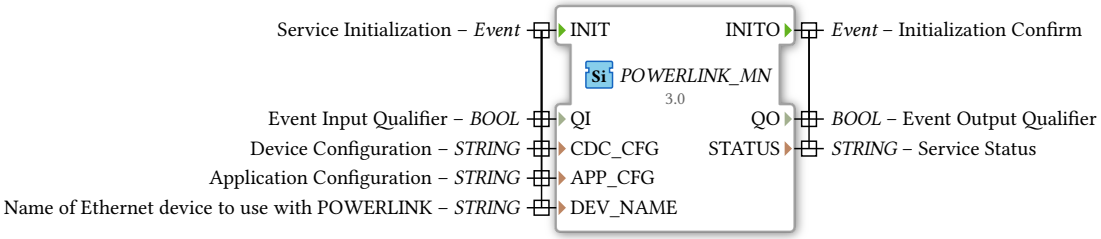
2.4.4 powerlink

2.4.4.1 POWERLINK_MN

• – ◦ • – ◦ • – ◦ •

Einleitung

Der POWERLINK_MN Funktionsblock dient zur Konfiguration des openPOWERLINK Feldbusses. Er fungiert als Managing Node (MN) und initialisiert das Netzwerk.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierung des Dienstes. Startet den Konfigurationsprozess.

Ereignis-Ausgänge

- **INITO**: Bestätigung der Initialisierung. Zeigt an, ob der Start erfolgreich war.

Daten-Eingänge

- **QI** (BOOL): Qualifizierer für den INIT-Eingang (TRUE = Start, FALSE = Stopp).
- **CDC_CFG** (STRING): Gerätekonfiguration (Device Configuration).
- **APP_CFG** (STRING): Anwendungskonfiguration (Application Configuration).
- **DEV_NAME** (STRING): Name des Ethernet-Geräts, das für POWERLINK verwendet werden soll (z.B. „eth0“).

Daten-Ausgänge

- **QO** (BOOL): Qualifizierer für den INITO-Ausgang.
- **STATUS** (STRING): Statusinformation des Dienstes.

Funktionsweise

Der Block initialisiert den openPOWERLINK Stack auf dem angegebenen Ethernet-Interface (`DEV_NAME`). Die Konfigurationsdateien für Gerät (`CDC_CFG`) und Anwendung (`APP_CFG`) werden geladen.

Anwendungsszenarien

- Zentrale Steuerung eines POWERLINK-Netzwerks.
- Master-Komponente in einem Echtzeit-Ethernet-System.

Metadaten

Attribut	Wert
Copyright	(c) 2011 AIT
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	powerlink

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

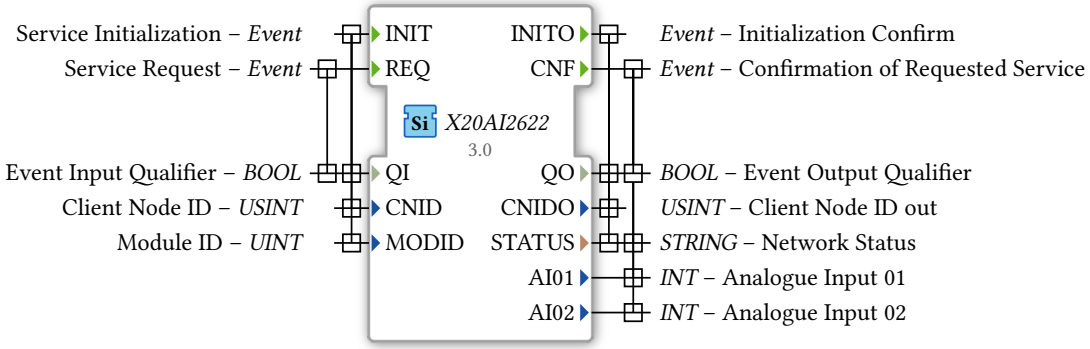
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.4.2 X20AI2622

• – ◦ • – ◦ • – ◦ •

Einleitung

Der X20AI2622 Funktionsblock stellt eine Schnittstelle für analoge Eingänge im openPOWERLINK Feldbus dar. Er entspricht dem B&R X20AI2622 Modul.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierung des Dienstes.
- **REQ**: Anforderung zur Aktualisierung der Eingangsdaten.

Ereignis-Ausgänge

- **INITO**: Bestätigung der Initialisierung.
- **CNF**: Bestätigung der Anfrage mit neuen Daten.

Daten-Eingänge

- **QI** (BOOL): Qualifizierer für INIT und REQ.
- **CNID** (USINT): Client Node ID (Adresse des Knotens im POWERLINK Netzwerk).
- **MODID** (UINT): Modul-ID (Position des Moduls im Knoten).

Daten-Ausgänge

- **QO** (BOOL): Qualifizierer für Ausgänge.
- **CNIDO** (USINT): Bestätigte Client Node ID.
- **STATUS** (STRING): Netzwerk-Status.
- **AI01** (INT): Analoger Eingang 01.
- **AI02** (INT): Analoger Eingang 02.

Funktionsweise

Nach der Initialisierung mit `CNID` und `MODID` liefert der Baustein bei jedem `REQ` die aktuellen Analogwerte des physischen Moduls an den Ausgängen `AIxx`.

Metadaten

Attribut	Wert
Copyright	(c) 2014 AIT
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	powerlink

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

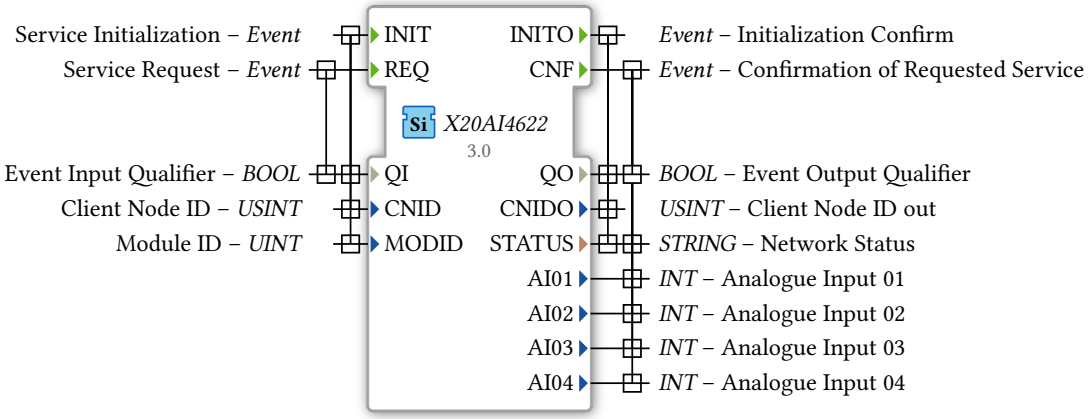
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.4.3 X20AI4622

• – ◦ • – ◦ • – ◦ •

Einleitung

Der X20AI4622 Funktionsblock stellt eine Schnittstelle für analoge Eingänge im openPOWERLINK Feldbus dar. Er entspricht dem B&R X20AI4622 Modul mit 4 analogen Eingängen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierung des Dienstes.
- **REQ**: Anforderung zur Aktualisierung der Eingangsdaten.

Ereignis-Ausgänge

- **INITO**: Bestätigung der Initialisierung.
- **CNF**: Bestätigung der Anfrage mit neuen Daten.

Daten-Eingänge

- **QI** (BOOL): Qualifizierer für INIT und REQ.
- **CNID** (USINT): Client Node ID.
- **MODID** (UINT): Modul-ID.

Daten-Ausgänge

- **QO** (BOOL): Qualifizierer für Ausgänge.
- **CNIDO** (USINT): Bestätigte Client Node ID.
- **STATUS** (STRING): Netzwerk-Status.
- **AI01** (INT): Analoger Eingang 01.
- **AI02** (INT): Analoger Eingang 02.
- **AI03** (INT): Analoger Eingang 03.
- **AI04** (INT): Analoger Eingang 04.

Funktionsweise

Nach der Initialisierung liefert der Baustein die aktuellen Analogwerte der vier Kanäle des Moduls.

Metadaten

Attribut	Wert
Copyright	(c) 2011 AIT
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	powerlink

Passende Themen-Unterseiten auf ms-muc-docs.de

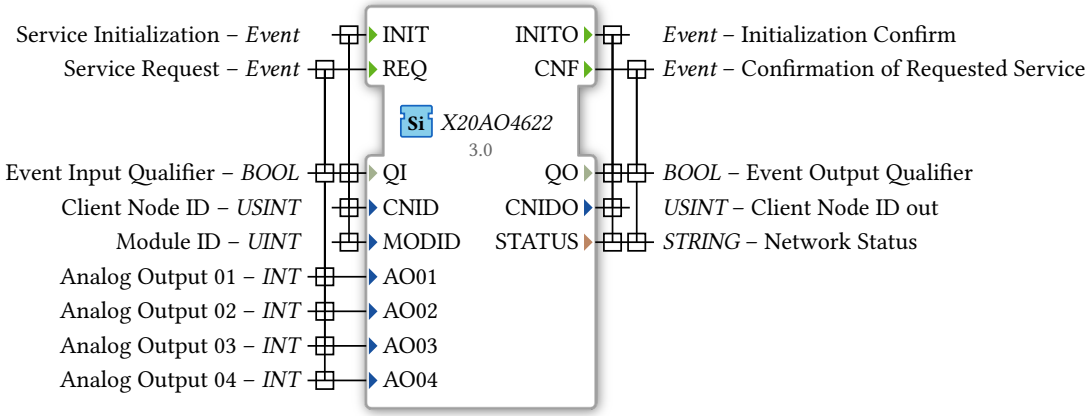
- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.4.4 X20AO4622

• – ◦ • – ◦ • – ◦ •

Einleitung

Der X20AO4622 Funktionsblock dient zur Ansteuerung von analogen Ausgängen über den openPOWERLINK Feldbus. Er entspricht dem B&R X20AO4622 Modul (4 Ausgänge).



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierung.
- **REQ**: Schreibt die Werte der Eingänge `AOxx` auf das Modul.

Ereignis-Ausgänge

- **INITO**: Initialisierung bestätigt.
- **CNF**: Schreibvorgang bestätigt.

Daten-Eingänge

- **QI** (BOOL): Qualifizierer.
- **CNID** (USINT): Client Node ID.
- **MODID** (UINT): Modul-ID.
- **AO01** (INT): Analogwert für Ausgang 01.
- **AO02** (INT): Analogwert für Ausgang 02.
- **AO03** (INT): Analogwert für Ausgang 03.
- **AO04** (INT): Analogwert für Ausgang 04.

Daten-Ausgänge

- **QO** (BOOL): Qualifizierer.
- **CNIDO** (USINT): Bestätigte Node ID.
- **STATUS** (STRING): Status.

Funktionsweise

Die an den Eingängen `AO01-AO04` anliegenden Integer-Werte werden bei Auslösung des `REQ`-Ereignisses an das Ausgabemodul gesendet.

Metadaten

Attribut	Wert
Copyright	(c) 2011 AIT
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	powerlink

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

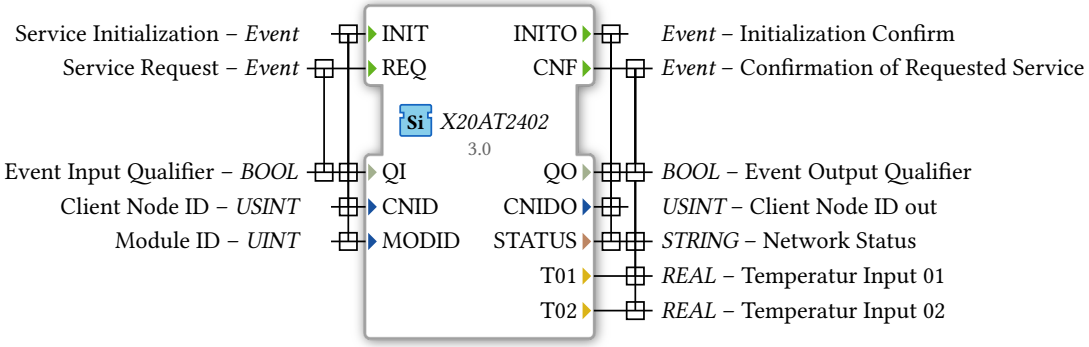
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.4.5 X20AT2402

• – ◦ • – ◦ • – ◦ •

Einleitung

Der X20AT2402 Funktionsblock dient zum Einlesen von Temperaturwerten über den openPOWERLINK Feldbus. Er entspricht dem B&R X20AT2402 Temperatur-Eingangsmodul (2 Kanäle).



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierung.
- **REQ**: Datenaktualisierung anfordern.

Ereignis-Ausgänge

- **INITO**: Initialisierung bestätigt.
- **CNF**: Neue Temperaturdaten verfügbar.

Daten-Eingänge

- **QI** (BOOL): Qualifizierer.
- **CNID** (USINT): Client Node ID.
- **MODID** (UINT): Modul-ID.

Daten-Ausgänge

- **QO** (BOOL): Qualifizierer.
- **CNIDO** (USINT): Bestätigte Node ID.
- **STATUS** (STRING): Status.
- **T01** (REAL): Temperatureingang 01 (typischerweise in °C, abhängig von Modulkonfiguration).
- **T02** (REAL): Temperatureingang 02.

Funktionsweise

Liest Temperaturwerte vom physischen Modul und stellt sie als REAL-Werte zur Verfügung.

Metadaten

Attribut	Wert
Copyright	(c) 2012 AIT
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	powerlink

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

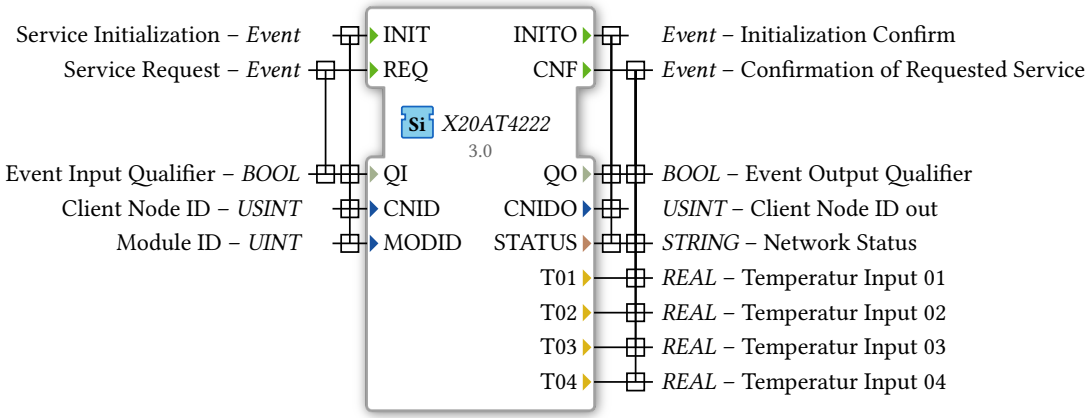
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.4.6 X20AT4222

• – ◦ • – ◦ • – ◦ •

Einleitung

Der X20AT4222 Funktionsblock dient zum Einlesen von Temperaturwerten über den openPOWERLINK Feldbus. Er entspricht dem B&R X20AT4222 Temperatur-Eingangsmodul (4 Kanäle).



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierung.
- **REQ**: Datenaktualisierung anfordern.

Ereignis-Ausgänge

- **INITO**: Initialisierung bestätigt.
- **CNF**: Neue Temperaturdaten verfügbar.

Daten-Eingänge

- **QI** (BOOL): Qualifizierer.
- **CNID** (USINT): Client Node ID.
- **MODID** (UINT): Modul-ID.

Daten-Ausgänge

- **QO** (BOOL): Qualifizierer.
- **CNIDO** (USINT): Bestätigte Node ID.
- **STATUS** (STRING): Status.
- **T01** (REAL): Temperatureingang 01.
- **T02** (REAL): Temperatureingang 02.
- **T03** (REAL): Temperatureingang 03.
- **T04** (REAL): Temperatureingang 04.

Funktionsweise

Liest Temperaturwerte vom physischen Modul und stellt sie als REAL-Werte zur Verfügung.

Metadaten

Attribut	Wert
Copyright	(c) 2012 AIT
Lizenz	EPL-2.0

Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	powerlink

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

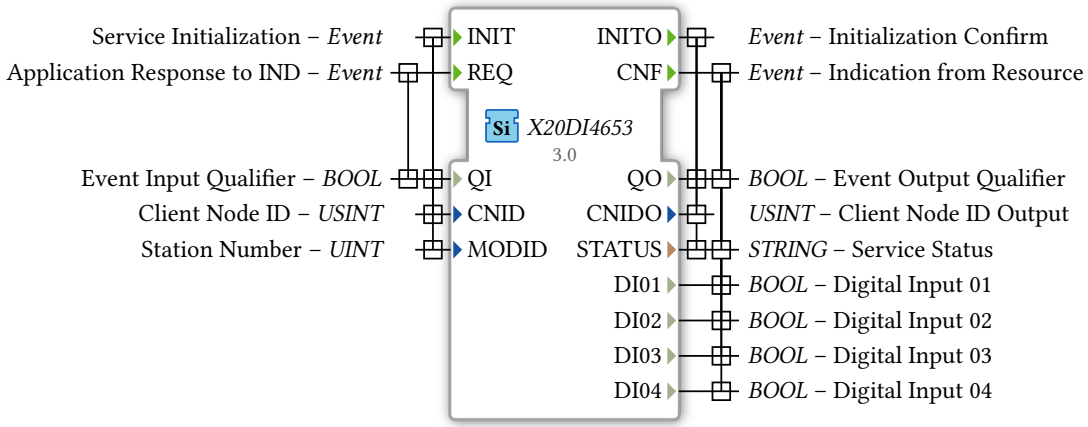
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.4.7 X20DI4653

• – ◦ • – ◦ • – ◦ •

Einleitung

Der X20DI4653 Funktionsblock dient zum Einlesen von digitalen Signalen über den openPOWERLINK Feldbus. Er entspricht dem B&R X20DI4653 Digital-Eingangsmodul (4 Kanäle).



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierung.
- **REQ**: Eingangsdaten aktualisieren.

Ereignis-Ausgänge

- **INITO**: Initialisierung bestätigt.
- **CNF**: Neue Daten verfügbar.

Daten-Eingänge

- **QI** (BOOL): Qualifizierer.
- **CNID** (USINT): Client Node ID.
- **MODID** (UINT): Modul-ID.

Daten-Ausgänge

- **QO** (BOOL): Qualifizierer.
- **CNIDO** (USINT): Bestätigte Node ID.
- **STATUS** (STRING): Status.
- **DI01** (BOOL): Digitaler Eingang 01.
- **DI02** (BOOL): Digitaler Eingang 02.
- **DI03** (BOOL): Digitaler Eingang 03.
- **DI04** (BOOL): Digitaler Eingang 04.

Funktionsweise

Liest den Zustand der digitalen Eingänge vom Modul und stellt sie als Boolesche Werte bereit.

Metadaten

Attribut	Wert
Copyright	(c) 2011 AIT
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	powerlink

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

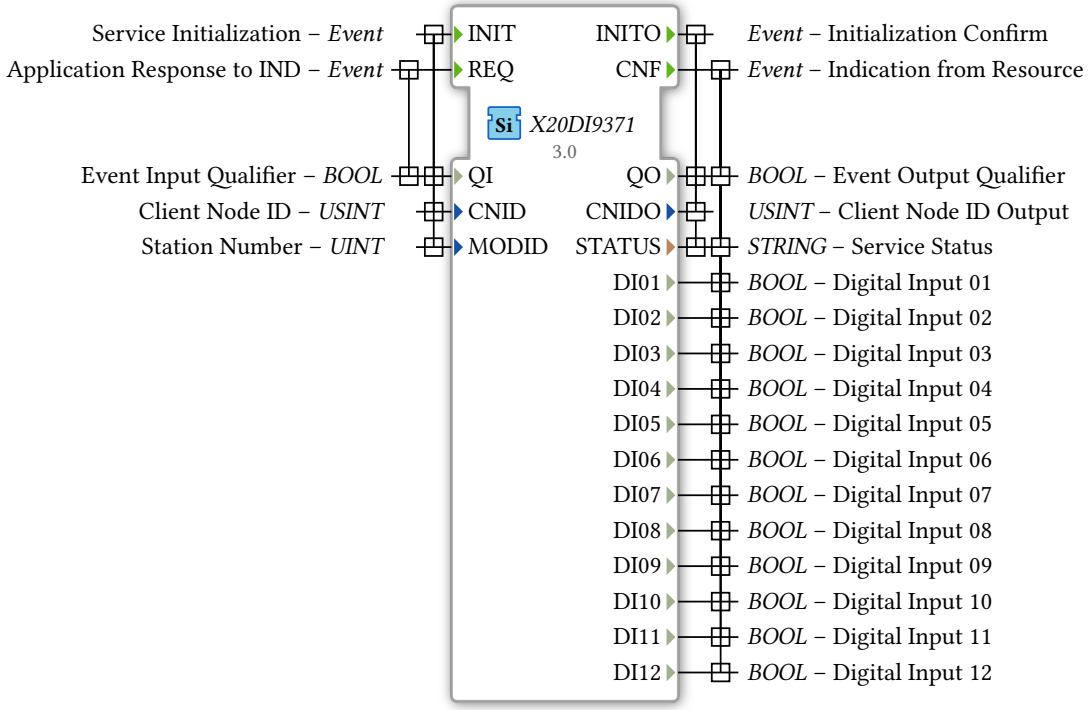
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.4.8 X20DI9371

• – ◦ • – ◦ • – ◦ •

Einleitung

Der X20DI9371 Funktionsblock dient zum Einlesen von digitalen Signalen über den openPOWERLINK Feldbus. Er entspricht dem B&R X20DI9371 Digital-Eingangsmodul (12 Kanäle).



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierung.
- **REQ**: Eingangsdaten aktualisieren.

Ereignis-Ausgänge

- **INITO**: Initialisierung bestätigt.
- **CNF**: Neue Daten verfügbar.

Daten-Eingänge

- **QI** (BOOL): Qualifizierer.
- **CNID** (USINT): Client Node ID.

- **MODID** (UINT): Modul-ID.

Daten-Ausgänge

- **QO** (BOOL): Qualifizierer.
- **CNIDO** (USINT): Bestätigte Node ID.
- **STATUS** (STRING): Status.
- **DI01** - **DI12** (BOOL): Digitale Eingänge 01 bis 12.

Funktionsweise

Liest den Zustand der 12 digitalen Eingänge vom Modul.

Metadaten

Attribut	Wert
Copyright	(c) 2011 AIT
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	powerlink

Passende Themen-Unterseiten auf ms-muc-docs.de

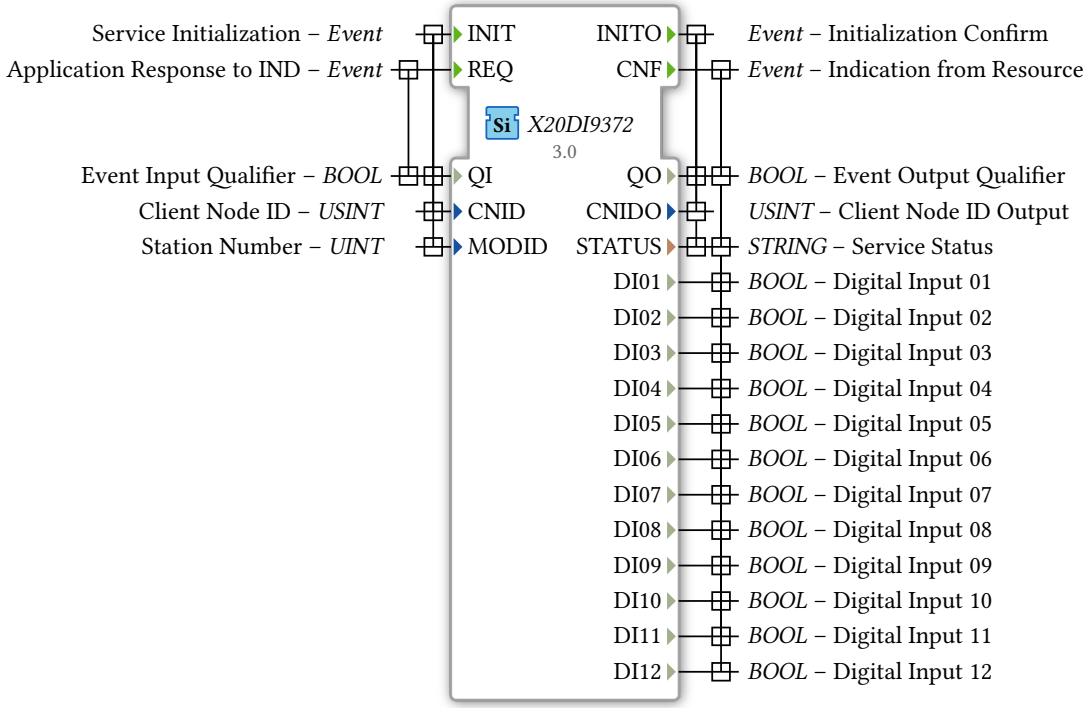
- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.4.9 X20DI9372

• – ◦ • – ◦ • – ◦ •

Einleitung

Der X20DI9372 Funktionsblock dient zum Einlesen von digitalen Signalen über den openPOWERLINK Feldbus. Er entspricht dem B&R X20DI9372 Digital-Eingangsmodul (12 Kanäle).



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierung.
- **REQ**: Eingangsdaten aktualisieren.

Ereignis-Ausgänge

- **INITO**: Initialisierung bestätigt.
- **CNF**: Neue Daten verfügbar.

Daten-Eingänge

- **QI** (BOOL): Qualifizierer.
- **CNID** (USINT): Client Node ID.
- **MODID** (UINT): Modul-ID.

Daten-Ausgänge

- **QO** (BOOL): Qualifizierer.
- **CNIDO** (USINT): Bestätigte Node ID.
- **STATUS** (STRING): Status.
- **DI01** - **DI12** (BOOL): Digitale Eingänge 01 bis 12.

Funktionsweise

Liest den Zustand der 12 digitalen Eingänge vom Modul.

Metadaten

Attribut	Wert
Copyright	(c) 2012 AIT
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	powerlink

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

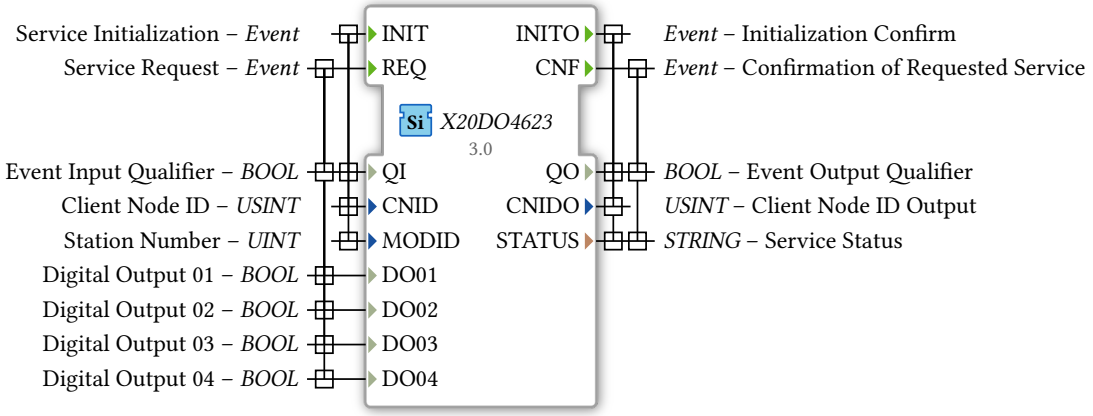
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.4.10 X20DO4623

• – ◦ • – ◦ • – ◦ •

Einleitung

Der X20DO4623 Funktionsblock dient zur Ansteuerung von digitalen Ausgängen über den openPOWERLINK Feldbus. Er entspricht dem B&R X20DO4623 Digital-Ausgangsmodul (4 Kanäle).



Schnittstellenstruktur

Ereignis-Eingänge
- **INIT**: Initialisierung.
- **REQ**: Ausgänge schreiben.

Ereignis-Ausgänge
- **INITO**: Initialisierung bestätigt.
- **CNF**: Schreibvorgang bestätigt.

Daten-Eingänge
- **QI** (BOOL): Qualifizierer.
- **CNID** (USINT): Client Node ID.
- **MODID** (UINT): Modul-ID.
- **DO01** (BOOL): Digitaler Ausgang 01.
- **DO02** (BOOL): Digitaler Ausgang 02.
- **DO03** (BOOL): Digitaler Ausgang 03.
- **DO04** (BOOL): Digitaler Ausgang 04.

Daten-Ausgänge
- **QO** (BOOL): Qualifizierer.
- **CNIDO** (USINT): Bestätigte Node ID.
- **STATUS** (STRING): Status.

Funktionsweise

Setzt die digitalen Ausgänge am Modul entsprechend den Eingangswerten `DO01-DO04`.

Metadaten

Attribut	Wert
Copyright	(c) 2011 AIT
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	powerlink

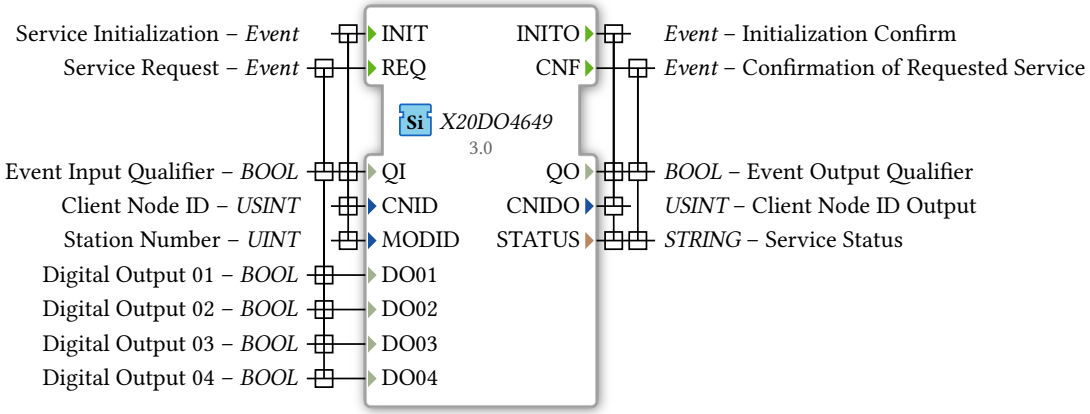
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.4.11 X20DO4649

• – ◦ • – ◦ • – ◦ •

Einleitung

Der X20DO4649 Funktionsblock dient zur Ansteuerung von digitalen Ausgängen über den openPOWERLINK Feldbus. Er entspricht dem B&R X20DO4649 Digital-Ausgangsmodul (4 Kanäle).



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierung.
- **REQ**: Ausgänge schreiben.

Ereignis-Ausgänge

- **INITO**: Initialisierung bestätigt.
- **CNF**: Schreibvorgang bestätigt.

Daten-Eingänge

- **QI** (BOOL): Qualifizierer.
- **CNID** (USINT): Client Node ID.
- **MODID** (UINT): Modul-ID.
- **DO01** (BOOL): Digitaler Ausgang 01.
- **DO02** (BOOL): Digitaler Ausgang 02.
- **DO03** (BOOL): Digitaler Ausgang 03.
- **DO04** (BOOL): Digitaler Ausgang 04.

Daten-Ausgänge

- **QO** (BOOL): Qualifizierer.
- **CNIDO** (USINT): Bestätigte Node ID.
- **STATUS** (STRING): Status.

Funktionsweise

Setzt die digitalen Ausgänge am Modul entsprechend den Eingangswerten `DO01-DO04`.

Metadaten

Attribut	Wert
Copyright	(c) 2011 AIT
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	powerlink

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

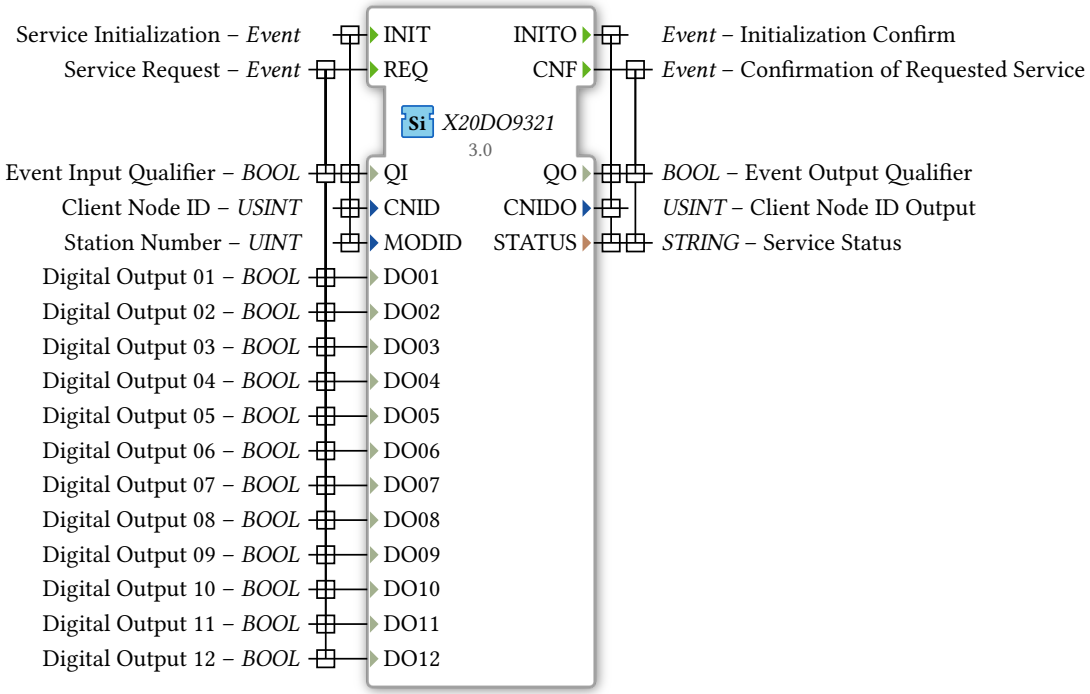
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.4.12 X20DO9321

• – ◦ • – ◦ • – ◦ •

Einleitung

Der X20DO9321 Funktionsblock dient zur Ansteuerung von digitalen Ausgängen über den openPOWERLINK Feldbus. Er entspricht dem B&R X20DO9321 Digital-Ausgangsmodul (12 Kanäle).



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierung.
- **REQ**: Ausgänge schreiben.

Ereignis-Ausgänge

- **INITO**: Initialisierung bestätigt.
- **CNF**: Schreibvorgang bestätigt.

Daten-Eingänge

- **QI** (BOOL): Qualifizierer.
- **CNID** (USINT): Client Node ID.
- **MODID** (UINT): Modul-ID.
- **DO01** - **DO12** (BOOL): Digitale Ausgänge 01 bis 12.

Daten-Ausgänge

- **QO** (BOOL): Qualifizierer.
- **CNIDO** (USINT): Bestätigte Node ID.
- **STATUS** (STRING): Status.

Funktionsweise

Setzt die 12 digitalen Ausgänge am Modul.

Metadaten

Attribut	Wert
Copyright	(c) 2012 AIT
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)

4diac-Paket	powerlink

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

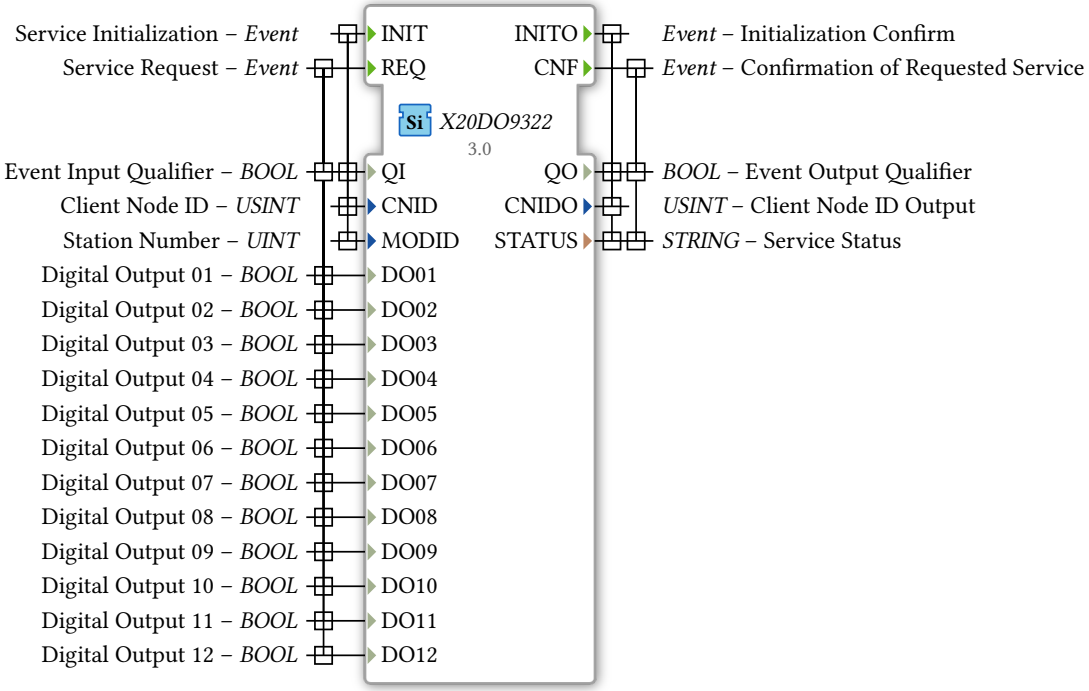
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.4.13 X20DO9322

• – ◦ • – ◦ • – ◦ •

Einleitung

Der X20DO9322 Funktionsblock dient zur Ansteuerung von digitalen Ausgängen über den openPOWERLINK Feldbus. Er entspricht dem B&R X20DO9322 Digital-Ausgangsmodul (12 Kanäle).



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierung.
- **REQ**: Ausgänge schreiben.

Ereignis-Ausgänge

- **INITO**: Initialisierung bestätigt.
- **CNF**: Schreibvorgang bestätigt.

Daten-Eingänge

- **QI** (BOOL): Qualifizierer.
- **CNID** (USINT): Client Node ID.
- **MODID** (UINT): Modul-ID.
- **DO01** - **DO12** (BOOL): Digitale Ausgänge 01 bis 12.

Daten-Ausgänge

- **QO** (BOOL): Qualifizierer.
- **CNIDO** (USINT): Bestätigte Node ID.
- **STATUS** (STRING): Status.

Funktionsweise

Setzt die 12 digitalen Ausgänge am Modul.

Metadaten

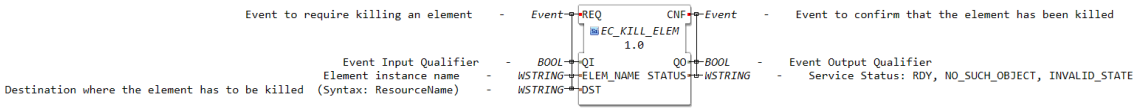
Attribut	Wert
Copyright	(c) 2011 AIT
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	powerlink

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.5 reconfiguration

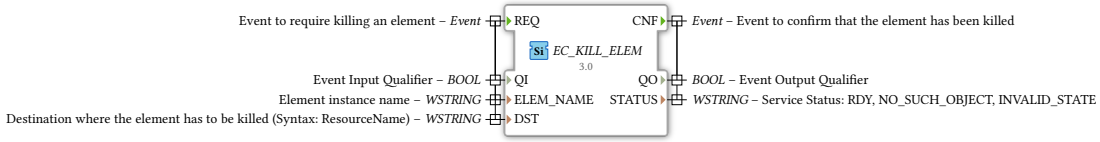
2.4.5.1 EC_KILL_ELEM



• — ○ • — ○ • — ○ •

Einleitung

Der EC_KILL_ELEM Funktionsblock dient zum Beenden von Instanzen gemäß der Zustandsmaschine von IEC 61499 Funktionsblöcken. Er ermöglicht das gezielte Beenden von Funktionsblock-Instanzen, Verbindungen (Ereignis/Daten), Ressourcen oder Geräten innerhalb eines 4diac-Systems.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Ereignis zum Anfordern des Beendens eines Elements

Ereignis-Ausgänge

- **CNF**: Ereignis zur Bestätigung, dass das Element beendet wurde

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangs-Qualifier
- **ELEM_NAME** (WSTRING): Name der Elementinstanz
- **DST** (WSTRING): Zielort, an dem das Element beendet werden muss (Syntax: ResourceName)

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangs-Qualifier
- **STATUS** (WSTRING): Service-Status (RDY, NO_SUCH_OBJECT, INVALID_STATE)

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Funktionsblock reagiert auf das REQ-Ereignis und versucht, das spezifizierte Element (Funktionsblock, Verbindung, Ressource oder Gerät) an der angegebenen Zielressource zu beenden. Nach der Ausführung wird das CNF-Ereignis mit dem entsprechenden Status ausgegeben.

Technische Besonderheiten

- Unterstützt das Beenden verschiedener Elementtypen (FB, Verbindungen, Ressourcen, Geräte)
- Verwendet WSTRING-Datentypen für Elementnamen und Ziele
- Bietet detaillierte Statusrückmeldungen über den Beendigungsvorgang
- Implementiert gemäß IEC 61499 Execution Control Services

Zustandsübersicht

Der Funktionsblock verfügt über mehrere Service-Sequenzen:

- **normal_establishment**: Erfolgreiche Initialisierung
- **unsuccessful_establishment**: Fehlgeschlagene Initialisierung
- **request_confirm**: Erfolgreiche Beendigungsanfrage
- **request_inhibited**: Unterdrückte Beendigungsanfrage
- **request_error**: Fehlerhafte Beendigungsanfrage
- **application_initiated_termination**: Anwendungsinitiierte Beendigung
- **resource_initiated_termination**: Ressourceninitiierte Beendigung

Anwendungsszenarien

- Dynamische Rekonfiguration von Automatisierungssystemen
- Gezieltes Beenden fehlerhafter Komponenten
- Ressourcenmanagement in verteilten Systemen
- Systemwartung und -updates
- Fehlerbehandlung und Systemwiederherstellung

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Execution-Control-Blöcken bietet EC_KILL_ELEM spezifische Funktionen zum Beenden von Elementinstanzen, während ähnliche Blöcke oft Erstellungs- oder Managementfunktionen bereitstellen. Die Fähigkeit, verschiedene Elementtypen (FB, Verbindungen, Ressourcen, Geräte) zu behandeln, macht ihn besonders vielseitig.

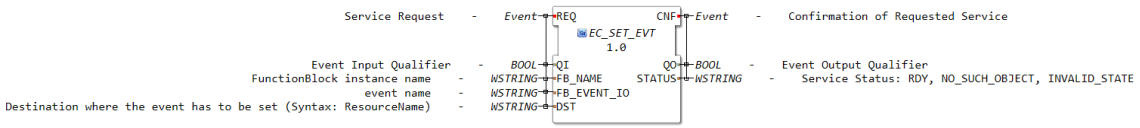
Fazit

EC_KILL_ELEM ist ein essentieller Baustein für Rekonfigurationsaufgaben in IEC 61499-basierten Systemen. Seine Fähigkeit, gezielt Elemente zu beenden, ermöglicht dynamische Systemanpassungen und robuste Fehlerbehandlungsstrategien in industriellen Automatisierungslösungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

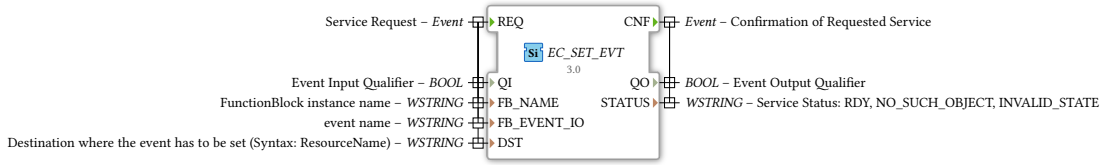
2.4.5.2 EC_SET_EVT



• — ○ • — ○ • — ○ •

Einleitung

Der EC_SET_EVT Funktionsblock dient zum Setzen von Ereignis-Ein- oder -Ausgängen einer Funktionsblock-Instanz. Er ermöglicht die gezielte Auslösung von Ereignissen in anderen Funktionsblöcken und kann beispielsweise als Ereignis-Sniffer eingesetzt werden. Der Baustein gehört zur Kategorie der Rekonfigurationsdienste gemäß IEC 61499-1 Standard.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Service-Anfrage - löst die Ausführung des Setzvorgangs aus

Ereignis-Ausgänge

- **CNF**: Bestätigung der angeforderten Service-Operation

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangs-Qualifier - aktiviert/deaktiviert den Service
- **FB_NAME** (WSTRING): Name der Funktionsblock-Instanz
- **FB_EVENT_IO** (WSTRING): Name des zu setzenden Ereignisses
- **DST** (WSTRING): Zielort, an dem das Ereignis gesetzt werden soll (Syntax: ResourceName)

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangs-Qualifier - zeigt den Status der Service-Ausführung
- **STATUS** (WSTRING): Service-Status (RDY, NO_SUCH_OBJECT, INVALID_STATE)

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der EC_SET_EVT Funktionsblock empfängt über das REQ-Ereignis eine Anfrage zum Setzen eines spezifischen Ereignisses in einer anderen Funktionsblock-Instanz. Über die Eingangsdaten werden die Ziel-Funktionsblock-Instanz (FB_NAME), das zu setzende Ereignis (FB_EVENT_IO) und der Zielort (DST) spezifiziert. Nach erfolgreicher Ausführung wird das CNF-Ereignis mit entsprechendem Status zurückgemeldet.

Technische Besonderheiten

- Implementierung in C++ für GCC-Compiler ab Version 3.0
- Unterstützt Wide-String-Datentypen für internationale Zeichensätze
- Bietet detaillierte Statusrückmeldungen für Fehlerdiagnose
- Arbeitet als Rekonfigurationsdienst zwischen Anwendung und Ressource

Zustandsübersicht

Der Funktionsblock unterstützt mehrere Service-Sequenzen:

- **normal_establishment**: Erfolgreiche Initialisierung
- **unsuccessful_establishment**: Fehlgeschlagene Initialisierung
- **request_confirm**: Erfolgreiche Service-Ausführung
- **request_inhibited**: Unterdrückte Service-Anfrage
- **request_error**: Fehlerhafte Service-Ausführung
- **application_initiated_termination**: Anwendungsseitige Beendigung
- **resource_initiated_termination**: Ressourcenseitige Beendigung

Anwendungsszenarien

- Debugging und Monitoring von Ereignisflüssen
- Automatisierte Testumgebungen
- Dynamische Rekonfiguration von Steuerungssystemen
- Ereignis-basierte Diagnosefunktionen
- Simulation von externen Ereignissen

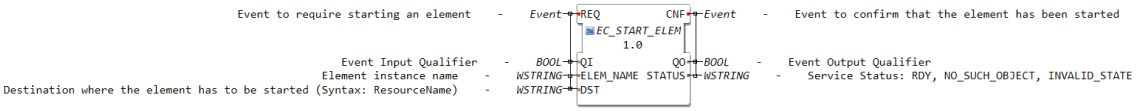
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu Standard-Funktionsblöcken bietet EC_SET_EVT spezielle Rekonfigurationsfähigkeiten, die über normale Ereignisverarbeitung hinausgehen. Er ermöglicht externen Zugriff auf Ereignisse anderer Funktionsblöcke, was für Diagnose- und Testzwecke besonders wertvoll ist.

Fazit

Der EC_SET_EVT Funktionsblock stellt ein leistungsfähiges Werkzeug für erweiterte Steuerungsanwendungen dar, insbesondere für Debugging, Testing und dynamische Rekonfiguration. Seine Fähigkeit, gezielt Ereignisse in anderen Funktionsblock-Instanzen zu setzen, macht ihn zu einem wertvollen Baustein für komplexe Automatisierungssysteme.

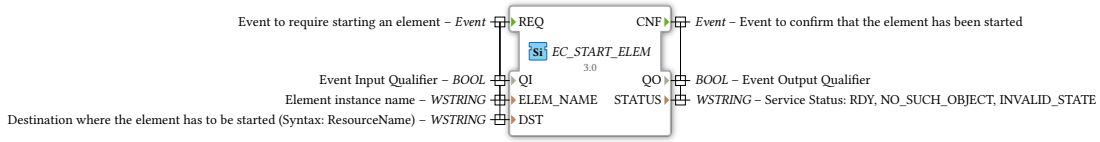
2.4.5.3 EC_START_ELEM



• — ○ • — ○ • — ○ •

Einleitung

Der EC_START_ELEM Funktionsblock dient zum Starten von Instanzen gemäß dem Zustandsautomaten von IEC 61499 Funktionsblöcken. Er ermöglicht das gezielte Starten von Funktionsblock-Instanzen, Verbindungen (Ereignis/Daten), Ressourcen oder Geräten innerhalb eines verteilten Automatisierungssystems.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Ereignis zum Anfordern des Starts eines Elements. Wird mit den Daten ELEM_NAME, DST und QI verknüpft.

Ereignis-Ausgänge

- **CNF**: Ereignis zur Bestätigung, dass das Element gestartet wurde. Wird mit den Daten STATUS und QO verknüpft.

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangs-Qualifier zur Steuerung der Anforderung
- **ELEM_NAME** (WSTRING): Name der Elementinstanz, die gestartet werden soll
- **DST** (WSTRING): Zielort, an dem das Element gestartet werden muss (Syntax: ResourceName)

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangs-Qualifier zur Statusrückmeldung
- **STATUS** (WSTRING): Service-Status mit möglichen Werten: RDY, NO_SUCH_OBJECT, INVALID_STATE

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Funktionsblock reagiert auf das REQ-Ereignis und versucht, das spezifizierte Element (Funktionsblock, Verbindung, Ressource oder Gerät) am angegebenen Zielort zu starten. Der Startvorgang erfolgt gemäß dem IEC 61499-Zustandsautomaten. Bei erfolgreichem Start wird das CNF-Ereignis mit dem Status „RDY“ ausgegeben. Bei Fehlern werden entsprechende Statusmeldungen wie „NO_SUCH_OBJECT“ oder „INVALID_STATE“ zurückgegeben.

Technische Besonderheiten

- Unterstützt Wide-String-Datentypen für Elementnamen und Zielorte
- Implementiert ein Qualifier-System zur Steuerung der Ausführungslogik
- Bietet detaillierte Statusrückmeldungen für verschiedene Fehlerzustände
- Folgt dem IEC 61499-Standard für Ausführungskontrollservices

Zustandsübersicht

Der Funktionsblock verfügt über mehrere Service-Sequenzen:

- **normal_establishment**: Erfolgreiche Initialisierung
- **unsuccessful_establishment**: Fehlgeschlagene Initialisierung
- **request_confirm**: Erfolgreiche Anforderungsbearbeitung
- **request_inhibited**: Unterdrückte Anforderung
- **request_error**: Fehlerhafte Anforderungsbearbeitung
- **application_initiated_termination**: Anwendungsseitige Beendigung
- **resource_initiated_termination**: Ressourcenseitige Beendigung

Anwendungsszenarien

- Dynamisches Starten von Funktionsblöcken in laufenden Systemen
- Rekonfiguration von Automatisierungssystemen zur Laufzeit
- Verteiltes Management von Ressourcen und Geräten
- Fehlertolerante Systeme mit Wiederanlauf-Funktionalität

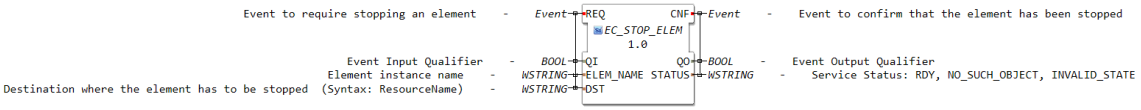
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen Start-Funktionsblöcken bietet EC_START_ELEM erweiterte Funktionalität für verteilte Systeme mit spezifischer Zielortangabe und detaillierter Statusrückmeldung. Er ist Teil der Rekonfigurationsservices und unterstützt verschiedene Elementtypen über reine Funktionsblöcke hinaus.

Fazit

EC_START_ELEM ist ein leistungsstarker Funktionsblock für die dynamische Steuerung von Automatisierungssystemkomponenten. Seine Fähigkeit, verschiedene Elementtypen an spezifischen Zielorten zu starten, macht ihn besonders wertvoll für rekonfigurierbare und adaptive Steuerungssysteme gemäß IEC 61499.

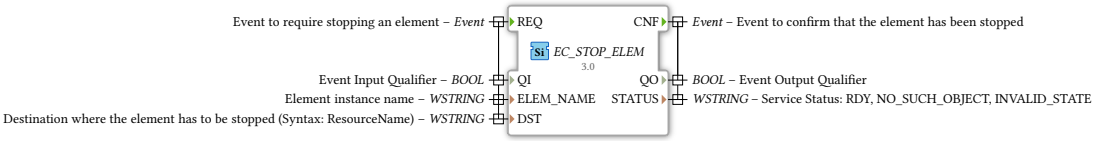
2.4.5.4 EC_STOP_ELEM



• — ○ • — ○ • — ○ •

Einleitung

Der EC_STOP_ELEM Funktionsblock dient zum Stoppen von Instanzen gemäß dem Zustandsautomaten von IEC 61499 Funktionsblöcken. Er kann Funktionsblock-Instanzen, Verbindungen (Ereignis/Daten), Ressourcen oder Geräte anhalten. Dieser Baustein gehört zur Kategorie der Rekonfigurationsdienste und implementiert einen Ausführungssteuerungsdienst gemäß IEC 61499-1.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Ereignis zum Anfordern des Stoppens eines Elements

Ereignis-Ausgänge

- **CNF**: Ereignis zur Bestätigung, dass das Element gestoppt wurde

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangsqualifizierer
- **ELEM_NAME** (WSTRING): Element-Instanzname
- **DST** (WSTRING): Ziel, an dem das Element gestoppt werden muss (Syntax: ResourceName)

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifizierer
- **STATUS** (WSTRING): Dienststatus: RDY, NO_SUCH_OBJECT, INVALID_STATE

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der EC_STOP_ELEM Funktionsblock verarbeitet Stopp-Anforderungen für verschiedene Elemente innerhalb eines IEC 61499-Systems. Bei Eingang eines REQ-Ereignisses mit aktiviertem QI-Eingang wird der Stopp-Vorgang für das spezifizierte Element initiiert. Nach Abschluss des Stopp-Vorgangs wird ein CNF-Ereignis mit dem entsprechenden Status ausgegeben.

Technische Besonderheiten

- Unterstützt das Stoppen verschiedener Elementtypen: FBs, Verbindungen, Ressourcen und Geräte
- Verwendet WSTRING-Datentypen für Elementnamen und Ziele
- Bietet detaillierte Statusrückmeldungen über mögliche Fehlerzustände
- Implementiert gemäß IEC 61499-1 Standard für Ausführungssteuerungsdienste

Zustandsübersicht

Der Funktionsblock verfügt über mehrere Service-Sequenzen:

- **normal_establishment**: Erfolgreiche Initialisierung
- **unsuccessful_establishment**: Fehlgeschlagene Initialisierung
- **request_confirm**: Erfolgreiche Stopp-Anforderung
- **request_inhibited**: Unterdrückte Stopp-Anforderung
- **request_error**: Fehlerhafte Stopp-Anforderung
- **application_initiated_termination**: Anwendungsinitiierte Beendigung
- **resource_initiated_termination**: Ressourceninitiierte Beendigung

Anwendungsszenarien

- Dynamisches Rekonfigurieren von Steuerungssystemen
- Geplantes Herunterfahren von Systemkomponenten
- Fehlerbehandlung durch gezieltes Stoppen fehlerhafter Komponenten
- Wartungsarbeiten an laufenden Systemen

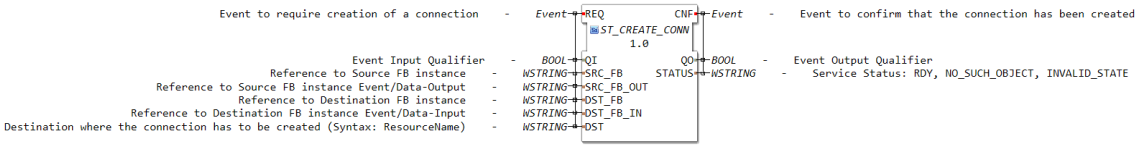
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Rekonfigurationsbausteinen bietet EC_STOP_ELEM eine spezialisierte Funktion zum gezielten Stoppen einzelner Systemkomponenten. Während allgemeine Rekonfigurationsbausteine oft komplexere Operationen unterstützen, konzentriert sich dieser Baustein auf die spezifische Aufgabe des kontrollierten Anhaltens.

Fazit

EC_STOP_ELEM ist ein essentieller Baustein für dynamische Steuerungssysteme, die Rekonfiguration zur Laufzeit erfordern. Seine klare Schnittstelle und zuverlässige Statusrückmeldung machen ihn zu einem wertvollen Werkzeug für die Entwicklung wartbarer und flexibler Automatisierungslösungen gemäß IEC 61499.

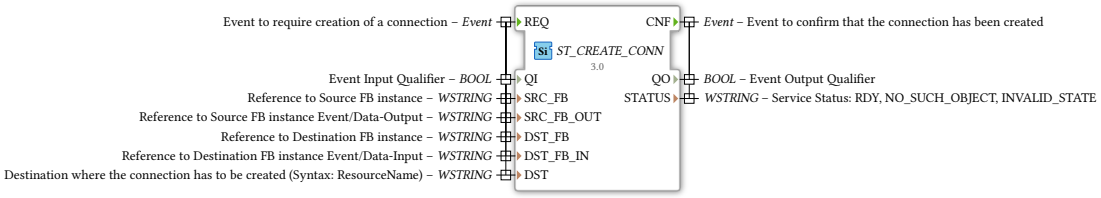
2.4.5.5 ST_CREATE_CONN



• — ○ • — ○ • — ○ •

Einleitung

Der ST_CREATE_CONN Funktionsblock dient zur Erstellung von Ereignis- oder Datenverbindungen zwischen zwei Funktionsblock-Instanzen zur Laufzeit. Dieser Baustein ermöglicht die dynamische Rekonfiguration von Verbindungen in einem 4diac-System und ist als struktureller Service klassifiziert.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Ereignis zur Anforderung der Erstellung einer Verbindung

Ereignis-Ausgänge

- **CNF**: Ereignis zur Bestätigung, dass die Verbindung erstellt wurde

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangsqualifizierer
- **SRC_FB** (WSTRING): Referenz zur Quell-FB-Instanz
- **SRC_FB_OUT** (WSTRING): Referenz zum Ereignis-/Daten-Ausgang der Quell-FB-Instanz
- **DST_FB** (WSTRING): Referenz zur Ziel-FB-Instanz
- **DST_FB_IN** (WSTRING): Referenz zum Ereignis-/Daten-Eingang der Ziel-FB-Instanz
- **DST** (WSTRING): Zielort, an dem die Verbindung erstellt werden muss (Syntax: ResourceName)

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifizierer
- **STATUS** (WSTRING): Service-Status (RDY, NO_SUCH_OBJECT, INVALID_STATE)

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der Funktionsblock erstellt auf Anforderung (REQ-Ereignis) eine Verbindung zwischen einer Quell- und einer Ziel-Funktionsblock-Instanz. Die Verbindung kann sowohl für Ereignisse als auch für Datenströme verwendet werden. Nach erfolgreicher Verarbeitung wird eine Bestätigung (CNF-Ereignis) mit dem entsprechenden Status ausgegeben.

Technische Besonderheiten

- Unterstützt dynamische Rekonfiguration zur Laufzeit
- Verwendet Wide-String-Datentypen für alle Referenzen
- Bietet detaillierte Statusrückmeldungen
- Implementiert als struktureller Service gemäß IEC 61499-1

Zustandsübersicht

Der Baustein unterstützt mehrere Service-Sequenzen:

- Normale Verbindungserstellung

- Erfolglose Verbindungserstellung
- Anforderungs-Bestätigungs-Zyklus
- Anforderungs-Verhinderung
- Fehlerbehandlung
- Anwendungs- und Ressourcen-initiierte Beendigung

Anwendungsszenarien

- Dynamische Systemrekonfiguration
- Laufzeit-Anpassung von Steuerungslogik
- Flexible Verbindungsverwaltung in verteilten Systemen
- Adaptive Automatisierungssysteme

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu statisch konfigurierten Verbindungen ermöglicht ST_CREATE_CONN die dynamische Erstellung von Verbindungen zur Laufzeit. Andere Rekonfigurationsbausteine fokussieren sich typischerweise auf das Erstellen oder Löschen von FB-Instanzen, während dieser Baustein speziell für Verbindungsmanagement optimiert ist.

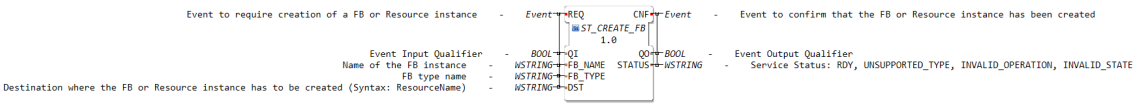
Fazit

ST_CREATE_CONN ist ein essentieller Baustein für Systeme, die dynamische Rekonfiguration erfordern. Seine Fähigkeit, Verbindungen zwischen Funktionsblöcken zur Laufzeit zu erstellen, macht ihn besonders wertvoll für adaptive und flexible Automatisierungslösungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

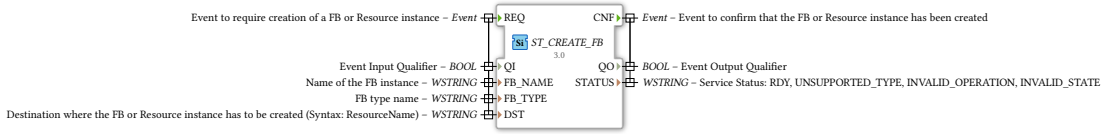
2.4.5.6 ST_CREATE_FB



• — ○ • — ○ • — ○ •

Einleitung

Der ST_CREATE_FB Funktionsblock dient zur dynamischen Erstellung von Funktionsblock- oder Resource-Instanzen während der Laufzeit eines IEC 61499-Systems. Dieser Baustein ermöglicht die Rekonfiguration von Automatisierungssystemen durch das Erzeugen neuer Funktionsblöcke basierend auf bestimmten FB-Typen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Ereignis zur Anforderung der Erstellung einer FB- oder Resource-Instanz

Ereignis-Ausgänge

- **CNF**: Ereignis zur Bestätigung, dass die FB- oder Resource-Instanz erstellt wurde

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangsqualifizierer
- **FB_NAME** (WSTRING): Name der zu erstellenden FB-Instanz
- **FB_TYPE** (WSTRING): FB-Typname
- **DST** (WSTRING): Zielort, an dem die FB- oder Resource-Instanz erstellt werden soll (Syntax: ResourceName)

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifizierer
- **STATUS** (WSTRING): Service-Status (RDY, UNSUPPORTED_TYPE, INVALID_OPERATION, INVALID_STATE)

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der ST_CREATE_FB Funktionsblock empfängt über das REQ-Ereignis eine Anforderung zur Erstellung einer neuen Funktionsblock-Instanz. Dabei werden die erforderlichen Parameter (FB_NAME, FB_TYPE, DST) mitgeliefert. Nach erfolgreicher Verarbeitung der Anforderung gibt der Baustein über das CNF-Ereignis eine Bestätigung zurück, die den Status der Operation im STATUS-Ausgang enthält.

Technische Besonderheiten

- Unterstützt dynamische Rekonfiguration von IEC 61499-Systemen
- Verwendet WSTRING-Datentypen für internationale Zeichensatzunterstützung
- Bietet detaillierte Statusrückmeldungen für Fehlerdiagnose
- Implementiert als Structural Service gemäß IEC 61499-1 Standard

Zustandsübersicht

Der Funktionsblock verfügt über mehrere Service-Sequenzen:

- **normal_establishment**: Erfolgreiche Initialisierung
- **unsuccessful_establishment**: Fehlgeschlagene Initialisierung
- **request_confirm**: Erfolgreiche Anforderungsbearbeitung
- **request_inhibited**: Unterdrückte Anforderung
- **request_error**: Fehlerhafte Anforderungsbearbeitung
- **application_initiated_termination**: Anwendungsinitiierte Beendigung
- **resource_initiated_termination**: Resource-initiierte Beendigung

Anwendungsszenarien

- Dynamische Erweiterung von Automatisierungssystemen
- Adaptive Steuerungssysteme, die sich an veränderte Produktionsanforderungen anpassen
- Systeme mit modularem Aufbau, bei denen Komponenten zur Laufzeit hinzugefügt werden
- Rekonfigurierbare Fertigungsanlagen

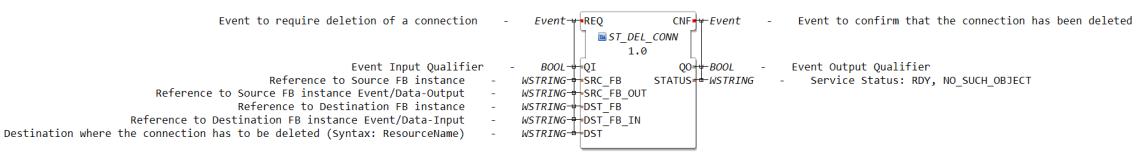
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu statischen Funktionsblöcken bietet ST_CREATE_FB die einzigartige Fähigkeit, Instanzen zur Laufzeit zu erzeugen. Während Standard-Funktionsblöcke zur Kompilierungszeit festgelegt werden, ermöglicht dieser Baustein eine flexible Systemarchitektur.

Fazit

Der ST_CREATE_FB Funktionsblock ist ein essentielles Werkzeug für die Entwicklung dynamischer und rekonfigurierbarer Automatisierungssysteme nach IEC 61499. Seine Fähigkeit, Funktionsblock-Instanzen zur Laufzeit zu erzeugen, macht ihn besonders wertvoll für Anwendungen, die Anpassungsfähigkeit und Flexibilität erfordern.

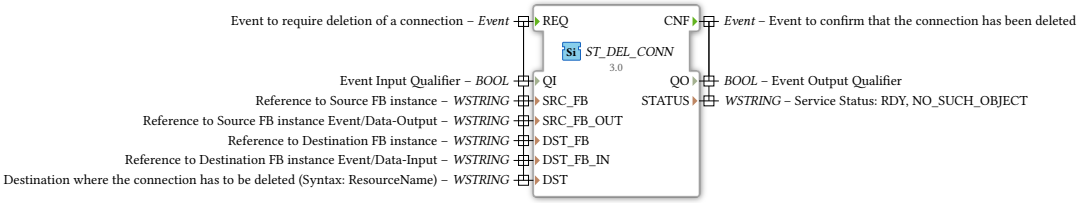
2.4.5.7 ST_DEL_CONN



• — ○ • — ○ • — ○ •

Einleitung

Der Funktionsblock ST_DEL_CONN dient zum Löschen von Ereignis- oder Datenverbindungen zwischen zwei Funktionsblock-Instanzen innerhalb eines 4diac-Systems. Er ermöglicht die dynamische Rekonfiguration von Verbindungen zur Laufzeit und gehört zur Kategorie der Strukturellen Dienste gemäß IEC 61499-1.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Ereignis zum Anfordern des Löschens einer Verbindung

Ereignis-Ausgänge

- **CNF**: Ereignis zur Bestätigung, dass die Verbindung gelöscht wurde

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangs-Qualifier
- **SRC_FB** (WSTRING): Referenz zur Quell-FB-Instanz
- **SRC_FB_OUT** (WSTRING): Referenz zum Ereignis-/Daten-Ausgang der Quell-FB-Instanz
- **DST_FB** (WSTRING): Referenz zur Ziel-FB-Instanz
- **DST_FB_IN** (WSTRING): Referenz zum Ereignis-/Daten-Eingang der Ziel-FB-Instanz
- **DST** (WSTRING): Ziel, wo die Verbindung gelöscht werden muss (Syntax: ResourceName)

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangs-Qualifier
- **STATUS** (WSTRING): Dienststatus: RDY, NO_SUCH_OBJECT

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Beim Auslösen des REQ-Ereignisses wird die spezifizierte Verbindung zwischen den angegebenen Funktionsblock-Instanzen gelöscht. Der Block erwartet vollständige Referenzen sowohl für die Quell- als auch die Ziel-FB-Instanzen inklusive deren Ein- und Ausgänge. Nach erfolgreicher Verarbeitung wird das CNF-Ereignis mit dem entsprechenden Status ausgegeben.

Technische Besonderheiten

- Unterstützt das Löschen sowohl von Ereignis- als auch Datenverbindungen
- Arbeitet mit Wide-String-Datentypen für alle Referenzparameter
- Bietet Statusrückmeldung über erfolgreiche oder fehlgeschlagene Operationen
- Implementiert gemäß IEC 61499-1 Standard für strukturelle Dienste

Zustandsübersicht

Der Funktionsblock unterstützt verschiedene Service-Sequenzen:

- Normale Initialisierung (INIT+ ⟶ INITO+)
- Fehlgeschlagene Initialisierung (INIT+ ⟶ INITO-)
- Erfolgreiche Anfrage (REQ+ ⟶ CNF+)
- Unterdrückte Anfrage (REQ- ⟶ CNF-)
- Fehlerhafte Anfrage (REQ+ ⟶ CNF-)
- Anwendungs- und ressourcenseitige Beendigung

Anwendungsszenarien

- Dynamische Rekonfiguration von Steuerungssystemen
- Entfernen von Verbindungen zur Laufzeit
- Fehlerbehandlung bei Verbindungsproblemen
- Adaptive Systeme, die sich an veränderte Anforderungen anpassen müssen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Rekonfigurations-Bausteinen spezialisiert sich ST_DEL_CONN ausschließlich auf das Löschen von Verbindungen, während andere Bausteine möglicherweise das Erstellen oder Modifizieren von Verbindungen unterstützen.

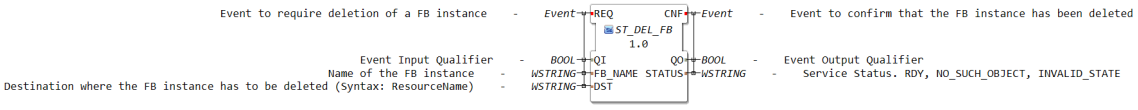
Fazit

ST_DEL_CONN ist ein essentieller Baustein für Systeme, die dynamische Rekonfiguration erfordern. Seine klare Schnittstellendefinition und zuverlässige Statusrückmeldung machen ihn zu einem zuverlässigen Werkzeug für die Verwaltung von Verbindungen in IEC 61499-basierten Steuerungssystemen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

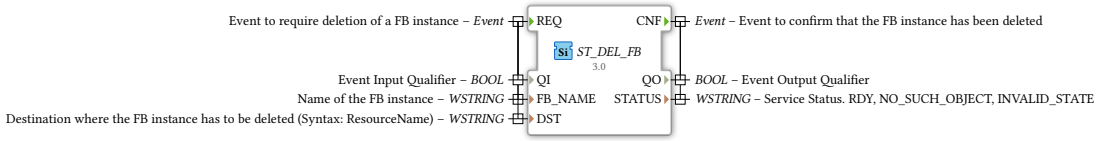
2.4.5.8 ST_DEL_FB



• — ○ • — ○ • — ○ •

Einleitung

Der ST_DEL_FB Funktionsblock dient zum Löschen einer Funktionsblock-Instanz eines bestimmten Funktionsblock-Typs. Dieser Baustein gehört zur Kategorie der Rekonfigurationsdienste und ermöglicht die dynamische Verwaltung von Funktionsblock-Instanzen zur Laufzeit.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Ereignis zum Anfordern des Löschens einer FB-Instanz

Ereignis-Ausgänge

- **CNF**: Ereignis zur Bestätigung, dass die FB-Instanz gelöscht wurde

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangs-Qualifier
- **FB_NAME** (WSTRING): Name der zu löschenden FB-Instanz
- **DST** (WSTRING): Zielort, an dem die FB-Instanz gelöscht werden muss (Syntax: ResourceName)

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangs-Qualifier
- **STATUS** (WSTRING): Servicestatus (RDY, NO_SUCH_OBJECT, INVALID_STATE)

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Der ST_DEL_FB Funktionsblock reagiert auf das REQ-Ereignis und versucht, die spezifizierte Funktionsblock-Instanz am angegebenen Zielort zu löschen. Bei erfolgreicher Ausführung wird ein CNF-Ereignis mit entsprechendem Status ausgegeben. Der Baustein unterstützt verschiedene Betriebszustände und Fehlerbehandlungen.

Technische Besonderheiten

- Unterstützt dynamische Rekonfiguration von Funktionsblöcken
- Verwendet WSTRING-Datentypen für internationale Zeichensätze
- Bietet detaillierte Statusrückmeldungen über den Löschvorgang
- Implementiert gemäß IEC 61499-1 Standard für strukturelle Dienste

Zustandsübersicht

Der Funktionsblock unterstützt mehrere Service-Sequenzen:

- Normale Einrichtung (normal_establishment)
- Erfolglose Einrichtung (unsuccessful_establishment)
- Anforderungsbestätigung (request_confirm)
- Unterdrückte Anforderung (request_inhibited)
- Anforderungsfehler (request_error)
- Anwendungsinitiierte Beendigung (application_initiated_termination)
- Ressourceninitiierte Beendigung (resource_initiated_termination)

Anwendungsszenarien

- Dynamische Rekonfiguration von Steuerungssystemen
- Ressourcenmanagement in verteilten Automatisierungssystemen
- Laufzeit-Anpassung von Funktionsblock-Architekturen
- Wartung und Updates von Automatisierungsanlagen

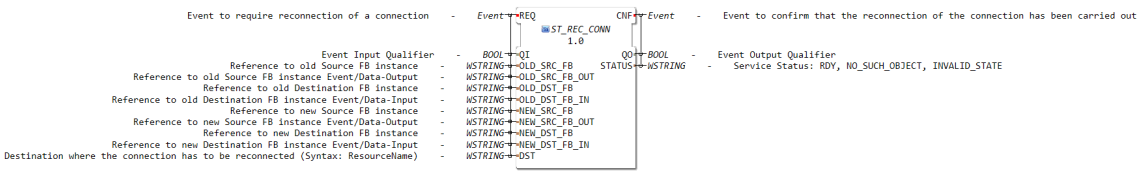
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu statischen Funktionsblöcken ermöglicht ST_DEL_FB die dynamische Verwaltung von FB-Instanzen zur Laufzeit. Er unterscheidet sich von Erstellungs-Bausteinen durch seine spezifische Löschfunktionalität und bietet damit komplementäre Funktionalität für vollständige Lebenszyklusverwaltung.

Fazit

Der ST_DEL_FB Funktionsblock ist ein essentielles Werkzeug für Systeme, die dynamische Rekonfiguration erfordern. Seine robuste Fehlerbehandlung und detaillierte Statusrückmeldungen machen ihn zu einer zuverlässigen Lösung für die Verwaltung von Funktionsblock-Instanzen in komplexen Automatisierungsumgebungen.

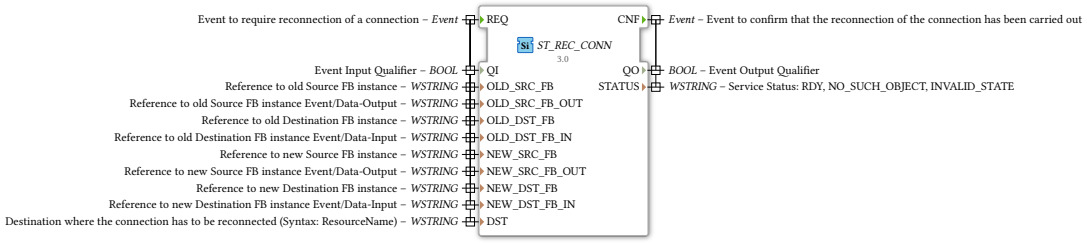
2.4.5.9 ST_REC_CONN



• — ○ • — ○ • — ○ •

Einleitung

Der ST_REC_CONN Funktionsblock dient zur dynamischen Neukonfiguration von Verbindungen zwischen Funktionsbausteinen in 4diac-Systemen. Er ermöglicht das Umleiten von Ereignis- oder Datenverbindungen von bestehenden Quell- und Ziel-Funktionsbausteinen zu neuen Instanzen während der Laufzeit.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet den Rekonfigurationsprozess für eine Verbindung

Ereignis-Ausgänge

- **CNF**: Bestätigt die erfolgreiche oder fehlgeschlagene Durchführung der Verbindungsrekonfiguration

Daten-Eingänge

- **QI** (BOOL): Qualifiziert die Ereigniseingabe
- **OLD_SRC_FB** (WSTRING): Referenz zur alten Quell-FB-Instanz
- **OLD_SRC_FB_OUT** (WSTRING): Referenz zum alten Quell-FB-Ausgang (Ereignis/Daten)
- **OLD_DST_FB** (WSTRING): Referenz zur alten Ziel-FB-Instanz
- **OLD_DST_FB_IN** (WSTRING): Referenz zum alten Ziel-FB-Eingang (Ereignis/Daten)
- **NEW_SRC_FB** (WSTRING): Referenz zur neuen Quell-FB-Instanz
- **NEW_SRC_FB_OUT** (WSTRING): Referenz zum neuen Quell-FB-Ausgang (Ereignis/Daten)
- **NEW_DST_FB** (WSTRING): Referenz zur neuen Ziel-FB-Instanz
- **NEW_DST_FB_IN** (WSTRING): Referenz zum neuen Ziel-FB-Eingang (Ereignis/Daten)
- **DST** (WSTRING): Zielort für die Rekonfiguration (Syntax: ResourceName)

Daten-Ausgänge

- **QO** (BOOL): Qualifiziert die Ereignisausgabe
- **STATUS** (WSTRING): Service-Status (RDY, NO_SUCH_OBJECT, INVALID_STATE)

Adapter

Keine Adapter-Schnittstellen vorhanden.

Funktionsweise

Beim Eintreffen des REQ-Ereignisses analysiert der Funktionsblock die übergebenen Parameter und führt folgende Schritte durch:

1. Identifizierung der bestehenden Verbindung zwischen OLD_SRC_FB und OLD_DST_FB
2. Trennung der bestehenden Verbindung
3. Aufbau einer neuen Verbindung zwischen NEW_SRC_FB und NEW_DST_FB
4. Rückmeldung des Operationsstatus über das CNF-Ereignis

Der QI-Eingang steuert dabei die Ausführung: Bei FALSE wird die Operation nicht durchgeführt.

Technische Besonderheiten

- Unterstützt sowohl Ereignis- als auch Datenverbindungen
- Verwendet WSTRING-Datentypen für alle Referenzparameter
- Arbeitet mit Ressourcen-spezifischen Zielangaben
- Bietet detaillierte Statusrückmeldungen
- Implementiert gemäß IEC 61499-1 Standard für Rekonfiguration

Zustandsübersicht

Der Funktionsblock arbeitet nach einem einfachen Anfrage-Bestätigungs-Prinzip:

- Wartezustand: Keine aktive Operation
- Ausführungszustand: Verarbeitung der REQ-Anfrage
- Abschlusszustand: Senden der CNF-Bestätigung

Anwendungsszenarien

- Dynamische Systemrekonfiguration während des Betriebs
- Hot-Swapping von Funktionsbausteinen
- Lastverteilung durch Umleitung von Verbindungen
- Fehlertolerante Systeme mit redundanten Komponenten
- Adaptive Steuerungssysteme mit wechselnden Betriebsmodi

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu statischen Verbindungen in 4diac ermöglicht ST_REC_CONN:

- Dynamische Anpassung von Verbindungen zur Laufzeit
- Höhere Flexibilität in adaptiven Systemen
- Komplexere Rekonfigurationsszenarien im Vergleich zu einfachen Verbindungsblöcken

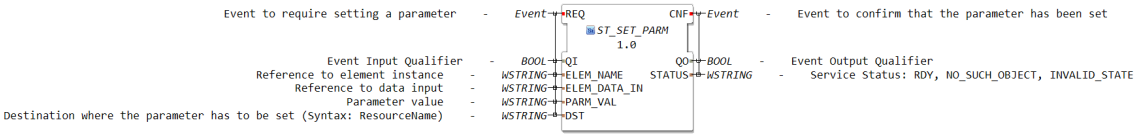
Fazit

ST_REC_CONN ist ein essentieller Baustein für flexible und adaptive Automatisierungssysteme, die dynamische Rekonfiguration während des Betriebs erfordern. Seine umfassende Parametrisierung ermöglicht präzise Steuerung von Verbindungsänderungen, während die Statusrückmeldungen zuverlässige Überwachung der Rekonfigurationsprozesse gewährleisten.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

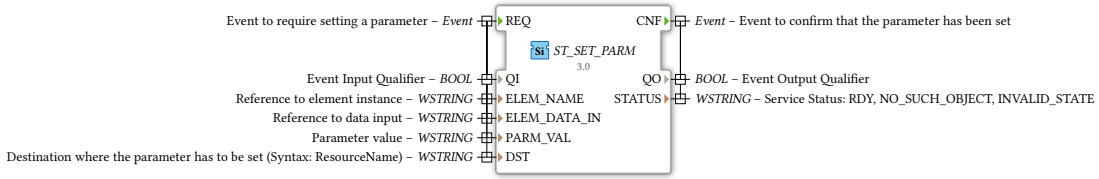
2.4.5.10 ST_SET_PARM



• — ○ • — ○ • — ○ •

Einleitung

Der ST_SET_PARM Funktionsblock dient zum Setzen von Parametern für Dateneingänge von Funktionsblöcken, Ressourcen oder Geräteinstanzen innerhalb eines 4diac-Systems. Dieser Baustein ermöglicht die dynamische Konfiguration von Systemparametern zur Laufzeit.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet den Vorgang zum Setzen eines Parameters. Wird mit den Daten ELEM_NAME, ELEM_DATA_IN, PARM_VAL, DST und QI assoziiert.

Ereignis-Ausgänge

- **CNF**: Bestätigt, dass der Parameter erfolgreich gesetzt wurde. Wird mit den Daten STATUS und QO assoziiert.

Daten-Eingänge

- **QI** (BOOL): Ereignis-Eingangsqualifikator
- **ELEM_NAME** (WSTRING): Referenz zur Elementinstanz
- **ELEM_DATA_IN** (WSTRING): Referenz zum Dateneingang
- **PARM_VAL** (WSTRING): Parameterwert

- **DST** (WSTRING): Zielort, an dem der Parameter gesetzt werden muss (Syntax: ResourceName)

Daten-Ausgänge

- **QO** (BOOL): Ereignis-Ausgangsqualifikator
- **STATUS** (WSTRING): Servicestatus (RDY, NO_SUCH_OBJECT, INVALID_STATE)

Adapter

Keine Adapter vorhanden.

Funktionsweise

Beim Eintreffen des REQ-Ereignisses versucht der ST_SET_PARM Funktionsblock, den angegebenen Parameterwert (PARM_VAL) für den spezifizierten Dateneingang (ELEM_DATA_IN) der Zielinstanz (ELEM_NAME) am angegebenen Zielort (DST) zu setzen. Nach Abschluss des Vorgangs wird das CNF-Ereignis mit dem entsprechenden Status ausgegeben.

Technische Besonderheiten

- Unterstützt Wide-String-Datentypen für alle textbasierten Parameter
- Bietet detaillierte Statusrückmeldungen über den Erfolg oder Fehler der Operation
- Ermöglicht die Konfiguration über Ressourcengrenzen hinweg
- Implementiert als Rekonfigurationsdienst gemäß IEC 61499-1 Standard

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf REQ-Ereignis
2. **Verarbeitungszustand**: Setzt den Parameter basierend auf den Eingabedaten
3. **Bestätigungszustand**: Sendet CNF-Ereignis mit Statusinformationen

Anwendungsszenarien

- Dynamische Anpassung von Systemparametern während des Betriebs
- Konfiguration von Funktionsblock-Parametern in verteilten Systemen
- Automatisierte Parametrierung in Produktionsanlagen
- Wartung und Optimierung von Steuerungssystemen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu statischen Parametrierungsmethoden bietet ST_SET_PARM die Möglichkeit, Parameter zur Laufzeit zu ändern, was eine höhere Flexibilität in adaptiven Steuerungssystemen ermöglicht. Andere Konfigurationsbausteine arbeiten typischerweise nur während der Initialisierungsphase.

Fazit

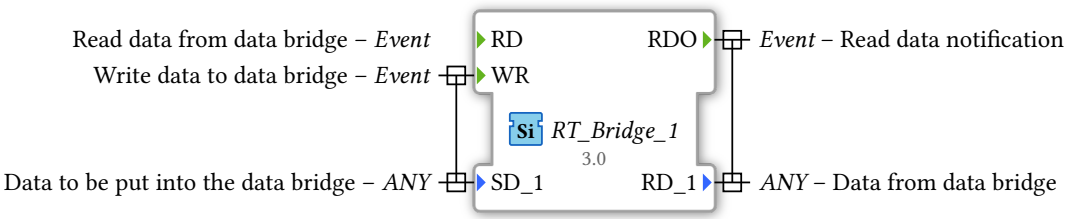
Der ST_SET_PARM Funktionsblock ist ein essentielles Werkzeug für die dynamische Rekonfiguration von 4diac-Systemen. Seine Fähigkeit, Parameter zur Laufzeit zu setzen, macht ihn besonders wertvoll für Anwendungen, die Anpassungsfähigkeit und Flexibilität erfordern. Die klare Schnittstellendefinition und detaillierte Statusrückmeldung gewährleisten eine zuverlässige Integration in komplexe Automatisierungssysteme.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.6 rtevents

2.4.6.1 RT_Bridge_1



• – ◦ • – ◦ • – ◦ •

Einleitung

Der RT_Bridge_1 Funktionsblock dient als Entkopplungsbrücke für eine Datenverbindung zwischen Echtzeit-Ereignisketten. Er ermöglicht den sicheren Datenaustausch zwischen verschiedenen Ausführungskontexten.

Schnittstellenstruktur

Ereignis-Eingänge

- **RD**: Daten aus der Brücke lesen.
- **WR**: Daten in die Brücke schreiben.

Ereignis-Ausgänge

- **RDO**: Benachrichtigung über gelesene Daten.

Daten-Eingänge

- **SD_1** (ANY): Daten, die in die Brücke geschrieben werden sollen.

Daten-Ausgänge

- **RD_1** (ANY): Daten, die aus der Brücke gelesen wurden.

Funktionsweise

Der Baustein speichert die über `WR` empfangenen Daten zwischen und stellt sie bei `RD` am Ausgang zur Verfügung. Dies entkoppelt den Schreib- vom Leseprozess zeitlich.

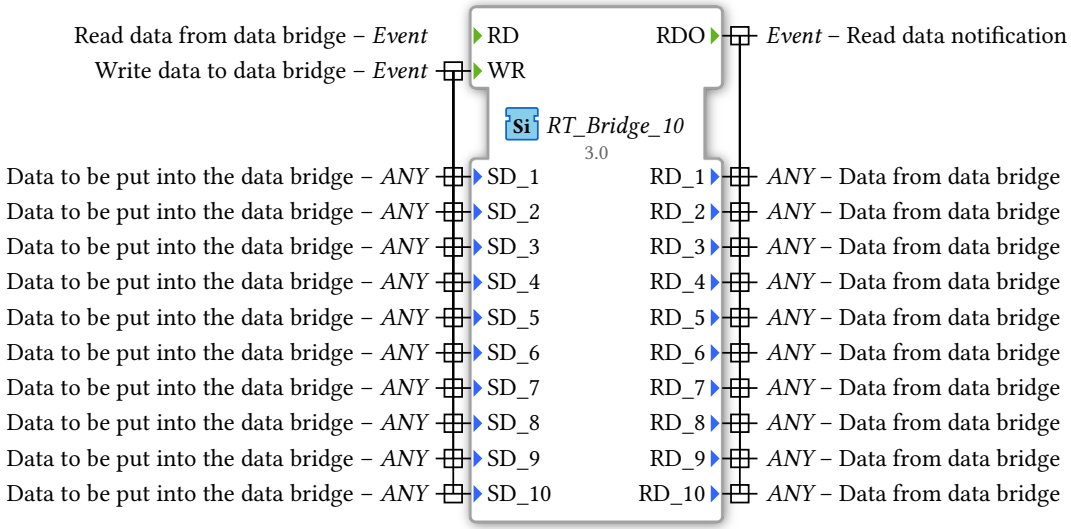
Metadaten

Attribut	Wert
Copyright	(c) 2023 Johannes Kepler University Linz
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	eclipse4diac::rtevents

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.6.2 RT_Bridge_10



• – ◦ • – ◦ • – ◦ •

Einleitung

Der RT_Bridge_10 Funktionsblock dient als Entkopplungsbrücke für 10 Datenverbindungen zwischen Echtzeit-Ereignisketten.

Schnittstellenstruktur

Ereignis-Eingänge

- **RD**: Daten lesen.
- **WR**: Daten schreiben.

Ereignis-Ausgänge

- **RDO**: Lese-Bestätigung.

Daten-Eingänge

- **SD_1** - **SD_10** (ANY): Eingangsdaten.

Daten-Ausgänge

- **RD_1** - **RD_10** (ANY): Ausgangsdaten.

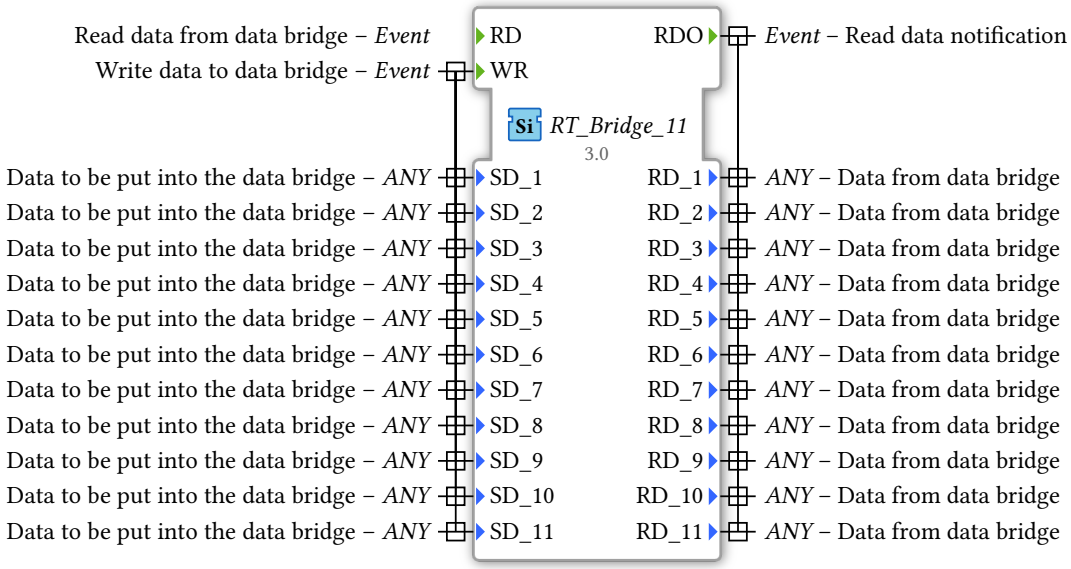
Metadaten

Attribut	Wert
Copyright	(c) 2023 Johannes Kepler University Linz
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	eclipse4diac::rtevents

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.6.3 RT_Bridge_11



• – ◦ • – ◦ • – ◦ •

Einleitung

Der RT_Bridge_11 Funktionsblock dient als Entkopplungsbrücke für 11 Datenverbindungen zwischen Echtzeit-Ereignisketten.

Schnittstellenstruktur

Ereignis-Eingänge

- **RD**: Daten lesen.
- **WR**: Daten schreiben.

Ereignis-Ausgänge

- **RDO**: Lese-Bestätigung.

Daten-Eingänge

- **SD_1** - **SD_11** (ANY): Eingangsdaten.

Daten-Ausgänge

- **RD_1** - **RD_11** (ANY): Ausgangsdaten.

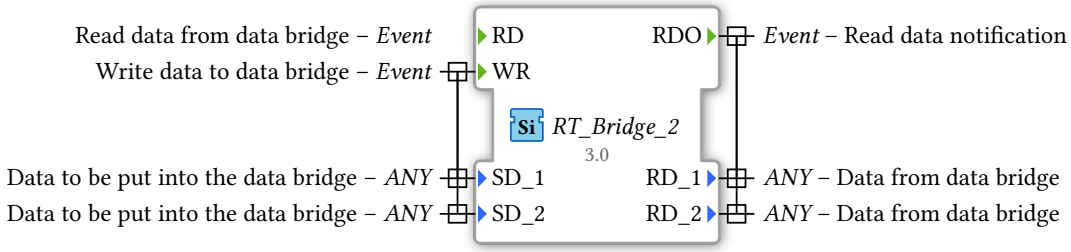
Metadaten

Attribut	Wert
Copyright	(c) 2023 Johannes Kepler University Linz
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	eclipse4diac::rtevents

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.6.4 RT_Bridge_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Der RT_Bridge_2 Funktionsblock dient als Entkopplungsbrücke für 2 Datenverbindungen zwischen Echtzeit-Ereignisketten.

Schnittstellenstruktur

Ereignis-Eingänge

- **RD**: Daten lesen.
- **WR**: Daten schreiben.

Ereignis-Ausgänge

- **RDO**: Lese-Bestätigung.

Daten-Eingänge

- **SD_1** - **SD_2** (ANY): Eingangsdaten.

Daten-Ausgänge

- **RD_1** - **RD_2** (ANY): Ausgangsdaten.

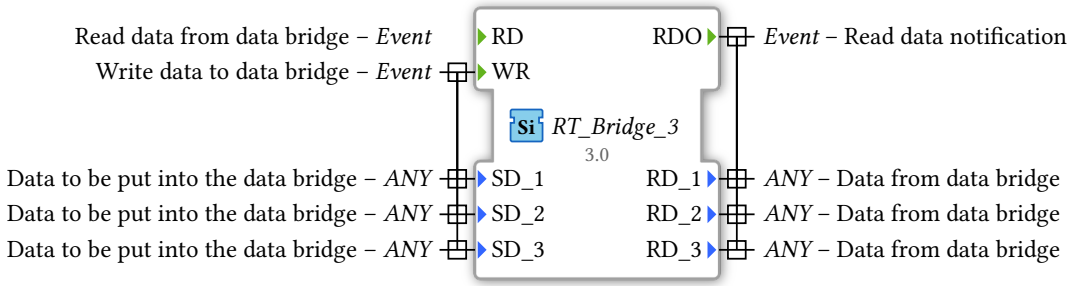
Metadaten

Attribut	Wert
Copyright	(c) 2023 Johannes Kepler University Linz
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	eclipse4diac::rtevents

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.6.5 RT_Bridge_3



• – ◦ • – ◦ • – ◦ •

Einleitung

Der RT_Bridge_3 Funktionsblock dient als Entkopplungsbrücke für 3 Datenverbindungen zwischen Echtzeit-Ereignisketten.

Schnittstellenstruktur

Ereignis-Eingänge

- **RD**: Daten lesen.
- **WR**: Daten schreiben.

Ereignis-Ausgänge

- **RDO**: Lese-Bestätigung.

Daten-Eingänge

- **SD_1** - **SD_3** (ANY): Eingangsdaten.

Daten-Ausgänge

- **RD_1** - **RD_3** (ANY): Ausgangsdaten.

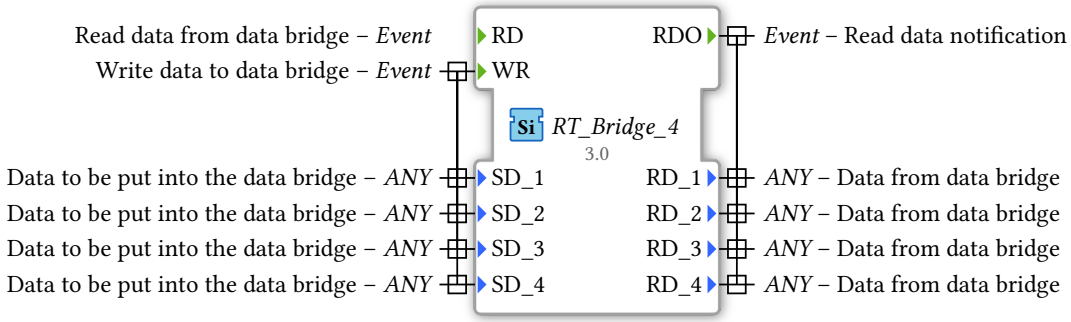
Metadaten

Attribut	Wert
Copyright	(c) 2023 Johannes Kepler University Linz
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	eclipse4diac::rtevents

🌐 **Passende Themen-Unterseiten auf ms-muc-docs.de**

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.6.6 RT_Bridge_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Der RT_Bridge_4 Funktionsblock dient als Entkopplungsbrücke für 4 Datenverbindungen zwischen Echtzeit-Ereignisketten.

Schnittstellenstruktur

Ereignis-Eingänge

- **RD**: Daten lesen.
- **WR**: Daten schreiben.

Ereignis-Ausgänge
- **RDO**: Lese-Bestätigung.

Daten-Eingänge
- **SD_1** - **SD_4** (ANY): Eingangsdaten.

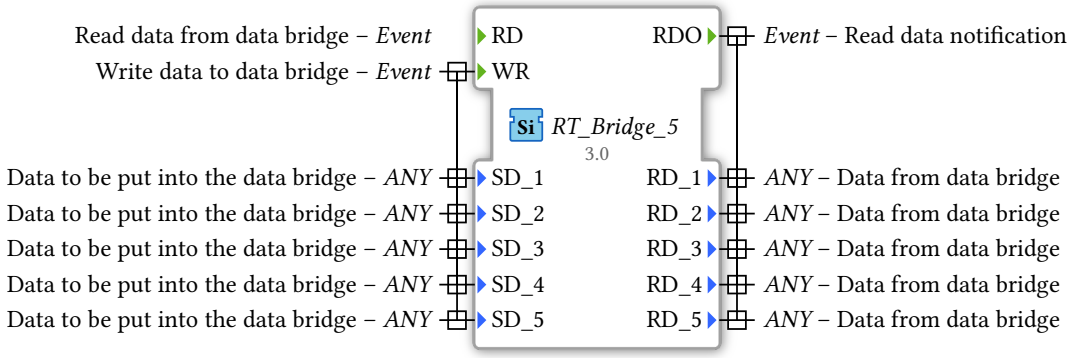
Daten-Ausgänge
- **RD_1** - **RD_4** (ANY): Ausgangsdaten.

Metadaten

Attribut	Wert
Copyright	(c) 2023 Johannes Kepler University Linz
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	eclipse4diac::rtevents

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.6.7 RT_Bridge_5



• – ◦ • – ◦ • – ◦ •

Einleitung

Der RT_Bridge_5 Funktionsblock dient als Entkopplungsbrücke für 5 Datenverbindungen zwischen Echtzeit-Ereignisketten.

Schnittstellenstruktur

Ereignis-Eingänge
- **RD**: Daten lesen.
- **WR**: Daten schreiben.

Ereignis-Ausgänge
- **RDO**: Lese-Bestätigung.

Daten-Eingänge
- **SD_1** - **SD_5** (ANY): Eingangsdaten.

Daten-Ausgänge
- **RD_1** - **RD_5** (ANY): Ausgangsdaten.

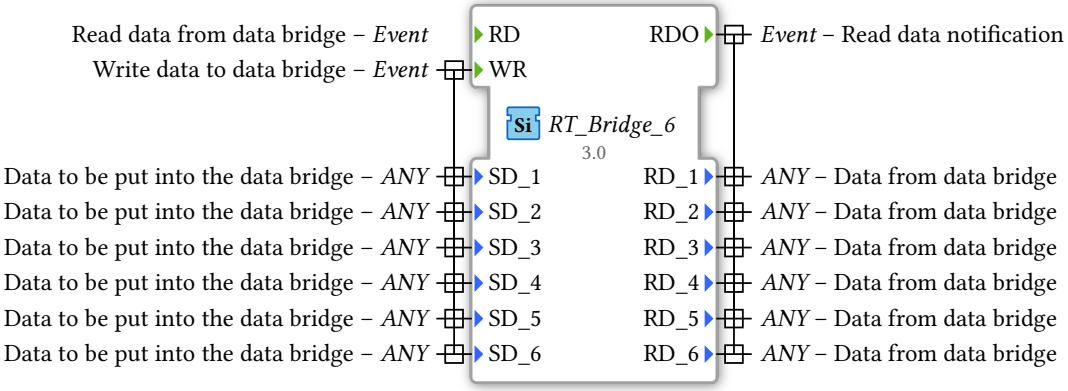
Metadaten

Attribut	Wert
Copyright	(c) 2023 Johannes Kepler University Linz
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	eclipse4diac::rtevents

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.6.8 RT_Bridge_6



• – ◦ • – ◦ • – ◦ •

Einleitung

Der RT_Bridge_6 Funktionsblock dient als Entkopplungsbrücke für 6 Datenverbindungen zwischen Echtzeit-Ereignisketten.

Schnittstellenstruktur

Ereignis-Eingänge

- **RD**: Daten lesen.
- **WR**: Daten schreiben.

Ereignis-Ausgänge

- **RDO**: Lese-Bestätigung.

Daten-Eingänge

- **SD_1** - **SD_6** (ANY): Eingangsdaten.

Daten-Ausgänge

- **RD_1** - **RD_6** (ANY): Ausgangsdaten.

Metadaten

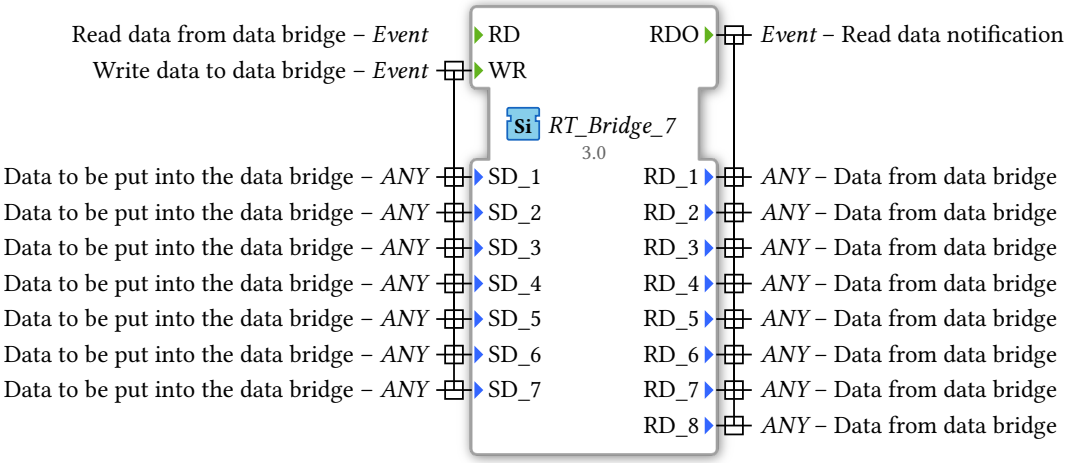
Attribut	Wert
Copyright	(c) 2023 Johannes Kepler University Linz
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)

4diac-Paket	eclipse4diac::rtevents

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.6.9 RT_Bridge_7



• – ◦ • – ◦ • – ◦ •

Einleitung

Der RT_Bridge_7 Funktionsblock dient als Entkopplungsbrücke für 7 Datenverbindungen zwischen Echtzeit-Ereignisketten.

Schnittstellenstruktur

Ereignis-Eingänge

- **RD**: Daten lesen.
- **WR**: Daten schreiben.

Ereignis-Ausgänge

- **RDO**: Lese-Bestätigung.

Daten-Eingänge

- **SD_1** - **SD_7** (ANY): Eingangsdaten.

Daten-Ausgänge

- **RD_1** - **RD_7** (ANY): Ausgangsdaten.

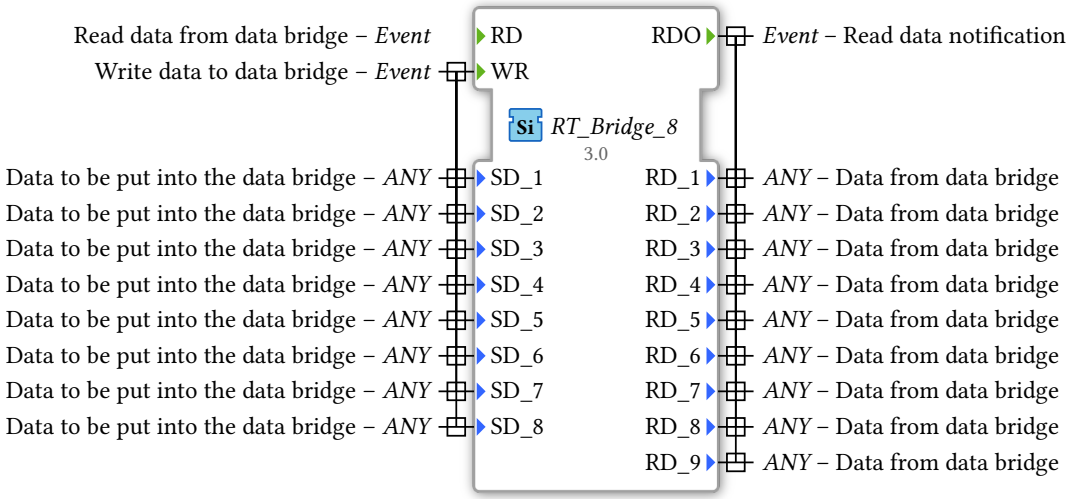
Metadaten

Attribut	Wert
Copyright	(c) 2023 Johannes Kepler University Linz
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	eclipse4diac::rtevents

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.6.10 RT_Bridge_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Der RT_Bridge_8 Funktionsblock dient als Entkopplungsbrücke für 8 Datenverbindungen zwischen Echtzeit-Ereignisketten.

Schnittstellenstruktur

Ereignis-Eingänge

- **RD**: Daten lesen.
- **WR**: Daten schreiben.

Ereignis-Ausgänge

- **RDO**: Lese-Bestätigung.

Daten-Eingänge

- **SD_1** - **SD_8** (ANY): Eingangsdaten.

Daten-Ausgänge

- **RD_1** - **RD_8** (ANY): Ausgangsdaten.

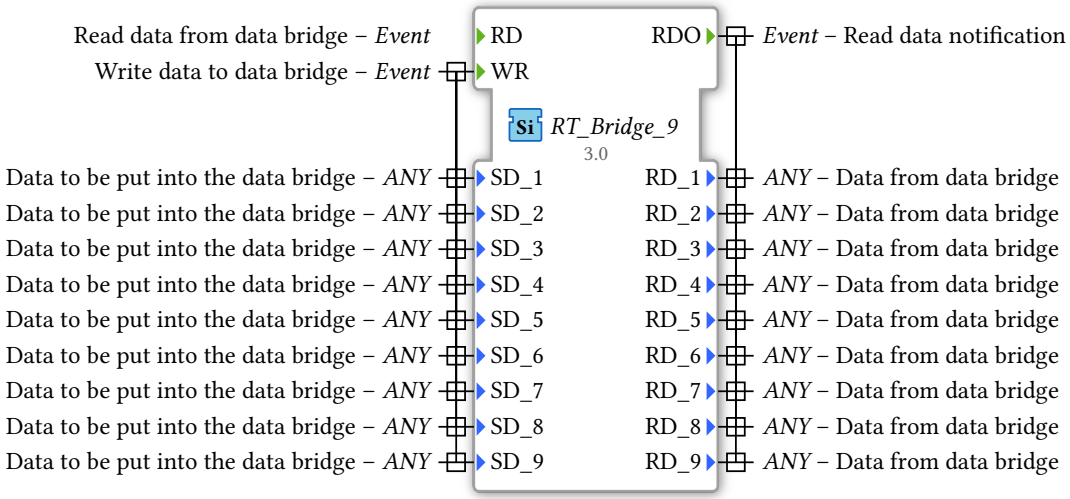
Metadaten

Attribut	Wert
Copyright	(c) 2023 Johannes Kepler University Linz
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	eclipse4diac::rtevents

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.6.11 RT_Bridge_9



• – ◦ • – ◦ • – ◦ •

Einleitung

Der RT_Bridge_9 Funktionsblock dient als Entkopplungsbrücke für 9 Datenverbindungen zwischen Echtzeit-Ereignisketten.

Schnittstellenstruktur

Ereignis-Eingänge

- **RD**: Daten lesen.
- **WR**: Daten schreiben.

Ereignis-Ausgänge

- **RDO**: Lese-Bestätigung.

Daten-Eingänge

- **SD_1** - **SD_9** (ANY): Eingangsdaten.

Daten-Ausgänge

- **RD_1** - **RD_9** (ANY): Ausgangsdaten.

Metadaten

Attribut	Wert
Copyright	(c) 2023 Johannes Kepler University Linz
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	eclipse4diac::rtevents

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

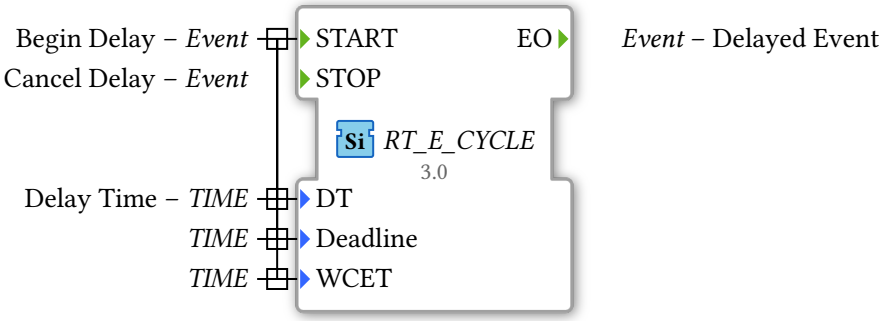
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.6.12 RT_E_CYCLE

• – ◦ • – ◦ • – ◦ •

Einleitung

Der RT_E_CYCLE Funktionsblock implementiert eine periodische Ereigniserzeugung (Zyklus) mit Echtzeiteigenschaften. Er ermöglicht das Definieren von Deadlines und Ausführungszeiten (WCET).



Schnittstellenstruktur

Ereignis-Eingänge

- **START**: Startet den Zyklus.
- **STOP**: Stoppt den Zyklus.

Ereignis-Ausgänge

- **EO**: Das zyklisch erzeugte Ereignis.

Daten-Eingänge

- **DT** (TIME): Zykluszeit (Delay Time).
- **Deadline** (TIME): Deadline für die Ereignisverarbeitung.
- **WCET** (TIME): Worst Case Execution Time.

Daten-Ausgänge

- **QO** (BOOL): Statusanzeige.

Funktionsweise

Nach dem `START`-Ereignis wird das `EO`-Ereignis periodisch mit der Zeit `DT` ausgelöst. Die Parameter `Deadline` und `WCET` dienen der Echtzeitplanung und -überwachung.

Metadaten

Attribut	Wert
Copyright	(c) 2008 ACIN
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	eclipse4diac::rtevents

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

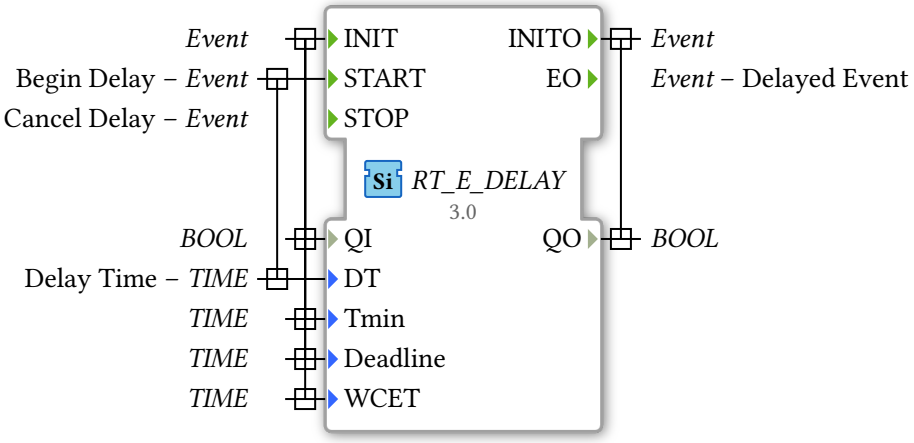
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.6.13 RT_E_DELAY

• – ◦ • – ◦ • – ◦ •

Einleitung

Der RT_E_DELAY Funktionsblock verzögert ein Ereignis um eine definierte Zeit (DT) unter Berücksichtigung von Echtzeitanforderungen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierung.
- **START**: Startet die Verzögerung.
- **STOP**: Bricht die Verzögerung ab.

Ereignis-Ausgänge

- **INITO**: Initialisierung bestätigt.
- **EO**: Das verzögerte Ereignis.

Daten-Eingänge

- **QI** (BOOL): Qualifizierer.
- **DT** (TIME): Verzögerungszeit.
- **Tmin** (TIME): Minimale Zeit.
- **Deadline** (TIME): Deadline.
- **WCET** (TIME): Worst Case Execution Time.

Daten-Ausgänge

- **QO** (BOOL): Status.

Metadaten

Attribut	Wert
Copyright	(c) 2008, 2014 ACIN, Profactor GmbH
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	eclipse4diac::rtevents

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

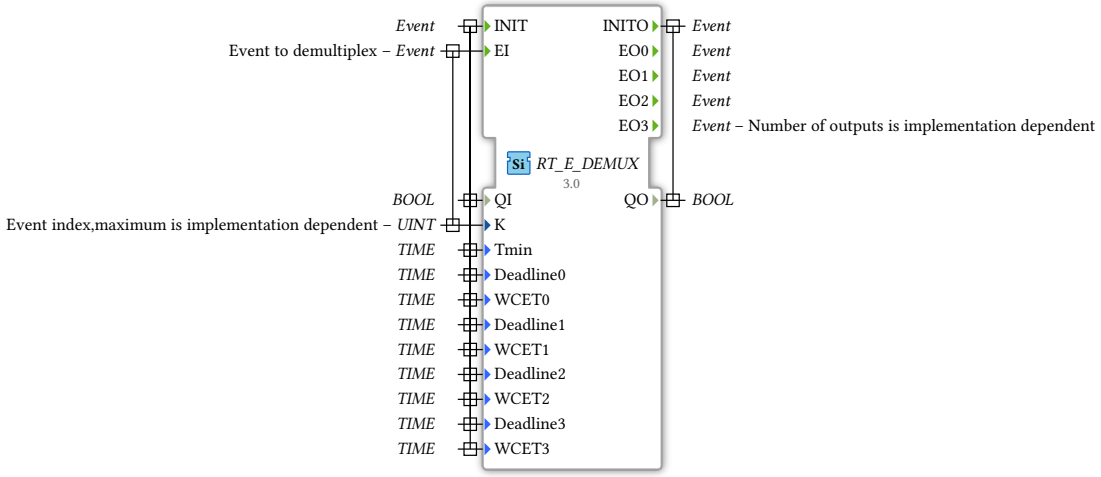
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.6.14 RT_E_DEMUX

• – ◦ • – ◦ • – ◦ •

Einleitung

Echtzeit-Variante des E_DEMUX Bausteins. Leitet ein Eingangsereignis an einen von mehreren Ausgängen weiter.



Schnittstelle

Ereigniseingänge

Ereignis	Zugehörige Variablen	Beschreibung
INIT	QI, Tmin, Deadline0..Deadline3, WCET0..WCET3	Initialisierung
EI	K	Zu demultiplexendes Ereignis

Ereignisausgänge

Ereignis	Zugehörige Variablen	Beschreibung
INITO	QO	Bestätigung der Initialisierung
EO0..EO3	-	Ausgangsereignisse (die Anzahl der Ausgänge ist implementierungsabhängig)

Eingangsvariablen

Name	Typ	Beschreibung
QI	BOOL	Eingang der Initialisierung
K	UINT	Ereignisindex, das Maximum ist implementierungsabhängig
Tmin	TIME	Zeitparameter der Echtzeit-Variante (im Quelltext nicht weiter kommentiert)
Deadline0..Deadline3	TIME	Zeitparameter der Echtzeit-Variante (im Quelltext nicht weiter kommentiert), je Ausgang EO0..EO3
WCET0..WCET3	TIME	Zeitparameter der Echtzeit-Variante (im Quelltext nicht weiter kommentiert), je Ausgang EO0..EO3

Ausgangsvariablen

Name	Typ	Beschreibung
QO	BOOL	Ausgang der Initialisierung (zu INITO)

Metadaten

Attribut	Wert

Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	eclipse4diac::rtevents

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

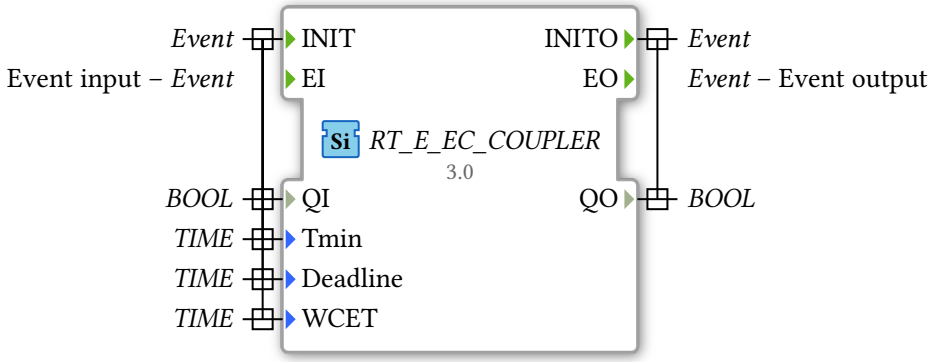
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.6.15 RT_E_EC_COUPLER

• – ◦ • – ◦ • – ◦ •

Einleitung

Koppelt Ereignisketten mit Echtzeitparametern.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierung, trägt `QI`, `Tmin`, `Deadline` und `WCET`.
- **EI**: Ereigniseingang (Beginn der Kopplung); trägt keine Daten.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung, trägt `QO`.
- **EO**: Ereignisausgang (Fortsetzung der Ereigniskette); trägt keine Daten.

Daten-Eingänge

- **QI** (`BOOL`): Qualifier für die Initialisierung.
- **Tmin** (`TIME`): Echtzeitparameter (minimaler Abstand).
- **Deadline** (`TIME`): Echtzeitparameter (Deadline).
- **WCET** (`TIME`): Echtzeitparameter (maximale Ausführungszeit).

Daten-Ausgänge

- **QO** (`BOOL`): Ausgangsqualifier der Initialisierung.

Metadaten

Attribut	Wert
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	eclipse4diac::rtevents

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

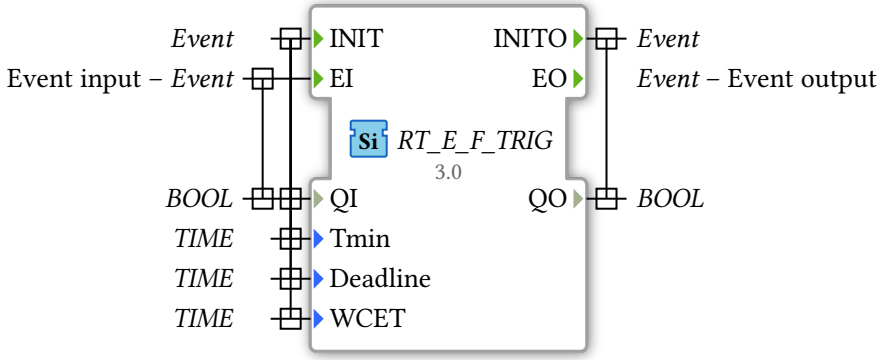
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.6.16 RT_E_F_TRIG

• – ◦ • – ◦ • – ◦ •

Einleitung

Echtzeit-Variante des E_F_TRIG Bausteins (Fallende Flanke).



Schnittstelle

Port	Typ	Beschreibung
INIT	Event	Initialisierung (mit QI, Tmin, Deadline, WCET)
EI	Event	Ereigniseingang (mit QI)
INITO	Event	Initialisierung abgeschlossen (mit QO)
EO	Event	Ereignisausgang
QI	BOOL	Eingangssignal
Tmin	TIME	Echtzeit-Parameter (mit INIT)
Deadline	TIME	Echtzeit-Parameter (mit INIT)
WCET	TIME	Echtzeit-Parameter (mit INIT)
QO	BOOL	Ausgangssignal (mit INITO)

Metadaten

Attribut	Wert
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	eclipse4diac::rtevents

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

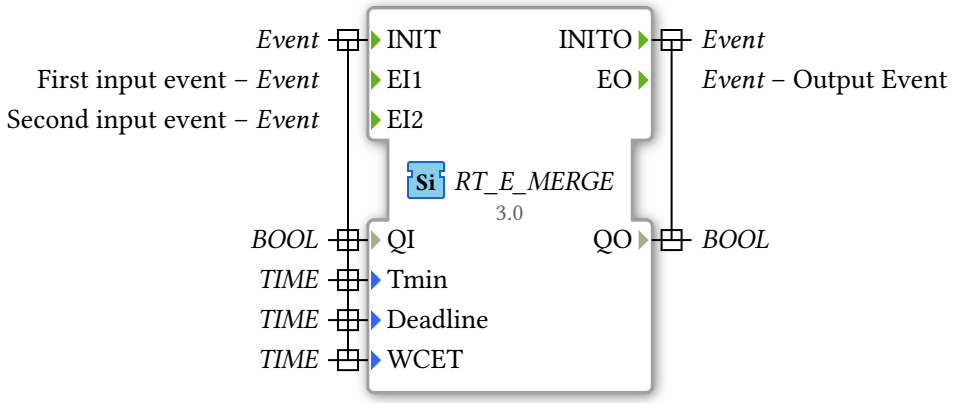
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.6.17 RT_E_MERGE

• – ◦ • – ◦ • – ◦ •

Einleitung

Echtzeit-Variante des E_MERGE Bausteins. Führt mehrere Ereignisse zusammen.



Schnittstelle

Ereigniseingänge

Ereignis	Zugehörige Variablen	Beschreibung
INIT	QI, Tmin, Deadline, WCET	Initialisierung
EI1	-	Erstes Eingangsereignis
EI2	-	Zweites Eingangsereignis

Ereignisausgänge

Ereignis	Zugehörige Variablen	Beschreibung
INITO	QO	Bestätigung der Initialisierung
EO	-	Ausgangsereignis

Eingangsvariablen

Name	Typ	Beschreibung
QI	BOOL	Eingang der Initialisierung
Tmin	TIME	Zeitparameter der Echtzeit-Variante (im Quelltext nicht weiter kommentiert)
Deadline	TIME	Zeitparameter der Echtzeit-Variante (im Quelltext nicht weiter kommentiert)
WCET	TIME	Zeitparameter der Echtzeit-Variante (im Quelltext nicht weiter kommentiert)

Ausgangsvariablen

Name	Typ	Beschreibung
QO	BOOL	Ausgang der Initialisierung (zu INITO)

Metadaten

Attribut	Wert
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	eclipse4diac::rtevents

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

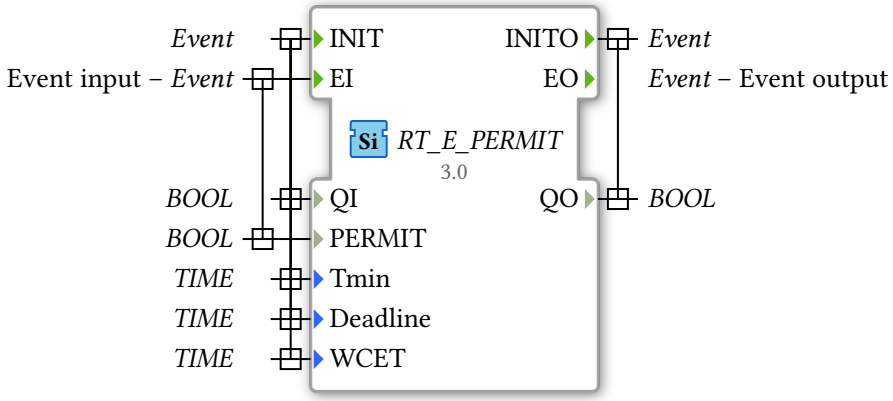
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.6.18 RT_E_PERMIT

• – ○ • – ○ • – ○ •

Einleitung

Echtzeit-Variante des E_PERMIT Bausteins. Lässt ein Ereignis nur durch, wenn die Bedingung (PERMIT) wahr ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierung, trägt `QI`, `Tmin`, `Deadline` und `WCET`.
- **EI**: Ereigniseingang, trägt `PERMIT`.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung, trägt `QO`.
- **EO**: Ereignisausgang; trägt keine Daten.

Daten-Eingänge

- **QI** (`BOOL`): Qualifier für die Initialisierung.
- **PERMIT** (`BOOL`): Freigabebedingung für die Weitergabe des Ereignisses.
- **Tmin** (`TIME`): Echtzeitparameter (minimaler Abstand).
- **Deadline** (`TIME`): Echtzeitparameter (Deadline).
- **WCET** (`TIME`): Echtzeitparameter (maximale Ausführungszeit).

Daten-Ausgänge

- **QO** (`BOOL`): Ausgangsqualifier der Initialisierung.

Metadaten

Attribut	Wert
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	eclipse4diac::rtevents

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

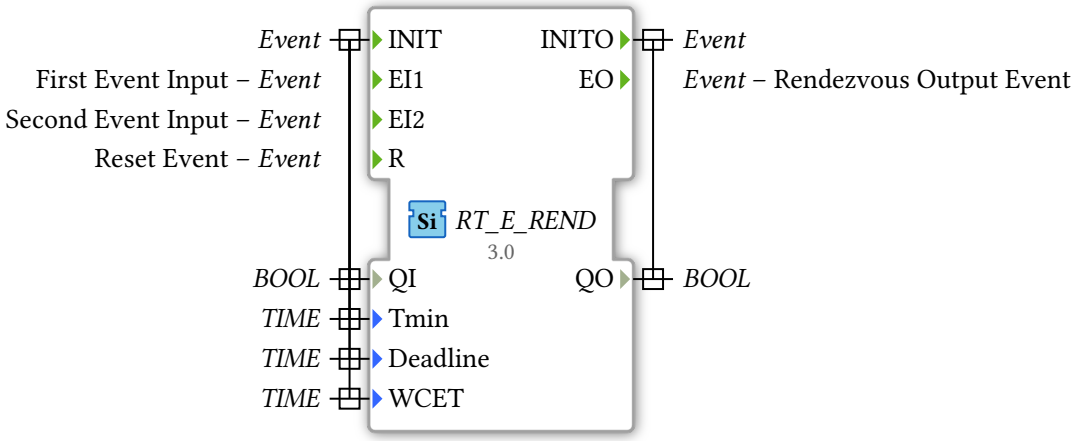
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.6.19 RT_E_REND

• – ◦ • – ◦ • – ◦ •

Einleitung

Echtzeit-Variante des E_REND Bausteins (Rendezvous). Wartet auf zwei Ereignisse.



Schnittstelle

Port	Typ	Beschreibung
INIT	Event	Initialisierung (mit QI, Tmin, Deadline, WCET)
EI1	Event	Erster Ereigniseingang
EI2	Event	Zweiter Ereigniseingang
R	Event	Reset-Ereignis
INITO	Event	Initialisierung abgeschlossen (mit QO)
EO	Event	Rendezvous-Ausgangsereignis
QI	BOOL	Eingang (mit INIT)
Tmin	TIME	Echtzeit-Parameter (mit INIT)
Deadline	TIME	Echtzeit-Parameter (mit INIT)
WCET	TIME	Echtzeit-Parameter (mit INIT)
QO	BOOL	Ausgang (mit INITO)

Metadaten

Attribut	Wert
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	eclipse4diac::rtevents

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

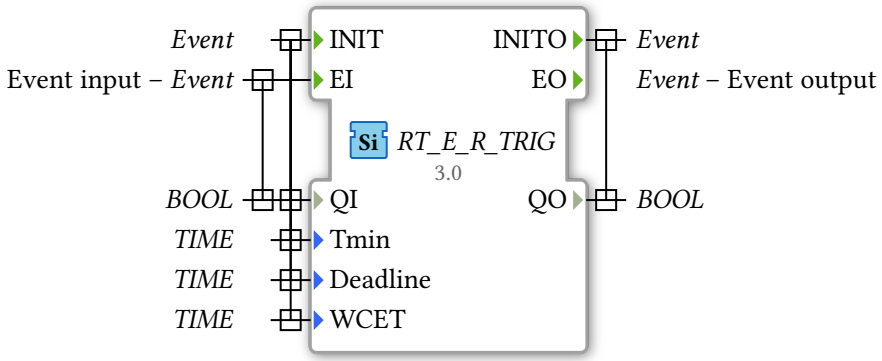
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.6.20 RT_E_R_TRIG

• – ◦ • – ◦ • – ◦ •

Einleitung

Echtzeit-Variante des E_R_TRIG Bausteins (Steigende Flanke).



Schnittstelle

Ereigniseingänge

Ereignis	Zugehörige Variablen	Beschreibung
INIT	QI, Tmin, Deadline, WCET	Initialisierung
EI	QI	Ereigniseingang

Ereignisausgänge

Ereignis	Zugehörige Variablen	Beschreibung
INITO	QO	Bestätigung der Initialisierung
EO	-	Ereignisausgang

Eingangsvariablen

Name	Typ	Beschreibung
QI	BOOL	Eingang (bei INIT: Initialisierung; bei EI: zu bewertender Wert)
Tmin	TIME	Zeitparameter der Echtzeit-Variante (im Quelltext nicht weiter kommentiert)
Deadline	TIME	Zeitparameter der Echtzeit-Variante (im Quelltext nicht weiter kommentiert)
WCET	TIME	Zeitparameter der Echtzeit-Variante (im Quelltext nicht weiter kommentiert)

Ausgangsvariablen

Name	Typ	Beschreibung
QO	BOOL	Ausgang der Initialisierung (zu INITO)

Metadaten

Attribut	Wert
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	eclipse4diac::rtevents

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

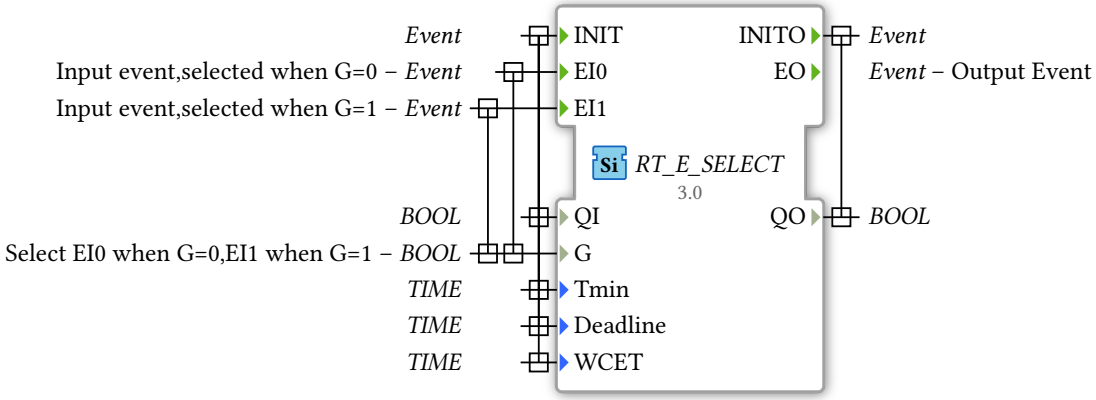
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.6.21 RT_E_SELECT

• – ◦ • – ◦ • – ◦ •

Einleitung

Echtzeit-Variante des E_SELECT Bausteins. Wählt eines von zwei Eingangsereignissen basierend auf einem Selektor aus.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisierung, trägt `QI`, `Tmin`, `Deadline` und `WCET`.
- **EI0**: Eingangsereignis, wird bei `G` = 0 ausgewählt; trägt `G`.
- **EI1**: Eingangsereignis, wird bei `G` = 1 ausgewählt; trägt `G`.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung, trägt `QO`.
- **EO**: Ausgangsereignis; trägt keine Daten.

Daten-Eingänge

- **QI** (`BOOL`): Qualifier für die Initialisierung.
- **G** (`BOOL`): Selektor: `EI0` bei `G` = 0, `EI1` bei `G` = 1.
- **Tmin** (`TIME`): Echtzeitparameter (minimaler Abstand).
- **Deadline** (`TIME`): Echtzeitparameter (Deadline).
- **WCET** (`TIME`): Echtzeitparameter (maximale Ausführungszeit).

Daten-Ausgänge

- **QO** (`BOOL`): Ausgangsqualifier der Initialisierung.

Metadaten

Attribut	Wert
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	eclipse4diac::rtevents

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

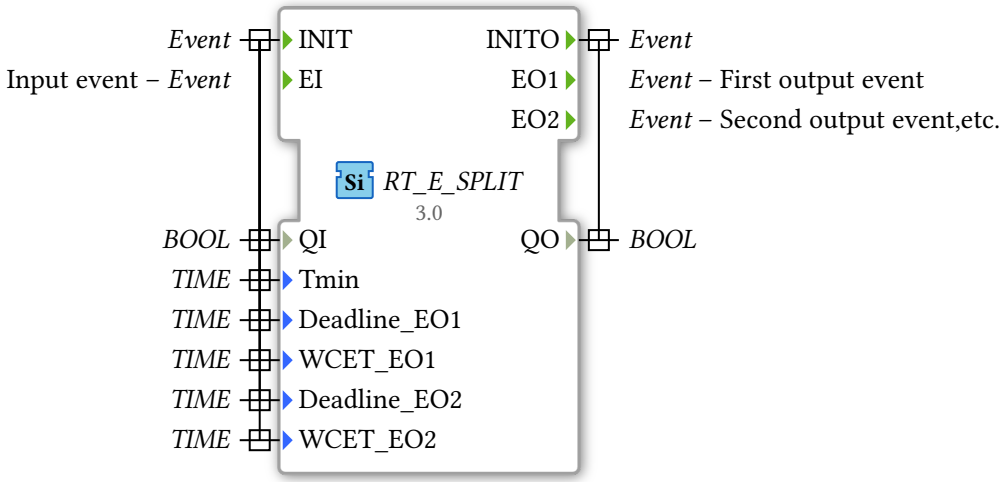
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.6.22 RT_E_SPLIT

• – ◦ • – ◦ • – ◦ •

Einleitung

Echtzeit-Variante des E_SPLIT Bausteins. Teilt ein Ereignis in mehrere auf.



Schnittstelle

Port	Typ	Beschreibung
INIT	Event	Initialisierung (mit QI, Tmin, Deadline_EO1, WCET_EO1, Deadline_EO2, WCET_EO2)
EI	Event	Eingangsereignis
INITO	Event	Initialisierung abgeschlossen (mit QO)
EO1	Event	Erstes Ausgangsereignis
EO2	Event	Zweites Ausgangsereignis
QI	BOOL	Eingang (mit INIT)
Tmin	TIME	Echtzeit-Parameter (mit INIT)
Deadline_EO1	TIME	Echtzeit-Parameter für EO1 (mit INIT)
WCET_EO1	TIME	Echtzeit-Parameter für EO1 (mit INIT)
Deadline_EO2	TIME	Echtzeit-Parameter für EO2 (mit INIT)
WCET_EO2	TIME	Echtzeit-Parameter für EO2 (mit INIT)
QO	BOOL	Ausgang (mit INITO)

Metadaten

Attribut	Wert
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	eclipse4diac::rtevents

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

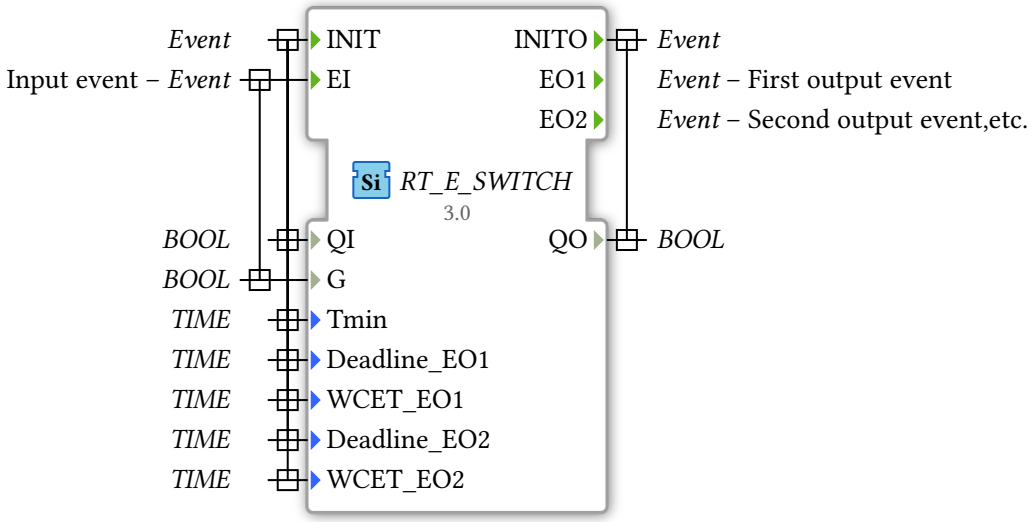
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.6.23 RT_E_SWITCH

• – ◦ • – ◦ • – ◦ •

Einleitung

Echtzeit-Variante des E_SWITCH Bausteins. Leitet das Eingangsereignis EI abhängig von G an einen der beiden Ausgänge EO1/EO2 weiter.



Schnittstelle

Ereigniseingänge

Ereignis	Zugehörige Variablen	Beschreibung
INIT	QI, Tmin, Deadline_EO1, WCET_EO1, Deadline_EO2, WCET_EO2	Initialisierung
EI	G	Eingangsereignis

Ereignisausgänge

Ereignis	Zugehörige Variablen	Beschreibung
INITO	QO	Bestätigung der Initialisierung
EO1	-	Erstes Ausgangsereignis
EO2	-	Zweites Ausgangsereignis

Eingangsvariablen

Name	Typ	Beschreibung
QI	BOOL	Eingang der Initialisierung
G	BOOL	Schaltbedingung, mit EI verknüpft
Tmin	TIME	Zeitparameter der Echtzeit-Variante (im Quelltext nicht weiter kommentiert)
Deadline_EO1, Deadline_EO2	TIME	Zeitparameter der Echtzeit-Variante (im Quelltext nicht weiter kommentiert), je Ausgang
WCET_EO1, WCET_EO2	TIME	Zeitparameter der Echtzeit-Variante (im Quelltext nicht weiter kommentiert), je Ausgang

Ausgangsvariablen

Name	Typ	Beschreibung
QO	BOOL	Ausgang der Initialisierung (zu INITO)

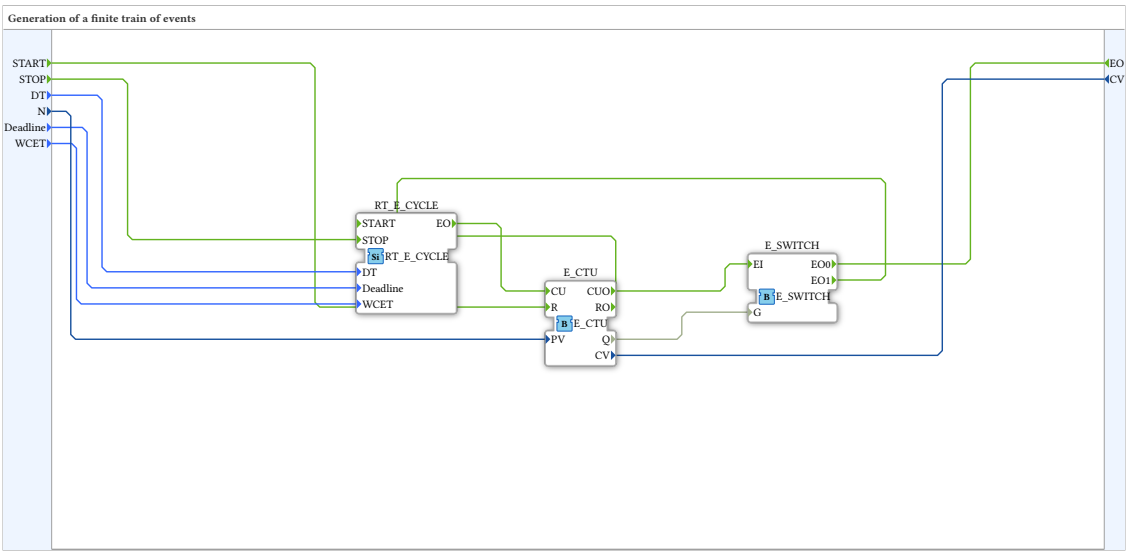
Metadaten

Attribut	Wert
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	eclipse4diac::rtevents

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

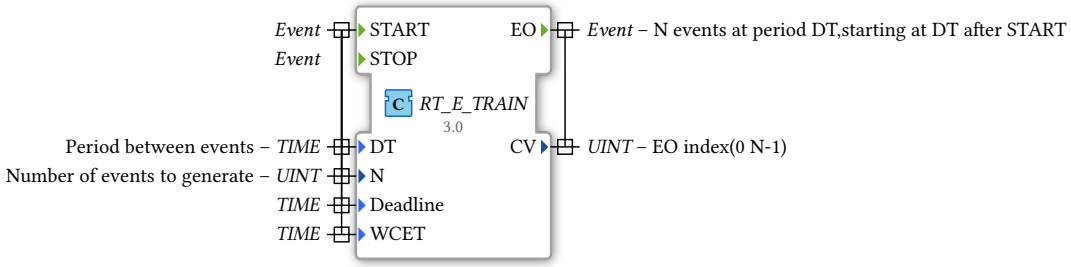
2.4.6.24 RT_E_TRAIN



• – ○ • – ○ • – ○ •

Einleitung

Echtzeit-Variante des E_TRAIN Bausteins. Erzeugt eine Folge von Ereignissen.



Schnittstellenstruktur

Ereignis-Eingänge

- **START**: Startet die Ereignisfolge, trägt `DT`, `N`, `Deadline` und `WCET`.
- **STOP**: Stoppt die Ereignisfolge; trägt keine Daten.

Ereignis-Ausgänge

- **EO**: `N` Ereignisse im Abstand `DT`, beginnend `DT` nach `START`; trägt `CV`.

Daten-Eingänge

- **DT** (`TIME`): Zeitabstand zwischen den Ereignissen.
- **N** (`UINT`): Anzahl der zu erzeugenden Ereignisse.
- **Deadline** (`TIME`): Echtzeitparameter (Deadline).
- **WCET** (`TIME`): Echtzeitparameter (maximale Ausführungszeit).

Daten-Ausgänge

- **CV** (`UINT`): Index des aktuellen `EO` (0 bis N-1).

Metadaten

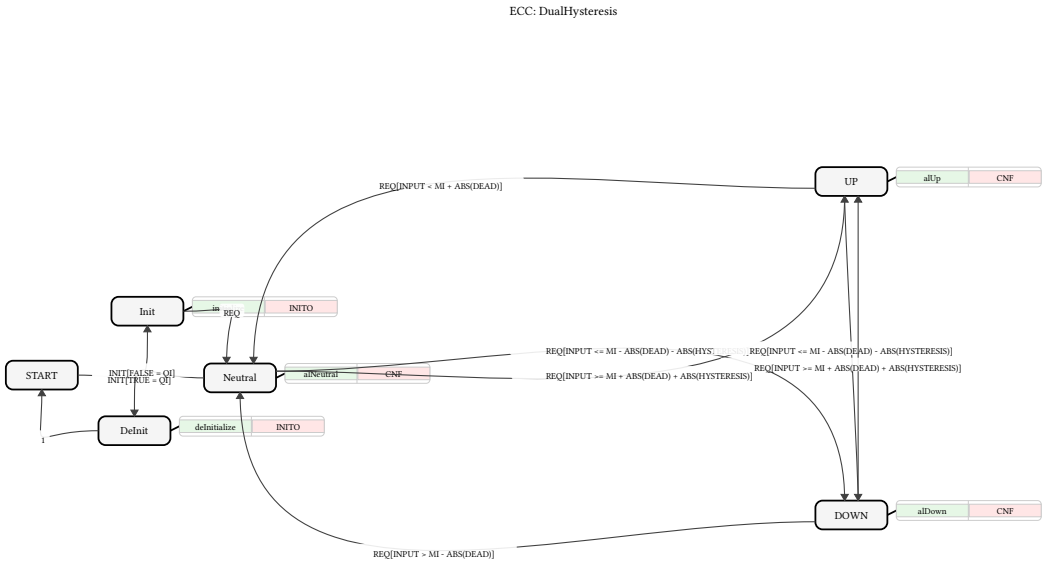
Attribut	Wert
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	eclipse4diac::rtevents

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.7 signalprocessing

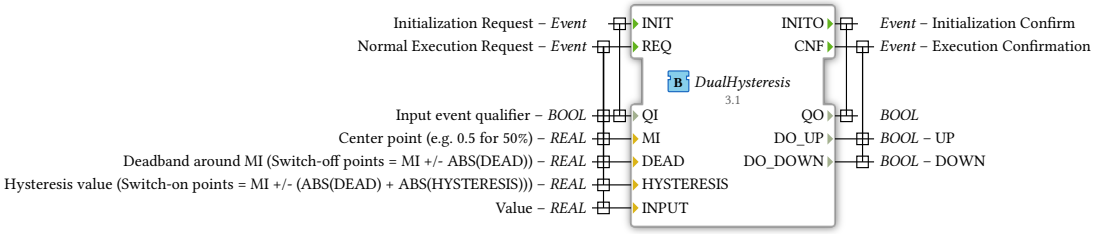
2.4.7.1 DualHysteresis



• – ○ • – ○ • – ○ •

Einleitung

Der DualHysteresis-Funktionsblock ermöglicht eine bidirektionale Umwandlung von analogen zu digitalen Signalen unter Verwendung einer Hysterese. Er ist besonders nützlich in Steuerungsanwendungen, wo eine zuverlässige Schaltpunktumsetzung mit Totband und Hysterese benötigt wird.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisierungsanforderung (mit `QI`)
- `REQ`: Normale Ausführungsanforderung (mit `QI`, `MI`, `DEAD`, `INPUT`, `HYSTERESIS`)

Ereignis-Ausgänge

- `INITO`: Initialisierungsbestätigung (mit `QO`)
- `CNF`: Ausführungsbestätigung (mit `DO_UP`, `DO_DOWN`)

Daten-Eingänge

- `QI` (BOOL): Eingangsereignis-Qualifizierer
- `MI` (REAL, Initialwert: 0.5): Mittelwert-Einstellung (z.B. 0.5 für 50%)
- `DEAD` (REAL, Initialwert: 0.1): Totband um MI (z.B. 0.1 für 10%)
- `HYSTERESIS` (REAL, Initialwert: 0.1): Hysterese (z.B. 0.1 für 10%)
- `INPUT` (REAL): Eingangswert

Daten-Ausgänge

- `QO` (BOOL): Ausgangsereignis-Qualifizierer
- `DO_UP` (BOOL): Signal für Aufwärtsrichtung
- `DO_DOWN` (BOOL): Signal für Abwärtsrichtung

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock durchläuft verschiedene Zustände:

1. **Initialisierung**: Setzt die Ausgänge auf sichere Werte
2. **Neutral**: Wartet auf gültige Eingangswerte
3. **UP/DOWN**: Aktiviert entsprechende Ausgänge bei Überschreiten der Schwellwerte
4. **Deinitialisierung**: Kehrt in den sicheren Zustand zurück

Die Hysterese verhindert ein zu häufiges Schalten bei kleinen Signaländerungen um den Schwellwert.

Technische Besonderheiten

- Verwendet ein Totband (DEAD) um den Mittelwert (MI)
- Implementiert eine symmetrische Hysterese (HYSTERESIS)
- Alle Parameter sind konfigurierbar
- Sicherer Zustand bei Deaktivierung (QI=FALSE)

Zustandsübersicht

1. **START → Init** (bei INIT mit QI=TRUE)
2. **Init → Neutral** (bei REQ)
3. **Neutral → UP** (wenn INPUT ≥ MI + DEAD + HYSTERESIS)
4. **Neutral → DOWN** (wenn INPUT ≤ MI - DEAD - HYSTERESIS)
5. **UP/DOWN → Neutral** (bei Rückkehr in den Totbandbereich)
6. **Neutral → DeInit** (bei INIT mit QI=FALSE)
7. **DeInit → START** (automatisch)

Anwendungsszenarien

- Positionsregelungen mit Totband
- Temperaturüberwachung mit Hysterese
- Füllstandserkennung mit unterschiedlichen Schaltpunkten für Füllen/Entleeren
- Allgemeine Signalaufbereitung mit Rauschunterdrückung

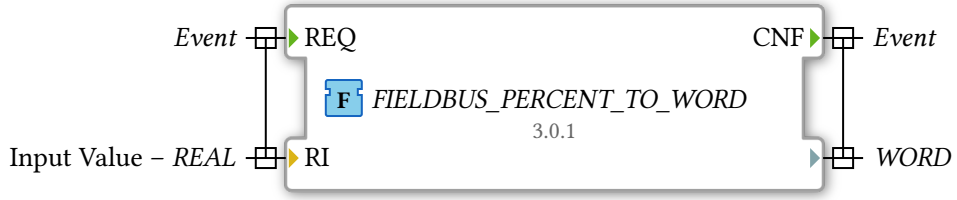
⚖ Vergleich mit ähnlichen Bausteinen

- Einfache Hysterese-Blöcke haben typischerweise nur einen Schaltausgang
- Standard-Komparatoren bieten keine Totband- oder Hysterese-Funktionalität
- Dieser Block kombiniert beide Richtungen in einem Baustein mit konfigurierbaren Parametern

Fazit

Der DualHysteresis-Block bietet eine flexible Lösung für Anwendungen, die eine zuverlässige Signalumsetzung mit Totband und Hysterese in beiden Richtungen benötigen. Die klare Zustandsmaschine und konfigurierbaren Parameter machen ihn besonders vielseitig einsetzbar.

2.4.7.2 FIELDBUS_PERCENT_TO_WORD



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `FIELDBUS_PERCENT_TO_WORD` konvertiert einen REAL-Wert im Bereich von 0.0 bis 100.0 in einen WORD-Wert im Bereich von 0 bis FAFF (hexadezimal). Dieser Baustein ist besonders für die Signalverarbeitung in Feldbussystemen nach den Standards SAE J1939 und ISO 11783 geeignet.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Auslöseereignis für die Konvertierung. Wird mit dem Eingangswert `RI` verknüpft.

Ereignis-Ausgänge

- `CNF`: Bestätigungsereignis, das nach erfolgreicher Konvertierung ausgelöst wird.

Daten-Eingänge

- `RI` (REAL): Eingangswert im Bereich 0.0 bis 100.0, der konvertiert werden soll.

Daten-Ausgänge

- (WORD): Der konvertierte Ausgangswert im Bereich 0 bis FAFF (hexadezimal).

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock führt folgende Schritte durch:

1. Multiplikation des Eingangs-REAL-Werts (`RI`) mit dem konvertierten maximalen gültigen Signalwert (`FIELDBUS_SIGNAL::VALID_SIGNAL_W`).
2. Konvertierung des Ergebnisses in einen UDINT-Wert.
3. Umwandlung des UDINT-Werts in den finalen WORD-Ausgangswert.

Die Konvertierung erfolgt nach der Formel:

```
FIELDBUS_PERCENT_TO_WORD = UDINT_TO_WORD(REAL_TO_UDINT(RI *
UDINT_TO_REAL(WORD_TO_UDINT(FIELDBUS_SIGNAL::VALID_SIGNAL_W))))
```

Technische Besonderheiten

- Verwendet spezielle Typkonvertierungsfunktionen (REAL_TO_UDINT, UDINT_TO_WORD etc.)
- Arbeitet mit dem konstanten Wert `FIELDBUS_SIGNAL::VALID_SIGNAL_W` als maximale Obergrenze
- Eingebettet im Package `eclipse4diac::signalprocessing`

Zustandsübersicht

1. Wartezustand: Auf `REQ`-Ereignis warten
2. Verarbeitungszustand: Bei `REQ`-Eingang wird die Konvertierung durchgeführt
3. Ausgabezustand: Nach erfolgreicher Konvertierung wird `CNF` ausgelöst

Anwendungsszenarien

- Signalverarbeitung in landwirtschaftlichen Maschinen
- Fahrzeugsteuerungssysteme
- Industrielle Automatisierung mit Feldbuskommunikation
- Prozessdatenkonvertierung zwischen verschiedenen Systemen

⚖️ Vergleich mit ähnlichen Bausteinen

- Spezialisiert auf die spezifischen Anforderungen von SAE J1939 und ISO 11783
- Bietet genauere Kontrolle über den Wertebereich als generische Konvertierungsbausteine
- Optimiert für die Verwendung in der Signalverarbeitung

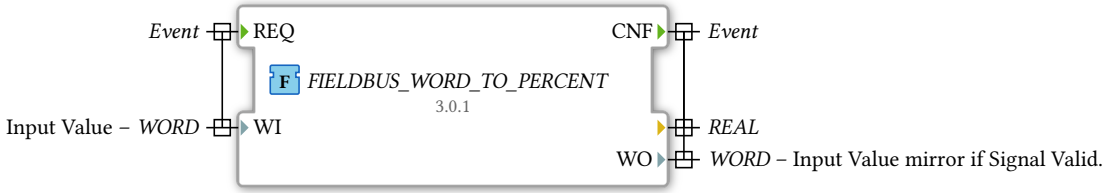
Fazit

Der `FIELDBUS_PERCENT_TO_WORD`-Funktionsblock ist ein spezialisiertes Werkzeug für die präzise Konvertierung von Prozentwerten in feldbusspezifische WORD-Werte. Seine klare Schnittstelle und effiziente Implementierung machen ihn besonders geeignet für Anwendungen in der mobilen Arbeitsmaschinensteuerung und industriellen Automatisierung.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.7.3 FIELDBUS_WORD_TO_PERCENT



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `FIELDBUS_WORD_TO_PERCENT` konvertiert einen REAL-Wert im Bereich von 0.0 bis 100.0 in einen WORD-Wert im Bereich von 0 bis FAFF (hexadezimal). Dies ist besonders nützlich für die Signalverarbeitung in Feldbussystemen, die nach den Standards SAE J1939 und ISO 11783 arbeiten.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Konvertierung. Wird mit dem Daten-Eingang `WI` verknüpft.

Ereignis-Ausgänge

- **CNF**: Bestätigt die abgeschlossene Konvertierung. Liefert den konvertierten Wert über den Daten-Ausgang `WO`.

Daten-Eingänge

- **WI** (WORD): Der Eingangswert, der konvertiert werden soll.

Daten-Ausgänge

- **WO** (WORD): Eine Spiegelung des Eingangswerts `WI`, falls das Signal gültig ist.
- **(REAL)**: Der konvertierte REAL-Wert im Bereich von 0.0 bis 100.0.

Funktionsweise

Der Funktionsblock überprüft zunächst, ob der Eingangswert `WI` kleiner oder gleich dem definierten `VALID_SIGNAL_W` ist. Falls ja, wird der Wert gespiegelt (`WO := WI`) und anschließend in einen REAL-Wert zwischen 0.0 und 100.0 umgerechnet. Die Umrechnung erfolgt durch die Division des konvertierten WORD-Werts durch den maximalen gültigen Signalwert (`VALID_SIGNAL_W`).

Technische Besonderheiten

- Der Block unterstützt die Standards SAE J1939 und ISO 11783.
- Die Konvertierung erfolgt nur, wenn das Eingangssignal gültig ist (`WI <= VALID_SIGNAL_W`).
- Der Ausgang `WO` dient zur Validierung des Eingangssignals.

Zustandsübersicht

1. **Initialisierung**: Der Block wartet auf ein `REQ`-Ereignis.
2. **Verarbeitung**: Bei `REQ` wird `WI` überprüft und ggf. konvertiert.
3. **Ausgabe**: Bei erfolgreicher Konvertierung wird `CNF` ausgelöst und die Ergebnisse ausgegeben.

Anwendungsszenarien

- Signalverarbeitung in landwirtschaftlichen Maschinen (ISO 11783).
- Steuerungssysteme in Nutzfahrzeugen (SAE J1939).
- Allgemeine Feldbuskommunikation, wo WORD- zu REAL-Konvertierungen benötigt werden.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu generischen Konvertierungsblöcken ist `FIELDBUS_WORD_TO_PERCENT` speziell auf die Anforderungen von SAE J1939 und ISO 11783 optimiert.
- Andere Blöcke bieten möglicherweise keine direkte Spiegelung des Eingangssignals zur Validierung.

Fazit

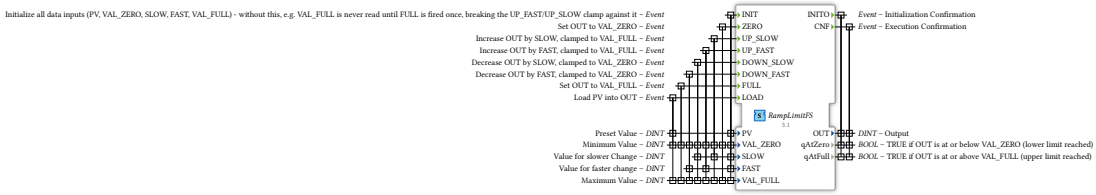
Der `FIELDBUS_WORD_TO_PERCENT`-Block ist ein spezialisierter Konverter für die Signalverarbeitung in Feldbussystemen. Durch seine einfache Schnittstelle und robuste Funktionsweise eignet er sich ideal für Anwendungen, die eine zuverlässige Umwandlung von WORD- zu REAL-Werten erfordern.

2.4.7.4 RampLimitFS

• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `RampLimitFS` dient zur schrittweisen Erhöhung oder Verringerung eines Ausgabewertes mit verschiedenen Geschwindigkeiten (schnell/langsam), ähnlich wie bei einer Tempomat-Funktion in Fahrzeugen. Zusätzlich bietet er die Möglichkeit, Minimal- und Maximalwerte zu setzen sowie einen vordefinierten Wert zu laden.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisiert alle Dateneingänge (`PV`, `VAL_ZERO`, `SLOW`, `FAST`, `VAL_FULL`) gemeinsam und setzt `OUT` auf `VAL_ZERO`. **Muss vor jedem anderen Ereignis einmal ausgelöst werden** — ohne `INIT` bleibt z. B. `VAL_FULL` auf seinem Vorgabewert 0, bis `FULL` selbst einmal ausgelöst wurde, wodurch die Begrenzung bei `UP_SLOW`/`UP_FAST` fehlschlägt (siehe Technische Besonderheiten).
- `ZERO`: Setzt den Ausgang `OUT` auf den Wert von `VAL_ZERO`.

- `UP_SLOW`: Erhöht `OUT` um den Wert von `SLOW`.
- `UP_FAST`: Erhöht `OUT` um den Wert von `FAST`.
- `DOWN_SLOW`: Verringert `OUT` um den Wert von `SLOW`.
- `DOWN_FAST`: Verringert `OUT` um den Wert von `FAST`.
- `FULL`: Setzt `OUT` auf den Wert von `VAL_FULL`.
- `LOAD`: Lädt den Wert von `PV` in `OUT`.

Ereignis-Ausgänge

- `INITO`: Bestätigt die Initialisierung (Antwort auf `INIT`) und gibt den aktuellen Wert von `OUT` aus.
- `CNF`: Bestätigt die Ausführung und gibt den aktuellen Wert von `OUT` aus.

Daten-Eingänge

- `PV` (DINT): Voreingestellter Wert, der mit `LOAD` geladen wird.
- `VAL_ZERO` (DINT): Minimaler Wert für `OUT`.
- `SLOW` (DINT): Schrittweite für langsame Änderungen.
- `FAST` (DINT): Schrittweite für schnelle Änderungen.
- `VAL_FULL` (DINT): Maximaler Wert für `OUT`.

Daten-Ausgänge

- `OUT` (DINT): Aktueller Ausgabewert.
- `qAtZero` (BOOL): `TRUE`, wenn `OUT` auf oder unter `VAL_ZERO` liegt (unteres Limit erreicht).
- `qAtFull` (BOOL): `TRUE`, wenn `OUT` auf oder über `VAL_FULL` liegt (oberes Limit erreicht).

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock ermöglicht eine schrittweise Anpassung des Ausgabewertes `OUT` durch verschiedene Ereignisse:

- **Schrittweise Erhöhung/Verringerung**:
  - `UP_SLOW`/`DOWN_SLOW`: Ändert `OUT` um den `SLOW`-Wert.
  - `UP_FAST`/`DOWN_FAST`: Ändert `OUT` um den `FAST`-Wert.
- **Direkte Wertezuweisung**:
  - `ZERO` setzt `OUT` auf `VAL_ZERO`.
  - `FULL` setzt `OUT` auf `VAL_FULL`.
  - `LOAD` lädt den Wert von `PV` in `OUT`.

Der Ausgabewert wird dabei immer auf die Grenzen `VAL_ZERO` (Minimum) und `VAL_FULL` (Maximum) begrenzt.

Technische Besonderheiten

- **Begrenzung des Ausgabewertes**: Der Wert von `OUT` wird automatisch auf die definierten Grenzen (`VAL_ZERO`, `VAL_FULL`) begrenzt, falls die Schrittweiten (`SLOW`, `FAST`) diese überschreiten würden.
- **Flexible Schrittweiten**: Unterschiedliche Geschwindigkeiten der Wertänderung durch `SLOW` und `FAST`.
- **`INIT` ist Pflicht vor dem ersten Ereignis**: Jedes Ereignis liest per `WITH`-Deklaration nur die Dateneingänge ein, die es für seine eigene Berechnung braucht. Da jedes Ereignis am Ende `qAtZero`/`qAtFull` neu berechnet, steht in jeder WITH-Liste sowohl `VAL_ZERO` als auch `VAL_FULL` — z. B. liest `UP_SLOW` `SLOW`, `VAL_FULL` (für die eigentliche Begrenzung) **und** `VAL_ZERO` (nur für `qAtZero`), `DOWN_SLOW` liest `SLOW`, `VAL_ZERO` (Begrenzung) **und** `VAL_FULL` (nur für `qAtFull`). Ohne einen vorherigen `INIT`-Aufruf haben diese Werte noch nie einen gültigen Wert erhalten und stehen auf ihrem Vorgabewert 0 — bei `VAL_ZERO` fällt das meist nicht auf, bei `VAL_FULL` klemmt dann aber jeder `UP_SLOW`/`UP_FAST`-Schritt sofort auf 0.
- **Limit-Erkennung über `qAtZero`/`qAtFull`**: Jedes Ereignis (auch `INIT`) berechnet nach der eigentlichen Wertänderung `qAtZero := OUT <= VAL_ZERO` und `qAtFull := OUT >= VAL_FULL` neu und gibt beide über `INITO`/`CNF` mit aus. Damit muss ein aufrufender Baustein nicht selbst prüfen, ob eine Grenze erreicht ist — praktisch z. B. um bei Erreichen einer Grenze eine Bedienfläche auszublenden (siehe `ScrollFS`/`ScrollFS_PHYS_Softkey` in `isobus::UT::Q`, die genau das für die Scroll-Pfeiltasten nutzen).

Zustandsübersicht

Der Funktionsblock hat keinen internen Zustand. Jedes Ereignis führt sofort zur entsprechenden Berechnung und Ausgabe. Einzige Ausnahme: `INIT` muss einmal vor allen anderen Ereignissen laufen, damit `VAL_ZERO`/`VAL_FULL`/`SLOW`/`FAST`/`PV` überhaupt einen definierten Wert im Baustein haben.

Anwendungsszenarien

- **Tempomat-Steuerung**: Schrittweise Geschwindigkeitsanpassung mit langsamem/schnellem Tastendruck.
- **Prozesssteuerung**: Begrenzte Sollwertvorgabe mit manueller Feinjustierung.
- **Parameterladen**: Laden von vordefinierten Werten (z. B. aus einem nichtflüchtigen Speicher).

⚖️ Vergleich mit ähnlichen Bausteinen

- **Vergleich mit Standard-Rampenfunktionen**: `RampLimitFS` bietet zusätzlich die Möglichkeit, zwischen schnellen und langsamen Schrittweiten zu wechseln und direkte Werte (Min/Max) zuzuweisen.
- **Flexibilität**: Im Gegensatz zu einfachen Zählern ermöglicht dieser Baustein eine dynamische Anpassung der Schrittweiten und Grenzwerte.

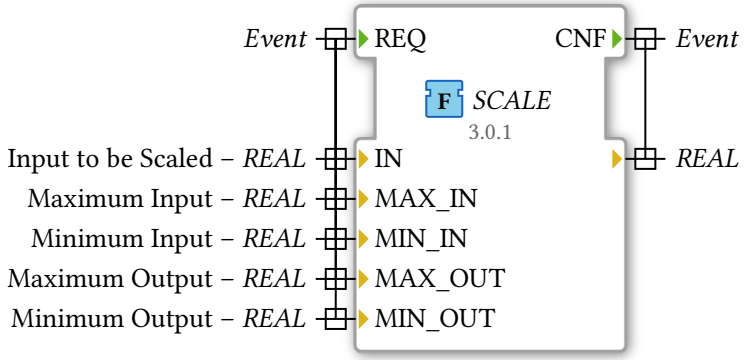
🛠️ Zugehörige Übungen

- Uebung_009a

Fazit

Der `RampLimitFS`-Funktionsblock ist ideal für Anwendungen, bei denen ein Wert schrittweise innerhalb definierter Grenzen angepasst werden muss. Durch die Unterscheidung zwischen schnellen und langsamen Änderungen sowie die Möglichkeit, direkte Werte zu laden, bietet er eine hohe Flexibilität für verschiedene Steuerungsaufgaben.

2.4.7.5 SCALE



• – ◦ • – ◦ • – ◦ •

Einleitung

Der SCALE-Funktionsblock dient zur Skalierung eines Eingangswertes von einem Eingangsbereich auf einen Ausgangsbereich. Dies ist besonders nützlich in der Signalverarbeitung, wenn Werte aus einem Messbereich in einen anderen Bereich transformiert werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Skalierungsoperation. Wird mit den Daten-Eingängen IN, MAX_IN, MIN_IN, MAX_OUT und MIN_OUT verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert die Beendigung der Skalierungsoperation.

Daten-Eingänge

- **IN**: Der Eingangswert, der skaliert werden soll (REAL).
- **MAX_IN**: Der maximale Wert des Eingangsbereichs (REAL).
- **MIN_IN**: Der minimale Wert des Eingangsbereichs (REAL).
- **MAX_OUT**: Der maximale Wert des Ausgangsbereichs (REAL).
- **MIN_OUT**: Der minimale Wert des Ausgangsbereichs (REAL).

Daten-Ausgänge

- **(unnamed)**: Der skalierte Ausgangswert (REAL).

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der SCALE-Funktionsblock berechnet den skalierten Wert nach folgender Formel:
`SCALE := (IN - MIN_IN) * (MAX_OUT - MIN_OUT) / (MAX_IN - MIN_IN) + MIN_OUT;`
Diese Formel transformiert den Eingangswert linear vom Eingangsbereich [MIN_IN, MAX_IN] auf den Ausgangsbereich [MIN_OUT, MAX_OUT].

Technische Besonderheiten

- Der Funktionsblock ist in ST (Structured Text) implementiert.
- Die Berechnung erfolgt in einem einzigen Schritt, was eine hohe Effizienz gewährleistet.
- Keine Zustandsverwaltung erforderlich, da es sich um eine statische Transformation handelt.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Operation wird bei jedem REQ-Ereignis neu ausgeführt.

Anwendungsszenarien

- Skalierung von Sensorwerten (z.B. von 4-20 mA auf 0-100 %).
- Transformation von Messwerten für die Anzeige oder Weiterverarbeitung.
- Allgemeine Signalverarbeitung in industriellen Steuerungssystemen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Skalierungsbausteinen ist SCALE besonders einfach und direkt in der Anwendung.
- Keine zusätzlichen Parameter oder komplexen Einstellungen erforderlich.
- Ideal für einfache lineare Skalierungen ohne zusätzliche Logik.

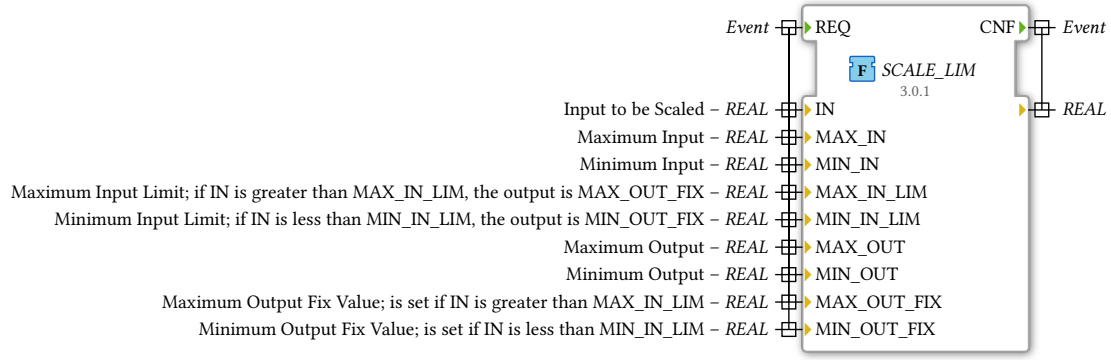
🛠️ Zugehörige Übungen

- Uebung_042

Fazit

Der SCALE-Funktionsblock bietet eine effiziente und einfache Lösung für lineare Skalierungsaufgaben. Seine klare Schnittstelle und direkte Implementierung machen ihn zu einem wertvollen Baustein in der Signalverarbeitung.

2.4.7.6 SCALE_LIM



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock SCALE_LIM ist ein Skalierungsbaustein mit zusätzlichen Limitierungsfunktionen. Er ermöglicht die lineare Skalierung eines Eingangswerts innerhalb eines definierten Bereichs und bietet zusätzlich die Möglichkeit, Ausgangswerte zu fixieren, wenn bestimmte Eingangsgrenzwerte überschritten werden.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Skalierungsoperation. Wird mit allen Daten-Eingängen verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert die Beendigung der Skalierungsoperation.

Daten-Eingänge

- **IN** (REAL): Eingangswert, der skaliert werden soll
- **MAX_IN** (REAL): Maximaler Eingangswert für die Skalierung
- **MIN_IN** (REAL): Minimaler Eingangswert für die Skalierung
- **MAX_IN_LIM** (REAL): Obere Eingangsgrenze, bei deren Überschreitung MAX_OUT_FIX ausgegeben wird
- **MIN_IN_LIM** (REAL): Untere Eingangsgrenze, bei deren Unterschreitung MIN_OUT_FIX ausgegeben wird
- **MAX_OUT** (REAL): Maximaler Ausgangswert für die Skalierung
- **MIN_OUT** (REAL): Minimaler Ausgangswert für die Skalierung
- **MAX_OUT_FIX** (REAL): Fixer Ausgangswert bei Überschreitung von MAX_IN_LIM
- **MIN_OUT_FIX** (REAL): Fixer Ausgangswert bei Unterschreitung von MIN_IN_LIM

Daten-Ausgänge

- (REAL): Skalierter Ausgangswert

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der FB führt folgende Operationen durch:

1. Prüft, ob der Eingangswert IN die definierten Grenzen MIN_IN_LIM oder MAX_IN_LIM überschreitet
2. Falls IN < MIN_IN_LIM: Ausgabe von MIN_OUT_FIX
3. Falls IN > MAX_IN_LIM: Ausgabe von MAX_OUT_FIX
4. Im normalen Betriebsbereich: Lineare Skalierung des Eingangswerts nach der Formel:

```
(IN - MIN_IN) * (MAX_OUT - MIN_OUT) / (MAX_IN - MIN_IN) + MIN_OUT
```

Technische Besonderheiten

- Implementiert in Structured Text (ST)
- Unterstützt sowohl Skalierung als auch Limitierung
- Teil des Pakets `eclipse4diac::signalprocessing`

Zustandsübersicht

Der FB hat einen einfachen Zustandsautomaten:

1. Wartet auf REQ-Ereignis
2. Führt Skalierung/Limitierung durch
3. Sendet CNF-Ereignis

Anwendungsszenarien

- Signalverarbeitung in Steuerungssystemen
- Skalierung von Sensorwerten mit Überwachung von Grenzwerten
- Prozessregelung mit definierten Betriebsbereichen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfachen Skalierungsbausteinen bietet SCALE_LIM zusätzlich:

- Grenzwertüberwachung
- Möglichkeit zur Ausgabe fixer Werte bei Grenzwertüberschreitung
- Flexiblere Definition von Skalierungsbereichen

🛠️ Zugehörige Übungen

- Uebung_043

Fazit

SCALE_LIM ist ein vielseitiger Funktionsblock für Anwendungen, die sowohl Skalierung als auch Limitierung von Werten erfordern. Die zusätzlichen Grenzwertfunktionen machen ihn besonders geeignet für sicherheitsrelevante Anwendungen, bei denen definierte Reaktionen auf Grenzwertüberschreitungen erforderlich sind.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

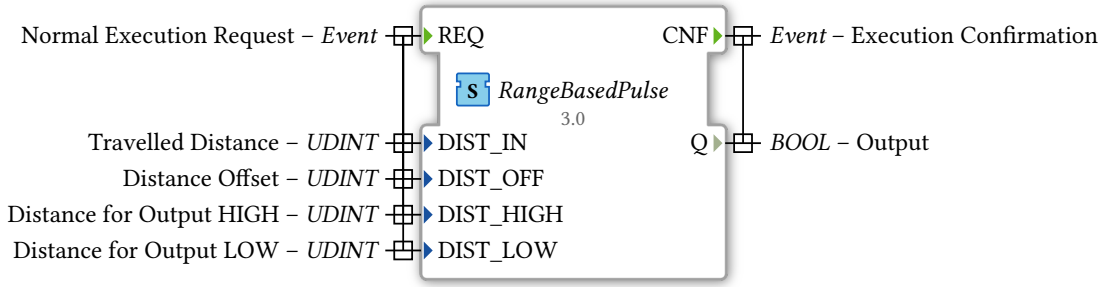
2.4.7.7 distance

RangeBasedPulse

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `RangeBasedPulse` ist ein distanzbasierter Impulsgenerator, der einen booleschen Ausgang (Q) basierend auf der zurückgelegten Distanz (DIST_IN) steuert. Der FB wird mit einem HIGH-Puls gestartet und erzeugt periodisch Ausgangsimpulse entsprechend den konfigurierten Distanzwerten für HIGH- und LOW-Phasen.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Normaler Ausführungsanforderung. Wird mit den Eingabedaten DIST_IN, DIST_OFF, DIST_HIGH und DIST_LOW verknüpft.

Ereignis-Ausgänge

- `CNF`: Ausführungsbestätigung. Wird mit dem Ausgang Q verknüpft.

Daten-Eingänge

- `DIST_IN` (UDINT): Zurückgelegte Distanz
- `DIST_OFF` (UDINT): Distanz-Offset
- `DIST_HIGH` (UDINT): Distanz für HIGH-Ausgang
- `DIST_LOW` (UDINT): Distanz für LOW-Ausgang

Daten-Ausgänge

- `Q` (BOOL): Ausgangssignal

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock berechnet den Restwert (DIST_REMAINDER) der Summe aus zurückgelegter Distanz (DIST_IN) und Offset (DIST_OFF) modulo der Summe von DIST_HIGH und DIST_LOW. Der Ausgang Q wird auf HIGH gesetzt, wenn DIST_REMAINDER größer oder gleich DIST_HIGH ist, andernfalls bleibt er LOW.

Technische Besonderheiten

- Verwendet UDINT (unsigned double integer) für Distanzberechnungen
- Implementiert eine einfache Modulo-Operation für die periodische Impulserzeugung
- Unterstützt einen Offset-Wert für flexible Startpositionen

Zustandsübersicht

- `REQ`: Einziger Zustand, der den Algorithmus ausführt und die CNF-Bestätigung sendet

Anwendungsszenarien

- Steuerung von Maschinen mit distanzbasierten Schaltvorgängen
- Erzeugung von periodischen Signalen basierend auf Wegstrecken
- Positionsabhängige Steuerung in Fahrzeug- oder Förderanlagen

⚖️ Vergleich mit ähnlichen Bausteinen

Im Gegensatz zu einfachen Zeitgebern arbeitet dieser FB mit Distanzwerten statt Zeitintervallen. Er bietet mehr Flexibilität durch den Offset-Parameter und ermöglicht die Definition von individuellen HIGH- und LOW-Phasen.

🛠️ Zugehörige Übungen

- Uebung_071b
- Uebung_072b

Fazit

Der RangeBasedPulse FB ist ein spezialisierter Funktionsblock für Anwendungen, die distanzbasierte Steuerungssignale benötigen. Durch seine einfache Konfiguration und flexible Offset-Einstellung eignet er sich gut für verschiedene industrielle Steuerungsaufgaben.

2.4.7.8 measurement

2.4.8 system

2.4.8.1 🎧 Podcast

- Eclipse 4diac: Ihr Fahrrad für die Industrie 4.0 – Software-Aufwand meistern und verteilte Systeme steuern
- IEC 61499: Die Zukunft der industriellen Automatisierung und verteilten Systeme
- Industrie 4.0: Die Magie selbstkonfigurierender Systeme mit fortiss, IEC 61499 und Eclipse 4diac
- Eclipse 4diac: Revolutionizing Industrial Control with Open-Source Cyber-Physical Systems
- Der E_PERMIT-Baustein: Der „Türsteher“ für Ereignisse in IEC 61499-Systemen entschlüsselt

---

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.8.2 devices

BeagleBoneBlack

• – ◦ • – ◦ • – ◦ •

Einleitung

Dieses Gerät repräsentiert eine FORTE-Laufzeitumgebung auf einem BeagleBone Black Board.

Metadaten

Attribut	Wert
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	iec61499::system

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

FBRT_WINDOW

• – ◦ • – ◦ • – ◦ •

Einleitung

Dieses Gerät repräsentiert eine FBRT (Function Block RunTime) Fenster-Anwendung.

Metadaten

Attribut	Wert
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	iec61499::system

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

FORTE_PC

• – ◦ • – ◦ • – ◦ •

Einleitung
Dieses Gerät repräsentiert eine FORTE-Laufzeitumgebung auf einem Standard-PC (Windows oder Linux).

Metadaten

Attribut	Wert
Copyright	(c) 2015 fortiss GmbH
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	iec61499::system

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

PFC200

• – ◦ • – ◦ • – ◦ •

Einleitung
Dieses Gerät repräsentiert eine FORTE-Laufzeitumgebung auf einem WAGO PFC200 Controller.

Metadaten

Attribut	Wert
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	iec61499::system

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

RaspberryPI

• – ◦ • – ◦ • – ◦ •

Einleitung
Dieses Gerät repräsentiert eine FORTE-Laufzeitumgebung auf einem Raspberry Pi.

Metadaten

Attribut	Wert
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	iec61499::system

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.8.3 resources

Config_EMB_RES

• – ◦ • – ◦ • – ◦ •

Einleitung

Eine konfigurierbare Embedded-Ressource für die Ausführung von IEC 61499 Anwendungen.

Metadaten

Attribut	Wert
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	iec61499::system

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

EMB_RES

• – ◦ • – ◦ • – ◦ •

Einleitung

Eine grundlegende Embedded-Ressource für die Ausführung von Funktionsblock-Netzwerken. Sie beinhaltet standardmäßig einen E_RESTART Funktionsblock für Start/Stopp-Ereignisse.

Metadaten

Attribut	Wert
Copyright	(c) 2017 fortiss GmbH
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	iec61499::system

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.8.4 segments

Ethernet

• – ◦ • – ◦ • – ◦ •

Einleitung

Ein Ethernet-Netzwerksegment zur Verbindung von Geräten.

Metadaten

Attribut	Wert
Copyright	(c) 2008 Profactor GmbH
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	iec61499::system

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

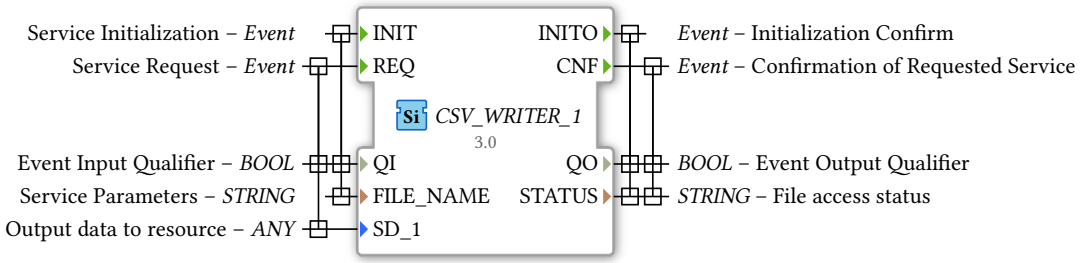
2.4.9 utils

2.4.9.1 CSV_WRITER_1

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `CSV_WRITER_1` dient als Service-Interface zum Schreiben von Daten in eine CSV-Datei. Er ermöglicht die Initialisierung des Schreibvorgangs sowie das kontrollierte Schreiben von Daten in eine Datei mit Statusrückmeldungen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisiert den Schreibservice. Wird mit den Variablen `QI` (Qualifier) und `FILE_NAME` (Dateiname) ausgelöst.
- **REQ**: Fordert das Schreiben von Daten an. Wird mit den Variablen `QI` und `SD_1` (zu schreibende Daten) ausgelöst.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung. Liefert `QO` (Qualifier) und `STATUS` (Zustand der Initialisierung).
- **CNF**: Bestätigt den Abschluss des Schreibvorgangs. Liefert `QO` und `STATUS` (Zustand des Schreibvorgangs).

Daten-Eingänge

- **QI**: Boolescher Qualifier für Ereignisse.
- **FILE_NAME**: String, der den Namen der zu schreibenden Datei angibt.
- **SD_1**: Daten vom Typ `ANY`, die in die CSV-Datei geschrieben werden sollen.

Daten-Ausgänge

- **QO**: Boolescher Qualifier für Ausgangsereignisse.
- **STATUS**: String, der den Status des Dateizugriffs zurückgibt.

Adapter

Keine Adapter vorhanden.

Funktionsweise

1. **Initialisierung**: Durch das `INIT`-Ereignis wird der Schreibservice mit dem angegebenen Dateinamen (`FILE_NAME`) initialisiert. Der Status wird über `INITO` zurückgemeldet.
2. **Daten schreiben**: Das `REQ`-Ereignis löst den Schreibvorgang der Daten (`SD_1`) aus. Der Erfolg oder Fehler wird über `CNF` und den `STATUS` gemeldet.

Technische Besonderheiten

- Unterstützt generische Datentypen (`ANY`) für die zu schreibenden Daten.
- Rückmeldung des Status als String für detaillierte Fehleranalyse.

Zustandsübersicht

- **Initialisierungszustand**: Wird durch `INIT` gesetzt und durch `INITO` bestätigt.
- **Schreibzustand**: Wird durch `REQ` aktiviert und durch `CNF` abgeschlossen.

Anwendungsszenarien

- Protokollierung von Prozessdaten in CSV-Dateien.
- Export von Messdaten für weitere Analysen.
- Automatisierte Datenspeicherung in industriellen Steuerungssystemen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu einfachen Dateischreib-Blöcken bietet `CSV_WRITER_1` eine detaillierte Statusrückmeldung und Unterstützung für generische Datentypen.
- Andere Blöcke könnten auf bestimmte Datentypen beschränkt sein oder weniger Feedback bieten.

Fazit

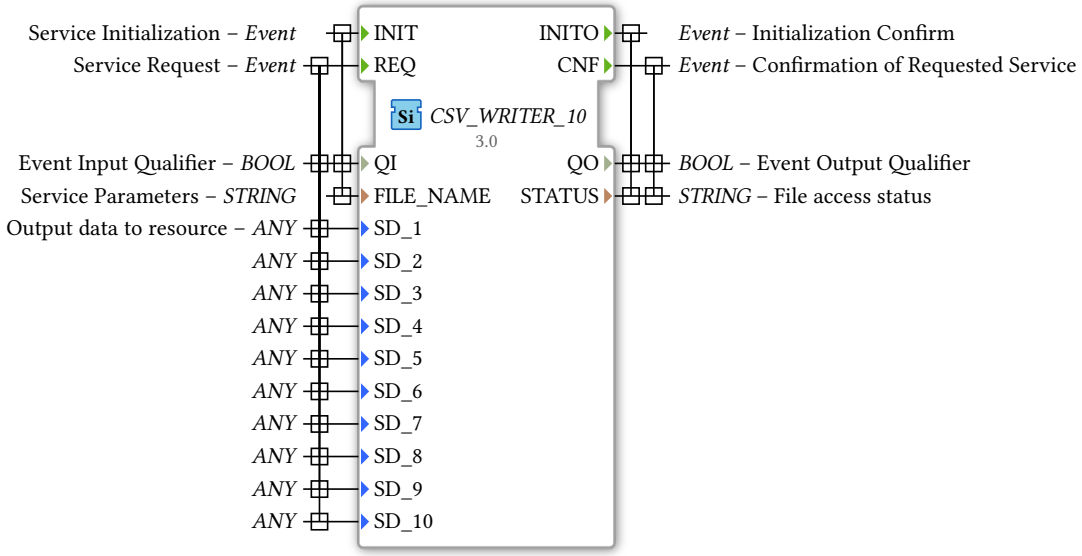
Der `CSV_WRITER_1` ist ein flexibler und zuverlässiger Funktionsblock für das Schreiben von Daten in CSV-Dateien, ideal für Anwendungen, die eine robuste Protokollierung und Datenexportfunktionalität erfordern.

2.4.9.2 CSV_WRITER_10

• – ◦ • – ◦ • – ◦ •

Einleitung

Der CSV_WRITER_10 Funktionsblock dient zum Schreiben von Daten in eine CSV-Datei. Er ermöglicht das Speichern von bis zu 10 verschiedenen Datenwerten in einer strukturierten Textdatei. Der Block ist besonders nützlich für Datenerfassung, Protokollierung und den Austausch von Daten mit anderen Systemen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisiert den Funktionsblock und gibt den Dateinamen an (mit QI und FILE_NAME)
- **REQ**: Löst den Schreibvorgang aus (mit QI und SD_1 bis SD_10)

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung (mit QO und STATUS)
- **CNF**: Bestätigt den Schreibvorgang (mit QO und STATUS)

Daten-Eingänge

- **QI**: BOOL - Qualifiziert die Ereigniseingänge
- **FILE_NAME**: STRING - Name der CSV-Datei
- **SD_1 bis SD_10**: ANY - Zu schreibende Datenwerte (beliebiger Typ)

Daten-Ausgänge

- **QO**: BOOL - Qualifiziert die Ereignisausgänge
- **STATUS**: STRING - Statusmeldung des Dateizugriffs

Adapter

Keine Adapter vorhanden.

Funktionsweise

1. **Initialisierung**: Durch INIT mit gültigem FILE_NAME wird die CSV-Datei vorbereitet
2. **Daten schreiben**: Bei REQ werden die aktuellen Werte von SD_1 bis SD_10 als neue Zeile in die Datei geschrieben
3. **Statusrückmeldung**: Jede Aktion generiert eine Statusmeldung (erfolgreich/fehlgeschlagen)

Technische Besonderheiten

- Unterstützt beliebige Datentypen (ANY) für die zu schreibenden Werte
- Automatische Konvertierung der Daten in Textformat
- Thread-sichere Dateizugriffe
- Maximale Kapazität von 10 Datenwerten pro Zeile

Zustandsübersicht

1. **Nicht initialisiert**: Vor INIT
2. **Bereit**: Nach erfolgreicher Initialisierung
3. **Fehler**: Bei Dateizugriffsproblemen

Anwendungsszenarien

- Datenerfassung von Prozesswerten
- Erstellung von Logdateien
- Export von Messdaten für externe Analyse
- Schnittstelle zu Tabellenkalkulationsprogrammen

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu einfacheren CSV-Writern bietet dieser Block mehr Flexibilität durch 10 Datenkanäle
- Unterstützt im Gegensatz zu fest typisierten Writern beliebige Datentypen
- Bietet detailliertere Statusrückmeldungen als Basisversionen

Fazit

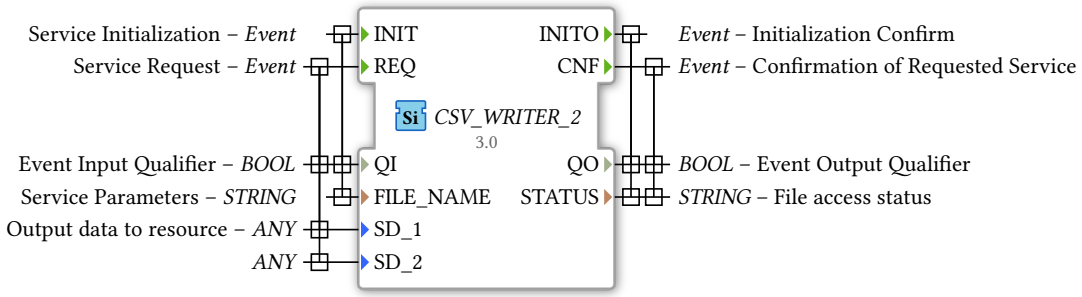
Der CSV_WRITER_10 ist ein leistungsfähiger Funktionsblock für strukturierte Datenspeicherung, der sich besonders für Anwendungen mit mehreren gleichzeitig zu protokollierenden Werten eignet. Die flexible Typunterstützung und detaillierte Statusrückmeldung machen ihn zu einer robusten Lösung für industrielle Datenerfassungsaufgaben.

2.4.9.3 CSV_WRITER_2

• – ◦ • – ◦ • – ◦ •

Einleitung

Der CSV_WRITER_2 ist ein Funktionsblock, der dazu dient, Daten in eine CSV-Datei zu schreiben. Er bietet eine einfache Schnittstelle für die Initialisierung und den Schreibvorgang, wobei zwei verschiedene Datenfelder unterstützt werden. Der Funktionsblock ist besonders nützlich für Anwendungen, bei denen Daten persistent gespeichert oder für weitere Analysen exportiert werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisiert den Funktionsblock und konfiguriert den Dateinamen für die CSV-Datei.
  - Eingangsdaten: `QI` (BOOL), `FILE_NAME` (STRING)
- **REQ**: Startet den Schreibvorgang der Daten in die CSV-Datei.
  - Eingangsdaten: `QI` (BOOL), `SD_1` (ANY), `SD_2` (ANY)

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung des Funktionsblocks.
  - Ausgangsdaten: `QO` (BOOL), `STATUS` (STRING)
- **CNF**: Bestätigt den erfolgreichen Abschluss des Schreibvorgangs.
  - Ausgangsdaten: `QO` (BOOL), `STATUS` (STRING)

Daten-Eingänge

- **QI**: Qualifiziert das Ereignis (TRUE für gültig, FALSE für ungültig).
- **FILE_NAME**: Der Name der CSV-Datei, in die geschrieben werden soll.
- **SD_1**: Das erste Datenfeld, das in die CSV-Datei geschrieben wird (beliebiger Datentyp).
- **SD_2**: Das zweite Datenfeld, das in die CSV-Datei geschrieben wird (beliebiger Datentyp).

Daten-Ausgänge

- **QO**: Qualifiziert das Ausgangsereignis (TRUE für erfolgreich, FALSE für fehlgeschlagen).
- **STATUS**: Gibt den Status des Dateizugriffs zurück (z.B. Erfolgsmeldung oder Fehlerbeschreibung).

Adapter

Der Funktionsblock verfügt über keine Adapter.

Funktionsweise

1. **Initialisierung**: Durch das Senden eines `INIT`-Ereignisses mit dem gewünschten Dateinamen (`FILE_NAME`) wird der Funktionsblock initialisiert. Der Status der Initialisierung wird über `INITO` zurückgemeldet.
2. **Daten schreiben**: Das `REQ`-Ereignis löst den Schreibvorgang aus. Die Daten `SD_1` und `SD_2` werden in die CSV-Datei geschrieben. Der Erfolg oder Misserfolg des Schreibvorgangs wird über `CNF` gemeldet.

Technische Besonderheiten

- Unterstützt beliebige Datentypen für die zu schreibenden Daten (`SD_1` und `SD_2`).
- Der Status (`STATUS`) gibt detaillierte Informationen über den Erfolg oder Fehler des Dateizugriffs.
- Der Funktionsblock ist Teil des `eclipse4diac::utils`-Pakets.

Zustandsübersicht

1. **Inaktiv**: Der Funktionsblock wartet auf Initialisierung.
2. **Initialisiert**: Der Funktionsblock ist bereit, Daten zu schreiben.
3. **Schreibvorgang**: Der Funktionsblock schreibt Daten in die CSV-Datei.

Anwendungsszenarien

- Protokollierung von Prozessdaten in einer CSV-Datei für spätere Analysen.
- Export von Messdaten für externe Auswertungen.
- Speicherung von Konfigurationsdaten in einem lesbaren Format.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu einfacheren Schreibfunktionsblöcken unterstützt `CSV_WRITER_2` zwei Datenfelder und bietet eine detaillierte Statusrückmeldung.
- Andere CSV-Schreibfunktionsblöcke könnten weniger flexibel in der Datentypunterstützung sein oder keine Statusrückmeldung bieten.

Fazit

Der `CSV_WRITER_2` ist ein flexibler und zuverlässiger Funktionsblock für das Schreiben von Daten in CSV-Dateien. Seine Unterstützung für beliebige Datentypen und detaillierte Statusrückmeldungen machen ihn ideal für Anwendungen, bei denen Daten persistent gespeichert werden müssen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

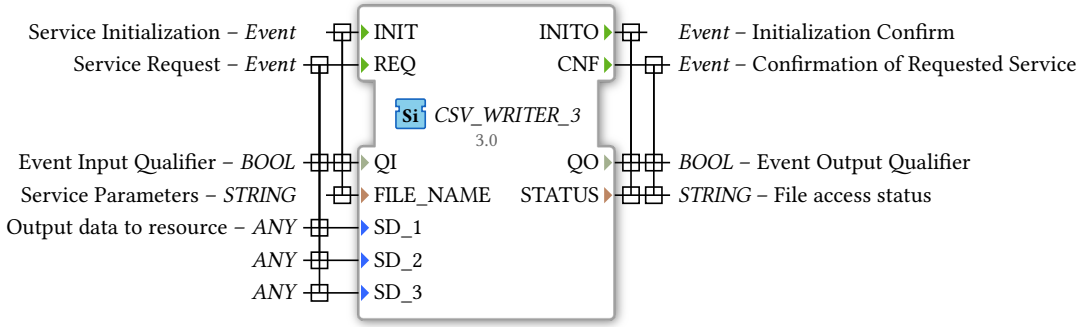
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.9.4 CSV_WRITER_3

• – ◦ • – ◦ • – ◦ •

Einleitung

Der CSV_WRITER_3 ist ein Funktionsblock, der dazu dient, Daten in eine CSV-Datei zu schreiben. Er unterstützt das Schreiben von bis zu drei verschiedenen Datenfeldern in eine Datei, die über den Parameter `FILE_NAME` angegeben wird. Der Funktionsblock ist besonders nützlich für die Protokollierung und Datenspeicherung in Automatisierungssystemen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisiert den Funktionsblock. Wird mit den Daten `QI` (Qualifier Input) und `FILE_NAME` (Dateiname) ausgelöst.
- **REQ**: Startet den Schreibvorgang. Wird mit den Daten `QI`, `SD_1`, `SD_2` und `SD_3` (die zu schreibenden Daten) ausgelöst.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung. Gibt `QO` (Qualifier Output) und `STATUS` (Status der Initialisierung) zurück.
- **CNF**: Bestätigt den Abschluss des Schreibvorgangs. Gibt `QO` und `STATUS` zurück.

Daten-Eingänge

- **QI**: Boolescher Wert, der die Qualität des Ereigniseingangs angibt.
- **FILE_NAME**: String, der den Namen der CSV-Datei angibt.
- **SD_1**, **SD_2**, **SD_3**: Datenfelder vom Typ `ANY`, die in die CSV-Datei geschrieben werden.

Daten-Ausgänge

- **QO**: Boolescher Wert, der die Qualität des Ereignisausgangs angibt.
- **STATUS**: String, der den Status des Dateizugriffs angibt.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock initialisiert sich beim Empfang des `INIT`-Ereignis und bestätigt dies mit `INITO`. Beim Empfang des `REQ`-Ereignis werden die Daten `SD_1`, `SD_2` und `SD_3` in die angegebene CSV-Datei geschrieben. Der Status des Schreibvorgangs wird über `CNF` zurückgemeldet.

Technische Besonderheiten

- Unterstützt das Schreiben von bis zu drei verschiedenen Datenfeldern.
- Der Datentyp der zu schreibenden Daten ist `ANY`, was eine hohe Flexibilität ermöglicht.
- Der Status des Dateizugriffs wird detailliert zurückgemeldet.

Zustandsübersicht

1. **Initialisierungszustand**: Wartet auf `INIT`-Ereignis.
2. **Bereit-Zustand**: Nach erfolgreicher Initialisierung, wartet auf `REQ`-Ereignis.
3. **Schreibzustand**: Schreibt Daten in die CSV-Datei und meldet den Status zurück.

Anwendungsszenarien

- Protokollierung von Sensordaten in einer CSV-Datei.
- Speicherung von Prozessdaten für spätere Analysen.
- Datenaustausch mit anderen Systemen über CSV-Dateien.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu einfacheren CSV-Writer-Funktionsblöcken unterstützt `CSV_WRITER_3` das Schreiben von mehreren Datenfeldern (`SD_1`, `SD_2`, `SD_3`).
- Andere Funktionsblöcke könnten auf bestimmte Datentypen beschränkt sein, während `CSV_WRITER_3` den `ANY`-Typ verwendet.

Fazit

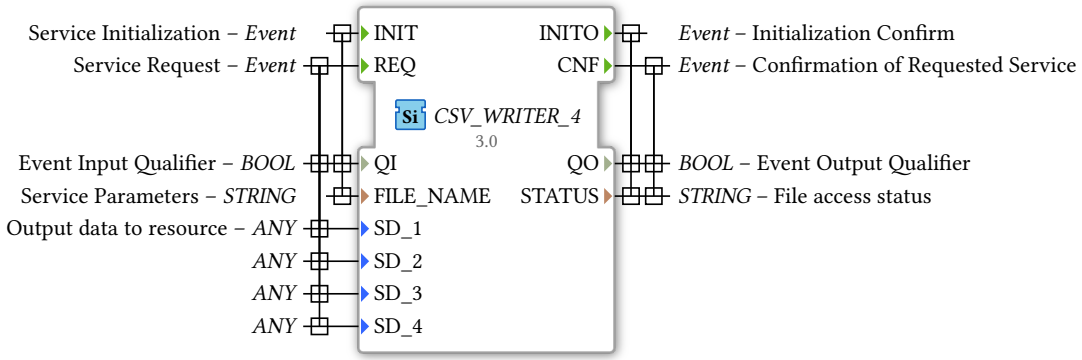
Der `CSV_WRITER_3` ist ein flexibler und leistungsfähiger Funktionsblock für die Datenspeicherung in CSV-Dateien. Seine Unterstützung für mehrere Datenfelder und den `ANY`-Datentyp macht ihn besonders vielseitig einsetzbar.

2.4.9.5 CSV_WRITER_4

• – ◦ • – ◦ • – ◦ •

Einleitung

Der CSV_WRITER_4 Funktionsblock dient zum Schreiben von Daten in eine CSV-Datei. Er unterstützt das Speichern von bis zu vier verschiedenen Datenwerten in einem strukturierten Format. Der FB ist besonders nützlich für Datenerfassung und -protokollierung in Automatisierungsanwendungen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisiert den Funktionsblock und konfiguriert die Ausgabedatei. Wird mit `QI` und `FILE_NAME` Daten verknüpft.
- **REQ**: Löst den Schreibvorgang aus. Wird mit `QI` und den Datenwerten `SD_1` bis `SD_4` verknüpft.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung. Liefert `QO` und `STATUS`.
- **CNF**: Bestätigt den abgeschlossenen Schreibvorgang. Liefert `QO` und `STATUS`.

Daten-Eingänge

- **QI** (BOOL): Qualifiziert die Ereignisse INIT und REQ.
- **FILE_NAME** (STRING): Name der CSV-Datei.
- **SD_1** bis **SD_4** (ANY): Die zu schreibenden Datenwerte (bis zu 4 verschiedene).

Daten-Ausgänge

- **QO** (BOOL): Qualifiziert die Ausgangsereignisse.
- **STATUS** (STRING): Statusmeldung über den Dateizugriff.

Adapter

Keine Adapter vorhanden.

Funktionsweise

1. **Initialisierung**: Durch das INIT-Ereignis mit aktivem `QI` wird die CSV-Datei mit dem angegebenen `FILE_NAME` vorbereitet.
2. **Daten schreiben**: Das REQ-Ereignis mit aktivem `QI` löst den Schreibvorgang aus. Die Werte von `SD_1` bis `SD_4` werden in die Datei geschrieben.
3. **Bestätigung**: Nach jedem Vorgang (INIT oder REQ) wird der Status über `INITO` bzw. `CNF` gemeldet.

Technische Besonderheiten

- Unterstützt generische Datentypen (ANY) für die zu schreibenden Daten.
- Der FB ist Teil des `eclipse4diac::utils` Pakets.
- Die generische Klasse ist als `GEN_CSV_WRITER` definiert.

Zustandsübersicht

1. **Inaktiv**: Wartet auf Initialisierung.
2. **Initialisiert**: Datei ist bereit, wartet auf Schreibanfragen.
3. **Schreibend**: Verarbeitet eine Schreibanfrage.

Anwendungsszenarien

- Protokollierung von Prozessdaten.
- Datenerfassung für Analysezwecke.
- Export von Messwerten in ein standardisiertes Format.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu einfacheren Schreib-FBs bietet CSV_WRITER_4 die Möglichkeit, mehrere Datenwerte gleichzeitig zu speichern.
- Die Verwendung von ANY-Typen macht ihn flexibler als FBs mit festen Datentypen.

Fazit

CSV_WRITER_4 ist ein vielseitiger Funktionsblock für die Datenerfassung, der sich besonders für Anwendungen eignet, bei denen strukturierte Daten protokolliert werden müssen. Seine Flexibilität und einfache Integration machen ihn zu einer praktischen Lösung für viele Automatisierungsaufgaben.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

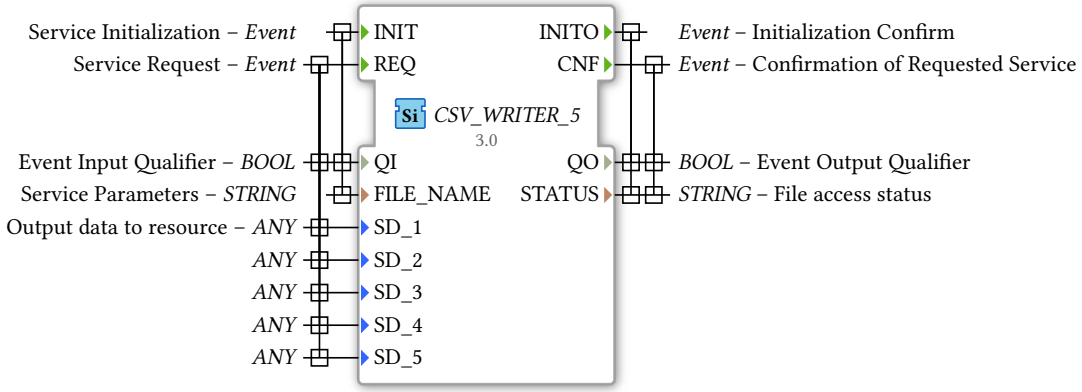
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.9.6 CSV_WRITER_5

• – ◦ • – ◦ • – ◦ •

Einleitung

Der CSV_WRITER_5 ist ein Funktionsblock zur Ausgabe von Daten in eine CSV-Datei. Er ermöglicht das Schreiben von bis zu fünf verschiedenen Datenpunkten in eine benutzerdefinierte Datei. Der Funktionsblock ist besonders nützlich für Datenerfassungs- und Protokollierungsanwendungen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisiert den Funktionsblock und öffnet die angegebene CSV-Datei.
  - Verknüpfte Daten: QI (BOOL), FILE_NAME (STRING)
- **REQ**: Löst den Schreibvorgang aus.
  - Verknüpfte Daten: QI (BOOL), SD_1 bis SD_5 (ANY)

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung.
  - Verknüpfte Daten: QO (BOOL), STATUS (STRING)
- **CNF**: Bestätigt den erfolgreichen Schreibvorgang.
  - Verknüpfte Daten: QO (BOOL), STATUS (STRING)

Daten-Eingänge

- **QI**: Qualifiziert das Ereignis (TRUE für aktiv, FALSE für inaktiv).
- **FILE_NAME**: Name der CSV-Datei, in die geschrieben werden soll.
- **SD_1 bis SD_5**: Die zu schreibenden Daten (beliebiger Typ).

Daten-Ausgänge

- **QO**: Gibt den Status des Funktionsblocks zurück (TRUE für erfolgreich, FALSE für Fehler).
- **STATUS**: Enthält eine Statusmeldung über den letzten Vorgang.

Adapter

Keine Adapter vorhanden.

Funktionsweise

1. **Initialisierung**: Durch das INIT-Ereignis wird die CSV-Datei mit dem angegebenen Namen geöffnet. Falls die Datei nicht existiert, wird sie erstellt.
2. **Daten schreiben**: Das REQ-Ereignis löst den Schreibvorgang aus. Die Daten SD_1 bis SD_5 werden als eine Zeile in die CSV-Datei geschrieben.
3. **Bestätigung**: Nach jedem Schreibvorgang wird das CNF-Ereignis ausgelöst, das den Status des Vorgangs zurückgibt.

Technische Besonderheiten

- Unterstützt beliebige Datentypen für die Ausgabedaten (ANY).
- Der Statusausgang (STATUS) gibt detaillierte Fehlermeldungen zurück.

- Die Initialisierung kann bei Bedarf wiederholt werden, um eine neue Datei zu öffnen.

Zustandsübersicht

- **Inaktiv**: Wartet auf Initialisierung.
- **Bereit**: Datei ist geöffnet und bereit für Schreibvorgänge.
- **Fehler**: Ein Fehler ist aufgetreten (z.B. Dateizugriff verweigert).

Anwendungsszenarien

- Protokollierung von Sensordaten.
- Speicherung von Prozessdaten für spätere Analysen.
- Export von Daten in ein tabellenbasiertes Format.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu einfacheren CSV-Writer-Blöcken bietet CSV_WRITER_5 mehr Flexibilität durch die Unterstützung von fünf Datenpunkten und beliebigen Datentypen.
- Andere Blöcke könnten auf bestimmte Datentypen beschränkt sein oder weniger Datenpunkte unterstützen.

Fazit

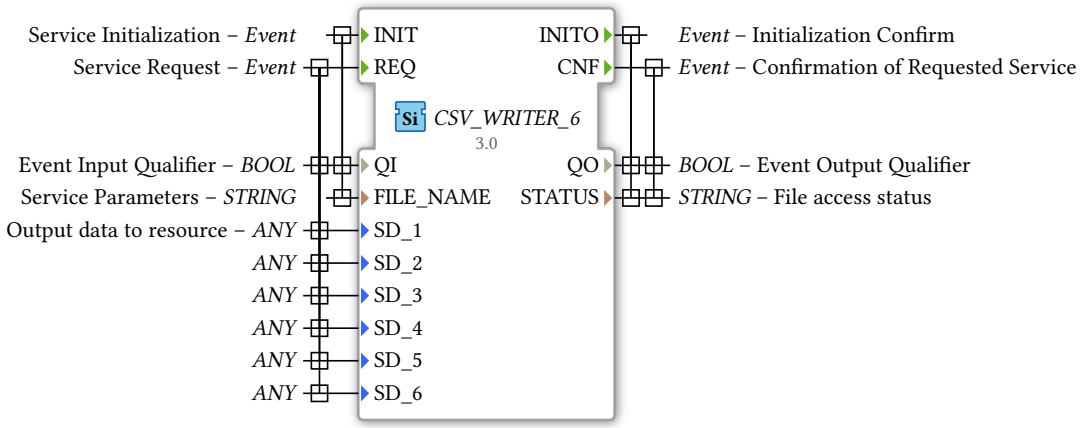
Der CSV_WRITER_5 ist ein leistungsfähiger und flexibler Funktionsblock für die Datenerfassung und -protokollierung. Seine Unterstützung für beliebige Datentypen und mehrere Datenpunkte macht ihn ideal für komplexe Anwendungen.

2.4.9.7 CSV_WRITER_6

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `CSV_WRITER_6` dient zum Schreiben von Daten in eine CSV-Datei. Er ermöglicht die Speicherung von bis zu sechs verschiedenen Datenwerten in einem komma-separierten Format. Der Block ist besonders nützlich für Datenerfassung und -protokollierung in Automatisierungssystemen.



Schnittstellenstruktur

Ereignis-Eingänge

- `INIT`: Initialisiert den Funktionsblock. Wird mit den Variablen `QI` und `FILE_NAME` ausgelöst.
- `REQ`: Startet den Schreibvorgang. Wird mit den Variablen `QI`, `SD_1` bis `SD_6` ausgelöst.

Ereignis-Ausgänge

- `INITO`: Bestätigt die Initialisierung. Gibt `QO` und `STATUS` zurück.
- `CNF`: Bestätigt den erfolgreichen Schreibvorgang. Gibt `QO` und `STATUS` zurück.

Daten-Eingänge

- `QI` (BOOL): Qualifiziert das Ereignis (z.B., ob der Schreibvorgang aktiviert werden soll).
- `FILE_NAME` (STRING): Name der CSV-Datei, in die geschrieben werden soll.

- `SD_1` bis `SD_6` (ANY): Die zu schreibenden Datenwerte.

Daten-Ausgänge
- `QO` (BOOL): Qualifiziert das Ausgangsereignis (z.B., ob der Vorgang erfolgreich war).
- `STATUS` (STRING): Statusmeldung des Dateizugriffs.

Adapter

Keine Adapter vorhanden.

Funktionsweise
1. **Initialisierung**: Durch das `INIT`-Ereignis wird der Funktionsblock mit dem Dateinamen (`FILE_NAME`) initialisiert. Der Status wird über `INITO` zurückgemeldet.
2. **Schreibvorgang**: Das `REQ`-Ereignis löst den Schreibvorgang aus. Die Daten `SD_1` bis `SD_6` werden in die CSV-Datei geschrieben. Der Erfolg wird über `CNF` bestätigt.

Technische Besonderheiten
- Unterstützt beliebige Datentypen (`ANY`) für die zu schreibenden Daten.
- Der Status (`STATUS`) gibt detaillierte Informationen über Fehler oder Erfolg des Dateizugriffs.

Zustandsübersicht
1. **Initialisierungszustand**: Wartet auf `INIT`-Ereignis.
2. **Bereit-Zustand**: Nach erfolgreicher Initialisierung wartet auf `REQ`-Ereignis.
3. **Schreibzustand**: Verarbeitet die Daten und schreibt sie in die Datei.

Anwendungsszenarien
- Protokollierung von Sensordaten in Echtzeit.
- Speicherung von Prozessdaten für spätere Analysen.
- Datenerfassung in Test- und Validierungsumgebungen.

⚖️ Vergleich mit ähnlichen Bausteinen
- Im Vergleich zu einfacheren CSV-Writer-Blöcken bietet `CSV_WRITER_6` die Möglichkeit, bis zu sechs verschiedene Datenwerte gleichzeitig zu speichern.
- Andere Blöcke könnten auf bestimmte Datentypen beschränkt sein, während dieser Block `ANY`-Typen unterstützt.

Fazit

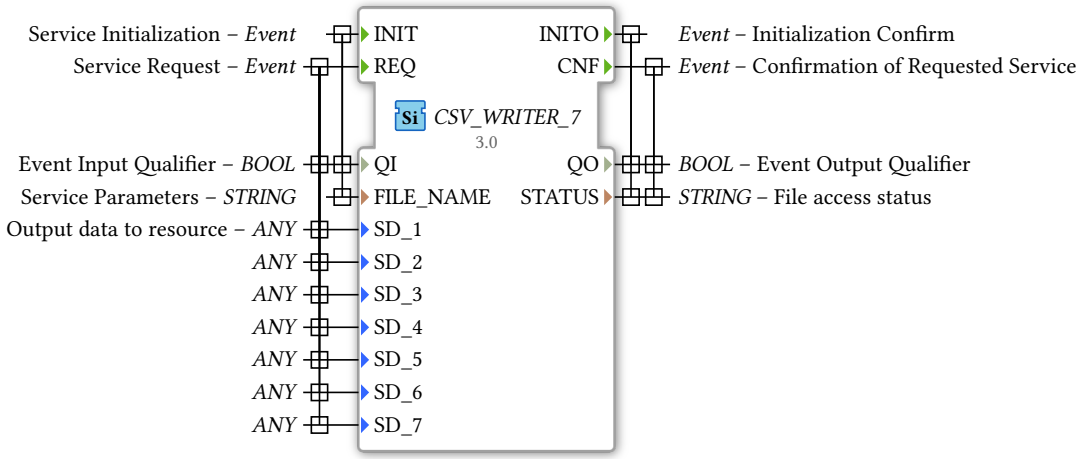
Der `CSV_WRITER_6` ist ein flexibler und leistungsfähiger Funktionsblock für die Datenspeicherung in CSV-Dateien. Seine Unterstützung für verschiedene Datentypen und die Möglichkeit, mehrere Werte gleichzeitig zu speichern, machen ihn ideal für komplexe Datenerfassungsaufgaben.

2.4.9.8 CSV_WRITER_7

• – ◦ • – ◦ • – ◦ •

Einleitung

Der CSV_WRITER_7 ist ein Funktionsblock, der Daten in eine CSV-Datei schreibt. Er unterstützt das Schreiben von bis zu sieben verschiedenen Datenfeldern in eine angegebene Datei. Der Funktionsblock ist besonders nützlich für die Protokollierung und Datenspeicherung in Automatisierungsanwendungen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisiert den Funktionsblock. Wird mit den Variablen `QI` (Qualifier Input) und `FILE_NAME` (Name der CSV-Datei) verwendet.
- **REQ**: Startet den Schreibvorgang. Wird mit den Variablen `QI` und den sieben Datenfeldern `SD_1` bis `SD_7` verwendet.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung. Gibt `QO` (Qualifier Output) und `STATUS` (Status der Initialisierung) zurück.
- **CNF**: Bestätigt den abgeschlossenen Schreibvorgang. Gibt `QO` und `STATUS` zurück.

Daten-Eingänge

- **QI**: Boolescher Qualifier, der angibt, ob der Vorgang ausgeführt werden soll.
- **FILE_NAME**: Name der CSV-Datei, in die geschrieben werden soll.
- **SD_1** bis **SD_7**: Datenfelder, die in die CSV-Datei geschrieben werden. Jedes Feld kann einen beliebigen Datentyp (`ANY`) haben.

Daten-Ausgänge

- **QO**: Boolescher Qualifier, der den Erfolg oder Misserfolg des Vorgangs anzeigt.
- **STATUS**: Zeichenkette, die den Status des Dateizugriffs beschreibt.

Adapter

Es sind keine Adapter vorhanden.

Funktionsweise

1. **Initialisierung**: Der Funktionsblock wird durch das `INIT`-Ereignis initialisiert. Dabei wird der Dateiname (`FILE_NAME`) übergeben.
2. **Schreibvorgang**: Das `REQ`-Ereignis löst den Schreibvorgang aus. Die Daten `SD_1` bis `SD_7` werden in die angegebene CSV-Datei geschrieben.
3. **Bestätigung**: Nach Abschluss des Vorgangs wird das `CNF`-Ereignis ausgelöst, das den Status des Vorgangs zurückgibt.

Technische Besonderheiten

- Unterstützt das Schreiben von bis zu sieben verschiedenen Datenfeldern.
- Die Datenfelder können beliebige Datentypen (`ANY`) haben.
- Der Status des Dateizugriffs wird detailliert zurückgegeben.

Zustandsübersicht

1. **Initialisierungszustand**: Wartet auf das `INIT`-Ereignis.
2. **Bereit**: Nach erfolgreicher Initialisierung wartet der Funktionsblock auf das `REQ`-Ereignis.

3. **Schreibvorgang**: Während des Schreibvorgangs ist der Funktionsblock aktiv.
4. **Abgeschlossen**: Nach Abschluss des Schreibvorgangs wird das `CNF`-Ereignis ausgelöst.

Anwendungsszenarien

- Protokollierung von Prozessdaten in der Industrieautomation.
- Speicherung von Messdaten für spätere Analysen.
- Datenerfassung in Echtzeitsystemen.

⚖️ Vergleich mit ähnlichen Bausteinen

- **CSV_WRITER_5**: Unterstützt nur fünf Datenfelder.
- **FILE_WRITER**: Allgemeinerer Baustein, der nicht speziell für CSV-Dateien optimiert ist.

Fazit

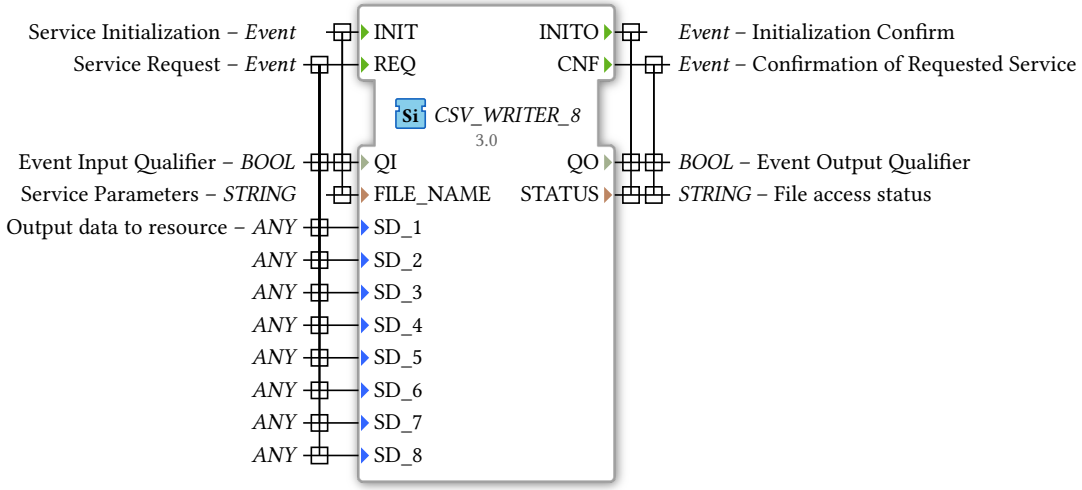
Der CSV_WRITER_7 ist ein leistungsfähiger Funktionsblock für die Datenspeicherung in CSV-Dateien. Seine Flexibilität bei der Unterstützung verschiedener Datentypen und die detaillierte Statusrückmeldung machen ihn zu einer idealen Wahl für viele Automatisierungsanwendungen.

2.4.9.9 CSV_WRITER_8

• – ◦ • – ◦ • – ◦ •

Einleitung

Der CSV_WRITER_8 ist ein Funktionsblock zur Ausgabe von Daten in eine CSV-Datei. Er ermöglicht das Schreiben von bis zu 8 verschiedenen Datenwerten in eine spezifizierte Datei. Der Funktionsblock ist Teil der eclipse4diac::utils-Paketfamilie und eignet sich besonders für Datenerfassungs- und Protokollierungsaufgaben.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisiert den Funktionsblock. Wird mit den Variablen QI (Qualifier) und FILE_NAME (Dateiname) verwendet.
- **REQ**: Löst einen Schreibvorgang aus. Verwendet QI sowie die Datenvariablen SD_1 bis SD_8.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung. Gibt QO (Qualifier) und STATUS (Zustandsmeldung) zurück.
- **CNF**: Bestätigt den abgeschlossenen Schreibvorgang. Gibt ebenfalls QO und STATUS zurück.

Daten-Eingänge

- **QI**: Boolescher Qualifier für Ereignisse.
- **FILE_NAME**: String, der den Pfad und Namen der CSV-Datei angibt.
- **SD_1 bis SD_8**: Datenvariablen vom Typ ANY, die in die CSV-Datei geschrieben werden.

Daten-Ausgänge

- **QO**: Boolescher Qualifier für Ausgangsereignisse.
- **STATUS**: String, der den Status des Dateizugriffs beschreibt.

Funktionsweise

Der Funktionsblock initialisiert sich beim Empfang des INIT-Ereignisses und bestätigt dies mit INITO. Beim REQ-Ereignis werden die aktuellen Werte von SD_1 bis SD_8 in die angegebene CSV-Datei geschrieben. Nach Abschluss des Schreibvorgangs wird das CNF-Ereignis ausgelöst.

Technische Besonderheiten

- Unterstützt generische Datentypen (ANY) für die Ausgabedaten.
- Bietet Statusmeldungen für erfolgreiche und fehlgeschlagene Operationen.
- Teil des eclipse4diac::utils-Pakets.

Zustandsübersicht

1. **Initialisierungszustand**: Wartet auf INIT-Ereignis.
2. **Bereit**: Nach erfolgreicher Initialisierung.
3. **Schreibvorgang**: Aktiv bei REQ-Ereignis.
4. **Bestätigung**: Sendet CNF nach Abschluss.

Anwendungsszenarien

- Protokollierung von Prozessdaten.
- Datenerfassung für Analysezwecke.
- Export von Steuerungsdaten in CSV-Format.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu einfacheren CSV-Writer-Blöcken bietet CSV_WRITER_8 die Möglichkeit, bis zu 8 verschiedene Datenwerte gleichzeitig zu schreiben und generische Datentypen zu verarbeiten.

Fazit

Der CSV_WRITER_8 ist ein leistungsfähiger Funktionsblock für die Datenerfassung und -protokollierung in CSV-Dateien. Seine Flexibilität bei der Datenverarbeitung und die detaillierten Statusmeldungen machen ihn zu einer guten Wahl für komplexe Anwendungen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

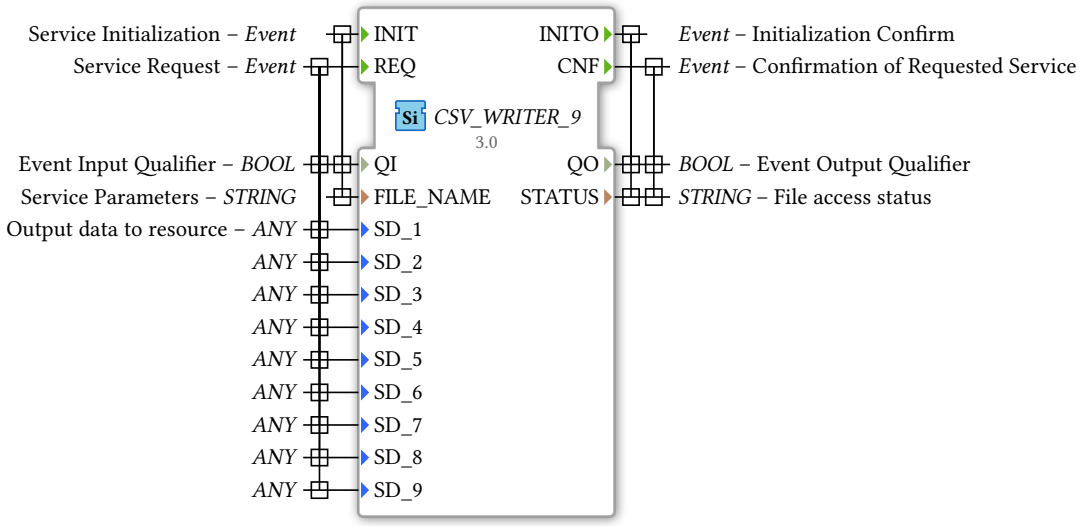
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.9.10 CSV_WRITER_9

• – ○ • – ○ • – ○ •

Einleitung

Der CSV_WRITER_9 ist ein Funktionsblock zur Erstellung von CSV-Dateien. Er ermöglicht das Schreiben von bis zu neun verschiedenen Datenpunkten in eine CSV-Datei. Der Funktionsblock ist Teil des eclipse4diac::utils-Pakets und eignet sich für Anwendungen, bei denen Daten in einem standardisierten Format gespeichert werden müssen.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisiert den Funktionsblock und legt den Dateinamen fest. Wird mit `QI` und `FILE_NAME` verwendet.
- **REQ**: Startet den Schreibvorgang in die CSV-Datei. Wird mit `QI` und den Datenpunkten `SD_1` bis `SD_9` verwendet.

Ereignis-Ausgänge

- **INITO**: Bestätigt die Initialisierung und gibt den Status zurück. Wird mit `QO` und `STATUS` verwendet.
- **CNF**: Bestätigt den erfolgreichen oder fehlgeschlagenen Schreibvorgang. Wird mit `QO` und `STATUS` verwendet.

Daten-Eingänge

- **QI**: Boolescher Qualifier für Ereigniseingänge.
- **FILE_NAME**: String, der den Namen der CSV-Datei angibt.
- **SD_1** bis **SD_9**: Datenpunkte vom Typ `ANY`, die in die CSV-Datei geschrieben werden.

Daten-Ausgänge

- **QO**: Boolescher Qualifier für Ereignisausgänge.
- **STATUS**: String, der den Status des Dateizugriffs angibt.

Adapter

Es sind keine Adapter definiert.

Funktionsweise

1. **Initialisierung**: Der Funktionsblock wird durch das `INIT`-Ereignis initialisiert. Dabei wird der Dateiname über `FILE_NAME` festgelegt.
2. **Schreibvorgang**: Das `REQ`-Ereignis löst den Schreibvorgang aus. Die Datenpunkte `SD_1` bis `SD_9` werden in die CSV-Datei geschrieben.
3. **Bestätigung**: Der Status des Vorgangs wird über `INITO` oder `CNF` zurückgemeldet.

Technische Besonderheiten

- Unterstützt bis zu neun Datenpunkte (`SD_1` bis `SD_9`) vom Typ `ANY`.
- Der Status des Dateizugriffs wird über den `STATUS`-Ausgang zurückgegeben.
- Der Funktionsblock ist Teil des eclipse4diac::utils-Pakets und unterliegt der Eclipse Public License 2.0.

Zustandsübersicht

- **Initialisierung**: Erfolgreich (`INITO+`) oder fehlgeschlagen (`INITO-`).
- **Schreibvorgang**: Erfolgreich (`CNF+`) oder fehlgeschlagen (`CNF-`).

Anwendungsszenarien

- Datenerfassung und -speicherung in industriellen Steuerungssystemen.
- Protokollierung von Prozessdaten in einer standardisierten CSV-Datei.
- Integration in größere Automatisierungslösungen zur Datenverarbeitung.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu einfacheren CSV-Writer-Funktionsblöcken bietet der CSV_WRITER_9 die Möglichkeit, bis zu neun Datenpunkte gleichzeitig zu schreiben.
- Andere Funktionsblöcke könnten weniger Flexibilität bei den Datentypen bieten, da `ANY` verwendet wird.

Fazit

Der CSV_WRITER_9 ist ein leistungsfähiger Funktionsblock für die Erstellung von CSV-Dateien in 4diac-IDE. Seine Flexibilität bei der Datenaufnahme und die klare Statusrückmeldung machen ihn zu einer guten Wahl für Anwendungen, die eine strukturierte Datenspeicherung erfordern.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

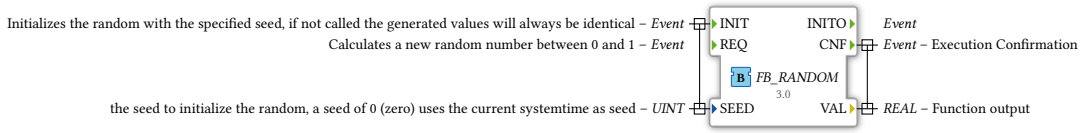
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.9.11 FB_RANDOM

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock FB_RANDOM generiert eine zufällige REAL-Zahl zwischen 0 und 1.



Schnittstellenstruktur

Ereignis-Eingänge

- **INIT**: Initialisiert den Zufallsgenerator mit einem Seed.
- **REQ**: Berechnet eine neue Zufallszahl.

Ereignis-Ausgänge

- **INITO**: Initialisierung abgeschlossen.
- **CNF**: Berechnung abgeschlossen, neuer Wert verfügbar.

Daten-Eingänge

- **SEED** (UINT): Der Startwert (Seed) für den Zufallsgenerator. Ein Wert von 0 verwendet die aktuelle Systemzeit.

Daten-Ausgänge

- **VAL** (REAL): Die generierte Zufallszahl.

Metadaten

Attribut	Wert
Copyright	(c) 2012 Profactor GmbH
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	eclipse4diac::utils

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

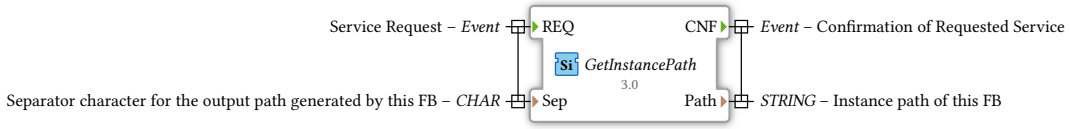
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.9.12 GetInstancePath

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `GetInstancePath` dient dazu, den Instanzpfad des eigenen Funktionsblocks innerhalb der 4diac-IDE zu ermitteln. Dieser Pfad kann beispielsweise für Diagnosezwecke oder zur dynamischen Konfiguration verwendet werden. Der FB ist besonders nützlich in komplexen Anwendungen, wo eine genaue Identifikation der Instanzposition erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Service-Anfrage. Löst die Ermittlung des Instanzpfads aus.

Ereignis-Ausgänge

- **CNF**: Bestätigung der Service-Anfrage. Signalisiert die erfolgreiche Ermittlung des Instanzpfads.

Daten-Eingänge

- **Sep**: Separator-Zeichen (CHAR) für den generierten Pfad. Standardwert ist `'/'`.

Daten-Ausgänge

- **Path**: Der ermittelte Instanzpfad als STRING.

Adapter

Keine Adapter vorhanden.

Funktionsweise

1. Bei Empfang des `REQ`-Ereignisses beginnt der FB mit der Ermittlung des eigenen Instanzpfads.
2. Der Pfad wird mit dem angegebenen Separatorzeichen (`Sep`) formatiert.
3. Das `CNF`-Ereignis wird ausgelöst und der ermittelte Pfad über den `Path`-Ausgang ausgegeben.

Technische Besonderheiten

- Der FB nutzt eine Service-Schnittstelle zur Kommunikation mit der Ressourcenebene.
- Der Standard-Separator ist `'/'`, kann jedoch durch den Benutzer angepasst werden.

Zustandsübersicht

- **Idle**: Wartet auf eine `REQ`-Anfrage.
- **Processing**: Ermittelt den Instanzpfad und gibt ihn über `CNF` aus.

Anwendungsszenarien

- Diagnose und Logging: Identifikation der genauen Position einer FB-Instanz in einer komplexen Anwendung.
- Dynamische Konfiguration: Verwendung des Pfads zur dynamischen Anpassung von Parametern oder Verhaltensweisen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu statischen Pfadangaben ermöglicht `GetInstancePath` eine dynamische Ermittlung des Pfads, was besonders in wiederverwendbaren oder verschachtelten Komponenten vorteilhaft ist.

Fazit

Der `GetInstancePath`-FB ist ein nützliches Werkzeug zur dynamischen Ermittlung des eigenen Instanzpfads in 4diac-Anwendungen. Seine einfache Schnittstelle und flexible Konfiguration machen ihn ideal für Diagnose- und Konfigurationszwecke.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

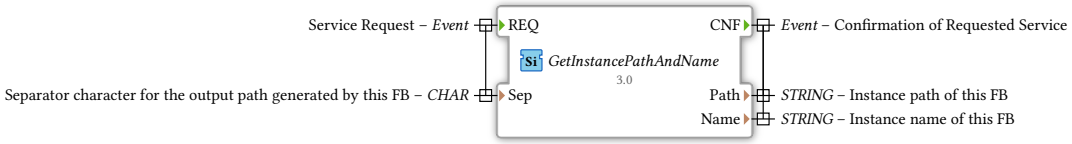
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.9.13 GetInstancePathAndName

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `GetInstancePathAndName` dient dazu, den Instanzpfad und den Instanznamen aus der Laufzeitumgebung abzurufen. Dies ist besonders nützlich für Debugging-Zwecke oder für dynamische Konfigurationen, bei denen der genaue Pfad einer Instanz benötigt wird.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Service-Anfrage, die den Abruf des Instanzpfades und -namens auslöst. Wird mit dem Separatorzeichen `Sep` verknüpft.

Ereignis-Ausgänge

- **CNF**: Bestätigung der angeforderten Service-Anfrage. Wird mit den Ausgabevariablen `Path` (Instanzpfad) und `Name` (Instanzname) verknüpft.

Daten-Eingänge

- **Sep** (CHAR): Separatorzeichen für den generierten Ausgabepfad. Standardwert ist `'/'`.

Daten-Ausgänge

- **Path** (STRING): Der Instanzpfad des Funktionsblocks.
- **Name** (STRING): Der Instanzname des Funktionsblocks.

Adapter

Keine Adapter vorhanden.

Funktionsweise

1. Bei Empfang eines `REQ`-Ereignisses wird der Funktionsblock aktiviert.
2. Der Funktionsblock ermittelt den aktuellen Instanzpfad und den Instanznamen aus der Laufzeitumgebung.
3. Der Instanzpfad wird mit dem angegebenen Separatorzeichen `Sep` formatiert.
4. Das `CNF`-Ereignis wird ausgelöst und die ermittelten Werte für `Path` und `Name` ausgegeben.

Technische Besonderheiten

- Der Standard-Separator für den Pfad ist `'/'`, kann jedoch über den Eingang `Sep` angepasst werden.
- Die Ausgabevariablen `Path` und `Name` sind vom Typ `STRING` und enthalten den vollständigen Pfad bzw. den Namen der Instanz.

Zustandsübersicht

Der Funktionsblock hat keinen internen Zustand. Er reagiert ausschließlich auf das `REQ`-Ereignis und gibt sofort die entsprechenden Daten aus.

Anwendungsszenarien

- **Debugging**: Ermittlung des genauen Pfades einer Instanz zur Fehlersuche.
- **Dynamische Konfiguration**: Verwendung des Instanzpfades für dynamische Konfigurationen oder Logging.
- **Systemüberwachung**: Überwachung und Protokollierung von Instanzpfaden in komplexen Systemen.

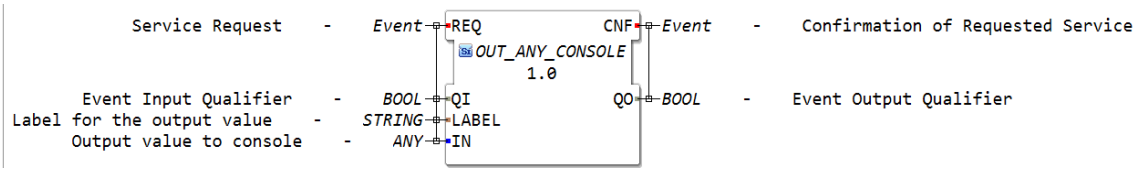
⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Funktionsblöcken, die ähnliche Informationen liefern, bietet `GetInstancePathAndName` die Flexibilität, den Separator für den Pfad anzupassen.
- Andere Bausteine könnten nur den Pfad oder nur den Namen liefern, während dieser Funktionsblock beides kombiniert.

Fazit

Der `GetInstancePathAndName`-Funktionsblock ist ein einfaches, aber mächtiges Werkzeug, um Instanzinformationen aus der Laufzeitumgebung abzurufen. Seine Flexibilität und Einfachheit machen ihn ideal für Debugging und dynamische Konfigurationen.

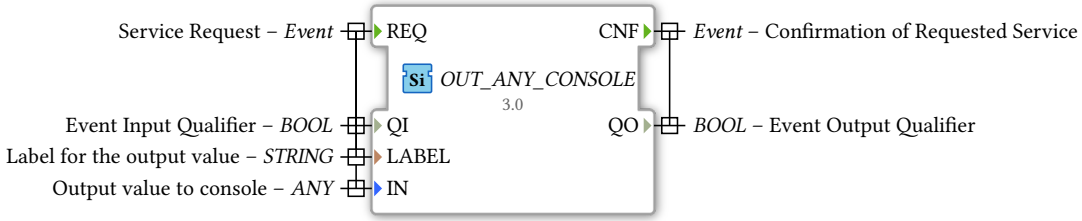
2.4.9.14 OUT_ANY_CONSOLE



• — ◦ • — ◦ • — ◦ •

Einleitung

Der **OUT_ANY_CONSOLE**-Funktionsbaustein ist ein universelles Debugging-Tool zur Ausgabe von Variablenwerten auf der Systemkonsole, entwickelt unter EPL-2.0 Lizenz (Version 3.0). Der Baustein unterstützt die Ausgabe beliebiger Datentypen für Entwicklungs- und Diagnosezwecke.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Auslöser für Konsolenausgabe (mit Qualifier `QI`, Wert `IN` und optionalem `LABEL`)

Ereignis-Ausgänge

- `CNF`: Bestätigung der Ausgabeoperation

Daten-Eingänge

Parameter	Typ	Beschreibung
`QI`	BOOL	Aktiviert die Ausgabe bei TRUE
`LABEL`	STRING	Optionaler Beschreibungstext
`IN`	ANY	Beliebiger auszugebender Wert

Daten-Ausgänge

Parameter	Typ	Beschreibung
QO	BOOL	Bestätigt erfolgreiche Ausgabe

Funktionsweise

1. **Auslösung**:
   - REQ-Ereignis mit QI=TRUE startet Ausgabe
   - Kombination aus LABEL und IN wird formatiert
2. **Ausführung**:
   - Wert wird auf Standardausgabe geschrieben
   - Automatische Typkonvertierung für ANY
3. **Bestätigung**:
   - CNF mit QO=TRUE bei Erfolg
   - CNF mit QO=FALSE bei Fehlern

Technische Besonderheiten

✓ **Universelle Typunterstützung** (ANY-Datentyp) ✓ **Flexible Beschriftung** mit LABEL-Parameter ✓ **Eclipse 4diac Integration** (Package: eclipse4diac::utils) ✓ **Plattformunabhängig** für verschiedene Zielsysteme

Anwendungsszenarien

- **Debugging**: Variableninspektion zur Laufzeit
- **Protokollierung**: Ereignisverfolgung
- **Systemdiagnose**: Zustandsüberwachung
- **Entwicklung**: Schnelles Testen von Algorithmen

Beispielausgaben

LABEL	IN (Wert)	Konsolenausgabe
„Temperatur“	23.5	„Temperatur: 23.5“
„Status“	TRUE	„Status: TRUE“
““	42	„42“

⚖️ **Vergleich mit ähnlichen Bausteinen**

Feature	OUT_ANY_CONSOLE	Standard-Debug	Erweitertes Logging
Datentypen	Beliebig (ANY)	Begrenzt	Begrenzt
Beschriftung	Optional	Nein	Ja
Integration	4diac-spezifisch	Systemabhängig	Framework-abhängig
Echtzeitfähig	Ja	Teilweise	Nein

Fazit

Der OUT_ANY_CONSOLE-Baustein ist ein unverzichtbares Werkzeug für die Entwicklung:

- **Flexibel**: Unterstützt alle Datentypen
- **Einfach**: Schnelle Integration in bestehende Systeme
- **Leistungsstark**: Für Echtzeit-Debugging geeignet

Idealer Einsatz bei:

- Prototypenentwicklung
- Systemtests

- Fehleranalyse
- Live-Diagnose

Das Schweizer Taschenmesser für Entwickler in Eclipse 4diac-Umgebungen

---

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

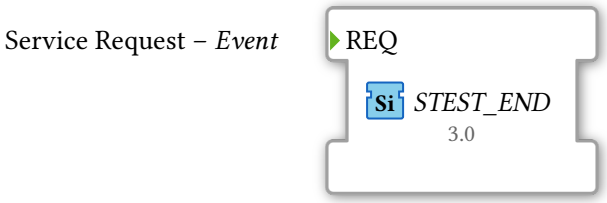
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.9.15 STEST_END

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock STEST_END ist ein Service Interface Function Block, der als Endpunkt für Service-Anfragen dient. Er wird typischerweise in Verbindung mit anderen Service-Funktionsblöcken verwendet, um eine Service-orientierte Architektur in 4diac-IDE zu implementieren.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Service Request - Wird ausgelöst, um eine Service-Anfrage zu starten.

Ereignis-Ausgänge

Keine Ereignis-Ausgänge vorhanden.

Daten-Eingänge

Keine Daten-Eingänge vorhanden.

Daten-Ausgänge

Keine Daten-Ausgänge vorhanden.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der STEST_END Funktionsblock dient als Endpunkt für Service-Anfragen. Wenn der REQ-Eingang ausgelöst wird, wird die Service-Anfrage verarbeitet. Da es sich um einen Endpunkt handelt, gibt es keine weiteren Ereignis- oder Datenausgänge.

Technische Besonderheiten

- Implementiert als Service Interface Function Block.
- Unterstützt die Kommunikation zwischen RESOURCE und APPLICATION Schnittstellen.

Zustandsübersicht

Der Funktionsblock hat keine komplexen Zustände. Er reagiert einfach auf die REQ-Anfrage und verarbeitet diese.

Anwendungsszenarien

- Als Endpunkt in einer Service-orientierten Architektur.
- In Testumgebungen, um Service-Anfragen zu simulieren oder zu beenden.

⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zu anderen Service-Funktionsblöcken hat STEST_END keine Ausgänge oder zusätzliche Schnittstellen, was ihn ideal als Endpunkt macht.

Fazit

Der STEST_END Funktionsblock ist ein einfacher, aber essentieller Baustein für Service-orientierte Architekturen in 4diac-IDE. Seine Einfachheit macht ihn zu einem zuverlässigen Endpunkt für Service-Anfragen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

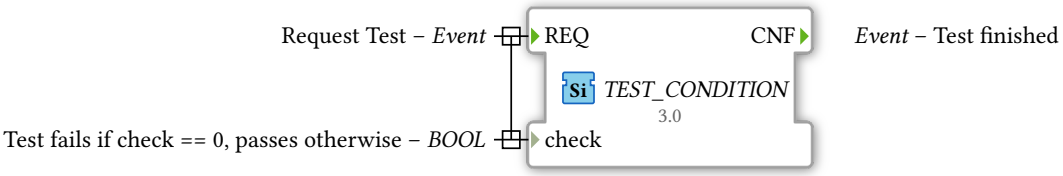
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.4.9.16 TEST_CONDITION

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `TEST_CONDITION` dient zur Überprüfung einer Bedingung und signalisiert, ob diese erfüllt ist oder nicht. Er wird typischerweise in Testumgebungen eingesetzt, um bestimmte Bedingungen zu validieren.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Überprüfung der Bedingung. Wird dieser Eingang ausgelöst, beginnt der Funktionsblock mit der Auswertung des Daten-Eingangs `check`.

Ereignis-Ausgänge

- **CNF**: Signalisiert den Abschluss der Überprüfung. Dieses Ereignis wird ausgelöst, nachdem die Bedingung ausgewertet wurde.

Daten-Eingänge

- **check**: Ein boolescher Wert (`BOOL`), der die zu überprüfende Bedingung darstellt.
  - Wenn `check == TRUE`, gilt die Bedingung als erfüllt (Test bestanden).
  - Wenn `check == FALSE`, gilt die Bedingung als nicht erfüllt (Test nicht bestanden).

Daten-Ausgänge

- Keine vorhanden.

Adapter

- Keine vorhanden.

Funktionsweise

1. Der Funktionsblock wird durch das Ereignis `REQ` aktiviert.
2. Der Wert des Daten-Eingangs `check` wird ausgewertet.
3. Das Ergebnis der Auswertung wird intern verarbeitet (keine direkte Rückmeldung über Daten-Ausgänge).
4. Das Ereignis `CNF` wird ausgelöst, um den Abschluss der Überprüfung zu signalisieren.

Technische Besonderheiten

- Der Funktionsblock ist einfach aufgebaut und enthält keine komplexen Zustandsautomaten.

- Es werden keine Daten-Ausgänge bereitgestellt, da das Ergebnis der Überprüfung nur intern verarbeitet wird.

Zustandsübersicht

Der Funktionsblock hat einen einfachen Ablauf:

1. **Idle-Zustand**: Wartet auf das `REQ`-Ereignis.
2. **Auswertungszustand**: Wertet `check` aus und löst `CNF` aus.
3. Rückkehr zum **Idle-Zustand**.

Anwendungsszenarien

- **Testautomatisierung**: Überprüfung von Bedingungen in automatisierten Testabläufen.
- **Fehlererkennung**: Validierung von Systemzuständen oder Eingabedaten.
- **Prototypische Implementierung**: Schnelle Integration in Testumgebungen zur Validierung von Logiken.

⚖️ Vergleich mit ähnlichen Bausteinen

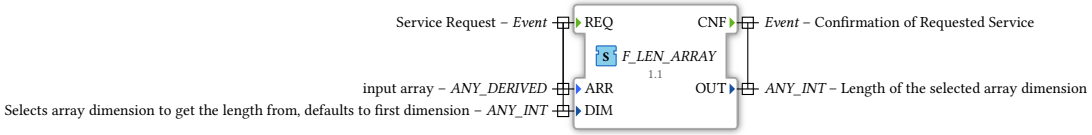
- Im Vergleich zu anderen Testbausteinen bietet `TEST_CONDITION` eine minimalistische Schnittstelle und ist auf die Auswertung einer einzelnen Bedingung spezialisiert.
- Komplexere Testbausteine könnten zusätzliche Funktionen wie Zeitüberwachung oder mehrere Bedingungen unterstützen.

Fazit

Der `TEST_CONDITION`-Funktionsblock ist ein einfaches, aber effektives Werkzeug zur Überprüfung von Bedingungen in Testumgebungen. Seine schlanke Schnittstelle macht ihn leicht integrierbar, während seine Funktionalität für grundlegende Validierungen ausreicht. Für komplexere Anforderungen sollten jedoch erweiterte Testbausteine in Betracht gezogen werden.

2.4.9.17 Arrays

F_LEN_ARRAY



• – ○ • – ○ • – ○ •

Einleitung

Der `F_LEN_ARRAY` ist ein einfacher Funktionsbaustein, der die Länge (Anzahl der Elemente) einer gewählten Dimension eines Arrays beliebigen Typs zurückgibt. Er ergänzt die Standardfunktionen `UPPER_BOUND`/`LOWER_BOUND` um eine direkt nutzbare Längenberechnung.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Löst die Berechnung aus, trägt `ARR` und `DIM`.

Ereignis-Ausgänge

- **CNF**: Bestätigt den Abschluss, trägt `OUT`.

Daten-Eingänge

- **ARR** (ANY_DERIVED): Das Array, dessen Länge ermittelt werden soll.
- **DIM** (ANY_INT, Default `DINT#1`): Die zu betrachtende Array-Dimension (bei mehrdimensionalen Arrays).

Daten-Ausgänge

- **OUT** (ANY_INT): Die Länge der gewählten Dimension, also Anzahl der Elemente.

Funktionsweise

Der Baustein ist als `SimpleFB` mit einem einzelnen ECC-Zustand implementiert. Bei `REQ` berechnet die Algorithmus-ST-Anweisung `OUT := ADD(SUB(UPPER_BOUND(ARR, DIM), LOWER_BOUND(ARR, DIM)), 1)` — also die obere minus die untere Indexgrenze der gewählten Dimension, plus 1 (da beide Grenzen inklusive sind). Anschließend wird `CNF` ausgelöst.

Technische Besonderheiten

- **`ANY_DERIVED`-Eingang**: `ARR` akzeptiert Arrays beliebigen Elementtyps und beliebiger Dimensionalität.
- **Dimensionsauswahl über `DIM`**: Bei mehrdimensionalen Arrays kann über `DIM` gezielt die Länge einer bestimmten Dimension abgefragt werden; Standardwert ist die erste Dimension (`DINT#1`).
- **Basiert auf `UPPER_BOUND`/`LOWER_BOUND`**: Der Baustein ist eine direkte Kombination dieser beiden IEC-61131-3-Standardfunktionen und spart deren manuelle Verrechnung.

Zustandsübersicht

Zustandslos: jedes `REQ` führt unmittelbar zur Berechnung und zu `CNF`.

Anwendungsszenarien

- **Dynamische Array-Verarbeitung**: Schleifen über ein Array, dessen tatsächliche Größe erst zur Laufzeit (z. B. abhängig vom Aufrufer) bekannt ist.
- **Plausibilitätsprüfung**: Vergleich der tatsächlichen Array-Länge mit einer erwarteten Größe, bevor auf einzelne Elemente zugegriffen wird.

⚖️ Vergleich mit ähnlichen Bausteinen

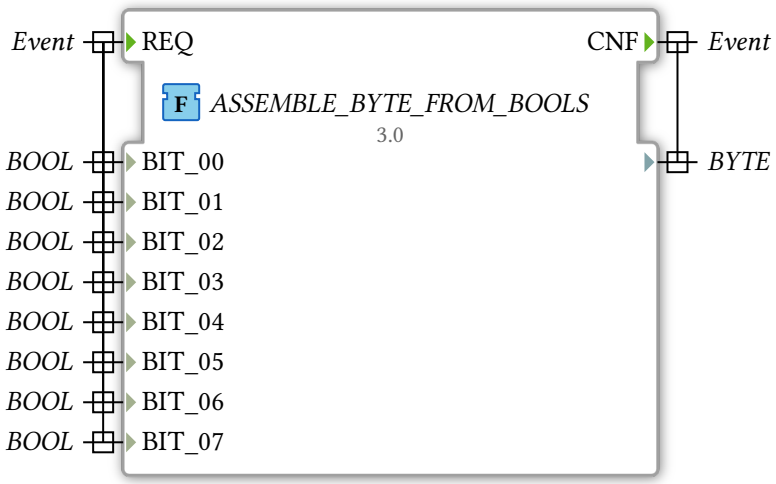
- **`UPPER_BOUND`/`LOWER_BOUND`**: die beiden Standardfunktionen, aus denen `F_LEN_ARRAY` seine Berechnung zusammensetzt.

Fazit

`F_LEN_ARRAY` liefert eine einfache, direkt nutzbare Längenermittlung für Arrays beliebigen Typs und einer wählbaren Dimension, ohne dass `UPPER_BOUND`/`LOWER_BOUND` manuell miteinander verrechnet werden müssen.

2.4.9.18 assembling

ASSEMBLE_BYTE_FROM_BOOLS



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ASSEMBLE_BYTE_FROM_BOOLS` kombiniert 8 boolesche Eingangswerte zu einem Byte. Jeder boolesche Eingang repräsentiert dabei ein Bit des resultierenden Bytes. Dieser Baustein ist besonders nützlich, wenn einzelne boolesche Signale zu einem kompakten Byte-Wert zusammengefasst werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge
- `REQ`: Startet die Verarbeitung der booleschen Eingänge und erzeugt das resultierende Byte.

Ereignis-Ausgänge
- `CNF`: Signalisiert die erfolgreiche Zusammenfassung der booleschen Werte zu einem Byte.

Daten-Eingänge
- `BIT_00` bis `BIT_07`: Acht boolesche Eingänge, die jeweils einem Bit des resultierenden Bytes entsprechen.

Daten-Ausgänge
- `BYTE`: Das zusammengesetzte Byte, das aus den 8 booleschen Eingängen generiert wird.

Adapter
Keine Adapter vorhanden.

Funktionsweise
Der Funktionsblock setzt die 8 booleschen Eingänge (`BIT_00` bis `BIT_07`) zu einem Byte zusammen. Jeder boolesche Wert wird auf das entsprechende Bit des Bytes abgebildet:

- `BIT_00` entspricht Bit 0 (LSB)
- `BIT_01` entspricht Bit 1
- ...
- `BIT_07` entspricht Bit 7 (MSB)

Bei Auslösung des Ereignisses `REQ` werden die aktuellen Werte der booleschen Eingänge gelesen und zu einem Byte kombiniert. Das Ergebnis wird am Datenausgang ausgegeben und das Ereignis `CNF` signalisiert die erfolgreiche Verarbeitung.

Technische Besonderheiten
- Der Funktionsblock ist in ST (Structured Text) implementiert.
- Die Bit-Zuordnung erfolgt direkt über die Zuweisung der booleschen Werte zu den entsprechenden Bits des Bytes.

Zustandsübersicht
1. **Idle-Zustand**: Wartet auf das `REQ`-Ereignis.
2. **Verarbeitungszustand**: Liest die booleschen Eingänge und setzt das Byte zusammen.
3. **Bestätigungszustand**: Gibt das resultierende Byte aus und signalisiert `CNF`.

Anwendungsszenarien
- Zusammenfassen von 8 einzelnen Schalterstellungen zu einem Byte für die Kommunikation mit anderen Systemen.
- Komprimierung von booleschen Statusinformationen für effizientere Datenübertragung.
- Verwendung in Steuerungssystemen, wo mehrere boolesche Signale als ein Byte verarbeitet werden müssen.

⚖️ Vergleich mit ähnlichen Bausteinen
- Im Vergleich zu manuellen Bit-Operationen in ST oder anderen Sprachen bietet dieser Baustein eine vordefinierte und wartungsfreundliche Lösung.
- Ähnliche Bausteine könnten weniger Eingänge haben oder zusätzliche Features wie invertierte Logik bieten, was hier nicht der Fall ist.

🛠️ Zugehörige Übungen
- Uebung_035a1b_AX
- Uebung_053

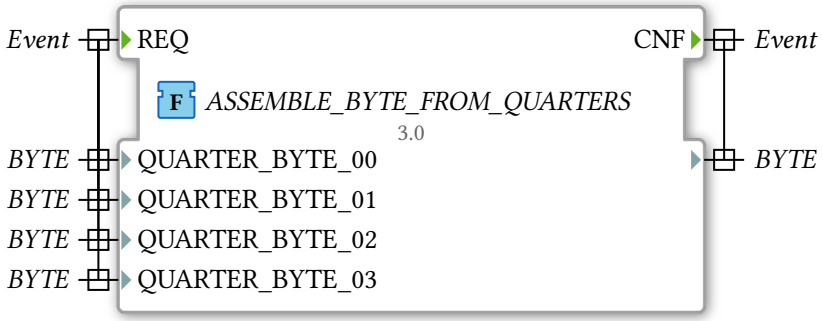
Fazit

Der ASSEMBLE_BYTE_FROM_BOOLS-Funktionsblock ist eine einfache und effiziente Lösung, um 8 boolesche Werte zu einem Byte zu kombinieren. Durch seine klare Schnittstelle und direkte Implementierung eignet er sich ideal für Anwendungen, bei denen boolesche Signale kompakt verarbeitet werden müssen.

ASSEMBLE_BYTE_FROM_QUARTERS

🎧 Podcast

- QUARTER



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock ASSEMBLE_BYTE_FROM_QUARTERS kombiniert vier einzelne Quarter-Bytes zu einem vollständigen Byte. Dies ist besonders nützlich in Szenarien, wo Daten in Teilen übertragen oder verarbeitet werden müssen und anschließend wieder zusammengesetzt werden sollen.

Schnittstellenstruktur

Ereignis-Eingänge

- REQ: Startet die Kombination der Quarter-Bytes zu einem Byte. Wird mit den vier Quarter-Bytes (QUARTER_BYTE_00 bis QUARTER_BYTE_03) verknüpft.

Ereignis-Ausgänge

- CNF: Signalisiert die erfolgreiche Zusammenführung der Quarter-Bytes zu einem Byte.

Daten-Eingänge

- QUARTER_BYTE_00: Erstes Quarter-Byte (niedrigstwertige Bits).
- QUARTER_BYTE_01: Zweites Quarter-Byte.
- QUARTER_BYTE_02: Drittes Quarter-Byte.
- QUARTER_BYTE_03: Viertes Quarter-Byte (höchstwertige Bits).

Daten-Ausgänge

- BYTE: Das zusammengesetzte Byte als Ergebnis der Kombination der vier Quarter-Bytes.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock nimmt vier Quarter-Bytes entgegen und kombiniert sie zu einem vollständigen Byte. Dabei werden die einzelnen Quarter-Bytes mittels Bit-Shift-Operationen (SHL) an die richtige Position im resultierenden Byte verschoben und durch logische ODER-Verknüpfungen (OR) zusammengefügt. Die genauen Verschiebungen werden durch Konstanten (quarterconst::SHIFT_QUARTER_00 bis quarterconst::SHIFT_QUARTER_03) definiert.

Technische Besonderheiten

- Verwendet Bit-Shift-Operationen (`SHL`) und logische ODER-Verknüpfungen (`OR`) zur Kombination der Quarter-Bytes.
- Die Verschiebungspositionen sind als Konstanten definiert, was eine flexible Anpassung ermöglicht.

Zustandsübersicht

1. **Initialisierung**: Der Funktionsblock wird initialisiert und wartet auf das `REQ`-Ereignis.
2. **Verarbeitung**: Bei Empfang von `REQ` werden die vier Quarter-Bytes verarbeitet und zum resultierenden Byte kombiniert.
3. **Bestätigung**: Nach erfolgreicher Kombination wird das `CNF`-Ereignis ausgelöst und das resultierende Byte ausgegeben.

Anwendungsszenarien

- **Datenkommunikation**: Kombination von in Teilen übertragenen Daten zu einem vollständigen Byte.
- **Datenverarbeitung**: Verarbeitung von Daten, die in kleineren Einheiten vorliegen, aber als Ganzes benötigt werden.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Bit-Manipulations-Funktionsblöcken bietet `ASSEMBLE_BYTE_FROM_QUARTERS` eine spezialisierte und optimierte Lösung für die Kombination von Quarter-Bytes.
- Andere Blöcke könnten ähnliche Funktionalität bieten, sind jedoch oft weniger spezifisch und erfordern mehr Konfiguration.

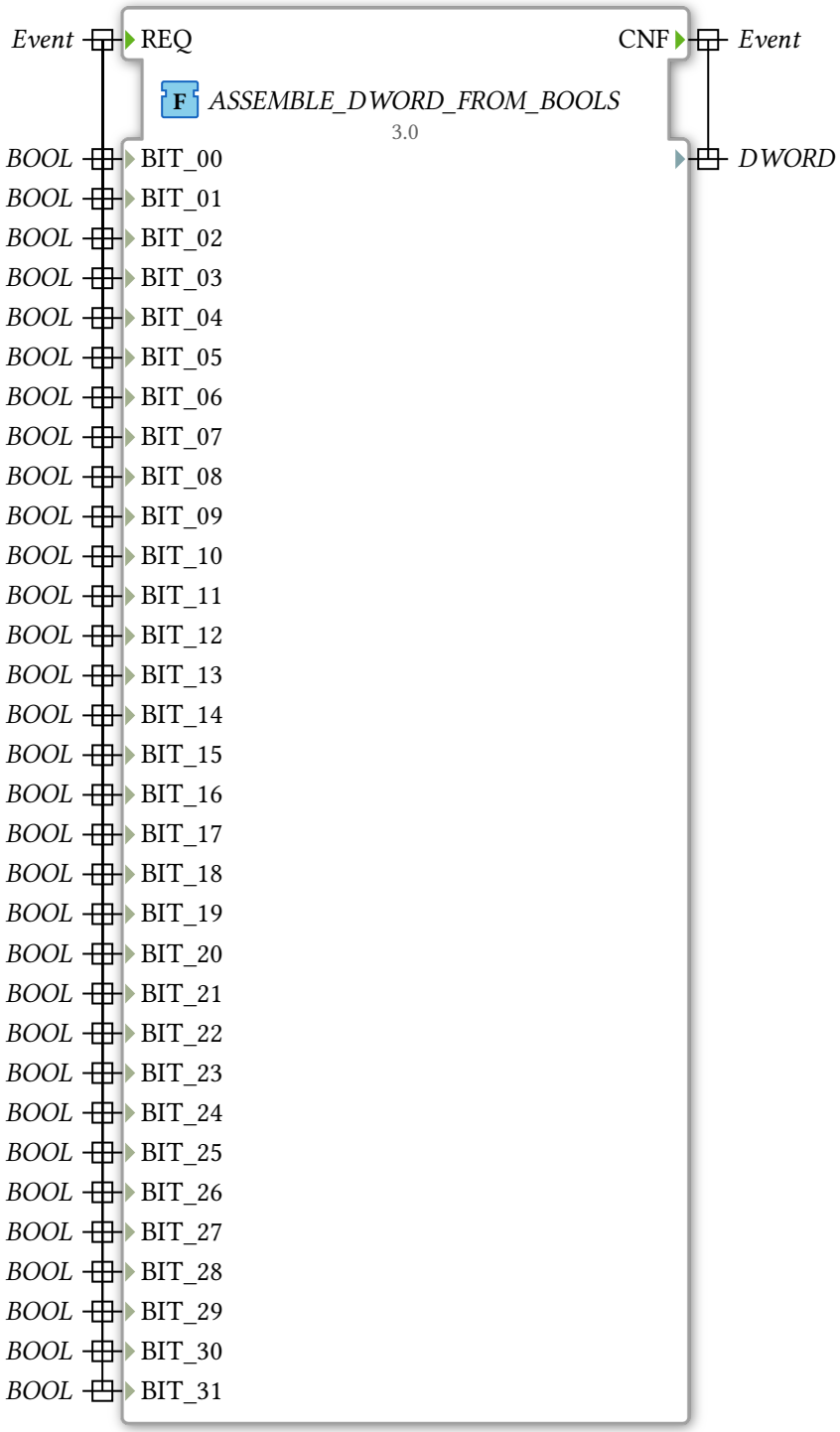
🛠️ Zugehörige Übungen

- Uebung_056

Fazit

Der `ASSEMBLE_BYTE_FROM_QUARTERS`-Funktionsblock ist eine effiziente und spezialisierte Lösung für die Kombination von vier Quarter-Bytes zu einem vollständigen Byte. Durch die Verwendung von Bit-Shift- und logischen Operationen bietet er eine performante und flexible Möglichkeit zur Datenverarbeitung in entsprechenden Anwendungsszenarien.

ASSEMBLE_DWORD_FROM_BOOLS



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ASSEMBLE_DWORD_FROM_BOOLS` kombiniert 32 einzelne BOOL-Werte zu einem einzigen DWORD-Wert. Dies ist besonders nützlich, wenn mehrere binäre Zustände in einer kompakten Form verarbeitet oder übertragen werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Auslöseereignis, das die Verarbeitung der Eingangsdaten startet. Wird mit allen 32 BIT-Eingängen verknüpft.

Ereignis-Ausgänge

- `CNF`: Bestätigungsereignis, das nach erfolgreicher Kombination der BOOL-Werte zu einem DWORD ausgegeben wird.

Daten-Eingänge

32 BOOL-Eingänge (`BIT_00` bis `BIT_31`), die jeweils ein Bit im resultierenden DWORD repräsentieren:

- `BIT_00`: Bit 0 (niederwertigstes Bit)
- `BIT_01`: Bit 1
- ...
- `BIT_31`: Bit 31 (höchstwertigstes Bit)

Daten-Ausgänge

- `OUT`: DWORD, das aus den 32 Eingangs-BOOLs zusammengesetzt wird.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock setzt bei Auslösung des `REQ`-Ereignisses die 32 BOOL-Eingänge zu einem DWORD zusammen. Jeder BOOL-Eingang entspricht einem Bit im DWORD:

- `BIT_00` wird zu Bit 0 (LSB)
- `BIT_31` wird zu Bit 31 (MSB)

Nach der erfolgreichen Kombination wird das `CNF`-Ereignis ausgegeben.

Technische Besonderheiten

- Der Funktionsblock ist in ST (Structured Text) implementiert.
- Die Bit-Zuordnung erfolgt direkt über die Zuweisung der BOOL-Werte zu den entsprechenden Bit-Positionen des DWORD.
- Keine Pufferung oder Zustandsspeicherung: Die Verarbeitung erfolgt bei jedem `REQ`-Ereignis neu.

Zustandsübersicht

Der Funktionsblock hat keinen internen Zustand. Die Verarbeitung erfolgt rein ereignisgesteuert:

1. `REQ` empfangen ⟶ Verarbeitung starten.
2. BOOL-Werte lesen und zu DWORD kombinieren.
3. `CNF` ausgeben.

Anwendungsszenarien

- Komprimierung mehrerer binärer Signale für die Übertragung über Kommunikationsprotokolle.
- Speicherung mehrerer Schaltzustände in einem einzigen Register.
- Effiziente Verarbeitung von Bitmasken in Steuerungsalgorithmen.

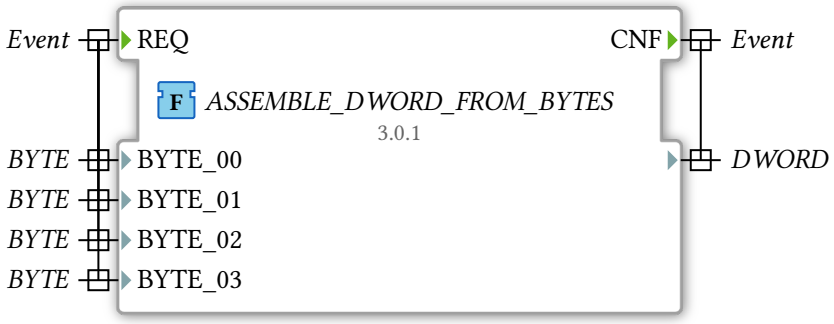
⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu manuellen Bit-Operationen in ST oder anderen FB-Typen bietet dieser Baustein eine vordefinierte, fehlerfreie Lösung für die Kombination von BOOLs zu einem DWORD.
- Ähnliche Bausteine könnten weniger Bits verarbeiten oder zusätzliche Funktionen wie Bit-Verschiebungen bieten.

Fazit

Der `ASSEMBLE_DWORD_FROM_BOOLS`-Funktionsblock ist eine effiziente und zuverlässige Lösung, um 32 binäre Eingänge in einem einzigen DWORD zu kombinieren. Durch seine einfache und klare Schnittstelle eignet er sich ideal für Anwendungen, die kompakte Darstellungen mehrerer binärer Zustände erfordern.

ASSEMBLE_DWORD_FROM_BYTES



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ASSEMBLE_DWORD_FROM_BYTES` kombiniert vier BYTE-Werte zu einem einzigen DWORD-Wert. Dies ist besonders nützlich, wenn Daten aus mehreren Quellen zusammengeführt werden müssen, um sie als eine einzige Einheit zu verarbeiten oder zu speichern.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Kombination der BYTE-Werte zu einem DWORD. Wird mit den Daten-Eingängen `BYTE_00`, `BYTE_01`, `BYTE_02` und `BYTE_03` verknüpft.

Ereignis-Ausgänge

- **CNF**: Signalisiert die erfolgreiche Kombination der BYTE-Werte zu einem DWORD. Der kombinierte DWORD-Wert ist am Daten-Ausgang verfügbar.

Daten-Eingänge

- **BYTE_00**: Erstes Byte des zu kombinierenden DWORD.
- **BYTE_01**: Zweites Byte des zu kombinierenden DWORD.
- **BYTE_02**: Drittes Byte des zu kombinierenden DWORD.
- **BYTE_03**: Viertes Byte des zu kombinierenden DWORD.

Daten-Ausgänge

- **(unnamed)**: Der kombinierte DWORD-Wert, der aus den vier Eingabe-Bytes erstellt wurde.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock nimmt vier BYTE-Werte entgegen und kombiniert sie zu einem einzigen DWORD-Wert. Die Kombination erfolgt durch direkte Zuweisung der einzelnen Bytes zu den entsprechenden Positionen im DWORD:

- `BYTE_00` wird dem niedrigstwertigen Byte (B0) des DWORD zugewiesen.
- `BYTE_01` wird dem nächsten Byte (B1) zugewiesen.
- `BYTE_02` wird dem nächsten Byte (B2) zugewiesen.
- `BYTE_03` wird dem höchstwertigen Byte (B3) zugewiesen.

Technische Besonderheiten

- Der Funktionsblock ist in der ST (Structured Text) Sprache implementiert.
- Die Zuweisung der Bytes erfolgt direkt über die `%B0` bis `%B3` Syntax, die auf die einzelnen Bytes des DWORD zugreift.

Zustandsübersicht

1. **Idle**: Wartet auf das `REQ`-Ereignis.

2. **Processing**: Bei Empfang von `REQ` werden die Eingabe-Bytes kombiniert und der DWORD-Wert berechnet.
3. **Completed**: Sendet das `CNF`-Ereignis mit dem kombinierten DWORD-Wert.

Anwendungsszenarien

- Kombination von Daten aus mehreren Sensoren oder Quellen zu einem einzigen Wert.
- Vorverarbeitung von Daten für die Übertragung oder Speicherung.
- Verwendung in Protokollimplementierungen, wo Daten in Bytes segmentiert und später wieder zusammengesetzt werden müssen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke könnten einzelne Bytes zu anderen Datentypen (z.B. WORD) kombinieren.
- Im Gegensatz zu manuellen Kombinationen in ST-Code bietet dieser Block eine vordefinierte und wiederverwendbare Lösung.

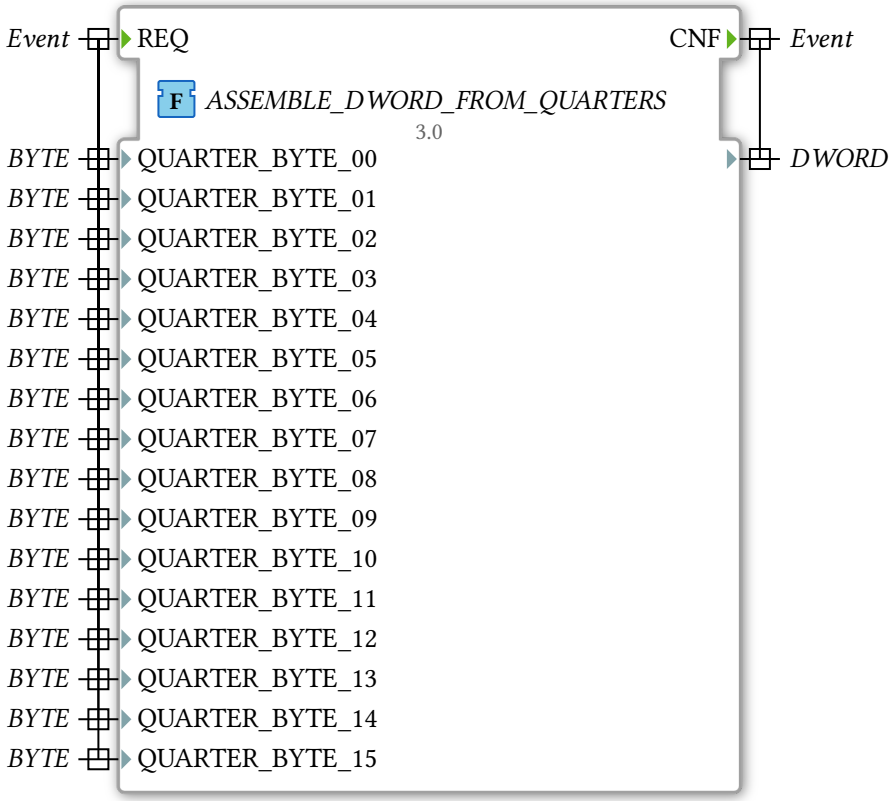
Fazit

Der `ASSEMBLE_DWORD_FROM_BYTES`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, vier BYTE-Werte zu einem DWORD zu kombinieren. Seine klare Schnittstelle und direkte Implementierung machen ihn zu einem nützlichen Werkzeug in verschiedenen Anwendungsfällen.

ASSEMBLE_DWORD_FROM_QUARTERS

🎧 Podcast

- QUARTER



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ASSEMBLE_DWORD_FROM_QUARTERS` kombiniert 16 Quarter-Bytes (je 4 Bit) zu einem DWORD (32 Bit). Dies ist besonders nützlich, wenn Daten in kleineren Einheiten vorliegen und zu einer größeren Einheit zusammengefasst werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge
- `REQ`: Startet die Kombination der Quarter-Bytes zu einem DWORD. Wird mit allen 16 Quarter-Bytes verknüpft.

Ereignis-Ausgänge
- `CNF`: Signalisiert die erfolgreiche Kombination der Quarter-Bytes zu einem DWORD.

Daten-Eingänge
- `QUARTER_BYTE_00` bis `QUARTER_BYTE_15`: 16 Eingänge vom Typ `BYTE`, die jeweils ein Quarter-Byte (4 Bit) enthalten.

Daten-Ausgänge
- (Unbenannt): Ein Ausgang vom Typ `DWORD`, der das kombinierte Ergebnis enthält.

Adapter
Keine Adapter vorhanden.

Funktionsweise
Der Funktionsblock nimmt 16 Quarter-Bytes entgegen und kombiniert sie zu einem 32-Bit-DWORD. Jedes Quarter-Byte wird entsprechend seiner Position im DWORD verschoben und dann mit den anderen verknüpft. Die Verschiebung erfolgt mithilfe von Konstanten (`quarterconst::SHIFT_QUARTER_XX`), die die Position jedes Quarter-Bytes im DWORD definieren.

Technische Besonderheiten
- Verwendet Bitverschiebungen (`SHL`) und logische ODER-Operationen (`OR`), um die Quarter-Bytes korrekt zu kombinieren.
- Die Konvertierung von `BYTE` zu `DWORD` erfolgt mit `BYTE_TO_DWORD`.

Zustandsübersicht
1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Processing**: Verarbeitet die Quarter-Bytes und kombiniert sie zu einem DWORD.
3. **Complete**: Sendet das `CNF`-Ereignis und stellt das kombinierte DWORD bereit.

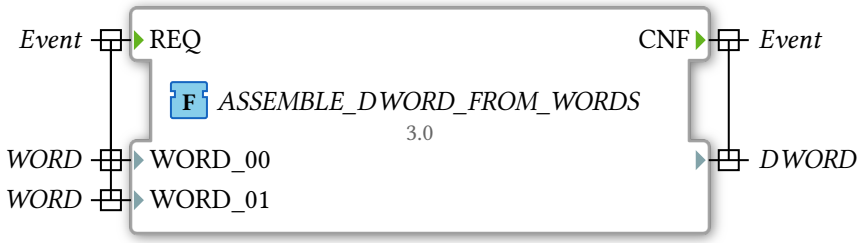
Anwendungsszenarien
- Datenkomprimierung: Kombination von mehreren kleinen Datenpaketen zu einem größeren.
- Kommunikationsprotokolle: Verarbeitung von Daten, die in kleineren Einheiten übertragen werden.
- Signalverarbeitung: Zusammenfassung von Teilen eines Signals zu einem vollständigen Wert.

⚖️ Vergleich mit ähnlichen Bausteinen
- Ähnliche Bausteine könnten `BYTE_TO_WORD` oder `WORD_TO_DWORD` sein, die jedoch weniger Flexibilität bieten, da sie nur feste Größen kombinieren.
- `ASSEMBLE_DWORD_FROM_QUARTERS` bietet mehr Kontrolle über die Position jedes einzelnen Bits im resultierenden DWORD.

🛠️ Zugehörige Übungen
- Uebung_060

Fazit
Der `ASSEMBLE_DWORD_FROM_QUARTERS`-Funktionsblock ist ein leistungsfähiges Werkzeug zur Kombination von 16 Quarter-Bytes zu einem DWORD. Durch seine flexible und präzise Arbeitsweise eignet er sich ideal für Anwendungen, bei denen Daten in kleineren Einheiten vorliegen und zu einer größeren Einheit zusammengefasst werden müssen.

ASSEMBLE_DWORD_FROM_WORDS



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ASSEMBLE_DWORD_FROM_WORDS` kombiniert zwei 16-Bit-Wörter (WORD) zu einem 32-Bit-Doppelwort (DWORD). Diese Funktionalität ist besonders nützlich, wenn Daten aus zwei separaten Quellen zusammengeführt werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Startet die Verarbeitung des Funktionsblocks. Muss mit den Daten-Eingängen `WORD_00` und `WORD_01` verknüpft sein.

Ereignis-Ausgänge

- `CNF`: Signalisiert die erfolgreiche Verarbeitung und Ausgabe des kombinierten DWORD.

Daten-Eingänge

- `WORD_00`: Erstes 16-Bit-Wort (WORD), das in das resultierende DWORD eingefügt wird.
- `WORD_01`: Zweites 16-Bit-Wort (WORD), das in das resultierende DWORD eingefügt wird.

Daten-Ausgänge

- (kein Name): Das kombinierte 32-Bit-Doppelwort (DWORD), das aus den beiden Eingabewörtern besteht.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock nimmt die beiden 16-Bit-Eingabewörter (`WORD_00` und `WORD_01`) entgegen und kombiniert sie zu einem einzigen 32-Bit-Doppelwort (DWORD). Dabei wird `WORD_00` in die unteren 16 Bits und `WORD_01` in die oberen 16 Bits des resultierenden DWORD platziert.

Technische Besonderheiten

- Die Implementierung erfolgt in Structured Text (ST).
- Der Funktionsblock ist Teil des Pakets `eclipse4diac::utils::assembling`.

Zustandsübersicht

1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Processing**: Verarbeitet die Eingabewörter und erzeugt das DWORD.
3. **Complete**: Sendet das `CNF`-Ereignis mit dem kombinierten DWORD.

Anwendungsszenarien

- Kombination von zwei 16-Bit-Sensordaten zu einem 32-Bit-Wert.
- Verarbeitung von Daten aus zwei separaten Kommunikationskanälen.
- Erweiterung von 16-Bit-Daten auf 32-Bit für weitere Verarbeitung.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu einfachen Bit-Operationen bietet dieser Funktionsblock eine klar definierte Schnittstelle und ist in 4diac-IDE integriert.
- Ähnliche Funktionalität könnte mit Basisoperationen erreicht werden, dieser Baustein vereinfacht jedoch die Handhabung.

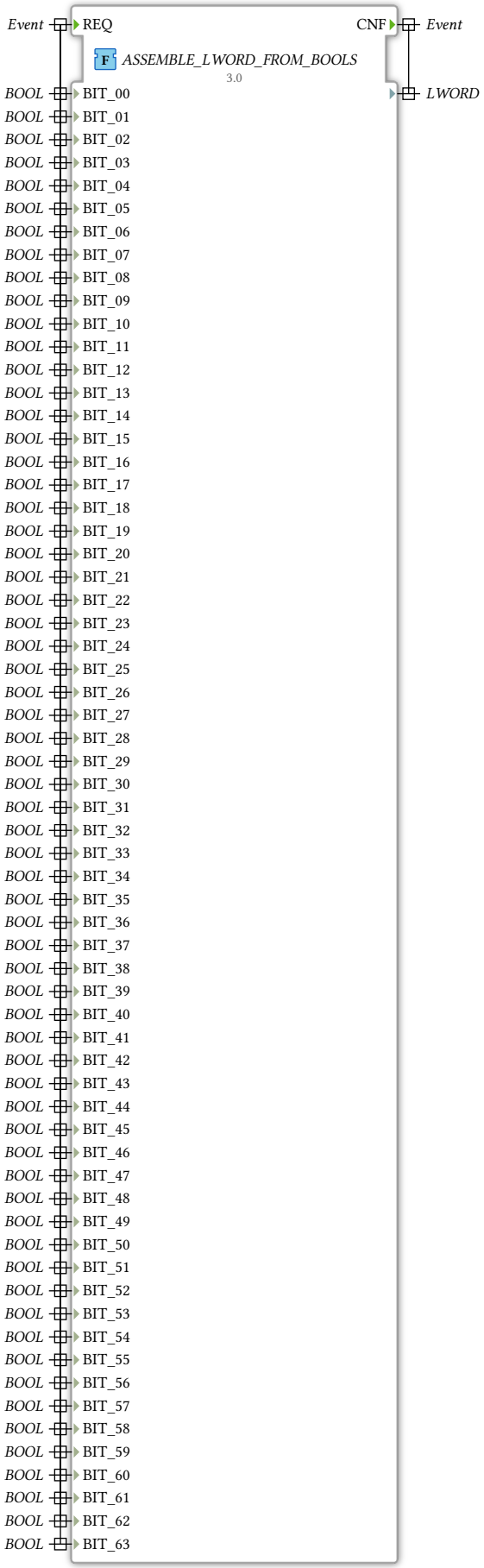
Fazit

Der `ASSEMBLE_DWORD_FROM_WORDS` Funktionsblock bietet eine einfache und effiziente Möglichkeit, zwei 16-Bit-Wörter zu einem 32-Bit-Doppelwort zu kombinieren. Seine klare Schnittstelle und direkte Integration in 4diac machen ihn besonders nützlich für Anwendungen, die mit der Kombination von Wortdaten arbeiten.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

ASSEMBLE_LWORD_FROM_BOOLS


Event
REQ
CNF
Event
F
ASSEMBLE_LWORD_FROM_BOOLS
3.0
BOOL BIT_00
BOOL BIT_01
BOOL BIT_02
BOOL BIT_03
BOOL BIT_04
BOOL BIT_05
BOOL BIT_06
BOOL BIT_07
BOOL BIT_08
BOOL BIT_09
BOOL BIT_10
BOOL BIT_11
BOOL BIT_12
BOOL BIT_13
BOOL BIT_14
BOOL BIT_15
BOOL BIT_16
BOOL BIT_17
BOOL BIT_18
BOOL BIT_19
BOOL BIT_20
BOOL BIT_21
BOOL BIT_22
BOOL BIT_23
BOOL BIT_24
BOOL BIT_25
BOOL BIT_26
BOOL BIT_27
BOOL BIT_28
BOOL BIT_29
BOOL BIT_30
BOOL BIT_31
BOOL BIT_32
BOOL BIT_33
BOOL BIT_34
BOOL BIT_35
BOOL BIT_36
BOOL BIT_37
BOOL BIT_38
BOOL BIT_39
BOOL BIT_40
BOOL BIT_41
BOOL BIT_42
BOOL BIT_43
BOOL BIT_44
BOOL BIT_45
BOOL BIT_46
BOOL BIT_47
BOOL BIT_48
BOOL BIT_49
BOOL BIT_50
BOOL BIT_51
BOOL BIT_52
BOOL BIT_53
BOOL BIT_54
BOOL BIT_55
BOOL BIT_56
BOOL BIT_57
BOOL BIT_58
BOOL BIT_59
BOOL BIT_60
BOOL BIT_61
BOOL BIT_62
BOOL BIT_63
LWORD

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ASSEMBLE_LWORD_FROM_BOOLS` kombiniert 64 einzelne BOOL-Werte zu einem einzigen LWORD-Datentyp. Dies ist besonders nützlich, wenn mehrere binäre Zustände in einer kompakten Form verarbeitet oder übertragen werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Startet die Kombination der 64 BOOL-Eingänge zu einem LWORD.

Ereignis-Ausgänge

- `CNF`: Signalisiert die erfolgreiche Kombination der BOOL-Werte zu einem LWORD.

Daten-Eingänge

64 BOOL-Eingänge (`BIT_00` bis `BIT_63`), die jeweils ein Bit im resultierenden LWORD repräsentieren.

Daten-Ausgänge

- Ein LWORD, das aus den 64 BOOL-Eingängen zusammengesetzt wird.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock weist jedes der 64 Bits des LWORD-Ausgangs einem entsprechenden BOOL-Eingang zu. Bei Auslösung des `REQ`-Ereignisses werden alle BOOL-Eingänge gelesen und in das LWORD eingefügt. Das `CNF`-Ereignis signalisiert den Abschluss dieses Vorgangs.

Technische Besonderheiten

- Verarbeitet 64 BOOL-Eingänge parallel.
- Direkte Bit-Zuordnung ohne zusätzliche Berechnungen.
- Effiziente Implementierung durch direkte Zuweisung der Bits.

Zustandsübersicht

1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Processing**: Kombiniert die BOOL-Werte zu einem LWORD.
3. **Complete**: Sendet das `CNF`-Ereignis und gibt das kombinierte LWORD aus.

Anwendungsszenarien

- Komprimierung mehrerer binärer Signale für die Übertragung.
- Verarbeitung von Bitmasken in Steuerungsanwendungen.
- Effiziente Speicherung von binären Zuständen.

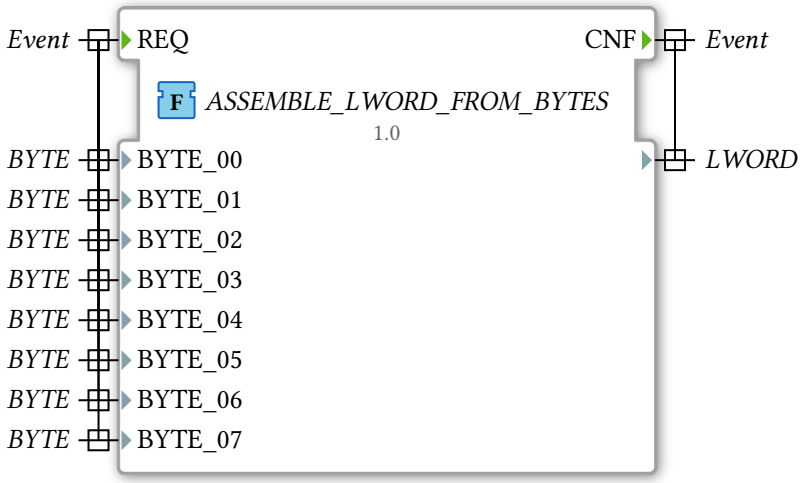
⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnlich zu `BOOL_TO_*`-Funktionsblöcken, aber für 64 Bits optimiert.
- Bietet mehr Flexibilität als fest codierte Bit-Operationen.

Fazit

Der `ASSEMBLE_LWORD_FROM_BOOLS`-Funktionsblock ist ein effizientes Werkzeug zur Kombination mehrerer BOOL-Werte in einen einzigen LWORD-Datentyp, ideal für Anwendungen mit hoher Bit-Dichte.

ASSEMBLE_LWORD_FROM_BYTES



• – ◦ • – ◦ • – ◦ •

Einleitung

Die Funktion `ASSEMBLE_LWORD_FROM_BYTES` kombiniert acht BYTE-Werte zu einem einzigen LWORD-Wert. Sie ist nützlich, wenn Daten aus mehreren Byte-Quellen (z. B. Protokollfelder) zu einer 64-Bit-Einheit zusammengesetzt werden sollen.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Kombination der BYTE-Werte zu einem LWORD. Verknüpft mit `BYTE_00` bis `BYTE_07`.

Ereignis-Ausgänge

- **CNF**: Signalisiert die abgeschlossene Kombination. Der LWORD-Wert liegt am Daten-Ausgang an.

Daten-Eingänge

- **BYTE_00**: Niedrigstwertiges Byte (B0) des LWORD.
- **BYTE_01**: Byte B1.
- **BYTE_02**: Byte B2.
- **BYTE_03**: Byte B3.
- **BYTE_04**: Byte B4.
- **BYTE_05**: Byte B5.
- **BYTE_06**: Byte B6.
- **BYTE_07**: Höchstwertiges Byte (B7) des LWORD.

Daten-Ausgänge

- **(unbenannt)**: Der zusammengesetzte LWORD-Wert (Rückgabewert der Funktion).

Adapter

Keine Adapter vorhanden.

Funktionsweise

Die Funktion weist jedes Eingangs-Byte direkt dem entsprechenden Byte des Rückgabewerts zu:

- `BYTE_00` ⟶ `%B0` (niedrigstwertig)
- `BYTE_01` ⟶ `%B1`
- `BYTE_02` ⟶ `%B2`
- `BYTE_03` ⟶ `%B3`
- `BYTE_04` ⟶ `%B4`
- `BYTE_05` ⟶ `%B5`
- `BYTE_06` ⟶ `%B6`

- `BYTE_07` ⟶ `%B7` (höchstwertig)

Technische Besonderheiten

- Implementiert als Funktion (`FUNCTION ... : LWORD`) in ST (Structured Text), Paket `eclipse4diac::utils::assembling`.
- Der Zugriff auf einzelne Bytes erfolgt über die partielle Zugriffssyntax `%B0` bis `%B7`.
- Keine eigene Zustandshaltung.

Anwendungsszenarien

- Zusammensetzen von 64-Bit-Werten aus Einzelbytes, z. B. bei Protokollimplementierungen oder CAN-Daten.

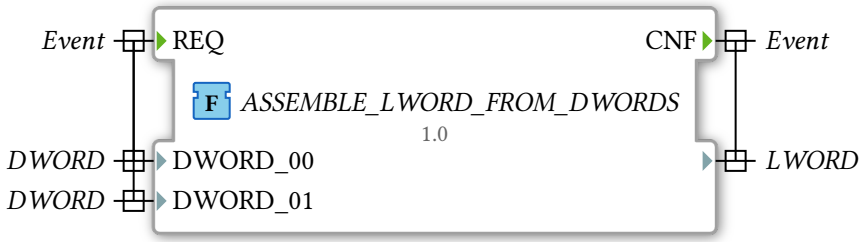
⚖️ Vergleich mit ähnlichen Bausteinen

- ASSEMBLE_DWORD_FROM_BYTES setzt vier Bytes zu einem DWORD zusammen, ASSEMBLE_LWORD_FROM_WORDS vier Worte zu einem LWORD.

Fazit

`ASSEMBLE_LWORD_FROM_BYTES` setzt acht BYTE-Werte byteweise zu einem LWORD zusammen.

ASSEMBLE_LWORD_FROM_DWORDS



• − ◦ • − ◦ • − ◦ •

Einleitung

Die Funktion `ASSEMBLE_LWORD_FROM_DWORDS` kombiniert zwei 32-Bit-Doppelwörter (`DWORD`) zu einem 64-Bit-Langwort (`LWORD`).

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Startet die Verarbeitung. Verbundene Daten: `DWORD_00`, `DWORD_01`

Ereignis-Ausgänge

- `CNF`: Signalisiert die Verarbeitung und die Ausgabe des kombinierten `LWORD`.

Daten-Eingänge

- `DWORD_00`: Doppelwort 0 (`DWORD`), unterste 32 Bit des Ergebnisses
- `DWORD_01`: Doppelwort 1 (`DWORD`), oberste 32 Bit (Bits 32 bis 63)

Daten-Ausgänge

- (kein Name): Das kombinierte 64-Bit-Langwort (`LWORD`), zugleich der Rückgabewert der Funktion

Adapter

Diese Funktion verwendet keine Adapter.

Funktionsweise

Die Funktion weist die beiden Eingabewerte über den Teilzugriff `.%D0` und `.%D1` dem Rückgabewert zu:

```
ASSEMBLE_LWORD_FROM_DWORDS.%D0 := DWORD_00;
ASSEMBLE_LWORD_FROM_DWORDS.%D1 := DWORD_01;
```

`DWORD_00` bildet damit das niederwertigste und `DWORD_01` das höchstwertige Doppelwort des `LWORD`.

Technische Besonderheiten

- Implementierung in Structured Text (ST) als `FUNCTION` ohne internen Zustand
- Teil des Pakets `eclipse4diac::utils::assembling`
- Die Umkehrfunktion ist `SPLIT_LWORD_INTO_DWORDS`
- Version 1.0 (2026-05-16)

Anwendungsszenarien

- Zusammensetzen eines 64-Bit-Werts aus zwei 32-Bit-Registern (z. B. Fieldbus-Daten)
- Erzeugen von 64-Bit-Bitmustern aus zwei Doppelwörtern

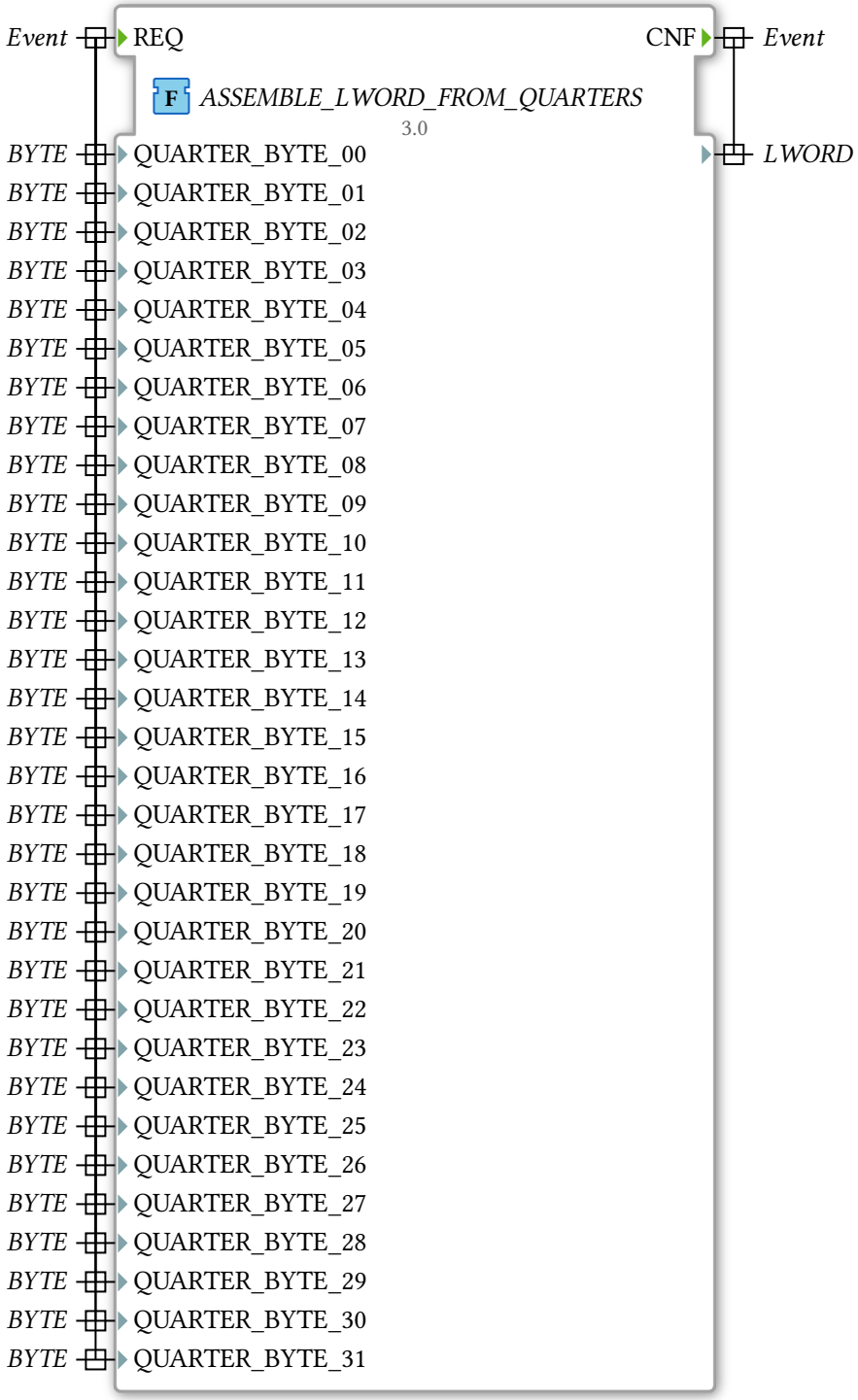
Fazit

`ASSEMBLE_LWORD_FROM_DWORDS` setzt zwei `DWORD`-Werte, beginnend mit dem niederwertigsten, zu einem `LWORD` zusammen.

ASSEMBLE_LWORD_FROM_QUARTERS

🎧 Podcast

- QUARTER



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ASSEMBLE_LWORD_FROM_QUARTERS` kombiniert 32 Byte-Werte (QUARTER_BYTE_00 bis QUARTER_BYTE_31) zu einem 64-Bit LWORD-Wert. Dies ist besonders nützlich, wenn Daten in kleineren Einheiten vorliegen und zu einem größeren Datenwort zusammengesetzt werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Startet die Verarbeitung und kombiniert die 32 Byte-Eingänge zu einem LWORD.

Ereignis-Ausgänge

- `CNF`: Signalisiert die erfolgreiche Zusammenführung der Bytes zu einem LWORD.

Daten-Eingänge

- `QUARTER_BYTE_00` bis `QUARTER_BYTE_31` (32 Eingänge vom Typ `BYTE`): Die einzelnen Byte-Werte, die zu einem LWORD kombiniert werden.

Daten-Ausgänge

- (Unbenannt, Typ `LWORD`): Der kombinierte 64-Bit-Wert, der aus den 32 Byte-Eingängen zusammengesetzt wird.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock nimmt die 32 Byte-Eingänge entgegen und kombiniert sie zu einem 64-Bit LWORD. Dabei wird jeder Byte-Wert an eine bestimmte Position im LWORD verschoben (mittels `SHL`-Operation) und dann mit dem aktuellen LWORD-Wert logisch verknüpft (`OR`-Operation). Die Verschiebung erfolgt gemäß vordefinierter Konstanten (`quarterconst::SHIFT_QUARTER_XX`), die die genaue Position jedes Bytes im LWORD festlegen.

Technische Besonderheiten

- Verwendet bitweise Operationen (`SHL`, `OR`) für die effiziente Kombination der Bytes.
- Die Reihenfolge der Kombination ist von `QUARTER_BYTE_31` (höchstwertiges Byte) bis `QUARTER_BYTE_00` (niederwertigstes Byte).
- Der Funktionsblock ist in ST (Structured Text) implementiert und nutzt eine Bibliothek (`quarterconst`) für die Verschiebungskonstanten.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Verarbeitung erfolgt direkt bei Empfang des `REQ`-Ereignisses, und das Ergebnis wird sofort über `CNF` ausgegeben.

Anwendungsszenarien

- Zusammensetzen von Daten, die in kleineren Einheiten (Bytes) übertragen oder gespeichert wurden.
- Verarbeitung von Protokollen oder Datenformaten, die große Werte in segmentierter Form übermitteln.
- Einsatz in Systemen, die mit 64-Bit-Daten arbeiten, aber nur 8-Bit-Schnittstellen zur Verfügung haben.

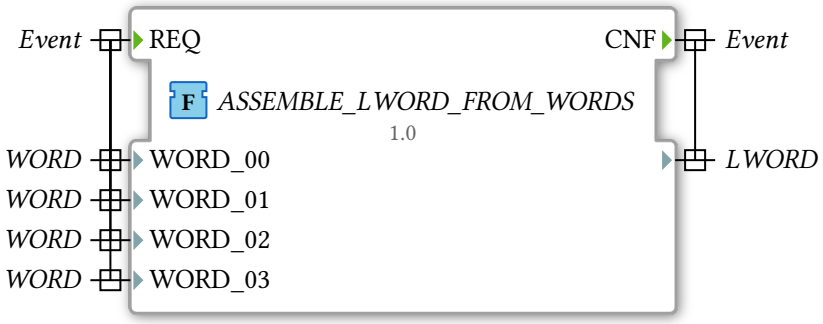
⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Funktionsblöcke könnten z. B. `ASSEMBLE_DWORD_FROM_BYTES` sein, die jedoch nur 32-Bit-Werte kombinieren.
- Im Gegensatz zu manuellen Implementierungen bietet dieser Baustein eine standardisierte und optimierte Lösung für die Kombination von 32 Bytes zu einem LWORD.

Fazit

Der `ASSEMBLE_LWORD_FROM_QUARTERS`-Funktionsblock bietet eine effiziente und zuverlässige Methode, um 32 Byte-Werte zu einem 64-Bit LWORD zu kombinieren. Durch die Verwendung von bitweisen Operationen und vordefinierten Verschiebungskonstanten ist die Lösung performant und leicht in bestehende Systeme integrierbar. Ideal für Anwendungen, die mit großen Datenwerten in segmentierter Form arbeiten.

ASSEMBLE_LWORD_FROM_WORDS



• – ◦ • – ◦ • – ◦ •

Einleitung

Die Funktion `ASSEMBLE_LWORD_FROM_WORDS` kombiniert vier 16-Bit-Wörter (`WORD`) zu einem 64-Bit-Langwort (`LWORD`).

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Startet die Verarbeitung. Verbundene Daten: `WORD_00`, `WORD_01`, `WORD_02`, `WORD_03`

Ereignis-Ausgänge

- `CNF`: Signalisiert die Verarbeitung und die Ausgabe des kombinierten `LWORD`.

Daten-Eingänge

- `WORD_00`: Wort 0 (`WORD`), unterste 16 Bit des Ergebnisses
- `WORD_01`: Wort 1 (`WORD`), Bits 16 bis 31
- `WORD_02`: Wort 2 (`WORD`), Bits 32 bis 47
- `WORD_03`: Wort 3 (`WORD`), oberste 16 Bit (Bits 48 bis 63)

Daten-Ausgänge

- (kein Name): Das kombinierte 64-Bit-Langwort (`LWORD`), zugleich der Rückgabewert der Funktion

Adapter

Diese Funktion verwendet keine Adapter.

Funktionsweise

Die Funktion weist die vier Eingabewörter über den Teilzugriff `.%W0` bis `.%W3` dem Rückgabewert zu:

```
ASSEMBLE_LWORD_FROM_WORDS.%W0 := WORD_00;
ASSEMBLE_LWORD_FROM_WORDS.%W1 := WORD_01;
ASSEMBLE_LWORD_FROM_WORDS.%W2 := WORD_02;
ASSEMBLE_LWORD_FROM_WORDS.%W3 := WORD_03;
```

`WORD_00` bildet damit das niederwertigste und `WORD_03` das höchstwertige Wort des `LWORD`.

Technische Besonderheiten

- Implementierung in Structured Text (ST) als `FUNCTION` ohne internen Zustand
- Teil des Pakets `eclipse4diac::utils::assembling`
- Die Umkehrfunktion ist `SPLIT_LWORD_INTO_DWORDS` bzw. die jeweiligen `SPLIT_*`-Funktionen im Paket `splitting`

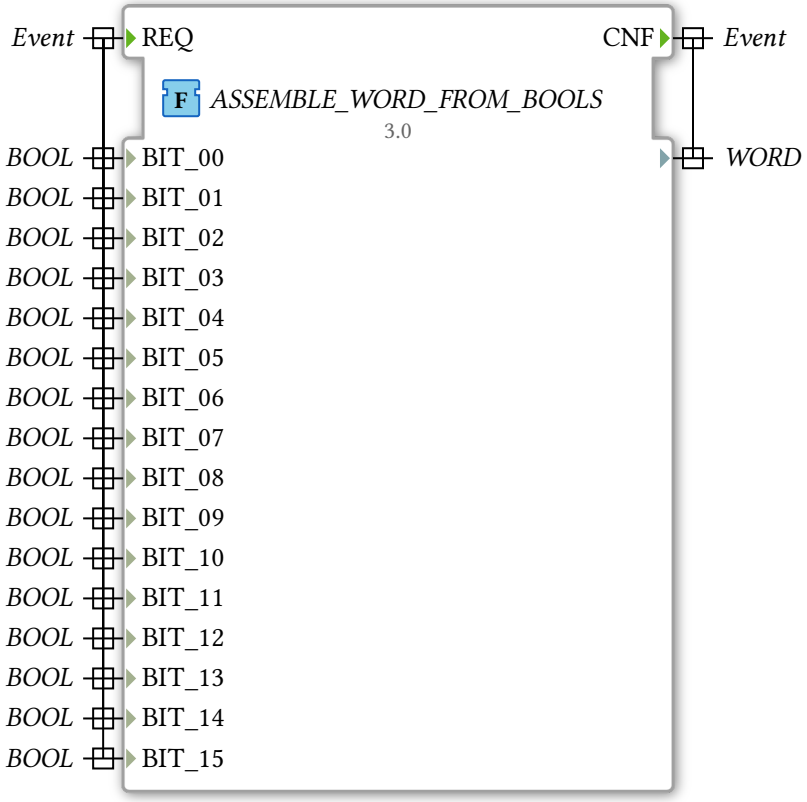
Anwendungsszenarien

- Zusammensetzen eines 64-Bit-Werts aus vier 16-Bit-Registern (z. B. Fieldbus-/Modbus-Daten)
- Erzeugen von 64-Bit-Bitmustern aus einzelnen Wörtern

Fazit

`ASSEMBLE_LWORD_FROM_WORDS` setzt vier `WORD`-Werte, beginnend mit dem niederwertigsten, zu einem `LWORD` zusammen.

ASSEMBLE_WORD_FROM_BOOLS



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ASSEMBLE_WORD_FROM_BOOLS` kombiniert 16 boolesche Eingangswerte zu einem einzigen WORD-Ausgangswert. Jeder boolesche Eingang repräsentiert dabei ein einzelnes Bit im resultierenden WORD.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Startet die Verarbeitung und übernimmt alle aktuellen Werte der booleschen Eingänge.

Ereignis-Ausgänge

- `CNF`: Signalisiert die erfolgreiche Zusammenführung der booleschen Werte zu einem WORD.

Daten-Eingänge

16 boolesche Eingänge (BIT_00 bis BIT_15), die jeweils einem Bit im resultierenden WORD entsprechen:

- `BIT_00` bis `BIT_15` (BOOL): Einzelne Bits für das zusammenzusetzende WORD

Daten-Ausgänge

- `(Unbenannt)` (WORD): Das resultierende WORD, das aus den 16 booleschen Eingängen zusammengesetzt wurde.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Bei Auslösung des `REQ`-Ereignisses werden alle 16 booleschen Eingangswerte gelesen und zu einem 16-Bit-WORD kombiniert. Dabei entspricht:

- `BIT_00` dem Bit 0 (LSB)
- `BIT_15` dem Bit 15 (MSB)

Die Zuweisung erfolgt direkt über Bitoperationen, wobei jeder boolesche Eingangswert auf das entsprechende Bit im WORD abgebildet wird.

Technische Besonderheiten

- Die Bitreihenfolge ist fest vorgegeben (BIT_00 = LSB, BIT_15 = MSB)
- Der Block führt keine zusätzlichen Berechnungen oder Transformationen durch
- Die Operation ist atomar und erfolgt in einem Schritt

Zustandsübersicht

1. **Idle**: Wartet auf `REQ`-Ereignis
2. **Processing**: Verarbeitet die Eingänge und setzt das WORD zusammen
3. **Output**: Sendet `CNF` mit dem resultierenden WORD

Anwendungsszenarien

- Zusammenführen von einzelnen Schaltzuständen zu einem kompakten Datenwort
- Vorbereitung von Daten für die Kommunikation mit Geräten, die WORD-Daten erwarten
- Bitweise Steuerung von Hardware-Registern

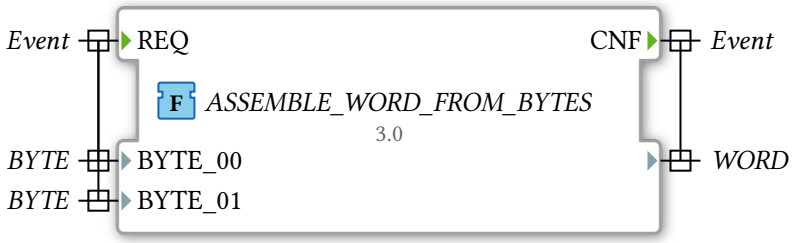
⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu allgemeinen Kombinationsblöcken ist dieser FB speziell auf die Konvertierung von BOOL zu WORD optimiert
- Bietet eine direktere und effizientere Lösung als die Verwendung mehrerer Einzelblöcke
- Ähnlich zu BIT_TO_* Blöcken, aber mit fester 16-Bit-WORD-Ausgabe

Fazit

Der `ASSEMBLE_WORD_FROM_BOOLS` Funktionsblock bietet eine effiziente und direkte Methode, um 16 boolesche Signale in ein einzelnes WORD zu konvertieren. Seine einfache und deterministische Funktionsweise macht ihn besonders geeignet für Anwendungen, bei denen einzelne Bits zu kompakten Datenwörtern zusammengefasst werden müssen.

ASSEMBLE_WORD_FROM_BYTES



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ASSEMBLE_WORD_FROM_BYTES` kombiniert zwei Byte-Werte zu einem Wort (WORD). Dies ist besonders nützlich, wenn Daten aus zwei separaten Byte-Quellen zu einem zusammenhängenden 16-Bit-Wert verknüpft werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Verarbeitung. Die Eingabedaten `BYTE_00` und `BYTE_01` werden mit diesem Ereignis übergeben.

Ereignis-Ausgänge

- **CNF**: Signalisiert die erfolgreiche Zusammenführung der Bytes zu einem WORD.

Daten-Eingänge

- **BYTE_00**: Das niederwertige Byte (LSB) des resultierenden WORD.
- **BYTE_01**: Das höherwertige Byte (MSB) des resultierenden WORD.

Daten-Ausgänge

- **(kein Name)**: Das resultierende WORD, zusammengesetzt aus `BYTE_00` (LSB) und `BYTE_01` (MSB).

Adapter

Es werden keine Adapter verwendet.

Funktionsweise

Der Funktionsblock setzt die beiden Eingabe-Bytes `BYTE_00` und `BYTE_01` zu einem 16-Bit-WORD zusammen. Dabei wird `BYTE_00` als niederwertiges Byte (LSB) und `BYTE_01` als höherwertiges Byte (MSB) verwendet. Die Operation erfolgt unmittelbar nach dem Eintreffen des `REQ`-Ereignisses, und das Ergebnis wird mit dem `CNF`-Ereignis ausgegeben.

Technische Besonderheiten

- Die Zusammenführung erfolgt direkt durch Zuweisung der Bytes zu den entsprechenden Positionen im WORD.
- Der Block ist für die Verwendung in Echtzeit-Systemen optimiert und hat eine deterministische Laufzeit.

Zustandsübersicht

1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Processing**: Verarbeitet die Eingabedaten und erzeugt das WORD.
3. **Output**: Sendet das `CNF`-Ereignis mit dem resultierenden WORD.

Anwendungsszenarien

- Kombination von Sensorwerten, die als separate Bytes vorliegen.
- Verarbeitung von Kommunikationsprotokollen, bei denen Daten in Byte-Paaren übertragen werden.
- Allgemeine Datenmanipulation in Steuerungssystemen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Rechenbausteinen ist `ASSEMBLE_WORD_FROM_BYTES` spezialisiert und effizienter für diese spezielle Aufgabe.
- Ähnliche Bausteine könnten zusätzliche Funktionen wie Byte-Swapping oder Fehlerbehandlung bieten, sind aber komplexer.

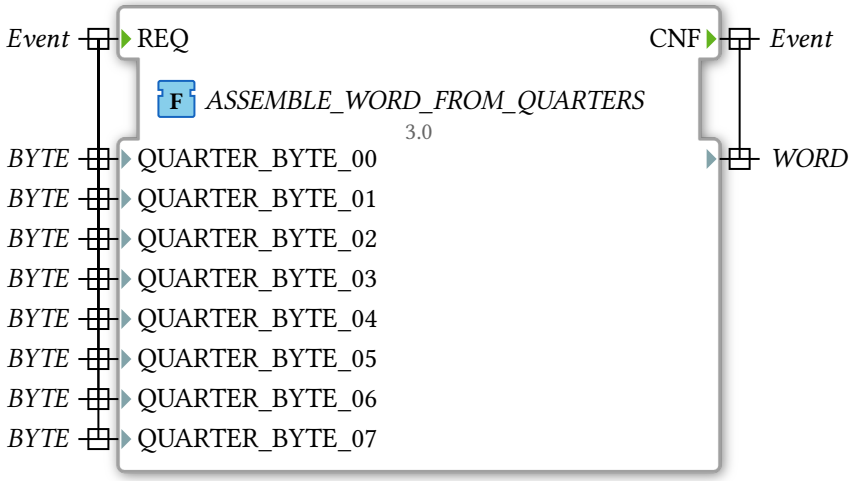
Fazit

Der `ASSEMBLE_WORD_FROM_BYTES`-Funktionsblock ist ein einfacher und effizienter Baustein zur Kombination von zwei Bytes zu einem WORD. Er eignet sich besonders für Anwendungen, bei denen Performance und Einfachheit im Vordergrund stehen.

ASSEMBLE_WORD_FROM_QUARTERS

🎧 Podcast

- QUARTER



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `ASSEMBLE_WORD_FROM_QUARTERS` kombiniert 8 Quarter-Bytes zu einem WORD. Dies ist besonders nützlich, wenn Daten in kleineren Einheiten vorliegen und zu einer größeren Einheit zusammengefügt werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Startet die Verarbeitung und kombiniert die 8 Quarter-Bytes zu einem WORD.

Ereignis-Ausgänge

- `CNF`: Signalisiert die erfolgreiche Kombination der Quarter-Bytes zu einem WORD.

Daten-Eingänge

- `QUARTER_BYTE_00` bis `QUARTER_BYTE_07` (Typ: `BYTE`): Die 8 Quarter-Bytes, die zu einem WORD kombiniert werden.

Daten-Ausgänge

- `OUT` (Typ: `WORD`): Das resultierende WORD, das aus den 8 Quarter-Bytes zusammengesetzt wurde.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock nimmt 8 Quarter-Bytes entgegen und kombiniert sie zu einem WORD. Jedes Quarter-Byte wird an eine bestimmte Position im WORD verschoben und mit logischen OR-Operationen zusammengefügt. Die genauen Verschiebungen werden durch Konstanten (`quarterconst::SHIFT_QUARTER_00` bis `quarterconst::SHIFT_QUARTER_07`) definiert.

Technische Besonderheiten

- Verwendet bitweise Operationen (SHL und OR) zur Kombination der Bytes.
- Die Verschiebungspositionen sind durch Konstanten definiert, was eine flexible Anpassung ermöglicht.

Zustandsübersicht

1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Processing**: Kombiniert die Quarter-Bytes zu einem WORD.
3. **Complete**: Sendet das `CNF`-Ereignis mit dem resultierenden WORD.

Anwendungsszenarien

- Datenkomprimierung: Kombination von kleineren Dateneinheiten zu größeren.

- Kommunikationsprotokolle: Zusammensetzung von Datenpaketen aus einzelnen Bytes.
- Signalverarbeitung: Kombination von Signalabschnitten zu einem Gesamtsignal.

⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Bausteine könnten `BYTE_TO_WORD` oder `SHL` sein, jedoch bietet `ASSEMBLE_WORD_FROM_QUARTERS` eine spezialisierte Funktion zur Kombination von 8 Quarter-Bytes zu einem WORD.
- Im Gegensatz zu generischen Kombinationsbausteinen ist dieser Funktionsblock auf die spezifische Anforderung der Quarter-Byte-Kombination optimiert.

Fazit

Der `ASSEMBLE_WORD_FROM_QUARTERS`-Funktionsblock ist ein effizientes Werkzeug zur Kombination von 8 Quarter-Bytes zu einem WORD. Durch die Verwendung von bitweisen Operationen und definierten Verschiebungspositionen bietet er eine zuverlässige und flexible Lösung für diese spezifische Aufgabe.

2.4.9.19 const

quarterconst

🎧 Podcast

- QUARTER

• – ◦ • – ◦ • – ◦ •

Einleitung

Globale Konstanten für Quarter-Werte.

Metadaten

Attribut	Wert
Lizenz	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac-Paket	eclipse4diac::utils

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

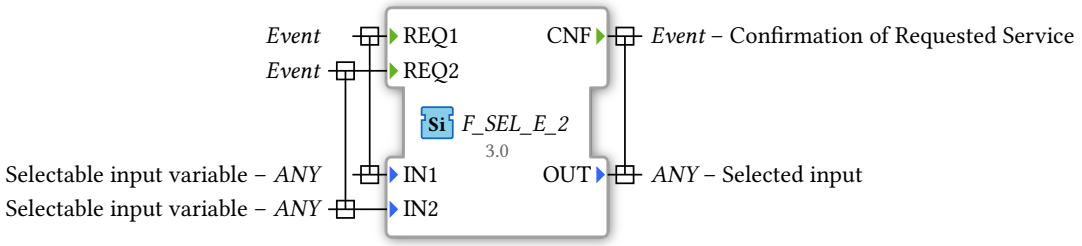
2.4.9.20 selection

F_SEL_E_2

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `F_SEL_E_2` ist ein ereignisbasierter Selektor, der zwischen zwei Eingangsvariablen auswählt. Er gehört zur Kategorie der Standard-Selektionsfunktionen gemäß IEC 61131-3 und wird typischerweise in Steuerungsanwendungen eingesetzt, wo eine bedingte Auswahl zwischen zwei Eingangssignalen erforderlich ist.



Schnittstellenstruktur

Ereignis-Eingänge

- `REQ0`: Ereigniseingang, der die Auswahl der ersten Eingangsvariable (`IN0`) auslöst.
- `REQ1`: Ereigniseingang, der die Auswahl der zweiten Eingangsvariable (`IN1`) auslöst.

Ereignis-Ausgänge

- `CNF`: Bestätigungsereignis, das nach der Auswahl einer Eingangsvariable ausgegeben wird. Dieses Ereignis ist mit der Ausgangsvariable `OUT` verknüpft.

Daten-Eingänge

- `IN0`: Erste selektierbare Eingangsvariable (Typ: `ANY`).
- `IN1`: Zweite selektierbare Eingangsvariable (Typ: `ANY`).

Daten-Ausgänge

- `OUT`: Ausgewählte Eingangsvariable (Typ: `ANY`). Der Wert entspricht entweder `IN0` oder `IN1`, abhängig vom auslösenden Ereignis (`REQ0` oder `REQ1`).

Adapter

- Dieser Funktionsblock verfügt über keine Adapter.

Funktionsweise

Der Funktionsblock `F_SEL_E_2` wählt zwischen zwei Eingangsvariablen basierend auf den eingehenden Ereignissen aus:

- Wird das Ereignis `REQ0` ausgelöst, wird der Wert von `IN0` an `OUT` weitergegeben.
- Wird das Ereignis `REQ1` ausgelöst, wird der Wert von `IN1` an `OUT` weitergegeben.
- Nach der Auswahl wird das Bestätigungsereignis `CNF` ausgegeben, um die erfolgreiche Selektion zu signalisieren.

Technische Besonderheiten

- Der Funktionsblock unterstützt den Datentyp `ANY`, was bedeutet, dass er mit verschiedenen Datentypen kompatibel ist.
- Die Auswahl erfolgt ausschließlich ereignisbasiert, d.h., die logische Bedingung für die Auswahl wird durch externe Ereignisse gesteuert.

Zustandsübersicht

1. **Initialzustand**: Der Funktionsblock wartet auf ein Ereignis (`REQ0` oder `REQ1`).
2. **Auswahlzustand**:
   - Bei `REQ0` wird `IN0` ausgewählt.
   - Bei `REQ1` wird `IN1` ausgewählt.
3. **Bestätigungszustand**: Nach der Auswahl wird `CNF` ausgegeben, und der Block kehrt in den Initialzustand zurück.

Anwendungsszenarien

- Auswahl zwischen zwei Sensordaten basierend auf externen Bedingungen.
- Umschaltung zwischen zwei Betriebsmodi in einer Steuerung.
- Dynamische Auswahl von Eingangssignalen in Abhängigkeit von Prozessereignissen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Selektionsblöcken (z.B. `SEL` in der IEC 61131-3) ist `F_SEL_E_2` rein ereignisgesteuert und benötigt keine zusätzliche logische Bedingung für die Auswahl.
- Ähnliche Blöcke wie `MUX` (Multiplexer) erlauben die Auswahl aus mehreren Eingängen, sind aber oft datenflussgesteuert und nicht ereignisbasiert.

🛠️ Zugehörige Übungen

- Uebung_019
- Uebung_074

Fazit

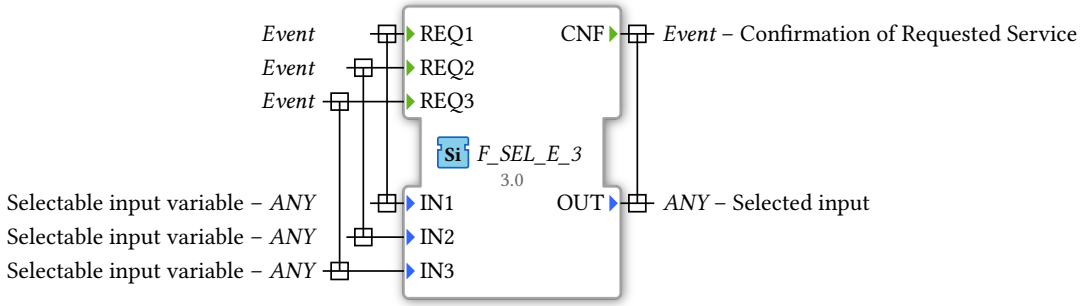
Der F_SEL_E_2 ist ein flexibler und einfach zu verwendender Funktionsblock für Anwendungen, bei denen eine ereignisbasierte Auswahl zwischen zwei Eingangsvariablen erforderlich ist. Seine Kompatibilität mit dem ANY-Datentyp macht ihn besonders vielseitig einsetzbar.

F_SEL_E_3

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock F_SEL_E_3 ist ein ereignisbasierter Selektor, der eine von drei Eingangsvariablen basierend auf dem auslösenden Ereignis auswählt und an den Ausgang weiterleitet. Er ist Teil des eclipse4diac::utils::selection Pakets und entspricht dem IEC 61131-3 Standard für Standard-Selektionsfunktionen.



Schnittstellenstruktur

Ereignis-Eingänge

- REQ0: Wählt die Eingangsvariable IN0 aus
- REQ1: Wählt die Eingangsvariable IN1 aus
- REQ2: Wählt die Eingangsvariable IN2 aus

Ereignis-Ausgänge

- CNF: Bestätigt die Ausführung der Selektion und liefert den ausgewählten Wert

Daten-Eingänge

- IN0: Erste selektierbare Eingangsvariable (Typ: ANY)
- IN1: Zweite selektierbare Eingangsvariable (Typ: ANY)
- IN2: Dritte selektierbare Eingangsvariable (Typ: ANY)

Daten-Ausgänge

- OUT: Ausgewählte Eingangsvariable (Typ: ANY)

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock wartet auf ein eingehendes Ereignis (REQ0, REQ1 oder REQ2). Beim Eintreffen eines solchen Ereignisses wird die entsprechende Eingangsvariable (IN0, IN1 bzw. IN2) ausgewählt und an den Ausgang OUT weitergeleitet. Gleichzeitig wird das Bestätigungsereignis CNF ausgelöst.

Technische Besonderheiten

- Unterstützt den Datentyp ANY für maximale Flexibilität
- Ereignisbasierte Auslösung für deterministisches Verhalten
- Einfache und direkte Selektion ohne zusätzliche Logik

Zustandsübersicht

1. Wartezustand: Kein aktives Ereignis
2. Auswahlzustand: Ereignis empfangen, entsprechender Eingang ausgewählt

3. Bestätigungszustand: Ausgang aktualisiert und CNF-Ereignis ausgelöst

Anwendungsszenarien

- Auswahl zwischen verschiedenen Sensordatenströmen
- Umschaltung zwischen Betriebsmodi
- Multiplexing von Datenquellen in Steuerungssystemen
- Implementierung von Prioritätslogik

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu `F_SEL_E_2` bietet dieser Baustein drei statt zwei Eingänge
- Ähnlich zu `F_MUX`, aber mit ereignisbasierter statt datengesteuerter Auswahl
- Komplexer als einfache Gatter, aber spezialisierter für Selektionsaufgaben

🛠️ Zugehörige Übungen

- Uebung_015a
- Uebung_016a

Fazit

Der `F_SEL_E_3` ist ein effizienter und flexibler Baustein für Anwendungen, bei denen zwischen drei verschiedenen Datenquellen ereignisgesteuert gewechselt werden muss. Seine einfache Schnittstelle und Unterstützung für beliebige Datentypen machen ihn zu einem vielseitigen Werkzeug in der Automatisierungstechnik.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

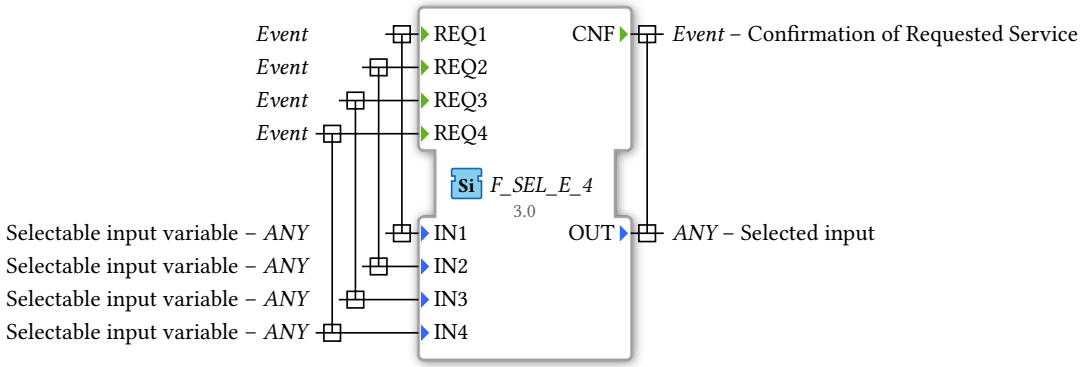
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

F_SEL_E_4

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock F_SEL_E_4 ist ein ereignisbasierter Selektor, der aus bis zu vier Eingängen einen auswählt und an den Ausgang weiterleitet. Er ist gemäß der Norm IEC 61131-3 klassifiziert und dient als Standard-Selektionsfunktion in Automatisierungssystemen.



Schnittstellenstruktur

Ereignis-Eingänge

- **REQ0**: Ereignis, das den Wert von IN0 auswählt.
- **REQ1**: Ereignis, das den Wert von IN1 auswählt.
- **REQ2**: Ereignis, das den Wert von IN2 auswählt.
- **REQ3**: Ereignis, das den Wert von IN3 auswählt.

Ereignis-Ausgänge

- **CNF**: Bestätigungsereignis, das ausgelöst wird, sobald ein Eingang ausgewählt und der Wert an OUT weitergeleitet wurde.

Daten-Eingänge

- **IN0**: Erster selektierbarer Eingang (Typ: ANY).
- **IN1**: Zweiter selektierbarer Eingang (Typ: ANY).
- **IN2**: Dritter selektierbarer Eingang (Typ: ANY).
- **IN3**: Vierter selektierbarer Eingang (Typ: ANY).

Daten-Ausgänge

- **OUT**: Ausgewählter Eingang (Typ: ANY).

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der F_SEL_E_4 wählt basierend auf dem auslösenden Ereignis (REQ0 bis REQ3) den entsprechenden Eingang (IN0 bis IN3) aus und gibt den Wert an OUT weiter. Das CNF-Ereignis bestätigt die erfolgreiche Auswahl und Weitergabe.

Technische Besonderheiten

- Unterstützt den Datentyp ANY, was eine flexible Verwendung mit verschiedenen Datentypen ermöglicht.
- Ereignisbasierte Auswahl, die eine präzise Steuerung ermöglicht.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Auswahl und Weitergabe erfolgt unmittelbar bei Eintreffen des entsprechenden REQ-Ereignisses.

Anwendungsszenarien

- Auswahl zwischen verschiedenen Sensordaten in einer Steuerung.
- Multiplexing von Steuersignalen in Automatisierungssystemen.
- Flexible Datenweiterleitung in Abhängigkeit von externen Ereignissen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu anderen Selektoren bietet F_SEL_E_4 eine ereignisbasierte Steuerung, was eine präzisere Auswahl ermöglicht als zeit- oder zustandsbasierte Alternativen.
- Die Unterstützung des ANY-Datentyps macht ihn flexibler als typenspezifische Selektoren.

🛠️ Zugehörige Übungen

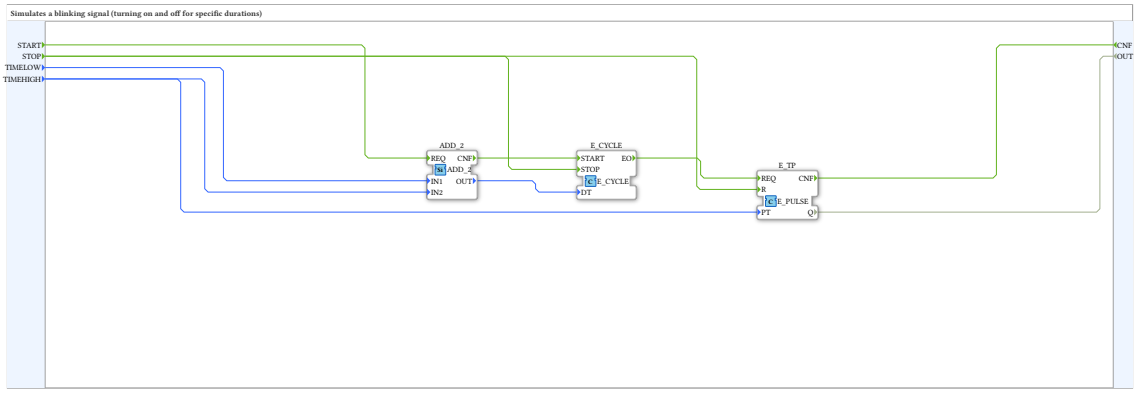
- Uebung_019a
- Uebung_019b
- Uebung_019c

Fazit

Der F_SEL_E_4 ist ein vielseitiger und flexibler Funktionsblock für die ereignisbasierte Auswahl von Daten in Automatisierungssystemen. Seine einfache Schnittstelle und die Unterstützung verschiedener Datentypen machen ihn zu einer praktischen Lösung für viele Anwendungsfälle.

2.4.9.21 signals

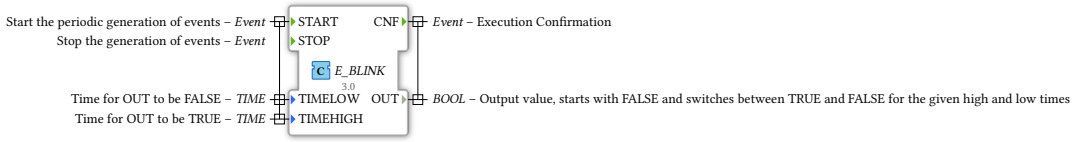
E_BLINK



• – ○ • – ○ • – ○ •

Einleitung

Der E_BLINK Funktionsblock simuliert ein blinkendes Signal, das zwischen den Zuständen TRUE und FALSE wechselt. Die Dauer der einzelnen Zustände kann individuell konfiguriert werden. Im Gegensatz zu ähnlichen Implementierungen kehrt dieser Funktionsblock beim STOP-Ereignis immer auf FALSE zurück.



Schnittstellenstruktur

Ereignis-Eingänge

- **START**: Startet die periodische Generierung von Ereignissen. Verwendet die Daten `TIMELOW` und `TIMEHIGH`.
- **STOP**: Stoppt die Generierung von Ereignissen und setzt den Ausgang auf FALSE.

Ereignis-Ausgänge

- **CNF**: Bestätigt die Ausführung und liefert den aktuellen Ausgangswert `OUT`.

Daten-Eingänge

- **TIMELOW**: Zeit in Millisekunden, während der der Ausgang FALSE ist.
- **TIMEHIGH**: Zeit in Millisekunden, während der der Ausgang TRUE ist.

Daten-Ausgänge

- **OUT**: Der Ausgangswert, der zwischen TRUE und FALSE wechselt.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der E_BLINK Funktionsblock verwendet intern einen E_PULSE Timer und einen E_CYCLE Generator, um die periodischen Zustandswechsel zu realisieren. Beim START-Ereignis werden die Zeiten für TRUE und FALSE addiert und als Zykluszeit für E_CYCLE verwendet. Der E_PULSE Timer steuert dann die Dauer des TRUE-Zustands, während die Differenzzeit den FALSE-Zustand bestimmt.

Technische Besonderheiten

- Der Funktionsblock setzt den Ausgang beim STOP-Ereignis immer auf FALSE zurück.
- Die Implementierung unterscheidet sich von ähnlichen Lösungen wie CODESYS BLINK, die den letzten Zustand beibehalten.

Zustandsübersicht

1. **Inaktiv**: Ausgang ist FALSE.
2. **Aktiv**: Ausgang wechselt periodisch zwischen TRUE und FALSE.
3. **Gestoppt**: Ausgang wird auf FALSE gesetzt.

Anwendungsszenarien

- Steuerung von blinkenden Warnleuchten
- Generierung von Taktsignalen für Testzwecke
- Visualisierung von Zustandsänderungen in Bedienoberflächen

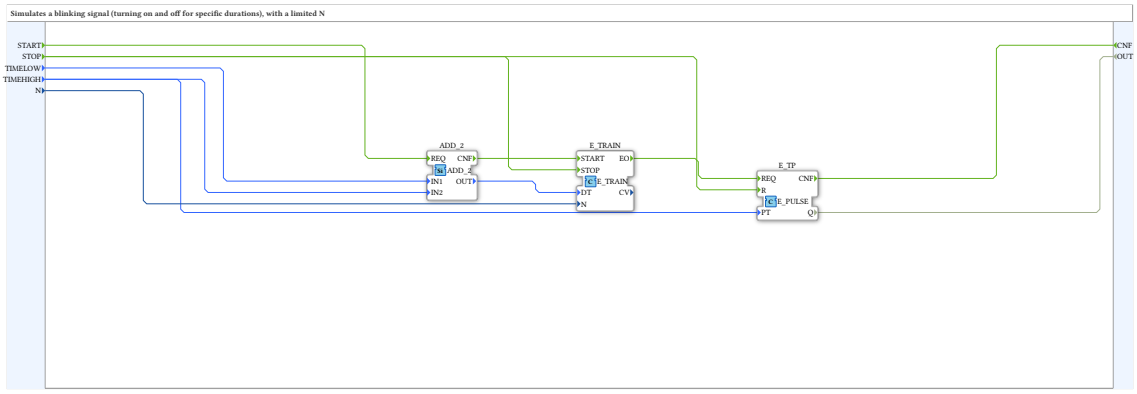
⚖️ Vergleich mit ähnlichen Bausteinen

Im Vergleich zum CODESYS BLINK Funktionsblock bietet E_BLINK eine vereinfachte Handhabung durch das automatische Zurücksetzen auf FALSE beim Stoppen. Die Funktionalität ist ähnlich, aber das Verhalten bei STOP ist konsistenter.

Fazit

Der E_BLINK Funktionsblock bietet eine einfache und zuverlässige Möglichkeit, periodische Signale mit konfigurierbaren Ein- und Ausschaltzeiten zu generieren. Die klare Definition des STOP-Verhaltens macht ihn besonders für sicherheitsrelevante Anwendungen geeignet.

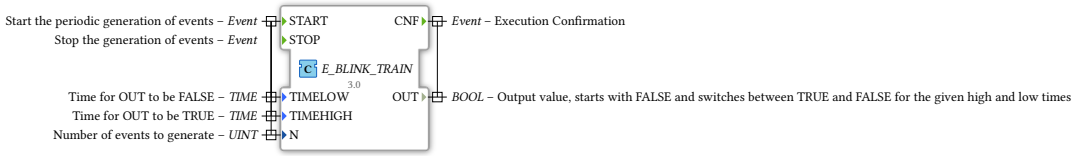
E_BLINK_TRAIN



• – ○ • – ○ • – ○ •

Einleitung

Der Funktionsblock `E_BLINK_TRAIN` simuliert ein blinkendes Signal, das für bestimmte Zeitintervalle ein- und ausgeschaltet wird. Die Anzahl der Blinkzyklen kann begrenzt werden. Im Gegensatz zu ähnlichen Blink-Funktionsblöcken kehrt dieser FB beim Stoppen immer auf den Zustand `FALSE` zurück.



Schnittstellenstruktur

Ereignis-Eingänge

- `START`: Startet die periodische Generierung von Ereignissen. Verknüpft mit den Daten-Eingängen `TIMELOW`, `TIMEHIGH` und `N`.
- `STOP`: Stoppt die Generierung von Ereignissen.

Ereignis-Ausgänge

- `CNF`: Bestätigt die Ausführung und liefert den aktuellen Ausgabewert `OUT`.

Daten-Eingänge

- `TIMELOW` (TIME): Zeit, während der `OUT` den Wert `FALSE` hat.

- `TIMEHIGH` (TIME): Zeit, während der `OUT` den Wert `TRUE` hat.
- `N` (UINT): Anzahl der zu generierenden Blinkzyklen.

Daten-Ausgänge

- `OUT` (BOOL): Ausgabewert, der zwischen `TRUE` und `FALSE` wechselt. Startet mit `FALSE`.

Funktionsweise

1. Bei einem `START`-Ereignis wird die Summe aus `TIMELOW` und `TIMEHIGH` berechnet und als Periodendauer für den Blinkzyklus verwendet.
2. Der FB generiert `N` Blinkzyklen, wobei `OUT` für `TIMEHIGH` auf `TRUE` und für `TIMELOW` auf `FALSE` gesetzt wird.
3. Bei einem `STOP`-Ereignis wird die Generierung beendet und `OUT` auf `FALSE` zurückgesetzt.
4. Jeder Zustandswechsel wird über das `CNF`-Ereignis bestätigt.

Technische Besonderheiten

- Verwendet interne Funktionsblöcke `E_PULSE` (für die Zeitsteuerung) und `E_TRAIN` (für die Begrenzung der Zyklen).
- Die Periodendauer wird durch Addition von `TIMELOW` und `TIMEHIGH` bestimmt.
- Im Gegensatz zu einigen anderen Blink-FBs wird beim Stoppen immer `FALSE` ausgegeben.

Zustandsübersicht

1. **Inaktiv**: `OUT` = `FALSE` (Initialzustand oder nach `STOP`).
2. **Aktiv**: Generiert Blinkzyklen gemäß den eingestellten Zeiten und der Zyklenanzahl.
   - `OUT` wechselt zwischen `TRUE` (`TIMEHIGH`) und `FALSE` (`TIMELOW`).

Anwendungsszenarien

- Anzeige von Warn- oder Statuslichten mit definierter Blinkfrequenz.
- Steuerung von periodischen Signalen in industriellen Automatisierungsprozessen.
- Simulation von Blinkmustern für Testzwecke.

⚖️ Vergleich mit ähnlichen Bausteinen

- **CODESYS BLINK-FB**: Behält den letzten Zustand bei Stopp bei, während `E_BLINK_TRAIN` immer auf `FALSE` zurückkehrt.
- **Einfache Timer-FBs**: Bieten keine integrierte Begrenzung der Zyklenanzahl (`N`).

🛠️ Zugehörige Übungen

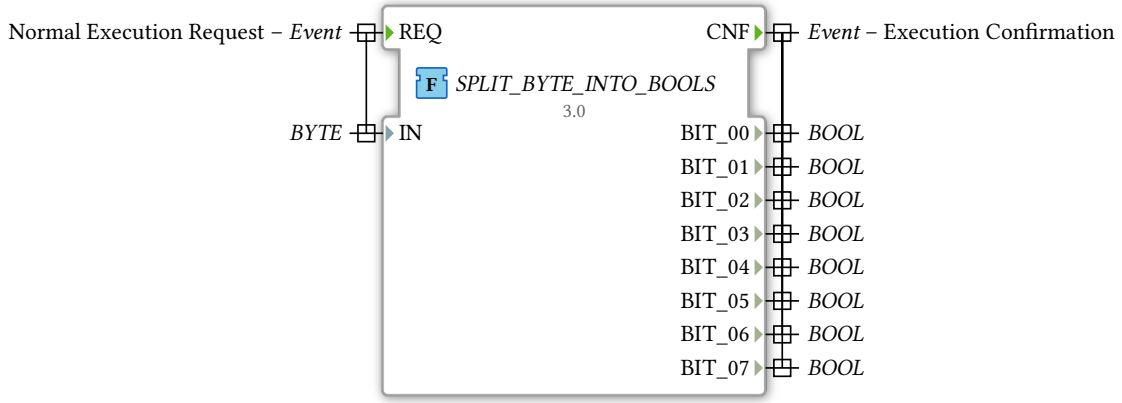
- Uebung_035a3

Fazit

Der `E_BLINK_TRAIN` ist ein vielseitiger Funktionsblock für die Erzeugung von blinkenden Signalen mit konfigurierbaren Zeiten und einer begrenzten Anzahl von Zyklen. Seine Besonderheit liegt in der Rückkehr zu `FALSE` beim Stoppen, was ihn für sicherheitsrelevante Anwendungen prädestiniert. Die klare Schnittstellenstruktur und die interne Verwendung bewährter Timer-FBs machen ihn zuverlässig und einfach einzusetzen.

2.4.9.22 splitting

SPLIT_BYTE_INTO_BOOLS



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `SPLIT_BYTE_INTO_BOOLS` dient dazu, ein Byte in seine einzelnen Bits zu zerlegen und diese als separate BOOL-Werte auszugeben. Dies ist besonders nützlich, wenn mit Hardware gearbeitet wird, die einzelne Bits eines Bytes auswertet oder setzt.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung des Funktionsblocks. Wird dieser Event ausgelöst, werden die Bits des aktuellen Byte-Wertes an `IN` ausgewertet.

Ereignis-Ausgänge

- **CNF**: Bestätigt die Ausführung des Funktionsblocks. Dieser Event wird ausgelöst, nachdem alle Bits ausgewertet und an die entsprechenden Ausgänge weitergeleitet wurden.

Daten-Eingänge

- **IN**: Das Byte, das in seine einzelnen Bits zerlegt werden soll. Der Datentyp ist `BYTE`.

Daten-Ausgänge

- **BIT_00** bis **BIT_07**: Die einzelnen Bits des Bytes, wobei `BIT_00` das niederwertigste Bit (LSB) und `BIT_07` das höchstwertige Bit (MSB) repräsentiert. Der Datentyp aller Ausgänge ist `BOOL`.

Adapter

Der Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock zerlegt das eingehende Byte `IN` in seine 8 Bits und weist diese den entsprechenden BOOL-Ausgängen zu. Jedes Bit wird direkt aus dem Byte extrahiert und als separater BOOL-Wert ausgegeben. Die Auswertung erfolgt bei jedem `REQ`-Event.

Technische Besonderheiten

- Die Bit-Extraktion erfolgt direkt über die Zugriffsnotation `IN.%X0` bis `IN.%X7`, was eine effiziente und schnelle Verarbeitung ermöglicht.
- Der Funktionsblock ist in der Programmiersprache ST (Structured Text) implementiert.

Zustandsübersicht

Der Funktionsblock hat keine internen Zustände. Die Ausführung erfolgt direkt bei Empfang des `REQ`-Events, und die Ergebnisse werden sofort über den `CNF`-Event und die Datenausgänge bereitgestellt.

Anwendungsszenarien

- **Hardware-Steuerung**: Bei der Ansteuerung von digitalen Ein- und Ausgängen, die als Byte organisiert sind, können einzelne Bits einfach ausgewertet werden.
- **Datenverarbeitung**: Bei der Verarbeitung von Byte-Daten, bei denen einzelne Bits eine spezifische Bedeutung haben (z.B. Statusflags).

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Bit-Manipulationsfunktionen bietet dieser Baustein eine spezialisierte und einfache Lösung für die Zerlegung eines Bytes in Bits.
- Andere Bausteine könnten komplexere Operationen wie Bit-Shifts oder Masken erfordern, während dieser Baustein die direkte Extraktion der Bits ermöglicht.

🛠️ Zugehörige Übungen

- Uebung_053

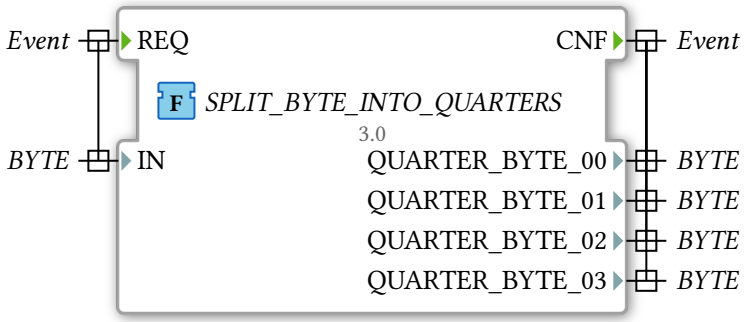
Fazit

Der `SPLIT_BYTE_INTO_BOOLS`-Funktionsblock ist ein einfaches und effizientes Werkzeug zur Zerlegung eines Bytes in seine einzelnen Bits. Er eignet sich besonders für Anwendungen, bei denen eine schnelle und direkte Bit-Extraktion benötigt wird. Durch seine klare Schnittstelle und einfache Funktionsweise ist er leicht in bestehende Systeme integrierbar.

SPLIT_BYTE_INTO_QUARTERS

🎧 Podcast

- QUARTER



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `SPLIT_BYTE_INTO_QUARTERS` dient dazu, ein Byte in vier Viertelbytes (Quarter Bytes) aufzuteilen. Diese Funktion ist besonders nützlich, wenn einzelne Teile eines Bytes separat verarbeitet werden müssen, z.B. bei der Dekodierung von Daten oder bei der Bitmanipulation.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Startet die Verarbeitung des Eingabebytes. Wird mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- `CNF`: Signalisiert den Abschluss der Verarbeitung und liefert die vier Viertelbytes. Wird mit den Daten-Ausgängen `QUARTER_BYTE_00`, `QUARTER_BYTE_01`, `QUARTER_BYTE_02` und `QUARTER_BYTE_03` verknüpft.

Daten-Eingänge

- `IN` (BYTE): Das Eingabebyte, das in vier Viertelbytes aufgeteilt werden soll.

Daten-Ausgänge

- `QUARTER_BYTE_00` (BYTE): Das erste Viertelbyte (Bits 0-1).

- `QUARTER_BYTE_01` (BYTE): Das zweite Viertelbyte (Bits 2-3).
- `QUARTER_BYTE_02` (BYTE): Das dritte Viertelbyte (Bits 4-5).
- `QUARTER_BYTE_03` (BYTE): Das vierte Viertelbyte (Bits 6-7).

Adapter

Es werden keine Adapter verwendet.

Funktionsweise

Der Funktionsblock extrahiert die vier Viertelbytes aus dem Eingabebyte `IN` mithilfe von Bitoperationen:

1. Jedes Viertelbyte wird durch eine bitweise UND-Operation mit einer entsprechenden Maske isoliert.
2. Anschließend wird das Ergebnis durch eine Rechtsverschiebung (`SHR`) in die richtige Position gebracht.
3. Die verwendeten Masken und Verschiebungswerte sind in der Konstanten-Datei `quarterconst` definiert.

Technische Besonderheiten

- Die Bitoperationen sind effizient und schnell, da sie direkt auf Prozessorebene ausgeführt werden.
- Die Verwendung von Konstanten (`quarterconst`) ermöglicht eine flexible Anpassung der Masken und Verschiebungswerte.

Zustandsübersicht

Der Funktionsblock hat keinen internen Zustand. Die Verarbeitung erfolgt ausschließlich bei Auftreten des `REQ`-Ereignisses.

Anwendungsszenarien

- Dekodierung von Daten, die in einem Byte gepackt sind.
- Extraktion von einzelnen Steuerbits aus einem Statusbyte.
- Verarbeitung von Daten in Kommunikationsprotokollen, bei denen Informationen in Viertelbytes codiert sind.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Bitmanipulations-Funktionsblöcken bietet `SPLIT_BYTE_INTO_QUARTERS` eine spezialisierte und optimierte Lösung für die Aufteilung eines Bytes in vier Teile.
- Andere Blöcke wie `EXTRACT_BITS` oder `BIT_SLICE` sind allgemeiner, erfordern aber mehr Konfiguration und sind möglicherweise weniger effizient.

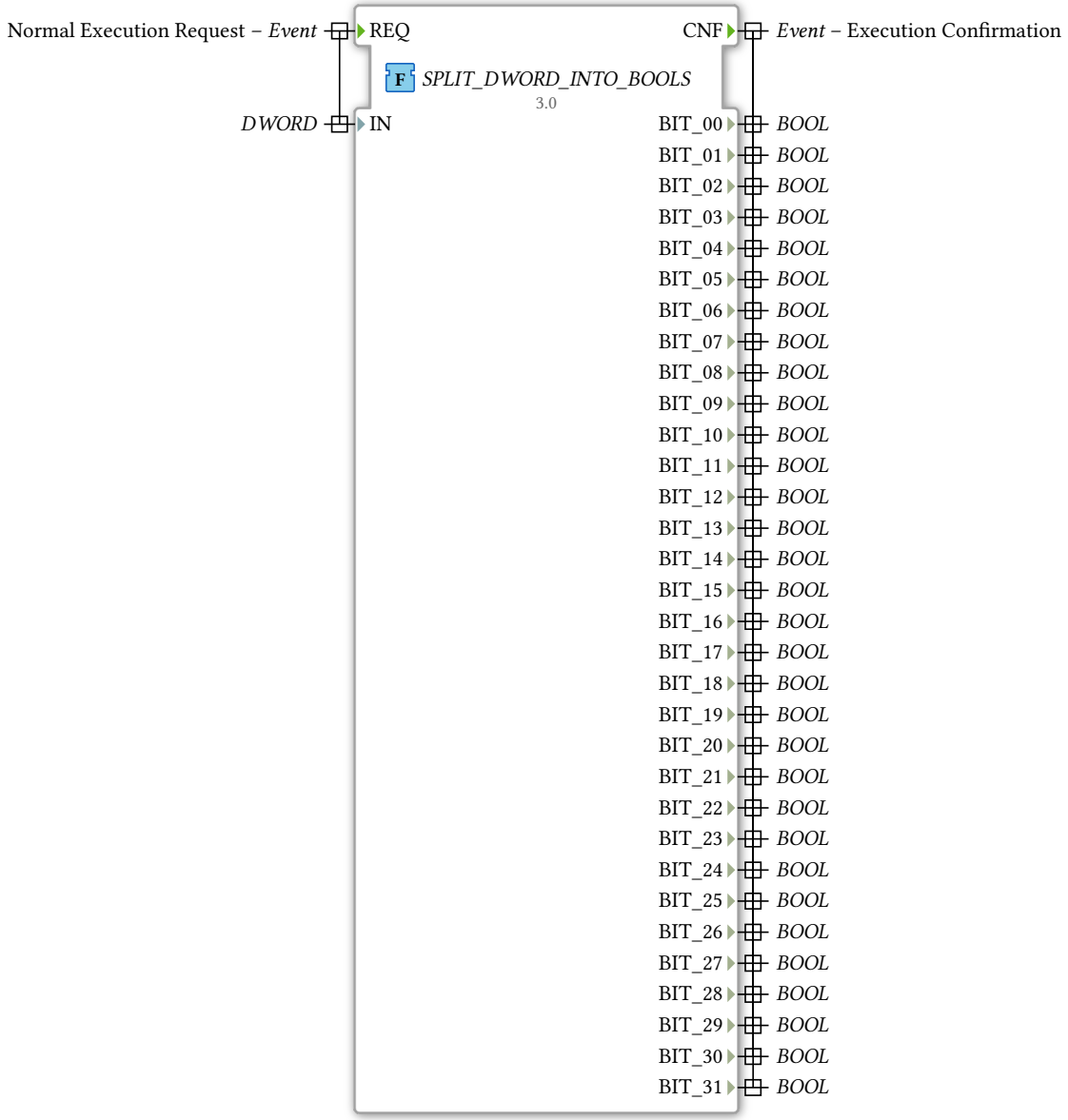
🛠️ Zugehörige Übungen

- Uebung_056

Fazit

Der `SPLIT_BYTE_INTO_QUARTERS`-Funktionsblock ist eine effiziente und spezialisierte Lösung für die Aufteilung eines Bytes in vier Viertelbytes. Durch die Verwendung von Bitoperationen und Konstanten ist er schnell und flexibel einsetzbar. Ideal für Anwendungen, bei denen Daten in kompakter Form vorliegen und separat verarbeitet werden müssen.

SPLIT_DWORD_INTO_BOOLS



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `SPLIT_DWORD_INTO_BOOLS` dient dazu, ein 32-Bit-DWORD in seine einzelnen Bits aufzuteilen und diese als separate BOOL-Werte auszugeben. Dies ist besonders nützlich, wenn komprimierte Daten in einer Steuerung verarbeitet werden müssen und die einzelnen Bits separat angesprochen werden sollen.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung des Funktionsblocks. Wird dieser Eingang aktiviert, werden die Bits des Eingangs-DWORD ausgewertet und an den Ausgängen bereitgestellt.

Ereignis-Ausgänge

- **CNF**: Bestätigt die erfolgreiche Ausführung des Funktionsblocks. Dieses Ereignis wird zusammen mit den aktualisierten BOOL-Ausgängen ausgelöst.

Daten-Eingänge

- **IN**: Das Eingangs-DWORD, das in 32 einzelne Bits aufgeteilt werden soll.

Daten-Ausgänge

- **BIT_00** bis **BIT_31**: Die 32 einzelnen Bits des Eingangs-DWORD, jeweils als BOOL-Wert ausgegeben. Jeder Ausgang repräsentiert ein Bit des DWORD, wobei `BIT_00` dem niederwertigsten Bit (LSB) und `BIT_31` dem höchstwertigen Bit (MSB) entspricht.

Adapter

Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise

Der Funktionsblock extrahiert jedes Bit des Eingangs-DWORD `IN` und weist es den entsprechenden BOOL-Ausgängen zu. Die Zuordnung erfolgt direkt:

- `BIT_00` = Bit 0 (LSB)
- `BIT_01` = Bit 1
- ...
- `BIT_31` = Bit 31 (MSB)

Bei Aktivierung des `REQ`-Ereignisses werden alle Bits ausgewertet, und das `CNF`-Ereignis wird ausgelöst, sobald die Ausgänge aktualisiert sind.

Technische Besonderheiten

- Der Funktionsblock ist in ST (Structured Text) implementiert und verwendet direkte Bit-Zugriffe auf das DWORD (`IN.%X0` bis `IN.%X31`).
- Die Ausgänge werden synchron mit dem `CNF`-Ereignis aktualisiert, was eine deterministische Verarbeitung ermöglicht.

Zustandsübersicht

Der Funktionsblock hat keinen internen Zustand. Bei jedem `REQ`-Ereignis werden die Ausgänge neu berechnet.

Anwendungsszenarien

- Verarbeitung von komprimierten Statusinformationen, z. B. aus einem Feldbus oder einer Schnittstelle.
- Dekodierung von Bitmasken in Steuerungsanwendungen.
- Aufteilung von DWORD-Werten für individuelle Bit-Manipulationen.

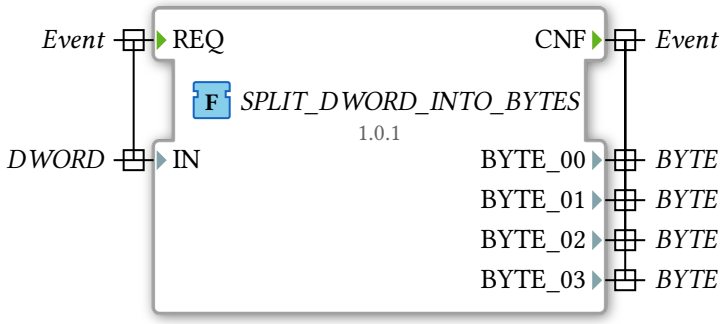
⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Bit-Manipulations-Funktionen bietet dieser Baustein eine direkte und effiziente Methode zur Extraktion aller Bits eines DWORD.
- Ähnliche Bausteine könnten nur Teile eines DWORD verarbeiten oder zusätzliche Logik enthalten, was hier nicht der Fall ist.

Fazit

Der `SPLIT_DWORD_INTO_BOOLS`-Funktionsblock ist ein spezialisiertes Werkzeug zur effizienten Aufteilung eines DWORD in seine einzelnen Bits. Durch seine einfache und deterministische Funktionsweise eignet er sich ideal für Anwendungen, in denen Bit-Manipulationen erforderlich sind.

SPLIT_DWORD_INTO_BYTES



• – ◦ • – ◦ • – ◦ •

Einleitung

Die Funktion `SPLIT_DWORD_INTO_BYTES` zerlegt einen `DWORD`-Wert in seine vier `BYTE`-Anteile. Sie ist das Gegenstück zu `ASSEMBLE_DWORD_FROM_BYTES`.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Zerlegung. Verbundene Daten: `IN`.

Ereignis-Ausgänge

- **CNF**: Signalisiert die abgeschlossene Zerlegung. Verbundene Daten: `BYTE_00` bis `BYTE_03`.

Daten-Eingänge

- **IN** (`DWORD`): Zu zerlegender Wert

Daten-Ausgänge

- **BYTE_00** (`BYTE`): niedrigstwertiges Byte (`%B0`)
- **BYTE_01** (`BYTE`): Byte `%B1`
- **BYTE_02** (`BYTE`): Byte `%B2`
- **BYTE_03** (`BYTE`): höchstwertiges Byte (`%B3`)

Adapter

Keine Adapter vorhanden.

Funktionsweise

Die Funktion weist jedem Ausgang das entsprechende Byte des Eingangs zu:

- `BYTE_00 := IN.%B0`
- `BYTE_01 := IN.%B1`
- `BYTE_02 := IN.%B2`
- `BYTE_03 := IN.%B3`

Technische Besonderheiten

- Implementiert als ST-Funktion (`FUNCTION`) mit `VAR_OUTPUT`, Paket `eclipse4diac::utils::splitting`
- Der Zugriff auf einzelne Bytes erfolgt über die partielle Zugriffssyntax `%B0` bis `%B3`
- Keine eigene Zustandshaltung

Anwendungsszenarien

- Zerlegen von 32-Bit-Werten in Einzelbytes, z. B. für Protokolle oder CAN-Daten

⚖️ Vergleich mit ähnlichen Bausteinen

- `SPLIT_WORD_INTO_BYTES` zerlegt ein `WORD` in zwei Bytes, `SPLIT_DWORD_INTO_WORDS` ein `DWORD` in zwei Worte.

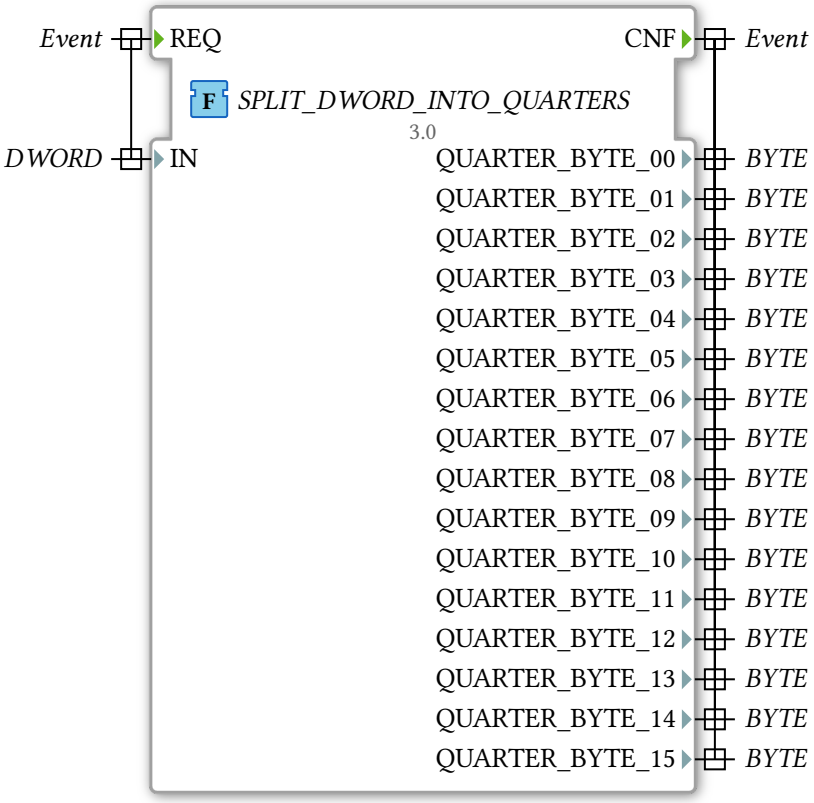
Fazit

`SPLIT_DWORD_INTO_BYTES` zerlegt einen `DWORD` byteweise in vier `BYTE`-Ausgänge.

SPLIT_DWORD_INTO_QUARTERS

🎧 Podcast

- QUARTER



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `SPLIT_DWORD_INTO_QUARTERS` dient dazu, ein 32-Bit-Wort (DWORD) in seine 16 einzelnen 4-Bit-Blöcke (QUARTER_BYTE) aufzuteilen. Jeder dieser 4-Bit-Blöcke wird als separate BYTE-Variable ausgegeben. Der Block ist besonders nützlich, wenn einzelne Teile eines DWORDs separat verarbeitet oder analysiert werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Startet die Verarbeitung des Eingangs-DWORD.

Ereignis-Ausgänge

- `CNF`: Signalisiert die erfolgreiche Verarbeitung und die Bereitstellung aller 16 QUARTER_BYTE-Ausgänge.

Daten-Eingänge

- `IN`: Das Eingangs-DWORD, das in 16 QUARTER_BYTE aufgeteilt wird.

Daten-Ausgänge

- `QUARTER_BYTE_00` bis `QUARTER_BYTE_15`: Die 16 einzelnen 4-Bit-Blöcke des Eingangs-DWORDs, jeweils als BYTE ausgegeben.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock extrahiert die 16 QUARTER_BYTE aus dem Eingangs-DWORD durch bitweise Operationen. Jeder QUARTER_BYTE wird durch eine Kombination aus bitweiser UND-Verknüpfung und Rechtsverschiebung isoliert und anschließend in einen BYTE-Wert konvertiert. Die verwendeten Konstanten (`DWORD_QUARTER_XX` und `SHIFT_QUARTER_XX`) sind in einer separaten Konstante-Datei (`quarterconst`) definiert.

Technische Besonderheiten

- Der Block verwendet bitweise Operationen, um die einzelnen 4-Bit-Blöcke zu isolieren.
- Die Ausgabewerte sind als BYTE definiert, obwohl sie nur 4 Bit enthalten. Die oberen 4 Bit jedes BYTE sind daher immer 0.
- Die Verwendung von vordefinierten Konstanten (`quarterconst`) ermöglicht eine flexible Anpassung der Bitpositionen.

Zustandsübersicht

Der Funktionsblock hat keinen internen Zustand. Die Verarbeitung erfolgt bei jedem `REQ`-Ereignis neu.

Anwendungsszenarien

- Verarbeitung von Datenpaketen, bei denen einzelne 4-Bit-Blöcke eines DWORDs ausgewertet werden müssen.
- Dekodierung von Status- oder Steuerinformationen, die in einem DWORD kodiert sind.
- Aufteilung von Daten für die weitere Verarbeitung in nachgelagerten Systemen.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Bitmanipulations-Blöcken bietet `SPLIT_DWORD_INTO_QUARTERS` eine spezialisierte Funktionalität für die Aufteilung eines DWORDs in 4-Bit-Blöcke.
- Andere Blöcke wie `SPLIT_DWORD_INTO_BYTES` teilen ein DWORD in 8-Bit-Blöcke auf, während dieser Block eine feinere Granularität bietet.

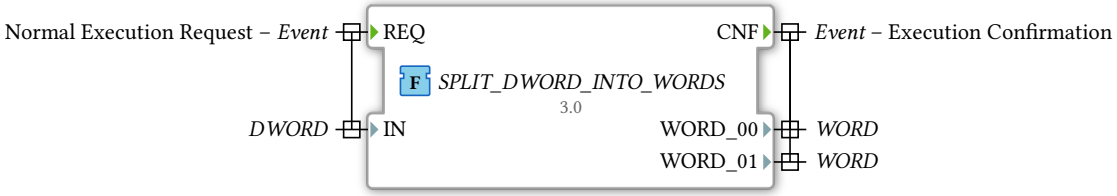
🛠️ Zugehörige Übungen

- Uebung_060

Fazit

Der `SPLIT_DWORD_INTO_QUARTERS`-Funktionsblock ist ein spezialisiertes Werkzeug für die Aufteilung eines 32-Bit-Wortes in 16 einzelne 4-Bit-Blöcke. Durch seine einfache und effiziente Implementierung eignet er sich ideal für Anwendungen, bei denen eine detaillierte Analyse oder Verarbeitung von DWORD-Daten erforderlich ist.

SPLIT_DWORD_INTO_WORDS



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `SPLIT_DWORD_INTO_WORDS` dient dazu, ein 32-Bit DWORD in zwei 16-Bit WORDS aufzuteilen. Diese Funktionalität ist besonders nützlich, wenn mit Daten gearbeitet wird, die als DWORD vorliegen, aber in einzelnen WORD-Teilen weiterverarbeitet werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge
- **REQ**: Startet die Ausführung des Funktionsblocks. Wird mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge
- **CNF**: Signalisiert die erfolgreiche Ausführung des Funktionsblocks. Wird mit den Daten-Ausgängen `WORD_00` und `WORD_01` verknüpft.

Daten-Eingänge
- **IN**: DWORD (32-Bit), das in zwei WORDS aufgeteilt werden soll.

Daten-Ausgänge
- **WORD_00**: Erstes WORD (16-Bit) des DWORD.
- **WORD_01**: Zweites WORD (16-Bit) des DWORD.

Adapter
Dieser Funktionsblock verwendet keine Adapter.

Funktionsweise
Der Funktionsblock nimmt ein DWORD als Eingabe und teilt es in zwei WORDS auf:

- `WORD_00` enthält die unteren 16 Bits des DWORD (`IN.%W0`).
- `WORD_01` enthält die oberen 16 Bits des DWORD (`IN.%X1`).

Die Aufteilung erfolgt bei jedem `REQ`-Ereignis, und das Ergebnis wird mit dem `CNF`-Ereignis bestätigt.

Technische Besonderheiten
- Die Implementierung erfolgt in Structured Text (ST).
- Der Funktionsblock ist Teil des Pakets `eclipse4diac::utils::splitting`.

Zustandsübersicht
Der Funktionsblock hat keinen internen Zustand. Die Verarbeitung erfolgt bei jedem `REQ`-Ereignis unmittelbar.

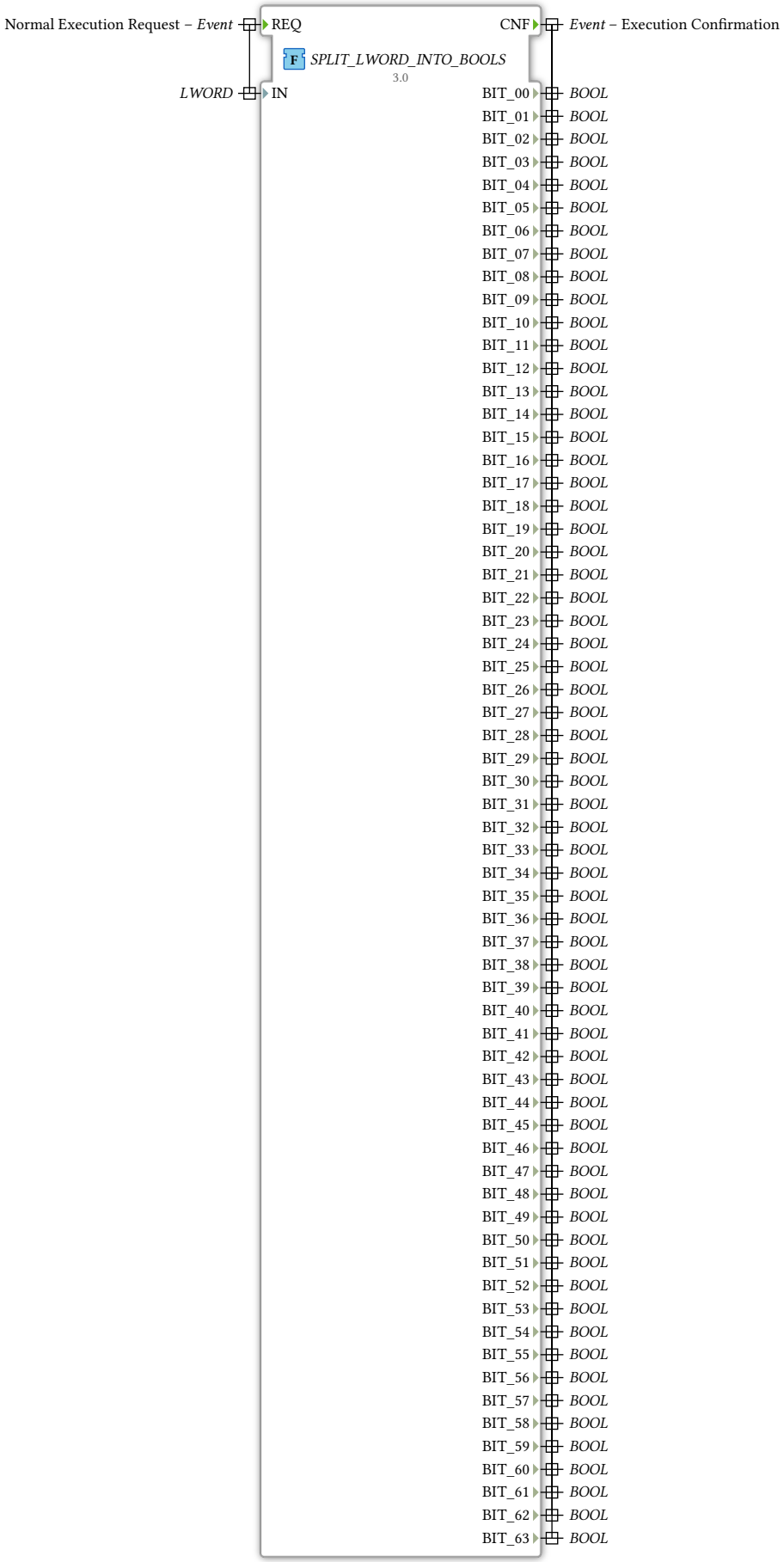
Anwendungsszenarien
- Verarbeitung von DWORD-Daten, die aus Hardware-Registern oder Kommunikationsprotokollen stammen.
- Aufteilung von 32-Bit Werten für die weitere Verarbeitung in 16-Bit Systemen.

⚖️ Vergleich mit ähnlichen Bausteinen
- Im Gegensatz zu generischen Bit-Manipulationsblöcken ist `SPLIT_DWORD_INTO_WORDS` spezialisiert auf die Aufteilung von DWORD in WORDS.
- Ähnliche Blöcke könnten zusätzliche Flexibilität bieten, wie z.B. die Auswahl der zu extrahierenden Bits, sind aber oft komplexer in der Handhabung.

Fazit
Der `SPLIT_DWORD_INTO_WORDS`-Funktionsblock bietet eine einfache und effiziente Möglichkeit, DWORD-Daten in ihre WORD-Komponenten aufzuteilen. Seine spezialisierte Funktionalität macht ihn besonders nützlich für Anwendungen, die regelmäßig mit solchen Datenumwandlungen arbeiten müssen.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

SPLIT_LWORD_INTO_BOOLS


Normal Execution Request – Event
REQ
CNF
Event – Execution Confirmation
SPLIT_LWORD_INTO_BOOLS
3.0
LWORD
IN
BIT_00 BOOL
BIT_01 BOOL
BIT_02 BOOL
BIT_03 BOOL
BIT_04 BOOL
BIT_05 BOOL
BIT_06 BOOL
BIT_07 BOOL
BIT_08 BOOL
BIT_09 BOOL
BIT_10 BOOL
BIT_11 BOOL
BIT_12 BOOL
BIT_13 BOOL
BIT_14 BOOL
BIT_15 BOOL
BIT_16 BOOL
BIT_17 BOOL
BIT_18 BOOL
BIT_19 BOOL
BIT_20 BOOL
BIT_21 BOOL
BIT_22 BOOL
BIT_23 BOOL
BIT_24 BOOL
BIT_25 BOOL
BIT_26 BOOL
BIT_27 BOOL
BIT_28 BOOL
BIT_29 BOOL
BIT_30 BOOL
BIT_31 BOOL
BIT_32 BOOL
BIT_33 BOOL
BIT_34 BOOL
BIT_35 BOOL
BIT_36 BOOL
BIT_37 BOOL
BIT_38 BOOL
BIT_39 BOOL
BIT_40 BOOL
BIT_41 BOOL
BIT_42 BOOL
BIT_43 BOOL
BIT_44 BOOL
BIT_45 BOOL
BIT_46 BOOL
BIT_47 BOOL
BIT_48 BOOL
BIT_49 BOOL
BIT_50 BOOL
BIT_51 BOOL
BIT_52 BOOL
BIT_53 BOOL
BIT_54 BOOL
BIT_55 BOOL
BIT_56 BOOL
BIT_57 BOOL
BIT_58 BOOL
BIT_59 BOOL
BIT_60 BOOL
BIT_61 BOOL
BIT_62 BOOL
BIT_63 BOOL

• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `SPLIT_LWORD_INTO_BOOLS` dient zur Zerlegung eines 64-Bit-LWORD-Datentyps in seine einzelnen Bits. Jedes Bit wird als separater BOOL-Ausgang bereitgestellt. Dieser Block ist besonders nützlich, wenn einzelne Bits eines langen Wortes ausgewertet oder verarbeitet werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ` (Event): Startet die Zerlegung des LWORD in einzelne Bits. Wird mit dem Eingang `IN` verknüpft.

Ereignis-Ausgänge

- `CNF` (Event): Bestätigt die erfolgreiche Zerlegung des LWORD. Wird mit allen 64 BOOL-Ausgängen verknüpft.

Daten-Eingänge

- `IN` (LWORD): Das 64-Bit-Wort, das in einzelne Bits zerlegt werden soll.

Daten-Ausgänge

- `BIT_00` bis `BIT_63` (BOOL): Die einzelnen Bits des Eingangs-LWORD, wobei `BIT_00` das niederwertigste Bit (LSB) und `BIT_63` das höchstwertige Bit (MSB) darstellt.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock extrahiert jedes Bit des Eingangs-LWORD (`IN`) und weist es den entsprechenden BOOL-Ausgängen zu. Die Extraktion erfolgt über direkte Bit-Zugriffe (`IN.%X0` bis `IN.%X63`). Bei Auslösung des `REQ`-Ereignisses werden alle Bits aktualisiert und das `CNF`-Ereignis bestätigt die erfolgreiche Verarbeitung.

Technische Besonderheiten

- Verarbeitet 64-Bit-Daten (LWORD).
- Jedes Bit wird einzeln als BOOL ausgegeben.
- Keine zusätzliche Logik oder Filterung – direkte Bit-Zuordnung.

Zustandsübersicht

Der Block hat keinen internen Zustand. Die Zerlegung erfolgt bei jedem `REQ`-Ereignis neu.

Anwendungsszenarien

- Auswertung von Bitmasken oder Statusregistern.
- Verarbeitung von digitalen Signalen, die als LWORD kodiert sind.
- Debugging oder Monitoring von einzelnen Bits in langen Datenwörtern.

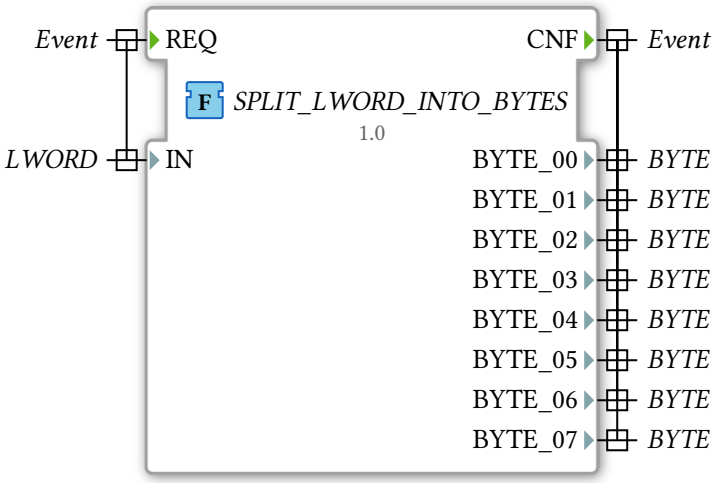
⚖️ Vergleich mit ähnlichen Bausteinen

- Ähnliche Blöcke wie `SPLIT_WORD_INTO_BOOLS` oder `SPLIT_DWORD_INTO_BOOLS` arbeiten mit kürzeren Datentypen (WORD bzw. DWORD).
- Dieser Block ist speziell für 64-Bit-Daten optimiert.

Fazit

Der `SPLIT_LWORD_INTO_BOOLS`-Block ist ein effizientes Werkzeug zur Zerlegung von 64-Bit-Daten in einzelne Bits. Seine einfache und direkte Funktionsweise macht ihn ideal für Anwendungen, bei denen eine detaillierte Bit-Analyse erforderlich ist.

SPLIT_LWORD_INTO_BYTES



• – ◦ • – ◦ • – ◦ •

Einleitung

Die Funktion `SPLIT_LWORD_INTO_BYTES` zerlegt ein 64-Bit-Langwort (`LWORD`) in seine acht Bytes (`BYTE`).

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Startet die Verarbeitung. Verbundene Daten: `IN`

Ereignis-Ausgänge

- `CNF`: Signalisiert die Verarbeitung. Verbundene Daten: `BYTE_00` bis `BYTE_07`

Daten-Eingänge

- `IN`: Das zu zerlegende Langwort (`LWORD`)

Daten-Ausgänge

- `BYTE_00`: Byte 0 (`BYTE`), niederwertigstes Byte (Bits 0 bis 7)
- `BYTE_01`: Byte 1 (`BYTE`), Bits 8 bis 15
- `BYTE_02`: Byte 2 (`BYTE`), Bits 16 bis 23
- `BYTE_03`: Byte 3 (`BYTE`), Bits 24 bis 31
- `BYTE_04`: Byte 4 (`BYTE`), Bits 32 bis 39
- `BYTE_05`: Byte 5 (`BYTE`), Bits 40 bis 47
- `BYTE_06`: Byte 6 (`BYTE`), Bits 48 bis 55
- `BYTE_07`: Byte 7 (`BYTE`), höchstwertiges Byte (Bits 56 bis 63)

Adapter

Diese Funktion verwendet keine Adapter.

Funktionsweise

Die Funktion liest die acht Bytes über den Teilzugriff `.%B0` bis `.%B7` aus dem Eingang `IN`:

```
BYTE_00 := IN.%B0;
BYTE_01 := IN.%B1;
BYTE_02 := IN.%B2;
BYTE_03 := IN.%B3;
BYTE_04 := IN.%B4;
BYTE_05 := IN.%B5;
BYTE_06 := IN.%B6;
BYTE_07 := IN.%B7;
```

`BYTE_00` ist damit das niederwertigste und `BYTE_07` das höchstwertige Byte des `LWORD`.

Technische Besonderheiten

- Implementierung in Structured Text (ST) als `FUNCTION` ohne internen Zustand
- Teil des Pakets `eclipse4diac::utils::splitting`
- Version 1.0 (2026-05-16)

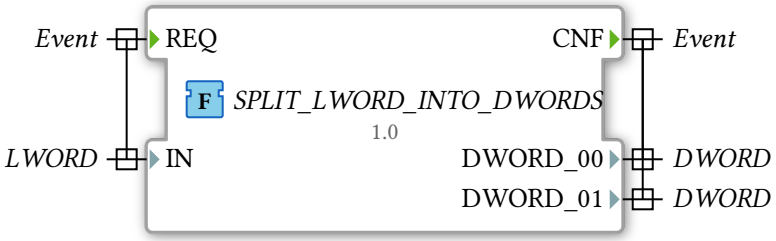
Anwendungsszenarien

- Zerlegen eines 64-Bit-Werts in einzelne Bytes, z. B. für Telegramme oder Bus-Nachrichten (CAN: 8 Datenbytes)
- Analyse von 64-Bit-Bitmustern byteweise

Fazit

`SPLIT_LWORD_INTO_BYTES` extrahiert die acht Bytes eines `LWORD`, beginnend mit dem niederwertigsten.

SPLIT_LWORD_INTO_DWORDS



• – ◦ • – ◦ • – ◦ •

Einleitung

Die Funktion `SPLIT_LWORD_INTO_DWORDS` extrahiert die zwei 32-Bit-Doppelwörter (`DWORD`) aus einem 64-Bit-Langwort (`LWORD`).

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Startet die Verarbeitung. Verbundene Daten: `IN`

Ereignis-Ausgänge

- `CNF`: Signalisiert die Verarbeitung. Verbundene Daten: `DWORD_00`, `DWORD_01`

Daten-Eingänge

- `IN` (`LWORD`): Zu zerlegendes Langwort

Daten-Ausgänge

- `DWORD_00` (`DWORD`): Niederwertiges Doppelwort (Bits 0 bis 31)
- `DWORD_01` (`DWORD`): Höherwertiges Doppelwort (Bits 32 bis 63)

Adapter

Diese Funktion verwendet keine Adapter.

Funktionsweise

Die Funktion weist die Teile über den Teilzugriff zu:

```
DWORD_00 := IN.%D0;
DWORD_01 := IN.%D1;
```

Technische Besonderheiten

- Implementierung in Structured Text (ST) als `FUNCTION` ohne internen Zustand
- Teil des Pakets `eclipse4diac::utils::splitting`
- Die Umkehrung bildet `ASSEMBLE_LWORD_FROM_WORDS` bzw. die `ASSEMBLE_*`-Funktionen im Paket `assembling` (z. B. `ASSEMBLE_LWORD_FROM_WORDS` für `WORD`-Teile).

Anwendungsszenarien

- Zerlegen eines 64-Bit-Werts in zwei 32-Bit-Werte, z. B. für Fieldbus- oder Registerzugriffe

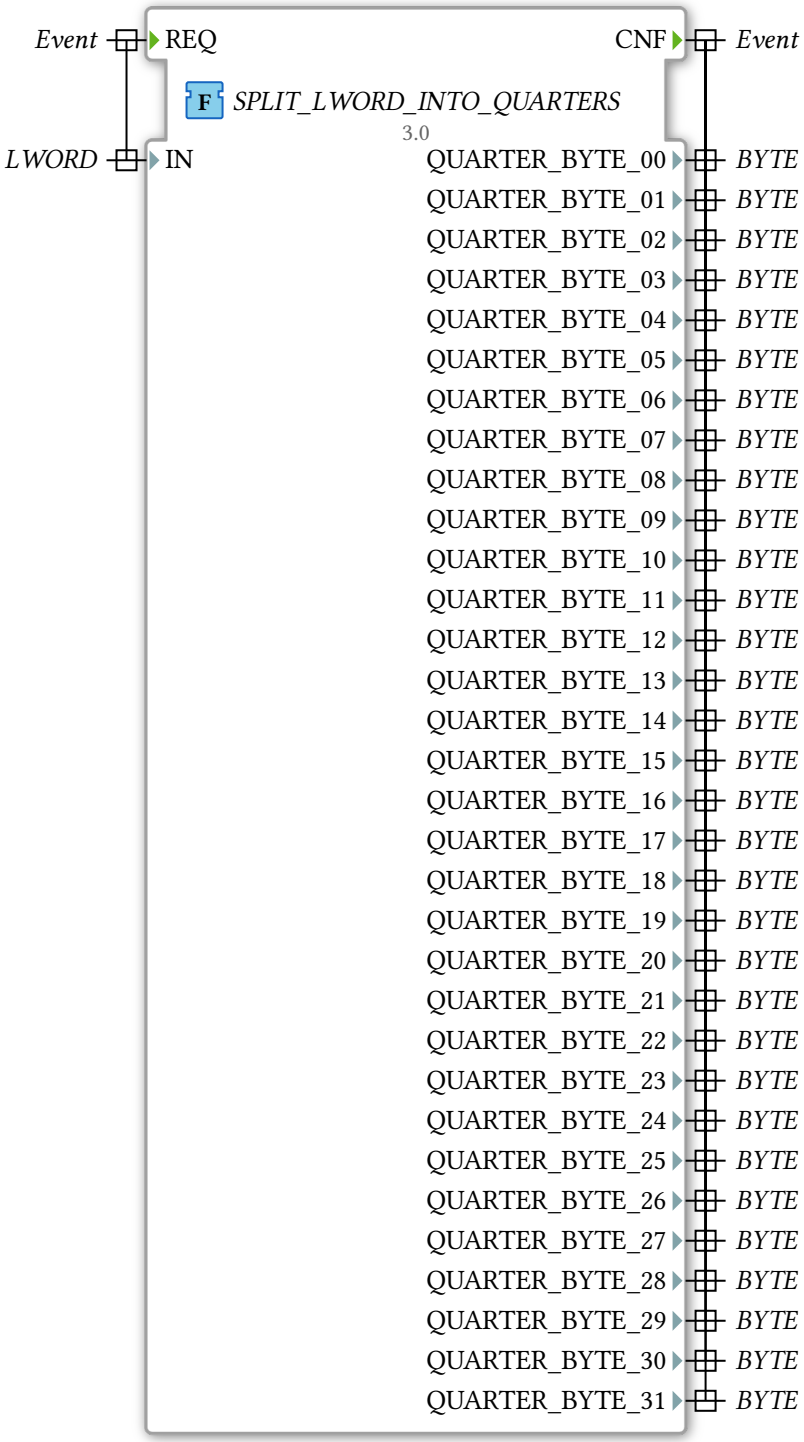
Fazit

`SPLIT_LWORD_INTO_DWORDS` teilt ein `LWORD` in sein niederwertiges und sein höherwertiges `DWORD`.

SPLIT_LWORD_INTO_QUARTERS

Podcast

- QUARTER



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `SPLIT_LWORD_INTO_QUARTERS` dient zur Extraktion der 32 Quarter Bytes (je 8 Bit) aus einem LWORD (64 Bit). Dies ermöglicht die gezielte Verarbeitung einzelner Byte-Segmente innerhalb eines langen Wortes.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Startet die Verarbeitung des Eingangswertes `IN`.

Ereignis-Ausgänge

- `CNF`: Bestätigt die erfolgreiche Verarbeitung und liefert alle 32 Quarter Bytes.

Daten-Eingänge

- `IN` (LWORD): Das 64-Bit-Eingangswort, das in Quarter Bytes aufgeteilt werden soll.

Daten-Ausgänge

32 Ausgänge vom Typ BYTE:

- `QUARTER_BYTE_00` bis `QUARTER_BYTE_31`: Die extrahierten 8-Bit-Segmente des Eingangswortes.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock nutzt Bitoperationen (AND und Rechts-Shift) in Kombination mit Konstanten (`quarterconst::LWORD_QUARTER_xx` und `quarterconst::SHIFT_QUARTER_xx`), um jedes der 32 Quarter Bytes aus dem Eingangs-LWORD zu isolieren. Jedes Quarter Byte wird anschließend in einen separaten BYTE-Ausgang geschrieben.

Technische Besonderheiten

- Verwendet spezielle Bitmasken und Verschiebungskonstanten für präzise Extraktion
- Arbeitet mit dem größten IEC-Datentyp LWORD (64 Bit)
- Liefert alle 32 Quarter Bytes parallel in einem Durchlauf

Zustandsübersicht

1. **Idle**: Wartet auf REQ-Ereignis
2. **Processing**: Extrahiert alle Quarter Bytes
3. **Output**: Trigger CNF-Ereignis mit allen Ausgangswerten

Anwendungsszenarien

- Verarbeitung von langen Datenwörtern in Kommunikationsprotokollen
- Extraktion einzelner Parameter aus gepackten Datenstrukturen
- Bitweise Analyse von Sensor- oder Steuerdaten

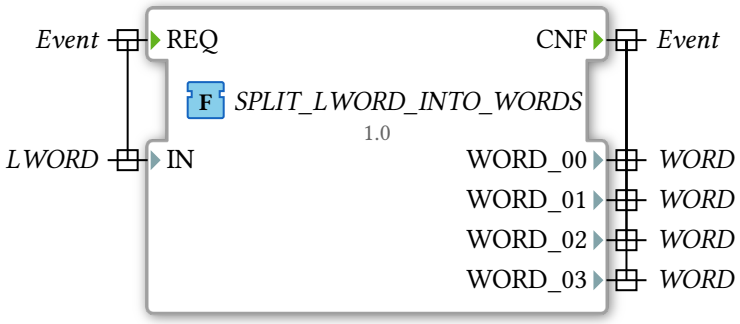
⚖️ Vergleich mit ähnlichen Bausteinen

- Im Gegensatz zu einfachen BYTE-Extraktionsblöcken kann dieser FB alle 32 Segmente gleichzeitig verarbeiten
- Effizienter als mehrere hintereinandergeschaltete Extraktionsblöcke
- Spezialisiert auf die Verarbeitung von LWORD-Datentypen

Fazit

Der `SPLIT_LWORD_INTO_QUARTERS`-Funktionsblock bietet eine hochoptimierte Lösung für die Aufteilung von 64-Bit-Wörtern in ihre einzelnen Byte-Komponenten. Durch die parallele Verarbeitung aller Segmente eignet er sich besonders für Anwendungen, die eine effiziente Verarbeitung großer Datenwörter erfordern.

SPLIT_LWORD_INTO_WORDS



• – ◦ • – ◦ • – ◦ •

Einleitung

Die Funktion `SPLIT_LWORD_INTO_WORDS` extrahiert die vier `WORD`-Werte aus einem `LWORD`. Sie ist nützlich, wenn ein 64-Bit-Wert (z. B. aus einem Kommunikationsprotokoll) in 16-Bit-Teile zerlegt werden muss.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: startet die Verarbeitung (assoziiert mit `IN`)

Ereignis-Ausgänge

- **CNF**: signalisiert den Abschluss (assoziiert mit `WORD_00`..`WORD_03`)

Daten-Eingänge

- **IN** (LWORD): aufzuteilender 64-Bit-Wert

Daten-Ausgänge

- **WORD_00** (WORD): erstes Wort, Bits 0-15 (`IN.%W0`)
- **WORD_01** (WORD): zweites Wort, Bits 16-31 (`IN.%W1`)
- **WORD_02** (WORD): drittes Wort, Bits 32-47 (`IN.%W2`)
- **WORD_03** (WORD): viertes Wort, Bits 48-63 (`IN.%W3`)

Adapter

Es werden keine Adapter verwendet.

Funktionsweise

Die Funktion weist die vier Wörter per Partial-Access zu: `WORD_00 := IN.%W0;`, `WORD_01 := IN.%W1;`, `WORD_02 := IN.%W2;`, `WORD_03 := IN.%W3;`.

Technische Besonderheiten

- Reine Bit-Zugriffe ohne Rechenaufwand; keine Zustände
- Paket `eclipse4diac::utils::splitting`, Version 1.0 (2026-05-16)

Anwendungsszenarien

- Zerlegen von 64-Bit-Werten (z. B. aus OPC-UA oder CAN-Daten) in 16-Bit-Register

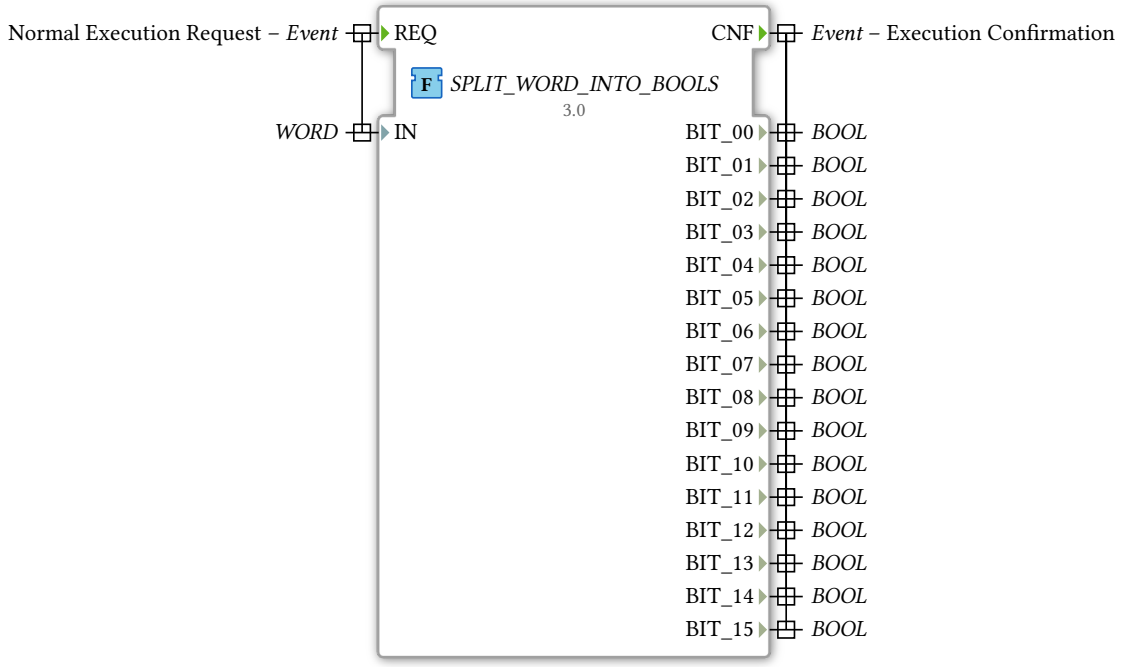
⚖️ Vergleich mit ähnlichen Bausteinen

Analoge Bausteine zerlegen kleinere Werte, etwa `SPLIT_BYTE_INTO_QUARTERS` (Byte in vier Viertelbytes) oder `SPLIT_DWORD_INTO_BOOLS`.

Fazit

`SPLIT_LWORD_INTO_WORDS` zerlegt ein `LWORD` in vier `WORD`-Werte und liefert sie mit einem einzigen `CNF`-Ereignis.

SPLIT_WORD_INTO_BOOLS



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `SPLIT_WORD_INTO_BOOLS` dient dazu, ein 16-Bit-Wort in seine einzelnen Bits zu zerlegen. Jedes Bit des Eingangswortes wird auf einen separaten BOOL-Ausgang abgebildet. Dies ist besonders nützlich, wenn einzelne Bits eines Wortes weiterverarbeitet oder überwacht werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ` (Normal Execution Request): Startet die Zerlegung des Eingangswortes. Wird mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- `CNF` (Execution Confirmation): Bestätigt die erfolgreiche Zerlegung des Wortes. Wird mit allen 16 BOOL-Ausgängen verknüpft.

Daten-Eingänge

- `IN` (WORD): Das 16-Bit-Wort, das in einzelne Bits zerlegt werden soll.

Daten-Ausgänge

- `BIT_00` bis `BIT_15` (BOOL): Die einzelnen Bits des Eingangswortes, wobei `BIT_00` das niederwertigste Bit (LSB) und `BIT_15` das höchstwertige Bit (MSB) darstellt.

Adapter

- Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock zerlegt das Eingangswort `IN` in seine 16 einzelnen Bits und weist diese den entsprechenden BOOL-Ausgängen zu. Die Zuweisung erfolgt direkt über die Bit-Adressierung des WORD-Datentyps (z.B. `IN.%X0` für das erste Bit). Bei Auslösung des `REQ`-Ereignisses werden alle Bits extrahiert und das `CNF`-Ereignis bestätigt die erfolgreiche Ausführung.

Technische Besonderheiten

- Die Bit-Extraktion erfolgt direkt über die Hardware-nahe Bit-Adressierung, was eine effiziente Verarbeitung ermöglicht.
- Der Funktionsblock ist deterministisch und hat keine internen Zustände.

Zustandsübersicht

- **Idle**: Wartet auf das `REQ`-Ereignis.
- **Processing**: Zerlegt das Wort und aktualisiert die Ausgänge.
- **Complete: Löst das CNF`-Ereignis aus und kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- Verarbeitung von digitalen Eingangssignalen, die als WORD übertragen werden.
- Dekodierung von Status- oder Steuerworten in Steuerungssystemen.
- Schnittstelle zwischen Wort-basierten und Bit-basierten Systemkomponenten.

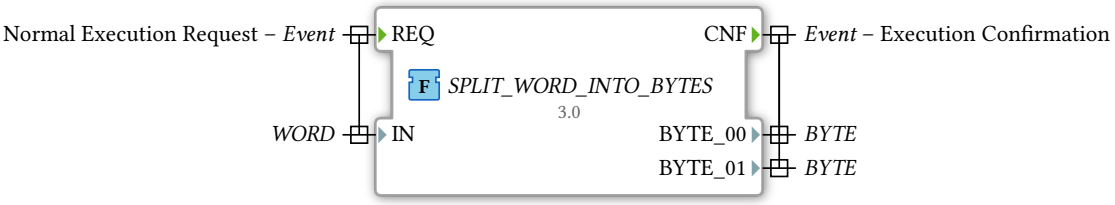
⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Bit-Manipulationsfunktionen bietet dieser Baustein eine spezialisierte und einfache Schnittstelle für die Zerlegung von Wörtern.
- Ähnliche Bausteine könnten eine flexible Anzahl von Bits unterstützen, was hier nicht der Fall ist.

Fazit

Der `SPLIT_WORD_INTO_BOOLS`-Funktionsblock ist ein einfacher und effizienter Baustein für die Zerlegung von 16-Bit-Wörtern in einzelne Bits. Er eignet sich besonders für Anwendungen, bei denen eine klare und direkte Bit-Extraktion benötigt wird.

SPLIT_WORD_INTO_BYTES



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `SPLIT_WORD_INTO_BYTES` dient dazu, ein 16-Bit-Wort (WORD) in seine beiden 8-Bit-Bytes aufzuteilen. Dies ist besonders nützlich, wenn mit Daten gearbeitet wird, die als WORD übertragen werden, aber byteweise verarbeitet werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ**: Startet die Ausführung des Funktionsblocks. Wird dieser Ereigniseingang ausgelöst, beginnt die Verarbeitung des Eingangswerts `IN`.

Ereignis-Ausgänge

- **CNF**: Signalisiert die erfolgreiche Beendigung der Verarbeitung. Dieses Ereignis wird zusammen mit den beiden ausgegebenen Bytes `BYTE_00` und `BYTE_01` ausgelöst.

Daten-Eingänge

- **IN**: Das 16-Bit-Wort (WORD), das in zwei Bytes aufgeteilt werden soll.

Daten-Ausgänge

- **BYTE_00**: Das niederwertige Byte (Bit 0 bis 7) des Eingangsworts `IN`.
- **BYTE_01**: Das höherwertige Byte (Bit 8 bis 15) des Eingangsworts `IN`.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock extrahiert die beiden Bytes aus dem Eingangswort `IN` und gibt sie über die Ausgänge `BYTE_00` (niederwertiges Byte) und `BYTE_01` (höherwertiges Byte) aus. Die Extraktion erfolgt durch direkte Zugriffe auf die Bytes des WORD-Datentyps (`IN.%B0` für das niederwertige Byte und `IN.%B1` für das höherwertige Byte).

Technische Besonderheiten

- Der Funktionsblock ist in der Programmiersprache ST (Structured Text) implementiert.
- Die Verarbeitung erfolgt synchron mit dem Auslösen des `REQ`-Ereignisses.
- Die Ausgabe der Bytes erfolgt zusammen mit dem `CNF`-Ereignis.

Zustandsübersicht

1. **Idle-Zustand**: Wartet auf das `REQ`-Ereignis.
2. **Verarbeitungszustand**: Bei Auslösung von `REQ` werden die Bytes extrahiert.
3. **Ausgabe-Zustand**: Die extrahierten Bytes werden mit dem `CNF`-Ereignis ausgegeben, und der Block kehrt in den Idle-Zustand zurück.

Anwendungsszenarien

- **Datenkommunikation**: Bei der Verarbeitung von Netzwerkdaten, die als WORD übertragen werden, aber byteweise interpretiert werden müssen.
- **Hardwarenahe Programmierung**: Bei der Arbeit mit Registerinhalten, die als WORD gelesen werden, aber einzelne Bytes enthalten.
- **Datenkonvertierung**: Bei der Umwandlung von WORD-Daten in zwei separate BYTE-Daten für weitere Verarbeitungsschritte.

⚖️ Vergleich mit ähnlichen Bausteinen

- Im Vergleich zu generischen Extraktionsfunktionen bietet `SPLIT_WORD_INTO_BYTES` eine spezialisierte und optimierte Lösung für die Aufteilung von WORD-Daten in Bytes.
- Andere Bausteine könnten zusätzliche Funktionen wie die Behandlung von Vorzeichen oder die Extraktion von Bits bieten, sind aber für diese spezielle Aufgabe weniger effizient.

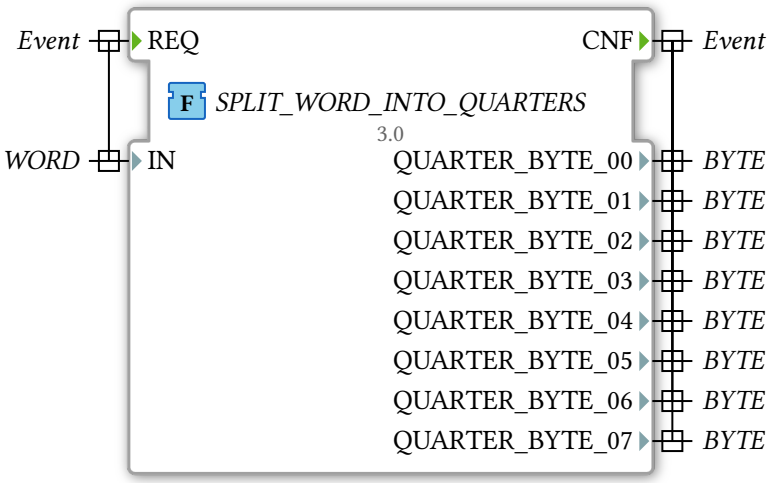
Fazit

Der `SPLIT_WORD_INTO_BYTES`-Funktionsblock ist ein einfacher, aber effektiver Baustein für die Aufteilung von 16-Bit-Wörtern in ihre einzelnen Bytes. Seine klare Schnittstelle und effiziente Implementierung machen ihn ideal für Anwendungen, bei denen diese Funktionalität benötigt wird.

SPLIT_WORD_INTO_QUARTERS

🎧 Podcast

- QUARTER



• – ◦ • – ◦ • – ◦ •

Einleitung

Der Funktionsblock `SPLIT_WORD_INTO_QUARTERS` dient dazu, ein 16-Bit-Wort in acht 2-Bit-Quarter-Bytes aufzuteilen. Dies ist besonders nützlich, wenn komprimierte Daten verarbeitet oder bestimmte Bitmuster extrahiert werden müssen.

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Startet die Verarbeitung des Eingangsworts. Wird mit dem Daten-Eingang `IN` verknüpft.

Ereignis-Ausgänge

- `CNF`: Signalisiert den Abschluss der Verarbeitung. Wird mit allen acht Quarter-Byte-Ausgängen verknüpft.

Daten-Eingänge

- `IN` (WORD): Das 16-Bit-Eingangswort, das in Quarter-Bytes aufgeteilt wird.

Daten-Ausgänge

- `QUARTER_BYTE_00` bis `QUARTER_BYTE_07` (BYTE): Die acht extrahierten 2-Bit-Quarter-Bytes. Jeder Ausgang enthält eines der Quarter-Bytes des Eingangsworts.

Adapter

Keine Adapter vorhanden.

Funktionsweise

Der Funktionsblock nutzt Bitmasken und Schiebeoperationen, um die einzelnen 2-Bit-Abschnitte aus dem Eingangswort zu extrahieren. Jedes Quarter-Byte wird durch eine bitweise UND-Verknüpfung mit einer spezifischen Maske isoliert und dann an die entsprechende Position geschoben, um es als BYTE auszugeben.

Technische Besonderheiten

- Verwendet Konstanten (`quarterconst::WORD_QUARTER_XX` und `quarterconst::SHIFT_QUARTER_XX`) für die Masken und Schiebepositionen.
- Die Extraktion erfolgt durch Kombination von UND-Verknüpfungen und Rechts-Schiebeoperationen.

Zustandsübersicht

1. **Idle**: Wartet auf das `REQ`-Ereignis.
2. **Processing**: Extrahiert die Quarter-Bytes aus dem Eingangswort.
3. **Completion**: Sendet das `CNF`-Ereignis und gibt die Quarter-Bytes aus.

Anwendungsszenarien

- Dekodierung von komprimierten Daten.
- Extraktion von Steuerbits aus einem Statuswort.
- Verarbeitung von kodierten Sensordaten.

⚖️ Vergleich mit ähnlichen Bausteinen

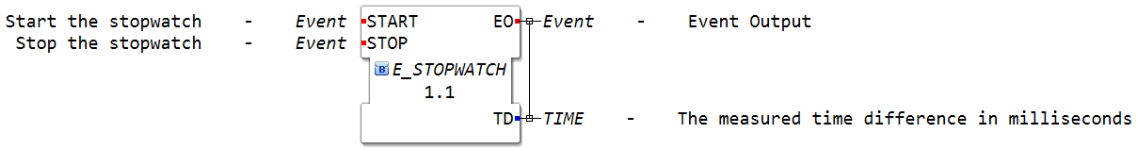
- **SPLIT_WORD_INTO_BYTES**: Teilt ein Wort in zwei Bytes auf, nicht in acht Quarter-Bytes.
- **BIT_EXTRACT**: Extrahiert einzelne Bits oder Bitgruppen, ist aber weniger spezialisiert für Quarter-Bytes.

Fazit

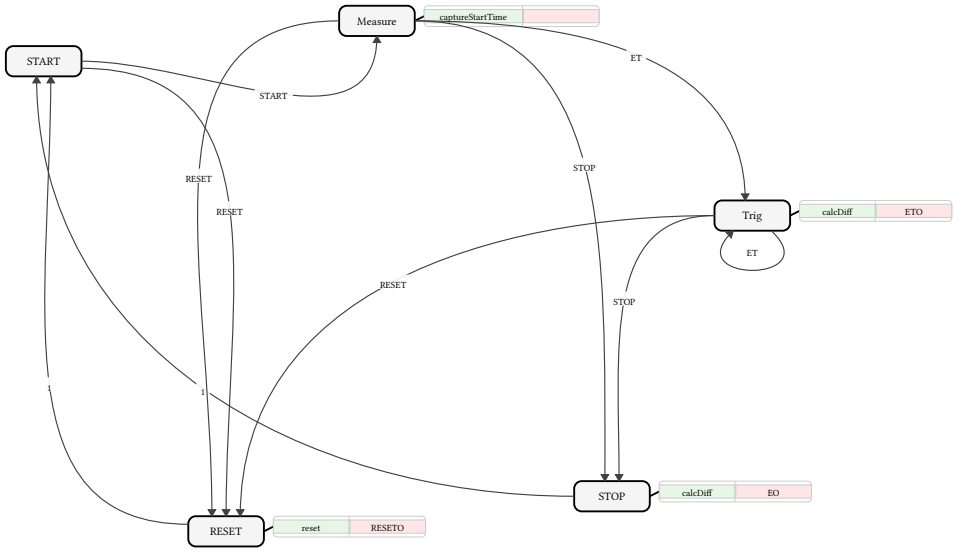
Der `SPLIT_WORD_INTO_QUARTERS`-Funktionsblock ist ein spezialisiertes Werkzeug zur effizienten Extraktion von 2-Bit-Daten aus einem 16-Bit-Wort. Seine klare Schnittstelle und effiziente Implementierung machen ihn ideal für Anwendungen, die mit komprimierten oder bitkodierten Daten arbeiten.

2.4.9.23 timing

E_STOPWATCH



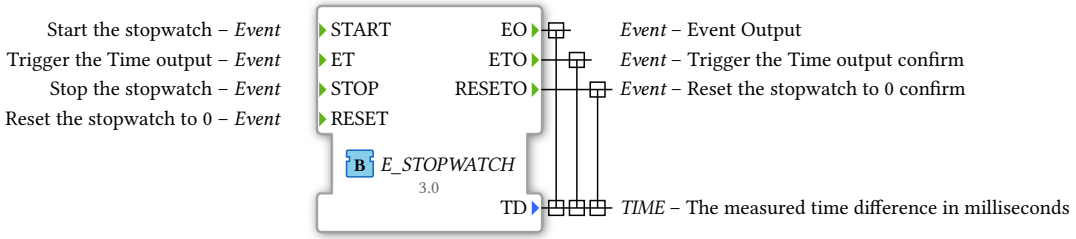
ECC: E_STOPWATCH



• – ○ • – ○ • – ○ •

Einleitung

Der **E_STOPWATCH**-Funktionsbaustein ist ein hochpräziser Zeitmessbaustein nach IEC 61499-2 Standard, entwickelt unter EPL-2.0 Lizenz. Der Baustein ermöglicht millisekundengenaue Zeitmessungen zwischen Ereignissen in Echtzeitsystemen.



Schnittstellenstruktur

Ereignis-Eingänge

- `START`: Startet die Zeitmessung
- `ET`: Trigger für Zwischenausgabe der gemessenen Zeit
- `STOP`: Beendet die Zeitmessung und gibt Ergebnis aus
- `RESET`: Setzt die Stoppuhr zurück

Ereignis-Ausgänge

- `EO`: Zeitdifferenzausgabe nach STOP
- `ETO`: Zeitdifferenzausgabe nach ET-Trigger
- `RESETO`: Bestätigung des Reset-Vorgangs

Daten-Ausgänge

Parameter	Typ	Beschreibung	Genauigkeit
`TD`	TIME	Gemessene Zeitdifferenz	1 ms

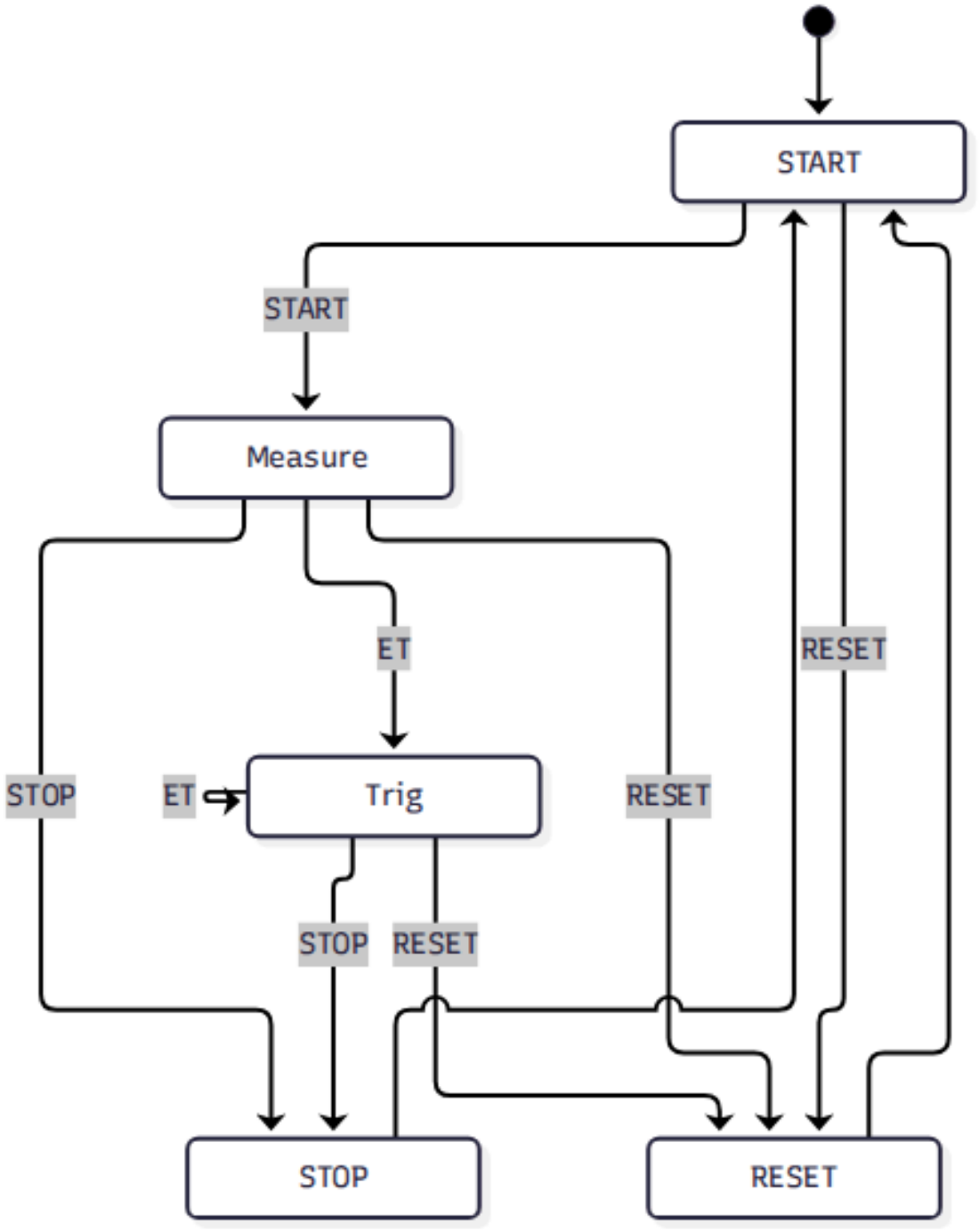
Funktionsweise

1. **Startphase**:
   - `START` initialisiert die Zeitmessung
   - Monotone Systemzeit wird erfasst (`NOW_MONOTONIC`)
2. **Messphase**:
   - `ET` liefert Zwischenergebnisse ohne Messung zu stoppen
   - Mehrfache Triggers möglich
3. **Abschlussphase**:
   - `STOP` beendet Messung und gibt Endergebnis aus
   - `RESET` setzt alle Werte zurück

Technische Besonderheiten

✓ **IEC 61499-2 konform** ✓ **Monotone Zeitmessung** (keine Systemzeitanpassungen) ✓ **Millisekundengenau** ✓ **Mehrfach triggerbar** während der Messung ✓ **Eclipse 4diac Integration** (Package: eclipse4diac::utils::timing)

Zustandsautomat (ECC)



Anwendungsszenarien

- **Leistungsmessung**: Algorithmen-Laufzeiten
- **Prozesssteuerung**: Zeitkritische Abläufe
- **Systemdiagnose**: Reaktionszeiten
- **Produktion**: Taktzeiterfassung

Beispielwerte

Operation	Ergebnis (TD)
START -> STOP nach 1,5s	T#1s500ms

START -> ET nach 750ms -> STOP nach 1,5s	T#750ms (ETO), T#1s500ms (EO)
RESET während Messung	T#0s

⚖️ Vergleich mit ähnlichen Bausteinen

Feature	E_STOPWATCH	Standard-Timer	High-Res-Timer
Genauigkeit	1ms	10ms	1µs
Trigger während Lauf	Ja	Nein	Ja
Monotone Zeit	Ja	Nein	Ja
Reset-Funktion	Ja	Teilweise	Ja

🛠️ Zugehörige Übungen

- Uebung_020i
- Uebung_020i_AX

Fazit

Der E_STOPWATCH-Baustein ist das präzise Instrument für Zeitmessungen:

- **Zuverlässig**: Monotone Zeitbasis verhindert Fehlmessungen
- **Flexibel**: Zwischenausgaben ohne Messunterbrechung
- **Einfach**: Intuitive Bedienung mit Start/Stop/Reset

Idealer Einsatz bei:

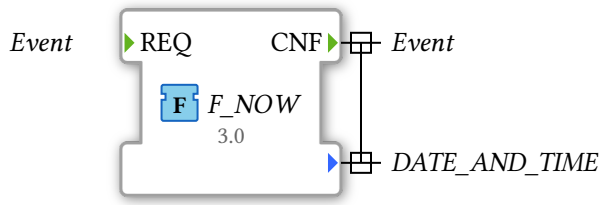
- Performance-Optimierungen
- Echtzeit-Systemen
- Prozessanalysen
- Benchmark-Tests

Die präzise Stoppuhr für industrielle Steuerungssysteme

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

F_NOW



• – ◦ • – ◦ • – ◦ •

Einleitung

Die **F_NOW**-Funktion ist eine IEC 61499-1-konforme Hilfsfunktion zur Abfrage des aktuellen lokalen Datums und der Uhrzeit, entwickelt unter EPL-2.0 Lizenz (Version 3.0). Die Funktion dient als Wrapper für die ST-Funktion NOW().

Schnittstellenstruktur

Ereignis-Eingänge

- `REQ`: Löst die Zeitabfrage aus

Ereignis-Ausgänge

- `CNF`: Bestätigung der Zeitabfrage

Daten-Ausgänge

Parameter	Typ	Beschreibung	Format
(implizit)	DATE_AND_TIME	Aktueller Zeitstempel	DT#2025-04-14-15:30:45

Funktionsweise

1. **Abfrage**:
   - `REQ` triggert die Zeitabfrage
   - Systemzeit wird in lokaler Zeitzone erfasst
2. **Ausgabe**:
   - `CNF` liefert das Ergebnis im DATE_AND_TIME-Format
   - Enthält Datum und Uhrzeit mit Sekundengenauigkeit

Technische Besonderheiten

✓ **IEC 61499-1 konform** ✓ **Lokale Zeitzonenunterstützung** (inkl. Sommer/Winterzeit) ✓ **Einfache Integration** als NOW()-Wrapper ✓ **Eclipse 4diac Integration**

Anwendungsszenarien

- **Protokollierung**: Zeitstempel für Ereignisse
- **Berichte**: Generierung mit Erstellungsdatum
- **Zeitgesteuerte Prozesse**: Aktuellzeitprüfung
- **Diagnose**: Zeitliche Fehleranalyse

Beispielwerte

Aufrufzeitpunkt	Rückgabewert
14.04.2025 15:30:45	DT#2025-04-14-15:30:45
31.12.2024 23:59:59	DT#2024-12-31-23:59:59

⚖️ Vergleich mit ähnlichen Funktionen

Feature	F_NOW	NOW	TIME()
Rückgabetyp	DATE_AND_TIME	DATE_AND_TIME	TIME
Zeitzone	Lokal	Lokal	System
Genauigkeit	Sekunden	Sekunden	Millisekunden
Sommerzeit	Ja	Ja	Nein

Einschränkungen

⚠️ **Zeitsprünge möglich** bei:

- Sommer/Winterzeit-Umstellung
- Manuellen Zeitanpassungen
- NTP-Synchronisation

Fazit

Die F_NOW-Funktion bietet einfachen Zugriff auf die Systemzeit:

- **Praktisch**: Schnelle Integration vorhandener NOW-Funktion
- **Lokalisiert**: Automatische Zeitzonenanpassung
- **Standardkonform**: IEC-kompatible Implementierung

Idealer Einsatz bei:

- Ereignisprotokollierung
- Zeitgesteuerten Prozessen
- Diagnosefunktionen
- Berichtsgenerierung

Die einfache Lösung für zeitkritische Anwendungen in 4diac

Passende Themen-Unterseiten auf ms-muc-docs.de

- Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

F_NOW_MONOTONIC

• – ◦ • – ◦ • – ◦ •

Einleitung

`F_NOW_MONOTONIC` ist eine IEC 61499-konforme Funktion, die einen monotonen Zeitstempel zurückgibt. Im Gegensatz zur Systemzeit (`NOW`) ist diese Zeit unabhängig von externen Änderungen (z. B. Zeitzonen oder NTP-Updates) und eignet sich daher ideal für zeitkritische Steuerungen oder Performance-Messungen. Die Funktion dient als Wrapper für die ST-Funktion `NOW_MONOTONIC()`.

Schnittstellenstruktur

Ereignis-Eingänge

- **REQ** (Trigger): Startet die Ausführung der Funktion. Bei Eingang dieses Ereignisses wird der aktuelle monotone Zeitwert berechnet und über den Ausgang `CNF` ausgegeben.

Ereignis-Ausgänge

- **CNF** (Bestätigung): Signalisiert den Abschluss der Operation. Das Ereignis wird zusammen mit dem berechneten Zeitwert (`TIME`) ausgegeben.

Daten-Eingänge

- *Keine* – Die Funktion benötigt keine zusätzlichen Eingabeparameter.

Daten-Ausgänge

- **TIME** (Ausgangsvariable): Rückgabewert vom Typ `TIME`, der den aktuellen Wert der monotonen Uhr repräsentiert. *Beispiel:* `T#123456ms` (abhängig vom Hardware/OS-Referenzpunkt).

Funktionsweise

1. **Initialisierung**: Die Funktion wird durch das Ereignis `REQ` aktiviert. Ressourcen werden nicht persistent gespeichert, da es sich um eine zustandslose Berechnung handelt.
2. **Makroausführung**:
   - Ruft die systemnahe Funktion `NOW_MONOTONIC()` auf.
   - Weist den Rückgabewert der Ausgangsvariable `TIME` zu.
   - Löst das Ereignis `CNF` aus.
3. **Fehlerbehandlung**:
   - Keine spezifische Fehlerbehandlung, da die Funktion keine externen Abhängigkeiten hat.
   - Bei Hardware/OS-Fehlern ist der Rückgabewert undefiniert.

Technische Besonderheiten

- **Monotone Zeit**: Garantiert stetig steigende Werte, selbst bei Systemzeitänderungen. *Hinweis:* Der Referenzpunkt ist hardwareabhängig (z. B. Systemstart).
- **Lizenz**: Eclipse Public License 2.0 (EPL-2.0).

Rückgabecodes

- **Erfolg**: Gültiger `TIME`-Wert (z. B. `T#42s`).
- **Fehler**: Keine expliziten Fehlercodes, aber undefinierte Werte bei Systemfehlern.

Anwendungsszenarien

1. **Echtzeit-Messungen**: Laufzeitanalyse von Steuerungszyklen ohne Einfluss durch Systemzeitänderungen.
2. **Synchronisation**: Zeitstempel für Logging in verteilten Systemen.
3. **Timeout-Steuerung**: Zuverlässige Timer für kritische Prozesse.

⚖️ Vergleich mit ähnlichen Bausteinen

Funktion	`F_NOW_MONOTONIC`	`NOW()`	`NOW_SYSTEM()`
Zeitquelle	Monoton (Hardware)	Systemzeit	Systemzeit (UTC)
Beeinflussbar	Nein	Ja (NTP, Zeitzone)	Ja (NTP)
Anwendungsfall	Echtzeit-Messungen	Allgemeine Zeitstempel	Zeitstempel (global)

Fazit

`F_NOW_MONOTONIC` ist ein schlanker, aber unverzichtbarer Baustein für Anwendungen, die eine zuverlässige und monotone Zeitquelle benötigen. Durch seine Hardwarenähe und Unabhängigkeit von Systemzeitänderungen eignet er sich besonders für industrielle Echtzeitanwendungen. Die Integration in Eclipse 4diac gewährleistet Kompatibilität mit modernen Automatisierungslösungen.

Hinweis: Für absolute Zeitreferenzen sollte `NOW_SYSTEM()` verwendet werden.

🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

TIMESTAMP_NS

• – ○ • – ○ • – ○ •

Einleitung

Der **TIMESTAMP_NS** ist ein IEC 61499-konformer Funktionsbaustein zur Erzeugung hochauflösender Zeitstempel in Nanosekunden. Standardmäßig generiert er Unix-Epoch-Timestamps (ab 01.01.1970), unterstützt aber durch Anpassung des Startdatums (`startDate`) alternative Zeitreferenzen. Entwickelt unter EPL-2.0 Lizenz.

Schnittstellenstruktur

Ereignis-Eingänge
- **REQ** (Trigger): Startet die Berechnung des Zeitstempels. Muss mit dem Eingangsparameter `startDate` verknüpft sein. *Beispiel:* `REQ(startDate := DT#2000-01-01-00:00:00)`

Ereignis-Ausgänge
- **CNF** (Bestätigung): Signalisiert den Abschluss der Berechnung. Der generierte Zeitstempel steht dann als `ULINT`-Wert zur Verfügung.

Daten-Eingänge
- **startDate** (DT): Startdatum für die Zeitstempelberechnung. *Default:* `DT#1970-01-01-00:00:00` (Unix-Epoch). *Hinweis:* Andere Werte ermöglichen benutzerdefinierte Referenzen (z. B. Systemstart).

Daten-Ausgänge
- **ULINT** (Ausgangsvariable): Zeitstempel in Nanosekunden seit dem definierten `startDate`. *Beispiel:* `1680000000000000000` (1,68e18 ns ≈ 2023-03-28).

Funktionsweise
1. **Initialisierung**:
   - Wird durch `REQ`-Ereignis aktiviert.
   - Nutzt den aktuellen Systemzeitpunkt (`NOW()`).
2. **Berechnung**:
   - Subtrahiert `startDate` von der aktuellen Systemzeit.
   - Konvertiert die Differenz via `TIME_IN_NS_TO_ULINT()` in Nanosekunden.
   - Gibt Ergebnis als `ULINT` aus und löst `CNF` aus.
3. **Fehlerbehandlung**:
   - Keine explizite Fehlermeldung, aber Überlauf möglich bei:
     - Extrem großen Zeitdifferenzen (>584 Jahre).
     - Ungültigem `startDate` (z. B. Zukunft).

Technische Besonderheiten
- **Nanosekunden-Präzision**: 64-bit-Zähler (`ULINT`) vermeidet Überlauf bis 584 Jahre.
- **Flexible Referenzen**: Beliebige Startdaten (z. B. `DT#2000-01-01` für Y2K-Referenz).
- **Lizenz**: Eclipse Public License 2.0 (EPL-2.0).

Rückgabecodes
- **Erfolg**: Gültiger `ULINT`-Wert (Nanosekunden).
- **Fehler**: Keine expliziten Codes, aber:
  - `0` bei ungültiger Berechnung (z. B. `startDate > NOW()`).
  - Überlauf bei Differenzen >2^{64} ns.

Anwendungsszenarien
1. **High-Performance-Logging**: Synchronisation von Ereignissen in verteilten Systemen.
2. **Echtzeit-Messungen**: Laufzeitanalyse mit Nanosekunden-Präzision.
3. **Benutzerdefinierte Zeiträume**: Berechnung von Intervallen ab beliebigen Referenzpunkten.

⚖️ Vergleich mit ähnlichen Bausteinen

Funktion	`TIMESTAMP_NS`	`NOW()`	`F_NOW_MONOTONIC`
Ausgabeformat	Nanosekunden (ULINT)	Zeitintervall (TIME)	Zeitintervall (TIME)

Referenzpunkt	Konfigurierbar	Systemstart	Hardwareabhängig
Präzision	1 ns	1 ms	1 ns (typisch)
Anwendungsfall	Absolute Zeitstempel	Relative Zeitmessung	Monotone Messung

Fazit

`TIMESTAMP_NS` ist ein essentieller Baustein für Anwendungen, die hochpräzise und flexible Zeitstempel benötigen. Seine Unterstützung benutzerdefinierter Startdaten und Nanosekunden-Präzision macht ihn ideal für:

- **Industrielle Datenaufzeichnung**
- **Echtzeit-Steuerungen**
- **Wissenschaftliche Messsysteme**

Hinweis: Für monotone Zeitmessungen sollte `F_NOW_MONOTONIC` bevorzugt werden.

2.5 ⚖️ Vergleich mit anderen Normen

In diesem Bereich werden Konzepte und Bausteine anderer Normen, insbesondere der **IEC 61131-3**, dargestellt, um Unterschiede und Gemeinsamkeiten zur IEC 61499 zu verdeutlichen.

2.5.1 IEC 61131-3 (Alte Norm)

Dieser Abschnitt beschreibt Funktionsbausteine gemäß der „klassischen“ SPS-Norm **IEC 61131-3**. Diese Bausteine dienen als Referenz und Vergleichsbasis zu den IEC 61499-Konzepten.

!!! warning „Wichtiger Hinweis“ Diese Bausteine entsprechen der **IEC 61131-3**. In der IEC 61499 können sich Verhaltensweisen, insbesondere bezüglich der Ausführung und Ereignisse, unterscheiden.

2.5.1.1 Zeitgeber (Timer)

2.5.1.2 Bistabile Elemente (Flip-Flops)

2.5.1.3 RS (Bistabil, vorrangig rücksetzen) - IEC 61131-3

Der Funktionsbaustein **RS** ist ein bistabiles Element (Flip-Flop), bei dem das Rücksetzen Vorrang vor dem Setzen hat.

Schnittstelle

Typ	Name	Datentyp	Beschreibung
Eingang	S	BOOL	Setz-Eingang (Set)
Eingang	R1	BOOL	Rücksetz-Eingang (Reset)
Ausgang	Q1	BOOL	Ausgangszustand

Funktion

Das Verhalten entspricht der **Tabelle 43** (Bistabile Standardfunktionsbausteine) der Norm DIN EN 61131-3.

Die Logik wird durch folgende Wahrheitstabelle beschrieben:

S	R1	Q1 (aktuell)	Beschreibung
0	0	Q1 (alt)	Zustand halten
0	1	0	Rücksetzen
1	0	1	Setzen
1	1	0	**Vorrangig Rücksetzen**

Anmerkung: Der Anfangszustand der Ausgangsvariable `Q1` ist standardmäßig `0` (FALSE).

Logische Gleichung

Gemäß dem Funktionsbaustein-Rumpf in der Norm (Tabelle 43, 2a):
`Q1 := NOT R1 AND (S OR Q1);`
Wenn `R1` TRUE ist, wird der Ausdruck `NOT R1` zu FALSE, und somit wird `Q1` zwangsläufig `FALSE` (0), unabhängig vom Zustand von `S`. Dies realisiert die Rücksetz-Dominanz.

Siehe auch

- E_RS (IEC 61499): Das ereignisgesteuerte Äquivalent in der IEC 61499. Beachten Sie, dass dort keine garantierte „Dominanz“ bei gleichzeitigen Ereignissen existiert.

2.5.1.4 SR (Bistabil, vorrangig setzen) - IEC 61131-3

Der Funktionsbaustein **SR** ist ein bistabiles Element (Flip-Flop), bei dem das Setzen Vorrang vor dem Rücksetzen hat.

Schnittstelle

Typ	Name	Datentyp	Beschreibung
Eingang	S1	BOOL	Setz-Eingang (Set)
Eingang	R	BOOL	Rücksetz-Eingang (Reset)
Ausgang	Q1	BOOL	Ausgangszustand

Funktion

Das Verhalten entspricht der **Tabelle 43** (Bistabile Standardfunktionsbausteine) der Norm DIN EN 61131-3.

Die Logik wird durch folgende Wahrheitstabelle beschrieben:

S1	R	Q1 (aktuell)	Beschreibung
0	0	Q1 (alt)	Zustand halten
0	1	0	Rücksetzen
1	0	1	Setzen
1	1	1	**Vorrangig Setzen**

Anmerkung: Der Anfangszustand der Ausgangsvariable `Q1` ist standardmäßig `0` (FALSE).

Logische Gleichung

In der grafischen Darstellung (FBS) wird dies oft als ODER-Gatter am Setz-Eingang und einem Speicherglied dargestellt, aber logisch entspricht der Zustand:
`Q1 := S1 OR (Q1 AND NOT R);`
(Unter Berücksichtigung der Dominanz von S1 bei gleichzeitiger Aktivierung) Genauer laut Norm-Rumpf (Tabelle 43, 1a):
`Q1 := S1 OR (NOT R AND Q1);`
Wenn `S1` TRUE ist, ist `Q1` TRUE, unabhängig von `R`.

Siehe auch

- E_SR (IEC 61499): Das ereignisgesteuerte Äquivalent in der IEC 61499. Beachten Sie, dass dort keine garantierte „Dominanz“ bei gleichzeitigen Ereignissen existiert.

2.5.1.5 TOF (Ausschaltverzögerung) - IEC 61131-3

Der Funktionsbaustein **TOF** (Timer Off-Delay) implementiert eine Ausschaltverzögerung. Der Ausgang bleibt nach Wegfall des Eingangssignals noch für eine definierte Zeit aktiv.

Schnittstelle

Typ	Name	Datentyp	Beschreibung

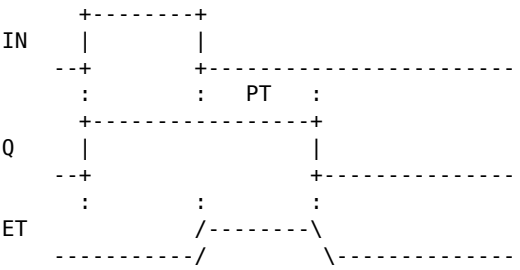
Eingang	IN	BOOL	Startsignal
Eingang	PT	TIME	Preset Time (Verzögerungszeit)
Ausgang	Q	BOOL	Ausgangssignal
Ausgang	ET	TIME	Elapsed Time (verstrichene Zeit seit fallender Flanke)

Funktion

Das Verhalten entspricht der **Tabelle 46** (Standardfunktionsbausteine Zeitgeber) und dem **Bild 15 c)** (Ausschalt-Verzögerung (TOF) Zeitverhalten) der Norm DIN EN 61131-3.

- **Aktivierung:** Wenn `IN` `TRUE` ist, wird der Ausgang `Q` sofort auf `TRUE` gesetzt und `ET` auf 0 gehalten.
- **Start Verzögerung:** Wenn `IN` von `TRUE` auf `FALSE` wechselt (fallende Flanke), startet die Zeitmessung `ET`. `Q` bleibt weiterhin `TRUE`.
- **Ablauf:** Wenn `ET` den Wert von `PT` erreicht, wird der Ausgang `Q` auf `FALSE` gesetzt.
- **Reset:** Wenn `IN` wieder `TRUE` wird, bevor die Zeit abgelaufen ist, wird `ET` wieder auf 0 gesetzt und `Q` bleibt `TRUE`.

Zeitdiagramm (Prinzip)



2.5.1.6 TON (Einschaltverzögerung) - IEC 61131-3

Der Funktionsbaustein **TON** (Timer On-Delay) implementiert eine Einschaltverzögerung. Der Ausgang wird erst aktiviert, wenn das Eingangssignal für eine definierte Zeit anliegt.

Schnittstelle

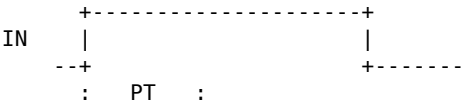
Typ	Name	Datentyp	Beschreibung
Eingang	IN	BOOL	Startsignal
Eingang	PT	TIME	Preset Time (Verzögerungszeit)
Ausgang	Q	BOOL	Ausgangssignal
Ausgang	ET	TIME	Elapsed Time (verstrichene Zeit)

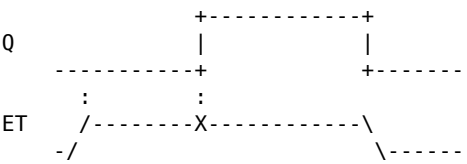
Funktion

Das Verhalten entspricht der **Tabelle 46** (Standardfunktionsbausteine Zeitgeber) und dem **Bild 15 b)** (Einschalt-Verzögerung (TON) Zeitverhalten) der Norm DIN EN 61131-3.

- **Start:** Wenn `IN` `TRUE` ist, beginnt die Zeitmessung `ET`.
- **Ablauf:** Wenn `ET` den Wert von `PT` erreicht hat und `IN` immer noch `TRUE` ist, wird der Ausgang `Q` auf `TRUE` gesetzt.
- **Halten:** `Q` bleibt `TRUE`, solange `IN` `TRUE` ist.
- **Abbruch:** Wenn `IN` auf `FALSE` wechselt (bevor oder nachdem `PT` abgelaufen ist), wird `ET` auf 0 zurückgesetzt und `Q` wird `FALSE`.

Zeitdiagramm (Prinzip)





2.5.1.7 TP (Puls) - IEC 61131-3

Der Funktionsbaustein **TP** (Timer Pulse) implementiert einen Impulsgeber (Monoflop). Er erzeugt einen Ausgangsimpuls definierter Dauer.

Schnittstelle

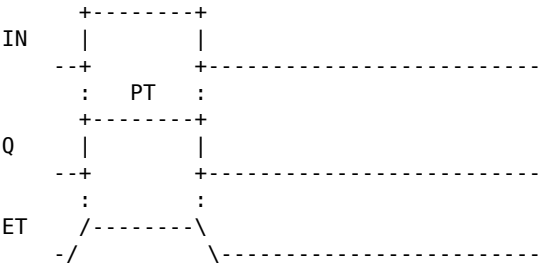
Typ	Name	Datentyp	Beschreibung
Eingang	IN	BOOL	Startsignal (steigende Flanke löst Impuls aus)
Eingang	PT	TIME	Preset Time (Impulsdauer)
Ausgang	Q	BOOL	Ausgangssignal
Ausgang	ET	TIME	Elapsed Time (verstrichene Zeit seit Impulsstart)

Funktion

Das Verhalten entspricht der **Tabelle 46** (Standardfunktionsbausteine Zeitgeber) und dem **Bild 15 a)** (Puls (TP) Zeitverhalten) der Norm DIN EN 61131-3.

- **Start:** Wenn `IN` von `FALSE` auf `TRUE` wechselt (steigende Flanke), wird der Ausgang `Q` auf `TRUE` gesetzt und die Zeitmessung `ET` startet.
- **Dauer:** Der Ausgang `Q` bleibt für die Zeitdauer `PT` auf `TRUE`, unabhängig davon, ob `IN` währenddessen wieder auf `FALSE` wechselt.
- **Rücksetzen:** Nachdem die Zeit `PT` abgelaufen ist (d.h. `ET >= PT`), fällt `Q` auf `FALSE` zurück.
- **Re-Trigger:** Der Impuls ist **nicht** nachtriggerbar. Flanken am Eingang `IN` während der Impulsausgabe (solange `ET < PT` und `Q = TRUE`) werden ignoriert.
- **Initialisierung:** `ET` wird zurückgesetzt, wenn `IN FALSE` ist und die Zeit abgelaufen ist oder noch nicht gestartet wurde.

Zeitdiagramm (Prinzip)



3. static

3.1 Fonts

3.1.1 TGL Normschrift – _std Font Variants

3.1.1.1 Overview

The `_std` variants of the TGL Normschrift fonts fix the **U+0023 (#)** glyph, which in the original fonts incorrectly contains a „m/m“ (millimeters) symbol instead of the standard number sign / hash character.

This makes the fonts usable in contexts where a proper # is required (e.g. programming, technical documentation, IEC 61499 function block diagrams).

3.1.1.2 Fonts

File	Style	Based on
`TGL 0-16_std.ttf`	Italic (75° slant)	TGL 0-16
`TGL 0-17_std.ttf`	Upright	TGL 0-17
`TGL 0-17 alt_std.ttf`	Upright (alternate)	TGL 0-17 Alt

3.1.1.3 Changes

New glyph at U+0023 (#)

A number sign was designed from scratch to match each font's style:

- **Stroke width** matches the font (207–208 units)
- **Round stroke caps** using the same quadratic Bézier curve pattern as the original glyphs (I, l, hyphen)
- **Height** is approximately 2/3 of the capital letter height, consistent with typical number sign proportions
- **TGL 0-16_std** (italic): vertical strokes are slanted at 75° from the baseline, matching the font's standard lean angle per DIN/TGL specification for slanted Normschrift
- **TGL 0-17_std / TGL 0-17 alt_std** (upright): all strokes are perpendicular to each other
- The glyph is constructed as a single non-overlapping outer contour with one inner counter (hole), avoiding rendering artifacts from overlapping paths

Original m/m glyph preserved at U+E000

The original „m/m“ symbol that previously occupied the # position has been moved to **U+E000** (Unicode Private Use Area) so it remains accessible if needed.

Internal font names

All internal name table entries (family name, full name, PostScript name, unique ID) have `_std` appended to avoid conflicts when both original and modified fonts are installed on the same system.

3.1.1.4 Technical Details

- Format: TrueType (.ttf)
- Units per em: 2048
- Modified using Python `fonttools` library
- No other glyphs were changed
- All original font metrics, hinting, and licensing information are preserved

3.1.1.5 License

The original TGL Normschrift fonts are Copyright (c) 2010 by Peter Wiegel, released under GPL with font exception and OFL (Open Font License). The `_std` modifications are distributed under the same terms.